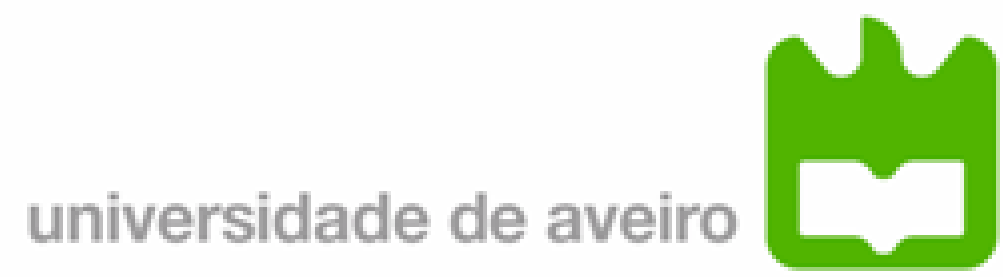

Relatório final - Projeto MPEI

Turma P3 - Diogo Daniel Soares Ferreira (76504) e Fábio Danilo Pinhal Cunha (76677)
Universidade de Aveiro - Métodos Probabilísticos para Engenharia Informática

Supervisionado por Professor Tomás Oliveira e Silva

16 de Dezembro de 2015

Conteúdo

Introdução

Este é um projeto feito no âmbito da disciplina de Métodos Probabilísticos para Engenharia Informática, da Universidade de Aveiro, lecionado no segundo ano curricular, no primeiro semestre, às cadeiras de Mestrado Integrado em Engenharia de Computadores e Telemática e Engenharia Informática. Neste projeto, foi-nos proposto desenhar uma aplicação, usando a linguagem *MatLab*, que implementasse dois métodos probabilísticos explicados nas aulas: o *Bloom Filter*[1] e o *MinHash*[2]. O tema da aplicação foi deixado à criatividade de cada grupo.

A nossa ideia inicial seria uma aplicação que comparasse duas músicas, usando as suas ondas e frequências, e manipulando-as através do *MatLab*. Rapidamente nos apercebemos de que não seria possível, pois o tempo disponível para o projeto não nos permitiria analisar com tanto detalhe o processo de comparação de duas músicas. Depois, tivemos a ideia de comparar as músicas pelas suas letras. No entanto, a falta de bases de dados de letras de músicas disponíveis na *internet* e as questões legais associadas às editoras afastaram-nos relutantemente do projeto. Finalmente, decidimos apenas considerar duas músicas similares pelas palavras presentes no seu título. A aplicação funcionará assim como um "rádio"simples modificado com uma base de dados de músicas, onde se pode reproduzir alguma música ou pesquisar por nome a música desejada.

A aplicação começará por identificar todas as músicas em formato *mp3* presentes em todos os diretórios e sub-diretórios de um ficheiro definido. Todas as músicas devem ser guardadas na base de dados, exceto se encontrem nomes iguais. Nesse caso, para manter a integridade da aplicação, uma delas será descartada.

Seguidamente, será apresentado um menu com cinco opções. A primeira permitirá ao utilizador ouvir uma música (durante um tempo inserido pelo utilizador), escolhida com base no seu número (*index*). A segunda opção dará a oportunidade ao utilizador de listar todas as músicas similares à expressão introduzida, apresentado também o seu *index*. O menu também deixará ao utilizador a opção de apagar a base de dados e inserir um diretório a partir de onde deseja que as suas músicas sejam carregadas. Na quarta opção, a aplicação poderá selecionar uma música aleatória pertencente à sua base de dados e reproduzi-la. Finalmente, a última opção permite ao utilizador iniciar uma sequência de músicas, iniciando na música com nome mais similar com o nome introduzindo e percorrendo toda a lista de músicas similares. Se não houver músicas similares restantes, é reproduzida uma aleatória e todo o processo é repetido.

