Stellenbosch University

Honours project

Music identification tool: Stelhound

Project report

Matthew Jenje

supervised by
Prof. Brink van de Merwe

October 27, 2017

Contents

List of Figures

Listings

1 Introduction

Music is a daily part of every persons life. With so much music around and new songs being created every single day it is impossible to identify every song you might hear. Having the ability to determine what song is playing at any given moment is a valuable service with wide commercial appeal. Giving a user the ability to discover new music that they otherwise would never have identified is the ultimate goal of Stelhound, an open source music identification tool. The aim for this project is to create an open-source music identification tool using technologies and algorithms with other potential audio processing uses such as voice recognition.

Stelhound uses the basic principles of spectral analysis to identify and classify important points in an audio files information. These points are used to create custom song fingerprints which allow the program to search for matches between two different audio files. The specifics of the algorithm will be described in more detail in the Overview and Design sections.

There are currently music identification tools available such as Shazam and Sound Hound, however the techniques these companies use are proprietary and shrouded in ambiguity. The project aims to shed some light on the mystery surrounding these applications and to open the door to some great open source projects using similar algorithms.

This project does not intend to improve on the accuracy or efficiency of existing music identification tools but to act as a proof of concept for small scale audio pattern matching.

2 Overview

Stelhound is an application designed to allow a user to record sound from their environment with their device and have it identified. Once a song has been identified its information is stored on the device and the user is able to see the last 5 matches made by the program.

The program was created in Java 8 with the intent that if it were ever to be converted into an Android application the conversion process would be relatively straightforward.

The initial design and requirements documents written for this project were altered significantly and so no longer accurately reflect the final program. Mid way through the projects implementation it became apparent that certain design choices had to be altered and this affected the scope significantly.

2.1 Initial approach

The final system created for this project differs significantly from the original scope thus making most of the original requirements invalid. The initial plan for this project was to generate Parsons code [1] from an audio recording and use it to search through the Musipedia webservice. After spending some time on the project and working with the webservice it quickly became clear that Musipedia could not be used. The main reasons for this are listed below.

[1] Parsons code is a compact way to represent the melody in a piece of music, where each note is given a value of U,D or R (Up, Down or Repeat) depending on the note that came before it.

1. Parsons code has no specificity:
   Parsons code is a good representation of a song's melody and can provide a good understanding of the melodic pattern, however in popular music melodies are often replicated across multiple songs. This means that if two songs share the same melody they will generate the same Parsons code. This problem becomes even more prevalent when looking at small recordings which only contain part of a melody. This increases the likelihood of generating false positives and makes it almost impossible to ensure accuracy.

2. The Musipedia database is not transparent:
   The webservice has a large but incomplete database with no way of knowing its exact contents and reliability. Only partial Parsons codes are stored and there is no indication as to where in the song the Parsons code is from. This makes it very difficult to do any testing with accuracy. Not knowing the structure and contents of the database creates a layer of complexity that this problem does not need.

3. There is no way to do any kind of searching optimisation:
   The Musipedia webservice uses SOAP to do its API calls and an unknown searching algorithm to search through the database. In a program such as this one where search speeds are critical the programmer needs as much control as possible over the data. Without having direct access to the database the program can never be efficient.

Once it was clear that the approach needed to be altered a new system was devised. After some reading and research it was decided that the most accurate way to determine song matches was to create a custom Fingerprinting algorithm.

2.2 Modified approach

The modified approach needed to combat the three main issues highlighted above: specificity, database transparency and searching optimisation. To do this a custom fingerprinting algorithm needed to be created which could consistently reproduce fingerprints in noisy environments. Having read through both Wang [1] and Kurth's [2] papers it was decided that the most effective way to generate reproducible fingerprints was through the principles of spectral analysis.

Spectral analysis is the process of analysing data which falls within a spectrum, in this case audio frequencies between 0 and 20,000 Hz. The best known example of this is the spectrogram which is a visual representation of the time, frequency and intensity data in an audio file **SHOULD I INCLUDE SPECTOGRAM FIGURES?**. Full spectrograms contain too much information for rapid comparison so key information is picked and used to generate a much smaller fingerprint. By using all this information to create a fingerprint we eliminate the problem of specificity in the matching criteria, a major issue in the initial solution. This is gone into more detail later in the Design section.

Fixing the other two problems is done by creating a custom database which contains all the information a user might want. This provides database transparency and allows for a custom searching algorithm, both of which will be explained in depth later on.

