Implementing and Evaluating Cuckoo Tree on Searching Read Sets

EN.601.647 - Final Project

Omar Ahmed, Andrew Rojas, Erfan Sharafzadeh, Kathleen Newcomer

Contents

1 Abstract

The amount of sequencing information in public databases is growing at a rapid rate which requires intelligent approaches to search it efficiently. Previous techniques used alignment-based approaches which can be expensive in computational resources, however one of the recent promising advances was the sketch-based approach of Sequence Bloom Trees [12]. In this project, we present Cuckoo Trees which use a similar approach of using a binary tree with sketch data structures, cuckoo filters, in each of the nodes to help do a rapid search of the database of read sets. In this paper, we show our cuckoo filter implemented with Python is able perform similarly to a bloom filter implementation in terms of memory usage. Unfortunately, we discovered our Cuckoo Tree construction was significantly slower in terms of time and worse in terms of memory than our Sequence Bloom Tree construction. We present reasons for this issue along with potential paths to improve the short-comings. The repository containing the sources and evaluation scripts is publicly available at github.com/amrojas/comp_genom_final_project.

2 Introduction

The amount of sequencing data being produced has out-paced the improvement in computer speeds in recent years without question. The well-known Moore's Law expects that computer speeds to double around every 24 months [9], however the amount of sequencing appears to increase by 5-fold every year [10]. Therefore, scientific questions that require searching large databases of sequencing data will require an efficient approach to be able to complete the query in an efficient and reliable time-frame.

An example of a specific biological question that can be answered by searching an entire sequencing database is rare-isoform detection. There may be a particular isoform of a gene of interest that a researcher wants to study further. Allowing the researcher to search the entire database for studies that contain this rare isoform can give the researcher an insight into better understanding the function of this rare isoform based on the experiments that contain it.

The NIH Sequence Read Archive (SRA) is a well-known database that stores sequencing reads, and it contains about 3 petabases of sequencing as of 2016[8]. Large databases such as the SRA open up a vast array of opportunities for individual researchers, however the question remains as to how to efficiently tackle the problem of searching the entire database. There are various approaches that can be used to search the SRA, however one of the very promising approaches and the one that inspired this project is using Sequence Bloom Trees (SBT) [12]. The basic premise of SBTs is that in the leaves of the tree represent sequencing experiments and the non-leaf nodes contain Bloom Filters loaded with all the kmers in that node's subtree. This tree structure allows the user to efficiently query all the sequencing experiments loaded into the tree for an certain sequence, for example an isoform of a certain gene of interest. This approach has shown to be superior to other approaches of searching the SRA [12], however there are certain limitations of their approach that has motivated our project.

One of the limitations of SBTs is associated with the bloom filters, which is the data-structure used to help traverse the tree. Standard bloom filters do not have the ability to remove items without having to rebuild the entire filter. In the context of sequencing datasets, this can be quite expensive if there specific experiments you want to focus on and therefore want to take an existing SBT and trim it down. The bloom filters would require the user to have to rebuild many bloom filters which could be computationally expensive. Another area for potential improvement has to do with the memory usage of the SBTs. It has been shown that a SBT built on 2652 RNA-seq experiments was able to be stored using only 2.3% of the original size of the the total sequencing experiment [12]. This significant reduction in space shows the power of using sketch data-structures such as bloom filters. However, there are various other data structures namely cuckoo filters that also have the ability to approximately answer set-membership questions.

Cuckoo filters were introduced in 2014, and were shown to be more memory-efficient than bloom filters in practical situations where the target false positive rate ε is less than 3% [4]. The nice aspect of cuckoo filters is that they be more memory efficient than bloom filters with added functionality of being able to delete items without having to rebuild the entire filter.

In this project, we introduce Cuckoo Trees, a data-structure that aims to allow for efficient queries on large databases of sequencing datasets for a target sequence. The concept of the Cuckoo Tree is similar to SBTs in the sense that the leaves of the tree correspond to sequencing datasets, and the nodes contain cuckoo filters constructed with the kmers of that subtree.

