A

SPEED TESTING REPORT

ON

“speed of the different options benchmarked on both reread query options.”

DEVELOPED BY

NWAMAH IKECHUKWU ARINZE

SUBMITTED TO

ALGORITHMIC SCIENCES

In partial fulfillment for the role of a python developer at algorithmic sciences

DATE

28 May 2024

ACKNOWLEDGEMENTS

I express profound gratitude to Divine Providence for bestowing upon me abundant grace and wisdom, essential for navigation through this pivotal phase of the interview process. While acknowledging the challenges encountered, I attribute the successful traversal of this stage to unwavering determination, perseverance, patience, consistency, diligence, and the application of my pertinent skill set. This achievement stands as a testament to the guiding hand of providence.

I extend heartfelt appreciation to the esteemed leadership of Algorithmic Science for granting me the opportunity to be considered for this position. Furthermore, I extend my sincere gratitude to all personnel, associates, stakeholders, and contributors associated with Algorithmic Sciences whose collective effort contribute to the organization’s success.

ABSTRACT

This technical report presents a comprehensive documentation of the speed analysis conducted on various search algorithms benchmarked against each other.

Chapter One introduces the purpose and significance of the speed report analysis, providing an overview of the search algorithms integrated into the project codebase.

Chapter Two delves into the methodologies employed for the speed report analysis. It discusses the algorithms tested in a well detailed format including addition of code screenshot

Chapter Three presents the results of the performance benchmarking exercise, meticulously tabulated and visually depicted through charts for clarity and easy interpretation.

Chapter Four provides a thorough analysis of the findings, followed by a succinct summary of the documented results. Moreover, recommendations for the most suitable search algorithm based on the analysis are provided.

Finally, an exhaustive list of reference to all materials utilized throughout the analysis is appended at the conclusion of the report.

Keywords: search algorithm, speed, benchmarking, layout design, methodologies, performance analysis, recommendations, reference materials.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

List of Charts

CHAPTER ONE

INTRODUCTION

Before we delve into the main focus of this report, it is best we first make a brief discussion on what speed report is and why it is required especially for a client/server communication, lastly for this introductory aspect, we will carefully consider in details the various search algorithm options. Speed report analysis is regarded as a detailed document that assesses and compares the performance of various processes or algorithms in terms of their execution speed. It typically includes empirical data derived from benchmarks or tests, illustrating how quickly each process or algorithm completes a given task under similar conditions.

1.1 Purpose of the Report: The objective of this report is simply to evaluate and benchmark the performance of various file-search algorithms that was successfully implemented during the development of the client/server communication system. File-search operators are fundamental in numerous applications, ranging from text editors to large-scale data processing systems. Given that speed is a critical aspect of these operations, getting a thorough understanding of which algorithms perform best under different conditions is very necessary.

1.2 Importance of Speed in File Search: In the realm of data processing and information retrieval, the efficiency of file-search algorithms can significantly impact overall system performance. Fast and efficient search algorithms reduce latency, improve user experience, and optimize resource utilization. In scenarios where large volumes of data are involved, the choice of the search algorithm can be the difference between a responsive system and one that is sluggish and inefficient.

1.3 Overview of The Speed Of Search Algorithm: The speed of a search algorithm refers to how quickly it can find a solution or locate a file or an item within a data set or search space, It's a crucial aspect of algorithmic efficiency, especially in fields like computer science and information retrieval, where the speed of searching can greatly impact overall system performance. Several factors influence the speed of a search algorithm, these factors are:

- **Time Complexity:** This is a measure of the amount of time an algorithm takes to complete as a function of the length of the input. Common notations used to describe time complexity include Big O notation (O()), which provides an upper bound on the time required for an algorithm to run.

- **Space Complexity:** This measures the amount of memory an algorithm uses relative to the size of the input data. Efficient algorithms tend to use minimal memory, especially for large datasets, to conserve resources and improve performance.

- **File Search Options:** Provided by operating systems or file management software, these options enable users to locate files based on criteria like name, type, date, modified, or content. Examples include Windows File Explorer, MacOS Finder, and command-line tools like find and grep.

- **Search Space:** The size of the search space impacts the speed of a search algorithm. A larger search space typically requires more time to find a solution compared to a smaller one.

- **Algorithm Design**: The design of the algorithm itself greatly influences its speed. Some algorithms are inherently faster for certain types of search problems due to their design and optimization techniques.

- **Data Structures:** The choice of data structures used by the algorithm can significantly affect its speed. Efficient data structures, such as hash tables or balanced trees, can speed up searches by allowing for fast retrieval and manipulation of data.

Heuristics: In some cases, search algorithms may employ heuristics or rules of thumb to guide the search process, especially in large or complex search spaces. Well-designed heuristics can help prune the search space and focus the algorithm’s efforts on the most promising paths, leading for faster search times.

Overall, the speed of a search algorithm is a complex interplay of various factors, including algorithmic efficiency, data structures, and problem-specific considerations. By understanding these factors and employing techniques to optimize performance, developers can design search algorithms that deliver fast and effective results for a wide range of applications.

CHAPTER TWO

METHODOLOGY

The next stage of this analysis is to have a clearer understanding of the methodology of a search algorithm. In computer science, a search algorithm is designed to solve search problem. Search algorithms work to retrieve information stored within particular data structure, or calculated in the search space of a problem domain, with either discrete or continuous values. Although search engines use search algorithms, they belong to the study of information retrieval, not algorithmics. The appropriate search algorithm to use often depends on the data structure being searched, and may also include prior knowledge about the data. Search algorithms can be made faster or more efficient by specially constructed database structures, such as search trees, hash maps, and database indexes. Search algorithms can be classified based on their mechanism of searching into three types of algorithms: linear, binary, and hashing. Linear search algorithms check every record for the one associated with a target key in a linear fashion. Binary, or half-interval, searches repeatedly target the center of the search structure and divide the search space in half. Comparison search algorithms improve on linear searching by successively eliminating records based on comparisons of the keys until the target record is found, and can be applied on data structures with a defined order. Digital search algorithms work based on the properties of digits in data structures by using numerical keys. Finally, hashing directly maps keys to records based on a hash function. Algorithms are often evaluated by their computational complexity, or maximum theoretical run time. Binary search functions, for example, have a maximum complexity of $O(\log n)$, or logarithmic time. In simple terms, the maximum number of operations needed to find the search target is a logarithmic function of the size of the search space.

