DTU Compute
Department of Applied Mathematics and Computer Science
June, 2020

Algorithms For Music Indexing

Bachelor Thesis

Alexander Bilton (s165635)

Gustav Hartz (s174315)

Abstract

Given a short recording of a piece of music, the music indexing problem is to identify from a music database, the song that the recording originated from. This report considers different solutions to this problem based on spectrograms. We explore how to identify peak frequencies, filter out noise from these spectrograms and how to store these filtered spectrograms (fingerprints) efficiently such that they can be compared fast. Our main result is a thorough investigation of an existing method using target zones. We benchmark this against our own solutions. Our primary solution utilises locality sensitive hashing with min hash that seeks to compress the fingerprints into even smaller signatures and quickly be able to identify near-similar signatures by hashing them using the banding technique. Based on preliminary results, our solution seems to be slightly faster but also marginally less robust in finding correct results. We conjecture that the discrepancy in robustness is due to an extra step in the target zone solution that ensures temporal alignment. In the future work section, we suggest that this is implemented in our solution as well. Along the way, we also examine trade-offs between the time spent pre-processing the music database and its implications on search times.

Preface

This thesis was prepared during the spring of 2020 at the Department of Applied Mathematics and Computer Science, at the Technical University of Denmark, DTU, in partial fulfilment for the degree Bachelor of Science in Engineering (Strategic Analysis and Systems Design), BSc. Eng.

It is assumed that the reader has knowledge of algorithms and data structures as well as basic probability theory.

Supervisors

- Philip Bille
- Inge Li Gørtz (co-supervisor)

Contents

1 Introduction

Music indexing is the process of storing and searching for music efficiently and is implemented in many different variations. A common application of music indexing is the one addressed by companies as Shazam, Soundhound and Youtube's ContentID.

The in-production application of Shazam works by

> *A user records a snippet of a song playing from e.g a car radio and wants to know what song it is. Given the snippet, Shazam searches its internal database and returns the appropriate song, if found.*

Since the user only records a small subset of the song and real-world recordings contain noise and distortions from poor audio recording devices, robust and efficient algorithms are needed for this problem.

1.1 Digital music

In the real world, music is comprised of a set of continuous sound waves. In the digital world, however, one cannot store continuous signals and must therefore resort to a discrete representation of the continuous signal.

A standard way of representing digital audio is *pulse-coded modulization (PCM)*. The PCM contains structural information about the amplitude (loudness) of the piece of audio at discrete points in time. There are two important definitions here, namely the **sampling rate** and the **bit-depth**. The sampling rate is the number of times the analogous signal is sampled per second, the industry standard being 44.1 kHz, i.e 44,100 samples for 1 second of audio. The bit-depth determines the amplitude resolution or the number of loudness intervals in the song. The industry standard is a bit-depth of 16 bits meaning that there are 65,536 discrete levels of loudness in the song.

Thus, a PCM-encoded representation of a 3-minute song using the industry standard is an array of integers:

$$a = (a_1, a_2 ... a_n)$$

, where $n = 44,100 \cdot (3 \cdot 60) \approx$ 8 million and $\forall_{a_i}(i \in \{1..n\} \wedge a_i \in [-32,768, 32,767])$

1.2 Spectrograms

The spectrogram is a 3-dimensional representation of a piece of sound. It is very informative in the way it reveals how frequencies change over time.

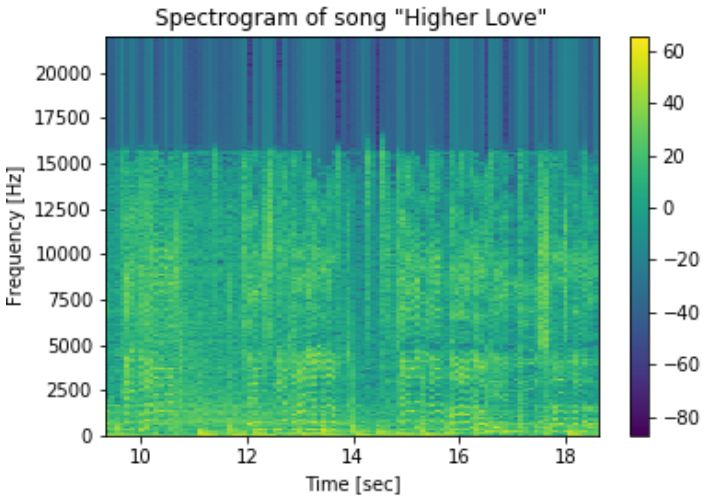


Figure 1.1: Spectrogram of a snippet of the song ”Higher love”

In figure 1.1, an example of such spectrogram is given. The y-axis represents the frequencies present in the song, the x-axis the time, while the colouring indicates the power a particular frequency at a particular time relative to the carrier signal dBc.

Converting a signal to from the time domain to a frequency domain needed for a spectrogram can be done by applying a fourier transformation to the signal. The fourier transformation is a mathematical function from signal processing, designed for the conversion of continuous signals. The version utilised in this project is the fast-fourier transformation which is a divide-and-conquer approach to the discrete fourier transformation that can be performed on the PCM-signal in $O(n \cdot \log(n))$ where n is the length of the sample [1]. This report is not concerned with algorithms for the creation of spectrograms, and will as such ignore the time required for computing spectrograms.

1.3 Problem definition and scope

We define the problem of music indexing informally as follows. Given a recorded audio sample of a piece of (unknown) music, retrieve from the music database the closest match, that is, the song in the database that the recorded sample is most likely to be a part of.

In this project, we will restrict ourselves to techniques that are based on spectrograms, partly to give a clear boundary for this report and partly because current literature indicates that spectrograms provide good results.

We will introduce techniques for transforming spectrograms to sparse fingerprints that are to some extent noise resistant. However, these techniques rely mainly on the structure and composition of music along with psycho acoustic models and are therefore not of particular algorithmic interest. We will therefore resort mostly to known techniques and focus on investigating different data structures for storing these fingerprints and algorithms for comparing them efficiently.

1.4 Related work

Algorithms for music indexing is a fairly well researched topic and proposed solutions spans from deep learning as in the case of Google’s Music Recognition to more classic algorithmic approaches based on spectrograms as in the case of Shazam [2]. The work

in this project is closely related to solutions presented in the 2003 article ”An Industrial-Strength Audio Search Algorithm” [2]. Numerous other projects based on that same article has been a source of inspiration for this project as well [3][4]

1.5 Preliminaries

The **spectrogram**, denoted S, has two associated parameters f and t which denotes the number of frequency bins and the number of time bins, respectively. Thus, an entry in the spectrogram $S(i, j)$ corresponds to the loudness of the $i'th$ frequency bin at time bin j.

A **fingerprint**, denoted fp, is derived directly from the spectrogram. We shall see that the fingerprint distincts itself mainly by being more space-efficient and containing less noise, depending on which fingerprinting algorithm is used. Fingerprints are what will be used for comparing songs and the music database is essentially a collection of fingerprints.

A few other important definitions are:

- A **snippet** is a, potentially noisy, subset of some unknown song, for which we wish to identify the unknown song
- A **candidate** is a song in the music database that might be the unknown song of which the snippet is a subset
- *fp-db* denotes the collection of the fingerprints in the fingerprint database. The number of songs in the database is denoted C

1.6 Performance metrics

In designing algorithms for music indexing, there are a few metrics that characterize the overall quality of the system [5]

- **Running time**. The algorithm needs to return results fast as a response to a snippet.
- **Fingerprint space**. A desirable property of a fingerprinting algorithm is that the space consumption of the resulting fingerprint is as small as possible.
- **Robustness**. An algorithm should be able to return appropriate candidates for snippets that contain noise.
- The **granularity** describes how large a snippet is required for the algorithm to return an appropriate match. In an applications of music indexing, it would not be desirable if the user has to record half the song for the algorithm to be able to identify it.

Some of these metrics will counteract each other. For instance, aiming for a very robust algorithm that rarely returns an incorrect result can potentially lead to storing large fingerprints, which in turn will increase space usage and search times for candidates. Therefore, clever algorithms must find the optimal *trade-off* between these characteristics.

2 Algorithm 1

The initial algorithm employs a naive approach to both fingerprinting and searching. The goal of this first algorithm is to have a benchmark for the algorithms presented later in the report.

2.1 Fingerprint

The naive approach for fingerprinting is to let the fingerprint equal to the spectrogram, that is, the fingerprint is a $F \times N$ matrix that contains 32-bit integers corresponding to the loudness of the frequencies at each time bin:

$$S = fp = \begin{pmatrix} s_{1,1} & . & . & s_{1,N} \\ . & . & . & . \\ . & . & . & . \\ s_{F,1} & . & . & s_{F,N} \end{pmatrix}$$

2.2 Searching

Searching for a match is the process of comparing the fingerprint of the snippet song (fp_s) to the collection of fingerprints, referred to as the fingerprint database (***fp-db***). We want to identify and return the candidate $fp_c \in$ ***fp-db*** with the lowest error by some error function.

Since the snippet song is by definition shorter than the full songs in the database, the error of matching the snippet with some candidate fp_c is the error of the *best alignment* between fp_s and fp_c. Let the dimensions of fp_s be (F, N_s) and of a given candidate song fp_c be (F, N_c). Since the snippet is a short recording, we have that $N_s < N_c$. The minimum error is then given by:

$$\underset{i \in \{1..N_c - N_s\}}{\arg\min} \; (fp_s - fp_c[:][i : i + N_s])^{|\cdot|} \tag{2.1}$$

That is, the error of matching fp_s and fp_c is the minimum sum of the absolute difference over all alignments between fp_s and fp_c. In figure 1 we provide the pseudocode for finding the best candidate in the database that matches the snippet.

