Matching Statistics Speed up BWT Construction

Francesco Masillo
Department of Computer Science, University of Verona, Italy

Abstract

Due to the exponential growth of genomic data, constructing dedicated data structures has become the principal bottleneck in common bioinformatics applications. In particular, the Burrows-Wheeler Transform (BWT) is the basis of some of the most popular self-indexes for genomic data, due to its known favourable behaviour on repetitive data.

Some tools that exploit the intrinsic repetitiveness of biological data have risen in popularity, due to their speed and low space consumption. We introduce a new algorithm for computing the BWT, which takes advantage of the redundancy of the data through a compressed version of matching statistics, the *CMS* of [Lipták et al., WABI 2022]. We show that it suffices to sort a small subset of suffixes, lowering both computation time and space. Our result is due to a new insight which links the so-called insert-heads of [Lipták et al., WABI 2022] to the well-known run boundaries of the BWT.

We give two implementations of our algorithm, called CMS-BWT, both competitive in our experimental validation on highly repetitive real-life datasets. In most cases, they outperform other tools w.r.t. running time, trading off a higher memory footprint, which, however, is still considerably smaller than the total size of the input data.

2012 ACM Subject Classification Theory of computation → Design and analysis of algorithms

Keywords and phrases Burrows-Wheeler Transform, matching statistics, string collections, compressed representation, data structures, efficient algorithms

Digital Object Identifier 10.4230/LIPIcs.ESA.2023.83

Supplementary Material *Software (Source Code)*: `https://github.com/fmasillo/CMS-BWT`

Acknowledgements I want to thank Sara Giuliani for listening and discussing the preliminary ideas contained in this paper. I also want to thank Zsuzsanna Lipták for giving helpful feedback during the writing of this paper.

1 Introduction

The Burrows-Wheeler Transform (BWT) [6] is a reversible permutation of the characters of the input text that can be computed in linear time and space. It is closely related to the suffix array, a permutation of the indices of T which is based on the lexicographic order of the suffixes. It is known that the BWT, when applied to repetitive texts, results in an easier-to-compress string, especially when using simple run-length encoding.

Another virtue of the BWT is that it can be used as a self-index to replace the original text. It can support pattern matching queries, more specifically, it counts how many times a pattern P occurs in T using time proportional to $|P|$. It can be incorporated into more elaborate indexes [11, 13, 14, 27] to support locating queries (finding the positions in T where P occurs) and more complex queries such as finding Maximal Exact Matches (MEMs) and Maximal Unique Matches (MUMs). MEMs and MUMs are of key importance, especially in the field of bioinformatics, where they are used for read alignment (e.g. MUMmer [23]). Widely used tools such as BowTie2 [18] and BWA [20] are based on the BWT for aligning short reads to a reference genome.

Nowadays, the amount of biological data that is publicly available is too big for most BWT construction algorithms. A key observation is that, even though the size of these datasets is massive, the information contained within them is highly redundant. This is why

31st Annual European Symposium on Algorithms (ESA 2023).
Editors: Inge Li Gørtz, Martin Farach-Colton, Simon J. Puglisi, and Grzegorz Herman; Article No. 83; pp. 83:1–83:15
LIPICS Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

tools such as `big-BWT` [5], `r-pfBWT` [26] and `grlBWT` [10] have emerged. These tools exploit the intrinsic repetitiveness of the input data to build large BWTs fast and in compressed space.

Recently, Lipták et al. [22] devised a variant of matching statistics [8] called *compressed matching statistics* (*CMS*). This data structure has been proven to be effective in expressing the redundancy of sets of highly similar strings and being used in the process of suffix sorting. Cunial et al. [9] work with a *compact* representation of matching statistics, i.e. a bitvector of length $2|P|$ storing the differentially encoded lengths of the matches. They also present several ways of compressing the bitvector based on both lossless and lossy methods. This representation is fundamentally different from that of [22] due to the fact that the *CMS*, and more prominently its enhanced version, packs more information than just the length of the matches. This additional information can then be used for other applications such as suffix sorting.

In this paper, we are going to show that the *CMS* of [22] not only can be used to build the generalized suffix array of a collection of strings but can also be highly useful in building large BWTs, due to the fact that it uses considerably less space than the input data. Experimental results show that our implementation, `CMS-BWT`, is competitive, if not better, than the state-of-the-art tools for constructing the BWT, although heavier on the space consumption side.

Our algorithm is based on a new insight that allows us to find the run boundaries of the BWT within special buckets of suffixes, which are closely connected to the fundamental element of the compressed matching statistics, the so-called *insert-heads*.

The paper is organized as follows. In Section 2, we give definitions and notations used in the remainder of the paper. Section 3 contains an overview of the compressed matching statistics. In Section 4, we present our contribution, describing the algorithm used for constructing the BWT. In Section 5, we describe details of our implementation and then report experimental results in Section 6. Finally, in Section 7, conclusions and future work are discussed.

2 Basics

Let Σ be an ordered alphabet of size σ. A string T over Σ is a finite sequence of characters from Σ. The ith character of T is denoted $T[i]$, its length is $|T| = n$, and $T[i..j]$ denotes the substring $T[i] \cdots T[j]$. If $i > j$, then $T[i..j]$ is the empty string ε. The suffix $T[i..] = T[i..n]$ is referred to as the i^{th} suffix $\mathit{suf}_i(T)$, and $T[..i] = T[1..i]$ is the i^{th} prefix $\mathit{pref}_i(T)$. When T is clear from the context, we write suf_i for $\mathit{suf}_i(T)$.

We assume that the last character of T is the sentinel character \$. It is set to be smaller than any other character in Σ and appears only once as the end-of-string character.

The *suffix array* SA of a string T is a permutation of the set $\{1, \ldots, n\}$ such that $SA[i] = j$ if $\mathit{suf}_j(T)$ is the ith in lexicographic order among all suffixes. Numerous suffix array construction algorithms (SACAs) exist in the literature [25, 24, 1, 21, 15]. SA-IS [25] is by far the most popular linear time SACA, being both simple and fast in practice.

The *inverse suffix array* ISA is the inverse permutation of SA, namely $ISA[SA[i]] = i$.

The *longest common prefix* (*lcp*) of a pair of strings T and S is the longest string U which is the prefix of both T and S. The *longest-common-prefix array* LCP is another array closely related to the SA. It is given by: $LCP[1] = 0$, and for $i > 1$, $LCP[i]$ is the length of the longest common prefix of the two suffixes $\mathit{suf}_{SA[i-1]}$ and $\mathit{suf}_{SA[i]}$. This array can be computed in linear time, too [17].

