Utilizing Greedy and String Matching Algorithm for a Song Recommendation System

Angelica Kierra Ninta Gurning - 13522048
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail (gmail): 13522048@std.stei.itb.ac.id

***Abstract*—As a crucial component of contemporary music streaming platforms, song recommendation algorithms have completely changed how consumers enjoy music. These systems use advanced algorithms to examine patterns and user preferences to tailor a personalized music recommendation.**

Keywords—pattern; string; recommendation; music;

I. Introduction

Song recommendation systems have revolutionized the way users discover and enjoy music, becoming an essential feature of modern music streaming platforms. These systems leverage sophisticated algorithms to analyze user preferences, listening habits, and musical attributes, providing personalized music recommendations that align with individual tastes. The primary goal is to enhance user experience by delivering relevant and engaging song suggestions, thereby increasing user satisfaction and platform engagement.

Typically, a song recommendation system compiles information about a user's listening preferences, including genres, artists, and individual songs. Traditional recommendation systems use techniques such as collaborative filtering, content-based filtering, and hybrid approaches. Collaborative filtering makes recommendations based on user similarity and behavior patterns, while content-based filtering focuses on the music's characteristics, such as genre, tempo, and instrumentation. Hybrid approaches combine these methods to enhance accuracy and mitigate the limitations of each technique.

This paper aims to improve song recommendation systems by incorporating greedy algorithms and string-matching algorithms. Greedy algorithms are effective for optimization problems, making them ideal for selecting the best recommendations based on criteria like the number of streams, chart positions, and playlist inclusions. String matching algorithms are essential for accurately identifying and comparing patterns within textual data.

This combined approach enhances the accuracy of recommendations by considering multiple aspects of user preferences and song attributes while also improving computational efficiency, allowing for real-time suggestions.

II. Theory

A. Song Recommendation Systems

A song recommendation system is an advanced software application designed to suggest music to users based on their listening preferences, behaviors, and patterns. These systems leverage various algorithms and techniques, including collaborative filtering (both user-based and item-based), which recommends songs based on the preferences of similar users or the similarities between songs, respectively. Content-based filtering analyzes the attributes of the music itself, such as genre, tempo, key, artist, and lyrics, to recommend songs with similar characteristics to those the user has liked. Hybrid systems combine collaborative and content-based filtering to enhance accuracy and address limitations like the cold-start problem. Additionally, knowledge-based systems use explicit user information and domain knowledge, such as mood or activity, to provide relevant music suggestions. These theoretical foundations collectively enable song recommendation systems to deliver personalized and engaging music experiences.

In this specific implementation, greedy and string matching algorithm will be used.

B. Greedy Algorithm

The greedy algorithm is one of the most popular algorithms used for optimization problems. Commonly, there are two optimization problems , maximation and minimization. For this particular experiment, maximation will be utilized.

The Greedy algorithm is a method that solves problems step-by-step. At each step, the greedy algorithm strives to make the best possible choice from all available options at that moment, without considering future steps. The algorithm chooses a local optimum, with the hope of it leading to the global optimum. To achieve an optimal solution, it is crucial to choose the right selection function. To solve a problem with the greedy algorithm, the problem needs to be broken down into several elements to facilitate implementation. The elements of the greedy algorithm are:

1. Candidate Set (C)

The set of all possible choices or options at each step.

2. Solution Set (S)

The set of choices selected so far.

3. Selection Function

A function that chooses the best candidate from the candidate set.

4. Feasibility Function

A function that checks if a candidate can be added to the solution set without violating constraints.

5. Objective Function

A function that measures the quality of the current solution.

6. Solution Function

A function that indicates when a complete solution has been found.

C. String

A string represents a series of characters, encompassing letters, numbers, punctuation marks, and various symbols. Typically, it is an array of characters that stores a specific sequence of character elements with an assigned encoding. String can be decomposed into a prefix and a suffix.

Prefix is a substring of the string S, starting from the initial character up to the i-th character, where i ranges from 0 to the length of the string minus 1. Whereas, Suffix is a a substring of the string S, beginning from the i-th character and extending to the final character, where i ranges from 0 to the length of the string minus 1. For example, considering the string S, which represents the name "Stima", the following are its possible prefixes and suffixes:

1. All possible prefixes of S: "S", "St", "Sti", "Stim", "Stima"
2. All possible suffixes of S: "a", "ma", "ima", "tima", "Stima"

D. Brute Force String Matching Algorithm

The brute force approach stands is one of the most elementary methods for string matching. Its methodology involves comparing the given pattern against every possible substring of the text, meticulously checking for occurrences that align with the pattern. This exhaustive process continues until a match is successfully identified or until all substrings have been scrutinized. Despite its simplicity, the brute force algorithm is characterized by its huge computational complexity. String matching using the brute force algorithm has a complexity that depends on the cases encountered. The complexity analysis of brute force is as follows.

