Chapter 7

Compressed Full-Text Indexes

Until now, we have used the suffix array as an index data structure, and exact string matching was done in forward direction. This chapter is dedicated to index data structures and applications in which a forward search is replaced with a backward search. It is organized as follows. First, we review the Burrows-Wheeler transform [48]—a well-known technique employed in lossless data compression—on which backward search is based. Second, we describe the search algorithm discovered by Ferragina and Manzini [100]. Third, we introduce the wavelet tree invented by Grossi et al. [134]. This data structure supports backward search and has many other virtues. In subsequent sections, we shed new light on solutions to problems faced in Chapter 5, such as developing new algorithms that address space efficiency issues, as well as problems related to bidirectional searches and approximate string matching.

7.1 The components of a compressed full-text index

Many variations of *compressed suffix trees* (CSTs) have been proposed in the literature (see e.g. [273]), and these do not all have the same functionality. Because we focus on backward search, which is normally not supported by a CST, we prefer the term "compressed full-text index".

Definition 7.1.1 A *compressed full-text index* of a string S is a space-efficient data structure that supports (at least) the following operations:

1. backward search,
2. access to the suffix array of S,
3. access to the LCP-array of S,
4. navigation on the (virtual) suffix tree of S.

The compressed full-text index of the string S that is used throughout this book consists of the following four components:

1. the wavelet tree of the Burrows-Wheeler transformed string of S,
2. the sparse suffix array of S from Section 6.2.1,
3. the compressed LCP-array as explained in Section 6.2.2,
4. the balanced parentheses sequence BPS of the LCP-array that was introduced in Section 6.3.

We emphasize that each of the four components can be replaced with another component that has the same functionality. For example, the wavelet tree can be substituted by the compressed suffix array [135, 270] sketched in Section 6.2.1 because backward search can be done with the ψ-function; see Exercise 7.3.3. Further alternatives are described in [238]. However, the wavelet tree has many sophisticated properties that make it most suitable for many applications. Alternative compressed representations of the LCP-array are discussed in [126], among which is a representation that is based on the array LCP′ from Section 6.3.6. There are also alternatives to the BPS of the LCP-array, most notably the BPS_{pre} introduced in Section 6.1; cf. [231, 273]. We refer to [124] for an in-depth experimental study of the various incarnations of compressed full-text indexes.

7.2 The Burrows-Wheeler transform

The Burrows-Wheeler transform was introduced in a technical report written by David Wheeler and Michael Burrows [48]; see the historical notes in Adjeroh et al. [6]. In practice, the Burrows-Wheeler transformed string tends to be easier to compress than the original string; see e.g. [48, Section 3] and [215] for reasons why the transformed string compresses well.

Here we assume that the string S of length n is terminated by the sentinel character \$. Although this is not necessary for the Burrows-Wheeler transform to work correctly (cf. [48]), in virtually all practical cases the file to be compressed is terminated by a special symbol, the EOF (end of file) character. Moreover, it allows us to use a fast suffix sorting algorithm to compute the transformed string.

7.2.1 Encoding

The Burrows-Wheeler transform transforms a string S in three steps:

1. Form a conceptual matrix M' whose rows are the cyclic shifts of the string S.

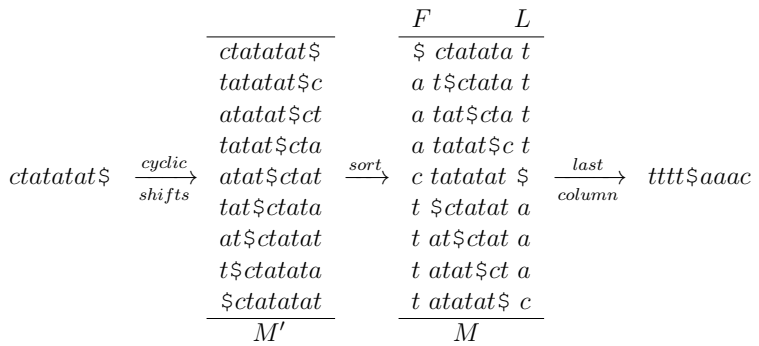

Figure 7.1: The Burrows and Wheeler transform applied to the string $S = ctatatat\$$ yields the output $L = tttt\$aaac$.

