The Burrows-Wheeler compression algorithm is even better than what you have thought

Shir Landau* Elad Verbin*

April 8, 2005

Abstract

The best compression algorithm today for English text is based on the Burrows-Wheeler transform. This algorithm (whose common implementation is bzip2) consists of the following three essential steps: 1) Obtain the Burrows-Wheeler transform of the text, 2) Convert the transform into a sequence of integers using the move-to-front algorithm, 3) Encode the integers using arithmetic code or any order-0 encoding (possibly with run length encoding).

In this paper we achieve a strong bound on the worst-case compression ratio of this algorithm, that is significantly better than bounds known to date and is obtained via simple analytical techniques. Specifically, for any input string s, and $\mu > 1$, the length of the compressed string is bounded by $\mu \cdot |s| H_k(s) + \log(\zeta(\mu)) \cdot |s| + g_k$ where H_k is the k-th order empirical entropy, g_k is a constant depending only on k and on the size of the alphabet, and $\zeta(\mu) = \frac{1}{1^\mu} + \frac{1}{2^\mu} + \ldots$ is the standard zeta function.

In fact we prove a stronger result: That this bound without the additive term g_k holds when we replace $H_k(s)$ by the sum of the logarithms of the integers obtain by the move-to-front encoding of the transform. This refined bound is tight and close to the actual compression achieved in practice. To obtain this result we prove a tight result on the compressibility of integer sequences, which is of independent interest.

1 Introduction

In 1994, Burrows and Wheeler [4] introduced the Burrows-Wheeler Transform, and two new lossless text-compression algorithms that are based on this transform. Following [13], we refer to these algorithms as BW0 and BW_{RL}. A well known implementation of these algorithms, known as bzip2 [18], is among the best compressors for English text, and is definitely the fastest among them. For example, bzip2 typically shrinks an English to about 20% of its original size while gzip only gets 26% (see Table 1 and also [1] for detailed results). In this paper we refine and tighten the analysis of BW0. For this purpose we introduce new techniques and statistical measures. We believe these may be useful in the analysis of other compression algorithms, and in predicting the performance of these algorithms in practice.

The algorithm BW0 compresses the input text s in three steps.

1. Compute the Burrows-Wheeler Transform, $\hat{s}$, of s. We elaborate on this stage shortly.[1]

*School of Computer Science, Tel Aviv University, Tel Aviv, Israel. email: {landaush, eladv}@post.tau.ac.il.

[1] For compatibility with other definitions, we actually need to compute the BWT of s in reversed order, that is from right to left. This will, of course, not change our results and does not effect the compression ratio significantly (see [7] for a discussion on this), so we will ignore this point from now on.

2. Transform $\hat{s}$ to a string of integers $\dot{s} = \text{MTF}(\hat{s})$ by using the move to front algorithm. This algorithm maintains the characters of the alphabet in a list and encodes the next character by its index in the list (see Sec. 2).

3. Encode the string $\dot{s}$ of integers by using an order-0 encoder, to obtain the final bit stream $\text{BW0}(s) = \text{ORDER0}(\dot{s})$. An order-0 encoder assigns a unique bit string to each integer independently of its context, such that we can decode the concatenation of these bit strings. Common order-0 encoders are Huffman codes or Arithmetic code.

The algorithm BW_{RL} performs an additional run-length encoding procedure between steps 2 and 3. See [4, 13] for more details on BW0 and BW_{RL}.

Next we define the Burrows-Wheeler Transform (BWT). Let n be the length of s. We obtain $\hat{s}$ as follows. Add a unique end-of-string symbol '\$' to s. Take all cyclic shifts of the string $s\$$ and place them in an $(n+1) \times (n+1)$ matrix. Sort the rows of this matrix in lexicographic order ('\$' is considered smaller than all other characters). The last column of this matrix, with the symbol '\$' omitted, is the Burrows-Wheeler Transform, $\hat{s}$. See an example in Fig. 1. Although it may not be obvious at first glance, BWT is an invertible transformation (one also needs to save the location of the symbol '\$' to do this.) Efficient methods exist for computing and inverting $\hat{s}$ in linear time (see for example [14]).

The BWT is effective for compression since in $\hat{s}$ characters with the same context[2] are consecutive. This means that if in the input text a reasonably small context tends to predict the character itself, then the string $\hat{s}$ will show local similarity – that is, symbols tend to recur at close vicinity. It is interesting to note that the Burrows-Wheeler Transform seems related, in some senses, to the Fourier Transform (intuitively speaking, both transform recurring motifs in the original text to spikes in the transformed text, and, more interestingly, they work even if the motifs in the text are only *approximately* recurring). For a discussion of the similarity of BWT to the Fourier Transform see [6].

Therefore, if s is say a text in English, we would expect $\hat{s}$ to be a string with symbols recurring at close vicinity. As a result $\dot{s} = \text{MTF}(\hat{s})$ is an integer string which we expect to be composed mainly of small numbers. (Note that by "integer string" we mean a string over an integer alphabet). Furthermore, the frequencies of the integers in $\dot{s}$ are skewed, and therefore it now makes sense to run an order-0 encoder on $\dot{s}$. This, of course, is an intuitive explanation as to why BW0 "should" work on *typical* inputs. As we discuss next, our work is in a worst-case setting, which means that we will give upper bounds that hold for *any* input. These upper bounds will be relative to statistics which measure how "well-behaved" our input string is. Part of the question which we try to address is which statistics actually capture the compressibility of the input text.

