12

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 [3, 7],[1] and since it was rejected from the Data Compression Conference (as Mike Burrows stated in its foreword to [8][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* [8]. 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` [16] or the `booster` [5]) as well as a new powerful family of compressed indexes (such as `FM-index` [6], and many variations [13]). 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.

12.1 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 12.2. 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 12.3.

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!

12.1.1 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 12.3 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 [1], 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 12.1 shows the rotated matrix $\mathcal{M}$ built over s; whereas the right side of Figure 12.1 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.

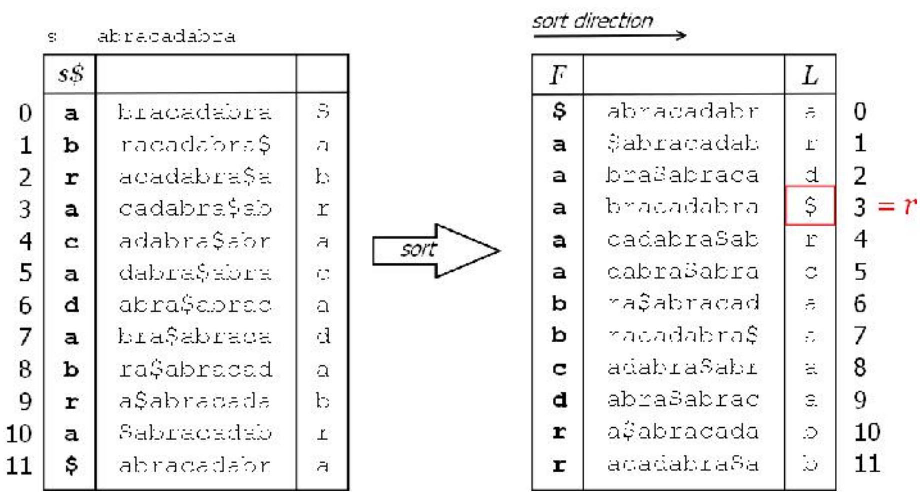

FIGURE 12.1: Forward Burrows-Wheeler Transform of the string s = `abracadabra`.

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.

12.1.2 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 12.1 *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 12.1, row 2 of $\mathcal{M}'$ is prefixed by `abra`, followed by `$abracad`.

Property 12.1 *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 12.1 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 12.2 *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 $[0, n-1]$.

Algorithm 12.1 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

DEFINITION 12.1 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.

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 12.1. 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 12.2, as Algorithm 12.1 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 12.2 allows to map the current symbol occurring in L (initially $L[0]$) to its corresponding copy in F; then Property 12.1 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

[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 12.2 Reconstructing s from $bw(s)$

Derive column L from $bw(s)$;
Compute $LF[0, n-1]$ from L;
$k = 0$; $i = n - 1$;
while $i \geq 0$ **do**
  $s[i] = L[k]$;
  $k = LF[k]$;
  $i--$;
end while

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 12.2.

As an example, refer to Figure 12.1 where we have that $L[0] = s[n-1] = \mathtt{a}$, and execute the while-cycle of Algorithm 12.2. Definition 12.1 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 $F[1] = \mathtt{a}$), and the preceding symbol in s is thus $L[1] = \mathtt{r}$. These two basic steps are repeated until the whole string s is reconstructed. Just continuing the previous running example, we have that $L[1] = \mathtt{r}$ is LF-mapped to the symbol in F at position $LF[1] = 10$ (and indeed $F[10] = \mathtt{r}$). 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 $L[10] = \mathtt{b}$. And so on...

THEOREM 12.3 *The original string s can be reconstructed from its* BWT *in $O(n)$ time and space. Algorithm 12.2 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. [15, 10, 11, 9]), 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!

12.2 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 [2]. In general the compression performance of those algorithms is very poor: BWT is magically their killer application!

12.2.1 The Move-To-Front transform

The MTF-transformation [2] 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.