2. Compute the matrix M by sorting the rows of M' lexicographically.

3. Output the last column L of M.

An example can be found in Figure 7.1. We next show that computing the Burrows-Wheeler transformed string of S boils down to sorting the suffixes of S, or more precisely, the output L of the Burrows-Wheeler transform can be derived in linear time from the suffix array SA. To this end, we define a string BWT and show that it coincides with L.

Definition 7.2.1 For a string S of length n having the sentinel character at the end (and nowhere else), the string $\mathsf{BWT}[1..n]$ is defined by $\mathsf{BWT}[i] = \$$ if $\mathsf{SA}[i] = 1$ and $\mathsf{BWT}[i] = S[\mathsf{SA}[i] - 1]$ if $\mathsf{SA}[i] \neq 1$.

Obviously, the string $\mathsf{BWT}[1..n]$ can be derived in linear time from the suffix array SA; see Algorithm 7.1.

Algorithm 7.1 Computing BWT from SA and the string S.

```
for i ← 1 to n do
  if SA[i] = 1 then BWT[i] ← $
  else BWT[i] ← S[SA[i] − 1]
```

If we truncate each string in the matrix M after the sentinel $\$$, then the truncated strings are still lexicographically ordered; see Figure 7.2. Since these truncated strings are exactly the suffixes of S, the string BWT coincides with the string L (this crucially relies on the fact that S is terminated by $\$$; see Exercise 7.2.2).

F	L
$ ctatata	t
a t$ctata	t
a tat$cta	t
a tatat$c	t
c tatatat	$
t $ctatat	a
t at$ctat	a
t atat$ct	a
t atatat$	c

$\xrightarrow[after\ \$]{truncate}$

F	L
$	t
a t$	t
a tat$	t
a tatat$	t
c tatatat$	$
t $	a
t at$	a
t atat$	a
t atatat$	c

$\xrightarrow[L=\mathsf{BWT}]{observe}$

BWT	F	L
t	$	t
t	at$	t
t	atat$	t
t	atatat$	t
$	ctatatat$	$
a	t$	a
a	tat$	a
a	tatat$	a
c	tatatat$	c

Figure 7.2: Truncate the strings (rows) after the sentinel character, and observe that $L = \mathsf{BWT}$.

i	1	2	3	4	5	6	7	8	9
$L[i]$	t	t	t	t	$	a	a	a	c
$LF(i)$	6	7	8	9	1	2	3	4	5
$F[i]$	$	a	a	a	c	t	t	t	t

Figure 7.3: LF maps the last column L to the first column F.

Exercise 7.2.2 For a string S of length n without the sentinel character at the end, define $\mathsf{BWT}[i] = S[n]$ if $\mathsf{SA}[i] = 1$ and $\mathsf{BWT}[i] = S[\mathsf{SA}[i]-1]$ if $\mathsf{SA}[i] \neq 1$. Find a string S (without sentinel) for which $\mathsf{BWT} \neq L$.

7.2.2 Decoding

It is not obvious how the string BWT can be retransformed into the original string S. The key to this back-transformation is the so-called LF-mapping.

Definition 7.2.3 Let F and L be the first and last column in the matrix M; cf. Figure 7.1. The function $LF : \{1,\ldots,n\} \to \{1,\ldots,n\}$ is defined as follows: If $L[i] = c$ is the k-th occurrence of character c in L, then $LF(i) = j$ is the index so that $F[j]$ is the k-th occurrence of c in F.

The function LF is called last-to-first mapping because it maps the last column L to the first column F; see Figure 7.3 for an example. In the following, when we regard the LF- mapping as an array, we will the use the notation $LF[i]$ instead of $LF(i)$.