1.1 History and Motivation

It is well known that the zeroth order empirical entropy of a string s, $H_0(s)$, is a lower bound on the possible bits-per-character ratio of any order-0 compressor. Similarly, the k-th order empirical entropy of the string s, $H_k(s)$ (defined in Sec. 2) gives a lower bound on the possible bits-per-character ratio of any encoder that is allowed to use only the k-character context of character x in order to encode it. For this reason compression algorithms are traditionally measured against $H_k(s)$, for various values of k.

[2]The context of length k of a character is the string of length k that precedes it.

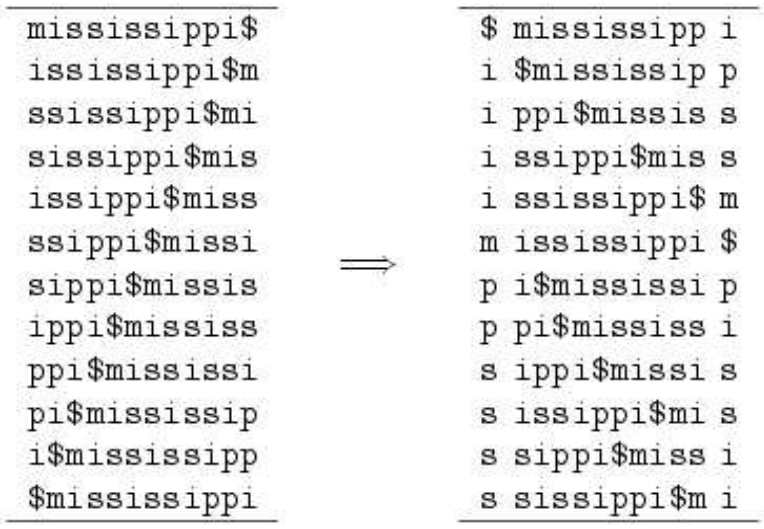

Figure 1: The Burrows-Wheeler transform for the string $s = mississippi$. The matrix on the right has the rows sorted in lexicographic order. $\hat{s}$ is the last column of the matrix, i.e. *ipssmpissii*, and we need to store the index of the character '$', i.e. 6, to be able to get the original string

In 1999, Manzini [13] presented the first worst-case upper bounds for the compression of BWT-based algorithms. He bounded the bit-length of the compressed text BW0(s) by the expression

$$8 \cdot nH_k(s) + 0.08 \cdot n + g'_k. \tag{1}$$

Here $k \geq 0$ is any number, and $g'_k = h^k(2h \log h + 9)$ is a constant that depends only on k and on the alphabet size. Manzini also bounded the bit-length of $\text{BW}_{RL}(s)$ by the expression $5 \cdot nH^*_k(s) + g''_k$. g''_k is some other constant that depends only on k and on the alphabet size, and H^*_k, which we will not define, is a statistic larger than H_k (see [13]).

In 2004, Ferragina, Giancarlo, Manzini and Sciortino [7] introduced a BWT-based compression booster. In their paper, they show how the compression booster can produce two different algorithms achieving $1 \cdot nH_k(s) + 1 \cdot n + g''_k$ and $2.5 \cdot nH^*_k(s) + g''_k$, respectively. The upper bounds of these algorithms are theoretically superior to those in [13]. However, when we implemented their algorithms, we got the results summarized in table 1. These empirical results show a discrepancy between the theoretical bounds and the actual compression ratios. This was the starting point of our research. We looked for tight bounds on the length of the compressed text, possibly in terms of statistics of the text that might be more appropriate than H_k.

1.2 Our Results

We tighten the analysis of BW0 and give a tradeoff result that shows that for any constant $\mu > 1$ and for any k, the length of the compressed text is upper-bounded by the expression

$$\mu \cdot nH_k(s) + (\log \zeta(\mu) + C_{\text{ORDER0}}) \cdot n + g_k. \tag{2}$$

Here C_{ORDER0} is a small constant, defined in Sec. 3, and $g_k = \mu 2k \log h + \mu h^k \cdot h \log h$. In particular, for $\mu = 1.5$ we obtain the bound $1.5 \cdot nH_k(s) + 1.5 \cdot n + 1.5g_k$. Our proof is simpler than that of Manzini in [13].

File Name	size	gzip	bzip2	BW0	Booster(HC)	Booster(RHC)
alice29.txt	152089	54181	43196	48915	74576	79946
asyoulik.txt	125179	48819	39569	44961	59924	61757
cp.html	24603	7965	7632	8726	16342	16342
fields.c	11150	3122	3039	3435	10235	10028
grammar.lsp	3721	1232	1283	1409	2297	2297
lcet10.txt	426754	144562	107648	127745	166043	177682
plrabn12.txt	481861	194551	145545	168311	172471	183855
xargs.1	4227	1748	1762	1841	2726	2726

Table 1: Results (in bytes) of running various compressors on the non-binary files from the Canterbury Corpus [1]. The gzip results are taken from [1]. BW0 is our implementation (in C++) of the BW0 algorithm (using Huffman encoding) as the order-0 compressor. Booster(HC) and Booster(RHC) are our implementation of the compression booster of [7]. [7] gives two suggested methods to implement it: One using the algorithm HC, and one using the algorithm RHC (the interested reader is referred to [7]).

