Dipartimento di Informatica
Università del Piemonte Orientale "A. Avogadro"
Via Bellini 25/G, 15100 Alessandria
http://www.di.unipmn.it

Move-to-Front, Distance Coding, and Inversion Frequencies Revisited

Authors: T. Gagie (travis@mfn.unipmn.it) G. Manzini (manzini@mfn.unipmn.it)

TECHNICAL REPORT TR-INF-2008-03-02-UNIPMN
(March 2008)

The University of Piemonte Orientale Department of Computer Science Research Technical Reports are available via WWW at URL `http://www.di.mfn.unipmn.it/`.
Plain-text abstracts organized by year are available in the directory

Recent Titles from the TR-INF-UNIPMN Technical Report Series

2008-01 *Space-Conscious Data Indexing and Compression in a Streaming Model*, Ferragina, P., Gagie, T., Manzini, G., February 2008.

2007-05 *Scheduling Algorithms for Multiple Bag-of-Task Applications on Desktop Grids: a Knowledge-Free Approach*, Canonico, M., Anglano, C., December 2007.

2007-04 *Verifying the Conformance of Agents with Multiparty Protocols*, Giordano, L., Martelli, A., November 2007.

2007-03 *A fuzzy approach to similarity in Case-Based Reasoning suitable to SQL implementation*, Portinale, L., Montani, S., October 2007.

2007-02 *Space-conscious compression*, Gagie, T., Manzini, G., June 2007.

2007-01 *Markov Decision Petri Net and Markov Decision Well-formed Net Formalisms*, Beccuti, M., Franceschinis, G., Haddad, S., February 2007.

2006-03 *New challenges in network reliability analysis*, Bobbio, A., Ferraris, C., Terruggia, R., November 2006.

2006-03 *The Engineering of a Compression Boosting Library: Theory vs Practice in BWT compression*, Ferragina, P., Giancarlo, R., Manzini, G., June 2006.

2006-02 *A Case-Based Architecture for Temporal Abstraction Configuration and Processing*, Portinale, L., Montani, S., Bottrighi, A., Leonardi, G., Juarez, J., May 2006.

2006-01 *The Draw-Net Modeling System: a framework for the design and the solution of single-formalism and multi-formalism models*, Gribaudo, M., Codetta-Raiteri, D., Franceschinis, G., January 2006.

2005-06 *Compressing and Searching XML Data Via Two Zips*, Ferragina, P., Luccio, F., Manzini, G., Muthukrishnan, S., December 2005.

2005-05 *Policy Based Anonymous Channel*, Egidi, L., Porcelli, G., November 2005.

2005-04 *An Audio-Video Summarization Scheme Based on Audio and Video Analysis*, Furini, M., Ghini, V., October 2005.

2005-03 *Achieving Self-Healing in Autonomic Software Systems: a case-based reasoning approach*, Anglano, C., Montani, S., October 2005.

2005-02 *DBNet, a tool to convert Dynamic Fault Trees to Dynamic Bayesian Networks*, Montani, S., Portinale, L., Bobbio, A., Varesio, M., Codetta-Raiteri, D., August 2005.

2005-01 *Bayesan Networks in Reliability*, Langseth, H., Portinale, L., April 2005.

2004-08 *Modelling a Secure Agent with Team Automata*, Egidi, L., Petrocchi, M., July 2004.

Move-to-Front, Distance Coding, and Inversion Frequencies Revisited*

Travis Gagie† Giovanni Manzini†

Abstract

Move-to-Front, Distance Coding and Inversion Frequencies are three somewhat related techniques used to process the output of the Burrows-Wheeler Transform. In this paper we analyze these techniques from the point of view of how effective they are in the task of compressing low-entropy strings, that is, strings which have many regularities and are therefore highly compressible. This is a non-trivial task since many compressors have non-constant overheads that become non-negligible when the input string is highly compressible.

Because of the properties of the Burrows-Wheeler transform, being *locally optimal* ensures an algorithm compresses low-entropy strings effectively. Informally, local optimality implies that an algorithm is able to effectively compress *an arbitrary partition* of the input string. We show that in their original formulation neither Move-to-Front, nor Distance Coding, nor Inversion Frequencies is locally optimal. Then, we describe simple variants of the above algorithms which are locally optimal. To achieve local optimality with Move-to-Front it suffices to combine it with Run Length Encoding. To achieve local optimality with Distance Coding and Inversion Frequencies we use a novel "escape and re-enter" strategy.

1 Introduction

Burrows-Wheeler compression is important in itself and as a key component of compressed full-text indices [17]. It is therefore not surprising that the theory and practice of Burrows-Wheeler compression has recently received much attention [6, 8, 9, 11, 12, 13, 14, 16, 15].

In the original Burrows-Wheeler compression algorithm [4] the output of the Burrows-Wheeler Transform (bwt from now on) is processed by Move-to-Front encoding followed by an order-0 encoder. Recently, [14] has provided a simple and elegant analysis of this algorithm and of the variant in which Move-to-Front encoding is replaced by Distance Coding. This analysis improves the previous one in [16] and provides new insight on the Move-to-Front and Distance Coding procedures. In [14] the output size of Burrows-Wheeler algorithms on input s is bounded by $\mu|s|H_k(s) + \Theta(|s|)$ for any $\mu > 1$, where H_k is the k-th order empirical entropy (more details in Sect. 2). We point out that this is a significant bound only as long as the string s is not too compressible. For highly compressible strings, for example $s = ab^n$, we have $|s|H_k(s) = O(\log |s|)$ so the $\Theta(|s|)$ term in the above bound becomes the dominant term. In this case, the bound tells one nothing about how close the compression ratio is to the entropy of the input string.

The above considerations suggest that further work is required on the problem of designing algorithms with an output size bounded by $\lambda|s|H_k(s) + \Theta(1)$ where $\lambda > 1$ is a constant independent of k, $|s|$, and of the alphabet size. We call this an "entropy-only" bound. An algorithm achieving an "entropy-only" bound guarantees that even for highly compressible strings the compression ratio will be proportional to the entropy of the input string. Note that the capability of achieving "entropy-only" bounds is one of

*Partially supported by Italian MIUR Italy-Israel FIRB Project "Pattern Discovery Algorithms in Discrete Structures, with Applications to Bioinformatics". A preliminary version of this work has appeared in the *Proceedings of the 18th Symposium on Combinatorial Pattern Matching (CPM '07)*.

†Dipartimento di Informatica, Università del Piemonte Orientale, Alessandria, Italy. {travis,manzini}@mfn.unipmn.it.

the features that differentiate Burrows-Wheeler compression algorithms from the family of Lempel-Ziv compressors [16].

As observed in [16], the key property for achieving "entropy-only" bounds is that the compression algorithm used to process the bwt be *locally optimal*. Informally, a locally optimal algorithm has the property of being able to compress efficiently *an arbitrary partition* of the input string (see Definition 2). Starting from these premises, in this paper we prove the following results:

1. We analyze Move-to-Front (Mtf), Distance Coding (Dc), and Inversion Frequencies Coding (If)—which is another popular variant of Move-to-Front encoding. We prove that in their original formulation none of these algorithms is locally optimal.
2. We describe simple variants of the above algorithms which are locally optimal. Therefore, when used together with the bwt these variants achieve "entropy-only" bounds. To achieve local optimality with Mtf it suffices to combine it with Run Length Encoding. To achieve local optimality with Dc and If we use an "escape and re-enter" technique.
3. The procedures Mtf, Dc and If all output sequences of positive integers. One can encode these sequences either using a prefix-free encoding [8] or feed the whole sequence to an Order-0 encoder [4]. Taking advantage of previous results of Burrows-Wheeler compression [14, 15], we are able to provide a simple analysis for both options.

In terms of "entropy-only" bounds our best result is the bound $(2.69+C_0)|s|H_k^*(s)+\log|s|+\Theta(1)$ bits where C_0 is a constant characteristic of the Order-0 final encoder (for example $C_0 \approx .01$ for arithmetic coding). This bound is achieved by our variant of Distance Coding and it improves the previously known "entropy-only" bound of $(5+3C_0)|s|H_k^*(s)+\log|s|+\Theta(1)$ established in [16] for Mtf combined with run length encoding. At the same time we prove that, under mild assumptions, no compression algorithm can achieve an "entropy-only" bound of the form $\lambda|s|H_0^*(s)+\Theta(1)$ for a constant $\lambda \leq 2$.

2 Notation and Background

Let s be a string drawn from the alphabet $\Sigma = \{\sigma_1, \ldots, \sigma_h\}$. For $i = 1, \ldots, |s|$ we write $s[i]$ to denote the i-th character of s. For each $\sigma_i \in \Sigma$, let n_i be the number of occurrences of σ_i in s. The 0-*th order empirical entropy* of the string s is defined as[1] $H_0(s) = -\sum_{i=1}^{h}(n_i/|s|)\log(n_i/|s|)$. It is well known that H_0 is the maximum compression we can achieve using a fixed codeword for each alphabet symbol. The following definition captures the notion of a compressor which is able to achieve H_0 up to a constant overhead per symbol and an additional overhead depending on the alphabet size.

Definition 1 *An algorithm* A *is an order-0 algorithm if for any input string s we have*

$$|\mathsf{A}(s)| \leq |s|H_0(s) + C_0|s| + O(h\log h)\,.$$

where $h = |\Sigma|$. The parameter C_0 is the per character overhead of A. ∎

Examples of order-0 algorithms are Huffman coding, for which $C_0 = 1$, and Arithmetic coding, for which in principle the overhead per symbol can be made arbitrarily small (for typical implementations it is $C_0 \approx .01$). It is well known that we can achieve a compression ratio better than $H_0(s)$ if the codeword we use for each symbol depends on the k symbols preceding it. In this case, the maximum compression is bounded by the k-th order entropy $H_k(s)$ (see [16] for the formal definition).

In [14] the authors analyze the original Burrows-Wheeler compressor [4] in which the output of the bwt is processed by Mtf followed by an order-0 algorithm and they prove a bound of the form

$$\mu|s|H_k(s) + (\log(\zeta(\mu)) + C_0)|s| + \log|s| + O\Big(h^{k+1}\log h\Big) \tag{1}$$

[1] In the following log means $\log_2$ and ln denotes the natural logarithm. We assume $0\log 0 = 0$.

where $\zeta(\mu) = \sum_{j>0} j^{-\mu}$ is the Riemann zeta function and C_0 is the per symbol overhead of the order-0 algorithm. The above bound holds simultaneously for any $\mu > 1$ and $k \geq 0$. This means we can get arbitrarily close to the k-th order entropy for any $k \geq 0$. Unfortunately, in (1) there is also a $\Theta(|s|)$ term which becomes dominant when s is highly compressible. For example, for $s = \sigma_1\sigma_2^n$ we have $|s|H_0(s) = \log|s| + O(1)$. In this case, the bound (1) does not guarantee that the compression ratio is close to the entropy.