2.1 Description of Algorithm Tested:

This report specifically benchmarks seven distinct search algorithms that was duly tested and implemented, below is a well-detailed list of the tested search algorithms in clearer view

(a) **Aho-Corasick Algorithm**:

This algorithm is a powerful multi-pattern string searching algorithm that constructs a finite state machine to process the search in linear time. It is an optimal string search algorithm invented by Alfred V. Aho and Margaret J. Corasick and was first published in June 1975. It offers the best complexity for any input and doesn't require much additional memory. This algorithm is helpful to find all occurrences of all given set of keywords. This is a type of dictionary-matching algorithm. It locates elements of a finite set of strings within an input text and matches all these strings simultaneously. The main idea behind it is to go through the text just once, search for all of the patterns in the dictionary at once. This algorithm uses a trie data structure to keep track of search patterns and uses a simple method to efficiently find all occurrences of a given set of patterns in any blob of text. A trie is a special data structure used to store strings that can be visualized like a graph. Once we make the tree, it coverts or try to convert tree in the form of automaton which helps to take linear time to complete or make the search. This is a type of dictionary-matching algorithm. It locates elements of a finite set of strings within an input text and matches all these strings simultaneously. The main idea behind it is to go through the text just once, search for all of the patterns in the dictionary at once. This algorithm uses a trie data structure to keep track of search patterns and uses a simple method to efficiently find all occurrences of a given set of patterns in any blob of text. A trie is a special data structure used to store strings that can be visualized like a graph. Once we make the tree, it converts or try to convert tree in the form of automaton which helps to take linear time to complete or make the search. Aho-Corasick Algorithm is subdivided into three major steps:

- Go Stage: It forms the tree with the help of provided keywords and the data which we feed in as recognizable pattern
- Failure Stage: It looks for the backward transformation to get appropriate appendix of keywords
- Output Stage: It searches all the words which ends at a particular stage.

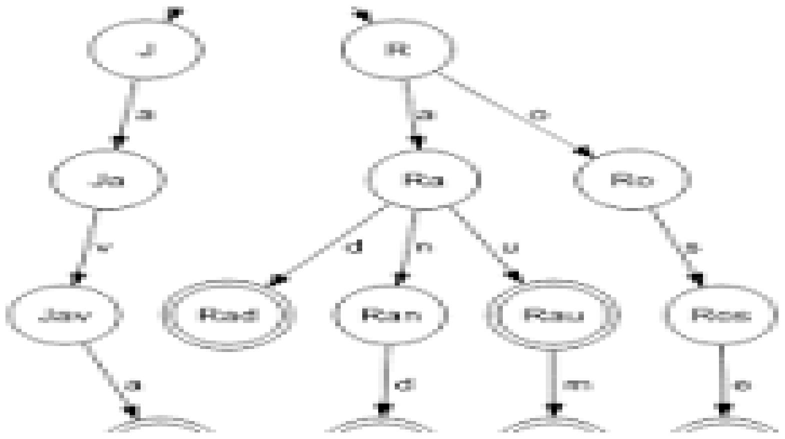
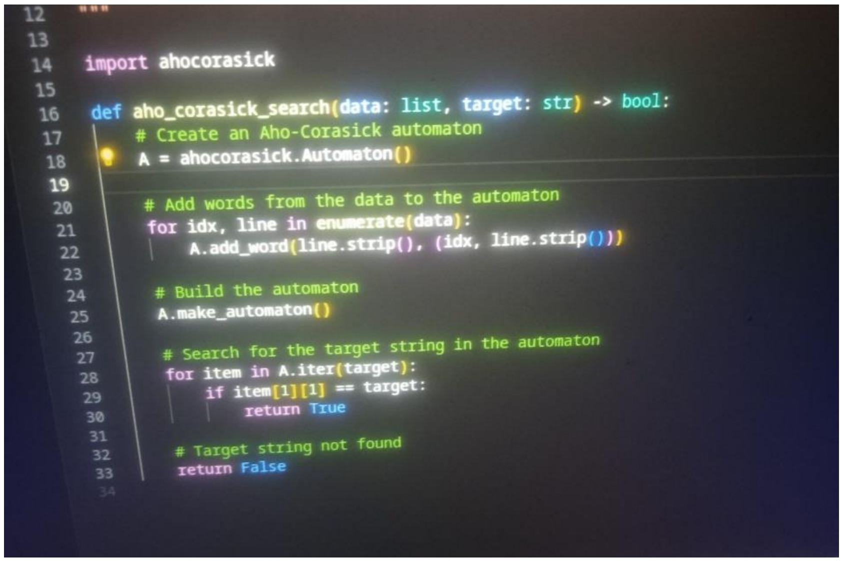


FIG 2.0 Aho Corasick Algorithm



```python
"""

import ahocorasick

def aho_corasick_search(data: list, target: str) -> bool:
    # Create an Aho-Corasick automaton
    A = ahocorasick.Automaton()

    # Add words from the data to the automaton
    for idx, line in enumerate(data):
        A.add_word(line.strip(), (idx, line.strip()))

    # Build the automaton
    A.make_automaton()

    # Search for the target string in the automaton
    for item in A.iter(target):
        if item[1][1] == target:
            return True

    # Target string not found
    return False
```

FIG 2.1 Code Implementation of Aho Corasick Algorithm

(b) **Binary Search Algorithm**:

Binary search algorithm is an efficient algorithm for finding an item from a sorted list of items. It works by repeatedly dividing in half the portion of the list that could contain an item, until you have narrowed down the possible locations to just one. Binary search algorithm was greatly implemented in this project as another option for users. Binary search can also be described as a search algorithm used to find the position of a target value within a sorted array. The search interval is halved by comparing the target element with the middle value of the search space. Implementing binary search is of two ways:

- Iterative Binary Search Algorithm: In this form of binary search, while loop is used to continue the process of comparing the key and splitting the search space into two halves
- Recursive Binary Search Algorithm: In this form, recursive function is created and then the mid of the search space is compared with the key. The result either returns the index where the key is found or calls the recursive function for the next search phase.

Binary search can be used as a building block for more complex algorithms used in machine learning, such as algorithms for training neural networks, or finding the optimal hyperparameters for a model. It can also be used for searching in compute graphics such as algorithms for ray tracing or texture mapping. The major advantage of binary search is that it is more efficient than other searching algorithms with a similar time complexity such as interpolation search or exponential search but the disadvantage is that the array should always be sorted.

