UNIVERSIDADE REGIONAL DE BLUMENAU

CENTRO DE CIÊNCIAS EXATAS E NATURAIS

CURSO DE CIÊNCIA DA COMPUTAÇÃO – BACHARELADO

MECANISMO DE BUSCA SEMÂNTICA DE ÁUDIO

ANDERSON DOROW

BLUMENAU
2011

2011/1-05

ANDERSON DOROW

MECANISMO DE BUSCA SEMÂNTICA DE ÁUDIO

Trabalho de Conclusão de Curso submetido à Universidade Regional de Blumenau para a obtenção dos créditos na disciplina Trabalho de Conclusão de Curso II do curso de Ciência da Computação — Bacharelado.

Prof. Aurélio Faustino Hoppe, Mestre - Orientador

BLUMENAU
2011

2011/1-05

MECANISMO DE BUSCA SEMÂNTICA DE ÁUDIO

Por

ANDERSON DOROW

Trabalho aprovado para obtenção dos créditos na disciplina de Trabalho de Conclusão de Curso II, pela banca examinadora formada por:

Presidente:	Prof. Aurélio Faustino Hoppe, Mestre – Orientador, FURB
Membro:	Prof. Fernando dos Santos, Mestre – FURB
Membro:	Prof. Roberto Heinzle, Doutor – FURB

Blumenau, 29 de junho de 2011

Dedico este trabalho a minha família e amigos, especialmente àqueles que me ajudaram, direta ou indiretamente, na realização deste.

AGRADECIMENTOS

A Deus, pelo seu imenso amor e graça.

À minha família, que sempre me deu forças.

Aos meus amigos, e principalmente minha namorada, Suria, por terem tido paciência e entenderem os meus momentos de ausência.

Ao meu orientador, Aurélio Faustino Hoppe, pois sua insistência em revisar e reavaliar o desenvolvimento deste trabalho ajudaram a extrair o melhor de mim na realização do mesmo.

Quanto mais brilhante você é, tão mais você tem a aprender.

Don Herold

RESUMO

Este trabalho descreve o desenvolvimento de um mecanismo de busca de áudio a partir do seu conteúdo. O mecanismo obtém/recebe um trecho de áudio, aplica a transformada de Fourier para extrair características do áudio, gera *audio fingerprints,* submete-os ao servidor para encontrar *audio fingerprints* iguais. Ao final deste processo, tem-se uma lista das correspondências, aos quais são usados métodos estatísticos para definir o grau de similaridade entre o áudio de entrada e os existentes na base. Os resultados demonstram que as técnicas de similaridades implementadas são suficientes para obter resultados positivos, inclusive na presença de ruídos e, que o tamanho do trecho de entrada tem influencia direta sobre a eficiência do mecanismo da busca.

Palavras-chave: Recuperação de informação musical. *Audio fingerprinting*. Busca baseada em conteúdo.

ABSTRACT

This paper describes the development of an audio search engine based on its content. The mechanism obtains/receive an audio snippet, apply the Fourier transform to extract features from the audio, generates audio fingerprints, submit them to the server to find audio fingerprint matches. At the end of this process, there is a list of matches, which statistical methods are to define the degree of similarity between the audio input and those in the base. The results show that the similarity techniques implemented are sufficient to obtain positive results, even in the presence of noise and, the size of the input has influence in the efficiency of the search mechanism.

Key-words: Musical information retrieval. Audio fingerprinting. Content-based search.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE SIGLAS

API – *Application Programming Interface*

CD – *Compact Disc*

DFT – *Discrete Fourier Transform*

DSP – *Digital Signal Processing*

DTFT – *Discrete Time Fourier Transform*

FFT – *Fast Fourier Transform*

IDE – *Integrated Development Environment*

MIR – *Music Information Retrieval*

MP3 – MPEG-1/2 *Audio Layer* 3

MPEG – *Moving Picture Experts Group*

RI – Recuperação de Informação

SUMÁRIO

1 INTRODUÇÃO

O uso de sistemas de informática tem sido democratizado de forma crescente e vem facilitando muito a vida e o dia-a-dia de todos. Através deste novo meio de comunicação foi se criando um ambiente cooperativo de livre disseminação de informações que culminou na grande rede mundial que hoje é chamada de Internet.

O rápido crescimento da Internet trouxe consigo uma enorme gama de informações. Inicialmente, a organização destas informações era pouca ou nenhuma, pois essas ficavam espalhadas pela rede. Mas, este crescimento deu-se de forma explosiva, trazendo o grande problema de como localizar algo em um ambiente tão desorganizado. Assim, não restava outro caminho senão catalogá-las, utilizando bancos de dados com o objetivo de realizar buscas eficientes e rápidas. Surgem então os mecanismos de busca, a exemplo do Google e Yahoo. Todavia, tais mecanismos de busca e recuperação de informação são focados em dados textuais. Ou seja, o conteúdo dos arquivos são descritos por palavras e os mecanismos são projetados para recuperá-los através delas. Porém, para o caso de arquivos de áudio, onde não se consegue descrever precisamente o conteúdo a ser buscado através de texto, este tipo de busca passa a gerar resultados pouco relevantes quando o tamanho da base de áudios começa a ficar maior (CASEY et al., 2008, p. 669).

Assim como documentos de texto são formados por palavras e sentenças, arquivos de áudio são constituídos de diversas propriedades que os caracterizam em relação aos demais, tais como frequência, amplitude e comprimento de onda. Para atingir resultados mais relevantes na busca de áudios, faz-se necessária a utilização de uma nova estratégia de recuperação de informação, direcionada a busca de dados não-textuais que leve em consideração informações encontradas no próprio arquivo sonoro.

Diante do exposto, observou-se uma demanda por um mecanismo que possa auxiliar tal tipo de busca, ou seja, um mecanismo que seja capaz de identificar uma música, ou conteúdo de áudio, que o usuário esteja buscando, utilizando para isso apenas um trecho reproduzido pelo próprio usuário. Tal mecanismo seria análogo aos sistemas de busca correntes, porém aplicado ao domínio de áudio.

Assim, neste trabalho desenvolveu-se um mecanismo de busca baseado no conteúdo do áudio. No mecanismo desenvolvido, inicialmente informa-se um trecho de áudio como entrada e, a partir disso, são extraídas características do áudio e os *audio fingerprints*, que posteriormente são submetidas a uma base para comparação, retornando os áudios com

trechos semelhantes.

1.1 OBJETIVOS DO TRABALHO

O objetivo deste trabalho é desenvolver um mecanismo de busca baseado no conteúdo do áudio.

Os objetivos específicos do trabalho são:

a) possibilitar que o usuário grave trechos de áudio e os utilize como entrada de dados no mecanismo de busca;
b) gerar um modelo de representação baseado nas características estruturais do áudio (*audio fingerprints*);
c) implementar técnicas de similaridade para identificar áudios em uma base de dados multimídia (áudio), a partir do trecho gravado pelo usuário.

1.2 ESTRUTURA DO TRABALHO

Este trabalho está estruturado em quatro capítulos. O segundo capítulo contém a fundamentação teórica necessária para o desenvolvimento do trabalho. Nele são discutidos tópicos relacionados a mecanismos de busca baseados em conteúdo, *Audio Fingerprinting*, técnicas de similaridade e a transformada de Fourier. Por último, são apresentados três trabalhos correlatos.

O terceiro capítulo trata do desenvolvimento do mecanismo, onde são relacionados seus requisitos, bem como explicando a especificação dele através de diagramas de caso de uso, de classe, de sequência e a modelagem da base de dados. Também são feitos comentários sobre a implementação, apresentando técnicas e ferramentas utilizadas, operacionalidade e, por fim, são apresentados e discutidos os resultados obtidos.

O quarto capítulo refere-se às conclusões do presente trabalho e sugestões para trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

Neste capítulo são descritos os conteúdos pesquisados para auxiliar o desenvolvimento do mecanismo de busca. A seção 2.1 descreve o funcionamento de mecanismos de busca baseados em conteúdo. Na seção 2.2 é apresentada a técnica de *audio fingerprinting* e sua importância no desenvolvimento deste trabalho. Na seção 2.3 é discutido sobre técnicas de similaridade. Na seção 2.4 é apresentada a transformada de Fourier. Por fim, são apresentados três trabalhos correlatos.

2.1 MECANISMOS DE BUSCA BASEADOS EM CONTEÚDO

Hoje, uma tendência de desenvolvimento da área de recuperação de informação (RI) é a especialização dos mecanismos de buscas, levando em conta, por exemplo, aspectos semânticos do tema que constitui o espaço de busca. Relacionado a isso, há também um grande interesse em expandir as fronteiras da ciência de RI para outros tipos de conteúdo, como áudio e vídeo.

Os mecanismos de busca atuais efetuam buscas baseadas na similaridade de palavras-chave. Para diversos usuários é difícil descrever uma pesquisa de maneira textual, especialmente quando eles estão procurando por algum documento multimídia em uma base de dados multimídia. A busca baseada em conteúdo é considerada como uma solução efetiva para este problema (MALIK, 2002, p. 1).

Um mecanismo de busca de áudio baseado em conteúdo é um sistema de busca que se utiliza de propriedades contidas nos arquivos de áudio, de modo a classificá-los e posteriormente identificá-lo através de um trecho do mesmo.

Segundo Casey et al. (2008, p. 670), os principais componentes de um sistema de *Music Information Retrieval* (MIR) são:

a) formação da *query*[1];
b) extração da descrição do áudio;
c) comparação;

[1] Termo de entrada da pesquisa

d) recuperação do arquivo de áudio.

Um MIR funciona da seguinte maneira: um trecho de áudio é dado como entrada para formar a *query* (item a); propriedades são extraídas do áudio para gerar uma "identificação" (item b). Este processo, também é executado previamente em todos os áudios do repositório de dados. Com as propriedades extraídas, a *query* é submetida a um repositório, para que possíveis resultados sejam retornados, resultados referentes à mesma informação são agrupados e submetidos a uma comparação mais aguçada (item c), sendo possível identificar os resultados mais relevantes, que são então listados como saída da pesquisa (item d).

2.2 AUDIO FINGERPRINTING

O objetivo do *fingerprinting* é prover métodos rápidos e confiáveis para identificação de conteúdo (KALKER et al., 2001, p. 190).

> Sistemas de identificação já possuem mais de cem anos de idade. Em 1893, Sir Francis Galton foi o primeiro a 'provar' que não existem duas digitais humanas iguais. Aproximadamente dez anos depois a Scotland Yard aceitou um sistema desenvolvido por Sir Edward Henry para identificar digitais de pessoas. (HAITSMA; KALKER, 2002, p. 107).

De acordo com Haitsma e Kalker (2002, p. 107), um *audio fingerprint* pode ser visto como um pequeno resumo de um objeto de áudio. Um sistema de busca baseado em conteúdo pode utilizar-se de *fingerprints* para fazer o armazenamento do áudio em uma base de dados, tendo em vista que os *fingerprints* tendem a ocupar o mesmo espaço que um arquivo de áudio normal.

Segundo Wang (2003, p. 7), para gerar os *fingerprints* são executados os seguintes passos:

a) analisa-se a música, dividindo-a em um espectro de tempo vs. freqüência (obtida através da transformada de Fourier, detalhada na seção 2.4);
b) são marcados os pontos de máximo local de frequência. A opção do uso dos máximos de frequência ocorre pois tem mais chance de sobreviver a ruídos;
c) os pontos máximos escolhidos têm sua quantidade limitada por área, de forma que sejam distribuídos uniformemente;
d) criam-se *hashes* combinatoriais, ou seja, estes pontos são analisados dentro de áreas pré-determinadas. Com isso localiza-se um ponto âncora para todo o

espectro;

e) estas âncoras são associadas com cada um dos pontos de dentro da área verificada. Cada par obtido contém a frequência de ambos os componentes e a diferença de tempo entre eles. As *hashes* são números de 32 bits extraídas das informações de cada um destes pares;
f) além das *hashes*, associa-se também seu tempo relativo ao começo da música e a seu identificador, isso resulta em 64 bits para cada *hash*. Estes tipos de *hashes* são resistentes mesmo com ruídos e distorções provocadas pelos *codecs* de voz.

No item (a), é realizada a divisão da música em faixas de tempo pelo fato de que, se fosse utilizada a música inteira, o *hash* gerado seria muito difícil de obter a partir de uma música que fosse muito semelhante. A divisão da música em faixas de tempo faz com que, se algumas das faixas ficarem muito diferente (devido a um ruído, por exemplo), a música ainda possa ser identificada a partir dos outros *hashes*.

Os *fingerprints* possuem dados retirados da própria informação musical, onde é possível analisar os níveis de similaridade entre elas, permitindo que sejam localizadas semelhantes. Ao utilizar estes recursos, é possível realizar uma análise automatizada e robusta de áudios contidos em um servidor. Por este motivo, *audio fingerprints* são consideradas uma ferramenta eficiente para ser utilizada no tipo de mecanismo desenvolvido neste trabalho, e por isso também foi definida a sua utilização no mesmo.

2.3 TÉCNICAS DE SIMILARIDADE

As funções de similaridade funcionam sobre a representação criada para o áudio. Se a representação não for precisa, então a medida de similaridade será correspondentemente imprecisa (WIGNALL, 2003, p. 5). Ou seja, uma técnica de similaridade mal aplicada pode gerar dois tipos de problemas:

a) falsos positivos: a técnica pode identificar erroneamente um trecho de áudio como sendo parte do resultado da busca;
b) falsa rejeição: a técnica pode descartar um trecho de áudio que faz parte do resultado esperado da busca.

De acordo com Cano (2006, p. 53), as medidas de similaridade estão muito relacionadas ao tipo de modelo escolhido para representar a informação desejada. A distância

euclidiana, ou versões modificadas dela que lidam com sequências de diferentes tamanhos podem ser utilizadas na comparação de sequências de valores. Para comparação de *fingerprints*, Cano (2006, p. 54) diz que pode ser utilizado um paradigma de correspondência de modelos quando os *fingerprints* do repositório e o *fingerprint* extraído de outro áudio qualquer possuírem o mesmo formato. Uma medida de similaridade que ele cita neste caso é a distância de Hamming.

De acordo com Cano (2006, p. 54-56), um problema de usabilidade na identificação de similaridades de *fingerprints* é a eficiência das comparações realizadas, pelo fato de poder haver um número grande de *fingerprints* para se realizar comparações. Para tanto, uma solução de força-bruta não é suficiente. Ele apresenta algumas técnicas que podem ser utilizadas como solução para reduzir a complexidade, e tornar o processo de comparação mais eficiente:

a) pré-cálculo de distâncias *off-line*: não é possível pré-calcular os *fingerprints* da busca, pois ela ainda não está presente no repositório. Porém, é possível pré-calcular distâncias entre as *fingerprints* registradas no repositório e montar uma estrutura de dados de modo a reduzir o número de verificações de similaridade quando a *query* é apresentada;
b) filtrar candidatos improváveis com uma medida de similaridade mais barata: outra possibilidade é utilizar uma medida de similaridade mais barata para eliminar rapidamente muitos candidatos, deixando que uma técnica mais robusta seja utilizada posteriormente apenas nos candidatos mais prováveis;
c) poda de candidatos: uma otimização simples consiste de cancelar a verificação por similaridade quando se percebe que, do ponto atual, não é mais possível obter uma pontuação para o nível de similaridade que seja melhor do que o que já foi obtido até então;
d) busca primeiro em sons mais populares: o repositório poderia ser dividido em duas partes, uma contendo os áudios mais populares, e outro contendo os áudios menos populares. Primeiro é realizada uma busca no repositório de áudios populares, que são normalmente mais buscados, e, somente se não houver sucesso, é efetuada uma busca no repositório de áudios menos populares.

As técnicas citadas acima servem para otimizar o processo de identificação de similaridade de *fingerprints*. Uma ou mais destas técnicas podem ser utilizadas ao mesmo tempo, e também em conjunto com outras técnicas de comparação numa busca de similaridades.

2.4 TRANSFORMADA DE FOURIER

O áudio, em seu formato mais bruto (conhecido como domínio do tempo), é uma sequência de variações da pressão do ar obtidas a intervalos de tempo regulares (BUNNELL, 1996b). Porém, com apenas este tipo de informação, não é possível extrair as principais características do áudio. Para isto, é necessário que se tenha o áudio no que é conhecido como domínio da frequência. O termo "domínio da frequência" é utilizado para descrever as amplitudes das ondas de seno e cosseno (BUNNELL, 1996a). Uma maneira de executar a transformação entre domínio do tempo e domínio da frequência é através da transformada de Fourier.

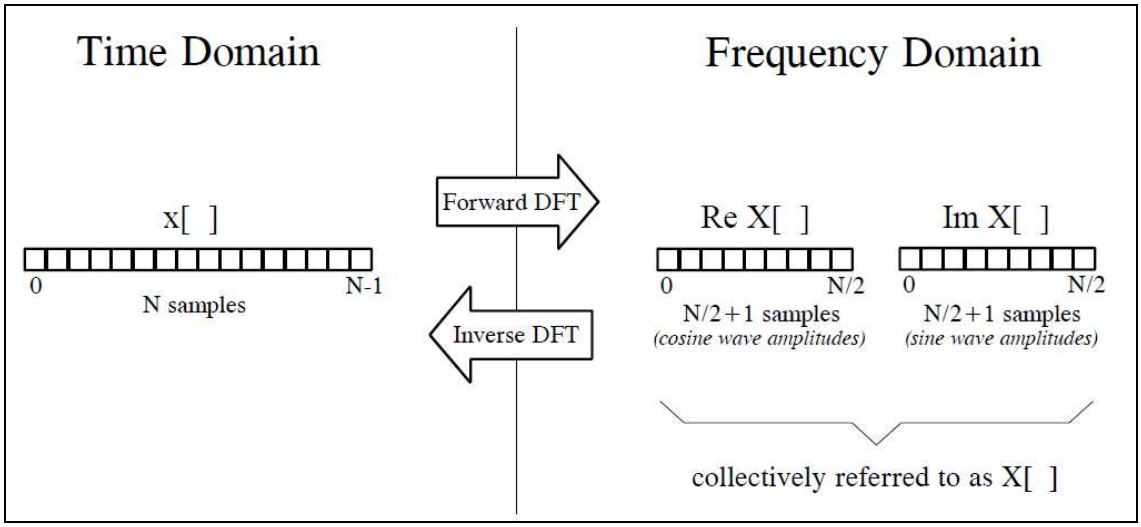
De acordo com Smith(1997, p. 142-145), o termo geral "transformada de Fourier", pode ser quebrado em quatro categorias, resultando em quatro tipos básicos de sinais que podem ser encontrados. Um sinal pode ser contínuo ou discreto, e pode ser também periódico ou aperiódico. A combinação destas duas características gera as quatro categorias.

a) aperiódico-contínuo: inclui, por exemplo, decaimento exponencial e curva Gaussiana. Os sinais estendem-se a positivo e negativo infinito, sem repetições num padrão periódico. A transformada de Fourier para este tipo de sinal é chamada simplesmente de "Transformada de Fourier";
b) periódico-contínuo: inclui: ondas de seno e cosseno, e qualquer forma de onda que se repete em um padrão regular, de negativo a positivo infinito. Esta versão da transformada de Fourier denomina-se Série de Fourier;
c) aperiódico-discreto: estes sinais são os únicos definidos em pontos discretos entre positivo e negativo infinito, e não se repetem de maneira periódica. Este tipo de transformada de Fourier é chamado de Transformada de Fourier de tempo discreto (DTFT);
d) periódico-discreto: estes são sinais discretos que se repetem de maneira periódica, de negativo a positivo infinito. Esta classe de transformada de Fourier é às vezes chamada de série discreta de Fourier, mas é mais conhecida por transformada discreta de Fourier (DFT).

