Comparision on various searching strategies for Auto seggestion

November 11, 2024

Bhavika Sharma (2023MCB1290) ,
Sukriti Bhandari (2023MCB1314) ,
Vaishnavi Pal (2023MCB1318)

Instructor:
Dr. Anil Shukla

Teaching Assistant:
Abdul Razique

Summary: Our project explores efficient techniques for pattern searching and auto-suggestion within text files, utilizing advanced data structures and algorithms to enhance search functionality. We implemented Trie and Suffix Tree data structures for optimized pattern searches, along with the Knuth-Morris-Pratt (KMP) and Finite Automata algorithms, to compare their performance in terms of time complexity. Each method was assessed to determine its effectiveness and efficiency for specific search tasks. Additionally, an auto-suggestion feature was integrated to provide real-time word suggestions as input is typed. The output includes details such as line numbers and the frequency of pattern occurrences, making this tool valuable for both precise pattern matching and responsive, predictive text support.

1. Introduction

1.1 Trie

The Trie is a tree-based data structure specifically designed for storing collections of strings and enabling efficient search operations on them. Tries are especially useful for applications that involve string matching, such as text autocompletion, dictionary lookups, and IP routing in networking. They provide space efficiency by sharing common prefixes among strings, making them ideal when dealing with large sets of strings that have overlapping prefixes.

Key Properties of a Trie

1. **Single Root Node**: Each Trie contains one root node, from which all strings in the collection are represented.
2. **Nodes Represent Strings; Edges Represent Characters**: Each node in a Trie represents a string, while each edge corresponds to a character that builds the path for a specific string.
3. **Storage Mechanism in Nodes**: Nodes in a Trie consist of either hashmaps or arrays of pointers. Each pointer represents a character, and a flag is often included to indicate the end of a string.
4. **Character Flexibility**: A Trie can handle a wide range of characters, including letters, numbers, and symbols. For example, a Trie storing only lowercase English alphabets (a-z) requires an array of 26 pointers at each node, with index 0 representing 'a' and index 25 representing 'z'.
5. **Paths Represent Words**: Each unique path from the root to any node within the Trie represents a complete word or string.

Example of Storing Words in a Trie Let's illustrate how words are stored in a Trie with the examples of "and" and "ant."

1. **Storing the Word "and"**
   - **Step 1**: Begin at the root. The first character of "and" is 'a', so mark the position for 'a' in the root node as filled. This denotes the usage of 'a'.
   - **Step 2**: Move to the node representing 'a'. Here, an array of 26 positions (one for each letter) is available. The next character in "and" is 'n', so we mark the 'n' position within the array for 'a' as filled.
   - **Step 3**: Proceed to the node for 'n'. The final character in "and" is 'd', so we mark the position for 'd' as used in the array at the 'n' node, completing the storage for "and."

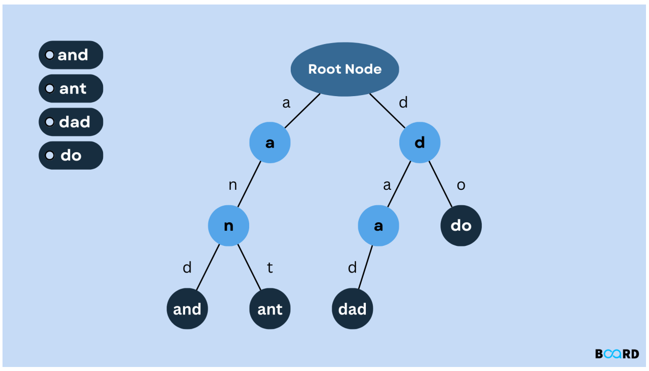


Figure 1: Simple Trie Data Structure

2. **Storing the Word "ant"**
   - **Step 1**: Start at the root. The first character, 'a', is already marked as filled from the previous insertion of "and," so we move directly to the 'a' node.
   - **Step 2**: At the 'a' node, the 'n' position is already marked due to "and," so we move directly to the 'n' node.
   - **Step 3**: For the final character 't' in "ant," the position within the array at the 'n' node is currently unmarked. We mark this position to complete the storage of "ant" in the Trie.

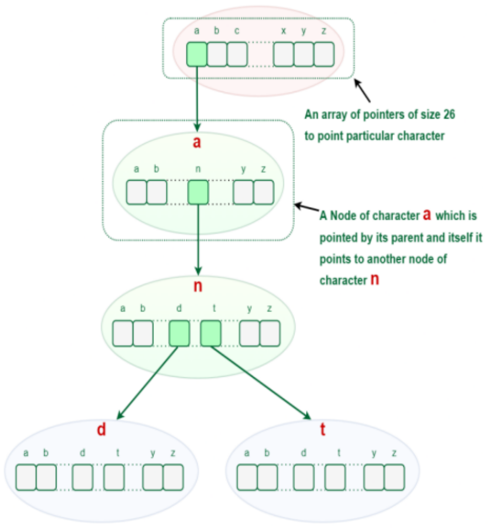

a b c x y z
An array of pointers of size 26
to point particular character
a
a b n y z
A Node of character a which is
pointed by its parent and itself it
points to another node of
character n
n
a b d t y z
d
a b d t y z
t
a b d t y z

Figure 2: Representation of How Words are Stored in Trie

1.1.1 Construction of Trie

A Trie is constructed by organizing nodes in a tree-like structure to store and retrieve strings efficiently. Each node contains an array or hashmap to store character pointers, and a flag to indicate whether a string ends at that node. For simplicity, if the Trie stores only lowercase letters (a-z), an array of 26 pointers can replace a hashmap for efficient memory use.

