Theoretical Computer Science 483 (2013) 134–148

Contents lists available at SciVerse ScienceDirect

Theoretical Computer Science

journal homepage: www.elsevier.com/locate/tcs

Lightweight algorithms for constructing and inverting the BWT of string collections

Markus J. Bauer [a], Anthony J. Cox [a], Giovanna Rosone [b,*]

[a] Illumina Cambridge Ltd., United Kingdom

[b] University of Palermo, Dipartimento di Matematica e Informatica, Via Archirafi 34, 90123 Palermo, Italy

ARTICLE INFO

Keywords:
BWT
Text indexes
Next-generation sequencing

ABSTRACT

Recent progress in the field of DNA sequencing motivates us to consider the problem of computing the Burrows–Wheeler transform (BWT) of a collection of strings. A human genome sequencing experiment might yield a billion or more sequences, each 100 characters in length. Such a dataset can now be generated in just a few days on a single sequencing machine. Many algorithms and data structures for compression and indexing of text have the BWT at their heart, and it would be of great interest to explore their applications to sequence collections such as these. However, computing the BWT for 100 billion characters or more of data remains a computational challenge.

In this work we address this obstacle by presenting a methodology for computing the BWT of a string collection in a lightweight fashion. A first implementation of our algorithm needs $O(m \log m)$ bits of memory to process m strings, while a second variant makes additional use of external memory to achieve RAM usage that is constant with respect to m and negligible in size for a small alphabet such as DNA. The algorithms work on any number of strings and any size. We evaluate our algorithms on collections of up to 1 billion strings and compare their performance to other approaches on smaller datasets.

We take further steps toward making the BWT a practical tool for processing string collections on this scale. First, we give two algorithms for recovering the strings in a collection from its BWT. Second, we show that if sequences are added to or removed from the collection, then the BWT of the original collection can be efficiently updated to obtain the BWT of the revised collection.

1. Introduction

We consider the problem of computing the Burrows–Wheeler transform (BWT) of very large datasets. Our interest in this topic arises from DNA sequencing, a field that has been transformed by the advent of 'next-generation' sequencing technologies [18]. Whole-genome sequencing of human DNA may sample the 3 billion base-pairs of the human genome to 30× redundancy or higher. A modern DNA sequencing machine can produce this quantity of data in just a few days and datasets of 100Gbases or larger have therefore become commonplace.

The BWT of such a dataset would be useful to have for the purposes of data compression or for creating self-indexing data structures such as the FM-index [6]. Apart from its intrinsic use in data compression, answering generic `count` and `locate` queries efficiently on large string collections can be used in a variety of bioinformatics tasks, ranging from de novo assembly [23,24], computing counts for k-mers in de novo error correction [20] to predicting splice variants without a reference alignment [4]. However, generating the BWT for datasets on this scale is the challenge that we address in this paper.

* Corresponding author. Tel.: +39 09123891037.
E-mail addresses: mbauer@illumina.com (M.J. Bauer), acox@illumina.com (A.J. Cox), giovyros@virgilio.it, giovanna@math.unipa.it (G. Rosone).

0304-3975/$ – see front matter © 2012 Elsevier B.V. All rights reserved.
doi:10.1016/j.tcs.2012.02.002

Computing the BWT of a string from its suffix array (SA) is simple and fast, and much effort has been devoted to devising algorithms for SA construction that are efficient in both space and CPU time [12–14]. However, all share the need for the entire SA to be held in RAM, which becomes problematic for datasets of the size in which we are interested. For instance, Simpson and Durbin [23] extrapolate from their experiences with a variant of the SA construction algorithm by Nong et al. [19] to estimate 700Gbytes of RAM would be necessary to build the FM-index of a 20× oversampling of the human genome (60Gbases) by this route.

Kärkkäinen [11] addresses this bottleneck in an interesting way by constructing the suffix array a small block at a time without the need to store the rest of the suffix array anywhere. Ferragina et al. [5] take a different blockwise approach, building the BWT of a large string T by logically partitioning it into r blocks $T_r \cdots T_1$. Working from right to left in T, r passes through the data are made. At pass h, the characters of $\mathsf{bwt}(T_h \cdots T_1)$ are read sequentially from an external file and the suffix array of T_{h+1} is used to compute the updates necessary to generate $\mathsf{bwt}(T_{h+1} \cdots T_1)$ (following the idea of [9]), which again is written to disk in a sequential fashion. By using external memory in this way, the RAM usage of the algorithm is dominated by the suffix array of the block T_{h+1} being merged, so the number of blocks r can be adjusted to 'tune' the RAM usage of the algorithm to fit the memory available, at the expense of a dependence on r^2 of the disk I/O needed to read and write the BWT strings.

With applications to text indexing in mind, Sirén [25] is one of the relatively few papers to consider explicitly the problem of BWT computation for a *collection* of strings (as opposed to a single large string) and gives a divide-and-conquer strategy for computing the BWT of such a collection by first splitting the collection into batches, computing the BWT of each in a distributed fashion and then merging the results. The RAM requirements are highest at this final merge stage — Sirén quotes 32Gbytes to index 42.03Gbytes of English text if 8 parallel threads are used, and double that if 16 threads are used.

Some studies [10,22] showed that one can compute the BWT without a SA by implicitly adding suffixes from the shortest ones to longest ones. However, these algorithms are slow due to the large constant factor.

In this paper, we present a lightweight algorithm for constructing the BWT of a collection of strings (introduced in [1]) and for reconstructing the collection of strings from the BWT. Before this, we must consider how the notion of the BWT should be extended from a single string to a collection of strings. This generalization can be defined in more than one way, the differences centering around how string comparisons that extend across string boundaries are handled. A conceptually simple approach is to append an end marker character to each member of the collection and then concatenate them, thus obtaining a single string to which standard methods for BWT construction can be applied. Mantaci et al. [15] gave an alternative generalization of the BWT to a collection of strings that does not rely on concatenating the members of the collection (see also [16,17]). Here, we do not concatenate the strings of the collection and append a unique terminating character to each string.

In Section 3, we first outline the basic idea of our algorithm for computing the BWT of a collection of strings of any length (see also [1]) and then describe two practical implementations, called `BCR` and `BCRext` of the method that make different tradeoffs between the RAM used and the volume of data held in external memory. Our implementations are lightweight in the sense that, for a collection of m strings of length k, one uses only $O(m \log(mk))$ bits of space and $O(k\,\mathrm{sort}(m))$ time, where $\mathrm{sort}(m)$ is the time taken to sort m integers. The second implementation works almost entirely in external memory, taking $O(km)$ time with RAM usage that is constant for collections of any size, dependent only on the alphabet size and therefore negligible for DNA data. The overall I/O volume is $O(mk^2 \log(\sigma))$ bits, so the length of the longest string in the collection dominates the I/O complexity for a collection of m strings. Our algorithms are scan-based in the sense that all files are accessed in a manner that is entirely sequential and either wholly read-only or wholly write-only. Table 1 states the complexities in terms of CPU and I/O for both implementations.

In Section 4, we first prove that the generalization of the BWT we study in this paper is reversible, in that the members of a string collection can be recovered from the BWT of the collection. We then introduce two algorithms to retrieve the collection of strings from the BWT: a *forward* algorithm that retrieves the characters of the strings in left-to-right order and a *backward* algorithm that retrieves them in right-to-left order (or, equivalently, can be thought of as constructing the reverse of the strings in the collection). Both algorithms keep the BWT in the external memory.

Finally, in Section 5 we report on computational experiments in which we compare the BWT construction methods described in Section 3 to the Sirén [25] and Ferragina et al. [5] algorithms and assess how the performance of our algorithms scales to string collections as large as one billion 100-mers.

2. Preliminaries

Let $\Sigma = \{c_1, c_2, \ldots, c_\sigma\}$ be a finite ordered alphabet with $c_1 < c_2 < \cdots < c_\sigma$, where $<$ denotes the standard lexicographic order. Given a finite string $w = w_0 w_1 \cdots w_{k-1}$ with each $w_i \in \Sigma$, a *substring* of a string w is written as $w[i, j] = w_i \cdots w_j$. A substring of type $w[0, j]$ is called a *prefix*, while a substring of type $w[i, k-1]$ is called a *suffix*. We denote by *j-suffix* the suffix of w that has length j and by *j-prefix* the prefix of w that has length j. The *concatenation* of two words w and v, written wv, is simply the string consisting of the symbols of w followed by the symbols of v. We denote by $\tilde{w}$ the reversal of w, given by $\tilde{w} = w_{k-1} \cdots w_1 w_0$. We say that two words $x, y \in \Sigma^*$ are *conjugate* if $x = uv$ and $y = vu$ for some $u, v \in \Sigma^*$. A word x is called a *cyclic shift* or a *cyclic rotation* of y if x and y are conjugate. A word $v \in \Sigma^*$ is *primitive* if $v = u^n$ implies $v = u$ and $n = 1$.

Table 1
Time and space complexities for the two implementations of our algorithm, BCR and BCRext, for m sequences of length k.

	BCR	BCRext
CPU time	$O(k \text{ sort}(m))$	$O(km)$
RAM usage (bits)	$O((m+\sigma^2)\log(mk))$	$O(\sigma^2 \log(mk))$
I/O (bits)	$O(mk^2 \log(\sigma))$ (partial BWT) $O(mk \log(\sigma))$ (current symbols)	$O(mk^2 \log(\sigma))$ (partial BWT) $O(mk^2 \log(\sigma))$ (sequences) $O(mk \log(mk))$ (P array) $O(mk \log(mk))$ (N array)

			F					L
			↓					↓
		0	*a*	*a*	*b*	*r*	*a*	*c*
I	→	1	*a*	*b*	*r*	*a*	*c*	*a*
		2	*a*	*c*	*a*	*a*	*b*	*r*
		3	*b*	*r*	*a*	*c*	*a*	*a*
		4	*c*	*a*	*a*	*b*	*r*	*a*
		5	*r*	*a*	*c*	*a*	*a*	*b*

Fig. 1. The matrix of all cyclic rotations of the word $w = abraca$.

