Project Name: Genes and Trees

Roger Zou - rzou@college.harvard.edu
Aidi Zhang - aidizhang@college.harvard.edu
Belinda Zeng - belindazeng@college.harvard.edu
Charles Zhang - charleszhang@college.harvard.edu

Brief Overview

Since the completion of the human genome project in February of 2011, DNA sequencing has become more and more popular, affecting even the US Criminal Justice System through the Innocence Project. DNA sequencing draws heavily from computational biology, and one of the major problems facing the field today is how to best compare similarity between two genes or strands of DNA. Two DNA sequences could have 99% of the genome in common and in the correct order, but if the strands are offset by one nucleotide, then comparing directly would result in perhaps as little as 1% in common.

Thus, our goal is to further explore the process of DNA sequencing in a computational context--namely through Hirschberg's algorithm, which returns the longest common sequence. This would let us to determine optimal alignment to compare similarity between DNA sequences.

In order to better gage the accuracy of Hirshberg's through real life application, we aim to use Hirshberg's to create a phylogenetic tree. By dividing the length of the LCS (longest common subsequence) by the length of the original DNA sequence, we can determine similarity between species. Using these, we can construct a phylogenetic tree and compare accuracy to generally accepted phylogenetic trees today to determine the accuracy of our algorithm.

Feature List

- **Vital feature:** Implement everything in Python instead of Ocaml. No member is currently proficient in Python so it'll be a blast to learn.

Commented [BZ1]: Learning Python turned out to be a pretty

- **Vital feature:** We want to implement Hirschberg's algorithm for finding the longest common subsequence of two given DNA sequences. Hirschberg's achieves this in

O(mn) time and O(min{m,n}) time, where m and n are the lengths of the two DNA sequences respectively. Hirschberg's uses dynamic programming/memoization by keeping a dynamic programming 2D array f[i][j], where f[i][j] represents the length of the common subsequence of two strings s[i:] = $s_i s_{i+1} ... s_n$ and t[j:] = $t_j t_{j+1} ... t_m$. We want to eventually find f[0][0]. It will also make use of the divide and conquer technique to reduce space complexity to linear space.

- **Vital feature:** phylogenetic trees for storage and showing results of sorting the different species. Currently, we plan to implement this via binary tree. Ordered list by similarity to bacteria (which we will have as the root of the phylogenetic tree). Then go back and check how similar other elements are to the elements closest to bacteria to tell if make new nodes or not, and make sure order is correct.
- **Cool Extension (prioritized):** Implement Smith-Waterman algorithm or Needleman-Wunsch algorithm for sequence alignment (both are viable alternatives to Hirschberg's algorithm with differing running times of O(m^2 n) and O(mn), m>n, respectively), and comparing resource usage (time and space)
- **Cool Extension:** Add a jazzy front end for viewing the data. Online website and/or mobile app (iOS/Android. Would require Obj-C/Java). Maybe includes imagery
- **Cool Extension:** Have reset options for comparing humans, chimpanzees, fish, and other common species instead of having to manually enter genome information.

Commented [BZ2]: We instead implemented Levenshtein's

Commented [BZ3]: Prints a tree to the screen

Technical Specification

In order to apply this to DNA, we'll have to pass in dummy values for the DNA sequences of various species (these dummy values will stay true to the similarity ratios between species in the real world). We've split the technical specification into two sections: implementation of Hirschberg's and the implementation of the phylogenetic tree.

Pseudocode for implementation of Hirschberg's:

Let f(i,j) be the length of the longest subsequence of the two strings s[i:] = $s_i s_{i+1} \ldots s_n$ and t[j:] = $t_j t_{j+1} \ldots t_m$. We use bottom up dynamic programming to ultimately find f(0,0). The recursive function for f(i,j) would look like this:

--insert image here :DDDDD--

Using memoization, we can compute our desired value in O(mn) time, where m and n are the respective lengths of the two sequences. We can alter our lookup table in memoization to remember what choice we made at each recursive step in our optimal longest common subsequence, so as to obtain the actual subsequence itself.

In addition to achieving O(mn) time complexity, Hirschberg's achieves O(min{m,n}) space complexity. We use a divide and conquer technique: let f(i,j) be the longest common subsequence between s[i:] and t[j:]. Also let g(i,j) be the longest common subsequence between s[:i] and t[:j]. Computing g(i,j) would be similar to computing f(i,j).

To divide and conquer, we compute f(n/2, t) and g(n/2, t) for all values of 0 <= t <= m and we choose t such that f(n/2, t) + g(n/2, t) is maximized. The space in this divide and conquer would be O(m) since when computing values of f and g, we can reuse the space used to compute previous values.

Aidi and Charles will focus on implementing Hirschberg's algorithm, keeping in mind that we want the input to be 2 DNA sequences and the result to be the similarity between the two sequences.

Implementation of the Phylogenetic Tree

For the implementation of the phylogenetic tree, we will use an ordered list of the species ranked from most similar to our root (which is bacteria by default) to least similar to our root, which Belinda and Roger will work on. Our first two items in the linked list will be inserted into the two nodes of our binary search tree, after which we will compare the similarity between our third element and the two nodes we just inserted. We will then insert into the branch that is most similar to our third element, and so on.

Once we are done with the implementing the Hirschberg's algorithm and the phylogenetic trees, and if we don't have any problems with either, we will proceed to try to implement the Smith-Waterman algorithm, which returns the same result as the Hirschberg's algorithm. We want to compare the two algorithms and see how accurate each is, and determine which algorithm is better.

We do not forsee any sections that cannot be easily modularized.

Next Steps

The language that we will use is Python. Since the majority of us are uncomfortable with coding in Python, we will focus on learning Python, try to fully understand Hirschberg's algorithm, and start attempting to implement that in Python before the Final Draft of the Specification.

Commented [BZ4]: Detailed more specifically in the final specification