The Burrows-Wheeler Transform BWT [6] is a reversible permutation of the input text T. It is defined as $\mathrm{BWT}[i] = \$$ if $SA[i] = 1$, and $\mathrm{BWT}[i] = T[SA[i] - 1]$ otherwise.

Let R and S be two strings. The *matching statistics* MS of S w.r.t. R is an array of length $|S|$ in which every entry is a pair of integers defined as follows. Fix i, let U_i be the longest prefix of suffix $suf_i(S)$ which occurs as a substring in R. Then, entry $MS[i] = (p_i, \ell_i)$, where p_i is an occurrence of U_i in R, or -1 if $U_i = \varepsilon$, and $\ell_i = |U_i|$. We will call U_i *matching factor* and the character $c_i = S[i + \ell_i]$ will be referred to as *mismatch character* of position i. We set the end-of-string character of R to be smaller than the one of S ($\# < \$$).

For an integer array A of length n and an index i, the previous and next smaller values are defined as follows: $PSV(A, i) = \max\{i' < i \; : \; A[i'] < A[i]\}$, $NSV(A, i) = \min\{i' > i \; : \; A[i'] < A[i]\}$. The minimum of the empty set is $-\infty$ and the maximum is $+\infty$. There exists a data structure of size $n \log(3 + 2\sqrt{2}) + o(n)$ bits that can be built in $\mathcal{O}(n)$ time and answers both PSV and NSV queries in constant time [12].

Given a set of integers X and an integer x, the predecessor of x is the largest element in X less than or equal to x. In other words, $pred_X(x) = \max\{y \in X \; : \; y \leq x\}$. Predecessor queries can be answered in $\mathcal{O}(\log\log |X|)$ time using the y-fast trie data structure of Willard [29] which uses $\mathcal{O}(|X|)$ space.

Let $\mathcal{C} = \{S_1, S_2, \ldots, S_m\}$ be a collection of strings not necessarily distinct, i.e. $\mathcal{C}$ is a multiset. The total length of $\mathcal{C}$ will be denoted by N, where we use end-of-string characters to delimit the strings, i.e. $N = \sum_{d=1}^{m} |S_d| + m$. From now on, we will treat $\mathcal{C}$ as this concatenated string, slightly abusing notation.

Our problem is defined as follows:

Problem Statement: Given a string collection $\mathcal{C} = \{S_1, \ldots, S_m\}$ and a reference string R, compute the Burrows-Wheeler Transform BWT of $\mathcal{C}$.

The end-of-string character $\#$ of R is assumed to be smaller than any of $\mathcal{C}$. Moreover, in our setting, we assume that each $S_i \in \mathcal{C}$ is highly similar to R.

3 Compressed Matching Statistics

Recently, the authors of [22] introduced a new data structure called Compressed Matching Statistics (CMS). This data structure exploits the redundancy of plain MS, where we have the following property: if $\ell_i > 0$, then $\ell_{i+1} \geq \ell_i - 1$. We can identify sequences of the form $(x, x-1, x-2, \ldots)$ where $x = \ell_i$, called *decrement runs*. A decrement run ends when $\ell_j > \ell_{j-1} - 1$, and j is the starting position of a head. For an example see Figure 1.

▶ **Definition 1** (Compressed matching statistics, [22]). *Let R, S be two strings over Σ, and MS be the matching statistics of S w.r.t. R. The* compressed matching statistics *(CMS) of S* w.r.t. *R is a data structure storing $(j, MS[j])$ for each head j, and a predecessor data structure on the set of heads H.*

It was shown in [22] that it is possible to recover each individual value $MS[i]$ for any i using the following formula: $MS[i] = (p_i + k, \ell_i - k)$, where $j = pred_H(i)$ and $k = j - i$. This can be done in $\mathcal{O}(\log\log \chi)$ time and $\mathcal{O}(\chi)$ space, where $\chi = |H|$.

It was shown in [22] that storing the matching statistics information only for heads leads to a compression ratio of up to 100 times on real-life data.

3.1 Enhanced Compressed Matching Statistics

In [22], the *CMS* was refined with additional information to get the *enhanced compressed matching statistics* (*eCMS*). Assuming that all characters occurring in S also occur in R at least once, the information of p_i can be made more specific, namely, one can compute the position that a suffix from S would have if it was present in SA_R the SA of R. This position is called *insert point* of i:

$$ip(i) = \begin{cases} 1 & \text{if } U_i = \varepsilon, \\ \max\{j \mid U_i \text{ is a prefix of } R[SA_R[j]..] \text{ and } R[SA_R[j]..] < U_i c\} & \text{if this set is non-empty,} \\ \min\{j \mid U_i \text{ is a prefix of } R[SA_R[j]..]\} & \text{otherwise.} \end{cases}$$

The first case is satisfied only for the end-of-string characters of the collection $\mathcal{C}$, because the sentinel character of R is smaller than any other character ($\# < \$$). In the other two cases, the insert point is the lexicographic rank of suffix i among all suffixes of R. Suffix i ideally points to the next smaller occurrence of U_i in R, if it exists (case 2). Otherwise, it coincides with the smallest occurrence of U_i in R (case 3).

For the *eCMS*, the positions for which the *MS* information is saved are called *insert-heads* and are defined as follows: j is an insert-head if $SA_R[ip(j)] \neq SA_R[ip(j-1)] + 1$. Some additional information is also stored in each insert-head: c_i, the mismatching character, and x_i, a boolean value associated with c_i. This value is set to be smaller (S = 0) if $c_i < R[SA[ip(i)] + \ell_i]$ or larger (L = 1) otherwise. Referring to the definition of *ip*, $x_i = 1$ whenever we are in case 2, $x_i = 0$ when we are in case 3.

▶ **Definition 2** (Enhanced compressed matching statistics, [22]). *Let R, S be two strings over Σ. Define the* enhanced matching statistics *of S w.r.t. R as follows: for $1 \leq i \leq |S|$, let $ems(i) = (q_i, \ell_i, x_i, c_i)$, where $q_i = SA_R[ip(i)]$, ℓ_i is the length of the matching factor U of i, c_i is the mismatch character, and $x_i \in \{S, L\}$ indicates whether $U_i c_i$ is smaller (S) or greater (L) than $R[q_i..]$. The* enhanced compressed matching statistics (*eCMS*) of S w.r.t. R *is a data structure storing $(j, ems(j))$ for each insert-head j, and a predecessor data structure on the set of insert-heads K.*

The size of K is denoted by $|K| = \kappa$. The time for recovering $MS[i]$ becomes $\mathcal{O}(\log\log\kappa)$, while the space becomes $\mathcal{O}(\kappa)$ [22].

