Efficient Construction of the BWT for Repetitive Text using String Compression

Diego Díaz-Domínguez[a,c], Gonzalo Navarro[b,c]

[a] *Department of Computer Science, University of Helsinki, Finland*
[b] *Department of Computer Science, University of Chile, Chile*
[c] *CeBiB — Center for Biotechnology and Bioengineering, Chile*

Abstract

We present a new semi-external algorithm that builds the Burrows–Wheeler transform variant of Bauer et al. (a.k.a., BCR BWT) in linear expected time. Our method uses compression techniques to reduce computational costs when the input is massive and repetitive. Concretely, we build on induced suffix sorting (ISS) and resort to run-length and grammar compression to maintain our intermediate results in compact form. Our compression format not only saves space but also speeds up the required computations. Our experiments show important space and computation time savings when the text is repetitive. In moderate-size collections of real human genome assemblies (14.2 GB - 75.05 GB), our memory peak is, on average, 1.7x smaller than the peak of the state-of-the-art BCR BWT construction algorithm (`ropebwt2`), while running 5x faster. Our current implementation was also able to compute the BCR BWT of 400 real human genome assemblies (1.2 TB) in 41.21 hours using 118.83 GB of working memory (around 10% of the input size). Interestingly, the results we report in the 1.2 TB file are dominated by the difficulties of scanning huge files under memory constraints (specifically, I/O operations). This fact indicates we can perform much better with a more careful implementation of our method, thus scaling to even bigger sizes efficiently.

Keywords: BWT, string compression, repetitive text

1. Introduction

The Burrows–Wheeler transform (BWT) [1] is a reversible string transformation that reorders the symbols of a text T according to the lexicographical

ranks of its suffixes. The features of this transform have turned it into a critical component for text compression and indexing [2, 3]. In addition to being reversible, the reordering reduces the number of equal-symbol runs in T, thus improving the compressibility. The BWT is also the main component of the so-called FM-index [4, 5], a self-index that supports pattern matching in time proportional to the pattern length. Briefly, the FM-index encodes T as its BWT and then uses its combinatorial properties [6] to look for patterns in the text efficiently. Popular bioinformatic tools [7, 8] rely on the FM-index to process data, as collections in this area are typically massive and repetitive, and the patterns to search for are short.

The FM-index is an important breakthrough as it dramatically reduces space usage compared to the classical suffix tree [9] and suffix array [10]. However, it still uses space proportional to T (i.e., succinct), making it impractical for massive input. This problem is relevant as massive collections are nowadays standard in many fields. A fortunate coincidence is that massive collections are usually highly repetitive too, and in that case, the number of equal-symbol runs in the BWT (denoted r in the literature) is considerably smaller than the text size. Gagie et al. [5] exploited this property to design the r-index, a compressed self-index that requires $O(r)$ bits of space and still supports efficient pattern matching.

The r-index is a promising solution to compress and index massive collections, but it lacks efficient construction algorithms that scale well with the input size. This limitation, of course, hampers its adoption in practical applications. One of the most important challenges (although not the only one) is how to obtain the BWT of T. Several algorithms in the literature produce the BWT in linear time [11, 12, 13, 14, 15]. Nevertheless, the computational resources their implementations require with large inputs are still too high. This limitation is particularly evident in Genomics, where the data can easily reach terabytes [16].

Some authors [17, 18, 19, 20, 21] have tackled the problem of computing big BWTs by exploiting the repetitiveness of the input. Their approach consists of extracting a set of representative strings from the text, performing calculations on them, and then extrapolating the results to the copies of those strings. For instance, the methods of Boucher et al. [18, 21] based on prefix-free parsing (PFP) use Karp–Rabin fingerprints [22] to create a dictionary of prefix-free phrases from T. They then create a parse by replacing the phrases in T with metasymbols, and finally construct the BWT using the dictionary and the parse. Similarly, Kempa et al. [17] consider a subset of positions in

T they call a string synchronizing set, from which they compute a partial BWT they then extrapolate to the whole text.

Although these repetition-aware techniques are promising, some are at a theoretical stage [17, 19, 20], while the rest [18, 21] have been empirically tested only under controlled settings, and their results depend on parameters that are not simple to tune. Thus, it is difficult to assess their performance under real circumstances.

Recently, Nunes et al. [23] proposed a method called GCIS that adapts the concept of *induced suffix sorting* (ISS) for compression. Their ideas are closely related to the linear-time BWT algorithm of Okanohara et al. [11]. Briefly, Okanohara et al. cut the text into phrases using ISS, assigning symbols to the phrases, and then replacing the phrases with their symbols. They apply this procedure recursively until all the text symbols are unique. Then, when they return from the recursions, they induce an intermediate BWT^i for the text of every recursion i using the previous BWT^{i+1}. The connection between these two methods is that GCIS captures in the grammar precisely the information that Okanohara et al. use to compute the BWT. Additionally, Díaz-Domínguez et al. [24] recently demonstrated that ISS-based compressors such as GCIS require much less computational resources than state-of-the-art methods like RePair [25] to encode the data while maintaining high compression ratios. The simple construction of ISS makes it an attractive alternative to processing high volumes of text. In particular, combining the ideas of Okanohara et al. with ISS-based compression is a promising alternative for computing big BWTs.

Our contribution. *Induced suffix sorting* (ISS) [26] has proved useful for compression [23, 24] and for constructing the BWT [11]. In this work, we show that compression can be incorporated into the internal stages of the BWT computation in a way that saves both working space and time. Okanohara et al. [11] use ISS to construct the BWT as follows: they build the texts $T^1, T^2, \ldots, T^h$, with $h = O(\log n)$, by applying recursive rounds of parsing that cut each T^i into phrases and replace the phrases by new symbols. Then, when they return from the recursions, they induce the BWT of every T^i from the BWT of the previous text T^{i+1}, generating the final BWT when they reach the first recursion level again. We use a technique similar to grammar compression to store the sets of phrases generated in the rounds of parsing, and run-length compression for the intermediate BWTs. This approach is shown not only to save the space required for those intermediate results but, importantly, the format we choose speeds up the computation

of the final BWT as we return from the recursion because the factorizations that help save space also save redundant computations. Unlike Okanohara et al., we receive as input a string collection and output its BCR BWT [12], a variant for string collections. The reason is that massive datasets usually contain multiple strings, in which case the BCR BWT variant is simpler to construct.

Early versions of this work appeared in *Proc. DCC'21* [24] and *Proc. CPM'22* [27]. In this extended article, we explain how to produce a smaller set of phrases in each recursion of parsing than the one we obtain by applying the regular ISS procedure. We aim to keep working memory and CPU consumption low, even for not-so-repetitive collections. Our technique to reduce the set of phrases is simple enough, so the extra time we spend in this step increases the overall performance. Additionally, we explain how to extend our ideas to compute the smallest BCR BWT (in terms of the number of runs) one can obtain by reordering the strings of the text. Finally, we empirically assessed our techniques in massive datasets.

Our experiments show that when the input is a collection of human genomes (a repetitive dataset), we are, on average, 5x faster than `ropebwt2` [8], one of the most efficient implementations of the BCR BWT algorithm. In the same datasets, we also outperform the PFP-based methods `pfp-ebwt` [21] and `r-pfpbwt` [28], being on average 2.8x faster than them while using much less memory. We also report the construction of the BCR BWT for a collection of 1.2 TB using an amount of working memory that did not exceed 10% of the input size, and a running time of less than 42 hours. Under not-so-repetitive scenarios (short Illumina reads), we are the second fastest tool and the second most space-efficient on average, being outperformed only by `ropebwt2` and `BCR_LCP_GSA` [12], respectively.

2. Related Concepts

2.1. Grammar and Run-length Compression

Grammar compression [29] consists of encoding a text T as a small context-free grammar $\mathcal{G}$ that produces only T. Formally, a grammar is a tuple $(V, \Sigma, \mathcal{R}, \mathsf{S})$, where V is the set of nonterminals, Σ is the set of terminals, $\mathcal{R}$ is the set of replacement rules and $\mathsf{S} \in V$ is the start symbol. The right-hand side of $\mathsf{S} \rightarrow C \in \mathcal{R}$ is referred to as the compressed form of T. The size of $\mathcal{G}$ is usually measured in terms of the number of rules, the sum

of the lengths of the right-hand sides of $\mathcal{R}$, and the length of the compressed string.

Run-length compression encodes the equal-symbol runs of maximal length in T as a sequence $(c_1, \ell_1), (c_2, \ell_2), \ldots, (c_{n'}, \ell_{n'})$ of $n' \leq n$ pairs, where every (c_i, ℓ_i), with $i \in [1, n']$, stores the symbol $c_i \in \Sigma$ of the ith run and its length $\ell_i \geq 1$. For instance, let $T[j..j'] = cccc$ be a substring with four consecutive copies of c, where $T[j-1] \neq a$ and $T[j'+1] \neq c$. Then $T[j..j]$ compresses to $(c, 4)$.

2.2. The Suffix Array

The *suffix array* [10] of a string $T[1..n] \in \Sigma^*$ is a permutation $SA[1..n]$ that enumerates the suffixes $T[j..n]$ of T in increasing lexicographic order, $T[SA[j]..n] < T[SA[j+1]..n]$, for $j \in [1..n-1]$. It is customary to divide SA into σ *buckets*. Specifically, a bucket $c \in \Sigma$ is a contiguous range $SA[j_c..j_{c+1}-1]$ storing the text positions of the suffixes of T prefixed by c.

The *generalized suffix array* [30] is a variant of SA that enumerate the suffixes of a string collection $\mathcal{T} = \{T_1, T_2, \ldots, T_k\}$ over the alphabet Σ. Let $T = T_1\$_1T_2\$_2 \cdots T_k\$_k$ be a string over the alphabet $\{\$_1, \$_2, \ldots, \$_k\} \cup \Sigma$ storing the concatenation of $\mathcal{T}$ such that each $T_x \in \mathcal{T}$ ends in T with a unique sentinel $\$_x$. The values of the k distinct sentinels $\$_1, \ldots, \$_k \notin \Sigma$ are chosen arbitrarily but are smaller than any symbol in Σ. The generalized suffix array of $\mathcal{T}$ is then a vector $GSA[1..n = |T|]$ equal to the suffix array of T. Put simply, GSA sorts the suffixes in lexicographical order, breaking ties for equal suffixes according to the order of the strings in $\mathcal{T}$ that the sentinels induce. Still, in practice, a construction algorithm for GSA does not require keeping an explicit set of k sentinels. One can get the same result by concatenating the elements of $\mathcal{T}$ in $T = T_1\$T_2\$ \cdots T_k\$$ using the same symbol $\$$ as a boundary between strings, sorting the suffixes of $\mathcal{T}$ (substrings in T) lexicographically, and breaking ties for equal suffixes according to an arbitrary rule. Figure 1 shows an example of GSA.

2.3. The Burrows–Wheeler Transform

Let $T[1..n]$ be a string over the alphabet $\Sigma \cup \{\$\}$ where $\$$ is smaller than any symbol in Σ and only occurs in $T[n]$. The *Burrows–Wheeler transform* (BWT) [1] of T is a reversible string transformation that stores in $BWT[j]$ the symbol that precedes the jth suffix of T in lexicographical order, i.e., $BWT[j] = T[SA[j]-1]$ (assuming $T[0] = T[n] = \$$).

The mechanism to revert the transformation is the so-called LF mapping. Given an input position $BWT[j]$ that maps a symbol $T[u]$, $\mathsf{LF}(j) = j'$ returns the index j' such that $BWT[j'] = T[u-1]$ maps the preceding symbol of $T[u]$. Thus, spelling T reduces to continuously applying LF from $BWT[1]$, the symbol to the left of $T[n] = \$$, until reaching $BWT[j] = \$$.

The BCR BWT [12] is a reversible transformation that reorders the symbols of $\mathcal{T} = \{T_1, T_2, \ldots, T_k\}$. Consider again $T = T_1\$_1T_2\$_2 \cdots T_k\$_k$, the sequence of length $n = |T|$ storing the concatenation of the strings in $\mathcal{T}$ separated by unique sentinels $\$_1 \ldots \$_k \notin \Sigma$. Additionally, let us define the vector $GSA[1..n]$ for $\mathcal{T}$ using the order $\$_1 < \$_2 \ldots < \$_k$. The BCR BWT of $\mathcal{T}$ is a vector $BWT_{bcr}[1..n]$ storing in $BWT_{bcr}[j]$ the symbol $T[GSA[j]-1]$. It is worth mentioning that when $T[GSA[j]] = T_x[1]$ maps to the leftmost symbol of a string $T_x \in \mathcal{T}$, $BWT_{bcr}[j] = \$_x$ is (theoretically) the sentinel at the end of T_x in T. Spelling strings in $\mathcal{T}$ from BWT_{bcr} works similarly to the procedure in the standard BWT. Successive rounds of LF operations starting from $BWT_{bcr}[1]$ and finishing when the symbol in $BWT_{bcr}[j']$ is a sentinel spells T_1 from right to left. The same procedure but starting from $BWT_{bcr}[2]$ spells T_2, and so on. Figure 1 depicts an example of BWT_{bcr}.

A common measure of compression for a text is the number of equal-symbol runs in its BWT (denoted r in the literature), but when the text is a collection, the value r associated with its BCR BWT varies depending on the order of the special symbols $\$_1, \ldots, \$_k$. Thus, the optimal BCR BWT (BWT_{opt}) is the transform built with the sentinel ordering that minimizes r. Bentley et al. [31] proposed a linear-time procedure (referred to here as CAO[1]) that receives as input BWT_{bcr} and produces BWT_{opt}. They consider the partition $(s_1, e_1), (s_2, e_2), \ldots, (s_x, e_x)$ of BWT_{bcr} induced by equal suffixes of $\mathcal{T}$. That is, every block $BWT_{bcr}[s_u..e_u]$, with $u \in [1..x]$, stores the left-context symbols of different suffixes of $\mathcal{T}$ that spell the same sequence. They regard the partition as a vector $\mathcal{A}$ where every *uth* element is a tuple collapsing the symbols of $BWT_{bcr}[s_u..e_u]$ by their values. Thus, the *uth* tuple is a sequence $\mathcal{A}[u] = (c_1, \ell_1), \ldots, (c_b, \ell_b)$ of $1 \leq b \leq |\Sigma \cup \{\$\}|$ pairs where (c_p, ℓ_p), with $p \in [1, b]$, groups the ℓ_p occurrences of symbol $c_p \in \Sigma$ within $BWT_{bcr}[s_u..e_u]$.

The key observation to produce BWT_{opt} is that reordering the symbol within each block $BWT_{bcr}[s_x..e_x]$ *only* affects the order in which one spells

[1]The word stands for *Constraint Alphabet Ordering*, the original name Bentley et al. gave to the problem they were studying.

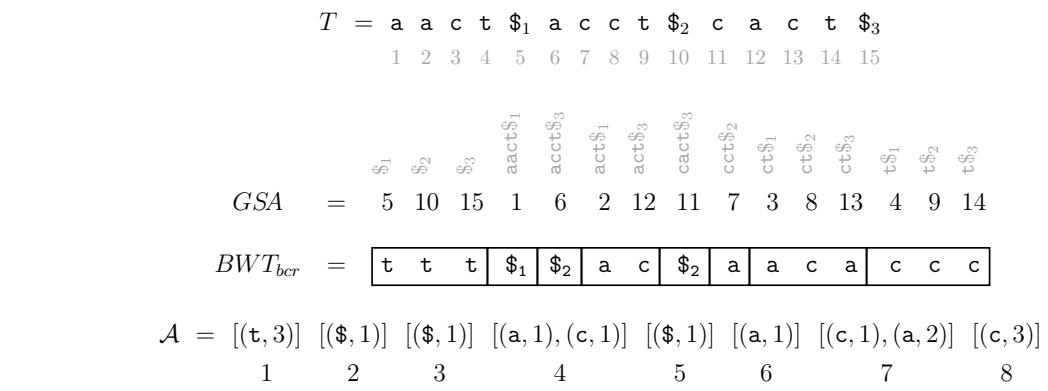

Figure 1: The generalized suffix array and the BCR BWT for the collection $\mathcal{T} = \{\texttt{acct}, \texttt{acct}, \texttt{cact}\}$. The string T is the concatenation of $\mathcal{T}$ separated by sentinel symbols. The boxes in BWT_{bcr} represent the partition induced by equal suffixes of $\mathcal{T}$. The numbers below $\mathcal{A}$ map the blocks in the partition of BWT_{bcr} to tuples in $\mathcal{A}$.

strings in $\mathcal{T}$ from the transform. For example, permuting symbols within $BWT[s_x..e_x]$ might produce that the string T_x the BCR BWT spells from $BWT_{bcr}[1]$ via LF operations is no longer T_1. However, T_x is still a member of $\mathcal{T}$. Bentley et al. noticed that one could minimize r by sorting the pairs of every tuple $\mathcal{A}[j]$, maximizing the number of matches between adjacent tuples. A match between adjacent tuples occurs when the symbol in the rightmost pair of $\mathcal{A}[j]$ equals the symbol of the leftmost pair of $\mathcal{A}[j+1]$.

Example 2.1. Consider the BWT_{bcr} of Figure 1, which has $r = 9$ equal-symbol runs, and its associated vector $\mathcal{A}$ (also in Figure 1). By rearranging the tuple $\mathcal{A}[7] = [(\texttt{c}, 1), (\texttt{a}, 2)]$ to $[(\texttt{a}, 2), (\texttt{c}, 1)]$, the symbol of $(\texttt{a}, 2)$ matches the symbol in $\mathcal{A}[6] = [(\texttt{a}, 1)]$. On the other hand, the symbol of $(\texttt{c}, 1)$ matches the symbol in $\mathcal{A}[8] = [(\texttt{c}, 3)]$. The collapse of $\mathcal{A}$ then produces the run-length-compressed vector $BWT_{opt} = (\texttt{t}, 3)\ (\$, 2)\ (\texttt{a}, 1)\ (\texttt{c}, 1)\ (\$, 1)\ (\texttt{a}, 3)\ (\texttt{c}, 4)$, which has $r = 7$ equal-symbol runs.

CAO finds a sorting for $\mathcal{A}$ that maximizes the number of adjacent matches in linear time. Nevertheless, it does not compute $\mathcal{A}$; it receives it as input. In this regard, Bentley et al. do not give many details on how to compute this vector efficiently. Recently, Cenzato et al. [32] pointed out that it is possible to use the algorithm SA-IS (Section 2.4) to get a bit vector marking the blocks in the partition of BWT_{bcr} induced by equal suffixes (the SAP-array [33]), from which one can easily obtain $\mathcal{A}$.

2.4. Induced Suffix Sorting

Induced suffix sorting (ISS) [26] computes the lexicographical ranks of a subset of suffixes in T and uses the result to induce the order of the rest. This method is the underlying procedure in several algorithms that build the suffix array [34, 35, 36, 37] and the BWT [11, 18] in linear time. The ISS idea introduced by the suffix array algorithm SA-IS of Nong et al. [34] is of interest to this work. The authors give the following definitions:

Definition 2.1. A symbol $T[j]$ is called L-type if $T[j] > T[j+1]$ or if $T[j] = T[j+1]$ and $T[j+1]$ is also L-type. On the other hand, $T[j]$ is said to be S-type if $T[j] < T[j+1]$ or if $T[j] = T[j+1]$ and $T[j+1]$ is also S-type. By definition, the symbol $T[n]$, the one with the sentinel, is S-type.

Definition 2.2. A symbol $T[j]$, with $j \in [1..n]$, is called leftmost S-type, or LMS-type, if $T[j]$ is S-type and $T[j-1]$ is L-type.

Definition 2.3. An LMS substring is (i) a substring $T[j..j']$ with both $T[j]$ and $T[j']$ being LMS-type symbols, and there is no other LMS symbol in the substring, for $j \neq j'$; or (ii) the sentinel itself.

SA-IS recursively sorts a subset of suffixes of $\mathcal{T}$ using ISS and then uses the result to induce the relative order of another set of suffixes. The recursion continues until there are no more suffixes of $\mathcal{T}$ to sort. When that happens, it returns from the recursion merging the distinct subsets of suffixes it processed before, producing the final suffix array SA of T when it reaches the first level of the recursion again. The key idea that makes SA-IS linear-time is that every recursion level operates over a sequence that is at most half the length of the sequence in the previous level.

The suffixes that SA-IS processes in every level i are those prefixed by LMS substrings. The level receives as input a string $T^i[1..n^i]$ over an alphabet Σ^i (when $i = 1$, $T^i = T$ and $\Sigma^i = \Sigma$) and initializes an empty suffix array $SA^i[1..n^i]$. Then, it scans T^i from right to left to classify the symbols as L-type, S-type, or LMS-type. As it moves through the text, the algorithm records the text positions of the LMS substrings in SA^i. More specifically, if $T^i[j] = b$ is the leftmost symbol of an LMS substring, it inserts j in the rightmost empty position in the bucket b of SA^i. After scanning T^i, SA-IS sorts the LMS substrings in SA^i using two linear scans.

Algorithm 1 SA-IS

Require: $T^i[1..n^i]$
1: $SA^i[1..n_i] \leftarrow$ empty array
2: scan T^i right to left an insert LMS-type positions in SA^i
3: scan SA^i twice to sort the LMS substrings with different sequences
4: **if** all the symbols in T^i are distinct **then**
5: **return** SA^i
6: **end if**
7: scan SA^i to get the set $\mathcal{F}^i$ sorted in LMS order
8: $o \leftarrow 1$
9: **for** $F \in \mathcal{F}^i$ **do**
10: assign symbol $o \in \Sigma^{i+1}$ to F
11: $o \leftarrow o + 1$
12: **end for**
13: $T^{i+1} \leftarrow$ replace LMS substrings in T^i with their assigned symbols in Σ^{i+1}
14: $SA^{i+1} \leftarrow$ SA-IS(T^{i+1})
15: $SA^i \leftarrow$ store LMS-type positions of T^i as their symbol in Σ^{i+1} appear in SA^{i+1}
16: scan SA^i twice again to the sort the suffixes of T^i
17: **return** SA^i

Scans of the Suffix Array. SA-IS sorts the suffixes of T^i prefixed by different LMS substrings in two linear scans of SA^i. The first scan traverses the array left to right, and for each index j such that $T^i[SA^i[j] - 1] = c$ is L-type, it inserts the text position $SA^i[j] - 1$ in the leftmost empty cell of bucket c in SA^i. The second scan traverses SA^i from right to left. This time, for each index j such that $T^i[SA^i[j]] = c$ is S-type, it stores the text position $SA^i[j]-1$ in the rightmost empty cell in the bucket c of SA^i.

