Algorithms in Molecular Biology
Final Project Technical Report

Dimakopoulos Vasilis, Psallidas Kyriakos

[1]Department of Computer Science and Telecommunications, National and Kapodistrian University of Athens

Abstract

This report presents a replication attempt in C++ of the study by Phoophakdee Benjarath, and Mohammed J. Zaki., on Genome-scale disk-based suffix tree indexing. Our goal was to replicate the four-stage of researchers from scratch and evaluate their original findings. However, we encountered challenges due to the limited guidance for its implementation. As a result, instead of a one-to-one replication of the study, we decided to implement our interpretation of the algorithm, this solution while functional and able to replicate the algorithm's main stages, had its own limitations and was unable to be utilized for genome-scale sequences. This experience showcases the importance of thoroughness in reporting research to ensure the reproducibility and reliability of findings.

1 Introduction

Efficient identification and retrieval of biological information encoded in the alphabets of DNA and RNA are crucial in genomics, making efficient string searching and matching solutions highly important for the field. With the cost of sequencing technologies rapidly decreasing, the data in genomic databases, such as Gen-Bank has exponentially increased [1]. Thus, the demand for efficient solutions has intensified. The suffix tree data structure provides this needed functionality by representing all the suffixes of a sequence and consequently allowing for important genome analysis tasks in databases, such as exact or homology sequence matching, pattern searching, and sequence alignment [2]. Apart from the advantages discussed prior, suffix trees also allow for O(n) space efficiency, where $n = sequence\ length$ by not repeating shared prefixes among the suffixes. However, constructing suffix trees for genome-scale sequences poses a significant challenge in terms of optimizing memory management efficiency while maintaining low algorithmic complexity.

The reference paper titled: "Genome-scale Disk-based Suffix Tree Indexing" by Poophakdee Benjarath and Mohammed J. Zaki [3] was presented at the 2007 ACM SIGMOD international conference on the management of data. It introduces an algorithm called *TRELLIS* that enables the construction of suffix trees for sequences up to 3 Gbp in size. The *TRELLIS* algorithm builds upon previous disk-based suffix tree construction methods by a combination of three factors. First, by incorporating variable-length prefixes it addresses the partition skew problem that arises when the unique (A+T)/(G+C) ratio of different species is not taken into account. Second, it achieves the 3Gbp scalability mentioned before by utilizing a parameter named *(t)* at its core that ensures each stage of the algorithm fits in provided memory. Finally, *TRELLIS* in its final optional stage allows for the reconstruction of suffix links for the trees produced, which allows for significantly sped-up query times. The paper claims that *TRELLIS* surpasses previous disk-based suffix tree algorithms in performance. It is capable of indexing the entire human genome using only 2GB of memory in a mere 4 hours, while the retrieval of suffix links requires an extra 2 hours. This statement caught our attention and its validation was the deciding factor for the replication analysis. However, as it will be explained in depth in the following sections this proved to be challenging due to the variable and sometimes sparse levels of detail of the algorithm's main phases. As a result, our attention shifted towards implementing our comprehension of the algorithm's techniques to assess the methodology as described in the original paper. In the "Background & Related work" section, we provide an introduction to the suffix tree data structure, which we deem an essential preliminary for the paper. Following that, we then explain the methodology used by the researchers for the *TRELLIS* algorithm. Next, we present our own implementation of the algorithm, including pseudocode where we believe it assists a deeper understanding, in the "Methodologies" section. The "Evaluation" section is dedicated to discussing the reasons behind any deviations from the reference methods and our implementation journey. Finally, in the "Conclusion" section, we present our overall findings regarding the core methodology of the reference paper and our implementation.

2 Background & Related work

A suffix tree, as mentioned in the previous section represents all the suffixes of a sequence. Consider for example a DNA sequence: *ATGCGATG*. A suffix is a sub-string that starts from any point in this sequence and continues to the end. To ensure that the end of each suffix is unique, particularly when substrings repeat in the input sequence, a unique character is added, represented here as *"$"*, to the end of the sequence. By definition *ATGCGATG$* has as many suffixes as characters, from the longest to shortest, the

suffixes are showcased in (Table 1) below. To represent

Suffix	Length	Index
ATGCGATG$	9	0
TGCGATG$	8	1
GCGATG$	7	2
CGATG$	6	3
GATG$	5	4
ATG$	4	5
TG$	3	6
G$	2	7
$	1	8

Table 1: *Suffixes with their index and length for the sequence: ATGCGATG$*

the above information with a suffix tree, we need to first define its components. The tree begins with a **root node** that represents an empty string. Below the root, we find **internal nodes**. Each internal node has at least two children where the branches diverge, denoting different suffixes. The branches of the suffix tree are labeled with substrings of the input sequence called **edge labels**. At the end of each branch, we have the **leaf nodes**. These represent the starting index of a suffix in the input sequence. **Path labels** correspond to the string obtained by concatenating all edge labels from the root to a given node, if that node is a leaf node, the path represents a suffix of the input sequence, thus a sequence of length n has n suffixes, and thus its suffix tree n leaves. A final optional component to the suffix tree data structure are **suffix links**, pointers from any internal node representing a path from root $\alpha\gamma$ to another node representing a path from root γ, where α is a single character and γ is a string, node $7 \rightarrow$ node 9 in this case. All of these key components are illustrated in the suffix tree of sequence: *ATGCGATG$* in (Figure 1) below.

