Bioinformatics
doi.10.1093/bioinformatics/xxxxxx
Advance Access Publication Date: 2 April 2015
Applications Note

Computational Biology Fall 2017

Implementing Long Read Mapping Algorithms

Gopalakrishnan Nallasamy (110975029), Shreyas Chikkabilathi Harisha (111464609), Atmika Sharma (111464371) and Sourav Mishra (110946489)

Department of Computer Science, Stony Brook University, Stony Brook, New York

Abstract

Motivation: Long 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. Specifically, approaches that use the idea of Min Hashing or related algorithms 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.

Supplementary information: The code for our implementation is available here. *https://github.com/Sheldor-007/CSE549*

1 Introduction

As data sets become larger and more high-dimensional, it becomes increasingly important to find data representations that allow compact storage and efficient distance computation and retrieval. Among the common methods to achieve this is Locality Sensitive Hashing (LSH)[5]. The goal of this project is to implement the Min Hash and Containment Hash approaches in C++ and using the simulators SimulatedBiologicalData and/or SimulatedBiologicalDataSmall, compare the performance of these two approaches on the simulated data. We then implement a very basic mapper that given a set of input long reads and a reference (a genome or transcriptome) maps long reads to the reference just if the Min Hash sketch of long read and genome have a similarity higher than a threshold.

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 as an advantage to Min Hashing is addressing potential length differences between the two strings being compared.

Here are some terminologies we will be coming across:

- *Locality Sensitive Hashing:* Locality-sensitive hashing (LSH) reduces the dimensionality of high-dimensional data. LSH hashes input items so that similar items map to the same âŁœbucketsâŁž with high probability (the number of buckets being much smaller than the universe of possible input items)[1].
- *Jaccard Index:* The Jaccard index, also known as Intersection over Union and the Jaccard similarity coefficient (originally coined coefficient de communautÃ© by Paul Jaccard), is a statistic used for comparing the similarity and diversity of sample sets. The Jaccard coefficient measures similarity between finite sample sets, and is defined as the size of the intersection divided by the size of the union of the sample sets[3]:

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

- *Containment Index:* To compare the relative size of the intersection to the size of A, we similarly define the containment index of A in B (both non-empty) as[4]:

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

- *Bloom Filter:* Given a set B with cardinality $n = |B|$, a bloom filter $\tilde{B} = (\tilde{B}_i)_{i=1}^{m}$ is a bit array of length m where m is a chosen natural number. Fix a set of hash functions $h_1, \ldots, h_k$ each with domain containing B and with range $\{1, \ldots, m\}$. Initially, each entry in the bloom filter is set to zero: $\tilde{B}_i = 0$ for each $i = 1, \ldots, m$. For each $x \in B$ and $j = 1, \ldots, k$, we set $\tilde{B}_{h_i(x)} = 1$.

1.1 Min-Hash

Min Hash: MinHash (or the min-wise independent permutations locality sensitive hashing scheme) is a technique for quickly estimating how similar two sets are. The scheme was invented by Andrei Broder (1997), and initially used in the AltaVista search engine to detect duplicate web pages and eliminate them from search results.[2] It has also been applied in large-scale clustering problems, such as clustering documents by the similarity of their sets of words[2]. It uses random sampling to estimate the Jaccard index. Using the Chernoff bounds, the probability of deviating from the true value grows exponentially as the size of the true Jaccard value decreases to zero. Hence, min hash returns tight estimates only when the true Jaccard index is large. This requirement for a large Jaccard index limits this technique to situations where the sets under consideration are of similar relative size and possess significant overlap with each other[4].

Here is a brief description of how the traditional min hash works:

The traditional min hash randomly samples from the union $A \cup B$ and uses the number of sampled points that fall in $A \cap B$ to estimate the Jaccard index. With more sampled elements falling in $A \cap B$, the more accurate the Jaccard estimate will be[4].