By definition, the number of insert-heads is larger than the number of heads. Although in [22] the difference in numbers is noticeable, the compression effect is still very strong. For actual numbers see Section 6.2, more specifically Table 1.

For an example of *eCMS* refer to Figure 1.

3.2 Comparing two suffixes using eCMS

The additional information of insert-heads helps bucketing suffixes with respect to the insert point. We will call these buckets *insert-buckets*. Assessing the order of any two suffixes having different insert point has been proven in the following lemma:

▶ **Lemma 3** ([22]). *Let $1 \leq i, j \leq N$. If $ip(i) < ip(j)$, then $suf_i < suf_j$.*

On the other hand, when two suffixes belonging to the same insert-bucket are compared the following lemma refines the order:

▶ **Lemma 4** ([22]). *Let $1 \leq i, j \leq N$, and $ip(i) = ip(j)$.*

i	1	2	3	4	5	6	7	8	9	10	11	12
R	C	A	T	T	A	G	A	T	T	A	G	#
S	T	A	G	A	G	A	T	T	A	T	T	$
p_i	4	5	6	5	6	7	8	9	2	3	4	-1
ℓ_i	4	3	2	6	5	4	3	2	3	2	1	0
head	✓			✓					✓			
q_i	4	5	6	5	6	2	3	4	7	8	9	12
insert-head	✓			✓		✓			✓			✓
c_i	G			T		T			$			$
x_i	S			L		L			S			L

Figure 1 An example for matching statistics and corresponding *CMS* and *eCMS*. In rows 1 and 2, we report $MS[i] = (p_i, \ell_i)$ of S w.r.t. R. In row 3, we mark the starting positions of the heads (*CMS*). In row 4, for each index, we give the special position $q_i = SA_R[ip(i)]$, where $ip(i)$ is the insert point of suf_i in SA_R. In row 5, we mark insert-heads (*eCMS*). In the last two rows, we complete the information stored in insert-heads, namely the mismatch character c_i and x_i, the associated boolean value (S = smaller, L = larger).

1. *If $\ell_i < \ell_j$ and $x_i = S$, then $suf_i < suf_j$.*
2. *If $\ell_i < \ell_j$ and $x_i = L$, then $suf_j < suf_i$.*
3. *If $\ell_i = \ell_j$ and $x_i = S$ and $x_j = L$, then $suf_i < suf_j$.*
4. *If $\ell_i = \ell_j$ and $x_i = x_j$ and $c_i < c_j$, then $suf_i < suf_j$.*

To achieve the final correct order of two suffixes having the same insert-head information, in [22] it was suggested sorting only the insert-heads. Then, using the new rank for each head, the total order of two suffixes can be established.

If two arbitrary suffixes from $\mathcal{C}$ are being compared, one needs to perform two predecessor queries to get the insert-head of each suffix. This implies that the time spent for a single comparison is $\mathcal{O}(\log\log\kappa)$. As we will see in Section 4, we can avoid the predecessor queries when scanning the collection left to right, resulting in constant time comparisons. This is because we will only perform comparisons of suffixes of one insert-head at a time with other insert-heads of the same insert-bucket.

3.3 Computing the eCMS

We will use the procedure outlined in [22].

The data structures needed to compute the *eCMS* of $\mathcal{C}$ w.r.t. R are the suffix array SA_R, the inverse suffix array ISA_R, the *LCP*-array LCP_R, and the *RMQ* data structure for *PSV-NSV* queries on LCP_R. Every data structure can be constructed in $\mathcal{O}(|R|)$ time and space.

This procedure takes $\mathcal{O}(N\log|R|)$ time and $\mathcal{O}(|R|)$ space and outputs the set of insert-heads of size $\mathcal{O}(\kappa)$.

To speed up the practical running time, we will use also the following proposed heuristic of [22]. Because we work with highly similar strings, it is common to have a singleton interval (an interval of size one) after the failure of a sequence of right extensions. A key insight is that also after the subsequent left contraction, the interval remains of size one. This means that the matching factor U_i lies within a leaf branch in a hypothetical suffix tree of R. In order to detect these cases, we can compare ℓ_i to the maximum value in LCP_R. If $\ell_i - 1 > \max(LCP_R)$, then it means that there is no other suffix in R with a prefix equal to U_i. This means that we are in a leaf branch. Computing the left contraction is now equal to accessing $ISA[p_i + 1]$. This bypasses the PSV and NSV queries on LCP_R, avoiding the corresponding cache misses. Computing a single maximum can be too restricting for some datasets, so a refinement of this strategy is to divide LCP_R into blocks and compute a maximum value for each of them.

The practical speedup can be of an order of magnitude when using this last strategy on sets of highly repetitive strings, as it was shown in [22].

4 Computing the BWT with enhanced Compressed Matching Statistics

In this section, we are going to outline the procedure used to compute the BWT of $\mathcal{C}$ using only data structures built on R and the *eCMS* of $\mathcal{C}$ w.r.t. R.

We will use the following heuristic in order to speed up the computation of BWT($\mathcal{C}$): suffixes in text order between two insert-heads are preceded by the same character present in the reference. This can be intuitively explained by looking at the way *eCMS* is built: any position between two consecutive insert-heads is consecutive in text order both in R and $\mathcal{C}$. Therefore, by knowing the insert point of each suffix we know what its position is in the previously computed SA_R, and consequently which is the preceding character stored in BWT_R. This insight tells us that we just have to "expand" BWT_R based on the number of suffixes with the same insert-point while taking care of insert-heads. The suffixes corresponding to the starting positions of insert-heads are the only ones that need to be sorted inside each insert-bucket. See Figure 2 for an example.

