

The TAG array of a multiple sequence alignment

Jannik Olbrich 

Ulm University, Germany

Enno Ohlebusch 

Ulm University, Germany

Abstract

Modern genomic analyses increasingly rely on pangenomes, that is, representations of the genome of entire populations. The simplest representation of a pangenome is a set of individual genome sequences. Compared to e.g. sequence graphs, this has the advantage that efficient exact search via indexes based on the Burrows-Wheeler Transform (BWT) is possible, that no chimeric sequences are created, and that the results are not influenced by heuristics. However, such an index may report a match in thousands of positions even if these all correspond to the same locus, making downstream analysis unnecessarily expensive. For sufficiently similar sequences (e.g. human chromosomes), a multiple sequence alignment (MSA) can be computed. Since an MSA tends to group similar strings in the same columns, it is likely that a string occurring thousands of times in the pangenome can be described by very few columns in the MSA. We describe a method to tag entries in the BWT with the corresponding column in the MSA and develop an index that can map matches in the BWT to columns in the MSA in time proportional to the output. As a by-product, we can efficiently project a match to a designated reference genome, a capability that current pangenome aligners based on the BWT lack.

2012 ACM Subject Classification Theory of computation → Data structures design and analysis

Keywords and phrases Burrows-Wheeler Transform, pattern matching, index data structure, pangenomics

Acknowledgements We would like to thank Travis Gagie for suggesting this problem. Without him, this paper would not exist.

1 Introduction

Genomic analyses and diagnostics is often reference-based, that is, samples are compared with a reference genome of e.g. humans to speed up the analysis. For a long time, such a reference consisted of a single genome. However, a single reference sequence cannot capture the genetic diversity of a population. As a consequence, modern tools focus on representations that include (common) variations. Such a representation of the genomes of a population is commonly called a *pangenome*.¹ A common use case for pangenome representations is *read mapping*, where one seeks to determine the locus in the genome of a short substring of a DNA sample called a *read*, while accounting for sequencing errors and natural variation. Using pangenome representations can significantly reduce reference bias and thus mapping errors [8, 23, 26, 27, 35].

Pangenome representations are mostly based on *sequence graphs* (see e.g. [1, 2]). However, no widely agreed-upon metric for the quality of such graphs exists because several desirable qualities are at odds with each other [2]. For example, using the same subgraph to represent a variant that occurs in multiple sequences may lead to a smaller graph, but may also induce chimeric sequences (i.e., sequences that are valid in the graph but do not exist in the pangenome), e.g. by allowing the combination of variants that do not occur together in nature.

¹ The concept of a pangenome was originally developed for bacterial studies [38], but now can refer to the entire genomic variation of any population. In this paper, we focus on the pangenome of a species.

Additionally, under the strong exponential-time hypothesis (SETH) it is impossible to index a sequence graph in polynomial time such that string matching queries can be answered in sub-quadratic time [16]. For this reason, methods based on sequence graphs either resort to heuristic matching or cannot guarantee a good worst-case time complexity. In contrast, there are indexes for strings or sets of strings which achieve optimal construction and query time complexities [19]. While these data structures would require space proportional to the combined size of all haplotypes present in a sequence graph and thus be far less practical, almost as good time complexities can be achieved with indexes based on the run-length compressed Burrows-Wheeler Transform (BWT) [9] such as variants of the *r*-index [22] (see e.g. [4, 5, 12, 30]).

In combination with algorithms to construct these indexes for such huge datasets [6, 14, 27, 28, 31, 33], this opens the possibility to represent a population just by the set of sequences and not worry about e.g. graph indexing.

One problem of such a simple representation is that now a match may occur at thousands of positions in the index even when those occurrences all correspond to the same locus. However, in practice, it is often desirable to locate a match relative to a linear reference sequence. Recent works that use the BWT for pangenome indexing are for instance *ropebwt3* [27] and *Moni-align* [39]. The former aligns samples using the BWT-based index, while the latter merely uses the index to search for seeds (maximal exact matches between the pangenome and the pattern) and then aligns only to the most promising sequences in the pangenome.

Recently, we showed how to compute a *multiple sequence alignment (MSA)* of even very long sequences (such as human chromosomes), given that those sequences are very similar [32]. In an MSA, equal substrings that correspond the same locus are aligned. Therefore, we can identify the aforementioned thousands of positions in the index by just the corresponding column in the MSA. Note that it is trivial to directly map such a column in the MSA to a single reference sequence if that reference is part of the alignment. Tools such as *ropebwt3* “cannot project the alignment [of a pattern] to a designated reference genome” [27], so “it seems [...] interesting to know which column of the alignment a character in the BWT comes from” [21].

1.1 Our contributions

In this paper, we describe a space-efficient index that reports the columns of the alignment where a string occurs. In particular, we build an index that is able to quickly report the distinct columns in the MSA where a match occurs. To this end, we use the TAG array [3, 21], which lists the columns in the alignment for each suffix in lexicographic order. We first show how the run-length encoded TAG array can be built in linear time and small space, and describe a sampled index and use known document-listing techniques to enable reporting of the distinct TAG values in a BWT interval in optimal time. As an example application, we demonstrate that we can map multiple exact matches (MEMs) in the BWT-based index of *ropebwt3* to a linear reference sequence quickly while using small space.

We focus on pangenomes for very closely related genomes, e.g. those of humans, since an MSA cannot sensibly represent the variations otherwise.

1.2 Related work

A TAG array associates some data (a “tag”) with each BWT position. In our case, this tag is a column in the MSA, but there are other tags that result in a run-length compressible

TAG array: In [17], positions in the BWT are tagged with the corresponding coordinates in a sequence graph. Compared to our algorithms, theirs are more complex and also orders of magnitude more expensive to run in terms time and memory consumed, but it should be noted that it may arguably be the case that tagging BWT positions with graph positions is inherently harder than tagging them with columns of an MSA. In [13], positions in the BWT are tagged with metagenomic class identifiers, resulting in an index that can be used to perform accurate metagenomic read classification in small space.

The remainder of this paper is structured as follows: Section 2 introduces the definitions and notations used throughout this paper. Section 3 describes our algorithm for constructing the TAG array, and in Section 4 we describe a strategy and corresponding index for sampling only a fraction $\frac{1}{s}$ of the TAG runs. In Section 5 we experimentally evaluate our algorithms before Section 6 concludes the paper.