Este relatório irá explicar a aplicação começando pelas principais funções usadas, algumas funções auxiliares criadas, os testes efetuados com os módulos do programa, e finalmente as respetivas conclusões.

[1] Estrutura de dados probabilística usada para testar se um elemento pertence a um grupo.
[2] Técnica probabilística para estimar quão similares são dois conjuntos.

O ficheiro entregue contém este relatório e todo o código utilizado dividido em quatro pastas:

Bloom Filter
Contém todos os exercícios do guião 6, bem como a classe *Bloom Filter* e todos os testes feitos relacionados com a classe e seus métodos.

Hash Functions
Contém duas funções de *hash* usadas em partes dos códigos.

Exercicios Jaccard e MinHash(Guião 7)
Todos os exercícios do guião 7 sobre *Jaccard* e *MinHash* e respetivos testes.

Músicas
Todos os ficheiros principais, funções, *scripts* e testes.

Como nem todos os *scripts*, funções e testes efetuados são relevantes, apenas irão ser explicados neste relatório os mais importantes para o projeto em causa.

NOTA: Para funcionar corretamente a maior parte dos *Scripts* precisa de módulos que estão noutras pastas no seu diretório. Os ficheiros apenas estão divididos por pastas para melhor entendimento do leitor do relatório. É aconselhável para testar os módulos e a aplicação principal que se extraia tudo para uma única pasta.

NOTA 2: A aplicação está desenhada para sistemas operativos *Windows*. Para funcionar em *UNIX*, recomendamos que altere em todos os ficheiros que trabalham com diretórios o caracter "\"para "/".

Funções Principais

.1 *Musicas.m*

A função *Musicas.m* é o *script* principal da nossa aplicação. É aqui que o utilizador irá controlar tudo o que se passa na aplicação, sem precisar de abrir mais nenhum ficheiro.

Começa por receber do método *readMp3Files.m* (sub-secção .2) todas as músicas presentes no diretório e sub-diretórios do indicado. Posteriormente, calcula a *MinHash* (sub-secção .4) das músicas, guardando também os parâmetros necessários para comparar os resultados posteriormente. Ainda antes de apresentar o menu ao utilizador, calcula também uma matriz com as distância de *Jaccard*[3] (sub-secção .2). Não seria preciso calcular no início do programa, pois apenas seria preciso quando apresentasse músicas com nome similar, no entanto, para depois poupar

[3]Medida de similaridade entre dois conjuntos

tempo, e para não estar a calcular a matriz sempre que precisamos de encontrar músicas similares, optámos por invocar a função inicialmente.

O menu apresenta ao utilizador cinco opções.

Tocar música por índice

Nesta opção, o utilizador tem a oportunidade de reproduzir uma canção à sua escolha, segundo o índice introduzido (que pode ser procurado na opção seguinte). Também pode escolher o tempo que deseja ouvir a música, ou inserir uma letra para ouvir a música toda (isto deve-se à impossibilidade do *MatLab* de sair de uma música a meio sem travar a execução do programa). Ainda antes de reproduzir o áudio, o programa apresenta as músicas com nome similar existentes na base de dados.

Pesquisar índice de música por nome

Aqui, o utilizador pode pesquisar por nome uma música que deseja ouvir. Caso existam na base de dados músicas com o nome parecido, a aplicação irá apresenta-las. Caso contrário, irá apresentar uma mensagem afirmando que não encontrou músicas com nome similares.

Limpar base de dados inteira e adicionar todas as músicas novamente

Na terceira opção, é dada a opção de limpar a base de dados e adicionar novas músicas segundo um novo diretório. Caso o endereço recebido seja válido, a aplicação irá novamente procurar todos os ficheiro com formato *mp3*, calcular a matriz de *MinHash* (sub-secção .4) e a distância de *Jaccard*(sub-secção .2).

Tocar música aleatória

Na última opção, o programa recebe o tempo em segundos que o utilizador deseja ouvir uma música e reproduz uma música aleatória existente na base de dados. Também apresenta as músicas com nome similar, caso existam.