Let $\mathsf{S} = \{S_1, S_2, \ldots, S_m\}$ be a collection of m strings, each comprising k symbols drawn from Σ. Each S_i can be imagined to have appended to it an end marker symbol \$ that satisfies $\$ < c_1$. We define lexicographic order among the strings in the usual way, except that each end marker \$ is considered a different symbol, so that every suffix of every string is unique in the collection. The (implicit) end marker is in position k, i.e. $S_i[k] = S_j[k] = \$$, and we define $S_i[k] < S_j[k]$, if $i < j$.

For simplicity of explaining, when we refer to an end marker of a specific string $S_i \in \mathsf{S}$, we will use $\$_i$ rather than \$. We define the 0-suffix as the suffix that contains only the end marker \$ and the $(k+1)$-prefix as the prefix that contains the entire string concatenated by the end marker \$.

The suffix array *SA* of a string w is the permutation of integers giving the starting positions of the suffixes of w in lexicographical order. We refer interested readers to [21] for further reading on suffix arrays.

2.1. The original BWT

The Burrows–Wheeler transform (BWT from now on) is a transformation that defines a permutation of an input string w. The two keys to its importance in data compression are that it is *reversible* (in that the original string can be reconstructed from its BWT) and that the transformed string tends to be more *compressible* (in that the symbols of the input string tend to group into runs of like characters, rendering the transformed string more amenable to compression by standard methods such as run-length encoding). Combined, these two properties meant that the BWT quickly became a basic technique in lossless compression of text data once its description was published in 1994 [3].

In this section we will define the BWT and describe some properties of the transform that will be important in later sections. We distinguish between an *encode* transformation, which produces the string transformed, and a *decode* transformation which gives back the original string from the transformed one.

One way to define the encode transformation *encodeBWT* of a string $w = w_0 w_1 \cdots w_{k-1}$ (although not the most efficient way to compute it) is to construct all k cyclic rotations of w and sort them lexicographically. The output of *encodeBWT* consists of the pair (L, I), where L is the sequence of the last character of each rotation in the sorted list and I is an integer denoting the position of the original word in the list.

For instance, suppose we want to compute the encode transformation $w = abraca$. Consider the matrix M, shown in Fig. 1, which consists of all cyclic shifts of w, lexicographically sorted.

The last column $L = caraab$ of the matrix and $I = 1$ are the output of *encodeBWT*. The first column F contains the characters of w, lexicographically sorted.

Notice that, for each member of a set of mutually conjugate words, the BWT will have the same L and differ only in I, whose purpose is only to distinguish between the different members of the conjugacy class. However, I is not necessary for the construction of the matrix M from the last column L.

The cyclic shift of the rows of M is crucial to define the decode BWT, which is based on two easy to prove observations [3] of the following proposition.

Proposition 2.1. *Let w be a string and let (L, I) be its encode transformation. Let F be the sequence of the sorted characters of L. The following properties hold:*

1. *For all $i = 0, \ldots, k-1$, $i \neq I$, the character $F[i]$ follows $L[i]$ in the original string;*
2. *for each character c, the r-th occurrence of c in F corresponds to the r-th occurrence of c in L.*

Property 2 of Proposition 2.1 can be reformulated in the following way (see [6]) in order to define the *LF-mapping*: let $L[i] = c$, for $0 \leq i \leq k-1$, and r_i be the number of occurrences of c in the prefix $L[0, i]$. Let $M[j]$ be the r_i-th row of M starting with c. The character in the first column F corresponding to $L[i]$ is located at $F[j]$. So $LF[i] = j$.

From the above properties it follows that the BWT is reversible in the sense that, given L and I, it is possible to recover the original string w.

Actually, according to Property 2 of Proposition 2.1, we can define two permutations $\tau\colon \{0, \ldots, k-1\} \to \{0, \ldots, k-1\}$ and $\pi\colon \{0, \ldots, k-1\} \to \{0, \ldots, k-1\}$ where τ gives the correspondence between the positions of characters of the first and the last column of the matrix M. Its inverse π gives the correspondence between the positions of characters of the last and the first column of the matrix M.

The function τ represents also the order in which we have to rearrange the elements of F to reconstruct the original word w. Hence, starting from the position I, we can recover the word w as follows:

$$w_i = F[\tau^{i-1}(I)], \quad \text{where } \tau^0(x) = x, \quad \text{and} \quad \tau^{i+1}(x) = \tau(\tau^i(x)).$$

The function π represents also the order in which we have to rearrange the elements of L to reconstruct the reverse original word $\tilde{w}$.

Starting from the position I, we can recover the reverse word $\tilde{w}$ as follows:

$$\tilde{w}_i = L[\pi^{i-1}(I)], \quad \text{where } \pi^0(x) = x, \quad \text{and} \quad \pi^{i+1}(x) = \pi(\pi^i(x)).$$

We show, for instance, how the reconstruction works for the example in Fig. 1:

$$\tau = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 \\ 1 & 3 & 4 & 5 & 0 & 2 \end{pmatrix},$$

$$\begin{aligned} w_0 &= F[1] = a \\ w_1 &= F[3] = b \\ w_2 &= F[5] = r \\ w_3 &= F[2] = a \\ w_4 &= F[4] = c \\ w_5 &= F[0] = a. \end{aligned}$$

There is a strong relation between the matrix M and the suffix array SA of the string w. When sorting the rows of the matrix M we are essentially sorting the suffixes of $w\$$. Moreover, the index I corresponds to the position of the symbol \$ in L, so it can be recovered from L and we can omit it.

3. The encode transform

In this section we describe an algorithm, inspired by [15,5] and introduced in [1], that computes the BWT of a collection of strings without concatenating the strings and without needing to compute their suffix array. In the sequel, we suppose that the collection S has m strings of length k and we assume that $j = 1, 2, \ldots, k$ and $i = 1, 2, \ldots, m$. However, our algorithms are not bound to string collections where all the strings have the same length.

Our algorithm computes the BWT of the collection S incrementally via k iterations. At each of the iterations $j = 1, 2, \ldots, k-1$, the algorithm computes a partial BWT string $\mathsf{bwt}_j(\mathsf{S})$ by inserting the symbols preceding the j-suffixes of S at their correct positions into $\mathsf{bwt}_{j-1}(\mathsf{S})$. Each iteration j simulates the insertion of the j-suffixes in the SA. The string $\mathsf{bwt}_j(\mathsf{S})$ is a 'partial BWT' in the sense that the addition of m end markers in their correct positions would make it the BWT of the collection $\{S_1[k-j-1, k], S_2[k-j-1, k], \ldots, S_m[k-j-1, k]\}$.

A trivial 'iteration 0' sets the initial value of the partial BWT by simulating the insertion of the end markers \$ in the SA. Since their ordering is entirely determined by the position in S of the string they belong to, $\mathsf{bwt}_0(\mathsf{S})$ is just the concatenation of the last non-\$ symbol of each string, that is $S_1[k-1]S_2[k-1]\cdots S_m[k-1]$.

Finally, iteration k inserts m end markers into $\mathsf{bwt}_{k-1}(\mathsf{S})$ at their correct positions. This simulates the insertion of the suffixes corresponding to the entire strings into the SA.

Like in [5], the fundamental observation is that going from $\mathsf{bwt}_{j-1}(\mathsf{S})$ to $\mathsf{bwt}_j(\mathsf{S})$ at iteration j only requires that we insert m new symbols and does not affect the relative order of the symbols already in $\mathsf{bwt}_{j-1}(\mathsf{S})$. We can think of $\mathsf{bwt}_j(\mathsf{S})$ as being partitioned into $\sigma + 1$ strings $B_j(0), B_j(1), \ldots, B_j(\sigma)$, with the symbols in $B_j(h)$ being those that are associated with the suffixes of S that are of length j or less and begin with $c_0 = \$$ and $c_h \in \Sigma$, for $h = 1, \ldots, \sigma$. We note that $B_j(0)$ is constant for all j and, at each iteration j, we store $B_j(h)$ in $\sigma + 1$ external files that are sequentially read one-by-one.

During the iteration $j = 1, \ldots, k$, we must insert the symbol associated with the new suffix $S_i[k-j, k]$ of each string $S_i \in \mathsf{S}$ (this symbol is $S_i[k-j-1]$ for $j < k$, or \$ at the final iteration) into the BWT segment $B_j(z)$, where $c_z = S_i[k-j]$ (we

recall that $B_j(z)$ contains all symbols associated with suffixes starting with the symbol c_z). Our main idea is that the position in $B_j(z)$ where this symbol needs to be inserted can be computed from the position r where, in the previous step, the symbol c_z has been inserted into the BWT segment $B_{j-1}(v)$, where $c_v = S_i[k-(j-1)]$ (we recall that $B_{j-1}(v)$ contains all symbols associated with suffixes that have already been inserted and that start with the symbol c_v).