Algorithm 1

```
1:  procedure search-alignment(fp, fp-db)                    ▷ find best candidate
2:      Ns ← fp.columns.size
3:      errors[] ← fp-db.length
4:
5:      for each candidate in fp-db do          ▷ Calculate min error for each candidate
6:          Ns ← candidate.columns.size
7:          best_error = ∞
8:
9:          for i ← 1, (Nc − Ns) do                  ▷ Number of possible alignments
10:             error ← sum of element-wise abs difference of fp and candidate[:][i:i+Ns]
11:             if error < best_error then
12:                 best_error ← error
13:         errors[candidate] ← best_error
14:     return candidate with lowest error
```

For each of the candidates in the database, algorithm 1 goes linearly through the possible alignments of the snippet and the candidate. The error in line 10 calculates the error for the alignment. When a candidate is processed, line 13 saves the best error for the candidate. Finally, we return the candidate with the lowest error.

2.3 Performance

We do not consider the time complexity for the fourier transformation creating the spectrograms and thus in the case of naive fingerprints, the running time for fingerprinting is $O(1)$.

To smoothen the analysis of running time, we introduce a new parameter $N_T = \sum_{i=0}^{C} N_i$, denoting the total number of time bins in our music database.

For each of the fingerprints in the database, **algorithm 1** calculates an error for *each* possible alignment of the snippet and the candidate. There are $N_c - N_s$ possible alignments of the matrices each of size $F \times N_s$. Since F is not dependent on the input size and determined by the parameters of the Fourier transformation, we regard it as constant in this setting and thus apply $O(N_s)$ operations for each alignment yielding a running time of $O(N_s \cdot (N_c - N_s))$ for each candidate.

Since N_T denotes the total number of time bins in our database, the total number of alignments between the snippet and the whole database is $O(N_T - N_s)$. By substituting into the running time of aligning a single candidate, we obtain a total running time of $O(N_s \cdot (N_T - N_s)) = O(N_s \cdot N_T - N_s^2)$. When the database is large, we have that $N_T \gg N_s$ and the size of the database becomes the bottleneck of this algorithm.

Space

The space consumption for the database is $O(N_T)$ as that is the total number of time bins in our database, each of constant size F.

3 Algorithm 2

Recall the common application of music indexing: *"A user records a snippet of a song playing from e.g a car radio and wants to know what song it is"*. A key challenge in this issue is to distinguish between music and noise in the recording. This fact is at the core of the next fingerprinting solution.

3.1 Fingerprint

In a real-world recording of music, noise will most often be less significant than the music itself. That is, the background noise is expected to have a lower amplitude. Therefore, an approach to creating the fingerprint is to filter out the values with a non-significant amplitude in the spectrogram.

$$S = \begin{pmatrix} s_{1,1} & . & . & s_{1,N} \\ . & . & . & . \\ . & . & . & . \\ s_{F,1} & . & . & s_{F,N} \end{pmatrix} \quad \rightarrow \quad fp = \begin{pmatrix} f_{1,1} & . & . & f_{1,N} \\ . & . & . & . \\ . & . & . & . \\ f_{F,1} & . & . & f_{F,N} \end{pmatrix}$$

Formally, the transformation from the spectrogram to the fingerprint, by filtering, is a mapping $\mathbb{Z}^{F,N} \rightarrow \{0,1\}^{F,N}$.

Informally, the transformation from the spectrogram to the fingerprint converts an entry $(i,j) \in S$ to a binary indicator $(i,j)' \in fp$, indicating whether the particular frequency bin i is *significant* w.r.t the other frequency bins at time bin j.

On the notion of whether an entry (i,j) is *significant*, we resort to a method inspired by a similar project [4]. The filtering method divides the spectrogram into 7 logarithmic bands. This band selection is based on psycho acoustic models of the human hearing [4][6]. The bands are as follows:

- 0-100 Hz - very low sound band
- 100-200 Hz - low sound band
- 200-400 Hz - mid low sound band
- 400-800 Hz - mid sound band
- 800-1600 Hz - mid high sound band
- 1600-5000 Hz - high sound band
- Above 5000 Hz - very high sound band

For each time bin and we identify the 7 entries, one from each band, that have the highest amplitude in their respective bands. From those 7 entries, we keep the ones that are above the mean calculated from the same 7 values. That is, we set $fp(i,j) = 1$ iff $S(i,j)$ is larger than the average of the strongest bins in each band at a given time bin and 0, otherwise. With this procedure we identify which frequencies are the most important at each time bin of the song across the different *categories* (bands) of frequencies, largely without being

effected by artificial increases in low frequency amplitudes[4] which is common in modern music. Algorithm 2 provides pseudocode for the implementation.

Algorithm 2

```
1:  procedure transform-binary(S)          ▷ transform spectrogram to binary fingerprint
2:      F, N ← S.size
3:      fp ← array[F,N]
4:      band-max ← array[7]
5:
6:      for i ← 1, N do
7:          band-max ← List of maximum values for each band in column i
8:          threshold ← Mean of band-max
9:          bands-max-index ← i index of the max values
10:
11:         for j ← 1, F do                    ▷ Assign values to fingerprint matrix
12:             if S[i, j] ≥ thresholdandj ∈ bands-max-index then
13:                 fp[i, j] ← 1
14:             else
15:                 fp[i, j] ← 0
16:
17:     return fp                              ▷ return fingerprint matrix
```

A related version of this fingerprinting algorithm is *bands-mean*. This algorithm filters the frequencies based on a running mean threshold calculated from neighbouring time bins. We will not provide pseudocode for this as it is a slight modification to the above algorithm.

3.2 Searching

The approach to searching is to compare the binary fingerprint of the snippet song with the fingerprint in the database. The score for each candidate will be the minimum error of aligning the two fingerprints. We again utilise the error function from equation 2.1 and algorithm 1, this time only on binary matrices instead of 32-bit integer matrices. The error produced from algorithm 1 now corresponds to the hamming-distance between the two alignments. "The hamming distance between x and y, denoted by $H(x, y)$, is defined to be the number of places where x and y disagree. That is, if $x = x_1, x_2...x_n$ and $y = y_1, y_2...y_n$ then $H(x, y)$ is the number of positions i for which $x_i \neq y_i$" [7]. Minimising the hamming distance will then give the minimum number of frequency-time points in which the snippet and the candidate song disagree in the *best alignment*.

3.3 Performance

The total running time of the algorithm is the running time for fingerprinting the database plus the time for searching for a match in the fingerprint database. We analyse the algorithms separately.

In **algorithm 2** we loop N arrays each of size F, Again, we regard F as constant. First we find the band-max values, second we assign values to fp. Looping N arrays of size F is $O(N)$ thus yielding a $O(N)$ running time for fingerprinting a song with N time-bins in its spectrogram. Fingerprinting the whole music database then becomes $O(N_T)$.

We have already stated the running time of aligning fp_s with the candidates in algorithm **algorithm 1** as $O(N_s \cdot N_T - N_s^2)$

The total running time for fingerprinting and searching with these two procedures is then $O(N_T) + O(N_s \cdot N_T - N_s^2) = O(N_s \cdot N_T - N_s^2)$. Thus, the running time of the second algorithm is dominated by the search.

Space

The space consumption for the database is asymptotically the same as in the previous algorithm, $O(N_T)$, as we still store N_T time bins each of constant size F.

4 Algorithm 3

So far we have considered algorithms that scans linearly through the fingerprints of the database and searches for the best possible alignment. However, when the database becomes large, this quickly becomes infeasible. In this section, we describe, with an offset in chapter 3 of Mining Massive Datasets [8], a method using locality sensitive hashing with minhash signatures. The aim is to reduce the dimensionality of the fingerprints and only investigate the fingerprints in the database that are likely to be candidates.

Fingerprinting
In this algorithm we re-use the fingerprinting procedure from algorithm 2 such that we produce a binary matrix $F \times N$:

$$fp = \begin{pmatrix} f_{1,1} & . & . & f_{1,N} \\ . & . & . & . \\ . & . & . & . \\ f_{F,1} & . & . & f_{F,N} \end{pmatrix}$$

4.1 Searching

A fingerprint of the above type can be thought of as a collection of sets such that each set corresponds to a column N_i in the fingerprint that indicates the significant frequencies at time bin i. Thus, for a fingerprint with N time bins, the collection contains N binary sets each of dimension F.

The problem of matching a snippet fingerprint against the music database can then be defined as retrieving from the database the song that has *the most sets* that are similar to the sets of the snippet.

However, we want to avoid comparing every set (column) of the snippet to every set in the database. Doing so exhaustively would require us to do $N_s \cdot N_T$ comparisons each of length F. Instead, we can pre-process the database by creating minhash signatures of the sets and use locality sensitive hashing to hash similar sets together. Thus, when receiving a snippet, we only have to create minhash signature for the snippet and then investigate the sets that are hashed to the same buckets as those of the snippet.

4.1.1 Minhashing

The jaccard similarity is a measure for how similar two sets are. It is defined as the ratio between the intersection and the union $J(A, B) = \frac{A \cap B}{A \cup B}$. For two binary sets A and B, the jaccard similarity is computed as

$$J(A, B) = \frac{M_{11}}{M_{01} + M_{10} + M_{11}}$$

such that M_{11} is the number of elements where $A_i = B_i = 1$, M_{10} is the number of elements where $A_i = 1$ and $B_i = 0$ and so on. Exemplified:

$$A = \begin{pmatrix} 0 \\ 1 \\ 1 \end{pmatrix} \qquad B = \begin{pmatrix} 1 \\ 1 \\ 0 \end{pmatrix} \qquad J(A, B) = \frac{1}{1+1+1} = \frac{1}{3}$$

When the number of sets is large and the number binary attributes is large, calculating the jaccard coefficient between every set becomes time consuming. Minhashing is a way of reducing the dimensionality of the sets while preserving an estimate of the jaccard similarity between them.

