Efficient Text Searching

Abhijeet Singh 2023CSB1094 ,
Parashdeep Singh 2023MCB1306 ,
Aadit Mahajan 2023CSB1091

Course Instructor:
Dr. Anil Shukla

TA :
Sravanthi Shede

Overview: We have used data structures like Tries and Suffix Trees, and algorithms like KMP and Finite Automata algorithms to perform pattern searching in a text file. Our code basically gives the output of the line of occurrence of searched pattern and the pattern's starting index in that line. As we are using these various methods to perform pattern searching, so we have done complexity analysis for all the methods in order to determine which method is optimal.

1. Trie and Suffix Tree

Trie
The **Trie data structure** is a tree-like data structure used for storing a dynamic set of strings. It is commonly used for efficient **retrieval** and **storage** of keys in a large dataset. The structure supports operations such as **insertion**, **search**, and **deletion** of keys, making it a valuable tool in fields like computer science and information retrieval. They are particularly useful when we have a large number of strings with significant overlap in their prefixes.

Properties: 1. It contains a single root node.
2. Each node represents a string and each edge represents a character.
3. Every node consists of an array of pointers, with each index representing a character and a flag to indicate if any string ends at the current node.
4. Although, trie data structure can contain any number of characters including alphabets, numbers, and special characters but here we only need 26 pointers for every node, where the 0th index represents 'a' and the 25th index represents 'z' characters.
5. Each path from the root to any node represents a word or string.

1.1. Construction Of Trie

1.1.1 Representation of Trie Node

Every Trie node consists of a character pointer array or hashmap and a flag to represent if the word is ending at that node or not. But if the words contain only lower-case letters (i.e. a-z), then we can define Trie Node with an array instead of a hashmap.

Representation of Trie Node

```
Constructing a trie node,
  struct TrieNode
      TrieNode *children[26];
      bool isEndOfWord;
```

Creating a new Trie Node

```
Initializing a trie node,
    struct TrieNode* node;
    node->isEndOfWord=false;
    for i in range(26):
        node->children[i]=NULL
```

1.1.2 Insertion in Trie

Algorithm 1. Define a function insert(TrieNode *root, string word) which will take two parameters one for the root and the other for the string that we want to insert in the Trie data structure.
2. Now take another pointer currentNode and initialize it with the root node.
3. Iterate over the length of the given string and check if the value is NULL or not in the array of pointers at the current character of the string.
4. If It's NULL then, make a new node and point the current character to this newly created node. Move the currentNode pointer to the newly created node.
5. Finally, change the flag value of "isEndOfWord" of the last currentNode to true, this implies that there is a string ending currentNode.

Insertion in Trie (struct TrieNode *root, string key)

```
struct TrieNode *node = root;
for $1 \leq i \leq key.length$ do
    int index = $key[i]-$ 'a';
    if $!node->children[index]$ then
        $node->children[index] = getNode();$
    end if
    $node = node->children[index]$
end for
$node->isEndofWord = true$
```

For example, let us take a look at a trie and see how words are stored in it.
1. Store "and" in Trie data structure:
The word "and" starts with "a", so we will mark the position "a" as filled in the Trie node, which represents the use of "a". After placing the first character, for the second character, again there are 26 possibilities. So from "a", there is an array of size 26 for storing the 2nd character. The second character is "n", so from "a", we move to "n" and mark "n" in the 2nd array as used. The third character is "d", so we mark the position "d" as used in the respective array.
2. Store "ant" in the Trie data structure:
The word "ant" starts with "a", and the position of "a" in the root node has already been filled. So, no need to fill it again; just move to the node 'a' in the Trie. For the second character 'n', we observe that the position of 'n' in the 'a' node has already been filled. No need to fill it again; just move to the node 'n' in the Trie. For the last character 't' of the word, the position for 't' in the 'n' node is not filled. So, fill the position of 't' in the 'n' node and move to the 't' node.

