Efficient Construction of the BWT for Repetitive Text Using String Compression

Díaz-Domínguez, Diego

Schloss Dagstuhl- Leibniz-Zentrum fur Informatik GmbH, Dagstuhl Publishing
2022-06-01

Díaz-Domínguez , D & Navarro , G 2022 , Efficient Construction of the BWT for Repetitive Text Using String Compression . in H Bannai & J Holub (eds) , 33rd Annual Symposium on Combinatorial Pattern Matching, CPM 2022 . , 29 , Leibniz International Proceedings in Informatics, LIPIcs , vol. 223 , Schloss Dagstuhl- Leibniz-Zentrum fur Informatik GmbH, Dagstuhl Publishing , pp. 1-18 , 33rd Annual Symposium on Combinatorial Pattern Matching, CPM 2022 , Prague , Czech Republic , 27/06/2022 . https://doi.org/10.4230/LIPIcs.CPM.2022.29

http://hdl.handle.net/10138/356030
https://doi.org/10.4230/LIPIcs.CPM.2022.29

Efficient Construction of the BWT for Repetitive Text Using String Compression

Diego Díaz-Domínguez
Department of Computer Science, University of Helsinki, Finland

Gonzalo Navarro
CeBiB – Centre For Biotechnology and Bioengineering, Department of Computer Science, University of Chile, Santiago, 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 the 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 it also speeds up the required computations. Our experiments show important savings in both space and computation time when the text is repetitive. On average, we are 3.7x faster than the baseline compressed approach, while maintaining a similar memory consumption. These results make our method stand out as the only one (to our knowledge) that can build the BCR BWT of a collection of 25 human genomes (75 GB) in about 7.3 hours, and using only 27 GB of working memory.

2012 ACM Subject Classification Theory of computation → Data compression

Keywords and phrases BWT, string compression, repetitive text

Digital Object Identifier 10.4230/LIPIcs.CPM.2022.29

Supplementary Material *Software (Source Code)*: `https://github.com/ddiazdom/grlBWT`
archived at `swh:1:dir:db4691b6162da5d456f6f3005daf8c23424b0120`

Funding *Diego Díaz-Domínguez*: Academy of Finland Grant 323233
Gonzalo Navarro: ANID Basal Funds FB0001 and Fondecyt Grant 1-200038, Chile

1 Introduction

The Burrows–Wheeler transform (BWT) [6] is a reversible string transformation that reorders the symbols of a text T according the lexicographical ranks of its suffixes. The features of this transform have turned it into a key component for text compression and indexing [32, 25]. In addition to being reversible, the reordering produced by the BWT reduces the number of equal-symbol runs in T, thus improving the compresibility. On the other hand, its combinatorial properties [10] enable the creation of self-indexes [9, 28] that support pattern matching in time proportional to the pattern length. Popular bionformatic tools [18, 21] rely on the BWT to process data, as collections in this discipline are typically massive and repetitive, and the patterns to search for are short.

There are several algorithms in literature that produce the BWT in linear time [34, 1, 23, 8, 3]. Nevertheless, the computational resources their implementations require when the input is large are still too high for practical purposes. This problem is particularly evident in Genomics applications, where the amount of data is growing at an astronomical rate [35].

Although genomic collections are becoming more and more massive, the effective information they contain remains low compared to their sizes [27]. A promising solution to deal with this kind of data is then to design BWT algorithms that scale with the amount of information in the collection, not with its size.

Motivated by these ideas, some authors have developed new BWT algorithms that exploit text repetitions to reduce the computational requirements [14, 5, 13, 15, 4]. Their approach consists of extracting a set of representative strings from the text, perform calculations on them, and then extrapolate the results to the copies of those strings. For instance, the methods of Boucher et al. [5, 4] based on prefix-free parsing (PFP) use Karp–Rabin fingerprints [12] to create a dictionary of prefix-free phrases from T. Then, they 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. [14] consider a subset of positions in T that 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 of them are at a theoretical stage [14, 13, 15], while the rest [5, 4] 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. [31] 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. [34]. Briefly, Okanohara et al. cut the text into phrases using ISS, assign symbols to the phrases, and then replace the phrases with their symbols. They apply this procedure recursively until all the symbols in the text are different. Then, when they go back from the recursions, they induce an intermediate BWT^i for the text of every recursion i using the previous BWT^{i+1}. On the other hand, GCIS stores the dictionaries that ISS generates in the recursions in a context-free grammar. 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. [7] recently demonstrated that ISS-based compressors such as GCIS require much less computational resources than state-of-the-art methods like RePair [19] to encode the data, while maintaining high compression ratios. The simple construction of ISS makes it an attractive alternative to process 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) [17] has proved useful for compression [31, 7] and for constructing the BWT [34]. In this work, we show that compression can be incorporated in the internal stages of the BWT computation in a way that saves both working space and time. Okanohara et al. [34] use ISS to construct the BWT in recursive stages of parsing (which cuts the text into metasymbols) and partial construction of the BWT of the metasymbols; the final BWT is obtained when returning from the recursion. We use a technique similar to grammar compression to store the dictionaries of metasymbols, and run-length compression for the partial BWTs. This approach is shown not only to save the space required for those intermediate results, but importantly, the format we choose actually 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 [1], 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. Our experiments show that, when the input is a collection of human genomes (a repetitive dataset), our implementation requires 3.7x less computation time than `ropeBWT2` [21], an efficient implementation of the BCR BWT algorithm. Additionally, we use 7.6x less working memory than `pfp-ebwt` [4], a recent method that uses an strategy similar to ours. Under not so repetitive scenarios, our performance is competitive with `BCR_LCP_GSA` [1] or `gsufsort` [8].

2 Related Concepts

2.1 The Burrows–Wheeler Transform

Consider a string $T[1, n-1]$ over alphabet $\Sigma[2, \sigma]$, and the sentinel symbol $\Sigma[1] = \$$, which we insert at $T[n]$. The *suffix array* [26] of T is a permutation $SA[1, n]$ that enumerates the suffixes $T[i, n]$ of T in increasing lexicographic order, $T[SA[i], n] < T[SA[i+1], n]$, for $i \in [1, n-1]$.

The *Burrows–Wheeler transform* (BWT) [6] is a reversible string transformation that stores in $BWT[i]$ the symbol that precedes the i*th* suffix of T in lexicographical order, i.e., $BWT[i] = T[SA[i]-1]$ (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 $T[i]$, $\mathsf{LF}(j) = j'$ returns the index j' such that $BWT[j'] = T[i-1]$ maps the preceding symbol of $T[i]$. 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 [1] is a variant of the original BWT that encodes a string collection $\mathcal{T} = \{T_1, T_2, \ldots, T_k\}$ instead of a single string T. Briefly, if two (or more) symbols $a = T_x[k]$ and $b = T_y[k']$, from different strings $T_x, T_y \in \mathcal{T}$, are preceded by identical suffixes $T_x[k+1..] = T_y[k'+1..]$, the order of a and b in BWT is the same as the relative order of T_x and T_y in $\mathcal{T}$. The BCR BWT also appends sentinel symbols $\$$ to the strings of $\mathcal{T}$ to detect their boundaries in the BWT. A position $BWT[j] = \$$ represents the start of a string T_u, and $BWT[\mathsf{LF}(j)]$ maps the end of a string $T_{u'} \in \mathcal{T}$ that is not necessarily T_u.

2.2 Grammar and Run-length Compression

Grammar compression [16] consists of encoding a text T as a small context-free grammar $\mathcal{G}$ that only produces 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 $\mathsf{S} \to C \in \mathcal{R}$ 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 pairs. More specifically, T becomes a sequence $(a_1, l_1), (a_2, l_2), \ldots, (a_{n'}, l_{n'})$ of $n' \leq n$ pairs, where every (a_i, l_i), with $i \in [1, n']$, stores the symbol $a_i \in \Sigma$ of the i*th* run and its length $l_i \geq 1$. For instance, let $T[i, j] = aaaa$ be a substring with four consecutive copies of a, where $T[i-1] \neq a$ and $T[j+1] \neq a$. Then $T[i, j]$ compresses to $(a, 4)$.

