Suffix Tree: A Detailed Study of the Data Structure

José G. Galvez P., Jesús V. Niño C., Milton J. Cordova N.
Faculty of Computer Science, University of Engineering and Technology, Lima, Peru
Emails: jose.galvez.p@utec.edu.pe, jesus.nino@utec.edu.pe, milton.cordova@utec.edu.pe

***Abstract*—This paper provides a detailed study of the suffix tree, covering its construction, properties, and key operations. We analyze its time complexity and compare it to other string processing data structures. In addition, we discuss real-world applications, demonstrating its practical significance. Finally, we provide insights into its implementation and visualization to facilitate understanding and usability.**

***Index Terms*—Suffix Tree, String Matching, Data Structures, Trie, Compressed Trie**

I. Introduction

Efficient text processing is a fundamental challenge in computer science, especially in applications such as search engines, bioinformatics, and data compression, where rapid substring searches and pattern matching over large datasets are essential. The naive approach, which attempts to match a substring against every possible alignment, is straightforward but becomes prohibitively slow as data volumes increase [3]. This challenge has led to the development of specialized data structures that optimize these operations. Among them, the suffix tree stands out as a versatile tool that organizes all suffixes of a given string into a compact structure, enabling both exact and approximate pattern matching in optimal time. Despite its powerful capabilities, the construction and memory requirements of suffix trees present significant challenges, particularly when dealing with large texts that may not fit into main memory [2]. Consequently, understanding their structure, construction methods, and optimization strategies is critical to leveraging their full potential in modern computing applications.

II. Related Work

Since their inception, suffix trees have been extensively researched and refined. Weiner [1] introduced the first linear-time construction method for suffix trees, which was later improved by McCreight [3] to reduce storage overhead and improve efficiency. These foundational contributions established suffix trees as indispensable tools in text indexing and bioinformatics. Further investigations, such as those of Tian et al. [2], have addressed the memory limitations of suffix trees by exploring disk-based construction methods for large-scale datasets, thus improving I/O performance. More recent studies, including those by Ferragina [4], have extended the scope of suffix trees by integrating compression strategies, cache optimizations, and architectural improvements. These advancements not only overcome some of the original limitations but also pave the way for hybrid approaches—such as compressed suffix trees and external-memory solutions—that continue to enhance their scalability and practical applicability.

III. The Structure of the Suffix Tree

A. Definition and Properties

Some key properties of the suffix tree are:

- Each node represents a suffix of the string.
- The root node connects to the main branches of the tree.
- Leaf nodes should end with a '$' symbol, indicating the end of a suffix.
- Edges represent transitions between nodes.

B. Structure and Node Representation

A suffix tree is a compressed trie that represents all suffixes of a given string. In this structure, we distinguish between:

1) Implicit vs. Explicit Suffix Trees:

- An *implicit suffix tree* for a string S is derived from the suffix tree of $S\$$ by removing every occurrence of the terminal symbol '$' from the edge labels. Then, any edge that no longer has a label is removed, and any node with fewer than two children is also deleted [6].
- An *explicit suffix tree* is the final structure where all suffixes are explicitly stored, ensuring that each suffix is uniquely represented as a path from the root to a leaf. To transition from an implicit to an explicit suffix tree, a unique terminal character ('$') is added at the end of the string. This guarantees that no suffix is a prefix of another, making the structure well-defined.

2) Node and Edge Representation:

- Each internal node has at least two children.
- The tree contains at most as many leaves as the number of suffixes inserted.
- Edges are labeled with substrings, and paths from the root to the leaves represent the suffixes.

3) Role of Suffix Links: Suffix links are pointers that help in efficiently linking internal nodes, allowing the tree to be constructed in linear time ($O(n)$). These links reduce redundant work by allowing the algorithm to jump between related suffixes rather than restarting from the root each time.

To efficiently construct this tree, Ukkonen's algorithm builds it in phases and extensions [8]. Each phase processes a new character, and each extension ensures that all relevant suffixes are correctly represented. The process follows these steps:

1) Start with an empty tree.
2) Insert substrings incrementally, maintaining an implicit suffix tree.
3) Use suffix links to speed up insertions.