To do this, we need to retain the BWT segments $B_{j-1}(h)$, for $0 \leq h \leq \sigma$, and keep track of the positions within them of the symbols that correspond to the $(j-1)$-suffixes of S, which we do by associating to each $B_{j-1}(h)$ an array $P_{j-1}(h)$ of integers that stores the absolute positions of the $(j-1)$-suffixes starting with c_h. Each $P_{j-1}(h)$ is in turn associated with an array $N_{j-1}(h)$ that has the same number of entries and is such that $N_{j-1}(h)[q]$ stores i, the original position in S of the string S_i whose $(j-1)$-suffix is pointed to by $P_{j-1}(h)[q]$. Here q is a generic subscript of the array $N_{j-1}(h)$ or (equivalently, since their number of entries is the same) $P_{j-1}(h)$. The maximum value of q is determined by the number of $(j-1)$-suffixes starting with c_h and will therefore vary with both h and j.

Stated formally, at the start of iteration j, we assume the following structures are available for each $h = 0, \ldots, \sigma$, where $c_0 = \$$ and $c_n \in \Sigma$, for $n = 1, \ldots, \sigma$, and the maximum value of q depends on the number of the $(j-1)$-suffixes starting with c_h:

$B_{j-1}(h)$ is a segment of the partial BWT.
$N_{j-1}(h)$ is an array of integers such that $N_{j-1}(h)[q]$ is associated with the $(j-1)$-suffix of the string $S_i \in \mathsf{S}$, where $i = N_{j-1}(h)[q]$.
$P_{j-1}(h)$ is an array of integers such that $P_{j-1}(h)[q]$ is the absolute position of the symbol $S_i[k-j]$, associated with the $(j-1)$-suffix of S_i, in $B_{j-1}(h)$, where $i = N_{j-1}(h)[q]$.

Hence, at the end of the iteration $j-1$, for each element in N_{j-1} and P_{j-1}, we have that the symbol $c_z = S_i[k-j]$, with $i = N_{j-1}(v)[q]$, has been inserted in the position $P_{j-1}(v)[q]$ in $B_{j-1}(v)$, where $c_v = S_i[k-(j-1)]$.

During the iteration j, we have to update these structures for each string $S_i \in \mathsf{S}$. The crucial point is to insert the new symbol associated with the j-suffix of S_i into $B_{j-1}(z)$, where $c_z = S_i[k-j]$, for some $z = 1, \ldots, \sigma$, at its correct position in order to obtain $B_j(z)$. Hence, our task is to compute $P_j(h)$ by considering how many suffixes of S that are of length j or less are smaller than each suffix of length j.

The following lemma (similar to [5, Lemma 1]) is the key to this point and it is based on a function called *LF-mapping* [7] that is also used extensively in compressed self-indexes. This method is based on the count of symbols, from first position to the position of the last inserted symbol of S_i in $\mathsf{bwt}_{j-1}(\mathsf{S})$, that are smaller than $c_z = S_i[k-j]$. It is equivalent to count the number of symbols that are associated with suffixes smaller than $S_i[k-j, k]$. We observe that we do not need to do exactly this, because the suffixes starting with a symbol smaller than c_z are associated with symbols in $B_{j-1}(r)$ for $r = 0, \ldots, z-1$. So, we only need to count how many suffixes of length j or less starting with the symbol c_z are smaller than the suffix $S_i[k-j, k]$.

Lemma 3.1. *For any iteration $j = 1, 2, \ldots, k$, given a symbol c_h, with $0 \leq h \leq \sigma$, let q be an index that depends on the number of the $(j-1)$-suffixes starting with c_h. For each string $S_i \in \mathsf{S}$, with $i = N_{j-1}(h)[q]$, we assume that the suffix $S_i[k-(j-1), k]$ is lexicographically larger than precisely $r = P_{j-1}(v)[q]$ suffixes of length $0, 1, \ldots, j-1$ that begin with the symbol $c_v = S_i[k-(j-1)]$. Now, we fix $c_z = S_i[k-j]$. Then the new suffix $S_i[k-j, k] = c_z S_i[k-(j-1), k]$ is lexicographically larger than precisely r' suffixes of length $0, 1, \ldots, j$, where $r' = \mathsf{rank}(c_z, r, c_v)$ and $\mathsf{rank}(c_z, r, c_v)$ denotes the number of occurrences of c_z in $B_{j-1}(0) \cdots B_{j-1}(v-1) B_{j-1}(v)[0, r-1]$.*

Proof. The proof is similar to that of Ferragina et al. in [5]. By hypothesis $c_z = S_i[k-j]$ and $c_v = S_i[k-(j-1)]$. Clearly, $S_i[k-j, k]$ is larger than the suffixes starting with a symbol smaller than c_z (they are associated with the symbols in $B_{j-1}(0), \ldots, B_{j-1}(z-1)$), and is smaller than all suffixes starting with a symbol greater than c_z (they are associated with the symbols in $B_{j-1}(z+1), \ldots, B_{j-1}(\sigma)$). Since the suffixes starting with c_z are associated with the symbols in $B_{j-1}(z)$, the correct position of the symbol associated with the suffix $S_i[k-j, k]$ is in $B_{j-1}(z)$. Now, the crucial point is to compute how many suffixes of length j or less starting with c_z are smaller than $S_i[k-j, k]$. The sorting of the rows in BWT implies that counting how many suffixes starting with c_z in $\{S_1[k-j, k], S_2[k-j, k], \ldots, S_m[k-j, k]\}$ that are smaller than $S_i[k-j, k]$ is equivalent to counting the number of occurrences of c_z in $B_{j-1}(0), \ldots, B_{j-1}(v-1)$ and in $B_{j-1}(v)[0, r-1]$. This is precisely $\mathsf{rank}(c_z, r, c_v)$. □

The positions of each j-suffix $S_i[k-j, k]$ are computed using Lemma 3.1 and stored in P_j according to the symbol $S_i[k-j]$. In other words, if $c_z = S_i[k-j]$, the computed position r' is stored into $P_j(z)$ and i is stored into $N_j(z)$. Moreover, the value r' corresponds to the *absolute* position in $B_j(z)$ where we have to insert the new symbol associated with $S_i[k-j, k]$ starting with c_z. This means that, for each symbol c_h, with $0 \leq h \leq \sigma$, we consider, in the computation of the new positions, all new j-suffixes in S that begin with c_h. Hence, if $S_r[k-j] = S_s[k-j]$ and $S_r[k-(j-1), k] < S_s[k-(j-1), k]$, for some $1 \leq r, s \leq m$, it follows that $S_r[k-j, k] < S_s[k-j, k]$. For this reason and since each $B_j(h)$ is stored in an external file, we have to sort each $P_j(h)$ (respectively $N_j(h)$) and insert the new symbols according to the value of their position, from the smallest to the largest. Given this information, we can build $B_j(1), \ldots, B_j(\sigma)$ by using the current files $B_{j-1}(1), \ldots, B_{j-1}(\sigma)$. The idea is very simple; we read — in a sequential way — each external file associated with each $B_j(h)$ once and insert all symbols associated with the j-suffixes starting with the symbol c_h. Once all the symbols are read and copied, $B_{j-1}(0) \cdots B_{j-1}(\sigma)$ form

$B_j(0) \cdots B_j(\sigma)$ respectively, i.e. the partial BWT string required by the next iteration. Since we no longer need $P_{j-1}(h)$ and $B_{j-1}(h)$, we can write $P_j(h)$ and $B_j(h)$ over the already processed $P_{j-1}(h)$ and $B_{j-1}(h)$.

The counts for $B_{j-1}(d)$, $c_d < S[k-j]$, are dealt with by keeping a count of the number of occurrences of each symbol for all $B_{j-1}(h)$ in memory, which takes $O(\sigma^2 \log(mk))$ bits of space. For $B_{j-1}(z)$, $c_z = S[k-j]$, the pointer value corresponding to S—which we read from $P_{j-1}(h)$—tells us how far along the count needs to proceed in $B_{j-1}(z)$. So for each B_{j-1} we need $O(\sigma \log(mk))$ bits of space: a trivial amount for DNA data, although potentially an issue for very large alphabets.

We can summarize the steps at the iteration j in the following way:

1. For each symbol c_v, with $0 \le v \le \sigma$ and for each element q (we observe that the maximum value of q depends on the number of the $(j-1)$-suffixes starting with c_v), we know:
   - The number of the sequence $i = N_{j-1}(v)[q]$ (clearly $S_i[k-(j-1)] = c_v$).
   - $r = P_{j-1}(v)[q]$ (it means that $c_z = S_i[k-j]$ has been inserted in the position r of $B_{j-1}(v)$ at the end of the previous step).
   - By using c_z, r and c_v, we compute $r' = \mathsf{rank}(c_z, r, c_v)$ (see Lemma 3.1), *i.e.* the position where we have to insert the new symbol in $B_j(z)$. We store r' into $P_j(z)$.
   - We store i into $N_j(z)$.
   - We observe that we do not need to store c_z, because we can read the symbol c_z from $B_{j-1}(v)$ when we compute the new position.
2. For each symbol c_z, with $0 \le z \le \sigma$, we sort the pair $(P_j(z), N_j(z))$ in ascending order, where $P_j(z)$ is the primary key.
3. For each symbol c_z, with $0 \le z \le \sigma$, and for each element q (where the maximum value of q depends on the number of the j-suffixes starting with c_z), we insert the new symbol associated with j-suffix of the string S_i, where $i = N_j(z)[q]$, into $B_j(z)$ in the position $P_j(z)[q]$.
4. Return B_j, P_j, N_j.

We developed prototypical implementations `BCR` and `BCRext` of the algorithm described which are available upon request from the authors. `BCR` and `BCRext` differ in the way they sort the suffixes that are inserted in iteration $j+1$, given the BWT for all suffixes up to iteration j.

