The power of Algorithms!

Lectures on algorithmic pearls

Paolo Ferragina, Università di Pisa

These notes have been taken by the master students of the course on "Algorithm Engineering" within the Laurea Magistrale in Informatica and Networking.

P.F.

Contents

1

From BWT to bzip

Alessandro Adamou
adamou AT cs unibo it

Paolo Parisen Toldin
paolo.parisentoldin AT gmail com

The following notes describe a lossless text compression technique devised by Michael Burrows and David Wheeler in 1994 at the DEC Systems Research Center. This technique, which combines simple compression algorithms with an input transformation algorithm (the so called Burrows-Wheeler Transform, or BWT), offered a revolutionary alternative to dictionary-based approaches and is currently employed in widely used compressors such as *bzip2*.

These notes are structured as follows: we begin by describing the algorithm both for the forward transform and for the backward transform in Section 1.1. As we will show, the BWT alone is not a compression algorithm, so two simple compressors, i.e. Move-To-Front and Run-Length Encoding, are introduced in Section 1.2. How these algorithms can be combined for boosting textual compression is discussed with examples in Section 1.3. Section 1.4 introduces the notion of entropy and discusses the performance of the BWT with respect to this notion. The notes conclude with a description of experiments that compare the implementation of BWT in the bzip2 compressor with several implementations of dictionary-based Lempel-Ziv algorithm variants, along with a discussion on the findings.

1.1 The Burrows-Wheeler transform

The **Burrows-Wheeler Transform** (**BWT**) [4] is not a compression algorithm *per se*, as it does not tamper with the size, or the encoding, of its input. What the BWT *does* perform is what we call *block-sorting*, in that the algorithm processes its input data by reordering them block-wise (i.e. the input is divided into blocks before being processed) in a convenient manner. What a "convenient manner" is, in the case of the BWT, will be shown later, but for the time being, suffice it to say that the resulting output lends itself to greater and more effective compression by simple algorithms. Two such algorithms are the *Move-To-Front* optionally followed by *Run Length Encoding*, both to be described in section 1.2.

The Burrows-Wheeler Transform consists of a pair of inverse operations; a *forward transform* rearranges the input text so that the resulting form shows a degree of "locality" in the symbol ordering; a *backward transform* reconstructs the original input text, hence being the reverse operation of the forward transform.

1.1.1 Forward Transform

Let $s = s_1 s_2 ... s_n$ be an input string on n characters drawn from an alphabet Σ, with a total ordering defined on the symbols in Σ.

Given s, the forward transform proceeds as follows:

1. Let $s\$$ be the string resulting from the concatenation of s with a string consisting of a single occurrence of symbol $\$$, where $\$ \notin \Sigma$ and $\$$ is smaller than any other symbol in the alphabet, according to the total ordering defined on Σ.[1]
2. Consider matrix $\mathcal{M}$ of size $(n+1)\times(n+1)$, whose rows contain all the cyclic left shifts of string s. $\mathcal{M}$ is also called the *rotation matrix* of s.
3. Let $\mathcal{M}'$ be the matrix obtained by sorting the rows of $\mathcal{M}$ *left-to-right* according to the ordering defined on alphabet $\Sigma \cup \{\$\}$. Recall that $\$$ is smaller than any other symbol in Σ and, by construction, appears only once, therefore the sort operation will always move the last row from position $n+1$ to position 0.
4. Let F and L be the first and last columns of $\mathcal{M}'$ respectively. Let $\hat{l}$ be the string obtained by reading column L top-to-bottom, sans character $\$$. Let r be the index of $\$$ in L.
5. Take $bw(s) = (\hat{l}, r)$ as the output of the algorithm.

Note that $\mathcal{M}$ is a conceptual matrix, in that there is no actual need to build it entirely: only the first and last columns are required for applying the forward transform. Section 1.3.1 will show an example of BWT application that does not require the rotation matrix to be actually constructed in full.

Remark

An alternate enunciation of the algorithm, less frequent yet still present in the literature [5], constructs matrix $\mathcal{M}'$ by sorting the rows of $\mathcal{M}$ *right-to-left* (i.e. starting from character at index n) in step 3. Then, in step 4, it takes string $\hat{f}$ instead of $\hat{l}$, where $\hat{f}$ is obtained by reading column F top-to-bottom skipping character $\$$, and sets r as the index of $\$$ in F instead of L. The output is then $bw(s) = (\hat{f}, r)$. This enunciation is dual to the one given in the following sense: although its output differs from the one generated by *left-to-right* sorting, it showcases the same properties of reversibility and compression, to be illustrated below, as the former.

Example 1.1

We will now show an example of Burrows-Wheeler Transform applied to an input block which, historically, is known to transform to a highly compressible string.

[1] The step that concatenates character $\$$ to the initial string was not part of the original version of the algorithm as described by Burrows and Wheeler. It is here introduced with the intent to simplify the enunciation. Due to its role, $\$$ is sometimes called a *sentinel character*.

Let us have $s =$ `abracadabra`. Figure 1.1 shows, on its left side, how matrix $\mathcal{M}$ is constructed for the given input. Starting from row 0, which reads `abracadabra$` i.e. the original string concatenated with `$`, each subsequent row contains the same string left-shifted by one.

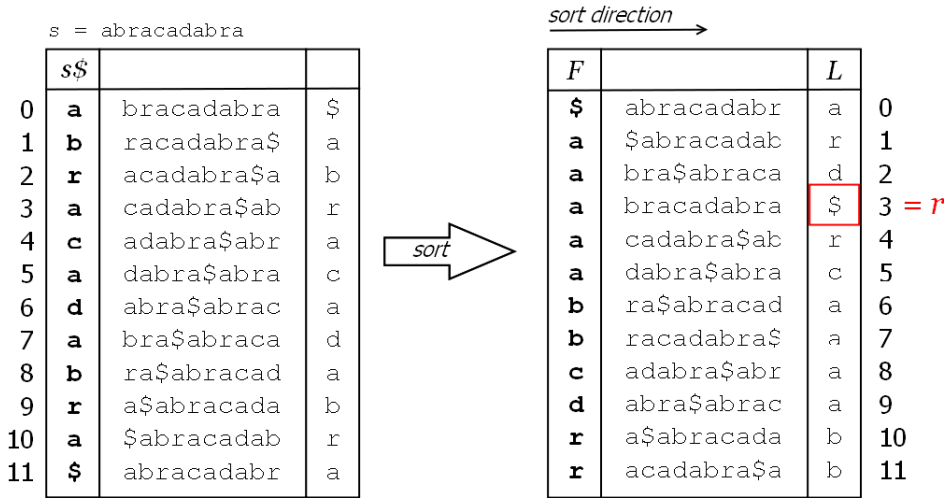

FIGURE 1.1: Example of forward Burrows-Wheeler transform applied to the input block $s =$ `abracadabra$`, resulting in the transformed string `ardrcaaaabb`. The cyclic shift ending with `$`, assigned to variable r, is placed at index 3 in the sorted matrix $\mathcal{M}'$.

