Estimativa do número de itens diferentes guardados num Bloom Filter

Lucas Barros, Nº Mec. 83895 MEI
Pedro Cavadas, Nº Mec. 85090 MEI

***Resumo* – Este relatório foi escrito no âmbito da cadeira de "Algoritmos Avançados" no curso de "Mestrado em Informática" na UA. O objetivo deste trabalho é estimar o número de itens distintos de um conjunto de dados, explorando métodos que permitem processar conjuntos de dados de grande dimensão. Ao longo deste relatório serão explicado todos os passos da estratégia, e também será efetuado um estudo da variação de certos parâmetros. Por fim, serão apresentadas as nossas conclusões e serão discutidos os resultados obtidos.**

***Abstract* – This report was written under the subject of "Algoritmos Avançados" under the course of "Mestrado em Engenharia Informática" in UA. The purpose of this work is to determine the number of distinct items in a data structure, by exploring methods that allow the process of large amounts of data. Through this report it will be explained all the steps of the strategy and it will also be presented a study on the variation of certain parameters. In the end, it will be presented our conclusions and the discussion of results.**

I. Introdução

No âmbito da Unidade Curricular de Algoritmos Avançados, foi proposto aos alunos a escolha de um problema para analisarem. O problema escolhido foi o de estimar o número de elementos distintos num "Bloom Filter". Um Bloom Filter possui vários parâmetros que dependendo do problema, devem ser alterados para melhorar a sua performance. Por último, foi requisitado aos alunos que realizassem um estudo estatístico variando esses mesmos parâmetros, para retirarem conclusões e procederem a uma breve sumarização destas.

Este projeto tem várias componentes: um script para gerar n números primos, uma classe "Bloom Filter" que vai permitir realizar as operações possíveis de um Bloom Filter, e o programa principal que vai conter toda a experiência efetuada.

Neste relatório vão ser primeiro explicados conceitos teóricos importantes para a perfeita compreensão de todo o trabalho. De seguida, vão ser explicadas todas as bibliotecas utilizadas e em seguida, todos os passos da experiência efetuada. Após isso, vão ser explicados mais detalhadamente todos os módulos principais do projeto. Por fim, vão ser apresentados os resultados em gráficos e a sua discussão e por último, as conclusões.

II. Bloom Filter

Um bloom filter é uma estrutura de dados probabilística, que tem como principal objetivo ser capaz de afirmar se um elemento pertence ao conjunto ou não. Bloom Filters são utilizados em aplicações de análise de texto, aplicações ao nível da rede, por exemplo guardar os "web objects" requisitados pelos utilizadores, etc…

A ideia principal do funcionamento de um Bloom Filter a seguinte: O Bloom Filter, que pode ser pensado como um array de inteiros, é inicializado a 0's. Assumindo que nos é dada uma string "s" para inserir, recorre-se a uma Hash Function "h(x)" que mapeie qualquer string num número inteiro. Fazendo h(s) irá retornar um número inteiro "i". De seguida lê-se o valor do índice i do Bloom Filter e caso este seja 0, o valor passa a ser 1. Caso contrário, o valor mantém-se a 1.

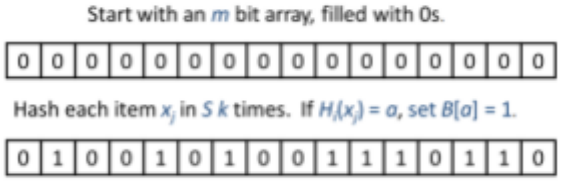

Figura 1 - Inicialização de *Bloom Filter* (em cima); *Bloom Filter* após ser inserido um elemento (em baixo)

Para saber se um valor pertence ou não ao Bloom Filter, basta aplicar a mesma Hash Function a esse valor e ir ao índice correspondente e verificar se o valor é 1 ou não. Caso seja 1, então pertence, caso contrário não pertence.