4) Convert the implicit structure into an explicit suffix tree by adding a unique terminal character ('$') to ensure that no suffix is a prefix of another.

This incremental construction results in an efficient $O(n)$ time complexity, making Ukkonen's algorithm one of the most optimal methods for suffix tree construction [8].

C. Structure Definitions

First, the alphabet size is predefined as a global macro of 26, which corresponds to the number of letters from 'A' to 'Z' in the ASCII format. This should suffice for representing words in the English language.

$$\Sigma = 26$$

Now we define the structure for the Node of the Suffix Tree and its initilization:

struct Node:

- parent: A pointer to the parent node.
- children: Array of size `ALPHABET_SIZE` of `Node*`.
- edgeLabel: Stores the substring label for the edge to this node.
- start, end: Indices representing the substring in the original text.
- isLeaf: Boolean flag indicating if it's a leaf node.

Function Node(startIdx, endIdx):

- parent = nullptr
- start = startIdx, end = endIdx
- isLeaf = false
- For i from 0 to $\Sigma - 1$:
  - children[i] = nullptr

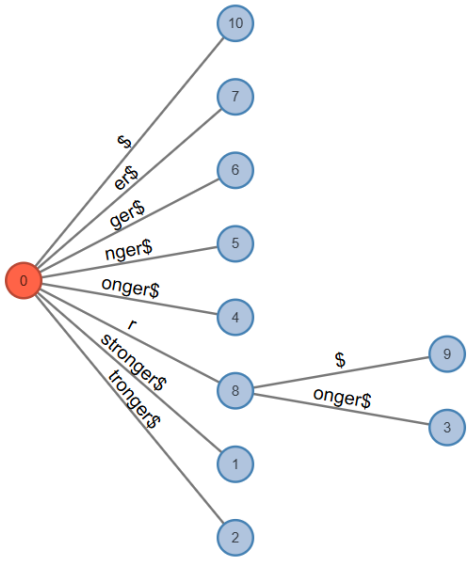

Fig. 1. Suffix Tree representation for the word "stronger"

IV. Operations on the Suffix Tree

In this section we will show the operations to implement for the suffix tree and explain the algorithms involved.

A. Construction

For the construction of the suffix tree, we use the Ukkonen algorithm, which is an efficient method that builds the tree in linear time, where n is the length of the input string. This algorithm processes the string from left to right, adding one character at a time in a series of phases. In each phase, multiple extensions are performed to ensure that all current suffixes are represented in the tree.

Key functions used in this algorithm include:

- **extendSuffixTree(i):** Handles the extension of the tree with the i-th character of the string.
- **walkDown(nextNode):** Moves the active point deeper in the tree when the current active length exceeds the edge length.
- **splitEdge(pos):** Splits an edge at a given position when a mismatch is encountered.
- **createSuffixLink(node):** Sets up a suffix link from the last created node to optimize subsequent insertions.
- **setActivePoint(i):** Updates the active point after processing an extension.

Algorithm 1 Construction(S)

Data: A string S of length n.
Result: A suffix tree T built for S.
text $\leftarrow$ S $n \leftarrow$ length(S) Create an empty root node; let root be T.root activeNode $\leftarrow$ root activeEdge $\leftarrow$ "" activeLength $\leftarrow$ 0 remainingSuffixCount $\leftarrow$ 0 lastCreatedNode $\leftarrow$ null end $\leftarrow -1$
for $i \leftarrow 0$ ***to*** $n - 1$ **do**
  extendSuffixTree(i)
return T

Algorithm 2 walkDown(nextNode)

Data: A node pointer `nextNode`.
Result: Moves the active point deeper in the tree if possible; returns `true` if walked down, else `false`.
Function `walkDown(`*nextNode*`)`**:**
  if *activeLength* $\geq$ *nextNode.edgeLength()* **then**
    activeEdge $\leftarrow$ text[nextNode.start + activeLength] activeLength $\leftarrow$ activeLength $-$ nextNode.edgeLength() activeNode $\leftarrow$ nextNode
    return true
  return false

