Edmund Zhi

UID: 805328871

Professor Harold Pimentel

COM SCI CM121, Lecture 1

18 March 2022

Pseudoalignment Project Report

**Introduction**

In the context of RNA-seq quantification, a computational problem arises when classical approaches for exact matching of isoform targets to RNA-seq fragments to recover RNA distribution is too slow. Using a unique approach that drops the assumption that position matters and taking advantage of the redundancy of target sequences, we recognized that equivalence classes, or a set of transcripts that a read is consistent with, are sufficient for such quantifications. We use equivalence classes in the context of pseudoalignment, a process in which we create a map from sequence to equivalence classes.

Our objective was to implement a pseudoalignment algorithm that could take in a transcriptome and RNA-seq reads in FASTA file format. Using the transcriptome, the algorithm should be able to build an index, or hash table of k-mers, given some specified k, and hash the isoforms in the transcriptome based on whether they contain a particular k-mer. These are the equivalence classes. Then, iterating through the reads from RNA-seq data, we want to know how many reads belong to each equivalence class, and return these values as our equivalence class counts and generating a table of counts and sizes, and items in the equivalence classes and producing some statistics about the results.

**The Approach**

I chose to implement a naïve pseudoalignment algorithm to tackle this problem. Recognizing that working on this project bit by bit would mean that I would need to re-import the data sets multiple times, I chose to write this project in a Jupyter Notebook file. The packages that I needed to import were gzip to open the fasta.gz file, and SeqIO for ease of parsing the FASTA files. I also imported a package called tabulate to make my output file look pretty and matplot to help with building plots.

I designed my functions so that running them in consecutive order would allow them to build off each other, allowing me to easily follow my own code and identify where my potential bugs would be.

When parsing the transcriptome in my first function, I chose to store the isoforms in a dictionary with the key being the gene name and the value being the sequence. This was so that when building the hash tables, I could easily access the isoform belonging to an equivalence class and its corresponding sequence.

To build the index (i.e., another dictionary) of k-mers, I iterate through every transcript and search through each k-mer in the transcript’s sequence, add the k-mer as a unique hash if it is not there already, and add the current transcript to this hash. I choose to store my reads in a list since the ID of each read was too difficult to parse without the SeqIO package. When storing these reads, I knew that I could later access the sequence using the Seq.seq function.

I then create a function to align one read so that I can reuse it when I align everything else. In this function, I take in a read, iterate through the k-mers in the read, and access the equivalence class at that particular k-mer key in the hash table. As I iterate through the k-mers and access the equivalence classes, I take the intersection of the overall equivalence class and the current equivalence class being accessed, so by the end of my iteration, I will have the overall intersection of all equivalence classes of all the k-mers we have traversed in this particular read.

Finally, this last function ties every other function together. It takes in the k-mer hash table that we initialized earlier, all of the reads of the FASTA file, and the same value of k that was used to build our hash table, and pseudoaligns all the reads. For every read, I find its overall equivalence class and check if our equivalence class count vector contains this equivalence class. If it does, I add one to the count, and if not, I add this equivalence class to our vector with a count of one. In a special circumstance, if I find that a read has an empty set as its equivalence class, I attempt to pseudoalign its reverse complement. Once I have iterated through all the reads, I return the vector of equivalence class counts (Table 1).

**Results**

When running my pseudoalignment algorithm with k = 100, which is the length of the read, I find that out of the 1282526 reads, I was able to find equivalence classes for all but 231720 reads. This is a 1050806/1282526 = 0.819 pseudoalignment rate, which is a very successful procedure. After pseudoalignment, I find 10184 unique equivalence classes, the largest being of size 54 different isoforms. For k = 100, my total run-time was around two minutes, which was exceptionally fast and I was very impressed with, but when I tried other values of k, my run-time increases significantly.

I thought it would be interesting to see the most common sizes of equivalence classes all mapped up together, and how many different reads mapped to an equivalence class of that size. We can see this in Figure 1, and we find a histogram that is skewed right, which makes sense because it is harder to map a read to an equivalence class that contains a lot of isoforms. Some noticeable numbers are that 231720 different reads map to an empty equivalence class, 205885 reads map to an equivalence class of one isoform, and 220662 map to an equivalence class of two isoforms. There is at least one read that maps to an equivalence class of each size all the way until a class of size 54, which surprisingly contained 57 different reads, which I noticed was quite a large amount given the likelihood.

Along those lines, I also thought it would be interesting to look at how many different equivalence classes of each size there were (Figure 2). I again saw a histogram which was skewed right, with smaller equivalence classes being significantly more frequent than larger equivalence classes.

**Issues and Problems and Potential Improvements**

Much of the issues that arose during this problem was related to my inexperience with python. Having only taken CS31 and CS32, which both used C++, I spent much of this time getting familiar with the VSCode environment and learning how to use python and all of the possible packages that I could work with. A lot of my errors came from not keeping track of my variable types and trying to access things that I couldn’t. I had many errors like “int is not iterable” or “not callable.” In addition, this was also the first time that I ever had to translating theory, which we learned in class, into code for a project of this size. For example, I was stuck and confused for the longest time about how to read and interpret the FASTA files, which required a lot of googling. I read the .gz file wrong the first time, and I wasn’t able to extract any sequences, which lead me very confused as how I was supposed to align reads with no sequences.

Despite these errors, I found it extremely rewarding to this project as it forced me to transition into using a new language. Before, I was very stubborn with sticking to C++ since it was what I was comfortable with. I also got to put into practice efficiency, making sure that my code was readable and interpretable and fast, so that when I was debugging, I wouldn’t have to wait an insanely long time for compilation. Aside from being thrown into writing a project into a new language, I think I transitioned very well into writing a whole project in python, and most of the time, my project ran smoothly. I was able to follow the lecture slides and implement it into code fairly straightforwardly, with only minor errors like syntax, out of bounds, or iterating through the wrong list.

Future improvements to this project would be to implement it in the form of a de Bruijn graph algorithm to speed up the run time with skips. Another improvement I could make, this time to my naïve algorithm in particular, would be to reduce the amount of variable type conversions I had to make. I often went from dictionary to list to string and back and forth to each other, and I feel like this could have saved me a little bit more of run time. It also means that I probably could have done multiple operations at once in less for loops, since I can modify the same data type in each iteration. However, making the conversions was the easiest way for me to follow the theory behind pseudoalignment upon first implementation, so I am satisfied with being able to translate class content into a practical program.

**Figure 1. Size vs Number of Reads Mapping to an Equivalence Class**

**Chart, histogram

Description automatically generated**

**Figure 2. Histogram of Number of Items in an Equivalence Class**

**Histogram

Description automatically generated**

**Table 1. The First Rows of My Output Data File**

**Graphical user interface, text

Description automatically generated**