RESEARCH

Prefix-Free Parsing for Building Big BWTs

Christina Boucher[1*], Travis Gagie[2,3], Alan Kuhnle[1,4], Ben Langmead[5], Giovanni Manzini[6,7] and Taher Mun[5]

*Correspondence:
christinadotboucher@gmail.com
[1]CISE, University of Florida, Gainesville, FL USA
Full list of author information is available at the end of the article

Abstract

High-throughput sequencing technologies have led to explosive growth of genomic databases; one of which will soon reach hundreds of terabytes. For many applications we want to build and store indexes of these databases but constructing such indexes is a challenge. Fortunately, many of these genomic databases are highly-repetitive—a characteristic that can be exploited to ease the computation of the Burrows-Wheeler Transform (BWT), which underlies many popular indexes. In this paper, we introduce a preprocessing algorithm, referred to as *prefix-free parsing*, that takes a text T as input, and in one-pass generates a dictionary D and a parse P of T with the property that the BWT of T can be constructed from D and P using workspace proportional to their total size and $O(|T|)$-time. Our experiments show that D and P are significantly smaller than T in practice, and thus, can fit in a reasonable internal memory even when T is very large. In particular, we show that with prefix-free parsing we can build an 131-megabyte run-length compressed FM-index (restricted to support only counting and not locating) for 1000 copies of human chromosome 19 in 2 hours using 21 gigabytes of memory, suggesting that we can build a 6.73 gigabyte index for 1000 complete human-genome haplotypes in approximately 102 hours using about 1 terabyte of memory.

Keywords: Burrows-Wheeler Transform; prefix-free parsing; compression-aware algorithms; genomic databases

1 Introduction

The money and time needed to sequence a genome have shrunk shockingly quickly and researchers' ambitions have grown almost as quickly: the Human Genome Project cost billions of dollars and took a decade but now we can sequence a genome for about a thousand dollars in about a day. The 1000 Genomes Project [1] was announced in 2008 and completed in 2015, and now the 100,000 Genomes Project is well under way [2]. With no compression 100,000 human genomes occupy roughly 300 terabytes of space, and genomic databases will have grown even more by the time a standard research machine has that much RAM. At the same time, other initiatives have began to study how microbial species behave and thrive in environments. These initiatives are generating public datasets, which are larger than the 100,000 Genomes Project. For example, in recent years, there has been an initiative to move toward using whole genome sequencing to accurately identify and track foodborne pathogens (e.g. antibiotic-resistant bacteria) [3]. This led to the GenomeTrakr initiative, which is a large public effort to use genome sequencing for surveillance and detection of outbreaks of foodborne illnesses. Currently, GenomeTrakr includes over 100,000 samples, spanning several species available through this initiative—a number that

continues to rise as datasets are continually added [4]. Unfortunately, analysis of this data is limited due to their size, even though the similarity between genomes of individuals of the same species means the data is highly compressible.

These public databases are used in various applications — e.g., to detect genetic variation within individuals, determine evolutionary history within a population, and assemble the genomes of novel (microbial) species or genes. Pattern matching within these large databases is fundamental to all these applications, yet repeatedly scanning these — even compressed — databases is infeasible. Thus, for these and many other applications, we want to build and use indexes from the database. Since these indexes should fit in RAM and cannot rely on word boundaries, there are only a few candidates. Many of the popular indexes in bioinformatics are based on the Burrows-Wheeler Transform (BWT) [5] and there have been a number of papers about building BWTs for genomic databases, e.g., [6] and references therein. However, it is difficult to process anything more than a few terabytes of raw data per day with current techniques and technology because of the difficulty of working in external memory.

Since genomic databases are often highly repetitive, we revisit the idea of applying a simple compression scheme and then computing the BWT from the resulting encoding in internal memory. This is far from being a novel idea — e.g., Ferragina, Gagie and Manzini's `bwtdisk` software [7] could already in 2010 take advantage of its input being given compressed, and Policriti and Prezza [8] showed how to compute the BWT from the LZ77 parse of the input using $O(n(\log r + \log z))$-time and $O(r+z)$-space, where n is the length of the uncompressed input, r is the number of runs in the BWT and z is the number of phrases in the LZ77 parse — but we think the preprocessing step we describe here, *prefix-free parsing*, stands out because of its simplicity and flexibility. Once we have the results of the parsing, which are a dictionary and a parse, building the BWT out of them is more involved, yet when our approach works well, the dictionary and the parse are together much smaller than the initial dataset and that makes the BWT computation less resource-intensive.

Our Contributions. In this paper, we formally define and present prefix-free parsing. The main idea of this method is to divide the input text into overlapping variable-length phrases with delimiting prefixes and suffixes. To accomplish this division, we slide a window of length w over the text and, whenever the Karp-Rabin hash of the window is 0 modulo p, we terminate the current phrase at the end of the window and start the next one at the beginning of the window. This concept is partly inspired by `rsync`'s [9] use of a rolling hash for content-slicing. Here, w and p are parameters that affect the size of the dictionary of distinct phrases and the number of phrases in the parse. This takes linear-time and one pass over the text, or it can be sped up by running several windows in different positions over the text in parallel and then merging the results.

Just as `rsync` can usually recognize when most of a file remains the same, we expect that for most genomic databases and good choices of w and p, the total length of the phrases in the dictionary and the number of phrases in the parse will be small in comparison to the uncompressed size of the database. We demonstrate experimentally that with prefix-free parsing we can compute BWT using less memory and equivalent time. In particular, using our method we reduce peak memory usage

up to 10x over a standard baseline algorithm which computes the BWT by first computing the suffix array using the algorithm SACA-K [10], while requiring roughly the same time on large sets of salmonella genomes obtained from GenomeTrakr.

In Section 3, we show how we can compute the BWT of the text from the dictionary and the parse alone using workspace proportional only to their total size, and time linear in the uncompressed size of the text when we can work in internal memory. In Section 4 we describe our implementation and report the results of our experiments showing that in practice the dictionary and parse often are significantly smaller than the text and so may fit in a reasonable internal memory even when the text is very large, and that this often makes the overall BWT computation both faster and smaller. In Section 5 we describe how we build run-length compressed FM-indexes [11] (which only support counting and not locating) for datasets consisting of 50, 100, 200 and 500 using prefix-free parsing. Our results suggest that we can build a roughly 6.73-gigabyte index for 1000 complete human genomes in about 102 hours using about 1.1 terabytes of memory. Prefix-free parsing and all accompanied documents are available at `https://gitlab.com/manzai/Big-BWT`.

2 Review of the Burrows-Wheeler Transform

As part of the Human Genome Project, researchers had to piece together a huge number of relatively tiny, overlapping pieces of DNA, called reads, to assemble a reference genome about which they had little prior knowledge. Once the Project was completed, however, they could then use that reference genome as a guide to assemble other human genomes much more easily. To do this, they indexed the reference genome such that, after running a DNA sample from a new person through a sequencing machine and obtaining another collection of reads, for each of those new reads they could quickly determine which part of the reference genome it matched most closely. Since any two humans are genetically very similar, aligning the new reads against the reference genome gives a good idea of how they are really laid out in the person's genome.

In practice, the best solutions to this problem of indexed approximate matching work by reducing it to a problem of indexed exact matching, which we can formalize as follows: given a string T (which can be the concatenation of a collection of strings, terminated by special symbols), pre-process it such that later, given a pattern P, we can quickly list all the locations where P occurs in T. We now start with a simple but impractical solution to the latter problem, and then refine it until we arrive at a fair approximation of the basis of most modern assemblers, illustrating the workings of the Burrows-Wheeler Transform (BWT) along the way.

Suppose we want to index the three strings GATTACAT, GATACAT and GATTAGATA, so $T[0..n-1] = \text{GATTACAT}\$_1\text{GATACAT}\$_2\text{GATTAGATA}\$_3$, where $\$_1$, $\$_2$ and $\$_3$ are terminator symbols. Perhaps the simplest solution to the problem of indexing T is to build a trie of the suffixes of the three strings in our collection (i.e., an edge-labelled tree whose root-to-leaf paths are the suffixes of those strings) with each leaf storing the starting position of the suffix labelling the path to that leaf, as shown in Figure 1.

