Implementing String Matching Algorithm to Trace Song Identities from fragment of lyrics

Albert - 13522081
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail (gmail): albert.choe73@gmail.com

***Abstract*—Identifying songs from fragments of lyrics is a significant challenge in the field of music information retrieval. This paper explores the implementation of string matching algorithms to address this problem. Specifically, the Knuth-Morris-Pratt (KMP), Boyer-Moore, and Brute Force algorithms are utilized to trace song identities from partial lyrics. The methodology includes gathering song data from Spotify and lyrics from Genius, preprocessing the lyrics, and applying the string matching algorithms to identify songs based on user-provided lyric fragments. Experimental results demonstrate the efficiency and accuracy of each algorithm, highlighting their strengths and limitations in practical applications. The findings contribute to the development of robust music identification systems that can effectively handle large datasets and diverse lyrical content.**

Keywords—string matching, KMP algorithm, Boyer-Moore algorithm, song identification, music information retrieval

I. Introduction

In the digital age, the ability to access and enjoy music has become easier, thanks to the advent of streaming services and extensive digital libraries. These platforms offer users a seemingly infinite collection of songs from various genres, artists, and eras. Music plays a crucial role in our lives, providing entertainment, emotional expression, and a means of cultural connection. However, with such an expansive catalog, identifying a specific song based on limited information, such as a fragment of lyrics, remains a significant challenge. This problem is particularly relevant in scenarios where users remember only a portion of the lyrics but are unable to recall the song title or the artist's name.

The proliferation of music content has necessitated the development of efficient and accurate song identification systems. Traditional methods of song identification, such as manual searching or relying on expert knowledge, are impractical given the vast amount of available music. Consequently, the field of music information retrieval (MIR) has emerged, focusing on leveraging computational techniques to solve this problem. Among the diverse approaches in MIR, string matching algorithms have emerged as a powerful tool for identifying songs from partial lyrics. String matching involves finding occurrences of a substring (pattern) within a larger string (text), making it an ideal technique for matching lyric fragments to complete song lyrics.

The need for effective song identification systems is driven by various use cases. For instance, individuals often experience moments where they hear a song in a public place or on the radio but can only recall a few words. In such situations, being able to quickly and accurately identify the song enhances the user experience and satisfaction. Additionally, music streaming services can benefit from improved recommendation systems that utilize partial lyric matching to suggest songs based on user input. This capability not only enriches the user's interaction with the service but also promotes the discovery of new music.

This paper focuses on the implementation and evaluation of three well-known string matching algorithms: the Knuth-Morris-Pratt (KMP) algorithm, the Boyer-Moore algorithm, and the Brute Force algorithm. Each of these algorithms offers unique advantages and has been extensively studied in the context of computer science and text processing. The approach includes collecting song data from Spotify using its API and fetching the corresponding lyrics from Genius, followed by preprocessing the lyrics to enhance the accuracy of the string matching algorithms.

II. Limitation

Despite the promising results, this paper have several limitations. The dataset, while diverse, is limited in size and may not fully represent the global music catalog, affecting the generalizability of the findings. The accuracy of the lyrics obtained from Genius is crucial, and any errors or discrepancies can impact the identification process. The preprocessing step, which involves cleaning and tokenizing the lyrics, might alter the lyrics' structure and affect the match accuracy. Additionally, the performance of the string matching algorithms may vary with very short lyric fragments or highly repetitive text. Future work should address these limitations by using larger, more comprehensive datasets, improving preprocessing techniques, and exploring enhancements to the

algorithms to better handle short and repetitive lyric fragments.

III. Theoretical Basis

A. Strings

In computer science, a string is a sequence of characters used to represent text. Each character in a string can be a letter, digit, punctuation mark, or any other symbol, and the length of a string is determined by the number of characters it contains.

Strings are used extensively in programming languages for handling input and output operations, constructing messages, and creating user interfaces. Common operations on strings include concatenation (joining two strings together), substring extraction (retrieving a portion of the string), and searching for patterns within the string. These operations are essential for tasks such as parsing text, formatting data, and performing text analysis.

