PROJECT 4: BIOINFORMATICS 26

GEORGE WANG
gswang.cs61a@gmail.com
Department of Electrical Engineering and Computer Sciences
University of California, Berkeley

August 5, 2010

Note: The data files linked below are not yet ready, but should be shortly. This document should be enough to give you a strong head start.

1 Background

First off, why is there all the buzz about DNA? Why do we care about DNA? In short, DNA is a coding scheme, a language, that describes your physical biological body. DNA goes through a process called transcription which converts it to mRNA. This mRNA is further transformed into proteins through translation. Proteins form the major building blocks of who you are from a biological point of view. Note that this is a dramatically understated version of what DNA influences. Studies have correlated DNA sequences with everything from intelligence to ability to play sports.

This project explores two powerful examples of bioinformatics: genome sequencing and phylogenetic tree construction. In a real way, these techniques would not be possible on the scale they are used without the aid of computers.

2 Reassembly

2.1 Background

The Human Genome Project had a fundamental roadblock in its way. The chain termination technique used to sequence a section of DNA is not good enough to sequence an exceptionally long section of DNA. Specifically, it is unable to sequence a sequence of base pairs longer than about 1,000 base pairs. Thus, in order to do this, a method known as SHOTGUN SEQUENCING is used. Biologically, this means they use something called a restriction enzyme to 'cut up' a long DNA strand into small pieces. Once done, you are left with many pieces of DNA that have overlaps, and this is where the computational component kicks in to reassemble these into our original DNA.

2.2 Our Abstraction and Algorithm

We will abstract out the chemistry and treat DNA as just a word, in the same vein of the words and sentences you are accustomed to. For example, we start with a string like: 'maryhadalittlelamb'. From there, we cut it up into 'mary', 'ryhada', 'littlela', 'adalittle', 'lelamb'. Our job is to reassemble them.

We apply an algorithm called a 'greedy algorithm'. In other words, we do the best thing that we can see immediately, and we do that, and we'll repeatedly do that. Your job is to code this algorithm. Your input will be a list of words, some of which overlap. You will write a function that combines the two words with the most overlap. In the example above, 'littlela' and 'adalittle' overlap by 6 characters, and these two would be combined into a single word: 'adalittlela'. You repeatedly do this process until you are left with a single word. You may need helper procedures to calculate the overlap between any two words, then repeatedly run this helper function. You may also wish to test this on smaller input before running it on a large segment of code.

We've provided an input file called `cutSequence.scm`. This will define a list named '`snippets`', which contains snippets of DNA. You should run your code on `snippets`.

2.3 Problem Analysis

We've changed this problem from a problem about recombining DNA to a problem known as the SHORTEST COMMON SUPERSTRING. The problem asks us, given a set of strings (words), what is the shortest superstring (word) that contains each of the words in the set?

There's always two things we must be concerned with whenever we develop a computational model for anything. The first thing is whether our abstracted problem closely resembles the original. Fortunately, by and large, the shortest common superstring of all the segments is equivalent to the original sequence for DNA. However, this isn't always true, and this is something we must keep in mind.

Secondly, is whether our algorithm correctly solves our abstract problem. Although our Greedy Algorithm solves the problem in the sense that it always produces a Superstring, the superstring isn't necessarily the shortest. Thus, it has some chance of returning a 'wrong' answer. We are going to assume that so long as the amount of overlaps is 'large', so then our solutions will be okay.

Moral of the story: our problem formulation isn't perfect and neither is our solution, but they're good enough to get the work done.

3 Phylogenetic Tree Construction

Once you have the full genome of an organism, you begin to embark on the second phase of the project. Your job here is to be able to construct a phylogenetic tree. To do so, there are two major sections. First, we must create a way to find a way to measure how far apart evolutionarily two organisms are. Once we know that, we'll use a method called UPGMA to compute a phylogenetic tree. Since this is a project designed to simulate the experience of a researcher doing bioinformatics, the two algorithms used in this section will not be explicitly discussed below. It is your responsibility to utilize the considerable resources at your disposal to learn them on your own.

3.1 Evolutionary Distance

By evolutionary distance, what I mean is that we want to find out how long ago they branched off from a common ancestor. The assumption we will make is that the number of edits between the DNA of two

organisms is representative of the amount of time it took for these two organisms to evolve in different directions.

To do so, we'll use an algorithm called the Levenshtein Edit Distance algorithm. This is a well-documented algorithm, so exercise your research skills and learn it for yourself. You have discretion over what kind of input and output your program should take in. Theoretically, your code should work and be able to calculate a distance for each of the 4 organisms (6 pairs) of calculations. To save you on computation time, email me the edit distance between the two words in the `editdistance.scm` file and you will receive the numbers you will require as input to the next section.

3.2 UPGMA

Unweighted Pair Group Method with Arithmetic Mean (UPGMA) is a method used to build these phylogenetic trees. This method uses a key assumption known as the Molecular Clock Hypothesis[1]. This hypothesis assumes that the rate of mutation in a genome is relatively constant. In other words, different branches of the tree are going to undergo evolutionary changes at that constant rate. Although this assumption is valid in many situations, there are also cases where the assumption is not kept. These deviations from the clock hypothesis are outside the scope of our computer science course, but they are interesting and and the linked wikipedia article goes into more details. For this project, we will make this assumption and adopt UPGMA as our tree-building technique.

UPGMA is a well-document algorithm, and so once you have your distances, you should run these distances through the UPGMA algorithm and build the tree. For more information on how to implement this method, Google and Wikipedia are your best friends.

The 4 organisms, HIV1, HIV2, SIV, and FIV, you're building a tree of are all retroviruses. Specifically, they are immunodeficiency viruses that infect Humans, Simians, and Felines. Your program may take any input or output that represents the problem clearly. Think about how you want to structure the input and outputs of your program.

[1] http://en.wikipedia.org/wiki/Molecular_clock