

CS 482: Computational Techniques in Biological Sequence Analysis Homework #2

ERIC HAORAN HUANG¹

¹e48huang@uwaterloo.ca, 20880126, e48huang

Abstract

This assignment was an exercise in phylogeny, alignment-free methods and genome assembly.

Setup

I used Python 3.10.12 on the student server machines in this run with external dependencies described in 'requirements.txt'. Run 'pip install requirements.txt' for proper setup.

Part 1: K-mer Composition

All code and examples are found under 'q1/'. This problem was tackled in a $O(nk)$ runtime where we had the following strategy:

1. Grab the sequence using the BioPython.
2. Calculate the k-mer frequency array by iterating over all k-substrings (running $n - k + 1$ times with the length of the sequence being n).
 - ◇ For each k-mer, we generate all possible k-mers, e.g.: 'NA' will produce 'AA', 'CA', 'GA', 'TA'.
 - ◇ We do this by iterating through each k-mer string one nucleotide at a time and keeping all possible k-mer prefixes. We extend each prefix by the possible next nucleotide base according to IUPAC notation. This process takes $O(nk)$ times because we have $O(n)$ possible k-mers which we spend $O(k)$ time reconstructing all possible k-mers.
 - ◇ Taking the total k-mers possible, weigh each possible string equally across each k-mer. Add this weight to a k-mer frequency array whose index is defined as mapping of the possible kmer string to its lexicographic index in the 4^k frequency matrix. This can be easily found by setting the weights of A to 0, C to 1, G to 2, T to 3 and then turning the string (alias of base 4) to base 10.
3. Return and print out to a file as needed.

Run the code with the following line: 'python kmer_comp.py -i <input_file> -k <kmer-length>' with optional flags of '-o <output_dir>' to output to the file under '<output_dir>/<input_file_name>_len_<kmer_length>_k-mers.txt' and '--debug' for more logging information.

I decided to use this method of averaging over the possible k-mers for ambiguous bases, i.e. if we have 'NA' we give 0.25 frequency weights to 'AA', 'CA', 'GA', 'TA'. This assumes that given an ambiguous base, that there is equal chance of any base that represents it to be the true base. This might not necessarily be true, but is the best way to utilize the heuristic of the ambiguous base given.

Part 2: deBruijn Graph Construction

All code and examples are found under 'q2/'. This problem was tackled in a $O(nk \log n)$ runtime with the following strategy, where n is the number of strings, k is the length of the strings:

1. Calculate the reverse complements of all the strings, taking $O(nk)$ time.
2. Calculate the edge list, taking $O(nk)$ time of the joint set of the given set and the reverse complements.

3. Return the sorted list, taking $O(k)$ time in comparison and needing $O(n \log n)$ comparisons, or done in $O(nk \log n)$ time.

Run the code with the following line: `python build_deBruijn.py -i <input_file>` with optional flags of `-o <output_dir>` to output to the file under `<output_dir>/<input_file_name>_deBruijn.txt` and `--debug` for more logging information.

Part 3: Alignment-free analysis of viral phylogenies

All code and examples are found under 'q3/'. This problem heavily utilized the existing libraries as follows:

1. Downloaded the viral sequences using BioPython's Entrez package. This resulted in '.gb' files which were then cached and parsed through using BioPython again.
2. Defined metrics as following:
 - ◊ Euclidean distance. This metric is already a distance, no changes needed, used the L2-norm.
 - ◊ Cosine similarity. This metric is not a distance and in fact, given that each component itself must be non-negative (given that frequency must be greater than or equal to 0), according to this Wikipedia article*, it is bounded by $[0, 1]$, where it being 0 means that it is completely dissimilar, and being 1 means that it is an exact match. So, we need to translate this to distance which is bounded by $[0, \infty)$. Therefore, we need to map the 1 to a 0 and the 0 to an infinity. A natural solution then is given d as our cosine similarity, to take $\frac{1}{d} - 1$ as this maps 0 to infinity, and also maps 1 to 0. I came up with this by noting that $\lim_{d \rightarrow 0} \frac{1}{d} = \infty$ but we need to shift it by 1 for $\frac{1}{d} - 1$ to be 0 at $d = 1$.
 - ◊ Pearson Correlation. Similarly, I noted that according to Wikipedia*, that the expression will always lie between -1 and 1 , with -1 being completely negatively correlated and 1 being completely positively correlated. This might look like if given two vectors, the AA frequency increasing from one sequence to another will cause the frequency of CC to decrease. So, a negative correlation means that they are more dissimilar, whereas a positive correlation means that they are more similar. Therefore, we need to map -1 to ∞ and 1 to 0. Then we have, given the Pearson correlation of ρ :

