504.420 Algorithms for Bioinformatics

William McElhenney

JHU ID: wmcelhe1

Project 3 Analysis

Due Date: 12/3/2019

Turned In: 12/3/2019

Project 3 Analysis

Analysis of Longest Common Subsequence

Longest Common Subsequence (LCS) is a bioinformatically relevant problem for the global alignment of oligonucleotide sequences as well as amino acid sequences in proteins. LCS can be made to either return a "score" of the alignment of string to another or can return a string representing the pairs that could be aligned. The implementation presented here does both, as to return the alignment string we must calculate the alignment score first.

The algorithm implemented for calculating the alignment score is an example of dynamic programming and is a direct pull from the textbook page 394. Since the implementation tries to faithfully interpret the algorithm presented in the chapter in Python, we can presume that the algorithm will complete in the textbook specified time-complexity of $O(nm)$ where n is length of one sequence and m is the length of the other.

What I Learned

When in doubt (and when possible) use outside sources to verify your work. I tried to verify the functionality by looking for LCS examples on Google. I came across one from Columbia University that provided the wrong answer, which I verified through a sourceforge LCS calculator (my output for Columbia's input matched the calculator).

What I Might Do Differently Next Time

The error check for multiple equals signs is implemented as a *for* loop instead of as a regular expression as I would like. This is because I could not get a regular expression that would properly detect two equals signs, so next time I would like to spend a little more time to get that properly working.

Design Justification

The project is broken into three files *LCS.py*, *fileIO.py*, and *project3.py*.

Project3.py is the driver script of the program. It manages the command-line arguments and coordinates file IO with the LCS implementation. It is designed to be as straight forward as possible. It simply takes the input parameters (ensuring they are good) and defines sequence generators before running through all combination of the input sequences (lines 59 – 71). As LCSs are calculated the driver also reports them to the command-line along with the labels of the sequences that generated that LCS.

LCS.py oversees calculating the length of the Longest Common Subsequence and building the sequence from the matrices generated. *Calc_lcs()* (lines 28 – 58) is an implementation of the LCS-Length algorithm on page 394 of our text. I tried to make this as one-to-one as possible with the text, but some changes were necessary due to differences in start indexing and personal/python quirks. This

function progresses by checking for an alignment at each index, incrementing the score from the right-top diagonal if a match is found or mirroring the best score achieved either above or to the left of it if no match is found. This file also contains the function for building the LCS from the b-matrix that is produced by *calc_lcs()*, which is called *build_seq()*. *Build_seq()* (lines 74 – 104) uses the directions encoded in the b-matrix to determine the correct LCS sequence, however it is different than its book counterpart because I wanted to be able to return the whole string and not just print out alignments as the function progressed. As such, *build_seq()* is an iterative solution to building the sequence that by default will run from the bottom-right most cell until it reaches zero either row- or column-wise at which point it will return the sequence for output. Doing this means little to no change in the time-complexity to building the LCS, but a slight increase in memory complexity due to need to store the LCS. This method also cuts down on the number of times we need to access storage, as the entire string is built before output (whereas if I implemented it like in the book it would likely need to access the drive every sequence match though we could probably get around this some other way).

FileIO.py contains functions for managing file input and output. It contains one input function (lines 16 – 80), which reads from the command-line parameter specified input file and makes sure it is congruent with what we expect the input format to be. This input function is a generator function meaning it will only load one sequence from the file at a time, which reduces memory complexity. The two output functions are to initialize the output file (*init_outp()*; lines 105 – 114) and to append the found and labeled LCS sequence (*write_outp()*; lines 91 – 96) to the output file, respectively.

Issues of Efficiency

The LCS calculation function in *LCS.py* is as close to the pseudocode in the book as possible and therefore should have the same time-complexity (*O(nm)*). This time-complexity can be empirically confirmed by observing the two nested for loops for traversing both sequences. As in the book, this dynamic algorithm makes and records the necessary calculations for LCS as we progress, so therefore the time complexity to calculate the alignment score at any given cell is $\theta(1)$ where if we did not record the preceding scores we would have to redo all the calculations that led to that cell.

Of additional note is that the file input is implemented as a generator function. Generator functions allow for us to iterate through data piece-by-piece, which is useful when we might be reading in multiple large strings (a collection of DNA sequences for instance) as it allows us to save memory by only reading in the current chunk (essentially treating the input file as an iterable stored on the hard drive instead of in memory). This likely slows the programs execution slightly as we must wait for the storage to return the current line but should free up a good portion of memory space when reading in large sequences.