Anuved Verma

MPCS56420

**Bloom Filter Trie: an alignment-free and reference-free data structure for pan-genome storage**

**Introduction:** As sequencing technology improves, our genomic datasets are now larger than ever before. To extract valuable information from this data, it is crucial to be able to store and query it efficiently. This is especially true of pan-genomes, which are a redundant collection of similar sequencing reads, often extracted from members of the same species. Since most of the base pairs in these sequences are identical, pan-genomes are a prime candidate for compression. One method, proposed by Holley et al [1] uses a data structure called the Bloom Filter Trie (BFT) to compress, store, and query pan-genomes. The BFT essentially represents and compresses a Colored De Bruijn Graph (C-DBG), using Bloom Filters and Burst Tries to store k-mers from pan-genomic sequences. The authors of the paper primarily compared the BFT with the Sequence Bloom Tree (SBT), and found that the BFT performed better in query times, insertion times, and data compression. In this paper, I will discuss the details of this data structure, as well as my own experiences and results after implementing it.

**Background:** The BFT utilizes key ideas from three distinct data structures: Colored De Bruijn graphs, Bloom Filters, and Burst Tries.

C-DBG: A common way to represent and study genomic data is to use Colored De Bruijn Graphs. These graphs are composed of k-mers extracted from sequencing data, and combined to form a directed graph that can be used for genome reconstruction, mutation analysis, and more. Each vertex in a C-DBG is a (k-1)-mer, with a directed edge to all other vertices that share their first (k-2) characters with the last (k-2) characters of the vertex pointing to it. That is, a directed edge from vertex **v** to **v’** exists if **x[1…k-1] = x’[2…k]**. Furthermore, each vertex has a “color” associated with it, which represents a set of genomes that contain a k-mer. Colors are represented as bit-arrays, with each index of the array corresponding to a different genome (a bit is set to 1 if the genome corresponding that bit contains that k-mer).

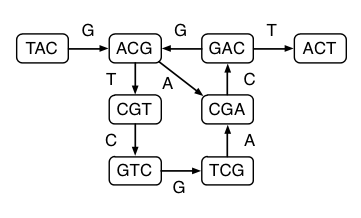


Figure : Example of a C-DBG (without colors) [2]

Althrough C-DBGs have numerous applications, they are very expensive to store in memory, and therefore must be compressed before they can be useful for practical applications. The BFT attempts to solve this problem by compressing k-mers using techniques which will be discussed in a later section.

Bloom Filter: The Bloom Filter is a data structure, represented as a constant-size bit-array, which allows us to test whether or not an element has been seen before. An incoming item goes through a hash function ***f***, which maps to one of the bits in the bit-array, which is then set to 1. If the same item appears again, it would be hashed to the same bit; however, we would see that the bit has already been set to 1 which indicates that we have already seen this item. Note that it is possible, by random chance, that a never-before-seen item gets hashed to a bit that was already set to 1. However, if an item is hashed to a bit that is 0, we can conclude with complete confidence that we have never seen this item before. In other words, Bloom Filters cannot produce false negatives, but do have a small probability of producing false positives. The probability of a false positive can be reduced by using multiple hash functions such that an incoming item must land on a 1 for *every* hash function before it can be considered a “seen” item. Bloom Filters are used in BFTs to optimize its querying algorithms, which will be discussed in a later section.

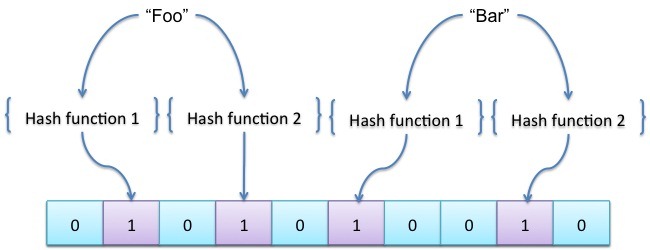


Figure : Example of a Bloom Filter with 2 hash functions.