Estas quatro classes de sinais se estendem ao infinito positivo e negativo. Sendo assim, se há uma amostra finita, como por exemplo, um sinal formado por 1024 pontos, não é possível aplicar nele nenhuma das transformadas citadas anteriormente. Não se pode utilizar um grupo de sinais infinitamente longos para sintetizar algo finito em tamanho. A maneira de

contornar isto é fazer com que os dados finitos pareçam um sinal de tamanho infinito. Isso é feito imaginando que o sinal possui um número infinito de amostras à esquerda e à direta dos pontos amostrados. Se todos estes pontos "imaginados" possuírem valor zero, o sinal irá parecer discreto e aperiódico, e então poderia se aplicar a DTFT. Como alternativa, as amostras imaginadas podem ser uma duplicação dos atuais 1024 pontos. Neste caso, o sinal parecerá discreto e periódico, com um período de 1024 amostras. Neste caso é necessário que a transformada discreta de Fourier seja utilizada. Como um número infinito de sinusoidais seria necessário para sintetizar um sinal que é aperiódico, seria impossível calcular a Transformada de Fourier de tempo discreto. Então, por eliminação, o único tipo de transformada de Fourier que pode ser utilizada é a DFT.

Segundo Smith(1997, p. 146-147), a transformada de Fourier altera um sinal de entrada de `N` amostras em dois sinais de saída de `N/2+1` amostras. O sinal de entrada contém o sinal sendo decomposto (domínio do tempo), que é composto, normalmente, por amostras deste sinal obtidas a intervalos regulares de tempo, enquanto os dois sinais de saída contêm as amplitudes das ondas seno e cosseno (domínio da frequência). A Figura 1 apresenta em alto nível a transformação realizada.



Fonte: Smith (1997, p. 147).

Figura 1 – Entrada e saída da DFT

Como pode ser visto na Figura 1, `x` representa o sinal no domínio do tempo, que é formado por `N` amostras (de `0` a `N-1`), enquanto `Re X` e `Im X` representam a parte real e a parte imaginária do sinal no domínio da frequência, que são formados por `N/2+1` amostras (de `0` a `N/2`). A Figura 1 apresenta também duas flechas, que indicam aplicações da transformada, pois além da transformada normal, existe a inversa da transformada, que realiza o processo contrário (transforma do domínio da frequência para o domínio do tempo). A utilização da transformada inversa não é de interesse neste trabalho, por este motivo é descrita apenas a

transformada normal.

Segundo Smith (1997, p. 147), o domínio da frequência possui exatamente a mesma informação que o domínio de tempo, porém de uma maneira diferente. Se um domínio é conhecido, o outro pode ser calculado. Utilizando a transformada de Fourier obtém-se o domínio da frequência a partir do domínio do tempo, e a partir da transformada inversa obtém-se o domínio do tempo a partir do domínio da frequência. A transformada pode ser representada na forma de equação. A Figura 2 apresenta a fórmula da transformada de Fourier.

$$ReX[k] = \sum_{i=0}^{N-1} x[i] \cos(2\pi k i / N)$$

$$ImX[k] = -\sum_{i=0}^{N-1} x[i] \sin(2\pi k i / N)$$

Fonte: Smith (1997, p. 158).

Figura 2 - Fórmula da DFT

Conforme pode ser visto nestas equações apresentadas na Figura 2, `x[i]` representa o sinal do domínio do tempo sendo analisado, enquanto `Re X[k]` e `Im X[k]` representam os sinais do domínio da frequência que estão sendo calculados. O índice `i` vai de `0` até `N-1`, enquanto o índice `k` vai de `0` a `N/2`.

Ao final deste processo, tem-se finalmente a informação no domínio da frequência. No caso deste trabalho, esta informação representa as frequências de um determinado trecho de áudio, a partir das quais se podem obter as características (ver seção 2.2).

2.5 TRABALHOS CORRELATOS

Nos últimos anos surgiram diversos estudos relacionados a busca de áudio baseada em conteúdo, onde alguns desses tornaram-se *softwares* comercializados. Normalmente estes estudos focam em características específicas do áudio, para que o algoritmo seja mais eficiente em situações específicas. Alguns trabalhos que podem ser citados são: o método desenvolvido por Kline e Glinert (2003), o método baseado em comparação de matrizes de similaridade, desenvolvido por Izumitani e Kashino (2008), e também o *software* comercial Shazam, descrito por Wang (2003).

Em Kline e Glinert (2003), é apresentado um método de busca baseado na técnica de *query-by-humming*, onde o usuário deve cantar, cantarolar ou assoviar um trecho de música que se deseja buscar. É uma alternativa interessante, sendo que depende apenas da memória do usuário para se efetuar uma busca, porém, a reprodução humana é muito suscetível a erros, o que pode gerar um grande número de falsos positivos[2] utilizando este tipo de pesquisa. Kline e Glinert (2003) desenvolveram um algoritmo baseado em técnicas de comparação por aproximação, com isso foi possível buscar músicas que tenham sequências similares às reproduzidas pelo usuário.

Izumitani e Kashino (2008) propuseram um método que permitisse a busca e identificação de músicas que possuíssem variações de notas e velocidade. Para tanto são extraídas características dos áudios em determinados intervalos e a partir delas são geradas matrizes de auto-similaridade (*self-similarity matrices*). Uma técnica de programação dinâmica baseada no algoritmo de maior subseqüência crescente é utilizada para efetuar a comparação das características, onde, para cada comparação realizada, é gerada uma pontuação, caso trechos semelhantes sejam encontrados.

Wang (2003) descreve o Shazam como um serviço de reconhecimento de músicas, capaz de reconhecer um trecho de áudio, mesmo na presença de ruídos e reverberação. Para cada arquivo de áudio são gerados *fingerprints*, onde estão armazenadas características de trechos dos áudios. Estes *fingerprints* são gerados também para um arquivo de entrada. Os *fingerprints* do áudio de entrada são comparados aos de uma base, que por sua vez retorna os resultados correspondentes. Nestes resultados, são verificadas as músicas que possuem mais ocorrências, e estas são analisadas mais assiduamente para identificar a saída correta, que possui um trecho contínuo de música identificado.

O quadro 1 mostra de forma resumida as principais características destes sistemas, tendo como base critérios extraídos a partir dos conceitos descritos no decorrer desta seção. As informações foram dispostas em colunas, representando na vertical os trabalhos analisados e as linhas apresentam as características de cada sistema, indicando semelhanças e/ou diferenças.

[2] Uma contagem de acerto identificada pelo algoritmo, mas que não faz parte do resultado final esperado.

características / trabalhos relacionados	Kline e Glinert (2003)	Izumitami e Kashino (2008)	Wang (2003)
utiliza reprodução humana da música, baseada na memória do usuário, como entrada de pesquisas	X	-	-
utiliza trecho de uma gravação musical como entrada de pesquisas	-	X	X
resultados da busca aceitam variações na música (como velocidade)	-	X	-
bons resultados de pesquisa na presença de ruídos	-	-	X
utilização de *Audio fingerprints* para a representação do áudio	-	-	X
utilização de sequências (que possuem características do áudio) para representação do áudio	X	X	-
solução de código livre (implementação de referência)	-	-	-
possibilidade de uma base colaborativa	-	-	-

Quadro 1 - Características dos trabalhos relacionados

A partir do quadro 1, pode-se observar que existem divergências entre as propostas de cada autor. Cada trabalho é desenvolvido tendo em vista características e objetivos específicos. Há ferramentas que fazem buscas baseadas numa entrada gerada por reprodução do próprio usuário, que facilitam a geração de entrada, porém também costumam aumentar o número de falso positivos, e outras que utilizam gravações de trechos reais de música, que apesar de serem um pouco mais trabalhosas de se obter, possuem representação igual ou semelhante ao resultado que se busca obter, agilizando a pesquisa. O software Shazam (WANG, 2003) teve como uma de suas preocupações o tratamento de ruídos, por este motivo consegue obter bons resultados em buscas que os possuem, situação que já dificulta a pesquisa nos casos de Kline et al. (2003) e Izumitani et al. (2008).

Através dos trabalhos relacionados, é possível verificar que diversos trabalhos já foram desenvolvidos nesta área, chegando a gerar inclusive softwares comercias, como o Shazam (WANG, 2003), porém, ainda assim percebe-se que algumas lacunas ainda precisam ser preenchidas. Vê-se, por exemplo, a falta de soluções de código livre, que poderia ser um ponto de partida para novas soluções, e referência para quem precisa iniciar estudos no tema. Outro ponto percebido, é que as soluções encontradas atualmente, permitem apenas a busca de áudios através de seus algoritmos, mas não permitem que um usuário colabore com a inclusão de novos áudios na base, esse trabalho é feito sempre pelo próprio desenvolvedor.

Para suprir tais lacunas/limitações, neste trabalho, desenvolveu-se um mecanismo de busca guiado por estes requisitos e, comumente sem fins comerciais (*open source*).

3 DESENVOLVIMENTO

Neste capítulo são apresentadas as etapas para o desenvolvimento do mecanismo proposto. Na seção 3.1 é apresentado o levantamento de requisitos. Na seção 3.2 é realizada a especificação do trabalho, seguido da implementação do mecanismo na seção 3.3. Por fim, na seção 3.4 são discutidos os resultados obtidos através dos experimentos realizados.

3.1 REQUISITOS PRINCIPAIS DO PROBLEMA A SER TRABALHADO

O mecanismo de busca semântica proposto deverá:

a) armazenar representações de áudio em uma base de dados (Requisito Funcional – RF);
b) permitir que o usuário grave um trecho de áudio com 5, 10 ou 15 segundos, servindo de entrada para o mecanismo de busca (RF);
c) estabelecer relações espaciais, identificando elementos de interesse a partir da estrutura do áudio (RF);
d) utilizar métodos estatísticos para estabelecer o grau de similaridade entre o áudio de entrada e os existentes na base de dados (RF);
e) identificar dentro de um repositório limitado qual música corresponde ao trecho informado pelo usuário (RF);
f) disponibilizar uma interface gráfica para utilização do sistema (RF);
g) ser desenvolvida em arquitetura cliente/servidor (Requisito Não-Funcional – RNF);
h) ser implementado utilizando o ambiente de desenvolvimento Eclipse (RNF);
i) ser implementado utilizando a linguagem de programação Java (RNF);
j) ser multiplataforma (RNF).

3.2 ESPECIFICAÇÃO

Na sequência é apresentada a especificação do mecanismo de busca, que foi modelada na ferramenta Entreprise Architect. O sistema foi desenvolvido seguindo a análise orientada a objetos, utilizando a notação *Unified Modeling Language* (UML) para a criação dos diagramas de casos de uso, classe e de sequência.

A Figura 3 apresenta o processo geral da solução proposta.

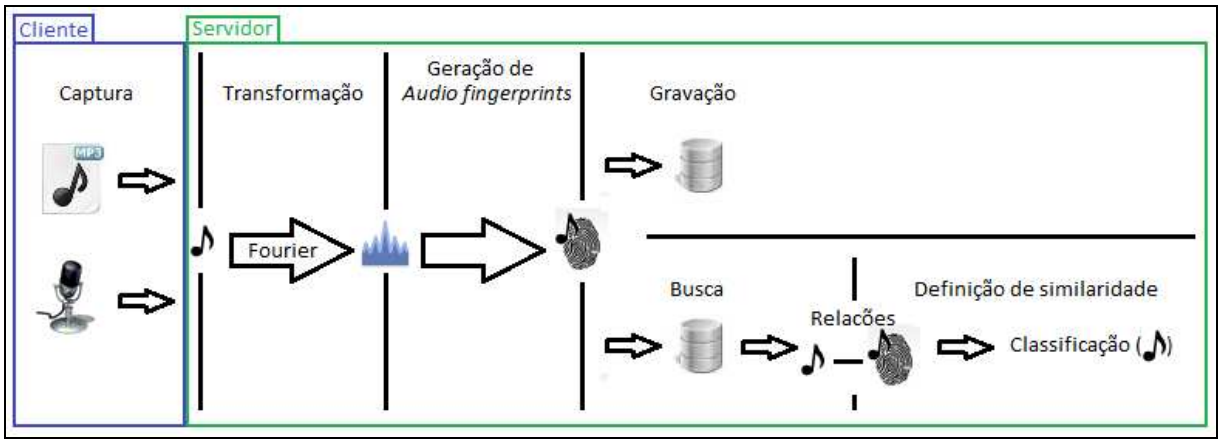


Figura 3 – Processo geral da solução

A Figura 3 mostra o processo geral da solução proposta neste trabalho. Inicialmente tem-se a captura de áudio, em uma aplicação cliente. Este áudio é submetido a uma aplicação rodando num servidor. É aplicada uma transformada neste áudio, de modo a obter suas frequências, e a partir delas extrair características para gerar os *audio fingerprints*. A partir do momento que se possua os *audio fingerprints*, pode-se realizar a gravação numa base, para posterior busca. Ou pode-se também submetê-los à base para realização de uma busca, obtendo relações com os áudios da base, e aplicando ao final deste processo, técnicas de similaridade para definir o áudio mais similar ao trecho informado.

3.2.1 Diagrama de casos de uso

O mecanismo possui três (3) casos de uso, que são apresentados na Figura 4.

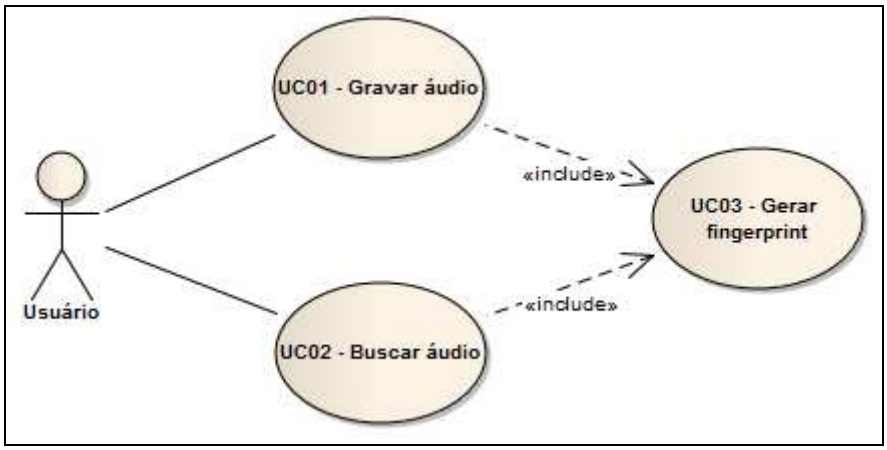


Figura 4 - Diagrama de casos de uso

Conforme pode-se observar na Figura 4, não existe um caso de uso principal, visto que todos são importantes para execução do sistema. A única restrição é que, para que a busca de um trecho de áudio (UC02), tenha um resultado positivo[3], o áudio original referente a este trecho tenha sido previamente cadastrado (UC01).

Abaixo são apresentados três quadros. O quadro 2 apresenta do caso de uso 01. O quadro 3 apresenta o caso de uso 02. E o quadro 4 apresenta o caso de uso 03.

UC01 - Gravar áudio	
Pré-condições	Não há pré-condições.
Cenário principal	01) O usuário inicia o aplicativo cliente de gravação. 02) O usuário digita o caminho para o arquivo de áudio a ser gravado. 03) O usuário solicita o registro do áudio. 04) O sistema cliente realiza a conversão do arquivo para o formato utilizado pelo sistema. 05) O sistema cliente envia solicitação de registro do áudio para o servidor. 06) O sistema servidor recebe o áudio, e divide-o em partes, gerando *fingerprints* para cada trecho do áudio (através do UC03). 07) O sistema servidor salva as informações geradas para o áudio. 08) O sistema servidor envia resposta ao sistema cliente indicando que o registro foi realizado. 09) O sistema cliente exibe mensagem indicando que o áudio foi registrado.
Fluxo alternativo 01	No passo 02: 02.1) O usuário opta por selecionar um arquivo através da navegação.
Exceção 01	No passo 04, o arquivo pode ter um formato não reconhecido pelo sistema cliente: 04.1) O sistema cliente exibe uma mensagem indicando que ocorreu um erro.
Exceção 02	No passo 05, pode não haver comunicação com o sistema servidor: 05.1) O sistema cliente exibe uma mensagem indicando que ocorreu um erro.
Exceção 03	No passo 07, pode já existir um áudio com o mesmo nome: 07.1) O sistema servidor envia resposta ao sistema cliente indicando que ocorreu um erro durante o registro do áudio. 07.2) O sistema cliente exibe mensagem indicando que o áudio não pôde ser registrado.
Pós-condição	O áudio e os *fingerprints* gerados a partir dele devem estar salvos.