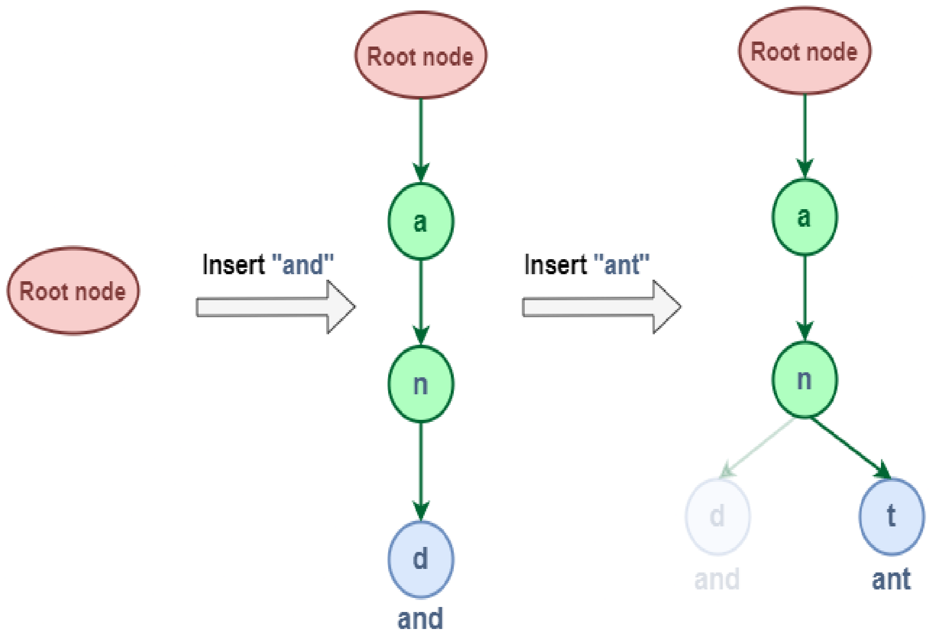


Figure 1: Representation of how words are stored in Trie

1.1.3 Searching in Trie

Search operation in Trie is performed in a similar way as the insertion operation but the only difference is that whenever we find that the array of pointers in currentNode does not point to the current character of the word then we return false instead of creating a new node for that current character of the word. This operation is used to search whether a string is present in the Trie data structure or not. There are two search approaches in the Trie data structure.
1. Find whether the given word exists in Trie.
2. Find whether any word that starts with the given prefix exists in Trie.
There is a similar search pattern in both approaches. The first step in searching a given word in Trie is to convert the word to characters and then compare every character with the trie node from the root node. If the current character is present in the node, move forward to its children. Repeat this process until all characters are found.

```
Searching in Trie ( struct TrieNode *root, string key )
  **Initialize:** struct TrieNode *node = root
  **for** $i = 0$ to $key.length - 1$ **do**
    int index = key[i] − 'a'
    **if** $node \rightarrow children[index]$ = NULL **then**
        **return** false
    **end if**
  node = node $\rightarrow children[index]$
  **end for**
  **return** $(node \rightarrow isEndOfWord)$
```

Time Complexity The time taken by this algorithm is $O(n^2)$ where 'n' is the length of the text. This time is essentially taken to build the trie. Note that this is one time activity and subsequent searches of another pattern in this text would take $O(m)$ time where m is the length of the pattern.

Suffix Tree
A Suffix Tree is basically a compressed trie for all suffixes of the given text. The construction of such a tree for

the string S takes time and space linear in the length of S.

Properties 1.The tree has exactly n leaves numbered from 1 to n.
2. Except for the root, every internal node has at least two children.
3. Each edge is labeled with a non-empty substring of S.
4. No two edges starting out of a node can have string-labels beginning with the same character
5. The string obtained by concatenating all the string-labels found on the path from the root to the leaf i spells out suffix $S[i...n]$, for i from 1 to n.

How to build a Suffix Tree for a given text?
1. Generate all suffixes of given text.
2. Consider all suffixes as individual words and build a compressed trie.

Let us consider an example text "banana\0" where '\0' is a string termination character.

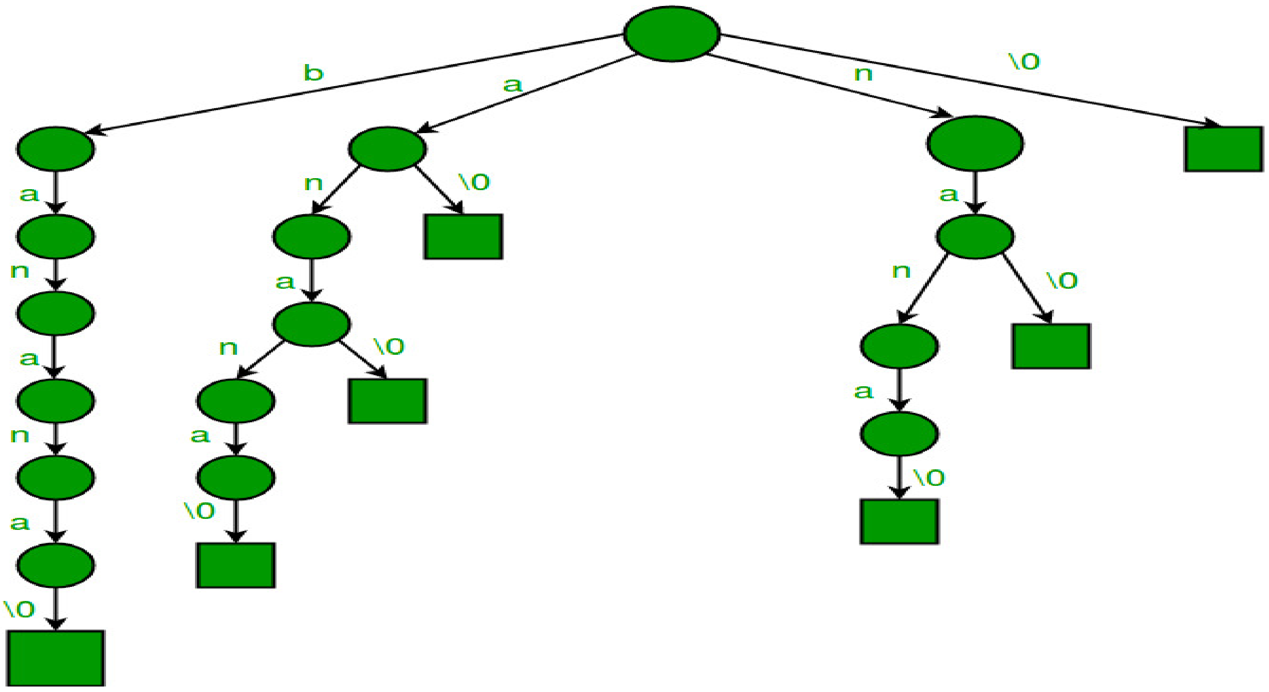


Figure 2: Creation of Trie

If we join chains of single nodes, we get the following compressed trie, which is the Suffix Tree for given text "banana\0"

