Pattern Matching in Simple Audio Recognition

Felicia Christie - 13512039
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
13512039@std.stei.itb.ac.id

Abstract—**Audio recognition is now popularly used. For example, almost every smartphone distributed nowadays has a build-in audio recognition function to translate speech to text, or to receive orders by sound or by speech. This paper discusses the general method to compare an audio pattern to an audio 'text' by using the means of string matching algorithms**

Index Terms—**audio, string matching, pattern matching, sound**

I. Introduction

In this modern era, the use of audio has expanded from a method of communication to a method of education, entertainment, relaxation and even employment. In order to provide users a better service, many smartphones now has a build-in speech recognition program for users who prefer aurally stating what they request from the phone instead of the regular method of pressing buttons or touching screens, in cases where their hands are not available for that particular use.

Another popular usage of audio recognition is security. Audio recognition provides security by recognizing the speech patterns of a certain user, then when another user requests access to a particular protected data, the security system will prompt an audio input, for example, a particular keyword. Other than matching the phrase, a more advanced security system will also be able to match the particular speaking method and the voice with the database of allowed personnels.

Audio recognition is also used to parse input in human-computer interactions using sound as the method of communication. The system searches for certain keywords that are programmed into the system, then if the input matches a certain pattern, the system calls an appropriate response function.

Pattern matching can be used in audio recognition. Although in real-life cases, the patterns are not as simple as string that we can easily simplify audio recognition as a problem of string matching. The scope of this paper is a simple audio, especially the MIDI file format.

II. Fundamental Theories[1]

String matching, or pattern matching, is a searching problem where we try to locate the first occurence of a certain pattern (P) in a document or a part of text (T). An example of a string matching problem:

Pattern: not
Text: nobody **not**iced him

There are several common algorithms that are used to solve this problem, which are *brute force* algorithm (in string-matching context, also called the Naive String-matching[4]) , *Knuth-Morris-Pratt* Algorithm and *Boyer-Moore* Algorithm.

There are several terminology that will be used in the explanation below.

- Prefix: any substring that starts from the beginning of the original string and ended at any but the last character of the original string. Also includes the empty string.
- Suffix: any substring that ends with the last character of the original string, and starts in any point of the original string except the beginning character. Also includes the empty string.

A. Naive String Matching

There are three simple steps in this algorithm;

1. Pattern P is aligned to the begining of text T
2. From left to right, compare each character of P with each character of T that aligned with it.
3. If all the characters are the same, then end of searching because the pattern is found.
4. If there was an inequality of the characters and the next character after the last-pattern-aligned character is not an empty string, move the pattern by one step. Start the searching again from step 2.

In the best case of this algorithm, where the pattern is found right at the beginning of the text, the number of character comparison is the length of the pattern (n), which makes the complexity *O(n)*. On the worst case, the algorithm checks all character in the pattern, only to found an incorrect character at the last character of the pattern for every shift, making the complexity *O(m*n)* and (m – n + 1) comparisons.

B. Knuth-Morris-Pratt (KMP)

With brute force, we are forced to shift just one character, meanwhile with the Knuth-Morris-Pratt algorithm, we can calculate a more optimized number of shifting, making the Knuth-Morris-Pratt algorithm more efficient than brute force in most cases. KMP itself is quite similar to Naive String Matching, similarities include the direction of the matching. The goal of KMP is to reduce unnecessary rematches as much as possible without missing any potential matches.

This algorithm makes use of the border function, which counts the number the characters of the suffix that is equal to the number of characters of the prefix for all substring of pattern P. The border function is generally represented by *b(j)*, with *j* as the index of the last character in the substring to be calculated.