The rows of $\mathcal{M}$ are then sorted lexicographically from left to right, thus obtaining matrix $\mathcal{M}'$. Because the last row of $\mathcal{M}$ (row 11, which reads `$abracadabra`) is the only one to begin with `$`, which is the lowest-ordered symbol in the (extended) alphabet, it will then become the first row of $\mathcal{M}'$. The next five rows will be the ones beginning with `a` (in the given order: `a$abracadabr`, `abra$abracad`, `abracadabra$`, `acadabra$abr` and `adabra$abrac`), then the two shifts beginning with `b` (i.e. `bra$abracada` and `bracadabra$a`), and so on. The order in which the rows starting with the same character appear in $\mathcal{M}'$ has an extremely important property, which will be shown when discussing the reverse transform in the next section.

The sorted matrix $\mathcal{M}'$ is shown on the right side of Figure 1.1. If we read the first column F of $\mathcal{M}'$, we obtain the string `$aaaaabbcdrr`, which is itself sorted lexicographically, while the last column L of $\mathcal{M}'$ reads `ard$rcaaaabb`. We therefore obtain $\hat{l}$ by excluding the single occurrence of `$`, thus having $\hat{l} =$ `ardrcaaaabb`. Since `$` appears as the fourth element of L, which is of length $n = 12$, we take $r = 3$. We finally obtain $bw(s) =$ (`ardrcaaaabb`, 3).

Notice how both occurrences of `b` end up juxtaposed in the transformed string, as are four of the five occurrences of `a`. The following chapters in this lecture will show how this layout favors text compression.

1.1.2 Backward Transform

We can observe, both by construction and from the example provided, that each column of the sorted cyclic shift matrix $\mathcal{M}'$ contains a permutation of $s\$$. In particular, its first column F represents the best-compressible transformation of the original input block. However, it would not be possible to determine how the transformed string should be reversed, even by knowing the distribution of its alphabet Σ. The Burrows-Wheeler transform represents a trade-off which retains some properties that render a transformed string reversible, yet still remaining highly compressible.

In order to prove these properties more formally, let us define a useful function that tells us how to locate in $\mathcal{M}'$ the predecessor of a character at a given index in s.

DEFINITION 1.1 For $1 \leq i \leq n$, let $s[k_i, n-1]$ denote the suffix of s prefixing row i of $\mathcal{M}'$. Let $\Psi(i)$ be the index of the row prefixed by $s[k_i + 1, n-1]$.

The function $\Psi(i)$ is not defined for $i = 0$, because row 0 of $\mathcal{M}'$ begins with \$, which does not occur in s at all, so that row cannot be prefixed by a suffix of s.

For example in Figure 1.1 it is $\Psi(2) = 6$. Row 2 of $\mathcal{M}'$ is prefixed by `abra`, followed by \$. We look for the one row of $\mathcal{M}'$ that is prefixed by the suffix of s that is immediately shorter (`bra`), and find that row 6 (which reads `bra$abracada`) is the one. Similarly, $\Psi(11) = 4$ since row 11 of $\mathcal{M}'$ is prefixed by `racadabra` and row 4 is the one prefixed by `acadabra`.

Property 1

For each i, $L[i]$ is the immediate predecessor of $F[i]$ in s. Formally, for $1 \leq i \leq n$, $L[\Psi(i)] = F[i]$.

Proof. Since each row of $\mathcal{M}'$ contains a cyclic shift of $s\$$, the last character of the row prefixed by $s[k_i + 1, n-1]$ is $s[k_i]$. Because the index of that row is what we have defined to be $\Psi(i)$ in Definition 1.1, this implies the claim $L[\Psi(i)] = s[k_i] = F[i]$.

Intuitively, this property descends from the very nature of every row in $\mathcal{M}$ and $\mathcal{M}'$ that is a cyclic shift of the original string concatenated with \$, so if we take two extremes of each row, the symbol on the right extreme is immediately followed by the one on the left extreme. For index j such that $L[j] = \$$, $F[j]$ is the first character of s.

Property 2

All the occurrences of a same symbol in L maintain the same relative order as in F. Using function Ψ, if $1 \leq i < j \leq n$ and $F[i] = F[j]$ then $\Psi(i) < \Psi(j)$. This means that the first occurrence of a symbol in L maps to the first occurrence of that symbol in F, its second occurrence in L maps to its second occurrence in F, and so on, regardless of what other symbols separate two occurrences.

Proof. Given two strings s and t, we shall use the notation $s \prec t$ to indicate that s lexicographically precedes t.

Let $s[k_i, n-1]$ (resp. $s[k_j, n-1]$) denote the suffix of s prefixing row i (resp. row j). The hypothesis $i < j$ implies that $s[k_i, n-1] \prec s[k_j, n-1]$. The hypothesis $F[i] = F[j]$ implies $s[k_i] = s[k_j]$ hence it must be $s[k_i + 1, n-1] \prec s[k_j + 1, n-1]$. The thesis follows since by construction $\Psi(i)$ (resp. $\Psi(j)$) is the lexicographic position of the row prefixed by $s[k_i + 1, n-1]$ (resp. $s[k_j + 1, n-1]$).

In order to re-obtain s, we need to map each symbol from the L column of $\mathcal{M}'$ to its corresponding occurrence in the sorted column F. This way, we are able to obtain the permutation of rows that generated matrix $\mathcal{M}'$ from $\mathcal{M}$. This operation is what we call

LF-mapping, and it is essentially analogous to determining the values of function Ψ for matrix $\mathcal{M}'$. The process is rather straightforward for symbols that occur only once, as is the case of `$`, `c` and `d` in `abracadabra$`. However, when it comes to symbols `a`, `b` and `r`, which occur several times in the string, the problem is no longer trivial. It can however be solved thanks to the properties we proved [7], which hold for transformed matrix $\mathcal{M}'$.

Let us now illustrate a simple algorithm for performing the LF-mapping, which will be the starting point for the actual inverse transformation algorithm to be shown later in this section. The LF-mapping will be stored in an array $LF, ||LF|| = n$.

The algorithm uses an auxiliary vector C, with $||C|| = |\Sigma \cup \{\$\}|$. For each symbol c, C stores the total number of occurrences in L of symbols *smaller than* c. Vector C is indeed indexed by *symbol* rather than by integer[2].