Quadro 2 – Caso de Uso 01

[3] Consiga identificar corretamente o áudio do qual o trecho buscado faz parte.

UC02 - Buscar áudio	
Pré-condições	Não há pré-condições.
Cenário principal	01) O usuário inicia o aplicativo cliente de busca. 02) O usuário inicia a gravação de áudio para capturar o áudio do ambiente. 03) O sistema cliente efetua a gravação durante determinado tempo. 04) O sistema cliente envia solicitação de busca do áudio para o servidor. 05) O sistema servidor recebe o áudio, e divide-o em partes, gerando *fingerprints* para cada trecho do áudio (através do UC03). 06) O sistema servidor executa comparação do áudio com o conteúdo existente na base. 07) O sistema servidor envia resposta ao sistema cliente indicando os áudios considerados mais similares ao trecho gravado, de acordo com as técnicas utilizadas para comparação. 08) O sistema cliente exibe na tela as principais similaridades encontradas.
Exceção 01	No passo 02, pode não haver um microfone disponível no computador: 02.1) O sistema cliente exibe uma mensagem indicando que ocorreu um erro.
Exceção 02	No passo 04, pode não haver comunicação com o sistema servidor: 04.1) O sistema cliente exibe uma mensagem indicando que ocorreu um erro.
Pós-condição	Áudios mais similares encontrados, de acordo com cada técnica utilizada.

Quadro 3 – Caso de Uso 02

UC03 - Gerar fingerprint	
Pré-condições	Frequências obtidas de um trecho de áudio devem ser informadas
Cenário principal	01) O sistema identifica os pontos de frequência mais importantes dentre as frequências recebidas. 02) O sistema gera um *hash* baseado nos pontos identificados.
Pós-condição	*Hash* do *fingerprint* gerado

Quadro 4 – Caso de Uso 03

3.2.2 Diagrama de classes

Responsável por fornecer o panorama geral do relacionamento entre as classes do projeto, o diagrama de classes é apresentado por funcionalidades (como armazenamento, busca e classificação) para facilitar o entendimento da estrutura construída.

Na Figura 5 são apresentadas as classes referentes ao armazenamento de dados necessários a este trabalho.

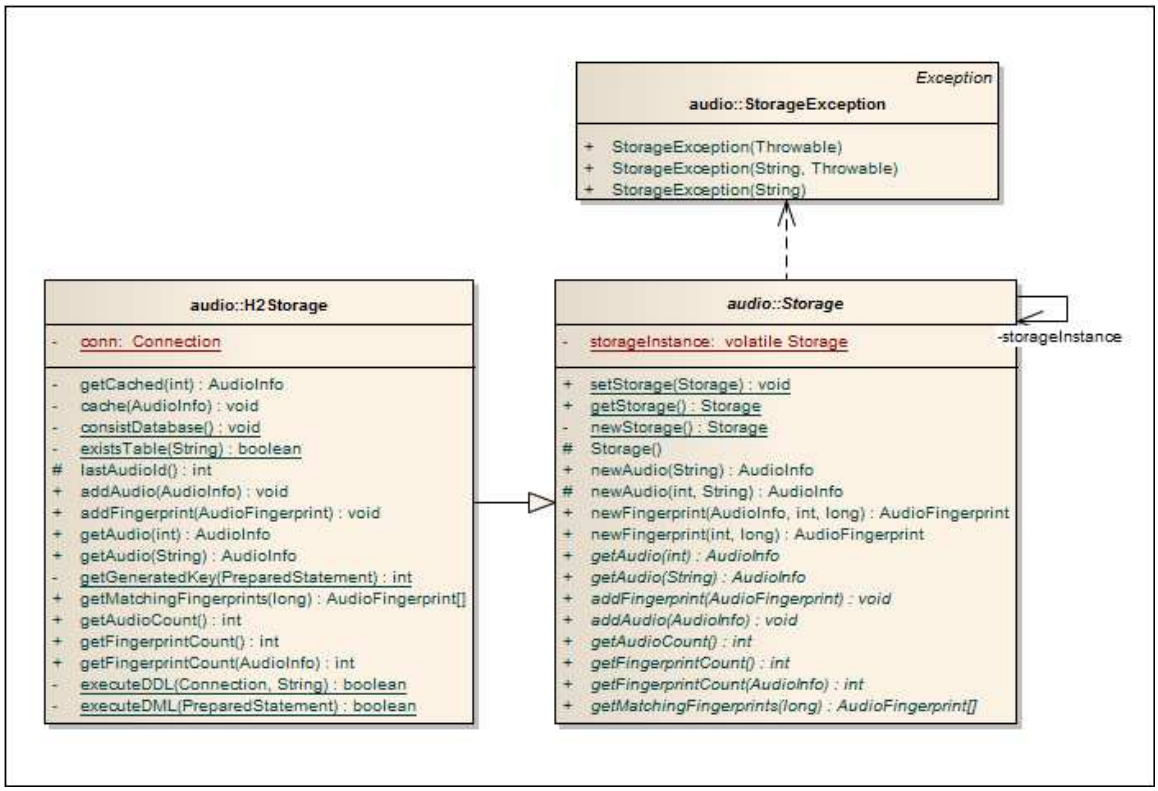


Figura 5 - Diagrama de classes de armazenamento

As classes especificadas na Figura 5 são as seguintes:

a) `StorageException`: é uma abstração para erros que possam vir a ocorrer durante o armazenamento de alguma informação;

b) `Storage`: é a classe base de persistência. Ela é uma classe abstrata, e possui a implementação dos métodos de criação de objetos, mas os métodos inserção e busca são definidos como abstratos. Esta classe, possui um método `setStorage`, que é utilizado para configurar a implementação de `Storage` que deve ser utilizada como padrão, e que será retornada no método `getStorage`;

c) `H2Storage`: é uma especialização da classe `Storage`, e implementa os métodos de inserção e busca de dados, utilizando acesso ao banco de dados H2 (H2 DATABASE, 2011).

A geração de *fingerprints* é uma parte importante neste trabalho, sendo necessário para o funcionamento correto da gravação, assim como da busca de áudios. Esta funcionalidade está representada na Figura 6.

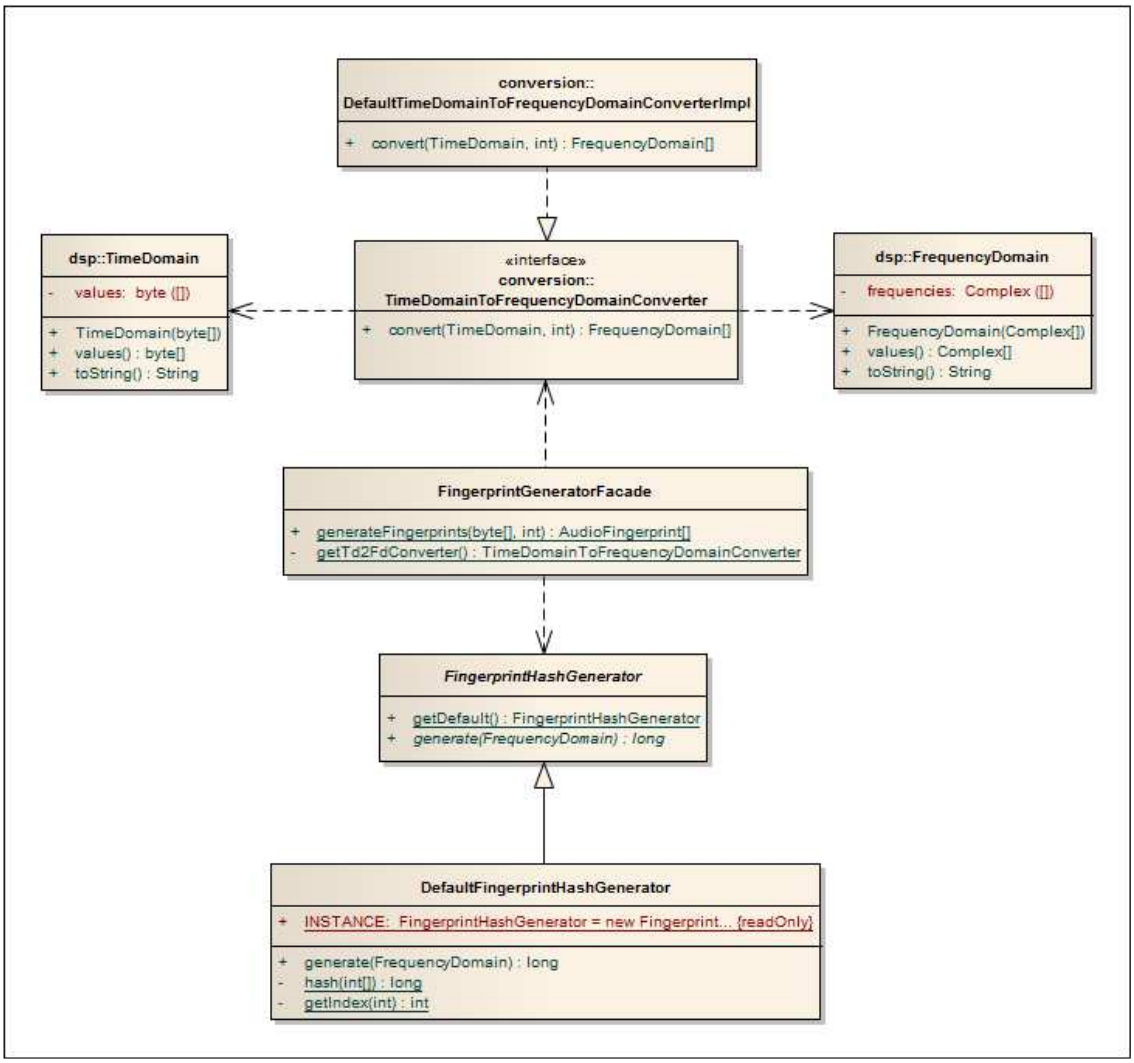


Figura 6 - Diagrama de classes de geração de *fingerprints*

Para poder representar o domínio do tempo e o domínio da frequência, foram especificadas duas classes: `TimeDomain` e `FrequencyDomain`, que podem ser vistas na Figura 6. Para efetuar a conversão da informação no domínio do tempo para o domínio da frequência, foi definida a interface `TimeDomainToFrequencyDomainConverter`, que é realizada pela classe `DefaultTimeDomainToFrequencyDomainConverterImpl`, que por sua vez utiliza a transformada de Fourier (ver seção 2.4) para realizar esta conversão. A classe abstrata `FingerprintHashGenerator` foi definida para identificar uma classe capaz de realizar a geração de *hashes* de *fingerprints* a partir das informações do domínio da frequência. Foi então descrita a classe `DefaultFingerprintHashGenerator`, que é uma especialização da classe anterior, para denominar a implementação da geração dos *hashes* de *fingerprints* neste trabalho. Para facilitar a criação das *fingerprints*, e tornar esta lógica mais fácil de reutilizar, foi definida a classe `FingerprintGeneratorFacade`.

As principais informações geridas neste procedimento são a identificação do áudio, e

fingerprints, que são geradas a partir do conteúdo extraído. Para gerenciar estas informações, foram especificadas as classes definidas na Figura 7.

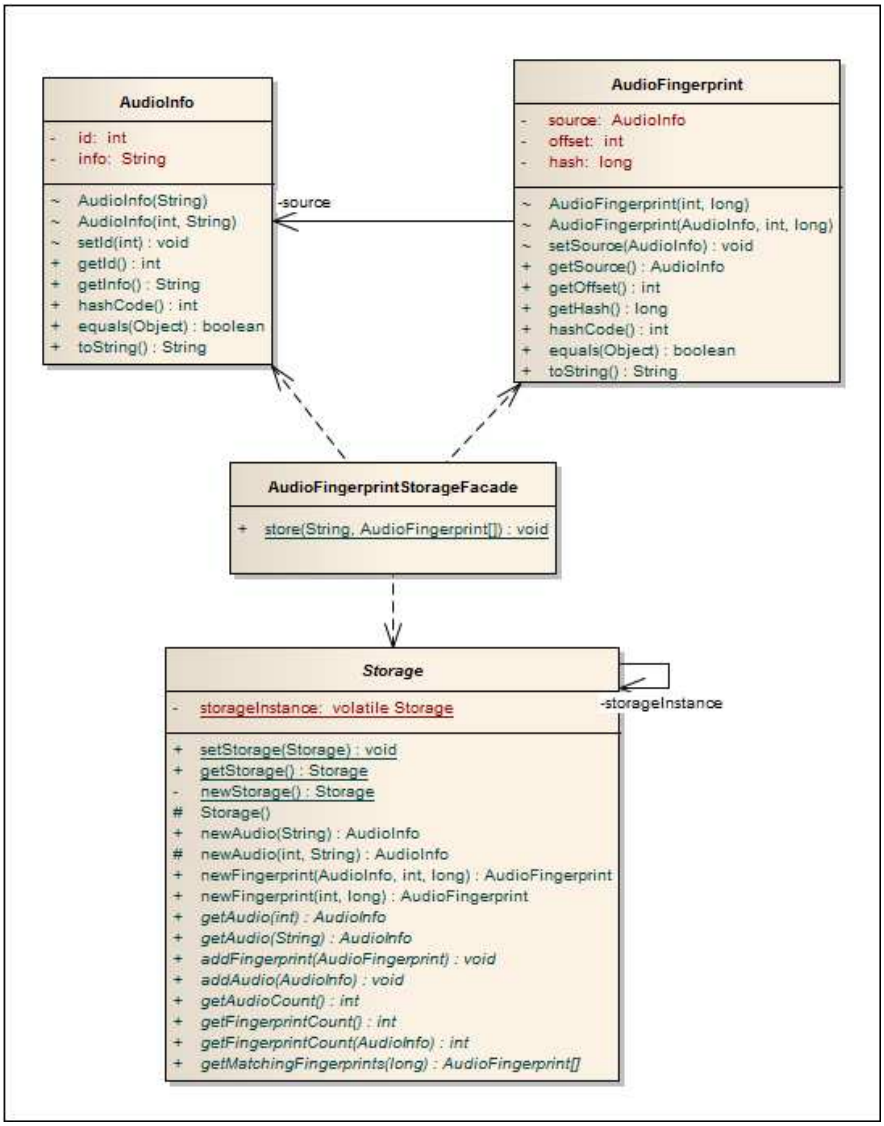


Figura 7 - Diagrama de classes do armazenamento da identificação do áudio e *fingerprints*

Para tornar mais simples a gravação das *fingerprints*, foi especificada a classe `AudioFingerprintStorageFacade`, como pode ser visto na Figura 7. As classes `AudioInfo` e `AudioFingerprint`, também visualizadas na Figura 7, representam a identificação do áudio, e o *fingerprint* criado a partir de suas características. Nos quadros 5 e 6 são detalhados os atributos dessas duas classes.

AudioInfo	
Atributo	**Descrição**
Id	identificar numérico do áudio
Info	nome descritivo do áudio, para que um usuário identifique mais facilmente

Quadro 5 – Atributos da classe `AudioInfo`

AudioFingerprint	
Atributo	**Descrição**
Source	o áudio do qual foi gerado o *fingerprint*
Offset	identificação espacial deste *fingerprint* dentro do áudio de onde ele foi originado
Hash	*hash* gerado a partir das características do áudio

Quadro 6 – Atributos da classe `AudioFingerprint`

Na Figura 8 é mostrada a especificação de classes referentes a busca de áudio.

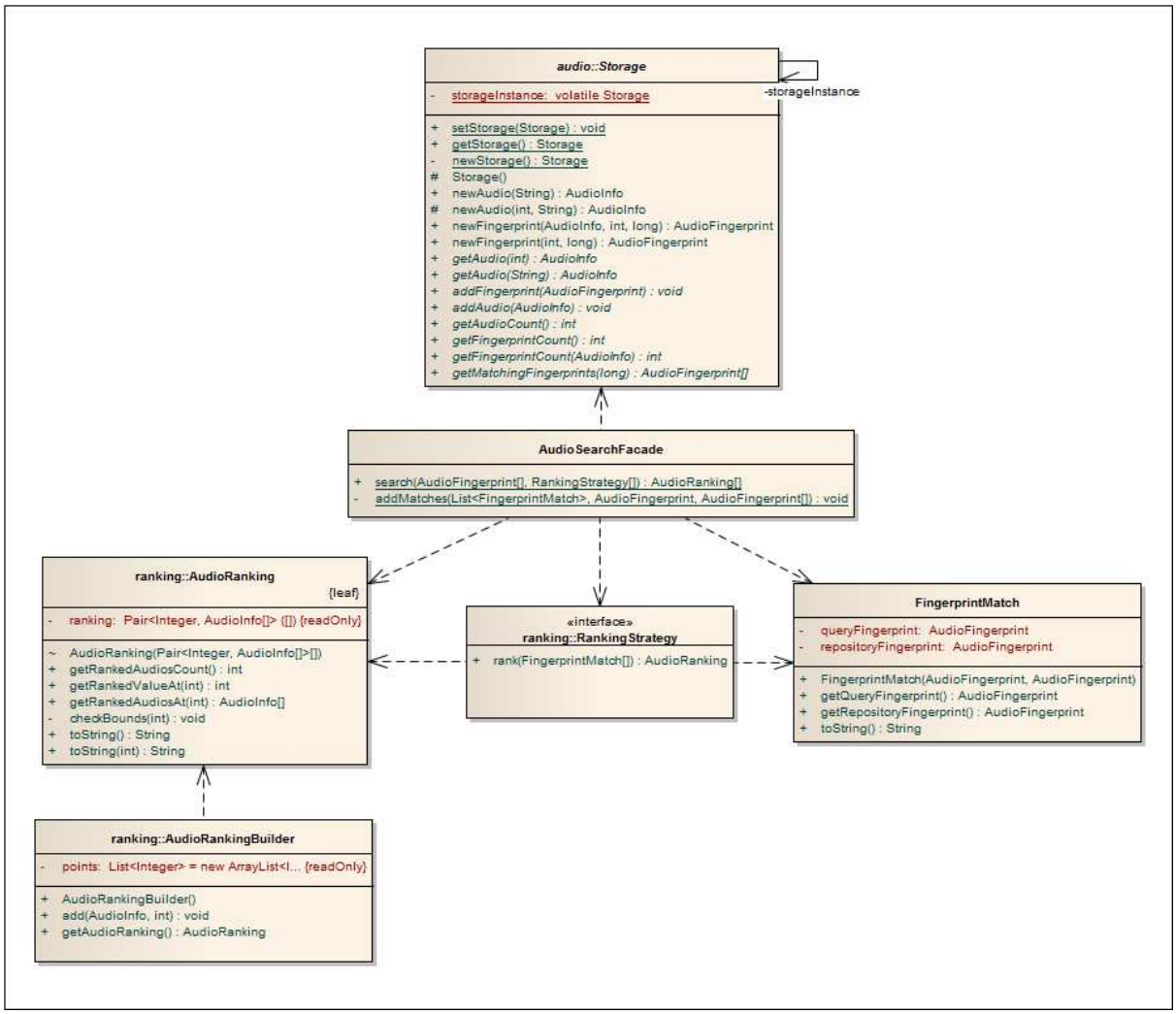


Figura 8 - Diagrama de classes da busca de áudio

Na Figura 8, a classe `FingerprintMatch` representa uma similaridade encontrada durante a busca. `RankingStrategy` é uma interface que define um método para realizar a classificação de áudios após uma busca de similaridades ter sido executada. `AudioRanking` é o resultado de uma busca, ele mantém, de maneira ordenada, os resultados obtidos com a classificação dos áudios a partir das similaridades encontradas. A classe `AudioRankingBuilder`, apesar de não ter uma dependência direta com a interface `RankingStrategy`, acaba sendo utilizada pelas classes que a implementam, facilitando a montagem da classificação final. A classe `AudioSearchFacade` foi criada para facilitar o

reuso de suas funcionalidades, que neste caso é a busca de áudio.

