Projeto de MPEI

Trabalho elaborado por:

- Pedro Escaleira, NMec 88821
- Rafael Simões, NMec 88984

1. Resultados dos testes de cada módulo

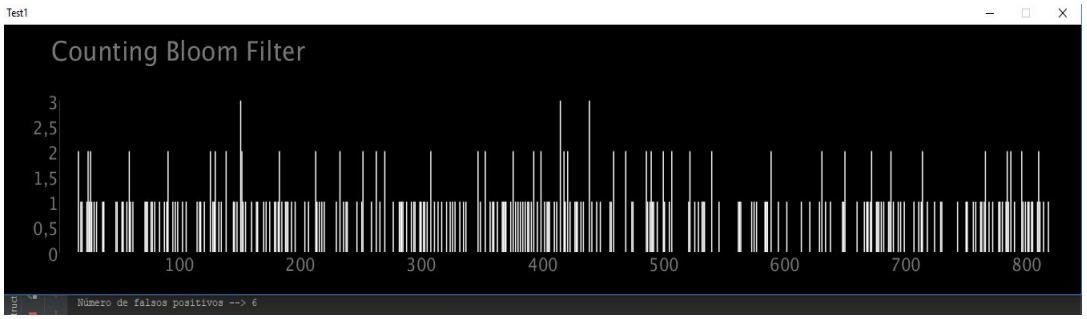
1.1. Testes do módulo contador estocástico

```
Insira a probabilidade (entre 0 e 1) de o contador estocástico ser incrementado: 0.1


Contagem dada pelo contador estocástico: 41920
Contagem real: 41835



Mais informações sobre o contador estocástico:
-> Média: 4183.5
-> Variância: 418.3500000000001
```

No **Test1** deste módulo, decidirmos fazer um teste básico para verificar se o contador estocástico implementado estava a funcionar devidamente. Para isso, este programa pede inicialmente a probabilidade de o contador estocástico ser incrementado cada vez que essa operação é desejada, sendo depois este contador usado para contar o número de palavras presentes num ficheiro. Por fim, o nosso programa apresenta o número de palavras contadas pelo contador estocástico *vs* o número de palavras que o ficheiro realmente tem, sendo que no final apresenta informações adicionais do contador estocástico no final da contagem.

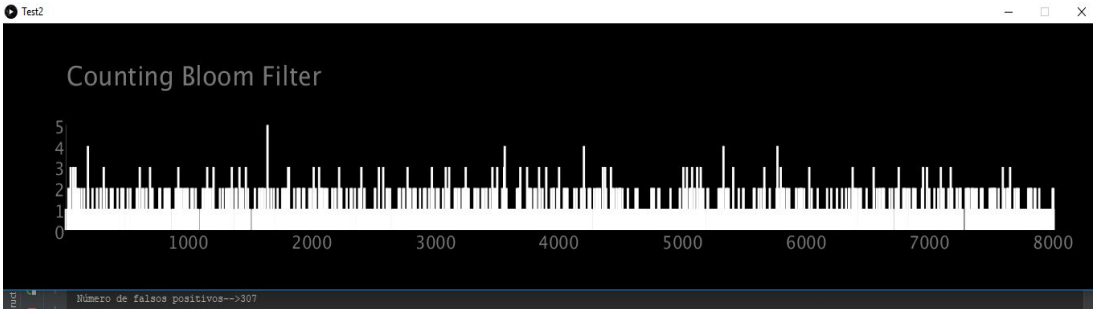
1.2. Testes do módulo Counting Bloom Filter (cbf)

No **Test1** (exercício 2 do guião 6), criamos um array de 205 países (sem paises repetidos), inserimos a primeira metade no cbf e verificamos a existência de cada país da segunda parte do array nesse counting bloom filter, contabilizando o número de falsos positivos e ilustrando o gráfico do cbf. De ter em conta que o tamanho do cbf é obtido através da multiplicação de 8 pela metade tamanho do array de países (já que só metade dos países são inseridos no cbf), tal como tinha sido feito nas aulas práticas.

