Comparison of String Matching in Plain Text Using Different Search Algorithms

A

Project Report

submitted in partial fulfillment of the

requirements for the award of the degree of

BACHELOR OF TECHNOLOGY

in

COMPUTER SCIENCE & ENGINEERING

by

Name	Roll No.	SAP ID
Vaswati Gogoi	R2142221419	500110490
Sagnik Roy	R2142221403	500109927

under the guidance of

Dr. Surbhi Saraswat

School of Computer Science

University of Petroleum & Energy Studies

Bidholi, Via Prem Nagar, Dehradun, Uttarakhand

December – 2024

CANDIDATE'S DECLARATION

I/We hereby certify that the project work entitled **"Comparison of String Matching in Plain Text Using Different Search Algorithms "**in partial fulfilment of the requirements for the award of the Degree of BACHELOR OF TECHNOLOGY in COMPUTER SCIENCE AND ENGINEERING with specialization in BIG DATA and submitted to the Department of Systemics, School of Computer Science, University of Petroleum & Energy Studies, Dehradun, is an authentic record of my/ our work carried out during a period from **August**, **2024** to **December**, **2024** under the supervision of **Dr. Surbhi Saraswat.**

The matter presented in this project has not been submitted by me/ us for the award of any other degree of this or any other University.

Vaswati Gogoi
Roll No.- R2142221419
Sagnik Roy
Roll No.- R2142221403

This is to certify that the above statement made by the candidate is correct to the best of my knowledge.

Date: 16th December, 2024

Dr. Surbhi Saraswat
(Project Guide)

<u>ACKNOWLEDGEMENT</u>

We wish to express our deep gratitude to our guide **Name**, for all advice, encouragement, and constant support he/she has given us throughout our project work. This work would not have been possible without his support and valuable suggestions.

We sincerely thanks to our respected **Virender Kadyan, Head Department of <u>Data Science</u>,** for his great support in doing our project in **<u>Comparison of String Matching in Plain Text Using Different Search Algorithms</u>.**

We are also grateful to Dean SOCS UPES for giving us the necessary facilities to carry out our project work successfully. We also thanks to our Course Coordinator, Dr Prabhat Ranjan Singh and our Activity Coordinator Dr Sachin Chaudhary for providing timely support and information during the completion of this project.

We would like to thank all our **friends** for their help and constructive criticism during our project work. Finally, we have no words to express our sincere gratitude to our **parents** who have shown us this world and for every support they have given us.

Name	**Vaswati Gogoi**	**Sagnik Roy**
Roll No.	**R2142221419**	**R2142221403**
SAP ID	**500110490**	**500109927**

ABSTRACT

This study comprehensively examines and compares the effectiveness of various search algorithms in detecting string matches within plain text documents. The research delves into multiple dimensions, presenting a systematic evaluation of algorithm performance and applicability:

- **Algorithmic Comparison**: A detailed analysis of algorithms such as KMP, Boyer-Moore, Rabin-Karp, and Aho-Corasick, focusing on their strengths and limitations.
- **Performance Metrics**: Evaluation based on critical metrics, including speed, accuracy, memory usage, and computational efficiency across diverse datasets.
- **Scalability**: Exploration of how each algorithm performs under varying text sizes and pattern complexities to determine adaptability in real-world scenarios.
- **Benchmarking and Testing**: Rigorous experimental setups to simulate various operational conditions and assess practical effectiveness.
- **Application Relevance**: Insights into the role of these algorithms in domains like plagiarism detection, search optimization, text analysis, and information retrieval systems.
- **Tool Enhancement**: Recommendations for integrating the most reliable algorithms into advanced string-matching tools to optimize detection capabilities.

This project aims to bridge gaps in algorithmic understanding while offering actionable outcomes to improve tools' efficiency and reliability for broader adoption in critical applications.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION

1.1 History

- **Early String-Matching Algorithms**: The earliest string matching techniques like **Naive Search** simply involve scanning the text sequentially for matches with the pattern. The **Naive Search Algorithm** has a time complexity of O(n×m)O(n \times m)O(n×m), where nnn is the length of the text and mmm is the length of the pattern.
- **Knuth-Morris-Pratt (KMP)**: Introduced by Donald Knuth, Vaughan Pratt, and Robert Morris in 1977, KMP reduces unnecessary re-checking by preprocessing the pattern and using that information during the search process. Its time complexity is O(n+m)O(n + m)O(n+m), where mmm is the length of the pattern, and nnn is the length of the text.
- **Boyer-Moore**: This algorithm, introduced in 1977 by Robert S. Boyer and J Strother Moore, uses heuristic methods like **bad character rule** and **good suffix rule** to skip unnecessary text sections, making it efficient for large datasets. It has a best-case time complexity of O(n/m)O(n/m)O(n/m) and worst-case time complexity of O(n×m)O(n \times m)O(n×m).
- **Rabin-Karp**: This algorithm, developed by Richard Rabin and Michael Karp in 1987, relies on hashing techniques for pattern matching. It is particularly effective in multi-pattern matching scenarios. The time complexity for a single pattern is O(n+m)O(n + m)O(n+m), but for multiple patterns, the average complexity is close to O(n+m+z)O(n + m + z)O(n+m+z), where zzz is the number of matches found.
- **Aho-Corasick**: This algorithm, introduced by Alfred Aho and Margaret Corasick in 1975, is specifically designed for **multi-pattern matching**. It builds a **finite automaton** based on the patterns and processes the text in linear time. Its time complexity is O(n+z)O(n + z)O(n+z), where zzz is the number of matches found.

1.2 Requirement Analysis

- **Need for Comparison**: String matching is fundamental in applications such as **text searching**, **plagiarism detection**, **bioinformatics**, and **data retrieval systems**. Understanding the best algorithm under different conditions—such as pattern length, text length, and complexity—is essential.
    - The aim is to determine which algorithm is optimal based on speed, accuracy, and resource utilization.
- **Current Tools**: Existing tools, like **grep** and **findstr**, are efficient for small-scale tasks, but they lack a comparative framework for handling large text data efficiently across varying algorithmic conditions.

1.3 Main Objective

- To **compare** the effectiveness of search algorithms in **detecting string matches in plain text** in terms of **speed**, **accuracy**, and **computational efficiency**, providing a reliable solution for real-world text matching problems.

1.4 Sub-Objectives

- **Evaluation Criteria**: The algorithms will be tested on different **datasets** to assess:
    1. **Speed**: Time taken to perform matching for various sizes of text and patterns.
    2. **Accuracy**: Correctness of the match, considering false positives and negatives.
    3. **Memory/CPU Usage**: Measuring the resources required for each algorithm.
- **Benchmarking Performance**: Create different conditions for testing to highlight the performance of each algorithm.