Por fim, na Figura 9 são especificadas as cinco (5) técnicas de classificação de áudio baseadas em similaridade que foram desenvolvidas neste trabalho.

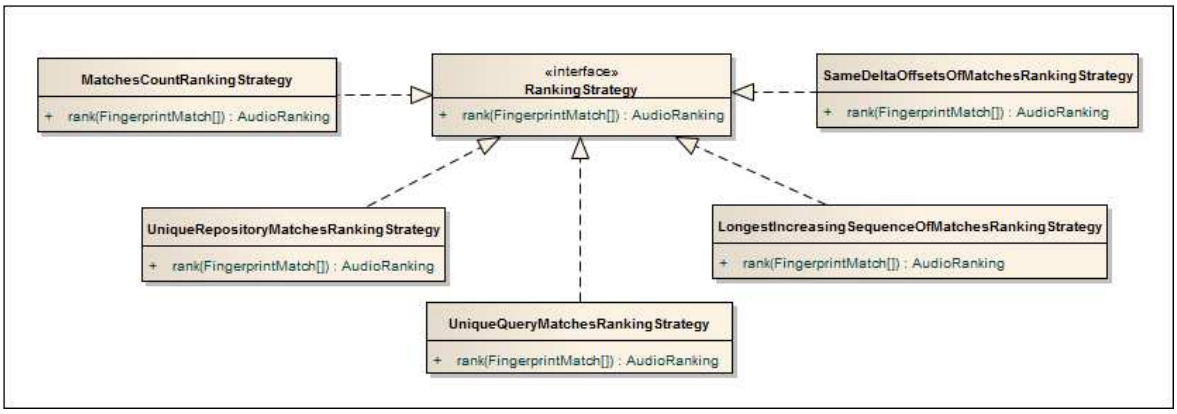


Figura 9 - Diagrama de classes das técnicas de classificação utilizadas

Cada uma das realizações da interface `RankingStrategy` exibidas na Figura 9 utiliza um algoritmo diferente para identificar os áudios mais relevantes, a partir de pares de *fingerprints* da *query* com *fingerprints* de mesmo *hash*, encontrados no repositório. Os algoritmos utilizados serão descritos de maneira mais aprofundada na seção 3.3.2.5.2.

3.2.3 Diagrama de sequência

Na Figura 10 é apresentado o processo de gravação de um áudio. Este processo é definido pelo caso de uso UC01 - Gravar áudio.

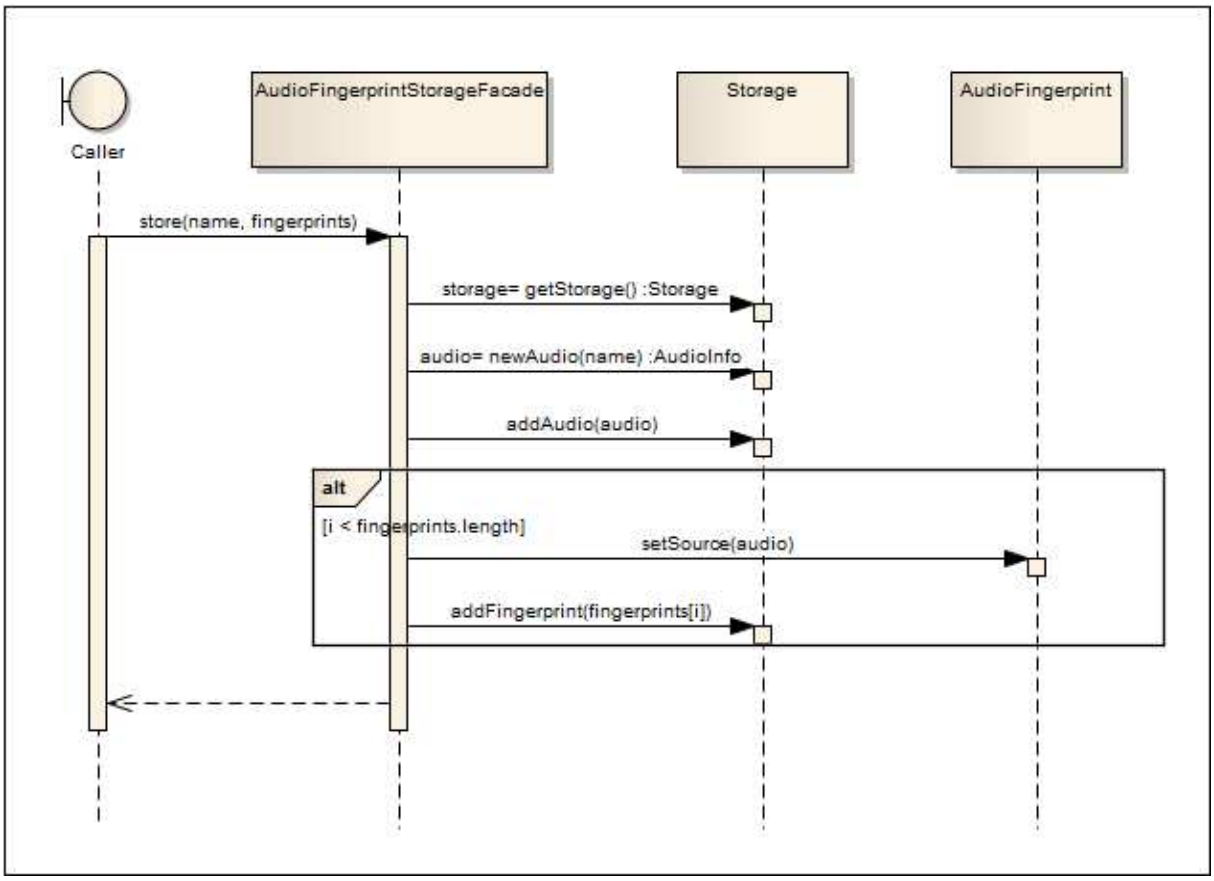


Figura 10 - Diagrama de sequência da gravação de áudio

Como pode ser visto na Figura 10, o processo é executado na classe `AudioFingerprintStorageFacade`, sendo iniciado pela chamada do método `store`. Este obtém a instância de `Storage` a partir do método `getStorage`, da mesma classe. Uma instância de `AudioInfo` é criada utilizando o nome recebido na chamada do método, e este é então gravado utilizando o método `addAudio`. Para cada um dos `AudioFingerprint` é configurado seu áudio de origem como sendo o áudio gravado no momento anterior, isto é feito utilizando o método `setSource`. Tendo esta associação feita, é possível fazer a gravação do *fingerprint*, através de `addFingerprint`.

O diagrama da Figura 11 apresenta o processo para gerar uma *fingerprint*, que é definido no caso de uso UC03 - Gerar fingerprint.

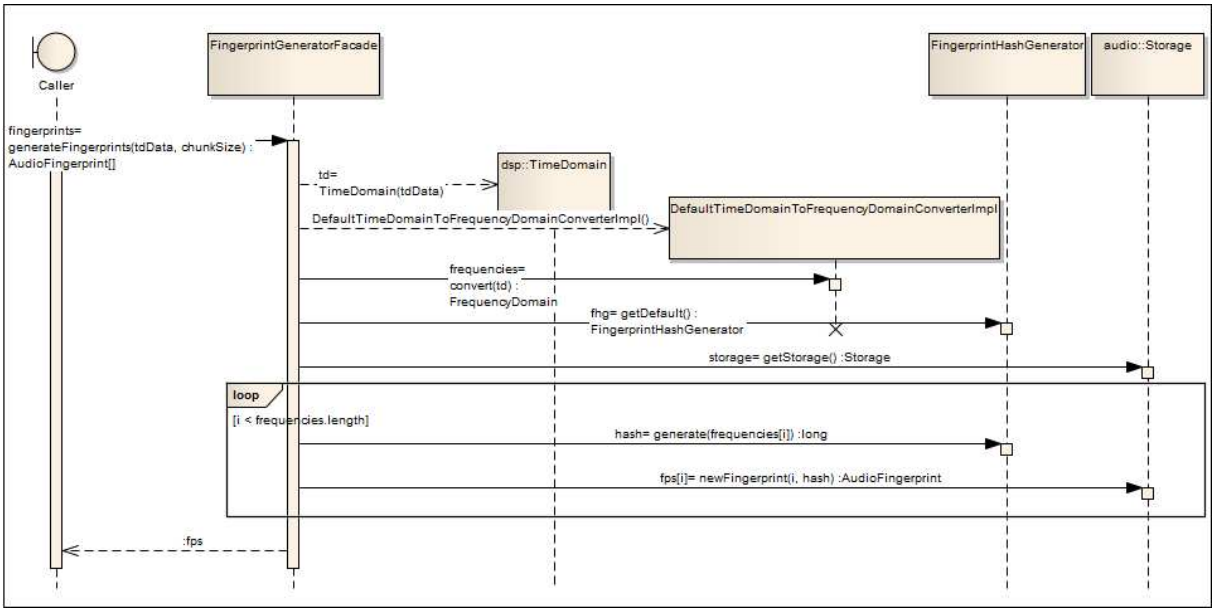


Figura 11 - Diagrama de sequência da geração de *fingerprint*

Como pode ser visto na Figura 11, a geração de uma *fingerprint* é inicializada a partir da chamada do método `generateFingerprints` definido na classe `FingerprintGeneratorFacade`. Esta cria uma instância de `TimeDomain`, e utiliza uma instância de `TimeDomainToFrequencyDomainConverter` para transformar a entrada em uma série de representações do domínio da frequência, representados por um vetor de `FrequencyDomain`. É obtida então uma instância de `FingerprintHashGenerator`, e cada `FrequencyDomain` é repassado a ele para que sejam gerados *hashes* referentes a cada `FrequencyDomain`. É criado um *fingerprint* para cada `FrequencyDomain`, sendo que o índice do vetor é utilizado como *offset*, e o *hash* gerado para o `FrequencyDomain` é utilizado como *hash* do *fingerprint*.

Na Figura 12 é apresentado o processo que realiza a busca do áudio na base de dados, definida no caso de uso UC02 - Buscar áudio.

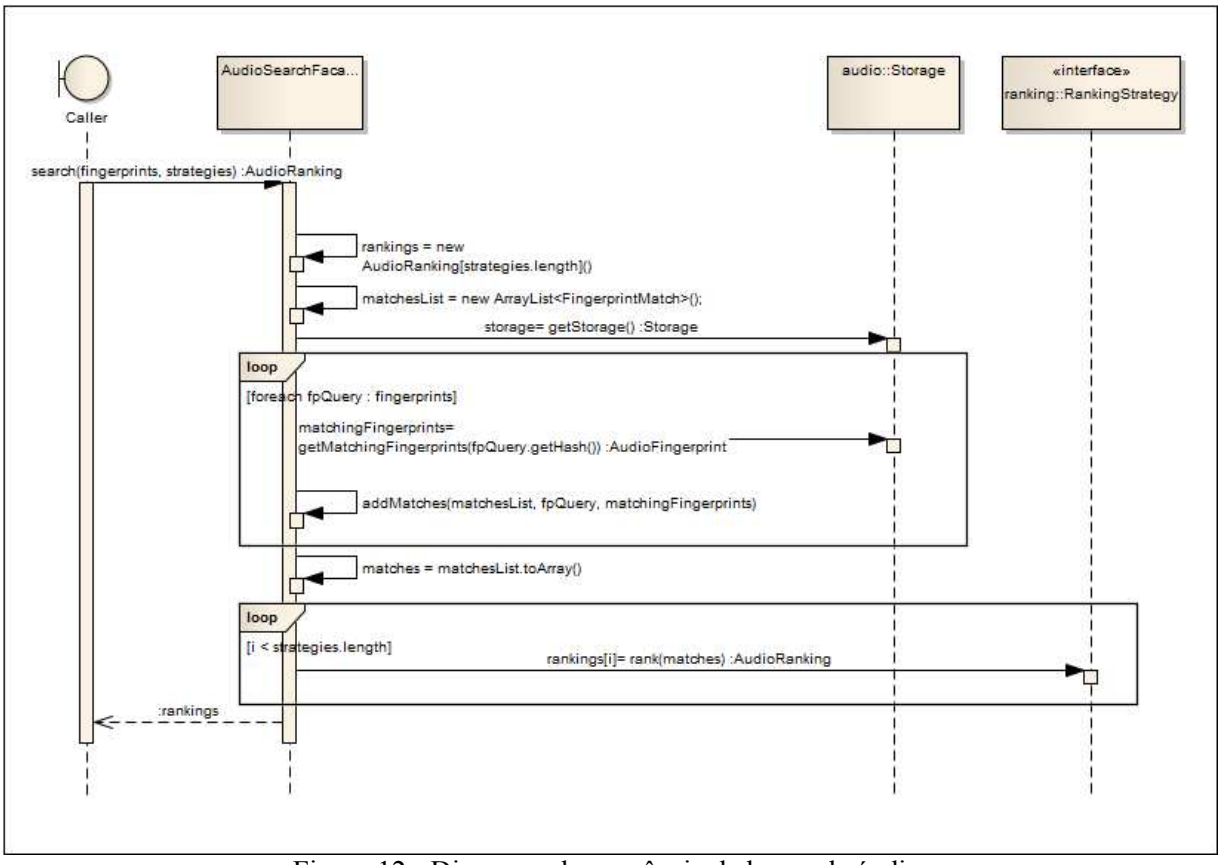


Figura 12 - Diagrama de sequência da busca de áudio

Como apresentado na Figura 12, a busca do áudio é processada a partir da classe `AudioSearchFacade`. Esta recebe uma série de `AudioFingerprint`, que são os termos a serem buscados, e também uma série de `RankingStrategy`, que define técnicas que serão utilizadas para realizar a classificação das similaridades encontradas, afim de identificar as similaridades mais relevantes, e desta forma, os áudios mais similares à busca realizada.

Inicialmente é obtida a instância de `Storage`, que mantém os *fingerprints*, ele é obtido através do método `getStorage`. Para cada `AudioFingerprint` dado de entrada, são encontradas as suas similaridades, através do método `getMatchingFingerprints`. Para cada `AudioFingerprint` retornado desta chamada, é criada uma instância de `MatchingFingerprint`, que mantém uma referência para o `AudioFingerprint` da *query*, e uma referência a um `AudioFingerprint` similar encontrado no repositório. Tendo a lista formada com todas as similaridades encontradas, é obtida cada instância derivada de `RankingStrategy` que foi recebida no início da chamada, sendo aplicado o seu método `rank`, passando a lista com as similaridades. O resultado obtido disto é a classificação dos áudios a partir da utilização desta técnica. Após executar o mesmo método para cada `RankingStrategy` , são retornados todos os `AudioRanking` gerados no processo.

3.2.4 Modelagem do banco de dados

O mecanismo de busca possui apenas duas tabelas: `Audio` e `Fingerprint`, conforme mostra a Figura 13.

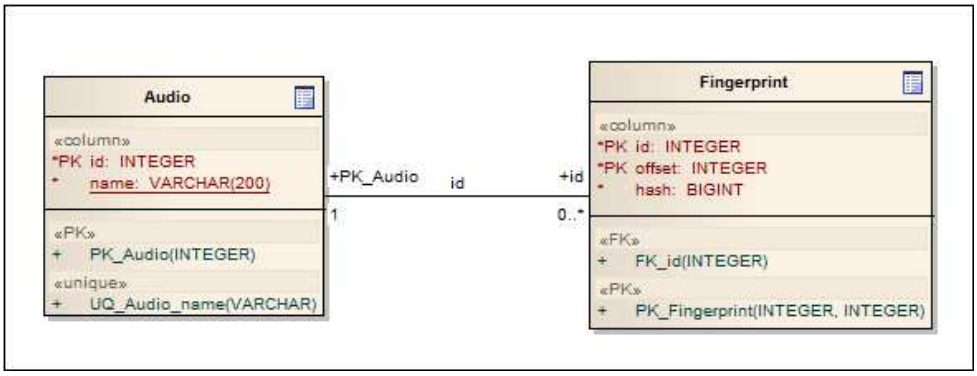


Figura 13 – Tabelas e seus relacionamentos

No quadro 7 são apresentadas as tabelas e suas descrições.

Tabela	**Descrição**
AUDIO	Tabela com a definição dos áudios que estão armazenados. Possui um identificador (`ID`) para identificação do áudio em buscas; e um nome (`NAME`) para que o usuário possa assimilar mais facilmente a quê áudio se refere o resultado gerado em uma busca.
FINGERPRINT	Tabela para armazenar as *fingerprints* geradas a partir de um áudio. Possui o identificador do áudio (`ID`), para saber de que áudio foi gerado este *fingerprint*; um *offset* (`OFFSET`), para indicar de qual trecho do áudio este *fingerprint* foi extraído; e um *hash* (`HASH`), que é a informação que foi gerada a partir das características do trecho de áudio.