1.2 Containment Min-Hash:

Containment Min Hash: The containment min hash approach we propose differs from the classic min hash in that the family of k hash functions $\{h_1, \ldots, h_k\}$ have domain containing A and we randomly sample from A instead of $A \cup B$. This results in estimating the containment index $C = C(A, B)$ instead of the Jaccard index $J = J(A, B)$, but we will late show how to recover an estimate of $J(A, B)$. The containment approach proceeds as follows: Let $\tilde{B}$ be a bloom filter with given false positive rate p and define the random variables

$$Y_i = \left\{ \begin{array}{ll} 1 & h_i^{\min}(A) \in \tilde{B} \\ 0 & \text{otherwise.} \end{array} \right\}$$

These random variables essentially sample (uniformly) randomly from A and tests for membership in B via the bloom filter $\tilde{B}$[4].
Here is what the Containment Min-Hash essentially does:
In the containment min hash approach, we randomly sample elements only from the smaller set (in this case, A) and use another probabilistic technique (a bloom filter) to quickly test if this element is in B (and hence in A âˆ© B). This is used to estimate the containment index, which is then used to estimate the Jaccard index itself[4].
Why is there a need for Containment Min-Hash?
This containment approach experiences decreasing error as more points in A âˆ© B are sampled, and so sampling from a smaller space (the set A instead of A âˆª B) leads to significant performance improvements[4].

2 Approach

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 as an advantage to Min Hashing is addressing potential length differences between the two strings being compared[4].

To start with, we run the Make file which creates executables for Min-Hash and Containment Hash. This results in the creation of shared libraries that are then used as input modules in the basicWorkflow.cpp file that we have created. Make also calls the createVirusesMinHashSketches.cpp which computes the Min-Hash on the data set that has been downloaded. This is where the min hashes for the data is computes and results in a 40 minute execution period in python which turns out to be only 8 minutes for the C++ implementation.

Basicworkflow: Here, we deal with 2 strings, 1 small and 1 large. The following gives us an estimate of how the comparison occurs: We first append the small string to the larger string making sure the larger string ow contains the smaller string. We also define values of variables like k-mer size, probability error rate, maximum number of hashes etc. We then create k-mers out of the two strings and calculate their True Jaccard value. We save the True Jaccard value obtained.

Successively, we create a CountEstimator object for Min-Hash. We then add the sequence of the small string and create another CountEstimator and add the large string's k-mers as well. The above two CountEstimator objects will have different min hashes calculated for both the strings. We then call the jaccard() function on those two sets of min hashes between the small string and the large string so that the estimated jaccard index value through Min-Hash is stored in min_{jac}
Containment Hash follows along the same lines. The bloom filter parameters are initialized. We feed the k-mers of the larger string as input. We then calculate Containment Hash for the smaller string and with that computer the Containment estimate.
We have displayed the Min-Hash and Containment Hash index estimates along with the relative errors from the True Jaccard index.

Now we will discuss the working of our minhash.cpp workflow. Here, we first define a class called CountEstimator. Each time we need to create min hashes for a large string, we create a CountEstimator object and send that particular string to the CountEstimator object which internally creates a list of k-mers based on the length of the string and the k-mer size we defined that can be passed as an argument to this CountEstimator while creating the object.
We then obtain k-mers out of it and from which we get the min hashes. We avail the option of saving these to a file.

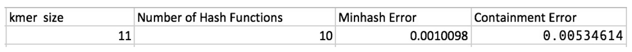
The working of the CountEstimator is as follows:
There exist small basic functions within the CountEstimator class.
The constructor initializes parameters such as ksize, list, $_mins$, $_counts$ etc.
The $down_sample()$ function down samples the number of sketches for it to have h elements which has been defined as 500 to reduce the number of elements we need to save.
When we create a count estimator we add the string through a function called $add_sequence()$. This $add_sequence()$ function takes the sequence we wish to create a min hash of as the input. It internally calls the $add()$ function which in turn creates the k-mers and also creates the hash function for them using murmur hash. Once the min hashes for these are created, they are stored in the $_min$ list which is internally maintained. A particular $_min$ list has the list of min hashes for a given sequence.
The jaccard() function calculates the jaccard similarity between this object's min hashes and takes as argument, a CountEstimator object. Thus, it basically compares another CountEstimator object's min hashes to the min hashes of this object and gives us the jaccard similarity.
Now, we consider the Containment Hash. It uses a Bloom Filter for storing the available data set. The object is initialized with the false positive probability rate, element count, seeds etc.
We again call the add() function which inserts the list of k-mers that we sent to the containment hash and store it.
The contains() function gives output an output of either a 1 or a 0 based on the query of whether it contains the k-mer or not. Based on this query of presence of the k-mer, similarity will be calculated.
Looking at the createVirusesMinHashSketches.cpp file, we consider the directory where the downloaded dataset with the virus genomes is present. A file containing a list of other filenames for the virus genome sequences is available. It populates the list and assigns it to multiple

different threads which run in parallel. The number of threads made use of is decided based on the number of cores that the system is running on. A filename will be assigned to each thread based on the availability of that particular thread. Each thread is invoked with the $make_minhash()$ function which creates a min hash from the sequence in the file and saves it along with the k-mers. The min hashes will require these k-mers when we compute the presence of a virus sequence in the mapper function. We will use these k-mers for querying the presence of the new virus k-mers against the k-mers in the database that we already have.
This is where we parallelize our implementation which reduces the execution time from 40 to 8 minutes.

