Computing the Burrows–Wheeler transform in place and in small space

Maxime Crochemore, Roberto Grossi, Juha Kärkkäinen, Gad M. Landau

Computing the Burrows–Wheeler transform in place and in small space ☆

Maxime Crochemore [a], Roberto Grossi [b,*], Juha Kärkkäinen [c], Gad M. Landau [d,e]

[a] King's College London, UK
[b] Dipartimento di Informatica, Università di Pisa, Italy
[c] Department of Computer Science, University of Helsinki, Finland
[d] Department of Computer Science, University of Haifa, Israel
[e] Department of Computer Science and Engineering, NYU-Poly, Brooklyn NY, USA

A B S T R A C T

We introduce the problem of computing the Burrows–Wheeler Transform (BWT) using small additional space. Our in-place algorithm does not need the explicit storage for the suffix sort array and the output array, as typically required in previous work. It relies on the combinatorial properties of the BWT, and runs in $O(n^2)$ time in the comparison model using $O(1)$ extra memory cells, apart from the array of n cells storing the n characters of the input text. We then discuss the time–space trade-off when $O(k \cdot \sigma_k)$ extra memory cells are allowed with σ_k distinct characters, providing an $O((n^2/k + n)\log k)$-time algorithm to obtain (and invert) the BWT. For example in real systems where the alphabet size is a constant, for any arbitrarily small $\epsilon > 0$, the BWT of a text of n bytes can be computed in $O(n\epsilon^{-1}\log n)$ time using just ϵn extra bytes.

1. Introduction

The Burrows–Wheeler Transform [4] (known as BWT) of a text string is at the heart of the `bzip2` family of text compressors, and finds also applications in text indexing and sequence processing. Consider an input text string $T \equiv T[0\,..\,n-1]$ and the set of its suffixes $T_i \equiv T[i\,..\,n-1]$ $(0 \le i < n)$ under the lexicographic order, where $T[n-1]$ is an endmarker character \$ smaller than any other character in T. The alphabet Σ from which the characters in T are drawn can be unbounded.

A classical way to define the BWT uses the n circular shifts of the text $T = \texttt{mississippi\$}$ as shown in the first column of Table 1. We perform a lexicographic sort of these shifts, as shown in the second column: if we mark the last character from each of the circular shifts in this order, we obtain a sequence L of n characters that is called the BWT of T. Its relation

☆ A preliminary version of the results in this paper appeared in [6]. The work of the second author has been supported in part by the Italian Ministry of Education, University, and Research (MIUR) under PRIN 2012C4E3KT project. The work of the third author has been supported by the Academy of Finland Grant 118653 (ALGODAN). The work of the fourth author has been partially supported by the National Science Foundation Award 0904246, Israel Science Foundation Grants 347/09 and 571/14, Yahoo, Grant No. 2008217 from the United States–Israel Binational Science Foundation (BSF) and DFG.

* Corresponding author.
E-mail addresses: Maxime.Crochemore@kcl.ac.uk (M. Crochemore), grossi@di.unipi.it (R. Grossi), Juha.Karkkainen@cs.helsinki.fi (J. Kärkkäinen), landau@cs.haifa.ac.il (G.M. Landau).

Table 1
BWT L for the text $T = \texttt{mississippi\$}$ and its relation with suffix sort.

Cyclic shifts	Sorted cyclic shifts		Suffixes	
		L	i	T_i
mississippi$	$mississipp	i	11	$
$mississippi	i$mississip	p	10	i$
i$mississipp	ippi$missis	s	7	ippi$
pi$mississip	issippi$mis	s	4	issippi$
ppi$mississi	ississippi$	m	1	ississippi$
ippi$mississ	mississippi	$	0	mississippi$
sippi$missis	pi$mississi	p	9	pi$
ssippi$missi	ppi$mississ	i	8	ppi$
issippi$miss	sippi$missi	s	6	sippi$
sissippi$mis	sissippi$mi	s	3	sissippi$
ssissippi$mi	ssippi$miss	i	5	ssippi$
ississippi$m	ssissippi$m	i	2	ssissippi$

to suffix sort is well known, as illustrated in the third column: the rth character in L is $T[j-1]$ if and only if T_j is the rth suffix in the sort (except the borderline case $j = 0$, for which we take $T[n-1]$ as character).

As it can be seen in the example of Table 1, the BWT produces a text of the same length as the input text T. The transform is reversible since it is a one-to-one function when the input text is terminated by an endmarker \$. Thus, not only we can recover T from L alone, but typically L is more compressible than T itself using 0th-order compressors [21]. There are now efficient methods that convert T to L and vice versa, taking $O(n \log n)$ time for unbounded alphabets in the worst case [1].[1] The BWT is also a key element of some compressed text indexing implementations due to the small amount of space it requires: some examples are the solution by Ferragina and Manzini [8] or that by Grossi et al. [10], where the transform is associated with the techniques of wavelet trees and of succinct data structures using rank-select queries on binary sequences [22].

