Bloom Filter Trie – a data structure for pan-genome storage

Guillaume Holley[1,2,3], Roland Wittler[1,2,3], and Jens Stoye[1,2,3]

[1] Genome Informatics, Faculty of Technology, Bielefeld University, Germany
[2] Center for Biotechnology, Bielefeld University, Germany
[3] International Research Training Group 1906, Bielefeld University, Germany

Abstract. High throughput sequencing technologies have become fast and cheap in the past years. As a result, large-scale projects started to sequence tens to several thousands of genomes per species, producing a high number of sequences sampled from each genome. Such a highly redundant collection of very similar sequences is called a pan-genome. It can be transformed into a set of sequences "colored" by the genomes to which they belong. A colored de-Bruijn graph (C-DBG) extracts from the sequences all colored k-mers, strings of length k, and stores them in vertices. In this paper, we present an alignment-free, reference-free and incremental data structure for storing a pan-genome as a C-DBG: the Bloom Filter Trie. The data structure allows to store and compress a set of colored k-mers, and also to efficiently traverse the graph. Experimental results prove better performance compared to another state-of-the-art data structure.

1 Introduction

A *string* x is a sequence of characters drawn from a finite, non-empty set, called the *alphabet* $\mathcal{A}$. Its *length* is denoted by $|x|$. The character at position i is denoted by $x[i]$, the substring starting at position i and ending at position j by $x[i..j]$. Strings are concatenated by juxtaposition. If $x = ps$ for (potentially empty) strings p and s, then p is a *prefix* and s is a *suffix* of x.

A *genome* is the collection of all inheritable material of a cell. Ideally it can be represented as a single string over the DNA alphabet $\mathcal{A} = \{a, c, g, t\}$ (or as a few strings in case of species with multiple chromosomes). In practice, however, genomes in databases are often less perfect, either left unchanged in form of the raw data as produced by sequencing machines (millions of short sequences called *reads*), or after some incomplete assembly procedure in form of contiguous chromosome regions (hundreds of *contigs* of various lengths). We are interested in the problem of storing and compressing a set of multiple highly similar genomes, e.g. the pan-genome of a bacterial species, comprising hundreds, or even thousands of strains that share large sequence parts, but differ by individual mutations from one another. An abstract structure that has been proposed for this task is the *colored de-Buijn graph* (C-DBG) [13]. It is a directed graph $G = (V_G, E_G)$ in which each vertex $v \in V_G$ represents a k-mer, a string of length k over $\mathcal{A}$, associated with a set of colors representing the genomes in which the k-mer occurs. A directed edge $e \in E_G$ from vertex v to vertex v', respectively from k-mer x to k-mer x', exists if $x[2..k] = x'[1..k-1]$. Each k-mer x has $|\mathcal{A}|$ possible successors $x[2..k]c$ and $|\mathcal{A}|$ possible predecessors $cx[1..k-1]$ with $c \in \mathcal{A}$. An implementation of such a graph does not have to store edges since they are implicitly given by vertices overlapping on $k-1$ characters.

In this paper, we propose a new data structure for indexing and compressing a pan-genome as a C-DBG, the Bloom Filter Trie (BFT). It allows any format for the input genomes (completely sequenced, set of contigs, set of reads, and even mixtures of them), is alignment-free, reference-free and incremental, i.e., it does not need to be entirely rebuilt when a new genome is inserted. BFTs provide insertion and look-up operations for strings of fixed length associated with an annotation.

In the next section, existing data structures and software for pan-genome representation are reviewed. Section 3 presents the BFT and Section 4 the operations it supports. Then, Section 5 describes the traversal of a C-DBG stored as a BFT. Finally, Section 6 contains experimental results showing the performance of the data structure. Section 7 concludes. Our implementation of the BFT is available at `https://github.com/GuillaumeHolley/BloomFilterTrie`.

2 Existing approaches

The BFT, as well as existing tools for pan-genome storage, uses a variety of basic data structures reviewed in the following. Existing tools for pan-genome storage will then be discussed in Section 2.2.

2.1 Data structures

One common way to index and compress a set of strings is to use as a first step the *Burrows-Wheeler Transform* (BWT) [2] that rearranges the input data to enable better compression by aggregating characters with similar context. For multiple sets of strings, a disk-based approach [4] or different terminator characters must be used. The *FM-Index* [9] allows to count and locate the occurrences of a substring in the BWT.

Introduced by Bloom [1], a *Bloom filter* (BF) records the presence of elements in a set. Based on the hash table principle, look-up and insertion times are constant. The BF is composed of a bit array $B[1..m]$, initialized with 0s, in which the presence of n elements is recorded. A set of f hash functions $h_1, ..., h_f$ is used, such that for an element e, $h_i(e) : e \to \{1, .., m\}$. Inserting an element into B and testing for its presence are then

$$\mathsf{Insert}(e, B) : B[h_i(e)] \leftarrow 1 \text{ for all } i = 1, ..., f$$

and

$$\mathsf{MayContain}(e, B) : \bigwedge_{i=1}^{f} B[h_i(e)],$$

respectively, where $\bigwedge$ is the logical conjunction operator. The BF does not generate false negatives but may generate false positives, as MayContain can report the presence of elements which are not present but a result of independent insertions.