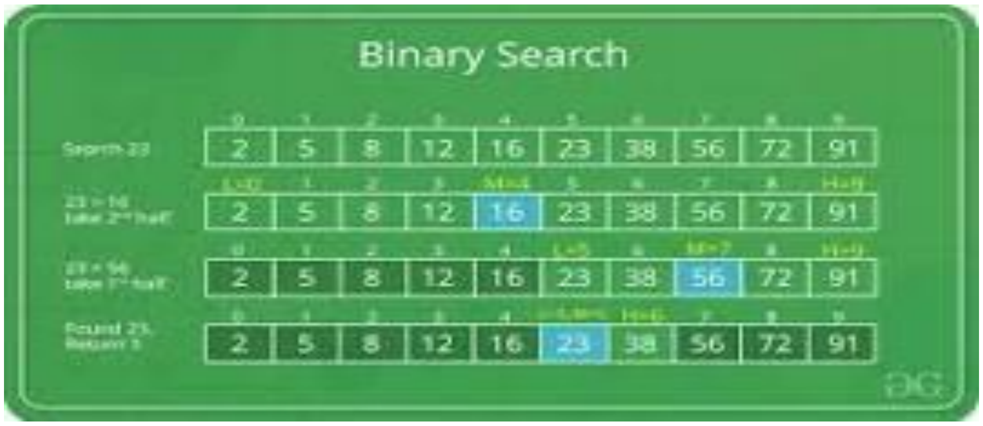
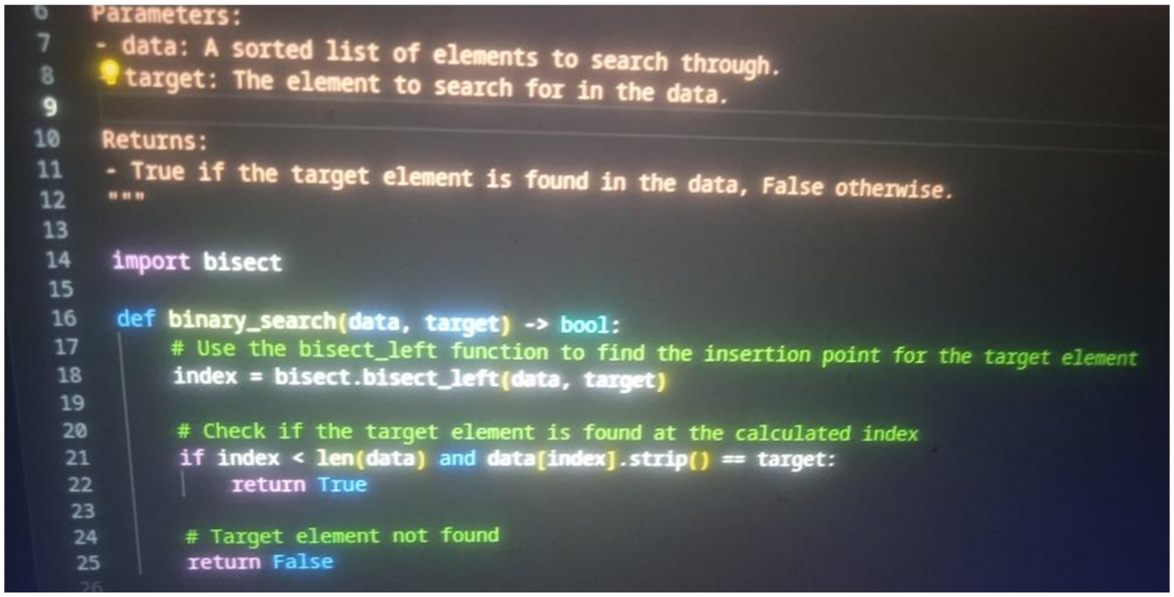


FIG 2.2 Binary Search Algorithm



```
Parameters:
- data: A sorted list of elements to search through.
- target: The element to search for in the data.

Returns:
- True if the target element is found in the data, False otherwise.
"""

import bisect

def binary_search(data, target) -> bool:
    # Use the bisect_left function to find the insertion point for the target element
    index = bisect.bisect_left(data, target)

    # Check if the target element is found at the calculated index
    if index < len(data) and data[index].strip() == target:
        return True

    # Target element not found
    return False
```

FIG 2.3 Code Implementation of Binary Search Algorithm

(c) **Boyer Moore Search Algorithm:**

The Boyer Moore algorithm is a string-matching algorithm. It finds all occurrences of a pattern string P or a substring in a text string T. The algorithm pre-processes the string pattern P being searched for but does not pre-process the text string T in which the pattern is being searched. The Boyer Moore algorithm speeds up searching by using pre-processed data to skip parts of the text, making it faster than many other string search algorithms. Its distinct characteristic involves matching from the pattern’s end instead of the beginning. Also, it traverses the text in larger jumps of multiple characters instead of checking each character one by one. The Boyer–Moore algorithm looks for instances of pattern P in text T by conducting targeted character comparisons at various alignments. Instead of implementing a brute-force approach that checks all alignments, Boyer–Moore uses pre-processed information from pattern P to skip as many alignments as feasible. For example, consider that we have a text string “Hello World!” and we want to locate the pattern string “World!”. The Boyer Moore algorithm will help find the pattern much quicker than the brute force approach.

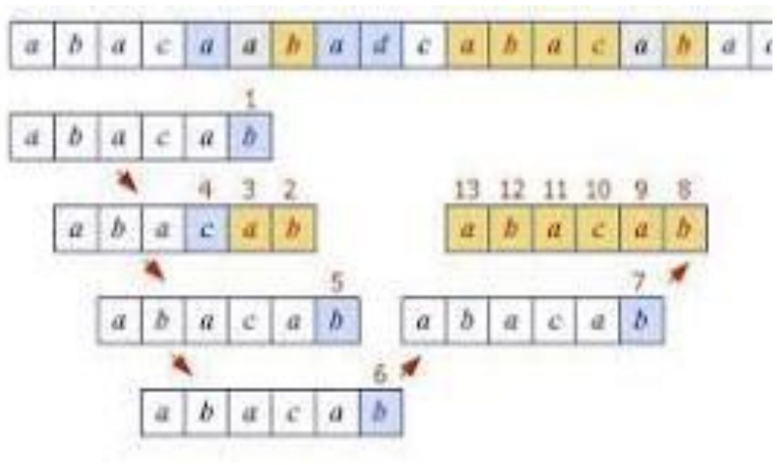
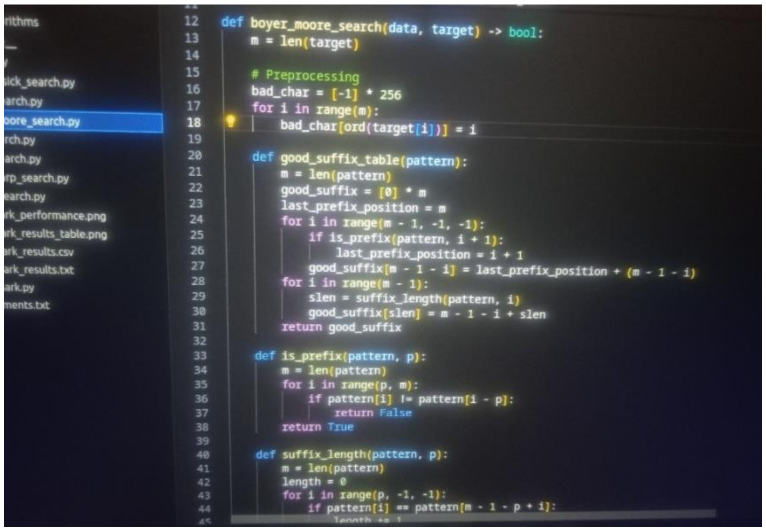