One of the prominent applications of the BWT is for software dealing with Next Generation Sequencing, where millions of short strings, called reads, are mapped onto a reference genome. Typical and popular software of this type are Bowtie [19], BWA [18] and SOAP2 [16]. Here it is crucial that the genome is indexed in a compact manner to get reasonable running time. Space issues for computing the BWT are thus relevant: frequently the input data is so large that the input text T stays in main memory while any additional data structure of similar size cannot fit in the rest of the main memory [14].

All the previous work for computing the BWT of T relies on the fact that (a) we need first to *store* the suffix sorting of T (also known as suffix array [20]), thus occupying n memory cells for storing integers, and (b) we need to output the BWT in *another* array storing n characters. Motivated by these observations, we want to study the case in which (a) and (b) are avoided, thus saving on the space occupied by them.

In this paper, our goal is to obtain the BWT by directly permuting T and using just $O(1)$ memory cells, i.e., we aim at an *in-place* algorithm for computing the BWT. We consider the model in which the text T is stored as an array of n entries, where each entry stores exactly one character of T. Note that storing an integer usually takes more space than a character, so we assume that only the characters of T can be kept in the array T. Moreover, T is not read-only but it can be modified at any time, and just $O(1)$ additional memory cells (besides T) can be kept for storing auxiliary information.[2]

Note that our model represents some realistic situations in which one has to handle large text collections, or large genomic sequences, without relying on extra memory for (a) and (b). Hence it is crucial to maximize the amount of data that can fit into main memory: not storing explicitly (a) and (b) permits to save space, which is typically regarded as taking more than half of the total space required. For instance, DNA sequences are stored by using 2 bits per character and machine integers take 64 bits. Here we just need $2n$ bits to store the (genomic) text and save the $64n$ bits used for storing the intermediate suffix sorting in (a) and the $2n$ bits for storing the output of BWT in another array in (b): this means that during the BWT construction, we can fit almost 33 times more text using the same main memory size, thus eliminating the usage of the slower external memory for this time-consuming task in these cases.

From the combinatorial point of view, the in-place BWT is an interesting question to solve on strings. There are space saving approaches storing the suffix sorting in compressed form [13,24,14,27] or only partially at a time [15], but none of them provides an in-place algorithm. In-place selection and sorting does not seem to help either [7,9,12,23,28]. It is well known that in-place sorting requires the same comparison cost of $\Theta(n \log n)$ as in standard sorting. But for the BWT, we only know its comparison cost of $\Theta(n \log n)$ for the standard construction. As far as we know, no result is known for the in-place construction of BWT: a naive solution is not that simple, even if it results in exponential time. Indeed, any

[1] As is standard in many string algorithms, we assume that any two characters in Σ can only be compared and this takes $O(1)$ time. Hence, comparing characterwise any two suffixes may require $O(n)$ time in the worst case.

[2] In C code, we would declare T as `unsigned char T[n]` and use this storage plus $O(1)$ local variables of constant size. A more formal model would say that each memory cells hosts a character from Σ and so an integer of $\log n$ bits requires $\log_{|\Sigma|} n$ cells. We prefer to keep it simpler and say that an extra cell can contain an integer.

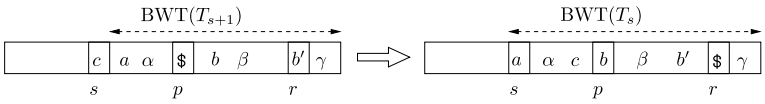

Fig. 1. An illustration of Steps 1–4 of the in-place construction of the BWT.

movement of a character $T[j]$ to another position inside T at least changes the content of its suffixes T_i for $0 \le i \le j$, making the algorithmic flavor of this problem different from that of in-place sorting n elements.

The above discussion suggests that a careful orchestration of the movement of the characters inside T is needed to avoid losing the content of some suffixes before they contribute to the BWT. Our idea is to define a sequence of transformations $B_0, B_1, \ldots, B_n$, where B_0 is the input text T and B_n is the final BWT of T. For $1 \le \ell \le n$, we have that B_ℓ is the BWT of the last ℓ characters in T and is computed from $B_{\ell-1}$ (re)using just $O(1)$ extra memory cells. We think that this sequence of transformations could be of independent interest for the community of string algorithms, and some of the combinatorial properties that we use can be found in [3,14,17,29].

In this paper we propose an $O(n^2)$-time approach that builds the above sequence of transformations using four integer variables and one character variable, taking $O(n)$ time per transformation in the worst case. The resulting in-place algorithm is simple and can be easily encoded in few lines of C code or similar programming languages. However we do not claim any practicality of our solution due to its quadratic worst-case cost. Our contribution is that it could lay out the path towards faster methods for the space-efficient computation of the BWT: any method to compute B_k from B_{k-1} in $t(n)$ time (re)using $s(n)$ space, would lead to a construction of the BWT in $O(n \cdot t(n))$ time using $O(1 + s(n))$ space. To this end, it is worth noting that the inputs for BWT are typically large and a fast algorithm that is in-place or uses very low additional memory, would be relevant in practice.