Iniciar rádio com base em nome recebido

Inicia a reprodução de músicas baseado numa expressão introduzida pelo utilizador. A aplicação procura músicas com nome similar e, se as encontrar, reprodu-las. Se não as encontrar, ou se já as tiver reproduzido todas, toca uma música aleatória e repete o mesmo processo.

.2 *readMp3Files.m*

Esta função receberá um diretório e, caso seja válido, procurará nesse diretório todas as músicas com formato *mp3*. Isto é usado com recurso à função *rdir* (criada por Gus Brown). Seguidamente, cria um *Bloom Filter* (sub-secção .3) para armazenar todas as músicas.

Para cada música, se ela não se encontrar já no *Bloom Filter*, será inserida na base de dados e no *Bloom Filter*, que serão devolvidos pela função no final da execução.

.3 *BloomFilter.m*

A classe *BloomFilter* foi criada para ajudar na manipulação de funções associadas ao *Bloom Filter*. A classe é derivada de *handle*, para suportar a passagem de argumentos por referência (útil para o método *insert*).

O construtor recebe como argumentos o tamanho do filtro e o número de elementos esperados. Com base nestes argumentos, calcula o número ideal de *hash functions* usadas, com recurso ao método *optimalK*. Caso o número de argumentos seja menor do que dois, a classe estima o número de elementos esperados e o seu tamanho. A fórmula utilizada é a seguinte:

$$nhash = m/n * ln(2) \tag{1}$$

onde n é o número de elementos esperados inseridos e o m é o tamanho do *Bloom Filter*[4]

A classe contém o método privado *string2hash*, criado por *D.Kroon*, para calcular o *hash* de uma *string*. Os métodos públicos da classe são dois. O primeiro, *isMember*, verifica se um certo objeto pertence ao *Bloom Filter*. O método *insert* insere o objeto no *Bloom Filter*.

.4 *musicMinHash.m*

Este método calcula a tabela de *MinHash*. Caso a base de dados seja grande, pode ser significativamente lento, pelo que ao longo desta secção vamos indicar as decisões que tivemos de tomar com base na otimização de código e precisão.

No início, seria preciso calcular os *shingles*[5] de cada música. Há várias opções para o fazer. Uma das opções seria dividir as músicas por palavras, e dividir cada palavra numa sequência de caracteres com o mesmo tamanho. Para esta aplicação, isso causaria resultados nada precisos (dado que cada música apenas contém no seu título duas ou três palavras, geralmente) e demoraria bastante

[4]Slides da aula teórica 22 - *Bloom Filters(Continuation)*

[5]Neste caso, *shingles* são as palavras de cada música usadas pelo algoritmo de *MinHash*

tempo. Portanto, descartámos essa opção e optámos por calcular os *shingles* apenas dividindo os títulos por palavras ou alguns caracteres especiais.

Sendo assim, para cada música, o seu título é inicialmente dividido em *shingles*. Depois, cada *shingle* válido é adicionado a uma *cell* de *shingles*, e a sua posição noutra *cell*, que guarda as posições dos *shingles* de cada música. Como o cálculo é feito usando *cell*'s, o cálculo vetorial fica mais lento do que o normal. Logo, comentámos o próximo passo no código do programa, que seria procurar nas *cell*'s *shingles* iguais, para evitar duplicações. Apesar de ser mais preciso e de poupar memória, é um processo bastante lento para uma base de dados significativa, por isso optámos por usar *shingles* repetidos.

No próximo passo, calculamos mil funções de *hash* para cada *shingle*. Segundo os nossos testes (capítulo "Testes", sub-secção .2), o número de *hash functions* não influenciaria largamente o tempo de execução do programa. Portanto, escolhemos um número de hash functions que nos permite afirmar com segurança que os resultados serão precisos.

