Sequence analysis

Implementing Long Read Mapping Algorithms

Charu Mehndiratta, Ritu Prajapati and Saish Sali

Department of Computer Science, Stony Brook University, Stony Brook, NY 11790

Abstract

Motivation: Sequencing is the process of copying and fragmenting the DNA into reads. Assembly is the process of constructing genome from sequencing reads. However, assembly becomes difficult when the repeat length exceeds the read length because shorter reads fail to resolve repetitive sequences. Longer reads are the results of third-generation sequencing of genomes and transcriptomes which, unlike the short read sequencing, (i.e. second or "next"-generation sequencing) methodologies, generate a very long substrings of the reference, but have a higher error rate compared to short reads (~10-15% vs < 1% for short reads) and hence should be treated using algorithms, data structures and approaches other than those commonly used in short read analysis. Overlapping is the slowest part of the assembly. The naïve approach to find overlap between reads is all vs all alignment. Other approachs based on suffix tree and glocal alignment (using dynamic programming) have a complexity of $O(N^2)$ where N is the total length of all reads. To solve the computational problem of long read assembly, approaches that use the idea of Min hash and Containment Hash create a compact representation of sequencing reads. The goal of this project is to implement Min Hash and Containment Hash approachses in C++ and compare the performance of these two approaches on the simulated data.

Results: Containment Min Hash shows a significant improvement in terms of space and time complexity and is more suitable for estimating similarity (using Jaccard index) of sets of different size. In addition to being time and memory efficient, it is significantly more accurate than Min Hash. The comparision of the relative error of the traditional min hash and containment hash approach on sets of simulated data as a function of number of hash functions used and k mer size shows that the containment hash approach has significantly less error.

Availability and implementation: https://github.com/CharuMehndiratta/cse549

Contact: saish.sali@stonybrook.edu

1 Introduction

Genome assembly is a process of constructing a genome from a set of sequencing reads. One of the ingredients for a good assembly is read length. Genome assembly is required because the genome sequencing technologies cannot read the whole genome in a single walk, instead it produces many small segments of the genome by making multiple copies of the genome and then randomly breaking them in many small segments. One of the naive solution for genome assembly is finding the shortest common superstring (SCS) on the reads. Although SCS is known to be a NP-Hard problem, a greedy algorithm can find a non-optimal superstring in reasonable time and space. However, there is one problem with the greedy approach, it collapses repeats in the genomes with its 'shortest' criterion. Hence the read length should be large enough to cover the repetitive regions. Assembly becomes difficult when there are repetitive sequences and the repeat length exceeds the read length. Reads must be longer than the repeats because shorter reads have false overlaps that forms hairball assembly graphs. With long enough reads, we can assemble entire chromosome into contigs. Shorter reads fail to resolve repetitive sequences and thus we require reads of sufficient length (spanning across repeats) that are able to resolve repeats in the genome.

The long reads generated as a part of genome sequencing have high error rates and low accuracy and new algorithms are needed to perform computations on long reads. One of the bottlenecks of long read assembly is the all-versus-all alignment for determining reads that overlap. This computational problem can be solved by using approaches that use the idea of Min Hashing and related algorithms that have recently drawn a lot of attention in the computational biology community, and have been shown to be useful in assembling and mapping long reads. In this approach, which is borrowed from NLP and estimating document similarity, we use a Min Hash to approximate overlap of two long reads or approximately map a long read to a reference. In addition to this, another approach named Containment Hashing has been proposed that has an advantage over Min Hashing in the way it addresses potential length differences between the two strings being compared.

The goal of this paper is to implement the Min Hash and Containment Hash approaches in C++ (one of the implementations is available in Python in **MinHashMetagenomics** repository on github) and using the simulators **SimulatedBiologicalData** and **SimulatedBiologicalDataSmall**, compare the performance of these two approaches on the simulated data.

2 Methods

Two sets are said to be more similar when the ratio of their intersection to their union is close to 1. This ratio is the Jaccard Similarity Coefficient. The Jaccard similarity for sets A and B is given as follows:

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|}.$$

Set similarity is abundantly used in finding the similarity between two documents. The documents are represented as sets by segmenting the documents in k fixed length overlapping substrings called shingles. The two sets are compared with each other to find the jacquard index. If the jacquard index is 1 or near 1, the documents are said to be similar otherwise not.

The same approach used in document comparison can also be applied to long reads comparison. One of the most common applications of set similarity is a comparison of long read sequences. By representing long reads as sets of words, we can use the Jaccard Similarity as a measure of how much overlap is there between read pairs. Every length k substring that appears in a sequence is known as kmer (shingles) and the technique of hashing substring is referred to as shingling. The conventional way to calculate the set intersections and union is by comparing each pair of kmers in the two sets. However, this approach would cost a lot of computational time because high coverage yielding large number of overlapping reads is necessary for good genome assembly. The goal of Min Hash is to calculate the Jaccard index quickly by limiting the number of comparisons. A further improvement can also be made using containment hash for better performance and true results. This paper is a brief overview of our implementation of Min Hash and Containment Hash algorithm for long read assembly.