1.5 PERT Chart Legend

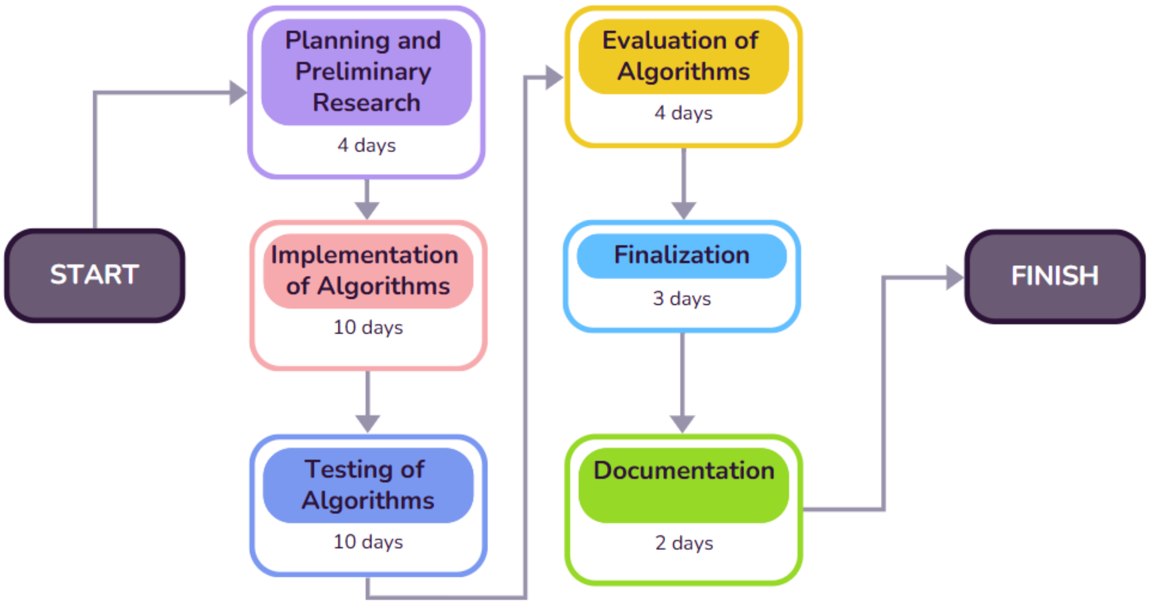


Fig. 1.1 Pert Chart

2. SYSTEM ANALYSIS

2.1. Existing System

In Java, several **built-in methods** and **libraries** exist to perform string matching and search tasks, both for simple pattern matching and more complex cases. Here is an overview of the **existing system** in the context of Java, including built-in string functions and libraries for string matching.

Built-in String-Matching Methods

- **String.indexOf()**:
  Java provides an in-built function `indexOf()` for string matching, where it returns the index of the first occurrence of a specified substring in a given string.
  - **Advantages**:
    - **Simple to implement**: Easy-to-use method for basic substring searching.
    - **Fast for smaller tasks**: Performs well on small strings.
  - **Limitations**:
    - **Inefficient for large data**: It performs a linear search (`O(n)` complexity) which can be slow for large texts.
    - **Limited functionality**: Cannot be used for more complex pattern searches, such as matching multiple patterns or using regex.

String.matches():
This method allows you to search for substrings that match a regular expression.

- **Advantages**:
  - **Regex support**: Can match more complex patterns using regular expressions.
  - **Simple syntax**: Useful for one-liner implementations.
- **Limitations**:
  - **Slower performance**: The internal regex matching can be slower than direct substring searches, particularly on large datasets.

Summary of Existing Tools:

- These existing systems and tools are useful for small-scale use cases with basic string-matching requirements. However, they face significant challenges in larger and more complex scenarios, particularly when it comes to efficiently matching multiple patterns or scaling for enormous datasets.

2.2. Motivations

There are several practical challenges that motivate the comparison and improvement of string-matching algorithms:

- **Efficiency Concerns**: Existing algorithms and tools face problems handling large datasets. For instance, handling multi-gigabyte files (e.g., books, entire datasets) or databases of strings can quickly strain simple algorithms, especially when complex patterns and search conditions are involved.
- **Large-scale Matching**: Patterns may need to be matched against hundreds or thousands of datasets or documents. The system should also handle multiple, simultaneously occurring patterns.
- **Need for Multi-pattern Search**: Searching for **multiple patterns** (like finding different types of malicious strings in network packets or looking up keywords in large corpora) often requires more sophisticated algorithms. This capability is not built into basic search tools.
- **Computational Complexity**: Algorithmic solutions need to be efficient in terms of both time and space. In **resource-constrained environments** (e.g., mobile devices, embedded systems), the matching system should work without consuming too many system resources (RAM and CPU).

Thus, there is a clear need for this study to evaluate and compare the time complexity, accuracy, and performance of algorithms across different datasets and test conditions.

2.3. Proposed System

To address the challenges and inefficiencies found in existing tools, the **proposed system** focuses on a **comparative analysis** of several string-matching algorithms:

- **Selection of Algorithms**: We will implement and test several common search algorithms:
  - **Knuth-Morris-Pratt (KMP)**: This algorithm efficiently handles prefix overlap by preprocessing the pattern, allowing for a faster matching process.
  - **Boyer-Moore**: Known for its efficiency on average, especially for larger text, Boyer-Moore applies heuristics to skip unnecessary character comparisons.
  - **Rabin-Karp**: The Rabin-Karp algorithm uses hash functions for fast matching, making it especially effective when searching for multiple patterns simultaneously.
  - **Aho-Corasick**: This algorithm supports the search for multiple patterns at once, building a finite automaton for all possible matching patterns.
- **Real-World Data**: The algorithms will be evaluated across different datasets including:
  - **Text files (books, web pages)**: For typical applications such as text search.
  - **DNA sequences**: For bioinformatics use cases where string matching is critical.

  - **Large plain-text corpora**: Simulating larger datasets found in social media or corporate systems to evaluate scalability.

- **Performance Metrics**: The primary objective of the proposed system is to evaluate these algorithms based on:

1. **Run-time performance** (speed of matching): How long the algorithm takes to run on datasets of various sizes.
2. **Memory and CPU usage**: How much system memory (RAM) and processing power (CPU) is required for each algorithm.
3. **Accuracy of matching**: The correctness of the detected matches, focusing on reducing false positives/negatives.
4. **Scalability**: How well the algorithms perform as the size of the dataset increases or the complexity of the pattern's changes.
5. **Handling Multiple Patterns**: Evaluating how well multi-pattern search is implemented, especially for Rabin-Karp and Aho-Corasick algorithms.

2.4. Modules

To organize the development of the system effectively, the following **modules** will be implemented:

2.4.1. Text Preprocessing:

Before testing the algorithms, all datasets will be pre-processed to ensure consistency. This preprocessing module will handle:

- **Text normalization**: Removing non-alphabetic characters, handling case insensitivity, and ensuring uniform encoding (e.g., UTF-8).
- **Tokenization**: Breaking text into manageable units like words or lines.
- **Data cleaning**: Removal of unwanted characters or whitespaces that may impact the matching process.

2.4.2. Algorithm Implementation:

Each search algorithm will be implemented according to the standard approach:

- **KMP Algorithm**: Efficiently match a pattern by preprocessing and skipping redundant checks.
- **Boyer-Moore**: Optimize the search by applying the **bad character** and **good suffix heuristics** to skip sections of the text.
- **Rabin-Karp**: Use **hashing** to compare substrings in the text with the pattern(s), perfect for finding multiple matches at once.
- **Aho-Corasick**: Build a **trie-based automaton** to quickly search for multiple patterns within the text simultaneously.

Each of these algorithms will be tailored for consistent input formatting so they can be directly compared.

2.4.3. Performance Evaluation:

Once the algorithms are implemented, **benchmarking tests** will be conducted to measure:

- **Execution time** for different sizes of texts and patterns.
- **Resource consumption** (e.g., memory usage and CPU usage during the run).
- **Matching accuracy**: Analyze the **number of matches**, checking for false positives and negatives.
- **Scalability**: Test how the algorithms scale on increasingly large datasets.

Performance evaluation will be visualized using **graphs** and **charts** to compare the efficiency and correctness of each algorithm under various conditions.

3. DESIGN

3.1. Modelling in String Matching

The **modelling** of string-matching operations refers to how we translate a real-world problem—finding patterns or substrings in larger text documents—into a computational system. A proper model is essential for creating scalable, efficient, and easy-to-understand implementations.

1. **Search Patterns and Data Representation**:
    - **Input Text**: A large body of plain text or string that we want to perform search operations on.
    - **Pattern/Substring**: The sequence or set of characters (a pattern) that we are searching for within the input text.
    - We may model these as:
        - Text as a **String** object or **char []** array.
        - Patterns as **substring literals** or **regular expressions** for matching.
    - Each string-matching algorithm, whether it is **Knuth-Morris-Pratt (KMP)**, **Boyer-Moore**, **Rabin-Karp**, or **Aho-Corasick**, uses different internal data structures (arrays, tables, or hash functions).
2. **Matching Context**:
    - **Single-pattern Matching**: A scenario where we find occurrences of a single pattern.
    - **Multi-pattern Matching**: A scenario where multiple patterns are searched at once (e.g., Aho-Corasick).