We would be interested, therefore, in proving "entropy-only" bounds of the form $\lambda|s|H_k(s) + \Theta(1)$. Unfortunately, such bounds cannot be established. To see this, consider the family of strings $s = a^n$; we have $|s|H_0(s) = 0$ for all of them and we cannot hope to compress all strings in this family in $\Theta(1)$ space. For that reason, [16] introduced the notion of 0-*th order modified empirical* entropy:

$$H_0^*(s) = \begin{cases} 0 & \text{if } |s| = 0 \\ (1 + \lfloor \log|s| \rfloor)/|s| & \text{if } |s| \neq 0 \text{ and } H_0(s) = 0 \\ H_0(s) & \text{otherwise.} \end{cases} \tag{2}$$

Note that if $|s| > 0$, $|s|H_0^*(s)$ is at least equal to the number of bits needed to write down the length of s in binary. The *k-th order modified empirical entropy* H_k^* is then defined in terms of H_0^* as the maximum compression we can achieve by looking at *no more than* k symbols preceding the one to be compressed. With a rather complex analysis [16] proves the bound $(5+3C_0)|s|H_k^*(s)+\log|s|+\Theta(1)$ bits for the output size of an algorithm described in [4].

This paper is the ideal continuation of [14] and [16]. We analyze the classical Move-to-Front encoding [2], and two popular (and effective) alternatives: Distance Coding [3, 6], and Inversion Frequencies Coding [1]. We prove that simple variants of these algorithms achieve "entropy-only" bounds of the form $\lambda|s|H_k^*(s)+\log|s|+\Theta(1)$ bits with λ as small as $(2.69+C_0)$. At the same time we prove that no algorithm can achieve a bound of the form $\lambda|s|H_0^*(s) + \Theta(1)$ bits with $\lambda \leq 2$.

The key tool for the analysis of Move-to-Front and its variants is the notion of local optimality introduced in [16].

Definition 2 *A compression algorithm* A *is* locally λ-optimal *if there exists a constant c_h such that for any string s and for any partition $s_1 s_2 \cdots s_t$ of s we have*

$$\mathsf{A}(s) \leq \lambda\left[\sum_{i=1}^{t} |s_i|H_0^*(s_i)\right] + c_h t,$$

where the constant c_h depends only on the alphabet size h. ∎

The importance of local optimality stems from the following lemma which establishes that processing the output of the bwt with a locally optimal algorithm yields an algorithm achieving an "entropy-only" bound.

Lemma 2.1 ([16]) *If* A *is locally λ-optimal then the bound*

$$|\mathsf{A}(\mathsf{bwt}(s))| \leq \lambda|s|H_k^*(s) + \log|s| + c_h h^k \tag{3}$$

holds simultaneously for any $k \geq 0$. ∎

Note that the term $\log|s|$ in (3) is due to the fact that $\mathsf{bwt}(s)$ consists of a permutation of s, which is compressed using A, and an integer in $[1, |s|]$ whose encoding takes $1 + \lfloor \log|s| \rfloor$ bits. Since $\log|s| \leq |s|H_0^*(s) \leq |s|H_k^*(s)$, we could rewrite the right hand size of (3) as $(\lambda+1)|s|H_k^*(s) + c_h h^k$ (this justifies the expression "entropy-only" bound); however, since for most strings it is $\log|s| \ll |s|H_k^*(s)$, keeping the term $\log|s|$ explicit provides a better picture of the performance of bwt-based compressors.

We conclude this section with two lemmas relating the order zero entropy of a string with its length and the number of runs in it. Given a string s, a run is a substring $s[i]s[i+1]\cdots s[i+k]$ of identical symbols, and a maximal run is a run which cannot be extended; that is, it is not a proper substring of a larger run.

Lemma 2.2 ([15, Sect. 3]) *The number of maximal runs in a string s is bounded by $1 + |s|H_0(s)$.* ∎

Lemma 2.3 *Let s be a string containing $\mathsf{runs}(s)$ maximal runs and let α, β and ϵ be positive constants; then*

$$\alpha \log |s| + \beta \mathsf{runs}(s) \leq \max(\alpha, \beta + \epsilon)|s|H_0^*(s) + O(1).$$

Proof: First suppose $\mathsf{runs}(s) \leq 2\alpha/\epsilon + 2 = O(1)$; since $\log |s| \leq |s|H_0^*(s)$ we have

$$\alpha \log |s| + \beta \mathsf{runs}(s) \leq \alpha |s|H_0^*(s) + O(1). \tag{4}$$

Now suppose $\mathsf{runs}(s) > 2\alpha/\epsilon + 2$. This assumption implies that the frequency of the most common character in s is at most $|s| - \lfloor \mathsf{runs}(s)/2 \rfloor < |s| - \alpha/\epsilon$, with equality if and only if all odd-numbered runs contain the same character and every even-numbered run has length 1. Since $H_0(s)$ is minimized when the distribution of characters is as skewed as possible, we have

$$\begin{aligned} |s|H_0(s) &> (|s| - \alpha/\epsilon) \log \left(\frac{|s|}{|s| - \alpha/\epsilon} \right) + (\alpha/\epsilon) \log \left(\frac{|s|}{\alpha/\epsilon} \right) \\ &\geq (\alpha/\epsilon) \log \left(\frac{|s|}{\alpha/\epsilon} \right) = (\alpha/\epsilon) \log |s| - O(1)\,, \end{aligned}$$

so $\log |s| \leq (\epsilon/\alpha)|s|H_0(s) + O(1)$. Combining this inequality with Lemma 2.2 we get

$$\alpha \log |s| + \beta \mathsf{runs}(s) \leq (\beta + \epsilon)|s|H_0^*(s) + O(1)$$

which, together with (4) proves the lemma. ∎

3 Integer and Order-0 encoders

Move-to-Front, Distance Coding, and Inversion Frequencies all output sequences of positive integers. These sequences are usually compressed using either an order-0 encoder (see Definition 1) or a prefix free encoding of the integers. Prefix free encoders use a fixed codeword for each integer regardless of its frequency and are therefore faster and easier to implement. order-0 encoders (especially arithmetic coders) are slower but usually achieve a significantly better compression. Unfortunately, they are also more difficult to analyze when used in connection with the bwt. In this section we show that the compression achieved by a generic order zero encoder can be bounded in terms of the best compression achieved by a *family* of integer coders. This result will make it possible to translate compression bounds for integer coders to compression bounds for order zero encoders.

In the following we denote by Pfx a uniquely decodable encoder of the positive integers (not necessarily prefix free). Our only assumption is that there exist two positive constants a and b such that for any positive integer i we have $|\mathsf{Pfx}(i)| \leq a \log i + b$. For example, for γ-coding [7] the above inequality holds for $a = 2$ and $b = 1$. In our analysis we will often make use of the following property.

Lemma 3.1 (Subadditivity) *Let a, b be two constants such that for $i > 0$ it is $|\mathsf{Pfx}(i)| \leq a \log i + b$. Then, there exists a constant d_{ab} such that for any sequence of positive integers $x_1, x_2, \ldots, x_k$ we have*

$$\left| \mathsf{Pfx}\Big(\textstyle\sum_{j=1}^{k} x_j \Big) \right| \;\leq\; \Big(\textstyle\sum_{j=1}^{k} |\mathsf{Pfx}(x_j)| \Big) \;+\; d_{ab}.$$

Proof: Using elementary calculus it is easy to show that $\mathsf{Pfx}(x_1 + x_2) \leq \mathsf{Pfx}(x_1) + \mathsf{Pfx}(x_2)$ whenever $\min(x_1, x_2) \geq 2$. Hence we only need to take care of the case in which some of the x_j's are 1. For $x \geq 1$ we have:

$$\begin{aligned} |\mathsf{Pfx}(x+1)| - |\mathsf{Pfx}(x)| - |\mathsf{Pfx}(1)| &= a\log(1+(1/x)) - b & (5)\\ &= (a\log e)\ln(1+(1/x)) \; - \; b \\ &\leq (a\log e)/x - b, & (6)\end{aligned}$$

where the last inequality holds since $t \geq 0$ implies $\ln(1+t) \leq t$. Let $c_{ab} = (a\log e)/b$. From (5) we get that $x \geq 1$ implies $\mathsf{Pfx}(x+1) \leq \mathsf{Pfx}(x) + \mathsf{Pfx}(1) + (a-b)$ and from (6) we get that $x \geq c_{ab}$ implies $\mathsf{Pfx}(x+1) \leq \mathsf{Pfx}(x) + \mathsf{Pfx}(1)$. Combining these inequalities we get

$$\left|\mathsf{Pfx}\Big(\sum_{j=1}^{k} x_j\Big)\right| \;\leq\; \Big(\sum_{j=1}^{k} |\mathsf{Pfx}(x_j)|\Big) + c_{ab}\max(a-b,0),$$

and the lemma follows with $d_{ab} = c_{ab}\max(a-b,0)$. ∎

The next lemma, which follows from the analysis in [14], establishes a connection between integer and order-0 coders by showing that if we feed a sequence of integers to an order-0 encoder the output is essentially no larger than the output produced by an integer encoder Pfx with parameters $a = \mu$ and $b = \log(\zeta(\mu)) + C_0$ for any $\mu > 1$.

Lemma 3.2 *Let* $\mathsf{Order0}$ *be an order zero encoder with per character overhead* C_0 *and let* $x_1x_2\cdots x_n$ *be a sequence of integers such that* $1 \leq x_i \leq h$ *for* $i = 1,\ldots,n$. *Then, for any* $\mu > 1$ *we have*

$$|\mathsf{Order0}(x_1x_2\cdots x_n)| \leq \sum_{i=1}^{n}\Big(\mu\log(x_i) + \log\zeta(\mu) + C_0\Big) \;+\; O(h\log h)\,.$$

Proof: For any $\mu > 1$, consider the probability distribution over the positive integers defined by $q(j) = (\zeta(\mu)\, j^{\mu})^{-1}$. By the definition of Riemann zeta function it is $\zeta(\mu) = \sum_{j>0} j^{-\mu}$ hence $\sum_{j>0} q(j) = 1$. This implies that $nH_0(x_1\cdots x_n) \leq \sum_{i=1}^{n}\log(q(x_i))$. By Definition 1 we get

$$\begin{aligned} |\mathsf{Order0}(x_1x_2\cdots x_n)| &\leq nH_0(x_1\cdots x_n) + nC_0 + O(h\log h)\\ &\leq -\sum_{i=1}^{n}\log(q(x_i)) + nC_0 + O(h\log h)\\ &\leq \sum_{i=1}^{n}(\mu\log(x_i) + \log\zeta(\mu) + C_0) + O(h\log h)\,.\end{aligned}$$

∎

In the following we will make use also of a compression algorithm that combines the good features of both integer and order-0 encoders. The reason is that many of the procedures considered in this paper produce a sequence of positive integers whose size can be as large as the length of the input string s. This is not a problem when we compress such sequences using a prefix-free integer encoder: these encoders are able to handle arbitrarily large integers. Unfortunately, this is not longer true if we use an order-0 encoder. Typical order-0 algorithms have an overhead of $O(h\log h)$ bits, where h is the size of the input alphabet (see Definition 1). If we allow the encoding of values as large as $|s|$ such overhead makes the use of the encoder not profitable. Practitioners are well aware of this phenomenon and circumvent it using an order-0 algorithm for encoding "small" integers and prefix-free codes for handling the (usually few) occurrences of large integers. We now describe one of such schemes and we show that it is equivalent to an integer coder Pfx with parameters $a = \mu$ and $b = \log(\zeta(\mu)) + C_0 + \nu$ for any constants $\mu > 1$ and $\nu > 0$.

