Compressing and Indexing Aligned Readsets

Travis Gagie, Garance Gourdel, Giovanni Manzini

Compressing and Indexing Aligned Readsets

Travis Gagie
Dalhousie University, Halifax, Canada

Garance Gourdel
IRISA – Inria Rennes – Université Rennes 1 – ENS, France

Giovanni Manzini
University of Pisa, Italy

Abstract

Compressed full-text indexes are one of the main success stories of bioinformatics data structures but even they struggle to handle some DNA readsets. This may seem surprising since, at least when dealing with short reads from the same individual, the readset will be highly repetitive and, thus, highly compressible. If we are not careful, however, this advantage can be more than offset by two disadvantages: first, since most base pairs are included in at least tens reads each, the uncompressed readset is likely to be at least an order of magnitude larger than the individual's uncompressed genome; second, these indexes usually pay some space overhead for each string they store, and the total overhead can be substantial when dealing with millions of reads.

The most successful compressed full-text indexes for readsets so far are based on the Extended Burrows-Wheeler Transform (EBWT) and use a sorting heuristic to try to reduce the space overhead per read, but they still treat the reads as separate strings and thus may not take full advantage of the readset's structure. For example, if we have already assembled an individual's genome from the readset, then we can usually use it to compress the readset well: e.g., we store the gap-coded list of reads' starting positions; we store the list of their lengths, which is often highly compressible; and we store information about the sequencing errors, which are rare with short reads. There is nowhere, however, where we can plug an assembled genome into the EBWT.

In this paper we show how to use one or more assembled or partially assembled genome as the basis for a compressed full-text index of its readset. Specifically, we build a labelled tree by taking the assembled genome as a trunk and grafting onto it the reads that align to it, at the starting positions of their alignments. Next, we compute the eXtended Burrows-Wheeler Transform (XBWT) of the resulting labelled tree and build a compressed full-text index on that. Although this index can occasionally return false positives, it is usually much more compact than the alternatives. Following the established practice for datasets with many repetitions, we compare different full-text indices by looking at the number of runs in the transformed strings. For a human Chr19 readset our preliminary experiments show that eliminating separators characters from the EBWT reduces the number of runs by 19%, from 220 million to 178 million, and using the XBWT reduces it by a further 15%, to 150 million.

2012 ACM Subject Classification Theory of computation → Data compression

Keywords and phrases data compression, compact data structures, FM-index, Burrows-Wheeler Transform, EBWT, XBWT, DNA reads

Digital Object Identifier 10.4230/LIPIcs.WABI.2021.13

Supplementary Material *Software (Source Code)*: `https://github.com/fnareoh/Big_XBWT` archived at `swh:1:dir:ec67fec8b70e6b837a1b497c7fc07cb5d179c512`

Funding *Travis Gagie*: Funded by NSERC Discovery Grant RGPIN-07185-2020, NIH R01HG011392 and NSF IIBR 2029552.
Garance Gourdel: Partially funded by the grant ANR-20-CE48-0001 from the French National Research Agency (ANR).
Giovanni Manzini: Supported by the Italian MIUR PRIN project 2017WR7SHH.

Acknowledgements Many thanks to Jarno Alanko and Uwe Baier for their XBWT-construction software, and to Diego Díaz, Richard Durbin, Filippo Geraci, Giuseppe Italiano, Ben Langmead, Gonzalo Navarro, Pierre Peterlongo, Nicola Prezza, Giovanna Rosone, Jared Simpson, Jouni Sirén and Jan Studený for helpful discussions.

1 Introduction

The FM-index [23] is an important data structure in both combinatorial pattern matching and bioinformatics. Its most important application so far has been in standard short-read aligners – Bowtie [39, 38] and BWA [41] have together over 70 thousand citations and are used every day in clinics and research labs worldwide – but it has myriad other uses and more are still being discovered. Just within computational genomics, FM-indexes have been generalized from single strings to collections of strings for tools such as BEETL [15], RopeBWT [40] and Spring [11], to de Bruijn graphs for tools such as BOSS [8], VARI [48] and Rainbowfish [2], and to graphs for tools such as vg [27]. Recent breakthroughs [25] mean we can now scale FM-indexes to massive but highly repetitive pan-genomic datasets for a new generation of tools [36].

As genomic datasets grow exponentially (from the Human Genome Project to the 1000 Genomes Project and the 100K Genomes Project) and standards for sequencing coverage increase (from less than 10x a few years ago to 30x and 50x now and over 100x for some applications), an obvious question is whether and how the recent breakthroughs in FM-indexing of repetitive datasets can be turned into comparable advances in indexing readsets, so more researchers can efficiently mine them for biomedical insights. For example, extrapolating from previous experiments [36], it should be possible to index both haplotypes from 2705 individuals in less than 100 GB of RAM. In contrast, the readset from the final phase of the 1000 Genomes Project consisted of reads from 2705 individuals and was released as a 464 GB Burrows-Wheeler Transform (BWT) [17], which is beyond the resources of most labs to process. This almost five-fold increase (from 100 to 464 GB) seems reasonable, given the range of lengths and the error rate of short-read sequencing technologies, but those reads were trimmed and error-corrected before their BWT was computed, making that increase harder to justify and thus a target for improvement. Although experimenting with that particular readset is beyond the scope of this paper, since it occupies 87 TB uncompressed, we expect the insights and techniques we develop here will eventually be useful in software able to handle efficiently inputs of that scale.

Recent results on FM-indexing repetitive datasets [25] have shown that the index performance depends on the number of runs in the transformed sequence, where a run is a maximal non-empty unary substring. For example, if the BWT of a dataset of (uncompressed) size n has r runs, we can design an FM-index of size $O(r \log \log n)$ supporting the count and locate operations in optimal linear time. Hence, if a BWT variant produces a transformed string with a smaller number of runs, the resulting index will be smaller and equally fast. The naïve approach to FM-indexing readsets is to concatenate the reads with copies of a separator character between them, and FM-index the resulting single string. However, computing the BWT of such a long string is a challenge and each separator character causes several runs in that BWT. The most competitive indexes for readsets are based on Mantaci et al.'s [46] Extended Burrows Wheeler Transform, which is also easier to build for readsets. The first index for readsets based on the EBWT was BEETL [15], followed by RopeBWT [40]; recently the EBWT has been used also by the Spring compressor [11] specialized for FASTQ reads. BEETL and RopeBWT use explicit separator characters but such characters could be replaced by bitvectors marking positions at the ends of reads.

BEETL and RopeBWT use a heuristic to reduce the number of runs in the EBWT: they conceptually put the separator characters at the ends of reads into the co-lexicographic order (lexicographic order on the reverse string, also referred to as reverse lexicographic order) of the reads, so that the final characters or reads with similar suffixes are grouped together in

the EBWT. This often works surprisingly well but in the worst case it cannot make up for the lack of context for sorting those characters into their places in the EBWT. Our proposal in this paper is to graft the reads onto their assembled genome, or a reference genome to which they align well, and index the resulting labelled tree with Ferragina et al.'s [22] XBWT. To this end we assume that we know how the reads align to the assembled/reference genome: this is not an unreasonable assumption since alignment is the initial step of any readset analysis.

In order to implement our idea we have to overcome a significant hurdle: as the coverage increases so does the amount of raw data produced by a single NGS experiment. Although the high coverage implies that the data is highly compressible, the actual compression process, ie the construction and the compression of the XBWT, must be done partially in externally memory since the input will be usually much larger than the available RAM. Another contribution of the paper is therefore the adaptation of the prefix-free parsing (PFP) technique [7] to the construction of the XBWT. PFP has been proposed for the construction of BWTs of collections of similar genomes: the initial parsing phase is able to compress the input maintaining enough information to compute the BWT working on the compressed representation. In this paper we adapt PFP to readsets, taking care also of the "grafting" of the single reads to the reference/assembled genome. Given a pattern P, our index could answer $count(P)$ and $locate(P)$ queries which report respectively the number of positions where P occurs and the list of positions where P occurs. The main drawback to our index, apart from taking one or more assembled or partially assembled genomes as a base, is that it can return a false-positive in the *count* operation when an occurrence of a pattern starts in the trunk of an alignment tree and ends in a branch. In other words, the index can report a match that is not completely contained within a read but would be if we padded the read on the left with enough characters copied from just before where it aligns. In a locate operation false-positives could be identified, but this operation is much slower. Even this is not entirely bad, however, and it is conceivable this bug could sometimes be a feature. The analysis of those false positive and the size of the bit vectors marking the end of reads is left as future work.

The rest of the paper is organized as follows. In Section 2 we first describe the BWT and FM-indexes, then the EBWT and XBWT and the concept of Wheeler graph that unifies them. In Section 3 we introduce our idea for indexing aligned readsets with the XBWT and we prove some theoretical results supporting it. In Section 4 we describe how we adapt PFP to indexing readsets, which allows us to experiment with larger files than would otherwise be possible with reasonable resources. In Section 5 we present our experimental results showing that applying the XBWT to index readsets works well in practice as well as in theory. Finally, we outline in Section 6 how our study of storing reads with the XBWT may improve the space usage of the hybrid index [20, 21, 26].

