PAPER

Measuring Genomic Data with PFP

Zsuzsanna Lipták,[1,*] Simone Lucà[2] and Francesco Masillo[3]

[1,2]Department of Computer Science, University of Verona, Strada le Grazie, 15, 37134, Verona, Italy and [3]Faculty of Computer Science, Technische Universität Dortmund, Otto-Hahn-Straße, 14, 44227, Dortmund, Germany
[*]Corresponding author. zsuzsanna.liptak@univr.it

Abstract

Summary: Prefix free parsing [Boucher et al., Alg. Mol. Biol., 2019], is a highly effective heuristic for computing text indexes for very large amounts of biological data. The algorithm constructs a data structure, the prefix-free parse (PFP) of the input, consisting of a dictionary and a parse, which is then used to speed up computation of the final index. In this paper, we study the *size* of the PFP, which we refer to as π, and show that it is a powerful theoretical tool in its own right. To show this, we present two use cases. We first study the application of π as a *repetitiveness measure* of the input text, and compare it to other currently used repetitiveness measures, including z, r, and δ. We then turn to the use of π as a measure for *pangenome openness*. In both applications, our results are similar to existing measures, but our tool, in almost all cases, is more efficient than those computing the other measures, both in terms of time and space, sometimes by an order of magnitude. We close the paper with the first systematic study of the parameter choice for PFP (window size w and modulus p). This gives rise to interesting open questions.
Availability and implementation: The source code is available at `https://github.com/simolucaa/piPFP`, the accession codes for all the datasets used at `https://github.com/simolucaa/piPFP_experiments`.

Key words: compression, repetitiveness measures, prefix free parsing, pangenome openness, text indexing, massive data

1. Introduction

With the availability of unprecendented amounts of genomic data today, it is becoming more and more important to be able to store, handle, and process this data efficiently [15]. One central issue is that datasets, in addition to their size, can have different properties, such as number of sequences, length of individual sequences, or level of repetitiveness. Current software is frequently tailor-made for specific types of data, taking advantage of certain properties of the input. Therefore it is paramount that users be able to test their data's properties.

One such central property is *repetitiveness*. Intuitively, this is a measure of how much of the sequence data occurs more than once in the input. This concept is intimately related to our ability to store the data using compressed text indexes, which require little storage space while at the same time enabling efficient string processing tasks such as pattern matching [34]. To date there is no generally accepted measure of repetitiveness [35], and different measures are being used, such as the number z of the phrases of Lempel-Ziv 77 parse [48], the number r of runs of the Burrows-Wheeler Transform (BWT) [11], or the more recent δ-measure [42], also called string complexity.

Another property of interest is *pangenome openness* [45]. This concept essentially measures the amount of additional data necessary to have complete information about a species, given partial information about the pangenome in the form of a dataset of individual genomes. Several methods for measuring pangenome openness were introduced recently [40, 5, 39, 23, 14].

Prefix free parsing [10], is a highly effective heuristic for computing text indexes for very large amounts of biological data. The algorithm computes a data structure, the prefix-free parse (PFP) of the input, consisting of a dictionary and a parse, which is then used to speed up computation of the final index. Originally introduced as a preprocessing step to speed up the computation of the Burrows-Wheeler Transform of very large genomic sequence data (with the implementation named `BigBWT` for this reason), it has since been extended to computing other data structures such as the suffix array [28], the extended BWT [6, 7], an extension of the BWT to labelled trees, called XBWT [17], or as a text index [8, 36]. For BWT construction, a recursive variant was also proposed [37] in order to be able to further scale the computation with huge amounts of data. It has shown competitive results also for building grammars [18, 26], computing the Lempel-Ziv factorization [22], and computing the smallest suffixient set [13]. Other applications of PFP for constructing pangenome-sized data structures include building the r-index [19] augmented with thresholds [3]. This combination of data structures enable the computation of matching statistics, which in turn allows finding Maximal Exact Matches (MEMs). MEMs are a crucial ingredient for speeding up the alignment phase of sets of reads to a reference genome. Recent tools, such as `MONI` [43], `PHONI` [9], `AUG-PHONI` [33], `LAZY-PHONI` [20], `SPUMONI` [1], `SPUMONI2` [2], `SIGMONI` [44], `MOVI` [46], all use PFP as preprocessing step to then build the underlying r-index.

In this paper, we explore the power of PFP for measuring certain properties of the input data. We introduce the measure π, the *size* of the PFP data structure, and show that it is a powerful theoretical tool in its own right. We show that it can be used as a repetitiveness measure, as well as for measuring pangenome openness. Our experiments in fact indicate that π could be a good candidate for a new measure in both of these applications. It gives similar results regarding repetetiveness as the established repetitiveness measures z, r, and δ. When used

to decide pangenome openness, classification using π gives the same result as recent comparable measures on most input data.

We also present a tool, `piPFP`, which computes π. Our experiments show that `piPFP` is highly efficient both as regards computation time and memory requirements, and outperforms other methods sometimes by orders of magnitude. We believe that using π as an alternative method for measuring these properties can be beneficial to users, as it requires little computational resources and has similar accuracy as other, more complex and/or slower, methods.

Finally, we turn our attention to parameter choice in PFP. The PFP algorithm has two user-defined parameters, the window size w and the modulus p. In the original paper, a brief study of parameter choice was presented, on the basis of which the default parameters of the accompanying tool were chosen. Here, we explore these parameters in depth, concluding that for certain applications, very small window sizes may not always be appropriate. Our study on the parameter choices leads to several open problems of a theoretical nature.

The rest of the paper is organized as follows. In Section 2, we give the necessary technical background and present our experimental setup. Section 3 contains the experimental results: as a repetitiveness measure, for pangenome openness, and exploring parameter choice for PFP. We close with a discussion and open problems in Section 4.