Recall that δ-coding [7] is a prefix-free encoding of the integers such that for any $x > 0$ it is $|\delta(x)| \leq 1 + \log x + 2\log(1 + \log x)$. Hence, for any $\mu > 1$ and $\nu > 0$ we can find an integer t such that for $x \geq t$ it is

$$|\delta(x)| \leq \mu \log(x) + \log \zeta(\mu) + \nu - \log(2^\nu/(2^\nu - 1)). \tag{7}$$

Given an encoder Order0 with per-character overhead C_0 we define a new encoder Order0* that is based on Order0 but uses δ-coding for encoding the integers larger than t. Given the sequence $x_1 \cdots x_n$ let $y_1 \cdots y_n$ denote the sequence with all integers greater than t replaced by copies of t and let $z_1 \cdots z_\ell$ be all the integers at least t. To encode $x_1 \cdots x_n$ the algorithm Order0* encodes $y_1 \cdots y_n$ with Order0 and $z_1 \cdots z_\ell$ with δ-coding.

Lemma 3.3 *Let* Order0 *be an order zero encoder with per-character overhead C_0. For any sequence of positive integers $x_1 x_2 \cdots x_n$ and constants $\mu > 1$ and $\nu > 0$ we have*

$$|\mathsf{Order0}^*(x_1 x_2 \cdots x_n)| \leq \sum_{i=1}^n \Big(\mu \log(x_i) + \log \zeta(\mu) + \nu + C_0\Big) + O(1)\,.$$

Proof: Note that the sequence $y_1 \cdots y_n$ contains only integers between 1 and t. Assign to each integer j in this range the weight $q(j)$ defined by $q(j) = (2^\nu \zeta(\mu) j^\mu)^{-1}$ for $j = 1, \ldots, t-1$, and $q(t) = 1 - 2^{-\nu}$. Since $\sum_{j=1}^t q(j) \leq 1$, it is $nH_0(y_1 \cdots y_n) \leq -\sum_{i=1}^n \log(q(y_i))$. We have

$$\begin{aligned}
|\mathsf{Order0}^*(x_1 \cdots x_n)| &= \sum_{i=1}^{\ell} |\delta(z_i)| \;+\; \mathsf{Order0}(y_1 \cdots y_n) \\
&\leq \sum_{i=1}^{\ell} |\delta(z_i)| \;+\; nH_0(y_1 \cdots y_n) \;+\; nC_0 \;+\; O(t \log t) \\
&\leq \sum_{x_i \geq t} |\delta(x_i)| \;-\; \sum_{i=1}^{n} \log(q(y_i)) \;+\; nC_0 \;+\; O(t \log t) \\
&\leq \sum_{x_i \geq t} \Big(\mu \log x_i + \log \zeta(\mu) + \nu - \log(2^\nu/(2^\nu - 1))\Big) \;+ \\
&\quad \sum_{x_i < t} \Big(\mu \log x_i + \log \zeta(\mu) + \nu\Big) \;-\; \sum_{x_i \geq t} \log(q(t)) \;+\; nC_0 \;+\; O(t \log t)\,.
\end{aligned}$$

Observing that $\log(q(t)) = -\log(2^\nu/(2^\nu - 1))$, we conclude that

$$|\mathsf{Order0}^*(x_1 \cdots x_n)| \leq \sum_{i=1}^{n} \Big(\mu \log x_i + \log \zeta(\mu) + \nu + C_0\Big) \;+\; O(t \log t)\,.$$

The thesis follows since t depends only on the constants μ and ν. ∎

4 Local Optimality with Move-to-Front encoding

The Move-to-Front (Mtf) procedure [2, 18] encodes a string by replacing each symbol with the number of distinct symbols seen since its last occurrence plus one. To this end, Mtf 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. If the input string is defined over the alphabet Σ we assume that ranks are in the range $[1, h]$, where $h = |\Sigma|$. To completely determine the encoding procedure we must specify the initial status of the recency list. However, changing the initial status increases the output size by at most $O(h \log h)$ bits so we will add this overhead and ignore the issue. We denote by Mtf + Pfx the algorithm in which the ranks produced by Mtf are encoded using Pfx. The analysis in [2] implies that for any string s the output of Mtf + Pfx is bounded by $a|s|H_0(s) + b|s|$ bits. The following example shows that Mtf + Pfx is not locally optimal according to Definition 2.

Example 1 Consider the string $s = \sigma^n$ and the partition consisting of the single element s. We have $\mathsf{Mtf}(s) = 1^n$ and $|\mathsf{Pfx}(\mathsf{Mtf}(s))| = |s|b$. Since $|s|H_0^*(s) = 1 + \lfloor \log |s| \rfloor$ it follows that $\mathsf{Mtf} + \mathsf{Pfx}$ is not locally optimal. A similar result holds even if we replace Pfx with an order-0 encoder: in that case the output size is at least $C_0|s|$ bits. ∎

The above example shows that if any compressor feeds to the final integer or order-0 encoder $\Theta(|s|)$ symbols, there is no hope of achieving "entropy-only" bounds. This observation suggests the algorithm $\mathsf{Mtf\_rle}$ that combines Mtf with Run Length Encoding. Assume $\sigma = s[i+1]$ is the next symbol to be encoded. Instead of simply encoding the Mtf rank r of σ, $\mathsf{Mtf\_rle}$ finds the maximal run $s[i+1] \cdots s[i+\ell]$ of consecutive occurrences of σ and encodes the pair[2] $\langle r, \ell \rangle$. We define the algorithm $\mathsf{Mtf\_rle} + \mathsf{Pfx}$ as the algorithm which encodes each such pair with Pfx. Since the Mtf rank r is always greater than one, to save space we encode each pair as follows: If $\ell = 1$, we encode $\langle r, \ell \rangle$ with the codewords $\langle \mathsf{Pfx}(1), \mathsf{Pfx}(r) \rangle$, while if $\ell > 1$ we encode $\langle r, \ell \rangle$ with the codewords $\langle \mathsf{Pfx}(r), \mathsf{Pfx}(\ell - 1) \rangle$.

Lemma 4.1 *Let $\mathsf{A}_0 = \mathsf{Mtf\_rle} + \mathsf{Pfx}$. For any string s we have*

$$|\mathsf{A}_0(s)| \leq 2a|s|H_0^*(s) + a \log \ell + (2b - a)\mathsf{runs}(s) + O(h \log h) .$$

where $\mathsf{runs}(s)$ is the number runs in s, and ℓ is the length of the last run.

Proof: Assume $H_0(s) \neq 0$ (otherwise $s = \sigma^n$ and the proof follows by an easy computation). Let $\langle r_1, \ell_1 \rangle$, $\langle r_2, \ell_2 \rangle$, ..., $\langle r_t, \ell_t \rangle$ denote the set of pairs generated by $\mathsf{Mtf\_rle}$. Because of the way A_0 encodes the pairs $\langle r_j, \ell_j \rangle$'s, if we define $|\mathsf{Pfx}(0)|$ to be equal to $|\mathsf{Pfx}(1)|$ the encoding of each pair $\langle r_j, \ell_j \rangle$ takes precisely $|\mathsf{Pfx}(r_j)| + |\mathsf{Pfx}(\ell_j - 1)|$ bits. Hence, we can write

$$|\mathsf{A}_0(s)| = \sum_{j=1}^{t} (|\mathsf{Pfx}(r_j)| + |\mathsf{Pfx}(\ell_j - 1)|) \ + \ O(h \log h) .$$

To bound $|\mathsf{A}_0(s)|$ we charge each term in the above summation to a character $\sigma \in \Sigma$ as follows: we charge the term $|\mathsf{Pfx}(r_j)|$ to the character forming the j-th run and the term $|\mathsf{Pfx}(\ell_j - 1)|$ to the character forming the $j+1$-st run. Note that this leaves out the last run length ℓ_t: its corresponding cost $|\mathsf{Pfx}(\ell_t - 1)|$ is accounted for explicitly in the statement of the lemma.

For any given character σ let (α_1, β_1), (α_2, β_2), ..., (α_k, β_k) denote the starting and ending positions of the runs of σ. For $i = 1, \ldots, k$ let $\langle r'_i, \ell'_i \rangle$ denote the pair encoding the run (α_i, β_i) (so we have $\ell'_i = \beta_i - \alpha_i + 1$). Finally, let m_i denote the length of the run immediately preceding the run (α_i, β_i). The total cost charged to σ is therefore

$$\sum_{i=1}^{k} |\mathsf{Pfx}(r'_i)| + |\mathsf{Pfx}(m_i - 1)|. \tag{8}$$

Define $\beta_0 = 0$. We now show that for $i \geq 1$ we have

$$|\mathsf{Pfx}(r'_i)| + |\mathsf{Pfx}(m_i - 1)| \leq 2 \log(\alpha_i - \beta_{i-1}) + 2b - a. \tag{9}$$

Assume first $m_i > 1$. Recall r'_i is the number of distinct characters in the substring from $s[\beta_{i-1} + 1]$ to $s[\alpha_i]$. If, immediately before $s[\alpha_i]$ there is a run of m_i equal symbols, we have $r'_i \leq \alpha_i - \beta_{i-1} - (m_i - 1)$. Hence

$$\begin{aligned} |\mathsf{Pfx}(r'_i)| + |\mathsf{Pfx}(m_i - 1)| &= a(\log(r'_i) + \log(m_i - 1)) + 2b \\ &\leq 2a \log((r'_i + m_i - 1)/2) + 2b \\ &\leq 2a \log(\alpha_i - \beta_{i-1}) + 2b - a. \end{aligned}$$

[2]Here and in the following we use angle brackets to show that certain values form a pair or a triple with a particular meaning: such brackets are not part of the output.

If $m_i = 1$, then $|\mathsf{Pfx}(m_i - 1)| = b$. Since $2 \leq r'_i \leq \alpha_i - \beta_{i-1}$, we have

$$\begin{aligned} |\mathsf{Pfx}(r'_i)| + |\mathsf{Pfx}(m_i - 1)| &= a \log(r'_i) + 2b \\ &\leq 2a \log(\alpha_i - \beta_{i-1}) + 2b - a \end{aligned}$$

thus establishing (9). Using (9), the total cost (8) charged to σ can be bounded by

$$2a \left[\log(\alpha_1 - \beta_0) + \log(\alpha_2 - \beta_1) + \cdots + \log(\alpha_k - \beta_{k-1})\right] + k(2b - a) \tag{10}$$

bits. Summing the cost $k(2b - a)$ over all characters in Σ yields a total of $(2b - a)\,\mathsf{runs}(s)$ bits. To complete the proof we bound the content of the square brackets in (10). Since $\log(1) = 0$, the content of the square brackets is equal to

$$\log(\alpha_1 - \beta_0) + \cdots + \log(\alpha_k - \beta_{k-1}) + (\beta_1 - \alpha_1 + \beta_2 - \alpha_2 + \cdots + \beta_k - \alpha_k) \log(1). \tag{11}$$

The number of logarithms in (11) is $\sum_{i=1}^{k} (\beta_i - \alpha_i + 1)$ which is equal to the number n_σ of occurrences of σ in s. Hence, by Jensen's inequality, (11) is bounded by

$$n_\sigma \log\left(\frac{(\alpha_1 - \beta_0) + \cdots + (\alpha_k - \beta_{k-1}) + (\beta_1 - \alpha_1 + \cdots + \beta_k - \alpha_k)}{n_\sigma}\right) = n_\sigma \log((\beta_k - \beta_0)/n_\sigma)$$

which is at most $n_\sigma \log(|s|/n_\sigma)$. Summing $n_\sigma \log(|s|/n_\sigma)$ over all σ's yields $|s|H_0(s)$ and the lemma follows. ∎

Since the length of the last run is bounded by $|s|$, combining Lemmas 4.1 and 2.3 we get

Corollary 4.2 *Let* $\mathsf{A}_0 = \mathsf{Mtf\_rle} + \mathsf{Pfx}$. *For any string* s *and* $\epsilon > 0$ *we have*

$$|\mathsf{A}_0(s)| \leq \max(3a, a + 2b + \epsilon)|s|H_0^*(s) + O(h \log h)\,. \quad ∎$$

Theorem 4.3 *The algorithm* $\mathsf{A}_0 = \mathsf{Mtf\_rle} + \mathsf{Pfx}$ *is locally* $\max(3a, a + 2b + \epsilon)$*-optimal for any* $\epsilon > 0$.