Quadro 7 – Tabelas e suas descrições

É importante ressaltar que o armazenamento de diversos tipos de representações, assim como a especialização dos áudios, não é foco deste trabalhado, e por este motivo foi optado pela simplicidade na definição da base de dados.

3.3 IMPLEMENTAÇÃO

Nesta seção são apresentadas informações sobre as técnicas e ferramentas utilizadas para a implementação do mecanismo, bem como a implementação de cada etapa e a operacionalidade do mecanismo de busca desenvolvido no presente trabalho.

3.3.1 Técnicas e ferramentas utilizadas

O mecanismo de busca foi desenvolvido na linguagem Java, pelo fato de ela ser multiplataforma. Foi utilizada a IDE Eclipse na versão 3.6, pois ela possui uma boa aderência à linguagem, além de ser uma IDE gratuita. Para armazenamento dos *audio fingerprints* gerados a partir das amostras foi utilizado o banco de dados embarcado H2 Database Engine, por sua facilidade de uso.

3.3.2 Implementação do mecanismo

Nesta seção são apresentadas as etapas para a implementação das principais classes e funções desenvolvidas neste trabalho.

3.3.2.1 Obtenção de áudio

Para criar o mecanismo de busca, primeiro é necessário obter a sua matéria-prima: o áudio. Foram utilizadas duas formas de obtenção do áudio: áudio gravado em tempo real, através de uma entrada de áudio (microfone), e áudio obtido a partir de uma gravação já existente, de um arquivo de áudio. Este áudio obtido está no que é chamado "domínio do tempo" (ver seção 2.4).

Para obter os áudios de entrada, é necessário definir também o formato em que os mesmos serão obtidos. Foi definida a utilização de apenas um canal do áudio (mono), 8 bits, e com taxa de amostragem de 44100 Hz (44100 amostras por segundo). Como a gravação e a conversão do áudio não são focos neste trabalho, não serão explanadas suas especificidades e funcionamento.

Os dados obtidos nesta seção (áudio) também pertencem ao domínio do tempo, por se tratarem de amostras obtidas a intervalos regulares de tempo, onde este intervalo é definido pela taxa de amostragem do formato de áudio.

3.3.2.1.1 Obtenção de áudio a partir de microfone

Uma das maneiras de se obter o áudio é através da gravação do mesmo, diretamente do dispositivo de entrada de áudio (microfone). Para obter o áudio, foi utilizada a Java Sound API (ORACLE, 2010).

Com a Java Sound API, basta informar o formato de áudio definido no item anterior para obter o dispositivo de entrada de áudio (microfone). O código fonte que realiza este processo é apresentado no quadro 8.

```
private TargetDataLine getMicrophoneTargetDataLine(AudioFormat format)
throws AudioRecordingException {
    DataLine.Info info = new DataLine.Info(TargetDataLine.class,
AudioUtils.asJavaSoundAudioFormat(format));
    try {
        TargetDataLine line = (TargetDataLine) AudioSystem.getLine(info);
        if (line == null) {
            throw new AudioRecordingException("Microphone not found for
format " + format);
        }
        return line;
    } catch (LineUnavailableException e) {
        throw new AudioRecordingException("Microphone not found for format
" + format, e);
    }
}
```

Quadro 8 – Obtenção de entrada de microfone

Como pode ser visto no quadro 8, primeiro é necessário criar uma instância de `DataLine.Info`, onde é informado o tipo `TargetDataLine.class`, e o formato a ser recebido do microfone (definido na seção 3.3.2.1). Então executa-se o método `getLine` da classe `AudioSystem`, passando como parâmetro o objeto criado anteriormente. Se existir um microfone capaz de realizar gravações com o formato especificado, ele será retornado por este método.

Tento o microfone, basta realizar a gravação. O código fonte que realiza esta gravação é apresentado no quadro 9.

```
public byte[] record(AudioFormat format, long time, TimeUnit unit) throws
AudioRecordingException {
    final long millis = unit.toMillis(time);

    try {
        line.open(AudioUtils.asJavaSoundAudioFormat(format));
    } catch (LineUnavailableException e) {
        throw new AudioRecordingException("A linha de entrada está
indisponível", e);
    }

    line.start();

    long start = System.currentTimeMillis();

    byte[] buffer = new byte[line.getBufferSize()];

    ByteArrayOutputStream out = new ByteArrayOutputStream(4096);
    try {
        // keep reading until passed 'timeInMillis' milliseconds
        while ((System.currentTimeMillis() - start) <= millis) {
            int count = line.read(buffer, 0, buffer.length);
            if (count > 0) {
                out.write(buffer, 0, count);
            }
        }
        out.close();
    } catch (IOException ioe) {
        throw new AudioRecordingException(ioe);
    }
    long end = System.currentTimeMillis();

    System.out.println("Listened to the microphone for: " + (end - start)
+ "ms");

    line.stop();

    byte[] bytes = out.toByteArray();
    return bytes;
}
```

Quadro 9 – Código da gravação de áudio

Conforme quadro 9, primeiramente é realizada a chamada do método `open` para o microfone (representado pela variável `line`), para indicar que recurso do microfone deve ser reservado para uso da aplicação, e que as gravações realizadas nele devem ter um determinado formato. A seguir, é utilizado o método `start` para que este inicie a recepção de áudio. Durante o tempo que for necessário, deve-se utilizar um *buffer* intermediário para ler o conteúdo sendo gravado pelo microfone, esta leitura é realizada através do método `read`. E este conteúdo lido para o *buffer*, deve ser gravado em outra estrutura para que se possam manter tudo o que for lido. Após a gravação ter sido realizada por tempo suficiente, o método `stop` deve ser chamado para que a gravação de áudio seja finalizada.

O problema com este tipo de gravação é a interferência de ruídos oriundos do ambiente, eco e reverberação, que podem degradar a qualidade do áudio gravado, dificultando posteriormente a obtenção das características do áudio, e consequentemente, a obtenção de resultados mais precisos em uma busca.

3.3.2.1.2 Obtenção de áudio a partir de arquivo de mídia

Outra maneira de se obter um áudio é através de um arquivo de mídia. Para este trabalho foi escolhido apenas o formato de arquivo MP3, por ser bem difundido, e portanto, torna-se mais fácil encontrar arquivos a serem utilizados para testes e bibliotecas capazes de lerem arquivos neste formato. Para leitura dos arquivos MP3, novamente, foi utilizada a Java Sound API.

Para utilizar a Java Sound API na leitura dos arquivos MP3, foi necessária a utilização de uma extensão da mesma, pois ela, por si só, não é capaz de ler arquivos MP3. A extensão utilizada chama-se MP3SPI for Java Sound (JAVAZOOM, 2010). Após conseguir obter o áudio no formato MP3, surgiu um novo problema: o formato de áudio destes arquivos. O formato dos arquivos MP3 nem sempre é igual ao que foi definido na seção 3.3.2.1. Muitas vezes o formato utilizado é multicanal (Estéreo) e 16 bits, assim como é o padrão em CDs. Para resolver este problema, pesquisou-se uma biblioteca que fosse capaz de realizar a conversão entre diferentes formatos, e a biblioteca encontrada foi a Harmonic (HARMONIC, 2011), pois a mesma permite realizar a conversão entre diversas definições de formatos, de 8 a 32 bits, e de 1 a N canais de áudio, para as definições do formato desejado. Estes dois processos, a leitura do arquivo de áudio, e a conversão para o formato desejado, estão representados no quadro 10.

```
public static AudioInputStream getAudioInputStream(
File audioFile, AudioFormat targetFormat) throws AudioConversionException
{
    try {
        AudioInputStream ais =
AudioSystem.getAudioInputStream(audioFile);
        AudioFormat audioBaseFormat =
AudioUtils.fromJavaSoundAudioFormat(
ais.getFormat());
        AudioFormat decodedFormat =
audioBaseFormat.withSigned(true).withSampleSizeInBits(16).withByteOrder(
ByteOrder.LITTLE_ENDIAN);

        ais =
AudioSystem.getAudioInputStream(AudioUtils.asJavaSoundAudioFormat(
decodedFormat), ais);
        ais = new AudioFloatFormatConverter().getAudioInputStream(
AudioUtils.asJavaSoundAudioFormat(targetFormat), ais);

        return ais;
    } catch (IOException ioe) {
        throw new AudioConversionException(ioe);
    } catch (UnsupportedAudioFileException uafe) {
        throw new AudioConversionException(uafe);
    }
}

public static byte[] readFile(File audioFile, AudioFormat targetFormat)
throws AudioConversionException {
    AudioInputStream ais = getAudioInputStream(audioFile, targetFormat);

    byte[] data = new byte[8192];
    int bytesRead;

    final ByteArrayOutputStream out = new ByteArrayOutputStream(4096);

    try {
        while ((bytesRead = ais.read(data, 0, data.length)) > -1) {
            out.write(data, 0, bytesRead);
        }
    } catch (IOException e) {
        throw new AudioConversionException(e);
    }

    return out.toByteArray();
}
```

Quadro 10 – Métodos para leitura de arquivo de áudio

No quadro 10 é apresentado o código para leitura de arquivos de áudio. Para executar esta operação, primeiro utiliza-se o método `getAudioInputStream` da classe `AudioSystem` para obter um *stream* do arquivo de áudio no seu formato padrão (este método utiliza a biblioteca capaz de ler arquivos do formato especificado, no caso de arquivos MP3, a biblioteca é o MP3SPI for Java Sound, já citado anteriormente). Após isso, é necessário utilizar o mesmo método (`getAudioInputStream`) para indicar que o *stream* obtido deve ser lido utilizando outro formato (esta conversão intermediária se mostrou necessária, pois erros ocorriam ao tentar converter direto do formato original para o desejado). Possuindo o *stream* no formato intermediário, pode-se fazer a conversão para o formato desejado, isto é realizado através do método `getAudioInputStream` da classe `AudioFloatFormatConverter` (do

Harmonic). Com isso, é gerado um novo *stream*, que ao ser lido, trará as informações no formato esperado. Tendo este *stream*¸ basta ler então todo o seu conteúdo, e obtém-se o áudio do arquivo no formato desejado.

3.3.2.2 Obtenção das frequências do áudio

Até então, estava-se trabalhando apenas sobre o domínio do tempo, porém, para conseguir obter as características do áudio, é necessário trabalhar sobre o domínio da frequência. Para transformar um conjunto de dados no domínio do tempo, em um conjunto de dados no domínio da frequência, é necessário aplicar a transformada discreta de Fourier (ver seção 2.4). Neste trabalho foi utilizada a implementação disponibilizada pela Apache Software Foundation, encontrada na biblioteca Apache Commons Math (APACHE SOFTWARE FOUNDATION, 2011). Esta biblioteca utiliza um algoritmo chamado FFT, que possui um desempenho melhor que a DFT padrão (o FFT executa em O(nlogn), enquanto a DFT padrão executa em $O(n^2)$). A única restrição deste algoritmo é que o número de valores da entrada seja uma potência de 2. O algoritmo que aplica a transformada de Fourier é apresentado no quadro 11.

```
public FrequencyDomain[] convert(TimeDomain timeDomain, int chunkSize) {
    byte[] tdValues = timeDomain.values();
    int tdSize = tdValues.length;

    final int fdSize = tdSize / chunkSize;
    //final int residual = tdSize % chunkSize;// the rest of the time
domain. will not be used

    FrequencyDomain[] fds = new FrequencyDomain[fdSize];

    // translate bytes to complex numbers to deal with the frequency
domain

    FastFourierTransformer fft = new FastFourierTransformer();

    for (int fdIndex = 0; fdIndex < fdSize; fdIndex++) {
        // creates a Complex[] with 'chunkSize' complex numbers
        Complex[] complex = new Complex[chunkSize];
        // the complex numbers created are 'chunkSize' sequential values
in the time domain
        int offset = (fdIndex * chunkSize);
        for (int i = 0; i < chunkSize; i++) {
            // Put the time domain data into a complex number with
imaginary part as 0:
            complex[i] = new Complex(tdValues[offset + i], 0);
        }
        // Perform the FFT on the chunk
        Complex[] transformed = fft.transform(complex);
        fds[fdIndex] = new FrequencyDomain(transformed);
    }

    return fds;
}
```

Quadro 11 – Transformação do domínio do tempo para o domínio da freqüência

O quadro 11 demonstra a transformação do áudio no domínio do tempo para o domínio da frequência. Para utilizar uma DFT é necessário utilizar números complexos. Números complexos são formados de uma parte real, e uma parte imaginária. Para utilizar os valores obtidos do domínio do tempo como entrada na DFT, é necessário que, para cada valor do domínio do tempo, seja utilizado um número complexo correspondente que possua a parte real igual ao valor original, e parte imaginária igual a 0. Após executar a transformada, têm-se o áudio no domínio da frequência, as informações relativas a tempo do áudio são perdidas, por este motivo, é necessário dividir o áudio em trechos, cujo tamanho é definido pelo parâmetro `chunkSize`, e aplicar a transformada em cada um destes trechos.

3.3.2.3 Geração dos *audio fingerprints*

Após obtidas as frequências do áudio, é possível gerar os *audio fingerprints*. Para isso, são definidas faixas de frequências, que são as seguintes: 0 a 40, 41 a 80, 81 a 120 e 121 a 180. De cada frequência resultante é obtida a sua magnitude. As frequências de maior magnitude em cada faixa são então selecionadas, resultando assim em quatro valores. Estes

valores são reunidos num único valor através de operações binárias, de deslocamento de bits. O novo número é formado de modo que cada pico de frequência obtido ocupe um *byte*, este número gerado é o *hash*. Este *hash*, junto com a informação do *offset* deste trecho dentro do áudio de origem, formarão o *audio fingerprint* (descrito na seção 2.2). O algoritmo para isso é apresentado no quadro 12.

```
private static final int LOWER_LIMIT = 0;
private static final int UPPER_LIMIT = 180;
private static final int[] RANGE = new int[] { 40, 80, 120, UPPER_LIMIT };
private static final int AMOUNT_OF_POINTS = RANGE.length;

@Override
public long generate(FrequencyDomain fd) {
      Complex[] frequencies = fd.values();

      double[] highscores = new double[AMOUNT_OF_POINTS];
      int[] recordPoints = new int[AMOUNT_OF_POINTS];

      int freqRange = 0;

      for (int freq = LOWER_LIMIT; freq <= UPPER_LIMIT; freq++) {
            // Get the magnitude:
(http://en.wikipedia.org/wiki/Magnitude_(mathematics)#Complex_numbers)
            // Others:
(http://en.wikipedia.org/wiki/Magnitude_(mathematics)#Practical_math)

            if (RANGE[freqRange] < freq) {
                  freqRange++;
            }

            double magnitude = frequencies[freq].abs();

            // Save the highest magnitude and corresponding frequency:
            if (magnitude > highscores[freqRange]) {
                  highscores[freqRange] = magnitude;
                  recordPoints[freqRange] = freq;
            }
      }

      return hash(recordPoints);
}

private static long hash(int[] p) {
      // & 0xFE to ignore the least significant bit
      int p0 = (RANGE[0] - p[0]) & 0xFE;
      int p1 = (RANGE[1] - p[1]) & 0xFE;
      int p2 = (RANGE[2] - p[2]) & 0xFE;
      int p3 = (RANGE[3] - p[3]) & 0xFE;

      return (p0) | ((p1) << 8) | ((p2) << 16) | ((p3) << 24);
}
```

Quadro 12 – Geração de *fingerprints*

O quadro 12 apresenta o algoritmo para a geração das *fingerprints*. Nele, é recebido como entrada para o algoritmo, um trecho do áudio no domínio da frequência. Dois vetores são criados para armazenar o valor do pico de frequência em uma determinada faixa, e a frequência em que este pico ocorre. Todavia, varrem-se as frequências pertencentes a estas faixas verificando suas magnitudes e, se as mesmas são a maioria da faixa. Ao obter a informação de todos os picos é aplicado o algoritmo de *hash*, que transforma o valor das

quatro frequências em valores que caibam em um *byte*, e então unem estes quatro *bytes* para formarem o *hash*.

Com os *fingerprints* gerados, duas operações são possíveis: a gravação dos *audio fingerprints* na base de dados, para que o áudio referente a eles possam ser posteriormente identificado através de uma busca; ou a busca de áudio a partir dos *fingerprints* gerados, onde nesta busca são também aplicadas técnicas para identificação do grau de similaridade entre os áudios da base e o trecho gravado.

3.3.2.4 Gravação de *audio fingerprints*

Para identificar o áudio na base, é criado um registro indicando informações sobre o áudio, onde, neste trabalho, foi utilizada uma representação simples, com apenas um identificador e um nome descritivo do áudio. Estas entidades são detalhadas na seção 3.2.4. Um registro para cada áudio distinto é gerado.

Para cada *audio fingerprint* gerado, é criado um registro para identificar a qual áudio ele se refere, e indicando o *offset* deste *fingerprint* dentro do áudio original, assim como o *hash* gerado a partir das frequências resultantes deste trecho.

3.3.2.5 Busca de áudio

A pesquisa do áudio consiste de duas etapas: busca de similaridades (seção 3.3.2.5.1) e classificação dos resultados (seção 3.3.2.5.2).

3.3.2.5.1 Busca de similaridades

A busca de similaridades consiste em pegar cada uma das *fingerprints* geradas a partir do áudio utilizado como *query* e buscar na base por *fingerprints* que possuam *hash* igual. A partir disso, é montada uma lista de similaridades, que é formada por pares de *fingerprints* que possuem mesmo *hash*¸ onde um *fingerprint* é oriundo da base, e outro da *query*.

Um exemplo de busca de similaridades pode ser visualizado na Figura 14.

