BIRMINGHAM CITY
University

CMP6200/DIG6200

INDIVIDUAL UNDERGRADUATE PROJECT

2023–2024

A3: Dissertation

EXACT FILE/STRING MATCHING ALGORITHMS-COMPARATIVE ANALYSIS

d

Course: BSc. (Hons) in Computer Science
Student Name: Dhananjay Tewari
Student Number: 21150306
Supervisor Name: Khaled Mahabub

Abstract

String matching algorithm plays a major role in a wide range of applications. The role of text preprocessing in these algorithms is also significant which cleans noisy data and makes the algorithms more efficient. The exploration and evaluation of the exact string matching is proposed to build an understanding of string matching algorithms and text preprocessing using NLP. In this paper, Four major exact matching algorithms which are the Boyer Moore algorithm, Knuth-Morris-Pratt, Rabin Karp, and Brute Force algorithms are explored they have major differences in their searching techniques and preprocessing was done using NLP preprocessing techniques like stop word removal, punctuation removal, and tokenization. Using these 4 algorithms and preprocessing techniques the comparison was done between the algorithms as well as for the pre-processed text and Non pre-processed text. The result demonstrates the effectiveness of text preprocessing in all of the algorithms these results are more significant in Language-based patterns and text rather than on the genome text pattern or in special character text.

Acknowledgments

I would like to express my sincere gratitude to Dr. Khaled Mahbub for their invaluable guidance, support, and mentorship throughout this project. Their expertise, encouragement, and insightful feedback have been instrumental in shaping the direction of this research and ensuring its successful completion.

I am deeply thankful to Birmingham City University for providing a conducive academic environment and institutional support has been pivotal in facilitating my academic journey and enabling me to pursue this endeavor.

Thank you to everyone who has contributed to the realization of this project. Your support has been invaluable, and I am truly grateful for the opportunity to undertake this research.

Sincerely,

Dhananjay Tewari

Table of Contents

Glossary

BM :	Boyer-Moore Algorithm
KMP:	Knuth-Morris-Pratt algorithm
BF :	Brute Force algorithm
Rk :	Rabin Karp algorithm
BMH:	Boyer-Moore-Horspool algorithm
BMHS:	Boyer-Moore-Horspool-Sunday algorithm
BMHS2:	Boyer-Moore-Horspool-Sunday version 2 algorithm

List of Figures

List of Tables

1 Introduction

The primary objective of this project is to explore and evaluate various exact string/ file matching algorithms. The aim is to gain a deeper grasp of challenges encountered during the processing of these algorithms. This report will describe the key facts of the project, including its aim and objectives, resources, and strategic methodologies. The report will commence with a methodology for the literature search, outlining the approach for reviewing relevant literature which will be followed by a comprehensive literature review. Subsequently, the report will delve into the methods and implementation of the project. This section will detail the practical application of the methodologies discussed earlier in this report. Finally, the report will evaluate the results, complete it with testing, and thoroughly discuss the outcomes. The report will conclude with a summary of the findings and their implications.

1.1 Problem Definition

The field of computer science has witnessed extensive research on string matching over the past four decades, primarily due to its wide-ranging applications. More than 80 string-matching algorithms have been proposed, underscoring the pivotal role string matching plays in various applications. The necessity for comparative analysis in this field is a good practice to select the most suitable algorithm for a specific application. Therefore, keeping this field up-to-date with improvements through the comparison of algorithms using new techniques is crucial. (Chayapathi, 2021)

In the realm of exact string matching, the challenges lie in developing an algorithm that can accurately identify and link related information within large datasets. Existing algorithms often grapple with issues such as scalability, accuracy, and handling noisy or unstructured data. Therefore, there is a need to explore innovative techniques and methodologies to improve the accuracy and efficiency of exact string-matching algorithms ((Zhou, Y. and Pang, R., 2019, December); (Leonardo, B. and Hansun, S., 2017); (Glück, R. and Yokoyama, T., 2022); (Vayadande, K., Mandhana, R., Paralkar, K., Pawal, D., Deshpande, S. and Sonkusale, V., 2022); (Zhao, C. and Sahni, S., 2019); (Tarhio, J. and Ukkonen, E., 1993))

The role of text preprocessing in these algorithms is also significant. It can help in improving the efficiency of these algorithms by reducing the noise in the data. However, the choice of preprocessing methods can greatly affect the performance of these algorithms ((Vijayarani, S., Ilamathi, M.J. and Nithya, M., 2015); (Chai, 2023); (Camacho-Collados, J. and Pilehvar, M.T., 2017)).

1.2 Scope

The scope of this project encompasses the development and evaluation of several exact string matching algorithms. Additionally, the project will explore the integration of preprocessing techniques to enhance the performance of these algorithms on various datasets representative of real world data to assess their accuracy and efficiency.

1.3 Rationale

Current approaches to file matching primarily involve the use of exact string-matching algorithms such as KMP and BMHS2 (Zhou, Y. and Pang, R., 2019, December), Rabin-Karp, and Jaro-Winkler Distance (Leonardo, B. and Hansun, S., 2017). These algorithms are designed to detect even the slightest modification in text/files. However, these methods often grapple with issues such as scalability, accuracy, and handling noisy or unstructured data (Glück, R. and Yokoyama, T., 2022).Furthermore, the choice of preprocessing methods can greatly affect the performance of these algorithms ((Vijayarani, S., Ilamathi, M.J. and Nithya, M., 2015); (Chai, 2023); (Camacho-Collados, J.

and Pilehvar, M.T., 2017)). Therefore, there is a need for more efficient and accurate file-matching methods and Comparative analysis of the algorithm takes a step in that direction to analyze the differences in different ways.

1.4 Project Aim and Objectives

1.4.1 Aim

This project aims to explore and evaluate advanced string-matching algorithms to enhance accuracy and efficiency. This project also aims to design a user-friendly interface to ensure accessibility and ease of use for a wider audience. The exponential growth of digital data in recent years has underscored the importance of efficient data management and analysis techniques. The report aims to contribute to this field by Exploring and evaluating advanced exact string-matching algorithms: Boyer Moore algorithm, KMP algorithm, Rabin Karp Algorithm, and Brute Force Algorithm.

1.4.2 Objectives

- Reviewing the Exact string-matching Algorithms: BM, KMP, Brute Force, and Rabin Karp.
- Exploring and evaluating the Exact string-matching algorithm
- Exploring NLP preprocessing techniques like Stop word removal, Lemmatization, and tokenization
- Comparing the results for different algorithms.

1.5 Background Information

The proposed project primarily focuses on exact string-matching algorithms and preprocessing techniques. Extensive research in this field has led to the development of numerous efficient algorithms ((Zhou, Y. and Pang, R., 2019, December); (Leonardo, B. and Hansun, S., 2017), (Glück, R. and Yokoyama, T., 2022)). Furthermore, it has been observed that there are various variations of these efficient algorithms, indicating the vastness and diversity of this area of interest ((Chayapathi, 2021); (Tarhio, J. and Ukkonen, E., 1993)).

Numerous reports and studies have been conducted on similar algorithms, and comparative analyses are commonly found in the literature ((Vijayarani, S., Ilamathi, M.J. and Nithya, M., 2015); (Chai, 2023) (Camacho-Collados, J. and Pilehvar, M.T., 2017)). However, a comprehensive study that combines these algorithms with a user-friendly interface is noticeably absent. This presents a unique opportunity to contribute to the field by not only improving the accuracy of exact string-matching algorithms but also enhancing their accessibility and usability ((Saho, 2023) (Vayadande, K., Mandhana, R., Paralkar, K., Pawal, D., Deshpande, S. and Sonkusale, V., 2022)).

2 Literature Review

2.1 Themes

The two primary themes are string-matching Algorithms and Natural Language Processing (NLP)Preprocessing techniques. These themes are fundamental to understanding text processing and analysis, playing a crucial role in various domains such as data mining, machine learning, etc.

- String Matching Algorithm: This theme focuses on algorithms designed to efficiently search for specific patterns, strings, and substrings in text data. Examples include Knuth-Morris-Pratt (KMP), Boyer Moore (BM), Rabin-Karp, and Brute Force Algorithm.
  - Exact Matching Algorithms: This is the subcategory of String-matching algorithms that focuses on Algorithms whose results are exact matches to the pattern.
  - Approximate String matching: This is the second subcategory of string-matching algorithms that focuses on the Algorithm that gives closest matches to the pattern.
- Natural Language Processing Text preprocessing: NLP preprocessing involves transforming raw text into a format suitable for processing and analysis. Techniques include lowercasing, tokenization, punctuation removal, and stop word removal.

2.2 Review of Literature

2.2.1 String Matching Algorithm

String matching algorithms play a pivotal role across various domains such as data mining, machine learning, forensics, network security, defense, and space exploration. These algorithms are instrumental in identifying patterns within given texts. This literature review aims to provide an extensive overview of the seminal studies and projects that have significantly contributed to the evolution and optimization of these algorithms.

Data Validation and Reconciliation of Student Marks and Data (Saho, 2023.):

Saho proposes a methodology to ensure the accuracy of student marks stored across disparate files. The approach involves file matching to detect discrepancies, followed by rectification using the same program, leveraging data reconciliation techniques and matching algorithms. (Saho, 2023)

Pattern Matching in File System (Vayadande, Mandhana, & Paralkar, 2022):

Vayadande et al. utilize three string matching algorithms - KMP, Naïve, and finite automata - to search for file names within a file system. Initially command-line-based, the project holds promise for enhancement through the integration of a graphical user interface (GUI). (Vayadande, K., Mandhana, R., Paralkar, K., Pawal, D., Deshpande, S. and Sonkusale, V., 2022)

Survey and Comparison of String Matching Algorithms (Chayapathi, A.R., 2021):

This survey presents a succinct comparison of various algorithms to identify the most efficient ones for string matching. It concludes that Boyer Moore and Knuth Morris algorithms exhibit lower time complexity and are versatile enough to be applied across diverse scenarios, mitigating each other's worst-case scenarios. (Chayapathi, 2021)

Approximate Boyer-Moore String Matching (Tarhio, J. and Ukkonen, E., 1993):

Tarhio and Ukkonen introduce and analyze new algorithms addressing two approximate string problems: K mismatches and K differences, leveraging the Boyer-Moore string matching algorithm. (Tarhio, J. and Ukkonen, E., 1993)

Fast-Search: A New Efficient Variant of the Boyer-Moore String Matching Algorithm (Cantone & Faro, 2016):

Cantone and Faro propose a variant of the Boyer-Moore string-matching algorithm called fast search. Despite a quadratic worst-case scenario, the algorithm demonstrates exceptional practical performance. (Cantone, D. and Faro, S., 2003)

Reversible Programming: A Case Study of Two String Matching Algorithms (Gluck & Yokoyama, 2022):

Gluck and Yokoyama explore the development of two reversible versions of string matching algorithms: the naïve (brute force) algorithm and the Rabin Karp algorithm. (Glück, R. and Yokoyama, T., 2022)

The Analysis of KMP Algorithm and its Optimization (Lu, 2019, November)**:**

This paper delves into the KMP algorithm and its optimizations, introducing a new variation, L-I_KMP, which utilizes a letter-numbered table and last-identical array to facilitate larger jumps and eliminate unnecessary comparisons. (Lu, 2019, November)

Text Documents Plagiarism Detection using Rabin-Karp and Jaro-Winkler Distance Algorithms (Leonardo, B. and Hansun, S., 2017)**:**

This study compares Rabin-Karp and Jaro-Winkler distance algorithms for detecting plagiarism in text documents, with Rabin-Karp demonstrating superior efficiency in terms of similarity rate and completion time. (Leonardo, B. and Hansun, S., 2017)

Research of Pattern Matching based on KMP and BMHS2 (Zhou & Pang, 2019):

Zhou and Pang focus on enhancing the time performance of pattern-matching algorithms, primarily by minimizing character comparisons and increasing the matching window's rightward movement distance upon a mismatch. (Zhou, Y. and Pang, R., 2019, December)

A Variation on Boyer Moore Algorithm (Lecroq, 1992):

Lecroq presents a significant modification to the Boyer-Moore string matching algorithm, extending its capabilities by computing the length of the longest prefix of the word ending at a given position in the text. (Lecroq, 1992)

On Improving the Worst Case Running Time of the Boyer-Moore String Matching Algorithm (Galil, 1979):

Galil proposes a modification to the Boyer-Moore string matching algorithm to ensure linear worst-case running time. (Galil, 1979)

String Correction using Damerau-Levenshtein Distance (Zhao & Sahni, 2017):

Zhao and Sahni investigate the Damerau-Levenshtein distance metric and develop space and cache-efficient algorithms for computing the distance between two strings and determining a sequence of edit operations equal to the distance. (Zhao, C. and Sahni, S., 2019)

Research on String Similarity Algorithm based on Levenshtein Distance (Zhang, Hu, & Bian, 2017):

Zhang, Hu, and Bian introduce an improved similarity algorithm based on Levenshtein distance, combined with the longest common subsequence (LCS) and longest common substring (LCCS) to provide more precise and accurate string similarity results. (Zhang, S., Hu, Y. and Bian, G., 2017)

2.2.2 Natural Language Processing (NLP) Text Preprocessing

Text preprocessing is a critical stage in Natural Language Processing (NLP), significantly impacting the final performance of NLP systems. This literature review aims to offer a comprehensive overview of the key studies contributing to the development and optimization of text preprocessing techniques.

