Follow up to Minerva: An Alignment and Reference Free Approach to Deconvolve Linked-Reads for Metagenomics

Waris Barakzai, Zihao Gary Wu

May 2019

1 Abstract

In this project, we propose using Ukkonen's Algorithm for Linear Time construction of a suffix tree for the purpose of deconvolving Linked Reads. We used suffix trees to find overlaps in read sequences between barcodes, unsuccessfully. Somewhere in the abstraction to store multiple documents, we've run into a bug that we haven't been able to solve although, we do have a correct implementation for single strands of DNA. Running on smaller datasets has proved successful, although results are meaningless in application as smaller datasets are much less likely to create overlaps in suffix trees.

2 Introduction

In Danko et. al.'s previous publication, Minerva: An Alignment and Reference Free Approach to Deconvolve Linked-Reads for Metagenomics, an approximate solution to the problem of deconvolving linked-reads was put forward. Next generation sequencing technologies, or long-read sequencing technologies have become the primary sequencing technology in de novo assembly and in metagenomics but due to technological limitations, have high base error rates and substantially higher costs compared to short-read sequencing technologies. Long reads eliminate the issue faced with short reads which is the ability to sequence an entire fragment (much longer percentage of an organisms DNA) of DNA whereas short reads are sourced by shearing fragments of DNA into much smaller pieces often fewer than 100 base pairs long. Recently though, the innovation of Linked-reads, which have 3' barcodes tagged to short-reads have revived interest in the viability of short-reads in metagenomics. By tagging these reads though, there is convolution in the ordering of data. Deconvolution is an algorithm based process to reverse the effects of this convolution and for Linked-reads, this means algorithmically reconstructing the fragment from which short-reads came from. Ideally, linked-reads easily solve this issue

if there is a 1:1 mapping of barcode to fragment, although often times the ratio can reach 20:1 barcodes per fragment. For further information on Linked-read technologies we refer to Danko et. al (2018).

In our follow up to Minerva, we propose using Ukkonen's Algorithm for linear time and space construction of a suffix tree to improve the results of Minerva. Minerva currently uses Minimum Sparse Hashing to store k-mers, or discrete length "cut-ups" of short reads and in the overlap checking step, checks if k-mers from one barcode exist in another barcode. By using suffix trees to store all the reads in a barcode, we believe that there is a higher possibility of detecting overlaps, since k-mers are discrete whereas storing reads in a suffix trees makes the overlap phase essentially a subsequence searching problem. So essentially, given a barcode with a set of DNA reads, we construct the suffix tree of the combined reads in a barcode. We then proceed to search for k-mer overlaps in other barcodes to begin creating "enhanced" barcodes. Simply put, by finding overlaps between barcodes using suffix trees, we can begin clustering barcodes and reasonably assume that they originate from the same fragment.

3 Suffix Trees

3.1 Introduction

A suffix tree is a data structure that contains all the substrings of a given text. It can be used to solve problems like finding the longest common substring and searching for patterns in DNA.

3.2 Definition

The suffix tree for string S of length n is defined as a tree such that:

- The tree has exactly n leaves
- Every internal node has at least two children
- No two edges starting out at the same node can have string labels beginning with the same character
- the string obtained by concatenating all the string labels on the path from the root to leaf generates suffix S[i...n] for i from 1 to n.

3.3 Ukkonen's Algorithm

Ukkonen's algorithm is a linear time algorithm that utilizes edge label compression to achieve linear space. In order to build a suffix tree for a string S

of length n using Ukkonen's algorithm in linear time and space, we will loop through the string once. A phase in the algorithm is defined as inserting suffix S[i+1] to the existing suffix tree. For i from 1 to n, we need to follow these three rules:

- Rule 1: For phase i+1, if S[j...i] ends at last character of leaf edge, then append S[i+1] to the end.
- Rule 2: For phase i+1, if S[j..i] ends in the middle of the edge and S[j..i+1] does not overlap with one of the existing edges around the node, create a new edge with label S[i+1]
- Rule 3: For phase i+1, if S[j..i] ends in the middle of the edge and S[j...i+1] overlaps with one of the existing edges around the node, do nothing.

Instead of storing the actual strings in the suffix tree which can easily cost quadratic space, substrings are stored as a pair of indices that represents the start and the end in the original string, where the end index is stored as a global variable. For each iteration, we increment global end and update all the current leaves in O(1) time. Besides, such representation achieves linear space which is crucial for our application. To make updates done in rule2 more efficient, for every node with label S[i][i+1..j] where S[i] is a single character and [i+1..j] is the substring of S(could be an empty string), we store a suffix link from such node to another node with with label S[i]. If S[i+1...j] is empty, then the suffix link is the root. By storing suffix links, we can traverse between nodes in O(1) time. In addition, we also need to keep track of active point, active position, active node, active edge and active length. The definition of these variables are described below:

- Active point: Consists of active position, active node and active edge.
- Active position: A point where next phase or next rule extension starts. Active position always starts from the root.
- Active node: The node where active point starts
- Active edge: The edge we use around the active node. Each edge is represented by its start index around the node.
- Active length: How far we go in active edge

The following rules are applied if the corresponding conditions are satisfied:

1. If case = rule 3, increment active length by 1 if active length is less than the length of the edge
2. If case = rule 3 and active length gets greater than the length of the edge, change the active node to the next internal node, active edge is none, active length is 0
3. If active length is 0, start from the root

4. If case = rule 2 and active node = root, increment active edge by one(pick the next edge around the root) and decrement active length by one.