Suppose every node stores pointers to its children and its leftmost and rightmost leaf descendants, and every leaf stores a pointer to the next leaf to its right. Then

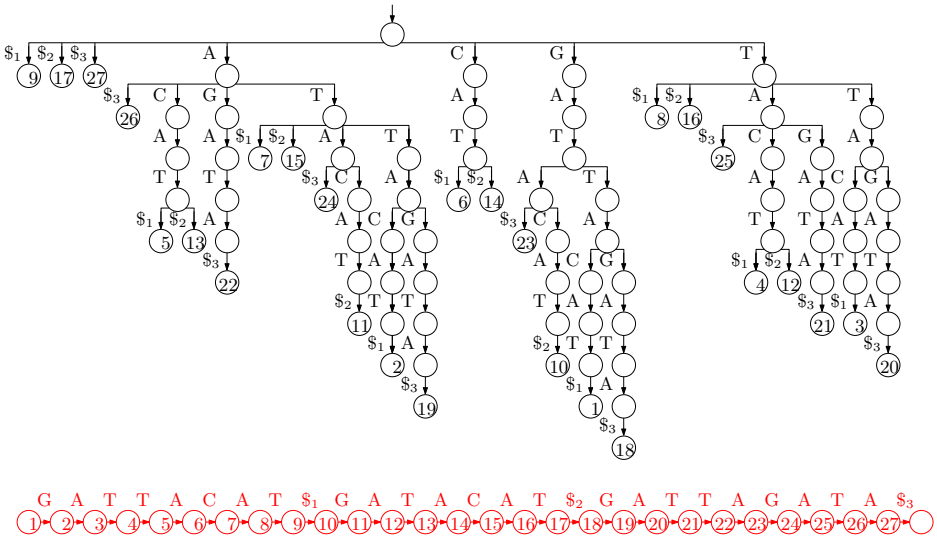


Figure 1: The suffix trie for our example with the three strings GATTACAT, GATACAT and GATTAGATA. The input is shown at the bottom, in red because we do not need to store it.

given $P[0..m-1]$, we can start at the root and descend along a path (if there is one) such that the label on the edge leading to the node at depth i is $P[i-1]$, until we reach a node v at depth m. We then traverse the leaves in v's subtree, reporting the the starting positions stored at them, by following the pointer from v to its leftmost leaf descendant and then following the pointer from each leaf to the next leaf to its right until we reach v's rightmost leaf descendant.

The trie of the suffixes can have a quadratic number of nodes, so it is impractical for large strings. If we remove nodes with exactly one child (concatenating the edge-labels above and below them), however, then there are only linearly many nodes, and each edge-label is a substring of the input and can be represented in constant space if we have the input stored as well. The resulting structure is essentially a suffix tree (although it lacks suffix and Weiner links), as shown in Figure 2. Notice that the label of the path leading to a node v is the longest common prefix of the suffixes starting at the positions stored at v's leftmost and rightmost leaf descendants, so we can navigate in the suffix tree, using only the pointers we already have and access to the input.

Although linear, the suffix tree still takes up an impractical amount of space, using several bytes for each character of the input. This is significantly reduced if we discard the shape of the tree, keeping only the input and the starting positions in an array, which is called the suffix array (SA). The SA for our example is shown in Figure 3. Since the entries of the SA are the starting points of the suffixes in lexicographic order, with access to T we can perform two binary searches to find the endpoints of the interval of the suffix array containing the starting points of suffixes starting with P: at each step, we consider an entry SA$[i]$ and check if $T[\text{SA}[i]]$ lexicographically precedes P. This takes a total of $O(m \log n)$ time done naïvely,

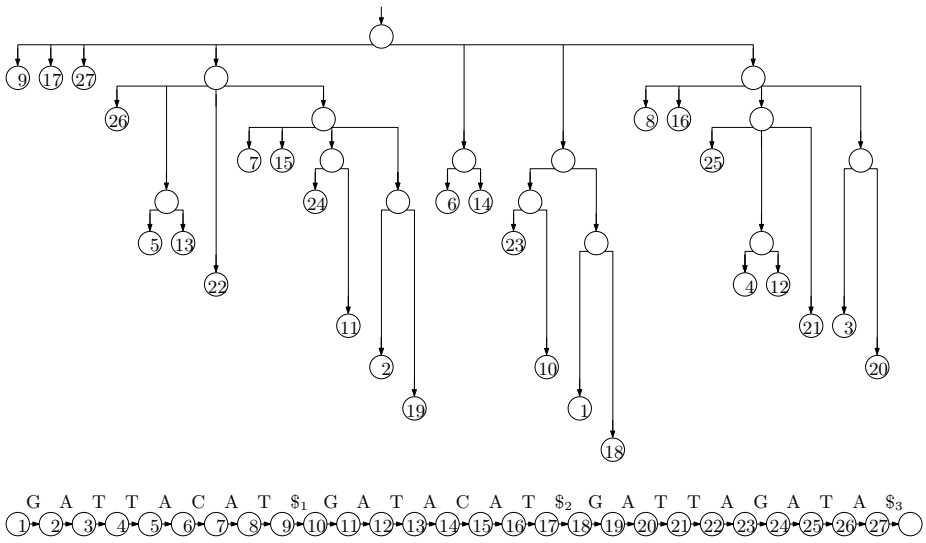


Figure 2: The suffix tree for our example. We now also need to store the input.

and can be sped up with more sophisticated searching and relatively small auxiliary data structures.

Even the SA takes linear space, however, which is significantly more than what is needed to store the input when the alphabet is small (as it is in the case of DNA). Let Ψ be the function that, given the position of a value $i < n - 1$ in the SA, returns the position of $i + 1$. Notice that, if we write down the first character of each suffix in the order they appear in the SA, the result is a sorted list of the characters in T, which can be stored using using $O(\log n)$ bits for each character in the alphabet. Once we have this list stored, given a position i in SA, we can return $T[\mathrm{SA}[i]]$ efficiently.

Given a position i in SA and a way to evaluate Ψ, we can extract $T[SA[i]..n-1]$ by writing $T[\mathrm{SA}[i]], T[\mathrm{SA}[\Psi(i)]], T[\mathrm{SA}[\Psi^2(i)]], \ldots$. Therefore, we can perform the same kind of binary search we use when with access to a full suffix array. Notice that if $T[\mathrm{SA}[i]] \prec T[\mathrm{SA}[i+1]]$ then $\Psi(i) < \Psi(i+1)$, meaning that $\Psi(1), \ldots, \Psi(n)$ can be divided into σ increasing consecutive subsequences, where σ is the size of the alphabet. Here, $\prec$ denotes lexicographic precedence. It follows that we can store $nH_0(T) + o(n \log \sigma)$ bits, where $H_0(T)$ is the 0th-order empirical entropy of T, such that we can quickly evaluate Ψ. This bound can be improved with a more careful analysis.