2. Materials and Methods

2.1. Prefix Free Parsing

Prefix-free parsing (PFP), introduced in [10], is a technique that processes a string T of length n using two integers, w and p, both greater than 1. This method produces a parse of T consisting of overlapping phrases, each uniquely stored in a dictionary denoted as D. The resulting parse P is represented as a meta-string, where each phrase is encoded to as its rank in the lexicographically sorted dictionary. We refer to the prefix-free parse of T as $PFP(T)$, consisting of the dictionary D and the parse P. Below, we provide an outline of the approach.

As implied by its name, PFP ensures that no suffix of any phrase in D with a length of at least w is a prefix of another phrase. This property is formalized in the lemma below:

Lemma 1 ([10], Lemma 1) For a string T and its prefix-free parse $PFP(T)$, which consists of a dictionary D and a parse P, the set S of suffixes of phrases in D of length at least w forms a prefix-free set.

The PFP process begins by appending w occurrences of a special symbol # to both the beginning and end of T. This symbol #, which is lexicographically smaller than any character in the alphabet of the string and does not appear in T, results in an augmented string denoted as $T' = \#^w T \#^w$.

Next, we define a set of trigger strings, E, that will guide the parsing of T. Using a given parameter p, we compute the Karp-Rabin hash $H_p(t)$ for all substrings of length w in T'. A substring $t = T'[i..i+w-1]$ is in E if $H_p(t) = 0$ or if $t = \#^w$. This can be computed efficiently via the well-known rolling hash algorithm of Karp and Rabin [25]. These trigger strings are then used to parse T'.

The dictionary D for PFP is constructed as follows: each element $d \in D$ is a substring of T' that starts and ends with a trigger string from E and does not contain any other trigger string within it. These substrings, referred to as phrases, are identified by scanning T and adding a new phrase to D whenever the next trigger string is encountered.

Once the dictionary D is built, it is sorted lexicographically. Using the sorted dictionary D and the input string T, we generate the parse P, which consists of overlapping phrases from D. Each phrase overlaps with the next by w characters, corresponding to a trigger string. This parsing process creates an array of integers in P, where each integer is the rank of the corresponding phrase in D. With D and P, the original string T can be reconstructed.

Throughout this paper we will refer to the combined size of the dictionary and the parse for a combination of parameters w and p as $\pi_{w,p} = ||D|| + |P|$, where $||D||$ denotes the total length of the phrases in D, and $|P|$ the length of the parse. It was shown in [10] how to construct the Burrows-Wheeler Transform (BWT) [11] using space and time proportional to $\pi_{w,p}$. Later, in [28], it was shown how to compute the suffix array samples at run-boundaries while computing the BWT to produce the r-index [19] within the same time and space bounds.

2.2. Repetitiveness measures

The study of string repetitiveness focuses on quantifying the complexity and redundancy present in a string. For relationships among the wide spectrum of repetitiveness measures, we point the reader to the comprehensive survey [35].

2.2.1. Number of phrases in the Lempel-Ziv factorization

The Lempel-Ziv compression algorithms are foundational approaches to understanding repetitiveness in strings [29, 47]. The core idea behind the LZ compression algorithm is parsing a string T into a sequence of phrases, where each phrase is either a new character or a substring of maximum length that appears also earlier in the text. The number of phrases, denoted as z, serves as a measure of repetitiveness, as highly repetitive strings tend to have fewer phrases.

The Lempel-Ziv parsing is significant not only as a compression technique but also as a way to define theoretical bounds for other measures of repetitiveness. It captures the idea of redundancy in a string by representing repeated patterns succinctly.

2.2.2. Number of runs of the Burrows-Wheeler Transform

The Burrows-Wheeler Transform (BWT) [11] is another tool widely used in string processing and compression. Given a string T, the BWT is computed by sorting lexicographically every suffix of T and then concatenating the preceding character of each sorted suffix. A crucial property of the BWT is that repetitive strings tend to produce long runs of identical characters in the transformed string. The number of runs, denoted as r, is an effective measure of repetitiveness. Smaller values of r indicate higher levels of repetitiveness in the original string.

The number of runs r is closely related to the compressibility of T and can be exploited for building text indexes using methods like the FM-index [16], or the more recent r-index [19].

2.2.3. The measure δ

The measure δ quantifies the repetitiveness of a string by estimating the number of distinct substrings that can be extracted using a sliding window of fixed length k [42] and applying a normalization factor. Formally, for a string T of

length n, the measure δ is defined as:

$$\delta(T) = \max_{1 \leq k \leq n} \frac{S(k)}{k},$$

where $S(k)$ is the number of distinct substrings of length k of T.

The measure δ is known to lower-bound both z and r for every text T [27].

2.3. Pangenome openness

The term pangenome openness [45] refers to a method to estimate a pangenome's total size, given a set of genomes of individuals from the same species. This estimation requires calculating the pangenome growth (f_{tot}), which monitors the number of distinct attributes observed as a function of the number of entities considered. Heap's Law [21], known in the information retrieval field, can be used to describe this phenomenon as follows: the rate at which new attributes are found decreases as one considers more and more entities, as this rate is proportional to $kN^{\gamma-1} = kN^{-\alpha}$, where $\alpha = 1 - \gamma$ and k are constants that depend on the particular text, and N is the total number of attributes. That is, as sampling proceeds, discovering a new attribute becomes increasingly harder [45, 21, 40].

In pangenomics, the entities are the genomes, and the attributes can be genes, ORFs, or genome intervals of fixed size (e.g., k-mers [40]). The parameter α is used to classify a genome as either closed or open:

- If $\alpha < 1$ the pangenome is classified as *open*. This means that the number of attributes discovered increases significantly with the addition of new genomes. This indicates a high level of genomic diversity, suggesting that each newly sequenced genome contributes unique genes.
- If $\alpha > 1$ the pangenome is classified as *closed*. This indicates that the number of new attributes plateaus as more genomes are added to the set.

This definition of pangenome growth, however, makes α dependent on the order in which the genomes are added. To address this issue, one could compute the average f_{tot} over all possible permutations, but this is prohibitive even for small n. A solution is to estimate the growth by selecting a subset of all possible permutations [45, 5]. Here, we employ an alternative method, as described by Parmigiani et al. [40], but replacing k-mers by the phrases from the PFP algorithm (see Sec. 2.5 for details).

