Prefix-Free Parsing for Building Big BWTs

Christina Boucher
CISE, University of Florida
Gainesville, FL, USA

Travis Gagie[1]
EIT, Diego Portales University
Santiago, Chile

CeBiB
Santiago, Chile

Alan Kuhnle
CISE, University of Florida
Gainesville, FL, USA

Giovanni Manzini
University of Eastern Piedmont
Alessandria, Italy

IIT, CNR
Pisa, Italy

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 and enable 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. Therefore, prefix-free parsing eases BWT construction, which is pertinent to many bioinformatic applications.

2012 ACM Subject Classification E.4 Coding and Information Theory; F.2.2 Nonnumerical algorithms and problems

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

Digital Object Identifier 10.4230/LIPIcs.WABI.2018.23

Acknowledgements The authors would like to thank Risto Järvinen for explaining `rsync`'s content-slicing.

[1] Partly funded by FONDECYT grant 1171058.

The 18th Workshop on Algorithms in Bioinformatics (WABI)).
Editors: Laxmi Parida and Esko Ukkonen ; Article No. 23; pp. 23:1–23:12

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

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 [21] was announced in 2008 and completed in 2015, and now the 100,000 Genomes Project is well under way [22]. 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 just are as equally challenging from a size perspective as 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) [5]. This led to the existence of GenomeTrakr, which is a large public effort to use genome sequencing for surveillance and detection of outbreaks of foodborne illnesses. Currently, the GenomeTrakr effort includes over 100,000 samples, spanning several species available through this initiative—a number that continues to rise as datasets are continually added [19]. 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 also 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) [4] and there have been a number of papers about building BWTs for genomic databases; see, e.g., [18] 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 [17] recently 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. Specifically, the parsing algorithm itself is straightforward and it can either be made to work using a single pass over the data on disk or it can be parallelized. Once we have the results of the parsing, which are a dictionary and a parse, building the BWT out of them is more involved, but 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 [1] 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 [15], while requiring roughly the same time on large sets of salmonella genomes obtained from GenomeTrakr.

In Section 2, 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 3 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. We conclude in Section 4 and discuss directions for future work. Prefix-free parsing and all accompanied documents are available at `https://gitlab.com/manzai/Big-BWT`.

2 Theory

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 $d \in D$,

- 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'.

We let S be the set of suffixes of length greater than w of elements of D.

Given T and a way to recognize strings in E, we can build D iteratively by simulating scanning $\# T \$^w$ to find occurrences of elements of E', 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 $\# T \$^w$). We then build S from D and sort it.

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 it follows that we get

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

and, identifying elements of D by their lexicographic ranks, $P = 0, 1, 3, 1, 4, 2$.

▶ **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. ◀

Let f be the function that maps 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 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 infomation 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. 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` $\in S$

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

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.

If we want, 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

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$

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 and then from them, 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 8.** *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.*

3 Practice

We have implemented our BWT construction in order to test our conjectures that, first, 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 both be small in comparison to the uncompressed size of the database; second, computing the dictionary and the parse first and then computing the BWT from them leads to an overall speedup and reduction in memory usage. In this section we describe our implementation and then report our experimental results.

3.1 Implementation

As described in Sections 1 and 2, 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 NULL character to the first phrase and append w copies of NULL 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 NULL. We note that we prepend and append copies of the same NULL character; although using different characters simplifies the proofs in Section 2, 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 [16] which, among the linear time algorithms, is the one using the smallest workspace. 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 words in lexicographic order, and, for each word d_i, we write the list of BWT positions where d_i appears. We call this the inverted list for word d_i. Since the size of the inverted list of each word 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.

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

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 2 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 [14] which computes the SA and LCP array for the set of dictionary phrases. We then proceed as in Section 2. 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 word d we write the characters retrieved from W from the positions obtained from d's inverted list.

3.2 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 [2], with three settings of the parameters w and p. We note that `cere` contains long runs of `Ns` 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 `Ns` 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 [20]. 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

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

parsing for the database of Salmonella genomes. As a baseline for comparison, `simplebwt` computes the BWT by first computing the Suffix Array using algorithm SACA-K [16]; SACA-K is a linear time algorithm that uses $O(1)$ workspace and is fast in practice. 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 database of 5000 genomes, computing the BWT from the parsing achieved significant runtime reduction over `simplebwt`; with $w = 10, p = 100$ on this instance, the runtime reduction is more than a factor of 2. 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 orginal file size on the larger databases of 1000 genomes. For the database of 5000 genomes, the most expensive steps were parsing and computing the missing characters — about 23% of the total BWT — to fill in the subsequence. Extrapolating to 10000 genomes, it seems it would take at least about three hours and 100 GB to build the BWT for each of our three settings of parameters for prefix-free parsing, and at least about six hours and 450 GB to build it with `simplebwt`; due to time and memory constraints, we leave this as future work.

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

4 Conclusion and Future Work

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

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 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. Finally, we show how the BWT can be constructed from a dictionary and parse alone. In the full version of this paper we will investigate using compressed suffix arrays during the construction of the BWT, instead of suffix arrays, which should reduce our memory usage at the cost of increasing the running time by approximately a factor logarithmic in the size of the input.

