Bloom-filter + Locality-Sensitive Hashing

Métodos Probabilísticos para Engenharia Informática

Universidade de Aveiro

Pedro Martins 76551
Ricardo Jesus 76613

17 de Dezembro de 2015

Bloom-filter + Locality-Sensitive Hashing

Departamento de Electrónica, Telecomunicações e Informática

UNIVERSIDADE DE AVEIRO

Pedro Martins 76551, pbmartins@ua.pt
Ricardo Jesus 76613, ricardojesus@ua.pt

17 de Dezembro de 2015

Resumo

ABSTRACT

Conteúdo

Lista de Figuras

Capítulo 1

Introdução

Na área da informática, muitas vezes é necessário saber se algo pertence a conjunto de forma eficiente. Em muitas linguagens de programação, a maneira mais comum seria percorrer todo o conjunto e comparar, um a um, todos os elementos até encontrar o que se inicialmente andava à procura. No entanto, este método é deveras ineficiente, principalmente se se estiver a trabalhar com conjuntos com milhões ou mais elementos, e aí que surgem os *Bloom-filters*.

Este filtros usam uma ou várias funções de *hashing* para determinar a pertença de um elemento no *set*. São muito utilizados em grandes conjuntos de dados e em diversas aplicações como o corretor ortográfico dos computadores, *smartphones*, etc., análise textual, entre outras. Contudo, é um método probabilístico com um valor de erro associado.

Neste trabalho, foi desenvolvido em Matlab um *Bloom-filter* e desenhada uma interface para um função de *hashing* desenhada em C++ para atingir este objetivo.

Capítulo 2

Bloom-filter

Os *Bloom-filters* usam uma ou várias funções de *hashing* para determinar a pertença de um elemento no *set*. Deste modo, mesmo que se opere sobre um conjunto com milhões ou milhares de milhões de elementos, será sempre um processo muito mais eficiente do que se se iterasse sobre todo o *set* à procura do elemento em questão (dependendo também do número e da qualidade das funções de *hashing* utilizadas).

No entanto, é um método probabilístico, e tem um erro associado. Na criação de um *Bloom-filter* define-se sempre um erro máximo esperado, erro esse que apenas ocorre na "inclusão", isto é, um membro que à partida já se saiba que pertença ao conjunto nunca será considerado não pertencente, mas alguns membros que, à partida, se saiba que não estão no *set*, podem ser considerados como pertencentes ao conjunto.

Para diminuir esse erro, por norma, para além de aumentar o tamanho do *set*, usa-se um número variado de funções de *hashing* descorrelacionadas entre si, em vez de utilizar apenas uma. Isto permite que os resultados do *hashing* estejam mais dispersos por todo o vetor. Neste caso, em vez de utilizarmos k funções diferentes, utilizaremos, para uma mesma *string* "s", várias que variarão entre "s1", "s12", ..., "s12...k".

Na implementação do filtro neste trabalho (*BloomFilter.m*) foi desenvolvida uma classe, baseada num *Counting Bloom-filter* (conta-se o número de vezes que cada elemento é adicionado ao filtro), com 5 atributos:

k Número de funções de *hashing* utilizadas.

byteArray Limite de vezes que o contador de cada posição é incrementado (255).

arraySize Tamanho do vetor de *bits*.

amountAdded Número total de elementos adicionados ao ⌊array⌋.

expectedMaxSize Tamanho do conjunto que se pretende adicionar do vetor.

No construtor da classe, são calculados os valores do tamanho do vetor (`arraySize`) e do número de funções de *hash* necessárias, consoante os valores passados como argumentos do próprio construtor, a probabilidade de falsos positivos, isto é, o erro esperado (`falsePositiveProbability`) e o tamanho do conjunto que se pretende adicionar ao vetor (`expectedMaxSize`).

Assumindo que a probabilidade de falsos positivos p é

$$p = \left(1 - e^{-\frac{km}{n}}\right)^k$$

e, usando o tamanho do vetor de *bits* n e o tamanho do conjunto que queremos adicionar ao filtro m,

$$a = \left(1 - \frac{1}{n}\right)^m$$

para determinar o número k ótimo de funções de *hashing* que se devem utilizar, deduz-se que

$$\ln p = k * \ln\left(1 - a^k\right) \Leftrightarrow k = \frac{n * \ln 2}{m}$$

A partir das fórmulas acima encontradas, também se deduz que

$$n = \frac{m * \ln\left(\frac{1}{p}\right)}{(\ln 2)^2}$$

Para além do construtor, existem também métodos para adicionar e verificar a existência de elementos no filtro.

getIndexes Devolve os vários índices para os quais a função de *hashing* aplicada à *string* passada como argumento aponta.

add Adiciona um elemento ao filtro.

contains Verifica se o elemento passado como argumento existe no filtro (poderá haver ocorrência de falsos positivos).

count Devolve o número de vezes que um dado elemento foi adicionado ao vetor.

remove Remove do filtro, caso exista, o elemento passado como argumento.

maxCount Devolve o número máximo de vezes que um elemento foi adicionado ao vetor.

minCount O inverso da função `maxCount`.

