Project Report: String Matching Algorithms

1. EXPLANATION OF THE CONSIDERED ALGORITHMS

In this project, we have implemented and analyzed three fundamental string matching algorithms: the Brute Force algorithm, Knuth-Morris-Pratt (KMP) algorithm, and Boyer-Moore algorithm. Each of these algorithms has unique approaches and efficiencies in solving the problem of finding occurrences of a pattern within a given text.

Brute Force Algorithm

The Brute Force algorithm is the simplest and most intuitive string matching algorithm. It works by checking for the pattern at every possible position in the text. For each position in the text, it compares the substring starting at that position with the pattern. If all characters match, it records the position; otherwise, it moves to the next position and repeats the process.

Steps:

1. Start at the beginning of the text.
2. Compare the pattern with the substring of the text of the same length.
3. If a mismatch is found, move one position to the right in the text.
4. Repeat steps 2 and 3 until the end of the text is reached.

Complexity:

- Worst-case time complexity: O(n * m), where n is the length of the text and m is the length of the pattern.
- Best-case time complexity: O(n), if the pattern is found at the first position.

Knuth-Morris-Pratt (KMP) Algorithm

The KMP algorithm improves the efficiency of the string matching process by avoiding unnecessary comparisons. It achieves this by preprocessing the pattern to create a longest prefix suffix (LPS) array, which helps in skipping characters in the text that have already been matched.

Steps:

1. Preprocess the pattern to create the LPS array.
2. Use the LPS array to skip comparisons in the text.
3. Compare the pattern with the text and move based on the LPS array when a mismatch occurs.

Complexity:

- Worst-case time complexity: O(n + m), due to the preprocessing step and the linear scanning of the text.
- Best-case time complexity: O(n), similar to the worst-case as the preprocessing is done once.

Boyer-Moore Algorithm

The Boyer-Moore algorithm is known for its efficiency and uses two heuristics to improve the performance: the bad character heuristic and the good suffix heuristic. These heuristics help in determining how far to shift the pattern when a mismatch occurs, often allowing the algorithm to skip large sections of the text.

Steps:

- Preprocess the pattern to create the bad character table.
- Start comparing the pattern from the end of the pattern to the beginning.
- If a mismatch occurs, use the bad character table to determine the shift.
- Repeat steps 2 and 3 until the pattern is found or the text is exhausted.

Complexity:

- Worst-case time complexity: O(n * m), although this rarely occurs due to the heuristics.
- Average-case time complexity: O(n / m), which makes it very efficient in practice for large alphabets and long patterns.

2. Explanation of the Implementations

The implementation of the string matching algorithms (Brute Force, KMP, and Boyer-Moore) was done in Python. This section provides an overview of the main data structures, input/output, and key aspects of the implementation.

Brute Force Algorithm

Data Structures:

- The primary data structures used are strings for both the text and the pattern.
- A list is used to store the positions where the pattern matches the text.

Implementation: The Brute Force algorithm is straightforward. It iterates through each possible starting position in the text and checks if the pattern matches the substring starting at that position. If a match is found, the starting position is recorded.

Input/Output:

- **Input:** The algorithm takes a text string and a pattern string as input.
- **Output:** It returns a list of starting indices where the pattern is found in the text.

```
def brute_force_search(text, pattern):
    n = len(text)
    m = len(pattern)
    matches = []
    for i in range(n - m + 1):
        match = True
        for j in range(m):
            if text[i + j] != pattern[j]:
                match = False
                break
        if match:
            matches.append(i)
    return matches
```

[fig 0: brute force code]

Knuth-Morris-Pratt (KMP) Algorithm

Data Structures:

- The primary data structures are strings for the text and the pattern.
- An additional array, the LPS (Longest Prefix Suffix) array, is used to store the preprocessing information about the pattern.