Algorithm 7.2 Computing LF from BWT and the C-array.

```
for all c ∈ Σ do
    count[c] ← C[c]
for i ← 1 to n do
    c ← BWT[i]
    count[c] ← count[c] + 1
    LF[i] ← count[c]
```

Next, we develop a linear-time algorithm that computes LF. To achieve this goal, we must be able to find the k-th occurrence of a character $c \in \Sigma$ in F. Employing the C-array (if we consider all characters in Σ that are smaller than c, then $C[c]$ is the overall number of their occurrences in S), the index of the first occurrence of character c in the array F is $C[c] + 1$. Therefore, the k-th occurrence of c in F can be found at index $C[c] + k$.

Algorithm 7.2 shows the pseudo-code for the computation of LF. It scans the BWT from left to right and counts how often each character appeared already. The algorithm uses an auxiliary array $count$ of size σ. Initially, $count[c] = C[c]$. Each time character c appears during the scan of BWT, $count[c]$ is incremented by one. As discussed above, if the algorithm finds the k-th occurrence of character c at index i in BWT, then the k-th occurrence of character c in F appears at index $count[c] = C[c]+k$. In other words, the index $LF[i]$ we are searching for is $count[c]$.

It remains to compute the original string S from BWT and LF. Lemma 7.2.4 states the crucial property of the LF-mapping that makes this possible.

Lemma 7.2.4 *The first row of the matrix M contains the suffix $S_n = \$$. If row i, $2 \leq i \leq n$, of the matrix M contains the suffix S_j, then row $LF(i)$ of M contains the suffix S_{j-1}.*

Proof Since $\$$ is the smallest character in Σ, the first row of M contains $\$$, which is the n-th suffix of S. Let $c \neq \$$ be a character in S, and let $i_1 < i_2 < \cdots < i_m$ be all the indices with $\mathsf{BWT}[i_k] = c$, $1 \leq k \leq m$. (So if we would number the m occurrences of c in $L = \mathsf{BWT}$ as $c_1, c_2, \ldots, c_m$, then $\mathsf{BWT}[i_k] = c_k$.) Because the suffixes in M are ordered lexicographically, we have $S_{\mathsf{SA}[i_1]} < S_{\mathsf{SA}[i_2]} < \cdots < S_{\mathsf{SA}[i_m]}$. Obviously, this implies $cS_{\mathsf{SA}[i_1]} < cS_{\mathsf{SA}[i_2]} < \cdots < cS_{\mathsf{SA}[i_m]}$. (With the occurrence numbers as subscripts, $c_1S_{\mathsf{SA}[i_1]} < c_2S_{\mathsf{SA}[i_2]} < \cdots < c_mS_{\mathsf{SA}[i_m]}$.) By definition, $LF(i_k)$ is the index so that $F[LF(i_k)]$ is the k-th occurrence of c in F. Since $cS_{\mathsf{SA}[i_k]} = S_{\mathsf{SA}[i_k]-1}$, it follows that row $LF(i_k)$ of M contains the suffix $S_{\mathsf{SA}[i_k]-1}$. □

Theorem 7.2.5 *If $L = \mathsf{BWT}$ is the output of the Burrows-Wheeler transform applied to the string S, and LF is the corresponding last-to-first mapping, then Algorithm 7.3 computes S.*

Algorithm 7.3 Computing the string S from BWT and LF.

```
S[n] ← $
j ← 1
for i ← n − 1 downto 1 do
  S[i] ← BWT[j]
  j ← LF(j)
```

Proof Initially, the algorithm assigns $ to $S[n]$. This is correct because $ is the last character of S. Since $ is the smallest character in Σ, row $j = 1$ of the matrix M contains the suffix $S_n = \$$. Now $L[1] = \mathsf{BWT}[1] = S[n-1]$ implies that the $(n-1)$-th character of S is correctly decoded in the first iteration of the for-loop. After the assignment $j \leftarrow LF(j)$, row j contains the suffix S_{n-1}. In the second iteration of the for-loop, the $(n-2)$-th character of S is correctly decoded because $L[j] = \mathsf{BWT}[j] = S[n-2]$, and so on. □

Exercise 7.2.6 Extend Algorithm 7.3 so that it also computes the suffix array of the string S. Is it possible to overwrite the LF-array with the suffix array? (This would save space if the LF-array is no longer needed.)