Given n d-dimensional sets, we can illustrate these in a characteristic matrix M:

$$M = \begin{array}{c} \textbf{row} \\ 1 \\ 2 \\ . \\ . \\ d \end{array} \begin{array}{c} \begin{array}{cccccc} S_1 & S_2 & . & . & S_n \end{array} \\ \begin{pmatrix} 1 & 0 & . & . & 1 \\ 0 & 1 & . & . & 1 \\ . & . & . & . & . \\ . & . & . & . & . \\ S_{d1} & S_{d2} & . & . & S_{dn} \end{pmatrix} \end{array}$$

Imagine a random permutation M' on the rows characteristic matrix. If two sets S_i and S_j both have a 1 in the first row in the permuted order, the belief about the similarity of S_i and S_j increases. At the core of the minhashing procedure is the fact that the probability of S_i and S_j both have a 1 in the first row of M' is equal to $J(S_1, S_2)$. Therefore, by permuting M in a random order a number of times, we can estimate the jaccard similarity of the sets.

Instead of permuting the rows of M explicitly, which is time consuming, we simulate k permutations by defining k hash functions $h_1(r), ..., h_k(r)$ that maps a row to a new location in the table in form of an integer $h_i(r) \in \{1..d\}$. If there are no collisions, we have a perfect permutation. In general, there might be a few collisions but as long as the number of collisions is small compared to d it does not impact the result.

$$M = \begin{array}{c} \textbf{row} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{ccc|cc} S_1 & S_2 & S_3 & \textit{2x +1 mod 5} & \textit{3x +3 mod 5} \end{array} \\ \left(\begin{array}{ccc|cc} 1 & 0 & 0 & 1 & 3 \\ 0 & 1 & 1 & 3 & 1 \\ 1 & 1 & 0 & 0 & 4 \\ 0 & 1 & 0 & 2 & 2 \\ 1 & 1 & 1 & 4 & 0 \end{array}\right) \end{array}$$

Figure 4.1: A characteristic matrix with 5 rows and 3 sets shown with two hash functions that define permutations on the rows of $[2, 0, 3, 1, 4]$ and $[4, 1, 3, 0, 2]$, respectively.

The minhashing procedure evalutes the columns in every permutation defined by $h_1(r), ..., h_k(r)$ and notes the row number of the fist entry in each column that has a 1 in each permutation. The result is a signature matrix with the same number of columns as M and a row for each hash function. An entry $sig(i, c)$ in the signature matrix contains the row number of the first entry that has a 1 in column c in the permuted order defined by the $i'th$ hash function $h_i(r)$. Thus, the example from figure 4.1 yields the signature matrix seen in figure 4.2.

$$sig(M) = \begin{array}{c} \\ h_1 \\ h_2 \end{array} \begin{array}{c} \begin{array}{ccc} S_1 & S_2 & S_3 \end{array} \\ \begin{pmatrix} 0 & 0 & 3 \\ 0 & 0 & 0 \end{pmatrix} \end{array}$$

Figure 4.2: Signature matrix from minhashing the matrix in figure 4.1

As figure 4.2 verifies, if $sig(i, S_u) = sig(i, S_v)$ for some $u \neq v$, it is the case that both S_u and S_v have a 1 in the same row in the characteristic matrix, and our belief about their similarity thus increases.

On the above notion, the jaccard similarity can be estimated from the signature matrix. In this example, the estimated jaccard similarity between S_1 and S_2 is $J(S_1, S_2) = 1$ whereas the true jaccard similarity is $\frac{2}{5}$. As the number of hash functions increase, so does the precision of the estimate.

In the case of music indexing, we can view the columns of the fingerprint as sets in the characteristic matrix. We can then create smaller signatures of the columns of the fingerprint. Recall that the dimensions of the fingerprint is $F \times N$. The signature of the fingerprint will have the size $k \times N$ where k is the number of hash functions. k will be less than F. Thus, the columns of the signature matrix will be faster to compare than the original columns of the fingerprint. However, even when $k \ll F$, calculating the similarity between columns of a snippet fingerprint and those of the music database is still time consuming when the database is large.

4.1.2 Locality sensitive hashing using minhash

Locality sensitive hashing (LSH) is a technique used to hash items such that near-similar items have a higher probability of being hashed to the same bucket than dissimilar items. There are numerous ways of determining how similar two items are. One such example is to use the banding technique on the signatures from the minhash.

The banding technique divides the signatures into b bands each consisting of r rows such that $b \cdot r$ equals the number of rows in the signature matrix. Now, a set S_i contains b r-dimensional vectors. Two sets S_i and S_j are considered candidates for being similar if any of their b subsets hash to the same value using direct addressing. The choice of b and r controls the probability of two sets becoming a candidate pair for being similar. If r is very small, there will be many collisions and thus many candidate pairs. However, if r is too large, we might prevent near-similar sets from becoming a candidate pair. An illustration of the banding technique is given in figure 4.3:

	S_1	S_2
h_1	112	98
h_2	450	440
h_3	48	48
h_4	692	692
h_5	332	249
h_6	120	179

Figure 4.3: Illustration of the banding technique with signatures of length 6. In this example we have $b = 3$ and $r = 2$. The colours indicate the bands.

From the example in figure 4.3, we see two sets S_1 and S_2 with signatures of length 6. An element-wise comparison would make these signatures different. However, by the banding technique, we see that S_1 and S_2 are similar in the second band and therefore become candidates for being similar.

The probability of becoming a candidate pair
Assume that two sets S_i and S_j have $J(S_1, S_2) = s$. Recall that the probability of two sets agreeing on a particular entry is equal to the jaccard similarity. The probability of becoming a candidate pair depends on b and r in the following way [8]:

- Since the probability of two sets agreeing in one entry is s and a band consists of r rows, the probability of two sets agreeing in one band is s^r.
- Thus, the probability two sets having different values in at least one entry of a particular band is $1 - s^r$
- If there are b bands, the probability of two sets having different values in at least 1 entry of each band is $(1 - s^r)^b$

From the above calculations it can be inferred that the probability that two sets S_i and S_j becomes a candidate pair, that is they hash to the same value for at least one of the b bands, is $1 - (1 - s^r)^b$. For instance, with a signature length of 100 if $J(S_i, S_j) = .6$ then they become a candidate pair with ≈ 5% probability if $b = r = 10$. However, if $r = 5$ and $b = 20$ then there is an 80% chance of becoming a candidate pair. Thus, adjusting the size of the bands is a trade off between not having too many candidates and not missing out on important ones.

Output of LSH
The result of the locality sensitive hashing procedure using minhash signatures is a set of b hash tables as illustrated in figure 4.4. Instead of storing the signatures themselves in the hash table, we store a pointer to the song of the particular signature.

Hash table for band 1	
Key	**Pointers to songs**
k_1	$[p_1, p_2 \ldots.]$
k_2	$[p_j \ldots.]$
.	[….]
.	[….]
.	[….]

Hash table for band 2	
Key	**Pointers to songs**
k_1	[….]
k_2	$[p_i \ldots.]$
.	[….]
.	[….]
.	[….]

.....

Hash table for band b	
Key	**Pointers to songs**
k_1	[….]
k_2	$[p_i \ldots.]$
.	[….]
.	[….]
.	[….]

Figure 4.4: Example of hash tables produced by LSH. Each key corresponds to a direct hash of an r-dimensional vector. The values for a key k are pointers to the songs such that k is a subset in one of their signatures

Finding a candidate
Given a snippet song, we apply the minhash procedure with the same k hash functions as applied to the database and hash the signatures using the LSH procedure. We then check which song has the most signatures that have been hashed to the same buckets as the signatures of the snippet. The song that is returned corresponds to the song from the music database with the most near-similar columns in their fingerprint.

4.2 Performance

Precomputing the music database

Given a set of spectrograms we can precompute the music database and store it as the set of hash tables produced by LSH. It requires the following steps:

- Fingerprinting the database
- Computing the minhash signatures
- Applying the LSH procedure to the signature matrix

As shown previously, fingerprinting the music database using the bands procedure in algorithm 2 is $O(N_T)$.

Computing the min hash signatures requires looping over the rows of the characteristic matrix and for each column calculate each of the k hash values. There is a constant number (F) of rows and N_T columns in the characteristic matrix. Thus the running time for minhashing the signatures become $O(N_T \cdot k)$.

Computing the hash tables in LSH depends on the number of bands. For each of the b bands we use direct addressing on N_T vectors of size r. Since $k = b \cdot r$ we obtain a running time of $O(N_T \cdot k)$ for hashing the signatures assuming that hashing is $O(1)$.

The total running time for pre-computing the music database is then $O(N_T \cdot k)$

Searching for a result

Given a snippet of a song, searching for a result in this algorithm requires the same steps as when precomputing the music database plus the additional time for searching the hash tables for similar signatures.

For the first part, we substitute N_T for N_s in the above and get a running time of $O(N_s \cdot k)$. If we keep note of which buckets we hash to when hashing the signature of the snippet, we only have to search through the pointers in these buckets.

In the worst case, each band of each signature hashes to a different address in which case we need to look through $O(b \cdot N_s)$ addresses each containing $O(C)$ pointers. This only happens if every song in the database has hashed to every address of the snippet. However, the expected number of addresses is much lower and so is the number of pointers at each address.