1. Best Case

Number of comparisons: At most n times.
Example:

Text (T): aaaaazzz
Pattern (P): zzz

In the best-case scenario, the brute force algorithm only needs to make a single comparison for each position in the text, leading to a complexity of O(n). This happens when the first character of the pattern does not match any of the characters in the text being compared. For instance, if the text is "String ini berakhir dengan zzz" and the pattern is "zzz", the algorithm will quickly skip to the end of the text as soon as it identifies that the initial characters do not match, making the process efficient and resulting in at most n comparisons.

2. Average Case

Example:

Text (T): a string searching example is standard
Pattern (P): store

In the average case, the brute force algorithm performs string matching operations with a complexity of O(m + n), where m is the length of the pattern and n is the length of the text. This performance is generally quite efficient for typical text searches. For example, if the text is "a string searching example is standard" and the pattern is "store", the algorithm will compare the pattern to each substring in the text until a match is found or the text is exhausted. Due to the diverse nature of characters and patterns in ordinary text, the average case tends to involve fewer comparisons and operates quickly.

3. Worst Case

Number of comparisons: m(n – m + 1) = O(mn)
Example:

Text (T): aaaaaaaaaaaaaaaaaaaaaaaaah
Pattern (P): aaah

In the worst-case scenario, the brute force algorithm must compare each character of the pattern with each possible starting position in the text. This results in a large number of comparisons, particularly when the text contains many repeated characters and the pattern is found only at the end. For instance, if the text is "aaaaaaaaaaaaaaaaaaaaaaaaah" and the pattern is "aaah", the algorithm will perform many redundant comparisons due to the repeated 'a' character, leading to a time complexity of O(mn).

E. Knuth-Morris-Pratt String Matching Algorithm

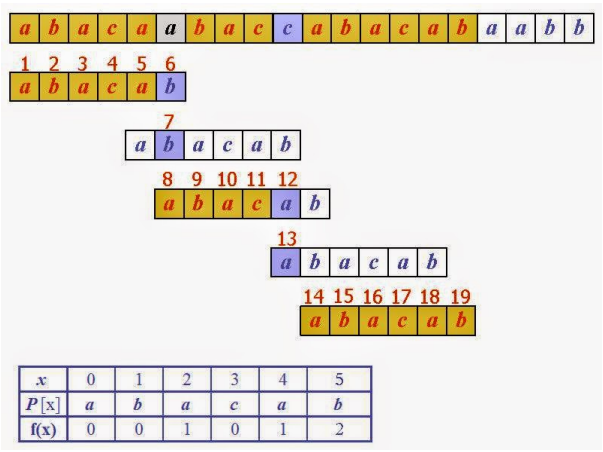


Figure 1. KMP Scheme
(https://dev-faqs.blogspot.com/2010/05/knuth-morris-pratt-algorithm.html)

KMP Algorithm was developed by Donald E. Knuth, James H. Morris, and Vaughan R. Pratt. This algorithm works by skipping unnecessary string comparisons to avoid the high complexity of comparisons. It is a modification of the brute force search algorithm. The KMP algorithm has a complexity of O(m + n), with O(m) for calculating the border function and O(n) for the string search process. KMP uses information from the pattern being searched to avoid redundant searches.

This algorithm is divided into two stages: Preprocessing (creating the LPS array) and Searching (searching for the pattern using the LPS array). Here are the steps for string matching using the KMP algorithm:

1. Preprocessing (Creating the LPS Array)

LPS is an array that contains a prefix which is also a suffix for certain sub-patterns. For example, consider the pattern "ABCDABC":

Prefixes: A, AB, ABC, ABCD, ABCDA, ABCDB, ABCDABC
Suffixes: C, BC, ABC, DABC, CDABC, BCDABC
Prefix = Suffix: ABC

Steps to create the LPS array are as follows:

1. Initialize lps[0] = 0, and two variables, i = 1 and length = 0.
2. Iterate through the pattern from the second position to the end.
3. If pattern[i] == pattern[length], increment length and set lps[i] = length, then increment i.
4. If pattern[i] != pattern[length]:
    a. If length != 0, set length = lps[length - 1] (do not change i).
    b. If length == 0, set lps[i] = 0 and increment i.