$$\lim_{\rho \rightarrow -1} \frac{1}{\rho + 1} = \infty, \frac{1}{\rho + 1}|_{\rho=1} = \frac{1}{2}$$

which matches the proper value for $\rho = -1$, however is wrong for $\rho = 1$. Then, we can do:

$$\begin{aligned} \frac{2}{\rho + 1} - 1 &= \frac{2 - \rho - 1}{\rho + 1} \\ &= \frac{1 - \rho}{1 + \rho} \end{aligned}$$

which is what we implemented.

3. Used each of these distances on the kmer frequency vector calculated using q1, which created a distance matrix.
4. Took this distance matrix and used the DistanceTreeConstructor module to build a UPGMA tree.

Run the code with the following line: `python build_newick_tree.py -i <input_file> -c <input_candidate_file>` with optional flags of `-o <output_dir>` to output to that directory and `--debug` for more logging information. There is also a `-k` flag to attempt this with different k-mer lengths.

Then, using this result we can answer the given questions.

Subtype Clustering

The subtypes are very well clustered in the initial tree as shown in the following figures representing the Newick tree, generated through IcyTree. We can see that for every single tree, we have that the subtypes are all grouped together before grouping with any other subgroup.

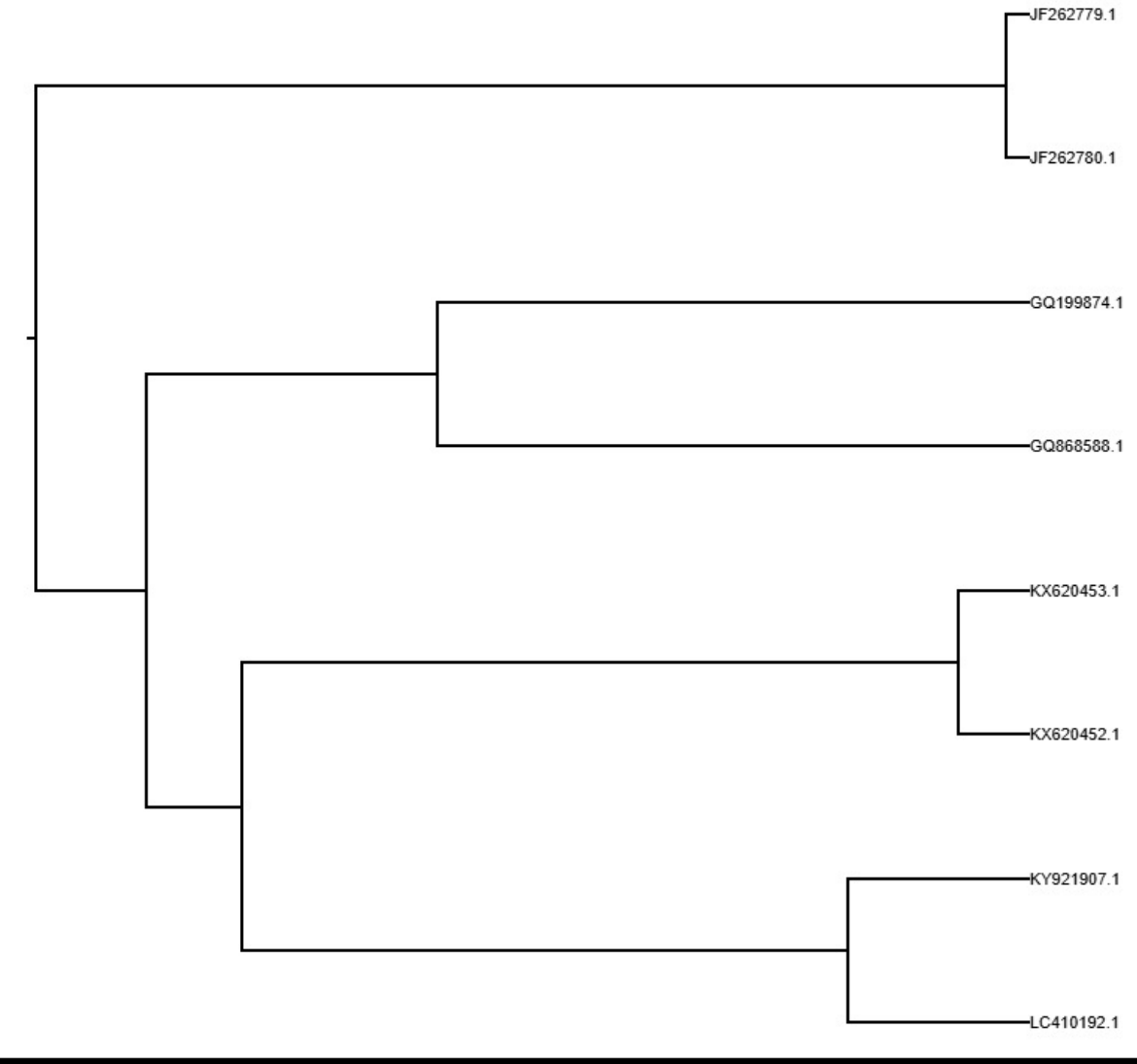


Figure 1: Euclidean Distance Newick Tree of the original subtypes

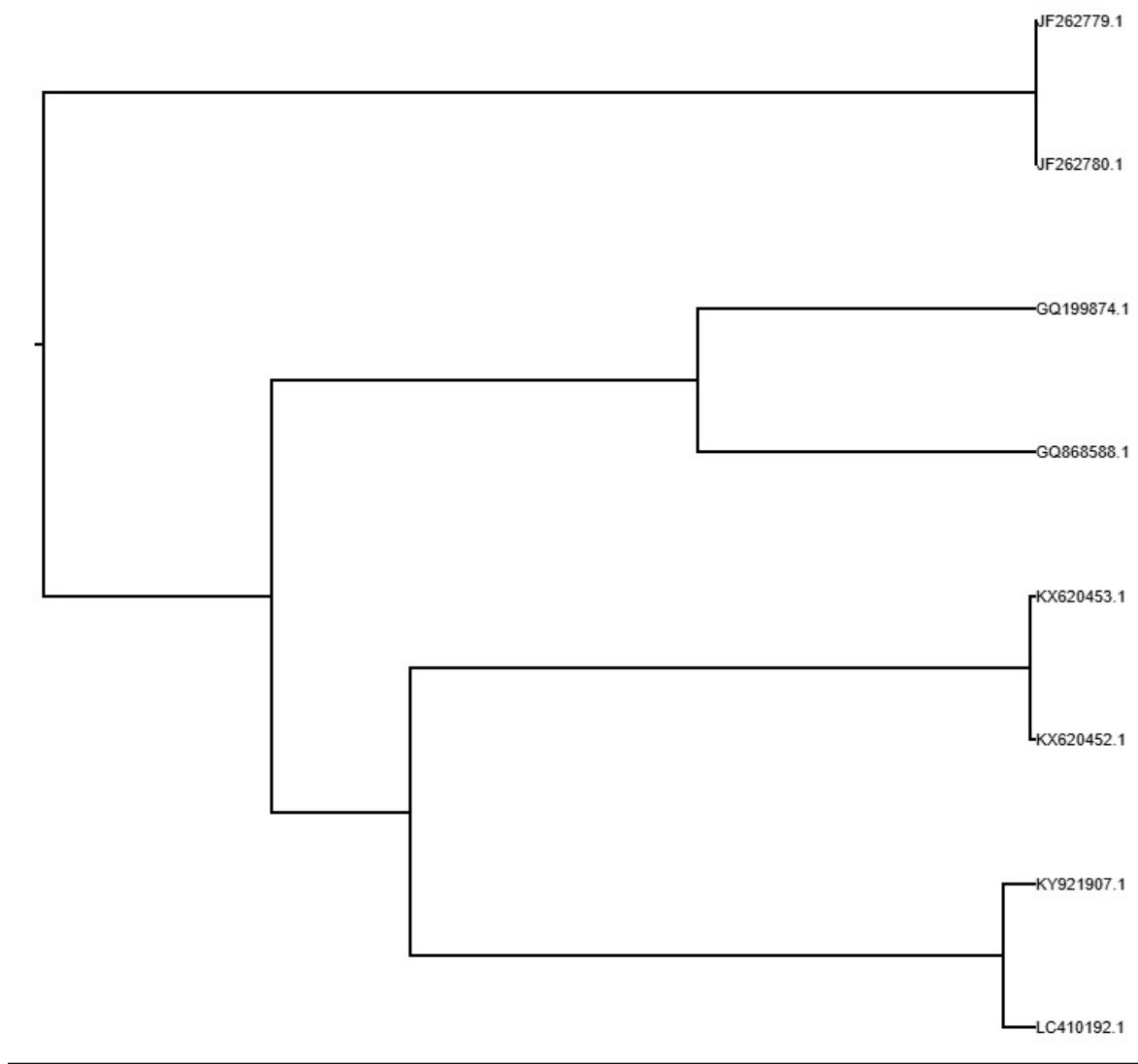


Figure 2: Cosine Similarity Newick Tree of the original subtypes

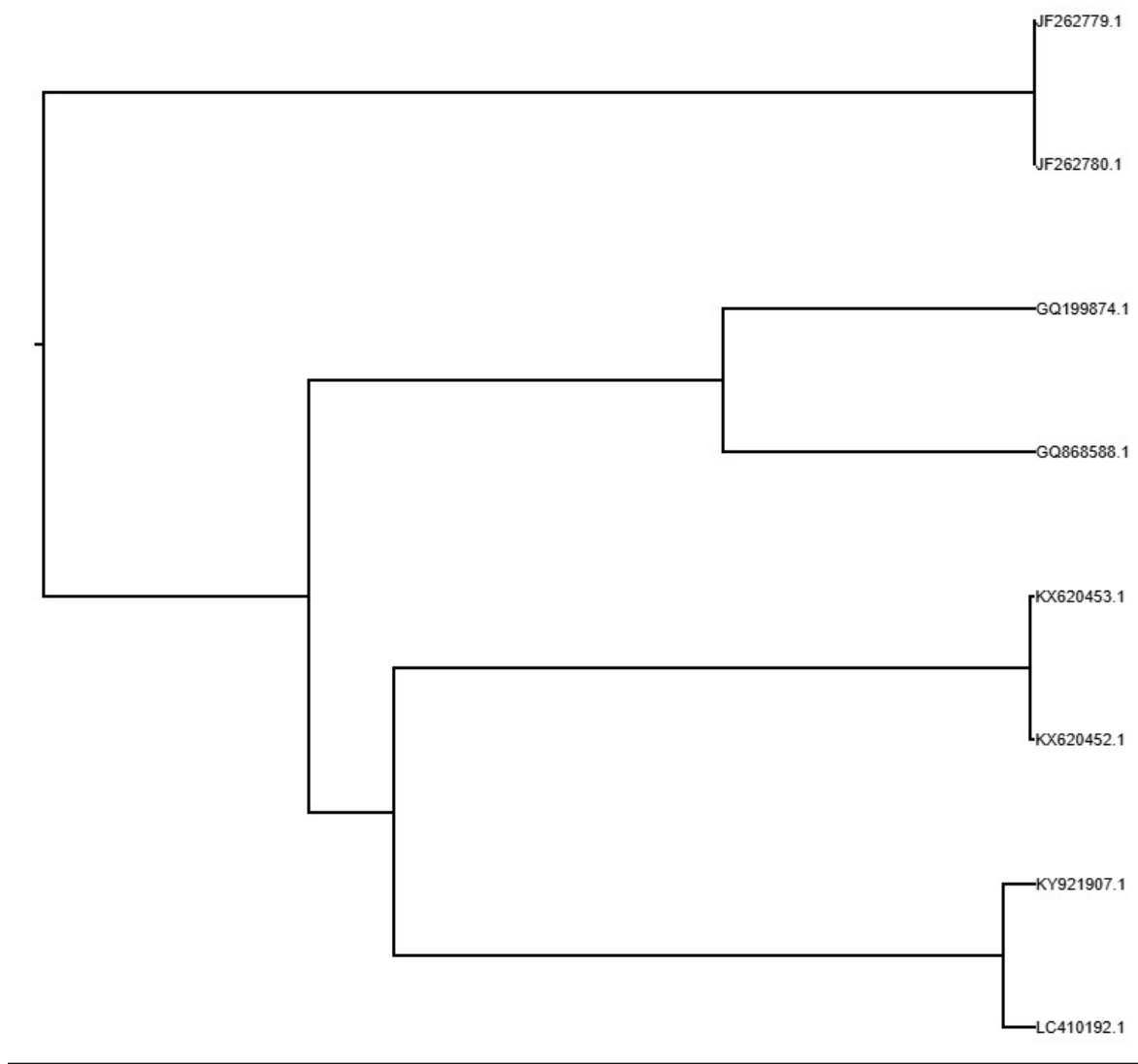


Figure 3: Pearson Correlation of Newick Tree of the original subtypes

Test Sequence

With the test sequence, we see a slight change in the tree, where we simply add sequence 9 in.

