Trabalho Prático 1 – Algoritmos I

Logística de transporte das vacinas contra o covid-19

Gilliard Gabriel Rodrigues
Matrícula: 2019054609

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

gilliard2019@ufmg.br

1. Introdução

O problema proposto foi implementar um sistema que calculasse, entre todos os postos de vacinação (PVs), quais seriam aqueles que receberiam as vacinas contra o covid-19 em temperatura abaixo da temperatura máxima indicada pelo fabricante, sem que houvesse alterações na logística já utilizada pela Secretaria de Saúde, além de verificar se existe alguma rota que percorra o mesmo posto de vacinação mais de uma vez, a fim de melhorar a logística futuramente.

As vacinas são mantidas a uma temperatura de –90ºC nos centros de distribuição (CDs), e podem ser transportadas a uma temperatura máxima de -60ºC, seja X o incremento de temperatura a cada vez que um PV é percorrido, a quantidade de PVs que podem ser alcançados por um caminhão sem que a temperatura ultrapasse o limite é dada pela fórmula [-60-(-90)] /X, ou seja: 30/X.

Além disso, entende-se como rota, um caminho com direção definida que passa por cada PV uma única vez, até um PV final, do qual não é possível alcançar nenhum outro posto, mesmo os que já foram visitados no mesmo caminho.

Com base nessas especificações, será apresentada nesta documentação uma implementação para esse sistema utilizando o conceito de grafos direcionados visto em aula.

2. Implementação

O programa foi desenvolvido na linguagem C++, compilada pelo compilador G++ da GNU Compiler Collection.

2.1 Estrutura de Dados e Algoritmos

A estrutura de dados utilizada foi a de um grafo direcionado, usando listas de adjacências, onde os CDs e os PVs compõem os vértices e cada CD possui arestas para os PVs, assim como os PVs possuem arestas para outros PVs. Foram feitas algumas modificações na classe, como por exemplo a exclusão de alguns métodos que não seriam utilizados, como também a adição de métodos extras para servir aos propósitos do trabalho. O algoritmo escolhido para caminhar no grafo foi o da busca em profundidade (DFS), também com modificações conforme as necessidades do TP.

2.2 Classe e seus métodos

A fim de seguir as boas práticas de programação e implementar um código modularizado, o programa foi dividido em 3 arquivos: “grafo.h”, “grafo.cpp” e “main.cpp”.

A classe *Grafo* possui 8 atributos privados, são eles: *int qtd_CD* (armazena a quantidade de centros de distribuição), *int qtd_PV* (armazena a quantidade de postos de vacinação), *int num_vertices* (armazena o número total de vértices do grafo), *int incremento_temperatura* (armazena o valor do incremento da temperatura), *int num_viagens_caminhao* (armazena o número máximo de PVs que podem ser percorridos por cada caminhão), *int PVrepetido* (armazena 1 caso haja algum PV sendo percorrido mais de uma vez na mesma rota, ou 0, em caso contrário), *int *visitados* (um ponteiro para um *array* de PVs visitados) e *list<int> *adj* (um ponteiro para um array contendo as listas de adjacências).

A classe possui um construtor (público) que recebe como parâmetros a quantidade de CDs, a quantidade de PVs e o incremento da temperatura, atribuindo esses 3 parâmetros aos atributos da classe, além de calcular o número total de vértices (soma da quantidade de PVs e CDs), o número de viagens por caminhão (recebendo a chamada de uma função que será explicada posteriormente), inicializa o atributo *PVrepetido* como 0, além de alocar memória dinamicamente para os *arrays*.

Além dos atributos privados e do construtor, a classe Grafo possui 8 métodos públicos, que serão explicados separadamente a seguir.

O primeiro método chama-se *adicionaAresta* e trata-se de um método *void*, que recebe como parâmetros dois inteiros v e w, e adiciona w à lista de vértices adjacentes a v.

O segundo método chama-se *insereLigacoesImediatasCDs* e trata-se de um método *void*, sem parâmetros, que lê as C linhas contendo as ligações imediatas de cada CD através da função *getline* e adiciona ao grafo, chamando o método *adicionaAresta*. Para isso, ele utiliza um *while* dentro de um *for*. A condição de parada do *while* é o fim da linha e a do *for* é a quantidade de CDs.

