FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Real-time audio fingerprinting for advertising detection in streaming broadcast content

Eugénio Bettencourt Carvalhido

Mestrado Integrado em Engenharia Eletrotécnica e de Computadores

Supervisor in FEUP: Maria Teresa Andrade, PhD

Supervisor in INESC TEC: Matthew Davies, PhD

July 27, 2018

Resumo

Os conteúdos televisivos são transmitidos para vários países diferentes ao mesmo tempo. Esses conteúdos podem ser eventos desportivos ou eventos políticos ou religiosos importantes. Na televisão, é prática comum mostrar anúncios entre períodos de conteúdo programado. Quando o mesmo conteúdo está a ser assistido em vários países, idealmente, os anúncios devem ser adaptados de país para país, visando produtos ou serviços locais e de acordo com características específicas do país, incluindo o idioma. Enquanto que este objetivo pode ser alcançado quando a transmissão consiste apenas em conteúdo pré-gravado, o seu cumprimento para eventos ao vivo representa desafios consideráveis. Em particular, devido a restrições de tempo, em transmissões ao vivo surge um problema específico: *como detectar automática e efetivamente e substituir anúncios específicos da região?* A abordagem principal para resolver esse problema é detectar os limites entre o conteúdo da transmissão programada e regiões que contêm publicidades. Num cenário em tempo real, essa deteção precisa de ser realizada automaticamente, aplicando algoritmos de software que analisem o conteúdo *on-the-fly* e extraiam características que possam permitir a detecção automática dos limites entre o conteúdo agendado e conteúdo publicitário. Num cenário em tempo real, que impõe restrições rigorosas de tempo para o processamento do conteúdo, uma solução viável seria explorar o conceito de audio fingerprinting que é aplicado na área da detecção de temas musicais. Essa solução exige que os anúncios inseridos no conteúdo da transmissão sejam conhecidos antecipadamente.

Tomando como referência os métodos do estado de arte relativos a audio fingerprinting, o objetivo principal desta dissertação é reformulá-los de modo a permitir a operação em tempo real para streaming de conteúdo publicitário, que é substituir a identificação de temas musicais por publicidades. A principal contribuição deste trabalho é a incorporação de um tipo diferente de modelagem das features, agrupando-as em quartetos - ou quads. Isso é feito teorizando que as impressões digitais podem ser únicas e, portanto, a procura e a identificação podem ser mais rápidas, tornando possível o funcionamento em tempo real.

O trabalho desenvolvido nesta dissertação pode ser usado como ponto de partida para outros sistemas de maior escala, por exemplo, um que implemente a comutação específica de conteúdo comercial em tempo real.

Devido às diferenças que cada canal de televisão apresenta relativamente à quantidade de publicidades presentes num determinado momento, o *dataset* foi feito a partir de 500 *spots* publicitários. Antes de usar a detecção em tempo real, a base de dados deve ser criada ou atualizada, e deve incluir todas as publicidades que possam aparecer.

A avaliação do sistema apresentado neste trabalho contempla dois objetivos. O primeiro é simplesmente verificar que os quads acima mencionados são realmente únicos na base de dados, validando essa suposição, e o segundo é realizar testes em cenários reais, identificando corretamente blocos de conteúdo publicitário ou de programação sem introduzir atrasos no processamento. Os resultados mostraram que ambos os objetivos foram cumpridos.

Abstract

It is now commonplace for television content to be broadcast to multiple different countries at the same time. This broadcast content could be a sports event, or some major political or religious event. In television, it is common practice to air advertisements between periods of scheduled content. When the same content is being watched in multiple countries, ideally, the advertisements should be adapted from country to country, targeting local products or services and according to specific characteristics of the country, including the language. Whereas this objective may be achieved when the broadcast consists only of pre-recorded content, its fulfillment for live events poses considerable challenges. In particular, due to time constraints, in live broadcasts a specific problem arises: *how to automatically and effectively detect and replace region-specific advertisements?* The main approach to address this problem is to detect the boundaries between scheduled broadcast content and regions containing commercials. In a real-time scenario, this needs to be done automatically, applying software algorithms that analyse the content on-the-fly and extract characteristics that may allow the automatic detection of the boundaries between scheduled content and commercials. In such a real-time scenario, which imposes strict time constraints for processing the content, one feasible solution would be to borrow the concept of audio fingerprinting from the music application domain. This solution requires that the advertisements which are inserted originally in the broadcast content are known in advance.

Taking the state of the art audio fingerprinting methods as a baseline, the main goal of this dissertation is to reformulate them in order to allow operation in real-time for streaming broadcast advertising content, that is to replace the idea of music identification with advertisement identification. The main contribution of this work is the incorporation of a different kind of feature modelling, by grouping the features into quartets - or quads. This is done theorizing that the fingerprints can be unique and hence the search and retrieval can be faster, making it possible to work in real-time.

The work developed in this dissertation could be used as a starting point for other larger scale systems, for example one which implements real-time region-specific switching of commercial content.

Due to the differences in each tv channel regarding the amount of commercials present at a given time, the dataset was built with 500 commercials. Before using the real-time detection, the database was assembled, and it includes all the commercials that could appear.

The evaluation of the system presented in this work addresses two objectives. The first one is to simply verify that the above-mentioned quads are indeed unique in a database, validating that assumption, and the second one is to perform tests in real-case scenarios, correctly identifying blocks of commercials or programming without introducing delays in the processing. The results showed that both objectives were accomplished.

Agradecimentos

Primeiramente, gostaria de dar um agradecimento especial aos meus pais que sempre me motivaram, encorajaram e acreditaram em mim. Se me felicitam nos melhores momentos, também me encorajam nos piores, e sei que poriam as mãos no fogo por mim. Não seria a pessoa que sou hoje se não fosse pelos enormes pais que tenho. Ao meu irmão mais novo, que nunca desistiu do seu sonho de estudar saxofone jazz e que o conseguiu atingir apesar dos percalços. É uma inspiração para mim e sei que ambos chegaremos longe. Ao meu avô Emílio por desde há uns anos me ter incutido o gosto pela eletrotecnia e telecomunicações. Embora as suas histórias sejam compridas, são também edificantes, repletas de conhecimento e fazem-me querer chegar mais longe. À minha avó Gessy que, para além de cuidar piamente de mim (e de todos) nas horas de fome, sempre me disse que eu era capaz. Se lhe dissesse que queria ser presidente dos Estados Unidos, seguramente dir-me-ia que era capaz. À minha avó Lila por também me transmitir confiança e apreço. Provavelmente a avó mais malandra do mundo, se não fosse por ela eu nunca iria saber que estou cada vez mais bonito e jeitoso.

De seguida, um forte agradecimento aos orientadores desta dissertação, Professor Matthew Davies e Professora Maria Teresa Andrade. A Maria Andrade pela confiança depositada em mim. A Matthew Davies por todo o apoio, interesse, proximidade e amizade. Devido à natureza da dissertação, a maior parte do contacto foi feito com o Matthew, e só desejo a qualquer aluno que o possa ter como orientador.

A todos os elementos do Sound and Music Computing Group, que me inspiraram com os seus trabalhos e que deram indicações quando lhes apresentei este mesmo projecto, ainda numa fase intermédia.

Um obrigado especial também a dois colegas do INESC TEC: António Ramires, que já era um bom amigo e colega meu da FEUP, e ao Américo Pereira, que tive o gosto de conhecer durante esta dissertação. Ambos foram muito úteis, quer com a recolha do dataset e insights sobre os problemas que atravessei, quer com momentos de ócio na cantina do INESC TEC ou a tomar café ao sol.

Obrigado também a toda a família Kickbox e CrossfitBGT da Legion Combat. Se por vezes chegava mais cansado ou pensativo aos treinos, era num ápice que se me passava tudo. Corpo são, mente sã, e boas ideias surgiram no rescaldo de uma sessão de exercício físico e boa camaradagem.

Ao pessoal dos OhSeeDisco pelos momentos de creatividade musical e descontração acompanhados de umas cervejas fresquinhas, e por terem aguentado com o meu horário mais limitado nesta última fase da dissertação.

A todos os outros familiares, colegas, professores e amigos que me apoiaram ao longo da vida.

Eugénio Carvalhido

“All you touch and all you see is all your life will ever be.”

Pink Floyd in "Breathe"

Contents

List of Figures

List of Tables

Abbreviations

DB	Database
FFT	Fast Fourier Transform

Chapter 1

Introduction

1.1 Context

With the advances in audio fingerprinting [3], more applications started to appear related to audio recognition. Numerous systems that are capable of identifying songs have been developed, and they work very well. However, such technology is not typically applied in real-time detections, which is the case of this dissertation. Also, they address songs, and not commercials. The work here is an attempt at using such existing technologies, but in a different setup, and seeing if a real-time commercial detection system can be successfully deployed.

This dissertation was conducted in collaboration with an ongoing industrial research project CloudSetup[1] at INESC TEC. Together with a set of techniques related to video processing, an offline audio-only approach has been developed which uses heuristics related to boundary silences and local and long-term structure to identify the starting and ending points of regions of commercials [5]. Taking a different perspective, the aim of this dissertation is to detect such starting and ending points of regions of commercials, but using audio fingerprinting instead.

This dissertation is a result of the project MOG CLOUD SETUP - No17561, supported by Norte Portugal Regional Operational Programme (NORTE 2020), under the PORTUGAL 2020 Partnership Agreement, through the European Regional Development Fund (ERDF).

It is also a result of the project TEC4Growth[2] – Pervasive Intelligence, Enhancers and Proofs of Concept with Industrial Impact/NORTE-01- 0145-FEDER-000020" financed by the North Portugal Regional Operational Programme (NORTE 2020), under the PORTUGAL 2020 Partnership Agreement, and through the European Regional Development Fund (ERDF).

1.2 Objectives

The objectives defined for this project are the following:

[1]www.mog-technologies.com/financed-projects/norte2020/

[2]foureyes.inesctec.pt/

- To compile a dataset of advertising content across a range of international television channels.
- To devise an audio fingerprinting method that generates unique fingerprints, aiming for a faster matching process.
- To develop a real-time fingerprint detection method able to operate for streaming broadcast content.
- To conduct a detailed evaluation which explores the trade-off between the speed and accuracy of advertisement detection via fingerprinting.
- To actively collaborate and integrate with the other members of the research team at INESC TEC and to contribute results to the CloudSetup project

1.3 Motivation

On a personal level, as a musician and music producer, working with audio has always been one of my goals. In the first semester of the present academic year, I underwent an internship on facial recognition, where I gained experience in fingerprinting, feature extraction and image processing. Therefore, both from the perspective of the actual programming tools to be used and research methods, I believe this dissertation project with its strong connection to audio-visual content will be an excellent combination of my personal interest in audio and current experience in image and signal processing.

As for the problem itself, there is interest in its resolution. In an era where we have the technology to automatize tasks, continuing to manually identify and replace region-specific commercials seems like an obsolete and unneeded task. This dissertation is motivated by this, and intends to provide a precise and effective way of identifying the boundaries between scheduled broadcast content and commercials. Others might use this application in order to implement the actual commercial replacement.

1.4 Dissertation Structure

Besides the introduction, this report contains five chapters. In chapter 2, the state of the art is described and the evolution of the work in this area is presented. In chapter 3, the approach and problem characterization are detailed. In chapter 4, the implementation is exhibited, as well as the system operations. In chapter 5 the data preparation is described and the system evaluation is presented. Chapter 6 is dedicated to conclusions.

Chapter 2

Background and State of the Art

Here, the research about the audio fingerprinting state of the art is presented. It was decided to also include computer vision techniques because it seemed like another way to look at the audio, which could complement the fingerprinting. Firstly there is some analysis on how to convert an audio signal to an image that can be analysed by the computer, and how to extract its features and match them. Afterwards we go into more detail on the audio fingerprinting state of the art, first with the established principles and components of such systems that are common to most approaches, and then providing disclosure on each of them, referring different existing ways to tackle the problem. Then, the state of the art commercial detection based on audio fingerprinting is also reviewed. A brief summary is presented after, concluding this section and extracting the main points to consider while doing the dissertation and developing the application.

2.1 Computer Vision

Contrary to modern audio and music recognition systems, Computer Vision has been a popular research topic for a long time, beginning in the 1960s when artificial intelligence research was just starting [6]. Even though most researchers tend to develop independently and look for new algorithms to implement audio fingerprints, some researchers have begun using algorithms traditionally developed for image recognition in order to recognize music [7]. This is still a relatively unexplored approach to audio recognition but one with a lot of potential, especially if there are noticeable peaks in the spectrogram.

2.1.1 From Audio to an Image

The first problem that arises is how to convert the one dimensional audio signal into a two dimensional image that can be "seen" by the computer. For this, a time-frequency representation can be used, for example the spectrogram, when analysing the audio. A standard audio signal waveform and its corresponding spectrogram are shown in Figure 2.1. The former only shows the amplitude of the song, which is great for identifying the points with the highest amplitude in an audio signal, but it doesn't provide as much visual information as the latter, which plots the amplitude of particular frequencies across time. The spectrogram has time on the x-axis, frequency on the y-axis, and the amount of energy at a specific frequency as a color scale in that same point. This allows the user to see not only the overall amplitude, but also the range of frequencies that are more prominent. To accomplish this task, a "Short-time Fourier Transform" can be used [8] which takes milliseconds of an audio file at a time and gets the sine wave functions with the highest amplitude at each "window" of time, in order to find the frequencies present at that moment.

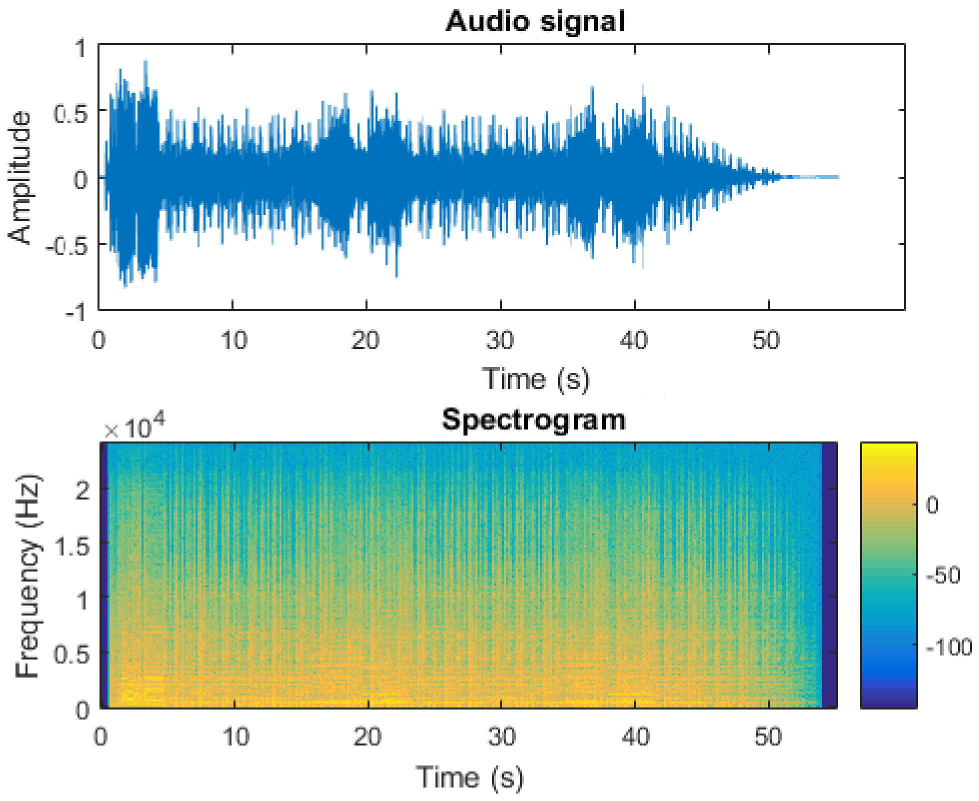


Figure 2.1: Audio signal and its corresponding spectrogram.

2.1.2 Detecting Features in Spectrograms

After obtaining the spectrogram, the next step would be to generate features that the computer could use to perform the matching. To do this, there are various options. One uses a custom feature detection algorithm based on Adaboost and Viola-Jones features [7]. It uses a class of Haar wavelet-like filters introduced by Viola and Jones for face detection [9], after treating the spectrogram as a grayscale image, as seen in Figure 2.2. From the various possible filters, M discriminative filters and corresponding thresholds are selected to generate an M-bit vector that represents overlapping segments of audio. This vector, called the descriptor, can be quickly computed using integral images.

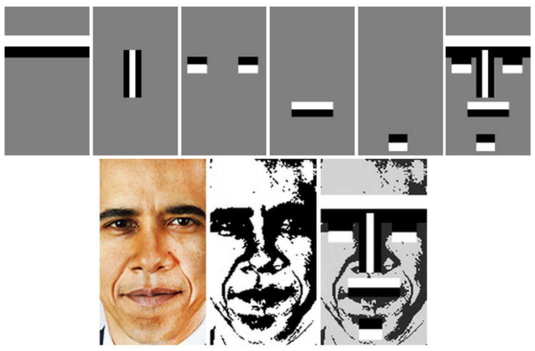


Figure 2.2: How the Haar filter are applied into a picture to detect zones. Original figure taken from [1].

Other possible methods used in general computer vision problems are SIFT [10] and SURF [11]. Both methods are in the category of "blob detection algorithms" that use a "Laplacian of Gaussian" measurement to find the center of blobs in images. The Laplacian is found by adding the second derivative in the x direction to the second derivative in the y direction for each point on a gray scale image. It has been found that the center of a "blob" in an image generally corresponds to relative maxima in the Laplacian. This method only finds extremely specific and uniform blobs in an original image. However, if a Gaussian blur is first applied to an image before finding the maxima of the Laplacian, it is possible to find larger and more generic blobs. By finding the Laplacian maxima at differing amounts of Gaussian blur, it is possible to calculate a set of key points that represent these blobs and store them in a way that can be compared to other images. SIFT and SURF then calculate edge strengths and orientations in a neighborhood around each key point and store this information in a feature vector that characterizes the local image structure near the key point.