▶ **Lemma 5.** *Let suf_i and suf_j be two suffixes of $\mathcal{C}$. If $ip(i) = ip(j)$ and the two suffixes are not the start of an insert-head, then suf_i and suf_j are preceded by the same character $c = R[SA[ip(i)] - 1]$, i.e. $\mathcal{C}[i-1] = \mathcal{C}[j-1] = c = R[SA[ip(i)] - 1]$.*

Proof. By assumption we know that $ip(i) = ip(j)$, therefore $SA[ip(i)] = SA[ip(j)]$. Because suf_i and suf_j are not the starting positions of any insert-head, it is true that $SA[ip(i-1)] = SA[ip(i)] - 1$ and $SA[ip(j-1)] = SA[ip(j)] - 1$. Therefore, $SA[ip(i-1)] = SA[ip(j-1)]$ and also $ip(i-1) = ip(j-1)$. Since $U_{i-1}, U_{j-1} \neq \varepsilon$ it follows that the first character of U_{i-1} and U_{j-1} is the same. ◀

While computing the *eCMS* of $\mathcal{C}$ w.r.t. R, we can simultaneously count how many suffixes fall in each insert-bucket. We recall that we can have at most $|R|$ insert-buckets, so the size of the array of counters called *bucket-counters* is $|R| \log N$ bits. By Lemma 5 we know that suffixes in the same insert-bucket have different preceding character only if one of them is an insert-head. By scanning again the collection, we just need to count how many suffixes belonging to the same insert-bucket come before each insert-head. In a sense, insert-heads work as run boundaries inside their insert-bucket, because they are preceded by a character that is different from the one preceding other non-insert-head suffixes. Therefore, we only need an additional counter for each insert-head to keep track of this quantity. We will store the counters in an array called *head-counters*. Inside a given insert-bucket we already know the total order of insert-heads, because we have sorted the whole set K after the computation of *eCMS*, as mentioned in Section 3.2.

Since we are scanning $\mathcal{C}$ left-to-right, we know $MS[i]$ for every suffix, without the need of using predecessor queries as we explain next. By saving the *eCMS* in text order, we start from the first insert-head k_1. Every suffix i before the starting position of k_2 have $MS[i] = (q_1 + (i - j_1), \ell_1 - (i - j_1))$, where $j_1 = 1$ is the starting position of k_1. Then, when

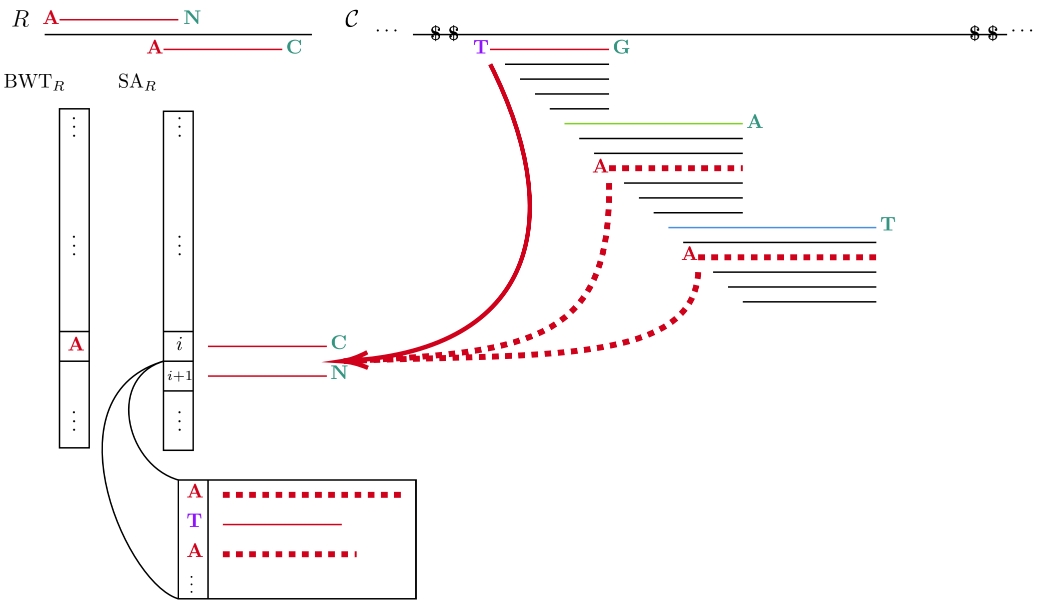


Figure 2 Example showcasing the proposed heuristic. Solid coloured lines under $\mathcal{C}$ are matching factors for insert-heads, while solid black lines are matching factors for suffixes inbetween insert-heads. Dotted red lines are for suffixes inbetween insert-heads that share the same insert-point as the first solid red line (an insert-head). Suffixes inbetween insert-heads share the same preceding character (red A) with the reference R, while the red-coloured insert-head is preceded by a different character (purple T). Zooming in on the insert-bucket, we see that it can happen that the purple T goes between the red As, breaking what would have been a run of only red As.

we reach the starting position of k_2, we just have to repeat this procedure until every couple of insert-heads has been processed. We will compare suffix i only with insert-heads stored in the corresponding insert-bucket, therefore we do not need to perform any predecessor query. Ultimately, the comparisons are made in constant time. Given suf_i, we can perform a binary search in the bucket corresponding to $ip(i)$ taking $\mathcal{O}(\log K_i)$ time, where K_i is the set of insert-heads in the bucket with $ip(i)$. After finding the correct index using Lemma 4 and, if necessary, resorting to the rank of the sorted insert-heads, we increment the counter for that insert-head. The array of *head-counters* takes $\kappa \log N$ bits of space.

Building the BWT of $\mathcal{C}$ is then just a matter of interleaving bucket-counters and head-counters. For $1 \leq i \leq |R|$, let $x = \text{bucket-counter}[i]$ be the number of suffixes in that bucket. If no insert-heads are present in the bucket, write $c = R[SA[i] - 1]$ in the output $\text{BWT}(\mathcal{C})$ x times. Otherwise, if at least one insert-head is in the bucket, for each head-counter in the current insert-bucket write c repeated as many times as indicated in the head-counter. Then, after the head-counter is processed, write the character that precedes the insert-head itself, namely $\mathcal{C}[j-1]$, where j is the starting position of the insert-head. Each time we write a character from either the head-counter or the head itself, we subtract one from x. If at the end of this procedure, x is not equal to 0, it means we still need to write that number of c characters in the output BWT. This is because this amount of suffixes was bigger than any insert-head in their insert-bucket.

The main procedure is outlined in Algorithm 1 and the running time and space consumption are reported in Proposition 6. A full example can be found in Figure 3.

