CS MS Capstone Project: Analyzing Tradeoffs Between RNA-Sequencing Pseudoalignment Implementations

Bradley Mont
UID: 804-993-030
University of California, Los Angeles

Abstract

In this project, I design a pseudoalignment program that takes in a transcriptome and RNA-sequencing reads and partitions the reads into equivalence classes based on which isoforms they pseudoalign to. I implement three different pseudoalignment implementations and compare their accuracy and efficiency. Finally, I examine basic statistics about the equivalence classes generated by my pseudoalignment procedure.

1. Introduction

Understanding the genetic composition of one's cells is crucial in several areas such as disease detection, and techniques such as RNA-sequencing exist to recover one's genetic composition. Specifically, RNA-sequencing is a process used to gain insight into one's transcriptome, which is defined as the collection of RNA transcripts present in an organism.

When performing RNA-sequencing, a common goal is quantification: given a gene's transcriptome and some read fragments obtained through RNA-sequencing, we seek to find the distribution of how well-represented each transcript is in terms of reads. This problem can be divided into alignment and counting, and we focus on the alignment portion.

Specifically, we seek to pseudoalign each read, where our main concern is the set of transcripts that a read could have possibly come from, but not its specific position within each transcript. This gives much faster performance compared to alignment since the exact positions do not need to be found.

Ideally, each read would map to exactly one transcript, but in reality, a majority of reads will map to many transcripts. Therefore, in pseudoalignment, we divide reads into equivalence classes. Two reads belong to the same equivalence class if they pseudoalign with the same set of transcripts. In theory, there could be an equivalence class for every combination of including or not including each transcript; however, in practice, it is found that the observed number of equivalence classes is under one million, which is a very reasonable amount of space.

To reiterate, since our goal is quantifying the expressiveness of each transcript, simply computing equivalence classes for each read suffices. We can then examine the relative abundances of each equivalence class to obtain the expressiveness of each transcript and equivalence class of transcripts.

2. Programming Environment and Input Parameters

I chose to implement the algorithm using a Jupyter Notebook because Python provides simple syntax and extensive library support for the computations necessary for pseudoalignment. Additionally, Jupyter Notebook allows me to execute portions of the code at a time, which allowed me to avoid re-executing computationally expensive portions of the code when I made changes that didn't affect those portions.

2.1. Parsing Input Files and Setting Program Parameters

As input, the program takes an integer `k` representing the k-mer size to use in the algorithm, a string `transcript_data_file_path` representing the file path for the transcriptome data in fasta format, and a string `reads_data_file_path` representing the file path for the RNA-seq reads data in fasta format. There are two more booleans that can be set for utilizing the reverse complement optimization and for outputting the results to CSV, but those will be explained in later sections. These values can be changed at any time by modifying their corresponding variables at the beginning of the code.

I utilized Python's `Bio.SeqIO` module to read in fasta files and create `SeqRecord` objects for each read and transcript.

3. Pseudoalignment Implementation 1: Hash Table

I created a hash table-based index for the first pseudoalignment implementation. In this approach, the index is a hash table that maps k-mers to a set containing all the isoforms that contain that k-mer. Next, I pseudoalign a read by segmenting it into k-mers and returning the intersection of all its k-mers' equivalence classes (obtained using the hash table index) as its overall equivalence class. This approach offers the greatest simplicity of all approaches.

3.1. Building the Hash Table Index from Transcriptome Data

I represented the hash table index as a Python `dictionary` named `kmer_to_matching_isoforms`, which

maps each k-mer to a `set` containing all the transcripts that contain that k-mer. This required looping through each transcript's k-mers, which seemed to be a large contributor to the program's run time.

3.2. Pseudoaligning Each Read to its Equivalence Class

For performing pseudoalignment on each read, I created a function `hash_table_pseudoalignment` that takes in a read as input and outputs its equivalence class. Specifically, I iterated through each k-mer in the read and found its matching isoforms using `kmer_to_matching_isoforms`. I then took the set intersection of all these equivalence classes of matching isoforms to get the final equivalence class for the read.