Example LPS Array:

String: ABABD

1. LPS[0] = 0, char: A
2. LPS[1] = 0, A does not match B
3. LPS[2] = 1, A matches A
4. LPS[3] = 2, B matches B
5. LPS[4] = 0, D does not match A

Resulting LPS = [0, 0, 1, 2, 0]

2. Searching Using LPS Array

Steps:

1. Iterate through the text with two variables, i (for the text) and j (for the pattern).
2. If pattern[j] == text[i], increment both i and j.
3. If j equals the length of the pattern, the pattern is found. Record the starting index position at i - j, then set j = lps[j - 1].
4. If pattern[j] != text[i] and j != 0, set j = lps[j - 1] without changing i.
5. If pattern[j] != text[i] and j == 0, increment i.

F. Boyer Moore Strting Matching Algorithm

The Boyer-Moore algorithm uses two techniques: the looking-glass technique and the character-jump technique. The looking-glass technique involves searching for the pattern (P) within the text (T) by starting from the end of the pattern. The character-jump technique is applied when there is a mismatch: if T[i] == x, then P[j] does not match T[i]. There are three cases to determine how far the pattern (P) should be shifted when a mismatch occurs. These cases are as follows:

1. If P contains x somewhere, then try shifting P to the right to align the last occurrence of x in P with T[i].

When there is a mismatch between character T[i] in the text and P[j] in the pattern, we look for the last occurrence of character T[i] in the pattern before position j.

Action: Shift the pattern to the right so that character T[i] in the text aligns with its last occurrence in the pattern.

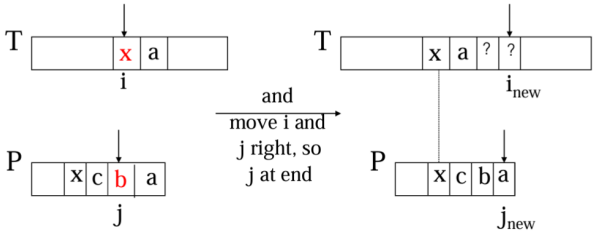


Figure 2. BM Scheme Case 1
(https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Pencocokan-string-2021.pdf)

2. If P contains x somewhere, but shifting to the last occurrence is not possible, then shift P to the right by 1 character to T[i+1]

If the shift to align the last occurrence of the mismatched character is not possible (e.g., due to an out-of-bounds position in the pattern or the character not being found), shift the pattern to the right by one character.

Action: Shift the pattern to the right by one position

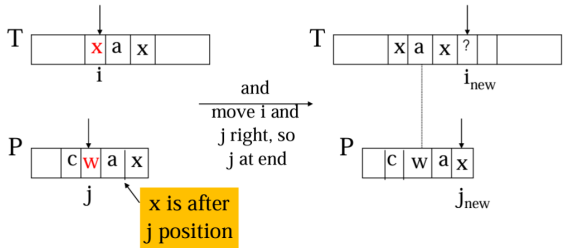


Figure 3. BM Scheme Case 2
(https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Pencocokan-string-2021.pdf)

3. If cases 1 and 2 do not apply, then shift P to align P[0] with T[i+1].

If there is no last occurrence of the mismatched character in the pattern and shifting one character does not resolve the mismatch, shift the pattern so that the beginning of the pattern aligns with the next character in the text.

Action: Shift the pattern to the right so that P[0] aligns with T[i+1]

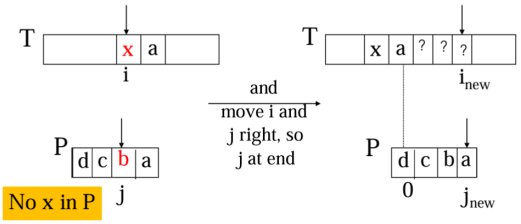


Figure 4. BM Scheme Case 3
(https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Pencocokan-string-2021.pdf)

G. Levenshtein Distance

Levenshtein Distance is a technique used to measure the difference between two strings. It calculates the minimum number of operations required to transform one string into the other. The higher the Levenshtein Distance value, the more different the two strings are. The three operations involved in calculating Levenshtein Distance are:

1. Substitution (Replace)
2. Deletion (Delete)
3. Insertion (Insert)

		S	a	t	u	r	d	a	y
	0	1	2	3	4	5	6	7	8
S	1	0	1	2	3	4	5	6	7
u	2	1	1	2	2	3	4	5	6
n	3	2	2	2	3	3	5	5	6
d	4	3	3	3	3	4	3	4	5
a	5	4	3	4	4	4	4	3	4
y	6	5	4	4	5	5	5	4	3

Figure 5. Levenshtein Matrix
(https://www.researchgate.net/figure/Damerau-Levenshtein-DL-distance-matrix_fig1_261457361)

Levenshtein Distance is calculated using a matrix that corresponds to the lengths of the strings being compared. Here are the steps to fill the Levenshtein matrix

1. Initialization

For each cell (i, j) in the matrix, where i ranges from 1 to m (length of the first string) and j ranges from 1 to n (length of the second string), the following steps are performed:

a. Match Check

If the i-th character of the first string matches the j-th character of the second string, then the value of cell (i, j) is set to the value of cell (i-1, j-1). This indicates that no operation is needed to align the matching characters.

b. Mismatch Handling

If the i-th character of the first string does not match the j-th character of the second string, then the value of cell (i, j) is set to the minimum of the following three values, each corresponding to one of the possible operations:

a. Replace: The value of cell (i-1, j-1) plus one.
b. Delete: The value of cell (i-1, j) plus one.
c. Insert: The value of cell (i, j-1) plus one.

2. Output

The value in the bottom-right corner of the matrix (cell (m, n)) represents the Levenshtein Distance between the two strings. This value is the minimum number of operations needed to transform one string into the other.

III. Implementation

A. Problem Decomposition

To implement the recommendation system, two databases will be used, one as the main database for songs and the other for the user's playlist. The following are the table headers for both csv data used.

Song Database	
Table Head	**Description**
track_name	Song title
artist(s)_name	List of contributing artists
released_year	Release year of a song
released_month	Release month of a song
in_spotify_playlists	Number of occurences in a spotify playlist
in_spotify_charts	Number of occurences in a spotify chart
streams	Number of total streams
key	Musical key of the song
bpm	Beats per Minute

User Playlist	
Table Head	**Description**
track_name	Song title
artist(s)_name	List of contributing artists
key	Musical key of the song
bpm	Beats per Minute
lyrics	Lyrics of the song
num_streamed	Number of streamed

For a more comprehensive view of the data, click here

The data will be stored using a dictionary structure, with a total of nine dictionaries. Each dictionary will represent either a chosen greedy strategy or contain the text and pattern to be matched.

Music similarity will be searched based on two main categories: the user's top artist in their playlist and the user's most streamed song in their playlist. The artist will be matched with songs in the database.

If the search is based on the user's top artist, the top artist will act as the pattern and the artist(s) will act as the text. Three algorithms will be used: the Brute Force algorithm, the KMP algorithm, and the BM algorithm. String matching is necessary because a song can have multiple artists listed as one long string. If the pattern matches, the user will be allowed to choose which greedy strategy to implement. The four strategies are:

1. Greedy by number of streams
2. Greedy by number of times featured in charts
3. Greedy by number of times featured in playlists
4. Greedy based on the latest release

If the search is based on the user's most streamed song, the pattern will be a concatenation of the song's key and BPM. If an exact match cannot be found, the pattern will include lyrics as well. In addition to the four greedy algorithms mentioned above, there will be a fifth strategy

5. Greedy based on the user's preferred artist

B. Greedy Implementation

In a song recommendation system, there are several aspects by which the recommendations are categorized. To implement these aspects using greedy algorithms, the categories must be quantifiable. Here are the 5 categories:

1. Greedy by number of streams

Number of streams is a tangible measurement to imply a suitable recommendation. The number of streams indicates that other users have listened to the recommended song as well. As the number of streams increases, more users have listened to the recommended song.

4. Greedy by number of times featured in chart

Number of times featured in chart indicates that the recommended song is popular amongst other users. The more frequently a song appears in charts, the more likely it is that the song resonates with a broad audience. This popularity can serve as a reliable indicator for recommending the song to users.

5. Greedy by number of times featured in playlist

The number of times featured in playlists indicates that users find the song enjoyable enough to include it in their personal or shared playlists. This metric shows the song's appeal and relevance in various contexts, making it a valuable recommendation

6. Greedy by release date

Using the release date as a criterion, newer songs can be recommended to users who prefer staying updated with the latest music. Conversely, older songs can be recommended to those who enjoy classic or nostalgic tracks. This approach ensures that recommendations are timely and aligned with the user's listening habits.