When using PFP for this purpose, we chose small values for the parameters w and p. We did this to better capture the differences between genomes as, on average, we would get n/p phrases. Larger values for the parameters would not segment the genome enough to be able to calculate f_{tot}. This is conceptually the same as using the k-mer based approach.

For example, choosing $w = 4$ and $p = 10$ (see Sec. 3.2), on expectation, every 10th window is a trigger string. Therefore, due to the fact the computation is performed using a rolling window, the average length of the phrases will be $w+p-1 = 13$, which is basically subsampling the k-mer space with $k = 13$.

2.4. Datasets used

We used different datasets to study PFP's parameters (Table 1) and to compute pangenome openness (Table 2). They can be accessed on NCBI via accession codes provided in the repository `https://github.com/simolucaa/piPFP_experiments`.

We selected the datasets for the study of PFP parameters to represent five different biological categories: a complex animal (*Homo sapiens*), a plant (*Arabidopsis thaliana*), a simple eukaryote (*Saccharomyces cerevisiae*), two prokaryotic organisms (*Salmonella enterica* and *Escherichia coli*), and a virus (SARS-CoV2). Each genome was downloaded from NCBI. The only exceptions are the *S. enterica* genomes, which were downloaded via accession codes provided in [5].

The datasets for pangenome openness consist mainly of prokaryotes, given their prevalence in this field. We downloaded most of the data via accession codes provided by [40]. The only exceptions are the large datasets used to evaluate the scalability of the tools (9 *H. sapiens* and 7,785 *E. coli* genomes), which were downloaded from NCBI.

We used the following filters for all the data downloaded from NCBI: "Assembly level: complete", "Exclude: atypical genomes, metagenome-assembled genomes (MAGs), genomes from large multi-isolate projects".

Table 1. Summary of the data used for the study of PFP parameters.

Species	#genomes	Average genome size (Mb)	Total size (Mb)	σ
H. sapiens	9	3129.03	27736.43	16
A. thaliana	21	136.64	2869.43	16
S. cerevisiae	28	12.04	337.23	9
S. enterica	60	4.80	288.25	14
E. coli	120	5.74	689.31	8
SARS-CoV2	500	0.03	14.90	11

Table 2. Summary of the data used for the study of pangenome openness.

Species	#genomes	Average genome size (Mb)	Total size (Mb)	σ
B. aphidicola	35	0.61	21.18	5
B. cereus	83	5.65	468.76	13
C. botulinum	50	3.95	197.55	5
C. burnetii	13	2.06	26.78	4
C. jejuni	234	1.68	392.21	15
F. tularensis	57	1.90	108.37	5
H. pylori	234	1.63	381.75	15
P. marinus	9	1.73	15.58	4
R. palustris	8	5.38	43.01	4
S. pneumoniae	88	2.11	185.81	8
S. pyogenes	247	1.85	455.88	11
Y. pestis	56	4.71	263.87	6

2.5. Implementation details

Our implementation is based on the original implementation of PFP [10], see `https://gitlab.com/manzai/Big-BWT`.

We modified the original implementation in order to speed up the computation and to save working space. In the dictionary, we only store the hash of the phrases instead of the full explicit phrase since we are not interested in the actual characters of the phrase but rather on the number of distinct phrases and their length. In practice, we maintain an hashmap that stores the hashes of the phrases as keys and their lengths

as items. We also avoid storing explicitly the parse on disk by counting the number of total phrases that are found during computation, effectively reducing the parse to a single integer counter. This is done in order to avoid costly I/O operations.

When using multiple threads, we also slightly deviate from the original implementation by using a concurrent hashmap from the oneAPI Threading Building Blocks (oneTBB) library [38] instead that of the standard implementation found in the C++ library. This allows for faster parallel insertions of new entries in the hashmap without an explicit lock mechanism.

For the computation of pangenome openness, we followed the approach of Parmigiani et al. [40], which, rather than estimating the average f_{tot} across a number of input permutations like done in [45, 5], computes a histogram $h(i)$ counting how many "items" are present in exactly i genomes. However, we replace the k-mers used in [40] with PFP phrases. To enable fast computation across different threads, we assign to each thread a range of sequences to process and populate a thread-specific hashset. This thread-specific hashset stores only the hashes of the phrases. After having populated the hashset for one genome, there is a merging phase in which we merge together the populated thread-specific hashset into a collective hashmap. This collective hashmap stores hashes of phrases as keys and the corresponding counter as items. The collective hashmap is then processed to output the final histogram after every genome is processed. In our program, for the hashsets we use the standard C++ library implementation, while for the collective hashmap we again use the oneTBB implementation.

Our tool `piPFP` is available at `https://github.com/simolucaa/piPFP`.

3. Experimental Results

For all experiments presented in this section, we used a machine equipped with an Intel(R) Core(R) i9-11900 @ 2.50GHz (with turbo speed @ 5GHz) with 16 MB of cache and 8 cores (16 threads via hyperthreading) and 64 GB of DDR4-3200MHz RAM. The operating system was Ubuntu 22.04 LTS and the compiler used was `g++` version 11.4.0. Our implementation was compiled with options `-std=c++20 -O3 -march=native -ldl -pthread -ltbb` enabled. Every measurement regarding running times and space consumptions were recorded using the built-in command of the operating system: `/usr/bin/time -v`.

3.1. PFP as a repetitiveness measure

In this subsection, we will compare $\pi_{w,p}$ to other well-known repetitiveness measures.

The data used to carry out this experiment is the same set of *S. enterica* genomes found in [5]. To compute the Lempel-Ziv factorization, we used the tool `PFP_LZ77`[1] [22]. We ran the PFP preprocessing step with the default parameters `-w 10 -p 100`. For computing r, we chose the tool `optimalBWT`[2] [12], which computes the optimal value for r when considering a set of strings. We ran the tool with parameter `-a sais`. Lastly, to compute δ, we used the tool `substring-complexity`[3] [27]. We used the executable `delta` in order to obtain the exact value of δ.