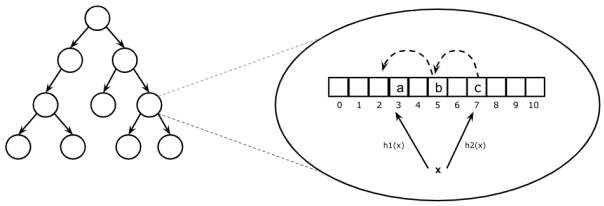


Figure 1: Diagram of cuckoo tree and the cuckoo filter inside of a node of the tree

The main advantage of our approach can be thought of along the same lines of the comparison between bloom filters and cuckoo filters. The improved memory-efficiency of cuckoo filters and their ability to dynamically delete items gives our cuckoo trees an edge over SBTs.

3 Prior Work

Our work builds on a history of prior work which is thoroughly detailed here [7]. In this section we will give an overview of some of the more well known algorithms and data structures that have been developed, some of which are still in use today.

3.1 Bloom and Cuckoo filters

The cuckoo filter and bloom filter both allow us to tackle the problem of approximate set-membership in an efficient way by introducing a one-sided error, in the form of false-positives. The bloom filter is widely used in databases, caches, and networking [6] where approximate set membership allows us to avoid explicitly checking for membership in an extremely large set that may prove costly [1]. In [4], it's seen that in practical applications, the cuckoo filter offers a variety of advantages over the bloom filter for this problem, providing removal of item dynamically, improved query performance when near full, and typically uses less space.

3.2 Smith-Waterman

Smith-Waterman uses a dynamic programming approach. At any given base comparison the algorithm looks at all ways it could get from the previous base to this base, and keeps track of a scoring mechanism. In effect this means seeing which choice is best for this base: match, substitution, insertion, deletion, or resetting the score to zero. The score for each move is determined from a substitution matrix and a gap-penalty scheme. Finally a traceback is run to find the alignment [11].

Using this approach offers a guarantee to find the local optimal alignment, but comes with a high computational cost. Improvements have been made, but the basic algorithm has a quadratic run time, as well as a quadratic space requirement. Obviously this is not ideal for handling large datasets. This algorithm is not in wide scale use today.

3.3 FASTA

The original FASTA program was designed to align amino acid sequences and was extended in 1987 to include DNA searches. It implements some heuristic searching and then locally applies Smith-Waterman only when looking at potential match sites. Essentially, it creates a kmer index to identify potential match sites, and a more intensive search at each site [2].

This approach offers a trade-off between time and sensitivity. Using a smaller length kmer offers a more sensitive search, but increases the amount of time that search takes. The FASTA algorithm achieved sensitivity similar to that of

Smith-Waterman, but took much less time [7].

The legacy of this program is by and large the FASTA file format that it created, and not the algorithm.

3.4 BLAST & STAR

Basic local alignment search tool (BLAST) operates in a very similar manner to FASTA, but offers a few noted improvements [13]. BLAST removes low-complexity regions before aligning the sequences. Highly repetitious regions prove difficult for most any alignment tool and by removing them BLAST is able to achieve faster results. FASTA offers more opportunity to create a fine-tuned search than BLAST does.

BLAST has been implemented as a web interface with the National Center for Biotechnology Information. Due to this ease of access and relative speed, BLAST is used by many researchers without a computational background to perform alignment searches.

STAR is also a widely used read-alignment program [5]. It is often the main program used for alignment of RNA-seq reads in analysis pipelines. However, in the context of large databases such as SRA, an alignment-based approach for querying sequences can be computationally quite expensive. These large computational costs of alignment-based approaches is what helps encourage sketch-based approaches such as Sequence Bloom Trees discussed in the Introduction [4].

4 Design and Implementation

This section details how different data-structures were designed with respect to each other, and the relevant implementation details. All of our data-structures were implemented in Python.

4.1 Bucket

One of the first data-structures that we implemented was a bucket class. This was a basic data-structure that represents a bucket of items, more specifically a bucket of fingerprints. This bucket was implemented as a Python list. Cuckoo filters were defined to store fingerprints which are bit-strings obtained from a hash of the input item [4]. In practice, cuckoo filters are used with more than one entry per bucket. Therefore, implementing buckets as a separate object helps to obfuscate the details of the bucket and allows for further optimizations such as the idea of semi-sorting the buckets for space efficiency [4].