Preprocessing Techniques for Text Mining - An Overview (Vijayarani & Ilamathi, 2015):

Vijayarani and Ilamathi discuss the significance of text preprocessing in text mining, emphasizing key steps such as stop words removal, stemming, and TF/IDF algorithms, highlighting its pivotal role in text mining techniques and applications. (Vijayarani, S., Ilamathi, M.J. and Nithya, M., 2015)

Comparison of Text Preprocessing Methods (Chai, 2023):

Chai investigates the impact of various text preprocessing decisions on the performance of a standard neural text classifier, providing insights into the pros and cons of tokenizing, lemmatizing, lowercasing, and multiword grouping, underscoring the importance of selecting the appropriate method for each dataset and application. (Chai, 2023)

On the Role of Text Preprocessing in Neural Network Architectures: An Evaluation Study on Text Categorization and Sentiment Analysis (Camacho-Collados & Pilehvar, 2017):

Camacho-Collados and Pilehvar evaluate the role of text preprocessing in neural network architectures, conducting extensive assessments on standard benchmarks from text categorization and sentiment analysis. Their findings indicate that simple tokenization generally suffices, but they note significant variability across preprocessing techniques. (Camacho-Collados, J. and Pilehvar, M.T., 2017)

2.3 Review Summary

Reviewing the above Literature gives an Overview of different types of String-matching algorithms and Text preprocessing techniques. The above literature gives multiple variations of strings matching algorithms for both approximate and exact matching. For instance, there are various variations of the BM algorithm: approximate, BMH, BMHS, and BMHS2 which gives the proper understanding of the String-matching algorithms. For Text preprocessing there are multiple techniques was summarized in the above literature like Stop word removal, TI-IDF vectorization, tokenization, lemmatizing, and stemming which used to make the text clear for Neural Network Architecture and also for data mining, etc. This gives an overall understanding of many algorithms and preprocessing techniques.

3 Methods and Implementation

3.1 Methodology

The Methodology of this project involves a systematic and structured approach to evaluating and improving file-matching algorithms. The first step is to conduct a comprehensive review of some of the existing algorithms. This review will focus on understanding the strengths and weaknesses of each algorithm, particularly in terms of their accuracy and efficiency.

Understanding the Objectives The most suitable methodology and model for this project will be the agile model as It emphasizes flexibility which is useful for user-friendly design and development. It Focuses on iterative and incremental development which is repeating the cycle of software development phases. Each process is revisited in every cycle

where little improvement is done with dynamic planning Next exploration of Natural language processing techniques to enhance the performance of these algorithms. NLP techniques can help preprocess the data, making it more suitable for file-matching tasks. Experiment with various preprocessing techniques like tokenization, and stop word removals and evaluate their impact on the performance of the exact String-matching algorithms.

3.2 Design

3.2.1 User requirement:

User requirements for the file-matching algorithms are:

- ❖ The system should allow users the ability to choose different file-matching algorithms.
- ❖ User must input the file path and pattern to the system.
- ❖ System should preprocess the file given
- ❖ The system should search for the pattern accurately through the file.

3.2.2 Flow Chart:

3.2.2.1 System Flow chart:

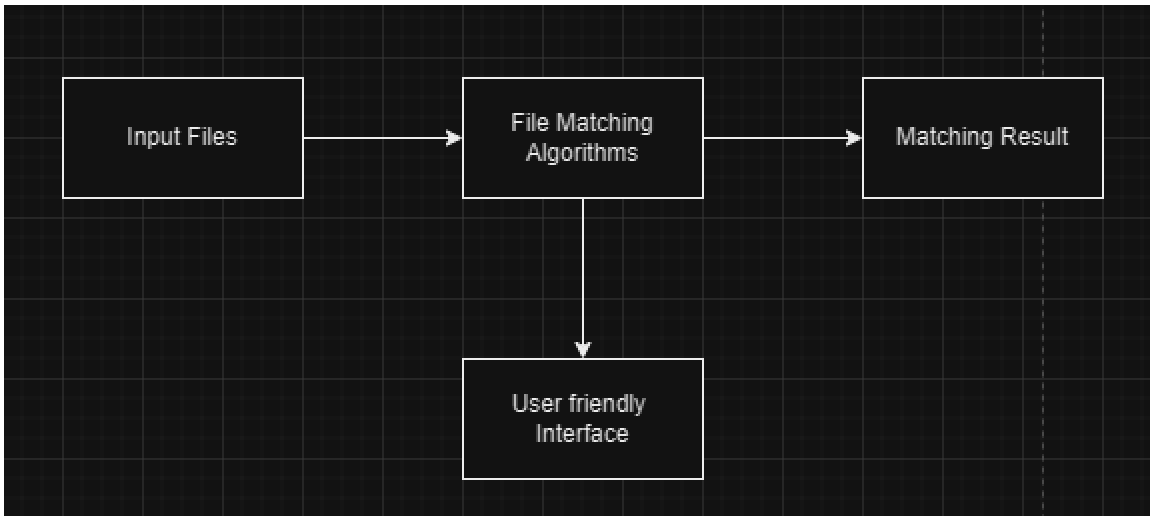


Figure 1: System Flowchart

The Above diagram shows the basic understanding of the file matching system which goes from input files to matching results all this is happening on the user-friendly system.

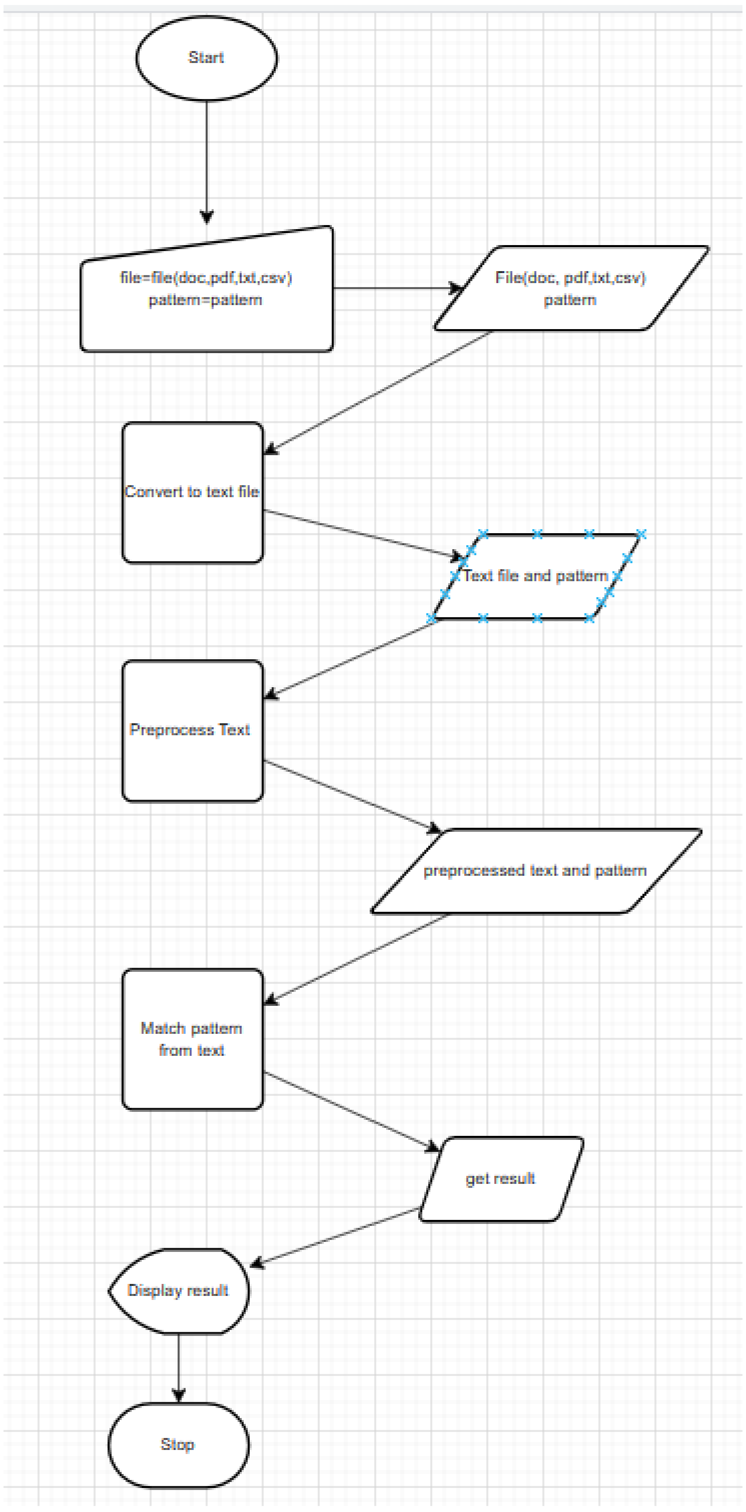
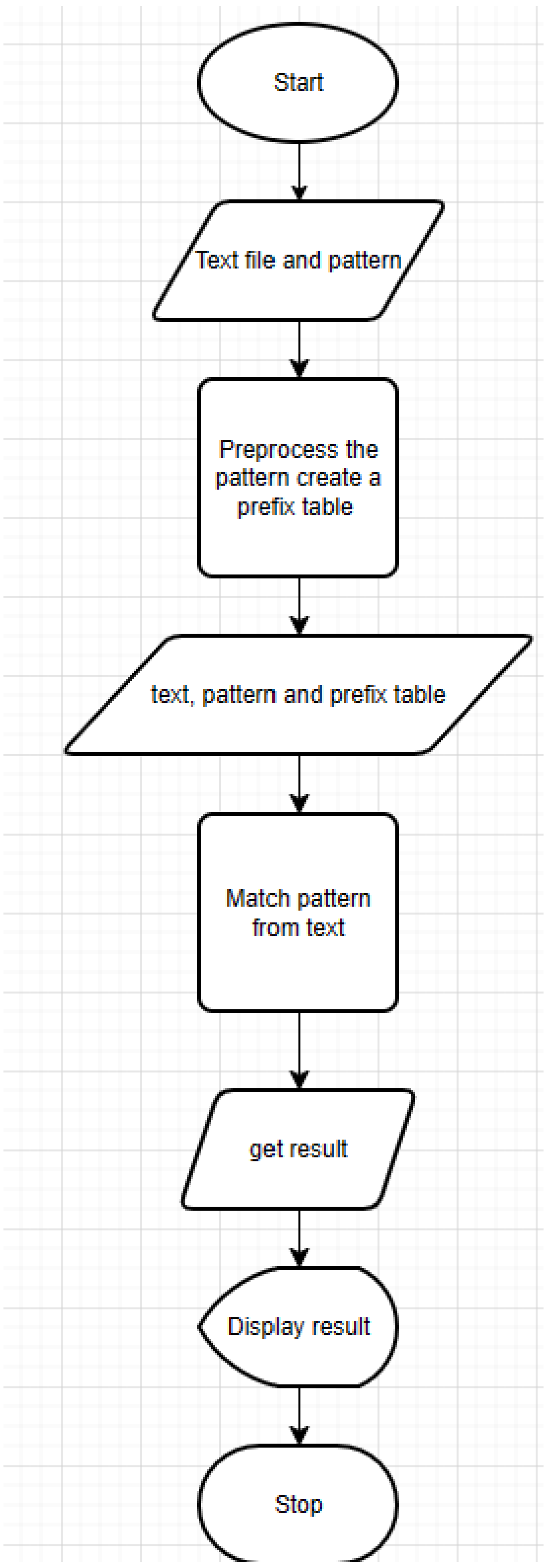


Figure 2: System Flow chart

The above flowchart shows the basic workings of pattern matching in the system which starts with taking the file(.txt,.docx,.csv,.pdf) and pattern input then converting the inputted file into the text file then preprocessing the file and the pattern using the text preprocessing stop word removal and punctuation removal than inputting the pre-processed file and pattern to the matching algorithm after matching the pattern in the text display result.