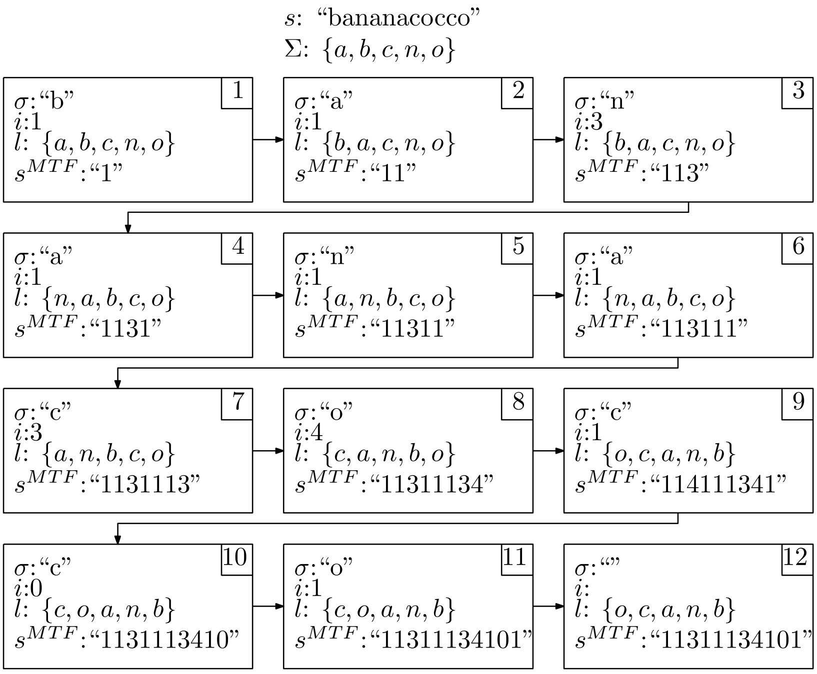

FIGURE 12.2: An example of MTF-transform over the string t = `bananacocco`, alphabet $\Sigma = \{a, b, c, n, o\}$ and thus index set $\{0, 1, 2, 3, 4\}$.

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, described in Chapter 9, 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 10).

Figure 12.2 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 [2] to prove some compressibility bounds for the compressor that applies δ-coding over the integers in s^{MTF} (see Theorem 12.5 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 12.3. 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.

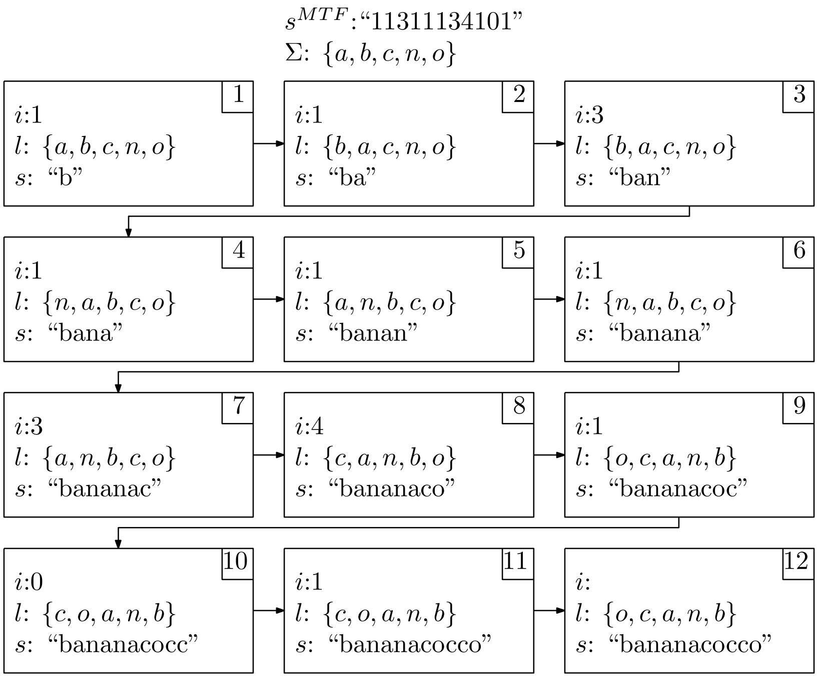

FIGURE 12.3: An example of MTF-inversion over the string s^{MTF} = `11311134101`, starting with the list $\mathcal{L}^{MTF} = \{a, b, c, n, o\}$.

THEOREM 12.4 *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 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 12.2 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 [2]) 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 9).