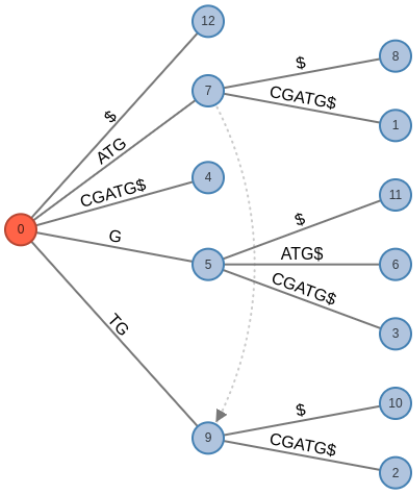

Figure 1: *Suffix tree of sequence: ATGCGATG$*

Source: Created with Visualization of Ukkonen's Algorithm

The two widespread algorithms for suffix tree construction are McCreight's and Ukkonen's algorithms [4, 5]. The two algorithms differentiate themselves by their approach to building the suffix tree. McCreight's algorithm constructs the tree by iteratively adding the suffixes of the input sequence in descending order of length. Thus this method necessitates a sequence of predefined length. On the contrary, Ukkonen's algorithm has the on-line property, which allows it to construct the suffix tree progressively, character by character. This approach allows the flexibility of building the tree until the end of a sequence that its length does not need to be predefined beforehand.

In relation to the four primary stages of the *TRELLIS* algorithm, we believe it's essential to present the researchers' methods for an informed comparison with of our subsequent methodology. *TRELLIS* takes as input a genome scale sequence and the **parameter *t***. This parameter works as a threshold for multiple stages to ensure that their execution fits in memory, it is calculated by the equation:

$$M \geq \frac{n}{4} + ((1.4 \times 40) + 16)t \rightarrow t \leq \frac{M - \frac{n}{4}}{72} \quad (1)$$

where M is the allowed memory in bytes, $\frac{n}{4}$ is the bit encoded input sequence size and finally the values 40, 16, and 1.4 correspond to the maximum number of bytes required for internal nodes, leaf nodes, and the approximate count of internal nodes that must be stored in memory during the algorithm in the researchers' approach, respectively. The researchers utilize a $t = 10^6$ for the human genome.

Variable length prefixes phase takes as input the sequence and the t parameter and outputs variable length prefixes to address the issue of skewed partitioning caused by the non-uniform distribution of characters in DNA sequences, where some partitions become very small while others are too large to fit in memory. The process begins by defining a set $P = \{\}$ that will contain the identified variable length prefixes. All prefixes added to P must have a frequency $\leq t$. Additionally, we define L_i and EP_i to track the length of the longest prefix and the set of prefixes requiring further extension in the next scan, respectively. Initially, $L_0 = 0$ and $EP_0 = \{""\}$ a set with an empty string. During each scan i, prefixes that meet the frequency threshold are added to P, while the remaining prefixes are added to EP_i for extension before the next scan. If necessary, L_i is adjusted to satisfy the inequality:

$$|EP_i| \sum_{j=1}^{L_i - L_{i-1}} |\Sigma|^j \leq t \quad (2)$$

where Σ represents the alphabet set of the input sequence. The final output is the set of variable length prefixes $P = P_0, P_1, ..., P_{m-1}$, which ensures that the final merged trees can fit within the available memory.

Partitioning phase again takes as input the sequence and the t parameter. Its purpose is to partition the sequence into sections, with each section's suffix tree fitting in memory. A suffix tree has as many leaves as the characters in the

input sequence and t is calculated for a tree with t leaves to always fit in memory. Thus the input sequence is split into $r = \frac{n+1}{t}$ partitions. However, all the partitions except the last do not contain the unique ending character: "$" and thus there is no guarantee that the leaves of each partition suffix tree are equal to t.

To overcome this the researchers utilize Ukkonen's suffix tree construction algorithm with its on-line property to construct character by character the partition suffix tree and if necessary read characters from the next partition until t leaves are created. For every partition tree created, the researchers further split it into multiple prefixed suffix subtrees, each containing only paths in the partition suffix tree that begin with each prefix in P. These subtrees are finally saved to disk.

Merging Phase takes the prefixed suffix subtrees with the same prefix as input. It iteratively merges these subtrees until the complete prefixed suffix tree is obtained for each prefix in P. During the merging process, only two trees are kept in memory: the merged tree constructed so far and the next prefixed suffix subtree to be merged. To merge the trees, the researchers traverse both trees from the root to the leaves and consider two cases. In the first case, if there are no edges representing the same substring, all children can be added to the left node. In the second case, if there are edges with a common prefix, a new node is created in the merging tree. This new node inherits the children from both original nodes.

Suffix Link Recovery Phase is the optional final phase of the *TRELLIS* algorithm. Despite Ukkonen's algorithm being utilized for the construction of the suffix tree, which generates suffix links, many of these links do not survive the merging phase. Thus, the researchers opt to eliminate these original suffix links and recover the suffix links for all internal nodes in the final prefixed trees for each prefix.

This recovery process is executed by traversing each prefixed suffix tree in a depth-first manner. For every internal node, its parent and the parent's suffix link are identified. As the parent's suffix link might reference a node in another tree, that tree is loaded into memory, allowing the algorithm to 'skip/count down' from the parent link node to identify the suffix link of the current node. This process is recursively executed for all children of the current node, recovering all suffix links. Finally, the suffix tree with the newly recovered suffix links is saved to disk.