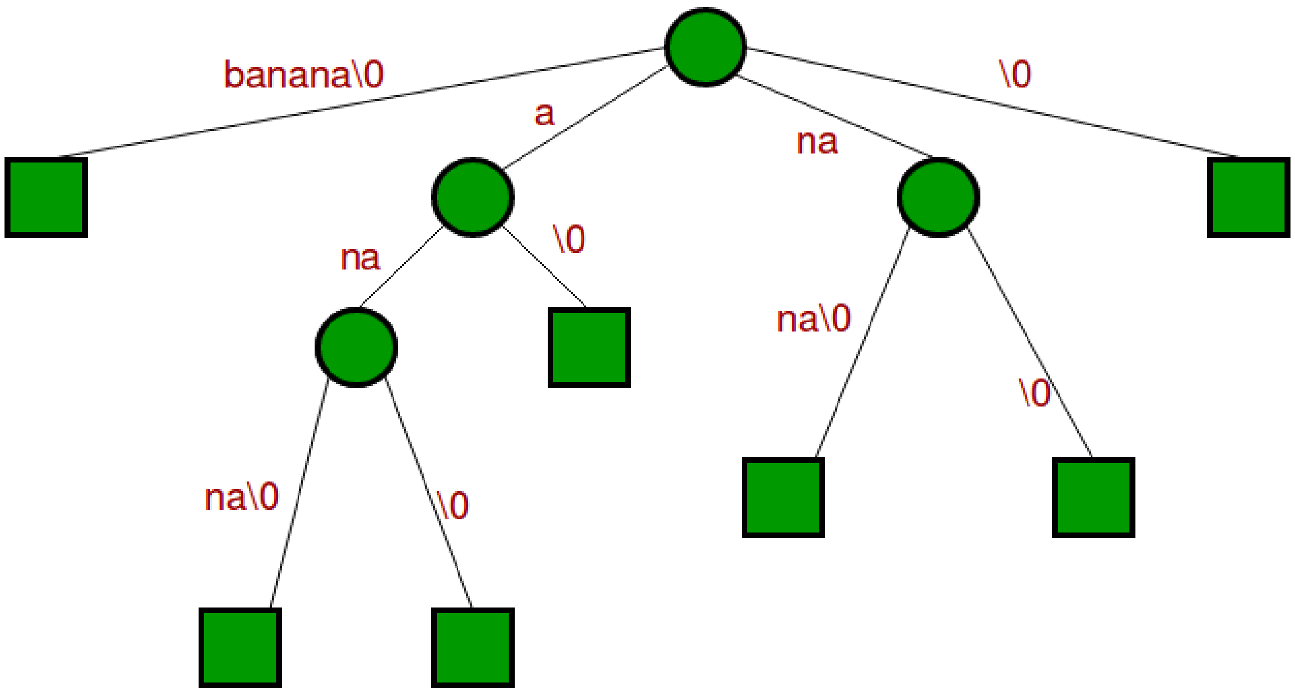


Figure 3: Creation of Suffix Tree by compressing Trie

How to search a pattern in the built suffix tree?
1. Starting from the first character of the pattern and root of the Suffix Tree, do the following for every character:
For the current character of the pattern, if there is an edge from the current node of suffix tree, follow the edge. If there is no edge, print "pattern doesn't exist in the text" and return.
2. If all characters of pattern have been processed, i.e., there is a path from root for characters of the given pattern, then print "Pattern found". Let us consider the example pattern as "nan" to see the searching process. Following diagram shows the path followed for searching "nan" or "nana".
Every pattern that is present in text (or we can say every substring of text) must be a prefix of one of all possible suffixes.

2. Knuth–Morris–Pratt algorithm

The KMP matching algorithm uses degenerating property (pattern having the same sub-patterns appearing more than once in the pattern) of the pattern and improves the worst-case complexity to O(n+m).
The basic idea behind KMP's algorithm is: whenever we detect a mismatch (after some matches), we already know some of the characters in the text of the next window. We take advantage of this information to avoid matching the characters that we know will anyway match.

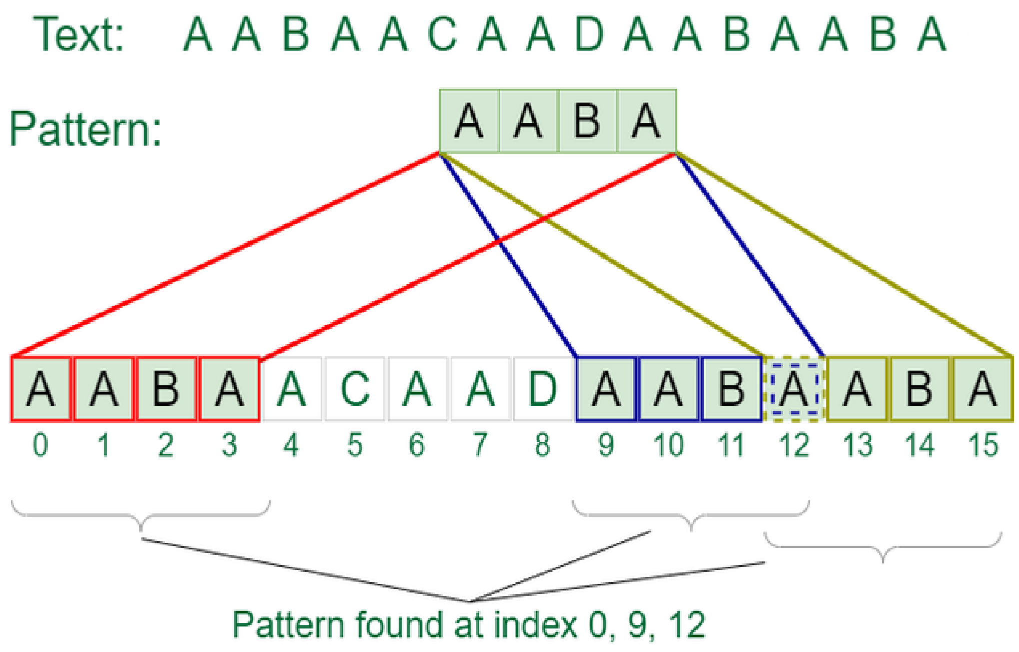

Text: A A B A A C A A D A A B A A B A
Pattern: A A B A
A A B A A C A A D A A B A A B A
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
Pattern found at index 0, 9, 12