The technique which we use to obtain this bound is even more interesting than the bound itself. We define a new natural statistic of a text which we call the "Local Entropy" (LE). This statistic was implicitly considered by Bentley et al. [3] as well as by Manzini [13]. Using two observations on the behavior of LE we bypass some of the technical hurdles in the analysis of [13].

Our analysis actually proves a considerably stronger result: That the size of the compressed text is bounded by

$$\mu \cdot n\mathrm{LE}(\hat{s}) + (\log \zeta(\mu) + C_{\mathrm{ORDER0}}) \cdot n \tag{3}$$

Empirically, this seems to give estimations which are quite close to the actual compression, as seen in Table 2.

File Name	size (bits)	$H_0(\dot{s})$ (bits)	$LE(\hat{s})$ (bits)	(3) (bits)	(2) (bits)	(1) (bits)
alice29.txt	1216712	386367	144247	396813	766940	2328219
asyoulik.txt	1001432	357203	140928	367874	683171	2141646
cp.html	196824	67010	26358	69857.6	105033.2	295714
fields.c	89200	24763	8855	25713	43379	119210
grammar.lsp	29768	9767	3807	10234	16054	45134
lcet10.txt	3414032	805841	357527	1021440	1967240	5867291
plrabn12.txt	3854888	1337475	528855	1391310	2464440	8198976
xargs.1	33816	13417	5571	13858	22317	64673

Table 2: Results of computing various statistics on the non-binary files from the Canterbury Corpus [1]. $H_0(\dot{s})$ gives the result of the algorithm BW0 assuming an optimal order-0 compressor. The difference between this and the column marked (3) shows that our bound (3) is quite tight in practice. The final three columns show the bounds given by the equations (3), (2), (1). It should be noted that in order to get the bound of (3) we needed to minimize the expression in (3) over μ. To get the bound of (2) and (1) we needed to calculate their value for all k and pick the best one.

In order to get our upper bounds we needed to solve a problem on compressing integer strings

in which the integers are typically small. From these results it also follows that no algorithm can have a worst-case performance better than that in Eq. (3) (up to the constant C_{ORDER0}). The result on compression of integer sequences is of independent interest. This is shown in Sec. 3

1.3 Related Work

A series of recent results on compression present *Compressed Text Indexes*. A compressed text index is a scheme that given a (typically large) text builds a representation of it. This representation allows very fast pattern matching queries and decompression of parts of the original text. In particular, the entire original text can be read back from its representation, thus eliminating the need to save the original text. The representation's size is typically much smaller than that of the original text. A Compressed Text Index is therefore simultaneously both a compression algorithm and an indexing data structure. Early progress on Compressed Text Indexes was made by Manzini and Ferragina in [15]. A recent result by Grossi, Gupta and Vitter [10] presents a Compressed Text Index that comes within additive lower-order terms of the order-k entropy of the input text. This result makes heavy use of data structures for indexable dictionaries by Raman, Raman, and Rao [17]. For more on Compressed Text Indexing, see [11, 15, 8].

We leave open the question of how our techniques can be applied to the subject of Compressed Text Indexing.

2 Preliminaries

Throughout the paper we assume $0 \log 0 = 0$, as customary. We will be careful with this when needed, for example in the proof of Thm. 3.2. All logarithms are in base 2 unless otherwise stated. We usually refer to a string s, which is a sequence of length $|s| = n$ whose elements are from the alphabet Σ. The alphabet $\Sigma = \{\sigma_0, \ldots, \sigma_{h-1}\}$ is of size $|\Sigma| = h$. However, sometimes (mainly in Sec. 3) s will be a string over the alphabet $[h] = \{0, \ldots, h-1\}$. This will be clear from the context. When we consider a compression algorithm A, we refer to $|A(s)|$ as the size *in bits* of the encoding $A(s)$. The function $\zeta(\mu)$ is the usual zeta function, $\zeta(\mu) = \frac{1}{1^\mu} + \frac{1}{2^\mu} + \ldots$, which we will need only in the range $\mu \in \{x | x > 1\}$. $\zeta_h(\mu)$ is this sum limited to the first h terms: $\zeta_h(\mu) = \frac{1}{1^\mu} + \ldots + \frac{1}{h^\mu}$.

The following definitions are taken from [13]. Let s be a string of length n and let n_i denote the number of occurrences of the symbol σ_i in s. The zeroth order empirical entropy of the string s is defined as

$$H_0(s) = \sum_{i=0}^{h-1} \frac{n_i}{n} \log \frac{n}{n_i}.$$

For any length-k word $w \in \Sigma^k$ let w_s denote the string consisting of the concatenation of the single characters following each occurrence of w in s. The value

$$H_k(s) = \frac{1}{n} \sum_{w \in \Sigma^k} |w_s| \, H_0(w_s)$$

is called the k-th order empirical entropy of the string s. In [13] these terms, as well as BWT, are discussed in more depth.

Our analysis does not use the definitions of H_k and BWT directly. Instead, it uses the fact, observed by Manzini in [13], that $H_k(s)$ can be put in terms of the H_0 of parts of $\hat{s}$. More specifically:

Proposition 2.1 ([13]) *There is a string $\tilde{\hat{s}}$ that is equal to $\hat{s}$ except that k characters are missing, and there is a partition of $\tilde{\hat{s}}$, $\tilde{\hat{s}} = s_1 \dots s_t$, with $t \leq h^k$, such that:*

$$|s|\, H_k(s) = \sum_{i=1}^{t} |s_i|\, H_0(s_i) \tag{4}$$

Now let us define the move to front (MTF) transformation, which was introduced in [3]. MTF encodes the symbol σ_i with an integer equal to the number of distinct symbols encountered since the previous occurrence of σ_i. More precisely, the encoding maintains a list of the symbols ordered by recency of occurrence. When the next symbol arrives, the encoder outputs its current rank and moves it to the front of the list. Therefore, a string over the alphabet Σ is transformed to a string over $[h]$ (note that the length of the string does not change). To completely determine the encoding we must specify the status of the recency list at the beginning of the procedure. We denote by MTF_π the algorithm in which the initial status of the recency list is given by the permutation π defined over Σ.

