Pattern Searching and Implementing Algorithms

November 5, 2023

Aayan Soni (2022CSB1061) ,
Aditya Garg (2022CSB1062) ,
Ananya Sethi (2022CSB1066)

Instructor:
Dr. Anil Shukla

Teaching Assistant:
Soumya Sarkar

Summary: We did pattern searching in a text file using data structures like Trie and Suffix Tree. We also implemented KMP and the Finite Automata algorithm for pattern searching. We compared the time complexity of searching the pattern using all these different methods. In the output we gave the line number and the number of occurrences of the pattern in the text.

1. Trie and Suffix Tree

Trie
Trie data structure is defined as a Tree-based data structure that is used for storing some collection of strings and performing efficient search operations on them. Tries are particularly useful for tasks involving string matching, text autocompletion, and searching, such as dictionary lookups and IP routing in networking. They can be space-efficient because common prefixes among strings are shared in the tree structure. This makes them particularly useful when you have a large number of strings with significant overlap in their prefixes.

Properties: 1.There is one root node in each Trie.
2. Each node of a Trie represents a string and each edge represents a character.
3. Every node consists of hashmaps or an array of pointers, with each index representing a character and a flag to indicate if any string ends at the current node.
4. Trie data structure can contain any number of characters including alphabets, numbers, and special characters. But for this article, we will discuss strings with characters a-z. Therefore, only 26 pointers need 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.

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", again there is an array of size 26, for storing the 2nd character. The second character is "n", So from "a", we will move to "n" and mark "n" in the 2nd array as used. After "n", the 3rd character is "d", So mark the position "d" as used in the respective array.

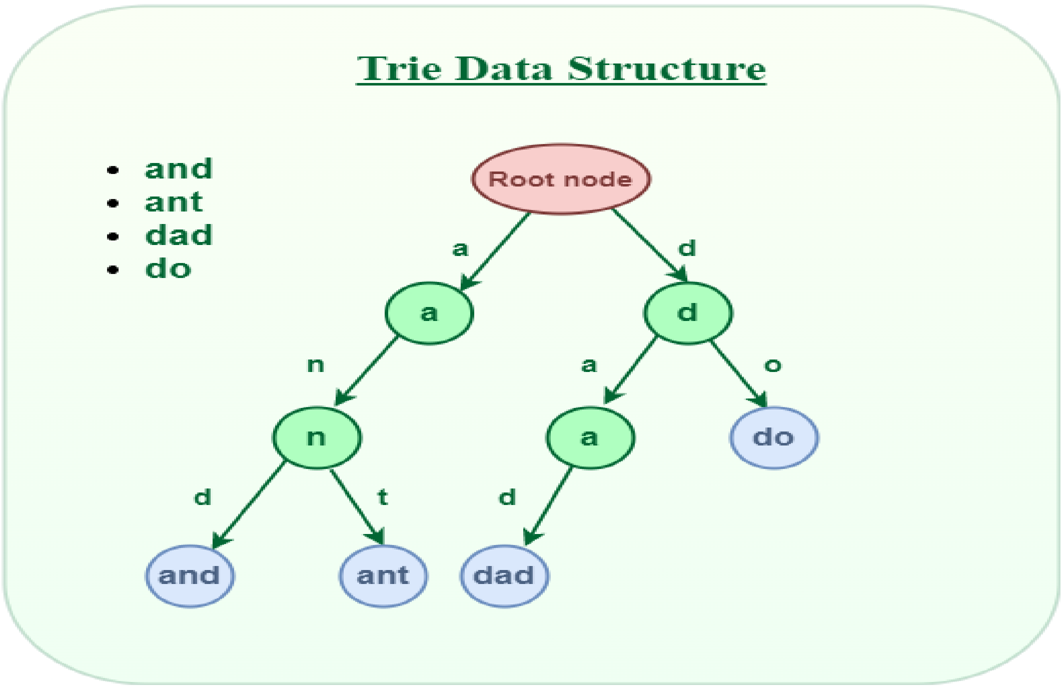


Figure 1: Simple Trie Data Structure

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 Trie. For the second character ‘n‘ we can observe that the position of ‘n’ in the ‘a’ node has already been filled. So, no need to fill it again, just move to node ‘n’ in Trie. For the last character ‘t‘ of the word, The position for ‘t‘ in the ‘n‘ node is not filled. So, filled the position of ‘t‘ in ‘n‘ node and move to ‘t‘ node.

