PFP Compressed Suffix Trees*

Christina Boucher† Ondřej Cvacho‡ Travis Gagie§ Jan Holub‡ Giovanni Manzini¶
Gonzalo Navarro‖ Massimiliano Rossi†

Abstract

Prefix-free parsing (PFP) was introduced by Boucher et al. (2019) as a preprocessing step to ease the computation of Burrows-Wheeler Transforms (BWTs) of genomic databases. Given a string S, it produces a dictionary D and a parse P of overlapping phrases such that BWT(S) can be computed from D and P in time and workspace bounded in terms of their combined size $|\text{PFP}(S)|$. In practice D and P are significantly smaller than S and computing BWT(S) from them is more efficient than computing it from S directly, at least when S is the concatenation of many genomes. In this paper, we consider PFP(S) as a *data structure* and show how it can be augmented to support *full suffix tree functionality*, still built and fitting within $O(|\text{PFP}(S)|)$ space. This entails the efficient computation of various primitives to simulate the suffix tree: computing a longest common extension (LCE) of two positions in S; reading any cell of its suffix array (SA), of its inverse (ISA), of its BWT, and of its longest common prefix array (LCP); and computing minima over ranges and next/previous smaller value queries over the LCP. Our experimental results show that the PFP suffix tree can be efficiently constructed for very large repetitive datasets and that its operations perform competitively with other compressed suffix trees that can only handle much smaller datasets.

*The authors thank Manuel Cáceres for help with the code of the block-tree compressed suffix tree. CB and MR funded by the National Science Foundation (NSF) IIS (Grant No. 1618814), IIBR (Grant No. 2029552) and National Institutes of Health (NIH) R01 (Grant No. HG011392). TG and JH funded by OP VVV project Research Center for Informatics (no. CZ.02.1.01/0.0/0.0/16_019/0000765). GM funded by PRIN grant 2017WR7SHH. GN funded by ANID Basal Funds FB0001 and Fondecyt Grant 1-200038, Chile.

†Department of Computer and Information Science and Engineering, University of Florida, Gainesville, Florida, USA. {christinaboucher, rossi.m}@ufl.edu

‡Department of Theoretical Computer Science, Czech Technical University in Prague, Czech Republic. {cvachond, jan.holub}@fit.cvut.cz

§Faculty of Computer Science, Dalhousie University, Halifax, Canada. travis.gagie@gmail.com

¶Department of Science and Technological Innovation, University of Eastern Piedmont, Alessandria, Italy. giovanni.manzini@uniupo.it

‖CeBiB — Center for Biotechnology and Bioengineering, Department of Computer Science, University of Chile, Chile. gnavarro@dcc.uchile.cl

1 Introduction

The emergence of large genome repositories offers a new world of opportunities for research and development in biology and medicine, but they also pose serious performance challenges to the computational infrastructure, not only in terms of the time complexity of the sequence searching and mining problems to solve, but also in terms of the sheer memory space needed to simply store the data. Computer memories are getting larger every day, of course, but not as quickly as genomic databases: the 1000 Genomes Project Consortium announced the sequencing of 1092 human genomes in 2012 and Genomics England announced the sequencing of 100K human genomes in 2018, significantly outpacing Moore's Law. Indeed, storing the raw data is not as challenging as storing the appropriate data structures that allow us solving complex problems on the sequences within reasonable time.

Suppose we want to index a thousand human genomes in such a way that we can support standard bioinformatics tasks such as DNA sequence read alignment (see [36]). For example, given a sequence read, we might want to determine which of its substrings occur in the database, and where. Finding the maximal substrings of the read that occur in the database is an important step in the seed-and-extend approach to sequence read alignment. This is only one of the wealth of problems that can be solved with *suffix trees* [39], one of the most powerful data structures in stringology [2, 8] and bioinformatics [15, 22].

Suffix trees can be built in linear time and space [39, 25, 38]. In practice, however, they require more than the compacted sequence itself. One human genome, which is easily encoded in less than 800 MB, requires about 60 GB to store a classical implementation of its suffix tree. This is already challenging because interesting suffix tree algorithms require a lot of random access to the data structure, and becomes totally infeasible if we aim to handle thousands of genomes.

This limitation has been sidestepped by various

compressed suffix tree (CST) representations, which simulate the suffix tree functionality within a space close to that of the sequence [33, 12, 10, 32, 28, 14]. Some recent variants are aimed at exploiting the repetitiveness that arises in repositories of genomes of the same species [1, 27, 7, 13]. Such representations make it feasible to maintain and operate in main memory the suffix tree of large genome collections.

Even these compressed representations, however, do not really solve the problem of handling very large repositories. As Ferragina et al. [9] pointed out, "to use [an index] one must first *build* it!" The construction of the current CSTs still requires a lot of main memory space — at least 34 times the input sequence, on our experiments —, even if the final product is much smaller. This inability to be built within small space is the key limitation to scale up the power of this versatile data structure to the large sequence repositories where they should be used.

Significant progress towards solving the construction problem was made by Boucher et al. [5] and Kuhnle et al. [20], who introduced a preprocessing step called *prefix-free parsing* (PFP). PFP compresses the sequence collection in such a way that its Burrows-Wheeler Transform (BWT) [6] can be computed directly in run-length compressed form, which is known to be a very compact representation on highly repetitive genome collections [23]. Recent compressed indexes like the r-index [13] can then be built from the run-length compressed BWT.

The r-index simulates part of the functionality of a *suffix array* [24], which is a key component of the suffix tree. It can, for example, determine whether the whole read appears in the sequence collection, but cannot efficiently find which of its maximal substrings occur. A suffix tree requires some additional components in order to be fully functional. In fact, Gagie et al. [13] showed how to add such components to the r-index, increasing its size by a factor logarithmic in the length of the sequence, but their design is complex and has not been implemented.

Fischer et al. [12] show that a CST can be simulated if we implement the following primitives: access to the suffix array (SA), to its inverse permutation (ISA), to the BWT, to the longest common prefix array (LCP), and some sophisticated operations on the LCP array. In this paper we show that PFP can be viewed as a *data structure* by itself, which supports the required primitives. Although the resulting CST is not as small as others, its construction time and peak memory are much smaller and thus can be built for very large datasets. For example, the PFP data structures can be built for 1000 distinct variants of human chromosome 19 in slightly more than 1 hour using 54 GB of internal memory, which is almost the size of the raw data. With the same amount of internal memory, the other CSTs cannot be built for more than 32 distinct variants. In the scenarios where others can be built, we show that our CST performs competitively in practice.

2 PFP

To compute a PFP of $S[0..n-1]$, conceptually we choose a subset of all possible strings of some length w, with the chosen strings called *trigger strings*, and then divide S into overlapping phrases such that each starts with a trigger string (except possibly the first), ends with a trigger string (except possibly the last), and contains no other trigger string.

In practice we choose the trigger strings implicitly, by choosing a Karp-Rabin hash function and a parameter p and passing a sliding window of length w over S, putting a phrase break wherever the hash of the contents of the window is congruent to 0 modulo p (with the contents of the window there becoming the last w characters of the previous phrase and the first w character of the next one).

PFP is inspired by `rsync` [37] and `spamsum` (`https://www.samba.org/ftp/unpacked/junkcode/spamsum/README`; see also [19]), which have been in popular use for about twenty years. In some cases it works badly — e.g., if S is unary then either we split it into $n-w+1$ phrases or we do not split it at all — but we usually end up with a parse consisting of roughly n/p phrases of length roughly p.

It is plausible that PFP can be adapted to have good worst-case bounds, possibly by combining it either with string synchronizing sets [18] or locally consistent parsing [4]. As it is, the parsing uses only sequential access and small workspace, so it runs well even in external memory, and it can easily be parallelized. When S consists of genomes from individuals of the same species, then the genomes are parsed roughly the same way, so the total length of the strings in the dictionary of distinct phrases can be significantly less than the total length of the genome.