3.1.1. Behaviour-based Design

The **behaviour-based design** outlines how the different string-matching algorithms interact with the input data to detect matches or identify approximate matches. This design focuses on the various algorithmic behaviors for finding exact or fuzzy matches, their efficiency in processing large texts, and their optimal use cases based on pattern complexity, size, and expected matching results

1. Knuth-Morris-Pratt (KMP) Algorithm:

- **Preprocessing Behavior**: KMP optimizes the matching process by preprocessing the pattern into a **partial match table (also called a "prefix table")** that indicates the longest prefix that is also a suffix at each index of the pattern. This allows the algorithm to avoid unnecessary comparisons by skipping ahead in the pattern once a mismatch is detected.
- **Search Behavior**: When a mismatch happens, KMP uses the partial match table to shift the pattern rather than shifting it one character at a time.

2. Boyer-Moore Algorithm:

- **Preprocessing Behaviour**: The Boyer-Moore algorithm preprocesses the pattern into two important tables: the **bad-character shift table** and the **good-suffix shift table**.
    - The **bad-character shift** table tells you how far to shift the pattern when a mismatch happens, based on the character that failed the match.
    - The **good-suffix shift** table allows the algorithm to jump forward by skipping sections of the text that have been proven to match previous characters.
- **Search Behaviour**: By scanning the text from right to left and utilizing these heuristic tables, Boyer-Moore skips ahead to locations where it is more likely to find a match, making it efficient for large pattern searches.

3. Rabin-Karp Algorithm:

- **Preprocessing Behaviour**: The Rabin-Karp algorithm uses **hashing** to find potential matches. The first step involves computing the **hash value** for the pattern and for the substring of the text (of the same length).
    - A hash function is used to convert text and pattern into a fixed-size hash code.
- **Search Behaviour**: When comparing a pattern with a substring, Rabin-Karp compares the hash values of the two. If the hash values match, a character-by-character comparison follows, else it moves to the next substring.

4. Aho-Corasick Algorithm:

- **Preprocessing Behaviour**: Aho-Corasick is designed to search for multiple patterns simultaneously. It builds a **finite state machine (automaton)** from the set of patterns, creating a **trie (prefix tree)** where each state represents a potential match of a character in a pattern.
    - **Failure links** are created during preprocessing that allow the algorithm to efficiently backtrack through previously matched characters when mismatches occur, reducing redundant comparisons.
- **Search Behaviour**: Aho-Corasick allows for **multi-pattern matching** by transitioning between states as each character in the text is read. It is highly efficient for searching large numbers of patterns as it processes the text in linear time with respect to the length of the text and patterns.

5. Levenshtein Distance Algorithm (Fuzzy Matching):

- **Preprocessing Behaviour**: Unlike the previous algorithms that are mainly used for exact matching, Levenshtein Distance measures **edit distance**—the minimum number of operations (insertions, deletions, or substitutions) required to transform one string into another.
- **Search Behaviour**: Levenshtein Distance allows for **fuzzy string matching**, meaning it identifies not only exact matches but also close matches by comparing the calculated **edit distance** between a given pattern and substrings in the text.
    - The lower the **edit distance**, the closer the strings are. This algorithm is ideal for scenarios like spelling correction, DNA sequence alignment, or identifying misspelled words.
    - **Behaviour**:
        - For each potential substring of the text, calculate the **edit distance** to the pattern.
        - If the **edit distance** is within a certain threshold, consider it a match.

3.1.2. Use Case Model for Requirement Analysis

The **use case model** captures the functional requirements and user interactions with the string-matching tool.