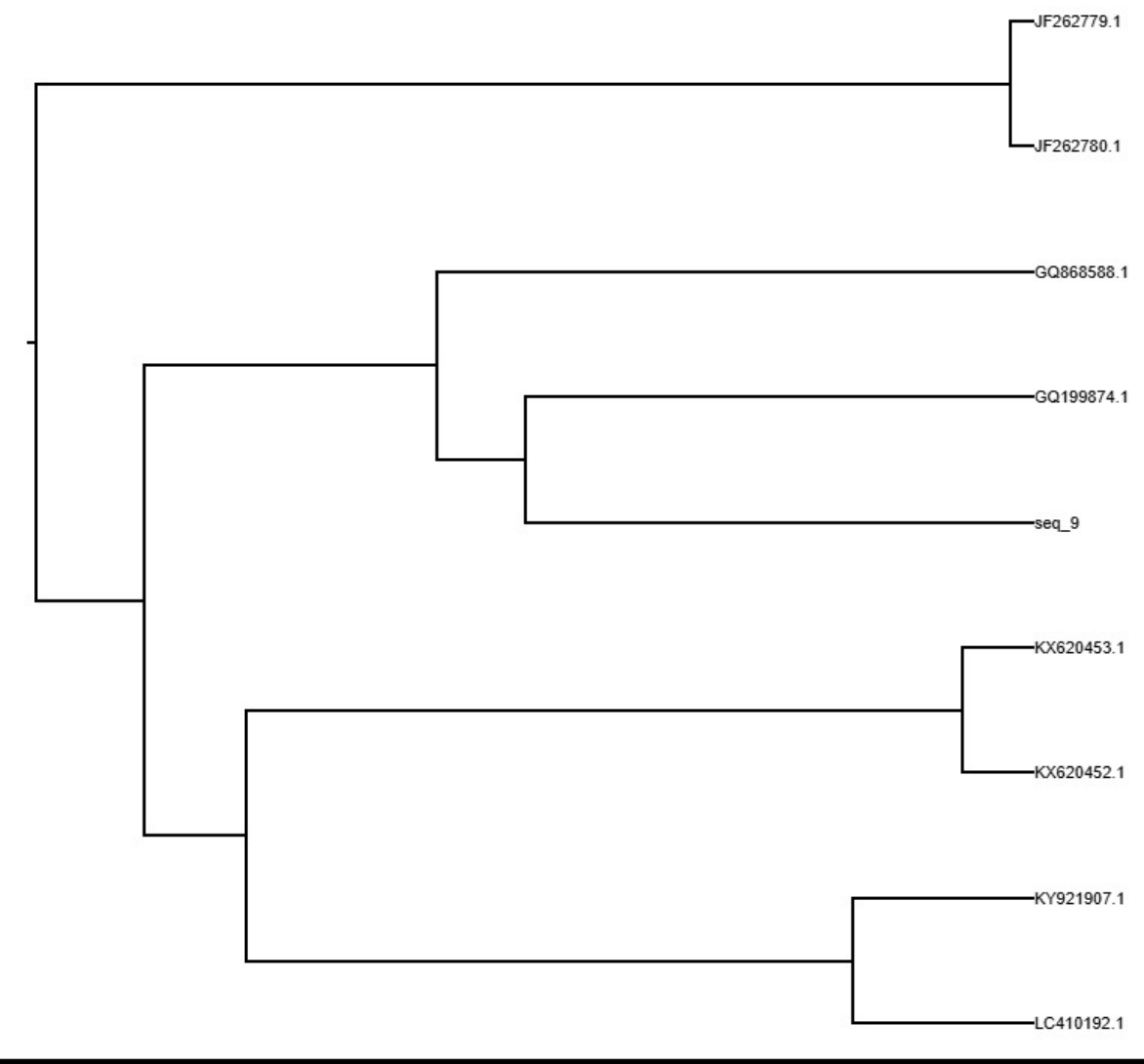


Figure 4: Euclidean Distance Newick Tree with the test sample

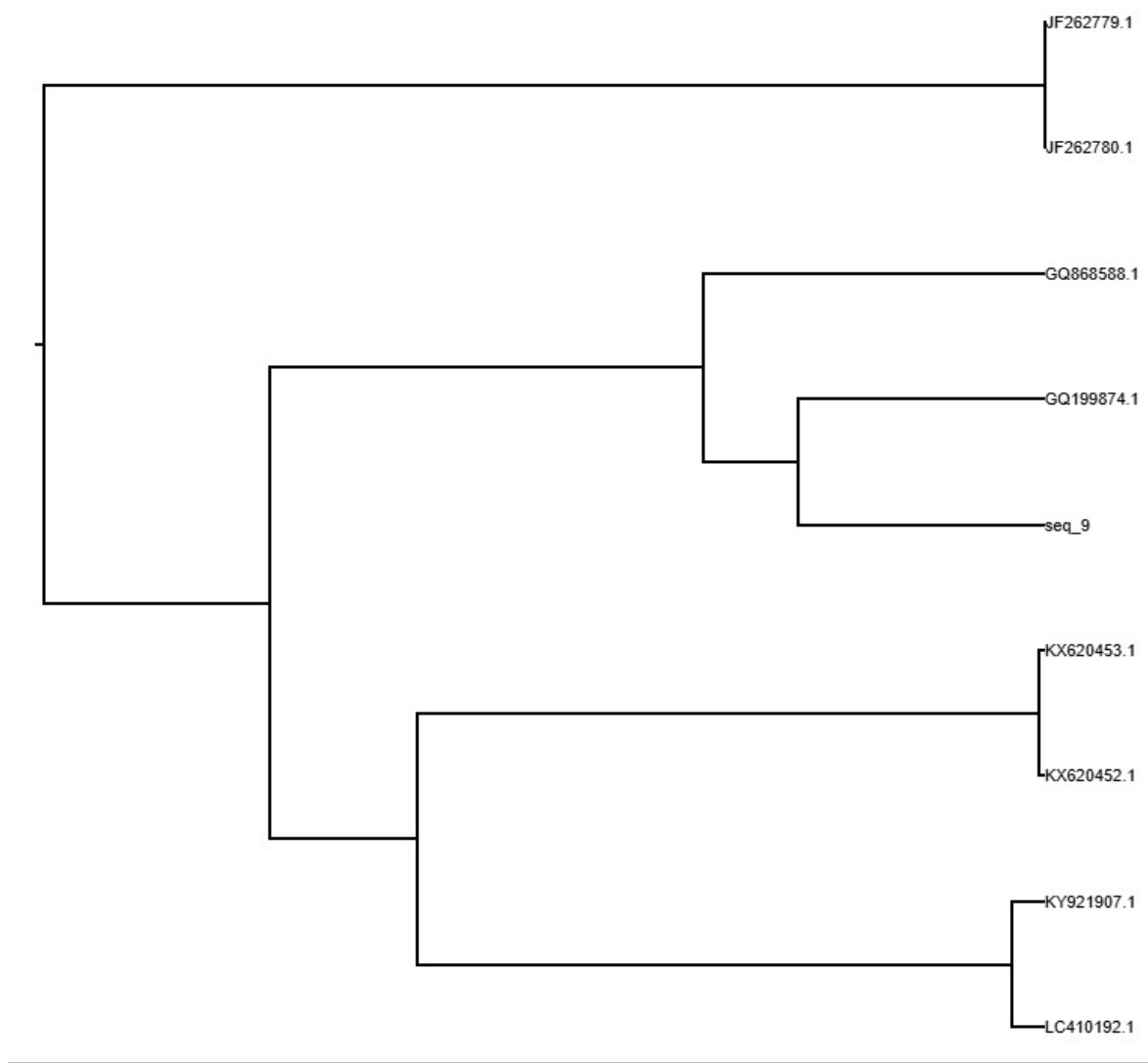


Figure 5: Cosine Similarity Newick Tree with the test sample

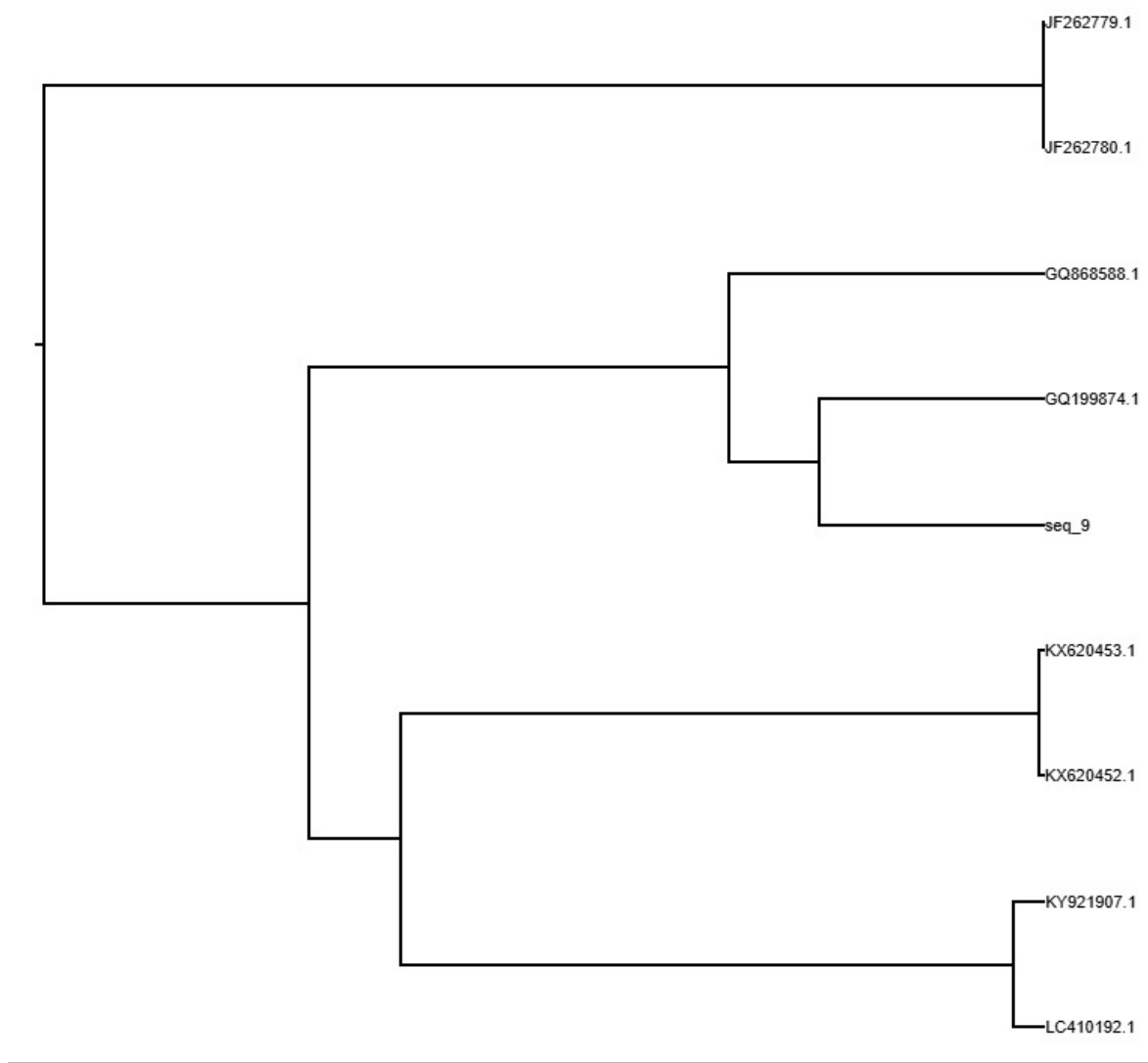


Figure 6: Pearson Correlation of Newick Tree with the test sample

Predicted subtype

In all cases, we have that our subtype sequence 9 belongs to subtype2, and is a sibling of GQ199874.1.

Appropriate Metric

In my opinion, I believe that cosine similarity to be the best metric as it is the most interpretable and grounded in what we want.

For Euclidean distance, a problem arises if we have two sequence frequencies of $[0, 1]$ and $[0, 2]$. If the direction is the same for the vectors, as it is here, then we know that the relative frequencies of the kmers are the same and therefore should not be penalized. However, for Euclidean distances, this is penalized.

