FM-index: a Memory-Efficient Full-Text Index using the Burrows-Wheeler Transform

Pavel Veselý

December 2025

In this note, we describe a data structure, called the FM-index, for searching a pattern in a text that is more efficient than the suffix array. In particular, while the suffix array (together with the rank and LCP arrays) requires $O(n)$ *memory words*, the FM-index needs just $O(n)$ *bits*, assuming a constant-size alphabet. Still, the FM-index is able to count the number of occurrences of a pattern of length k in $O(k)$ time, without the need for binary search as in the suffix array. We present a simplified version of the FM-index, without the possibility to get the starting positions of the pattern occurrences; see Section 4 for the missing bits. The FM-index was developed by Paolo Ferragina and Giovanni Manzini in [2].

We use the same notation in the lecture note "Strings" (describing the suffix array) by Martin Mareš.

1 The Burrows-Wheeler transform

Burrows and Wheeler [1] proposed the following permutation of a string α. To make it easily invertible, we append a new character \$ to α such that \$ does not appear in α and is lexicographically smaller than any character of α. For convenience, we use n for the length of $\alpha\$$.

Definition 1. *Given a string α of length $n-1$ over alphabet Σ, consider all the rotations of $\alpha\$$ sorted lexicographically, where character $\$ \notin \Sigma$ is lexicographically smaller than all characters in Σ. Define the* Burrows-Wheeler (BW) matrix *as the $n \times n$ table where the i-th row is the i-th rotation of $\alpha\$$ in the lexicographic order. The* Burrows-Wheeler transform *of α, denoted* $\mathrm{BWT}(\alpha)$*, is the last column of the BW matrix.*

Let us see a small example of the BW matrix first, for string `banana`:

```
$ b a n a n a
a $ b a n a n
a n a $ b a n
a n a n a $ b
b a n a n a $
n a $ b a n a
n a n a $ b a
```

Therefore, BWT(`banana`) = `annb$aa`. Lecture notes *Strings* contain the suffix array for `annbansbananas`, for which the BWT is `sbn$bnsnaanaaan`:

$	a	n	n	b	a	n	s	b	a	n	a	n	a	**s**
a	n	a	n	a	s	$	a	n	n	b	a	n	s	**b**
a	n	a	s	$	a	n	n	b	a	n	s	b	a	**n**
a	n	n	b	a	n	s	b	a	n	a	n	a	s	**$**
a	n	s	b	a	n	a	n	a	s	$	a	n	n	**b**
a	s	$	a	n	n	b	a	n	s	b	a	n	a	**n**
b	a	n	a	n	a	s	$	a	n	n	b	a	n	**s**
b	a	n	s	b	a	n	a	n	a	s	$	a	n	**n**
n	a	n	a	s	$	a	n	n	b	a	n	s	b	**a**
n	a	s	$	a	n	n	b	a	n	s	b	a	n	**a**
n	b	a	n	s	b	a	n	a	n	a	s	$	a	**n**
n	n	b	a	n	s	b	a	n	a	n	a	s	$	**a**
n	s	b	a	n	a	n	a	s	$	a	n	n	b	**a**
s	$	a	n	n	b	a	n	s	b	a	n	a	n	**a**
s	b	a	n	a	n	a	s	$	a	n	n	b	a	**n**

A few simple observations:

- The first row of the BW matrix is $\$\alpha$. Thus, the first character of $\mathrm{BWT}(\alpha)$ is the last character of α.
- The first column starts with \$, followed by the sorted characters of α. Thus, the first column can be obtained from the BWT by counting the number c_x of occurrences of each character x.
- By trimming every row just before \$, we obtain all the suffixes of α sorted lexicographically. It is thus possible to get the BWT from the suffix array X, namely by taking $\mathrm{BWT}(\alpha)[i] = (\alpha\$)[X[i] - 1 \mod n]$.

  Vice versa, we may compute the suffix array from the BW matrix or even from the BWT; see Exercise 2.