Algorithm 1 - BCR. In order to minimize the I/O needed to read the m symbols associated with the m suffixes of length j, our first algorithm begins with a preprocessing step that splits the elements of S into k files in a 'columnwise' fashion, the j-th file containing the j-th character of each sequence in S. In this way, only a single file of m symbols needs to be read at each step.

For each iteration j, we allocate a unique array P_j for all $P_j(h)$ and a unique array N_j for all $N_j(h)$ in internal memory. We observe that P_j and N_j contain exactly one integer for each sequence in the collection, P_j uses $O(m \log(mk))$ bits of workspace and N_j uses $O(m \log m)$ bits of workspace. Since $P_j[q]$, for some q, denotes the position in $B_j(z)$ of the new symbol associated with the j-suffix $S_i[k-j, k]$ starting with $c_z = S_i[k-j]$ and $i = N_j[q]$, we need another array Q_j, setting $Q_j[q] = z$. It uses $O(m \log \sigma)$ bits of workspace. We do not want to read the σ external files containing the BWT segments B_j more than once and since the values in P_j are absolute positions (see the above description), we need to sort the values in P_j before inserting the new symbols. The first, second and third keys of the sort are the values in Q_j, P_j and N_j respectively.

Algorithm 2 - BCRext. Our second algorithm is based on least-significant-digit radix sort. For this variant, sorting of arrays is not required because the sequences themselves are sorted externally. At the start of iteration j, the elements of S are assumed to be ordered by $(j-1)$-suffix, this ordering being partitioned into external files $T_{j-1}(1), \ldots, T_{j-1}(\sigma)$ according to the first characters of the $(j-1)$-suffixes. Files $P_{j-1}(1), \ldots, P_{j-1}(\sigma)$ are such that $P_{j-1}(h)$ contains the positions of the $(j-1)$-suffixes in $B_{j-1}(h)$, ordered the same way.

All files are assumed to be accessed sequentially via read-only $\mathsf{R}()$ or write-only $\mathsf{W}()$ file streams. In the order $h = 1, \ldots, \sigma$, we open read-only file streams to each of $T_{j-1}(h)$ and $P_{j-1}(h)$, while *two* read-only file streams $\mathsf{R}_1(B_{j-1}(h))$ and $\mathsf{R}_2(B_{j-1}(h))$ reading from each segment of the partial BWT remain open throughout the iteration.

Reading a string $S \in \mathsf{S}$ from $\mathsf{R}(T_{j-1}(h))$ and its associated pointer P (which points to the position of its $(j-1)$-suffix in $B_{j-1}(h)$) from $\mathsf{R}(P_{j-1}(h))$), each S is then placed into one of σ distinct output files $T_j(1), \ldots, T_j(\sigma)$ according to the value of $S[k-j]$. Once all the sequences are processed, reading these files in the order $T_j(1) \ldots T_j(\sigma)$ forms the j-suffix ordering of the collection S that is needed for the next iteration.

The key observation here is that since the strings of S are presented in $(j-1)$-suffix order, so also must be the subset whose $(j-1)$-suffixes share a common first symbol c_h. Thus we use $\mathsf{R}_1(B_{j-1}(h))$ to count the number of occurrences of each symbol seen so far in $B_{j-1}(h)$, keeping track of how far into $B_{j-1}(h)$ we have read so far. We then read forward to the position pointed to by P, updating the counts as we go. Since the strings are processed in $(j-1)$-suffix order, we never need to backtrack.

Having determined where to put the new BWT symbol $S[k-j-1]$ in $B_{j-1}(z)$, where $c_z = S[k-j]$, we use $\mathsf{R}_2(B_{j-1}(z))$ to read up to that position, then write those symbols plus the appended $S[k-j-1]$ to $\mathsf{W}(B_j(z))$. All strings S' whose symbols need to be inserted into $B_{j-1}(z)$ arrive in $(j-1)$-suffix order and also satisfy $S'[k-j] = c_z$. They are therefore j-suffix ordered so, again, we never need to backtrack.

Finally, we must write to $\mathsf{W}(P_j(z))$ the entry that corresponds to S. To do this, we need to keep count of the number of additional symbols that have so far been inserted between the symbols from $B_{j-1}(z)$ and sent to $\mathsf{W}(B_j(z))$. This provides an offset that must be added to the number of symbols read from $\mathsf{R}_2(B_{j-1}(z))$ so far to create the value we need.

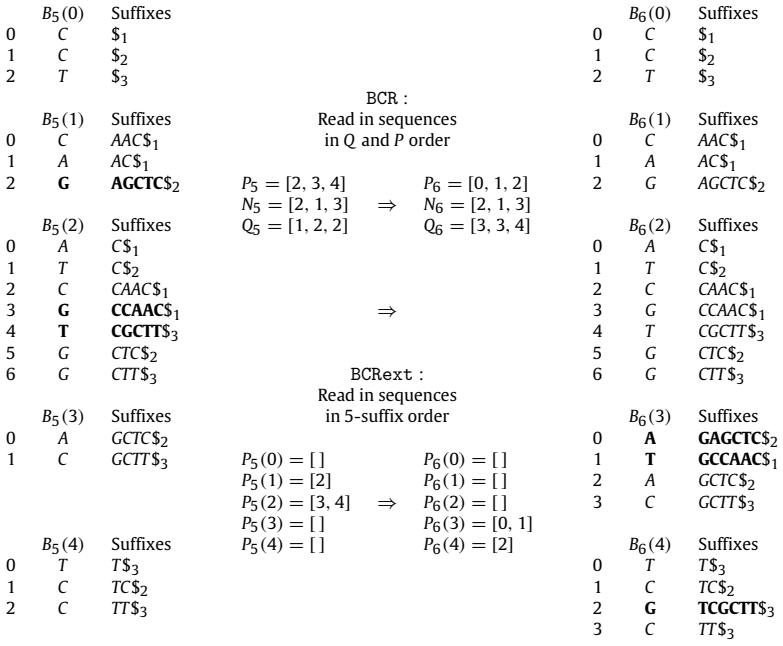

Fig. 2. Iteration 6 of the computation of the BWT of the collection $\mathsf{S} = \{TGCCAAC, AGAGCTC, GTCGCTT\}$ on the alphabet $\{A, C, G, T\}$. The two columns represent the partial BWT before and after the iteration and, in between, we see how the auxiliary data stored in the algorithm changes during the iteration. The positions of the new symbols corresponding to the 6-suffixes (shown in bold on the right) are computed from the positions of the 5-suffixes (in bold on the left), which were retained in the arrays P after the previous iteration. For clarity, we give distinct subscripts to the end markers of each of the sequences in the collection.

Once the last element of $T_{j-1}(\sigma)$ has been read, we update the cumulative count values to reflect any symbols not yet read from each $\mathsf{R}_1(B_{j-1}(h))$ and send any symbols not yet read from $\mathsf{R}_2(B_{j-1}(h))$ to $\mathsf{W}(B_j(h))$.

Fig. 2 uses a simple example to illustrate how the data structures associated with both variants of the algorithm are updated during an iteration.

The following proposition shows two important properties connecting the characters of L and F, where $L = \mathsf{bwt}(\mathsf{S}) = B(0) \cdots B(\sigma)$ and F is the sequence of the sorted characters of L.

Proposition 3.2. *Let S be a collection of strings and let $\mathsf{bwt}(\mathsf{S}) = L$. Let F be the sequence of the sorted characters of L. The following properties hold:*

1. *For every p, $0 \le p \le mk - 1$, such that $L[p] \neq \$_i$, for some $i = 1, \ldots, m$, the character $L[p]$ is followed by the character $F[p]$ in word S_i in S.*
2. *For a fixed character $a \in \Sigma$, its occurrences in F appear in the same order as in L, i.e. its r-th instance of a in F corresponds to its r-th instance of a in L.*

The proof follows the same arguments of the proof of Proposition 2.1 (see [3]).

Remark 3.3. We observe that the symbol $L[p] = \$_i$, for some $0 \le p \le mk-1$ and $i = 1, \ldots, m$, is followed by the character $F[p]$ in S_i by considering the strings in circular way. Hence $F[p]$ corresponds to $S_i[0]$.

Example 3.4. Let $\mathsf{S} = \{TGCCAAC, AGAGCTC, GTCGCTT\}$ be a collection of strings on the alphabet $\{A, C, G, T\}$, the BWT of S is the concatenation of the $\sigma + 1$ BWT segments after the iteration 7, that is

$$\mathsf{bwt}(\mathsf{S}) = CCTCA\$_2GATCGTGGATAC\$_3TCG\$_1C.$$

Remark 3.5. We note that, when the collection S contains only one element, the output of our transformation is the same output as that obtained by applying the original BWT to that element.

Remark 3.6. We can add new strings to an existing BWT of a collection S. We simply use Lemma 3.1 to add the symbols of the new string and obtain the BWT of $\mathsf{S} \cup \{S\}$, where S is a new string. There is no need to construct the BWT from scratch.

4. The decode transform

In this section, we show that our transformation is reversible, in that we can recover the original collection S from $\mathsf{bwt}(\mathsf{S})$. We give two algorithms to do this that keep the $\mathsf{bwt}(\mathsf{S})$ in external memory. The two algorithms differ in the direction in

which the strings of the collection are built. The first builds all the strings of S at the same time from left to right, while the second builds the strings of S at the same time from right to left.

Before describing our external memory based algorithm to recover the members of the collection, we describe a naive method that is interesting from a combinatorial point of view and that can be used to delete the symbols of one or more strings of the collection S from bwt(S) without the need to recompute the BWT from scratch.

