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

Commented [BZ1]: Mostly, we modified the feature list and the timeline

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.
- **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.

Commented [BZ2]: While Hirschberg's runs at O(m,n) time, the constant is so large that it still runs inefficiently.

- **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 [BZ3]: At Mike's recommendation and since the Levenshtein's algorithm is much more efficient, we decided to implement the Levenshtein's algorithm instead of either the Smith-Waterman or Needleman Wunsch algorithm.

Commented [BZ4]: We're printing the phylogenetic tree to the screen

Commented [BZ5]: Didn't have time to implement this

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.

Hirschberg's Algorithm

Aidi and Charles will focus on implementing Hirschberg's algorithm, keeping in mind that the input will be 2 DNA sequences and the result will be the longest common subsequence, allowing us to determine similarity between the two sequences.

Signatures/Interfaces
We're not using signatures and interfaces to Hirschberg's.

Things to think about: (we should all do this for our own sections)
What types or abstractions will you provide? How will these abstractions reduce the conceptual complexity of your project? Which values and functions will you expose? Which will you hide?
We will expose all of our values and functions, as according to some research that we did, abstraction in Python using the abc (abstract base class) module is of limited utility and you usually avoid using it altogether. We will be using concrete classes instead.

What properties should clients of the component respect (e.g., all input values must be of a certain format)?
We will be using the hash table from Belinda and Roger's component to pull the corresponding list of chars from the two organisms that we are currently considering. This will feed into Hirschberg's function, which will take in two parameters with type char list.

What properties will the component ensure?

This component will ensure that the longest common subsequence found is also of the type char list. It will also ensure optimal time and space complexity such that long sequences of genomes may be compared efficiently.

Pseudocode:
This is our pseudocode for Hirschberg's in paragraph form:

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

$$f(i,j) = \begin{cases} 0, & if\ i = n\ or\ j = m \\ \max\big(f(i+1,j), f(i,j+1)\big), & if\ s_i \neq t_j \\ 1 + f(i+1,j+1), & if\ s_i = t_j \end{cases}$$

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.

Modules/Actual Code (we all do this for our own sections)

Begin writing code for the most important parts of your project. At this point you have the skeleton of your project, and you should begin writing the code most fundamental to the success of your project. All of your projects are very different, so we cannot tell you specifically what you need to do. Use the writeup as a place to show us the progress you have made on your project.

```
def lookup(s,t,i,j):
      if i == length_s or j == length_t:
            return 0
      elif s[i] == t[j]:
            return 1 + lookup(s,t,i+1,j+1)
      else:
            return max(lookup(s,t,i+1,j),lookup(s,t,i,j+1))
print lookup(s,t,0,0)
```

The above code returns the length of the common subsequence between two strings (or lists of chars, in Python), s and t. We then modified the code to give us the longest common subsequence itself:

```
def lookup(s,t,i,j):
      if i == len(s) or j == len(t):
            return ""
      elif s[i] == t[j]:
            return s[i] + lookup(s,t,i+1,j+1)
      else:
            f = lookup(s,t,i+1,j)
            g = lookup(s,t,i,j+1)
            if len(f) > len(g):
                  return f
            else:
                  return g
print lookup(s,t,0,0)
```

A slight modification still needs to be made to implement Hirshberg's properly, which involves a space-saving divide and conquer trick.

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.

Signatures/Interfaces

We've implemented the basic tree we will be working with in this part of the final project. While the tree itself will be a simple binary tree, the algorithm determining where to input which elements where will be more difficult.

```
Class Node(object)
    def __init__(self, data):
        self.data = data
        self.children = []
        my_hash_table = {}

    def add_child(self, obj):
        self.children.append(obj)
```

Commented [BZ6]: We also added self.parent

Commented [BZ7]: We didn't end up using a hashtable, and instead used a list

Things to think about:

What types or abstractions will you provide? How will these abstractions reduce the conceptual complexity of your project?

Since we'll be working in Python, we've been doing some research. According to Stack Overflow:

> *"If you want to create abstract base classes, you can, but they are of limited utility. It's more normal to start your class hierarchy (after inheriting from object, or some other third-party class) with concrete classes."*

As a result, we probably will not be using many abstractions at all. However, we will be creating new methods to call information from our hash table, which is a kind of abstraction.

Which values and functions will you expose? Which will you hide?

We will expose the organism names (in string form), the tree structure (for seeing phylogenetic tree), and the organism's genome (stored via hash table), but will be not displaying the calls to Hirschberg's algorithm that we will use to generate a similarity value, nor will we be displaying the whole hash table. Instead, we'll only use calls to reference values in the hash table.

What properties should clients of the component respect (e.g., all input values must be of a certain format)?

Normally, Hirschberg's will return a longest common subsequence, from which we can calculate a similarity value (which will be longest common subsequence divided by the length of the genome). However, since we'll be working on these two sections concurrently in order to achieve the ambitious deadlines we've set in the timeline, we'll be working with dummy values as follows:

```
bacteria = ("bacteria", ['A','A','T','A'])
plant = ("plant" , ['A', 'A', 'A', 'T'])
fish = ("fish", ['C', 'C', 'T', 'A'])
human = ("human", ['G', 'C', 'T', 'G'])
```

Commented [BZ8]: We added in more dummy values, especially for Levenshtein's

Each element will be defined as a string representing the species name. To use the genome, we'll save it as a value in our dictionary/hash table, with the string name as the key.

name: bacteria = "bacteria"

Genome: my_hash_table[bacteria].