One key property of strings in many programming languages, such as Python and Java, is their immutability. This means that once a string is created, its content cannot be changed. Any operation that appears to modify a string actually creates a new string with the desired changes. Immutability ensures that strings are thread-safe and can be used reliably in concurrent programming environments, preventing issues related to data corruption or unexpected modifications.

In different programming languages, strings are represented and managed in various ways. For example, in C, a string is typically an array of characters terminated by a null character (\0), whereas in higher-level languages like Python, strings are objects with built-in methods for manipulation and processing. This abstraction in higher-level languages simplifies string operations and enhances productivity.

In the context of string processing, the concepts of **prefix** and **suffix** are fundamental:

1. Prefix

   A prefix of a string is any leading contiguous portion of that string. For instance, given the string "algorithm," its prefixes include "", "a", "al", "alg", "algo", "algor", "algori", "algorit", "algorith", and "algorithm." Each of these prefixes starts from the beginning of the string and includes progressively more characters until the entire string is included. Prefixes are critical in various string processing tasks, such as searching and pattern matching, where the initial segments of the string are compared to determine matches.

2. Suffix

   A suffix of a string is any trailing contiguous portion of that string. Using the same example string "algorithm," its suffixes include "", "m", "hm", "ithm", "rithm", "orithm", "gorithm", "lgorithm", and "algorithm." Each suffix starts from some position within the string and extends to the end of the string. Suffixes are essential in various algorithms, especially those involving searching and sorting operations. In pattern matching, suffixes can be used to efficiently skip sections of the text that do not match the pattern, thus enhancing the performance of the algorithm.

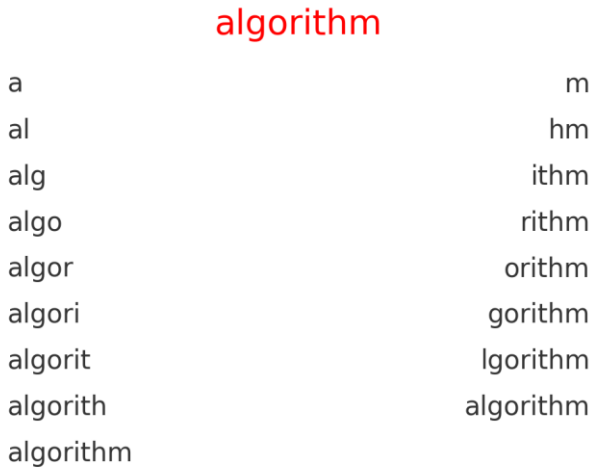

Fig 1. Prefix and Suffix

Source : Author's personal documentation

B. Sring Matching

String matching, also known as pattern matching, is a fundamental problem in computer science where the objective is to find all occurrences of a substring (referred to as the pattern) within a larger string (referred to as the text). It is an algorithm used to find occurrences of a short string within a long string. This problem is integral to various applications, including text search engines, DNA sequence analysis, data compression, and, in this project, music information retrieval. The process involves comparing the pattern to the text and identifying positions where the pattern matches the text exactly.

Generally, there are 3 main algorithms for performing string matching :

1. Brute Force Algorithm

   Brute force algorithm is the simplest method for string matching. It operates by sliding the pattern over the text one character at a time and checking for a match at each position. This is done by comparing each character of the pattern to the corresponding character in the text until either a mismatch is found or the entire pattern is matched.

   Algorithm Steps:

   1. Align the pattern at the start of the text.

2. Compare the pattern to the text character by character.

3. If all characters match, record the starting position.

4. Shift the pattern one position to the right and repeat the comparison until the end of the text is reached.

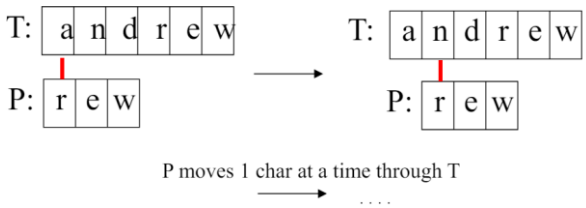

Fig 2. Brute Force Algorithm

Source : https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Pencocokan-string-2021.pdf .Accessed

While this method is straightforward and easy to implement, it is not efficient for large texts or long patterns. The time complexity of the brute force algorithm is O(n*m), where n is the length of the text and m is the length of the pattern. This inefficiency arises because the algorithm does not take advantage of any information gained from previous comparisons, leading to redundant checks.