MTF has the property that if the input string has high local similarity, that is if symbols tend to recur at close vicinity, then the output string will consist mainly of small integers. This is formalized by the concept of the Local Entropy (LE) statistic, which we define next. LE was used implicity in [3, 13]. We use some properties of LE to get our results in Sec. 4. LE is defined as follows:

$$\text{LE}_\pi(s) = \sum_{i=1}^{n} \log(\text{MTF}_\pi(s)[i] + 1)$$

That is, LE is the sum of the logarithms of the move-to-front values plus 1. For example, for the string "aa" the MTF value of the second a is 0. That is the reason that we add the $+1$ in the logarithm – so that a repeating symbol will contribute 0 to the sum. Let us now define $\text{LE}_W(s) = max_\pi \text{LE}_\pi(s)$. This is the "worst-case" local entropy.[3] Analogously, MTF_W is a MTF in which the initial recency list is a π that maximizes $\text{LE}_\pi(s)$. We will sometimes write just LE instead of LE_W or LE_π when the initial permutation of the recency list is not significant. At times, this notation suffices because the difference between $\text{LE}_{\pi_1}(s)$ and $\text{LE}_{\pi_2}(s)$ is always $O(h \log h)$. Similarly, we may write MTF instead of MTF_W or MTF_π.

The intuition behind LE is that after running BWT and then MTF we get a string $\text{MTF}(\hat{s})$. This string consists of n numbers. Naively, we would expect to compress this string of integers using a number of bits close to the sum of their logarithms. Obviously, this doesn't really work since we have to distinguish where one number ends and another starts. This intuition, however, gives us the statistic LE which turns out to be very useful. For the sake of formality we define the statistic $\widehat{\text{LE}}(s) = \text{LE}(\hat{s})$.

The statistics $H_0(s)$ and $H_k(s)$ are normalized in the sense that they represent the *bits-per-character* rate attainable for compressing s. However, for our purposes it is most convenient to work with un-normalized statistics. Thus we define our new statistic LE to be un-normalized. We define the statistics $n \cdot H_0$ and $n \cdot H_k$ to be the un-normalized counterparts of the original statistics, i.e. $(n \cdot H_0)(s) = n \cdot H_0(s)$.

Let $f : \Sigma^* \to \mathbb{R}^+$ be an (un-normalized) statistic on strings, for example f can be $n \cdot H_k$, LE. Then a compression algorithm A is called (μ, C)-f-competitive if for every string s it holds that $|A(s)| \leq \mu f(s) + Cn + o(n)$, where $o(n)$ denotes a function $g(n)$ such that $\lim_{n\to\infty} \frac{g(n)}{n} = 0$.

[3] LE_W is defined to make the presentation more elegant later on, but one could use $\text{LE}_\pi(s)$ with π a constant permutation, and the analysis will work much the same.

We will often use the following inequality, derived from Jensen's inequality:

Lemma 2.2 *For any $k \geq 1$, $x_1, \ldots, x_k > 0$ and $y_1, \ldots, y_k > 0$ it holds that:*

$$\sum_{i=1}^{k} y_i \log x_i \leq \left(\sum_{i=1}^{k} y_i \right) \cdot \log \frac{\sum_{i=1}^{k} x_i y_i}{\sum_{i=1}^{k} y_i} \tag{5}$$

This inequality roughly means that a sum of logarithms is maximized when all of the logarithms are equal. For integers y_i this is a version of the inequality of the means.

3 Optimal Results on Compression vs. SL

In this section we look at a string s of length n over the alphabet $[h]$. Let us define the *sum of logarithms statistic*: $\text{SL}(s) = \sum_{i=1}^{n} \log(s[i] + 1)$. We will show that in a strong sense the best SL-competitive compression algorithm is an order-0 compressor. At the end of the section we will show how to get analogous results regarding LE-competitive and $\widehat{\text{LE}}$-competitive compression algorithms.

The problem we deal with in this section is related to the problem of universal encoding of integers. In the problem of universal encoding of integers [5, 3] the goal is to find a prefix-free encoding for integers, $U : \mathbb{Z}^+ \to \{0,1\}^*$, such that for every $x \geq 0$, $|U(x)| \leq \mu \log(x+1) + C$. A particularly nice solution for this is the Fibonacci Encoding [2, 9], for which $\mu = \log_\phi 2$, $C = 1 + log_\phi \sqrt{5} \simeq 2.6723$. An additional solution for this problem was proposed by Elias [5]. This is an optimal solution, in the sense described in [12]. For more information on universal encoding of integers see the (somewhat outdated) survey paper [12].

It can be seen that a universal encoding scheme with parameters μ, C easily gives a (μ, C)-SL-competitive compressor. However, in this section we get a better competitive ratio, which cannot be achieved by using the above method. This is possible because we deal with a relaxed version of the problem: Our goal is to encode a string which is actually a long sequence of integers taken from a fixed alphabet. We are also allowed to incur an $o(n)$ additive term.

