Ethan Muldoon

CMSC 12300 Computer Science with Applications 3

Final Report

Introduction:

My project this quarter has been focused on genetic data. I would like to be able to take the genetic sequences of a group of species and compare them. Based on the differences between individual genes, I want to be able to calculate an average number of mutations separating a pair of species. Then, by calculating the differences between several different species, I would like to attempt to reconstruct an evolutionary tree showing how all the target species are related. Finally, at each branching point of the resulting tree, I want to determine which genes are most different between the species on either side of the branch.

I used the so-called ENSEMBL Annotated Human Genome Data for my project. This dataset is available as public data set on Amazon Web Services, making it convenient for use with the Amazon EC2 system. I used the FASTA format version, which consists of a brief header followed by a long genetic sequence. The actual contents of this data set are quite different than what the name implies. This dataset contains genetic data for 60 other species besides humans, giving me plenty to compare. Unfortunately, another difference from the name is that the data is not annotated in the slightest. There is very little information about the sequences and none at all about their origin or functions, making this dataset hard to get much interesting information from.

Because comparing genetic sequences is very computationally expensive, I have limited my investigation to the files containing cDNA data. This stands for complementary DNA, and is

DNA reverse-engineered from messenger RNA inside cells. In biology, cDNA is often used to add genes for foreign proteins into cells, but for my purposes it is useful because it is clearly grouped into individual genes. In principle the gene coding regions of the DNA could be detected by scanning the DNA itself, but that is far too complicated for this project. Using only the cDNA files drops the amount of data involved from 210 GB to only a few gigabytes, but It still qualifies as a big data problem because processing the data would take an unreasonable amount of time using only a single machine.

Text Comparison Algorithm:

The core of my project is based on finding the mutations between pairs of genes, which is essentially calculating the number of changes between two strings. While this sounds simple, it is actually a fairly complex task when you include the possibilities of addition and subtraction of characters. Therefore, the first thing I did was research text comparison algorithms. I settled on an algorithm described in a research paper where the problem is envisioned as a search for the shortest path through a graph. The graph is constructed by matching the strings to a grid of nodes as shown in figure 1, and adding diagonal edges where the strings match. Finding the shortest path from the top left corner to the bottom right corner of this graph is equivalent to calculating the shortest series of edits to convert one string into the other.

I made some changes to this algorithm to make it more suited to DNA specifically. The algorithm from the paper only allowed horizontal or vertical movements, corresponding to additions and deletions of characters. I want to determine a number of mutations, and while it is possible to have insertions and deletions in DNA, there are also substitutions. I want to treat these as a single mutation instead of an addition/deletion pair, and to support this I modified

the algorithm to always allow a diagonal move. However, the graph edges are now weighted, and diagonal edges at matching characters in the strings have zero weight while other diagonal edges and horizontal and vertical edges all have a weight of one.

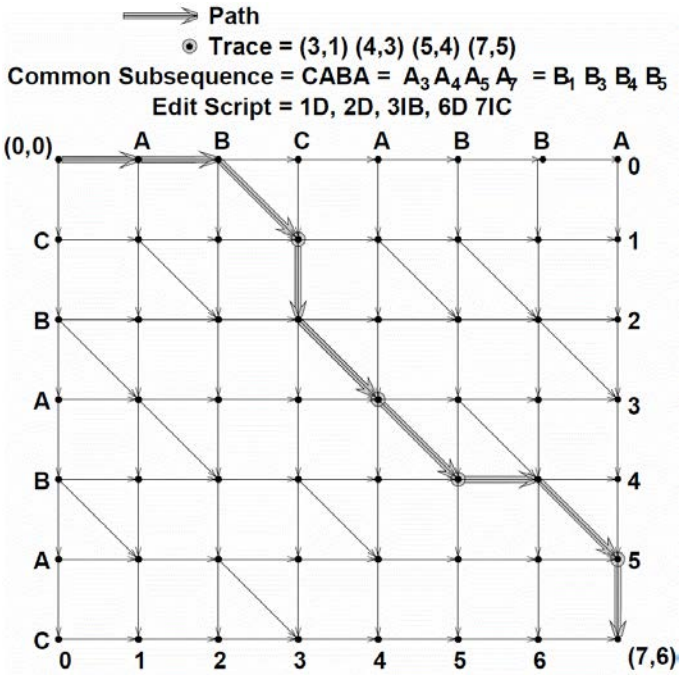

Figure 1: An example edit graph comparing 'CBABAC' to 'ABCABBA'

To calculate the best path through these graphs, I use a method described in the paper where I only need to keep track of the position I have reached along each diagonal of the graph. The search starts from the top left, and calculates the furthest possible point we could have reached along each diagonal over several iterations. When one of our endpoints reaches the bottom right corner of the graph, we have found the number of differences between the strings. However, for the later versions of my project, I switched to a more complicated method involving searching simultaneously from both the top and bottom corners of the graph and checking if the paths have crossed. This refinement means that for strings with many differences the algorithm can be up to twice as fast since half as many diagonals are searched.