Figure 4: Pattern searching using KMP's algorithm

Preprocessing overview
• KMP algorithm preprocesses pat[] and constructs an auxiliary lps[] of size m (same as the size of the pattern) which is used to skip characters while matching.
• Name lps indicates the longest proper prefix which is also a suffix. A proper prefix is a prefix with a whole string not allowed. For example, prefixes of "ABC" are "", "A", "AB" and "ABC". Proper prefixes are "", "A" and "AB". Suffixes of the string are "", "C", "BC", and "ABC".
• We search for lps in subpatterns. More clearly we focus on sub-strings of patterns that are both prefix and suffix.
• For each sub-pattern pat[0..i] where i = 0 to m-1, lps[i] stores the length of the maximum matching proper prefix which is also a suffix of the sub-pattern pat[0..i].

Preprocessing algorithm
• We calculate values in lps[]. To do that, we keep track of the length of the longest prefix suffix value (we use len variable for this purpose) for the previous index
• We initialize lps[0] and len as 0.
• If pat[len] and pat[i] match, we increment len by 1 and assign the incremented value to lps[i].
• If pat[i] and pat[len] do not match and len is not 0, we update len to lps[len-1]

KMP algorithm

1. We start the comparison of pat[j] with j = 0 with characters of the current window of text.
2. We keep matching characters txt[i] and pat[j] and keep incrementing i and j while pat[j] and txt[i] keep matching.
3. When we see a mismatch

   We know that characters pat[0..j-1] match with txt[i-j…i-1] (Note that j starts with 0 and increments it only when there is a match).

   We also know (from the above definition) that lps[j-1] is the count of characters of pat[0…j-1] that are both proper prefix and suffix.