Now suppose that instead of a way to evaluate Ψ, we have a way to evaluate quickly its inverse, which is called the last-to-first (LF) mapping. (This name was not chosen because, if we start with the position of n in the suffix array and repeatedly apply the LF mapping we enumerate the positions in the SA in decreasing order of their contents, ending with 1; to some extent, the name is a lucky coincidence.) The LF mapping for our example is also shown with arrows in Figure 3. Since it is the inverse of Ψ, the sequence $\mathrm{LF}(1), \ldots, \mathrm{LF}(n)$ can be partitioned into σ incrementing subsequences: for each character c in the alphabet, if the starting positions of suffixes preceded by copies of c are stored in $\mathrm{SA}[j_1], \ldots, \mathrm{SA}[j_t]$ (appearing in that order in

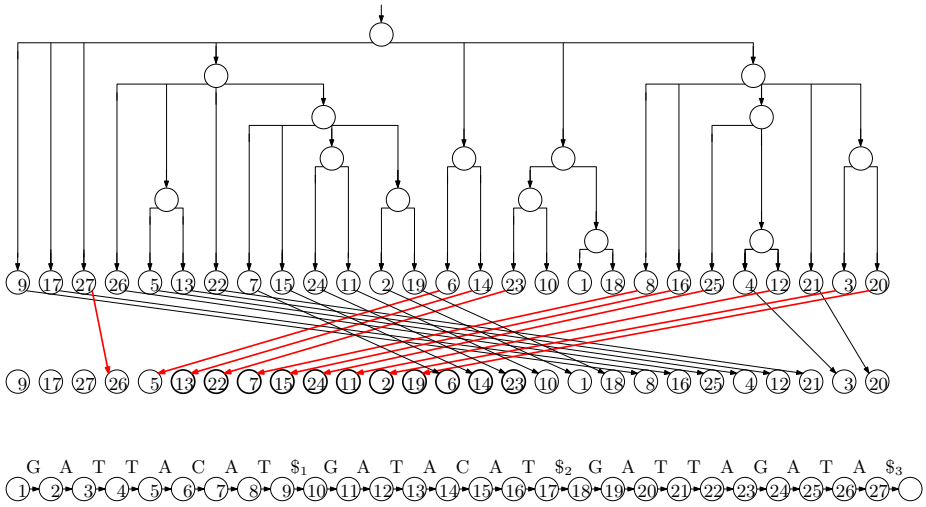


Figure 3: The suffix array for our example is the sequence of values stored in the leaves of the tree (which we need not store explicitly). The LF mapping is shown as the arrows between two copies of the suffix array; the arrows to values i such that $T[\mathrm{SA}[i]] = \mathtt{A}$ are in red, to illustrate that they point to consecutive positions in the suffix array and do not cross. Since Ψ is the inverse of the LF mapping, it can be obtained by simply reversing the direction of the arrows.

the SA), then $\mathrm{LF}(j_1)$ is 1 greater than the number of characters lexicographically less than c in T and $\mathrm{LF}(j_2), \ldots, \mathrm{LF}(j_t)$ are the next $t-1$ numbers. Figure 3 illustrates this, with heavier arrows to values i such that $T[\mathrm{SA}[i]] = \mathrm{A}$, to illustrate that they point to consecutive positions in the suffix array and do not cross.

Consider the interval $I_{P[i..m-1]}$ of the SA containing the starting positions of suffixes beginning with $P[i..m-1]$, and the interval $I_{P[i-1]}$ containing the starting positions of suffixes beginning with $P[i-1]$. If we apply the LF mapping to the SA positions in $I_{P[i..m-1]-1}$, the SA positions we obtain that lie in $I_{P[i-1]}$ for a consecutive subinterval, containing the starting positions in T of suffixes beginning with $P[i-1..m-1]$. Therefore, we can search also with the LF mapping.

If we write the character preceding each suffix of T (considering it to be cyclic) in the lexicographic order of the suffixes, the result is the Burrows-Wheeler Transform (BWT) of T. A rank data structure over the BWT (which, given a character and a position, returns the number of occurrences of that character up to that position) can be used to implement searching with the LF-mapping, together with an array C indicating for each character in the alphabet how many characters in T are lexicographically strictly smaller than it. Specifically,

$$\mathrm{LF}(i) = \mathrm{BWT}.\mathrm{rank}_{\mathrm{BWT}[i]}(i) + C[\mathrm{BWT}[i]]\,.$$

If follows that, to compute $I_{P[i-1..m-1]}$ from $I_{P[i..m-1]}$, we perform a rank query for $P[i-1]$ immediately before the beginning of $I_{P[i..m-1]}$ and add $C[P[i+1]]+1$ to the result, to find the beginning of $I_{P[i-1..m-1]}$; and we perform a rank query for $P[i-1]$ at the end of $I_{P[i..m-1]}$ and add $C[P[i+1]]$ to the result, to find the

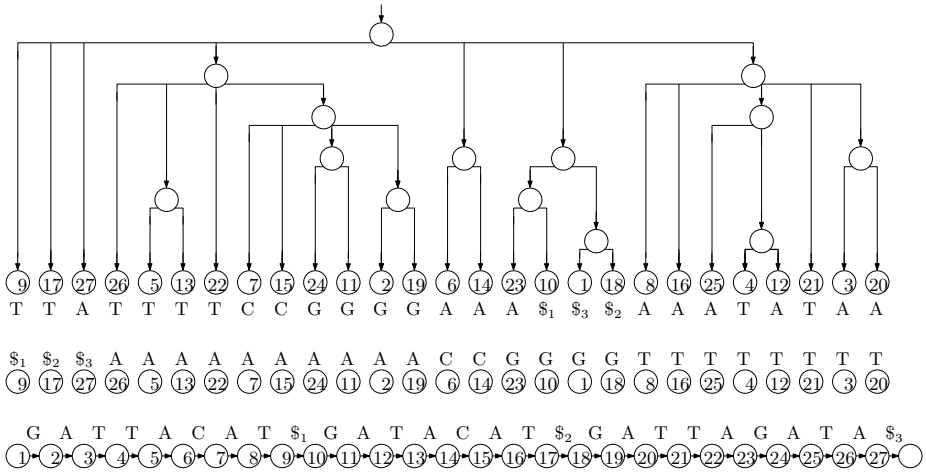


Figure 4: The BWT and the sorted list of characters for our example. Drawing arrows between corresponding occurrences of characters in the two strings gives us the diagram for the LF-mapping.

end of $I_{P[i-1..m-1]}$. Figure 4 shows the BWT for our example, and the sorted list of characters in T. Comparing it to Figure 3 makes the formula above clear: if BWT$[i]$ is the jth occurrence of that character in the BWT, then the arrow from LF(i) leads from i to the position of the jth occurrence of that character in the sorted list. This is the main idea behind FM-indexes [11], and the main motivation for bioinformaticians to be interested in building BWTs.

3 Theory of Prefix Free Parsing

We let $E \subseteq \Sigma^w$ be any set of strings each of length $w \geq 1$ over the alphabet Σ and let $E' = E \cup \{\#, \$^w\}$, where $\#$ and $\$$ are special symbols lexicographically less than any in Σ. We consider a text $T[0..n-1]$ over Σ and let D be the maximum set such that for $d \in D$ the following conditions hold

- d is a substring of $\# T \$^w$,
- exactly one proper prefix of d is in E',
- exactly one proper suffix of d is in E',
- no other substring of d is in E'.

Given T and a way to recognize strings in E, we can build D iteratively by scanning $\# T \$^w$ to find occurrences of elements of E', and adding to D each substring of $\# T \$^w$ that starts at the beginning of one such occurrence and ends at the end of the next one. While we are building D we also build a list P of the occurrences of the elements of D in T, which we call the parse (although each consecutive pair of elements overlap by w characters, so P is not a partition of the characters of $\# T \$^w$). In P we identify each element of D with its lexicographic rank.

For example, suppose we have $\Sigma = \{!, \mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$, $w = 2$, $E = \{\mathtt{AC}, \mathtt{AG}, \mathtt{T!}\}$ and

$$T = \mathtt{GATTACAT!GATACAT!GATTAGATA}\,.$$

Then, we get

$$D = \{\#\texttt{GATTAC}, \texttt{ACAT!}, \texttt{AGATA\$\$}, \texttt{T!GATAC}, \texttt{T!GATTAG}\},$$

the parse of $\#\,T\,\$^w$ is

$$\#\texttt{GATTAC ACAT! T!GATAC ACAT! T!GATTAG AGATA\$\$}$$

and, identifying elements of D by their lexicographic ranks, the resulting array P is $P = [0, 1, 3, 1, 4, 2]$.

