PL 4

Algoritmos Probabilísticos

4.1 Funções de dispersão

O objectivo destes exercícios é avaliar o funcionamento de uma estrutura de dados importante, a *Chaining Hash Table*, e um dos conceitos que a suportam, *Hash Functions* (funções de dispersão).

1. Crie uma função para gerar chaves constituídas por caracteres. O comprimento (i.e., o número de caracteres) de cada chave deve ser escolhido aleatoriamente (distribuição uniforme) entre i_{min} e i_{max}. A função deve ter como parâmetros de entrada o número N de chaves a gerar, os valores de i_{min} e i_{max}, um vector com os caracteres a usar nas chaves e um vector com as probabilidades de cada um dos caracteres a utilizar. Se a função for chamada sem o último parâmetro, deve considerar os caracteres equiprováveis (ver a documentação da função `nargin`).

   A função deve devolver um "cell array"com o conjunto de chaves geradas garantindo que as chaves são todas diferentes.

   (a) Gere um conjunto de $N = 10^5$ chaves usando todas as letras maiúsculas e minúsculas ('A' a 'Z' e 'a' a 'z') com igual probabilidade e em que $i_{min} = 6$ e $i_{max} = 20$.

   (b) Gere um conjunto de $N = 10^5$ chaves usando todas as letras minúsculas ('a' a 'z') com as probabilidades contidas no ficheiro `prob_pt.txt` e que correspondem às frequências das letras em Português (`https://pt.wikipedia.org/wiki/Frequ%C3%AAncia_de_letras`). Considere novamente $i_{min} = 6$ e $i_{max} = 20$.

2. Considere a função Matlab `string2hash()` [1] que implementa duas funções de dispersão diferentes. Considere ainda 2 funções Matlab fornecidas que são adaptações para Matlab das funções de dispersão `hashstring()` [2] e `DJB31MA()` [3].

   Utilizando separadamente cada uma destas quatro funções de dispersão, simule a inserção das chaves criadas no exercício 1a) em 3 *Chaining Hash Tables*, uma de tamanho 5×10^5, outra de tamanho 10^6 e a terceira de tamanho 2×10^6. Para cada uma das simulações (4 funções de dispersão $\times$ 3 tamanhos):

   (a) Guarde um vetor com os *hashcodes* obtidos.

   (b) Registe o número de atribuições a cada uma das posições de cada *Hash Table*.

   (c) Calcule o número de colisões (em cada *Hash Table* e para cada função de dispersão).

   (d) O tempo de execução da simulação.

3. Utilizando a informação obtida no exercício anterior, compare o desempenho das quatro funções de dispersão para cada tamanho diferente da *Hash Table*, relativamente a:

[1] `https://www.mathworks.com/matlabcentral/fileexchange/27940-string2hash`

[2] Função usada em Programação II que poderá ser adaptada ao matlab.

[3] Função baseada no algorítmo de Daniel J. Bernstein, ver sumário de MPEI de 2014 (Prof. Paulo Jorge Ferreira) ou slides TP.

(a) Uniformidade, de duas formas diferentes:

i. visualize os histogramas dos *hascodes* com 100 intervalos e verifique se os valores nos diferentes intervalos são similares;

ii. calcule os momentos de ordem 2, 5 e 10 das variáveis aleatórias correspondentes aos valores dos *hashcodes* divididos pelo comprimento da *Hash Table* (i.e, variável aleatória toma valores entre 0 e 1) e compare com os valores teóricos da distribuição uniforme. [4]

(b) Número de colisões e número máximo de atribuições numa mesma posição da *Hash Table*.

(c) Tempos de execução.

4. Repita os exercícios 2 e 3 usando agora as chaves criadas no exercício 1b). As conclusões destes resultados são semelhantes às do exercício 3?

4.2 Filtros de Bloom

Os objetivos destes exercícios são criar e testar um módulo que suporte a criação de *Bloom filters* (ex: [2, Cap. 4: Mining Data Streams]).

Crie um conjunto de funções Matlab que implementem as funcionalidades de um *Bloom Filter* básico. As funções devem ter os parâmetros necessários para que seja possível criar *Bloom Filters* de diferentes tamanhos (n) e a utilização de diferentes números de funções de dispersão (k). Na criação das diferentes funções de dispersão, use o método descrito no slide 49 da apresentação TP sobre funções de dispersão com a função que melhor desempenho teve na secção 4.1 anterior.

Sugestão: Criar pelo menos 3 funções [1, sec. 3.2]: uma para inicializar a estrutura de dados; outra para inserir um elemento (ou elementos) no filtro; uma terceira para verificar se um elemento pertence ao conjunto.

1. Com as funções que desenvolveu, crie um *Bloom Filter* para guardar um conjunto, U_1, de 1000 palavras diferentes[5]. Use um *Bloom Filter* de tamanho $n = 8000$ e $k = 3$ funções de dispersão.

2. Teste o *Bloom Filter* criado anteriormente, verificando a pertença de todas as palavras do conjunto U_1. Obteve algum falso negativo?

3. Teste o *Bloom Filter* criado anteriormente, verificando a pertença de um novo conjunto, U_2, com 10000 palavras todas diferentes das de U_1. Indique a percentagem de falsos positivos obtidos.

4. Compare a percentagem de falsos positivos obtida anteriormente com a estimativa que aprendeu nas TPs.

5. Repita os exercícios 1. e 3. para um número de funções de dispersão k de 4 até 10. Faça um gráfico com a percentagem de falsos positivos em função de k. Analisando os resultados, qual o valor ótimo k? Compare este valor com o valor teórico que aprendeu nas TPs.

