1 Dictionary-based compressors

The methods based on a *dictionary* take a totally different approach to compression than the statistical ones: here we are not interested in deriving the characteristics (i.e. probabilities) of the source which generated the input sequence S. Rather, we assume to have a *dictionary of strings*, and we look for those strings in S, replacing them with a *token* which identifies them in the dictionary. The choice of the dictionary is of course crucial in determining how well the file is compressed. An English dictionary will have a hard time to compress an Italian text, for instance; and it would be totally unappropriate to compress an executable file. Thus, while a *static* dictionary can be used to compress very well certain specific and known in advance kinds of files, it cannot be used for a good *general-purpose compressor*. Moreover, we don't want to transmit the full dictionary along with each compressed file – and it's often unreasonable to assume the receiver already has a copy of our dictionary.

So, starting from 1977, Ziv and Lempel introduced a family of compressors which addressed successfully these problems by designing two algorithms, named LZ77 and LZ78 from the initials of the inventors and the years of the proposal, which use the input sequence they are compressing as the dictionary, and substitute each occurrence of an already seen string with either the offset of its previous position or an ID assigned incrementally to new dictionary phrases. The dictionary is *dynamically built* in the sense that it starts empty and then it grows as the input sequence is processed; at the beginning low compression is achieved, but after some kbs good compression is obtained. For typical textual files, those methods achieve about 33% compression ratio. The Lempel-Ziv's compressors are very popular because of their `gzip` instantiation, and constitute the base of more sophisticated compressors in use today, such as `7zip` and `LZMA` and `LZ4`. In the following paragraphs, we will show them in detail, along with some interesting variants.

1.1 LZ77

Ziv and Lempel, in their seminal paper of 1977 [11], described their contribution as follows *"[. . .] universal coding scheme which can be applied to any discrete source and whose performance is comparable to certain optimal fixed code book scheme designed for completely specified sources [. . .]"*. They key expression is "comparable to [...] fixed code book scheme designed for completely specified sources", because the authors compare to previously designed statistical compressors, such as Huffman and Arithmetic, for which a statistical characterization of the source was necessary.

Conversely, dictionary-based compressors waive this characterization which is derived *implicitly* by observing *substring repetitiveness* via a fully syntactic approach.

We will not dig into the observations which provide a mathematical ground to these comments [11, 3], rather we will concentrate only on the algorithmic issues. LZ77's compressor is based on a *sliding window* $W[1, w]$ which contains a portion of the input sequence that has been processed so far, typically consisting of the last w characters, and a *look-ahead buffer* B which contains the suffix of the text still to be processed. In the following picture the window $W = aabbababb$ is of size 9, and the rest of the input sequence is $B = baababaabbaa\$$.

$$\longleftarrow \cdots \boxed{\text{aabbababb}}\; baababaabbaa\$ \longrightarrow$$

The algorithm works inductively by assuming that everything occurring before B has been processed and compressed by LZ77; where W is initially set to the empty string. The compressor operates in two main stages: *parsing* and *encoding*. Parsing consists of transforming the input sequence S into a sequence of triples of integers (called *phrases*). Encoding turns these triples into a (compressed) bit stream by applying either a statistical compressor (i.e. Huffman or Arithmetic) to each triplet-component individually, or an integer encoding scheme.

So the interesting algorithmic stage is the parsing stage, which works as follows. LZ77 searches for the longest prefix α of B which occurs as a substring of WB. We write the concatenation WB rather than the single string B because the previous occurrence we are searching for may start in W and extend up to within B. Say α occurs at distance d from the current position (namely the beginning of B), and it is followed by character c in B, then the triple generated by LZ77 is $\langle d, |\alpha|, c\rangle$ where $|\alpha|$ is the length of the *copied* string. If a match is not found the output triple becomes $\langle 0, 0, B[1]\rangle$. We notice that any occurrence of α in W must be followed by a character different of c, otherwise α would be *not* the longest prefix of B which repeats in W.

After that this triple is emitted, LZ77 advances in B by $|\alpha| + 1$ positions, and slides W correspondingly. We talk about LZ77 as a dictionary-based compressor because "the dictionary" is not explicitly stored, rather it is implicitly formed by all substrings of S which start in W and extend rightward, possibly ending in B. Each of those substrings is represented by the triple indicated above. The dictionary is *dynamic* because at every shift it has to be updated by removing the substrings starting in $W[1, |\alpha| + 1]$, and adding the substrings starting in $B[1, |\alpha| + 1]$.

The role of the sliding window is easy to explain, it delimits the size of the dictionary which is quadratic in W's length, so it impacts significantly onto the time cost for the search of α. As a running example, let us consider the following sequence of LZ77-parsing steps:

$$\begin{array}{rcl}
\boxed{\,|\text{aabbabab}} & \Longrightarrow & \langle 0, 0, a\rangle \\
\boxed{\text{a}|\text{abbabab}} & \Longrightarrow & \langle 1, 1, b\rangle \\
\boxed{\text{aab}|\text{babab}} & \Longrightarrow & \langle 1, 1, a\rangle \\
\boxed{\text{aabba}|\text{bab}} & \Longrightarrow & \langle 2, 3, EOF\rangle \\
\cdots\cdots & & \cdots\cdots
\end{array}$$

It is interesting to note that the last phrase $\langle 2, 3, EOF\rangle$ presents a copy-length which is larger than the copy-distance; this actually indicates the special situation mentioned above in which α starts in W and ends in B. Even if this *overlapping* occurs, the copy-step that must be executed by LZ77 in decompression is not affected, provided that it is executed sequentially according to the following piece of code:

```
for i = 0 to L-1 do { S[s+i] = S[s-d+i]; }
s = s+L;
```

where the triple to be decoded in $\langle d, L, c\rangle$ and $S[1, s-1]$ is the prefix of the input sequence which has been already decompressed. Since $d \le |W|$ and the window size is up to few Megabytes, the copy operation does not elicit any cache miss, thus making the decompression process very fast indeed. The longer is the window W, the longer are possibly the phrases, the fewer is their number and thus possibly the shorter is the compressed output; but of course, in terms of compression time, the longer is the time to search for the longest copied α. Vice versa, the shorter is W, the worse is the compression ratio but the faster is the compression time. This trade-off is evident and its magnitude depends on the input sequence.

To slightly improve compression we make the following observation which is due to Storer and Szymanski [8] and dates back to 1982. In the parsing process two situations may occur: a longest match has been found, or it has not. In the former case it is not reasonable to add the character following α (third component in the triple), given that we anyway advance in the input sequence. In the latter case it is not reasonable to emit two 0s (first two components in the triple) and thus waste one integer encoding. The simplest solution to these two inefficiencies is to always output a pair, rather than a triple, with the form: $\langle d, |\alpha|\rangle$ or $\langle 0, B[1]\rangle$. This variant of LZ77 is named LZss, and it is often confused with LZ77, so we will use it from this point on.

By referring to the previous running example, LZss would obtain the parsing:

$$\begin{array}{|r|l|cl}
\hline
 & \text{aabbabab} & \Longrightarrow & \langle 0, a\rangle \\
\hline
\text{a} & \text{abbabab} & \Longrightarrow & \langle 1, 1\rangle \\
\hline
\text{aa} & \text{bbabab} & \Longrightarrow & \langle 0, b\rangle \\
\hline
\text{aab} & \text{babab} & \Longrightarrow & \langle 1, 1\rangle \\
\hline
\text{aabb} & \text{abab} & \Longrightarrow & \langle 3, 2\rangle \\
\hline
\text{aabbab} & \text{ab} & \Longrightarrow & \langle 2, 2\rangle \\
\hline
\end{array}$$

Gzip: a smart and fast implementation of LZ77. The key programming problem when implementing LZ77 is the *fast search* for the longest prefix α of B which repeats in W. A brute-force algorithm that checks the occurrence of every prefix of B in W, via a linear backward scan, would be very time-consuming and thus unacceptable for compressing Gbs files.