THEOREM 12.5 *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 $\rho_{MTF}(s) \leq 2H + O(|\Sigma|)$, 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 s_i is at most $s_i - s_{i-1}$. In fact, when processing position s_{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 s_i. This means that the integer emitted for the occurrence of c at position s_i is $\leq s_i - s_{i-1}$ (number of symbols processed). This integer is then encoded via γ-code, thus using no more than $\gamma(s_i - s_{i-1}) \leq 2(\log_2(s_i - s_{i-1})) + 1$ bits. As far as the first occurrence of c is concerned, we can assume that $s_0 = 0$, and thus encode it with at most $\gamma(s_1) \leq 2(\log_2 s_1) + 1$ bits. Overall the cost in bits for storing the occurrences of c in string s is

$$\begin{aligned} &\leq \gamma(s_1) + \sum_{i=2}^{n_c} \gamma(s_i - s_{i-1}) \\ &\leq \log_2(s_1) + 1 + \sum_{i=2}^{n_c} \log_2(s_i - s_{i-1}) + 1. \end{aligned} \tag{12.1}$$

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

$$\begin{aligned} &\leq 2\, n_c\, \log_2\left(\frac{1}{n_c}\left(s_1 + \sum_{i=2}^{n_c}(s_i - s_{i-1})\right)\right) + 1 \\ &= 2\, n_c\, \log_2\left(\frac{s_{n_c}}{n_c}\right) + 1 \\ &\leq 2\, n_c\, \log_2\left(\frac{n}{n_c}\right) + 1 \end{aligned} \tag{12.2}$$

where the last inequality comes from the simple observation that $s_{n_c} \leq n$. If now we sum for every symbol $c \in \Sigma$ and divide for the string length n we get:

$$\begin{aligned} \rho_{MTF}(s) &\leq 2\left(\sum_{c \in \Sigma} \frac{n_c}{n} \log_2\left(\frac{n}{n_c}\right)\right) + 1 \\ &\leq 2H + O(|\Sigma|) \end{aligned} \tag{12.3}$$

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 12.1 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. ∎

12.2.2 The RLE transform

This is a very simple transform which maps every maximal contiguous substring of ℓ occurrences of symbol c into a pair $\langle c, \ell \rangle$. 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 [1]: 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 12.1.

LEMMA 12.2 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 $\Omega(\frac{n}{\log n})$, where n is the length of the string to be compressed.

Proof Take the string $s = 1^n 2^n \cdots n^n$, and recall from the proof of Lemma 12.1 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!

12.3 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

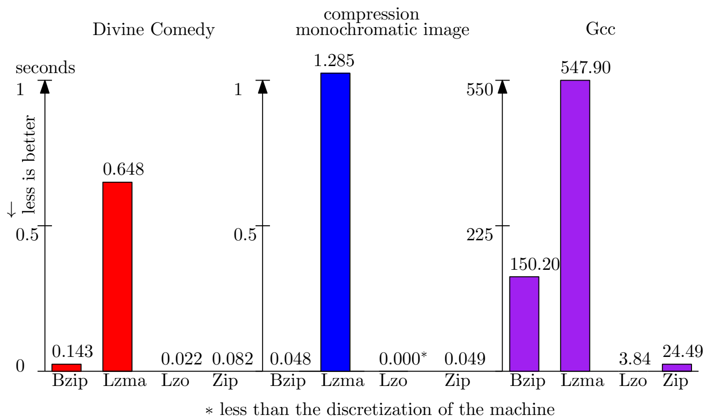

FIGURE 12.4: Compression speed (lower is better)

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` [16] 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 12.4–12.6 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.

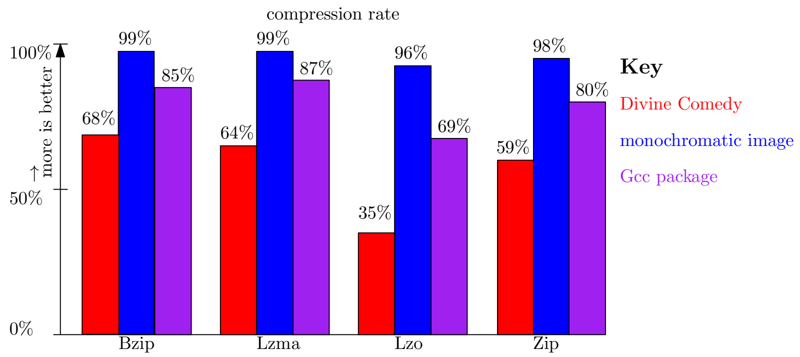

FIGURE 12.5: Compression savings (higher is better)