Existe no entanto a possibilidade de haver falsos positivos, pois uma função de hash pode mapear duas strings diferentes para o mesmo inteiro. A probabilidade de tal acontecer é pouca, caso estejamos a usar uma boa função de hash, mas ainda assim existe.

Existem uma série de parâmetros subjacentes ao Bloom Filter como "n" (tamanho do conjunto de elementos a inserir no filtro), "m" (número de células / tamanho do filtro que é normalmente um múltiplo de n), "k" (número de funções de hash) e "f" (fração de células a 1).

As questões principais deste trabalho são como escolher os parâmetros m e k de forma a minimizar o número de falsos positivos, e dar uma estimativa mais aproximada do número de itens distintos.

V. Bibliotecas/Programas Utilizados

A. *Os* – o módulo *os* do python possui um conjunto de funções para aceder e manipular o sistema operativo. Neste projeto, este módulo foi utilizado para aceder ao diretório com todos os ficheiros de texto e aceder a estes de uma forma dinâmica;

B. *Re* – o módulo *re* do python possui ferramentas para verificar expressões regulares. Neste projeto, esta ferramenta foi utilizada para extração dos tokens (palavras) dos ficheiros de texto;

C. *String* – módulo nativo do *Python3*. Este módulo contém algumas funcionalidades relacionadas com *strings*. Foi utilizado neste trabalho para remover sinais de pontuação;

D. *Math* – Também um módulo nativo ao *Python3*, que fornece ferramentas matemáticas. Utilizado para arredondamentos e cálculos de logaritmos e raízes quadradas;

E. *Random* – Outro módulo nativo ao *Python3*. Contém ferramentas de geração de valores aleatórios. Neste trabalho utilizado para a escolher aleatoriamente números primos para as funções de *hashing*;

F. a. *Venv* – módulo nativo do *Python3* (a partir da versão 3.3). Este módulo permite tornar o código *python* portátil e sem necessidade de instalar bibliotecas no ambiente *default* do *python*, o que por outro lado permite remover todas as bibliotecas e módulos não nativos do *python* apagando uma pasta. Neste trabalho é utilizado para que não haja a necessidade de instalar o módulo *prettytable* e *matplotlib*;

G. *Prettytable* – módulo não nativo do *Python3*. Este módulo permite de forma simples imprimir tabelas em ficheiros ou no *standard output* e neste trabalho foi utilizado para mostrar os resultados dos testes.

H. *Matplotlib* - módulo não nativo do *Python3*. Permite facilmente criar gráficos com resultados estatísticos. Neste trabalho foi utilizado para comparar o valor ótimo teórico de *k* (número de funções de hashing) e o valor prático, bem como o tamanho do filtro.

VI. Experiência

Na experiência efetuada, recorreu-se a 2 livros "Os Maias" e "Os Lusíadas". Extraindo os tokens (palavras) dos livros, adicionou-se ao Bloom Filter apenas as palavras distintas. Em cada inserção, verifica-se se a palavra já "pertence" ao filtro e caso sim, então estamos na presença de um falso positivo (pois sabemos de antemão que todas as palavras são distintas). Para cada livro, fez-se um estudo a variar os parâmetros "m" e "k". Sabendo que na teoria o valor de k que minimiza o número de falsos positivos é dado pela expressão
ln(2) x m / n, comparamos os valores obtidos ao valor teórico. Por fim, recorrendo à biblioteca Matplotlib fazemos gráficos com os resultados. Possuímos um gráfico para cada valor de "m" e em cada gráfico, no eixo dos x é variado o "k" e no y é mostrado o erro relativo que é a diferença entre o número de itens distintos no Bloom Filter e o número de itens distintos real.

VII. Módulos do Projeto

A. prime_generator.py

Este módulo tem como única funcionalidade a de permitir a geração de *n* números primos, começando em *x* é o valor mínimo que os números gerados podem ser, isto é, se *x* for um número primo então será o menor gerado pelo módulo, se não for, então o menor será o número primo mais pequeno a seguir a *x*.