Implementation: The KMP algorithm involves two main steps:

1. **Preprocessing the Pattern:** This step computes the LPS array, which indicates the longest proper prefix of the pattern that is also a suffix. This array helps in skipping unnecessary comparisons.
2. **Pattern Matching:** Using the LPS array, the algorithm scans the text to find matches, efficiently skipping parts of the text where mismatches are guaranteed.

Input/Output:

- **Input:** The algorithm takes a text string and a pattern string as input.
- **Output:** It returns a list of starting indices where the pattern is found in the text.

```
def compute_lps(pattern):
    m = len(pattern)
    lps = [0] * m
    length = 0
    i = 1
    while i < m:
        if pattern[i] == pattern[length]:
            length += 1
            lps[i] = length
            i += 1
        else:
            if length != 0:
                length = lps[length - 1]
            else:
                lps[i] = 0
                i += 1
    return lps
```

[fig 1: KMP Code (1/2)]

```
def kmp_search(text, pattern):
 n = len(text)
 m = len(pattern)
 lps = compute_lps(pattern)
 matches = []
 i = 0
 j = 0
 while i < n:
 if pattern[j] == text[i]:
 i += 1
 j += 1
 if j == m:
 matches.append(i - j)
 j = lps[j - 1]
 elif i < n and pattern[j] != text[i]:
 if j != 0:
 j = lps[j - 1]
 else:
 i += 1
 return matches
```

[fig 1 : KMP Code (2/2)]

Boyer-Moore Algorithm

Data Structures:

- The primary data structures are strings for the text and the pattern.
- An array (bad character table) is used to store the rightmost occurrence of each character in the pattern.

Implementation: The Boyer-Moore algorithm involves the following steps:

1. **Preprocessing the Pattern:** This step constructs the bad character table, which is used to determine how far to shift the pattern when a mismatch occurs.
2. **Pattern Matching:** The algorithm compares the pattern with the text from right to left. If a mismatch occurs, the bad character table is used to shift the pattern efficiently, often allowing large jumps.

Input/Output:

- **Input:** The algorithm takes a text string and a pattern string as input.
- **Output:** It returns a list of starting indices where the pattern is found in the text.

```
def bad_character_heuristic(pattern):
    bad_char = [-1] * 256
    for i in range(len(pattern)):
        bad_char[ord(pattern[i])] = i
    return bad_char

def boyer_moore_search(text, pattern):
    n = len(text)
    m = len(pattern)
    bad_char = bad_character_heuristic(pattern)
    s = 0
    matches = []
    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 += (m - bad_char[ord(text[s + m])] if s + m
< n else 1)
        else:
            s += max(1, j - bad_char[ord(text[s + j])])
    return matches
```

[fig 2: Boyer moore]

3. Description of the Experiment Scenarios

The experiments conducted aimed to evaluate the performance of the three string matching algorithms (Brute Force, KMP, Boyer-Moore) in terms of their running times on different input sizes. The experimental setup and methodology are detailed below.

Experiment Setup

Data Generation:

- **Corpus Generation:** A corpus of documents was generated randomly. Each document was a string consisting of lowercase alphabetic characters. The size of each document varied from 100 to 10,000 characters, increasing by 100 characters for each subsequent document.
- **Pattern Generation:** Patterns of lengths 1, 3, 5, and 10 characters were randomly generated. These patterns were used to test the algorithms across various scenarios.

Inclusion of Patterns:

- In each document, the pattern was included at random positions in half of the documents (documents with even indices). This setup ensured that each pattern had a 50% chance of being present in a document, simulating a realistic search scenario.

Number of Repeats:

- Each experiment was repeated 5 times for statistical significance. The running times were averaged to smooth out any anomalies or variations.

Methodology

Algorithm Execution:

- Each algorithm (Brute Force, KMP, Boyer-Moore) was executed on the generated corpus for each pattern length. The algorithms were run sequentially on the same set of documents to ensure consistency.
- The running time for each execution was recorded.

Measurements:

- The running times were measured using Python's **time.time()** function, capturing the start and end times of each algorithm's execution.
- The times were averaged over the 5 runs to obtain meaningful and reliable results.