`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]

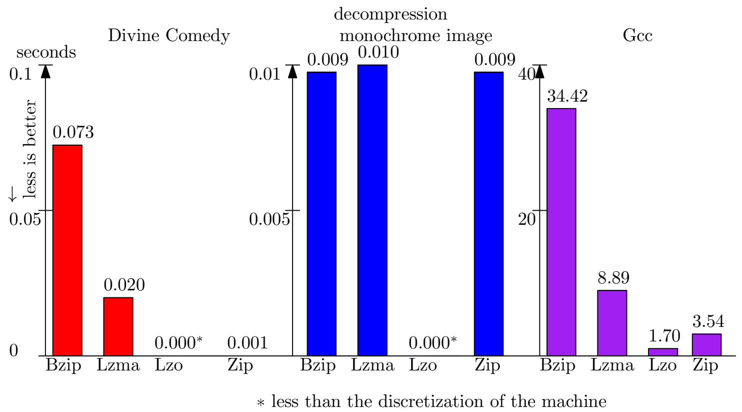

FIGURE 12.6: Decompression speed (lower is better)

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 12.7: 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 8, 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. [12, 14, 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 12.7 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 12.6 *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.

12.4 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 $\mathtt{a}$ and $n/2$ symbols $\mathtt{b}$ 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 [5]. 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 10).

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.

- Compress each L_w with C_0, and concatenated the output bit sequences by alphabetically increasing w (or, equivalently, by occurrence of L_w in L).

It is immediate to notice that L_w is a substring of L, and not a subsequence, because rows prefixed by w in M' are contiguous. Given the LCP-array of string s the partitioning of L takes linear time (see Chapter 8) and thus it does not impact onto the efficiency of the final compressor C_k. As far as the compression performance per symbol is concerned, we easily derive that it can be bounded as:

$$\frac{1}{|s|} \sum_{w_s \in \Sigma^k} |w_s| \; (H_0(w_s) + f(|w_s|) \;\; = \;\; H_k(s) + O(|\Sigma|^k)$$

where we have applied the definition of $H_k(s)$ onto the summation of the $H_0(w_s)$, and the fact that $f(|w_s|) \leq 1$. It is clear that the more effective is the 0-th order compressor, the more it is closer to H_0, the more evanishing is the term $f(|w_s|)$ and thus negligible is the additive term $O(|\Sigma|^k)$. In [5] the authors showed that one actually does not need to fix k, since it does exist a Compression Booster which identifies in optimal $O(|s|)$ time a partition of L which achieves a compression ratio which is better than the one obtained by C_k, for any possible $k \geq 0$. The algorithm is elegant and not much involved, but it would require some space to be described in sufficient details, so that we refer the interested reader to that paper.

12.5 On compressed indexing$^\infty$

We have already highlighted the bijective correspondence between the rows of the rotated matrix M and the suffixes of the string s, as well as the strong relationship between the string L and the suffix array built on s (see Figure 12.7). This is relationship is at the core of FM-index's design, which has been the first compressed full-text index to achieve efficient substring search and space occupancy up to the k-th order empirical entropy of the indexed string. Given these features we can look at the FM-index as the *compressed version* of a suffix array, or as the *searchable version* of `bzip`-compressed format. The nature of these notes does not allow to dig into the technical details of the FM-index, so in the rest of this chapter we will just fly over its technicalities and concentrate on the main algorithmic ideas; the interested reader may look at the seminal paper [6] and the survey [13].

In order to simplify the presentation we distinguish between three basic operations, which underlie the design of many search toolbox:

- `Count`(P) returns the *range of rows* $[first, last]$ in M (and thus suffixes in the suffix array) which are prefixed by the string P. The value $(last - first + 1)$ accounts for the number of these pattern occurrences.
- `Locate`(P) returns the *list* of all positions in s where P occurs (possibly unsorted).
- `Extract`(i, j) returns the substring $s[i, j]$ by accessing its compressed representation in FM-index.