For example, if the text is "ABABDABACDABABCABAB" and the pattern is "ABABCABAB", the brute force approach will start comparing the pattern with the text from the first character of the text, then shift one character to the right, and repeat the process until it finds a match or reaches the end of the text. For each possible starting position in the text, the algorithm checks if the pattern matches the substring of the text starting from that position. This involves comparing each character of the pattern with the corresponding character in the text. If a mismatch is found, the pattern is shifted one position to the right, and the comparison starts again from the first character of the pattern. This process continues until either a match is found or the end of the text is reached. This method can become extremely slow if the text or pattern is large.

2. Knuth-Morris-Pratt (KMP) Algorithm

The Knuth-Morris-Pratt (KMP) algorithm is an improvement over the naive approach, designed to reduce the number of comparisons by preprocessing the pattern. The key idea behind KMP is to preprocess the pattern to create an array of longest prefix suffix (LPS) values. These values are used to skip unnecessary comparisons during the search phase.

Algorithm Steps:

1. Preprocess the pattern to create the LPS array.

2. Use the LPS array to skip sections of the text that have already been matched.

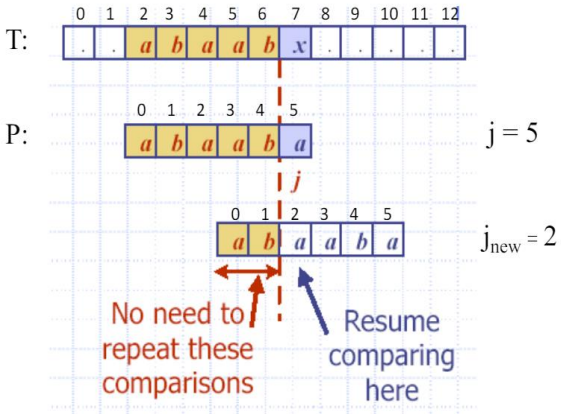

Fig 3. KMP Algorithm

Source : https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Pencocokan-string-2021.pdf .Accessed

The LPS array indicates the longest proper prefix of the pattern which is also a suffix. During the search phase, if a mismatch occurs after some matches, the algorithm uses the LPS array to determine how far the pattern can be shifted to avoid rechecking characters that are known to match.

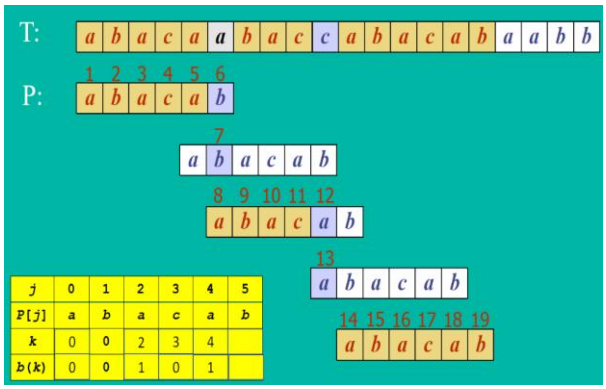

Fig 4. KMP Algorithm with Border Function

Source : https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Pencocokan-string-2021.pdf .Accessed

For instance, consider the pattern "ABABCABAB". The LPS array for this pattern is [0, 0, 1, 2, 0, 1, 2, 3, 4], indicating the lengths of the longest proper prefix which is also a suffix for each position in the pattern. Using this array, the KMP algorithm can efficiently skip sections of the text, reducing the overall number of comparisons.

The time complexity of KMP is O(n + m), where n is the length of the text and m is the length of the pattern. This makes it significantly more efficient than the brute force algorithm, especially for large texts and patterns.

3. Boyer-Moore Algorithm

The Boyer-Moore algorithm is one of the most efficient string matching algorithms, known for its practical performance on large texts. It uses two key heuristics: the looking-glass technique and the character-jump technique, which correspond to the good suffix and bad character rules commonly mentioned in other contexts.

