PROGRAMMING ASSIGNMENT #3

Cpt S 471/571, Spring 2019

Due: **April 23, 2018, by 11:59pm (Firm deadline)** @ Blackboard Project #3 dropbox
(24 hour grace period allowed with 10% late penalty)

General Guidelines:

- For this programming project, you *are allowed to work in teams of size (up to) 2 each*. Teaming up with someone else is not mandatory but highly encouraged. Note that regardless of whether you decide to work in teams or not, the assignment will be graded strictly on its merit and the same grade will be given to all team members.
- Note: If you worked as a team in Project 2, I would like you to stay in the same team for this project as well, unless there is a compelling reason to change.
- Reproduction of source codes from online resources or other's people's assignments is *strictly not* allowed, and will be considered cheating. All the source code should be solely yours. You are allowed to use any standard library functions supported by the underlying programming language, but if that directly implements any of the functions posed in the question of the assignment then you should **not** use those libraries and instead write your own code.
- You are encouraged (in fact, *expected*) to use your own code that you wrote from Projects 1 and 2 for this project.
- Grading will be based on a combination of factors such as correctness, coding style, implementation efficiency, exception handling, source code documentation, modularity of design, and code modularity/reusability.
- Grading will be based on a combination of factors such as correctness, coding style, implementation efficiency, exception handling, source code documentation, and code reusability.
- Note that there will *not* be any project demo for this project. Therefore I should be able to run and test your code on my end. Usability therefore is important.

Submission: The assignment should be zipped folder and both the folder *and* the zip file name should have your name on it. The folder naming convention is as follows: Program3-X.zip (if you are working alone and if your name is X) or Program3-X_Y.zip (if you are working as part of a team and the member names are X and Y - in any order). The zip file is the one that should be uploaded onto the WSU Blackboard dropbox (learn.wsu.edu) for Project #3. Submissions not following this naming convention risk being not graded! So please pay close attention.
Submissions are due by 11:59pm PST on the due date. A 24-hour grace period is allowed although it will incur a late penalty of 10%. Submissions that are more than 24 hours late will be returned without grading.

- Note: If you are submitting as a team (of 2 people) then both of you should submit TWO IDENTICAL COPIES separately on Blackboard. Just make sure you add the information of your team participants in the COVER SHEET.

Assignment: READ MAPPING

CLASSROOM/LECTURE SCRIBE PDF

The goal of this programming assignment is to implement an algorithm and test it for the read mapping problem, which is defined as follows:

Given a *reference genome* sequence G of length N, and a set of m sequences (called "*reads*") each of length l, identify the genomic locus to which each read best aligns.

In practice, N is very large (~10^6-10^9characters), whereas the length of the individual reads is really small (~10^2 characters). However, if M denotes the sum of the lengths of all the m reads, then typically M tends to be anywhere between 10x and 100x bigger than N. (This factor is also sometimes called the *coverage* of sequencing.)

For example, if the reference genome's length N is 10^6 characters, and if the coverage is 10x, and if the reads are all of length 100 characters (l), then the number of reads m is given by: N x *coverage* / l = 10^6 x 10 / 100 = 10^5 reads.

Given the above setting, it is simply not practical to align every read to the entire reference genome in a brute-force manner. What is required is a fast and efficient way to first identify potential locations along the reference genome to which the read exhibits a good promise of aligning up, and then perform a more rigorous alignment computation of the read against those short-listed loci alone. The goal therefore is to map the reads to the reference genome, in such a way so as to drastically reduced both the number of alignment tasks and the size of each alignment task. This can be achieved using a combination of the suffix trees and alignment methods that you have already developed as follows:

********************** **ReadMapping: MAIN begin** ***********************

Step 1) (ConstructST)

Construct the suffix tree for G (reference genome) using McCreight's algorithm (i.e., your project #2 code). Before you do this, however, just change your old suffix tree code to add two fields to every node structure: { int start_leaf_index; int end_leaf_index; }. And while creating a new node (internal node or leaf), initialize these two new fields values to -1. Nothing else changes to your McCreight's code.