For example, in Figure 12.7 for the pattern $P =$ `ab` we have $first = 2$ and $last = 3$ for a total of two occurrences. These two rows correspond, as the picture clearly illustrates, to the two suffixes $s[0,]$ and $s[7,]$ which are prefixed by P.

Let us start from the description of `Count`(P). The retrieval of the rows $first$ and $last$ is not implemented via a binary search, as it occurred in Suffix Arrays (see Chapter 8), but it uses a peculiar search method which deploys the column L, the array C (which counts in $C[c]$ the number of occurrences in s of all symbols smaller than c) and an additional data structure which supports efficiently the very basic counting `Rank`(c, k) which reports the number of occurrences of the symbol c in the string prefix $L[0, k-1]$. All data structures L, C and `Rank` can be stored compressed and

Algorithm 12.3 Counting the occurrences of pattern $P[0, p-1]$ in s

```
i = p − 1, c = P[p − 1];
first = C[c], last = C[c + 1] − 1;
while (first ≤ last) and i ≥ 1 do
    c = P[i − 1];
    first = C[c] + Rank(c, first − 1);
    last = C[c] + Rank(c, last) − 1;
    i = i − 1;
end while
return (first, last).
```

still retrieve efficiently their entries: namely access $L[i]$ or $C[c]$, or answer $\mathtt{Rank}(c, k)$. The literature offers many solutions for this problem (see e.g. some classic results [6, **?**, **?**, 13]), here we report some of them (possibly not the best ones at the time we write these notes):

LEMMA 12.3 Let $s[0, n-1]$ be a string over alphabet Σ and let L be its BW-Transform.

- For $|\Sigma| = O(\mathtt{polylog}(n))$, there exists a data structure which supports Rank queries on L in $O(1)$ time using $nH_k(s) + o(n)$ bits of space, for any $k = o(log_{|\Sigma|}n)$, and retrieves any symbol L in the same time bound.
- For general Σ, there exists a data structure which supports Rank queries on L in $O(\log\log|\Sigma|)$ time, using $nH_k(s) + o(n\log|\Sigma|)$ bits of space, for any $k = o(log_{|\Sigma|}n)$, and retrieves any symbol of L in the same time bound.

This means that Rank can be implemented in constant, or almost constant time and in space which is very much close to the k-th order entropy of the string s we wish to index. The array C takes only $O(|\Sigma|)$ space, which is negligible for real alphabets. This means that this ensemble of data structures is very compact indeed.

We are left to show how this ensemble allows us to implement $\mathtt{Count}(P)$. Algorithm 12.3, usually called *backward search*, reports the pseudo-code of such implementation which takes $O(p)$ optimal time, working in p constant-time phases numbered from $p-1$ to 0. Each phase preserves the following invariant: *At the i-th phase, the parameter "first" points to the first row of the sorted rotated matrix M' prefixed by* $P[i, p-1]$ *and the parameter "last" points to the last row of M' prefixed by* $P[i, p-1]$. Initially the invariant is true by construction: $F[C[c]]$ is the first row in M' starting with c, and $F[C[c+1]-1]$ is the last row in M' starting with c (recall that rows are numbered from 0).[12] As running example take $P = \mathtt{ab}$, so at the beginning we have: $C[b] = 6$ and $C[b+1] = C[c] = 8$ in Figure 12.7 and thus $[6, 7]$ is the range of rows prefixes by b before that the backward-search starts.

At each subsequent phase, Algorithm 12.3 has found the range of rows $[first, last]$ prefixed by $P[i, p-1]$. Then it determines the new range of rows $[first, last]$ prefixed by $P[i-1, p-1] = P[i-1]\,P[i, p-1]$ by proceeding as follows. First it determines the first and last occurrence of the symbol $c = P[i-1]$ in the substring $L[first, last]$ by deploying the function Rank properly queried. Specifically $\mathtt{Rank}(c, first-1)$ counts how many occurrences of c precede position $first$ in L, and $\mathtt{Rank}(c, last)$ counts how many occurrences of c precede position $last$ in L. These values are

[12] We adopt the shorthand notation that $C[c+1]$ is the entry storing the counting for the symbol following c in the alphabet.

