

# Sistema GIT

UA

João Gameiro, Marco Ramos



VERSAO

# Sistema GIT

DETI - MPEI

UA

João Gameiro, Marco Ramos  
(93097) joao.gameiro@ua.pt, (93388) msramos99@ua.pt

5-11-2019

# Capítulo 1

## Introdução

O principal objectivo deste relatório é ajudar na percepção do código desenvolvido, explicar o funcionamento da aplicação concebida e outras eventuais questões relativas ao projeto desenvolvido no âmbito da unidade curricular de Métodos Probabilísticos para Engenharia Informática (MPEI), do curso Engenharia de Computadores e Telemática (ECT).

A aplicação constitui uma espécie de simulação do que poderia ser um Sistema Git para armazenamento de ficheiros. As funcionalidades da mesma incluem:

- Adição de ficheiros;
- Remoção de ficheiros;
- Pesquisa e listagem de ficheiros;
- Procura de similaridade entre ficheiros.

A aplicação foi desenvolvida usando a linguagem de programação *Java* e é composta por vários componentes, que serão explicitados nos capítulos à frente.

Este documento está dividido em quatro capítulos. Depois desta introdução, no Capítulo 2 são apresentados todos os constituintes da aplicação, no Capítulo 3 são expostos e analisados os testes realizados enquanto que o Capítulo 4 refere-se ao funcionamento da aplicação. Finalmente, no Capítulo 5 é apresentada uma visão geral do trabalho acompanhada de uma conclusão sobre o mesmo.

## Capítulo 2

# Aplicação e os seus Componentes

### 2.1 Módulos

#### 2.1.1 BloomFilter

A classe BloomFilter tem como atributos: um array de inteiros, um double que representa a probabilidade de falsos positivos, e três inteiros que representam respectivamente, o número de posições do BloomFilter, número de elementos do conjunto de entrada e o nº de Hash Functions necessárias. Tem também como atributo um array de arrays de inteiros que serve maioritariamente para auxiliar no processo de hashCode dos nomes dos ficheiros (Figura 2.1).

O construtor da classe é inicializado com um conjunto (neste caso uma LinkedList que contém ficheiro) e a partir do mesmo irá calcular o número de posições do BloomFilter, o número de Hash Functions necessárias e inicializar o array de inteiros com todas as posições a 0 excepto aquelas que foram mapeadas pelas HashFunctions (inicializadas com 1). Inicia também o array bidimensional com os valores necessários para o cálculo do hashCode.

O BloomFilter possui 3 métodos (Figura 2.1). O método *membro(File fic)* recebe como argumento um ficheiro e verifica utilizando o nome do ficheiro se este pertence ou não ao BloomFilter devolvendo true ou false de acordo com a situação em questão.

O método *adicionar(File fic)*, ao verificar se um ficheiro é membro ou não, adiciona-o ao BloomFilter com auxilio das HashFunctions que mapeiam a sua posição e a alteram para 1.

O método *hashFunct(String key, int idx)* serve para mapear o nome de um ficheiro (String key) para uma posição do BloomFilter. Recebe como argumento para além do nome do ficheiro, o número de uma das k hash functions que estão disponíveis. É realizado um *hashCode()* ao nome do ficheiro e posteriormente verificado se o valor obtido se insere dentro dos limites do array do BloomFilter,

para que a posição obtida seja posta a 1.

```
private int hashFunc(String key, int idx) {
    int h = 0;

    for(int i=0; i<key.length(); i++)
        h = 37 * h + (int)(key.charAt(i));

    for(int i=0; i<key.length(); i++)
        h += key.charAt(i)*conjAltr[idx][i%50];

    h = h%n;
    if(h < 0)
        h += n;
    return h;
}

//adicionar novo elemento ao array B
public void adicionar(File elem) {
    assert !membro(elem);

    for(int i=0; i<k; i++) {
        B[hashFunc(elem.getName(), i)] = 1;
    }
}

//testar se o elemento existe
public boolean membro(File elem) {
    for(int i=0; i<k; i++) {
        if(B[hashFunc(elem.getName(), i)] == 0)
            return false;
    }
    return true;
}
```

Figura 2.1: Métodos da classe BloomFilter

### 2.1.2 MinHash

A classe MinHash foi desenvolvida com o intuito de verificar a similaridade entre ficheiros. Tem como atributos uma LinkedList de Strings que serve para guardar os Shingles, um array de Strings para as assinaturas do conjunto, um ficheiro, (o que será futuramente introduzido), dois inteiros (dimensão dos shingles e número de hashFunctions necessárias) e um array bidimensional de inteiros aleatórios que são associados a cada hashFunction.