The first step of the Knuth-Morris-Pratt algorithm is to count the border function for each substring of the pattern. For example, P : aabaabb, so the resulting border function for each substring will be:

j	1	2	3	4	5	6	7
P	a	a	b	a	a	b	b
b(j)	0	1	0	1	2	3	0

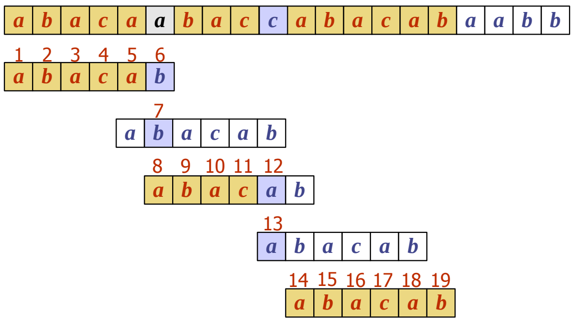
If when matched with a certain text, and the n^th^ character of the pattern did not match the text character, then the number of characters to shift is

$$X = l - b(n)$$

l = index of last character that was correctly matched, counting the index of the first character as 1

b(n) = border function of the wrongly matched character, n as the index on the pattern.

After shifting the pattern, the checking starts on the X^{th} character of the pattern.



Picture 1. An example of KMP Algorithm
Source: Dr. Andrew Davison, Pattern Matching Presentations

On the complexity of this algorithm, firstly, KMP requires the user to first calculate the border function for each character on the pattern, which has the complexity of *O(m)*. Then, in the process of string matching, it would need to check the mismatched character at least twice, having the complexity of *O(n)*. The overall complexity of this algorithm is *O(m+n)*.

C. Boyer-Moore (BM)

Boyer-Moore algorithm aligns the pattern to the beginning of the text, but unlike by brute force or by KMP, the character-checking is done from the rightmost character to the leftmost character. Firstly, we have to list all the alphabets from both the text and the pattern, and the index of last occurences in the pattern of all the alphabets, with the rightmost character as the last character.

When matching, there are three possible cases;

1. When matching, the character of the pattern did not match the aligned character, and the last occurence of the wrongly matched character of the text has not yet passed.
   Solution: Shift the pattern so that the character in text are aligned with the last occurence of the character, that is
   (pattern length – index of last occurence)

An example of this case is as following;
Before shifting:

Text:	a	b	c	d	e	f	g	h
Pattern:	a	a	a	a	h	a	a	a

After shifting:
\

Text	a	b	c	d	e	f	g	h	....
Pattern	-	-	-	a	a	a	a	h	a

2. The second case, that is the last occurence of the mismatched character on the text has passed the current checked character of the pattern.
   Solution: Shift the pattern once

An example of this case is as following;
Before shifting:

Text	a	b	c	d	e	f	g	f
Pattern	-	-	-	a	a	a	g	f

After shifting:

Text	a	b	c	d	e	f	g	f	....
Pattern	-	-	-	-	a	a	a	g	f

3. The third case, for events not covered by the previous two cases, including an alphabet that was matched has never appeared on the pattern.
   Solution: shift the pattern minimally so that the non-existant character in the pattern was skipped.

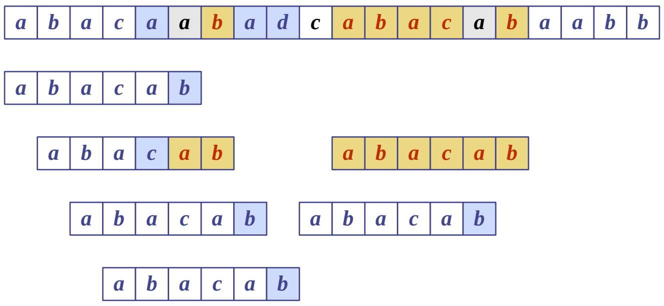
An example of this case is as following:
Before shifting

Text:	a	b	c	d
Pattern:	a	a	a	a

After shifting:

Text	a	b	c	d	..	..	..	..
Pattern	-	-	-	-	a	a	a	a

An example of string matching done by the Boyer-Moore algoritihm:



Picture 2. An example of KMP Algorithm
Source: Dr. Andrew Davison, Pattern Matching Presentations

The worst case running time of Boyer-Moore is $O(n*m+A)$, with A as the number of alphabets. This algorithm is fast when the value of A is large, and slower when the alphabet is small. When searching on an English text, Boyer-Moore is significantly faster

Other than the mentioned three algorithms, there are several commonly used algorithms for string matching, for example the Rabin-Karp algorithm, Finite-state automaton based search, and the Bitap algorithm.