We now look at the mapper.cpp file: It takes a small string as the input and calculates the similarity of that small string against the database of the genomes for which we have already calculated the min hashes and k-mers. We obtain the similarity between this small string and the virus database already present with us. It makes use of both Min-Hash and Containment Hash and gives us the Jaccard similarity for both of them.

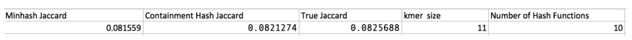
3 Results

The results were generated for a wide variety of k-mer values and number of hash functions.
The values for a k-mer size 11 and number of hash functions 10 is given below:-



kmer size	Number of Hash Functions	Minhash Error	Containment Error
11	10	0.0010098	0.00534614

From the result, it is evident that Containment MinHash performs better compared to Minhash. This can be attributed to the fact that Containment MinHash combines MinHash and Bloom Filters to allow Jaccard Index for sets of very different sizes. Although, MinHash performs similar to Containment Hash when it comes to short reads it underperforms and Containment MinHash performs much better.



Minhash Jaccard	Containment Hash Jaccard	True Jaccard	kmer size	Number of Hash Functions
0.081559	0.0821274	0.0825688	11	10

The increase in the k-mer size led to improved results. This is possible due to the fact that with longer reads the functions are better able to deal with repeating sequences and that leads to a lesser chances of not being able to account for them.
The change in the number of hash functions also led to a change in the performance. This can be attributed to the reduction in collisions and better efficiency with the increase in the number of hash functions.

4 Discussion

The time taken for calculating the min hash of the sketches improve 5 fold with the new C++ implementation. The previous python implementation had a running time of around 40 minutes which our C++ implementation is able to perform in around 8 minutes. We observe that varying the parameters gives us varying results for values such as relative error etc. We see that with the same basic approach and similar data used for both the implementations, the only difference we observe is the execution time.
According to the reference paper [4], given sets A and B of size m and n respectively, both the classic min hash and containment min hash require O(m+n) time to form their respective data structures. When calculating their estimates of the Jaccard index J(A, B), both approaches use computational time linear in the number of hash functions required (in the case of the containment min hash approach, this is due to Bloom filter queries being constant time). The ratio of time complexity of the classical min hash to the containment min hash is $O(\frac{k_X}{k_Y})$. When B is very large in comparison to A, this implies that the containment min hash approach is significantly faster than the classical min hash approach[4]. Our results show that this inference is correct and that improvement has been seen in the execution time for the containment min hash.

5 Conclusion

We have implemented MinHash and Containment Hash for long reads along with a very basic mapper for the two. These have been implemented in C++. Our implementation has resulted in execution time improvements which is the primary goal for implementing Min-Hash and Containment Min-Hash in C++. Although the results reflected the expectations we had from the code, changing the way the reads are taken and file I/O is conducted in general, further improvements can be made in performance.

Acknowledgements

We would like to thank Professor Rob Patro and Teaching Assistant Fatemeh Almodaresi for their guidance throughout the course of this project. This project would not have been a success without their support.

6 References

$1 : https :$
$//en.wikipedia.org/wiki/Locality - sensitive_hashing$
2: https://en.wikipedia.org/wiki/MinHash
$3 : https : //en.wikipedia.org/wiki/Jaccard\_index$
4: David Koslicki, Hooman Zabeti: Improving Min Hash via the Containment Index with Applications to Metagenomic Analysis:
https://github.com/dkoslicki/MinHashMetagenomics/blob/master/Paper/ImprovedMinHashForMetagenomics
5: Sergey Ioffe: Improved Consistent Sampling, Weighted Minhash and L1 Sketching:
https://static.googleusercontent.com/media/research.google.com/en//pubs/archive/36928.pdf
6: https://sites.google.com/site/murmurhash/