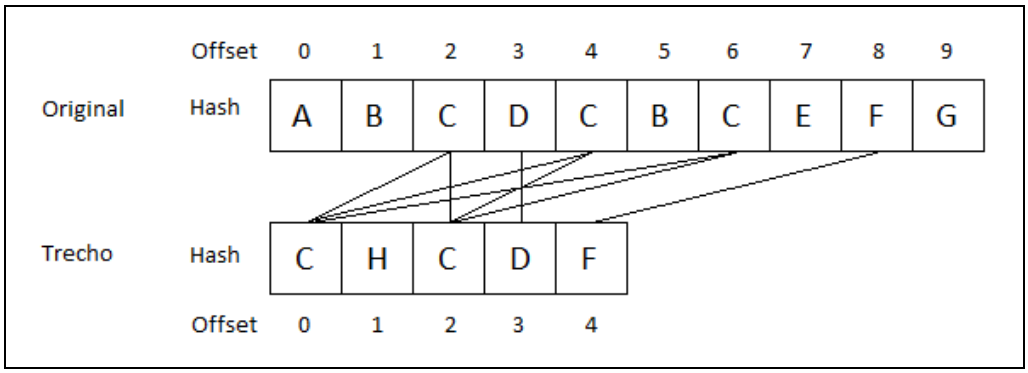


Figura 14 – Demonstração da busca de similaridades

A Figura 14 mostra um exemplo fictício de uma busca de similaridades. Nessa figura são visualizados dois áudios: `Original`, que representa o áudio da base de dados, e `Trecho`, que representa um trecho gravado, utilizado como *query*. Os *fingerprints* são conjuntos de `Hash` e `Offset`, e os *hashes* estão representados através de letras, visando simplificar o exemplo. A lista de similaridades deste exemplo se formaria pelos pares de *fingerprints* que estão ligados por uma linha. É importante salientar que, apesar deste exemplo mostrar o processo de busca relacionado a um áudio, o mesmo processo ocorre com todos os áudios contidos na base que possuam pelo menos um *fingerprint* com *hash* igual a algum *fingerprint* do trecho buscado. Nas próximas seções serão discutidos os métodos de classificação de implementados. Tais métodos foram empregados dos trabalhos relacionados e, replicados ou otimizados neste trabalho.

3.3.2.5.2 Classificação dos resultados

A partir da lista de similaridades obtida na seção 3.3.2.5.1, é necessário definir um grau de similaridade entre o trecho de áudio gravado e cada áudio da base. Para tanto, foram implementadas algumas técnicas de comparação para classificar os resultados obtidos, cujo objetivo era aumentar a confiabilidade dos resultados. A entrada das técnicas é uma lista com todas as similaridades encontradas através da busca de similaridades, e a saída gerada pelas técnicas será uma lista que terá um par de valores (grau de similaridade e a referência do áudio original) ordenados pelo grau de similaridade. O grau de similaridade é definido pela quantidade de *fingerprints* que a técnica identificou como sendo relevantes na comparação da *query* em relação a um determinado áudio. Nas próximas seções são descritas as cinco técnicas implementadas no presente trabalho.

3.3.2.5.2.1 Classificação por total de resultados

Nesta técnica apenas são contados o total de resultados obtidos para cada áudio, ou seja, a quantidade de elementos da lista de similaridades que correspondem a um determinado áudio do repositório. A Figura 15 demonstra como a classificação por total de resultados é composta.

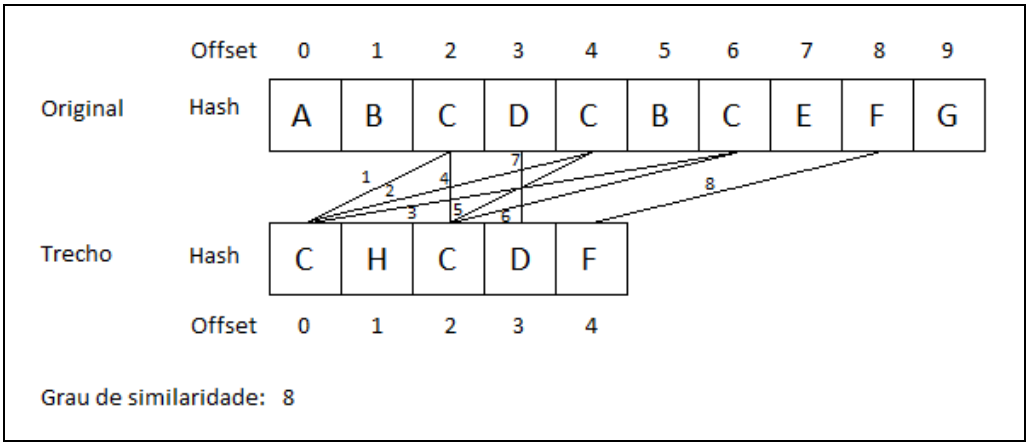


Figura 15 – Exemplo de classificação por total de resultados

A contagem da lista é representada através de uma numeração ao lado das linhas que formam as similaridades. Esta técnica também pode ser utilizada como forma de estatística, pois os resultados apresentados indicam a quantidade de resultados retornados pela busca de similaridades, para cada áudio do repositório de dados. No caso da Figura 15 o grau de similaridade é 8, pois existem 8 pares de *fingerprints* relacionados. É importante observar que nesta técnica o que vale é a ocorrência/relacionamento entre os *hashes*. Onde se ignora a distribuição em relação ao tempo. O algoritmo que implementa esta técnica pode ser visualizado no quadro 13.

```
public class MatchesCountRankingStrategy implements RankingStrategy {

    @Override
    public AudioRanking rank(FingerprintMatch[] matchingFps) {
        Bag bag = new HashBag();
        for (FingerprintMatch match: matchingFps) {
            // just add this source to the bag
            AudioInfo source =
match.getRepositoryFingerprint().getSource();
            bag.add(source);
        }

        final AudioRankingBuilder builder = new AudioRankingBuilder();

        for (Object obj : bag.uniqueSet()) {
            AudioInfo audio = (AudioInfo) obj;
            // retrieve the amount of similarities for this audio
            int count = bag.getCount(audio);

            builder.add(audio, count);
        }

        return builder.getAudioRanking();
    }
}
```

Quadro 13 – Implementação da classe `MatchesCountRankingStrategy`

O algoritmo apresentado no quadro 13 funciona da seguinte maneira: inicialmente é criado um `HashBag` (APACHE SOFTWARE FOUNDATION, 2008) para manter a quantidade de resultados para cada áudio do repositório. Realiza-se uma iteração sobre os elementos da lista de similaridades, representadas pelo vetor `matchingFps`, onde o áudio de origem de cada *fingerprint* da base é inserido no `HashBag`. Uma instância de `AudioRankingBuilder` é criada para gerar a classificação final dos áudios. A partir disso, é contraído cada áudio contido no `HashBag`, assim como a contagem de vezes que aquele áudio foi inserido no mesmo, estas duas informações são inseridas no `AudioRankingBuilder`. Após realizar este procedimento em todos os áudios contidos no `HashBag`, é retornada a classificação gerada pelo `AudioRankingBuilder`.

3.3.2.5.2.2 Classificação por acertos únicos de *fingerprints* similares na base

Nesta técnica são contados para o resultado final, apenas a quantidade de *fingerprints* do repositório contidos na lista de similaridades de um áudio, sem aceitar repetições. Por exemplo: se dois ou mais *fingerprints* da *query* geram o mesmo *hash*, eles trarão os mesmos resultados numa busca dentro do repositório, portanto, estes *fingerprints* similares que se repetirem, serão contados como apenas uma similaridade na primeira vez que forem encontrados, nas próximas vezes serão descartados. Um exemplo desta técnica pode ser visto na Figura 16.

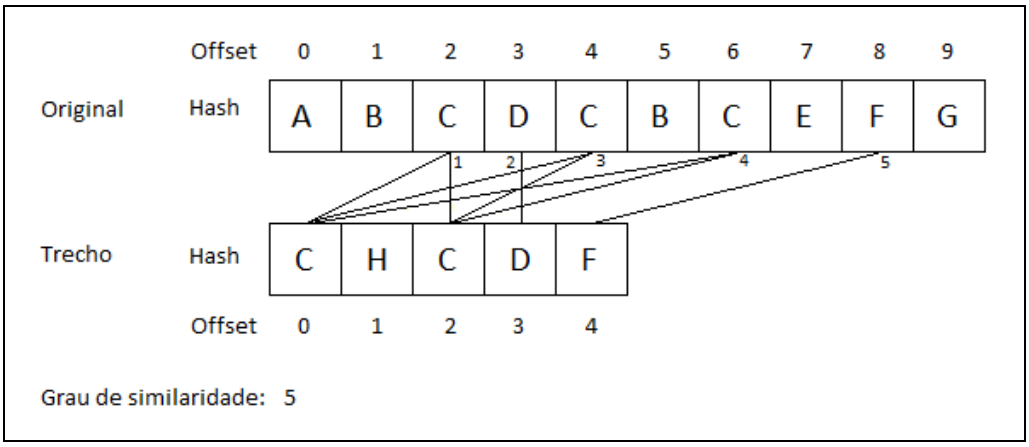


Figura 16 – Exemplo de classificação por acertos únicos de *fingerprints* similares na base

Na Figura 16 pode-se observar o exemplo da classificação por acertos únicos na base, onde a contagem para definição do grau de similaridade é representada através de uma numeração abaixo dos *hashes* do áudio da base que possuem algum *fingerprint* com *hash* correspondente do trecho gravado. Assim como a técnica definida na seção3.3.2.5.2.1, esta técnica também pode ser utilizada como forma de estatística, pois os resultados apresentados por ela indicam a quantidade de *fingerprints* do áudio da base que foram retornados pela busca de similaridades. No caso da Figura 16 o grau de similaridade é 5, pois considera-se apenas a quantidade de *fingerprints* relacionados com o áudio da base. O algoritmo que implementa esta técnica pode ser visualizado no quadro 14.

```
public class UniqueRepositoryMatchesRankingStrategy implements
RankingStrategy {

    @Override
    public AudioRanking rank(FingerprintMatch[] matchingFps) {
        Map<AudioInfo, Set<AudioFingerprint>>
audioRepositoryUniqueMatches = new HashMap<AudioInfo,
Set<AudioFingerprint>>();

        for (FingerprintMatch match: matchingFps) {
            // just add this source to the bag
            AudioInfo source =
match.getRepositoryFingerprint().getSource();
            AudioFingerprint fingerprint =
match.getRepositoryFingerprint();
            add(audioRepositoryUniqueMatches, source, fingerprint);
        }

        final AudioRankingBuilder builder = new AudioRankingBuilder();

        for (Entry<AudioInfo, Set<AudioFingerprint>> entry :
audioRepositoryUniqueMatches.entrySet()) {
            AudioInfo audio = entry.getKey();
            int rankValue = entry.getValue().size();

            builder.add(audio, rankValue);
        }

        return builder.getAudioRanking();
    }

    private static void add(Map<AudioInfo, Set<AudioFingerprint>>
audioOffsets, AudioInfo source, AudioFingerprint fingerprint) {
        Set<AudioFingerprint> set = audioOffsets.get(source);
        if (set == null) {
            set = new HashSet<AudioFingerprint>();
            audioOffsets.put(source, set);
        }
        set.add(fingerprint);
    }
}
```

Quadro 14 - Implementação da classe `UniqueRepositoryMatchesRankingStrategy`

O algoritmo apresentado no quadro 14 funciona da seguinte maneira: primeiro é criado um mapeamento para referenciar um áudio com um conjunto de *fingerprints*. A lista com as similaridades é iterada, o áudio da base é obtido, e buscado no mapa por este item, se ainda não houver um conjunto relacionado a este áudio, um novo conjunto é criado, e associado a ele, se já houver um conjunto associado, este conjunto é obtido. Tendo este conjunto, o *fingerprint* da base contido no par de *fingerprints* similares é adicionado ao conjunto, sendo que, se este *fingerprint* já foi adicionado previamente, não irá gerar nenhuma alteração no conjunto, pois o conjunto não adiciona itens duplicados. Após executar esta ação para todos os itens da lista de similaridades, é criada uma instância de `AudioRankingBuilder`, nela é adicionada cada áudio contido no mapa, junto da quantidade de itens no conjunto associado a ele. Então é retornada a classificação gerada pelo `AudioRankingBuilder`.

3.3.2.5.2.3 Classificação por acertos únicos de *fingerprints* da *query*

Nesta técnica, cada *fingerprint* gerado a partir da *query* e, que possui alguma relação com o áudio base, será contado apenas uma vez na definição do grau de similaridade. Por exemplo: se uma *fingerprint* retornar uma única *fingerprint* similar no repositório para um determinado áudio, esta será contada como uma similaridade neste áudio. Se a *fingerprint* retornar duas ou mais *fingerprints* similares no repositório para este áudio, será contada apenas uma similaridade também. Se a *fingerprint* não retornar nenhuma similaridade no repositório para este áudio, não será contada similaridade a mais para este áudio. Um exemplo desta técnica é apresentado na Figura 17.

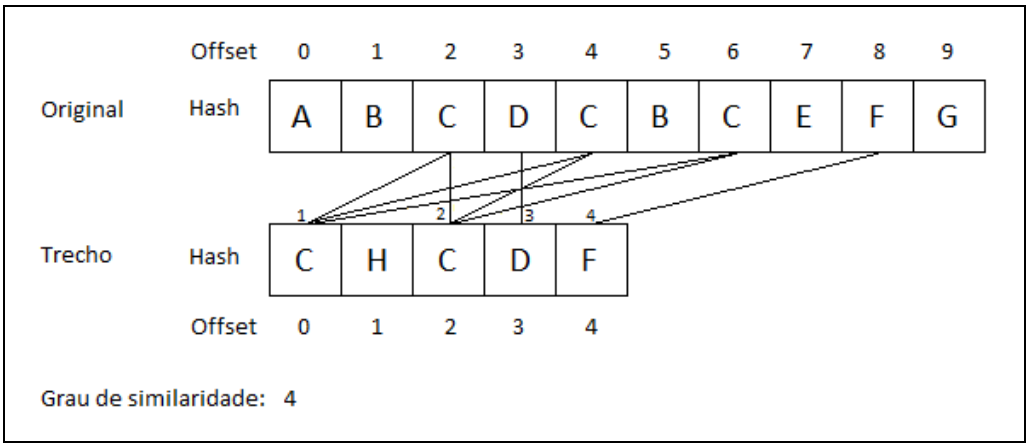


Figura 17 – Exemplo de classificação por acertos únicos de *fingerprints* da *query*

Na Figura 17 têm-se um exemplo da classificação por acertos únicos na *query*, onde a contagem para definição do grau de similaridade é representada através de uma numeração acima dos *hashes* do áudio da *query* que possuem algum *fingerprint* com *hash* correspondente do áudio da base. Assim como as duas técnicas definidas nas seções anteriores, esta técnica também pode ser utilizada como forma estatística, pois os resultados apresentados por ela indicam a quantidade de *fingerprints* da *query* que foram retornados pela busca de similaridades. No caso da Figura 17 o grau de similaridade é 4, pois existem apenas 4 *hashes* da *query* relacionados com o áudio da base. O algoritmo que implementa esta técnica pode ser visualizado no quadro 15.

```
public class UniqueQueryMatchesRankingStrategy implements RankingStrategy
{

    @Override
    public AudioRanking rank(FingerprintMatch[] matchingFps) {
        Map<AudioInfo, Set<AudioFingerprint>> audioQueryUniqueMatches =
new HashMap<AudioInfo, Set<AudioFingerprint>>();

        for (FingerprintMatch match: matchingFps) {
            // just add this source to the bag
            AudioInfo source =
match.getRepositoryFingerprint().getSource();
            AudioFingerprint fingerprint = match.getQueryFingerprint();
            add(audioQueryUniqueMatches, source, fingerprint);
        }

        final AudioRankingBuilder builder = new AudioRankingBuilder();

        for (Entry<AudioInfo, Set<AudioFingerprint>> entry :
audioQueryUniqueMatches.entrySet()) {
            AudioInfo audio = entry.getKey();
            int rankValue = entry.getValue().size();

            builder.add(audio, rankValue);
        }

        return builder.getAudioRanking();
    }

    private static void add(Map<AudioInfo, Set<AudioFingerprint>>
audioOffsets, AudioInfo source, AudioFingerprint fingerprint) {
        Set<AudioFingerprint> set = audioOffsets.get(source);
        if (set == null) {
            set = new HashSet<AudioFingerprint>();
            audioOffsets.put(source, set);
        }
        set.add(fingerprint);
    }
}
```

Quadro 15 - Implementação da classe `UniqueQueryMatchesRankingStrategy`

O algoritmo apresentado no quadro 15 funciona da seguinte maneira: primeiro é criado um mapeamento para referenciar um áudio com um conjunto de *fingerprints*. A lista com as similaridades é iterada, o áudio da base é obtido e buscado no mapa, se ainda não houver um conjunto relacionado a este áudio, um novo conjunto é criado e associado a ele, se já houver um conjunto associado, este conjunto é obtido. Tendo este conjunto, o *fingerprint* da *query* contido no par de *fingerprints* similares é adicionado ao conjunto, sendo que, se este *fingerprint* já foi adicionado previamente, não gerará nenhuma alteração no conjunto, pois o conjunto não adiciona itens duplicados. Após executar esta ação para todos os itens da lista de similaridades, é criada uma instância de `AudioRankingBuilder`, nela é adicionada cada áudio contido no mapa, junto da quantidade de itens no conjunto associado a ele. Então é retornada a classificação gerada pelo `AudioRankingBuilder`.

3.3.2.5.2.4 Classificação por maior subsequência crescente de tempo

Nesta técnica, é utilizado um algoritmo de programação dinâmica para encontrar a

maior subsequência crescente de tempo (*offset*) das similaridades. Nesta verificação é considerado tanto o *offset* do *fingerprint* similar (da base), quando o *offset* do *fingerprint* utilizado como termo de pesquisa. Ou seja, o resultado para um determinado áudio, é o tamanho da maior subsequência encontrada, tal que, duas similaridades podem fazer parte de uma mesma subsequência se, e somente se, o *offset* do *fingerprint* da base, e o *offset* do *fingerprint* da *query* de uma similaridade, são respectivamente menores que o *offset* do *fingerprint* da base, e o *offset* do *fingerprint* da *query* da outra similaridade. Um exemplo desta técnica é apresentado na Figura 18.