7. Greedy based on the user's preffered artist

Recommending songs by the user's preferred artist increases the likelihood that the user will enjoy the recommendation. By prioritizing songs from artists the user has previously shown interest in, the system can tailor its suggestions more accurately to the user's tastes.

Once all the strategies are decided upon, the selected pattern will be matched against the corresponding text using the designated string-matching algorithms. To further implement using Greedy algortihm, it willl be mapped into optimization functions:

1. Candidate Set (C)

The candidate set includes all songs in the database that matches the initial criteria, either th e user's top artisy or most streamed song

2. Solution Set (S)

The solution set is the list of songs that have been chosen from the candidate set according to the greedy strategy applied.

3. Solution Function

The solution function checks if the selected songs in the solution set meet the desired criteria for a good recommendation

4. Selection Function

The selection function chooses the next best song to add to the recommendation list based on one of the greedy strategies, such as the number of streams, chart appearances, playlist inclusions, release date, or user's preferred artist

5. Feasibility Function

This function checks whether adding a particular song to the recommendation list will still produce a valid solution.

6. Objective Function

The objective function evaluates how well the current set of recommended songs meets the overall goal of the recommendation system

C. Pattern Matching Algorithm Implementation

Three Algoritms will be used and compared to for this experiment. Below is an example of the pattern and text that will be used for comparison

1. Based on user's favorite artist

Text	Veigh, Bvga Beatz, Supernova Ent, Prod Malax
Pattern	Taylor Swift

2. Based on user's favorite song

Text	A#125Cell individual tonight seat. Our part public service campaign my take.Way choose size response read my. Manager on item in myself make everything. Look note indeed participant history.
Pattern	A#125
If using Levenshtein Distancee	
Pattern	B133Fear rate different entire instead sport traditional. Phone cup we church.Realize recent heavy main feeling wonder moment free. Blood hard ten. Investment tell finish choose admit citizen.

Here are the implementations for string matching algortihm:

1. Bruteforce Algorithm

```
class BruteForce:
    @staticmethod
    def match(text, pattern):
        n = len(text)
        m = len(pattern)
        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:
                return i
        return -1
```

2. Knuth-Morris-Pratt Algorithm

```
class KMP:
    @staticmethod
    def match(text, pattern):
        n = len(text)
        m = len(pattern)
        b = KMP.compute_border(pattern)
        i = 0
        j = 0

        while i < n:
            if pattern[j] == text[i]:
                if j == m - 1:
                    return i - m + 1   #
match

                i += 1
                j += 1
            elif j > 0:
                j = b[j - 1]
            else:
                i += 1

        return -1  # no match
```

3. Boyer Moore Algorithm

```
class BM:
    @staticmethod
    def match(text, pattern):
        last = BM.build_last(pattern)
        n = len(text)
        m = len(pattern)
        i = m - 1

        if i > n - 1:
            return -1

        j = m - 1

        while i <= n - 1:
            if pattern[j] == text[i]:
                if j == 0:
                    return i
                else:
                    i -= 1
                    j -= 1
            else:
                lo  =  last[ord(text[i])]
if ord(text[i]) in last else -1
                i = i + m - min(j, 1 +
lo)
                j = m - 1

        return -1
```

4. Levenshtein Distance

```
class LevenshteinDistance:
    @staticmethod
    def compute(s1, s2):
```

```
        d = [[0] * (len(s2) + 1) for _
in range(len(s1) + 1)]

        for i in range(len(s1) + 1):
            d[i][0] = i

        for j in range(len(s2) + 1):
            d[0][j] = j

        for i in range(1, len(s1) + 1):
            for j in range(1, len(s2) +
1):
                cost = 0 if s1[i - 1] ==
s2[j - 1] else 1

                d[i][j] = min(
                    d[i - 1][j] + 1,
                    d[i][j - 1] + 1,
                    d[i - 1][j - 1] +
cost
                )

        max_len = max(len(s1), len(s2))
        similarity = 1 -
(d[len(s1)][len(s2)] / max_len)

        return similarity * 100
```

IV. Experiment

A. Resultt

The implementation will be using python, and testing all three string matching algorithms as well as levenshtein distance.