Step 2) (PrepareST)

"Prepare" the suffix tree (in order to answer queries posed in Step 3). I have explained below in function *PrepareST()* what is required to be done here in this prepare step. Please read on.
FYI: This step will use the DFS routine you should have already implemented for your suffix tree in project #2.

Step 3) (MapReads)

For i=1 to m do {

Step 3a) Let r_i be the i^{th} read. Let l denote the length of r_i.

Step 3b) (*FindLoc*)

Find the set L_i of all locations on the genome G that share a longest common substring of length >=x characters with the read r_i. For example, if the reference genome sequence is the string "***accg**__accg__*tact" and the read is "tac**accg**", then the longest common substring between the read and the reference is "**accg**", which occurs starting at two locations - from index 1 and index 5 along the reference genome. So the L_i for this read should be output as {1,5}, assuming x is 4 or less.

The implementation of this step will use the suffix tree constructed in Step 1. x is a parameter to the code, and in practice it can be calculated as a function of both the read length and the estimated error rate of the sequencing process that led to the generation of the reads. For this project, however, just use x=25 in all your experiments.

The proposed algorithm to do this "FindLoc" is elaborated **below**.

Note that the set L_i represents a candidate list of all indices j's along the reference genome G which are starting positions for the longest common exact match of length >=x characters between r_i and G. (This also implies, $1 \le j \le N-x+1$).

Let the number of identified indices along G for r_i be $|L_i|$.

Step 3c) (*Align*)

For each $j \in L_i$ {

i) Extract substring *G[j-l... j+l]*, where *l* is the length of the read. (Of course, make sure you handle boundary cases here - i.e., if j is at the beginning or ending parts of G, then you should correspondingly retrieve as many characters that exist in G without going out of bounds.).

ii) Perform a local alignment computation (using Smith-Waterman) between read r_i and *G[j-l... j+l]*. For the alignment, you can use following parameters: {m_a =+1, m_i=-2, h=-5, g=-1}. After computing the DP table, record two pieces of information corresponding to the computed Optimal Local Alignment: a) the number of matches (*#matches*), and b) the alignment length (*#alignlen*), which is nothing but the number of aligned columns in your final alignment (should be equal to #matches + #mismatches + #gaps). One simple way to calculate these two values (#matches, #alignlen) will be to simply call the optimal path traceback function (*without* doing a display of the path) and calculate the numbers from there. (PS: There is actually second, more efficient way that will allow you to calculate these numbers during the forward computation of the DP table itself.)

iii) Let *PercentIdentity = #matches/#alignlen*.

iv) Let *LengthCoverage = #alignlen / l* .

v) If (*PercentIdentity ≥ X% AND LengthCoverage ≥ Y%*) {

// that means, the alignment is of "good" quality. All we need to do now is to keep track of and output the best quality alignment.

// Note: Here, X and Y are user supplied parameters. By default, set X=90%, Y=80%.

Is this value of *LengthCoverage* > previously seen best value for *LengthCoverage* for this read from any other value of *j* seen so far?

If so, update the new value of LengthCoverage and also the record this as the "best hit". Basically, "best hit" = (j0,j1), which are the start and end indices of the substring on the reference genome sequence corresponding to this optimal local alignment.

}

} // end for j

Step 4) (Output)

Output the best hit calculated from Step *3c.v* as the hit for read r_i. Your output can be simply: <Read_name> <Start index of hit> <End index of hit>.

If no hit was identified for this read, output <Read_name> "No hit found".

} // end for i

********************** **ReadMapping: MAIN end** ***********************

///

Algorithm for FindLoc function (Step 3b):

The goal of this function is to find a longest common substring between an input read *r* and the reference genome G, and return all its starting positions along the reference genome. To do this, we will try to reuse as much of the function FindPath() that you wrote for PA2 as possible. The main steps of the method is as follows.

**************** **Algorithm for FindLoc function (Step 3b) begin** ***********************

// note: throughout this procedure you will never modify anything in the suffix tree. In other words, the suffix tree will be used as read-only.

0. Initializations:

```
    i) struct node *T = the root of the suffix tree.        // "tree pointer"
    ii) int  read_ptr = 1;                                  // "read pointer" (again, use 0 in your code).
                                                            // Update this pointer as you match up each consecutive
character along the read successfully.
                                                            // Don't increment it if you see a mismatch.
```

1. Starting at T, find a path below the node T in the tree that spells out as many remaining characters of r starting from read_ptr. This would be a simple call to the FindPath() routine starting at the root (T) and the starting index for comparison on the read (read_ptr).