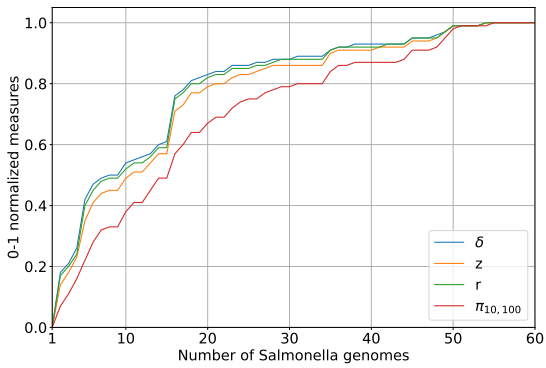

Fig. 1. 0-1 normalized measures (δ, z, r and $\pi_{10,100}$) show a similar pattern of growth with the progressive addition of 60 genomes.

Following the procedure of [5], we chose an arbitrary order of the input set of genomes. Then, we computed the repetitiveness measures z, r, δ, and $\pi_{10,100}$ on increasingly larger subsets of genomes, from 1 to 60. The resulting values were then 0-1 normalized. The results are reported in Figure 1. As we can observe, $\pi_{10,100}$ closely follows the trend of the other repetitiveness measures, showing large increases that are consistent with the compared measures. Further, we note that the relative increase in $\pi_{10,100}$ is more prominent than in the other measures. This suggests a finer sensitivity of π when adding new information to the collection of data, making it a good candidate for detecting differences between genomes.

We also assessed the practicality of `piPFP` comparing running time and space consumption against the other tools. We also chose another tool for computing z, `kkp3`[4] [24], which uses a different data structure to compute the LZ factorization, namely the suffix array [32]. For computing r, we also tried `ropeBWT2`[5] [30] and `ropeBWT3`[6] [31]. We are omitting their results due to being slightly worse both for running time and space consumption compared to `optimalBWT` at this size of data. Morever, for δ, we also ran the streaming version of `substring-complexity` [4], using the executable `delta-stream`, and `DandD`[7] [5] in dashing mode, using the command `dandd tree` with parameters `-k 15 -n 10`. We ran every tool using a single thread on the full set of 60 *S. enterica* genomes and on another dataset consisting of 60 *E. coli* genomes. As we can see in Figure 2, our tool largely outperforms the competitors time-wise, by at least an order of magnitude. It is worth noting that our tool also allows for high parallelization: using 16 threads the time is reduced from 2.59 seconds to 0.37 seconds for *S. enterica* and from 2.84 seconds to 0.43 seconds for *E. coli*. Regarding space consumption, `piPFP` uses considerably less space than most of the other tools, with the exception of `DandD` and the streaming version of `substring-complexity`.

[1] `https://github.com/AaronHong1024/PFP_LZ77`
[2] `https://github.com/davidecenzato/optimalBWT`
[3] `https://github.com/regindex/substring-complexity`
[4] `https://www.cs.helsinki.fi/group/pads/lz77.html`
[5] `https://github.com/lh3/ropebwt2`
[6] `https://github.com/lh3/ropebwt3`
[7] `https://github.com/jessicabonnie/dandd`

Table 3. α values for `piPFP`, `Pangrowth`, `DandD`, `Roary`, `Pantools` and `BPGA` for each species.

Species	#genes	piPFP(4,10)	piPFP(5,10)	Pangrowth	DandD	Roary	Pantools	BPGA
B. aphidicola	2911	0.22	0.24	0.30	0.25	0.20	0.21	0.20
B. cereus	295	0.68	0.71	0.78	0.79	0.68	0.68	0.69
C. botulinum	902	0.63	0.64	0.74	0.72	0.66	0.70	0.68
C. burnetii	2319	1.44	1.51	0.90	1.25	0.87	0.92	0.83
C. jejuni	242	0.84	0.87	0.87	0.90	0.76	0.78	0.69
F. tularensis	767	0.88	0.90	0.76	0.85	0.71	0.73	0.64
H. pylori	790	0.55	0.62	0.68	0.86	0.49	0.54	0.50
P. marinus	421	0.24	0.27	0.28	0.27	0.29	0.32	0.29
R. palustris	788	0.40	0.48	0.56	0.48	0.49	0.49	0.51
S. pneumoniae	818	0.85	0.88	0.75	0.78	0.73	0.77	0.69
S. pyogenes	674	0.76	0.78	0.75	0.76	0.81	0.74	0.68
Y. pestis	889	1.69	1.76	0.83	0.97	0.87	0.62	0.64

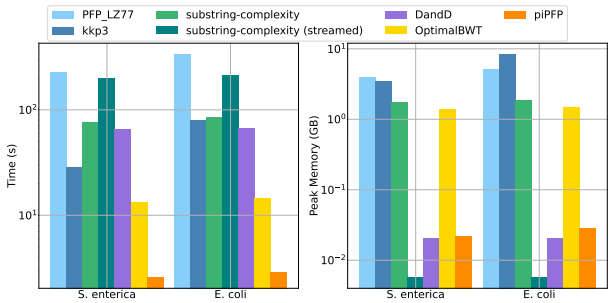

Fig. 2. Efficiency comparison of the seven tools (all single thread, y-axis in log scale). The parameters choice for `piPFP` is $w = 10$ and $p = 100$.

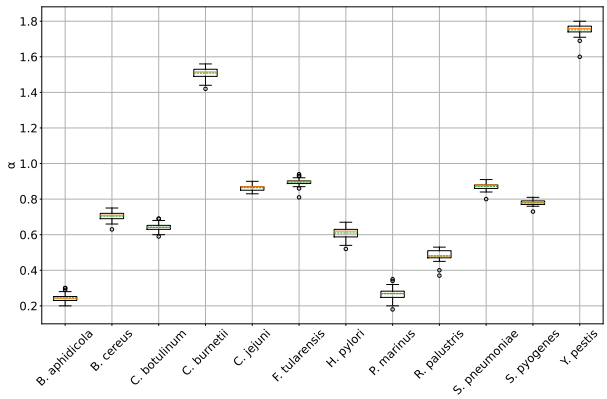

Fig. 3. Distribution of α values on the 12 datasets used for pangenome openness, taken over 24 parameter choices. See text for more details.

