

BW Transform and its applications

Lecture #3: Stefano Cataudella, Antonio Gulli

3.1 Introduction and Overview

In these notes we introduce the Burrows and Wheeler Transform (BWT) and its applications, resulting in an innovative text compression algorithm belonging to the so-called “block sorting” category. The BWT is a peculiar permutation of input text which is usually followed by a Move-To-Front transform (MTF) and an optional Run-Length-Encoding (RLE) module. The output produced is then processed by a statistical compressor, such as Huffman or Arithmetic Coder. Note that, from a certain point of view, BWT and MTF could be regarded as a performance booster for statistical compressors. We describe such schema in figure 3.1.

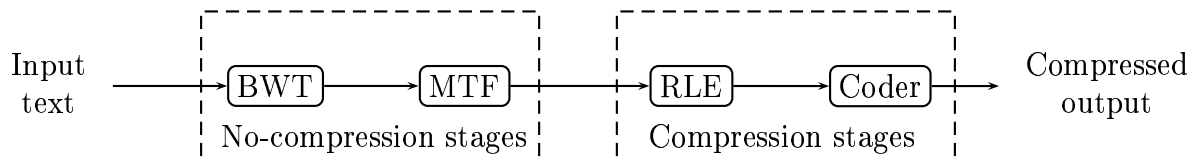


Figure 3.1: BW stages

The technique offers excellent compression performance, as well as good speed. Compression is generally considerably better than the one achieved by more conventional dictionary-based compressors (LZ77 — LZ78), and competitive with all but the best compressors of the PPM family.

Notes are organized as follows. In section 3.2 we review the basics of statistical compression and some theoretical compression bounds. Later, we will use these results to compare them with the better ones achieved by the BWT compression algorithm. In section 3.3 we show how to construct and reverse the BWT on a generic text string. Then, in section 3.4 we discuss the original compression algorithms proposed in [11], and in section 3.5 we point out some theoretical bounds for BWT. Section 3.6 presents several variants introduced into the various steps of the basic algorithm. Then, in section 3.7 we describe some BWT-based compressors. Experimental results are given in section 3.8. Finally, in section 3.9 we conclude our notes and present some innovative works developed using BWT ideas.

3.2 Statistical compression algorithms

The most intuitive method to compress an input string is to represent more frequent symbols with shorter codes. This idea is the basis of statistical compression algorithms. Naturally, one can extend this approach to code also groups of symbols, instead of a single one. The frequency of a (group of) symbol(s) can be calculated by simply counting the occurrences of the symbol in the input. A more sophisticated way is to consider conditional probabilities or frequencies, using the fact that the already seen symbols are generally a good predictor for the next one.

Given an input source S on an alphabet $\mathcal{A} = \{\alpha_1, \dots, \alpha_k\}$, it is well known that each symbol α_i with probability p_i needs at least $H_0(S) = -\sum_{i=1}^k p_i \log p_i$ bits to be represented. The H_0 function is commonly named order-0 *statistical entropy*, because the probabilities p_i do not depend on previous seen symbols [24]. When we need to compress a particular N -length input string s^N (exponentiation is used to denote string length), usually we don't know the probabilities p_i . It is then necessary to use symbols' frequencies n_i/N (where n_i denotes the number of occurrences for the symbol α_i) to define the order-0 *empirical entropy* $H_0(s^N) = -\sum_{i=1}^k \frac{n_i}{N} \log \frac{n_i}{N}$. Both statistical and empirical entropy can be order- k defined, by considering probabilities or frequencies dependent on the k already seen symbols.

In the following of this section, we shortly review Huffman and Arithmetic coding, two classical order-0 statistical compression algorithms. Both of them show a similar theoretical bound on the size of their output $|\text{Order0}(s^N)|$:

$$|s^N|H_0(s^N) = NH_0(s^N) \leq |\text{Order0}(s^N)| \leq |s^N|H_0(s^N) + \mu|s^N| = NH_0(s^N) + \mu N \quad (3.1)$$

where $\mu = 1$ for Huffman coding, and $\mu \cong \frac{1}{100}$ for Arithmetic coding.

These two algorithms can be used, alternatively, as a “black box” to implement the final step of Burrows and Wheeler compression algorithm. As we will see, the Arithmetic Coder can achieve better compression ratio than the Huffman Coder, but it was patented for a long time, and so several commercial and public domain BW based programs still use the Huffman coder. Furthermore, the Arithmetic Coder is almost always slower than the Huffman one.

3.2.1 Huffman Coding

This algorithm assigns a codeword to each symbol, so that the most frequent symbols receive a shorter codeword. Huffman coder follows an iterative approach. At the beginning, input symbols are seen as the leaves of a binary tree T . They are weighted by their frequencies, and they are inserted in a priority queue Q . At each step, a new node is created in T . It takes as children the two nodes with the lowest weights in Q , and as weight the sum of its children's weights. This new node is then inserted in Q , and its two children are dequeued. The two edges just created are labeled with '0' and '1'. The algorithm ends when Q is empty.

Each symbol α_i is then represented by a codeword which is the concatenation of the binary labels in the path from the root of T to α_i .

There exist several variants of Huffman coding, among the others:

- Dynamic Huffman, in which frequencies are calculated dynamically, during the reading of the input (the original algorithm read the input twice: the first time to calculate frequencies, and the second one to compress the input). In equation 3.1, dynamic Huffman has $\mu = 2$;
- Block-based Huffman, in which groups of symbols are coded instead of single ones. In equation 3.1, block-based Huffman takes $\mu = \frac{1}{k}$ where k is the size of the block;
- Huffword, in which words instead of symbols are coded.
- Multi-Huffman, which we will review in section 3.6.4.2

3.2.2 Arithmetic Coding

Arithmetic Coding assumes that the alphabet order is fixed and shared between both compressor and decompressor programs. The compressor partitions the real interval $I = [0, 1)$ in $|\mathcal{A}|$ subintervals (where \mathcal{A} is the input alphabet), according to the frequencies of the input symbols. The input string s is then processed, one symbol s_i at time, by taking the segment corresponding to s_i as new interval I and partitioning it again according to the symbols' frequencies. At the end, every real number in the final interval is a good codeword for the whole string s .

The main difference between Huffman and Arithmetic Coding is that the first algorithm associates a codeword to each alphabet symbol, while the second one uses just a single codeword to represent the whole input string.

In an ideal environment with infinite mathematical precision, we would have in equation 3.1 $\mu = \frac{1}{N}$. Working with finite precision yields $\mu \cong \frac{1}{100}$.

Like for Huffman, there exists a dynamic version of Arithmetic Coding where frequencies are calculated dynamically during the input reading.

3.3 The Burrows and Wheeler Transform

Burrows-Wheeler transform is based on two inverse operations. A forward transform which rearranges the input text into a form which exhibits “local” symbols regularity, and a reversal transform which reconstructs the original input from the permuted data. BWT is also known as a *block-sort* transform since the input is divided into blocks before the transform

is executed, and since the transform uses a sort operation on the permuted data. Each block can be as large as the entire file. A larger block size improves the compression ratio achieved, but requires a much larger amount of memory especially for compression (but also for decompression).

Note that the forward transform gives an output block which contains exactly the same data symbols of the input block, differing only in their ordering. So, BW transform does not compress anything. It just prepares data for a successive compression achieved with other well known techniques (such as order-0 statistical compressors).

3.3.1 Forward Transform

We can think the forward transform as a three step algorithm called hereafter *BW*:

1. Let $b = b_1b_2 \dots b_u$ be the input block over some finite alphabet Σ , extracted as a substring from an input text t . Consider " $b\$$ " the input block modified by appending a special symbol $\$$ which is lexicographically smaller than any symbol in Σ . Let $\beta = |b\$| = u + 1$. Build \mathcal{M} , a (conceptual) matrix of size $\beta \times \beta$ containing all the cyclic left shifts of " $b\$$ ", one for each row.
2. Sort the rows of \mathcal{M} according to the ascending lexicographical order, considering each row from right to left.
3. Let F and L be the first and the last columns of the sorted \mathcal{M} . Let \hat{F} be the F column without the $\$$ symbol and r the number of the row which starts with $\$$ (counting from 0). Take as output $bw(b) = (\hat{F}, r)$.

The above steps define the transform $bw(b)$, given the initial block b . Let's see an example by taking $b =$ "mississippi" in figure 3.2. The output is $bw(b) = (msspipissii, 2)$.

Note that each symbol in F is sorted using as a key the *context* given by *all* the symbols preceding it in the cyclic permutations of b . As a consequence, symbols which follow identical contexts in b are forced to be adjacent in F . This property holds for contexts of *any* length. In fact, let w be a k -length substring of the input s and let w_s be the set of all symbols which follow w in s . Taking $s =$ mississippi and $k = 1$ we have $m_s = i$, $i_s = ssp$, $s_s = ssii$, $p_s = pi$. Taking $k = 2$ we have $mi_s = s$, $is_s = ss$, $ss_s = ii$, $si_s = sp$, $ip_s = p$, $pp_s = i$, $pi_s = \emptyset$. In figure 3.3, we can see how the w_s 's are grouped together in $bw(b)$ for $k = 1$ and $k = 2$ (actually this holds for *any* k).

Working with a sufficiently large context w , only few distinct symbols are likely to be seen in w_s . The string F will result *locally homogeneous* since it will be the concatenation of several substrings containing only a few distinct symbols. This similarity in nearby contexts can be later exploited for achieving good compression using other well-known techniques.

$k = 1$			$k = 2$		
w_s		w	w_s		w
m	ississippi	\$	m	ississippi	i\$
s	sissippi\$m	i	s	sissippi\$	mi
\$	mississippi	i	\$	mississippi	pi
s	sippi\$miss	i	s	sippi\$mis	si
p	pi\$mississ	i	p	pi\$mississ	si
i	ssissippi\$	m	i	ssissippi	\$m
p	i\$mississi	p	p	i\$mississ	ip
i	\$mississip	p	i	\$mississi	pp
s	issippi\$mi	s	s	issippi\$m	is
s	ippi\$missi	s	s	ippi\$miss	is
i	ssippi\$mis	s	i	ssippi\$mi	ss
i	ppi\$missis	s	i	ppi\$missi	ss

Figure 3.3: The w_s in “mississippi” for $k = 1$ and $k = 2$

Now we have all we need to reverse the forward transform $bw(b) = (\hat{F}, r)$. Given F we obtain L , using property 1. Since \$ is smaller than any other symbol, we know that $F[0] = m$ is the first symbol in the original block. By property 3, we derive that $F[0]$ corresponds to $L[5]$ (the first m in column L). By property 2, we get that the second symbol of b is $F[5] = i$. Using again property 3 we obtain that $F[5]$ (first i in column F) correspond to $L[1]$ (first i in column L). Again, by property 2 we get that the third symbol of original block b is $F[1] = s$. We can proceed in this way until we meet the symbol \$. Usually, this process is called FL-Mapping.