2.1.3 Feature Matching

In [7], the signatures are hashed into a standard hash table, keyed by appropriate M-bit descriptors which are within a Hamming distance of 2 from the given query to be near-neighbors. Once all of the near neighbors have been found, the song that best matches the set of descriptors in the query needs to be identified. The verification is done by employing a form of geometric verification that is similar to that used in object recognition using local features [12]. For each candidate song, Ke et al. [7] determine whether the matched descriptors are consistent over time. For this, RANSAC [13] can be used to iterate through candidate time alignments and use the EM score, the likelihood of the query signature being generated by the same original audio as the candidate signature, as the distance metric. For the alignment models, we can assume that the query can be aligned to the original once a single parameter (temporal offset) has been determined. In this case, the minimal set is a single pair of matching descriptors. Once all of the retrieved candidates have been aligned, the audio with the best EM score is chosen, assuming that it passes a minimum threshold.

As far as comparing the vectors obtained from SIFT or SURF methods, one could use open source libraries, for instance the ones from OpenCV, more specifically the Brute Force or FLANN matchers, based on [14]. The former does an exhaustive search and is guaranteed to find the best neighbour, while the latter builds an efficient data structure that will be used to search for an approximate neighbour. The real benefit of the FLANN matcher is seen with large data sets. The resulting matching vectors from both SIFT and SURF are seen in Figure 2.3.

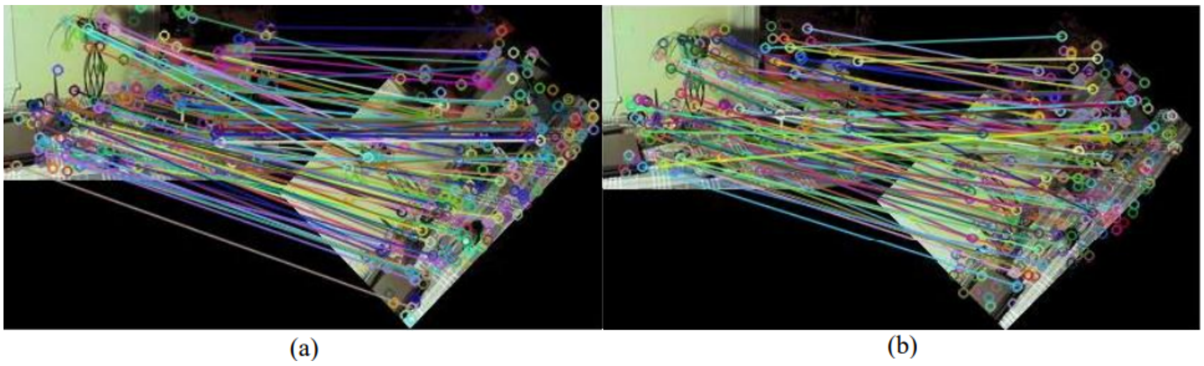


Figure 2.3: Example of resulting matches in both SIFT (a) and SURF (b) methods, on a rotated image. Original figure taken from [2].

2.2 Audio Fingerprinting

An audio fingerprint is a compact content-based signature that summarizes an audio recording [3], which has the capability to identify audio. It extracts relevant acoustic characteristics of a piece of audio content and stores them in a database. When presented with an unidentified piece of audio content, characteristics of that piece are calculated and matched against those stored in the database. Using fingerprints and matching algorithms, distorted versions of a single recording can be identified as the same music title.

2.2.1 Principles

Audio Fingerprinting has some standard principles behind its functioning, which may refer to requirements that the fingerprint or the fingerprinting system need to follow. These may depend on the application but are useful in order to evaluate and compare different technologies.

Requirements

The audio fingerprint application needs, in general, to follow these requirements, depending on the application [3], [15]:

- **Accuracy:** The number of correct, missed and incorrect identifications .
- **Reliability:** The ability to avoid false positives.

- **Robustness:** The ability to accurately identify an item, regardless of the level of compression, distortion, interference or other sources of degradation.
- **Granularity:** The ability to identify whole audio signals from short excerpts.
- **Security:** The vulnerability of the solution to intentional manipulations that are designed to fool the fingerprint identification algorithm.
- **Versatility:** The ability to identify different audio formats, and to use the same database for different applications.
- **Scalability:** The capability of performing in large databases.
- **Complexity:** The general computational costs of the fingerprint extraction, its size, the complexity of the search and matching, the cost of adding new items to the database, etc.
- **Fragility:** The ability to detect changes in the content, which goes against the robustness requirement but is needed for plagiarism identification.

As for the objectives of this dissertation, not all of these requirements are needed. Since the input stream contains the original commercials, without any kind of degradation, content robustness is assured. On the same train of thought, the audio is not accessible by people besides the company using it, and there is no obvious interest from a possible third party to manipulate this application in its favor, so the system should be secure on its own. The system will also use standard broadcast media formats, so this application doesn't need to be versatile in this aspect. Regarding the scalability requirement, it is not expected in the INESC TEC project to deal with a large database, like a music database, as it is not plausible that a TV channel will have even a thousand different commercials at a certain given time. However, this is not the focus of this dissertation. As mentioned before, the content analyzed will not have any kind of changes, so the fragility requirement is also passable. This leaves us to focus on the accuracy, reliability and complexity requirements. We need an accurate application that will correctly identify or dismiss any audio, in the shortest possible time (for this we cannot have a high complexity, because this application needs to be used in real-time).

Regarding the requirements of the fingerprint itself, it generally needs to be, as mentioned in [3]:

- A perceptual representation of the recording. It must retain the maximum of acoustically relevant information and should allow the discrimination over a large number of fingerprints. This may be conflicting with other requirements, such as complexity and robustness.
- Invariant to distortions. This derives from the robustness requirement. Content-integrity applications, however, relax this constraint for content-preserving distortions in order to detect deliberate manipulations.
- Compact. A small-sized representation is interesting for complexity, as it can speed up the process of comparison and retrieval. An excessively short representation, however, might not be sufficient to discriminate among recordings, thus affecting accuracy, reliability and robustness.

- Easily computable. For complexity reasons, the extraction of the fingerprint should not be excessively time-consuming.

Again, we do not have to care about the fingerprint being invariant to distortions as we do not foresee differences between the commercials in the database and the ones present in the real time audio stream. Also, since the database will be small, the fingerprint does not need to contain a lot of information. Knowing this, we can strive for uniqueness of the fingerprints, which means that the database will not have duplicates. As for compactness, the commercials to be identified will be very or relatively small, so the resulting fingerprints can be compact. With these insights we can conclude that we can strive for a simple and straightforward fingerprint, which will allow the application to perform well in real-time conditions.

Application in Identification of Audio

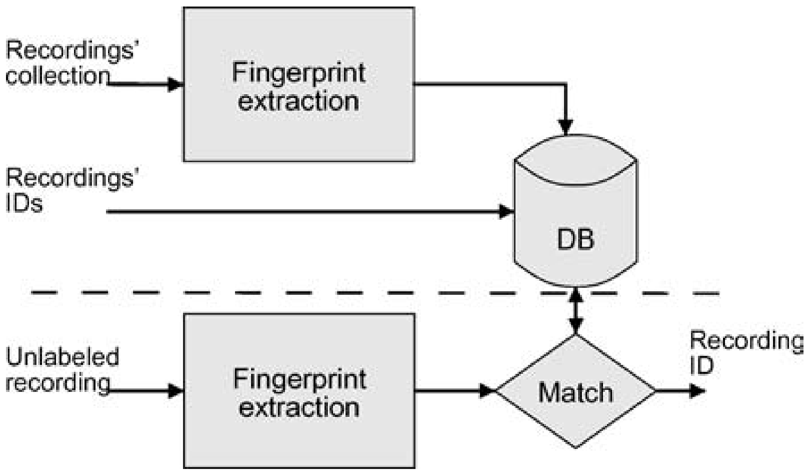


Figure 2.4: Basic architecture of a musical audio identification system. Original Figure taken from [3].

Even though there are different approaches to audio fingerprinting systems, they all follow the same basic architecture [3]. As shown in Figure 2.4, there are two fundamental processes: the fingerprint extraction and the matching algorithm.

The fingerprint extraction consists of a front-end and a fingerprint modeling block, as shown in Figure 2.5. The front-end computes a set of measurements from the signal. The fingerprint model block defines the representation of the final fingerprint. Given an input fingerprint, the matching algorithm searches a database of fingerprints to find the best match. A way of comparing fingerprints, that is a similarity measure, is therefore needed.

2.2.2 The Front-End

The front-end converts an audio signal into a sequence of relevant features to feed the fingerprint model block. When designing the front-end, one has to take into account the requirements for

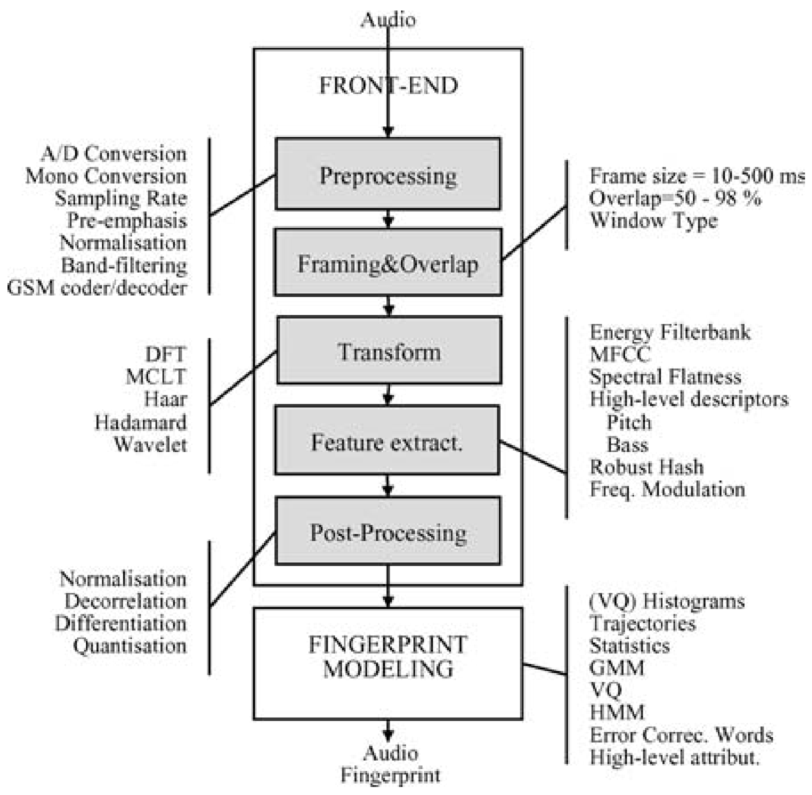


Figure 2.5: Fingerprint extraction framework. Original picture taken from [3].

audio fingerprints and audio fingerprinting systems, previously mentioned in the Principles subsection.

Preprocessing

Since the audio that this will work with is already digitalized, this stage will only convert the signal to mono by averaging the left and right channels of a stereo pair, if necessary.

Framing and Overlap

Framing means dividing the audio signal into frames of equal length by a window function (e.g. Hann or Hamming windows). During this process, a large portion of the audio signal may be suppressed by the window function [16]. since the values near the boundaries of the window function drop by a significal amount. In order to make up for loss of energy, the frames overlap, as shown in Figure 2.6.

Transformation

In this step, the set of frames is transformed into a new set of features, essentially changing its representation. Most solutions choose standard transformation from time domain to frequency domain, like the FFT [17]. There are also some other transformations including the Discrete

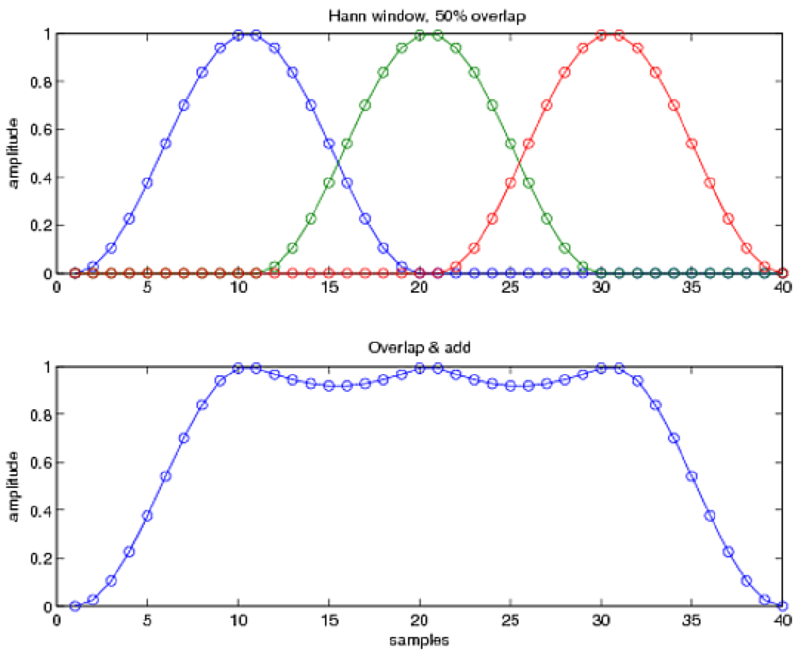


Figure 2.6: Overlapped Hann Windows. Note that the lower values are eliminated by this technique. Original Figure taken from [4].

Cosine Transform [18], the Walsh-Hadamard Transform [19], the Modulated Complex Transform [20], etc.

Feature Extraction

Once a time-frequency representation has been obtained additional transformations are applied in order to generate the final acoustic vectors. After transformation, final acoustic features are extracted from the time-frequency representation. The main purpose is to reduce the dimensionality and increase the robustness to distortions.

One of the most commonly used method in the field of audio recognition is the Mel-frequency cepstral coefficients (MFCC) analysis [21], [22]. It is based on the auditory mechanism of the human ear, and has relatively high recognition and robustness qualities. To get the MFCC features, the spectrum is passed through a set of Mel filters (triangular filters spaced according to the Mel scale), which results in a Mel spectrum. Afterward, this spectrum is subjected to a cepstral analysis in order to obtain its coefficients, which correspond to the features of the analyzed audio frame. Its application in music has been shown in [23].

Another proposed scheme is the Spectral Energy Peak. This method was described for music identification systems in [24] and [25] the latter being known as the system used by Shazam application. Here, time-frequency coordinates of the energy peaks were described as sparse landmark points. Then, by using pairs of landmark points rather than single points, the fingerprints exploits the spectral structure of sound sources. In [26], the method was shown to be intrinsically robust to even high level background noise and can provide discrimination in sound mixtures.

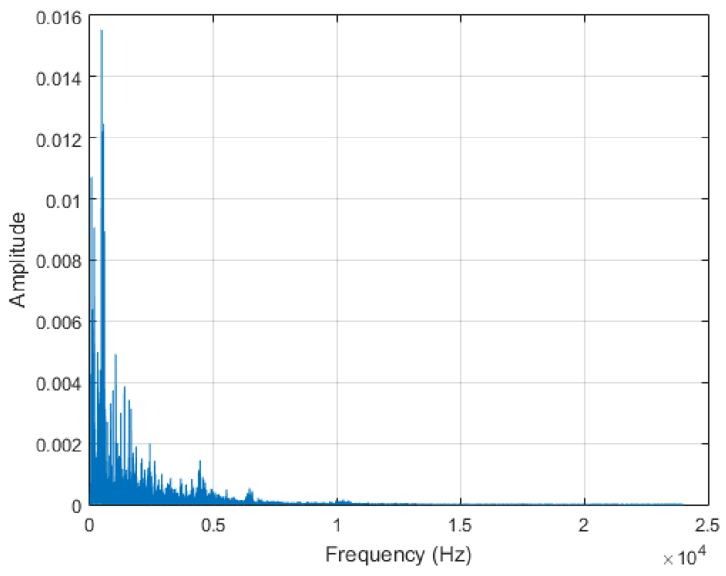


Figure 2.7: FFT of the audio signal in Figure 2.1

Another approach is Spectral Band Energy. This referes to the energy in each time frame and frequency subband range. Haitsma et al. [8] proposed a famous fingerprint where these energies were first computed in a block containing 257 time frames and 33 Bark-scale frequency subbands, then each feature was quantized to a binary value (either 0 or 1) based on its differences compared to neighboring points. Other fingerprinting algorithm exploiting these features were found for instance in [27]. Variances of this subband energy difference features can be found in more recent approaches, as for example in [28].

There is also the Spectral Flatness Measure. Also known as Wiener entropy, relates to the tonality aspect of audio signals and it is therefore often used to distinguish different recordings. It is computed in each time-frequency subband point. A high resulting value indicates the similarity of signal power over all frequencies while a low one means that signal power is concentrated in a relatively small number of frequencies over the full subband. A similar method to this one, which is also a measure of the tonal-like or noise-like characteristic of audio signal, is Spectral Crest Factor. It indicates how extreme the peaks are in a waveform, and can also be exploited as a fingerprint. Both of these methods were found to be the most promising features for audio matching with common distortions in [29] and were both considered in other fingerprinting algorithms [27], [30].

Finally, another popular measure used in audio signal processing is the Spectral Centroid. It indicates where the ”center of mass” of a subband spectrum is. The features extracted with this method are said to be robust over equalization, compression, and noise addition. It was reported in [31] and [30] that fingerprints based in this method offered better audio recognition than MFCC-based fingerprints with 3 to 4 second length audio clips.

Post-Processing

Most of the features described so far are absolute measurements. In order to better characterize temporal variations in the signal, higher order time derivatives are added to the signal model. In [21] and [32], the feature vector is the concatenation of MFCCs, their derivative (delta) and the acceleration (delta-delta), as well as the delta and delta-delta of the energy. Some systems only use the derivative of the features, not the absolute features [27], [33]. Using the derivative of the signal measurements tends to amplify noise [34] but, at the same time, filters the distortions produced in linear time invariant, or slowly varying channels (like an equalization). Cepstrum Mean Normalization (CMN) is used to reduce linear slowly varying channel distortions in [32]. If Euclidean distance is used, mean subtraction and component wise variance normalization are advisable. Some systems compact the feature vector representation using transforms (for example by using Principal Component Analysis [21], [32]). It is quite common to apply a very low resolution quantization to the features: ternary [35] or binary [8]. The purpose of quantization is to gain robustness against distortions [8], normalize [35], ease hardware implementations, reduce the memory requirements and for convenience in subsequent parts of the system. In [20] binary sequences are required to extract error correcting words utilized, and the discretization is designed to increase randomness in order to minimize fingerprint collision probability.