O construtor da classe, inicializa LinkedList de Shingles e o array com assinaturas do conjunto. É lido o conteúdo do ficheiro que posteriormente vai ser separado em Shingles e comparado. O tamanho dos Shingles varia conforme o tamanho do ficheiro, 5 se este tiver menos de 1000 caracteres e 10 se tiver mais de 1000.

O método *hashShingle()* retorna o resto da divisão do somatório de todos os Chars (representados pelo seu respectivo código ASCII) de um Shingle, multiplicados por um número aleatório gerado na classe ArrayGenerator. Representa o processo de hash a um Shingle.

*minHashArray()* passa todos os Shingles por uma função de hash (corresponde a uma posição do array), e o menor valor fica guardado nessa posição. Assim sendo no fim deste processo vamos ter o array de assinaturas preenchido e pronto para comparar com outro com o intuito de obter a similaridade.

*jaccard()* compara os Shingles presentes em dois arrays de assinaturas de dois conjuntos (dois ficheiros) e devolve o resultado da divisão entre o número de vezes em que os Shingles eram iguais nas mesmas posições do array e o tamanho do array, ou seja calcular a similaridade de jaccard.

```
private void minHashArray(LinkedList<String> tmp) {
    for(int i=0; i<k; i++) {
        try {
            String min = tmp.get(0);

            for(int j = 1; j < tmp.size(); j++ ) {
                if(hashShingle(tmp.get(j), i) < hashShingle(min, i))
                    min = tmp.get(j);
            }
            minHash[i] = min;
        } catch(IndexOutOfBoundsException e) {
            System.out.println("ERRO: Ficheiro introduzido demasiado pequeno.");
            System.exit(1);
        }
    }
}

//similaridade de jaccard
public double jaccard(MinHash c2) throws IOException {
    if(c2.getDimSgl() != dimSgl) {
        if(c2.getDimSgl() < dimSgl) {
            return new MinHash(f, 5).jaccard(c2);}
        else {
            c2 = new MinHash(c2.getFile(), 5);}
    }

    double inter = 0;
    for(int i=0; i<k; i++) {
        if(minHash[i].equals(c2.getElem(i)))
            inter++;
    }
    return inter/k;
}
```

Figura 2.2: Métodos minHashArray e jaccard da classe MinHash

## 2.2 Aplicação

### 2.2.1 GitSystem

A classe GitSystem tem como atributos uma LinkedList e um BloomFilter e vai servir no geral para gerir os ficheiros introduzidos ou removidos no sistema. O seu uso geral verifica-se na ligação com a janela principal em que cada método é chamado ao premir de botões.

O seu construtor inicia a `LinkedList` de ficheiros como vazia. Os seus métodos realizam operações tanto sobre a `LinkedList` como sobre o `BloomFilter`.

### **addFile**

Método que devolve um inteiro de acordo com o resultado da operação realizada e recebe como argumento um ficheiro. A função verifica se o ficheiro já pertence ou não ao `BloomFilter` e verifica também a similaridade entre os ficheiros para sinalizar na janela principal se for esse o caso.

Ao pressionar este botão vai ser aberta uma nova `JFrame` com uma pequena caixa de texto e um botão que permitem enviar um ficheiro ao sistema. O sistema irá sinalizar se o ficheiro não existe, ou se foi adicionado com sucesso, ou se já tinha sido adicionado anteriormente (pertença ao `BloomFilter`), ou se existem outros ficheiros com conteúdo similar (através da `MinHash`).

