Genes and Trees:

Hirschberg's Algorithm, Levenshtein's Algorithm, and Binary Trees

By Aidi Zhang, Belinda Zeng, Charles Zhang, Roger Zou.

Our Fantastic Demo Video

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.

We ended up coding both Hirschberg's algorithm and Levenshtein's algorithm in python in order to compare similarity values between different DNA sequences. We then constructed a phylogenetic tree based on the similarity values.

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, and Levenshtein's algorithm, which returns the edit distance. These two algorithms let us compare similarity values between different DNA sequences and then construct a phylogenetic tree. By dividing the length of the LCS (longest common subsequence) returned by Hirschberg's by the length of the original DNA sequence, we can determine the similarity between species. We can also divide the edit distance obtained by the Levenshtein's by the length of the original DNA sequence, to find another similarity measure, Using these, we can construct a phylogenetic tree and compare accuracy to generally accepted phylogenetic trees today to determine the accuracy of our algorithm.

Planning

We planned to build Hirschberg's Algorithm to compute similarity values between genome strings and Z. Here are our *initial draft specification* and *final specification*. Overall, we were able to stay on task and actually implement quite a few of the "cool extensions" we listed in the timeline. For example, we did implement a front end for the phylogenetic tree, which now prints to the screen. We also implemented Levenshtein's as an alternative to Hirschberg's and were able to compare time and space complexity. Here's a breakdown of our milestones and how they differed from our actual implementation, if at all.

- **Learn Python (intended deadline: Friday 4/18) -** we did learn a lot of Python by this deadline, but learning Python also turned out to be a more continuous process. That is, we also learned as we coded, mostly by taking full advantage of Google. ;)
- **Pseudo Code and Basic Code (Modules, etc.) (Sunday 4/20) :** We did have all our pseudo code for Hirschberg's algorithm(Charles & Aidi) and the phylogenetic trees (Roger & Belinda) by Sunday
- **All Code Drafted, even if slightly buggy (Wednesday 4/23) :** We missed this deadline by two days, and ended up having all our code down (though still buggy) for both Hirschberg's algorithm (Charles & Aidi) and the phylogenetic trees (Roger & Belinda) by **Saturday 4/26**.
- **Check-in Meeting (Wednesday 4/23):** We discussed some of the problems we ran into, reevaluated the timeline, and assessed potential to begin work on cool extensions
- **Functionality Checkpoint (Friday 4/25):** We turned it in.
- **Finish Debugging Code (Saturday 4/27):** Debugging took much longer than expected, mostly because of issues with optimization, indexing into nodes, and general

unfamiliarity with Python. We only finished debugging on **Wednesday 4/30**, and that was without optimizations.

- We then began work on extensions
    - **Cool Extension :** We didn't have time to 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). Instead, we implemented the Levenshtein algorithm and compared time and space complexity.
    - **Cool Extension:** We did add a jazzy front end for viewing the data by printing the phylogenetic tree to the screen. We didn't really have time to build an online website and/or mobile app (iOS/Android. Would require Obj-C/Java).
    - **Cool Extension:** We didn't have time to reset options for comparing humans, chimpanzees, fish, and other common species instead of having to manually enter genome information.

Design and implementation

Hirschberg's Algorithm

Hirschberg's algorithm compares two strings and finds the longest common subsequence between the two using O(min(m,n)) space in O(mn) time.

It was difficult to fully understand the algorithm, and of course, fully implement. Once we did fully understand it, the next difficult part was coding it so that it would work for all examples. Corner cases were often difficult to detect, especially when sometimes the algorithm would work and sometimes it wouldn't.

It was relatively simple to implement the algorithm using O(mn) space and the same time. Minimizing space used was more difficult because there wasn't much online documentation on how we should go about doing this. We ended up having to draw multiple n x m diagrams with multiple examples before we could find what was wrong with our program.

