Audio Feature Extraction for Fingerprinting & Similarity Search

IT UNIVERSITY OF COPENHAGEN

Vlad Limbean

IT University of Copenhagen

Algorithms Specialization

A thesis submitted for the degree of

Master of Science in Software Development and Technology

2018

Abstract

This master thesis paper describes and documents the open-source implementation of a Landmark based audio fingerprinting algorithm. Additionally, a new method of audio similarity search is introduced. Both algorithms are evaluated and results are presented. Finally, the source code and this thesis paper are included in a public repository for the benefit of the open source community.

Acknowledgements

I would like to thank my supervisor Martin Aumüller for excellent guidance, support and professionalism. Ivan Mladenov for endearing friendship and help throughout this challenging Master's program. Gorm Galster for sage advice in some of the trickier implementation details.

My gratitude to the many friends and family who have encouraged me in my academic pursuits. In no particular order, Borbála Szabó, Maria Koosh, Lorant Tordé, Dániel Lajkó, Gergö Nefelejcs Varga, Veronika Kukovačec, thank you.

Furthermore, I would like to thank all of the unsung heroes who support the many Python libraries employed in this project. Thank you Will Drevo for DejaVu. And, of course, a big thank you for all the people supporting Latex.

I thank my mother, Tatiana Oană, for unending support and faith.

Contents

List of Figures

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 in this field range from simple endeavors like feature extraction, filtering and watermarking all the way to difficult and not-yet-solved problems like music transposition, instrument detection, cover song detection and music recommendation[1].

This thesis covers aspects of music information retrieval, with emphasis on audio fingerprinting in the context of similarity search and audio file recognition. The current section will provide insight into the author's motivation in pursuing this topic. Additionally, there will be a short description of what the reader can expect to find in the following chapters.

The problem domain and aims of the thesis are underlined in chapter 2 Research Question. An overview of the state of various audio fingerprinting methods is also offered in order to garner an intuition regarding avenues of research in the field. This is an implementation heavy thesis containing a database schema, numerous libraries and algorithms and large amounts of data. The methodology employed to implement, run and test is also described in this section.

A thorough description of the inputs, algorithms and outputs is provided in chapter 3 Algorithm Analysis. The intent of this chapter is to ensure the implementation details and choices are reasonably justified and that the project can be reproduced without too many hurdles.

The performance of the algorithm is presented in chapter 4, Evaluation and Results. Here the author details the methodology used to test the algorithms implemented and the metrics tracked in order to assess performance.

Topics of further research and area of application of the algorithm are provided in chapter 5, Discussion.

1.2 A Landmark-Based Algorithm

The primary algorithm used in the implementation of this thesis is commonly referred to as a **Landmark-based algorithm** [7], or Landmark for short. More commonly it is known as the algorithm behind popular mobile music recognition applications like Shazam[18].

To provide an initial overview of how it works, Landmark takes audio format files in as input. It processes the raw audio data into a spectrogram. On this spectrogram, the algorithm identifies local frequency maxima, and it creates a map like structure of peak points. The peak points uniquely identify an individual audio track. The algorithm finally pairs peak points together creating fingerprints and stores them for later use.

The algorithm can then apply the same method on an arbitrary number of audio tracks. It can identify an unknown audio track by cross referencing its extracted pairs of peak points by matching them to stored pairs of peak points.

The author returns to the details of how this is accomplished in chapter 3 Algorithm Analysis.

1.3 Intellectual Property Controversy

Avery Wang, founder and chief scientist at Shazam, describes a Landmark-based algorithm implementation in the 2003 paper ”An Industrial-Strength Audio Search Algorithm”[29]. Since the paper’s publication, Shazam Entertainment Limited holds numerous patents on the ”features and services provided” [17] by the company. 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 [26] and subsequent cease and desist letter due to patent infringement in 2010.

Additionally, Milos Miljkovic [22] describes similar legal quandaries five years later in his appearance at the PyData conference in New York City. He describes that any implementation of a Landmark based algorithm may fall in a legal grey area.

Intellectual property issues aside, the algorithm has seen tremendous attention from the open source community. A quick web query on ’how to implement shazam’ will yield tutorials and repositories of the algorithm in different programming languages and states of completion.

More so, the original Matlab implementation of the fingerprinting algorithm authored by Dan Ellis[7] is also available online.

1.4 Motivation

The author derives his motivation from intellectual curiosity, a need to understand the building blocks of a successful implementation and finally a desire to improve and extend existing technology.

The implementation of this thesis employs an open source library named DejaVu [30]. The author wrote numerous changes, modifications and enhancements to DejaVu in order to adapt it to emerging requirements and gain insight into its functionality.

1. INTRODUCTION

Ultimately, this project involves an implementation of the Landmark algorithm, a structured assessment on its performance, an implementation of a new non−metadata similarity search and an assessment of this features' performance and limitations. All implementation and findings are committed to a public repository for the benefit of the open source community.

2

Research Question

2.1 Research Question

The algorithm implemented for this thesis must be able to receive audio format files as input. It has two core functions: (1) process and index audio data for later use and (2) precisely recognize an arbitrary audio track by cross referencing the information of the track with already stored information.

Broadly speaking, the problem solved by the algorithm is signal recognition with high precision.

Following this premise, the following research question is raised: **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"[29]. This publication is the basis on which many Landmark-based algorithms are constructed, including the one implemented for this thesis.

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.

2.3 State Of The Art

This section briefly covers fingerprinting algorithm types. Additionally, common features used in generating fingerprints are presented in the context of current research. Due to its success, Shazam is used as a standard point of comparison to new fingerprinting algorithms. Two other audio features are prominent in fingerprinting algorithms: chroma based features (which bin audio signal samples by musical pitches) and mel frequency cepstral coefficients (which aim to model the spectral audible peaks as perceived by the human ear).

The field of audio fingerprinting has seen a lot of attention in recent years. Fingerprinting methods can be separated into three broad categories as described by Kekre et al. in "A Review of Audio Fingerprinting and Comparison of Algorithms": "Group 1: Systems that use features based on multiple sub bands, namely Philips' Robust Hash algorithm, which is reported to be very robust against distortions. Phillips Robust Hash Algorithm uses Haitsma and Kalker's algorithm. Group 2: Systems that use features based on a single band such as the spectral domain, namely Avery Wang's Shazam and Fraunhofer's Audio ID algorithms. Group 3: Systems that a use a combination of sub bands or frames, which is optimized through training, namely Microsoft's Robust Audio Recognition Engine (RARE)."[12]

Conceptually all fingerprinting systems create a database of information from audio features. The database is then used to cross reference, verify, compare or match other freshly extracted features. By doing so the fingerprinting system is able to provide valuable information regarding the queried audio.

There are numerous studies comparing fingerprinting algorithms. Nieuwenhuizen et al. notably compare Haitsma and Kalker's algorithm to Avery Wang's Shazam raising many of the issues encountered and tackled in the implementation phase of this thesis: peak detection in a spectrogram, database space overhead, and fingerprinting across various audio formats. "The following were observed in the study. Increasing and decreasing both algorithms frame size will decrease speed but increase the accuracy respectfully. Defining more peaks in the Shazam's algorithms frame (normally 5) would also result in better accuracy but decrease speed. Increasing the overlapping regions in both algorithms will increase robustness but decrease speed. (...) In the future the following should be tried. Translating the algorithms to a faster programming environment. Further research should be done on different techniques to access the database quicker. More application uses should be investigated e.g. gunshots, engine noise etc." [13]

Some fingerprinting algorithms focus on different single band features like mel frequency cepstral coefficients (MFCC) or chroma vectors. MFCCs are used to model human speech, and by extension attempts were made to use the features to distinguish music from speech as proposed in "Mel Frequency Cepstral Coefficients for Music Modeling" [5]. Further work with MFCCs is on musical instrument and timbre detection. This application determines the type of instrument playing in a musical piece with the use of an artificial neural network trained on input data: "This classifier only looks at one specific measure of a sound, the MFCCs, and yet still achieves quite accurate results. To improve the standard of this classifier even further more spectral and temporal features of the sounds need to be included. Now that we have decided on our optimum data set from MFCCs we can combine this with these other features to create a more robust classifier." [25]

Recently, Chen et al introduced GammaChirp features in "Robust audio fingerprinting based on GammaChirp frequency cepstral coefficients and chroma" [24] as a means to efficiently identify audio tracks despite audio tampering like pitch shifting and time stretching of the input audio data.

Chroma vectors are also a feature prominently used in audio fingerprinting. "The proposed algorithm is based on a modified spectrogram representation of the audio signal called the time-chroma representation. In this representation, certain attacks on audio signals such as pitch shift and tempo change. (...) The proposed algorithm is shown to outperform the well-known audio copy detection algorithm: Shazam. We also applied the famous image feature extraction algorithm (SIFT) on the proposed time-chroma representation and showed that we still get better results compared to Shazam.” [20]

Further work on fingerprinting using chroma features underlines fast and space efficient search algorithms: ”We proposed a music fingerprint approach for classical music cover song identification. The proposed music fingerprint not only outperforms the conventional state of the art cover song identification system in terms of accuracy, but also requires very low memory and computing power. It was able to reduce the search time significantly, by up to 60 times, and improve accuracy by 40% relatively.” [27]

2.4 Methodology

2.4.1 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”[28]. In addition, the author will use a personal 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 files and 500 mpeg files as testing data. Testing is performed to assess algorithm performance and indexing features.