In total, we obtain a running time of $O(N_s \cdot k + N_s \cdot b \cdot C)$ for searching. In a large setting, when $C \cdot b > k$, the bottleneck will, as expected, come from the $N_s \cdot b \cdot C$ term, making the processing of the database data more computationally demanding than the pre-processing of the snippet .

Space

After producing the hash tables in *LSH*, we do not have to store the fingerprints or the minhash signatures. Thus, the space consumption for the database is the space consumption of the hash tables. The signature matrix is $k \times N_T$. We hash b vectors of size r for each of the signatures, thus taking up $O(b \cdot r \cdot N_T) = O(k \cdot N_T)$. These elements consist of the direct addresses of the keys and the associated song pointers.

5 Algorithm 4

Algorithm 4 for the music indexing problem uses a method described as *target zoning*, which is a term coined in the original Shazam paper [2] and further examined in other projects [4]. This algorithm also introduces a new way of fingerprinting.

The **fingerprint** used in this algorithm is based on the method applied in the Dejavu project [3]. It utilises concepts from non-linear signal processing to identify peak frequencies. This report will not go into the underlying concepts, but merely utilise it as the method has proven results.

The **search** for a match is two-fold. The first part aims to quickly identify potential matches with the use of an inverted index hash table. The second part evaluates the temporal alignment between a snippet and the candidates found in the first search. The latter could potentially also be solved by the linear alignment from algorithm 1 and algorithm 2.

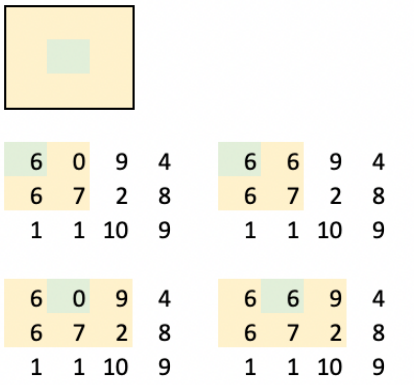
Inverted index
In general, an inverted index stores a list of unique tokens where each element has pointers to its source. An example of this is the word list found in the final pages of almost any non-fiction book. Here is listed a set of words and their location in the book. It is a fast way of finding sources with specific tokens. Inverted indexes is a well researched topic and is implemented across many applications, most notably in many web and search engines [9].

5.1 Fingerprint

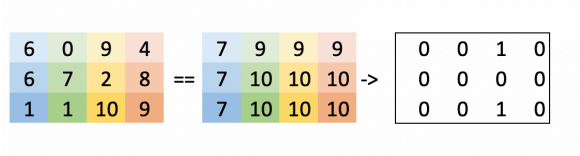
The general idea of this fingerprinting algorithm is to keep values that are the largest in their *neighbourhood*. The algorithm filters the spectrogram in three steps:

- Apply a morphology mask over the spectrogram and propagate the largest values through the neighbourhood. Store the result in S'
- Compare S and S' and keep an element (i, j) iff $S(i, j) = S'(i, j)$. Those are the largest values in their respective neighbourhood
- As a final step, remove the values from the above step that are below an amplitude threshold value.

The first step is illustrated in figure 5.1a. The yellow area is the current neighbourhood of the cell in the green box. The second step is illustrated in 5.1b where each value is compared to the same coloured cell.



(a) Maximum filter generation operation



(b) Maximum filter compare operation

Figure 5.1: 5.1a illustrates the maximum filter operation. Upper left corner shows the mask and below the operations are show where the left column is the original used for calculations and the right shows the result of each calculation. In 5.1b the compare operation of the maximum filter is performed to find the location of the maximum values of each neighbourhood

Thus, we have $fp(i,j) = 1$ iff 1) It is the largest in its neighbourhood and 2) it is above a threshold amplitude value. $fp(i,j) = 0$, otherwise. Instead of using the matrix data structure as with the previous fingerprinting algorithms, we utilise a more dense representation only indicating the index of the peak frequencies. This is a more efficient when the matrix is sparse.

The fingerprint is stored as a list of N lists such that an entry $[i][j]$ contains the index of the $j'th$ peak frequency at time bin i in the spectrogram S. The fingerprint is illustrated in figure 5.2.

$$((p_{11}, p_{21}, ..), .., (p_{1N}, p_{2N}, ..))$$

Figure 5.2: Illustration of a fingerprint on a spectrogram with N timebins

5.2 Searching

Target zones

The general idea of the method is to utilise groups of peak frequencies, called target zones, instead of considering each point in the fingerprint individually. A clever solution of hashing these points together with an *anchor point* allows for efficient assessment of temporal links between points. An illustration of the concept is given in figure 5.3.

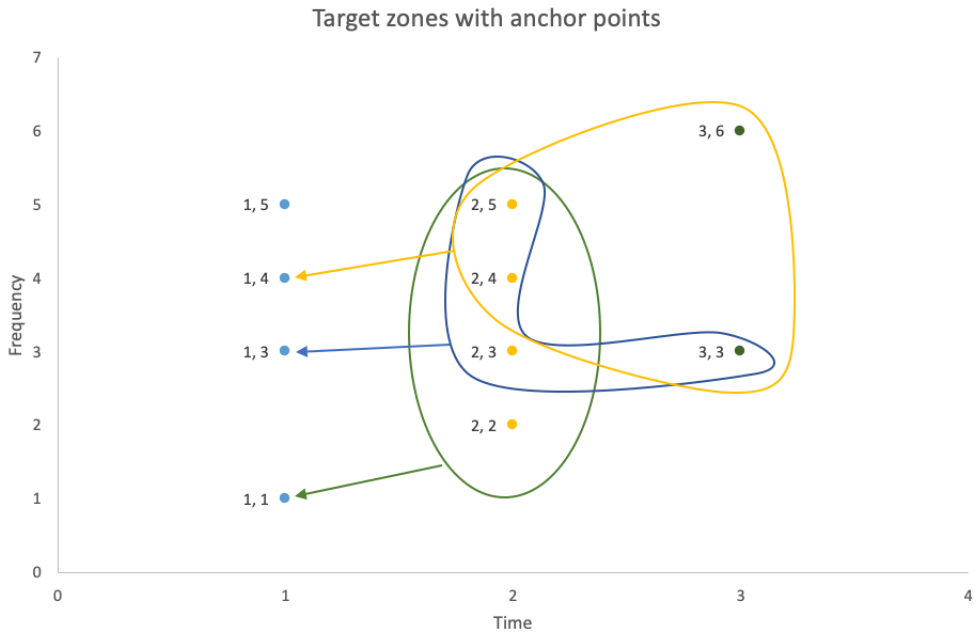


Figure 5.3: Illustration of target zones of size 4. The labelling on the points, e.g $(1, 4)$ indicates a peak frequency at time bin 1, frequency bin 4. The anchor point offset is -4

Figure 5.3 shows an example with three target zones, indicated by colour, and their associated anchor point. For instance, the yellow target zone consists of the peak frequencies $\{(2, 4), (2, 5), (3, 3), (3, 6)\}$ with $(1, 4)$ as its anchor point. Notice that the size of the target zone approximately determines the number of target zones to which each peak frequency belongs.

Fingerprint hashing

Each of the frequency points in each target zone will be hashed to an address in the following way. The **address** is a tuple constructed as (f_a, f_p, Δ), where f_a is the frequency index of the anchor point, f_p is the frequency index of the point itself and Δ is the number of time bins between f_a and f_p. The **value** connected to each address is a tuple containing the id of the song from which the fingerprint originated and the absolute time of the target zone anchor point in the song. In the case of hashing the snippet, the value is just the absolute time of the anchor point, since we have no song id.

In figure 5.3 the green target zone has the anchor point $(1, 1)$. This would generate the following 4 addresses $(1, 2, 1)$, $(1, 3, 1)$, $(1, 4, 1)$, $(1, 5, 1)$ all pointing to the tuple $(song\_id, 1)$

The strength of this hashing technique lies in how it incorporates temporality in that if two fingerprints share an address, it means that they both have a point in time with the *same* peak frequencies and the same Δ space between them. Algorithm 3 provides the pseudocode for hashing a fingerprint.

Algorithm 3 Fingerprint target zone hash generation

```
1:  procedure generate-fingerprint-hash(fp,t-size)
2:      N ← fp.size
3:      target_zone_hashes ← array[N]
4:      D ← empty dictionary
5:
6:      for i ← 1, N do                          ▷ Loop peak frequencies for each time bin
7:          for j ← 1, t-size do:                                  ▷ points in target zone
8:              address ← generate address for target zone point
9:              value ← generate value for address
10:             if address ∈ D then
11:                 Add value to address
12:             else
13:                 Add address to D
14:                 Add value to address
15:     return D                                  ▷ return inverted index structure for fp
```

For generating the inverted index for the complete database, the above method is applied for every fingerprint in the music database and added to the *same* inverted index D.

Identifying candidates

Given a snippet fingerprint to match against the database, we want to retrieve the candidates that are likely to be a match with the snippet. We do this by performing the following steps:

- Generate addresses for the snippet fingerprint
- For each address, add the values that are present at the address in the inverted index to a list. Recall that the values are a tuple of a song id and absolute point in time of anchor.
- Go through the list and count the number of times each tuple is present. This corresponds to counting the number of matches per target zone in a particular song.
- If the count of a tuple is equal to the size of the target zones (a full match), note which song it belongs to and increment a counter of that song by 1
- Return the set of songs where the counter is at least some fraction of the number of target zone points generated by the snippet.