Lastly, for reads that initially had empty equivalence classes, there is an option to attempt to align its reverse complement, which will happen if the `attempt_reverse_complement` variable is set to `True` at the beginning of the program. This will be explained in greater depth during the optimizations portion.

3.3. Counting the Number of Reads in Each Equivalence Class

Next, I populate the Python `dictionary equivalence_class_to_num_reads_hash_table`, which maps each equivalence class to the number of reads in that equivalence class. I simply iterate through each read, call `hash_table_pseudoalignment` with that read as input, and then increment the value for the equivalence class outputted by the function.

4. Pseudoalignment Implementation 2: Colored de Bruijn Graph

For my second pseudoalignment implementation, I represented the index using a Colored de Bruijn graph. First off, a de Bruijn graph for an isoform is a graph where the nodes are k-mers, and each k-mer's node has an edge pointing to all the k-mers that appear directly after it in the isoform. As a trivial example, a transcript of TGA with a k of 2 would have a node with value TG pointing to a node with value GA.

For our index, we create a large de Bruijn graph containing the entire transcriptome, and we "color" each node with a set of all the transcripts that its k-mer appears in. Next, to align a read, we follow its k-mers in order through the Colored de Bruijn graph, returning the intersection of all the equivalence classes of the k-mers in the path as the read's overall equivalence class.

4.1. Building the Colored de Bruijn Graph Index from Transcriptome Data

I represented the Colored de Bruijn Graph Index as a Python `dictionary` named `de_bruijn_graph`, which maps each k-mer to a `Kmer_Info` object containing its equivalence class and k-mers that its node is pointing to. This required iterating through every transcript's k-mers, keeping track of a `curr_kmer` and `next_kmer`, ensuring that the `curr_kmer` points to the `next_kmer`, and updating these variables accordingly as I iterate through each transcript.

4.2. Pseudoaligning Each Read to its Equivalence Class

For performing pseudoalignment on each read, I created a function `de_bruijn_pseudoalignment` that takes in a de Bruijn graph and a read as input and outputs its equivalence class. In order to find a read's equivalence class, I obtained the k-mer node for its first k-mer using `de_bruijn_graph`. Then, I iteratively checked to see if the current k-mer node pointed to any nodes that matched the next k-mer in the read. I then took the set intersection of all the nodes' equivalence classes in this path through the de Bruijn graph. I also utilized the reverse complement optimization similarly to the first implementation which will be explained later on.

4.3. Counting the Number of Reads in Each Equivalence Class

Similarly to the hash table implementation, I populate a Python dictionary `equivalence_class_to_num_reads_de_bruijn` using the same logic, but instead using `de_bruijn_pseudoalignment` to pseudoalign each read.

5. Pseudoalignment Implementation 3: Colored de Bruijn Graph with "Skipping" Optimization

For my final pseudoalignment implementation, I augmented the Colored de Bruijn graph from the second implementation with a "skipping" distance. This optimization utilizes a key observation: if there exists a non-branching path in the Colored de Bruijn graph that does not change equivalence classes, we can "skip" to the end of that non-branching path. The only k-mers relevant to the equivalence class of a read are the ones at potential branching points in the graph, in addition to the first k-mer to initially index into the graph.

This optimization changes pseudoalignment from being a function of all k-mers in a read to being a function of all "branch points" in the Colored de Bruijn graph for that read.

5.1. Augmenting the Colored de Bruijn Graph Index with Skipping Distances

I created the augmented Colored de Bruijn Graph index, `optimized_de_bruijn_graph`, by first creating a deep copy of the `de_bruijn_graph`. I then iterated through each k-mer, searching for the longest possible non-branching path. Once I found this non-branching path, I updated the `branch_distance` and `kmers_pointing_to` fields for all nodes in the non-branching path. I calculated the correct branching distance for each k-mer in the path by keeping track of its offset from the beginning of the path in `kmers_in_path_to_offset`. I ensured to add all

nodes in the path to a `seen_kmers set` in order to ensure that I do not traverse the same path multiple times.