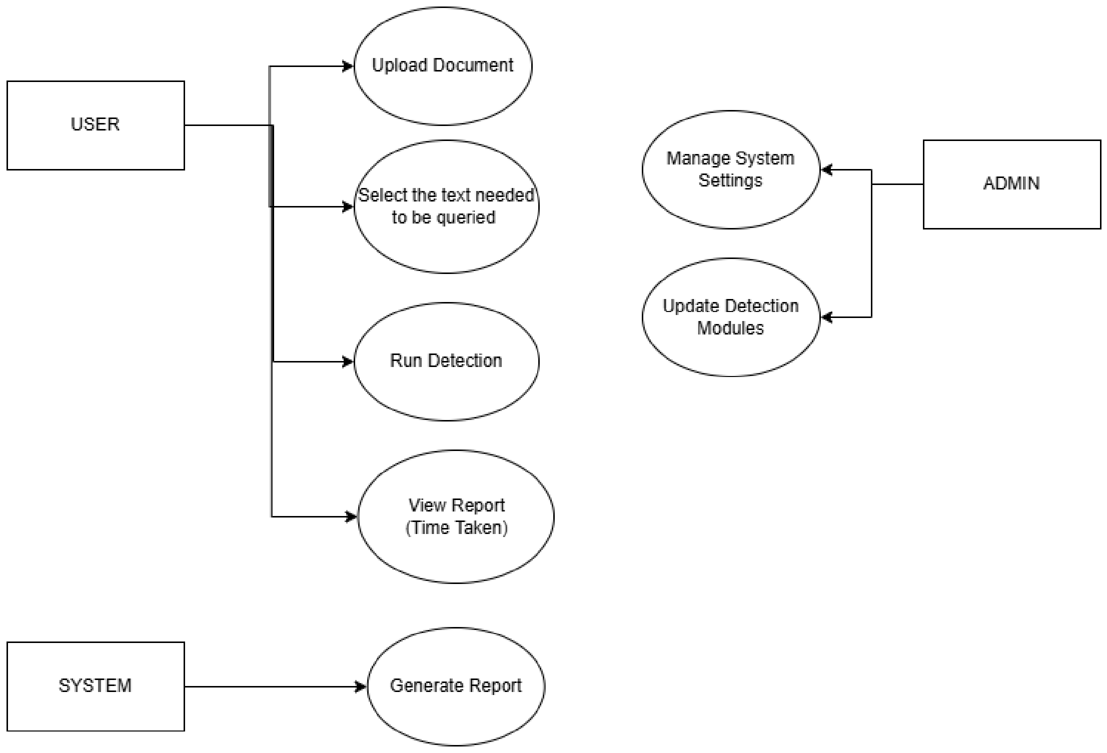


Fig. 3.1 Use Case Diagram

3.1.3. Class Diagram

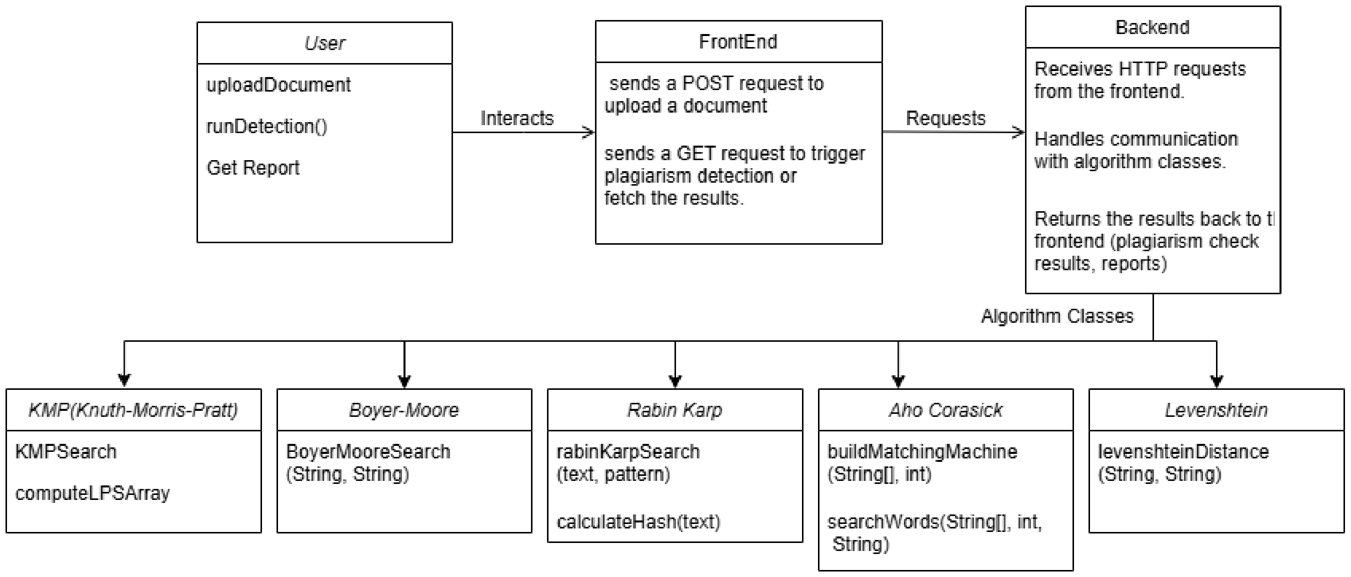

User
uploadDocument
runDetection()
Get Report
Interacts
FrontEnd
sends a POST request to upload a document
sends a GET request to trigger plagiarism detection or fetch the results.
Requests
Backend
Receives HTTP requests from the frontend.
Handles communication with algorithm classes.
Returns the results back to t frontend (plagiarism check results, reports)
Algorithm Classes
KMP(Knuth-Morris-Pratt)
KMPSearch
computeLPSArray
Boyer-Moore
BoyerMooreSearch (String, String)
Rabin Karp
rabinKarpSearch (text, pattern)
calculateHash(text)
Aho Corasick
buildMatchingMachine (String[], int)
searchWords(String[], int, String)
Levenshtein
levenshteinDistance (String, String)

Fig. 3.2 Class Diagram

3.1.4. Algorithmic Steps

Detailed Algorithm Steps for the string-matching techniques:

Naive String Matching:

- **Step 1**: Iterate over the string from 0 to (n-m), where n is the text length and m is the pattern length.
- **Step 2**: Compare the substring starting from the current position with the pattern.
- **Step 3**: If a match is found, record the starting position.
- **Step 4**: Repeat for all possible shifts.

Knuth-Morris-Pratt (KMP):

- **Step 1**: Preprocess the pattern to build the **partial match table**.
- **Step 2**: Iterate over the text, using the table to skip unnecessary comparisons (matching failures).
- **Step 3**: If the pattern matches, record the occurrence in the text.

Boyer-Moore:

- **Step 1**: Create two heuristic tables:
  - **Bad-character shift table**.
  - **Good-suffix shift table**.
- **Step 2**: Start matching from the rightmost side of the pattern.
- **Step 3**: Use the heuristics to skip over non-matching characters in the text.

Rabin-Karp:

- **Step 1**: Calculate the **hash** of the pattern and the initial substring in the text.
- **Step 2**: Slide the window across the text and compute the **hash** of the new substring.
- **Step 3**: If hashes match, check for exact match by comparing the characters.

Aho-Corasick:

- **Step 1**: Build a **finite state machine** (automaton) based on the patterns.
- **Step 2**: Traverse the automaton as you iterate over the text to find all occurrences of the patterns.

3.1.5. Technical Diagram

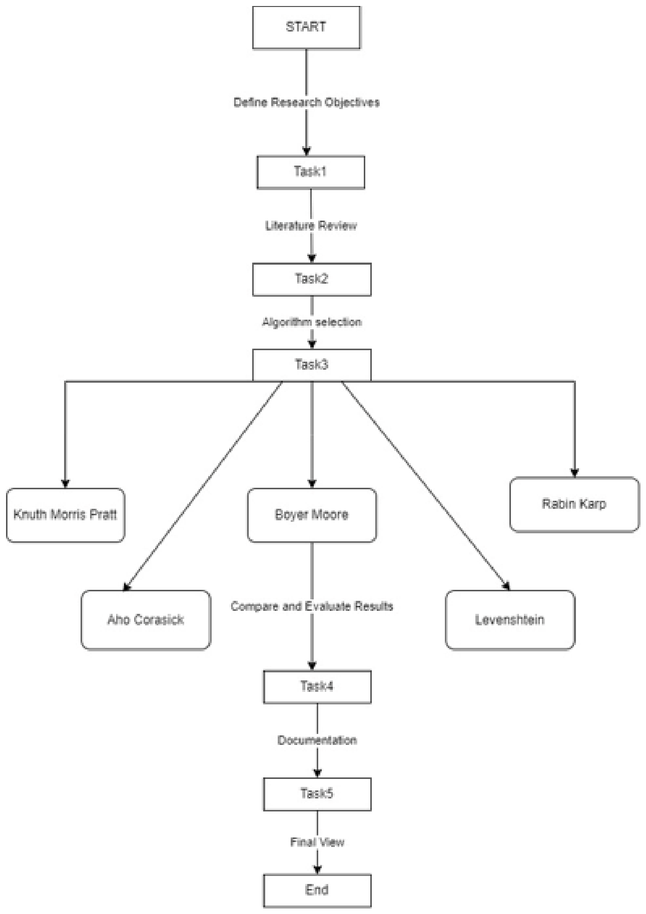


Fig. 3.3 Technical Diagram

4. ALGORITHMS OVERVIEW

A comprehensive overview of the different string-matching algorithms, outlining their features, specifications, and implementation details for detecting exact or approximate string matches in plain text. The focus is on **Knuth-Morris-Pratt (KMP)**, **Boyer-Moore**, **Rabin-Karp**, **Aho-Corasick**, and **Levenshtein Distance**, with a discussion on their suitability, advantages, and real-world applications.

4.1. Search Algorithms Overview

This part discusses each algorithm's internal working, its strengths, and weaknesses, and when it is most beneficial to use them. Below are the key algorithms for string matching:

1. Knuth-Morris-Pratt (KMP) Algorithm:

- **Internal Working**: KMP operates by preprocessing the pattern to create a **prefix table** that stores the length of the longest proper prefix which is also a suffix for any substring of the pattern. This information is used to avoid redundant character comparisons when a mismatch occurs.
- **Use Cases**: This algorithm is ideal for searching for a **single pattern** within a text where performance and matching efficiency are important.
- **Complexity**: Preprocessing is **O(m)** (where **m** is the length of the pattern), and the search phase is **O(n)** (where **n** is the length of the text).

2. Boyer-Moore Algorithm:

- **Internal Working**: The Boyer-Moore algorithm preprocesses the pattern into two key tables—**bad-character** and **good-suffix**. The **bad-character table** helps shift the pattern based on a mismatch, while the **good-suffix table** allows it to skip ahead after matching partial substrings. Boyer-Moore typically scans the text from **right to left** for optimal performance.
- **Use Cases**: Effective when searching for a single, long pattern, especially in large text files (e.g., text processing, search engines).
- **Complexity**: Worst-case is **O(n)**, but best-case performance (for large alphabets and mismatched characters) can achieve **O(n/m)**.

3. Rabin-Karp Algorithm:

- **Internal Working**: The Rabin-Karp algorithm uses **hashing** to speed up matching. It computes a hash value for the pattern and compares it with the hash values of text substrings of the same length. If the hashes match, it performs a character-by-character verification to avoid false positives.
- **Use Cases**: Best suited for **multiple patterns matching** or when detecting duplicates or plagiarism within a dataset of plain text.

- **Complexity**: For **one pattern**, the expected time complexity is **O(n)**, but for **multiple patterns**, it performs best when combining it with hash functions like **rolling hash**.

4. Aho-Corasick Algorithm:

- **Internal Working**: Aho-Corasick is designed for **multi-pattern matching**. It constructs a **finite state machine (automaton)**, which is a trie (prefix tree) built from multiple patterns. It can efficiently match multiple patterns simultaneously, scanning the text and transitioning between states.
- **Use Cases**: Used in applications like **virus scanners**, **search engines**, and **spell checkers**, where many patterns need to be searched at once.
- **Complexity**: **O(n + k)**, where **k** is the number of matches, making it highly efficient for large-scale searches with many patterns.