Our theoretical study has also an impact on the practical algorithm design. A natural question is what we get if we allow for some extra space. We prove that using $O(k \cdot \sigma_k)$ additional space for any given parameter $k \le n$, where $\sigma_k \le \min\{|\Sigma|, k\}$ is the maximum number of distinct characters found among k consecutive positions in T, we can compute the BWT (and its inverse) of a text of n characters in $O((n^2/k + n)\log k)$ time in the comparison model. We observe that σ_k is practically a constant in many applications. The practical implications of this trade-off in space versus time can be appreciated by observing that, for any arbitrarily small $\epsilon > 0$, we can obtain the BWT of a text of n bytes in $O(n\epsilon^{-1}\log n)$ time using just ϵn extra bytes for a constant-size alphabet. This is useful when the text occupies a great part of the available main memory, and only ϵn free cells are available. This avoids using external-memory algorithms, which are clearly slower as I/O access takes several orders of magnitudes more time than main memory access.

The paper is organized as follows. We describe how to perform the in-place BWT in Section 2. We then discuss how to invert the BWT, so as to obtain the original text T, in Section 3. We illustrate the trade-off between space and time in Section 4. Finally, we draw some conclusions in Section 5.

2. In-place BWT

Given the input text $T = T[0\,..\,n-1]$ where $T[n-1] = \$$, moving a single character inside T can change the content of many suffixes. The idea to circumvent this difficulty without using storage for the suffix sort is to proceed by induction from right to left in T, while maintaining the BWT of the current suffix T_s, denoted by BWT(T_s). We assume $0 \le s \le n-3$, since the last two suffixes of T are equal to their respective BWT.

To compute BWT(T_s), suppose that BWT(T_{s+1}) has been already computed and stored in the last positions of T, i.e. $T[s+1\,..\,n-1]$. Consider the current character $c = T[s]$: if we look at the content of $T[s\,..\,n-1]$, we no longer find T_s, but the character c followed by the permutation BWT(T_{s+1}) of T_{s+1}. Nevertheless, we still have enough information as we will show in the proof of Theorem 1 that the position of \$ inside BWT(T_{s+1}) is related to the rank of T_{s+1} among the suffixes $T_{s+1}, \ldots, T_{n-1}$ in lexicographic order. We exploit this fact in the following steps (see Fig. 1).

1. Find the position p of the \$ in $T[s+1\,..\,n-1]$: note that $p - s$ is the (local) rank of the suffix T_{s+1} that originally was starting at position $s+1$.
2. Find the rank r of the suffix T_s (originally in position s). Using character c, scan $T[s+1\,..\,n-1]$ and compute the sum of two counts: how many characters are strictly smaller than c, and how many occurrences of c appear in $T[s+1\,..\,p]$ (and add s as an offset to obtain r).
3. Store c into $T[p]$ (thus replacing the \$).
4. Insert the character \$ in $T[r]$ by shifting $T[s+1\,..\,r]$ by one position to the left, so as to occupy positions $s, \ldots, r-1$ of T.

The C code reported in Fig. 2 implements Steps 1–4, where `END_MARKER` denotes \$. For example, consider $T =$ `mississippi`\$ and $s = 4$, where we use capital letters to denote the BWT partially built on the last positions of T. Suppose that we have already computed the BWT for the last 7 characters in T, namely, we have `missiIPSPIS`\$. We then have $p = 11$ and, since there is one character (\$) smaller than $c =$ `i`, and two characters that are equal to c and occur

```
void inplaceBWT( unsigned char T[ ], int n ){
 int i, p, r, s;
 unsigned char c;

 for ( s = n-3;  s >= 0; s-- ){
   c = T[ s ];

   /* steps 1 and 2 */
   r = s;
   for ( i = s+1; T[ i ] != END_MARKER; i++ )
     if ( T[ i ] <= c ) r++;
   p = i;
   while ( i < n )
     if ( T[ i++ ] < c ) r++;

   /* step 3 */
   T[ p ] = c;

   /* step 4 */
   for ( i = s; i < r; i++ )
     T[ i ] = T[ i+1 ];
   T[ r ] = END_MARKER;
 }
}
```

Fig. 2. In-place construction of BWT.

before position p, we have $r = s + 3 = 7$. This means that we have to replace \$ by c and shift `IPS` by one position left so as to insert \$ in position r. The next configuration is `missIPS$PISI`, which ends with the BWT of T_s.

Theorem 1. *Given a text T of n characters, we can compute its Burrows–Wheeler Transform (*BWT*) in $O(n^2)$ time in the comparison model using $O(1)$ additional memory cells.*

Proof. We prove first the correctness. Let T be the input text and T' be its modification at a generic iteration s, where $0 \le s \le n-3$. Note that $T'[0\,..\,s] = T[0\,..\,s]$ while $T'[s+1\,..\,n-1] = \mathrm{BWT}(T[s+1\,..\,n-1])$. By induction, the position p of \$ in $T'[s+1\,..\,n-1]$ indicates the rank $p-s$ of T_{s+1} among the suffixes in $\{T_{s+1}, T_{s+2}, \ldots, T_{n-1}\}$ in lexicographic order. The base case for T_{n-2} and T_{n-1} is trivially satisfied. Hence, we show how to preserve this property for $0 \le s \le n-3$.