Algorithm 3 createSuffixLink(node)

Data: A node pointer (`node`).
Result: Updates the suffix link of the last created node.
Function `createSuffixLink(`*node*`)`**:**
  if *lastCreatedNode* $\neq$ *null* **then**
    lastCreatedNode.suffixLink $\leftarrow$ node
  lastCreatedNode $\leftarrow$ node

Algorithm 4 splitEdge(nextNode, activeLength)

Data: Node pointer `nextNode`, current `activeLength`.
Result: Splits the edge at the given `activeLength` and returns the new internal node (`splitNode`).
Function `splitEdge(`*nextNode, activeLength*`)`**:**
splitPosition ← nextNode.start + activeLength − 1 Create new internal node `splitNode` with: start ← nextNode.start end ← splitPosition activeNode.children[activeEdge] ← splitNode splitNode.children[text[splitPosition + 1]] ← nextNode nextNode.start ← splitPosition + 1 **return** splitNode

Algorithm 5 extendSuffixTree(i)

Data: Current character index i
Result: Extends the suffix tree with text$[i]$
end ← end + 1 remainingSuffixCount ← remainingSuffixCount + 1 lastCreatedNode ← null
while *remainingSuffixCount* > 0 **do**
  if *activeLength = 0* **then**
    activeEdge ← text$[i]$
  if *activeNode does not have a child for activeEdge* **then**
    Create new leaf node with start = i and end = end activeNode.children[activeEdge] ← new leaf node
    if *lastCreatedNode* $\neq$ *null* **then**
      lastCreatedNode.suffixLink ← activeNode lastCreatedNode ← null
  else
    nextNode ← activeNode.children[activeEdge]
    if *activeLength* $\geq$ *nextNode.edgeLength()* **then**
      activeEdge ← text[nextNode.start + activeLength] activeLength ← activeLength - nextNode.edgeLength() activeNode ← nextNode **continue** while loop
    if *text[nextNode.start + activeLength] = text*$[i]$ **then**
      activeLength ← activeLength + 1
      if *lastCreatedNode* $\neq$ *null* **then**
        lastCreatedNode.suffixLink ← activeNode
      break out of while loop
    splitNode ← splitEdge(nextNode, activeLength) Create new leaf node with start = i and end = end splitNode.children[text$[i]$] ← new leaf node
    if *lastCreatedNode* $\neq$ *null* **then**
      lastCreatedNode.suffixLink ← splitNode
    lastCreatedNode ← splitNode
  remainingSuffixCount ← remainingSuffixCount - 1
  if *activeNode = root* ***and*** *activeLength* > 0 **then**
    activeLength ← activeLength - 1 activeEdge ← text$[i-$ remainingSuffixCount $+1]$
  else if *activeNode* $\neq$ *root* **then**
    activeNode ← activeNode.suffixLink

Algorithm 6 setActivePoint(i)

Data: Current character index `i`.
Result: Updates the active point for the next extension.
Function `setActivePoint(`*i*`)`**:**
  if *activeNode = root* ***and*** *activeLength ¿ 0* **then**
    activeLength ← activeLength - 1 activeEdge ← text[i - remainingSuffixCount + 1]
  else if *activeNode* $\neq$ *root* **then**
    activeNode ← activeNode.suffixLink

B. Destruction

Destruction of a suffix tree is similar to freeing a trie structure. We must recursively delete all nodes to ensure proper memory de-allocation.

To destroy the suffix tree, we perform a post-order traversal by deleting all child nodes before deleting their parent. This ensures that we do not leave dangling pointers in memory.

More formally, given a suffix tree T, we recursively delete all child nodes before deleting the current node itself. If a node has children, we first traverse through all of them, freeing memory, and then delete the node itself.

Algorithm 7 Destroy()

Data: A suffix tree `T` rooted at `R`.
Result: Frees all memory allocated for the tree `T`.
Function `destroyNode(`*v*`)`**:**
  foreach *child* `c` *in* `v.children` **do**
    if `c` *is not null* **then**
      `destroyNode(`*c*`)`
  Delete `v`
Function `Destroy()`**:**
  `destroyNode(`*R*`)` R ← null

C. String Matching

Given an already constructed suffix tree, to search for a pattern P we start from the root R and follow the edges based on the characters of P.