FIG 2.4 Boyer Moore Algorithm



```python
def boyer_moore_search(data, target) -> bool:
    m = len(target)

    # Preprocessing
    bad_char = [-1] * 256
    for i in range(m):
        bad_char[ord(target[i])] = i

    def good_suffix_table(pattern):
        m = len(pattern)
        good_suffix = [0] * m
        last_prefix_position = m
        for i in range(m - 1, -1, -1):
            if is_prefix(pattern, i + 1):
                last_prefix_position = i + 1
            good_suffix[m - 1 - i] = last_prefix_position + (m - 1 - i)
        for i in range(m - 1):
            slen = suffix_length(pattern, i)
            good_suffix[slen] = m - 1 - i + slen
        return good_suffix

    def is_prefix(pattern, p):
        m = len(pattern)
        for i in range(p, m):
            if pattern[i] != pattern[i - p]:
                return False
        return True

    def suffix_length(pattern, p):
        m = len(pattern)
        length = 0
        for i in range(p, -1, -1):
            if pattern[i] == pattern[m - 1 - p + i]:
                length += 1
```

FIG 2.5 Code Implementation of Boyer Moore

(d) **Naïve Pattern Search Algorithm:** A naive algorithm is used for pattern matching. It is a straightforward and easy-to-implement algorithm that makes it popular among other algorithms. It is used to find all the matching occurrences of specified text in the given string. It is useful for small texts and doesn't occupy extra memory space for searching and matching. It is the simplest method among other pattern searching algorithms. Although, it is more efficient than the brute force approach, however, it is not the most optimal method available. Like brute force, it also checks for all characters of the main string in order to find the pattern. Hence, its time complexity is O(m*n) where the 'm' is the size of pattern and 'n' is the size of the main string. This algorithm is helpful for smaller texts only.

The naive pattern searching algorithm does not require any pre-processing phases. We can find substring by checking once for the string. It also does not occupy extra space to perform the operation. If a match is found, the end result of the pattern matching operation will be the index of specified pattern, otherwise -1. Furthermore, this operation can return all the indices if the desired pattern appears multiple times within the main string.

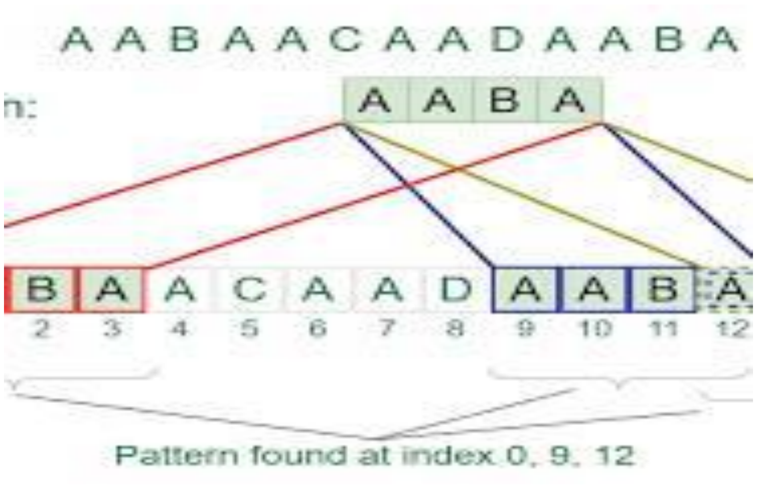
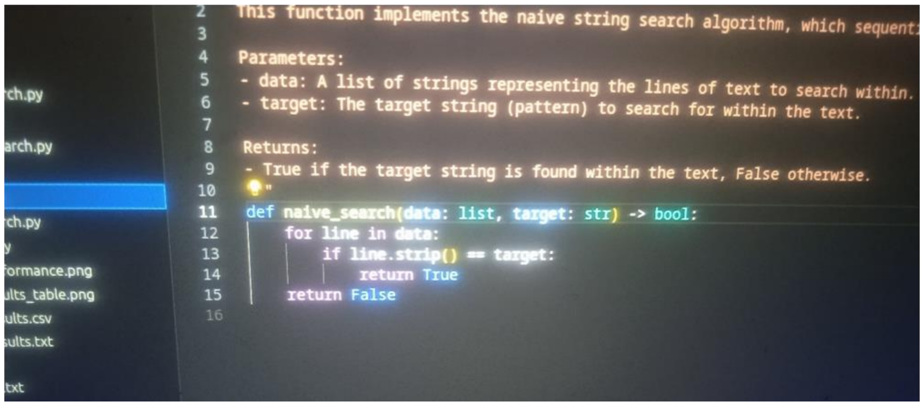


FIG 2.6 Naïve Pattern Search Algorithm



```
This function implements the naive string search algorithm, which sequent

Parameters:
- data: A list of strings representing the lines of text to search within.
- target: The target string (pattern) to search for within the text.

Returns:
- True if the target string is found within the text, False otherwise.
"
def naive_search(data: list, target: str) -> bool:
    for line in data:
        if line.strip() == target:
            return True
    return False
```

FIG 2.7 Code Implementation of Naïve Pattern Search Algorithm

(e) **Regular Expression Search Algorithm**:

A method for locating specific character strings embedded in character text is described and an implementation of this method in the form of a compiler is discussed. The compiler accepts a regular expression as source language and produces an IBM 7094 program as object language. The object program then accepts the text to be searched as input and produces a signal every time an embedded string in the text matches the given regular expression. Examples, problems, and solutions are also presented. Previous search algorithms involve backtracking when a partially successful search path fails. This necessitates a lot of storage and bookkeeping, and executes slowly. In the regular expression recognition technique described in this paper, each character in the text to be searched is examined in sequence against a list of all possible current characters. During this examination a new list of all possible next characters is built. When the end of the current list is reached, the new list becomes the current list, the next character is obtained, and the process continues. In the terms of Brzozowski [1], this algorithm continually takes the left derivative of the given regular expression with respect to the text to be searched. The parallel nature of this algorithm makes it extremely fast. The specific implementation of this algorithm is a compiler that translates a regular expression into IBM 7094 code. The compiled code, along with certain runtime routines, accepts the text to be searched as input and finds all substrings in the text that match the regular expression. The compiling phase of the implementation does not detract from the overall speed since any search routine must translate the input regular expression into some sort of machine accessible form. In the compiled code, the lists mentioned in the algorithm are not characters, but transfer instructions into the compiled code. The execution is extremely fast since a transfer to the top of the current list automatically searches for all possible sequel characters in the regular expression. This compile-search algorithm is incorporated as the context search in a time-sharing text editor. This is by no means the only use of such a search routine. For example, a variant of this algorithm is used as the symbol table search in an assembler. It is assumed that the reader is familiar with regular expressions and the machine language of the IBM 7094 computer. The compiler consists of three concurrently running stages. The first stage is a syntax sieve that allows only syntactically correct regular expressions to pass. This stage also inserts the operator "." for juxtaposition of regular expressions. The second stage converts the regular expression to reverse Polish form. The third stage is the object code producer. The first two stages are straightforward and are not discussed. The third stage expects a syntactically correct, reverse Polish regular expression. The regular expression a(b I c),d will be carried through as an example. This expression is translated into abc I * " d • by the first two stages. A functional description of the third stage of the compiler follows: The heart of the third stage is a pushdown stack. Each entry in the pushdown stack is a pointer to the compiled code of an operand. When a binary operator ("1" or ". ") is compiled, the top (most recent) two entries on the stack are combined and a resultant pointer for the operation replaces the two stack entries. The result of the binary operator is then available as an operand in another operation. Similarly, a unary operator ("*") operates on the top entry of the stack and creates an operand to replace that entry. When the entire regular expression is compiled, there is just one entry in the stack, and that is a pointer to the code for the regular expression. The compiled code invokes one of two functional routines. The first is called NNODE. NNODE matches a single character and will be represented by an oval containing the character that is recognized. The second functional routine is called CNODE. CNODE will split the current search path. It is represented by @ with one input path and two output paths.