III. Converting Audio to Text

Audio files are generally viewed in waveform or in audio spectrums. These forms can be directly transformed to string form by using *Hough* transform, which is generally used to recognize visual patterns.[2] In this particular usage, Hough algorithm is then used to transform the audio spectrum model to a string that can be compared.

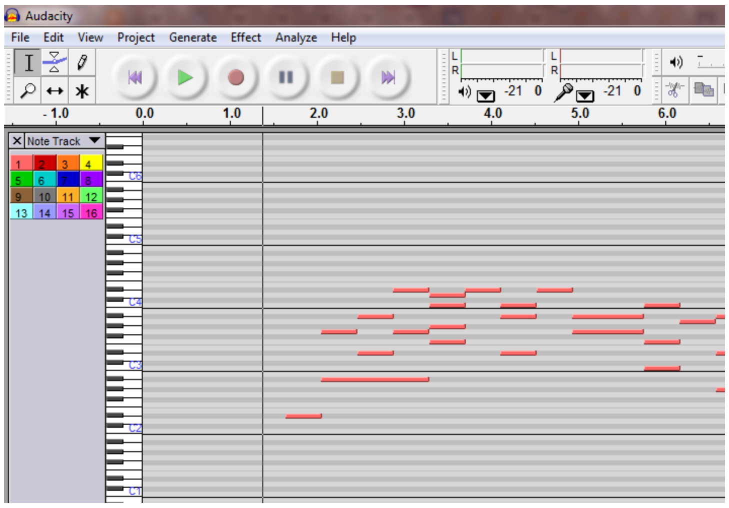
Hough transform is a method to calculate the value of the frequency in an instance of the audio. In short, by using the Hough transform, we calculate the root mean square (RMS) of each column from the spectrogram of both the audio sample and the audio we want to match. Of course, the intervals of the calculations must be the same for the sample and the pattern to match. Each product of calculation is then arranged to become a string.

An alternative to using Hough transform is by Fast Fourier Transform (FFT). FFT is commonly used to process digital audio files. The Fourier analysis computes time (or space) to frequency, and the opposite. In code, we can use the Fast Fourier Transform to convert an array of audio bytes toan array of Fourier Transform Coefficients of the audio signal. These coefficients will be in complex numbers so we would need to calculate the magnitude of each FFT coefficient.

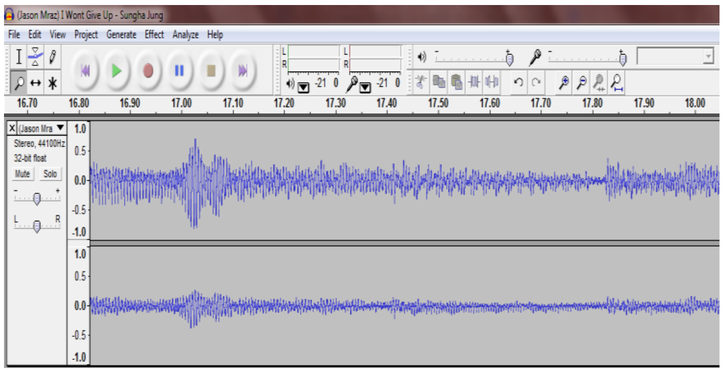
The FFT coefficient can be calculated to result in the frequency content of the signal at N equally spaced frequencies. [5] If a sample 256 points were taken to be transformed, with sampling frequency is at 44100 Hz, then the frequency spacing will be 44100/256 = approximately 172 Hz.