Next, we define S as the set of suffixes of length greater than w of elements of D. In our previous example we get

$$\begin{aligned} S \;=\; \{&\#\texttt{GATTAC}, \texttt{GATTAC}, \ldots, \texttt{TAC},\\ &\texttt{ACAT!}, \texttt{CAT!}, \texttt{AT!},\\ &\texttt{AGATA\$\$}, \texttt{GATA\$\$}, \ldots, \texttt{A\$\$},\\ &\texttt{T!GATAC}, \texttt{!GATAC}, \ldots, \texttt{TAC},\\ &\texttt{T!GATTAG}, \texttt{!GATTAG}, \ldots, \texttt{TAG}\}. \end{aligned}$$

Lemma 1 *S is a prefix-free set.*

Proof If $s \in S$ were a proper prefix of $s' \in S$ then, since $|s| > w$, the last w characters of s — which are an element of E' — would be a substring of s' but neither a proper prefix nor a proper suffix of s'. Therefore, any element of D with s' as a suffix would contain at least three substrings in E', contrary to the definition of D. □

Lemma 2 *Suppose $s, s' \in S$ and $s \prec s'$. Then $sx \prec s'x'$ for any strings $x, x' \in (\Sigma \cup \{\#, \$\})^*$.*

Proof By Lemma 1, s and s' are not proper prefixes of each other. Since they are not equal either (because $s \prec s'$), it follows that sx and $s'x'$ differ on one of their first $\min(|s|, |s'|)$ characters. Therefore, $s \prec s'$ implies $sx \prec s'x'$. □

Lemma 3 *For any suffix x of $\#\,T\,\$^w$ with $|x| > w$, exactly one prefix s of x is in S.*

Proof Consider the substring d stretching from the beginning of the last occurrence of an element of E' that starts before or at the starting position of x, to the end of the first occurrence of an element of E' that starts strictly after the starting position of x. Regardless of whether d starts with # or another element of E', it is prefixed by exactly one element of E'; similarly, it is suffixed by exactly one element of E'. It follows that d is an element of D. Let s be the prefix of x that ends at the end of that occurrence of d in $\#\,T\,\$^w$, so $|s| > w$ and is a suffix of an element of D and thus $s \in S$. By Lemma 1, no other prefix of x is in S. □

Because of Lemma 3, we can define a function f mapping each suffix x of $\#\,T\,\$^w$ with $|x| > w$ to the unique prefix s of x with $s \in S$.

Lemma 4 *Let x and x' be suffixes of $\#\,T\,\$^w$ with $|x|, |x'| > w$. Then $f(x) \prec f(x')$ implies $x \prec x'$.*

Proof By the definition of f, $f(x)$ and $f(x')$ are prefixes of x and x' with $|f(x)|, |f(x')| > w$. Therefore, $f(x) \prec f(x')$ implies $x \prec x'$ by Lemma 2. □

Define $T'[0..n] = T\,\$$. Let g be the function that maps each suffix y of T' to the unique suffix x of $\#\,T\,\$^w$ that starts with y, except that it maps $T'[n] = \$$ to $\#\,T\,\$^w$. Notice that $g(y)$ always has length greater than w, so it can be given as an argument to f.

Lemma 5 *The permutation that lexicographically sorts $T[0..n-1]\,\$^w, \ldots, T[n-1]\,\$^w, \#\,T\,\$^w$ also lexicographically sorts $T'[0..n], \ldots, T'[n-1..n], T'[n]$.*

Proof Appending copies of \$ to the suffixes of T' does not change their relative order, and just as $\#\,T\,\$^w$ is the lexicographically smallest of $T[0..n-1]\,\$^w, \ldots, T[n-1]\,\$^w, \#\,T\,\$^w$, so $T'[n] = \$$ is the lexicographically smallest of $T'[0..n], \ldots, T'[n-1..n], T'[n]$. □

Let β be the function that, for $i < n$, maps $T'[i]$ to the lexicographic rank of $f(g(T'[i{+}1..n]))$ in S, and maps $T[n]$ to the lexicographic rank of $f(g(T')) = f(T\,\$^w)$.

Lemma 6 *Suppose β maps k copies of a to $s \in S$ and maps no other characters to s, and maps a total of t characters to elements of S lexicographically less than s. Then the $(t+1)$st through $(t+k)$th characters of the BWT of T' are copies of a.*

Proof By Lemmas 4 and 5, if $f(g(y)) \prec f(g(y'))$ then $y \prec y'$. Therefore, β partially sorts the characters in T' into their order in the BWT of T'; equivalently, the characters' partial order according to β can be extended to their total order in the BWT. Since every total extension of β puts those k copies of a in the $(t+1)$st through $(t+k)$th positions, they appear there in the BWT. □

From D and P, we can compute how often each element $s \in S$ is preceded by each distinct character a in $\#\,T\,\$^w$ or, equivalently, how many copies of a are mapped by β to the lexicographic rank of s. If an element $s \in S$ is a suffix of only one element $d \in D$ and a proper suffix of that — which we can determine first from D alone — then β maps only copies of the the preceding character of d to the rank of s, and we can compute their positions in the BWT of T'. If $s = d$ or a suffix of several elements of D, however, then β can map several distinct characters to the rank of s. To deal with these cases, we can also compute which elements of D contain which characters mapped to the rank of s. We will explain in a moment how we use this information.

For our example, $T =$ `GATTACAT!GATACAT!GATTAGATA`, we compute the information shown in Table 1. To ease the comparison to the standard computation of the BWT

Table 1: The information we compute for our example, $T =$ `GATTACAT!GATACAT!GATTAGATA`. Each line shows the lexicographic rank r of an element $s \in S$; the characters mapped to r by β; s itself; the elements of D from which the mapped characters originate; the total frequency with which characters are mapped to r; and the preceding partial sum of the frequencies.

rank	mapped characters	suffix	sources	frequency	preceding partial sum
0	A	#GATTAC	1	1	0
1	T	!GATAC	2	1	1
2	T	!GATTAG	3	1	2
3	T	A$$	5	1	3
4	T	ACAT!	4	2	4
5	T	AGATA$$	5	1	6
6	C	AT!	4	2	7
7	G	ATA$$	5	1	9
8	G	ATAC	2	1	10
9	G	ATTAC	1	1	11
10	G	ATTAG	3	1	12
11	A	CAT#	4	2	13
12	A	GATA$$	5	1	15
13	!	GATAC	2	1	16
14	$	GATTAC	1	1	17
15	!	GATTAG	3	1	18
16	A	T!GATAC	2	1	19
17	A	T!GATTAG	3	1	20
18	A	TA$$	5	1	21
19	T, A	TAC	1; 2	2	22
20	T	TAG	3	1	24
21	A	TTAC	1	1	25
22	A	TTAG	3	1	26

of $T'\$$, shown in Table 2, we write the characters mapped to each element $s \in S$ before s itself.

By Lemma 6, from the characters mapped to each rank by β and the partial sums of frequencies with which β maps characters to the ranks, we can compute the subsequence of the BWT of T' that contains all the characters β maps to elements of S, which are not complete elements of D and to which only one distinct character is mapped. We can also leave placeholders where appropriate for the characters β maps to elements of S, which are complete elements of D or to which more than one distinct character is mapped. For our example, this subsequence is `ATTTTTTCCGGGGAAA!$!AAA--TAA`. Notice we do not need all the information in P to compute this subsequence, only D and the frequencies of its elements in P.

Suppose $s \in S$ is an entire element of D or a suffix of several elements of D, and occurrences of s are preceded by several distinct characters in $\#\,T\,\$^w$, so β assigns s's lexicographic rank in S to several distinct characters. To deal with such cases, we can sort the suffixes of the parse P and apply the following lemma.

Lemma 7 *Consider two suffixes t and t' of $\#\,T\,\$^w$ starting with occurrences of $s \in S$, and let q and q' be the suffixes of P encoding the last w characters of those occurrences of s and the remainders of t and t'. If $t \prec t'$ then $q \prec q'$.*

Proof Since s occurs at least twice in $\#\,T\,\$^w$, it cannot end with $\$^w$ and thus cannot be a suffix of $\#\,T\,\$^w$. Therefore, there is a first character on which t and t' differ.