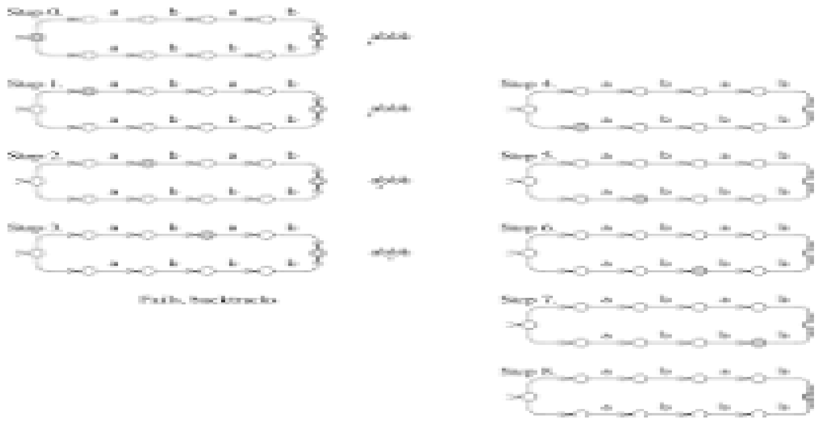
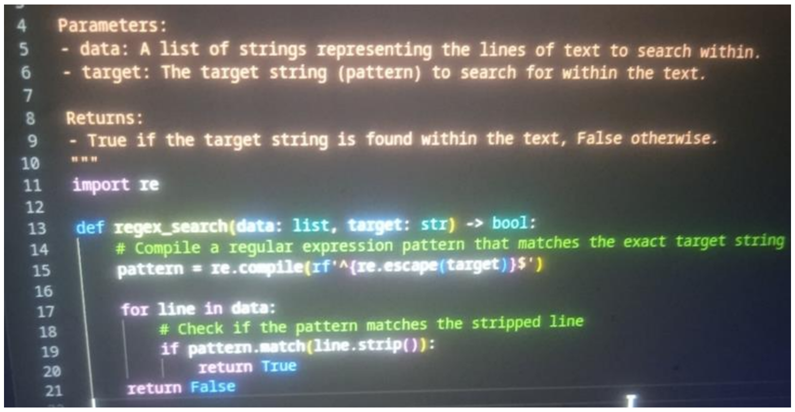


FIG 2.8 Regular Expression Search Algorithm



```
Parameters:
- data: A list of strings representing the lines of text to search within.
- target: The target string (pattern) to search for within the text.

Returns:
- True if the target string is found within the text, False otherwise.
"""
import re

def regex_search(data: list, target: str) -> bool:
    # Compile a regular expression pattern that matches the exact target string
    pattern = re.compile(rf'^{re.escape(target)}$')

    for line in data:
        # Check if the pattern matches the stripped line
        if pattern.match(line.strip()):
            return True
    return False
```

FIG 2.9 Code Implementation of Regular Expression Search Algorithm

(f) **Knuth Morris Prath Search Algorithm**:

KMP is a linear string-matching algorithm with low time complexity. It is the solution to the drawbacks of previous text-matching algorithms. KMP is the fastest string-matching algorithm. Its full form is Knuth Morris Pratt Algorithm. The KMP algorithm got this name from the name of its inventors. In 1970, James H Morris, Vaughan Pratt, and Donald Knuth invented this linear string-matching algorithm. The KMP is the only string-matching algorithm with a time complexity of O(n+m), where n is the string length and m is the pattern length. In string-matching algorithms, there are two terminologies: sting or text and pattern.

- String or text is the original string that is used for matching.
- The pattern is the sample text which is to be matched.

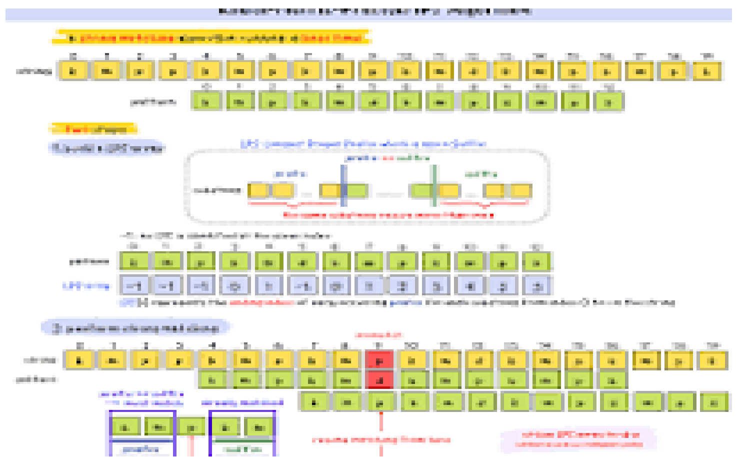
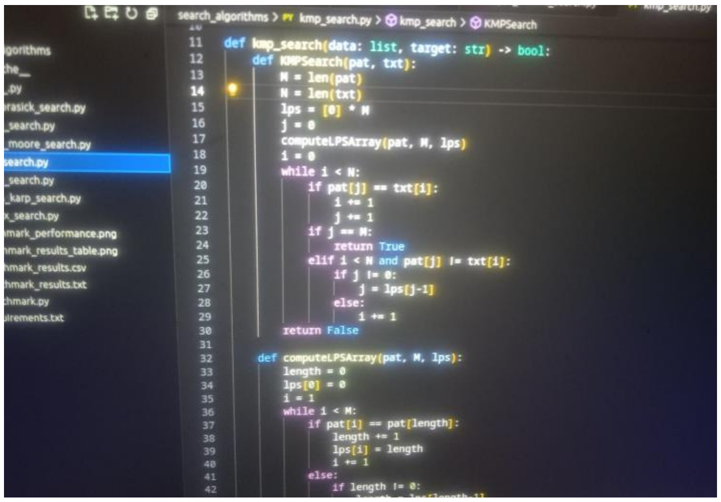