Given F and $L = \text{bwt}(\mathtt{S})$, we can define a permutation θ on $\{0, \ldots, mk-1\}$ as follows: $\theta(p) = q$ if $F[p]$ and $L[q]$ correspond to the same character in a word of S, according to Item 2 of Proposition 3.2.

Remark 4.1. We observe that Proposition 3.2 allows us to determine all suffixes that come from a given string in the collection, and from Item 2 of Proposition 3.2 it follows that this permutation relates all and only the positions where the suffixes of the same string appear in the list of associated suffixes. In other words, the permutation θ can be decomposed into as many cycles as there are strings in the collection S. This fact is of fundamental relevance for the invertibility of our transform.

The following theorem proves that our transformation is injective. The proof is similar to the proof of Theorem 13 in [16].

Theorem 4.2. *The transformation BWT that associates to a collection $\mathtt{S}$ the bwt($\mathtt{S}$) is injective.*

Proof. Let S and T be two collections such that $\text{bwt}(S) = \text{bwt}(T)$ holds true. Since $L(S) = L(T)$, it follows that $F(S) = F(T)$. This implies that the permutations θ_S and θ_T, which we obtain by associating each character of $L(S)$ and $L(T)$ with the same occurrence of that character in $F(S)$ and $F(T)$, are equal. We recall also the well known combinatorial property that for any permutation there exists a unique decomposition into disjoint cycles: $\theta_S = \theta_T = \sigma_1\sigma_2 \cdots \sigma_m$. From item 1 of Proposition 3.2, as noticed in Remark 4.1, one can derive that m is equal to the cardinality of the collection; collections S and T have the same number of strings. Beyond that, each σ_i corresponds to a suffix of a string in S and in T. We can reconstruct the strings of S as described in the following. All strings S_i, for $i = 1, \ldots, m$, are primitive, and therefore a unique index p exists such that $L[i_p] = \$_i$, that is moved by σ_i. Let $i_1, i_2, \ldots, i_m$ be the indices where $L[i_1] = \$_i$, $L[i_2] = \$_2, \ldots, L[i_m] = \$_m$. So, one can reconstruct each string in S by using Item 2 of Proposition 3.2:

$$
\begin{aligned}
u_1 &= F[i_1]F[\theta_S(i_1)]F[\theta_S^2(i_1)] \cdots F[\theta_S^{l_1}(i_1)],\\
u_2 &= F[i_2]F[\theta_S(i_2)]F[\theta_S^2(i_2)] \cdots F[\theta_S^{l_2}(i_2)],\\
&\cdots\\
u_m &= F[i_m]F[\theta_S(i_m)]F[\theta_S^2(i_m)] \cdots F[\theta_S^{l_m}(i_m)],
\end{aligned}
$$

where $l_1, l_2, \ldots, l_m$ are the lengths of the cycles $\sigma_1, \sigma_2, \ldots, \sigma_m$, respectively.

Since T is also a collection of primitive strings, $\theta_T = \theta_S$ and $L(T) = L(S)$, we can easily deduce that $T = S$. □

Algorithm `decodeForwardByPermutation` recovers the original collection S given bwt(S).

```
Algorithm decodeForwardByPermutation (bwt(S));
1. Create F by alphabetically sorting the characters of L =bwt(S);
2. Build the permutation θ;
3. S := ∅; N := length(L);
4. for p = 0, ..., N − 1 do
5.   if L[p] = $ then
6.       u := ϵ; {ϵ is the empty word}
7.       i := p;
8.       repeat
9.           u := u + F[i]; {+ is concatenation between strings}
10.          i := θ(i);
11.      until F[i] = $;
12.      S := S ∪ {u}; {u is the i-th string in the collection S}
13. Return S;
```

In the following example, we show how the reconstruction works for the collection of Example 3.4.

Example 4.3. Let $\text{bwt}(\mathtt{S}) = L = CCTCA\$_2GATCGTGGATAC\$_3TCG\$_1C$ be the BWT of the collection $\mathtt{S} = \{TGCCAAC, AGAGCTC, GTCGCTT\}$. One can obtain $F = \$_1\$_2\$_3AAAACCCCCCCGGGGGTTTTT$ by alphabetically sorting the characters of L. The permutation θ from F and L is the following:

$$
\theta = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 & 17 & 18 & 19 & 20 & 21 & 22 & 23 \\ 22 & 5 & 18 & 4 & 7 & 14 & 16 & 0 & 1 & 3 & 9 & 17 & 20 & 23 & 6 & 10 & 12 & 13 & 21 & 2 & 8 & 11 & 15 & 19 \end{pmatrix},
$$

yielding the following decomposition into disjoint cycles:

$$\theta = (22\ 15\ 10\ 9\ 3\ 4\ 7\ 0)\ (5\ 14\ 6\ 16\ 12\ 20\ 8\ 1)\ (18\ 21\ 11\ 17\ 13\ 23\ 19\ 2).$$

By starting from indices 22, 5, 18 we have $L[22] = \$_1$, $L[5] = \$_2$, $L[18] = \$_3$, we reconstruct the original strings:

$$S_1 = F[22]F[15]F[10]F[9]F[3]F[4]F[7]F[0] = TGCCAAC\$_1,$$
$$S_2 = F[5]F[14]F[6]F[16]F[12]F[20]F[8]F[1] = AGAGCTC\$_2,$$
$$S_3 = F[18]F[21]F[11]F[17]F[13]F[23]F[19]F[2] = GTCGCTT\$_3.$$

The following proposition shows that, since the cycles are disjoint, the properties in Proposition 3.2 still hold after the removal of one or more cycles. The new permutation is the transform of the original collection minus the strings associated with the deleted cycles.

Proposition 4.4. *Given a collection $\mathcal{S} = S_1, \ldots, S_m$ of strings and bwt($\mathcal{S}$), one can obtain the BWT of $\mathcal{S} \setminus \{S_i\}$, for $i = 1, \ldots, m$, without the need to construct the BWT from scratch.*

Proof. By hypothesis, we know that, given bwt(S), we can define the permutation θ on $\{0, \ldots, mk-1\}$ according to item 2 of Proposition 3.2 and we know that θ can be decomposed into disjoint cycle $\theta = \sigma_1\sigma_2 \cdots \sigma_m$. We suppose that cycle σ_p, for some $p = 1, \ldots, m$, corresponds to S_i; the positions defined by cycle σ_p correspond to the symbols of the string S_i, so we can delete them from bwt(S). As the cycles in θ are disjoint, the properties of Proposition 3.2 still hold, and so we obtain bwt(S $\setminus$ $\{S_i\}$). □

The cost of `decodeForwardByPermutation` is proportional to the length of the reconstructed string plus the time to build the permutation. The main bottleneck is the workspace, so the computation could be expensive in practice.

In the following we present an alternative way to recover the strings of the collection that keeps bwt(S) in external memory and does not need to compute the permutation in advance. We suppose that the collection S has m strings of length k and that bwt(S) is split in σ BWT segments $B(h)$ for $h = 0, \ldots, \sigma$, where $c_0 = \$$ and $c_h \in \Sigma$ for $h = 1, \ldots, \sigma$. We observe that $B(h)$ for $h = 0, \ldots, \sigma$ are the BWT segments concatenated together to obtain the output of the encode transform. This observation is not restrictive, because one can count the number of occurrences of each symbol of the alphabet in the bwt(S), and hence split the entire bwt(S) into σ segments: each $B(h)$ segment contains a number of symbols equal to the number of occurrences of suffixes starting with c_h. We define $j = 1, 2, \ldots, k$ and $i = 1, 2, \ldots, m$.

Remark 4.5. From item 1 of Proposition 3.2, we know that if the symbol c_z is at position s in bwt(S) $= B(0) \cdots B(\sigma)$, then c_z is followed by the symbol at position s in F. By using σ BWT segments, we do not need to build F by alphabetically sorting the characters of $B(0) \cdots B(\sigma)$ in order to find the symbol that follows c_z. Indeed, if c_z is at the (absolute) position s in bwt(S) then c_z is at the (relative) position r of some BWT segment. We suppose that c_z is at position r in $B(v)$, for some $v = 0, \ldots, \sigma$. Since $B(v)$ contains all symbols associated with suffixes starting with the symbol c_v, it means that c_v is the first symbol of the suffix associated with c_z, so c_v follows the symbol c_z in the string that we want to recover. For sake of simplicity, we can think of F as being split into σ segments $F(0) \cdots F(\sigma)$ similar to bwt(S).

There are two possible ways to decode the BWT: either *forwards* to construct the original strings of the collection or *backwards* to retrieve the reverse strings (or, equivalently, the strings built from right to left) of the collection. Our algorithms compute the strings of S incrementally via k iterations. At each iteration, the algorithms add a new symbol for each string $S_i \in$ S, for $i = 1, \ldots, m$. We obtain the $(j+1)$-prefixes for all strings $S_i \in S$ if we decode in forward direction; we get the $(j+1)$-suffixes if we decode in the backward direction.

The fundamental observation is that going either from j-prefixes to $(j+1)$-prefixes or from j-suffixes to $(j+1)$-suffixes requires only that we find m new symbols and does not affect the relative order of the symbols already inserted in the prefixes or the suffixes.

First, we will describe the decode transform in forward order, by analogy to `decodeForwardByPermutation`.

In the following description, the function $(v, r) = select(c_z, s, L)$ gives the pair (v, r), where r is the position of the s-th occurrence of the symbol c_z in $B(v)$, for some $v = 0, \ldots, \sigma$. The counts for $B(h)$, $h = 0, \ldots, \sigma$, are dealt with by keeping a count of the number of occurrences of each symbol for all $B(h)$ in memory, which takes $O(\sigma^2 \log(mk))$ bits of space. This way v is easy to find and we only need to read $B(v)$ to find the position r.