Implementation – C++:

My original implementation for this project utilized python and ran on my own computer. However, I soon realized that this was far too slow to produce anything close to useable results. I therefore decided that I had to rewrite much of my code in C++ for speed. In order to retain the functionality of python, I decided to split my program into two main parts: C++ code for comparing many genes as fast as possible, and python code for collecting and analyzing the resulting pre-processed data. The C++ code takes the raw data and a file containing a list of species to be compared as inputs. It outputs files containing lists of the best match gene pairs and their differences. The python code then reads these output files and uses them to construct a genetic tree and calculate which genes are most distinct between different species.

My C++ code essentially uses a functional programming approach, but I used a small amount of object-oriented programming. I used two small classes, one representing a single gene and another containing a pair of sequences waiting to be compared. However, I mostly used these simply as containers to hold related data, and they didn't have any functions associated with them. I also had a more developed class representing a species with a list of genes. This class had a set of functions designed to look into its corresponding cDNA file and extract relevant information. When a species class is made, it scans through the cDNA file looking for the headers, and stores the names and locations of each gene. When that gene's sequence is required, the file is opened again to retrieve it. This approach might seem to sacrifice speed in favor of saving memory, but the comparisons between genes take long enough that this method is easily able to keep up.

To compare a pair of species, I compare each gene in species A to each gene in species B, looking for matches. Since the species in the ENSEMBL dataset have thousands of genes listed, this adds up to a huge amount of comparisons. To help speed this process up, I wrote this part of the program in a multi-threaded way. I implemented a producer-consumer system where one thread figures out genetic sequences that need to be compared and saves them to a buffer, while a set of worker threads take sequence pairs out of the buffer and compare them. This code is designed to run on Amazon EC2, on cc2.8xlarge type instances. These machines are designed for very compute-heavy tasks, and have a pair of 8-core processors with hyper threading, giving a total of 32 virtual CPUs. Therefore, my program is designed to take advantage of these by creating 32 worker threads all processing data in parallel. My testing has shown that a single supplier thread is more than fast enough to supply input to this many workers, and with this setup the worker threads are processing data almost continuously. While the program is running the average CPU utilization of the machine will stay steady at 100%, which indicates that I am able to harness the full computational power available.

Compared to my original prototype implementation using Python on my own computer, the performance of the final version has improved by several orders of magnitude. I do not have information on the exact timing, but my prototype took several seconds to compare a single pair of genes while the final version takes about an hour to compare two sets of 1000 genes. This means that the final version is running something like 1000 times faster than the prototype was. Unfortunately, this increase is still not enough to compare all the genes in a species in a reasonable amount of time. Therefore, I have included a parameter setting the

maximum number of genes to compare. For testing I used a limit of 2000 genes, which caused the program to run for several hours.

Implementation – Python:

The other part of my project, called analyze.py, is designed to analyze the output data from my C++ program. This part is written in python since it is easier to work with and debug, and also so I can take advantage of pylab to plot calculated genetic trees. This part also includes code from my original prototype. Speed is not a major concern for the additional processing steps after initial comparisons between the species, so there was no reason to try to port this part of the code to C++. This code is intended to be run on my own computer after I have collected the output data files off of Amazon EC2.

The main purpose of this part of the program is to attempt to generate a genetic tree from the data. This starts by generating a matrix containing the differences between each pair of species. To determine the differences between species from only a subset of data, I make two simplifying assumptions: that all the genes in one species have a good match in the other species, and that the genes are randomly distributed throughout the file. In this case the probability that any given gene in species A has been paired with its true match in species B should be roughly the same as the fraction of the genes in species B that we have looked at. If we further assume that matching genes which have been highly mutated are still more similar than non-matching genes, then we can take that same fraction of the best matched gene pairs and conclude that they are likely to be correct matches.

