DNA Motif Sequencing using Suffix Trees

November 3, 2024

Jagrati Gupta (2023MCB1211) ,
Ishita Garg (2023CSB1125) ,
Yashasvi Chaudhary (2023CSB1174)

Instructor:
Dr. Anil Shukla

Teaching Assistant:
Monisha Singh

Summary: In this project, we aimed to implement a suffix tree for string processing in C++. This implementation was developed with the goal of understanding the basic structure of suffix trees and exploring their applications, especially in areas like DNA sequencing. Suffix trees are particularly useful in bioinformatics because they allow efficient searching for patterns and repeated sequences within a long string, such as a DNA sequence.

---

1. Introduction

In recent years, the volume and speed of biological data generation have increased exponentially, making it possible to sequence a bacterial genome in just a day. This rapid increase has created a pressing need for efficient data structures to store and retrieve biological information, as well as algorithms that can analyze large datasets in reasonable time frames. Traditional methods like dynamic programming, while effective for smaller-scale problems, are insufficient for comparing entire genomes. Heuristics are often necessary for identifying highly conserved regions between genomes.

Suffix trees provide an optimal solution for efficiently analyzing DNA and protein sequences, enabling exact matches and supporting many bioinformatics applications. Unlike natural languages, where there are clear sentence structures and word boundaries, DNA and protein sequences lack such properties, making traditional methods like hash tables less effective. Suffix trees and their generalized variants allow for more efficient analysis, particularly in solving problems related to DNA motif sequencing. Our project focuses on leveraging suffix trees for this purpose.

2. What are Suffix Tries?

A suffix trie is a special kind of data structure designed to store all possible suffixes of a given string, making it easy to search for any substring or pattern within that string. In a suffix trie, each node represents a character, and each path from the root to a leaf node forms a unique suffix of the original string. To build a suffix trie, we add nodes for each suffix, creating a structure where all suffixes are stored separately.

However, suffix tries have some downsides. They use a lot of memory because each suffix has its own path, and overlapping parts of the string don't share nodes, which leads to redundancy. This can be especially problematic for long strings, as it requires many nodes and connections, making the structure large and inefficient in terms of space. Additionally, building a suffix trie can be slow and resource-intensive, which limits its usefulness for some tasks.

To overcome these limitations, suffix trees are often used as a more efficient alternative.

3. Suffix Trees

Suffix trees improve upon suffix tries by compressing paths that share the same prefixes. In a suffix trie, each unique suffix of a string has its own path, leading to high memory usage and redundancy. Suffix trees solve this problem by merging all nodes along a path that share a prefix into a single edge. This compression means that each repeated substring or pattern is stored only once, creating a more compact and efficient structure.

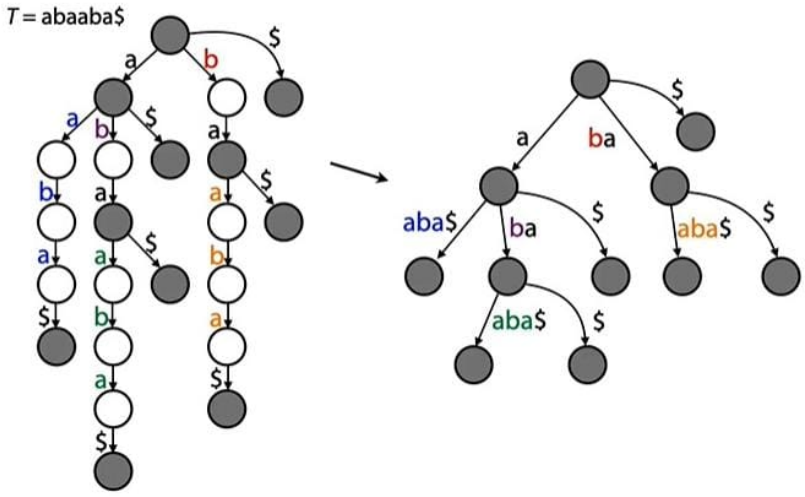


Figure 1: Suffix Trie to Tree for the string T = `abaaba$`

An Example

For the string `abaaba$`, a **suffix trie** is constructed by adding each suffix starting from every position. The suffixes are:

- `abaaba$`
- `baaba$`
- `aaba$`
- `aba$`
- `ba$`
- `a$`
- `$`

Each suffix creates a new path in the trie, leading to many nodes. For example, `abaaba$` creates a path from the root labeled with `a` → `b` → `a` → `a` → `b` → `a` → `$`. This redundancy in storing overlapping substrings results in a large and memory-intensive structure.