Proof: By Corollary 4.2 it suffices to prove that

$$|\mathsf{A}_0(s_1 s_2)| \leq |\mathsf{A}_0(s_1)| + |\mathsf{A}_0(s_2)| + O(h \log h).$$

To prove this inequality observe that compressing s_2 independently of s_1 changes the encoding of the Mtf rank only of the first occurrence of each character in s_2. This gives a $O(h \log h)$ overhead. In addition, there could be a run of equal characters crossing the boundary between s_1 and s_2. In this case the length of the first part of the run will be encoded in s_1 and the length of the second part in s_2. By Lemma 3.1 this produces an $O(1)$ overhead and the theorem follows. ∎

Using Lemma 3.3, we can extend the above theorem to the case in which the output of Mtf_rle is compressed with the algorithm Order0* described at the end of Section 3. Recall that Order0* combines δ coding with an order-0 encoder with per-symbol overhead C_0.

Theorem 4.4 *The algorithm* $\mathsf{Mtf\_rle} + \mathsf{Order0*}$ *is locally* $(4.40 + C_0)$*-optimal.*

Proof: By Lemma 3.3 we know that for any $\mu > 1$ and $\nu > 0$ the output of Order0* on input $\mathsf{Mtf\_rle}(s)$ is bounded by the output of an integer coder with parameters $a = \mu$ and $b = \log(\zeta(\mu)) + \nu + C_0$. The thesis follows by Theorem 4.3 taking $\mu = 22/15$ and $\nu = \epsilon = 0.001$. ∎

Corollary 4.5 *For any string* s *and* $k \geq 0$ *we have*

$$|\mathsf{Order0*}(\mathsf{Mtf\_rle}(\mathsf{bwt}(s)))| \leq (4.40 + C_0)|s|H_k^*(s) + \log|s| + O\Big(h^{k+1} \log h\Big)\,.$$

Proof: Immediate by Theorem 4.4 and Lemma 2.1. ∎

Procedure Distance Coding

1. Write the first character in s;
2. for each other character $\sigma \in \Sigma$, write the distance to the first σ in s, or 1 if σ does not occur (notice no distance is 1, because we do not reconsider the first character in s);
3. for each maximal run of a character σ, write the distance from the ending position of that run to the starting position of the next run of σ's, or 1 if there are no more σ's (again, no distance is 1);
4. encode the length ℓ of the last run in s.

Figure 1: Distance coding of a string s over the alphabet $\Sigma = \{\sigma_1, \ldots, \sigma_h\}$.

5 Local Optimality with Distance Coding

Distance Coding (Dc) is an encoding procedure which is relatively little-known, probably because it was originally described only on a Usenet post [3]. Recently, [14] has proven that processing $\mathsf{bwt}(s)$ with Dc followed by an order-0 encoder produces an output bounded by $1.7286|s|H_k(s) + C_0|s| + o(|s|)$ bits. Using Lemma 2.2, the analysis in [14] can be easily refined to get the bound $(1.7286 + C_0)|s|H_k(s) + o(|s|)$ bits. Note that this is not an "entropy-only" bound because of the presence of the $o(|s|)$ term.

The basic idea of Dc is to encode the starting position of each maximal run. The details of the algorithm are given in Figure 1. Note that Dc does not encode the length of the runs since the ending position of the current run is determined by the starting position of the next run. The distance between two characters is defined as the number of characters between them plus one (so the distance is one if the two characters are consecutive). The distance of a character from the beginning of s is defined as the number of characters preceding it plus one (so the distance is one for the first character of the string s). We define $\mathsf{Dc} + \mathsf{Pfx}$ as the algorithm in which the integers produced by Dc are encoded using Pfx.

Lemma 5.1 *Let* $\mathsf{A}_1 = \mathsf{Dc} + \mathsf{Pfx}$. *For any string* s *and for any* $\epsilon > 0$ *we have*

$$|\mathsf{A}_1(s)| \leq \max(2a, a + b + \epsilon)|s|H_0^*(s) + O(h).$$

Proof: Assume $H_0(s) \neq 0$ (otherwise $s = \sigma^n$ and the proof follows by an easy computation). Writing the first character in s takes $O(\log h)$ bits; we write h copies of 1 while encoding s (or $h + 1$ if the first character is a 1), which takes $O(h)$ bits. Writing the length of the last run takes $|\mathsf{Pfx}(\ell)|$ which is at most $a \log \ell + b$ bits. We are left with the task of bounding the cost of encoding: 1) the starting position of the first run of each character, 2) the distances between the ending position of a run and the starting position of the next run of the same character. We account these costs separately for each $\sigma \in \Sigma$. Let (α_1, β_1), (α_2, β_2), ..., (α_k, β_k) denote the starting and ending positions of the runs of σ. Dc encodes these runs with the sequence of codewords

$$\mathsf{Pfx}(\alpha_1), \mathsf{Pfx}(\alpha_2 - \beta_1), \mathsf{Pfx}(\alpha_3 - \beta_2), \ldots, \mathsf{Pfx}(\alpha_k - \beta_{k-1})$$

whose overall size is bounded by (setting $\beta_0 = 0$)

$$a\left[\log(\alpha_1 - \beta_0) + \log(\alpha_2 - \beta_1) + \cdots + \log(\alpha_k - \beta_{k-1})\right] + bk \tag{12}$$

bits. Summing the above term over all σ and reasoning as in the proof of Lemma 4.1 (compare (12) with (10)) we get

$$|\mathsf{A}_1(s)| \leq a \log \ell + a|s|H_0(s) + b\,\mathsf{runs}(s) + O(h),$$

where $\mathsf{runs}(s)$ is the number of runs in s. The thesis follows by Lemma 2.3. ∎

Unfortunately, the following example shows that $\mathsf{Dc} + \mathsf{Pfx}$ is not locally optimal.

Example 2 Consider the partition $s = s_1 s_2 s_3$ where

$$s_1 = s_3 = \sigma_1 \sigma_2 \cdots \sigma_h, \quad s_2 = \sigma_2^n.$$

We have $\sum_{i=1}^{2h} |s_i| H_k^*(s_i) = \log n + O(h \log h)$, whereas $\mathsf{Dc} + \mathsf{Pfx}$ produces an output of size $\Theta(h \log n)$. To see this, observe that for each character $\sigma \neq \sigma_2$ it has to write a distance greater than n. ∎

The above example suggests that to achieve local optimality with Dc we should try to avoid the encoding of "long jumps". To this end, we introduce Distance Coding with escapes (Dc_esc). The main difference between Dc and Dc_esc is that, whenever Dc would write a distance, Dc_esc compares the cost of writing that distance to the cost of escaping and re-entering later, and does whichever is cheaper.

Whenever Dc would write 1, Dc_esc writes $\langle 1, 1 \rangle$; this lets us use $\langle 1, 2 \rangle$ as a special re-entry sequence. To escape after a run of σ's, we write $\langle 1, 1 \rangle$; to re-enter at the the next run of σ's, we write $\langle 1, 2, \ell, \sigma \rangle$, where ℓ is the length of the preceding run (necessarily of some other character). To see how Dc_esc works suppose we are encoding the string

$$s = \cdots \sigma_1^j \sigma_2^k \sigma_3^\ell \sigma_1^m \cdots.$$

When Dc reaches the run σ_1^j it encodes the value $k + \ell + 1$ which is the distance from the last σ_1 in σ_1^j to the first σ_1 in σ_1^m. Instead, Dc_esc compares the cost of encoding $k + \ell + 1$ with the cost of encoding an escape (sequence $\langle 1, 1 \rangle$) plus the cost of re-entering. In this case the re-entry sequence would be written immediately after the code associated with the run σ_3^ℓ and would consist of the sequence $\langle 1, 2, \ell, \sigma_1 \rangle$. When the decoder finds such sequence it knows that the current run (in this case of σ_3's) will only last for ℓ characters and, after that, there is a run of σ_1's. (Recall that Dc only encodes the starting position of each run: the end of the run is induced by the beginning of a new run. When we re-enter an escaped character we have to provide explicitly the length of the ongoing run).

Notice we do not distinguish between instances in which $\langle 1, 1 \rangle$ indicates a character does not occur, cases in which it indicates a character does not occur again, and cases in which it indicates an escape; we view the first two types of cases as escapes without matching re-entries.

Lemma 5.2 *Let* $\mathsf{A}_1 = \mathsf{Dc} + \mathsf{Pfx}$ *and let* $\mathsf{A}_2 = \mathsf{Dc\_esc} + \mathsf{Pfx}$. *For any string* s *and for any partition* $s = s_1, \ldots, s_t$

$$|\mathsf{A}_2(s)| \leq \sum_{i=1}^{t} |\mathsf{A}_1(s_i)| + O(ht \log h).$$

Proof: We consider the algorithm Dc_esc* that, instead of choosing at each step whether to escape or not, escapes if and only if the current distance crosses the boundary between two different partition elements. That is, Dc_esc* uses the escape sequence every time it encodes the distance between a run ending in s_i and a run starting in s_j with $j > i$. Let $\mathsf{A}_2^* = \mathsf{Dc\_esc*} + \mathsf{Pfx}$. Since Dc_esc always performs the most economical choice, we have $|\mathsf{A}_2(s)| \leq |\mathsf{A}_2^*(s)|$; we prove the lemma by showing that

$$|\mathsf{A}_2^*(s)| \leq \sum_{i=1}^{t} |\mathsf{A}_1(s_i)| + O(ht \log h).$$

Clearly Dc_esc* escapes at most th times. The parts of an escape/re-enter sequence that cost $\Theta(1)$ (that is, the codewords for $\langle 1, 1 \rangle$, $\langle 1, 2 \rangle$ and the encoding of the escaped character σ) are therefore included in the $O(ht \log h)$ term. Thus, for each escape sequence we only have to take care of the cost of encoding of the value ℓ that provides the length of the run immediately preceding the re-entry point. We now show that the cost of encoding the run lengths ℓs is bounded by costs paid by Dc and not paid by Dc_esc*. Let σ denote the escaped character. Let s_j denote the partition element containing the re-entry point and let m denote the position in s_j where the new run of σ's starts (that is, at position m of s_j there starts a run of σ's; the previous one ended in some s_i with $i < j$ so Dc_esc* escaped σ and is now re-entering). Let σ_p denote the character immediately preceding the re-entry point: with our notation we have that the re-entry point is preceded by the run σ_p^ℓ. We consider two cases:

$\ell \leq m$. In this case the run σ_p^ℓ starts within s_j. This implies that the cost $|\mathsf{Pfx}(\ell)|$ paid by $\mathsf{Dc\_esc}^*$ is no greater than the cost $|\mathsf{Pfx}(m)|$ paid by Dc for encoding the first position of σ in s_j.

