Final Project Report: SMASH

Nathan Roach, Patricia Walchessen, Charlie Wang, and Yue Zhang

December 8, 2017

Abstract

This project is the exploration of MinHash and SimHash in the field of computational genomics, specifically with comparing genomic sequences. We survey MinHash and confirm the results are as expected, investigate SimHash and how it applies to the field, and identify a shortcoming with regards to k-mer frequencies. We therefore developed SMASH, which addresses the identified shortcoming of SimHash and brings another LSH-like algorithm into the computational genomics field for use in comparing long genomic sequences.

1 Introduction

An intriguing and common problem that arises in computation is efficiently determining similarities between high-dimensional data. The problem reaches into the far corners of computation, including machine learning and kernels. In bioinformatics, the problem takes the form of determining the similarity between both close and disparate long genomic sequences. Solving the challenge of comparing long genomic sequences would aid in evaluating evolutionary relationships, determining the organismal source of sequencing data, and comparing metagenomic data.

One class of algorithms that could solve this problem is Locality-Sensitive

Hashing (LSH) algorithms. LSH algorithms reduce data dimensionality and bin inputs such that similar data sets are grouped together with high probability. They solve the problem of approximate or exact Near Neighbor Search. In recent years, LSH algorithms have offered solutions to problems such as near-duplicate detection, genome-wide association, and both image and audio similarity identification. There are two prevalent LSH algorithms: MinHash and SimHash.

MinHash computes the differences between two genomic sequences through the estimation of a Jaccard Index, a ratio of the shared identity between the sets. SimHash, on the other hand, estimates the cosine similarity by weighting the frequency of the bits, compiling them into vectors and evaluating similarity by the distance between the vectors. The application of MinHash across multiple fields of research has been explored, including in the field of computational genomics. The application of SimHash, on the other hand, has been more limited. After additional research, we found no reference of the exploration of SimHash or any other LSH in the comparison of genomic sequences. Thus, we pursued a research project on the competing LSH, SimHash. This project is a pioneering foray of SimHash into the field of computational genomics.

This project encompasses the exploration of both MinHash and SimHash with respect to the resemblance between genomes. The approach started with the exploration of MinHash, as the research already exists to better further the understanding and knowledge of the field. Next, SimHash was applied to the field. From the ensuing results, SMASH was developed to address a major shortcoming within SimHash.

2 Prior Work

The main contribution of LSH into the computational genomics field was an implementation of MinHash to analyze the resemblance between genomes and metagenomes by Ondov et al. in 2016.[4] They altered the MinHash algorithm to include pairwise mutation distance, or Mash distance, which estimates the rate of sequence mutation under a simple evolutionary model.

Their modifications allowed Mash to efficiently cluster and search massive sequence collections. Mash, as they call their implementation, has two main components: *sketch* and *dist*. The *sketch* function creates MinHash sketches from a sequence or a collection of sequences. *Dist* compares two sketches and returns an estimate of the Jaccard index (ratio of the shared identity between sets), a P value, and the Mash distance.
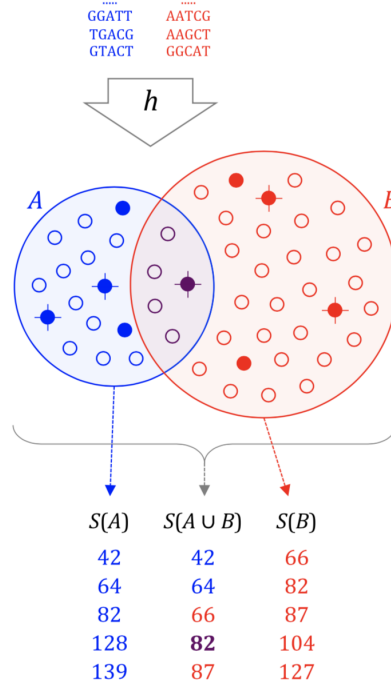
$$J(A, B) = \frac{|A \cap B|}{|A \cup B|} \approx \frac{|S(A \cup B) \cap S(A) \cap S(B)|}{|S(A \cup B)|}$$

Figure 1: Description of MinHash: First, the sequences are divided into their constituent k-mers. Each k-mer is then passed through a hash function to obtain a 32- or 64-bit hash. The Jaccard index is ratio of shared hashes between A and B over all distinct hashes in A and B. This index can be approximated by considering a smaller subset of A and B. MinHash sketches S(A) and S(B) of size 5 are shown for A and B, comprising the five smallest hash values. The Jaccard Index is determined by dividing the similarity between S(A) and S(B) by the distinct values in S(A) and S(B).