3.2.2.2 Algorithmic flowchart:

3.2.2.2.1KMP Algorithm:

The Flow diagram of the KMP Algorithm is as follows:

Input: The system takes two inputs - a text string and a pattern string.

Preprocessing: The system preprocesses the pattern to create a prefix table. This table helps in determining the next position the pattern should be checked from when a mismatch occurs.

Matching: The system then scans the text string from left to right. For each character in the text, it checks for a match with the pattern. If a match is found, it moves to the next character in both the text and the pattern. If a mismatch is found, it uses the prefix table to determine the next position in the pattern to check from.

Output: The system outputs the positions in the text where the pattern is found.

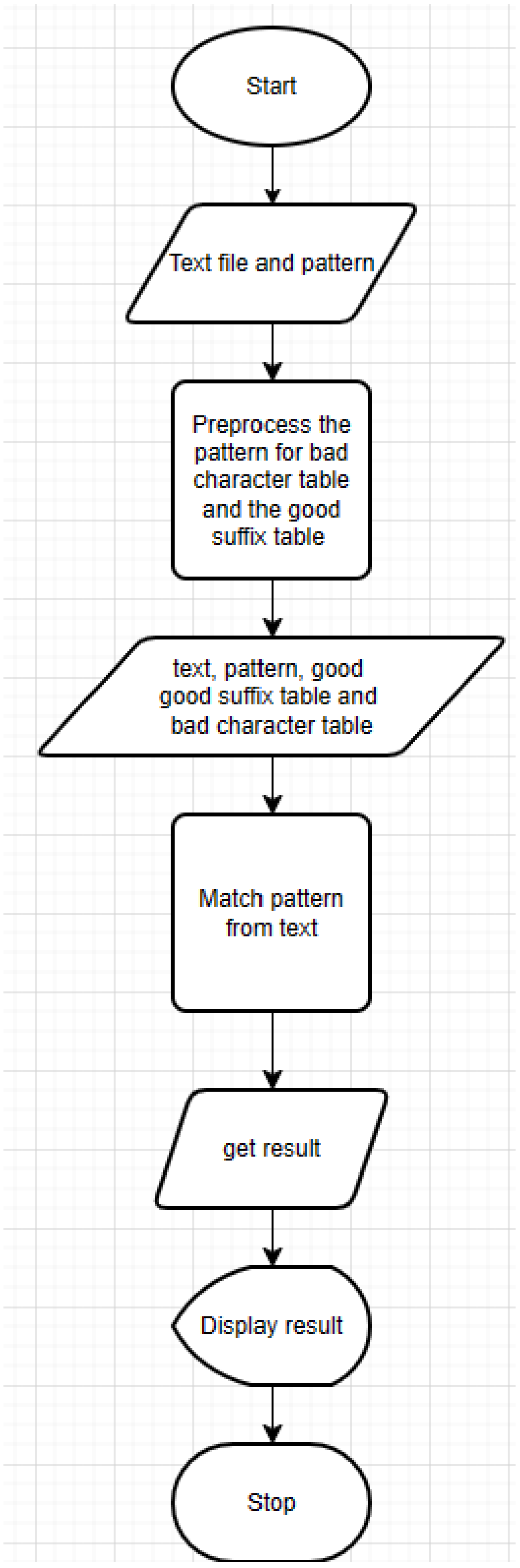
Figure 3: Flowchart for KMP Algorithm

3.2.2.2.2 BM Algorithm

The Flow diagram for the BM Algorithm is as follows:

Input: The system takes two inputs - a text string and a pattern string.

Preprocessing: The system preprocesses the pattern to create two tables:


Start
Text file and pattern
Preprocess the pattern for bad character table and the good suffix table
text, pattern, good good suffix table and bad character table
Match pattern from text
get result
Display result
Stop

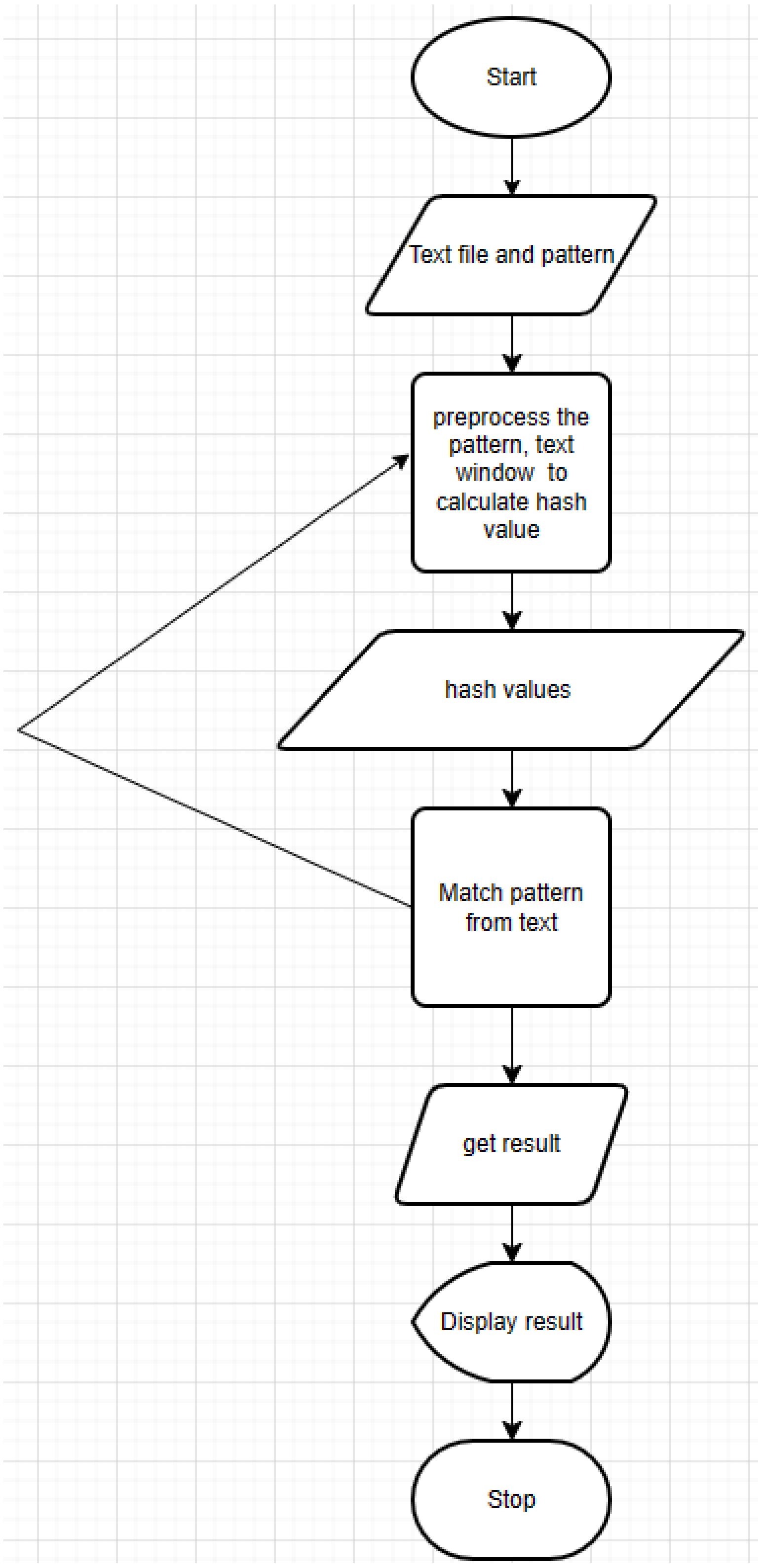
Bad Character Table: This table is used when a character in the text does not match the current character in the pattern.

Good Suffix Table: This table is used when all characters in the current window of the text match the pattern.

Matching: The system then scans the text string from right to left. For each character in the text, it checks for a match with the pattern. If a match is found, it moves to the next character in both the text and the pattern. If a mismatch is found, it uses the bad character table and the good suffix table to determine the next position in the pattern to check from.

Output: The system outputs the positions in the text where the pattern is found.

Figure 4: Flowchart for Boyer Moore Algorithm



3.2.2.2.3 Rabin Karp algorithm

The flow chart diagram for the Rabin Karp algorithm is as follows:

Input: The system takes two inputs - a text string and a pattern string.

Preprocessing: The system preprocesses the pattern to calculate its hash value.

Matching: The system then scans the text string from left to right. For each substring of the text of the same length as the pattern, it calculates the hash value and compares it with the hash value of the pattern. If the hash values match, it performs a character-by-character comparison.

Output: The system outputs the positions in the text where the pattern is found.

Figure 5: Rabin Karp Algorithm Flow chart

3.2.2.2.4 Brute Force Algorithm

Input: The system takes two inputs - a text string and a pattern string.

Matching: The system then scans the text string from left to right. For each character in the text, it checks for a match with the pattern. If a match is found, it moves to the next character in both the text and the pattern. If a mismatch is found, it moves the pattern one character to the right.

Output: The system outputs the positions in the text where the pattern is found.

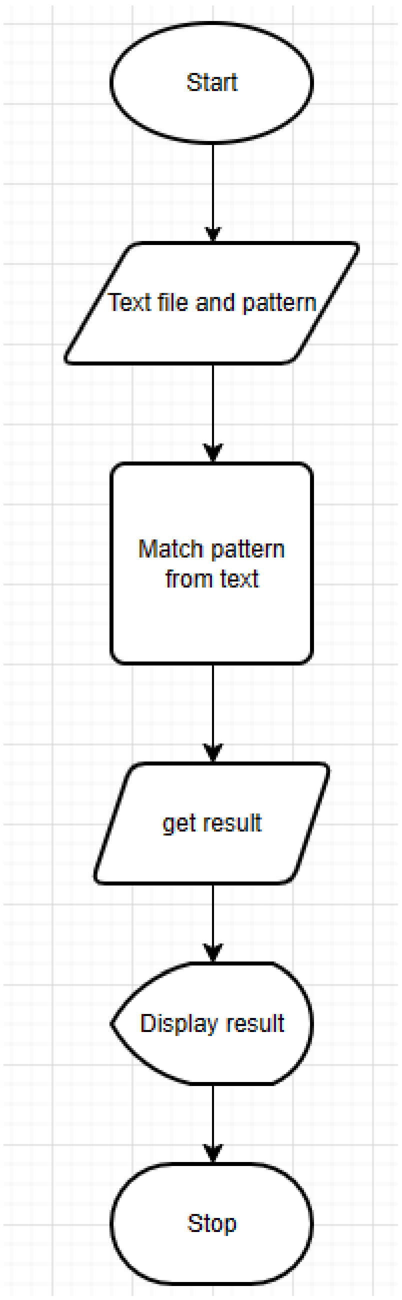


Figure 6: Flow chart for Brute force algorithm

3.3 Implementation

3.3.1 Overview

The System was designed with modular architecture, allowing different file matching algorithms to be plugged in as needed. This design enables easy testing and comparison of different algorithms.

The user interface was designed to be simple and intuitive, allowing users to easily select initiate the file matching process.

In this section, the detailed practical implementation of file/string matching algorithms is discussed in the preceding literature review. The implementation includes the KMP algorithm, BM algorithm, Rabin Karp Algorithm, and Brute Force Algorithm which are four main algorithms. Before the Implementation of the Algorithm mentioned Implementation of the preprocessing of the text is done using the NLP preprocessing technique including Stop word removal and lowercasing.