Setters Coleção de funções utilizadas para modificar os atributos do *Bloom-filter*, caso o campo `debug` (argumento passado ao construtor) esteja com o valor 1.

2.1 Testes

Para testar este módulo, foram desenvolvidos diversos, de entre os quais uns para verificar qual seria o número ideal de funções de *hashing* (`k`) e outro para verificar o tamanho ideal do vetor de *bits* (`n`).

No entanto, também foram realizados outros testes relativamente às funções de *hashing* utilizadas pelo *Bloom-filter*.

2.1.1 Distribuição de funções de *hashing*

Este módulo (*test_ hashFunction_ distribution.m*) tem como principalmente objetivo provar que as funções de *hashing* têm uma distribuição uniforme, para diversos valores de `k` (neste caso, irá variar entre 1 e 10).

A função de *hashing* base usada tanto no filtro como nos testes é designada por *FarmHash*, desenvolvida pela Google. Foi apenas criada uma interface em C++ para que pudesse ser usada em Matlab.

Neste teste, foi gerado um conjunto de *strings* aleatórias, usando a função `generateStrings`, que aceita como argumentos o tamanho do conjunto que devolverá e o tamanho máximo das *strings* que irá gerar (caso o terceiro argumento seja 0, elas terão tamanho fixo, caso contrário, será definido como tamanho máximo), e gerado diversos valores de *hashing*, consoante os diversos elementos do conjunto e da `seed` correspondente (caso a *string* seja "s12", a `seed` será "12"). Isto é, caso tenhamos `k = 2`, será aplicada a função para "s1"e "s12", para uma determinada *string* "s".

As distribuições para cada valor de `k` deverão ser o mais uniformes possíveis, e os resultados comprovam-no, tal como podemos ver nos histogramas da 2.1.

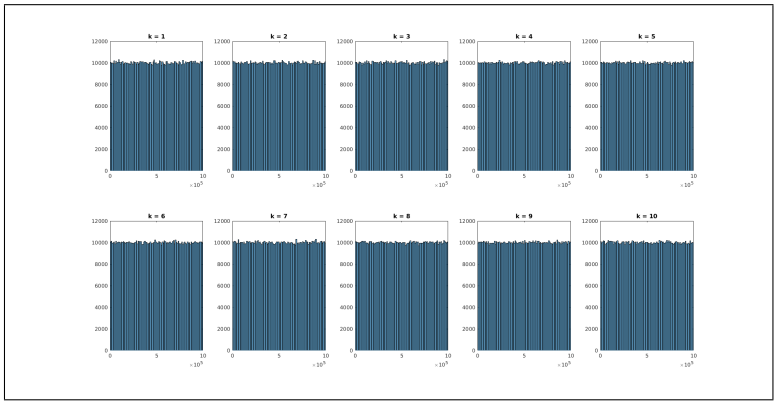

Figura 2.1: Resultados das distribuições das `k` funções de *hashing*

2.1.2 Correlação das funções de *hashing*

Para que se possa utilizar as várias funções de *hashing*, é necessário que as mesmas sejam descorrelacionadas, isto é, que o coeficiente de correlação seja 0. No entanto, neste caso, vamos considerar um erro de 0.01.

Este módulo encontra-se no ficheiro *test_ hashFunction_ correlation.m* e, tal como noutros testes, gerar-se-á um *set* de *strings* aleatórias e os respetivos valores de *hashing* para diferentes valores de `k`. Foi também criada uma matriz de dimensões `k` por `numTests`, isto é, cada linha terá diferentes valores de *hashing* para um mesmo valor de `k`. De seguida, calcular-se-á os coeficientes de correlação entre cada linha desses mesmos valores com o auxílio da função `corrcoef` (calcula o coeficiente de correlação entre dois conjuntos, neste caso, duas linhas distintas).

Por mim, é gerado um gráfico recorrendo á função `surf` para mostrar os valores de correlação resultantes. Como verificamos, todos os valores situam-se abaixo do erro assumido (á exceção dos valores em que comparamos a mesma linha, sendo que, nesse caso, o valor será 1), daí podermos considerar que as funções de `hashing` são descorrelacionadas, tal como pretendíamos.