Given this initial content of C, the LF-mapping is performed as follows.

```
1    for (int i=0; i<n; i++) {
2     LF[i] = C[L[i]];
3     C[L[i]]++;
4    }
```

This algorithm scans the entire transformed column L (lines 1,4). When a symbol is encountered in the process, the algorithm checks the value of the element in C having that symbol as a key. That value is assigned to the LF element with the same index as the one of L (line 2). Having found an occurrence of that symbol, the counter expressed by the value of LF having that symbol as a key is incremented by one (line 3). This means that vector C contains, for each symbol c and at a given instant, the number of occurrences smaller than *or equal to* symbol c found *so far*. Therefore, once the algorithm has completed, vector C will contain, for each c, the *total* number of occurrences smaller than or equal to c, while array LF will contain the LF-mapping that can be used straight away for applying the backward transform algorithm we will now illustrate.

Given the LF-mapping algorithm and the fundamental properties shown earlier, we are able to reconstruct s backwards starting from the transformed output $bw(s)$, which we remind to be consisting of a permuted s plus an index r. Having $bw(s)$ is equivalent to having the L column, since the latter can simply be obtained by placing symbol `$` at the r-th position and right-shifting the remainder by one. Before showing the algorithm itself, it can be helpful to exemplify it.

Example 1.2

Let us show how the to re-obtain the original input block starting from its BWT-transformed block (`ardrcaaaabb`, 3).

Since $r = 3$, we insert the sentinel character `$` at position 3 and right-shift the remainder by one. We obtain the last column L of $\mathcal{M}'$, which reads `ard$rcaaaabb` top-to-bottom. By sorting this column lexicographically, we obtain column F, which reads `$aaaaabbcdrr`. *There will be no need to compute any other permutations in* $\mathcal{M}'$.

We start reconstructing s\$ from the last character `$`. Its index in F is 0, and $L[0] =$ `a`, so we append this character to the end of the string we are rebuilding, thus obtaining `a$`.

[2]By custom, the term "array" is used when referring to the linear data structures whose elements are indexed by *integer ranges*, while the term "vector" is used for structures whose indices belong to arbitrary sets, as is the case of C in this lecture.

$L[0]$ contains the first occurrence of `a`, so we also locate the first occurrence of `a` in F (by Property 2 we know that this relative order is preserved). Its index in F is 1 and $L[1] =$ `r`. The rebuilt string now ends with `ra$`.

$L[1]$ contains the first occurrence of `r`, and its index in F is 10. Since $L[10] =$ `b`, we append it to the output and obtain `bra$`.

Note that this process of locating in L the predecessor of each character is equivalent to determining the values of function Ψ in $\mathcal{M}'$, albeit without having to know the prefixes of each row of $\mathcal{M}'$.

By following this rationale, we eventually rebuild the entire string to `abracadabra$`, and by stripping the sentinel character, the original input `abracadabra` is reconstructed.

Having intuitively demonstrated how it works, let us now illustrate and explain the algorithm itself. An auxiliary array T, $||T|| = ||L||$ is used for storing the output of the algorithm.

```
1    Compute LF[0,n-1];
2    k = 0;
3    i = n;
4    while (i>0) {
5     T[i]=L[k];
6     k = LF[k];
7     i--;
8    }
```

Given the array LF obtained by LF-mapping (line 0), we begin scanning this array (lines 2,4) starting from the first element. Due to Property 1 and the fact that F is sorted starting from symbol `$`, the first symbol in L is the final symbol of the original string, and $LF[0]$ tells us the index in L of its immediate predecessor.

We fill the new array T, starting from its last element (lines 3,5) and proceeding backwards (line 7). The array LF is not scanned linearly, since every subsequent index to be scanned both in L (for reading the actual symbol) and from LF (for knowing the next index) is determined by the value of LF at the previously found index.

Once the algorithm has returned, we can obtain the string s`$` by reading the content of T left-to-right. By stripping the last symbol, the original input string is reconstructed.

1.2 Two simple compressors: MTF and RLE

Let us now focus on two simple data compression algorithms that come in very useful, now that the BWT has been described in detail, in order to build up the final bzip2 compressor. These compression algorithms are respectively called **Move-To-Front** (**MTF**) and **Run-Length Encoding** (**RLE**).

1.2.1 Move-To-Front Transform

The MTF [3] (Move-To-Front) transformation is a technique used to improve the performance of entropy encoding. It is based on the idea that every character is replaced with its index in list of the given alphabet.

Given a string s and an alphabet Σ, the algorithm produces a string s^{MTF} on the alphabet $\Sigma^{MTF} = \{0, \ldots, |\Sigma| - 1\}$.

At the beginning, a list l is initialized to the alphabet Σ and s^{MTF} to the empty string. On each step, we consider, in order, a simbol σ in s and add the index i of σ in l to the string s^{MTF}. Then we modify l by *moving* the element σ to the *front* of the list.

In the example presented in figure 1.2 we consider the string "`bananacocco`" and we show all the steps needed to encode it with MTF.

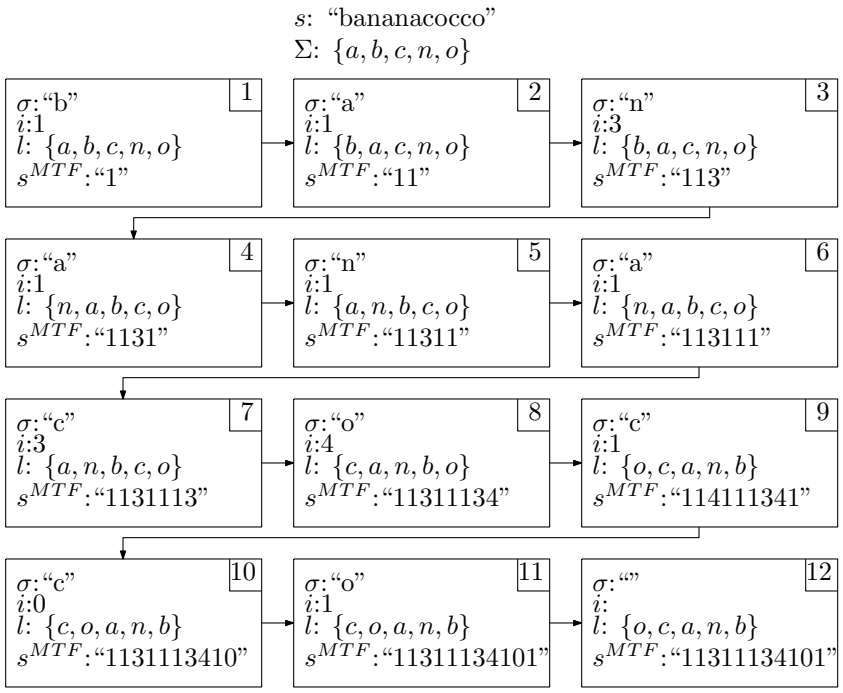

FIGURE 1.2: MTF example

As we can infer from the example 1.2, MTF assigns codes with lower values to symbols that are more frequent [5]. We can also notice two local homogeneity in the considered string: "`banana`" and "`cocco`". Indeed, these two substrings show redundancy in a few symbols only. For this reason, the output string contains numbers with low value at homogeneity. Higher values suggest a change of homogeneity.