2 Preliminaries

For $i, j \in \mathbb{N}_0$ we denote the set $\{k \in \mathbb{N}_0 \mid i \leq k \leq j\}$ by the interval notations $[i..j] = [i..j+1) = (i-1..j] = (i-1..j+1)$. A *string* (or *array*) S of *length* n over an *alphabet* Σ is a sequence of n characters from Σ . We denote the length n of S by $|S|$ and the i th symbol of S by $S[i-1]$, i.e., strings and arrays are zero-indexed. The *substring* (or *subarray*) of S from i to j is denoted by $S[i..j] = S[i..j+1) = S(i-1..j] = S(i-1..j+1) = S[i]S[i+1] \dots S[j]$. For $i > j$, $S[i..j]$ is the *empty string* ϵ of length 0. A substring of the form $S[i..n)$ is a *suffix* of S and is denoted by $\text{suf}_i(S)$. A *bit vector* is an array over the binary alphabet $\{0, 1\}$.

A *multiple sequence alignment* (MSA) of a set of strings \mathcal{S} is obtained by inserting a number of *gap characters* ‘-’ into each string in \mathcal{S} such that the resulting strings all have the same length. An example MSA of the strings $\{\text{ACGACT}\$, \text{AAACT}\$, \text{ACGCAGT}\$\}$ is given in the top-left of Figure 1. A “good” MSA has few gap-symbols and columns mostly contain the same character. In this paper, we are not concerned with the precise optimization objective or methods for construction, so we refer the interested reader to [11] for an overview. However, it is noteworthy that any “good” MSA of sufficiently similar sequences will exhibit *contextual locality*, that is, the suffixes starting in a column in the MSA are likely to be similar (i.e., share a long prefix) [3, 21]. For instance, the suffixes starting in the fifth column of the MSA in Figure 1 are $\text{ACT}\$, \text{ACT}\$$ and $\text{AGT}\$$.

We assume totally ordered alphabets. This induces a total order on strings. Specifically, we say a string S of length n is *lexicographically smaller* than a string T of length m if and only if there is some $\ell < \min\{n, m\}$ such that $S[0..\ell) < T[0..\ell)$ and either $n = \ell < m$ (S is a prefix of T) or $\ell < \min\{n, m\}$ and $S[\ell] < T[\ell]$, and write $S <_{\text{lex}} T$ in this case.

Given a string S , $\text{rank}_c(S, i)$ is the number of c ’s occurring in S up to (but excluding) index i , i.e., $\text{rank}_c(S, i) = |\{j \in [0..i) \mid S[j] = c\}|$. The *select* function returns the index of the i th occurrence of c (zero-based) in S , i.e., $\text{rank}_c(S, \text{select}_c(S, i)) = i$.

The *generalized suffix array* for a collection of strings S_1, \dots, S_n is an array GSA where $\text{GSA}[i] = (k, j)$ indicates that there are i lexicographically smaller suffixes than $\text{suf}_j(S_k)$ among the suffixes of S_1, \dots, S_n . We assume the strings to be *dollar-terminated*, that is, the last character of each string S_i is ‘\$’, which is smaller than all other characters. In the case of a tie between equal suffixes, we define the one occurring earlier in the input to be smaller.²

² This is equivalent to using n terminal symbols $\$1 < \dots < \n and terminating S_i with $\$i$. Hence the name ‘multidollar-EBWT.’

Throughout this paper, we use the multidollar-EBWT as the BWT for string collections (see [10] for an overview of such BWT variants). In the remainder of this paper, we refer to the multidollar-EBWT just by BWT. It is defined as follows. Let $\text{GSA}[i] = (k, j)$. Then $\text{BWT}[i] = S_k[j - 1]$ if $j > 0$ and $\text{BWT}[i] = S_k[|S_k| - 1] = '\$'$ otherwise. An example can be seen in Figure 1. The *LF-mapping* is a function such that $\text{LF}[i] = \text{GSA}^{-1}[(k, j - 1)]$ for $\text{GSA}[i] = (k, j)$ if $j > 0$ and $\text{LF}[i] = \text{GSA}^{-1}[(k, |S_k| - 1)]$ otherwise. LF can thus be used to iterate over an input string in reverse order, given the index p of the last character of the input string in BWT. Specifically, $S_k = \text{BWT}[\text{LF}^{|S_k|-1}[p]] \dots \text{BWT}[\text{LF}^1[p]]\text{BWT}[\text{LF}^0[p]]$ where $\text{GSA}[p] = (k, 0)$.

Although we use the dollar-EBWT, the algorithms presented in this paper are applicable to all BWT variants where for each input string S there is an index i such that the indices $\text{LF}^0[i], \text{LF}^1[i], \dots, \text{LF}^{|S|-1}[i]$ are all distinct and $S = \text{BWT}[\text{LF}^{|S|-1}[i]] \dots \text{BWT}[\text{LF}^0[i]]$.³

► **Definition 1** (TAG [21]). *Let $\text{GSA}[i] = (k, j)$. Then $\text{TAG}[i]$ is the column in the alignment of character j of string k .*

Note that this definition gives an immediate linear-time algorithm for computing TAG: For each character in the alignment, use the inverse of GSA to find the corresponding position in TAG and write the character's column. However, this naïve approach requires holding the inverse of GSA in main memory. Definition 1 is equivalent to the following definition based on the BWT instead of the suffix array, which follows immediately from the definition of the BWT given above.

► **Fact 1.** *$\text{TAG}[i]$ is the column in the alignment of the character immediately following $\text{BWT}[i]$ in the dataset.*

We will use this definition from now on as it does not depend on GSA. In simpler terms, consider a character in column i in row s which corresponds to $\text{BWT}[j]$. Then $\text{TAG}[j]$ is *col*, where *col* is the first column after i where there is a non-gap character in row s . An example of the TAG array of a multiple string alignment (MSA) is shown in Figure 1. For instance, the base G in the first string corresponds to $\text{BWT}[9] = \text{G}$, and the character following this G in the alignment occurs in column 4. Therefore, we have $\text{TAG}[9] = 4$.

In the next section, we present an algorithm that requires only access to the BWT, LF and the positions of the gaps in the alignment to compute the run-length encoded TAG array.