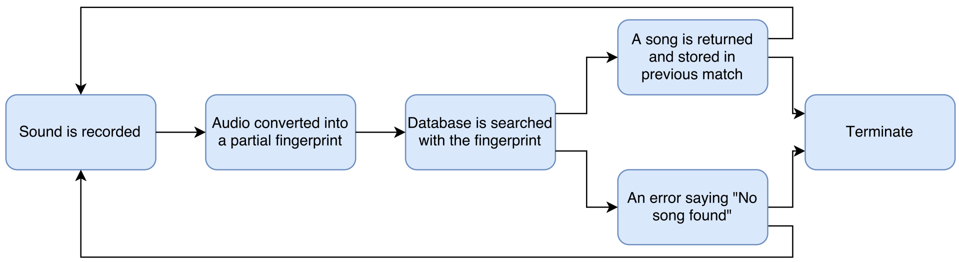

Figure 1: User pipeline

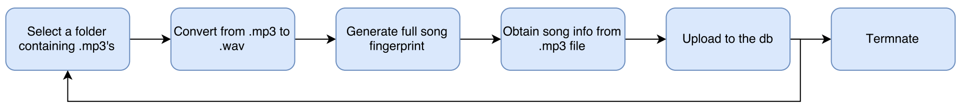

Figure 2: Administrator pipeline **MAKE BIGGER?**

With the addition of a database there is a second application which needs to be made which allows an administrator to add files to the database easily. This application will follow a similar process to the main application but will act on larger pieces of audio. Both the user and administrator pipelines are given in Figure 1 and Figure 2 respectively.

Once a song has been matched by the algorithm it is stored on the device as a previous match. This allows the user to then go back and have a look at the previous matches found using the program. This feature only remembers the last 5 matches found.

With the modified approach the goal was not to reproduce the performance and accuracy of Shazam. This is reflected in the non-functional requirements. It was expected that the program would have an accuracy of close to 75%, matching 3 out of every 4 songs correctly. The program was also expected to find matches within 25 seconds and display a message letting the user know if the recorded audio is not in the database.

2.3 The final requirements simplified

2.3.1 Functional requirements

- The system shall record audio through the devices built-in microphone
- The system shall convert audio into search-able values using a custom fingerprinting algorithm
- The system shall search through a local database checking for Substrings and storing the Offsets of any it may find
- The system shall attempt to match audio recordings to database entries and will return a result when confident

- The system shall store the 5 most recent matches locally allowing the user to see their previous matches

2.3.2 Non-functional requirements

- The system shall be able to determine whether there is or is not a match in the database within 25 seconds of starting a recording
- The system shall have an accuracy of approximately 75%, allowing 1 false positive in 4 matching attempts

This project has a simple concept which requires some intensive and complex algorithms. These algorithms and their implementations will be discussed thoroughly in the Design and Implementation section. **SHOULD THIS BE MORE FORMALLY STRUCTURED?**

3 Design/Implementation

The design of this system, as outlined in the Overview section, can be split into two applications: one for the Client and one for the Administrator. Both applications utilise the same underlying fingerprinting system but the Client searches and returns results whilst the Administrator uploads songs to the database. By using the same fingerprinting algorithm for both systems we ensure that fingerprints are consistent.

The main components of this program can be broken down as follows:

1. Client
    (a) Audio recording
    (b) Audio preprocessing
    (c) Fingerprint construction
    (d) Database searching
    (e) Returning results
2. Administrator
    (a) Reading in audio files
    (b) Audio preprocessing
    (c) Fingerprint construction
    (d) .mp3 information extraction
    (e) Database updating

The following sections will contain a full description of each of these components and their interactions with one another. Thereafter the implications of each component will be explored.

Before getting into the specifics of the program this document will go through some of the theory related to the fundamentals of sound, audio recordings and the Fourier transform.

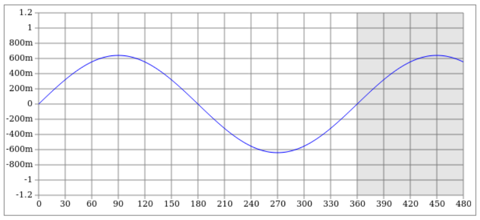

Figure 3: Pure sine wave

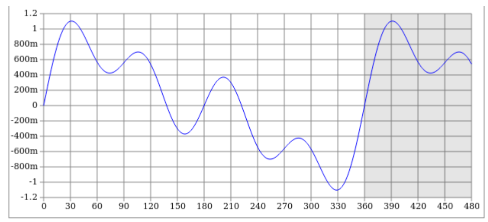

Figure 4: Typical sound wave

3.1 The fundamental properties of sound

To understand this project fully we first need to learn about what makes a sound or a song unique. Once we know that we can design a program which looks for these unique features.

Sound can be defined as a series of vibrations which propagates as an audible wave of pressure through a medium. In people and animals these pressure waves are picked up by the ears and interpreted by the brain into meaningful information. The number of times a pressure wave repeats itself is called the Frequency of that wave and the higher the frequency of the wave the higher the pitch of the sound. It is important to note that the pitch of a sound is not the same as the "loudness" of a sound. Loudness is determined by the amplitude of the pressure wave and not by its frequency. Sounds can occur at any frequency but humans generally only have the ability to hear between 20 Hz and 20,000 Hz.

Sounds which only have a single frequency are produced by *sine* waves with varying wave lengths and are almost never found in nature. Sounds almost always consist of multiple frequencies with differing amplitudes and can be thought of as a combination of multiple pure tone sine waves [3]. This distinction can be seen in Figures 3 and 4.

The important things to remember moving forward are that:

- Most sounds are made up of multiple frequencies which can span across a broad range
- Sound waves can contain multiple frequencies but only those that lie between 20 Hz and 20,000 Hz are relevant to humans
- Frequency relates to pitch
- and Amplitude relates to loudness

3.2 Audio recordings

Audio recording is the process of converting an analogue sound (a sound produced in the real world) into a digital one. This is not as intuitive as one might think because of how computers handle information.