O terceiro método chama-se *insereLigacoesImediatasPVs* e trata-se também de um método *void*, sem parâmetros, que lê as P linhas contendo as ligações imediatas de cada PV através da função *getline* e adiciona ao grafo, chamando o método *adicionaAresta*. Para isso, ele utiliza um *while* dentro de um *for*. A condição de parada do *while* é o fim da linha ou o valor 0 (que indica que um PV não tem uma aresta saindo para nenhum outro PV no grafo). Aqui o contador do *for* começa a partir da quantidade de CDs e vai até a penúltima posição do *array* de listas de adjacências.

O quarto método chama-se *calculaViagens*, que recebe como parâmetro o incremento da temperatura e retorna um inteiro representando o número máximo de PVs que o caminhão pode percorrer. Para isso, ele utiliza a fórmula 30/incremento da temperatura.

O quinto método chama-se *inicializaVisitados* e trata-se de um método *void*, sem parâmetros, que inicializa todas as posições do *array* de visitados com 0, representando que nenhum PV foi visitado inicialmente. Para isso, ele utiliza um for, cuja condição de parada é a quantidade de PVs.

O sexto método chama-se *navegaPeloGrafo* e trata-se de um método *void*, sem parâmetros, que chama a DFS modificada para os PVs vizinhos de cada CD, passando

como parâmetro o PV e a distância inicial como 1. Para isso ele utiliza um *for* dentro de outro *for*. A condição de parada do for externo é a quantidade de CDs, enquanto a condição de parada do *for* interno, que usa um *iterator*, é o fim de cada lista de adjacência.

O sétimo método chama-se *DFSmodificada* e trata-se de um método *void*, que recebe como parâmetro um vértice e um contador de distância (que inicialmente é sempre 1). Esse método trata-se da busca em profundidade, modificada conforme os interesses do TP, ou seja, ela caminha até alcançar o limite máximo de PVs que o caminhão pode percorrer e também verifica se algum PV foi visitado mais de uma vez na mesma rota. Inicialmente, há um *if* que verifica se o vértice já foi visitado (ou seja, se visitados[v-1] é igual a 1), se for verdadeiro, atribui 1 à variável *PVrepetido*. Em seguida, há um if que verifica se já não há mais caminhos a partir de v (ou seja, se visitados[v-1] é igual a 2), se for verdadeiro, retorna. Após isso, há outro *if* que verifica se o limite máximo de PVs que o caminhão pode percorrer já foi atingido (ou seja, se *distancia* é maior que *num_viagens_caminhao*), e em seguida, há um for que itera na lista de adjacência do PV passado como parâmetro, chamando recursivamente a DFS modificada e passando como parâmetro cada vizinho de PV, além de *distancia + 1*. No fim do método, é atribuído o valor 2 para a posição v-1 no *array* de visitados, representando que encerrou o caminho. Como tanto os CDs quanto os PVs estão no mesmo array de listas de adjacências, para acessar corretamente cada PV equivalente aos da entrada, é feito o seguinte cálculo: qtd_CD + v - 1, que representa a posição do PV "v" no array.

O oitavo método chama-se *imprimeResultado* e trata-se também de um método void, sem parâmetros, que imprime as saídas esperadas, ou seja, a quantidade de PVs alcançáveis, os PVs alcançáveis em ordem crescente pelos códigos e, por fim, 1 caso haja algum PV que é percorrido mais de uma vez na mesma rota, ou 0, caso contrário.

2.3 Estrutura do main

No *main()*, ocorrem as declarações de variáveis, além da leitura da primeira linha através da função *getline*, a instanciação de um objeto do tipo *Grafo*, seguido pelas chamadas dos métodos *insereLigacoesImediatasCDs*, *insereLigacoesImediatasPVs*, *inicializaVisitados*, *navegaPeloGrafo* e *imprimeResultado*.

2.5 Instruções de compilação e execução

Para compilar o trabalho, acesse o diretório do plano de dominação que deseja executar via terminal, e digite "*make"*. Em seguida, para executar o código, digite "*./tp01"*.

Logo após, na primeira linha, insira a quantidade de CDs, seguida pela quantidade de PVs e o incremento da temperatura, nas próximas C linhas insira as ligações imediatas de cada CD e, em seguida, nas próximas P linhas insira as ligações imediatas de cada PV.

Para ler a entrada de um arquivo, basta digitar "./tp01 < nomeDoArquivo.txt".

3. Análise de complexidade de tempo

Para chegar a uma conclusão sobre o custo de complexidade do programa, serão analisadas as chamadas da função *main(),* passo a passo. Seja "*c*" a quantidade de CDs e de "*n*" a quantidade de PVs.

A primeira chamada trata-se do método *insereLigacoesImediatasCDs*. Esse método possui um *for*, que itera *c* vezes, e dentro desse *for*, temos um *while* cuja condição de

parada é o fim da linha, que possui as ligações imediatas (arestas). Dentro do *while,* há uma chamada para o método *adicionaAresta*, como o custo para se adicionar um elemento no final da lista é O(1), a análise de complexidade será dada em função da quantidade de vezes em que o *for* e o *while* iteram. A quantidade de arestas saindo de cada CD pode variar entre 0 e *n*, como vamos analisar o pior caso, temos $c*n$. Logo, $c*n = O(c*n)$ para esse método.

A segunda chamada trata-se do método *insereLigacoesImediatasPVs*. Esse método é similar ao anterior, só que dessa vez temos que o *for* itera *n* vezes e o *while* itera até o final da linha. Dentro do *while,* há uma chamada para o método *adicionaAresta*, como o custo para se adicionar um elemento no final da lista é O(1), a análise de complexidade será dada em função da quantidade de vezes em que o *for* e o *while* iteram. Como a quantidade de arestas saindo de cada PV pode variar entre 0 e *n-1*, como vamos analisar o pior caso, temos então, $n(n-1) = n^2 - n$. Logo, $O(n^2)$ para esse método (no pior caso).

A terceira chamada trata-se do método *inicializaVisitados*, que possui apenas um for que itera *n* vezes e não chama nenhum outro método, então sua complexidade é O(*n*).

A quarta chamada trata-se do método *navegaPeloGrafo*. Esse método possui um *for*, que itera *c* vezes, e dentro desse *for* temos outro *for* que itera sobre cada lista de adjacência (conforme o contador do *for* externo), e chama a DFS modificada para cada PV vizinho. Antes de tudo, vamos falar sobre a complexidade da DFS: como visto em aula, ela possui um comportamento da ordem de O(*m*+*n*), onde *m* é a quantidade de arestas e *n* é a quantidade de vértices. Como vamos analisar o pior caso, teremos *n(n-1)* arestas, logo: $[n(n-1) + n] = (n^2 - n + n) = n^2$, então temos um pior caso de $O(n^2)$ para a DFS. No pior caso, o número de vezes em que o *for* interno do método *navegaPeloGrafo* itera é *n* (quando cada CD possui um vértice para todos os PVs), e o *for* externo, *c*. Logo, temos: $c*n * n^2 = c*n^3$. Enfim, temos $O(n^3)$ para esse método.

A quinta chamada trata-se do método *imprimeResultado*. Esse método possui um *for* que itera *n* vezes para contabilizar os PVs visitados e um *for* para imprimir os PVs. Como não são aninhados, temos que a ordem de complexidade do método como um todo é O(*n*) apenas.

Portanto, analisando todos esses métodos, temos que a complexidade de tempo assintótica do programa é $O(n^3)$ no pior caso.

4. Conclusão

Os pontos mais importantes para conseguir implementar o sistema foram, sem dúvidas, conseguir modelar o problema usando grafos, saber qual algoritmo de busca utilizar para caminhar no grafo e também identificar a presença de um ciclo, além de saber modificar a DFS de acordo com os requisitos do problema.

No decorrer desse trabalho, ficou evidente a importância de fazer testes para ter certeza de que o algoritmo estava funcionando de forma correta. Por fim, foi dada uma atenção especial ao processo de refatoração e modularização do código a fim de atender às boas práticas de programação e facilitar a identificação e correção de erros.

Em suma, foi possível praticar o uso de grafos para modelar problemas complementando o aprendizado adquirido durante as aulas.