4.3 Deteção de similaridade - Algoritmo MinHash

O objetivo destes exercícios é a criação e teste de um "módulo" que suporte a descoberta de conjuntos similares (ex: [2, Cap. 3: Finding Similar Items]).

Nestes exercícios iremos utilizar o ficheiro `u.data` do conjunto de dados (release 4/1998) MovieLens 100k, disponível em `http://grouplens.org/datasets/movielens/`. Este ficheiro contém informação sobre 943 utilizadores e 1682 filmes. Tem cerca de 100 000 linhas, como as seguintes:

```
196  242  3  881250949
186  302  3  891717742
 22  377  1  878887116
```

[4] Na distribuição uniforme entre 0 e 1, o valor do momento de ordem n é igual a $\frac{1}{n+1}$.

[5] Sugestão: pode utilizar as palavras válidas portuguesas do ficheiro `wordlist-preao-20201103.txt` utilizado na secção de avaliação do guião PL03.

As colunas são separadas por *tabs*: a primeira coluna contém o ID do utilizador, a segunda contém o ID de um filme (avaliado pelo utilizador mencionado na primeira coluna), a terceira é a avaliação dada pelo utilizador ao filme e a quarta coluna é um *timestamp* do momento da avaliação.

O objectivo é descobrir utilizadores que avaliaram conjuntos similares de filmes. Para este objectivo as colunas 3 e 4 não são necessárias.

1. Analise o código Matlab disponibilizado no final desta secção e complete-o por forma a conseguir calcular a **distância de Jaccard** entre os conjuntos de filmes avaliados pelos vários utilizadores.

   Inclua no código a possibilidade de calcular o tempo que demora cada uma das partes (cálculo da distância e determinação das distâncias abaixo de um determinado limiar). Veja a informção relativa às funções Matlab `tic, toc, cputime, etime`.

   No final, o programa deve mostrar informação com: (1) número de pares de utilizadores com distâncias inferiores ao limiar definido; (2) informação sobre cada par (identificação dos utilizadores e distância de Jaccard).

   Adicione, também, a capacidade de gravar em ficheiro a matriz de distâncias calculada. Sugere-se que consulte a informação da função `save`.

2. Com base no código que adaptou, crie funções para:

   (a) criar a estrutura de dados com os conjuntos de filmes;

   (b) calcular as distâncias entre conjuntos;

   (c) processar as distâncias e devolver os pares de conjuntos similares. Um dos parâmetros de entrada desta função deve ser o limiar de decisão.

   Faça um primeiro teste ao código considerando 100 utilizadores seleccionados de forma aleatória.

3. Depois do teste anterior com um número reduzido de utilizadores e da resolução de eventuais problemas detectados, execute o seu programa com todo o conjunto de dados por forma a determinar todos os pares de utilizadores com uma distância de Jaccard inferior a 0.4

   Tome nota dos tempos e dos resultados obtidos.

4. Crie uma nova versão da função de cálculo de distância recorrendo a uma aproximação probabilística usando *MinHash*. Comece por testar esta nova implementação com um número pequeno de utilizadores e depois teste-a com o conjunto total de utilizadores.

   Na implementação do *MinHash*, considere um número k de funções de dispersão de 50, 100 e 200.

   Compare os pares considerados como similares (e o seu valor de similaridade) para cada valor de k com os obtidos com a implementação não probabilística e retire conclusões.

```
% Código base para deteção de pares similares

udata=load('u.data');  % Carrega o ficheiro dos dados dos filmes
% Fica apenas com as duas primeiras colunas
u= udata(1:end,1:2); clear udata;

% Lista de utilizadores
users = unique(u(:,1));     % Extrai os IDs dos utilizadores
Nu= length(users);          % Número de utilizadores

% Constrói a lista de filmes para cada utilizador
Set= cell(Nu,1);            % Usa células
for n = 1:Nu,  % Para cada utilizador
    % Obtém os filmes de cada um
    ind = find(u(:,1) == users(n));
    % E guarda num array. Usa células porque utilizador tem um número
    % diferente de filmes. Se fossem iguais podia ser um array
    Set{n} = [Set{n} u(ind,2)];
end
```

```
%% Calcula a distância de Jaccard entre todos os pares pela definição.

J=zeros(...);  % array para guardar distâncias
h= waitbar(0,'Calculating');
for n1= 1:Nu,
    waitbar(n1/Nu,h);
    for n2= n1+1:Nu,
           %%  Adicionar código aqui
    end
end
delete (h)

%%  Com base na distância, determina pares com
%%  distância inferior a um limiar pré-definido

threshold =0.4  % limiar de decisão

% Array para guardar pares similares (user1, user2, distância)
SimilarUsers= zeros(1,3);
k= 1;
for n1= 1:Nu,
    for n2= n1+1:Nu,
        if  % ......
            SimilarUsers(k,:)= [users(n1) users(n2) J(n1,n2)]
            k= k+1;
        end
    end
end
```

A ser completado ...

Bibliografia

[1] James Blustein and Amal El-Maazawi. Bloom filters - a tutorial, analysis, and survey. Tecnhical Report CS-2002-10, Dalhousie University, Dec 2002.

[2] Jure Leskovec, Anand Rajaraman, and Jeff Ullman. *Mining of Massive Datasets*. Cambridge University Press, 2014.