Throughout the paper we refer to an algorithm ORDER0. By this we mean any order-0 algorithm, which is assumed to be a $(1, C_{\text{ORDER0}})$-$n \cdot H_0$-competitive algorithm. For example, $C_{\text{HUFFMAN}} = 1$ and $C_{\text{ARITHMETIC}} \approx 10^{-2}$ [19, 16].

3.1 An optimal (μ, C)-SL-competitive algorithm

We show, using a technique based on Lemma 2.2, that the algorithm ORDER0 is $(\mu, \log \zeta(\mu) + C_{\text{ORDER0}})$-SL-competitive *for any* $\mu > 1$. In fact, we prove a somewhat stronger theorem:

Theorem 3.1 *For any constant $\mu > 0$ it holds that the algorithm* ORDER0 *is $(\mu, \log \zeta_h(\mu) + C_{\text{ORDER0}})$-SL-competitive.*

Proof. Let s be a string of length n over alphabet $[h]$. We need to prove that for any constant $\mu > 0$:

$$nH_0(s) \leq \mu \text{SL}(s) + n \log \zeta_h(\mu) \tag{6}$$

From the definition of H_0 it follows that $nH_0(s) = \sum_{i=0}^{h-1} n_i \log \frac{n}{n_i}$, and from the definition of SL we get that $\text{SL}(s) = \sum_{j=1}^{n} \log(s[j]+1) = \sum_{i=0}^{h-1} n_i \log(i+1)$. So, (6) is equivalent to:

$$\sum_{i=0}^{h-1} n_i \log \frac{n}{n_i} \leq \mu \sum_{i=0}^{h-1} n_i \log(i+1) + n \log \zeta_h(\mu) \tag{7}$$

Pushing the μ into the logarithm and moving terms around we get that (7) is equivalent to:

$$\sum_{i=0}^{h-1} n_i \log \frac{n}{n_i (i+1)^{\mu}} \leq n \log \zeta_h(\mu) \tag{8}$$

Defining $p_i = \frac{n_i}{n}$ and dividing the two sides of the inequality by n we get that (8) is equivalent to:

$$\sum_{i=0}^{h-1} p_i \log \frac{1}{p_i (i+1)^{\mu}} \leq \log \zeta_h(\mu)$$

Using Lemma 2.2,

$$\sum_{i=0}^{h-1} p_i \log \frac{1}{p_i (i+1)^{\mu}} = \sum_{\substack{0 \leq i \leq h-1 \\ p_i \neq 0}} p_i \log \frac{1}{p_i (i+1)^{\mu}} \leq \log \left(\sum_{\substack{0 \leq i \leq h-1 \\ p_i \neq 0}} p_i \frac{1}{p_i (i+1)^{\mu}} \right) =$$

$$= \log \left(\sum_{\substack{0 \leq i \leq h-1 \\ p_i \neq 0}} \frac{1}{(i+1)^{\mu}} \right) \leq \log \zeta_h(\mu)$$

□

In particular we get:

Corollary 3.2 *For any constant $\mu > 1$ it holds that the algorithm* ORDER0 *is $(\mu, \log \zeta(\mu) + C_{\text{ORDER0}})$-*SL*-competitive.*

3.2 A lower bound for SL-Competitive compression

In Sec. 3.1 we have seen that for any $\mu > 0$ there exists a $(\mu, \log \zeta_h(\mu) + C_{\text{ORDER0}})$-SL-competitive algorithm. In this section we show that this is a tight bound in a strong sense. That is, besides proving that no algorithms achieves a better competitive ratio, we prove that this holds separately for any values of μ and h. Note however, that the lower bounds that we get in this section do not include the constant C_{ORDER0}.

Theorem 3.3 *Let $\mu > 0$ be some constant. Then there is no constant $C < \log \zeta_h(\mu)$ such that there exists a (μ, C)-*SL*-competitive algorithm*

The proof of this theorem is deferred to Appendix A.

By setting a large enough alphabet size in the proof of Thm. 3.3, we get the following corollaries:

Corollary 3.4 *Let $\mu > 1$ be some constant. Then there is no constant $C < \log \zeta(\mu)$ such that there exists a (μ, C)-*SL*-competitive algorithm*

Corollary 3.5 *There is no $(1, C)$-*SL*-competitive algorithm, for any $C \in \mathbb{R}$*

3.3 Analogous Results vs. $\widehat{\text{LE}}$

By performing the inverse transformations MTF_π^{-1} and BWT^{-1} on both the term SL and the term ORDER0 in Thms. 3.1 and 3.3 we get analogous results vs. $\widehat{\text{LE}}$:

Corollary 3.6 *For any constant $\mu > 0$ it holds that the algorithm* BW0 *is $(\mu, \log \zeta_h(\mu) + C_{\text{ORDER0}})$-$\widehat{\text{LE}}$-competitive.*

Corollary 3.7 *Let $\mu > 0$ be some constant. Then there is no constant $C < \log \zeta_h(\mu)$ such that there exists a (μ, C)-$\widehat{\text{LE}}$-competitive algorithm*

Proof Sketch. The only complication is that in order to refer to the inverse transformations MTF_π^{-1} and BWT^{-1} we need to note that both MTF_π and BWT are indeed invertible and have inverting procedures. For BWT to be invertible one needs to consider it without omitting the end-of-string symbol. This, however, poses no obstacle as it cannot change the compressibility of the sequence by more than $\log n = o(n)$ bits, for keeping the position of the end-of-string symbol. □

Further analogous statements about LE as well as LE can be derived similarly from the results of this Section.

