Audio Extraction and Similarity Search

Vlad Limbean

IT University of Copenhagen

Algorithms Specialization

A thesis submitted for the degree of

Master of Science in Software Development and Technology

2018

1. Reviewer: Name

2. Reviewer:

Day of the defense:

Signature from head of academic committee:

Abstract

This master thesis paper describes and documents the open-source implementation and expansion of a landmark-based algorithm.

Acknowledgements

I would like to thank Martin Aumüller for the excellent guidance, supervision and professionalism.

Ivan Mladenov for endearing support and friendship.

Gorm Galster for many genuinely helpful suggestions.

Furthermore, I would like to thank all of the unsung heroes who support the many Python libraries employed in this project.

Moreover, a big thank you for all the people who support Latex.

Finally, I would like to thank my mother, Tatiana Oană, for unending support and faith.

Contents

List of Figures **v**

List of Tables **vii**

1 Introduction **1**

1.1 Audio Fingerprinting, GridHash & Music Information Retrieval 1

1.2 Intellectual Property Controversy 1

1.3 Motivation 2

2 Research question **3**

2.1 Research Question 3

2.2 Overview 3

2.3 Methodology 4

2.3.1 Implementation Process 4

2.3.2 Tools & Libraries 5

2.3.3 DejaVu 6

3 Algorithm Analysis **9**

3.1 The Landmark Algorithm 9

3.1.1 Database Schema 9

3.1.2 Audio File Indexing 10

3.1.2.1 Identifying Time-Frequency Peaks 12

3.1.2.2 Hashing Time Frequency Peaks 14

3.2 Audio File Recognition & Search 15

3.2.1 Matching Audio Files 15

3.2.2 Variations in Fingerprint Collision 15

3.2.3 Refining the Matched Results 17
3.2.4 Weighing Function 18
3.2.5 Weight Cut-off Function 20
3.3 GridHash 21
3.3.1 Grid Construction & Morphology 21
3.3.2 minHashing the Grid 25
3.3.3 Grid Settings & Similarity Accuracy 25

4 Evaluation & Results **27**
4.1 Testing the Landmark Implementation 27
4.1.1 Evaluation Results - Precision & Recall 28
4.1.2 Evaluation Results - Sensitivity 32
4.2 Testing GridHash 33
4.2.1 Results 33

5 Applications and usages **35**

6 Further research **37**

7 Conclusion **39**

References **41**

List of Figures

3.1 Database Schema 10
3.2 Two-channel waveform of "Lark's Tongues in Aspic, Part One" 11
3.3 Spectrogram of "Larks Tongues in Aspic" 12
3.4 Spectrograms with different peak density 13
3.5 Progressive Fingerprint Matching Rate 16
3.6 Scatter plot of frequency over time peaks 22
3.7 Grid structure of time frequency peaks 23
3.8 Frequency time grids at different tolerance values 24

LIST OF FIGURES

List of Tables

3.1 Small section of candidate results 19
3.2 Results sorted by weight 20

4.1 Preliminary results over 483 audio tracks 29
4.2 Results after Landmark algorithm improvements - wave files 30
4.3 Results after Landmark algorithm improvements - MPEG music files . . 31
4.4 Results for Landmark specificity - WAV music files 32
4.5 Results for Landmark specificity - MPEG music files 32

LIST OF TABLES

1

Introduction

1.1 Audio Fingerprinting, GridHash & Music Information Retrieval

Music information retrieval deals with the extraction, processing and interpretation of audio data. Tasks range from simple endeavors like feature extraction, filtering and watermarking all the way to the difficult and not-yet-solved problems like music transposition, instrument detection, cover song detection and music recommendation.[1]

This paper covers aspects of music information retrieval, with emphasis on audio fingerprinting. This is done for the purpose of similarity search. The focus is on expanding a known Landmark-based algorithm with a newly created audio recommendation algorithm. We will refer to this new system as GridHash.

1.2 Intellectual Property Controversy

The primary algorithm used in this projects' implementation is less-commonly referred to as the **Landmark algorithm**, or Landmark for short. More commonly it is know as the algorithm behind the popular mobile application Shazam.[2]

The Landmark algorithm is described in Avery Wang's excellently titled "An Industrial-Strength Audio Search Algorithm"[3]

Since the paper's publication, Shazam Entertainment Limited holds a patent on the algorithm. This has led to a few cease and desist letters to programmers in the open-source community who have attempted to implement Avery Wang's algorithm.

One such example is Roy van Rijn's Java implementation and subsequent cease and desist letter due to patent infringement in 2010. [4]

Additionally, Milos Miljkovic describes similar legal quandaries five years later in his appearance at the PyData conference in New York City. [5]

Intellectual property issues aside, the algorithm has seen tremendous attention from the open-source community. A quick web search of 'how to implement shazam' will lead to tutorials and repositories of the algorithm in different languages and states of completion.

More so, the original Matlab implementation of the fingerprinting algorithm authored by Dan Ellis is also available online. [6]

1.3 Motivation

Intellectual property rights controversy aside, the Landmark algorithm has seen endearing curiosity and attention from the open-source community at large.

Consequently, the author's motivation in building on top of the Landmark algorithm is driven by curiosity and appreciation. This drive is complemented by a desire to understand and expand on the original concept.

Consequently, this project involves an implementation of the Landmark algorithm, an assessment of the algorithm's efficiency, and finally, an newly developed recommendation algorithm titled GridHash.

2

Research question

2.1 Research Question

The research question for this project as entailed by the author's motivation: **how can the landmark based algorithm be extended and what are its applications?**

2.2 Overview

The core paper behind this project is Avery Wang's "An industrial-strength audio search algorithm". [3] This publication is the basis on which many Landmark-based algorithms are constructed, including the one implemented for this project.

The paper describes an algorithm capable of listening to an audio signal and determining whether it is part of an already known audio track. The classic example is using your phone to record a few seconds of something playing over the radio and getting back a song title and artist name.

TODO: include lit review on chroma, MFCC, deep learning based fingerprinting techniques

Aside from the welcome novelty of recognizing specific songs or audio snippets, the family of Landmark-based algorithms has seen additional interesting applications.

The more intuitive application is the algorithm's use in the field of digital rights management. [7] The proposed application in this instance is to used the algorithm to check if a specific ad was played over radio broadcast in the agreed upon form, at a specific time.

The reasoning being that ads can be sped up or cropped by broadcasters. With the use of a Landmark based algorithm it is possible to verify if an ad was broadcast per specification.

Inversely, the algorithm has recently been upgraded to feature image recognition. [8] The case being that app users can scan certain Shazam-specific QR coded products, like a can of soda. This will offer a custom experience to the user, allowing them to receive special offers and other commercially oriented type of targeted advertising.

More recent innovation involving the use of the Landmark algorithm concerns the identification of naval vessels. The concept is quite similar to the original Shazam application: submersible vehicles or naval vessels all have a specific audio pattern as they move through a body of water. The specific signature of any ship is granted by their motors, rotor pallets and general environment. In "Landmark based Audio Fingerprinting for Naval Vessels" [9] Hashmi and Raza propose the use of Dan Ellis's original Landmark algorithm in Matlab in order to maintain a small database of naval vessels types and their respective fingerprints.