When a sound is created and the sound wave propagates through the atmosphere the wave is continuous. In other words the sound wave can be divided into an infinite number of parts and each will still contain some information relating to the sound. This is a problem when thinking about computers as they do not contain an infinite amount of memory to store this data. We thus need to limit the number of times per second a recording device will *sample* the sound whilst keeping all of the frequency information which makes the sound unique. We do not want this rate to be too high as it will waste space and computation time but we also do not want it to be too low and miss important sound information. To solve this problem we use the *Nyquist-Shannon sampling theorem*.

The Nyquist-Shannon theorem states that the sampling frequency must be greater than twice the maximum frequency one wishes to reproduce **ADD REFERENCE**. So in order to get every frequency within the audible human spectrum we need a sampling rate of at least 40,000 samples per second (Hz). The sampling rate for our recording device is thus chosen as 44.1 kHz in accordance with industry standards.

If we were to sample at a lower rate our recording would not contain any of the higher frequencies in the human hearing range. This can be useful when very high quality audio is not a priority but can cause *aliasing* which gives the audio a "stutter" effect.

3.3 Fourier transform

After the program has made a recording through the microphone the information is stored as a byte array. This byte array represents the recording in the time domain, meaning every entry in the array directly correlates to a sample. Unfortunately this byte information is almost useless when it comes to audio matching. In order to get usable information we need to convert this data from the time domain into the frequency domain. To do this we use a *Fourier transform*.

An example of how a Fourier transform changes a signal is given in Figure 5. The problem with using a Fourier transformation on the microphone data is that all time information would be lost. For this reason a sliding window only performs the transformation on small chunks of each recording. This allows the program to keep the time information of the recording and extract the frequency data. Keeping the time information creates a new problem regarding performance. A Fourier transform now takes place many times a second instead of just once, increasing our computation time drastically. The solution to this problem is to use a modified version of the Fourier transform known as the Fast Fourier transform (FFT) [4].

The FFT gives exactly the same results as a normal Fourier transform but at a fraction of the computation time. Where a normal Fourier transform takes approximately $\mathcal{O}(N^2)$ time, the FFT only takes $\mathcal{O}(NlogN)$ time, where N is the number of samples being converted. It is important to note that from this point forward any mention of a Fourier transformation is in fact referring to the FFT.

The discussed theory will now be used to explain the design decisions and algorithms used in this project.

3.4 Client side

3.4.1 Recording

Recording sound is the first steps in the matching process and it is important that the recorded sound is accurate and contains all the fundamental frequencies needed for matching. There are several parameters that can be tweaked and changed in this area and testing was done to ensure that the ones which produced the best results were chosen.

As discussed previously in this document a recording needs to be sampled at least 40,000 times per second in order to contain all of the frequencies audible to the human ear. The

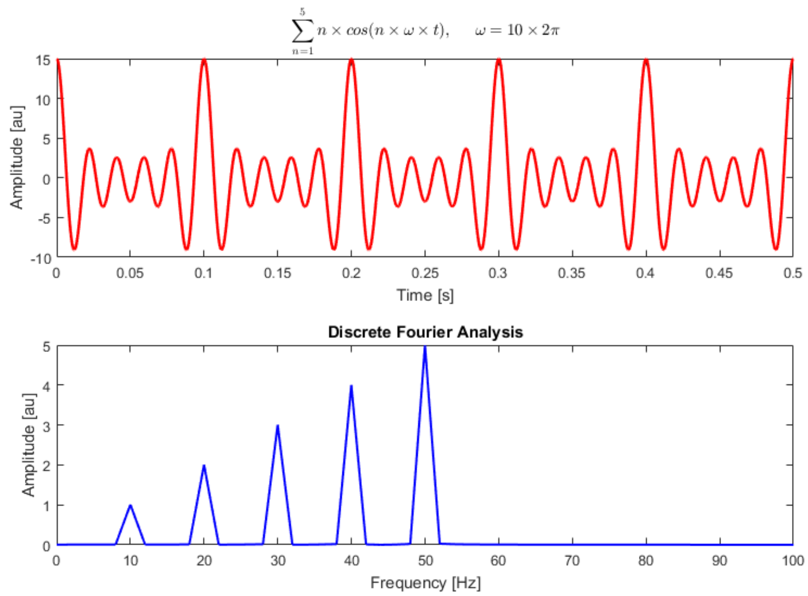

Figure 5: Fourier transform on cosine summation function [6]

sample rate is one of the parameters used by the system to determine the quality and clarity of the recording which will later be converted into a fingerprint. Any change to the parameters in the audio format will have a direct impact on the generation of the fingerprint. Another important parameter to consider is the *sampleSizeInBits*. This is the number of bits recorded per sample and in most recordings is standardised with a size of 16. **In this application a size of 8 is used to reduce the amount of noise picked up by the microphone as well as the amount of information needing to be processed**.