Plotting Results:

- The average running times for each algorithm were plotted against the input size (document size) to visualize and compare the performance.
- Separate plots were created for each pattern length (1, 3, 5, and 10) to observe how the pattern length affects the performance of the algorithms.

Example of Experiment Execution:

For a pattern length of 3:

- Generate a random pattern of length 3.
- Generate a corpus of 10 documents for each size from 100 to 10,000 characters, ensuring the pattern is included in half the documents.
- Execute each algorithm on the corpus and record the running times.
- Repeat steps 1-3 five times and average the running times.
- Plot the average running times against the document sizes.

Experiment Code

The following code snippet in next page shows how the experiments were conducted and the results were plotted:

Experiment Code

```
import random
import string
import time
import matplotlib.pyplot as plt

# Brute Force Algorithm
def brute_force_search(text, pattern):
 n = len(text)
 m = len(pattern)
 matches = []
 for i in range(n - m + 1):
 match = True
 for j in range(m):
 if text[i + j] != pattern[j]:
 match = False
 break
 if match:
 matches.append(i)
 return matches

# KMP Algorithm
def compute_lps(pattern):
 m = len(pattern)
 lps = [0] * m
 length = 0
 i = 1
 while i < m:
 if pattern[i] == pattern[length]:
 length += 1
 lps[i] = length
 i += 1
 else:
 if length != 0:
 length = lps[length - 1]
 else:
 lps[i] = 0
 i += 1
 return lps

def kmp_search(text, pattern):
 n = len(text)
 m = len(pattern)
 lps = compute_lps(pattern)
 matches = []
 i = 0
 j = 0
 while i < n:
 if pattern[j] == text[i]:
 i += 1
 j += 1
 if j == m:
 matches.append(i - j)
 j = lps[j - 1]
 elif i < n and pattern[j] != text[i]:
 if j != 0:
 j = lps[j - 1]
 else:
 i += 1
 return matches

# Boyer-Moore Algorithm
def bad_character_heuristic(pattern):
 bad_char = [-1] * 256
 for i in range(len(pattern)):
 bad_char[ord(pattern[i])] = i
 return bad_char


def boyer_moore_search(text, pattern):
 n = len(text)
 m = len(pattern)
 bad_char = bad_character_heuristic(pattern)
 s = 0
 matches = []
 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 += (m - bad_char[ord(text[s + m])] if s + m < n else 1)
 else:
 s += max(1, j - bad_char[ord(text[s + j])])
 return matches

# Function to generate random text
def generate_random_text(size, include_pattern=False, pattern=""):
 text = ''.join(random.choices(string.ascii_lowercase, k=size))
 if include_pattern:
 pos = random.randint(0, len(text) - len(pattern))
 text = text[:pos] + pattern + text[pos + len(pattern):]
 return text

# Function to generate a random pattern
def generate_random_pattern(length):
 return ''.join(random.choices(string.ascii_lowercase, k=length))

# Function to search in a corpus
def search_in_corpus(corpus, pattern, search_function):
 result_documents = []
 for i, document in enumerate(corpus):
 if search_function(document, pattern):
 result_documents.append(i)
 return result_documents

# Main function for running experiments
def main():
 pattern_lengths = [1, 3, 5, 10] # Different lengths of the random patterns
 sizes = list(range(100, 10001, 100))
 corpus_size = 10 # Number of documents in the corpus
 num_repeats = 5 # Number of repetitions for each experiment

 results = []

 for pattern_length in pattern_lengths:
 times_brute_force = []
 times_kmp = []
 times_boyer_moore = []

 for size in sizes:
 times_brute_force_sum = 0
 times_kmp_sum = 0
 times_boyer_moore_sum = 0

 for _ in range(num_repeats):
 pattern = generate_random_pattern(pattern_length)
 print(f"Pattern of length {pattern_length}: {pattern}")
 corpus = [generate_random_text(size, include_pattern=(i % 2 == 0), pattern=pattern) for i in range(corpus_size)]

 start_time = time.time()
 for doc in corpus:
 brute_force_search(doc, pattern)
 elapsed_time = (time.time() - start_time)
 times_brute_force_sum += elapsed_time

 start_time = time.time()
 for doc in corpus:
 kmp_search(doc, pattern)
 elapsed_time = (time.time() - start_time)
 times_kmp_sum += elapsed_time

 start_time = time.time()
 for doc in corpus:
 boyer_moore_search(doc, pattern)
 elapsed_time = (time.time() - start_time)
 times_boyer_moore_sum += elapsed_time

 times_brute_force.append(times_brute_force_sum / num_repeats)
 times_kmp.append(times_kmp_sum / num_repeats)
 times_boyer_moore.append(times_boyer_moore_sum / num_repeats)

 results.append({
 'pattern_length': pattern_length,
 'sizes': sizes,
 'times_brute_force': times_brute_force,
 'times_kmp': times_kmp,
 'times_boyer_moore': times_boyer_moore
 })

 # Plotting results
 for result in results:
 pattern_length = result['pattern_length']
 plt.figure(figsize=(12, 8))
 plt.plot(result['sizes'], result['times_brute_force'], label='Brute Force', marker='o')
 plt.plot(result['sizes'], result['times_kmp'], label='KMP', marker='x')
 plt.plot(result['sizes'], result['times_boyer_moore'], label='Boyer-Moore', marker='s')
 plt.xlabel('Text Size')
 plt.ylabel('Average Running Time (seconds)')
 plt.title(f'Performance of String Matching Algorithms (Pattern length {pattern_length})')
 plt.legend()
 plt.grid(True)
 plt.show()

 # Running search on a single corpus for demonstration
 pattern_length = 10
 pattern = generate_random_pattern(pattern_length)
 corpus = [generate_random_text(1000, include_pattern=(i % 2 == 0), pattern=pattern) for i in range(corpus_size)]

 bf_results = search_in_corpus(corpus, pattern, brute_force_search)
 kmp_results = search_in_corpus(corpus, pattern, kmp_search)
 bm_results = search_in_corpus(corpus, pattern, boyer_moore_search)

 print(f"Pattern used for search: {pattern}\n")
 print("Documents containing the pattern using Brute Force:", bf_results)
 print("Documents containing the pattern using KMP:", kmp_results)
 print("Documents containing the pattern using Boyer-Moore:", bm_results)

 for i, doc in enumerate(corpus):
 print(f"Document {i}:\n{doc}\n")

if __name__ == "__main__":
 main()
```