From the algorithmic point of view, the first occurrence of each symbol in L can be found efficiently by using a vector Occ of $|\Sigma|$ elements, such that $Occ[i] = \sum_{j < i} occ(\sigma_j, b\$)$ where $occ(\sigma_j, b\$)$ denotes the number of occurrences of σ_j in the input block $b\$$. Besides, we should construct another vector FL , such that $FL[i] = Occ[F[i]] + Rank[i]$ where $Rank$ is a precomputed vector defined by $Rank[i] = occ(F[i], F[0 \dots i - 1])$. In other words, $Rank[i]$ contains the relative rank of the symbol denoted by $F[i]$.

Using the FL auxiliary vector we can easily describe the reverse transformation algorithm as follows (we recall that r is the index of the row containing the original input block in the conceptual sorted matrix \mathcal{M} , and u is the length of the input block). In the figure we have also included the above vectors for our example string “mississippi”.

<i>sorted M</i>					<i>M'</i>			
<i>i</i>	<i>F</i>		<i>L</i>			<i>L</i>	<i>F</i>	
0	m	ississippi	\$		0	ississippi	\$	m
1	s	sissippi\$m	i		1	sissippi\$m	i	s
2	\$	mississipp	i		2	mississipp	i	\$
3	s	sippi\$miss	i		3	sippi\$miss	i	s
4	p	pi\$mississ	i		4	pi\$mississ	i	p
5	i	ssissippi\$	m	<i>left rotation</i> →	5	ssissippi\$	m	i
6	p	i\$mississi	p		6	i\$mississi	p	p
7	i	\$mississip	p		7	\$mississip	p	i
8	s	issippi\$mi	s		8	issippi\$mi	s	s
9	s	ippi\$missi	s		9	ippi\$missi	s	s
10	i	ssippi\$mis	s		10	ssippi\$mis	s	i
11	i	ppi\$missis	s		11	ppi\$missis	s	i

Figure 3.4: FL-Mapping: symbols maintain the same relative order

1. $i = 0; j = r;$	\$ i m p s
2. while $i \leq u$ do	A 0 1 5 6 8
3. $j = FL[j];$	
4. $b[i] = F[j];$	
5. $i = i + 1;$	V 0 0 0 1 0 0 1 1 2 3 2 3
6. endw	FL 5 8 0 9 6 1 7 2 10 11 3 4

3.4 $BW0$ and $BW0_{RLE}$ Compression Algorithms

In this section we review the original compression algorithm proposed by Burrows and Wheeler in [11] which uses the BW transform described in the previous section.

Actually, in the original formulation given in [11], rows are sorted in left-to-right lexicographical order and, using the same notation given in section 3.3, it is defined $bw'(b) = (\hat{L}, r)$. Moreover, one can define a LF-Mapping to reverse this transform. This means that each symbol in the BW transform output is sorted using as a key the context given by all the symbols *following* it in the cyclic permutations of the input. Note that, defining $rev(b)$ as the reverse string of b , we have:

$$bw(rev(b)) = bw'(b)$$

The original transform is easier to implement (because a left-to-right sort routine is usually available in all software development environments), but the transform here defined is more

intuitive, in the sense that it uses the already seen symbols as a memory. Moreover, these two transforms present the same asymptotic compression performance since their output's entropy is the same. In the following, then, we will adopt the transform defined in section 3.3.

The output of the BW transform presents a local homogeneity, as pointed out at the end of section 3.3.1. This local homogeneity can be converted in a global one using a Move-To-Front transform, *MTF*. Burrows and Wheeler proposed two different algorithms. The first one, named *BW0*, takes the output given by $MTF(BWT(input))$ and compresses it by an order-0 statistical compression algorithm, such as Huffman or Arithmetic coder. The second one pre-processes $MTF(BWT(input))$ using Run Length Encoding (*RLE*) to compress long runs of 0's symbols. By processing $RLE(MTF(BWT(input)))$ with an order-0 statistical compressor, we have the $BW0_{RLE}$ Algorithm.

3.4.1 Move-To-Front Transform

Given an input string t on a Σ alphabet, MTF transforms the string t in an output string on the alphabet $\Sigma^{mtf} = [0, \dots, |\Sigma| - 1]$. The algorithm uses a symbols' list l initialized to Σ . Each symbol σ in t is processed in turn outputting its position in l , and the list is modified moving σ from its current position to the front. The reverse operation MTF^{-1} can be obtained having the initial list l and performing the same steps of the forward MTF.

$MTF(BW(input))$				MTF^{-1}		
Symbol	List	Code		Code	Symbol	List
m	imps	1		1	imps	m
s	mips	3		3	mips	s
s	smip	0		0	smip	s
p	smip	3		3	smip	p
i	psmi	3	<i>reverse</i> ↦	3	psmi	i
p	ipsm	1		1	ipsm	p
i	pism	1		1	pism	i
s	ipsm	2		2	ipsm	s
s	sipm	0		0	sipm	s
i	sipm	1		1	sipm	i
i	ispm	0		0	ispm	i

Figure 3.5: MTF transform

As an example, we take the string $t = msspipissii$ defined on $\Sigma = i, m, p, s$. The output string will be defined on $\Sigma^{mtf} = [0, \dots, 3]$, and is obtained as in figure 3.5.

Generally, MTF assigns codes with lower values to more frequent symbols. This becomes more and more evident as the length of string t or the size of Σ increase. Picture 3.6, taken from [19], points out the frequencies of the first 128 MTF-ranks for three files of the standard Calgary Corpus [12]. In [20], it is shown that typically about 60% are 0, and, for many files, the value n has a frequency almost proportional to $1/n^2$.

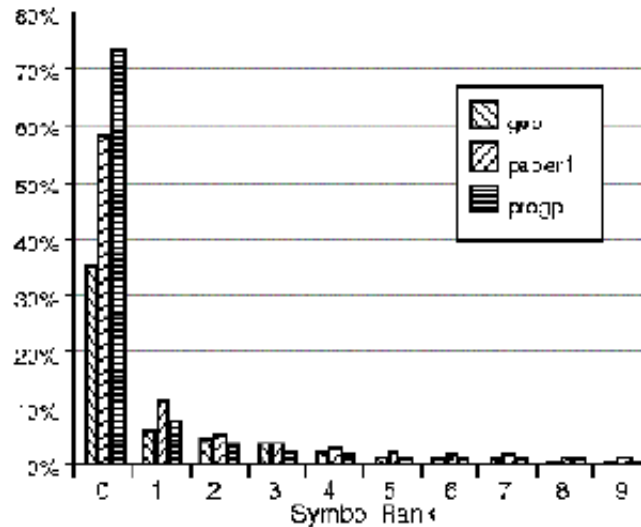


Figure 3.6: Distribution of symbols in the output of MTF.

3.4.2 Run-Length Encoding

MTF output is characterized by long runs of consecutive low ranks. This is a direct consequence of the local homogeneity in the output of BWT . Then, it is quite intuitive to compress each run with a code which expresses just the symbol and the length of the run. Several alternatives have been proposed. In [11] Burrows and Wheeler suggest to encode just the runs of zeroes with their length. In [41] Nelson proposes to replace the runs of a symbol σ of length n with a sequence " $\sigma \sigma$ " followed by an integer representing $n - 2$.

A similar approach consists in encoding the runs 0^m using two special symbols $\mathbf{0}$, $\mathbf{1}$ not belonging to Σ^{mtf} , to represent the γ -Elias codes $\gamma(m)$, the δ -Elias codes $\delta(m)$ or the Wheeler codes $W(m)$. These codes show the benefit of coding positive integers with a variable length, and so they don't need to waste unnecessary space. Naturally, the codes must guarantee the absence of ambiguity in their output.

In the gamma code, a positive integer x is firstly converted in binary, $bin(x)$, and then a

sequence of zeroes of length $|bin(x)| - 1$ is premitted to $bin(x)$. Thus the decimal number 100 is represented by 0000001100100, since $bin(100) = 1100100$, and $|1100100| - 1 = 6$. Note that, while usually an integer is represented using 16 or 32 bits, in this case we are using only 13 bits. In general, the length of the γ -Elias code of an integer x is $|\gamma(x)| = 2\lfloor \log_2 x \rfloor + 1$.

In the delta code, a positive integer x is firstly converted in binary, and then the length of $bin(x)$ is represented using a gamma code, and premitted to $bin(x)$. As instance, the decimal number 100 is represented by 001111100100, since $bin(100) = 1100100$, $|bin(100)| = 7$, and $\gamma(7) = 00111$. Thus, in this case we have saved another bit. In general, the length of the δ -Elias code of x tends to $|\delta(x)| = \log_2 x + \log_2 \log_2 x + 1$.

In the Wheeler code, a positive integer x is simply represented using $bin(x+1)$ and discarding the most significant bit. Thus the decimal number 100 is coded by 100100, using only 6 bits. The length of this code is $|W(x)| = \lfloor \log_2(x+1) \rfloor$. Note that, while the above representations are unambiguous, this one cannot be used to code a sequence of integers (for example, the code 0101 could represent the decimal number 22 or the number 4, followed by another 4). However, when we have a run 0^m , we need to encode just the number m and nothing else (if two runs are consecutive, they are really a single run!). The advantage of $W(x)$ is that $|W(x)| \leq m$; so it always compress, while the others approaches don't.

3.5 Theoretical Results

Recently, BWT algorithms have been investigated from a theoretical point of view and several results have been obtained, linking their compression performance to the input's entropy. A first class of theoretical bounds, usually enough tight, is obtained under very specific hypothesis on the input source. A second class of less tight bounds, does not depend on the source type and, then, can be widely used. Nevertheless, several theoretical issues still remain open to investigation.

3.5.1 Source-Dependent Performance Bounds

Some results were given in [1] under the assumption that the source is a stationary ergodic k -finite order Markov process. Roughly speaking, this means:

- **Stationary:** The symbols distribution doesn't depend on time;
- **Finite Order:** Each symbol probability depends on the k preceding symbols, at most;
- **Ergodic:** As the length of a generic sequence S increases, any subsequence shows a frequency which approaches a definite limit independent of the particular S .