2.1 True Jaccard Algorithm:

The conventional approach for finding the similarity between two reads is to calculate true jaccard according to the algorithm below

1. Take two sequencing reads.
2. Divide both the reads into k-mers and populate them into corresponding sets.
3. Find intersection of the two sets.
4. Find union of the two sets.
5. Calculate the jaccard index by dividing the intersection of the sets by the union of the sets
6. If true jacquard index of the two reads is near 1, the reads are said to be similar.

The issue with this approach is that the total number of elements in each set depends on the length of the read and as the length of the read increases the number of elements increases. Moreover, to find the intersection between two sets of kmers, each pair must be compared. In the worst case the complexity of finding jaccard similarity is quadratic O(nm), where n and m are the sizes of kmer sets of the two reads.

This problem can be solved by MinHash and Containment Hash where the size of the set is bounded by the number of hashes. The basic idea of Min Hash approach is to convert all kmers to integer fingerprints using multiple, randomized hash functions. For each hash function only the min valued integer fingerprint is retained. The collection of such minimum valued fingerprint for a sequence makes a sketch. This allows estimating the Jaccard similarity of two sets by computing Hamming Distance between their sketches. As the sketch size is significantly smaller than the total number of k-mers to be compared, this is a computationally efficient technique for estimating similarity.

2.2 Min Hash Algorithm:

1. Take two sequencing reads as input.
2. Select size of the sketch(Number of hash functions)
3. Generate a random hash vector upto the size of sketch.
4. Select a sequence one by one and generate k-mer substrings.
5. For each k-mer, generate all the hash functions required in the sketch and keep only the one that has the smallest value.
6. Together with sketch values, keep the k-mer that generated the min-hash value for ith hash function. This k-mer string will later be used in the sequencing procedure.
7. For calculating the hash functions, we used MurmurHash3. Murmur hash is a non-cryptographic function which uses three operations multiply, rotate and XOR to provide the hash value.
8. As now we have the sketch for both the reads, the jaccard similarity can be calculated using the intersection and union on these two sketches.
9. The jaccard similarity between two sequencing reads is calculated as:

$$\frac{\text{Count of positions where sketch 1 and sketch 2 values are equal}}{\text{Size of the sketch}}$$

Greater the value of jaccard similarity (near 1), greater the similarity between two sequencing reads. The total number of comparisons in Min Hash algorithm is bounded by the size of the sketch. The advantage here is that we can choose the size of the sketch i.e, the number of hash functions according to the input.

The problem with this approach is that the performance reduces when the two sequencing reads are different in size. We are using the same sketch size for both the sequences. However that will not be the case every time. Suppose we want to calculate the jaccard index between a reference genome and a read. A size of genome is very large compared to the read. So this method may not produce the best results here.

Containment min hash, a new and efficient approach is more suitable for estimating the Jaccard Index of sets of different sizes. This approach is significantly more accurate, has smaller computational complexity and utilizes less memory than the traditional Min hash approach.

2.3 Containment Hash Algorithm:

1. For the larger sequencing read (say, s1), instantiate and configure bloom filter B with projected number of elements (l1 - k + 1), false positive probability (p) of 0.001 and random seed
2. Add kmers for sequencing read s1 to the bloom filter
3. Generate shingles for sequencing read s2
4. Convert all k mers generated in step 3 to integer fingerprints using multiple, randomized hash functions (say, Murmur3 hash with random seed values)
5. For each hash function, only the shingle corresponding to the minimum valued fingerprint, or min-mers is retained. The collection of min-mers of a sequence makes the sketch.
6. For all the kmers in the sketch of sequencing read s2, query for its existence in the bloom filter B. Compute number of such memberships in Y.
7. Compute containment estimate, C = (Y / h) – p
8. Compute Jaccard index = (C * l_2) / (l_1+ l_2 – l_2 * C)

3 Implementation

The code has been implemented in two parts. In first part the program asks for a reference genome file, number of hash functions to be used and false positive rate of the bloom filter. If not specified, default values for kmer size (11), number of hash functions (100) and false positive rate (0.001) are used. The program reads the genomes from the input file and calculate the minimum sketch according to the minHash algorithm, and the bloom filter according to the containment hash algorithm. The sketches and bloom filters for all the input genomes are stored in a file which will be an input to the second part of the algorithm. All code is available at

Github URL: https://github.com/CharuMehndiratta/cse549

Command line to run part 1 of the implementation,
bash> make

bash> ./build_indices -r <reference_genome_file> -f <False_positive> -k <kmer size> -h < number of hash functions >

The second part of the algorithm calculates the jaccard index using true jaccard, minHash and containment Hash algorithms. The input parameters such as kmer size, number of hash functions and false positivity are used from part 1. A long reads file should be provided to the program. The program reads the intermediate files of reference genome bloom filter and minimum sketch generated from part 1. For each read the program calculates the jaccard index using the theree algorithms. The output is then stored in files.