$\ell > m$. In this case the run σ_p^ℓ starts in a partition element preceding s_j. Let $m' = \ell - m$. If $m' < |s_{j-1}|$ the run σ_p^ℓ starts within s_{j-1}. Under this assumption, by Lemma 3.1 the cost $|\mathsf{Pfx}(\ell)|$ paid by $\mathsf{Dc\_esc}^*$ is at most d_{ab} plus the cost $|\mathsf{Pfx}(m)|$ paid by Dc for encoding the first position of σ in s_j, plus the cost $|\mathsf{Pfx}(m')|$ paid by Dc to encode the length of the last run in s_{j-1}. If $m' > |s_{j-1}|$ then the run σ_p^ℓ spans several partition elements s_{j-k}, s_{j-k+1}, ..., s_j. In this case, again by Lemma 3.1, the cost $|\mathsf{Pfx}(\ell)|$ is bounded by d_{ab} plus the cost paid by Dc for encoding the following items: 1) the last run in s_{j-k}, 2) the last (and only) run in s_{j-k+1}, ..., s_{j-1}, 3) the first position of σ in s_j.

■

Combining Lemma 5.2 with Lemma 5.1 we immediately get

Theorem 5.3 *The algorithm* $\mathsf{A}_2 = \mathsf{Dc\_esc} + \mathsf{Pfx}$ *is locally* $\max(2a, a + b + \epsilon)$*-optimal for any* $\epsilon > 0$. ■

We now consider the case in which the output of $\mathsf{Dc\_esc}$ is compressed with the encoder $\mathsf{Order0}^*$. The main tool for our analysis will be again Lemma 3.3 that establishes a relationship between the output size of $\mathsf{Order0}^*$ of that of an integer coder. However, there is the technical difficulty that for a generic order zero encoder we do not necessarily have the concept of codeword assigned to each input symbol: the concept of codeword is well defined for example for Huffman coding but not for Arithmetic coding. This could be a problem for the algorithm $\mathsf{Dc\_esc}$ that, in order to decide whether to escape or not, has to compare the cost of encoding two different set of symbols.

Theorem 5.4 *The algorithm* $\mathsf{Dc\_esc} + \mathsf{Order0}^*$ *is locally* $(2.94 + C_0)$*-optimal.*

Proof: Fix $\mu > 1$ and $\nu > 0$. Let $\mathsf{Pfx}_{\mu\,\nu}$ be the (ideal) integer coder such that $|\mathsf{Pfx}_{\mu\,\nu}(i)| = \mu \log i + \log(\zeta(\mu)) + \nu + C_0$. Let $\mathsf{Dc\_esc}_{\mu\,\nu}$ denote the algorithm that decides whether to escape or not on the basis of the costs given by $\mathsf{Pfx}_{\mu\,\nu}$. By Theorem 5.3 $\mathsf{Dc\_esc}_{\mu\,\nu} + \mathsf{Pfx}_{\mu\,\nu}$ is locally $\max(2\mu, \mu + \log(\zeta(\mu)) + \nu + \epsilon + C_0)$-optimal for any $\epsilon > 0$. Since by Lemma 3.3 $|\mathsf{Order0}^*(\mathsf{Dc\_esc}_{\mu\,\nu}(s))| \leq |\mathsf{Pfx}_{\mu\,\nu}(\mathsf{Dc\_esc}_{\mu}(s))|$ the local optimality result stated in Theorem 5.3 holds for $\mathsf{Dc\_esc}_{\mu\,\nu} + \mathsf{Order0}^*$ as well. The theorem follows taking $\mu = 1.47$ and $\nu = \epsilon = 0.001$. ■

5.1 Using an explicit escape symbol

We now show how to improve the performance of $\mathsf{Dc\_esc}$ by using a special escape symbol to introduce escape/re-enter sequences. The rationale is that escape/re-enter sequences are relatively rare so it pays to use a special low-probability symbol for them. The escape symbol will be used also by our variant of the Inversion Frequencies algorithm.

Lemma 5.5 *Let* Pfx *be a code for the integers such that for* $i > 0$ *it is* $|\mathsf{Pfx}(i)| \leq a \log i + b$*. For any* $\delta > 0$ *there exists a code* $\mathsf{Pfx}^{(\delta)}$ *such that: 1) for* $i > 0$ *it is* $|\mathsf{Pfx}^{(\delta)}(i)| \leq (1 + \delta)(a \log i) + b$*, 2) in addition to the positive integers* $\mathsf{Pfx}^{(\delta)}$ *can encode a special escape symbol* esc*.*

Proof: Given $\delta > 0$ let i_δ denote the smallest integer such that $\log(i + 1) \leq (1 + \delta) \log i$. We define the code $\mathsf{Pfx}^{(\delta)}$ as follows: $\mathsf{Pfx}^{(\delta)}(\mathsf{esc}) = \mathsf{Pfx}(i_\delta)$ and

$$\mathsf{Pfx}^{(\delta)}(i) = \begin{cases} \mathsf{Pfx}(i) & \text{for } i < i_\delta, \\ \mathsf{Pfx}(i+1) & \text{for } i \geq i_\delta. \end{cases}$$

The lemma follows since the concavity of $\log x$ ensures $|\mathsf{Pfx}^{(\delta)}(i)| \leq (1 + \delta)(a \log i) + b$ for any $i > 0$. ■

Let Esc_1 denote the procedure that given a sequence of positive integers replaces every occurrence of 1 with the symbol esc. For example: $\mathsf{Esc}_1(2113314) = 2\,\mathsf{esc}\,\mathsf{esc}\,3\,3\,\mathsf{esc}\,4$. Let $\mathsf{B}_2 = \mathsf{Dc\_esc} + \mathsf{Esc}_1 + \mathsf{Pfx}^{(\delta)}$. Note that in B_2 every occurrence of the symbol 1 produced by $\mathsf{Dc\_esc}$ is eventually encoded with the codeword $\mathsf{Pfx}^{(\delta)}(\mathsf{esc})$. We assume that $\mathsf{Dc\_esc}$ assigns the cost $|\mathsf{Pfx}^{(\delta)}(\mathsf{esc})|$ to the symbol 1 when it has to decide whether to escape or not.

Lemma 5.6 *For any positive constants ϵ, δ, the algorithm $\mathsf{B}_2 = \mathsf{Dc\_esc} + \mathsf{Esc}_1 + \mathsf{Pfx}^{(\delta)}$ is locally λ-optimal with $\lambda = \max(2a', a' + b + \epsilon)$, $a' = a(1+\delta)$.*

Proof: Let $\mathsf{B}_1 = \mathsf{Dc} + \mathsf{Esc}_1 + \mathsf{Pfx}^{(\delta)}$. Since Dc outputs the symbol 1 at most $2h$ times, replacing it with esc introduces a $O(h)$ overhead. Replacing Pfx with $\mathsf{Pfx}^{(\delta)}$ introduces a multiplicative overhead of $(1+\delta)$ to each log term, repeating the proof of Lemma 5.1 we get

$$|\mathsf{B}_1(s)| \leq \max(2a', a' + b + \epsilon)|s|H_0^*(s) + O(h). \tag{13}$$

Consider now $\mathsf{B}_2^* = \mathsf{Dc\_esc*} + \mathsf{Esc}_1 + \mathsf{Pfx}^{(\delta)}$, where $\mathsf{Dc\_esc*}$ is defined as in the proof of Lemma 5.2. Reasoning as in Lemma 5.2 we have that for any partition $s = s_1 \cdots s_t$

$$|\mathsf{B}_2(s)| \leq |\mathsf{B}_2^*(s)| \leq \sum_{i=1}^{t} |\mathsf{B}_1(s_i)| \; + \; O(ht \log h)$$

where the second inequality follows by the fact that $\mathsf{Dc\_esc*}$ outputs the esc symbol at most $O(ht)$ times. The lemma follows combining the above inequality with (13). ∎

Theorem 5.7 *The algorithm $\mathsf{Dc\_esc} + \mathsf{Esc}_1 + \mathsf{Order0*}$ is locally $(2.69 + C_0)$-optimal.*

Proof: Fix $\mu > 1$ and $\nu > 0$. Let $\mathsf{Pfx}_{\mu\,\nu}$ be the (ideal) coder for the set $\{2, 3, \ldots\}$ such that $|\mathsf{Pfx}_{\mu\,\nu}(i)| = \mu \log i + \log(\zeta(\mu) - 1) + \nu + C_0$, and let $\mathsf{Pfx}_{\mu\,\nu}^{(\delta)}$ be the coder obtained applying Lemma 5.5. Note that $\mathsf{Pfx}_{\mu\,\nu}^{(\delta)}$ can encode only symbols over the set $\Sigma' = \{\mathsf{esc}\} \cup \{2, 3, 4, \ldots, \}$ but this is fine since the string $\mathsf{Esc}_1(\mathsf{Dc\_esc}(s))$ does not contain the symbol 1. By Lemma 5.6, for any $\epsilon, \delta > 0$, the algorithm $\mathsf{Dc\_esc} + \mathsf{Esc}_1 + \mathsf{Pfx}_{\mu\,\nu}^{(\delta)}$ is λ-optimal with $\lambda = \max(2\mu(1+\delta), \mu(1+\delta) + \log(\zeta(\mu) - 1) + \nu + C_0 + \epsilon)$. Reasoning as in the proof of Lemma 3.3 we have that replacing $\mathsf{Pfx}_{\mu\,\nu}^{(\delta)}$ with $\mathsf{Order0*}$ can only reduce the output size so the local optimality result holds for $\mathsf{Order0*}$ as well. The theorem follows taking $\mu = 1.343$, and $\nu = \epsilon = \delta = 0.001$. ∎

Corollary 5.8 *For any string s and $k \geq 0$ we have*

$$|\mathsf{Order0*}(\mathsf{Esc}_1(\mathsf{Dc\_esc}(\mathsf{bwt}(s))))| \leq (2.69 + C_0)|s|H_k^*(s) + \log|s| + O\Big(h^{k+1} \log h\Big).$$

Proof: Immediate by Theorem 5.7 and Lemma 2.1. ∎

6 Local Optimality with Inversion Frequencies Coding

Inversion Frequencies (If for short) is a coding strategy proposed in [1] as an alternative to Mtf. Given a string s over an ordered alphabet $\Sigma = \{\sigma_1, \sigma_2, \ldots, \sigma_h\}$, in its original formulation If works in $h - 1$ phases. In the i-th phase If encodes the distance between every pair of consecutive occurrences of σ_i: in the computation of such distances If ignores the characters smaller than σ_i. In other words, in the i-th phase If conceptually builds the string $s^{(i)}$ removing from s the characters smaller than σ_i and encodes the distances between consecutive occurrences of σ_i in $s^{(i)}$. Note that there is no need to encode the occurrences of σ_h. The output of If consists of the concatenation of the output of the single phases

Inversion Frequencies with Run-Length Encoding (If_rle)

1. Write $h = |\Sigma|$ bits to indicate which characters are actually present in s (from now on we assume all characters are present);
2. For $i = 1, \ldots, h-1$: write the number ℓ_i of characters greater than σ_i preceding the first occurrence of σ_i in s; if $\ell_i = 0$ write esc instead.
3. Set $j = 1$ and repeat until $j \leq |s|$:
   (a) Let $\sigma_i = s[j]$. Let $s[m]$ be the first occurrence of a symbol greater than σ_i to the right of $s[j]$, and let $s[p]$ be the first occurrence of the symbol σ_i to the right of $s[m]$.
   (b) Write the pair $\langle k, \ell \rangle$ where k is the number of occurrences of σ_i in $s[j] \cdots s[m-1]$ and ℓ is the number of occurrences of symbols greater than σ_i in $s[m] \cdots s[p-1]$.
   (c) Set j to be the next position in s containing a character different from σ_i and σ_h.
4. Write the pair $\langle \mathsf{esc}, \mathsf{esc} \rangle$.