Fortunately, this process can be accelerated by using a suitable data structure. `Gzip`, the most popular implementation of LZ77, uses a hash table to determine α and find its previous occurrence in W. The idea is to store in the hash table all 3-grams occurring in W, namely all triplets of contiguous characters, by using as `key` the 3-gram and as its `satellite` data the position in B where that 3-gram occurs. Since a 3-gram may repeat multiple times in W, the hash table saves for a given 3-gram all of its multiple occurrences, sorted by increasing position in S. This way, when W shifts to the right because of the emission of the pair $\langle d, \ell\rangle$, the hash table can be updated by deleting the ℓ 3-grams starting at $W[1, \ell]$, and inserting the ℓ 3-grams starting at $B[1, \ell]$.

The search for α is implemented as follows:

- first, the 3-gram $B[1, 3]$ is searched in the hash table. If it does not occur, then `Gzip` emits the phrase $\langle 0, B[1]\rangle$, and the parsing advances of one single character. Otherwise, it determines the list $\mathcal{L}$ of occurrences of $B[1, 3]$ in W.
- second, for each position i in $\mathcal{L}$ (which is expressed as absolute position in S), the algorithm compares character-by-character $S[i, n]$ against B in order to compute their longest common prefix. At the end, the position $i^* \in \mathcal{L}$ sharing this longest common prefix is determined, as well as it is found α.

- finally, let p be the current position of B in S, the algorithm emits the pair $\langle p - i^*, |\alpha| \rangle$, and advances the parsing of $|\alpha|$ positions.

`Gzip` implements the encoding of the phrases by using Huffman over two alphabets: the one formed by the lengths of the copies plus the literals, and the one formed by the distances of the copies. This trick is sufficiently smart to save one extra bit to distinguish between the two types of pairs. In fact, $\langle 0, c \rangle$ is represented as the Huffman encoding of c, and $\langle d, \ell \rangle$ is represented reversed by anticipating the Huffman encoding of ℓ. Given that literals and copy-lengths are encoded within the same alphabet, the decoder fetches the next codeword and decompresses it, so being able to distinguishing whether the next item is a character c or a length ℓ. According to the result, it can either restart the decoding of the next pair (c has been decoded), or it can decode d (ℓ has been decoded) by using the other Huffman code.

`Gzip` deploys an additional programming trick that further speed ups the compression process. It consists of sorting the list of occurrences of the 3-grams from recent to oldest matches, and possibly stop the search for α when a sufficient number of candidates has been checked. This trades the length of the longest match against the speed of the search. As far as the size of the window W is concerned, `Gzip` allows to specify `-1,...,-9` which actually means that the size may vary from 100Kb to 900Kb, with a consequent improvement of the compression ratio, at the cost of slowing down the compression speed. Not surprisingly, the longer is W, the faster is the decompression because the smaller is the number of encoded phrases, and thus the smaller is the number of cache misses induced by the Huffman decoding process.

For other implementations of LZ77, the reader can look at Chapter **??** where we discussed the use of the Suffix Tree and unbounded window; as well as we refer to [4] for details about implementations which take into account the size of the compressed output (in bits) which clearly depends on the number of phrases but also from the values of their integer components, in a way that cannot be underestimated. Briefly, it is not necessarily the case that a longer α induces a shorter compressed output, because it might need to copy α from a far distance d, thus taking many bits for its encoding; rather, it might be better to divide α into two substrings which can be copied closer enough that the total number of bits required for their encoding is less than the ones needed for d.

1.2 LZ78

The sliding window used by LZ77 from the one hand speeds up the search for the longest phrase to encode, but from the other hand limits the search space, and thus the ultimate compression ratio. In order to avoid this problem and still keep a fast compression stage, Ziv and Lempel devised in 1978 another algorithm, which has been consequently called LZ78 [12]. The key idea is to build *incrementally* an *explicit dictionary* that contains only a subset of the substrings of the input sequence S, selected according to a simple rule that is detailed below. Concurrently, S is decomposed into phrases which are taken from the current dictionary.

Phrase detection and dictionary update are deeply intermingled. Let S' be the sequence to be parsed yet, and let $\mathcal{D}$ be the current dictionary in which every phrase f is identified via the integer $\mathtt{id}(f)$. The parsing of S' consists of determining its longest prefix f' which is also a phrase of $\mathcal{D}$, and substituting it with the pair $\langle \mathtt{id}(f'), c \rangle$ where c is the character following f' in S'. Next, $\mathcal{D}$ is updated by adding the new phrase $f'c$, which is just one character longer than the phrase $f' \in \mathcal{D}$. Therefore the dictionary is *prefix-complete* because it will contain all the prefixes of every phrase in $\mathcal{D}$, moreover its size grows with the length of the input sequence.

It goes without saying that, as it occurred for LZ77, the stream of pairs generated by the LZ78-parsing will be encoded via a statistical compressor (such as Huffman or Arithmetic) or via any variable-length integer encoder. This will produce the compressed bit stream, eventual output of LZ78.

2

The Burrows-Wheeler Transform

This chapter describes a lossless data compression technique devised by Michael Burrows and David Wheeler in 1994 at the DEC Systems Research Center. This technique was published in a Technical Report of the company [4, 8],[1] and since it was rejected from the Data Compression Conference (as Mike Burrows stated in its foreword to [9][2]), the two authors decided of not publishing their paper anywhere. Fortunately, Mark Nelson drew attention to it in a Dr. Dobbs article, and that was enough to ensure its survival.

A wonderful thing about publishing an idea is that a greater number of minds can be brought to bear on the surrounding problems. This is what happened around the Burrows-Wheeler Transform, whose studies exploded around the year 2000, leading me, Giovanni Manzini and S. Muthukrishnan to celebrate a ten-years-later resume in a special issue of *Theoretical Computer Science* [9]. In that volume, Mike Burrows again declined to publish the original TR but wrote a wonderful Foreword dedicated to the memory of David Wheeler, who passed away in 2004, and finally stated: *"This issue of Theoretical Computer Science is an example of how an idea can be improved and generalized when more people are involved. I feel sure that David Wheeler would be pleased to see that his technique has inspired so much interesting work."*

The so called *Burrows-Wheeler Transform* (or BWT) offered a revolutionary alternative to dictionary-based compressors and actually initiated a new family of compressors (such as `bzip2` [18] or the `booster` [6]) as well as a new powerful family of compressed indexes (such as `FM-index` [7], and many variations [15]). In the following we will detail the `BWT` and the other two simple compressors, i.e. Move-To-Front and Run-Length Encoding, whose combination constitutes the `bzip`-based compressors. We will also briefly mention few theoretical issues about the `BWT` performance expressed in terms of the k-th order empirical entropy of the data to be compressed.

[1]M. Burrows: "In the technical report that described the BWT, I gave the year as 1981, but later, with access to the memory of his wife Joyce, we deduced that it must have been 1978."

[2]Years passed, and it became clear that David had no thought of publishing the algorithm—he was too busy thinking of new things. Eventually, I decided to force his hand: I could not make him write a paper, but I could write a paper with him, given the right excuse.

© Paolo Ferragina, 2009-2014

2.1 The Burrows-Wheeler Transform

The *Burrows-Wheeler Transform* (BWT) is not a compression algorithm *per se*, as it does not squeeze the input size, it is a *permutation* (and thus, a lossless transformation) of the input symbols which are laid down in a way that the resulting string is most suitable to be compressed via simple algorithms, such as *Move-To-Front coding* (shortly MTF) and *Run Length Encoding* (shortly RLE), both to be described in Section 2.2. This permutation forces some "locally homogeneous" properties in the ordering of the symbols that can be fully deployed, efficiently and efficaciously, by the combination `MTF + RLE`. A last statistical encoding step (e.g. Huffman or Arithmetic) is finally executed in order to eventually squeeze the output bit stream. All these steps constitute the backbone of any `bzip`-like compressor which will be discussed in Section 2.3.

The BWT consists of a pair of inverse transformations: a *forward transform*, which rearranges the symbols in the input string; and a *backward transform*, which somewhat magically reconstructs the original string from its BWT. It goes without saying that the invertibility of BWT is necessary to guarantee the decompression of the input file!