For consistency sake the same audio format is used on the server side when adding songs into the database. The audio format used can be found in Listing 1.

```
protected static AudioFormat audioFormating () {
     final float sampleRate = 44100.0f;
     final int sampleSizeInBits = 8;
     final int channels = 1;
     final boolean signed = true;
     final boolean bigEndian = false;
     AudioFormat format = new AudioFormat(sampleRate, sampleSizeInBits, channels,
         signed, bigEndian);

     return format;
}
```

Listing 1: Audio formatting for recordings

Recoding sound in Java is a fairly simple process which involves opening a line to a recording device and reading the information received into a byte array. This is all made possible with the `javax.sound.sampled` library. All audio recording is done in a separate thread from the main program so that sound can be recorded and searched for concurrently. After every second of recording the program generates 10 **MAY CHANGE**partial fingerprints, henceforth know as *Fingerprint fraction* , based on the audio. Each fingerprint fraction will then be used to search through the database. The fingerprinting and searching processes are described in more detail below.

3.4.2 Preprocessing

The audio data needs to be preprocessed before the frequency information can be extracted and turned into a fingerprint. The raw sound data first needs to be converted from an array of `bytes` to an array of `doubles`. This conversion is necessary for when we change the recording information from the time domain into the frequency domain using the FFT discussed earlier.

Converting the byte array is not as simple as parsing each value as a double and storing it in a new array. In order to preserve all of the sound information stored the conversion needs to take into account what audio parameters were used in its storage. For example if the audio was recorded using a `sampleSizeInBits` of 8, then the original byte array needs to be split up into segments of 8 bits each and then converted. Neglecting to do this will result in erroneous, inconsistent data.

After the conversion the new array is split up into chunks in order to preserve the time information of the recording. After testing with chunk sizes of 2, 4 and 8 KB it was found **that the best size was 4KB**. The chunk size in combination with the sample rate determines how many fingerprint fraction are produced per second. From the parameters used it can be worked out as $44,100/4096 = 10.76$ chunks per second **MAY CHANGE**. This value gets rounded down to 10 which means a small amount of information is cut off at the end of each second which provides extra time for FFT computation.

On each chunk of data a FFT is applied and the result is returned as two arrays, one real and one imaginary. The imaginary array given by the FFT can be used to determine the phase of the audio wave but in this application this is unnecessary and so the imaginary array is dropped. Each value in the array of real numbers is an amplitude associated with a frequency calculated by $(1+i) \times 10.766$ where i is the index of the value in the array. So for example if the maximum value in the array is 140 at position 55 then the frequency can be calculated as $(55+1) \times 10.766 = 602.896$Hz. This example has be visualised in Figure 6.

The values are multiplied by 10.766 because of a feature in Fourier transforms called *spectral resolution*. Spectral resolution is the degree of accuracy to which a Fourier transform can get a frequency and is calculated as $sampleRate/windowSize$. This effectively bins all of the frequencies we might get and provides a 10.7Hz margin of error. It does also mean however that frequencies obtained may not be exact but are rounded down to the nearest bin.

3.4.3 Fingerprint generation

The task is to now use the frequency, amplitude and time information gathered from the recording and preprocessing to generate a fingerprint fraction for each chunk of data. This fraction will then be used to search through the database and potentially find a match. The algorithm used to create the fingerprint fractions, and on the server side full fingerprints,

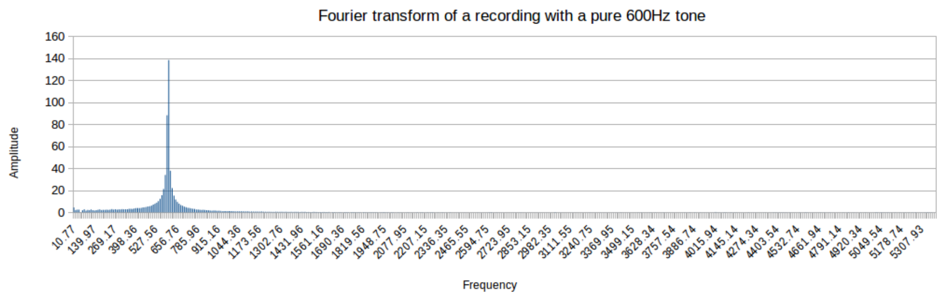

Figure 6: Fourier transform of a pure 600Hz tone

is simple and efficient using the amplitude information to determine which frequencies to store and which to ignore.

As discussed earlier most sounds found in the real world do not consist of just single tones, often having several peaks and valleys in the sound wave. This translates into the frequency domain as well, meaning that most sounds have several peak frequencies used in their construction. Generating fingerprint fractions from just the frequency with the highest amplitude would be too general and produce too many false positives in the searching and matching phases. At the same time using too many of the top frequencies would produce results which could not be repeatable if these peak frequencies were were not much louder than the average. Looking at the full audio spectrum for loud frequencies also takes too much computation time. In general songs exist in several frequency bands such as the baseline being 10-100 Hz, the low range being 100-200Hz, the mid range being 200-800Hz and the high range being anything above this. Using similar principles a fingerprint fraction can be generated by getting the highest frequencies between certain ranges and then combining them together. This is the logic behind the fingerprinting algorithm.

After testing to see which frequency ranges gave the best results it was decided to use **GIVE RANGES**. The testing methodology can be seen in the testing section: Parameters.

Below is the algorithm used in generating song and recording fingerprints.

