Suffix Trees for DNA analysis

Due date: October 29, midnight (Thursday).

About the Lab

Your BIO 441 partners need to find a variety of interesting things in the DNA fragments they are studying. Their needs can be met by a proper implementation of suffix trees, tuning the suffix trees for the specific problem they are trying to solve, and providing a working program (complete with some basic UI, as requested by your partners) that performs the requested searches and outputs the information.

The overall timeline for the lab is presented here:

Time	CSC 448	CHEM 441
October 15 lab	Discuss requirements	
October 20 lab	Work on requirements and design	
October 20 – October 27	Software development	
October 27 lab	Work on testing	
October 27 – October 28	Finishing software/ testing	
October 29 lab	Software delivery	Software use

Lab Assignment

Your BIO 441 partners must discover information in the DNA fragments they are working on. While the specific details of what they need will be provided in the requirements they give you, they are essentially searching for certain types of repeated sequences.

Your goal in this lab is to

- Implement suffix trees
- Use the suffix tree implementation to develop software that discovers the information requested by your BIO 441 partners.

Implementing Suffix Trees.

In order to help your BIO 441 partners with their project, you need to essentially develop a library for working with suffix trees.

NOTE: You must build the suffix tree implementation from scratch, using only the functionality available to you in the standard libraries of the programming language of your choice. **Use/Modification of existing Suffix Tree implementations is considered cheating in this lab.**

You have two tasks: (a) determine the appropriate structure of the suffix tree node, and (b) implement the procedures for creating and searching the suffix trees.

Suffix Tree Structure. Figure out what data needs to be stored in both the internal and the leaf nodes of the suffix tree. Please note that that actual node structure may vary from application to application (e.g., if you need to use suffix trees later in the course, you may need to alter the node structure), and you are allowed to make the contents of the tree specific to the assignment at hand. At the same time, make sure that your code can be easily ported to a different suffix tree node structure.

In your design, you are allowed to assume that you are dealing with fixed alphabets ({ A,T,C,G} plus whatever extra characters you need, or the amino-acid alphabet. What this means is that you can set the size of the `children` array in the `node` of a suffix tree to a preset size.). You **may** need to be able to build suffix trees in **both** alphabets, so your implementation may also need to (re-)use the nucleotide sequence-to-amino acid sequence translation.

Procedures for working with a suffix tree. While this lab has no specific requirements for how your suffix tree implementation works, the following is a good starting point for the functionality needed to be implemented:

1. **Core operations:**

(a) create an empty tree

(b) create a new leaf node, append it to an existing node in a tree (input: node to append to, edge label, leaf label).

(c) split an edge by inserting a new internal node.

(d) insert a single suffix into a given tree (input: suffix, position).

2. Suffix tree creation procedure. For this lab you can use the naïve, $O(n^2)$ procedure for creating the suffix tree (the only improvement you are expected to implement is not storing edge labels in the tree).

3. Implementation of string matching: given a suffix tree and a query string, determine if the query string is found in the tree and report all occurrences.

Notes: The size of the string input cannot be limited. The actual inputs used in the assignment will be on the order of tens of thousands of characters. Your code needs to be able to handle strings up to hunderds of thousands of characters long. If traditional ways of managing strings that long in your programming language of choice fail (e.g., if this goes beyond the limitation of, say, Java's `String` data type) you must implement your own abstract data type for management of long strings. You can assume that all data given to your program will fit in main memory, and that your suffix trees will fit in main memory, but beyond that, no other size assumption shall be made.

Dealing with input

Your programs may need to take as input the data in the formats you have seen before.

1. DNA strings in FASTA format

2. (if necessary) the coding regions information in the GTF format (might not be needed for everyone)

3. extra DNA strings (patterns) in multi-FASTA format (FASTA file representing multiple sequences)

Specific instructions will come from your BIO 441 partners. The requirements may include the need to develop UI.

Submission Instructions

As usual, two sets of deliverables are needed: one for `handin` and one - for `Piazza`. The core deliverables are the code and two requirements documents: the original document you built, and the final version.

handin deliverables. The `handin` deliverables are: *source code*, *compilation/running instructions (README)*, *user documentation* and *both requirements documents*. Submit them using the following command

```
$handin dekhtyar 448-lab04 <files>
```

Piazza delivarables. The final working version of the software (executable) must be delivered to your partners via Piazza. Additionally, the user documentation and the requirements documents need to be available on Piazza as well.

Deadlines. There is only one deadline: submit everything by the end of the calendar day on Thursday, October 28.