A função devolve:

- 0 - Caso de sucesso;
- 1 - O ficheiro não existe;
- 2 - O ficheiro pertence ao `BloomFilter`;
- 3 - Existem ficheiros similares no sistema.

```
//-----//
//Adicionar ficheiros
//-----//
public int addFile(String filename) throws IOException
{
    File f = new File("./src/Projeto/Files/"+filename);

    if(f.exists()) {
        if(list.isEmpty()) { //para o primeiro ficheiro a ser adicionado
            list.add(f);
            bf = new BloomFilter(list);
            return 0; //sucesso
        }
        else {
            if(bf.membro(f))
                return 2; //ficheiro já foi adicionado
            else {
                list.add(f);
                bf = new BloomFilter(list);
                if(similar(filename))
                    return 3;
                return 0;
            }
        }
    }
    else
        return 1; //ficheiro não existe
}
```

Figura 2.3: Função `addFiles`

Sempre que um ficheiro for adicionado com sucesso ao sistema significa que este foi adicionado à `LinkedList` que posteriormente foi usada para criar um novo `BloomFilter` com todos os ficheiros já adicionados.

Para verificar a similaridade o método usa uma função auxiliar (`similar()`) que devolve `true` se um ficheiro tiver uma percentagem de similaridade maior ou igual a que 50%.

## removeFile e searchFile

Ambas estas funções recebem como argumento um nome de um ficheiro e ambas vão verificar se o mesmo existe no BloomFilter. Se ele existir a removeFile vai removê-lo tanto do BloomFilter como da LinkedList que contem os ficheiros e devolver true, já a função searchFile irá devolver uma String composta pelo conteúdo do ficheiro para posterior apresentação ao utilizador.

## searchSimilarFiles

Este método recebe como argumento um ficheiro e vai devolver uma String com a percentagem de similaridade entre o mesmo e todos os ficheiros adicionados ao sistema. Para esse efeito usa o método da descrito anteriormente da MinHash intitulado *jaccard()* que devolve a similaridade de jaccard entre dois ficheiros.

```
//-----//
//Pesquisar ficheiros similares //
//-----//
public String searchSimilarFiles(String filename) throws IOException
{
    String result = "Similar Files:\n\n";
    File f = new File("./src/Projeto/Files/"+filename);

    if(bf.membro(f)) {
        MinHash m = new MinHash(f);

        for(File file : list) {
            if(file.compareTo(f)!=0) {
                double sim = m.jaccard(new MinHash(file));
                result = result + file.getName() + " -> " + sim*100+"%\n";
            }
        }
        return result;
    }
    else
        return "";
}
```

Figura 2.4: Função searchSimilarFiles

Caso o ficheiro não exista no sistema a função devolve uma String vazia para auxiliar na posterior sinalização dessa ocorrência.

### 2.2.2 GitWindow

Esta classe serve maioritariamente para implementação da interface gráfica da aplicação que é descrita mais à frente neste mesmo documento. A GitWindow usa também a classe GitSystem para implementar todas as operações do sistema ao que são executadas quando o utilizador selecciona os botões. Através desta classe são processadas todas as acções do sistema desde inserir ficheiros até à sinalização de erros e apresentação de informação ao utilizador. O resultado e descrição da interface encontram-se no Capítulo 4.



## Capítulo 3

# Testes

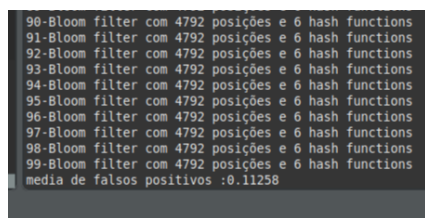
### 3.1 BloomFilterTeste

O teste do BloomFilter baseia-se num processo de procura de falsos positivos. São gerados 1000 ficheiros aleatórios usando a classe *GeradorAleatorioNomes* e desses 1000, 500 são adicionados ao BloomFilter. Posteriormente testa-se se os restantes 500 ficheiros pertencem ou não ao BloomFilter, tendo em conta que estes não foram adicionados.

Se algum destes últimos 500 que não foram adicionados for membro significa que estamos na presença de um falso positivo. Logo sendo assim todos os falsos positivos vão ser contados.

Este processo vai ser repetido 100 vezes e no fim será calculada a média dos falsos positivos.

