BCB 5300
Final project details

For the final homework (and in place of an exam), I'd like to assign an implementation problem. I'm trying to give some flexibility here, so you are - at least in principle - welcome to come discuss with me and implement any algorithm that we have covered.

I'm listing a few suggestions here, many of which were compiled by the authors of our text book and a few of which were my ideas. Note that many of them have publicly available solutions. However, I'd like you to NOT just copy an online solution (which I'm pretty good at figuring out anyway), but rather am asking you to test your ability to translate a theoretical solution into working code.

As far as details, you are welcome to implement in any language, but please be sure to check with me to be sure I am able to compile and test your solution. You are also welcome to work with a partner on this one - just be sure that both contribute and are interested in the project, and that it's a challenging enough problem to be worth a two person effort!

Some suggested projects:

1. **Deciphering codes:**

   Background: Many problems in bioinformatics, such as discovering regulatory motifs, can be motivated by deciphering text codes. The encrypted text writen by Captain Kidd in section 4.4 uses a simple character substitution scheme. Mr. Legrand solves this with his knowledge that the most frequent word in English is 'the'. Had Legrand had access to a computer, he could have used more general knowledge of English to decode the text. For example, since 'the' is just a combination of three letters, called a 3-tuple, he could have compared the distribution of all 3-tuples throughout all English text to the distribution of 3-tuples in the encrypted text. Likewise, this can be done with all 2-tuples, or even by simply comparing the distribution of single letters.

   Implementation: Write a program to find the distribution of all k-tuples in a corpus of English text. The choice of k should be left to the user, but should be less than, say, 10. For simplicity, remove all punctuation and spaces, and convert all letters to lower case. As a test case, apply this to Captain Kidd's text to see the best way to decipher it. Keep in mind that not all k-tuples will necessarily appear in the corpus.

2. **Motif Search:**

   Background: Regulatory motifs are the short sequences flanking the genes that are responsible for regulation of transcription when certain proteins, called transcription factors, bind to these motifs. Motif finding is the problem of discovering such motifs without any prior knowledge of how the motifs look. We consider several approaches to Motif Finding that range from infeasible to fast but inaccurate.

   Implementation: Implement the brute-force-median-string algorithm and the branch-and-bound median string algorithm described in chapter 4. Also implement the Greedy Motif Search algorithm. Assume that the input sequences will be written in the DNA alphabet.

The input to a "Motif Finding Algorithm" should be a list of sequences, and an integer parameter, w, denoting the width of the motif the algorithm is looking for. The output of the algorithm is another set of sequences, each sequence being w letters long.

A popular format for multiple sequence data is the Fasta format. Poke around the biojava code to see how you can read Fasta format without actually writing the fasta reading code. While Fasta is an easy format, there are other formats that aren't so simple and being able to reuse someone else's code will save a lot of time.

Your test cases should be fairly small and simple so that they can run in a reasonable amount of time. It will be inconvenient for you to hardcode your test cases into each algorithm class—you can create a "unit test class" that does nothing but run a number of different test cases on the various algorithms you've implemented.

3. **Greedy algorithms and Genome Rearrangements:**

Background: The discovery that many species with similar genomes differ in gene ordering has lead to the study of the genome rearrangement problem. Research in this area has lead to the discovery of "evolutionary hotspots" between human and mouse as well as metrics for phylogenetic tree reconstruction (a topic explored in Chapter 10). A common formulation of solving genome rearrangements is NP-complete, and so greedy approximations are frequently used to get suboptimal answers in a reasonable time.

Implementation: Implement the pseudocode for SimpleReversalSort and BreakPointReversalSort (but read carefully the paragraph after the proof of the lemma, since it demonstrates a potential bug in the BreakPointReversalSort algorithm!). The input to your program should be a permutation of numbers $1 \ldots n$. The algorithm should return all of the permutations that it goes through in order, with one line printing after each reversal. Design several test cases, as BreakPointReversalSort is a simple algorithm, but surprisingly easy to get wrong.

4. **Pairwise Alignment Using Dynamic Programming:**

Background: Dynamic programming provides a framework for understanding DNA sequence comparison algorithms, many of which have been used by biologists to make important inferences about gene function and evolutionary history.

Implementation: Implement the longest common subsequence (LCS) algorithm described in section 6.5. Given two sequences, your program should output a dynamic programming table or grid, and a final alignment. The input to your algorithm will be two sequences. The output will be the LCS of the two strings, but this is actually kind of tricky. The LCS is simply a string of characters, but it should also be able to give the coordinates of each string in the LCS to each of the input strings.

Next, implement one of the other dynamic programming algorithms: global alignment, local alignment, alignment with affine gap penalties, or the linear space alignment. Each of these should take two sequences and a scoring matrix.

Note that tools like Biojava have some classes already built for these types of problems. You are welcome to use anything that biojava has implemented, *except* for the algorithm itself.

5. **Multiple Alignment:**

   Background: Multiple alignment of more than two sequences using the dynamic programming alignment algorithms that work for two sequences ends up in an exponential algorithm. Several heuristics have been proposed.

   Implementation: Implement the dynamic multiple alignment algorithm for n DNA sequences, where n is a parameter. Accept a scoring matrix as an input, but if one is not supplied use the following method to optimize for the maximum number of shared nucleotides at an alignment position: score A,A,A = 3, score A,A,T = 2, score A,C,T = 1.

   Next, implement the more efficient greedy pairwise multiple alignment algorithm.

6. **DNA Sequencing Using Graph Algorithms:**

   Background: Graph algorithms can be helpful in the problem of measuring the sequence of a piece of DNA. Typically, a long DNA segment is broken into pieces, the sequence of each of the pieces is measured, and then the larger segment's sequence is calculated. One sequencing technology that isn't used very much but is still instructive to study is Sequencing By Hybridization, covered in class and in Chapter 8 of the book.

   Implementation: Given a (relatively short) DNA sequence, implement an algorithm that can generate a list of all l-tuples in the sequence, and from that list, the overlap graph. Then implement another algorithm that calculates an Eulerian path through the overlap graph.

   Again, you are welcome to use existing software packages for graphs, but NOT one to solve this problem.

7. **Clustering implementation:**

   Background: Many of the new biological experimental techniques generate vast amounts of data. When viewed as a whole these data can be perplexing, however they are more sensible when alike data is clustered into groups.

   Implementation: Implement Lloyd's k-means clustering algorithm, as well as the Progressive Greedy k-means algorithm (on page 348 of the book). As a test case, use the points listed in figure 10.1, or a similar input.

8. **Clustering comparison:**

   Background: In addition to implementing clustering, it is important to be able to use and compare existing algorithms.

   Implementation: For this problem, I'd like you to find libraries or implementations for at least 3-4 different clustering methods, and a set of different types of challenging inputs. (Note that this project will require some research online, but probably a bit less pure implementation.) Find a way to compare and contrast the different methods on the data sets, so as to illustrate the good and bad aspects of each of the clustering algorithms you are examining. For your final submission on this one, I'd like both your code to actually run all of these (so that I can do it as well), and also an overview document describing those strengths and weaknesses. Essentially, your final product should be something that I could hand to students as an overview the next time I teach this class, after they have learned the basic clustering algorithms/pseudocode.