5. If case = rule 2 and active node is not the root, change the active node to the node that suffix link is pointing to.

By keeping track of the variables and following the rules described above, we can construct a suffix tree in n iterations, with each iteration taking O(1) time.

4 Implementation and Difficulties

In order to process such a huge dataset, we decided to implement a more generalized suffix tree that worked for multiple documents by using different special characters as terminators for each read in the barcode. Since there are only fourty to sixty number of reads in each barcode, the special characters used in our implementation can be obtained from the ascii table. Basically, we concatenate each read in the barcode with terminators like !, @, etc. Whenever we reach a special character, we then convert our implicit suffix tree into a complete suffix tree. By constructing the our suffix tree this way, we managed to save space by building one suffix tree instead of building fourty to sixty suffix trees for each barcode.

4.1 Difficulties

1. Keeping track of the active point. 2 specific cases of Ukkonnen's algorithm proved to be especially difficult to work with. The first is the case of having an active node with no suffix link which requires that the active point be reset to the root traverse to the active point of the next remaining suffix to be inserted. The other case was for when skipping through the tree by using suffix links, the active position can be larger than the length of the active edge, so in that case the suffix of the suffix must be traversed much like the previous case from the active point to the correct active point down some path.

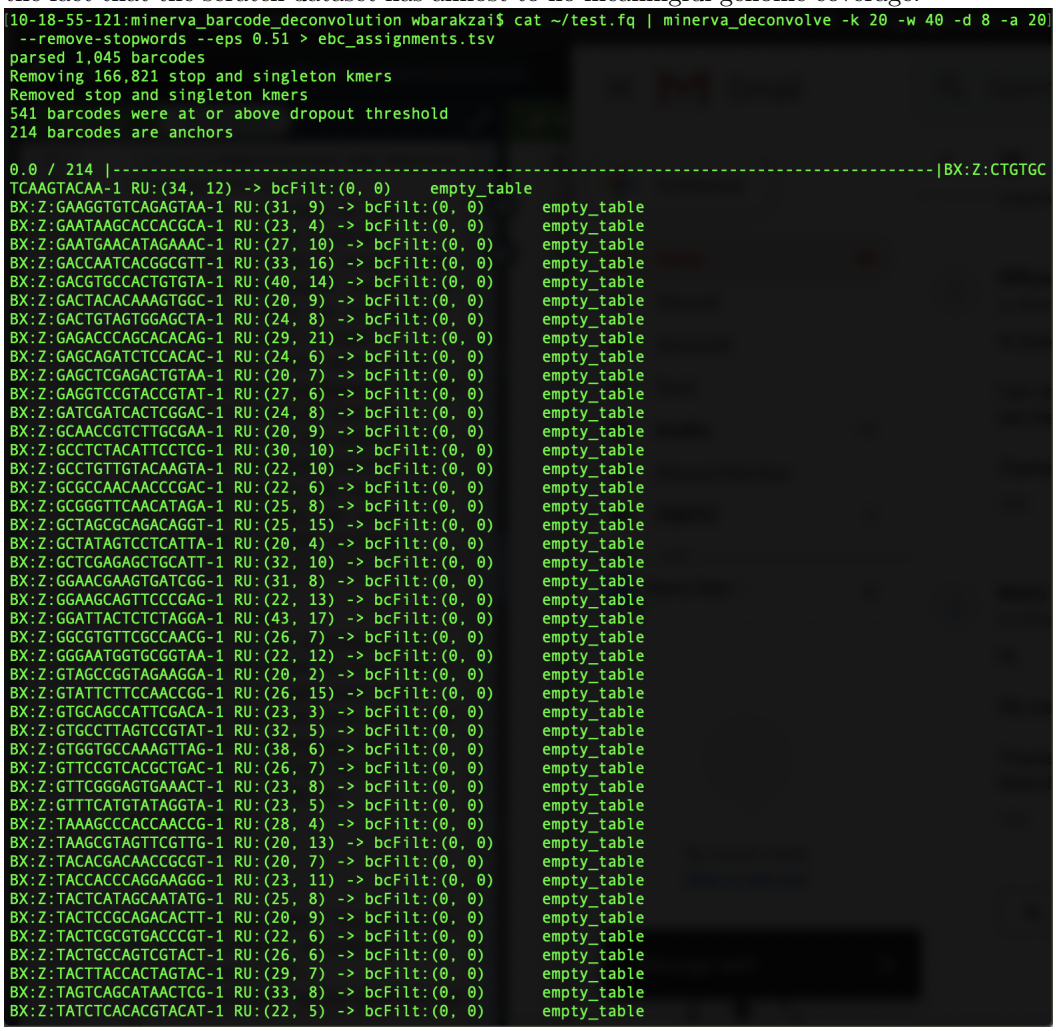
2. Debugging. In order to replicate the errors experienced in implementing the above two cases, Python's lack of a standard debugger required us to work through examples very slowly with large strings in order to recreate the errors in building the tree.