First, we examine the first character of P and follow the edge labeled with this character. If there is no edge corresponding to the next character of P, then P is not in the tree. If the label of the edge partially matches P, compare it character by character, if the entire label of the edge matches, continue to the next node. This process repeats until the pattern is fully matched; if all the characters of P are found in the tree, then P exists in the original string used to construct the tree. Otherwise, if at any point there is a mismatch, P does not exist.

Algorithm 8 Search(P)

Data: A suffix tree `T` built from string `S`, a pattern `P`.
Result: Returns `true` if `P` exists in `S`, `false` otherwise.
v ← Root(T) pos ← 0
while *pos* $< |P|$ **do**
  if *v has no child with edgeLabel starting with P[pos]* **then**
    return false
  v ← v.child(P[pos]) // Move to the matching child edge
  ← edgeLabel(v) // Get the label on the edge len ← min(|edge|, $|P|$ - pos)
  if *edge[0:len]* $\neq$ *P[pos:pos+len]* **then**
    return false // Mismatch found
  pos ← pos + len
return true

D. Finding All Matches

Similarly to the previous operation, we will search for the pattern P = (p0, p1, p2, …, p(m-1)) in a suffix tree that has already been built from a string S. We start at the root R and examine the first character of P, p0, then follow the appropriate edge in the tree. If such an edge does not exist, the pattern P is not in S. Otherwise, we must check whether the label of the edge fully or partially matches the corresponding substring of P:

- Full Match: If the entire edge label exactly matches the corresponding segment of P, we continue following the next characters of P.
- Partial Match: If only a portion of the edge label is needed (because the remaining part of P is shorter than the edge label), we compare the necessary portion of the edge label with the corresponding substring of P.

If at any point a mismatch is encountered during these comparisons, the algorithm concludes that P does not exist in S and returns an empty list. If we successfully reach the end of P while following the edges, it means that P is a substring of S. In this case, all the leaf nodes in the subtree of the last matched node represent the occurrences of P in S, and each leaf node stores the index indicating the starting position of that occurrence. Finally, we return a list of those positions.

Algorithm 9 FindAllMatches(P)

Data: A suffix tree `T` built from string `S` (with termination symbol), and a pattern `P` = $(p_0, p_1, \ldots, p_{m-1})$.
Result: Returns a list of positions in `S` where `P` occurs, or an empty list if not found.
v ← Root(T) pos ← 0
while *pos* $< |P|$ **do**
  if *v does not have a child whose edge label starts with P[pos]* **then**
    return empty list
  v ← v.child(P[pos]) edge ← label on the edge from v.parent to v len ← min(|edge|, $|P|$ - pos)
  if *substring(edge, 0, len)* $\neq$ *substring(P, pos, len)* **then**
    return empty list
  pos ← pos + len
matches ← GetLeafIndices(v) **return** matches

E. Longest Repeated Substring

We can find the longest repeated substring in a given string S using a suffix tree built from S and identifying the deepest internal node with the longest path label. This is because repeated substrings in S correspond to internal nodes with multiple children, and the deepest of such nodes represents the longest repeated substring. For this algorithm we use the concept of path labels, which represents the concatenation of the edge labels along the path followed. The path label of a node corresponds to a substring of S. We start at the root R and traverse the tree, maintaining the longest path found so far. Each internal node represents a substring that appears multiple times, and we track the node with the longest path label.