The following observation is key to inverting the BWT or searching; it is called the *LF mapping*, where "L" stands for the last column of the BW matrix and "F" stands the first column.

Observation 1 (The LF mapping)**.** *For any character x of α and any $i \leq n$, the i-th occurrence of x in $\mathrm{BWT}(\alpha)$ and the i-th occurrence of x in the first column of the BW matrix correspond to the same occurrence of x in α.*

Proof. Let j be the index in α such that $\alpha[j]$ is the i-th occurrence of x in $\mathrm{BWT}(\alpha)$. Define the right context of any occurrence $\alpha[j']$ of x as $\alpha[j'+1:]\alpha[:j']$. The observation follows by noticing that the occurrences of x are ordered by their right context in both $\mathrm{BWT}(\alpha)$ and the first column of the BW matrix. □

Observation 1 defines the *LF mapping* in the following way: For any $0 \leq \ell < n$, the position ℓ in $\mathrm{BWT}(\alpha)$, with the i-th occurrence of x, is mapped to position j in the first column where $j = \sum_{y<x} c_y + i$; here, the sum is over all characters y smaller than x, including \$, and c_y is the number of occurrences of y in α. To simplify notation, let $C_{<x} := \sum_{y<x} c_y$ be the prefix sum.

Remark 1. *Instead of appending \$ to α, one can just store the index of the row that contains α in the Burrows-Wheeler matrix computed from sorted rotations of α.*

Remark 2. *The BWT makes the string potentially much more compressible, especially if there are repetitions of the same substring, which makes it suitable for compression of certain types of data. The `bzip2` software tool is based on the BWT.*

2 Pattern Searching in the BWT

To turn the BWT into a data structure for pattern searching, we just implement the LF mapping. To this end, we will need the *occurrence function* $\mathrm{occ}(x, i)$ that for any index i and character x, gives the number of occurrences of x in $\mathrm{BWT}(\alpha)[:i]$, i.e., in the first i characters of the BWT. Before we discuss a space-efficient data structure for answering $\mathrm{occ}(x, i)$ in constant time, we show how search can be performed on the BWT. We note that our simplified search algorithm will only determine how many times a given pattern appears in α as a substring; see Section 4 for a brief description of recovering its starting position(s) in α.

Given a pattern γ of length k (not containing \$), we perform the search *backwards*, maintaining the range of BW matrix rows that contain the current suffix as a prefix. Specifically, in the ℓ-th step, for $\ell = 1, \ldots, k$, we obtain a maximal interval $[i_\ell, j_\ell)$ such that for any $m \in [i_\ell, j_\ell)$, the m-th row of the BW matrix starts with $\gamma[k - \ell :]$. Once $i_\ell = j_\ell$, i.e., the interval gets empty, so $\gamma[k - \ell :]$ is not present in α, and we can stop the search. More generally, $j_\ell - i_\ell$ is the number of occurrences of $\gamma[k - \ell :]$ in α. Thus, for $\ell = k$, we obtain the number of occurrences of γ.

The first interval $[i_1, j_1)$ is simply defined by $i_1 = C_{<\gamma[k-1]}$ and $j_1 = i_1 + c_{\gamma[k-1]}$. For $\ell > 1$, to compute $[i_\ell, j_\ell)$ from $[i_{\ell-1}, j_{\ell-1})$, we use the occurrence function. Namely, we compute

$$i_\ell := C_{<\gamma[k-\ell]} + \mathrm{occ}(\gamma[k - \ell], i_{\ell-1})$$
$$j_\ell := C_{<\gamma[k-\ell]} + \mathrm{occ}(\gamma[k - \ell], j_{\ell-1})$$

Note that $\mathrm{occ}(\gamma[k - \ell], j_{\ell-1}) - \mathrm{occ}(\gamma[k - \ell], i_{\ell-1})$ is the number of occurrences of $\gamma[k - \ell]$ in $\mathrm{BWT}(\alpha)[i_{\ell-1} : j_{\ell-1}]$.