2.3 Induced Suffix Sorting

Induced suffix sorting (ISS) [17] computes the lexicographical ranks of a subset of suffixes in T and then uses the result to induce the order of the rest. This method is the underlying procedure in several algorithms that build the suffix array [30, 29, 22] and the BWT [34, 5] in linear time. The ISS idea introduced by the suffix array algorithm SA-IS of Nong et al. [30] is of interest to this work. The authors give the following definitions:

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

▶ **Definition 2.** *A symbol $T[i]$, with $i \in [1, n]$, is called leftmost S-type, or LMS-type, if $T[i]$ is S-type and $T[i-1]$ is L-type.*

▶ **Definition 3.** *An LMS substring is (i) a substring $T[i, j]$ with both $T[i]$ and $T[j]$ being LMS symbols, and there is no other LMS symbol in the substring, for $i \neq j$; or (ii) the sentinel itself.*

SA-IS is a recursive method. In every recursion i, it initializes an empty suffix array A^i for the input text T^i (i=1). 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 A^i. More specifically, if $T^i[j] = a$ is the first symbol of an LMS substring, it inserts j in the right-most empty position in the bucket a of A^i. After scanning T^i, SA-IS sorts the LMS substrings in A^i using ISS. This procedure only requires two linear scans of A^i (we refer the reader to Nong et al. [30] for further detail).

ISS sorts the LMS substrings in a way that is slightly different from lexicographic ordering, we refer to it as $\prec_{LMS}$ ordering. In particular, if an LMS substring $T^i[a, b]$ is a prefix of another LMS substring $T^i[a', b']$, then $T^i[a, b]$ gets higher order. However, the higher rank of $T^i[a, b]$ implies that the suffix $T^i[a..]$ is lexicographically greater than the suffix $T^i[a'..]$. The cause of this property is explained in Section 2 of Ko and Aluru [17].

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, LMS substring with the same sequence are still unsorted in A^i. Nong et al. solve this problem by creating a new string T^{i+1} in which they replace the distinct LMS substrings with their orders in A^i, and use T^{i+1} as input for another recursive call $i + 1$. The base case for the recursion is when all the suffixes in A^i are prefixed by different symbols, in which case they return A^i without further processing.

When the $(i + 1)th$ recursive call ends, the suffixes of T^i prefixed by the same LMS substrings are completely sorted in A^{i+1}, so SA-IS proceeds to complete A^i. For doing so, it resets A^i, inserts the LMS substrings arranged as they respective symbols appear in A^{i+1}, and performs ISS again to reorder the unsorted suffixes of T^i. Once it finishes, it passes A^i to the previous recursion $i - 1$. The final array A^1 is the suffix array for T.

3 Methods

3.1 Definitions

We consider a collection $\mathcal{T} = \{T_1, T_2, \ldots, T_k\}$ of k strings over the alphabet $\Sigma[2, \sigma]$. The input for our algorithm is thus the sequence $T = T_1\$T_2 \ldots T_k\$$ of total length $n = |T|$ that represents the concatenation of $\mathcal{T}$. The symbol $\$$ is a sentinel that we use as a boundary between consecutive strings in T. We map $\$ = \Sigma[1]$ to the smallest symbol in the alphabet.

Let $\mathcal{D} = \{D_1, D_2, \ldots, D_y\}$ be a string set that is not suffix-free. A suffix $D_j[u..]$, with $D_j \in \mathcal{D}$, is *proper* if $1 < u \leq |D_j|$. Additionally, we consider a suffix $D_j[u..]$, with $D_j \in \mathcal{D}$, to be *left-maximal* if there is at least one other suffix $D_{j'}[u'..]$, with $D_{j'} \in \mathcal{D}$, such that (i) $j \neq j'$, (ii) $D_{j'}[u'..] = D_j[u..]$, and (iii) both $D_{j'}[u'..]$ and $D_j[u..]$ are proper suffixes with $D_{j'}[u' - 1] \neq D_j[u - 1]$ or one of them is not a proper suffix.

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 2.3), but includes elements of grammar and run-length compression (Section 2.2). These new features reduce the space usage of the temporal data that grlBWT maintains in memory, thus decreasing both working memory and computing time. We now give a brief overview of our approach.

Our method works in two phases: the *parsing* phase and the *induction* phase. The parsing phase is similar to the recursive steps of SA-IS. In every iteration i (or parsing round), we first scan the input string T^i ($T^1 = T$) to build a dictionary D^i with the phrases that occur as LMS substrings. We also record the frequency of every phrase, i.e., the number of times it occurs as an LMS substring in T^i. Subsequently, we use the phrases in D^i and their frequencies to construct a preliminary BWT for T^i ($pBWT^i$), which we complete in the induction phase. We say $pBWT^i$ is partial because it has empty spaces we can not fill just with the information in D^i. To make the completion more efficient during the induction phase, we encode $pBWT^i$ using run-length compression and D^i using a technique similar to grammar compression. Finally, we store $pBWT^i$ and D^i on disk, and create a new text T^{i+1} for the next parsing round $i+1$. We construct T^{i+1} by replacing the LMS substrings of T^i with their associated symbols in D^i. The parsing phase finishes when no new dictionary phrases can be extracted from the input text T^i (see Section 3.3).

Let h be the number of iterations the parsing phase of grlBWT incurred with T. The induction phase starts by building the BWT for T^h. After obtaining BWT^h, we start a new iterative process in which we revisit the data we dumped to disk during the parsing phase in reverse order (i.e., from round $h-1$ to round 1). In every iteration i, the BWT^{i+1} of T^{i+1} is already computed, and we use it along with compressed version of D^i to induce the order of the symbols in the empty entries of $pBWT^i$. Once we finish the induction, we compact $pBWT^i$ using run-length encoding to create BWT^i. The final BCR BWT for T is thus in BWT^1.

3.3 The Parsing Phase

In this section, we explain the steps we perform during the ith iteration of the parsing phase of grlBWT. Assume we receive as input a text T^i over the alphabet $\Sigma^i = [1, \sigma^i]$. We first initialize a hash table H^i that we will use to construct the dictionary D^i. The keys in H^i will be the phrases that occur as LMS substrings in T^i while the values of H^i will be the frequencies of the keys, i.e., the number of times the keys occur as LMS substrings in T^i.

The basic idea to fill H^i consists of scanning T^i from right to left to classify its symbols according the definitions of Section 2.3, and hash an LMS subtring every time we reach an LMS-type symbol. This mechanism is almost the same as the one described by Nong et al. [30] to detect the LMS substrings (except for the hashing). However, we add an extra consideration. The detection of LMS substrings is oblivious of the fact that T^1 encodes a string collection rather than a single string. More precisely, in any parsing round $i > 1$, we could have an LMS substring of T^i whose *expansion*[1] produces a substring of T^1 that covers two or more strings of $\mathcal{T}$. These border phrases make the computation of the BCR BWT a bit more difficult as we need to treat them differently. We avoid this problem by maintaining a bit vector $B^i[1, \sigma^i]$ that marks which symbols in T^i expand to suffixes of strings in $\mathcal{T}$. Thus, during the right-to-left scan of T^i, each time we reach a position $T^i[j]$, such that $B^i[T^i[j]] = 1$, we truncate the active LMS substring. We record the phrase $F = T^i[j+1, j']$ in H^i, where $T[j']$ is the last LMS-type symbol we accessed, and start a new phrase from position $T^i[j]$. Notice that the truncated strings are not LMS substrings by definition, but

[1] Let $T^i[j, j']$ be a substring of T^i. We define the *expansion* of $T^i[j, j']$ as the string in Σ^1 we obtain by recursively replacing every symbol $T^i[j] \in \Sigma^i$, for $j \in [j, j']$, with its corresponding phrases in the dictionaries $D^{i-1}, D^{i-2}, \ldots, D^1$.

they do not affect our algorithm (the reasons are explained in Definition 4 and Lemma 3 of Díaz-Domínguez et al. [7]). We receive B^i as input along with T^i at the beginning of the parsing round i, and we compute the next $B^{i+1}[1, \sigma^{i+1}]$ when we finish the round.