However, one of the major downsides of the object-oriented approach of using a Bucket object for every entry of the cuckoo filter is all the added overhead in Python with each object. In addition, since Python stores binary literals as integers, it increases the memory usage with each additional fingerprint more than it needs to be. Theoretically, with each additional item, the cuckoo filter should only store a fingerprint the size of the fingerprint size in bits. However with this approach, each additional input item adds the size of an integer to the data-structure, which is 24 bytes since an integer is considered an object in Python so it has additional overhead.

The downsides of using a Bucket object for every entry of the cuckoo filter is what inspired an alternate approach called the bitBucketArray. This data-structure is implemented using a Python package called bitarray which is implemented as an array of bits with C under the hood. This approach was much more space efficient in comparison to using Bucket objects since it avoids storing all the fingerprints as integers. In addition, this object itself is meant to represent an array of buckets which avoids the added overhead by having an object for each additional entry of the cuckoo filter.

Figure 2 shows how the two previously discussed bucket classes, bucket and bitBucketArray, are associated with the cuckoo filters which will be discussed in the following subsection.

4.2 Cuckoo Filter

As previously described, the basic cuckoo filter was implemented as a list of Bucket objects. The key parameters for a cuckoo filter are the number of

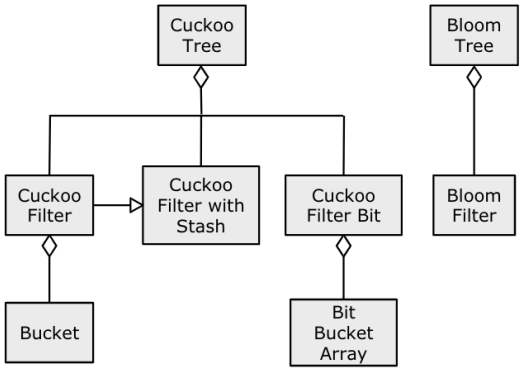


Figure 2: Class diagram for this project

buckets, number of entries per bucket, and the size of the fingerprints. The basic implementation for each of the key methods such as insert, contains, and remove were implemented following the algorithms shown in the original cuckoo filter paper [4]. Figure 3 shows an example of an insertion into a cuckoo filter where you use two hash functions to get two potential locations. In this case, since the indices in the filter calculated by the two hash functions are already full, the method will proceed to try to move fingerprints around. The insertion method is implemented to continue to move fingerprints around for a large pre-specified number of iterations which at that point we assume we are in an infinite loop and that insertion is considered to have failed. The specific number of iterations used is 500 which is the same number used in the cuckoo filter paper [4].

In addition to the basic cuckoo filter, there were two other variants of the cuckoo filter that were implemented. The first variant was called CuckooFilterStash which is the basic cuckoo filter with a stash. A stash is a small list of fingerprints

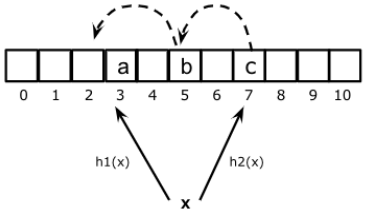


Figure 3: Schematic of inserting an item into a cuckoo filter

which correspond to items that were not able to be inserted into the cuckoo filter. This added stash is theoretically meant to reduce the probability of a failed insertion [3]. The second variant was called CuckooFilterBit which is a cuckoo filter implemented using the bitBucketArray opposed to using a list of Bucket objects. As previously discussed in Section 4.1, the memory efficiency of the bitBucketArray in comparison to Bucket objects is what makes the CuckooFilterBit the most space-optimized cuckoo filter among the three variants discussed in this section.

4.3 Bloom Filter

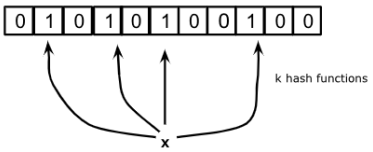


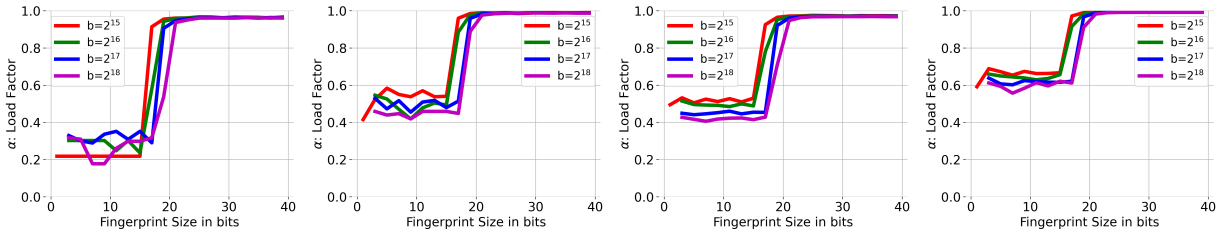
Figure 4: Schematic of inserting an item into a bloom filter