then used to compute the LF-mapping of those first/last occurrences of c. In fact Property 12.2 and Definition 12.1 imply the equality $LF[i] = C[L[i]] + \mathtt{Rank}(L[i], i)$. This means that the computation of the LF-mapping can occur efficiently and succinctly provided that we store compactly the data structure that implements $\mathtt{Rank}(c, k)$. For a formal proof that this mapping actually retrieves the new range of rows $[first, last]$ prefixed by $P[i-1, p-1]$ we refer the reader to the seminal publication [6]. Here we make an example to convince experimentally the reader that everything works fine. Refer again to Figure 12.7 and consider, as before, the pattern $P = \mathtt{ab}$ and the range $[6, 7]$ of rows in M' prefixes by $P[2] = \mathtt{b}$. Now pick the previous pattern symbol $P[1] = \mathtt{a}$, Algorithm 12.3 computes $\mathtt{Rank}(a, 5) = 1$ and $\mathtt{Rank}(a, 7) = 3$ because $L[0, first-1]$ contains 1 occurrences of a and $L[0, last]$ contains 3 occurrences of a. So the algorithm computes the new range as: $first = C[a] + \mathtt{Rank}(a, 5) = 1 + 1 = 2$, $last = C[a] + \mathtt{Rank}(a, 7) - 1 = 1 + 3 - 1 = 3$, which is indeed the contiguous range of rows prefixed by the pattern $P = \mathtt{ab}$.

After the final phase (i.e. $i = 0$), $first$ and $last$ will delimit the rows of M' containing all the suffixes prefixed by P. Clearly if $last < first$ the pattern P does not occur in s. The following theorem summarizes what we have sketched.

THEOREM 12.7 *Given a string $s[0, n-1]$ drawn from an alphabet Σ, there exists a compressed index that takes $O(p \times t_{\mathrm{rank}})$ time to support $\mathtt{Count}(P[0, p-1])$, where t_{rank} is the time cost of a single* Rank *operation over the BW-transform L of string s. The space usage is bounded by $nH_k(s) + o(n \log |\Sigma|)$ bits, for any $k = o(\log_{|\Sigma|} n)$.*

The interesting corollary of the Theorem above is that, by plugging Lemma 12.5, we get an implementation of $\mathtt{Count}(P)$ which takes optimal $O(p)$ time and compressed space. However this solution suffers of I/O-inefficiency because every phase elicits some cache/IO misses due to the jumping around L and Rank. Several efforts have been dedicated in the literature to make FM-index cache-oblivious or cache-aware but yet an equally elegant solution for those issues is still missing.

Let us now describe the implementation of the location of pattern occurrences via procedure $\mathtt{Locate}(P)$. For a fixed parameter μ, we sample the rows i of M' which correspond to suffixes that start at positions of the form $pos(i) = j\mu$, for $j = 0, 1, 2, \ldots$. Each such pair $\langle i, pos(i)\rangle$ is stored explicitly in a data structure $\mathcal{P}$ that supports membership queries in constant time (on the *row*-component). Now, given a row index r, the value $pos(r)$ can be derived immediately if $r \in \mathcal{P}$ is a sampled row; otherwise, the algorithm computes $j = LF^t(r)$, for $t = 1, 2, \ldots$, until j is a sampled row and thus is found in $\mathcal{P}$. In this case, $pos(r) = pos(j) + t$. The sampling strategy ensures that a row in $\mathcal{P}$ is found in at most μ iterations, and thus the occ occurrences of the pattern P can be located via $O(\mu \times occ)$ queries to the Rank-data structure.

THEOREM 12.8 *Given a string $s[0, n-1]$ drawn from an alphabet Σ, there exists a compressed index that takes $O(\mu\, occ)$ time and $O(\frac{n}{\mu} \log n)$ bits of space to support* $\mathtt{Locate}(P)$*, provided that the range $[first, last]$ of rows prefixed by P is available.*

By fixing $\mu = \log^{1+\epsilon} n$, the solution above takes poly-logarithmic time per occurrence and sub-linear space (in bits). Trade-offs are possible and they were investigated [4].