The reverse operation is easy to compute. Having the initial list l we can obtain s by performing the reverse steps in MTF, as shown in figure 1.3

Theoretical analysis

If the message exhibits locality of references, MTF performs better than Huffman coding because a word will have a short encoding when it is used frequently. Contrariwise if the message does not exhibits any kind of homogeneity, MTF performs worse that Huffman coding; indeed, letters used rarely are encoded with long encoding [3].

Let $\rho_{MTF}(x)$ be the avarage number of bits per words used to compress a sequence X with the MFT scheme. Let $\rho_H(X)$ the number of bits used to compress the sequence X with Huffman coding.

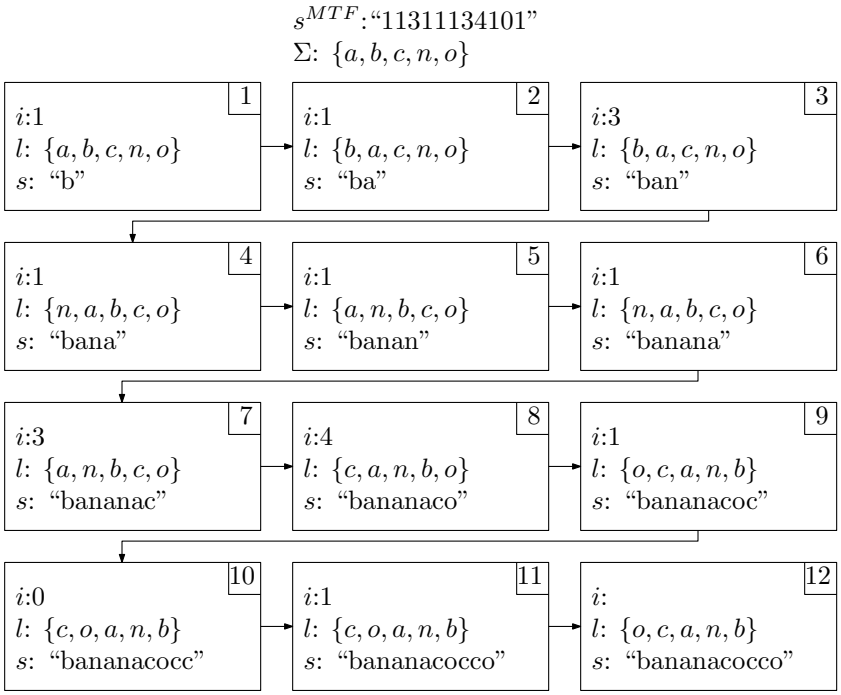

FIGURE 1.3: MTF inverted example

Let $f(i)$ be the number of bits needed to encode the integer i (every letter can be encoded in an integer number). We will use γ-coding, so $f(i) \leq 2 \times \log_2 i + 1$.

THEOREM 1.1 *Let N_a the number of occurences of a symbol a in X. $\frac{N_a}{N}$ represents the probability of symbol a.*

$$\rho_{MTF}(x) \leq \sum_{a \in S} \frac{N_a}{N} f\left(\frac{N}{N_a}\right)$$

Proof

Let $t_1, \ldots, t_{N_a}$ the times when simbol a is sent. Eg. when a occurs at time t_1, its position in the list is at most t_1. When a occurs at time t_i, with $i > 1$, its position is at most $t_i - t_{i-1}$. The cost of encoding the first a is $f(t_1)$ and the cost of encoding the i-th a is at most $f(t_i - t_{i-1})$.

Let $R_a(X)$ the total number of bits used to transmit the N_a occurences of simbol a, we have:

$$R_a(X) \leq f(t_1) + \sum_{i=2}^{N_a} f(t_i - t_{i-1})$$

Applying Jensen's inequality:

$$R_a(X) \leq N_a f\left(\frac{1}{N_a}\left(t_1 + \sum_{i=2}^{N_a}(t_i - t_{i-1})\right)\right) = N_a f\left(\frac{t_{N_a}}{N_a}\right) \leq N_a f\left(\frac{N_a}{N_a}\right)$$

If now we sum for every $a \in S$ the constrain for $R_a(X)$ and divide for N we reach the thesis.

$$\sum_{a\in S}\frac{R_a(X)}{N} \equiv \rho_{MTF}(x) \leq \sum_{a\in S}\frac{N_a}{N} f\left(\frac{N}{N_a}\right)$$

THEOREM 1.2 *We will prove the following disequation:*

$$\rho_{MTF}(x) \leq 2\rho_H(X) + 1$$

Proof We will start from the result in theorem 1.1. We substitute f with the γ-coding.

$$\begin{aligned}
\rho_{MTF}(x) &\leq \sum_{a\in S}\left(\frac{N_a}{N} f\left(\frac{N}{N_a}\right)\right) \\
&\leq \sum_{a\in S}\left(\frac{N_a}{N}\cdot 2\cdot \log\left(\frac{N}{N_a}\right) + 1\right) \\
&\leq 2\cdot -\sum_{a\in S}\left(\frac{N_a}{N}\cdot \log\left(\frac{N}{N_a}\right)\right) + O(1) \\
&\leq 2\cdot H_0(X) + O(1)
\end{aligned} \tag{1.1}$$

1.2.2 Run Length Encoding

RLE [4] (Run Length Encoding) is a lossless encoding algorithm that compresses sequences of the same symbol in a string, into the length of the sequence plus an occurrence of the symbol itself.

It was invented and used for compressing data for fax transmission. Indeed, a sheet of paper is viewed as a binary (i.e. monochromatic) bitmap with many white pixels and very few black pixels.