1. The array of amplitudes received from the Fourier transform is iterated through to the 465th entry (5000Hz).

2. Each item in the array gets checked against the current maximum amplitude in the current range. If the amplitude is higher both it and its related frequency are stored. If not the next value is checked.

3. Once every value has been checked we now have the frequencies with the highest amplitudes in each of the 4 **MAYBE NOT 4** ranges. Every value which is not above the mean of these values is rounded down to the lower bound of the range, effectively zeroing it out. This is done to prevent weak, erroneous tones from being used.

4. The dominant frequencies all go into a function which concatenates them together, forming the fingerprint fraction.

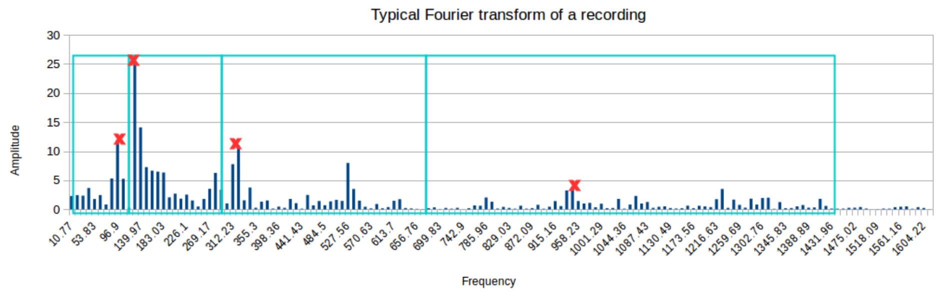

Figure 7: Visualisation of the fingerprint fraction algorithm

5. Steps 1-4 happen for each chunk of the recording and get added into an `arrayList`. Once every chunk has been utilised the algorithm completes.

6. The final fingerprint is created by doing a `toString()` operation on the `arrayList` of fingerprint fractions. An example output for half a second would be `[880440118074, 880386220110, 880386220110, 880408204110, 880440204110]` **CHANGE IF RANGE CHANGES**

An example of how the fingerprint process is done can be seen in figure 7. Each of the cyan coloured rectangles represents a frequency range and each of the red x's represents the dominant frequency in that range. From there the algorithm would determine that the frequency in the fourth range is below the mean and so is converted to the lower bound of its range. The other three would keep their frequencies as they are above the mean.

3.4.4 Searching

During the fingerprinting process 10 fingerprint fractions will be generated per second. Over the course of 25 seconds of recording this amounts to 250 fractions being created. These fractions need to be used to search through the database quickly and accurately whilst remembering that there is a good chance individual fingerprint fractions may occur in multiple songs as well as in multiple positions in a single song. While all of this is happening the search also needs to be robust, accounting for noise during recordings.

The first step in accomplishing all of the goals set out is determining the matching algorithm. Fingerprints are stored as strings in the database meaning that the searching algorithm needs to be substring based. The best algorithm for this task is Boyer-Moore with a time complexity of $\mathcal{O}(nm)$ in the worst case and $\Omega(n/m)$ in the best case, where m is the length of the text and n is the length of the fraction. The Boyer-Moore algorithm is perfect for this application as it specialises in finding matches where the substring is much shorter than the full string. The algorithm works by going back to front, checking if the current character is the same as the last character in the search and if it's not, skipping forward [7]. This makes the algorithm very efficient. A small modification is made so that once a match is found its offset in the fingerprint is stored and the algorithm continues. The pseudo code for the algorithm can be found in Algorithm 1 [8].

For each fingerprint fraction the Boyer-Moore algorithm needs to search through every song in the database. This means that the larger the database, the longer it takes to search

Algorithm 1 Pseudo code for the Boyer Moore algorithm

```
procedure COMPUTE FUNCTION LAST
arrayList a
i ← m − 1
j ← m − 1
Repeat:
    if P[j] = T[i] then
        if j = 0 then a.add (i)
        else
            i ← i − 1
            j ← j − 1
    else
        i ← i + m − min(j, 1 + last[T[i]]
        j ← m − 1
until i > n − 1
return a
```

through the entire database meaning that this solution is not scalable. The programs solution to this problem is to apply concurrency. The database can be divided into sections with each section performing its own search on a subset of the data. This makes the solution more scalable but potentially very expensive. This is an area for further research in the project.

After running the searching algorithm through every song with each fingerprint fraction the result is a list of songs with positions in their fingerprints that had matches. These positions, or *offsets*, represent potential matching locations where the recording could be taking place. This information is used when determining whether a song is a match or not.

3.4.5 Matching

The first instinct when searching for correct matches is to pick the song that gets the most fingerprint fraction matches over the course of the recording. This can give a good indication of which song might be correct but is a poor metric for determining exact matches. Part of the reason for this is because of how repetitive most songs are. In popular music a song will often have the same baseline and chorus in multiple positions and these areas have a high chance of containing similar fingerprint fractions. This would then cause issues for songs of varying lengths. For example if the song the user is trying to match is only one minute long but there is a similar song in the database that is five minutes long the program is more likely to pick the longer song and produce an incorrect match. It is for this reason that timing needs to be taken into account.