3.2. PFP for estimating pangenome openness

We analyzed 12 prokaryotic genomes (Table 2) with three different tools: `Pangrowth`[8] [40], `DandD` [5], and `piPFP` (our tool).

In Table 3, we report the α values that are used to classify whether a pangenome is open or closed. For completeness, we also reported the results for the gene-based tools used in [40] (Table 2 there): `Roary`[9] [39], `Pantools`[10] [23] and `BPGA`[11] [14]. Out of the 12 datasets used, `piPFP` classifies 10 datasets as open, as the other tools do, but it is not in agreement on *C. burnetii* ($\alpha = 1.44$) and on *Y. pestis* ($\alpha = 1.69$). The only other tool that also is not in agreement with the rest on *C. burnetii* is `DandD`, having $\alpha = 1.25$. We can see that the α values found by `piPFP` are comparable to the ones found by other tools in most cases.

In Figure 3, we show that even varying the parameters of `piPFP` does not have a big impact on the output α value. We tried every combination of $w = \{4, 5, 6, 7\}$ and $p = \{10, 11, 12, 13, 14, 15\}$. As depicted in the boxplots, the distribution of the different α values has a very low variance. The standard deviation in all cases lay between 0.016 and 0.042.

We evaluated the efficiency of `piPFP` against `Pangrowth` and `DandD`. The results for `Pangrowth` and `piPFP` are shown in Figure 4. `DandD`'s results are excluded because its computation time was significantly higher than that of the other two; this is due to its method for calculating pangenome growth, which involves averaging over a subset of permutations of genome unions. In this regard, we chose a subset of size 20 since the datasets are small and this parameter was chosen for datasets of similar size in the original paper [5].

In our experiments, `DandD` consistently used less space compared to the other tools, but the running times ranged from around 20 seconds up to 45 minutes depending on the dataset taken from Table 2. While using a larger amount of space, our tool always stays below 1 GB of RAM, but, on the other hand, runs in considerably less time. Compared to `Pangrowth`, `piPFP` consistently proved to be faster, used less space in every case except on *Y. Pestis*, and scaled well with the data.

3.3. Parameters of PFP

The authors of the original paper which introduced the PFP [10] included a small study on the choice of the two parameters, the window size w and the modulus p. They gave experimental results on the Pizza and Chili repetitive corpus [41] with respect to compression (i.e., our π) with 3 parameter settings $(6, 20), (8, 50), (10, 100)$, and on a biological dataset of 2.7 GB (1000 Salmonella genomes) with respect to running time and peak memory, with 15 combinations of parameters ($w = 6, 8, 10$ and $p = 50, 100, 200, 400, 800$). Based on their results, they chose the parameters $(10, 100)$ as default setting of their tool `BigBWT`.

We decided to study the choice of the parameters more in depth. First we chose regular intervals for our

[8] `https://github.com/gi-bielefeld/pangrowth`

[9] `https://github.com/sanger-pathogens/Roary`

[10] `https://git.wur.nl/bioinformatics/pantools`

[11] `https://sourceforge.net/projects/bpgatool/`

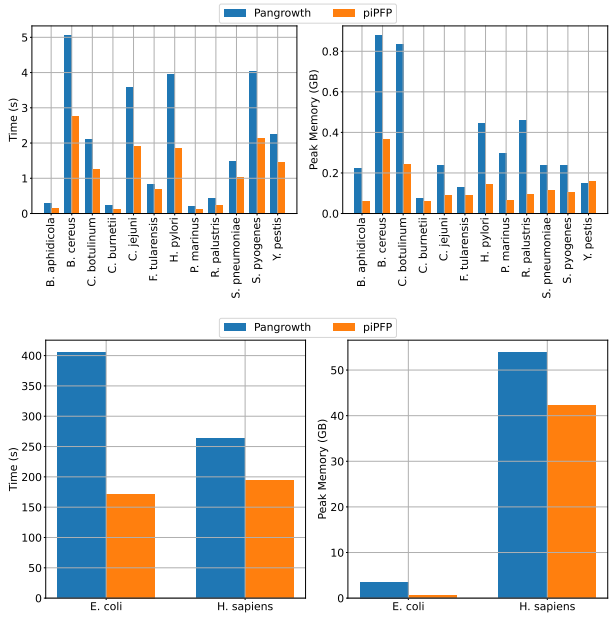

Fig. 4. Efficiency comparison of `Pangrowth` and `piPFP` on small (top) and large datasets (bottom). The parameters choice for `piPFP` is $w = 4$ and $p = 10$.

parameters, computing $\pi_{w,p}$ for $w = 5, 10, \ldots, 30$ and $p = 40, 50, 60, \ldots, 290, 300$. The results seemed to show a smooth function for all window sizes except $w = 5$, where we saw a very irregular behaviour (data not shown). Therefore, in a second set of experiments, we decided to use randomly chosen parameters, in order to avoid artifacts due to possible number theoretic issues such as divisibility (see Sec. 3.3.1). On each of our six datasets, we computed $\pi_{w,p}$ for over 1000 parameter choices, choosing in each case 15 values for w from the interval $[4, 100]$ and 70 values for p from the interval $[10, 500]$. We plot the size $\pi_{w,p}$ of the resulting $PFP(T)$ in Fig. 5. The values for w can be seen in the plots, while we give the values for p in the Appendix.