2.1.1 The forward transform

Let $s = s_1 s_2 ... s_n$ be an input string on n symbols drawn from an *ordered* alphabet Σ. We append to s a special symbol \$ which does not occur in Σ and it is assumed to be smaller than any other symbol in the alphabet, according to its total ordering.[3] The forward transform proceeds as follows:

1. Build the string $s\$$.
2. Consider the *conceptual* matrix $\mathcal{M}$ of size $(n+1) \times (n+1)$, whose rows contain all the cyclic left-shifts of string $s\$$. M is called the *rotation matrix* of s.[4]
3. Sort the rows of $\mathcal{M}$ reading them *left-to-right* and according to the ordering defined on alphabet $\Sigma \cup \{\$\}$. The final matrix is called $\mathcal{M}'$. Since \$ is smaller than any other symbol in Σ and, by construction, appears only once, the first row of $\mathcal{M}'$ is $\$s$.
4. Set $bw(s) = (\widehat{L}, r)$ as the output of the algorithm, where $\widehat{L}$ is the string obtained by reading the last column of M', sans symbol \$, and r is the position of \$ there.

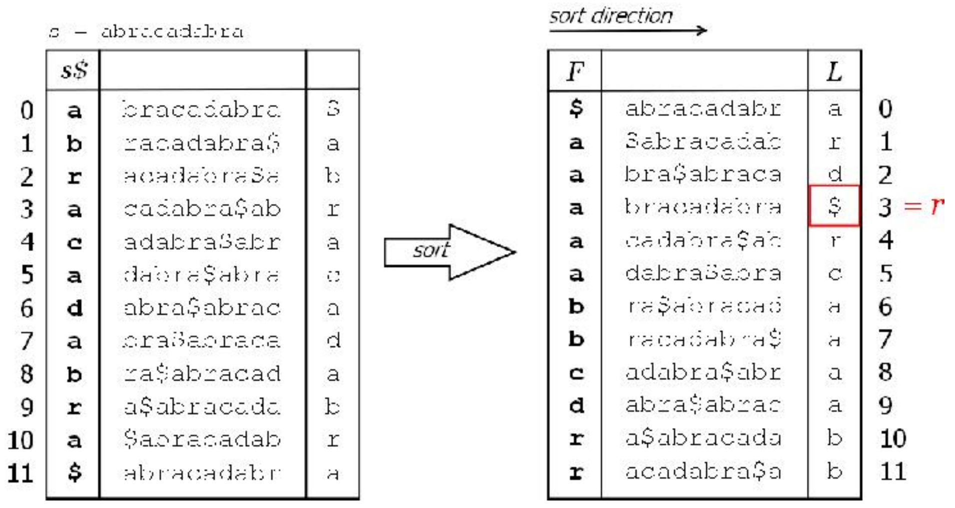
We said above that $\mathcal{M}$ is a conceptual matrix because we have to avoid its explicit construction, which otherwise would make the BWT an elegant mathematical object: the size of $\mathcal{M}$ is quadratic in $bw(s)$'s length, so the conceptual matrix has size $2^{48} \approx 1000$Tb just for transforming a string of 16Mb. In Section 2.3 we will actually show that M' can be built in time and space linear in the length of the input string s, by resorting Suffix Arrays.

An alternate enunciation of the algorithm, less frequent yet still present in the literature [19], constructs matrix $\mathcal{M}'$ by sorting the rows of $\mathcal{M}$ reading them *right-to-left* (i.e. starting from the last symbol of every row). Then, it takes the string $\widehat{F}$ formed by scanning the first column of $\mathcal{M}'$ top-to-bottom and, again, skipping symbol \$ and storing its position in r'. The output is then $bw(s) = (\widehat{F}, r')$. This enunciation is the dual of the one given above because it is possible to formally prove that both strings $\widehat{F}$ and $\widehat{L}$ exhibit the same *local-homogeneity* properties and thus compression, to be illustrated below. In the rest of the chapter we will refer to the left-to-right sorting of M's rows and to $(\widehat{L}, r)$ as the BWT of the string s, somehow forgetting the integer r.

[3]The step that concatenates the special symbol \$ 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 description.

[4]The left shift of a string $a\alpha$ is the string αa, namely the first symbol is moved to the end of the original string.

In order to better understand the power of the Burrows-Wheeler Transform, let us consider the following running example formulated over the string s = `abracadabra`. The left side of Figure 2.1 shows the rotated matrix $\mathcal{M}$ built over s; whereas the right side of Figure 2.1 shows sorted matrix M'. Because the first row of $\mathcal{M}$ is the only one to end with \$, which is the lowest-ordered symbol in the alphabet, row `$abracadabra` is the first row of $\mathcal{M}'$. The other three rows of $\mathcal{M}'$ are the ones beginning with `a`, and then follow the rows starting with `b`, `c`, `d` and finally `r`, respectively.



	s$		
0	a	bracadabra	$
1	b	racadabra$	a
2	r	acadabra$a	b
3	a	cadabra$ab	r
4	c	adabra$abr	a
5	a	dabra$abra	c
6	d	abra$abrac	a
7	a	bra$abraca	d
8	b	ra$abracad	a
9	r	a$abracada	b
10	a	$abracadab	r
11	$	abracadabr	a

F		L	
$	abracadabr	a	0
a	$abracadab	r	1
a	bra$abraca	d	2
a	bracadabra	$	3 = r
a	cadabra$ab	r	4
a	dabra$abra	c	5
b	ra$abracad	a	6
b	racadabra$	a	7
c	adabra$abr	a	8
d	abra$abrac	a	9
r	a$abracada	b	10
r	acadabra$a	b	11

FIGURE 2.1: Forward Burrows-Wheeler Transform of the string s = `abracadabra`.

If we read the first column of $\mathcal{M}'$, denoted by F, we obtain the string `aaaaabbcdr` which is the sorted sequence of all symbols in s. We finally obtain $\widehat{L}$ by excluding the single occurrence of `$` from the last column L, so $\widehat{L}$ = `ardrcaaaabb`, and set $r = 3$.

The example is illustrative of the locally-homogeneous property we were mentioning before: the last 6 symbols of the last column of M form a highly repetitive string `aaaabb` which can be easily and highly compressed via the two simple compressors `MTF + RLE` (described below). The soundness of this statement will be mathematically sustained in the following pages, here we content ourselves by observing that this repetitiveness occurs not by chance but it is induced by the way M's rows are sorted (left-to-right) and texts are written down by humans (left-to-right). The nice issue here is that many real sources (they are called Markovian) do exist that generate data sequences, other than texts, that can be turned to be locally homogeneous via the Burrows-Wheeler Transformation, and thus can be highly compressed by `bzip`-like compressors.

2.1.2 The backward transform

We observe, both by construction and from the example provided above, that each column of the sorted cyclic-shift matrix $\mathcal{M}'$ (and also M) contains a permutation of s`$`. In particular, its first column F = `aaaaabbcdr` is alphabetically sorted and thus it represents the best-compressible transformation of the original input block. But unfortunately F cannot be used as `BWT` because it is not invertible: every text of length 11 and consisting of 5 occurrences of symbol `a`, 2 occurrences of `b`, 1 occurrence `c`, `d`, `r` respectively, originates a `BWT` whose F is the same as the one above.

The Burrows-Wheeler transform represents, in some sense, the best column of M' to be chosen as transformed s in terms of reversibility and compressibility of s.

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 symbol at a given index in s.

FACT 2.1 *For $1 \leq i \leq n$, let $s[k_i, n-1]$ denote the suffix of s prefixing row i of $\mathcal{M}'$. Clearly, row i is then followed by symbol \$, and then by the prefix $s[1, k_i - 1]$ because of the leftward cyclic shift.*

For example in Figure 2.1, row 2 of $\mathcal{M}'$ is prefixed by `abra`, followed by `$abracad`.

Property 2.1 *The symbol $L[i]$ precedes the symbol $F[i]$ in the string s, except for the row i such that $L[i] = \$$, in which case $F[i] = s[1]$.*

Proof Because of Fact 2.1 the last symbol of the row i is $L[i] = s[k_i - 1]$ and its first symbol is $F[i] = s[k_i]$. So the statement follows. ∎

Intuitively, this property descends from the very nature of every row in $\mathcal{M}$ and $\mathcal{M}'$ that is a *left* cyclic-shift of $s\$$, so if we take two extremes of each row, the symbol on the right extreme (i.e. on L) is immediately followed by the one on the left extreme (i.e. on F) over the string s.