5.2. Pseudoaligning Each Read to its Equivalence Class

For performing pseudoalignment on each read, I reused the `de_bruijn_pseudoalignment` function, but instead passed in `optimized_de_bruijn_graph` as input. All other logic is identical to the unoptimized Colored de Bruijn graph version.

5.3. Counting the Number of Reads in Each Equivalence Class

For counting the number of reads in each equivalence class, I use the same logic as the previous implementations to populate the dictionary `equivalence_class_to_num_reads_skipping`.

6. Implementation: Preparing Data for Analysis

After completing the pseudoalignment process using the three different implementations, I organized the data and produced graphs summarizing the results.

6.1. Storing the Result in pandas DataFrames and (Optionally) Outputting to CSV

After computing the necessary equivalence class data and storing it in the `equivalence_class_to_num_reads` dictionaries, I used them to populate a `pandas` DataFrame for each implementation containing the same information. The DataFrames each contain 3 columns: `'# of reads'` representing the number of reads in the equivalence class, `'# of items in equivalence class'` representing the number of isoforms in the equivalence class, and `'isoforms in equivalence class'` which gives the names of the isoforms in the equivalence class.

Additionally, if the `output_to_csv` variable at the beginning of the code is set to `True`, the program outputs these DataFrames into CSV files named `hash_table_output.csv`, `de_bruijn_output.csv`, and `skipping_de_bruijn_output.csv`. The program also sorts the rows by `'# of reads'` descending in order to easily view the most expressed equivalence classes.

6.2. Providing Basic Statistics about the Equivalence Classes

Finally, I used Python's `pandas` library to output relevant statistics about the DataFrames. These will be described in more detail in later sections.

In addition, I utilized Python's `matplotlib.pyplot` library to output histograms about the number of reads and number of isoforms in each equivalence class and to output bar graphs about the time efficiency of each pseudoalignment implementation. These will also be described in greater detail later on.

7. Comparing Accuracy of Three Pseudoalignment Implementations

After viewing the CSV outputs produced by each pseudoalignment implementation, it is evident that they reached extremely similar conclusions with slight variations. The only specific statistic used in this section will be the number of reads that did not pseudoalign to a single isoform. This can be used as a rough measurement of the "strictness" of matching required by each pseudoalignment implementation.

First off, the hash table implementation theoretically loses accuracy because it simply takes the set intersection of all its k-mers' equivalence classes: this fails to capture the order of the k-mers. A read will pseudoalign to a transcript as long as all its k-mers appear in the transcript, but they are not required to appear contiguously in order to match.

The Colored de Bruijn graph implementation addresses this issue and improves accuracy by requiring the k-mers to appear in order. A read will only pseudoalign to a transcript if the entire read matches a path in the transcript's Colored de Bruijn graph. This means that matching k-mers but out of order is not sufficient for a match in this implementation.

Lastly, adding the "skipping" optimization to the Colored de Bruijn graph implementation greatly improves performance, but it does technically relax the criteria for a match and lower the strictness of a match. Since it only checks the crucial k-mers at branching points in the de Bruijn graph, there is a possibility of a read matching a transcript at the branch point k-mers, but not matching at some of the intermediate k-mers between branch points. This is empirically supported by the fact that the "skipping" optimization implementation contained the smallest amount of reads that did not align with a single transcript, since its matching criteria is arguably the weakest of the three.

From this analysis, I concluded that the unoptimized de Bruijn graph implementation is the most accurate, as it enforces the k-mers to appear in the same order in the read and transcript, and it checks all k-mers, including those not at branching points in the Colored de Bruijn graph. Therefore, for simplicity, I will only use the results outputted by the unoptimized de Bruijn graph implementation when providing statistics about the equivalence class data.