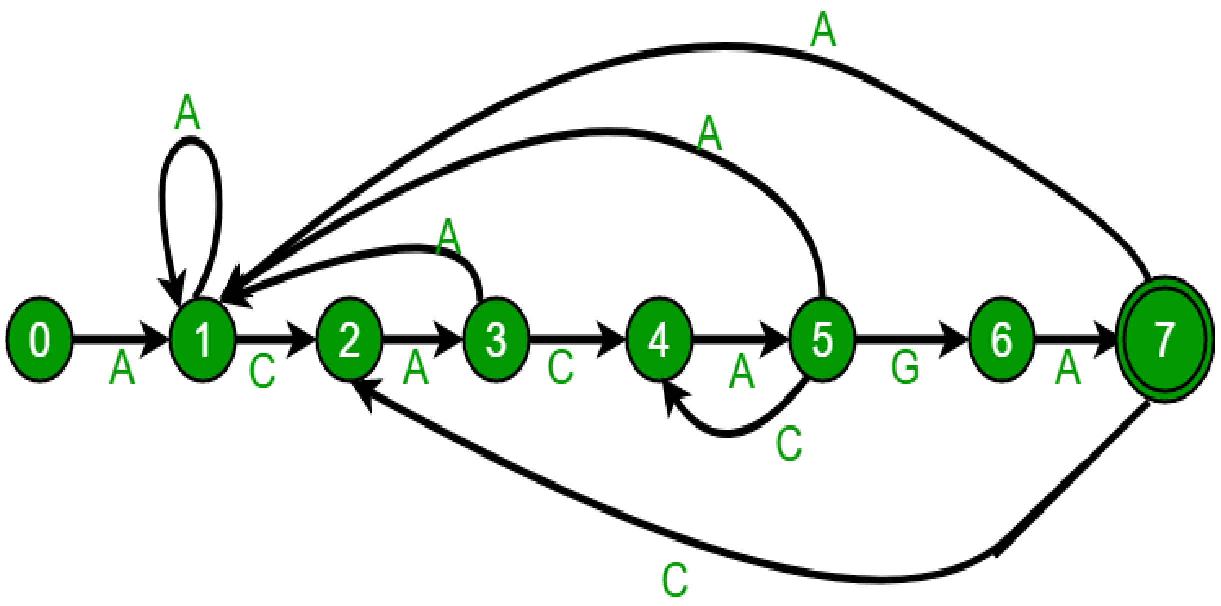
3. Finite Automata algorithm

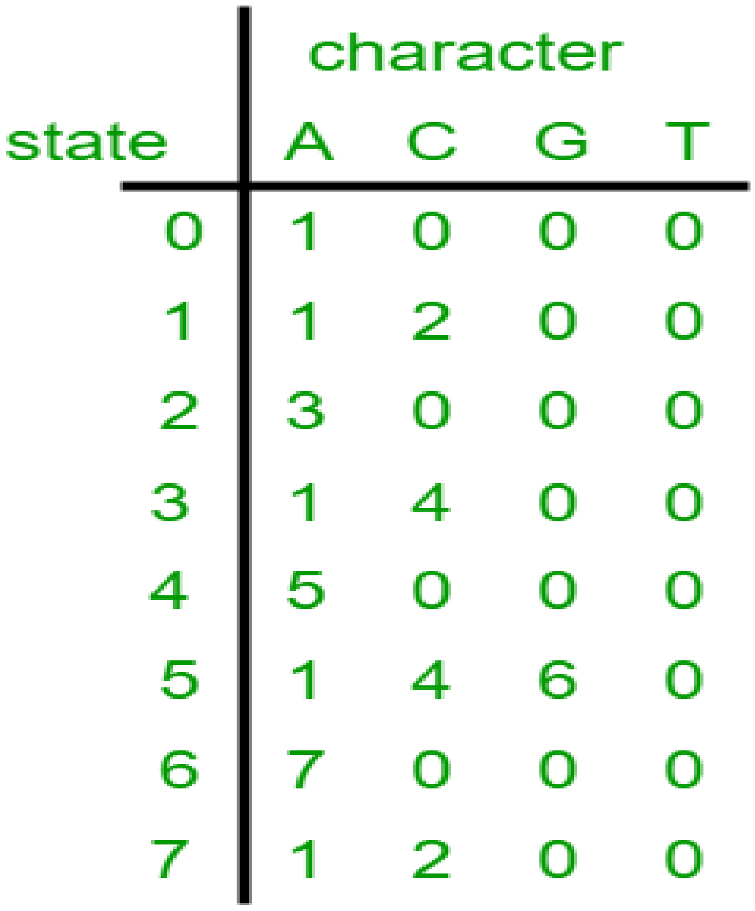
1. Idea of this approach is to build finite automata to scan text T for finding all occurrences of pattern P. 2. This approach examines each character of text exactly once to find the pattern. Thus it takes linear time for matching but preprocessing time may be large. 3. It is defined by tuple M = Q, $\sum$, q, F, σ

Where Q = Set of States in finite automata
$\sum$=Sets of input symbols
q = Initial state
F = Final State
σ = Transition function
Time Complexity = O(M³|$\sum$|)
A finite automaton M is a 5-tuple (Q, q0,A,$\sum$,δ), where
Q is a finite set of states, q0 ϵ Q is the start state, A subset of Q is a notable set of accepting states, σ is a finite input alphabet, δ is a function from Q x σ into Q called the transition function of M.
The finite automaton starts in state q0 and reads the characters of its input string one at a time. If the automaton is in state q and reads input character a, it moves from state q to state δ (q, a). Whenever its current state q is a member of A, the machine M has accepted the string read so far. An input that is not allowed is rejected.
A finite automaton M induces a function ϕ called the called the final-state function, from ϵ* to Q such that ϕ(w) is the state M ends up in after scanning the string w. Thus, M accepts a string w if and only if ϕ(w) ϵ A.

Finite automata algorithm

```
FINITE AUTOMATA (T, P)
State <- 0
for l <- 1 to n
State <- δ(State, ti)
If State == m then
Match Found
else
end
```





	character			
state	A	C	G	T
0	1	0	0	0
1	1	2	0	0
2	3	0	0	0
3	1	4	0	0
4	5	0	0	0
5	1	4	6	0
6	7	0	0	0
7	1	2	0	0

Figure 5: Finite automation for ACACAGA pattern

4. Aho-Corasick Algorithm

Given an input text and an array of k words, arr[], find all occurrences of all words in the input text. Let n be the length of text and m be the total number of characters in all words, i.e. m = length(arr[0]) + length(arr[1]) + . . . + length(arr[k-1]). Here k is total numbers of input words.
If we use a linear time searching algorithm like KMP, then we need to one by one search all words in text[]. This gives us total time complexity as O(n + length(word[0]) + O(n + length(word[1]) + O(n + length(word[2]) + . . . O(n + length(word[k-1]). This time complexity can be written as O(n*k + m).
Aho-Corasick Algorithm finds all words in O(n + m + z) time where z is total number of occurrences of words in text.

Algorithm Overview

This algorithm is a **preprocessing** approach that processes the text in three main components:

1. **Go To:** This function follows the edges of a Trie constructed from all words in `arr[]`. Represented as a 2D array $g[][]$, it stores the next state for a given state and character.
2. **Failure:** This function manages the fallback edges when the current character lacks a direct edge in the Trie. Represented as a 1D array $f[]$, it stores the next state for each state when no matching edge exists.
3. **Output:** Contains indexes of all words that end at the current state, stored in a 1D array $o[]$ as a bitmap indicating all matches for a given state.

Preprocessing the Text to Generate Automaton

- Build a Trie (or Keyword Tree) for all words, filling entries in the `goto` array $g[][]$ and `output` array $o[]$.