Remark 4.6. The idea in each iteration $j = 1, \ldots, k$, according to Proposition 3.2 and Remark 4.5, is to obtain the symbol $S_i[j]$, for each $i = 1, \ldots, m$ by using the following two steps for each string:

1. Since c_z is the s-th occurrence of c_z in F, we have to find the s-th occurrence of c_z in L. We can obtain this by using the select function $(c_v, r) = select(c_z, s, L)$.
2. Since c_z is at the position r in $B(v)$, the symbol that follows c_z corresponds to c_v, so $S_i[j] = c_v$.

A trivial 'iteration 0' sets the symbols $S_i[0]$ for each string belonging to S. From Remark 3.3 we know that $S_i[0]$ is the symbol that follows the symbol $\$_i$; we only need to know in which $B(v)$, for $v = 0, \ldots, \sigma$, we will find $\$_i$. Clearly, $S_i[0]$ corresponds to the symbol c_v. Because of the primitivity of strings in the collection S (they become primitive by adding the end marker symbol \$), there exists a unique index i_i such that $L[i_i] = \$_i$, for each $i = 1, \ldots, m$.

Finally, iteration k inserts m end markers into $S_i[k]$, for each $i = 1, \ldots, m$. This denotes that we have build the entire strings of the collection.

During the iteration $j = 1, \ldots, k$, we need to keep track of the positions within F of the symbols that we have found at the step $j-1$. We keep an array $P^F_{j-1}(h)$ of integers for each $F(h)$ that stores the positions of the last symbols found in $F(h)$. Each $P^F_{j-1}(h)$ is in turn associated with an array $N^F_{j-1}(h)$ that has the same number of entries and is such that $N^F_{j-1}(h)[q]$ stores i, the original position in S of the string S_i whose symbol found is pointed to by $P^F_{j-1}(h)[q]$. Here q is a generic subscript of the array $N^F_{j-1}(h)$ or (equivalently, since their number of entries is the same) $P^F_{j-1}(h)$. The maximum value of q is determined by the number of the symbols equal to c_h found at the iteration $j-1$.

Stated formally, at the start of iteration j we assume the following structures are available for each $h = 0, \ldots, \sigma$, where $c_0 = \$$ and $c_n \in \Sigma$, for $n = 1, \ldots, \sigma$. The maximum value of q depends on the number of the symbols found, at the present iteration, equal to c_h:

$N^F_{j-1}(h)$ is an array of integers such that $N^F_{j-1}(h)[q]$ is associated with the last symbol $S_i[j-1]$ found from the string $S_i \in \mathsf{S}$, where $i = N^F_{j-1}(h)[q]$.
$P^F_{j-1}(h)$ is an array of integers such that $P^F_{j-1}(h)[q]$ is the position of the last symbol found $S_i[j-1]$ in $F(h)$, where $i = N^F_{j-1}(h)[q]$.

Hence, at the start of the iteration j, for each element in N^F_{j-1} and P^F_{j-1}, symbol $c_z = S_i[j-1]$ found at the previous step, with $i = N^F_{j-1}(z)[q]$, is at the position $P^F_{j-1}(z)[q]$ in $F(z)$.

During the iteration j, we have to update these structures for each string $S_i \in \mathsf{S}$. The crucial point is to insert the new symbol c_v in $S_i[j]$ where c_v is obtained as outlined in Remark 4.6 with $(c_v, P^F_j(v)[q]) = \mathit{select}(c_z, P^F_{j-1}(z)[q], L)$. Fig. 3 shows an example.

Now, we will describe the decode transform in backward order. This transform is useful mainly to implement the backward search.

Remark 4.7. The aim of each iteration $j = 1, \ldots, k-1$, according to Proposition 3.2 and Remark 4.5, is to obtain the symbol $S_i[k-j-1]$, for $i = 1, \ldots, m$ by using the following two steps for each sequence:

1. Since c_z is at position s in L, it means that c_z is at position r of $B(v)$, for $v = 0, \ldots, \sigma$. We have to find the position of c_z in F. We can obtain this by using the rank function: $r' = \mathit{rank}(c_z, r, c_v)$, where we recall that $\mathsf{rank}(c_z, r, c_v)$ denotes the number of occurrences of c_z in $B(0) \cdots B(v-1)B(v)[0, r-1]$ (note that the BWT segments are constant for all iterations in the decode transform). So c_z is the r' occurrence of c_z in L which means that c_z in $B(v)$ corresponds to the r'-th occurrence of c_z in $F(z)$.
2. Since c_z is at position r' in $F(z)$, symbol $S[k-j-1]$ that preceding c_z corresponds to the symbol at the position r' in $B(z)$.

A trivial 'iteration 0' sets the symbols $S_i[k-1]$ for each string belonging to S, i.e. it finds the symbols that precede each end marker. By definition we have $\$ < c_h$ for each $c_h \in \Sigma$, so the end markers $\$$ are at the first m positions in F. The symbols that precede each end marker are at the first m positions in L, in particular they are in $B(0)$. So, $S_i[k-1] = B(0)[i-1]$ for each $i = 1, \ldots, m$.

Finally, iteration k inserts m end markers into $S_i[k]$, for $i = 1, \ldots, m$. This denotes that we have build the entire strings of the collection.

During iteration $j = 1, \ldots, k$ we need to keep track of the positions of the symbols that we retrieved at step $j-1$, which we do by keeping an array $P^L_{j-1}(h)$ for each $B(h)$ that stores the positions of the last symbols found in L in corresponding to the BWT segment $B(h)$. Each $P^L_{j-1}(h)$ is in turn associated with an array $N^L_{j-1}(h)$ that has the same number of entries and is such that $N^L_{j-1}(h)[q]$ stores i, the original position in S of the string S_i whose symbol found is pointed to by $P^L_{j-1}(h)[q]$. Here q is a generic subscript of the array $N^L_{j-1}(h)$ or (equivalently, since their number of entries is the same) $P^L_{j-1}(h)$. The maximum value of q is determined by the number of the symbols found (at the iteration $j-1$) in $B(h)$.

Stated formally, at the start of iteration j, we assume the following structures are available for each $h = 0, \ldots, \sigma$, where $c_0 = \$$ and $c_n \in \Sigma$, for $n = 1, \ldots, \sigma$. The maximum value of q depends on the number of the symbols found in $B(h)$ at the current iteration:

$N^L_{j-1}(h)$ is an array of integers such that $N^L_{j-1}(h)[q]$ is associated with the last symbol found $S_i[k-(j-1)-1]$ of the string $S_i \in \mathsf{S}$ where $i = N^L_{j-1}(h)[q]$.
$P^L_{j-1}(h)$ is an array of integers such that $P^L_{j-1}(h)[q]$ is the position of the last symbol found $S_i[k-(j-1)-1]$ in $B(h)$, where $i = N^L_{j-1}(h)[q]$.

Hence, for each element in N^L_{j-1} and P^L_{j-1} at the start of the iteration j, the character $c_z = S_i[k-(j-1)-1]$ with $i = N^L_{j-1}(v)[q]$, is at the position $P^L_{j-1}(v)[q]$ in $B(v)$ for $v = 0, \ldots, \sigma$. During iteration j, we have to update these structures for each string $S_i \in \mathsf{S}$. The crucial point is to find the new symbols $S_i[k-j-1]$, for $i = 1, \ldots, m$ by using the steps of Remark 4.7.

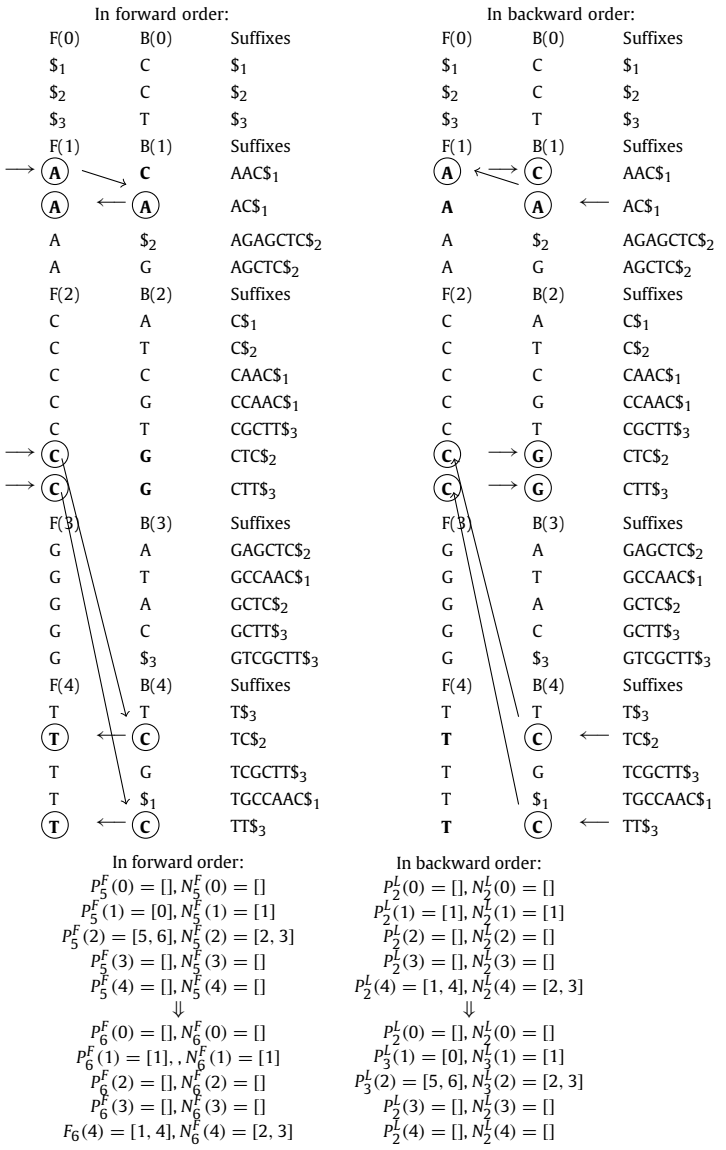

Fig. 3. This example shows the string collection from Example 3.4. On the left, we show iteration 6 of the decode transform in forward order. For instance, we know from the iteration 5 that the symbol C of S_2 is at the position 5 in $F(1)$, by using the select function. This C corresponds to symbol C at the position 1 in $B(4)$, so the new symbol that follows C is T at the position 1 in $F(4)$. This yields the prefix $AGAGCT$ of S_2. In a similar way, we obtain the prefixes $TGCCAA$ of S_1 and $GTCGCT$ of S_3. On the right, we show iteration 3 of the decode transform in backward order. From iteration 2 and by using the rank function we know that symbol C of S_2 is at position 1 in $B(4)$. This symbol corresponds to the C at position 5 in $F(2)$, so the next symbol that precedes C is G at position 5 in $B(2)$. This yields the suffix CTC of S_2. We obtain the suffixes AAC of S_1 and CTT of S_3 correspondingly.

