Trabalho prático de AEDS II - Pagerank

Pedro Gabriel Drumond Pereira

27 de setembro de 2015

1 Introdução

PageRank é um algoritmo utilizado pela ferramenta de busca Google para posicionar websites entre os resultados de suas buscas. O PageRank mede a importância de uma página contabilizando a quantidade e qualidade de links apontando para ela. Não é o único algoritmo utilizado pelo Google para classificar páginas da internet, mas foi o primeiro utilizado pela companhia e o mais conhecido.

2 Implementação

As funções utilizadas pelo programa estão divididas em 2 TAD's um chamado vector.h e outro chamado matrix.h, o primeiro implementa funções simples de vetores e o segundo implementa as funções matricias do programa.

2.1 vector.h

As funções vetoriais do programa são:

- int isNullVector(double *, unsigned);
- void setVectorToOne(double *, unsigned);
- void normalizeVector(double *, unsigned);
- void dampVector(double *, double, unsigned);
- void printVector(double *, unsigned);
- void printIndexedVector(double *, unsigned);

2.1.1 int isNullVector(double *vector, unsigned size)

Essa função verifica se o **vector* só possui elementos de valor nulo. Para fazer isso ela verifica cada elemento do vetor, e caso todos sejam 0(zero) retorna 'verdadeiro', caso um dos elementos não seja 0 retorna 'falso'.

2.1.2 void setVectorToOne(double *vector, unsigned size)

Essa função troca todos *size* valores de **vector* para 1.

2.1.3 void normalizeVector(double *vector, unsigned size)

Transforma o **vector* em um vetor unitário, ou seja, calcula a soma de todos os seus elementos e em seguida divide cada um deles pela soma total dos elementos.

2.1.4 void dampVector(double *vector, double damp, unsigned size)

Aplica a equação de damping à todos os elementos do **vector*.

$$M[i] = (1 - \alpha) \times S[i] + \frac{\alpha}{N}$$

2.1.5 void printVector(double *vector, unsigned size)

Imprime todos os elementos de **vector* na saída padrão, em uma única linha.

2.1.6 void printIndexedVector(double *vector, unsigned size)

Imprime todos os elementos de **vector* numa lista ordenada, do primeiro ao último, cada um em sua linha de texto, com o seu número índice na frente.

2.2 matrix.h

As funções matriciais do programa são:

- void matrixAlloc(double ***, unsigned, unsigned);
- void matrixFree(double **, unsigned);
- void printMatrix(double **, unsigned, unsigned);
- void squareMatrixMultiplication(double **, double **, double **, unsigned);
- void matrixMemorySet(double **, unsigned, unsigned, int);
- void copyMatrix(double ***, double **, unsigned, unsigned);
- void matrixDifference(double **, double **, double ***, unsigned, unsigned);
- int matrixHasConverged(double **, double **, unsigned);
- void stochasticMatrix(double **, unsigned, unsigned);
- void dampMatrix(double **, double, unsigned, unsigned);
- void expontiationUntilConverge(double ***, double **, unsigned, unsigned);
- double matrixNorm(double **, unsigned, unsigned);

2.2.1 void matrixAlloc(double ***matriz, unsigned lines, unsigned columms)

Função para alocar uma matriz dinamicamente. Utiliza-se internamente da função Calloc, já que para o uso no programa é mais apropriado que a matriz tenha 0(zero) em todos os seus elementos.

2.2.2 void matrixFree(double **matrix, unsigned lines)

Função utilizada para liberar uma matriz alocada dinamicamente.

2.2.3 void printMatrix(double **matrix, unsigned lines, unsigned columms)

Função utilizada para imprimir todos os elementos de uma matriz. Dentro dessa matriz existem chamadas para a função *printVector*, para imprimir cada linha individualmente.

2.2.4 void stochasticMatrix(double **matrix, unsigned lines, unsigned columms)

Transforma uma matriz em uma matriz estocástica, aplicando as seguintes regras em ordem:

- Verifica se a matriz possui elementos nulos, utilizando da função *isNullVector*.
- Caso a primeira regra se aplique, troca todos os elementos da linha por um, utilizando da função *setVectorToOne*
- Normaliza a linha da matriz, utilizando da função *normalizeVector*.

2.2.5 void dampMatrix(double **matrix, double damp, unsigned lines, unsigned columms)

Aplica a equação de damping à matriz.

$$M[i] = (1 - \alpha) \times S[i] + \frac{\alpha}{N}$$

Utiliza-se da função *dampVector* aplicada a todas as linhas para fazer isso.