Table 2: The BWT for $T' =$ `GATTACAT!GATACAT!GATTAGATA$`. Each line shows a position in the BWT; the character in that position; and the suffix immediately following that character in T'.

i	BWT[i]	suffix
0	A	$
1	T	!GATACAT!GATTAGATA$
2	T	!GATTAGATA$
3	T	A$
4	T	ACAT!GATACAT!GATTAGATA$
5	T	ACAT!GATTAGATA$
6	T	AGATA$
7	C	AT!GATACAT!GATTAGATA$
8	C	AT!GATTAGATA$
9	G	ATA$
10	G	ATACAT!GATTAGATA$
11	G	ATTACAT!GATACAT!GATTAGATA$
12	G	ATTAGATA$
13	A	CAT!GATACAT!GATTAGATA$
14	A	CAT!GATTAGATA$
15	A	GATA$
16	!	GATACAT!GATTAGATA$
17	$	GATTACAT!GATACAT!GATTAGATA$
18	!	GATTAGATA$
19	A	T!GATACAT!GATTAGATA$
20	A	T!GATTAGATA$
21	A	TA$
22	T	TACAT!GATACAT!GATTAGATA$
23	A	TACAT!GATTAGATA$
24	T	TAGATA$
25	A	TTACAT!GATACAT!GATTAGATA$
26	A	TTAGATA$

Since the elements of D are represented in the parse by their lexicographic ranks, that character forces $q \prec q'$. □

We consider the occurrences in P of the elements of D suffixed by s, and sort the characters preceding those occurrences of s into the lexicographic order of the remaining suffixes of P which, by Lemma 7, is their order in the BWT of T'. In our example, `TAC` $\in S$ is preceded in $\#T\$\$$ by a `T` when it occurs as a suffix of `#GATTAC` $\in D$, which has rank 0 in D, and by an `A` when it occurs as a suffix of `T!GATAC` $\in D$, which has rank 3 in D. Since the suffix following 0 in $P = 0, 1, 3, 1, 4, 2$ is lexicographically smaller than the suffix following 3, that `T` precedes that `A` in the BWT.

Since we need only D and the frequencies of its elements in P to apply Lemma 6 to build and store the subsequence of the BWT of T' that contains all the characters β maps to elements of S, to which only one distinct character is mapped, this takes space proportional to the total length of the elements of D. We can then apply Lemma 7 to build the subsequence of missing characters in the order they appear in the BWT. Although this subsequence of missing characters could take more space than D and P combined, as we generate them we can interleave them with the first subsequence and output them, thus still using workspace proportional to the total length of P and the elements of D and only one pass over the space used to store the BWT.

Notice, we can build the first subsequence from D and the frequencies of its elements in P; store it in external memory; and make a pass over it while we

generate the second one from D and P, inserting the missing characters in the appropriate places. This way we use two passes over the space used to store the BWT, but we may use significantly less workspace.

Summarizing, assuming we can recognize the strings in E quickly, we can quickly compute D and P with one scan over T. From D and P, with Lemmas 6 and 7, we can compute the BWT of $T' = T\,\$$ by sorting the suffixes of the elements of D and the suffixes of P. Since there are linear-time and linear-space algorithms for sorting suffixes when working in internal memory, this implies our main theoretical result:

Theorem 1 *We can compute the BWT of $T\,\$$ from D and P using workspace proportional to sum of the total length of P and the elements of D, and $O(n)$ time when we can work in internal memory.*

The significance of the above theorem is that if the text T contains many repetitions the dictionary of distinct phrases D will be relatively small and, if the dictionary words are sufficiently long, also the parse P will be much smaller than T. Thus, even if T is very large, if D and P fit into internal memory then using Theorem 1 we can efficiently build the BWT for T doing the critical computations in RAM after a single sequential scanning of T during the parsing phase.

4 Prefix free parsing in practice

We have implemented our BWT construction based on prefix free parsing and applied it to collections of repetitive documents and genomic databases. Our purpose is to test our conjectures that 1) with a good choice of the parsing strategy the total length of the phrases in the dictionary and the number of phrases in the parse will both be small in comparison to the uncompressed size of the collection, and 2) computing the BWT from the dictionary and the parse leads to an overall speed-up and reduction in memory usage. In this section we describe our implementation and then report our experimental results.

4.1 Implementation

Given a window size w, we select a prime p and we define the set E described in Section 3, as the set of length-w strings such that their Karp-Rabin fingerprint modulo p is zero. With this choice our parsing algorithm works as follows. We slide a window of length w over the text, keeping track of the Karp-Rabin hash of the window; we also keep track of the hash of the entire prefix of the current phrase that we have processed so far. Whenever the hash of the window is 0 modulo p, we terminate the current phrase at the end of the window and start the next one at the beginning of the window. We prepend a NUL character to the first phrase and append w copies of NUL to the last phrase. If the text ends with w characters whose hash is 0 modulo p, then we take those w character to be the beginning of the last phrase and append to them w copies of NUL. We note that we prepend and append copies of the same NUL character; although using different characters simplifies the proofs in Section 3, it is not essential in practice.

We keep track of the set of hashes of the distinct phrases in the dictionary so far, as well as the phrases' frequencies. Whenever we terminate a phrase, we check if

its hash is in that set. If not, we add the phrase to the dictionary and its hash to the set, and set its frequency to 1; if so, we compare the current phrase to the one in the dictionary with the same hash to ensure they are equal, then increment its frequency. (Using a 64-bit hash the probability of there being a collision is very low, so we have not implemented a recovery mechanism if one occurs.) In both cases, we write the hash to disk.

When the parsing is complete, we have generated the dictionary D and the parsing $P = p_1, p_2, \ldots, p_z$, where each phrase p_i is represented by its hash. Next, we sort the dictionary and make a pass over P to substitute the phrases' lexicographic ranks for their hashes. This gives us the final parse, still on disk, with each entry stored as a 4-byte integer. We write the dictionary to disk phrase by phrase in lexicographic order with a special end-of-phrase terminator at the end of each phrase. In a separate file we store the frequency of each phrase in as a 4-byte integer. Using four bytes for each integer does not give us the best compression possible, but it makes it easy to process the frequency and parse files later. Finally, we write to a separate file the array W of length $|P|$ such that $W[j]$ is the character of p_j in position $w+1$ from the end (recall each phrase has length greater than w). These characters will be used to handle the elements of S that are also elements of D.

Next, we compute the BWT of the parsing P, with each phrase represented by its 4-byte lexicographic rank in D. The computation is done using the SACA-K suffix array construction algorithm [10] which, among the linear time algorithms, is the one using the smallest workspace and is particularly suitable for input over large alphabets. Instead of storing $BWT(P) = b_1, b_2, \ldots, b_z$, we save the same information in a format more suitable for the next phase. We consider the dictionary phrases in lexicographic order, and, for each phrase d_i, we write the list of BWT positions where d_i appears. We call this the inverted list for phrase d_i. Since the size of the inverted list of each phrase is equal to its frequency, which is available separately, we write to file the plain concatenation of the inverted lists using again four bytes per entry, for a total of $4|P|$ bytes. In this phase we also permute the elements of W so that now $W[j]$ is the character coming from the phrase that precedes b_j in the parsing, i.e. $P[SA[j]-2]$.

In the final phase of the algorithm we compute the BWT of the input T. We deviate slightly from the description in Section 3 in that instead of lexicographically sorting the suffixes in D larger than w we sort all of them and later ignore those which are of length $\leq w$. The sorting is done applying the gSACAK algorithm [12] which computes the SA and LCP array for the set of dictionary phrases. We then proceed as in Section 3. If during the scanning of the sorted set S we meet s which is a proper suffix of several elements of D we use a heap to merge their respective inverted lists writing a character to the final BWT file every time we pop a position from the heap. If we meet s which coincides with a dictionary phrase d we write the characters retrieved from W from the positions obtained from d's inverted list.

It turns out that the the most expensive phases of the algorithm are the first, where we compute the parsing of T, and the last, where we compute $BWT(T)$ from the SA of D and the inverted lists for D's phrases. Fortunately, both phases can be sped-up using multiple threads in parallel. To parallelize the first phase we split the input into equal size chunks, and we assign each chunk to a different

Table 3: The dictionary and parse sizes for several files from the Pizza & Chili repetitive corpus, with three settings of the parameters w and p. All sizes are reported in megabytes; percentages are the sums of the sizes of the dictionaries and parses, divided by the sizes of the uncompressed files.