2 Concepts

For a better understanding of the problem context, we give a succinct description of the second generation sequencing technique. Most publicly available readsets are from Illumina sequencers [35] which rely on sequencing by synthesis. For this process, millions or billions of single-stranded snippets of DNA called templates are deposited onto a slide and amplified into clusters of clones. In each sequencing cycle we learn one base of each template: we add DNA polymerase and specially terminated bases; the polymerase attaches a terminated base to each strand, complementary to the next base in the strand; we shine a light on the slide

and the terminated bases glow various colours; we take a photo and note the colour of each cluster; and finally, we treat the slide to remove the terminators. Sometimes, however, one of the added bases is not correctly terminated, so the polymerase attaches first it and then another base to a strand in some cluster; that strand is then out of step with the rest of the cluster, and the cluster will have a mix of colours in the photos for subsequent sequencing cycles. As we go through more and more sequencing cycles, more strands tend to fall out of step, resulting in less reliable results. (For futher discussion we refer the reader to, e.g., Langmead's lecture on this topic [37].) This tendency means sequencing by synthesis has an asymmetric error profile, with errors more likely towards the ends of the reads. It follows that sequencing errors tend to be near the end of the reads: our index is designed to take advantage of this feature (see Theorem 2).

2.1 BWT and FM-index

The Burrows-Wheeler Transform (BWT) [10] of a string S is a permutation of the characters in S into the lexicographic order of the suffixes that immediately follow them, considering S to be cyclic. For example, as shown on the left in Figure 1, the BWT of `GATTAGATACAT$` is `TTTCGGAA$AATA`, assuming `$` is a special end-of-string symbol lexicographically smaller than all other characters. Because the BWT groups together characters that precede similar suffixes, it tends to convert global repetitiveness into local homogeneity: e.g., for any string α, the BWT of α^t consists of $|\alpha|$ unary substrings of length t each; even the BWT in our example has length 13 but consists of only 8 maximal unary substrings (called runs). This property led Burrows and Wheeler to propose the BWT as a pre-processing step for data compression and Seward [55] used it as the basis for the popular `bzip2` compression program.

The BWT is also the basis for the FM-index [23], one of the first and most popular compressed indexes, which is essentially a rank data structure over the BWT combined with a suffix-array sample. The FM-index is an important data structure in combinatorial pattern matching and bioinformatics, and is itself the basis for popular tools such as Bowtie [39, 38] and BWA [41] that align DNA reads to reference genomes. We refer the reader to Navarro's [49] and Mäkinen et al.'s [45] textbooks for detailed discussions of how FM-indexes are implemented and used for read alignment.

2.2 EBWT

Although alignment against one or more reference genomes remains a key task in bioinformatics, there is growing interest in compressed indexing of sets of reads [17, 34]. The FM-index plays a central role here too: Mantaci et al. [47] generalized the BWT to the Extended BWT (EBWT), which applies to collections of strings, and then Cox et al. [5, 13, 30] used an FM-index built on the EBWT in their index BEETL for readsets. The same construction was also used in subsequent indexes for readsets, such as RopeBWT [40] and Spring [11].

The EBWT of a collection of strings is a permutation of the characters in those strings into the lexicographic order of the suffixes that immediately follow them, considering each string to be cyclic. For example, as shown on the right in Figure 1, the EBWT of `GATTA$`, `TTAGA$`, `TAGATA$`, `GATAC$` and `ATACAT$` is `TCAAATTGTTTTCGG$GAAAA$$ATAAAT$A$`. When we see the BWT and EBWT as permutations of characters, the BWT of a single string has a single cycle, whereas the EBWT of a collection of strings has a cycle for each string. This means it is easier to build the EBWT and update it when a string is added or deleted, than to build and update the BWT of the concatenation of the collection with the strings separated by copies of a special character. We refer the reader to Egidi et al.'s [18, 19] and Díaz-Domínguez and Navarro's [16] recent papers for descriptions of efficient construction and updating algorithms.

	F	L
0	$GATTAGATACAT	
1	ACAT$GATTAGAT	
2	AGATACAT$GATT	
3	AT$GATTAGATAC	
4	ATACAT$GATTAG	
5	ATTAGATACAT$G	
6	CAT$GATTAGATA	
7	GATACAT$GATTA	
8	GATTAGATACAT$	
9	T$GATTAGATACA	
10	TACAT$GATTAGA	
11	TAGATACAT$GAT	
12	TTAGATACAT$GA	

	F	L		F	L
0	$ATACAT		16	ATTA$	G
1	$GATA	C	17	C$GAT	A
2	$GATT	A	18	CAT$ATA	
3	$TAGATA		19	GA$TT	A
4	$TTAG	A	20	GATA$TA	
5	A$GAT	T	21	GATAC	$
6	A$TAGAT		22	GATTA	$
7	A$TTA	G	23	T$ATACA	
8	AC$GA	T	24	TA$GA	T
9	ACAT$AT		25	TA$TAGA	
10	AGA$T	T	26	TAC$G	A
11	AGATA$T		27	TACAT$A	
12	AT$ATAC		28	TAGA$	T
13	ATA$TAG		29	TAGATA$	
14	ATAC$	G	30	TTA$G	A
15	ATACAT$		31	TTAGA	$

Figure 1 The matrices whose rows are the lexicographically sorted rotations of `GATTAGATACAT$` **(left)** and of `GATTA$`, `TTAGA$`, `TAGATA$`, `GATAC$` and `ATACAT$` **(right)**. The BWT and EBWT are `TTTCGGAA$AATA` and `TCAAATTGTTTTCGG$GAAAA$$ATAAAT$A$` with 8 and 19 runs, respectively.

Despite its benefits, the EBWT sometimes does not take full advantage of its input's compressibility. In our example, as Figure 1 shows, even though all the strings in the collection are substrings of `GATTAGATACAT$` with copies of `$` appended to them, their EBWT has more than twice as many runs as its BWT. As a heuristic for reducing the number of runs, and thus reducing BEETL's space usage, Cox et al. suggested considering the lexicographic order of the copies of `$` to be the strings' co-lexicographic order. This does not help in cases such as our example, however, for which the EBWT still has 19 runs even with that ordering. Bentley et al. [6] recently gave a linear-time algorithm to find the ordering of the copies of `$` that minimizes the number of runs, but it has not been implemented and it is unclear whether it is practical for large readsets.

Another way to potentially reduce the number of runs is to remove the copies of `$` entirely, and store an auxiliary ternary vector marking which characters in the EBWT are the first and last characters in the strings. If there are t strings in the collection with total length n, then storing this vector takes $O(t \log(n/t) + t)$ bits (even if some of the strings are empty or consist of only one character). As shown in Figure 2, the EBWT becomes `TTTTTTGTCGGGAACAAAAAATTAAAA`, with only 10 runs. The idea of replacing `$`'s with an auxiliary vector is relatively new since it originates from seeing the EBWT as a special case of Wheeler graphs [24] which are described in the next section.

2.3 Wheeler Graphs and XBWT

Wheeler graphs were introduced by Gagie, Manzini and Sirén [24] as a unifying framework for several extensions of the BWT, including the EBWT, Ferragina et al.'s [22] eXtended BWT (XBWT) for labelled trees, Bowe, et al's. [9] index (BOSS) for de Bruijn graphs, and Sirén et al.'s [56] Generalized Compressed Suffix Array (GCSA) for variation graphs. A directed edge-labelled graph is a Wheeler graph if there exists a total order on the vertices such that

- vertices with in-degree 0 are earliest in the order;
- if (u, v) is labelled a and (u', v') is labelled b with $a \prec b$, then $v < v'$;
- if (u, v) and (u', v') are both labelled a and $u < u'$ then $v \leq v'$.

		F L			F L
0	0	ACATAT	14	+	GATA C
1	0	ACGA T	15	0	GATATA
2	0	AGATAT	16	0	GATT A
3	−	AGAT T	17	+	GATT A
4	0	AGAT T	18	0	TACATA
5	+	ATACAT	19	0	TACG A
6	0	ATAC G	20	+	TAGATA
7	−	ATAGAT	21	0	TAGA T
8	0	ATATAC	22	0	TAGA T
9	0	ATATAG	23	−	TATACA
10	0	ATTA G	24	0	TATAGA
11	−	ATTA G	25	0	TTAG A
12	0	CATATA	26	+	TTAG A
13	−	CGAT A			

Figure 2 The matrix whose rows are the lexicographically sorted rotations of GATTA, TTAGA, TAGATA, GATAC and ATACAT. The EBWT is TTTTTTGTCGGGAACAAAAAATTAAAA with 10 runs.

Figure 3 shows an example of a Wheeler graph with a valid order on the vertices. The ordering is obtained by lexicographically sorting the strings spelling the labels in the upward path from each vertex to the root where the ties are broken deterministically (following an arbitrary order on the branches). For example, vertex 0 has upward path ε, vertex 3 has upward path AG, vertex 30 has upward path TAG and so on. Notice that for directed acyclic graphs such as trees, such order on the vertices can be computed quickly with an adaptation of the doubling algorithm [33].