Suppose to have to compress the following string which represents a part of a monochromatic bitmap (where W stands for "white" and B for "Black").

$$WWWWWWWWWWWWBWWWWWWWWWWWWWWWBBBBBWWWWWW$$

We can take the first block of W and compress in the following way:

$$\underbrace{WWWWWWWWWWWW}_{11W}BWWWWWWWWWWWWWWWBBBBBWWWWWW$$

We can proceed in the same fashion until the end of the line is encountered. We will achieve the following compression:

$$\underbrace{WWWWWWWWWWW}_{11W}\underbrace{B}_{1B}\underbrace{WWWWWWWWWWWW}_{12W}\underbrace{BBBBB}_{5B}\underbrace{WWWWWW}_{6W}$$

We can encode the initial string with the string $\langle 11, W\rangle, \langle 1, B\rangle, \langle 12, W\rangle, \langle 5, B\rangle, \langle 6, W\rangle$. It is easy to see that the encoding is lossless and simple to reverse. In the previous example we have show a general scheme, even if, in this case, we could have encoded the string in a more convenient way: $\langle W, 11\rangle, \langle -, 1\rangle, \langle -, 12\rangle, \langle -, 5\rangle, \langle -, 6\rangle$. Indeed, if $|\Sigma| = 2$ we can simply emit the numbers (which indicate the length), while we have the alternation of the two simbols in the alphabet.

RLE can perform better or worse than the Huffman scheme: this depends on the message we want to encode.

Example 1.3

Here is an example on how RLE can perform better than the Huffman scheme. Suppose we want to encode the following string s:

$$WWWWWWWWWBBBBBBBBBCCCCCCCC$$

with $\Sigma = \{B, W, C\}$. RLE encodes s in this way:

$$\langle W, 8\rangle, \langle B, 8\rangle\langle C, 7\rangle$$

we associate 0 to B, 10 to W and 11 to C. We use gamma elias coding to encode numbers and we code 7 as 00111 and 8 as 0001000:

$$100001000000010001100111$$

We now encode s with the Huffman coding (we shall omit the calculus). The result of encoding string s with the Huffman scheme is:

$$00000000010101010101010111111111111111$$

where we associated 0 to W, 01 to B and 11 to C. The result prove that $|C_{Huffman}| = 38 > |C_{RLE}| = 24$

Example 1.4

Here is now an example on how RLE can perform worse than the Huffman scheme. Suppose we want to encode the following string s:

$$ABCAABCABCAB$$

where $\Sigma = \{A, B, C\}$. According to RLE scheme, we encode s in the following way:

$$\langle A, 1\rangle, \langle B, 1\rangle, \langle C, 1\rangle, \langle A, 2\rangle, \langle B, 1\rangle, \langle C, 1\rangle, \langle A, 1\rangle, \langle B, 1\rangle, \langle C, 1\rangle, \langle A, 1\rangle, \langle B, 1\rangle$$

We associate to A the encoding 0, 10 to B and 11 to C. Even if we encode the numbers 1 and 2 with bits 0 and 1, respectively, we cannot make a shorter encoding than:

$$00100110011001100010011000100$$

Here, the length of the encoding of s if 29.

The encoding with Huffman scheme assigns 0, 10, 11, according to their probability, respectively to A, B and C.

$$010110101101011010$$

We can therefore conclude that $|C_{Huffman}(s)| = 18 < |C_{RLE}(s)| = 29$

THEOREM 1.3 *For any binary string $s = a_1^{l_1} a_2^{l_2} \dots a_n^{l_n}$, with $a_i \in \{0,1\}$ and $a_i \neq a_{i+1}$ we have:*

$$|C_{RLE}(s)| = 1 + \sum_{i=0}^{k} |C_{PF}(l_i)|$$

where $|C_{RLE}(s)|$ is the length of encoding the string s with RLE and $|C_{PF}(l_i)|$ is the length of encoding l_i with PreFixed codes.

Proof Proving the equality is very easy, as it simply derives from the definition. Let us then analyze all the addenda in the right part of our equation.

- We have put +1 because it represents the first bit (remember that s is a binary string). We don't need to take care of the other bits in the succession because we know that there's an alternation of them in the string.
- The other addendum sums the lengths of the exponents, coded with PreFixed scheme.

As we can deduce from the above example, RLE [5] works well when the sequences of the same symbol are very long.

1.3 Implementation

Having seen how the BWT algorithm [4] works, let us now focus on how is it used. We assume from now on that the reader has well understood the previous chapters.

Take a string s and a substring w. Suppose that our substring appears n times within the main string s. As we have seen, in the matrix created by BWT there will be n consecutive rows prefixed with the substring w. Let us call these rows as $r_w + 1, r_w + 2, \dots, r_w + n$ and let's call $\hat{s} = bwt(s)$.

All of these n rows will contain all the symbols that precede w in s.

Example 1.5

Recall the example in figure 1.1. The substring $w \equiv bra$ appears 2 times within the main string *abra*cada*bra*.

As we could expect [11], in the BWT matrix we will find two consecutive rows prefixed with the substring *bra*. In the example in figure 1.1, these rows are located in the second and third positions respectively.

If it is possible to identify some patterns in the string s, that are more frequent than others, then for all patterns w_i, there will be several rows that differ from each other by only a few characters (as we may see in the example in figure 1.4).

a	bracadacabrapica	$		a	bracadacabrapica	$
b	racadacabrapica $	a		a	brapica $ abracada	c
r	acadacabrapica $ a	b		a	cabrapica $ abraca	d
a	cadacabrapica $ ab	r		a	cadacabrapica $ ab	r
c	adacabrapica $ abr	a		a	dacabrapica $ abra	c
a	dacabrapica $ abra	c		a	pica $ abracadacab	r
d	acabrapica $ abrac	a		a	$ abracadacabrapi	c
a	cabrapica $ abraca	d		b	racadacabrapica $	a
c	abrapica $ abracad	a		b	rapica $ abracadac	a
a	brapica $ abracada	c		c	a $ abracadacabrap	i
b	rapica $ abracadac	a		c	abrapica $ abracad	a
r	apica $ abracadaca	b		c	adacabrapica $ abr	a
a	pica $ abracadacab	r		d	acabrapica $ abrac	a
p	ica $ abracadacabr	a		i	ca $ abracadacabra	p
i	ca $ abracadacabra	p		p	ica $ abracadacabr	a
c	a $ abracadacabrap	i		r	acadacabrapica $ a	b
a	$ abracadacabrapi	c		r	apica $ abracadaca	b
$	abracadacabrapic	a		$	abracadacabrapic	a

FIGURE 1.4: Example of BWT applied to *abracadacabrapica*.

For this reason, the $\hat{s}$ string is `locally homogeneous`. As we can see from the examples in figure 1.1, 1.4, there are some substrings of equal characters in the string $\hat{s}$.

The base idea is to exploit this property [5]. For this purpose, we could use `Move To Front` (MTF) encoding applied to $\hat{s}$. If the $\hat{s}$ string is `locally homogeneous`, then $mtf(\hat{s})$ is encoded with small numbers.