3 Methodologies

3.1 Algorithm stages overview

Before we begin the detailed explanation of our methodology, we believe it's important for the reader to have a visual outline of the algorithm's key stages. This includes a depiction of how these stages interconnect with each other and interact with both the memory and the disk, thus in (Figure 2) we provide a complete schematic representation. In order to maintain clarity we have decided to divide the researchers' partitioning phases into three distinct components: sequence partitioning, partition tree construction, and splitting into prefixed subtrees.

3.2 The algorithm's input

As mentioned, the input of the *TRELLIS* algorithm consists of a DNA sequence and a parameter denoted as t. For the main implementation of the algorithm, we selected the NCBI's Reference Sequence: *N_045512.2*, which represents the complete genome of the "Severe acute respiratory syndrome coronavirus 2 isolate Wuhan-Hu-1," in FASTA format. This genome has a length of $29,903$ base pairs (bp). We intentionally chose a relatively short sequence for testing purposes, as we did not utilize bit encoding for the input due to reasons that will be discussed further in the evaluation section.

The FASTA file contains a header line and the whole sequence is split into lines of equal length which varies between different FASTA files. Additionally, we observed that other FASTA files may contain lowercase characters {a,g,c,t,}. In order to convert the input file into a single-line string ready for the partitioning phase we utilize a function that skips the header line starting with the ">" character, then reads each line of the file and converts it to uppercase before appending it to the sequence string variable. Of course, we append the unique ending character "$" after the final line is read.

Regarding the parameter t, we implemented two distinct functions for Linux and Windows-based systems that is able to recover the total memory of the system with the *Windows.h*, *sysinfo.h* libraries and *GlobalMemoryStatusEx*, *sysinfo* functions respectively, but we were not able to restrict it in any way to a specific number of Gigabytes as reported in the paper. We used the identified system memory M to calculate t as described in equation 1. Another approach to replicate t as defined in the paper is to identify the relation between the input sequence size and the t parameter used in the reference paper. Since the researchers utilize the human genome of length 3 billion base pairs and an optimal $t = 10^6$ for $M = 2Gb$, then for any sequence of size n:

$$t = \frac{n \times 10^6}{3\ billion} \approx n \times 0.0003$$

This resulted in 3322 partitions for our input sequence and thus the algorithm runs very slowly. We also considered allowing t to be set by the user as input, but since t governs the memory the algorithm will utilize this can easily cause issues and terminate the algorithm. Finally, for testing/demonstration purposes decided to limit t to $n/30 = 996$ which produces 30 large-string partitions.

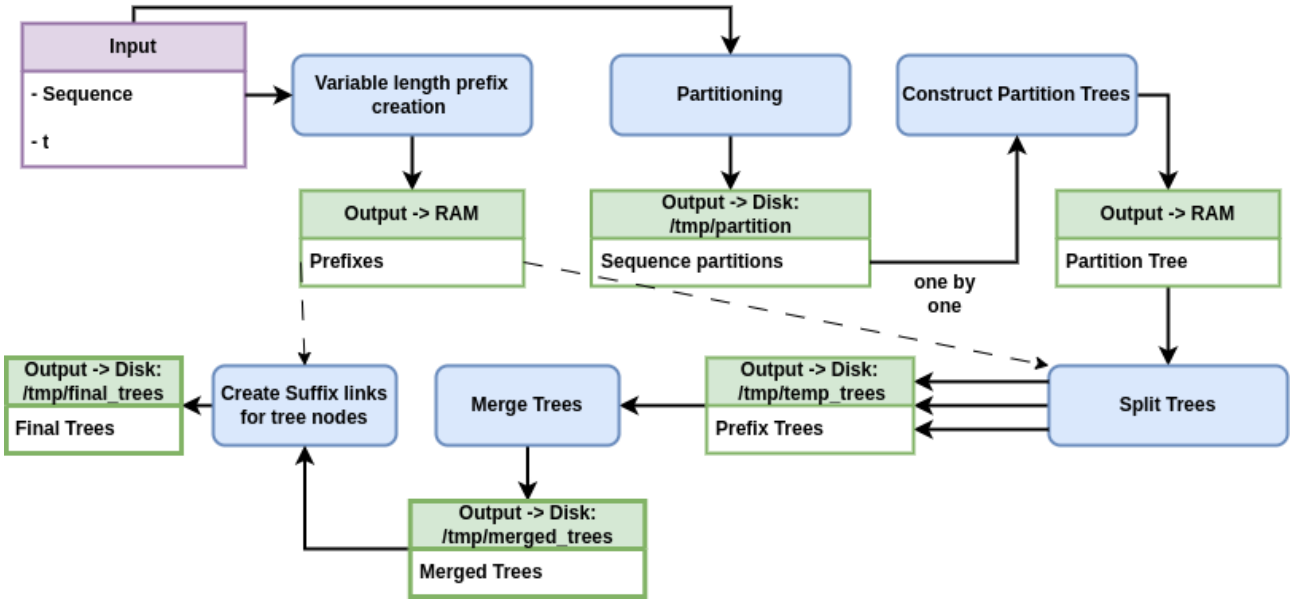

Figure 2: *Algorithm stages, replication pipeline*

3.3 Variable length prefix creation

Our implementation of the algorithm for this stage is the *runMultiPass* function, it begins by initializing an empty set EP_0 to hold prefixes that do not fit within the memory limit and need to be extended. Another empty set P is initialized to store prefixes of the input sequence denoted as:S that fit within the memory limit. The variable i is set to 0 to represent the current scan.