First note that the character $c = T[s]$ goes in position p, since it precedes T_{s+1} inside T. Next, we have to find the new position r for T_s, so that $r-(s-1)$ is its rank among the suffixes in $S = \{T_s, T_{s+1}, \ldots, T_{n-1}\}$ in lexicographic order. First count how many characters smaller than c occur in $T'[p\,..\,n-1]$: there are as many suffixes in S that are smaller than T_s since their first character is smaller than c. To this quantity, add the number of occurrences of c in $T'[s+1\,..\,p-1]$: these suffixes are also smaller since they start with c but have rank smaller than p, i.e. the rank of T_{s+1}. In this way, we discover how many suffixes are smaller than T_s in S: inserting \$ in the corresponding location r of T', by shifting the characters in $T'[s+1\,..\,r]$ to the left, which thus occupy positions $s, \ldots, r-1$ (see Fig. 1), we maintain the induction. Hence, $T'[0\,..\,s-1] = T[0\,..\,s-1]$ and $T'[s\,..\,n-1] = \mathrm{BWT}(T[s\,..\,n-1])$. When $s = 0$, we obtain the BWT of T.

As for the complexity, note that each of the $n-2$ iterations requires $O(n)$ time, since it can be implemented by $O(1)$ scans of $T'[s\,..\,n-1]$. This gives a total cost of $O(n^2)$. We use four integer variables (`i`, `p`, `r`, `s`) and one character variable (`c`) in the C code shown in Fig. 2, and thus we need $O(1)$ memory cells for the local variables. □

3. Inverting the BWT

Reversing the permutation performed by the in-place BWT is called *inverting* the BWT. Initially we have the BWT of the original input text T, denoted $\mathrm{BWT}(T)$. We want to invert the latter by permuting its characters. Thus we reverse the approach described in Section 2. We maintain the invariant that there is a pointer L to a certain position in the input buffer storing $\mathrm{BWT}(T)$ so that, at any time, (*a*) the prefix of the buffer to the left of L stores the prefix of T obtained so far by the inverting process and (*b*) the remaining suffix of the buffer (pointed by L till the end of the input buffer) stores the portion of the BWT still to be inverted. For the sake of notation, we identify L with the entire suffix of the input buffer that still has to be inverted.

Under this invariant, which is initially true by setting L to the beginning of the input buffer for $\mathrm{BWT}(T)$, we proceed as follows. We find the position p of \$ in L, and then select the pth character in the alphabet order in the multiset given by the characters of L. Stability is needed, since equal characters should be considered in the order of their appearance in L, as detailed below.

```
void inplaceIBWT( unsigned char L[ ], int n ){
  int f, i, p, q, count;
  unsigned char c;

  /* step 1 */
  p = 0;
  while( L[ p ] != END_MARKER )
    p++;
  p++;

  while ( n > 2 ){
    /* step 2  */
    c = select( L, p );
    count = 0;
    for ( i = 0; i < n; i++ ){
      if (L[i] < c) count++;
    }
    /* step 3 */
    f = p - count;
    q = -1;
    while ( f > 0 ){
      q++;
      if ( L[ q ] == c ) f--;
    }
    /* step 4 */
    L[ q ] = END_MARKER;
    for ( i = p-1; i > 0; i-- ){
      L[i] = L[i-1];
    }
    /* step 5 */
    L[0] = c;
    L++; n--;
    /* step 1 */
    if (p-1 > q)
      p = q+1; /* also the new END_MARKER has been shifted */
    else
      p = q;
  }
}
```

Fig. 3. Reverting the permutation of the inverse BWT.

1. Find the position p of the \$ in L, and increment p (since array indexing starts from 0).
2. Let `select` be a selection algorithm that works on read-only input, i.e., it does not move elements around while finding the pth smallest element. Using `select` on L, select the pth character c in the multiset of the characters of L or, equivalently, the pth character in the sorted list of characters of L.
3. Let q denote the position inside L of the fth occurrence of c, which we hit in a stable fashion when finding c in L. Here f is the difference between p and the number of characters c' of L such that $c' < c$.
4. Replace the occurrence of c at position q by \$, and remove the old occurrence of \$ by shifting to the right the first p characters of L.
5. At this point, the first position in L is free: store the character c in it, and shorten L by one character at the beginning (i.e. advance the pointer L by one position towards the end of the input buffer).

The C code in Fig. 3 implements Steps 1–5 above, where `END_MARKER` denotes \$. Note that it is a bit longer than the code for the in-place BWT in Fig. 2. As it can be seen below, the original text is reconstructed from left to right as a prefix of increasing length (indicated with small letters).

```
IPSSM$PISSII → mIPSS$PISSII → miIPSSPISSI$ → misIPSSPIS$I →
missIPS$PISI → missiIPSPIS$ → missisIPSPI$ → mississIP$PI →
mississiIPP$ → mississipIP$ → mississippI$ → mississippi$
```

The proof of correctness proceeds along the same lines as in the proof of Theorem 1, since we are reversing the procedure described there. As for the complexity, each of the $n-2$ iterations is dominated by the cost of `select`.

Let $t_s(n)$ be the time complexity in the comparison model and $s_s(n)$ be the space complexity required by `select`. Using the result in [23], we have $t_s(n) = O(n^{1+\epsilon})$ in the worst case for any fixed small constant $\epsilon > 0$ with $s_s(n) = O(1)$, and we have $t_s(n) = O(n^{1+\epsilon}) = O(n \log\log n)$ on the average (which meets the randomized lower bound in [5]), with $s_s(n) = O(1)$.

