Wheeler maps *

Andrej Baláž[1], Travis Gagie[2], Adrián Goga[1], Simon Heumos[3,4,5], Gonzalo Navarro[6], Alessia Petescia[1], and Jouni Sirén[7]

[1]Comenius University in Bratislava, Slovakia
[2]CeBiB & Dalhousie University, Canada
[3]Quantitative Biology Center (QBiC), University of Tübingen, Germany
[4]Department of Computer Science, University of Tübingen, Germany
[5]M3 Research Center, University Hospital Tübingen, Germany
[6]CeBiB & DCC, University of Chile, Chile
[7]University of California Santa Cruz Genomics Institute, USA

Abstract. Motivated by challenges in pangenomic read alignment, we propose a generalization of Wheeler graphs that we call Wheeler maps. A Wheeler map stores a text $T[1..n]$ and an assignment of tags to the characters of T such that we can preprocess a pattern $P[1..m]$ and then, given i and j, quickly return all the distinct tags labeling the first characters of the occurrences of $P[i..j]$ in T. For the applications that most interest us, characters with long common contexts are likely to have the same tag, so we consider the number t of runs in the list of tags sorted by their characters' positions in the Burrows-Wheeler Transform (BWT) of T. We show how, given a straight-line program with g rules for T, we can build an $O(g + r + t)$-space Wheeler map, where r is the number of runs in the BWT of T, with which we can preprocess a pattern $P[1..m]$ in $O(m \log n)$ time and then return the k distinct tags for $P[i..j]$ in optimal $O(k)$ time for any given i and j.

1 Introduction

For years, geneticists have been worried about the fact that using a single reference for the human genomes biases scientific studies and medical diagnoses, undermining the potential of personalized medicine, particularly for people from under-represented groups. To address this bias, researchers [18] recently published a pangenome consisting of nearly complete genomes from 47 people from

* Research funded in part by European Union's Horizon 2020 research and innovation program under Marie Skłodowska-Curie grant agreement No 956229 (ALPACA) and by grants 1/0463/20 and 1/0538/22 from the Scientific Grant Agency of the Ministry of Education, Science, Research, and Sport of the Slovak Republic and Slovak Academy of Sciences (VEGA) and grant APVV-22-0143 from the Slovak Research and Development Agency. T.G. and G.N. funded in part by Basal Funds FB0001, ANID, Chile. T.G. funded in part by NSERC RGPIN-07185-2020. S.H. funded in part by the Central Innovation Programme (ZIM) for SMEs of the Federal Ministry for Economic Affairs and Energy of Germany. J.S. funded in part by National Human Genome Research Institute (NHGRI) award R01HG010485.

diverse origins and took, according to the *New York Times* [8], "a major step toward a deeper understanding of human biology and personalized medicine for people from a wide range of racial and ethnic backgrounds". Eventually, the plan is to include 350 genomes, but even this many genomes cannot fully capture humanity's genetic diversity. As the *Guardian* [34] put it, "as long as the reference contains only a subset, arguably someone will not make the cut". Ultimately, there will be pressure for a reference of at least thousands of genomes.

One of the primary use of a reference is during read alignment. As a DNA sample passes through a sequencing machine, the machine records the genome in short substrings called *reads*. The length and accuracy of the reads vary depending on the sequencing technology used. Next, software called a read aligner uses an index of a reference to find *seeds*, sections of the reads that exactly match sections in the reference, and uses dynamic programming to extend those seeds to approximate matches of the whole read. These approximate matches form alignments, which are used in many subsequent bioinformatics analyses.

Indexing 47 human genomes is feasible even with standard read aligners such as Bowtie [16] and BWA [17], and even indexing 350 may be possible on supercomputers, but indexing thousands will require new algorithmic insights. The emerging consensus is that we should represent the combined reference sequences as a *pangenome graph* [6] that shows variation between genomes as detours on an otherwise shared path. The necessity of mapping reads to the version of the path that best fits the sample leads to the question of how to index pangenome graphs.

Equi et al. [9] showed that, unless the strong exponential-time hypothesis is false, one cannot index a graph in polynomial time such that pattern matching can run in sub-quadratic time, so several groups have tried constraining pangenome graphs to have a particular structure, such as Wheeler graphs [12], p-sortable graphs [7], elastic degenerate strings [1] or founder block graphs [19]. Unfortunately, merging reference sequences into a graph hides certain variations' tendencies to co-occur, known as *linkage disequilibrium* [30], and creates *chimeric* paths whose labels are not in any of the original sequences. Indexing and using such a graph can result in false-positive matches to these chimeric paths.

The more variations are represented, the noisier the graph becomes and the more possibilities there are for spurious matches. The number of false positives can be reduced by excluding rare variations, but sacrificing inclusivity for the sake of computational convenience goes against the spirit of pangenomics, and the pressure to include more genomes will probably force bioinformaticians to index all the variations. Moreover, excluding variations could be viewed as trading false positives for false negatives. Another approach is to filter out false positives by checking matches against the reference sequences represented as strings, but then the overall query time cannot be bounded in terms of the patterns and the true matches reported. Furthermore, the number of false positives will likely grow as the pangenome does.

Some researchers have eschewed using a pangenome graph altogether and indexed the genomes in the pangenome as a set of strings. This approach allowed

them to draw on a rich history of indexing compressible texts: the Burrows-Wheeler Transform [4] (BWT) and FM-indexes [10], for which Burrows, Ferragina and Manzini recently shared the Paris Kanellakis Award and which underpin Bowtie and BWA; RLCSA [20]; the r-index [13], subsampled r-index [5] and r-index-f [29]. Recently, Rossi et al. [32] and Boucher et al. [3] showed how, given a straight-line program with g rules for a text $T[1..n]$, they can build an $O(g+r)$ space index, where r is the number of runs in the BWT of T, with which they can find the maximal exact matches (MEMs) of a given pattern $P[1..m]$ with respect to T in $O(m \log n)$ time and list the occurrences of each MEM in constant time per occurrence. This result means they can index the pangenome compactly with no chance of false positives, find good seeds reasonably quickly, and list the occurrences of those seeds in constant time per occurrence.