Property 2.2 *All the occurrences of a same symbol c in L maintain the same relative order as in F. This means that the kth occurrence in L of symbol c corresponds to the kth occurrence of the symbol c in F.*

Proof Given two strings t and t', we shall use the notation $t \prec t'$ to indicate that string t lexicographically precedes string t'.

Fix now the symbol c. If c occurs once in s then the proof derives immediately because the single occurrence of c in F obviously maps to the single occurrence of c in L. (Both columns are permutations of s.) To prove the more complicate situation that c occurs at least twice in s, let us fix two of these occurrences and pick their rows of the sorted matrix $\mathcal{M}'$, say $r(i)$ and $r(j)$ with $i < j$. We can observe few interesting things:

- row $r(i)$ precedes lexicographically row $r(j)$, given the ordering of $\mathcal{M}'$'s rows and the fact that $i < j$, by assumption;
- both rows $r(i)$ and $r(j)$ start with symbol c, by assumption;
- given that $r(i) = c\,\alpha$ and $r(j) = c\beta$, it is $\alpha \prec \beta$.

Since we are interested in the respective positions of those two occurrences of c when they are mapped to L, we consider the two rows $r(i')$ and $r(j')$ which are obtained by rotating those two rows leftward by one single symbol: $r(i') = \alpha\, c$ and $r(j') = \beta\, c$. This way, this rotation brings the first symbol $F[i]$ (resp. $F[j]$) into the last symbol $L[i']$ (resp. $L[j')$ of the rotated rows. Since $\alpha \prec \beta$, it is $r(i') \prec r(j')$ and so the preservation of the ordering in L holds true for that pair of occurrences of c. Given that this order-preserving property holds for every pair of occurrences of c in F/L, it holds true for all of them. ∎

We have now all mathematical tools to design an algorithm which reconstructs s from its $bw(s) = (\widehat{L}, r)$ by exploiting the so called LF-mapping, an array of n integers in the range $[0, n-1]$.

DEFINITION 2.1 It is $LF[i] = j$ iff the symbol $L[i]$ maps to symbol $F[j]$. This way, if $L[i]$ is the kth occurrence in L of symbol c, then $F[LF[i]]$ is the kth occurrence of c in F.

Algorithm 2.1 Constructing the LF-mapping from column L

```
for $i = 0, 1, \ldots, n-1$ do
    $C[L[i]]$++;
end for
$temp = 0,\ sum = 0$;
for $i = 0, 1, \ldots, |\Sigma|$ do
    $temp = C[i]$;
    $C[i] = sum$;
    $sum$+= $temp$;
end for
for $i = 0, 1, \ldots, n-1$ do
    $LF[i] = C[L[i]]$;
    $C[L[i]]$++;
end for
```

Building LF is pretty straightforward for symbols that occur only once, as it is the case of `$`, `c` and `d` in $s =$ `abracadabra$`, see Figure 2.1. But when it comes to symbols `a`, `b` and `r`, which occur several times in the string s, computing LF efficiently is no longer trivial. Nonetheless it can be solved in optimal $O(n)$ time thanks to Property 2.2, as Algorithm 2.1 details. This algorithm uses an auxiliary vector C, of size $|\Sigma| + 1$. For the sake of description, we assume that array C is indexed by a *symbol* rather than by a integer.[5]

The first `for`-cycle computes, for each symbol c, the number n_c of its occurrences in L, and stores $C[c] = n_c$. Then, the second `for`-cycle, turns these symbol-wise occurrences into a cumulative sum, so that the new $C[c]$ stores the total number of occurrences in L of symbols *smaller than* c, namely $C[c] = \sum_{x<c} n_x$. This is done by adopting two auxiliary variables, so that the overall working space is still $O(n)$. We notice that $C[c]$ gives the first position in F where symbol c occurs. Therefore, before the last `for`-cycle starts, $C[c]$ is the landing position in F of the first c in L (we thus know the LF-mapping for the first occurrence of every alphabet symbol). Finally, the last `for`-cycle scans the column L and, whenever it encounters symbol $L[i] = c$, then it sets $LF[i] = C[c]$. This is correct when c is met for the first time; then $C[c]$ is incremented so that the next occurrence of c in L will map to the next position in F (given the contiguities in F of all rows starting with that symbol). So the algorithm keeps the invariant that $LF[i] = \sum_{x<c} n_x + k$, after that k occurrences of c in L have been processed. It is easy to derive the time complexity of such computation which is $O(n)$.

Given the LF-mapping and the fundamental properties shown above, we are able to reconstruct s backwards starting from the transformed output $bw(s) = (\widehat{L}, r)$ in $O(n)$ time and space. Clearly it is easy from $bw(s)$ to construct L, just insert \$ at position r of $\widehat{L}$. The algorithm then picks the last symbol of s, namely $s[n-1]$, which can be easily identified at $L[0]$, given that the first row of M' is \$$s$. Then it proceeds by moving one symbol at a time to the left in s, deploying the two Properties above: Property 2.2 allows to map the current symbol occurring in L (initially $L[0]$) to its corresponding copy in F; then Property 2.1 allows to find the symbol which precedes that copy in F by taking the symbol at the end of the same row (i.e. the one in L). This double step, which returns on L, allows to move one symbol leftward in s. Repeating this up to the beginning of s we are able to reconstruct this string. The pseudo-code is reported in Algorithm 2.2.

As an example, refer to Figure 2.1 where we have that $L[0] = s[n-1] =$ `a`, and execute the

[5]Just implement C as a hash table, or observe that in practice any symbol is encoded via an integer (ASCII code maps to the range $0, \ldots, 255$) which can be used as its index in C.

Algorithm 2.2 Reconstructing s from $bw(s)$

```
1: Derive column $L$ from $bw(s)$;
2: Compute $LF[0, n-1]$ from $L$;
3: $k = 0$;  $i = n - 1$;
4: while $i \geq 0$ do
5:     $s[i] = L[k]$;
6:     $k = LF[k]$;
7:     $i--$;
8: end while
```

while-cycle of Algorithm 2.2. Definition 2.1 guarantees that $LF[0]$ points to the first row starting with `a`, this is the row 1. So that copy of `a` is LF-mapped to $F[1]$ (and in fact $F[1] =$ `a`), and the preceding symbol in s is thus $L[1] =$ `r`. These two basic steps are repeated until the whole string s is reconstructed. Just continuing the previous running example, we have that $L[1] =$ `r` is LF-mapped to the symbol in F at position $LF[1] = 10$ (and indeed $F[10] =$ `r`). In fact, $L[1]$ and $F[10]$ is the first occurrence of symbol `r` in both columns L and F, respectively. The algorithm then takes as preceding symbol of `r` in s the symbol $L[10] =$ `b`. And so on...

THEOREM 2.3 *The original string s can be reconstructed from its* `BWT` *in $O(n)$ time and space. Algorithm 2.2 elicits possibly one cache-miss per symbol.*

Several recent results addressed the problem of reducing the number of cache misses as well as the working space of algorithms inverting `BWT`. Some progress has been made in the literature (see e.g. [17, 12, 13, 11]), but yet reductions are limited, e.g. small constants for the cache-misses, say $2 \div 4$, which get larger if the data is highly repetitive. Much has still to be discovered here!

2.2 Two other simple transforms

Let us now focus on two simple algorithms that come in very useful to design the compressor `bzip2`. These algorithms are called *Move-To-Front* (`MTF`) and *Run-Length Encoding* (`RLE`). The former maps symbols into integers, the latter maps runs of equal symbols into pairs. For the sake of completeness we observe that `RLE` is a compressor indeed, because the output sequence may be reduced in length in the presence of long runs of equal symbols; while `MTF` can be turned into a compressor by encoding the run-lengths via proper integer encoders [3]. In general the compression performance of those algorithms is very poor: `BWT` is magically their killer application!