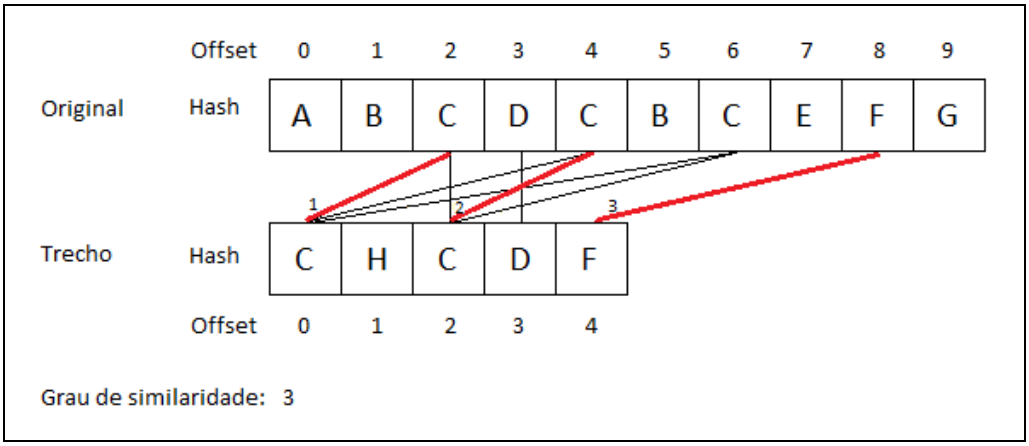


Figura 18 – Exemplo de classificação por maior subsequência crescente de tempo

Na Figura 18 é apresentado um exemplo da classificação por identificação de maior subsequência crescente de tempo entre pares de *fingerprints* similares, onde a maior subsequência está apresentada através de linhas vermelhas, e ao lado destas linhas uma numeração indica o comprimento da mesma até aquele momento. Percebe-se que outras subsequências de mesmo tamanho podem ser obtidas, como por exemplo, ao retirar a segunda ligação da subsequência representada na Figura 18 pela ligação existente entre o *offset* 3 de `Trecho` e o *offset* 3 de `Original`. Mesmo sabendo que outras subsequências de mesmo tamanho podem ser obtidas a partir deste exemplo, esta informação é irrelevante, pois esta técnica só leva em consideração o tamanho da maior subsequência, e não de quantas maneiras este tamanho de subsequência pode ser obtido. No caso da Figura 18, o grau de similaridade é 3, pois este é o valor da maior subsequência que se pode encontrar, a exemplo da subsequência definida na imagem. O algoritmo com a implementação desta técnica é apresentado no quadro 16.

```
public class LongestIncreasingSequenceOfMatchesRankingStrategy implements
RankingStrategy {

      private static final Comparator<FingerprintMatch>
QUERY_SMALLER_OR_REPOSITORY_IF_EQUALS = new
Comparator<FingerprintMatch>() {

            @Override
            public int compare(FingerprintMatch o1, FingerprintMatch o2)
{
                  int delta = o1.getQueryFingerprint().getOffset() -
o2.getQueryFingerprint().getOffset();
                  if (delta == 0) {
                        delta = o1.getRepositoryFingerprint().getOffset()
- o2.getRepositoryFingerprint().getOffset();
                  }
                  return delta;
            }
      };

      @Override
      public AudioRanking rank(FingerprintMatch[] matchingFps) {
            Map<AudioInfo, List<FingerprintMatch>> map = new
HashMap<AudioInfo, List<FingerprintMatch>>();

            for (FingerprintMatch match : matchingFps) {
                  add(map, match.getRepositoryFingerprint().getSource(),
match);
            }

            final AudioRankingBuilder builder = new
AudioRankingBuilder();

            for (Entry<AudioInfo, List<FingerprintMatch>> entry :
map.entrySet()) {
                  AudioInfo audio = entry.getKey();
                  FingerprintMatch[] matches =
entry.getValue().toArray(new FingerprintMatch[0]);

                  int bestSequence =
findLongestIncreasingSequence(matches);

                  builder.add(audio, bestSequence);
            }

            return builder.getAudioRanking();
      }

      private int findLongestIncreasingSequence(FingerprintMatch[]
matches) {
            int matchesLen = matches.length;

            /*
             * first sort by smallest query offset (or repository offset,
if query offset equals) so that matches[i] is 'smaller' than matches[i+1]
(in
             * this case 'smaller' means that it has a smaller query
offset, or that the query offsets are equal, but it is smaller in the
repository
             * offset).
             *
             * This makes the search faster, because with the entries
ordered we solve it in O(n^2)
             */
            Arrays.sort(matches, QUERY_SMALLER_OR_REPOSITORY_IF_EQUALS);

            int best = 1;
            int[] DP = new int[matchesLen];
```

```
            DP[0] = 1;
            for (int i = 1; i < matchesLen; i++) {
                  DP[i] = 1;

                  for (int aPrev = 0; aPrev < i; aPrev++) {
                        // if it makes a sequence...(this also implicitly
checks uniqueness of both query fingerprint and repository fingerprint in
the
                        // sequence)
                        if
(queryAndRepositoryOffsetSmaller(matches[aPrev], matches[i])) {
                              // checks if it is the best sequence
                              if (DP[aPrev] + 1 > DP[i]) {
                                    // get this as the new best sequence
from 0 to i
                                    DP[i] = DP[aPrev] + 1;
                              }
                        }
                  }
                  // if DP[i] is a longest sequence than 'best', update
'best'
                  if (DP[i] > best) {
                        best = DP[i];
                  }
            }

            return best;
      }

      private static boolean
queryAndRepositoryOffsetSmaller(FingerprintMatch a, FingerprintMatch
test) {
            return a.getQueryFingerprint().getOffset() <
test.getQueryFingerprint().getOffset() &&
a.getRepositoryFingerprint().getOffset() <
test.getRepositoryFingerprint().getOffset();
      }

      private static void add(Map<AudioInfo, List<FingerprintMatch>> map,
AudioInfo source, FingerprintMatch match) {
            List<FingerprintMatch> list = map.get(source);
            if (list == null) {
                  list = new ArrayList<FingerprintMatch>();
                  map.put(source, list);
            }
            list.add(match);
      }

}
```

Quadro 16 - Implementação da classe `LongestIncreasingSequenceOfMatchesRankingStrategy`

O algoritmo apresentado no quadro 16 funciona da seguinte maneira: primeiramente é criado um mapa para armazenar a lista de similaridades de cada áudio. Para alimentar este mapa, é iterado pelo vetor `matchingFps`, identificado o áudio original do *fingerprint* da base, e posteriormente o par de *fingerprints* similares é adicionado na lista daquele áudio. Após montadas as listas, é criada uma instância de `AudioRankingBuilder`, para manter os áudios e seus níveis de similaridade. A partir disso, itera-se pelo mapa para se obter a lista de similaridades de cada áudio. Esta lista de similaridades é ordenada de modo que os pares cujo *offset* do *fingerprint* da *query* sejam menores apareçam sempre na frente dos outros, e caso

dois pares possuam o mesmo *fingerprint* da *query*, é colocado na frente o par que possui *offset* do *fingerprint* da base menor. Com a lista ordenada, é criado um vetor `DP` de `N` posições, onde `N` é a quantidade de elementos da lista ordenada de similaridades, este vetor é utilizado para manter o tamanho da maior subsequência obtida da posição zero até a posição indicada pelo índice do vetor, em que este índice é incluído na subsequência. Para realizar isto, inicializa-se o índice zero com o valor um (para indicar que do índice zero até ele mesmo, é possível conseguir apenas uma subsequência de um elemento, que é o elemento contido no próprio índice zero), depois é realizada a iteração, onde a variável `i` recebe como valor o índice um, e vai até o último índice do vetor. Dentro desta iteração é inicializado o vetor `DP` no índice `i` da iteração com o valor um, aí uma outra iteração, interna a esta, é realizada, onde uma variável `aPrev` recebe zero como valor, e vai até o valor `i-1`. Dentro desta iteração, é verificado se os *offsets* dos *fingerprints* da base e da *query* no índice `aPrev` são respectivamente menores que os *offsets* dos *fingerprints* da base e da *query* no índice `i`, e se o conteúdo de `DP[i]` for menor que `DP[aPrev]+1`, se ambas as condições forem verdadeiras, o valor contido em `DP[i]` é atualizado para `DP[aPrev]+1`. O maior valor contido no vetor `DP` é o tamanho da maior subsequência contida na lista de similaridades para o áudio em questão. O áudio em questão e o tamanho de sua maior subsequência são adicionados à instância de `AudioRankingBuilder`. Após o mesmo processamento ser realizado em todos os áudios, é retornada a classificação gerada no `AudioRankingBuilder`.

3.3.2.5.2.5 Classificação por variação de tempo entre *fingerprint* da base e *fingerprint* da *query*

Nesta técnica, são verificadas as variações de tempo entre os *fingerprints* da *query* e das suas similaridades encontradas no repositório, que nada mais é do que a subtração do *offset* do *fingerprint* do repositório pelo *offset* do *fingerprint* da *query*. Para cada variação de tempo que se repete, é somado um na pontuação da classificação para o áudio correspondente. Esta técnica prevê que as correspondências corretas do áudio em relação à *query* tendem a estar sempre a uma distância igual referente ao início de cada áudio, gerando um resultado que potencialmente terá menos falsos positivos. A ideia desta técnica é a mesma da aplicação de uma janela de Hamming (ver seção 2.3), pois o ato de contabilizar a quantidade de variações gera o mesmo resultado de se mover uma janela dos *fingerprints* do trecho sobre os *fingerprints* do áudio da base, e verificar a quantidade de *fingerprints* sobrepostos que possuem o mesmo valor. Um exemplo desta técnica é apresentado na Figura 19.

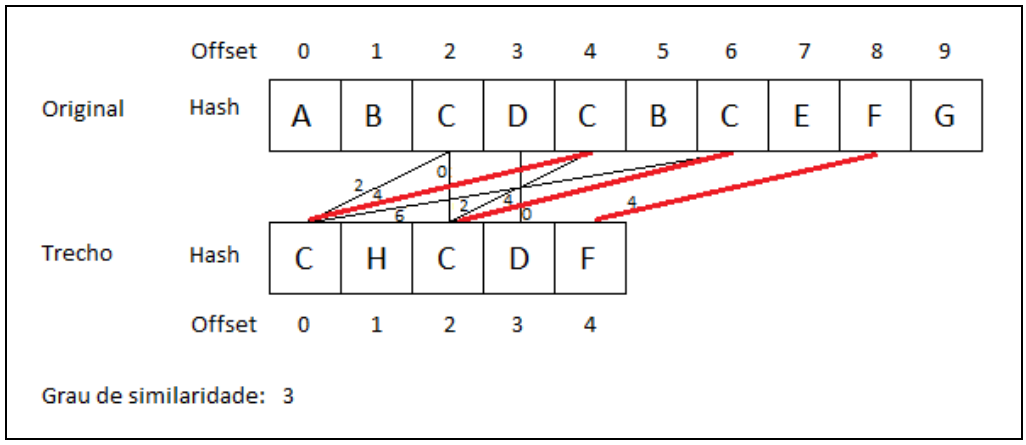


Figura 19 – Exemplo de classificação por variação de tempo

Na Figura 19 é apresentado um exemplo da classificação por variação de tempo, onde ao lado das linhas que ligam *hashes* de *fingerprints* da base com *fingerprints* da *query*, são apresentados os valores calculados das diferenças de *offset* entre estes *fingerprints*. O melhor resultado possível para este exemplo está com as linhas destacadas em vermelho. No caso da Figura 19, o grau de similaridade é 3, pois este é o maior número de similaridades que geraram uma mesma diferença de *offsets*, que neste caso foi a diferença 4. No quadro 17 é apresentada a implementação desta técnica.

```
public class SameDeltaOffsetsOfMatchesRankingStrategy implements
RankingStrategy {

    @Override
    public AudioRanking rank(FingerprintMatch[] matchingFps) {
        Map<AudioInfo, List<FingerprintMatch>> map = new
HashMap<AudioInfo, List<FingerprintMatch>>();

        for (FingerprintMatch match : matchingFps) {
            add(map, match.getRepositoryFingerprint().getSource(),
match);
        }

        final AudioRankingBuilder builder = new AudioRankingBuilder();

        for (Entry<AudioInfo, List<FingerprintMatch>> entry :
map.entrySet()) {
            AudioInfo audio = entry.getKey();
            List<FingerprintMatch> matches = entry.getValue();

            // the biggest amount of equal delta of offsets found
            int max = 0;

            Bag bag = new HashBag();
            for (FingerprintMatch match : matches) {
                AudioFingerprint queryFp = match.getQueryFingerprint();
                AudioFingerprint repositoryFp =
match.getRepositoryFingerprint();

                // computes the difference
                int deltaOffset = repositoryFp.getOffset() -
queryFp.getOffset();

                bag.add(deltaOffset);
                max = Math.max(max, bag.getCount(deltaOffset));
            }

            builder.add(audio, max);
        }

        return builder.getAudioRanking();
    }

    private static void add(Map<AudioInfo, List<FingerprintMatch>> map,
AudioInfo source, FingerprintMatch match) {
        List<FingerprintMatch> list = map.get(source);
        if (list == null) {
            list = new ArrayList<FingerprintMatch>();
            map.put(source, list);
        }
        list.add(match);
    }

}
```

Quadro 17 - Implementação da classe `SameDeltaOffsetsOfMatchesRankingStrategy`

O algoritmo contido no quadro 17 funciona da seguinte maneira: primeiro é criado um mapa para armazenar a lista de similaridades de cada áudio. Para alimentar este mapa, é iterado pelo vetor `matchingFps`, identificado o áudio original do *fingerprint* da base, e então o par de *fingerprints* similares é adicionado na lista daquele áudio. Uma instância de `AudioRankingBuilder` é criada para armazenar o grau de similaridade de cada áudio, e em seguida, para cada áudio do mapa criado é obtida a lista de similaridades gerada para ele.

Então é iterado pela lista de similaridades deste áudio, onde a diferença entre o *offset* do *fingerprint* da base e o *offset* do *fingerprint* da *query* é calculada, a diferença de *offsets* que mais ocorrer tem sua quantidade utilizada como grau de similaridade do áudio. O áudio e este grau de similaridade são adicionados ao `AudioRankingBuilder`. Depois de repetir tal procedimento em todos os áudios, é retornada a classificação gerada pelo `AudioRankingBuilder`.

3.3.3 Operacionalidade da implementação

Nesta seção será apresentada a operacionalidade do mecanismo de busca desenvolvido. Esta seção é dividida em duas partes: registro de novos áudios e busca de áudio.

3.3.3.1 Registro de novos áudios

Nesta seção é apresentado o registro de novos áudios (UC01). Na Figura 20 é apresentada a tela que foi desenvolvida para realizar o carregamento de novos áudios na base de *fingerprints*.

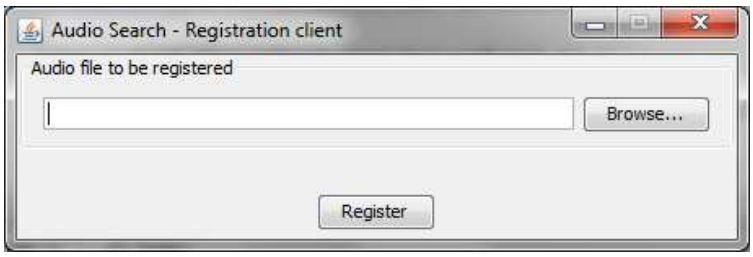


Figura 20 – Tela de registro de áudio

Nesta tela (Figura 20), é possível digitar o caminho do arquivo de áudio, ou selecioná-lo através da navegação do botão `Browse...`.

Ao pressionar o botão `Register`, o arquivo contido no campo de texto é lido, convertido para o formato de áudio utilizado pelo mecanismo (definido na seção 3.3.2.1), enviado ao aplicativo servidor para que seja registrado na base.

3.3.3.2 Busca de áudio

Nesta seção é apresentada a busca de áudios (UC02). A Figura 21 apresenta a tela desenvolvida para realizar a gravação de um áudio, e sua subsequente busca na base de *fingerprints*.

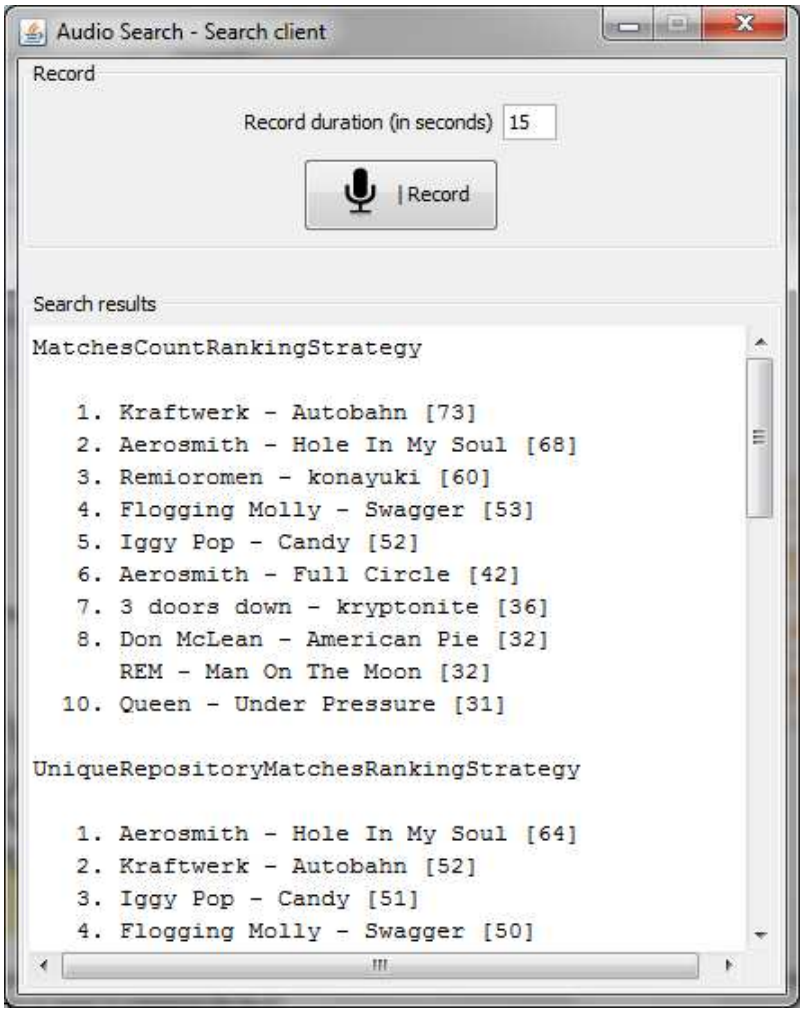


Figura 21 – Tela de busca de áudio

Nesta tela (Figura 21), é possível definir uma duração de gravação (em segundos) no campo de texto `Record duration (in seconds)`, podendo-se iniciar uma gravação através do botão `Record`. Ao pressionar o botão `Record`, é iniciada uma gravação de áudio, que dura a quantidade de segundos definida no campo de texto citado anteriormente. Esta gravação é realizada através do microfone padrão encontrado automaticamente no computador onde o programa está sendo executado. Após finalizar a gravação do áudio, ele é enviado ao servidor. O servidor processa a busca, retorna os resultados encontrados e os exibe na área de texto identificada por `Search results`.