GridHash is a newly developed extension to the algorithm written by the author. The purpose of GridHash is to compare one audio track to another and return an indication of their similarity.

This is achieved by converting a song into a list of audio features represented as strings. Consequently, this will be a list of strings that uniquely identify a song. In order to calculate the similarity of a song A relative to a song B, the author computes the Jaccard Similarity Coefficient [15]. Assume song A has been transformed into a list of features A and song B was transformed into a similar list titled B. The similarity coefficient is calculated by the formula below.

$$Similarity = \frac{\mid A \cap B \mid}{\mid A \cup B \mid}$$

This computation can be sped up by using the minHash algorithm [3] to index and quickly estimate the similarity of two sets of arbitrary sizes.

2.4.2 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[30], authored by Will Drevo. The present implementation has seen a lot of changes relative to the original library.

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**: used for data storage
- **dataSketch**: used for minHash implementation

2.4.3 DejaVu

Drevo describes DejaVu's functionality in his blog post 'Audio Fingerprinting with Python and Numpy' [32]. To summarize, the library is able to add audio tracks to a data base, it can recognize an audio track by listening to a segment of audio. The application can listen via microphone or by reading from disk. It always returns an answer, and consequently has no ability to tell the user that no results were found.

DejaVu is written in Python 2.7 and as of the moment of writing still has a few problematic bugs. The most notable one involves feature extraction: the frequency and time values of a song are stored in separate data structures; at one point in the script these two get swapped. This ultimately affects algorithm accuracy.

Furthermore, Will Drevo has documented tests on the algorithm using 45 mp3 songs. While the results he presents in his blog are promising, the algorithm requires further testing.

The implementation prepared for this thesis ports the original DejaVu to Python 3.6 and corrects the feature extraction bug. Additionally, the author has written further features and developments:

- the algorithm can now read wav, ogg, flac, vorbis and more audio formats. [11] at 8, 16, 24 and 32 bit depths. Previously, audio files at 24bit depth would be problematic on Python 3.6.
- the algorithm segregates audio format type. This thesis works with two separate databases, one with mpeg (mp3) format files, the other with wave files.
- the algorithm has been tested with batches of up to 500 songs for both audio formats. The performance of the algorithm is tested using three different metrics: precision, recall and sensitivity.
- the algorithm's ability to recognize a song has been enhanced. It can now return a 'no results found' message in case a song cannot be recognized.

3

Algorithm Analysis

This section contains a detailed analysis of the database used to store extracted audio features, the Landmark algorithm used in feature extraction and recognition, and the new GridHash algorithm used in similarity detection.

3.1 The Landmark Algorithm

Before looking into Landmark, it is important to note that the audio features are stored and indexed to local instance of a MySQL database.

3.1.1 Database Schema

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

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.

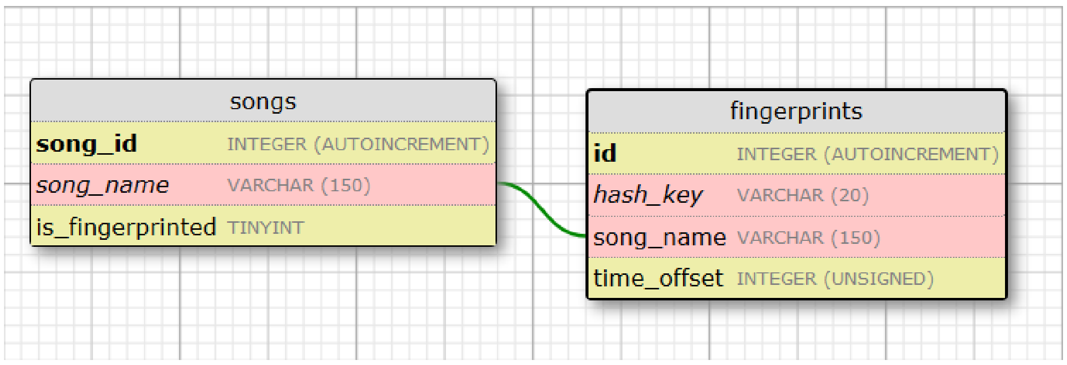


Figure 3.1: Database Schema

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 steps required to build the database are: (1) read in raw audio data. (2) transform this data from the amplitude over time domain to the frequency over time domain. (3) this results in a spectrogram of the audio file, on this spectrogram the algorithm detects the points of maximum frequency. (4) Using these maximum frequency points the algorithm constructs a list of unique fingerprints. (5) The fingerprints uniquely describe a song and are added to the database.

The first step is to read the audio format files. This is done with PyDub [16] for mp3 formats and SoundFile [4] for other formats. 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 retrieved from any audio file is raw waveform data. **Figure 3.2** represents the waveform of King Crimson's "Lark's Tongues in Aspic, Part One". The music is recorded in stereo and stored as an mp3. Consequently, the two neatly overlapping colors in the waveform graph represent the two channels of the file.

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

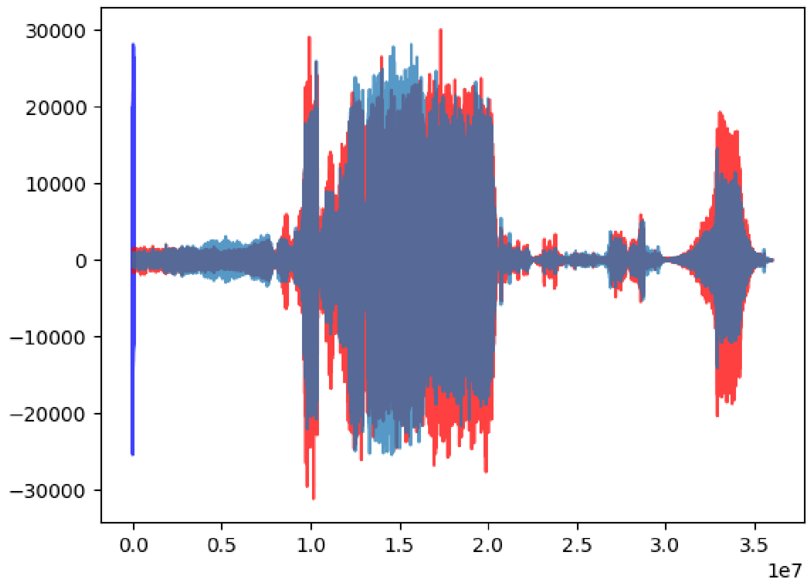


Figure 3.2: Two-channel waveform of "Lark's Tongues in Aspic, Part One"

the respective audio file. This takes the waveform from the amplitude over time domain and represents it as a function of frequency over time. For stereo recordings the transformation is performed once for each channel.

According to the Nyquist sampling theorem, "A sampled waveform contains all the information without any distortions, when the sampling rate exceeds twice the highest frequency contained by the sampled waveform" [8]. This implies that in order to capture a specific spectrum of audio, the rate at which the signal is processed must be twice the value of the frequency of the physical wave.

This matter is relevant in the transformation of the raw audio data to the frequency domain. In his book "Information Retrieval for Music and Motion", Meinard Müller describes the uses of the windowed Fourier transform. "The STFT (short time Fourier transform), which was introduced by Dennis Gabor in the year 1946, is used to represent the frequency and phase content of windowed frames of the audio signal. (...) For example, suppose the audio signal is sampled at 22050 Hz, then using a window

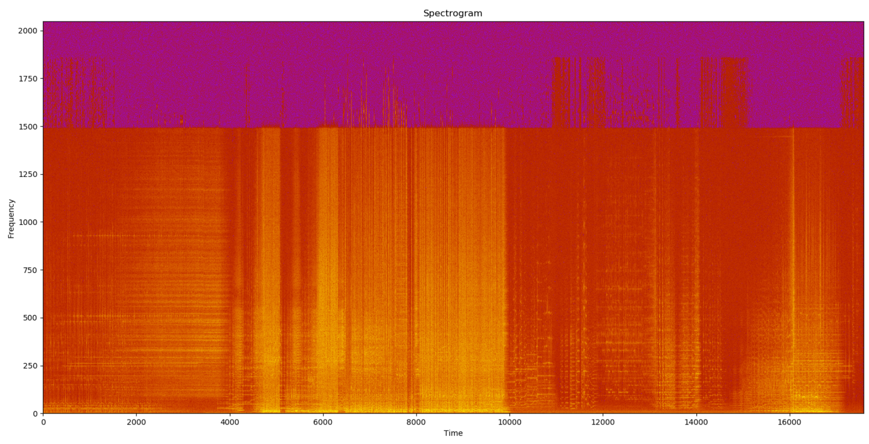


Figure 3.3: Spectrogram of "Larks Tongues in Aspic"

size of 2048 samples in the STFT will yield (approximated) Fourier coefficients for the frequencies $k * 10.77Hz$, where $k = 0, 1, ..., 1024$" [21]

In the author's implementation the Fourier transform is performed with a default frame rate of 44100 Hz, using a fixed window of 4096 points. This implementation detail is unchanged from the original version of DejaVu. It is meant to ensure a transformation to the frequency domain which captures the entire frequency spectrum perceivable by the human ear.