2.2.6 void matrixMemorySet(double **matrix, unsigned lines, unsigned columms, int value)

Função memset extendida para ser utilizada em uma matriz. Utiliza-se da função *memset* (que se encontra na biblioteca **string.h**) aplicada a todas as linhas da matriz.

2.2.7 void squareMatrixMultiplication(double **matrixA, double **matrixB, double **saida, unsigned lado)

Aplica a multiplicação de uma matriz quadrada utilizando a seguinte fórmula:

$$C[i][j] = \sum_{k=0}^{lado-1} a[i][k] \times b[k][j] \forall i, j < lado$$

2.2.8 int matrixHasConverged(double **matrixM, double **matrixN, unsigned size)

Verifica se uma matriz convergiu, utilizando de acordo com a seguinte equação:

$$M^n = \|M^n - M^m\| \leq 10^{-12} \text{ tal que } \|X\| = \sum_i \sum_j X[i][j]^2 \text{ e } m > n$$

Para fazer isso a função utiliza-se de uma chamada para *matrixDifference* para calcular a diferença, em seguida a norma da mesma é obtida com a função *matrixNorm*. Em seguida a matriz diferença é liberada da memória utilizando *matrixFree*. Como retorno da função temos $norma \leq 10^{-12}$.

2.2.9 void matrixDifference(double **M, double **N, double ***difference, unsigned lines, unsigned columms)

Calcula a diferença de duas matrizes. A única chamada para uma função externa é para *matrixAlloc*, onde alocamos um espaço na memória para a matriz diferença.

2.2.10 double matrixNorm(double **matrix, unsigned lines, unsigned columms)

Calcula a norma da matriz de acordo com a equação

$$\|X\| = \sum_i \sum_j X[i][j]^2$$

2.2.11 void expontiationUntilConverge(double ***matrixExp, double **matrixBase, unsigned size, unsigned iteration)

Função recursiva para calcular

$$M^n = \begin{cases} M^n \text{ se } n \geq 2000 \\ \|M^n - M^m\| \leq 10^{-12} \text{ tal que } \|X\| = \sum_i \sum_j X[i][j]^2 \text{ e } m > n. \end{cases}$$

Utiliza chamadas para as funções *matrixAlloc*, *squareMatrixMultiplication*,*matrixHasConverged*, *exponentiationUntilConverge*, *MatrixFree* e funciona da seguinte maneira:

1. Aloca-se uma matriz para receber M^{n+1}, calculada com *squareMatrixMultiplication*.
2. Verifica-se se a matriz convergiu usando a função *matrixHasConverged* com parâmetros de entrada M^{n+1}, M^n e o tamanho das matrizes.

3. Se não convergiu e o expoente atual é menor que 2000, a função chama a si mesma com os parâmetros: M^{n+1}, M^n, o tamanho e o expoente atual.

4. Caso tenha convergido ou a chamada tenha retornado, é liberada a matriz M^N e atribuida ao seu endereço a matriz M^{n+1}.

2.2.12 void copyMatrix(double ***output, double **input, unsigned lines, unsigned collumms)

Faz uma cópia dos elementos de uma matriz A para uma matriz B.

3 Estudo de complexidade

3.1 int isNullVector(double *, unsigned)

Operação relevante: Comparação.
N: Tamanho do vetor.
$f(n) = \begin{cases} \text{Melhor caso:} & 1 \\ \text{Caso médio:} & \frac{n+1}{2} \\ \text{Pior caso:} & n. \end{cases}$
Limite assintótico: $f(n) = O(n)$

3.2 void setVectorToOne(double *, unsigned)

Operação relevante: Atribuição.
N: Tamanho do vetor.
$f(n) = n$
Limite assintótico: $f(n) = O(n)$

3.3 void normalizeVector(double *, unsigned)

Operações relevantes: Soma e divisão.
N: Tamanho do vetor.
$f(n) = 2n$
Limite assintótico: $f(n) = O(n)$

3.4 void dampVector(double *, double, unsigned)

Operação relevante: Damping.
N: Tamanho do vetor.
$f(n) = n$
Limite assintótico: $f(n) = O(n)$

3.5 void printVector(double *, unsigned)

Operação relevante: Impressão na tela.
N: Tamanho do vetor.
$f(n) = n + 1$
Limite assintótico: $f(n) = O(n)$

3.6 void printIndexedVector(double *, unsigned)

Operação relevante: Impressão na tela.
N: Tamanho do vetor.
$f(n) = n$
Limite assintótico: $f(n) = O(n)$