The algorithm then enters a loop that continues until EP_i is empty. Within this loop, another loop is initiated that continues until the condition $|EP_i| \sum_{j=1}^{L_i - L_{i-1}} |\Sigma|^j \leq t$ is met. This condition ensures that the next extension of prefixes in EP_i fit in memory. In this inner loop, each prefix in EP_i is extended by each character in the set A, G, C, T. For each prefix Pr in EP_i,we count how many time it appears and if its frequency in S is less than or equal to t, Pr is added to P and removed from EP_i. The variable L_i is then updated to hold the length of the longest prefix in P thus far.

After the inner loop, if the extension of prefixes in EP_i does not fit in memory, we begin a new scan with a new EP_i set. The algorithm continues until all prefixes that their final prefixed trees can fit within the memory limit have been generated and added to P and thus EP_i remains empty.

The pseudo-code for the variable-length prefix creation algorithm is showcased in (Pseudocode 1).

Regarding the way we count the prefixes in each scan, (lines 8-10 in Pseudocode 1), we have developed two distinct methods. The first method operates by performing multiple passes over the sequence. In each scan, it calculates the frequency for each prefix in EP_i until all prefixes have been processed. The second method performs a single pass over the sequence and calculates the frequency in the the current EP_i in one go. The multi-scan method achieves its functionality by comparing every prefix in EP_i at each position in the sequence and incrementing the frequency counter of the specific prefix if a match is found.

Algorithm 1: Variable length prefix creation implementation

Input: Sequence
Result: Set of prefixes of S that their final prefixed trees fit in memory

1 **Initialize:**
2 $EP_0 = \{""\}$
3 $P = \{\}$
4 $i = 0$
5 **while** EP_i *is not empty* **do**
6 **while** *Condition* $|EP_i| \sum_{j=1}^{L_i - L_{i-1}} |\Sigma|^j \leq t$ *is not true* **do**
7 Extend each prefix in EP_i by each character in $\{A, G, C, T\}$;
8 **for** *each prefix* Pr *in* EP_i **do**
9 **if** *freq(*Pr*)* $\leq t$ *in* S **then**
10 Add Pr in P and remove it from EP_i;
11 **end**
12 **end**
13 $L_i = \max\{|Pr|\}$
14 **end**
15 $i = i + 1$
16 **end**

Finally after the vector of variable length prefixes is completed, we keep it in memory to be reference for proceeding stages of the algorithm, specifically the partition tree splitting into prefixed sub-trees and creation of suffix links as illustrated in (Figure 2).

3.4 Sequence partitioning

Our sequence partitioning implementation is straightforward. The function *partitionSequence* works by dividing the input sequence, denoted again as S, into smaller subsequences, the length of which is equal to the parameter t, so that each partition's suffix tree fits in memory. Of course the last partition may be smaller than t, however this is not an issue.

The process begins with the initialization of a counter which tracks the number of partition files that have been created. It then checks for the existence if a temporary directory in disk to store the partitions exists. If this directory does not exist, the function creates it. Then the sequence S is partitioned by iterating from the start to the end of S, with each step incriminating by t. For each iteration, the function extracts a sub-string of S from the current index to t, storing this in file content. It then constructs a file name by appending the current file number to the string "partition_", the file content is then written to the file.

However, all partitions except the last one do not contain the unique character ending character: "$" and thus the suffix trees produced by them are not explicit and not guaranteed to have t leaves that fit in memory, to counter this issue we add "$" to the end of each partition and remove it at the merging phase, since it is a pseudo ending. Finally, the file number counter is incremented by one, and the process repeats until the entire sequence S has been partitioned. Now we have in a temporary folder in the disk all the partitions of the sequence whose trees fit in memory and we can easily read them by accessing their file, based on their unique file count number. The pseudocode of the described implementation is showcased in (Pseudocode 2).

Algorithm 2: Sequence partitioning

Input: Sequence, t
Result: Sequence partition files stored in the disk
Initialize:
file number = 0;
if *temp partition directory does not exist* **then**
    Create a temporary directory:
    "./temp_partition" to store the partitions
end
for *i from 0 to the length of the S with a step of t)* **do**
    file content = sub-string of S form i to t)
    file name = "./temp_partition/partition_" + file number + ".txt"
    Write file content + "$" to file name's location
    file number = file number + 1;
end

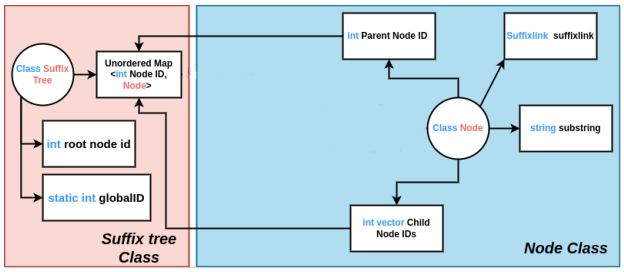

Figure 3: *Suffix tree & Node Class: member variables*

3.5 Partition tree construction

At this stage, our approach deviates the most from that of the researchers in the reference paper. We developed a suffix tree construction method that mirrors Mc-Creight's algorithm, which constructs the suffix tree starting from the longest suffix of a sequence down to the shortest. However, our method *addSuffix* does not utilize suffix links. Furthermore, we did not use edges to depict the sub-strings in the tree and pointers to reference the tree's nodes. The reasoning behind these decisions is detailed in the evaluation section. As illustrated in (Figure 3) below, we have created two classes to form our complete suffix tree data structure.