In this project, we decided to implement a bloom filter as well in order to be able to compare Cuckoo Trees to Bloom Trees. Since our Cuckoo Tree implementation is in Python, we made the decision to compare it to an implementation to Bloom Trees in Python as well to make the comparison more focused on the data-structure opposed to the languages.

The bloom filter class was implemented using the Python packages, bitarray and mmh3. As the Figure 4 shows, given a new item to insert into the filter, it is hashed using k hash functions and those k generated positions are turned on in the filter. Specifically, the mmh3 hash function is used with k different seed values to help figure out the k indices with the bitarray to turn on to True.

4.4 Sequence Bloom Tree

Before introducing the Cuckoo Tree's implementation we will quickly review the SBT and talk about



(a) BucketSize = 4 No Stash (b) BucketSize = 8 No Stash (c) BucketSize = 4 Stashed (d) BucketSize = 8 Stashed

Figure 5: Evaluating the effect of different hyper parameters on the load factor of the Cuckoo filter. The first two figures demonstrate the filter without stash and the other two figures show the results for stashed Cuckoo filter. With larger fingerprints, the impact of bucket count and stash fades, since the load factor approaches 1. With smaller fingerprints, stash can considerably improve the load factor. Additionally the larger buckets improve the load factor by up to 20%

our implementation quickly. More details are of course found in [12].

The SBT is a binary tree which associates leaves with sequencing experiments. Each node in the tree contains a bloom filter, and this bloom filter represents the kmers present in the subtree rooted in that node. So for a leaf, the filter contains only the kmers from that experiment, whereas for any internal nodes, the filter contains kmers from any leaves underneath that node.

Our SBT was made with two classes, a Node class which we use only to house parent/child relationships and its filter, and the BloomTree class, which is responsible for maintaining the root to our tree and traversing the tree for insertion or querying. Our insertion method takes in a list of read objects that represent a sequencing experiment (we assume they all belong to a single file for simplicity), and our query method takes in a query string for which we want to find similar sequencing experiments, and returns a list of sequencing experiments ids (filenames in our case).

The tree exposes all of the parameters of the bloom filter, as well as θ, which decides how many kmers should be present from the query string in any sequencing experiments we return.

4.5 Cuckoo Tree

When moving from the SBT to Cuckoo Tree, we keep many of the important ideas. Leaves are still experiments and nodes still contain the kmer information of the leaves under its subtree, but now we insert into a cuckoo filter located at each node instead of bloom filter.

There are a few important distinctions. First, dealing with duplicate entries in the filter. A bloom filter has no concept of duplicate entries because the 3-mer "ABC" would hash to the same values every time and set the bits corresponding to it equal to 1. In that sense, there is no way to tell "how many" occurrences of a kmer exist in the filter. However in the cuckoo filter, we naturally allow duplicates. For simplicity we decided to instead remove the possibility of duplicate fingerprints existing in the cuckoo filter. One reason to consider this is space. For many occurrences of the same kmer, it could store that same kmer over and over, which obviously could have memory drawbacks. Now the filter behaves more similarly to a bloom filter, in that the fingerprint exists or it does not, and there is no notion of how often it occurs. Obviously the major drawback is it is now not possible to remove from the filter (how could you distinguish two kmers that may have the same fingerprint), and we relegate this whole idea to future work.

Also, the insertion of the kmer information of a new leaf into internal nodes during insertion differs. One of the nicest parts of the bloom filter to understand was that inserting kmer information is simple, a bitwise-OR between the bitarrays of two filters gives a bloom filter with the kmer information of the two filters. However with a cuckoo tree this is not as simple, and we are stuck instead with insert-

ing each kmer from the dataset to be inserted into the cuckoo filter at each node. It may be possible to come up with a better rebuilding strategy using the two already built cuckoo trees, but one thing to take into account is you can't just copy information from the same buckets into the new filter, because then the bucket may overflow and items would need to be reinserted elsewhere. This is an area that could be better explored with more time.