What properties will the component ensure?

The binary tree will ensure that we will be able to judge at a glance similarity between two different organisms based on the number of common links.

Pseudo-code

In order to fully understand how things should work without having to worry about the quirks of whichever language or tools you are using, we've written out pseudocode for implementing the phylogenetic tree.

```
# compute similarity values as necessary
  hirschberg's return value / total dna sequence size
# sorted list [most similar to root...least similar to root]
    bacteria = ('bacteria', genome)
    list of tuples [bacteria,
# Insert the first item as the root
# Insert the second element as a child of the root
# Rest of the elements
   Compare to most recently inserted & parent (using INSERT UP)
    INSERT UP (called on the node and the parent node) ->
        - hit the root, call INSERT DOWN
        - closer to the child node than to the parent,  call
INSERT
        DOWN
    INSERT DOWN (called on the node and the child node)
```

```
        - reach a leaf, then INSERT NEW NODE
        - branches into two children, compare similarity values,
          INSERTDOWN on the more similar branch
        - traverse down until closer to the parent than the
child,
          INSERT NEW NODE
```

More specifically for INSERT DOWN:

```
     #INSERT DOWN
     #if len(b.children) == 0 :
         # creeate new node, add as child
     #if len(b.children) == 1 :
         # go to the next child
     #if len(b.children) == 2 :
         # compare similarities between node we want to create
     and 2 children
         # traverse the branch with more similarity
```

Modules/Actual Code

We just got started with the actual coding, and are working on implementing the algorithms described above. So far, we've only added the root element and it's first child (the first two elements from the list).

```
elements = [bacteria, plant, fish, human]

for theelement in elements :
    print theelement

def create_tree (el = elements) :
    b = Node (el[0])
    p = Node (el[1])
    b.add_child(p)
    print b.children
#f = Node(fish)
#h = Node (human)
create_tree()
```

Progress Report (we all do this for our own sections)

We've implemented the basic tree in python and have inserted the first two elements of the list.

We also already have all our pseudocode for the algorithm to insert the rest of the elements (delineated above). Please refer to the next section (Next Steps & Timeline) for a comprehensive listing of our upcoming goals and deadlines.

Next Steps & Timeline

We've set up concrete goals and checkpoints for ourselves. The following is an ambitious roadmap for us, including total milestones and upcoming deadlines. Even though it is quite ambitious, we believe that having this timeline will encourage us to work ahead of time and plan for potential difficulties with debugging.

- **Learn Python @ www.trypython.org (Friday 4/18)**
- **Meet w/ Mike (Friday 4/18)**: Go over the draft spec and prep for the final spec. People going: Belinda, Aidi, Charles (maybe)
- **Pseudo Code and Basic Code (Modules, etc.) (Sunday 4/20) :** Hirschberg's algorithm for finding the longest common subsequence of two given DNA sequences (Charles & Aidi) and the phylogenetic trees for storage and showing results of sorting the different species (Roger & Belinda).
- **All Code Drafted, even if slightly buggy (Wednesday 4/23) :** Have all the code down, even if there are still bugs, for both Hirschberg's algorithm (Charles & Aidi) and the phylogenetic trees (Roger & Belinda).
- **Check-in Meeting (Wednesday 4/23):** Discuss problems we've run into, reevaluate the timeline, assess potential to begin work on cool extensions
- **Functionality Checkpoint (Friday 4/25)**
- **Finish Debugging Code (Saturday 4/27):** Both for Hirschberg's (Charles & Aidi) and phylogenetic trees (Roger & Belinda)
- If everything goes as planned, begin work on cool extensions:
    - **Cool Extension (prioritized) :** Implement Smith-Waterman algorithm or Needleman-Wunsch algorithm for sequence alignment (both are viable

Commented [BZ9]: Learning Python turned out to be a continuous process

Commented [BZ10]: We met this deadline!

Commented [BZ11]: We met this deadline as well!

Commented [BZ12]: So glad we scheduled this! It was really helpful!

Commented [BZ13]: Debugging took much longer than expected—we finished debugging by Wednesday 4/30
Optimizing the algorithms actually also ate up a lot of time

Commented [BZ14]: We ended up actually implementing Levenshtein's instead, and then compared the time and space complexity in our video 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 [BZ15]: Since we didn't have time to code an online website and/or mobile app we just printed the phylogenetic tree to the screen

Version Control

We will be github to conduct version control. Roger is sharing his repository with the rest of us.

:) We will be committing to https://github.com/rogergzou/cs51

Suggestions?

Please let us know if you have any thoughts or concerns. You can reach us at aidizhang@college.harvard.edu, belindazeng@college.harvard.edu, charleszhang@college.harvard.edu, rzou@college.harvard.edu