2.2.3 Fingerprinting Models and Searching Methods

Fingerprinting Models

The fingerprint modeling block computes the final fingerprint based on the sequence of feature vectors extracted by the front-end. Every frame generates a feature vector, so the initial sequence of feature vectors is too large to be used as fingerprint directly. Exploiting redundancies of spectral features is useful to further reduce the fingerprint size. A great option in this case is to adapt the feature vectors to a statistical model. Three popular models, namely Gaussian Mixture Model, Hidden Markov Model, and Nonnegative Matrix Factorization are described in this section.

- **Gaussian Mixture Model**
  This model investigated for audio fingerprinting in [30], where spectral feature vectors F_n are modeled as a multidimensional K-state Gaussian mixture with probability density function given by

$$p(F_n) = \sum_{k=1}^{K} \alpha_k N_c(F_n|\mu_k, \Sigma_k)$$

  where α_k, which satisfies $\sum_{k=1}^{K} \alpha_k = 1$, μ_k and Σ_k are the weight, the mean vector and the covariance matrix of the k-th state.
  The model parameters $\theta = \alpha_k, \mu_k, \Sigma_{kk}$ are then estimated in the maximum likelihood sense via the expectation-maximization (EM) algorithm, with the global log-likelihood defined as

$$\mathscr{L}_{ML} = \sum_{n=1}^{N} \log p(F_n|\theta)$$

As a result, the parameters are iteratively updated via two EM steps. In the expectation step, the posterior probability that feature vector F_n is generated from the k-th GMM state is computed. Then, the parameters are updated, in the maximization step.
With this method, N-dimensional feature vectors F_n are characterized by K set of parameters $\{\alpha_k, \mu_k, \Sigma_k\}_{k=1,...,K}$ where K is often very small compared to N. However, since this method does not explicitly model the amplitude variation of sound sources, signals with different amplitude level but similar spectral shape may result in different estimated mean and covariance templates.

- **Hidden Markov Model** Similarly to the previous method, this one also uses a probability density function, which is given by:

$$p(F_n) = \sum_{q1,q2,...,qd} \pi_{q_1} b_{q_1} F(n,1) a_{q_1 q_2} b_{q_2} F(n,2) ... a_{q_{d-1} q_d} b_{q_d} F(n,d)$$

where π_{qi} denotes the probability that qi is the initial state, $a_{q_i q_j}$ is state transition probability, and $b_{q_i}(F_n, i)$ is the probability density function for a given state.
Given a sequence of observations $F_n, n = 1,...,N$ extracted from a labeled audio signal, the model parameters $\theta = \{\pi_{q_i}, a_{q_i q_j}, b_{q_i}\}_{i,j}$ are learned via EM algorithm and stored as a fingerprint.
Cano et al. modeled MFCC feature vectors using this method in [21]. In [36], fingerprints based in this method were shown to achieve high compaction by exploiting structural redundancies on music and to be robust to distortions. Note that when applying these two former methods for the fingerprint design, a captured signal at the user side is considered to be matched with an original signal fingerprinted by the model parameter θ in the database if its corresponding feature vectors $\hat{F}_n$ are most likely generated by θ.

- **Nonnegative Matrix Factorization** This method was applied to the spectral subband energy matrix in [37] and to the MFCC matrix in [38]. In the context of audio fingerprinting, a $d * N$ matrix of the feature vectors $V = [F1,...,F_N]$ is approximated by

$$V = WH$$

where where W and H are non-negative matrices of size $d * Q$ and $Q * N$, respectively, modeling the spectral characteristics of the signal and its temporal activation, and Q is much smaller than N. The model parameters $\theta = \{W, H\}$ can be estimated by minimizing the following cost function:

$$C(\theta) = \sum_{bn} d_{IS}([V]_{b,n} | [WH]_{b,n})$$

where $d_{IS}(x|y) = xy - log(xy) - 1$ is Itakura-Saito divergence, and $[A]_{b,n}$ denotes an entry of matrix A at b-th row and n-th column. In [39], Févotte et al. estimate the parameters by using the following Multiplicative Update rules:

$$H \leftarrow H \frac{W^T((WH)^{\cdot -2})}{W^T(WH)^{\cdot -1}}$$

$$W \leftarrow W \frac{((WH)^{\cdot -2})H^T}{(WH)^{\cdot -1}H^T}$$

where denotes the Hadamard entrywise product, $A^{\cdot p}$ being the matrix with entries $[A]_{ij}^{\cdot p}$, and the division is entrywise. Fingerprints are then generated compactly from the resulting matrix W, which has much smaller size compared to the original feature matrix V.

2.2.3.1 Searching Methods

After fingerprints are extracted and modeled, the search for similar fingerprints is performed on the database for the matching. The similarity is essentially the measure of how alike two fingerprints are, and is described as a distance. Small distance indicates high degree of similarity, and vice versa.

In [3], the requirements for a good searching methods are stated:

- **Fast:** Sequential scanning and similarity calculation can be too slow for huge databases.
- **Correct:** Should return the qualifying objects, without missing any—i.e. low False Rejection Rate (FRR).
- **Memory efficient:** The memory overhead of the search method should be relatively small.
- **Easily updateable:** Insertion, deletion and updating of objects should be easy.

There are many possibilities for the distance chosen. Two of them are especially straightforward and easy to compute. The first and most popular one is the Euclidean distance [22], which is basically the straight-line distance between the points. In [40], the Manhattan distance is considered instead. It assumes that the points are layered on a grid and defines the distance as the sum of the horizontal and vertical components instead of the direct diagonal path. Both of them can be seen in Figure 2.8.

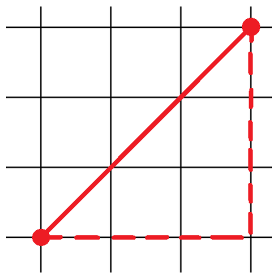


Figure 2.8: Euclidean distance depicted as a regular line, whereas Manhattan distance is shown as the dashed line.

Regarding the search itself, it might not be fast and efficient if the database is big. However, the application devised in this dissertation is not aimed at a database of millions of audio references, as Shazam is for example. In this case each television channel will have a relatively small number of references in the database. In any case, the retrieval has to be as fast as possible, so comparing the fingerprints one by one might still not be the fastest option, as we want a real-time retrieval. A good general strategy to speed up the process is to design an index data structure in order to decrease the number of distance calculations.

To further accelerate the searching procedure, some searching algorithms adopt multi-step searching strategy. In [40], Haitsma et al. design a two-phase search algorithm. Full fingerprint comparisons are only performed when they have been selected by a sub-fingerprint search. Lin et al. [41] propose a matching system consisting of three parts: “atomic” subsequence matching, long subsequence matching and sequence matching.

2.2.3.2 Hypothesis Testing

The final step is to decide whether there is a matching item in the database. If the similarity between the query fingerprint and other reference fingerprints in the database is above a predetermined threshold, the reference item will be returned as the matching result, otherwise the system will think that there is no matching item in the database. Based on the matching results, the performance of an audio fingerprinting system is measured as a fraction of the number of correct match out of all the queries that are used to test. Most systems report this recognition rate as their evaluation results [25], [42], [43], [44].

2.3 Commercial Detection

In March of 2017, after the start of the "Cloud-Setup" project, Facebook was granted a patent for a Commercial Detection based on Audio Fingerprinting [45]. The system they came up with shares the objectives of this dissertation, but using program guide information for a better performance. It also verifies the commercial detection across multiple media streams on different broadcast channels. It can determine whether the same match is observed across multiple broadcast programs and/or multiple broadcast channels over a specific period of time and, based on a predetermined threshold, determine that the media stream contains commercial content. It claims it extracts multiple acoustic features in a frequency domain and a time domain from the audio signal associated with the media stream. These encompass spectral features computed on the magnitude spectrum of the audio signal, Mel-frequency cepstral coefficients of the audio signal, a spectral bandwidth and spectral flatness measure of the audio signal, a spectral fluctuation, extreme value frequencies, and silent frequencies of the audio signal. It then applies a trained feature analysis model to the extracted features, which can be trained using one or more machine learning algorithms to analyze them. Afterwards, a confidence score is assigned by the system to indicate the likelihood that the media stream contains commercial content. Although the specifications are not detailed in the document, they do specify that the window frames are 25ms, 50ms, 100ms, 200ms, (...) and the transform used is the DCT (discrete cosine transform). From their description, it seem that they base their application in the state of the art audio fingerprinting that has been mentioned in this report, confirming that it should be a promising path to follow.

2.4 Summary

As can be seen in this chapter, there are many successful approaches to audio fingerprinting. Apart from the Computer Vision way, they all follow basically the same system structure, each using its own combination of settings and specifications relative to each system block. One can work with many different options for windowing, transforming, extracting and modeling features, and for pre and post-processing. Each combination will output different results and will be better or worse, according to the end application's needs and goals. The existing approaches have been applied in the context of music identification, to be used as a means to counter plagiarism, copyright infringement, or even just for a regular user to identify a song that the user likes but doesn't know. Since the goal of this dissertation is to perform in real-time, the state of the art material will have to undergo some changes in order to work. In any case, there is already a lot of information to pursue the goal of detecting commercial in real-time.

Chapter 3

Approach

In this chapter the definition of the problem to solve is presented, as well as the approach used for the development of the system's components. This device's framework is based on the typical audio recognition system showed in [3], with some other blocks related to the real-time performance. The focus of the approach was creating a model capable of accurately and promptly detect blocks of commercials from a stream.

3.1 Problem Characterization

The problem consists on the creation of an application that detects advertisement blocks in streaming broadcast content based on audio fingerprinting. This application should receive the audio of a streaming broadcast and output whether it detects an advertisement block or not, calculating the detection offset as well as the starting and ending times of that same block. The system should be as fast as possible, while maintaining a good performance.

The applications mentioned in the previous chapter have been shown to be efficient for music detection, but they deal with slightly different conditions. For instance, even though they focus on speed, this speed is relative to the huge databases (millions of songs) that comprise the training data. In this real-time detection, the database is not foreseen to be big, so the search has to detect the advertisement much faster, which also directly impacts the performance. A good way to tackle this problem is to be able to obtain unique fingerprints, which could be done grouping the features. A possible number of features for a group could be four, as explored by Sonnleitner and Widmer [46]. This generates a quartet, or a quad, which is much more specific than a single feature.

By conceptualizing this problem, one can see that detecting and matching advertisements is practically the same as matching songs. The main difference is the requirement to work in a real-time scenario. The problem then becomes applying the audio fingerprinting state of the art in practice for this specific application.

The hypothesized basic architecture for this system can be seen in Figure 3.1

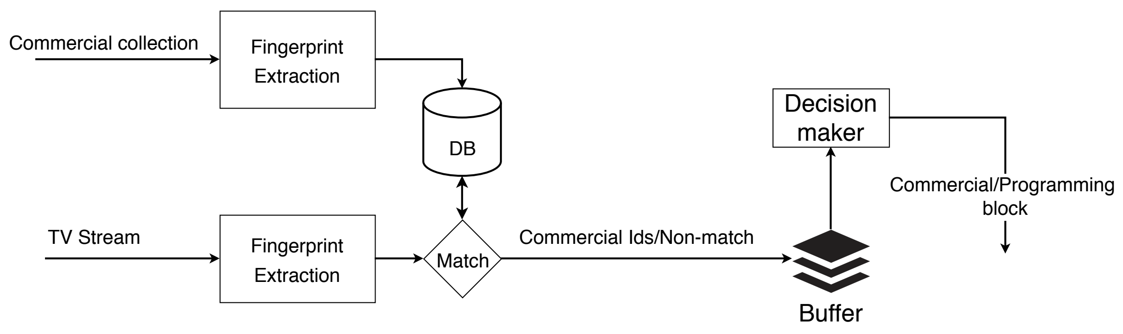


Figure 3.1: Basic architecture for the developed system

3.2 Preprocessing

The audio is converted to mono. Different approaches can be done, for example only analyzing one channel of the audio, but the detection would be compromised if channels suddenly switched. Since the audio format and characteristics are previously stipulated and are expected to be always the same, no further preprocessing is needed.

For this project, a sample rate of 48000Hz was used, which is the one that the data set commercials have, as well as being the one specified in the CloudSetup project.

3.3 Framing, Overlap and Transform

The Fast Fourier Transform was used as the transforming algorithm that will sample the audio signal and divides it into its frequency components. The FFT window was set at 2048 samples and a Hann Window of 50% was applied. Since the Sample Rate is 48000Hz, we then have a time frame, in seconds, of:

$$w = \frac{2048}{48000} = 0.04266(6)\ s. \tag{3.1}$$

If we would get the spectrogram just by calculating the FFT of the audio each time frame, some information would be lost at the limit between each time frame. In order to avoid this and have more analysis points, resulting in smoother results across time, the spectrogram is calculated every half of the time frame, resulting in a hop size, also in seconds, of:

$$hop = \frac{0.04266(6)}{2} = 0.02133(3)\ s. \tag{3.2}$$

This results in a matrix, which contains the values of energy of each frequency bin at a given time. Figure 3.2 illustrates how the spectrogram is obtained through the junction of each FFT step.

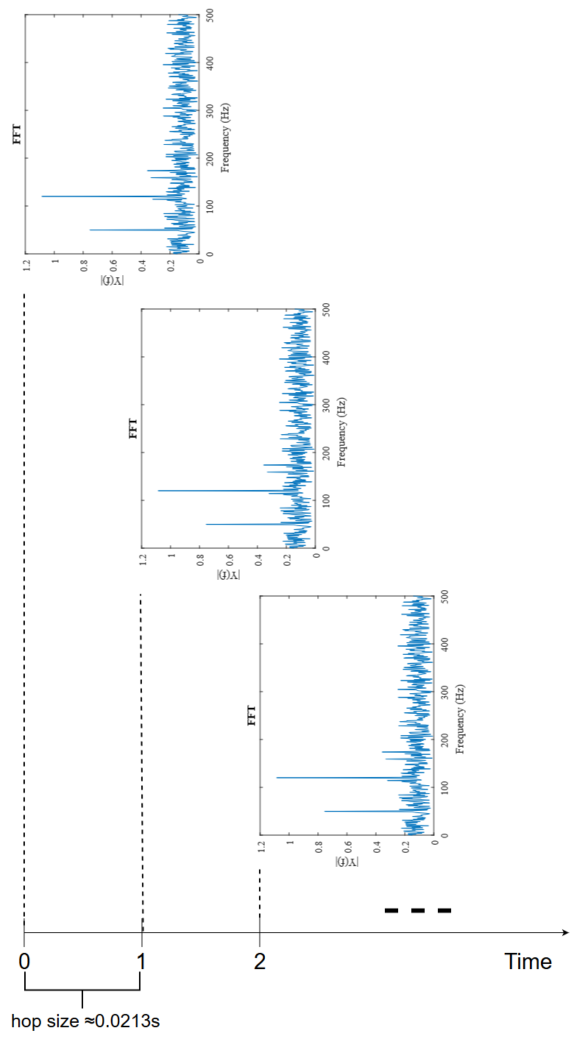


Figure 3.2: How the spectrogram is obtained

3.4 Feature Extraction

The spectrogram obtained previously contains a great number of candidates for the features. So far, all the points are valid, and it is important for the perfomance of the system that we select those which best describe the audio. In order to do this, we search for the local maxima, which are the peaks with the highest energy in a predefined neighbourhood. These peaks are the ones that are the most specific to the audio and the most resistant to distortions or noises.

From this, the result is another matrix which has the same size as the spectrogram. The difference is that the points that were not considered local maxima will not have a value associated. This way we filter out the worst candidates out of the spectrogram. The remaining points, the features, will be used to form quads.

Figure 3.3 illustrates a spectrogram, where the black dots represent its local maxima.

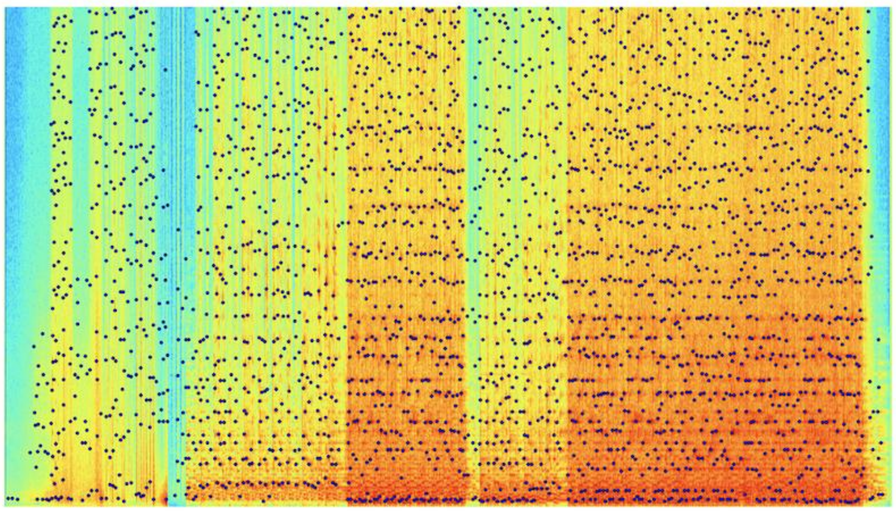


Figure 3.3: Local maxima (represented by black dots) of a spectrogram. The x-axis represents time and the y-axis represents frequency.

3.5 Feature Modeling

The same features might appear in different signals. A good way to avoid this and be able to create fingerprints as unique as possible, would be to group the features. A group of features is much more specific than a feature alone, and instead of searching the database for many consecutive matches, we can just look it up for this specific group. Inspired by the work done in [46], the approach used is grouping the features in quartets (quads). While previously working with pairs of features, as used in [25], this proved to still show redundant results in a real-time scenario, thus needing many consequent landmark matches in order to ensure an accurate result, which would compromise the real-time performance. A quad is a much stronger fingerprint because it is made by four points instead of two, and this greatly minimizes the probability of redundant fingerprints in the database. This way we will need fewer consequent matches to be accurate.