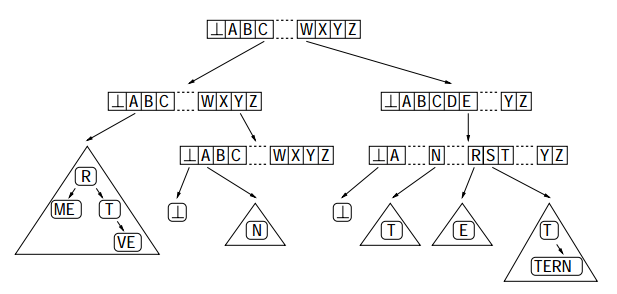
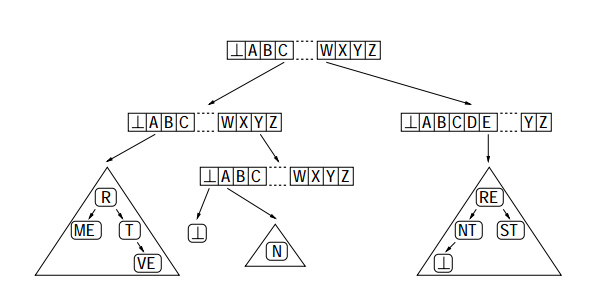
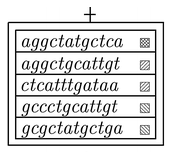
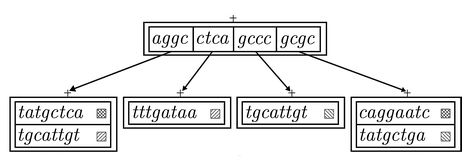
Burst-Trie: A Burst Trie is an efficient data structure to store strings, and is the “back-bone” of the BFT. The Burst Trie is essentially a tree composed of containers that are either compressed or uncompressed. As strings are initially inserted into the Burst-Trie, they are stored in an uncompressed container. After a certain threshold, this uncompressed container “bursts” into a compressed container, which only stores the first letter of the strings, as well as pointers to child nodes that contain the rest of the strings. The threshold at which a container bursts is empirically determined for the specific application. 

Figure : Bursting of a Burst-Trie after insertion of the word "WESTERN". Elements of the compressed container point to uncompressed child containers.

**Bloom Filter Trie:** The BFT utilizes elements from the three data structures discussed above to compress and query k-mers of a pan-genomic dataset. In this section, I will discuss my implementation of the BFT, which followed as closely as possible to the implementation described in Holley et al.

Structure: The BFT is composed of vertices. Each vertex holds one uncompressed container (can be empty), as well as zero to many compressed containers. When k-mers are first read from file, they are stored in “tuples”. A tuple holds the k-mer and a color (a bit-array with a single bit set to 1, representing the genome it this k-mer was just read from). Tuples are stored alphabetically in uncompressed containers of a vertex until it reaches its threshold capacity ***c*** (some tune-able parameter). At this point, the uncompressed container is converted (“burst”) into a compressed container. A compressed container has four components: a bloom filter *quer*, a prefix bit-array *pref*, a suffix bit-array *suf*, and a cluster bit-array *clust*. We will take a closer look at the components of the compressed container in the following section.



*gcgccaggaatc*

Figure 4: Bursting of uncompressed container into a compressed container within a vertex of the BFT.

Compressed Container: A compressed container at the root (*d=0)* of our BFT is responsible for storing the first L characters (prefix) of K-mers (where K divides L evenly). To be stored efficiently, the prefix is further divided into a prefix-prefix of length ***a***, and a prefix-suffix of length ***b*** where ***a + b = L***. Our *pref* bit-array has a single index for every possible value of the prefix-prefix alphabetically. For example, in Fig. 5, the first index corresponds to ‘aa’, the second to ‘ac’, third to ‘ag’, and so on. The prefix-prefixes from Fig. 4 are ‘ag’, ‘ct’, ‘gc’, and ‘gc’, and we see that the corresponding bits in *pref* are set to 1. The prefix-suffixes are stored in *suf*, along with some cluster data *clust* which helps to optimize querying efficiency. Importantly, along with the prefix-suffix, each element of the *suf* array holds a pointer to a child vertex. The child vertices at *d=1* store the remaining **K – L** characters of the original K-mer tuple, along with its color data, in its uncompressed container. This process repeats recursively as we add more and more k-mers to our BFT, with each leaf node holder a shorter and shorter tuple of length **K-dL** where **d** is the depth of the BFT. Furthermore, any time a prefix is added to a compressed container via bursting, that prefix is also hashed into *quer,* which helps optimize query times.