The use current use case of identifying vessels via sonar involves a trained and experienced officer as the key factor in the decision process. The addition of a landmark based system makes a lot of sense given its robustness to noise.

2.3 Methodology

2.3.1 Implementation Process

The author's approach to this project is to put together a working instance of a Landmark-based algorithm built for easy customization and experimentation.

Once the core algorithm is functionally in place, a database of wave files will be indexed. The Game Developer's Convention creates yearly repositories of professional audio tracks for games, film and independent projects distributed by "Sonnis.com".[10] In addition, the author will use a collection of MPEG files for further experimentation.

The audio files will serve as the main basis for testing the algorithm. It is important to note that this project will not test large volumes of songs. The database contains 500 wave and XXX MPEG files as testing data. Testing is done to validate the functioning and relative accuracy of the Landmark implementation. Additionally, the same files are used in validating the functionality of the GridHash algorithm.

The GridHash extension focuses on extracting and hashing the features of whole audio files and storing them remotely as a list of coordinate. Each list is a low-dimensional, low-memory multi-set of audio features that uniquely identify an individual audio file.

By converting songs to a set representation one can use the minHash algorithm to compute the Jaccard similarity of one audio file relative to another. The GridHash algorithm involves a degree of fine tuning, from the extraction of the raw audio data, to the creation of the feature lists, but it shows promise in identifying similar songs without any metadata.

2.3.2 Tools & Libraries

The project is conducted entirely on an Asus Laptop running Windows 10 and sporting a 3.1 GHz Intel i5 7th Generation processor, 8GB RAM and 512 Gb of SSD memory.

In order to store wave files, the author makes use of a local instance of MySQL and a Seagate 2TB external HDD to handle the expensive storage of uncompressed audio.

The project implementation was achieved with the use of an open-source Landmark-based algorithm called DejaVu, authored by Will Drevo. [11] The project implementation has seen a lot of changes relative to the original library. The changes are addressed later in the analysis of the algorithm.

Python 3.6 is used throughout the process for quick prototyping and testing. The list bellow describes the libraries used and their purpose.

- **wave, soundfile, pyaudio, pydub**: Used for extraction of raw audio data and wave forms
- **numpy, matplotlib, scipy(filters & image morphology packs)**: used for processing FFT, spectrogram creation and visualization
- **hashlib**: used for SHA1 computation
- **pickle**: for storing data in specific formats
- **dataSketch**: for minHash implementation

2.3.3 DejaVu

The original DejaVu implementation is meant to mimic Shazam. It does so by allowing the use of a microphone to detect any audio signal for a set amount of time, generally a few seconds. The algorithm then searches in the database of indexed files if it can detect a match. DejaVu always returns an answer, and it does not support any cut-off function to indicate that a given signal is not present in the database.

DejaVu is written in Python 2.7 and as of the moment of writing still has a few critical bugs. The most notable one involves the main features extracted from audio files: frequency value and the corresponding time index. Each audio track is composed of a large number of such coordinates. In the cited version of DejaVu present on GitHub the time and frequency data structures get swapped. This does not render the algorithm useless, but it deviates sharply from the algorithm it models.

While Avery Wang's Landmark algorithm processes each song to mono with a frame rate of 8KHz and a 16-bit depth, DejaVu does not down sample and it uses the frame rate read from the audio encoding. If the file is raw, it uses a default frame rate of 44100 Hz.

The drawback to this is space complexity. By not down-sampling, each audio file will have tremendous weight in the database. Despite this, test results suggest no impact on accuracy.

TODO: cite pages in avery's paper

3

Algorithm Analysis

3.1 The Landmark Algorithm

The Landmark algorithm used for this project was ported from Will Drevo's Python 2.7 implementation and adapted to Python 3.6. It can fingerprint audio files then index these in a MySQL database. This process can be referred to as the learning phase of the algorithm. After the learning phase, the algorithm is be able to recognize indexed songs by listening to a slice of any audio track. If the audio track was previously fingerprinted and indexed, then the algorithm will recognize the track and return the title of the audio track.

In addition to the original DejaVu implementation, if the algorithm fails to recognize a track it will return the string "No results found". This could be a True Negative result, in which the queried audio track was not in the database, conversely a False Negative where the audio track is present in the database, but it was not recognized.

In the sections bellow we will be looking at how the database which holds the fingerprints is set up, how a file is fingerprinted and finally how it is recognized.

3.1.1 Database Schema

The MySQL database schema contains two tables. One table to hold audio files titled **"songs"**, respectively a second table to hold the fingerprint information of added audio tracks titled **"fingerprints"**.

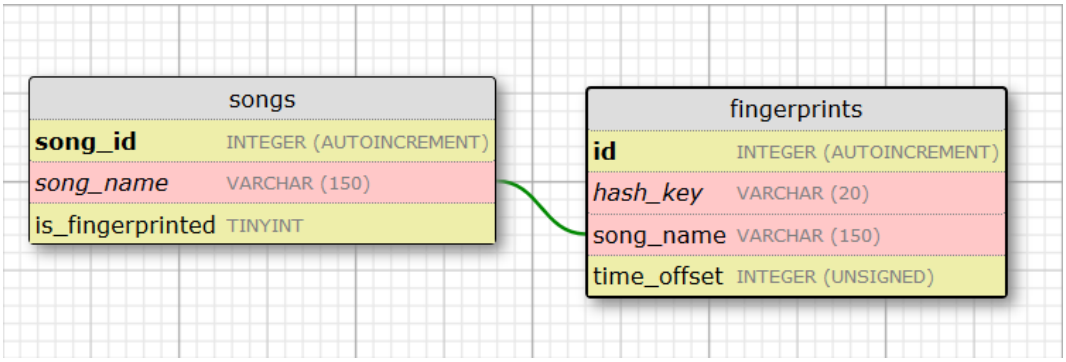

Figure 3.1: Database Schema

As described in figure 3.1, the **"songs"** table holds an auto-incrementing id number, a 150 character string for the audio file title, and an indicator on whether or not the file has fingerprints in the "fingerprints" table.

Similarly, the **"fingerprints"** table contains an auto-incrementing id, a 20 character hash-key representing a fingerprint, a 150 character song-name, and an integer time-offset value which tells us when in the respective audio file the fingerprint occurs.

To speed up query time an index is created for the hash-key column. Additionally, song-name serves as a foreign key linking the two tables.

3.1.2 Audio File Indexing

The first step is reading in audio format files. This is done with SoundFile [12] and PyDub [13]. Combined the two libraries allow the algorithm to read most available audio encodings including but not limited to wav, mp3, mpeg, ogg, vorbis, flac, aiff. Additionally, the libraries accommodate different bit-depth files ranging from the light 8, to 16 and 24-bit, and the heavy 32-bit depth. More so, the audio can be single-channel (mono) or two-channel (stereo).