2.2.1 The Move-To-Front transform

The `MTF`-transformation [3] implements the idea that every symbol of a string s can be replaced with its index in a proper *dynamic* list $\mathcal{L}^{MTF}$ containing all alphabet symbols. The string produced in output, denoted hereafter as s^{MTF}, is initialized to the empty string and contains as symbols integers in the range $[0, |\Sigma| - 1]$. At each step i, we process the symbol $s[i]$ and find its position p in $\mathcal{L}^{MTF}$. Then p is added to the string s^{MTF}, and $\mathcal{L}^{MTF}$ is modified by *moving* the symbol $s[i]$ to the *front* of the list.

It is greatly advantageous to apply this processing over the column L of $bw(s)$ because, as it will be clear next, it transforms *locally homogeneous substrings* of L into a *globally homogeneous string* L^{MTF} in which abound small integers. At this point we could apply any integer compressor,

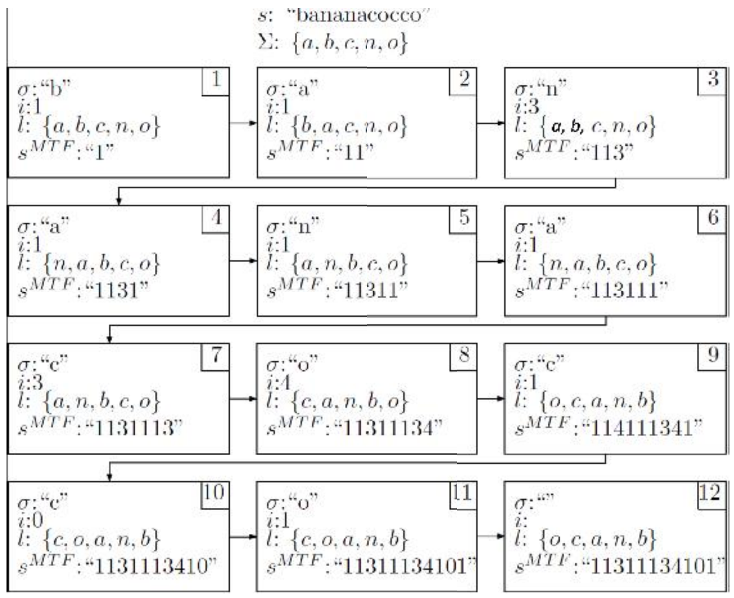

s: "bananacocco"
Σ: {a, b, c, n, o}
1 σ:"b" i:1 l: {a, b, c, n, o} s^MTF:"1"
2 σ:"a" i:1 l: {b, a, c, n, o} s^MTF:"11"
3 σ:"n" i:3 l: {a, b, c, n, o} s^MTF:"113"
4 σ:"a" i:1 l: {n, a, b, c, o} s^MTF:"1131"
5 σ:"n" i:1 l: {a, n, b, c, o} s^MTF:"11311"
6 σ:"a" i:1 l: {n, a, b, c, o} s^MTF:"113111"
7 σ:"c" i:3 l: {a, n, b, c, o} s^MTF:"1131113"
8 σ:"o" i:4 l: {c, a, n, b, o} s^MTF:"11311134"
9 σ:"c" i:1 l: {o, c, a, n, b} s^MTF:"114111341"
10 σ:"c" i:0 l: {c, o, a, n, b} s^MTF:"1131113410"
11 σ:"o" i:1 l: {c, o, a, n, b} s^MTF:"11311134101"
12 σ:"" i: l: {o, c, a, n, b} s^MTF:"11311134101"

FIGURE 2.2: An example of MTF-transform over the string t = `bananacocco`, alphabet $\Sigma = \{a, b, c, n, o\}$ and thus index set $\{0, 1, 2, 3, 4\}$.

described in Chapter **??**, instead the `bzip` deploys the structural properties of L^{MTF} to apply, in cascade, `RLE` and finally a Statistical encoder (such as Huffman, Arithmetic, or some of their variations, see Chapter **??**).

Figure 2.2 shows a running example for `MTF` over the string t = `bananacocco` which consists of 5 distinct symbols. It is evident that more frequent symbols are to the front of the list $\mathcal{L}^{MTF}$ and thus get smaller indices in s^{MTF}; this is the principle exploited in [3] to prove some compressibility bounds for the compressor that applies δ-coding over the integers in s^{MTF} (see Theorem 2.5 below).

We notice two local homogeneous substrings in s— "`anana`" and "`cocco`"— which show individually some redundancy in a few symbols. This is turned by `MTF` into two substrings of s^{MTF} consisting of small integers. The nice thing of the `MTF`-mapping is that homogeneous substrings which possibly involve different symbols (such as $\{a, n\}$ and $\{c, o\}$ in our running example), are changed into the homogeneous string s^{MTF} = `11311134101` which involves small numbers (mostly 0s and 1s) and is thus defined over a unique (integer) alphabet. The strong local-homogeneity properties of the column L in $bw(s)$ will make L^{MTF} full of 0s, so that the use the *single and simple compressor* `RLE` is worth and effective.

Inverting s^{MTF} is easy provided that we start with the same initial list $\mathcal{L}^{MTF}$ used for the `MTF`-transformation of s. A running example is provided in Figure 2.3. The algorithm maps an integer i in s^{MTF} onto the symbol which occurs at position i in $\mathcal{L}^{MTF}$, and then *moves* that symbol *to the front* of the list. This way the inversion algorithm mimics the transformation algorithm, by keeping both `MTF`-lists synchronized.

THEOREM 2.4 *Transforming a string s via* `MTF` *takes* $O(|s|)$ *time and* $O(|\Sigma|)$ *working space.*

A key concept for evaluating the compression performance of `MTF` is the one named *locality of reference*, which we have previously called *locally homogeneous substrings*. Locality of references

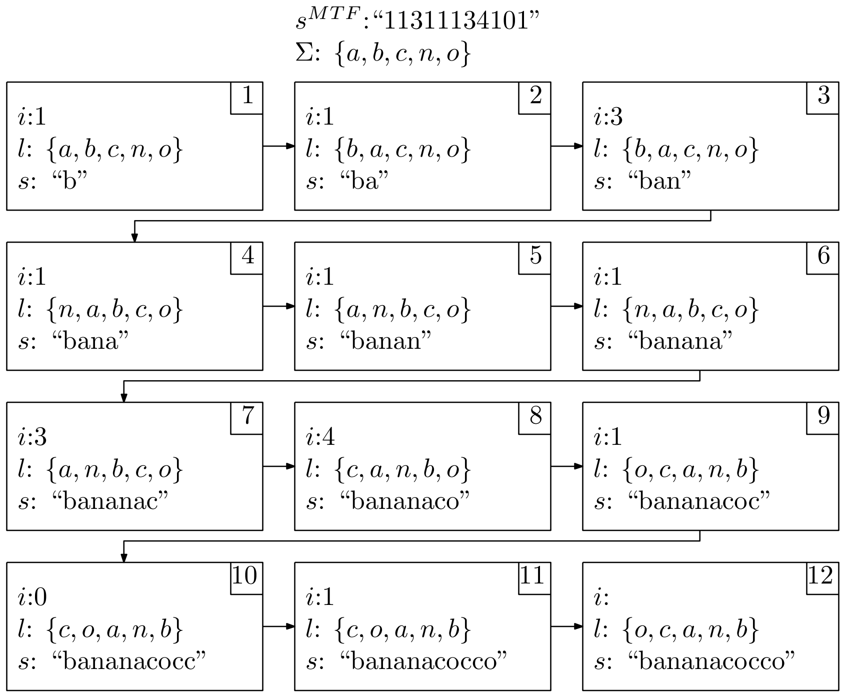

s^{MTF}:"11311134101"
Σ: $\{a,b,c,n,o\}$
1 i:1 l: $\{a,b,c,n,o\}$ s: "b"
2 i:1 l: $\{b,a,c,n,o\}$ s: "ba"
3 i:3 l: $\{b,a,c,n,o\}$ s: "ban"
4 i:1 l: $\{n,a,b,c,o\}$ s: "bana"
5 i:1 l: $\{a,n,b,c,o\}$ s: "banan"
6 i:1 l: $\{n,a,b,c,o\}$ s: "banana"
7 i:3 l: $\{a,n,b,c,o\}$ s: "bananac"
8 i:4 l: $\{c,a,n,b,o\}$ s: "bananaco"
9 i:1 l: $\{o,c,a,n,b\}$ s: "bananacoc"
10 i:0 l: $\{c,o,a,n,b\}$ s: "bananacocc"
11 i:1 l: $\{c,o,a,n,b\}$ s: "bananacocco"
12 i: l: $\{o,c,a,n,b\}$ s: "bananacocco"

FIGURE 2.3: An example of MTF-inversion over the string s^{MTF} = `11311134101`, starting with the list $\mathcal{L}^{MTF} = \{a, b, c, n, o\}$.

in s means that the distance between consecutive occurrences of the same symbol are small. For example the string `bananacocco` shows this feature in the substrings `anana` and `cocco`. We are perfectly aware that this concept is roughly specified but, for now, let us stick onto this abstract formulation which we will make mathematically precise next.