In this paper we assume we have already computed for S a PFP parse P with dictionary D, using Boucher et al.'s implementation [5], and we now restrict ourselves to using memory proportional to their combined size $|\mathrm{PFP}(S)|$. We say a phrase $S[i..j]$ *contains* a character $S[k]$ if $i \leq k \leq j-w$. Notice that, since consecutive phrases overlap by w characters, each character of S is contained in this sense in exactly one phrase, except the last w characters of S. To simplify the presentation, assume S is cyclic and starts with a trigger string — if need be, we can prepend one — so each character of S

is contained in exactly one phrase, with no exceptions.

For example, consider the string GATTACAT#GATACAT#GATTAGATA containing the trigger strings AC, AG and T# of length $w = 2$. We append $w = 2$ copies of # and consider the string as cyclic,

$$S = \texttt{GATTACAT\#GATACAT\#GATTAGATA\#\#GATTACAT\#GATACAT\#GATTAGATA\#\#}\ldots$$

of length $n = 28$, and treat ## as a trigger string as well. Therefore, the parse and dictionary are

$$\begin{aligned} P &= \texttt{\#\#GATTAC}, \texttt{ACAT\#}, \texttt{T\#GATAC}, \texttt{ACAT\#}, \\ &\quad \texttt{T\#GATTAG}, \texttt{AGATA\#\#} \\ &= D[0], D[1], D[3], D[1], D[4], D[2]\,, \end{aligned}$$

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

Notice that phrase $D[1] = \texttt{ACAT\#}$ occurs twice in P.

The most important property of a prefix-free parse is, as one would expect, that it is prefix-free. In particular, no proper phrase suffix of length at least w is a prefix of any other proper phrase suffix of length at least w. To see why, consider that each proper phrase suffix (i.e., a phrase suffix that is not a complete phrase) of length at least w ends with a trigger string and contains no other complete trigger string. Therefore, if a proper phrase suffix α of length at least w is a prefix of another such phrase suffix β, then $\alpha = \beta$.

LEMMA 2.1. ([5]) *The distinct proper phrase suffixes of length at least w are a prefix-free set of strings.*

A useful corollary of this is that each character $S[i]$ immediately precedes in S an occurrence of exactly one proper phrase suffix of length at least w, which is the suffix following $S[i]$ in the phrase containing it.

COROLLARY 2.1. ([5]) *We can partition S into subsequences such that the characters in the ith subsequence precede in S occurrences of the lexicographically ith proper phrase suffix of length at least w.*

Boucher et al. used this corollary as a starting point for building the BWT of S: for each proper phrase suffix α of length at least w that is preceded by only one distinct character c in D, they found the beginning of the interval for α in the BWT by summing up the frequencies in P of phrases ending with proper phrase suffixes of length at least w lexicographically less than α, then filled in the interval for α with as many copies of c as there are phrases in P ending with α.

To fill in the BWT intervals for a proper phrase suffix β of length at least w preceded by more than one distinct character in D, Boucher et al. used the following lemma, which is easily proven by induction. Essentially, they considered the phrases ending with β in the order they appear in the BWT of P (viewed as a sequence of lexicographically-sorted phrase identifiers), since the lemma means they are sorted by the suffixes that follow them in S.

LEMMA 2.2. ([5]) *Let $S[i..]$ and $S[j..]$ be suffixes of S starting at the beginning of occurrences of trigger strings, and let P_i and P_j be the parses of those suffixes with each phrase represented by its lexicographic rank in D. Then $S[i..]$ is lexicographically less than $S[j..n]$ if and only if P_i is lexicographically less than P_j.*

3 Our Compressed Suffix Tree

A suffix tree on $S[0..n-1]$ is a compact trie containing all the suffixes of S; each internal node v represents a distinct repeated string $s(v)$ in S and each leaf represents a suffix of S. The children of a node are sorted lexicographically by the next symbol, and thus the concatenation of all the labels on the leaves is equivalent to the suffix array $\mathrm{SA}[0..n-1]$, where $\mathrm{SA}[i]$ represents the suffix $S[\mathrm{SA}[i]..]$. This is simply a permutation of the suffixes of S in lexicographic order, and its inverse permutation is called inverse suffix array, $\mathrm{ISA}[0..n-1]$. The other relevant structure is the longest common prefix array, $\mathrm{LCP}[0..n-1]$, where $\mathrm{LCP}[0] = 0$ and $\mathrm{LCP}[i]$ is the length of the longest common prefix of $S[\mathrm{SA}[i-1]..]$ and $S[\mathrm{SA}[i]..]$. The BWT of S is the string $\mathrm{BWT}[0..n-1]$ with $\mathrm{BWT}[i] = S[\mathrm{SA}[i]-1 \bmod n]$.

Table 1 lists the key suffix tree operations that can be efficiently simulated with the CST of Fischer et al. [12]. Since the suffix tree has no unary nodes, Fischer et al. [12] identify a suffix tree node with the range of the suffix array covered by its descendant leaves. This makes operations like ROOT, ANC and COUNT trivial. They then show that all the other listed operations can be efficiently simulated with a data structure that supports the following primitives:

1. Access to individual cells $\mathrm{SA}[i]$, $\mathrm{ISA}[i]$, and $\mathrm{LCP}[i]$.

2. Operations range-minimum-query (RMQ), next-smaller-value (NSV), and previous-smaller-value (PSV) on LCP: $\mathrm{RMQ}(i,j)$ gives the position k of the minimum in $\mathrm{LCP}[i..j]$; $\mathrm{NSV}(i)$ and $\mathrm{PSV}(i)$ give the closest position following and preceding i, respectively, with value less than $\mathrm{LCP}[i]$.

Operation	Definition	Our simulation
ROOT()	The root of the suffix tree.	Return $[0, n-1]$.
LOCATE(v)	The suffix position i s.t. v is the leaf of suffix $S[i..]$.	Return $\text{SA}[v_l]$ ($v_l = v_r$ as v is a leaf).
ANC(v, w)	True iff v is an ancestor of w.	Return $v_l \le w_l \le w_r \le v_r$.
SDEPTH(v)	The length of $s(v)$.	Return $\text{Min}(v_l, v_r)$.
COUNT(v)	The number of leaves in the subtree rooted at v.	Return $v_r - v_l + 1$.
PARENT(v)	The parent node of v.	Compute $h = \max(\text{LCP}[v_l], \text{LCP}[v_r{+}1])$; return $[\text{Prev}(v_l{+}1, h), \text{Next}(v_r, h){-}1]$.
FCHILD(v)	The alphabetically first child of v.	Return $[v_l, \text{Next}(v_l, \text{SDEPTH}(v){+}1){-}1]$.
NSIBLING(v)	The alphabetically next sibling of v.	If $\text{LCP}[v_r+1] < \text{LCP}[v_l]$, v is the last child, else return $[v_r+1, \text{Next}(v_r+1, \text{LCP}[v_r+1]+1)-1]$.
SLINK(v)	The suffix link of v, i.e., the node w s.t. $s(v) = a \cdot s(w)$ for a symbol a.	Compute $x = \psi(v_l)$ and $y = \psi(v_r)$, $h = \text{Min}(x, y)$; return $[\text{Prev}(x+1, h), \text{Next}(y, h) - 1]$. Here $\psi(p) = \text{ISA}[\text{SA}[p] + 1 \bmod n]$.
SLINK^i(v)	The suffix link of v iterated i times.	Same as above, using $\psi^i(p) = \text{ISA}[\text{SA}[p] + i \bmod n]$ instead of $\psi(p)$.
LCA(v, w)	The lowest common ancestor node of v and w.	If one is ancestor of the other, return the ancestor. Else, let $v_l < w_l$. Compute $h = \text{Min}(v_l, w_r)$; return $[\text{Prev}(v_l{+}1, h), \text{Next}(v_r, h){-}1]$.
CHILD(v, a)	The node w s.t. the first letter on edge (v, w) is a.	Traverse the children w with FCHILD and NSIBLING; choose w s.t. LETTER(w, SDEPTH(v)+1) $= a$.
LETTER(v, i)	The letter $s(v)[i]$.	Compute $p = \text{SA}[v_l]{+}i{-}1$; return $S[p]$.
LAQ(v, d)	Level ancestor query, i.e., the highest ancestor w of v with SDEPTH(w) $\ge d$.	Return $[\text{Prev}(v_l{+}1, d), \text{Next}(v_r, d){-}1]$.