We can see that in most cases, the graph for a fixed window size w and increasing modulus p shows an initial decrease, a minimum around $p = 50$ or $p = 100$, and then a very gradual increase. This is because with modulus p, on average there should be n/p phrases, where n is the total length of the dataset. Therefore, the parse will require $4 * n/p$ bytes, since in the parse, the phrases are referred to by their rank in the dictionary, and each integer requires 4 bytes. On the other hand, the dictionary will contain unique phrases only. If the sequences are very similar, a slight increase in phrase length should result in a slightly larger dictionary. These two effects result in a slight increase in the total size $\pi_{w,p}$ for increasing p. Indeed, our experiments show that the parse length is very close to n/p for larger window sizes. (We discuss small window sizes separately below.) We observe that the SARS-CoV2 data behaves more irregularly than the other datasets for larger window and modulus values. This is possibly due to the fact that viruses mutate frequently and, therefore, the genomes have more differences when compared to the other species.

3.3.1. Small window sizes

As can be seen in several of the plots in Fig. 5, there appears to be an irregular behaviour of the size of the data structure for small window sizes. For certain values of p, there are no or very few trigger strings, resulting in one long phrase, and thus no compression at all. This phenomenon can be seen much more in detail in Fig. 6, where we plot $\pi_{w,p}$ for window sizes $w = 4, 5, \ldots, 10$ and p taking values of multiples of 10 between 40 and 300. The plot shows the results for *S. enterica*; the results are similar for our other datasets (see Appx.). For example, for $w = 5$ and $p = 160, 170$, or for $w = 4$ and $p = 40, 80, 120, 160, 170, 200, 230, 240, 250, 280$, there are no trigger strings in the text at all, resulting in one long phrase.

This effect is due to a phenomenon which we refer to as *coefficient scarsity*. As the alphabet is very small, and the characters are encoded according to their ASCII-code, it can happen easily that no string of size w is hashed to 0 modulo p. For example, it can be shown that if p is a multiple of 40, then no string of length 4 over the alphabet $\{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$ maps to 0 modulo p (see Appendix for the proof).

As can be seen from Fig. 5, this effect disappears with increasing window size.

4. Discussion

In this paper, we presented π, the size of the PFP data structure, and proposed it as a measure of repetitiveness of the text on the one hand, and as a method for deciding pangenome openness, on the other. For both uses, we showed that our new measure gives comparable results to other, more established, measures. We also presented a tool, `piPFP`, computing it, and showed that it outperforms other tools both w.r.t. running time and space. To be fair, our tool is tailor-made to compute only π, while several (but not all) of the other methods compute the measures only as a by-product. Nonetheless, we would argue that it makes sense to use a more efficient tool if one only needs the value itself.

On the theoretical side, our experiments show that the PFP heuristic does a good job in capturing the string's repetitiveness, similar to more structured (and well-defined) measures such as z, r, or δ. From our study of the pangenome openness, we conclude that PFP can also be used successfully to simulate k-mer counts.

It remains an open problem to compute $\pi^*(T)$, the minimum size of the PFP of the text T, taken over all choices of parameter pairs (w, p). Our results from Section 3.3 can serve as a starting point here. In particular, we saw that while the coefficient scarsity problem seems to be restricted to small window sizes for most of our datasets, on the SARS-CoV2 data, we observed an irregular behaviour for all window sizes, possibly due to the fact that this data set, being a viral genome, is far more variable than the others. This seems to imply that on more variable data, the choice of parameters has a higher impact on π.

Future work includes exploring further applications of π, such as for pairwise string similarity resp. distance, i.e., as a non-alignment based string similarity measure. Finally, we believe that studying the dictionary size alone could be of interest, especially as a measure of inter-sequence similarity within a dataset of pairwise similar sequences, such as a pangenome.

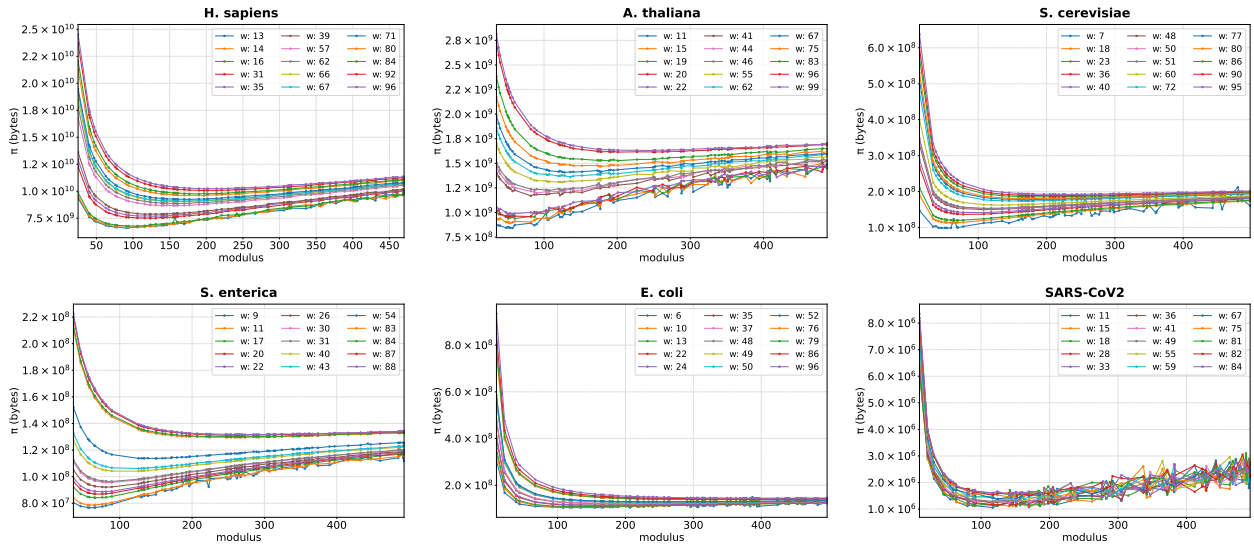

Fig. 5. $\pi_{w,p}$ with randomly chosen w and p. For each species, we chose 15 values for w from the range $[4, 100]$ and 70 values for p from $[10, 500]$.