Two important parameters within Mash are sketch and k-mer sizes. The larger the sketch size, the better accuracy of the Mash estimates, particularly for divergent genomes. However, the downside is the increase in space and time required for computing the distances. Similarly, the determination of k-mer size alters sensitivity and specificity. Smaller values of k-mer size are more sensitive for divergent genomes, but lose specificity for large genomes due to k-mer collisions. Larger values of k-mer size lose sensitivity. Thus, choosing the smallest k-mer size that reduces chance collisions is important and the optimal k-mer size will change depending on the divergence of the genomes. To maximize sensitivity and specificity, Mash uses a default k-mer size of 21 and a sketch size of 1,000. According to preliminary research by Ondov et al., Mash can be used to cluster whole genomes, identify the genome in real-time from assemblies or reads, and cluster massive metagenomic databases.

Mash has been used to cluster all of the NCBI RefSeq Release 70, totaling 54,118 organisms and 618 Gbp of genomic sequence. The resulting sketches yielded a compression factor of more than 7000-fold versus the uncompressed FASTA (93 MB vs. 674 GB) and only took 33 CPU hours of a typical computer in 2016.[4] Additionally, adding a genome to the database took approximately 0.9 CPU seconds per 5 MB genome. However, fast and space-efficient only works if results are valid. A comparison of Mash distance and ANI, a measure of genetic similarity, among a subset of 500 *E. coli* genomes, found Mash distances correlate well with ANI. For a sketch size of 1,000 and k-mer size of 21, Mash approximates 1 - ANI with a root-mean-square error of 0.00274. Thus, Mash enables scalable, whole-genome clustering.

With a pre-computed sketch database, Mash is able to identify isolated genomes from both assemblies within a few seconds and identify the Lowest Common Ancestor (LCA) from raw sequencing reads within a couple of minutes. Additional work is needed so Mash can be used to discern genome identity from raw sequencing reads, possibly by filtering out low frequency k-mers or increasing the sketch size.

Another important accomplishment is in regard to overcoming the challenge of computing large-scale metagenomic comparison, a problem that more and more are facing. Mash has been shown to cluster metagenomic datasets in a fraction of the time other metagenomic comparison tools use. DSM, which

uses an exact Jaccard Index, and COMMET, which attempts to identify a set of similar reads between two samples using Bloom filters, are both metagenomic comparison tools commonly used today. In a comparison between COMMET and Mash on a pool of Global Ocean Survey (GOS) data, Mash was over ten times faster than COMMET and correctly identified clusters from the original GOS study. Thus, Mash provided a time efficient way to analyze and cluster large-scale metagenomic data. Mash enables the comparison and clustering of whole genomes and metagenomes on a massive scale.

The original paper on SimHash laid out the intuition behind SimHash, allowing a better theoretical knowledge in the quest to develop SMASH.[5] SimHash is an algorithm that first sums up the number of hits per tag used and then runs a hash function on the sum table to convert it to a key. The key is then compared to other keys using the sum of the absolute values of the differences between their entries.

In 2014, Shrivastava et al. compared MinHash and SimHash on binary data and determined MinHash was consistently better than SimHash irregardless of the dataset or the number of hash functions used [6]. When compared in a worst case scenario as well as in an idealized scenario, Shrivastava et al. stated that MinHash outperformed SimHash, which held up in a number of tests on actual data. For binary data, Shrivastava et al. proved theoretically that MinHash is a sensitive family of hash functions for cosine similarity. Although the results for binary data suggest that MinHash is a better LSH algorithm, no tests have yet been done on how SimHash realistically performs on genomic data, an area that is yet to be explored.

3 Research Methods and Software

3.1 Exploration with MinHash

We explored the algorithm and libraries of MinHash to better understand its properties, better understand how MinHash performs and works, and confirm we are able to achieve the results of the MASH paper.

3.1.1 Dataset

The data we ran the MinHash library on was downloaded from the NCBI reference database on December 7, 2017. To determine how MinHash performs on similar data, we downloaded 300+ complete *E. coli* genomes, which were then compared between each other. For development of the phylogenetic tree, we dewnloaded the reference genomes of the following species: *Callithrix jacchus*, *Carlito syrichta*, *Chlorocebus sabaeus*, *Gorilla gorilla gorilla*, *Homo sapiens*, *Macaca fascicularis*, *Macaca mulatta*, *Microcebus murinus*, *Nasalis larvatus*, *Nomascus leucogenys*, *Otolemur garnettii*, *Pan paniscus*, *Pan troglodytes*, *Papio anubis*, *Pongo abelii*, *Rhinopithecus roxellana*, and *Saimiri boliviensis boliviensis*.