To create a **suffix tree**, we compress these paths by merging common prefixes. Instead of storing each character separately, we label edges with **number representation**, which uses start and end indices from the original string. For instance, an edge labeled $(0, 2)$ represents the substring `aba` from positions 0 to 2 in `abaaba$`, while $(3, 6)$ corresponds to `aba$`. This compression reduces the overall size of the structure, as repeated substrings are represented only once, leading to a more efficient representation of the suffixes.

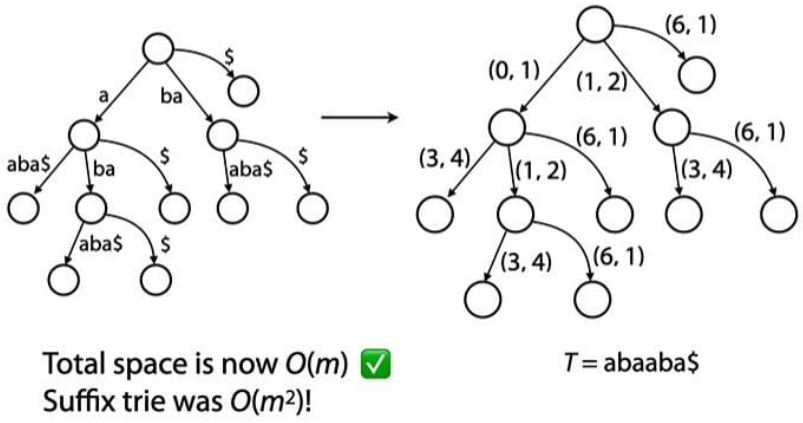
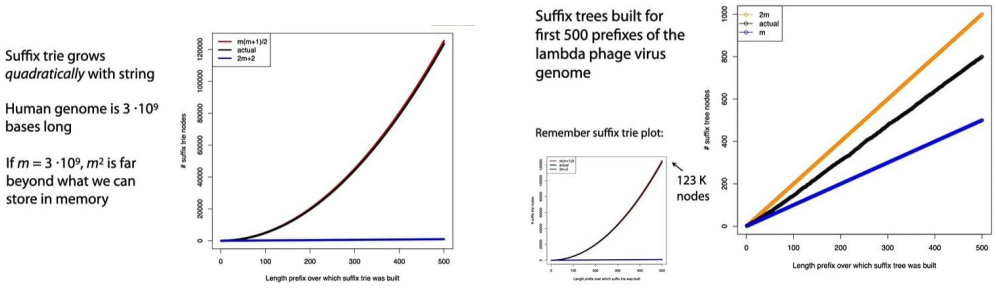


Figure 2: Suffix Tree Number Representation for the string T = `abaaba$`

Complexities



(a) Suffix Trie (b) Suffix Tree

Figure 3: Comparison between the efficiency of Suffix Tries and Suffix Tree in relation to data the size of the human genome

4. Suffix Tree: building

4.1. The Naive Approach

The naive approach to building a suffix tree involves the following steps:

1. **Generate Suffixes**:
   - For a given string T, generate all possible suffixes. Each suffix is a substring that starts from a given position in the string and extends to the end.
2. **Initialize the Tree**:
   - Start with an empty root node for the suffix tree.
3. **Insert Each Suffix**:
   - For each generated suffix, insert it into the tree by following these steps:
     - Begin at the root node.
     - For each character in the suffix, check if there is an edge that starts with the current character from the current node:
       - **If an edge exists**: Follow the edge to the next node and continue with the next character of the suffix.
       - **If no edge exists**: Create a new edge labeled with the remaining substring starting from the current character and attach it to the current node. Move to the newly created edge's node.
4. **Repeat for All Suffixes**:

- Continue this process until all suffixes have been inserted into the tree.