Representation of a 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. However, if the words contain only lower-case letters (i.e. a-z), we can define a Trie node with an array instead of a hashmap.

1.1.2 Insertion in Trie

Algorithm: Insertion

1. **Function Definition**: Define a function `insert(TrieNode *root, string word)`, which takes two parameters: the root of the Trie and the string to be inserted.
2. **Pointer Initialization**: Initialize a pointer, `currentNode`, to point to the root node of the Trie.
3. **Character Iteration**: Iterate over each character in the input string: For each character, check if the corresponding entry in the array of pointers at `currentNode` is NULL.
4. **Node Creation**: If the pointer is NULL, create a new TrieNode and update the pointer to point to this newly created node. Move `currentNode` to the newly created node.
5. **End of Word Marking**: Once all characters have been processed, increment the `wordCount` of the last `currentNode` to signify that a complete word ends at this node.

1.1.3 Searching in Trie

The search operation in a Trie is executed in a manner analogous to the insertion operation, with a key distinction: instead of creating new nodes when a character is not found, the search operation terminates early, returning false.

Search Functionality: The primary purpose of the search operation is to determine whether a specific string exists within the Trie data structure. There are two primary search approaches:

1. **Exact Word Search**: This approach checks for the existence of a specific word in the Trie.
2. **Prefix Search**: This approach verifies if any words in the Trie start with a specified prefix.

Search Procedure:

- **Character Comparison**: The search begins by converting the input word into its constituent characters and initiating a comparison with the nodes in the Trie, starting from the root.
- **Traversal**: For each character:
  - If the character exists in the current node's array of pointers, the search progresses to the corresponding child node.
  - If any character is absent (i.e., the pointer is NULL), the search terminates, and the function returns false, indicating that the word or prefix is not present in the Trie.
- **Completion Check**: If all characters are successfully traversed and the last node is marked as an end-of-word, the search returns true, confirming the presence of the complete word.

By leveraging the hierarchical structure of the Trie, the search operation efficiently determines the existence of words and prefixes, making it a powerful tool for various applications in string matching and autocomplete features.

Time Complexity of the Algorithm 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.

1.2 Suffix Tree

A **Suffix Tree** for a given text is a specialized data structure that represents all suffixes of that text in a compressed trie format. The construction of a suffix tree for a string SSS operates in linear time and space relative to the length of SSS.

Properties of Suffix Trees

1. **Leaf Nodes**: The tree contains exactly nnn leaves, where nnn represents the length of the text.
2. **Internal Nodes**: Every internal node, except the root, must have at least two children, ensuring a branching structure.
3. **Edge Labels**: Each edge in the tree is labeled with a non-empty substring of SSS, representing a segment of the original text.
4. **Unique Edge Labels**: No two edges emerging from a single node can have string labels that start with the same character, facilitating efficient traversal.
5. **Suffix Representation**: The string obtained by concatenating all string labels along the path from the root to the iii-th leaf corresponds to the suffix S[i. . . n]S[i \ldots n]S[i. . . n], for iii ranging from 1 to nnn.

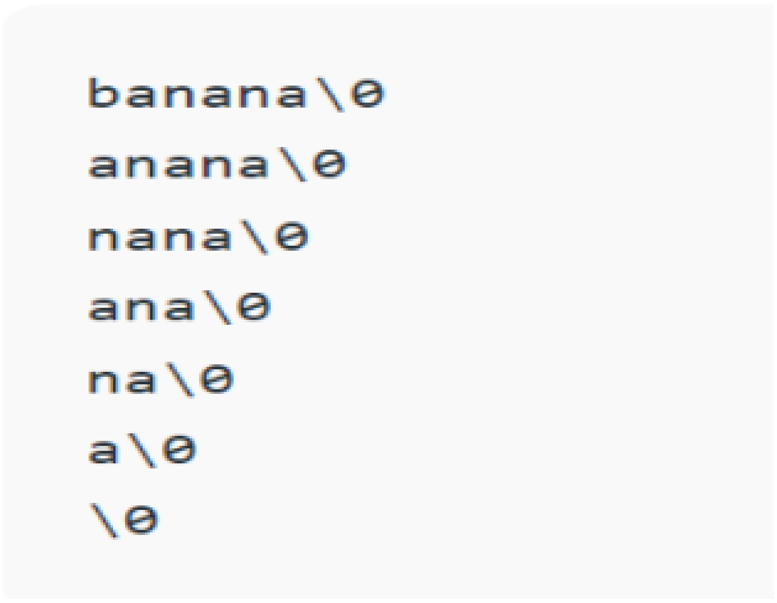
1.2.1 Construction of a Suffix Tree

To build a suffix tree for a given text, follow these steps:

1. **Suffix Generation**: Generate all possible suffixes of the given text.

2. **Compressed Trie Creation**: Treat each suffix as an individual string and construct a compressed trie to represent these suffixes.