For each element in N_{j-1}^L and P_{j-1}^L we first use the rank function $r' = rank(c_z, P_{j-1}^L(v)[q], c_v)$, where $i = N_{j-1}^L(v)[q]$ and $c_z = S_i[k-(j-1)-1]$. So c_z is the r'-th occurrence of c_z in $F(z)$ and it is preceded by the symbol at the position r' in $B(z)$. This symbol corresponds to $S_i[k-j-1]$ and $P_j^L(v)[q] = r'$. Fig. 3 shows an example.

Recall that the BWT is stored in $\sigma + 1$ external files, each of which is read at most once during each iteration of both the forward and backward decode transforms. For the backward decode transform, the positions $P^L(h)$, where $h = 0, \ldots, \sigma$, are sorted before computing the rank function, so the BWT files can be read sequentially, similar to the encode transform. This algorithm uses $O(m \log(mk))$ bits of workspace and $O(k\,\mathrm{sort}(m))$ of time, where $\mathrm{sort}(m)$ is the time needed to sort P_j^L and N_j^L.

In the case of the forward decode transform, we do not know the positions we need in the files until we compute the select function. This means each file is still opened once, but random accesses are made within it. We speed up the computation of the select function by logically dividing each $B(h)$ into s blocks and storing counts of the occurrences of each symbol in each block. It takes $O(s\sigma \log(mk))$ bits of space for each segment. With this optimization, as m is less than $s\sigma$ the algorithm uses

$O(s\sigma^2 \log(mk))$ bits of space. We note that we do not need to sort P_j^F and N_j^F, since an element in P_j^F denotes the number of occurrence of the symbol whose position we want to find. So, for each sequence we only need to find the block containing the occurrence of the symbol that we want to find, read that block in the external file and compute the position. This takes $O(s)$ for each sequence to find the block inside a BWT segment plus the time required to read the relevant block in the BWT segment from disk. If we suppose that the maximum block size is less than the number of blocks, then the algorithm takes $O(ms)$ time.

In this section we described algorithms to efficiently compute the BWT of a collection of strings and retrieve the original collection given bwt(S). We can use the algorithms described in this section to efficiently add and delete single strings without rebuilding the entire BWT from scratch.

5. Computational experiments

We tested our BWT construction algorithms on subsets of a collection of one billion human DNA sequences, each one 100 bases long, sequenced from a well-studied African male individual [2] (available from the Sequence Read Archive [8] using the accession number ERA015743[1]). All tests were carried out on one of two identical machines, each having 16Gbytes memory and two Intel Xeon X5450 (Quad-core) 3GHz processors (we only used one processor for our tests). Each machine was directly connected to its own array of 146Gbytes SAS disks in RAID6 configuration, each array having a Hewlett-Packard P6000 RAID controller with 512Mbytes cache. We had exclusive access to both test machines and their associated disk arrays for the duration of our experiments.

For smaller input instances, we compared our programs BCR and BCRext to bwte from the bwtdisk toolkit (version 0.9.0[2]), which implements the blockwise BWT construction algorithm described in [5]. Since bwte constructs the BWT of a single string, we concatenated our string collections into this form using 0 as a delimiter, choosing 0 because it is lexicographically smaller than any A, C, G, T, or N. An entirely like-for-like comparison would use a different end marker for each string and we note that the actual algorithm described in [5] can be extended to deal with string collections, however the bwte implementation does not support the many millions of distinct end marker symbols this would require.

Intuitively, the BWT is more work to compute for a string of size km than for a collection of m strings of length k, since the number of symbol comparisons needed to decide the order of two suffixes is not bounded by k. In our particular case, however, the periodic nature of the concatenated string means that 99 out of 100 suffix/suffix comparisons will still terminate within 100 symbols, because one suffix will hit 0 but the other will not, the only exception being the case where both suffixes start at the same position in different strings. The problem bwte is being asked to solve is therefore of comparable complexity to ours. We ran bwte using 4Gbytes and 14Gbytes of memory (denoted as bwte-4G and bwte-14G) to assess the impact of the amount of memory used on the performance of the program.

We also constructed the compressed suffix array (CSA) on smaller instances using Sirén's program rlcsa [25][3]. On those input instances rlcsa poses an interesting alternative, especially since this algorithm is geared toward indexing text collections as well. We split the input data into 10 batches, constructing a separate index for each batch then merging the indexes afterward. With increasing data volumes, however, the computational requirements for constructing the CSA become prohibitive on our testing environment. In [25, Section 6], 8 threads and up to 36 − 37Gbytes of memory are used to construct the CSA of a text collection 41.48Gbytes in size, although we note the author describes other variants of the algorithm that would use less RAM than this. In addition, we ran rlcsa with default settings, changing the parameters might affect the RAM usage as well.

Table 2 gives the results for all the input instances that we generated. The first two (0043M and 0085M) were sized to match the largest datasets considered in [5]. We created the latter three to show the effectiveness of our approach on very large string collections. rlcsa and bwte show efficiency (defined as user CPU time plus system CPU time as a fraction of wallclock time) approaching 100% in all our experiments (not only on our test machines, but on other servers as well), whereas both BCR and BCRext exhibit a drop in efficiency for large datasets. Even so, BCR and BCRext are both able to process the 1000M dataset at a rate that exceeds the performance of bwte on the 0085M dataset, which is less than one tenth of the size. This efficiency loss, which we believe is due to the internal cache of the disk controller becoming saturated, starts to manifest itself on smaller datasets for BCRext than for BCR, which is likely to be a consequence of the greater I/O demands of BCRext . Since the I/O of BCRext is dominated by the repeated copying of the input sequences during the radix sort, we modified BCRext to minimize the data read to and from disk during this activity.

In initial experiments with the zlib[4] library, the CPU overhead of on-the-fly compression and decompression of the input sequences to and from gzip format more than outweighed any possible efficiency gain that could arise from the reduced file sizes. We had more success by using a 4-bits-per-base encoding and by observing that, during a given iteration, we do not need to copy the entire input sequences but only the prefixes that still remain to be sorted in future iterations. The resulting new version BCRext++ was otherwise identical to BCRext but reduced the processing time for the 1 billion read

[1] Available at ftp://ftp.sra.ebi.ac.uk/vol1/ERA015/ERA015743/srf/.

[2] Available at http://people.unipmn.it/manzini/bwtdisk/.

[3] Available at http://www.cs.helsinki.fi/group/suds/rlcsa/rlcsa.tgz, version tested was downloaded on 8th December 2010.

[4] Available at http://www.zlib.net.

Table 2
The input string collections were generated on an Illumina GAIIx sequencer, all reads are 100 bases long. We chose the first two instances to have datasets comparable in size to the largest ones tested in [5]. Size is the input size in gigabytes, wall clock time—the amount of time that elapsed from the start to the completion of the instance—is given as microseconds per input base, and memory denotes the maximal amount of memory (in gigabytes) used during execution. BCRext and BCRext++ need to store only a constant and (for the DNA alphabet) negligibly small number of integers in RAM regardless of the size of the input data, we therefore state a –. The efficiency column gives the CPU efficiency values, *i.e.* the proportion of time for which the CPU was occupied and not waiting for I/O operations to finish, as taken from the output of the `/usr/bin/time` command. Some of the tests were repeated on a solid-state hard drive (SSD), the results from these are shown last. For all tests, the best wall clock time achieved is marked in bold.