2. Let say, after some number of successful character comparisons, a mismatch is observed and so the matching path terminates. There are two subcases here. The matching path could either terminate at the end of an edge (case A), or it could terminate be in the middle of an edge (case B). Either way, let u be the internal node last visited along the matching path. In case A, u will be the node at which this edge ends. For case B, u will be the node from which this edge spawns off. If case B, then decrease the value of read_ptr by the number of characters that successfully matched below u along the edge before you hit a mismatch - i.e., if you matched r characters successfully along an edge below u before you hit a mismatch along that edge, then read_ptr = read_ptr - r. This will effectively reset the read_ptr back to where it was right after u. (Note, for case A, you don't need to do this since the mismatch occurred right after visiting node u.)

3. If the string-depth(u) $\geq$ x and if the string-depth is the longest seen so far for this read, then store a pointer, called "*deepestNode"* to the node u. We will update this pointer in future iterations if need be.

4. Now, simply take the suffix link pointer from u to v. Set T=v, and then go back to step 1, and repeat the process until you find the next mismatching point and so on.

5. At some point you will exhaust comparing all characters in your read. That signifies the end of iterations. Exit out of the while/for loop (from steps 1-4).

Special note: For steps 2-5, you have two options - either implement this suffix link based approach (which will be cool), or implement a naive approach where you simply do a FindPath from every index in the read and starting all the way at the root (less cooler but at least can get it to work if you don't trust your suffix links).

6. Upon exiting the loop, go to the node pointed to by the most up-to-date *deepestNode* pointer. The suffix ids in the interval A[deepestNode->start_index] to A[deepestNode->end_index] is to be returned as the candidate list of genomic positions L_i for the read. (Now, there is a possibility that this node's path-label doesn't really correspond to the "longest" common substring between the read and genome, but if that happens it will only be slightly shy of the length in practice. So ignore this slight approximation in algorithm and use this algorithm.)

****************** Algorithm for FindLoc function (Step 3b) end *************************

//

Algorithm for PrepareST function (Step 2):

The goal of this step is to compute the leaf list corresponding to the each internal node and leaf in the suffix tree. The *leaf list* of a node is the list of all leaves under a node's subtree from left to right. For example, in the suffix tree corresponding to the string banana$ (see here for a picture) the leaf list for the root is [7, 6, 4, 2, 1, 5, 3]; leaf list for the node with path-label "a" is [6,4,2]; leaf list for the node with path-label "ana" is [4, 2]; and so on. Obviously, as you can see, if we were to store the entire list from every node explicitly, then the space complexity of the tree could become quadratic ($O(n^2)$). Instead, if A is the array containing all leaves (i.e., with only their suffix id information stored) in the left to right order of the entire suffix tree, then every node's leaf list becomes nothing but a simple *interval <start_index,end_index>* within this array. For example, in the suffix tree picture for banana$, look at the array A below the tree (note, its starting index is 1 and not 0) where A[1]=7 ... A[7]=3. Then, the leaf list for the root, represented in the interval form, is <1,7>; leaf list for the node with path-label "a" is <2,4>; leaf list for the node with path-label "ana" is <3,4>; leaf list for the leaf suffix 2 is <4,4>; and so on. In other words, this interval approach requires only 2 integers per node, regardless of the size of the corresponding leaf list.

In your PrepareST function, use the above interval approach to compute and store the leaf list for all nodes (leaves and internal nodes) in the tree that have a string-depth (i.e., length of their path-label) $\geq$ l. You can do this by simply modifying your DFS function (which you must have already written for your PA2) as follows:

****************** Algorithm for PrepareST function (Step 2) begin ************************

(Assumes that the node structure has been updated to contain two additional fields: { int start_index; int end_index; } and initialized to -1.)
// This function will calculate the leaf lists of all the nodes with string-depth at least x while simultaneously populating array A (the list of leaf suffix IDs from left to right)

1. Create an array A of size n (string length of input + 1 for $), and initialize content with -1.

2. Initialize a global integer variable *nextIndex* to the start of the A array (i.e., in all my pseudocodes I have been using start indices as 1. Please initialize this to 0 in your real code.) This variable represents the next index in A which has to be populated.