O valor obtido deverá estar próximo do valor da probabilidade de falsos positivos, no entanto este não é o caso. Foi obtido um valor muito superior ao esperado visto que o valor definido na classe é 0.01.



```
90-Bloom filter com 4792 posições e 6 hash functions
91-Bloom filter com 4792 posições e 6 hash functions
92-Bloom filter com 4792 posições e 6 hash functions
93-Bloom filter com 4792 posições e 6 hash functions
94-Bloom filter com 4792 posições e 6 hash functions
95-Bloom filter com 4792 posições e 6 hash functions
96-Bloom filter com 4792 posições e 6 hash functions
97-Bloom filter com 4792 posições e 6 hash functions
98-Bloom filter com 4792 posições e 6 hash functions
99-Bloom filter com 4792 posições e 6 hash functions
media de falsos positivos :0.11258
```

Figura 3.1: Resultado do Teste ao BloomFilter

Devido à imprecisão deste e de outros testes efectuados(foi procurado um valor que ocupasse a menor memória possível e que apresentasse poucos falsos positivos), foi dificultada a procura do valor da probabilidade de falsos positivos perfeito. O valor apresentado é imperfeito.

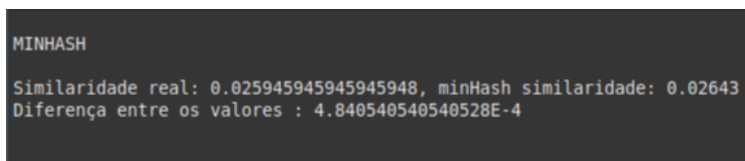
## 3.2 MinHashTeste

O Teste da MinHash consiste no cálculo da similaridade real de dois ficheiros e comparar o valor obtido com o que a classe devolve.

O valor da similaridade real obtém-se através da razão entre o número de shingles iguais entre dois ficheiros o número total de shingles. Logo utilizando dois ficheiros de teste e extraíndo os shingles dos mesmos é-nos permitido calcular a similaridade real.

Ao termos o valor real resta-nos apenas o aplicar o método da classe e comparamos os dois valores. O resultado esperado seria de grande similaridade entre os dois valores.

Ao executarmos o programa vamos obter o resultado especificado na Figura 3.2.

A terminal window with a dark background and light green text. The text displays the results of a MinHash test, showing a high real similarity and a very small difference between the real and minHash similarity values.

```
MINHASH  
Similaridade real: 0.025945945945945948, minHash similaridade: 0.02643  
Diferença entre os valores : 4.840540540540528E-4
```

Figura 3.2: Resultado do Teste à MinHash

Como podemos verificar a diferença entre os valores é muito pequena o que nos leva a concluir que a procura de similaridade entre ficheiros encontra-se operacional.

## 3.3 Aplicação Conjunta

Para o teste da aplicação conjunta encontra-se no directório da aplicação uma pasta intitulada de Files que possui uma série de ficheiros prontos a ser usados para o teste.

Alguns dos ficheiros possuem similaridade entre eles o que no irá permitir observar o correto funcionamento da MinHash. Todos os ficheiros pertencem aos autores (possuem conteúdo de outras unidades curriculares) e servem apenas para testar a aplicação conjunta.

Para testar com ficheiros para além dos que se encontram com a aplicação, seria apenas necessário colocá-los nesse mesmo directório.

O correto funcionamento e descrição da aplicação conjunta pode ser observado no Capítulo 4.

## Capítulo 4

# Funcionamento

### 4.1 Janela Principal

Para observar o funcionamento que irá ser descrito é simplesmente necessário executar o ficheiro *Main.java*.

#### 4.1.1 Interface Gráfica

A interface gráfica (Figura 4.1) desta aplicação assenta numa *JFrame* base que suporta outros componentes através dos quais nos apresenta a informação e possibilita o funcionamento do sistema em causa.

Possui três *JPanels*: um que serve como header para apresentar o título da aplicação, outro para colocação dos botões através dos quais se vai realizar a interacção com a o sistema e finalmente um painel de rodapé.

Esta interface foi desenvolvida e implementada com as ferramentas de *Java Swing* e derivados.

### 4.2 RemoveFiles e SearchFiles

Ambos estes tal como o anterior ao serem pressionados abrem uma nova janela para que o utilizador possa especificar o ficheiro que pretende que a aplicação use para realizar a ação necessária.

Ambos vão sinalizar o inserir nome de ficheiros inválidos, ou outras situações plausíveis de erro.

### 4.3 ListFiles e RemoveAllFiles

Ao serem pressionados, ListFiles imprime o nome de todos os ficheiros adicionados até ao momento ou sinaliza que ainda não foram adicionados se for esse o caso, enquanto que RemoveAllFiles remove todos os ficheiros do sistema e apresenta uma mensagem a indicar esse acto.