		$w=6, p=20$			$w=8, p=50$			$w=10, p=100$		
file	size	dict.	parse	%	dict.	parse	%	dict.	parse	%
`cere`	440	61	77	31	43	159	46	**89**	**17**	**24**
`cere_no_Ns`	409	33	77	27	**43**	**33**	**18**	60	17	19
`dna.001.1`	100	8	20	27	**13**	**9**	**21**	21	4	25
`einstein.en.txt`	446	2	87	20	3	39	9	**4**	**17**	**5**
`influenza`	148	16	28	30	**32**	**12**	**29**	49	6	37
`kernel`	247	14	52	26	14	20	13	**15**	**10**	**10**
`world_leaders`	45	5	5	21	**8**	**2**	**21**	11	1	26
`world_leaders_no_dots`	23	4	5	34	**6**	**2**	**31**	7	1	33

thread. Using this simple approach, we obtained a speed-up of a factor 2 using four threads, but additional threads do not yield substantial improvements. There are two likely reasons for that: 1) during the parsing all threads need to update the same dictionary, and 2) each thread has to write to disk its portion of the parsing and, unless the system has multiple disks, disk access becomes a bottleneck. To parallelize the computation of the final $BWT(T)$ we use a different approach. The main thread scans the suffix array of the dictionary and as soon as it finds a range of equal suffixes it passes such range to an helper thread that computes and writes to disk the corresponding portion of $BWT(T)$. Again, we were able to reduce the running time of this phase by factor 2 using four threads. In the next section we only report the running times for the single thread algorithm since we are still working to improve our multi-thread version.

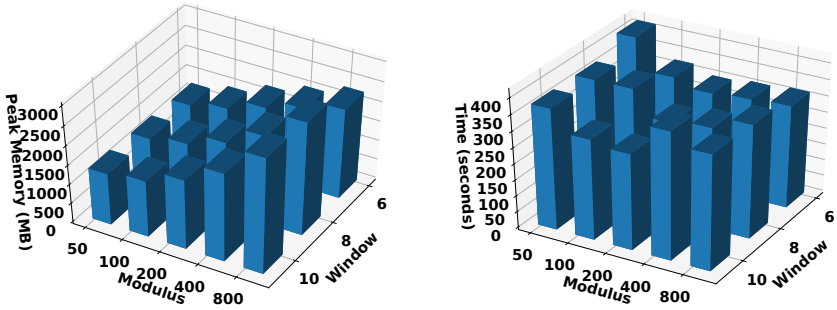
4.2 Experiments

In this section, the parsing and BWT computation are experimentally evaluated. All experiments were run on a server with Intel(R) Xeon(R) CPU E5-2640 v4 @ 2.40GHz and 756 gigabytes of RAM.

Table 3 shows the sizes of the dictionaries and parses for several files from the Pizza & Chili repetitive corpus [13], with three settings of the parameters w and p. We note that `cere` contains long runs of `N`s and `world_leaders` contains long runs of periods, which can either cause many phrases, when the hash of w copies of those characters is 0 modulo p, or a single long phrase otherwise; we also display the sizes of the dictionaries and parses for those files with all `N`s and periods removed. The dictionaries and parses occupy between 5 and 31 percent of the space of the uncompressed files.

Table 4 shows the sizes of the dictionaries and parses for prefixes of a database of Salmonella genomes [14]. The dictionaries and parses occupy between 14 and 44 percent of the space of the uncompressed files, with the compression improving as the number of genomes increases. In particular, the dataset of ten thousand genomes takes nearly 50 GB uncompressed, but with $w = 10$ and $p = 100$ the dictionary and parse take only about 7 GB together, so they would still fit in the RAM of a commodity machine. This seems promising, and we hope the compression is even better for larger genomic databases.

Table 5 shows the runtime and peak memory usage for computing the BWT from the parsing for the database of Salmonella genomes. As a baseline for comparison,



(a) Peak memory (MB) vs. (w, p) (b) Time (s) vs. (w, p)

Figure 5: Results versus various choices of parameters (w, p) on a collection of 1000 Salmonella genomes (2.7 GB).

`simplebwt` computes the BWT by first computing the Suffix Array using algorithm SACA-K [10] which is the same tool used internally by our algorithm since it is fast and uses $O(1)$ workspace. As shown in Table 5, the peak memory usage of `simplebwt` is reduced by a factor of 4 to 10 by computing the BWT from the parsing; furthermore, the total runtime is competitive with `simplebwt`. In some instances, for example the databases of 5000, 10,000 genomes, computing the BWT from the parsing achieved significant runtime reduction over `simplebwt`; with $w = 10$, $p = 100$ on these instances the runtime reduction is more than factors of 2 and 4 respectively. For our BWT computations, the peak memory usage with $w = 6$, $p = 20$ stays within a factor of roughly 2 of the original file size and is smaller than the original file size on the larger databases of 1000 genomes.

Qualitatively similar results on files from the Pizza & Chili corpus are shown in Table 6.

4.2.1 On the choice of the parameter w and p

Finally, Fig. 5 shows the peak memory usage and runtime for computing the BWT on a collection of 1000 Salmonella genomes of size 2.7 gigabytes, for all pairs of parameter choices (w, p), where $w \in \{6, 8, 10\}$ and $p \in \{50, 100, 200, 400, 800\}$. As shown in Fig. 5a, the choice $(w, p) = (10, 50)$ results in the smallest peak memory usage, while Fig. 5b shows that $(w, p) = (10, 200)$ results in the fastest runtime. In general, for either runtime or peak memory usage, the optimal choice of (w, p) differs and depends on the input. However, notice that all pairs (w, p) tested here required less than 1.1 times the input size of memory and the slowest runtime was less than twice the fastest.

5 Indexing

The BWT is widely used as part of the FM index [11], which is the heart of popular DNA sequencing read aligners such as Bowtie [15, 16], BWA [17] and SOAP 2 [18]. In these tools, rank support is added to the BWT using sampled arrays of precalculated ranks. Similarly, locate support is added using a sampled suffix array (SA). Until recently, SA samples for massive, highly repetitive datasets were much larger than the BWT, slow to calculate, or both. Gagie, Navarro, and Prezza [19] have shown

Table 4: The dictionary and parse sizes for prefixes of a database of Salmonella genomes, with three settings of the parameters w and p. Again, all sizes are reported in megabytes; percentages are the sums of the sizes of the dictionaries and parses, divided by the sizes of the uncompressed files.

number of		$w=6, p=20$			$w=8, p=50$			$w=10, p=100$		
genomes	size	dict.	parse	%	dict.	parse	%	dict.	parse	%
50	249	68	43	44	**77**	**20**	**39**	91	10	40
100	485	83	85	35	**99**	**39**	**28**	122	19	29
500	2436	273	424	29	314	194	21	**377**	**96**	**19**
1000	4861	475	847	27	541	388	19	**643**	**192**	**17**
5000	24936	2663	4334	28	2915	1987	20	**3196**	**985**	**17**
10000	49420	4190	8611	26	4652	3939	17	**5176**	**1955**	**14**

Table 5: Time (seconds) and peak memory consumption (megabytes) of BWT calculations for prefixes of a database of Salmonella genomes, for three settings of the parameters w and p and for the comparison method `simplebwt`.

number of	$w=6, p=20$		$w=8, p=50$		$w=10, p=100$		`simplebwt`	
genomes	time	peak	time	peak	time	peak	time	peak
50	71	**545**	63	642	65	782	**53**	2247
100	118	**709**	**100**	837	102	1059	103	4368
500	570	**2519**	443	2742	**402**	3304	565	21923
1000	1155	**4517**	876	4789	**776**	5659	1377	43751
5000	7412	**42067**	5436	46040	**4808**	51848	11600	224423
10000	19152	**68434**	12298	74500	**10218**	84467	43657	444780

Table 6: Time (seconds) and peak memory consumption (megabytes) of BWT calculations on various files from the Pizza & Chili repetitive corpus, for three settings of the parameters w and p and for the comparison method `simplebwt`.