The approach used to assemble the quads is as follows: first, we go through the feature matrix obtained in the previous step. If we find a point in the matrix which isn't a 0, this means that it is a peak. Then we try to find an opposite corner of the quad, which will be located at a limited temporal and frequency distance into the future from this point. This distance was set to between 3 to 8 time and frequency bins from the first candidate. The minimum distance of 3 ensures that the remaining two points can be found, since no two points of a quad should share the same time or frequency bins. This will greatly improve the probability of all the quads being unique. The upper limit of 8 bins in the time and frequency distances is just a ceiling to prevent the system to spend too much time looking for candidates, as 8 bins is enough to form the quads. The actual maximum possible euclidean distance between any two points (A and B) is $d = \sqrt{8^2+8^2} = 8\sqrt{2} \approx 11$ bins.

After having the points A and B, we try to find a third point inside this area which is not zero, and then a fourth point inside the area formed by the first three. When all four points are found, we obtain the quad. Figure 3.4 demonstrates this process, while Figure 3.5 illustrates an example of an extracted quad.

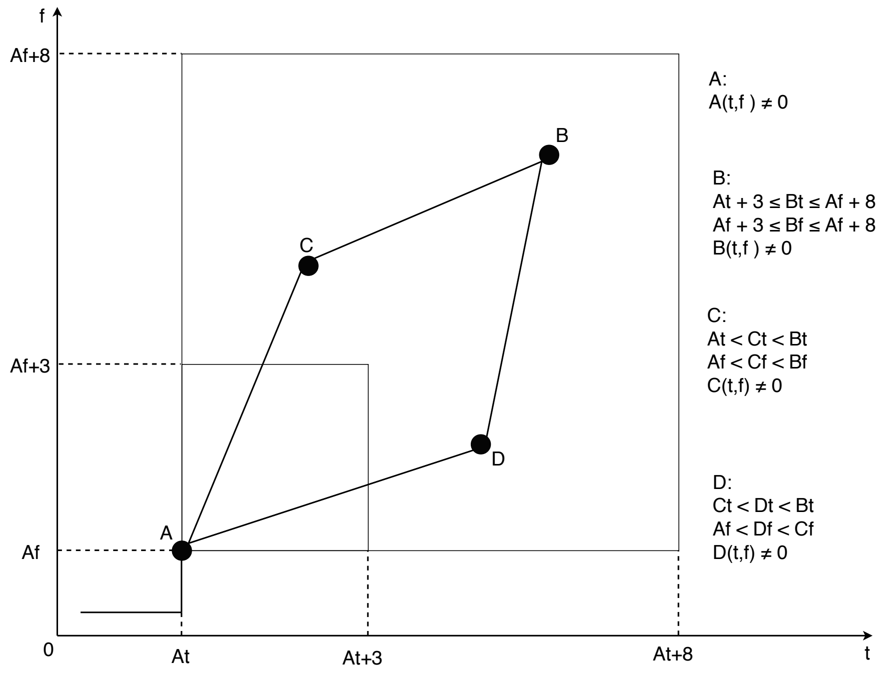


Figure 3.4: Quad look-up process. At represents the time component of A, and Af represents its frequency component.

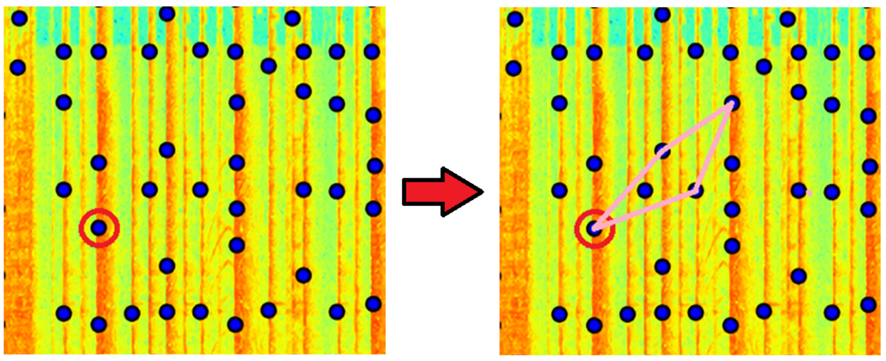


Figure 3.5: Example of an extracted quad, having found point A.

3.6 Database

The Database should be simple to use and fast to access, in order to minimize the time needed. For the database fingerprinting, it must store the quads or some representation of them and their corresponding commercials into a table. For the real time detection, it should be able to search a given hash and return the result, or an error in case it does not exist or a malfunction occurs.

3.7 Decider

The main goal of the system is to be able to determine if the broadcast is airing a commercial or a programming block. This is where the final part of the system takes part. It decides the current block based on the match results it gets from the quad and search blocks.

The program can be in any of the following three states:

- **0**: The initial state, where we don't know yet if we are in the presence of a programming or commercial block
- **1**: The state that represents the presence of a programming block
- **2**: The state that represents the presence of a commercial block

The decisions should be made based on the detection or not of commercials in the stream.

Chapter 4

Implementation

This system was implemented as a C++ program, elaborated in Microsoft Visual Studio 2013. For this, some external libraries where used, as they are very well designed and can improve the performance of this real-time system. The ones used are the BASS[1] audio library, developed by Un4Seen Developments, the SHA-1[2] implementation in C++, originally made by Steve Reid, and OpenCV 3.3.1[3].

4.1 Preprocessing

The audio library is capable of converting the audio into mono, with the following command:

```
HSTREAM stream = BASS_StreamCreateFile(FALSE, path, 0, 0, BASS_SAMPLE_MONO |
    BASS_STREAM_DECODE);
```

This creates a stream from an audio file, which is specified in the *path* variable, and converts it into mono.

4.2 Framing, Overlap and Transform

The audio library is also capable of performing these steps with great performance, which is desired for this project. First, we get the length in bytes of the stream formed in the previous step. Then a cycle is started, which will continue to perform until the end of the audio is reached. During this cycle, the audio library does the FFT for each position in the audio and then the resulting vector is put into a matrix, rendering the desired spectrogram. These steps are illustrated in Figure 4.1. The increment in position corresponds to the hop size mentioned in Section 3.3.

[1]http://www.un4seen.com/
[2]github.com/vog/sha1
[3]opencv.org/

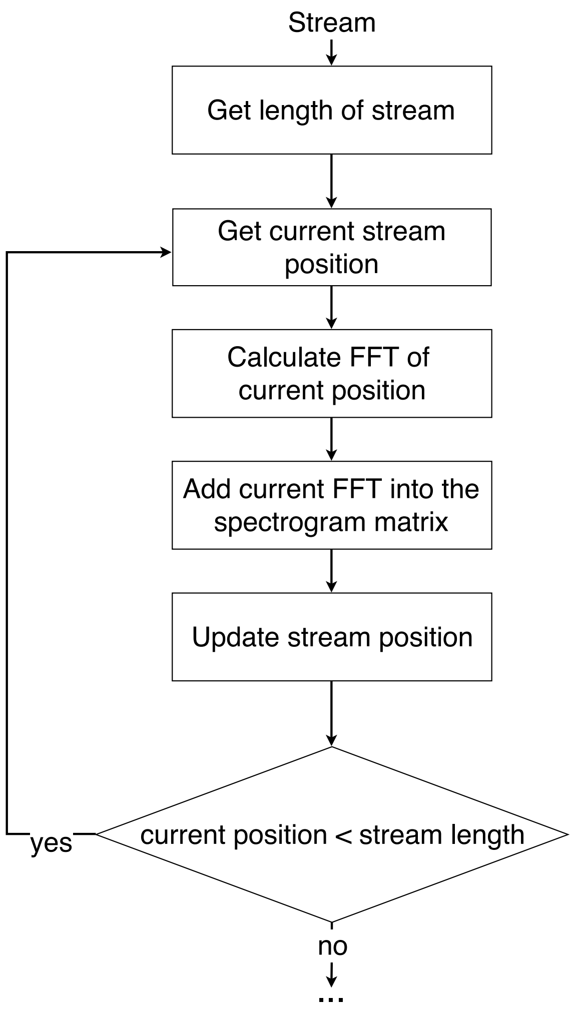

Stream
Get length of stream
Get current stream position
Calculate FFT of current position
Add current FFT into the spectrogram matrix
Update stream position
current position < stream length
yes
no
...

Figure 4.1: Implementation of the Spectrogram Block.

4.3 Feature Extraction

In order to select the best candidates for the features, the matrix containing the spectrogram is converted into an OpenCV Mat image. This means that the program uses computer vision for this step, borrowed from Yan Ke et al. in [7]. The idea is to dilate all the points with a mask of a certain shape and size. The dilations that prevail correspond to the maxima peaks, and share the same values as them. After the dilate operation, both original and dilated Mat images are compared. The result is a Mat that contains a 1 in the points of the original Mat that share the same value and region with the dilated Mat. This means that each dilated region will not have more than a local maximum, excluding the original pixel.

The resulting local maxima are illustrated in 4.2.

The result is a matrix which only contains the best candidates to form quads.

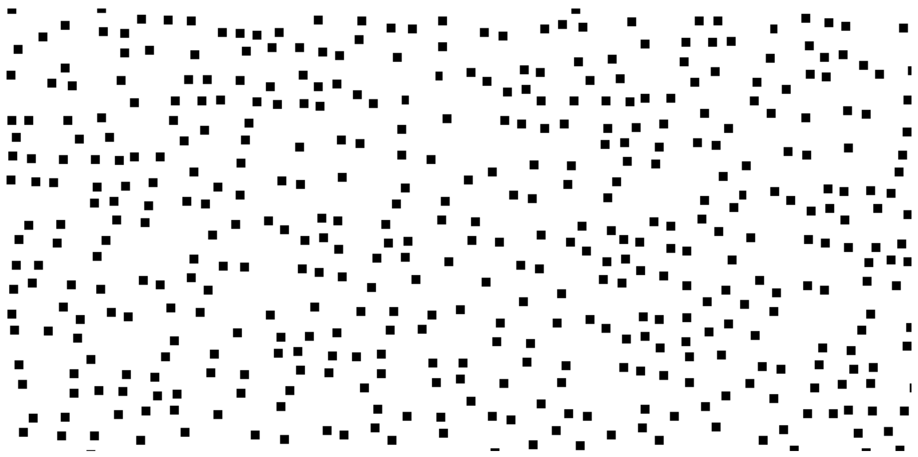


Figure 4.2: The resulting local maxima

4.4 Feature Modeling

In this step, the previously extracted features will be grouped into quads. There are differences in this process if it is done during the database creation or the real-time detection. In the first case, it can generate all quads and then save them in the database, in second one it needs to search the database each time it finds a quad.

4.4.1 Database Creation

For this operation, the feature modeling block will generate all possible quads, saving them into a vector as they are generated. The diagram that describes the quad making process can be seen in Figure 4.3

The value that represents a quad and will be used to generate a hash is a concatenation of four values. These four values are:

- the distance in time bins from point A to point C
- the distance in frequency bins from point A to point C
- the distance in time bins from point D to point B
- the distance in frequency bins from point D to point B

The resulting input for the hash function will have this format:

$$(Ct\text{-}At)||(Cf\text{-}Af)||(Bt\text{-}Dt)||(Bf\text{-}Df)$$

This value is then passed to a Hash function. The one used in this project is the one in [47]. Its implementation is simple and it works well. The SHA-1 algorithm [48] takes an input and produces a 160-bit hash value output. This means that there can be a total of around $1.46x10^{48}$ possible hashes, a number which is more than enough for this project.

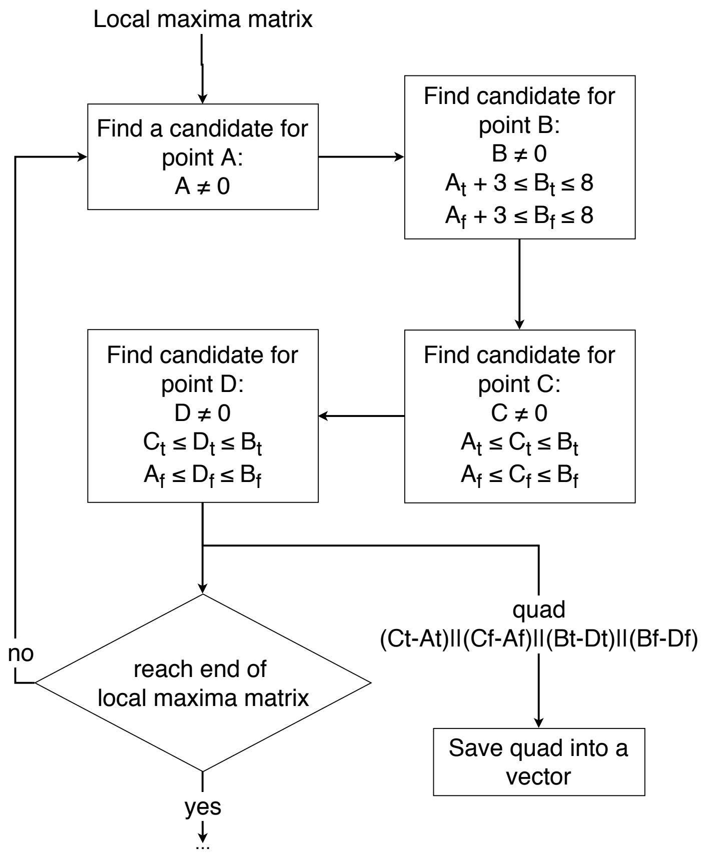

Local maxima matrix
Find a candidate for point A: A ≠ 0
Find candidate for point B: B ≠ 0 $A_t + 3 \leq B_t \leq 8$ $A_f + 3 \leq B_f \leq 8$
Find candidate for point C: C ≠ 0 $A_t \leq C_t \leq B_t$ $A_f \leq C_f \leq B_f$
Find candidate for point D: D ≠ 0 $C_t \leq D_t \leq B_t$ $A_f \leq D_f \leq B_f$
quad (Ct-At)||(Cf-Af)||(Bt-Dt)||(Bf-Df)
reach end of local maxima matrix
no
yes
Save quad into a vector

Figure 4.3: The quad generation for the Database Creation operation

4.4.2 Real-time Detection

When performing the real-time detection, the modeling block needs to perform differently than the one described in Section 4.4.1. Each time a quad is generated, it is searched in the database. If at any time on the buffer we get n-occurences of the same result, then the quad generation stops and sends a *1* to the decider block. This n value corresponds to the needed amount of similar occurrences in order to validate the match for the buffer with 100% certainty. For such cases, a flag is also sent to the Decider block, indicating a certain match. In case n is not reached, and the match is not certain, then it means that the local maxima matrix has reached its end. When this happens, this block should also calculate the percentage of a match. If it equals or exceeds 33% of n, then it sends a *1* to the decider block, otherwise it sends a *0* which corresponds to a *Non-Match*. Figure 4.4 illustrates this block.

4.5 Database

The database was implemented by using the SQLite [49]. This database has no intermediary server process, and is comprised by a single file saved into the computer's disk.

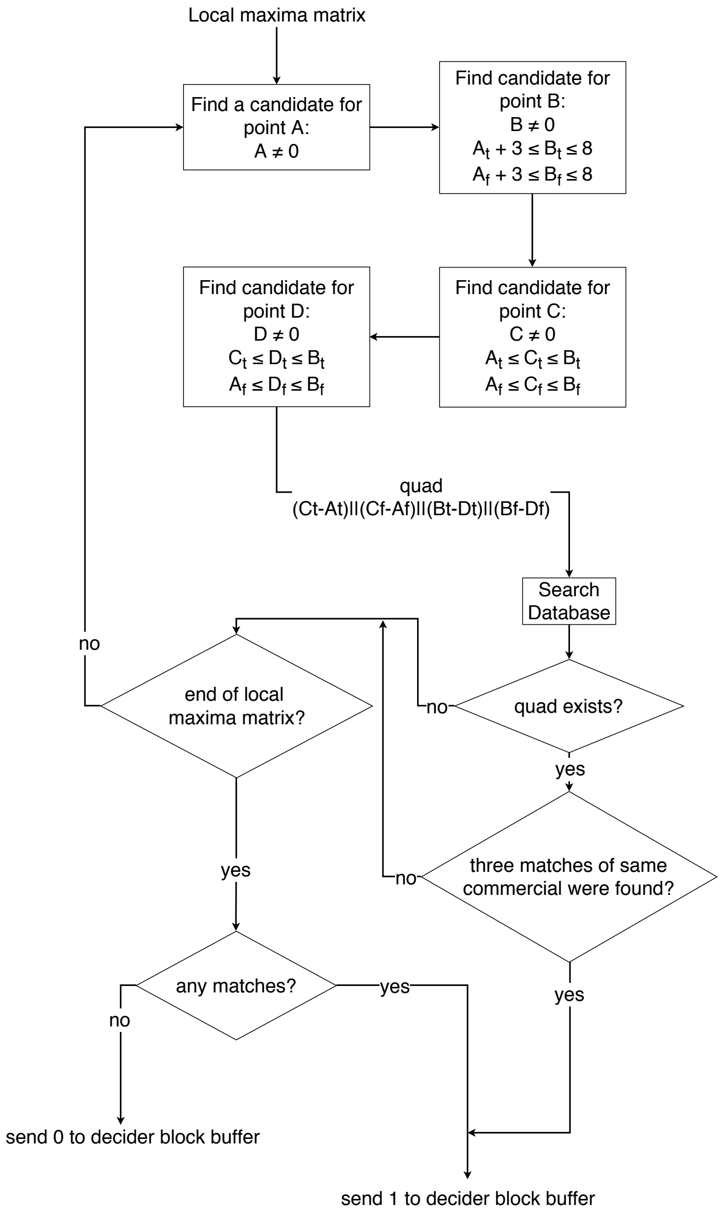

Local maxima matrix
Find a candidate for point A: A ≠ 0
Find candidate for point B: B ≠ 0 At + 3 ≤ Bt ≤ 8 Af + 3 ≤ Bf ≤ 8
Find candidate for point C: C ≠ 0 At ≤ Ct ≤ Bt Af ≤ Cf ≤ Bf
Find candidate for point D: D ≠ 0 Ct ≤ Dt ≤ Bt Af ≤ Df ≤ Bf
quad (Ct-At)||(Cf-Af)||(Bt-Dt)||(Bf-Df)
Search Database
quad exists?
no
yes
three matches of same commercial were found?
no
yes
end of local maxima matrix?
no
yes
any matches?
no
yes
send 0 to decider block buffer
send 1 to decider block buffer

Figure 4.4: The quad generation for the Real-time Detection operation

4.5.1 Creating the Database

The table that will constitute the database is comprised of three columns:

- *id PRIMARY KEY*, only used as the order in which the hashes are stored, having no impact on the system
- *hash STRING*, which will keep the hash value associated with its entry (*id PRIMARY KEY*)
- *comName STRING*, which contains the name of the commercial to which the *hash STRING* belongs

4.5.2 Inserting into the Database

When fingerprinting a commercial, the last step is inserting its corresponding hashes into the database. This is done with a single SQLite query, however there was an important improvement made that greatly influenced the performance of this block. The SQLite, as it is, will always wrap and commit each single query, which means that for each hash generated, it would wrap it in a query, insert it, and then commit the changes in the database. This was greatly slowing down the system. A good way around this was the SQLite concept of a "transaction". This allows multiple insertions to be done, without immediately committing each one of them. It waits for the code inside the *TRANSACTION* to process, and then commits any changes to the database at once, as seen in the following code snippet.

```
sqlite3_exec(db, "BEGIN TRANSACTION;", NULL, NULL, NULL);
    ...
    (insert all the hashes)
    ...
sqlite3_exec(db, "END TRANSACTION;", NULL, NULL, NULL);
```

4.5.3 Searching the database

When performing the real-time commercial detection, each quad must be searched in the database. This is done with SQLite commands, which retrieve a table with the commercial names associated with the hash searched. Since the quad-based approach makes it possible that all quads are unique, each search will only retrieve one result, as shown in Section 4.7.2. The function devised would have to take a hash and the database as inputs, and must output the resulting commercial associated with the hash or an error, in case there are executing errors or the hash doesn't exist in the database. Figure 4.5 shows the implementation of the search function, where we have two SQLite commands. The first one selects all the commercial names which have the searched hash associated, which in this implementation is only returning one result. The second command creates a table with the results obtained in the previous command. After this, if there is no result, then the hash doesn't exist in the database and this function will return *"err"*. If the search is successful, then the resulting commercial name will be returned.