The initial information we get from any audio file is the raw waveform data. In figure 3.2. we see the waveform of King Crimson's "Lark's Tongues in Aspic, Part One". The music is retrieved from a two-channel MPEG file. The two colors represent

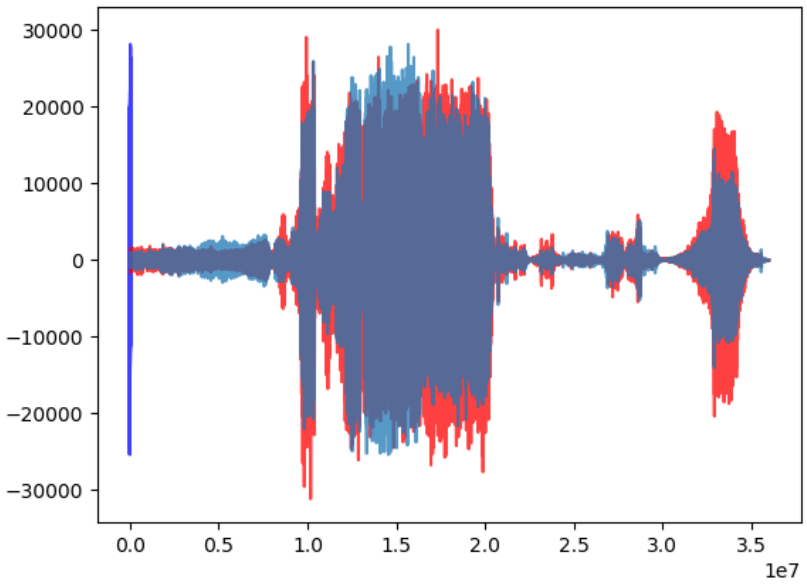

Figure 3.2: Two-channel waveform of "Lark's Tongues in Aspic, Part One"

the channels as they overlap.

Once fetched, the waveform gets processed using matplotlib's Fast Fourier Transform in order to generate a spectrogram, alternately referred to as a periodogram, of the respective audio file. This takes the waveform from the amplitude over time domain and represents it as a function of frequency over time. If the file is recorded in stereo, the process is done for each channel. The algorithm only takes into account the frequency spectral band, no other features as captured.

The Fourier transform is performed with a default frame rate of 44100 Hz, an FFT window of 4096 points and a Hanning window with an overlap ratio of 50%. The algorithm's default sampling rate of 44.1 KHz get overwritten if the audio file in question is sampled at any different rate. For example some of the wave files used in testing are sampled at 96000 Hz. This is done in order to allow audio editing and compression without significant loss in audio quality.

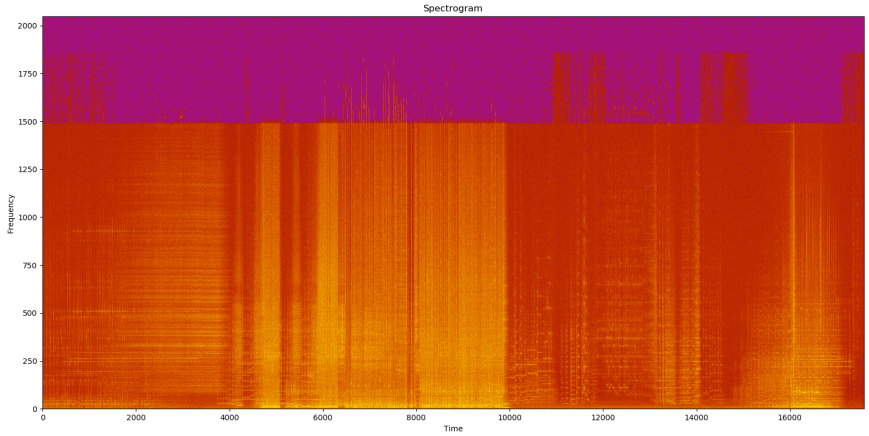

Figure 3.3: Spectrogram of "Larks Tongues in Aspic"

Figure 3.3 is a spectrogram of "Lark's Tongues in Aspic" by King Crimson as read from an MPEG file.

3.1.2.1 Identifying Time-Frequency Peaks

The relevant data points from the spectrogram are the frequency peaks and their respective time index. A scatter plot of these peaks over the frequency-time spectrum would look like the spectrograms in figure 3.4. The x-axis represents the time domain, the y-axis is the frequency domain. The blue dots are peaks.

The present implementation treats the spectrogram as an image. Peaks are determined within a neighborhood area of the image. This is achieved using Scipy image morphology and filter methods [14].

The neighborhood for peak detection is set to a value of 20. This means that an amplitude point must be maximal across an area of 20 points in order to be selected. Larger values would yield fewer peaks potentially impacting the accuracy of the algorithm. This is visualized in Figure 3.4. The top image has a peak neighborhood value of 20 points, yielding 11658 peak points. The bottom image has a peak neighborhood of value of 40 points, yielding 3377 peaks.

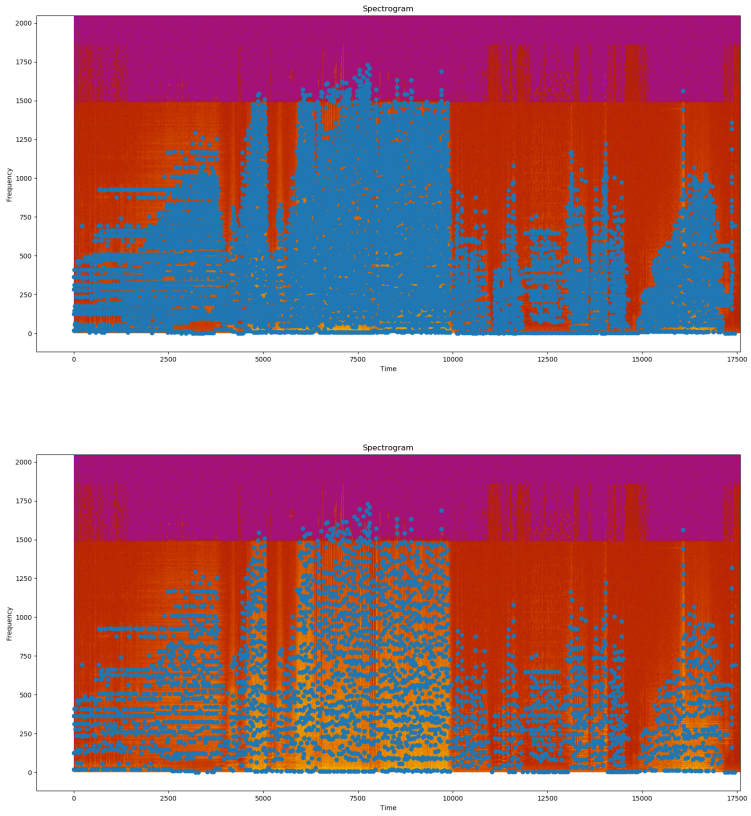

Figure 3.4: Spectrograms with different peak density

3.1.2.2 Hashing Time Frequency Peaks

Once the time and frequency peaks are extracted from the spectrogram they are ready to be converted to hash-keys. Each peak coordinate is paired with a determined number of adjacent other peaks. The number of pairs is referred to as the **"fan value"**. The current implementation has the number set to 15 pairs.