Example 1.6

Consider the example in figure 1.4. Let's encode the $\hat{s}$ string.
In this case, the $\hat{s} \equiv cdrcrcaaiaaapabba$. (we omit the computation).
If we apply the mtf to $\hat{s}$ we find the following string: 2362113510061602.

Since we obtain a good distribution with the mtf applied to $\hat{s}$, we could, finally, apply Huffman [8] or Arithmetic coding [9] to the result. In this case we will obtain a good compression.

1.3.1 Construction with suffix arrays

Given how the Burrows-Wheeler forward transform employs a rotation matrix $\mathcal{M}$, as well as its sorted matrix $\mathcal{M}'$, to transform an entire input block, the reader might be reasonably doubtful as to how spatially efficient this algorithm can be. Indeed, being n the length of the input block and assuming that each character is stored as one single byte, the construction of $\mathcal{M}$ would require $(n+1)^2$ bytes, since the sentinel character `$` is also part of the permutations. That is why most compressors that actually implement the BWT exploit some "tricks" in order to avoid constructing the actual matrix. One such "trick" involves the usage of the so-called Suffix Arrays.

suffix	index	sorted suffix	value
abracadabra$	0	$	11
bracadabra$	1	a$	10
racadabra$	2	abra$	7
acadabra$	3	abracadabra$	0
cadabra$	4	acadabra$	3
adabra$	5	adabra$	5
dabra$	6	bra$	8
abra$	7	bracadabra$	1
bra$	8	cadabra$	4
ra$	9	dabra$	6
a$	10	ra$	9
$	11	racadabra$	2

FIGURE 1.5: Construction of the suffix array for string `abracadabra$`.

The **suffix array** of a string s is an array of integers, where each integer gives the starting position of a suffix of s when all the suffixes are sorted lexicographically. It is, in fact, a permutation of the indices for all the characters in the string.

For example, the suffix array for string `abracadabra$` is $\{11, 10, 7, 0, 3, 5, 1, 4, 6, 9, 2\}$, since the first suffix in lexicographical order is `$` and it starts at index 11 in the string; the second suffix is `a$` and it starts at index 10; the third is `abra$` at index 7, and so on. Figure 1.5 shows how this array is constructed. For convenience, all the suffixes of the string are sorted by length, longest-first, on the first column, and the indices are sorted on the second column. The third column shows the same suffixes as in the first column, but sorted lexicographically. In order to obtain the suffix array value (fourth column) for a certain suffix, one must find the row that contains that same suffix in the first column: the index in the second column indicates what value should be filled.

Sorting suffixes is equivalent to sorting rotations in $\mathcal{M}$ to obtain $\mathcal{M}'$, if we ignore the symbols following the sentinel character `$` [1]. Therefore, if s is the original input string, T denotes the array of characters in s`$`, and $\mathcal{A}_T$ denotes the suffix array of T, then we can compute column L of $\mathcal{M}'$ as follows:

$$L[i] = \begin{cases} T[\mathcal{A}_T[i] - 1] & \text{if } \mathcal{A}_T[i] \neq 0 \\ \$ & \text{otherwise} \end{cases}$$

Informally, this means that every position of L is filled with the character of T immediately preceding the corresponding suffix. If, however, that suffix is the whole string (thus the suffix array has value 0 for that index), `$` will be used instead.

Figure 1.6 shows how to obtain $bw(s)$ given s and its suffix array $\mathcal{A}_T$. The first four columns are a repetition, for convenience, of the table from Figure 1.5. The fifth column shows how the corresponding rotation matrix $\mathcal{M}'$ would look like. This is not actually needed, but is displayed here to show that every string in this column is prefixed by the corresponding suffix in the third column. Finally, the last column shows the transformed input, i.e. the last column of $\mathcal{M}'$, as obtained by applying the above formula.

The construction of suffix arrays is itself a computationally challenging effort, albeit not as challenging as by employing the actual rotation matrix $\mathcal{M}'$. Many algorithms with linear spatial complexity and pseudolinear temporal complexity have been proposed over the years. One such algorithm, presented in [10], uses $O(n)$ bytes and, although bounded by $O(n \log n)$ in the worst case, runs in $\Theta(n)$ time in the majority of cases. For an extensive classification of suffix array construction algorithms, see [13].

1.4 Theoretical results and compression boosting

suffix	index	sorted suffix	value	$\mathcal{M}'$	**L**
abracadabra$	0	$	11	$abracadabra	a
bracadabra$	1	a$	10	a$abracadabr	r
racadabra$	2	abra$	7	abra$abracad	d
acadabra$	3	abracadabra$	0	abracadabra$	$
cadabra$	4	acadabra$	3	acadabra$abr	r
adabra$	5	adabra$	5	adabra$abrac	c
dabra$	6	bra$	8	bra$abracada	a
abra$	7	bracadabra$	1	bracadabra$a	a
bra$	8	cadabra$	4	cadabra$abra	a
ra$	9	dabra$	6	dabra$abraca	a
a$	10	ra$	9	ra$abracadab	b
$	11	racadabra$	2	racadabra$ab	b

FIGURE 1.6: Application of the BWT from the suffix array for string `abracadabra$`.

1.4.1 Entropy

Let us first define the concept of entropy. The **information entropy** is the measure of uncertainty associated with a random variable. In other (and simpler) words, entropy represents a measure of information [6].

DEFINITION 1.2 [Entropy] The entropy H of a random discrete variable X over values $x_1, \cdot, x_n$ is defined as:

$$H(X) \equiv E(I(X))$$

where E is the *expected value* and I is the *self-information* of X, also known as $-log(p(x))$. If we denote p as the *probability mass function* of X, then we can rewrite the definition as follows:

$$H(X) \equiv -\sum_{i=1}^{n} p(x) \times \log p(x)$$

Burrows-Wheeler performance

An encoding algorithm is considered better than another one if it takes a lower number of bits.

For this reason, is it importat to study the theoretical performances of the **BW** function. The Burrows-Wheeler Transform (seen in 1.1) made no hypothesis on the source text, but, potentially, it uses all the input symbols previously encountered.

Given a string s, we need to investigate the k-order empirical entropy of s:

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

where w_s represents the set of all symbols that follow w in s and $|s|$ represents the length of the string s.

Now, consider to compress all the w_s blocks, given in output by BW transform, with a 0-order statistical compressor. At this moment we will not consider to use MTF algorithm. We shall use ρ_{bw} to denote the number of bits per symbol required for this kind of compression.