Shortly, the ergodic property means statistical homogeneity. Arimura and Yamamoto showed that a BW algorithm *variant* is optimal in the sense that “the expected codeword length asymptotically converges to the entropy of the Markov source”.

In particular, they use a δ -Elias code to represent integers in the MTF output, instead of a statistical order-0 compressor.

Theorem 3.1 *When the output of the BW transform y^N (where N is the length of y) is encoded by MTF, and a universal code of positive integers bounded by a positive, concave and monotonically increasing function f , then the $L(y^N)$ codeword length per symbol is bounded by:*

$$L(y^N) \leq \sum_{w \in \mathcal{A}^k} \sum_{a \in \mathcal{A}(w_y)} \frac{N(a, c)}{N} f\left(\frac{N(c) + A}{N(a, c)}\right)$$

where $N(c)$ and $N(a, c)$ denote the frequencies of c and ca in the input text x^N , while $\mathcal{A}(w_y)$ is the alphabet \mathcal{A} restricted to the given context w_y .

□

The intuition behind this theorem is that BWT output is grouped by contexts. Given a context c and a symbol a , integer codes $n_1, \dots, n_{N(c)}$ produced by MTF can be bounded as:

$$n_j \leq \begin{cases} A & \text{for } j = 1 \\ t_j - t_{j-1} & \text{for } 2 \leq j \leq N(a, c) \end{cases}$$

where $t_j - t_{j-1}$ represents the number of symbols (even if not distinct) between the $(j-1)^{th}$ and the j^{th} occurrence of a symbol in y^N .

Theorem 3.2 *Let $EL_{BS}(X^N)$ be the expected codeword length of x^N for the stationary ergodic finite order Markov source with entropy $H(X) = \lim_{l \rightarrow \infty} \frac{1}{l} H(X^l)$. Then for any $\epsilon > 0$ and sufficiently large N ,*

$$EL_{BS}(X^N) \leq H(X) + O(\log H(X)) + \epsilon$$

where $O(\log s) = 2 \log(s+1) + 1$

□

The above bound holds using the δ -Elias code to represent integers in the output of MTF. The proof is given partitioning the possible input sequences in two classes: the typical ones, which show a symbols' frequency similar to the source probability distribution, and the rest, whose total probability approaches zero. Then, the bound given by theorem 3.1 is used on the first class of sequences, along with the δ -Elias length bound:

$$\delta_E(s) \leq f(s) = \log s + 2 \log(1 + \log s) + 1$$

Theorem 3.3 *Let $EL_{BS}^{(m)}(X^N)$ be the expected codeword length of the m -symbol extension of $x^N = x^{mL}$ (i. e. the symbols in x are packed in groups of m -length). Then, for any $\varepsilon > 0$ and sufficiently large m and N we have:*

$$EL_{BS}^{(m)}(X^N) \leq H(X) + \varepsilon$$

□

This theorem states the asymptotic optimality of the BW algorithm, for a stationary ergodic finite order Markov source. It is proved using theorem 3.2 on the input string, whose symbols are packed in groups of m symbols each. We point out that the logarithmic term is no more present in the bound, since it becomes $O((\log mH(x))/m)$, which tends to 0 for sufficiently large m .

Note that these results could be potentially improved by using a statistical order-0 compressor instead of the δ -Elias codes for integers. The interested reader can refer to [18].

3.5.2 Source-Independent Performance Bounds

In [38] Manzini showed two theoretical bounds for the $BW0$ and the $BW0_{RLE}$ algorithms, obtained without any assumptions about the input source.

Since BWT algorithms use potentially all previously seen input symbols, and no hypothesis on the source is made, it is natural to exploit the k -order empirical entropy of a string s :

$$H_k(s) = \frac{1}{|s|} \sum_{w \in \mathcal{A}^k} |w_s| H_0(w_s) \quad (3.2)$$

where w_s was defined in section 3.3.1.

Suppose that we use a statistical order-0 compressor to encode each block w_s of BWT output, and suppose also that we don't apply MTF. Let ρ_{bw} be the number of bits per symbol requested by the output of this compression. As we saw in equation 3.1, the number ρ of bits per symbol requested by a statistical order-0 compressor is bounded by $H_0 \leq \rho \leq H_0 + \mu$. Let's use this relation to derive an upper bound for ρ_{bw} . Since we are compressing each block w_s on its own, our result is the sum of the outputs of each single compression. Dividing by $|s|$ (the length of the input), since we want to calculate a number of bits per symbol, we have:

$$\rho_{bw} \leq \frac{1}{|s|} \sum_{w \in bw(s)} |w_s| (H_0(w_s) + \mu)$$

If we denote by $f_{w_s}(c)$ the frequency of the symbol c within the context w_s , and by $\mathcal{A}(w_s)$ the alphabet of the string w_s , we can expand the above equation obtaining the following:

$$\rho_{bw} \leq \frac{1}{|s|} \sum_{w \in bw(s)} |w_s| \left(\sum_{c \in \mathcal{A}(w_s)} f_{w_s}(c) \log_2 \frac{1}{f_{w_s}(c)} + \mu \right)$$

The frequency $f_{w_s}(c)$ can also be written as a conditioned frequency, $f(c|w)$, because when c is contained in w_s it means that c follows the context w in the input. Using this observation we obtain:

$$\begin{aligned}
\rho_{bw} &\leq \mu + \frac{1}{|s|} \sum_{w \in bw(s)} |w_s| \left(\sum_{c \in \mathcal{A}(w_s)} f(c|w) \log_2 \frac{1}{f(c|w)} \right) \\
&= \mu + \frac{1}{|s|} \sum_{w \in bw(s)} |w_s| H_0(w_s) \\
&= \mu + \frac{1}{|s|} \sum_{w \in \mathcal{A}^k} |w_s| H_0(w_s) \\
&= H_k(s) + \mu
\end{aligned} \tag{3.3}$$

Note that the bound 3.3 holds independently of the particular k chosen to calculate the entropy. To be more precise, we should take in account that selecting a particular order k for the entropy, the first k contexts w do not actually exist in the input string since they are generated only through cyclic shifts. As result we have k “noisy” symbols in the w_s , whose total cost is at most $2k \log(n+1)$ (using the γ -Elias code to represent the first k symbols of the input). Moreover, we need $\log n + 1$ further bits to represent the position of the transformed string in the shifts matrix.

The problem of applying directly a statistical compressor for each block w_s is that this approach requires the delimitation of each w_s and an extra-space to restart the statistical compressor on each w_s . Besides, when k increases, the number of different contexts also increases and the length of each w_s decreases. This implies that the statistical compressor applied on w_s will hardly reach a good performance. Moreover, when w_s decreases, it increases the dimension of the model transmitted by the compressor to the decompressor. These problems give a theoretical motivation to the use of *MTF* before the statistical compression.

Sometimes the entropy definition 3.2 provides an unrealistic lower bound. As example, consider the string $s = cc(ab)^n$. We have $a_s = b^n$, $b_s = a^{n-1}$, $c_s = ca$. This yields

$$|s|H_1(s) = nH_0(b^n) + (n-1)H_0(a^{n-1}) + 2H_0(ca) = 2$$

Hence the lower bound for s compression, $|s|H_1(s)$, would not depend on n !!

To avoid this problem, in [38] Manzini defines a *modified* order-0 empirical entropy, which takes in account the number of bits needed to represent the length of sequences w_s which have $H_0(w_s) = 0$:

$$H_0^*(s) = \begin{cases} 0 & \text{if } |s| = 0 \\ (1 + \lfloor \log |s| \rfloor) / |s| & \text{if } |s| \neq 0 \text{ and } H_0(s) = 0 \\ H_0(s) & \text{otherwise} \end{cases}$$

Unfortunately, if $H_0(s)$ is replaced by $H_0^*(s)$ in 3.2, the entropy would lose the monotonicity property (i.e. when the context length k increases, $H_k^*(s)$ would not necessarily decrease). For this reason, Manzini defines H_k^* as the maximum compression ratio achieved using for each symbol a codeword which depends on a context of size *at most* k .

The two main theorems proved by Manzini bound the compression ratio achievable by $BW0$ and $BW0_{RLE}$ algorithms in terms of the entropy $H_k(s)$ and $H_k^*(s)$, respectively.

Theorem 3.4 *Let s be any string of the alphabet $\{\alpha_1, \dots, \alpha_h\}$, and $\hat{s} = MTF(s)$. For any partition $s = s_1 \dots s_t$ we have*

$$|\hat{s}|H_0(\hat{s}) \leq 8 \left[\sum_{i=1}^t |s_i|H_0(s_i) \right] + \frac{2}{25}|s| + t(2h \log h + 9)$$

This theorem establishes that if MTF is used, the entropy of the resulting string cannot be too far from $\sum_i |s_i|H_0(s_i)$.

The following result is easily obtained, using theorem 3.4 and applying equation 3.1.

Corollary 3.5 *For any string s over $\mathcal{A} = \{\alpha_1, \dots, \alpha_h\}$ and $k \geq 0$ we have:*

$$|BW0(s)| \leq 8|s|H_k(s) + \left(\mu + \frac{2}{25} \right) |s| + h^k(2h \log h + 9)$$

where μ is defined in equation 3.1.

The following theorem bounds the performance of the $BW0_{RLE}$ algorithm, in terms of the modified empirical entropy H_k^* . Here, the RLE is used only to code long runs of zeroes. RLE is obtained by using two new symbols, $\mathbf{0}$ and $\mathbf{1}$, not included in the MTF output alphabet $\Sigma^{mtf} = \{0, \dots, h-1\}$. For a run of zeroes of length $m \geq 1$, let $W(m)$ denote the Wheeler code for m written using $\mathbf{0}$ and $\mathbf{1}$. We have $|W(m)| = \lfloor \log(m+1) \rfloor \leq m$ which implies $|RLE(\hat{s})| \leq |\hat{s}|$.

Theorem 3.6 *For any $k \geq 0$ there exists a constant g_k such that for any string s we have:*

$$|BW0_{RLE}(s)| \leq (5 + 3\mu)|s|H_k^*(s) + g_k$$

where μ is defined in equation 3.1.

From this result, we can derive that RLE plays a very important role in reducing the output size, both from a practical and a theoretical point of view.