Now, we can note that this is not penalized in both cosine similarity and Pearson correlation. For cosine similarity this is because the two vectors are in the exact same direction. For Pearson correlation this is because the correlation between the variables matches. In fact, one can show that my metrics I've used have the same derivative and grows at the same rate as each other. i.e. for any change in Pearson correlation distance metric, we would expect similar changes in the cosine similarity distance metric. In fact, looking at the previous figures, there is very little discrepancy between the two models. Therefore this boils down to how interpretable the two metrics are.

I argue that the cosine similarity is a more interpretable metric. Cosine similarity simply measures the angle between two vectors. This therefore measures the relative changes in components of both vectors. For frequency, we can imagine that this means how much the relative frequency matches with each other. This relative frequency can then be used as a heuristic for how similar two species are.

However, for the Pearson correlation, we are assuming that the various frequencies act essentially as independent samples of a distribution that might be correlated with another distribution. It is not exactly clear what $\rho = 0$ and its difference from $\rho = -1$, and how much this negative correlation should be taken into account as dissimilarity.

Therefore for its interpretability and usage as a metric, I believe that the cosine similarity should be the best metric to use.

Part 4: Genome Assembly

All code and examples are found under 'q4/'. This problem was tackled with the following strategy:

1. Grab the sequences and create an aggregate kmer-frequency array of length kmer by summing up the frequency over all possible kmers.
2. With the optional flag '-t' or '-tol', allow the user to define what kmer frequency should be merged with its neighbours, defined as all possible kmers that are a single nucleotide away. This is the error correction step.
3. Generate the deBruijn graph based on the total frequency of kmers by simply grabbing each kmer and setting the from kmer as the prefix and the to kmer as the suffix.
4. Do some analysis on this graph, including finding all the components. From these components, remove all components except for the largest one. Empirically, most of these components should be safe to remove; with kmer of size 9, we see that the component sizes are $\{7679, 2, 2, 6, 2, 3, 3, 2\}$. Since these are small enough and would potentially cause deadends within the Eulerian path finding, we remove them.
5. Next, we clean the graph optionally using the '-clean_graph' flag which if used will add and remove potential edges that are making certain nodes' indegrees and outdegrees unbalanced.

Given more time, I would have added the marking of a node’s depth and the removal of any branches that lead nowhere.

6. Finally, we find the Eulerian path (or a close one) by traversing over the edges randomly until we cannot anymore. Note this was done by turning the number of edges into weights for a probability distribution.

Run the code with the following: ‘python assemble_genome.py -i <input_file>’ with the following optional flags:

1. ‘-output_dir’ specifies the output directory. If set, this also changes the output file to be of form: <input> _assembled_genome_kmer_ <kmer-length> _tol_ <tolerance> _clean_graph_ <clean-graph> _seed_ <seed>.fna
2. ‘-kmer_len’ specifies the kmer length to be used. Note that this makes the runtime exponentially slower due to the frequency array analysis. From experience, anything beyond $k = 10$ might be unbearable, and not worth it; see the k-mer length ablations section.
3. ‘-tol’ specifies the upper bound for where we will move the kmer frequencies from one variable to another. See the error correction step ablations section for more details.
4. ‘-clean_graph’ is a boolean flag which if set to true will add edges from some nodes to others and remove them where possible to balance the tree more. More details found under cleaning graph ablations.

For my best run, use:

BLAST Testing

For BLAST testing, I simply tested against the BLAST website. This will be a verification alongside other heuristics for how good alignments are alongside how certain ablations led to better results.

Error Correction Step Ablations

For this set of ablations, I looked at understanding how the tolerance flag will help or reduce the accuracy of the alignment. Note: To keep things consistent, I decided to use one seed of 42 throughout my testing and ablations. While this might not be accurate of the entirety of the tests (and would require averaging over many seeds), it does show some possible situations of using certain seeds.

Below is a table of the resultant number of nodes, edges and components based on different tolerances for $k = 9$:

Tolerance	0	1	2
Node Count	8261	7699	7666
Edge Count	82770	82770	82770
Component Count	1	8	21

Table 1: Graph Analysis on Various Tolerance Ablations

Generally, there are less nodes (with the same number of edges as expected) but a higher number of components due to cutting off the bridges between components. This however, is not an issue because each component is composed of a very small number of kmers, meaning that they are likely to be errors than anything else. Switching these reads to an existing read is a much more likely situation.

This is backed up by the BLAST testing where

Graph Analysis; K-mer Length Ablations

Cleaning Graph Ablations