4.5.4 Displaying the database

Although not necessary for the program's operation, having a function that could show the contents of the database is useful. This way we can visually see what it contains. Since one of the main advantages about this program's implementation is the fact that the generated quads are unique, this function also provides the number of entries and the amount of repetition that exist in it. Ideally, there would be none. Figure 4.6 illustrates this implementation.

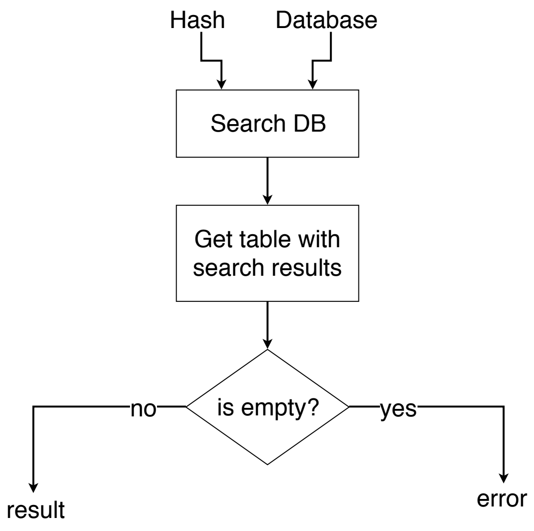


Figure 4.5: Searching the database

4.6 Decider

The Decider block is responsible for changing the current state of the program. It decides if we are in presence of a commercial or programming block, based on the last match results.

The three possible states are:

- **0**: The initial state, where it is unknown if we are in the presence of a programming or commercial block
- **1**: The state of a programming block
- **2**: The state of a commercial block

When the program starts, it doesn't have enough information to decide if there is a commercial or programming block happening. The minimum size of the decision buffer, or occured matches, is 5. So, from state **0**, if a certain match (100%) is found, this means that the stream has commercials, and the Decider block changes the state to **2**. If, in case the percentage is lower than 30%, then the stream contains non-commercial material and the state becomes **1**. Then, since the program is in a state of programming or commercial, it can only change between those two. The initial state only serves at the beginning, in order to chose one of the two possible states. From state **1**, it changes to state **2** if a certain match is found. From state **2**, it changes back to state **1** if the percentage of positive matches is lower than 30%. This implementation is meant to take advantage of the uniqueness of the quads generated, and provide a faster decision, with less latency.

Figure 4.7 depicts the state diagram and the how the states change between each other, whereas Figure 4.8 describes the corresponding flow chart.

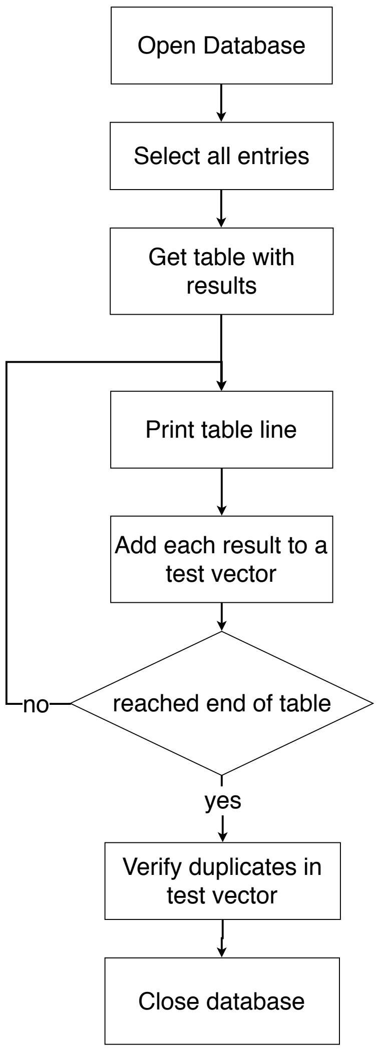


Figure 4.6: Display the database

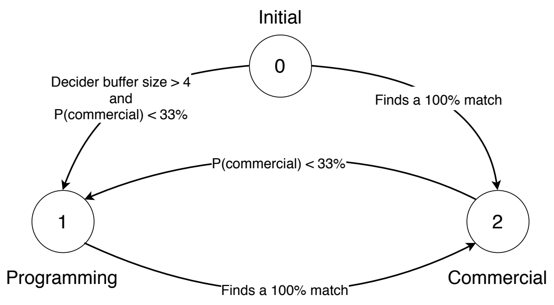


Figure 4.7: Decider Block state diagram.

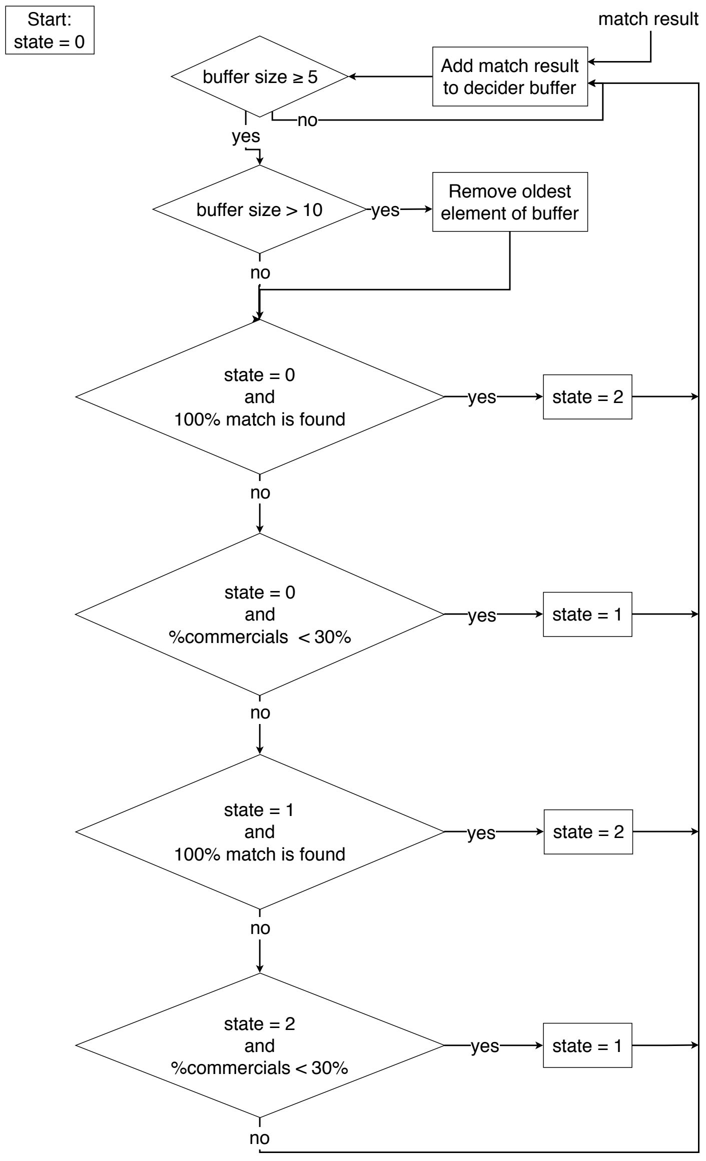

Start:
state = 0
match result
Add match result to decider buffer
buffer size ≥ 5
no
yes
buffer size > 10
yes
Remove oldest element of buffer
no
state = 0 and 100% match is found
yes
state = 2
no
state = 0 and %commercials < 30%
yes
state = 1
no
state = 1 and 100% match is found
yes
state = 2
no
state = 2 and %commercials < 30%
yes
state = 1
no

Figure 4.8: Flowchart of the Decider Block implementation. The exit is the end of file.

4.7 Program Operation

In this section, the two major operations that the system should be capable of performing are detailed. First, the database creation and fingerprinting is described, and then the functioning of the real-time commercial detection is explained. These operations relate to the code developed, provided in A.

4.7.1 Database Creation

The first system operation is the creation of the database, the fingerprinting, of all the commercials that can be present in the given streams. For this, one must first have a folder containing all the commercials that will be fingerprinted. The program then pre-loads all the file paths into a vector and fingerprints one file at a time. To fingerprint a commercial, the program first loads it as a stream, and performs a spectrogram by using the method described in Section 4.2. The resulting spectrogram matrix will then have its features extracted by using the local maxima method described in Secgtion 4.3, which will then output a matrix with the local maxima, the extracted features. The quad generation process mentioned in Section 4.4.1 will then group these feature into quartets, or quads, creating unique ways of representing the audio associated with the commercial. These quads are saved into a vector, which will be taken as input for the database insertion block detailed in Section 4.5.2. When finished, the program will start fingerprinting the next file, and this will occur until the program reaches the end of the folder. Figure 4.9 shows the flow of these operation.

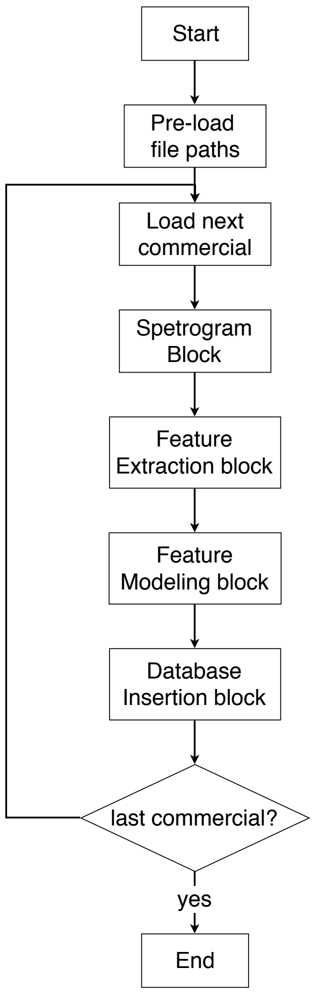


Figure 4.9: Database Creating diagram

4.7.2 Real-time Detection

The second operation is the actual real-time detection of commercials, which will try to identify stream audio by comparing it to the database. It works in a similar way as the Database Creation, but it has few changes. The input is the stream, and the system then buffers 640ms milliseconds of the audio. After this, it performs the spectrogram of these 640ms, as seen in 4.2. The features are then extracted the same way they were in the Database Creation, by the method described in Section 4.3. The next step is different, and it uses the approach detailed in Section 4.4.2, where each generated quad will be instantly searched in the database, resulting in sending a *"programming"* (0) or *"commercial"* (1) to the Decider block, described in Section 4.6. This block will then decide if the stream contains a commercial or a programming block at the current time. A diagram illustrating these steps can be seen in Figure 4.10.

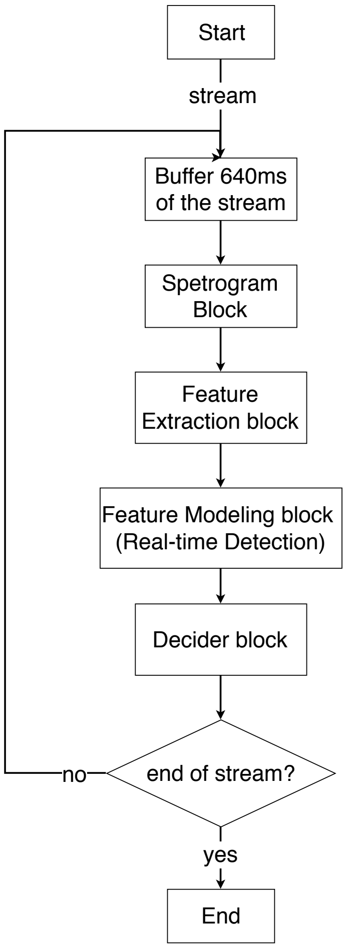


Figure 4.10: Real-time Detection diagram

Chapter 5

Evaluation

In this chapter, the methods used to test and evaluate the developed system. First, the method used to gather the dataset, then the system testing: the Database Creation operation and the Real-Time Detection.

5.1 Data Preparation

In order to collect a dataset, it was needed to gather a great number of commercials from different sources. Since access to channels that are not Portuguese was difficult, it was decided to include some international channels which air international content and channel-specific commercials.

The contemplated channels from which the commercials where gathered are:

- RTP1 - Public general interest channel
- RTP2 - Public general interest channel with no commercials, but has non-programming content
- RTP MS - Special and temporary broadcast of RTP1
- SIC - Private general interest channel
- TVI - Private general interest channel
- CMTV - Private tabloid channel
- Eurosport 1 - Private sports channel, that airs international content and has channel-specific commercials
- NBA TV - Private sports channel, whose content is the same as the country it comes from, including commercials

With 8 TV channels to work with, 2 of them airing international content and commercials, it is possible to test this project in different scenarios. The amount of commercials and their duration can be seen in Table 5.1.

With 500 commercials comprising more than 3 hours of audio, there is plenty of depth to test this project.

At first, the goal was to annotate the whole dataset from streams provided by INESC TEC colleagues that are working with the CloudSetup project. After some work done, it was realized

Table 5.1: Selected commercials for the dataset

	Number of commercials	Total duration
RTP1	46	25”56
RTP2	50	19”53
RTP MS	19	9”27
SIC	13	2”30
TVI	323	2’03”24
CMTV	14	5”58
Eurosport 1	12	7”30
NBA TV	23	11”27
Total	500	3’26”09

that, in order to build such a big dataset out of many hours of streaming, it would take too much time of exclusive dedication to this component. So, in order to facilitate the creation of the dataset, cut commercials from the "Cloud-Setup" project were used. In total, 13 commercials from SIC and 14 from TVI were annotated and cut without using the CloudSetup project dataset, and the other 473 were taken from it.

However, these provided commercials came in the *mxf* file format, which is a video format, and this project is working with *wav* files, that are audio.

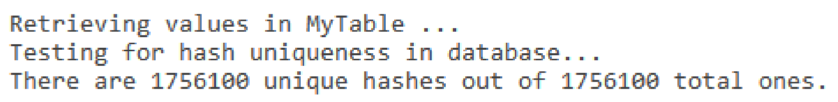
The conversion was done with FFmpeg [50], a multimedia framework that is able to decode, encode, transcode, mux, demux, stream, filter and play any multimedia content. A batch file was created with the following command line:

```
FOR /F "tokens=*" %%G IN ('dir /b *.mxf') DO ffmpeg -i "%%G" -ac 2
    "K:\\ad\\%%~nG.wav"
```

This converts all the *mxf* files in the current folder and extracts the audio, saving it as *wav* files in the folder specified at the end of the command line.section

One of the goals of this system was to generate unique hashes, theorizing that the matching would be faster. This allow for a smaller real-time buffer which would mean a faster decision. After generating the hashes for the 500 commercial collected, the following results were obtained:

- Number of hashes: 1765100 hashes
- Time elapsed: 20″52
- Space in Disk: 203 megabytes
- Hash uniqueness: true, as seen in Figure 5.1



```
Retrieving values in MyTable ...
Testing for hash uniqueness in database...
There are 1756100 unique hashes out of 1756100 total ones.
```

Figure 5.1: Hash uniqueness test results the database.

This reveals that the quad-based approach proved to be effective in creating unique quads, hence generating unique hashes. This was possible despite having such a large number of hashes.

Another thing to have in consideration is the time it takes to access and search the database each time a hash is generated. Having to perform so many searches in very small time windows, the performance is expected to drop significantly with the increase of the database size. This is because with a bigger database, more entries need to be read, which takes computational resources. In order to solve this, the main and only table of the database, which contains the hashes and their associated commercials, was indexed. This means that all rows were aligned in ascending order of hash values. This way, whenever a hash is searched, the table points to the place in the database where the hashes have the same beginning. The indexing was done with the software *DB Browser for SQLite* [51]. After indexing, the database file increased in size from 203 to 286 megabytes.

5.2 Real-time Detection

The main objective of this system is to effectively detect commercials with the least latency possible, while assuring accuracy. Because of the complications of doing multiple tests on a large stream, which would take a long time of waiting, two smaller broadcast test audios were cut from larger streams.