In more detail, each peak is paired with 15 other adjacent peaks. The peak against which the other adjacent points are checked is named **the anchor point**. Every peak paired with the anchor point must occur either at the same time or within a set upper temporal bound. In the current implementation each point must be in between zero and two hundred points on the spectrogram's time axis. Consequently, all peak points relative to an anchor points happen either at the same time with the anchor point, or in the future.

A hash is constructed by taking the value of the current frequency peak **f1**, an adjacent peak within the fan value range **f2**. Both, peaks have associated time indices **t1**, respectively **t2** out of which we derive $\Delta t = t2 - t1$.

Any fingerprint is the hash digest of **(f1, f2,** Δt**)** after it is passed through SHA1. Now, SHA1 produces a string of 40 characters. In order to reduce the space complexity, the original DejaVu implementation drops the last 20 bits of the digest. This implementation details has been also kept in this project.

Ultimately, the fingerprint will look like a string in the example below.

```
Example fingerprint:  89b64c1a491f804aa34b
```

The row format in the database will track the id of the row itself, the digest, the name of the corresponding file, and the time index value of **t1**.

```
Row format:  id, hash-key, audio track title, t1
Row example:  12, 89b64c1a491f804aa34b, 'example.wav', 76
```

3.2 Audio File Recognition & Search

In this section we are looking at how a section of an audio file gets recognized by the Landmark algorithm.

Recall, in order for the algorithm to recognize any audio track, it must first learn it. For the sake of this example we will assume that the algorithm has learned King Crimson's "Larks Tongues in Aspic, Part One". Consequently, we expect that after offering the algorithm a few seconds of the song, it will return us the song title.

3.2.1 Matching Audio Files

The algorithm listens to any audio format file. The signal may come from an analog source, say, the speaker of a radio. Alternately, the signal may be read from disk. In this project the authored has focused on providing the algorithm signals read from the local disk.

Now, the algorithm only needs a few seconds sample from any audio track in order to accurately determine what the track title is. It does this by passing the audio snippet through the same fingerprinting process a full track goes through. Consequently, if the algorithm produces a set of 100 fingerprints for the full track, it will produce a subset of around 10 fingerprints for roughtly a 10% section of the audio track.

The intuition is that the algorithm will take the subset of produced fingerprints from the audio snippet and cross-reference it with the fingerprints stored in the database. When a fingerprint from the subset matches one from the database, the name of the matching track is added to a list of results.

3.2.2 Variations in Fingerprint Collision

It is important to note that for smaller sections of an audio track the algorithm does not actually produce an amount of fingerprints with linear growth. Meaning, the algorithm does not produce 10 fingerprints by listening to 10% of an audio track which, in its entirety, produces 100 fingerprints.

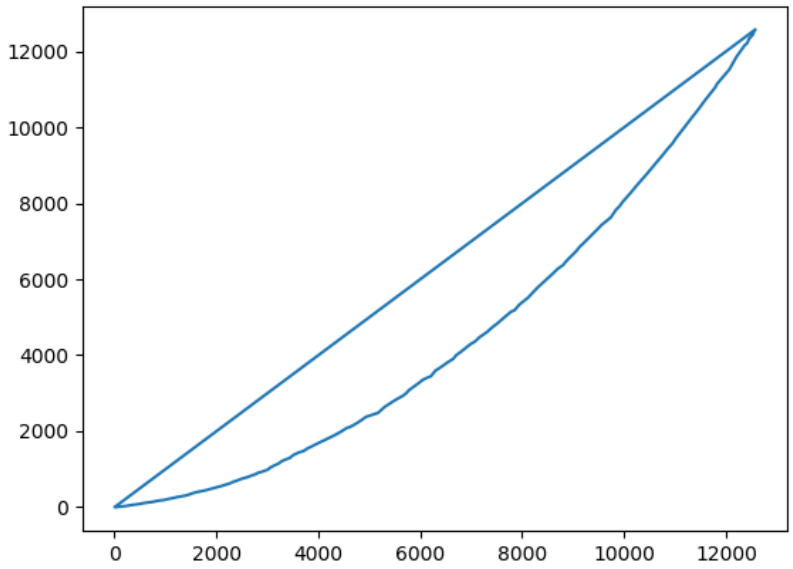

Figure 3.5: Progressive Fingerprint Matching Rate

This is due to the **peak neighborhood** value used to detect peaks points on the spectrogram of the audio track, respectively snippet. The author noticed through experimentation that snippets of audio tracks will contain some peaks that match the original track and some peaks unique only to itself. This phenomenon is caused by the image morphology package of SciPy which detects peaks based on a set constant area of any spectrogram. The consequence of this is that subsections of a spectrogram will yield many of the peaks present in its complete counterpart, however, due to having changed the structure of the spectrogram by slicing a subsection, the resulting part will also contain fingerprints unique to itself.

Hence, the question arose: if one generates fingerprints progressively over an audio track, how many fingerprints will collide with the approximate number of total generate fingerprints of the entire song?

Figure 3.5 describes the effect. The audio track used to generate this graph is a 2 minute and 11 seconds long recording of a quiet street with people chatting. The entire

song generates around 12300 fingerprints. The curved line represents the number of fingerprints that match the original track. This value is calculated and graphed incrementally starting at the 1st second up to the 131st second of the track.

Notice that for smaller snippets of the audio track the number fingerprints that match the actual set of all fingerprints at that point in time are a fraction of the number of fingerprints generated by the full track. It is only when the entire track is fingerprinted that the generated subset of fingerprints perfectly matches the super-set of fingerprints in the audio track.

3.2.3 Refining the Matched Results

So the algorithm listens to a section of an audio track and generates a list of fingerprints. The list of fingerprints is then cross-referenced with the database and a matching set of audio tracks is returned.

In this section we will look closer at how results are refined. The need for a refinement step is brought forth by the number of fingerprints in the database. The more fingerprints there are, the higher the chance of collisions with other audio tracks, and consequently incorrect results.

Now, let us recall how a fingerprint appears in the database. It contains an id, a hash-key representing the fingerprint, a track title and, importantly, a time index Δt. The time index serves as an indicator of where in the associated audio track the fingerprint occurred.

This serves us greatly in determining whether a matching fingerprint actually occurs in the same audio track or not. Consequently, when the fingerprint of a track is matched, we return a tuple consisting of: track title, and the Δt of the matching fingerprint in the database minus the time coordinate of the fingerprint generated.

$$Matching\_result = (track\_title, \Delta t - t\_input)$$

This is done because a fingerprint at time index 0 represents an exact match. The farther away we move from index zero the higher the probably that the fingerprint takes place at a different place, possibly in a different audio track.

For any query we will receive back a list of candidate results. Now consider we are using **SFX Large Wave Splash on Rocks 21.wav** as the basis for an example. This is a stereo 8.63 second wave file. From this track we give the algorithm 2 seconds in hopes that I will retrieve the correct answer.

A section of candidate results looks like the table 3.1. The data structure will have a time index, and a list of candidates. The numbers associated with each candidate represent the number of matches for that track at the specified time index.