Example: CLet us consider an example text “banana\0” where ‘\0’ is a string termination character. Following are all suffixes of “banana\0



```
banana\0
anana\0
nana\0
ana\0
na\0
a\0
\0
```

Figure 3: Representation of how words are stored in trie

This comprehensive construction and property set makes suffix trees a powerful tool for various string processing applications, including substring search, pattern matching, and data compression.
If we join chains of single nodes, we get the following compressed trie, which is the Suffix Tree for given text “banana\0”

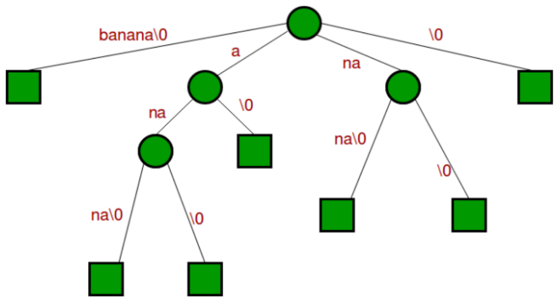


Figure 4: Creation of Suffix Tree

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.
2. For the current character of the pattern, if there is an edge from the current node of suffix tree, follow the edge.
3. If there is no edge, print “pattern doesn’t exist in the text” and return.
4. If all characters of the pattern have been processed, i.e., there is a path from the root for characters of the given pattern, then print “Pattern found”.
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.
Every substring of the text must be a prefix of at least one of the suffixes in the suffix tree, which allows for efficient pattern searching.

1.3 Knuth–Morris–Pratt Algorithm

The Knuth-Morris-Pratt (KMP) matching algorithm utilizes the degenerative property of patterns, where sub-patterns may repeat, allowing it to achieve a worst-case time complexity of O(n+m)O(n + m)O(n+m). The core idea of KMP is to leverage information from previous character matches when a mismatch occurs. Instead of starting over with the pattern, the algorithm uses a precomputed "partial match" or "prefix" table to determine the next positions to compare in the pattern. This approach effectively reduces redundant comparisons, resulting in a more efficient string matching process.

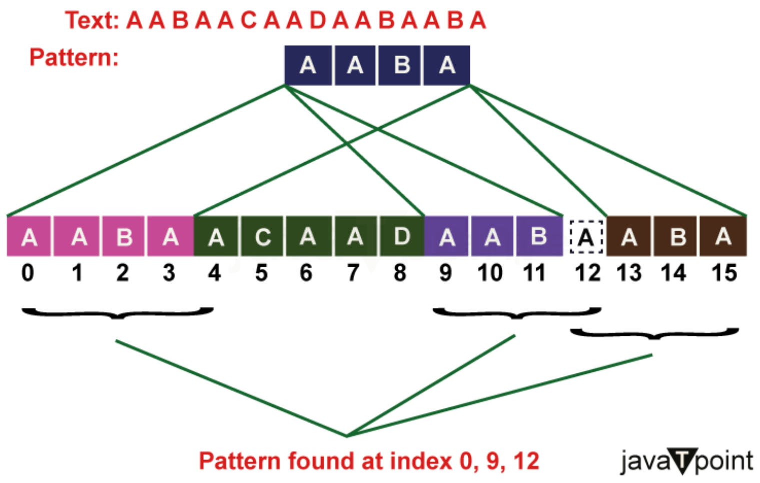

Text: A A B A A C A A D A A B A A B A
Pattern:
A A B A
A A B A A C A A D A A B A A B A
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
Pattern found at index 0, 9, 12
javaTpoint

Figure 5: Pattern searching using KMP's algorithm

1.3.1 Preprocessing Overview

The Knuth-Morris-Pratt (KMP) algorithm utilizes a preprocessing step to enhance the efficiency of string matching. This step involves analyzing the pattern (denoted as `pat[]`) to construct an auxiliary array called `lps[]`, which stands for "longest proper prefix which is also a suffix." The size of the `lps[]` array is equal to the length of the pattern, `m`.
The `lps[]` array plays a crucial role in the KMP algorithm by enabling the search process to skip certain characters, thereby avoiding unnecessary comparisons when a mismatch occurs. To clarify, a proper prefix of a string is defined as any prefix that does not include the entire string itself. For example, the string "ABC" has the following prefixes: "", "A", "AB", and "ABC". Among these, the proper prefixes are "", "A", and "AB". The suffixes of the string include "", "C", "BC", and "ABC".
The `lps[]` array focuses on identifying sub-patterns within the main pattern that serve as both prefixes and suffixes. Specifically, for each sub-pattern `pat[0..i]`, where `i` ranges from `0` to `m-1`, the value of `lps[i]` stores the length of the longest proper prefix that matches a suffix of that sub-pattern.

1.3.2 Preprocessing Algorithm

The following steps outline how the `lps[]` array is constructed:

1. **Initialization**: We maintain a variable `len` to track the length of the longest prefix-suffix found for the previous index. We initialize both `lps[0]` and `len` to `0` since the longest proper prefix for a single character is always `0`.
2. **Matching Process**: We iterate through the pattern starting from index `1` to `m-1`. For each character at index `i`:
   - If the characters `pat[len]` and `pat[i]` match, we increment `len` by `1` and assign this new length to `lps[i]`. This indicates that we have found a longer matching prefix-suffix.
   - If there is no match and `len` is not `0`, we update `len` to `lps[len-1]`. This step allows us to check for the longest prefix-suffix using previously computed values, enabling the algorithm to skip unnecessary comparisons and potentially find a match.
3. **Handling Mismatches**: If a mismatch occurs and `len` is `0`, we simply set `lps[i]` to `0`, indicating that there is no proper prefix that matches a suffix for the sub-pattern ending at index `i`.