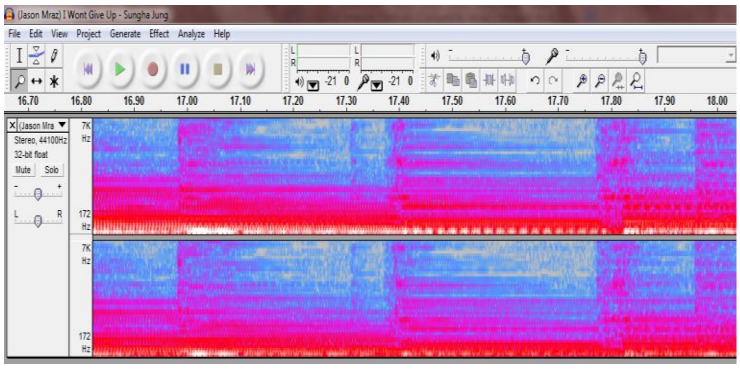
Another far simpler method is to manually list out the frequencies of each intervals of both the audio sample and the audio pattern. Similar to the previous method, the intervals used when listing the frequencies of both audio must be the same. This method would only work in simple digital audio, because in real life, no audio is clear that we can determine the frequency of a certain point of an audio file. The approach that will be taken in this paper is that the audio is processed in this method, and results in a string which is its frequencies at certain parts of the audio..

The smaller the interval between two points from audio that was calculated, the better. With smaller intervals, we can detect similarity with much higher accuracy. Generally, we use thirty frequency takes in one second. Thus the interval is 0.033 miliseconds.



Picture 3. A simple audio file (.mid) viewed by Audacity


(Jason Mraz) I Wont Give Up - Sungha Jung
File Edit View Project Generate Effect Analyze Help
Stereo, 44100Hz
32-bit float
Mute Solo
Picture 4. A usual audio file (.mp3) in waveform by Audacity[6]


(Jason Mraz) I Wont Give Up - Sungha Jung
File Edit View Project Generate Effect Analyze Help
Stereo, 44100Hz
32-bit float
Mute Solo
Picture 5. A usual audio file (.mp3) in spectrum by Audacity[6]

IV. Some Common Misses

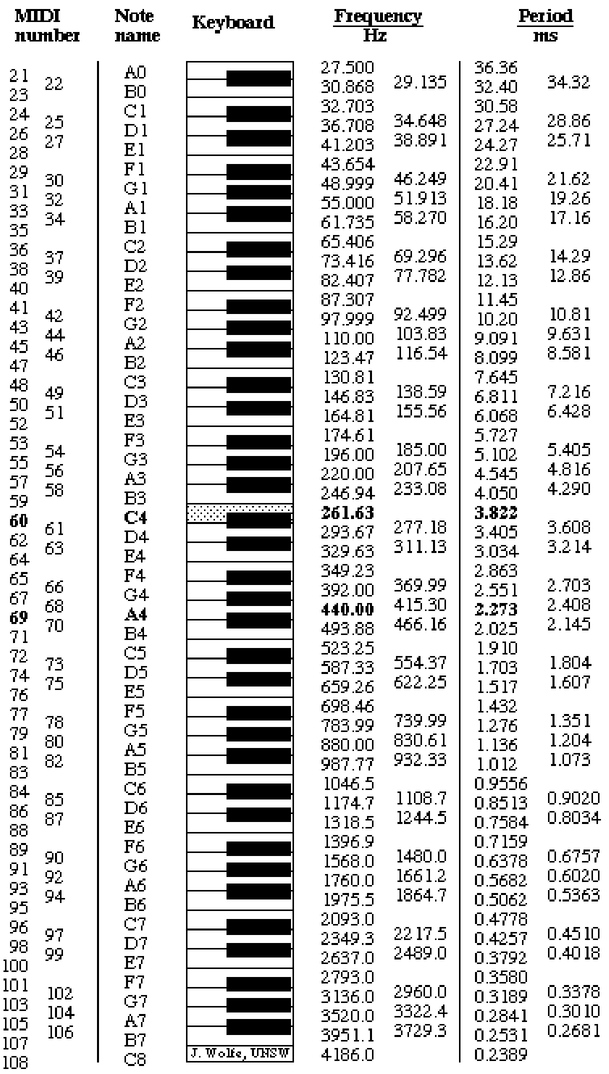
When collecting frequencies, keep in mind that the frequency data should be taken in regular intervals, and that the interval of the sample audio and the pattern audio should be the same. But even with this, there might be some inconsistencies between the frequencies that we had listed, especially when we use a larger interval.