The only variable element in this situation is the sampling frequency. Most audio tracks are sampled at 44100 Hz, hence this is set as the default sampling rate. However, if audio files are sampled at higher values like 48000 or 96000 Hz, then the higher value is used in the transformation. An audio track's sampling frequency can be obtained from the file's audio encoding.

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

3.1.2.1 Identifying Time-Frequency Peaks

This section describes how local maxima are recognized on the spectrogram. The frequency over time domain can be interpreted like an image. Local maxima can be detected on an image with the use of the SciPy image processing library [32].

With the spectrogram generated, the following step is to discover points of maximum frequency of the spectrum. These points will be referred to as peaks, and they can be understood as the frequency and time coordinates points in the two dimensional plane of the spectrogram.

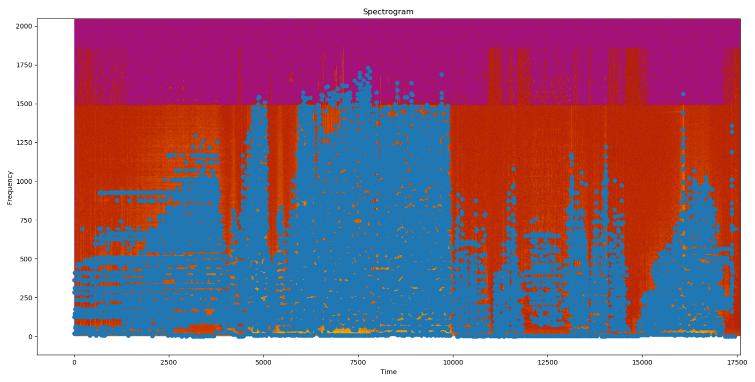
A scatter plot of the peaks would look like **figure 3.4**. The x-axis represents the time domain, the y-axis is the frequency domain. The blue dots are detected peaks.

Peaks are determined by considering a neighborhood area around a point of maximum frequency. The neighborhood is represented by the "number of cells around an amplitude peak in the spectrogram in order for Dejavu to consider it a spectral peak. Higher values mean fewer fingerprints and faster matching, but can potentially affect accuracy." [31] The number of cells is in this case set to 20. **Figure 3.4** describes the difference in number of generated peaks at different neighborhood values. The top image has a peak neighborhood value of 20 points, yielding 11658 peaks. The bottom image has a peak neighborhood of value of 40 points, yielding 3377 peaks.

3.1.2.2 Hashing Time Frequency Peaks

This section focuses on using the detected peaks to generate a list of fingerprints to uniquely represent a given audio track.

This is how Avery Wang describes the generation of fingerprints. "Fingerprint hashes are formed from the constellation map, in which pairs of time-frequency points are combinatorially associated. Anchor points are chosen, each anchor point having a target zone associated with it. Each anchor point is sequentially paired with points within its target zone, each pair yielding two frequency components plus the time difference between the points (...). These hashes are quite reproducible, even in the presence



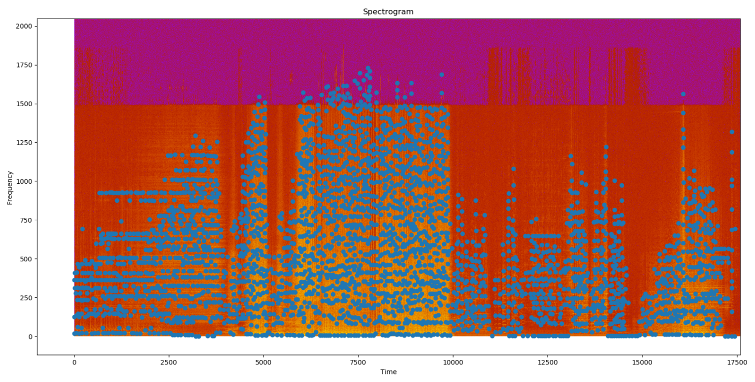


Figure 3.4: Spectrograms with different peak densities

of noise and voice codec compression.” [29]

A constellation map in the present implementation is equivalent to the scatter of detected peaks over the spectrogram. An anchor point refers to one designated peak point relative to which other peaks are paired. A target zone refers to a neighborhood of peaks. Consider the anchor point as the root of a tree with depth 1 where each leaf node is one of the peaks available in the target zone.

Avery Wang does not detail the selection of anchor points, or how to determine target zones. The approach in the author’s implementation and in the original DejaVu implementation is as follows: each peak on the spectrogram is considered an anchor point. Each anchor point is paired with a set number of other peaks, the present implementation considers maximum 15 points. This number is referred to as the **fan value** of the anchor point. Before pairing the anchor with an associated peak, the algorithm checks whether the point is between 0 and 200 sample points on the time axis. This check constitutes the target zone. Consequently, a pair is formed only if the associated peak point happens simultaneously or in the future (by 200 samples on the time axis) relative to the anchor point.

A fingerprint is constructed by taking the value of an anchor point **f1** and an adjacent peak within the fan value range **f2** if the peak is within the target zone. Both peaks have associated time indices **t1**, respectively **t2**.

$$anchor\_point = (\mathbf{f1}, \mathbf{t1})$$

$$associated\_peak = (\mathbf{f2}, \mathbf{t2})$$

In order to check if the associated peak is within the target zone, the following time difference is computed: $\Delta t = t2 - t1$.

Each created pair becomes a fingerprint. The following details are contained **(f1, f2,** Δt**)**. The details are then converted to a string and passed through the Secure Hash Algorithm 1 or SHA1 to produces a string of 40 characters. The fingerprint itself is the hash digest of SHA1. In order to reduce space complexity, the original DejaVu implementation drops the last 20 bits of the digest. This implementation detail is also present in the author's implementation.

```
Example hash digest:  89b64c1a491f804aa34b
```

Each fingerprint constitutes a row in the **fingerprints** table of the database. Every row has the following structure: **id** of the row, the **hash digest**, the **name of audio file**, the time index value **t1** of the anchor point.

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

Once all fingerprints of an audio track are added to the database the file is known by the algorithm and can be recognized.

3.2 Audio File Recognition & Search

This section describes how the algorithm can recognize an audio track by listening to a few seconds of a signal. This implies the following steps: (1) fingerprint matching, where all songs that match are returned. (2) Result refinement, where all weak candidate results are removed, and finally. (3) Result weighing, where the algorithm determines if the result is correct or, alternately it cannot detect a correct result.

3.2.1 Matching Audio Files

The algorithm can listen to any audio format file. The signal is read from disk. The algorithm only requires a few seconds sample from any audio track in order to recognize it. The audio snippet is processed just like entire audio tracks during the indexing process and it generates a list of fingerprints. However, instead of inserting fingerprints to a database, in the recognition phase the algorithm asks the database for all the audio track titles that match the fingerprints generated.

The algorithm will take the list of produced fingerprints from the audio snippet and cross reference it with all 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

There is no difference between how the algorithm listens to a full track or a snippet of the track. Peak detection is performed exactly the same in both cases using a peak neighborhood (of 20 points around a peak point). The peak neighborhood does not scale to the size of the track.

Consequently, the larger the peak neighborhood is, the fewer peaks are detected. If the spectrogram is smaller in size (because the algorithm is only listening to a section of an audio track), then the algorithm must identify fewer peaks relative to the size of the full track.

Figure 3.5 describes this phenomenon. The audio track used to generate this graph is a 43 seconds long recording of an acoustic guitar musical piece. The entire song generates 15548 fingerprints. The blue curve represents the rate at which the algorithm identifies peaks over the length of the song. On a second by second basis the growth of the number of generated fingerprints is roughly linear. However, if the algorithm listens to incrementally larger slices of the song it will always detect fewer matching fingerprints.

Notice that for smaller snippets of the audio track, the number of 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 fingerprints of the full audio track.