An alternative way to retransform the BWT into the original string S uses the ψ-function instead of the LF-mapping. We are already familiar with the ψ-function: For a string of length n (without the sentinel character $ at the end), $\psi(i) = \mathsf{ISA}[\mathsf{SA}[i]+1]$ for all i with $\mathsf{SA}[i] < n$; see Definition 5.5.4. Here, we assume that the string under consideration is terminated by $. If S is a string of length n having the sentinel character at the end (and nowhere else), then $\mathsf{SA}[1] = n$ because $ is the lexicographically smallest suffix of S. So with the previous definition of the ψ-function, the value $\psi(1)$ is undefined. Definition 7.2.7 provides a value for $\psi(1)$ so that ψ becomes a permutation.

Definition 7.2.7 The function $\psi : \{1,\ldots,n\} \to \{1,\ldots,n\}$ is defined by $\psi(i) = \mathsf{ISA}[\mathsf{SA}[i]+1]$ for all i with $2 \le i \le n$ and $\psi(1) = \mathsf{ISA}[1]$.

The next two lemmata reveal the close relationship between the functions LF and ψ.

Lemma 7.2.8 *We have* $LF(i) = \mathsf{ISA}[\mathsf{SA}[i]-1]$ *for all* i *with* $\mathsf{SA}[i] \neq 1$ *and* $LF(i) = 1$ *for the index* i *so that* $\mathsf{SA}[i] = 1$.

Proof If $\mathsf{SA}[i] = 1$, then $\mathsf{BWT}[i] = \$$. Since $ occurs at index 1 in the array F, we have $LF(i) = 1$. Now suppose that $\mathsf{SA}[i] \neq 1$. According to Lemma 7.2.4, if $\mathsf{SA}[i] = j$, then $\mathsf{SA}[LF(i)] = j - 1$. So the equation $\mathsf{SA}[LF(i)] = \mathsf{SA}[i] - 1$ holds true. Thus, $LF(i) = \mathsf{ISA}[\mathsf{SA}[i]-1]$. □

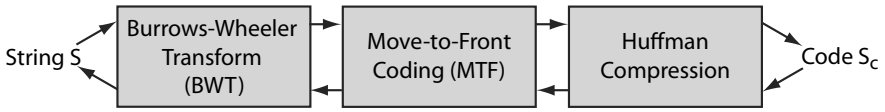

Figure 7.4: The main phases of the bzip2 compression program.

Lemma 7.2.9 *The functions LF and ψ are inverse of each other.*

Proof We will show $LF(\psi(i)) = i$ for all i with $1 \leq i \leq n$. (The equality $\psi(LF(i)) = i$ similarly follows.) If $i = 1$, then $\psi(1) = \mathsf{ISA}[1]$ is the index so that $\mathsf{SA}[\mathsf{ISA}[1]] = 1$. Hence $LF(\psi(1)) = 1$ by Lemma 7.2.8. For $i > 1$, it follows from Lemma 7.2.8 and Definition 7.2.7 that $LF(\psi(i)) = \mathsf{ISA}[\mathsf{SA}[\psi(i)] - 1] = \mathsf{ISA}[\underbrace{\mathsf{SA}[\mathsf{ISA}}_{cancel}[\mathsf{SA}[i] + 1]] - 1] = \mathsf{ISA}[\mathsf{SA}[i] + 1 - 1] = i$. □

Exercise 7.2.10 This exercise makes clear that LF can be replaced with ψ in BWT-decoding.

- Modify Algorithm 7.2 so that it computes the ψ-array from BWT. You may assume that the index index_of_$, at which the character $ occurs in the string BWT, is known (it can easily be computed during the Burrows-Wheeler transform).
- Modify Algorithm 7.3 so that it computes the string S from BWT, index_of_$, and ψ.
- Show how to compute the suffix array SA from BWT, index_of_$, and ψ. Is it possible to overwrite the ψ-array with the suffix array? (This would save space if the ψ-array is no longer needed.)