Instance	Size	Program	Wall clock	Efficiency	Memory
0043M	4.00	bwte-4G	5.00	0.99	4.00
	4.00	bwte-14G	2.64	0.94	14.00
	4.00	rlcsa	2.21	0.99	7.10
	4.00	BCR	0.99	0.84	0.57
	4.00	BCRext	2.15	0.58	–
	4.00	BCRext++	**0.93**	0.66	–
0085M	8.00	bwte-4G	7.99	0.99	4.00
	8.00	bwte-14G	3.84	0.94	14.00
	8.00	rlcsa	2.44	0.99	13.40
	8.00	BCR	1.01	0.83	1.10
	8.00	BCRext	4.75	0.27	–
	8.00	BCRext++	**0.95**	0.69	–
0100M	9.31	BCR	**1.05**	0.81	1.35
	9.31	BCRext	4.6	0.28	–
	9.31	BCRext++	1.16	0.61	–
0800M	74.51	BCR	**2.25**	0.46	10.40
	74.51	BCRext	5.61	0.22	–
	74.51	BCRext++	2.85	0.29	–
1000M	93.13	BCR	5.74	0.19	13.00
	93.13	BCRext	5.89	0.21	–
	93.13	BCRext++	**3.17**	0.26	–
0085M (SSD)	8.00	bwte	8.11	0.99	4.00
	8.00	rlcsa	2.48	0.99	13.40
	8.00	BCR	0.78	0.99	1.10
	8.00	BCRext	0.89	0.99	–
	8.00	BCRext++	**0.58**	0.99	–
1000M (SSD)	93.13	BCR	**0.98**	0.91	13.00
	93.13	BCRext	2.26	0.53	–
	93.13	BCRext++	1.24	0.64	–

dataset by 47%, with even greater gains on the smaller datasets. We note that it is straightforward to add the I/O footprint optimizations implemented in BCRext++ also to BCR which would decrease wall clock time as well.

To see how performance scales with respect to sequence length, we concatenated pairs of sequences from our collection of 100 million 100-mers to create a set of 50 million 200-mers. While this collection contains the same number of bases, BCRext and BCRext++ both needed a similar proportion of additional time to create the BWT (69% and 67% respectively), whereas the time needed by BCR was only 29% more than was required for the original collection. The likely explanation for the difference is that the radix sort performed by BCRext and BCRext++ requires twice as much I/O for the 200-mer dataset than for the original collection.

To look further at the relationship between disk hardware and efficiency, we also performed some tests on a machine whose CPU was identical to those used for our previous tests but that was also equipped with a solid state hard drive (SSD)[5]. Since both `rlcsa` and `bwte` already operate at close to maximum efficiency, we would not expect the run time of these programs to benefit from the faster disk access speed of the SSD and their performance when the 0085M dataset was stored on the SSD was in line with this expectation. However the SSD greatly improved the efficiency of our algorithms, reducing the run times of BCRext and BCRext++ on the 1000M dataset by more than 5-fold and 2-fold respectively, meaning that the BWT of 1 billion 100-mers was created in just over 27 hours using BCR, or 34.5 hours with BCRext++.

We took the two instances containing 43 and 85 million reads to assess the practical performance of the decode algorithm described in Section 4. We constructed the BWT (encode) and constructed the original string collection given the BWT (decode). Table 3 shows the results for the encode and decode operations.

[5] We used an OCZ Technology R2 p88 Z-Drive with 1Tbyte capacity and a claimed maximum data transfer rate of 1.4Gbytes per second.

Table 3
Results for encoding and decoding the 43 and 85 million read instances. Size is the input size in gigabytes, wall clock time—the amount of time that elapsed from the start to the completion of the instance—is given as microseconds per input base, and memory denotes the maximal amount of memory (in gigabytes) used during execution. The efficiency column gives the CPU efficiency values, *i.e.* the proportion of time for which the CPU was occupied and not waiting for I/O operations to finish, as taken from the output of the `/usr/bin/time` command.

Instance	Size	Program	Wall clock	Efficiency	Memory
0043M	4.00	BCR encode	0.98	0.90	0.58
	4.00	BCR decode	0.75	0.98	0.68
0085M	8.00	BCR encode	1.03	0.89	1.00
	8.00	BCR decode	0.77	0.98	1.30

6. Discussion

The algorithms given in Section 3 allow the BWT of the large string collections encountered in DNA sequencing experiments to be computed in an acceptably short time on hardware that is relatively modest. In particular, RAM requirements are low.

The reliance on external memory does mean that attention must be paid to the specification and configuration of disk systems and that, as we saw in Section 5, significant performance gains can be achieved by compression techniques that reduce the amount of disk access required without introducing an excessive amount of additional CPU time to perform the compression. We continue to optimize the code in this way: a recent modification of `BCRext++` that stores the partial BWT strings in run-length encoded form was able to generate the BWT of the 1 billion read dataset we used in our experiments in under 20 hours on the SSD drive mentioned at the end of Section 5.

Our other ongoing work focuses on applications of our algorithms to the analysis of DNA sequences, but we note our methods may well have uses in other fields such as computational linguistics.

In Section 4 we give practical algorithms for computing the decode transform and thus recovering the all the strings from the collection given their BWT. These algorithms can be used to know whether or not a string belongs to the set of factors of the collection. We can change, for instance, the decode transform in reverse order to implement backward search of more factors or sequences at the same time.

One area we want to explore in the future is to parallelize both the encode and decode transforms. Our experiments show that one of the main handles to improve practical performance of our algorithms is to optimize disk I/O. Table 2 shows that CPU efficiency is the main factor impacting on wall clock time. Therefore, it may be more important to divide work across multiple hard drives than across multiple CPUs to increase efficiency and reduce wall clock time accordingly.

Acknowledgement

The third author wishes to thank Illumina Cambridge Ltd for financial support.

References

[1] M.J. Bauer, A.J. Cox, G. Rosone, Lightweight BWT construction for very large string collections, in: CPM 2011, in: LNCS, vol. 6661, Springer, 2011, pp. 219–231.
[2] D.R. Bentley, et al., Accurate whole human genome sequencing using reversible terminator chemistry, Nature 456 (7218) (2008) 53–59.
[3] M. Burrows, D.J. Wheeler, A block sorting data compression algorithm, Technical report, DIGITAL System Research Center, 1994.
[4] A.J. Cox, O.B. Schulz-Trieglaff, I. Khrebtukova, M.J. Bauer, E.P. Murchison, Z. Ning, M. Hims, S. Luo, D.J. Evers, Poster Abstract: Hypothesis-free detection of splice junctions in RNA-Seq data, in: Proceedings of CSHL conference on Genome Informatics, 2011.
[5] P. Ferragina, T. Gagie, G. Manzini, Lightweight data indexing and compression in external memory, in: LATIN, in: LNCS, vol. 6034, Springer, 2010, pp. 697–710.
[6] P. Ferragina, G. Manzini, Opportunistic data structures with applications, in: Proceedings of the 41st Annual Symposium on Foundations of Computer Science, IEEE Computer Society, Washington, DC, USA, 2000, pp. 390–398.
[7] P. Ferragina, G. Manzini, Indexing compressed text, J. ACM 52 (2005) 552–581.
[8] National Center for Biotechnology Information. Sequence Read Archive. http://trace.ncbi.nlm.nih.gov/Traces/sra/sra.cgi?.
[9] G.H. Gonnet, R.A. Baeza-Yates, T. Snider, New Indices for Text: PAT Trees and PAT Arrays, Prentice-Hall, Inc, Upper Saddle River, NJ, USA, 1992, pp. 66–82.
[10] W.K. Hon, T.W. Lam, K. Sadakane, W.K. Sung, S.M. Yiu, A space and time efficient algorithm for constructing compressed suffix arrays, Algorithmica 48 (2007) 23–36.
[11] J. Kärkkäinen, Fast BWT in small space by blockwise suffix sorting, Theor. Comput. Sci. 387 (2007) 249–257.
[12] J. Kärkkäinen, P. Sanders, S. Burkhardt, Linear work suffix array construction, J. ACM 53 (2006) 918–936.
[13] D. Kim, J. Sim, H. Park, K. Park, Linear-time construction of suffix arrays, in: CPM 2003, in: LNCS, vol. 2676, Springer-Verlag, Berlin, Heidelberg, 2003, pp. 186–199.
[14] P. Ko, S. Aluru, Space efficient linear time construction of suffix arrays, J. Discrete Algorithms 3 (2–4) (2005) 143–156.
[15] S. Mantaci, A. Restivo, G. Rosone, M. Sciortino, An extension of the Burrows Wheeler transform and applications to sequence comparison and data compression, in: CPM 2005, in: LNCS, vol. 3537, 2005, pp. 178–189.
[16] S. Mantaci, A. Restivo, G. Rosone, M. Sciortino, An extension of the Burrows–Wheeler transform, Theor. Comput. Sci. 387 (3) (2007) 298–312.
[17] S. Mantaci, A. Restivo, G. Rosone, M. Sciortino, A new combinatorial approach to sequence comparison, Theory Comput. Syst. 42 (3) (2008) 411–429.
[18] M.L. Metzker, Sequencing technologies — the next generation, Nature Reviews Genetics 11 (1) (2009) 31–46.

[19] G. Nong, S. Zhang, W.H. Chan, Linear time suffix array construction using d-critical substrings, in: CPM 2009, in: LNCS, Springer-Verlag, Berlin, Heidelberg, 2009, pp. 54–67.
[20] P.A. Pevzner, H. Tang, M.S. Waterman, A new approach to fragment assembly in DNA sequencing, in: RECOMB, 2001, pp. 256–267.
[21] S.J. Puglisi, W.F. Smyth, A.H. Turpin, A taxonomy of suffix array construction algorithms, ACM Comput. Surv. 39 (2007).
[22] B.P. Walenz, R.A. Lippert, C.M. Mobarry, A space-efficient construction of the Burrows–Wheeler transform for genomic data, J. Comput. Biol. 12 (7) (2005) 943–951.
[23] J.T. Simpson, R. Durbin, Efficient construction of an assembly string graph using the FM-index, Bioinformatics 26 (12) (2010) i367–i373.
[24] J.T. Simpson, R. Durbin, Efficient de novo assembly of large genomes using compressed data structures, Genome Research, December 2011 (in press).
[25] J. Sirén, Compressed suffix arrays for massive data, in: SPIRE, in: LNCS, vol. 5721, Springer, 2009, pp. 63–74.