3.1.2 Method

We used a Python implementation of Mash called SourMash, which was developed by Titus Brown and Luiz Irber from UC Davis and is available as an open-source software [1].

3.2 Exploration with SimHash

3.2.1 Dataset

The data we tested on encompassed many of the 300+ *E. coli* genomes as well as multiple diverse yeast genomes. For the purpose of the discussion in this section, we will talk about the methods developed based on empirical testing on a dataset consisting of 1 *E. coli* genome and 2 yeast genomes named *S. cerevisiae* and *S. pombe*.

3.2.2 Method

To fully explore the application of SimHash in the genomics field, we commenced by surveying the existing SimHash literature and implementations.

Next, we looked for existing SimHash libraries, many of which were designed for analyzing websites or texts of English words, and compiled the most reputable ones. From there, we selected two based on their reputation and fork numbers for us to base our code on and to adapt into the genomic field to explore SimHash more extensively. The first is Python-based [3] and the second is Go-based [8].

3.2.3 Python-based SimHash

Upon surveying the existing code in the Python-based SimHash, we found there were hard-coded values in the library, such as k-mer or shingle sizes of 4. We thus took account of hard-coded values, then created our own SimHash class based on the Python-based SimHash. The goal with our SimHash class is to enable it to fundamentally accept genomic data and build shingles based on genomic data. Of note was its reliance on using MD5 to hash the keys, then performed the operations of SimHash and the hamming distance on the MD5 hashes. The hashes were also only 32 bit. In all, we coded a more flexible Python-based SimHash library for running our tests of SimHash on genomic data.

3.2.4 Go-based SimHash

Our design and coding of the Go-based SimHash was similar to the Python-based SimHash. By basing our code on the Go library, we made a fundamental modification to accept genomic data. A special particularity with the unmodified Go-based SimHash library is that although the library has a method for making shingles of words, it does not appear to be used. Also, the unmodified Go-based SimHash library assumes the input is string of English words that could or could not be encoded in Unicode. Thus, we made fundamental modifications to the feature construction in the Go-based SimHash library to accept genomic data. Some of the modifications included building the features based on whether it is a substring of lowercase bases or uppercase bases, or building the features based on a k-mer size of 15. We continued with the use of 64 bit hashes in Go.

3.2.5   Evolution of our Research

With both our Python-based SimHash and Go-based SimHash code, we fed in genomic data to test how the core SimHash algorithm performs. Recall that the genomic data consisted of one strain of *E. coli*, approximately 4.5 million bases, and two yeast strands, approximately 12 million bases each. After running the SimHash algorithm in each respective language and observing the output of the distance between the genomic data, we found the results were inaccurate above a certain k-mer length and not always consistent for data below the threshold.

In a comparison between one of the yeast strands and the *E. coli* the SimHash algorithm was able to distinguish that they were quite different. However, the SimHash algorithm outputted the difference between the other yeast strand and *E. coli* vs. the difference between both of the yeast strands was approximately equivalent. Fundamentally, from the sole fact of differences in the length of the genome, the result showed that SimHash was losing resolution.

We first thought these results might be due to how the Python-based SimHash was using the MD5 hash and was implementing SimHash. However, by bringing in the Go-based SimHash, which does not use a MD5 hash but the raw bit values and computes the SimHash fingerprint based on a bit by bit operation with weighting, both returned similarly lackluster results showing the implementation of SimHash is not likely the factor.

We next explored whether the results were because of how we inputted the genomic data. We explored different k-mer sizes and also decided to normalize the data by converting all the bases to lowercase letters. However, even attempts at standardizing the genomic data or exploration of different shingling sizes did not alter the lackluster results from SimHash. Thus, we concluded it is likely due to how the SimHash algorithm is designed.

We further reviewed the literature and discovered that SimHash was primarily designed for finding small discrepancies between websites. Thus, the design principle of SimHash is to have a good high resolution of small amount of changes. Thus, it can be implied that SimHash is unable to accurately handle large differences or hamming distances. With a hamming distance

of over 20, SimHash started losing resolution of the distance between the hashes. Thus, we needed to find a way to make up for the loss of resolution.

3.3 SMASH

Based on what we learned and observed from MinHash and SimHash, we developed our own implementation, SMASH. Because the existing SimHash algorithm was losing resolution when the Hamming distance was too great, we went back to the drawing board and examined each part of the SimHash algorithm closely. From examination and additional brainstorming, we came up with SMASH, designed to primarily address the shortcoming of resolution loss of the distance between the hashes with SimHash.