3.3.2 Software tools

Python programming language is used due to its simplicity, versatility, and rich set of libraries for text processing. Additionally, the following libraries were used:

- Matplotlib: for visualizing performance metrics and algorithmic comparison
- Fitz: for working with pdf file
- Docx: for working with docx
- CSV: for working with CSV
- Nltk: preprocessing library for NLP
- Pandas and numpy for data handling

```
❖ import time
❖ import re
❖ import memory_profiler
❖ import pandas as pd
❖ import numpy as np
❖ import matplotlib.pyplot as plt
❖ from memory_profiler import memory_usage
❖ import fitz  # PyMuPDF
❖ import docx
❖ import csv
❖ import os
❖ import nltk
❖ from nltk.corpus import stopwords
❖ from nltk.tokenize import word_tokenize
❖ nltk.download('stopwords')
❖ nltk.download('punkt')
```

Figure 7Libraries Imported

3.3.3 Data Preprocessing

Before applying the Algorithmic implementation, the input data underwent preprocessing to ensure compatibility and optimize the performance.

Preprocessing First step includes checking up the file types. This project is working mainly on four file types all are preprocessed and converted into text files.

```
def convert_to_text(file_path):
    _, file_extension = os.path.splitext(file_path)
    text = ""

    if file_extension.lower() == ".pdf":
```

```python
        with fitz.open(file_path) as pdf:
            for page in pdf:
                text += page.get_text()

    elif file_extension.lower() == ".docx":
        doc = docx.Document(file_path)
        for paragraph in doc.paragraphs:
            text += paragraph.text + "\n"

    elif file_extension.lower() == ".txt":
        with open(file_path, "r", encoding="utf-8") as txt_file:
            text = txt_file.read()

    elif file_extension.lower() == ".csv":
        with open(file_path, "r", encoding="utf-8") as csv_file:
            csv_reader = csv.reader(csv_file)
            for row in csv_reader:
                text += ",".join(row) + "\n"

    else:
        print("Unsupported file format:", file_extension)
        return None

    return text.strip()
```

Figure 8:Convert to text

In the second step, the text extracted from the files is preprocessed using the NLP preprocessing techniques which are tokenization to make it easier to preprocess, stop word removal by removing the stop words there will be less matching can, which will help in improving the efficiency and Lower casing while working with exact matching algorithm it was discovered that they are case sensitive as well which will effect there accuracy if there is one word in different case in the pattern or text so both the patterns and text is lowercased before matching.

```python
def preprocess_text(text):
    text=re.sub(r'[^\w\s]', '', text)

    tokens = word_tokenize(text)



    # Remove stopwords
    stop_words = set(stopwords.words('english'))
    filtered_tokens = [token.lower() for token in tokens if token not in
stop_words]

    # Join tokens back into a string
    preprocessed_text = " ".join(filtered_tokens)
    return preprocessed_text
```
Figure 9: preprocessing text function

The third step is batch preprocessing, if there is too much data the algorithm is likely to struggle as all the algorithm efficiency depends on the length of the pattern and text. So, to overcome this problem the text can be divided into batches so that the system should not crash during the processing/searching.

```
def batch_text(text, batch_size):
    # Split text into batches of specified size
    batches = [text[i:i+batch_size] for i in range(0, len(text), batch_size)]
```
Figure 10: batch processing for large files

3.3.4 Algorithmic Implementations

3.3.4.1 Knuth-Morris-Pratt (KMP) Algorithm

The KMP algorithm was implemented according to the principles outlined in the literature review. We followed the steps and pattern matching and utilized the optimized techniques proposed to enhance efficiency.

The KMP algorithm avoids unnecessary character comparisons by utilizing a precomputed prefix function, which helps to determine the optimal shift when a mismatch occurs between the pattern and the text.

The compute prefix function computes the prefix of the given pattern, it initializes an array ‘prefix’ to store the prefix values for each position in the pattern, then iterates through the pattern, updating the prefix values based on the longest proper suffix that is also a prefix of the substring ending at the current position. Value ’j’ keeps track of the length of the longest proper prefix-suffix match and then returns the prefix array after the iterations are complete.

```
def compute_prefix_function(pattern):
    prefix = [0] * len(pattern)
    j = 0
    for i in range(1, len(pattern)):
        while j > 0 and pattern[j] != pattern[i]:
            j = prefix[j - 1]
        if pattern[j] == pattern[i]:
            j += 1
        prefix[i] = j
    return prefix
def kmp_string_match(text, pattern):
    prefix = compute_prefix_function(pattern)
    j = 0
    positions = []  # to store the positions of the pattern
    for i in range(len(text)):
        while j > 0 and text[i] != pattern[j]:
            j = prefix[j - 1]
        if text[i] == pattern[j]:
            j += 1
        if j == len(pattern):
            positions.append(i - (j - 1))  # add the position to the list
            j = prefix[j - 1]  # prepare for the next match
    return positions  # return the list of positions
```
Figure 11: KMP Algorithm Code snippet

The KMP string match function performs string matching using the KMP algorithm, it computes the prefix function for the pattern. It initializes ‘j’ and ‘positions’, where ‘j’ represents the pointer pointing to the current matching position and ‘positions’ stores the pointers of the pattern occurrences in the text. It iterates through text, updating ‘j’ based on the prefix function and comparing characters between the text and pattern. When a complete match is found, it adds the starting position of the match to the ‘positions’ list. The function returns the list of positions where a pattern occurs in the text.

3.3.4.2 Boyer-Moore Algorithm (BM):

As most of the variation does improve the time of the BM algorithm in the literature review but the good suffix heuristics and bad character heuristics maintain the accuracy of the Algorithm so by considering that there is not much difference in the original and the variations in terms of accuracy. The original principle of the BM algorithm is used.

The provided implementation of the Boyer-Moore Algorithm to efficiently search for the occurrence of patterns within the text. The Algorithm achieves its efficiency through two preprocessing steps: constructing the bad character table and the good suffix table. These tables enable the algorithm to determine optimal shifts when mismatches occur during the string-matching process.

```
def preprocess_strong_suffix(shift, bpos, pat, m):
    i = m
    j = m + 1
    bpos[i] = j

    while i > 0:
        while j <= m and pat[i - 1] != pat[j - 1]:
            if shift[j] == 0:
                shift[j] = j - i
            j = bpos[j]
        i -= 1
        j -= 1
        bpos[i] = j

def preprocess_case2(shift, bpos, m):
    j = bpos[0]
    for i in range(m + 1):
        if shift[i] == 0:
            shift[i] = j
        if i == j:
            j = bpos[j]

def bm_string_match(text, pattern):
    matches = []
    s = 0
    m = len(pattern)
    n = len(text)
    bpos = [0] * (m + 1)
    shift = [0] * (m + 1)

    preprocess_strong_suffix(shift, bpos, pattern, m)
    preprocess_case2(shift, bpos, m)

    while s <= n - m:
        j = m - 1
        while j >= 0 and pattern[j] == text[s + j]:
            j -= 1
        if j < 0:
            matches.append(s)
            s += shift[0]
        else:
            s += shift[j + 1]

    return matches
```
Figure 12: BM algorithm code snippet

This preprocess_strong_suffix function preprocesses the strong suffix table and calculates the position of potential suffixes in the pattern. It initializes variables ‘i’ and ‘j’, where ‘i’ represents the current position in the pattern, and ‘j’ represents the position of a potential suffix and iterates through the pattern right to left, updating the ‘bpos’ array to store the position of potential suffixes. The position also computes the shift values for the mismatches encountered during string matching and updates the shift array accordingly

Preprocess_case2 function preprocess the case2 table, which is used to determine the shift values for the specific mismatch scenarios. It initializes ‘j’ to the position of the last potential suffix in the pattern and also iterates through the pattern, updating the shift values in the ‘shift’ array based on the position of potential suffixes stored in the ‘bpos’ array.

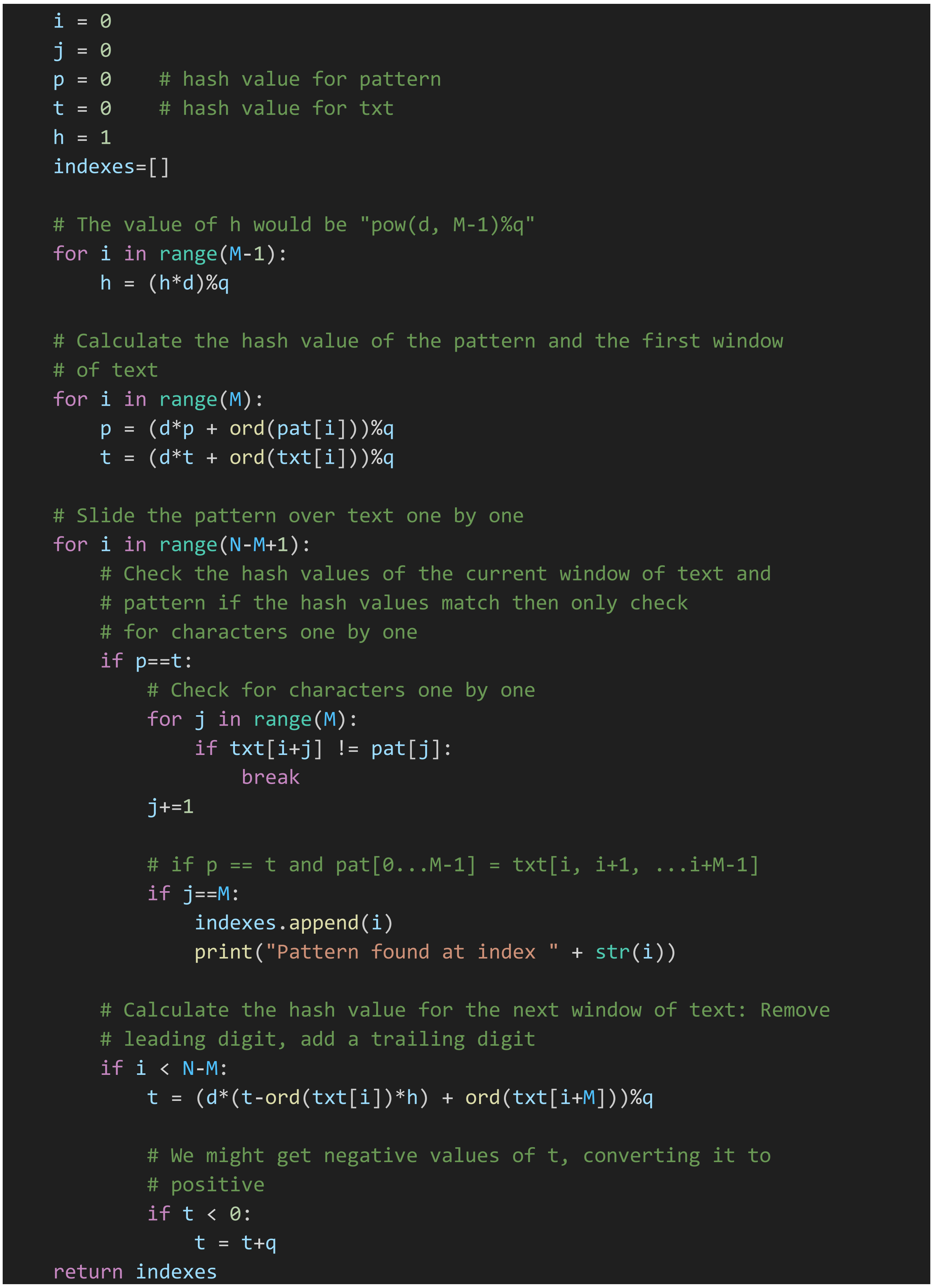
Bm_string_match function is the main function that performs the string-matching using the BM algorithm. It initializes variables ‘matches’ (list of match indexes),’s’(shift), ‘m’ (pattern length), and n (text length).it constructs the strong suffix and case2 tables using the preprocessing functions and iterates through the text, comparing substrings of the text with pattern and adjust the shift value accordingly. When the match is found it will be appended to the ‘matches’ list.

3.3.4.3 Rabin Karp Algorithm

Rabin Karp algorithm utilizes hashing to efficiently search for the occurrence of the pattern within a text. Comparing hash values rather than the individual characters which filers the potential matches and reduces the number of character comparisons required during the string-matching process.