ISS rearranges the LMS substrings in SA^i in a slightly different way from the lexicographic order. Let $F_x[1..n_x]$ and $F_y[1..n_y]$ be two strings over Σ^i labelling LMS substrings of T^i, and let $\prec_{lex}$ be the operator that denotes lexicographical order. LMS order ($\prec_{LMS}$) is defined as

$$F_x \prec_{LMS} F_y \Leftrightarrow \begin{cases} n_y < n_x \text{ and } F_x[1..n_y] = F_y \\ F_x \prec_{lex} F_y \text{ otherwise.} \end{cases}$$

For instance, it holds that `actca` $\prec_{LMS}$ `actc`, which differs from the standard `actc` $\prec_{lex}$ `actca`. However, for two occurrences $F_y = T^i[j..j']$ and $F_x = T^i[u..u']$, the higher LMS order of F_y implies that the suffix $T^i[j..n^i]$ is lexicographically greater than the suffix $T^i[u..n^i]$. The cause of this property

is explained in Section 2 of Ko and Aluru [26].

The idea now is to use the sorted LMS substrings to induce the order of the suffixes in T^i that are not prefixed by LMS substrings. Still, the relative order of LMS substrings with the same sequence is unknown in SA^i. Nong et al. solve this problem by creating a new string T^{i+1} in which they replace the LMS substrings with their LMS orders. Let $\mathcal{F}^i$ be the set of strings labelling LMS substrings in T^i. If a string $F \in \mathcal{F}^i$ has LMS order o in the set, then SA-IS replaces the LMS substrings of T^i labelled F by o. Notice that now it is possible to compute the LMS orders in one scan of SA^i. The resulting sequence T^{i+1} is used as input for another recursive call $i+1$. The base case for the recursion is when all the suffixes in SA^i are prefixed by different symbols, in which case the algorithm returns SA^i without further processing.

When the $(i+1)th$ recursive call ends, all the suffixes of T^i prefixed by the same LMS substrings are sorted in SA^{i+1}, so SA-IS proceeds to complete SA^i. For doing so, it resets SA^i, inserts the LMS substrings arranged as their respective symbols appear in SA^{i+1}, and performs the two scans of paragraph 2.4 to reorder the unsorted suffixes of T^i. Once it finishes, it passes SA^i to the previous recursion $i-1$. The final array SA^1 is the suffix array for T. Algorithm 1 explains the general idea of SA-IS. We also refer the reader to Figure 2 in Louza et al. [36] for a detailed running example of a recursion level in SA-IS.

3. Methods

3.1. Definitions

We assume the standard string notation. For a string $T[1..n]$ over an arbitrary alphabet, $T[1..j]$ is the jth prefix of T, and $T[j..n]$ is the jth suffix. Additionally, the operator $|T| = n$ represents the number of symbols in T (i.e., its length). We will refer to $T[j..n]$ as a *proper* suffix if $1 < j \leq n$, and non-proper otherwise. When T is run-length-compressed, we use the format c^ℓ to denote an equal-symbol run of ℓ copies of c. We also use the alternative notation (c, ℓ) with the same meaning. We choose one format or the other depending on the context. Additionally, we use ε as the empty symbol.

Let $\mathcal{S} = \{S_1, \ldots, S_z\}$ be a string set. We consider a suffix $S_x[j..n_x]$ to be *left-maximal* if there is at least one other string $S_y[1..n_y] \neq S_x \in \mathcal{S}$ with a suffix $S_y[j'..n_y]$ such that (i) $S_y[j'..n_y] = S_x[j..n_x]$, and (ii) either both $S_y[j'..n_y]$ and $S_x[j..n_x]$ are proper suffixes with $S_y[j'-1] \neq S_x[j-1]$, or one of them ($S_x[j..n_x]$ or $S_y[j'..n_y]$) is not proper. We will use the operator $|\mathcal{S}|$

for the total number of strings in $\mathcal{S}$ and $||\mathcal{S}||$ to express the total number of symbols $\sum_{S_x \in \mathcal{S}} |S_x|$.

Through the paper, we use the superscript i to denote elements of the ith recursion level in SA-IS. Thus, the string $T^i[1..n^i]$ over the alphabet $\Sigma^i[1..\sigma^i]$ is the input for level i. Notice that, for instance, σ^i does not mean $\sigma = \sigma^1$ raised to i. The same idea applies to other elements with the i superscript. The expressions T and T^1 are equivalent. The same holds for Σ and Σ^1.

For a given position $T^i[j]$, with $j \in [1..n_i]$, the operator $exp^u(T^i[j]) \in \Sigma^i$, with $u < i$, denotes the string we obtain by recursively replacing the symbol $o = T^i[j] \in \Sigma^i$ with its associated phrase $F = o_1 \cdots o_r \in \mathcal{F}^{i-1}$ over the alphabet Σ^{i-1}. The formal recursive definition of exp^u is as follows:

$$exp(o)^u \Leftrightarrow \begin{cases} o \text{ if } o \in \Sigma^u \\ exp^{u-1}(o_1) \cdots exp^{u-1}(o_{r-1}) \text{ if } exp^1(o_r) \text{ does not end with \$} \\ exp^{u-1}(o_1) \cdots exp^{u-1}(o_r) \text{ if } exp^1(o_r) \text{ ends with \$} \end{cases}$$

Note that exp^u removes the overlap between consecutive LMS substrings of T^1. Additionally, the function $map^u(T^i[j]) = T^u[z..z']$, with $u < i$, returns the substring in T^u from where $T^i[j]$ was formed. Formally, the boundaries (z, z') in $map^u(T^i[j])$ are

$$z = 1 + \sum_{q=1}^{j-1} |exp^u(T^i[q])|$$
$$z' = z + |exp^u(T^i[j])| - 1.$$

The expressions $exp(T^i[j])$ and $map(T^i[j])$ are equivalent to $exp^1(T^i[j])$ and $map^1(T^i[j])$, respectively.

We use the term *parsing* to refer to a procedure that breaks $T^i[1..n^i]$ into a sequence of (possibly overlapping) substrings. The strings labelling the substring are the *phrases* of the parsing. On the other hand, we use the term *dictionary* to refer to an abstract associative data structure where the keys are strings and the associated values are integers.

Let $\Sigma = [1..\sigma]$ be an alphabet of σ symbols and let $\mathcal{T} = \{T_1, \ldots, T_k\}$ be a collection of k strings over the alphabet $\Sigma \setminus \{1\}$. The input for our algorithm is the sequence $T = T_1\$T_2 \ldots T_k\$ \in \Sigma^*$ of total length $n = |T|$ representing the concatenation of $\mathcal{T}$. The symbol \$ is a sentinel that we use as a boundary between consecutive strings in T. We set $\$ = 1$ to the smallest symbol in Σ.

3.2. Overview of Our Algorithm

We call our algorithm for computing the BCR BWT of $\mathcal{T}$ grlBWT. This method relies on the ideas developed by Nong et al. in the SA-IS algorithm (Section 2.4) but includes elements of grammar and run-length compression (Section 2.1 to reduce the space usage of the temporary data that grlBWT maintains in memory. Overall, this idea allows us to decrease working memory and computing time.

Similar to SA-IS, our method grlBWT is recursive. We start by parsing T^1 using a mechanism that relies on the symbol types of Section 2.4 and stores the resulting parsing phrases in a set $\mathcal{F}^1$. Then, we use the set $\mathcal{S}^1$ with the strings labelling the suffixes of $\mathcal{F}^1$ to partition SA^1 such that each block $SA^1[s_x..e_x]$ encodes the suffixes of T^1 prefixed by the same string $S_x \in \mathcal{S}^1$. An important observation is that if S_x is always preceded by the same symbol c in the parsing phrases, we can compute the associated substring $BWT^1_{bcr}[s_x..e_x] = c^\ell$, with $\ell = e_x - s_x + 1$, directly from $\mathcal{F}^1$. We use this idea to create a sparse version of BWT_{bcr} that only contains symbols for blocks that meet our observation. We leave the other areas of BWT_{bcr} empty for the moment. We refer to the sparsely populated version of BWT_{bcr} as the *preliminary* BCR BWT of T^1 ($pBWT^1$). To fill the *unsolved* areas of $pBWT^1$, we create another string T^2 by replacing the substrings in the parsing of T^1 with their associated LMS orders in $\mathcal{F}^1$, and then apply the same procedure recursively over T^2. We keep recursing until we reach a base-case level h where the input T^h has k symbols (i.e., the number of strings in $\mathcal{T}$). At this point, the BCR BWT of T^h (BWT^h_{bcr}) is T^h itself (we explain this idea in Section 3.5). We refer to the process of recursing from level 1 to level h as the *parsing phase* of grlBWT.

The parsing phase produced the string BWT^h_{bcr}, the preliminary BCR BWTs $pBWT^1, pBWT^2, \ldots, pBWT^{h-1}$, and the sets $\mathcal{F}^1, \mathcal{F}^2, \ldots, \mathcal{F}^{h-1}$. Now we need to go back in the recursions to complete the execution of grlBWT. When we return to level $i < h$, we use BWT^{i+1}_{bcr} and $\mathcal{F}^i$ to induce the symbols in the unsolved blocks of $pBWT^i$, thus producing BWT^i_{bcr} (the BCR BWT of T^i). We keep BWT^{i+1}_{bcr} and $pBWT^i$ in run-length-compressed format to reduce space usage and speed up the computation of BWT^i_{bcr}. Our induction procedure reads a position $BWT^{i+1}_{bcr}[j]$ and uses its information to insert symbols in multiple unsolved blocks of $pBWT^i$. However, equal symbols of BWT^{i+1}_{bcr} yield the same information and for the same unsolved blocks of $pBWT^i$. Thus, if we have a run $BWT^{i+1}_{bcr}[j..j+\ell-1] = o^\ell$ of ℓ consecutive copies of $o \in \Sigma^{i+1}$, we perform the induction from o only once and copy the

Algorithm 2 Overview of grlBWT

Require: $T[1..n]$, k ▷ T is the concatenation of the k string in $\mathcal{T}$

$i \leftarrow 1$
$T^1 \leftarrow T$
while $length(T^i) \neq k$ **do** ▷ parsing phase
    Build $\mathcal{F}^i$ by parsing T^i
    Produce $pBWT^i$ from $\mathcal{F}^i$ and satellite data
    Encode $\mathcal{F}^i$ using grammar compression
    Build T^{i+1} using $\mathcal{F}^i$ and T^i
    Store $pBWT^i$ and $\mathcal{F}^i$ on disk
    $i \leftarrow i + 1$
end while
$BWT^i_{bcr} \leftarrow T^i[1..k]$ ▷ induction phase
$i \leftarrow i - 1$
while $i > 0$ **do**
    Load $\mathcal{F}^i$ from disk to main memory
    Build P^i using BWT^{i+1}_{bcr} and $\mathcal{F}^i$
    Merge P^i and $pBWT^i$ to produce BWT^i_{bcr}
    $i \leftarrow i - 1$
end while
return BWT^1_{bcr} ▷ the BCR BWT of $\mathcal{T}$

result in $pBWT^i$ ℓ times. When we reach the recursion level $i = 1$ again, BWT^1_{bcr} becomes the BCR BWT of $\mathcal{T}$. We refer to the process of returning from recursion h to recursion 1 as the *induction phase* of grlBWT.

Practical Considerations. We implement grlBWT as a semi-external algorithm that executes the recursion levels as iterations in a loop. In every iteration i, we perform the steps of recursion level i, keeping the information of the other levels on disk. When we execute iteration i in the parsing phase, we scan T^i linearly from the disk to produce $\mathcal{F}^i$. The only elements that are always in main memory are $\mathcal{F}^i$ and a couple of satellite vectors that help us to build $pBWT^i$, which is also accessed linearly from disk during its construction. Then, we compute T^{i+1} from $\mathcal{F}^i$ externally and store $\mathcal{F}^i$ on disk to use it later. We encode $\mathcal{F}^i$ with a scheme similar to grammar compression to reduce space usage and facilitate the remaining computations. When we return to level i in the induction phase, we access BWT^{i+1}_{bcr} and $pBWT^i$ linearly from the disk. The only elements in main memory are $\mathcal{F}^i$ and a run-

length-compressed vector P^i that keeps the symbols we need to introduce in the unsolved blocks of $pBWT^i$. We produce BWT^i_{bcr} by merging $pBWT^i$ and P^i in a semi-external way. Algorithm 2 presents a general overview of grlBWT.

3.3. The Parsing Phase

This section explains the steps we perform in one iteration of the parsing phase (Lines 3-10 of Algorithm 2). The inputs for the iteration are T^i and a bit vector $B^i[1..\sigma^i]$ that indicates with $B^i[c] = 1$ if and onlu if symbol $c \in \Sigma$ expands to a string $exp(c) \in \Sigma^*$ suffixed by the sentinel \$. The output of the iteration is the preliminary BCR BWT of T^i ($pBWT^i$) and a compressed version of $\mathcal{F}^i$.

3.3.1. LMS parsing

We start the iteration by producing the set $\mathcal{F}^i$ with the distinct parsing phrases of T^i (Line 4 of Algorithm 2). Our mechanism to break T^i uses the symbol types of Section 2.4, but includes some modifications to take into consideration that T^i is a (compressed) collection rather than a single string.

Definition 3.1. LMS breaks: given a string T^i over the alphabet $\Sigma^i[1..\sigma^i]$, its sequence of LMS breaks is a set of strictly increasing integers $\mathcal{B}^i = \{1 < j_1 < j_2 < \ldots < j_z < n^i < n^i + 1\}$ such that each $T^i[j_p]$, with $p \in [1..z]$, is either (i) an LMS-type position, (ii) a position where $map(T^i[j_p]) = T^1[u..u']$ is suffixed by a sentinel $\$ = T^1[u']$, or (iii) a position where $map(T^i[j_p]) = T^1[u..u']$ is preceded by a sentinel $T^1[u-1] = \$$. Positions $1, n^i + 1$, and those meeting condition (iii) are left-border breaks. On the other hand, n^i and the positions meeting condition (ii) are right-border breaks.

Definition 3.2. LMS parsing: the parsing of T^i induced by consecutive breaks of $\mathcal{B}^i$. Let $(j_p, j_{p+1}) \in \mathcal{B}^i$ be two consecutive breaks. They form the substring $T^i[j_p..j_{p+1}]$ in the LMS parsing if the following conditions hold: (i) $T^i[j_p]$ is a left-border break or an LMS-type position, and (ii) $T^i[j_{p+1}]$ is a right-border break or an LMS-type position. Additionally, (j_p, j_{p+1}) forms the phrase $T^i[j_p..j_{p+1} - 1]$ instead of $T^i[j_p..j_{p+1}]$ iff $j_{p+1} = j_p + 1$ and both $T^i[j_p]$ and $T^i[j_{p+1}]$ are left-border breaks. If (j_p, j_{p+1}) does not meet any of the conditions above, it does not produce a substring in the LMS parsing.

Intuitively, our definition of LMS parsing prevents the formation of phrases in T^i that cover multiple strings of $\mathcal{T}$. Specifically, the parsing will never

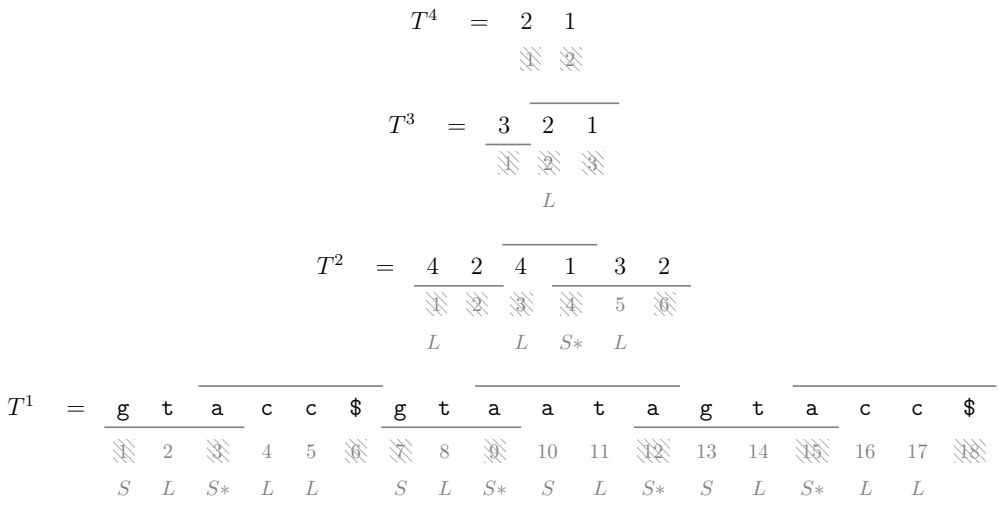

Figure 2: Successive rounds of LMS parsing for the text T^1 = `gtacc$gtaatagtacc$`. The symbols L, S and $S*$ represent the L-type, S-type and LMS-type positions of T^i, respectively. The dashed boxes indicate the breaks in $\mathcal{B}^i$ (Definition 3.1). Positions 1 and 7 of T^1 are left-border breaks, while positions 6 and 18 are right-border breaks. Similarly, positions 1 and 3 of T^2 are left-border breaks, and positions 2 and 6 are right-border breaks. The horizontal lines indicate the substrings induced by $\mathcal{B}^i$ as described in Definition 3.2. Each $T^i[j]$ is the LMS order in $\mathcal{F}^i$ of the jth substring in the LMS parsing of T^{i-1}. The parsing rounds end when the number of symbols in T^i equals the number of strings in $\mathcal{T}$.

yield a substring $T^i[j..j']$ such that $map(T^i[j]) = T^1[u..u']$ falls within the boundaries of a string $T_x \in \mathcal{T}$ and $map(T^i[j']) = T^1[o..o']$ falls within the boundaries of another string $T_y \in \mathcal{T}$. Figure 2 depicts an example of LMS parsing.

Our procedure to build $\mathcal{F}^i$ works as follows: we initialize a dictionary D^i where each key $F \in \mathcal{F}^i$ in D^i will be associated with an integer that indicates the number of times F occurs as a phrase in the LMS parsing of T^i. We fill the dictionary by accessing the breaks in $\mathcal{B}^i$ in one right-to-left scan of T^i that enumerates the LMS-type positions. We also use B^i during the scan to detect right and left border breaks. For each pair of consecutive integers $(j_p, j_{p+1}) \in \mathcal{B}^i$ forming a valid substring in the LMS parsing (see Definition 3.2), we record an occurrence of the phrase $F = T^i[j_p..j_{p+1}]$ in D^i (or $F = T^i[j_p..j_{p+1} - 1]$ if both positions are left-border breaks). If F does not exist as a key, we create a new key-value entry $(F, 1)$ in D^i. On the other hand, if F already exists in the keys, we just increase its associated value by

one.

3.3.2. The Generalized Suffix Array

As explained in the overview of grlBWT (Section 3.2), every recursion level i constructs the vector BWT^i_{bcr} of T^i (besides performing LMS parsing). This step requires us to define the generalized suffix array GSA^i of T^i to fit the description of the BCR BWT.

Definition 3.3. Generalized suffix array of T^i. Consider a partition over $T^i[1..n^i]$ such that each block $T^i[j..j']$ is a substring where (i) $T^i[j]$ is a left-border break in $\mathcal{B}^i$, (ii) $T^i[j']$ is a right-border break in $\mathcal{B}^i$, and (iii) $exp(T^i[j..j']) = T_x\$$ expands to a string $T_x \in \mathcal{T}$. $GSA^i[1..n^i]$ is a vector that enumerates the suffixes of the blocks $T^i[j..j']$ in lexicographical order. When two suffixes $T^i[u..j'] = T^i[u'..p']$ from different blocks $T^i[j..j']$ and $T^i[p..p']$ spell the same sequence, the ties are broken by $min(u, u')$.

We remark that our algorithm never constructs GSA^i, but it uses its definition to produce BWT^i_{bcr}. Figure 4 shows an example of GSA^i.

3.3.3. Constructing the Preliminary BCR BWT

The next step in iteration i is to construct the preliminary BCR BWT of T^i from D^i (Line 5 of Algorithm 2). Keep in mind that D^i is an encoding for $\mathcal{F}^i$ that includes the frequencies in T^i of the LMS parsing phrases. We use the following observations to carry out the construction:

Lemma 3.1. Let $S_x[1..n_x]$ and $S_y[1..n_y]$ be two different strings over the alphabet Σ^i. Assume both occur as suffixes in one or more phrases of $\mathcal{F}^i$ and meet one of the following conditions: (i) $n_x > 1$ or (ii) $n_x = 1$ and the sequence $exp(S_x)$ is suffixed by the sentinel \$ (the same applies for S_y). Let O_x be the set of positions in T^i where S_x occurs as a suffix of a parsing phrase. Specifically, each $j \in O_x$ is a position such that $S_x = T^i[j..j+n_x-1]$ and $T^i[j-j'..j+n_x-1]$, with $j' \geq 0$, is a substring in the LMS parsing. Let us define an equivalent set O_y for S_y. If $S_x \prec_{LMS} S_y$ (see Section 2.4), then all the suffixes of T^i starting at positions in O_x are lexicographically smaller than those starting at positions in O_y.

Proof. When S_x is not a prefix of S_y (and vice versa), the demonstration of this lemma is trivial: we compare the sequences of S_x and S_y from left to right until we find a mismatching position u (i.e., $S_x[u] \neq S_y[u]$). If $S_x[u] < S_y[u]$,

we know that all the suffixes in O_x are lexicographically smaller than those in O_y because this is how lexicographical sorting works. In the other scenario, when one string is a prefix of the other, we can not use this mechanism as we will not find a mismatching position. However, this scenario is not possible when S_x or S_y has length 1 and expands to a string in Σ^* suffixed by \$. For instance, if S_x were a prefix of S_y, it would imply that $exp(S_y)$ contains a sentinel in a position that is not the end, but Definition 3.2 (LMS parsing) prevents that situation from happening.