4 The Entropy Hierarchy

In this section we will show that the statistics H_k and $\widehat{\text{LE}}$ conveniently form a hierarchy, which allows us to percolate upper bounds down and lower bounds up. Specifically, we will show that for each k

$$\widehat{\text{LE}}(s) \leq nH_k(s) + O(1)$$

So the hierarchy is as follows:

$$\widehat{\text{LE}}(s) \leq \ldots \leq nH_2(s) + O(1) \leq nH_1(s) + O(1) \leq nH_0(s) + O(1) \tag{9}$$

Thus any (μ, C)-$\widehat{\text{LE}}$-competitive algorithm is also (μ, C)-$n \cdot H_k$-competitive. To get this we need to prove two properties of LE_W: that it is at most $nH_0 + O(1)$, and that it is convex (in a sense which we will define).

4.1 Some properties of LE

Define $\text{MTF}_{ignorefirst}(s)$ to be a string which is identical to $\text{MTF}_\pi(s)$ except that we omit the characters representing the first occurrence of each symbol (so $\text{MTF}_{ignorefirst}(s)$ is of length less than n). Note that in this case when we perform the move-to-front transformation the initial status of the MTF recency list is not significant. Similarly, define $\text{LE}_{ignorefirst}(s) = \sum_i \log(\text{MTF}_{ignorefirst}(s)[i]+1)$.

The following is a theorem from Bentley et al. [3]:

Theorem 4.1 ([3]) $\text{LE}_{ignorefirst}(s) \leq n \cdot H_0(s)$.

The proof of Thm. 4.1 can be found in Appendix B. Let us show a corollary of this Theorem:

Lemma 4.2 $\text{LE}_W(s) \leq n \cdot H_0(s) + h \log h$.

Proof. $\text{LE}_W(s)$ is equal to $\text{LE}_{ignorefirst}(s)$ plus the contribution of the first occurrence of each symbol. The number of such contributions is at most h, and each such contribution is bounded by $\log h$, and so we get $\text{LE}_W(s) \leq \text{LE}_{ignorefirst}(s) + h \log h \leq n \cdot H_0(s) + h \log h$, as needed. □

In addition, we need the following lemma about LE_W, whose proof we omit:

Lemma 4.3 *For a string s of length n and a string s' that we get by deleting exactly one character from s it holds that $\text{LE}_W(s) \leq \text{LE}_W(s') + 2 \log h$.*

Now let's prove that LE_W is a convex statistic:

Lemma 4.4 (LE_W is a convex statistic) *For $s = s_1 \dots s_t$ it holds that $\text{LE}_W(s) \leq \sum_i \text{LE}_W(s_i)$*

Proof. The intuition is that the MTF_π encoding has a locality property such that if you stop it in the middle and start again from this point using a different recency list then you make little profit if any. Here is the formal proof of the Lemma: From the definition of LE_W we get that $\text{LE}_W(s) = \sum_{j=1}^{n} \log(\text{MTF}_{\pi_1}(s)[j] + 1)$ for a worst-case permutation π_1. Let us look at the recency list π_i that we use when we begin the $\text{LE}_W(s)$ run on sub-string s_i. Each of the summands of $\sum_i \text{LE}_W(s_i)$ is calculated with a worst-case permutation, which must be at least as bad as π_i, and thus we are finished. □

4.2 The Hierarchy Result

Theorem 4.5 *For any $k \geq 0$ and any string s,*

$$|s|\, H_k(s) \geq \text{LE}_W(\hat{s}) - 2k \log h - h^k \cdot h \log h$$

Proof. Recall Proposition 2.1: There must be a string $\tilde{\hat{s}}$ that is equal to $\hat{s}$ except that k characters are missing. and there is a partition of $\tilde{\hat{s}}$, $\tilde{\hat{s}} = s_1 \dots s_t$, such that $t \leq h^k$ and

$$|s|\, H_k(s) = \sum_{i=1}^{t} |s_i|\, H_0(s_i) \tag{10}$$

Observe that using the convexity of LE_W (Lemma 4.4) and using the relation of LE_W to nH_0 (Lemma 4.2) we have

$$\text{LE}_W(\tilde{\hat{s}}) \leq \sum_{i=1}^{t} |s_i|\, H_0(s_i) + th \log h \tag{11}$$

Using Lemma 4.3 we get

$$\text{LE}_W(\hat{s}) - 2k \log h \leq \text{LE}_W(\tilde{\hat{s}}) \tag{12}$$

From (10), (11) and (12) we get

$$\text{LE}_W(\hat{s}) - 2k \log h - th \log h \leq |s|\, H_k(s)$$

And using $t \leq h^k$ we are finished. □

5 Conclusions and Open Problems

Using Corollary 3.2 and Thm. 3.1 together with Thm. 4.5 gives:

Corollary 5.1 *For any $k \geq 0$ and for any constant $\mu > 1$ it holds that the algorithm* BW0 *is $(\mu, \log \zeta(\mu) + C_{\text{ORDER0}})$-$n \cdot H_k$-competitive*

Corollary 5.2 *For any $k \geq 0$ and for any constant $\mu > 0$ it holds that the algorithm* BW0 *is $(\mu, \log \zeta_h(\mu) + C_{\text{ORDER0}})$-$n \cdot H_k$-competitive (on strings from an alphabet of size h).*

We have also shown that our analysis vs. $\widehat{\text{LE}}$ is tight. That is, we may get better results vs. $n \cdot H_k$, but we cannot get better worst-case results vs. $\widehat{\text{LE}}$, neither with this nor with any other algorithm (this statement is valid, of course, up to the constant C_{ORDER0}, and notwithstanding getting better lower-order terms in the upper bound).