The suffix tree class has a root node id and a global static ID variable. This ID variable is crucial in the generation of unique keys for each node in each tree. The final member variable of this class is a hash map, which uses the unique node IDs as keys and stores the corresponding nodes as values. The second class is the Node class and it represents each node in the suffix tree. We have incorporated the sub-string representation of the edge directly into each node using a string member variable. To allow for traversal of the tree both from the root to the leaves and vice versa each node "knows" its parent node ID in the suffix tree's nodes hash map, and also the node IDs of all its children, which are stored in a vector for each node.

Regarding the suffix tree construction, our implementation iterates over each character in the longest suffix of input sequence S. Beginning from the root node, if there's no matching starting character in a child node, it creates a new node. If there is a match, it checks for a partial sub-string match with the node's sub-string, splitting the node if necessary. This process repeats for each character in the suffix and then starts over with the next longest suffix. Our implementation for suffix tree construction is showcased in (Pseudocode 3).

We employ this method to construct suffix trees for the partitions saved in the disk from the previous stage. When constructing a partition suffix tree we keep it in memory and proceed with it until the completion of the next stage, before loading the next partition and beginning its tree construction as showcased in (Figure 2).

Algorithm 3: Suffix tree construction

```
Input: Sequence
Result: Suffix tree
Initialize: tree root node, with id = −1
for each ordered by descending length suffix in S do
    Start from the root node of the tree
    for each character in suffix do
        if no child node starts with the current character then
            Create a new node with the remaining substring of the suffix
            Add this new node to the tree at that depth
            Return
        else
            If a child node starts with the current character
            Check for a partial match between the suffix and the child node's substring
            if a partial match is found then
                Split the child node into two nodes
                Update the tree structure to include the new node
            else
                Move to the next character in the suffix
            end
        end
    end
end
```

3.6 Split of partition trees into prefixed sub-trees

As described we begin this stage by having a partition tree in memory. Our goal is to locate nodes in the partition tree that match the prefixes in the set of variable-length prefixes: P. Then, we can split the partition tree into new sub-trees, each one with a root node sub-string matching (or being a prefix) of a specific prefix. To achieve our implementation utilizes a method: *splitTree* of the suffix tree class, that traverses the tree from the root to child nodes and appends all the nodes involved in building the path in the partition suffix tree that matches the prefix into a vector, by gradually removing the matching so far prefix from the path until the latter remains empty as showcased in (Pseudocode 4).

Next, we utilize the final node from the involved nodes. We assign its sub-string as the concatenation of the sub-strings from all the nodes within the involved nodes and assign it as the root node for the newly formed prefixed sub-tree. The root node of the prefixed sub-tree is also assigned all the direct and indirect child nodes it possesses from the partition tree by inserting their node ids into the new tree's node-storing hash map. To avoid repeating search operations for the next prefix path in the partition tree, we remove all involved nodes for the current prefix path, including those branching from the final nodes from the partition tree.

Algorithm 4: Querying partition tree for prefix

```
Input: Prefix
Result: Involved nodes for the prefix path in the partition tree
Initialize:
current node = root node
remaining path = prefix in P
involved nodes = empty vector
while true do
    for each child in current node's children do
        if remaining path is found at start of child's substring then
            remaining path = remaining path excluding found part
            Add child to involved nodes
            if remaining path is empty then
                return involved nodes
            end
            current node = child
            break
        end
    end
    if no child updated current node then
        return involved nodes
    end
end
```

To progress with the next partition tree, we need to save the prefixed sub-trees derived from all variable length prefixes in the current partition tree. This requires storing these trees on the disk. Initially, we attempted to use the serialization feature provided by the Boost C++ library. However, we couldn't use it effectively. As a result, we decided to create our own method for serializing and deserializing suffix trees from the disk.

Our method involves encoding each element of the suffix tree into a string format in a text file using the *serialize* method of the Suffix tree class. All suffix tree elements, namely nodes, leaf nodes, and root ID are joined with a newline. The elements of each node are separated with a comma and a unique delimiter is used to separate sub-elements, such as the children of each node. This encoding method was chosen because it allows for easy reversal when deserializing the suffix tree file back into a suffix tree data structure. We implemented this by creating a method in the suffix tree class that serializes the leaves and nodes of the tree into a file, given a file name path.

For leaf nodes, we concatenate their node IDs and index in the sequence into a string using the "." character (e.g., ".leaf-nodeID.suffixindex."). Once all leaf nodes are serial-

ized, we append a newline character to the string. Non-leaf nodes require a more complex serialization process as we need to serialize its sub-string, the vector of children IDs, and the parent ID. The node sub-string is joined by ",", then the node's children IDs are joined together with "_" and the with rest of the elements with ",". The parent ID is joined with ",". And finally, the nodes suffix link of format: *"treeName"*linkNodeID* which is not yet assigned has the default initialization *""*-1* and is also joined with ",". Thus, a serialized node takes the form: *".nodeID.sub-string,child1ID_...childnID,parentID,treeName*linkNodeID.*.. After the node serialization is complete, we add a new line character. In the last line of the file, we add the root node of the tree, converted to a string.