When both S_x and S_y have length > 1 and one is a prefix of the other, we resort to the symbol types of Section 2.4. We assume for this proof that S_y is a prefix of S_x, but the other way is equivalent. We know that $S_y[n_y]$ and $S_x[n_y]$ have different types. On the one hand, $S_y[n_y]$ is LMS-type because S_y is a suffix of an LMS substring, and n_y is the last position of S_y. On the other hand, $S_x[n_y]$ is L-type because if it were S-type, then it would also be LMS-type, and thus $S_x[1..n_y]$ would be an occurrence for S_y. This observation is due to $S_y[n_y - 1] = S_x[n_y - 1]$ being L-type. Given the types of $S_y[n_y]$ and $S_x[n_y]$, the occurrences of S_y in O_y are always followed in T^i by symbols that are greater than $S_x[n_y + 1]$, meaning that the suffixes of T^i starting at positions in O_y are lexicographically greater than the suffixes starting at positions in O_x. This observation does not hold when S_x or S_y have length one: $S_y[n_y]$ equals $S_x[1]$, and both are LMS-type, so there is not enough information to decide the lexicographical order of the suffixes in O_x and O_y. Figure 3 shows an example of this lemma. □

The consequence of Lemma 3.1 is that the suffixes of length > 1 in $\mathcal{F}^i$ induce a partition over GSA^i (Definition 3.3):

Lemma 3.2. Let $\mathcal{S}^i = \{S_1, S_2, \ldots, S_{n'}\}$ be the set of strings where each element $S_x[1..n_x]$ occurs as a suffix in the phrases of $\mathcal{F}^i$ and either (i) has length $n_x > 1$ or (ii) has length $n_x = 1$ but $exp(S_x)$ is suffixed by \$. Additionally, let $\mathcal{O} = \{O_1, O_2, \ldots, O_{n'}\}$ be the set of occurrences in T^i for the strings in $\mathcal{S}$. For every $S_x \in \mathcal{S}^i$, its associated set $O_x \in \mathcal{O}$ stores each position j such that $T^i[j..j + n_x - 1]$ is an occurrence of S_x and $T^i[j - j'..j + n_x - 1]$, with $j' \geq 0$, is a phrase in the LMS parsing. It holds that $\mathcal{O}$ is a partition over GSA^i as the lexicographical sorting places the elements of each $O_x \in \mathcal{O}$ in a consecutive range of GSA^i.

Proof. We demonstrate the lemma by showing that the lexicographical sorting does not interleave suffixes of T^i in GSA^i that belong to different sets of

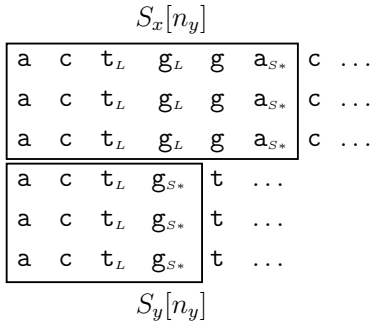

Figure 3: Example of Lemma 3.1. The figure shows a subset of suffixes for some string T^1 sorted in lexicographical order. The upper box is the set O_x of suffixes in T^i prefixed by the string S_x = `actgga` that occurs as a suffix in some LMS parsing phrases of T^1. Equivalently, the lower box is the set O_y of suffixes prefixed by S_y = `actg`, which also occurs as a suffix in some LMS substrings of T^1. The L-type and LMS-type positions of T^1 are marked with the symbols L and $S*$ in the figure, respectively. Because $S_x \prec_{LMS} S_y$, all the suffixes in O_y are lexicographically greater than those in O_x.

$\mathcal{O}$. Assume the string $S_x \in \mathcal{S}^i$, associated with the set $O_x \in \mathcal{O}$, is a prefix in another string $S_y \in \mathcal{S}^i$, which in turn is associated with the set $O_y \in \mathcal{O}$. Even though we do not know the symbols that occur to the right of S_x in its occurrences of O_x, we do know that both S_x and S_y are suffixes of substrings in the LMS parsing, and by Lemma 3.1, we know that all the suffixes of T^i in O_x are lexicographically greater than the suffixes in O_y. Hence, the interleaving of suffixes in GSA^i from different sets of $\mathcal{O}$ is not possible, even if $\mathcal{S}^i$ is not a prefix-free set. □

Lemma 3.2 gives us a simple way to construct the preliminary BCR BWT of T^i from $\mathcal{F}^i$. Our explanation requires projecting the partition of GSA^i induced by $\mathcal{S}^i$ to BWT^i_{bcr} (the BCR BWT of $\mathcal{T}^i$) such that, if $GSA^i[s_x..e_x]$ is the block formed by $S_x \in \mathcal{S}^i$, then $BWT^i_{bcr}[s_x..e_x]$ is the projected block. There are three cases to consider:

Lemma 3.3. Let $S_x \in \mathcal{S}^i$ be the string with LMS order o among all the elements of $\mathcal{S}^i$. If S_x is left-maximal in $\mathcal{F}^i$, then the oth block $BWT^i_{bcr}[s_x..e_x]$ in the projected partition contains more than one distinct symbol. Therefore, $\mathcal{F}^i$ does not have enough information to compute the sequence of symbols in $BWT^i_{bcr}[s_x..e_x]$.

Proof. Let F and F' be two phrases of $\mathcal{F}^i$ where S_x occurs as a suffix. Assume the left symbol of S_x in F is $c \in \Sigma^i$ and the left symbol in F' is $c' \in \Sigma^i$.

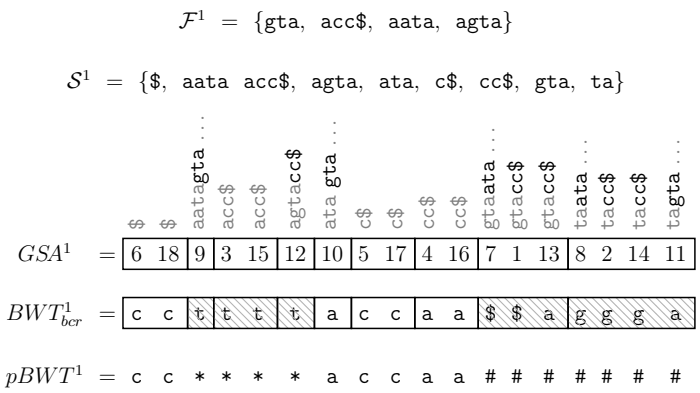

Figure 4: Preliminary BCR BWT of $\mathcal{T}$ = {gtacc, gtaatagtacc} of Figure 2. GSA^i is the generalized suffix array of $\mathcal{T}$ built on top of T^1 = gtacc$gtaatagtacc$. The vertical strings are the suffixes sorted in lexicographical order. The boxes in GSA^i indicate the partition induced by $\mathcal{S}^i$ (strings labelled in grey on top of GSA^i). The boxes in BWT^1_{bcr} are the projected blocks in GSA^i's partition, with the dashed boxes being the blocks we can not solve using $\mathcal{F}^1$. The first three dashed boxes (from left to right) of BWT^i_{bcr} are represented with * in $pBWT^1$ because their blocks meet Lemma 3.4. In contrast, the last two dashed boxes are represented with #, indicating they meet Lemma 3.3. The final run-length-compressed vector is $pBWT^1 = (\mathtt{c}, 2), (*, 4), (\mathtt{a}, 1), (\mathtt{c}, 2), (\mathtt{a}, 2), (\#, 7)$.

In this scenario, the relative order of c and c' is not decided by S_x, but for the sequences that occur to the right of F and F' in T^i. However, those sequences are not accessible directly from $\mathcal{F}^i$. Hence, it is not possible to decide the order of c and c' in $BWT^i_{bcr}[s_x..e_x]$. □

Lemma 3.4. Consider the string $S_x \in \mathcal{S}^i$ of Lemma 3.3. When S_x only occurs as a non-proper suffix in a phrase $S_x = F \in \mathcal{F}^i$, it is not possible to complete the sequence of symbols in $BWT^i_{bcr}[s_x..e_x]$.

Proof. The symbols that occur to the left of S_x in T^i are in the substrings of the LMS parsing that precede F in T^i. However, it is not possible to know from $\mathcal{F}^i$ the sequences of those substrings. □

We now describe the information of the preliminary BCR BWT that we can extract from $\mathcal{F}^i$:

Lemma 3.5. Let $S_x \in \mathcal{S}^i$ be the string of Lemma 3.3. Additionally, let $O_x \in \mathcal{O}$ be the set of occurrences of S_x in T^i as described in Lemma 3.2. If all the suffixes of T^i in O_x are preceded by the same symbol $c \in \Sigma^i$ (i.e., S_x

is not left-maximal), $BWT^i_{bcr}[s_x..e_x] = c^\ell$ is an equal-symbol run of length $\ell = |O_x|$.

Proof. By Lemma 3.2, we know that S_x prefixes the suffixes of T^i in O_x, and that they form a consecutive range $GSA^i[s_x..e_x]$. Additionally, the symbols that occur to the left of the suffixes in $GSA^i[s_x..e_x]$ are those for the projected block $BWT^i_{bcr}[s_x..e_x]$. However, we still have not resolved the relative order of the suffixes in $GSA^i[s_x..e_x]$, so (in theory) we do not know how to rearrange the symbols in $BWT^i_{bcr}[s_x..e_x]$. The suffixes of T^i in O_x are preceded by the same symbol c, so it is not necessary to further sort $GSA^i[s_x..e_x]$ because the outcome for the projected block will always be an equal-symbol run c^ℓ of length $\ell = |O_x| = e_x - s_x + 1$. □

Now we have all the necessary elements to describe the preliminary BCR BWT formally:

Definition 3.4. Preliminary BCR BWT of $\mathcal{T}^i$: a vector $pBWT^i[1..n^i]$ over the alphabet $\Sigma^i \cup \{\#, *\}$, where # and * are special symbols out of Σ^i denoting *unsolved* areas of BWT^i_{bcr} for which we do not know the order of the symbols. Let $S_x \in \mathcal{S}^i$ be the string inducing the projected block $BWT^i_{bcr}[s_x..e_x]$, with $\ell = e_x - s_x + 1$. If $BWT^i_{bcr}[s_x..e_x]$ meets Lemma 3.3, then $pBWT^i[s_x..e_x] = \#^\ell$. On the other hand, if $BWT^i_{bcr}[s_x..e_x]$ meets Lemma 3.4, then $pBWT^i[s_x..e_x] = *^\ell$. Finally, if $BWT^i_{bcr}[s_x..e_x]$ meets Lemma 3.5 and is always preceded by symbol $c \in \Sigma^i$ in $\mathcal{F}^i$, then $pBWT^i[s_x..e_x] = c^\ell$. The areas of $pBWT^i$ storing # or * symbols are its unsolved blocks.

We use two special symbols $\$, \# \notin \Sigma^i$ because the mechanism to fill the unsolved blocks of $pBWT^i$ that meet Lemma 3.3 during the induction phase is different from the one to fill the unsolved blocks meeting Lemma 3.4. The difference will become clear in Section 3.5. Additionally, we keep $pBWT^i$ in run-length-compressed format to reduce space usage. Despite our encoding, we will keep using the notation $pBWT^i[s_x..e_x]$ to refer to an uncompressed area of $pBWT^i$. Figure 4 shows an example of the construction of $pBWT^i$ using Lemmas 3.3,3.4, and 3.5.

Succinct Dictionary Encoding. The construction of $pBWT^i$ starts by changing the representation of D^i to a more convenient data structure. First, we concatenate all the keys of D^i in one single vector $R^i[1..||\mathcal{F}^i||]$ without changing their relative order. We mark the boundaries of consecutive phrases in R^i with a bit vector $L^i[1..||\mathcal{F}^i||]$ in which we set $L^i[j] = 1$ if $R^i[j]$ is the

first symbol of a phrase and set $L^i[j] = 0$ otherwise. We store the values of D^i in another integer vector $N^i[1..|\mathcal{F}^i|]$. We maintain the relative order so that the value $N^i[o]$ is associated in D^i with *oth* phrase we inserted into R^i. We also augment L^i with a data structure that supports rank_1 queries [38] to map each symbol $R^i[j]$ to its corresponding phrase in D^i. Thus, given the phrase $F = R^i[j..j'] \in \mathcal{F}^i$, we can obtain its associated value in D^i as $N^i[\mathsf{rank}_1(L, p)]$, with $p \in [j..j']$.

Generalized Suffix Array of the Parsing Set. We will use (R^i, L^i) to compute an array $SA_{\mathcal{F}^i}[1..|R^i|]$ that enumerates the suffixes of $\mathcal{F}^i$ in LMS order. This vector serves a double purpose as we will use it to get the unsolved blocks of $pBWT^i$ and the LMS orders of the phrases in $\mathcal{F}^i$. The values in $SA_{\mathcal{F}^i}$ are text positions in R^i, which we sort as follows: consider two substrings $R^i[j..j']$ and $R^i[p..p']$ encoding suffixes of $\mathcal{F}^i$ that are consecutive in $SA_{\mathcal{F}^i}$. That is, $SA_{\mathcal{F}^i}[u] = j$, $SA_{\mathcal{F}^i}[u+1] = p$, $L^i[j'+1] = 1$, and $L^i[p'+1] = 1$. If $R^i[j..j'] \neq R^i[p..p']$, then $R^i[j..j'] \prec_{LMS} R^i[p..p']$. On the other hand, if $R^i[j..j'] = R^i[p..p']$, it implies $j < p$. We compute $SA_{\mathcal{F}^i}$ using a modified version of the ISS method described in Section 2.4. We first create an empty array $SA_{\mathcal{F}^i}[1..|R^i|]$, which we divide into Σ^i buckets. Then, we scan R^i from left to right, and for each symbol $R^i[j] = c$ with $L^i[j+1] = 1$ (i.e., the rightmost symbol of a phrase), we insert j in the rightmost empty position in the bucket c of $SA_{\mathcal{F}^i}$. Then, in one left-to-right scan of $SA_{\mathcal{F}^i}$, we perform the first step of ISS: for every $SA_{\mathcal{F}^i}[u]$ such that $R^i[SA_{\mathcal{F}^i}[u]-1] = c$ is L-type, we insert $SA_{\mathcal{F}^i}[u] - 1$ in the leftmost empty cell in the bucket c of $SA_{\mathcal{F}^i}$. In the next ISS step, we perform a right-to-left scan of $SA_{\mathcal{F}^i}$. This time, for every $SA_{\mathcal{F}^i}[u]$ such that $R^i[SA_{\mathcal{F}^i}[u] - 1] = c$ is S-type, we insert $SA_{\mathcal{F}^i}[u] - 1$ in the rightmost empty cell in the bucket c of $SA_{\mathcal{F}^i}$. In both ISS scans, if $L^i[SA_{\mathcal{F}^i}[u]] = 1$ (i.e., $R^i[SA_{\mathcal{F}^i}[u]]$ is the leftmost symbol of a phrase), we skip the position as it does not induce the order of any suffix in $\mathcal{F}^i$.

Computing $pBWT^i$ in Compressed Space. Algorithm 3 produces $pBWT^i$ from $SA_{\mathcal{F}^i}$, $D^i = (R^i, L^i, N^i)$, and B^i. We consider the partition of $SA_{\mathcal{F}^i}$ where every block $SA_{\mathcal{F}^i}[s_x..e_x]$ encodes the suffixes that come from different phrases of $\mathcal{F}^i$ but spell the same sequence $S_x \in \Sigma^{i2}$. For $SA_{\mathcal{F}^i}[s_x..e_x]$, we

[2]In practice, we compute every distinct block $SA_{\mathcal{F}^i}[s_x..e_x]$ during the construction of $SA_{\mathcal{F}^i}$. We reserve the least significant bit in the cells of $SA_{\mathcal{F}^i}$to mark every $SA_{\mathcal{F}^i}[s_x]$. We flag these positions during the execution of our modified version of ISS (Paragraph 3.3.3).

Algorithm 3 Computing the preliminary BCR BWT of T^i

Require: $SA_{\mathcal{F}^i}, D^i = (R^i, L^i, N^i), B^i$

1: $\ell, b, f \leftarrow 0$
2: $s_x, e_x, o \leftarrow 1$
3: $c', c \leftarrow \varepsilon$
4: $pBWT^i, M, SA_s \leftarrow \emptyset$
5: **for** $u = 1$ to $|SA_{\mathcal{F}^i}|$ **do**
6: **if** $u > 1$ and $SA_{\mathcal{F}^i}[u]$ is the start of a new block in $SA_{\mathcal{F}^i}$ **then**
7: $j \leftarrow SA_{\mathcal{F}^i}[s_x]$, $e_x \leftarrow u - 1$ ▷ process block $SA_{\mathcal{F}^i}[s_x..e_x]$ for S_x
8: **if** $L^i[j+1] \neq 1$ **or** $B^i[R^i[j]] = 1$ **then** ▷ S_x belongs to $\mathcal{S}^i$
9: **if** $b > 1$ **or** $f = 1$ **then** ▷ S_x produces an unsolved block in $pBWT^i$
10: **if** $b > 1$ **then** ▷ S_x is a left-maximal suffix in $\mathcal{F}^i$
11: $c \leftarrow$ #
12: $j' \leftarrow$ leftmost index $j' \geq j$ with $L[j'+1] = 1$ or $j' = |R^i|$
13: $S_x \leftarrow R^i[j..j']$
14: $M \leftarrow M \cup (S_x, \sigma^i + o)$
15: mark occurrences of S_x in R^i as left-maximal
16: **else** ▷ S_x is a non-proper suffix in $\mathcal{F}^i$
17: $c \leftarrow$ *
18: **end if**
19: $SA_s \leftarrow SA_s \cup \{j\}$ ▷ sample one occurrence of S_x
20: $o \leftarrow o + 1$
21: **end if**
22: **if** c equals the symbol in the rightmost run of $pBWT^i$ **then**
23: increase the length of that run by ℓ
24: **else**
25: $pBWT^i \leftarrow pBWT^i \cup \{(c, \ell)\}$ ▷ append new run
26: **end if**
27: **end if**
28: $\ell, b, f \leftarrow 0, s_x \leftarrow u, c' \leftarrow \varepsilon$ ▷ initialize new block's information
29: **end if**
30: **if** $L[SA_{\mathcal{F}^i}[u]] = 1$ **then** ▷ S_x is a phrase in $\mathcal{F}^i$
31: $c \leftarrow$ @, $f \leftarrow f + 1$
32: **else**
33: $c \leftarrow R^i[SA_{\mathcal{F}^i}[u] - 1]$
34: **end if**
35: $b \leftarrow b + c \neq c'$, $\ell \leftarrow \ell + N^i[\mathsf{rank}(L^i, j)]$
36: $c' \leftarrow c$
37: **end for**
38: process last block of $SA_{\mathcal{F}^i}$ as in lines 6-29
39: store $pBWT^i$ on disk
40: **return** (SA_s, M)

compute the number b of left-context breaks, the number f of non-proper suffixes, and the accumulative frequency ℓ of the phrases of $\mathcal{F}^i$ enclosing the occurrences of S_x in $SA_{\mathcal{F}^i}[s_x..e_x]$. A left-context break between consecutive positions $SA_{\mathcal{F}^i}[u-1]$ and $SA_{\mathcal{F}^i}[u]$, with $u \in [s_x+1, e_x]$, occurs when $R^i[SA_{\mathcal{F}^i}[u-1]-1]$ differs from $R^i[SA_{\mathcal{F}^i}[u]-1]$. For technical convenience, we compare the left context of $SA_{\mathcal{F}^i}[s_x]$ against the empty symbol $\varepsilon \neq \Sigma^i$, so every block in $SA_{\mathcal{F}^i}$ has at least one left-context break. On the other hand, a suffix $SA_{\mathcal{F}^i}[u]$, with $u \in [s_x, e_x]$, is non-proper when $R^i[SA_{\mathcal{F}^i}[u]]$ is the leftmost symbol of a phrase. In this case, we assign a special symbol $\texttt{@} \notin \Sigma^i$ as its left context. Lines 30–36 describe the process of computing the values (b, f, ℓ) for $SA_{\mathcal{F}^i}[s_x..e_x]$ on the fly as we scan $SA_{\mathcal{F}^i}$. Once we reach the start of a new block in $SA_{\mathcal{F}^i}$, we process the information of the previous $SA_{\mathcal{F}^i}[s_x..e_x]$, but only if its associated sequence S_x belongs to $\mathcal{S}^i$. This condition holds when $|S_x| > 1$, or $|S_x| = 1$ and $exp(S_x[1])$ is suffixed by $\texttt{\$}$. If $b > 1$ or $f = 1$, S_x induces an unsolved block in $pBWT^i$. Consequently, we append ℓ copies of a special symbol to $pBWT^i$. The value of this symbol depends on f and b. If $b = 1$ and $f = 1$, S_x is always a non-proper suffix in $\mathcal{F}^i$ (Lemma 3.4), so we append $*^\ell$. On the other hand, if S_x is left-maximal ($b > 1$), we append $\#^\ell$ instead (Lemma 3.3). When the symbol is $\#$, we also store S_x associated with an integer in a dictionary M. If $SA_{\mathcal{F}^i}[s_x..e_x]$ is the oth block producing a special symbol in $pBWT^i$, the integer for S_x is $\sigma^i + o$. After deciding the special symbol we insert in $pBWT^i$, we append $SA_{\mathcal{F}^i}[s_x]$ into another array SA_s representing a sampled version of $SA_{\mathcal{F}^i}$. On the other hand, when $f = 0$ and $b = 1$, S_x always occurs as a proper suffix in $\mathcal{F}^i$, and it is always preceded by $c \in \Sigma^i$ (Lemma 3.5). Thus, we append c^ℓ to $pBWT^i$. Lines 6–29 describe the processing of $SA_{\mathcal{F}^i}[s_x..e_x]$. Once we finish the scan of $SA_{\mathcal{F}^i}$, we store $pBWT^i$ into the disk and return SA_s and M. Figure 5 shows an example of the computation of $pBWT^i$ in compressed space.

Example 3.1. Construction of $pBWT^i$ in Figure 5. The first run in $pBWT^i$ is $\texttt{c}^2$ because the suffix $R^i[SA_{\mathcal{F}^i}[1] = 7] = \texttt{\$}$ of the first block belongs to $R^1[4..7] = \texttt{acc\$}$, that has frequency $N^1[\texttt{rank}_1(L^1, 7) = 2] = 2$. In contrast, the second run is $*^4$ because the suffixes 8, 4 and 12 of the next three $SA_{\mathcal{F}^i}$ blocks map to the full phrases $R^i[8..11] = \texttt{aata}$, $R^i[4..7] = \texttt{acc\$}$, and $R^i[12..15] = \texttt{agta}$, respectively. The run has length 4, and not 3, because $\texttt{acc\$}$ has frequency $N^1[\texttt{rank}_1(L^1, 4) = 2] = 2$. We add $(\texttt{gta}, 9)$ into M because $\texttt{gta}$ is left-maximal in $\mathcal{F}^i$, and its associated block $SA_{\mathcal{F}^i}[9..10]$ is the $4th$ block producing special symbols in $pBWT^i$. Now, assuming $\sigma^1 = 5$, we

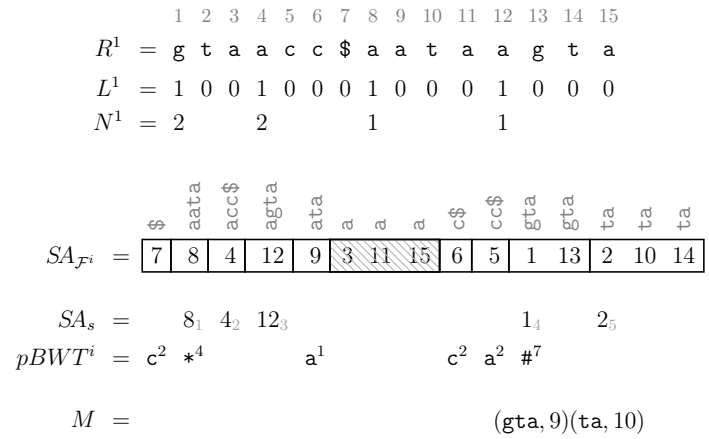

Figure 5: Construction of $pBWT^1$ in compressed space using the succinct dictionary $D^1 = (R^1, L^1, N^1)$ of the LMS parsing of Figure 2. The boxes in $SA_{\mathcal{F}^i}$ are the blocks of equal suffixes in $\mathcal{F}^i$. We skip block 6 (dashed box) because its suffixes overlap other phrases in $\mathcal{F}^i$, and hence, are redundant to build $pBWT^i$.

have that the integer value we store in M is $4 + 5 = 9$. A similar situation occurs with $(\texttt{ta}, 10)$.