We know that the number of bits per symbol required for a 0-order statistical compressor is bounded by

$$H_0 \leq \rho \leq H_0 + \mu \tag{1.2}$$

Since we are compressing separately all the w_s blocks we are sure that the length of the whole compression is the sum of the length of the single blocks compressed. We can infer the following inequality:

$$\rho_{bw} \leq \frac{1}{|s|} \sum_{w_s \in bw(s)} |w_s|(H_0(w_s) + \mu) \tag{1.3}$$

where $|s|$ is the length of the input (indeed we would like to calculate the number of bits per symbol) and w_s are the blocks, in output, created by BW transform.

Let's denote with $f_{w_s}(c)$ the frequency of the symbol c in the block w_s and with $A(w_s)$ the alphabet used in the string w_s. We can rewrite the formula 1.3 in the following way:

$$\rho_{bw} \leq \frac{1}{|s|} \sum_{w_s \in bw(s)} |w_s|(\sum_{c \in A(w_s)} f_{w_s}(c) \log_2 \frac{1}{f_{w_s}(c)} + \mu) \tag{1.4}$$

Indeed $H_0(w_s)$ is the entropy of the block w_s. So, we have rewritten $H_0(w_s)$, expanding with its definition. We can do better. Since $f_{w_s}(c)$ is the frequency of the simbol c within the block w_s, it can be rewritten as the conditioned frequency $f(c|w)$. Indeed when the symbol c is contained in the block w_s, it means that c follows the context w in the input.

THEOREM 1.4 *We will prove the following disambiguation:*

$$\rho_{bw} \leq H_k(s) + \mu$$

Proof

Starting from equation 1.4, we replace $f_{w_s}(c)$ with the conditioned frequency and carry out the μ element from the sum. We have:

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

and by definition of H_0 we have:

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

and, generalizing:

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

$|w_s|(H_0(w_s))$ is the number of bits necessary for our compressor of order 0 to represent w_s. We sum for all the possibly w_s strings and we have the thesis.

1.5 Some experimental tests

We will now show some results of testing an implementation of a BWT-based compressor against a few other well-known compression algorithms. All implementations were tested on a GNU/Linux platform.

- Bzip (combination of BWT with multiple encoding schemes including MTF, RLE and Huffman): for this algorithm we shall use the "bzip2" package from http://www.bzip.org, version 1.0.5-r1.
- LZMA (LempelZivMarkov chain algorithm): we shall use the "lzip" package from http://www.nongnu.org, version 1.10.
- LZ77 (original Lempel Ziv algorithm): since there are no direct implementations of this algorithm, we will use the "lzop" package (LZ-oberhumer zip), version 1.02_rc1-r1. It is the last implementation of LZO1A compression algorithm. It compresses a block of data into matches (using a LZ77 sliding dictionary) and runs of non-matching literals. LZO1A takes care about long matches and long literal runs so that it produces good results on high redundant data and deals acceptably with non-compressible data. Some ideas are borrowed from LZRW and LZV compression algorithms [12]. We chose this kind of implementation beacuse it is a commercial and military (it has been used by NASA [2]) program.
- LZ77: for this algorithm we shall also test the DEFLATE implementation, which combines LZ77 with Huffman coding. We shall use the "zip" package from http://www.info-zip.org/, version 3.0. It is a lossless data compression algorithm.

All the tests are run in RAM (using ramfs) and made under the following configuration:

- AMD Athlon(tm)X2 DualCore QL-64
- 2048 PC2-6400 SoDimm ram
- GNU/Linux Gentoo distribution, using zen-sources-2.6.33_p1 with bfs scheduling.

We shall test all the implementations on three different kinds of files:

- all the canticas of Divine Comedy: we're going to test this file because it represents a text with some degree of correlation.
- a raw monochromatic non-compressed image: the reason why we choose this kind of file is because it contains a lot of "similar" information (black and white pixels). We therefore expected a high compression rate.
- gcc-4.4.3: we will test the algorithms on this package in order to assess how they behave on compressing a large amount of files, which are distributed across a sparse directory structure with great ramification.

All the tests are run 100 times and we will show the avarage result.

From figure 1.7,1.8,1.9 we can easly deduce some conclusions. First of all, we can conclude that *lzma* is very bad. It has ever the worst compress-time and does not reach the best compression rate. *bzip2* seems to be quite in the middle. Sometimes it takes a lot of time to encode, but it reaches the best compression rate. By considering the decompression time, *bzip2* seems to slow down too much as the size of the file grows.

Perhaps the best solution seems to be the *zip*: it takes a short time to encode/decode and reaches a very good compression rate.

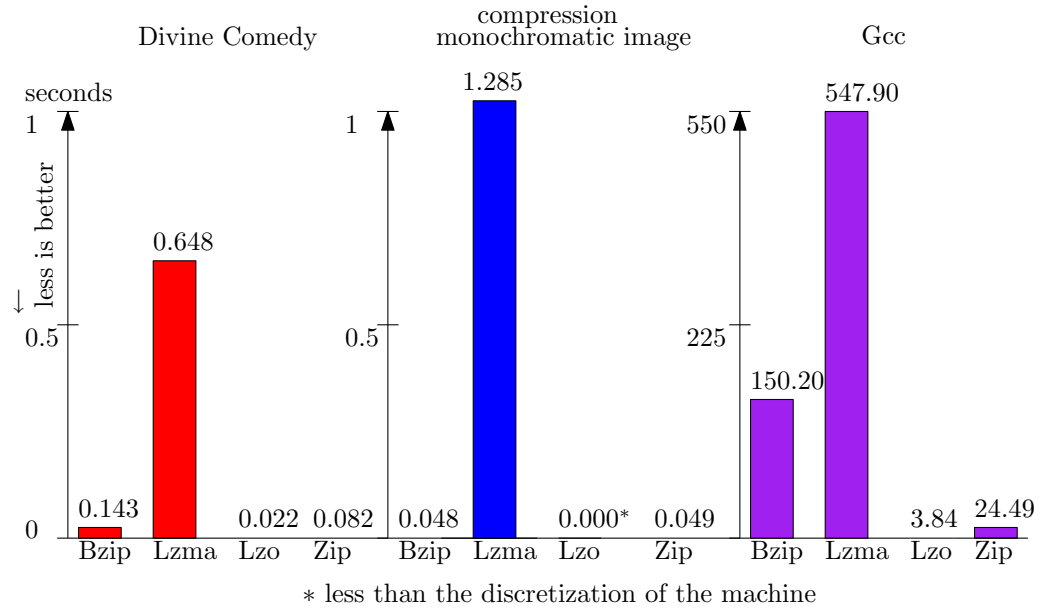

FIGURE 1.7: Encoding test (lower is better)