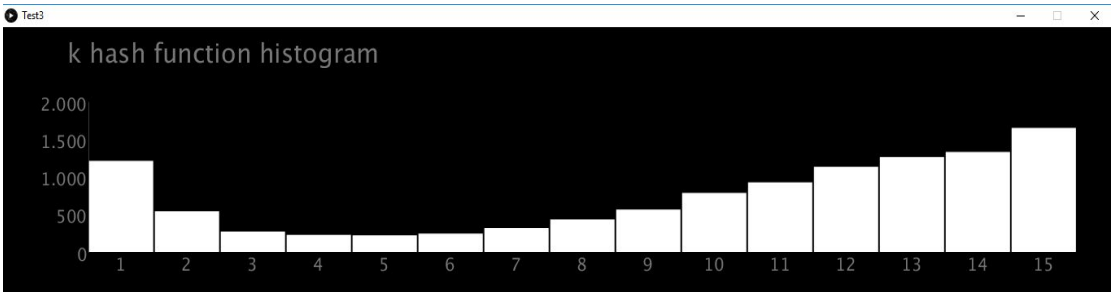


No **Test2** (exercício 3 do guião 6), criamos um cbf de tamanho 8000 e com 3 funções de dispersão, onde são inseridas 1000 palavras geradas aleatoriamente, onde cada letra de cada palavra foi obtida com base na probabilidade de cada letra do alfabeto português (para isso, foram

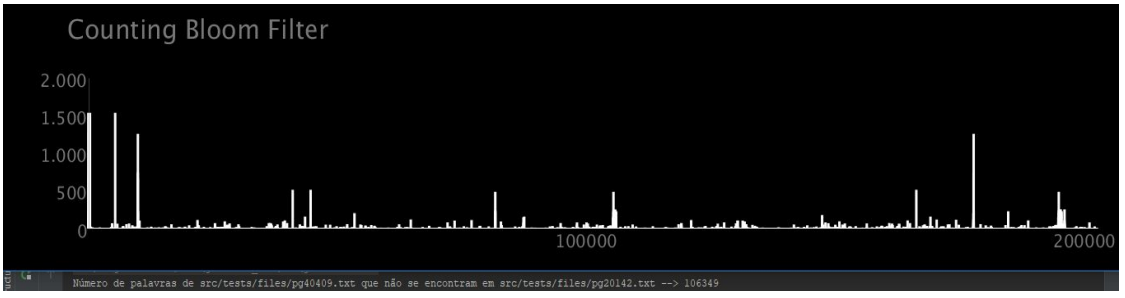
analisados 4 textos portugueses). O número de falsos positivos foi contado através da verificação se 10000 palavras aleatórias pertenciam ao cbf. Por fim, é apresentado também um gráfico do counting bloom filter.



No **Test3** (exercício 4 do guião 6) repetimos o **Test2** com o propósito de analisar qual o número de funções de dispersão a usar no cbf é ideal neste problema. Para isso, foram criados 15 cbf diferentes (cada um com um número de funções de dispersão diferente, de 1 até 15), onde foram inseridas 1000 palavras e contados os números de falsos positivos para 10000 palavras diferentes (todas estas palavras foram originadas aleatoriamente). Desta forma, verificamos que o número ideal de funções de dispersão neste problema é um valor próximo de 4.



No **Test4** (exercício 5 do guião 6), inserimos todas as palavras de um ficheiro no cfb (de tamanho dado pela multiplicação de 8 pelo número de palavras e de 3 funções de dispersão), e verificamos a existência de todas as palavras de outro ficheiro no cbf, contabilizando assim o nr de palavras diferentes entre ambos os ficheiros e ilustrando o respetivo gráfico do cbf. Este teste é propício a alguns erros, porque o número de falsos positivos nunca é 0, pelo que se deve ter em conta que o número de palavras do 2º texto que não se encontram no 1º é apenas uma aproximação.