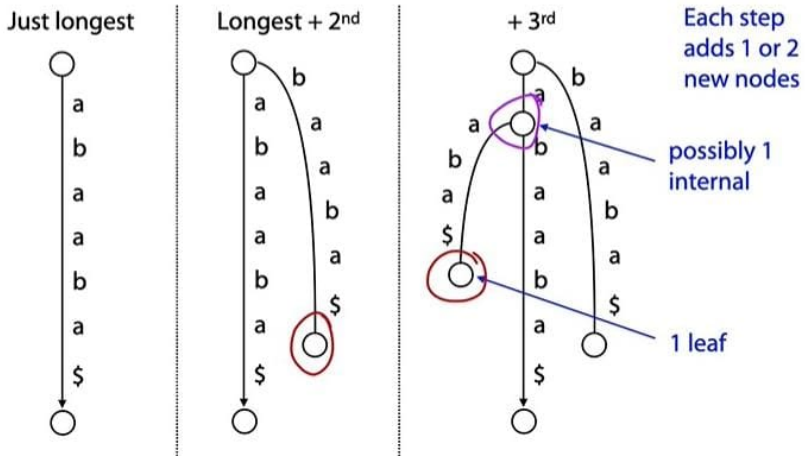


Figure 4: Suffix Tree building: Naive Approach

Characteristics

- **Efficiency**: The naive approach has a time complexity of $O(n^2)$ in the worst case, where n is the length of the string. This inefficiency arises from the need to check existing edges for every character in each suffix.
- **Redundancy**: The naive approach may lead to redundancy in the tree structure, as overlapping prefixes of different suffixes will be stored separately rather than shared.
- **Resulting Structure**: The final suffix tree contains nodes and edges where each unique path from the root to a leaf node represents a suffix of the original string.

4.2. The Ukkonen's Algorithm

Ukkonen's algorithm is an efficient method for constructing a suffix tree in linear time $O(n)$, where n is the length of the string. It utilizes suffix links and a systematic approach to handle suffixes incrementally. Here's a detailed breakdown of the algorithm:

Key Concepts

- **Suffix Links**: These are shortcuts that connect nodes in the suffix tree, helping to quickly find the next position to continue building the tree when processing suffixes.

  A suffix link is a pointer between nodes that connect similar substrings. Specifically, if there's a node for a string S, the suffix link points from this node to the node representing the suffix of S (i.e., the string S with the first character removed).

  For example, if there's a node representing the substring `"banana"`, the suffix link would connect it to the node representing `"anana"`.

  Without suffix links, the algorithm would repeatedly need to traverse up to the root and down to find the appropriate place to insert each suffix. Suffix links reduce this redundancy by allowing the algorithm to quickly jump from one part of the tree to the next related part.

  This saves time, especially in long strings with repeated patterns, because the algorithm can skip over redundant calculations for each suffix by following suffix links directly.

- **Active Point**: This consists of an active node, an active edge, and an active length that helps track where to insert the next suffix.

Steps of Ukkonen's Algorithm

1. **Initialization**:
   - Append a unique end marker (e.g., $) to the string to signify the end of all suffixes.
   - Set the active point at the root node, with an active edge set to null and an active length of 0.
2. **Iterate Over Each Character**:
   - For each character $T[i]$ in the string:
     - **Add the Character**: Extend the tree by adding the character to the currently processed suffix.
     - **Repeat the Extension Process**:
       - Use a loop to handle any potential suffix extensions that might require creating new nodes or edges.
       - Update the active point as necessary.
3. **Handle Suffix Extensions**:
   - **Check Existing Edges**: If the current active point can follow an existing edge, follow it:
     - If the next character matches, increment the active length.
     - If it does not match, split the edge and create a new node.
   - **If No Edge Exists**: Create a new edge from the active node for the current character.
4. **Use Suffix Links**:
   - After processing the current character, update the active point using suffix links:
     - If the active length is 0, move to the suffix link of the active node to continue processing.
     - If active length is greater than 0, continue along the current edge.
5. **Continue Until All Characters Are Processed**:
   - Repeat the above steps until all characters in the string (including the unique end marker) are processed.