The *Sequence Bloom Tree* (SBT) [21] is a binary tree with BFs as vertices. An internal vertex is the union of its two children BFs, i.e., a BF where a cell is set to 1 if the cell at the same position in at least one of the two children is set to 1.

A *trie* [10] is a rooted edge-labeled tree $T = (V_T, E_T)$ storing a set of strings. Each edge $e \in E_T$ is labeled with a character. A path from the root to a leaf represents the string obtained by concatenating all the characters on this path. The depth of a vertex v in T is denoted by $depth(v, T)$ and is the number of edges between the root of T and v. The height of T, denoted by $height(T)$, is the number of edges on the longest path from the root of T to a leaf. The *burst trie* [11] is an efficient implementation of a trie which reduces its number of branches by compressing sub-tries into leaves. Its internal vertices are labeled with multiple prefixes of length 1, linked to children. The leaves are labeled with multiple suffixes of arbitrary length. A leaf has a limited capacity of suffixes and is *burst* when this capacity is exceeded. A burst splits suffixes of a leaf into prefixes of length 1, linked to new leaves representing the remaining suffixes.

2.2 Software for pan-genome storage

Existing tools for pan-genome storage are mostly alignment-based or reference-based and take a set of assembled genomes as input. Alignments naturally exhibit shared and unique regions of the pan-genome but are computationally expensive to obtain. In addition, misalignments can lead to an inaccurate estimation of the pan-genome regions [7]. PanCake [8] is an extension of string graphs, known from genome assembly [17], which achieves compression based on pairwise alignments. Experiments showed compression ratios of 3:1 to 5:1. Nguyen *et al.* [18] formulated the pan-genome construction problem as an optimization problem of arranging alignment blocks for a set of genomes partitioned by homology. The complexity of the problem has been shown to be NP-hard, and a heuristic using Cactus graphs [19] was provided. A multiple sequence alignment is required for creating the blocks, another NP-hard problem.

Among the reference-based tools, Huang *et al.* [12] proposed to build a pan-genome by adding all the variants detected between a set of genomes to a reference genome. The BWT of the augmented reference is then computed and can be used by an aligner based on the FM-Index. While being more accurate with the augmented reference genome than BWA [14] with the reference alone, the aligner is between 10 to 100 times slower, uses significantly more memory and can introduce false positive alignments. RCSI [22] (Referentially Compressed Search Index) uses referential compression with a compressed suffix tree to store a pan-genome and to search for exact or inexact matches. The inexact matching allows a limited number of edit distance operations. 1,092 human genomes totaling 3.09 TB of data were compressed into an index of 115 GB, offering a compression ratio of about 28:1. Yet, the index is built for a maximum length query and a maximum number of edit operations.

Close to our approach is SplitMEM [16], which uses a C-DBG to build a pan-genome made of assembled genomes and to extract the shared regions. Although the C-DBG is directly constructed in a compressed way, where a non-branching path is stored in a single vertex, the resulting size of the data structure is larger than the sum of the original sizes of the input sequences, due to the use of an augmented suffix tree.

Recently, the authors of Khmer [5] introduced in their software library a de-Bruijn graph labeling method. Khmer provides a lightweight representation of de-Bruijn graphs [20] based on Bloom filters and a graph labeling method based on graph partitioning. Unfortunately, this functionality was made available only a few days before submission.

The SBT [21] is an alignment-free, reference-free and incremental data structure that allows to label sequences with their colors. The proposed tool is designed to index and compress data from sequencing experiments for effective query of full-lenth genes or transcripts by separation into k-mers. A leaf of an SBT is used to represent a sequencing experiment by extracting all its k-mers and storing them in the BF of the leaf. SBTs do not represent exactly the set of k-mers of the sequencing experiments they contain, though, due to the inexact nature of BFs.

3 The Bloom Filter Trie

The Bloom Filter Trie (BFT) that we propose in this paper is an implementation of a C-DBG. It is based on a burst trie and is used to store k-mers associated with a set of colors. For the moment we may assume that colors are represented by a bit array *color* initialized with 0s. Each color has an index i_{color} such that $color_x[i_{color}] = 1$ records that k-mer x has color i_{color}. Sets of colors will later be compressed as explained in Section 4.3. All arrays in a BFT are dynamic: An insertion at position *pos* in an array reallocates it and shifts every cell having an index $\geq pos$ by one position.

In the following, let $t = (V_t, E_t)$ be a BFT created for a certain value of k where we assume that k is a multiple of an integer l such that k-mers can be split into $\frac{k}{l}$ equal-length substrings. The maximum height of t is $height_{max}(t) = \frac{k}{l} - 1$. The alphabet we consider is the DNA alphabet $\mathcal{A} = \{a, c, g, t\}$, and because $|\mathcal{A}| = 4$, each character can be stored using two bits. A vertex in a BFT is a list of containers, zero or more of which are *compressed*, plus zero or one *uncompressed* container. In the following, we will explain how the containers are represented and how an uncompressed container is burst when its capacity is exceeded.