We can state the complexity in general terms.

Theorem 2. *Let $t_s(n)$ be the time complexity in the comparison model and $s_s(n)$ be the space complexity required by* `select`. *Given the* BWT *of a text T of n characters, we can recover T by permuting the* BWT *(also known as inverse* BWT*) in $O(n \cdot t_s(n))$ time in the comparison model using $O(1 + s_s(n))$ additional memory cells.*

We give some examples of the bounds that can be attained with Theorem 2.

Corollary 1. *Given the* BWT *of a text T of n characters, we can recover T by inverting the* BWT *in $O(n^{2+\epsilon})$ time in the worst case, or $O(n^2 \log\log n)$ time on the average, in the comparison model using $O(1)$ additional memory cells.*

Using slightly more additional space than a constant—literally speaking, the algorithm is no more in-place—and the result in [28], where $t_s(n) = O(n(\log n)^2)$ and $s_s(n) = O(\log n)$, we derive the following.

Corollary 2. *Given the* BWT *of a text T of n characters, we can recover T by inverting the* BWT *in $O((n \log n)^2)$ time in the comparison model using $O(\log n)$ additional memory cells.*

Finally, for the special case in which the alphabet of the distinct characters in T is of constant size (as in DNA and ASCII texts), we obtain an improved bound since `select` can be immediately implemented by a simple scheme that employs $O(|\Sigma|) = O(1)$ counters.

Corollary 3. *Given the* BWT *of a text T of n characters drawn from a constant-size alphabet, we can recover T by inverting the* BWT *in $O(n^2)$ time in the comparison model using $O(1)$ additional memory cells.*

4. Practical trade-off between space and time

The inplace algorithms described so far have the drawback of requiring $\Omega(n^2)$ time, which make them unfeasible for long texts. A natural question is how much the latter bound can be improved using extra space. For example, using the dynamic wavelet tree data structure [26] in additional $O(n + |\Sigma| \log n)$ bits of space, we can maintain the BWT through insertion and deletion operations of individual symbols, supporting rank and select operations, with a cost of $O(\log n / \log\log n)$ time per operation. Using the latter data structure, our algorithms in Sections 2 and 3 would give a bound of $O(n \log n)$ time with additional $O(n + |\Sigma| \log n)$ bits of space besides that needed for storing the n characters of the input text T.[3] However the resulting solution is not very practical as the data structure in [26] is quite sophisticated. We show next how to smoothly adapt our algorithms in Sections 2 and 3 to a situation where extra memory is allowed, producing some trade-off solutions that are amenable for implementation with a flexible parameter k for the additional space.

Theorem 3. *Given a text T of n characters, we can compute its Burrows–Wheeler Transform (*BWT*) and its inverse in $O((n^2/k + n) \log k)$ time in the comparison model using $O(k \cdot \sigma_k)$ additional space, where $\sigma_k \leq \min\{|\Sigma|, k\}$ is the maximum number of distinct characters found among k consecutive positions in T.*

To appreciate the bound in Theorem 3 from a practical point of view, consider the situation in which the text T occupies a great part of the available memory, and the remaining free cells are a constant fraction of the text size. Our algorithm takes $O(n \log n)$ time by fixing k to be a suitable fraction of n. This avoids to use external-memory algorithms, which are clearly slower as I/O access takes several order of magnitudes with respect to main memory access. In general, if the available memory size is M, we obtain the following result by setting $k = \Theta(M - n)$.

Corollary 4. *Let M be the number of available cells in main memory. Given a text T of $n < M$ characters over a constant alphabet, we can compute its Burrows–Wheeler Transform (*BWT*) and its inverse in $O((\frac{n^2}{M-n} + n) \log n)$ time in the comparison model using $\leq M$ total memory cells including those containing T. When $M \geq (1 + \epsilon)n$ for a constant $\epsilon > 0$, this gives $O(n \log n)$ time using just ϵn additional cells.*

The idea to prove Theorem 3 is to have n/k batches. Each batch simulates k consecutive iterations in the external `for` loop on s in Fig. 2, taking $O((n + k) \log k)$ time and using $O(k \cdot \sigma_k)$ space as follows.

[3] This theoretical solution has been suggested by Rossano Venturini (private communication).

Base case. Let s_1 be the largest multiple of k that is smaller than n. We can compute BWT($T[s_1 \,..\, n-1]$) and store it in $T[s_1 \,..\, n-1]$ in $O(k \log k)$ time and $O(k)$ additional space by observing that $|T[s_1 \,..\, n-1]| < k$.

Inductive case. Suppose that BWT($T[s_1 \,..\, n-1]$) has been already stored in $T[s_1 \,..\, n-1]$, where $s_1 > 0$ is now a generic multiple of k. Letting $s_0 = s_1 - k$, we want to show how to store BWT($T[s_0 \,..\, n-1]$) in $T[s_0 \,..\, n-1]$ in $O((n+k)\log k)$ time using $O(k \cdot \sigma_k)$ additional space.