The main practical problem with those results is that if there are thousands of genomes in the pangenome, then a MEM can occur thousands of times in those genomes, even if all those occurrences map to only one place in the standard single reference genome. This observation makes extending the seeds and combining the approximate matches of the reads much slower. In this paper, we show how we can combine Rossi et al.'s result with a pangenome graph such that we can still find seeds quickly, with no chance of false positives, but then report their non-chimeric occurrences in the graph in constant time per occurrence. Moreover, we put no constraints on the graph.

A set of genomes can be annotated so that for each character in the genomes, we know at which vertex in a pangenome graph that character occurs. Then, our idea is that if someone gives us a set of genomes and the corresponding annotation, we can store them in a small space so we can later quickly report for each seed its starting positions in the graph. The seeds of a read with respect to the set of genomes can be MEMs, but also f-MEMs [25, 35] (maximal substrings that occur at least f times in the genomes) or other kinds of substrings.

We can formalize this problem as follows: we want to store a text $T[1..n]$ and an assignment of tags to the characters of T such that we can preprocess a pattern $P[1..m]$ and then, given i and j, quickly return all the distinct tags labeling the first characters of the occurrences of $P[i..j]$ in T. In a pangenome, characters with long common contexts are more likely to have the same tag, so we consider the number of runs t in the list of tags sorted by their characters' positions in the Burrows-Wheeler Transform (BWT) of T.

Our contribution. In this paper, we show how, given a straight-line program with g rules for T, we can build an $O(g + r + t)$ space data structure, where r is the number of runs in the BWT of T, with which we can preprocess a pattern P in $O(m \log n)$ time and then return the k distinct tags for $P[i..j]$ in the optimal $O(k)$ time for any given i and j.

We call our data structure a *Wheeler map* since it resembles a Wheeler graph [12] but with less structure. One reason Wheeler graphs were introduced was to provide a model for alignment with a pangenome graph: we start with a string dataset, build a graphical representation, and index that graph; the graphical representation is inherently lossy but, to filter out chimeric matches,

we can verify matches against the original dataset. (Even before Wheeler graphs were defined, software for indexing variation graphs [15] used a procedure for making them Wheeler or almost Wheeler, falling back on unwinding the graph and indexing substrings when that procedure failed.) Our idea is to reverse that approach of indexing a graph and then filtering out false positives using the strings. Instead, we index the strings, and then map occurrences onto a graph — but without considering all the occurrences in the strings.

Some researchers (see, e.g., [31] and references therein) argue that having a graph index return matches not found in the original strings is a feature, not a bug, since it allows the index to find matches that can be obtained by recombination. As computer scientists, however, it is not our place to decide what combination of alleles are reasonable and which not, and so we should offer the option of indexing the datasets we are given and nothing else. Indexing the strings means we index all the variations they contain, so we can presumably capture most reasonable combinations by increasing the number of genomes in our dataset. Scaling to larger datasets is thus a solution for us, whereas it is a problem for graphical indexes, which tend to produce more false positives when they include all the variations in large datasets.

From Rossi et al. [32], we know r and g are reasonably small for the datasets in which we are most interested. To check that t is comparable, we computed it for the chromosome-19 component in a Minigraph-Cactus graph based on 90 human haplotypes from the Human Pangenome Reference Consortium [18]. This component was built from 1100 contigs with total length $n = 5{,}070{,}072{,}154$ and t was 208,649,680, almost 25 times smaller than n. For comparison, r was 71,512,609, just over 70 times smaller than n and not quite 3 times smaller than t.

Roadmap. In Section 2 we describe the basic concepts that will be used throughout the rest of the work, together with a preliminary method of computing the tags for the occurrences of a pattern P. In Section 3 we show how extended matching statistics can be computed $O(m \log n)$ time without the need for buffering that Rossi et al. [32] used, and extend the method for computing the tag statistics. In Section 4 we describe how the tag statistics together with range successor queries on the tag array can be used to get the k distinct tags for the occurrences of $P[i..j]$ in $O(\log^{\epsilon} t + k)$ time for any $\epsilon > 0$. Using more sophisticated techniques, we improve this time to the optimal $O(k)$ in Section 5. We conclude in Section 6 with some future work directions.

2 Preliminaries

Our model of computation throughout is the standard word-RAM with $\Theta(\log n)$-bit words. For the sake of brevity, we assume the reader is familiar with suffix arrays (SAs), the Burrows-Wheeler Transform (BWT), FM-indexes, LF-mapping and straight-line programs (SLPs); otherwise, we refer them to appropriate surveys [24, 26]. We recall only that $\mathrm{LCP}(S_1, S_2)$ denotes the length of the longest

	L				
	B	W	tag	BWT	context
1	0		9	T	$AGATACA
2			9	T	$GATACA
3		0	9	T	$GATTACA
4			9	T	$GATTAGA
5	0	0	10	A	$GATTAGAT
6	0	1	9	T	A$GATTAGA
7	1		4	T	ACAT$AGA
8		4	4	T	ACAT$GA
9	4		5	T	ACAT$GAT
10		1	5	T	AGAT$GAT
11			5	T	AGATA$GAT
12	5	8	0	$	AGATACAT
13	1		7	C	AT$AGATA
14			7	C	AT$GATA
15		2	7	C	AT$GATTA
16			7	G	AT$GATTA
17			7	G	ATA$GATTA