In algorithm 4, we provide pseudocode for identifying candidates with the above procedure. *fp* is the fingerprint of the snippet, *t-size* is the size of the target zone and *hash-db* is the inverted index of the music database.

Algorithm 4 Assessing candidates and target zones matches

```
1:  procedure find-candidates(fp, t-size, hash-db)
2:      M ← empty list
3:      fp-tz-count ← Number of target zones in fp
4:      tuple-count ← empty dictionary
5:      song-count ← empty dictionary
6:      candidates ← empty set
7:
8:      addresses ← Generate addresses for fp using algorithm 3
9:
10:     for address in addresses do          ▷ Find tuples present at each snippet address
11:         Add tuples of hash-db[address] to M
12:
13:     for each tuple in M do                          ▷ Count presence of each tuple
14:         Increment tuple-count[tuple] by 1
15:
16:     for each tuple in tuple-count do      ▷ Count number of full target zones per song
17:         if tuple.size = t-size then
18:             Increment song-count for tuple.song by 1
19:
20:     for each song in song-count do                          ▷ Find candidates
21:         if song-count[song] ≥ c · fp-tz-count then
22:             add song to candidates
23:
24:     return candidates
```

The above procedure returns a list of candidate songs. Line 21 uses a parameter $c \in [0; 1]$ controlling the fraction of matching target zones required between a snippet and a song to become a candidate.

Finding a match

We now have a list of potential candidate songs that match most of the target zones in the snippet. However, in the selection of candidates the ordering of the target zones was of no concern. Figure 5.4 illustrates the potential issues with this.

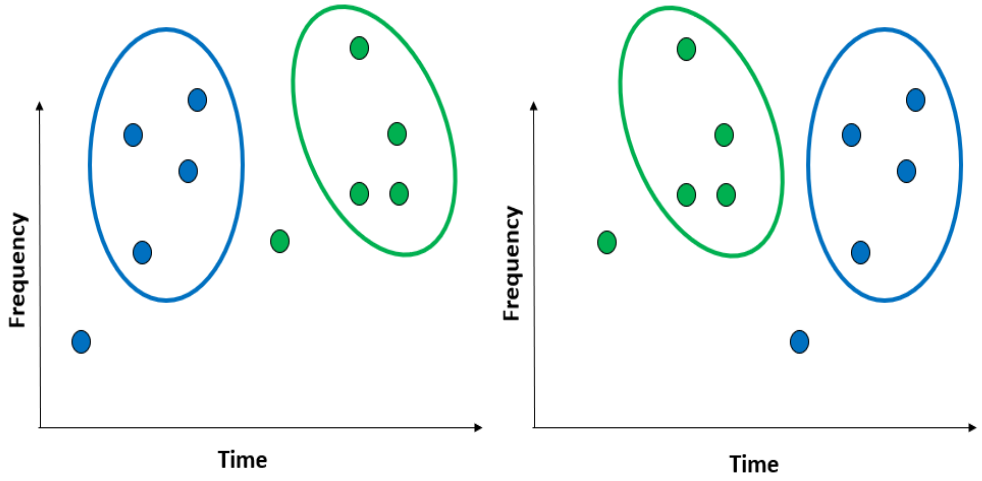


Figure 5.4: Illustration of two fingerprints with the same target zones but in different order

The algorithm described for identifying candidates would not be able to distinguish between the two fingerprints illustrated in figure 5.4. They both have the same target zones and distances to their anchor point. However, the important frequencies are in reversed order of each other and the two songs that the fingerprints represent might not sound similar.

The principle of the temporal alignment search is based on the notion that if the snippet and the song originates from the same piece of music then the snippet is a subset of the song, shifted with some Δ-value. For all of the potential candidates we wish to find the particular Δ that optimises this relation. It works as follows:

- For each address in the snippet we find the values present in the hash db belonging to candidate songs.
- We calculate Δ as the difference between the time of the anchor points present in the hash db for this address and the absolute time of the snippet anchor point.
- We store each of the above values in a list for each song
- We return the song with the **highest amount of duplicate Δ's** in their list

The intuition is that the song with the largest amount of duplicate Δ's is the candidate song with the most matching target zones having the same temporal alignment of target zones as in the snippet. The Δ value will, in a perfect example, correspond to the amount of time (in time bin units) into the song at which the snippet was recorded. Figure 5.5 illustrates what we are searching for:

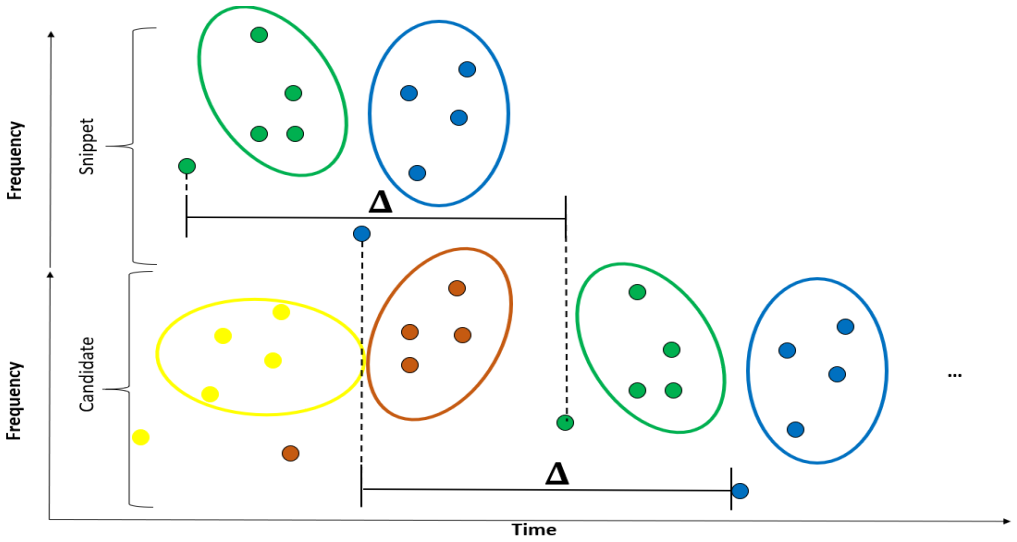


Figure 5.5: Illustration of a perfect Δ-shift between a snippet and a candidate. Matching target zones are indicated by colour

In the example of figure 5.5, both target zone matches have the same Δ-shift between them, exemplifying a perfect temporal alignment.

Algorithm 5 provide pseudocode for finding the candidate with the best temporal alignment.

Algorithm 5 Finding the best match from the candidates

```
1: procedure find-match(addresses, M, candidates)
2:     time-deltas ← empty dictionary
3:
4:     for each tuple in M do                          ▷ Loop each candidate tuple
5:         if tuple.song is in candidates then
6:             for each snippet-tuple in addresses[tuple.address] do
7:                 time-deltas[tuple.song] ← tuple.time - snippet-tuple.time
8:
9:     return song with highest number of duplicates in time-deltas
```

Note that the algorithm 5 simplifies in that it assumes that each of the M tuples have a property, *address*, that reveals the address it hashed to. We defined the tuples as *(song id, absolute time of anchor point*. Extending the tuple with the address information needed in this algorithm will not affect the running time.

5.3 Performance

The running time of this algorithm is split up into two main parts. One for computing the inverted index of the music database and one for fingerprinting a snippet and searching for a match.

Pre-computing the database

First, we fingerprint our database using the dejavu fingerprinting algorithm. Since the morphology mask applied over S has a constant size m_s and each entry in S in turn investigates its neighbourhood, we make $O(N)$ operations. Next we do an element-wise

comparison of S and the matrix of the maximum filter which is also $O(N)$. Lastly, we remove elements that are below an amplitude threshold in linear time as well, thus yielding $O(N)$ for fingerprinting a song with N time bins. Substituting N for the total number of time bins in the database, fingerprinting the music database becomes $O(N_T)$

Secondly, we generate the addresses for all fingerprints in the database. The number of addresses generated for a fingerprint is roughly equal to the number of peak frequencies in the fingerprint times the size of the target zones (t_s) as we generate one address per point in the target zones. Since the number of peak frequencies is $O(N)$ (there cannot be more than a constant number of 1's per time bin when F is constant), the number of addresses generated for one fingerprint is $O(N \cdot t_s)$ and $O(N_T \cdot t_s)$ for generating addresses for all fingerprints in the database.

Pre-computing the database then becomes $O(N_T) + O(N_T \cdot t_s) = O(N_T \cdot t_s)$.

Searching for a match
When searching for a match we have to:

- Fingerprint the snippet
- Generate addresses for the snippet
- Search the pre-computed database for potential candidates
- Find the candidate with the best temporal alignment

The running time of the first two steps can be seen from the above as being $O(N_s)$ for fingerprinting and $O(N_s \cdot t_s)$ for generating addresses.

Referring to the pseudocode for algorithm 4, searching for candidates is dependent on the number of addresses and tuples that are found at the addresses of the snippet. The total number of addresses is $O(N_s \cdot t_s)$. Reusing the notation and denoting this number of tuples returned M, we get that finding potential candidates is $O(M + N_s \cdot t_s)$.

Finding the candidate with the best temporal alignment is done using algorithm 5. Line 4 loops the number tuple

s defined as M as found in 4. For all tuples belonging to the earlier found candidates , we at line 6 iterate the snippet-tuples originating from the same address in the snippet. There can be at most at most $O(N_s \cdot t_s)$ snippet tuples. The running time for finding the candidate with the best temporal alignment is thus $O(M \cdot N_s \cdot t_s)$.