Since we have one base case and $\leq n/k$ inductive cases, the final cost will be $O(k \log k + (n/k) \cdot (n+k) \log k) = O((n^2/k + n) \log k)$ time using $O(k \cdot \sigma_k)$ additional space, as stated in Theorem 3.

4.1. Inductive case

We can abstract the problem for a string X (i.e. $T[s_0 \,..\, n-1]$) of length m, where the characters in $X[k \,..\, m-1]$ are already permuted according to their BWT, and the characters in $X[0 \,..\, k-1]$ are still in their original order. We want to compute BWT(X) in $O((m+k)\log k)$ time using $O(k \cdot \sigma_k)$ additional space.

Let Z denote $X[k \,..\, m-1]$ where the \$ character is virtually removed from its position, say j. Hence Z is of length $m-k-1$ and the pair $\langle \$, j \rangle$ is a breakpoint for Z. In general, a *breakpoint* is a pair $\langle c, j \rangle$ such that c is virtually occupying position j of Z: if two or more breakpoints claim the same position j, there should be a relative order among them.

Our goal is to compute the $k+1$ breakpoints for Z so that (a) their characters are those in $X[0 \,..\, k-1]$ plus \$, and (b) flattening Z and these breakpoints correctly produces BWT(X) as follows. Given Z and an ordered list of $k+1$ breakpoints $B = \langle c_0, j_0 \rangle, \ldots, \langle c_k, j_k \rangle$, where $0 \leq j_0 \leq \cdots \leq j_k \leq |Z|$, *flattening* Z and B produces a string with the characters of Z suitably shifted to the left to make room for the characters in the breakpoints of B as follows. We scan Z starting with $j = 0$: if $j = j_r$ for the breakpoint $\langle c_r, j_r \rangle$ at the beginning of B, we output c_r and remove $\langle c_r, j_r \rangle$ from B; else ($j \neq j_r$), we output $Z[j]$ and increase j. (If $c_r = \$$, we do not really output it, but we retain its position j_r for the next batch.) The computation ends when Z has been completely scanned and the list B has been emptied. The required time is $O(m \log k)$ and the computation can be performed using $O(k \cdot \sigma_k)$ additional space.

For this we need the following auxiliary data structures for string Z, which require $O(k \cdot \sigma_k)$ additional space. (Note that Z and X require just $O(1)$ space as they originate from T.)

1. Static array C of $\sigma_k \leq k$ entries, where $\alpha_1 < \cdots < \alpha_{\sigma_k}$ are the distinct characters in $X[0 \,..\, k-1]$: entry $C[i]$ is the number of positions j in Z such that $Z[j] < \alpha_i$.
2. Static rank data structure R_1 on Z supporting queries that, for an integer j' and a character α_i, report how many positions j satisfy $Z[j] = \alpha_i$ and $j \leq j'$.
3. Dynamic list B of breakpoints, initially containing only the pair $\langle \$, j \rangle$.
4. Dynamic rank data structure R_2 on B supporting queries that, for a character α_i, report how many breakpoints $\langle c, l \rangle$ to the left of $\langle \$, j \rangle$ in B satisfy $c = \alpha_i$.

As it is clear, we want to populate the list B by simulating the algorithm described in Section 2. Namely, we want to find the breakpoint of character $X[s']$ for $s' = k-1, k-2, \ldots, 0$ in Z.

Consider the breakpoint $\langle \$, j \rangle$, which exists in B by construction. Let $\alpha_i = X[s']$, and $r' = r_0 + r_1 + r_2$ be sum of three quantities: the number r_0 of positions j' in Z such that $Z[j'] < \alpha_i$, the number r_1 of positions j' in Z such that $Z[j'] = \alpha_i$ and $j' \leq j$, and the number r_2 of breakpoints $\langle c, l \rangle$ that are to the left of $\langle \$, j \rangle$ in B and have $c = \alpha_i$. Note that we can compute r_0 using entry $C[i]$ in point 1, r_1 using the data structure R_1 in point 2, and r_2 using the data structures B and R_2 in points 3–4.

Fact 1. *The value of r' is the rank of $X[s' \,..\, m-1]$ among $X[s'+1 \,..\, m-1]$, $X[s'+2 \,..\, m-1], \ldots, X[m-1 \,..\, m-1]$ in lexicographic order.*

Proof. It follows from the fact that $X \equiv T[s_0 \,..\, n-1]$ and $X[s'+d \,..\, m-1] \equiv T_{s+d}$, where $s = s_0 + s'$ and $d \geq 0$. □

After computing r', we replace the breakpoint $\langle \$, j \rangle$ by $\langle c, j \rangle$, and create a new breakpoint $\langle \$, r' \rangle$ to be inserted in B, updating the data structures in points 3–4 accordingly.