We would like to remark again that the above results hold *simultaneously* for all $k \geq 0$, and thus k is not a parameter of the algorithm.

Recently, Giancarlo and Sciortino in [25] improved the above bound, using an BW variant called $BW0_{CD}$ which will be described in section 3.6.2.3. They showed that the following theorem holds:

Theorem 3.7 For any $k \geq 0$ there exists a constant g_k such that for any string s we have:

$$|BW0_{CD}(s)| \leq 2.5|s|H_k^*(s) + g_k$$

3.6 Burrows and Wheeler Variants

We have already seen how BWT algorithms consist of a sequence of three (possibly four) steps. In this section we present several variants proposed in order to improve their performance. The most of these variants aim to achieve better compression ratio in reduced time, but very often lack of a strong theoretical foundation.

3.6.1 First Step: BWT variants and optimizations

The most expensive step in BWT compression algorithms is certainly the sorting needed to construct the BW transform. In fact, it is clear that *MTF*, *RLE*, statistical order-0 compressors and the reverse BW transform are linear-time algorithms, while standard sorting operations cost $O(u \log u)$ (we recall that u is the length of a single block processed by BWT). As a consequence, a lot of research was accomplished in order to reduce sorting time. Some optimizations in the reverse transform have also been proposed in literature, to reduce the time required to reconstruct the original text. Another interesting line of research aims at finding different transformations, trying to further improve the final compression ratio.

3.6.1.1 Sorting optimizations

In the original formulation of the BW transform [11], Burrows and Wheeler proposed some variants of a standard quicksort algorithm to construct their forward transform. Since we need to sort the β rows of the $\beta \times \beta$ matrix \mathcal{M} , we have an average-case time complexity of $O(u^2 \log u)$ and a worst-case time complexity of $O(u^3)$ using this approach.

An improvement is the Bentley-Sedgwick algorithm [9] based on quicksort and radix sort. It uses a multi-way ordering method and a ternary partition technique. It has worst case time complexity $O(u^2)$ and an average complexity $O(u \log u)$. The memory occupancy here is $5u$ plus the memory needed for the quicksort stack. Its running time can be reduced, but at the cost of an extra $4u$ bytes. This algorithm is the one used by the `bzip2` program [46].

A different approach proposed in [11] exploits the suffix tree data structure (ST) to improve the forward transform calculation time. Given an input string $s\$$, the ST is a data structure which allows to retrieve all the suffixes of s in lexicographical order, simply using a depth-first visit. The tree exploits the fact that some suffixes can share a common substring, which can be stored in the tree by creating an intermediate node. The edge (i, j) of the tree is

labelled using the substring shared among all the nodes in the subtree rooted at j . As an example, see figure 3.7 for the input string “mississippi”.

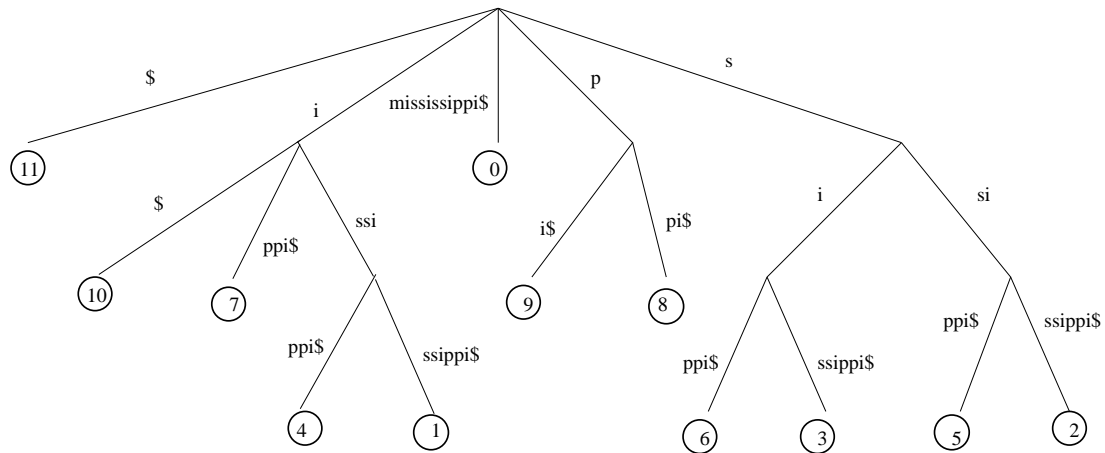


Figure 3.7: The suffix tree $ST(\text{“mississippi$”})$

Note that the space required by such tree representation is quadratic in the length of the input, u , since we store a suffix (which has a length proportional to u) for each leaf of the tree (and we have u leaves). A linear space representation can be obtained by storing for each edge e a couple of pointers $start_e, stop_e$ which denote the fragment of suffix associated to e .

There are many effective algorithms for building a suffix tree. In particular, Ukkonen in [49] showed a linear time construction algorithm. It can be used to build BWT in linear time, as follows. As we said, the suffix tree consists of all the suffixes of a string $s\$$ sorted lexicographically. As a consequence, if we build a suffix tree for $rev(s\$) = \$rev(s)$ we obtain a new tree consisting of all the prefixes of $s\$$, reading from the leaves to the root and from right to left. In figure 3.8 we represent the suffix tree for the reverse of “mississippi”. We have also represented the sorted matrix \mathcal{M} used in the construction of BWT (see section 3.3.1). Besides, we have assigned to each leaf of ST a number representing the order in which leaves are encountered in a depth-first visit. For each prefix in the tree, this number corresponds, in the sorted \mathcal{M} , to the row at the end of which the prefix is located. Using this observation, the symbol $F[i]$ in the sorted \mathcal{M} is the one which precedes the i -th suffix of the tree in the reversed input string $\$rev(s)$. This symbol can be retrieved in time $O(1)$ by using the label associated to the leaves of the tree (subtracting 1 to each of them, module the length of $s\$$, and taking the corresponding symbol in $\$rev(s)$). Balkenhol and Kurtz proposed a sorting linear-time algorithm, based on suffix trees, [33], whose memory complexity was reduced to $16u$ in the worst case and to an average of $8,83u$ on Calgary Corpus. A comparative survey of linear time suffix tree construction is given in [42].

Due to the huge memory requirements of these algorithms, other less memory-expensive

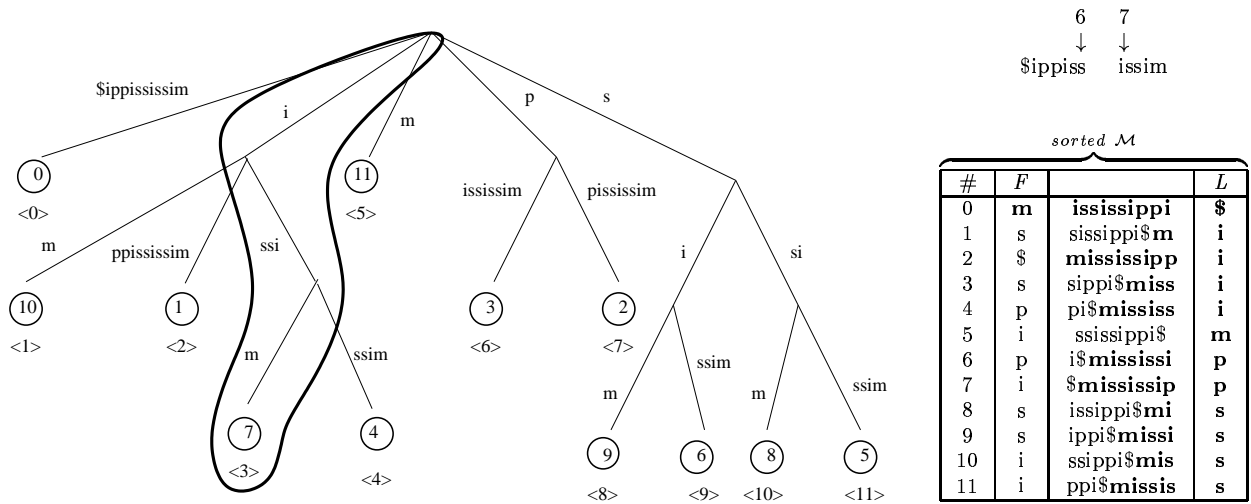


Figure 3.8: Suffix tree for $\$rev(\text{“mississippi”})$. Inside the bold line, we can see the prefix “missi”, reading from the leaf to the root and from right to left. This prefix is located at the end of the third row of the sorted \mathcal{M} (counting from zero). The symbol $F[3]$ is the sixth in the string “\$ippississim”, as we can see in the far right corner.

approaches are used in practice. In particular, an effective and elegant data structure which is often used is the suffix array, proposed by Manbers-Myers in [35]. It stores the starting positions of all suffixes of the input string s , sorted lexicographically (symbols are numbered from 0). Figure 3.9 gives the suffix array for the input string $s = \text{“mississippi”}$ (we have also listed the suffixes, even if the suffix array does not store them). Note that, a suffix array can just be seen as the list of the labels associated to the leaves of the suffix tree built on a given input string. Leaves are visited in a depth first way. Hence, a suffix array can be built in linear time by using the linear time suffix tree construction.

In [35] Manber and Myers presented an algorithm for directly building the suffix array with worst case time complexity $O(u \log u)$. It requires $8u$ bytes of space. Another algorithm was presented by Sadakane in [43]. It has the same worst case time complexity $O(u \log u)$ and memory complexity of $8u$, and combines the Manber-Myers with the Bentley-Sedgwick algorithm. In practice, this algorithm is faster than other sorting algorithms when LCP is large (LCP is the length of the longest common prefix between two consecutive suffixes in the suffix array).

In [47] Seward proposed another direct-comparison based algorithm. It only uses $5u$ bytes and has worst time complexity $\Theta(u^2 \log u)$, but in practice it is one of the fastest applied to blocks with small LCP. One among the fastest algorithms known is given in [39] by Ferragina and Manzini. Their algorithm requires $u(5 + c)$ bytes, where c is a tunable parameter generally not larger than 0,03. It has $O(u^2 \log u)$ worst-case time complexity, suffered just in pathological cases when average LCP is $\Theta(u)$. In practice, it works very well in almost all

10	7	4	1	0	9	8	6	3	5	2
i	i	i	i	m	p	p	s	s	s	s
	p	s	s	i	i	p	i	i	s	s
	p	s	s	s		i	p	s	i	i
	i	i	i	s			p	s	p	s
		p	s	i			i	i	p	s
		p	s	s				p	i	i
		i	i	s				p		p
			p	i				i		p
			p	p						i
			i	p						
				i						

Figure 3.9: Suffix array for “mississippi\$”

cases.

