A String Matching Approach to Identifying and Finding Modern Pop Music Based on Its Melody

Hafizh Adi Prasetya - 13511092
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
13511092@std.stei.itb.ac.id

Abstract **- In recent years, the global music industry has undergone a massive and rapid development. The appearance of new genres, such as dubstep and country-house, also new marketing gimmicks such as idol groups have supercharged music producers into creating lots of new, revolutionary sounds. Although this seems like a great thing to the industry, it also creates a troubling phenomenon where a song's lifespan is reduced to mere weeks. It is often the case where a particular song becomes a hit but was quickly forgotten in one or two weeks, except for parts of its melody. We see this as a motivation to develop a practical way for identifying a music only based on parts of its melody. One way to solve this problem is by using a string matching approach. By representing melody inputs as strings, we can use it as a pattern that could be matched to a text. The text to match is none other than a database of songs that exists. This paper tries to prove its feasibility by providing a simple algorithm design to identify a song using the string matching approach.**

Index Terms- **Music, Song, Identification, Pattern Matching, String.**

I. INTRODUCTION

The advent of digital music in the early 2000s has brought some dramatic changes to the music industry. Physical sales dropped significantly because the illegal distribution of music via internet. A lot of digital processing tools were developed, enabling people to create music in their rooms with nothing more than a laptop. The music trend shifts from the traditional music played by a band, into those with digital and synthetic sound created in advanced application. In recent years, the music industry is trying its hardest to adapt with the digital technology, shifting its market into the digital sector. The biggest music market in the world right now is Apple Inc.'s iTunes.

Even though the music industry is trying to cope with the loss caused by digital distribution, the number of music produced is actually increasing. This leads into a faster, harsher mainstream scene for the songs produced. Songs rises and falls down the charts very quickly giving them the exposure time of only weeks, or even days. This situation demands a quick and effective method to archive and find and identify songs. One main aspect of a song that's already used for archiving is the lyrics. It's common practice to find a song by searching it's lyrics in Google or other web search engine.

Nevertheless, it's still a good idea to find another way to finding song, one that's simpler than typing the words of the lyrics. One of such way is finding by pitch. Imagine humming to a song and then a program quickly identifies it. With the advancement of technology in handheld gadgets and better hardware, this approach would be very viable and effective for users. However, in the implementation of such app, we need a quick and effective approach to matching the pitch input of the user and the ones in the database.

When it comes to matching items, string matching is one of the oldest and most practical approaches. If two items can be represented as strings, then we can use a string matching algorithm to check for correctness. In this this case, if we can represent a user pitch input and the database of songs as strings, we can use a typical string matching algorithm like Boyer-Moore or Knuth-Morris-Pratt to find the input in the database. This paper tries to explore such idea, proposing a scheme to represent the input pitch and the database as strings, and using a string matching algorithm to find songs quickly and effectively using the pitch string.

II. FUNDAMENTAL THEORIES

A. Pattern Matching

Pattern matching is the act of checking whether or not a given sequence (pattern) is the same with another sequence given. Usually the sequence that is matched with the pattern is bigger in size so that the problem becomes finding the pattern in the bigger sequence (text). It is one of the most popular approaches in the problem of object identification or search.

A pattern matching approach could be further broken down into different sub-approaches. One sub-approach that we will use in this paper in particular is the string matching approach. A string matching approach reduces the pattern and the text into strings, and then uses a string-matching algorithm to find the pattern string inside the

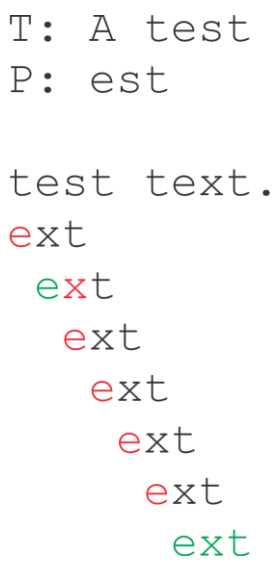
text string. In theory, as long as anything can be represented in the form of strings, the string matching algorithm can be used.