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.

A natural extension of our method is to consider efficient parallelization of the parsing. With k processors, we could divide the input string into k equal blocks, with each consecutive pair of blocks overlapping by w characters; we scan each block with a processor, to find the locations of the substrings of length w with Karp-Rabin hashes congruent to 0 modulo p; and then we scan the input with the k processors in parallel to compute the dictionary and parse, starting at roughly evenly-spaced locations of such substrings. Alternatively, our approach can be viewed as a modified Schindler Transform [3] and since previous authors [6] have shown how the Schindler Transform benefits from GPU parallelization, we believe that GPU-based parallelization could be both easy and effective.

Perhaps the main use of BWTs is in FM-indexes [8], which are at the heart of the most popular DNA aligners, including Bowtie [10, 11], BWA [12] and SOAP 2 [13]. With only rank support over a BWT, we can count how many occurrences of a given pattern there are in the text, but we cannot tell where they are without using a suffix-array sample. Until recently, suffix-array samples for massive, highly repetitive datasets were usually either much larger than the datasets BWTs, or very slow. Gagie, Navarro and Prezza [9] have now shown we need only store suffix array values at the ends of runs in the BWT, however, and we conjecture that we can build this sample while computing the BWT from the dictionary and the parse. Indeed, we were initially motivated to study new approaches to BWT construction because without them, Gagie et al.'s result may never realize its full potential.

References

1 https://rsync.samba.org.

2 http://pizzachili.dcc.uchile.cl/repcorpus.html.

3 http://www.compressconsult.com/st.

4 Michael Burrows and David J. Wheeler. A block-sorting lossless compression algorithm. Technical report, Digital Equipment Corporation, 1994.

5 H.A. Carleton and P. Gerner-Smidt. Whole-genome sequencing is taking over foodborne disease surveillance. *Microbe*, 11:311–317, 2016.

6 Chia-Hua Chang, Min-Te Chou, Yi-Chung Wu, Ting-Wei Hong, Yun-Lung Li, Chia-Hsiang Yang, and Jui-Hung Hung. sBWT: memory efficient implementation of the hardware-acceleration-friendly Schindler transform for the fast biological sequence mapping. *Bioinformatics*, 32(22):3498–3500, 2016.

7 Paolo Ferragina, Travis Gagie, and Giovanni Manzini. Lightweight data indexing and compression in external memory. *Algorithmica*, 63(3):707–730, 2012.

8 Paolo Ferragina and Giovanni Manzini. Indexing compressed text. *Journal of the ACM (JACM)*, 52(4):552–581, 2005.

9 Travis Gagie, Gonzalo Navarro, and Nicola Prezza. Optimal-time text indexing in bwt-runs bounded space. In *Proceedings of the 29th Symposium on Discrete Algorithms (SODA)*, pages 1459–1477, 2018.

10 Ben Langmead, Cole Trapnell, Mihai Pop, and Steven L Salzberg. Ultrafast and memory-efficient alignment of short dna sequences to the human genome. *Genome biology*, 10(3):R25, 2009.

11 Ben Langmead, Cole Trapnell, Mihai Pop, and Steven L Salzberg. Ultrafast and memory-efficient alignment of short dna sequences to the human genome. *Genome biology*, 10(3):R25, 2009.

12 Heng Li and Richard Durbin. Fast and accurate long-read alignment with burrows–wheeler transform. *Bioinformatics*, 26(5):589–595, 2010.

13 Ruiqiang Li, Chang Yu, Yingrui Li, Tak-Wah Lam, Siu-Ming Yiu, Karsten Kristiansen, and Jun Wang. Soap2: an improved ultrafast tool for short read alignment. *Bioinformatics*, 25(15):1966–1967, 2009.

14 Felipe Alves Louza, Simon Gog, and Guilherme P. Telles. Inducing enhanced suffix arrays for string collections. *Theor. Comput. Sci.*, 678:22–39, 2017.

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

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

17 Alberto Policriti and Nicola Prezza. From LZ77 to the run-length encoded burrows-wheeler transform, and back. In *Proceedings of the 28th Symposium on Combinatorial Pattern Matching (CPM)*, pages 17:1–17:10, 2017.

18 Jouni Sirén. Burrows-Wheeler transform for terabases. In *Proccedings of the 2016 Data Compression Conference (DCC)*, pages 211–220, 2016.

19 E.L. Stevens, R. Timme, E.W. Brown, M.W. Allard, E. Strain, K. Bunning, and S. Musser. The public health impact of a publically available, environmental database of microbial genomes. *Frontiers in Microbiology*, 8:808, 2017.

20 Eric L. Stevens, Ruth Timme, Eric W. Brown, Marc W. Allard, Errol Strain, Kelly Bunning, and Steven Musser. The public health impact of a publically available, environmental database of microbial genomes. *Frontiers in Microbiology*, 8:808, 2017.

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

22 C. Turnbull et al. The 100,000 genomes project: bringing whole genome sequencing to the nhs. *British Medical Journal*, 361:k1687, 2018.