Table 1: The suffix tree operations we simulate with our CST, where v and w are tree nodes.

We will not use exactly the same primitives in our PFP-CST, but the following alternative ones:

1. Access to individual entries SA[i], ISA[i], and $S[i]$.
2. LCE(p, q), the length of the longest common prefix of $S[p..]$ and $S[q..]$. This is used to implement
   - $\text{LCP}[i] = \text{LCE}(\text{SA}[i], \text{SA}[i-1])$.
   - $\text{Min}(i, j) = \text{LCE}(\text{SA}[i], \text{SA}[j])$, the smallest value in $\text{LCP}[i+1..j]$.
3. Prev(i, h) and Next(i, h), the closest positions preceding and following i, respectively, with LCP value less than h.

The operations are then solved as shown on the right of Table 1 (we note that Abeliuk et al. [1] already solved some of the CST operations using Prev and Next, which they call PSV' and NSV'). Nodes v are represented by suffix array ranges $[v_l, v_r]$. The correctness of our version is immediate by comparison with the original solutions [12, 1]; typically they answer $[\text{PSV}(k), \text{NSV}(k)]$ with $k = \text{RMQ}(i, j)$, whereas we use $[\text{Prev}(i, h), \text{Next}(j, h)]$ with $h = \text{LCP}[\text{RMQ}(i, j)]$. In the sequel, we show how we compute our primitives.

4 Data Structures

Our PFP-CST stores the following components. We also describe how to build them efficiently within $O(|\text{PFP}(S)|)$ space.

P and D. We store the structures P and D, coming from the PFP, compactly but such that we can support fast random access to them. That is, each entry of P uses $\lceil \log_2 |D| \rceil$ bits and each entry of D uses $\lceil \log_2 \sigma \rceil$ bits, where $[0, \sigma - 1]$ is the alphabet of S.

Bitvector B_P. We store a cyclic bitvector $B_P[0..n-1]$ with a 1 marking the position of the first character in each trigger string in S. We can find the index of the phrase containing a character $S[i]$ with a rank query, modulo the number of phrases in P, and then find the offset of $S[i]$ in that phrase with a select query and a subtraction. Symmetrically, if we know the index of the phrase containing a character and its offset in that phrase, we can find the character's position in S. For our example $\ldots$##GATTACAT#GATACAT#GATTAGATA##$\ldots$ we store

$$B_P = 000010010000100100000010000010\,.$$

Notice that, because the bitvector is cyclic and it is convenient for the bits to align with the corresponding

characters, the 1 marking the first character of the trigger string at the beginning of the first phrase is the penultimate bit.

Since B_P has P 1s, it can be represented in $O(|P|)$ space [30], while efficiently supporting the queries $\text{rank}(B_P, i)$, which gives the number of 1s in $B_P[0..i]$ and $\text{select}(B_P, j)$, which gives the position of the jth 1 in B_P.

Bitvector B_{BWT}. We store a bitvector $B_{\text{BWT}}[0..n-1]$ that, for each distinct proper phrase suffix of length at least w, has a 1 marking the first position in SA that points to that phrase suffix. Recall that, by Corollary 2.1, every character of S precedes an occurrence of exactly one such phrase suffix. Figure 1 shows that for our example

$$B_{\text{BWT}} = 11111101101111101111110110111\,.$$

We do not need to build the BWT of S in order to build B_{BWT}. Instead, we append a unique terminator symbol to each phrase in D; build the suffix array and LCP array for D with those terminators; tag each suffix with the frequency in P of the phrase containing that suffix; and then scan the arrays, ignoring the suffixes that are whole phrases or shorter than w (ignoring the terminators) and aggregating the frequencies of the suffixes that differ only by their terminators. Figure 2 shows how we build B_{BWT} for our example.

Bitvector B_{BWT} has at most D 1s, so it can also be represented in $O(|D|)$ space [30].

Grid W. We store a two-dimensional discrete grid W over the BWT of the phrase identifiers in P, with the y-coordinates corresponding to the set of phrase identifiers in increasing co-lexicographic order. This implies that coordinates corresponding to identifiers of phrases ending with the same suffix α are consecutive. Figure 3 shows W for our example, both as a grid and illustrating its implementation as a wavelet tree [26].

We use W for orthogonal range queries, in particular counting the number of points that fall within a rectangle, or reporting one of those in some coordinate order [3]. For example, given j and r, we can say how many of the first j phrases in the BWT of P have co-lexicographic rank at least r, or, given a co-lexicographic range and a value j, we can return the index of the jth phrase in that interval to appear in the BWT of P.

Table and Grid M. We build a table M such that $M[r]$ tells us

1. the length ℓ of the lexicographically rth proper phrase suffix α of length at least w,
2. the lexicographic range of the reversed phrases starting with α reversed, starting with 0.

We compute the lengths while building B_{BWT}, and the lexicographic range by reversing the phrases and sorting them. Figure 4 shows M for our example, for which the reversed phrases are ##`ATAGA`, #`TACA`, `CATAG#T`, `CATTAG##` and `GATTAG#T`. The lexicographic range of `CA` in Figure 4 is $[2, 3]$ since the two reversed phrases starting with `CA` are in positions 2 and 3 in this sorted list (counting from 0).

We also store longest common prefix data of M in grid form: for each r, we store a point at row r and column c, where c is the longest common prefix of the phrase suffixes represented by $M[r]$ and $M[r-1]$. We use the same wavelet tree implementation as for the grid W. Those values c can be computed as longest common prefixes inside D, by brute force or building appropriate structures on D.

Suffix ranks on D We store a structure of size D that, for each position in D, which starts the proper suffix α, records the rank r of α among the distinct proper suffixes of length at least w. Abusing the notation a little, we call this structure ISA_D. This structure can be built from the actual inverse suffix array of D, replacing every value j by $B_{\text{BWT}}.\text{rank}(j)$.

Suffix tree data structures on P We regard P as a sequence of symbols, with their lexicographic order defined according to the dictionary strings they represent. We then store the suffix array SA_P, inverse suffix array ISA_P, and longest common prefix array LCP_P, of this sequence, with the only twist that longest common prefixes are measured in terms of original characters, not symbols of P, and that trigger strings are not counted (because they are duplicated across consecutive entries of P). In our example,

$$\begin{aligned} \text{SA}_P &= [0, 1, 3, 5, 2, 4] \\ \text{ISA}_P &= [0, 1, 4, 2, 5, 3] \\ \text{LCP}_P &= [0, 6, 0, 0, 3]. \end{aligned}$$

We also build the succinct RMQ data structure on top of LCP_P. All those structures can be computed within space $O(|\text{PFP}(S)|)$ using classical linear-space constructions [16, 17, 11], or slightly adapting them.

Finally, we build a geometric data structure storing points at row $\text{LCP}_P[i]$ and column i, for every position i in P. We use the same implementation as our other geometric structures. Note that this structure can be used as a replacement for the array LCP_P, since $\text{LCP}_P[i]$ is the row of the only point at column i in the grid.