There are a lot of string matching algorithms, each with its own strength. However, this paper does not cover the comparison between string matching algorithms; it only tries to prove the feasibility of the string matching approach. The most primitive algorithm, the brute force algorithm, will be used to illustrate that it is actually possible to use string matching.

A.1. Brute Force Algorithm

It is possible to find pattern in text using a brute force approach, that is checking all the possible places for a pattern to appear until it was found or the end of the text is reached. This method, however, shares the drawback of all brute force oriented approach, which is a long response time. Nevertheless, it gives a result and a solution to the problem, which is sufficient to prove the point.

The brute force algorithm for string matching is very simple and is best illustrated using an example:



An example of string matching with brute force

In the picture above, we try to find the pattern "ext" in the text "test text". The brute force algorithm first place the pattern in the beginning of the text and try to match each corresponding character in the pattern and the text. The red letters represent a mismatch, and the green represents a match. If a mismatch happens, the algorithm moves the pattern one ahead, and repeats the matching process to the start of the pattern. In pseudo-code notation:

```
function BruteStringMatching(Pattern P, Text T):
    // The position of the matching in P and T
    i = 0
    j = 0

    WHILE pattern not found AND i < Size(T) - Size(P):
        // Try to match
        IF (T[i] == P[j]) // Match
                i++
                j++
                IF (j == Size(P)) // Pattern found
                        RETURN true
        ELSE
                i = i - j // return
                j = 0

   // While ended, pattern not found
   RETURN false
```

B. Music Structure

Music, especially modern popular music all have certain structure that we can use to increase the efficiency of our algorithm design. The musical structure described here would be the structure of a typical modern pop song, keeping in mind that it's those kinds of songs that are mostly found in today's song library. The study of a using this approach for other genres of music such as classical music would not be discussed here.

B.1. Popular Song Structure

In popular music, songs are usually divided into sections, typically *chorus*, *verse*, *bridge*, and (but not necessarily) *intro/outro*. These sections are typically repeated throughout the song, only differing in lyrics in case of a lyrical section. Here, a short explanation of each section would be provided.

Intro is a unique section that usually comes in the beginning of the song. Although more recent songs have either a *chorus* or a *verse* in the beginning, *intro* is still unique enough to be considered a section of its own. In songs with lyrics, intro are usually (although not exclusively) instrumental and have its own melody. On the other hand, an *outro* is similar to an *intro*, but it comes at the end of the song.

Chorus is essentially the highlight of the song. It is a part that is specifically designed with greater intensity, musically and emotionally. It usually sends the main idea of the song and often repeated more than one time, with similar lyrics.

Verse is the main part of the song. It usually can be identified as the part that appears often in the song, and is on a middle level of intensity. In contrary with the chorus, a verse usually sets up the narrative with different lyrics in different verses.

Bridge as the name implies, is the part between sections of the song. In typical modern pop songs, a bridge usually appears once, after the second chorus leading to the final chorus. Bridges often have unique melodies, with intensity building up so that the final chorus would have the peak intensity in the song.

III. Algorithm Design

When using the pattern matching approach to solve a certain problem, three fundamental things must first be defined. The pattern that we want to search, the

text in which we search the pattern and the algorithm used in matching the pattern to the text. In this particular case, the pattern would be the pitch data inputted by the user, the text would be all the song data in the database, and the algorithm would be a basic pattern matching algorithm modified to tackle some of the domain-specific issue that we will talk about later on.