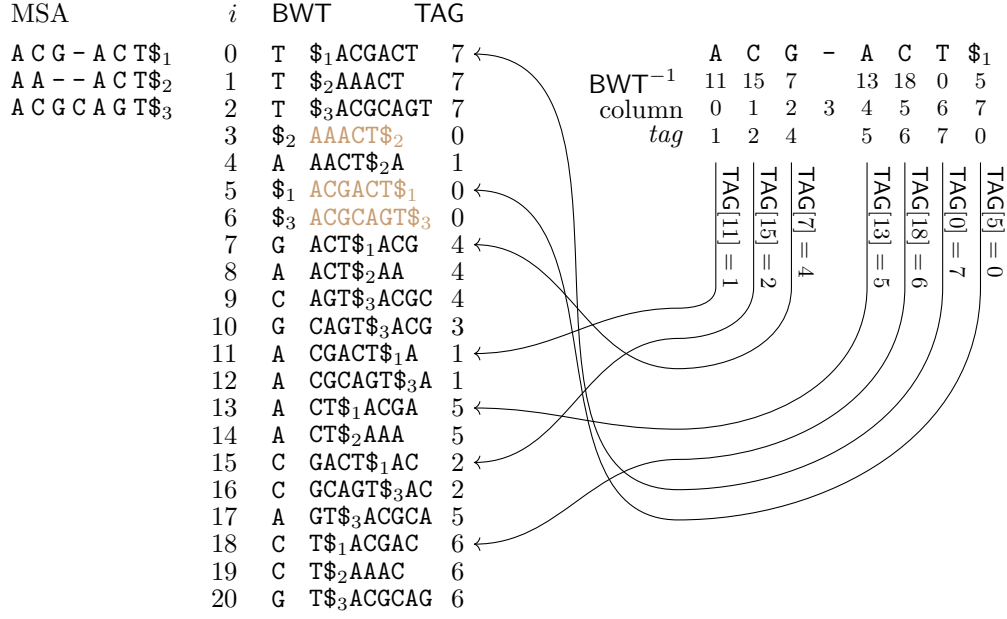
3 Computing the TAG array

Before describing our algorithm we briefly recall how we can reconstruct an input string from the BWT using LF. For this, we start at the position p of the string's last character (a '\$' in our case) in the BWT.⁴ $\text{LF}[p]$ now gives us the position in the BWT of the character preceding the '\$' in the string, $\text{LF}[\text{LF}[p]]$ gives us the character preceding that, and so on.

Essentially, we compute TAG using the inverse of Fact 1: for each TAG value *tag*, we find the indices where the TAG array TAG has value *tag*. Consider a column *col*. By Fact 1, TAG

³ Notably, this excludes the original EBWT for non-primitive input strings, because there each root of such a string corresponds to a distinct cycle in LF. As far as we can tell, all other BWT variants for string collections satisfy this requirement.

⁴ Equivalently, the rank of the input string in the sorted list of all rotations. Many tools for computing the BWT directly output these indices [6, 31, 33].



■ **Figure 1** An example MSA (top left) with the corresponding BWT (centre, instead of the sorted suffixes we display the sorted rotations for clarity). Strings represented in the MSA are coloured (■). On the right, the first string is displayed with the corresponding positions in the BWT and TAG values.

should have value col at the indices where the characters preceding the characters in column col are in BWT.

We consider each column of the alignment from right to left and maintain the BWT positions by iterating over the input strings from right to left using LF as is done when reconstructing the strings from BWT. By incorporating the positions of the gap symbols, this is synchronized such that we consider a column of the alignment at each time step.

Note that, given the indices of the characters in column col in the BWT, the indices of the preceding characters can be found with a single LF step each. If a symbol in the alignment in column col is a gap ('-'), the first preceding non-gap character has a TAG value greater than col . Therefore, in this case we just ignore the current column in this string.

Algorithm 1 shows the procedure. Note that any two iterations of the second for-loop of Algorithm 1 concern two different col values (columns). Therefore, no two iterations of this loop can concern the same TAG run. It is thus possible to immediately compute the TAG runs without having to store the entire TAG array.

Modifying Algorithm 1 accordingly, we obtain an algorithm that outputs the TAG runs in order of decreasing TAG value. To obtain the run-length compressed TAG array, it is hence necessary to sort the TAG runs by (start or end) index afterwards.

3.1 Practical construction

In the previous section, it is noted that one can use Algorithm 1 to directly compute the TAG runs. Indeed, in the inner for-loop, all indices i in TAG are considered where $TAG[i] = col$ for a specific value of col . Thus, it is possible to collect all these indices and then sort them in increasing order after the inner for-loop. Each run of consecutive indices in the resulting sorted list then corresponds to a TAG run of value col . In the following, we say that there is

```

1 for  $i = 0 \rightarrow n - 1$  do
2   |  $\text{bwt}_{\text{pos}}[i] = \text{LF}(p_i)$  where  $p_i$  is the position of the $ of string  $i$  in BWT;
3 end
4 for  $col = m - 1 \rightarrow 0$  do
5   | for  $i = 0 \rightarrow n - 1$  do
6     | if  $\text{alignment}[i][col] \neq '-'$  then
7       |    $\text{TAG}[\text{bwt}_{\text{pos}}[i]] \leftarrow col$ ;
8       |    $\text{bwt}_{\text{pos}}[i] \leftarrow \text{LF}(\text{bwt}_{\text{pos}}[i])$ ;
9     | end
10  | end
11 end

```

■ **Algorithm 1** Simple algorithm for computing TAG from an alignment of length m of n strings. Throughout the algorithm, $\text{bwt}_{\text{pos}}[i]$ is the position in the BWT of the character preceding the column col in string i .

a TAG run of value tag from l to r ($l \leq r$) if $\text{TAG}[l] = \dots = \text{TAG}[r] = tag$ and denote this with $tag-[l, r]$.

The number of objects involved in sorting and the number of LF accesses of the approach just described can be reduced with the following observations:

- If $\text{TAG}[i] = \text{TAG}[j]$, it is likely that $\text{TAG}[\text{LF}[i]] = \text{TAG}[\text{LF}[j]]$ [21].
- If i and j belong to the same BWT run, we have $\text{LF}[j] = \text{LF}[i] + (j - i)$ [19].

Because both the TAG array and the BWT have contextual locality [3, 21], two indices in the same TAG run likely belong to the same BWT run. Therefore, a TAG run $tag-[l, r]$ likely implies a TAG run $(tag - 1)-[\text{LF}[l], \text{LF}[r]]$. We can thus often operate on these runs instead of on the individual strings.