The first test query is 8 minutes long and was taken from SIC channel. It tests the stream starting as a programming block, then changing to a commercial block, then changing back to programming block. It has the following format:

- 0”0.000: Programming block
- 3”16.396: Commercial block, composed by 13 commercials
- 5”52.396: Programming block
- 8”0.000: end

The second test query is 16″24.725 long and was taken from TVI channel. This query tests the opposite, starting with a commercial block, then a programming block and changing back to a commercial block until it ends. Its format is as follows:

- 0”0.000: Commercial block, composed by 8 commercials
- 1”19.030: Programming block
- 14”52.882: Commercial block, composed by 6 commercials
- 16”24.725: end

In order to ensure the real-time performance, timers were placed in the Feature Modeling block. If at any point the operation time exceeds 600 milliseconds, the search for quads would end and the match would be calculated with the current information. Using a buffer of 640 milliseconds, a 40 millisecond interval was estimated to suffice for the remaining steps, based on multiple experiments. In any case, having the program force the system to work in real-time could still hypothetically produce delays. This was meant to secure the real-time functioning but, as it will be demonstrated, the system works without any delays or forced early decisions.

The tables which contain the results will be presented with the following columns:

- **#SMR** Amount of similar search results needed in order to guarantee a 100% match: a higher value should give a greater guarantee but, since the non-100% matches are given based on the fraction

$$\frac{similar\ search\ results\ obtained}{similar\ search\ results\ needed\ for\ 100\%\ match}$$

  requiring more similar results can mean that good matches are lost because they are below the threshold percentage, or that a possibly certain match is not considered as certain because the required amount of similar search results was not achieved
- **#FP** False Positives: The amount of positive matches that were found in blocks of programming
- **#FN** False Negatives: The amount of negative matches found in blocks of commercials. This, together with the *False Positives*, are part of a low-level, frame based decision. They relate to the output of a single buffer, or time window
- **#BD** The amount of bad decisions made by the Decider block, for example changing the state into programming when commercials are still being aired. Different than the *False Positives* and *False Negatives*, this has to do with a high-level decision model, which is maintained by the Decider block shown in Section 4.6
- **Tp1** Time of the detected first Program block in the first test
- **Tc** Time of the detected Commercial block in the first test
- **Tp2** Time of the detected second Program block in the first test
- **Tc1** Time of the detected first Commercial block in the second test
- **Tp** Time of the detected Program block in the second test
- **Tc2** Time of the detected second Commercial block in the second test
- **#D** Amount of delays: Even with the program forcing the real-time operation, the search can surpass the limit, as there is no way to control the search itself.
- **#O** Amount of times the program had to enforce the real-time operation.

The tests ere done for different values of #SMR, hoping to find its optimal value for the developed system by minimizing #BD.

First test

The results for 9 different *#SMR* values under the first test can be seen in Table 5.2.

As we can see, there are no delays or forced decisions, which means that the system is indeed running in real-time. Another conclusion is that with $\#SMR \geq 10$ there are no bad decisions. Also, with $\#SMR \geq 12$ the decision comes later, especially when the Commercial block starts. This is explained by the Decider block: it only changes the state into Commercial block if it encounters

Table 5.2: Results of first test

#SMR	#FP	#FN	#BD	Tp1	Tc	Tp2	#D	#O
6	197	69	5	18.538	3”17.738	6”12.458	0	0
8	83	83	1	3.178	3”17.738	5”57.738	0	0
9	83	83	1	3.178	3”17.738	5”57.738	0	0
10	33	75	0	3.178	3”17.738	5”57.738	0	0
11	33	92	0	3.178	3”17.738	5”57.738	0	0
12	37	90	0	3.178	3”21.578	5”57.738	0	0
15	18	98	0	3.178	3”21.578	5”57.738	0	0
18	8	100	0	3.178	3”21.578	5”57.98	0	0
21	4	102	0	3.178	3”21.578	5”57.98	0	0

a 100% match and, if we increase the needed #SMR, then the probability of the system not being able to find that amount increases, thus not considering a certain match where it should be. The best performance comes with $11 \geq \#SMR \geq 10$, with the time differences shown in Table 5.3 and a visual representation of the time delays in Figure 5.2.

Table 5.3: Best time differences in the first test

Original time (Ot)	Detected time (Dt)	Time difference ($Td = Dt - Ot$)
0”0.000	3.178	3.178
3”16.396	3”17.738	1.342
5”52.396	5”57.738	5.342

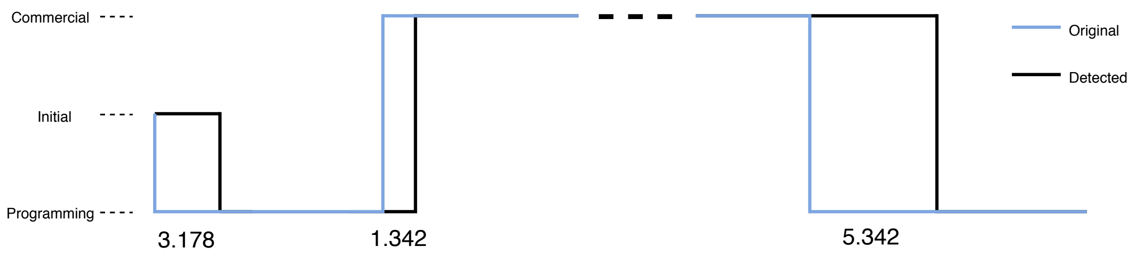


Figure 5.2: Time delays in the first test

The results show an asymmetry problem, since the system seems to be fast at recognizing the Commercial block, but it takes more time to detect a Programming block. This happens because the commercials are in the database, ready to be matched, but the programming content is not. According to the Decider block, a Commercial block will be detected when a certain match (100%) is found. Since none of these happen during programming, it’s a fast, simple and efficient method. There are some *False Positives* showing up, but they are not certain matches so the system doesn’t consider them. They are, however, considered for the Decision buffer, but this buffer will only be used for detecting a Programming block. As mentioned in Section 4.6, a commercial percentage lower than 30% in the buffer will trigger the Decider block into changing states from Commercial to Programming. However, since many *False Negatives* are present, this percentage couldn't be too big, otherwise we would encounter more *Bad Decisions*.

Second test

The results for 9 different *#SMR* values regarding the second test can be seen in Table 5.4.

Table 5.4: Results of second test

#SMR	#FP	#FN	#BD	Tc1	Tp	Tc2	#D	#O
6	517	307	13	0.618	2"11.178	14"57.258	0	0
8	243	104	3	0.618	1"25.98	14"57.258	0	0
9	245	88	1	0.618	1"25.98	14"57.258	0	0
10	82	90	0	0.618	1"24.458	14"57.258	0	0
11	82	90	0	0.618	1"24.458	14"57.258	0	0
12	82	90	0	0.618	1"24.458	14"57.258	0	0
15	32	102	0	0.618	1"24.458	14"57.258	0	0
18	17	105	0	0.618	1"24.458	14"57.258	0	0
21	12	110	0	0.618	1"24.458	14"57.258	0	0

As in the previous test, there are no delays or forced early decisions. Likewise, there are no bad decisions with *#SMR* $\geq$ 10. Here, it is interesting to note the increase in *False Negatives* for *#SMR* $>$ 9. This is due to the fact that, with increased #SMR, more non certain matches get discarded by the minimum match threshold (33% of #SMR), resulting in *False Negatives*. The differences between original and detected times are shown in Table 5.5, and a visual representation of the time delays can be seen in Figure 5.3

Table 5.5: Best time differences in the second test

Original time (Ot)	Detected time (Dt)	Time difference ($Td = Dt - Ot$)
0"0.000	0.618	0.618
1"19.030	1"24.458	5.028
14"55.882	14"57.258	1.624

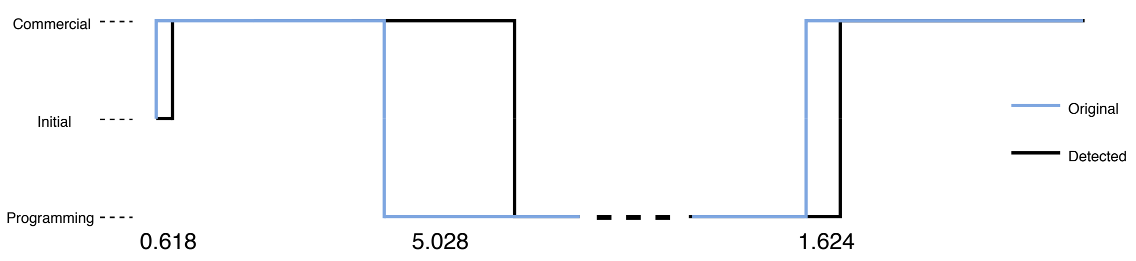


Figure 5.3: Time delays in the second test

Again, the asymmetry problem is present. The Commercial blocks are rapidly detected as it happened with the previous test. Again, the problem is the change into Programming state.

The following graphs show the comparison between the results of both tests. Figure 5.4 depicts the *False Positives*, whereas Figure 5.5 shows the *False Negatives*. Lastly, Figure 5.6 portrays the *Bad Decisions*.

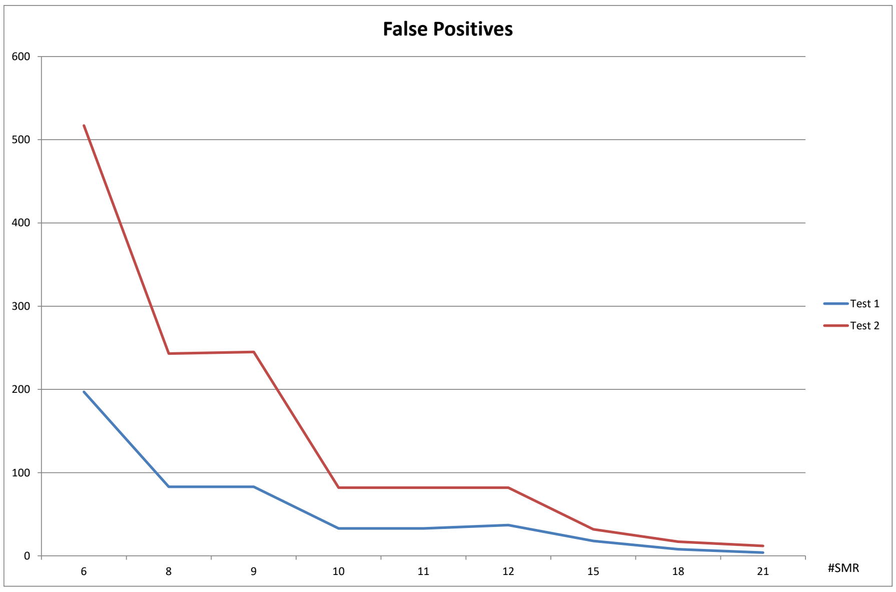


Figure 5.4: False positives comparison

The conclusion is that, for this developed system, having $\#SMR = 10$ seems to maximize efficiency and performance.

5.3 Summary

The goal to achieve uniqueness for the dataset was obtained. However, some false positives appeared, which meant that some quads of the programming blocks existed in the database. The objective of the #SMR value is to assure that a false positive will not be considered as a certain match. What this value is is the required similar search results needed during a buffer so that the match can be cartain. With increased value of #SMR, less false positives appear, however, a value which is too high can compromise the system, as good matches become false negatives.

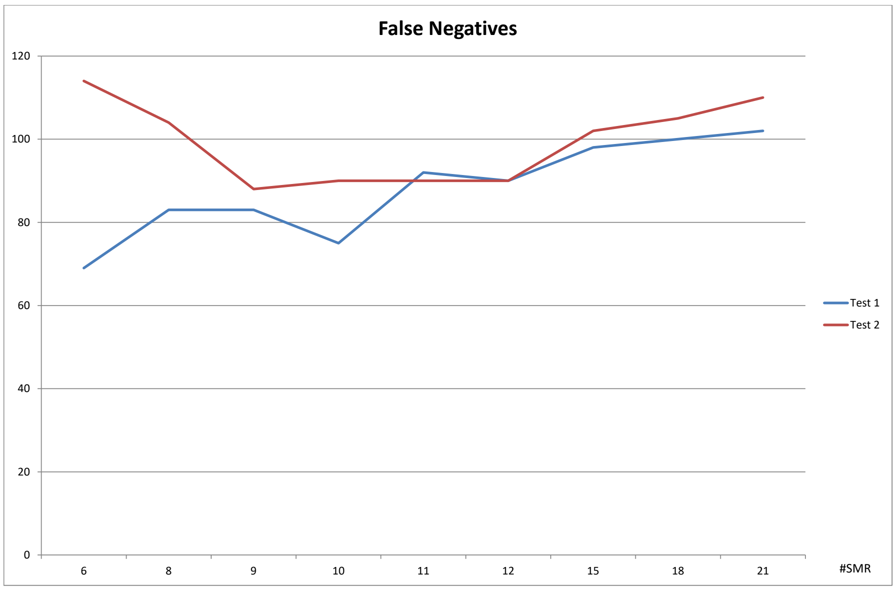


Figure 5.5: False Negatives comparison

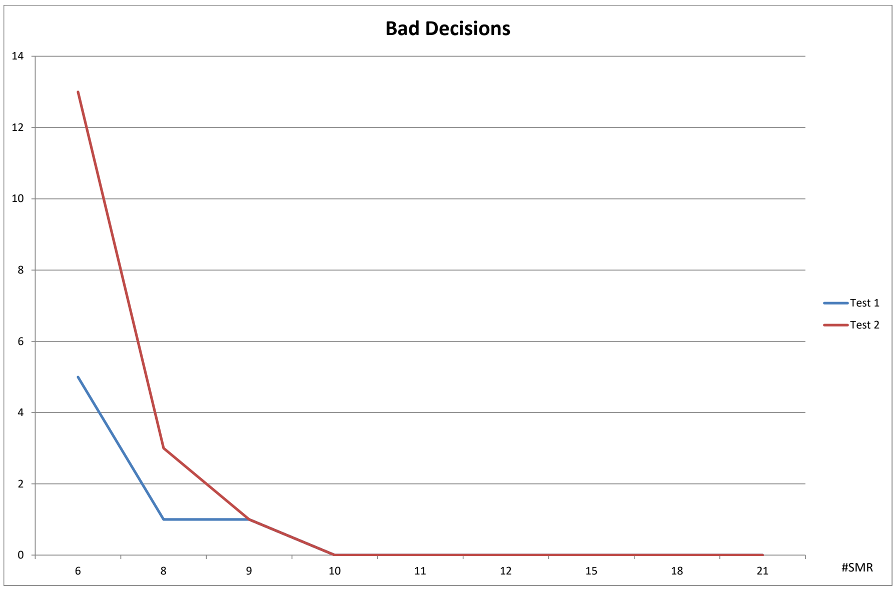


Figure 5.6: Bad decision comparison

Chapter 6

Conclusion

6.1 Summary

In this dissertation, a new system for real-time commercial detection in TV broadcast was presented. This system allows for the automatic detection of commercials on a frame-based, lower level decision, and ultimately concludes if the current audio corresponds to a commercial or a programming blocks. The state of the art systems are only aimed at song identification, and do not focus on the necessity for real-time performance. Even if these perform quite accurately and fast most of the time, there is always the chance of those systems to take a long time to retrieve the results. With this in mind, a new way of modeling the features was presented, by grouping the features in quads, which meant that we could work with unique representations of the audio. Since these representations are specific, the matching process can obtain a certain identification, or match, with fewer iteration through the query's time windows, or buffers. This system is implemented as a C++ application for efficiency

The evaluation of this system was done by running tests on a dataset that was partly annotated, part taken from INESC TEC colleagues working with the CloudSetup project. The evaluation of the performance was done by verifying the presence of bad decisions by the decider block, and then by comparing the detection times with the original times in which the commercial and programming blocks appeared on the stream. Since there are no other known systems doing this currently, the results seem to be rather satisfactory and this work can be further use for developing other kinds of applications: for example one that can implement region-specific switching of commercial content, or even allowing the television channel company to know which commercials are seen the most, for example by allowing the consumer to change between commercials during a commercial block.

6.2 Future work

Despite the good results of the system, there are some features that can be added to this system in order to improve its usability and performance. Due to time restrictions and the amount of work some of these features need, they were not yet implemented. These improvements are the following:

- Increase the overlap in the spectrogram formation: With a considerable amount of *False Negatives* still showing up, it means that parts of the query are not being identified. This could probably be minimized by increasing the overlap from 50% to a higher value.
- Change the decision logic when looking for a Programming block: Related to the previous point, since there are *False Negatives* showing up the buffer needs to be quite big, having a size of 10. Even though the system works in real-time, the results show that the detection of a Programming block comes a few seconds delayed. By changing the buffer, or by adding another one with the IDs of the previous retrievals, we could know if a Commercial block is being aired (similar consecutive retrieved IDs) or not (different consecutive retrievals). This way, the focus would not be not only in consecutive non-matches, but also on what kind of matches are showing up. Two similar IDs can be retrieved with some non-matches being shown in the middle, which makes it difficult for the current system to perform a faster decision. Another way around this problem could be to save in the database the time at which a quad was generated when forming the database. This could allow the system to know how much time was left until the current commercial ends.
- More features extracted: Since the quad uniqueness was successful for a dataset of 500 commercials, one could try to increase the density of obtained local maxima, which would generate a bigger amount of quads, possibly minimizing the amount of *False Negatives*.
- Possibility to update the database: As of now, the database needs to be built from scratch each time there are changes in it. The insertion process works, because it doesn't overwrite, but removing entries needs to be done manually in the code. It would be useful to add such function that would ask which commercial to remove, and then do it.
- Further testing: More testing should be done, with different sampling rates, FFT windows, hop sizes, local maxima masks, limit look-up distances for quad generation, real-time detection buffers. Also, it would be interesting to test this system with streams that have lower quality, introduced noise or other obstacles to the performance of the system.

6.3 Perspectives on the Project

To tackle a problem which hasn't been directly studied was a rewarding experience, albeit difficult at times. This project provided me with greater knowledge about C++ programming, audio fingerprinting and audio processing.

Appendix A

System C++ code

This appendix contains the C++ code of the developed system.