A. The Pattern

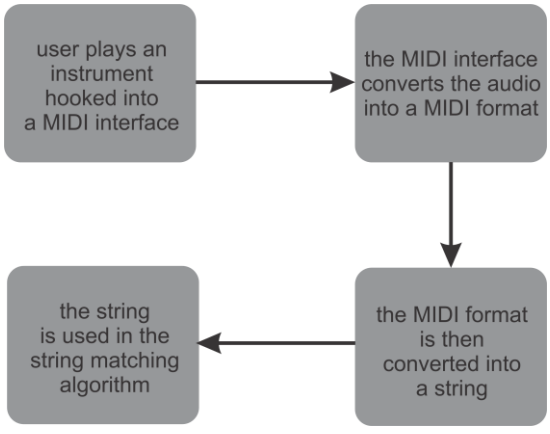
There are lots of ways a user can input a sequence of pitch to be matched. One could input it manually by writing it as a string with a predefined protocol, plug a guitar and play it to the tune of the song, using a specific tool to create a certain music format, or just simply sing the song, record it, and put it in a voice processing tool.

In order to simplify the problem, this paper generalizes those above into two types of user input: an accurate input, for example using an established format such as MIDI, and an inaccurate input, for example singing to a recorder and converting the sound waves it to a standard notation using a pitch-detection algorithm. We will briefly take a look at those two types of input and how to convert them into strings.

A.1. MIDI

MIDI stands for Musical Instrument Digital Interface, founded about at 1982, pioneered by leading electronic musical instrument manufacturer like Roland, Yamaha, Korg, and Kawai. It's a technical standard that describes a protocol, digital interface, and connector, designed to allow the communication and transfer of data between electronic music instruments and the computer. MIDI also allows recording of music data into a hardware/software called sequencer and editing it later on.

The data carried by MIDI are common music data, such as note events, timing events, pitch bends, tempo, etc. These data, especially the pitch and note events are exactly the data we need as a pattern. A typical scenario would be hooking an electronic instrument into the computer, playing the music on it, passing it to the midi sequencer, and then converting it as a string to be used in the string matching algorithm.



A typical MIDI input translation scheme

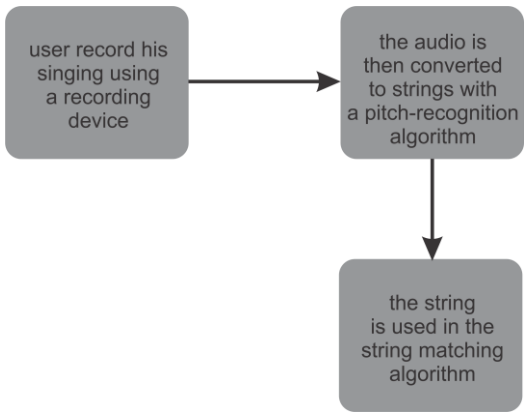
The reason why MIDI is very compatible with our string matching approach is because the separation of data in MIDI format makes it very easy for programmers to convert it into regular text. In fact, a lot of MIDI to ASCII programs are readily available on the net, some of them are midi2text code.google.com/p/midi2text/) and MIDI-OX (www.midiox.com/app.htm). Also, as long as the user gets the note right, MIDI format would deliver a precise text, without any tempo or pitch deviation.

```
  BA 2 CR 0 TR 1 CH 1 NT  C#- 4 von=95   voff=80
  BA 2 CR 1/24 TR 1 CH 1 NT  F 3+23/24   voff=80
  BA 2 CR 1/12 TR 1 CH 1 NT G# 3+11/12   voff=80
  BA 2 CR 1/8 TR 1 CH 1 NT G#    3+7/8   voff=80
  BA 2 CR 1/6 TR 1 CH  1 NT Eb'  3+5/6   voff=80
```

An example of midi conversion to string, using mid2asc from http://www.archduke.org/midi/instrux.html

A.2. User Audio Input

The second input, and the more practical and marketable one is regular speech recording. The typical scenario in using this input would be the user sing into a recording device (which is readily available in most handheld device now, such as smartphone and tablets), the recording get converted into a string, and the string is then used as a pattern in searching for the song.



A typical recording input translation scheme

The issue that should be tackled in receiving a speech input is the conversion from audio waves to a readable string. This is possible using a pitch-detection algorithm (PDA) that has been developed over the course of the decade, and has been implemented in many end user applications, notably in singing games that score according to pitch correctness such as Ultra Star. The algorithm would not be discussed in this paper, and all points explained afterward will assume a precise conversion from a user's speech to a string.