All the above algorithms require an extra space of $\Theta(u)$ (in most cases the constant is at least 4) in addition to the input and the suffix array. There is a new algorithm requiring $O(u \log u)$ worst-case running time and sublinear extra space [10]. Moreover, very recently a number of direct linear-time algorithms have been proposed in [30, 31, 32]. The description of such state-of-the-art algorithms is out of our notes’ scope and we suggest the interested readers to refer to the bibliography.

A different line of optimization tries to improve compression by using an ordering schema different from the lexicographical one. In [14] Chapin proposed a schema which reorders the alphabet using the frequencies of symbols observed in a given corpus (for instance “AEIOU...” instead of “ABCDE...”). Then, this initial idea is extended considering the order of contexts as a walk through the various contexts. The problem is based on conditional probabilities instead of frequencies, and become an NP-hard one. It is solved as an instance of a traveling salesman tour. Alphabet reordering is also used in [7, 26]

Another optimization is based on the fact that, during the sort phase, a lot of time can be wasted, if we compare two contexts which share a long common prefix. For instance, consider the suffixes $s_1 = a^n X$ and $s_2 = a^n Y$. The first different symbol is placed at least in position $n + 1$, and so comparing s_1 and s_2 takes at least $n + 1$ steps. As a consequence, Nelson in [41] adopts a run length encoding before the performing of the BW forward transformation.

3.6.1.2 Inverse BWT improvements

At the end of section 3.3.2 we presented the classical algorithm proposed in [11] to reverse the BWT. A number of different optimizations have been proposed by Seward in [48] and one of them is currently used in the bzip2 compressor.

Recalling the notation given in section 3.3.2, we have that $FL[i] = Occ[F[i]] + Rank[i]$ is

the vector used to implement the FL-Mapping. Since both *Occ* and *Rank* are not modified during the reverse transform, we can avoid to effectively materialize the vector *FL*, by substituting the line number 3 of the given algorithm with $j = Occ[F[i]] + Rank[i]$. In this way we can save $\Theta(u)$ memory space, where u is the input block length. However, using the above approach we introduce a potential problem of cache-miss when the input block size is larger than the machine cache. In fact, both *F* and *Rank* are accessed for each iteration of the algorithm in an essentially random order, potentially generating 2 cache-misses per iteration.

Seward suggested a simple optimization to mitigate this kind of problem. Assuming that each input block is not larger than $2^{24} = 16\text{M}$ bytes and that $|\Sigma| \leq 256$, he can pack in a 32-bit machine word both $V[i]$ (in the first 24 bits) and $F[i]$ (in the last byte). Using this approach we are sure that we can calculate $j = A[F[i]] + V[i]$ accessing only one word, and then we can save one cache-miss.

Even if this modification could appear trivial, it surprisingly achieves a speedup between 61% and 67% on the Canterbury corpus.

3.6.1.3 Different Transforms

Sorting Limited Order Contexts. Schindler in [44] proposed a transform slightly different from the BW one. It aims to reduce the sorting time, at cost of a little loss in compression ratio. The key idea is to construct the transform using an ordering based on contexts of *fixed* length k .

Given an integer k and the input string s , the frequencies of all length- k contexts (substrings) in s are calculated. Then, the contexts are sorted, from right to left, using a counting sort routine in linear time. The final output is composed by all the symbols which follow the sorted contexts in s . If we have two or more equal contexts, symbols are outputted in the same order in which they are found in the input string s . It is necessary to precede this output with the initial length- k context of s . For instance, in figure 3.10 you can see the forward transform applied on “mississippi”

The forward and the reverse transform must agree on the context length k . The reverse transform works using the following steps:

- Output the first k symbols of the given $sz(s)$. Working on the substring z composed by the remaining $|sz(s)| - k$ symbols of $sz(s)$, reconstruct all length- k contexts, as follows:
 - Sort lexicographically the symbols in z and obtain the string z' ;
 - Note that by coupling each symbol in z with the corresponding one in z' we obtain all the length-2 contexts;

$\begin{array}{cccccccccccc} \uparrow & i & i & i & i & m & p & p & s & s & s & s \\ \hline & m & s & s & p & i & p & i & i & i & s & s \\ \hline \end{array}$ $sz_{k=1}(\text{'mississippi'}) = \text{"mmsspippiiss"}$	$\begin{array}{cccccccccccc} & m & p & s & s & i & i & p & i & i & s & s \\ \uparrow & i & i & i & i & m & p & p & s & s & s & s \\ \hline & s & m & s & p & i & p & i & s & s & i & i \\ \hline \end{array}$ $sz_{k=2}(\text{'mississippi'}) = \text{"mismspipissii"}$
-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Figure 3.10: The Schindler forward transform. On the left side using a context of length $k = 1$, on the right side a context of length $k = 2$. In bold we indicate the initial length- k context of s . We remark that the contexts are sorted following the direction showed by the arrows.

- Sort lexicographically these contexts from right to left using radix sort, where all length-1 contexts are already sorted, and obtain z'' ;
- Note that the previous steps can be iterated for contexts of any length $k > 2$, since a context of length i and its successor form a context of length $i + 1$.
- The contexts re-constructed at previous steps maintain the relative order, just like in the BW. So it is possible to reconstruct the original string following a similar approach.

For instance, in figure 3.11 we calculate the reverse transform $sz_{k=2}^{-1}(\text{'mismspipissii'})$, obtained on the right side of figure 3.10. We start by outputting the length-2 prefix “mi” and obtaining the two strings $z = \text{'smspipissii'}$ and $z' = \text{'iiiihppssss'}$. Then, we sort the first four symbols in z and couple them with the first four identical symbol of z' ; the 5-th symbol of z is coupled with the 5-th symbol of z' ; the next two symbols of z are sorted and coupled with the next two identical symbols of z' ; the last four symbols of z are sorted and coupled with the last four identical symbols of z' . The resulting string z'' is “mpssiipiiss”. To conclude the reconstruction, we start from the first context “mi”, explicitly premitted to the forward transform. Then, for each context, we output the corresponding symbol in the string z , obtaining a new length-2 context. In our example we first output ‘s’, obtaining the new context “is” (the ‘i’ of “mi” and the ‘s’ just outputted). Then we look for the first “is” context, outputting ‘s’ and obtaining the new context “ss”. The next outputted symbol will then be ‘i’, and so on. When we encounter two or more identical contexts, we take the first yet unused (by construction). This iterative process stops after $|z|$ steps, since z is a permutation of the input string.

LPSA. In [3] Arnavut and Magliveras proposed a interesting reduction of BWT to another transform which they called LPSA (Lexical Permutation Sorting Algorithm). They firstly examined the case in which the input block is a permutation (i.e. an unsorted sequence of length u containing all natural numbers between 1 and u). In this case, they showed that the columns of the matrix used by BWT form a cyclic group whose elements are all

$$\begin{array}{cccccccccccc}
z'' = & m & p & s & s & i & i & p & i & i & s & s \\
z' = & i & i & i & i & m & p & p & s & s & s & s \\
\hline
z = & s & m & s & p & i & p & i & s & s & i & i
\end{array}$$

Figure 3.11: The Schindler reverse transform. The reconstructed contexts are formed by the couples $z''[i] z'[i]$ for each i . They are in order “mi”, “pi”, “si”, “si”, “im”, “ip”, “pp”, “is”, “is”, “ss”, “ss”. Starting from the context “mi” we obtain “s” and, iterating the method, the whole original string “mississippi” is retrieved.

permutations. It is then possible to select a specific column g to generate the entire group, by applying g to itself iteratively. As an example, consider the permutation $\pi = (5, 3, 1, 2, 4)$. Given an input block of 5 symbols, this permutation places the 5-th symbol of the input in the first position of the result, the 3-rd symbol in the 2-nd, the first symbol in the 3-rd, the 2-nd symbol in the 4-th, and the 4-th symbol of the input in the 5-th position of the result. If we form the *BWT* matrix for all the left cyclic shifts of π , and then sort lexicographically from right to left, we obtain:

$$M' = \begin{pmatrix} 2 & 4 & 5 & 3 & 1 \\ 4 & 5 & 3 & 1 & 2 \\ 1 & 2 & 4 & 5 & 3 \\ 5 & 3 & 1 & 2 & 4 \\ 3 & 1 & 2 & 4 & 5 \end{pmatrix}$$

If we choose one specific column as a generator, for example $g = (3, 1, 5, 2, 4)$, we can obtain all other columns by applying iteratively g to itself: $g^2 = (5, 3, 4, 1, 2)$, $g^3 = (4, 5, 2, 3, 1)$, $g^4 = (2, 4, 1, 5, 3)$, $g^5 = I = (1, 2, 3, 4, 5)$, $g^6 = g$. Then, the permutation g is a generator for all permutations which are the columns of M' . The input permutation λ can be entirely recovered if we know one generator g of the cyclic group formed by the columns of $M'(\lambda)$, and the power i such that $g^i = I$. There are $\phi(u)$ generators in this group, where n is the length of the input permutation (and then the number of elements of the group) and ϕ is the Euler’s function giving the number of generators of a group of u elements.

In order to handle the general case in which the input is not a permutation, the authors propose, given the input string s , to construct the *lexical index permutation* of s , λ_s , defined as that permutation which sorts the symbols of s , maintaining the relative order existing in s between two identical symbols. For example, the lexical index permutation of $s =$ “mississippi”, would be $\lambda_s = (2, 5, 8, 11, 1, 9, 10, 3, 4, 6, 7)$. To encode λ_s we can use the above result, choosing one generator of the cyclic group $M'(\lambda)$, and its relative power.

The main difference between LPSA and *BWT* is that the latter always chooses the same generator in this group (it always chooses the last column in the shifts matrix), but nobody tells us that this is the best choice we could make among all generators.