For example, in a case where an audio pattern 20 seconds long and the audio sample is 50 seconds long, we gathered the data from both pattern and sample with the interval of 1 second. If both audio files are prone to sharp differences of frequencies in short intervals, then exact matching would have a higher possiblity to be false. In this case, if the pattern actually starts in the 10,50th second of the sample while we did not acquire the data of that particular second then by exact match (eg. the 10th and 11th second), the comparison would prove wrong, because the exact match was skipped.

V. STRING MATCHING IN AUDIO

After converting both sample and pattern to text, the searching can be done by various means not limited to the algorithms mentioned before. In the case of analyzing MIDI files, the analysis might result in an array of arrays, as can be seen on Picture 3, which has four parallel notes at 3:50. The following will be a case where the analyzed audio file is a MIDI file.

In a MIDI file, instead of listing the frequencies we could identify each note with its MIDI number.


Keyboard
J. Wolfe, UNSW

MIDI number	Note name	Frequency Hz	Period ms
21	A0	27.500	36.36
22		29.135	34.32
23	B0	30.868	32.40
24	C1	32.703	30.58
25		34.648	28.86
26	D1	36.708	27.24
27		38.891	25.71
28	E1	41.203	24.27
29	F1	43.654	22.91
30		46.249	21.62
31	G1	48.999	20.41
32		51.913	19.26
33	A1	55.000	18.18
34		58.270	17.16
35	B1	61.735	16.20
36	C2	65.406	15.29
37		69.296	14.29
38	D2	73.416	13.62
39		77.782	12.86
40	E2	82.407	12.13
41	F2	87.307	11.45
42		92.499	10.81
43	G2	97.999	10.20
44		103.83	9.631
45	A2	110.00	9.091
46		116.54	8.581
47	B2	123.47	8.099
48	C3	130.81	7.645
49		138.59	7.216
50	D3	146.83	6.811
51		155.56	6.428
52	E3	164.81	6.068
53	F3	174.61	5.727
54		185.00	5.405
55	G3	196.00	5.102
56		207.65	4.816
57	A3	220.00	4.545
58		233.08	4.290
59	B3	246.94	4.050
60	**C4**	**261.63**	**3.822**
61		277.18	3.608
62	D4	293.67	3.405
63		311.13	3.214
64	E4	329.63	3.034
65	F4	349.23	2.863
66		369.99	2.703
67	G4	392.00	2.551
68		415.30	2.408
69	**A4**	**440.00**	**2.273**
70		466.16	2.145
71	B4	493.88	2.025
72	C5	523.25	1.910
73		554.37	1.804
74	D5	587.33	1.703
75		622.25	1.607
76	E5	659.26	1.517
77	F5	698.46	1.432
78		739.99	1.351
79	G5	783.99	1.276
80		830.61	1.204
81	A5	880.00	1.136
82		932.33	1.073
83	B5	987.77	1.012
84	C6	1046.5	0.9556
85		1108.7	0.9020
86	D6	1174.7	0.8513
87		1244.5	0.8034
88	E6	1318.5	0.7584
89	F6	1396.9	0.7159
90		1480.0	0.6757
91	G6	1568.0	0.6378
92		1661.2	0.6020
93	A6	1760.0	0.5682
94		1864.7	0.5363
95	B6	1975.5	0.5062
96	C7	2093.0	0.4778
97		2217.5	0.4510
98	D7	2349.3	0.4257
99		2489.0	0.4018
100	E7	2637.0	0.3792
101	F7	2793.0	0.3580
102		2960.0	0.3378
103	G7	3136.0	0.3189
104		3322.4	0.3010
105	A7	3520.0	0.2841
106		3729.3	0.2681
107	B7	3951.1	0.2531
108	C8	4186.0	0.2389

Picture 6. Frequency-Note-MIDI Number Relation
Source: http://www.phys.unsw.edu.au/jw/notes.html, accessed at 17th May 17 2014, 14.01PM

Other than by comparing each notes manually, there are several Java or C# libraries that can be used to analyze and export MIDI files.