If the input string s exhibits locality of references, then the `MTF`-compressor (namely one that `MTF`-transforms s and then compresses someway the integers in s^{MTF}) performs better than the Huffman's compressor. This might appear surprising because Huffman's compressor is an *optimal prefix-code*; but, actually this is not surprising, because the `MTF`-compressor is not a prefix-code given that a symbol may be *dynamically* associated to different codewords. As an example look at Figure 2.2 and notice that symbol `c` gets three different numbers in s^{MTF}— i.e. $3, 1, 0$— and thus three different codewords.

Conversely, if the input string s does not exhibits any kind of locality of reference (e.g. it is a (quasi-)random string over the alphabet Σ), then the `MTF`-compressor performs much worse than Huffman's compressor. The following theorem (proved in [3]) makes this rough analysis precise by combining the `MTF`-transform with the γ-code. It goes without saying that the upper bound stated below could be made closer to the entropy H by substituting the γ-code with the δ-code or any other better universal compressor for integers (see Chapter **??**).

THEOREM 2.5 *Let n_c be the number of occurrences of a symbol c in the input string s, whose total length is $n = |s|$. We denote by $\rho_{MTF}(s)$ the average number of bits per symbol used by the compressor that squeezes the string s^{MTF} using the γ-code over its integers. It is $\rho_{MTF}(s) \leq 2H + 1$, namely that compressor can be no more than twice worse than the entropy of the source, and thus it* *cannot be more than twice worse than the Huffman compressor.*

Proof Let $p_1, \ldots, p_{n_c}$ be the positions in s where symbol c occurs. Clearly, between any two consecutive occurrences of c in s, say p_i and p_{i-1}, there may exist no more than $p_i - p_{i-1}$ distinct symbols (including c itself). So the index encoded by the MTF-compressor for the occurrence of c at position p_i is at most $p_i - p_{i-1}$. In fact, when processing position p_{i-1} the symbol c is moved to the front of the list, then it can move (at most) one position back per symbol processed subsequently, until we reach the occurrence of c at position p_i. This means that the integer emitted for the occurrence of c at position p_i is $\leq p_i - p_{i-1}$ (number of symbols processed). This integer is then encoded via γ-code, thus using no more than $\gamma(p_i - p_{i-1}) \leq 2(\log_2(p_i - p_{i-1})) + 1$ bits. As far as the first occurrence of c is concerned, we can assume that $p_0 = 0$, and thus encode it with at most $\gamma(p_1) \leq 2(\log_2 p_1) + 1$ bits. Overall the cost in bits for storing the occurrences of c in string s is

$$\begin{aligned}
&\leq \gamma(p_1) + \sum_{i=2}^{n_c} \gamma(p_i - p_{i-1}) \\
&\leq 2\log_2(p_1) + 1 + \sum_{i=2}^{n_c} (2\log_2(p_i - p_{i-1}) + 1) \qquad (2.1)\\
&\leq \sum_{i=1}^{n_c} (2\log_2(p_i - p_{i-1}) + 1).
\end{aligned}$$

By applying Jensen's inequality we can move the logarithm function outside the summation, so that a telescopic sum comes out:

$$\begin{aligned}
&\leq n_c \left(2\ \log_2\left(\frac{1}{n_c}\left(\sum_{i=1}^{n_c}(p_i - p_{i-1})\right)\right) + 1\right) \\
&= n_c \left(2\ \log_2\left(\frac{p_{n_c}}{n_c}\right) + 1\right) \qquad (2.2)\\
&\leq n_c \left(2\ \log_2\left(\frac{n}{n_c}\right) + 1\right)
\end{aligned}$$

where the last inequality comes from the simple observation that $p_{n_c} \leq n$. If now we sum for every symbol $c \in \Sigma$ and divide for the string length n, because the Theorem is stated as number of bits per symbol in s, we get:

$$\rho_{MTF}(s) \leq 2\left(\sum_{c\in\Sigma} \frac{n_c}{n} \log_2\left(\frac{n}{n_c}\right)\right) + 1 \ \leq \ 2H + 1 \qquad (2.3)$$

The thesis follows because H lower bounds the average codeword length of Huffman's code. ■

There do exist cases for which the MTF-based compressor performs much better than Huffman's compressor.

LEMMA 2.1 The compressor based on the combination of MTF-transform and γ-code can be better than Huffman compressor by the unbounded factor $\Omega(\log n)$, where n is the length of the string to be compressed.

Proof Take the string $s = 1^n 2^n \cdots n^n$ defined over an alphabet of size n and having length $|s| = n^2$. Since every symbol occurs n times, the distribution is uniform and thus Huffman uses for each symbol $\log_2 n$ bits. The overall compression of s by Huffman takes $\Theta(|s| \log n) = \Theta(n^2 \log n)$ bits. We used the asymptotic notation because constants here do not matter.

If we adopt the MTF-transform we get the string $s^{MTF} = 0^n\, 10^{n-1}\, 20^{n-1}\, 30^{n-1} \cdots$. Applying the γ-code, with the warning that integer i is encoded as $\gamma(i + 1)$ since i may be null, we get an output bit sequence of length $O(n^2 + n \log n)$. This is due to the fact that the $\Theta(n^2)$ integers equal to 0 are encoded as $\gamma(1) = 1$, thus taking 1 bit, whereas all other integers (they are $n - 1$ and smaller than n) are encoded with $O(\log n)$ bits each. ∎

2.2.2 The RLE transform

This is a very simple transform which maps every maximal contiguous substring of ℓ occurrences of symbol c into a pair $\langle \ell, c\rangle$. As an example, suppose we have to compress the following string which represents a line of pixels of a monochromatic bitmap (where W stands for "white" and B for "Black").

$$WWWWWWWWWWWBWWWWWWWWWWWWBBBBBWWWWWW$$

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

$$\underbrace{WWWWWWWWWWW}_{\langle 11,W\rangle}BWWWWWWWWWWWWBBBBBWWWWWW$$

We can proceed in the same way until the end of the line is encountered, thus obtaining the sequence of pairs $\langle 11, W\rangle, \langle 1, B\rangle, \langle 12, W\rangle, \langle 5, B\rangle, \langle 6, W\rangle$. It is easy to see that the encoding is lossless and simple to reverse. A remarkable observation is that if $|\Sigma| = 2$, as in the previous example, we can simply emit individual numbers (which indicate the run length) rather than pairs, plus the first symbol of the string to compress (W in the example), and still be able to decode back to the original string. In the previous example we could emit: $W, 11, 1, 12, 5, 6$.

RLE is actually more than a transform because it can be turned into a simple compressor by combining it with an integer encoder (as we did for MTF). Its best known context of application is fax transmission [19]: a sheet of paper is viewed as a binary (i.e. monochromatic) bitmap, this bitmap is first transformed by XORing two consecutive lines of pixels, then every output line is RLE-transformed and, finally, the integers are compressed via Huffman or Arithmetic (recall that in binary images, the alphabet has size two). Provided that the paper to be faxed is pretty regular, the XORed lines will be full of 0s, and thus their RLE-transformation will originate few runs, whose compression will be significant. Nothing prevents to apply this argument to colored images, but the XORing of contiguous lines will get less 0s. More sophisticated methods are needed in this setting!

RLE can perform better or worse than the Huffman scheme: this depends on the message we want to encode. The following lemma shows that RLE can be much better than Huffman, by adopting the same string we used to prove Lemma 2.1.