The provided implementation is a Python function rabinkarp_string_match that utilizes the Rabin-Karp algorithm to efficiently search for occurrences of a pattern within a text. The Rabin-Karp algorithm achieves its efficiency by hashing the pattern and sliding a window over the text, comparing hash values to determine potential matches.

```
d=250
def rabinkarp_string_match(pat, txt, q=101):
    M = len(pat)
    N = len(txt)
```



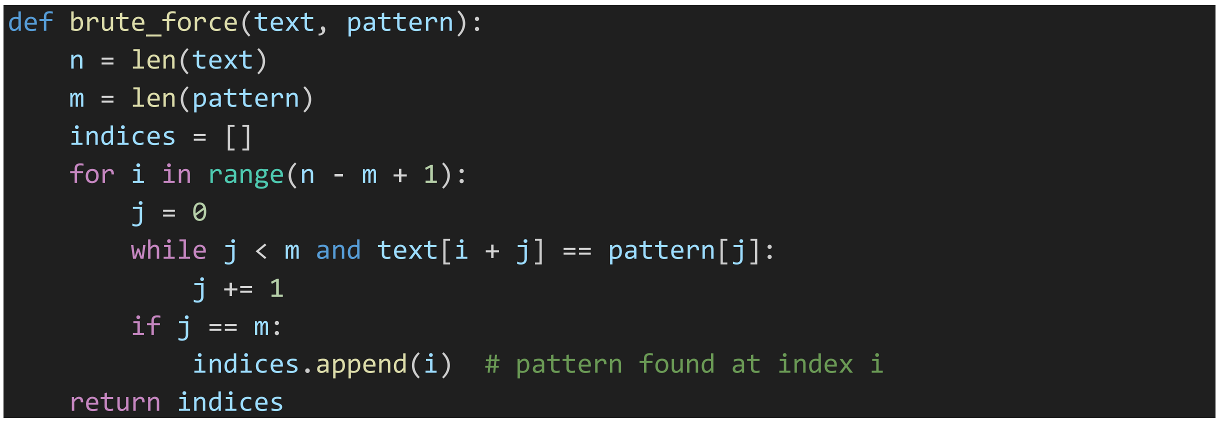
```
    i = 0
    j = 0
    p = 0    # hash value for pattern
    t = 0    # hash value for txt
    h = 1
    indexes=[]

    # The value of h would be "pow(d, M-1)%q"
    for i in range(M-1):
        h = (h*d)%q

    # Calculate the hash value of the pattern and the first window
    # of text
    for i in range(M):
        p = (d*p + ord(pat[i]))%q
        t = (d*t + ord(txt[i]))%q

    # Slide the pattern over text one by one
    for i in range(N-M+1):
        # Check the hash values of the current window of text and
        # pattern if the hash values match then only check
        # for characters one by one
        if p==t:
            # Check for characters one by one
            for j in range(M):
                if txt[i+j] != pat[j]:
                    break
            j+=1

            # if p == t and pat[0...M-1] = txt[i, i+1, ...i+M-1]
            if j==M:
                indexes.append(i)
                print("Pattern found at index " + str(i))

        # Calculate the hash value for the next window of text: Remove
        # leading digit, add a trailing digit
        if i < N-M:
            t = (d*(t-ord(txt[i])*h) + ord(txt[i+M]))%q

            # We might get negative values of t, converting it to
            # positive
            if t < 0:
                t = t+q
    return indexes
```

Figure 13: Rabin Karp algorithm code snippet

rabinkarp_string_match function performs string matching using the Rabin Karp algorithm. It initializes variables ‘M’ (pattern length),’N’ (text length),’p’ (hash value for the pattern),’t’ (hash value for the text), ‘h’ (initial hash value) and indexes (list of match indexes).it calculates the initial hash values for the pattern and the first window of the text, then slides the pattern over the text one character at a time, recalculating the hash value for each window of text. If the hash value of the pattern and current window of the text matches, it performs a character -by- character comparison to confirm the match. If the match is confirmed it appends the starting index of the match to 'indexes' list and return the list after each iteration.

3.3.4.4 Brute Force Algorithm

The brute force algorithm is a straightforward approach to string matching. It systematically compares each string of the text with a pattern, character by character, to determine potential matches. While not as efficient as the other advances string matching algorithm brute force is easy to append and suitable for small texts or patterns.



```python
def brute_force(text, pattern):
    n = len(text)
    m = len(pattern)
    indices = []
    for i in range(n - m + 1):
        j = 0
        while j < m and text[i + j] == pattern[j]:
            j += 1
        if j == m:
            indices.append(i)  # pattern found at index i
    return indices
```

Figure 14 Brute force Algorithm code snippet

This function performs string matching using a brute force algorithm. Initialize variables 'n' text length and 'm' pattern length and 'indices' list of match indexes. It iterates through the text, comparing the substring of the length 'm' with the pattern. For each substring, it compares each character with the corresponding character in the pattern. If a complete match is found it appends the index of the string position of the 'indices' list. Then function returns the list of match indexes.

3.3.4.5 User Friendly Interface (Tkinter):

The provided code implements a graphical user interface using TKInter in Python for performing string-matching operations. The Key components are as follows:

- File Selection: Allow user to browse and select a file from their system.
- Pattern Entry: Provides a text entry field for the user to input the pattern they want to search for within the selected file
- Algorithm Selection: Offers a dropdown menu for users to choose the string-matching algorithm they want to use (KMP, Boyer Moore, Rabin Karp, or Brute Force)
- Preprocessing checkbox: This enables the user to choose whether they want to preprocess the text and pattern before performing the matching operation.
- Submit button: to submit the user input to calculate and evaluate the output
- Output Text Area: Displays the positions where the pattern was found in the file, or notifies the user if the pattern was not found.

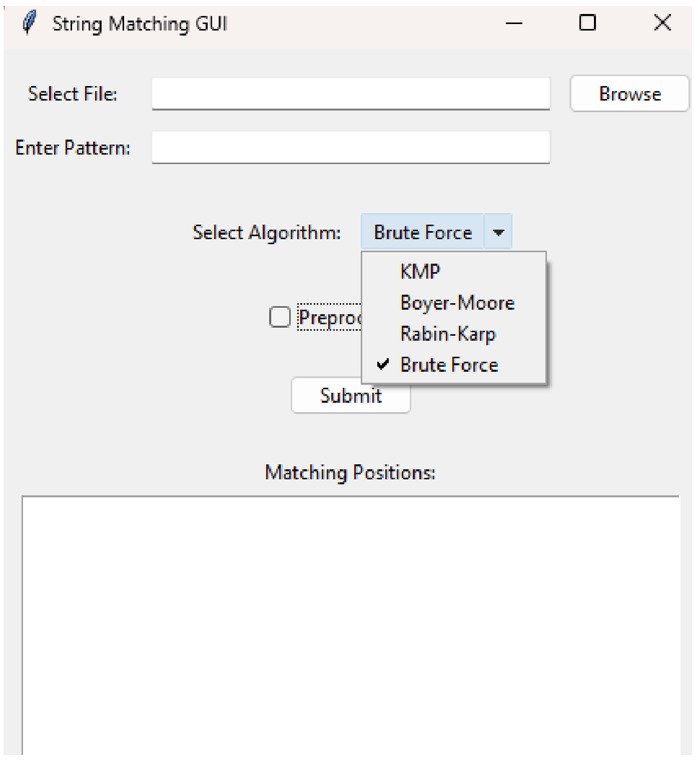


Figure 15 Exact string matching Gui

The above figure shows the output display of the GUI code.

4 Evaluation

4.1 Evaluation Methodology

4.1.1 Evaluation Metrics

The evaluation of the implemented algorithms will be conducted based on the following metrics:

- ❖ Accuracy: Measure the proportion of correctly identified patterns
- ❖ Time Complexity: Measure the computational efficiency of the algorithms
- ❖ Space complexity: Measure the memory usage of algorithms

As these are the Predefined Exact matching algorithms with 100% precision, precision score cannot be tested as it will not have false outputs, whereas the Accuracy score can be tested as they tend to miss some of the patterns if there are certain changes in characters like Uppercased.

4.1.2 Baseline systems

The evaluation compares the performance of implemented string matching algorithms (KMP vs BM vs RK vs Brute Force). These baseline algorithms will be compared between pre-processed text and without pre-processed text for the same files. These algorithms are commonly used algorithms in the real world Exact matching scenarios like information retrieval, data mining, etc.

4.1.3 Dataset

The evaluation utilizes diverse datasets sourced from various domains including text documents, PDFs, and CSV files. While the datasets are small in number, they vary in size and complexity, ensuring a comprehensive evaluation of the string matching system's performance across different scenarios. The datasets are carefully selected to represent real-world scenarios and provide reliable consistency in comparative analysis.

The data set collected is then turned into a CSV file of three columns File (contains file path), Pattern(contains patterns to be compared), and Expected number of outcomes (Count of number of patterns expected to be found in the text to get the 100% Accuracy). Then the count of the number of the pattern found is taken to compare the result, Time taken and memory usage are calculated for both pre-processed text and without preprocessing.

The information about Dataset used for comparison is as follows:

Table 1: Dataset used for Comparison

File Index	File Type	File Size(MB)	No. of Stop Words	No. of Punctuations
1	.txt	385.9893798828125	135709899	12211500
2	.txt	442.3799514770508	0	0
3	.pdf	46.00362777709961	267400	280200
4	.docx	63.55571746826172	573700	442500
5	.docx	1464.4465446472168	43300	60500
6	.pdf	175.37479400634766	116199	63200
7	.docx	1.6242027282714844	665600	17700
8	.pdf	82.122802734375	81599	62000
9	.pdf	82.122802734375	22998601	62000
10	.docx	7.677745819091797	0	22100
11	.txt	235.88180541992188	8879299	15487200
12	.txt	28.93962860107422	0	1422200

13	.txt	67.39702224731445	0	2440600
14	.txt	125.14152526855469	0	6063000
15	.txt	43.68324279785156	0	2408700
16	.pdf	53.89509201049805	0	2587400
17	.txt	146.7411994934082	0	5986700
18	.txt	27.46114730834961	0	1010600
19	.docx	276.59912109375	0	0
20	.pdf	42.79890060424805	0	0
21	.txt	42.79890060424805	0	0
22	.txt	314.3073081970215	0	0
23	.txt	48.59151840209961	0	0
24	.txt	48.59151840209961	0	0
25	.txt	48.59151840209961	0	0

4.2 Results and Discussion:

There are 25 files on which these four algorithms Accuracy, Time complexity, and memory complexity are measured.

The memory complexity of the 4 algorithms came to 0 so there was no memory used by the algorithm itself.

By going through the memory usage for the algorithm. It was found that the memory usage done by the algorithm itself for searching patterns and for preprocessing the text is negligible.

This can also be observed in the theoretical where the space complexity of the algorithm depends on the length of the pattern or whether it is constant.

The space complexity of the algorithms is as follows:

KMP Algorithm The space complexity of the KMP algorithm is O(m), where m is the length of the pattern string. This space is required for storing the longest prefix suffix array(LPS)used in the pattern-matching process.

Rabin Karp Algorithm: The space complexity of the Rabin Karp algorithm is O(1). It only requires a constant amount of space for storing variables used in hashing, such as the hash value of the pattern and the hash value of the current window.

Brute force algorithm: The space complexity of the Brute force algorithm is also O(1). It only requires space for the pointers and temporary variables.

Boyer Moore algorithm: The space complexity of the Boyer Moore algorithm is O(m+|a|), where m is the length of the pattern and a is the size of the alphabet.

4.2.1 Accuracy score

Talking about the accuracy the accuracy score is measured from 0 to 1 for the pattern. For these four exact matching patterns, the accuracy score follows a similar trend for both preprocessed text and non-preprocessed text.