3.3.1 Design

A major loss of resolution with SimHash occured in the following case: when a *E. coli* genome with approximately 4.5 million bases is compared to one of the yeast genomes with approximately 12 million bases, SimHash outputs the same approximate hamming distance apart as a comparison between to two yeast genomes, both of which have about 12 million bases. Thus, we sought to address the loss of frequency in the resolution.

First, we observed that the basic property of LSHs was the consensus to find a way to build a feature structure out of the dataset. In our case, we decided to build our feature structure based on frequency. For each genome file inputted, we parse it such that it becomes one complete contiguous string. Next, we divide it into its respective k-mers. An important thing to note is that the size of the k-mer needs to be specified beforehand. We test for optimal k-mer size and describe results in the Results section. The frequency is based on how frequently the k-mers show up in the respective genome files.

Based on the order of the feature structure, we then construct vectors consisting of the frequency of the k-mers. This is then fed into the cosine similarity function to compute the similarity. Notice that a fundamental difference between SimHash and SMASH is that, while SimHash focuses on computing

a weighting of frequency of bits, SMASH instead focuses on computing a weighting of frequency of k-mers. For a finer look into what SMASH does, the pseudocode for is in the next section.

3.3.2   Algorithm

```
file1, file2 = input 2 genome files
k = k-mer size

for each file:
    parse and format into one contiguous string

for each file:
    file1_kmers, file2_kmers = divide up file into k-mers

file1_count = dict()
file2_count = dict()

for each kmer in file1_kmers:
    file1_count[kmer] += 1

for each kmer in file2_kmers:
    file2_count[kmer] += 2

for subset of observed k-mers in file_1:
    vector1 = vector of kmer counts from file1_count

for subset of observed k-mers in file_2:
    vector2 = vector of kmer counts from file2_count

similarity_distance = 1 - cosine(vector1, vector2)
```

3.3.3 Further Development

To speed up the counting of k-mers, we explored the use of bloom filters through a Python library called Jellyfish [7]. With Jellyfish, we saw a decrease in the computation time needed to achieve the results highlighted by the above algorithm.

4 Results

4.1 MinHash

In our exploration of MinHash, we compared whole genomes of varying similarity to both replicate the data in the paper by Ondov et al. and evaluate the MinHash algorithm as a whole. To replicate the results of Ondov et al., we ran MinHash on a set of primate genomes to create a pylogenic tree. This tree was then compared to the trees obtained by the UCSC genome browser and Ondov et al. to determine if MinHash was performing correctly (Figure 2). The resulting phylogenetic tree was slightly different from the reference trees. In our result, the Philippine tarsier was closer to the Northern greater galago and Gray mouse lemur. Additionally, we obtained the result that Chimpanzees are closer to Humans than to Bonobos, a result echoed by Ondov et al.'s implementation of MinHash.

