Trabalho Prático 3: Depósitos de vacinas

Ricardo Avelar

Departamento de Ciência da Computação – Universidade Federal de Minas Gerais (UFMG) Belo Horizonte – MG – Brazil
ricardodiasmode@hotmail.com

1. Introdução

Esta documentação lida com o problema de desenvolver um script com a funcionalidade de realizar a construção de depósitos de vacinas para vilas. O problema gira em torno de fazer essa distribuição de forma mínima, ou seja, com o mínimo de depósitos possíveis, mas fazendo com que para toda trilha haja pelo menos um depósito.

Essa documentação tem como objetivo facilitar o entendimento do script construído com a finalidade de completar essa tarefa. Aqui serão explicados todos os conceitos sobre as funções do script e como foram tomadas as decisões para se resolver os problemas necessários.

Para resolver o problema citado, foi seguida uma abordagem de utilização de uma DFS num grafo para a primeira tarefa e para a segunda tarefa foram criados loops que visitam os vértices e sinalizam as arestas como visitadas, tratando cada ponto como um nó e cada trilha como uma aresta.

A seção 2 trata de uma visão sobre a implementação deste problema, tratando as decisões para a resolução dos problemas. Já na seção 3 é apresentado um sucinto passo-a-passo de como se executar o programa. A seção 4 faz a análise de complexidade do script. A seção 5 é uma prova de corretude do algoritmo da tarefa 2. A seção 6 é a avaliação experimental. A seção 7 é a conclusão e visão final do projeto.

2. Implementação

O código desenvolvido para solução desse problema está organizado em:

Main.cpp: Aqui ocorre a inicialização do programa. Primeiramente é lido o número de pontos e o número de trilhas. Depois da leitura das informações criamos o grafo de acordo com as arestas passadas. Depois disso é impresso o resultado no terminal.

Graph.h: Essa é a classe que representa o grafo. Aqui temos variáveis que representam cada característica e funcionalidade do grafo, sendo válido descrever dois métodos em especial, que são definidas como ‘int FindMinimumVertexCover()’ e ‘int FindAproxVertexCover()’. O primeiro retorna o mínimo de vértices necessários para cumprir a tarefa 2 e o segundo faz esse trabalho para a tarefa 2.

Graph.cpp: Contém todos os métodos da classe Graph, que são utilizados no algoritmo que nos dá a resposta. Abaixo há uma breve descrição daqueles métodos não tão intuitivos.

FindMinimumVertexCover: Esse método primeiramente inicializa um vetor de inteiros que vai nos dizer qual vértice foi visitado. Em seguida, utilizamos uma DFS para seguir um procedimento recursivo que ou exclui um nó da cobertura e inclui os seus vizinhos ou inclui o nó na cobertura pegando o mínimo das possibilidades(incluir ou excluir) dos seus vizinhos.

FindAproxVertexCover: Esse método primeiramente inicializa um array de booleans que nos diz qual vértice foi visitado. Em seguida, temos um loop que para cada vértice, verifica se suas arestas ainda não foram visitadas, e se uma não foi, define todas as suas arestas como visitadas.

Configuração utilizada para testar o programa:

- Sistema Operacional Windows 10;
- Linguagens C++;
- Compilador Mingw;
- Processador Ryzen 5 3600x e 16Gb RAM;

3. Instruções de compilação e execução

- Acesse o diretório do projeto (2019054960_RicardoDiasAvelar);
- Utilizando um terminal, execute o arquivo [Makefile] utilizando o seguinte comando: < make >;
- Com esse comando, devem ser criados os arquivos .o na pasta obj e o arquivo .exe na pasta bin;
- Proceda Utilizando o terminal para acessar o diretório bin, posicione os arquivos entrada.txt em qualquer pasta que queira utilizar e execute o arquivo [tp03.exe] utilizando o seguinte comando: < “tp03 <tarefa1 ou tarefa2> < caminho/para/trilha1.txt”> ;
- Dessa forma será impresso no prompt o relatório do processo;

4. Análise de complexidade

No main.cpp é iniciado o grafo com uma complexidade de tempo de O(n), onde n é o número de trilhas. Além disso, o array “ArrayASerImpresso” é alocado, fazendo com que a complexidade de espaço seja O(n).

void FindMinimumVertexCover - complexidade de tempo: essa função é baseada no algoritmo de uma DFS, e tem a mesma complexidade de O(n), com ‘n’ sendo o número de vértices.