Once we have a valid order, the standard representation of a Wheeler graph is defined considering the vertices in that order and listing the labels on the outgoing edges of each vertex. In addition, for each vertex we represent its out-degree and in-degree in unary thus obtaining two additional binary arrays. For example, for the graph in Figure 3 the first five vertices have outgoing edges labelled GG T T T TT, so the label array starts with GGTTTTT$\cdots$ and the out-degree bit-array starts with 001010101001$\cdots$. This simple representation, combined with rank and select primitives, supports efficient search and navigation operations on Wheeler graphs. We refer the reader to Prezza's [53] recent survey for a discussion of Wheeler graphs and related results.

Note that the graph in Figure 3 is a labelled tree: indeed its Wheeler Graph representation is equivalent to the output of the XBWT [22] applied to the same tree (details in the full paper). For clarity of presentation in the following we will still refer to the EBWT and XBWT even if they are both special cases of Wheeler graphs.

3 Our contribution

Figure 3 can be seen as a representation of a "genome" GATTAGATACAT and of five "reads" GATTA, TTAGA, TAGATA, GATAC and ATACAT extracted, without errors, from it. Starting with the vertex with rank 28, corresponding to the last symbol of the "genome", and navigating the tree we are able to recover all the individual strings. Notice however, that the XBWT has only 7 runs while the BWT of the "genome" and the EBWT of the "reads" in Figure 1 have 8 and 19 runs, respectively. The EBWT without $ of the reads alone in Figure 2 has 10

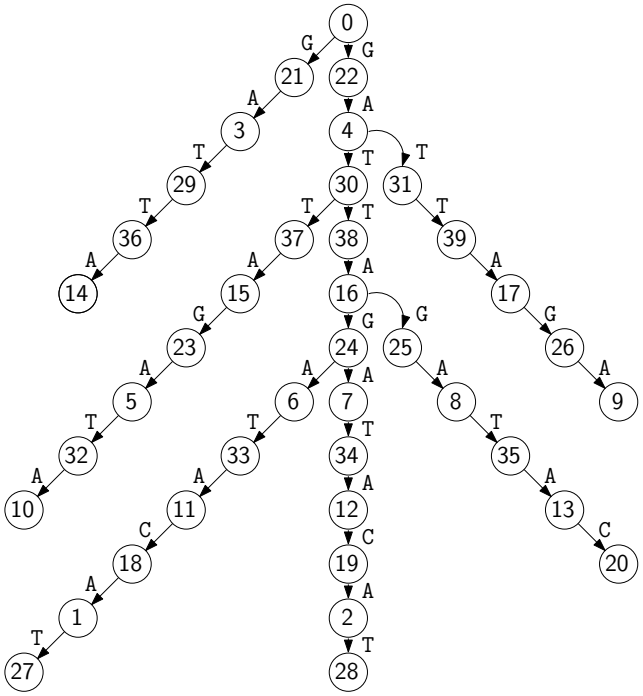


Figure 3 A directed, edge-labelled tree whose vertices are labelled to show it is a Wheeler graph. The XBWT is `GGTTTTTTTTTCCCGGGGAAAAAAAAATTTTAAAAAAAA` with 7 runs.

runs. We refer the reader to Giuliani et al.'s [28, 29] recent papers for a discussion of the impact of the `$` and of the direction of the string on the number of runs in the BWT. The following theorem shows that the example in Figure 3 is not a coincidence: if the "reads" have no errors and they are appended to the reference in the proper positions, then the XBWT has the same number of runs as the BWT of the *reverse* of the "genome".

▶ **Theorem 1.** *Suppose we sample substrings from a string and we form a labelled tree by grafting (appending) the substrings in the same position they were sampled so that all edge labels at the same depth are equal. Then the XBWT of the tree has the same number of runs as the BWT of the reverse of the string.*

Proof. Consider the tree shown in Figure 3. The tree satisfies the hypothesis of Theorem 1 since it was obtained by sampling some substrings from `GATTAGATACAT` and then grafting them onto it such that all the edge labels at the same depth are equal (so a horizontal line always hits edges with only the same label). Clearly, all the labels at the same depth not only are equal, but they have the same upward-path label, which is the prefix preceding the corresponding character in the string. Since the XBWT is built by sorting labels according to the string spelled by their upward path, we see that each symbol of the original string will be adjacent to all reads symbols at the same horizontal level, and that all such symbols are identical. Finally, observe that also in the BWT of the reverse of the string symbols are sorted according to the prefix preceding them; hence the XBWT can be obtained by replacing each symbol in the BWT, except the `$`, by a run of the same symbol and the thesis follows. ◀

Figure 3 and Theorem 1 suggest a new way to compress and index readsets: graft the reads onto a fully or partially assembled genome, or a reference genome if need be, and store the XBWT of the resulting tree. We note that, although assembly-free indexing is a more

general problem, indexing assembled reads is still of practical interest [17]. Many readsets have coverage of 30x or even 50x, which makes them extremely large but should also make run-length compression practical on the XBWTs. If we want to index readsets from several individuals, we can simply graft the reads onto the appropriate assembled genomes and compute the XBWT of the forest, which is also a Wheeler graph.

Theorem 1, provides an extremely good estimate of the number of runs of the XBWT, but it holds under the unrealistic assumption that the reads have no errors. However, we can take advantage of the fact that sequencing by synthesis has an asymmetric error profile: errors are much more likely at the end of a read than at the beginning. The following result shows that errors at the end of the reads have a limited impact to the overall number of runs in the XBWT.

▶ **Theorem 2.** *In the hypothesis of Theorem 1 suppose that the sampled substrings may differ from the reference string and that the average distance from* first *difference (insertion, deletion, or substitution) to the* end *of the substring is δ. Then, with respect to Theorem 1 the XBWT of the tree will have at most 2δ additional runs per substring.*

Proof. Consider a single substring of length ℓ in which the distance between the first difference and the end of the substring is d (we assume $d = 0$ if there are no differences). Reasoning as in the proof of Theorem 1, we see that the first $\ell - d$ symbols of the substring will end up in the same run as the corresponding symbol of the reference string (the one at the same depth in the tree). Each of the other d symbols will, in the worst case, end in the middle of a run of a different symbol thus creating two additional runs. Summing this additional runs over all substrings we get a total number of additional runs upper bounded by 2δ runs per substring. ◀

To guarantee that most of the errors are at the end of the reads, we propose to build two trees: one for the assembled genome and one for its reverse complement. Having two trees means we do not have to reverse and complement half the reads before grafting them onto a single tree: the reversal of the string would be problematic in view of Theorem 2 since it would move an error from the end of the read to its front. We can build two trees with a small additional cost since the alignment algorithm will tell us whether each read aligns to the reference or to its reverse complement.

Assuming our scheme guarantees an improvement in compression we want to be sure the resulting index is also efficient. Prezza [52] recently showed how to generalize Gagie, Navarro and Prezza's [25] results about fast locating from run-length compressed BWTs to run-length compressed XBWTs, at the cost of storing the trees' shapes, which takes a linear number of bits. For trees with far more internal vertices than leaves, however, it is relatively easy to support fast locating in small space, as a corollary of the following theorem.

▶ **Theorem 3.** *Let G be a Wheeler graph and r be the number of runs in a Burrows-Wheeler Transform of G, and suppose G can be decomposed into v edge-disjoint directed paths whose internal vertices each have in- and out-degree exactly 1. We can store G in $O(r + v)$ space such that later, given a pattern P, in $O(|P| \log\log |G|)$ time we can count the vertices of G reachable by directed paths labelled P, and then report those vertices in $O(\log\log |G|)$ time per vertex.*

▶ **Corollary 4.** *Let T be a labelled tree on n vertices obtained by grafting reads onto their assembled genome as described. Let r be the number of runs in the XBWT and let t be the number of reads. We can store T in $O(r + t)$ words of space such that later, given a pattern P, in $O((|P| + k) \log\log n)$ time we can report all the k vertices reachable by paths labelled P.*

We sketch a proof of Theorem 3 in A, although we omit the details because, at least when dealing with short reads, it may be more practical just to descend until we reach a branching node (in which case the pattern is in the assembled genome, not in a read) or a leaf. We have not yet considered carefully whether Nishimoto and Tabei's [50] faster locating can be applied to improve Theorem 3 or Corollary 4.

Before we concentrate on optimizations we should consider two basic questions: are our XBWTs for readsets significantly smaller than their EBWTs in practice and, if so, how can we build them efficiently? Theorem 2 offers some guarantees of compression, but to test how our idea works in practice in Section 5 we build the XBWT and EBWT for a real, high-coverage readset and see how the numbers of runs in them compare. In Section 4 instead we face the problem of the efficient construction of XBWTs for large datasets.