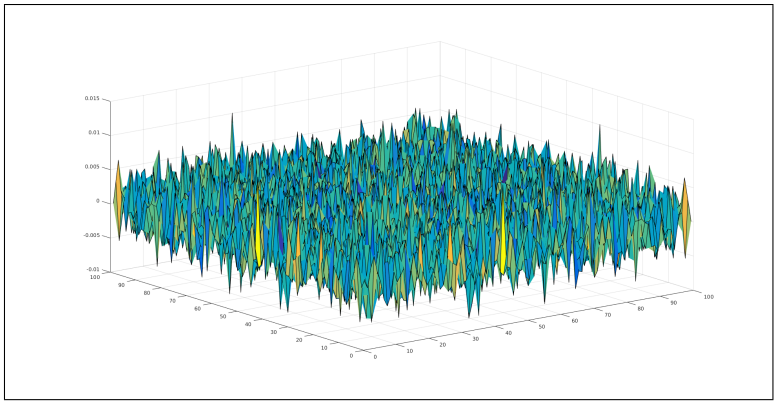

Figura 2.2: Resultados do teste de correlação de `k` funções de *hashing*

2.1.3 Número ideal de funções de *hashing*

O programa de teste para o valor de `k` ideal encontra-se com o nome *test_ optimalK.m.*

Começa-se por criar uma instância da classe de *Bloom-filter* anteriormente desenvolvida (com o campo `debug` inicializado a 1, para que se possam utilizar as funções de alteração de valores) e gerar dois conjuntos (um dos quais será adicionado ao filtro, outro não) distintos de cem mil (valor que pode ser alterado) *strings* (como poderá haver *strings* iguais nos *sets*, o tamanho será sempre ligeiramente inferior ao definido anteriormente). De seguida, define-se o tamanho do vetor de *bits* do filtro como oito vezes maior

do que o tamanho dos conjuntos de *strings* gerados.

Por fim, itera-se sobre um vetor de `k` que se definira anteriormente (neste caso, é um vetor com valores de 1 a 15) e, cada ciclo, define-se um `k` no *Bloom-filter*, adiciona-se os elementos do vetor incialmente escolhido como aquele que se iria adicionar ao filtro e verifica-se se algum dos elementos do outro conjunto de *strings* pertence ou não ao filtro. Caso pertença, é incrementado um contador, para, no final do ciclo, ser calculada a probabilidade de falsos positivos.

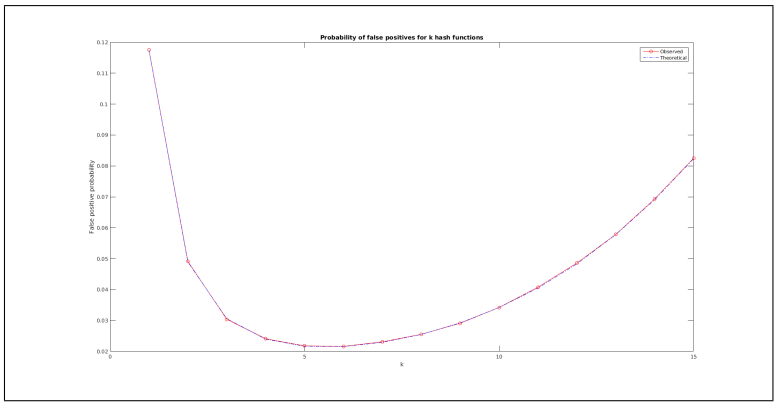

Figura 2.3: Resultados do teste do valor ideal de funções de *hashing* (`k`)

De acordo com o gráfico da 2.3, usando a fórmula

$$p = \left(1 - e^{-\frac{km}{n}}\right)^k$$

para determinar a probabilidade de falsos positivos (teórica) para diferentes valores de `k` e usando os valores do teste acima, verificamos que as diferenças entre o valor teórico e observado são mínimas (o que significa que a função de *hashing* devolve excelentes resultados, mas isso será abordado mais abaixo), e que o valor ideal é 6, para a função de *hashing* em questão. Para uma função diferente, os valores poderão diferir.

2.1.4 Número ideal do vetor de *bits*

O módulo para o teste do valor ideal do tamanho do vetor `n` encontra-se no ficheiro *test_ optimalN.m*.