Output Data Structures in the Construction of $pBWT^i$. In addition to the preliminary BCR BWT of T^i, Algorithm 3 produces two auxiliary data structures: SA_s and M. We will use these extra elements to assist in the compression of $\mathcal{F}^i$ in the next iteration step. SA_s is a sampled version of $SA_{\mathcal{F}^i}$ storing one occurrence in R^i for each string $S_x \in \mathcal{S}^i$ whose corresponding block $pBWT^i[s_x..e_x]$ is unsolved. Thus, if there are $n' \geq |\mathcal{F}^i|$ blocks in $SA_{\mathcal{F}^i}$ that have an unsolved projected block in $pBWT^i$, then $SA_s[1..n']$ has n' sampled positions. Notice that because SA_s maintains the relative order of the elements in $SA_{\mathcal{F}^i}$, the strings encoded by SA_s are sorted in LMS order. On the other hand, the dictionary M stores as a key each string $S_x \in \mathcal{S}^i$ that is left-maximal in $\mathcal{F}^i$. The associated value of each key is the rank of the unsolved block that S_x produced in $pBWT^i$. For instance, if the area $SA_{\mathcal{F}^i}[s_x..e_x]$ for S_x is the *oth* block producing an unsolved segment in $pBWT^i$, the integer in M for S_x is $o + \sigma^i$. We add σ^i just for technical convenience.

3.3.4. Grammar Compression

Once we finish constructing $pBWT^i$, the next step in the parsing iteration i is to store $\mathcal{F}^i$ in a compact form to use it later during the induction phase

$$\mathcal{F}^i_{exp} = \{\texttt{aa}\underline{\texttt{a}}\texttt{ta},\ \texttt{acc\$},\ \underline{\texttt{a}}\texttt{gta},\ \underline{\texttt{g}}\texttt{ta},\ \texttt{ta}\}$$
$$G^1 = \texttt{a10}\quad \texttt{@\$}\quad \texttt{a9}\quad \texttt{g10}\quad \texttt{@t}$$

Figure 6: Example of our grammar-like encoding for the set $\mathcal{F}^i_{exp}$ constructed from the parsing set $\mathcal{F}^1$ of Figure 4. Each suffix in grey is the longest left-maximal suffix of its phrase. The underlined symbol is the left context of that suffix.

(Line 6 of Algorithm 2). We first explain why we need the parsing set during the induction phase and then describe the format we choose to encode it.

Broadly speaking, the induction process consists of scanning BWT^{i+1}_{bcr} left to right, mapping every symbol $BWT^{i+1}_{bcr}[j] \in \Sigma^{i+1}$ back to the phrase $F[1..n_f] \in \mathcal{F}^i$ from which it originated, and then checking which of the proper suffixes of F produced unsolved blocks in $pBWT^i$ (see Lemmas 3.3 and 3.4). Assume the suffix $F[u..n_f] = S_x \in \mathcal{S}^i$ produced the unsolved block $pBWT^i[s_x..e_x]$, then we insert $F[u-1]$ into $pBWT^i[s_x..e_x]$.

The process described above requires a mechanism to map left-maximal suffixes in $\mathcal{F}^i$ back to the unsolved block they produce in $pBWT^i$. We implement this feature by encoding $\mathcal{F}^i$ with a representation that is similar to grammar compression (Section 2.1).

We start by modifying the set to make it suitable for our encoding. First, we insert each string $S_x \in \mathcal{S}^i \setminus \mathcal{F}^i$ associated with the unsolved block $pBWT^i[s_x..e_x]$ into $\mathcal{F}^i$. These strings are those meeting Lemma 3.3 and that only appear as a proper suffix in $\mathcal{F}^i$, not as a full phrase. Then, we sort the strings in $\mathcal{F}^i$ in $\prec_{LMS}$ order. We refer to this new version of $\mathcal{F}^i$ as the *expanded* parsing set $\mathcal{F}^i_{exp}$.

Definition 3.5. Expanded parsing set: a string set $\mathcal{F}^i_{exp} \subset \mathcal{S}^i$ storing each element $S_x \in \mathcal{S}^i$ whose associated block $pBWT^i[s_x..e_x]$ either meets Lemma 3.3 or Lemma 3.4. Additionally, the strings in $\mathcal{F}^i_{exp}$ are sorted in LMS order.

Notice the convenience of the expanded set: if we need to access the string S_x associated with the *oth* unsolved block $pBWT^i[s_x..e_x]$, we visit the *oth* string in $\mathcal{F}^i_{exp}$. Now, to fill $pBWT^i[s_x..e_x]$, we also need to know the symbols in Σ^i preceding S_x. We support this functionality by compressing $\mathcal{F}^i_{exp}$. Specifically, we replace the suffix occurrences of S_x in $\mathcal{F}^i_{exp}$ with its LMS order o in $\mathcal{F}^i_{exp}$. Thus, for instance, if we access a string F in the compressed version of $\mathcal{F}^i_{exp}$ that is suffixed by $c{\cdot}o$, we know that c is the left context of S_x, and that goes within the *oth* unsolved block $pBWT^i[s_x..e_x]$.

This grammar-like encoding is lossless because there is a phrase within $\mathcal{F}^i_{exp}$ for each symbol $o \notin \Sigma^i$ we insert. Therefore, if we need to access the nested left-maximal suffixes of S_x, we just visit the oth string in $\mathcal{F}^i_{exp}$. We can further compress $\mathcal{F}^i_{exp}$ by removing the suffixes that do not produce unsolved blocks in $pBWT^i$. The result is a lossy grammar-like encoding that gives us access to the left-maximal suffixes of $\mathcal{F}^i_{exp}$ and their left-context symbols. This feature is enough for us to run the induction phase.

In practice, the sampled suffix array SA_s that we produced with Algorithm 3 is an implicit representation of $\mathcal{F}^i_{exp}$, so we do not have to compute it. On the other hand, the dictionary M contains the information we need to compress the suffixes of $\mathcal{F}^i_{exp}$ that generate unsolved blocks in $pBWT^i$.

Grammar-compressing the Expanded Parsing Set. Algorithm 4 shows in detail the steps we perform during this procedure. We start by initializing a new vector $G^i[1..2|SA_s|]$ to store the compressed version of $\mathcal{F}^i_{exp}$. Then, we scan SA_s from left to right, and for every position $j = SA_s[u]$ we visit, we access its associated phrase $F[1..n_f] = R^i[j..j'] \in \mathcal{F}^i_{exp}$, with $j' \geq j$ being the leftmost index with $L^i[j'+1] = 1$. First, if $j = j'$ and $L^i[j'] = 1$, then $F = R^i[j]$ has length one and expands to a full string $exp(F[1]) = T_x\$$, with $T_x \in \mathcal{T}$. This situation is a corner case generated by the LMS parsing, so we set $F = \texttt{@}R^i[j]$ (see Line 6), where $\texttt{@} = |SA_s| + 1$ is a special symbol that denotes an invalid element in the encoding. On the other hand, when $n_f > 1$, we scan the phrase from left to right to find its longest proper suffix $S_x = F[u..n_f] = R^i[p..j']$, with $j > p \leq j'$, which exists in M as a key. If such a key exists, we obtain its left context $c = F[u-1] = R^i[p-1]$ and replace F with $c{\cdot}o'$, where o' is the integer associated with S_x in M (Lines 10–15). If no proper suffix of F exists in M as a key, we replace F with a sequence of length two. The left symbol will be @, while the right symbol depends on F' sequence. If $F[n_f] = R^i[j']$ expands to a string in Σ^* suffixed by \$, we set the right symbol to $F[n_f]$, or set the right symbol to $F[n_f - 1] = R^i[j'-1]$ otherwise (Lines16–22). Once we update F, we append it to G^i. After we finish the scan of SA_s, we destroy R^i, L^i, SA_s, and M, and finally store G^i on disk. Figure 4 depicts an example of our grammar-like encoding.

Decompressing Strings of the Expanded Parsing Set. Consider again the phrase F, which we replaced with the string $c{\cdot}o'$ in G^i. The symbol $o' > \sigma^i$ is the integer we obtained when we performed a lookup operation of $S_x = F[u..n_f]$ in M during the execution of Algorithm 4, while $c = F[u-1]$. Further, the

value $o' = o + \sigma^i$ is the LMS order o of S_x in $\mathcal{F}^i_{exp}$ plus σ^i (see Algorithm 3). The existence of S_x in M's keys implies that S_x is a left-maximal suffix in $\mathcal{F}^i$, which in turn implies that S_x is a full phrase in $\mathcal{F}^i_{exp}$. Additionally, the left-maximal condition of S_x implies that at least two suffix occurrences of S_x in $\mathcal{F}^i$ were preceded by different symbols. This is why S_x produces the oth unsolved block $pBWT^i[s_x..e_x]$. Now, recall that the phrases of $\mathcal{F}^i_{exp}$ are stored in $\prec_{LMS}$ order. Therefore, if we want to access the area where S_x lies, we have to set $o = o' - \sigma^i$ and go to $G^i[2o - 1..2o]$. This substring does not encode the full sequence of S_x, but its longest left-maximal suffix $G^i[2o]$ (which is also a left-maximal suffix of F) along with the left-context symbol for that suffix ($G^i[2o-1] \in \Sigma^i$). Recursively, the longest left-maximal symbol of S_x is not a sequence either but a pointer to another position of G^i. We access this nested left-maximal suffix by setting $o = G^i[2o] - \sigma^i$ and updating the area $G^i[2o-1..2o]$. We continue applying this idea until we reach a range $G^i[2o - 1..2o]$ where $c = G^i[2o] \leq \sigma^i$, which implies that we reached the last suffix of F. In most of the cases, c is not $F[n_f]$, but $F[n_f - 1]$. The reason is that the LMS substrings overlap by one symbol in T^i, so $F[n_f]$ is redundant as it also appears as a prefix in another phrase. The only exception to this rule is when F expands to a suffix of a string in $\mathcal{T}$. In that case, $G^i[2o]$ is indeed $F[n_f]$ because F does not overlap the prefixes of other LMS parsing phrases.

Example 3.2. Spelling the left-context symbols of the left-maximal suffix `agta` (Figure 6). The string `agta` has LMS order 3 in $\mathcal{F}^i_{exp}$ so we visit the $3th$ phrase in $G^1[2{\times}3 - 1..2{\times}3] = G^1[5..6] =$ `a`9, and access the left symbol `a`. Then, we compute the next string using the right symbol 9 as $4 = 9{-}\sigma^i = 9{-}5$, visit the $4th$ string $G^i[2{\times}4{-}1..2{\times}4] = G^1[7..8] =$ `g`10 (`gta`) and output `g`. We repeat the same process, computing the string $5 = 10 - 5$ and visiting $G^1[9..10] =$ `@t` (`t`a). However, this time we have `t` $\leq \sigma^1$, which means we reached the rightmost suffix of `agta` that does match the prefix of other phrases in the LMS parsing of T^1 (see the phrases ending with `ta` in the string T^i of Figure 2). This situation does not occur, for instance, with `acc$`, whose rightmost suffix in G^1 is indeed `$` because `acc$` can not overlap.

3.4. Creating the String for the Next Iteration of Parsing

The final step of iteration i during the parsing phase is to create the text T^{i+1}. For this purpose, we produce a new dictionary D^i containing the strings in $\mathcal{F}^i$ (i.e., the parsing phrases) as keys. The value associated with

Algorithm 4 Grammar compressing $\mathcal{F}^i_{exp}$

Require: R^i, L^i, B^i, SA_s, M

$G^i \leftarrow \emptyset$ ▷ grammar-compressed $\mathcal{F}^i_{exp}$
for $u = 1$ to $|SA_s|$ **do** ▷ visit each $F = R^i[j..j'] \in \mathcal{F}^i_{exp}$
  $j \leftarrow SA_s[u]$
  $j' \leftarrow$ leftmost index $j' \geq j$ with $L[j'+1] = 1$ or $j' = |R^i|$
  $o' \leftarrow \varepsilon, c \leftarrow$ @
  if $L^i[j'] = 1$ and $B^i[R^i[j']] = 1$ **then** ▷ $exp(R^i[1] = F[1]) = T_x$\$
    $o' \leftarrow R^i[j]$
  else
    $p \leftarrow j + 1$
    while $p \leq j'$ **and** $R^i[p..j']$ is not left-maximal **do**
      $p \leftarrow p + 1$
    end while
    if $S_x = R^i[p..j']$ is left-maximal **then**
      $c \leftarrow R^i[p-1]$
      $o' \leftarrow$ value associated to S_x in M
    else ▷ F does not have left-maximal suffixes
      if $B^i[R^i[j']] = 1$ **then** ▷ F expands to a suffix in $\mathcal{T}$
        $o' \leftarrow R^i[j']$
      else ▷ F is a regular parsing phrase with overlap
        $o' \leftarrow R^i[j'-1]$
      end if
    end if
  end if
  $F \leftarrow c \cdot o'$
  $G^i \leftarrow G^i \cup F$
end for
destroy R^i, L^i, B^i, SA_s and M
store G^i on disk

each key is its LMS order in $\mathcal{F}^i_{exp}$. We construct T^{i+1} by running LMS parsing over T^i again to replace the phrases with their associated values in D^i. If T^{i+1} has length k (the number of strings in $\mathcal{T}$), we stop the parsing phase as all the strings in $\mathcal{T}$ are now compressed to one symbol.

The only caveat with this construction is that the symbols in the alphabet Σ^{i+1} of T^{i+1} are not consecutive if $|\mathcal{F}^i_{exp}| > |\mathcal{F}^i|$, but this feature does not

change the correctness of our method. Specifically, if $T^{i+1}[j] < T^{i+1}[j']$, it still hold that $exp(T^{i+1}[j]) \prec_{LMS} exp(T^{i+1}[j'])$. We will use the fact that the symbols in T^{i+1} are the LMS order in $\mathcal{F}^i_{exp}$ and not in $\mathcal{F}^i$ during the induction phase.

3.5. The Induction Phase

The induction phase starts with the computation of BWT^h_{bcr}, the BCR BWT for the text T^h of the last iteration h of the parsing phase (Line 11 of Algorithm 2). This step is trivial as each symbol in T^h encodes a full string of $\mathcal{T}$ (see the ending condition of the parsing phase). Hence, the left context of every symbol is the symbol itself. The BCR BWT maintains the relative order of the strings in $\mathcal{T}$ (see Section 2.3), so BWT^h_{bcr} is T^h itself.

Before explaining our induction procedure, we describe some important properties of BWT^{i+1}_{bcr}. As a quick reminder, $\mathcal{F}^i_{exp}$ is the expanded parsing set encoding the strings in $\mathcal{F}^i$ (see Lemma 3.2) plus the sequences that are left-maximal suffixes in $\mathcal{F}^i$. The strings in $\mathcal{F}^i_{exp}$ are precisely those inducing unsolved blocks in $pBWT^i$.

Lemma 3.6. Let $BWT^{i+1}_{bcr}[j]$ and $BWT^{i+1}_{bcr}[j']$ be two symbols in Σ^{i+1}, with $j < j'$, whose corresponding phrases in $\mathcal{F}^i$ are $F[1..n_f]$ and $F'[1..n_{f'}]$, respectively. Additionally, let the proper suffixes $F[u..n_f] = F'[u'..n_{f'}] = S_x \in \mathcal{F}^i_{exp}$ be left-maximal in $\mathcal{F}^i$. Now consider the substrings $map^i(T^{i+1}[GSA^{i+1}[j] - 1]) = T^i[p..p + n_f - 1]$ and $map^i(T^{i+1}[GSA^{i+1}[j'] - 1]) = T^i[p'..p' + n_{f'} - 1]$ with the occurrences of F and F' that formed the symbols $BWT^{i+1}_{bcr}[j]$ and $BWT^{i+1}_{bcr}[j']$ in T^{i+1}, respectively. The suffix $T^i[p + u - 1..n^i]$ prefixed by $S_x = F[u..n_f]$ precedes in GSA^i the suffix $T^i[p' + u' - 1..n^i]$ prefixed by $S_x = F[u'..n_{f'}]$.

Proof. As the prefixes $T^i[p + u - 1..n_f] = F[u..n_f] = S_x$ and $T^i[p' + u' - 1..n_{f'} - 1] = F'[u'..n_{f'}] = S_x$ are equal, the relative order of $T^i[p + u - 1..n^i]$ and $T^i[p' + u' - 1..n^i]$ is decided by the right contexts in T^{i+1} of the occurrences $BWT^{i+1}_{bcr}[j]$ and $BWT^{i+1}_{bcr}[j']$. By induction, we know that BWT^{i+1}_{bcr} is complete, and as $BWT^{i+1}_{bcr}[j]$ precedes $BWT^{i+1}_{bcr}[j']$ in the BWT, the right context of $T^i[p+u-1..n_f]$ is lexicographically smaller than the right context of $T^i[p + u' - 1..n_{f'}]$. □

We generalize Lemma 3.6 to compute the block $pBWT^i[s_x..e_x] = \#^\ell$ that meets Lemma 3.3 and whose string $S_x \in \mathcal{S}^i \setminus \mathcal{F}^i$ *only* occurs as a proper suffix in $\mathcal{F}^i$.

Lemma 3.7. Let $J = \{j_1, j_2, \ldots, j_\ell\}$ be a set of strictly increasing positions of BWT^{i+1}_{bcr}. Every $BWT^{i+1}_{bcr}[j_b]$, with $j_b \in J$, is a symbol $o \in \Sigma^{i+1}$ assigned to a phrase $F[1..n_f] \in \mathcal{F}^i$ where $S_x = F[u..n_f] \in \mathcal{F}^i_{exp}$ occurs as a proper suffix. The symbols of BWT^{i+1}_{bcr} referenced by J are not necessarily equal, and hence, their associated phrases in $\mathcal{F}^i$ are not necessarily the same. However, these phrases of $\mathcal{F}^i$ are all suffixed by S_x. Assume we scan J from left to right, and for every j_b, we extract the symbol $F[u-1] \in \Sigma^i$ that precedes S_x and append it to a vector L_{S_x}. The resulting sequence for $L_{S_x} \in \Sigma^{i*}$ matches the unsolved block $pBWT^i_{bcr}[s_x..e_x] = \texttt{\#}^\ell$ generated by S_x.

Proof. Lemma 3.6 tells us that the suffix of T^i prefixed by the occurrence of S_x encoded by $BWT^{i+1}[j_b]$ precedes the suffix of T^i prefixed by the occurrence encoded by $BWT^{i+1}[j_{b+1}]$. This property holds for every j_b, with $b \in [1, \ell - 1]$. Hence, the suffixes of T^i prefixed by S_x are already sorted in J. On the other hand, Lemma 3.2 tells us that all the occurrences of S_x as a suffix of a parsing phrase appear consecutively in $GSA^i[s_x..e_x]$. Thus, by taking the left-context symbols of S_x's occurrences encoded by J, we obtain $pBWT^i_{bcr}[s_x..e_x]$. □

Example 3.3. Filling the unsolved block $pBWT^i[15..18] = \texttt{\#}^4$ of Figure 4 with Lemma 3.7. Consider the BCR BWT $BWT^2_{bcr} = 4\ 4\ 3\ 1\ 2\ 2$ for the text $T^2 = 4\ 2\ 4\ 1\ 3\ 2$ of Figure 2. The expanded parsing set $\mathcal{F}^i_{exp}$ from which the symbols of BWT^2_{bcr} were generated is shown in Figure 6. Additionally, consider the string $\texttt{ta} \in \mathcal{S}^1$ generating the block $GSA^1[15..18]$ in the partition induced by $\mathcal{S}^1$ (see Figure 4). As $SA^1[15..18]$ meets Lemma 3.3, the projected block $pBWT^i[15..18] = \texttt{\#}^4$ is unsolved. Lemma 3.7 tells us that we can solve $pBWT^1[15..18]$ provided we know BWT^{i+1}_{bcr} and the phrases of $\mathcal{F}^1$ where `ta` occurs as a suffix. The prefix $BWT^2_{bcr}[1..4] = 4\ 4\ 3\ 1$ contains all the symbols in Σ^{i+1} whose associated phrases are suffixed by `ta`. In particular, if we replace 4 4 3 1 with their phrases in $\mathcal{F}^i_{exp}$, we obtain `gta`, `gta`, `agta`, `aata`. We apply Lemma 3.7 by taking the left context of `ta` in those strings without changing the relative order and thus obtain $L_{\texttt{ta}} = pBWT^1[15..18] =$ `g g g a`, which is precisely the substring $BWT_{bcr}[15..18]$ of Figure 4.

When S_x does not occur as a nested proper suffix in $\mathcal{F}^i$ (Lemma 3.4), there is no left-context symbol we can extract from the parsing set, so Lemma 3.7 does not work. Nevertheless, we can use BWT^i_{bcr} in other ways to complete the unsolved block $pBWT^i[s_x..e_x] = \texttt{*}^\ell$ that S_x generated.

Lemma 3.8. Let $pBWT^i[s_x..e_x] = *^\ell$ be an unsolved block induced by a string $S_x \in \mathcal{S}^i$ that meets Lemma 3.4. Additionally, let o' and $o \in \Sigma^{i+1}$ be the LMS orders of S_x in $\mathcal{F}^i$ and $\mathcal{F}^i_{exp}$, respectively. Let $GSA^i[u..u']$ be the bucket $o' \leq o$ storing the suffixes of T^i prefixed by o. The element in $pBWT[s_x + j - 1]$ is the rightmost symbol in $exp^i(BWT^{i+1}_{bcr}[u + j - 1])$, with $u + j - 1 \leq u'$.