Offsets retrieved from the searching algorithm are in terms of character positions in the string and are hard to read and use accurately. For this reason the first step towards getting matches is converting the character offsets into fingerprint fraction offsets. The below equation represents the conversion where h is the character offset in the fingerprint, $cPerSec$ is the number of characters generated per second of audio and $fPerSec$ is the number of fingerprint fractions generated per second of audio.

$$\text{fracOffset} = (h/cPerSec \times fPerSec) + ((h \mod cPerSec)/cPerSec \times fPerSec) + 1$$

For every second of runtime fingerprint fractions are generated and searched for, giving a list of offset values. The easiest way to incorporate timing information into this data is to subtract the time in the recording from the suspected time in the matched fingerprint. This is done by subtracting the number of fractions recorded from the offset of the matched fraction. The value is then stored in a `HashMap` where the key is the result and the value is a counter. Every time the same key is made by subtracting these values the counter is incremented. By subtracting the values and storing this number the algorithm is effectively running a side-by-side comparison between the song's fingerprint and the recording's fingerprint, incrementing a counter every time that the values are the same. The algorithm can then find a match by seeing which song has a key with the largest counter value. If there is not a conclusively large counter then the algorithm will continue running until either the time limit is reached or a significantly large value is found. To get a better understanding of how this algorithm works figure 8 has been given as an example:

The figure shows two arrays of coloured blocks where each colour is representative of a unique fingerprint fraction. The large array represents the full fingerprint of the song being matched against and the small array represents the fingerprint generated from one second of recording. The position chosen in the songs fingerprint is arbitrary. The example is explained step-by-step below.

1. A search for the fraction at position 1 in the recording is conducted through the song fingerprint
2. A match for this fraction is found in the songs fingerprint at position 115
3. The offset value is calculated $(115 - 1 = 114)$.
4. There are currently no stored values so a new `HashMap` entry is made with 114 as the `key` and 1 as the `value`.
5. The rest of the fingerprint is searched for another instance of the fraction but none is found. The algorithm then resets and does the same process used on the fraction at position 2.
6. A match is found at position 112
7. The offset is now calculated as $112 - 2 = 110$ and stored.
8. This algorithm continues until the end of recording and produces the resulting map which shows 114 to have the highest counter.

After running this algorithm for a while each song should have a large `HashMap` of keys and values. Each song's maximum counter is compared and the song with the highest maximum is then checked against the second highest. The algorithm decides if a song is a good enough match when the next closest song is less than 80% of the maximum **COULD CHANGE**. Once a definite match is found the algorithm completes and sends the top ten matches to get displayed to the user. **MIGHT NEED EXPANSION**

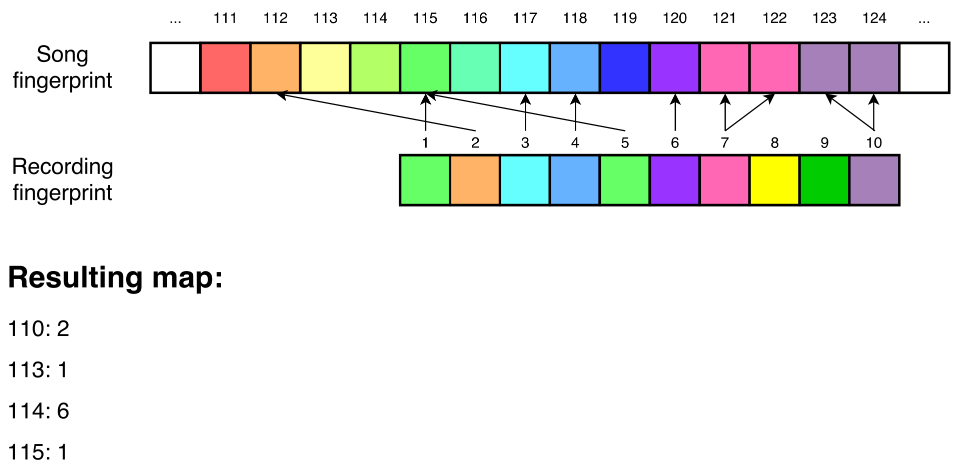

Figure 8: Visual example of the time inclusion matching process

3.5 Server side

The Server side application has many similarities to the Client in how it generates song fingerprints. In cases where there is a function overlap between the two applications the reader will be directed towards sections in the Client documentation in order to reduce repetition.

The server application was a late but necessary addition to the programs requirements which allows an Administrator to add songs to the database individually or in large groups. This application follows the pipeline displayed in figure 2. Each step in the pipeline will be fully explained in the sections below.

3.5.1 Reading in files

The first step in the administrator process is selecting a file or folder to upload. To do this the user is given a simple *GUI* (Graphical User Interface) which uses `JFileChooser` to select a file location on the users device. If a folder is chosen the application will go through every .mp3 file in that location and upload them one by one. If the user just selects a single .mp3 file the application will remain open in case the user wishes to upload a file from a different location.

For each file uploaded a process of conversion needs to take place from .mp3 to .wav. This needs to happen for a couple reasons:

1. .mp3 files are encoded as stereo files and the program requires files in mono [2]
2. Extracting the sound information from a .mp3 file is much harder than extracting the information from a .wav file because there is no standard .mp3 encoding type.