3.1 Uncompressed container

An uncompressed container of a vertex v in a BFT is a limited capacity set of tuples $<s, color_{ps}>$ where s is a suffix and p is the prefix represented by the path from the root to v. Uncompressed containers are burst when the number of suffixes stored exceeds their capacity $c > 0$. Then, each suffix s of the uncompressed container is split into a prefix s_{pref} of length l and a suffix s_{suf} of length $|s| - l$ such that $s = s_{pref}s_{suf}$. Prefixes are stored in a new compressed container. Suffixes, attached with their colors, are stored in new uncompressed containers, themselves stored in the children of the compressed container. An example of a BFT and a bursting is given in Figure 1.

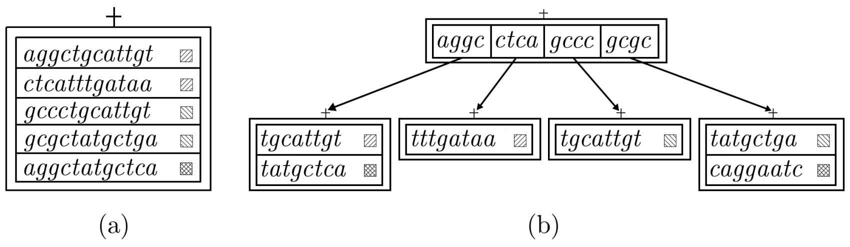

Fig. 1. Insertion of six suffixes (that are here complete k-mers) with different colors (boxes with diagonal lines) into a BFT with $k = 12$, $l = 4$ and $c = 5$. In (a), the first five suffixes are inserted at the root into an uncompressed container. When a sixth suffix *gcgccaggaatc* is inserted, the uncompressed container exceeds its capacity and is burst, resulting in the BFT structure shown in (b).

3.2 Compressed container

A bursting replaces an uncompressed container by a compressed one, used to:

- store q suffix prefixes in compressed form (in Figure 1(b), $q = 4$),
- store links to children containing the suffixes,
- reconstruct suffix prefixes and find the corresponding children.

In the following, each suffix prefix s_{pref} is split into a prefix a and a suffix b with respective binary representations α and β. A compressed container is composed of four structures *quer*, *pref*, *suf* and *clust*, where:

- *quer* is a BF represented as a bit array of length m and f hash functions, used to record and filter for the presence of q suffix prefixes;
- *pref* is a bit array of $2^{|\alpha|}$ bits initialized with 0s and used to record prefix presence exactly. Here the binary representation α of a prefix a is interpreted as an integer such that $pref[\alpha]$ set to 1 records the presence of a;
- *suf* is an array of q suffixes b sorted in ascending lexicographic order of the original suffix prefixes they belong to;
- *clust* is an array of q bits, one per suffix of array *suf*, that represents cluster starting points. A cluster is a list of consecutive suffixes in array *suf* that share the same prefix. It has an index $i_{cluster}$ with $1 \leq i_{cluster} \leq 2^{|\alpha|}$ and

a start position $pos_{cluster}$ in the array *suf* with $i_{cluster} \leq pos_{cluster} \leq q$. Position *pos* in array *clust* is set to 1 to indicate that the suffix in *suf*[*pos*] starts a cluster because it is the lexicographically smallest suffix of its cluster. A cluster contains $n \geq 1$ suffixes and, therefore, position i in array *clust* is set to 0 for $pos < i < pos + n$. The end of a cluster is indicated by the beginning of the next cluster or if $pos \geq q$.

For example, the internal representation of the compressed container shown in Figure 1(b) with $|a| = 2$ and $|b| = 2$ would be:

quer	0	0	1	0	1	1	0	0	0	1	1	1				
pref	0	0	1	0	0	0	0	1	0	1	0	0	0	0	0	0

suf	*gc*	*ca*	*cc*	*gc*
clust	1	1	1	0

The size required by a set of q substrings in a compressed container is $m + 2^{|\alpha|} + q \cdot (|\beta| + 1)$ bits. A bursting minimizes this size by choosing a prefix length $|a|$ and a BF size m such that the set of substrings stored in a compressed container does not occupy more memory than their original representation in an uncompressed container, i.e., $m + 2^{|\alpha|} \leq q \cdot (|\alpha| - 1)$. Each suffix prefix inserted after a bursting costs only $|\beta| + 1$ bits. When the average size per suffix prefix stored is close to $|\beta| + 1$ bits, arrays *pref*, *suf* and *clust* can be recomputed by increasing $|a|$ and decreasing $|b|$, such that $2^{|\alpha'|} + q \cdot |\beta'| < 2^{|\alpha|} + q \cdot |\beta|$, where α' and β' are the values of α and β, respectively, after resizing.

4 Operations supported by the Bloom Filter Trie

The BFT supports all operations necessary for storing, traversing and searching a pan-genome, as well as to extract the relevant information of the contained genomes and subsets thereof. Here we describe the most basic ones of them, Look-up (Section 4.1) and Insertion (Section 4.2), as well as how the sets of colors are compressed (Section 4.3). Traversal of the graph is discussed in Section 5.