void FindMinimumVertexCover - complexidade de espaço: essa função realiza todas as operações alocando apenas um vetor ‘Visited’ de tamanho Θ(n), com ‘n’ sendo o número de vértices.

void FindAproxVertexCover - complexidade de tempo: essa função realiza operações constantes, em tempo O(1). Além disso, há um ‘for’ que itera de ‘k=0’ a ‘n’, dentro de um ‘for’ que itera em todas as arestas do nó em questão, dentro de outro ‘for’ que itera em todos os nós. Esses três ‘for’ podem ser resumidos pela complexidade Θ(E*(n^2)/2), sendo ‘n’ o número de vértices e ‘E’ o número de arestas.

void FindAproxVertexCover - complexidade de espaço: essa função realiza todas as operações alocando apenas um vetor ‘Visited’ de tamanho Θ(n^2), com ‘n’ sendo o número de vértices.

5. Prova de corretude

1. Prova de corretude de “FindMinimumVertexCover”:

Precisamos provar que o algoritmo nos retorna uma cobertura de vértices mínima.

Após iniciar o vetor de vértices visitados, realizamos a DFS e utilizamos uma estratégia de programação dinâmica. Para cada vértice nesse vetor temos a primeira posição iniciada com 0 e a segunda iniciada com 1. A primeira posição representa o número de vértices necessários para cobrir o vértice atual e seus vizinhos, e a segunda é esse mesmo número, porém incluindo o vértice atual.

É notável, portanto, que chamando a DFS recursivamente e em cada passo adicionando o menor valor necessário para cobrir todos os vértices vizinhos dos seus vizinhos na primeira posição, e o menor valor necessário para cobrir todos os vértices vizinhos do vértice atual na segunda posição, a cada passo estamos verificando a resposta para o vértice atual, sem contar com o vértice anterior na DFS. Assim, quando chegamos no vértice de partida, temos a resposta para esse vértice, que nos dá a cobertura de vértice exata para o grafo.

Note que esse algoritmo irá entrar em um loop infinito caso você tente executá-lo para um grafo com ciclos, pois nunca encontrará o final da DFS.

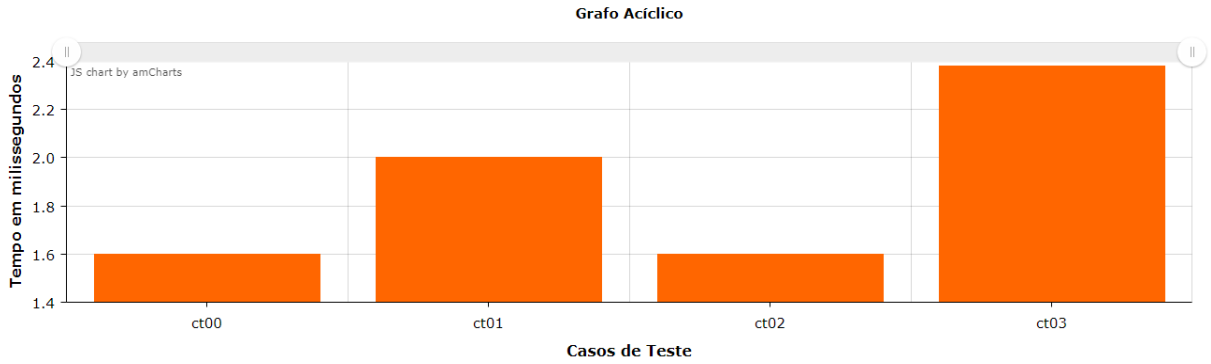
2. Prova de corretude de “FindAproxVertexCover”:

Precisamos provar que o algoritmo nos retorna uma cobertura de vértices, mas não necessariamente mínima.

Após iniciar o array de arestas visitadas, temos um loop que varre todos os vértices, e para cada vértice nós varremos seus vizinhos. Aqui, verificamos se essa aresta já foi visitada, e se não foi, definimos todas as arestas do primeiro vértice como visitadas - basicamente estamos dizendo que esse vértice faz parte da cobertura de vértice. Como esses loops varrem todas as arestas e verificam se todas foram visitadas, não é possível que no final do loop haja uma aresta que ainda não foi visitada. Portanto, o algoritmo retorna um número que representa necessariamente uma cobertura de vértice válida, mas não necessariamente mínima.