4 XBWT via Prefix Free Parsing

The problem of building the XBWT for a set of reads as described in Section 3 is non trivial because the input typically consists in tens of gigabytes of data and we cannot make use of the available algorithms [1, 3] which are designed to work in RAM. However, the fact that reads are copies (possibly with errors), of portions of a relatively small reference suggests that the overall amount of information content is relatively small. Therefore we decided to compute the XBWT using the technique of Prefix Free Parsing (PFP) that has been successfully utilized for computing the BWT for large collections of genomes from individuals of the same species. Our implementation was done in C++ and is available on `https://github.com/fnareoh/Big_XBWT`. Note that our algorithm does not take as input a labelled tree, but rather a reference genome and a set of reads aligned to that genome (in the format of a `.bam` file); the alignment implicitly defines a labeled tree as described in Section 3.

In the PFP construction of the BWT the input is parsed into overlapping phrases using context-triggered piecewise hashing [7]. If the input contains many repetitions, the use of context-triggered hashing ensures that the parsing will contain a relatively small number of distinct phrases. The actual construction of the BWT is done using only the dictionary of distinct phrase and the parse (which describes how the dictionary phrases can be used to reconstruct the input). For repetitive datasets the dictionary and the parse fit in RAM even when the original input does not. Unmodified, however, PFP does not work well on readsets since the phrases generated at the beginning and end of each read will likely be unique. As a result, the dictionary will be quite large and the algorithm inefficient. To prevent this, we extend the reads forward and backward so they begin and end with complete phrases. The extension is done using the symbols in the reference immediately before and after the position where the read aligns, so that the phrases are likely to be not unique (if the read has no errors the phrases will be exactly the same generated when parsing the reference). Although this technique maintains the dictionary small, the tricky part is to exclude these extensions when computing the actual XBWT.

Summing up, our implementation is divided in three main phases. In the first phase we partition the reference and the reads into phrases; the set of distinct phrases is called the *dictionary* and the way phrases form the reference and the reads is called the *parse*. We use the extension trick mentioned before, and ,if the reference and the reads are similar, the dictionary will be relatively small. In the second phase we compute the XBWT of the parse. Since phrases are relatively large, the number of symbols in the parse is much smaller than in the original input, so the parse fits in RAM and the computation can be done using a doubling algorithm [33]. Finally, in the third phase we recover the XBWT of the input from the XBWT of the parse. The details of the three phases are given below.

4.1 Construction of the Dictionary and the Parse

We start by scanning the reference as in the PFP BWT construction algorithm. The algorithm takes as input parameters a window size w, and a modulo m. We slide a window of length w over the text, at each step computing the Karp-Rabin fingerprint [32] of the window. We define a terminating windows as a window with Karp-Rabin fingerprint equal to zero modulo m. Terminating windows decompose the text into overlapping phrases: each phrase is a minimal substring that begins and ends with a terminating window. Note that each terminating window is a suffix of the current phrase and the prefix of the next phrase so consecutive phrases have a size-w overlap. Note that defining phrases using terminating windows ensures that no phrase is a prefix (or a suffix) of another phrase, hence the name "prefix free parsing".

In addition to keeping track of window fingerprints, we also maintain a different hash $h(p_i)$ of the current phrase p_i. For simplicity in the following we assume distinct phrases always have distinct hashes, if not we detect it and crash. At the end of this scanning phase, the reference has been parsed into the (overlapping) phrases $p_1, p_2, \ldots, p_z$. We build a vector $S[1, z]$ storing for each phrase p_i its starting position s_i in the reference and its hash $h(p_i)$. We also build as we go the dictionary that associate to each hash value $h(p_i)$ the corresponding phrase p_i (stored as a simple string) and $occ(p_i)$ the number of occurrences of that phrase. We will later also need the length of each phrase but we don't store it explicitly, just deduce it from the string stored in the dictionary.

After parsing the reference, we process the reads one by one. From the file of aligned reads, we obtain both the read r as a string and the position l where the read aligns to the reference. We binary search in S for the rightmost phrase p_s that starts before position l and for the leftmost phrase p_e that ends after position $l + |r| - 1$. Let p'_s (resp. p'_e) denote the prefix (resp. suffix) of p_s (resp. p_e) ending (resp. starting) immediately before (resp. after) position l (resp. $l + |r| - 1$). We define the extended read $r_{ext} = p'_s \cdot r \cdot p'_e$ where $\cdot$ here denotes string concatenation. We slide a window onto r_{ext}, decomposing it into phrases, as we did for the reference. Since r_{ext} starts and ends with a terminating window the phrases we add while parsing r_{ext} still form a prefix-free parsing. However, as we do not want to index the whole r_{ext} in the final XBWT, for each read we keep track and store to disk the starting and ending position of r in r_{ext}.

When processing the reads we continue adding the hashes of the phrases to the end parse, using a special value as separator between reads. If we parse a new phrase, we add it to the dictionary. However, as previously pointed out, the phrases coming from the extended reads are likely to be equal to phrases in the reference so we expect the dictionary not to grow significantly (the dictionary would not grow at all if all the reads were substrings of the reference). From the starting and ending position of the original read in the extended read we deduce for each phrase what characters are part of the original read (the reads without extensions) and we store a starting and ending position for each phrase.

Once all the reads have been processed, we sort the phrases in the dictionary in reverse lexicographic order and we output a new parse where each hash of phrase is replaced by its reverse lexicographic rank, the separator symbol is replaced by the number of phrases plus one. To summarize, at the end of this phase we have produced the following output files:

1. `file.dict`: the dictionary in co-lexicographic order;
2. `file.occ`: the frequency of each phrases;
3. `file.parse`: the parse with each phrase represented by its co-lexicographic rank;
4. `file.limits`: the starting and ending position of the original input (reads without extension) in each phrase.

4.2 XBWT of the Parse

The main goal of this phase is to construct the XBWT of the parse, using the co-lexicographic rank as meta-characters. To this end we load the parse on RAM, reconstruct its tree structure, and compute the XBWT of this tree via a doubling algorithm [33]. Then, rather than storing the XBWT as is, we construct an inverted list as this structure will be more appropriate for the next phase. For each phrase p_i we store the list of XBWT positions where p_i appears. The size of the inverted list for p_i is equal to its frequency; since frequencies were computed in the first phase, we can output the inverted list as a plain concatenation of positions.

In this phase we also permute the limits (the starting and ending position in the original input) of each phrase according to their order in the XBWT. This way, in the next phase, with the inverted list, we can easily access the limit of any given phrase in the parse. In this phase, we also compute and write to disk for every phrase, the list of phrases (with multiplicities) that immediately follow in the parse. This list will be used to index the characters that precede a full word. However because we only want to index the characters that are in the original input, we only add it after checking the limits. Finally, because we are not storing special characters to mark the end of a read or of the reference (as they would break runs), we construct a bit vector marking such positions and we permute it according to the XBWT order. To summarize, at the end of this phase we have produced the following output files:

1. `file.dict`: the dictionary of the reversed phrases (from the first phase).
2. `file.occ`: the frequency of each phrases (from the first phase).
3. `file.ilist`: the inverted list of the parse.
4. `file.xbwt_limits`: the limits of the phrases in XBWT order.
5. `file.xbwt_end`: markers of the phrases where a read or reference ends in XBWT order.
6. `file.full_children`: for every word, the list of words that follows it.

4.3 Building the final XBWT

This is the final phase where we compute the XBWT of the reference and of the readset. We start by sorting lexicographically the suffixes of the strings in the dictionary D. At this stage the dictionary D contains the phrases reversed, so this is equivalent to sort in reverse lexicographic order the prefixes of all phrases. We ignore the suffixes of length $\leq w$ as they correspond to the terminating window which also belongs to the previous phrase. The sorting is done by the gSACAK algorithm [43] which computes the SA and LCP array for the set of dictionary phrases. We scan the sorted elements of D, for s a proper suffix, there are two cases, all the elements in D which have s as a proper suffix have the same preceding character, in this case we add it the correct number of times using the frequency of each phrase. In the other case, we use a heap to merge the inverted list writing the appropriate characters accordingly. Here when writing a character we first check that the suffix length is between the limits and only write it to file if it does. We also check if the character to be added is the last of its sequence (read or reference), if so output a 1 to signal the end of a sequence, else 0. When finding a suffix s' that corresponds to an entire phrase, we use the children file to output the character at the start of the following phrase. At the end of this phase we have written to disk a file with the XBWT of the reference and readset as well as a bit vector marking which positions are the last character of a read or a genome. To summarize, in this phase we use `file.dict`, `file.occ`, `file.ilist`, `file.full_children` and `file.xbwt_limits`; all other files can be discarded. We output the XBWT in plain text as `file.bwt` and `file.is_end` is the compressed bit vector marking the end of reads.