	$w=6, p=20$		$w=8, p=50$		$w=10, p=100$		`simplebwt`	
file	time	peak	time	peak	time	peak	time	peak
`cere`	90	603	79	**559**	**74**	801	90	3962
`einstein.en.txt`	53	196	40	88	**35**	**53**	97	4016
`influenza`	**27**	**166**	27	284	33	435	30	1331
`kernel`	43	170	29	**143**	**25**	144	50	2216
`world_leaders`	7	**50**	7	74	**7**	98	7	405

that only the SA values at the ends of runs in the BWT need to be stored. We are currently studying how to build this sample during the process of computing the BWT from the dictionary and the parse. We show that prefix-free parsing can be incorporated into the construction of a counting-only run-length FM index (RLFM) and we measure the time and space efficiency of the RLFM construction and its "count" query in a DNA sequencing context using data from the 1000 Genomes Project. We compare the performance of the RLFM based methods to the popular Bowtie [15] read aligner.

5.1 Implementation

Constructing the counting-only RLFM requires three steps: building the BWT from the text, generating the F array, and run-length encoding the BWT. We use prefix-free parsing to build the BWT. The F array is easily built in a single pass over the text. Run-length encoding is performed using the implementation by Gagie, et. al [19], which draws upon data structures implemented in the Succinct Data Structure Library (SDSL) [20]; the concatenated run-heads of the BWT are stored in a Huffman shaped wavelet tree, and auxiliary bit-vectors are used to refer to the positions of the runs within the BWT. During index construction, all characters that are not A, C, G, T, or N are ignored.

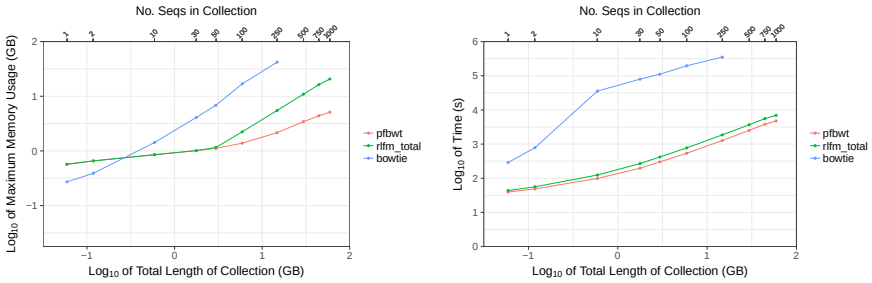
Typically, the BWT is built from a full SA, and thus a sample could be built by simply retaining parts of the initial SA. However, prefix-free parsing takes a different approach, so to build a SA sample the method would either need to be modified directly or a SA sample would have to be generated *post-hoc*. In the latter case, we can build a SA sample solely from the BWT by "back-stepping" through the BWT with LF mappings, and storing samples only at run-starts and run-ends. The main caveats to this method are that an LF mapping would have to be done for every character in the text, and that the entire BWT would need to be in memory in some form. These drawbacks would be especially noticeable for large collections. For now, we focus only on counting support, and so SA samples are excluded from these experiments except where otherwise noted.

5.2 Experiments

The indexes were built using data from the 1000 Genomes Project (1KG) [21]. Distinct versions of human chromosome 19 ("chr19") were created by using the `bcftools consensus` [22] tool to combine the chr19 sequence from the GRCh37 assembly with a single haplotype (maternal or paternal) from an individual in the 1KG project. Chr19 is 58 million DNA bases long and makes up 1.9% of the overall human genome sequence. In all, we built 10 sets of chr19s, containing 1, 2, 10, 30, 50, 100, 250, 500, and 1000 distinct versions, respectively. This allows us to measure running time, peak memory footprint and index size as a function of the collection size. Index-building and counting experiments were run on a server with Intel(R) Xeon(R) CPU E5-2650 v4 @ 2.20GHz and 512 gigabytes of RAM.

5.2.1 Index building

We compared our computational efficiency to that of Bowtie [15] v1.2.2, using the `bowtie-build` command to build Bowtie indexes for each collection. We could not



(a) Peak memory usage (b) Running time

Figure 6: RLFM indexing efficiency for successively larger collections of genetically distinct human chr19s. Results for the prefix-free parsing step ("pfbwt") are shown alongside the overall RLFM index-building ("rlfm_total") and Bowtie ("bowtie") results.

measure beyond the 250 distinct versions as Bowtie exceeded available memory. Peak memory was measured using the Unix `time -v` command, taking the value from its "Maximum resident set size (kbytes)" field. Timings were calculated and output by the programs themselves. The peak memory footprint and running time for RLFM index building for each collection are plotted in Figure 6.

Compared to Bowtie, the resources required for RLFM index-building grew much more slowly. For example, the RLFM required about 20 GB to build an index for 1,000 chr19 sequences, whereas Bowtie required twice that amount to build an index for just 250 sequences. For data points up to 50 sequences in Figure 6a, the "pfbwt" and "rlfm_total" points coincided, indicating that prefix-free parsing drove peak memory footprint for the overall index-building process. After 50 sequences, however, "pfbwt" fell below "rlfm_total" and accounted for a diminishing fraction of the footprint as the collection grew. Similarly, prefix-free parsing accounted for a diminishing fraction of total index-building time as the sequence collection grew (Figure 6b). These trends illustrate the advantage of prefix-free parsing when collections are large and repetitive.

We can extrapolate the time and memory required to index many whole human genomes. Considering chr19 accounts for 1.9% of the human genome sequence, and assuming that indexing 1,000 whole human-genome haplotypes will therefore require about 52.6 times the time and memory as indexing 1,000 chr19s, we extrapolate that indexing 1,000 human haplotypes would incur a peak memory footprint of about 1.01 TB and require about 102 hours to complete. Of course, the latter figure can be improved with parallelization.

We also measured that the index produced for the 1,000 chr19s took about 131MB of disk space. Thus, we can extrapolate that the index for 1,000 human haplotypes would take about 6.73 GB. While this figure makes us optimistic about future scaling, it is not directly comparable to the size of a Bowtie genome index since it excludes the SA samples needed for "locate" queries.

5.2.2 Count query time

We measured how the speed of the RLFM "count" operation scales with the size of the sequence collection. Given a string pattern, "count" applies the LF mapping

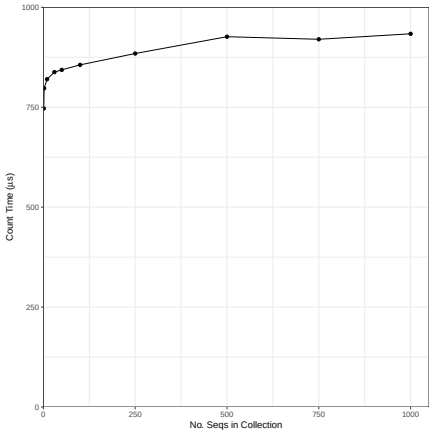


Figure 7: RLFM average "count" query time for successively larger collections of genetically distinct human chr19s.

repeatedly to obtain the range of SA positions matching the pattern. It returns the size of this range.

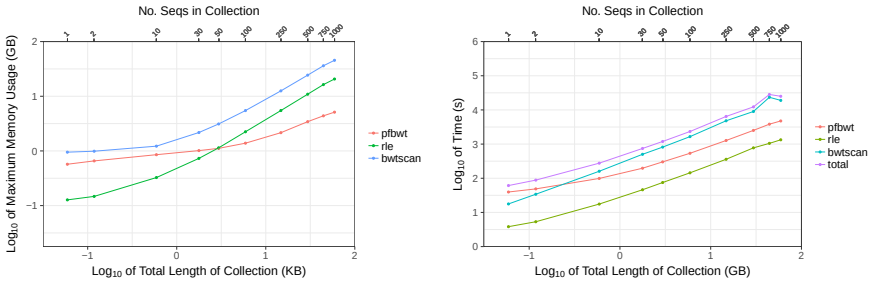
We measured average "count" time by conducting a simple simulation of DNA-sequencing-like data. First we took the first chr19 version and extracted and saved 100,000 random substrings of length 100. That chr19 was included in all the successive collections, so these substrings are all guaranteed to occur at least once regardless of which collection we are querying.