To evaluate MinHash on datasets with a high degree of similarity, we ran Minhash on over 300 *E. coli* genomes downloaded from the NCBI Reference database. We were able to cluster *E. coli* strains and map similarities between strains (Figure 3) The minimum similarity within all genomes compared was a distance of 0.378 between *Escherichia coli* strain K-12 substrand MDS42 and *Escherichia coli* APEC O1. This is to be expected as a Multilocus sequence typing (MLST) of these two strains of *E. coli* yielded a similarity of only 98.1, far below the similarity with other *E. coli* strains.[2]
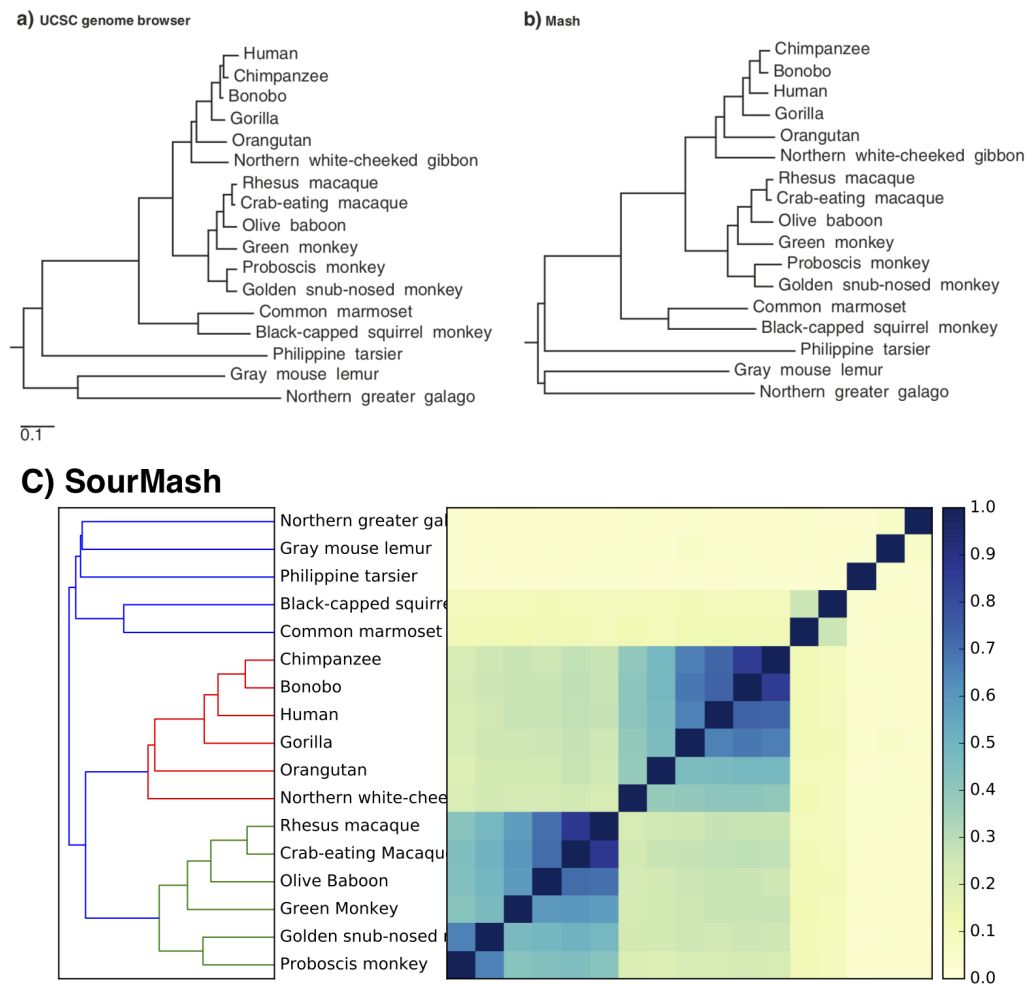
Figure 2: A Comparison of Primate Phylogenetic Trees. **a** A primate phylogenetic tree model from the UCSC genome browser, with branch lengths derived from fourfold degenerate sites extracted from reference gene multiple alignments. **b** Ondov et al.'s Mash-based tree generated from whole genomes using a sketch size of s = 1000 and k-mer size of k = 21.[4] **c** SourMash-based tree generated from whole genomes using a sketch size of s = 1000 and k-mer size of k = 21.
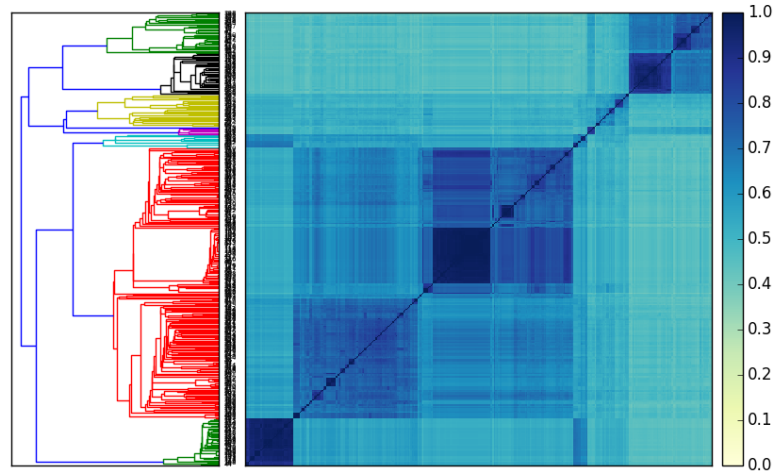
12