7.2.3 Data compression

The Burrows-Wheeler transform is used in many lossless data compression programs, of which the best known is Julian Seward's bzip2. Figure 7.4 shows bzip2's main phases. (Its ancestor bzip used arithmetic coding [267] instead of Huffman coding [158]. The change was made because of a software patent restriction.) It is possible to further use a run-length encoder (RLE) in between move-to-front (MTF) and Huffman coding, or to replace MTF with RLE. As a matter of fact, many more variations of the coding scheme are possible. The reader is referred to Adjeroh et al. [6] for a detailed introduction to the current state of knowledge about data compression with the Burrows-Wheeler transform.

An application of the coding scheme from Figure 7.4 to the string $S = ctatatat\$$ yields the code $S_c = 0111100010111$. The intermediate steps are

$$S = ctatatat\$ \overset{\texttt{BWT}}{\Longrightarrow} L = tttt\$aaac \overset{\texttt{MTF}}{\Longrightarrow} R = 300012003 \overset{\texttt{Huffman}}{\Longrightarrow} S_c = 011110000011101$$

We have already seen how the Burrows-Wheeler transform works, so we now turn to the other two steps: move-to-front and Huffman coding.

Move-to-front coding

Bentley et al. [36] introduced the *move-to-front* transform in 1986 but the method was already described in 1980 by Ryabko; see [269]. The MFT is an encoding of a string designed to improve the performance of entropy encoding techniques of compression like Huffman coding [158] and arithmetic coding [267]. The idea is that each character in the string is replaced by its rank in a list of recently used characters. After a replacement, the character is moved to the front of the list of characters. Algorithm 7.4 makes this precise.

Algorithm 7.4 Move-to-front coding of a string $L \in \Sigma^n$.

Initialize a $list$ containing the characters from Σ in increasing order.
for $i \leftarrow 1$ **to** n **do**
  $R[i] \leftarrow$ number of characters preceding character $L[i]$ in $list$
  move character $L[i]$ to the front of $list$

Figure 7.5 shows the application of Algorithm 7.4 to the string $L = tttt\$aaac$; note that $'0'$ occurs more often in the resulting string R than t or a do in L. As you can see, every *run* (a run is a substring of identical characters) is replaced by a sequence of zeros (except for the first rank). Because a Burrows-Wheeler transformed string usually has many runs, the proportion of zero ranks after MTF has been applied is relatively high.

Pseudo-code for the decoding of the rank vector R is shown in Algorithm 7.5, and Figure 7.6 illustrates the behavior of this algorithm applied to the rank vector $R = 300012003$.

Algorithm 7.5 Move-to-front decoding of R.

Initialize a $list$ containing the characters from Σ in increasing order.
for $i \leftarrow 1$ **to** n **do**
  $L[i] \leftarrow$ character at position $R[i] + 1$ in $list$ (numbering elements from 1)
  move character $L[i]$ to the front of $list$

i	$list$	$L[i]$	$R[i]$
1	$\$act$	t	3
2	$t\$ac$	t	0
3	$t\$ac$	t	0
4	$t\$ac$	t	0
5	$t\$ac$	$\$$	1
6	$\$tac$	a	2
7	$a\$tc$	a	0
8	$a\$tc$	a	0
9	$a\$tc$	c	3

Figure 7.5: Move-to-front coding of a string $L = tttt\$aaac$.

i	$list$	$R[i]$	$L[i]$
1	$\$act$	3	t
2	$t\$ac$	0	t
3	$t\$ac$	0	t
4	$t\$ac$	0	t
5	$t\$ac$	1	$\$$
6	$\$tac$	2	a
7	$a\$tc$	0	a
8	$a\$tc$	0	a
9	$a\$tc$	3	c

Figure 7.6: Move-to-front decoding of $R = 300012003$ with $\Sigma = \{\$, a, c, t\}$.