5 Implementing the Primitives

5.1 Access to S The simplest query we consider is random access to S. To find $S[i]$ when given i, we use

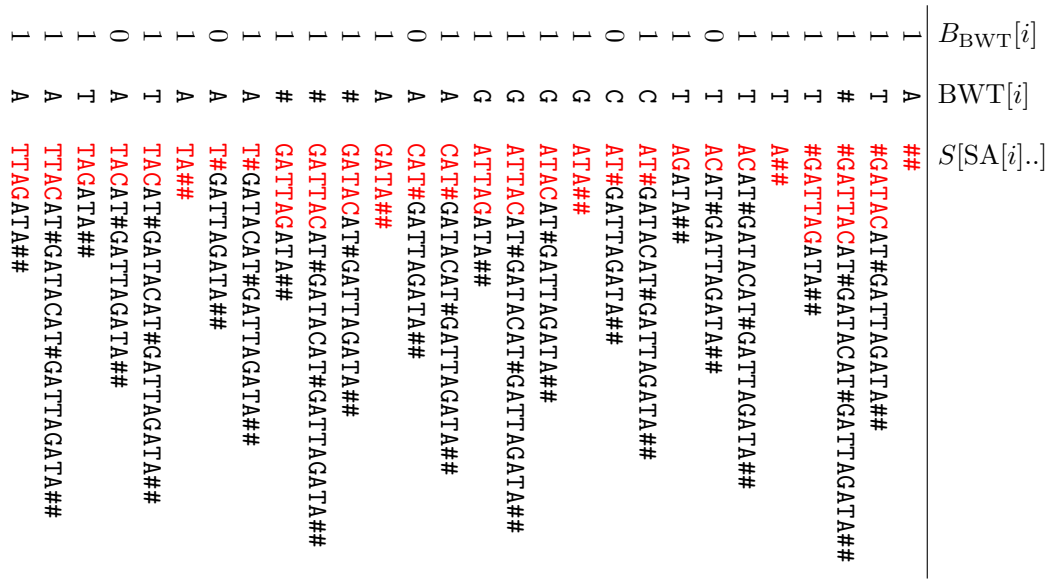

Figure 1: B_{BWT} for our example, with the BWT of S and the suffixes of S in lexicographic order. We have highlighted in red the unique proper phrase suffix of length at least w following each character, to clarify how B_{BWT} is defined. (We show $S[n-1] = \#$ and the empty suffix as #GATTACAT#GATACAT#GATTAGATA## and GATTACAT#GATACAT#GATTAGATA## instead, because we consider S to be cyclic and this should make clearer how the characters in the BWT are sorted.)

1	##2	1	0	AC3	1	-	C0	1	-	T#GATAC3	1
-	##GATTAC0	1	-	ACAT#1	2	-	C3	1	-	T#GATTAG4	1
-	#1	2	1	AG4	1	10	CAT#1	2	1	TA##2	1
-	#2	1	-	AGATA##2	1	-	G4	1	1	TAC0	1
1	#GATAC3	1	10	AT#1	2	1	GATA##2	1	0	TAC3	1
1	#GATTAC0	1	1	ATA##2	1	1	GATAC3	1	1	TAG4	1
1	#GATTAG4	1	1	ATAC3	1	1	GATTAC0	1	1	TTAC0	1
1	A##2	1	1	ATTAC0	1	1	GATTAG4	1	1	TTAG4	1
1	AC0	1	1	ATTAG4	1	10	T#1	2			

Figure 2: Suppose we append a unique terminator symbol to each phrase in D; sort the phrase suffixes **(center column)**; tag each suffix with the frequency in P of the phrase containing that suffix **(right column)**; mark with copies of - the suffixes which are whole phrases or shorter than w (ignoring the terminators), with 1 the first copy of each suffix (ignoring terminators) and with 0s the other copies **(left column)**; and then append to each 1 and 0 as many copies of 0 as the phrase frequency, minus 1 **(right column)**. Then the concatenation of the 0s and 1s is B_{BWT}, which is 11111101101111101111110110111 in this example.

B_P to find the index p of the phrase containing $S[i]$ and $S[i]$'s offset o in that phrase. More precisely, we compute $p = B_P.\mathrm{rank}(i)$ and $o = i - B_P.\mathrm{select}(p)$. We then use random access to $P[p]$ to identify that phrase, and random access to D to return the appropriate character.

We can return BWT$[i]$ by computing SA$[i]$ and then returning $S[\mathrm{SA}[i]-1]$, but we can do better: since, once we find the index of the phrase containing BWT$[i]$ in the BWT of P, we can extract BWT$[i]$ directly from D.

5.2 LCE queries A longest common extension (LCE) query $\mathrm{LCE}(i,j)$ returns the length of the longest common prefix of $S[i..]$ and $S[j..]$. In our example, $\mathrm{LCE}(3,11) = 9$ because the longest common prefix of TACAT#GATACAT#GATTAGATA## and TACAT#GATTAGATA## is TACAT#GAT.

Given i and j, we use the bitvector B_P as before to find the phrase indices p and q containing $S[i]$ and $S[j]$ and their offsets in those phrases. Let α and β be

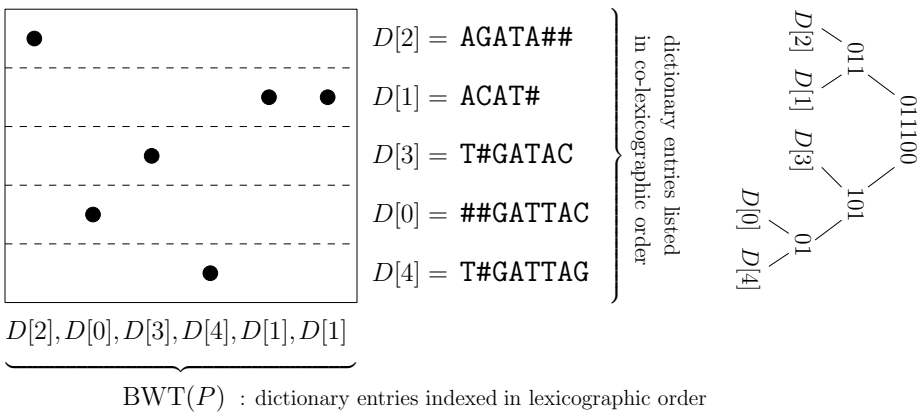

Figure 3: The grid W for our example **(left)** and its implementation as a wavelet tree **(right)**.

##	2	[0]	GA	2	[4]	#TAC	4	[1]	##AT	4	[0]
CATAG#	6	[2]	#TA	3	[1]	##ATAG	6	[0]	CAT	3	[2, 3]
CATTAG#	7	[3]	##ATA	5	[0]	CATAG	5	[2]	GAT	3	[4]
GATTAG#	7	[4]	CATA	4	[2]	CATTAG	6	[3]	CATT	4	[3]
##A	3	[0]	CATTA	5	[3]	GATTAG	6	[4]	GATT	4	[4]
CA	2	[2, 3]	GATTA	5	[4]	#T	2	[1]			

Figure 4: The reversed proper phrase suffixes of length at least w **(left column)**, their lengths **(center column)**, and the lexicographic range of the reversed suffixes starting with those reversed proper phrase suffixes **(right column)**.

the suffixes of those phrases starting at $S[i]$ and $S[j]$, so $|\alpha|, |\beta| > w$. In our example, the phrases containing $S[3]$ and $S[11]$ are $S[0..5] =$ GATTAC and $S[9..13] =$ GATAC.

By Lemma 2.1, neither α nor β is a proper prefix of the other, so there are only the following two possibilities: first, $\alpha[k] \neq \beta[k]$ for some $k < |\alpha|, |\beta|$, so $\mathrm{LCE}(i, j)$ is the length of the longest common prefix of α and β; second, $\alpha = \beta$, so $\mathrm{LCE}(i, j) = |\alpha| + \mathrm{LCE}(i + |\alpha|, j + |\alpha|)$, where $S[i + |\alpha|..]$ and $S[j + |\alpha|..]$ are both suffixes of S starting immediately after trigger strings. In our example, $\alpha = \beta =$ TAC, so $\mathrm{LCE}(3, 11) = 3 + \mathrm{LCE}(6, 14)$.

Since $\mathrm{LCE}(i, j) = \mathrm{LCP}(\mathrm{RMQ}(\mathrm{ISA}[i] + 1, \mathrm{ISA}[j]))$ (assuming $\mathrm{ISA}[i] < \mathrm{ISA}[j]$), there are several ways we can find the length of the longest common prefix of phrase suffixes quickly using $O(|\mathrm{PFP}(S)|)$ space. One is to take the dictionary D of distinct phrases as a text and store its corresponding arrays ISA_D, LCP_D, and RMQ structure on LCP_D. To reduce space in practice, we can map i and j to the suffix array of S using ISA queries, and use $B_{\mathrm{BWT}}.\mathrm{rank}(\mathrm{ISA}[i])$ and $B_{\mathrm{BWT}}.\mathrm{rank}(\mathrm{ISA}[j])$ to find the index of α and β among the distinct phrase suffixes in D, in lexicographic order. We can then build LCP and RMQ data structures on this set, which is of size at most D but usually smaller. We opt, in practice, for the simplest and least space-consuming alternative: we compare the phrase suffixes machine-word-wise on our plain representation of D, until finding a mismatch or until both phrases end.