3.4 RESULTADOS E DISCUSSÃO

Para validar o mecanismo de busca proposto, foram realizados três experimentos, que são descritos na sequência.

No primeiro experimento, o objetivo era analisar a eficiência das técnicas de similaridade implementadas (ver seção 3.3.2.5.2). Para estabelecer a eficiência dessas técnicas, analisou-se a quantidade de falsos positivos, ou seja, o grau de similaridade identificado para as músicas corretas em relação às incorretas. Para este experimento, montou-se uma base de dados com 10 áudios, dos quais 3 deles referem-se a diferentes versões da música Basket Case, da banda Green Day, sendo: uma versão de estúdio de 3 minutos e 3 segundos (3945 *fingerprints*), uma versão ao vivo com poucos ruídos de 2 minutos e 52 segundos (3722 *fingerprints*), e uma versão ao vivo com bastante ruído de 3 minutos e 43 segundos (4816 *fingerprints*). Ao qual, se estas músicas ficassem entre as primeiras, ou seja, obtivessem um grau de similaridade maior que as outras músicas, se estabelecia que a técnica em questão produz bons resultados, e se as outras músicas tiverem um resultado baixo (grau de similaridade 1 ou 0), então a técnica de comparação em questão produz um baixo número de falsos positivos.

Este experimento foi realizado da seguinte maneira: foram dados como entrada para o mecanismo de busca três gravações, com durações de: 5 segundos (107 *fingerprints*), 15 segundos (322 *fingerprints*) e 30 segundos (645 *fingerprints*). Estas entradas referem-se a gravações feitas através do microfone de um notebook Toshiba, com processador Intel Core 2 Duo 1.30 GHz, 3 GB de memória RAM, rodando o sistema operacional Windows 7 de 64 bits, tendo como dispositivo emissor do áudio de entrada um celular Samsung I5500 Galaxy 5, processador ARM 11 de 600MHz. Na tabela 1, são mostradas as pontuações alcançadas por cada técnica de similaridade na busca da música de estúdio.

Tabela 1 – Resultados do experimento 1 para a música de estúdio

RESULTADOS DO EXPERIMENTO 1 PARA A MÚSICA DE ESTÚDIO			
	Estúdio		
	5s	**15s**	**30s**
Total de resultados	37	105	150
Acertos únicos na base	37	57	96
Acertos únicos na *query*	11	37	80
Maior subseqüência de tempo	7	26	54
Variação de tempo	5	21	49

A partir da tabela 1, pode-se notar que as técnicas de similaridade apresentam um nível de similaridade maior entre o trecho gravado e o áudio da base conforme o tempo do trecho aumenta. Isso se explica pelo fato de que quanto maior o trecho, mais características comuns são encontradas entre o trecho gravado e o áudio da base. A primeira vista tem-se a percepção de que técnicas que geram valores mais altos de similaridade são as melhores, porém será visto nos próximos experimentos que isso nem sempre é verdade. Isso se deve ao fato de que as três primeiras técnicas relacionam partes dos áudios sem levar em consideração a informação do tempo em que ocorrem, além disso, também consideram a mesma informação mais de uma vez. As duas últimas técnicas computam características iguais a partir de subseqüências de mesmo tamanho.

Na tabela 2 são apresentadas as pontuações alcançadas pelas técnicas de similaridade em cada uma das principais músicas da base, utilizando um trecho de 30 segundos, pois trechos desta natureza produzem melhores resultados, conforme visto na tabela 1.

Tabela 2 – Resultados do experimento 1 com trechos de 30 segundos

RESULTADOS DO EXPERIMENTO 1 COM TRECHOS DE 30 SEGUNDOS				
	Estúdio	**Ao vivo**	**Ao vivo (ruídos)**	**Outras**
Total de resultados	150	26	5	15
Acertos únicos na base	96	15	3	14
Acertos únicos na *query*	80	18	5	15
Maior subseqüência de tempo	54	7	2	6
Variação de tempo	49	2	1	1

A partir da tabela 2 pode-se observar que a música de estúdio obteve melhores resultados em todas as situações, seguida da música ao vivo que possuía poucos ruídos, enquanto a música que possui muitos ruídos não trouxe resultados bons, e inclusive chegando a gerar resultados piores do que outras músicas que não tinham relação alguma com o áudio buscado. Quanto aos falsos positivos, a única técnica de similaridade que apresentou um baixo número de falsos positivos foi à técnica de similaridade por variação do tempo (apresentada na seção 3.3.2.5.2.5), pois todas as outras técnicas identificaram que outras

músicas eram mais similares ao trecho informado do que a música ao vivo com ruídos.

Com este experimento, pode-se concluir que é possível obter resultados positivos a partir das técnicas de similaridade implementadas. Quando o áudio de entrada é exatamente o mesmo que o áudio da base (caso da música de estúdio), todas as técnicas geram resultados satisfatórios, porém, apenas a técnica de similaridade por variação de tempo (seção 3.3.2.5.2.5) gera um número baixo de falsos positivos. Quando o áudio buscado possui algumas diferenças do áudio da base (caso das músicas ao vivo), é possível obter resultados bons, mas o número de falsos positivos é elevado.

O segundo experimento realizado teve por objetivo analisar a eficácia das técnicas de similaridade apresentadas da seção 3.3.2.5.2 na identificação correta de músicas, quando a busca é realizada a partir de um áudio com pouco ruído. Para estabelecer a eficácia das técnicas, foi analisada a porcentagem de vezes que ela gerou o resultado correto. Para isso foi utilizada uma base contendo 85 músicas, de diferentes estilos, onde todas as músicas são versões de estúdio. A música utilizada como *query* foi a música Kryptonite, da banda 3 Doors Down. Para a busca, foram definidos novamente que seriam utilizados trechos de 5 segundos, 15 segundos e 30 segundos. Para cada um destes três tamanhos de trecho de áudio, foram realizadas cinco buscas, gravando trechos diferentes do áudio em questão. Na tabela 3, é mostrada a porcentagem de vezes que a música correta apareceu de maneira isolada como melhor resultado de cada técnica.

Tabela 3 – Resultados do experimento 2

RESULTADOS DO EXPERIMENTO 2			
	5s	**15s**	**30s**
Total de resultados	20%	0%	0%
Acertos únicos na base	0%	0%	0%
Acertos únicos na *query*	20%	0%	0%
Maior subseqüência de tempo	20%	40%	20%
Variação de tempo	80%	60%	100%

As informações foram dispostas em colunas, representando na vertical os tempos de gravação e as linhas apresentam a técnica de similaridade aplicada, e a porcentagem de acertos que ela teve dentre os testes realizados.

A partir da tabela 3 pode-se observar que as três primeiras técnicas (descritas nas seções 3.3.2.5.2.1, 3.3.2.5.2.2 e 3.3.2.5.2.3) não obtiveram bons resultados. A técnica de maior subsequência (seção 3.3.2.5.2.4) obteve resultados pouco satisfatórios, e a técnica de variação de tempo (seção 3.3.2.5.2.5) gerou bons resultados, ressaltando que esta última técnica, nos testes onde se consideravam 30 segundos, obteve 100% de acerto, ou seja, obteve

o resultado correto em todos os 5 testes.

Diante dos resultados apresentados neste experimento, pode-se concluir que dentre as técnicas utilizadas, apenas a técnica de variação de tempo consegue efetivamente alcançar bons resultados, e que estes resultados têm uma tendência de serem mais eficientes de acordo com o tamanho do trecho de áudio.

Um terceiro experimento foi realizado com o objetivo de analisar a eficácia das técnicas na identificação correta das músicas diante de ruídos na gravação utilizada como *query*. A mesma definição de eficácia utilizada no segundo experimento foi utilizada neste experimento. Para este experimento foi utilizada a mesma base do experimento anterior, e a mesma música utilizada como *query*. A única diferença deste experimento, é que, enquanto cada *query* era gravada, eram gerados ruídos no ambiente para que fossem gravados junto do áudio da música. Estes ruídos gerados incluem: outras músicas sendo tocadas no ambiente, som de programas de TV sendo reproduzidos, batidas em superfícies de madeira, assovios, entre outros. A mesma quantidade de gravações do experimento anterior foi realizada, assim como os mesmos tamanhos de trechos de áudio foram gravados. Na tabela 4, são apresentadas as porcentagens de vezes que a música correta apareceu de maneira isolada como melhor resultado de cada técnica.

Tabela 4 – Resultados do experimento 3

RESULTADOS DO EXPERIMENTO 3			
	5s	**15s**	**30s**
Total de resultados	0%	20%	0%
Acertos únicos na base	0%	20%	0%
Acertos únicos na *query*	0%	60%	0%
Maior subseqüência de tempo	0%	20%	0%
Variação de tempo	20%	80%	100%

As informações foram dispostas em colunas, representando na vertical os tempos de gravação e as linhas apresentam a técnica de similaridade aplicada, e a porcentagem de acertos que ela teve dentre os testes realizados.

A partir da tabela 4 pode-se observar que as quatro primeiras técnicas (descritas nas seções 3.3.2.5.2.1, 3.3.2.5.2.2, 3.3.2.5.2.3 e 3.3.2.5.2.4) não obtiveram nenhum resultado satisfatório para trechos de 5 segundos e 30 segundos, sendo que para 15 segundos alguns resultados foram obtidos, inclusive um surpreendente 60% para a terceira técnica, sendo que nos testes sem ruídos foi atingido um resultado de 0% para a mesma técnica. Uma possível explicação para esta situação é que, devido aos ruídos, o trecho gravado gerou vários *fingerprints* com as mesmas características, e coincidentemente estas características também

estavam presentes em algum *fingerprint* do áudio original, fazendo com que a técnica gerasse uma pontuação alta para o áudio, pois a técnica em questão considera *fingerprints* do trecho gravado que tenham tido alguma correspondência (característica em comum) com o áudio da base. A técnica de variação de tempo (seção 3.3.2.5.2.5) novamente demonstrou bons resultados, sendo eles semelhantes aos obtidos no experimento anterior, sem ruídos.

Diante deste experimento, pode-se concluir que a técnica de variação consegue obter os melhores resultados, mesmo na presença de ruídos durante a gravação do áudio, pois na maioria dos testes ela conseguiu trazer corretamente o áudio reproduzido. E quanto maior o trecho de áudio gravado, maiores as chances de um resultado positivo ser atingido.

4 CONCLUSÕES

Este trabalho propôs o desenvolvimento de um mecanismo de busca de áudio, que extraísse características dos áudios e as utilizasse para localizar arquivos sonoros semelhantes dentro de um repositório, a partir de um trecho de áudio gravado pelo usuário.

Os objetivos estabelecidos neste trabalho foram alcançados. Através dele, é possível realizar a gravação de trechos de áudio para posterior busca; *audio fingerprints* foram utilizados para gerar uma representação do áudio baseada em suas características; e técnicas de similaridade foram implementadas para se obter resultados significativos na identificação de áudios. Os resultados obtidos foram bastante favoráveis. Mesmo sob degradações de sinal intensas, o mecanismo de busca manteve sua robustez e localizou corretamente a maioria dos trechos informados. Tais resultados nos levam a concluir a aplicabilidade de um mecanismo desta natureza em ferramentas de nível comercial.

Como limitação, pode-se citar que em alguns casos, o mecanismo de busca mostrou-se ineficiência para lidar com variações de gravações (mesma música, porém com tipos diferentes de gravação – *remix*, ao vivo, entre outros), pois nem sempre geram bons resultados. Todavia, os resultados pioram na presença de ruídos. Há também outros tipos de áudio que não foram testados, como sons produzidos por animais, sons de efeitos naturais (como chuva, raios).

Contudo, este trabalho poderá ser utilizado como referência para novos estudos sobre busca e extração de informações em áudio. Além disso, este mecanismo também poderá ser utilizado como recurso pessoal, para catalogar áudios e realizar buscas na base de áudios gerada.

4.1 EXTENSÕES

Como sugestão de extensões para o mecanismo propõem-se:

a) estudo e implementação de novas técnicas de similaridade, utilizando RBC (raciocínio baseado em caso);
b) testar e implementar a identificação de outros tipos de áudio, além de músicas;
c) disponibilizar para plataformas móveis.

REFERÊNCIAS BIBLIOGRÁFICAS

APACHE SOFTWARE FOUNDATION. **HashBag (Commons Collections 3.2.1)**. [S.l.], 2008. Disponível em: <http://commons.apache.org/collections/apidocs/org/apache/commons/collections/bag/HashBag.html>. Acesso em: 30 jun. 2011.

_____. **Math – Commons Math: The Apache Commons Mathematics Library**. [S.l.], 2011. Disponível em: <http://commons.apache.org/math>. Acesso em: 30 jun. 2011.

BUNNELL, Timothy. **Sound in the Frequency Domain**. [S.l.], fev. 1996a. Disponível em: <http://www.asel.udel.edu/speech/tutorials/acoustics/freq_domain.html>. Acesso em: 30 jun. 2011.

_____. **Sound in the Time Domain**. [S.l.], fev. 1996b. Disponível em: <http://www.asel.udel.edu/speech/tutorials/acoustics/time_domain.html>. Acesso em: 30 jun. 2011.

CANO, Pedro. **Content-based audio search**: from fingerprinting to semantic audio retrieval. 2006. 197 f. Tese (Doutorado em ciências da computação e comunicação digital) - Centro de Ciências Exatas e Naturais, Pompeu Fabra University, Barcelona, Espanha. Disponível em: <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.75.738&rep=rep1&type=pdf>. Acesso em 30 abr. 2011.

CASEY, Michael A. et al. **Content-based music information retrieval**: current directions and future challenges. PROCEEDINGS OF THE IEEE, [S.l.], v. 96, n. 4, p. 668-696, Apr., 2008.

H2 DATABASE. **H2 Database Engine**. [S.l.], [2011]. Disponível em: <http://www.h2database.com>. Acesso em: 30 jun. 2011.

HAITSMA, Jaap; KALKER, Ton. A highly robust audio fingerprinting system. In: INTERNATIONAL CONFERENCE ON MUSIC INFORMATION RETRIEVAL, 3., 2002, Paris. **Proceedings...** Paris: IRCAM, 2002. p. 107-115.

HARMONIC. **harmonic – Light Pure Java MIDI Synthesizer**. [S.l.], [2011]. Disponível em: <http://code.google.com/p/harmonic>. Acesso em: 30 jun. 2011.

IZUMITANI, Tomonori; KASHINO, Kunio. A robust musical audio search method based on diagonal dynamic programming matching of self-similarity matrices. In: INTERNATIONAL CONFERENCE ON MUSIC INFORMATION RETRIEVAL, 9., 2008, Philadelphia. **Proceedings...** Philadelphia: IRCAM, 2008. p. 609-613.

JAVAZOOM. **MP3 SPI for Java Sound**. [S.l.], [2010]. Disponível em: <http://www.javazoom.net/mp3spi/mp3spi.html>. Acesso em: 30 jun. 2011.

KALKER, Ton; HAITSMA, Jaap; OOSTVEN, Job. **Issues with digital watermarking and perceptual hashing**. In: PROCEEDINGS OF THE SPIE, Denver, v. 4518, p. 189-197, Nov., 2001.

KLINE, Richard L.; GLINERT, Ephraim P. Approximate matching algorithms for music information retrieval using vocal input. In: INTERNATIONAL MULTIMEDIA CONFERENCE, 11., 2003, Berkeley. **Proceedings...** Berkeley: ACM, 2003. p. 130-139.

MALIK, Hafiz. **Content-based audio indexing and retrieval:** an overview. [S.l.], 2002. Disponível em: <http://www-personal.engin.umd.umich.edu/~hafiz/CBAIR_survey.pdf>. Acesso em: 23 set. 2010

ORACLE. **Java Sound API**. [S.l.], [2010]. Disponível em: <http://www.oracle.com/technetwork/java/index-jsp-140234.html>. Acesso em: 30 jun. 2011.

SEO, Jin S. et al. Audio fingerprinting based on normalized spectral subband centroids. In: IEEE INTERNATIONAL CONFERENCE ON ACOUSTICS, SPEECH, AND SIGNAL PROCESSING, 30., 2005, Philadelphia. **Proceedings...** Philadelphia: IEEE, 2005. p. 213-216.

SMITH, Steven W. **The scientist and engineer's guide to digital signal processing**. [S.l.]: California Technical Publishing, 1997. Disponível em: <http://www.dspguide.com /pdfbook.htm>. Acesso em: 19 mar. 2011.

WANG, Avery L. C. An industrial strength audio search algorithm. In: INTERNATIONAL SYMPOSIUM ON MUSIC INFORMATION RETRIEVAL, 4., 2003, Baltimore. **Proceedings...** Baltimore: IRCAM, 2003. p. 7-13.

WIGNALL, Michael D. **Content-based music similarity.** 2003. 175 f. TCC (Bacharelado em engenharia de software) - The University of Queensland, Brisbane, Austrália. Disponível em: <http://www-ctp.di.fct.unl.pt/~fb/musicSite/docs/Content-Based%20%20Music%20Similarity.pdf>. Acesso em: 30 abr. 2011.