B. The Text

The text that would be searched is of course a

collection/database of songs. To be able to use this collection of songs as the text in our string matching algorithm, it should be represented as a string. One could view the whole database as one big chunk of text, concatenating the whole songs, or view the database as a collection of patterns to iterate through the song finding process.

This paper would not fully tackle the problem of converting a huge database of song into string forms, nor would it discuss the system used to manage and organize the collection. Instead, this paper would assume a separation in every song, and propose a simple system to represent the songs and reduce it's size as much as possible in regard to the characteristic of a song.

B.1. Reducing Repetition

It has been established that most songs includes repetition (some even would count on it to better cement the melody of the song to the listener's head). The problem arising with this repetition is it causes a repeat in the text, causing the pattern to search in the same substring of text more than once. This could lead to overhead in response time, one that should be easily avoided.

It's a given that when redundancy occurs, we should eliminate the redundancy, and that's what we will do in this case. The problem would be, how small of a redundancy should we eliminate. It's not possible to eliminate a redundant note, because it might be a part of the melody and reducing it to one note would change the melody. It might also not right to remove a redundancy in one bar of the song, because the user might remember the song as having a two-bar hook. This paper would propose repetition elimination by the song structure.

When a song repeats a verse, or a chorus, and there's no notable difference between the first and second verse/chorus, only one should be included in the text. In typical modern pop song, the structure would be:

intro-verse-chorus-verse-chorus-bridge-chorus

This approach would reduce the text into:

intro-verse-chorus-bridge

Assuming all the parts have the same length, this effectively eliminates the size of the text to only 4/7 of the original length, or about 57% percent. This means in the case of a full mismatch (the pattern does not exist in the song), it reduces 43% of the search time. Of course this does not apply to all songs, but considering the trend in modern music industry, this should give a significant boost to the algorithm.

B.2. Reducing Instruments

B.2.1. Main Melodic Line

Not all parts/instruments of a song are meant to be catchy and remembered by the listener. Some more subtle parts, like the bass and effect (especially in modern digital music) are meant as an accompaniment to further maximize the impacts of the main melody, like the vocal, piano, or synths. The lines delivered by these melodically strong instruments are usually the one stuck to the listeners head, thus used as a search pattern to find the song it was from. Although it doesn't mean that all the subtler instruments would never be used as a search pattern, it implies that including the whole melodic line of a song in the text would not be necessary in most of the cases.

In response to this, this paper proposes that the songs stored in the database should be broken down to two parts, according to the importance and impact it brings to the song. The part with the most impact (and thus, more likely to be remembered) should be separated and given a higher priority searched as a text. We shall call this part from here onward as the Main Melodic Line.

The main melodic line would mostly comprises of the melodic instruments and vocal line, but not exclusively so. A lot of music also incorporates subtler instruments like the bass to deliver the hook, for example, the intro to "Starlight" by the English rock band Muse. Because of this, constructing the main melody line is not as simple as dividing musical instruments; it's more complex than that. It requires a careful listening of the music, picking out instruments that stick out the most in every part of the song. This is especially hard on songs that have multiple layering of instruments, but seeing that the text would be provided by the service, not the user, this would not be a problem for the user. The heuristics to construct the main melodic line is not covered on this paper.

B.2.2. Secondary Melodies

The secondary melodies would be all the melodic line not included in the main melodic line. As we cannot guarantee that the user would not use the subtler line as a search pattern, we should also store these secondary melodies in the database. The difference with the main melodic lines would be the search priority. The main melodic line would be searched first as a text, and then if there's no text matching the pattern, the secondary melodies would be searched.

There are some schemes possible in regard to the searching priority. For example, you could search the secondary melodies for a song right after the main melodic line of it. Another scheme would be searching all the main melodic line first, then the secondary melodies. All this schemes would have advantage and disadvantages, but it would not be discussed in this paper.