	L				
	B	W	tag	BWT	context
18	3		2	G	ATACAT$A
19			2	G	ATACAT$
20		2	2	G	ATTACAT$
21			2	G	ATTAGAT$
22			2	G	ATTAGATA$
23	0		6	A	CAT$AGAT
24			6	A	CAT$GAT
25		0	6	A	CAT$GATT
26			6	A	GAT$GATT
27			6	A	GATA$GATT
28	4		1	A	GATACAT$
29			1	$	GATACAT
30		3	1	$	GATTACAT
31			1	$	GATTAGAT
32			1	$	GATTAGATA

	L				
	B	W	tag	BWT	context
33	0		8	A	T$AGATAC
34			8	A	T$GATAC
35		1	8	A	T$GATTAC
36			8	A	T$GATTAG
37			8	A	TA$GATTAG
38	2		3	A	TACAT$AG
39		5	3	A	TACAT$G
40	5		4	T	TACAT$GA
41		2	4	T	TAGAT$GA
42			4	T	TAGATA$GA
43	1		3	A	TTACAT$G
44		3	3	A	TTAGAT$G
45			3	A	TTAGATA$G

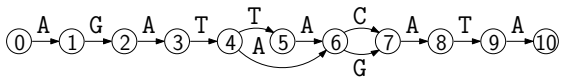

Fig. 1. Tables **(top)** for a set of toy genomes `GATTACAT$`, `AGATACAT$`, `GATACAT$`, `GATTAGAT$` and `GATTAGATA$` and a pangenome graph **(bottom)**. The BWT column shows the characters sorted by their contexts, which are the rest of the genomes (consider to be cyclic) and shown in the right column. The tag array is shown in the column to the left of the BWT, with each entry identifying the source in the pangenome graph of the edge labelled by the first character in the context in the same row (not the BWT character). The leftmost two columns (together called L) show the LCP values for the runs in the tag array, discussed in Section 3: column B contains the LCP value *between* each run in the tag array and the preceding run (so the LCP value at the beginning of the run), while column W contains the LCP value *within* each run (the length of the longest prefix common to all the contexts in the run). LCPs extend only up to and not including the terminators \$ because they are not searchable. Due to space constraints, the tables are displayed split into three pieces (at the end of runs).

common prefix of two strings S_1 and S_2 (which need not be lexicographically consecutive suffixes of a text), and of the bounds for Muthukrishnan's [23] classic document-listing data structure:

Theorem 1 (Muthukrishnan, [23, Thm. 3.1]). *Given an array $A[1..h]$, we can build an $O(h)$-space data structure with which, given i and j, we can return the k distinct elements in $A[i..j]$ in $O(k)$ time.*

In our model, each text suffix $T[i..]$ is labeled with a "tag", which can also be seen as labeling the position i. The tags of T are collected in a so-called "tag array"; see Figure 1.

Definition 1. *Let $T[1..n]$ be a labeled text, such that the label for the ith position is $T[i]$.*lab*. The tag array* $\mathrm{Tag}[1..n]$ *of T is then defined as* $\mathrm{Tag}[j] = T[SA[j]]$.lab.

We say that an occurrence $T[i..i + |P| - 1]$ of a pattern P in T is labeled by the tag that labels $T[i..]$. Consequently, the labels of all the occurrences of P in T are listed in $\mathrm{Tag}[s..e]$, where $\mathrm{SA}[s..e]$ is the suffix array interval for P. For example, in Figure 1 the range for $P = \mathtt{A}$ is $\mathrm{SA}[6..22]$, and $\mathrm{Tag}[6..22]$ contains the tags 9, 4, 5, 0, 7, and 2. Those are the labels with which P appears in the graph.

For convenience, we first extend the standard definition of matching statistics to include the lexicographic ranks of the suffixes of T starting with the occurrences we consider, and then further extend it to mention the tag array.

Definition 2. *The* extended matching statistics *of a pattern* $P[1..m]$ *with respect to a text* $T[1..n]$ *are an array* $\mathrm{XMS}[1..m+1]$ *of* $(\mathrm{len}, \mathrm{pos}, \mathrm{rank})$ *triples such that*

- $\mathrm{XMS}[i].\mathrm{len}$ *is the length of the longest prefix of* $P[i..m]$ *that occurs in* T*,*
- $\mathrm{XMS}[i].\mathrm{pos}$ *is the starting position of one occurrence of* $P[i..i+\mathrm{XMS}[i].\mathrm{len}-1]$ *in* T*,*
- $\mathrm{XMS}[i].\mathrm{rank}$ *is the lexicographic rank of* $T[\mathrm{XMS}[i].\mathrm{pos}..n]$ *among the suffixes of* T*.*

We emphasize that we expect the tag array to have long runs of equal consecutive symbols. The following definition considers those runs in the process of matching P in T.

Definition 3. *The* tag statistics *of a pattern* $P[1..m]$ *with respect to a text* $T[1..n]$ *and its tag array* $\mathrm{Tag}[1..n]$ *are an array* $\mathrm{TS}[1..m+1]$ *of* $(\mathrm{len}, \mathrm{pos}, \mathrm{rank}, \mathrm{run}, \mathrm{up}, \mathrm{down})$ *sextuples such that* $\mathrm{TS}[i].\mathrm{len}$*,* $\mathrm{TS}[i].\mathrm{pos}$ *and* $\mathrm{TS}[i].\mathrm{rank}$ *are the same as in the* XMS *array and*

- $\mathrm{TS}[i].\mathrm{run}$ *is the index of the run* $\mathrm{Tag}[u..d]$ *in the tag array that contains position* $\mathrm{TS}[i].\mathrm{rank}$*,*
- $\mathrm{TS}[i].\mathrm{up} = \mathrm{LCP}(P[i..m], T[\mathrm{SA}[u]..n])$*,*
- $\mathrm{TS}[i].\mathrm{down} = \mathrm{LCP}(P[i..m], T[\mathrm{SA}[d]..n])$*.*

Finally, although we know of no previous work specifically addressing tag arrays, we note a solution that follows directly from the work by Mäkinen et al. [21]:

Theorem 2 (Mäkinen et al., [21, Thm 17.]). *Given a text* $T[1..n]$ *whose BWT has* r *runs, we can build an* $O(r)$*-space data structure called RLBWT such that later, given a pattern* $P[1..m]$*, we can return the lexicographic range of suffixes of* T *starting with* P *in* $O(m \log\log n)$ *time.*

Corollary 1. *Given a text* $T[1..n]$ *whose BWT has* r *runs, and a tag array with* t *runs, we can build an* $O(r + t)$*-space data structure such that later, given a pattern* $P[1..m]$*, we can return the* k *distinct tags of* P*'s occurrences in* T *in* $O(m \log\log n + k)$ *time.*

Proof. We store an $O(r)$-space RLBWT for T, an $O(t)$-space predecessor structure storing where the runs start in Tag, and an $O(t)$-space instance of Muthukrishnan's data structure from Theorem 1 for the array $A[1..t]$ obtained from Tag by replacing each run by a single copy of the same tag. Given P, we first use the RLBWT to find the lexicographic range $\mathrm{SA}[s..e]$ of suffixes of T starting with P, in $O(m \log\log n)$ time. We then use predecessor queries to find the range $A[s'..e']$ of the tag run indices overlapping $\mathrm{Tag}[s..e]$, in $O(\log\log n)$ time. Finally, we use Muthukrishnan's data structure to report the distinct tags in $A[s'..e']$, in $O(k)$ time. □

Our main concern with Corollary 1 is that if we want the distinct tags for a set of substrings of P that can overlap—such as the maximal exact matches (MEMs) of P with respect to T—and we apply this corollary to each one, then we can use $\Omega(m^2)$ total time even when the number of tags we return is small. Our plan is then to preprocess P in a first stage, so that in a second stage we can more quickly answer (many) questions about substrings of the form $P[i..j]$.

3 Computing tag statistics

We rely on results about straight-line programs (SLPs), which we can encapsulate in the following lemma.

Lemma 1. *Given an SLP with g rules for $T[1..n]$, in $O(n \log n)$ expected time we can build an $O(g)$-space data structure with which we can preprocess any pattern $P[1..m]$ in $O(m)$ time such that later, given i, j and q, we can return $\mathrm{LCP}(P[i..j], T[q..n])$ in $O(\log n)$ time and with no chance of error as long as $P[i..j]$ occurs somewhere in T.*

Proof. Bille et al. [2] showed how to build, in $O(n \log n)$ expected time, a Karp-Rabin hash function with no collisions between substrings of T. If $S = S' \cdot S''$ and we have the hashes of two of those strings, we can compute the hash of the third in constant time, as soon as we store some precomputed values that can also be maintained in constant time (see, e.g., [27]).

If necessary, we use Ganardi et al.'s [14] construction to balance the SLP such that it has $O(g)$ rules and height $O(\log n)$. We then label each symbol x in the SLP with the length and hash of x's expansion. This takes $O(g)$ time because we compute in constant time the hash of the left-hand side of a rule from those of the right-hand side.

When P arrives, we compute the hashes of its suffixes in $O(m)$ total time. The hash of any $P[i..j]$ can then be computed in constant time from the hashes of $P[i..m]$ and $P[j+1..m]$.

Given i, j and q, we descend to the qth leaf of the parse tree in $O(\log n)$ time. We then re-ascend toward the root in $O(\log n)$ time, keeping track of the length and hash of $T[q..e]$, where e is the index of the rightmost leaf in the subtree of the node we are currently visiting.

When we reach a node such that $T[q..e]$ is either longer than $P[i..j]$ or the hash of $T[q..e]$ does not match the hash of the corresponding prefix of $P[i..j]$, we re-descend in $O(\log n)$ time. At each step in the re-descent, we go left if $T[q..e]$ is either longer than $P[i..j]$ or the hash of $T[q..e]$ does not match the hash of the corresponding prefix of $P[i..j]$, where e is now the index of the rightmost leaf in the subtree of the left child. Otherwise, we go right.

We then find $\mathrm{LCP}(P[i..j], T[q..n])$ in $O(\log n)$ time. As long as $P[i..j]$ occurs somewhere in T, no hash of a prefix of $P[i..j]$ collides with the hash of a different substring of T, so we have no chance of error. □

We now show how to preprocess the tag array. Let $U[1..t]$ and $D[1..t]$ be the arrays such that $U[q]$ and $D[q]$ are the indices of the first and last tags, respectively, in the qth run in the tag array. Let $W[1..t]$ be the array with

$$\begin{aligned} W[q] &= \min_{U[q]+1 \le p \le D[q]} \{\mathrm{LCP}(T[\mathrm{SA}[p-1]..n], T[\mathrm{SA}[p]..n])\} \\ &= \mathrm{LCP}(T[\mathrm{SA}[U[q]]..n], T[\mathrm{SA}[D[q]]..n]) \end{aligned}$$

for $1 \le q \le t$, and let $B[1..t-1]$ be the array with

$$B[q] \;=\; \mathrm{LCP}(T[\mathrm{SA}[D[q]]..n], T[\mathrm{SA}[U[q+1]]..n])$$

for $1 \le q \le t-1$—so $W[q]$ is the LCP computed *within* run q and $B[q]$ is the LCP computed *between* runs q and $q+1$. Finally, let

$$L[0..2t] \;=\; 0, W[1], B[1], W[2], B[2], \ldots, W[t-1], B[t-1], W[t], 0\,.$$

From now on we will only use L, and we will not refer to U, D, W or B again. We recall that Figure 1 shows the L array on an example text.