To find the length of the longest common prefix of two suffixes of S starting immediately after the phrase indices p and q, we use the inverse suffix array ISA_P to find the positions $i_p = \mathrm{ISA}_P[p+1]$ and $i_q = \mathrm{ISA}[q+1]$ in the suffix array of P. Assume $i_p < i_q$, otherwise switch them. We then find $k = \mathrm{RMQ}(i_p+1, i_q)$, and the answer is $\mathrm{LCP}_P(k)$; recall that this array is twisted to return the longest common prefix measured in characters.

In our example query, having reduced computing $\mathrm{LCE}(3, 11)$ to computing $\mathrm{LCE}(6, 14)$, and knowing that 3 and 11 belong to the phrases $P[0]$ and $P[2]$, we map $i_p = \mathrm{ISA}_P[0 + 1] = 1$ and $i_q = \mathrm{ISA}[2 + 1] = 2$, so the range is $\mathrm{LCP}_P[1 + 1, 2]$ and the minimum is $\mathrm{LCP}_P[2] = 6$, the length of the longest common prefix of AT#GATACAT#GATTAGATA## and AT#GATTAGATA##. This finally yields $\mathrm{LCE}(3, 11) = 3 + 6 = 9$.

5.3 SA and ISA queries A suffix array (SA) query $\mathrm{SA}[i]$ returns the starting position in S (counting from 0) of its lexicographically ith suffix. In our example, $\mathrm{SA}[24] = 11$ because the suffix of S with lexicographic rank 24 (counting from 0) is $S[11..27] =$

TACAT#GATTAGATA##.

Given i, we use $r = B_{\text{BWT}}.\text{rank}(i) - 1$ and $j = i - B_{\text{BWT}}.\text{select}(B_{\text{BWT}}.\text{rank}(i))$ to find the lexicographic rank r (counting from 0) of the proper phrase suffix α of length at least w that starts at $\text{SA}[i]$, and the lexicographic rank j (counting from 0) of $S[\text{SA}[i]..]$ among the suffixes of S starting with α. In our example, $r = B_{\text{BWT}}.\text{rank}(24) - 1 = 19$ and $j = 24 - B_{\text{BWT}}.\text{select}(20) = 1$.

We access $M[r]$ to find the length ℓ of α and the lexicographic range $[c_1, c_2]$ of the reversed phrases starting with α reversed or, equivalently, the co-lexicographic range of the phrases ending with α. We then use W to find the index k of the jth phrase in that co-lexicographic interval to appear in the BWT of P, that is, the leftmost point in rows $[c_1, c_2]$. In our example, $M[19] = (3, [2, 3])$ and so $k = 2$.

Since α has length $\ell \geq w$, all of its occurrences in S are phrase suffixes. By Lemma 2.2, the lexicographic order of the suffixes of S starting with α, is the same as the lexicographic order of the parses starting at the trigger strings that are the last w characters of each of those occurrences of α.

Since the lexicographic order of those parses is what determines the order in which the phrases ending with α appear in the BWT of P, mapping the kth phrase of the BWT of P to its position in P tells us which phrase in P contains the starting point of the lexicographically jth suffix of S starting with α. Since we know the length of α from M, we can use B_P to find $\text{SA}[i]$.

Concretely, the phrase index containing α is $p = \text{SA}_P[k] - 1$ and the position of α is $\text{SA}[i] = B_P.\text{select}(p+1) - \ell + w$. In our example, since $\text{SA}_P[2] = 3$ and $M[19].\ell = 3$, we know $S[\text{SA}[24]]$ is the third-to-last character in $P[3-1]$. Since $w = 2$, the corresponding bit of B_P precedes the third 1 (which marks the start of the trigger string at the beginning of $P[3]$). Indeed, we compute $p = 2$ and $\text{SA}[24] = 12 - 3 + 2 = 11$.

Inverse suffix array (ISA) queries can be implemented as follows. Given a position i in S, we find the phrase $p = B_P.\text{rank}(i)$ it belongs and its offset $o = i - B_P.\text{select}(p)$ within the phrase. We then can map α inside D as for access queries. Given a position d of α in D, $r = \text{ISA}_D[d]$ yields its lexicographic rank r among the distinct phrases, so its range in ISA starts at $B_{\text{BWT}}.\text{select}(r)$.To determine the offset of our suffix among those starting with α, we find the column $k = \text{SA}_P[p+1]$ where the next phrase is at W, and with M we obtain the range $[c_1, c_2]$ of the rows corresponding to α reversed. We then count with W the number j of points within rows $[c_1, c_2]$ and column before k. The answer is then $j + B_{\text{BWT}}.\text{select}(r)$.

5.4 Prev and Next queries Let us focus on Prev queries; Next queries are analogous. A query $\text{Prev}(i, h)$ returns the largest $i' < i$ such that $\text{LCP}[i'] < h$. To solve this query, we first compute r and j as for the SA queries, where r identifies the distinct phrase suffix α starting at $\text{SA}[i-1]$. The situation is different depending on whether $|\alpha| = M[r].\ell \geq h$ or not.

In the positive case, we know that the answer is the first entry in a block of distinct phrase suffixes. We use the geometric data structure associated with M to find the largest row $r' \leq r$ with a point in column less than h. The answer is then the first entry of block r', $\text{Prev}(i, h) = B_{\text{BWT}}.\text{select}(r')$.

Otherwise, the answer could be in the same block of i. We then look for the largest $i - j < i' < i$ such that the longest common prefix between the phrases following α in $\text{SA}[i']$ and $\text{SA}[i'-1]$ is less than $h' = h - M[r].\ell + w$.

We note that the phrase-aligned suffixes that follow α in $\text{SA}[i-j+1], \ldots, \text{SA}[i-1]$ appear interspersed, though in the same order, in SA_P. We can then find the position p in SA_P of the suffix following α after $\text{SA}[i]$ by looking for the jth left-to-right point in the range $[c_1, c_2]$ of rows of W stored in $M[r]$. Let k be the column of this point; then we want the largest $k' < k$ such that the longest common prefix between $P[k..]$ and $P[k'..]$ (measured in characters) is less than h'. This is obtained with a range query on the geometric structure we associate with LCP_P. Since k' could correspond not to a suffix following α, we look at the first point in W within rows $[c_1, c_2]$ after or at column k'. If such point is k, we look backward, for the rightmost point in W within rows $[c_1, c_2]$ before column k'. If such a point exists, and its rank is j', then we have that $i' = i - j + j'$.

Note that this procedure may fail if we do not find a proper k' or j'. In such a case, the answer is not in $[i-j+1, i-1]$, so we revert to the first case where $M[r].\ell \geq h$.

6 Experiments

We implemented the data structures and measured their performance on real-world datasets. Experiments were performed on a server with Intel(R) Xeon(R) CPU E5-2640 v4 @ 2.40GHz with 40 cores and 756 gigabytes of RAM running Ubuntu 16.04 (64bit, kernel 4.4.0). The compiler was `g++` version 5.4.0 with `-O3 -DNDEBUG -funroll-loops -msse4.2` options. Runtimes were recorded with the C++11 `high_resolution_clock` facility and memory usage with the `malloc_count` tool (`https://github.com/bingmann/malloc_count`). The source code is available online at: *url removed for double blind purposes.*

Data We used real-world datasets from the Pizza&Chili repetitive corpus [31], *Salmonella* genomes

taken from the GenomeTrakr project [34], and human chromosome 19 genomes from the 1000 Genomes Project [35]; see Table 2. The Pizza&Chili repetitive corpus is a collection of repetitive texts characterized by different lengths and alphabet sizes. GenomeTrakr is an international project dedicated to isolating and sequencing foodbourne pathogens, including Salmonella. Hence, we used 6 collections of 50, 100, 500, 1000, 5000, and 10000 Salmonella genomes taken from GenomeTrakr. Lastly, we used 10 sets of variants of human chromosome 19 (`chr19`), containing 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, and 1000 distinct variants respectively. Each collection is a superset of the previous.