B. bloom_filter.py

Este módulo tem implementa um *Bloom Filter* que pode ser reutilizado por outras aplicações.

C. main.py

Este módulo é a principal componente do projeto. É o módulo que utiliza o *Bloom Filter* para realizar a inserção de palavras de um livro e a respetiva contagem (aproximada), contém uma função para carregar os livros em forma de lista de palavras para memória:

```
def load_book(path, encoding = 'ascii'):
    with open(path, 'r', encoding = encoding)
as fin:
        return [ word for word in re.split(r'
+',
```

```
fin.read().translate(translator).strip().lower(
)) ]
    return []
```

contém outra para escrever para ficheiro os resultados:

```
def write_results(file, metrics_pt, table_pt,
results):
    if type(file) == str:
        with open(file, 'w', encoding =
'utf-8') as fout:
            write_results(fout, metrics_pt,
table_pt, results)
    else:
        file.write('Info. Table\n')
        file.write('{}\n'.format(metrics_pt))

        for book, book_results in
results.items():
            for size, size_results in
book_results.items():
                inserted_pt = table_pt.copy()
                approxst_pt = table_pt.copy()
                for size, k, theoric,
counts_analysis, lengths_analysis in
size_results:
                    inserted_pt.add_row([ size,
k, theoric ] + counts_analysis)
                    approxst_pt.add_row([ size,
k, theoric ] + lengths_analysis)
                file.write('\nAnalysis by
inserted (book = "{}")\n'.format(book))

file.write('{}\n'.format(inserted_pt))

                file.write('\nAnalysis by
statistical approximation (book =
"{}")\n'.format(book))

file.write('{}\n'.format(approxst_pt))
```

contém também uma função que realiza *n runs* para um dado número de funções de hash, um dado tamanho de filtro do *Bloom Filter* e um dado conjunto de palavras:

```
def test(args):
    counts, lengths = list(), list()
    diff_words, k, size, amount = args
    for i in range(amount):
        bf = BloomFilter(m = size *
len(diff_words), k = k)
        counts.append(bf.extend(diff_words))
        lengths.append(len(bf))
    return lengths, counts, k
```

finalmente contém também uma função para analisar os resultados dos testes:

```
def analyze(expected_counter, counters):
    abs_errors = [ abs(c - expected_counter)
for c in counters ]
    rel_errors = [ error / expected_counter for
error in abs_errors ]

    mean_counter = mean(counters)
    diffs = [ abs(value - mean_counter) for
value in counters ]
    maxdev = max(diffs)
    mad = mean(diffs)
    stddev = math.sqrt(mean([ value * value for
value in counters ]))

    return [
        max(rel_errors),
        min(rel_errors),
        mean(rel_errors),
        mean(abs_errors),
        expected_counter,
        max(counters),
        min(counters),
        mean_counter,
        mad,
        maxdev,
        stddev,
        stddev * stddev,
    ]
```

VIII. Análise dos resultados

Após realizar os testes e gerar os resultados, pudemos verificar olhando para o valor média da contagem, quando esta é feita tendo em conta o número de palavras inseridas no *Bloom Filter*, que o número de funções de *hashing* "ótimo" na prática difere do valor teórico entre 1 a 2 unidades:

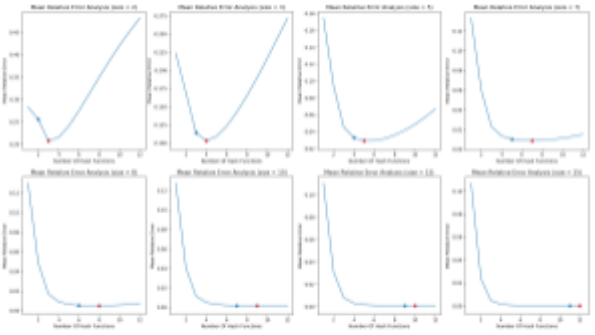

Figura 2 - Média dos erros relativos e comparação do valor ótimo teórico (ponto a azul) e do prático (ponto a vermelho)

este facto pode dever-se às funções de *hashing* utilizadas, já que o valor teórico é seria numa situação com funções de *hashing* "ideais".