C. The Algorithm

There are two major algorithm of this string matching approach. The first is the searching algorithm in the database that dictates the order of which the text files are

matched to the pattern. The second is the string matching algorithm itself, to find the pattern within a text file.

C.1. Database Search Algorithm

Since we were never concerned with the performance of the algorithm, we will use a simple database search scheme based on the structures we already established. The database would comprise of individual files, each file representing a song. In each files, two strings exists: the main melodic line, and the secondary algorithm. For this example, we would check the main melodic line first, and then the secondary algorithm. Hence, the pseudo-algorithm for this scheme would be:

```
function SearchSong(Database D, Pattern P):
     // Search main melodic lines
     FOR ALL mainmelodiclines M IN D:
        IF match(M, P):
             RETURN M
     // Search secondary melodies
     FOR ALL secondary melody SM IN D:
        IF match(SM, P):
             RETURN SM
     // Not found
     RETURN NULL
```

C.2. Pattern Matching Algorithm

C.2.1. Pitch Notation

Before we use the pattern matching algorithm, we should define the notation for the pattern. Assuming a very basic example song consisting of only one note long notes and with a one octave range (no same notes with different octaves), a numerical notation should suffice. This numerical notation would have '1' as do, '2' as re, '3' as mi, and so on. The reason of choosing a numerical notation is because it's intuitive, where a higher note means a higher numerical value, and it will help in solving the pitch error that we will face later.

As an example, here's a string for a simplified "do-re-mi-fa-sol-la-ti-la-sol-fa-mi-re-do":

```
1234567654321
```

C.2.2. Basic Algorithm

In the simplest case, assuming that the user's input is pitch and tempo perfect, for example a string produced from MIDI. The input would also have a single octave song range. In this case, the algorithm for matching a pitch from user input into a song is reduced to basic string matching, with the exactly same algorithm as we already discussed in chapter II.A.

```
function BruteStringMatching(Pattern P, Text T):
    // The position of the matching in P and T
    i = 0
    j = 0

    WHILE pattern not found AND i < Size(T) -
Size(P):
        // Try to match
        IF (match(T[i],P[j])) // Match
               i++
               j++
               IF (j == Size(P)) // Pattern
found
                     RETURN true
        ELSE
               i = i - j // return
               j = 0

   // While ended, pattern not found
   RETURN false
```

With the function match:

```
function match(char a, char b):
   IF(!a == b)
        RETURN false
   ELSE
        RETURN true
```

The reason of giving the matching process its own function would be apparent later when we tackle the problem related to the domain one by one. At this point, it's proven that the string matching approach works for simple cases. Here's one example to it:

```
T: 1234567654321
P: 543

1234567654321
543
 543
  543
   543
    543
     543
      543
       543
        543
         543
```

A basic, example search

C.3. Correcting the String Matching Algorithm

It's already proven that in the case of an ideal user input, with MIDI for example, a string matching approach works well. But the same can't be said for inputs originating from a user's vocal. The reason for this is that user's vocal is prone to error, whether it's pitch error or tempo error. The same also can't be said when the song began to have a higher range, with the same note having a different octave value. Here we will try to explore and propose a solution for some of those problems.

C.3.1. Pitch Correction

A problem arises when the input taken is from a user's recorded voice, such as humming, whistling, or singing.

The problem is that we can't count on users to produce a pitch-perfect vocal. Most people would sing a half note flatter or sharper, especially in extreme region of the octave (very high or very low). To facilitate this error, we need to modify our algorithm so that it can tolerate some of the note error produced. It should be able to take a pattern that's only a note different, yet still producing a match.

One solution would be giving a value to each character, and then determine a *tolerance value*. A tolerance value T would be the maximum difference between notes matched that would still count as a match. This is the reason why a numerical notation is chosen, because they intrinsically already have a value assigned.