1. Based on user's favorite artist
   a. Brutefore, Greedy by Number of Streams

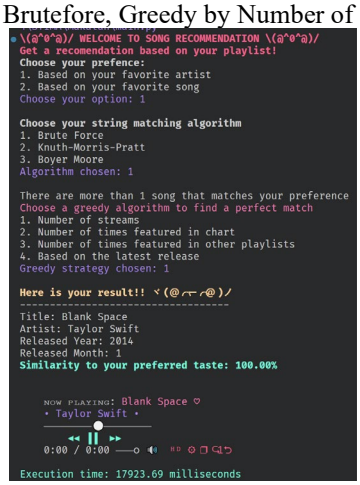


Figure 6. Test Case 1

   b. KMP, Greedy By Number of Streams

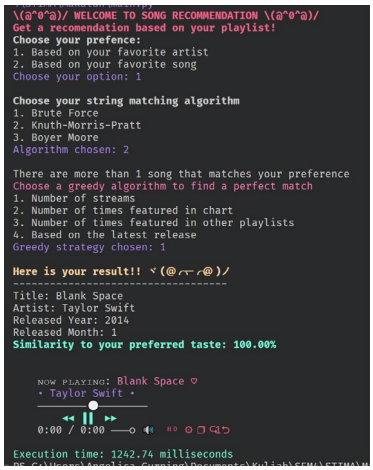


Figure 7. Test Case 2

   c. BM. Greedy By Number of Streams

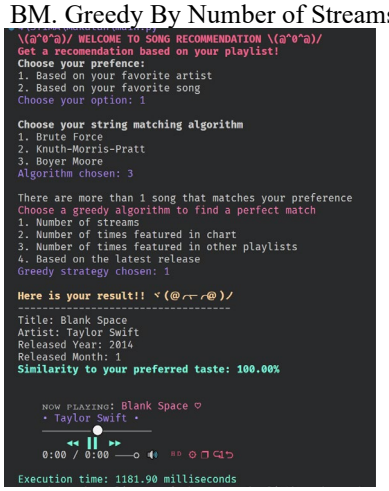


Figure 8. Test Case 3

   d. Bruteforce, Greedy By Number of Times Featured in Chart

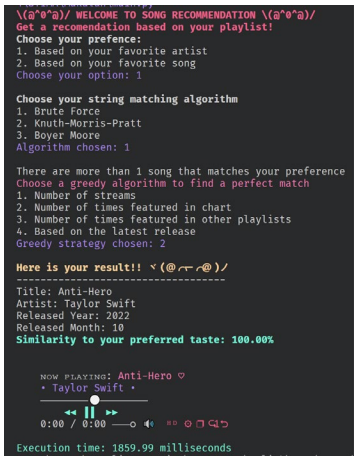


Figure 9. Test Case 4

   e. KMP, Greedy By Number of Times Featured in Playlist

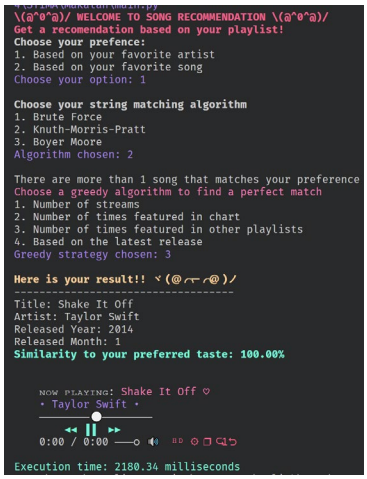


Figure `10. Test Case 5

f. BM, Based on Release Date

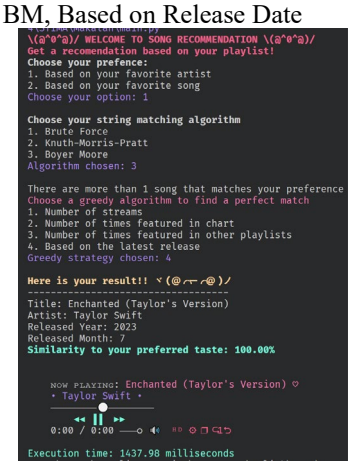


Figure `11. Test Case 6

2. Based on user's favorite song

When conducting the experiment, it was found that there were no exact match for the pattern. Therefore all of the results were presented using Levensthein Distance