However, to reconstruct the input string given its lexical index permutation λ_s we need a vector V containing the frequencies of the alphabetic symbols in the input. For example

in the case of $\lambda_s = (2, 5, 8, 11, 1, 9, 10, 3, 4, 6, 7)$ as before, we need that something tells us that we have 4 symbols ‘i’ in the input, 1 symbol ‘m’, 2 symbols ‘p’ and 4 symbols ‘s’. The transformed version of the input will then contain a generator for λ_s , the power of this generator, and V . This is not a big overhead, since we have to transmit also the permutation. In fact, the permutation has length u and requires $O(\log u)$ bits for each position, while V has length $|\mathcal{A}|$ and requires $O(\log u)$ bits for each position. So, we have that the overhead required by V is:

$$\frac{O(|\mathcal{A}| \log u)}{O(u \log(u)) + O(|\mathcal{A}| \log(u))} = O\left(\frac{1}{u}\right) \quad \text{if } |\mathcal{A}| \text{ is constant.}$$

3.6.2 Second Step: MTF variants

Move to Front is a rather severe policy. It replaces the symbol at the head immediately, even if it has been heavily used. In this section, we review different strategies proposed in the literature to mitigate this effect. All the strategies which were proposed in this field aim to reduce the ranks in the output of the MTF transform (in order to improve compression ratio), and try to avoid that very large numbers appear close to a run of very low ranks. In the following, we denote with Σ the alphabet used by $bw(b)$, the BW forward transformation, and with Σ^{mtf} the alphabet used by the MTF variants.

3.6.2.1 M1FF2, MTF-1, MTF-2, Time-Stamp, Best x of 2x - 1, MTF Grouping and Sticky MTF

A first class of MTF variants aims to introduce a certain kind of control in the head symbol replacement. For instance, Schindler in [44] proposed to send the newly accessed symbol to the second position, if the symbol at the head was the last one access. Otherwise, it is moved to the head. He also investigated a different methodology, called “M1FF2”, which sends the newly accessed symbol to the second position if the symbol at the head was accessed the last time or the time before. This happens unless the symbol accessed was already the second symbol, in which case it goes to the head.

Balkenhol [7] proposed a modification called “MTF-1” which improved compression ratio on some files. In this case, the difference with MTF is that only the symbol from the second position in the list are moved to the first position. The symbols from higher positions are moved to the second position. “MTF-2” is a slight variant of MTF-1 where the symbols from the second position are moved to the beginning of the list only when the last symbol output was not 0. For instance, consider the string ba^nba^m and start with a list containing the symbols a and b in alphabetical order. The standard MTF algorithm would output the sequence $110^{n-1}110^{m-1}$, while MTF-1 and MTF-2 would output respectively the sequences 10^n110^{m-1} and 10^n10^m .

Fenwick in [20] introduced the Sticky MTF method, where a symbol is left at the head of the MTF list if it has been referenced the last two times, producing a zero rank on the second reference. If the symbol at the head is not accessed twice in succession, this algorithm behaves like the standard MTF.

Another proposal was the “Time Stamp”. It uses a list of distinct symbols L , as in vanilla MTF, and, for each accessed symbol s_j , its position in L is outputted. Then, the algorithm moves the character s_j before the first item of L that has been requested at most once since the last request of s_j . If all the items of L have been requested at least twice, or s_j has not been requested so far, then s_j is left at the same position.

Generalizing these approaches, in [13] Chapin suggested to switch among M1FF2 and a new method called “Best x of $2x - 1$ ”, according to the type of text to compress. “Best x of $2x - 1$ ” ranks a new symbol s_j , taking into account all the already seen symbols. Given any two symbols s_i and s_j in the list, they are sorted such that the symbol that was requested x or more times in the previous $2x - 1$ requests for s_i or s_j is put in front. For example, in Best 2 of 3, if the last three request for symbols a or b were aaa , aab , aba , baa , then a is put in front of b , otherwise b is put in front of a . Best 1 of 1 is the vanilla MTF, while Best 2 of 3 is the Time Stamp algorithm.

An interesting proposal came by Balkenhol and Sharkov in [7]. They noted that Σ^{mft} typically shows a probability distribution of integers which decrease monotonically for large integer (as shown in figure 3.6). So they suggest to divide the sequence produced by MTF in two different sequences. The first one, called $x^{mft,1}$, is constructed over the alphabet $\{0, 1, 2\}$ by replacing in vanilla MTF all the occurrences of integers greater than 1 with 2. The second, called $x^{mft,2}$, is constructed over the alphabet Σ by removing the symbols for which the integers 0 or 1 appear in $x^{mft,1}$. The two subsequences are coded differently. $x^{mft,1}$ is seen as a Markov chain and is encoded using an arithmetic coder whose conditional frequencies are obtained from this chain. $x^{mft,2}$ is further segmented in groups and then encoded using again an arithmetic coder whose conditional frequencies are obtained using an heuristic approach.

3.6.2.2 Inverse Frequency, Distance Coding

The “Inversion Frequencies” is a completely different approach to convert the local homogeneity given by a BW transform into a global one. IF is a method, described by Arnavut and Magliveras in [2], which calculates a sequence x^{if} of integers in the $[0, u - 1]$ range, being u the length of $bw(b)$. For each symbol s_j in Σ , it scans $bw(b)$ and outputs the position of the first occurrence of s_j . For further occurrences of s_j , the method outputs an integer which is the number of symbols greater than s_j occurred since the last request of s_j . Note that, given an IF representation to recover the original $bw(b)$, there is the necessity to encode the number of occurrences of each symbol in Σ .

The “Distance Coding” proposed in [15] is strategy which scans the string $bw(b)$ and for each symbol s_j of Σ , placed in position i , looks for the next occurrence of s_j placed in position p . Then it outputs the distance $p - i$. When there are no more occurrences, it outputs 0. The encoder must encode the first position of all symbols in Σ .

3.6.2.3 BW compression without MTF

A very interesting approach was suggest by Moffat in [40]. It observes that the BTW text can be seen as a sequence of segments that correspond to conditioning classes. The arrangement of symbols within each segment, can be assumed as independent of all others.

As a consequence, it suggests to remove at all the usage of an MTF transform. Instead, he adopts directly an entropy 0 arithmetic coder (AC) imposing that the conditioning probabilities are *changed and recomputed* at the end of each segment. We point out that in this solution, AC encodes directly raw BW forward transform symbols, rather than MTF ranks. Several tests performed on the “Calgary” and “Large Canterbury” corpus show that this approach has very good compression performance (for instance it compress better than `szip`), but requires a lot of processing time since it needs to compute conditional probabilities for each segment (it takes as long as PPDM, one of the slowest compression algorithm, to encode some files in the corpus). Note that the encoder must explicitly inform the decoder of segment (conditioning class) boundaries. We return on this in the next sections, where we analyze the use of hierarchical arithmetic encoders.

In a recent paper Giancarlo and Sciortino [25] proposed to substitute MTF with a different approach. They calculate $BW(b)$ and then use a dynamic programming algorithm CD to compute the best partition of $BW(b)$ in terms of a base compressor. Then they compress $BW(b)$ according to that optimal partition using the base compressor. The dynamic program which computes the optimal partition of a string b is:

$$CD : E[j] = \min_{0 \leq k \leq j} E[K] + |C(b[k+1, j] \#)|, j = 2 \dots u \quad (3.4)$$

where $C(x \#)$ is the output of the base compressor (arithmetic coder), $|C(x \#)|$ is its length in bits, b is the string of length u , and $b[k, j]$ denotes the substring of b starting at position k and ending at position j . The problem with CD is its quadratic complexity. The work of Giancarlo and Sciortino was also interesting from a theoretical point of view since they showed one of the best bound without any explicit hypothesis about the source (see theorem 3.7).

3.6.3 Third Step: RLE variants

Run Length Encoding aims to shrink long runs of the same symbols. Long runs have a deep impact both on the cost of the sorting phase during BW forward transform, and on the cost of symbol encoding performed by an entropy 0 encoder. So, Run Length Encoding is successfully used in both the context.

Several RLE variants were proposed. We will review the most common ones. In [36] Maniscalco suggests to transform the runs of length 3 or greater for any symbol in the run. The encoded string is long $2 + \lfloor ((R - 1)/2) \rfloor$, where R is the length of the original run. To reverse this an additional bit is appended. It has value of 0, if the original run was odd in length, 1 otherwise. The decoder reverses the run length codification using the formula $((R' - 2) * 2) + 1$ where R' is the length of the encoded run. This value is incremented by one if the fixed length 1 bit code has the value of 1. For instance, the string “AAA” is encoded in “AAA” + 0, ‘AAAA” is encoded in “AAA” + 1, ‘AAAAA” is encoded in “AAAA” + 0, and so on.

As seen in section 3.4.2, Burrows and Wheeler [11] suggest to encode just the 0 symbol, since it is the most frequent symbol in Σ^{mtf} . So, a long run of 0 is replaced by a special symbol **0** non contained in Σ^{mtf} followed by the length of the run coded using a variable length coder. Different Run Length Encoding strategies are compared by Fenwick in [19].

We recall that Manzini in [38] showed the importance of RLE both from a theoretical and a practical point of view (see theorem 3.6).

3.6.4 Fourth Step: Statistical Compressor variants and Variable Length Encoders

In section 3.4 we introduced the $BW0$ and $BW0_{RLE}$ algorithms. Both of them use a standard order-0 entropy encoder. Note that real implementations of BW algorithms, such as `bzip2` and `gzip`, use Huffman or Arithmetic Coding variants known as Multi-Huffman and Hierarchical Arithmetic. In the following we review them, together with the use of variable length encoders (such as a δ -Elias or Wheeler code) for compression.

3.6.4.1 Hierarchical Arithmetic Encoders

Fenwick in [19] did an interesting experiment. He tested a number of different standard context-based compressors, both on raw BWT text and on MTF output, and he found that neither PPM nor dictionary-based compressors (LZ77/LZ78) performed better than any order-0 model. He showed that the same result also holds comparing order-0 compressors with any order- k compressors with $k > 0$. So, he concluded that BW transform removes all contextual information or, better, it exploits all the available context information during the

transform itself.

Although there is a general consensus about the fact that, working with BW text or MTF ranks, contextual compressors do not improve the ratio achieved by an Order-0, some authors showed that few contextual information still remains and can be exploited using the so called “hierarchical encoders”.

As we saw in paragraph 3.2, Arithmetic Coder offers a better compression ratio than the Huffman one. So researchers concentrated their attention on improving compression effectiveness of this algorithm.

Using a full Order-1 encoder, with an alphabet Σ of 256 symbols, we would have a 256 x 256 frequency matrix where each row represents a different conditioning class and each column represents a given symbol to be coded. The hierarchical model partitions the alphabet in a limited number of encoding classes (generally no more than 7-8). Each class contains a variable number of symbols. The frequency $f(C_i)$ of a class C_i is defined as the sum of the frequencies of all its symbols, and this value is maintained using a per class counter. Symbols are assigned to the classes so that $f(C_i)$ are almost equal each others.