We maintain a set of TAG runs such that after each iteration of the inner for-loop, these TAG runs are maximal and disjoint. Each such TAG run is associated with the set of strings corresponding to the contained BWT indices. For each col we thus

1. remove indices from their current TAG run where the corresponding row in the MSA has a gap symbol in the current column,
2. insert indices where the corresponding row in the MSA had a gap symbol in the column processed in the previous iteration (but a base in the current column),
3. perform the LF step for each run, and
4. merge adjacent runs (e.g. applying LF to the TAG runs $5 - [13, 14]$ and $5 - [17, 17]$ in Figure 1 results in $4 - [7, 8]$ and $4 - [9, 9]$, which can clearly be merged) and output the result.

Note that, during the LF step it may be possible that a run is split if it crosses a BWT run boundary. In our example, this happens with the TAG run $6 - [18, 20]$ which results in the TAG runs $5 - [13, 14]$ and $5 - [17, 17]$. Thus, the data structure used for maintaining the indices in a run must support concatenation (for merging), splitting, and removal of an element (which may result in two distinct runs). Data structures which support these operations in (amortized) $\mathcal{O}(\log n)$ time are e.g. Red-Black trees or Splay trees [36, 37]. We use Splay trees because of their simpler implementation.

4 A sampled TAG array

Because both the MSA and BWT possess contextual locality, the TAG array of an MSA of similar strings is likely run-length compressible [21]. One can facilitate random access to

the run-length compressed TAG array using a sparse bit vector indicating the boundaries of the TAG runs. Assuming r_{TAG} TAG runs and an alignment with length m and N non-gap characters, this would require at least $r_{\text{TAG}} \cdot \log_2 m + r_{\text{TAG}} \cdot \log_2 \frac{N}{r_{\text{TAG}}}$ bits for the TAG run labels and the run boundaries.

However, r_{TAG} is likely to be larger than the number r of runs of the BWT (and guaranteed to be at least the length m of the alignment). Additionally, the alphabet used for TAG is much larger than that for BWT. Therefore, a naïve run-length compressed TAG array would require many times as much memory as the run-length compressed BWT, with the majority of the memory used for the run labels. In this section we present a technique for sampling the TAG array where we store the labels of only a fraction $\frac{1}{s}$ of the TAG runs, for some user-defined parameter s . For this, LF is assumed to be available.⁵

Note that there is no obvious method to just sample every s -th TAG value and use LF to walk to the next sampled TAG run for a query. This is because, regardless of which cells are chosen as the starting points for LF steps in the TAG runs, the LF steps may “jump over” sampled TAG runs.⁶ As a consequence, it would not be possible to guarantee that a sampled TAG run can be reached with a given number of LF steps. In the following, we present a more involved sampling strategy which guarantees that we can always reach a sampled TAG run with fewer than s LF steps.

The following concepts and data structures are illustrated in Figure 2 for our running example.

Let $e[i]$ be the index of the TAG run containing $\text{LF}[b_i]$, where b_i is the start of TAG run i . Let \mathbb{RB} be a (sparse) bit vector indicating the TAG run boundaries, i.e., $\mathbb{RB}[i] = 1$ if and only if i is the start of a TAG run. We have $b_i = \text{select}_1(\mathbb{RB}, i)$ and $e[i] = \text{rank}_1(\mathbb{RB}, \text{LF}[b_i] + 1) - 1$. Therefore, $e[i]$ can be computed with one LF computation and a select_1 and a rank_1 query and we do not have to store e explicitly.

Now let TAG' be such that $\text{TAG}'[i]$ is the TAG value of the i th TAG run. Note that there are at most n indices i where $\text{TAG}'[e[i]] \geq \text{TAG}'[i]$, because the run head of such a TAG run must correspond to the first character in a string of the alignment. Let the set of these indices be R . Now consider the digraph $G = (V, E)$ with $V = \{0, \dots, r_{\text{TAG}} - 1\}$ and $E = \{(i, e[i]) \mid i \in [0..r_{\text{TAG}} - 1] \setminus R\}$. That is, G is a graph where each TAG run is a node and there is an edge from a node u to v if and only if an LF step from the run-head of TAG run u results in an index in TAG run v . By definition, for each edge $(i, e[i]) \in E$ we have $\text{TAG}'[e[i]] < \text{TAG}'[i]$. Therefore, G is acyclic. Additionally, the out-degree of each node is at most 1. Thus, G is not only a DAG but a rooted forest with R as the roots and each edge pointing “upwards” towards a root (cf. Figure 2).

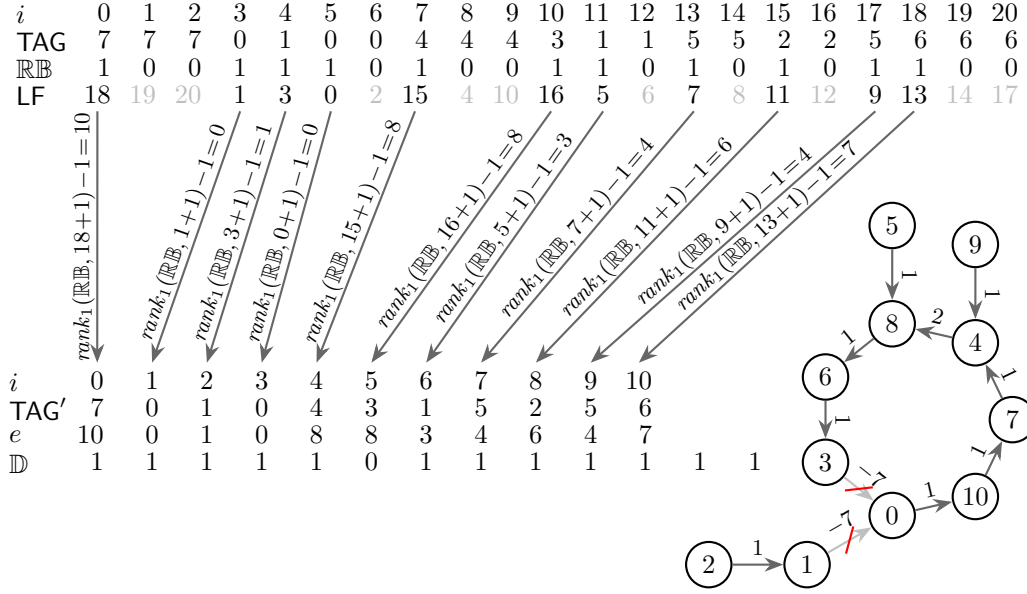
We require a set of sampled TAG runs to satisfy one constraint: Given a sampling rate s , the distance (in G) from an unsampled node to a sampled node should be less than s . Note that this implies that the roots R of the forest are sampled.