Data structures We compared the PFP data structures implementation (`pfp`); the compressed suffix tree implementation (`sdsl`) from the `sdsl-lite` library [14]; and the block tree compressed suffix tree implementation (`bt`) of Cáceres and Navarro [7]. The latter is shown to be the best CST for repetitive collections, whereas the former is a well-established CST implementation for regular sequence collections.

Implementation We implemented the PFP data structures using `sdsl-lite` library [14] bitvectors and their rank and select supports. We used wavelet matrices, which are a variant of wavelet trees better suited for point grids. We used SACA-K [29] to sort the parse lexicographically, and gSACA-K [21] to compute the SA, LCP array and document array of the dictionary. Using gSACA-K to sort the dictionary, we can use the same phrase terminator to concatenate each phrase. The result of gSACA-K is equivalent to the result obtained if we concatenate unique terminators in a lexicographically increasing order, as required for the computation of B_{BWT}.

Construction test setup We tested the running time and peak memory usage of the data structures during the construction. For building the PFP data structures, we first computed the prefix free parsing of the dataset using BigBWT [5] with 32 threads, a window size $w = 10$, and parameter $p = 100$. The resulting output is loaded in memory and used to build the PFP data structures. The running time for the construction of the PFP data structures includes the time to build the parse as well as the time to store the parse to disk.

We built each data structure 5 times for the Pizza&Chili corpus datasets, for the sets of chromosome 19 up to 64 distinct variants, and for Salmonella up to 1000 sequences. The remaining experiments have been tested only once. The experiments that exceeded 15 hours were omitted from further consideration, e.g. `chr19.1000` and `salmonella.10000` for `sdsl`. Furthermore, `bt` failed to successfully build for the sets of chromosome 19 greater than 16 distinct variants, and for Salmonella with more than 100 sequences due to integer overflows causing segmentation fault errors.

Querying test setup We implemented and tested all the queries reported on Table 1 on each data structure. Due to lack of space, we report the comparison of only five of them: Parent, NSibling, LCA, SLink, and Child. We also tested the data structures on a full task, that is, given two parameters k and t, count the number of substrings of the text of length at most k that occur at least t times. We used Google benchmarks (`https://github.com/google/benchmark`) for query testing.

For the suffix tree operations, we generated 1000 randomly distributed queries as in previous work [1, 7, 27]. For Parent and NSibling we randomly select a set of leaves and collect the nodes on their leaf-to-root paths; for LCA we randomly sample pairs of leaves; For SLink we randomly select a set of leaves and collect the nodes in their path to the root obtained by following the suffix links; finally, for the Child operation, we randomly sample a set of leaves, collect the nodes in their leaf-to-root path that have at least three children, and select the initial character of a randomly selected child. For the full task, we set k to 5 and t to 20.

Time Figures 5, 6, and 8 illustrate the construction time for all the data structures for Pizza&Chili, Chromosome 19, and Salmonella, respectively. From the reported data, we observe that the construction of `pfp` is always faster than `sdsl` and `bt`, except for the cases `chr19.1` and `chr19.2`, in which `sdsl` is the fastest to build. The maximum speedup of `pfp` with respect to `sdsl` is 21x (`einstein.en.txt`), 34.4x (`chr19.512`), and 9.2x (`salmonella.5000`). The speedup of `pfp` with respect to `bt` is 1329x (`einstein.en.txt`), 201x (`chr19.8`), and 203x (`salmonella.100`).

From Figure 6 we observe that doubling the length of the dataset, the running time of `pfp` increases by a factor of 1.9 when moving from 256 variants to 512, and a factor of 2 when moving from 512 variants to 1000. On the other hand, `sdsl` running time increases by a factor of 2.6 when moving from 256 variants to 512.

From Figure 8 we observe that increasing the length of the dataset by a factor of 10, when moving from 500 to 5000, increases the running time of `pfp` by a factor of 12, while for `sdsl` it increases by a factor of 24.

Space Figures 5, 7, and 9 illustrate the memory peak usage and the size of the data structure for all the data structures for Pizza&Chili, `chr19`, and `salmonella`, respectively. We observe that the peak memory usage of `pfp` is almost always less than both `sdsl` and `bt`. Yet, the size of `pfp` is larger than both `sdsl` and `bt`, except for `chr19.64`, `chr19.128`,

Name	Description	σ	$n/10^6$	n/r	Dict. (MB)	Parse (MB)
cere	Baking yeast genomes	5	461.29	157.19	90.34	16.99
einstein.de.txt	Wikipedia articles in German	117	92.21	5216.14	1.06	3.57
einstein.en.txt	Wikipedia articles in English	139	465.25	8961.42	3.16	17.82
Escherichia_Coli	Bacteria genomes	15	112.69	32.83	52.57	4.48
influenza	Virus genomes	15	154.81	251.30	49.10	6.27
kernel	Linux Kernel sources	160	249.51	499.82	14.78	9.94
para	Yeast genomes	5	429.27	111.78	84.87	16.34
world_leaders	CIA world leaders files	89	46.91	634.90	10.71	1.01
chr19.1000	Human chromosome 19	5	60 110.55	1287.38	274.63	2219.08
Salmonella.10000	Salmonella genomes database	4	51 820.38	36.61	4483.43	2039.16

Table 2: Datasets used in the experiments. We give the names and descriptions of the datasets in the first two columns. In column 3 we give the alphabet size. In columns 4 and 5 we report the length of the file and the ratio of the length to the number of runs in the BWT. Lastly, we give the size of the dictionary and the parse in columns 6 and 7, respectively.

chr19.256, and chr19.512, where pfp is the smallest one. We note that this is precisely the case of very large repetitive datasets (those in Pizza&Chili are not large, and salmonella is not that repetitive).

The difference between the memory peak usage and the data structure size is very small in pfp. Its maximum ratio is attained at chr.8 where the memory peak is 4.2x larger than the data structure size. For the Pizza&Chili dataset the maximum is 3.1x for kernel, while for the salmonella dataset the maximum is 3.7x for salmonella.100.

The change of trend in the memory usage of pfp from salmonella.1000 to salmonella.5000 occurs because we switched from gSACA-K 32 bit version to gSAKA-K 64 bit version, since the 32 bit version can sort text of length up to 2GB and the length of the dictionary is larger than 2GB.

Queries Figures 10, reports the time of each data structure to perform the operations Parent, NSibling, LCA, SLink, and Child, as well as the time to complete the full task. We observe that pfp is always slower than sdsl and bt on all queries. The main reason resides in the computation of LCP values, which has a central role in most of the suffix tree operations. We compute the LCP value using two SA queries, which perform a range select query on the wavelet tree W. This introduces a $\log(|P|)$ factor slowdown [3].

On the other hand, on the full task, the maximum speedup of sdsl with respect to pfp is 9.8x on chr19.512 and 16.1x salmonella.5000, with a maximum time gap of 4 seconds. Hence, considering also the construction time, the pfp is much faster than sdsl to perform the full task.

7 Conclusion

We have presented the first usage of PFP(S) as a data structure, augmenting it to support full suffix tree functionality (which involves LCE, SA, ISA, LCP, and BWT queries, among others) within $O(|\text{PFP}(S)|)$ space, which is small in practice when S is repetitive. We implemented this data structure and compared it to state-of-the-art compressed suffix trees on real-world datasets. Our experiments show that our PFP CST is almost always built more efficiently (both in time and space) than its competitors, allowing us to handle larger datasets. Although our PFP CST structure is somewhat larger than the compressed suffix trees and its query times are orders of magnitude slower, it is the only one whose construction can be scaled up within memory close to that of the final compressed suffix tree.

In particular, the PFP CST is faster than the alternatives (and can handle larger instances) on problems that, starting from the text collection, require the construction of the suffix tree and then some processing on it. Many tasks in bioinformatics, for example, become easily linear-time once we have suffix tree functionality [15]. Therefore, we expect the PFP CST to be useful when prototyping new solutions, even if eventually it can be replaced by more direct constructions.