By effectively using the `lps[]` array, the KMP algorithm significantly reduces the number of character comparisons needed during the actual search phase, enhancing the overall efficiency of the pattern matching process.

Time Complexity: O(N+M) where N is the length of the text and M is the length of the pattern to be found.
Auxiliary Space: O(M)

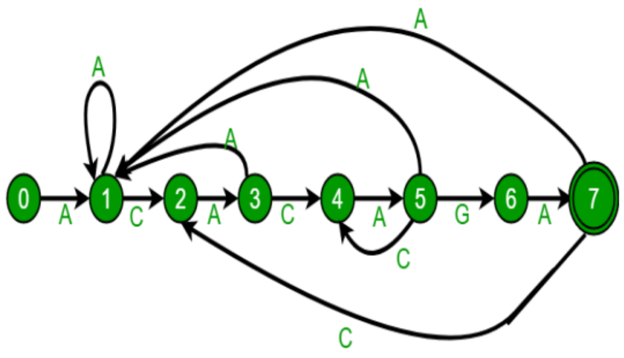
1.4 Finite Automata

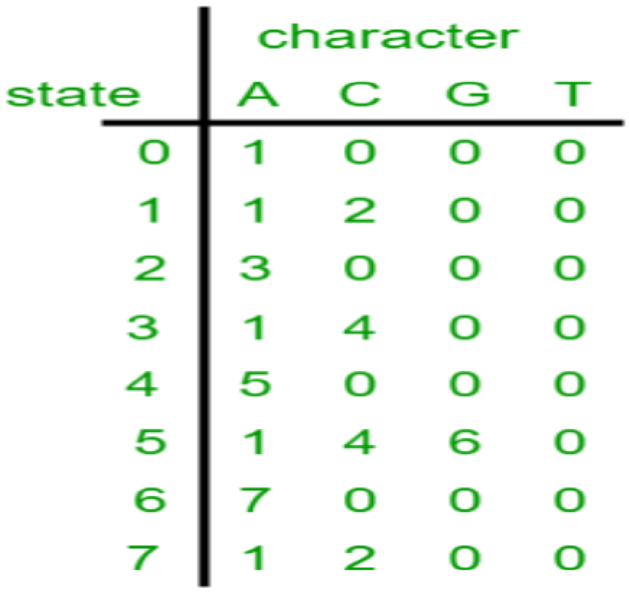
1.4.1 Finite Automata for Pattern Matching

The core idea of this approach is to utilize finite automata to efficiently scan a given text TTT in order to identify all occurrences of a specified pattern PPP. This method is designed to examine each character of the text exactly once, leading to a linear time complexity for matching, though the preprocessing time required to build the automaton can be substantial.

1.4.2 Finite Automaton Definition

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 6 matching but preprocessing time may be large.
3. It is defined by tuple M = Q, P, q, F, Where,
Q = Set of States in finite automata
P=Sets of input symbols
q = Initial state
F = Final State
= Transition function
Time Complexity = O(M^3| P|)
A finite automaton M is a 5-tuple (Q, q0,A,P,), where Q is a finite set of states, q0 Q is the start state, A subset of Q is a notable set of accepting states, is a finite input alphabet, is a function from Q x 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 (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 called the called the final-state function, from * to Q such that (w) is the state M ends up in after scanning the string w. Thus, M accepts a string w if and only if (w) A.





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 6: Finite automation for ACACAGA pattern

Time Complexity: O(m2)
Auxiliary Space: O(m)

2. Algorithms

2.1. Trie

2.1.1 Construction

Algorithm 1 Representation of Trie Node

```
1: Initial instructions
2: for $for - condition$ do
3:     Some instructions
4:     if $if - condition$ then
5:         Some other instructions
6:     end if
7: end for
8: while $while - condition$ do
9:     Some further instructions
10: end while
11: Final instructions
```

2.1.2 Insertion

Algorithm 2 Insertion in Trie (struct TrieNode *root, string key)

```
1:  Initial instructions
2:  for $for - condition$ do
3:      Some instructions
4:      if $if - condition$ then
5:          Some other instructions
6:      end if
7:  end for
8:  while $while - condition$ do
9:      Some further instructions
10: end while
11: Final instructions
```

2.1.3 Searching

Algorithm 3 Representation of Trie Node

```
1:  Initial instructions
2:  for $for - condition$ do
3:      Some instructions
4:      if $if - condition$ then
5:          Some other instructions
6:      end if
7:  end for
8:  while $while - condition$ do
9:      Some further instructions
10: end while
11: Final instructions
```

2.2. KMP

2.2.1 KMP Algorithm

Algorithm 4 KMP algorithm

```
1:  Initial instructions
2:  for $for - condition$ do
3:      Some instructions
4:      if $if - condition$ then
5:          Some other instructions
6:      end if
7:  end for
8:  while $while - condition$ do
9:      Some further instructions
10: end while
11: Final instructions
```