Algorithm 1 BackwardSearch

Input: BWT of $\alpha\$$; array C with prefix sums of letter occurrences; the occurrence function $\mathrm{occ}(x, i)$; pattern $\gamma = \gamma_1\gamma_2 \cdots \gamma_k$
Output: Interval $[i, j)$ of all rows of the BW matrix for $\alpha\$$ with γ as prefix

```
1:  i ← C_{<γ[k−1]}
2:  j ← i + c_{γ[k−1]}
3:  for ℓ = 2, ..., k do
4:      x ← γ[k − ℓ]
5:      i ← C_{<x} + occ(x, i)
6:      j ← C_{<x} + occ(x, j)
7:      if i ≥ j then
8:          return [i, j)                    ▷ empty interval = not found
9:      end if
10: end for
11:     return [i, j)                        ▷ the number of matches is j − i
```

3 Implementing the Occurrence Function

It remains to implement the occurrence function, which we do separately for each character; if the alphabet is larger, it is better to use a binary tree over the alphabet. We fix a character x and describe a data structure for evaluating $\mathrm{occ}(x, i)$ for any length-n string β in constant time.

First, we compute a characteristic bitvector v of x's occurrences in β, i.e., $v_i = 1$ iff $\beta[i] = x$; then $\mathrm{occ}(x, i)$ is typically referred to as the rank function for v. Let $L = \lceil \log_2^2 n \rceil$. We split v into blocks of length L. At each block end i, we store the value of $\mathrm{occ}(x, i)$, which takes at most $O(\log n)$ bits; these values thus take $O(n/L \cdot \log n) = O(n/\log n)$ bits. Further, we split each block into sub-blocks of length $\ell = \Theta(\sqrt{\log n})$, where the hidden constant is chosen so that ℓ divides L. For each sub-block end i' that is inside a block starting at i, we store the value of

$\mathrm{occ}(x, i') - \mathrm{occ}(x, i)$. Since these values are bounded by L, they only take $O(\log \log n)$ bits each, and overall, the sub-block counts take only $O(n/\ell \cdot \log \log n) = o(n)$ bits. Finally, note that there are only $2^\ell = o(n)$ different length-ℓ bitvectors, i.e., different sub-blocks. We thus precompute an array indexed by length-ℓ bitvectors containing the number of set bits for each of them; storing this array takes $O(2^\ell \cdot \log_2 \ell)$ bits of space. Overall, the rank data structure for evaluating $\mathrm{occ}(x, i)$ for a fixed x takes $o(n)$ bits of space.

To evaluate $\mathrm{occ}(x, i)$, we compute its block and sub-block, sum the stored values for the ends of the previous block and previous sub-block (if any). We extract the bits of v within the sub-block, zero all bits starting with the i-th, and lookup the number of set bits in T. All of the operations performed are just lookups in arrays or bit operations, yielding evaluation in $O(1)$ time.

illustration

4 Summary and Improvements

Retrieving the original coordinates of the occurrence. Our presentation above is a simplified FM-index, and the original algorithm from [2] allows to compute the original coordinates for each occurrence of the given pattern. This is done by including a *sampled* suffix array X. That is, we choose a parameter s and store only n/s entries of X, basically every s-th entry with respect to the original positions in the string α. Suppose that we now want to retrieve the original coordinates of an occurrence, which has coordinate i in the suffix array. If $X[i]$ is stored, we are done. Otherwise, we perform steps towards the beginning of the string with respect to the original coordinates, using the LF mapping, and we only need to perform at most s steps before we find a sampled position of X. If we perform t steps and end at position $X[i']$ of the original string, then we return $X[i'] + t$ as the answer. See Section 3.2 in [2] for more details.

Further improvements. [TBWL (to be written later)]

5 Exercises

Exercise 1. *Show that BWT can be inverted in linear time, i.e., one can reconstruct the original string from its BWT image.*

Exercise 2. *Show that the suffix array can be constructed from the BWT in linear time.*

References

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

[2] P. Ferragina and G. Manzini. Opportunistic data structures with applications. In *FOCS '00*, SFCS-00. IEEE, 2000.