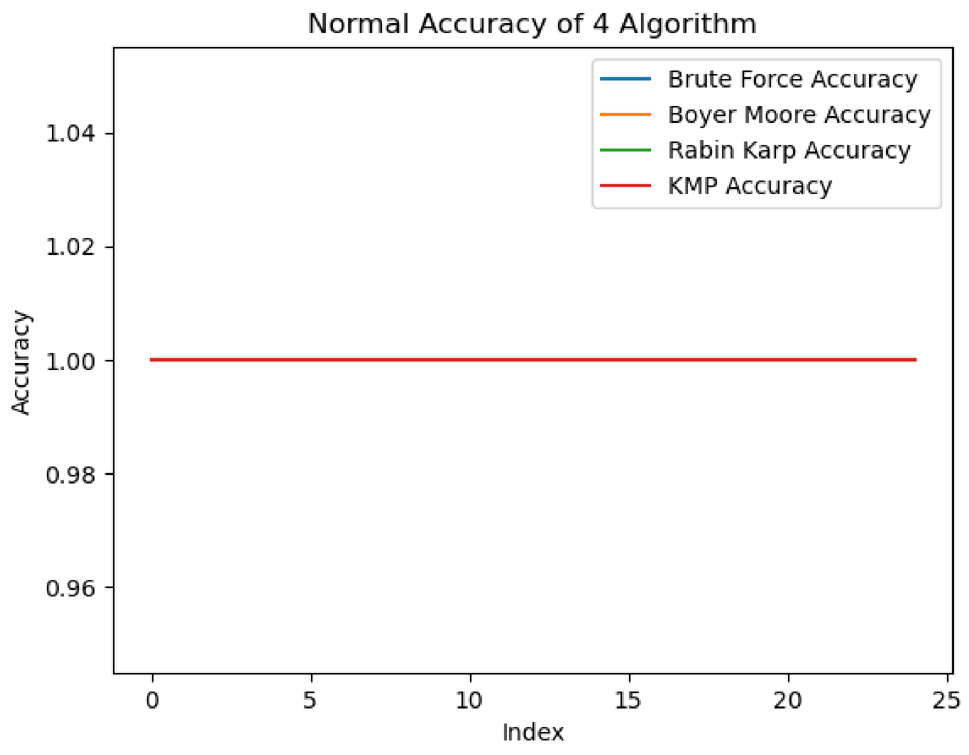


Figure 16: Without preprocessing accuracy

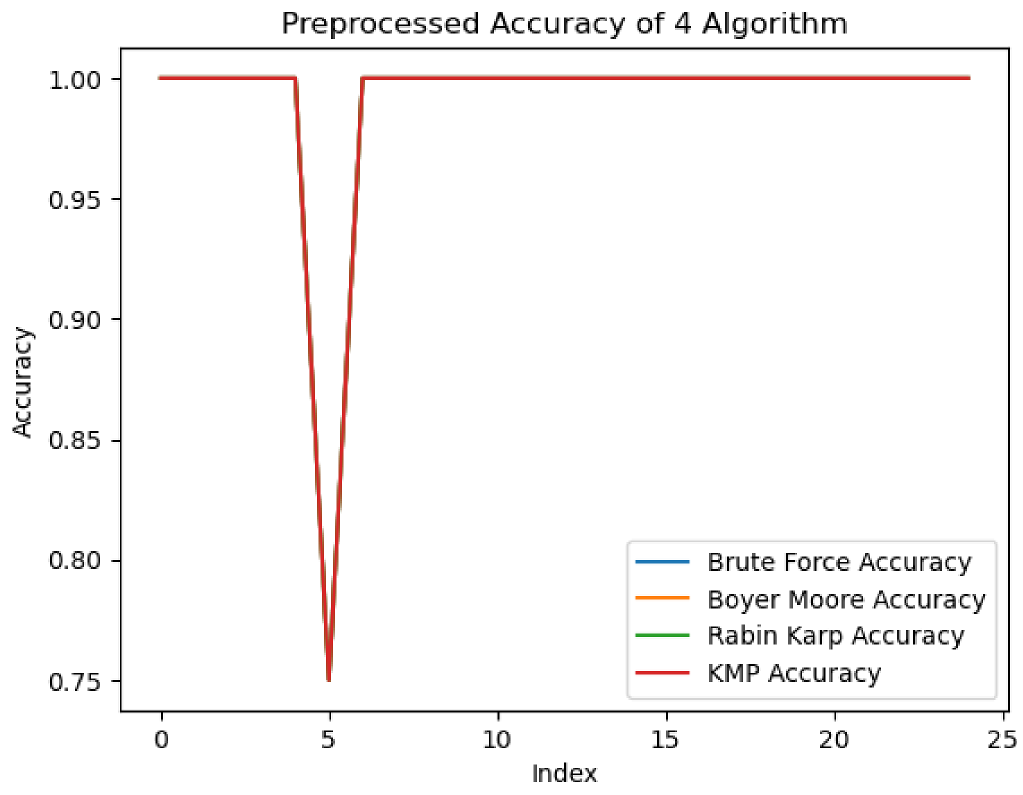


Figure 17: Pre-processed text Accuracy of 4 Algorithm

The sudden downfall was seen for the preprocessed accuracy for the file index 5 rather than that all the Accuracy scores for both files are 100%.

To visually represent these findings, a line graph was constructed to illustrate the accuracy scores of all four algorithms across different scenarios. This consolidated visualization provided a comprehensive overview of the algorithms' performance, highlighting notable accuracies and pinpointing areas of improvement, particularly in preprocessing strategies to accommodate special characters effectively.

The accuracy assessment of text-matching algorithms, including the Boyer-Moore (BM), Knuth-Morris-Pratt (KMP), Rabin-Karp, and Brute Force, revealed intriguing results. Across the board, the normal text matching displayed impeccable accuracy, registering a flawless 100% score across all algorithms. This outcome underscored the robustness and reliability of these renowned algorithms in their conventional form.

However, by introducing preprocessed text patterns into the evaluation. Surprisingly, the accuracy scores remained consistently 100%, except for a singular instance observed in the file index 5. Upon closer examination of the pattern and text used in this particular file, it was identified that the pattern contained special characters such as 'g (&', which might have undergone modification during the preprocessing stage. This could have occurred due to punctuation removal techniques employed during preprocessing. Due to using the NLP techniques the Preprocessing for Literature Files has improved and efficient whereas other fields of files can be improved more using the different preprocessing techniques

4.2.2 Time Complexity:

4.2.2.1 BM Algorithm:

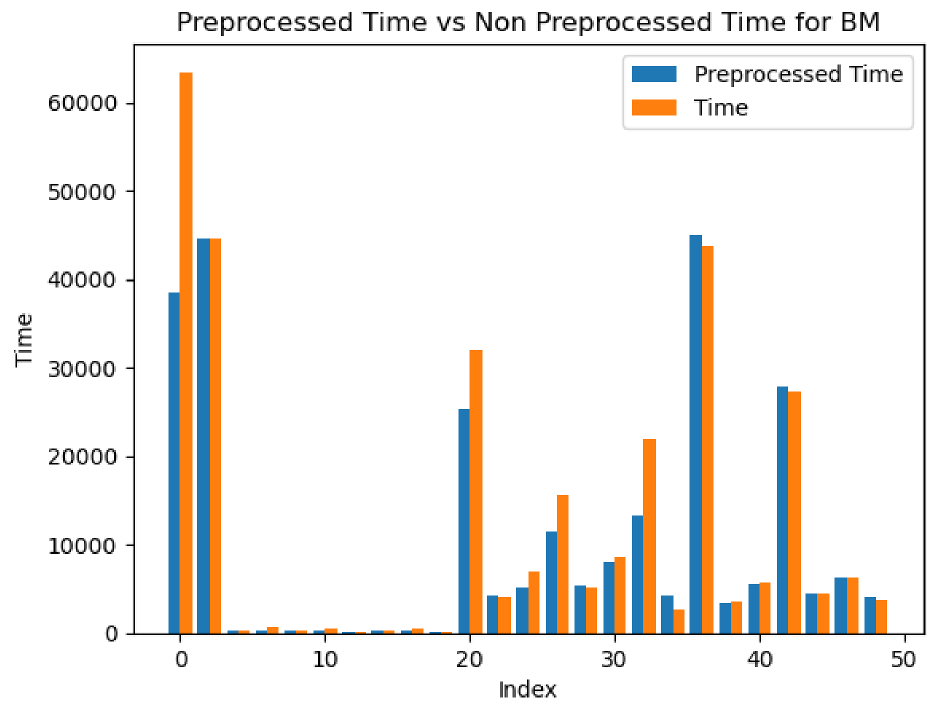


Figure 18: Pre-processed vs Non-pre-processed Text Time for Boyer-Moore Algorithm

The result shows a certain trend in the Time complexity of the BM Algorithm which is represented in the bar graph where blue bars show the BM searching time for Preprocessed Text and BM searching time for non-preprocessed text.

- Boyer Moore algorithm: The Bar graph showing the Time comparison for the Preprocessed and Non-Preprocessed text and pattern shows some good explanations for the theoretical equations.
  The time complexity of the Boyer Moore algorithm theoretically the Time complexity of the Boyer Moore is O (n * |a|) where n is the length of the text and |a| is the size of the alphabet. As can be seen in the graph the orange bar represents the Time for the Boyer Moore string searching algorithm.
  But applying the preprocessing text instead of the principle which removes stop words and punctuation. Theoretically, if we preprocess the text and pattern the O notation should have changed to O((n-s-p) *|a|) where s is the no. of the stop word removed and p is the no. of punctuation removed from the text. So theoretically more of the text had the stop words or punctuation faster is the algorithm after preprocessing.

This can be shown in graph file 1 is an English literature file of the highest size and is expected to contain most stop words that is why the time complexity to search for the pattern is reduced than the time complexity of the pattern searched in the non-preprocessed file. The Blue bar shows the searching time in preprocessed text using NLP. The second file is a genome text file that typically contains the genetic sequence so it will not have the stop word or punctuation in it so typically it doesn't affect the time complexity that much. In the rest of the files, where also the trend can be observed that the file containing the stop words or the punctuation is likely to work faster for searching when preprocessed.

4.2.2.2 KMP Algorithm:

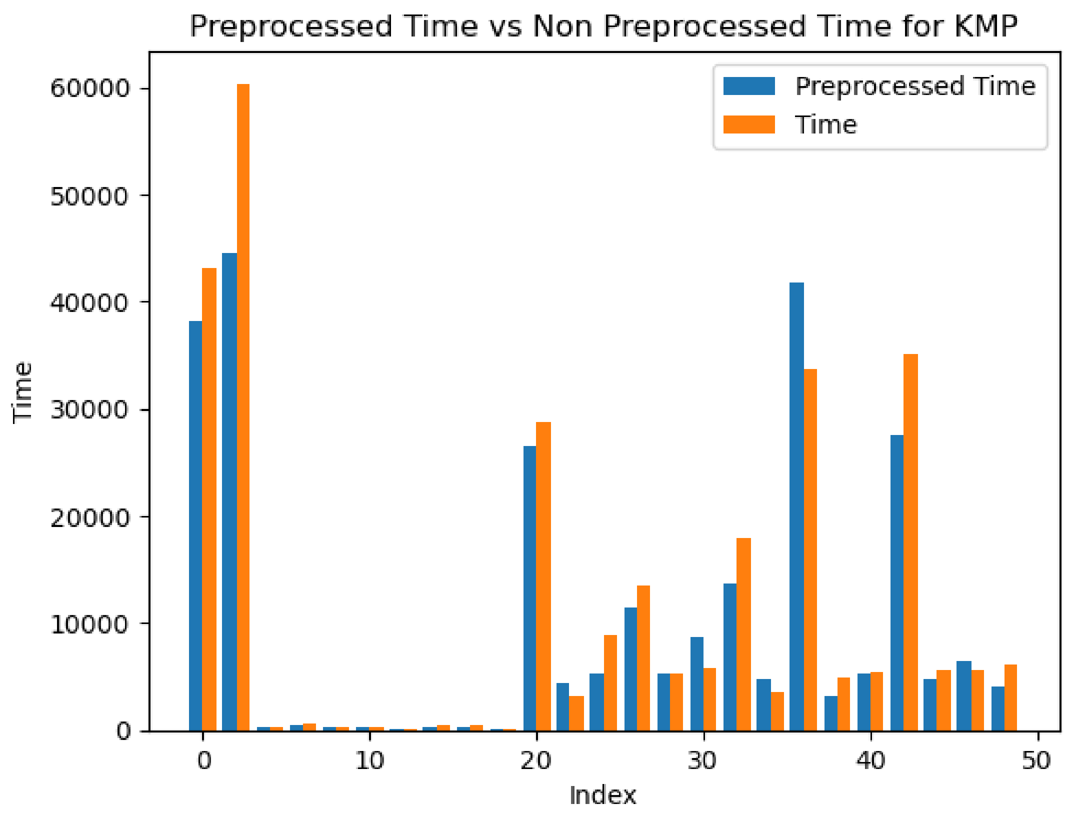


Figure 19: Pre-processed vs Non-Pre-processed Text Time for Knuth-Morris-Pratt Algorithm

For the KMP algorithm, the above bar graph shows the Time taken for the pattern searching where the blue bar shows the Time for Preprocessed text and pattern and the orange bar shows the time without preprocessing.