5 Data Access

All code from this project is available at
https://github.com/warisBarakzai/minerva_barcode_deconvolution

6 Applications and Results

We are currently trying to see if simply concatenating reads without having unique ends since a suffix tree stores all suffixes of a string, each read would still be encoded although there will be a waste of space. That being said, via the attached screenshot we have been able to confirm that the implementation of the suffix tree has worked, although results are meaningless due to the fact that the scratch dataset has almost to no meaningful genome coverage.



6.1 Proof of correctness on single string (via example that uses all cases of tree development)

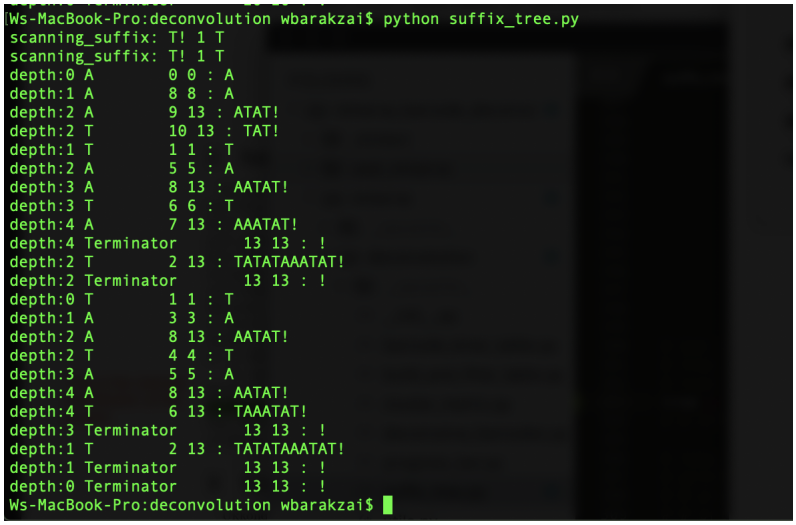


Figure: Output for depth first search from root to confirm structure and string encoding of tree.

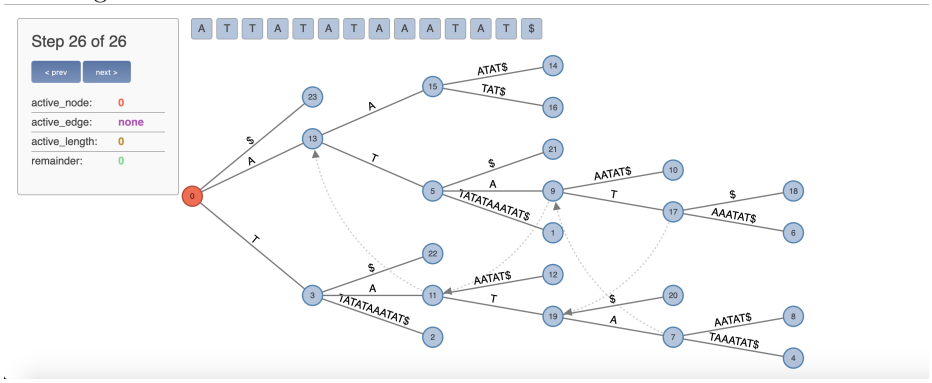


Figure: Visualization of string for evaluation

7 References

Most of the information of Ukkonen's algorithm in this report are quoted or paraphrased from Gusfield's book. We also watch the tutorial made by Tushar Roy and read the instructions on how to build a suffix tree on stackoverflow.

1. Gusfield, Dan. "Introduction to Suffix Trees." Introduction to Suffix Trees, 1997, web.stanford.edu/ mjkay/gusfield.pdf.

2. Danko, David C., et al. "Minerva: an Alignment- and Reference-Free Approach to Deconvolve Linked-Reads for Metagenomics." Genome Re-

search, Cold Spring Harbor Lab, genome.cshlp.org/content/early/2018/12/18/gr.235499.118. full.pdf+html.

3. Ridley, Nathan. "Ukkonen's Suffix Tree Algorithm in Plain English." Stack Overflow, stackoverflow.com/questions/9452701/ukkonens-suffix-tree-algorithm-in-plain-english.

4. Roy, Tushar. "Suffix Tree Using Ukkonen's Algorithm." YouTube, YouTube, 6 June 2015, www.youtube.com/watch?v=aPRqocoBsFQt=1224s.

5. Demaine, Erik. "Session 16: Strings." MIT OpenCourseWare, Massachusetts Institute of Technology, MIT, ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-851-advanced-data-structures-spring-2012/lecture-videos/session-16-strings/.