Proof. S_x is the LMS parsing phrase to which we assign the symbol $o \in \Sigma^{i+1}$ as a replacement for the text T^{i+1}, o being the LMS order of S_x in $\mathcal{F}^i_{exp}$. Recall that this construction produces the alphabet Σ^{i+1} of T^{i+1} to be non-contiguous. Therefore, the bucket $GSA^i[u..u']$ number $o' \leq o$ is the one storing the suffixes of T^{i+1} prefixed by o. We know that S_x does not occur as a nested proper suffix within $\mathcal{F}$ (Lemma 3.4), meaning that *all* its left-context symbols (those we insert in $pBWT^i[s_x..e_x]$) are captured[3] by the symbols that precede o in T^{i+1}. By induction, $BWT^{i+1}_{bcr}[u..u']$ already has these preceding symbols in BWT order. Thus, the remaining step is to decompress those symbols, take the rightmost element of their phrases, and place them in $pBWT^i[s_x..e_x]$ without changing their relative order. □

Example 3.4. Filling the unsolved block $pBWT^i[4..5] = *^2$ produced by `acc$` $\in \mathcal{F}^1$ of Figure 4 using Lemma 3.8. The phrase `acc$` has LMS order $o = 2 \in \Sigma^2$ in $\mathcal{F}^i_{exp}$. Further, $GSA^i[2..3]$ is the bucket number $o' = 2$ and has the suffixes of T^{i+1} prefixed by $o = 2$ (see Figure 7). The corresponding range in the BWT has the sequence $BWT^2_{bcr}[2..3] = 4\ 3$. Decompressing their phrases gives us $exp(4) =$ `gt` and $exp(3) =$ `agt` (recall that exp removes the last element in the overlapping LMS phrases of T^i). If we take their rightmost symbols, we produce $pBWT^1[4..5] =$ `t t`, which matches $BWT_{bcr}[4..5]$ in Figure 4.

Finally, we cover the case when S_x occurs as a phrase $F = S_x \in \mathcal{F}^i$ but also as a nested proper suffix $S_x = F'[p..n_{f'}]$ in another parsing phrase $F'[1..n_{n'}] \in \mathcal{F}^i$. We solve its block $pBWT^i[s_x..e_x] = \#^\ell$ using a hybrid strategy that combines Lemmas 3.7 and Lemma 3.8.

Lemma 3.9. Let J be the set of Lemma 3.7 with the occurrences in BWT^{i+1} of the phrases suffixed by S_x. This time, these suffixes could be proper or

[3]That is, the symbols in $pBWT^i[s_x..e_x]$ occur within the phrases of $\mathcal{F}^i$ that map to symbols preceding o in T^{i+1}.

non-proper. Assume we scan J from left to right to fill L_{S_x}, but with a small change: if $exp^i(BWT^i[j_b]) = F$, with $j_b \in J$, we append the special symbol $*$ in L_{S_x}. After the scan, we replace the occurrences of $*$ in L_{S_x} with Lemma 3.8. The replacement of the jth special symbol in L_{S_x} is the rightmost element of $exp^i(BWT^{i+1}_{bcr}[u+j-1])$, where $GSA^i[u..u']$ is the bucket with the suffixes of T^{i+1} prefixed by the symbol $o \in \Sigma^{i+1}$ assigned to $F = S_x$.

Our lossy grammar-like representation of $\mathcal{F}^i_{exp}$ (i.e., the vector G^i of Section 3.3.4) has all the information we need to fill the unsolved blocks as described in Lemmas 3.7, 3.8, and 3.9. The only extra information we need to complete $pBWT^i$, and that it is not in G^i, is the order in which we have to rearrange the left-context symbols of S_x within $pBWT^i_{bcr}[s_x..e_x]$. Fortunately, we can induce this information from BWT^{i+1}_{bcr}.

It is also worth mentioning that the special symbols $*$ and # we introduced in $pBWT^i$ indicate what lemma we should use to fill the unsolved blocks. We will use this fact in the next section.

3.6. The Induction Algorithm

We now describe the steps we perform during iteration $i < h$ in the induction phase (loop in lines 13-18 of Algorithm 2). Notice that the value of i decrements with each round as we simulate the return from a recursion that is equivalent to that of SA-IS. In this case, we assume we receive as input for the iteration the string (i) BWT^{i+1}_{bcr}, (ii) the vector G^i with the lossy grammar-like encoding of $\mathcal{F}^i_{exp}$, and (iii) $pBWT^i$. We also assume a bit vector $V^i[1..\sigma^{i+1}]$, with $\sigma^{i+1} = |\mathcal{F}^i_{exp}|$, indicates with $V^i[o] = 1$ if the oth string $S_x \in \mathcal{F}^i_{exp}$ in LMS order is left-maximal in $\mathcal{F}^i$. The output of the iteration is BWT^i_{bcr}, the BCR BWT of string T^i. The details of the procedure are shown in Algorithm 5.

Data Structures and Encoding. A central piece of our induction algorithm is an in-memory vector P^i storing the left-context symbols of the strings $S_x \in \mathcal{F}^i_{exp}$ occurring as left-maximal suffixes in $\mathcal{F}^i$ (Lemmas 3.7 and 3.9). We divide P^i into $\mathsf{rank}_1(V^i, \sigma^{i+1})$ buckets. Thus, for a left-maximal suffix $S_x \in \mathcal{F}^i_{exp}$ whose LMS order in $\mathcal{F}^i_{exp}$ is $o \in \Sigma^{i+1}$, we store its left-context symbols in the bucket $b = \mathsf{rank}_i(V^i, o)$. Put differently, the bucket b of P^i is an encoding for the sequence $L_{S_x} = pBWT^i[s_x..e_x]$ of the previous section. We keep P^i in run-length-compressed format to reduce working memory and CPU time. We estimate the (run-length-compressed) area of every bucket

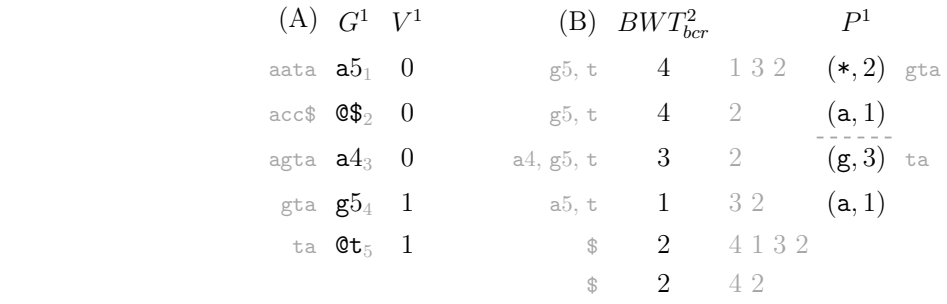

Figure 7: Construction of P^1 during the induction phase. (A) The grammar-like encoding G^i of $\mathcal{F}^i_{exp}$ in Figure 6. We subtracted $\sigma^1 = 5$ to the right symbols to simplify the example. The vector V^1 indicates which phrases of $\mathcal{F}^i_{exp}$ are left-maximal suffixes in $\mathcal{F}^1$. (B) The vector BWT^2_{bcr} for the string T^2 of Figure 2 and the vector P^1. The dashed line in P^1 marks the boundary between its buckets. The sequence to the left of each $BWT^2_{bcr}[j] = o$ stores the elements we decompress from G^1 starting from symbol o, while the sequence to the right of $BWT^2_{bcr}[j]$ is its following suffix in T^2 sorted as in GSA^2.

within P^i before the induction starts. To simplify our explanations, we will assume we already know this information and will describe how to compute it later (see Paragraph 3.6). We use the notation $P^i[b]$ to denote the complete run-length-compressed area of P^i storing the symbols of bucket b. We also define a process called *RLC append*: let L be a run-length-compressed string and let (c, ℓ) be an equal-symbol run we need to append into L. If c matches the symbol of the rightmost run in L, we increase the length of that run by ℓ or append (c, ℓ) as a new run otherwise.

The Induction Process. This step consists in computing the unsolved blocks $pBWT^i[s_x..e_x] = \#^\ell$ that meet Lemmas 3.7 and 3.9. However, instead of inserting the result right away into $pBWT^i$, we will insert it into P^i. We visit the equal-symbol runs of BWT^{i+1}_{bcr} from left to right. When we reach the jth run (o, ℓ), with $o \in \Sigma^{i+1}$, we check if its corresponding[4] parsing phrase $F[1..n_f] \in \mathcal{F}^i$ exists as a suffix in other phrases of $\mathcal{F}^i$. If that is the case, we RLC append ℓ copies of the special symbol $* \notin \Sigma^i$ to the bucket $b = \mathsf{rank}(V^i, o)$ of P^i (see Lines 4–6). Inserting $*$ into P^i is equivalent to handling a block of $pBWT^i$ that meets Lemma 3.9. The next step is to decompress the left-maximal suffixes of F from G^i (see Section 3.3.4). This

[4]The string from which we obtain the symbol o during iteration i of the parsing phase

process begins by accessing $G^i[2o-1..2o]$. If $o' = G^i[2o] > \sigma^{i+1}$, then o' encodes a string $F[u..n_f] = S_x \in \mathcal{F}_{exp}$, with $u > 1$, whose sequence is a left-maximal suffix in $\mathcal{F}^i$ (see Lemma 3.3). We encoded o as $o' = o + \sigma^i$ in G^i to differentiate it from the symbols in Σ^i. On the other hand, the left-context symbol of S_x is $G^i[2o-1] \in \Sigma^i$. With this information, we apply Lemma 3.7 by RLC appending ℓ copies of $G^i[2o-1]$ to the bucket $\mathsf{rank}_1(V^i, G^i[2o] - \sigma^i)$ of P^i. Then, we move to the next left-maximal suffix of F by setting $o = G^i[2o] - \sigma^i$ and updating the range $G^i[2o-1..2o]$. The decompression of F stops when $G^i[2o] \leq \sigma^i$, which means we reached the last symbol of F. For the moment, we do not know for which phrase of $\mathcal{F}^i$ $G^i[2o]$ is its left context. Hence, we update the value of the jth run in BWT^{i+1} to $G^i[2o] \in \Sigma^i$ and leave this run on hold to process it later when we solve the blocks of Lemma 3.8.

Example 3.5. Construction of the vector P^i in Figure 7. Consider the run $BWT^2_{bcr}[1..2] = 4^2$. Its phrase $F = \mathtt{gta}$ (the $4th$ string of $\mathcal{F}^i_{exp}$ in LMS order) is left-maximal in $\mathcal{F}^i$ as $V^1[4] = 1$. Hence, we RLC append $(*, 2)$ to the bucket $\mathsf{rank}(V^1, 4) = 1$ of P^1. Then, the decompression of 4 from G^1 produces $\mathsf{g}5$, which means we RLC append $(\mathsf{g}, 2)$ into the bucket $\mathsf{rank}_1(V^1, 5) = 2$ of P^1. Finally, the last decompressed element $\mathtt{t}$ indicates we reached the rightmost non-overlapping suffix of F. Therefore, we replace $BWT^2[1..2] = 4^2$ by $BWT^2[1..2] = \mathtt{t}^2$. Notice we decompressed the symbol $4 \in \Sigma^2$ in $BWT^2_{bcr}[1..2]$ only once but copied the information twice (i.e., the length of the run) to each bucket in P^1.

Merging the Induced Symbols. The last step in iteration i is to compute the blocks $pBWT^i[s_x..e_x] = *^\ell$ of Lemma 3.8 and solve the unfinished blocks $pBWT^i[s_x..e_x] = \#^\ell$ that meet Lemma 3.9 (buckets in P^i containing $*$ symbols). We carry out this process by merging $pBWT^i$, BWT^{i+1}_{bcr}, and P^i. This procedure is, in practice, a merge of three sorted vectors as the symbols are already sorted by their right contexts in T^i. We use the special symbols $*, \#$ in $pBWT^i$ and P^i to change the active vector in the merge. The change is equivalent to switching the lemma we use to build BWT^i_{bcr}. For instance, if we see $\#$ in $pBWT^i$, we go to P^i as this vector contains the symbols sorted with Lemma 3.7. Further, if we see $*$ in P^i, we reached an unfinished block of Lemma 3.9, so we need to visit BWT^{i+1}_{bcr}. Finally, if we see $*$ in $pBWT^i$, we go to BWT^{i+1}_{bcr} as this vector has the symbols sorted with Lemma 3.8. The merge algorithm is as follows: we scan $pBWT^i$ and RLC append its entries to BWT^i_{bcr} as long as the symbols we see in $pBWT^i$ are not $*$ or $\#$ (Line 34).

$$
\begin{aligned}
P^1 &= (*,2)\ (\mathtt{a},1)\ (\mathtt{g},3)\ (\mathtt{a},1)\\
BWT^2_{bcr} &= (\mathtt{t},4)\ (\$,2)\\
pBWT^1 &= (\mathtt{c},2)\ (*,4)\ (\mathtt{a},1)\ (\mathtt{c},2)\ (\mathtt{a},2)\ (\#,7)\\
\\
BWT_{bcr} &= (\mathtt{c},2)\ (\mathtt{t},4)\ (\mathtt{a},1)\ (\mathtt{c},2)\ (\mathtt{a},2)\ (\$,2)\ (\mathtt{a},1)\ (\mathtt{g},3)\ (\mathtt{a},1)
\end{aligned}
$$

Figure 8: Merge of $pBWT^1$, BWT^{i+1} and $pBWT^i$ to produce BWT_{bcr} for the string $T^1 = \mathtt{gtacc\$gtaatagtacc\$}$ of Figure 2. We constructed $pBWT^1$ in Figure 5, and vectors BWT^{i+1}_{bcr} and P^1 in Figure 7.

If we reach a run $*^\ell$, we RLC append the next ℓ symbols of BWT^{i+1}_{bcr} into BWT^i_{bcr}. On the other hand, if we reach $\#^\ell$, we RLC append the next ℓ symbols from P^i instead. Additionally, as we consume the ℓ elements from P^i, we might reach a run $(*, \ell')$. When this happens, we change the active list again and RLC append the next ℓ' symbols of BWT^{i+1}_{bcr} into BWT^i_{bcr}. If $\ell' > \ell$, we recover ℓ symbols of BWT^{i+1} instead and decrease the active run of BWT^{i+1}_{bcr} by ℓ. Once we consume the ℓ' symbols from BWT^{i+1}_{bcr}, we return to P^i to consume what it remains from the ℓ symbols. Equivalently, once we consume the ℓ symbols from P^i (and possibly BWT^{i+1}_{bcr}), we return to $pBWT^i$. Lines 18–32 show how we process the ℓ symbols of P^i.

Example 3.6. Producing BWT_{bcr} in Figure 8. The merge starts by appending $pBWT^1[1..2] = (\mathtt{c}, 2)$ into BWT_{bcr}. The next run $pBWT^i[3..6] = (*, 4)$ is flagged with the special symbol $*$. Therefore, we change the active vector in the merge to BWT^2_{bcr}, and RLC append $BWT^2_{bcr}[1..4] = (\mathtt{t}, 4)$ into BWT_{bcr}. Now the active merge position of BWT^2_{bcr} becomes $BWT^2_{bcr}[5]$. We switch back to $pBWT^1$ to RLC append $pBWT^1[7..11] = (a, 1)(c, 2)(a, 2)$ into BWT_{bcr} as they have symbols in Σ. However, the next run $pBWT^1[12..18] = (\#, 7)$ is flagged with the special symbol #, indicating we need to switch the active vector in the merge again and extract the next seven symbols of BWT_{bcr} from $P^1[1..7]$. Still, the first run $P^1[1..2] = (*, 2)$ is, in turn, flagged with $*$, which means we have to go to $BWT^2_{bcr}[5..7] = (\$, 2)$ (5 being the active merge position of BWT^2_{bcr}) and RLC append $(\$, 2)$ into BWT^1. Finally, we return to $P^1[3..7]$ to obtain the remaining $7-2 = 5$ symbols. Notice that the resulting BWT_{bcr} matches the BCR BWT of T^1 we presented in Figure 4.

Precomputing the Number of Buckets. The last aspect we address for the construction of BWT^i_{bcr} is how to compute the area of every bucket within P^i. We initialize a vector $P'[1, \mathsf{rank}(V^i, \sigma^{i+1})]$ of pairs. Every pair $P'[b] = (x, y)$

is a temporal variable to count the number of runs in the bucket b of P^i. We obtain the values for P' with one decompression of BWT^{i+1}_{bcr} from left to right. The procedure is almost equal to what we show in Lines 2–12 of Algorithm 5, although we do not modify BWT^{i+1}_{bcr}. Recall that this procedure yields a sequence $(c_1, b_1), \ldots, (c_x, b_x)$, where b_j is a bucket we visit in P^i and c_j is the left-context symbol for the phrase associated with the bucket b_j. We compare the symbol c_j with the left element in $P'[b_j]$: if they are equal, we do nothing. If, on the other hand, they differ, we increase the right element in $P'[b_j]$ by one and set the left element to c_j. Once we scan BWT^{i+1}, the right element of every pair $P'[b]$ will contain the maximum number of runs we could see in $P^i[b]$. These values can decrease once we recover the symbols for the unsolved areas (flagged with $*$) of P^i.

3.7. The Complexity of Our Method

We now present the theoretical bounds of grlBWT.

Theorem 3.10. Let $\mathcal{T} = \{T_1, T_2, \ldots, T_k\}$ be a collection with $k = |\mathcal{T}|$ strings over the alphabet Σ and let $T = T_1\$\cdots T_k\$$ be a string of $n = |T|$ symbols over the alphabet $\Sigma \cup \$$ storing the concatenation of $\mathcal{T}$. Additionally, let m be the length of the longest string in $\mathcal{T}$. The algorithm grlBWT constructs the BCR BWT of $\mathcal{T}$ in $O(n+k\log m)$ expected time and requires $O((n+k\log m)\log n)$ bits of working space.

Proof. Our algorithm grlBWT is an adaptation of the algorithm SA-IS of Nong et al. [34]. The authors showed that this method takes linear time and uses $O(n \log n)$ bits of space. The relevant observation is that the length n^{i+1} of the string T^{i+1} is at most $\frac{n^i}{2}$, n^i being the length of the previous T^i. Thus, the cumulative lengths of strings $T^1, T^2, \ldots, T^h$, with $h = O(\log n)$, that SA-IS produces is no more than $2n$. On the other hand, the amount of work in every level i is proportional to n^i, so the overall cost of the algorithm is linear on the input text. We adapt this argument to our method to probe our theoretical bounds.

Running the LMS parsing reduces each substring $T^i[j..j']$ of length $j' - j + 1 > 1$ representing a string $exp(T^i[j..j']) = T_x\$$ of $\mathcal{T}$ to another substring $T^{i+1}[p..p']$ that is at most half the length of $T^i[j..j']$. However, it might happen that the resulting substring $T^{i+1}[p..p']$ has length $p' - p + 1 = 1$. We say that $T^{i+1}[p]$ is uncompressible because it already covers a full string of $\mathcal{T}$, but we can not reduce its size with a new round of LMS parsing. If there

Algorithm 5 Induction of BWT^i_{bcr}

Require: $pBWT^i$, G^i, BWT^{i+1}_{bcr}, and V^i

$P^i \leftarrow$ estimate size of P^i's buckets
for $j = 1$ to number of runs in BWT^{i+1}_{bcr} **do** ▷ populate P^i's buckets
  $(o, \ell) \leftarrow jth$ run in BWT^{i+1}_{bcr} ▷ symbol $o \in \Sigma^{i+1}$ assigned to $F \in \mathcal{F}^i$
  if $V^i[o] = 1$ **then** ▷ F occurs as a left-maximal suffix in $\mathcal{F}^i$
    RLC append ℓ copies of $*$ to the bucket $\mathsf{rank}_1(V^i, o)$ of P^i
  end if
  while $Gi[2o] > \sigma^i$ **do** ▷ visit left-maximal suffixes of F
    RLC append ℓ copies of $G^i[2o-1]$ to $P^i[\mathsf{rank}_1(V^i, G^i[2o] - \sigma^i)]$
    $o \leftarrow G^i[2o] - \sigma^i$
  end while
  set $G^i[2o]$ as the symbol for the jth run of BWT^{i+1}_{bcr}
end for
for $j = 1$ to number of runs in $pBWT^i$ **do** ▷ assemble BWT^i_{bcr}
  $(o, \ell) \leftarrow jth$ run in $pBWT^i$
  if $o = *$ **then**
    RLC append next ℓ symbols of BWT^{i+1}_{bcr} into BWT^i_{bcr}
  else if $o =$ # **then**
    while $\ell > 0$ **do**
      $(o', \ell') \leftarrow$ active run in P^i
      if $\ell' > \ell$ **then**
        decrease length of active run in P^i by $\ell' - \ell$
        $\ell' \leftarrow \ell$
      else
        move to the next run in P^i
      end if
      if $o' = *$ **then**
        RLC append the next ℓ' symbols of BWT^{i+1}_{bcr} into BWT^i_{bcr}
      else
        RLC append ℓ' copies of o' into BWT^i_{bcr}
      end if
      $\ell \leftarrow \ell - \ell'$
    end while
  else
    RLC append ℓ copies of o to BWT^i_{bcr}
  end if
end for
return BWT^i_{bcr}

are substrings of T^{i+1} encoding elements of $\mathcal{T}$ that are still compressible, then grlBWT will incur in another recursion $i+2$ and carry $T^{i+1}[p]$ to T^{i+2} as another symbol. This feature implies that the length n^{i+2} of T^{i+2} is not guaranteed to be at most $\frac{n^{i+1}}{2}$ like in SA-IS. Instead, n^{i+2} is upper bounded by $O(\frac{n^{i+1}}{2} + k)$ as there are less than k uncompressible symbols $T^{i+1}[p]$ we can carry from T^{i+1} to T^{i+2}. Our method finishes the recursions when all the strings of $\mathcal{T}$ encoded by T^i are uncompressible, meaning that we can not produce more than $\log m$ recursions. Thus, the cumulative lengths of our strings $T^1, T^2, \ldots, T^h$, with $h = O(\log m)$, is no more than $\sum_{i=0}^{\log m} \frac{n}{2^i} + k \leq 2n + k \log m$.