The process of converting an .mp3 file to a .wav file is fairly simple, using the `javax.sound.sampled` library and the audio format discussed in listing 1 to change the data in the file from its

[2]Stereo sound means that the audio was recorded with multiple microphones on different tracks whilst mono means that the audio is only on a single track.

native format to the new format. This conversion effectively makes it so there is no audio difference between sound read in from a file and sound recorded through the microphone.

The generated .wav file is now converted into an array of bytes which is the same format as the raw microphone recording. The byte array is now converted into a full song fingerprint in the same way as the raw microphone recording. For clarification on this process please read section **3.4.3 Fingerprint generation**.

3.5.2 DB storage and management

The database was designed to store all the information the user might want as well as the important matching information. For these reasons the database stores the song title, artist, duration, album art and fingerprint for each song. Inputting all of these fields manually would take too much time and would prevent the quick uploading of files needed for testing. This is the reason why the administrator is only allowed to upload .mp3 files.

.mp3's are special in that they are highly compressed and can contain information about the song in the header, the first few bytes of information. Unfortunately there is no standard encoding for .mp3 files meaning that the song information can be stored in different parts of each individual header. In order to reliably get song information from each file the program uses a Java library called "MP3agic" [9]. This library has the ability to determine the kind of encoding an .mp3 has and then extract the desired information. Filling the database is then as easy as using this library to retrieve the desired fields from the song and then using a query to upload the values.

In the event that a song does not have the desired information stored in the header the system automatically assigns a default value of "No value". If there is no album artwork available the system assigns a default image.

TALK ABOUT THE KIND OF DATABASE

4 Testing

In testing this program there were two main objectives, testing to make sure that the program functioned as it should without errors and testing all of the parameters to get an optimal solution. In effect the first kind of testing it to make sure that the program has no computation errors or interface bugs and the second type of testing was to try optimise the searching and matching potential of the program. Both of these testing methods have been explained in their own sections.

4.1 Program Correctness

There are many complex functions in this program, all of which need to be thoroughly tested to make sure there are no programming errors. The easiest way to do this in Java is to create a JUnit test suite. JUnit tests allow the programmer to specify expected output from functions within the program, failing a test if the expected output does not match the actual output.

A JUnit test suite should ideally cover 100% of a programs code. Code coverage is measured by how many lines in each of the program's files are executed versus how many are not, thus if all the lines in the program are executed then coverage is 100%. To measure the coverage in this program the tool EclEmma is used. EclEmma is an Eclipse plug-in which has the ability to highlight lines of code in different colours depending on whether they were executed or not.

Unfortunately there are some functions in the program which cannot be automatically tested, such as the correctness of the audio recording. The correctness of functions like this can however can be inferred further along in the programs pipeline as well as with manual testing. In this case a function was created to playback the audio recorded to listen for audio artefacts. This is not an ideal situation but will still give a reasonable estimation of the functions correctness.

After conducting **X TESTS** the code coverage report came back as **X PERCENT COVERED**. **EXPLAIN IF GOOD OR BAD**.

There were two different kinds of tests designed for this program: typical and atypical tests. The typical tests attempted to run the program with the average user experience in mind, running the program as it was designed to run. The atypical tests aim to provide inputs that the system would not expect. After creating and passing all all of the tests I am as confident that the program will not break in the hands of a user.

4.2 Parameters

Once the correctness of the program was tested the next step was to test to see how to improve the accuracy of the program, trying to reduce the number of false positives and false negatives. There are several programmatic parameters which directly affect the way fingerprints are created and they all need to be tested against one another. The parameters included things like how many fingerprint fractions to make per second, which frequency ranges to use when creating each fingerprint fraction and the number of bits used to store recordings. When changing any of these parameters the fingerprint generator produces a different output making manual testing very difficult and time consuming. Because of this a brute force testing function was created which goes through every combination of parameters available. Each time a parameter changes the function creates a new set of fingerprints for 10 different songs and then tries to match snippets for each back to the original. The results are then saved to a file to be later examined. Once every combination of parameters has been used the program terminates.

By doing testing like this we guarantee that we will get the best parameters possible. The full list of parameters tested and their final decisions have been given below:

Table 1: My caption

Variable	**Zeroing rule**	**Frequency ranges (Hz)**	**No. of ranges**	**Chunk size**	**Bit depth**
Options	No zeroing	55,110,220,440,880,1760	2	2048	8
	20% of the max	107,214,418,836,1672,5467	3	4096	16
	40% of the max	40,80,120,180,300,600	4	8192	
	60% of the max		5		
	>= mean		6		
Best					

FIX TABLE WIDTH, ADD DESCRIPTION FOR RESUTLS, ADD RESULTS

5 Discussion

Some of the solutions implemented in this project are not as optimised as they could have been, namely the searching and storing off song information in the database. The main issue with the searching is that it is effectively a linear search, checking each song individually for matches. Because of this the solution will not scale well with a much larger database. I attempted to circumvent this problem by using parallelism but this solution does not fix the underlying issue. After doing some research I discovered a potential method to correct this problem which involved hashing. The general principle behind this was to group fingerprint fractions together which would compress the size of the fingerprint. The data would then be stored as an inverted lookup table which could be searched far quicker than the solution implemented in this program. Unfortunately this solution was only discovered late in the projects cycle and there was no time to implement it. This is one of the things that could be attempted in the future.