We now describe our preprocessing of the pattern. Our results in this section can be viewed as mainly extending Rossi et al.'s [32] work on computing (extended) matching statistics to computing tag statistics:

Theorem 3 (cf. [32]). *Given an SLP with g rules for a text $T[1..n]$ whose BWT has r runs, we can build an $O(g+r)$-space data structure such that later, given a pattern $P[1..m]$, we can compute the extended matching statistics* XMS *of P with respect to T in $O(m \log n)$ time.*

Proof. We apply Lemma 1 to the SLP to obtain an $O(g)$-space LCP data structure with $O(\log n)$ query time. We also store $\mathrm{SA}[u]$ and $\mathrm{SA}[d]$, for each run $\mathrm{BWT}[u..d]$, in an $O(r)$-space data structure supporting predecessor and successor queries on the keys u and d. Finally, we use the $O(r)$-space RLBWT of Theorem 2, which can also compute any $\mathrm{BWT}[j]$ and $\mathrm{LF}[j]$. These functions and the predecessor queries can run in $O(\log \log n)$ time, but $O(\log n)$ time is enough for our purposes.

As usual, for technical convenience we add to T a special symbol $T[n+1] = \$$ that is lexicographically smaller than all the other symbols in T (and in potential patterns P). This implies $\mathrm{BWT}[1] = \$$. For a start, then, considering

$P[m+1..m] = \epsilon$, we set $\mathrm{XMS}[m+1].\mathrm{len} = 0$, $\mathrm{XMS}[m+1].\mathrm{rank} = 1$ and $\mathrm{XMS}[m+1].\mathrm{pos} = n+1$.

Now, suppose we have already computed the suffix $\mathrm{XMS}[i+1..m+1]$ of the extended matching statistics and want to compute $\mathrm{XMS}[i]$. If $\mathrm{BWT}[\mathrm{XMS}[i+1].\mathrm{rank}] = P[i]$ then

$$\begin{aligned}\mathrm{XMS}[i].\mathrm{len} &= \mathrm{XMS}[i+1].\mathrm{len} + 1\,,\\ \mathrm{XMS}[i].\mathrm{pos} &= \mathrm{XMS}[i+1].\mathrm{pos} - 1\,,\\ \mathrm{XMS}[i].\mathrm{rank} &= \mathrm{LF}[\mathrm{XMS}[i+1].\mathrm{rank}]\,.\end{aligned}$$

Otherwise, let $\mathrm{BWT}[u]$ and $\mathrm{BWT}[d]$ be the occurrences of $P[i]$ immediately preceding and following $\mathrm{BWT}[\mathrm{XMS}[i+1].\mathrm{rank}]$. We find u and d with predecessor/successor queries.

By the definition of the BWT, at least one of $T[\mathrm{SA}[u]..n]$ and $T[\mathrm{SA}[d]..n]$ has the longest common prefix with $P[i+1..m]$ of any suffix of T preceded by a copy of $P[i]$. Since $\mathrm{BWT}[u]$ is the last character in a run and $\mathrm{BWT}[d]$ is the first character in a run, we have $\mathrm{SA}[u]$ and $\mathrm{SA}[d]$ stored. Therefore, we can compute

$$\begin{aligned}\ell_u &= \mathrm{LCP}(P[i+1..i+\mathrm{XMS}[i+1].\mathrm{len}-1], T[\mathrm{SA}[u]..n])\,,\\ \ell_d &= \mathrm{LCP}(P[i+1..i+\mathrm{XMS}[i+1].\mathrm{len}-1], T[\mathrm{SA}[d]..n])\,,\end{aligned}$$

in $O(\log n)$ time, since $P[i+1..i+\mathrm{XMS}[i+1].\mathrm{len}-1]$ occurs in T, with no chance of error.

If $\ell_u \geq \ell_d$ then

$$\begin{aligned}\mathrm{XMS}[i].\mathrm{len} &= \ell_u + 1\,,\\ \mathrm{XMS}[i].\mathrm{pos} &= \mathrm{SA}[u] - 1\,,\\ \mathrm{XMS}[i].\mathrm{rank} &= \mathrm{LF}[u]\,,\end{aligned}$$

and, symmetrically, if $\ell_u < \ell_d$ then

$$\begin{aligned}\mathrm{XMS}[i].\mathrm{len} &= \ell_d + 1\,,\\ \mathrm{XMS}[i].\mathrm{pos} &= \mathrm{SA}[d] - 1\,,\\ \mathrm{XMS}[i].\mathrm{rank} &= \mathrm{LF}[d]\,.\end{aligned}$$

□

Corollary 2. *Suppose we are given an SLP with g rules for a text $T[1..n]$ whose* BWT *has r runs, and a tag array for T with t runs. Then we can build an $O(g+r+t)$-space data structure such that later, given a pattern $P[1..m]$, we can compute the tag statistics of P with respect to T in $O(m \log n)$ time.*

Proof. We store an $O(t)$-space predecessor data structure on the starting positions of the runs in Tag. For each run $\mathrm{Tag}[u..d]$, we also store $\mathrm{SA}[u]$ and $\mathrm{SA}[d]$. Given P, we start by applying Theorem 3 to compute the extended matching statistics $\mathrm{XMS}[1..m+1]$ of P with respect to T in $O(m \log n)$ time. For

$1 \leq i \leq m+1$, we then set

$$\begin{aligned} \mathrm{TS}[i].\mathrm{len} &= \mathrm{XMS}[i].\mathrm{len}\,, \\ \mathrm{TS}[i].\mathrm{pos} &= \mathrm{XMS}[i].\mathrm{pos}\,, \\ \mathrm{TS}[i].\mathrm{rank} &= \mathrm{XMS}[i].\mathrm{rank}\,, \end{aligned}$$

and $\mathrm{TS}[i].\mathrm{run}$ to the index of the run $\mathrm{Tag}[u..d]$ in the tag array containing position $\mathrm{TS}[i].\mathrm{rank}$ (computed with a predecessor query). Further, we use the LCP data structure to compute

$$\begin{aligned} \mathrm{TS}[i].\mathrm{up} &= \mathrm{LCP}(P[i..m], T[\mathrm{SA}[u]..n])\,, \\ \mathrm{TS}[i].\mathrm{down} &= \mathrm{LCP}(P[i..m], T[\mathrm{SA}[d]..n])\,. \end{aligned}$$

This also takes a total of $O(m \log n)$ time. □

4 Using tag statistics

Once we have the tag statistics of P with respect to T, we no longer need Lemma 1, or even the SA samples or BWT, to find out which tags label the occurrences of any $P[i..j]$. We use Muthukrishnan's document-listing data structure in the same way as in the proof of Corollary 1: once we know which runs in the tag array overlap the BWT interval for $P[i..j]$, we use Muthukrishnan's structure to list the k distinct tags in $O(k)$ time. In this section we explain how we find which runs in the tag array overlap the BWT interval for $P[i..j]$, without computing the interval itself (which we do not know how to do quickly in $O(g+r+t)$ space).