Algorithm 10 LongestRepeatedSubstring()

Data: A suffix tree rooted at `R`, built from a string `S`.
Result: Returns the longest repeated substring in `S`.
v ← R maxDepth ← 0 deepestNode ← null
Function `DFS(`*v, depth*`)`**:**
  if *v is an internal node* **then**
    if *depth* $>$ *maxDepth* **then**
      maxDepth ← depth deepestNode ← v
    foreach *child c of v* **do**
      `DFS(`*c, depth* + $|L(c)|$`)`
`DFS` (R, 0)
return PathFromRoot(deepestNode)

F. Shortest Unique Substring

The shortest unique substring of a string S is the smallest substring that occurs exactly once in S. We start at the root R and traverse the suffix tree using depth-first search(DFS). At each internal node v, we check if the subtree rooted at v leads to exactly one leaf node. This indicates that the path from R to v represents a unique substring in S. If a node v corresponds to a unique substring, we compare its length with the shortest unique substring found so far. If it is shorter, we update our result. We shall continue this process recursively for all children of v, keeping track of the path label accumulated during the traversal. Finally, once the DFS completes, the algorithm returns the shortest unique substring found in S.

Algorithm 11 ShortestUniqueSubstring()

Data: A suffix tree built from a string S with a termination symbol \$.
Result: Returns the shortest unique substring in S.
v ← R minLength ← ∞ shortestSubstring ← ””
Function `DFS(`*v, depth, pathLabel*`)`**:**
  if *v leads to exactly one leaf node* **then**
    if *depth* $<$ *minLength* **then**
      minLength ← depth shortestSubstring ← pathLabel
  foreach *child c of v* **do**
    `DFS(`*c, depth* + $|L(c)|$, *pathLabel* + $L(c)$`)`
`DFS` (R, 0, ””)
return shortestSubstring

V. Time Complexity Analysis

In this section, we analyze the time complexity of key operations on a Suffix Tree: Construction, Destruction, String Matching, Finding All Matches, Longest Repeated Substring, and Shortest Unique Substring. We assume the alphabet size is Σ, and n represents the length of the input string.

A. Construction Complexity

The construction of a suffix tree involves inserting all suffixes of the given string into the tree. Using Ukkonen's Algorithm, this can be done in $O(n)$ time for a string of length n. The naive approach takes $O(n^2)$ time since each suffix is inserted independently. Ukkonen's algorithm optimizes this by introducing suffix links, allowing linear-time construction $O(n)$. Thus, the overall time complexity of suffix tree construction is: $O(n)$.

B. Destruction Complexity

The destruction of a suffix tree requires deleting all nodes. Since the suffix tree consists of at most $O(n)$ nodes and each node is visited once during deletion, the total complexity is: $O(n)$.

C. String Matching Complexity

The string matching operation searches for a substring P of length m within the suffix tree. The search follows a single path down the tree by comparing at most m characters along the edges. Since the suffix tree is a compressed trie, the search completes in $O(m)$ time. Thus, the time complexity of string matching is: $O(m)$.

D. Finding All Matches Complexity

To find all occurrences of a pattern P in the text, we first locate P in the tree, which takes $O(m)$ time. Once found, all occurrences correspond to leaf nodes in the subtree of the matched node. Collecting these occurrences takes $O(k)$ time, where k is the number of matches. Thus, the overall time complexity for finding all matches is: $O(m + k)$.

E. Longest Repeated Substring Complexity

The longest repeated substring in a suffix tree corresponds to the deepest internal node (i.e., the node with the longest label that appears at least twice). Traversing the tree takes $O(n)$ time. Identifying the deepest internal node also takes $O(n)$. Thus, the time complexity of finding the longest repeated substring is: $O(n)$.

F. Shortest Unique Substring Complexity

The shortest unique substring is the smallest substring that appears exactly once in the text. To find this, we traverse the suffix tree and locate the shallowest unique leaf. Each traversal step takes $O(n)$, and verifying uniqueness requires additional $O(n)$ operations. Thus, the time complexity of finding the shortest unique substring is: $O(n)$.

VI. Comparison with Other Data Structures

Suffix Trees are widely used for efficient substring searching and pattern matching. However, other data structures, such as Tries, Patricia Trees, and Suffix Arrays, also provide efficient mechanisms for handling string data. Below, we compare these structures based on their time complexity, space complexity, and usage scenarios.

A. Trie (Prefix Trie)

A Trie is a tree-like data structure designed to store and retrieve strings efficiently. Each node represents a character, and paths from the root to leaves represent entire words or prefixes. Tries are commonly used for prefix searches, autocomplete, and dictionary applications [14].