The initial scope of this project ended up being completely different to the final product but this was due to a misunderstanding of the resources that were intended for the project. If I were to do this project again I would not waste as much time as I did on the initial scope. There was also a lot of time spent on debugging due to errors I was getting while trying to match songs to recordings. Initially I had no strategies for countering these errors but knowing what I know now I would be able to solve the problems within a few days. Part of the reason I had the errors in the first place was because of inadequate continuous testing and that is something I would improve upon if I were to do this project again.

The design decision to create this program using Java 8 was slightly misguided and in hindsight I would have done this project in Python 3. The initial reason for using Java was so that the completed program could be adapted into an Android application with relative ease. Having not converted the program into an application it would have been far easier to do some of the computations in Python with Numpy's built in FFT functions. It would have also been much easier to visualise some of the diagrams used in the documentation of this project.

Compared to other solutions like Shazam and Soundhound this piece of software does not outperform them, however this was never the aim. Shazam was founded in 1999 with the first working version of the service available in 2002. Since then there have been many improvements to the service and there was no way that my own service could outperform theirs in just a year. The aim was to alleviate some of the mystery surrounding these applications and I believe that this program has done that.

6 Future Work

There are many things that could be done to improve the services accuracy and usability. The ones I believe to be the most useful have been listed below.

- Implement the better storage and searching solution mentioned in the Discussion section.
  The bottleneck of this entire application is based on searching through the database. By doing more research in this area the application could be sped up greatly. Any improvement here could also be of great use to other applications which require speed in searching through very large databases.
- Host the database online:
  Having access to the database from any location is not a hard task but would be useful if this application were to ever be deployed. It was not implemented in this project as it fell out of the immediate scope.
- Create an equivalent mobile application:
  A mobile application would greatly improve the programs accessibility and usefulness.
- Create a partner application which using song lyrics to search through a database instead of sound frequencies:
  This would be an interesting way to compare the accuracy of matching music based on frequencies to that of lyrics and could provide some kind of middle ground for higher matching accuracy in general.
- More research into different identification techniques:
  Finding an alternative approach which had higher accuracy would be beneficial in many areas of computer and data science. The results of this research could be used in applications like voice recognition and audio information extraction.

There are many more areas for future work on this application as it extends far beyond the realm of music recognition. With another year or two this program could be an open source all purpose audio matching program which was not limited to music.

SHOULD I ADD A CONCLUSION?

Glossary

.mp3 An audio file format for compressed data. 6

.wav An audio file format for uncompressed data. 15

array A datatype used in programming which stores a series of objects all of which are the same size and type . 8

EclEmma A plug-in for the eclipse IDE which displays and highlights code coverage. 17

Eclipse An IDE used in writing, compiling and running Java code. 17

Fingerprint The unique identifier created for each audio recording and audio file. 4

Fingerprint fraction A small subset inside a fingerprint which represents the frequency information of 0.1 seconds of recording. 10

HashMap A datatype built into the Java.util package which allows the storage of data as key and value pairs. 14

Java A programming language which supports Object Orientated Programming. 3

JFileChooser A function built into the java.io.file library which allows the selection of files on the system. 15

JUnit A method of testing specific to Java. 16

Musipedia An online webservice containing songs and their Parsons codes stored in a database. The API can be accessed through the SOAP protocol. 3

Offset A numeric value dictating the distance between the beginning and a given index in an array/ list. 5

SOAP A protocol specification for exchanging structured information in the implementation of web services in computer networks. 4

String A datatype in Java which represents a finite sequence of characters. 20

Substring A partial section of a larger String. 5

References

[1] A. Wang,"An industrial strength audio search algorithm," *inProc.ISMIR*, Baltimore, MD, 2003, pp. 7–13.

[2] Kurth, F. and Muller, M. (2008), "Efficient Index-Based Audio Matching".*IEEE Transactions on Audio, Speech, and Language Processing*,16(2), pp.382-395.

[3] Dittmar, T. (2013). *Audio engineering 101.* Waltham, Mass.: Focal Press, pp.6-55.

[4] Rahman, M. (2011). *Applications of Fourier Transforms to Generalized Functions.* Great Britain.: WIT Press, pp.129-156.

[5] Columbia University. "FFT.java Source File".
http://www.ee.columbia.edu/ ronw/code/MEAPsoft/doc/html/FFT_8java-source.html [Online; accessed April 2017]

[6] DaveSGage, CC BY-SA 4.0,
https://commons.wikimedia.org/w/index.php?curid=53372625 [Online; accessed October 2017]

[7] Boyer, S. (1977), "A Fast String Searching Algorithm".*Communications of the ACM*, Volume 20, Issue 10, Pages 762-772

[8] Roahid Bin Muhammed "Boyer-Moore Algorithm"
http://www.personal.kent.edu/ rmuhamma/Algorithms/MyAlgorithms/StringMatch/boyerMoore.htm [Online; accessed April 2017]

[9] Patricios, M "mp3agic".
https://github.com/mpatric/mp3agic [Online; accessed April 2017]