We leave the following idea for further research: In this paper we prove that the algorithm BW0 is roughly $(\mu, \log \zeta(\mu))$-$\widehat{\text{LE}}$-competitive. In work-in-progress, We also know how to show that BW0 is quite far from being $(1, 0)$-H_k-competitive. On the other hand, using the compression booster, [7] achieve results close to being $(1, 0)$-H_k-competitive, but we conjecture that they are very far from being $(\mu, \log \zeta(\mu))$-$\widehat{\text{LE}}$-competitive. So, a natural question to ask is whether there is an algorithm that in a deep way achieves both ratios. Of course, the algorithm that just performs both algorithms and selects the best compression achieves both ratios, but we would like an algorithm that gets this because of a deep reason which makes it sidestep the natural inefficiencies of both algorithms. We think that an "ultimative" algorithm such as this will beat all BWT-based algorithms known so far.

6 Acknowledgments

We would like to thank Nir Markus for writing the BW0 and COMPRESSION BOOSTER computer programs with Shir Landau, as this gave us motivation to begin our research. We would like to thank Gadi Landau for referring us to [7]. Elad Verbin would like to thank Adi Avidor for some lovely Friday discussions.

References

[1] The canterbury corpus. http://corpus.canterbury.ac.nz.

[2] Alberto Apostolico and Aviezri S. Fraenkel. Robust transmission of unbounded strings using fibonacci representations. *IEEE Transactions on Information Theory*, 33(2):238–245, 1987.

[3] Jon Louis Bentley, Daniel D. Sleator, Robert E. Tarjan, and Victor K. Wei. A locally adaptive data compression scheme. *Communications of the ACM*, 29(4):320–330, 1986.

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

[5] P. Elias. Universal codeword sets and representation of the integers. *IEEE Trans. on Information Theory*, 21(2):194–203, 1975.

[6] Peter Fenwick. Reflections on the burrows wheeler transform. Technical Report 172, Department of Computer Science, The University of Auckland, Auckland, New Zealand, 2004.

[7] P. Ferragina, R. Giancarlo, G. Manzini, and M. Sciortino. Boosting textual compression in optimal linear time. *Journal of the ACM*, to appear.

[8] P. Ferragina, G. Manzini, V. Mākinen, and G. Navarro. An alphabet friendly fm-index. In *Proc. 11th Symposium on String Processing and Information Retrieval (SPIRE '04)*, pages 150–160, 2004.

[9] Aviezri S. Fraenkel and Shmuel T. Klein. Robust universal complete codes for transmission and compression. *Discrete Applied Mathematics*, 64(1):31–55, 1996.

[10] Roberto Grossi, Ankur Gupta, and Jeffrey Scott Vitter. High-order entropy-compressed text indexes. In *SODA '03: Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 841–850, 2003.

[11] Roberto Grossi, Ankur Gupta, and Jeffrey Scott Vitter. When indexing equals compression: experiments with compressing suffix arrays and applications. In *SODA '04: Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 636–645, 2004.

[12] Debra A. Lelewer and Daniel S. Hirschberg. Data compression. *ACM Computing Surveys*, 19(3):261–296, 1987.

[13] G. Manzini. An analysis of the burrows-wheeler transform. *Journal of the ACM*, 48(3):407–430, 2001.

[14] G. Manzini and P. Ferragina. Engineering a lightweight suffix array construction algorithm. *Algorithmica*, 40:33–50, 2004.

[15] G. Manzini and P. Ferragina. On compressing and indexing data. *Journal of the ACM*, accepted for publication.

[16] Alistair Moffat, Radford M. Neal, and Ian H. Witten. Arithmetic coding revisited. *ACM Trans. Inf. Syst.*, 16(3):256–294, 1998.

[17] Rajeev Raman, Venkatesh Raman, and S. Srinivasa Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In *SODA '02: Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 233–242, 2002.

[18] Julian Seward. bzip2, a program and library for data compression. http://www.bzip.org/.

[19] Ian H. Witten, Radford M. Neal, and John G. Cleary. Arithmetic coding for data compression. *Communications of the ACM*, 30(6):520–540, 1987.

A Proof of Thm. 3.3

Let us prove Thm. 3.3:

Proof. The proof will be similar to the upper bound proof of Thm. 3.1, and will essentially prove that the upper bound is tight, via a counting argument that will work *for any algorithm*. In order

to get an information-theoretic lower bound, we need to develop the proper counting framework. Our biggest technical difficulty will be to deal with the fact that a string must have an integral number of occurrences of each symbol (the upper bounds work in any case, but to get the lower bounds we have to work harder). We will first ignore this difficulty, and then explain how to deal with it.