5. Levenshtein Distance Algorithm:

- **Internal Working**: Levenshtein Distance is a **dynamic programming** algorithm that measures how different two strings are by counting the minimum number of single-character edits (insertions, deletions, substitutions) needed to change one word into the other.
- **Use Cases**: This is an **approximate matching** technique useful in **fuzzy string matching** for cases where exact matches aren't critical, such as spelling correction, DNA sequencing, and natural language processing.
- **Complexity**: **O(mn)** where **m** is the length of the first string and **n** is the length of the second string.

4.2. Technical Specifications

This section discusses the **technical specifications** required for implementing each algorithm, the hardware/software environment, and any additional resources needed to run them efficiently.

1. General Hardware Requirements:

- **Processor**: Minimum of **2 GHz** processor (dual-core or higher recommended).
- **Memory (RAM)**: At least **4 GB** of RAM for handling larger text files (8 GB preferred for performance optimization during large searches).
- **Disk Space**: Sufficient space for storing input text files, pattern sets, and logs (depending on the size of the input files).

2. Software Requirements:

- **Java Development Kit (JDK)**: Version **8 or higher** for implementing and running algorithms.
- **Development IDE**: **IntelliJ IDEA** or **Eclipse** for building and debugging Java projects.
- **Libraries/Tools**:
  - **Apache Commons Lang**: To handle utility functions for string manipulation.

- **JUnit**: For writing test cases to ensure algorithm correctness.

3. Memory and Efficiency:

- **Pattern Storage**: Each pattern should be stored as a `String` object in Java, ensuring a balance between memory usage and ease of access during the search process.
- **Dynamic Data Structures**: Data structures like **arrays** (for KMP, Boyer-Moore) and **tries (prefix trees)** (for Aho-Corasick) will be used. Consider using **hash maps** for Rabin-Karp's hashing function and **2D arrays** for dynamic programming in Levenshtein.

4. Input and Output Formats:

- **Input**: Text and patterns to search will be accepted as **text files** (.txt) or **strings** (direct input in Java).
- **Output**: A list of positions where the patterns are found in the input text, or in the case of Levenshtein, the **edit distance** between strings and matching results will be provided.

4.3. Algorithm Features

Each of the algorithms has its own set of features that make it distinct and useful for various use cases. Below are the key features of each algorithm:

1. Knuth-Morris-Pratt (KMP):

- **Preprocessing Phase**: Helps in improving the overall search performance.
- **Time Efficiency**: **Linear time complexity (O(n + m))** is suitable for a single pattern search in large datasets.
- **Real-time Use Cases**: Web crawlers, document search engines, etc.

2. Boyer-Moore:

- **Substantial Pattern Shifting**: Efficiency comes from skipping large sections of text when possible.
- **Best suited for large and complex patterns** where the size of the alphabet is large (e.g., natural language texts).
- **Efficiency in Best Case**: Very fast compared to simpler algorithms.

3. Rabin-Karp:

- **Hashing Efficiency**: Can handle multiple patterns, making it optimal for tasks like plagiarism detection.
- **Randomized**: False positives may occur due to hash collisions, but can be reduced by refining the hash function.

4. Aho-Corasick:

- **Multi-pattern Matching**: Ideal for searching a fixed set of patterns in a text simultaneously.
- **Complexity Reduction**: Unlike simple algorithms, Aho-Corasick processes the entire text only once in **O(n + k)** time, regardless of the number of patterns.
- **Automaton Creation**: Constructs a **trie** to make searching faster.

5. Levenshtein Distance:

- **Approximate Matching**: Excellent for fuzzy string matching, allowing flexibility in real-world use cases like correcting typographical errors in text.
- **Edit Distance Calculation**: Not limited to exact matches, it's used for evaluating **similarity** between strings based on edit operations.

5. IMPLEMENTATION

5.1. Dataset Preparation

Before applying any string-matching algorithm, a comprehensive dataset needs to be prepared to evaluate how well each algorithm performs in different conditions. Here is a step-by-step breakdown of preparing the dataset:

1. **Data Collection**: The dataset should consist of varied text files that accurately represent typical use cases in which string matching will be performed. The dataset should include both simple, well-formatted documents and complex, noisy data with special characters, irregular spacing, and unstructured content.

   Examples of datasets:

   - **Plain Text Files**: Files with regular textual content such as books, articles, or essays.
   - **Large Document Collections**: A corpus of multiple documents to simulate real-world searching, such as a collection of web pages or technical documentation.
   - **Datasets with Noise**: Files containing irregularities such as special characters, HTML tags, formatting inconsistencies, etc., to test the algorithm's robustness.
2. **Pattern Collection**: These patterns should include:
   - **Fixed Patterns**: Single word or phrase-based patterns, typically words from the document or regular expressions.
   - **Multiple Patterns**: A list of words or phrases that the algorithms will search for simultaneously (for algorithms that support multi-pattern matching like Aho-Corasick).
   - **Variable-length Patterns**: Patterns of different lengths (e.g., short substrings or long sentences) to assess how well algorithms handle both short and long patterns.
3. **Data Preparation Steps**:
   - **Text Preprocessing**: Standard text cleaning procedures (removing special characters, whitespace normalization, etc.) should be applied to ensure the consistency of the datasets.
   - **Pattern Length Variations**: Patterns of varying lengths should be extracted and used for testing how the algorithms handle short and long patterns efficiently.
4. **Storage Format**: Store the dataset and patterns as plain text files (for ease of handling in the implementation), each containing the document text and corresponding pattern.

5.2. Scenarios

Different test scenarios are designed to check how well the string-matching algorithms perform under various conditions. Each scenario reflects a different real-world requirement for string matching.

Scenario 1: Small Text

• Description: This test case involves searching for a single pattern within a small body of text. The size of the text is limited to a few sentences or paragraphs to simulate real-time searching in smaller documents, emails, or smaller data sets.

• Example **Input**:

- **Text**: `" this is a simple example of a text to test pattern matching algorithms"`
- **Pattern**: `"pattern"`

• Objective: Assess the algorithm's performance in terms of speed and accuracy when working with smaller text files. Algorithms that perform string matching should be able to handle such input efficiently, as this case represents most typical usage for text searching applications.

• Expected **Outcome**: The algorithm should quickly find the pattern and return the correct index or indices within the text. The test will show how fast the algorithm performs, especially when there's less data involved.

Scenario 2: Large Text

• Description: This test case simulates a scenario where a pattern needs to be searched within a large document or a large collection of text, such as an article, long-form essay, or an entire book.

• Example **Input**:

- **Text**: `" this is a large example text used to test pattern matching algorithms on large inputs. "`

  `+ "Pattern matching is a crucial component in many computer science applications. "` *(Document is several pages long)*

- **Pattern**: `"large"`

• Objective: Evaluate how the algorithm scales to handle large amounts of text. This scenario will help assess the algorithm's efficiency in time-sensitive applications such as document search engines or systems that perform extensive text analysis.

• Expected **Outcome**: The algorithm should find the pattern in a reasonable time frame despite the large volume of text. The test will help assess how well algorithms perform as the data size increases.

Scenario 3: No Match

• Description: This test case represents the scenario where the pattern being searched does not exist within the text. It evaluates the algorithm's ability to handle situations where no matches are found and the necessary steps taken when this occurs (e.g., returning a "no match found" result).

• Example **Input**:

- **Text**: `"The quick brown fox jumped over the lazy dog."`
- **Pattern**: `"cat"`

• Objective: This case assesses whether the algorithm correctly identifies and reports no matches. A key part of efficient searching algorithms is handling cases where there are no hits in the document, without wasting unnecessary processing time.

• Expected **Outcome**: The algorithm should correctly report no match, ensuring efficient execution when there are no patterns to find.