We then queried each of the collections with the 100,000 substrings and divided the running time by 100,000 to obtain the average "count" query time. Query time increases as the collection grows (Figure 7) but does so slowly, increasing from 750 microseconds for 1 sequence to 933 microseconds for 1,000 sequences.

5.2.3 Including the SA sample

Though no SA sample was built for the experiments described so far, such a sample is needed for "locate" queries that return the text offset corresponding to a BWT element. A SA sample can be obtained using the "back-stepping" method described above. We implemented a preliminary version of this method to examine whether prefix-free parsing is a bottleneck in that scenario. Here, index building consists of three steps: (1) building the BWT using prefix-free parsing ("pfbwt"), (2) back-stepping to create the SA sample and auxiliary structures ("bwtscan"), and (3) run-length encoding the BWT ("rle"). We built RLFM indexes for the same chr19 collections as above, measuring running time and peak memory footprint for each of the three index-building step independently (Figure 8).

The "bwtscan" step consistently drives peak memory footprint, since it stores the entire BWT in memory as a Huffman shaped wavelet tree [20]. The "pfbwt" step has a substantially smaller footprint and used the least memory of all the steps for collections larger than 50 sequences. "pfbwt" is the slowest step for small collections, but "bwtscan" becomes the slowest for 10 or more sequences. We conclude that as the collection of sequences grows, the prefix-free parsing method contributes proportionally less to peak memory footprint and running time, and presents no bottlenecks for large collections.



(a) Peak memory usage (b) Running time

Figure 8: Computational efficiency of the three stages of index building when SA sampling is included. Results are shown for the prefix-free parsing ("pfbwt"), back-stepping ("bwtscan") and run-length encoding ("rle") steps. "total" is the sum of the three steps.

6 Conclusions

We have described how prefix-free parsing can be used as preprocessing step to enable compression-aware computation of BWTs of large genomic databases. Our results demonstrate that the dictionaries and parses are often significantly smaller than the original input, and so may fit in a reasonable internal memory even when T is very large. We show how the BWT can be constructed from a dictionary and parse alone. Lastly, we give experiments demonstrating how the run length compressed FM-index can be constructed from the parse and dictionary. The efficiency of this construction shows that this method opens up the possibility to cosntructing the BWT for datasets that are terabytes in size; such as GenomeTrakr [4] and MetaSub [23].

Finally, we note that when downloading large datasets, prefix-free parsing can avoid storing the whole uncompressed dataset in memory or on disk. Suppose we run the parser on the dataset as it is downloaded, either as a stream or in chunks. We have to keep the dictionary in memory for parsing but we can write the parse to disk as we go, and in any case we can use less total space than the dataset itself. Ideally, the parsing could even be done server-side to reduce transmission time and/or bandwidth — which we leave for future implementation and experimentation.

Acknowledgements
The authors thank Risto Järvinen for the insight they gained from his project on `rsync` in the Data Compression course at Aalto University.

Authors' contributions
TG and GM conceptualized the idea and developed the algorithmic contributions of this work. AK and GM implemented the construction of the prefix-free parsing and conducted all experiments. CB and TG assisted and oversaw the experiments and implementation. TM and BL implemented and tested the construction of the run-length compressed FM-index. All authors contributed to the writing of this manuscript.

Availability
Prefix-free parsing and all accompanied documents are available at `https://gitlab.com/manzai/Big-BWT`.

Competing interests
The authors declare that they have no competing interests.

Funding
CB and AK were supported by National Science Foundation (IIS 1618814). AK was also supported by a post-doctoral fellowship from the University of Florida Informatics Institute. TG was partially supported by FONDECYT (1171058). GM was partially supported by PRIN grant (201534HNXC).

Author details
[1]CISE, University of Florida, Gainesville, FL USA. [2]EIT, Diego Portales University, Santiago, Chile. [3]CeBiB, Santiago, Chile. [4]Informatics Institute, Gainesville, FL USA. [5]Johns Hopkins University, Baltimore, MD, USA. [6]University of Eastern Piedmont, Alessandria, Italy. [7]IIT, CNR, Pisa, Italy.

References

1. The 1000 Genomes Project Consortium: A global reference for human genetic variation. Nature **526**, 68–74 (2015)
2. Turnbull, C., *et al.*: The 100,000 genomes project: bringing whole genome sequencing to the nhs. British Medical Journal **361**, 1687 (2018)
3. Carleton, H.A., Gerner-Smidt, P.: Whole-genome sequencing is taking over foodborne disease surveillance. Microbe **11**, 311–317 (2016)
4. Stevens, E.L., Timme, R., Brown, E.W., Allard, M.W., Strain, E., Bunning, K., Musser, S.: The public health impact of a publically available, environmental database of microbial genomes. Frontiers in Microbiology **8**, 808 (2017)
5. Burrows, M., Wheeler, D.J.: A block-sorting lossless compression algorithm. Technical report, Digital Equipment Corporation (1994)
6. Sirén, J.: Burrows-Wheeler transform for terabases. In: Proccedings of the 2016 Data Compression Conference (DCC), pp. 211–220 (2016)
7. Ferragina, P., Gagie, T., Manzini, G.: Lightweight data indexing and compression in external memory. Algorithmica **63**(3), 707–730 (2012)
8. Policriti, A., Prezza, N.: From LZ77 to the run-length encoded burrows-wheeler transform, and back. In: Proceedings of the 28th Symposium on Combinatorial Pattern Matching (CPM), pp. 17–11710 (2017)
9. https://rsync.samba.org
10. Nong, G.: Practical linear-time $O(1)$-workspace suffix sorting for constant alphabets. ACM Trans. Inf. Syst. **31**(3), 15 (2013)
11. Ferragina, P., Manzini, G.: Indexing compressed text. Journal of the ACM (JACM) **52**(4), 552–581 (2005)
12. Louza, F.A., Gog, S., Telles, G.P.: Inducing enhanced suffix arrays for string collections. Theor. Comput. Sci. **678**, 22–39 (2017)
13. http://pizzachili.dcc.uchile.cl/repcorpus.html
14. Stevens, E.L., Timme, R., Brown, E.W., Allard, M.W., Strain, E., Bunning, K., Musser, S.: The public health impact of a publically available, environmental database of microbial genomes. Frontiers in Microbiology **8**, 808 (2017)
15. Langmead, B., Trapnell, C., Pop, M., Salzberg, S.L.: Ultrafast and memory-efficient alignment of short DNA sequences to the human genome. Genome biology **10**(3), 25 (2009)
16. Langmead, B., Salzberg, S.L.: Fast gapped-read alignment with Bowtie 2. Nature Methods **9**(4), 357–360 (2012). doi:10.1038/nmeth.1923
17. Li, H., Durbin, R.: Fast and accurate long-read alignment with burrows–wheeler transform. Bioinformatics **26**(5), 589–595 (2010)
18. Li, R., Yu, C., Li, Y., Lam, T.-W., Yiu, S.-M., Kristiansen, K., Wang, J.: Soap2: an improved ultrafast tool for short read alignment. Bioinformatics **25**(15), 1966–1967 (2009)
19. Gagie, T., Navarro, G., Prezza, N.: Optimal-time text indexing in bwt-runs bounded space. In: Proceedings of the 29th Symposium on Discrete Algorithms (SODA), pp. 1459–1477 (2018)
20. Gog, S., Beller, T., Moffat, A., Petri, M.: From theory to practice: Plug and play with succinct data structures. In: 13th International Symposium on Experimental Algorithms, (SEA 2014), pp. 326–337 (2014)
21. Consortium, T..G.P.: A global reference for human genetic variation. Nature **526**(7571), 68–74 (2015). doi:10.1038/nature15393. Accessed 2018-09-28
22. Narasimhan, V., Danecek, P., Scally, A., Xue, Y., Tyler-Smith, C., Durbin, R.: BCFtools/RoH: a hidden Markov model approach for detecting autozygosity from next-generation sequencing data. Bioinformatics **32**(11), 1749–1751 (2016)
23. MetaSUB International Consortium: The Metagenomics and Metadesign of the Subways and Urban Biomes (MetaSUB) International Consortium inaugural meeting report. Microbiome **4**(1), 24 (2016)