The interpretation of the table goes as such: There are three audio tracks whose fingerprints perfectly match the song queried. These are all at index 0. The track 'SFX Large Wave Splash on Rocks 21.wav' has 1620 matching fingerprints. There are other matches available at index 935, 1501, 850 and so on. The candidate results have one or two matching fingerprints, an insignificant frequency count. The correct result then is 'SFX Large Wave Splash on Rocks 21.wav'.

3.2.4 Weighing Function

After listening to 2 seconds of 'SFX Large Wave Splash on Rocks 21.wav' the full list of candidate results contains 2920 time index points, each with their own list of tracks which have matching fingerprints. So how do we sift through these?

In order to return a correct result, or no result at all, the author has written a weighing function. To begin with, we wish to prioritize items which are at time index 0, or close to 0. Secondly, we prioritize candidate results that have a high number of matches at any time index.

Table 3.1: Small section of candidate results

Time Index	Candidates
0	'EFX EXT GROUP Female Celebration Scream 01 A .wav': 1 'BluezoneBC0223noiseradiosignal029.wav': 1 **'SFX Large Wave Splash on Rocks 21.wav': 1620**
935	'EFX EXT GROUP Female Celebration Scream 01 A .wav': 1 'EFX EXT Bulkhead Door O_C Close Med 04.wav': 2
1501	'EFX EXT GROUP Female Celebration Scream 01 A .wav': 1 'GHOSTS PassBy Witchery Shiver.LR.wav': 1
850	'EFX EXT GROUP Female Celebration Scream 01 A .wav': 1 'BluezoneBC0214explosionwhooshe003.wav': 1
778	'EFX EXT GROUP Female Celebration Scream 01 A .wav': 1 'Upright Piano_Drone_Tension08.wav': 1
-1311	'EFX EXT GROUP Female Celebration Scream 01 A .wav': 1 'BluezoneBC0224sequence001.wav': 1 'Weapon Shot Spaceship Super Blaster04.wav': 1

For each time index and associated list of candidates we apply the following inverse-exponential weight formula:

$$weight = (e^{-|time\_index|} * 1000) + max\_frequency$$

Where:

e: Euler's constant

time index: the relative time distance between two matching fingerprints

max frequency: highest number of matches of an audio track at a specific time index

Once the weight is calculated we create a tuple: **(weight, time index, candidates)**

Table 3.2: Results sorted by weight

Weight	Time Index	Candidates
50.78706836786395	−3	'Medium Distance Machine, Clicking, Air Compressor 01.wav': 1 'Whoosh Transition 066.wav': 1 'Hvac,Ventilation,Exhaust,Industrial,Drone,Slight rumble,Loop.wav': 1 'PR CHAINS_SCRAPE 4_416.L.wav': 1 'PR LEATHER BACKPACK_JOG_416.L.wav': 1
136.3352832366127	2	'transition elastic 102.wav': 1 'EFX EXT GROUP Battle Approach 01 A.wav': 1 'EFX EXT GROUP Battle Celebration 02 A.wav': 1 'EFX EXT Shattered Glass Impact 10 A.wav': 1 'EFX SD Organic Gore Russle 05 B.M.wav': 1 'DetunizedInfiniteSouth24.wav': 1 'LM4 Crank Handle Various Rev 15 EDU172AB.wav': 1
136.3352832366127	−2	'BluezoneBC0227syntheticliquidlongtexture016.wav': 1 'explosion_large_08.wav': 1
368.87944117144235	−1	'c2.wav': 1 'BluezoneBC0226hitimpact004.wav': 1 'Latchlocker,cable,steel,wood,room,channel,slide,catch,alt2.wav': 1 'punch_general_body_impact_03.wav': 1 'Reverse_Production Element_Piano001.wav': 1
368.87944117144235	1	'BluezoneBC0223noiseradiosignal025.wav': 1 'Cardboard Tearing 02 Slow.wav': 1
2620.0	0	'EFX EXT GROUP Female Celebration Scream 01 A .wav': 1 'BluezoneBC0223noiseradiosignal029.wav': 1 **'SFX Large Wave Splash on Rocks 21.wav': 1620** 'DetunizedInfiniteSouth24.wav': 1 'Robot_Power On_03.wav': 1

Table 3.2 describes a small segment of the refined and weighted candidates. The row with the highest weight, 2620, is at index 0. In that row, the candidate with the largest amount of matches, 1620, is most likely the correct result.

3.2.5 Weight Cut-off Function

The next question is: at which weight value do we consider a result to be an inaccurate prediction?

One matching fingerprint at time index 0 will generate a weight value of 1001. Normally, correct results will have a weight higher than 1000, as we covered in the example above. However, the current cut-off time index was chosen to be at time index 1 or

-1 and a maximum result frequency of 1 matching fingerprint. These parameters will result in a constant of 368.87944117144235.

A correct result will have a weight higher than this constant. Weights lower than this value will return a 'No results found' thus improving the algorithm's ability to predict a result with accuracy.

3.3 GridHash

The GridHash search functionality is entirely new and developed as an extension to the Landmark algorithm.

As a preamble, like in the previous section where we covered frequency and time co-ordinates on a spectrogram. These coordinates are resulting points on a two-dimensional graph and are not converted in milliseconds respectively hertz. They are merely treated as points on a plane.

In this section we will explore how the GridHash method makes use of the same information as Landmark. It is interesting to note that Landmark makes its predictions by creating a large hash-table type structure where the many fingerprints serve as keys, and the values are a list of audio files linked through forward-chaining. The point being that there is no similarity search in the process of recognizing an audio track.

Landrmark instead performs a quick intersection of a small list of fingerprints with a much larger list and attempts to return a correct result. Despite the elegance of the algorithm, there is no actual similarity search in this.

GridHash on the other hand considers whole audio track features in determining the Jaccard similarity between two tracks.

3.3.1 Grid Construction & Morphology

Like the Landmark algorithm, GridHash also uses the frequency over time peaks of a spectrogram to abstract an audio track. Figure 3.6 is a scatter of the peaks extracted

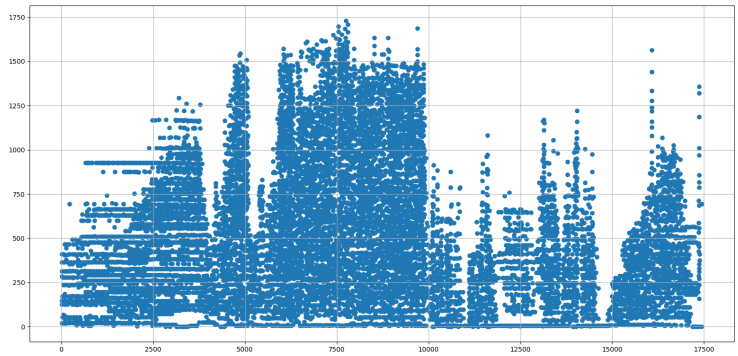

Figure 3.6: Scatter plot of frequency over time peaks

from King Crimson's 'Lark's Tongues in Aspic'.