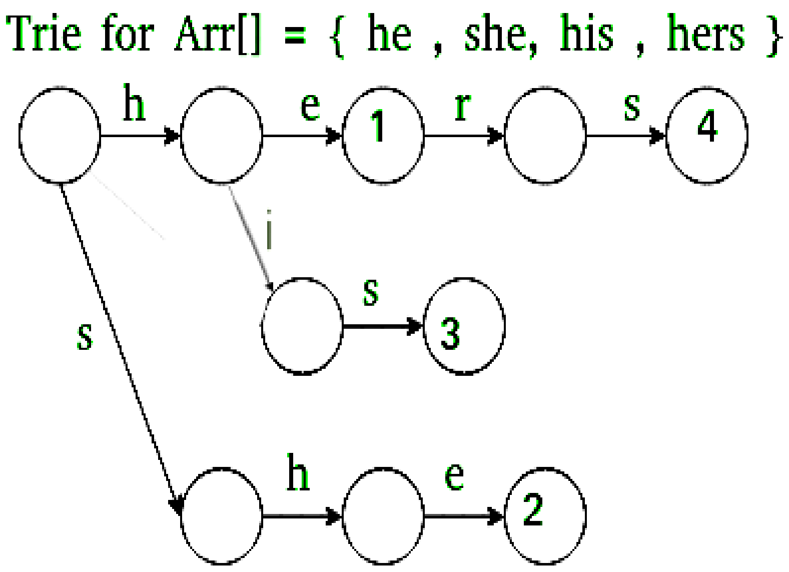
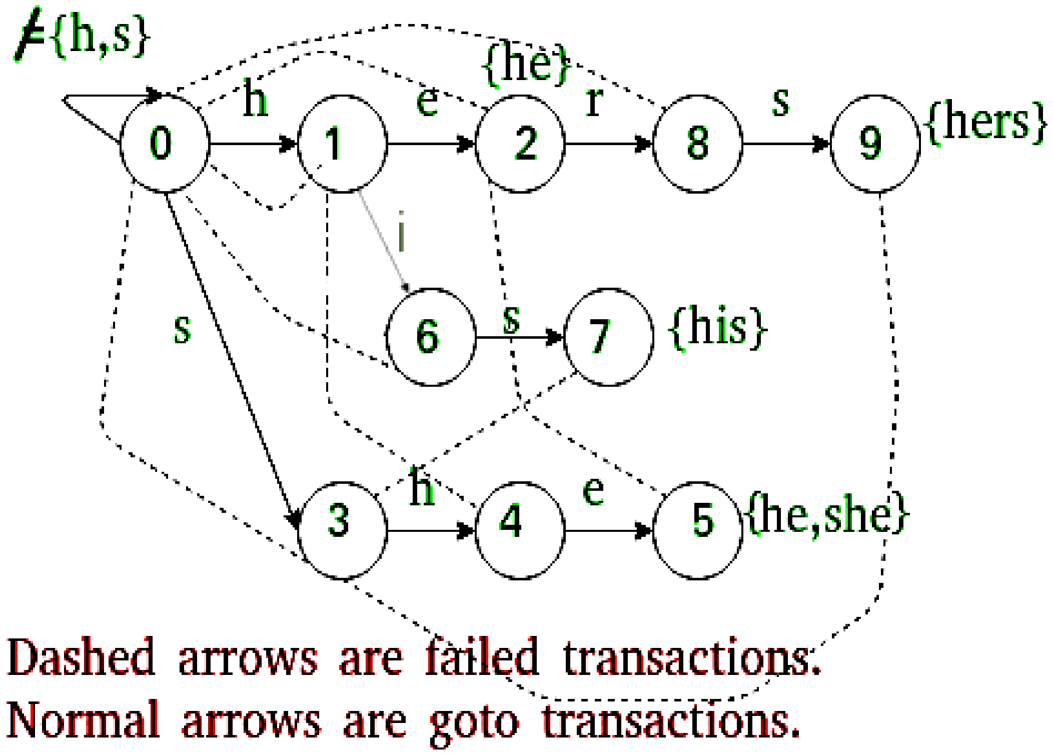


Figure 6: Trie

- Extend the Trie into an automaton to enable linear-time matching, filling entries in the `failure` array $f[]$ and updating `output` $o[]$.



≠{h,s}
0
h
1
e
2
{he}
r
8
s
9
{hers}
i
6
s
7
{his}
s
3
h
4
e
5
{he,she}
Dashed arrows are failed transactions.
Normal arrows are goto transactions.

Figure 7: Automaton

Detailed Steps

Go To: Construct the Trie. For characters lacking an edge at the root, add an edge back to the root.

Failure: For each state s, find the longest proper suffix that is also a prefix of a pattern using a Breadth-First Traversal (BFS) of the Trie.

Output: Store indexes of all words ending at each state s as a bitwise map (by performing bitwise OR operations). This is also computed using BFS with Failure.

Matching: Traverse the given text over the built automaton to locate all matching words.

5. Application

5.1. General Applications

Text Searching and Information Retrieval: Pattern searching algorithms are fundamental in information retrieval systems, such as search engines, where they help users find relevant documents, web pages, or content based on search queries or keywords.
Data Mining and Text Analysis: In data mining and text analysis, pattern searching is essential for extracting meaningful information from large datasets, including text data. This is used in sentiment analysis, topic modeling, and identifying trends.
Genomics and Bioinformatics: Pattern searching is crucial in analyzing DNA and protein sequences, finding specific motifs, detecting gene sequences, and identifying mutations or variations in biological data.
Text Compression: Pattern searching can be used in text compression techniques, such as finding repeated patterns to compress text more efficiently.
Network Security: Intrusion detection systems and network security applications use pattern searching algorithms to identify known attack patterns or suspicious network activities in log files and network traffic.
Image Processing: In image processing, pattern matching is used for object recognition, image retrieval, and locating specific features within images.
Spell Checkers and Autocorrection: Spell checkers and autocorrection systems use pattern searching to identify misspelled words and suggest corrections based on similar patterns.
File and Text Editing: Text editors and search tools often employ pattern searching algorithms to help users locate and edit text within documents or source code.

5.2. Technique Specific Applications

- **Trie**
  - **Search Engines**: Tries are used for autocomplete suggestions and predictive text by organizing and retrieving stored words based on prefixes.
  - **IP Routing**: In network routing, Tries store IP addresses and optimize routing paths.
  - **Spell Checkers**: Tries provide efficient spell correction and autocorrect by quickly checking for word presence or suggesting alternatives.
- **Suffix Trees**
  - **Genomics**: Suffix trees are used in DNA sequence alignment, allowing for rapid searches of genetic patterns within large genomes.
  - **Plagiarism Detection**: They help identify matching sequences between documents by efficiently finding repeated or similar phrases.
  - **Data Compression**: Suffix trees are instrumental in detecting repeating substrings, which is crucial in compression algorithms like Lempel-Ziv.
- **Knuth-Morris-Pratt (KMP) Algorithm**
  - **Word Processing Applications**: Employed for find-and-replace functions by efficiently locating patterns within text.
  - **Bioinformatics**: Facilitates search for specific gene sequences or motifs in DNA strands.
  - **Intrusion Detection Systems**: Used to detect known signatures or patterns in network traffic, ensuring quick threat detection.
- **Finite Automata**
  - **Lexical Analysis in Compilers**: Matches keywords and tokens in source code by using state transition models.
  - **Speech Recognition**: Matches audio input to known phoneme sequences or words, aiding in interpreting continuous speech.
  - **Text Editors**: Enables fast search and navigation through patterns within large documents.
- **Aho-Corasick Algorithm**
  - **Web Content Filtering**: Blocks or flags inappropriate content by searching for a set of banned

words across web pages.
  - **Spam Detection**: Detects specific keywords within email content that match spam patterns, providing real-time filtering.
  - **Digital Forensics**: Locates critical evidence across large data sets by searching for multiple keywords simultaneously.