5 Evaluation

This section is dedicated to reporting some performance results on our implementation of Bloom filters, three different variants of Cuckoo filters, and tree structures containing Bloom and cuckoo filters as originally suggested by [12]. We first describe the experiment methodology and the testbed configuration. We describe the datasets used in the evaluation next, followed by reports on load factor analysis, space efficiency analysis, construction and query performance, support for item deletion, and finally, a dedicated report on the performance of the Cuckoo and Bloom trees.

5.1 Methodology

While the correctness of the implementations were verified by general test cases shipped alongside the source code, we needed an automated process to evaluate the data structures under load. We augmented our main.py driver script with a facility that takes all supported hyper-parameters for the data structures as command-line arguments and wrote bash scripts that create and evaluate the data structures in different ways. The output of these scripts are in CSV format and can be simply input to *numpy* and *matplotlib* to process the results and create figures.

We report results that are generated by running experiments in a commodity PC featuring one Intel Core i7-8700 12-core CPU with hyperthreads enabled. The cores can operate at a maximum of 3.20GHz. We additionally enabled the performance CPU frequency governor in the testbed to ensure maximum frequency settings that can be boosted to 4.2GHz using the *Turboboost* technology. This is specially important since we didn't design our data structures to be thread safe, therefore used a single core to populate and query them all all below scenarios.

5.2 Dataset

We were initially planning to make use of the original datasets used in [12]. However, since getting access to the data set requires some proprietary command line tool, we decided to build our own dataset generator that can create arbitrary sized synthetic FASTQ files. We leverage a synthetic FASTQ dataset with 10 million reads of length 100 bases in all the experiments. Using another script, we ensured that the dataset does not contain repeated reads. While this might not be the case for Genomics data sets, it tries to simulate the worst case scenario for all the data structures under evaluation. The FASTQ generator can be found in the *util/* directory of the repository, however, due to its large size we can only share the original dataset we used using external storage tools upon request.

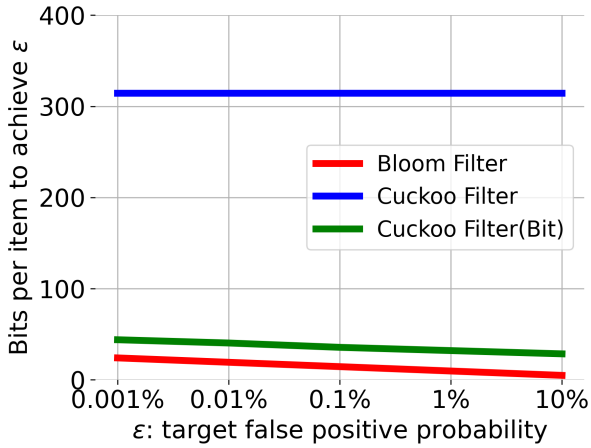


Figure 6: Bits-per-item (BPI) for naive Cuckoo filter, efficient implementation of Cuckoo filter (bitBucketArray), and Bloom filters. All filters have a similar declining rate as we increase ε, however, since bit-array implementations are far more efficient, the false positive rate can present itself more in the BPI performance.

Data structure	# items	Constr. speed (IPS)	Load Factor	Bits per Item	Achieved FP rate
Naive Cuckoo filter	361,608	106,341	0.996	265.17	0.000
Bit-array Cuckoo filter	2,846,388	19,929	0.995	33.24	0.001
Stashed Cuckoo filter	361,915	93,574	0.996	264.94	0.009
Bloom filter	10,000,000	511,674	1	9.58	0.010

Table 1: Construction speed (items per second), item capacity, load factor, BPI, and FP rate comparison of data structures limited to 11 megabytes of memory. Bloom filter is highly space efficient with considerably smaller insertion time per item than all the cuckoo filter implementations. The bit-array implementation of our cuckoo filter is the closest alternative to the Bloom filter.