- **Time Complexity:** Insert, search, and delete operations take $O(m)$ time, where m is the length of the string being processed.
- **Space Complexity:** The worst-case space requirement is $O(|\Sigma| \times n)$, where $|\Sigma|$ is the alphabet size and n is the total length of all stored strings. This can be memory-intensive, especially for large alphabets.

B. Patricia Tree (Compressed Trie)

A Patricia Tree (Practical Algorithm to Retrieve Information Coded in Alphanumeric) is an optimized version of a Trie that reduces space usage by compressing paths with a single child into one edge. This makes it more memory-efficient while maintaining fast lookup times [9].

- **Time Complexity:** The average time complexity for search, insertion, and deletion is $O(m)$, similar to standard Tries, but with reduced comparisons due to compression.
- **Space Complexity:** Patricia Trees require significantly less space than Tries, as they eliminate unnecessary nodes. However, they may introduce additional overhead for bitwise comparisons in some cases.

C. Suffix Array

A Suffix Array is a space-efficient alternative to the Suffix Tree, introduced by Manber and Myers as a sorted array of all suffixes of a given string. Instead of explicitly representing suffixes in a tree structure, it stores the starting indices of sorted suffixes, allowing for efficient substring searches via binary search while using significantly less memory [15]. This makes it particularly suitable for large-scale text indexing, such as in bioinformatics and search engines, where space efficiency is critical. However, its reliance on binary search makes it slower than Suffix Trees for operations like retrieving all occurrences of a pattern in one pass.

- **Time Complexity:**
  - **Construction:** Initially $O(n \log n)$ using radix sorting techniques, later improved to $O(n)$ with DC3 [16].
  - **Search:** $O(m \log n)$ using binary search, improved to $O(m + \log n)$ when using an LCP (Longest Common Prefix) array.

- **Space Complexity:** $O(n)$, making it more memory-efficient than Suffix Trees, which require $O(n \log n)$ due to pointers and internal nodes.

VII. Applications

A. Search Engines

Suffix trees are extensively employed in search engines to enable rapid substring searches within massive text corpora. By indexing every possible suffix of a given input string, suffix trees facilitate efficient pattern matching and enable advanced search functionalities such as autocomplete and spell-checking. The underlying data structure permits search operations to be executed in linear time relative to the query length, which is critical for real-time document retrieval. This approach to on-line pattern matching is comprehensively detailed in Ukkonen's work, which lays the foundation for constructing suffix trees in linear time [8].

B. Bioinformatics

In bioinformatics, suffix trees have become an indispensable tool for analyzing biological sequences such as DNA, RNA, and proteins. Their capability to rapidly identify common motifs, repeated subsequences, and perform sequence alignments makes them ideal for handling large genomic datasets. Generalized suffix trees, which can index multiple sequences simultaneously, are particularly useful for detecting the longest common substrings among different organisms, a process that is central to comparative genomics and gene prediction. The application of suffix trees in this context is extensively discussed in Gusfield's seminal work, which has significantly influenced computational approaches in molecular biology [6].

C. Data Compression

Data compression algorithms also benefit from the use of suffix trees, as they can effectively identify redundant patterns and repeated substrings within text data. By leveraging these repeated patterns, compression techniques can build dictionaries that enable efficient substitution and factorization of the input, thereby reducing the overall data size. The ability of suffix trees to perform these operations in linear time makes them a powerful asset in developing high-performance compression schemes. The role of suffix trees in enhancing data compression is elaborated in the research by Grossi and Italiano, which explores various applications of suffix trees in string algorithms [5].