Consequently, peak detection is directly responsible for the accuracy of the algorithm. A point of future research is to determine a peak detection method which scales

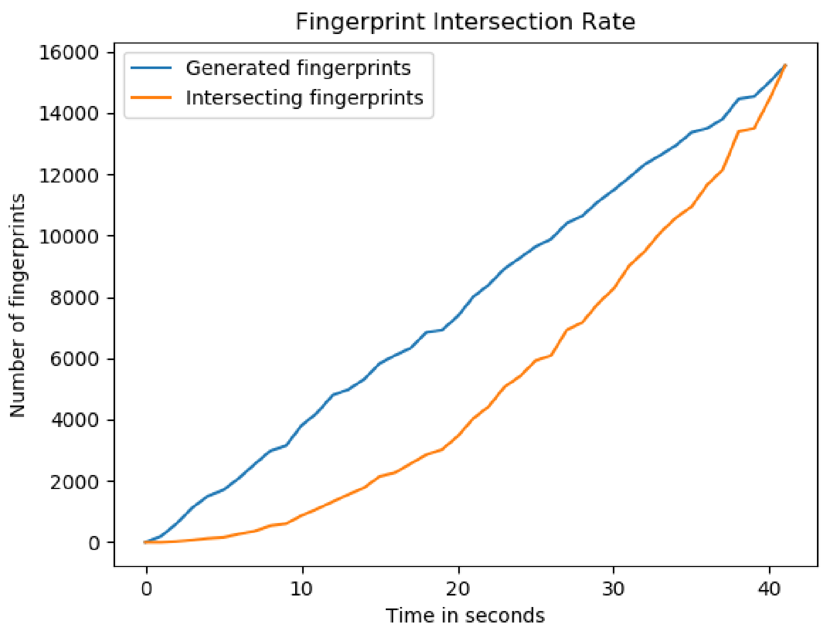


Figure 3.5: Progressive Fingerprint Matching Rate

to the size of the audio track provided to the algorithm. This matter will also be addressed in the Discussion chapter.

3.2.3 Refining the Matched Results

The algorithm returns a list of all audio tracks that match the fingerprints of a snippet. Depending on the size of the database this list can grow very large. This section describes how the list is reduced with the purpose of returning an accurate result.

Recall that a fingerprint in the database contains an id, a hash-key representing the fingerprint, a track title, and a time index **t1**. The time index serves as an indicator of where in the associated audio track the fingerprint occurred.

This serves in determining whether a matching fingerprint actually occurs in the same audio track or not. For example, track **example.mp3** has been added to the database. The algorithm, then receives a three second snippet of the song and returns one matching fingerprint. The matching result provided contains the name of the track from the database, and the time index of the fingerprint from the database minus the time index of the fingerprint from the three second sample.

$$Matching\_result = (track\_title, t1 - t\_sample)$$

Results with time index 0 are an exact match. The farther away from index 0, the higher the probability that the fingerprint is not an exact match. Another possibility is that the matched fingerprint is from another audio track..

SFX Large Wave Splash on Rocks 21.wav is the basis for the following example. This is a stereo 8.63 second wave file. It contains the sound of water splashing on rocks. The algorithm is provided with a two second snippet of this track.

Table 3.1 contains a section of the returned results. The table has a time index. For each time index there is a list of matching track titles.

The interpretation of the table 3.1: There are three audio tracks whose fingerprints perfectly match the queried song and other candidates results that occur at different time indices. The track 'SFX Large Wave Splash on Rocks 21.wav' has 1620 matching fingerprints. There are other songs with matching fingerprints at time index 935, 1501, 850 and so on. However, all these candidate results have one or two matching fingerprints, and may likely be from another song. The correct result then is 'SFX Large Wave Splash on Rocks 21.wav'.

3.2.4 Weighing Function & Cut-Off

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 and corresponding fingerprints. The goal is to eliminate imperfect matches and retain the most likely correct result.

By considering table 3.1, one could assume that the correct answer will always have the highest number of matches and exclusively be at index 0. Correct answers also

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

occur at other time indices.

Running the algorithm in its original state without any weighing function will provide less accurate results. Table 4.1 describes algorithm results without any improvements. Table 4.2 describes algorithm results with all implemented enhancements.

The weighing function is one of the core improvements and it offers greater value to indices closer to 0. At the time index 0 the weight is 1000. As the time index slides in either direction from 0 the weight value of the candidate result steeply declines.

Through experimentation, the author noticed that correct song results will contain at least around a few hundred matches. While any incorrect matches will possibly have less than 10 matches. Correct results do not exclusively occur at time index 0. To offset this aspect, the weight of a set of matches is compounded by the maximum frequency within the set. This is described in the formula below.

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

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 for each time index a list of tuples is generated. Each tuple contains: **(weight, time index, candidates)**. The list is sorted by weight. **Table 3.2** describes a small segment of the refined and weighted candidates.

The weight of a set of candidates is used as a threshold value. A correct match will most frequently find itself at time index 0, hence having a weight of 1000 plus the frequency of the candidate with most matches. Reiterating, a correct result will have at least a few hundred matches. Consequently, to guard against mismatches the author uses the weight of **1010 as a cutoff value**. If a candidate is above this value, it will be regarded as a highly probably correct result, otherwise the candidate is discarded and the algorithm returns 'No results found'.

This process resulted in the algorithm's measurably excellent ability to avoid mismatches, or false positive responses. More on this matter in the Evaluation and Results chapter.

3.3 GridHash

The Landmark algorithm can recognize audio tracks by listening to a section of a known track. However, it cannot perform any search based on a metric of similarity. This section describes the development and functioning of the GridHash algorithm. GridHash determines the degree of similarity between two audio tracks.

The goal of GridHash is to consider a primary audio track and return a list of similar tracks. It is a similarity search algorithm that does not use any metadata in producing a result.

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

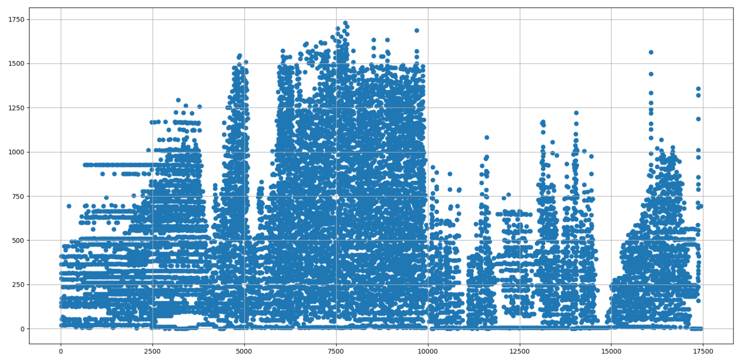


Figure 3.6: Scatter plot of frequency over time peaks

3.3.1 Grid Construction & Morphology

Like the Landmark algorithm, GridHash also uses the frequency time peaks of a spectrogram to abstract an audio track. **Figure 3.6** is a scatter of the peaks extracted from King Crimson's 'Lark's Tongues in Aspic'.

The aim is to transform the scatter of peaks into many overlapping points on a grid structure. A rectangular grid is laid over the scatter plot, like placing the square pattern of a math notebook page over the scatter.

The scale of the grid is determined by defining the vertical and horizontal interval steps. Specifically this determines how large the rectangular (or square) cells will be relative to the size of the spectrogram. Frequency and time interval values are defined for this purpose. At each interval step a new axis is laid down for the frequency and time axes thus creating the grid pattern.

Each peak point in the scatter will be within one cell of a grid, on one of the grid axes or on the intersection of two grid axes.

Furthermore, each cell also has a determined target area determined by time and frequency tolerance values. This defines where in the cell a peak can be situated in order for the it to be hashed to the grid. Peaks that fall outside a specified target area

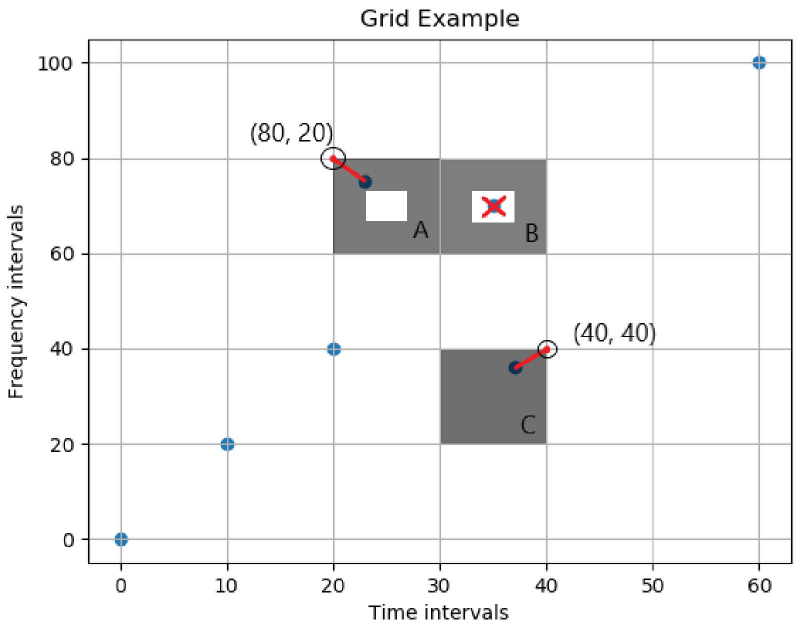


Figure 3.7: Grid structure

are ignored, while peaks within the target area are kept.