Começamos por definir um número primo suficientemente grande (10000019) para o cálculo do *hash*. Começámos por procurar números primos de *Mersenne*[6], no entanto eram maiores ou demasiado pequenos para o nosso objetivo, portanto fixámos o número primo mais perto de 10^7, um valor que considerámos razoável. Depois, geramos dois vetores (a e c) como parâmetros para usar nos *hash codes*. É preciso guardar estes valores, pois serão devolvidos para outras funções que gerem *MinHash* com base no nome de uma música. Para cada *shingle*, a matriz de *hash* é calculada da seguinte maneira:

$$hash[j,i] = mod((a[i] * shingle[j] + c[i]), p) \tag{2}$$

onde: p=10000019; i=número de hash function; j=número do shingle.

Finalmente, para o cálculo da *MinHash*, para cada música na base de dados, calcula o mínimo de todas as *hash functions* para cada *shingle* que contém, como demonstra a equação seguinte:

$$minHash[j,i] = min(hash[index,i], minHash[j,i]) \tag{3}$$

onde: i=número de hash function; j=número da música; index=índice dos *shingles* presentes em cada música.

No final, é devolvida a matriz de *MinHash*, bem como os vetores gerados para as *hash functions* (a e c). Nesta função conseguimos eliminar dois *for's* para percorrer as mil *hash functions* (na geração de *hash codes* para cada *shingle* e na geração da tabela de *MinHash*), calculando assim com vetores, em vez de elemento

[6]Números primos de *Mersenne* são números primos que também são números de *Mersenne*($Mn = 2^n - 1$). São bastante usados pelas suas propriedades para calcular *hash functions* em informática.

a elemento, o que poupa bastante tempo (ver capítulo "Testes", sub-secção .2). Ainda assim, pode ser uma função bastante demorada para uma base de dados significativa.

Funções auxiliares

.1 *musicSingleMinHash.m*

Este método calcula a *minHash* referente a apenas um nome de uma música inserido, para depois poder comparar com a matriz existente na base de dados. Efetua o mesmo processo que o método *musicMinHash.m*(sub-secção .4), mas mais simplificado.

Inicialmente, divide o nome recebido em *shingles*, como descrito acima. Depois, calcula as *hash functions* de acordo com os parâmetros recebidos criados na função *musicMinHash.m*. Finalmente, cria a coluna de *MinHash* de maneira similar à função *musicMinHash.m* (para mais informações, ler sub-secção .4). Como este método é relativamente rápido, não foi totalmente otimizado, optando-se por valorizar a legibilidade do código.

.2 *minHashDist.m*

Este método recebe a tabela de *MinHash* e calcula a distância de *Jaccard*.

Para cada coluna, calcula a similaridade entre todas as outras colunas e ela própria, calculando o número de *minHash*'s iguais a dividir pelo número total de *MinHash*'s (neste caso, mil). Irá retornar uma tabela com a similaridade entre cada linha coluna.

.3 *minHashSingleDist.m*

O método *minHashSingleDist.m* faz o mesmo que o método *minHashDist.m* (ver sub-secção .2), com a diferença de que apenas compara a coluna criada pelo *musicSingleMinHash.m* (ver sub-secção .1) com todas as outras do *MinHash* e uma coluna demonstrando a similaridade entre o nome introduzido e todas as outras músicas presentes na base de dados.

.4 *searchMusics.m*

Este método cria a matriz de similaridade entre a *MinHash* original e a *MinHash* do nome introduzido. Com um *threshold*[7] de 0,2, procura as músicas que têm índice de similaridade superior ao *threshold* e adiciona os índices e a respetiva distância de *Jaccard* a uma matriz, por ordem de similaridade, que irá devolver.

[7] *Threshold* é a margem com mínima usada para determinar se duas músicas são consideradas similares.