For practical reasons, we change the representation of D^i, encoded in H^i for the moment, to a more convenient data structure. First, we concatenate all the keys of D^i in one single vector R^i. We mark the boundaries of consecutive phrases in R^i with a bit vector L^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 also augment L^i with a data structure that supports rank_1 queries [33] to map each symbol $D^i[j]$ to its corresponding phrase. We store the values of D^i in another vector $N^i[1, |D^i|]$. We maintain the relative order so that the value $N^i[o]$ maps the *oth* phrase we inserted into R^i. For simplicity, we will refer to the representation (R^i, L^i, N^i) just like D^i. We still need H^i to construct the parse T^{i+1}, so we do not discard it but store it into disk.

The next step is to build $pBWT^i$ from D^i. For that purpose, we use the following observations:

▶ **Lemma 4.** *Let $X[1, x]$ and $Y[1, y]$ be two different strings over the alphabet Σ^i, with lengths $x > 1$ and $y > 1$ (respectively). Assume both occur as suffixes in one or more phrases of D^i. Let $\mathcal{X}$ be the list of positions in T^i where X occurs as a suffix of an LMS substring. More specifically, each $j \in \mathcal{X}$ is a position such that $T^i[j, j + x - 1]$ is an occurrence of X and $T^i[j - j', j + x - 1]$, with $j' \geq 0$, is an LMS substring. Let us define a list $\mathcal{Y}$ equivalent to $\mathcal{X}$, but for Y. If $X \prec_{LMS} Y$ (see Section 2.3), then all the suffixes of T^i starting at positions in $\mathcal{X}$ are lexicographically greater than the suffixes starting at positions in $\mathcal{Y}$.*

Proof. Assume first that X is not a prefix of Y (and vice versa). We compare the sequences of these strings from left to right until we find a mismatching position u (i.e., $X[u] \neq Y[u]$). We know that symbols $X[u]$ and $Y[u]$ define the lexicographical order of the suffixes in $\mathcal{X}$ relative to the suffixes in $\mathcal{Y}$. 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 $X[u] \neq Y[u]$. For this case, we resort to the symbol types of Section 2.3. We assume for this proof that X is a prefix of Y, but the other way is equivalent. We know that $X[x]$ and $Y[x]$ have different types. $X[x]$ is LMS type because X is a suffix of an LMS substring. On the other hand, $Y[x]$ is L type because if it were S type, then it would also be LMS type, and thus $Y[1, x]$ would be an occurrence for X. This observation is due to $Y[x - 1] = X[x - 1]$ is L type. Given the types of $X[x]$ and $Y[x]$, the occurrences of X in $\mathcal{X}$ are always followed in T^i by symbols that are greater than $Y[x + 1]$, meaning that the suffixes of T^i starting at positions in $\mathcal{X}$ are lexicographically greater than the suffixes starting at positions in $\mathcal{Y}$. This observation does not hold when X or Y have length one: $X[x]$ equals $Y[1]$ and both are LMS type, so there is no enough information to decide the lexicographical order of the suffixes in $\mathcal{X}$ and $\mathcal{Y}$. ◀

The consequence of Lemma 4 is that the suffixes of length > 1 in D^i induce a partition over SA^i (the suffix array of T^i):

▶ **Lemma 5.** *Let $\mathcal{S} = \{S_1, S_2, \ldots, S_k\}$ be the set of strings of length > 1 that occur as suffixes in the phrases of D^i. Additionally, let $\mathcal{O} = \{O_1, O_2, \ldots, O_k\}$ be the set of occurrences in T^i for the strings in $\mathcal{S}$. For every $S_u \in \mathcal{S}$, its associated list $O_u \in \mathcal{O}$ stores each position j such that $T^i[j, j + |S_j| - 1]$ is an occurrence of S_u and $T^i[j - j', j + |S_j| - 1]$, with $j' \geq 0$, is an LMS substring. It holds that $\mathcal{O}$ induces a partition over the suffix array of T^i (SA^i) as the lexicographical sorting places the elements of each $O_u \in \mathcal{O}$ in a consecutive range of SA^i.*

Proof. We demonstrate the lemma by showing that the lexicographical sorting does not interleave suffixes of T^i in SA^i that belong to different lists of $\mathcal{O}$. Assume a string $S_u \in \mathcal{S}$, associated with the list $O_u \in \mathcal{O}$, is a prefix in another string $S_{u'} \in \mathcal{S}$, which in turn is associated with the list $O_{u'} \in \mathcal{O}$. Even though we do not know the symbols that occur to the right of S_u in its occurrences of O_u, we do know that both S_u and $S_{u'}$ are suffixes of LMS substrings, and by Lemma 4, we know that all the suffixes of T^i in O_u are lexicographically greater than the suffixes in $O_{u'}$. Hence, the interleaving of suffixes in SA^i from different lists of $\mathcal{O}$ is not possible, even if $\mathcal{S}$ is not a prefix-free set. ◀

Lemma 5 gives us a simple way to construct the preliminary BWT for T^i ($pBWT^i$). We consider for the moment $pBWT^i$ to be a vector of lists to simplify the explanations. We first sort the strings of $\mathcal{S}$ in $\prec_{LMS}$ order. Then, for every *oth* string $S \in \mathcal{S}$ in $\prec_{LMS}$ order, we insert in the list $pBWT^i[o]$ the symbols that occur to the left of S in D^i. There are three cases to consider for this task:

▶ **Lemma 6.** *Let $S \in \mathcal{S}$ be the string with $\prec_{LMS}$ order o among the other strings in $\mathcal{S}$. If S is left-maximal in D^i, then the list $pBWT^i[o]$ contains more than one distinct symbol, and it is not possible to decide the relative order of those symbols with the information of D^i.*

Proof. Let X and Y be two phrases of D^i where S occurs as a suffix. Assume the left symbol of S in X is $x \in \Sigma^i$ and the left symbol in Y is $y \in \Sigma^i$. In this scenario, the relative order of x and y is not decided by S, but for the sequences that occur to the right of X and Y in T^i. However, those sequences are not accessible directly from D^i. Hence, it is not possible to decide the order of x and y in $pBWT^i[o]$. ◀

▶ **Lemma 7.** *Consider the string $S \in \mathcal{S}$ of Lemma 6. When S occurs as a non-proper suffix in a phrase $F \in D^i$, it is not possible to complete the sequence of symbols for $pBWT^i[o]$.*

Proof. The symbols that occur to the left of S in T^i are stored in the LMS substrings that precede F in T^i. However, it is not possible to know from D^i which are those substrings. ◀

We now describe the information of $pBWT^i$ that we can extract from D^i:

▶ **Lemma 8.** *Let $S \in \mathcal{S}$ be the string of Lemma 6. Additionally, let $O \in \mathcal{O}$ be the list of occurrences of S in T^i as described in Lemma 5. If all the suffixes of T^i in O are preceded by the same symbol $s \in \Sigma^i$ (i.e., S is not left-maximal), then $pBWT^i[o] = (s, l)$ is an equal-symbol run of length $l = |O|$, where o is the $\prec_{LMS}$ order of S in $\mathcal{S}$.*

Proof. By Lemma 5, we know that the suffixes of T^i in O are prefixed by S, and that they form a consecutive range $SA^i[j, j']$. Additionally, the symbols that occur to the left of the suffixes in $SA^i[j, j']$ are those for the list of $pBWT^i[o]$. However, we still have not resolved the relative order of the suffixes in $SA^i[j, j']$, so (in theory) we do not know how rearrange the symbols in $pBWT^i[o]$. The suffixes of T^i in O are preceded by the same symbol s, so it is no necessary to further sort $SA^i[j, j']$ because the outcome for $pBWT^i[o]$ will be always an equal-symbol run for s of length $l = |O|$. ◀

The problem is that we do not store O, so we do not know value for l in (s, l). Nevertheless, we do have the frequencies of the phrases in D^i, in the vector N^i. In this way, we can compute l by summing the frequencies in N^i for the phrases of D^i where S occurs as a suffix.

Now that we have covered all the theoretical aspects of the parsing phrase, we proceed to describe our procedure to build $pBWT^i$.