1. **Looking-Glass Technique**

   This heuristic involves matching the pattern from right to left, starting from the end of the pattern. If a mismatch occurs, the algorithm uses the information gained from the mismatch to skip sections of the text that cannot contain the pattern, thus reducing the number of comparisons.

2. **Character-Jump Technique**

   When a mismatch occurs, this technique determines how far to shift the pattern based on the character in the text that caused the mismatch. The shift is calculated to align the last occurrence of the mismatched character in the pattern with its position in the text. If the character does not appear in the pattern, the pattern is shifted past the mismatched character entirely.

   The calculation of the jump is as follows: The algorithm preprocesses the pattern to create a "last occurrence table" which records the last index of each character in the pattern. During the search, if a mismatch occurs and the mismatched character is in the text but not in the pattern, the pattern is shifted to the right by the full length of the pattern. If the mismatched character does appear in the pattern, the pattern is shifted to align the last occurrence of the mismatched character in the pattern with its position in the text. This allows the algorithm to make large jumps over sections of text that cannot contain the pattern, thereby skipping unnecessary comparisons.

Algorithm Steps:

1. Preprocess the pattern to create the last occurence table.
2. Align the pattern at the start of the text.
3. Compare the pattern from right to left.
4. Use the table to skip sections of the text that cannot match the pattern.

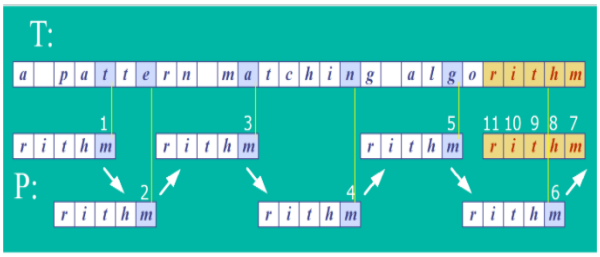

Fig 5. BM Algorithm

Source : https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Pencocokan-string-2021.pdf .Accessed

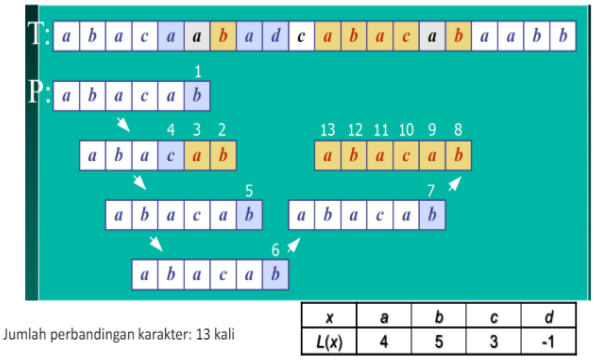

Fig 6. BM Algorithm with Last Ocurrence Table

Source : https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Pencocokan-string-2021.pdf .Accessed

The Boyer-Moore algorithm has an average-case time complexity of O(n/m), where n is the length of the text and m is the length of the pattern. This efficiency is achieved through its ability to skip large sections of the text. The use of the looking-glass technique and the character-jump technique allows the algorithm to make large jumps in the text, avoiding unnecessary comparisons. However, its performance can degrade in specific cases, such as with small alphabets and patterns with frequent characters.

In summary, the Boyer-Moore algorithm combines the looking-glass technique and the character-jump technique to efficiently match patterns within a text. By using the last occurrence table, it can skip sections of the text that do not match, significantly improving performance over brute force string matching approaches. This makes the Boyer-Moore algorithm highly effective for practical applications, especially in large texts.

IV. IMPLEMENTATION

A. *Data Sample*

To demonstrate the efficacy of string matching algorithms in identifying songs from lyric fragments, a comprehensive dataset is required. This dataset is constructed using song metadata and lyrics sourced from popular platforms. The following sections elaborate on the process of data collection, the specifics of the gathered data, and the rationale behind choosing these data sources.

Data was collected from Spotify and Genius, two leading platforms in the music streaming and lyrics domain, respectively. Spotify offers extensive metadata for songs, including attributes such as track name, artist name, album details, and popularity metrics. Genius provides accurate and detailed lyrics for a wide range of songs. The integration of these two sources allows for the creation of a rich dataset suitable for testing string matching algorithms.