Not much surprising is that $\mathtt{Count}(P)$ can be adapted to implement the last basic operation supported by FM-index: $\mathtt{Extract}(i, j)$. Let r be the row of M' prefixed by the suffix $s[j, n-1]$, and assume that the value of r is known. The algorithm sets $s[j] = F[r]$ and then starts a cycle which sets $s[j-1-t] = L[LF^t[r]]$, for $t = 0, 1, \ldots, j-i-1$. The main idea underlying this cycle is that we repeatedly compute the LF-mapping (implemented via the Rank-data structure) of the current symbol, so jumping backward in s starting from $s[j-1]$ (in fact $s[j]$ is found via F-array). We stop after $j - i - 1$ steps, when we have reached $s[i]$. This approach reminds the one we have taken in BWT-inversion, the difference relies in the fact that the array LF is not explicitly available, but its entries are generated on-the-fly via Rank-computations. This guarantees still constant-time access to LF-array, but succinct space storage (thanks to Lemma).

Given the appealing asymptotical performance and structural properties of the FM-index, several authors have investigated its practical behavior by performing an extensive set of experiments. We invite the reader to check paper [4] and to look at the Pizza&Chili's site[13] which offers many implementations of compressed indexes, not just FM-index. Experiments have shown that the FM-index is compact (its space occupancy is close to the one achieved by `bzip`), it is fast in counting the number of pattern occurrences (few micro-secs per pattern's symbol), and the cost of their retrieval is reasonable when they are few (about 100k occurrences/sec). In addition the FM-index allows to trade space occupancy for search time by choosing the amount of auxiliary information stored into it (i.e. parameter μ and few other parameters arising in the implementation of Rank). As a result the FM-index combines compression and full-text indexing: like `bzip` it encapsulates a compressed version of the original file (accessible via Extract); like suffix trees and suffix arrays it allows to search for arbitrary patterns (via Count and Locate). Everything works by looking only at a small portion of the compressed file, thus avoiding its full decompression.

References

[1] Donald Adjeroh, Tim Bell, and Amar Mukherjee. *The Burrows-Wheeler Transform: Data Compression, Suffix Arrays, and Pattern Matching*. Springer, 2008.

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

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

[4] P. Ferragina, R. Gonzalez, G. Navarro, R. Venturini. Compressed Text Indexes: From Theory to Practice. *ACM journal on Experimental Algorithmics*, vol. 13, art. 12, Febbraio 2009.

[5] Paolo Ferragina, Raffaele Giancarlo, Giovanni Manzini, Marinella Sciortino. Compression boosting in optimal linear time. *Journal of the ACM*, 52(4):688–713, 2005.

[6] Paolo Ferragina, Giovanni Manzini. Indexing compressed texts. *Journal of the ACM*, 52(4):552–581, 2005.

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

[8] Paolo Ferragina, Giovanni Manzini, S. Muthukrishnan, co-editors. *Special Issue on the Burrows-Wheeler Transform*. Theoretical Computer Science, 387(3), 2007.

[9] Juha Karkkainen, Dominik Kempa, and Simon J. Puglisi. Slashing the time for BWT inversion. *Proc. IEEE Data Compression Conference*, 99–108, 2012.

[10] Juha Karkkainen and Simon J. Puglisi. Medium-space algorithms for inverse BWT. *Proc. European Symposium on Algorithms*, LNCS 6346, 451–462, 2010.

[11] Juha Karkkainen and Simon J. Puglisi. Cache-friendly Burrows-Wheeler inversion. *Proc. International Conference on Data Compression, Communication and Processing*, 38–42, 2011.

[13] http://pizzachili.dcc.uchile.cl/

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

[13] Gonzalo Navarro and Veli Mäkinen. Compressed full-text indexes. *ACM Computing Surveys*, 39(1), 2007.

[14] Simon J. Puglisi, William F. Smyth, and Andrew Turpin. A taxonomy of suffix array construction algorithms. *Procs of the Prague Stringology Conference*, 1–30, 2005.

[15] Julian Seward. Space-time tradeoffs in the inverse B-W transform. In *Proc. IEEE Data Compression Conference*, 439–448, 2001.

[16] Julian Seward. The `bzip2` home page. `http://sources.redhat.com/bzip2`.

[17] Ian H. Witten, Alistair Moffat and Timothy C. Bell. Managing Gigabytes: Compressing and Indexing Documents and Images, Second Edition. Morgan Kaufmann, 1999.