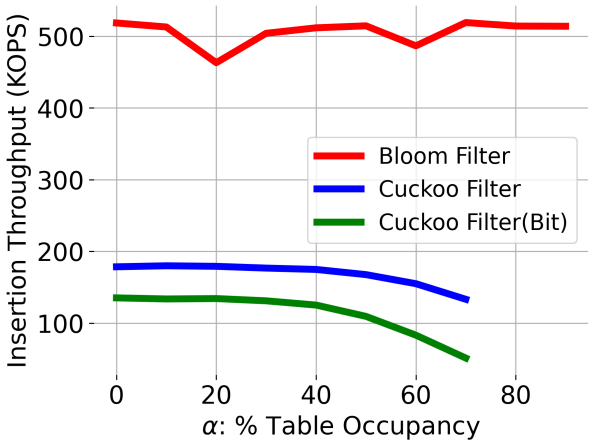


Figure 7: Measuring the insertion throughput of the filters as they become more loaded. As the bitfield become more ones, the insertion throughput is considerably reduced. However, since naive Cuckoo filter uses Buckets and lists, it has a fixed insertion throughput.

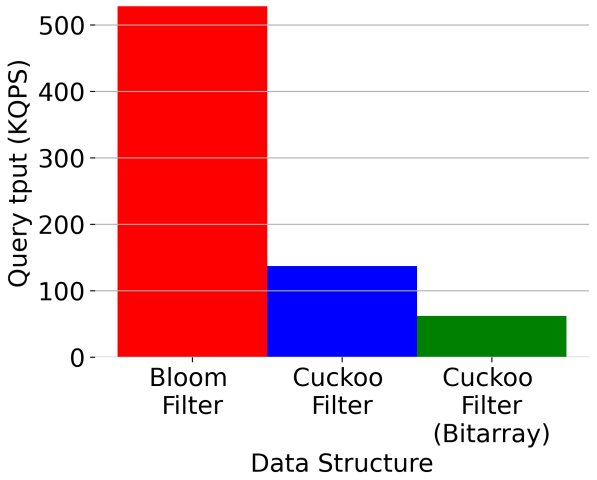


Figure 8: Measuring the query throughput of the filters when they are fully loaded. The space efficient implementation of Cuckoofilter (Bit) comes at the cost of around 2× slower queries on the filter. The bloom filter doesn't require any time consuming item search, therefore achieves near 0.5M queries per second throughput.

5.3 Load Factor Optimization

Our first evaluation scenario is a parameter sensitivity analysis mixed with load factor analysis of the base Cuckoo filter data structure. We use full-sized reads and try fingerprints of size 1 to 40 bits (shown in the X-axis of figure 5). We evaluate bucket counts of $2^{15}, 2^{16}, 2^{17}, 2^{18}$ shown in red, green, blue and magenta in figure 5 respectively. Finally, we try bucket sizes of 4 and 8 in two settings: 1. The naive implementation of Cuckoo filter (figures 5a and 5b), and 2. The cuckoo filter with a fixed-sized stash with a capacity of 50 items (figures 5d and 5d).

With smaller finger print size (< 20), we can see that a larger bucket size can improve the load factor by up to 20%. Additionally, we can improve the load factor by another 20% if we add a stash to our filter. However, for fingerprints larger than 20, the load factor quickly rises to over 0.96 for all parameter settings allowing us to easily find some near optimal inputs for the rest of the experiments.

5.4 Bit-per-item Comparison

Space efficiency is our next evaluation target. We compare: 1. The naive Cuckoo filter implementation, 2. The space efficient Cuckoo filter implementation using bit arrays, 3. The Bloom filter with regards to their bit-per-item (BPI) require-

ments. We derive the input parameters by fixing the expected items to 10M and modifying the expected False-Positive (FP) probability, denoted by ε, from 0.001% to 10%. Figure 6 presents the results.

The naive Cuckoo filter (Blue) uses Python classes for the implementation of Buckets. That results in huge space inefficiency. Looking at these results, we decided to implement a more efficient alternative using bit-arrays (bitBucketArray), achieving around tenfold improvement over the naive version (Green). The BPI of the Bloom filter (Red) is the best among all implementations, however it achieves this efficiency at the cost of losing the support for item deletion and lower false-positive rate.

5.5 Space Efficiency and Construction Speed

Table 1 compares the three implementations of Cuckoo filter with Bloom filter regarding the number of items they can fit in a fixed memory allocation, construction speed, load factor and their respective false positive rate. We read our large synthetic dataset into the data structure until each of them reaches 11MB in size. Then we measure and report the above statistics.