We now analyse the amount of work we perform in every recursion level i. Creating the dictionary D^i from T^i using a standard hash table takes $O(n^i)$ expected time and requires $O(n^i \log n^i)$ bits of space. The construction of $SA_{\mathcal{F}^i}$ runs in $O(n^i)$ time and space as we use ISS to build it, and the number of symbols in the keys of D^i is never greater than n^i. The extra steps of the parsing iteration only require a constant number of linear scans over $SA_{\mathcal{F}^i}$. During the induction phase, we only perform linear scans over BWT^{i+1}_{bcr} and $pBWT^i$. We still have the cost of accessing the left-maximal suffixes of $\mathcal{F}^i$ when we scan BWT^{i+1}_{bcr}. However, our simple grammar-like representation G^i (Section 3.3.4) supports random access in $O(1)$ time to the symbols, and the number of left-maximal suffixes we visit during the scan of BWT^{i+1}_{bcr} is no more than n^i. Thus, the cost of every recursion level i is $O(n^i)$ time and $O(n^i \log n^i)$ bits of space. Summing up the $h = O(\log p)$ recursions levels, the total cost of grlBWT is $O(n + k \log m)$ time and $O((n + k \log m) \log n)$ bits of space. □

While the only general bound in terms of n is $O(n + k \log m) \subseteq O(n \log n)$, this bound is reached only in degenerate cases (e.g., one string of length $n/2$ and $n/2$ strings of length 1 or 2). In typical cases, where $m = O(n/k)$, it holds $O(n + k \log m) = O(n)$. This holds even if the largest string in $\mathcal{T}$ is significantly larger than the average, for example $m = O((n/k)^c)$ for a constant c. In practical applications, the worst case is probably that of k short reads whose length ℓ is a few hundreds, and then still $k \log m = (n/\ell) \log \ell$ is an order of magnitude less than n.

4. Refining Our Algorithm

In this section, we explain how to produce smaller temporary data structures during the parsing of T^i. As before, we are interested in a lightweight compression scheme that improves the overall performance of grlBWT, especially when $\mathcal{T}$ is not that repetitive. The challenge is to find a balance between the extra compression we gain for $\mathcal{F}^i$ and T^{i+1} and the computational resources we use to achieve it. A bad choice could yield good space reductions at the cost of making grlBWT slower. On top of that, we also need to be careful not to compromise the induction phase of grlBWT with the changes we introduce.

Our strategy consists of merging consecutive substrings in the LMS parsing of T^i into one *super* phrase whenever the merge does not affect the construction of BWT^i_{bcr}. We formalize this idea as follows:

Definition 4.1. A super phrase is a substring $T^i[j_1..j_z]$ with the following properties: (i) it spans a group of consecutive substrings $F_1 = T^i[j_1..j_2], F_2 = T^i[j_2..j_3], \ldots, F_z = T^i[j_{z-1}..j_z]$ in the LMS parsing whose associated phrases $F_1, F_2, \ldots, F_z \in \mathcal{F}^i$ always appear together in T^i (also as parsing phrases), and in the same order. Thus, for each *oth* occurrence $F_p = T^i[j_{p,o}..j_{p+1,o}]$ of F_p, it always holds that the phrase following that occurrence is $F_{p+1} = T^i[j_{p+1,o}..j_{p+2,o}]$, the *oth* occurrence of F_{p+1}. Additionally, (ii) any of the phrases $F_1, F_2, \ldots, F_{z-1}$ contain a proper suffix that is left-maximal in $\mathcal{F}^i$.

If $F = T^i[j_1..j_z]$ is a super phrase, we store F in $\mathcal{F}^i$ instead of the internal phrases $F_1, F_2, \ldots, F_z$ individually. We refer to F as a *super* phrase. We now explain why super phrases do not affect the induction of BWT^i_{bcr}.

Lemma 4.1. Let $F = T^i[j_1..j_z] \in \mathcal{F}^i$ be a super phrase of T^i. The structure of F does not affect the construction of BWT^i_{bcr} during the induction phase.

Proof. We first prove property (i) of Definition 4.1 by contradiction. Assume that the LMS parsing phrase $F_p = T^i[j_p..j_{p+1}]$ within the super phrase $F = T^i[j_1..j_z]$ matches the sequence of another LMS parsing phrase $F_p = T^i[l..l']$, with $(l, l') \notin [j_1..j_z]$. Additionally, suppose the occurrence $F_p = T^i[l..l']$ does not meet the conditions to be encapsulated within a super phrase, so both F_p and F become members of $\mathcal{F}^i$. In $SA_{\mathcal{F}^i}$ (i.e., the generalized suffix array of $\mathcal{F}^i$), there will be a value that points to the suffix of F prefixed by $F_p = T^i[j_p..j_{p+1}]$ and another value pointing to the full phrase $F_p = T^i[l..l']$. Both suffix array values encode suffixes of $\mathcal{F}^i$ labelled F_p, and hence, our algorithm

will induce the lexicographical order of the suffixes of T^i prefixed by F_p from BWT^{i+1}_{bcr} (see Lemmas 3.6 and 3.7). Now, for the induction to happen, we need two symbols in T^{i+1}, one encoding the occurrence $F_p = T^i[j_p..j_{p+1}]$ and another encoding the occurrence $F_p = T^i[l..l']$. However, $T^i[j_p..j_{p+1}]$ will not have a symbol in T^{i+1} as it is fully encapsulated within F, meaning that we will not be able to perform the induction. This situation does not happen if F_p always occurs in T^i as a substring of F as we have enough context within the super phrase the solve the BWT range associated with F_p.

The proof for property (ii) of Definition 4.1 is similar: assume this time that the set of LMS parsing phrases $F_1, F_2, \ldots, F_z$ encapsulated by the super phrase F meet property (i). However, one of them (say F_u, with $u < z$) has a proper suffix S_x that is left-maximal. That is, S_x also occurs as a proper suffix in, at least, one other phrase $Y \in \mathcal{F}^i$, and the symbols preceding the occurrences of S_x in F_u and Y are different. During parsing iteration i, we do not have enough information in $\mathcal{F}^i$ to order the suffixes of T^i prefixed by S_x. We have the right context for the occurrence of S_x under F_u as F_u is a substring in F (not a suffix), but we do not have the right context for the occurrence under Y. Our method solves this problem in the next parsing iteration $i + 1$ when comparing the symbols in T^{i+1} assigned to F_u and Y. However, with the introduction of super phrases, we have symbols in T^{i+1} for F and Y, but not for F_u, and the right context of F is the right context of F_z, not of F_u. This situation leads to an error during the induction of BWT^i_{bcr} when solving the occurrences of S_x. □

The introduction of a super phrase $F = F_1, F_2, \ldots, F_z$ removes $z - 1$ regular phrases from $\mathcal{F}^i$ and decreases the number of symbols in $\mathcal{F}^i$ by $z-1$. In our regular parsing scheme, the last element of every F_u, with $u < z$, was a copy of the first element of F_{u+1}, because regular LMS parsing phrases overlap, but now that F_u and F_{u+1} belong to the same super phrase, that copy is not in the parsing set. Also, notice that the length of T^{i+1} also decreased by $z - 1$ symbols.

We restricted our definition of super phrases to avoid redundancy in $\mathcal{F}^i$. One could, for instance, allow an LMS parsing phrase F_u to occur in different super phrases as a substring as long as all the occurrences of F_u in T^i were covered by a super phrase. This relaxation could increase the number of super phrases without affecting the induction of BWT^i_{bcr}. However, it could also create multiple copies of F_u within $\mathcal{F}^i$, which is undesirable.

Introducing super phrases requires a small change in the algorithm that

constructs $SA_{\mathcal{F}^i}$. In the original version we described in Section 3.3.3, we first insert the last symbols of every phrase $F \in \mathcal{F}^i$ at the end of its corresponding bucket in $SA_{\mathcal{F}^i}$. Subsequently, we perform one scan of $SA_{\mathcal{F}^i}$ to insert L-type symbols and another scan to insert the S-type symbols. Now, with the introduction of super phrases, we proceed as follows: before the suffix induction, we visit every phrase $F \in \mathcal{F}^i$ and put its last symbol at the end of its corresponding bucket in $SA_{\mathcal{F}^i}$. Additionally, if F is a super phrase, we scan it to find all its internal LMS-type symbols and insert their positions in $\mathcal{F}^i$ at the end of their corresponding buckets in $SA_{\mathcal{F}^i}$. Finally, we can proceed with the induction of the L-type and S-type symbols as usual.

4.1. Practical Considerations of Super Phrases

When parsing T^i, we do not know beforehand if a specific group of consecutive LMS parsing substrings will produce a super phrase. The naive approach to check that information would be to construct a suffix tree of T^i, but this solution is impractical. An alternative idea would be building a regular version of $\mathcal{F}^i$ first, computing some satellite information about the phrases directly from the dictionary, and then performing an extra scan of T^i to gather the super phrases, hoping that there are enough of them so that the overhead of the extra scan produces a much smaller version of $\mathcal{F}^i$ and T^i. Still, there is a third alternative that might not capture all the super phrases, but it is for free. During every parsing round $i-1$, we mark which symbols in the alphabet of T^i are unique. Then, when we scan T^i in the ith parsing round, we proceed as follows: every time we reach an LMS-type symbol $T^i[j]$ (i.e., a break in the parsing), we check if $T^i[j-1]$ is unique. If so, we skip the break and continue extending the current phrase to the left. We keep applying this procedure until we reach a break where the condition does not hold.

It is easy to see that if the symbol $T^i[j-1]$ is unique in T^i, then its enclosing LMS parsing phrase $F = T^i[j'..j]$ spells a sequence that is unique in T^i. On the other hand, $T^i[j-1]$ is the last symbol we consider in F to get its left-maximal suffixes (recall that the last symbol in F is redundant), and because $T[j-1]$ is unique, F does not have left-maximal suffixes. These observations allow us to append $T[j'..j]$ directly to a super phrase without further preprocessing.

5. BCR BWT with Different String Order

One can reorder the strings of $\mathcal{T}$ to reduce the number of runs in its BCR BWT, thus improving the compression. This technique is useful for applications where the order of the strings is irrelevant. In this regard, Bentley et al. [31] showed a linear-time procedure to obtain the smallest BCR BWT (in terms of the number of runs) one can get by permuting the order of the strings in $\mathcal{T}$. This method, referred to here as CAO, requires as input the BCR BWT BWT_{bcr} of $\mathcal{T}$ and the partition $\mathcal{A}$ over BWT_{bcr} that induces equal suffixes of $\mathcal{T}$. More specifically, a block $BWT_{bcr}[s..e]$ belongs to $\mathcal{A}$ if $BWT_{bcr}[s..e]$ stores the left-context symbols for suffixes of $\mathcal{T}$ that spell the same sequence. The output of CAO is the optimal BCR BWT BWT_{bcr} for $\mathcal{T}$, referred to here as BWT_{opt} (Section 2.3 describes CAO in more detail).

In this section, we address the problem of efficiently building $\mathcal{A}$ so that we can apply CAO to transform BWT_{bcr} into BWT_{opt}. Our strategy involves inducing the tuples of $\mathcal{A}$ during the execution of grlBWT. Nevertheless, we compute a sampled version of $\mathcal{A}$ (denoted $\mathcal{A}'$) that only considers BWT blocks (tuples) with more than one distinct symbol. The important observation is that there is no point in keeping in main memory tuples of $\mathcal{A}$ with one symbol during the execution of CAO, as they are already sorted. For instance, the vector $\mathcal{A}$ of Figure 1 becomes $\mathcal{A}' = \ .\ .\ .\ [(\mathtt{a},1),(\mathtt{c},1)]\ .\ .\ [(\mathtt{c},1),(\mathtt{a},2)]\ .$, where the dots indicate the tuples we removed.

The execution of CAO requires an encoding that regards each block $BWT_{bcr}[s..e] \in \mathcal{A}$ as a tuple of up to σ pairs where each element (c, ℓ) stores a symbol c occurring in $BWT_{bcr}[s..e]$ and its frequency ℓ in $BWT_{bcr}[s..e]$. In practice, if we know the boundaries in BWT_{bcr} for the blocks in $\mathcal{A}$, we can compute the CAO encoding on the fly using the run-length compressed version of BWT_{bcr}. However, grlBWT produces the ranges for the blocks in $\mathcal{A}'$, so we need to do a small change to CAO: if for a block $BWT_{bcr}[s..e] \in \mathcal{A}'$ we need to visit its preceding block (respectively, the following block) to check for adjacent matches, and this block is not in $\mathcal{A}'$, we use the symbol $BWT_{bcr}[s-1]$ to do the check instead (respectively, $BWT_{bcr}[e+1]$).

Before explaining our idea, we will briefly redefine some key concepts for clarity. The parsing set $\mathcal{F}^i$ stores the distinct phrases in the LMS parsing of T^i. The suffixes in $\mathcal{F}^i$ form a string set $\mathcal{S}^i$ that induces a partition over GSA^i, the generalized suffix array of T^i. In this partition, every block $GSA^i[s_x..e_x]$ stores the suffixes of T^i prefixed by $S_x \in \mathcal{S}^i$, the xth string of $\mathcal{S}^i$ in LMS order. We say that $BWT^i_{bcr}[s_x..e_x]$ is associated with S_x because it stores its

left-context symbols in T^i. Additionally, $SA_{\mathcal{F}^i}$ is the generalized suffix array of $\mathcal{F}^i$. We also consider a partition over $SA_{\mathcal{F}^i}$, where every block $SA_{\mathcal{F}^i}[j..j']$ encodes the different suffixes of $\mathcal{F}^i$ spelling the same sequence $S_x \in \mathcal{S}^i$. We use $\mathcal{S}^i$ to define an expanded parsing set $\mathcal{F}^i_{exp}$ that contains the elements of $\mathcal{F}^i$ plus the strings in $\mathcal{S}^i \setminus \mathcal{F}^i$ that only occur as left-maximal suffixes in $\mathcal{F}^i$. The elements in the expanded parsing set are kept in LMS order.

5.1. Partitioning the BCR BWT

We will produce the sequence $A^i = (s_1, e_1), \ldots, (s_z, e_z)$ with the blocks in BWT^i_{bcr} induced by the substrings of T^i that expand to equal suffixes of $\mathcal{T}$. Specifically, each jth pair $(s_x, e_x) \in A^i$ is the jth block $BWT^i_{bcr}[s_x..e_x]$ whose string $S_x \in \mathcal{S}^i$ expands to a string $exp(S_x) \in \Sigma^*$ suffixed by \$ (i.e., $exp(S_x)$ is a suffix in $\mathcal{T}$). As before, we will construct a preliminary version of A^i, called pA^i, during the parsing iteration i, and then we will combine A^{i+1} and pA^i to produce A^i during iteration i of induction. The final list A^1 stores the ranges of $\mathcal{A}'$ that we will use to run CAO.

Encoding. We will update Algorithms 3 and 5 to implement the ideas in the paragraph above. Still, modifying these methods requires a bit of extra work as they operate over run-length-compressed data, and the ranges in A^i are indexes in the plain version of BWT^i_{bcr}. This difference in the encoding means that a range in A^i might not exist explicitly in the run-length-compressed version of BWT^i_{bcr}. From now on, we consider the arrays involved in the construction of A^i to be in plain format unless we state otherwise. For example, when we say that u is the *uncompressed* position of BWT^{i+1}_{bcr}, we mean that u is an index within the plain version of BWT^{i+1}_{bcr}. The same idea applies to the other vectors. This plain encoding is only logical and intended to simplify our explanations. We omit the details on how to implement the construction of A^i using run-length-compressed data structures.

5.1.1. Parsing Phase

Our algorithm to construct pA^i is a modification of Algorithm 3. During the execution of Line 6, when we start to consume a new range in the partition of $SA_{\mathcal{F}^i}$, we check if the previous range $SA_{\mathcal{F}^i}[s_x..e_x]$ is associated with a phrase $S_x \in \mathcal{S}^i$ such that $exp(S_x)$ is a suffix in $\mathcal{T}$. If this condition holds, and S_x is a left-maximal suffix in $\mathcal{F}^i$ (Line 10) that does not occur as a full phrase, we append a new pair into pA^i: let s_x be the uncompressed position in $pBWT^i$ where we inserted the leftmost symbol # in $pBWT^i$ associated with

S_x, and let e_x be the uncompressed position in $pBWT^i$ where we inserted the rightmost copy of #. We append the pair (s_x, e_x) into pA^i.

The left-maximal condition of S_x implies that its range $(s_x, e_x) \in pA^i$ will contain more than one distinct symbol in Σ^i. This is the kind of entries CAO needs to sort to produce BWT_{opt}. On the other hand, by not storing (s_x, e_x) when S_x is a full phrase in $\mathcal{F}^i$, we avoid redundancy: assume S_x is assigned the symbol $o \in \Sigma^{i+1}$ in the parsing iteration i. The pair $(s_x, e_x) \in pA^i$ of S_x will be equivalent to the pair $(s_{x'}, e_{x'}) \in pA^{i+1}$ obtained from the string $S'_x = o \in \mathcal{S}^{i+1}$.

Notice that pA^i stores each block $BWT^i_{bcr}[s_x..e_x]$ that later will become the range in BWT^1_{bcr} associated with $exp(S_x)$. CAO will use this block to produce BWT_{opt}.

5.1.2. Induction Phase

We now explain how to modify Algorithm 5 (the induction iteration) to compute A^i. Our method has three main steps. First, it produces a new sequence A^i_P with the pairs of A^i that we will induce from P^i (i.e., the vector of Algorithm 5 storing the sorted left-context symbols of the left-maximal suffixes in $\mathcal{F}^i$). Then, it updates the pairs in A^{i+1} and A^i_P so they reference positions in BWT^i_{bcr}. Finally, it merges A^{i+1}, A^i_P, and pA^i into one single sequence A^i.

We start with the construction of A^i_P during the execution of Lines 2-12, when we populate P^i in one scan of BWT^{i+1}_{bcr}. We logically divide A^i_P into buckets so that each bucket c stores the pairs induced from the bucket c of P^i. As we previously did with P^i, we use the notation $A^i_P[c]$ to refer to the area within A^i_P that contains the ranges of bucket c.

Before the scan of BWT^{i+1}_{bcr}, we set $A^{i+1}[1] = (s, e)$ as the active pair we use to fill A^i_P. We also initialize a set H that will store the different buckets of P we visit during the scan of $BWT^{i+1}_{bcr}[s..e]$. Subsequently, when we visit the symbols of BWT^{i+1}_{bcr}, we proceed as follows. Assume we are in the uncompressed position u of BWT^{i+1}_{bcr}, where $BWT^{i+1}_{bcr}[u] \in \Sigma^{i+1}$ is the symbol assigned to $F \in \mathcal{F}^i$. Also, assume that $u = s$ matches the left element of (s, e). If F occurs as a proper suffix in $\mathcal{F}^i$, we compute its bucket b in P^i, record b in H, and append a new pair (u', u') in $A^i_P[b]$, where u' is the uncompressed position within $P^i[b]$ where we store the left-context of F (Lines 4-6). Then, when we visit every left-maximal suffix $S_x \in \mathcal{S}^i$ of F (Lines 7-10), we apply the same procedure: we compute the bucket b in P^i associated with S_x, record b in H, and append a new pair (u', u') in

$A^i_P[b]$, where u' is the uncompressed position within $P^i[b]$ where we store the left-context of S_x. Later in the scan of BWT^{i+1}_{bcr}, when $s < u \leq e$, we proceed slightly differently: for every new bucket b of P^i we visit during the decompression of F, we check first if b exists in H. If that is the case, we increase the right value in the rightmost pair of $A^i_P[b]$ by one. On the other hand, if b is not in H, we record b in H and append a new pair (u', u') in the bucket $A^i_P[b]$. Additionally, when $u = e$, we flush the content in H and set the next pair in A^{i+1} as the active element we will use from now on to fill A^i_P.

Once we finish the traversal of BWT^{i+1}_{bcr}, we transform the pairs in A^i_P to absolute values. Concretely, a pair (s, e) in the bucket $A^i_P[b]$ becomes $(s + s', e + s')$, where s' is the cumulative number of symbols in the buckets $b' < b$ of P^i.

The next step in the induction iteration is to update the pairs in A^{i+1} and A^i_P so they reference ranges within BWT^i_{bcr}. We carry out this process during the merge of BWT^{i+1}_{bcr}, $pBWT^i$, and P^i (Lines 13-36). Recall that we change the active list of the merge depending on the special symbols we access in the vectors. Similarly, here we change the active list we are updating from A^{i+1} to A^i_P (or vice-versa) depending on whether the active list in the merge is BWT^{i+1}_{bcr} or P^i, respectively.

Assume that, at a given point of the loop in Lines 13-36, BWT^{i+1}_{bcr} becomes the active list (Line 15), and that the next pair to update in A^{i+1} is $A^{i+1}[u_a] = (s, e)$. We first check if the current uncompressed position $BWT^{i+1}_{bcr}[u]$ (i.e., the one that we are consuming in the merge) falls within (s, e). If $u = s$, we set $s = s'$, where s' is the uncompressed position in BWT^i_{bcr} where we store $BWT^{i+1}_{bcr}[u]$. Then, when u equals e, we update e accordingly, increase $u_a = u_a + 1$, and set $A^{i+1}[u_a]$ as the next pair to update in A^{i+1}. Now assume the active symbol in the merge in the xth uncompressed position of P^i (Line 17), and that the next pair we need to update in A^i_P is $A^i_P[u_b] = (s, e)$. In this case, we check if $x = s$ and update the left value $s = s'$, where s' is the uncompressed position in BWT^i_{bcr} where we store the active symbol of P^i. On the other hand, if $x = e$, we update e, increase the index $u_b = u_b + 1$, and set $A^i_P[u_b]$ as the next pair to update in A^i_P.

After we finish updating the values, we merge A^{i+1}, pA^i, and A^i_P in an orderly way to produce A^i. As mentioned, this step only requires a simultaneous scan of the vectors.

6. Experiments

We implemented grlBWT as a C++ tool, also called `grlBWT`. This software uses the `SDSL-lite` library [39] to operate with bit vectors and rank data structures. This implementation includes the improvements we described in Section 4 to reduce the size of the dictionary, but it does not contain the procedure to compute the optimal BWT (Section 5). Our source code is available at `https://github.com/ddiazdom/grlBWT`.

6.1. Implementation Details

Our algorithm constructs a dictionary of phrases in every text T^i and then replaces the occurrences of those phrases with their corresponding symbols in T^{i+1}. These two steps can be challenging to implement in massive inputs as they are linear-time. Using a hash table is a simple alternative to computing the dictionary, but it can impose a considerable overhead (in terms of time and space) if the text is not so repetitive (the less repetitiveness, the bigger the dictionary).

We implement a simple parallel hashing strategy to compute the dictionaries in a more efficient way. In every parsing round i (Section 3.3), we proceed as follows: we first set a buffer size b and the number p of parallel processes we will run. Both b and p are input parameters. Subsequently, we allocate b/p bits (assume for simplicity b is divisible by p) to store a semi-external hash table H_j, with $j \in [1, p]$, for every parallel process. It is semi-external in the sense that every time H_j has to grow beyond b/p bits (either because it exceeds the maximum load factor or because of the insertion of a new pair), it dumps all its content to disk and resets its state to empty. We implement H_j using Robing Hood probing to operate at high load factors.