Characteristics

- **Time Complexity**: Ukkonen's algorithm runs in linear time $O(n)$, making it suitable for large strings.
- **Space Complexity**: The space used is proportional to the number of unique substrings, which is efficient given the input size.
- **Resulting Structure**: The final suffix tree will represent all suffixes of the input string with shared prefixes efficiently compressed, reducing redundancy.

Aspect	First Code (Naive)	Second Code (Ukkonen's Algorithm)
Algorithm	Naive insertion of all suffixes	Ukkonen's linear time suffix tree construction
Time Complexity	$O(n^2)$	$O(n)$
Suffix Insertion	Inserts each suffix from scratch	Efficiently reuses common parts using suffix links
Use of Suffix Links	No suffix links	Uses suffix links for efficiency
Efficiency	Inefficient for large inputs	Highly efficient, scales linearly
Use Case	Small strings or teaching purposes	Large strings, optimal performance needed

Table 1: Comparison between Naive Suffix Tree Construction and Ukkonen's Algorithm.

5. Our Take

5.1. The Naive Approach

We measured the time taken to construct the suffix tree using the chrono library in C++. We recorded the time immediately before and after building the tree. This measurement provided us with a clear indication of the program's execution time in milliseconds.
Our observations showed that this naive approach is feasible for small text sizes, but performance degrades rapidly as the input size grows due to the quadratic time complexity of this method. This limitation is particularly noticeable with longer DNA sequences or large text files, where constructing the suffix tree takes considerably more time.

5.1.1 Graph

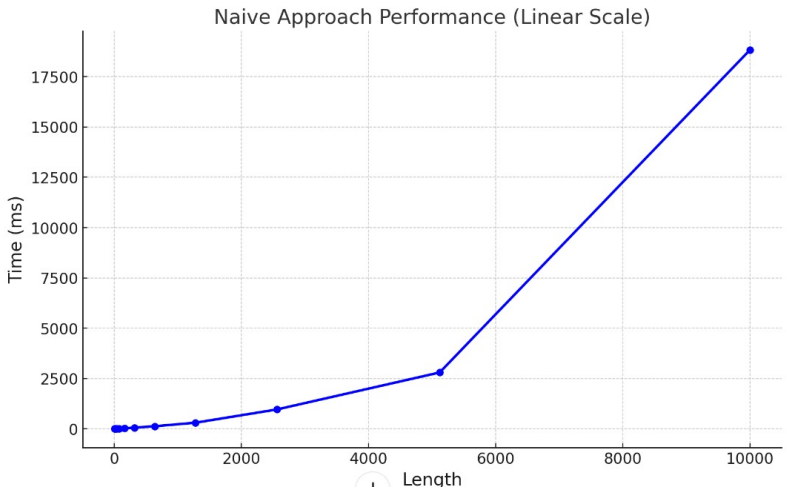


Figure 5: Our graph when implementing the Naive Approach

5.1.2 Code Logic

1. **Initialize Root Node**: Start with an empty root node for the suffix tree.
2. **Loop Through Suffixes**: For each suffix starting at position i in the text, traverse from the root.
3. **Build Path for Each Suffix**: For each character j in the current suffix:
   - If there is no existing edge for the character at `text[`j`]`, create a new node for it, representing a unique substring path.
   - Set the new node's `start` as j and `end` as the last index of the text.
   - Link this node as a child of the current node along the edge represented by `text[`j`]`.
4. **Return Root**: The function returns the root node, which now has paths representing all suffixes.