First let us fix the parameter μ to be some value in $(0, \infty)$. Now pick some constant $\epsilon > 0$ and suppose in contradiction that there is a compression algorithm A which is $(\mu, \log \zeta_h(\mu) - \epsilon)$-SL-competitive. Let $\alpha_i = n \cdot \frac{1}{\zeta_h(\mu) \cdot (i+1)^\mu}$ for $i \in [h]$. Assume for now that α_i are integers. Let $S(n)$ be the set of strings where integer i appears α_i times. Let $L(n) = \sum_{i=0}^{h-1} \log(i+1) \cdot \alpha_i$ and $N(n) = \frac{n!}{\alpha_0! \cdot \ldots \cdot \alpha_{h-1}!}$. Note that for each $s \in S(n)$, $|s| = n$, $\text{SL}(s) = L(n)$ and that $card(S(n)) = N(n)$.

Using Stirling's approximation $n! = (1 + o(1))\sqrt{2\pi n}\left(\frac{n}{e}\right)^n$ and taking n large enough so that

$$(1/2)\sqrt{2\pi n}\left(\frac{n}{e}\right)^n \leq n! \leq (3/2)\sqrt{2\pi n}\left(\frac{n}{e}\right)^n$$

we get:

$$\begin{aligned}
\log N(n) &\geq \log \frac{(1/2)\sqrt{2\pi n}\,(n/e)^n}{(3/2)^h \prod_{i=0}^{h-1} \sqrt{2\pi\alpha_i}\,(\alpha_i/e)^{\alpha_i}} = \\
&= \log \frac{(1/2)\sqrt{2\pi n}}{(3/2)^h \prod_{i=0}^{h-1} \sqrt{2\pi\alpha_i}} + n \log n - \sum_{i=0}^{h-1} \alpha_i \log \alpha_i \geq \\
&\geq -O(1) - h \log(2\pi n) + \sum_{i=0}^{h-1} \alpha_i \log(n/\alpha_i) \geq \\
&\geq -O(\log n) + \sum_{i=0}^{h-1} \alpha_i \log(n/\alpha_i) \qquad (13)
\end{aligned}$$

Using standard information-theoretic arguments, our algorithm A must compress at least one of the strings in $S(n)$ to at least $\log N(n)$ bits. So, from our assumption that A is $(\mu, \log \zeta_h(\mu) - \epsilon)$-SL-competitive we get

$$\log N(n) \leq \mu L(n) + n(\log \zeta_h(\mu) - \epsilon + o(1))$$

Which gives

$$\log N(n) - \mu L(n) \leq n(\log \zeta_h(\mu) - \epsilon + o(1)) \qquad (14)$$

On the other hand,

$$\begin{aligned}\log N(n) - \mu L(n) &= \log N(n) - \mu \sum_{i=0}^{h-1} \log(i+1)\cdot\alpha_i \geq \\ &\geq -O(\log n) + \sum_{i=0}^{h-1} \alpha_i \log(n/\alpha_i) - \mu \sum_{i=0}^{h-1} \log(i+1)\cdot\alpha_i = \\ &= -O(\log n) + \sum_{i=0}^{h-1} \alpha_i \log \frac{n}{\alpha_i (i+1)^\mu} = \\ &= -O(\log n) + \sum_{i=0}^{h-1} \alpha_i \log \zeta_h(\mu) = \\ &= -O(\log n) + n \log \zeta_h(\mu) \end{aligned} \tag{15}$$

From inequalities (14) and (15) we get

$$-O(\log n) + n \log \zeta_h(\mu) \leq n(\log \zeta_h(\mu) - \epsilon + o(1))$$

Which for large enough values of n gives a contradiction.

Now let us see how to handle the fact that the α_i's must be integers. Define for $i \in \{1, \ldots h-1\}$, $\alpha'_i = \left\lfloor n \cdot \frac{1}{\zeta_h(\mu)\cdot(i+1)^\mu} \right\rfloor$, and push the excess into α'_0: $\alpha'_0 = n - \sum_{i=0}^{h-1} \alpha'_i$. By rounding in this specific way we have actually decreased the sum of logarithms, so $\text{SL}(s) \leq L(n)$. Inequality (13) still holds because the rounding makes α_i and α'_i differ by at most $\pm h$, which contributes only an additional $-O(\log n)$ factor to (13). □

B Proof of Thm. 4.1

Let us prove Thm. 4.1:

Proof. Let us look separately at the contributions of the h different symbols to $A := \text{LE}_{ignorefirst}(s)$. For a symbol $\sigma \in \Sigma$ denote by n_σ the number of its occurrences in s, and let us look at its contribution to $\text{LE}_{ignorefirst}(s)$:[4]

$$A_\sigma = \sum_{i:s[i]=\sigma} \log(\text{MTF}_{ignorefirst}(s)[i] + 1)$$

It is easy to see that

$$\sum_{i:s[i]=\sigma} (\text{MTF}_{ignorefirst}(s)[i] + 1) \leq n$$

and using Lemma 2.2 we get

$$A_\sigma \leq n_\sigma \log \frac{\sum_{i:s[i]=\sigma}(\text{MTF}_{ignorefirst}(s)[i]+1)}{n_\sigma} \leq n_\sigma \log \frac{n}{n_\sigma}$$

[4]Note that for the sake of convenience, in the following equations we are disregarding the fact that some elements of $\text{MTF}_{ignorefirst}(s)$ are in a shifted position relative to the characters of s that they represent because the representations of the first appearances of symbols are omitted.

So,

$$A = \sum_{\sigma} A_\sigma \leq \sum_{\sigma} n_\sigma \log \frac{n}{n_\sigma} = nH_0(s)$$

as needed. □