The algorithms use two auxiliary functions. HammingWeight(α, *pref*) counts the number of 1s in *pref*[1..α] and corresponds to how many prefixes represented in array *pref* are lexicographically smaller than or equal to an inserted prefix a with binary representation α. This requires $\mathcal{O}(2^{|\alpha|})$ time. The second function, Rank(i, *clust*), iterates over array *clust* from its first position until the i-th entry 1 is found and returns the position of this entry. It corresponds to the start position of cluster i in array *clust*. If the entry is not found, the function returns $|clust|+1$ as a position. While Rank could be implemented in $\mathcal{O}(1)$ time [9], we use a more naive but space efficient $\mathcal{O}(q)$ time implementation.

4.1 Look-up

The function that tests whether a suffix prefix $s_{pref} = ab$ with binary representation $\alpha\beta$ is stored in a compressed container cc is given in Algorithm 1. Line 1 uses MayContain to filter for presence of s_{pref} inside cc by querying the BF *quer* in $\mathcal{O}(f)$ time. If present as a true or false positive, the presence of the prefix a is verified in the array *pref* in $\mathcal{O}(1)$ time. If a is not present, this was clearly a false positive and nothing else has to be done. If a is present, line 2 computes in $\mathcal{O}(2^{|\alpha|})$ time the Hamming weight i of a, i.e., the index of the cluster in which suffix b is possibly situated. Line 3 locates the rank of i, i.e., the start position of the cluster, and lines 4–7 compare the suffixes of the cluster to b. Lines 3–7 are computed in $\mathcal{O}(q)$ time. Algorithm 1 has therefore a worst case running time of $\mathcal{O}(f + 2^{|\alpha|} + q)$.

Algorithm 1 Contains(ab, cc)

```
1: if MayContain($ab$, $cc.quer$) and $cc.pref[\alpha] = 1$ then
2:     $i \leftarrow$ HammingWeight($\alpha$, $cc.pref$)
3:     $start \leftarrow$ Rank($i$, $cc.clust$)
4:     $pos \leftarrow start$
5:     while $pos \leq |suf|$ and ($pos = start$ or $cc.clust[pos] = 0$) do
6:         if $cc.suf[pos] = b$ then return $true$
7:         $pos \leftarrow pos + 1$
8: return $false$
```

The function that tests whether a k-mer x is present in a BFT $t = (V_t, E_t)$ is given in Algorithm 2. Each vertex $v \in V_t$ represents k-mer suffixes possibly stored in its uncompressed container or rooted from its compressed containers. The look-up traverses t and, for a vertex v, queries its containers one after the other for suffix $x_{suf} = x[l \cdot depth(v,t) + 1..k]$. If the queried container is a compressed container, its BF *quer* is queried for $x_{suf}[1..l]$ and, in case of a positive answer, the function Contains is used for an exact membership of $x_{suf}[1..l]$. If it is found, the traversing procedure continues recursively on the corresponding child. The absence of $x_{suf}[1..l]$ indicates the absence of x in t since $x_{suf}[1..l]$ cannot be in another container of v. If the container is an uncompressed container, its suffixes are compared to x_{suf}. As an uncompressed container has no children, a match indicates the presence of the k-mer. Algorithm 2 is initially called as TreeContains($x, 1, l, root$). In the worst case, all vertices on a traversed path represent all possible suffix prefixes and the BFs *quer* have a false positive ratio of 0. In such case, each traversed vertex contains $\lceil \frac{|\mathcal{A}|^l}{c} \rceil$ containers. The longest path of a BFT has $height_{max}(t) + 1$ vertices. Therefore, the worst case time of TreeContains is $\mathcal{O}(height_{max}(t) \cdot \lceil \frac{|\mathcal{A}|^l}{c} \rceil \cdot (f + 2^{|\alpha|} + q))$.

4.2 Insertion

Prior to any k-mer insertion into a BFT t, a look-up verifies if the k-mer is already present. If it is, only its set of colors is modified. Otherwise, the look-up stops the trie traversal on a container *cont* of a vertex v where the searched suffix prefix or k-mer suffix is not present. If *cont* is an uncompressed container, the

Algorithm 2 $\mathsf{TreeContains}(x, i, l, v)$

```
1: for each container $cont$ in $v$ do
2:     if $cont$ is compressed and $\mathsf{MayContain}(x[i..i+l-1], cont.quer)$ then
3:         if $\mathsf{Contains}(x[i..i+l-1], cont)$ then
4:             $v \leftarrow$ child associated with $x[i..i+l-1]$ in $cont.suf$
5:             return $\mathsf{TreeContains}(x, i+l, l, v)$
6:         else return $false$
7:     else if $cont$ is uncompressed then
8:         for each $<s, color_{x[1..i-1]s}>$ in $cont$ do
9:             if $s = x[i..k]$ then return $true$
10: return $false$
```