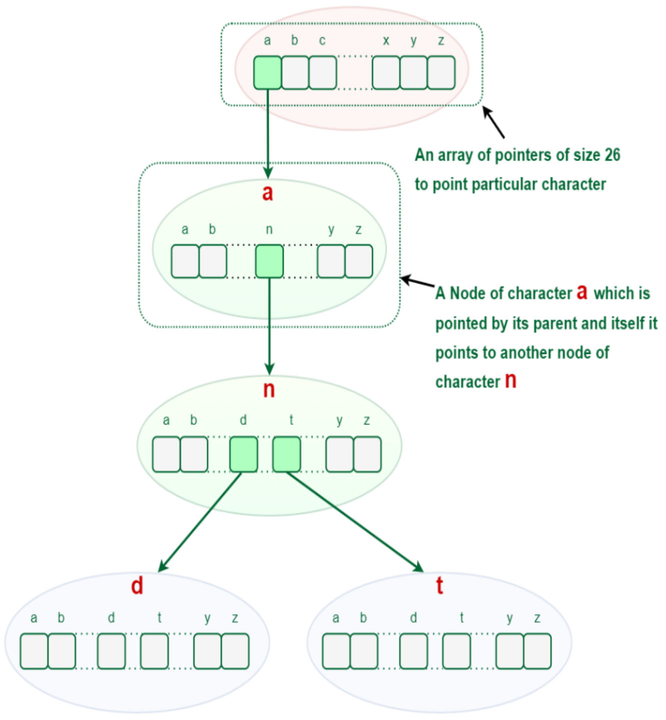

a b c x y z
An array of pointers of size 26
to point particular character
a
a b n y z
A Node of character a which is
pointed by its parent and itself it
points to another node of
character n
n
a b d t y z
d
a b d t y z
t
a b d t y z

Figure 2: Representation of how words are stored in trie

1.1. 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
  Construct trie Node,
    struct Trie Node
        TrieNode *children[ALPHABETSIZE];
        int wordCount = 0;
```

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 curr to the newly created node.
5. Finally, increment the wordCount of the last currentNode, 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
  $node->isEndofWord = true$
```

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 curr node does not point to the current character of the word then 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)

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

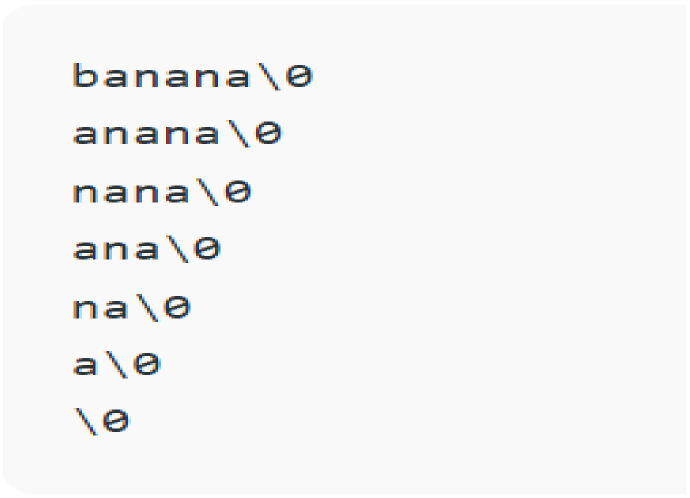
Time Complexity of the Algorithm The time taken by this algorithm is $O(n^2)$ where 'n' is the length of the text. This time is essentially taken to build the trie. Note that this is one time activity and subsequent searches of another pattern in this text would take $O(m)$ time where m is the length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

Suffix Tree A Suffix Tree for a given text is 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. 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