Applying the above reasoning for encoding a typical MTF output, which has a skewed distribution as seen in fig. 3.6, one can equalise the coding probabilities by grouping symbols so that the n -th group contains the next 2^n alphabet symbols. For each symbol α belonging to the class C_i we have a frequency counter $N(\alpha, C_i)$. Besides, we have a per class-counter $r(C_i)$ incremented whenever a symbol belonging to C_i is read. Processing the j -th symbol $s[j]$ in the input string the following actions are performed:

1. Let C_i be the class containing $s[j]$. We update all per class frequencies as follows:

$$f(C_p) = \begin{cases} \frac{f(C_p) * (j-1) + 1}{j} & p = i \\ \frac{f(C_p) * (j-1)}{j} & \text{o.w.} \end{cases}$$

2. We update all frequency counters related to the symbols in C_i as follows:

$$N(\alpha, C_i) = \begin{cases} \frac{N(\alpha, C_i) * r(C_i) + 1}{r(C_i) + 1} & \alpha = s[j] \\ \frac{N(\alpha, C_i) * r(C_i)}{r(C_i) + 1} & \text{o.w.} \end{cases}$$

3. We update the C_i per class counter setting $r(C_i) = r(C_i) + 1$

Using this method with the Arithmetic Compressor, the current MTF rank $s[j]$ selects a class C_i . The values $N(s[j], C_i)$ is then used to partition the current interval as described in section 3.2.2. The encoder emits an escape character to signal to the decoder the change of

class, when needed. The space wasted in the output for these escape symbols is more than compensated by the improved compression ratio.

Note that, given a class C_i , we consider the values $N(s[j], C_i)$ as a *per class* frequencies. This produces several benefits:

- i) We have a performance gain since updates will involve only the frequency counters related to a single class. Moreover, given the skewed distribution of MTF, we will usually update classes composed by few symbols;
- ii) We can exploit all the available machine precision within a class;
- iii) Classes with fewer elements will adapt quickly to the input behaviour. Instead, classes with many elements will adapt more slowly but these elements will occur less frequently. In this sense the coding probabilities are equalised.

This model can be specialized in two ways. The first is the *cascading* model used by Balkenhol and Kurtz in [8], which represents ranks as a linked list of classes. For instance, assuming that ranks greater than n are linked using an escape symbol $\Theta_{\geq n}$, we could have:

$$\{0, 1, \Theta_{\geq 2}\}, \{2, \Theta_{\geq 3}\}, \dots \{129, \dots 255\}$$

The second is the *structured* model, firstly introduced by Fenwick in [19], where there are symbols which acts as selectors for choosing the correct class. For instance, Balkenhol and Shtarkov suggest to adopt four different selectors. The α class which “adapts” very quickly, the β class moderately quickly, the γ class slowly, and the δ class which doesn’t adapt at all. Using the term “adapt” we mean that given a class, the smaller its cardinality, the sooner its frequencies are updated. Using a cascading strategy for encoding MTF ranks, Balkenhol achieved a compression ratio just 1% worse than PPMD, one of the best but slowest compressors. In figure 3.12 we show a graphical representation of these two approaches. In the cascading approach, we have a “cascade” of escape characters to encode an inner class, while in the structured approach we have a special symbol for each class. Note that, a hierarchical model can be also used to encode raw BW text instead of MTF ranks. In fact, this is the approach taken by Moffat in [40], a paper we reviewed in section 3.6.2.3

A similar two-class approach is used by Fenwick and others in [21]. They propose to exploit a cache system where the most probable symbols are stored in a prominent foreground model, and the bulk remaining symbols stored in a larger background model. Since the context of the highly probable symbols in the foreground model are not distorted by the more skewed background model, this method achieves a performance which closely approaches the PPM compression ratio.

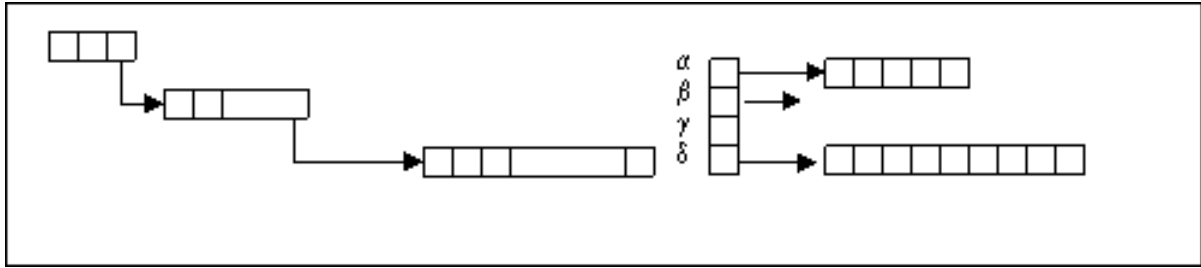


Figure 3.12: Cascading and Structured Hierarchical approaches

3.6.4.2 Multi-Huffman

The same idea explained above for Arithmetic Coder can be also exploited for the Huffman one. We can define a set of tables which gives adaptation to changing statistics and reduces inefficiencies of Huffman Coding. The input symbols are then grouped into classes according to their frequencies, and an Huffman table is built for each class. When the input is encoded, an escape character is emitted by the coder to signal to the decoder each change of class. Also in this case, the space wasted in the output for the transmission of tables and for the escape symbols is more than compensated by the improved compression ratio.

3.6.4.3 Variable Length Encoding

Recently, Fenwick in [20] moved the experimentation on final BW steps at the extreme point. He noted that since the first publication of Burrows and Wheeler vanilla method in 1994, there has been little more than 7% improvement in compression ratio. This is a witness about the cleverness of the original Burrows and Wheeler's proposal, but also posed an interesting question: *"Do the BW transform results depend intrinsically on the chosen order-0 encoders?"*

Fenwick, showed that simple variable-length codes, such as δ -Elias or Wheeler coding, are a viable alternative to the more usual block-adaptive Huffman coders usually employed in a Burrows-Wheeler compressor, giving compression within 10% of the ratio of the bzip2 compressor and 14% of the best reported results.

Variable Length is an encoding method also used in several theoretical analysis such as [1, 37]. This is true for essentially two reasons: firstly because these encoding methods provide well known bounds, and because they do not lose very much in compression ratio.

3.6.5 Word Based BWT

A new line of research is the application of BW transform to word based files. A word based file contains textual information organized in words and separated by space separators (such as spaces, tabs and so on).

In [4] Awan proposed a strategy to compress english text files using a dictionary of words and the BW transform. Starting from a given English corpus, a dictionary D is built. Words are sorted by length and, among words with the same length, by frequency. Dictionary is shared one time between the encoder and the decoder. Then, the input text is parsed, and each word w_i in D is replaced with a sequence of symbols representing the position of w_i in D . Words not in D are just marked to point out that they are left unchanged. Then, the modified input text is compressed using a BW transform on characters, a standard MTF and an entropy-0 encoding. This strategy gives an improvement of compression of about 5% on Calgary Corpus, excluding the cost of sending the dictionary.

Moffat in [27] extended this approach suggesting to prepend a parsing phase to word based text files compressed with BW algorithms. He pointed out that the dictionary should be encoded with the compressed text. In order to save space he compared two different approaches to build the dictionary. The first builds a dictionary composed of the most common n -grams (a n -gram is a substring of length n) of the analyzed words. The second isolates contiguous strings of characters, and transforms them into integers via the use of a dictionary of strings, with the additional rule that if the non-word string between two word strings consists of a single blank character, it is elided. This is the so called *spaceless words* mechanism. In his comparison word based dictionaries using spaceless model outperform the n -gram model. The dictionary is sent by the encoder compressing it separately and prepending it to the compressed input text. Both the compressions are made using BW algorithms.

Moffat also analyzed in great detail in [27, 29] the impact of applying a vanilla Move to Front transform to a word based file. Since symbols are full words they appear less frequently than single characters. So, a list based approach can be too expensive for the phase of search and replacing (also because we have to compare strings instead of symbols). He suggested to use a forest $\{T_i\}$ of search trees. The i th tree in the forest contains S_i words, with the most accessed items stored in tree T_0 , the next most accessed words being stored in T_1 and so on. Several different pseudo-MTF transformations can be realized following this approach, just changing the policy of update for trees in the forest and the type of trees used (i.e. binary, splay and so on).

3.7 BWT-based Compressors

In this section we review three of the most used *BWT*-based compressors, namely `bzip2`, `szip` and `bwzip`.

bzip2 [46] is one of the most used compression programs, and it was made by J. Seward. The software uses the Burrows-Wheeler block sorting text compression algorithm, and Huffman coding. Compression is generally considerably better than that achieved by more conventional LZ77/LZ78-based compressors, and approaches the performance of the PPM family of statistical compressors. It is possible to choose the block size, while the default one is fixed at 900K (the maximum possible size). Compression and decompression requirements, in bytes, are estimated as:

Compression: 300K + (8 * blockSize)
Decompression: 6 * blockSize

So for files compressed with the default 900k block size, bzip2 will require about 3,7 Mbytes to decompress. Seward suggests to use the largest block size available, since it maximises the compression achieved. Compression and decompression speed are virtually unaffected by block size. Several very interesting tricks and optimizations are used. In [47] he describes the optimizations related to the memory usage and cache misses during the block sorting used in forward BW transform. In [48] he describes the tradeoffs between time and space in the inverse BW transform. Some of them were reviewed in the previous sections.

szip [44] is a freeware portable general purpose lossless compression program. It has a high speed and compression, but high memory demands (up to 20MB) too. szip uses the blocksort variant described in section 3.6.1.3 with a maximum blocksize of 4.1MB (default 1.7MB), which means compression is better for larger files. It uses a byte-oriented arithmetic coder for output and a hierarchical prediction model specifically developed for blocksort.

bwtzip [34] is an ongoing project, distributed under the GNU General Public License, to implement a Burrows-Wheeler compressor in standard, portable C++. It is research-grade in that it is highly modularized and abstracted, so that it is simple to swap out parts of the compressor without affecting anything else. This makes it easy to experiment with different algorithms at different stages of compression. Bwtzip and bwtunzip both are true linear-time algorithms, since they use suffix trees. According to some experimental tests performed by the author, it seems that this program runs slightly better than bzip2 (at least on the files taken into consideration).