Figure 3.7 offers a detailed example of grid construction. The time interval is set to 10 points on the time axis, so every ten points there will be a vertical line on the grid. Similarly, the frequency interval is set to 20 points on the frequency axis, hence a horizontal grid line every 20 points. When building the grid of any audio track the interval and tolerance values are set once. For the sake of example, figure 3.7 has different tolerance values on the same grid. Specifically, in cell A the peak is at coordinate (75, 23). The tolerance values determine how far from a respective axis a peak can be placed and be considered to be in the target area. The time tolerance value is set to 3 and frequency tolerance is 6, resulting in a 32 square point white area in the center of the cell, the invalid area. The surrounding grey area of 168 square points represents the target zone. When a point falls in the target area its coordinates become that of the nearest grid intersection point. For cell A, peak (75, 23) hashes to (80, 20). If a peak falls outside the target zone, like in cell B, the point is disregarded. Additionally, the tolerance values can be set to cover the entire cell in which case any peak inside the cell will hash to its nearest corner. This occurs in cell C, where peak (37, 36) hashes to point (40, 40).

The intent behind tolerance values is to reduce the number of peaks for the purpose of tuning the degree of similarity. The author noticed through experimentation that at certain grid interval and tolerance values two marginally similar tracks can be considered identical. This matter will be discussed further in following chapters.

Based on figure 3.6 at the beginning of this section, in **figure 3.8**, the upper grid is created at 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. By modifying the tolerance values, the grid can be made sparse or dense.

Finding the optimal grid interval and tolerance values is an exploratory matter of importance when it comes to algorithm accuracy. This aspect will be covered in depth in the upcoming Evaluation and Results chapter.

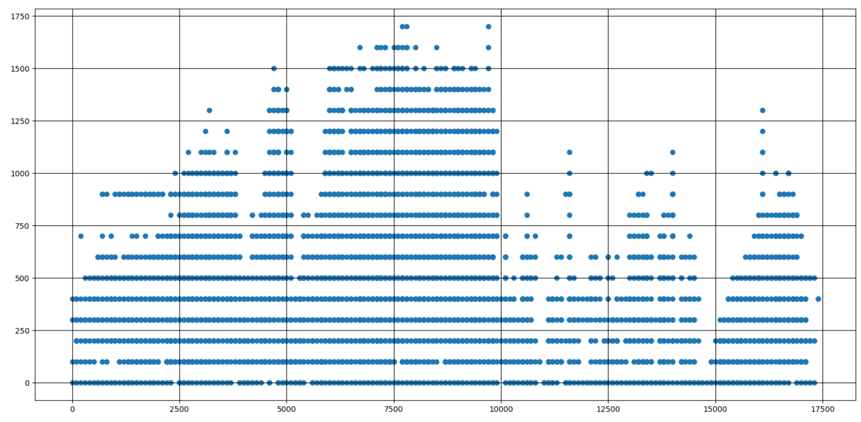
3.3.2 minHashing the Grid

The final step of the GridHash algorithm is to take the coordinates of each point on the grid and store them in a list. The aim is to have a list to represent one audio track. Consequently, by calculating the resemblance of two different lists one can obtain the degree of similarity between one track and another.

Each coordinate in a list is stored as a string with the following format:

time coordinate + "   " + **frequency coordinate**

The resulting list of coordinates will resemble this list: `['0 0', '0 75', '0 150', '0 75', '75 75', '75 0', '75 150', '75 75', '75 75', '150 150', '150 0', '150 75', '150 150', '150 75', '150 0', '150 75', '150 0', '150 75', '150 75', '150 150', (...)]`. The resulting list is composed of strings of numbers. Each string contains the time coordinate followed by an empty space and the frequency coordinate. Values may reoccur depending on the density of peaks in the original spectrogram and



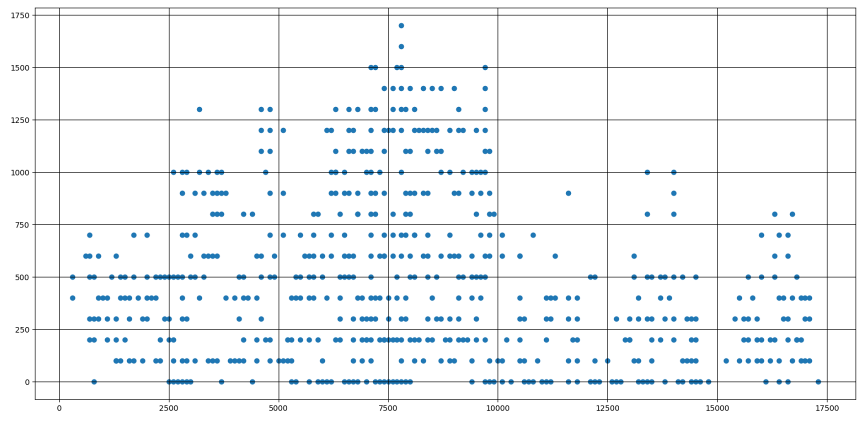


Figure 3.8: Frequency time grids at different tolerance values

on the interval and tolerance values of the grid.

In order to establish the resemblance of one coordinate list to another, the author employs the Jaccard similarity coefficient [15]. A Jaccard coefficient of 0 represents completely dissimilar sets, a value of 1 represents identical sets. The coefficient is computed by taking the cardinality of the intersection of two sets, divided by the cardinality of the union of the same sets. "Experiments seem to indicate that high resemblance (that is, close to 1) captures well the informal notion of near duplicate or roughly the same" [3].

The author uses minHash to "estimate the Jaccard similarity (resemblance) between sets of arbitrary sizes in linear time using a small and fixed memory space." [9]. Each audio track gets processed by the grid and indexed via minHash. The minHash objects generated can be stored locally for later use. Stored objects are contained in a folder and have a simple naming convention **name_of_audiofile.grid**.

Ultimately, the GridHash objects can be used to identify similar audio tracks from a large database of files. One audio track can be used as the basis for comparison. This base audio track can be a result produced by the Landmark algorithm.

3.3.3 Challenges of GridHash

Determining similarity between two audio files using the GridHash method implies a high degree of manual verification. Take for example two distinct recordings of the same musical tune played on an instrument. Due to the physical process of performing the tune the two recordings will be almost identical, but not quite.

Two tracks like this can be used to establish the optimal grid interval and tolerance values for the GridHash algorithm. The goal of the experiment is to find grid settings which yield GridHashes of two similar songs that yield high similarity. When calculating the Jaccard coefficient of the two, the result should be about 0.9 or above (thus the two being near-duplicates).

Following this method, the challenge is to create a set of pairs of sound tracks. Each pair consists of two tracks that are roughly similar to each other, but not similar to tracks from other pairs.

This method has further concerns like, does this method apply to more complex sound tracks, like mastered pop songs? Is the method audio format agnostic, or do separate formats need different grid settings?

Additionally, using minHash on a multiset of time and frequency of coordinates is a valid approach. However, since the grids are designed to contain many overlapping points (many reoccurring elements in the set), it may be valuable to explore creating gridHash object using a weighted minHash algorithm. This would account for the number of duplicate elements in the set.

These concerns are addressed further in the Discussion chapter.

4

Evaluation & Results

This chapter is split into two parts. The first is concerned with describing the testing methodology for the Landmark algorithm and presenting the results. The second describes the evaluation methodology of the GridHash algorithm, the results and further considerations regarding the algorithm.

4.1 Testing the Landmark Implementation

In measuring the Landmark-based algorithm implementation the author made use of Alastair Porter's testing methodology as presented in his master thesis 'Evaluating musical fingerprinting systems' [2].

The testing setup up involves checking the Landmark algorithm's performance in correctly identifying audio files from a large set of test tracks. The following metrics are tracked:

- **True positive [TP]**: the title of the queried audio track is returned
- **True negative [TN]**: 'No results found' is returned but the correct result is not in the database
- **False positive [FP]**: a result which does not coincide with the queried audio track is returned, 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 [FA]**: a result is returned but it does not correspond to the query track, but the query track is not in the database. This metric refers to situations when the algorithm mistakes a track that has not been indexed to the database to another song available in the database.

- **Candidate hit [HIT]**: tracks whether the correct result was present in the list of candidate results. This metric is specific to this thesis.

The implementation's performance was tested on increasingly larger batches of audio files. Each test keeps track of the measurements below.

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

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

- **Recall**: how often is a correct result returned and the result is also present in the database

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

- **Specificity**: how often the algorithm correctly recognizes that a result is not in the database. The equation used by Porter tracks the number of True Negative results divided by the sum of True Negative plus False Accept results. However, the author's implementation does not generate any False Accepts. Consequently, the metric tracks the rate at which the algorithm can determine that an audio track is not in the database.

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

4.1.1 Evaluation Results - Precision & Recall

The setup for all tests involves creating a set of tracks. The algorithm listens to each track for a specified amount of time, then makes a prediction. Once all tracks are queried, the test is run again, only this time each track is queried for a longer amount

Table 4.1: Preliminary results over 483 WAVE files - No algorithm improvements

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%

of time.

In the upcoming results the algorithm is only tested for precision and recall. Specificity testing is performed in the upcoming section. Each track in the set is queried for 1, 2, 4 and finally 8 seconds intervals.

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 verify the algorithm's precision and recall over two different audio formats.