insertion of the k-mer suffix and its color is a simple $\mathcal{O}(c)$ time process. If *cont* is compressed, the insertion of suffix prefix $s_{pref} = ab$ is a bit more intricate. In fact, it will only be triggered if *cont* is the first compressed container of v to have s_{pref} as a false positive ($\mathsf{MayContain}(s_{pref}, cont.quer) = true$ and $\mathsf{Contains}(s_{pref}, cont) = false$). False positives are therefore "recycled", which is a nice property of BFTs: The BF *quer* remains unchanged, and only *pref*, *suf* and *clust* need to be updated in a way similar to Algorithm 1: The presence of prefix a must be first verified by testing the value of $pref[\alpha]$ where α is the binary representation of a. If $pref[\alpha] = 0$, prefix a is not present and is recorded by setting $pref[\alpha]$ to 1. Then, the index $id_{cluster}$ and start position $pos_{cluster}$ of the new cluster are computed using $\mathsf{HammingWeight}$ and Rank. The suffix b is inserted into $suf[pos_{cluster}]$ and a 1 into $clust[pos_{cluster}]$. This takes $\mathcal{O}(2^{|\alpha|} + 2q)$ time. If $pref[\alpha] = 1$ prior to insertion, prefix a is already present, and $id_{cluster}$ and $pos_{cluster}$ have already been computed by $\mathsf{Contains}(s_{pref}, cont)$. Let n be the number of suffixes in cluster $id_{cluster}$. Suffix b is inserted into $suf[pos]$ such that $pos_{cluster} \leq pos \leq pos_{cluster} + n$ and $suf[pos-1] < suf[pos]$. If $pos = pos_{cluster}$, b starts its cluster: A 1 is inserted into $clust[pos]$ and $clust[pos+1]$ is set to 0. Otherwise, a 0 is inserted into $clust[pos]$. This takes $\mathcal{O}(2q)$ time. The worst case time insertion of a k-mer is $\mathcal{O}(d + 2^{|\alpha|} + 2q)$ with d being the worst case time look-up.

The internal representation of the compressed container shown in Figure 1(b) after insertion of the suffix prefix *gtat* is given below (inserted parts are highlighted). The presence of prefix *gt* is recorded in *pref*[12]. Then, its cluster index and start position are computed as 4 and 5, respectively. Consequently, after reallocation of arrays *suf* and *clust*, suffix *at* is inserted in *suf*[5] and *clust*[5] is set to 1 to indicate that *suf*[5] starts a new cluster.

quer	0	0	1	0	1	1	0	0	0	1	1	1				
pref	0	0	1	0	0	0	0	1	0	1	0	**1**	0	0	0	0

suf	*gc*	*ca*	*cc*	*gc*	**at**
clust	1	1	1	0	**1**

4.3 Color compression

Remember from Section 3 that color sets associated with k-mers in a C-DBG are initially stored as bit arrays in BFTs. However, these can be compressed. To this end, a list of all color sets occurring in the BFT is built and sorted in decreasing order of total size, i.e., the number of k-mers sharing a color set multiplied by its size. Then, by iterating over the list, each color set is added incrementally to an external array if the integer encoding its position in the array uses less space than the size of the color set itself. Finally, each color set present in the external array is replaced in the BFT by its position in the external array.

5 Traversing successors and predecessors

Let t be a BFT that represents a C-DBG G. For a k-mer x, visiting all its predecessors or successors in G, denoted $pred(x, G)$ and $succ(x, G)$, respectively, implies the look-up of $|\mathcal{A}|$ different k-mers in t. Such a look-up would visit in the worst case $|\mathcal{A}| \cdot (height_{max}(t) + 1)$ vertices in t. This section describes how to reduce the number of vertices and containers visited in t during the traversal of a vertex in G.

Observation 1. *Let G be a C-DBG represented by a BFT t and x a k-mer corresponding to a vertex of G. All k-mers of $succ(x, G)$ share $x[2..k]$ as a common prefix and therefore share a common subpath in t starting at the root. On the other hand, k-mers of $pred(x, G)$ have different first characters and, therefore, except for the root of t do not share a common subpath. Hence, the maximum number of visited vertices in t for all k-mers of $succ(x, G)$ is $1 + height_{max}(t)$ and for all k-mers of $pred(x, G)$ is $1 + |\mathcal{A}| \cdot height_{max}(t)$.*

Lemma 1. *Let G be a C-DBG represented by a BFT t, x a k-mer in t and v a vertex of t that terminates the shared subpath of the k-mers in $succ(x, G)$. If $depth(v, t) = height_{max}(t)$, $succ(x, t)$ suffixes may be stored in any container of v. If not, they are stored in the uncompressed container of v.*

Proof. A vertex v is the root of a sub-trie storing k-mer suffixes of length $l \cdot (height_{max}(t) - depth(v, t) + 1)$ with $l = \frac{k}{height_{max}(t)+1}$. Let s be a k-mer suffix of $succ(x, t)$ rooted at a vertex $v \in V_t$. If $depth(v, t) \neq height_{max}(t)$ but s is rooted at a compressed container in v, then this compressed container stores $s[1..l]$, and $s[l+1..|s|]$ is rooted in one of its children. As the divergent character between the k-mer suffixes of $succ(x)$ is in position $|s| - 1$, this character is in $s[l + 1..|s|]$, rooted at one child of this compressed container. Therefore v does not terminate the common subpath shared by $succ(x, t)$ k-mers. $\square$

Lemma 1 proves that the only two cases where a look-up of $pred(x, G)$ or $succ(x, G)$ must search in different containers of a vertex are:

- searching at the root of t for k-mers of $pred(x, G)$,
- if $depth(v, t) = height_{max}(t)$, searching at vertex v for suffixes of $succ(x, G)$.