- Spotify API

  The Spotify API is a vital tool for gathering the top 50 songs from a specific playlist. This playlist is curated to reflect current popular tracks, ensuring a broad representation of genres and artists. Authentication with the Spotify API is performed using the client credentials flow, which requires a client ID and client secret to obtain an access token. This access token allows the author to make authorized requests to the Spotify API.

  For each song retrieved from the Spotify playlist, the following details are collected:

  - **Track Name**: The official name of the song.
  - **Artist Name**: The primary artist or band performing the song.
  - **Popularity**: A numerical value provided by Spotify indicating the song's popularity.
  - **Spotify URL**: A direct link to the song on the Spotify platform.
  - **Track ID**: A unique identifier for the track on Spotify.
  - **Album Name**: The name of the album that includes the song.
  - **Release Date**: The release date of the album or single.
  - **Genres**: Musical genres associated with the song or artist, which are obtained by querying the artist's information.

- **Genius API**:

  To complement the song data obtained from Spotify, the Genius API is utilized to fetch the lyrics for these songs. For each track, a search query is constructed using the track name and artist name. If a match is found, the lyrics are retrieved from the song's Genius page. This process ensures that the lyrics are accurate and up-to-date.

  The combination of data from Spotify and lyrics from Genius forms a comprehensive dataset that includes not only the song metadata but also the lyrical content, which is essential for the subsequent string matching analysis. The detailed information gathered through these APIs allows for a robust examination of string matching algorithms in identifying songs from fragments of lyrics.

```python
def get_spotify_top_50(access_token):
    headers = {"Authorization": f"Bearer {access_token}"}
    playlist_id = '37i9dQZEVXbNG2KDcFcKOF'  # Top 50 playlist ID
    url = f"https://api.spotify.com/v1/playlists/{playlist_id}/tracks"

    response = requests.get(url, headers=headers)
    if response.status_code == 200:
        tracks_data = response.json().get('items', [])
        tracks_info = []
        for track in tracks_data:
            track_info = track.get('track', {})
            track_id = track_info.get('id')
            track_details = {
                'track_name': track_info.get('name'),
                'artist_name': track_info.get('artists', [{}])[0].get('name'),
                'popularity': track_info.get('popularity'),
                'spotify_url': track_info.get('external_urls', {}).get('spotify', ''),
                'track_id': track_id,
                'album_name': track_info.get('album', {}).get('name', ''),
                'release_date': track_info.get('album', {}).get('release_date', ''),
                'genres': fetch_genres(track_id, access_token)
            }
            tracks_info.append(track_details)
        return tracks_info
    else:
        print(
            f"Failed to fetch tracks: {response.status_code} - {response.text}")
        return []
```

Fig 7. Fetching Spotify API

Source : Author's personal documentation

In summary, the data collection methodology involves using the Spotify API to retrieve detailed song metadata and the Genius API to acquire accurate song lyrics. This dual approach ensures a rich dataset that supports the exploration and implementation of various string matching algorithms. The data obtained is stored in a JSON file, providing a structured and accessible format for further processing. This dataset, comprising the top 50 current popular songs from Spotify, forms the basis for the subsequent steps in this research.

B. Preprocessing

After collecting and storing the data in a JSON file, the next crucial step is preprocessing the lyrics to ensure that they are in a suitable format for the string matching algorithms. Preprocessing involves normalizing the text and removing unnecessary characters to facilitate accurate and efficient pattern matching.

The first step in preprocessing is normalizing the text. This involves converting all characters to lowercase to ensure uniformity, as string matching algorithms are case-sensitive. Additionally, numbers are removed since they are not typically relevant to the lyrical content and could introduce noise into the data.

```python
def preprocess(text):
    # Original text for exact matching
    original_text = text.lower()

    # Cleaned text
    cleaned_text = text.lower()
    cleaned_text = re.sub(r'[^\w\s]', '', cleaned_text)  # remove punctuation
    original_text = re.sub(r'[^\w\s]', '', cleaned_text)
    cleaned_text = re.sub(r'\d+', '', cleaned_text)  # remove numbers
    tokenizer = RegexpTokenizer(r'\w+')
    tokens = tokenizer.tokenize(cleaned_text)  # tokenize
    tokens = [word for word in tokens if word not in stopwords.words(
        'english')]  # remove stopwords
    cleaned_text = ' '.join(tokens)

    return original_text, cleaned_text
```

Fig 7. Preproccess Algorithm

Source : Author's personal documentation

In this preprocessing function, the text is first converted to lowercase. Then, a regular expression is used to remove any numerical digits. The RegexpTokenizer is utilized to split the text into individual words (tokens). Finally, the cleaned tokens are recombined into a single string.