▶ **Proposition 6.** *Given R and $\mathcal{C}$, we can compute $BWT(\mathcal{C})$ in $\mathcal{O}(N \log \kappa + N \log |R| + |R|)$ time and $\mathcal{O}(\kappa + |R|)$ space.*

```
                                1
     1  2  3  4  5  6  7  8  9  0  1  2
R    C  A  T  T  A  G  A  T  T  A  G  #
```

```
                                1                             2
     1  2  3  4  5  6  7  8  9  0  1  2  3  4  5  6  7  8  9  0  1  2  3  4
C    T  A  G  A  G  A  T  T  A  T  T  $1 G  A  T  T  A  C  A  T  T  A  G  $2
     ●        ●     ●        ●        ●  ●              ●                 ●
q    4  5  6  5  6  2  3  4  7  8  9  12 6  7  8  9  10 1  2  3  4  5  6  12
```

i	*eCMS*
1	(1, 4,4, G,S)
2	(4, 5,6, T,L)
3	(6, 2,4, T,L)
4	(9, 7,3,$\$_1$,S)
5	(12,12,0,$\$_1$,L)
6	(13, 6,5, C,S)
7	(18, 1,6,$\$_2$,S)
8	(24,12,0,$\$_2$,L)

i	BWT$_R[i]$	$SA_R[i]$	bucket-counter[i]
1	G	12	2
2	T	10	1
3	T	5	3
4	G	7	2
5	C	2	2
6	#	1	1
7	A	11	0
8	A	6	4
9	T	9	2
10	T	4	3
11	A	8	2
12	A	3	2

i	sorted *eCMS*	head-counter[i]	prev. char.
1	(12,12,0,$\$_1$,L)	0	T
2	(24,12,0,$\$_2$,L)	0	G
3	(4, 5,6, T,L)	2	G
4	(9, 7,3,$\$_1$,S)	0	T
5	(6, 2,4, T,L)	1	G
6	(18, 1,6,$\$_2$,S)	0	A
7	(13, 6,5, C,S)	2	$\$_2$
8	(1, 4,4, G,S)	1	$\$_1$

i	BWT$_\mathcal{C}[i]$
1	T
2	G
3	T
4	T
5	T
6	G
7	T
8	G
9	C
10	G
11	A
12	A
13	A
14	$\$_2$
15	A
16	T
17	T
18	T
19	$\$_1$
20	T
21	A
22	A
23	A
24	A

Figure 3 Example of the construction of BWT($\mathcal{C}$). The colour purple is used to indicate a relationship with insert-heads, whereas red is used to indicate a relationship with R and BWT$_R$. On the left, under $\mathcal{C}$ we mark insert-heads with a purple circle and highlight with the same colour q_j when j is an insert-head. In the middle part of the figure, entries of bucket-counter highlighted in red contains a positive number, meaning that no insert-head is contained within that bucket. On the other hand, entries coloured in purple tell us that we have at least one insert-head in that bucket. On the right, we show the full BWT of $\mathcal{C}$, where we use the same colour code. We also show with horizontal lines insert-buckets, highlighting how we interleaved information from bucket-counters and head-counters.

Proof. Computing all data structures for R can be done in linear time and space in $|R|$. Computing the *eCMS* of $\mathcal{C}$ takes $\mathcal{O}(N \log |R|)$ time and $\mathcal{O}(|R|)$ space using the approach described in Section 3.3. The computation of BWT($\mathcal{C}$) is bounded by the time of counting how many suffixes are smaller than each insert-head in a bucket with the same $ip(i)$. More specifically, $\sum_{1 \leq i \leq |R|} B_i \log K_i \leq |C| \log \kappa$, where B_i is the set of suffixes belonging to insert-bucket i and K_i the set of insert-heads within the same insert-bucket i. The space consumption is dominated by the number of insert-heads and the size of the data structures on R. ◀

Because we are working with highly similar strings, we expect to have few insert-heads, having long matches between any string of $\mathcal{C}$ and R. This makes the sorting part of insert-heads very fast in practice, due to κ being small. Also, the process of binary searching is conducted bucket by bucket, so the number of heads in the same bucket is expected to be smaller than κ.

Moreover, if the insert-heads are concentrated in a few insert-buckets we can entirely skip the computation for each bucket without insert-heads. More information on real-life datasets related to this insight can be found in Section 6.2.

5 Implementation details

The algorithm starts by first augmenting the reference with characters that occur in $\mathcal{C}$ but not in R so that we have a well-defined insert point for each suffix of the collection.

Algorithm 1 CMS-BWT.

```
Input: reference $R$, collection $\mathcal{C}$
Output: BWT($\mathcal{C}$)
compute $SA_R$, $ISA_R$, $LCP_R$, $PSV-NSV(LCP_R)$
bucket-counters $\leftarrow [0] * N$
$eCMS \leftarrow [\,]$
$p_{\text{prev}} \leftarrow -\infty$
for $i \leftarrow 1$ to $N$ do
    $\langle i, p_i, \ell_i, c_i, x_i \rangle \leftarrow$ computeMS($R, \mathcal{C}, i$)
    if $p_i \neq p_{prev} + 1$ then
        $eCMS$.add($\langle i, p_i, \ell_i, c_i, x_i \rangle$)
        mark bucket-counter[$p_i$]          // insert-bucket has an insert-head
        bucket-counter[$p_i$] $++$
        $p_{\text{prev}} \leftarrow p_i$
    else
        bucket-counter[$p_i$] $++$
    end
end
sort $eCMS$
head-counters $\leftarrow [0] * K$
for $i \leftarrow 1$ to $N$ do
    if bucket-counters[$p_i$] is marked then
        $j \leftarrow$ binary-search correct position of $suf_i$ in $K_{p_i}$
        head-counters[$j$] $++$
    end
end
for $i \leftarrow 1$ to $|R|$ do
    $x \leftarrow$ bucket-counters[$i$]
    if $i > 1$ then  $char \leftarrow R[SA_R[i-1]]$
    else  $char \leftarrow R[|R|]$
    if bucket-counter[$i$] is marked then
        for $j \in indices(K_i)$ do              // set of indices of $K_i$
            write $char$ head-counters[$j$] times
            write character preceding $j$th head
            $x = x -$ head-counters[$j$] $- 1$
        end
        if $x > 0$ then
            write $char$ $x$ times
        end
    else
        write $char$ bucket-counter[$i$] times
    end
end
```

Then, we use `libsais` [16] to build SA_R and LCP_R. We chose this tool because it has been experimentally proven to be one of the fastest tools for general-purpose suffix array construction. For the LCP array it uses the Φ method [17]. The data structure for PSV-NSV queries on the LCP_R is based on the work of Cánovas and Navarro [7].

For sorting the *eCMS*, we first rename each insert-head with a metacharacter based on the rank of the partial lexicographic order of the substrings associated with each insert-head. Then, by rearranging these metacharacters in text order we use again `libsais` to compute the suffix array of this metacharacter string.