Once we initialize the hash tables $H_i, \ldots, H_p$, we divide the input text $T^i[1..n_i]$ of the parsing round into p chunks of $\lceil n_i/p \rceil$ symbols each. Every jth parallel process will consume the jth chuck of T^i and store its phrases in its corresponding hash table H_j. When the parallel processes finish, we merge the dumps of the hash tables $H_1, \ldots, H_p$ into one single hash table H that contains all the phrases of T^i.

The reason why this approach is efficient is simple: in the first parsing round, the dictionary is, in most cases, small compared to T^1, so it is likely that the hash tables $H_1, \ldots, H_p$ contain near-identical copies of the same small string set and performed almost no data dumps. This observation

means that the construction of the final hash table (the merge) is fast in practice.

The dictionary size increases considerably in the next parsing rounds, but so does the number of unique phrases (those with frequency one in T^i). Thus, the chances of a phrase appearing in different hash tables decrease as the parsing rounds move forward.

Effects of Page Caching. We remark that our semi-external parsing strategy is efficient only if T^i fits the page cache. That is, the free area of the main memory where the operative system's kernel keeps the recently-accessed file pages. Reading different areas of T^i simultaneously from the disk is costly in standard hard drives as it requires the disk to spin back and forward to reach the sectors where the requested file pages reside. If we have never accessed T^i before, the kernel will inevitably perform these expensive I/O operations. However, `grlBWT` needs three parallel scans of T^i, first when it produces it from T^{i-1}, then when it gets D^i, and finally when it builds T^{i+1}. Thus, if T^i fits the page cache, we will perform the disk operations in the first scan, and the rest will mostly use the pages of T^i in the cache. On the other hand, when T^i does not fit the page cache, the number of page faults[5] increases considerably, but not just that, the operations become slower due to the non-linear disk access pattern of our method. This problem implies that the parallel processes will remain idle most of the time, waiting for the kernel to complete previous page requests, making thus the whole parsing phase slow. This phenomenon of constant paging and page faults is known as cache trashing.

6.2. Datasets

We consider two classes of Genomics collections for our experiments: *reads* and *pangenomes*. The BWT plays a key role in processing this kind of data, but constricting the transform is challenging in practice as reads and pangenomes are usually massive. It is worth mentioning that `grlBWT` works with any kind of byte alphabet, not just DNA.

Reads are overlapping strings that represent random and redundant fragments of a genome. The level of redundancy depends on the length of the reads and the *coverage*: the average number of times each position of the

[5]When an process asks for a file page, but that page is not in the cache and the kernel has to access the disk to retrieve it.

genome was sequenced. The more coverage the sequencing experiment has, the more repetitive the read collection is. The number of DNA samples also affects the repetitiveness. It is common in Genomics to concatenate reads of closely-related individuals into one dataset. These individuals are genetically almost identical, so their reads should yield nearly identical sequences. A common problem with reads, however, is that they are short and contain sequencing errors. These limitations make the repetitive patterns of the underlying genome more difficult to capture and compress.

A genome (from a Bioinformatics point of view) is a string collection resulting from the assembly[6] of a group of reads. A pangenome, on the other hand, is a collection that can contain several assembled genomes of closely-related individuals. A pangenome is massive and highly repetitive, but problems in the assembly process and sequencing errors can break the repetitive patterns, making the collection less compressible.

We now briefly describe our datasets. Table 1 presents basic statistics about these files.

- *Illumina* (`illx`): five collections of Illumina reads[7] generated from different human genomes. The name of each collection has the format `illx`, where `x` indicates the number of individuals in the collection. For instance, the file `ill5` encodes reads from five humans. We obtained the raw reads from the International Genome Project [8].

- *PacBio HiFi* (`pbhf`): one read collection from one human genome sequenced at deep coverage (40x) using the PacBio HiFi technology[9]. HiFi reads are longer than Illumina reads; hence, they are more repetitive.

- *Human Pangenomes* (`hgx`): 400 different human assemblies from NCBI[10] grouped into different files to simulate six different pangenomes. Every pangenome file has the format `hgx`, indicating that this collection contains `x` assemblies. The sources of this data are varied: some are different assemblies of the same genome, while others are assemblies of

[6]Merging the reads by computing suffix-prefix overlaps

[7]`https://www.illumina.com`

[8]`https://www.internationalgenome.org/data-portal/data-collection/hgdp`

[9]`https://www.pacb.com/technology/hifi-sequencing`

[10]`https://www.ncbi.nlm.nih.gov/data-hub/genome/?taxon=9606`

Dataset	σ	Number of strings	Max. str. length	Avg. str. length	Number of symbols (n)	n/r
`ill1`	5	8.4×10^7	151	151	1.3×10^{10}	3.18
`ill2`	5	1.6×10^8	151	151	2.4×10^{10}	4.07
`ill3`	5	2.4×10^8	151	151	3.6×10^{10}	4.67
`ill4`	5	3.1×10^8	151	151	4.7×10^{10}	5.03
`ill5`	5	3.8×10^8	151	151	5.7×10^{10}	5.33
`pbhf`	5	6.2×10^6	5.0×10^4	2.0×10^4	1.2×10^{11}	19.27
`hg05`	16	3.3×10^5	2.5×10^8	4.3×10^4	1.4×10^{10}	4.82
`hg10`	16	7.6×10^5	2.5×10^8	3.9×10^4	3.0×10^{10}	8.76
`hg15`	16	8.4×10^5	2.5×10^8	5.4×10^4	4.5×10^{10}	12.02
`hg20`	16	8.7×10^5	2.5×10^8	6.9×10^4	6.0×10^{10}	15.67
`hg25`	16	9.0×10^5	2.5×10^8	8.3×10^4	7.5×10^{10}	19.42
`hg400`	16	1.0×10^7	2.5×10^8	1.2×10^5	1.2×10^{12}	224.40
`ecg31k`	16	6.7×10^5	5.7×10^6	2.8×10^4	1.9×10^{10}	140.02

Table 1: Datasets. The rightmost column shows the ratio between n and the number of runs (r) in the BCR BWT obtained without changing the order of the strings.

different genomes (different humans and/or cell lines). The quality of the assemblies is also heterogeneous. Some are high quality, while others are poor or intermediate reconstructions. The file `hg400` encodes 400 assemblies, and it is the largest input in our experiments (1.2 TB). This file is the one that most closely resembles a real-life pangenome, given its massiveness and the variability in the strings it contains.

- *Escherichia Coli Pangenome* (`ecg31k`): 31,733 different assemblies of the Escherichia coli (E. coli) genome downloaded from NCBI[11]. As with humans, these assemblies come from different sources, and their qualities are variable. This file is highly repetitive but much smaller than our artificial pangenome `hg400` as the E. coli genome is small.

6.3. Competitor Tools and Experimental Setup

We compared the performance of `grlBWT` against other tools that compute BWTs for string collections:

[11] `https://www.ncbi.nlm.nih.gov/data-hub/genome/?taxon=562`

- `ropebwt2`[12]: a variation of the original BCR algorithm of Bauer et al. [12] that uses rope data structures [40]. This method is described in Heng Lee [41].
- `pfp-eBWT`[13]: the eBWT algorithm of Boucher et al. [21] that builds on PFP and ISS.
- `r-pfpbwt`[14]: implementation of the method of Oliva et al. [28] that applies recursive rounds of PFP.
- `BCR_LCP_GSA`[15] : the current implementation of the semi-external BCR algorithm [12].
- `egap`[16]: a semi-external algorithm of Edigi et al. [14] that builds the BCR BWT.
- `gsufsort`[17]: an in-memory method proposed by Louza et al. [13] that computes the BCR BWT and (optionally) other data structures.

We also considered the tool `bwt-lcp-em` [15] for the experiments. Still, by default, it builds both the BWT and the LCP array, and there is no option to turn off the LCP array, so we discarded it. We compiled all the tools according to their authors' descriptions. For `grlBWT`, we used the compiler flags `-O3 -msse4.2 -funroll-loops -march=native`.

Experiments on Reads. We ran `grlBWT`, `ropebwt2`, `egap`, `gsufsort`, `BCR_LCP_GSA`, and `pfp-bwt` on Illumina data. We did not use `r-pfpbwt` as it is unsuitable for short reads. We limited the RAM usage of `egap` to three times the input size. For `BCR_LCP_GSA`, we turned off the construction of the data structures other than the BCR BWT and left the memory parameters by default. In the case of `gsufsort`, we used the flag `-bwt` to build only the BWT. For `ropebwt2`, we set the flag `-L` to indicate that the data was in one-sequence-per-line format, and the flag `-R` to avoid considering the DNA reverse strands

[12] `https://github.com/lh3/ropebwt2`
[13] `https://github.com/davidecenzato/PFP-eBWT`
[14] `https://github.com/marco-oliva/r-pfbwt`
[15] `https://github.com/giovannarosone/BCR_LCP_GSA`
[16] `https://github.com/felipelouza/egap`
[17] `https://github.com/felipelouza/gsufsort`

in the BWT. We ran the experiments on the Illumina reads using one thread in all programs because not all support multi-threading. For this purpose, we set the extra flag `-P` to `ropebwt2` to indicate single-thread execution. Figure 9 summarises the results of our experiment on Illumina data. We only tested `grlBWT` and `ropebwt2` with the `pbhf` dataset (HiFi reads) as `egap`, `BCR_LCP_GSA`, and `gsufsort` are unsuitable for long strings. We did not use `r-pfpbwt` with `pbhf` either because we assumed it would exceed our available resources. We based our conclusions on the results we obtained with `r-pfpbwt` on the human pangenomes. Both `grlBWT` and `ropebwt2` support multi-threading, so we used four threads in both.

Experiments on Small Human Pangenomes. We assessed the performance of `ropebwt2`, `grlBWT`, `pfp-ebwt`, and `r-pfpbwt` in the small human pangenomes (files `hg5-25`). As with `pbhf`, we did not report experiments on the tools tailored for short strings[18] By default, `ropebwt2` uses four working threads, so we set the same number of threads for `grlBWT`, `pfp-ebwt`, and `r-pfpbwt`. The tool `r-pfp` has three steps, each requiring a different set of parameters. In the first step (`pfp++`), we used `-w 10 -p 71`. In the second one (recursive `pfp++`), we used `-w 5 -p 11`. Finally, we used `rpfpbwt64` with the parameter `-bwt-only` to produce the BWT. The input parameters for `ropebwt2` were the same as with Illumina data, except for the flag `-P`. We ran `pfp-ebwt` with default parameters. We did not report experiments with `pfp-ebwt` and `ill25` as their execution crashed. Our results on small human pangenomes are shown in Figure 10.

Experiments on the E. coli Pangenome. The file `ecg31k` (i.e., the E. coli pangenome) is particularly repetitive and not so big (see Table 1), so we used it to assess the performance improvement one could obtain in the BWT construction under highly-repetitive scenarios. We also used it to evaluate the impact of our parallel method (Section 6.1) as this file fits the page cache of our machine. Thus, we limited our experiments on `ecg31k` to the tools `ropebwt2`, `pfp-ebwt`, `r-pfpbwt`, and `grlBWT`. We included `ropebwt2` as a baseline because it is the non-repetition-aware tool (i.e., it does not exploit repetitions) with the best performance. We ran each software twice, one execution with one thread and the other with four threads. Figure 11

[18]We tried to test them, but they either crushed or their resource consumption was too high to compare against the other tools.

presents the results on the E. coli pangenome.

Experiments on the Big Human Pangenome. We evaluated the performance of `grlBWT` in the big human pangenome (`hg400`). We measured the running time and memory consumption of every round of `grlBWT` to look for potential problems that are not evident in small and repetitive instances (like `ecg31k`). We used 10 threads and a buffer for the parallel hash tables (Section 6.1) whose size in RAM is 10% of the input (near 120 GBs). The selection of 10% for the buffer was arbitrary, and it is an input parameter. We did not perform experiments on `hg400` with the other competitor tools because of the high computational resources they would require. Figures 14, 12, and 13 show the running time and memory usage of `grlBWT` with `hg400`.

Experiments on Super Phrases. We assessed the impact of super phrases in the LMS parsing. First, we ran our current implementation of `grlBWT`, which computes super phrases as described in Section 4.1, and then we ran our CPM'22 version[19] of `grlBWT` , which does not include super phrases. We ran both implementations with the inputs `hg10` (repetitive) and `ill1` (not so repetitive), recording the number of phrases in each $\mathcal{F}^i$ as well as its number of symbols $||\mathcal{F}^i||$. The results are shown in Figure 15.

Machine. We carried out the experiments on a machine with Debian 4.9, 736 GB of RAM, and processor Intel(R) Xeon(R) Silver @ 2.10GHz, with 32 cores.

7. Results and Discussion

7.1. Illumina and HiFi Reads

The fastest method in Illumina reads was `ropebwt2`, with a mean elapsed time of 4.14 hours. It is then followed by `grlBWT`, `gsufsort`, `BCR_LCP_GSA`, `pfp-bwt`, and `egap`, with mean elapsed times of 6.08, 9.43, 9.58, 13.08, and 27.30 hours, respectively (Figure 9B). Regarding the working space, the most efficient was `BCR_LCP_GSA`, with an average memory peak of 5.73 GB. It is then followed by `grlBWT` and `ropebwt2`, with average memory peaks of 23.95 and 26.64 GBs, respectively. In both cases, the memory consumption increases slowly with the input size (see Figure 9A). In contrast, `egap`,

[19]Pre-release v1.0.0-alpha in our GitHub repository.

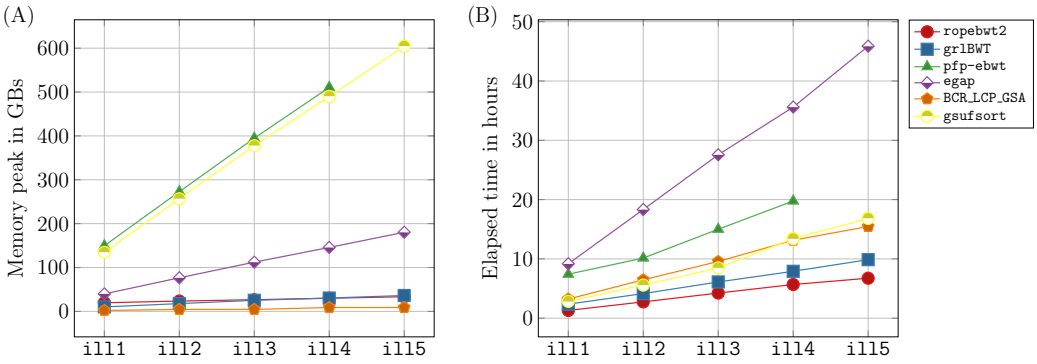

Figure 9: Memory peak usage (GBs) and elapsed time (in hours) for the Illumina collections.

`gsufsort`, and `pfp-ebwt` are far more expensive, and their memory consumption grows fast. The tool `egap` uses 110.94 GBs on average. On the other hand, `pfp-ebwt` and `gsufsort` have similar average memory peaks: 331.98 and 372.68 GBs, respectively. We notice `grlBWT` is the tool with the second-best overall performance, only outperformed in time by `ropebwt2` and in space by `BCR_LCP_GSA`. We consider this result remarkable as `grlBWT` does not perform any optimization on short reads. Besides, the repetitive patterns (the main feature `grlBWT` uses to reduce CPU time and space consumption) are highly fragmented in short reads.

One possible explanation for why we outperformed even the in-memory tool `gsufsort` is because `grlBWT` is less likely to have cache misses as it operates over small data structures. In particular, `gsufsort` resembles SA-IS, so it runs ISS over each T^i. The problem is that the cache misses triggered by the induction of distant suffix array buckets affect the running time, and the longer T^i is, the more cache misses we trigger. Our method, in contrast, performs ISS over the parsing set $\mathcal{F}^i$, which is considerably smaller than T^i, making cache misses far less likely.

Our results on HiFi reads (`pbhf`) differ from what we obtained with Illumina. The elapsed time for `grlBWT` was 8.48 hours, almost half the time spent by `ropebwt2` (16.02 hours). Still, they performed similarly in terms of memory peak: 25.15 GB of `grlBWT` versus 27.20 GB of `ropebwt2`.

We believe there are two reasons for our results on reads. The first reason is that `ropebwt2` is highly optimized for short reads, but not for long strings. The second reason is that HiFi reads are longer than Illumina reads, so

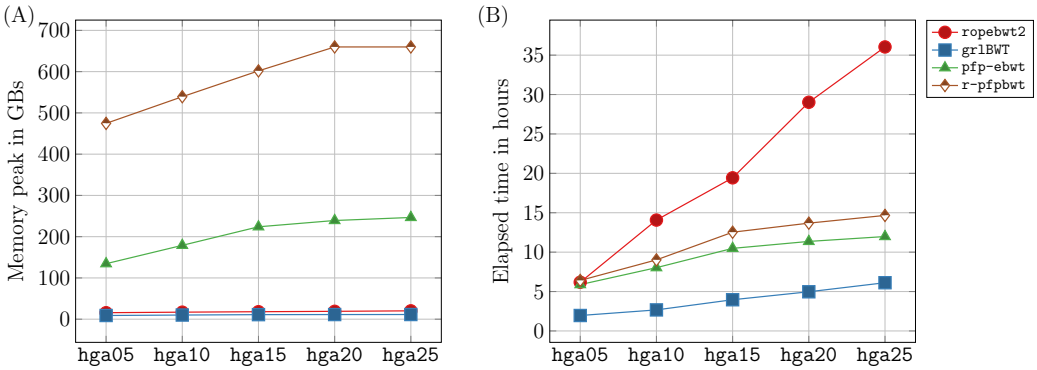

Figure 10: Memory peak usage (GBs) and elapsed time (in hours) for the small human pangenomes.

`grlBWT` can capture repetitive patterns more efficiently.

7.2. Small Human Pangenomes

Our tool `grlBWT` was the fastest software in small human pangenomes, with an average elapsed time of 3.94 hours versus 9.55, 20.95, and 11.26 hours for `pfp-ebwt`, `ropebwt2`, and `r-pfpbwt`, respectively. The time for `grlBWT`, `pfp-ebwt`, and `r-pfpbwt` grows smoothly with the input size, while the time for `ropeBWT` grows fast (see Figure 10B). These patterns of growth are because `grlBWT`, `pfp-ebwt`, `r-pfpbwt` exploit the text repetitions, while `ropeBWT2` does not.

Regarding memory peak, `grlBWT` is also the most efficient tool, with a mean of 10.52 GB versus 18.05, 204.62, and 587.05 GBs for `ropebwt2`, `pfp-ebwt` and `r-pfpbwt`, respectively. Although `grlBWT` outperforms `ropebwt2` on average, their memory functions have the same pattern: both grow smoothly with the input size. In contrast, the memory consumption of `pfp-ebwt` and `r-pfpbwt` is considerable, although they still have a smooth pattern of growth (see Figure 10A). We did not expect `pfp-ebwt` and `r-pfpbwt` to have a high memory consumption as they also exploit the text repetitions. Still, we acknowledge this result could be because we did not choose suitable input parameters or because the implementations of `pfp-ebwt` and `r-pfpbwt` are still incomplete. It might also be possible that the parsing scheme of `grlBWT` is better than PFP at capturing text repetitions.

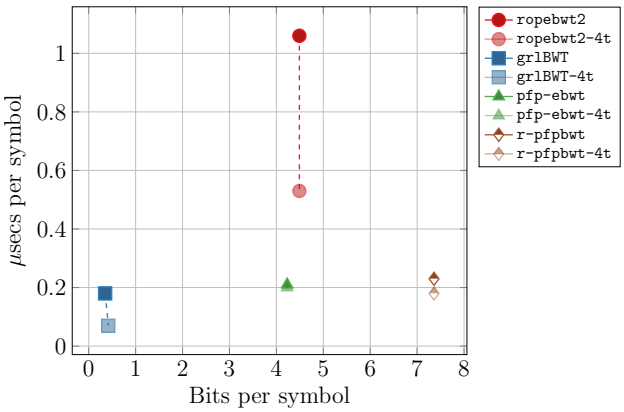

Figure 11: Time-space tradeoffs for constructing the BWT of `ecg31k` (E. coli pangenome). Time is given in microseconds per symbol and space in bits per symbol. The transparent points represent the instances using four threads (`-4t` suffix in the legend).

7.3. E. coli Pangenome

The performance of `grlBWT` is considerably better than that of `ropebwt2`, `pfp-ebwt`, and `r-pfpbwt` in highly-repetitive inputs (`ecg31k`). Using one thread, the elapsed time of `grlBWT` with `ecg31k` was 0.93 hours, while the elapsed times of `ropebwt2`, `pfp-ebwt`, and `r-pfpbwt` were 5.56, 1.08, and 1.19 hours, respectively. Thus, the average speed of `grlBWT` was 0.18 μsecs per symbol, while the average speed of the other tools was 1.06, 0.21, and 0.23 μsecs per symbol (respectively). We were also the most space-efficient method, with a memory peak of 0.82 GB (0.35 bits per symbol) for `grlBWT` versus memory peaks of 10.57, 9.95, and 17.30 GBs for `ropeBWT2`, `pfp-ebwt`, and `r-pfpbwt`, respectively. Our experiments on `ecg31k` also showed we could greatly improve our running time if we use parallelization. Four threads were enough to reduce `grlBWT`'s running time by more than half, from 0.93 hours to 0.34 hours (around 20 minutes). This improvement in the speed had a negligible impact on the memory peak as it increased from 0.82 GB to 0.99 GB. The only other tool that improved its performance significantly with parallelism was `ropebwt2`. It decreased its running time from 5.56 hours to 2.75 hours without changing its memory peak. However, its results were far from what we obtained with `grlBWT`. See Figure 11 for more details on the E. coli experiments.

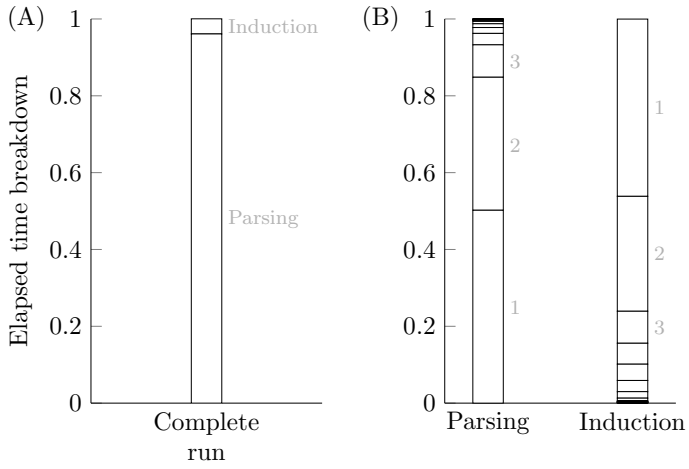

Figure 12: Elapsed time breakdown of `grlBWT` using the file `hg400`. (A) Breakdown of the phases. The bottom box is the parsing phase, and the upper box is the induction phase. The y-axis denotes the fraction of the total running time. (B) Breakdown of the rounds. Each box denotes one round in a phase. The y-axis, in this case, is the fraction of the total phase's running time. The rounds in the parsing phase (left bar) are read bottom-up, while the rounds in the induction phase (right bar) are read top-bottom. The numbers in grey to the right of the bars highlight the three most time-consuming rounds.