No **Test5** (exercício 6 do guião 6), inserimos todas as palavras de um ficheiro no cfb (de tamanho dado pela multiplicação de 8 pelo número de palavras e de 3 funções de dispersão). Seguidamente é apresentado um menu com duas opções, como o da seguinte imagem (a terceira opção é apenas a opção que permite terminar o programa):

```
"C:\Program Files\Java\jdk1.8.0_191\bin\java.exe" ...
MENU
1 -> Apresentar as ocorrencias de todas as palavras presentes no ficheiro
2 -> Apresentar a ocorrencia de uma palavra
3 -> Sair
>
```

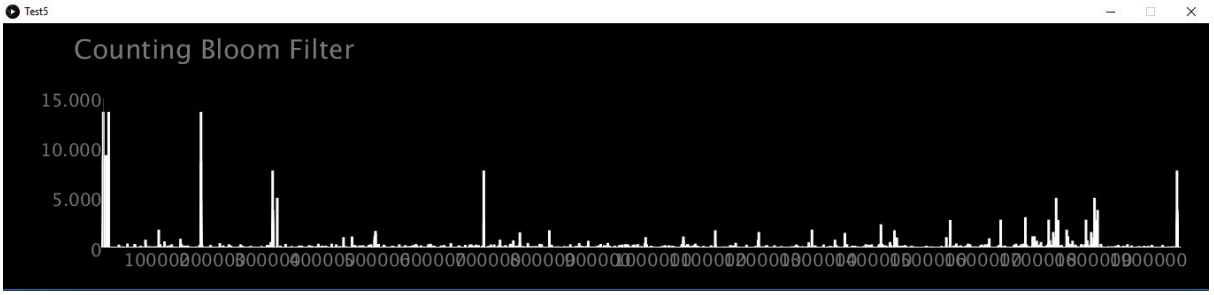
1. A opção 1 mostra o número de ocorrências de cada palavra presente no ficheiro, verificando a existência das mesmas no cbf (desta forma, obtemos uma estimativa do número de palavras presentes no ficheiro).

```
ultimamente-->16
ultimas-->10
ultimo-->20
ultimos-->16
ultrages-->1
ultrajado-->1
ultrajante-->2
ultraje-->2
um-->3777
uma-->2768
umas-->49
```

2. A opção 2 mostra o número de ocorrências de uma palavra pedida ao utilizador verificando a existência da mesma no cbf (e por isso no ficheiro passado).

```
> 2
Insira uma palavra
> uma
uma-->2768
```

Para além disso, é apresentado no final o gráfico do cbf:



1.3. Testes do módulo <u>Processo(s) para descobrir itens similares</u>

No programa **Test1**, são analisados os dados dos utilizadores e filmes obtidos no ficheiro *u.data* (tal como tinha sido realizado no guião 7 das aulas práticas), sendo apresentados três tipos de resultados acerca da similaridade entre os filmes que cada utilizador avaliou:

1. Distâncias de Jaccard

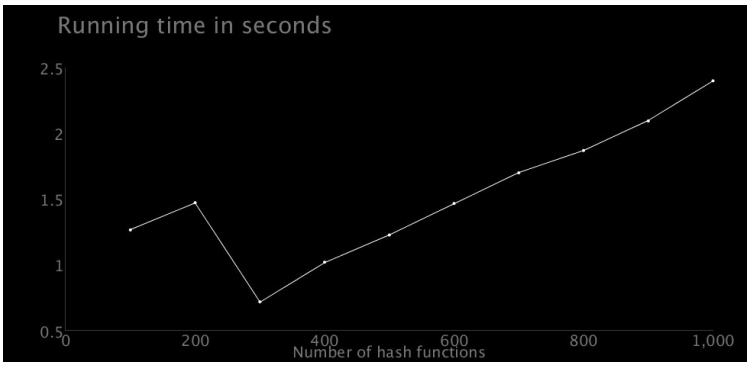
```
            Distância de Jaccard:
> Tempo de execução: 10.511 s
> Resultados:
-> 328 788 0.32704402515723274
-> 408 898 0.16129032258064513
-> 489 587 0.3700787401574803
```

2. Distâncias obtidas através de MinHash, fazendo-se variar o número de hash functions entre 100 e 1000 com intervalo de

```
          Distância obtida por MinHash usando 700 hash functions:
> Tempo de execução: 1.753 s
> Resultados:
-> 328 788 0.29428571428571426
-> 408 898 0.1528571428571428
-> 489 587 0.3828571428571429
```