5 Experiments

In this section we present a first experimental evaluation of our XBWT-based approach for compressing a set of aligned reads and we compare it with the known methods based on the EBWT. We compare ourselves to the EBWT and not other compression tools for aligned readset as our long-term goal is to create an index and not just compression. Recall that our implementation and experimental pipeline is available on github.com/fnareoh/Big_XBWT. For simplicity we compare the numbers of runs produced by the different algorithms. The actual compression depends on the algorithm used for encoding the run lengths: preliminary experiments with the γ encoder show that the number of runs is a good proxy for measuring the actual compression. An accurate comparison of the time efficiency is left as a future work: we only compared the number of runs produced by our XBWT with the number of runs produced by the EBWT and some of its variants. Note that our implementation computes the XBWT of the reference genome and the readset (as described in the previous section), while the EBWT and its variants were applied only to the readset. We computed all EBWT variants using ropeBWT2 [40]; in addition to plain EBWT we also tested 2 heuristics that reorder the reads to reduce the number of runs in the EBWT: Spring [11] and reverse lexicographic order (RLO) [14], the latter obtained using the option `-s` in ropeBWT2. Since our XBWT implementation does not use the `$` symbol, for a fair comparison we measured the number of runs with and without the `$` for EBWT, Spring+EBWT and RLO+EBWT (therefore ignoring for all algorithms the extra cost of implicitly encoding the ending position of each string). In our tests, we used the following readsets:

- **E.coli** and **S.aureus** from the single-cell dataset [12], the references used are those linked on the single-cell website[1],[2].
- **R.sphaeroides** We have HiSeq and MiSeq sequencing, raw and trimmed versions of the reads from the GAGE-B dataset [44]. The reference used is the longest contig assembled by MSRCA v1.8.3 [59] as it was the most accurate assembler according to the Gage-b companion paper [44]. We only considered the longest contig because our implementation doesn't handle forests of trees yet.
- **Human Chromosome 19** We used as a reference Chromosome 19 from the CHM1 human assembly [57] and one of the HiSeq 2000 readsets[3] used to compute that assembly, considering only the reads that aligned with the reference.

None of those readsets are aligned, so we used bwa mem [42] to align them to the chosen reference. In this preliminary experiments we discarded the reads that bwa aligned with the reverse-complement of the reference genome. As mentioned in Section 3 our final prototype will build an XBWT of the tree with the reference and of the tree of the reversed-complemented reference. In Table 1, we present statistics on the readsets we used: those statistics where computed only on the reads that aligned forward to the reference.

Preliminary experiments, not reported here, show that removing the `$` in the EBWT (all variants) reduces the number of runs between 2.7% and 29.2%. Consequently, we focus our analysis on the comparison of Plain (no read reordering) EBWT (without dollars), SPRING+EBWT (without dollars), RLO+EBWT, with and without `$` and XBWT.

The results of this comparison are reported in Figures 4a and 4b. They show that in general the plain EBWT performs worse followed by the SPRING reordering, RLO ordering with dollars then RLO ordering without dollars and finally XBWT performs best. XBWT yields a smaller number of runs than RLO+EBWT (with or without `$`) on all datasets,

[1] `https://www.ncbi.nlm.nih.gov/nuccore/NC_000913`
[2] `https://www.ncbi.nlm.nih.gov/nuccore/87125858`
[3] `https://www.ncbi.nlm.nih.gov/sra/SRX966833[accn]`

Table 1 Statistics on each dataset used in the experiments. Those statistics where computed only on the reads that aligned forward to the reference. We call sequencing error (or simply error) any difference between the genome and the reads. The coverage is simply defined as the total number of base-pairs in the reads compared to the number of base-pairs in the reference. The average distance between the first sequencing error and the end of the read and the end is computed considering that for error less read this distance is 0. Note that this parameter is exactly δ in Theorem 2.

Dataset	Number of reads	Read length	Coverage	Avg. dist. from the first sequencing err. to the end	Prop. of reads without seq. error	Error rate
E.coli [12]	14139182	100	304×	13	57.30%	0.01%
S.aureus [12]	26654420	100	927×	7	88.79%	0.01%
Human Chr19 [57]	34167479	100	57×	15	71.62%	0.01%
R.sphaeroides [44]						
HiSeq raw	166820	101	46×	27	31.34%	0.04%
HiSeq trimmed	134207	up to 101	37×	6	83.26%	0.01%
MiSeq raw	23102	251	24×	122	0.25%	0.15%
Miseq trimmed	20046	up to 251	20×	29	63.55%	0.03%

although the number is comparable on some datasets this is still a significant improvement considering that RLO+EBWT already has far less run than the EBWT baseline. On the Chr19 dataset, using RLO+EBWT-no-$ over plain BWT-no-$ (not reported in Figure 4a) reduced the number of runs by 49%; using the XBWT reduced the number of runs by an additional 16%. On S.aureus and E.coli the reduction between RLO+EBWT-no-$ and XBWT is of only 3% and 8% respectively.

The R.sphaeroides datasets are especially interesting as they involve two NGS technologies that generate reads of different lengths, different coverages, and with different error profiles. We can first notice that our method brings greater benefits on the HiSeq sequencing which has smaller reads with less errors that are located towards the end of the string. This is an experimental validation of the statement of Theorem 2. We can also observe the effect of trimming the reads on the number of runs. On the HiSeq sequencing, trimming reduces the coverage only from 46x to 37x but yields a reduction in the number of XBWT runs by 86%. Note that, as a result, on HiSeq trimmed, the number of XBWT runs is less than half the number of runs in plain RLO+EBWT.

6 Application to the JST

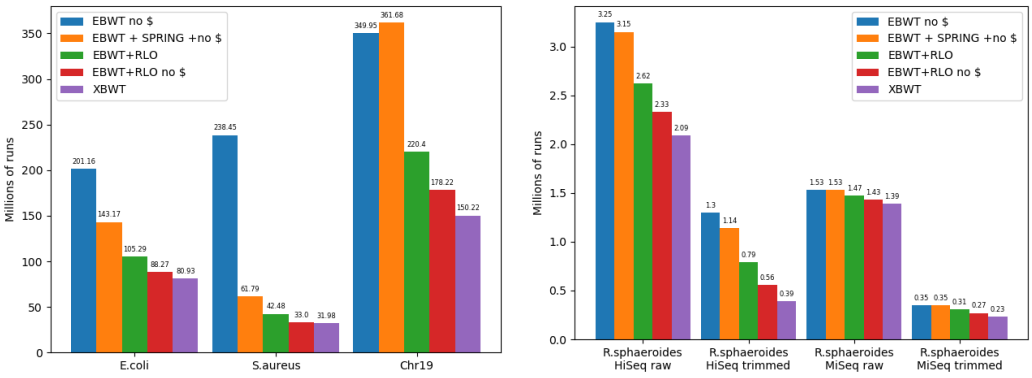
From a certain angle, Figure 3 is reminiscent of Figure 5, from Rahn, Weese and Reinert's [54] paper on their Journaled String Tree (JST). This raises the question of whether the XBWT and JST can be used to improve the space usage of the hybrid index [20, 26, 21] and eventually the PanVC [58] pan-genomic read aligner, which is based on the hybrid index.

Figure 5 shows a JST supporting search for patterns of length up to 4 in four aligned sequences: the reference

$r =$ TAGCGTAGCAGCTATGAGGAGGACCGAGTT

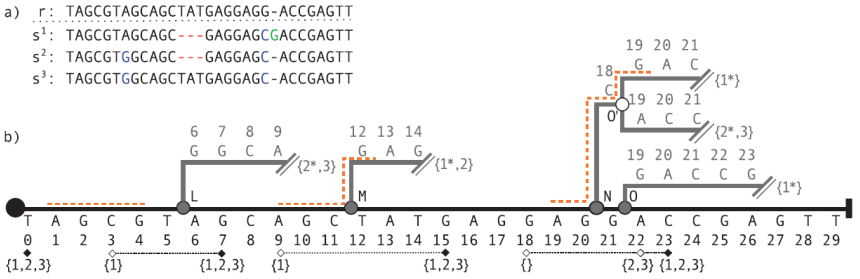
and three others,

$s^1 =$ TAGCGTAGCAGCGAGGAGCGACCGAGTT,
$s^2 =$ TAGCGTGGCAGCGAGGAGCACCGAGTT,
$s^3 =$ TAGCGTGGCAGCTATGAGGAGCACCGAGTT.



(a) E.coli, S.aureus, Human Chr19. **(b)** R.sphaeroides.

Figure 4 Comparison of run-lengths compression by RLO+EBWT with and without $ and XBWT on various species (4a) and on two sequencing of R.sphaeroides (HiSeq and MiSeq) and for reads both raw and trimmed (4b).



Online search of the pattern $p = AGCG$ over a JST depicted in (**b**) representing the four aligned sequences in (**a**), where r is chosen as the reference. The window length is 4. The bullets on the black line represent the branch-nodes: $L = (6, G, \{2, 3\}, R)$; $M = (12, TAT, \{1, 2\}, D)$; $N = (21, C, \{1, 2, 3\}, R)$; $O = (22, G, \{1\}, I)$. Below is plotted the reference with the reference coordinates. Branches are indicated by gray lines labeled with the respective sequence context depending on the window length and with the virtual positions of the chosen proxy (marked sequence number). The sets at the end denote the coverage. O' is a split-node induced by O in the branch of N. The dotted lines capped with diamonds below the reference positions represent the ranges in which invalid paths are suppressed. The orange (dashed) lines represent matches of the pattern p

Figure 5 An illustration of a JST [54].