2.3. Finite Automata Algorithm

2.3.1 Fintie Automata Algorithm

Algorithm 5 Finite Automata Algorithm

```
1: Initial instructions
2: for $for - condition$ do
3:     Some instructions
4:     if $if - condition$ then
5:         Some other instructions
6:     end if
7: end for
8: while $while - condition$ do
9:     Some further instructions
10: end while
11: Final instructions
```

2.4. Suffix Tree

2.4.1 Construction

Algorithm 6 Construction of Suffix Tree

```
1: Initialize the root of the suffix tree
2: for each suffix of the input string do
3:     Insert the suffix into the tree
4: end for
5: Finalize the suffix tree structure
```

2.4.2 Insertion

Algorithm 7 Insertion into Suffix Tree

```
1: Given a suffix and the current node
2: for each character in the suffix do
3:     if the character path does not exist then
4:         Create a new node
5:     end if
6:     Move to the child node corresponding to the character
7: end for
8: Record the occurrence details at the leaf node
```

2.4.3 Searching

Algorithm 8 Searching in Suffix Tree

```
1: Initialize the current node as root
2: for each character in the pattern do
3:     if the character is not found in current node's children then
4:         Pattern not found
5:         return
6:     end if
7:     Move to the child node corresponding to the character
8: end for
9: Output the occurrences stored at the leaf node
```

3. Applications of Pattern Searching Algorithms

Pattern searching algorithms are integral across a wide range of fields, where they enable efficient information retrieval, data analysis, and other computational tasks. Below are some key applications:

1. **Text Searching and Information Retrieval**
   Pattern searching algorithms are fundamental in information retrieval systems, such as search engines. They allow users to quickly locate relevant documents, web pages, or content based on search queries or keywords, enhancing search accuracy and relevance.
2. **Data Mining and Text Analysis**
   In data mining and text analysis, pattern searching plays a crucial role in extracting valuable information from large datasets. Applications include sentiment analysis, topic modeling, and trend identification, where patterns in textual data reveal underlying insights and connections.
3. **Genomics and Bioinformatics**
   In genomics, pattern searching is essential for analyzing DNA and protein sequences. It aids in identifying specific motifs, detecting gene sequences, and pinpointing mutations or variations within biological data, which are critical for research and diagnostics.
4. **Text Compression**
   Pattern searching is employed in text compression techniques by identifying repeated patterns, which allows for more efficient data compression. This technique reduces file sizes, optimizing storage and transmission requirements.
5. **Network Security**
   Intrusion detection systems and network security applications use pattern searching algorithms to detect known attack patterns or suspicious activities in log files and network traffic. This helps identify potential security threats and ensure network integrity.
6. **Image Processing**
   In image processing, pattern matching techniques are used for object recognition, image retrieval, and locating specific features within images. These applications are key in fields like computer vision, automated surveillance, and medical imaging.
7. **Spell Checkers and Autocorrection**
   Pattern searching algorithms power spell-check and autocorrection systems by identifying misspelled words and suggesting appropriate corrections based on similar patterns. This application improves text accuracy and user convenience.
8. **File and Text Editing**
   Text editors and search tools leverage pattern searching algorithms to enable users to locate and edit specific text within documents or code efficiently. This functionality is indispensable for software development, document editing, and content management.
9. **Machine Translation and Natural Language Processing (NLP)**
   In NLP, pattern searching algorithms support tasks like machine translation, text summarization, and question answering by identifying relevant linguistic patterns. These applications enhance the capabilities of language models and improve translation accuracy.
10. **Plagiarism Detection**
    Pattern searching algorithms are widely used in plagiarism detection systems to compare documents and identify copied content. These systems are crucial in academic, publishing, and content creation domains to maintain originality and intellectual integrity.

4. Pseudo-Codes

4.1 Trie

The following pseudo-code outlines the steps for building and searching a Trie data structure to find occurrences of a given pattern in a text file.

```
Define TrieNode:
    children: array of TrieNode pointers
    occurrenceList: list to store occurrences (line number, index)
    occurrenceCount: integer to track number of occurrences

Define Trie:
    root: pointer to TrieNode

Function createTrieNode():
    Initialize a new TrieNode with empty children array and set occurrenceCount to 0
    Return new TrieNode

Function initializeTrie():
    Initialize Trie with a root TrieNode
    Return Trie

Function insertWord(trie, word, lineNum, startIndex):
    currentNode <- trie.root
    For each character in word:
        If currentNode does not have a child for character:
            Create a new TrieNode as child of currentNode
        Move to the child node for character
        If occurrenceCount in currentNode < MAX_OCCURRENCES:
            Add (lineNum + 1, startIndex + 1) to currentNode's occurrenceList
            Increment occurrenceCount of currentNode

Function buildTrieFromLines(trie, lines, lineCount):
    For each line in lines:
        For each position in line:
            Call insertWord(trie, substring from position, line number, position)

Function searchPatternInTrie(node, pattern, index):
    If node is NULL:
        Return
    If pattern[index] is end of string:
        If node has occurrences:
            Print each occurrence's line number and start index
            Print total frequency of pattern
        Return
    Recursive call searchPatternInTrie with next character in pattern

Function freeTrie(node):
    If node is NULL:
        Return
    For each child in node:
        Recursively call freeTrie on child
    Free node

Function loadLinesFromFile(filename):
    Open file for reading
    Initialize list of lines
    For each line in file:
        Trim newline, add line to list
    Return list of lines and line count

Main Function:
    Load lines from file
    Initialize Trie
    Build Trie with loaded lines
    Input search pattern from user
    Record start time
    Call searchPatternInTrie with pattern
    Record end time, calculate and print elapsed time
    Free memory for Trie and lines
```