For example, the serialized suffix tree of the sequence *BANANA$* has the following format:

```
leaves:
1.0.2.1.3.2.5.3.7.4.9.5.10.6.\n
nodes:
10.$,,0,*-1.9.$,,8,*-1.8.A,4_9,0,*-1.7.$,,6,*-1. \
6.NA,3_7,0,*-1.5.$,,4,*-1.4.NA,2_5,8,*-1. \
3.NA$,,6,*-1.2.NA$,,4,*-1.1.BANANA$,,0,*-1.\
0.,1_8_6_10,-1,*-1.\n
root node id:
-1
```

To avoid confusion it is important to note that at this stage we have not assigned leaf nodes to the trees and thus the first line will be missing from the prefix sub-trees that are created after the splitting stage. Having established a method to store the prefixed sub-trees generated at this stage for each partition on the disk, we iteratively complete the tree-splitting process for all partition trees for all prefixes in P, with each serialized prefix sub-tree saved in a directory named identically to the prefix.

3.7 Prefix tree merging

As illustrated in Figure 2, the tree merging phase accepts serialized prefixed sub-trees from the disk as inputs. The aim of this phase is to deserialize the initial tree in each prefix's directory, then sequentially deserialize and merge the remaining prefix sub-trees. This process results in the final merged tree for each prefix. To create a suffix tree from the saved ".txt" files, we create a suffix tree constructor that reads the files line by line and separates the node elements into their respective string vectors using their unique delimiter. We then iterate over these vectors, assigning the node IDs as keys in the tree's hash map. The corresponding value for each key is a node with its sub-string, child IDs, parent ID, and default suffix link. Finally, we assign the root ID from the last line in the file as the root ID of the tree.

After describing the process of deserializing a ".txt" file into a suffix tree, our next step is to merge the prefixed sub-trees utilizing the *merge* method of the suffix tree class. To accomplish this, we refer to the currently constructed merged tree as the "left tree," and the next prefixed suffix sub-tree to be merged as the "right tree." Before proceeding our implementation, first, check if a temporary folder for storing the merged trees exists. If it does not, it creates one. Next, we deserialize the left and initial right tree.

The root nodes of these trees are required to have a substring that matches the current prefix for the merging to functional correctly. Thus, If the sub-string of the root node contains additional characters, we split the root node. The remaining sub-string is then added as a new child node to the root. Continuing we search for a pair of nodes that have the same path label and then merge their children. There are two cases: The pair of nodes do not have sub-strings that overlap, and the pair has substrings with a common prefix.

In the first case, we can simply add all children of the right node to the left node. However, in the second case, we need to consider three sub-cases: either both sub-strings are the same, or the common prefix is located in either the left or right node while the other contains more characters and finally both nodes while having a common prefix have more characters. Please refer to (Pseudocode 5) for a detailed overview of how we handle these cases.

Upon the completion of merging the left tree with the current right tree, we proceed to deserialize the next right tree. This process is repeated until we obtain a final merged tree for the current prefix after exhausting all prefixed sub-trees in its directory. This merged tree is then serialized and stored on the disk. Finally, this entire process is repeated for all prefix directories.

3.8 Suffix link recovery

As depicted in (Figure 2), the algorithm's final phase involves recovering suffix links for the nodes of all the final prefix merged trees stored on the disk from the previous stage. The completion of this stage speeds up subsequent query operations. Much like the previous stage, our implementation ensures a storage directory for the final trees. If such a directory doesn't exist, one is created.

The suffix link creation process begins by loading the first merged tree into memory, starting from the root node's children. Since we're looking for these nodes' suffix links, we're searching for a node with a path that matches the current node's path, excluding the first character. For instance, if the current node has a path of "GTCA", we're searching for a node with a path of "TCA" to assign its node ID as the suffix link of the node we are investigating. For the sake of brevity, we will refer to the path with the first character excluded simply as the "path".

Given that we're dealing with a forest of trees rather than a single suffix tree, there are two main scenarios: the node has a suffix link within its own tree, or the node has a suffix link in another tree. Both scenarios require an exhaustive search. However, if we've already identified the node's parent suffix link, the process becomes less complicated.

Algorithm 5: Tree merging

```
   Input: left nodeID, right nodeID, right tree
   Output: Merged left and right tree
1  Initialize:
2  left tree = deserialize(first tree in prefix directory)
3  right tree = deserialize(next tree in prefix directory)
4  if left or right root node substring > prefix then
5      Split either or both root nodes at the common
         part and add the remaining as a child
6  foreach left child in left node children do
7      L = left child
8      foreach right child in right node children do
9          R = right child
10         common prefix = commonPrefix(L
             sub-string, R sub-string)
11         if whole sub-strings == common prefix then
12             Recursive merging for L, R
13         else if there is no common prefix between
             L,R substrings then
14             Add the R and all its direct and indirect
                 children as child of L's parent
15         else if common prefix is part of sub-strings
             then
16             if L sub-string == common-Prefix then
17                 Add to L a child with sub-string the
                     remainder string of R and all its
                     direct and indirect children
18             else if right node sub-string ==
                 common-Prefix then
19                 Split L and add a new child to it with
                     the remaining non-common sub
                     string and the direct and indirect
                     children of L
20                 Add as a child to L the node R and
                     all its direct and indirect children
21             else
22                 Add in place of L a node with the
                     common prefix and add two
                     children to it with sub-string the
                     remainder string of R, L and all
                     their direct and indirect children
23             end
24     end
25 end
26 right tree = deserialize(subsequent tree in prefix
     directory)
```

We can traverse down from the parent's link node to find the current node's suffix link.