5.3. Algorithms

Each of the different search algorithms used for string matching has its own unique strengths and weaknesses. Below are descriptions of each algorithm's implementation in the context of the different test scenarios:

Test Case	Algorithm	Time(seconds)
Test Case 1	KMP	0.0100041
	Boyer-Moore	0.0013903
	Rabin-Karp	0.0014382
	Aho-Corasick	0.0122259
Test Case 2	KMP	0.0001799
	Boyer-Moore	0.0002087
	Rabin-Karp	0.0002514
	Aho-Corasick	0.0004428
Test Case 3	KMP	0.0000065
	Boyer-Moore	0.0000145
	Rabin-Karp	0.0000044
	Aho-Corasick	0.0002292
Test Case 4	Levenshtein	0.0010059

Table 5.1 Comparison of Test Cases

5.3.1. Scenario -1: KMP Algorithm for Prefix-based Searching

- **Objective**: Use the **Knuth-Morris-Pratt (KMP)** algorithm for efficiently searching for a single fixed pattern within a text.
- **How it Works**: KMP preprocesses the pattern to build a **prefix table** that helps it avoid redundant comparisons when a mismatch occurs. Once the pattern is prepared, the algorithm can traverse the text and match the pattern in **linear time** (O(n)).
- **Implementation**:
  - **Text Input**: A large document of normal text.
  - **Pattern Input**: A single fixed word or phrase to search for.
  - **Key Operation**: Preprocess the pattern, create the prefix table, and begin matching characters sequentially across the text using the preprocessed information to skip unnecessary steps.
- **Expected Outcome**: The algorithm should efficiently identify the pattern, showing minimal redundancy or re-checking, especially with large documents.

5.3.2. Scenario -2: Aho-Corasick for Multi-pattern Matching

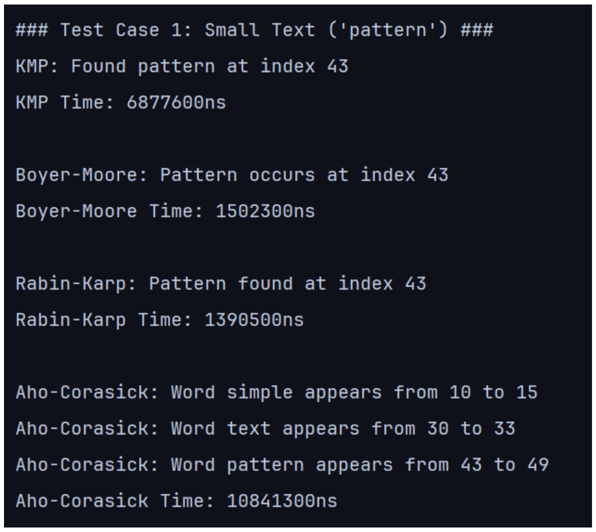
- **Objective**: Use the **Aho-Corasick algorithm** for matching **multiple patterns** against a text, leveraging its ability to create a **state machine (trie)** for efficient multi-pattern searching.
- **How it Works**: Aho-Corasick builds a **finite state automaton** that enables simultaneous matching of all given patterns within a text in linear time relative to the text length and the number of matches.
- **Implementation**:
  - **Text Input**: A large document or collection of multiple documents.
  - **Pattern Input**: Multiple patterns to search for simultaneously.
  - **Key Operation**: Build a trie for all input patterns, traverse through the text, and transition through states efficiently to find matches.
- **Expected Outcome**: The algorithm should return positions for all patterns found in the text, using a single pass through the document.

5.3.3. Scenario -3: Rabin-Karp for Hashing-based String Comparisons

- **Objective**: Use the **Rabin-Karp** algorithm for efficient string matching, particularly with **multiple patterns** by leveraging hash functions.
- **How it Works**: Rabin-Karp computes a hash of the pattern and compares it to the hash values of substrings of the text. In case of hash matches, it verifies the actual substring characters to avoid false positives.
- **Implementation**:
  - **Text Input**: A medium to large-size document.
  - **Pattern Input**: A list of patterns to detect.
  - **Key Operation**: Hash the patterns and compute hashes for substrings in the text. When a hash match is found, verify character-by-character to confirm the match.
- **Expected Outcome**: Rabin-Karp should efficiently process multiple patterns, though it may experience slower worst-case performance in certain hash collision scenarios.

6. OUTPUT SCREENS

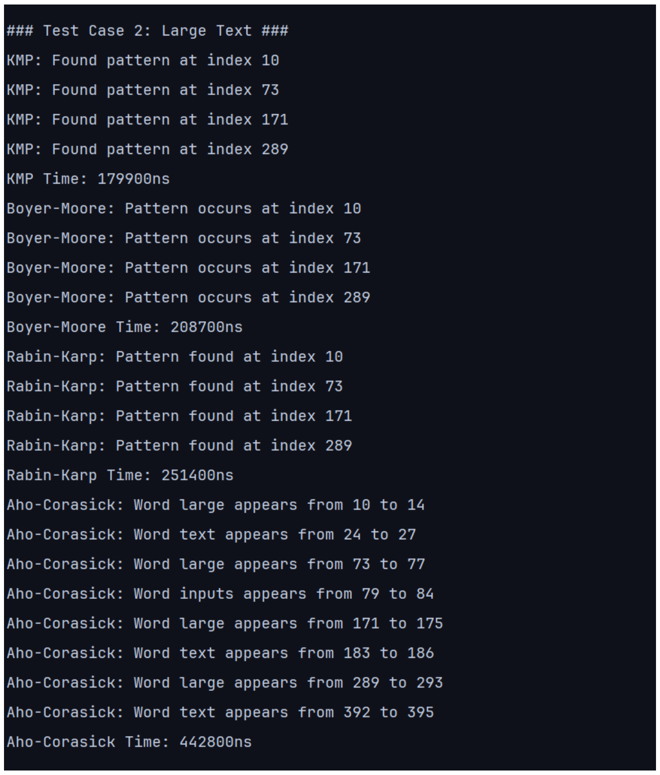
Test Case #1: Small Text



```
### Test Case 1: Small Text ('pattern') ###
KMP: Found pattern at index 43
KMP Time: 6877600ns

Boyer-Moore: Pattern occurs at index 43
Boyer-Moore Time: 1502300ns

Rabin-Karp: Pattern found at index 43
Rabin-Karp Time: 1390500ns

Aho-Corasick: Word simple appears from 10 to 15
Aho-Corasick: Word text appears from 30 to 33
Aho-Corasick: Word pattern appears from 43 to 49
Aho-Corasick Time: 10841300ns
```

Fig. 6.1 Small Text

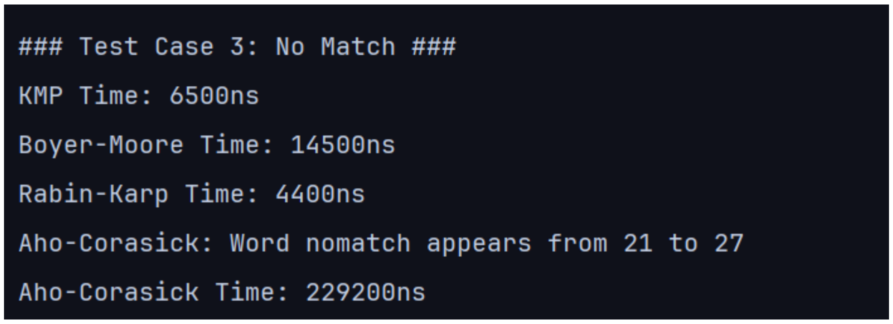
Test Case #2: Large Text



```
### Test Case 2: Large Text ###
KMP: Found pattern at index 10
KMP: Found pattern at index 73
KMP: Found pattern at index 171
KMP: Found pattern at index 289
KMP Time: 179900ns
Boyer-Moore: Pattern occurs at index 10
Boyer-Moore: Pattern occurs at index 73
Boyer-Moore: Pattern occurs at index 171
Boyer-Moore: Pattern occurs at index 289
Boyer-Moore Time: 208700ns
Rabin-Karp: Pattern found at index 10
Rabin-Karp: Pattern found at index 73
Rabin-Karp: Pattern found at index 171
Rabin-Karp: Pattern found at index 289
Rabin-Karp Time: 251400ns
Aho-Corasick: Word large appears from 10 to 14
Aho-Corasick: Word text appears from 24 to 27
Aho-Corasick: Word large appears from 73 to 77
Aho-Corasick: Word inputs appears from 79 to 84
Aho-Corasick: Word large appears from 171 to 175
Aho-Corasick: Word text appears from 183 to 186
Aho-Corasick: Word large appears from 289 to 293
Aho-Corasick: Word text appears from 392 to 395
Aho-Corasick Time: 442800ns
```