a. Levenshtein, Greedy By Number of Streams

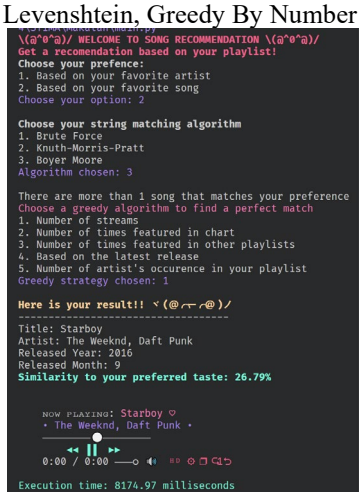


Figure `12. Test Case 7

b. Levenshtein, Greedy By Number of Times Featured in Chart

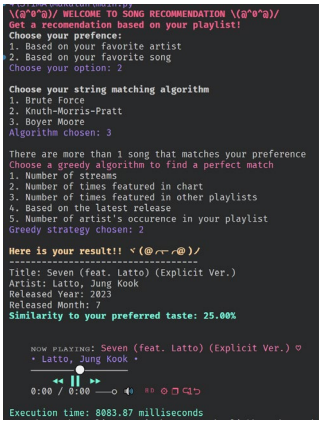


Figure `13. Test Case 8

c. Levenshtein, Greedy By Number of Times Featured in Playlist

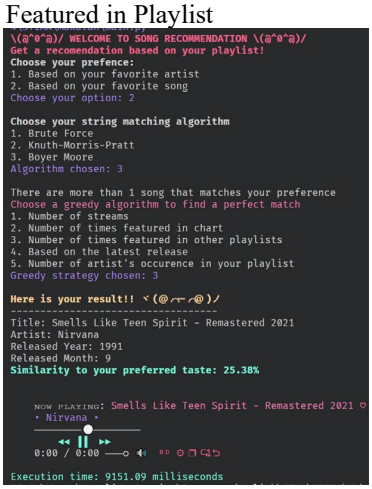


Figure `14. Test Case 9

d. Levenshtein, Based on Release Date

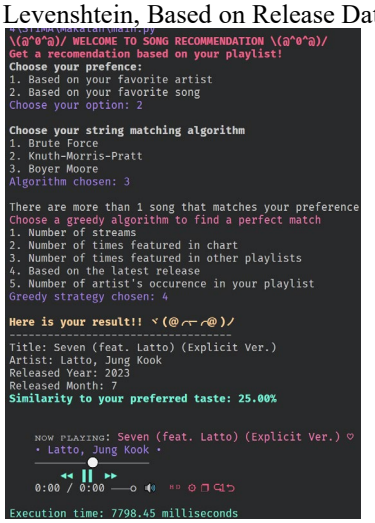


Figure `15. Test Case 10

e. Levensthein, Greedy by User's Preffered Artist

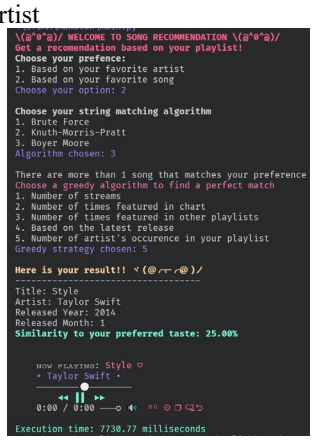


Figure `16. Test Case 11

B. Analysis

The result above will be mapped into a table for better comprehension.

To compare between the three pattern matching algorithms, recommendation data based on the user's favorite artist and greedy by number of streams will be used.

BruteForce	KMP	BM
17923.69 ms	1242.74 ms	1181.90 ms

Based on the execution time it can be inferred that the Brute Force algorithm is the simplest and most straightforward pattern matching algorithm. It checks each possible position in the text to find a match for the pattern. The KMP algorithm improves the efficiency of pattern matching by preprocessing the pattern to determine where mismatches can lead to, thus avoiding unnecessary comparisons. The Boyer-Moore algorithm is one of the most efficient pattern matching algorithms, using heuristics to skip sections of the text, thus reducing the number of comparisons needed. For this experiment involving the user's favorite artist and the "greedy by number of streams" strategy, the BM algorithm is the most efficient choice, followed by KMP, and finally the Brute Force algorithm.

Moreover, when comparing the results to those obtained using the Levenshtein Distance algorithm, it becomes evident that the Levenshtein Distance falls behind both the KMP and BM algorithms in terms of efficiency and speed. The Levenshtein Distance algorithm, which calculates the minimum number of single-character edits (insertions, deletions, or substitutions) required to change one word into another, is computationally intensive. This increased computational complexity results in longer execution times, making it less suitable for large datasets or real-time applications compared to KMP and BM.