To assist in our search across all three cases, we devise two helper functions: *compareSubstring* and *recursiveChildsearch*. The *compareSubstring* function serves to evaluate if a node's substring matches a prefix of the remaining path of the node being investigated for a suffix link, as we traverse the tree in our search of the suffix link. If a match is discovered, the length of the matching sub-string is returned otherwise it returns -1. The traversal in the tree is handled by *recurviseChildsearch*, which uses a depth-first search strategy, exploring each branch of the tree as far as possible before backtracking.

It achieves that by utilizing three main cases, which are based on whether or not *compareSubstring* identifies a match:

1. Case when the current node's substring is a prefix of the remaining part of the path.
   (a) Sub-case when the entire path is matched: The function returns the ID of the current child node, suffix link found.
   (b) Sub-case when only part of the path is matched: The function increases the path length covered by the matched sub-string and makes a recursive call to continue the search. If a match is found in the subtree, the function returns the ID of the matching node. If no match is found, the function backtracks.
2. Case when the current node's substring is not a prefix of the remaining part of the path. The function continues to the next child node.
3. Case when all child nodes have been checked and no match has been found: The function returns -1, the default suffix link, meaning that no match was found in the subtree rooted at the current node.

Having set up the helper functions, we proceed to the three main suffix link search cases. To avoid unnecessary searches, these are approached in an order of ascending computational complexity. We conduct a depth-first search to traverse the tree and initially check if a node's parent possesses a suffix link. If such a link exists, we load the merged tree it points to and call the *recursiveChildsearch* function on the suffix link node. However, if a parent lacks a link, we first check whether the node has a suffix link within the current tree by calling *recursiveChildsearch* for all the children of the node's parent, with the exception of the current node.

If we are still unable to locate a suffix link, we then limit our search to those trees whose root node substring begins with the path of the node under examination. To achieve that we apply *recursiveChildsearch* to the root node. With the completion of the suffix link search for all nodes within a merged tree, we serialize the nodes including their assigned suffix links as described before, and store it in the final trees

directory in the disk. We then proceed to repeat the process for the next tree.

4 Evaluation

We encountered a variety of issues during our implementation, some of which were due to time and technical limitations while others stemmed from the original paper itself. Certainly, the process of implementing the algorithm revealed difficulties and issues we had not initially foreseen.

Overall, the paper lacked any pseudocode and the stages of the algorithm were poorly explained in general. No mention was made of strategies for the traversal and searching necessary during any of the algorithm's stages. Consequently, we researched or created our own methods for many of the abstract concepts mentioned in the paper, meaning our implementation is significantly different than the authors'. For example, the sequence in the original paper is compressed via bitwise encoding. This concept is not explained by the authors, so we did not include it in our implementation, though we later found out it would have been fairly easy and very performant to do so. This was one of many such issues we encountered.

Starting at the **variable length prefix creation** stage, we realized there was an issue with t, the central concept of the algorithm. To calculate t, we would require an accurate estimation of available system memory M. It quickly became apparent that, in modern operating systems, this is not a simple value to obtain. There are many complexities to the memory management done by an operating system (we encountered concepts such as virtual memory and dynamic memory allocation), and that makes it difficult to measure a single value for available memory. The authors did not explain how they obtained this value, and we were unable to do so. Thus after the experimentation presented in the methodologies section, we resorted to a fixed value for t to proceed with the implementation.

The variable length prefix creation process itself was also poorly explained, with one dense paragraph covering the entire step of the algorithm. We developed our own pseudocode for this and implemented it, but it differs significantly from what the authors describe. In the paper, variable prefix creation is done in stages, where L is chosen to be the largest possible value that satisfies (2). But $|EP_i|$ contains prefixes that require further extension, something we can only know after counting each prefix's occurrences in the string. It is not clear, then, how the authors propose to find the value of L without iterating over each length between 0 and L, counting the occurrences of each prefix, and populating EP_i appropriately.

To determine the frequency of occurrence of a prefix in the sequence, we traversed the sequence string and compared with the prefix at each position. We used two different methods as described in 3.3, a single-pass method and a multi-pass method. We intuitively believed a single-pass method would be faster, because it would require a smaller number of traversals of the input string, but we found the opposite. This may be due to an issue with our implementation or because of differences in complexity costs of the two methods, but we have not been able to determine the cause.

In the **partitioning phase**, we encountered an issue that stemmed from our choice of construction algorithm. We elected to use McCreight's algorithm for suffix tree construction, while the authors used Ukkonen's algorithm. At this stage, the authors highlight an issue with the partition trees: because the partition strings do not have an ending character, the resulting trees are implicit, meaning some of the suffixes are implicitly part of the internal edges, rather than each being represented by a unique path. The authors solve this problem by extending each partition string until a unique prefix is found. This acts as an of ending character, making each partition tree explicit. We were unable to do so because McCreight's algorithm starts from the longest suffix, meaning the partition string cannot be extended one character at the time at the end of tree creation. We solved this problem on our end by adding terminal characters to the end of every partition and removing them during the merging phase, but this solution comes with a cost in complexity.