Lemma 2. *Suppose we are given a text $T[1..n]$ and a tag array for T with t runs. Then, for any constant $\epsilon > 0$, we can build an $O(g+t)$-space data structure such that later, given the tag statistics of a pattern $P[1..m]$ with respect to T and i and j, we can find which runs in the tag array overlap the BWT interval for $P[i..j]$ in $O(\log^{\epsilon} t)$ time.*

Proof. We store $O(t)$-space range-predecessor/successor data structures over L with $O(\log^{\epsilon} t)$ query time [28] (we call them collectively range-successor queries at times). With these data structures and given values ℓ and q, we can find the largest position of a value less than ℓ in $L[0..2q-2]$ and the smallest position of a value less than ℓ in $L[2q..2t]$ in $O(\log^{\epsilon} t)$ time. We note that ϵ can be made arbitrarily small for the cost of a larger constant multiplying the space consumption.

Given the tag statistics $\mathrm{TS}[1..m+1]$ of P with respect to T and i and j, we can check that $P[i..j]$ occurs in T at all by verifying that $\mathrm{TS}[i].\mathrm{len} \geq j-i+1$. Assuming it does, we can look up the index $q = \mathrm{TS}[i].\mathrm{run}$ of the run in the tag array containing $\mathrm{Tag}[\mathrm{TS}[i].\mathrm{rank}]$ and we can check in constant time whether

$$\begin{aligned} \mathrm{TS}[i].\mathrm{up} &\geq j-i+1\,, \\ \mathrm{TS}[i].\mathrm{down} &\geq j-i+1\,. \end{aligned}$$

If $\mathrm{TS}[i].\mathrm{up} < j-i+1$ then $L[2q-1] < j-i+1$ (note $L[2q-1]$ is the LCP within run q) and run q is the first in the tag array to overlap the BWT interval for $P[i..j]$. Otherwise, we use a range-predecessor query to find the largest position in $L[0..2q-2]$ with value less than $j-i+1$. This tells us the first run in the tag array to overlap the BWT interval for $P[i..j]$: If the range-predecessor query returns p, then the index of this first run is $1+\lfloor p/2 \rfloor$; the run is covered completely if p is even and partially if p is odd.

Symmetrically, if $\mathrm{TS}[i].\mathrm{down} < j-i+1$ then $L[2q-1] < j-i+1$ and run q is the last one in the tag array to overlap the BWT interval for $P[i..j]$. Otherwise, we use a range-successor query to find the smallest position in $L[2q..2t]$ of a value less than $j-i+1$, which tells us the last run in the tag array to overlap the BWT interval for $P[i..j]$. If the range-successor query returns p, then the index of this last run is $\lceil p/2 \rceil$, and it is covered completely iff p is even.

Notice we never compute the BWT interval for $P[i..j]$. □

Corollary 3. *Suppose we are given an SLP with g rules for a text $T[1..n]$ whose* BWT *has r runs, and a tag array for T with t runs. Then, for any constant $\epsilon > 0$, we can build an $O(g+r+t)$-space data structure with which we can preprocess any pattern $P[1..m]$ in $O(m \log n)$ time such that later, given i and j, we can return the k distinct tags labeling occurrences of $P[i..j]$ in T in $O(\log^{\epsilon} t + k)$ time.*

Proof. We store instances of the data structures from (i) Corollary 2, (ii) Lemma 2, and (iii) Corollary 1. Given P, we use the data structures (i) to compute the tag statistics of P with respect to T in $O(m \log n)$ time. Given i and j, we use the data structures (ii) to find the indices s and e of the runs in Tag that are contained in or overlap the BWT range of $P[i..j]$, in time $O(\log^{\epsilon} t)$. Finally, using the array $A[1..t]$ (iii) we run Muthukrishnan's algorithm on $A[s..e]$ to find the k distinct tags labeling occurrences of $P[i..j]$ in T, in $O(k)$ time. □

5 Optimal-time tag reporting

The time in Corollary 3 for reporting the k distinct tags labeling occurrences of $P[i..j]$ in T—that is, $O(\log^{\epsilon} t + k)$—is optimal if $k \in \Omega(\log^{\epsilon} t)$. We do not know k in advance, however, and if we always want optimal reporting time we cannot afford range-successor queries right away.

We start with an important property of the ranges we find in L in the proof of Corollary 3.

Lemma 3. *Let q, q' be positions in L with respective thresholds ℓ, ℓ', from which the predecessor/successor queries result in ranges $[u,d], [u',d']$. Then $[u,d], [u',d']$ can be equal, disjoint or nested, but cannot overlap.*

Proof. Consider $L[u..d]$ ($L[u'..d']$), which is as large as possible around q (q') not containing any values less than ℓ (ℓ'), and that $u < u' \leq d < 2t$. It follows that $\ell \leq L[u'-1] < \ell'$, therefore, since $L[d+1] < \ell < \ell'$, it must be $d' \leq d$, so $L[u'..d']$ is contained in $L[u..d]$. The case $u \leq d' < d$ is analogous. □

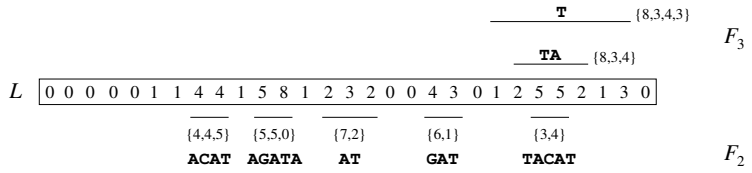

Fig. 2. The array L of Figure 1 and the sets of segments forming F_3 (above) and F_2 (below). The larger range of F_3 contains the smaller, and thus they represent the same set of tags.