5.1.3 Pseudocode

Algorithm 1 Build Suffix Tree

```
function buildSuffixTree(text)
    length ← length of text
    ROOT ← create new SuffixTreeNode with start = -1 and end = -1
    for i ← 0 to length - 1 do
        node ← ROOT                                   ▷ Start from the root for each suffix
        for j ← i to length - 1 do
            index ← ASCII value of text[j]                    ▷ Get index for character
            if node.children[index] = NULL then
                child ← create new SuffixTreeNode with start = j and end = length - 1
                child.suffixIndex ← i                ▷ Store the starting position of the suffix
                node.children[index] ← child                ▷ Link the new node as a child
            end if
            node ← node.children[index]                        ▷ Move to the next node
        end for
    end for
    return ROOT                          ▷ Return the root of the constructed suffix tree
end function
```

5.2. Ukkonen's Algorithm

5.2.1 Graph

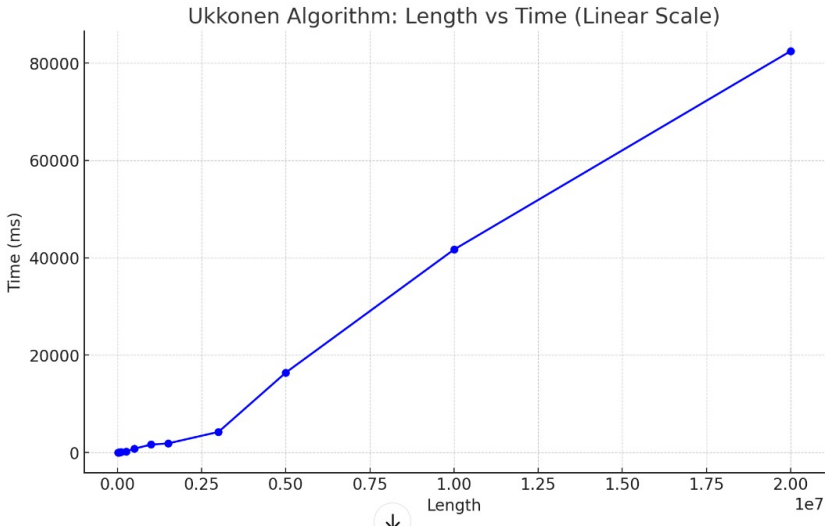


Figure 6: Our graph when implementing Ukkonen's Algorithm

5.2.2 Code Logic

1. **Initialization**: Set up the suffix tree and create a root node.
2. **Character Mapping**: Convert characters to indices for node access.
3. **Extension**: Add each character to the suffix tree, managing:
   - Active point (location and length).
   - Edge cases (no outgoing edge, character matches, and splitting edges).
4. **Suffix Link Management**: Maintain links to speed up tree construction.
5. **Walking Down**: Traverse to the appropriate child node when possible.

5.2.3 Pseudocode

Algorithm 2 Initialize Suffix Tree

```
function initializeSuffixTree
    Set current_position, r, active_node, active_length to 0
    Clear tree
    root ← createNewNode(start = -1, end = ∞)
    active_node ← root
end function
```

Algorithm 3 Create New Node

```
function createNewNode(start, end = ∞)
    Create new node with start and end
    Add node to tree
    return index of new node
end function
```

Algorithm 4 Character to Index Mapping

```
function charToIndex(char)
    if char is 'A' then return 0
    end if
    if char is 'T' then return 1
    end if
    if char is 'G' then return 2
    end if
    if char is 'C' then return 3
    end if
    if char is '$' then return 4
    end if
end function
```