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

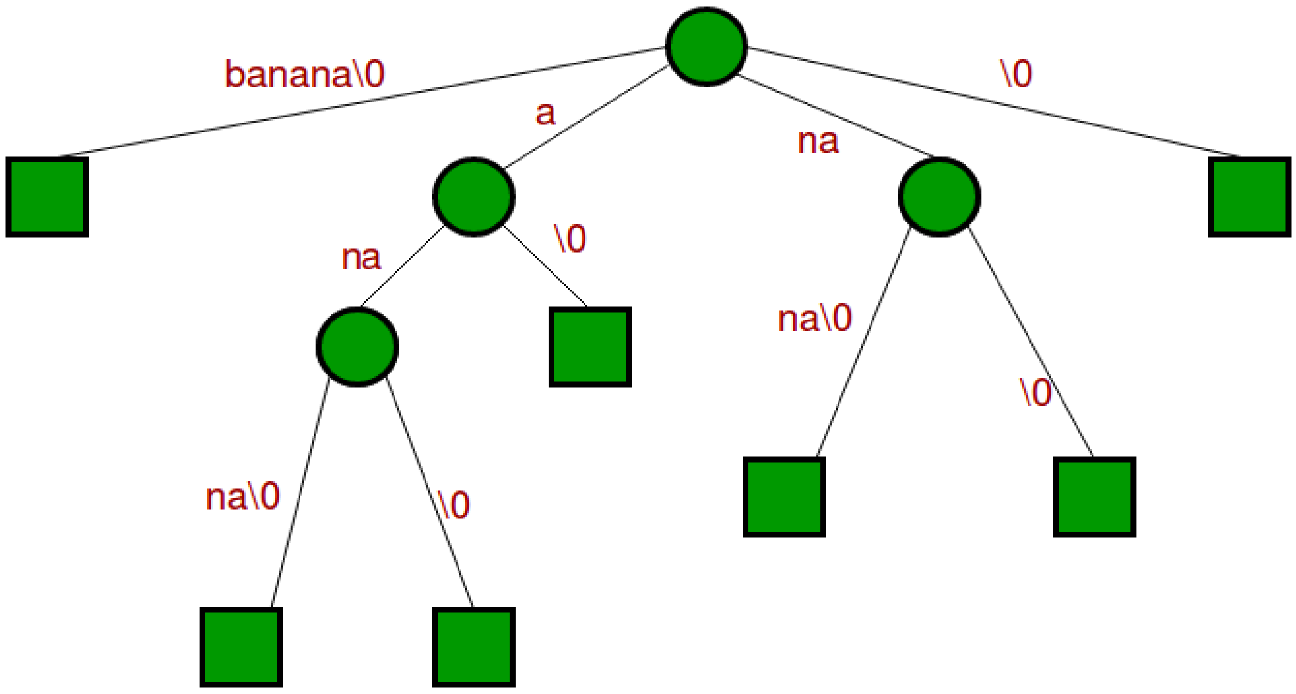


Figure 4: Creation of Suffix Tree

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.

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.

