Pseudoalignment Implementation

William Randall

Intro:

For my final project in CS 121, I chose to implement a pseudo alignment algorithm. The input would be gene annotation and RNA seq data. From this, we were able to find the vector of equivalence class counts. To get from a gene annotation and RNA seq data to vectors of equivalence class counts, I first used the gene annotation to construct a de Bruijn graph by iterating over the kmers in the gene annotation. After constructing the de Bruijn graph I was able to take each read and find if it was a match for any equivalence class. In the de Bruijn graph, I annotated it with the equivalence class so that I would be able to walk along the de Bruijn graph using each RNA read and from there I would be able to determine if the RNA read matched one or more equivalence classes. I then stored this count and continued with the next RNA read.

My Implementations:

There are two major steps in Pseudoalignment. First, we use the gene annotation (in this case “chr11_transcriptome.fasta”) to create the data structure which we will use later to align the RNA seq data. The data structure I chose to implement was the de Bruijn graph as opposed to the hash table method. I chose to use the de Bruijn graph method because it seemed like the hash table method would have longer runtimes because mapping the RNA seq data to the hash table would take more time than just traversing the de Bruijn graph one kmer/node at a time. My implementation of the de Bruijn graph does not collapse nodes, but even though this is the case, it works much faster in relation to the hash table method.

Creating the de Bruijn Graph:

```
kmers = {} // hash table
debruijn graph = Graph() // graph
for each isoform in the gene annotation:
     for each kmer in each isoform:
          if current kmer is in de bruijn graph:
               add isoform to kmers hashtable with a key of
                 … current kmer
          else:
               add kmer as a node to the de Bruijn graph
```

```
                append current isoform to kmers hashtable with a
                  … key of current kmer
            link the current node to the previous node
```

After creating the de Bruijn Graph, we can now iterate over each read in the RNA seq data and match it to its equivalence classes. We can do this by traversing the de Bruijn graph and keeping track of the intersection of the seen equivalence classes. We can do this by jumping to the point on the graph that corresponds to the first kmer in the read, and from there we walk along the directed de Bruijn graph until we either reach the end of the read or if we reach the end of all the equivalence classes we matched to so far.

Pseudoaligning the RNA seq reads:

```
equivalence dict = {} // hash table
for each read in the rna seq data:
    previous node = first kmer
    current isoforms = kmers[first kmer] // gives equivalence
    // … classes of the first kmer
    if current kmer not in debruijn graph:
        equivalence dict[null] += 1 // increase num items
    for each kmer in the read:
        if kmer is in the next nodes relative to the previous
          … node on the graph:
            current isoforms = intersection of current
              … isoforms and kmer[current kmer]
            previous kmer = current kmer
        else:
            current isoforms = {} // null
            break
    if current isoforms not null:
        if equivalence dict[current isoform] not null:
            equivalence dict[current isoforms] += 1
        else:
            equivalence dict[current isoform] = 1
    else:
        equivalence dict[null] += 1
```

After we create an equivalence dictionary that maps isoforms groups to counts of matches with RNA seq data we then sort this based on the count of matches. From here we then save this to the computer in a csv format.

Problems I faced:

I found it hard to quickly implement this solution because graphs are difficult to deal with in python because of the lack of pointers (blessing and a curse). I also found it hard to check my work because there is no real way to know if you are correct without creating a de Bruijn graph by hand.

Things I found interesting:

I found the speed difference between my implementation and my friends' implementations using the hash table was very interesting. It provided concrete evidence in the fact that the de Bruijn graph actually is helping speed up the computation of pseudo alignment.

Conclusion and improvements:

Overall, I think that this solution will only work quickly on files that are relatively medium size, like the one we were given, but if you increase the file size then you will need to use branch skipping. If I was to spend more time implementing this solution, I would implement the Kallisto algorithm of skipping to branch points instead of naively stepping one by one across the de Bruijn graph.