The straight branch of the tree running along the bottom of the figure is labelled with r, and the other branches indicate places where the other sequences differ from r. The other branches end just before a window of size 4 sliding over their sequences matches an aligned window of size 4 sliding over r. For example, the first branch ends at position 9 because a sliding window of length 4 over positions 7 to 10 of sequences s^2 and s^3 (that is, containing the characters in columns 7 to 10 and the rows for s^2 and s^3 in the alignment shown at the top right in the figure), matches a sliding window of length 4 over positions 7 to 10 in r (that is, containing the characters in columns 7 to 10 and the row for r in the alignment).

Suppose we are looking for the pattern $p = \texttt{AGCG}$: considering the circle at the left as the root, p occurs 3 times as a substring (marked in orange) of root-to-leaf paths, and we can find those occurrences using a depth-first traversal of the tree. Since the sequences are similar, such a traversal is faster than running a sliding window over each sequence separately. If we find an occurrence of p in the tree that ends at a node not in the branch for r, then we have found occurrences in each of the sequences labelling the leaves in that node's subtree. If we find an occurrence of p in the branch for r, then we have found occurrences in r and possibly other sequences. Unfortunately this case is not illustrated in the figure, but if we

were looking for `GTAG` then the occurrence at position 4 in r would have a corresponding occurrence in s^1 but not in s^2 or s^3; this is shown by the dashed line between 3 and 7, with $\{1\}$ at the left end indicating that s^1 matches r between 3 and 7 and $\{1, 2, 3\}$ indicating that s^1, s^2 and s^3 all match r from 7 onward (until the next such interval starts at 9).

The hybrid index is conceptually similar to the JST, but the former is an index and the latter performs pattern matching by scanning the tree sequentially. To build the hybrid index supporting search for patterns of length up to 4 in r, s^1, s^2, s^3, we first build a string kernel consisting of r and substrings from s^1, s^2, s^3 that contain all the characters within distance 3 of variations from r, all separated by copies of a special symbol `$`:

`TAGCGTAGCAGCTATGAGGAGGACCGAGTT$CGTGGCA$AGCGAG$GAGCGACC$GAGCACC` .

Any substring of length at most 4 of the the four sequences r, s^1, s^2, s^3 is a substring of the string kernel, and any substring of length at most 4 of the string kernel that does not include a copy of `$` is a substring of at least one of those sequences. We then build an FM-index for the string kernel, with auxiliary data structure that allow us to quickly map occurrences of a pattern in the string kernel to occurrences in the sequences.

It seems interesting that the string kernel for the four sequences in Figure 5 has more characters than the JST: on top of r, the string kernel has a substring `$CGTGGCA` and the JST has a branch labelled `GGCA`; the string kernel has `$AGCGAG` and the JST has `GAG`; the string kernel has `$GAGCGACC` and the JST has `CGAC` and `GACCG` (a tie in this one case); the string kernel has `$GAGCACC` and the JST has `CACC` (with the first `C` shared with the branch ending `CGAC`). This difference is because the string kernel stores copies of the characters both before and after variation sites, whereas the JST stores copies only of the characters after them. If we build an index using the XBWT of the JST, therefore, it may be smaller than the hybrid index while having the same basic functionality. We leave exploring this possiblity as future work.

References

1. Jarno Alanko, Giovanna D'Agostino, Alberto Policriti, and Nicola Prezza. Regular languages meet prefix sorting. In *Proceedings of the Thirty-First Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '20, page 911–930. Society for Industrial and Applied Mathematics, 2020.
2. Fatemeh Almodaresi, Prashant Pandey, and Rob Patro. Rainbowfish: a succinct colored de Bruijn graph representation. In *17th International Workshop on Algorithms in Bioinformatics (WABI 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.
3. Uwe Baier, Thomas Büchler, Enno Ohlebusch, and Pascal Weber. Edge minimization in de Bruijn graphs. In Ali Bilgin, Michael W. Marcellin, Joan Serra-Sagristà, and James A. Storer, editors, *Data Compression Conference, DCC 2020, Snowbird, UT, USA, March 24-27, 2020*, pages 223–232. IEEE, 2020. doi:10.1109/DCC47342.2020.00030.
4. Hideo Bannai, Travis Gagie, and I Tomohiro. Refining the r-index. *etical Computer Science*, 812:96–108, 2020.
5. Markus J Bauer, Anthony J Cox, and Giovanna Rosone. Lightweight algorithms for constructing and inverting the BWT of string collections. *Theoretical Computer Science*, 483:134–148, 2013.
6. Jason W Bentley, Daniel Gibney, and Sharma V Thankachan. On the complexity of BWT-runs minimization via alphabet reordering. In *28th Annual European Symposium on Algorithms (ESA 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.
7. Christina Boucher, Travis Gagie, Alan Kuhnle, Ben Langmead, Giovanni Manzini, and Taher Mun. Prefix-free parsing for building big BWTs. *Algorithms for Molecular Biology*, 14(1):1–15, 2019.

8 Alexander Bowe, Taku Onodera, Kunihiko Sadakane, and Tetsuo Shibuya. Succinct de Bruijn graphs. In Ben Raphael and Jijun Tang, editors, *Algorithms in Bioinformatics*, pages 225–235. Springer Berlin Heidelberg, 2012.

9 Alexander Bowe, Taku Onodera, Kunihiko Sadakane, and Tetsuo Shibuya. Succinct de Bruijn graphs. In *International workshop on algorithms in bioinformatics (WABI)*, pages 225–235. Springer, 2012.

10 Michael Burrows and David Wheeler. A block-sorting lossless data compression algorithm. In *Digital SRC Research Report*. Citeseer, 1994.

11 Shubham Chandak, Kedar Tatwawadi, Idoia Ochoa, Mikel Hernaez, and Tsachy Weissman. SPRING: a next-generation compressor for FASTQ data. *Bioinformatics*, 35(15):2674–2676, 2018. doi:10.1093/bioinformatics/bty1015.

12 Hamidreza Chitsaz, Joyclyn Yee-Greenbaum, Glenn Tesler, Mary-Jane Lombardo, Christopher Dupont, Jonathan Badger, Mark Novotny, Douglas Rusch, Louise Fraser, Niall Gormley, Ole Schulz-Trieglaff, Geoffrey Smith, Dirk Evers, Pavel Pevzner, and Roger Lasken. Efficient de novo assembly of single-cell bacterial genomes from short-read data sets. *Nature biotechnology*, 29:915–21, September 2011. doi:10.1038/nbt.1966.

13 Anthony J Cox, Markus J Bauer, Tobias Jakobi, and Giovanna Rosone. Large-scale compression of genomic sequence databases with the Burrows-Wheeler transform. *Bioinformatics*, 28(11):1415–1419, 2012.

14 Anthony J. Cox, Markus J. Bauer, Tobias Jakobi, and Giovanna Rosone. Large-scale compression of genomic sequence databases with the Burrows-Wheeler transform. *Bioinform.*, 28(11):1415–1419, 2012. doi:10.1093/bioinformatics/bts173.

15 Anthony J. Cox, Tobias Jakobi, Giovanna Rosone, and Ole B. Schulz-Trieglaff. Comparing DNA sequence collections by direct comparison of compressed text indexes. In Ben Raphael and Jijun Tang, editors, *Algorithms in Bioinformatics*. Springer Berlin Heidelberg, 2012.

16 D. Díaz-Domínguez and G. Navarro. A grammar compressor for collections of reads with applications to the construction of the BWT. In *Proc. 31th Data Compression Conference (DCC)*, 2021. To appear.

17 Dirk D Dolle, Zhicheng Liu, Matthew Cotten, Jared T Simpson, Zamin Iqbal, Richard Durbin, Shane A McCarthy, and Thomas M Keane. Using reference-free compressed data structures to analyze sequencing reads from thousands of human genomes. *Genome research*, 27(2):300–309, 2017.

18 Lavinia Egidi, Felipe A Louza, Giovanni Manzini, and Guilherme P Telles. External memory BWT and LCP computation for sequence collections with applications. *Algorithms for Molecular Biology*, 14(1):1–15, 2019.

19 Lavinia Egidi and Giovanni Manzini. Lightweight merging of compressed indices based on BWT variants. *Theoretical Computer Science*, 812:214–229, 2020.

20 Héctor Ferrada, Travis Gagie, Tommi Hirvola, and Simon J Puglisi. Hybrid indexes for repetitive datasets. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 372(2016):20130137, 2014.

21 Héctor Ferrada, Dominik Kempa, and Simon J Puglisi. Hybrid indexing revisited. In *2018 Proceedings of the Twentieth Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 1–8. SIAM, 2018.

22 Paolo Ferragina, Fabrizio Luccio, Giovanni Manzini, and Senthilmurugan Muthukrishnan. Compressing and indexing labeled trees, with applications. *Journal of the ACM (JACM)*, 57(1):1–33, 2009.