Restricting the hash functions used in the compressed containers to take only positions 2 through $l-1$ into account, allows to limit the search space.

Lemma 2. *Let t be a BFT where the f hash functions h_i of quer have the form $h_i(s_{pref}) : s_{pref}[2..l-1] \rightarrow \{1,..,m\}$ for $i = 1,...,f$. Then, for a vertex v of t and a suffix prefix s_{pref}, all possible substrings $s'_{pref} = c_1 s_{pref}[2..l-1]c_2$ are contained in the same container of v.*

Proof. Assume a k-mer suffix s inserted in a vertex v of t. A look-up for s analyzes the containers of v from the head to the tail of the container list. In the worst case, s can be rooted, according to BFs *quer*, in all compressed containers as a true positive or as a false positive. However, a look-up stops either on the first compressed container claiming to contain the suffix prefix $s_{pref} = s[1..l]$, or on the uncompressed container. As the hash functions of *quer* consider only $s_{pref}[2..l-1]$, a look-up will therefore stop on the same container for any substring $s'_{pref} = c_1 s_{pref}[2..l-1]c_2$. □

As a consequence of Lemma 2, each suffix prefix s_{pref} stored or to store in arrays *pref*, *suf* and *clust* is modified such that $s_{pref} = s_{pref}[2..l]s_{pref}[1]$, which guarantees that all $s'_{pref} = s_{pref}[2..l-1]c_2c_1$ are in the same container. Furthermore, suffixes stored in array *suf* are required to have a minimum length of two characters to ensure that characters c_1 and c_2, the variable parts between the different s'_{pref}, are stored in array *suf*. Hence, as all s'_{pref} share $s_{pref}[2..l-1]$ as a prefix, they share the same cluster in arrays *suf* and *clust*. Suffix prefixes $s'_{pref} = s_{pref}[1..l-1]c_2$ also have consecutive suffixes in their cluster.

6 Evaluation

We implemented the BFT in C and compared it to the SBT [21], version 0.3.1, on a mid-class laptop with an SSD hard drive and an Intel Core i5-4300M processor cadenced at 2.6 GHz. All software was run with a single thread. Both data structures were used to represent one real and one simulated pan-genome dataset. The real dataset (NCBI BioProject PRJEB5438) consists of raw sequencing data from 473 clinical isolates of *Pseudomonas aeruginosa*, sampled from 34 patients, resulting in 844.37 GB of FASTQ files. The simulated dataset was generated from 19 strains of *Yersinia pestis*. For each strain, we used Wgsim [1] to create 6,000,000 reads of length 100 with a substitution sequencing error rate of 0.5%, resulting in 31 GB of FASTQ files. We first used KmerGenie [3] on a subsample of the files for each dataset to estimate the best k-mer length and the minimum number of occurrences for considering a k-mer valid (not resulting from a sequencing error). A length of $k = 63$ with a minimum number of 3 occurrences was selected for the real and a length of $k = 54$ with a minimum of 15 occurrences for the simulated data set.

For the BFT, we used KMC2 [6] to extract all valid k-mers from each genome. The capacity c influences the compression ratio as well as the time for insertion

[1] `https://github.com/lh3/wgsim`

and look-up. We chose a value of $c = 248$ as it showed a good tradeoff in practice. The prefix length l determines the size of several internal structures of the BFT and how efficiently they can be stored. We selected $l = 9$ as this limits the internal fragmentation of the memory. The color set compression was applied regularly during the insertion process in order to keep the memory used to build the BFT as low as possible. After insertion of each dataset, the BFT was written to disk.

The SBT employs Jellyfish [15] to extract from each genome all valid k-mers. As the size of BFs used in the SBTs must be specified prior to the k-mer counting and should be the same for all vertices, the authors of the SBT suggested to estimate the number of unique k-mers in each dataset to design the size of BFs, at the price of an extra computation time (personal communication). Since we knew the exact number of unique k-mers from the BFT construction, we used this instead: 93,202,452 k-mers for the real dataset, resulting in a BF size of 11.1 MB. However, our simulated dataset corresponds to a very well conserved species with an average of 4,557,245 unique 54-mers per genome for a total of 5,121,443 unique 54-mers in the pan-genome: Each BF of the SBT would hold a very high false positive ratio, 59% on average, by choosing 5,121,443 bits for the BFs size. To avoid saturation, we computed a BF size of 24,910,142 bits (2.97 MB) for the simulated dataset to obtain a smaller false positive ratio of approximately 7.2% – similar to the ratio for the real dataset. We also reused k-mer counts computed for the BFT to estimate the number of hash functions: One hash function for the real dataset and four hash functions for the simulated dataset. The SBT counts the k-mers and builds the leaves in a one step process: It is not possible to differentiate these two sub-steps nor to extract the valid k-mers using a different software. According to the SBT paper and the CPU usage of this step, the insertion time is mainly dominated by the k-mer extraction. Note that SBTs are streamed on disk, each vertex being kept in a separate file. Running time and memory usage are shown in Table 1.