Sampled TAG runs are marked in a bit vector \mathbb{S} . Rank support on this bit vector is used to access an array L storing the TAG values of the sampled TAG runs. We can thus decide in constant time whether a TAG run is sampled and retrieve the TAG value if it is.

To retrieve the TAG value of an unsampled TAG run i , we compute $e[i]$ as described above, recursively find the TAG value of TAG run $e[i]$, and add the difference $\text{TAG}'[i] - \text{TAG}'[e[i]]$

⁵ This is not a major restriction because efficient access to LF can be facilitated using space proportional to the number of runs r of the BWT [22], and LF is available anyway in many tools using the BWT.

⁶ In any sensible MSA, there is no column which contains only gap symbols. Note however, that there may very well be TAG runs where each corresponding symbol in the alignment is preceded by a gap symbol.



■ **Figure 2** Top/Left: The TAG array, bit vector \mathbb{RB} and LF for our example (cf. Figure 1). Below, the run-length compressed TAG array TAG' is shown with the array e derived from LF. Values of LF that are not used (i.e., do not occur at the start of a TAG run) are printed in light gray. Bottom right: The graph G defined by e . We have $R = \{1, 3\}$ because $\text{TAG}'[e[1]] = \text{TAG}'[e[3]] = \text{TAG}'[0] = 7 \not\leq 0 = \text{TAG}'[1] = \text{TAG}'[3]$. For all other $i \in \{0, \dots, 10\}$ we have $\text{TAG}'[e[i]] < \text{TAG}'[i]$. Above each edge $(i, e[i])$, the difference $\text{TAG}'[e[i]] - \text{TAG}'[i]$ is displayed. Edges from nodes in R (i.e., where this difference is negative) are crossed out. Note that the remaining edges form a rooted forest where the roots are exactly the nodes in R .

to the result. The distance-constraint ensures that we can get any TAG value using fewer than s recursion steps. For this to work, we encode the difference $\text{TAG}'[i] - \text{TAG}'[e[i]]$ for all unsampled i with unary encoding in a bit vector \mathbb{D} . For the sampled TAG runs, we also encode the value 1 for simplicity. Specifically, if the i th one-bit in S is at position p , the $i+1$ th one-bit is at position $p + (\text{TAG}'[i] - \text{TAG}'[e[i]])$ if TAG run i is sampled and at position $p+1$ otherwise. Two select queries on this bit vector then suffice to extract this difference and thus to compute $\text{TAG}'[i]$ given $\text{TAG}'[e[i]]$. (We also add an additional 1 for an easier implementation, see Algorithm 2.)

```

1 Procedure getTAG( $i$ )
2   if  $S[i] = 1$  then return  $L[\text{rank}_1(S, i)]$ ;           // TAG run  $i$  is sampled
3    $\text{difference} \leftarrow \text{select}_1(\mathbb{D}, i+1) - \text{select}_1(\mathbb{D}, i)$ ; // Because an  $(r_{\text{TAG}} + 1)$ th 1 is
4      $\text{runStart} \leftarrow \text{select}_1(\mathbb{RB}, i)$ ;                 // called  $b_i$  in the text
5      $\text{precBwtPos} \leftarrow \text{LF}[\text{runStart}]$ ;
6      $\text{precRun} \leftarrow \text{rank}_1(\text{precBwtPos} + 1) - 1$ ;      //  $\text{precRun} = e[i]$ 
7   return  $\text{difference} + \text{getTAG}(\text{precRun})$ ;

```

■ **Algorithm 2** Computing the TAG value of a TAG run, given its index i .

Algorithm 2 shows how the TAG access is performed.

Space complexity

Let N be the total number of characters in the alignment, m the length of the alignment

and g the total number of gap symbols preceding the run-heads of the sampled TAG runs. In a good MSA, the number of gap symbols is expected to be very small compared to the size of the data set. Additionally, because the number of TAG runs is also expected to be small, g is expected to include only every N/r_{TAG} th gap symbol. We assume that there are $\frac{1}{s}r_{\text{TAG}}$ sampled TAG runs. In the next section, it is shown that this is possible.

In addition to access to LF, we need

- the array \mathbf{L} containing the sampled TAG values ($r_{\text{TAG}} \cdot \lceil \log_2 m \rceil / s$ bits),
- the bit vector \mathbb{RB} with rank_1 and select_1 support,
- the bit vector \mathbb{S} with rank_1 support ($r_{\text{TAG}}(1 + o(1))$ bits), and
- the bit vector \mathbb{D} with select_1 support $((r_{\text{TAG}} + g)(1 + o(1))$ bits).

The bit vector \mathbb{RB} can be implemented e.g. with a plain bit vector for $\mathcal{O}(1)$ rank/select operations ($N(1 + o(1))$ bits) [25], or using the Elias-Fano representation for non-decreasing sequences where rank and select take $\mathcal{O}\left(\log \frac{N}{r_{\text{TAG}}}\right)$ and $\mathcal{O}(1)$ time, respectively ($r_{\text{TAG}}\left(2 + \log_2 \frac{N}{r_{\text{TAG}}}\right)$ bits) [15, 18].

Time complexity

We obtain a time complexity of $\mathcal{O}(s \cdot (t_r + t_s + t_{\text{LF}}))$, where t_r , t_s and t_{LF} are the time complexities of rank and select queries to \mathbb{RB} and computing $\text{LF}[i]$, respectively.

4.1 Selecting the sampled TAG runs

For selecting the sampled TAG runs, we use an (optimal) greedy algorithm that minimizes the number of sampled TAG runs under the above constraints. Recall that we want to choose a smallest set $S \subseteq V$ of nodes in a forest $G = (V, E)$ such that it requires fewer than s steps to reach a node in S from any node not in S , where each step must go “upwards” (i.e., towards the corresponding root). Let this smallest distance of a node u upwards to the closest node $v \in S$ be $a(u)$ and call v the *witness* of u . For each node u in S we have $a(u) = 0$. Since the trees of the forest are independent of each other, we assume that the graph is a tree in the following.