Fig. 6.2 Large Text

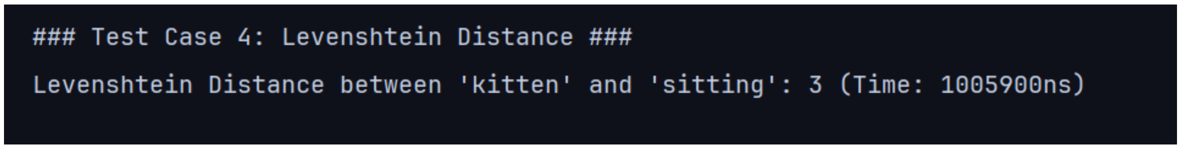
Test Case #3: No Match



```
### Test Case 3: No Match ###
KMP Time: 6500ns
Boyer-Moore Time: 14500ns
Rabin-Karp Time: 4400ns
Aho-Corasick: Word nomatch appears from 21 to 27
Aho-Corasick Time: 229200ns
```

Fig. 6.3 No Match

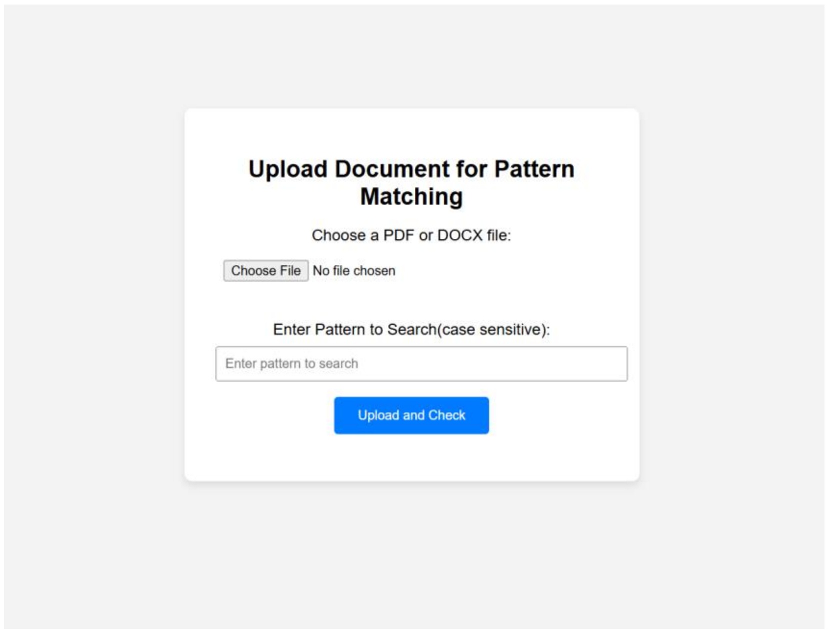
Test Case #4: Levenshtein Distance (Approximate Matching)



```
### Test Case 4: Levenshtein Distance ###
Levenshtein Distance between 'kitten' and 'sitting': 3 (Time: 1005900ns)
```

Fig. 6.4 Levenshtein

User Interface



Upload Document for Pattern Matching

Choose a PDF or DOCX file:

Enter Pattern to Search(case sensitive):

Fig. 6.5 Frontend UI

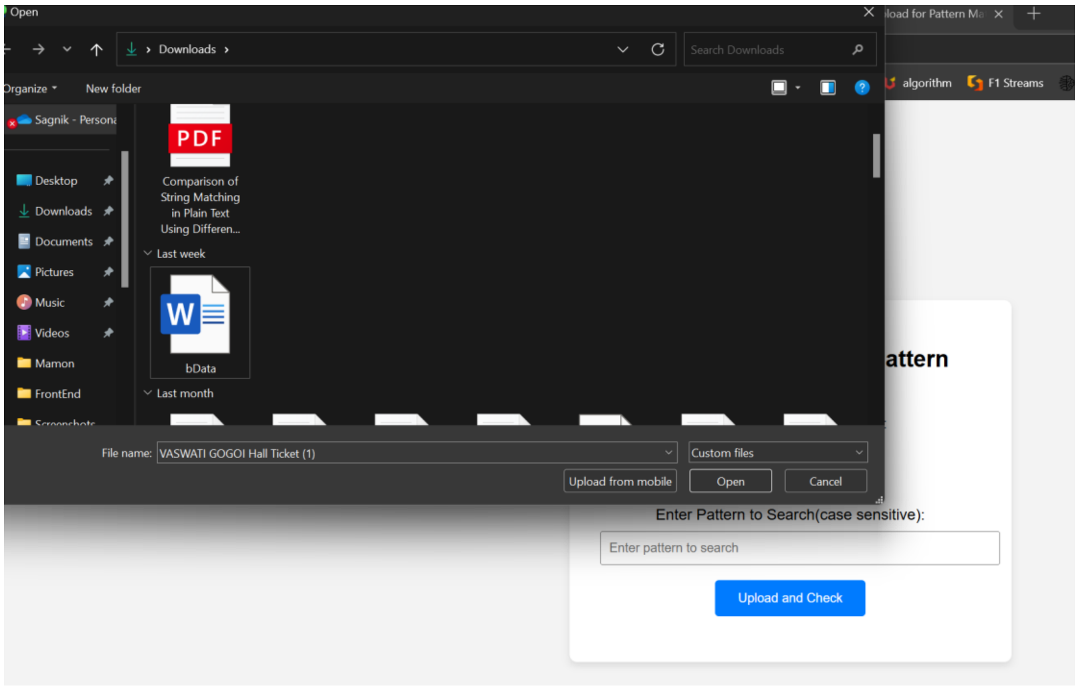
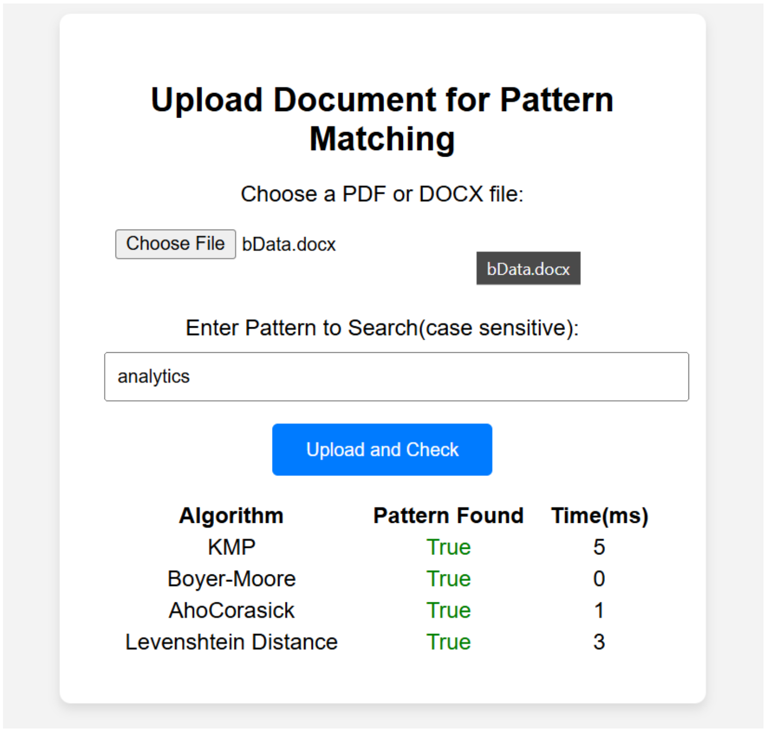


Fig. 6.6 Choosing Dataset



Upload Document for Pattern Matching

Choose a PDF or DOCX file:

Choose File bData.docx

bData.docx

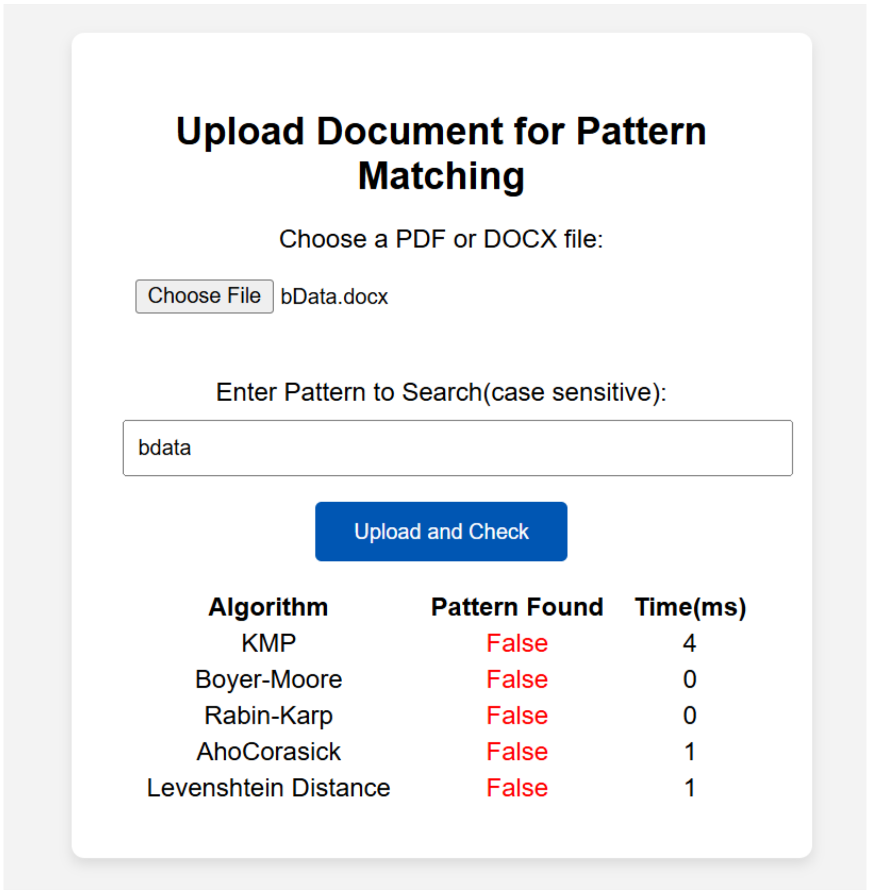
Enter Pattern to Search(case sensitive):

analytics

Upload and Check

Algorithm	Pattern Found	Time(ms)
KMP	True	5
Boyer-Moore	True	0
AhoCorasick	True	1
Levenshtein Distance	True	3

Fig. 6.7 Successful Match



Upload Document for Pattern Matching

Choose a PDF or DOCX file:

bData.docx

Enter Pattern to Search(case sensitive):

bdata

Algorithm	Pattern Found	Time(ms)
KMP	False	4
Boyer-Moore	False	0
Rabin-Karp	False	0
AhoCorasick	False	1
Levenshtein Distance	False	1

Fig. 6.8 Unsuccessful Match

7. LIMITATIONS & FUTURE ENHANCEMENT

7.1. Limitations

1. **Performance on Extremely Large Texts**: While the algorithms perform well for typical document sizes, handling **extremely large texts** (e.g., several gigabytes of data) might still require substantial computational resources, especially for complex algorithms like Levenshtein, which involve computing string distance for every comparison.
2. **Accuracy of Approximate Matching**: Although approximate matching is useful, it may not always provide perfect results. For instance, the **Levenshtein algorithm** may generate false positives if minor mismatches are interpreted as correct matches (for example, similar words that are completely unrelated but just a few characters off).
3. **Handling of Unicode or Multilingual Text**: This study focuses on **standard ASCII text**, and the performance of the algorithms may not be optimal for texts with special characters, non-ASCII characters, or content from **multiple languages** that introduce further complexities such as accents or character sets beyond the basic Latin alphabet.
4. **Pattern-Length Sensitivity**: While algorithms like **KMP** and **Rabin-Karp** perform well with short or fixed-length patterns, their performance could be impacted by searching for **very long patterns**, or those requiring computationally expensive pattern preprocessing.
5. **Memory Usage**: Some algorithms like **Aho-Corasick**, which constructs a large automaton, might use more memory as the number of patterns increases. This could be a limitation in scenarios requiring low memory consumption.

7.2. Future Enhancements

1. **Enhanced Approximate Matching**: Investigating **approximate pattern matching** algorithms based on more advanced techniques, such as **Dynamic Time Warping (DTW)** or **edit distance variations**, can improve accuracy, especially for noisy data or fuzzy matching.
2. **Parallelization**: Implementing parallel or multi-threaded versions of the string-matching algorithms could improve performance, especially for large-scale documents or when using complex patterns.
3. **Integration with Natural Language Processing (NLP)**: Combining string matching with NLP techniques could help further refine pattern recognition and enable more advanced searches (e.g., based on synonyms or semantic matches) by understanding the context rather than only relying on exact strings.
4. **Support for Unicode**: Modifying the algorithms to support **Unicode text** would improve the scope and applicability of the program for documents in different languages and text formats, making the string-matching system more robust across diverse datasets.
5. **Real-Time Matching Systems**: For applications like web search engines or live data processing, future enhancements could involve building algorithms that can handle **real-time streaming data**. This could include search systems capable of matching content from ongoing input, reducing latency in systems like live chat processing or security log analysis.

8. CONCLUSION

The project provided an in-depth comparison of string-matching algorithms, evaluating their performance in different scenarios and under varying conditions. We explored the strengths and weaknesses of algorithms like **Knuth-Morris-Pratt (KMP)**, **Aho-Corasick**, **Rabin-Karp**, and **Levenshtein Distance**, testing them across a variety of **small**, **large**, and **complex datasets**.

The results showed that:

- **Exact matching algorithms (like KMP)** perform well on small datasets but can struggle as the size of the text grows.
- **Multi-pattern search algorithms (like Aho-Corasick)** excel in scenarios that require identifying multiple patterns simultaneously.
- **Approximate matching algorithms** like Levenshtein provide valuable functionality by enabling fuzzier pattern matching in real-world data, where errors or slight differences are common.

Overall, this research highlights the importance of choosing the right algorithm for specific string-matching needs and provides insight into how well these algorithms perform when scaling with document size and pattern complexity.

Key Takeaways

- **Performance Factors**: The choice of algorithm significantly impacts the speed, accuracy, and computational requirements, especially as document size and the number of patterns increase.
- **Appropriate Matching Contexts**: Exact match algorithms work best for applications needing perfect string matches, while approximate matching is useful for handling noisy or imperfect data.

By analysing the results from all test cases and understanding the nature of each algorithm, users can select the most appropriate string-matching approach based on the specific challenges and data scenarios they encounter.

REFERENCES

1. Andrejko, H., & Glinos, K. I. (2011). Parallel Knuth-Morris-Pratt String Matching Algorithm on Multicore Processors. International Conference on High Performance Computing (HiPC).
2. Hidayati, N., & Mitra, S. K. (2008). A Fast Hybrid String Matching Algorithm: Boyer-Moore-Horspool with Simplified Shift Heuristics. International Journal of Computer Applications, 4(3), 18-24.
3. Mahmood, F., & Manzoor, S. (2016). Parallel Rabin-Karp String Matching Algorithm for Multicore Processors. Journal of Parallel and Distributed Computing, 96, 31-42.
4. Kochet, J., & Bala, P. (2013). Parallel Aho-Corasick Algorithm on GPU for Efficient Network Intrusion Detection Systems. International Conference on Networking and Services (ICNS).
5.Sun, Wenhai, et al. "Privacy-preserving multi-keyword text search in the cloud supporting similarity-based ranking." Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security. 2013.
6. Lankford, S., & Chandrasekaran, S. (2017). Vectorized Levenshtein Distance Calculation for Real-Time Applications. Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS).