Trabalho Prático 1
Algoritmos I

Euller Saez Lage Silva

2019054501

Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte – MG – Brasil

eullersaez@ufmg.br

1. Modelagem computacional

O trabalho proposto consistia em desenvolver um sistema computacional capaz de resolver o problema da distribuição de vacinas a partir de um centro para postos de vacinação levando em consideração o alcance propiciado dada a variação da temperatura, para obter quantos e quais postos seriam alcançados, além de descobrir se em algum caminho da rota um posto seria alcançado por mais de uma vez, o que geraria ineficiência.

Desse modo, a modelagem mais simples e adequada para a resolução foi por meio de um grafo no qual se inseriu a princípio os centros de vacinação como sendo os primeiros vértices e posteriormente os postos até o último vértice. Após estabelecidas as devidas ligações de arestas entre os vértices, bastou realizar o caminhamento Depth-First Search (DFS), ou Busca em Profundidade, realizando algumas alterações em seu funcionamento para fazer com que o nível de profundidade com o qual o caminhamento pudesse ocorrer fosse limitado (por causa da variação da temperatura), além do reconhecimento de ciclos no caminhamento, ou seja, a identificação de que em um mesmo caminhamento de uma rota haveria a passagem por mais de uma vez em um posto.

2. Estrutura de dados

As estruturas de dados utilizadas neste trabalho prático consistiram basicamente na utilização de um grafo implementado por lista de adjacência, conforme uma das recomendações vistas em aula, e teve como base a implementação disponível neste vídeo[1], fazendo as adaptações necessárias. A escolha pelas listas de adjacência se deu pelo fato de que se ocupa menos espaço para o armazenamento de arestas entre os vértices, além de ser mais intuitivo implementar a Busca em Profundidade.

A classe Grafo possui um atributo inteiro *numero_vertices,* uma lista do tipo inteiro *lista_adjacencia*, uma variável booleana *repeticao*, responsável por armazenar a condição de haver algum ciclo entre os vértices atingíveis por um caminhamento no grafo e por fim um array do tipo inteiro *vetor_de_postos_alcancados*, responsável por armazenar se o vértice em uma dada posição foi acessado durante alguma execução da DFS para algum vértice.

[1] https://youtu.be/BwZvcq5K1wU

O construtor da classe recebe o número de vértices, inicia a lista de adjacência para o tamanho de vértices, deixa *repeticao* atribuída inicialmente ao valor falso e utiliza um laço do tipo for para zerar todos os elementos de *vetor_de_postos_alcancados*, do tamanho do número de vértices. Os outros métodos da classe são *adiciona_aresta*, *imprime_postos_alcancados* e *dfs*.

O *adiciona_aresta*, como o nome sugere é responsável por adicionar uma aresta do vértice A ao vértice B.

---

adiciona_aresta(a,b):

```
lista_adjacencia[a].pushback(b)
```

O *imprime_postos_alcancados* é responsável pela saída conforme a especificação. O fato das primeiras posições dos vértices serem ocupadas com centros faz com que a contagem e impressão dos postos tenha que levar isso em consideração.

---

imprime_postos_alcancados(nº centros):

```
numero alcançado =  0
Para cada elemento do vetor_de_postos_alcancados começando da posição nº centros
        Se o elemento for maior ou igual a 1 então
                incrementa numero alcançado
        fim-se
fim-para
Se numero alcançado for 0 então
        imprime 0\n
        imprime *\n
fim-se
Senão
        imprime numero alcançado
        Para cada elemento do vetor_de_postos_alcancados começando em nº centros
                Se o elemento for maior ou igual a 1 então
                        imprime posicao -  nº centros+1
                fim-se
        fim-para
        imprime \n
fim-senão
Se atributo repeticao for verdadeiro então
        imprime 1\n
```

fim-se
Senão
imprime 0\n
fim-senão

O *dfs* por sua vez realiza recursivamente o caminhamento da Busca em Profundidade visto em aula e disponível no livro Algorithm Design, com adaptações pois é preciso observar a profundidade na qual cada vértice é acessado para estabelecer um limite e uma condição de parada, além de checar caso algum ciclo seja identificado durante o caminho. Para isso, irá receber o vértice inicial V, a profundidade percorrida PP, a profundidade máxima PM e um array Explorados do tamanho de vértices do grafo inicialmente zerado para sempre marcar quando um vértice foi acessado ou todos os caminhos possíveis a partir dele já foram feitos.

---

dfs(V, PP, PM, Explorados):
Se PP > PM **então**
retorne
fim-se
Senão se Explorados[V] for -1 **então**
retorne
fim-senão se
Senão se Explorados[V] for 1 **então**
marque atributo repeticao como verdadeiro
fim-senão se

marque Explorados[V] = 1
marque vetor_de_postos_alcancados[V] = 1

Para cada vertice U que é conectado a V
invoque recursivamente dfs(U,PP+1,PM,Explorados)
fim-para

//marca-se como -1 quando todos os caminhos a partir desse vértice tiverem sido //percorridos.

marque Explorados[V] = -1

---

3. Instruções de compilação e execução

O programa foi desenvolvido na linguagem C++ por meio do Visual Studio Code para Windows da Microsoft Corporation, utilizando o Windows Subsystem for Linux (WSL) e compilado pelo compilador G++ da GNU Compiler Collection.

3.1 Compilação

Para compilar o programa primeiro é necessário abrir o diretório 2019054501_EULLER/src no terminal do linux e digitar o comando “make”.

3.2 Execução

Para executar o programa, após compilado, ainda dentro do mesmo diretório 2019054501_EULLER/src no terminal do linux, basta digitar o comando “./tp01”.

3.3 Limpeza de arquivos temporários

Para limpar todos os arquivos temporários gerados durante a compilação, ainda dentro do mesmo diretório 2019054501_EULLER/src no terminal do linux, basta digitar o comando “make clean”.

4. Análise de complexidade

A complexidade de tempo do programa como um todo será o número de vezes em que o método *dfs* será chamado. Como para cada centro c precisamos executá-lo, e em cada execução o vetor de vértices explorados precisa estar com todos elementos iguais a 0 inicialmente, precisaremos percorrer antes de cada chamada todos os v elementos deste vetor representando os vértices.
O método em si terá custo $O(v+e)$ onde v é o número de vértices e e o número de arestas presentes no grafo pois no seu pior caso, em que não se estabelece um limite de vértices a serem percorridos, terá que percorrer todo o grafo se ele for conexo. Assim, temos que

$$c\ (\ O(v) + O(v+e)\ ) = O(\mathrm{n})$$

e portanto a complexidade de tempo do programa é linear $O(\mathrm{n})$.

References

KLEINBERG, J., TARDOS, É. Algorithm Design. 1st. Edition. Pearson, March 16, 2005.