Note that the root of the tree must always be in S , and that removing a subtree can never increase the size of the smallest solution. Now consider a node u with maximum depth $d(u)$ (i.e., distance to the root). Regardless of which ancestor v of u satisfying $d(u) - d(v) < s$ is chosen, by selection of u , v is a valid witness for all descendants of v (because their distance to v is at most $d(u) - d(v) < s$). Therefore, we may remove the subtree rooted at v (including v) and return v plus the solution of the remaining tree. Finally, it is optimal to choose v such that $d(u) - d(v) < s$ is maximal. This is because choosing any lower node would eliminate a strict subset of the descendants of v , and can therefore not lead to a better solution. We thus obtain the following algorithm: While the tree is non-empty, find a node u with maximum depth $d(u)$, determine the highest ancestor v of u that satisfies $d(u) - d(v) < s$, output v , and remove the subtree rooted at v .

Note that, for all choices except the last one where the root of the tree is chosen, v can always be chosen such that $d(u) - d(v) = s - 1$. Therefore, for each node v in S besides the root, there are at least $s - 1$ nodes not in S for which v is the witness. In a tree with n nodes we therefore have $|S| \leq \lfloor \frac{n}{s} \rfloor + 1$.

The above algorithm can be emulated simpler than described above as follows: for a node u , let $d'(u)$ be the maximum number of nodes on a “downwards” path that does not include nodes in S . For each u in S we have $d'(u) = 0$. For each u not in S , $d'(u) = 1 + \max(\{d'(v) \mid v \text{ is child of } u\} \cup \{0\})$ holds. By the observations above, each node in S besides the root has a child v with $d'(v) = s - 1$. Conversely, every node with a child v with

$d'(v) \geq s - 1$ must be in S . Since $d'(u)$ depends only on u 's children, d' (and thus S) can be computed during a bottom-up traversal. This immediately gives a simple optimal bottom-up traversal algorithm for deciding which nodes are in S .

Directly constructing the sampled index

Note that, during the construction algorithm presented in Section 3, we traverse the TAG runs in exactly such a bottom-up order as required for the optimal sampling algorithm above. It is therefore possible to immediately select the sampled TAG runs during the construction.

For this, we use a dynamic bit vector to mark the TAG run heads. The sampled TAG values are stored together with the start position of the respective TAG run, and then sorted afterwards according to this position. Finally, for the unsampled TAG runs, we need to store the difference $\text{TAG}'[i] - \text{TAG}'[e[i]]$. As explained in the previous section, the number of preceding gaps is usually small. Therefore, there are only few runs where $\text{TAG}'[i] - \text{TAG}'[e[i]]$ is greater than one. We store only these differences in combination with the start position of the respective TAG run. From these TAG run boundaries, sampled tag runs, and unsampled tag runs i with $\text{TAG}'[i] - \text{TAG}'[e[i]] > 1$ we can then construct all data structures needed for the sampled TAG index.

4.2 Reporting the distinct TAGs in a BWT interval

Given an interval $[l, r]$ (e.g. resulting from a backward search on the BWT), we would like to report the set of (distinct) TAG values in $\text{TAG}[l, r]$. In particular, the time complexity of this operation should depend only on the size of the output and not on the size $r + 1 - l$ of the interval $[l, r]$. Note that we can use rank_1 queries on \mathbb{RB} to “translate” the interval $[l, r]$ such that it refers to the run-length compressed TAG array TAG' instead of TAG (without affecting the set of distinct TAG values). We therefore consider the problem of reporting the distinct TAG values in $\text{TAG}'[l, r]$.

To achieve this, we use Muthukrishnan's [29] approach to the document listing problem. We recall it here for completeness. Let C be an array such that $C[i]$ is the largest $j < i$ where $\text{TAG}'[j] = \text{TAG}'[i]$ (or -1 if no such j exists). We want to find all distinct TAG values in the interval $\text{TAG}'[l, r]$. The first TAG value $\text{TAG}'[i]$ can be determined by a *range-minimum query* (RMQ) on $C[l, r]$, which yields the index i of the minimum in $C[l, r]$; note that $C[i] < l$. Then we recursively consider the sub-intervals $[l, i - 1]$ and $[i + 1, r]$ of $[l, r]$. Let $[l', r']$ be one of these two intervals. We determine the minimum $C[j]$ in $C[l', r']$ using the RMQ data structure. If $C[j] \geq l$, then $\text{TAG}'[l', r']$ solely contains TAG values that have already been found and the recursion stops. Otherwise, we output $\text{TAG}'[j]$ and recurse with $[l', j - 1]$ and $[j + 1, r']$. The number of range-minimum queries and recursion steps is clearly proportional to the size of the output.

It is undesirable to keep C in memory due to its size. However, it is possible to equivalently check whether $\text{TAG}'[i]$ has already been output with a global static bit vector where those TAG' values are marked that have already been output [34]. Note that this bit vector can be reset in time proportional to the size of the number of set markings [34].

A succinct RMQ data structure that needs $2r_{\text{TAG}}(1 + o(1))$ bits and supports constant-time range-minimum queries can be constructed in linear time [20].

5 Experimental evaluation

We implemented the algorithms and data structures described in this paper in C++. In particular, our implementation computes the sampled TAG array as described in Sections 3 and 4 and then computes the sampled TAG index as described in Section 4 together with the

Sampling rate	construction time	sampled TAG runs	index size	time/TAG
2	213 s	48 536 766	455 MiB	1.74 μ s
4	238 s	23 800 106	379 MiB	2.61 μ s
8	300 s	11 422 048	340 MiB	4.11 μ s
16	480 s	5 244 736	321 MiB	6.99 μ s
32	992 s	2 267 547	312 MiB	13.29 μ s
64	2 002 s	1 030 783	308 MiB	25.31 μ s

■ **Table 1** For different sampling rates, we list the time for constructing the TAG index (excluding the memory used for LF), the size of the index in memory (excluding LF) and the time per output TAG, averaged over 29 185 037 output TAGs and excluding the time to load the index. Constructing the index always needed 2225 MiB of memory.

data structure for listing the distinct TAGs described in Section 4.2. For the bit vector and *rank* and *select* implementations, we used the *Succinct Data Structure Library 2.0* [24]. The source code of our implementation is publicly available.⁷

As test data, we use 1000 human Chromosome 19 haplotypes from [7] and the corresponding alignment from [32]. All experiments were conducted on a Linux-6.8.0 machine with an Intel Xeon Gold 6338 CPU and 512 GB of RAM. As the compiler we used GCC 13.3.0. Currently, our construction algorithm is single-threaded. We only test our query algorithm with one thread. However, note that concurrent access to our index is trivially possible.