The goal is to transform this scatter so that it has a grid structure. We do this by creating a rectangular grid over the entire scatter plot, like placing the checker pattern of a math notebook page over the scatter.

The grid scale is determined by defining the vertical and horizontal interval steps: frequency interval, respectively time interval. At each interval step a new axis is laid down for the frequency and time axes thus creating the grid pattern.

As a consequence, each peak point in the scatter plot will be within one cell of a grid, on one of the grid lines, or on the intersection of two grid axes.

Now, each cell also has a determined target area. This refers to how far a peak point can be from the frequency or time axes of the cell. Peaks that fall outside the target area are ignored, while peaks withing the target area are kept. Figure 3.7 describes such a grid.

The grid is set at a value of 2 points of the frequency and time axes. The time and frequency tolerance values are set to 1. Meaning that each cell has a target zone of 1 point from the grid line, and a remaining invalid 2^2 point area in the center of the cell. If a peak point falls in the target area, its coordinates are overwritten by the coordinates

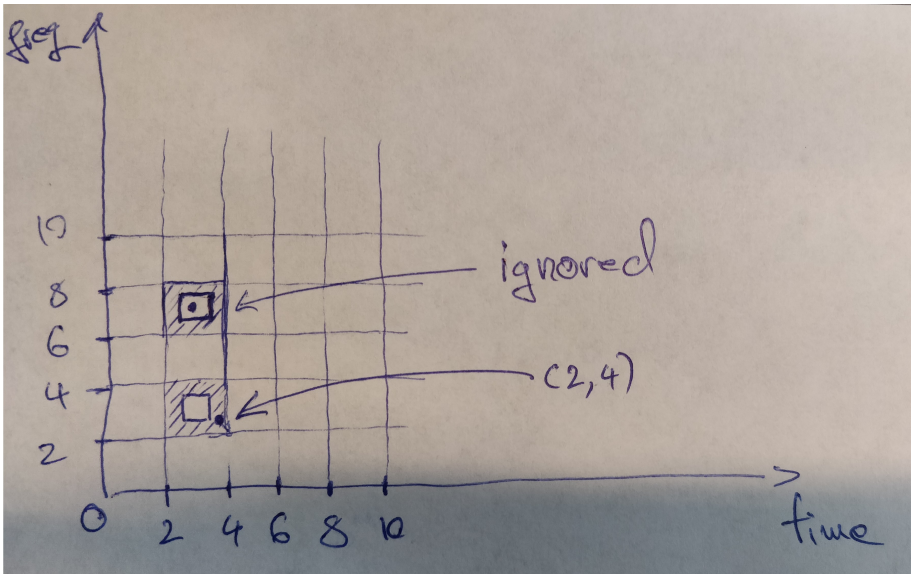

Figure 3.7: Grid structure of time frequency peaks

of the nearest corner. Thus, in Figure 3.7, the point in the lower cell is in the target zone and becomes (2,4) while the point in the upper cell is in the invalid zone and is ignored.

Let's look at a larger example based on the scatter plot in figure 3.6 at the beginning of this section. In figure 3.8, the upper grid is created over time and frequency intervals of 100 and respective tolerance values of 30. The lower grid is created with the same interval values, but significantly lower tolerance values of 10 points. Note how larger tolerance value cover more peak points while lower values provide a much sparser grid.

To further clarify, if we were to create a grid with interval values set to 100 and tolerances of 50, then all peak points would be included and ultimately overwritten with the coordinates of their respective closest cell corner. Conversely, if the tolerance values were set to zero, then no peaks would be included resulting in an empty grid.

Intuitively, the grid is meant to reduce the number of peaks and create a scatter plot with where many repeating peaks overlap on the grid intersection points.

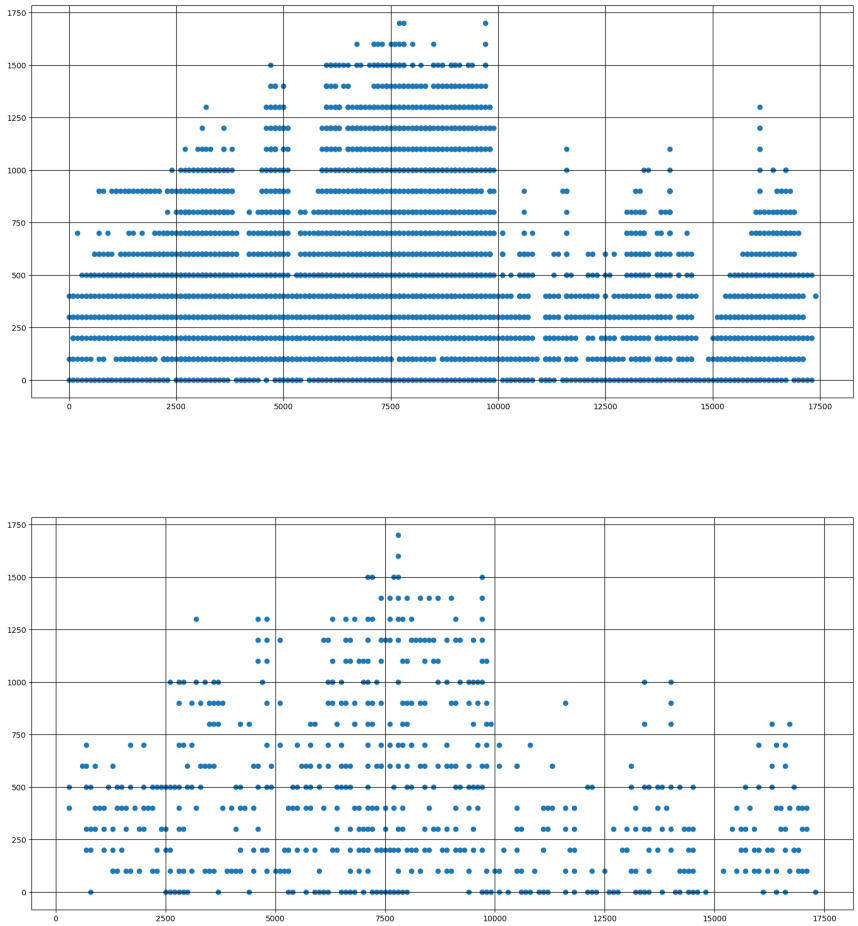

Figure 3.8: Frequency time grids at different tolerance values

3.3.2 minHashing the Grid

Once the grid for a given audio track is created each frequency and time peak is stored as a list of strings. The format of each string is **time coordinate+frequency coordinate**

The resulting list of strings will look something like this: `['03200', '03300', '03300', '03400', '03400', ..., '16007500', '17007700', '17009700', '17007800']`. The resulting list is composed of strings of numbers with the time coordinate immediately followed by the frequency coordinate of each point. Values may repeat depending on the density of peaks in the original spectrogram and depending on the interval and tolerance values of the grid.

In order to generate a minHash object of the data for later similarity searches the author has made use of the minHash package from the dataSketch [15] Python library.