Thus, the total running time for searching for a match is $O(N_s \cdot t_s) + O(M + N_s \cdot t_s) + O(M \cdot N_s \cdot t_s) = O(M \cdot N_s \cdot t_s)$

Space
In the case of target zones, we only have to store the inverted index as the database. The number of addresses in the hash-table varies, as multiple elements can hash to the same address, but the number of values associated is roughly the number of peak frequencies times the size of the target zone which is $O(N_T \cdot t_s)$.

6 Empirical studies

The objective of this empirical study is to measure the algorithms in terms of their efficiency with respect to the performance metrics defined in the beginning of this report. It was decided in the initial phase of the project to focus on implementing in-memory solutions as opposed to setting up a database. This implies some limitations on the size of the music databases.

On average, a high quality compressed .m4a song uses around 8 mega bytes of storage for 4 minutes of playtime with a 44.1 kHz sample rate, compared to around 90 mb of that same file uncompressed to the WAV-format. As the implementations presented in this paper are dependent on the full uncompressed data to allow for snippets to come from any part of the song, the size of the in-memory database is limited to 20-30 songs requiring 1.8 - 2.7 gB of raw data in memory before filtering and manipulation. Though this sample size is very small compared to an industrial-scale solution like Shazam with millions of songs [1], it still provides an opportunity to investigate performance from a practical perspective.

6.1 Experimental setup

To enable reproducibility of results, we state here the experimental setup that was used for implementation and measurement hereof:

- Language: Python 3.8.x 64-bit
- Machine: macOS 64-bit / RAM: 8GB 2133 MHz / CPU: 2.3GHz Intel Core i7
- Song format: .wav
- Adding noise to songs: The *Audiosegment*-library
- Background noise for testing: coffeehouse sample [2]
- Number of songs used: 20-30
- Sample rate: $44,100$

The creation of spectrograms was performed with the fast Fourier transformation (fft) method from the *Scipy* python library with the following parameters: *window size* = 4096, *noverlap* $= 4096 \cdot 0.5$, *window* = *hamming*.

These parameters of the fft was chosen on the basis relevant music theory presented in similar projects [3] [4]. The resulting frequency resolution is $\frac{\textit{sample rate}}{\textit{window size}} = \frac{44,100}{4096} = 10.77$ Hz and a time resolution of 0.1, meaning that changes can be detected per 0.1 seconds of music. These values control a trade off between the ability to detect changes in fast paced music and identification of specific frequencies.

6.2 Comparing filtering methods

Algorithm 2-4 have a high dependency on the applied fingerprinting method's ability to detect peak frequencies of music and filter out noise. Figure 6.1 and 6.2 seeks to illustrate this performance metric for the presented fingerprinting algorithms. The effectiveness of the binary fingerprints is measured by the jaccard similarity between the columns of

[1] https://www.shazam.com/company

[2] https://www.premiumbeat.com/blog/free-ambient-background-tracks/

the fingerprints, whereas the naive fingerprint is measured by the sum of the absolute difference.

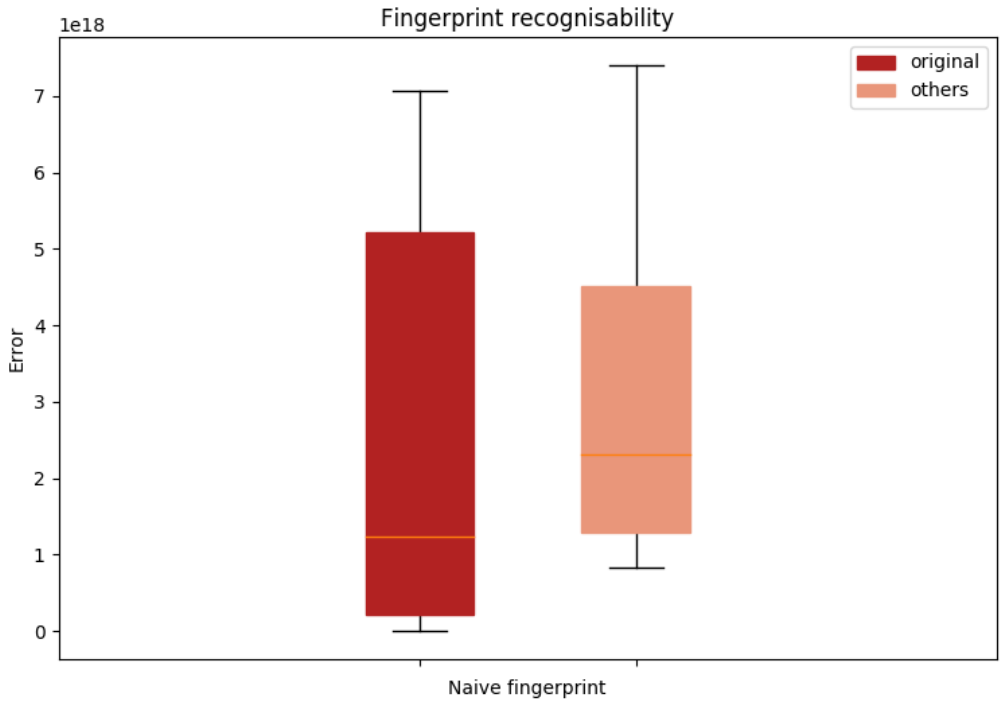


Figure 6.1: Sum of absolute difference for naive fingerprint. *original* is showing the similarity between the snippet containing noise and its true subset. *others* show the average similarity between the snippet other random subsets. Medium noise level (15 songs)

The two box plots in figure 6.1 have a large overlap. This means that the fingerprinting procedure, which is just a direct application of spectrogram, has difficulties in distinguishing between its true subset and other random songs. This merits the use of fingerprinting methods that filters the spectrograms, as it is unlikely that the use of the naive version will produce good results.

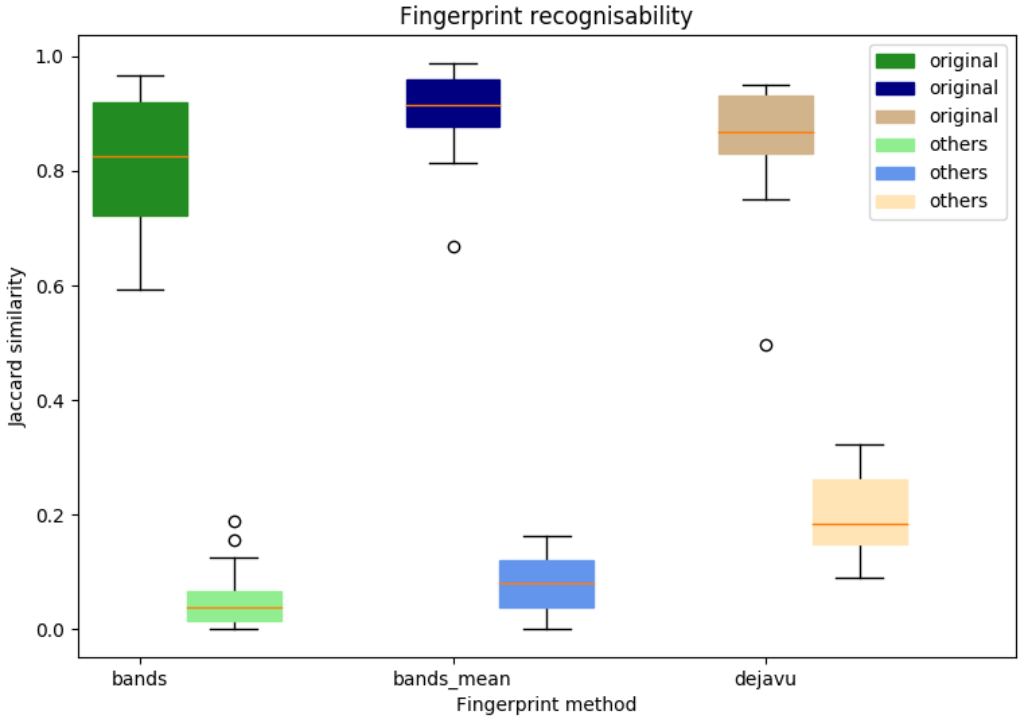


Figure 6.2: Jaccard similarity for the presented fingerprinting algorithms. *original* is showing the similarity between the noisy snippet and its true subset. *others* show the average similarity between the noisy snippet and other random subsets. Medium noise level (10 songs)

From the above plot we see that there are no overlaps between *original* and *others* for any of the fingerprinting algorithms. This indicates that these algorithms filter noise out in a way that empowers the distinction between the original song and random songs. Furthermore, it is seen that the fingerprinting procedures of *bands_mean* and *dejavu* seems to be slightly better at filtering out noise. A possible explanation for this is that the *bands* method is forced to find peak frequencies for every time bin and does not take into account the surrounding environment (neighbouring time bins). This can, especially in low energy sections of songs, lead to noise being picked up as important frequencies.

The number of peak frequencies

As seen in the theoretical analysis of the *target zone* method, the amount of peak frequencies in the fingerprints have an effect on running time and space. Below is a chart visualising the average number of peaks per time bin of the different fingerprinting methods over a sample of 30 songs.