Table 4.1 describes results of the original DejaVu implementation with no improvements to the result refinement or result weighing. Note the large number of false positive results generated. The algorithm cannot determine whether a track is a mismatch or not.

To contrast this, **table 4.2** contains test results over wave format files with all algorithm improvements described in the analysis. This includes result refinement, candidate weighing and cutoff. Besides the notable increase in precision and recall, it is worth underlining the strong decline of false positive results and the presence of false negatives. After improvements, the algorithm is significantly more precise, has very few mismatches and is prone to returning 'No results found' in situations where there is insufficient information to make a correct decision.

Table 4.3 considers tests run under the same setup as above (in table 4.2). However, in this situation the files are in mpeg format.

Notice how the 1 second query time column has low recall compared to wave format results. This is due to the file compression. Wave files are uncompressed and take up significantly larger memory space, while mpeg files are compressed in order to ensure better space complexity. This process however affects the Landmark algorithm's ability to correctly recognize the file.

The author's implementation does not normalize audio information. Fingerprinting is performed with no downsampling or reduction to mono (in case a file is recorded in stereo). For large scale applications normalization is a necessary feature in order to ensure scalability. Despite the author's implementation not being set up for large scale testing, audio format normalization would result in less of a contrast in the test results between different audio formats. The matter of audio format normalization will be addressed further in the Discussion chapter.

4.1.2 Evaluation Results - Sensitivity

This section continues the evaluation of the Landmark algorithm. Until this phase audio files used in algorithm testing were also available in the database. The goal in this section is to test the algorithm's ability to decide whether it recognizes any audio track. This extends to audio files that are not in the database.

The set of audio files prepared for testing contains 80% tracks that are in the database and 20% tracks that have not been indexed. The preferred outcome is for the algorithm to always correctly identify when a file cannot be recognized. Meaning that there should be an increase in True and False Negative results.

Tables 4.4 and 4.5 are the results for testing wave formats and mpeg format files, respectively. The algorithm excels at identifying all of the audio files that have never been added to the database. Additionally, there are no false positive results. The algorithm does not mistake one song for another. The results are either correct or 'No

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: Results after Landmark algorithm improvements - MPEG 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 - WAVE 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 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%

results found', meaning the algorithm cannot determine a correct result.

The algorithm's specificity performance is due to the candidate weighing function and cutoff value. Recall that for a result to be considered exact it must be a perfect match to the database entry (meaning it is at time index 0), thus giving it a weight of 1000. The weight value is further compounded by the number of matches of the primary candidate result. A correct result will have at least a few hundred matches, thus placing the real weight in the range of 1100+. Finally, the author's implementation has a cutoff value of 1010. This means that in order for a result to be correct it must be an exact match on at least 10 fingerprints or an inexact match with a few hundred matching fingerprints.

4.2 GridHash Evaluation

The primary challenge behind the GridHash evaluation is gathering appropriate audio test tracks. The goal is to create a list of pairs for testing. Each pair consists of two similar audio tracks. The expected behaviour of GridHash is to detect tracks in a pair as being similar to each other and dissimilar to other tracks. Like the Landmark algorithm evaluation, tests are going to be run using two types of audio files: one composed of wave files (simple soundtracks), the other composed of mpeg (popular songs).

The grid interval and tolerance values used in generating the gridHash objects were determined experimentally. The author took gradually increasing intervals of 50, 75, 100 and 150 points on the time and frequency axes and observed the similarity measurement between the prepared test files.

Tolerance values were selected such that both dense and sparse grids would be checked. For example, grids with time and frequency intervals of 50 and time and frequency tolerance values of 10 will exclude many peak points from any given spectrogram. Interval values of 50 and tolerance values of 25 however will cover the entire grid and will exclude no points, thus having full coverage.

The interval and tolerance values for both time and frequency axes have been kept equal. There is no testing performed on rectangular grids. The following grid settings were tested.

- Lower values offer poor results
- Interval 50, Tolerance: 10
- Interval 50, Tolerance: 50
- Interval 75, Tolerance: 20
- Interval 75, Tolerance: 35 (preferred for mpeg)
- Interval 100, Tolerance: 30 (preferred for wav)
- Interval 100, Tolerance: 50
- Interval 150, Tolerance: 50
- Interval 150, Tolerance: 75
- Higher values return similarity 1 for disimilar tracks

Grid interval values under 50 offer incorrect results where nearly identical songs have a Jaccard similarity coefficient below 0.5. Conversely, at interval values over 150, similar but not identical tracks have similarity coefficients of 1. After testing and observing, the preferred grid settings for mpeg files are at interval values of 75 and tolerance values of 35. For wave files, the most representative similarity results are

Table 4.6: WAVE GridHash Similarity Results

	Wave 1	Cover 1	E string 1	Thunder 1	Rain on Plastic	River 1	Splash 1
Wave 2	0.90625	0.1640625	0.0078125	0.90625	0.6328125	0.015625	0.203125
Cover 2	0.2109375	0.84375	0.109375	0.171875	0.28125	0.1015625	0.5625
E string 2	0.0078125	0.03125	0.234375	0.0	0.0078125	0.203125	0.0234375
Thunder 2	0.5859375	0.109375	0.0078125	0.6953125	0.3984375	0.015625	0.15625
Rain Hard	0.59375	0.2890625	0.0234375	0.4765625	0.8359375	0.015625	0.3671875
River 2	0.015625	0.09375	0.3828125	0.0078125	0.015625	0.8984375	0.09375
Splash 2	0.25	0.484375	0.078125	0.1875	0.304687	0.0859375	0.9609375

found under interval values of 100 and tolerance values of 30.

There are further arguments regarding the selection of grid interval and tolerance values. However, for the current evaluation, the author has chosen to present the most promising results in the section below. Further considerations will be addressed in the Discussion chapter.

4.2.1 Evaluation results - Wave format files

The wave files used for testing are a mix of stereo recordings selected from a large set of wave files. Additionally, short acoustic guitar recordings created by the author are used. Each audio track pair follows this naming convention: **title 1, title 2**.

Table 4.6 describes the degree of similarity between each pair element relative to its counterpart. The horizontal top row of the matrix contains the titles of audio files. All corresponding counterpart audio files are placed on the left most column. Consequently, the similarity coefficient of each matching track can be found down the matrix diagonal.

In the given test there is one notable inaccurate result for a pair of tracks. Specifically, the E string 1 and 2 files, which represent six notes played in close succession on an acoustic guitar. The only difference between them is the tempo.

Figure 4.1 describes the overlapped waveforms of the "E string" tracks. The amplitude spikes on the figure represent the onset of each of the six notes played during the recording. Notice how the onsets of each note played are increasingly out of synch.

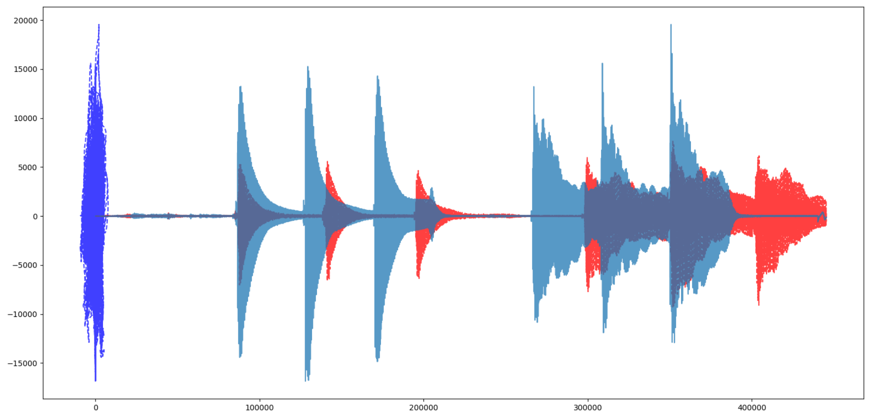


Figure 4.1: E String 1 & 2 Tempo Difference

This ultimately leads to a low similarity coefficient. The author also noticed that by increasing the interval and tolerance values the similarity coefficient can be increased. This phenomenon makes sense due to the fact that any increase in intervals and tolerance levels of the grid will generate more common points between two audio tracks.

Overall similarity results over wave files are promising. The use of this algorithm takes a primary audio track's gridHash and checks its similarity against secondary griHashed tracks. Ideally the functionality can work in tandem with the landmark algorithm, which would return a result, then using the gridHash of the respective result one could quickly find other similarly sounding audio files by comparing against a set of gridHash files.