The MSA uses $1.67 \cdot 10^8$ gaps and has $5.91 \cdot 10^{10}$ non-gap characters. We computed the dollar-EBWT strands using *1g* [31]. *Ropebwt3* requires that both forward and reverse-complemented strands are in the index, and we of course need to compute the TAG runs with the same index to ensure that the TAG runs match the BWT. However, we only compute the TAGs for the forward strands. This means that our TAG runs do not cover all positions in the BWT. Space between two TAG runs is treated like any other TAG run, except marked as “no TAG available”.

The dollar-EBWT has 91 081 437 runs, and there are 137 981 814 runs in our index, 97 979 057 of which have a TAG value.

To test mapping performance, we extracted 10^7 “reads” with 100bp each, chosen uniformly at random from the sequences used for the alignment. We mutated each base with a probability of 1% and reverse-complemented each read with a probability of 50%. We then used *ropebwt3* to find the corresponding MEMs and queried our index with the resulting BWT-interval. Finally, we projected each TAGs to the first sequence in the data set as a designated linear reference. *ropebwt3* finds 17 953 756 MEMs, which cumulatively correspond to 29 185 037 TAGs (columns in the MSA).

Table 1 shows construction and query performance for varying sampling rates. The increased construction time with increasing sample rate stems entirely from the construction of the index for reporting the distinct TAG values, constructing the TAG index always took about 175s. This is because, for constructing this index, we need to access all TAG values, which we do using our TAG index. Since the average query time of the TAG index is proportional to the sampling rate, a larger sampling rate necessarily slows this down.

⁷ <https://gitlab.com/qwerzuiop/msatag>

By using e.g. the non-sampled TAG array or using a more clever algorithm, this could be remedied in the future. The memory used for constructing the index is roughly the same for all tested sampling rates.

Note that with a sampling rate of 4, our program takes less time for mapping the MEMs to columns in the MSA with a single thread (76.0 s) as ropebwt3 uses for finding the MEMs with 8 threads (81.7 s).

6 Conclusion

We described an algorithm that can compute the TAG array of a multiple sequence alignment (MSA) using the LF function of the BWT whose working memory is proportional to the number of sequences. Additionally, we described a TAG index that is able to report the unique tags corresponding to a BWT interval (i.e., the columns where matches corresponding to the BWT interval occur in the MSA) in time proportional to the size of the output using standard document listing techniques. We also gave a non-trivial sampling strategy for the TAG index and showed that our TAG construction algorithm can be adapted to output just the sampled TAG values. Finally, we demonstrated experimentally that our construction algorithm and index perform well on real-world data.

Our work enables e.g. the efficient mapping of matches in BWT-based indices to a designated reference genome, which programs such as ropebwt3 currently lack [27]. Our techniques could also be used to obtain more effective chaining heuristics in programs using a BWT-based index to find seeds for a seed-and-extend approach (e.g. Moni-align [39]).

References

- 1 Francesco Andreace, Pierre Lechat, Yoann Dufresne, and Rayan Chikhi. Comparing methods for constructing and representing human pangenome graphs. *Genome Biology*, 24(1):274, 2023.
- 2 Jasmijn A Baaijens, Paola Bonizzoni, Christina Boucher, Gianluca Della Vedova, Yuri Pirola, Raffaella Rizzi, and Jouni Sirén. Computational graph pangenomics: a tutorial on data structures and their applications. *Natural computing*, 21(1):81–108, 2022. doi:10.1007/s11047-022-09882-6.
- 3 Andrej Baláž, Travis Gagie, Adrián Goga, Simon Heumos, Gonzalo Navarro, Alessia Petescia, and Jouni Sirén. Wheeler maps. In José A. Soto and Andreas Wiese, editors, *LATIN 2024: Theoretical Informatics*, pages 178–192, Cham, 2024. Springer Nature Switzerland. doi:10.1007/978-3-031-55598-5_12.
- 4 Hideo Bannai, Travis Gagie, et al. Refining the r-index. *Theoretical Computer Science*, 812:96–108, 2020.
- 5 Nico Bertram, Johannes Fischer, and Lukas Nalbach. Move-r: Optimizing the r-index. In Leo Liberti, editor, *22nd International Symposium on Experimental Algorithms (SEA 2024)*, volume 301 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 1:1–1:19, Dagstuhl, Germany, 2024. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.SEA.2024.1.
- 6 Christina Boucher, Travis Gagie, Alan Kuhnle, Ben Langmead, Giovanni Manzini, and Taher Mun. Prefix-free parsing for building big BWTs. *Algorithms for Molecular Biology*, 14:1–15, 2019. doi:10.1186/s13015-019-0148-5.
- 7 Christina Boucher, Travis Gagie, I Tomohiro, Dominik Köppl, Ben Langmead, Giovanni Manzini, Gonzalo Navarro, Alejandro Pacheco, and Massimiliano Rossi. PHONI: Streamed matching statistics with multi-genome references. In *2021 Data Compression Conference (DCC)*, pages 193–202. IEEE, 2021.
- 8 Thomas Büchler, Jannik Olbrich, and Enno Ohlebusch. Efficient short read mapping to a pangenome that is represented by a graph of ED strings. *Bioinformatics*, 39(5):btad320, 2023.