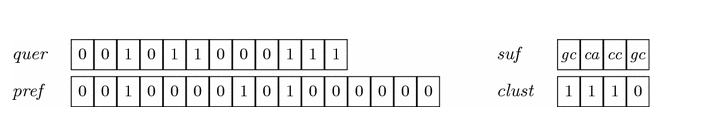


Figure : Internal representation of a compressed container.

Querying: Much of the added complexity in the structure of the BFT plays a role in optimizing queries. The general algorithm to query a BFT for a given k-mer is as follows, starting at the root vertex:

1. Check the uncompressed container of the vertex for the k-mer. This can be done in O(lg n) time, since tuples are stored alphabetically in uncompressed containers.
2. If not found, iterate through the vertex’s compressed containers and do the following:
   1. Check if the prefix of our query k-mer exists in the Bloom Filter *quer* in O(1) time. If the prefix does not exist, we terminate the algorithm and our k-mer is not present in the BFT.
   2. If the k-mer prefix passes the Bloom Filter test, verify that it’s not a false positive by comparing the prefix-prefix and prefix-suffix of the query with the values in *pref* and *suf*.
   3. If step (b) was a false positive, return false as our k-mer is not present in the BFT. Otherwise, repeat step (1) with the child vertex of the *suf* element that matched with our query prefix.

Insertion into the BFT is almost identical to querying. The main differences are that if a k-mer is found, we simply update the color of its tuple. If it’s not found, we either insert it into the uncompressed container if there is room, or we insert it into the compressed container and update *pref, suf,* and *clust* accordingly.

**Results:** After implementing and testing the algorithm locally in Java, it’s clear that the algorithm is not ideal for shorter datasets. I tested the BFT against basic HashMap and ArrayList data structures, using a small pan-genome dataset of 80kB.

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Time to Populate** | **Size Estimate (JVM)** | **Query Time** |
| **BFT** | 33039 milliseconds | 16873048 | 0 milliseconds |
| **HashMap** | 90 milliseconds | 10886744 | 0 milliseconds |
| **ArrayList** | 44 milliseconds | 13811464 | 2 milliseconds |

Figure : Data of BFT performance against basic HashMap and ArrayList data structures on small 80kB dataset.

**Discussion:** Although the above results do not look particularly favorable for the BFT, it’s important to note that the data structure is not optimized for small datasets. It is very likely that the BFT will begin to outperform the basic data structures on larger datasets. Furthermore, there are numerous parameters for the BFT (ie. k-mer length, prefix length, container capacity, etc.) that can be adjusted to improve performance for a given dataset. Ultimately, more testing will need to be done to determine whether or not BFT can be used for practical applications. One potential future direction to take this might be to adapt the algorithm to allow storage and querying of all characters in the English language.

**References:**

[1] Holley G, Wittler R, Stoye J. Bloom Filter Trie: an alignment-free and reference-free data structure for pan-genome storage. Algorithms Mol Biol. 2016;11:3.

[2] "Succinct De Bruijn Graphs - Alex Bowe." Alex Bowe. N.p., n.d. Web. 30 Nov. 2016.

[3] <http://goanna.cs.rmit.edu.au/~jz/fulltext/acmtois02.pdf>

[4] Compeau PE, Pevzner PA, Tesler G. How to apply de Bruijn graphs to genome assembly. Nat Biotechnol. 2011;29(11):987-91.