4.2.2 Evaluation results - Mpeg format files

Tests for mpeg format files are done under the same setup as for wave format files. The author selected ten pairs of songs from a personal collection. The songs are sourced from various artists: "And I love her" (Beatles cover) by Bob Marley & the Wailers, "And I love her" and "Yellow submarine" by The Beatles, Pink Floyd's "Another Brick in the Wall", Gary Moore's "Friday on my mind", "Wild Frontier", "Over the hills and far away", and "The loner", "Rain Song" by Led Zeppelin, "Voodoo Child" by Jimmy

Table 4.7: MPEG GridHash Similarity Results

	Love her	Brick wall	Friday	Over hills	Rain Song	Loner	Voodoo	Frontier	Wild Horses	Submarine
Love her 2	0.43	0.25	0.34	0.25	0.2734375	0.19	0.24	0.07	0.30	0.53
Brick Wall 2	0.52	0.70	0.80	0.64	0.40	0.54	0.22	0.85	0.56	0.48
Friday 2	0.28	0.77	0.63	0.78	0.68	0.89	0.38	0.61	0.77	0.26
Over Hills 2	0.19	0.57	0.44	0.58	0.79	0.63	0.37	0.42	0.60	0.21
Rain Song 2	0.12	0.36	0.26	0.37	0.60	0.44	0.29	0.25	0.42	0.13
Loner 2	0.26	0.67	0.51	0.67	0.76	0.75	0.42	0.54	0.70	0.25
Voodoo 2	0.35	0.83	0.69	0.85	0.61	0.76	0.31	0.65	0.80	0.375
Frontier 2	0.25	0.71	0.56	0.69	0.70	0.79	0.40	0.57	0.68	0.23
Wild Horses 2	0.27	0.66	0.54	0.68	0.57	0.63	0.25	0.50	0.69	0.31
Submarine 2	0.86	0.35	0.42	0.34	0.28	0.28	0.01	0.41	0.31	0.91

Hendrix and finally "Wild Horses" by the Rolling Stones.

The results for mpeg songs are tenuous. There is not clear discernible similarity established between elements in a pair with the exception of "Yellow Submarine". Additionally, there are cases of high similarity between songs that are of different musical composition and length.

It is important to note however, two songs in a pair are not identical. They are different remastered versions, cover songs, bootleg versions of their counterpart. Consequently, GridHash is at this phase inappropriate for the purpose of cover song detection. Additionally, for mpeg files GridHash results are inconclusive and require further experimentation.

Further research on grid settings for mpeg files is required in order to determine the feasibility of gridHash for the mpeg audio format. This research avenue implies a review of the mpeg compression effects on the spectrogram of an audio file.

5

Discussion

The Evaluation and Results presented in the previous chapter describe the Landmark-based implementation as a precise and robust algorithm for audio track recognition. Additionally, the gridHash algorithm is introduced as a method to quickly determine the similarity between two indexed audio files.

This chapter emphasizes the interesting research avenues relating to the presented results. In addition, this section considers alternative implementation approaches, algorithm and database scalability problems and audio data normalization.

5.1 Further Research

5.1.1 Input Data Normalization

In "A Industrial Strength Audio Search Algorithm" Avery Wang describes audio files being normalized in the following manner: "audio sampling and processing was carried out using 8KHz, mono, 16-bit samples." [29] The implementation in this thesis performs no downsampling, no conversion to mono and raw audio data is packed into 16-depth bits. The drawback of this approach implies high space complexity.

Having high space complexity leads to having a hefty database. The current implementation consists of two separate databases, one for wave format files, another for wave files. The wave files database consists of 500 tracks and 54.8 million fingerprints taking up 5.7 Gb, plus a 6.2 Gb index. The mpeg file database also consists of 500 tracks and 99.1 million fingerprints with the cost of 7.3 Gb for the data and 8.5 Gb for the index. Ultimately this is 25 Gb for two tables. Consequently, it would be very interesting to measure how this cost can be reduced via audio data normalization.

5.1.2 Peak Detection

As described in **figure 3.4** the current implementation uses the SciPy image morphology and filter package to detect peaks on a given spectrogram. The spectrogram gets interpreted as a 2D image. The algorithm detects peaks by using a maximum filter with a neighborhood area of 20 cells around any given frequency peak.

The issue in this instance is that the peak neighborhood value does not scale to the size of the spectrogram. If the spectrogram grows or shrinks the peak neighborhood stays the same, thus the number of detected peaks will noticeably vary.

An important point of future research is to experiment with peak detection algorithms which adapt to the size of the input 2D space. The author has performed summary checks of available peak detection algorithms all of which use some sort of neighborhood function. Software packages like **scipy.signal.find_peaks, peakdetect or OctaveForge findpeaks** [33] perform signal peak detection for a simple signal. However, the current problem requires peak detection in two dimensional space, the solution discovered for 2D peak detection implies the use of a neighborhood function.

Take **figure 3.5** as reference: an improvement in the peak finding algorithm would result in a more stable way to reproduce peaks on a spectrogram, or a section of a spectrogram. Visually, this would have the yellow line of the graph (which represents matching fingerprints) get closer to the blue line (representing all fingerprints generated for a specific track), thus having a more robust way to detect matches.

5.1.3 Fingerprint Creation & Anchor Point Selection

Figure 5.1 describes the number of fingerprints generated over the number of audio tracks added to the database. Notice how the number of generated fingerprints strongly differs for the two audio formats used in this project. This divergence underlines the

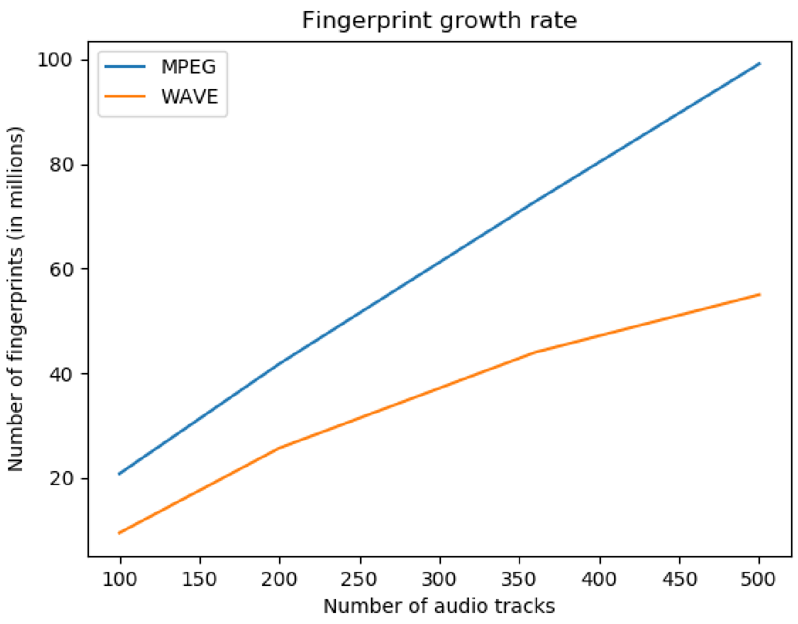


Figure 5.1: Fingerprint growth rate

two previous discussion points regarding peak detection and audio data normalization.

The two issues directly impact the number of detected peaks on a spectrogram. The number of detected peaks directly impacts the number of generated fingerprints for each audio track.

Avery Wang describes the following fingerprint creation steps: "Anchor points are chosen, each anchor point having a target zone associated with it." [29] However, there is no clear description of how anchor points are selected. The author's implementation sequentially treats each detected peak as an anchor point.

However, by using this method the author generates $O(N * fan\_value)$ where N represents the number of peaks detected for an audio track. A valuable pursuit is to experiment with alternate ways of selecting anchor points which does not simply iterate over all detected peaks, but considers anchor points at different intervals.

5.1.4 GridHash Settings & Weighted minHash

The gridHash algorithm is promising in determining the similarity between simple audio files in wave format. For other formats like mpeg this method is currently insufficient in providing a similarity calculation.

A point of further study is to define optimal grid setting (frequency and time interval and tolerance values) ranges and the effect these have on determining similarity. Further concerns involve the possibility to gridHash different format files. File compression introduces noise to the original signal and squeezes the representation into a frequency band. The effects of these compression are not accounted for in this implementation.

Another aspect worth inspecting is a different minHash algorithm for indexing the set of peak coordinates. Currently, an audio track is gridHashed by taking a list of peak coordinates, hashing it to a grid then indexing the resulting list with minHash. The problem in this scenario is that minHash treats a list of many coordinates (including numerous duplicates) as a set, and disregards the weight of distinct set elements. "MinHash can be used to compress unweighted set or binary vector, and estimate the unweighted Jaccard similarity. It is possible to modify MinHash for weighted Jaccard on **multisets** by expanding each item (or dimension) by its weight (usually its count in the multiset)." [10] Consequently, in order to take into account the weight of set elements the author proposes the use of Weighted minHash.

5.1.5 Choice of Database

The information required by the Landmark algorithm is currently stored in a MySQL database. This choice was inherited from the DejaVu implementation. Chris Kammermann, an infrastructure engineer at Shazam, "says Shazams original combination of a 'traditional tier one hardware supplier' and 'traditional tier one software supplier' operating on SQL was not flexible enough, and could not scale at the same rate as Shazams data."[6]

The author's primary concerns with the current database is scalability. Database setup is a tertiary concern for this thesis. However, a more informed choice in database systems for similar projects is an interesting area of further research.

5.2 Applications

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. [14] The proposed application in this instance is used to verify if specified ads were aired at agreed times. As an example, assume an instance where an advertisement is aired over radio. Using a Landmark based algorithm one could check if the ad was played at a specific time and if the ad was unaltered. Alterations like speeding the audio up would be measurable. By extension, the algorithm can be used as a verification tool.