3.3.1 Constructing the Preliminary BWT for the Parsing Round

The computation of $pBWT^i$ starts with the construction of a *generalized* suffix array SA_{D^i} for D^i. We say SA_{D^i} is generalized because it only considers the suffixes of the dictionary phrases. If a string $S \in \mathcal{S}$ appears as a suffix in two or more phrases, those occurrences maintain in SA_{D^i} the relative order in which their enclosing phrases appear in D^i. In practice, the values we store in SA_{D^i} are the positions in R^i, the vector storing the concatenated phrases of D^i (see the encoding of D^i in Subsection 3.3).

We compute SA_{D^i} using a modified version of the ISS method mentioned in Subsection 2.3. The first difference is that, in step one, we insert in SA_{D^i} the position in R^i of the last symbol of each phrase. Put it another way, suppose $R^i[j]$, with $L^i[j+1] = 1$, is the last symbol of a phrase F, then we insert j in the right-most available cell in the bucket $R^i[j]$ of SA_{D^i}. The step one in the original ISS puts LMS-type symbols at the end of the buckets. In our case, the last symbol of a phrase is, by definition, LMS type in T^i, so the operation is homologous. The second difference of our ISS variation is that, during step two and three, we skip each position $SA_{D^i}[u]$ representing the start of a phrase ($L^i[SA_{D^i}[u]] = 1$) as they do not induce suffixes.

The next step is to scan SA_{D^i} from left to right to compute $pBWT^i$. From now on, we consider $pBWT^i$ to be a run-length compressed vector instead of a vector of lists. As we move throughout the suffix array, we search for every range $SA_{D^i}[j, j']$, with $j' - j + 1 \geq 1$, that encode suffixes with the same sequence[2]. Nevertheless, we consider only the ranges that either represent suffixes of length > 1 or suffixes of length 1 that expand to suffixes of $\mathcal{T}$. Recall that the left-most symbol of an LMS substring is the same as the right-most symbol of the LMS substring that precedes it. Hence, considering all the suffixes in D^i will produce a redundant (and incorrect) BWT. The only exception to this rule are the LMS substrings at the beginning of the strings of $\mathcal{T}$ as they do not share a symbol with the LMS substring to their left. This kind of substrings only appear when we cross from T_{u+1} to T_u in T^i, with $T_u, T_{u+1} \in \mathcal{T}$. We can detect this situation using B^i, the bit vector marking the symbols in Σ^i that expand to suffixes of strings in $\mathcal{T}$ (see Section 3.3).

We define the length of $SA_{D^i}[j, j']$ as $l = \sum_{u=j}^{j'} N^i[\mathsf{rank}(L^i, SA_{D^i}[u])]$. This value is the sum of the frequencies of the phrases where the suffixes in $SA_{D^i}[j, j']$ occur.

If all the suffixes of $SA_{D^i}[j, j']$ are followed by the same symbol $s \in \Sigma^i$, we append (s, l) to $pBWT^i$ (see Lemma 8). Otherwise we append $(*, l)$. The symbol $*$ represents an empty entry and it is out of Σ^i. We will resolve $(*, l)$ in the next phase of grlBWT (see Lemmas 6 and 7). After scanning SA_{D^i}, we store $pBWT^i$ into disk and discard SA_{D^i}.

3.3.2 Grammar Compression and Next Parsing Round

Once we finish constructing $pBWT^i$, the next step in the parsing round i is to store D^i in a compact form to use it later during the induction phase of grlBWT. We first explain why we need D^i 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} from left to right, mapping every symbol $BWT^{i+1}[j] \in \Sigma^{i+1}$ back to the phrase $F \in D^i$ from which it originated, and then checking which of the proper suffixes of F produced empty entries in $pBWT^i$ (see Lemmas 6 and 7). Assume the suffix $F[u..] = S \in \mathcal{S}$ produced an empty entry, then we append $F[u-1]$ in the BWT range associated with S (see Lemma 5).

[2] In practice, we compute every distinct range $SA_{D^i}[j, j']$ during the construction of the suffix array. We reserve the least significant bit in the cells of SA_{D^i} to mark every position $SA_{D^i}[j]$. We flag these positions during the execution of our modified version of ISS (Section 2.3).

The process described above requires D^i and a mechanism to map the left-maximal suffixes in D^i back to the empty entries they produce in $pBWT^i$. We solve the problem by encoding D^i with a representation that is similar to grammar compression (Section 2.2).

We start by discarding N^i and the rank_1 data structure, as they are no longer necessary (see the current encoding of D^i in Section 3.3). For our method to work, we also need each string $S \in \mathcal{S}$ associated with an empty entry $(*, l)$ of $pBWT^i$ to be a member of D^i. This property might not hold when the string S that produced an empty entry meets Lemma 6. The problem arises if S always appears as a proper suffix in D^i, not as a full phrase. If that is the case, we *create*[3] a new independent entry for S in D^i.

After expanding D^i, we create a hash table M^i in which we insert each phrase $F \in D^i$ occurring as a left-maximal suffix. If F has $\prec_{LMS}$ rank b in D^i, then we insert the pair (F, b) into M^i, where F is the key and b is the value. Once we construct M^i, we reorder the way in which the phrases of D^i are concatenated in R^i according to their $\prec_{LMS}$ ranks.

The next step consists of *compressing* D^i. We scan R^i from left to right, and for every $F = R^i[j, j'] \in D^i$, with $L^i[j] = 1$ and $L^i[j' + 1] = 1$, we search for the longest proper suffix $F[u..]$ that exists in M^i as a key. If such key exists, then we replace F with $R[j, j + 1] = F[u - 1]{\cdot}b'$, where b' is the value associated with $F[u..]$ in M^i. If no proper suffix of F exists in M^i, then we replace F as $R[j, j + 1] = *{\cdot}F[|F| - 1]$, where $*$ is a dummy symbol. After updating the sequence of F, we mark the symbols in $R[j + 2, j']$ as discarded if $j' - j + 1 = |F| \geq 2$. When we finish the scan of the dictionary, we left-contract R^i by removing the discarded symbols. This process reduces the phrases in D^i to strings of length two, so the vector L^i is no longer necessary.

Now we explain the rationale of our encoding. We develop our argument as a chain of implications. Consider again the phrase F, which we replaced with the sequence $F[u - 1]{\cdot}b'$. We obtained $b' \in \Sigma^{i+1}$ when we performed a lookup operation of $S = F[u..]$ in M^i during the compression of D^i. The value b' that the lookup returned is the $\prec_{LMS}$ order of S in D^i. The membership of S to the keys of M^i implies that S appears as a left-maximal suffix in D^i, which in turn implies that S is a full phrase in D^i too (we enforced this property when we expanded the dictionary). Additionally, the left-maximal condition of S implies that there were at least two suffix occurrences of S preceded by different symbols. This is why S produces an empty entry in $pBWT^i$. Now, recall that we sorted the phrases of D^i in $\prec_{LMS}$ order in R^i. Therefore, if we want to access S, we have to go to the substring $R^i[2b' - 1, 2b']$. This substring does not encode the full sequence of S, but its longest left-maximal suffix $R^i[2b] \in \Sigma^{i+1}$ (which is also a left-maximal suffix of F) along with the left-context symbol for that suffix ($R^i[2b - 1] \in \Sigma^i$). Recursively, the longest left-maximal symbol of S is not a sequence either, but a pointer to another position of R^i. We access this nested left-maximal suffix by setting $b' = R^i[2b']$ and updating the values $R^i[2b' - 1, 2b']$. We continue applying this idea until we reach a range $R^i[2b' - 1, 2b']$ where $R^i[2b'] \in \Sigma^i$, which implies that we reached the last suffix of F. This last range will store the sequence $*{\cdot}F[|F| - 1]$. Notice that the right symbol in this case is not the last symbol of F but its left context $F[|F| - 1]$. This is because the LMS substrings overlap by one character in T^i, so $F[|F|]$ is redundant as it also appears as a prefix in another phrase. However, we need $F[|F| - 1]$, as we will append it to one of the empty entries of $pBWT^i$ in the next phase of grlBWT. The only exception to this rule is when F expands to a suffix of a string in $\mathcal{T}$. In that case, we store $F[|F|]$ instead of $F[|F| - 1]$ in $R[2b' - 1, 2b']$ as $F[|F|] = R^i[2b']$ is not a prefix in any other phrase. On the

[3] It means we append the sequence of F at the end of R^i and expand L^i accordingly.