Figure 2: Inversion Frequencies with Run Length Encoding.

prefixed by an encoding of the number of occurrences of each symbol σ_i (this information is needed by the decoder to figure out when a phase is complete). For example, if $s = \sigma_2\sigma_2\sigma_1\sigma_3\sigma_3\sigma_1\sigma_3\sigma_1\sigma_3\sigma_2$, the first phase encodes the occurrences of σ_1, producing the sequence $\langle 3, 3, 2 \rangle$, and the second phase encodes the occurrences of σ_2, producing the sequence $\langle 1, 1, 5 \rangle$. The output of If is an encoding of the number of occurrences of σ_1, σ_2, and σ_3 (3, 3, and 4 in our example), followed by the sequence $\langle 3, 3, 2, 1, 1, 5 \rangle$.

Recently, [10, Sect. 3.2] has shown that If is equivalent to coding the string s with a skewed wavelet tree combined with Gap Encoding. The analysis in [10] shows that, if the alphabet is reordered so that σ_h is the most frequent symbol (so that it gets encoded for free), the output of If + Pfx is bounded by

$$|\mathsf{Pfx}(\mathsf{If}(s))| \leq \max(a, b)|s|H_0(s) + (|\Sigma| + a)\log|s| + O(1).$$

Unfortunately, the following example shows that If is not locally optimal.

Example 3 Consider the partition $s = s_1 s_2$ where $s_1 = \sigma_1^n$, $s_2 = \sigma_2^n$. We have $|s_1|H_0^*(s_1) + |s_2|H_0^*(s_2) = O(\log n)$, whereas, no matter how we order the alphabet $|\mathsf{Pfx}(\mathsf{If}(s))| = \Theta(n)$. ∎

We now describe two variants of the basic If procedure and we prove that the second variant is locally optimal. The first variant, called Inversion Frequencies with Run Length Encoding (If_rle), will serve only as an intermediate step, but it is already much more efficient than If. In addition, If_rle does a single pass working on all characters simultaneously instead of doing $h - 1$ passes as If.

If_rle produces a sequence over the set $\{\mathsf{esc}\} \cup \{1, 2, \ldots\}$ so its output will be encoded using the $\mathsf{Pfx}^{(\delta)}$ encoder described in Lemma 5.5. The outline of the procedure If_rle is described in Figure 2. Note that in the main body of If_rle (Step 3) we are essentially encoding the following information: "starting from the current character $\sigma_i = s[j]$ there are k occurrences of σ_i before we reach the first character greater than σ_i; after that there are ℓ characters greater than σ_i before we find another occurrence of σ_i". Note also that, similarly to If, the procedure If_rle does not encode explicitly any information about the occurrences of σ_h. In Step 3a we are assuming that the characters $s[p]$ and $s[m]$ always exist: this is not the case for the last run of each character that consequently requires a slightly different encoding. Here is how it is done. If $s[p]$ does not exist (there are no occurrences of σ_i to the right of $s[m]$), then If_rle writes the pair $\langle k, \ell \rangle$, where ℓ is equal to the number of characters greater than σ_i in $s[m] \cdots s[n]$. If $s[m]$ does not exist (there are no characters greater than σ_i to the right of $s[j]$) If_rle writes the pair $\langle k, \mathsf{esc} \rangle$.

The procedure for decoding the output of If_rle is shown in Figure 3. The decoder maintains two arrays *To_be_written*$[1, \ldots, h-1]$ and *To_be_skipped*$[1, \ldots, h-1]$ such that *To_be_written*$[i]$ stores how many σ_i's have to be written before we find a character greater than σ_i and *To_be_skipped*$[i]$ stores how many characters greater than σ_i there are between the end of the current run of σ_i's and the next one

Decoding Procedure for If_rle

1. Read $h = |\Sigma|$ bits to determine which characters are actually present in s (from now on we assume all characters are present);
2. For $i = 1, \ldots, h-1$, read ℓ_i and set *To_be_skipped*$[i] \leftarrow \ell_i$ and *To_be_written*$[i] \leftarrow 0$ (if $\ell_i = \mathsf{esc}$ set *To_be_skipped*$[i] \leftarrow 0$ instead);
3. Repeat until the pair $\langle \mathsf{esc}, \mathsf{esc} \rangle$ has been read:
   (a) let i be the smallest index such that *To_be_skipped*$[i] = 0$, read the next pair $\langle k, \ell \rangle$ and set *To_be_written*$[i] \leftarrow k$, *To_be_skipped*$[i] \leftarrow \ell$ (if all *To_be_skipped*$[i]$ are nonzero do nothing);
   (b) let i be the smallest index such that *To_be_written*$[i] \neq 0$; if all *To_be_written*$[i]$ are zero let $i = h$;
   (c) write σ_i to the output file;
   (d) for $j = 1, 2, \ldots, i-1$ set *To_be_skipped*$[j] \leftarrow$ *To_be_skipped*$[j] - 1$;

Figure 3: Decoding procedure for Inversion Frequencies with Run Length Encoding .

(again runs and distances for σ_i are defined ignoring smaller characters). For a single character σ_i the decoding procedure works as follows. Until *To_be_written*$[i] > 0$ the decoder outputs σ_i and decreases *To_be_written*$[i]$ by one. When *To_be_written*$[i]$ reaches zero the decoder decreases *To_be_skipped*$[i]$ by one each time it outputs a character greater than σ_i. When also *To_be_skipped*$[i]$ reaches zero the decoder needs new instructions for σ_i so it reads a new pair $\langle k, \ell \rangle$ from the compressed file and sets *To_be_written*$[i] \leftarrow k$ and *To_be_skipped*$[i] \leftarrow \ell$. The actual decoding procedure is more complex since it has to work on all characters $\sigma_1, \ldots, \sigma_h$ at the same time. So it is often the case that more than one *To_be_written*$[i]$ is greater than zero: in this case the smallest i wins. The reason is that, if $i < j$, the encoding of σ_j ignores the occurrences of σ_i so σ_i must take precedence. Note that the decoder outputs a character σ_h every time *To_be_skipped*$[j] > 0$ for every $j < h$. The last run of each character is handled as follows. If the decoder reads the pair $\langle k, \mathsf{esc} \rangle$ it sets *To_be_written*$[i] \leftarrow k$ and *To_be_skipped*$[i] \leftarrow \infty$: this means that there are k more occurrences of σ_i and no more.

Despite our care in designing If_rle the following example shows that the combination If_rle + $\mathsf{Pfx}^{(\delta)}$ is still not locally optimal.

Example 4 Consider the partition $s = s_1 s_2 \cdots s_{2h}$ where

$$s_1 = \sigma_1 \sigma_2 \cdots \sigma_h \quad s_2 = \sigma_1^n \quad s_3 = s_1 \quad s_4 = \sigma_2^n \quad s_5 = s_1 \quad s_6 = \sigma_3^n$$

and so on up to $s_{2h} = \sigma_h^n$. We have $\sum_{i=1}^{2h} |s_i| H_0^*(s_i) = O(h \log n)$, whereas, no matter how we order the alphabet $\mathsf{Pfx}^{(\delta)}(\mathsf{If\_rle}(s)) = O(h^2 \log n)$. ∎

An obvious inefficiency of If_rle is that sometimes we are forced to pay the cost of a "long jump" many times. Consider the string: $s = \sigma_1 \sigma_2 \sigma_3^n \sigma_2 \sigma_1$. Assuming $\sigma_1 < \sigma_2 < \sigma_3$ we see that If_rle pays a $O(\log n)$ cost for the encoding of both σ_1 and σ_2 because of the presence of the σ_3^n substring. We now propose an *escape and re-enter* mechanism that essentially guarantees that in the above situation we pay the $O(\log n)$ cost at most once.

The new algorithm, called Inversion Frequencies with RLE and Escapes (If_rle_esc), works as follows. Assume that $s[j] = \sigma_i$ is the next character to be encoded, and let $s[m]$, $s[p]$, k, and ℓ be defined as for the algorithm If_rle. Moreover, let o denote the largest index such that $m < o < p$ and $s[o-1] > s[o]$ (o does not necessarily exist). If o does not exist, If_rle_esc behaves as If_rle and outputs the pair $\langle k, \ell \rangle$. If o exists, If_rle_esc chooses the most economical option between 1) encoding $\langle k, \ell \rangle$ and 2) escaping σ_i (which means encoding the pair $\langle k, \mathsf{esc} \rangle$) and re-entering it at the position o. It is possible to re-enter at o since the condition $s[o-1] > s[o]$ implies that when the decoder reaches the position o it will need to

read new data from the compressed file[3]. The code for re-entering is the triple $\langle \mathsf{esc}, \ell' + 1, \sigma_i \rangle$, where ℓ' is the number of characters greater than σ_i in $s[o] \cdots s[p-1]$: we encode $\ell' + 1$ since it is possible that $\ell' = 0$. Note however that $\ell' + 1$ is never larger than the value ℓ that would have been written if we had not escaped. After reading the re-enter triple $\langle \mathsf{esc}, \ell' + 1, \sigma_i \rangle$, the decoder sets *To_be_written*$[i] = 0$ and *To_be_skipped*$[i] = \ell'$ and reads the next pair from the compressed file (that would be the *To_be_written*, *To_be_skipped* pair for $s[o]$ unless there is another re-enter sequence).

Example 5 Consider the string $s = \cdots \sigma_1 \sigma_2^2 \sigma_3^3 \sigma_4^n \sigma_2^4 \sigma_3 \sigma_1 \sigma_2^5 \cdots$ over the alphabet $\Sigma = \{\sigma_1, \sigma_2, \sigma_3, \sigma_4\}$. If n is sufficiently large If_rle_esc escapes the characters σ_1 and σ_3 and produces the output

$$\cdots \underbrace{\langle 1, \mathsf{esc} \rangle}_{\sigma_1} \underbrace{\langle 2, n+3 \rangle}_{\sigma_2^2} \underbrace{\langle 3, \mathsf{esc} \rangle}_{\sigma_3^3} \underbrace{\langle \mathsf{esc}, 6, \sigma_1 \rangle}_{\sigma_1 \text{ re-enter}} \underbrace{\langle \mathsf{esc}, 1, \sigma_3 \rangle}_{\sigma_3 \text{ re-enter}} \underbrace{\langle 4, 1 \rangle}_{\sigma_2^4} \cdots$$

(recall σ_4's occurrences are not explicitly encoded). Notice If_rle_esc cannot escape σ_2 since between the runs σ_2^2 and σ_2^4 there is no position o such that $s[o-1] > s[o]$. ∎

Notice If_rle_esc does not distinguish between cases in which $\langle k, \mathsf{esc} \rangle$ indicates a character does not occur again, as in If_rle, and cases in which it indicates an escape sequence: the former is seen as an escape without matching re-enter. Note also that the decoder can always distinguish a re-enter sequence from a normal pair $\langle k, \ell \rangle$, an escape/end-of-character pair $\langle k, \mathsf{esc} \rangle$, and an end-of-file pair $\langle \mathsf{esc}, \mathsf{esc} \rangle$.