Lemma 1. *The static array C can be stored in $O(k \cdot \sigma_k)$ space and built in $O(m \log k)$ time.*

Proof. We scan $X[0 \,..\, k-1]$ and find the distinct characters $\alpha_1 < \cdots < \alpha_{\sigma_k}$. We then perform a scan of Z to store in $C[1]$ the number of positions j such that $Z[j] < \alpha_1$ and, for $i > 1$, to store in $C[i]$ the number of positions j such that $\alpha_{i-1} \leq Z[j] < \alpha_i$. We then store in C its prefix sums, thus giving the wanted C. Time is $O(m \log \sigma_k)$ since during the scan we perform a binary search among the σ_k characters. Space is $O(\sigma_k)$ by definition of C. □

Lemma 2. *The static data structure R_1 can be stored in $O(k \cdot \sigma_k)$ space and built in $O(m \log \sigma_k)$ time, so that each query requires $O(\log \sigma_k + m/k)$ time.*

Proof. We store R_1 as a collection of σ_k arrays R_1^i, for $1 \le i \le \sigma_k$. Entry $R_1^i[h]$, for $0 \le h \le k$, stores the number of occurrences of character α_i in the hth segment of m/k consecutive positions in Z: namely, $R_1^i[0] = 0$ and, for $h \ge 1$, $R_1^i[h]$ stores the number of positions j such that $Z[j] = \alpha_i$ and $(m/k) \cdot (h-1) \le j \le \max\{m-1, (m/k)h - 1\}$. After initializing the entries in all the arrays to zero, their correct value can be set with a single scan of Z in $O(m \log \sigma_k)$ time.[4] After that, we perform a post-processing and store in each R_1^i its prefix sums: in this way, $R_1^i[h]$ stores the number of positions j such that $Z[j] = \alpha_i$ and $0 \le j \le \max\{m-1, (m/k)h - 1\}$. For a query with a character c' and an integer j', it takes $O(\log \sigma_k)$ time to establish that $c' = \alpha_i$ for a certain i, and $O(1)$ time to find the largest h such that $(m/k)h < j'$. The correct answer for the query is then given by the sum of the content of $R_1^i[h]$ and the number of α_i's found in $Z[(m/k)h \,..\, j']$ by its direct inspection. Scanning the latter takes $O(m/k)$ time by definition of R_1^i. □

Lemma 3. *The dynamic list B and the dynamic data structure R_2 can be stored in $O(k)$ space and built in $O(k \log k)$ time, so that each query and each update requires $O(\log k)$ time.*

Proof. We handle the list $B = \langle c_0, j_0 \rangle, \ldots, \langle c_r, j_r \rangle$, where $0 \le r \le k$, and the data structures R_2 together. In particular, R_2 is the wavelet tree of height $O(\log r)$ built on the sequence $c_0 \cdots c_r$ of characters taken from B. The query for α_i can be performed as a count query of characters α_i in $c_0 \cdots c_d$, where $d \le r$ and $c_d = \$$ (e.g. [25]). Each update (insertion, deletion, replacement) can be handled in $O(\log r + \log k) = O(\log k)$ time [11,26]. □

We now have all the ingredients to prove the time and space bounds for computing the BWT as stated in Theorem 3. By the inductive scheme and Fact 1, the computation is correctly performed. As for the time bounds, we use Lemmas 1–3. Note that to invert the BWT, we can now implement the algorithm described in Section 3 in a simpler way, since R_1 and R_2 support also the selection of the pth symbol $c = \alpha_i$. Moreover, flattening Z and B removes the positions j stored in the pairs in B from Z, as this simulates the shift of some characters of Z to the right. The time and space analysis is similar to that of computing the BWT.

5. Conclusions

We presented an in-place BWT construction taking $O(n^2)$ time in the comparison model. It would be interesting to improve this bound. Note that the `while` loop in our in-place BWT can be avoided using $O(|\Sigma|)$ space, where Σ is the alphabet of characters occurring in T. Time can be further reduced to $O(n^2/\log_{|\Sigma|} n)$ by packing characters but this is still not useful for large text collections.

We do not know whether a lower bound better than $\Omega(n \log n)$ holds for the problem in the comparison model since the space is very constrained. This is an interesting question to investigate.

On the practical side, our in-place algorithms can be adapted to provide a trade-off between space and time, when $O(k \cdot \sigma_k)$ extra memory cells are allowed, providing an $O((n^2/k + n) \log k)$-time algorithm to obtain (and invert) the BWT. For example in real systems where the alphabet size is a constant, for any arbitrarily small $\epsilon > 0$, the BWT of a text of n bytes can be computed in $O(n\epsilon^{-1} \log n)$ time using just ϵn extra bytes.

Acknowledgements

The second author is grateful to Gianni Franceschini for some preliminary discussions and to Venkatesh Raman for pointing out the results in [23,28] and Rossano Venturini for his comments and indicating the result in [3]. We also thank the anonymous reviewers for their careful reading of our manuscript.