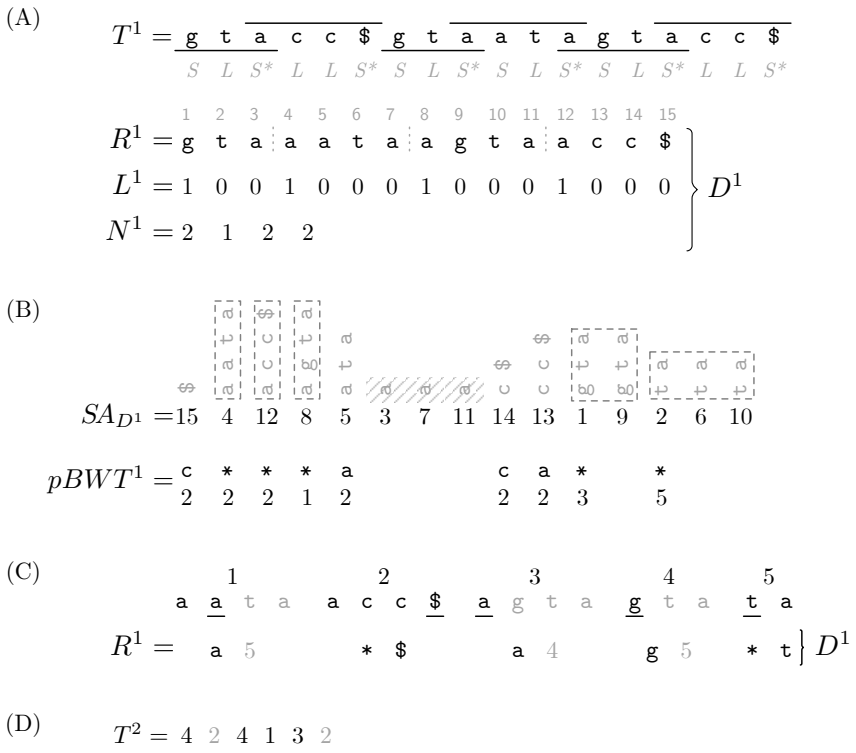

Figure 1 Parsing round $i = 1$ of grlBWT for the collection $\mathcal{T} = \{\texttt{gtacc}, \texttt{gtaatagtacc}\}$, with $T = \texttt{gtacc\$gtaatagtacc\$}$. (A) Construction of the dictionary $D^1 = \{R^1, L^1, N^1\}$ from $T^1 = T$. The sequence in gray below T^1 stores the symbol types: L-type is L, S-type is S and LMS-type is S^*. The dashed vertical lines in R^1 mark the boundaries between dictionary phrases. (B) Constructing $pBWT^1$ from the suffix array SA_{D^i} of D^i. The ranges of SA_{D^1} enclosed by dashed boxes produce empty entries in $pBWT^i$. Notice we do not use the entries $SA_{D^i}[5, 7] = 3, 7, 11$ for computing $pBWT^1$ as their corresponding symbols are redundant. For instance, consider $R^1[3] = \texttt{a}$, which is a suffix in $R^1[1, 3] = \texttt{gta}$. The last symbol in the occurrences of gta in T^1 overlaps the first symbol of acc$ or aata. Therefore, the symbol $R^1[3] = \texttt{a}$ of gta is covered by acc$ or aata in $pBWT^1$. We represent $pBWT^1$ as a sequence of equal-symbol runs. The upper row depicts the run symbols while the lower row show the run lengths. (C) Compressing D^1. The strings at the top are the dictionary phrases sorted in $\prec_{LMS}$ order. We add the string ta to the dictionary as it appears as a left-maximal suffix in D^i, and hence, produces an empty entry in $pBWT^1$. The sequences in gray are the longest left-maximal suffixes of the phrases. The underlined symbols are the left contexts of those suffixes. Notice we compress $F = \texttt{acc\$}$ directly to *$ as F does not have a left-maximal suffix. Besides, as F does not overlap other phrases in T^1 (because of $), we store $F[|F|] = \$$ instead of $F[|F| - 1] = \texttt{c}$. A different situation occurs with $S = \texttt{ta}$. This phrase does not have left-maximal suffixes of length > 1. However, in this case, we store $S[|S| - 1] = \texttt{t}$ instead of $S[|S|] = \texttt{a}$ as S overlaps other phrases in T^1. (D) The parse T^2 we obtain by replacing the phrases of T^1 with their $\prec_{LMS}$ orders in D^i. The gray symbols expand to suffixes of strings in $\mathcal{T}$.

other hand, we need the dummy symbol * to maintain the invariant that all the phrases encoded in R^i have length two. Once we finish the compression, we store R^i on disk. From now on, we use D^i to refer to R^i.

The final step for the parsing round i is to create the new text T^{i+1}. We first reload from disk the hash table H^i we produced at the beginning of the iteration, and replace its values with the keys' $\prec_{LMS}$ orders. More specifically, if a key $F \in D^i$ of H^i has $\prec_{LMS}$ order

b among the other strings that produced empty entries in $pBWT^i$, then we update the value of F in H^i to b. Notice that the strings we stored as keys in H^i are not the same as those we have now in D^i because we compress them. Therefore, we can not lookup the phrases of D^i in the keys of H^i to update the hash table values. Still, we can overcome this problem if we modify H^i after sorting D^i in $\prec_{LMS}$ order but before we compress it.

Once we update H^i, we construct T^{i+1} by scanning T^i again and replacing the LMS substrings with their associated values in H^i. If T^{i+1} has length k (the number of strings in $\mathcal{T}$), then we stop the parsing phase as all the strings in $\mathcal{T}$ are now compressed to one symbol. An example of the parsing step is depicted in Figure 1.

3.4 The Induction Phase

The induction phase starts with the computation of BWT^h, the BCR BWT for the text T^h of the last parsing round h. 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. BCR BWT maintains the relative order of the strings in $\mathcal{T}$ (see Section 2.1), so BWT^h is T^h itself.

We now describe the steps we perform during every induction step $i < h$. In this case, assume we receive as input (i) BWT^{i+1} from the previous phase, (ii) D^i, and (iii) $pBWT^i$. Before explaining our procedure, we describe some important properties of BWT^{i+1}.

▶ **Lemma 9.** *Let $BWT^{i+1}[j]$ and $BWT^{i+1}[j']$ be two symbols at different positions j and j', with $j < j'$, whose mapping phrases in D^i are F and F', respectively. Also, let the proper suffixes $F[u..] = F'[u'..] = S \in \mathcal{S}$ (see Lemma 5 for the description of $\mathcal{S}$) be two occurrences in T^i of a string S that appears as a left-maximal suffix in D^i. The suffix of T^i prefixed by $F[u..]$ precedes in SA^i (the suffix array of T^i) the suffix of T^i prefixed by $F[u'..]$.*

Proof. As $F[u..]$ and $F'[u'..]$ are equal, their relative orders are decided by the right contexts in T^i of the occurrences $BWT^{i+1}[j]$ and $BWT^{i+1}[j']$ of F and F' (respectively). By induction, we know that BWT^{i+1} is complete, and as $BWT^{i+1}[j]$ precedes $BWT^{i+1}[j']$ in the BWT, the right context of $F[u..]$ has a smaller order in SA^i than the right context of $F'[u'..]$. ◀

If we generalize Lemma 9 to $x \geq 1$ occurrences of S, then we can use the following lemma to compute the sequence for the empty entry of $pBWT^i$ generated by S:

▶ **Lemma 10.** *Let S be a string of $\mathcal{S}$. Additionally, let $J = \{j_1, j_2, \ldots, j_x\}$ be a list of strictly increasing positions of BWT^{i+1}. Every $BWT^{i+1}[j_o]$, with $j_o \in J$, is a symbol $b \in \Sigma^{i+1}$ generated from a phrase $F \in D^i$ where $S = F[u..]$ occurs as a proper suffix. The symbols of B^{i+1} referenced by J are not necessarily equal, and hence, their associated phrases in D^i are not necessarily the same. However, these phrases of D^i are all suffixed by S. Assume we scan J from left to right, and for every j_o, we extract the symbol $F[u-1] \in \Sigma^i$ that precedes S and append it to a list L_S. The resulting list $L_S \in \Sigma^{i*}$ has the same sequence of symbols as the BWT^i range that maps the block for S in the partition of $\mathcal{S}$.*