For each song in the dataset, the lyrics are preprocessed using the function defined above. This step ensures that the lyrics are in a standardized format, making them suitable for input into the string matching algorithms.

Preprocessing is a critical step in the implementation process. It ensures that the lyrics are in a consistent format, which is essential for accurate string matching. By removing extraneous characters and standardizing the text, the algorithms can focus on the meaningful content of the lyrics, improving their performance and accuracy.

C. String Matching Algorithm

The core of this project involves applying and comparing different string matching algorithms to identify songs from lyric fragments. This section details the implementation of three prominent string matching algorithms: Knuth-Morris-Pratt (KMP), Boyer-Moore, and Brute Force. Each algorithm is implemented and evaluated for its effectiveness and efficiency in matching lyric fragments to full song lyrics.

1. Knuth-Morris-Pratt (KMP) Algorithm

The KMP algorithm improves the efficiency of the search process by preprocessing the pattern to create a longest prefix suffix (LPS) array. This array helps the algorithm to avoid unnecessary comparisons by allowing it to skip sections of the text where mismatches occur.

```python
# Knuth-Morris-Pratt (KMP) Algorithm
def computeLPSArray(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(pattern, text):
    M = len(pattern)
    N = len(text)
    lps = computeLPSArray(pattern)
    i = j = 0
    results = []
    while i < N:
        if pattern[j] == text[i]:
            i += 1
            j += 1
        if j == M:
            results.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 results
```

Fig 8. KMP Algorithm

Source : Author's personal documentation

In this implementation, the computeLPSArray function generates the LPS array for the given pattern. The KMP_search function uses this array to efficiently find the pattern within the text.

2. Boyer-Moore Algorithm

The Boyer-Moore algorithm uses two heuristics: the looking-glass technique and the character-jump technique. These heuristics allow the algorithm to skip sections of the text that cannot match the pattern, thus reducing the number of comparisons.

```python
def bad_character_table(pattern):
    table = {}
    for i in range(len(pattern) - 1):
        table[pattern[i]] = len(pattern) - i - 1
    return table


def boyer_moore_search(pattern, text):
    M = len(pattern)
    N = len(text)
    if M > N:
        return []

    bad_char_table = bad_character_table(pattern)
    s = 0
    results = []
    while s <= N - M:
        j = M - 1
        while j >= 0 and pattern[j] == text[s + j]:
            j -= 1
        if j < 0:
            results.append(s)
            s += M
        else:
            s += max(1, bad_char_table.get(text[s + j], M))
    return results

```

Fig 9. BM Algorithm

Source : Author's personal documentation

In the Boyer-Moore algorithm, the bad_character_table(Last Ocurrence Table) function creates a table that stores the rightmost occurrence of each character in the pattern. The boyer_moore_search function uses this table to determine how far to shift the pattern when a mismatch occurs.

3. Brute Force Algorithm

The Brute Force algorithm is the simplest and least efficient of the three. It checks each possible position in the text for a match by comparing the pattern to the substring of the text at that position.

```python
def brute_force_search(pattern, text):
    M = len(pattern)
    N = len(text)
    results = []
    for i in range(N - M + 1):
        j = 0
        while j < M and text[i + j] == pattern[j]:
            j += 1
        if j == M:
            results.append(i)
    return results
```

Fig 10. Brute Force Algorithm

Source : Author's personal documentation

In the Brute Force algorithm, the brute_force_search function iterates through the text, comparing the pattern to each substring of the text to find matches.