To implement this, we would just modify the function match to facilitate error tolerance. The function would then become:

```
      function match(char a, char b, int T):
  IF(abs(int(a) - int(b)) > PT)
      RETURN false
  ELSE
      RETURN true
```

This solution would add only a little modification to the algorithm. Here's an example to illustrate the pitch tolerance given:

```
T: 1234567654321
P: 445


1234567654321
445
 445
  445
```

A basic search with pitch tolerance

The problem of how big is the optimal tolerance to maintain accuracy of the search is outside the scope of this paper.

C.3.2. Tempo Correction

Another problem that occurs in using a manual vocal input from user is the tempo. A single note can have multiple lengths; it can be one note long or double that. Moreover, we also can't count on the user to produce a vocal with precise tempo. So, to facilitate this, we need to modify our algorithm so that it also can take account of the note length, and tolerate the length error produced by the user's vocal.

There are two things we must consider: how to express this note length in our numerical notation, and how to allow tempo tolerance.

The proposed solution is to create a string of note length; each character corresponds to the character in the string of note pitch. These two strings would then be stored in an array, and become the pattern. The same would be done with the text, now our texts just become an array of strings with size two. Regarding the tolerance of tempo, we can just apply the same solution as the tolerance of pitch. In implementing this solution, the algorithm becomes so:

```
function BruteStringMatching(Pattern P, Text T):
    // The position of the matching in P and T
    i = 0
    j = 0

    WHILE pattern not found AND i < Size(T) -
Size(P):
        // Try to match
        IF (match(T[1][i],T[2][i],P[1][j],
P[2][j])) // Match
                i++
                j++
                IF (j == Size(P)) // Pattern
found
                        RETURN true
        ELSE
                i = i - j // return
                j = 0

   // While ended, pattern not found
   RETURN false
```

```
        function match(char a, char la, char b,
char lb, int PT):
   IF(abs(int(a) - int(b)) > PT) && (abs(int(la)
- int(lb)) > PT)
        RETURN false
   ELSE
        RETURN true
```

Once again, this solution would add only a little modification to the algorithm. Here's an example to illustrate the pitch and tempo tolerance given:

```
T: 1234567654321
   1122222122211
P: 445
   323


1234567654321
1112222122211
445
323
 445
 323
  445
  323
   445
   323
```

IV. Conclusion

A string matching approach to finding songs is certainly possible. Even though the scheme proposed above does not have the necessary quality to be a usable scheme, but it proves that such scheme is actually possible. The optimization of the scheme could be done using a better, more advanced string matching algorithm, such as Boyer-Moore, or Knuth-Morris-Pratt.

V. Further Studies

Further studies on this subject should cover the following:

- A research into a better database search scheme, possibly the benefits of inserting extra information such as genres or singers to make the database search quicker.
- The heuristics involved in making a main melodic line, or an alternative proposal for reducing the amount of strings that should be checked.
- A solution for the problems not covered in this paper, such as complex songs with range more than one octave, breaking down the note length to half a note or shorter, and many more.
- A comparison of multiple string matching algorithms in the problem domain, complete with the calculation of their complexity.

References

[1] Appen, Ralf von / Frei-Hauenschild, Markus *AABA, Refrain, Chorus, Bridge, Prechorus — Songformen und ihre historische Entwicklung*. In: *Black Box Pop - Analysen populärer Musik*. Ed. by Dietrich Helms and Thomas Phleps. Bielefeld: Transcript 2012, pp. 57-124. ISBN 978-3-8376-1878-5.

[2] D. Gerhard. Pitch Extraction and Fundamental Frequency: History and Current Techniques, technical report, Dept. of Computer Science, University of Regina, 2003.

[3] Huber, David Miles. "The MIDI Manual". Carmel, Indiana: SAMS, 1991.

[4] Munir,Rinaldi. 2009. *Diktat Kuliah IF3051 Strategi Algoritma*. Program Studi Teknik Informatika STEI ITB

Pernyataan

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

Bandung, 19 Desember 2013

ttd

Hafizh Adi Prasetya - 13511092