VIII. Conclusion

In conclusion, this paper has provided a comprehensive examination of suffix trees, detailing their structure, key operations, and real-world applications. We demonstrated that suffix trees, particularly when constructed using Ukkonen's algorithm, offer significant advantages for efficient substring searching and pattern matching, which are critical operations in fields such as search engines, bioinformatics, and data compression. Our comparative analysis with alternative data structures, including Tries, Patricia Trees, and Suffix Arrays, highlights that while suffix trees excel in query performance, they also face challenges related to memory consumption and construction complexity. Despite these challenges, ongoing developments in optimizations, such as compressed suffix trees and external-memory approaches, continue to enhance their scalability and practical applicability. Ultimately, the enduring relevance of suffix trees in both theoretical research and practical applications underscores their importance as a foundational tool in modern text processing, paving the way for future innovations that further address their limitations while leveraging their strengths.

Acknowledgments

We would like to express our gratitude to Ph.D Brenner Humberto Ojeda Rios for his outstanding dedication teaching us the Data Structures and Algorithms course.

References

[1] Weiner, P. (1973). Linear Pattern Matching Algorithms. *Scandinavian Workshop on Algorithm Theory*. https://ieeexplore.ieee.org/document/4569722.

[2] Tian, Y., Tata, S., Hankins, R. A., & Patel, J. M. (2005). Practical methods for constructing suffix trees. *The VLDB Journal, 14*(3), 281–299. http://hdl.handle.net/2027.42/47869.

[3] McCreight, E. M. (1976). A Space-Economical Suffix Tree Construction Algorithm. *J. ACM, 23*(2), 262–272. https://doi.org/10.1145/321941.321946.

[4] Ferragina, P. (2008). String algorithms and data structures. *arXiv preprint arXiv:0801.2378*. https://doi.org/10.48550/arXiv.0801.2378.

[5] Grossi, R., & Italiano, G. F. (1993, September). Suffix trees and their applications in string algorithms. In *Proceedings of the 1st South American Workshop on String Processing* (pp. 57–76). https://pages.di.unipi.it/grossi/aiw/survey.pdf.

[6] Gusfield, D. (1997). Algorithms on strings, trees, and sequences: Computer science and computational biology. *ACM SIGACT News (III)*. https://dl.acm.org/doi/pdf/10.1145/270563.571472

[7] HackerEarth. Suffix Trees Tutorial. *HackerEarth Practice Platform*. Available at: https://www.hackerearth.com/practice/data-structures/advanced-data-structures/suffix-trees/tutorial/.

[8] Ukkonen, E. (1995). On-line construction of suffix trees. *Algorithmica, 14*, 249–260. https://doi.org/10.1007/BF01206331.

[9] Morrison, D. R. (1968). PATRICIA—Practical Algorithm To Retrieve Information Coded in Alphanumeric. *J. ACM, 15*(4), 514–534. https://doi.org/10.1145/321479.321481.

[10] GeeksforGeeks. (2021). Pattern Searching using Suffix Tree. *GeeksforGeeks*. Available at: https://www.geeksforgeeks.org/pattern-searching-using-suffix-tree/.

[11] Kingsford, C. S. Suffix Trees. *Carnegie Mellon University, Lecture Notes*. Available at: https://www.cs.cmu.edu/~ckingsf/bioinfo-lectures/suffixtrees.pdf.

[12] Gregg, B. (2014). Ukkonen's Algorithm Animation. *brenden.github.io*. Available at: https://brenden.github.io/ukkonen-animation/.

[13] Schulz, K. S. Suffix Trees. *Stanford University, Lecture Notes*. Available at: https://web.stanford.edu/class/archive/cs/cs166/cs166.1146/lectures/10/Small10.pdf.

[14] Fredkin, E. (1960). Trie memory. *Communications of the ACM, 3*(9), 490–499. https://dl.acm.org/doi/pdf/10.1145/367390.367400.

[15] Manber, U., & Myers, G. (1990). Suffix arrays: A new method for on-line string searches. *SIAM Journal on Computing, 22*(5), 935–948. https://doi.org/10.1137/0222058.

[16] Kärkkäinen, J., & Sanders, P. (2003). Simple linear work suffix array construction. In *Automata, Languages and Programming: 30th International Colloquium, ICALP 2003, Eindhoven, The Netherlands, June 30–July 4, 2003, Proceedings 30* (pp. 943–955). Springer Berlin Heidelberg. https://link.springer.com/chapter/10.1007/3-540-45061-0_73.