Figure 3: A Similarity Comparison of 397 *E. coli* Species. This matrix shows similarity as darker in color. Note the darker colored diagonal line that stretches the matrix. This indicates the high similarity that occurs when a genome is compared to itself. The mminimum similarity within all genomes compared was a distance of 0.378 between *Escherichia coli* strain K-12 substrand MDS42 and *Escherichia coli* APEC O1.
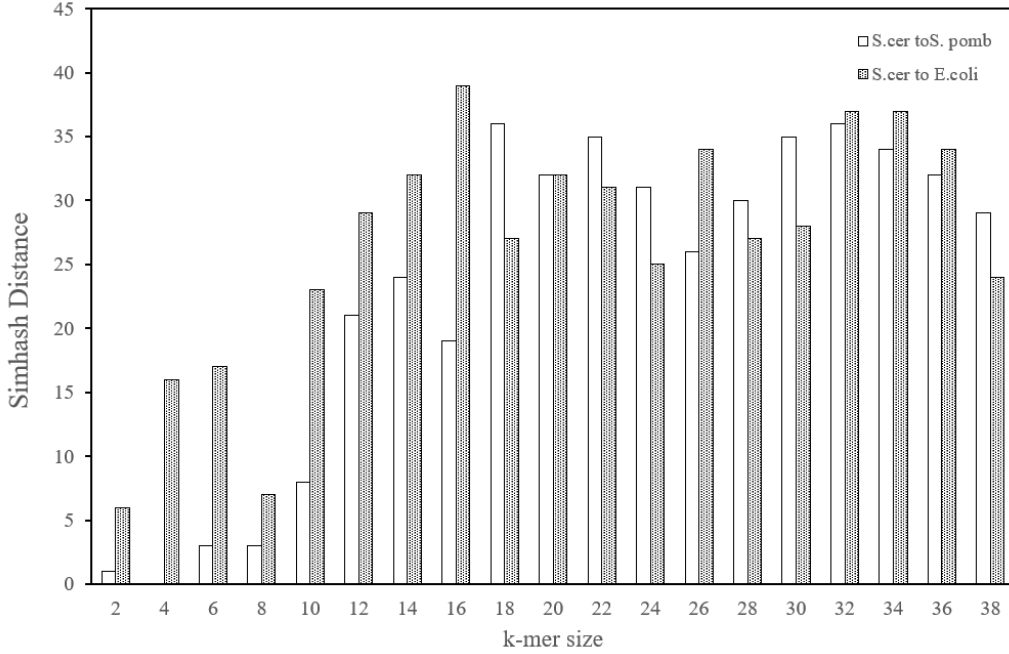
4.2 SimHash

Figure 4: The *S. cer*, *S. pomb* and *E. coli* genomes have been inputted into the SimHash algorithm and the relative SimHash distances have been calculated pairwisely at the k-mer size from 2 to 38 at intervals of 2. As described in the methods section, the SimHash distance between *S. cer* and *S. pomb* is relative shorter than the one between *S. cer* and *E. coli* when the k-mer size is less than or equal to 16. However, as the k-mer size increases above 16, the SimHash algorithm fails to distinguish the relative distance between these 3 genomes.

4.3 SMASH

We performed SMASH comparisons between 18 *E. coli* datasets under a variety parameters, recording both the Jaccard index and the cosine similarity of the counted k-mers. We considered k-mer sizes of 15, 18, and 21. We had two aims in performing this comparison: determining how different the Jaccard index and the cosine similarity are in this implementation, and determining if cosine similarity can successfully determine genetic similarity. We found cosine similarity and Jaccard index were extremely well correlated

14

(Figure 5), and remarkably, that the relationship between cosine similarity and Jaccard index appears to be roughly maintained between k-mer sizes. We also find both cosine similarity and Jaccard index in this implementation results in well defined clusters of similar genome assemblies, indicating evolutionary relationships are being reflected (Figure 6). Known evolutionary relationships appear to be maintained in these results, as in both cosine and Jaccard similarities the similarity between two closely related *E. coli* strains (K-12 AG 100 vs and O6:H16; cosine = 0.756061, jaccard = 0.692374 for k=21) is much higher than the relationship between either strain and a distantly related strain (K-12 AG 100 vs. O169:H41; cosine = 0.650387, jaccard = 0.489714 for k=21); (O6:H16 vs. O169:H41; cosine = 0.577237 , jaccard = 0.466792 for k=21), known evolutionary relationships via genomic BLAST phylogeny tree on NCBI.
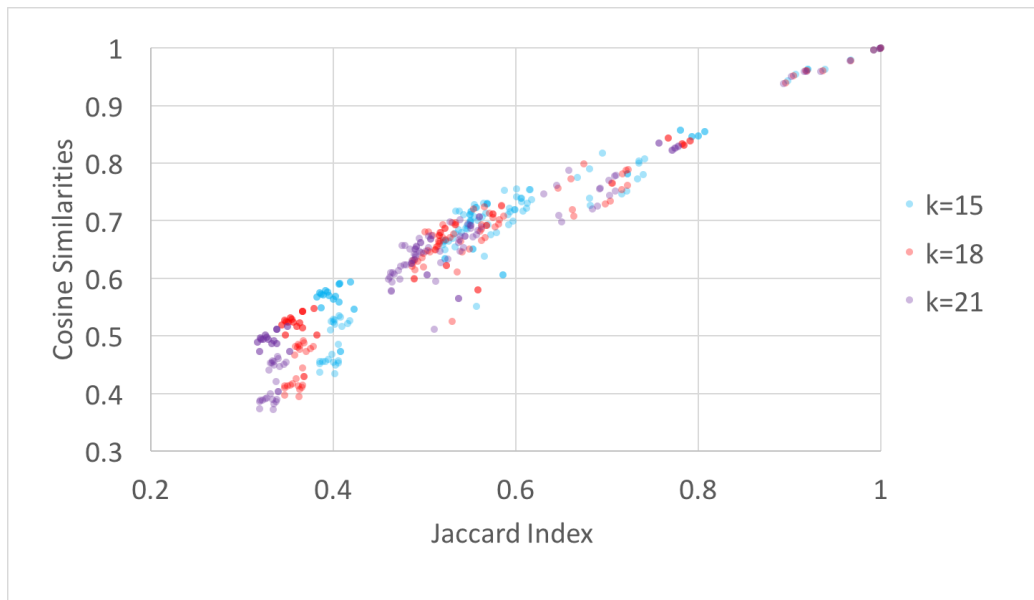
Figure 5: Jaccard index and cosine similarity of k-mers are well correlated when run on E. coli genomes. We plotted cosine similarity vs Jaccard index of compared E coli genomes, and found a strong linear correlation between these two metrics. In addition, this effect appears to be independent with regards to k-mer size, and in fact the relationship between Jaccard index and Cosine similarity appears to be independant of k-mer size in each individual comparison, as evidenced by the near identical dot plots between k-mer datasets.