The greedy algorithm also proves to be working as seen on the different recommendation given based on the greedy strategy. To illustrate the results will be mapped into a table

Based on user's favorite artist	
Greedy by number of streams	Blank Space – Taylor Swift
Greedy by number of times feautred in chart	Anti Hero – Taylor Swift
Greedy by number of times feautred in playlist	Shake It Off – Taylor Swift
Based on the latest release	Enchanted (Taylor's Version) – Taylor Swift

Based on user's favorite song	
Greedy by number of streams	Starboy – The Weeknd, Daft Punk
Greedy by number of times feautred in chart	Seven (feat. Latto) – Jungkook, Latoo
Greedy by number of times feautred in playlist	Smells Like Teen Spirit – Remastered 2021 - Nirvana
Based on the latest release	Seven (feat. Latto) – Jungkook, Latoo
Greedy by user's preffered artist	Style – Taylor Swift

The table demonstrates the effectiveness of the greedy algorithm in generating relevant song recommendations based on different strategies. Each strategy focuses on a specific aspect, streams, chart features, playlist inclusions, latest releases, or user preferences, allowing for tailored recommendations that align with the user's listening habits and preferences. By using the greedy algorithm, the recommendation system can dynamically adapt to various criteria, ensuring that the user receives the most suitable and engaging music suggestions.

V. Conclusion

The comparison of pattern matching algorithms reveals significant difference in performance. The Bruteforce algorithm, is the simplest , but unfortunately thw slowest to complete the task. The KMP algorithm improves efficiency by preprocessing the pattern, reducing the execution time. The BM Algorithm use heuristic to skip certain sections of the text. Levenshtein Distance algorithm, which calculates the minimum number of single-character edits, is computationally intensive and slower, making it less suitable for large datasets or real-time applications compared to KMP and BM.

The greedy algorithm proves effective in generating relevant song recommendations based on different criteria, tailored to the user's preferences. For instance, based on the user's favorite artist, different Taylor Swift songs are recommended depending on the strategy,such as "Blank Space" for the number of streams and "Enchanted (Taylor's

Version)" for the latest release. Similarly, based on the user's favorite song, recommendations include "Starboy" by The Weeknd for its high number of streams and "Seven (feat. Latto)" by Jungkook for its chart appearances and latest release status. The combination of efficient pattern matching algorithms and well-defined greedy strategies enhances the recommendation system's capability for delivering personalized music recommendations.

VIDEO LINK AT YOUTUBE

Demonstration video can be accessed at: https://youtu.be/1HKpqBuFuDY

ACKNOWLEDGMENT

Before everything, the researcher would like to thank the Lord for His Grace and kindness. The researcher would also like to express the biggest gratitude for Dr. Nur Ulfa Maulidevi, S.T, M.Sc., for acting as the researcher's Strategy for Algorithm and for sharing the knowledge and guidance which led into helping the researcher to write this essay.

Not to mention, to other Strategy for Algorithm lecturers who also have contributed to sharing their knowledge. Finally, the researcher would give thanks to her fellow peers for giving support and confidence in finishing this essay

.

REFERENCES

[1] Munir, Rinaldi. 2024. Algoritma Greedy (Bagian 1). https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Algoritma-Greedy-(2021)-Bag1.pdf Accessed 11 June 2024

[2] Munir, Rinaldi. 2024. Pencocokan String. https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Pencocokan-string-2021.pdf Accessed 11 June 2024

[3] Nam,Ethan.2019. Understanding the Levenshtein Distance Equation for Beginners. Accessed 12 June 2024 https://medium.com/@ethannam/understanding-the-levenshtein-distance-equation-for-beginners-c4285a5604f0

[4] GeeksforGeeks. 2024. KMP Algorithm for Pattern Searching. Accessed 12 June 2024. https://www.geeksforgeeks.org/kmp-algorithm-for-pattern-searching/

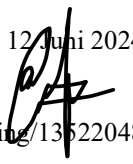
[5] GeeksforGeeks. 2024. Boyer-Moore Algorithm for Pattern Searching. Accessed 12 June 2024. https://www.geeksforgeeks.org/boyer-moore-algorithm-for-pattern-searching/

[6] Khant,Situ. 2023. https://medium.com/artificialis/music-recommendation-system-with-scikit-learn-30f4d07c60b3. Accesed 12 June 2024.

. PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan da0ri makalah orang lain, dan bukan plagiasi

Bandung, 12 Juni 2024



Angelica Kierra Ninta Gurning/13522048