We define If_rle_esc + $\mathsf{Pfx}^{(\delta)}$ as the algorithm that encodes the output of If_rle_esc with $\mathsf{Pfx}^{(\delta)}$. For the analysis of If_rle_esc + $\mathsf{Pfx}^{(\delta)}$ we need two preliminary lemmas.

Lemma 6.1 *Let z be a binary string of the form $z = 0^{\ell_1} 1^{\ell_2} \cdots \sigma^{\ell_m}$, where $\sigma = 0$ if m is odd, and $\sigma = 1$ if m is even. Define*

$$RLE(z) = \sum_{i=1}^{m} |\mathsf{Pfx}^{(\delta)}(\ell_i)|;$$

if $|\mathsf{Pfx}^{(\delta)}(\ell)| \leq (1+\delta)(a \log \ell) + b$ as in Lemma 5.5, then setting $a' = a(1+\delta)$ we have

$$RLE(z) \leq a'|z|H_0(z) \ + \ a' \log \ell_m + \ bm.$$

Proof: Assume 1 is the less frequent symbol (otherwise the proof is symmetrical) and let n_1 denote the number of occurrences of 1 in z. We distinguish three cases according to the size of n_1.

Case $n_1 = 0$. We have $m = 1$, $z = 0^{\ell_1}$ and $RLE(z) = |\mathsf{Pfx}^{(\delta)}(\ell_1)| \leq a' \log \ell_1 + b$.

Case $1 \leq n_1 \leq (|z|/e)$. We prove that $RLE(z) = \sum_{i=1}^{m} |\mathsf{Pfx}^{(\delta)}(\ell_i)| \leq a'|z|H_0(z) + bm$ assuming that m is even. If m is odd, the cost $|\mathsf{Pfx}^{(\delta)}(\ell_m)|$ of the last run is accounted for explicitly in the statement of the lemma. We have

$$RLE(z) = \sum_{i=1}^{m} |\mathsf{Pfx}^{(\delta)}(\ell_i)| \leq \sum_{i=1}^{m} a' \log \ell_i \ + \ bm. \tag{14}$$

Let t denote the number of nonzero logarithms in (14) (that is, we do not count the logarithms for which $\ell_i = 1$). We show that $t \leq n_1$ by charging each nonzero logarithm to a different 1 in z as follows. For $k = 1, \ldots, m/2$, if $\ell_{2k} > 1$ we charge both $\log(\ell_{2k-1})$ and $\log(\ell_{2k})$ to the the ones in $1^{\ell_{2k}}$; if $\ell_{2k} = 1$ then $\log(\ell_{2k})$ is zero and we charge $\log(\ell_{2k-1})$ to the single 1 in $1^{\ell_{2k}}$. Using Jensen's inequality and the fact that the function $x \log(|z|/x)$ is increasing for $x \leq n_1 \leq (|z|/e)$ we get

$$\sum_{i=1}^{m} \log(\ell_i) = \sum_{\ell_i > 1} \log(\ell_i) \leq t \log\left(\frac{\sum_{\ell_i > 1} \ell_i}{t}\right) \leq t \log(|z|/t) \leq n_1 \log(|z|/n_1) \leq |s| H_0(s).$$

Combining the above inequality with (14) yields the thesis.

[3]Let $s[o] = \sigma_e$. The decoder cannot output $s[o-1] > \sigma_e$ unless *To_be_written*$[e] = 0$. This implies that in order to write $s[o] = \sigma_e$ the decoder needs to read a new *To_be_written*$[e]$ value from the compressed file.

CASE $(|z|/e) < n_1 \leq (|z|/2)$. From Jensen's inequality we get

$$\sum_{i=1}^{m} |\mathsf{Pfx}^{(\delta)}(\ell_i)| = \sum_{i=1}^{m} a' \log \ell_i \; + \; bm \; \leq \; a' m \log(|z|/m) \; + \; bm$$

Since the function $x \log(|z|/x)$ has its maximum for $x = (|z|/e)$ the above inequality becomes

$$\sum_{i=1}^{m} |\mathsf{Pfx}^{(\delta)}(\ell_i)| \leq a'|z|(\log e)/e + bm.$$

The lemma follows since the hypothesis $(|z|/2) \geq n_1 > (|z|/e)$ implies $|z|H_0(z) \geq |z|(\log e)/e$. ∎

Lemma 6.2 *For $i = 1, 2, \ldots, h-1$ let $z^{(i)}$ denote the binary string obtained from s deleting all characters smaller than σ_i, replacing the occurrences of σ_i with $\mathsf{1}$, and replacing the occurrences of characters greater than σ_i with $\mathsf{0}$. We have*

$$\sum_{i=1}^{h-1} |z^{(i)}| H_0(z^{(i)}) = |s| H_0(s). \tag{15}$$

Proof: For $i = 1, 2, \ldots, h$ let n_i denote the number of occurrences of σ_i in s and let $w_i = n_i + \cdots + n_h$. Note that $|z^{(i)}| = n_i + w_{i+1} = w_i$. We have

$$\begin{aligned}
\sum_{i=1}^{h-1} |z^{(i)}| H_0(z^{(i)}) &= \sum_{i=1}^{h-1} [n_i \log(w_i/n_i) + w_{i+1} \log(w_i/w_{i+1})] \\
&= \sum_{i=1}^{h-1} [w_i \log(w_i) - n_i \log(n_i) - w_{i+1} \log(w_{i+1})] \\
&= w_1 \log(w_1) - \sum_{i=1}^{h-1} n_i \log(n_i) \; - \; w_h \log(w_h) \\
&= |s| \log |s| - \sum_{i=1}^{h} n_i \log(n_i) = |s| H_0(s). \quad ∎
\end{aligned}$$

Theorem 6.3 *For every $\epsilon, \delta > 0$ the algorithm $\mathsf{A}_3 = \mathsf{If\_rle\_esc} + \mathsf{Pfx}^{(\delta)}$ is locally λ-optimal for $\lambda = \max(4a', a' + b + \epsilon)$, where $a' = a(1+\delta)$.*

Proof: We need to show that for any partition $s = s_1 s_2 \cdots s_t$ we have

$$|\mathsf{A}_3(s)| \leq \max(4a', a' + b + \epsilon) \sum_{i=1}^{t} |s_i| H_0^*(s_i) + O(th \log h)\,. \tag{16}$$

We consider the algorithm **If_rle_esc*** that, instead of choosing at each step whether to escape or not, escapes the symbol σ_i only if the characters $s[m]$ and $s[p]$ belongs to two different partition elements (recall that whether **If_rle_esc*** actually escapes depends on the existence of a position o, such that $m < o < p$ and $s[o-1] > s[o]$). Let $\mathsf{A}_3^* = \mathsf{If\_rle\_esc*} + \mathsf{Pfx}^{(\delta)}$. Since **If_rle_esc** always performs the most economical choice, we have $|\mathsf{A}_3(s)| \leq |\mathsf{A}_3^*(s)|$. We prove the theorem by showing that (16) holds with $\mathsf{A}_3(s)$ replaced by $\mathsf{A}_3^*(s)$.

For $i = 1, \ldots, h-1$, let $s^{(i)}$ denote the string obtained removing from s the characters smaller than σ_i. If in $s^{(i)}$ we replace σ_i with $\mathsf{1}$ and $\sigma_{i+1}, \ldots, \sigma_h$ with $\mathsf{0}$ we get precisely the string $z^{(i)}$ defined in Lemma 6.2. Let RLE be defined as in Lemma 6.1. We preliminary prove that

$$|\mathsf{A}_3^*(s)| \leq \sum_{i=1}^{h-1} RLE(z^{(i)}) \; + \; O(th \log h)\,. \tag{17}$$

To prove this, we first consider lf_rle and we show that, apart from $O(h)$ bits at Step 1, its output consists precisely of the lengths of the runs of zeros and ones in $z^{(i)}$ for $i = 1, \ldots, h-1$. Consider for example the encoding of the character σ_i. At Step 2 lf_rle encodes the number ℓ_i of characters greater than σ_i preceding the first occurrence of σ_i in s: this is precisely the length of the first run of 0's in $z^{(i)}$. Then, each pair $\langle k, \ell \rangle$ written at Step 3 represents a run of σ_i in $s^{(i)}$—corresponding to a runs of 1's in $z^{(i)}$—followed by a run of characters greater than σ_i—corresponding to a run of 0's in $z^{(i)}$. For the algorithm lf_rle_esc* the only difference is that, because of the escape mechanism, a length-ℓ run of zeros[4] crossing the boundary of two partition elements is sometimes encoded with an escape pair $\langle k, \mathsf{esc} \rangle$ later followed by the re-enter triple $\langle \mathsf{esc}, \ell' + 1, \sigma_i \rangle$. Since lf_rle_esc* escapes at most th times, the parts of an escape/re-enter sequence that use $O(1)$ bits are included in the $O(th \log h)$ term. Since, by construction, we have $1 + \ell' \leq \ell$ we conclude that (17) holds.

For $i = 1, \ldots, h-1$, the partition $s = s_1 \cdots s_t$ naturally induces the partitions $s^{(i)} = s_1^{(i)} \cdots s_t^{(i)}$ and $z^{(i)} = z_1^{(i)} \cdots z_t^{(i)}$. By Lemma 3.1 we get

$$RLE(z^{(i)}) \leq \sum_{j=1}^{t} RLE(z_j^{(i)}) \; + \; O(t)\,. \tag{18}$$

Assume now that every partition element s_j contains every character $\sigma_i \in \Sigma$. Under this assumption $H_0(z_j^{(i)}) = H_0^*(z_j^{(i)})$. By Lemmas 6.1 and 2.3, for any $\epsilon > 0$ we have

$$RLE(z_j^{(i)}) \leq \max(2a', a' + b + \epsilon)|z_j^{(i)}|H_0(z_j^{(i)}) + O(1). \tag{19}$$

Combining (17) with (18) and (19) and Lemma 6.2 we get

$$\begin{aligned}
|\mathsf{A}_3^*(s)| &\leq \sum_{i=1}^{h-1}\sum_{j=1}^{t} RLE(z_j^{(i)}) \; + \; O(th \log h) \\
&\leq \sum_{i=1}^{h-1}\sum_{j=1}^{t} \max(2a', a' + b + \epsilon)|z_j^{(i)}|H_0(z_j^{(i)}) \; + \; O(th \log h) \\
&\leq \max(2a', a' + b + \epsilon) \sum_{j=1}^{t}\sum_{i=1}^{h-1} |z_j^{(i)}|H_0(z_j^{(i)}) \; + \; O(th \log h) \\
&\leq \max(2a', a' + b + \epsilon) \sum_{j=1}^{t} |s_j|H_0(s_j) \; + \; O(th \log h)
\end{aligned}$$

that implies (16). In the general case, in which some s_j does not contain all the characters of Σ, the above argument does not hold. To see this, assume that s_j does not contain σ_i but does contain a character σ_k with $k > i$. In this case we have $z_j^{(i)} = \mathsf{0}^\ell$, hence $H_0(z_j^{(i)}) = 0$ and (19) does not hold.