Consider the distinct ranges we can find in L such that the corresponding range in Tag (including both contained and overlapped runs) contains k' distinct tags, for some k'. If two of these ranges in L are nested, then their corresponding ranges in Tag contain exactly the same k' distinct tags—possibly with different multiplicities, but that does not concern us here. Let $F_{k'}$ be the $O(t)$-bit balanced-parentheses representation [22] of these distinct ranges in L, where every range is an ancestor of those it contains. With $O(t)$ further bits, we can find in $O(1)$ time the lowest node of $F_{k'}$ that contains any given entry $L[q]$ [33, Sec. 4.1]. Figure 2 gives an example.

While querying the data structure from Lemma 2, if we somehow guess correctly that our range-successor queries will return a range in L whose corresponding range in Tag contains exactly k' distinct tags, then we can replace those range-successor queries by the constant-time method described above to find the corresponding node in $F_{k'}$.

This node may correspond to a range nested strictly inside the one we would obtain from the range-successor queries but, as we noted above, that makes no difference to our final answer. In fact, the node of $F_{k'}$ we find has the smallest range—corresponding to the largest value of $j - i + 1$—we could obtain from our range-successor queries, while still returning k' distinct tags. If we store an $O(t)$-bits range-minimum data structure [11] over L—which we can reuse for all values of k'—then we can find that largest value $j - i + 1$ in constant time, as it is the minimum value of L in the range.

Of course, we cannot assume we will guess correctly the number k' of distinct tags we will eventually return. Instead, we keep an $O(t)$-bits representation $F_{k'}$ for every $k' \leq \lg^\epsilon t$, which takes $O\left(\frac{t \lg^\epsilon t}{\log t}\right) \subset O(t)$ space. We query $F_1, F_2, F_3, \ldots, F_{\lg^\epsilon t}$ in turn, using constant time for each. If, for some $F_{k'}$, the range-minimum data structure returns a value smaller than $j - i + 1$, then we know that $P[i..j]$ is labeled by $k = k' - 1$ distinct tags, so we use the formulas of Section 4 to convert the range in L given by F_k to a range $A[s..e]$, and use Muthukrishnan's algorithm (Corollary 1) to return the distinct tags in $A[s..e]$. Otherwise, after we query $F_{\lg^\epsilon t}$, we know that $k > \lg^\epsilon t$, so we can perform the range-successor queries safely as in Section 4. In both cases, we use $O(k)$ total time.

Theorem 4. *Suppose we are given an SLP with g rules for a text $T[1..n]$ whose* BWT *has r runs, and a tag array for T with t runs. Then we can build an $O(g + r + t)$-space data structure that can preprocess any pattern $P[1..m]$ in $O(m \log n)$ time such that later, given i and j, it returns the k distinct tags labeling occurrences of $P[i..j]$ in T in optimal $O(k)$ time.*

6 Discussion and future work

This paper lays out the theoretical basis for Wheeler maps. We have shown how using compressed space, we can preprocess a pattern P such that later, given any i and j, we report the distinct tags labeling the occurrences of $P[i..j]$ in the optimal constant time per tag reported. To the best of our knowledge, Wheeler maps are the first data structure allowing for an efficient tag listing of subpatterns. Further results on prioritizing and constraining the query tag frequencies will be included in the extended version of this article.

As a future work, we plan to address the question of whether mixing Wheeler graphs with Wheeler maps — to allow some kinds of recombinations while excluding others — is useful and viable. Besides pangenomics, we plan to explore the versatile nature of Wheeler maps and look for other applications. We also believe that for certain cases we can prove analytical bounds on the number of runs in the tag array by relating them to the repetitiveness of the input.

We are now investigating our approach experimentally. Together with the full implementations of the data structures described here, we also still need efficient algorithms for extracting tag arrays from pangenome graphs for large genomic datasets, and good compression schemes for those tag arrays. A tag could contain a lot of information, so representing it explicitly for every run of that tag in the tag array might be very wasteful. It is likely more space-efficient to store each distinct tag only once, separated from the tag array by one or more levels of indirection. Once we can build and store Wheeler maps well in practice, we intend to integrate them into current pangenomics pipelines.

References

1. Giulia Bernardini, Nadia Pisanti, Solon P Pissis, and Giovanna Rosone. Pattern matching on elastic-degenerate text with errors. In *Proc. 24th International Symposium on String Processing and Information Retrieval (SPIRE)*, pages 74–90, 2017.
2. Philip Bille, Inge Li Gørtz, Patrick Hagge Cording, Benjamin Sach, Hjalte Wedel Vildhøj, and Søren Vind. Fingerprints in compressed strings. *Journal of Computer and System Sciences*, 86:171–180, 2017.
3. Christina Boucher, Travis Gagie, I Tomohiro, Dominik Köppl, Ben Langmead, Giovanni Manzini, Gonzalo Navarro, Alejandro Pacheco, and Massimiliano Rossi. PHONI: Streamed matching statistics with multi-genome references. In *Proc. 31st Data Compression Conference (DCC)*, pages 193–202, 2021.
4. Michael Burrows and David Wheeler. A block-sorting lossless data compression algorithm. In *Digital SRC Research Report*. Citeseer, 1994.