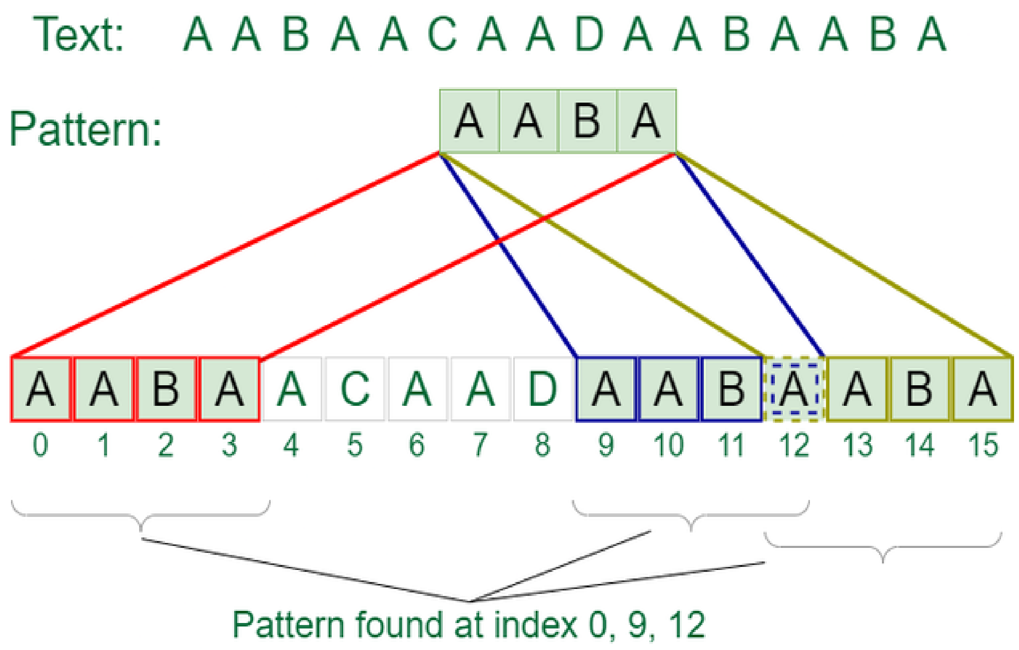

Text: A A B A A C A A D A A B A A B A
Pattern: A A B A
A A B A A C A A D A A B A A B A
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
Pattern found at index 0, 9, 12

Figure 5: 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
1. 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
2. We initialize lps[0] and len as 0.
3. If pat[len] and pat[i] match, we increment len by 1 and assign the incremented value to lps[i].
4. 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.