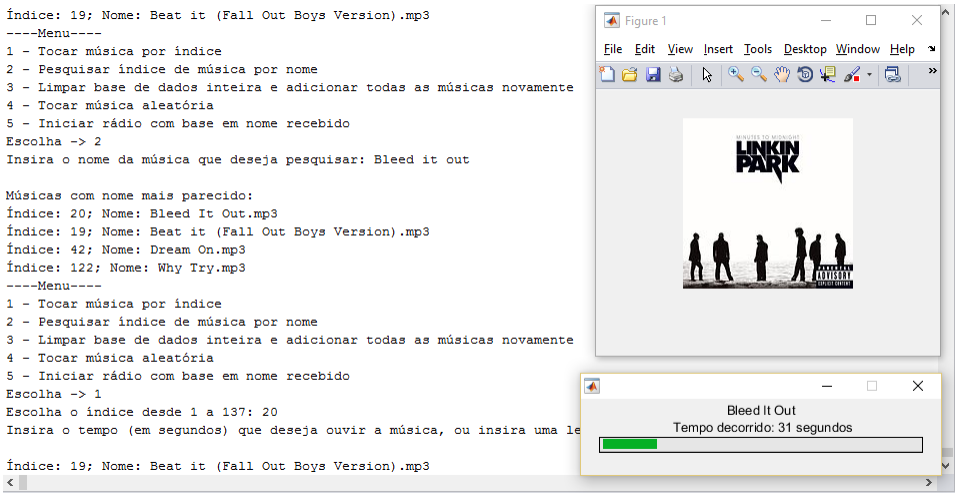

Figura 1: Exemplo da aplicação em funcionamento. Reprodução de uma música.

.5 *SimilarMusics.m*

O método *SimilarMusics.m* é bastante similar ao *searchMusics.m* (ver subsecção .4), no entanto em vez de calcular a distância de *Jaccard* apenas entre uma coluna correspondente ao nome inserido e à matriz de *MinHash*, calcula todas as distâncias de *Jaccard* entre todas as colunas e devolve uma matriz ordenada por distância com as distâncias e os índices similares maiores do que o *threshold*.

.6 *PlayMusics.m*

Esta função recebe um diretório e um tempo em segundos e irá reproduzir o áudio presente no diretório. Se o tempo for inválido, é assumido que a música irá tocar do início ao fim. A reprodução da música é representada por uma *waitbar*, e é aberta uma figura com a imagem associada à música (como pode ser visto na figura 1).

.7 *SearchPhoto.m*

Este método retorna uma imagem associada a um diretório recebido de uma música. Se a pasta onde a música existe contiver apenas uma imagem, irá devolver essa imagem. Caso contenha várias, será devolvida a imagem que tiver o mesmo nome que a música recebida. Caso não exista, ou não existam imagens na pasta, será enviada uma imagem padrão indicando a falta da imagem associada à música.

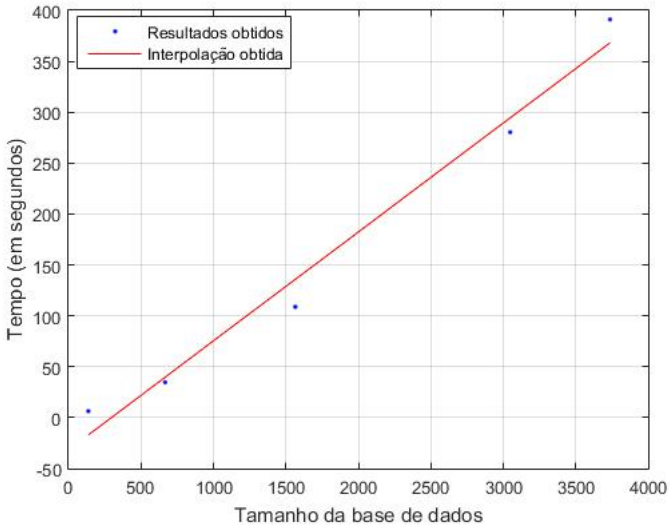

Figura 2: Gráfico que relaciona o tamanho da base de dados obtida com o tempo necessário para a leitura e cálculo de todos os valores. Podemos concluir que se relacionam de forma linear.