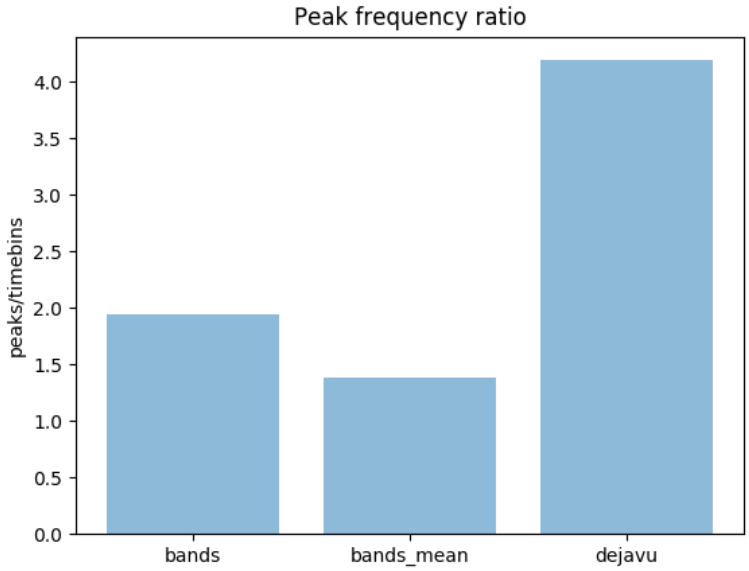


Figure 6.3: Average ratio between number of time bins and number of peak frequencies for a given method. Sample size is 30. Standard deviation is: $[0.25, 0.38, 0.81]$, respectively

Figure 6.3, in conjunction with the fingerprint recognisability in figure 6.2 indicates that *bands_mean* might be the most effective fingerprinting algorithm in that it produces the fewest peak frequencies (speed improvement) and still retains a high degree of recognisability. Therefore, it might be possible that *target zoning* would perform better overall with this fingerprint as it creates a around $\frac{1}{3}$ the amount of peak frequencies than that of *dejavu*.

6.3 Results

We now turn to an empirical study of the characteristics that we defined as being important in a good music indexing algorithm. The **running time** is split up into two parts: 1) Running time for pre-computing the music database and 2) Searching with a snippet against a pre-computed database. **Robustness** in terms of identifying the original song with varying degrees of noise added to the snippet. Finally, **granularity** in terms of how long a snippet is needed for correct identification.

6.3.1 Running time

In figure 6.4, we visualise the time for pre-computing a database of 30 songs.

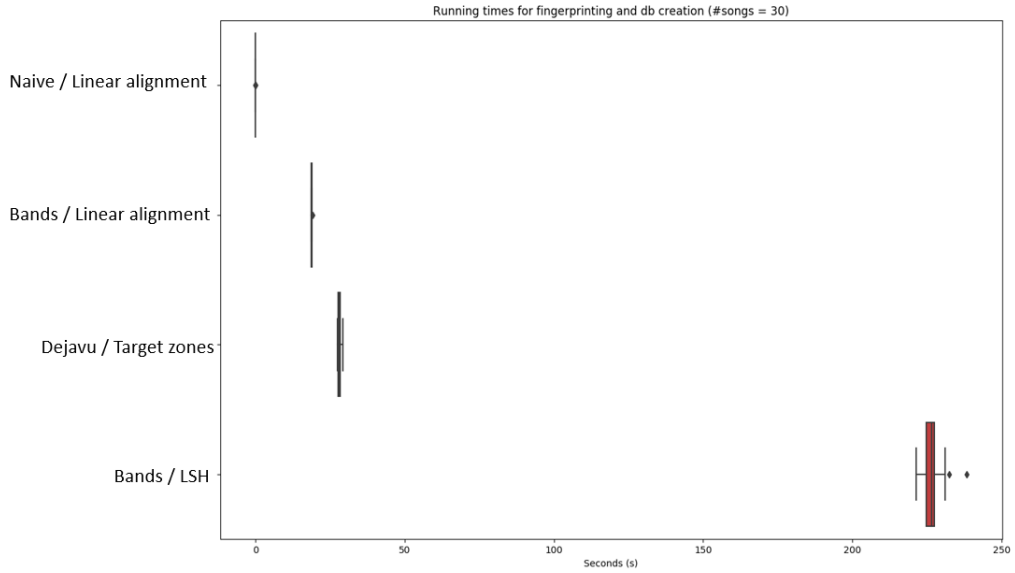


Figure 6.4: Running time for pre-computing the database in memory (30 songs per algorithm)

The ordering of the results from figure 6.4 resembles the theoretical running times which are summarized in table 6.1. The *naive* fingerprinting method is very fast since it only needs to create pointers to each song in the database. Computing the database using *bands* or *dejavu* with target zoning scales linearly with the number of time bins (N_T) in the database. Though they have the same asymptotic running time, target zoning is a constant factor slower due to computing the inverted index, requiring *one* extra iteration over each fingerprint. Pre-computing the database using locality sensitive hashing is naturally slower, as we have to create and hash the signatures from the minhash.

Algorithm	Running time
Naive / Linear alignment	$O(C)$
Bands / Linear alignment	$O(N_T)$
Dejavu / Target zone	$O(N_T)$
Bands / LSH	$O(k \cdot N_T)$

Table 6.1: Summary of theoretical running times for pre-computing the database

Running time for searching

Running time for searching is arguably the most important measure for a real-world application. Searching is comprised of fingerprinting the snippet and searching for a result in the pre-computed database. The result for each of the 4 algorithms are given in figure 6.5 below.

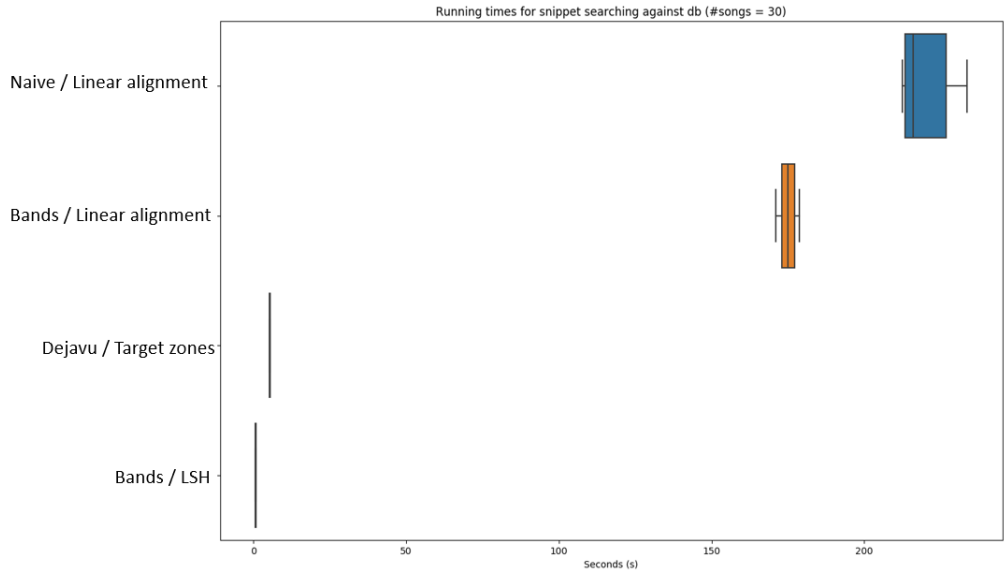


Figure 6.5: Running time for searching with a song snippet against a pre-computed database. Database size is 30 and a snippet from each song is used to search.

As expected, the results are close to flipped when looking at running times for search as opposed to pre-processing the database. The asymptotic running time of the linear alignment procedure using the *naive* and *bands* fingerprint are both dependent on the the total number of time bins in the database. Even though the number arithmetic operations is the same in both cases, the entries of the naive fingerprint have a larger word-size compared to the binary filtered fingerprints, resulting in a larger constant factor on the total search time for *naive*. The running time for search using the target zone approach and *LSH* is a bit different as they are dependent on the distribution of data in the music database.

Algorithm	Running time
Naive / Linear alignment	$O(N_s \cdot N_T - N_s^2)$
Bands / Linear alignment	$O(N_s \cdot N_T - N_s^2)$
Dejavu / Target zone	$O(N_s \cdot t_s \cdot M)$
Bands / LSH	$O(N_s \cdot k + N_s \cdot b \cdot C)$

Table 6.2: Summary of theoretical running times for each of the searching procedures

By rewriting the running time from figure 6.2, we see that:

- Linear alignment algorithms are $O(N_s \cdot N_T - N_2^2) = O(N_s(N_T - N_s))$
- LSH is $O(N_s \cdot k + N_s \cdot b \cdot C) = O(N_s(k + \cdot b \cdot C))$

It is now evident why *LSH* is much faster than linear alignment. N_T constitutes the number of time bins in the music database which is equal to 10 times the number of seconds of music. Assuming an average song duration of *3:30* we have that $N_T = C \cdot 210 \cdot 10$. Ignoring k, b and N_s because they are typically small, we get that LSH is more than 2000 times faster than both linear alignments.

It is also worth mentioning, that linear alignment has to iterate through all alignments in

any case. The term $N_s \cdot b \cdot C$ in the case of *LSH*, however, is only in the case that *all* signatures of the snippet hash to a different value and that *all* songs in database are present at every address, thus making the *expected* difference in running time between the algorithms even greater.

From 6.5 it is also evident that searching using *target zones* is way faster than any linear alignment and almost as fast as *LSH*. From the theoretical analysis, it was argued that the number of tuples M was bounded by the total number of time bins in the database times the target zone size, thus $M = O(N_T \cdot t_s)$. However, as this figure shows, that bound is not very tight and only happens in a fairly unrealistic data distribution setting where every song in the database has the same target zones as that of the snippet.

6.3.2 Space

The space consumption of the different algorithms is in-directly depicted in figure 6.6, which shows a box plot of the memory usage after creating the in-memory database.