3.7 void matrixAlloc(double ***, unsigned, unsigned)

Operação relevante: Calloc.
N: Quantidade de linhas na matriz.
$f(n) = n + 1$
Limite assintótico: $f(n) = O(n)$

3.8 void matrixFree(double **, unsigned)

Operação relevante: Free.
N: Quantidade de linhas na matriz.
$f(n) = n + 1$
Limite assintótico: $f(n) = O(n)$

3.9 void printMatrix(double **, unsigned, unsigned)

Operação relevante: Impressão na tela.
N: Quantidade de linhas e colunas na matriz.
$f(n) = n^2 + 2$
Limite assintótico: $f(n) = O(n^2)$

3.10 void squareMatrixMultiplication(double **, double **, double **, unsigned)

Operações relevantes: Multiplicação, Memset
N: Quantidade de linhas e colunas na matriz.
$f(n) = n^3 + n$
Limite assintótico: $f(n) = O(n^3)$

3.11 void matrixMemorySet(double **, unsigned, unsigned, int)

Operação relevante: Memset.
N: Quantidade de linhas na matriz.
$f(n) = n$
Limite assintótico: $f(n) = O(n)$

3.12 void matrixDifference(double **, double **, double ***, unsigned, unsigned)

Operação relevante: Subtração, Calloc.
N: Quantidade de linhas na matriz.
$f(n) = n^2 + n + 1$
Limite assintótico: $f(n) = O(n^2)$

3.13 void copyMatrix(double ***, double **, unsigned, unsigned)

Operação relevante: Atribuição, Free, Malloc.
N: Quantidade de linhas na matriz.
$f(n) = n^2 + 2n + 2$
Limite assintótico: $f(n) = O(n^2)$

3.14 void dampMatrix(double **, double, unsigned, unsigned)

Operação relevante: Damping.
N: Quantidade de linhas e colunas na matriz.
$f(n) = n^2$
Limite assintótico: $f(n) = O(n^2)$

3.15 double matrixNorm(double **, unsigned, unsigned)

Operação relevante: Exponenciação.
N: Quantidade de linhas e colunas na matriz.
$f(n) = n^2$
Limite assintótico: $f(n) = O(n^2)$

3.16 void stochasticMatrix(double **, unsigned, unsigned)

Operação relevante: Comparação, Atribuição, Soma, Divisão.
N: Quantidade de linhas e colunas na matriz.

$$f(n) = \begin{cases} \text{Melhor caso:} & 2n^2 + n \\ \text{Caso médio:} & \frac{6n^2+n}{2} \\ \text{Pior caso:} & 4n^2. \end{cases}$$

Limite assintótico: $f(n) = O(n^2)$

3.17 int matrixHasConverged(double **, double **, unsigned)

Operações relevante: Calloc, Subtração, Exponenciação, Free.
N: Quantidade de linhas e colunas na matriz.
$f(n) = 2n^2 + 2n + 2$
Limite assintótico: $f(n) = O(n^2)$

3.18 void expontiationUntilConverge(double ***, double **, unsigned, unsigned)

Operação relevante: Multiplicação.
N: Quantidade de linhas e colunas na matriz.
e: Expoente final da multiplicação.
$f(n) = e(n^3 + 2n^2 + 4n + 4)$
Limite assintótico: $f(n) = O(n^3 e)$

3.19 Função de complexidade do programa como um todo

O programa executa as seguintes funções:
matrixAlloc, *stochasticMatrix*, *dampMatrix*, *copyMatrix*, *expontiationUntilConverge*, *printIndexedVector* e duas vezes *matrixFree*.
Partindo do pressuposto (não necessariamente verdadeiro) que as operações relevantes de cada uma dessas funções é igualmente custosa à maquina, é possível afirmar que a função de complexidade do todo equivale à função de complexidade da soma de suas partes, logo:
n: Tamanho de um dos lados da matriz. (Ela deve necessariamente ser quadrada para o programa)
e: Expoente máximo atingido pela multiplicação de matrizes
$f(n) \simeq n^3 e + (e + 1)(2n^2 + 4n + 4)$
Limite assintótico = $f(n) = O(n^3 e)$

4 Avaliação Experimental - Estudo do comportamento empírico

4.1 Estudo da influência do valor de Damping no resultado final

Essa sessão trata-se de um estudo empírico, logo não prova ou desprova com acuracidade as hipóteses aqui levantadas, e se baseia apenas em evidências experimentais.