References

[1] A. Abeliuk, R. Cánovas, and G. Navarro, *Practical compressed suffix trees*, Algorithms, 6 (2013), pp. 319–351.

[2] A. Apostolico, *The myriad virtues of subword trees*, in Combinatorial Algorithms on Words, NATO ISI Series, Springer-Verlag, 1985, pp. 85–96.

[3] J. Barbay, F. Claude, and G. Navarro, *Compact binary relation representations with rich functionality*, Information and Computation, 232 (2013), pp. 19–37.

[4] O. Birenzwige, S. Golan, and E. Porat, *Locally consistent parsing for text indexing in small space*, in Proc. ACM-SIAM Symposium on Discrete Algorithms (SODA), 2020, pp. 607–626.

[5] C. Boucher, T. Gagie, A. Kuhnle, B. Langmead, G. Manzini, and T. Mun, *Prefix-free parsing for*

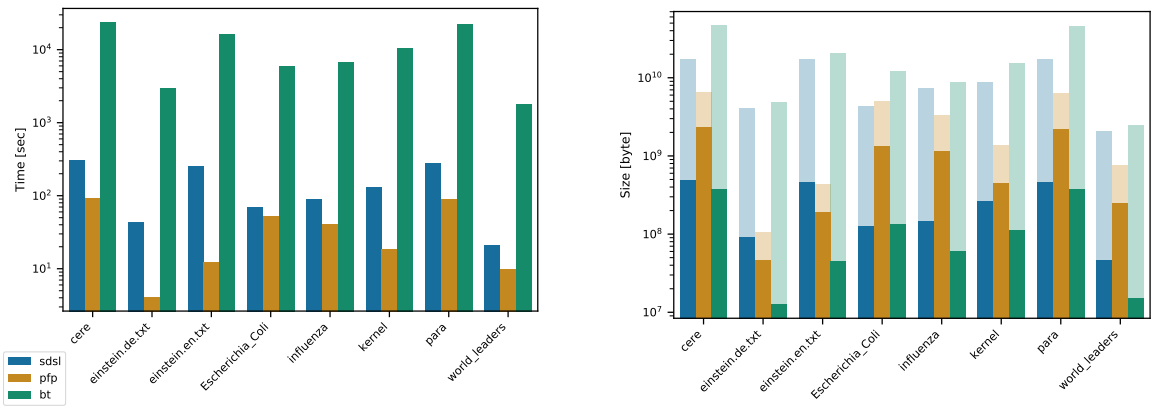

Figure 5: Pizza&Chili dataset construction running time (**left**) and peak memory usage (**right; light bars**) and data structure size (**right; dark bars**).

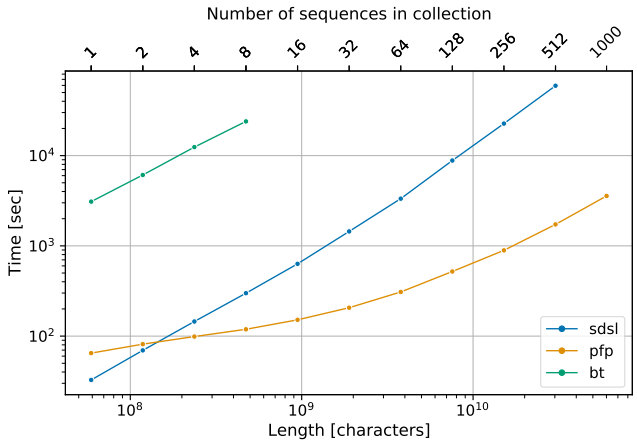

Figure 6: Construction time for Chromosome 19.

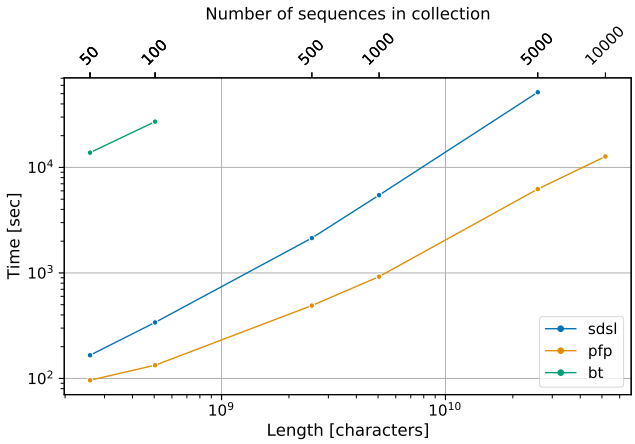

Figure 8: Construction time for Salmonella.

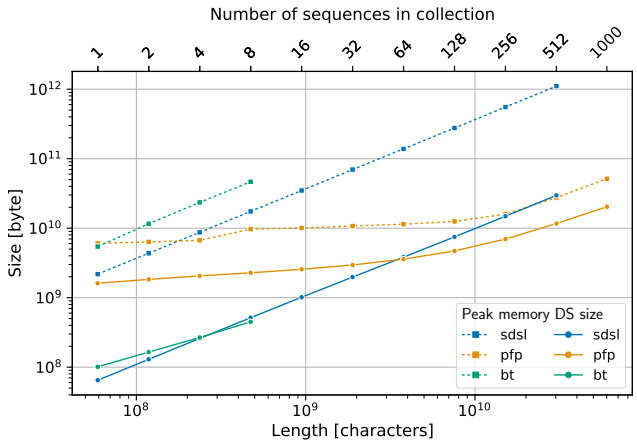

Figure 7: Peak memory and size for Chromosome 19.

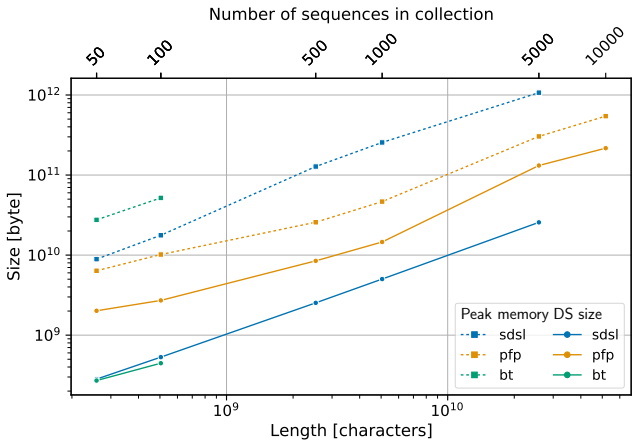

Figure 9: Peak memory and size for Salmonella.