	ASM274143v1	ASM274145v1	ASM274147v1	ASM274149v1	ASM274153v1	ASM274155v1	ASM274157v1	ASM274159v1	ASM276183v1	GCF_900092615.1	GCF_900096795.1	GCF_900096845.1	GCF_900149875.1	GCF_900174625.1	GCF_900186905.1	ASM276351v1	ASM276399v1	ASM276401v1
ASM274143v1	1	0.6	0.7	0.9	1	1	0.6	0.6	0.6	0.6	0.7	0.7	0.5	0.7	0.6	0.4	0.5	0.5
ASM274145v1	0.6	1	0.8	0.6	0.6	0.6	0.6	0.7	0.7	0.7	0.7	0.7	0.4	0.7	0.7	0.4	0.5	0.5
ASM274147v1	0.7	0.8	1	0.7	0.7	0.7	0.6	0.8	0.7	0.7	0.7	0.7	0.5	0.7	0.7	0.4	0.5	0.5
ASM274149v1	0.9	0.6	0.7	1	1	1	0.6	0.6	0.6	0.6	0.7	0.7	0.5	0.7	0.6	0.4	0.5	0.5
ASM274153v1	1	0.6	0.7	1	1	1	0.6	0.6	0.6	0.6	0.7	0.7	0.5	0.7	0.6	0.4	0.5	0.5
ASM274155v1	1	0.6	0.7	1	1	1	0.6	0.6	0.6	0.6	0.7	0.7	0.5	0.7	0.7	0.4	0.5	0.5
ASM274157v1	0.6	0.6	0.6	0.6	0.6	0.6	1	0.6	0.7	0.7	0.8	0.8	0.4	0.7	0.7	0.4	0.5	0.5
ASM274159v1	0.6	0.7	0.8	0.6	0.6	0.6	0.6	1	0.6	0.6	0.7	0.7	0.4	0.7	0.7	0.4	0.5	0.5
ASM276183v1	0.6	0.7	0.7	0.6	0.6	0.6	0.7	0.6	1	0.7	0.8	0.8	0.5	0.8	1	0.4	0.5	0.5
GCF_900092615.1	0.6	0.7	0.7	0.6	0.6	0.6	0.7	0.6	0.7	1	0.8	0.8	0.4	0.8	0.8	0.4	0.5	0.5
GCF_900096795.1	0.7	0.7	0.7	0.7	0.7	0.7	0.8	0.7	0.8	0.8	1	1	0.5	0.8	0.8	0.4	0.5	0.5
GCF_900096845.1	0.7	0.7	0.7	0.7	0.7	0.7	0.8	0.7	0.8	0.8	1	1	0.5	0.8	0.8	0.4	0.5	0.5
GCF_900149875.1	0.5	0.4	0.5	0.5	0.5	0.5	0.4	0.4	0.5	0.4	0.5	0.5	1	0.5	0.5	0.5	0.6	0.6
GCF_900174625.	0.7	0.7	0.7	0.7	0.7	0.7	0.7	0.7	0.8	0.8	0.8	0.8	0.5	1	0.8	0.4	0.5	0.5
GCF_900186905.1	0.6	0.7	0.7	0.6	0.6	0.7	0.7	0.7	1	0.8	0.8	0.8	0.5	0.8	1	0.4	0.5	0.5
ASM276351v1	0.4	0.4	0.4	0.4	0.4	0.4	0.4	0.4	0.4	0.4	0.4	0.4	0.5	0.4	0.4	1	0.6	0.6
ASM276399v1	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.6	0.5	0.5	0.6	1	1
ASM276401v1	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.6	0.5	0.5	0.6	1	1