Proof. Because of Lemma 9, the suffix of T^i prefixed by the occurrence $BWT^{i+1}[j_o]$ of S precedes the suffix of T^i prefixed by the occurrence $BWT^{i+1}[j_{o+1}]$. This property holds for every j_o, with $o \in [1, x-1]$. Hence, the suffixes of T^i prefixed by S are already sorted J. ◀

Our compressed representation for D^i (see Section 3.3.2) has precisely the information we need to construct L_S as described in the procedure of Lemma 10. Still, the idea only works when S *always* appears as a proper suffix in the phrases of D^i. When S matches a full phrase

(see Lemma 7), there is no left-context symbol for S we can extract from D^i. Nevertheless, there is only one phrase $F \in D^i$ where S can be a non-proper suffix, and because S comes from BWT^{i+1}, F has to be an LMS substring in T^i. This observation implies that F maps to a symbol $b \in \Sigma^{i+1}$ in T^{i+1}. Hence, we can extract the left-context symbols of S from the range in BWT^{i+1} that corresponds to the bth bucket of SA^{i+1}. We explain how to carry out this process in the next subsection.

3.5 The Induction Algorithm

Let p be the sum of the lengths in the empty entries of $pBWT^i$. These lengths correspond to the second field in the run-length representation of $pBWT^i$. We start the induction by creating a vector $P^i[1, p]$, which we logically divide into σ^{i+1} buckets (recall that σ^{i+1} matches the number of empty entries in $pBWT^i$). Every bucket b of P^i will be of size l_b, the length of the bth empty entry (from left to right) of $pBWT^i$. Subsequently, we perform a scan over BWT^{i+1} from left to right. For each symbol $BWT^{i+1}[j] = b \in \sigma^{i+1}$, we first check if its associated LMS substring $F \in D^i$ (the string from which we obtain the symbol b during the parsing round i) exists as a suffix in other phrases of D^i. This information is already encoded in a bit vector $V^i[1, \sigma^{i+1}]$ we constructed during the parsing round i. When F occurs as an LMS substring and as a proper suffix in other dictionary phrases ($V^i[b] = 1$), we append a dummy symbol in the bucket b of P. This is the situation we described at the end of the previous subsection. After processing b, we decompress the left-maximal suffixes of its phrase F from the compressed representation of D^i.

The decompression of F begins by accessing the range $D^i[2b-1, 2b]$ (see Section 3.3.2). If $o = D^i[2b]$ belongs to Σ^{i+1}, then the symbol o encodes a string $F[u..] = S$, with $u > 1$, whose sequence is a left-maximal suffix in D^i. During the parsing phase of grlBWT, we inserted S to D^i as an independent string as it yields an empty entry for $pBWT^i$ (see Lemma 6). The order of S in D^i is precisely o, its $\prec_{LMS}$ rank among the other phrases of D^i. On the other hand, the left-context symbol of S is $D^i[2b-1] \in \Sigma^i$. With this information, we apply Lemma 10 by appending the symbol $D^i[2b-1]$ to the bucket o of P. Then, we move to the next left-maximal suffix of F by setting $b = o$ and updating the range $D^i[2b-1, 2b]$.

The decompression of F stops when $D^i[2s]$ belongs to Σ^i, which means we reach the last symbol of F. For the moment, we do not know for which phrase of D^i $D^i[2s]$ is its left context. Hence, we set $BWT^{i+1}[j] = D^i[2s]$ and leave this position on hold to process it later. After finishing the scan of BWT^{i+1}, its symbols are now over the alphabet Σ^i. These values are the ones we have to insert in the dummy positions of P^i. Notice that the entries in P^i and BWT^{i+1} are already sorted by their right contexts in T^i. Hence, the completion of the dummy positions reduces to a merge of two sorted lists.

The last step in the induction round i consists of merging $pBWT^i$, BWT^{i+1} and P^i in BWT^i. We scan $pBWT^i$ and we append its entries to BWT^i as long as they are not empty. Then, when we reach an empty entry $(*, l)$, we proceed as follows: assume the current pair $(*, l)$ is the bth empty entry of $pBWT^i$. Then, we check if the phrase $F \in D^i$ that produced this entry (the one with $\prec_{LMS}$ order b) only occurs as a full LMS substring in T^i ($V^i[b] = 0$). If that is the case, we append the next l symbols of BWT^{i+1} into BWT^i. On the other hand, when F occurs as an LMS substring, but also as a proper suffix in other phrases of D^i ($V^i[b] = 1$), the next l symbols of BWT^i are a mix of entries from the bucket b of P^i and BWT^{i+1}. We append symbols from the bucket b of P^i as long as they are not dummy. When we reach a dummy symbol in P^i, we change the list, and append the next x symbols of BWT^{i+1} into BWT^i, where x is the number of consecutive dummy symbols we saw in P^i. Once we process the x entries of BWT^{i+1}, we go back to P and continue back and forth between P and BWT^i until we process all the symbols in the bucket b of P.

The last case we have to cover for the merge is when F always occurs as a proper suffix in D^i (i.e., it is not an LMS substring of T^i). This situation is simple as we marked F in V^i ($V^i[b] = 1$). Hence, we just copy the content of the bucket b of P into BWT^i. Notice this bucket will not have dummy entries as b does not appear in B^{i+1} as a symbol. We obtain occurrences of b while decompressing other phrases of D^i whose $\prec_{LMS}$ ranks do appear as symbols in B^{i+1}, and where F is a proper left-maximal suffix. Once we complete all the induction rounds, the final BCR BWT is in BWT^i.

3.5.1 Speeding up the Induction with Compression

If we run-length encode BWT^{i+1}, the induction becomes more efficient. Every position $BWT^{i+1}[j]$ is not a symbol b, but a pair (l, b) that represents l consecutive copies of b. Thus, instead of decompressing l times the left-maximal suffixes of the phrase associated with b, we decompress them only once, and copy the result l times to the different buckets of P^i.

Maintaining P^i as a run-length encoded sequence also improves efficiency. A compact representation of P^i reduces the working memory, which in turn reduces the number of cache misses. The only problem with this idea is that we do not know before the induction how many equal-symbols runs P^i will have. There are two solutions to this problem. First, we could represent P^i as a dynamic vector [2]. Its initial size would be σ^{i+1}, and then we expand the buckets as we insert new equal-symbol runs into them. The second option is to perform a preliminary scan of BWT^{i+1} to compute the size of the run-length compressed version of P^i, then we scan BWT^{i+1} again to perform the induction.

3.6 Complexity of our Method

We show that the construction of the BWT remains linear, even though we perform compression during the intermediate steps.

▶ **Theorem 11.** *Let $\mathcal{T} = \{T_1, T_2, \ldots, T_k\}$ be a collection with k strings and n symbols. The algorithm* grlBWT *constructs the BCR BWT of $\mathcal{T}$ in expected $O(n)$ time and requires $O(n)$ bits of working space.*

Proof. The complexities in the theorem were already proved for the linear-time algorithms that construct the suffix array [30] and the BWT [34] using ISS. We show that these complexities are not altered by our compression scheme. Let $n^i = |T^i|$ be the length of the input text we receive at parsing round i. Hashing the dictionary phrases from T^i runs in $O(n^i)$ expected time and requires $O(n^i)$ bits of space. The construction of SA_{D^i} runs in $O(n^i)$ time and space as we use ISS to build it, and the number of symbols in D^i is never greater than n^i. The extra steps of the parsing round only require a constant number of linear scans over SA_{D^i}. During the induction phase, we only perform linear scans over BWT^i and $pBWT^i$. We still have the cost of accessing the left-maximal suffixes of D^i when we scan BWT^{i+1} during the induction phase. However, our simple compressed representation for D^i (Section 3.3.2) 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} is no more than n^i. In every parsing round, the size of T^{i+1} is at most half the size of T^i, so the sum of the text lengths $n^1, n^2, \ldots, n^h$ is $O(n)$ (see Nong et al. [30]). This property also implies that grlBWT never visits more than $O(n)$ left-maximal suffixes during its induction phase. ◀

Table 1 Datasets. The upper rows are the Illumina reads while the lower rows are the human genomes. Columns four and five are the minimum and average string length (respectively) in the collection. The value for r is the number of equal-symbol runs in the BCR BWT of the collection.