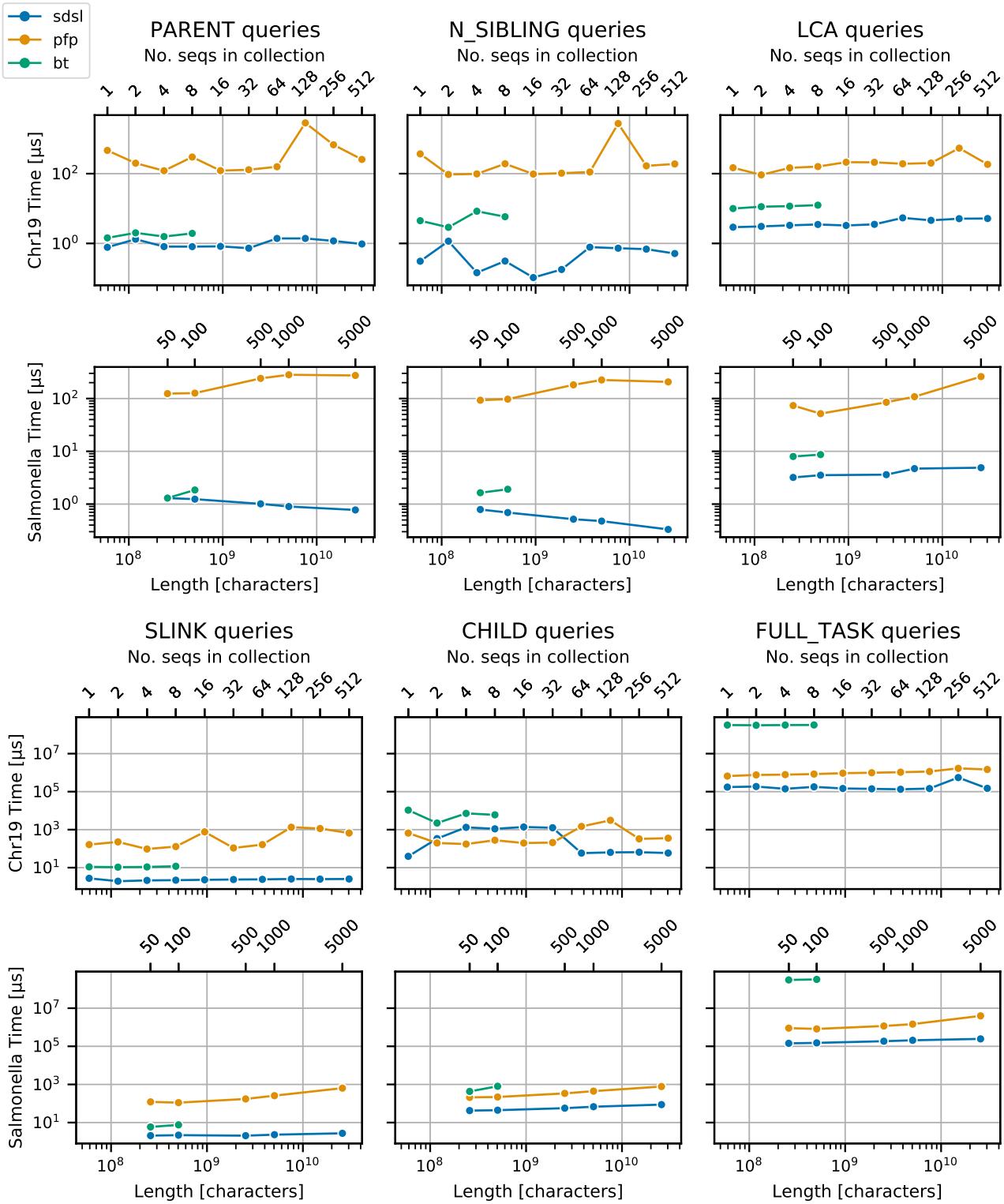

Figure 10: Running time for Parent, NSibling, LCA, SLink, Child, and Full-task queries for the chromosome 19 datasets and the salmonella datasets.

building big BWTs, Algorithms for Molecular Biology, 14 (2019), pp. 13:1–13:15.

[6] M. Burrows and D. Wheeler, *A block sorting lossless data compression algorithm*, Tech. Rep. 124, Digital Equipment Corporation, 1994.

[7] M. Cáceres and G. Navarro, *Faster repetition-aware compressed suffix trees based on block trees*, in Proc. 26th International Symposium on String Processing and Information Retrieval (SPIRE), 2019, pp. 434–451.

[8] M. Crochemore and W. Rytter, *Jewels of Stringology*, World Scientific, 2002.

[9] P. Ferragina, T. Gagie, and G. Manzini, *Lightweight data indexing and compression in external memory*, Algorithmica, 63 (2012), pp. 707–730.

[10] J. Fischer, *Wee LCP*, Information Processing Letters, 110 (2010), pp. 317–320.

[11] J. Fischer and V. Heun, *Space-efficient preprocessing schemes for range minimum queries on static arrays*, SIAM Journal on Computing, 40 (2011), pp. 465–492.

[12] J. Fischer, V. Mäkinen, and G. Navarro, *Faster entropy-bounded compressed suffix trees*, Theoretical Computer Science, 410 (2009), pp. 5354–5364.

[13] T. Gagie, G. Navarro, and N. Prezza, *Fully functional suffix trees and optimal text searching in bwt-runs bounded space*, Journal of the ACM, 67 (2020), pp. 1–54.

[14] S. Gog, T. Beller, A. Moffat, and M. Petri, *From theory to practice: Plug and play with succinct data structures*, in 13th International Symposium on Experimental Algorithms, (SEA), 2014, pp. 326–337.

[15] D. Gusfield, *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*, Cambridge University Press, 1997.

[16] J. Kärkkäinen, P. Sanders, and S. Burkhardt, *Linear work suffix array construction*, Journal of the ACM, 53 (2006), pp. 918–936.

[17] T. Kasai, G. Lee, H. Arimura, S. Arikawa, and K. Park, *Linear-time longest-common-prefix computation in suffix arrays and its applications*, in Proc. 12th Annual Symposium on Combinatorial Pattern Matching (CPM), 2001, pp. 181–192.

[18] D. Kempa and T. Kociumaka, *String synchronizing sets: sublinear-time BWT construction and optimal LCE data structure*, in Proc. 51st Annual ACM SIGACT Symposium on Theory of Computing (STOC), 2019, pp. 756–767.

[19] J. D. Kornblum, *Identifying almost identical files using context triggered piecewise hashing*, Digital Investigation, 3 (2006), pp. 91–97.

[20] A. Kuhnle, T. Mun, C. Boucher, T. Gagie, B. Langmead, and G. Manzini, *Efficient construction of a complete index for pan-genomics read alignment*, Journal of Computational Biology, (2020).

[21] F. A. Louza, S. Gog, and G. P. Telles, *Inducing enhanced suffix arrays for string collections*, Theoretical Computer Science, 678 (2017), pp. 22–39.

[22] V. Mäkinen, D. Belazzougui, F. Cunial, and A. I. Tomescu, *Genome-Scale Algorithm Design: Biological Sequence Analysis in the Era of High-Throughput Sequencing*, Cambridge University Press, 2015.

[23] V. Mäkinen, G. Navarro, J. Sirén, and N. Välimäki, *Storage and retrieval of highly repetitive sequence collections*, Journal of Computational Biology, 17 (2010), pp. 281–308.

[24] U. Manber and G. Myers, *Suffix arrays: a new method for on-line string searches*, SIAM Journal on Computing, 22 (1993), pp. 935–948.

[25] E. M. McCreight, *Priority search trees*, SIAM Journal on Computing, 14 (1985), pp. 257–276.

[26] G. Navarro, *Wavelet trees for all*, Journal of Discrete Algorithms, 25 (2014), pp. 2–20.

[27] G. Navarro and A. Ordóñez, *Faster compressed suffix trees for repetitive text collections*, Journal of Experimental Algorithmics, 21 (2016), p. article 1.8.

[28] G. Navarro and L. M. S. Russo, *Fast fully-compressed suffix trees*, in Proc. 24th Data Compression Conference (DCC), 2014, pp. 283–291.

[29] G. Nong, *Practical linear-time O(1)-workspace suffix sorting for constant alphabets*, ACM Transactions on Information Systems, 31 (2013), p. 15.

[30] D. Okanohara and K. Sadakane, *Practical entropy-compressed rank/select dictionary*, in Proc. 9th Workshop on Algorithm Engineering and Experiments (ALENEX), 2007, pp. 60–70.

[31] Pizza & Chili repetitive corpus. Available at `http://pizzachili.dcc.uchile.cl/repcorpus.html`. Accessed 16 April 2020.

[32] L. M. S. Russo, G. Navarro, and A. Oliveira, *Fully-compressed suffix trees*, ACM Transactions on Algorithms, 7 (2011), p. article 53.

[33] K. Sadakane, *Compressed suffix trees with full functionality*, Theory of Computing Systems, 41 (2007), pp. 589–607.

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

[35] The 1000 Genomes Project Consortium, *A global reference for human genetic variation*, Nature, 526 (2015), pp. 68–74.

[36] The Computational Pan-Genomics Consortium, *Computational pan-genomics: status, promises and challenges*, Briefings in Bioinformatics, 19 (2018), pp. 118–135.

[37] A. Tridgell, *Efficient Algorithms for Sorting and Synchronization*, PhD thesis, The Australian National University, 1999.

[38] E. Ukkonen, *On-line construction of suffix trees*, Algorithmica, 14 (1995), pp. 249–260.

[39] P. Weiner, *Linear pattern matching algorithms*, in Proc. 14th IEEE Annual Symposium on Switching and Automata Theory, 1973, pp. 1–11.