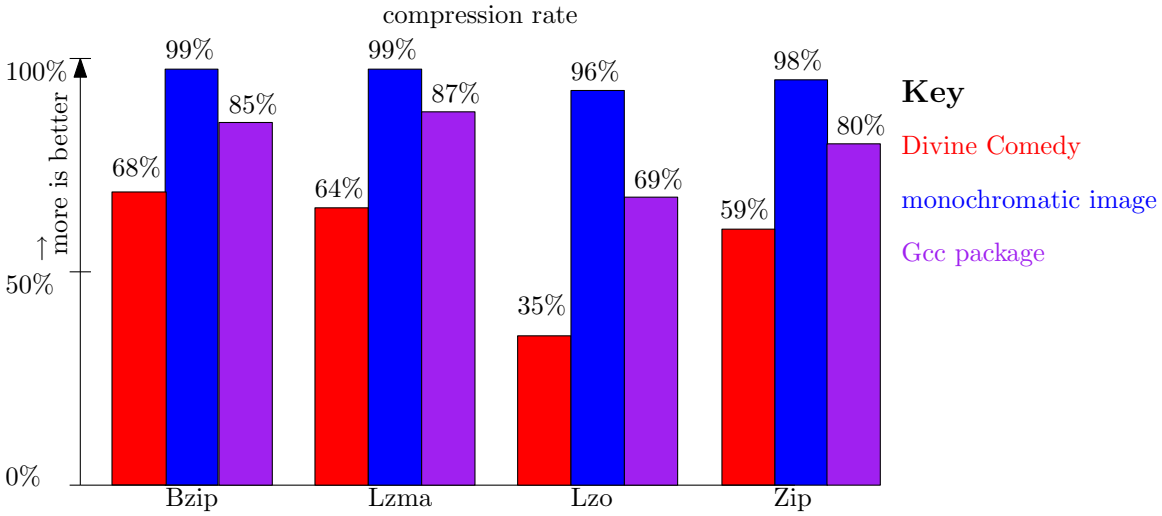

FIGURE 1.8: Ratio of compression (higher is better)

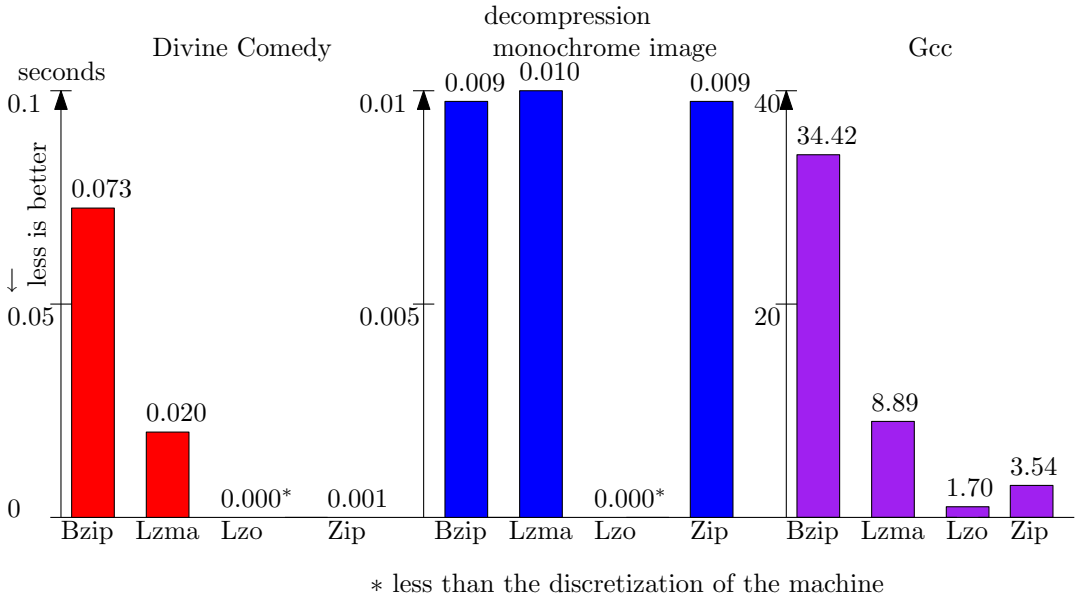

FIGURE 1.9: Decoding test (lower is better)

lzo is the fastest algorithm we tested. Unfortunately, its compression rate seems to be very bad.

It's not so easy to understand so well what kind of algorithm is the best. Every possible consideration depends on what the user needs (compression rate, encoding time, decoding time).

References

[1] D. Adjeroh, T. Bell, and A. Mukherjee. *The Burrows-Wheeler Transform: Data Compression, Suffix Arrays, and Pattern Matching.* Springer series in statistics. Springer, 2008.

[2] National Aeronautics and Space Administration. http://marsrovers.jpl.nasa.gov/home/.

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

[4] Michael Burrows and David J. Wheeler. A block-sorting lossless data compression algorithm. Technical Report 124, Digital Systems Research Center (SRC), 1994.

[5] Stefano Cataudella and Antonio Gulli. BW transform and its applications. Tutorial on BW-transform, 2003.

[6] Paolo Ferragina and Giovanni Manzini. Boosting textual compression. In Ming-Yang Kao, editor, *Encyclopedia of Algorithms*, pages 1–99. Springer US, 2008.

[7] Paolo Ferragina and Giovanni Manzini. Burrows-wheeler transform. In Ming-Yang Kao, editor, *Encyclopedia of Algorithms*, pages 1–99. Springer US, 2008.

[8] D.A. Huffman. A method for the construction of minimum redundancy codes. *Proceedings IRE 40*, 10:1098–1101, 1952.

[9] Glen G. Langdon. An introduction to arithmetic coding. *IBM J. Res. Dev.*, 28(2):135–149, 1984.

[10] Giovanni Manzini and Paolo Ferragina. Engineering a lightweight suffix array construction algorithm. *Algorithmica*, 40(1):33–50, 2004.

[11] Shoshana Neuburger. The burrows-wheeler transform: data compression, suffix arrays, and pattern matching by donald adjeroh, timothy bell and amar mukherjee springer, 2008. *SIGACT News*, 41(1):21–24, 2010.

[12] Markus Franz Xaver Johannes Oberhumer. ftp://ftp.matsusaka-u.ac.jp/pub/compression/oberhumer/lzo1a-announce.txt, 1996.

[13] Simon J. Puglisi, William F. Smyth, and Andrew Turpin. A taxonomy of suffix array construction algorithms. In Jan Holub and Milan Simánek, editors, *Stringology*, pages 1–30. Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University, 2005.