4.2 Suffix

The following pseudo-code outlines the steps for building and searching a Suffix Tree data structure to find occurrences of a given pattern in a text file.

```
Define SuffixTreeNode:
    children: array of SuffixTreeNode pointers
    occurrenceList: list to store occurrences (line number, index)
    occurrenceCount: integer to track number of occurrences

Define SuffixTree:
    rootNode: pointer to SuffixTreeNode

Function createSuffixTreeNode():
    Initialize a new SuffixTreeNode with empty children array
    Set occurrenceCount to 0
    Return new SuffixTreeNode

Function initializeSuffixTree():
    Initialize SuffixTree with a rootNode
    Return SuffixTree

Function addSuffixToTree(suffixTree, suffix, lineNum, startIndex):
    currentNode <- suffixTree.rootNode
    For each character in suffix starting at startIndex:
        If currentNode does not have a child for character:
            Create a new SuffixTreeNode as child of currentNode
        Move to the child node for character
        If occurrenceCount in currentNode < MAX_OCCURRENCES:
            Add (lineNum + 1, startIndex) to currentNode's occurrenceList
            Increment occurrenceCount of currentNode

Function buildSuffixTreeFromLines(suffixTree, lines, lineCount):
    For each line in lines:
        For each position in line:
            Call addSuffixToTree(suffixTree, line, line number, position)

Function findPatternInTree(node, pattern, index, lines):
    If node is NULL:
        Return
    If pattern[index] is end of string:
        If node has occurrences:
            Print each occurrence's line number, start index, and word
            Print total frequency of pattern
        Set flag to indicate pattern found
        Return
    Recursive call findPatternInTree with next character in pattern
```

```
Function releaseSuffixTree(node):
    If node is NULL:
        Return
    For each child in node:
        Recursively call releaseSuffixTree on child
    Free node

Function loadLinesFromFile(filename):
    Open file for reading
    Initialize list of lines
    For each line in file:
        Trim newline, add line to list
    Return list of lines and line count

Main Function:
    Load lines from file
    Initialize SuffixTree
    Build SuffixTree with loaded lines
    Input search pattern from user
    Record start time
    Call findPatternInTree with pattern
    Record end time, calculate and print elapsed time
    Free memory for SuffixTree and lines
```

4.3 KMP

The following pseudo-code outlines the steps for building and searching using the Knuth-Morris-Pratt (KMP) algorithm to find occurrences of a given pattern in a text file.

```
Initialize:
    v1: dynamic array to store positions of line breaks
    v1_size: integer to store the number of lines in the file
    count_kmp: integer to track the total pattern occurrences

Function computeLPSArray(pat, M, lps):
    Initialize len to 0 (length of the previous longest prefix suffix)
    Set lps[0] to 0
    For i from 1 to M:
        If pat[i] matches pat[len]:
            Increment len, set lps[i] to len, and increment i
        Else:
            If len is not zero, update len to lps[len - 1]
            Else, set lps[i] to 0 and increment i

Function KMPSearch(pat, txt):
    M <- length of pat
    N <- length of txt
    Allocate memory for lps array of size M
    Call computeLPSArray(pat, M, lps)
    Initialize i and j to 0 (indices for txt and pat)

    While (N - i) >= (M - j):
        If pat[j] matches txt[i]:
            Increment j and i
        If j equals M:
            Calculate line number and position using v1
            Identify start and end of the word containing the pattern
            Extract the matched word
            Print line number, position, and word
            Increment count_kmp
            Set j to lps[j - 1] for next possible match
        Else if mismatch after j matches:
            If j is not zero, update j to lps[j - 1]
            Else, increment i

    Free memory for lps array

Function loadLinesFromFile(filename):
    Open file for reading
    Initialize empty string s1 for the complete text
    Initialize dynamic array v1 to store line break positions
    For each line in the file:
        Append line to s1 with a space separator
        Store cumulative line length in v1
    Return s1 and v1

Main Function:
    Load file contents into s1 and line positions into v1
    Input pattern from user
    Start time measurement
    Call KMPSearch with pattern and text
    End time measurement
    Print count_kmp and execution time
    Free allocated memory for s1 and v1
```