Figura 4.1: Janela Principal da Aplicação

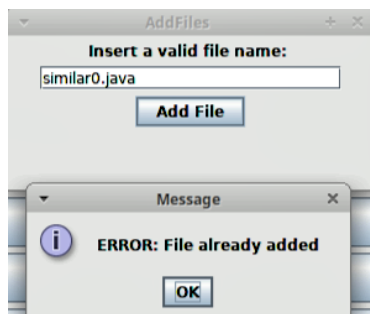


Figura 4.2: Sinalização de ficheiro já adicionado

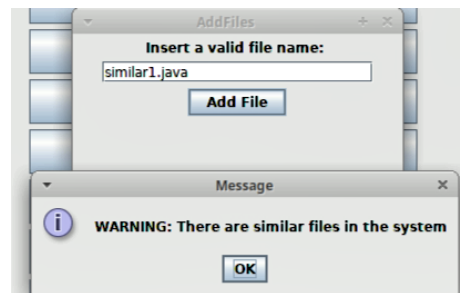


Figura 4.3: Sinalização de ficheiros similares

## 4.4 FindSimilarFiles

Ao ser pressionado este botão apresenta-nos uma nova janela na qual é possível pesquisar um ficheiro. Se este for válido vai-nos ser apresentada uma lista contendo os ficheiros adicionados e a percentagem de similaridade de cada um destes com o ficheiro pesquisado.

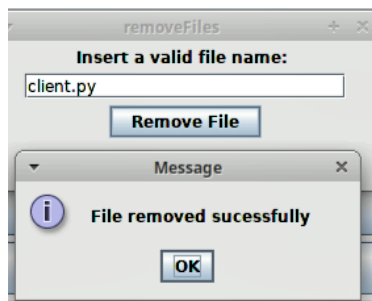


Figura 4.4: Sinalização de ficheiro removido

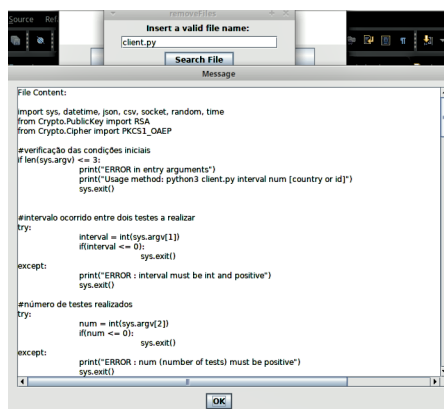


Figura 4.5: Apresentação do conteúdo de um ficheiro (pesquisa)

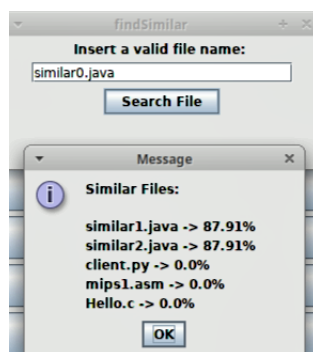


Figura 4.6: Similaridade entre ficheiros

## Capítulo 5

# Conclusões

A base da aplicação funciona, ou seja foi implementado um sistema do tipo Git que permita gerir ficheiros. O sistema sinaliza similaridade entre ficheiros e impede o utilizador de inserir ficheiros com nomes iguais.

Apesar de o módulo do BloomFilter estar a funcionar correctamente na aplicação conjunta, concluímos a partir dos testes efectuados que este apenas opera bem para poucas quantidades de ficheiros, ou seja se fossem inseridas grandes quantias a probabilidade de adição de ficheiros com nomes iguais aumentaria muito mais. Logo constata-se que este módulo não está nas melhores condições.

O módulo da MinHash funciona correctamente como foi demonstrado nos testes. Ao adicionar ficheiros à aplicação também se verificou uma execução correta desta classe que também é visível quando se pesquisa por ficheiros similares.

## Capítulo 6

# Contribuições dos autores

Este projeto foi realizado por João Gameiro (JG) e Marco Ramos (MR). Ambos os autores contribuíram igualmente para o trabalho tanto na discussão do projeto, como no desenvolvimento do mesmo.

# Acrónimos

**JG** João Gameiro

**MR** Marco Ramos

**MPEI** Métodos Probabilísticos para Engenharia Informática

**ECT** Engenharia de Computadores e Telemática