Each minHashed object of an audio track is then stored to a local folder under the format **name_of_file.grid**. This is done in order to enable quick computation of Jaccard similarity over two different minHash objects.

In order to compute the Jaccard similarity coefficient of two minHashed audio tracks we extract the hash information from local storage and retrieve their resemblance coefficient as a value between 0 (completely dissimilar) and 1 (identical sets).

3.3.3 Grid Settings & Similarity Accuracy

Determining similarity between two audio files using the GridHash method has a degree of manual checking involved. This is due to the grid settings, specifically: time interval, frequency interval and time tolerance and frequency.

Experimentally, interval values of 100 points with respective tolerance levels of 30 points offered the most accurate results. The validation process involved the recording different audio tracks using a rudimentary telephone microphone and a guitar. Each track was recorded twice and the two different takes were never played back exactly

the same.

Each take of a track was manually compared to its counterpart using the GridHash method. All similar tracks had a Jaccard similarity coefficient of 0.9 or higher. While dissimilar tracks had a similarity of 0.6 and bellow.

Other grid settings have been attempted, but yielded results where previously similar tracks now had perfect similarity. This is observed for interval values of 150, 200 and 250 with tolerance values of 60, 70, respectively 80.

The settings and audio format are the main determining factors in having high accuracy. However, due to the manual process of similarity verification this segment would require further study.

Alternately, we could normalize how audio signals are processed by the algorithm. Currently, there is no downsampling of stereo audio and the sampling rate is maintained at the rate of the original recording.

TO DO: provide simple example later one

4

Evaluation & Results

This chapter is split into two core parts: one that covers the evaluation of the Larnmark algorithm's accuracy. The other describing the testing setup for the GridHash and the respective results.

4.1 Testing the Landmark Implementation

In measuring the Landmark-based algorithm implementation the author made use of Alastair Porter's excellent master thesis 'Evaluating musical fingerprinting systems' [16].

Porter proposes tracking the following metrics and ratios in order to ascertain the accuracy of the algorithm:

- **True positive** [**TP**]: the title of the queried audio track is returned
- **True negative** [**TN**]: 'No results found' is returned and the correct result is not in the database
- **False positive** [**FP**]: a result which does not coincide with the queried audio track, the correct result exists in the database
- **False negative** [**FN**]: 'No results found' is returned, the correct answer is available in the database

- **False accept**: a result is returned which does not correspond to any audio file in the database. This metric is specific to Porter's testing setup. It is tracked in the current implementation, but is not applicable to any ratio used in this project.
- **Candidate hit [HIT]**: tracks whether the correct result was present over all candidates for a given query. This metric is specific to this project.

In order to test the implementation's performance the author fed the algorithm batches of tracks of increasing size. The specific configurations and their individual results will be addressed in the upcoming subsections.

Based on Alastair Porter's work, the following ratios are used to analyze the algorithm's performance:

- **Precision**: how many positive results returned by the algorithm were correct.

$$precision = \frac{TP}{TP + FP}$$

- **recall**: how often a correct result is returned while the result is also present in the database

$$recall = \frac{TP}{TP + FP + FN}$$

- **specificity**: how often does the algorithm recognize that a result is not in the database. The equation used here differs from Porter's approach. Here we track the rate at which the algorithm identifies True Negatives, while keeping track of the number of audio tracks not inserted in the database.

$$specificity = \frac{TN}{num\_tracks\_not\_in\_db}$$

4.1.1 Evaluation Results - Precision & Recall

The general setup for all tests involves gathering a set of tracks. The test involves listening to each track for a specified amount of time, then making a prediction. Once

Table 4.1: Preliminary results over 483 audio tracks

Query time (sec)	TP	TN	FP	FN	HIT	Precision	Recall
1	315	0	168	0	321	65.21%	65.21%
2	423	0	60	0	427	87.57%	87.57%
4	467	0	16	0	468	96.68%	96.68%
8	477	0	6	0	477	98.75%	98.75%

all tracks are queried, they may be requeried for a longer time interval.

The following test configurations involve offering the algorithm only audio tracks that have been already indexed. This means that there will be no true negative results. Each track in the set is queried for 1, 2, 4 and finally 8 seconds. The results are stored at each time interval.

In order to check the algorithm's performance over different audio formats. Two databases have been built. One database with 500 wave files and another with 500 mpeg files. This is done in order to contrast the contrast in the algorithm's recall over different format audio files.

The 'out of the box' version of DejaVu always returns a result, there is not functionality which enables the algorithm to return True or False negative results.

Table 4.1 describes results under this . The a rough statelgoriinitialthm has no improvements in refining results, or weighing the most probable result candidate. All tracks tested are in wave format.

Now, let us look at a set of tests on the algorithm under its final, improved state. The improvements involve results refinement and weighing of candidate results. **Table 4.2** involves only wave format files. Additionally, all tracks used for the queries are already in the database. Consequently, this test batch does not focus on the algorithm's specificity and there will be no true negative results.

Table 4.2: Results after Landmark algorithm improvements - wave files

Tracks queried	Query time (sec)	TP	TN	FP	FN	HIT	Precision	Recall
100	1	93	0	4	3	94	95.87%	93.00%
100	2	98	0	0	2	100	100.00%	98.00%
100	4	100	0	0	0	100	100.00%	100.00%
100	8	100	0	0	0	100	100.00%	100.00%
Tracks queried	Query time (sec)	TP	TN	FP	FN	HIT	Precision	Recall
200	1	191	0	0	9	192	100.00%	95.50%
200	2	198	0	0	2	200	100.00%	99.00%
200	4	200	0	0	0	200	100.00%	100.00%
200	8	200	0	0	0	200	100.00%	100.00%
Tracks queried	Query time (sec)	TP	TN	FP	FN	HIT	Precision	Recall
360	1	346	0	0	14	348	100.00%	96.12%
360	2	356	0	0	4	358	100.00%	98.89%
360	4	358	0	0	2	358	100.00%	99.45%
360	8	359	0	0	1	359	100.00%	99.73%
Tracks queried	Query time (sec)	TP	TN	FP	FN	HIT	Precision	Recall
500	1	484	0	1	15	486	99.79%	96.80%
500	2	495	0	0	5	497	100.00%	99.00%
500	4	497	0	0	3	497	100.00%	99.40%
500	8	498	0	0	2	498	100.00%	99.60%

Table 4.3 considers a test ran under the exact same circumstances as in 4.2. However, in this situation we are looking exclusively at MEPG music files. Notice how the 1 second query time column has abysmal results compared to what we have become accustomed to in table 4.2. This is due to the files being of a different format. Wave files are uncompressed and normally much larger than MPEG format files. It is due to this that the latter category requires more query time in order to reliably return a correct result.

This phenomenon speaks to the importance of normalizing signal input to the algorithm. We will touch on this topic in detail in the next chapter.