LEMMA 2.2 The compressor based on the combination of the RLE-transform and the γ-code can be better than Huffman's compressor by the unbounded factor $\Omega(n)$, where n is the length of the string to be compressed.

Proof Take the string $s = 1^n 2^n \cdots n^n$, and recall from the proof of Lemma 2.1 that Huffman's code takes $\Theta(n^2 \log n)$ bits to compress it. If we apply the RLE-transform on the string s we get the string $s^{RLE} = \langle 1, n\rangle \langle 2, n\rangle \langle 3, n\rangle \cdots \langle n, n\rangle$. The γ-code over the integers of s^{RLE} will use $O(\log n)$ bits per pair and thus $O(n \log n)$ bits overall. ■

But there are cases, of course, in which RLE-compressor can perform much worse than Huffman's. Just consider a string s in which runs are short, namely any English text!

2.3 The bzip compressor

As we anticipated in the previous sections, the compressor bzip hinges on the sequential combination of three transforms— i.e. BWT, MTF and RLE— which produce an output that is suitable to be highly squeezed by a classical statistical compressor— such as Huffman, Arithmetic, or some of their variations. The most time consuming step in this sequence is the computation/inversion of the BWT, both at compression/decompression time respectively. This is not just in terms of number of operations, which are $O(n)$ for all transforms and the statistical compressor, but because of the pattern of memory accesses that is very scattered thus inducing a lot of cache misses. This is an issue that we will comment more deeply next.

The key property that makes bzip work well is the local homogeneity of the string produced by the Burrows-Wheeler transform. To convince yourself of this property let us consider the input string s and one of its substrings w, which is assumed to occur n_w times in s. Say $c_1, \ldots, c_{n_w}$ are the symbols preceding the occurrences of w in s. Now given the way $bw(s)$ is computed, we can conclude that all rows prefixed by the substring w in M' (they are of course n_w) are contiguous, but possibly shuffled depending on the symbols which follow w in each of those rows. In any case, the symbols c_i which precede w are contiguous in L (shuffled, accordingly), and thus constitute a substring of L. If the string s is Markovian, in the sense that symbols are emitted based on their previous ones (like linguistic texts), then the symbols c_i are expected to be a few distinct ones, and this property holds the more the longer is w. Given that w can be of any length, we say that L is locally homogeneous because, as we observed, picking any of its substrings it will possibly show few distinct symbols. This homogeneity is the core property that makes the subsequent steps in bzip very effective in compressing L.

For the sake of clearness, let us consider the following example which runs bzip over the string s defined as the concatenation for three times of the string mississippi. This way a high repetitiveness is induced over s. The first step consists of computing $bw(s)$, for space reasons we do not detail this computation but just show the result that can be checked by hands: $L =$ `ippp ssss ssmm miip ppii isss sssi iiii i`, where groups of 4 symbols simplify the reading, and $r = 15$ (counting from 0). The next step is to apply the MTF-transform to L starting with a list $\mathcal{L}^{MTF} = \{i, m, p, s\}$ which consists of the distinct symbols appearing in s. The storage of r (using 4-8 bytes) and of $\mathcal{L}^{MTF}$ (plainly) occurs in the preamble of the compressed file. The result of MTF is the string $L^{MTF} =$ `0200 3000 0030 0303 0010 0300 0001 0000 0`. Notice that runs of equal symbols generate runs of 0, except for the first symbol of the run which is mapped to an integer which represents its position in $\mathcal{L}^{MTF}$ at the time of its processing.

The first specialty introduced by bzip is that RLE in not applied onto L^{MTF} but on a slightly different string in which all numbers, except 0, are increased by one: $L^{MTF+} =$ `0300 4000 0040 0404 0040 0400 0002 0000 0`. The ratio behind this change relies on the way runs of 0 are encoded. In fact bzip does not apply RLE to runs of all possible symbols, rather it applies a restricted variant, called RLE0, which squeezes only the runs consisting of 0s. So the construction of L^{MTF+}, instead of L^{MTF}, can be looked as a smart way to reserve the integers 0 and 1 for the binary encoding of the 0-runs. More precisely, the run `00000` consisting of 5 occurrences of 0s is encoded according to the following scheme, known as Wheeler's code: the length is increased by 1, hence $5 + 1 = 6$, then the binary encoding of 6 is returned, hence `110`, and finally the first bit (surely 1) is removed thus

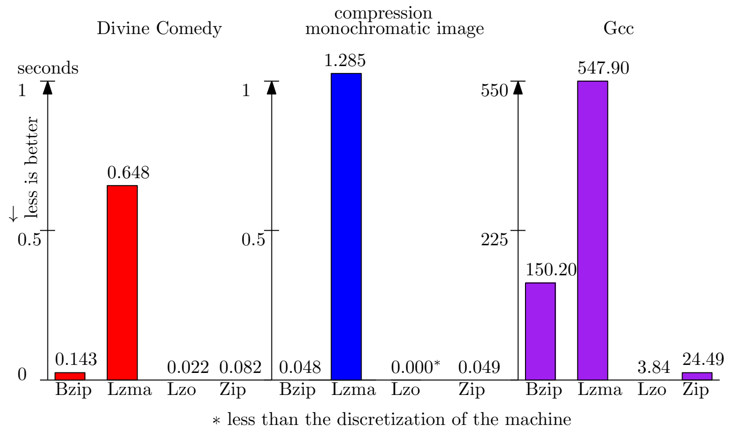


FIGURE 2.4: Compression speed (lower is better)

outputting the binary sequence `10`. The first increment guarantees that the (increased) run-length is at least 2, and thus it is represented in at least 2 binary digits in which the first one is surely a 1. So the 1-bit removal leaves at least one bit to be output. Decoding Wheeler's code is easy, just repeat the above steps in reverse order.

The key property of Wheeler's code is that the output bit sequence consists of no more digits than numbers in L^{MTF+}, so this step can be considered as a preliminary compression, which is more and more effective as longer and longer are the 0-runs in L^{MTF+}. The binary output for the sequence of our running example above is: `RLE0 = 0314 1041 4031 4141 0210`. It is evident that the decompressor can easily identify the run's encodings because they consist of maximal sequences of 0s and 1s; recall that these numbers have been reserved explicitly for this purpose.

Finally `RLE0` is compressed by using a Statistical compressor that operates on an alphabet which consists of integers in the range $[0, |\Sigma|]$. We observe that the alphabet size is $|\Sigma| + 1$, rather than $|\Sigma|$, because of the increment we did onto the non-null numbers in L^{MTF} to derive L^{MTF+}. The reader can look at the home page of `bzip2` [18] for further details, especially regarding the statistical-encoding step.

Just to have an idea of the power of the BW-Transform, we report here few experiments that compare a BWT-based compressor[6] against a few other well-known compression algorithms such as LZMA (Lempel-Ziv-Markov chain algorithm)[7], LZO1A (LZ-Oberhumer zip)[8], and the classic ZIP[9]. Tests were run in a commodity PC with 2GB RAM (using ramfs), AMD Athlon(tm)X2 Dual-Core QL-64, running Linux. We used three datasets of different type and size: La Divina Commedia, a raw monochromatic non-compressed image, and the package `gcc-4.4.3`. These experiments are not intended to provide an official comparison among these compressors, rather to give the reader a flavor of the differences in performance among them.

From Figures 2.4–2.6 we can easily draw some conclusions. First of all, `LZMA` is bad on these datasets; it has ever the worst compression time and it does not reach the best compression rate.

[6]We used `http://www.bzip.org, version 1.0.5-r1`

[7]We used the "lzip" package from `http://www.nongnu.org, version 1.10`

[8]We used the version `1.02_rc1-r1`. 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.

[9]We used the package from `http://www.info-zip.org/`, version 3.0.