8. Comparing Performance of Three Pseudoalignment Implementations

In order to compare the performance of each of the pseudoalignment implementations, I measured the execution time for each implementation to (1) create the index and (2) pseudoalign the provided reads using the index. For context, I created the index using 2,075 transcripts of variable length, and I pseudoaligned 1,282,526 reads of length 100 using the index.

8.1. Comparing the Time Taken by Each Implementation to Build the Index

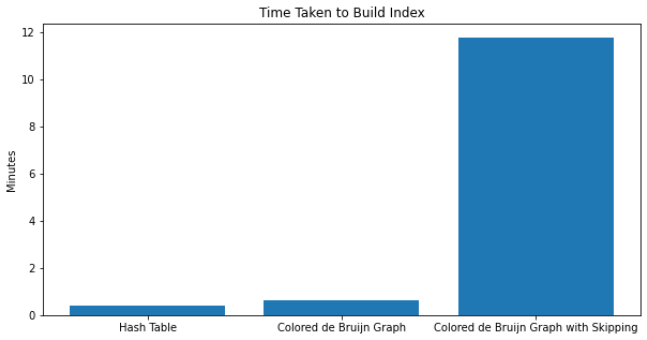

Figure 1: Time Taken to Build Each Index

As seen in Figure 1, the hash table and Colored de Bruijn graph implementations were able to build the index relatively quickly and equally, with execution times of 0.399 minutes and 0.633 minutes respectfully. On the other hand, the time taken to build the optimized Colored de Bruijn graph index, which was calculated by adding the time taken to add the "skipping" annotations to the time taken to build the normal Colored de Bruijn graph index, was 11.141 minutes in total, significantly longer than the previous two implementations. Clearly, the first two implementations are relatively fast and similar, while the third implementation takes considerably longer to build its index.

8.2. Comparing the Time Taken by Each Implementation to Pseudoalign Reads

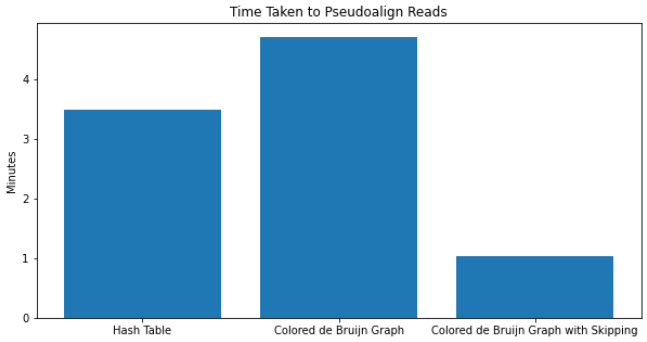

Figure 2: Time Taken to Pseudoalign Reads by Each Implementation

As seen in Figure 2, the hash table implementation took 3.492 minutes to pseudoalign the reads, and the Colored de Bruijn graph implementation took 4.707 minutes, making it the slowest implementation. This is to be expected as iterating through the graph and checking its nodes' next valid k-mers is more computationally expensive than simply hashing k-mer values into the hash table index.

Most importantly, the Colored de Bruijn graph with the "skipping" optimization was the most efficient by far, taking only 1.031 minutes to pseudoalign the 1,282,526 reads.

8.3. Performance Conclusions

Overall, simply in terms of performance (measured by execution time), the Colored be Bruijn graph implementation had the worst performance, as it took the longest to pseudoalign and took longer than the hash table implementation to build its index. The hash table implementation offered decent performance, with the benefit building its index the fastest. For the purpose of pseudoaligning an infinite amount of reads, the Colored be Bruijn graph with the "skipping" optimization performed the best. Although building the index was quite slow and has much room for optimization, the index only needs to be built once, so this is simply a larger precomputation. Once the index is built, this implementation pseudoaligns much faster than the other two implementations; therefore, as the number of reads scales, the optimized graph implementation will scale most optimally.