---

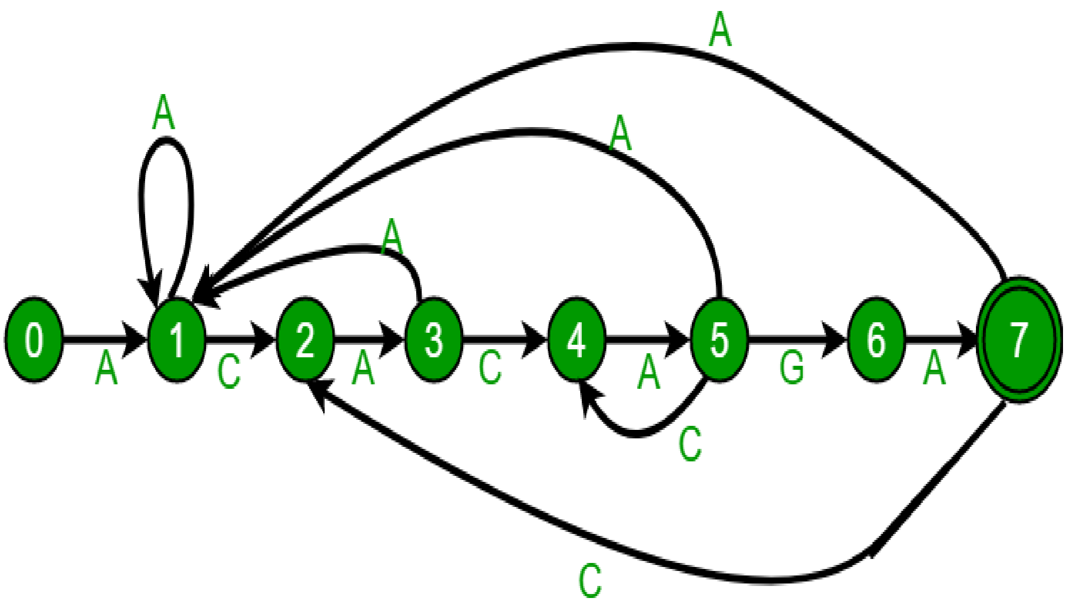
3. Finite Automata algorithm

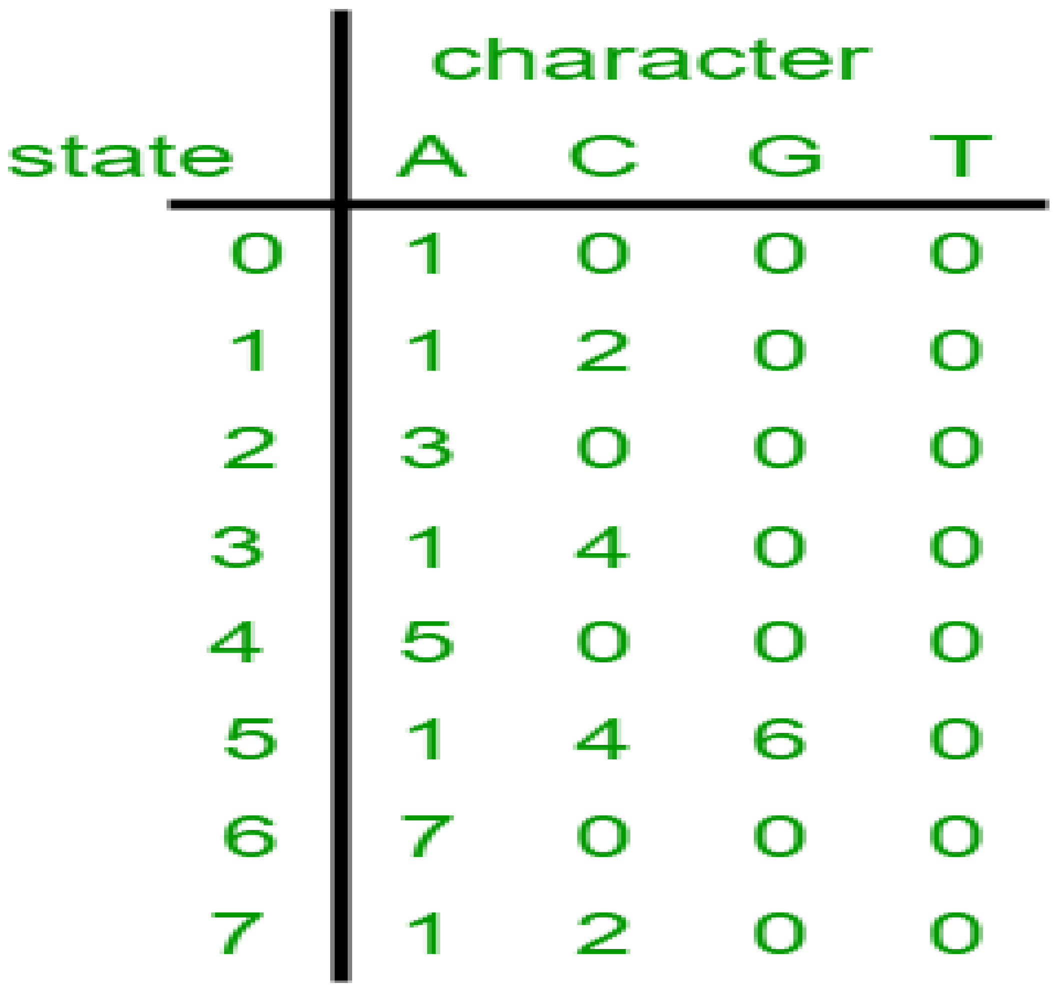
1. Idea of this approach is to build finite automata to scan text T for finding all occurrences of pattern P. 2. This approach examines each character of text exactly once to find the pattern. Thus it takes linear time for matching but preprocessing time may be large. 3. It is defined by tuple M = Q, $\sum$, 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 = $O(M^3|\sum|)$
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 <- $\delta$(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 6: Finite automation for ACACAGA pattern

Application

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.

4. Conclusions

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, which enable us to optimize string pattern searching and provide faster solutions for large-scale datasets. The use of these advanced data structures has opened up new possibilities and opportunities for handling complex pattern search tasks.

5. Bibliography and citations

Suffix tree
Suffix tree-2
Suffix tree-3
Tries
Kmp
Finite automata algorithm

6. 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