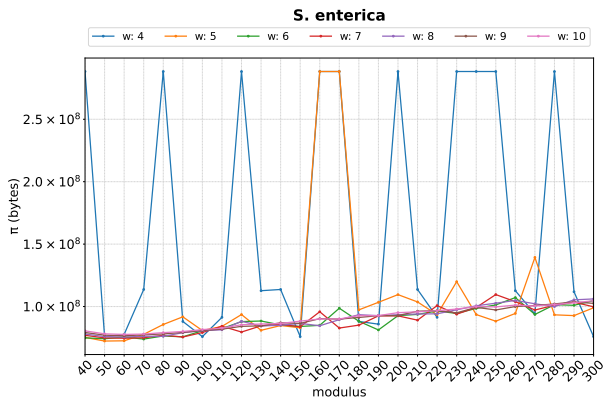

Fig. 6. Values of $\pi_{w,p}$ with small w.

5. Acknowledgments

ZsL is partially funded by the MUR PRIN project no. 2022YRB97K -'PINC' (Pangenome INformatiCs: From Theory to Applications), and by the INdAM-GNCS project no. E53C23001670001 (Compressione, indicizzazione, analisi e confronto di dati biologici).

6. Competing interests

No competing interest is declared.

References

1. Omar Ahmed, Massimiliano Rossi, Sam Kovaka, Michael C Schatz, Travis Gagie, Christina Boucher, and Ben Langmead. Pan-genomic matching statistics for targeted nanopore sequencing. *Iscience*, 24(6), 2021.
2. Omar Y Ahmed, Massimiliano Rossi, Travis Gagie, Christina Boucher, and Ben Langmead. Spumoni 2: improved classification using a pangenome index of minimizer digests. *Genome Biology*, 24(1):122, 2023.
3. Hideo Bannai, Travis Gagie, and Tomohiro I. Refining the r-index. *Theoretical Computer Science*, 812:96–108, 2020.
4. Ruben Becker, Matteo Canton, Davide Cenzato, Sung-Hwan Kim, Bojana Kodric, and Nicola Prezza. Sketching and streaming for dictionary compression. In *Proc. of the 2024 Data Compression Conference, DCC 2024*, pages 213–222. IEEE, 2024.
5. Jessica K. Bonnie, Omar Y. Ahmed, and Ben Langmead. DandD: efficient measurement of sequence growth and similarity. *IScience*, 27(3), 2024.
6. Christina Boucher, Davide Cenzato, Zsuzsanna Lipták, Massimiliano Rossi, and Marinella Sciortino. Computing the original ebwt faster, simpler, and with less memory. In *International Symposium on String Processing and Information Retrieval*, pages 129–142. Springer, 2021.
7. Christina Boucher, Davide Cenzato, Zsuzsanna Lipták, Massimiliano Rossi, and Marinella Sciortino. r-indexing the ebwt. *Information and Computation*, 298:105155, 2024.
8. Christina Boucher, Ondřej Cvacho, Travis Gagie, Jan Holub, Giovanni Manzini, Gonzalo Navarro, and Massimiliano Rossi. Pfp compressed suffix trees. In *Proc. of the 2021 Workshop on Algorithm Engineering and Experiments, ALENEX 2021*, pages 60–72. SIAM, 2021.
9. Christina Boucher, Travis Gagie, Tomohiro I, Dominik Köppl, Ben Langmead, Giovanni Manzini, Gonzalo Navarro, Alejandro Pacheco, and Massimiliano Rossi. Phoni: Streamed matching statistics with multi-genome references. In *2021 Data Compression Conference (DCC)*, pages 193–202. IEEE, 2021.
10. Christina Boucher, Travis Gagie, Alan Kuhnle, Ben Langmead, Giovanni Manzini, and Taher Mun. Prefix-free parsing for building big BWTs. *Algorithms Mol. Biol.*, 14(1):13:1–13:15, 2019.
11. Michael Burrows and David J. Wheeler. A block-sorting lossless data compression algorithm. Technical report, DIGITAL System Research Center, 1994.
12. Davide Cenzato, Veronica Guerrini, Zsuzsanna Lipták, and Giovanna Rosone. Computing the optimal bwt of very large