Command line to run part 2 of the implementation:

bash> ./query -r <long reads file> -i <index file>

The index file holds the values of kmers size, number of hashes and false positive from part 1 of the implementation.

4 Results

Figure 1a and 1b represents comparison of the relative error of the containment min hash approach to the classical min hash estimate of the Jaccard index as a function of number of hashes used on simulated biological data generated using Makepaper.sh script. a) SimulatedBiologicalDataSmall On 16 replicates of samples consisting of 20 genomes Gi with only 10K reads, showing the similarity of the methods in estimating J(M1, Gi) when the sets to be compared are roughly the same size. b) SimulatedBiologicalData On 16 replicates of samples consisting of 20 genomes Gi with 1M reads, demonstrating the improvement of the containment approach in estimating J(M2, Gi) when the sets are of significantly different size.

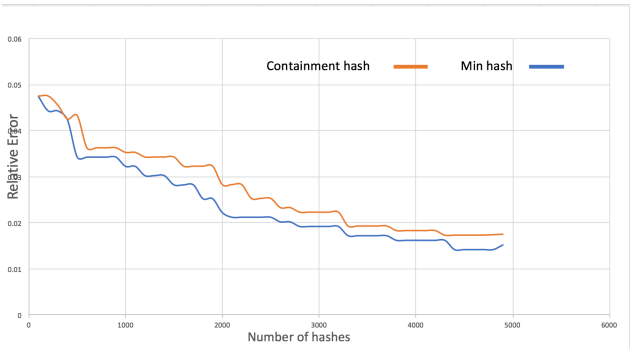

Figure 1a

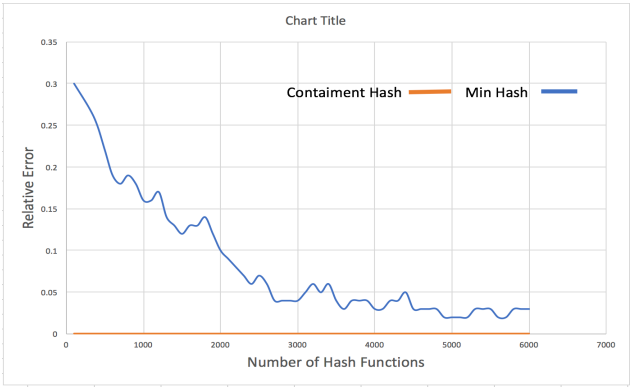

Figure 1b

Figure 2a and 2b represents comparison of the relative error of the containment min hash approach to the classical min hash estimate of the Jaccard index as a function of number of kmers on simulated biological data generated using Makepaper.sh script. a) SimulatedBiologicalDataSmall On 16 replicates of samples consisting of 20 genomes Gi with only 10K reads, showing the similarity of the methods in estimating J(M1, Gi) when the sets to be compared are roughly the same size. b) SimulatedBiologicalData On 16 replicates of samples consisting of 20 genomes Gi with 1M reads, demonstrating the improvement of the containment approach in estimating J(M2, Gi) when the sets are of significantly different size.

Figure 2a

Figure 2b

Analysis:
Construction Time:
Time to construct Min sketch and bloom filter for reference genomes - 22 minutes for 20 genomes generated using Simulated Biological Data

Query Time -
Read length = n
Hash functions = h
Kmer size = k

Time to construct read min hash sketch = n - k = O(n)
Time to fetch min sketch of reference genome = O(h)
Time to compute Jaccard similarity = O(h)
Overall time complexity = O(h + n)

Time to fetch reference genome bloom filter object = O(1)
Overall time complexity = time to check n - k kmers in bloom filter library

5 Conclusion

In this project we implemented MinHash and Containment Min Hash approaches in C++. We showed that Containment Min Hash is significantly more accurate, has smaller computational complexity and utilizes less memory than the traditional min hash approach. In case of Containment Min Hash, we sample elements only from the smaller set and use approximate membership query data structure (Bloom Filter) to quickly test if this element is in the larger set. This is used to estimate the containment index, which is then used to estimate the Jaccard index itself

6 References

- https://github.com/dkoslicki/MinHashMetagenomics
- https://www.biorxiv.org/content/biorxiv/early/2017/09/04/184150.full.pdf
- http://www.nature.com.proxy.library.stonybrook.edu/articles/nbt.3238.pdf
- https://rob-p.github.io/CSE549F17/lectures/Lec12.pdf
- http://mccormickml.com/2015/06/12/minhash-tutorial-with-python-code/