4.4 Finite Automata

The following pseudo-code outlines the steps for building and searching a finite automaton to find occurrences of a given pattern (prefix) in a text file.

```
Define global variables:
    v1: dynamic array to store line-ending positions
    count_fa: integer to track pattern occurrences
    v1_size: integer to store the number of lines in the file

Function getNextState(pat, M, state, x):
    If state < M and pat[state] matches x:
        Return state + 1
    For ns from state down to 1:
        If pat[ns - 1] matches x:
            For each character up to ns - 1:
                If mismatch with pat:
                    Break
            If match found, return ns
    Return 0

Function computeTF(pat, M, TF):
    For each state from 0 to M:
        For each character x:
            TF[state][x] = getNextState(pat, M, state, x)

Function search(pat, txt):
    M <- length of pat
    N <- length of txt
    Initialize TF as a transition table of size (M+1) x NO\_OF\_CHARS
    Call computeTF(pat, M, TF)
    Initialize i and state to 0

    For i from 0 to N - 1:
        Update state to TF[state][txt[i]]
        If state equals M (pattern found):
            Determine start and end of the word containing pattern
            Extract matched word from txt
            Calculate line number and position within line using v1
            Print line number, position, and word
            Increment count_fa

Main Function:
    Open the file and read contents line-by-line
    Append each line to s1 with space as separator
    Store cumulative length of each line in v1
    Input pattern to search
    Record start time
    Call search with pattern and text
    Record end time
    Print count_fa and execution time in milliseconds
    Free allocated memory for s1 and v1
```

This pseudo-code provides a concise overview of the main functions and logic used for the finite automaton-based pattern search, formatted for the 'verbatim' environment in LaTeX. Adjust the layout as needed based on your LaTeX document's formatting requirements.

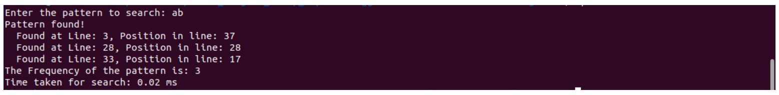
5. Complexity Analysis

In this section, we present the time and space complexity analysis of various pattern-searching algorithms used in our project. Let:

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

5.1. Trie

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

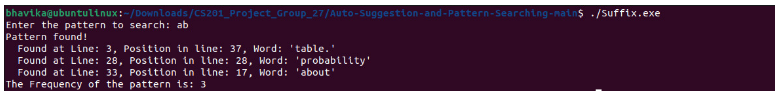


```
Enter the pattern to search: ab
Pattern found!
  Found at Line: 3, Position in line: 37
  Found at Line: 28, Position in line: 28
  Found at Line: 33, Position in line: 17
The Frequency of the pattern is: 3
Time taken for search: 0.02 ms
```

Figure 7: Output Of Trie

5.2. Suffix Tree (Naive Approach)

- **Space Complexity:** In the worst case (all unique characters), the suffix tree could have $O(n^2)$ nodes. Average case complexity is $O(n \cdot l)$.
- **Time Complexity:** Building the suffix tree takes $O(n \cdot l^2)$; pattern searching takes $O(m)$.

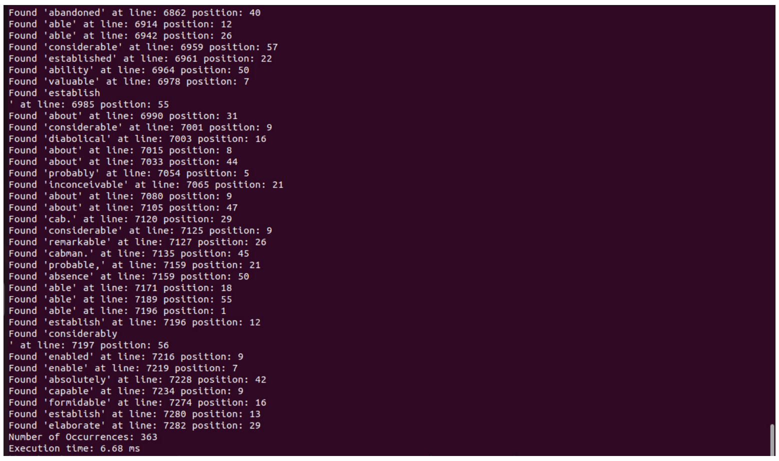


```
bhavika@ubuntulinux:~/Downloads/CS201_Project_Group_27/Auto-Suggestion-and-Pattern-Searching-main$ ./Suffix.exe
Enter the pattern to search: ab
Pattern found!
  Found at Line: 3, Position in line: 37, Word: 'table.'
  Found at Line: 28, Position in line: 28, Word: 'probability'
  Found at Line: 33, Position in line: 17, Word: 'about'
The Frequency of the pattern is: 3
```