Tal como no teste do Subseção 2.1.3, também são criados dois conjuntos de `strings` aleatórias com o mesmo propósito, um para ser adicionado ao filtro e o outro não. É também criado um vetor com diferentes valores de `n`, que vão desde o tamanho dos conjuntos de *strings* até 10 vezes esse valor, com uma diferença de metade do mesmo valor entre cada.

De seguida, itera-se sobre os valores deste último vetor de valores e vai-se criando uma instância de um *Bloom-filter* com os valores de `n` (`arraySize`)

e `k` (depende do tamanho do vetor) a cada passagem. Adiciona-se um dos conjuntos de *strings*, verifica-se a existência dos elementos do outro que não foi adicionado e, por fim, calcula-se a probabilidade de falsos positivos, tal como no teste anterior.

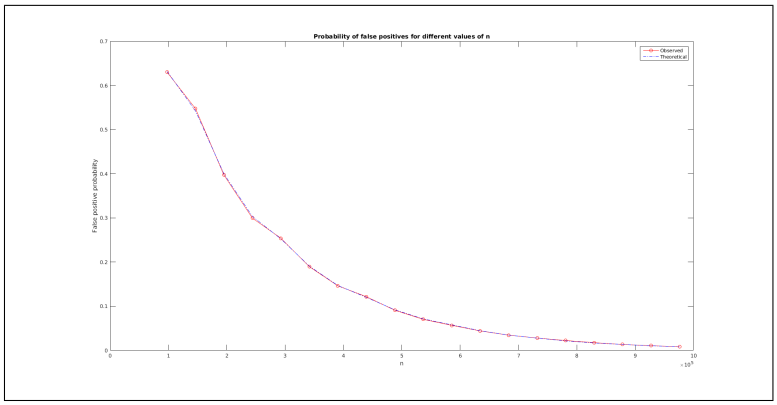

Figura 2.4: Resultados o teste do tamanho ideal do vetor de *bits* (`arraySize / n`)

Como se verifica no gráfico da 2.4, à medida que o tamanho do *array* aumenta, a probabilidade de falsos positivos diminui, isto é, são inversamente proporcionais.

Os valores ideias são os presentes na 2.5

```
Command Window
  Testing for n = 683137... Completed.
  Testing for n = 731933... Completed.
  Testing for n = 780728... Completed.
  Testing for n = 829524... Completed.
  Testing for n = 878319... Completed.
  Testing for n = 927115... Completed.
  Testing for n = 975910... Completed.

  Minimal probability of false positives (observed): 0.008185
  Optimal n (observed): 975910


  Minimal probability of false positives (thoeretical): 0.008194
  Optimal n (theoretical, that would have been used by default): 975881
fx >>
```

Figura 2.5: Resultados ideias do tamanho ideal do vetor de *bits* (`arraySize / n`)

2.1.5 Testes práticos do *Bloom-filter*

Foram desenvolvidos dois módulos semelhantes para um teste mais direcionado ao uso real de um *Bloom-filter*. Estão em dois ficheiros distintos,

test_big_BloomFilter.m e *test_small_BloomFilter.m*.

O primeiro segue o modelo dos testes anteriores, em que se gera dois *sets* de um certo tamanho (neste caso, cem mil elementos) de *strings* aleatórias, cria-se uma instância de um *Bloom-filter* com uma probabilidade de falsos positivos de 0.0001%, adiciona-se um dos conjuntos ao filtro e verifica-se a existência de falsos positivos. Por fim, compara-se o resultado observado da probabilidade de falsos positivos com a que se definiu aquando a criação do filtro. O objetivo é que os resultados sejam o mais próximos possíveis, algo que se conseguiu atingir em ambos os testes.

No segundo teste (*small*), contrariamente ao primeiro em que se geram os conjuntos, definem-se conjuntos muito pequenos inicialmente e, depois, executa-se o mesmo processo que o anterior.

```
Generated strings with maximum random length: 50
Probability of false positive: 0.000100
Length of the set to add (m): 97661

97661 randomly generated strings added to the filter.

97661 strings that were previously added are probably in the set.
0 strings that were previously added are not in the set.

12 strings that were not added are probably in the set.
96860 strings that were not added are not in the set.
Probability of false positives (observed): 0.000124
```

Figura 2.6: Resultados da execução de *test_big_BloomFilter.m*

2.2 MinHash

Referências

[1] *Hammond organ*, http://en.wikipedia.org/wiki/Hammond_organ, 2015.

[2] *Synthesizing hammond organ effecss*, http://www.soundonsound.com/sos/jan04/articles/synthsecrets.htm, 2015.

[3] *Synthesizing tonewheel organs*, http://www.soundonsound.com/sos/nov03/articles/synthsecrets.htm, 2015.