Além disto, pudemos também verificar que a partir de um certo tamanho do filtro e de um certo número de funções de *hashing* utilizar o método de contagem pelo número de palavras inseridas dá-nos uma melhor estimativa do que a aproximação estatística. Isto ocorre então para um tamanho de filtro de pelo menos 7 vezes o número de palavras a inserir e com pelo menos 2 funções de *hashing*. Este facto pode ser verificado olhando para a média do erro relativo que começa a ser menor para esses números para a estimativa com base na inserção:

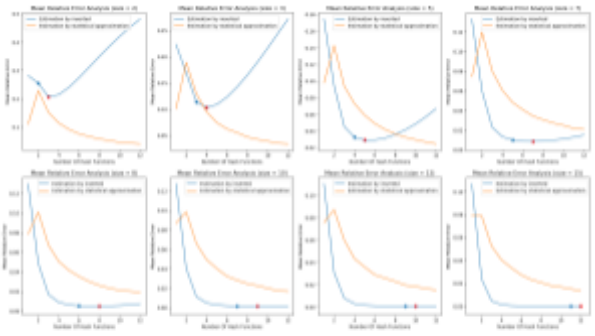

Figura 3 - Comparação entre estimativa por contagem de inserção e estimativa por aproximação estatística

isto implica que um *Bloom Filter* com um poucos falsos negativos necessite de 8 vezes o número de palavras a inserir em *bits* o que significa que serão utilizados nesse caso *n bytes*, onde *n* é o número de palavras a inserir.

Finalmente pudemos também verificar que o melhor *trade off* de memória e processamento para a obtenção de um *Bloom Filter* com uma probabilidade de falsos negativos baixa, é com um filtro de tamanho 8 vezes superior ao número de palavras e com 7 funções de *hashing*. Estes valores podem ser debatíveis, e portanto, aceita-se filtros de tamanho entre 7 e 8 e um número de funções de *hashing* entre 6 e 7, como sendo todos bons *trade offs*. Isto pode ser verificado olhando para o erro relativo, mas também para os valores de contagem:

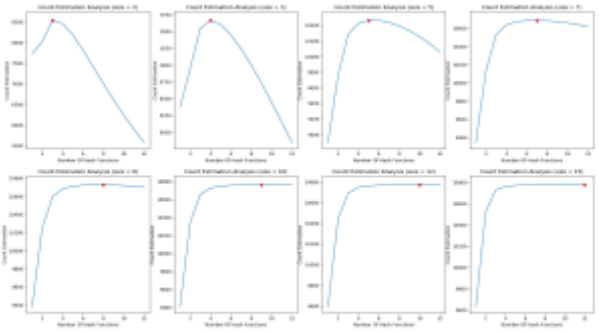

Figura 4 - Valores de contagem com uma estimativa por contagem de inserção

X. Conclusão

Podemos concluir com este trabalho 3 pontos principais:

- os valores ótimos de *k* teóricos de forma geral diferem dos práticos;
- os valores ideais de *m* são entre 7 e 8 vezes o número de palavras a inserir e os de *k* entre 6 e 7;
- a estimativa por aproximação estatística compensa apenas para valores poucas filtros de pouco tamanho e com poucas funções de *hashing*.

XI. Referências

[1]https://en.wikipedia.org/wiki/Bloom_filter- Bloom Filter

[2] https://docs.python.org/3/library/re.html - Regular expressions operations

[3] https://docs.python.org/2/library/os.html - os

[4]https://matplotlib.org/ - Matplotlib: Python plotting