- 9 Michael Burrows and David Wheeler. A block-sorting lossless data compression algorithm. *Digital SRC Research Report*, 124, 1994.
- 10 Davide Cenzato and Zsuzsanna Lipták. A survey of BWT variants for string collections. *Bioinformatics*, 40(7):btae333, 2024. doi:10.1093/bioinformatics/btae333.
- 11 Maria Chatzou, Cedrik Magis, Jia-Ming Chang, Carsten Kemena, Giovanni Bussotti, Ionas Erb, and Cedric Notredame. Multiple sequence alignment modeling: methods and applications. *Briefings in bioinformatics*, 17(6):1009–1023, 2016.
- 12 Dustin Cobas, Travis Gagie, and Gonzalo Navarro. Fast and Small Subsampled R-indexes. *ACM Transactions on Algorithms*, 22(1):1–29, 2025.
- 13 Lore Depuydt, Omar Y Ahmed, Jan Fostier, Ben Langmead, and Travis Gagie. Run-length compressed metagenomic read classification with SMEM-finding and tagging. 2025. doi:10.1016/j.isci.2025.114029.
- 14 Diego Díaz-Domínguez and Gonzalo Navarro. Efficient Construction of the BWT for Repetitive Text Using String Compression. In Hideo Bannai and Jan Holub, editors, *33rd Annual Symposium on Combinatorial Pattern Matching (CPM 2022)*, volume 223 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 29:1–29:18, Dagstuhl, Germany, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.CPM.2022.29.
- 15 Peter Elias. Efficient storage and retrieval by content and address of static files. *Journal of the ACM (JACM)*, 21(2):246–260, 1974.
- 16 Massimo Equi, Veli Mäkinen, and Alexandru I Tomescu. Graphs cannot be indexed in polynomial time for sub-quadratic time string matching, unless SETH fails. *Theoretical Computer Science*, 975:114128, 2023. doi:10.1016/j.tcs.2023.114128.
- 17 Parsa Eskandar, Benedict Paten, and Jouni Sirén. Lossless Pangenome Indexing Using Tag Arrays. *bioRxiv*, pages 2025–05, 2025. doi:10.1101/2025.05.12.653561.
- 18 Robert Mario Fano. *On the number of bits required to implement an associative memory*. Massachusetts Institute of Technology, Project MAC, 1971.
- 19 Paolo Ferragina and Giovanni Manzini. Opportunistic data structures with applications. In *Proceedings 41st annual symposium on foundations of computer science*, pages 390–398. IEEE, 2000.
- 20 Johannes Fischer. Optimal succinctness for range minimum queries. In *Latin American Symposium on Theoretical Informatics*, pages 158–169. Springer, 2010. doi:10.1007/978-3-642-12200-2_16.
- 21 Travis Gagie. Tag arrays. *arXiv preprint arXiv:2411.15291*, 2024.
- 22 Travis Gagie, Gonzalo Navarro, and Nicola Prezza. Optimal-time text indexing in bwt-runs bounded space. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1459–1477. SIAM, 2018.
- 23 Erik Garrison, Jouni Sirén, Adam M Novak, Glenn Hickey, Jordan M Eizenga, Eric T Dawson, William Jones, Shilpa Garg, Charles Markello, Michael F Lin, et al. Variation graph toolkit improves read mapping by representing genetic variation in the reference. *Nature biotechnology*, 36(9):875–879, 2018.
- 24 Simon Gog, Timo Beller, Alistair Moffat, and Matthias Petri. From theory to practice: Plug and play with succinct data structures. In *13th International Symposium on Experimental Algorithms, (SEA 2014)*, pages 326–337, 2014. doi:10.1007/978-3-319-07959-2_28.
- 25 Rodrigo González, Szymon Grabowski, Veli Mäkinen, and Gonzalo Navarro. Practical implementation of rank and select queries. In *Poster Proc. Volume of 4th Workshop on Efficient and Experimental Algorithms (WEA)*, pages 27–38. CTI Press and Ellinika Grammata Greece, 2005.
- 26 Daehwan Kim, Joseph M Paggi, Chanhee Park, Christopher Bennett, and Steven L Salzberg. Graph-based genome alignment and genotyping with HISAT2 and HISAT-genotype. *Nature biotechnology*, 37(8):907–915, 2019.
- 27 Heng Li. BWT construction and search at the terabase scale. *Bioinformatics*, 40(12):btae717, 11 2024. doi:10.1093/bioinformatics/btae717.

- 28 Francesco Masillo. Matching Statistics Speed up BWT Construction. In Inge Li Gørtz, Martin Farach-Colton, Simon J. Puglisi, and Grzegorz Herman, editors, *31st Annual European Symposium on Algorithms (ESA 2023)*, volume 274 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 83:1–83:15, Dagstuhl, Germany, 2023. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.ESA.2023.83.
- 29 Shanmugavelayutham Muthukrishnan. Efficient algorithms for document retrieval problems. In *Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 657–666, 2002.
- 30 Takaaki Nishimoto and Yasuo Tabei. Optimal-Time Queries on BWT-Runs Compressed Indexes. In Nikhil Bansal, Emanuela Merelli, and James Worrell, editors, *48th International Colloquium on Automata, Languages, and Programming (ICALP 2021)*, volume 198 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 101:1–101:15, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.ICALP.2021.101.
- 31 Jannik Olbrich. Fast and Memory-Efficient BWT Construction of Repetitive Texts Using Lyndon Grammars. In *33rd Annual European Symposium on Algorithms (ESA 2025)*, volume 351 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 60:1–60:19, Dagstuhl, Germany, 2025. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.ESA.2025.60.
- 32 Jannik Olbrich, Thomas Büchler, and Enno Ohlebusch. Generating multiple alignments on a pangenomic scale. *Bioinformatics*, 41(3):btaf104, 03 2025. doi:10.1093/bioinformatics/btaf104.
- 33 Marco Oliva, Travis Gagie, and Christina Boucher. Recursive prefix-free parsing for building big BWTs. In *2023 data compression conference*, pages 62–70. IEEE, 2023. doi:10.1109/DCC55655.2023.00014.
- 34 Kunihiko Sadakane. Succinct data structures for flexible text retrieval systems. *Journal of discrete Algorithms*, 5(1):12–22, 2007. doi:10.1016/j.jda.2006.03.011.
- 35 Jouni Sirén, Niko Välimäki, and Veli Mäkinen. Indexing graphs for path queries with applications in genome research. *IEEE/ACM transactions on computational biology and bioinformatics*, 11(2):375–388, 2014.
- 36 Daniel Dominic Sleator and Robert Endre Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, 32(3):652–686, 1985. doi:10.1145/3828.3835.
- 37 Robert Endre Tarjan. *Data structures and network algorithms*. SIAM, 1983.
- 38 Hervé Tettelin, Vega Massignani, Michael J Cieslewicz, Claudio Donati, Duccio Medini, Naomi L Ward, Samuel V Angiuoli, Jonathan Crabtree, Amanda L Jones, A Scott Durkin, et al. Genome analysis of multiple pathogenic isolates of *Streptococcus agalactiae*: implications for the microbial “pan-genome”. *Proceedings of the National Academy of Sciences*, 102(39):13950–13955, 2005.
- 39 Rahul Varki, Massimiliano Rossi, Eddie Ferro, Marco Oliva, Erik Garrison, Ben Langmead, and Christina Boucher. Accurate short-read alignment through r-index-based pangenome indexing. *Genome Research*, 35(7):1609–1620, 2025. doi:10.1101/gr.279858.124.