Trabalho Prático 1 de AEDS 2

Page rank Dayman Novaes

**INTRODUÇÃO**

O PageRank é um algoritmo capaz de identificar qual o grau de relevância de uma página na web, que leva em conta diversos fatores, sendo um deles quantos links cada site recebe, considerando todas as relações possíveis e capaz de atribuir uma nota para a página, em uma espécie de ranking. Esse algoritmo foi o principal diferencial do Google.

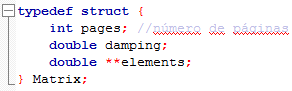
Esse trabalho tem como objetivo implementar um sistema de page ranking, através da modelagem de grafos a partir de uma matriz. A implementação inclui um Tipo Abstrato de Dados (TAD) matriz, que contém o número de páginas consideradas no problema, assim como as relações entre as páginas.

Dado como entrada, as relações entre as páginas (página 1 aponta para página 2, página 3 aponta para a página 5, etc.), o programa deverá calcular qual a relevância de cada uma.

**IMPLEMENTAÇÃO**

O algoritmo de PageRank considera também a probabilidade do usuário chegar na página diretamente, sem passar por outros sites, chamado de Damping Factor.

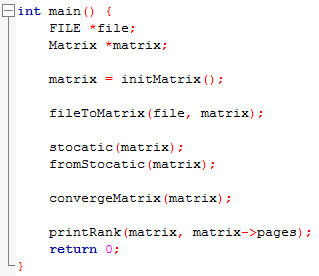
**ESTRUTURA DE DADOS**



A estrutura é bem simples e autoexplicativa, contém o número de páginas, e uma matriz de double que, inicialmente contém apenas 0’s ou 1’s e representa as relações entre as páginas. Ao longo da execução ela se torna a pontuação referente à cada página.

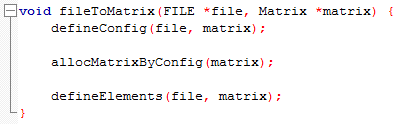
**FLUXO**

­­O fluxo do código é bem definido, no qual método executa apenas uma função, explicada no seu nome, como se pode ver no código da função main:

Primeiro alocamos a memória referente ao TAD (initMatrix()). Depois lemos todos os dados e organizamos na matriz (fileToMatrix()). E os passos seguintes são específicos do algoritmo de PageRank, transforma a matriz em estocática (stocatic()), monta a matriz de referência que se baseia na estocática e no damping fator (fromStocatic()), e depois realiza sucessivas multiplicações até o ponto de convergência (convergeMatrix()).

Será mostrado e explicado, na ordem de fluxo, cada função e o que ela faz.

**FUNÇÕES E PROCEDIMENTOS**



Essa função armazena os dados de configuração (número de páginas e damping factor) e, baseado no número de páginas, realiza a alocação dinâmica para, posteriormente, poder ler e guardar todas as relações entre as páginas (defineElements()).

O próximo passo é transformá-la em uma matriz estocática (onde todas as linhas somam 1). Para isso, antes precisamos converter todos as linhas que têm apenas elementos nulos, para uma linha de 1’s, é o trabalho da função

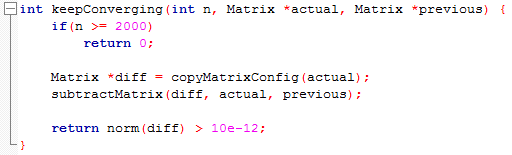
void normalize();

Essa função pega todos os índices das linhas que contém apenas elementos nulos (através da função int nullLineIndex()), e converte todos para um.

A função estocática, percorre todas as linhas e, para cada, calcula a soma dos elementos, e depois os converte para o novo número calculado:

|  |
| --- |
| total = totalInRow(row, pages);  changeOneTo(row, 1/total, pages); |

A função de convergir a matriz, realiza apenas uma sucessão de multiplicações da matriz por ela mesmo (através de um do while), enquanto a condição de convergência não estiver aceita.



A função de convergência verifica se o número de multiplicações for maior que 2000, ou se a norma da subtração da matriz atual com a anterior for menor que .