9. Optimizations Utilized in all Implementations

First off, due to the method in which RNA-sequencing reads are generated, it is sometimes necessary to first take the reverse complement of a read before attempting to pseudoalign it. Therefore, for reads that initially had empty equivalence classes, I added an option to attempt to align its reverse complement, which will only happen if the `attempt_reverse_complement` variable is set to `True` at the beginning of the program.

Additionally, I added another small optimization during the alignment process. If a read's equivalence class becomes empty at any point during the pseudoalignment process, the program halts the computation there and returns the empty set as its equivalence class. This preserves accuracy since the intersection of any set with the empty set will still be the empty set, and it saves computation time as well.

10. Pseudoalignment Equivalence Class Statistics

As previously stated, since the Colored de Bruijn graph version has the most strict matching criteria and is theoretically the most accurate, the data obtained through that implementation will be used for all equivalence class statistics.

10.1. Benefits of the Reverse Complement

One important statistic to note is that including the reverse complement optimization resulted in an additional 262,300 reads being aligned to non-empty equivalence classes. This demonstrates that the reverse complement had a significant impact on successfully aligning more reads.

10.2. Basic Equivalence Class Statistics

	# of reads	# of items in equivalence class
count	2744.000000	2744.000000
mean	467.392857	4.950802
std	14128.004375	4.521684
min	1.000000	0.000000
25%	2.000000	2.000000
50%	9.000000	4.000000
75%	49.000000	7.000000
max	737392.000000	54.000000

Table 1: Basic Equivalence Class Statistics

Table 1 displays the basic statistics for the number of reads and number of isoforms in each equivalence class. On average, each equivalence class has about 467 reads mapping to it and contains about 5 isoforms. However, as seen by the high standard deviation and maximum value for the number of reads, there are a small amount of equivalence classes pulling the mean amount of reads much higher than the median amount of reads, 9.

10.3. Most Highly-Expressed Equivalence Classes

# of reads	# of items in equivalence class	isoforms in equivalence class
737392	0	
50420	2	ENST00000329251, ENST00000496634
12500	1	ENST00000536684
11681	1	ENST00000393067
10042	5	ENST00000389939, ENST00000345732, ENST00000674...
10039	7	ENST00000422447, ENST00000545215, ENST00000379...
9344	1	ENST00000260197
9020	1	ENST00000449131
8939	6	ENST00000532829, ENST00000534180, ENST00000526...
8168	3	ENST00000530797, ENST00000321153, ENST00000530398

Table 2: Most Highly-Expressed Equivalence Classes

By an extremely large margin, the equivalence class with the most reads mapping to it is the empty equivalence class, containing 737,392 reads. This is expected as my pseudo-alignment algorithm is not at all tolerant to errors and requires exact k-mer matches. The most expressed non-empty equivalence class is the one with isoforms ENST00000329251 and ENST00000496634, with 50,420 reads mapping to it.

10.4. Histograms of Read Counts

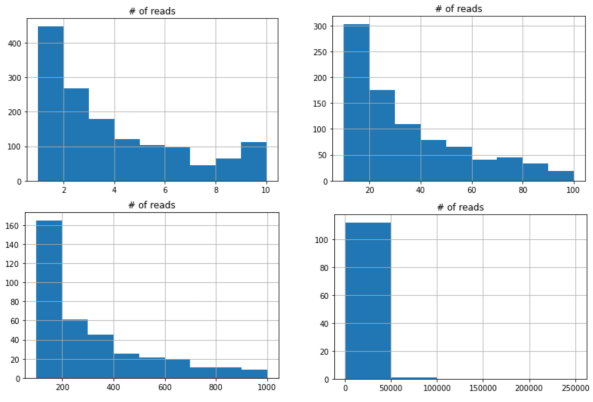

Figure 3: Histograms of Equivalence Class Read Counts

Figure 3 contains 4 histograms which display the amount of equivalence classes that fall within given ranges of read counts. I created multiple histograms because the amount of equivalence classes with single digit reads is orders of magnitude larger than those with a larger amount of reads, so a single histogram would not be able to clearly display the range of values.