5. Dustin Cobas, Travis Gagie, and Gonzalo Navarro. A Fast and Small Subsampled R-Index. In *Proc. 32nd Annual Symposium on Combinatorial Pattern Matching (CPM)*, pages 13:1–13:16, 2021.
6. Computational Pan-Genomics Consortium. Computational pan-genomics: status, promises and challenges. *Briefings in bioinformatics*, 19(1):118–135, 2018.
7. Nicola Cotumaccio and Nicola Prezza. On indexing and compressing finite automata. In *Proc. 32nd ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 2585–2599, 2021.
8. Elie Dolgin. Scientists unveil a more diverse human genome, 2023. [Online; accessed 3-January-2024].
9. Massimo Equi, Veli Mäkinen, and Alexandru I. Tomescu. Graphs cannot be indexed in polynomial time for sub-quadratic time string matching, unless SETH fails. In *Proc. Theory and Practice of Computer Science (SOFSEM)*, pages 608–622, 2021.
10. Paolo Ferragina and Giovanni Manzini. Indexing compressed text. *J. ACM*, 52(4):552–581, jul 2005.
11. Johannes Fischer and Volker Heun. Space-efficient preprocessing schemes for range minimum queries on static arrays. *SIAM Journal on Computing*, 40(2):465–492, 2011.
12. Travis Gagie, Giovanni Manzini, and Jouni Sirén. Wheeler graphs: A framework for BWT-based data structures. *Theoretical Computer Science*, 698:67–78, 2017.
13. Travis Gagie, Gonzalo Navarro, and Nicola Prezza. Optimal-time text indexing in BWT-runs bounded space. In *Proc. 29th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1459–1477, 2018.
14. Moses Ganardi, Artur Jeż, and Markus Lohrey. Balancing straight-line programs. *Journal of the ACM*, 68(4):1–40, 2021.
15. 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.
16. Ben Langmead and Steven L Salzberg. Fast gapped-read alignment with Bowtie 2. *Nature Methods*, 9(4):357–359, 2012.
17. Heng Li and Richard Durbin. Fast and accurate short read alignment with Burrows–Wheeler transform. *Bioinformatics*, 25(14):1754–1760, 2009.
18. Wen-Wei Liao, Mobin Asri, Jana Ebler, Daniel Doerr, Marina Haukness, Glenn Hickey, Shuangjia Lu, Julian K Lucas, Jean Monlong, Haley J Abel, et al. A draft human pangenome reference. *Nature*, 617(7960):312–324, 2023.
19. Veli Mäkinen, Bastien Cazaux, Massimo Equi, Tuukka Norri, and Alexandru I. Tomescu. Linear time construction of indexable founder block graphs. In *20th International Workshop on Algorithms in Bioinformatics (WABI)*, pages 7:1–7:18, 2020.
20. Veli Mäkinen and Gonzalo Navarro. Succinct suffix arrays based on run-length encoding. In *Combinatorial Pattern Matching: 16th Annual Symposium, CPM 2005, Jeju Island, Korea, June 19-22, 2005. Proceedings 16*, pages 45–56. Springer, 2005.
21. Veli Mäkinen, Gonzalo Navarro, Jouni Sirén, and Niko Välimäki. Storage and retrieval of highly repetitive sequence collections. *Journal of Computational Biology*, 17(3):281–308, 2010.
22. J. Ian Munro and Venkatesh Raman. Succinct representation of balanced parentheses and static trees. *SIAM Journal on Computing*, 31(3):762–776, 2001.

23. Shanmugavelayutham Muthukrishnan. Efficient algorithms for document retrieval problems. In *Proc. 13th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 657–666, 2002.
24. G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM Computing Surveys*, 39(1):article 2, 2007.
25. Gonzalo Navarro. *Compact data structures: A practical approach*. Cambridge University Press, 2016.
26. Gonzalo Navarro. Indexing highly repetitive string collections, part II: Compressed indexes. *ACM Computing Surveys*, 54(2):article 26, 2021.
27. Gonzalo Navarro and Nicola Prezza. Universal compressed text indexing. *Theoretical Computer Science*, 762:41–50, 2019.
28. Yakov Nekrich and Gonzalo Navarro. Sorted range reporting. In *Proc. 13th Scandinavian Symposium on Algorithmic Theory (SWAT)*, pages 271–282, 2012.
29. Takaaki Nishimoto and Yasuo Tabei. Optimal-time queries on BWT-runs compressed indexes. In *Proc. International Colloquium on Automata, Languages, and Programming (ICALP)*, pages 101:1–101:15, 2021.
30. David E. Reich, Michele Cargill, Stacey Bolk, James Ireland, Pardis C. Sabeti, Daniel J. Richter, Thomas Lavery, Rose Kouyoumjian, Shelli F Farhadian, Ryk Ward, et al. Linkage disequilibrium in the human genome. *Nature*, 411(6834):199–204, 2001.
31. Nicola Rizzo, Manuel Cáceres, and Veli Mäkinen. Finding maximal exact matches in graphs. In *Proc. Workshop on Algorithms in Bioinformatics (WABI)*, pages 10:1–10:17, 2023.
32. Massimiliano Rossi, Marco Oliva, Ben Langmead, Travis Gagie, and Christina Boucher. MONI: A pangenomic index for finding maximal exact matches. *Journal of Computational Biology*, 29(2):169–187, 2022.
33. Luís M. S. Russo, Gonzalo Navarro, and Arlindo Oliveira. Fully-compressed suffix trees. *ACM Transactions on Algorithms*, 7(4):article 53, 2011.
34. Ida Emilie Steinmark. The human genome needs updating. but how do we make it fair?, 2023. [Online; accessed 3-January-2024].
35. Igor Tatarnikov, Ardavan Shahrabi Farahani, Sana Kashgouli, and Travis Gagie. MONI can find k-MEMs. In *Proc. 34th Annual Symposium on Combinatorial Pattern Matching (CPM)*, pages 26:1–26:14, 2023.