A.1 C++ Code

```
  #include <iostream>
#include "sha1.hpp"
#include <cmath>
#include <math.h>
#include <ctime>
#include "bass.h"
#include <numeric>
#include <vector>
#include <dirent.h>
#include <string>
#include <algorithm>    // std::sort e find
#include <sqlite3.h>
#include <opencv2\opencv.hpp> //for the mat to find local maxima
#include <opencv2\highgui\highgui.hpp> //for the mat to find local maxima
#include <chrono>

using namespace std::chrono;
using namespace std;

void initDB(){
  int rc;
  char *error;
  sqlite3 *db;
  rc = sqlite3_open("MyDb.db", &db);
  if (rc)
  {
    cerr << "Error opening SQLite3 database: " << sqlite3_errmsg(db) << endl
   << endl;
    sqlite3_close(db);
    //return 1;
  }
  std::cout << "Creating MyTable ..." << endl;
  const char *sqlCreateTable = "CREATE TABLE MyTable (id INTEGER PRIMARY KEY,
    hash STRING, songName STRING);";
  rc = sqlite3_exec(db, sqlCreateTable, NULL, NULL, &error);
  if (rc)
  {
    cerr << "Error executing SQLite3 statement: " << sqlite3_errmsg(db) <<
    endl << endl;
    sqlite3_free(error);
  }
  else
  {
    std::cout << "Done creating MyTable." << endl << endl;
  }
  // Close Database
  sqlite3_close(db);
```

```
}

void displayDB(){
  int rc;
  int i = 0;
  char *error;
  sqlite3 *db;
  vector<string> uniqTest;
  rc = sqlite3_open("MyDb.db", &db);
  if (rc)
  {
    cerr << "Error opening SQLite3 database: " << sqlite3_errmsg(db) << endl
   << endl;
    sqlite3_close(db);
  }
  std::cout << "Retrieving values in MyTable ..." << endl;
  const char *sqlSelect = "SELECT * FROM MyTable;";
  char **results = NULL;
  int rows, columns;
  sqlite3_get_table(db, sqlSelect, &results, &rows, &columns, &error);
  if (rc)
  {
    cerr << "Error executing SQLite3 query: " << sqlite3_errmsg(db) << endl <<
   endl;
    sqlite3_free(error);
  }
  else
  {
    // Display Table
    for (int rowCtr = 0; rowCtr <= rows; ++rowCtr)
    {
      for (int colCtr = 0; colCtr < columns; ++colCtr)
      {
        // Determine Cell Position
        int cellPosition = (rowCtr * columns) + colCtr;

        // Display Cell Value
        std::cout.width(12);
        std::cout.setf(ios::left);
        //std::cout << results[cellPosition] << " ";
        if (cellPosition % 3 == 0){
          uniqTest.push_back(results[cellPosition]);
        }
      }
      // End Line
      //std::cout << endl;

      // Display Separator For Header
      if (0 == rowCtr)
      {
        for (int colCtr = 0; colCtr < columns; ++colCtr)
        {
          std::cout.width(12);
          std::cout.setf(ios::left);
          //std::cout << "~~~~~~~~~~~~  ";
        }
        //std::cout << endl;
      }
    }
    //test for hash uniqueness
    cout << "Testing for hash uniqueness in database..." << endl;
    std::sort(uniqTest.begin(), uniqTest.end());
    int uniqueCount = std::unique(uniqTest.begin(), uniqTest.end()) - uniqTest.begin();
    cout << "There are " << uniqueCount << " unique hashes out of " << uniqTest.size() << " total ones." << endl;
    //free table and close DB
    sqlite3_free_table(results);
    std::cout << "Closing MyDb.db ..." << endl;
    sqlite3_close(db);
    std::cout << "Closed MyDb.db" << endl << endl;
  }
}

string searchDB(string hash, sqlite3 *db){
  string res; //string that will be returned
  int rc;
  char *error;
  string temp = "SELECT songName FROM MyTable WHERE hash = '" + hash + "';"; //search the table for the hash
  const char *sqlSearchTable = temp.c_str();
  rc = sqlite3_exec(db, sqlSearchTable, NULL, NULL, &error);
  if (rc)
  {
    cerr << "Error executing SQLite3 statement: " << sqlite3_errmsg(db) << endl << endl;
    sqlite3_free(error);
    res = "err";
    return res;
  }
  char **results = NULL;
  int rows, columns;
  sqlite3_get_table(db, sqlSearchTable, &results, &rows, &columns, &error); //gets the table with the results
  if (rc)
  {
    cerr << "Error executing SQLite3 query: " << sqlite3_errmsg(db) << endl << endl;
```

```cpp
    sqlite3_free(error);
    return "err";
  }
  else
  {
    if (rows == 0 || columns == 0) {
    sqlite3_free_table(results);
    return "err";
  }
  else{
    res = results[1];
    sqlite3_free_table(results);
    return res;
  }
  }
}

int fingerPrint(void *path, string songName, int &numEr){
  std::cout << "Fingerprinting: " << songName << endl;
  vector<vector<double>> m; //matrix that contains de FFT vectors
  vector<vector<int>> p; //matrix that contains de quads
  int conta = 0;
  if (!BASS_Init(-1, 48000, 0, NULL, NULL))  //initializes a Bass container at 48000Hz
  {
    std::cout << "Can't initialize device";
    return -1;
  }

  HSTREAM stream = BASS_StreamCreateFile(FALSE, path, 0, 0, BASS_SAMPLE_MONO | BASS_STREAM_DECODE); // create decoding channel

  if (stream != 0)
  {
    long streamLength = BASS_ChannelGetLength(stream, 0);  //gets stream lenght in bytes
    int streamTime = (int)BASS_ChannelBytes2Seconds(stream, streamLength); //converts stream lenght to time
    int streamTimeMins = (int)floor(streamTime / 60.0);
    int streamTimeSecs = (int)streamTime % 60;
    double pos_now_sec = 0, pos_old_sec = 0;
    long streamPosition = 0;

    while (streamPosition < streamLength)
    {
      streamPosition = BASS_ChannelGetPosition(stream, BASS_POS_BYTE);  //gets current position in the stream
      pos_now_sec = BASS_ChannelBytes2Seconds(stream, streamPosition); //converts position to time

      float d[1024] = { 0 };  //current FFT vector
      BASS_ChannelGetData(stream, d, BASS_DATA_FFT2048);  //gets FFT
      //for checking updated positions in stream, both in bytes and time:
      /*int ms = static_cast<int>(floor(fmod((pos_now_sec * 1000.0), 1000.0)));
      int sec = static_cast<int>(pos_now_sec) % 60;
      int minutes = static_cast<int>(std::floor(pos_now_sec / 60));
      cout << "Byte Pos: {" << streamPosition << "/" << streamLength << "}; Time Pos: {" <<
      minutes << ":" << sec << "." << ms << "} out of: {" << streamTimeMins << ":" << streamTimeSecs << "}" << endl;*/
      streamPosition += 4096; //position in bytes to advance. for a hop size of half the window, this value should be 2*FFT
      BASS_ChannelSetPosition(stream, streamPosition, BASS_POS_BYTE); //advances on the stream
      vector<double> v(begin(d), end(d));  //convert fft array into vector
      m.push_back(v);
    }
  }
  else
  {
    // error creating the stream
    std::cout << "Stream error: {0}", BASS_ErrorGetCode();
    return -1;
  }

  BASS_StreamFree(stream); //free stream
  BASS_Free(); // free BASS

  //Get local maxima
  cv::Mat matIn(m.size(), m.at(0).size(), CV_64FC1);
  for (int i = 0; i < matIn.rows; ++i){
    for (int j = 0; j < matIn.cols; ++j){
      matIn.at<double>(i, j) = m[i][j];
    }
  }
  cv::Mat binStruct = cv::getStructuringElement(cv::MORPH_ELLIPSE, cv::Size(30, 30)); // original size   41
  //cv::imshow("MASK", binStruct);
  //cv::waitKey(0);
  cv::Mat dilatedMat, localMaxs;
  cv::dilate(matIn, dilatedMat, binStruct);
  //cv::imshow("DILATED", dilatedMat);
  //cv::waitKey(0);
  cv::compare(matIn, dilatedMat, localMaxs, cv::CMP_EQ);
  //cv::imshow("MAXIMA", localMaxs);
  //cv::waitKey(0);
  if (localMaxs.empty()) cout << "LocalMax vazio" << endl;

```

```
for (int i = 0; i < localMaxs.rows; ++i){
  for (int j = 0; j < localMaxs.cols; ++j){
    m[i][j] = localMaxs.at<uchar>(i, j);
  }
}

//Get quads
vector<int> fi;
bool flag = 0;
for (std::vector<std::vector<int>>::size_type i = 0; i < m.size()-8; i++)
{
  for (std::vector<int>::size_type j = 0; j < m[i].size(); j++)
  {
    if (m[i][j] != 0){
      for (int a = 3; a <= 8; a++)
      {
        for (int b = 3; b < m[i].size()-j; b++)
        {
          if (m[i + a][j + b] != 0){
            for (int c = 1; c < a; c++)
            {
              for (int l = 2; l < b; l++)
              {
                if (m[i + c][j + l] != 0 ){
                  for (int w = c + 1; w < a; w++)
                  {
                    for (int h = 1; h < l; h++)
                    {
                      if (m[i + w][j + h] != 0){
                        fi.push_back(c);
                        fi.push_back(l);
                        fi.push_back(a-w);
                        fi.push_back(b-h);
                        if (std::find(p.begin(), p.end(),fi)!=p.end()){
                          //cout << "REPEAT" << endl; //done to prevent quad
  duplicates in the same commercial. Was not needed
                        }
                        else {
                          p.push_back(fi);
                          conta++;
                        }
                        if (p.size() > 30000){
                          cout << "FLAG!"<< endl;
                          numEr++;
                          flag = 1;
                          break;
                        }
                        fi.clear();

                      }
                      if (flag == 1) break;
                    }
                    if (flag == 1) break;
                  }
                  if (flag == 1) break;
                }
                if (flag == 1) break;
              }
              if (flag == 1) break;
            }
            if (flag == 1) break;
          }
          if (flag == 1) break;
        }
        if (flag == 1) break;
      }
      if (flag == 1) break;
    }
    if (flag == 1) break;
  }
  if (flag == 1) break;
}

std::cout << "A total of " << conta << " quads for " << songName << " was found." << endl;

// Open Database
int rc;
char *error;
sqlite3 *db;
rc = sqlite3_open("MyDb.db", &db);
if (rc)
  {
    cerr << "Error opening SQLite3 database: " << sqlite3_errmsg(db) << endl << endl;
    sqlite3_close(db);
    return -1;
  }
//CREATE HASHES
SHA1 checksum;
int kij;
string r;
string xx;
sqlite3_exec(db, "BEGIN TRANSACTION;", NULL, NULL, NULL);
for (std::vector<std::vector<int>>::size_type i = 0; i < p.size(); i++)
{
  for (std::vector<int>::size_type j = 0; j < p[i].size(); j++)
  {
        kij = p[i][j];
        r = to_string(kij);
        xx = xx + r;
    }
    checksum.update(xx);
    const string hash = checksum.final();
    xx.clear();

    // Execute SQL
    //cout << "Inserting a value into MyTable ..." << endl;
    string aha = "INSERT INTO MyTable VALUES(NULL, '" + hash + "', '" +
   songName + "');";
    const char *sqlInsert = aha.c_str();
    rc = sqlite3_exec(db, sqlInsert, NULL, NULL, &error);
    if (rc)
    {
      cerr << "Error executing SQLite3 statement: " << sqlite3_errmsg(db) <<
   endl << endl;
      sqlite3_free(error);
      return -1;
    }
  }

  sqlite3_exec(db, "END TRANSACTION;", NULL, NULL, NULL);

  std::cout << "Done with: " << songName << endl << endl;
  sqlite3_close(db);
  return 1;
}

void rtMatch(void *path){
  // Open Database
  int state = 0;
  int contar = 0;
  int rc;
  sqlite3 *db;
  rc = sqlite3_open("MyDb.db", &db);
  if (rc)
    {
      cerr << "Error opening SQLite3 database: " << sqlite3_errmsg(db) << endl
   << endl;
      sqlite3_close(db);
      //return 1;
    }
  vector<int> deci;
  vector<int> tempos;
  int numatrasos = 0;
  int buff = 0, koi = 0, false_pos = 0, false_neg = 0, over = 0;
  int mycount;
```

```
std::cout << "Matching the stream: " << endl;
vector<vector<double>> m; //matrix that contains de FFT vectors
vector<vector<int>> p; //matrix that contains de quads
int conta;
if (!BASS_Init(−1, 48000, 0, NULL, NULL)) //initializes a Bass container at 48000Hz
{
  std::cout << "Can't initialize device";
  //return −1;
}

HSTREAM stream = BASS_StreamCreateFile(FALSE, path, 0, 0, BASS_SAMPLE_MONO | BASS_STREAM_DECODE); // create decoding channel

if (stream != 0)
{
  long streamLength = BASS_ChannelGetLength(stream, 0);  //gets stream lenght in bytes
  int streamTime = (int)BASS_ChannelBytes2Seconds(stream, streamLength); //converts stream lenght to time
  int streamTimeMins = (int)floor(streamTime / 60.0);
  int streamTimeSecs = (int)streamTime % 60;
  double pos_now_sec = 0, pos_old_sec = 0;
  long streamPosition = 0;
  int ms2 = 0, sec2 = 0, minutes2 = 0;

  while (streamPosition < streamLength)
  {
    streamPosition = BASS_ChannelGetPosition(stream, BASS_POS_BYTE);  //gets current position in the stream
    pos_now_sec = BASS_ChannelBytes2Seconds(stream, streamPosition); //converts position to time
    float d[1024] = { 0 };  //current FFT vector
    BASS_ChannelGetData(stream, d, BASS_DATA_FFT2048);  //gets FFT
    //for checking updated positions in stream, both in bytes and time:
    int ms = static_cast<int>(floor(fmod((pos_now_sec ∗ 1000.0), 1000.0)));
    int sec = static_cast<int>(pos_now_sec) % 60;
    int minutes = static_cast<int>(std::floor(pos_now_sec / 60));
    int msAnt = ms2;
    int secAnt = sec2;
    int minutesAnt = minutes2;
    //cout << "Byte Pos: {" << streamPosition << "/" << streamLength << "}; Time Pos: {" <<
    //minutes << ":" << sec << "." << ms << "} out of: {" << streamTimeMins << ":" << streamTimeSecs << "}" << endl;
    streamPosition += 4096; //position in bytes to advance. for a hop size of half the window, this value should be 2∗FFT
    buff += 4096;
    koi += 4096;
```

```cpp
    BASS_ChannelSetPosition(stream, streamPosition, BASS_POS_BYTE);
  //advances on the stream
    vector<double> v(begin(d), end(d));  //convert fft array into vector
    m.push_back(v);
    //faz o match
    if (koi >= 703 * 4096){
      cout << "Current time: " << minutes << ":" << sec << "." << ms << " ,
  with delays: " << numatrasos << endl;
      koi = 0;
    }
    if (buff >= 30 * 4096){
      ms2 = ms;
      sec2 = sec;
      minutes2 = minutes;
      auto com = high_resolution_clock::now();
      buff = 0;
      //Get local maxima
      cv::Mat matIn(m.size(), m.at(0).size(), CV_64FC1);
      for (int z = 0; z < matIn.rows; ++z){
        for (int y = 0; y < matIn.cols; ++y){
          matIn.at<double>(z, y) = m[z][y];
        }
      }
      cv::Mat binStruct = cv::getStructuringElement(cv::MORPH_ELLIPSE,
  cv::Size(30, 30));
      cv::Mat dilatedMat, localMaxs;
      cv::dilate(matIn, dilatedMat, binStruct);
      cv::compare(matIn, dilatedMat, localMaxs, cv::CMP_EQ);
      for (int z = 0; z < localMaxs.rows; ++z){
        for (int y = 0; y < localMaxs.cols; ++y){
          m[z][y] = localMaxs.at<uchar>(z, y);
        }
      }

      //QUADS and match
      vector<int> quant;
      vector<string> resu;
      vector<string>::iterator iter;
      vector<int>::iterator ite;
      int index = 0;
      bool flag = 0;
      bool flag2 = 0;
      SHA1 checksum;
      string r;
      string xx;

      for (int i = 0; i <= m.size() - 9; i++)
      {
        for (int j = 0; j < m[i].size(); j++)
      {
        if (m[i][j] != 0){
          for (int a = 3; a <= 8; a++)
          {
            for (int b = 3; b < m[i].size() - j; b++)
            {
              if (m[i + a][j + b] != 0){
                for (int c = 1; c < a; c++)
                {
                  for (int l = 2; l < b; l++)
                  {
                    if (m[i + c][j + l] != 0){
                      for (int w = c + 1; w < a; w++)
                      {
                        for (int h = 1; h < l; h++)
                        {
                          if (m[i + w][j + h] != 0){
                            int lo = a - w;
                            int ko = b - h;
                            xx = to_string(c) + to_string(l) + to_string(lo) + to_string(ko);
                            checksum.update(xx);
                            const string hash = checksum.final();
                            xx.clear();
                            string temp = searchDB(hash, db);
                            if (temp != "err"){
                              iter = find(resu.begin(), resu.end(), temp);
                              if (iter != resu.end())
                              {
                                index = distance(resu.begin(), iter);
                                quant[index]++;
                                if (quant[index] == 10){ //if it gets ten instances of the same result, there is a 100% match
                                  cout << "Match: " << searchDB(hash, db) << " with 100% chance" << endl;
                                  deci.push_back(1);
                                  flag = 1;
                                  for (int k = 0; k < deci.size(); k++) deci[k] = 1;
                                }
                              }
                              else{
                                resu.push_back(temp);
                                quant.push_back(1);
                              }
                            }

                          }
                          if (flag == 1 || flag2 == 1) break;
```

```
                              auto cois = high_resolution_clock::now();
                              auto dur = duration_cast<milliseconds>(cois - com);
                              if (dur.count() >= 600) flag2 = 1;
                              if (flag == 1 || flag2 == 1) break;
                            }
                            if (flag == 1 || flag2 == 1) break;
                            auto cois = high_resolution_clock::now();
                            auto dur = duration_cast<milliseconds>(cois - com);
                            if (dur.count() >= 600) flag2 = 1;
                            if (flag == 1 || flag2 == 1) break;
                          }
                        }
                        if (flag == 1 || flag2 == 1) break;
                        auto cois = high_resolution_clock::now();
                        auto dur = duration_cast<milliseconds>(cois - com);
                        if (dur.count() >= 600) flag2 = 1;
                        if (flag == 1 || flag2 == 1) break;
                      }
                      if (flag == 1 || flag2 == 1) break;
                      auto cois = high_resolution_clock::now();
                      auto dur = duration_cast<milliseconds>(cois - com);
                      if (dur.count() >= 600) flag2 = 1;
                      if (flag == 1 || flag2 == 1) break;
                    }
                  }
                  if (flag == 1 || flag2 == 1) break;
                  auto cois = high_resolution_clock::now();
                  auto dur = duration_cast<milliseconds>(cois - com);
                  if (dur.count() >= 600) flag2 = 1;
                  if (flag == 1 || flag2 == 1) break;

                }
                if (flag == 1 || flag2 == 1) break;
                auto cois = high_resolution_clock::now();
                auto dur = duration_cast<milliseconds>(cois - com);
                if (dur.count() >= 600) flag2 = 1;
                if (flag == 1 || flag2 == 1) break;
              }
            }
            if (flag == 1 || flag2 == 1) break;
          }

          if (flag == 1){
            break;
          }
          else if ((flag == 0) && (i == m.size() - 9) && (!quant.empty())){
            int max = *max_element(quant.begin(), quant.end());
            if (max < 4){
```

```
          cout << "NON-MATCH //but found few quads" << endl;
          deci.push_back(0);
          if (state == 2) false_neg++;
        }
        else{
          ite = max_element(quant.begin(), quant.end());
          index = distance(quant.begin(), ite);
          cout << "Match: " << resu[index] << " with "<<
    ((double)quant[index] / (double)10) * 100 << "% chance" << endl;
          deci.push_back(1);
          if (state == 1) false_pos++;
        }
      }
      else if ((flag == 0) && (i == m.size() - 9) && (quant.empty())){
        cout << "NON-MATCH" << endl;
        deci.push_back(0);
        if (state == 2) false_neg++;
      }
      else if (flag2 == 1 && !quant.empty()){
        int max2 = *max_element(quant.begin(), quant.end());
        if (max2 < 4){
          cout << "NON-MATCH but found few quads ~TIMEOUT" << endl;
          deci.push_back(0);
          over++;
          if (state == 2) false_neg++;
        }
        else{
          ite = max_element(quant.begin(), quant.end());
          index = distance(quant.begin(), ite);
          cout << "Match: " << resu[index] << " with " <<
    ((double)quant[index] / (double)10) * 100 << "% chance ~TIMEOUT" << endl;
          deci.push_back(1);
          over++;
          if (state == 1) false_pos++;
        }
        break;
      }
      else if (flag2 == 1 && quant.empty()){
        cout << "NON-MATCH ~TIMEOUT" << endl;
        deci.push_back(0);
        over++;
        if (state == 2) false_neg++;
        flag = 0;
        break;
      }
    }
    //End of QUADMATCHING

    resu.clear();
```

```
        quant.clear();
        p.clear();
        m.clear();

        if (deci.size() > 10){ //if deletes first element of buffer if it exceeedes capacity
          deci.erase(deci.begin());
        }
        ///Gives chance of having a block of commercial or programming

        mycount = count(deci.begin(), deci.end(), 1);
        if (state == 0 && (flag == 1)/*((deci.size() > 4 && ((double)mycount / (double)deci.size()) >= 0.4) || (flag == 1))*/){
          state = 2;
          cout << endl << "A BLOCK OF COMMERCIAL STARTED at: " << minutes << "'" << sec << "." << ms << endl;
        }
        else if (state == 0 && deci.size() > 4 && ((double)mycount / (double)deci.size()) < 0.3){
          state = 1;
          cout << endl << "A BLOCK OF PROGRAMMING STARTED at: " << minutes << "'" << sec << "." << ms << endl << endl;
        }
        else if ((state == 1 && (flag == 1)/*((((double)mycount / (double)deci.size()) >= 0.4) || (flag == 1))*/)){
          state = 2;
          for (int i = 0; i < deci.size(); i++) deci[i] = 1;
          cout << endl << "A BLOCK OF COMMERCIAL STARTED at: " << minutes << "'" << sec << "." << ms << endl;
          for (int k = 0; k < deci.size(); k++) deci[k] = 1;
          //cout << "ORIGINAL TIME OF START          : " << minutesAnt << "'" << secAnt << "." << msAnt << endl << endl;
        }
        else if (state == 2 && ((double)mycount / (double)deci.size()) < 0.3){
          state = 1;
          cout << endl << "A BLOCK OF PROGRAMMING STARTED at: " << minutes << "'" << sec << "." << ms << endl << endl;
        }
        auto term = high_resolution_clock::now();
        auto duration = duration_cast<milliseconds>(term - com);
        if (duration.count() >= 640){
          numatrasos++;
        }
        tempos.push_back(duration.count());
      }

    }
  }
  else
  {
    // error creating the stream
    std::cout << "Stream error: {0}", BASS_ErrorGetCode();
  }

  cout << endl << endl << "Highest Time elapsed in a match: " <<
    *max_element(tempos.begin(), tempos.end()) << endl;
  cout << "Lowest Time elapsed in a match: " << *min_element(tempos.begin(),
    tempos.end()) << endl;
  cout << "Average time elapsed in a match: " << (accumulate(tempos.begin(),
    tempos.end(), 0)) / (tempos.size()) << endl;
  cout << "Number of delayed decisions (longer than buffer window): " <<
    numatrasos << endl;
  BASS_StreamFree(stream); //free stream
  BASS_Free(); // free BASS

  //clear m
 m.clear();
  cout << "FP: " << false_pos << " FN: " << false_neg << " OVER: " << over <<
    endl;
  // Close Database
  sqlite3_close(db);
}

void createDB(const char * path){
  //Loads and fingerprints a folder into the database
  vector<string> paths;
  string temp;
  DIR *dir;
  struct dirent *ent;
  int numErr = 0;
  if ((dir = opendir(path)) != NULL) {
    // gets all the files and directories within directory //
    while ((ent = readdir(dir)) != NULL) {
      if (strcmp(ent->d_name, ".") != 0 && strcmp(ent->d_name, "..") != 0)  {
        temp = ent->d_name;
        temp = path + temp;
        paths.push_back(temp);
      }
    }
    closedir(dir);
  }
  else {
    // could not open directory //
    perror("");
  }
  //fingerprints each file according to its path
  int fp;
  for(int i = 0; i < paths.size(); i++)
  {
    char *path = new char[paths[i].size() + 1];
    std::copy(paths[i].begin(), paths[i].end(), path);
    path[paths[i].size()] = '\0';
    void *pathQu = path;
    fp = fingerPrint(pathQu, paths[i], numErr);
    memset(path, 0, sizeof(path));
  }
  cout << endl << endl << numErr << endl;
}

int main(int argc, const char **argv){

  initDB();

  auto start = high_resolution_clock::now();

  ////////////// FINGERPRINT FOLDER OF COMMERCIALS /////////
  //createDB("C:\\Users\\ADMIN\\Desktop\\dataset\\db\\");

  //Display whole db table + check for hash uniqueness//////
  //displayDB();

  //////////// DO REAL TIME MATCHING OF AN AUDIO FILE ////////////

    //rtMatch("C:\\Users\\ADMIN\\Desktop\\dataset\\novos_cortes_tudo\\test1.wav");

    //rtMatch("C:\\Users\\ADMIN\\Desktop\\dataset\\novos_cortes_tudo\\test2.wav");

  auto stop = high_resolution_clock::now();
  auto duration = duration_cast<seconds>(stop - start);
  cout << endl << "Ended!!" << endl << "Operation took " << duration.count()
    << " seconds to perform" << endl;
  getchar();
  return 0;
}
```