Combining and Comparing the Algorithms: The final step involves integrating these algorithms into a unified framework to search for lyric fragments within the dataset and compare their performance.

```python
def find_songs(fragment, data, search_function):
    results = []
    start_time = time.time()
    for track in data:
        if search_function(fragment, track['original_lyrics']) or search_function(fragment, track['cleaned_lyrics']):
            results.append(track)
    end_time = time.time()
    return results, end_time - start_time


def regex_search(pattern, text):
    return bool(re.search(pattern, text, re.IGNORECASE))


def display_results(results, algorithm_name, search_time):
    if not results:
        print(f"No matches found using {algorithm_name}.")
    else:
        for track in results:
            print(
                f"Track: {track['track_name']}, Artist: {track['artist_name']}, "
                f"Album: {track['album_name']}, Release Date: {track['release_date']}, "
                f"Genres: {', '.join(track['genres'])}, Spotify URL: {track['spotify_url']}"
            )
    print("\n")


if __name__ == "__main__":
    fragment = get_user_input()
    fragment_cleaned = ' '.join([word for word in RegexpTokenizer(
        r'\w+').tokenize(fragment.lower()) if word not in stopwords.words('english')])

    kmp_results, kmp_time = find_songs(fragment, data, KMP_search)
    brute_force_results, brute_force_time = find_songs(
        fragment, data, brute_force_search)
    boyer_moore_results, boyer_moore_time = find_songs(
        fragment, data, boyer_moore_search)
    regex_results, regex_time = find_songs(fragment, data, regex_search)

    display_results(kmp_results, "KMP Algorithm", kmp_time)

    print("Efficiency Comparison:")
    print(f"KMP Algorithm: {kmp_time:.4f} seconds")
    print(f"Brute Force Algorithm: {brute_force_time:.4f} seconds")
    print(f"Boyer-Moore Algorithm: {boyer_moore_time:.4f} seconds")
    print(f"Regex Search: {regex_time:.4f} seconds")
```

Fig 11. Main Algorithm

Source : Author's personal documentation

In the combined implementation, the find_songs function applies the specified string matching algorithm to search for the lyric fragment in the dataset. The display_results function presents the results and the efficiency of each algorithm.

By clearly detailing each step of the implementation, this section demonstrates how the collected data is processed and analyzed using different string matching algorithms. The comparison of these algorithms provides insights into their performance and suitability for identifying songs from fragments of lyrics.

V. TESTING AND ANALYSIS

The testing results demonstrated the capabilities of each algorithm in identifying songs from lyric fragments. Below are examples of the test cases and their outcomes.

1. Test Case 1

Fig 12. Test Case 1

2. Test Case 2

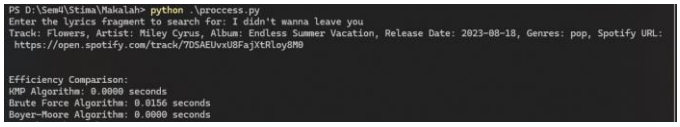

Fig 13. Test Case 2

The analysis of the testing results provides insights into the performance and suitability of each algorithm for the task of song identification from lyric fragments.

Knuth-Morris-Pratt (KMP) Algorithm: The KMP algorithm demonstrated consistent performance with quick search times across various test cases. Its preprocessing phase, which constructs the longest prefix suffix (LPS) array, enables efficient skipping of comparisons, making it suitable for large texts with frequent patterns. The KMP algorithm is particularly effective in situations where the pattern contains repetitive sub-patterns, as it avoids redundant comparisons by using the precomputed LPS array. This characteristic makes it robust for matching lengthy lyric fragments efficiently.

Boyer-Moore Algorithm: The Boyer-Moore algorithm outperformed the other algorithms in several test cases, particularly in terms of search time. The use of the looking-glass and character-jump techniques allows for significant reductions in comparisons by leveraging the last occurrence table. This makes Boyer-Moore highly efficient, especially for longer patterns and texts with fewer repeating characters. However, its performance can degrade with small alphabets or highly repetitive patterns. The Boyer-Moore algorithm's ability to skip large portions of the text by using the precomputed last occurrence table for characters in the pattern contributes to its superior performance in most practical scenarios.