string collections. In *2023 Data Compression Conference (DCC)*, pages 71–80. IEEE, 2023.
13. Davide Cenzato, Francisco Olivares, and Nicola Prezza. On computing the smallest suffixient set. In *International Symposium on String Processing and Information Retrieval*, pages 73–87. Springer, 2024.
14. Narendrakumar M. Chaudhari, Vinod Kumar Gupta, and Chitra Dutta. Bpga-an ultra-fast pan-genome analysis pipeline. *Scientific Reports*, 6(1):24373, 2016.
15. The Computational Pan-Genomics Consortium. Computational pan-genomics: status, promises and challenges. *Briefings in Bioinformatics*, 19(1):118–135, 2018.
16. Paolo Ferragina and Giovanni Manzini. Opportunistic data structures with applications. In *Proc. of the 41st Annual Symposium on Foundations of Computer Science, FOCS 2000*, pages 390–398. IEEE Computer Society, 2000.
17. Travis Gagie, Garance Gourdel, and Giovanni Manzini. Compressing and indexing aligned readsets. In *Proc. of the 21st International Workshop on Algorithms in Bioinformatics, WABI 2021*, volume 201 of *LIPIcs*, pages 13:1–13:21. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021.
18. Travis Gagie, Tomohiro I, Giovanni Manzini, Gonzalo Navarro, Hiroshi Sakamoto, and Yoshimasa Takabatake. Rpair: Rescaling repair with rsync. In *Proc. of the 26th International Symposium on String Processing and Information Retrieval, SPIRE 2019*, volume 11811 of *Lecture Notes in Computer Science*, pages 35–44. Springer, 2019.
19. Travis Gagie, Gonzalo Navarro, and Nicola Prezza. Fully functional suffix trees and optimal text searching in bwt-runs bounded space. *J. ACM*, 67(1):2:1–2:54, 2020.
20. Adrián Goga, Lore Depuydt, Nathaniel K. Brown, Jan Fostier, Travis Gagie, and Gonzalo Navarro. Faster maximal exact matches with lazy LCP evaluation. In *Proc. of the 2024 Data Compression Conference, DCC 2024*, pages 123–132. IEEE, 2024.
21. Harold Stanley Heaps. *Information Retrieval: Computational and Theoretical Aspects*. Academic Press, Inc., 1978.
22. Aaron Hong, Massimiliano Rossi, and Christina Boucher. LZ77 via prefix-free parsing. In *Proc. of the 25th Symposium on Algorithm Engineering and Experiments, ALENEX 2023*, pages 123–134. SIAM, 2023.
23. Eef M. Jonkheer, Dirk-Jan M. van Workum, Siavash Sheikhizadeh Anari, Balázs Brankovics, Jorn R. de Haan, Lidija Berke, Theo A. J. van der Lee, Dick de Ridder, and Sandra Smit. Pantools v3: functional annotation, classification and phylogenomics. *Bioinformatics*, 38(18):4403–4405, 2022.
24. Juha Kärkkäinen, Dominik Kempa, and Simon J. Puglisi. Linear time lempel-ziv factorization: Simple, fast, small. In *Proc. of the 24th Annual Symposium on Combinatorial Pattern Matching, CPM 2013*, volume 7922 of *Lecture Notes in Computer Science*, pages 189–200. Springer, 2013.
25. Richard M. Karp and Michael O. Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 31(2):249–260, 1987.
26. Justin Kim, Rahul Varki, Marco Oliva, and Christina Boucher. Re^{2}pair: Increasing the scalability of repair by decreasing memory usage. In *Proc. of the 32nd Annual European Symposium on Algorithms, ESA 2024*, volume 308 of *LIPIcs*, pages 78:1–78:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2024.
27. Tomasz Kociumaka, Gonzalo Navarro, and Nicola Prezza. Toward a definitive compressibility measure for repetitive sequences. *IEEE Trans. Inf. Theory*, 69(4):2074–2092, 2023.
28. Alan Kuhnle, Taher Mun, Christina Boucher, Travis Gagie, Ben Langmead, and Giovanni Manzini. Efficient construction of a complete index for pan-genomics read alignment. *J. Comput. Biol.*, 27(4):500–513, 2020.
29. Abraham Lempel and Jacob Ziv. On the complexity of finite sequences. *IEEE Transactions on Information Theory*, 22(1):75–81, 1976.
30. Heng Li. Fast construction of FM-index for long sequence reads. *Bioinformatics*, 30(22):3274–3275, 2014.
31. Heng Li. BWT construction and search at the terabase scale. *Bioinformatics*, 40(12):btae717, 2024.
32. Udi Manber and Eugene W. Myers. Suffix arrays: A new method for on-line string searches. *SIAM J. Comput.*, 22(5):935–948, 1993.
33. César Martínez-Guardiola, Nathaniel K. Brown, Fernando Silva-Coira, Dominik Köppl, Travis Gagie, and Susana Ladra. Augmented thresholds for MONI. In *Proc. of the 2023 Data Compression Conference, DCC 2023*, pages 268–277. IEEE, 2023.
34. Gonzalo Navarro. Indexing highly repetitive string collections, part ii: Compressed indexes. *ACM Computing Surveys (CSUR)*, 54(2):1–32, 2021.
35. Gonzalo Navarro. Indexing highly repetitive string collections, part I: repetitiveness measures. *ACM Comput. Surv.*, 54(2):29:1–29:31, 2022.
36. Marco Oliva, Davide Cenzato, Massimiliano Rossi, Zsuzsanna Lipták, Travis Gagie, and Christina Boucher. CSTs for terabyte-sized data. In *Proc. of the 2022 Data Compression Conference, DCC 2022*, pages 93–102. IEEE, 2022.
37. Marco Oliva, Travis Gagie, and Christina Boucher. Recursive prefix-free parsing for building big bwts. In *Proc. of the 2023 Data Compression Conference, DCC 2023*, pages 62–70. IEEE, 2023.
38. oneAPI Threading Building Blocks (oneTBB), UXL Foundation, https://github.com/uxlfoundation/oneTBB.
39. Andrew J. Page, Carla A. Cummins, Martin Hunt, Vanessa K. Wong, Sandra Reuter, Matthew T. G. Holden, Maria Fookes, Daniel Falush, Jacqueline A. Keane, and Julian Parkhill. Roary: rapid large-scale prokaryote pan genome analysis. *Bioinformatics*, 31(22):3691–3693, 2015.
40. Luca Parmigiani, Roland Wittler, and Jens Stoye. Revisiting pangenome openness with k-mers. *Peer Community Journal*, 4, 2024.
41. http://pizzachili.dcc.uchile.cl/repcorpus.html. Last accessed 22 Jan. 2025.
42. Sofya Raskhodnikova, Dana Ron, Ronitt Rubinfeld, and Adam D. Smith. Sublinear algorithms for approximating string compressibility. *Algorithmica*, 65(3):685–709, 2013.
43. Massimiliano Rossi, Marco Oliva, Ben Langmead, Travis Gagie, and Christina Boucher. MONI: A pangenomic index for finding maximal exact matches. *J. Comput. Biol.*, 29(2):169–187, 2022.
44. Vikram S Shivakumar, Omar Y Ahmed, Sam Kovaka, Mohsen Zakeri, and Ben Langmead. Sigmoni: classification of nanopore signal with a compressed pangenome index. *Bioinformatics*, 40(Supplement 1):i287–i296, 06 2024.
45. Hervé Tettelin, David Riley, Ciro Cattuto, and Duccio Medini. Comparative genomics: the bacterial pan-genome.

Current opinion in microbiology, 11(5):472–477, 2008.

46. Mohsen Zakeri, Nathaniel K Brown, Omar Y Ahmed, Travis Gagie, and Ben Langmead. Movi: a fast and cache-efficient full-text pangenome index. *iScience*, 27(12), 2024.
47. Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.
48. Jacob Ziv and Abraham Lempel. Compression of individual sequences via variable-rate coding. *IEEE Trans. Inf. Theory*, 24(5):530–536, 1978.