4. Analysis and Discussion of the Obtained Results

The experiments produced several plots that depict the performance of the three algorithms (Brute Force, KMP, Boyer-Moore) across different text sizes and pattern lengths. Here, we discuss the results, comparing the theoretical and empirical complexities, and analyze the implications of the findings.

Theoretical vs. Empirical Complexity

Brute Force Algorithm:

- Theoretical Complexity: O(n * m)
- Empirical Observation: The running time increased significantly with both text size and pattern length. This was expected as the Brute Force algorithm examines every possible position in the text.

KMP Algorithm:

- Theoretical Complexity: O(n + m)
- Empirical Observation: The KMP algorithm performed consistently well across different pattern lengths and text sizes. The running times were more stable and generally lower compared to the Brute Force algorithm, validating its theoretical advantage.

Boyer-Moore Algorithm:

- Theoretical Complexity: O(n * m) in the worst case, but typically O(n / m)
- Empirical Observation: The Boyer-Moore algorithm often outperformed both the Brute Force and KMP algorithms, especially for longer patterns. Its use of heuristics allowed it to skip large portions of the text, resulting in faster running times.

Analysis of the Plots

Pattern Length of 1:

- All algorithms performed similarly for very short patterns. The overhead of preprocessing in KMP and Boyer-Moore was not significant due to the small pattern size.