FIG 2.10 Knuth Morris Prath Search Algorithm



```
search_algorithms > kmp_search.py > kmp_search > KMPSearch
def kmp_search(data: list, target: str) -> bool:
    def KMPSearch(pat, txt):
        M = len(pat)
        N = len(txt)
        lps = [0] * M
        j = 0
        computeLPSArray(pat, M, lps)
        i = 0
        while i < N:
            if pat[j] == txt[i]:
                i += 1
                j += 1
            if j == M:
                return True
            elif i < N and pat[j] != txt[i]:
                if j != 0:
                    j = lps[j-1]
                else:
                    i += 1
        return False

    def computeLPSArray(pat, M, lps):
        length = 0
        lps[0] = 0
        i = 1
        while i < M:
            if pat[i] == pat[length]:
                length += 1
                lps[i] = length
                i += 1
            else:
                if length != 0:
```

FIG 2.11 Code Implementation of Knuth Morris Prath Search Algorithm

(g) **Rabin Karp Search Algorithm:**

The Rabin-Karp algorithm is a string-searching algorithm that uses hashing to find patterns in strings. A string is an abstract data type that consists of a sequence of characters. Letters, words, sentences, and more can be represented as strings. It is also an algorithm used for searching/matching patterns in the text using a hash function. Unlike Naive string-matching algorithm, it does not travel through every character in the initial phase rather it filters the characters that do not match and then performs the comparison. The way the algorithm works is that a sequence of characters is taken and checked for the possibility of the presence of the required string. If the possibility is found then, character matching is performed. Created by Richard M. Karp and Michael O. Rabin, the Rabin-Karp algorithm was developed in 1987. This is an algorithm that uses hashing to find patterns between two strings. Compared to other algorithms that look at large chunks of a string at a time to speed up the comparisons, such as Knuth-Morris-Pratt and Boyer-Moore string-search algorithm, the Rabin-Karp algorithm uses hashing to increase the speed of the comparison. This type of string compare algorithm is very useful for something like detecting plagiarism as it can quickly go through long sets of strings and see how many matches there are. The Rabin-Karp algorithm is one of the best plagiarism detection algorithms and can run very and simply if a strong hash function is used. String-searching algorithms have many important uses in society and their functionality cannot be undervalued. As previously mentioned, an algorithm such as Rabin-Karp is used for plagiarism detection as it can quickly scan through an entire document and search for patterns. This works by breaking down a document into a string or a series of strings which can then be scanned through another document looking for matches. Once the algorithm is ran, the results can be seen and the number of matches can be revealed. On top of plagiarism detection, string-searching algorithms are able to complete DNA sequencing, forensics, spell checking, large search engine searches, and intrusion detection. There is a need for good string-searching algorithms and the Rabin-Karp algorithm is able to fulfill this need. The Rabin-Karp string searching algorithm works by using a hash function, specifically a rolling hash function, to do the string comparison. The algorithm is passed 3 items: the entire text that is being searched through, the string that is searched for (the pattern), and a prime number for the hash function.

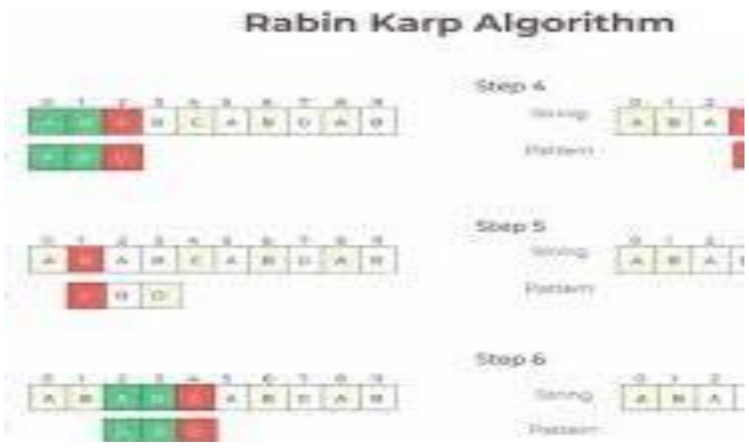
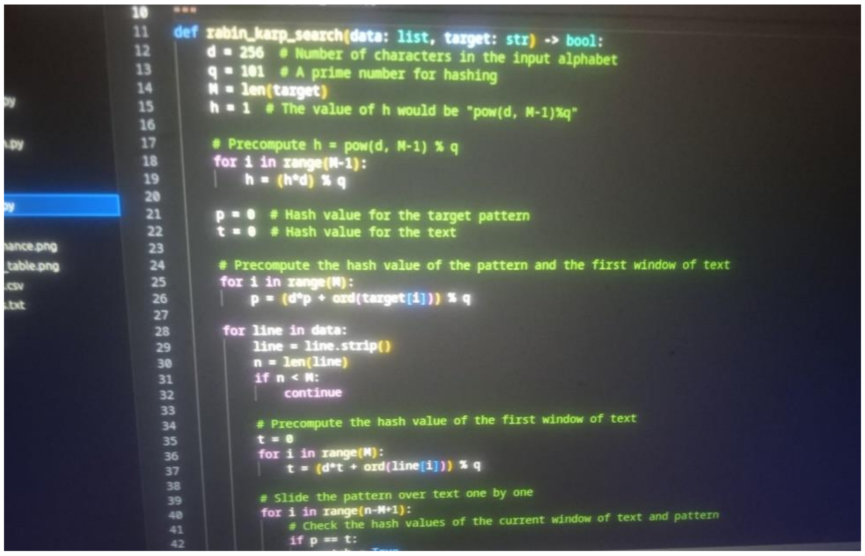


FIG 2.12 Rabin Karp Search Algorithm



```python
"""
def rabin_karp_search(data: list, target: str) -> bool:
    d = 256  # Number of characters in the input alphabet
    q = 101  # A prime number for hashing
    M = len(target)
    h = 1  # The value of h would be "pow(d, M-1)%q"

    # Precompute h = pow(d, M-1) % q
    for i in range(M-1):
        h = (h*d) % q

    p = 0  # Hash value for the target pattern
    t = 0  # Hash value for the text

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

    for line in data:
        line = line.strip()
        n = len(line)
        if n < M:
            continue

        # Precompute the hash value of the first window of text
        t = 0
        for i in range(M):
            t = (d*t + ord(line[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 p == t:
```