- KMP algorithm: The time complexity of the KMP algorithm is O(n+m) where n is the length of the text and m is the length of the pattern. This complexity arises from the linear time complexity of both the preprocessing step, which constructs the Longest Prefix Suffix array, and the pattern matching step.

Theoretically considering the preprocessing step O notation should be O(n-s-p+m) where n is the length of the text, s is the length of the stop words removed, p is the length of punctuation removed and m is the length of the pattern. while some fluctuation in the graph data shows better performance for preprocessed data in most of the pattern searching.

File 2 which doesn't have any stop word still shows a good result this might be because of tokenizing the data and rearranging it into one-line text rather than using new lines.

4.2.2.3 Rabin-Karp Algorithm

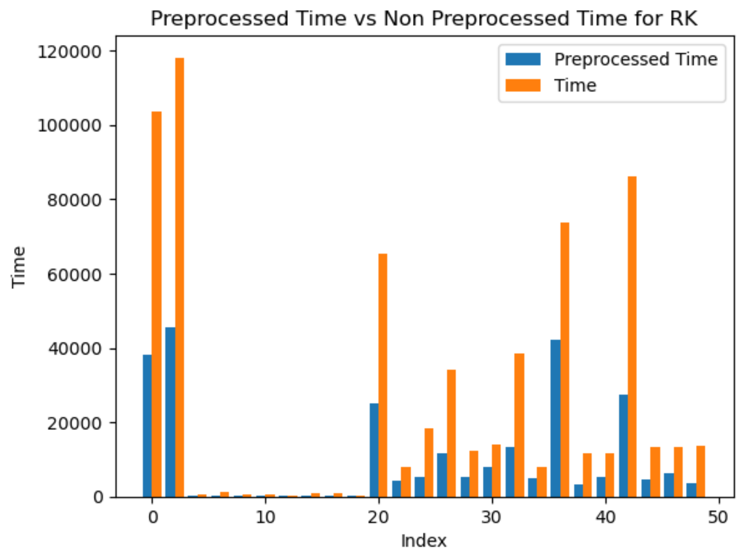


Figure 20: Pre-processed vs non pre-processed text Time for Rabin Karp algorithm

The above figure is the visual representation of the Time taken to search by Rabin Karp in both preprocessed and Non preprocessed Text and pattern.

- Rabin Karp algorithm: The Time complexity of the Rabin Karp algorithm is O((n-m+1) *m). This complexity arises from computing the hash value of each substring of length m in the text and comparing it with the hash value of the pattern.

This can be changed to O((n-s-p-m+1) *m) where s and p is the length of the removed stop words and punctuation. By removing the punctuation and stop words hash value calculation reduces which effects significantly the time complexity of the algorithm. Proving the theory is the practical result graph which shows the significant change in time taken to search for the pattern in the same trend

4.2.2.4 Brute Force Algorithm

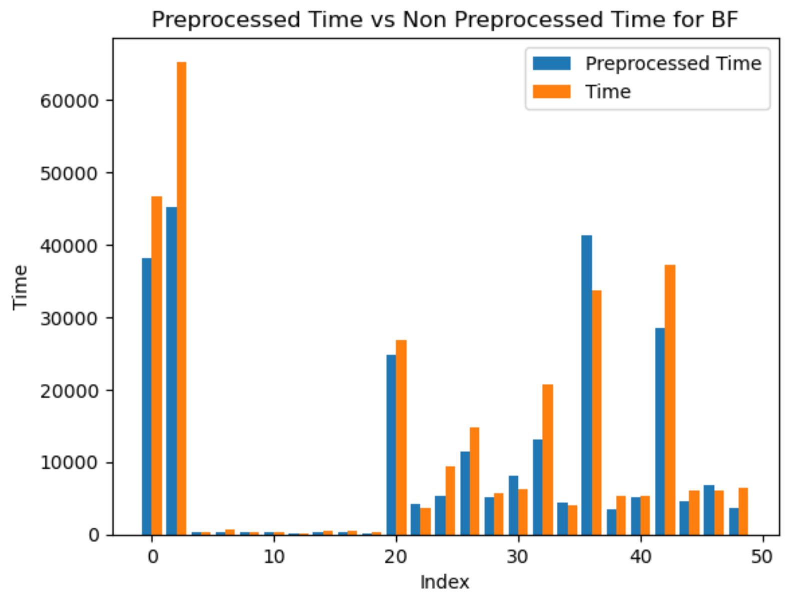


Figure 21: pre-processed vs Non-Pre-processed Text time for Brute Force algorithm

The above bar graph shows the result of preprocessed text searching and non-preprocessed searching by Brute force Algorithms.

- Brute force Algorithm: the theoretical time complexity of the Brute force algorithm is given as O((n-m+1) *m) where m is the length of the pattern and n is the length of the text. This complexity came from comparing each substring of length m in the text with the pattern. Which can be changed to O((n-s-p-m+1) *m) where s and p are the removed stop words and punctuation respectively.

Considering the above notations the large file will have good time complexity which can be seen in the graph above whereas as the size decreases for the files the preprocessing does not make much difference in most of the cases

4.2.2.5 Non-preprocessed text and pattern Search time comparison of all four Algorithms

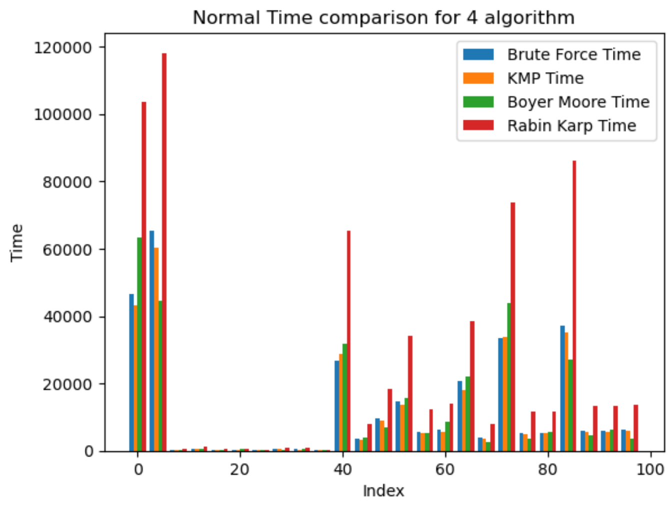


Figure 22: Bar graph Representation of Non-Pre-processed Text Time for All 4 Algorithms

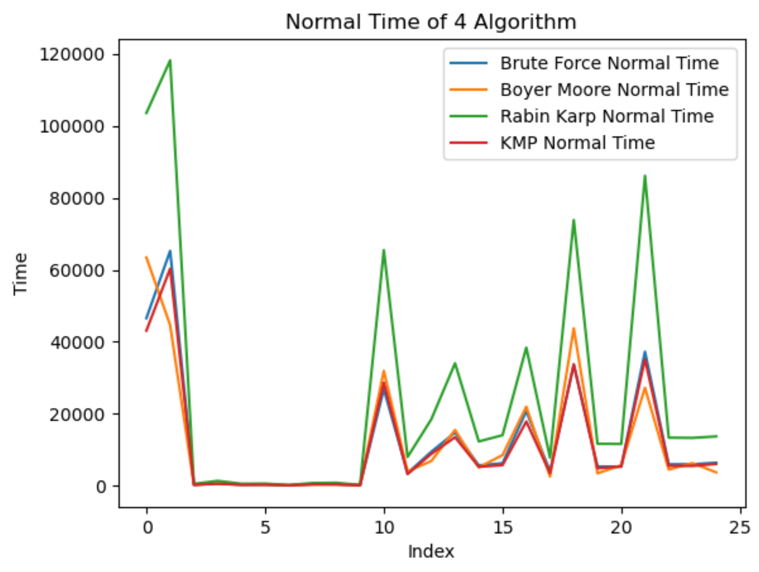


Figure 23: Line graph representation of non-pre-processed text for all 4 Algorithm

While Comparing Time the pattern matching algorithm for Not preprocesses text Shows that three algorithms worked efficiently for all files but Rabin Karp algorithms performed slowest in all of the cases. With a difference of more than 20000 ms in most of the cases.

This shows that KMP, Boyer Moore is more efficient in the case of text, PDF, Docx, and CSV files.

4.2.2.6 Preprocessed text and pattern Search time comparison of all four algorithms:

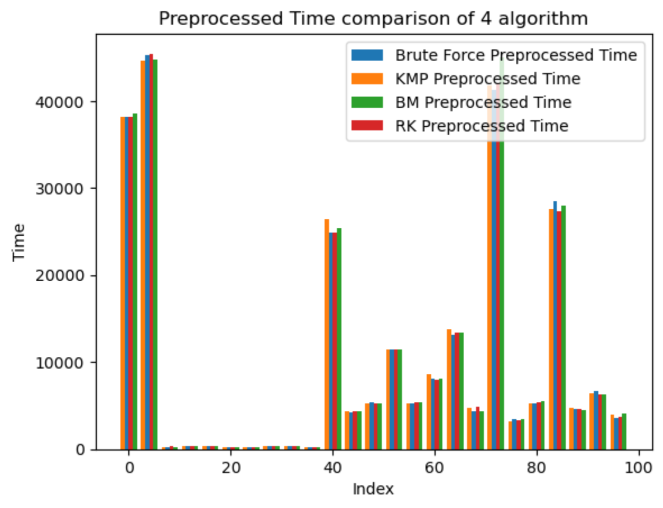


Figure 24: Preprocessed Text Time bar graph representation for all 4 algorithms

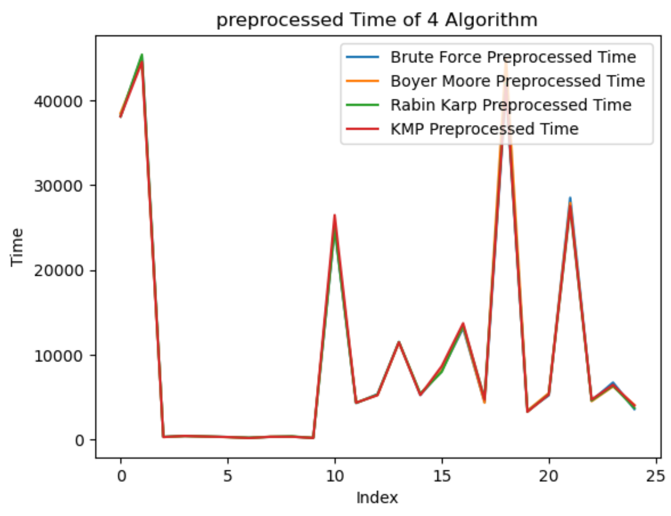


Figure 25: Line Graph representation for all 4 algorithm pre-processed time

While comparing Time with the pattern matching algorithm it was found for the preprocesses text all of the time has been almost the same with insignificant differences for all 25 files

4.2.2.7 Time taken to preprocess Text and pattern

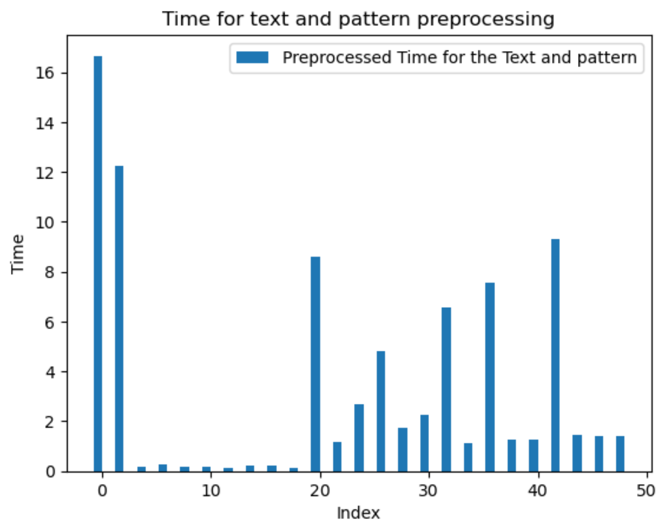


Figure 26: Time taken to preprocess the text

It takes negligible time to preprocess the data in comparison to the algorithm searching time.