main.cpp

References

[1] M. Jalal. *Available at* https://qph.fs.quoracdn.net/main-qimg-667ef28b0bfb903b1af6452aff6a247e.webp, 2017. [Accessed January 30, 2018].

[2] E. Karami, S. Prasad, and M. S. Shehata, "Image matching using sift, surf, BRIEF and ORB: performance comparison for distorted images," *CoRR*, vol. abs/1710.02726, 2017.

[3] P. Cano, E. Batlle, T. Kalker, and J. Haitsma, "A review of audio fingerprinting," *J. VLSI Signal Process. Syst.*, vol. 41, pp. 271–284, Nov. 2005.

[4] "abcd". *Available at* https://stackoverflow.com/questions/5887366/matlab-spectrogram-params, 2011. [Accessed January 30, 2018].

[5] A. Ramires, D. Cocharro, and M. Davies, "An audio-only method for advertisement detection in broadcast television content," in *23rd Portuguese Conference on Pattern Recognition (RECPAD)*, pp. 21–22, 2017.

[6] S. A. Papert, "The summer vision project," 1966.

[7] Y. Ke, D. Hoiem, and R. Sukthankar, *Computer vision for music identification*, vol. I, pp. 597–604. 2005.

[8] J. Haitsma and T. Kalker, "A highly robust audio fingerprinting system.," in *Ismir*, vol. 2002, pp. 107–115, 2002.

[9] P. Viola and M. Jones, "Rapid object detection using a boosted cascade of simple features," in *Proceedings of the 2001 IEEE Computer Society Conference on Computer Vision and Pattern Recognition. CVPR 2001*, vol. 1, pp. I–511–I–518 vol.1, 2001.

[10] D. G. Lowe, "Distinctive image features from scale-invariant keypoints," *Int. J. Comput. Vision*, vol. 60, pp. 91–110, Nov. 2004.

[11] H. Bay, A. Ess, T. Tuytelaars, and L. V. Gool, "Speeded-up robust features (surf)," *Computer Vision and Image Understanding*, vol. 110, no. 3, pp. 346 – 359, 2008. Similarity Matching in Computer Vision and Multimedia.

[12] D. G. Lowe, "Object recognition from local scale-invariant features," in *Proceedings of the Seventh IEEE International Conference on Computer Vision*, vol. 2, pp. 1150–1157 vol.2, 1999.

[13] M. A. Fischler and R. C. Bolles, "Random sample consensus: A paradigm for model fitting with applications to image analysis and automated cartography," *Commun. ACM*, vol. 24, pp. 381–395, June 1981.

[14] M. Muja and D. G. Lowe, “Fast approximate nearest neighbors with automatic algorithm configuration,” in *In VISAPP International Conference on Computer Vision Theory and Applications*, pp. 331–340, 2009.

[15] P. Cano, E. Batlle, E. Gómez, L. de C.T.Gomes, and M. Bonnet, *Audio Fingerprinting: Concepts And Applications*, pp. 233–245. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005.

[16] G. Heinzel, A. Rüdiger, and R. Schilling, “Spectrum and spectral density estimation by the discrete fourier transform (dft), including a comprehensive list of window functions and some new at-top windows,” 2002.

[17] W. T. Cochran, J. W. Cooley, D. L. Favin, H. D. Helms, R. A. Kaenel, W. W. Lang, G. C. Maling, D. E. Nelson, C. M. Rader, and P. D. Welch, “What is the fast fourier transform?,” *Proceedings of the IEEE*, vol. 55, pp. 1664–1674, Oct 1967.

[18] N. Ahmed, T. Natarajan, and K. R. Rao, “Discrete cosine transfom,” *IEEE Trans. Comput.*, vol. 23, pp. 90–93, Jan. 1974.

[19] S. R. Subramanya, R. Simha, B. Narahari, and A. Youssef, “Transform-based indexing of audio data for multimedia databases,” in *ICMCS*, 1997.

[20] M. K. Mihçak and R. Venkatesan, “A perceptual audio hashing algorithm: A tool for robust audio identification and information hiding,” in *Proceedings of the 4th International Workshop on Information Hiding*, IHW ’01, (London, UK, UK), pp. 51–65, Springer-Verlag, 2001.

[21] P. Cano, E. Batlle, H. Mayer, and H. Neuschmied, “Robust sound modeling for song detection in broadcast audio,” in *in Proc. AES 112th Int. Conv*, pp. 1–7, 2002.

[22] T. L. Blum, D. F. Keislar, J. A. Wheaton, and E. H. Wold, “Method and article of manufacture for content-based analysis, storage, retrieval, and segmentation of audio information,” June 29 1999. US Patent 5,918,223.

[23] B. Logan *et al.*, “Mel frequency cepstral coefficients for music modeling.,” in *ISMIR*, vol. 270, pp. 1–11, 2000.

[24] C. Yang, “Macs: music audio characteristic sequence indexing for similarity retrieval,” in *Proceedings of the 2001 IEEE Workshop on the Applications of Signal Processing to Audio and Acoustics (Cat. No.01TH8575)*, pp. 123–126, 2001.

[25] A. Wang *et al.*, “An industrial strength audio search algorithm.,” in *Ismir*, vol. 2003, pp. 7–13, Washington, DC, 2003.

[26] J. P. Ogle and D. P. Ellis, “Fingerprinting to identify repeated sound events in long-duration personal audio recordings,” in *Acoustics, Speech and Signal Processing, 2007. ICASSP 2007. IEEE International Conference on*, vol. 1, pp. I–233, IEEE, 2007.

[27] E. Allamanche, J. Herre, O. Hellmuth, B. Fröba, T. Kastner, and M. Cremer, “Content-based identification of audio material using mpeg-7 low level description.,” in *ISMIR*, 2001.

[28] X. Anguera, A. Garzon, and T. Adamek, “Mask: Robust local features for audio fingerprinting,” in *2012 IEEE International Conference on Multimedia and Expo*, pp. 455–460, July 2012.

[29] J. Herre, E. Allamanche, and O. Hellmuth, "Robust matching of audio signals using spectral flatness features," in *Proceedings of the 2001 IEEE Workshop on the Applications of Signal Processing to Audio and Acoustics (Cat. No.01TH8575)*, pp. 127–130, Oct 2001.

[30] A. Ramalingam and S. Krishnan, "Gaussian mixture modeling of short-time fourier transform features for audio fingerprinting," *IEEE Transactions on Information Forensics and Security*, vol. 1, pp. 457–463, Dec 2006.

[31] J. S. Seo, M. Jin, S. Lee, D. Jang, S. Lee, and C. D. Yoo, "Audio fingerprinting based on normalized spectral subband centroids," in *Proceedings. (ICASSP '05). IEEE International Conference on Acoustics, Speech, and Signal Processing, 2005.*, vol. 3, pp. iii/213–iii/216 Vol. 3, March 2005.

[32] E. Batlle, J. Masip, and E. Guaus, "Automatic song identification in noisy broadcast audio," in *IASTED International Conference on Signal and Image Processing*, vol. 2, p. 2, 2002.

[33] F. Kurth, A. Ribbrock, and M. Clausen, "Identification of highly distorted audio material for querying large scale databases," in *In Proc. 112th AES Convention*, 2002.

[34] J. W. Picone, "Signal modeling techniques in speech recognition," *Proceedings of the IEEE*, vol. 81, pp. 1215–1247, Sep 1993.

[35] G. Richly, L. Varga, F. Kovacs, and G. Hosszu, "Short-term sound stream characterization for reliable, real-time occurrence monitoring of given sound-prints," in *2000 10th Mediterranean Electrotechnical Conference. Information Technology and Electrotechnology for the Mediterranean Countries. Proceedings. MeleCon 2000 (Cat. No.00CH37099)*, vol. 2, pp. 526–528 vol.2, 2000.

[36] E. Batlle, J. Masip, E. Guaus, and P. Cano, "Scalability issues in hmm-based audio fingerprinting," in *International Conference on Multimedia and Expo*, vol. 1, (Taipei, Taiwan), pp. 735–738, 27/06/2004 2004.

[37] J. Deng, W. Wan, X. Yu, and W. Yang, "Audio fingerprinting based on spectral energy structure and nmf," in *2011 IEEE 13th International Conference on Communication Technology*, pp. 1103–1106, Sept 2011.

[38] N. Chen, H. D. Xiao, and W. Wan, "Audio hash function based on non-negative matrix factorisation of mel-frequency cepstral coefficients," *IET Information Security*, vol. 5, pp. 19–25, March 2011.

[39] C. Févotte, N. Bertin, and J.-L. Durrieu, "Nonnegative matrix factorization with the itakura-saito divergence: With application to music analysis," *Neural computation*, vol. 21, no. 3, pp. 793–830, 2009.

[40] J. Haitsma and T. Kalker, "A highly robust audio fingerprinting system.," in *Ismir*, vol. 2002, pp. 107–115, 2002.

[41] R. A. K.-l. Lin and H. S. S. K. Shim, "Fast similarity search in the presence of noise, scaling, and translation in time-series databases," in *Proceeding of the 21th International Conference on Very Large Data Bases*, pp. 490–501, Citeseer, 1995.

[42] T. Kastner, E. Allamanche, J. Herre, O. Hellmuth, M. Cremer, and H. Grossmann, "Mpeg-7 scalable robust audio fingerprinting," in *Audio Engineering Society Convention 112*, Apr 2002.

[43] S. Baluja and M. Covell, "Audio fingerprinting: Combining computer vision data stream processing," in *2007 IEEE International Conference on Acoustics, Speech and Signal Processing - ICASSP '07*, vol. 2, pp. II–213–II–216, April 2007.

[44] D. Jang, C. D. Yoo, S. Lee, S. Kim, and T. Kalker, "Pairwise boosted audio fingerprint," *IEEE Transactions on Information Forensics and Security*, vol. 4, pp. 995–1004, Dec 2009.

[45] S. Bilobrov and I. Poutivski, "Commercial detection based on audio fingerprinting," Mar. 9 2017. US Patent App. 15/357,683.

[46] R. Sonnleitner and G. Widmer, "Robust quad-based audio fingerprinting," *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, vol. 24, pp. 409–421, March 2016.

[47] S. Reid, B. Guenter, V. Diels-Grabsch, and E. Hopkinson, "Sha-1 implementation in c++," 2013. https://github.com/vog/sha1.

[48] P. FIPS, "180-1. secure hash standard," *National Institute of Standards and Technology*, vol. 17, p. 45, 1995.

[49] R. Hipp, "Sqlite," Aug 2017. https://www.sqlite.org/cgi/src/info/8d3a7ea6c5690d6b.

[50] F. Bellard, "Ffmpeg," 2018. https://ffmpeg.zeranoe.com/builds/.

[51] M. Piacentini, "Db browser for sqlite," 2017. https://github.com/sqlitebrowser/sqlitebrowser.