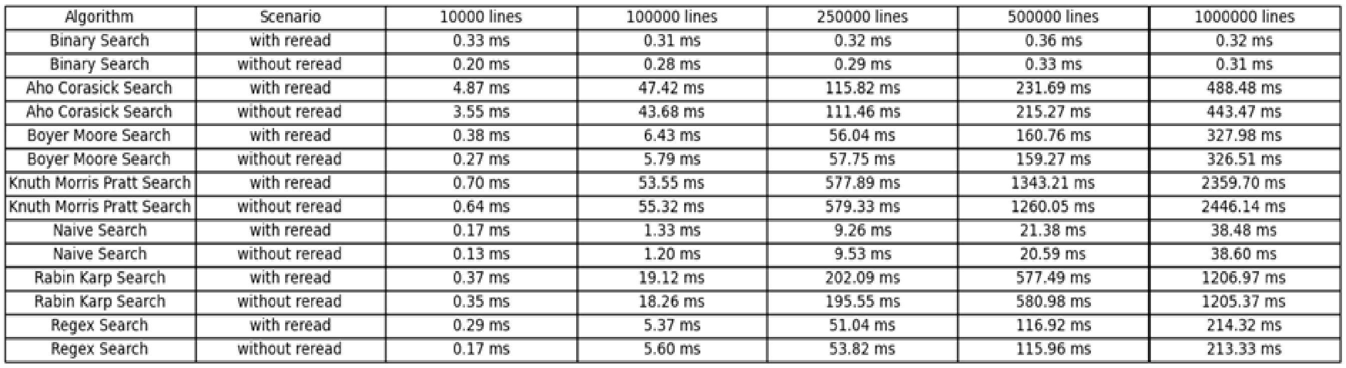
FIG 2.13 Code Implementation of Rabin Karp Search Algorithm

CHAPTER THREE

RESULT

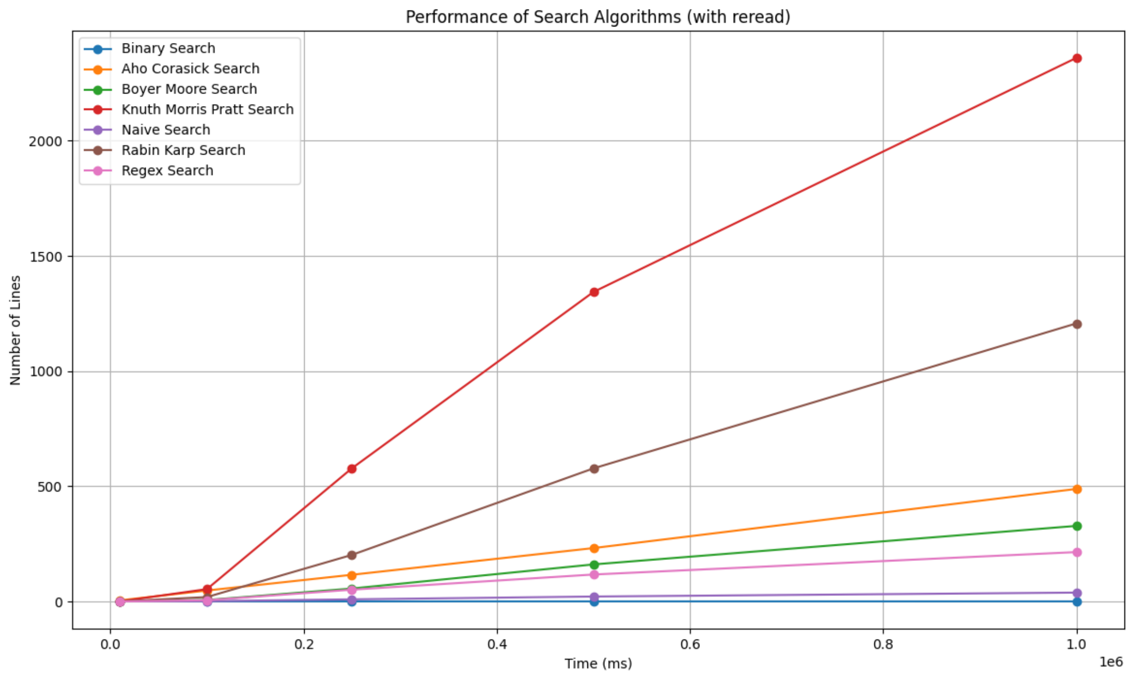
3.1 Search Algorithm against Performance

Benchmark Results:

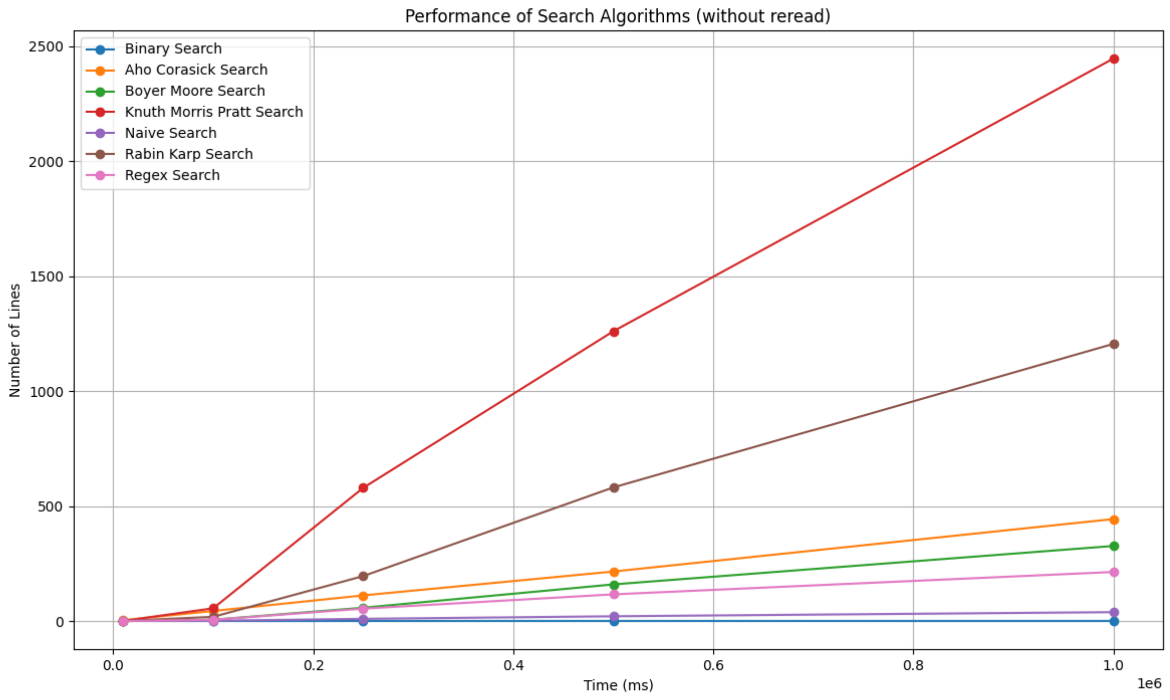


Algorithm	Scenario	10000 lines	100000 lines	250000 lines	500000 lines	1000000 lines
Binary Search	with reread	0.33 ms	0.31 ms	0.32 ms	0.36 ms	0.32 ms
Binary Search	without reread	0.20 ms	0.28 ms	0.29 ms	0.33 ms	0.31 ms
Aho Corasick Search	with reread	4.87 ms	47.42 ms	115.82 ms	231.69 ms	488.48 ms
Aho Corasick Search	without reread	3.55 ms	43.68 ms	111.46 ms	215.27 ms	443.47 ms
Boyer Moore Search	with reread	0.38 ms	6.43 ms	56.04 ms	160.76 ms	327.98 ms
Boyer Moore Search	without reread	0.27 ms	5.79 ms	57.75 ms	159.27 ms	326.51 ms
Knuth Morris Pratt Search	with reread	0.70 ms	53.55 ms	577.89 ms	1343.21 ms	2359.70 ms
Knuth Morris Pratt Search	without reread	0.64 ms	55.32 ms	579.33 ms	1260.05 ms	2446.14 ms
Naive Search	with reread	0.17 ms	1.33 ms	9.26 ms	21.38 ms	38.48 ms
Naive Search	without reread	0.13 ms	1.20 ms	9.53 ms	20.59 ms	38.60 ms
Rabin Karp Search	with reread	0.37 ms	19.12 ms	202.09 ms	577.49 ms	1206.97 ms
Rabin Karp Search	without reread	0.35 ms	18.26 ms	195.55 ms	580.98 ms	1205.37 ms
Regex Search	with reread	0.29 ms	5.37 ms	51.04 ms	116.92 ms	214.32 ms
Regex Search	without reread	0.17 ms	5.60 ms	53.82 ms	115.96 ms	213.33 ms

TAB 3.1 Algorithm sorted based on Performance



CHA 3.1 Performance Of Search Algorithms (with reread)



CHA 3.2 Performance of Search Algorithms (without reread)

CHAPTER FOUR

ANALYSIS

In the realm of file searching, efficiency is paramount. As the volume of data continues to escalate, the need for swift and reliable file-search algorithms becomes increasingly imperative. In this report, we delve into the performance analysis of various file-search options and algorithms. The objective is twofold: To explore seven distinct file-search algorithms and to discern their efficacy across different file sizes.

Our investigation encompasses an array of methodologies, ranging from the simplistic linear search to more sophisticated approaches such as binary and interpolation searches. Each algorithm is rigorously tested under diverse scenarios, including small, medium, and large file sizes. Through meticulous measurement and analysis, we aim to unravel the nuances of each algorithm's performance and unveil the optimal strategies for expedient file retrieval.

By synthesizing empirical data with theoretical frameworks, this analysis endeavors to furnish stakeholders with actionable insights to optimize file-search operations in various computational environments.

4.1 Detailed Analysis of Results Obtained

To further analyze the performance data in detailed format, it must first be broken down into six steps:

- Algorithm Performance Trend: The data shows the performance of several string search algorithms across different numbers of lines and with or without reread. The trend shows that as the number of line increases, the time taken by each algorithm generally increases as well. This is expected due to the increased amount of data that needs to be processed.

- Effect of Algorithm Complexity: The performance differences between algorithms are significant. For example, Binary Search consistently performs the fastest, followed by Naïve Search and Regex Search. On the other hand, algorithms like Rabin Karp Search and Knuth Morris Pratt Search exhibit consistent higher execution times, especially as the number of lines increases. This reflects the different complexities of these algorithms and how they scale with larger datasets.

- Impact of Reread: The performance data is provided for both with or without reread scenarios. Rereading likely involves additional disk I/O operations, which could impact the overall performance. However, in this dataset, the differences between with or without reread scenarios are minimal, suggesting that disk I/O might not be the bottleneck in this context.

- Scalability: As the number of lines increases, the performance gap between algorithms widens. This indicates that some algorithms scale better with larger datasets compared to others. Binary Search, for instance, maintains relatively stable performance even with a significant increase in the number of lines, while others like Rabin Karp Search and Knuth Morris Pratt Search experience substantial increases in execution time.

- Optimization Potential: The performance data provides insights into areas where optimization efforts could b focused. For instance, algorithms with higher execution times, such as Knuth Morris Pratt Search, might benefit from optimization techniques to improve their efficiency, especially when dealing with larger datasets.

- Real-world Application Consideration: The choice of algorithm can have a significant impact on the performance of a system. Understanding the trade-offs between execution time and other factors such as memory usage and ease of implementation is crucial when selecting an algorithm for a real-world application.

Therefore, the crucial analysis of this performance data highlights the importance of selecting the right algorithm based on the specific requirements and constraints of the problem at hand. It also emphasizes the need for ongoing optimization efforts to ensure efficient performance, particularly as the scale of data increases.

CHAPTER FIVE

CONCLUSION

In conclusion, the choice of string search algorithm significantly impacts performance, with some algorithms performing notably better than others across varying dataset sizes. While some algorithms exhibit consistent performance regardless of dataset size (e.g Binary Search), others experience a substantial increase in execution time as the dataset grows (e.g Rabin Karp Search, Knuth Morris Pratt Search).

The difference in performance highlights the importance of considering algorithmic complexity when designing and implementing string search functionality. Algorithms with higher complexities tend to exhibit longer execution time, particularly with larger datasets. Therefore, there are opportunities for algorithm with higher execution times, optimizing these algorithms can lead to significant improvements in overall performance, making them more suitable for real-world applications with large datasets.

5.1 Summary of Findings

The performance analysis of various string search algorithms reveals significant differences in execution times based on algorithm choice and dataset size. While some algorithms, such as Binary Search, maintain consistent and efficient performance across different dataset sizes, others, like Rabin Karp Search and Knuth Morris Pratt Search, experience notable increases in execution time as the dataset grows. These findings emphasize the importance of algorithm selection and optimization to ensure efficient performance, particularly with large datasets. Additionally, the minimal impact of reread suggests that disk I/O might not be a significant concern in the performance of these algorithms. Overall, understanding algorithmic complexity and scalability is critical for effectively designing and implementing string search functionality in real-world applications.

5.2 Recommendations of the Fastest Algorithm

Therefore, with the aid of a well-detailed analysis stated above, it can be inferred that binary search can be recommended as the fastest search algorithm.

REFERENCE

Aho, A.V., Corasick, M.J. (2021). Aho-Corasick algorithm. *(Medium)*. Retrieved from (https://medium.com/pattern-searching-algorithm/aho-corasick-algorithm-7e5c2e58861c)

Chat GPT. (2024).

Educative Inc. (2024). Naïve Pattern Search Algorithm. (https://www.educative.io/answers/thenaive-algorithm-for-pattern-searching).

Intellipaat Software Solutions. (2011-2024). Boyer Moore Algorithm *(Intellipaat)*.

Programmiz (2022). Rabin Karp Search Algorithm. *Programmiz)*.

Sanchhaya Education Private Limited (2021). Binary Search Algorithm.

Thompson, K. (1968). Regular Expression Search Algorithm (*Bell Laboratories, Inc. Murray Hill, New Jersey*.