Once I have produced this matrix of the differences between the species, the program attempts to generate a binary tree where the most closely matched species are on adjacent

branches. This tree should match the shape of the true evolutionary tree, though the lengths of the branches will be different. In my trees the horizontal position of each branch is proportional to the differences calculated between the species on either side of the branch. Once I have finished constructing the tree, the final step is to plot it, which I do using pylab.

Results:

Unfortunately, the results of all this are unsatisfactory. Despite spending several hours processing the genes, the tree I finally got out bears little resemblance to the true evolutionary tree. The wrong species are paired together, and the shape is wrong too: each species should branch off from the human's line one at a time. In fact, this tree looks significantly worse than the first test tree I produced using my prototype to examine a single gene, which looked correct except for having the gorilla and orangutan swapped. This makes me think that perhaps less of my matches are genuine than I thought. I suspect that despite the seeming randomness of the data files there is in fact an underlying order, meaning my assumptions about the probability of getting a genuine match are probably false.

I was also unable to finish my original goal of being able to identify the genes that changed most dramatically between neighboring species in the tree. This is because, with only a subset of the data, all the poorly matched genes most likely just match a gene I have not examined. It would be silly to try to pull out the genes with the greatest difference from their partners, as these will most likely just be ordinary genes.

I believe that both failings of my project could be solved by analyzing a larger subset of the genes. While I do not think I can make my comparison go much faster, I could probably make significant improvements by strategically choosing when not to compare genes. I see two

possibilities to increase the performance: having some criterion to skip comparing genes of significantly different lengths, and adding a method to short-circuit the comparisons if it becomes apparent that the genes being compared are a poor match. By further optimizing my comparison algorithm, I would probably be able to gain enough speed to examine significantly more gene pairs and hopefully produce some better results.

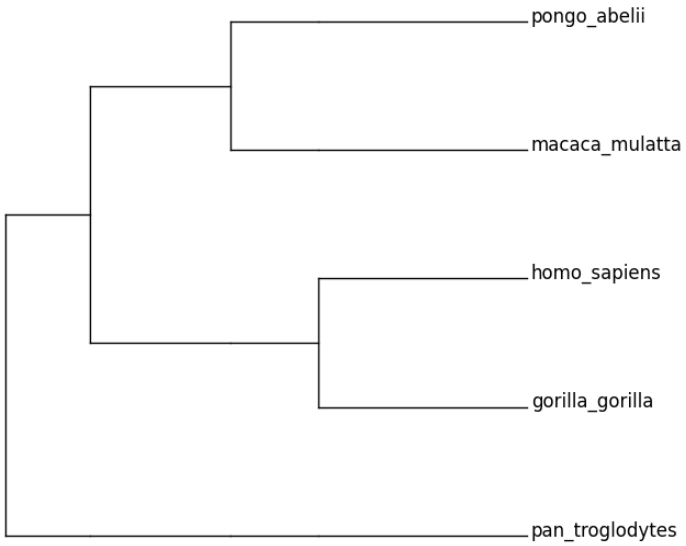

Figure 2: Output Tree using 2000 genes from each species

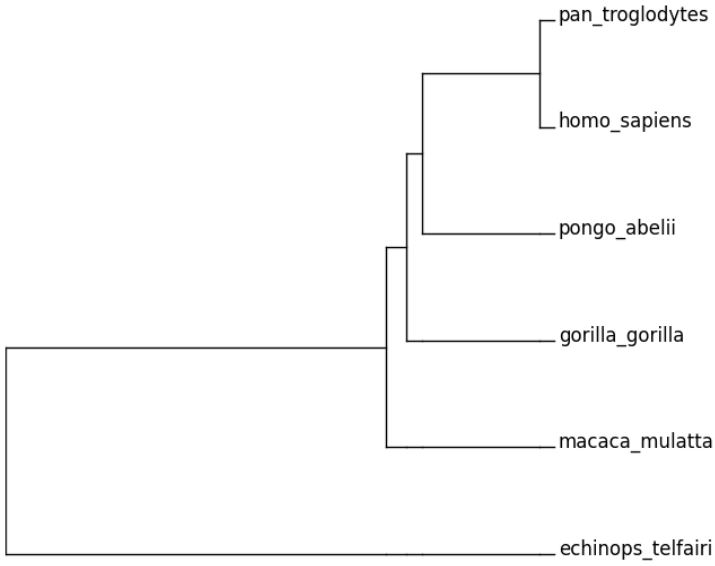

Figure 3: Test Tree using only the BRCA2 gene