To complete the proof we need to take into account the benefits of the escape mechanism. We say that a character σ_i is *escaped* in s_j if the following conditions hold: 1) $z_j^{(i)} = \mathsf{0}^\ell$, 2) the last run of σ_i's before the beginning of s_j produces and escape sequence, 3) the corresponding re-entry point is at a position $s[o]$ which is after the end of s_j. The crucial observation is that if σ_i is escaped in s_j the algorithm lf_rle_esc* does not "pay" for the encoding of $z_j^{(i)}$. To see this, observe that $z_j^{(i)} = \mathsf{0}^\ell$ is a substring of a larger run $\mathsf{0}^m$ in $z^{(i)}$. Because of the escape mechanism instead of the length m lf_rle_esc* encodes a length m' with $m' \leq m - \ell$ so $z_j^{(i)}$ is encoded essentially for free. For this reason, if we define $U_j \subseteq \Sigma$ as the set of

[4]Only runs of zeros are escaped by lf_rle_esc*. Indeed, when we escape σ_i we are trying to reduce the cost of encoding a run of characters greater than σ_i and such characters are represented by zeros in $z^{(i)}$.

characters that are not escaped in s_j, we have

$$|\mathsf{A}_3^*(s)| \leq \sum_{j=1}^{t} \sum_{i \in U_j} RLE(z_j^{(i)}) \ + \ O(th \log h) \tag{20}$$

We conclude the proof by showing that for $j = 1, \ldots, t$

$$\sum_{i \in U_j} RLE(z_j^{(i)}) \leq \max(4a', a' + b + \epsilon)|s_j|H_0^*(s_j) + O(1) \tag{21}$$

which combined with (20) proves (16). To prove (21) we consider the characters $\sigma_i \in U_j$ for which the string $z_j^{(i)}$ is non empty but $H_0(z_j^{(i)}) = 0$ (in other words $z_j^{(i)}$ is equal to $\mathsf{0}^\ell$ or $\mathsf{1}^\ell$ with $\ell > 0$).

Our first observation is that there can be at most one character $\sigma_i \in U_j$ such that $z_j^{(i)} = \mathsf{0}^\ell$. The reason is that if for both σ_i, σ_k we have $H_0(z_j^{(i)}) = \mathsf{0}^\ell$ and $H_0(z_j^{(k)}) = \mathsf{0}^m$ one of them—the one which occurs later after the end of s_j—will be certainly escaped. The second observation is that there can be at most one character σ_i such that $z_j^{(i)} = \mathsf{1}^\ell$. The reason is that $z_j^{(i)} = \mathsf{1}^\ell$ if and only if $i < h$ and σ_i is the largest character appearing in s_j. We conclude that the contribution to the summation in (21) of the strings $z_j^{(i)}$ with $H_0(z_j^{(i)}) = 0$ is bounded by $2\mathsf{Pfx}^{(\delta)}(|s_j|) \leq 2a' \log(|s_j|) + 2b$. Let $W_j \subseteq \Sigma$ be the set of characters σ_i such that $H_0(z_j^{(i)}) \neq 0$. If W_j is empty, then $\sum_{i \in U_j} RLE(z_j^{(i)}) = 2a' \log(|s_j|) + 2b$ and (21) holds. If W_j is not empty, using Lemmas 6.1 and 2.3 and the fact that for the smallest i in W_j we have $|z_j^{(i)}| = |s_j|$ we get that for any $\epsilon > 0$

$$\begin{aligned}
\sum_{i \in U_j} RLE(z_j^{(i)}) &\leq \sum_{i \in W_j} RLE(z_j^{(i)}) \ + \ 2a' \log(|s_j|) + 2b \\
&\leq \sum_{i \in W_j} \Big(a'|z_j^{(i)}|H_0(z_j^{(i)}) + a' \log(|z_j^{(i)}|) + b\,\mathsf{runs}(z_j^{(i)}) \Big) \ + \ 2a' \log(|s_j|) + 2b \\
&\leq \sum_{i \in W_j} \Big(a'|z_j^{(i)}|H_0(z_j^{(i)}) + 3a' \log(|z_j^{(i)}|) + b\,\mathsf{runs}(z_j^{(i)}) \Big) \ + \ 2b \\
&\leq \sum_{i \in W_j} \Big(\max(4a', a' + b + \epsilon)|z_j^{(i)}|H_0(z_j^{(i)}) \Big) \ + \ O(1) \\
&= \max(4a', a' + b + \epsilon)|s_j|H_0(s_j) \ + \ O(1)
\end{aligned}$$

as claimed. ∎

Repeating verbatim the proof of Theorem 5.4 with $\mu = 1.105$, $\nu = \epsilon = \delta = 0.001$, we get a bound for the output size of If_rle_esc followed by Order0*.

Theorem 6.4 *The algorithm* If_rle_esc + Order0* *is locally* $(4.45 + C_0)$*-optimal.* ∎

Corollary 6.5 *For any string* s *and* $k \geq 0$ *we have*

$$|\mathsf{Order0*}(\mathsf{If\_rle\_esc}(\mathsf{bwt}(s)))| \leq (4.45 + C_0)|s|H_k^*(s) + \log|s| + O\Big(h^{k+1} \log h\Big).$$

Proof: Immediate by Theorem 6.4 and Lemma 2.1. ∎

7 Lower bound for entropy-only compressors

In this section we show that any algorithm A achieving an "entropy-only" bound cannot compress every string s in less than $2|s|H_0^*(s) + \Theta(1)$ bits. We prove this result assuming only that A is non-singular, that is, for any pair of strings $s_1 \neq s_2$ we have $A(s_1) \neq A(s_2)$.

Theorem 7.1 *If A is a non-singular compressor, then the bound*

$$|A(s)| \leq \lambda |s| H_0^*(s) + \eta \qquad \textit{for every string } s$$

can only hold with a constant $\lambda \geq 2$.

Proof: For $i = 1, 2, \ldots$ let T_i denote the set of binary strings such that $s \in T_i$ if and only if $2^{i-1} < |s| \leq 2^i$ and s contains exactly one 1 and $(|s| - 1)$ 0's. Elementary calculus shows that

$$|T_i| = \frac{2^i(2^i+1)}{2} - \frac{2^{i-1}(2^{i-1}+1)}{2} \geq \frac{3}{8} \cdot 4^i. \tag{22}$$

In addition, recalling that $t \geq 1$ implies $(1 + \frac{1}{t})^t < e$, for $s \in T_i$ it is

$$\begin{aligned} |A(s)| &\leq \lambda |s| H_0^*(s) + \eta \\ &= \lambda \left(\log |s| + (|s| - 1) \log \left(\tfrac{|s|}{|s|-1} \right) \right) + \eta \\ &\leq \lambda (\log 2^i + \log e) + \eta \\ &= \lambda\, i + \eta' \end{aligned} \tag{23}$$

with $\eta' = \eta + \lambda \log e$. Since there are at most $2^{z+1} - 1$ distinct binary codewords of length at most z, we have that less than

$$2^{\lambda i + \eta' + 1} = 2^{\eta'+1} (2^\lambda)^i \tag{24}$$

are available for encoding the strings in T_i. Comparing (22) and (24) implies that, for sufficiently large i, if every $s \in T_i$ must get a different codeword we must have $2^\lambda \geq 4$ and therefore $\lambda \geq 2$. ∎

If, addition to non-singularity, we require that A be prefix-free, that is for $s_1 \neq s_2$, $A(s_1)$ is not a prefix of $A(s_2)$, then also a multiplicative constant $\lambda = 2$ is not admissible.

Theorem 7.2 *If A is a prefix-free compressor, then the bound*

$$|A(s)| \leq \lambda |s| H_0^*(s) + \eta \qquad \textit{for every string } s$$

can only hold with a constant $\lambda > 2$.

Proof: Let T_i be defined as in the proof of Theorem 7.1. Since A defines a prefix free encoding of the strings in the set $\cup_{i \geq 1} T_i$, by the extended Kraft's inequality [5, Theorem 5.2.2] we must have

$$\sum\nolimits_{i \geq 1} \sum\nolimits_{s \in T_i} 2^{-|A(s)|} \leq 1. \tag{25}$$

By (23) and (25) we get

$$\sum\nolimits_{i \geq 1} \sum\nolimits_{s \in T_i} 2^{-|A(s)|} \;\geq\; \sum\nolimits_{i \geq 1} 2^{-\lambda i - \eta'} |T_i| \;\geq\; \sum_{i > 0} 2^{-\eta'} \frac{3}{8} \left(\frac{4}{2^\lambda} \right)^i.$$

Hence, to satisfy (25) we must have $\lambda > 2$ as claimed. ∎

References

[1] Z. Arnavut and S. Magliveras. Block sorting and compression. In *Procs of IEEE Data Compression Conference (DCC)*, pages 181–190, 1997.

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

[3] E. Binder. Distance coder, 2000. Usenet group: comp.compression.

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

[5] T. M. Cover and J. A. Thomas. *Elements of Information Theory*. Wiley Interscience, 1990.

[6] S. Deorowicz. Second step algorithms in the Burrows-Wheeler compression algorithm. *Software: Practice and Experience*, 32(2):99–111, 2002.

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

[8] P. Fenwick. Burrows-Wheeler compression with variable length integer codes. *Software: Practice and Experience*, 32:1307–1316, 2002.

[9] P. Ferragina, R. Giancarlo, and G. Manzini. The engineering of a compression boosting library: Theory vs practice in BWT compression. In *Proc. 14th European Symposium on Algorithms (ESA '06)*, pages 756–767. Springer Verlag LNCS n. 4168, 2006.

[10] P. Ferragina, R. Giancarlo, and G. Manzini. The myriad virtues of wavelet trees. In *Proc. 33th International Colloquium on Automata and Languages (ICALP '06)*, pages 561–572. Springer Verlag LNCS n. 4051, 2006.

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

[12] L. Foschini, R. Grossi, A. Gupta, and J. Vitter. Fast compression with a static model in high-order entropy. In *Procs of IEEE Data Compression Conference (DCC)*, pages 62–71, 2004.

[13] R. Giancarlo and M. Sciortino. Optimal partitions of strings: A new class of Burrows-Wheeler compression algorithms. In *Proc. 14th Symposium on Combinatorial Pattern Matching (CPM '03)*, pages 129–143. Springer-Verlag LNCS n. 2676, 2003.

[14] H. Kaplan, S. Landau, and E. Verbin. A simpler analysis of Burrows-Wheeler based compression. *Theoretical Computer Science*, 387:220–235, 2007.

[15] V. Mäkinen and G. Navarro. Succinct suffix arrays based on run-length encoding. *Nordic Journal of Computing*, 12(1):40–66, 2005.

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

[17] G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM Computing Surveys*, 39(1), 2007.

[18] B. Y. Ryabko. Data compression by means of a 'book stack'. *Prob.Inf.Transm*, 16(4), 1980.