While the bloom filter could fit all the dataset into less that 11 megabytes, we could only achieve near 3M insertions using our efficient cuckoo filter. Using the stash did only have a negligible effect the the capacity of the naive cuckoo filter. We could achieve high load factors and very good false positive rates for all the filters due to automated parameter selection using a built-in function.

5.6 Insertion Throughput

Next, we measure the insertion throughput of the three filters in figure 7. We measure the insertion throughput in ten time windows until the filter becomes full. The x-axis shows the table occupancy from the scale of 0 to 10 where 10 means the filter is 100% loaded. Bit-arrays have this disadvantage that cause slowness when they become populated. This presents itself in the insertion throughput of both bit-array Cuckoo filter and Bloom filter. However, the naive Cuckoo filter is implemented using Bucket objects and python lists, therefore it always keeps a constant insertion throughput that is 2-3$\times$ better than its competitors. Another thing notice is that bit-array implementations cannot fully fill the data structure compared to the naive cuckoo filter.

5.7 Query Throughput

We repeat the throughput measurement experiments, this time for the queries to the filters. Figure 8 presents the average query throughput for three data structures filled with 10 million reads of size 100 bases. The results are totally aligned with the insertion query throughput results presented in figure 7. The bit-array implementation of the Cuckoo filter imposes a heavy tradeoff between space efficiency and performance while the Bloom filter only requires a few hash lookups for querying.

5.8 Evaluating the Bloom and Cuckoo Trees

The main use case for the tree data structures is to help find datasets that contain a specific read or a k-mer. Therefore, to build the trees we take a different approach than previous tests. We create 4 synthetic datasets, each containing 10K reads of length 100 bases. We input each data set separately to the trees, so tat each node in the tree represents a sketch for that dataset. Then using the *query* API, we are able to enquire which datasets may contain a specific k-mer. We set k to 80 for the Cuckoo-bit tree and 20 fro the Sequence Bloom tree, we also false positive probability to 0.01 and the expected number of items to 40000.

It takes around 74 minutes to create the cuckoo-bit-tree from four datasets. When built, the tree achieves bits-per-item ratio of 2891. Conversely, the Sequence Bloom tree is built in 5 seconds and takes only 1300 bits per item. Our initial query experiments show that both sketches can correctly report the datasets that contain user-specified k-mers. Nevertheless, the slow construction and high memory usage of the Cuckoo tree can be attributed to the use of Python objects and checking for potential duplicates before adding each k-mer. It is worthwhile to point that duplicate detection was disabled in previous sets of experiments, because this feature is only required for tree structures to ensure that a

k-mer is not repeated in the tree with an acceptable probability.

6 Conclusion

In this project, we studied two variants of sketching data structures for fast and efficient indexing of synthetic datasets. We first implemented Bloom filters and three variants of Cuckoo filters in Python and measured their construction speed, space efficiency, maximum achievable load factor, insertion throughput and query throughput. Then we implemented Sequence Bloom trees and Cuckoo trees to evaluate the applicability of inter dataset queries. We show that implementations of Cuckoo filters that use efficient bit arrays instead of Python lists are able to achieve close performance to the Bloom filter alternatives.

When we transitioned to testing our Cuckoo Tree in comparison to the Bloom Tree, we see significant slowdown in the construction time and the memory usage of the tree. One of the key differences that results in increased construction time is the difference between the insertion methods of the cuckoo and bloom filters. With cuckoo filters, we potentially have to move fingerprints around which can take time as the filters become filled more. In the next section, we try to summarize potential approaches to continue the project and improve the current implementation.

7 Future Work

Here, we list the potential paths for the future of this project in terms of adding new functionality and improving the current implementation.

7.1 Item Deletion

Item deletion is an optional feature of the cuckoo filters to support dynamic use cases like networking hardware. Since the general use case of the Cuckoo Tree would be searching the tree for query sequences, we postponed adding support for deletions to the future work.

7.2 Achieving Better Space Efficiency

One of the main caveats of implementing these data structure in Python is space inefficiency. However, with special care and design and use of native C++ bindings in Python, we can hope that better bit-per-item ratios are achievable.