Dataset	σ	Number of strings	Max. length	Avg. length	n	n/r
ILL1	5	84,006,956	151	151	12,769,057,312	3.18
ILL2	5	160,285,798	151	151	24,363,441,296	4.07
ILL3	5	235,805,550	151	151	35,842,443,600	4.67
ILL4	5	305,931,740	151	151	46,501,624,480	5.03
ILL5	5	377,453,488	151	151	57,372,930,176	5.33
HGA05	16	334,065	248,956,422	42,715	14,269,998,434	4.82
HGA10	16	759,341	250,522,664	39,025	29,634,170,092	8.76
HGA15	16	835,485	250,522,664	53,918	45,048,695,199	12.02
HGA20	16	874,235	250,522,664	68,650	60,017,146,889	15.67
HGA25	16	899,424	250,522,664	83,447	75,055,723,570	19.42

4 Experiments

We implemented grlBWT as a C++ tool, also called `grlBWT`. This software uses the `SDSL-lite` library [11] to operate with bit vectors and rank data structures. Our source code is available at `https://github.com/ddiazdom/grlBWT`. We compared the performance of `grlBWT` against other tools that compute BWTs for string collections:

- `ropebwt2`[4] : a variation of the original BCR algorithm of Bauer et al. [1] that uses rope data structures [2]. This method is described in Heng Lee [20].
- `pfp-eBWT`[5] : the eBWT algorithm of Boucher et al. [4] that builds on PFP and ISS.
- `BCR_LCP_GSA`[6] : the current implementation of the semi-external BCR algorithm [1].
- `egap`[7]: a semi-external algorithm of Edigi et al. [8] that builds the BCR BWT.
- `gsufsort`[8]: an in-memory method proposed by Louza et al. [23] that computes the BCR BWT and (optionally) other data structures.

We also considered the tool `bwt-lcp-em` [3] 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' description. For `grlBWT`, we used the compiler flags `-O3 -msse4.2 -funroll-loops`.

We considered two common types of genomic data for the experiments: short reads and assembled genomes. We downloaded five collections of Illumina reads produced from different human genomes [9]. We concatenated the strings so that our dataset 1 had one read collection, dataset 2 had two collections, and so on. We named the files `ILLX`, where `X` is the number of read collections concatenated. We also downloaded from NCBI[10] 25 collections of fully-assembled human genomes. Like with the reads, we created the inputs

[4] `https://github.com/lh3/ropebwt2`
[5] `https://github.com/davidecenzato/PFP-eBWT`
[6] `https://github.com/giovannarosone/BCR_LCP_GSA`
[7] `https://github.com/felipelouza/egap`
[8] `https://github.com/felipelouza/gsufsort`
[9] `https://www.internationalgenome.org/data-portal/data-collection/hgdp`
[10] `https://www.ncbi.nlm.nih.gov/assembly`

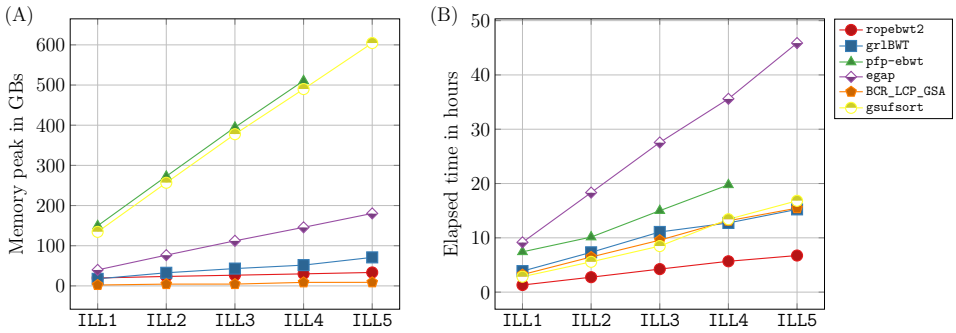

Figure 2 Memory peak usage (GBs) and elapsed time (in hours) for the Illumina reads.

for our experiments so that every dataset has five more genomes than the previous one. This setup aims to increase the repetitiveness as the collection size increases. We named each file using the prefix `HGA` concatenated with the number of genomes it had. The only prepossessing step we performed on the genomes was to put every chromosome in one line and set all the characters to upper case. All our inputs are described in Table 1.

We also investigated the effect of page cache [24, Ch. 16] in `grlBWT`. In every parsing round i, we keep T^i on disk, and linearly scan its file by loading from disk to RAM one data chunk of 8 MB at the time. Similarly, we keep a buffer of 8 MB in RAM for T^{i+1}, which we dump into disk every time it gets full. We manipulate BWT^i (respectively, BWT^{i+1}) in the same way. We used the function `posix_fadvise` to turn off the page cache for T^i, T^{i+1}, BWT^i, and BWT^{i+1}. Then we assessed the performance of `grlBWT` on the assembled genomes using `posix_advice` and not using it. We did not evaluate the effect of the page cache in the other tools.

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 in the BWT. We ran the experiments on the Illumina reads using one thread in all programs as not all support multi-threading. For this purpose, we set the extra flag `-P` to `ropebwt2` to indicate single-thread execution. We tested the human genomes only on `ropebwt2`, `grlBWT` and `pfp-ebwt`. By default, `ropebwt2` uses four working threads, so we set the same number of threads for `grlBWT` and `pfp-ebwt`. We did not report results for `pfp-ebwt` with dataset `ILL25` as the execution crashed. 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.

5 Results and Discussion

We summarize our experiments in Figures 2 and 3. The results we report for `grlBWT` do not consider the use of `posix_advice` to turn off the page cache. In Illumina reads, the fastest method was `ropeBWT2`, with a mean elapsed time of 4.14 hours. It is then followed by `BCR_LCP_GSA`, `gsufsort`, `grlBWT`, `pfp-bwt`, and `egap`, with mean elapsed times of 9.58, 9.43, 10.05, 13.08, and 27.30 hours, respectively (Figure 2B). We notice that `grlBWT` is competitive with `BCR_LCP_GSA` and `gsufsort`. However, it gets slightly faster than them from input `ILL4` onward. We expected this behaviour since the largest datasets are more repetitive.

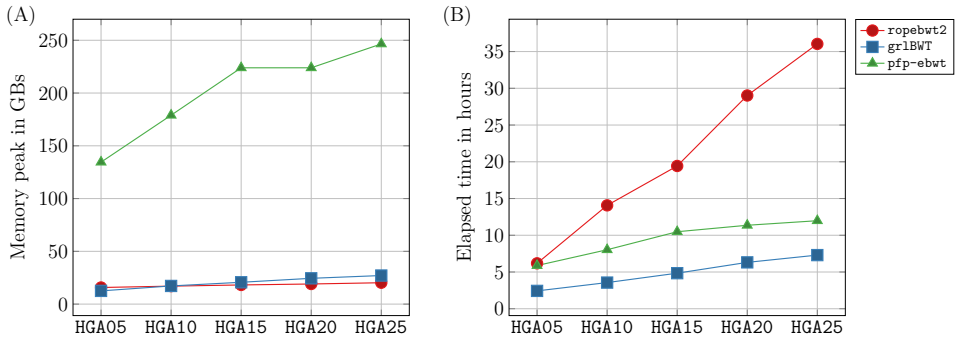

Figure 3 Memory peak usage (GBs) and elapsed time (in hours) for the assembled genomes.

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 `ropebwt2`, with an average memory peak of 26.64 GB. In both cases, the memory consumption increases slowly with the input size. In the case of `grlBWT`, the memory peak is more considerable; 42.20 GB on average, with a memory consumption that grows faster than the previous methods (see Figure 2A). However, `egap`, `gsufsort`, and `pfp-ebwt` are far more expensive, and their memory consumption grow even faster. 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.