.8 *pathToName.m*

Este simples método apenas recebe um diretório e devolve o nome do ficheiro, eliminando assim da sua descrição os diretórios anteriores. Será bastante útil para algumas funções que requerem apenas o nome do ficheiro e não o seu diretório completo.

Testes

.1 *MusicTimeTest.m*

Este primeiro teste percorre todo o processo de inicialização da aplicação com bases de dados de diferentes tamanhos, para analisar o tempo de execução da aplicação aquando da sua inicialização (Figura 2). A conclusão obtida mostra que o tamanho da base de dados lida se relaciona com o tempo de inicialização de maneira linear. É um resultado feliz e que demonstra o nosso sucesso na preocupação em otimização de código. Infelizmente, inviabilizaria bases de dados com número de músicas superiores dez mil (aproximadamente cem mil segundos). Portanto, concluímos que esta solução não é completamente escalável, no entanto, dadas as ferramentas usadas e a linguagem de programação, consideramos os resultados bastante satisfatórios.

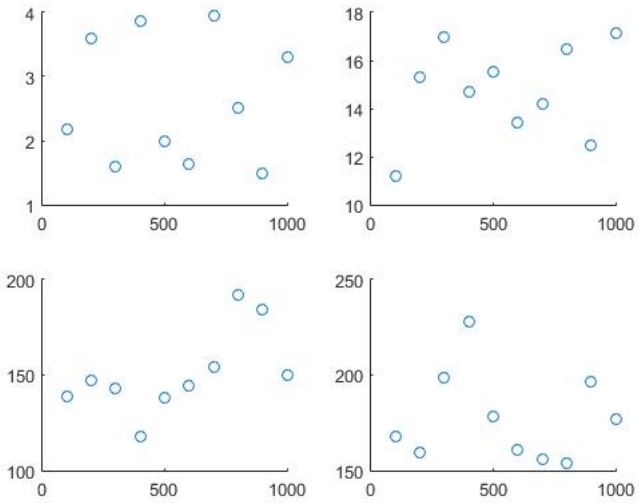

Figura 3: Quatro gráficos que relacionam o número de *hash functions* (eixo x) e o tempo em segundos demorado para calcular a *minHash* (eixo y). Surpreendentemente, não parece haver qualquer relação entre o tempo de execução e o número de *hash functions*. Os quatro gráficos referem-se a bases de dados com 137, 667, 3241 e 4032 músicas, respetivamente.

.2 *MusicHashTimeTest.m*

Este *script* testa os diferentes tempos de execução para diferentes *hash functions*, associadas a diferentes tamanhos de bases de dados. Surpreendentemente, o número de *hash functions* não parece fazer variar em larga escala o tempo de cálculo da *MinHash* (Figura 3). Podemos assim concluir que as otimizações efetuadas nos módulos foram bem-sucedidas. Sendo assim, decidimos fixar o número de *hash functions* em mil, que nos parece um valor com um erro bastante pequeno.

.3 *TesteMusicasBloomFilter.m*

Este *script* inicializa o *Bloom Filter* com todas as músicas existentes em três diretórios e calcula para cada um a probabilidade de falsos positivos experimental, comparando com a probabilidade teórica. Também mostra um gráfico comparando as duas probabilidades. Como pudemos concluir, a probabilidade experimental situa-se muito perto da probabilidade teórica, e também muito perto de 0.