Here is an example of the result of analyzing. By using the open-source program midicsv[8] we can convert a file with the extension .mid into a readable CSV (Comma Separated Value) file.

```
Sample CSV:
0, 0, Header, 0, 1, 480
1, 0, Start_track
1, 0, Title_t, ""
1, 0, Time_signature, 4, 2, 24, 8
1, 0, Tempo, 500000
1, 0, Tempo, 750000
1, 0, Control_c, 0, 0, 0
1, 0, Control_c, 0, 32, 0
1, 0, Program_c, 0, 94
1, 1920, Note_on_c, 0, 57, 100
1, 2400, Note_on_c, 0, 57, 0
1, 2400, Note_on_c, 0, 60, 100
1, 2880, Note_on_c, 0, 60, 0
1, 2880, Note_on_c, 0, 59, 100
1, 3360, Note_on_c, 0, 59, 0
1, 3360, Note_on_c, 0, 62, 100
1, 5184, Note_on_c, 0, 62, 0
1, 5760, Note_on_c, 0, 62, 100
1, 6192, Note_on_c, 0, 62, 0
1, 6240, Note_on_c, 0, 60, 100
1, 6672, Note_on_c, 0, 60, 0
1, 6720, Note_on_c, 0, 62, 100
1, 6936, Note_on_c, 0, 62, 0
1, 6960, Note_on_c, 0, 64, 100
1, 7176, Note_on_c, 0, 64, 0
1, 7200, Note_on_c, 0, 57, 100
1, 9120, Note_on_c, 0, 57, 0
1, 9600, Note_on_c, 0, 64, 100
1, 10032, Note_on_c, 0, 64, 0
1, 10080, Tempo, 666666
1, 10080, Note_on_c, 0, 66, 100
1, 10445, Note_on_c, 0, 66, 0
1, 10560, Tempo, 750000
1, 10560, Note_on_c, 0, 62, 100
1, 10992, Note_on_c, 0, 62, 0
1, 11040, Note_on_c, 0, 64, 100
1, 12960, Note_on_c, 0, 64, 0
1, 13440, Note_on_c, 0, 62, 100
1, 13872, Note_on_c, 0, 62, 0
1, 13920, Note_on_c, 0, 60, 100
1, 14352, Note_on_c, 0, 60, 0
1, 14400, Note_on_c, 0, 62, 100
1, 14616, Note_on_c, 0, 62, 0
1, 14640, Note_on_c, 0, 64, 100
1, 14856, Note_on_c, 0, 64, 0
1, 14880, Note_on_c, 0, 57, 100
1, 16800, Note_on_c, 0, 57, 0
1, 17280, Note_on_c, 0, 57, 100
1, 17712, Note_on_c, 0, 57, 0
1, 17760, Note_on_c, 0, 57, 100
1, 18192, Note_on_c, 0, 57, 0
1, 18240, Note_on_c, 0, 59, 100
1, 18672, Note_on_c, 0, 59, 0
```

```
1, 18720, Note_on_c, 0, 59, 100
1, 19152, Note_on_c, 0, 59, 0
1, 19200, Note_on_c, 0, 60, 100
1, 20064, Note_on_c, 0, 60, 0
1, 20160, Note_on_c, 0, 60, 100
1, 21024, Note_on_c, 0, 60, 0
1, 21600, Note_on_c, 0, 59, 100
1, 22032, Note_on_c, 0, 59, 0
1, 22080, Note_on_c, 0, 55, 100
1, 22512, Note_on_c, 0, 55, 0
1, 22560, Note_on_c, 0, 55, 100
1, 22992, Note_on_c, 0, 55, 0
1, 23040, Note_on_c, 0, 57, 100
1, 23472, Note_on_c, 0, 57, 0
1, 23520, Note_on_c, 0, 57, 100
1, 25824, Note_on_c, 0, 57, 0
1, 25824, End_track
0, 0, End_of_file
```

The format of the CSV file, ignoring the headers and footers are as following;

```
<Track No.>, <Time of note, in MIDI-
clock>, <Type>, <Channel>, <Note>,
<Velocity>
```

In this sample, we do not need to sort the data because there are only one track of audio. One track can only have one tone at a certain time, so in order to have more than one node at a time, MIDI files generally have more than one track. In cases where the MIDI file has more than one track, the notes will be arranged according to the track, so re-sorting by time should be done to the text.