100 em 100

3. Gráfico dos tempos de execução do algoritmo de MinHash em relação ao número de hash functions usadas.



Para além do mínimo exigido neste módulo, decidimos ainda implementar **Shingles** com MinHash e **Shingles** com **LSH**. Os testes para cada uma destas implementações foram realizados através dos programas (respectivamente):

- **TestShingles**

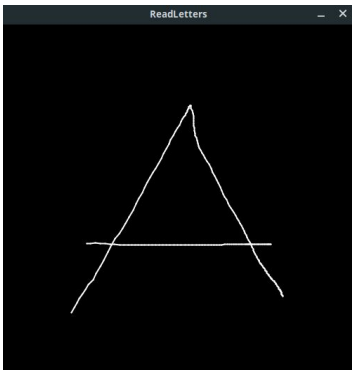
```
          Distância obtida por MinHash usando 200 hash functions:
> Tempo de execução: 2.844 s
> Resultados:
-> src/tests/files/pg20142.txt src/tests/files/pg21209.txt 1.0
-> src/tests/files/pg20142.txt src/tests/files/pg20142.txt 0.0
-> src/tests/files/pg21209.txt src/tests/files/pg20142.txt 1.0
```

- **TestShinglesLSH**

```
           LSH Distance for 200 hash functions:
> Tempo de execução: 3.02 s
> Resultados:
-> src/tests/files/pg20142.txt src/tests/files/pg20142.txt 0.0
```

Para cada um destes programas, são apresentados resultados para um número de hash functions entre 100 e 1000 com intervalo de 100 em 100, tendo sido comparados 3 ficheiros, sendo dois deles o mesmo, para verificar se os resultados correspondiam ao esperado.

2. Resultados da implementação de uso conjunto



Nesta fase do projeto, a nossa ideia inicial foi fazer um programa que lesse uma letra escrita à mão pelo utilizador e apresentasse uma hipótese de qual a letra escrita (imagem à direita).

Para a implementação deste programa, usamos um *dataset* que encontramos na *internet* com todas as letras do alfabeto escritas à mão por várias pessoas. Neste *dataset*, cujo formato é *csv*, a informação dos *pixeis* de cada imagem, de tamanho 28x28, de cada letra é apresentada em cada linha do documento, sendo a primeira coluna de cada linha a identificação numérica da letra dessa linha.

Para o programa comparar a letra desenhada pelo utilizador, foi necessário, numa primeira fase, diminuir o tamanho da letra desenhada pelo mesmo para um formato 28x28. Feito isto, foi essencial pensar numa forma de o programa facilmente comparar cada pixel da letra desenhada pelo

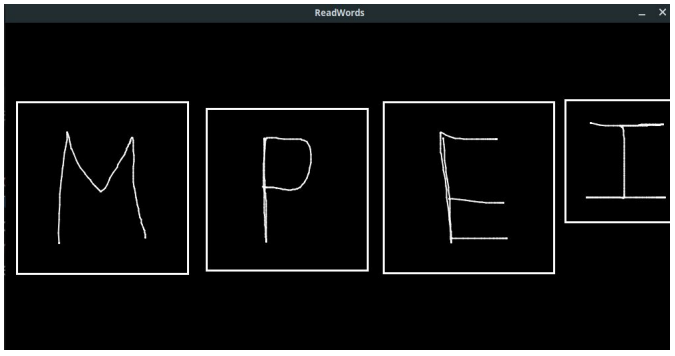
utilizador com cada pixel de cada letra do dataset. Desta forma, chegamos à conclusão a melhor forma de contornar este problema era comparar as imagens através das coordenadas onde, de facto, tinha sido desenhado algo, já que uma comparação *pixel* a *pixel* teria uma complexidade computacional muito mais elevada. Por fim, cada letra do *dataset* foi submetida a uma comparação com a letra desenhada, usando para esse efeito o algoritmo de MinHash já desenvolvido no terceiro módulo.

```
Similaridades (ordenadas das maiores para as menores):
A -> 0.7142857142857143
L -> 0.7142857142857143
H -> 0.6571428571428571
R -> 0.6571428571428571
C -> 0.6285714285714286
X -> 0.6285714285714286
Z -> 0.6285714285714286
N -> 0.6
E -> 0.5714285714285714
O -> 0.5714285714285714
J -> 0.5428571428571428
Q -> 0.5428571428571428
D -> 0.5142857142857142
F -> 0.5142857142857142
G -> 0.5142857142857142
```