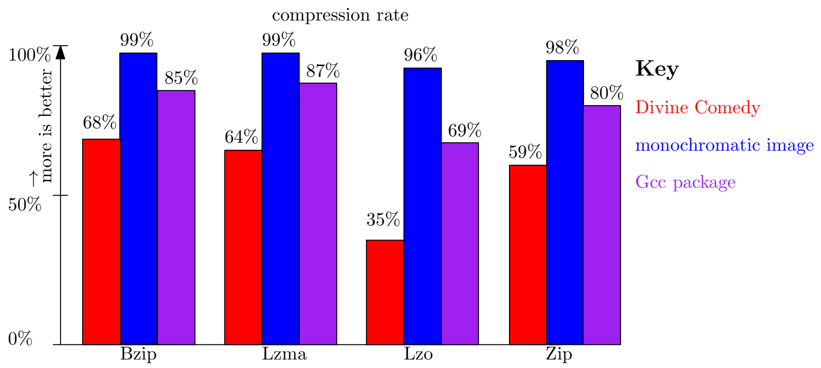


FIGURE 2.5: Compression savings (higher is better)

`bzip2` seems to be quite in the middle: sometimes it takes a lot of time to compress, but it reaches the best compression rate. By considering the decompression time, `bzip2` slows down too much as the size of the file grows, and this is not a surprise because of its algorithmic structure. Perhaps the best solution seems to be `zip`: it takes short time to compress/decompress and reaches a very good compression rate. `LZO` is the fastest algorithm we tested, but unfortunately its compression ratio seems to be not appealing, and this is due to the fact that it was engineered for speed rather than for space savings. We restate here that these considerations are not definitive for those compressors, they are just suitable for giving a glimpse on them about these three datasets. For more official and robust comparisons we refer the reader to the page of Matt Mahoney.[10]

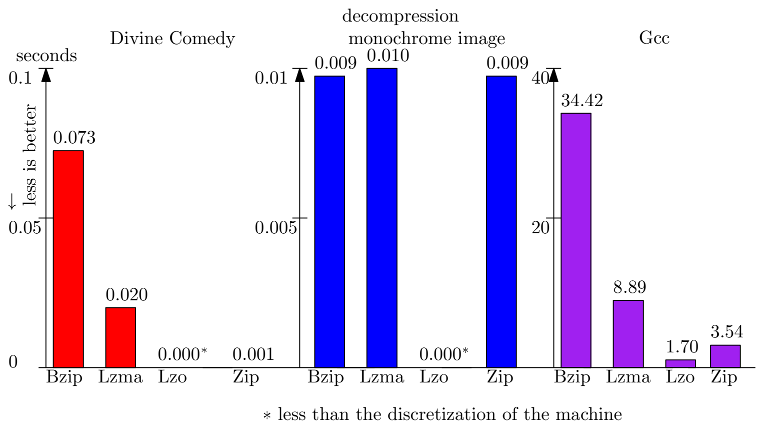


FIGURE 2.6: Decompression speed (lower is better)

We are left with the problem of constructing the Burrows-Wheeler forward transform given that,

[10]`http://mattmahoney.net/dc/dce.html`

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 2.7: Suffix Array *versus* sorted rotated matrix M' over the string s = `abracadabra$`.

as we observed above, we cannot construct explicitly the rotation matrix $\mathcal{M}$, and a fortiori its sorted version $\mathcal{M}'$, because this would take $\Theta(n^2)$ working space for a text s of length n. That is the why most `BWT`-based compressors exploit some "tricks" in order to avoid the construction of these matrices. One such "trick" involves the usage of Suffix Arrays, which were described in Chapter **??**, where we also detailed several algorithms to build them efficiently. The construction of `BWT` deploys one of them[11] and this use motivates the increased interest in the literature about the Suffix-Array construction problem (see e.g. [14, 16, 1]).

To see why Suffix Arrays and `BWT` are connected, let us consider the following example. Take the string `abracadabra$` and compute its Suffix Array $[11, 10, 7, 0, 3, 5, 1, 4, 6, 9, 2]$. Figure 2.7 summarizes these data structures for the running example at hand. The first four columns show the suffixes of the string s and its suffix array $\mathcal{SA}$. The fifth column shows the corresponding sorted-rotated matrix $\mathcal{M}'$ with its last column L. It is easy to notice that sorting suffixes is equivalent to sorting rows of $\mathcal{M}$, given the presence of the sentinel symbol `$`. The reader can check that the formula below ties $\mathcal{SA}$ with L:

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

This means that every symbol $L[i]$ equals to the symbol of s that precedes the suffix $\mathcal{SA}[i]$ which prefixes the ith row of M'. If, however, that suffix is the whole string s (thus $\mathcal{SA}[i] = 0$), then `$` will be used as preceding symbol.
So, given the suffix array of string s, it takes only linear time to derive the string L. We have therefore proved the following:

THEOREM 2.6 *Let us given an input string s, constructing bw(s) takes a time/IO complexity which is the one of Suffix Array construction. By using the Skew Algorithm, the overall cost of building bw(s) is optimal in several model of computations. In particular, this is O(n) for the RAM*

[11]M. Burrows: "So I enlisted his help in finding ways to execute the algorithm's sorting step efficiently, which involved considering constant factors as much as asymptotic behavior. We tried many things, only some of which made it into the paper, but we met my goals: we showed that the algorithm could be made fast enough to see practical use on modern machines...".

model and $O(\texttt{Sort}(n))$ for the external-memory model, where `Sort` *is the I/O-cost of sorting n atomic items in a model in which M is the size of the internal memory and B is the disk-page size.*

We conclude this section by observing that, in practice, $bw(s)$ is costly to be computed so that its implementations divide the input text into blocks and then apply the transform *block-wise*. This is the reason why these compressors are called *block-wise compressors*. Likewise dictionary-based compressors, the size of the block impacts onto the trade-off compression ratio *versus* compression speed; but, unlike dictionary-based compressors, this impacts unfavorably also onto the decompression speed which is slowed down when working on larger and larger blocks. Anyway, the current implementation of `bzip2` allows to specify the size of the block at compression time with the parameter `-1`, . . ., `-9`, that actually indicate a block of size 100Kb, . . ., 900Kb.

2.4 On compression boosting$^\infty$

Let us first recall the notion of entropy as a measure of uncertainty (or information) associated with a random source S emitting n symbols $\{x_1, \ldots, x_n\}$ with probabilities $p(x_i)$:

$$H(S) = \sum_{i=1}^{n} p(x_i) \times \frac{1}{\log p(x_i)}$$

The previous formula is often called 0th order entropy, and it is indicated with H_0, because it is computed with respect to the probabilities of the single symbols emitted by the source S, without exploiting any context (or equivalently, exploiting an empty context, hence of length 0). Given that we are dealing with compressors and real strings, most evaluations of their performance drop probabilities in favor of frequencies: hence $p(x_i)$ is the ratio between the number of occurrences of x_i in the input string s and the total length of s, say $|s|$. Clearly, in this setting any string containing $n/2$ symbols a and $n/2$ symbols b has entropy $H_0 = 1$ independently of the fact that it is either a random string or the regular string $\mathtt{a}^{n/2}\mathtt{b}^{n/2}$.

A more precise modeling of the information content of a string s (of of its uncertainty) can be obtained by measuring the entropy over blocks of k-symbols. This is called kth order (empirical) entropy of the string s, and can be computed as follows:

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

where w_s represents the set of all symbols that follow w in s. Clearly $H_k(s) \leq H_0(s)$, but it can be much smaller, and for $|s|$ and k that go to ∞ this value converges to the entropy of the source that emitted s.

We are interested in this formula because it suggests a way to design a compressor that achieves $H_k(s)$ starting from a compressor that achieves H_0 of its input strings. This kind of algorithm is called a *Compression Booster* because it is able to boost a compression performance up to H_0 into a compression performance up to H_k. The algorithmic tool to achieve this is, surprisingly, the Burrows-Wheeler Transform [6]. In order to illustrate this innovative and powerful idea, let us consider a generic 0-order statistical compressor C_0 whose performance, in bits per symbol, over a string t is bounded by $H_0(t) + f(|t|)$. We notice that function $f(|t|) = 2/(|t|)$ for the Arithmetic coding and it is $f(|t|) = 1$ for Huffman coding (see Chapter **??**).

In order to turn C_0 into an effective kth order compressor C_k, we proceed as follows.

- Compute the Burrows-Wheeler Transform $bw(s)$ of the input string s.
- Take all possible substrings w of the string t, and partition the column L in such a way that L_w is formed by the last symbols of rows prefixed by w.