The pattern P has the MIDI number sequence as the following; `59, 59, 62, 62, 62, 62, 60, 60, 62, 62.` After parsing and acquiring the sequence of MIDI Numbers in a CSV file, we can directly apply any string matching algorithm. The process of string matching by KMP is shown as the following:

P	59	59	62	62	62	62	60	60	62	62
b(j)	0	1	0	0	0	0	0	0	0	0

```
57 57 60 60 59 59 62 62 62 62 60 60 62..
x
59 59 62 62 62 62 60 60 62 62

57 57 60 60 59 59 62 62 62 62 60 60 62..
   x
   59 59 62 62 62 62 60 60 62 62

.. 57 60 60 59 59 62 62 62 62 60 60 62..
      x
      59 59 62 62 62 62 60 60 62 62

.. 60 60 59 59 62 62 62 62 60 60 62 62..
      x
      59 59 62 62 62 62 60 60 62 62

.. 60 59 59 62 62 62 62 60 60 62 62 64..
      √  √  √  √  √  √  √  √  √  √
      59 59 62 62 62 62 60 60 62 62
```

In a regular audio, it is possible that two audio patterns sound the same, but is not an exact match. The two audio file might be in different keys, but the same tone compared to the base key. For example, the note E in a C-major key would sound the same as F# in a D-major key as both notes has a difference of two tones from the base key, although exact match would prove them to be different. In other words, we can also conclude that the similarities between sounds are more determined by how the sound is overall rather than an exact match. So we can determine similarities by the differences between two consecutive notes instead of the exact notes.

By matching the differences, we can pinpoint the pattern more accurately when the pattern is longer and has more diversity of value. If the pattern is short though, the matches are more likely to be a false positive. With this method, audio samples that are pitch-atltered from the pattern (or likewise) can be detected correctly.

VI. CONCLUSION

Simple audio, for example ones in MIDI format (.mid extensions) can be easily compared by the string matching approach, while for more complicated forms of audio, there are several complicated steps that should be taken to render the audio forms (either waveform or spectrums) as something that can be compared. The rendering of these audio forms in general can be done by Fast Fourier Transform.

In the case of simple audio, like the MIDI file, we can extract the notes from the file by using an extractor. After extracting, depending on the complexity of the MIDI we might have to rearrange the data, then we can compare the rearranged sequence of notes. The comparison can be done by any algorithms for string/pattern matching. Although because we are comparing audio, there are several characteristics of audio that we have to consider to make the matching more accurate.

VII. ACKNOWLEDGMENT

The author would like to thank her parents for their utmost support. The author would also give her gratitude to Mr. Rinaldi Munir, and Mrs. Masayu Leylia Khodra for the knowledge they gave in class and all their support on the course IF2211 Algorithm Strategies (Strategi Algoritma) for the last semester, and also for the chance to write this paper. Last but not least, the author would also like to thank other people who had given their help and support in any way that lead to the finishing of this paper.

REFERENCES

[1] R. Munir, Diktat Kuliah Strategi Algoritma, Bandung: Penerbit Sekolah Teknik Elektro dan Informatika, 2009.
[2] http://stackoverflow.com/questions/5651725/compare-two-spectogram-to-find-the-offset-where-they-match-algorithm?rq=1, accessed at16th May 2014, 12.30PM

[3] Marchand, Sylvain, Vialard, Anne, The Hough Transform for Binaural Source Localization, 2009

[4] http://www.personal.kent.edu/~rmuhamma/Algorithms/MyAlgorithms/StringMatch/naiveStringMatch.htm, accessed at 16th May 2014, 14.20PM

[5] http://stackoverflow.com/questions/604453/analyze-audio-using-fast-fourier-transform, accessed on 16th May 2014, 11.30PM

[6] http://www.youtube.com/watch?v=tQqQJ4NCt4A, accessed on 20th March 2014, then converted to mp3 with bitrate 192kbps

[7] http://www.phys.unsw.edu.au/jw/notes.html, accesssed on 17th May 2014, 13.10PM

[8] http://www.fourmilab.ch/webtools/midicsv/, accessed on 17th May 2014, 23.50PM

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, 18th May 2014

[illegible]

Felicia Christie (13512039)