References

[1] Donald Adjeroh, Timothy Bell, Amar Mukherjee, The Burrows–Wheeler Transform: Data Compression, Suffix Arrays, and Pattern Matching, Springer, 2008.
[2] Alfred Aho, John Hopcroft, Jeff D. Ullman, The Design and Analysis of Computer Algorithms, Addison-Wesley, Reading, MA, 1974.
[3] Djamal Belazzougui, Linear time construction of compressed text indices in compact space, in: Proc. STOC 2014, 2014, pp. 148–193.
[4] Michael Burrows, David J. Wheeler, A block-sorting lossless data compression algorithm, Research Report 124, Digital SRC, Palo Alto, CA, USA, May 1994.
[5] Timothy M. Chan, Comparison-based time–space lower bounds for selection, ACM Trans. Algorithms 6 (2) (2010) 1–16.
[6] Maxime Crochemore, Roberto Grossi, Juha Kärkkäinen, Gad M. Landau, A constant-space comparison-based algorithm for computing the Burrows–Wheeler transform, in: Combinatorial Pattern Matching, 24th Annual Symposium (CPM), 2013, pp. 74–82.
[7] David J. Dobkin, J. Ian Munro, Optimal time minimal space selection algorithms, J. ACM 28 (3) (July 1981) 454–461.
[8] Paolo Ferragina, Giovanni Manzini, Indexing compressed text, J. ACM 52 (4) (2005) 552–581.
[9] Gianni Franceschini, S. Muthukrishnan, In-place suffix sorting, automata, languages and programming, in: 34th International Colloquium (ICALP), 2007, pp. 533–545.
[10] Roberto Grossi, Ankur Gupta, Jeffrey S. Vitter, High-order entropy-compressed text indexes, in: ACM–SIAM SODA, 2003, pp. 841–850.

[4] If $k \cdot \sigma_k = w(n \log \sigma_k)$, it is a standard trick to allocate $O(k \cdot \sigma_k)$ memory and initialize only what is needed in $O(m \log \sigma_k)$ time [2].

[11] Roberto Grossi, Giuseppe Ottaviano, The wavelet trie: maintaining an indexed sequence of strings in compressed space, in: ACM PODS, 2012, pp. 203–214.
[12] C.A.R. Hoare, Algorithm 65: find, Commun. ACM 4 (7) (July 1961) 321–322.
[13] Wing-Kai Hon, Tak Wah Lam, Kunihiko Sadakane, Wing-Kin Sung, Siu-Ming Yiu, A space and time efficient algorithm for constructing compressed suffix arrays, Algorithmica 48 (1) (2007) 23–36.
[14] Wing-Kai Hon, Kunihiko Sadakane, Wing-Kin Sung, Breaking a time-and-space barrier in constructing full-text indices, SIAM J. Comput. 38 (6) (2009) 2162–2178.
[15] Juha Kärkkäinen, Fast BWT in small space by blockwise suffix sorting, Theor. Comput. Sci. 387 (3) (2007) 249–257.
[16] T.W. Lam, Ruiqiang Li, Alan Tam, Simon Wong, Edward Wu, S.M. Yiu, High throughput short read alignment via bi-directional BWT, in: IEEE International Conference on Bioinformatics and Biomedicine, 2009, pp. 31–36.
[17] T.W. Lam, W.K. Sung, S.L. Tam, C.K. Wong, S.M. Yiu, Compressed indexing and local alignment of DNA, Bioinformatics 24 (6) (2015) 791–797.
[18] Heng Li, Richard Durbin, Fast and accurate long-read alignment with Burrows–Wheeler transform, Bioinformatics 26 (5) (2010) 589–595.
[19] Ben Langmead, Cole Trapnell, Mihai Pop, Steven L. Salzberg, Ultrafast and memory-efficient alignment of short DNA sequences to the human genome, Genome Biol. 10 (3) (2009) R25.
[20] Udi Manber, Gene Myers, Suffix arrays: a new method for on-line string searches, SIAM J. Comput. 22 (5) (October 1993) 935–948.
[21] Giovanni Manzini, An analysis of the Burrows–Wheeler transform, J. ACM 48 (3) (2001) 407–430.
[22] J. Ian Munro, Tables, in: V. Chandru, V. Vinay (Eds.), Proc. of Foundations of Software Technology and Theoretical Computer Science, 16th Conference, Hyderabad, India, December 18–20, 1996, in: Lect. Notes Comput. Sci., vol. 1180, Springer, 1996, pp. 37–42.
[23] J. Ian Munro, Venkatesh Raman, Selection from read-only memory and sorting with minimum data movement, Theor. Comput. Sci. 165 (2) (1996) 311–323.
[24] Joong Chae Na, Kunsoo Park, Alphabet-independent linear-time construction of compressed suffix arrays using $o(n \log n)$-bit working space, Theor. Comput. Sci. 385 (1–3) (2007) 127–136.
[25] Gonzalo Navarro, Wavelet trees for all, J. Discrete Algorithms 25 (2014) 2–20.
[26] Gonzalo Navarro, Yakov Nekrich, Optimal dynamic sequence representations, in: ACM–SIAM SODA, 2013, pp. 865–876.
[27] Daisuke Okanohara, Kunihiko Sadakane, A linear-time Burrows–Wheeler transform using induced sorting, in: 16th Symposium on String Processing and Information Retrieval (SPIRE), in: Lect. Notes Comput. Sci., vol. 5721, Springer, 2009, pp. 90–101.
[28] Venkatesh Raman, Sarnath Ramnath, Improved upper bounds for time–space trade-offs for selection, Nord. J. Comput. 6 (2) (1999) 162–180.
[29] Mikael Salson, Thierry Lecroq, Martine Léonard, Laurent Mouchard, A four-stage algorithm for updating a Burrows–Wheeler transform, Theor. Comput. Sci. 410 (43) (2009) 4350–4359.