7.3 Concurrent Access and Parallel Operations

The current implementation of Bloom filters and Cuckoo filter variants are neither thread-safe nor parallel. Implementing SIMD instructions, leveraging threads and tasklets for parallel insertions and querying are all among possible extensions to the project.

7.4 Assessing Different Scoring Metrics

With the current implementations of our cuckoo tree, we do not allow duplicate insertions. Our tree is therefore more similar to a bloom filter in that we can only say if kmers are present or not present. While allowing duplicates it could be possible to be more stringent in similarity by matching how often kmers exist from a query in a similar database, at memory cost, and we postpone the exploration of this idea to future work.

8 Contributions

8.1 Omar

In terms of the implementation, I worked on the developing the three variants of the Cuckoo Filter: the regular cuckoo filter, cuckoo filter with stash, and the cuckoo-bit filter which was an attempt to make the basic cuckoo filter more memory efficient. I also developed the Bloom Filter class. I wrote the unit-tests for all of those data structures. In terms of the paper, I worked on the abstract, introduction and the sections 4.1 - 4.3 regarding the implementation of the buckets, cuckoo and bloom filters.

8.2 Andrew

I implemented the SBT from [6] and then adapted the ideas from that paper to actually implement the

logic for the Cuckoo Tree (and the CuckooBit Tree variant). I wrote the unit tests for these data structures. In terms of the paper I wrote in section 4 and 7 which regard the implementation of our tree structures.

8.3 Erfan

I created the main driver code and project skeleton for building and evaluating the data structures. I augmented the data structures with performance measurement snippets, created the code for parsing hyper-parameters through command line arguments and created bash scripts for different scenarios in the evaluation section along with the plotting scripts. I was responsible for writing sections 5, 6 and 7 of the write-up.

8.4 Kathleen

I assisted with the original project proposal, and did the background and motivation sections of the mid-project presentation. I wrote section 3 of the write-up.

References

[1] BRODER, A., AND MITZENMACHER, M. Network applications of bloom filters: A survey. *Internet mathematics 1*, 4 (2004), 485–509.

[2] DJ LIPMAN, W. P. Rapid and sensitive protein similarity searches. *Science* (1985).

[3] EPPSTEIN, D. Cuckoo filter: Simplification and analysis. *arXiv preprint arXiv:1604.06067* (2016).

[4] FAN, B., ANDERSEN, D. G., KAMINSKY, M., AND MITZENMACHER, M. D. Cuckoo filter: Practically better than bloom. In *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies* (2014), pp. 75–88.

[5] HAAS, B. J., DOBIN, A., STRANSKY, N., LI, B., YANG, X., TICKLE, T., BANKAPUR, A., GANOTE, C., DOAK, T. G., POCHET, N., ET AL. Star-fusion: fast and accurate fusion transcript detection from rna-seq. *BioRxiv* (2017), 120295.

[6] JIN, X., LI, X., ZHANG, H., SOULÉ, R., LEE, J., FOSTER, N., KIM, C., AND STOICA, I. Netcache: Balancing key-value stores with fast in-network caching. In *Proceedings of the 26th Symposium on Operating Systems Principles* (2017).

[7] KOONIN EV, G. M. *Sequence - Evolution - Function: Computational Approaches in Comparative Genomics*. Kluwer Academic, 2003.

[8] LEINONEN, R., SUGAWARA, H., SHUMWAY, M., AND COLLABORATION, I. N. S. D. The sequence read archive. *Nucleic acids research 39*, suppl_1 (2010), D19–D21.

[9] MOORE, G. E., ET AL. Cramming more components onto integrated circuits, 1965.

[10] SCHATZ, M. C., LANGMEAD, B., AND SALZBERG, S. L. Cloud computing and the dna data race. *Nature biotechnology 28*, 7 (2010), 691–693.

[11] SMITH, T. F., AND WATERMAN, M. S. Identification of common molecular subsequences. *Journal of Molecular Biology* (1981).

[12] SOLOMON, B., AND KINGSFORD, C. Fast search of thousands of short-read sequencing experiments. *Nature biotechnology 34*, 3 (2016), 300–302.

[13] STEPHEN F. ALTSCHUL, WARREN GISH, E. A. Basic local alignment search tool. *Journal of Molecular Biology* (1990).