When profiling the implementation, we found that the number of distinct insert-heads, i.e. the number of different tuples in K, grows even slower than the total set. For example, looking at the dataset consisting of 333 copies of Human Chromosome 19 described in Section 6.2, we have only 4,355,600 unique insert-heads versus $\kappa = 174,532,868$. Moreover, when performing binary search comparisons, more than 60% of the total number of comparisons were resolved by comparing the length plus x_i information stored in the *eCMS*. Combining these two insights led us to another heuristic based on a two-layered binary search. First, we compare the length information $\ell_i = \ell_{k_i} - (i - j)$ along with x_{k_i} of a suffix i with insert-head $pred_K(i) = k_i$ starting at position j with the set of unique insert-heads of its insert-bucket. Then, if the pair ℓ_i and x_{k_i} is different from any other insert-head we increment the counter for the insert-head pointed to by this first binary search. Otherwise, we have to refine the search by comparing suf_i with the whole set of insert-heads having $ip(i)$, $\ell = \ell_i$ and x_{k_i}. This technique led to a speedup in the binary search phase of between 10% and 20%.

To avoid continuous cache misses due to loading different subsets of insert-heads with different insert points during binary searching, we put a number of suffixes in a buffer divided into insert-buckets. After the buffer is at its full capacity, we proceed to process in bulk suffixes in the same insert-bucket, easing the loading in cache of subsets of insert-heads. For all of our experiments, we set this buffer to 2GB, but it can be arbitrarily chosen by the user.

Lastly, we also implemented a variant of `CMS-BWT` trading off space for running time. This was achieved by writing to disk some of the data structures involved in different phases of the algorithm. This version saves roughly a third of the space used by the non-memory-saving implementation.

6 Experiments

We implemented our algorithm for computing the BWT in C++. Our implementation, `CMS-BWT`, is available at `https://github.com/fmasillo/CMS-BWT`. The experiments were conducted on a desktop equipped with 64GB of RAM DDR4-3200MHz and an Intel(R) Core(R) i9-11900 @ 2.50GHz (with turbo speed @ 5GHz) with 16 MB of cache. The operating system was Ubuntu 22.04 LTS, the compiler used was `g++` version 11.3.0 with options `-std=c++20 -O3 -funroll-loops -march=native` enabled.

6.1 Tools compared

We compared two different implementations of `CMS-BWT` (simple and memory-saving) to the following four tools:

1. `big-BWT` [5], a tool which computes both the BWT and the suffix array. It is specifically made for highly repetitive data. We used the default parameters (`-w = 10`, `-p = 100`) and the `-f` flag to parse fasta files as input. We chose the default parameters in order to be consistent with the literature [5, 3, 4]. This tool outputs the entire BWT.

To double-check the choice of parameters, we performed further experiments on each combination of $w \in \{6, 8, 10\}$ and $p \in \{50, 100, 200, 400, 800\}$ as reported in [5]. On the basis of these experiments, we concluded that the default parameters $w = 10$ and $p = 100$ lead to the best combination of memory and time for the two datasets evaluated. It should be noted that the variation in memory and time between the different parameters is non-negligible, reaching a 7 times larger peak memory consumption and a 40% slowdown in running time for the `sars-cov2` dataset (data not shown).

2. `r-pfBWT` [26], is a tool improving on plain PFP. It has been shown to be both faster and to use less space than `big-BWT` for big enough dataset sizes. We run the experiments using `--bwt-only --w1 10 --w2 5` as flags. This tool outputs the run-length encoded BWT.
3. `grlBWT` [10], a tool computing the BCR BWT [2] again with a focus on highly repetitive data. We used the default parameters. This tool outputs the run-length encoded BWT.
4. `ropeBWT2` [19], a highly optimized tool to compute the BCR BWT on DNA data. We used the flag `-R` to skip the reverse complement. We also compare the effect of adding the `-P` flag, which limits the software to execute in single-threaded mode at all times. This tool outputs the entire BWT.

6.2 Datasets

In our experiments, we used two publicly available datasets. The first dataset, called `chr19` contains copies of the Human Chromosome 19 from the 1000 Genomes Project [28]. The second dataset, named `sars-cov2`, consists of copies of SARS-CoV2 genomes taken from COVID-19 Data Portal [1]. Some additional metadata can be found in Table 1.

The total size of both datasets is 60GB. We took increasing prefixes of size 1GB, 10GB, 20GB, 40GB, and 60GB. For the explicit number of sequences contained in each dataset see Table 2.

For example, looking at 20GB of `chr19` data, where we have around 333 copies of Human Chromosome 19, we have insert-heads only in 6% of the buckets. This means that around 94% of the suffixes will not be compared against any insert-head, speeding up the whole process.

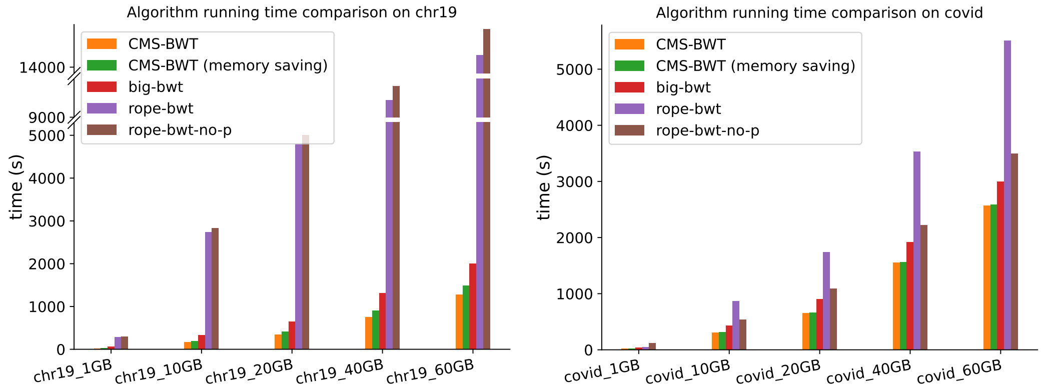
Table 1 Datasets used in experiments. In column 3, we specify the alphabet size σ, in column 4 the number r of runs of the BWT, in column 5 the number of insert-heads, and in column 6 the number of unique insert-heads. In our experiments, we use prefixes of each dataset up to 60GB. The last three columns refer to the 20GB prefix.