Table 1. Running time and memory usage for the real (*P. aeruginosa*) and simulated (*Y. pestis*) dataset. The compression ratio is given w.r.t. the original file sizes and (NA) indicates unavailable information.

	P. aeruginosa		*Y. pestis*	
	BFT	SBT	BFT	SBT
Insertion time	**14 h 34 min**	44 h 4 min	**11 min 29 s**	38 min 6 s
(without k-mer counting)	**(8 h 5 min)**	(NA)	**(2 min 18 s)**	(NA)
Uncompressed size	**7.25 GB**	11 GB	**79 MB**	115.2 MB
(compression ratio)	**(116:1)**	(77:1)	**(402:1)**	(276:1)
Compressed size	**2.2 GB**	4.8 GB	**76 MB**	117.2 MB
(compression ratio)	**(384:1)**	(176:1)	**(418:1)**	(271:1)

Suprisingly, the compressed version of the SBT for the simulated dataset takes more disk space than the uncompressed version. Memory usage during the insertion of the real dataset in the BFT is shown in Figure 2. Note that after storage on disk, the BFT can be compressed using a standard compressor. We compressed the BFT using 7z [2], resulting in a file of 980 MB for the real dataset and 40.1 MB for the simulated dataset (about 882:1 and 792:1 w.r.t. the original file sizes). We suspect that 7z delivers such compression ratio by taking advantage of the data redundancy among the uncompressed containers.

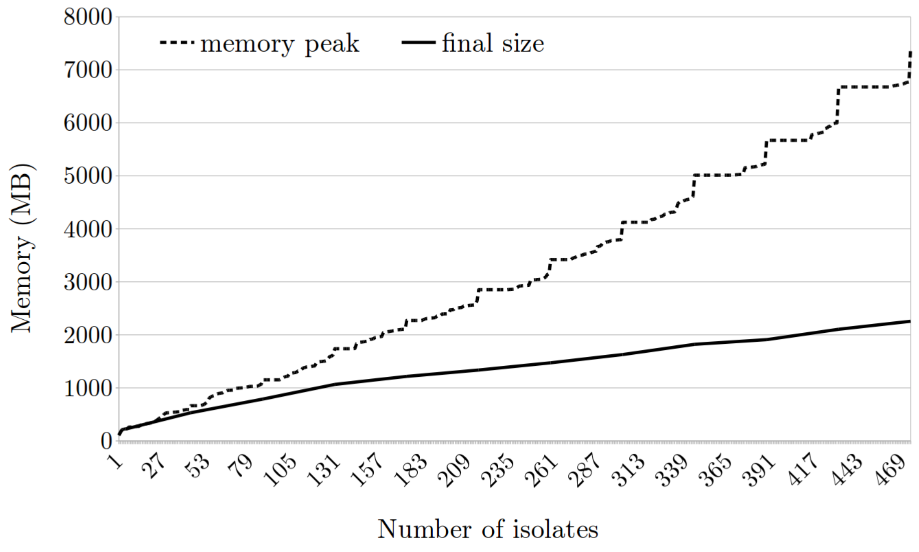

Fig. 2. Memory used by the BFT during the insertion of *P. aeruginosa* isolates.

For each dataset, the set of unique k-mers in the BFT was written to disk in random order and reused as a batch query for the presence of all unique k-mers in both data structures. It was not possible to query the SBT for a single batch query of all 93,202,452 63-mers for the real dataset as the memory used exceeded the 16 GB of memory available on the test machine, even when specifying that BFs could be loaded into memory separately. We suspect this is because k-mers are first loaded into memory before querying and the results are also stored in memory before writing to disk. Therefore, we divided the set of unique 63-mers into ten subsets, the first nine subsets containing 10,000,000 k-mers each and the last subset containing 3,202,452 k-mers. Query times are shown in Table 2.

[2] `http://www.7-zip.org/`

Table 2. Query time for the real (*P. aeruginosa*) and simulated (*Y. pestis*) dataset in total (and per k-mer). Real and simulated dataset batch queries contain 93,202,452 63-mers and 5,121,443 54-mers, respectively.

	P. aeruginosa	*Y. pestis*
BFT	**13 min (8.04 μs)**	**8 s (1.56 μs)**
SBT	9 h 18 min (359 μs)	3 min 47 s (44.35 μs)

A second experiment gives an estimation of the time required to traverse the graph represented by a BFT: It verifies for each k-mer whether its corresponding vertex in the graph is branching. This experiment first computes information about the root in a negligible amount of time and memory. Then, the BFT is queried for its branching vertices. For the real dataset, this experiment took 14 min (average time of 8.71 μs per 63-mer), resulting in 14,314,840 branching vertices. For the simulated dataset, this experiment took 14 s (average time of 2.73 μs per 54-mer), resulting in 6,312 branching vertices.

In summary, in our experiments the BFT was multiple times faster than the SBT on the building time while using about 1.5 times less memory. The BFT was about 30 times faster than the SBT for querying a k-mer.