6. Complexity Analysis

We have conducted a complexity analysis for each pattern searching technique utilized in our project. Let:

- n = length of the given text file
- m = length of the pattern to be searched
- l = average length of each line in the text file

6.1. Trie

- **Space Complexity:** $O(n^2)$ due to the formation of the trie from preprocessing the text file.
- **Time Complexity:** $O(n^2)$, dominated by trie creation; pattern searching takes $O(m)$.

6.2. Suffix Tree (Naive Approach)

- **Space Complexity:** In the worst case, for a string with all unique characters, the suffix tree could have $O(n^2)$ nodes, as each suffix could lead to a new node. In the average case, it could be $O(n \cdot l)$.
- **Time Complexity:** Building the suffix tree takes $O(n \cdot l^2)$; searching the pattern takes $O(m)$.

6.3. KMP Algorithm

- **Space Complexity:** $O(n)$
- **Time Complexity:** $O(n + m)$

6.4. Finite Automata

- **Space Complexity:** $O(n \cdot l + m \cdot 256) \approx O(n)$
- **Time Complexity:** $O(n \cdot l + m \cdot 256) \approx O(n)$

6.5. Aho-Corasick Algorithm

- **Space Complexity:** $O(m)$
- **Time Complexity:** $O(m + n)$

6.6. Performance Graphs

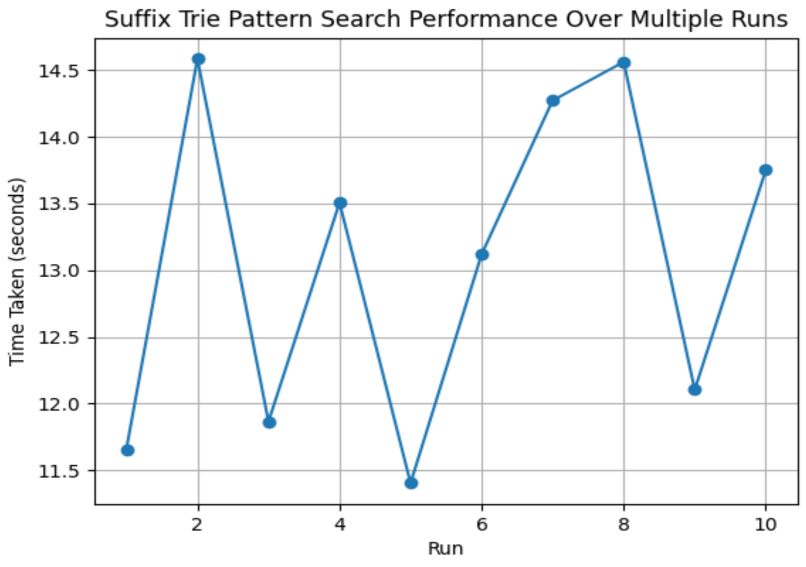


Figure 8: Performance of the Trie Algorithm

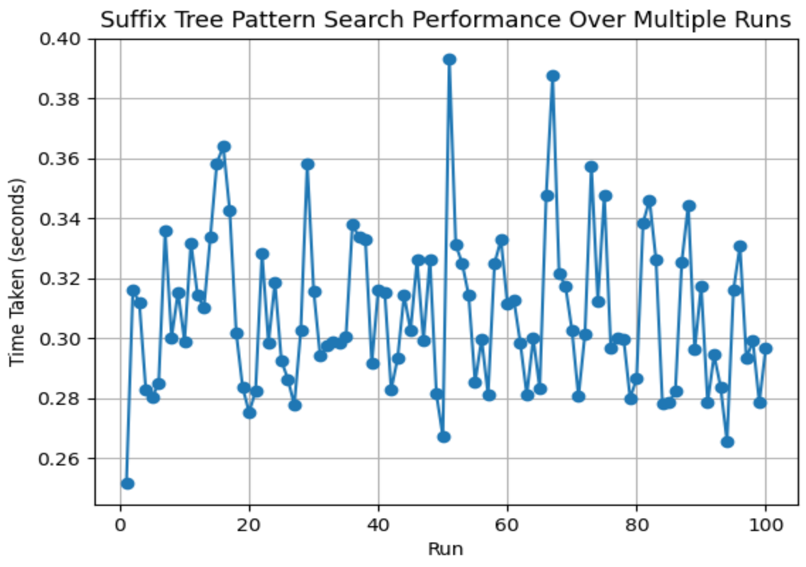


Figure 9: Performance of the Suffix Tree (Naive Approach)