The **merging phase** once again suffered from a fairly light explanation of a complex problem, but we believe our pseudocode is very close to the authors' solution.

In the **suffix link creation phase**, the authors give a strategy for efficiently searching for suffix links: if a node's parent has a suffix link pointing to a specific node, then that node's suffix link will point to one of the linked node's children. There is no indication of what is done in the cases where no such information is available, but we assumed an exhaustive search through all of the prefixed suffix trees must be done. We ran out of time when implementing this stage, so there is still some bug fixing and optimization to be done to this part of the pipeline. The suffix link implementation contained in our submitted code may skip some suffix links, especially in the case they exist in the same tree as the node under investigation.

Another overall issue with our implementation is the serialization: we save the suffix tree objects in uncompressed text files, which uses a disproportionate amount of disk space compared to the original string (several hundreds of times larger, in fact). This resulted from challenges with the C++ compiler, which meant we could not include an external library with a more refined and optimized implementation of class serialization.

After we had a working version of the algorithm, we sought to evaluate our code's performance and compare it to the authors. We quickly realized that our implementation heavily diverges from the original paper's, as evidenced by the extremely large differences in runtime. We did not come close to being able to process human-genome-

sized sequences - in fact, the largest sequence we were able to process before runtimes increased exponentially was around 1MB. Instead, we sought to build some basic querying functions to evaluate our query times against those presented in the paper. In doing so, we discovered what we believe to be the most major issue with the paper.

Our initial idea for querying was to first attach leaves to our prefix trees and then use that information to locate a query string's position in the original string. Each leaf's index could be calculated by subtracting the length of the path to that leaf from the length of the entire sequence, since each leaf would represent a suffix starting from a specific index and ending at the overall ending character \$. It quickly became apparent, however, that some prefix trees did not contain the ending character at all.

After investigating our partition strings, we realized the ending character was only part of those prefix trees that started with a prefix contained in the final partition string. This is obvious in hindsight, since if a prefix is not part of the final partition string then no prefix sub-tree will be created, and that partition string will be excluded from the merging phase of that particular prefix tree. This highlighted a more general issue with the merged prefix trees.

During the partitioning phase, each partition tree is split into multiple sub-trees, containing only the paths that start with a specific prefix. Each of these partition prefix suffix sub-trees only represents a segment of the partition string, starting from the first occurrence of the prefix in question and ending at the final character of the partition string. The characters before the first occurrence of the prefix are not part of that prefix's suffix sub-tree.

Consider the merging phase for these partition prefix sub-trees: an assumption is made for merging that each tree represents a partition string which flows seamlessly into the next partition. The overall merged tree should represent suffixes spanning multiple partitions and some that span all of them. But since each partition prefix subtree given above skips the characters of each partition that come before the first occurrence of the prefix, the merged tree will not end up representing the entire original string and the suffixes spanning multiple partitions will be missing those characters.

This seemed, to us, to be a major flaw with the algorithm's logic. The assumption that each merged prefix tree represents the entire sequence does not appear to be true. To query a string spanning multiple partitions, one would have to traverse the prefix tree of that string and then, if the path ends before the string does, perform a second search of the rest of the string in the remaining prefix trees. Even this method is flawed, however, because if a partition's overall prefix is not one of the variable length prefixes determined by the algorithm, then the start of the partition would not be included in *any* of the prefix trees. Only the part of the partition that starts with one of the variable length prefixes would be included.

Our understanding is that the partitioning and merging procedure proposed by the authors is flawed and can create trees with missing characters, meaning queries that span multiple partitions would never be found or would require a very complex traversal of the entire forest.

Finally, we elected to implement this algorithm in C++ in an effort to make it as performant and robust as possible and compare our runtimes to those presented in the paper. This did not turn out to be a good choice, since our implementation ended up heavily diverging from the authors' and thus was not comparable, and our choice of programming language resulted in a very steep learning curve, difficulty in installing third-party libraries and general issues with compiling in different operating systems.

5 Conclusions

Our work on the *TRELLIS* algorithm did not yield the expected replication of the results we set out to achieve. However, during the implementation process, we were able to understand a complex algorithm and data structure, create from scratch solutions in areas the paper's methodology proved to be vague, and become familiar with a compiled programming language. Crucially, it was a demonstration that even published, peer-reviewed work is subject to criticism and should be approached with caution. It is not clear whether the issues we have raised with the algorithm are valid, but further investigation of the authors' work is necessary.

References

[1] URL: `https://www.genome.gov/about-genomics/fact-sheets/Sequencing-Human-Genome-cost`.

[2] Bieganski et al. "Generalized suffix trees for biological sequence data: applications and implementation". In: *1994 Proceedings of the Twenty-Seventh Hawaii International Conference on System Sciences*. Vol. 5. IEEE. 1994, pp. 35–44.

[3] Benjarath Phoophakdee and Mohammed J Zaki. "Genome-scale disk-based suffix tree indexing". In: *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*. 2007, pp. 833–844.

[4] Esko Ukkonen. "Algorithms for approximate string matching". In: *Information and control* 64.1-3 (1985), pp. 100–118.

[5] Edward M McCreight. "A space-economical suffix tree construction algorithm". In: *Journal of the ACM (JACM)* 23.2 (1976), pp. 262–272.