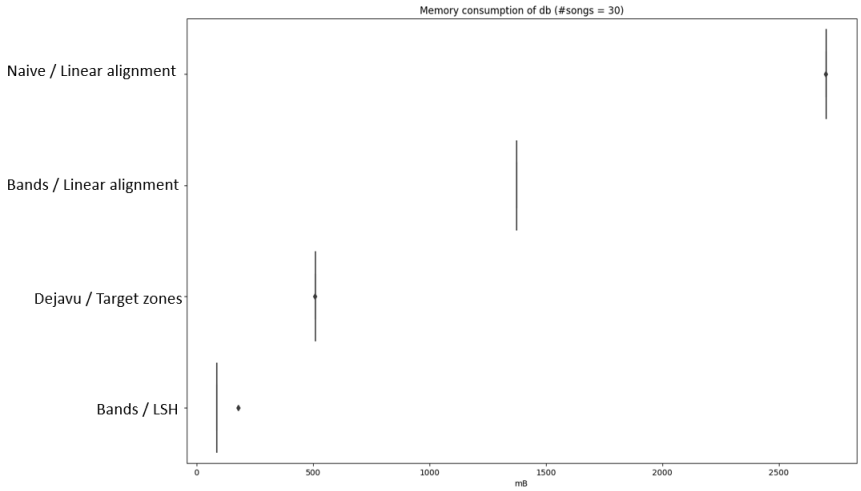


Figure 6.6: Memory usage after db creation of python process. 30 runs pr. algorithm

The difference in word size between the binary matrix of *bands* and 32-bit matrix of the *naive* fingerprint is evident from this plot as the memory usage of the former is almost $\frac{1}{2}$ the size of the latter.

Also, the significant benefit from not storing the complete fingerprints in memory are clear. The database consisting of the inverted index in the case of target zones and the hash tables in the case of *LSH* consumes much less memory and is thus more space efficient.

6.3.3 Robustness

Algorithm robustness describes the ability to identify songs under varying degree of noise. The simulation of noise was implemented by mixing the original song with a file containing background noise from a coffee house shop. In table 6.3, a summary of the results of searching with varying degrees of noise is given. As expected, when the noise increases, it becomes more difficult to identify the correct match. However, some algorithms seem to be more noise robust than others. Specifically, the *target zone* algorithm using the *dejavu* fingerprint performs well even with significant noise. This stems well with the results from the fingerprint recognisability plot in figure 6.2 that indicated that *dejavu* was filtering out noise effectively.

	Noise level			
Algorithm	None	Low	Medium	High
Naive / Linear alignment	1.0	.20	.20	.10
Bands / Linear alignment	1.0	.95	.80	.80
Dejavu / Target zone	1.0	1.0	.95	.95
Bands / LSH	1.0	1.0	.80	.80

Table 6.3: Fraction of correct guesses as a function of the noise level. Sample size is 20 songs

6.3.4 Granularity

Algorithm granularity is tested by letting the algorithms attempt to recognise a snippet from each song in the database with an varying length of the snippet. The results are given in table 6.4.

	Snippet duration (s)			
Algorithm	2	4	6	8
Naive / Linear alignment	.25	.25	.20	.15
Bands / Linear alignment	.95	1.0	1.0	1.0
Dejavu / Target zone	1.0	1.0	1.0	1.0
Bands / LSH	.80	.95	1.0	1.0

Table 6.4: Fraction of correct guesses as a function of snippet length using a medium noise level. Sample size is 20 songs

From table 6.4 it can be seen every algorithm except the naive fingerprinting with linear alignment performs reasonably well across snippet length. There is a slightly counter-intuitive result in that the performance of the *naive* algorithm *decreases* as the snippet becomes longer.

One explanation might be variance in the noise file. Recall that the naive fingerprinting does not filter out noise. There might be sounds in seconds 4 to 8 in the noise file that are more disturbing that in the beginning. It is believed if the test were run over a larger sample size with random overlays of noise, the result would have been different. However, due to processing limitations with an in-memory solution and for the sake of reproducibility, only a single fixed noise clip was used.

7 Future work

Numerous techniques that could improve performance of the algorithms presented in this report has been left out. We here provide suggestions on improvements that could benefit the performance on one or more of the algorithms presented.

Sparse matrix representation
In the case of algorithm 2, linear alignment using the bands fingerprint, the fingerprint is represented by a matrix. Although conceptually appealing, this is not a very efficient way of storing the data when the number of 1's is small compared to size of the matrix. A list of lists, like the one used to store the *dejavu* fingerprint in algorithm 4 would have been a lot more memory efficient and would not impact the running time of linear alignment. As the bands algorithm only allow 6 1's in a given time bin, optimising even further could be to encode the list of indexes of the peak frequencies in a 64-bit integer. where each of the frequency bands has a certain section of bits allocated.

Encoding addresses
This project has not considered any encoding of addresses in the hash tables used in the *LSH* and *target zone* algorithms. This could yield performance boosts in search speeds and storage. This is also the case of the tuples associated with the inverted index in algorithm 4.

Temporality considerations in LSH
Since *LSH* considers the columns of fingerprint as independent sets belonging to a song, ordering is of no concern. Let s and s_r be two songs in the music database such that s_r is s played in the reversed order and consider some snippet that originated from s. In theory, in the current implementation, if a song s returns a match of x similar signatures then s_r will have x similar signatures as well. Although unlikely that s_r exists for any $s \in db$, it is not the intent that s_r should be returned if it exists. The same applies to songs with different orderings of the same peak frequencies, which does not seem completely unlikely, for example with songs using sampling (remixes).

Therefore, the implementation might benefit from a post-processing procedure that takes into account the temporal alignment of the matches, like the one implemented for target zoning in algorithm 4.

Other distance measures for LSH
This project only considered the jaccard similarity derived from the family of minhash functions. Investigation into other LSH-families such as one for the hamming distance might provide good results as well.

Parameter tuning of target zone
There are many parameters of target zoning that can be tuned. This project has used parameters inspired by literature without optimising with regards to increasing the outlined performance. As mentioned in the original Shazam paper[4], the target zone size seems to have an effect on the noise robustness, but this has not been considered in this report.

Finally, an in-memory database solution enforces significant limitations on the ability to test these algorithms on a more realistic dataset (1000+ songs). Firstly, it is very time consuming to compute the database for every test and secondly because storing the database in memory, especially in the case of algorithm 1 and 2 requires better hardware than what the tests were performed on.

8 Conclusion

This report has investigated 4 different solutions for the music indexing problem. The results from the benchmarking of algorithm 1 showed the importance of filtering out noise such that songs still have a high similarity to themselves when they're distorted with noise. Algorithm 2 implemented a solution for filtering spectrograms, which achieved a much higher accuracy than algorithm 1 but was still too slow to be useful. Algorithm 3 and 4 presented clever solutions to avoid the major pitfall of algorithm 1 and 2, namely comparing the snippet exhaustively to every song in the database. They both yielded a large decrease in search times while keeping a high recognition rate. We found that by and large, the cost of improving search times is added time for pre-processing of the music database. However, in real-world applications, a fast search is of utmost importance as that is what the user is experiencing. The empirical study showed that even when the music database is small, comparing a snippet to *every* song in the music database is too time consuming. Therefore, good algorithms for music indexing will likely benefit from processing snippets in a way that allows for a fast identification of a small number of *candidates* that can be further evaluated. Given the slightly higher recognition rate of algorithm 4, algorithm 3 would arguably benefit from post-processing that ensures temporal alignment.

Bibliography

[1] M Balducci et al. “Benchmarking of FFT algorithms”. eng. In: *Ieee Southeastcon ’97 - Engineering the New Century, Proceedings* (1996), pp. 328–330. doi: 10.1109/SECON.1997.598704.

[2] Avery Li-chun Wang and Th Floor Block F. “An industrial-strength audio search algorithm”. eng. In: (2012).

[3] Will Drevo. *Dejavu: Audio Fingerprinting with Python and Numpy*. Nov. 2013. url: https://github.com/worldveil/dejavu.

[4] mawata. *How does Shazam work*. 2013. url: http://coding-geek.com/how-shazam-works/ (visited on 05/20/2020).

[5] Jaap Haitsma and Ton Kalker. “A Highly Robust Audio Fingerprinting System”. In: *ISMIR*. 2002.

[6] Dr. Ramesh Yerraballi. “Multimedia Systems Concepts Standards and Practice M4L4 Audio Compression”. University Lecture. 2004.

[7] Nando de Freitas. “Intermediate Algorithm Design and Analysis”. University Lecture. 2003.

[8] Jeff Ullman Jure Leskovec Anand Rajaraman. *Mining of Massive Datasets*. 2014. isbn: 9781107077232.

[9] Ajit Mahapatra and Sitanath Biswas. “Inverted indexes: Types and techniques”. In: *International Journal of Computer Science Issues* 8 (July 2011).

A Appendix

A.1 Project Description

Title:	Algorithms For Music Indexing
Type:	BSc
Students:	Alexander Bilton (s165635) and Gustav Hartz (s174315)
Advisors:	Philip Bille and Inge Li Gørtz
ECTS:	20
Evaluation:	7-point-scale
Period:	04.02.2020 - 19.06.2020

Project Description
The topic of this Bachelor thesis is algorithms for music indexing. The goal is to investigate and implement existing algorithms for music indexing as well as to design, develop and implement new solutions.

Teaching Goals

- Survey existing algorithms for music indexing
- Implement and compare existing state-of-the art algorithms for music indexing
- Design new algorithms based on experiences with existing state-of-the-art algorithms.
- Implement a prototype of the new algorithms.
- Analyze and evaluate the efficiency of the solutions from a theoretical and practical perspective.
- Document key relevant aspects of the work in a concise manner.

Technical
University of
Denmark

Richard Petersens Plads, Building 324
2800 Kgs. Lyngby
Tlf. 4525 1700

https://www.compute.dtu.dk