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

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

25 Travis Gagie, Gonzalo Navarro, and Nicola Prezza. Fully functional suffix trees and optimal text searching in BWT-runs bounded space. *Journal of the ACM (JACM)*, 67(1):1–54, 2020.

26 Travis Gagie and Simon J Puglisi. Searching and indexing genomic databases via kernelization. *Frontiers in Bioengineering and Biotechnology*, 3:12, 2015.

27 Erik Garrison, Jouni Sirén, Adam M Novak, Glenn Hickey, Jordan M Eizenga, Eric T Dawson, William Jones, Shilpa Garg, Charles Markello, Michael F Lin, et al. Variation graph toolkit improves read mapping by representing genetic variation in the reference. *Nature biotechnology*, 36(9):875–879, 2018.

28 Sara Giuliani, Shunsuke Inenaga, Zsuzsanna Lipták, Nicola Prezza, Marinella Sciortino, and Anna Toffanello. Novel results on the number of runs of the Burrows-Wheeler transform. In Tomáš Bureš, Riccardo Dondi, Johann Gamper, Giovanna Guerrini, Tomasz Jurdziński, Claus Pahl, Florian Sikora, and Prudence W.H. Wong, editors, *SOFSEM 2021: Theory and Practice of Computer Science*, pages 249–262, Cham, 2021. Springer International Publishing.

29 Sara Giuliani, Zsuzsanna Lipták, Francesco Masillo, and Romeo Rizzi. When a dollar makes a BWT. *Theoretical Computer Science*, 2019.

30 Lilian Janin, Ole Schulz-Trieglaff, and Anthony J Cox. BEETL-fastq: a searchable compressed archive for DNA reads. *Bioinformatics*, 30(19):2796–2801, 2014.

31 Juha Kärkkäinen, Giovanni Manzini, and Simon J Puglisi. Permuted longest-common-prefix array. In *Annual Symposium on Combinatorial Pattern Matching*, pages 181–192. Springer, 2009.

32 R. M. Karp and M. O. Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 31(2):249–260, 1987. doi:10.1147/rd.312.0249.

33 Richard M. Karp, Raymond E. Miller, and Arnold L. Rosenberg. Rapid identification of repeated patterns in strings, trees and arrays. In *Proceedings of the Fourth Annual ACM Symposium on Theory of Computing*, page 125–136, 1972. doi:10.1145/800152.804905.

34 Alice M Kaye and Wyeth W Wasserman. The genome atlas: Navigating a new era of reference genomes. *Trends in Genetics*, 2021.

35 Yuichi Kodama, Martin Shumway, and Rasko Leinonen. The sequence read archive: explosive growth of sequencing data. *Nucleic acids research*, 40(D1):D54–D56, 2012.

36 Alan Kuhnle, Taher Mun, Christina Boucher, Travis Gagie, Ben Langmead, and Giovanni Manzini. Efficient construction of a complete index for pan-genomics read alignment. *Journal of Computational Biology*, 27(4):500–513, 2020.

37 Ben Langmead. Algorithms for DNA sequencing: Base calling and sequencing errors, May 2015. URL: https://www.youtube.com/watch?v=U4QnpciIJhM&list=PL2mpR0RYFQsBiCWVJSvVAO3OJ2t7DzoHA&index=10.

38 Ben Langmead and Steven L Salzberg. Fast gapped-read alignment with Bowtie 2. *Nature methods*, 9(4):357, 2012.

39 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):1–10, 2009.

40 Heng Li. Fast construction of FM-index for long sequence reads. *Bioinform.*, 30(22):3274–3275, 2014. doi:10.1093/bioinformatics/btu541.

41 Heng Li and Richard Durbin. Fast and accurate short read alignment with Burrows-Wheeler transform. *Bioinformatics*, 25(14):1754–1760, 2009.

42 Heng Li and Richard Durbin. Fast and accurate short read alignment with Burrows-Wheeler transform. *Bioinformatics*, 25(14):1754–1760, 2009. doi:10.1093/bioinformatics/btp324.

43 Felipe A. Louza, Simon Gog, and Guilherme P. Telles. Inducing enhanced suffix arrays for string collections. *Theoretical Computer Science*, 678:22–39, 2017. doi:10.1016/j.tcs.2017.03.039.

44 Tanja Magoc, Stephan Pabinger, Stefan Canzar, Xinyue Liu, Qi Su, Daniela Puiu, Luke J. Tallon, and Steven L. Salzberg. GAGE-B: an evaluation of genome assemblers for bacterial organisms. *Bioinform.*, 29(14):1718–1725, 2013. doi:10.1093/bioinformatics/btt273.

45 Veli Mäkinen, Djamal Belazzougui, Fabio Cunial, and Alexandru I Tomescu. *Genome-scale algorithm design*. Cambridge University Press, 2015.

46 S. Mantaci, A. Restivo, G. Rosone, and M. Sciortino. An extension of the Burrows–Wheeler transform. *Theoretical Computer Science*, 387(3):298–312, 2007. doi:10.1016/j.tcs.2007.07.014.

47 Sabrina Mantaci, Antonio Restivo, Giovanna Rosone, and Marinella Sciortino. An extension of the Burrows-Wheeler transform. *Theoretical Computer Science*, 387(3):298–312, 2007.

48 Martin D Muggli, Alexander Bowe, Noelle R Noyes, Paul S Morley, Keith E Belk, Robert Raymond, Travis Gagie, Simon J Puglisi, and Christina Boucher. Succinct colored de Bruijn graphs. *Bioinformatics*, 33(20):3181–3187, 2017.

49 Gonzalo Navarro. *Compact data structures: A practical approach*. Cambridge University Press, 2016.

50 Takaaki Nishimoto and Yasuo Tabei. Faster queries on BWT-runs compressed indexes. *arXiv preprint*, 2020. arXiv:2006.05104.

51 Alberto Policriti and Nicola Prezza. LZ77 computation based on the run-length encoded BWT. *Algorithmica*, 80(7):1986–2011, 2018.

52 Nicola Prezza. On locating paths in compressed tries. In *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 744–760. SIAM, 2021.

53 Nicola Prezza. Subpath queries on compressed graphs: A survey. *Algorithms*, 14(1):14, 2021.

54 René Rahn, David Weese, and Knut Reinert. Journaled string tree - a scalable data structure for analyzing thousands of similar genomes on your laptop. *Bioinformatics*, 30(24):3499–3505, 2014. doi:10.1093/bioinformatics/btu438.

55 Julian Seward. bzip2 and libbzip2, 1996. avaliable at http://www.bzip.org.

56 Jouni Sirén, Niko Välimäki, and Veli Mäkinen. Indexing graphs for path queries with applications in genome research. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 11(2):375–388, 2014.

57 Karyn Meltz Steinberg, Valerie A. Schneider, Tina A. Graves-Lindsay, Robert S. Fulton, Richa Agarwala, John Huddleston, Sergey A. Shiryev, Aleksandr Morgulis, Urvashi Surti, Wesley C. Warren, Deanna M. Church, Evan E. Eichler, and Richard K. Wilson. Single haplotype assembly of the human genome from a hydatidiform mole. *Genome Research*, 24(12):2066–2076, 2014. doi:10.1101/gr.180893.114.

58 Daniel Valenzuela, Tuukka Norri, Niko Välimäki, Esa Pitkänen, and Veli Mäkinen. Towards pan-genome read alignment to improve variation calling. *BMC genomics*, 19(2):123–130, 2018.

59 Aleksey V. Zimin, Guillaume Marçais, Daniela Puiu, Michael Roberts, Steven L. Salzberg, and James A. Yorke. The MaSuRCA genome assembler. *Bioinformatics*, 29(21):2669–2677, August 2013. URL: https://academic.oup.com/bioinformatics/article-pdf/29/21/2669/18533361/btt476.pdf.

A Proof Sketch for Theorem 3

Let G be a Wheeler graph with the vertices sorted according to the permutation π. A Burrows-Wheeler Transform (BWT) of G according to π is a permutation of G's edge labels such that, for any pair of edges $e = (u, v)$ and $e' = (u', v')$ labelled a and a' respectively, if $u < u'$ then a precedes a' in that permutation. For convenience, we assume that the labels of each vertex's out-edges appear in the order in π of their destinations. Notice there may be many BWTs for G because it may have many permutations π satisfying the Wheeler graph conditions.

Let B be a BWT of G according to π. By the definition of a Wheeler graph, for any pattern P over the alphabet of edge labels, the vertices reachable by directed paths labelled P form an interval in π. Moreover, if we store a rank data structure for B and partial sum data structures for the frequencies of the distinct edge labels and the vertices' in- and out-degrees, then given P we can find its interval in $O(|P| \log \log |G|)$ time. Let r is the number of runs

(i.e., maximal non-empty unary substrings) in B and suppose G can be decomposed into υ edge-disjoint directed paths whose internal vertices each have in- and out-degree exactly 1. Then these data structures take a total of $O(r + \upsilon)$ space, measured in words.