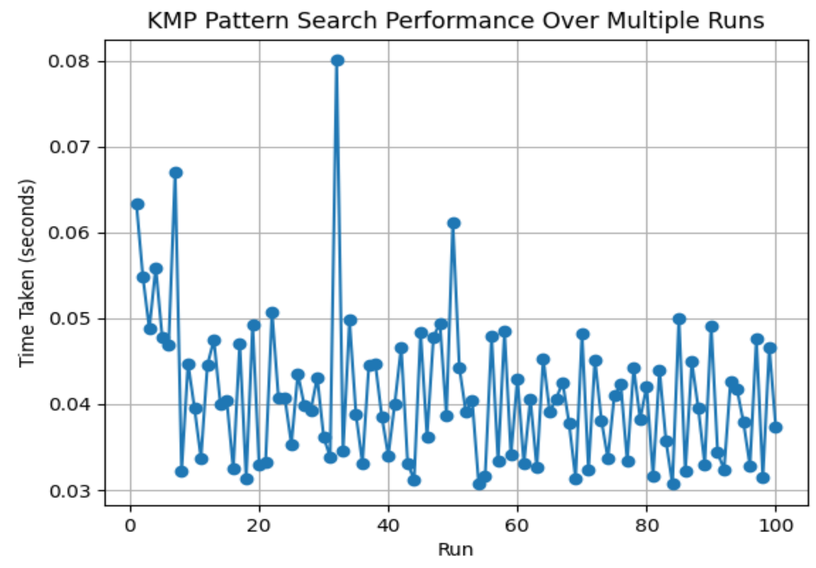


Figure 10: Performance of the KMP Algorithm

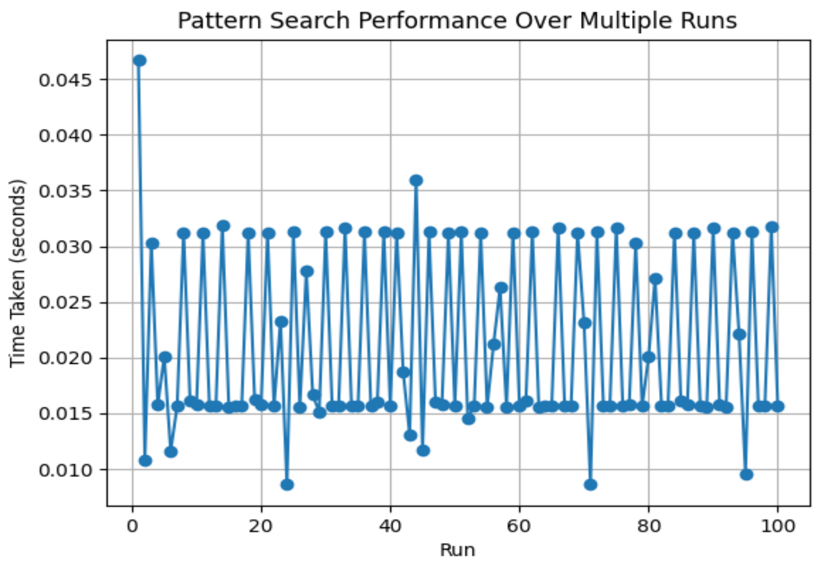


Figure 11: Performance of the Finite Automata Algorithm

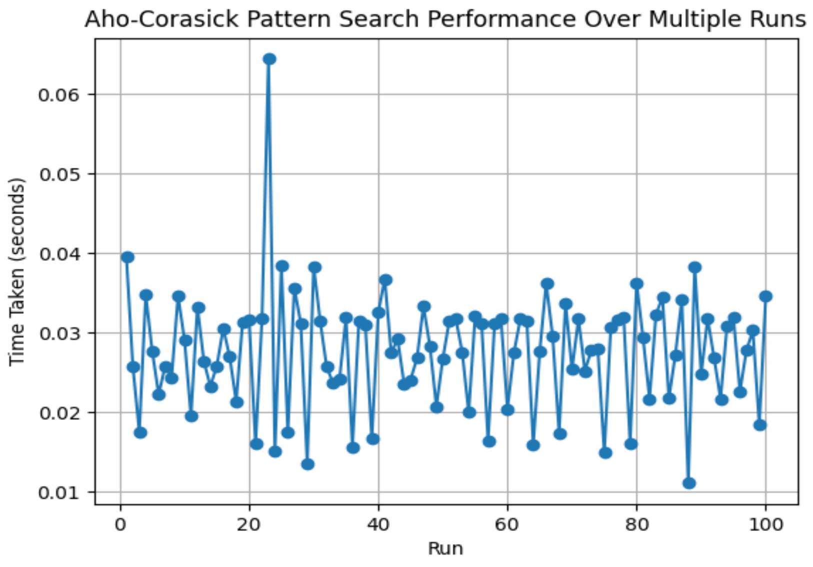


Figure 12: Performance of the Aho-Corasick Algorithm

7. Conclusion

In conclusion, the project on string pattern searching has provided us with a comprehensive understanding of the techniques, algorithms, and applications associated with this fundamental concept in computer science. Throughout this endeavor, we have explored a range of string searching algorithms, Knuth-Morris-Pratt, Finite automata, and more, each with its unique advantages and trade-offs. We have witnessed how string pattern searching plays a pivotal role in various domains. Furthermore, we have delved into the underlying data structures, like suffix trees.
Each method offers unique strengths: Tries for prefix searches, Suffix Trees for full-text searches, KMP and Finite Automata for efficient single-pattern matching, and Aho-Corasick for multi-pattern searches. Together, these methods provide scalable, adaptable solutions for diverse text-searching requirements.

8. Bibliography and Citations

Aho-Corasick Algorithm
Suffix tree
Tries
KMP Algorithm
Finite Automata algorithm

9. References

CTAN. BiBTeX documentation.
Leslie Lamport. LaTeX: A Document Preparation System. Pearson Education India, 1994.
Thomas H. Cormen(CLRS). Introduction to algorithm. *Library of Congress Cataloging-in-Publication.* 1990