In the repetitive datasets (human genomes), the results changed drastically (see Figure 3). Our tool `grlBWT` outperformed `ropebwt2` and `pfp-ebwt` in elapsed time, with an average of 4.89 hours versus 20.95 and 9.55 hours of `ropebwt2` and `pfp-ebwt`, respectively. As expected, the time for `grlBWT` grows smoothly with the input size, while the time for `ropeBWT` grows fast. The time function for `pfp-ebwt` also grows smoothly, but the results are still slower than those of `grlBWT` (see Figure 3B). Regarding memory peak, `ropebwt2` is the most efficient tool, with a mean of 18.05 GB. Still, `grlBWT` obtained competitive results, with an average peak of 20.38 GB. In this case, the memory consumption growth in `grlBWT` is slightly steeper than in `ropebwt2`, but it remains smooth. In contrast, `pfp-ebwt` has a more dramatic growth in memory consumption, with an average memory peak of 156.74 GB (Figure 3A).

We observe that `ropeBWT2` and `pfp-ebwt` performed well in one measure, but not in both. In contrast, `grlBWT` maintained a low footprint for both measures, elapsed time and memory consumption. This result demonstrates that our strategy of keeping the intermediate data of the BWT algorithm in compressed format works well when the text is repetitive.

Our experiments on the page cache showed there is an average slowdown of 19% in `grlBWT` when the cache is disabled with the function `posix_advice`. This slowdown factor increases with the input size, being the lowest with `HGA05` (12%) and the highest with `HGA20` (26%). This result is expected as we are only using a static buffer of 8 MB. A simple solution would be to set a dynamic buffer that uses, say, 0.5% of the input instead of the fixed 8 MB.

6 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 not just reduces space usage, but also reduces computation time. Our experimental results showed that our algorithm is competitive with the state-of-the-art tools under not so repetitive scenarios, and that it greatly reduces the computational requirements when the input becomes more repetitive, standing out as the most efficient tool to date (and to our knowledge) in this context.

References

1 Markus J. Bauer, Anthony J. Cox, and Giovanna Rosone. Lightweight algorithms for constructing and inverting the BWT of string collections. *Theoretical Computer Science*, 483:134–148, 2013.

2 Hans-J Boehm, Russ Atkinson, and Michael Plass. Ropes: An alternative to strings. *Software: Practice and Experience*, 25(12):1315–1330, 1995.

3 Paola Bonizzoni, Gianluca Della Vedova, Yuri Pirola, Marco Previtali, and Raffaella Rizzi. Computing the multi–string BWT and LCP array in external memory. *Theoretical Computer Science*, 862:42–58, 2021.

4 Christina Boucher, Davide Cenzato, Zsuzsanna Lipták, Massimiliano Rossi, and Marinella Sciortino. Computing the original eBWT faster, simpler, and with less memory. In *Proc. 28th International Symposium on String Processing and Information Retrieval (SPIRE)*, pages 129–142, 2021.

5 Christina Boucher, Travis Gagie, Alan Kuhnle, Ben Langmead, Giovanni Manzini, and Taher Mun. Prefix–free parsing for building big BWTs. *Algorithms for Molecular Biology*, 14, 2019. Article 13.

6 Michael Burrows and David Wheeler. A block sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, 1994.

7 Diego Díaz-Domínguez and Gonzalo Navarro. A grammar compressor for collections of reads with applications to the construction of the BWT. In *Proc. 31st Data Compression Conference (DCC)*, pages 83–92, 2021.

8 Lavinia Egidi, Felipe A. Louza, Giovanni Manzini, and Guilherme P. Telles. External memory BWT and LCP computation for sequence collections with applications. *Algorithms for Molecular Biology*, 14:6, 2019.

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

10 Travis Gagie, Giovanni Manzini, and Jouni Sirén. Wheeler graphs: A framework for BWT-based data structures. *Theoretical Computer Science*, 698:67–78, 2017.

11 Simon Gog, Timo Beller, Alistair Moffat, and Matthias Petri. From Theory to Practice: Plug and Play with Succinct Data Structures. In *Proc. 13th International Symposium on Experimental Algorithms (SEA)*, pages 326–337, 2014.

12 Richard Karp and Michael Rabin. Efficient randomized pattern–matching algorithms. *IBM Journal of Research and Development*, 31(2):249–260, 1987.

13 Dominik Kempa. Optimal construction of compressed indexes for highly repetitive texts. In *Proc. 30th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1344–1357, 2019.

14 Dominik Kempa and Tomasz 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)*, pages 756–767, 2019.

15 Dominik Kempa and Tomasz Kociumaka. Resolution of the Burrows–Wheeler transform conjecture. In *Proc. 61st Annual Symposium on Foundations of Computer Science (FOCS)*, pages 1002–1013. IEEE, 2020.

16 John C. Kieffer and En Hui Yang. Grammar–based codes: a new class of universal lossless source codes. *IEEE Transactions on Information Theory*, 46(3):737–754, 2000.

17 Pang Ko and Srinivas Aluru. Space efficient linear time construction of suffix arrays. *Journal of Discrete Algorithms*, 3(2-4):143–156, 2005.

18 Ben Langmead, Cole Trapnell, Mihai Pop, and Steven Salzberg. Ultrafast and memory–efficient alignment of short DNA sequences to the human genome. *Genome Biology*, 10(3), 2009. Article R25.

19 Jesper Larsson and Alistair Moffat. Off–line dictionary–based compression. *Proceedings of the IEEE*, 88(11):1722–1732, 2000.

20 Heng Li. Fast construction of fm-index for long sequence reads. *Bioinformatics*, 30(22):3274–3275, 2014.

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

22 Felipe A. Louza, Simon Gog, and Guilherme P. Telles. Inducing enhanced suffix arrays for string collections. *Theoretical Computer Science*, 678(1):22–39, 2017.

23 Felipe A. Louza, Guilherme P. Telles, Simon Gog, Nicola Prezza, and Giovanna Rosone. gsufsort: constructing suffix arrays, LCP arrays and BWTs for string collections. *Algorithms for Molecular Biology*, 15, 2020. Article 18.

24 Robert Love. *Linux kernel development*. Pearson Education, 2010.

25 Veli Mäkinen, Djamal Belazzougui, Fabio Cunial, and Alexandru Tomescu. *Genome-Scale Algorithm Design*. Camb. U. Press, 2015.

26 Udi Manber and Gene Myers. Suffix arrays: a new method for on–line string searches. *SIAM Journal on Computing*, 22(5):935–948, 1993.

27 Gonzalo Navarro. Indexing Highly Repetitive String Collections, Part I: Repetitiveness Measures. *ACM Computing Surveys*, 54(2), 2021. Article 29.

28 Gonzalo Navarro and Veli Mäkinen. Compressed full-text indexes. *ACM Computing Surveys*, 39:article 2, 2007.

29 Ge Nong. Practical linear-time O(1)-workspace suffix sorting for constant alphabets. *ACM Transactions on Information Systems*, 31(3):1–15, 2013.

30 Ge Nong, Sen Zhang, and Wai Hong Chan. Linear suffix array construction by almost pure induced-sorting. In *Proc. 19th Data Compression Conference (DCC)*, pages 193–202, 2009.

31 Daniel Saad Nogueira Nunes, Felipe A. Louza, Simon Gog, Mauricio Ayala-Rincón, and Gonzalo Navarro. A grammar compression algorithm based on induced suffix sorting. In *Proc. 28th Data Compression Conference (DCC)*, pages 42–51, 2018.

32 Enno Ohlebusch. *Bioinformatics Algorithms: Sequence Analysis, Genome Rearrangements, and Phylogenetic Reconstruction*. Oldenbusch Verlag, 2013.

33 Daisuke Okanohara and Kunihiko Sadakane. Practical entropy-compressed rank/select dictionary. In *Proc. 9th Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 60–70. SIAM, 2007.

34 Daisuke Okanohara and Kunihiko Sadakane. A linear–time Burrows–Wheeler transform using induced sorting. In *Proc. 16th International Symposium on String Processing and Information Retrieval (SPIRE)*, pages 90–101, 2009.

35 Zachary D. Stephens, Skylar Y. Lee, Faraz Faghri, Roy H. Campbell, Chengxiang Zhai, Miles J. Efron, Ravishankar Iyer, Michael C. Schatz, Saurabh Sinha, and Gene E. Robinson. Big data: astronomical or genomical? *PLoS Biology*, 13(7):e1002195, 2015.