Algorithm 5 Extend Suffix Tree

```
function extendSuffixTree(newChar)
    Append newChar to text
    Increment current_position
    r ← r + 1
    while r > 0 do
        if active_length = 0 then
            active_edge_index ← current_position
        end if
        edge_index ← charToIndex(text[active_edge_index])
        if tree[active_node].nextIndices[edge_index] = 0 then
            leaf_node ← createNewNode(current_position)
            tree[active_node].nextIndices[edge_index] ← leaf_node
            addSuffixLink(active_node)
        else
            next_node ← tree[active_node].nextIndices[edge_index]
            if walkDown(next_node) then
                continue
            end if
            if text[tree[next_node].start + active_length] = newChar then
                active_length ← active_length + 1
                addSuffixLink(active_node)
                break
            end if
            split_node ← createNewNode(tree[next_node].start,
tree[next_node].start + active_length)
            tree[active_node].nextIndices[edge_index] ← split_node
            new_leaf ← createNewNode(current_position)
            tree[split_node].nextIndices[charToIndex(newChar)] ← new_leaf
            tree[next_node].start ← tree[next_node].start + active_length
            tree[split_node].nextIndices[charToIndex(text[tree[next_node].start])]
← next_node
            addSuffixLink(split_node)
        end if
        r ← r − 1
        if active_node = root and active_length > 0 then
            active_length ← active_length - 1
            active_edge_index ← current_position - r + 1
        else
            active_node ← (tree[active_node].suffix_link > 0) ?
tree[active_node].suffix_link :  root
        end if
    end while
end function
```

6. Motif Search in Suffix Tree

At first glance, it might seem like our project is a simple string search problem. However, it is much more than that for several reasons:

Why It's More than a Simple Search Problem

Simple search algorithms are adequate for one-off pattern searches but do not generalize well to situations where you need to query for multiple motifs, manage large datasets efficiently, or handle advanced string matching tasks. In our project, we've built a structure that can efficiently handle multiple complex queries after the preprocessing step (suffix tree construction). This is significantly more powerful than just finding one occurrence of a pattern. Suffix trees allow you to solve entire classes of problems beyond simple searching, which is why they are more advanced and challenging.

6.1. Advanced Applications in DNA Sequencing

1. **Identifying Regulatory Motifs:** This means finding specific DNA sequences that control how genes turn on or off. These sequences are where proteins, called transcription factors, can attach and help regulate genes.
2. **Comparative Genomics:** Here, we compare DNA sequences across different species to see what parts are similar. By finding shared motifs (like small, important DNA patterns), scientists can learn how different species are related and trace their evolution.
3. **Analyzing Genetic Variants:** Some DNA sequences are linked to diseases. Finding motifs in these regions helps scientists understand how genetic differences might change how genes work, potentially leading to health issues.

Suffix trees make it super easy to find motifs (or repeated DNA patterns) and compare DNA from two people. They allow us to quickly spot shared sequences or motifs, which can show if two people have similar DNA in important regions. This is helpful for studying genetic similarities or differences in health and ancestry!

6.2. Time Complexity: Suffix Trees vs. Naive Search (Control + F)

To understand the efficiency of our approach using suffix trees, we compare it to the naive search method, such as the "Control + F" search function commonly used in text editors. Here's a breakdown of the time complexity for both methods:

6.2.1 Naive Search Method (Cntrl + F)

The naive approach to searching for a pattern in a text involves checking every possible starting position in the text to see if the pattern matches. This leads to the following time complexity:

- **Worst-Case Time Complexity**: $O(n \times m)$, where:
  - n is the length of the text.
  - m is the length of the pattern.

This quadratic complexity arises because, in the worst case, every character of the pattern may need to be compared at every position in the text. If you have a large text and a long pattern, this method becomes inefficient.

6.2.2 Suffix Tree Method

In contrast, our approach using a suffix tree dramatically improves the efficiency of searching:

- **Preprocessing Time**: Constructing the suffix tree using Ukkonen's algorithm takes $O(n)$ time, where n is the length of the text.
- **Search Time**: Once the suffix tree is constructed, any pattern of length m can be searched in $O(m)$ time.

Overall Efficiency:

- After the initial $O(n)$ preprocessing, querying for any pattern is extremely fast, $O(m)$, regardless of the size of the text. This is because the suffix tree structure allows us to follow edges corresponding to the characters of the pattern efficiently.

So, in conclusion,

- **Control + F**: If you were searching for a pattern in a long text (like a book), Control + F might have to scan through the entire text multiple times to confirm all matches. This becomes noticeably slow for large texts.
- **Suffix Tree Approach**: Imagine having a pre-built index of every substring of the text. Searching for any pattern is as simple as following a predetermined path, making it significantly faster, especially for repeated or multiple queries.