Brute Force Algorithm: The Brute Force algorithm, while simple and easy to implement, was the least efficient in terms of search time. It checks each possible position in the text for a match, leading to higher computational costs, especially for longer texts and patterns. Despite its simplicity, it serves as a baseline for comparison with more sophisticated algorithms. The Brute Force approach is straightforward, as it performs a character-by-character comparison for each possible position in the text. While this ensures accuracy, it results in significantly higher time complexity, making it less suitable for large datasets.

Efficiency Comparison: The recorded times indicate that Boyer-Moore often has the fastest execution, followed by KMP and then Brute Force. However, the differences in performance are more pronounced with longer texts and patterns. For shorter fragments, the times may be closer, as seen in some test cases. The Boyer-Moore algorithm's efficiency in skipping irrelevant portions of the text contributes to its superior performance, while the KMP algorithm also performs well due to its efficient handling of repetitive patterns. The Brute Force algorithm's higher time complexity makes it less competitive in larger datasets, but it remains a viable option for smaller or simpler use cases.

VI. CONCLUSION

In conclusion, this study demonstrates the application of string matching algorithms—Knuth-Morris-Pratt (KMP), Boyer-Moore, and Brute Force—in identifying songs from lyric fragments. The integration of Spotify and Genius APIs enables the creation of a rich dataset, facilitating comprehensive testing and analysis. The results highlight the efficiency and practicality of the Boyer-Moore algorithm, with KMP also showing strong performance. The findings contribute to the development of advanced music identification systems, capable of handling large datasets and diverse lyrical content, thereby improving user experience in music discovery and retrieval.

The study's findings indicate that the Boyer-Moore algorithm is particularly well-suited for practical applications in music information retrieval due to its ability to efficiently skip large sections of the text. The KMP algorithm also demonstrates strong performance, making it a reliable alternative for scenarios involving repetitive patterns. The Brute Force algorithm, while less efficient, provides a simple and straightforward method for smaller datasets or less demanding applications.

Future research could explore further optimization techniques for these algorithms, as well as their application in other domains of text and pattern matching. Additionally, expanding the dataset to include a wider range of songs and genres would enhance the generalizability of the findings, providing deeper insights into the performance of string matching algorithms in diverse contexts.

GITHUB LINK

https://github.com/AlbertChoe/Song_String_Matching

VIDEO LINK AT YOUTUBE

https://youtu.be/PXBKl3IDbXg

ACKNOWLEDGMENT

I would like to extend my deepest gratitude to God for providing me with the strength, perseverance, and wisdom to complete this research paper and project. Without His guidance and blessings, this work would not have been possible. I would also like to express my sincere appreciation to my professors, Dr. Ir. Rinaldi Munir, M.T., Dr. Ir. Rila Mandala, and Dr. Nur Ulfa Maulidevi. Their exceptional teaching and unwavering support in the IF2211 Algorithm Strategies course have been invaluable. Their profound knowledge and insightful guidance have greatly contributed to my understanding and application of algorithmic strategies, which have been fundamental to the success of this project. Their encouragement and constructive feedback have been instrumental in shaping this research paper, and for that, I am truly grateful.

REFERENCES

[1] R. Munir, 'IF2211 Strategi Algoritma - Semester II Tahun 2023/2024', https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Pencocokan-string-2021.pdf Diakses pada 11 Juni 2024.

[2] geeksforgeeks.org, 'What is String – Definition & Meaning', https://www.geeksforgeeks.org/what-is-string/ Diakses pada 11 Juni 2024.

[3] geeksforgeeks.org, 'Boyer Moore Algorithm for Pattern Searching', https://www.geeksforgeeks.org/boyer-moore-algorithm-for-pattern-searching/ Diakses pada 11 Juni 2024.

[4] geeksforgeeks.org, 'KMP Algorithm for Pattern Searching', https://www.geeksforgeeks.org/kmp-algorithm-for-pattern-searching/ Diakses pada 11 Juni 2024.

[5] geeksforgeeks.org, 'KMP Algorithm for Pattern Searching', https://www.geeksforgeeks.org/kmp-algorithm-for-pattern-searching/ Diakses pada 11 Juni 2024.

STATEMENT

I hereby declare that the paper I have written is my own work and is not a summary or translation or someone else's paper, and it is not plagiarized.

Bandung, 12 June 2024

Albert (13522081)