5 Conclusions

In conclusion, the evaluation of the four-string matching algorithms, namely Boyer-Moore(BM), Knuth-Morris-Pratt(KMP), Rabin Karp, and brute force string matching algorithms yielded insightful findings regarding their accuracy and time complexity.

Initially, the accuracy assessment demonstrated flawless performance by all four algorithms when dealing with normal text, achieving a 100% accuracy score. However, the introduction of pre-processed text patterns resulted in a slight decrease in accuracy in one instance due to special character modification during preprocessing.

In terms of time complexity, theoretical consideration suggested that preprocessing the text could potentially enhance algorithm performance, particularly in this scenario involving extensive stop words or punctuation in the text. This can also be seen in the practical comparison of the results as the size of the data increased the preprocessing made a significant impact on the time of the searching algorithms.

Furthermore, the assessment of space complexity revealed negligible memory usage by algorithms themselves for pattern searching and preprocessing, with complexities dependent on the pattern length or remaining constant. Using the NLP preprocessing techniques improved the Algorithm efficiency in one way. There are preprocessing techniques that can be applied to algorithms that work better in other criteria like handling special characters.

Overall, these findings underscored the importance of carefully assessing text characteristics and task requirements when selecting and implementing string-matching algorithms and preprocessing techniques. The comparison displayed a significant result for the language-based text than in the genome-based text special character text due to text preprocessing methods are NLP text preprocessing which is more accurate in cleaning data that has natural language rather than special characters or Genome data.

The analysis serves as valuable groundwork for further exploration and optimization within realms of string matching and natural language processing, emphasizing the potential of these algorithms and the pivotal role of preprocessing in maximizing their effectiveness.

6 Recommendations for future work

The Impact of preprocessing techniques is evident from this study. However, many other preprocessing techniques could be explored. For instance, techniques like n-grams. Potentially improve the string matching. Additionally, the handling of special characters and their impact on accuracy can be discussed further for analysis. In this project, the preprocessing for the NLP type of files is done which is a language file because the preprocessing was related to preprocessing done in NLP.

Comparative Study could be conducted with other string-matching algorithms not covered in this study. This would provide a more comprehensive understanding of the strengths and weaknesses of different algorithms.

Recent advances in deep learning have shown promising results in many areas of natural language processing. Exploring how these techniques can be applied to string matching could be an interesting direction for future work.

With the advent of big data, the size of text data has grown exponentially. This poses a challenge for traditional string-matching algorithms. Implementing these algorithms in a parallel or distributed computing environment could significantly reduce their time complexity.

7 References

Camacho-Collados, J. and Pilehvar, M.T. (2017). On the role of text preprocessing in neural network architectures: An evaluation study on text categorization and sentiment analysis. *arXiv preprint arXiv*, 1707.01780.

Cantone, D. and Faro, S. (2003). Fast-Search: A new efficient variant of the Boyer-Moore string matching algorithm. *In Experimental and Efficient Algorithms: Second International Workshop, WEA 2003, Ascona, Switzerland, May 26–28, 2003 Proceedings 2*, (pp. 47-58). Springer Berlin Heidelberg.

Chai, C. (2023). Comparison of text preprocessing methods. *Natural Language Engineering*, 29(3), pp.509-553.

Chayapathi, A. (2021). Survey and comparison of string matching algorithms. *Turkish Journal of Computer and Mathematics Education (TURCOMAT)*, 12(12), pp.1471-1491.

Galil, Z. (1979). On improving the worst case running time of the Boyer-Moore string matching algorithm. *Communications of the ACM, 22(9)*, pp.505-508.

Glück, R. and Yokoyama, T. (2022). Reversible programming: a case study of two string-matching algorithms. *arXiv preprint arXiv*, 2211.12225.

Lecroq, T. (1992). A variation on the Boyer-Moore algorithm. *Theoretical Computer Science, 92(1)*, pp.119-144.

Leonardo, B. and Hansun, S. (2017). Text documents plagiarism detection using Rabin-Karp and Jaro-Winkler distance algorithms. *Indonesian Journal of Electrical Engineering and Computer Science*, 5(2), pp.462-471.

Lu, X. (2019, November). The analysis of KMP algorithm and its optimization. *In Journal of Physics: Conference Series*, (Vol. 1345, No. 4, p. 042005). IOP Publishing.

Saho, S. (2023). Data Validation and Reconciliation of Student Marks and Data. *ournal of Computer Science and Technology*.

Tarhio, J. and Ukkonen, E. (1993). Approximate boyer–moore string matching. *SIAM Journal on Computing*, 22(2), pp.243-260.

Vayadande, K., Mandhana, R., Paralkar, K., Pawal, D., Deshpande, S. and Sonkusale, V. (2022). Pattern matching in file system. *International Journal of Computer Applications*, 975, p.8887.

Vijayarani, S., Ilamathi, M.J. and Nithya, M. (2015). Preprocessing techniques for text mining-an overview. *International Journal of Computer Science & Communication Networks*, 5(1), pp.7-16.

Zhang, S., Hu, Y. and Bian, G. (2017). March. Research on string similarity algorithm based on Levenshtein Distance. *In 2017 IEEE 2nd Advanced Information Technology, Electronic and Automation Control Conference (IAEAC)* , (pp. 2247-2251).

Zhao, C. and Sahni, S. (2019). String correction using the Damerau-Levenshtein distance. *BMC bioinformatics*, 20, pp.1-28.

Zhou, Y. and Pang, R. (2019, December). Research of Pattern Matching Algorithm Based on KMP and BMHS2. *In 2019 IEEE 5th International Conference on Computer and Communications (ICCC)*, pp. 193-197.

8 Bibliography

GeeksforGeeks. (n.d.). String matching Algorithm: Applications of String Matching Algorithms. Retrieved from GeeksforGeeks.:- Applications of String Matching Algorithms - GeeksforGeeks

Wikipedia contributors. (n.d.). Boyer–Moore string-search algorithm. In Wikipedia, The Free Encyclopedia. Retrieved from Wikipedia.:- Boyer–Moore string-search algorithm - Wikipedia

Wikipedia contributors. (n.d.). Knuth–Morris–Pratt algorithm. In Wikipedia, The Free Encyclopedia. Retrieved from Wikipedia.:- Knuth–Morris–Pratt algorithm - Wikipedia

Wikipedia contributors. (n.d.). Rabin–Karp algorithm. In Wikipedia, The Free Encyclopedia. Retrieved from Wikipedia.:- Rabin–Karp algorithm - Wikipedia

Wikipedia contributors. (n.d.). Brute-force search. In Wikipedia, The Free Encyclopedia. Retrieved from Wikipedia.:- Brute-force search - Wikipedia

Wikipedia contributors. (n.d.). Natural language processing. In Wikipedia, The Free Encyclopedia. Retrieved from Wikipedia.:- Natural language processing - Wikipedia

IBM. (n.d.). What Is Natural Language Processing? Retrieved from IBM.:- What Is Natural Language Processing? | IBM

Duong, V. H. T. (n.d.). NLP Text Preprocessing: Steps, tools, and examples. Retrieved from Towards Data Science. NLP Text Preprocessing: Steps, tools, and examples | by Viet Hoang Tran Duong | Towards Data Science

9 Appendices

9.1 Code snippet for TkInter User Interface

This is the code snippet for the user-friendly interface created for the string-matching Algorithm user interactions.

```
import tkinter as tk
from tkinter import filedialog
from tkinter import ttk
from functools import partial
import os

class StringMatchingGUI:
    def __init__(self, root):
        self.root = root
        self.root.title("String Matching GUI")

        self.file_path = tk.StringVar()
        self.pattern = tk.StringVar()
        self.algorithm = tk.StringVar(value="KMP")
        self.preprocess = tk.BooleanVar(value=True)

        self.create_widgets()

    def create_widgets(self):
        # File Selection
        file_frame = ttk.Frame(self.root)
        file_frame.pack(pady=10)

        ttk.Label(file_frame, text="Select File:").grid(row=0, column=0,
padx=5, pady=5)
        self.file_entry = ttk.Entry(file_frame, textvariable=self.file_path,
width=40)
        self.file_entry.grid(row=0, column=1, padx=5, pady=5)
        ttk.Button(file_frame, text="Browse",
command=self.browse_file).grid(row=0, column=2, padx=5, pady=5)


        ttk.Label(file_frame, text="Enter Pattern:").grid(row=1, column=0,
padx=5, pady=5)
        self.pattern_entry = ttk.Entry(file_frame, textvariable=self.pattern,
width=40)
        self.pattern_entry.grid(row=1, column=1, padx=5, pady=5)

        # Algorithm Selection
        algorithm_frame = ttk.Frame(self.root)
        algorithm_frame.pack(pady=10)
        ttk.Label(algorithm_frame, text="Select Algorithm:").grid(row=0,
column=0, padx=5, pady=5)
        algorithm_options = ["KMP", "Boyer-Moore", "Rabin-Karp", "Brute
Force"]
        self.algorithm_menu = ttk.OptionMenu(algorithm_frame,
self.algorithm,algorithm_options[0], *algorithm_options)
        self.algorithm_menu.grid(row=0, column=1, padx=5, pady=5)

        # Preprocessing Checkbox
        preprocess_frame = ttk.Frame(self.root)
        preprocess_frame.pack(pady=10)

        self.preprocess_check = ttk.Checkbutton(preprocess_frame,
text="Preprocess Text", variable=self.preprocess)
        self.preprocess_check.grid(row=0, column=0, padx=5, pady=5)

        # Submit Button
        submit_button = ttk.Button(self.root, text="Submit",
command=self.process_matching)
        submit_button.pack(pady=10)

        # Output Text Area
        output_frame = ttk.Frame(self.root)
        output_frame.pack(pady=10)

        ttk.Label(output_frame, text="Matching Positions:").pack(pady=5)
        self.output_text = tk.Text(output_frame, width=50, height=10)
        self.output_text.pack()

    def browse_file(self):
        file_path = filedialog.askopenfilename(filetypes=[("All Files",
"*.*")])
        if file_path:
            self.file_path.set(file_path)

    def preprocess_text(self, text):
        text=re.sub(r'[^\w\s]', '', text)
        # Tokenize the text
        tokens = text.split()
        # Remove stopwords
        stop_words = set(stopwords.words('english'))
        filtered_tokens = [token.lower() for token in tokens if token.lower()
not in stop_words]
        # Join tokens back into a string
        preprocessed_text = " ".join(filtered_tokens)
        return preprocessed_text
    def process_matching(self):
        self.output_text.delete('1.0', tk.END)

        file_path = self.file_path.get()
        if not file_path or not os.path.exists(file_path):
            self.output_text.insert(tk.END, "Please select a valid file.")
            return

        pattern = self.pattern.get()
        if not pattern:
            self.output_text.insert(tk.END, "Please enter a pattern.")
            return

        algorithm = self.algorithm.get()


        text=convert_to_text(file_path)

        if self.preprocess.get():
            text = self.preprocess_text(text)
            pattern=pattern.lower()

        if algorithm == "KMP":
            matches = kmp_string_match(text, pattern)
        elif algorithm == "Boyer-Moore":
            matches = bm_string_match(text, pattern)
        elif algorithm == "Rabin-Karp":
            matches = rabinkarp_string_match(text, pattern)
        elif algorithm == "Brute Force":
            matches = brute_force(text, pattern)
        else:
            self.output_text.insert(tk.END, "Invalid algorithm selection.")
            return

        if matches:
            self.output_text.insert(tk.END, "Pattern found at positions:\n")
            for match in matches:
                self.output_text.insert(tk.END, f"{match}\n")
        else:
            self.output_text.insert(tk.END, "Pattern not found.")

if __name__ == "__main__":
    root = tk.Tk()
    app = StringMatchingGUI(root)
    root.mainloop()
```