Name	Description	σ	r (20 GB)	no. of i-heads (20 GB)	no. unique i-heads (20 GB)
`chr19`	Human Chromosome 19	5	36 723 404	174 532 868	4 355 600
`sars-cov2`	SARS-CoV2 genome	14	19 075 277	253 188 521	1 466 183

[1] We used the following command to download in bulk the data using the CDP File Downloader: `java -jar cdp-file-downloader.jar - -domain=VIRAL_SEQUENCES - -datatype=SEQUENCES - -format=FASTA - -location=/home/data/ - -email=xxx@xxx.xx - -protocol=FTP`

Table 2 Number of sequences present in each dataset.

Name	1GB	10GB	20GB	40GB	60GB
chr19	17	167	333	666	1 000
sars-cov2	36 204	332 209	659 441	1 312 058	1 966 237



(a) Running time comparison on different subsets of copies of Chromosome 19.

(b) Running time comparison on different subsets of copies of SARS-CoV2 genomes.

Figure 4 Running time comparison of tools outputting the full BWT.

6.3 Results

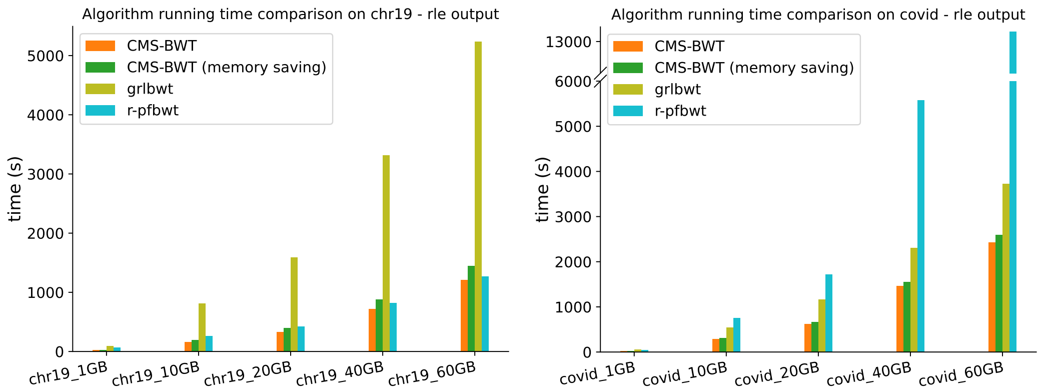
As already pointed out in Section 6.1, the output of the tools can either be the full BWT or the run-length encoded BWT. This can be a non-negligible time overhead. Therefore, when comparing to tools that output the whole BWT we will also write this version of the BWT to disk. On the other hand, when comparing CMS-BWT to r-pfBWT and grlBWT we will write to disk the run-length encoded BWT.

In Figures 4a, 4b, 5a and 5b, we report the comparison of the running time of the five tools divided by dataset and output type.

On the chr19 dataset, we are always the fastest tool compared to other tools that output the uncompressed BWT. More specifically, comparing the non-memory-saving implementation at 60GB of data, we are 57% faster than big-BWT and 10 times faster than ropeBWT2 with and without -P. Compared to the tools that output the run-length encoded BWT, our fastest implementation is always the winner, while at 60GB, our memory-saving implementation takes 12% more time than r-pfBWT. Both implementations outperform grlBWT, e.g. at 60GB of data they take a fourth of the time.

On the sars-cov2 dataset we are always the fastest tool in both settings. For example, at 60GB of data, we are faster than: big-BWT by 17%, ropeBWT2 with -P by 114%, ropeBWT2 with no -P by 35%, r-pfBWT by 445% and grlBWT by 53%.

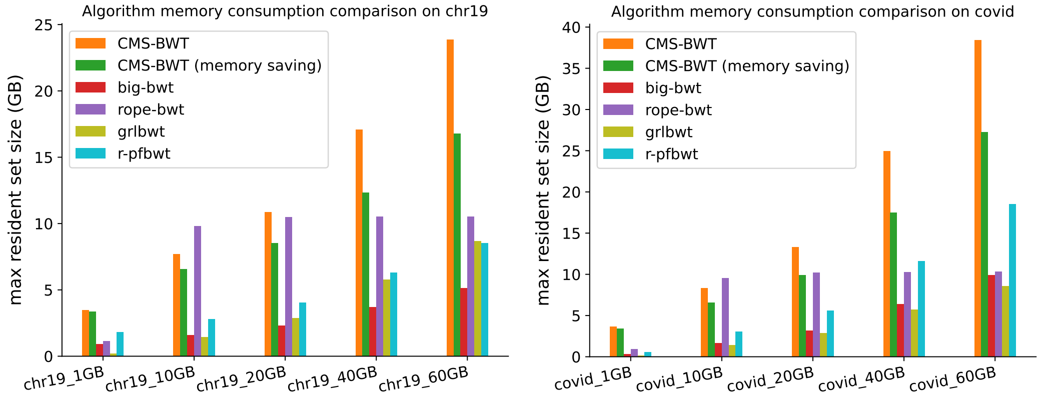
In Figure 6a and 6b we show the memory footprint of the five tools. As one can notice, our tool has the highest memory requirement. However, it can be noted that the memory-saving variant of CMS-BWT on bigger sizes of both datasets requires always less than half of the input size in space. On the sars-cov2 dataset we have a higher memory footprint than on chr19 because for the same size of the datasets we have a significantly higher number of strings in the collection, leading to more insert-heads.



(a) Running time comparison on different subsets of copies of Chromosome 19.

(b) Running time comparison on different subsets of copies of SARS-CoV2 genomes.

Figure 5 Running time comparison of tools outputting the run-length encoded BWT.



(a) Comparison of tools on different subsets of copies of Chromosome 19.

(b) Comparison of tools on different subsets of SARS-CoV2 genomes.

Figure 6 Peak memory measured as maximum resident set size in GB.

7 Conclusions

We presented a new algorithm for constructing the BWT of a collection of highly similar strings in compressed space. An experimental evaluation of two different implementations shows that our algorithm is competitive with state-of-the-art tools. Most of the time, both implementations outperform the other tools in terms of running time, but they are also the heaviest w.r.t. space consumption.

Future work will focus on parallelizing the implementation to allow taking advantage of multicore CPUs that are widespread nowadays. It is fairly straightforward to assign distinct sequences to a pool of multiple threads to compute the matching statistics. Another phase that would directly benefit from multi-threading is the for-loop at line 18 in Algorithm 1. With careful handling of locks for each head-counter, this is easily parallelizable, dividing the for-loop into equal parts. Moreover, we are going to investigate other ways of reducing memory consumption to close the gap between `CMS-BWT` and competing tools.