Um dos principais problemas que encontramos à medida que implementamos esta solução foi o tempo demasiado longo que o programa necessitava para fazer as comparações, sendo que a solução que encontramos foi a diminuir do número de imagens de cada letra lida para 150.

Tendo terminada esta implementação, pensamos em "ir um pouco mais longe" e, usando o código já feito, fazer uns acrescentos para que o programa conseguisse ler uma palavra inteira. Assim, criamos um novo programa, **ReadWords**, com o propósito de permitir ao utilizador escrever palavras inteiras.

O problema mais notável a ser resolvido nesta nova implementação foi desenvolver uma forma de o programa "saber" onde o utilizador escrevia cada letra. A resposta que pensamos para tal foi o programa desenhar um quadrado à volta de cada letra desenhada, que funciona como janela de cada letra, isto é, é em relação a essa janela que posteriormente cada letra vai ser redimensionada para o tamanho 28x28.



Para além de decifrar palavras, este programa apresenta-se mais robusto que o primeiro, possuindo um menu onde se podem selecionar certos parâmetros que o programa deve ter em conta quer durante a execução, quer na apresentação dos resultados.

```
MENU
1 -> Apresentar apenas a mensagem decifrada
2 -> Apresentar os tempos de leitura do ficheiro e de cálculo das similaridades de cada letra
3 -> Apresentar a similaridade de cada letra em relação á correspondente escrita
4 -> Apresentar média de letras "acertadas"
5 -> Apresentar a "quantidade" de cada letra armazenada no ficheiro usando um counting bloom filter
6 -> Usar um contador estocástico para "controlar" o número de vezes que cada letra já foi considerada
7 -> Sair
>
```

Neste menu, são de ter mais em consideração as opções 5 e 6, que quando selecionadas, para além dos resultados acima descritos, são apresentados também os resultados do uso dos primeiro e segundo módulos.

Quando a opção 5 é selecionada, é colocado cada caracter do *dataset* num cbf à medida que este é lido, de forma a, no final do programa ser executado, serem apresentadas as "quantidades" de cada letra no *dataset* usado.

```
Tamanho do counting bloom filter:
> 300

Número de hash functions a serem
  usadas no counting bloom filter:
> 3




Mensagem decifrada: H



"Quantidade" de cada letra
  armazenada no ficheiro:
A -> 14781
B -> 8977
C -> 23555
D -> 10599
E -> 11493
F -> 1164
```

Já a opção 6, quando selecionada, “indica” ao programa para que a contagem que faz em cada letra quando lê o *dataset* deve ser realizada por um contador estocástico, isto é, ao invés de o programa fazer a contagem das 150 imagens de forma exata (que é a predefinição), fá-lo usando um contador estocástico.

```
Insira a probabilidade (entre 0 e 1) de
  o contador estocástico ser
  incrementado: 0.2
```

3. Considerações finais

- Neste projeto foram usadas as bibliotecas *open source* **Processing** (biblioteca que permite criar aplicações de ambiente gráfico de forma facilitada) e **giCentre Utilities** (biblioteca baseada em **Processing** que permite desenhar gráficos) e o *dataset* encontrado em https://www.kaggle.com/ashishguptajiit/handwritten-az.
- Apesar de nas instruções dadas ser pedido um programa de teste para cada módulo, fomos forçados a dividir os testes de alguns módulos em vários programas não só por uma questão de organização, mas também por uma questão de complexidade no uso da biblioteca que utilizamos para implementar a parte gráfica (gráficos, desenhos, etc.).