Inversely, the popular Shazam app has recently been upgraded to feature image recognition. [19] The case being that app users can scan certain Shazam specific QR coded products, like a can of soda and immediately unlock brand oriented benefits. This offers a custom experience to users, allowing them to receive special offers and other commercially oriented types of targeted advertising.

More recent innovation involving the use of the Landmark algorithm concerns the fingerprinting and identification of naval vessels. The concept is similar to the original Shazam application: fingerprint the acoustic signature of a vessel as it moves through a body of water, store the information, use it later to identify unknown vessels. The signature of any ship is granted by motors, rotor pallets and hull construction. In "Landmark based Audio Fingerprinting for Naval Vessels" [23] 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 authors also propose the system be used on identifying marine life.

Currently, vessel identification is performed by a trained officer using a sonar system.[23] The addition of a landmark based system would serve as a support tool and verification system.

The GridHash algorithm takes numerous implementation ques from the Landmark algorithm. However, its use is in determining a degree of similarity between audio files. The current implementation verifies promising results for the GridHash algorithm when using wave files (uncompressed audio).

The algorithm can be included in applications that imply the use of numerous audio files. Such applications involve audio editing applications like the Ableton Live, video graphics engines like Unreal Engine or game development platforms like Unity. GridHash would work in the background by indexing the raw sound files (normally uncompressed) used in game development or audio editing. By creating an index over generated gridHash files developers could quickly identify similar sounding audio without the need of manual search or metadata categorization.

6

Conclusion

This thesis describes the open source implementation of a Python 3.6 Landmark based algorithm and a new similarity search algorithm titled GridHash. The Landmark implementation has been tested extensively to excellent accuracy results. The GridHash algorithm shows promising results and can be used in tandem with the Landmark algorithm as a recommendation system, or as a stand alone feature of a larger application.

The source code used in the thesis implementation has been made available on github: https://github.com/VladLimbean/ The repository stands as a resource for answers, clarifications and research avenues for the open source community. By doing so, the author has provided insight into the construction of a popular algorithm and a new similarity search method, evaluation and measurement for both, plus access and an easy way of continuing research in this field.

References

[1] **What is Music Information Retrieval?**, 2018. Available from: `https://musicinformationretrieval.com/why_mir.html` [cited 2018-04-20 16:15:06]. 1

[2] Porter Alastair. **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. 31

[3] Z. Broder Andrei. **Identifying and Filtering Near-Duplicate Documents**. *COM '00 Proceedings of the 11th Annual Symposium on Combinatorial Pattern Matching*, page 1 10, 2000. 9, 29

[4] Bechtold Bastian and Geier Matthias. **PySoundFile**, 2015. Available from: `https://pysoundfile.readthedocs.io/en/0.9.0/` [cited 2018-04-30 11:57:01]. 12

[5] Logan Beth. **Mel Frequency Cepstral Coefficients for Music Modelling**. *Cambridge Research Laboratory, Compaq Computer Corporation.* 7

[6] McDonald Clare. **How Shazam handles its data reserves**, 21 December 2016. Available from: `https://www.computerweekly.com/news/450409644/How-Shazam-handles-its-data-reserves/` [cited 2018-05-28 15:28:01]. 46

[7] Ellis Dan. **Robust Landmark-Based Audio Fingerprinting**, 2009. Available from: `http://labrosa.ee.columbia.edu/matlab/fingerprint/` [cited 2018-04-20 15:22:34]. 2, 3

[8] Lavry Dan. **Sampling Theory For Digital Audio**. *Lavry Engineering, Inc.*, page 27, 2004. 13

[9] Zhu Eric. **MinHash**, 2017. Available from: `https://ekzhu.github.io/datasketch/minhash.html` [cited 2018-05-21 18:33:23]. 29

[10] Zhu Eric. **Wighted MinHash**, 2017. Available from: `https://ekzhu.github.io/datasketch/weightedminhash.html` [cited 2018-05-28 14:12:23]. 46

[11] de Castro Lopo Erik. **Libsndfile**. Available from: `http://www.mega-nerd.com/libsndfile/#Features` [cited 2018-05-16 14:09:01]. 10

[12] Kekre H.B., Bhandari Nikita, Nair Nisha, Padmanabhan Purnima, and Bhandari Shravya. **A Review of Audio Fingerprinting and Comparison of Algorithms**. *International Journal of Computer Applications (0975 8887) Volume 70 No.13*, May 2013. 6

[13] A. van Nieuwenhuizen Heinrich, C. Venter Willie, and M.J. Grobler Leenta. **Comparison of Algorithms for Audio Fingerprinting**. *Telkom Grintek Centre of Excellence.* 7

[14] A van Nieuwenhuizen Heinrich, C Venter Willie, and MJ Grobler Leenta. **The Study and Implementation of Shazams Audio Fingerprinting Algorithm for Advertisement Identification**. *School of Electrical, Electronic and Computer Engineering North-West University, Potchefstroom Campus, South Africa.* 47

[15] Paul Jaccard. **Distribution de la flore alpine dans le bassin des Dranses et dans quelques régions voisines**, 1901. 9, 29

[16] Robert James. **Pydub**, 2011. Available from: `https://github.com/jiaaro/pydub` [cited 2018-04-30 11:58:01]. 12

[17] Shazam Entertainment Limited. **Patents**, 2002 2018. Available from: `https://www.shazam.com/gb/patents` [cited 2018-05-15 17:03:03]. 3

[18] Shazam Entertainment Limited. **Shazam**, 2018. Available from: `https://www.shazam.com/` [cited 2018-04-20 18:02:06]. 2

[19] Shazam Entertainment Limited. **Shazam Introduces Visual Recognition Capabilities, Opening Up A New World Of Shazamable Content**, May 28 2015 06:08. Available from: `http://news.shazam.com/pressreleases/shazam-introduces-visual-recognition-capabilities-opening-up-a-new-world-of-s` [cited 2018-04-20 16:33:12]. 47

[20] Malekesmaeili Mani and K. Ward Rabab. **A novel local audio fingerprinting algorithm**. *2012 IEEE 14th International Workshop on Multimedia Signal Processing*, 2012. 8

[21] Müller Meinard. *Information Retrieval for Music and Motion, Chapter 2 Fundamentals on Music and Audio Data.* Springer, 2007. 14

[22] Miljkovic Milos. **Milos Miljkovic: Song Matching by Analyzing and Hashing Audio Fingerprints**, Dec 4, 2015. Available from: `https://www.youtube.com/watch?v=xDFARS_oIfM` [cited 2018-04-20 18:39:13]. 3

[23] Abdur Rehman Hashmi Muhammad and Hammad Raza Rana. **Landmark based Audio Fingerprinting for Naval Vessels**. *2016 International Conference on Frontiers of Information Technology*, 2017. 47, 48

[24] Chen N, Xiao H.D., and Zhu J. **Robust audio fingerprinting based on GammaChirp frequency cepstral coefficients and chroma**. *ELECTRONICS LETTERS 13th February 2014 Vol. 50 No. 4 pp. 241 242*, 2014. 7

[25] Loughran Róisín, Walker Jacqueline, ONeill Michael, and OFarrell Marion. **The Use of Mel frequency Cepstral Coefficients in Musical Instrument Identification**. *International Computer Music Conference; 387 390*, 2008. 7

[26] van Rijn Roy. **Patent infringement**, Jul 7, 2010 22:15:34. Available from: `http://royvanrijn.com/blog/2010/07/patent-infringement/` [cited 2018-04-20 18:31:06]. 3

REFERENCES

[27] Kim Samuel, Unal Erdem, and Narayanan Shrikanth. **Music fingerprint extraction for classical music cover song identification**. *2008 IEEE International Conference on Multimedia and Expo, Pages: 1261 1264*, 2008. 8

[28] Sonnis. **30GB+ OF high-quality Sound Effects FOR GAMES, FILMS & iNTERACTIVE PROJECTS**, 2018. Available from: `https://sonniss.com/gameaudiogdc18/` [cited 2018-04-20 17:54:13]. 8

[29] 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. 3, 5, 17, 43, 45

[30] Drevo Will. **DejaVu**, 2014. Available from: `https://github.com/worldveil/dejavu` [cited 2018-04-28 18:17:01]. 3, 9

[31] Drevo Will. **fingerprint.py**, December 16, 2014. Available from: `https://github.com/worldveil/dejavu/blob/master/dejavu/fingerprint.py` [cited 2018-05-20 15:18:06]. 15

[32] Drevo Will. **Audio Fingerprinting with Python and Numpy: Peak Finding**, November 15, 2013. Available from: `http://willdrevo.com/fingerprinting-and-audio-recognition-with-python/` [cited 2018-04-30 15:19:01]. 10, 15

[33] Tournade Yoan. **Peak Detection in the Python World**, 01 Nov 2015. Available from: `https://blog.ytotech.com/2015/11/01/findpeaks-in-python/` [cited 2018-04-30 15:21:01]. 44