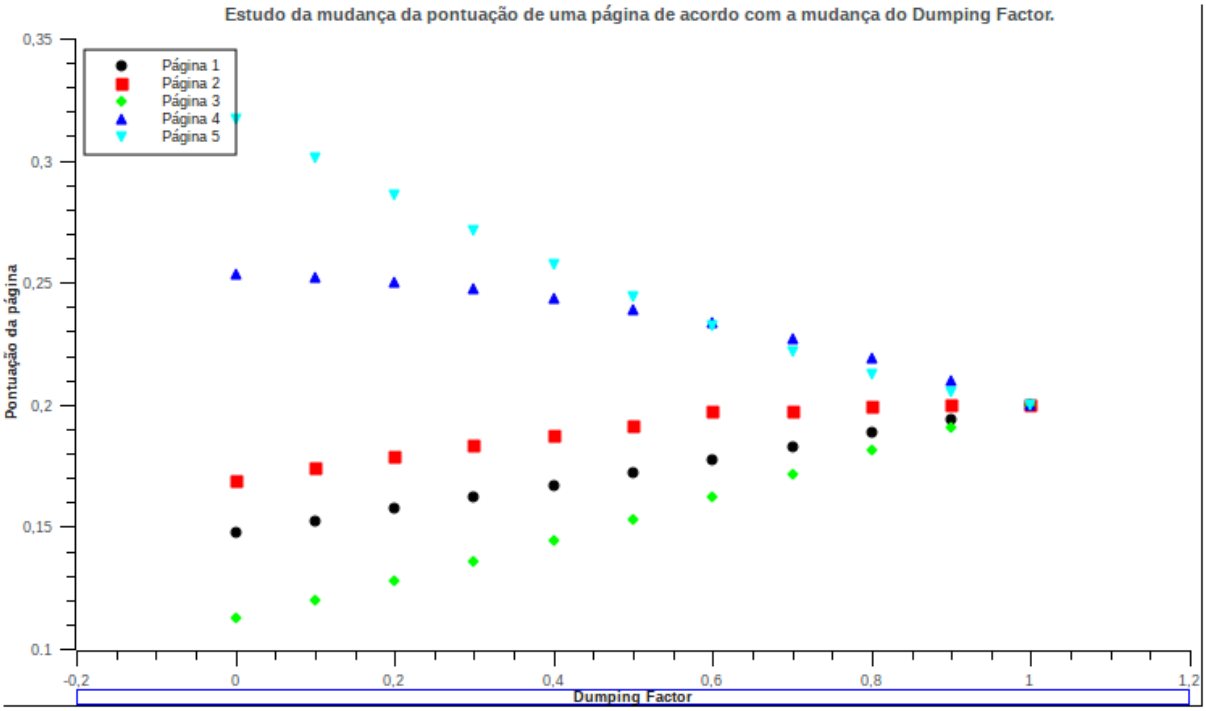
O teste consiste em utilizar a matriz $\mathcal{A}$ fixada e variar apenas o damping factor, para assim verificar sua relevância.

$$\mathcal{A} = \begin{bmatrix} 0 & 1 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

É possível perceber duas coisas com os testes realizados (os resultados registrados abaixo):

- Conforme o Damping tende a 1, tende-se à uma convergência dos valores para $\frac{1}{N}$.
- A página que está no topo quando o Damping Factor equivale a 0, não necessariamente vai estar no topo em todos os momentos.

Figura 1: Relevância do Damping Factor no resultado do Pagerank



Partindo dessas duas premissas e do gráfico, é possível afirmar concluir que:

1. No caso de haverem muitas páginas com a pontuação inicial $\geq \frac{1}{N}$ é possível que trocas de posição do encabeçamento da lista ocorram com uma frequência maior.
2. Existe a possibilidade de que as páginas entre a primeira e a última permutem a posição entre si.
3. O aumento do damping possui uma relevância diferente para cada página (A curva de crescimento é diferente entre as páginas).

5 Conclusão

A seção de análise empírica foi
bastante difícil de ser feita.
O TP é bastante honesto,
Embora a implementação não esteja perfeita.

A abordagem foi super simples,
Foi tranquila a heurística.
Meu calcanhar de Aquiles.
Foi dividir e conquistar,
Poderia ter sido $n^3 \log_2 e$,
mas não encaixou na minha logística.

Peço encarecidamente,
que analise com carinho.
Fiz o melhor que pude,
com o que apareceu em minha mente.
Desculpe-me pelo poema,
não resisti em fazê-lo,
é só um pouco de vicissitude,
me segurei para fazer só no finzinho.

6 Bibliografia

Pagerank - Wikipédia