7 Conclusion

We proposed a novel data structure called the Bloom Filter Trie for storing a pan-genome as a colored de-Bruijn graph. The trie stores k-mers and their colors. A new representation of vertices is proposed to compress and index shared substrings. It uses four basic data structures, which allow to quickly verify the presence of substrings. In the worst case, the compressed strings have a memory footprint close to their binary representation. However, we observe in practice substantial memory savings. Future work concerns the possiblity to compress non-branching paths that share the same colors [16], but also the extraction of the different pan-genome regions.

Acknowledgments. The authors wish to thank the anonymous reviewers and the authors of SBT for helpful comments. GH and RW are funded by the International DFG Research Training Group GRK 1906/1.

References

1. B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Comm. ACM*, 13(7):422–426, 1970.
2. M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. *Digital SRC Research Report 124*, 1994.

3. R. Chikhi and P. Medvedev. Informed and automated k-mer size selection for genome assembly. *Bioinformatics*, 30(1):31–37, 2014.
4. A. J. Cox, M. J. Bauer, T. Jakobi, and G. Rosone. Large-scale compression of genomic sequence databases with the Burrows–Wheeler transform. *Bioinformatics*, 28(11):1415–1419, 2012.
5. M. Crusoe, G. Edvenson, J. Fish, A. Howe, E. McDonald, J. Nahum, K. Nanlohy, H. Ortiz-Zuazaga, J. Pell, J. Simpson, C. Scott, R. R. Srinivasan, Q. Zhang, and C. T. Brown. The khmer software package: enabling efficient sequence analysis. 04 2014.
6. S. Deorowicz, M. Kokot, S. Grabowski, and A. Debudaj-Grabysz. KMC 2: Fast and resource-frugal k-mer counting. *Bioinformatics*, 31(10):1569–1576, 2015.
7. M. A. DePristo, E. Banks, R. Poplin, K. V. Garimella, J. R. Maguire, C. Hartl, A. A. Philippakis, G. del Angel, M. A. Rivas, M. Hanna, et al. A framework for variation discovery and genotyping using next-generation DNA sequencing data. *Nat. Genet.*, 43(5):491–498, 2011.
8. C. Ernst and S. Rahmann. PanCake: A Data Structure for Pangenomes. *Proc. of the German Conference on Bioinformatics 2013*, 34:35–45, 2013.
9. P. Ferragina and G. Manzini. An experimental study of an opportunistic index. *Proc. of the 12th ACM-SIAM Symposium on Discrete Algorithms*, 1:269–278, 2001.
10. E. Fredking. Trie Memory. *Comm. ACM*, 3(9):490–499, 1960.
11. S. Heinz, J. Zobel, and H. E. Williams. Burst tries: a fast, efficient data structure for string keys. *ACM Trans. Inf. Syst.*, 20(2):192–223, 2002.
12. L. Huang, V. Popic, and S. Batzoglou. Short read alignment with populations of genomes. *Bioinformatics*, 29(13):i361–i370, 2013.
13. Z. Iqbal, M. Caccamo, I. Turner, P. Flicek, and G. McVean. De novo assembly and genotyping of variants using colored de Bruijn graphs. *Nat. Genet.*, 44(2):226–232, 2012.
14. H. Li and R. Durbin. Fast and accurate short read alignment with Burrows-Wheeler transform. *Bioinformatics*, 25(14):17541760, 2009.
15. G. Marçais and C. Kingsford. A fast, lock-free approach for efficient parallel counting of occurrences of k-mers. *Bioinformatics*, 27(6):764–770, 2011.
16. S. Marcus, H. Lee, and M. C. Schatz. SplitMEM: a graphical algorithm for pangenome analysis with suffix skips. *Bioinformatics*, 30(24):3476–3483, 2014.
17. E. W. Myers. The fragment assembly string graph. *Bioinformatics*, 21:ii79–ii85, 2005.
18. N. Nguyen, G. Hickey, D. R. Zerbino, B. Raney, D. Earl, J. Armstrong, D. Haussler, and B. Paten. Building a pangenome reference for a population. *J. Comput. Biol.*, 22(5):387–401, 2015.
19. B. Paten, M. Diekhans, D. Earl, J. St. John, J. Ma, B. Suh, and D. Haussler. Cactus graphs for genome comparisons. *J. Comput. Biol.*, 18(3):469–481, 2011.
20. J. Pell, A. Hintze, R. Canino-Koning, A. Howe, J. M. Tiedje, and C. T. Brown. Scaling metagenome sequence assembly with probabilistic de bruijn graphs. *Proc. Natl. Acad. Sci. U.S.A.*, 109(33):13272–13277, 2012.
21. B. Solomon and C. Kingsford. Large-Scale Search of Transcriptomic Read Sets with Sequence Bloom Trees. *bioRxiv*, 017087, 2015.
22. S. Wandelt, J. Starlinger, M. Bux, and U. Leser. RCSI: Scalable similarity search in thousand(s) of genomes. *Proc. of the VLDB Endowment*, 6(13):1534–1545, 2013.