Figure 6: Use of cosine similarity can pull out clusters of similar E coli genomes, indicating that evolutionarily similar genomes are being recognized.

Despite jellyfish speeding up k-mer counting considerably, the time to compare our data sets remains high using this method, in part due to high memory overhead of loading a large hash of k-mer counts for comparing counts. An ideal implementation would reduce the size of k-mer vector being considered, thereby reducing memory overhead. We attempted to utilize the bloom counter function built in to jellyfish to filter out single-count k-mers in the counting step. Unfortunately, the documentation provided for using the jellyfish bloom-counter appears to be out of date, and debugging this approach was not completed.

5 Conclusion

Similarly to how MinHash needed to be adapted for genomic data processing with the culmination of the Mash algorithm, SimHash also needed to be adapted for genomic data processing with the culmination of the SMASH algorithm. Through the exploration and verification of Mash, followed by exploration of SimHash and identification of loss of resolution with respect to frequency of k-mers, to our development of the SMASH algorithm, we show that MinHash is not the only LSH applicable to the computational genomics field. Further refinement of the application of LSH into the computational genomics field would help increase performance not only of MinHash, but also SimHash.

6 Contributions of Each Member

- Nathan Roach: Nathan worked on a de novo implementation of simhash, implemented in smash.sh and compare_jfs.py (with a deprecated implementation in npr_smash.py). This involved counting k-mer frequencies using jellyfish and finding and comparing the cosine similarity and jaccard index of the counted k-mers between *E. coli* datasets. He also worked on speeding up the smash algorithm by implementing jellyfish's bloom counter. Unfortunately, as discussed above, this did not work.

- Patricia Walchessen: Patricia implemented MinHash, replicating results and exploring how MinHash performed calculating the similarity of genomes of varying divergence. Wrote prior works section and all sections regarding MinHash.

- Charlie Wang: Charlie worked on the direction and approach of the project, primarily with the evaluation of the SimHash algorithm, both the Python and Go, and the continuation from there to SMASH. Charlie also worked on the prototyping of the code and drafted many sections in the report.

- Yue Zhang: Yue Zhang worked on implementing the SimHash algorithm to analyze the relative genomic distance between *S. cer*, *S. pomb* and *E. coli* at different kmer sizes.

References

[1] C. Titus Brown and Luiz C. Irber, Jr. *SourMash*. GitHub repository, https://github.com/dib-lab/sourmash, 2016.

[2] Timothy J. Johnson, Subhashinie Kariyawasam, Yvonne Wannemuehler, Paul Mangiamele, Sara J. Johnson, Curt Doetkott, Jerod A. Skyberg, Aaron M. Lynne, James R. Johnson, and Lisa K. Nolan. *The Genome Sequence of Avian Pathogenic Escherichia coli Strain O1:K1:H7 Shares Strong Similarities with Human Extraintestinal Pathogenic E. coli Genomes*. Journal of Bacteriology, 189(8):3228-3236, 2007.

[3] Leon. *LeonSim* GitHub repository, https://github.com/leonsim/simhash, 2017.

[4] Brian D. Ondov, Todd J. Treangen, Páll Melsted, Adam B. Mallonee, Nicholas H. Bergman, Sergey Koren and Adam M. Phillippy. *Mash: fast genome and metagenome distance estimation using MinHash*. Genome Biology, 17:132, 2016.

[5] Caitlin Sadowski and Greg Levin. *SimHash: Hash-based Similarity Detection*. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.473.7179&rep=rep1&type=pdf, 2007.

[6] Anshumali Shrivastava and Ping Li. *In Defense of MinHash Over SimHash*. Journal of Machine Learning, 33, 2014.

[7] James Turk and Michael Stephens. *Jellyfish*. GitHub repository, https://github.com/jamesturk/jellyfish, 2017.

[8] *SimHash* GitHub repository, https://github.com/mfonda/simhash, 2015.