3.8 Experimental Results

Since 1994 researchers focused their attention on improving compression ratio achieved by BW algorithms. The compression performance (in bpc, or “bits per character”) for Calgary

Author	Date	Algorithm	bpc
Burrows & Wheeler	1994	16000 Huffman blocks	2,43
Fenwick	1995	Order-0 Arithmetic	2,52
Fenwick	1996	Structured Arithmetic	2,34
Schindler (szip)	1996	Hierarchical Arithmetic	2,36
Balkenhol	1998	Cascaded Arithmetic	2,30
Fenwick	1998	Sticky MTF, Structured Arithmetic	2,30
Balkenhol	1999	Cascaded Arithmetic	2,26
Deorowicz	2000	Arithmetic Context	2,27
Deorowicz	2000	Weighted Frequency Count	2,25
Seward (bzip2)	2000	Huffman blocks	2,37
Wirth, Moffat	2001	No MTF; PPM Context	2,35
Arnavut	2002	Inversion Ranks	2,30
Fenwick	2002	VL codes, 1000 blocks	2,57
<i>PPMD</i> ⁺		10 time slower than BWT	2,28

Figure 3.13: Experimental results on Calgary corpus

Corpus is summarized in table 3.13. As noted in [20] “*compression improvements are difficult, with even the best results improving on early 1996 results only 4%*”.

Note that, Burrows and Wheeler variants achieve very good results in term of speed and compression ratio when compared with *PPMD*⁺, a compressor generally considered among the most performant in terms of compression ratio.

3.9 Conclusions and Future Works

In this lecture we presented the Burrows and Wheeler transform and its application in lossless compression. We also reviewed several variants used to improve the compression ratio or the running time of compressors. Their experimental results lead us to conclude that *BWT* is a good choice for achieving very good compression with reduced time needs. We notice that the power of *BWT* relies in the cleaverness of the transform itself, and that the additional steps can be tuned up, mixed or modified (even omitted) for improving the final results. We think that, as long as BW-based compressors will reach a sufficient large number of users, they will replace the current generation of dictionary-based compressors as it happened in the past with dictionary-based compressors and Huffman-based ones.

Nowadays, a number of researchers is still interested in *BWT*. An interesting development consists in methodologies for *integrating* BW compression and indexing [22, 23].

Another interesting line of research is related to the use of *BWT* for encrypting [45]. In

fact, the key to reverse the transform is contained in the number enclosed in the *BWT* output (the row in the sorted matrix \mathcal{M}). Without this number an enemy takes $O(n)$ time to decrypt a message encoded by *BWT*.

References

- [1] M. Arimura and H. Yamamoto. Asymptotic optimality of the block sorting data compression algorithm. *IEICE Trans. Fundamentals*, E81-A(10):2117–2122, 1998.
- [2] Z. Arnavut. Move-to-front and inversion coding. In *Data Compression Conference*, page 532, 1998.
- [3] Ziya Arnavut and Spyros S. Magliveras. Block sorting and compression. In *Designs, Codes and Cryptography*, pages 181–190, 1997.
- [4] F.S. Awan, N. Zhang, N. Motgi, R.T. Iqbal, and A. Mukherjee. Lipt: A reversible lossless text transform to improve compression performance. In *Data Compression Conference*, 2001.
- [5] B. Balkenhol and S. Kurtz. Universal data compression based on the burrows and wheeler transformation: Theory and practice. Technical report, Universitat Bielefeld, <http://www.mathematik.uni-bielefeld.de/sfb343/preprints/>, 1998.
- [6] B. Balkenhol and S. Kurtz. Universal data compression based on the burrows and wheeler-transformation: Theory and practice,. *IEEE Transactions on Computers*, 49(10):1043–1053, 2000.
- [7] B. Balkenhol and Y. Shtarkov. One attempt of a compression algorithm using the BWT. Preprint 99-133, SFB343 Faculty of Mathematics, University of Bielefeld, 1999. <http://www.mathematik.uni-bielefeld.de/sfb343/preprints/pr99133.ps.gz>.
- [8] Bernhard Balkenhol, Stefan Kurtz, and Yuri M. Shtarkov. Modifications of the burrows and wheeler data compression algorithm. In *Data Compression Conference*, pages 188–197, 1999.
- [9] 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.
- [10] S. Burkhardt and J. Kärkkäinen. Fast lightweight suffix array construction and checking. *CPM*, 2003.
- [11] M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. Technical Report 124, Digital – Systems Research Center, 1994.
- [12] Calgary corpus
<ftp://ftp.cpsc.ucalgary.ca:/pub/projects/text.compression.corpus/>.
- [13] B. Chapin. Switching between two on-line list update algorithms for higher compression of burrows-wheeler transformed data. In *Data Compression Conference*, 2000.

- [14] Brenton Chapin and Stephen R. Tate. Higher compression from the burrows-wheeler transform by modified sorting. In *Data Compression Conference*, page 532, 1998.
- [15] Sebastian Deorowicz. An analysis of second step algorithms in the burrows-wheeler compression algorithm. Technical report, Silesian University of Technology, 2000.
- [16] Sebastian Deorowicz. Improvements to Burrows-Wheeler compression algorithm. *Software – Practice and Experience*, 30(13):1465–1483, 2000.
- [17] Sebastian Deorowicz. Second step algorithms in the Burrows-Wheeler compression algorithm. *Software – Practice and Experience*, 32(2):99–111, 2002.
- [18] M. Effros, K. Visweswariah, S. R. Kulkarni, and S. Verd. Universal lossless source coding with the burrows wheeler transform. *IEEE Transactions on Information Theory*, 48(5):1061–1081, May 2002.
- [19] P. Fenwick. Block-sorting text compression — final report. Technical report, The University of Auckland, Department of Computer Science, March 1996.
- [20] P. Fenwick. Burrows wheeler compression with variable length integer codes. *Software - Practice and Experience*, 32(13):1307–1316, Nov 2002.
- [21] P. Fenwick, M. Titchener, and M. Lorenz. Burrows wheeler - alternatives to move to front. In *Data Compression Conference*, 2003.
- [22] Paolo Ferragina and Giovanni Manzini. Opportunistic data structures with applications. In *41st IEEE Symposium on Foundations of Computer Science (FOCS)*, 2000.
- [23] Paolo Ferragina and Giovanni Manzini. An experimental study of an compressed index. In *ACM-SIAM SODA*, 2001.
- [24] G. Gestri. *Teoria dell'informazione*. ETS, Pisa, 1991.
- [25] Raffaele Giancarlo and Marinella Sciortino. Optimal partitions of strings: A new class of burrows-wheeler compression algorithms. To be published, January 2003.
- [26] S. Grabowski. Text preprocessing for burrows-wheeler block sorting compression. In *VII Konferencja "Sieci i Systemy Informatyczne - teoria, projekty, wdrozenia"*, 1999.
- [27] R. Yugo Kartono Isal and Alistair Moffat. Parsing strategies for bwt compression. In *IEEE Data Compression Conference*, march 2001.
- [28] R. Yugo Kartono Isal and Alistair Moffat. Word-based block-sorting text compression. In *Proc. 24th Australasian Computer Science Conference*, february 2001.
- [29] R. Yugo Kartono Isal, Alistair Moffat, and Alwin C. H. Ngai. Enhanced word-based block-sorting text. In *Proc. 25th Australasian Computer Science Conference*, january 2002.

- [30] J. Kärkkäinen and P. Sanders. Simple linear work suffix array construction. *ICALP*, 2003.
- [31] D. K. Kim, J. S. Sim, H. Park, and K. Park. Linear-time construction of suffix arrays. *CPM*, 2003.
- [32] P. Ko and S. Aluru. Linear time construction of suffix arrays. *CPM*, 2003.
- [33] Stefan Kurtz and Bernhard Balkenhol. Space efficient linear time computation of the burrows and wheeler-transformation. In *Numbers, Information and Complexity*, pages 375–383. Kluwer Academic Publisher, 2000.
- [34] Stephan T. Lavavej. bwtzip: A linear-time portable research-grade universal data compressor. Web site: <http://stl.caltech.edu/bwtzip.html>.
- [35] U. Manber and G. Myers. A new method for on-line string searches. In *1st ACM-SIAM Symposium on Discrete Algorithms*, pages 319–327, 1990.
- [36] M. Maniscalco. A second modified run length encoding scheme for blocksort transformed data. Web site: <http://www.geocities.com/m99datacompression/papers/rle2.html>.
- [37] G. Manzini. The burrows-wheeler transform: Theory and practice. In *Springer Verlag Lecture Notes in Computer Science*, Proc. 24th Int. Symposium on Mathematical Foundations of Computer Science (MFCS), pages 34–47, 1999.
- [38] G. Manzini. An analysis of the burrows-wheeler transform. *Journal of the Association for Computing Machinery*, 48(3):407–430, May 2001.
- [39] G. Manzini and P. Ferragina. Engineering a lightweight suffix array construction algorithm. In *European Symposium on Algorithms*, 2002.
- [40] A. Moffat and A. I. Wirth. Can we do without ranks in burrows wheeler transform compression? In *Proc. IEEE Data Compression Conference*, 2001.
- [41] M. Nelson. Data compression with the burrows-wheeler transform. *Dr. Dobb's Journal of Software Tools*, 21(9):46–50, 1996.
- [42] S.Kurtz R. Giegerich. From ukkonen to mcreight and weiner: A unifying view of linear-time suffix tree construction. *Algorithmica*, 19:331–353, 1997.
- [43] 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.
- [44] M. Schindler. A fast block-sorting algorithm for lossless data compression. Technical report, Vienna University of Technology, 1996. Web Site: <http://www.compressconsult.com/szip/>.

- [45] David Scott. David's view of bwt (burrows wheeler transform) in compression before encryption
<http://bijective.dogma.net/compres9.htm>.
- [46] J. Seward. The bzip2 program
<http://sources.redhat.com/bzip2/>.
- [47] J. Seward. On the performance of bwt sorting algorithms. In *Data Compression Conference*, 2000.
- [48] Julian Seward. Space-time tradeoffs in the inverse b-w transform. In *Data Compression Conference*, pages 439–448, 2001.
- [49] E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.
- [50] D. Wheeler. Upgrading bred with multiple tables
<ftp://ftp.cl.cam.ac.uk/users/djw3/bred3.ps>, 1997.