**ANÁLISE DE COMPLEXIDADE**

void allocMatrixByConfig(Matrix \*matrix);

Realiza duas operações O(1) e depois um loop executado N vezes realiza uma operação O(1), portanto: O(1) + n\*O(1) = **O(n)**.

void definePages(FILE \*file, Matrix \*matrix);

**&**

void defineDamping(FILE \*file, Matrix \*matrix);

Têm apenas uma atribuição, portanto: **O(1)**.

void fileToElements(FILE \*file, Matrix \*matrix);

Realiza leituras até o fim do arquivo. No melhor caso não há mais nenhuma leitura a ser feita, sendo **O(0)**. No pior caso é quando todas as páginas apontam para todas as páginas, ou seja, n\*n, portanto **O()**.

void fileToMatrix(FILE \*file, Matrix \*matrix);

Esta função contém todas as anteriores, portanto no melhor caso ela equivale à allocMatrixByConfig que é **O(n)**, e no pior equivale à fileToElements que é **O()**.

int nullLineIndex(Matrix \*matrix, int init);

Esta função tem dois loops aninhados, no qual o primeiro vai de init a n, e o segundo vai de 0 a n. Porém o segundo loop não é sempre executado completamente, pois quando encontra um elemento igual a 1, o primeiro loop incrementa um e o segundo volta a 0. Portanto, o melhor caso é quando há 1 no primeiro elemento de cada linha, sendo **(n – init) \* O(4)**, como depende de init, o melhor do melhor caso é quando init é igual a n, portanto **O(0)**, e o pior do melhor caso é quando init é 0, portanto **n\*O(4)** = **O(n)**. O pior caso é quando tudo é 0, mas apenas o último elemento de cada linha é 1, ficando **(n – init) \* n \* O(1)**, como depende de init, o melhor do pior caso é quando init é igual a n, portanto **O(0)**, e o pior do pior caso é quando init é 0, portanto **n\*n\*O(1)**, igual **O()**.

int normalize(Matrix \*matrix);

Esta função tem um loop dentro do while, que executa enquanto nullLineIndex não retorna -1, portanto seu melhor caso equivale ao “pior melhor” caso da função anterior, pois init vale 0 nesse caso, portanto **O(n)**. Dentro do while, o loop executa n vezes uma operação O(1), e chama a função nullLineIndex de novo. O pior caso é quando todas as linhas são nulas, levando a função nullLineIndex ser O(n). Portanto temos n \* (n + O(n)) = n \* (2\*O(n)) = **O()**.

void stocatic(Matrix \*matrix);

Esta função executa um loop n vezes, chamando duas funções que também executam o loop n vezes, portanto temos n\*(n + n) = 2\* = **O()**.

void fromStocatic(Matrix \*matrix);

Executa dois loops aninhados n vezes cada, com uma operação O(1), portanto: **O()**.

void printRank(Matrix \*matrix, int pages);

Possui apenas um loop que executa n vezes, portanto **O(n)**.

void multiplyMatrix(Matrix \*to, Matrix \*m1, Matrix \*m2);

Possui três loops aninhados para realizar a multiplicação das matrizes, portanto **O()**.

void subtractMatrix(Matrix \*to, Matrix \*m1, Matrix \*m2);

**&**

double norm(Matrix \*matrix);

Possui dois loops aninhados, portanto **O()**.

Matrix \*copyMatrix(Matrix \*from);

Possui uma chamada O(n), e depois dois loops aninhados, portanto: O(n) + O() = **O()**.

int keepConverging(int n, Matrix \*actual, Matrix \*previous);

No melhor caso é O(1), em todos os outros casos, realiza duas chamadas O(), portanto: **O()**.

Void keepConverging(Matrix \*matrix);

Tem um while que tem como condição de parada a função anterior. Dentro do while executa uma função O(). Portanto o melhor caso é O() + O(1), que é **O()**. O pior caso será 2000 \* ( O() + O() ) = **O()**.

**Programa Principal:**

O programa tem como ordem de complexidade a função que tem a maior complexidade, que no caso é a keepConverging(Matrix \*matrix), que é **O()**.

**CONCLUSÃO**

Obtive sucesso na implementação do trabalho ao obter os resultados esperados. A principal dificuldade encontrada foi modularizar o código de forma eficiente, de tal forma que cada função ficasse curta e realizando apenas uma função, porém o resultado foi satisfatório.