6.2.3 Real-World Example

- **Text Length (n)**: Suppose you are searching in a DNA sequence with 1 million characters.
- **Pattern Length (m)**: Your pattern is 100 characters long.
  - **Naive Search**: In the worst case, it could take 1,000,000 $\times$ 100 = 100,000,000 character comparisons.
  - **Suffix Tree Search**: Constructing the suffix tree takes $O(1{,}000{,}000)$ time, and searching takes $O(100)$ time. Thus, even if the preprocessing is slightly costly, the overall efficiency for multiple queries is significantly better.

6.3. PseudoCode

Algorithm 6 count_motif_occurrences

Input: pattern
Output: occurrences

```
current_node ← root
current_index ← 0
occurrences ← 0
                                        ▷ Traverse the suffix tree to find the pattern
while current_index < LENGTH(pattern) do
    current_char ← pattern[current_index]
    if tree[current_node].nextIndices[char_to_index(current_char)] = 0 then return occurrences                                        ▷ Motif not found
    end if
    edge ← tree[current_node].nextIndices[char_to_index(current_char)]
    edge_length ← tree[edge].edge_length()
    for i ← 0 edge_length - 1 do
        if current_index ≥ LENGTH(pattern) then                    ▷ Pattern fully matched
        end if
        if text[tree[edge].start + i] ≠ pattern[current_index] then return occurrences                                        ▷ Mismatch, motif not found
        end if
        current_index ← current_index + 1
    end for
    current_node ← edge                                            ▷ Move to the next node
end while
                                        ▷ Count occurrences from the matched node
stack ← INITIALIZE stack WITH current_node
while stack is not EMPTY do
    node ← stack.pop()
    for i ← 0 ALPHABET_SIZE - 1 do
        child_node ← tree[node].nextIndices[i]
        if child_node ≠ 0 then
            if tree[child_node].nextIndices[0] = 0 then                ▷ If leaf node
                occurrences ← occurrences + 1
            else
                stack.push(child_node)
            end if
        end if
    end for
end while
    return occurrences
```

7. Further Applications of Suffix Trees in DNA Sequencing

1. **Motif Searching:** Identifying specific DNA sequences or motifs associated with biological functions or regulatory elements within large genomic datasets. We have implemented this in code however the applications are immense.
2. **Finding Repeats:** Detecting repeated sequences in DNA, which can be crucial for understanding genetic diseases and evolutionary biology.
3. **Longest Common Substring:** Determining the longest common substring between two or more DNA sequences, useful for phylogenetic studies and understanding evolutionary relationships.
4. **Palindromic Sequence Identification:** Efficiently locating palindromic sequences in DNA, which are often involved in the formation of hairpin structures in RNA.

8. Conclusions

Suffix trees provide an efficient structure for substring search and pattern matching, particularly in large datasets. While the naive construction method has a quadratic time complexity, Ukkonen's algorithm optimizes this to linear time, making it suitable for extensive strings.

Suffix trees excel in handling multiple queries and complex string-related problems, making them crucial in bioinformatics. They enable rapid identification of DNA motifs, repetitive sequences, and genomic comparisons, facilitating the discovery of genetic patterns and potential gene locations. bioinformatics.

9. Bibliography and citations

Acknowledgements

We wish to thank our instructor, Dr Anil Shukla and our Teaching Assistant Monisha Singh for their guidance and valuable inputs provided throughout our project.

References

1. https://brenden.github.io/ukkonen-animation/
2. https://stackoverflow.com/questions/9452701/ukkonens-suffix-tree-algorithm-in-plain-english/9513423#9513423
3. https://www.youtube.com/watch?v=KT18KJouHWg
4. https://eecs.wsu.edu/~ananth/CptS571/Lectures/C061ch6.pdf
5. https://www.youtube.com/watch?v=odyGCviFmXA&list=PL2mpR0RYFQsDFNyRsTNcWkFTHTkxWREeb
6. https://github.com/kasravnd/SuffixTree/blob/master/README.md
7. ChatGPT