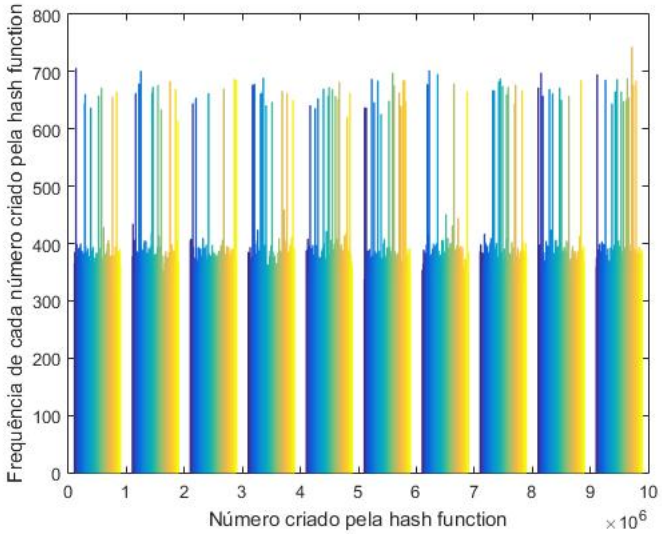

Figura 4: Gráfico que mostra o número de vezes que a *hash function* gera um certo número, desde 0 a 10^7. Podemos ver que, apesar de algumas anomalias, podemos considerar que é uma *hash function* satisfatória para o nosso objetivo.

.4 *TesteMusicasHashFunction.m*

Esta função testa a uniformidade da *hash function* usada. Começa por listar todas as músicas existentes num diretório. Depois, aplica cem *hash functions* diferentes a cada música e mostra um gráfico apresentando a primeira linha e coluna de cada gráfico distribuída pelo seu valor (Figura 4).

.5 *MusicMinHashTest.m*

Este teste imprime as músicas mais similares existentes num diretório, para testar as funções *readMp3Files, musicMinHash e minHashDist.*

.6 *MusicSingleMinHashTest.m*

Este *script* percorre os mesmos testes que o anterior, mas em vez de comparar as músicas mais similares entre si, compara-as com um nome de uma música inserida.

.7 *SmallCheckBloomFilter.m*

Este módulo faz um pequeno teste às funções *insert* e *isMember* do *Bloom Filter*, inserindo algumas *strings* e verificando a sua pertença e de outros elementos

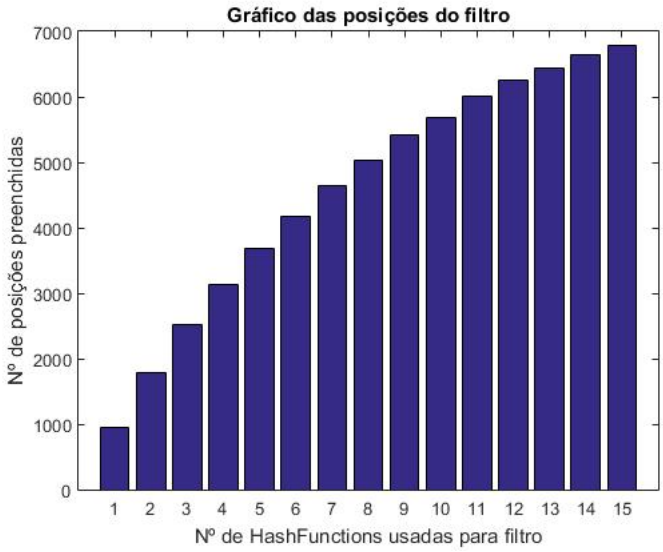

Figura 5: Gráfico que mostra o número de posições do *Bloom Filter* preenchidas pelo número de *hash functions* usadas. Como seria de esperar, o gráfico aumenta linearmente.

que não pertencem. No final, também calcula as colisões ocorridas.

.8 *BigCheckBloomFilterv1.m*

O *script* faz um teste exaustivo às funções do *Bloom Filter*. Começa por inserir 1000 *strings* aleatórias e calcular as colisões. Depois, gera 10000 *strings* aleatórias e verifica se pertencem ao *Bloom Filter*. Finalmente, desenha o gráfico das posições do *Bloom Filter* para verificar se se encontra uniforme.