We are also working on extending our algorithm to incorporate the computation of SA-samples. This will allow us to build the r-index. With careful implementation, our tool can be extended to compute SA-samples along with bucket- and head-counters, without changing either the time or space bounds given in Proposition 6.

References

1 Uwe Baier. Linear-time suffix sorting - A new approach for suffix array construction. In *Proc. of the 27th Annual Symposium on Combinatorial Pattern Matching (CPM 2016)*, volume 54 of *LIPIcs*, pages 23:1–23:12. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016.

2 Markus J. Bauer, Anthony J. Cox, and Giovanna Rosone. Lightweight algorithms for constructing and inverting the BWT of string collections. *Theor. Comput. Sci.*, 483:134–148, 2013.

3 Christina Boucher, Davide Cenzato, Zsuzsanna Lipták, Massimiliano Rossi, and Marinella Sciortino. r-indexing the eBWT. In *Proc. of the 28th International Symposium on String Processing and Information Retrieval (SPIRE 2021)*, volume 12944 of *Lecture Notes in Computer Science*, pages 3–12. Springer, 2021.

4 Christina Boucher, Ondrej Cvacho, Travis Gagie, Jan Holub, Giovanni Manzini, Gonzalo Navarro, and Massimiliano Rossi. PFP compressed suffix trees. In *Proc. of the Symposium on Algorithm Engineering and Experiments (ALENEX 2021)*, pages 60–72. SIAM, 2021.

5 Christina Boucher, Travis Gagie, Alan Kuhnle, Ben Langmead, Giovanni Manzini, and Taher Mun. Prefix-free parsing for building big BWTs. *Algorithms Mol. Biol.*, 14(1):13:1–13:15, 2019.

6 Michael Burrows and David J. Wheeler. A block-sorting lossless data compression algorithm. Technical report, DIGITAL System Research Center, 1994.

7 Rodrigo Cánovas and Gonzalo Navarro. Practical compressed suffix trees. In *Proc. of the 9th International Symposium Experimental Algorithms, SEA 2010*, volume 6049 of *LNCS*, pages 94–105. Springer, 2010.

8 William I. Chang and Eugene L. Lawler. Sublinear approximate string matching and biological applications. *Algorithmica*, 12(4/5):327–344, 1994.

9 Fabio Cunial, Olgert Denas, and Djamal Belazzougui. Fast and compact matching statistics analytics. *Bioinform.*, 38(7):1838–1845, 2022.

10 Diego Díaz-Domínguez and Gonzalo Navarro. Efficient construction of the BWT for repetitive text using string compression. In *Proc. of 33rd Annual Symposium on Combinatorial Pattern Matching (CPM 2022)*, volume 223 of *LIPIcs*, pages 29:1–29:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022.

11 Paolo Ferragina and Giovanni Manzini. Opportunistic data structures with applications. In *Proc. of the 41st Annual Symposium on Foundations of Computer Science (FOCS 2000)*, pages 390–398. IEEE Computer Society, 2000.

12 Johannes Fischer. Combined data structure for previous- and next-smaller-values. *Theor. Comput. Sci.*, 412(22):2451–2456, 2011.

13 Travis Gagie, Gonzalo Navarro, and Nicola Prezza. Fully functional suffix trees and optimal text searching in BWT-runs bounded space. *J. ACM*, 67(1):2:1–2:54, 2020.

14 Sara Giuliani, Giuseppe Romana, and Massimiliano Rossi. Computing maximal unique matches with the r-index. In *Proc. of the 20th International Symposium on Experimental Algorithms (SEA 2022)*, volume 233 of *LIPIcs*, pages 22:1–22:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022.

15 Keisuke Goto. Optimal time and space construction of suffix arrays and LCP arrays for integer alphabets. In *Proc. of the Prague Stringology Conference 2019*, pages 111–125. Czech Technical University in Prague, Faculty of Information Technology, Department of Theoretical Computer Science, 2019.

16 Ilya Grebnov. Code for libsais. URL: `https://github.com/IlyaGrebnov/libsais`.

17 Juha Kärkkäinen, Giovanni Manzini, and Simon J. Puglisi. Permuted longest-common-prefix array. In *Proc. of the 20th Annual Symposium on Combinatorial Pattern Matching (CPM 2009)*, volume 5577 of *LNCS*, pages 181–192. Springer, 2009.

18 Ben Langmead and Steven L Salzberg. Fast gapped-read alignment with Bowtie 2. *Nature methods*, 9(4):357–359, 2012.

19 Heng Li. Fast construction of FM-index for long sequence reads. *Bioinform.*, 30(22):3274–3275, 2014.

20 Heng Li and Richard Durbin. Fast and accurate long-read alignment with Burrows-Wheeler transform. *Bioinform.*, 26(5):589–595, 2010.

21 Zhize Li, Jian Li, and Hongwei Huo. Optimal in-place suffix sorting. *Inf. Comput.*, 285(Part):104818, 2022.

22 Zsuzsanna Lipták, Francesco Masillo, and Simon J. Puglisi. Suffix sorting via matching statistics. In *Proc. of the 22nd International Workshop on Algorithms in Bioinformatics (WABI 2022)*, volume 242 of *LIPIcs*, pages 20:1–20:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022.

23 Guillaume Marçais, Arthur L. Delcher, Adam M. Phillippy, Rachel Coston, Steven L. Salzberg, and Aleksey V. Zimin. Mummer4: A fast and versatile genome alignment system. *PLoS Comput. Biol.*, 14(1), 2018.

24 Ge Nong. Practical linear-time $O(1)$-workspace suffix sorting for constant alphabets. *ACM Trans. Inf. Syst.*, 31(3):15, 2013.

25 Ge Nong, Sen Zhang, and Wai Hong Chan. Two efficient algorithms for linear time suffix array construction. *IEEE Trans. Computers*, 60(10):1471–1484, 2011.

26 Marco Oliva, Travis Gagie, and Christina Boucher. Recursive prefix-free parsing for building big BWTs. *bioRxiv*, pages 2023–01, 2023.

27 Massimiliano Rossi, Marco Oliva, Ben Langmead, Travis Gagie, and Christina Boucher. MONI: A pangenomic index for finding maximal exact matches. *J. Comput. Biol.*, 29(2):169–187, 2022.

28 The 1000 Genomes Project Consortium. A global reference for human genetic variation. *Nature*, 526:68–74, 2015.

29 Dan E. Willard. Log-logarithmic worst-case range queries are possible in space $\Theta(N)$. *Inf. Process. Lett.*, 17(2):81–84, 1983.