2. Call DFS_PrepareST(root, A); // where root is the root of the suffix tree of the reference genome

```
DFS_PrepareST (struct node *T, int A[]) {

   if (T==NULL) return;

   if (T is a leaf node) {        // case: T is a leaf node
         A[nextIndex] =  suffix ID of this leaf node;
         if(T->stringdepth ≥x)  {
            T->start_index = nextIndex;
            T->end_index = nextIndex;
         }
         nextIndex++;
         return;

   }

   //case: T is an internal node

   For each child u under node T in the left-to-right order {
         - DFS_PrepareST(u, A);   // recursively visit each child of the current internal node, from left to right.
   }

   // the above step would have computed the leaf lists for all of T's children. Now its time to set the leaf list
interval for T. But do that only if T's string depth ≥ x.

   if(T->stringdepth ≥ x)  {
         Let u_left = T's first child (note: same as leftmost child).
         Let u_right = T's last child (note: same as rightmost child).
         T->start_index = u_left->start_index;
```

```
        T->end_index = u_right->end_index;
    }

} // end DFS_PrepareST (this is going to be a modified version of your old DFS code)
```

****************** Algorithm for PrepareST function (Step 2) end *************************

//

Input/Output specifications:

How to call the program?
$ <read_mapper executable> <FASTA file containing reference genome sequence G> <FASTA file containing reads> <input alphabet file>

Other Parameter files:

- Alignment parameters.config file (same that you used for your PA1, with match, mismatch, opening gap and continuing gap penalties).
- Optionally, you can have another parameters file in which to specify the value for l or you can even use the same alignment parameter file if you want for this purpose. In any case for all your experiments just use l=25.

What all data should your program report?

- Input stats: Length of the reference genome; the number of reads in the input;
- Main result: The top mapped hit for each read (as per instructions in Step 4 of the main function). Please output this into a separate file called "MappingResults_*.txt", where the * identifies which input read sequence file you used.
- Timing statistics: Total time for the program execution. This should also be broken down into the time taken for each of the main steps of your main function - viz. ConstructST, PrepareST, MapReads, Output.
- Other stats: What is the average number of alignments performed per read?

Test inputs:

Simulated Input: Peach reference genomic sequence (size stats) Simulated reads from peach genomic sequence (size stats) DNA alphabet

Testing: For testing, please use the simulated read set input first, so that you will be able to check correctness. These reads were generated by a sequencing simulator that I wrote, which basically samples different part of a given reference genomes, makes a few random changes as per the error rate and outputs the reads. The lengths are also slightly varied, but they are all very close to 100 characters each. Each read is named with the information of where it was derived from the reference genome. For instance, ">Read_1_from_Peach_reference.fasta_AT_712655" means that that read maps to location G[712655..712757], since that read has 103 characters. Initially, for testing, you can use small subsets of these reads, test them before going larger scale. There are a total of 500K reads in this input. Include the mapped output for the largest possible run (preferably the entire 500K set) in your submission.

After you have tested out with the synthetic input, then do one single run, on the other oinput and submit the answer.

Report:

In a separte Word or PDF document, report the following:

1. System configuration: CPU used, Clock rate, RAM, Cache size (if you know it).
2. Summary of results:

   Tabulate the performance statistics of your results output as per instructions above under the bullet "What all data should your program report?". Do this separately for each of the two inputs (Synthetic input and Real world input). Basically this section should contain all statistics except the mapping results output by your program.

   Justification: Also provide a brief justification to explain your observations against expectations.

-

CHECKLIST FOR SUBMISSION:

___ Cover sheet

___ Source code

___ A helpful readme file saying how to compile and run the code

___ Output all performance statistics and mapping results file, in a separate file, for the Peach test input.

___ Report (Word or PDF)

___ All the above zipped into an archive. The Zip file name should have your name in it.

Note: There will not be any time for demos for this project. Therefore, I need all your output files as part of your submission for this project and I will be grading based on your source code, your output files and your report.