Table 4.3: Results after Landmark algorithm improvements - MPEG music files

Tracks queried	Query time (sec)	TP	TN	FP	FN	HIT	Precision	Recall
100	1	7	0	3	90	44	70.00%	7.52%
100	2	66	0	1	30	80	98.51%	68.04%
100	4	99	0	0	1	99	100.00%	99.00%
100	8	100	0	0	0	100	100.00%	100.00%
Tracks queried	Query time (sec)	TP	TN	FP	FN	HIT	Precision	Recall
200	1	12	0	5	183	63	70.58%	6.00%
200	2	93	0	2	105	115	97.89%	46.50%
200	4	179	0	1	20	184	99.45%	89.50%
200	8	200	0	0	0	200	100.00%	100.00%
Tracks queried	Query time (sec)	TP	TN	FP	FN	HIT	Precision	Recall
360	1	21	0	15	324	121	58.34%	5.84%
360	2	179	0	7	174	212	96.23%	49.73%
360	4	317	0	1	42	327	99.68%	88.06%
360	8	358	0	0	2	358	100.00%	99.45%
Tracks queried	Query time (sec)	TP	TN	FP	FN	HIT	Precision	Recall
500	1	24	0	34	442	183	41.37%	4.80%
500	2	263	0	23	214	323	96.23%	52.60%
500	4	453	0	1	46	464	99.78%	90.60%
500	8	497	0	1	2	497	99.79%	99.40%

Table 4.4: Results for Landmark specificity - WAV music files

Tracks (indexed \ not indexed)	Query time (sec)	TP	TN	FP	FN	HIT	Precision	Recall	Specificity
40\10	1	33	10	0	7	34	100.00%	66.00%	100.00%
40\10	2	36	10	0	4	40	100.00%	72.00%	100.00%
40\10	4	39	10	0	1	40	100.00%	78.00%	100.00%
40\10	8	40	10	0	0	40	100.00%	100.00%	100.00%
Tracks (indexed \ not indexed)	Query time (sec)	TP	TN	FP	FN	HIT	Precision	Recall	Specificity
80\20	1	62	20	0	18	72	100.00%	62.00%	100.00%
80\20	2	75	20	0	5	80	100.00%	75.00%	100.00%
80\20	4	79	20	0	1	80	100.00%	79.00%	100.00%
80\20	8	80	20	0	0	80	100.00%	100.00%	100.00%

Table 4.5: Results for Landmark specificity - MPEG music files

Tracks (indexed \ not indexed)	Query time (sec)	TP	TN	FP	FN	HIT	Precision	Recall	Specificity
40\10	1	0	10	0	40	18	0.00%	0.00%	100.00%
40\10	2	2	10	0	38	30	100.00%	4.00%	100.00%
40\10	4	31	10	0	9	40	100.00%	62.00%	100.00%
40\10	8	40	10	0	0	40	100.00%	100.00%	100.00%
Tracks (indexed \ not indexed)	Query time (sec)	TP	TN	FP	FN	HIT	Precision	Recall	Specificity
80\20	1	0	20	0	80	34	0.00%	0.00%	100.00%
80\20	2	4	20	0	76	65	100.00%	16.67%	100.00%
80\20	4	62	20	0	18	80	100.00%	75.61%	100.00%
80\20	8	80	20	0	0	80	100.00%	100.00%	100.00%

4.1.2 Evaluation Results - Sensitivity

Tables 4.4 and 4.5 describe the algorithm's ability in dealing with songs that are not in the database. Up until this phase all tests have been using only audio data that has been indexed to the database. Hence, only data that the algorithm expected. This explains the complete lack of True Negative results.

In preparation for this experiment we are going to offer the algorithm incresingly large batches of songs containing 80% audio tracks that were added to the database, and 20% that are unknown. Under current parameters the algorithm excels at filtering out all False Positives results and successfully identifies files that have not been indexed to the database.

4.2 Testing GridHash

describe the method and how it is a subjective decision of similarity describe its feasibility and limitations

4.2.1 Results

describe example based results wav vs mpeg ???

5

Applications and usages

describe the use context for the development

- reference nautical vessel identification use
- describe the use shazam use in the context of ad indentification
- propose use of this on video like for youtube ripped video detection

6

Further research

the extension algorithm

- peak density and neighborhood
- converting frequency spectrum to a cepstral spectrum and the measurement of that
- cochlear modeling of peaks

7

Conclusion

References

[1] **What is Music Information Retrieval?**, 2018 [cited 2018-04-20 16:15:06]. 1

[2] Shazam Entertainment Limited. **Shazam**, 2018 [cited 2018-04-20 18:02:06]. 1

[3] Avery Li-Chun Wang. **An Industrial-Strength Audio Search Algorithm**. *Conference: ISMIR 2003, 4th International Conference on Music Information Retrieval, Baltimore, Maryland, USA, October 27-30, 2003, Proceedings*, 2003. 1, 3

[4] Roy van Rijn. **Patent infringement**, Jul 7, 2010 22:15:34 [cited 2018-04-20 18:31:06]. 2

[5] Milos Miljkovic. **Milos Miljkovic: Song Matching by Analyzing and Hashing Audio Fingerprints**, Dec 4, 2015 [cited 2018-04-20 18:39:13]. 2

[6] Dan Ellis. **Robust Landmark-Based Audio Fingerprinting**, 2009 [cited 2018-04-20 15:22:34]. 2

[7] Willie C. Venter Heinrich A. van Nieuwenhuizen and Leenta M.J. Grobler. **The Study and Implementation of Shazams Audio Fingerprinting Algorithm for Advertisement Identification**. 4

[8] Shazam Entertainment Limited. **Shazam Introduces Visual Recognition Capabilities, Opening Up A New World Of Shazamable Content**, May 28, 2015 06:08 EDT [cited 2018-04-20 16:33:12]. 4

[9] Rana Hammad Raza Muhammad Abdur Rehman Hashmi. **Landmark based Audio Fingerprinting for Naval Vessels**. *2016 International Conference on Frontiers of Information Technology*, 2017. 4

[10] Sonnis. **30GB+ OF high-quality Sound Effects FOR GAMES, FILMS & iNTERACTIVE PROJECTS**, 2018 [cited 2018-04-20 17:54:13]. 5

[11] Will Drevo. **DejaVu**, 2014 [cited 2018-04-28 18:17:01]. 5

[12] Matthias Geier Bastian Bechtold. **PySoundFile**, 2015 [cited 2018-04-30 11:57:01]. 10

[13] James Robert. **Pydub**, 2011 [cited 2018-04-30 11:58:01]. 10

[14] Will Drevo. **Audio Fingerprinting with Python and Numpy - Peak Finding**, November 15, 2013 [cited 2018-04-30 15:19:01]. 12

[15] Eric Zhu. **datasketch: Big Data Looks Small**, 21 Nov 2017 [cited 2018-05-07 17:47:23]. 25

[16] Alastair Porter. **Evaluating musical ngerprinting systems**. *A thesis submitted to McGill University in partial fulllment of the requirements for the degree of Master of Arts*, April 2013. 27

Declaration

I herewith declare that I have produced this paper without the prohibited assistance of third parties and without making use of aids other than those specified; notions taken over directly or indirectly from other sources have been identified as such. This paper has not previously been presented in identical or similar form to any other German or foreign examination board.

The thesis work was conducted from XXX to YYY under the supervision of PI at ZZZ.

CITY,