.9 *BigCheckBloomFilterv2.m*

Neste teste, é feito o segundo teste exaustivo antes de se criar uma classe com as funções do *Bloom Filter*. Começa por se definir um número máximo de *hash functions*. Depois, desde um até esse número (neste caso, 15), são criados *Bloom Filters* que respeitem esse número de *hash functions* e são inseridas *1000* strings aleatórias. No final desta primeira parte, mostra um gráfico do número de *hash functions* usada pelo filtro e o número de posições preenchidas por cada filtro (Figura 5).

Para a segunda parte, testa a inserção de 10000 *strings* geradas aleatoriamente, e grava também a probabilidade teórica de falsos positivos.

Finalmente, mostra um gráfico que contém a probabilidade de falsos positivos

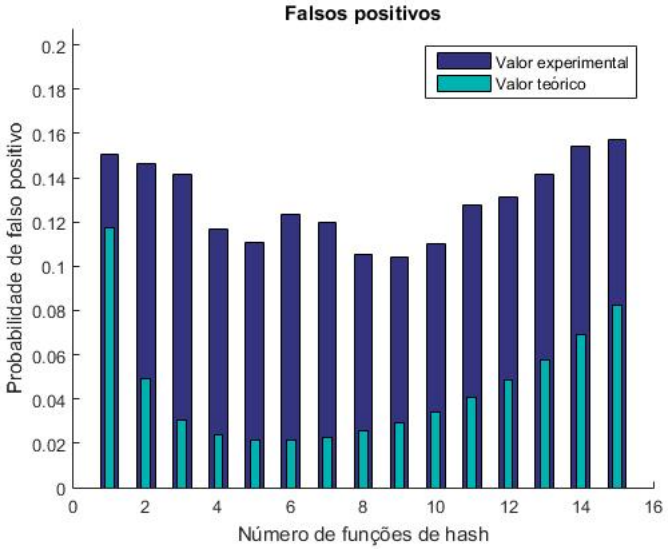

Figura 6: Gráfico que mostra a probabilidade de falso positivo teórica e experimental, variando consoante o número de *hash functions*. Tal como esperado, o valor ideal situa-se entre o 5 e o 6. O valor teórico é bastante mais baixo devido a estarmos a assumir que todas as *strings* criadas não pertencem ao filtro, o que não é verdadeiro.

teórica e a experimental, assumindo que todas as *strings* geradas não pertencem ao filtro (o que não corresponde à verdade, logo a probabilidade experimental irá ser mais elevada do que a teórica, Figura 6).

Este teste não usa diretamente o *Bloom Filter* usado na aplicação. Os parâmetros e as funções de *hash* são diferentes. Serve apenas para comprovar o método *OptimalK* na prática.

.10 *FinalTestBloomFilter.m*

Com todas as funções testadas, podemos agora passar ao teste final da classe *Bloom Filter*, que cria um filtro e testa a inserção de 10000 *strings*.

Conclusão

O objetivo deste trabalho era exemplificar o uso de um *Bloom Filter* e de *MinHash* adequadamente com um exemplo prático. Pensamos que atingimos esse objetivo, no entanto, temos algumas salvaguardas.

A linguagem de programação usada não foi a mais adequada para a resolução deste projeto, devido principalmente à sua morosidade quando trata ciclos longos. Esta dificuldade obrigou-nos a tornar o nosso trabalho mais impreciso para garantir a sua fiabilidade em tempo útil. Para bases de dados maiores, a aplicação será bastante demorada e imprecisa, sendo preciso ajustar o valor de *threshold* para obter resultados válidos. A falta de tempo para realizar o projeto também foi um grande problema, pois podíamos ter realizado testes mais exaustivos e importantes, que infelizmente não foram possíveis. Também não foi possível implementar a *Locality Sensitive Hashing*[8], algo que otimizaria o cálculo da distância de *Jaccard* no início da aplicação.

[8]Técnica probabilística que diminui significativamente o tempo de procura de *shingles* similares, dividindo as colunas da *MinHash* em *buckets*.