7.4. Big Human Pangenome

The complete execution of `grlBWT` with the big human pangenome (file `hg400`) took 41.21 hours and had a memory peak of 118.83 GB. Further inspection of this execution showed that the parsing phase of our algorithm (Section 3.3) contributed to 96.1% of the total running time of `grlBWT`, while the induction phase (Section 3.5) contributed to the remaining 3.94% (see Figure 12A). Additionally, the first three parsing rounds contributed 93.2% of the total running time of the parsing phase, with the first parsing round contributing more than 50% (see Figure 12B). These results indicate the bottleneck of the execution was in these rounds. A closer examination of the steps of the parsing rounds one, two, and three shows that transforming T^1 into T^2 is the most expensive step in the whole execution of `grlBWT` (see Figure 13).

We believe this problem arises due to poor management of the page caches. Our tool `grlBWT` keeps T^1 and T^2 mostly on disk, loading small chunks of them (pages) into main memory to produce T^2 in a semi-external way. As explained in Section 6.1, the Linux kernel speeds up disk accesses to

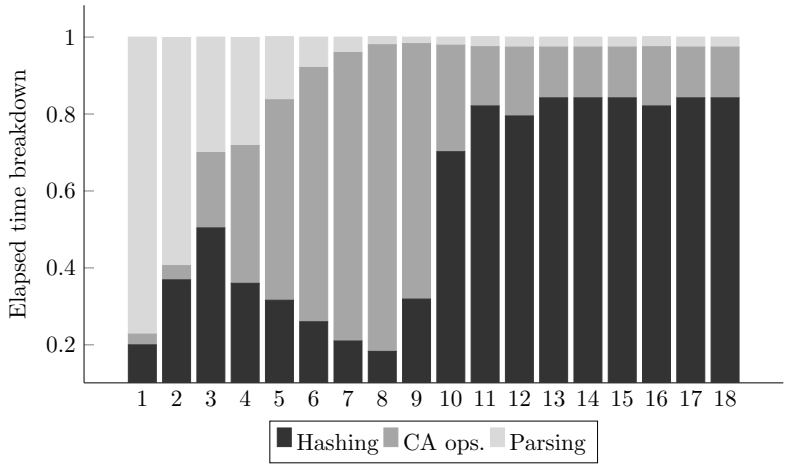

Figure 13: Elapsed time breakdown of the parsing phase of `grlBWT` when executed with the input `hg400` (i.e., the big human pangenome). Each *ith* bar (x-axis) represents the time breakdown of the *ith* round of parsing. "Hashing" (black box) refers to scanning T^i and inserting its LMS phrases into a hash table. "CA ops." (grey box) denotes the time spent performing compression-aware operations. That is, producing the dictionary's suffix array and the preliminary BCR BWT, compressing the dictionary, and assigning symbols to the dictionary phrases. Finally, "Parsing" (light grey box) is the time spent transforming T^i into T^{i+1}.

these files by keeping recently-accessed pages cached in free RAM sections so they are available for future disk accessions. However, the problem in `hg400` arises because T^1 and T^2 do not fit the page cache, so `grlBWT` triggers disk operation frequently due to page faults, making the parallel semi-external scans of T^1 and T^2 extremely slow. Despite the problem with T^1 and T^2, we note that the steps of `grlBWT` that operate over compressed data are remarkably efficient in terms of both time and space (see Figures 12 and 13). At the end of this section, we propose an alternative solution to parse T^i under page cache constraints.

The memory peak in the execution of `grlBWT` is dominated by the buffer of the parallel hash tables we use to construct the dictionary from the text (Section 6.1). Notice the peak is 118.83 GB because we defined a buffer for these hash tables that uses at most 10% of `hg400` (about 120 GB). Put another way, the memory peak of `grlBWT` is a user-defined parameter when we process a large file in parallel.

In practice, however, we are interested in the *real* memory peak of our

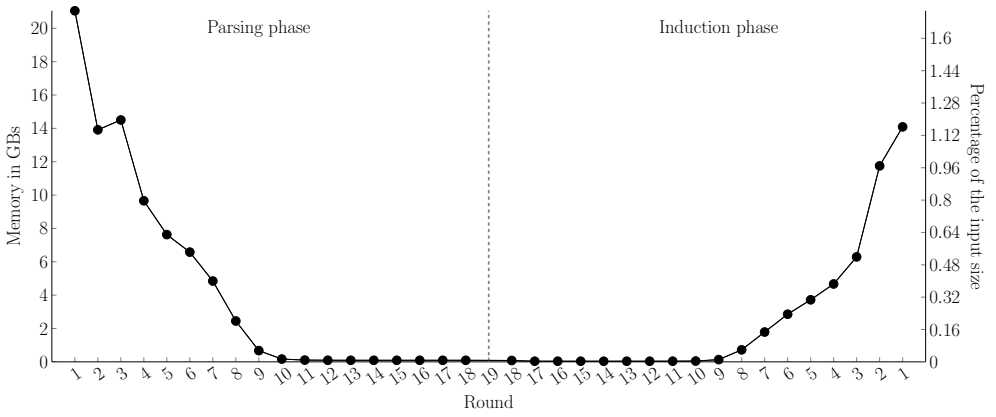

Figure 14: Heap memory usage for the intermediate data structures when running `grlBWT` with `hg400`. The x-axes are the iterations of grlBWT. The vertical dashed line marks the transition from the parsing phase to the induction phase. The left y-axis is the usage in GBs, while the right y-axis is the usage as a *percentage* of `hg400` in plain format. That is, the total bytes of the intermediate data structures divided by the bytes of the input (one byte per input symbol). The induction iterations are numbered backwards to match the way in which grlBWT works. The heap usage of every parsing iteration i considers $SA_{\mathcal{F}^i}$, V^i, and $D^i = (R^i, L^i, N^i)$ (Section 3.3) plus other minor data structures. The heap usage of every induction iteration i considers V^i, P^i, and G^i (Section 3.6).

implementation. That is, the memory footprint produced by the data structures we keep in the heap (except for the aforementioned buffer, whose sole purpose is to enable a parallel execution). In every parsing round i, these data structures are the dictionary D^i and its suffix $SA_{\mathcal{F}^i}$ (plus some other minor data structures). On the other hand, during every induction phase i, the most relevant structures are P^i, the vector where we insert the symbols of BWT^i_{bcr} that we induce from BWT^{i+1}_{bcr} (see Section 3.6), and G^i, the grammar-like encoding of the expanded parsing set $\mathcal{F}^i_{exp}$.

A close inspection of the memory footprint of the compressed data structures showed that the real memory peak of `grlBWT` is remarkably low compared to the input size: 21.05 GB (1.7% of `hg400`'s size) during the first parsing round, and then another smaller peak of 14.09 GB (1.1% of `hg400`'s size) during the construction of BWT^1_{bcr} from BWT^2_{bcr} in the last induction round (see Figure 14 for more details).

Parsing the Text Under Page Cache Constraints. We could tackle the problem of parsing a string T^i that does not fit the page cache using a parallel procedure that implements a producer-consumer pattern. As before, assume

we are allowed to use p processes. We first initialize in main memory a set of $b > p$ buffers of d bits each (b and d being parameters). Then, we create a producer process t_p that reads chunks of T^i semi-externally (and linearly) and puts them in the available buffers. After t_p fills a buffer, it appends it into a queue I flagged as "ready to be processed". On the other hand, we create a set of consumer processes $t_{c,1}, \ldots, t_{c,p-1}$ that actively check the state of I to see if there are available chunks. When a consumer process $t_{c,j}$ pops a buffer from I, it parses its text using LMS parsing and then appends the consumed buffer into another queue O that keeps the already processed data. Thus, once t_p uses all the available buffers, it pops elements of O to recycle buffers for new chunks of T^i, which appends into I and the cycle begins again. As the consumer processes parse T^i in parallel, they insert the phrases into one hash table H that uses lock-free CPU instructions to support concurrent queries. We have to choose b and d carefully so t_p is always reading from disk, while the processes $t_{c,1}, \ldots, t_{c,p-1}$ constantly consume chunks of T^i. This idea is more efficient than the scheme we presented in Section 6.1 as it almost removes the need for a page cache. No matter how many disk accesses the producer process performs, the consumer processes do not remain idle.

7.5. Effect of Super Phrases

Our heuristic of super phrases (Section 4.1) has a notorious impact between parsing iterations three and five. In `hg10`, the number of phrases in $\mathcal{F}^3$, $\mathcal{F}^4$, and $\mathcal{F}^5$ reduced by 14.4%, 28.2%, and 36.7% (respectively) when using our heuristic (top-left plot in Figure 15). In `ill1`, the reductions in those iterations were 17.8%, 32.8%, and 6.6%, respectively (top-right plot in Figure 15). However, our heuristic fails in iteration two (the one producing the largest $\mathcal{F}^i$) as it achieves a negligible reduction in both inputs. The reason could be that T^2 still has several repeated symbols, so our simple method, which relies on symbol frequencies to capture super phrases, does not work. Text T^3 is more likely to have unique symbols, so our heuristic probably works better than in T^2. This result is relevant as $\mathcal{F}^3$ is the second-largest parsing set in both `hg10` and `ill1`. Additionally, we noticed that the reduction of $\mathcal{F}^3$ is better in `ill1` than in `hg10` (17.8% versus 14.4%). A possible explanation is that `ill1` is less repetitive than `hg10`, so our heuristic becomes more efficient. Regarding the number of symbols in each $\mathcal{F}^i$ (second row of Figure 15), super phrases do not have a relevant impact. In `hg10`, the number of symbols in $\mathcal{F}^3, \mathcal{F}^4$, and $\mathcal{F}^5$ reduced by 2.4%, 13.3%, and

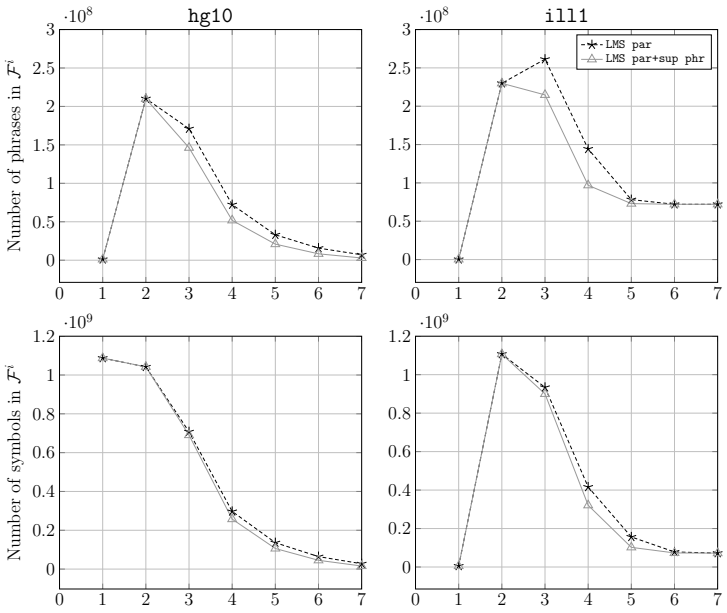

Figure 15: Effect of super phrases in the first seven iterations of LMS parsing. The left column shows the results for `hg10` and the right column shows the results for `ill1`. In all the plots, the x-axis is the parsing iteration (e.g., the information of $\mathcal{F}^3$ is in $x = 3$). The y-axes in the first row show the number of phrases in $\mathcal{F}^i$, with $1 \leq i \leq 7$. The plots in the second row show the total number of symbols $||\mathcal{F}^i||$. The dashed line with star shapes is LMS parsing without super phrases (`LMS par`), and the grey line with triangle shapes is the LMS parsing with super phrases (`LMS par+sup phr`).

21.9%, respectively. In contrast, in `ill1`, the reductions were 3.6%, 22.8%, and 34.6%, respectively (slightly better than in `hg10`). The compression of $\mathcal{F}^5$ seems remarkable (21.9% and 34.6%), but keep in mind that this parsing set is considerably smaller than $\mathcal{F}^2$ and $\mathcal{F}^3$ in both inputs, so the overall space reduction is not big after all. We expected these results, as we remove p symbols from $\mathcal{F}^i$ for each sequence of p consecutive parsing phrases we merge into one single super phrase, which is not much.

8. Concluding Remarks

We introduced a method for building the BCR BWT that maintains the data of intermediate stages in compressed form. The representation we chose reduces not only working memory but also computation time. Our experimental results showed that our algorithm is competitive with the state-of-

the-art tools under not-so-repetitive scenarios and greatly reduces the computational requirements when the input becomes more repetitive. This last feature proved an efficient solution for processing terabytes of redundant data under limited computational resources. For now, the hard drive is the main aspect that degrades our performance in large inputs and prevents us from running even more significant collections. However, we are confident we can solve these problems with a more careful implementation of our algorithm.

An important observation is that our framework enables the construction of the r-index [5] for large collections in practice, as it is possible to obtain this data structure in $O(r)$ bits using the BWT as input. This idea certainly facilitates the indexation of large-scale pangenomes. However, more is needed to make the r-index a practical solution for pangenomes as it does not support all the relevant queries necessary for Genomics analyses.

We believe it is possible to use our repetition-aware strategy for other operations. For instance, update and merge multiple BWTs. The intuition to merge BWTs is that if we have several texts, we first run the parsing round of grlBWT independently in each of them to produce a list of dictionaries (one dictionary set for each text). Then, we combine the dictionaries in one set, and finally, we run the induction phase of grlBWT over the combined dictionary set. Our experiments showed that manipulating dictionary sets and running the induction phase of grlBWT is fast, even in terabytes of data. Thus, the merge of the BWTs should be fast too. The update of a BWT should work similarly. We first produce an initial BWT by running grlBWT over a text collection and save its dictionary set. Then, if we need to append more sequences to this BWT, we run the parsing phase of grlBWT on the new sequences to produce a new dictionary set, which we combine with the one we previously saved. As before, we run the induction phase over the combined dictionary set to produce the updated BWT. We can keep the combined dictionary set again if we need to append more sequences in the future. These ideas (merge and update) could enable the efficient construction of huge BWTs in distributed systems.

There are also other applications for our compression-aware technique we would like to explore, not just BWT-related topics. For instance, computing all-vs-all maximal exact matches in string collections, grammar or Lempel-Ziv compression, self-indexes, and approximate or multiple alignments, among other things.

Acknowledgements

Funded in part by Basal Funds FB0001, ANID, Chile.

References

[1] M. Burrows, D. Wheeler, A block sorting lossless data compression algorithm, Tech. Rep. 124, Digital Equipment Corporation (1994).

[2] E. Ohlebusch, Bioinformatics Algorithms: Sequence Analysis, Genome Rearrangements, and Phylogenetic Reconstruction, Oldenbusch Verlag, 2013.

[3] V. Mäkinen, D. Belazzougui, F. Cunial, A. Tomescu, Genome-Scale Algorithm Design, Camb. U. Press, 2015.

[4] P. Ferragina, G. Manzini, Opportunistic data structures with applications, in: Proc. 41st Annual Symposium on Foundations of Computer Science (FOCS), 2000, pp. 390–398.

[5] T. Gagie, G. Navarro, N. Prezza, Fully-functional suffix trees and optimal text searching in BWT-runs bounded space, Journal of the ACM 67 (1) (2020) article 2.

[6] T. Gagie, G. Manzini, J. Sirén, Wheeler graphs: A framework for BWT-based data structures, Theoretical Computer Science 698 (2017) 67–78.

[7] B. Langmead, C. Trapnell, M. Pop, S. Salzberg, Ultrafast and memory–efficient alignment of short DNA sequences to the human genome, Genome Biology 10 (3), article R25 (2009).

[8] H. Li, R. Durbin, Fast and accurate long–read alignment with Burrows–Wheeler transform, Bioinformatics 26 (5) (2010) 589–595.

[9] P. Weiner, Linear pattern matching algorithms, in: Proc. 14th Annual Symposium on Switching and Automata Theory (SWAT), 1973, pp. 1–11.

[10] U. Manber, G. Myers, Suffix arrays: a new method for on–line string searches, SIAM Journal on Computing 22 (5) (1993) 935–948.

[11] D. Okanohara, K. Sadakane, A linear–time Burrows–Wheeler transform using induced sorting, in: Proc. 16th International Symposium on String Processing and Information Retrieval (SPIRE), 2009, pp. 90–101.

[12] M. J. Bauer, A. J. Cox, G. Rosone, Lightweight algorithms for constructing and inverting the BWT of string collections, Theoretical Computer Science 483 (2013) 134–148.

[13] F. A. Louza, G. P. Telles, S. Gog, N. Prezza, G. Rosone, gsufsort: constructing suffix arrays, LCP arrays and BWTs for string collections, Algorithms for Molecular Biology 15, article 18 (2020).

[14] L. Egidi, F. A. Louza, G. Manzini, G. P. Telles, External memory BWT and LCP computation for sequence collections with applications, Algorithms for Molecular Biology 14 (2019) 6.

[15] P. Bonizzoni, G. Della Vedova, Y. Pirola, M. Previtali, R. Rizzi, Computing the multi–string BWT and LCP array in external memory, Theoretical Computer Science 862 (2021) 42–58.

[16] Z. D. Stephens, S. Y. Lee, F. Faghri, R. H. Campbell, C. Zhai, M. J. Efron, R. Iyer, M. C. Schatz, S. Sinha, G. E. Robinson, Big data: astronomical or genomical?, PLoS Biology 13 (7) (2015) e1002195.

[17] D. Kempa, T. Kociumaka, String Synchronizing Sets: Sublinear–Time BWT Construction and Optimal LCE Data Structure, in: Proc. 51st Annual ACM SIGACT Symposium on Theory of Computing (STOC), 2019, p. 756–767.

[18] C. Boucher, T. Gagie, A. Kuhnle, B. Langmead, G. Manzini, T. Mun, Prefix–free parsing for building big BWTs, Algorithms for Molecular Biology 14, article 13 (2019).

[19] D. Kempa, Optimal construction of compressed indexes for highly repetitive texts, in: Proc. 30th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), 2019, pp. 1344–1357.

[20] D. Kempa, T. Kociumaka, Resolution of the Burrows–Wheeler transform conjecture, in: Proc. 61st Annual Symposium on Foundations of Computer Science (FOCS), IEEE, 2020, pp. 1002–1013.

[21] C. Boucher, D. Cenzato, Z. Lipták, M. Rossi, M. Sciortino, Computing the original eBWT faster, simpler, and with less memory, in: Proc. 28th International Symposium on String Processing and Information Retrieval (SPIRE), 2021, pp. 129–142.

[22] R. Karp, M. Rabin, Efficient randomized pattern–matching algorithms, IBM Journal of Research and Development 31 (2) (1987) 249–260.

[23] D. S. N. Nunes, F. A. Louza, S. Gog, M. Ayala-Rincón, G. Navarro, A grammar compression algorithm based on induced suffix sorting, in: Proc. 28th Data Compression Conference (DCC), 2018, pp. 42–51.

[24] D. Díaz-Domínguez, G. Navarro, A grammar compressor for collections of reads with applications to the construction of the BWT, in: Proc. 31st Data Compression Conference (DCC), 2021, pp. 83–92.

[25] J. Larsson, A. Moffat, Off–line dictionary–based compression, Proceedings of the IEEE 88 (11) (2000) 1722–1732.

[26] P. Ko, S. Aluru, Space efficient linear time construction of suffix arrays, Journal of Discrete Algorithms 3 (2-4) (2005) 143–156.

[27] D. Díaz-Domínguez, G. Navarro, Efficient construction of the BWT for repetitive text using string compression, in: Proc. 33rd Annual Symposium on Combinatorial Pattern Matching (CPM), 2022, p. article 29.

[28] M. Oliva, T. Gagie, C. Boucher, Recursive prefix-free parsing for building big BWTs, bioRxiv (2023) 2023–01.

[29] J. C. Kieffer, E. H. Yang, Grammar–based codes: a new class of universal lossless source codes, IEEE Transactions on Information Theory 46 (3) (2000) 737–754.

[30] F. Shi, Suffix arrays for multiple strings: a method for on-line multiple string searches, in: Proc. 2nd Annual Asian Computing Science Conference (ASIAN), 1996, pp. 11–22.

[31] J. Bentley, D. Gibney, S. V. Thankachan, On the Complexity of BWT–Runs Minimization via Alphabet Reordering, in: Proc. 28th Annual European Symposium on Algorithms (ESA), Vol. 173, 2020, p. article 15.

[32] D. Cenzato, V. Guerrini, Z. Lipták, G. Rosone, Computing the optimal BWT of very large string collections, arXiv preprint arXiv:2212.01156 (2022).

[33] A. J. Cox, M. J. Bauer, T. Jakobi, G. Rosone, Large-scale compression of genomic sequence databases with the Burrows–Wheeler transform, Bioinformatics 28 (11) (2012) 1415–1419.

[34] G. Nong, S. Zhang, W. H. Chan, Linear suffix array construction by almost pure induced-sorting, in: Proc. 19th Data Compression Conference (DCC), 2009, pp. 193–202.

[35] G. Nong, Practical linear-time O(1)-workspace suffix sorting for constant alphabets, ACM Transactions on Information Systems 31 (3) (2013) 1–15.

[36] F. A. Louza, S. Gog, G. P. Telles, Inducing enhanced suffix arrays for string collections, Theoretical Computer Science 678 (1) (2017) 22–39.

[37] J. Kärkkäinen, D. Kempa, S. J. Puglisi, B. Zhukova, Engineering external memory induced suffix sorting, in: Proc. 19th Workshop on Algorithm Engineering and Experiments (ALENEX), 2017, pp. 98–108.

[38] D. Okanohara, K. Sadakane, Practical entropy-compressed rank/select dictionary, in: Proc. 9th Workshop on Algorithm Engineering and Experiments (ALENEX), SIAM, 2007, pp. 60–70.

[39] S. Gog, T. Beller, A. Moffat, M. Petri, From Theory to Practice: Plug and Play with Succinct Data Structures, in: Proc. 13th International Symposium on Experimental Algorithms (SEA), 2014, pp. 326–337.

[40] H.-J. Boehm, R. Atkinson, M. Plass, Ropes: An alternative to strings, Software: Practice and Experience 25 (12) (1995) 1315–1330.

[41] H. Li, Fast construction of fm-index for long sequence reads, Bioinformatics 30 (22) (2014) 3274–3275.