We ended up doing lots of optimization for Hirschbergs. Other than the obvious optimization of memory space that Hirschberg is designed to implement, we optimized the function by setting variables for the outputs of functions (such as length of a string) so that we wouldn't have to run the function every time we needed the same value. We also created multiple helper functions that we referred to in our main function, "hershies".

Levenshtein's Algorithm

Levenshtein's distance represents the smallest number of insertions, deletions or substitutions it takes to "walk" from one string to the other. Since DNA sequences often undergo insertions, deletions and substitutions, all occurring on approximately the same timescale, Levenshtein's distance is an accurate measure of the similarity between two DNA sequences.

Some difficulties encountered included deciding whether the recursive or iterative version of Levenshtein's would be more efficient. After analyzing the stack calls, we decided to use the iterative version which is more suited for Python.

Our first implementation of Levenshtein's was purely recursive and worked relatively well but we subsequently improved it to be more time efficient by keeping a 2D array called lev that keeps track of subproblems and their solutions. We found that the iterative version is more time-efficient by comparing Python timestamps.

Phylogenetic Trees

For the implementation of the phylogenetic tree, we used 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 worked on. Our first two items in the linked list were inserted into the two nodes of our binary search tree, after which we compared the similarity between our third

element and the two nodes we just inserted. We then inserted into the branch that is most similar to our third element, and so on.

Implementing the phylogenetic tree was slightly more difficult than expected, mostly because we at first could not figure out why only the first two elements from the list we passed in were being added to the tree. We later realized that after adding the first two elements by default, the other elements were being added through a separate for loop, which was actually creating a whole new tree entirely, which was not connected to the tree with our first two elements. Moreover, we found that we couldn't index into the nodes of our tree, which made identifying and working with each node more difficult. Another challenge was that after working a semester in OCaml, we'd become attuned to the syntax of Ocaml. For example, we forgot to include return, which caused us a lot of confusion and frustration when debugging.

In terms of improvements, our first working implementation didn't really print anything to the screen. We ended up checking by printing something like "b.children[0].data" to the screen, and manually checking that way. While that worked, it was still difficult to keep track of everything, so in the end, we decided to make it easier by just printing the entire tree.

Now our phylogenetic tree prints something like:

parent
    son
        grandchild
        grandchild
    daughter
        grandchild

This format made it much easier to check our accuracy.

Reflection

We were pleasantly surprised by how many built in functions there were in Python. Considering the fact that all of us learned Python specifically for the CS51 Final Project, we were very happy with how (relatively) easy it was to pick up and learn on the fly.

We were not-so-pleasantly surprised by long Hirschberg's takes to run. We made a point not to look up pseudocode online so we could better understand the entire process of writing the algorithm from scratch, and so as not to be influenced by other people's code.

Our decision to write Levenshtein's iteratively worked really well--even passing in strings that are multi-line, Levenshtein's runs in less than an second.

Our decision to make Hirschberg's recursive did not work out so well--even though it runs in O(mn) time, because of the large constant coefficient, it still runs quite slowly.

If we had more time, we would continue to optimize Hirschberg's and Levenshtein's so we could pass in an entire human genome, fish genome, etc. and have the code run relatively quickly and efficiently.

If we were to redo this project from scratch, we would not necessarily change much at all, except to rewrite Hirschberg's iteratively instead of recursively. We've found that unlike OCaml, Python is not particularly efficient with recursion.

Through this entire project, we learned a lot about teamwork. Yes, we learned a lot about coding efficiently in python as well, but I think the greatest lesson was in working together. We divided our work into two sections initially, Aidi and Charles working on Hirschberg's while Roger and Belinda worked on the phylogenetic tree. These smaller groups made it easier to meet and code together on the separate parts, but at the same time, discuss as a group some of the larger issues like interfaces, progress, and upcoming deadlines. By dividing up the work, we were able to meet most of our ambitious deadlines as well as implement some of the desired cool extensions.