Let D be a such decomposition of G and n be the number of vertices in G, and assume the vertices are assigned numeric identifiers from 0 to $n - 1$ such that if (u, v) is an edge and neither u nor v is an endpoint of a path in D, and u has identifier i, then v has identifier $i + 1$. Notice these identifiers are not necessarily the vertices' ranks in π. For convenience, we assume that even though G is a multigraph, the number of edges is polynomial in n, so $\log\log|G| = O(\log\log n)$. We show how, still using $O(r + \upsilon)$ space, after we have found the interval for P we can then report the vertices in it using $O(\log\log n)$ time for each one.

We first prove a generalization of Bannai, Gagie and I's version [4] of Policriti and Prezza's Toehold Lemma [51], that lets us report the last vertex in the interval for P. We then define a generalization of Kärkkäinen, Manzini and Puglisi's ϕ function [31], that maps each vertex's identifier to the identifier of its predecessor in π. Finally, we give a generalization of a key lemma behind Gagie, Navarro and Prezza's r-index [25], that lets us compute our generalized ϕ function with $O(r + \upsilon)$-space data structures. Combined, these three results yield a generalized r-index for Wheeler graphs.

A.1 Generalized Toehold Lemma

For any pattern $P[0..m - 1]$, the interval for the empty suffix $P[m..m - 1]$ of P is all of π, because every vertex is reachable by an empty path. Assume we have found the interval $\pi[s_{i+1}, e_{i+1}]$ for $P[i + 1..m - 1]$ and now we want to find the interval $\pi[s_i, e_i]$ for $P[i..m - 1]$. With the partial sum data structure for the vertices' out-degrees, in $O(\log\log n)$ time we can find the interval in B containing the labels of the edges leaving the vertices in $\pi[s_{i+1}, e_{i+1}]$.

By the definition of a Wheeler graph, the edges labelled with the first and last occurrences of $P[i]$ in that interval in B, lead to the first and last vertices in the interval $\pi[s_i, e_i]$ for $P[i..m - 1]$. Using the partial sum data structures for the frequencies of the distinct edge labels and the vertices in-degrees, in $O(\log\log n)$ time we can find the ranks s_i and e_i in π of those first and last vertices in $\pi[s_i, e_i]$. It follows that in $O(\log\log n)$ time we can find $\pi[s_i, e_i]$ from $\pi[s_{i+1}, e_{i+1}]$; therefore, by induction, we can find the interval for P in $O(|P|\log\log n)$ time. We can count the vertices in that interval in the same asymptotic time by simply returning the size of the interval.

To be able to find the identifier of the last vertex in the interval for P, for each edge (u, v) we store u's and v's identifiers if any of the following conditions hold:

- (u, v)'s label a is the last label in a run in B;
- either u or v is an endpoint of a path in D;
- the vertex that follows u in π has out-degree 0.

We store a select data structure for B, a bitvector marking the labels a in B for whose edges (u, v) we have u's and v's identifiers stored, and a hash table mapping the position in B of each marked label a to the identifiers of its edge's endpoints. This again takes a total of $O(r + \upsilon)$ space.

By querying the rank data structure, the select data structure, the bitvector and the hash table in that order, we can find the identifier of the vertex reached by the edge labelled by the last copy of $P[m - 1]$ in B. By the definition of a Wheeler graph, this is the last vertex in the interval $\pi[s_{m-1}, e_{m-1}]$ for $P[m - 1]$. Assume we have found the interval $\pi[s_{i+1}, e_{i+1}]$ for $P[i + 1..m - 1]$ and the identifier of the last vertex u in that interval, and now we want to find the interval $\pi[s_i, e_i]$ for $P[i..m - 1]$ and the identifier of the last vertex v in that interval. We can find $\pi[s_i, e_i]$ as described above, so we need only say how to find v's identifier.

With the partial sum data structure on the vertices' out-degree and the rank data structure, in $O(\log\log n)$ time we can check whether u has an outgoing edge labelled $P[i]$. If it does then, of all its out-edges labelled $P[i]$, the one whose label appears last in B goes to v. By our assumption of how the vertices are assigned their identifiers, if neither u nor v are endpoints of a path in D, then v's identifier is u's identifier plus 1. If either u or v is an endpoint of a path in D, then we have v's identifier stored and we can use the hash table to find it from the position in B of the last label $P[i]$ on one of u's out-edges, again in $O(\log\log n)$ time.

If u does not have an outgoing edge labelled $P[i]$ then we can use the rank data structure to find the last copy of $P[i]$ in B that labels an edge leaving a vertex in $\pi[s_{i+1}, e_{i+1}]$. By the definition of a Wheeler graph, this edge (u', v) goes to v. Unlike in a BWT of a string, however, its label may not be the end of a run in B: u could have out-degree 0, u' could immediately precede u in π and the last of its outgoing edges' labels in B could be a copy of $P[i]$, and the first label in B of an outgoing edge of the successor of u in π could also be a copy of $P[i]$. This is why we store v's identifier if the vertex that follows u' in π has out-degree 0. If (u', v)'s label is the end of a run in B, of course, then we also have v's identifier stored. In both cases we use $O(\log\log n)$ time, so from the interval $\pi[s_{i+1}, e_{i+1}]$ for $P[i+1..m-1]$ and the identifier of the last vertex u in that interval, in $O(\log\log n)$ time we can compute the interval $\pi[s_i, e_i]$ for $P[i..m-1]$ and the identifier of the last vertex v in that interval. Therefore, by induction, in $O(|P|\log\log n)$ time we can find the interval for P and the identifier of the last vertex in that interval.

▶ **Lemma 5.** *We can store G in $O(r+v)$ space such that in $O(|P|\log\log n)$ time we can find the interval for P and identifier of the last vertex in that interval.*

A.2 Generalized ϕ

For a string S, the function ϕ takes a position i in S and returns the starting position of the suffix of S that immediately precedes $S[i..|S|-1]$ in the lexicographic order of the suffixes. In other words, ϕ takes the value in some cell of suffix array of S and returns the value in the preceding cell. Given a pattern P, if we can find the interval of the suffix array containing the starting positions of occurrences of P in S, and the entry in the last cell in that interval, then by iteratively applying ϕ we can report the starting positions of all the occurrences of P. This is the idea behind the r-index for strings, which uses a lemma saying it takes only space proportional to the number of runs in the BWT of S to store data structures that let us evaluate ϕ in $O(\log\log|S|)$ time.

We generalize ϕ to Wheeler graphs by redefining it such that it takes the identifier of some vertex u in G and returns the identifier of the vertex that immediately precedes u in π. (For our purposes here, it is not important how ϕ behaves when given the identifier of the first vertex in π.) Given a pattern P, if we can find the interval in π containing the vertices in G reachable by directed paths labelled P, and the identifier of the last vertex in that interval, then by iteratively applying ϕ we can report the identifiers of all those vertices.

Let J be the set that contains u's identifier if and only if any of the following conditions hold:

- u has out-degree not exactly 1;
- u has a single outgoing edge (u, v) but v has in-degree not exactly 1;
- the predecessor u' of u in π has out-degree not exactly 1;
- u' has a single outgoing edge (u', v') but v' has in-degree not exactly 1;
- the edges (u, v) and (u', v') have different labels.

We store a successor data structure for J and, if u's identifier is in J, then we store with it as satellite data the identifier of u's predecessor u' in π. Notice u's identifier is in J only if at least one of u or u' or v or v' is the endpoint of a path in D, or the label of (u', v') is the the last in a run in B and the label of (u, v) is the first in the next run. It follows that we can use $O(r + v)$ space for the successor data structure and have it support queries in $O(\log \log n)$ time.

Suppose we know the identifier of some vertex u with identifier i that is immediately preceded by u' in π with identifier i'. If $u \in J$ then we have i' stored as satellite data with $\succ (i) = i$. If $u \notin J$, then u has a single outgoing edge (u, v) and u' has a single outgoing edge (u', v') with the same label, say a, and v and v' each have in-degree exactly 1. By our assumption on how the identifiers are assigned, the identifiers of v and v' are $i + 1$ and $i' + 1$ and, by the definition of a Wheeler graph, v is immediately preceded by v' in π. It follows that if $i + \ell$ is the successor of i then it has stored with it as satellite data $i' + \ell$, and so we can compute ℓ and then i' in $O(\log \log n)$ time.

▶ **Lemma 6.** *We can store G in $O(r + v)$ space such that we can evaluate ϕ in $O(\log \log n)$ time.*

A.3 Discussion

Combining Lemmas 5 and 6, we generalize, we obtain Theorem 3. Since $v = 1$ for a single string labelling a simple path or cycle, Theorem 3 gives the same $O(r)$ space bound and $O(|P| + k \log \log n)$ time bound we achieve with the r-index for strings, where k is the number of occurrences. Nishimoto and Tabei [50] recently improved the query time of the r-index for strings to $O(P + k \log \log n)$ – or optimal $O(P + k)$ for polylogarithmic alphabets – without changing the space bound, and we conjecture this is achievable also for r-indexes for Wheeler graphs.