Figure 8: Output Of Suffix

5.3. Knuth-Morris-Pratt (KMP) Algorithm

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

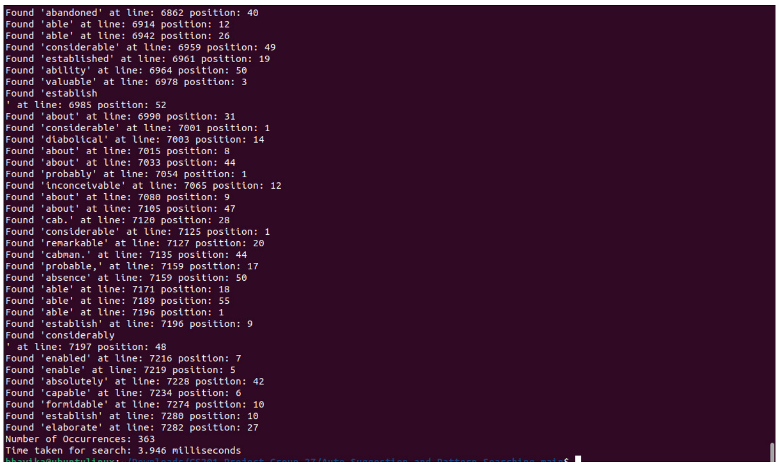


```
Found 'abandoned' at line: 6862 position: 40
Found 'able' at line: 6914 position: 12
Found 'able' at line: 6942 position: 26
Found 'considerable' at line: 6959 position: 57
Found 'established' at line: 6961 position: 22
Found 'ability' at line: 6964 position: 50
Found 'valuable' at line: 6978 position: 7
Found 'establish
' at line: 6985 position: 55
Found 'about' at line: 6990 position: 31
Found 'considerable' at line: 7001 position: 9
Found 'diabolical' at line: 7003 position: 16
Found 'about' at line: 7015 position: 8
Found 'about' at line: 7033 position: 44
Found 'probably' at line: 7054 position: 5
Found 'inconceivable' at line: 7065 position: 21
Found 'about' at line: 7080 position: 9
Found 'about' at line: 7105 position: 47
Found 'cab.' at line: 7120 position: 29
Found 'considerable' at line: 7125 position: 9
Found 'remarkable' at line: 7127 position: 26
Found 'cabman.' at line: 7135 position: 45
Found 'probable,' at line: 7159 position: 21
Found 'absence' at line: 7159 position: 50
Found 'able' at line: 7171 position: 18
Found 'able' at line: 7189 position: 55
Found 'able' at line: 7196 position: 1
Found 'establish' at line: 7196 position: 12
Found 'considerably
' at line: 7197 position: 56
Found 'enabled' at line: 7216 position: 9
Found 'enable' at line: 7219 position: 7
Found 'absolutely' at line: 7228 position: 42
Found 'capable' at line: 7234 position: 9
Found 'formidable' at line: 7274 position: 16
Found 'establish' at line: 7280 position: 13
Found 'elaborate' at line: 7282 position: 29
Number of Occurrences: 363
Execution time: 6.68 ms
```

Figure 9: Output Of KMP

5.4. Finite Automata

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



```
Found 'abandoned' at line: 6862 position: 40
Found 'able' at line: 6914 position: 12
Found 'able' at line: 6942 position: 26
Found 'considerable' at line: 6959 position: 49
Found 'established' at line: 6961 position: 19
Found 'ability' at line: 6964 position: 50
Found 'valuable' at line: 6978 position: 3
Found 'establish
' at line: 6985 position: 52
Found 'about' at line: 6990 position: 31
Found 'considerable' at line: 7001 position: 1
Found 'diabolical' at line: 7003 position: 14
Found 'about' at line: 7015 position: 8
Found 'about' at line: 7033 position: 44
Found 'probably' at line: 7054 position: 1
Found 'inconceivable' at line: 7065 position: 12
Found 'about' at line: 7080 position: 9
Found 'about' at line: 7105 position: 47
Found 'cab.' at line: 7120 position: 28
Found 'considerable' at line: 7125 position: 1
Found 'remarkable' at line: 7127 position: 20
Found 'cabman.' at line: 7135 position: 44
Found 'probable,' at line: 7159 position: 17
Found 'absence' at line: 7159 position: 50
Found 'able' at line: 7171 position: 18
Found 'able' at line: 7189 position: 55
Found 'able' at line: 7196 position: 1
Found 'establish' at line: 7196 position: 9
Found 'considerably
' at line: 7197 position: 48
Found 'enabled' at line: 7216 position: 7
Found 'enable' at line: 7219 position: 5
Found 'absolutely' at line: 7228 position: 42
Found 'capable' at line: 7234 position: 6
Found 'formidable' at line: 7274 position: 10
Found 'establish' at line: 7280 position: 10
Found 'elaborate' at line: 7282 position: 27
Number of Occurrences: 363
Time taken for search: 3.946 milliseconds
```

Figure 10: Output Of Finite Automata

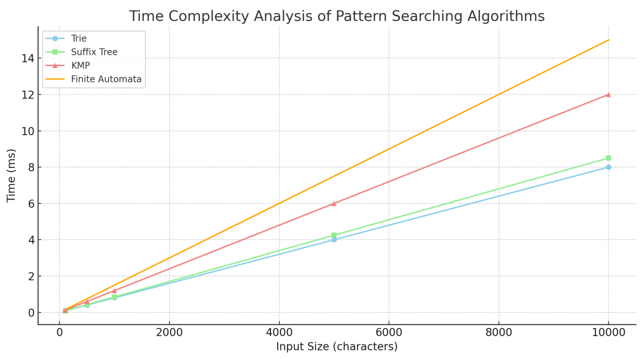


Figure 11: Comparision between Time Complexity Analysis of All the Pattern Searching Algorithms

6. Conclusions

In conclusion, this project on string pattern searching has enhanced our understanding of various algorithms, such as Knuth-Morris-Pratt and Finite Automata, as well as their applications in text processing. We have also explored advanced data structures like suffix trees and Tries, which significantly optimize search efficiency for large datasets. Additionally, the integration of an auto-suggestion feature improves user experience by providing real-time word predictions, showcasing the practical benefits of combining pattern searching with user-friendly interfaces.

7. Bibliography and citations

Suffix Tree on Wikipedia.
Pattern Searching Using Trie Suffixes
KMP Algorithm for Pattern Searching
Finite Automata Algorithm for Pattern Searching

8. References

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