A large amount of equivalence classes have a single digit amount of reads mapping to them. Specifically, 1 read is the highest with over 450 equivalence classes. This frequency steadily decreases to under 100 equivalence classes. From there, we measure the frequencies in terms of ranges of 10s, with about 300 equivalence classes having between 10 and 20 reads, decreasing to about 25 for between 90 and 100 reads. The last two histograms show the equivalence class frequencies continuing to decrease as the amount of reads increases.

10.5. Histograms of Isoform Counts

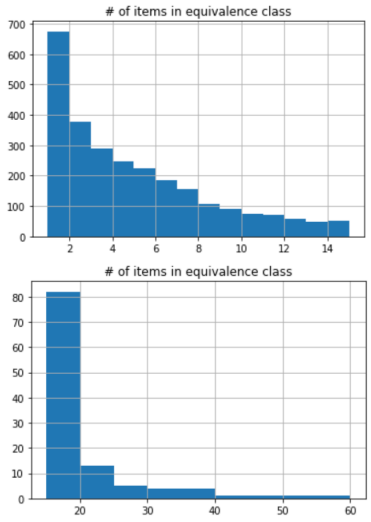

Figure 4: Histograms of Equivalence Class Isoform Counts

Figure 4 contains histograms which display the amount of equivalence classes that fall within given ranges of isoform counts. I created multiple histograms for the same reason as the read count histograms.

The first histogram shows how well over 650 equivalence classes have a single isoform in them. This number steadily decreases, and we can see that under 75 equivalence classes have 15 isoforms in them.

The second histogram demonstrates at a larger scale that the trend of number of equivalence classes going down as the number of isoforms increases continues. We can see that over 80 equivalence classes have between 15 and 20 isoforms in them, and this steadily decreases until the final range, which is 50 to 60 isoforms with under 5 equivalence classes.

11. Future Work

This project can be extended in several different areas. First off, since errors are an extremely common occurrence in RNA-sequencing, experimenting with allowing error thresholds when pseudoaligning reads could prove to be useful. My implementations required an exact match of k-mers, which led to a large amount of reads not aligning to any transcripts. Adding more leniency in terms of error thresholds would lead to more reads aligning to transcripts.

Additionally, my implementation of the "skipping" optimization for the Colored de Bruijn graph was a major bottleneck in terms of performance. Although this is a pre-computation that only needs to be executed once, cutting down its execution time would be a useful goal for the future.

Finally, there are plenty of other implementations and optimizations for pseudoalignment that I could continue to implement in order to optimize the performance and accuracy of my pseudoalignment program.

12. Conclusion

Overall, this project gave me a much more in-depth and practical understanding of pseudoalignment and RNA-sequencing. The hash table-based implementation provided excellent simplicity and readability, but it lacked in accuracy due to not requiring k-mers to appear in order. Its simple index could be built very quickly, and it pseudoaligned the reads somewhat efficiently. The Colored de Bruijn graph implementation had a stricter matching criterion, leading to greater accuracy but slower execution times than the hash table implementation. Lastly, the optimized Colored de Bruijn graph implementation struggled with its precomputation time to build its index, but it is clearly the most efficient algorithm for performing pseudoalignment. Furthermore, the histogram data showed that the number of equivalence classes decreased as both the number of reads and number of isoforms increased.

References

[1] *Fast and accurate pseudoalignment lecture slides*: https://bruinlearn.ucla.edu/courses/79602/files/folder/slides?preview=8089682

[2] *Near-optimal probabilistic RNA-seq quantification*: https://www.nature.com/articles/nbt.3519