Pattern Length of 3:

- The KMP and Boyer-Moore algorithms began to show their advantages over the Brute Force algorithm. The Boyer-Moore algorithm started to benefit from its heuristics, leading to faster running times.

Pattern Length of 5:

- The difference in performance became more pronounced. Boyer-Moore consistently had the shortest running times, followed by KMP, with Brute Force lagging behind.

Pattern Length of 10:

- The Boyer-Moore algorithm showed a significant performance boost, often skipping large sections of the text. KMP maintained stable performance, while the Brute Force algorithm continued to show the highest running times.

5. Conclusion

Which Algorithm is Best and Under What Circumstances?

- Brute Force: Simple to implement but inefficient for large texts or patterns. Best suited for small texts and patterns where ease of implementation is a priority.
- KMP: Efficient and stable for all text sizes and pattern lengths. It is a good general-purpose algorithm.
- Boyer-Moore: The fastest for larger patterns and texts, especially when the pattern length is significant compared to the text. Best used when performance is critical.

Classify the Algorithms' Performances in Function of the Input Size:

- For small texts ($n < 1000$), the difference in performance is less noticeable. However, as the text size increases, Boyer-Moore outperforms the others, followed by KMP, with Brute Force being the slowest.
- As the pattern length increases, the Boyer-Moore algorithm's advantage becomes more significant.

Findings:

- The Boyer-Moore algorithm's heuristics make it highly efficient for longer patterns, allowing it to skip large sections of the text and reduce the number of comparisons.
- The KMP algorithm's preprocessing step (LPS array) provides a balanced performance, making it a reliable choice for various text and pattern sizes.
- The Brute Force algorithm, while the simplest, demonstrates the need for more advanced algorithms in scenarios involving larger texts and patterns.

6. Appendices

Graphs:

fig: pattern Length 1]

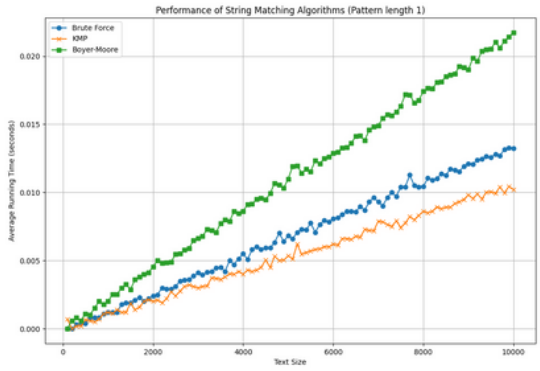

fig: pattern Length 3]

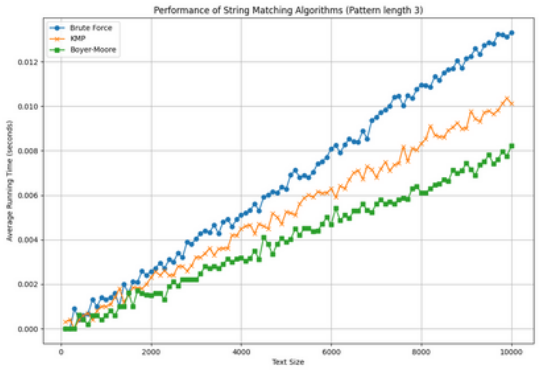

fig: pattern Length 5]

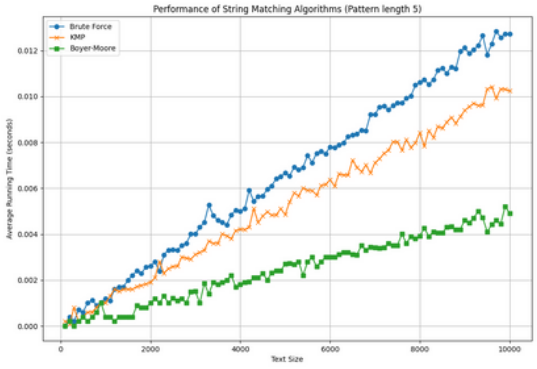

fig: pattern Length 10]