6. Avaliação experimental

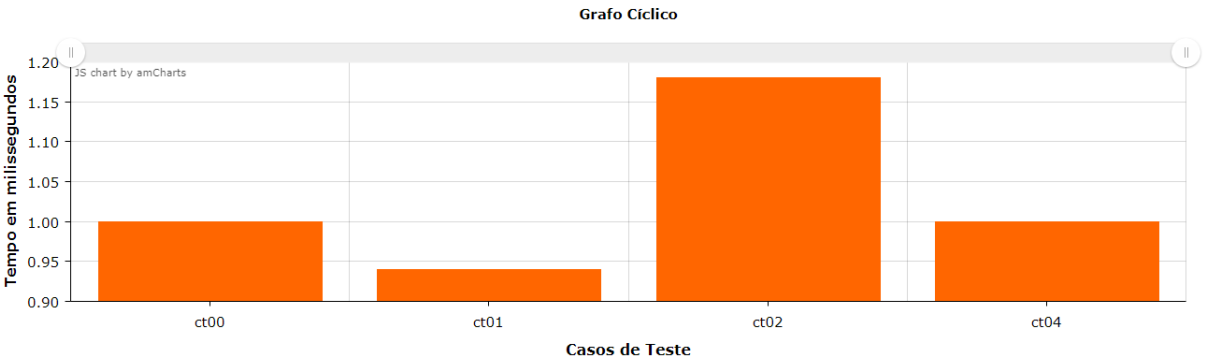
Abaixo temos um gráfico que no eixo Y está o tempo de execução em milissegundos de cada teste para o grafo **não** cíclico, e no eixo X o teste em questão. Os testes aqui referidos são específicos para o algoritmo de grafo não cíclico.



No grafo, o tempo de execução é a média de cinco execuções em cada teste. A média do resultado do gráfico é 1.895 e o desvio padrão é 0.37. Segue abaixo o resultado de cada teste:

Casos de Teste	Execução 01	Execução 2	Execução 3	Execução 4	Execução 5	Desvio Padrão:
ct00	4	1	1	1	1	1.34
ct01	6	1	1	1	1	1.23
ct02	4	1	1	1	1	1.34
ct03	6	2	2	0.9	1	2.1

Agora temos o gráfico para os testes cíclicos. Os testes aqui referidos são aqueles que foram criados exclusivamente para o algoritmo de grafo cíclico, ou seja, aqueles de nome “ct_ciclo<numero>”.



No grafo o tempo de execução é a média de cinco execuções em cada teste. A média do resultado do gráfico é 0,78 e o desvio padrão é 0.1. Segue abaixo o resultado de cada teste:

Casos de Teste	Execução 01	Execução 2	Execução 3	Execução 4	Execução 5	Desvio Padrão:
ct00	1	1	1	1	1	0
ct01	1	0.9	0.9	0.9	1	0.05
ct02	0.9	0.9	2	1.1	1	0.1
ct03	1	1	1	1	1	0

No teste ct_ciclo00 o algoritmo retornou que o número mínimo de vértices é 4, e a solução ótima é 3, portanto nesse caso o algoritmo retornou um número 1,3 maior que a solução ótima.

No teste ct_ciclo01 o algoritmo retornou que o número mínimo de vértices é 5, e a solução ótima é 3, portanto nesse caso o algoritmo retornou um número 1,6 maior que a solução ótima.

No teste ct_ciclo02 o algoritmo retornou que o número mínimo de vértices é 6, e a solução ótima é 5, portanto nesse caso o algoritmo retornou um número 1,2 maior que a solução ótima.

No teste ct_ciclo03 o algoritmo retornou que o número mínimo de vértices é 7, e a solução ótima é 5, portanto nesse caso o algoritmo retornou um número 1,4 maior que a solução ótima.

É importante ressaltar que em alguns casos os algoritmos executaram tão rápido que o programa retornou um tempo de 0 microssegundos de execução. Nesses casos a execução foi ignorada e o algoritmo foi executado novamente.

7. Conclusão

Este trabalho lidou com o problema de construção de depósitos de vacinas, no qual a abordagem utilizada para resolução foi o desenvolvimento de um script que faz a leitura de um arquivo txt com os dados das trilhas e pontos buscando fazer a distribuição mínima de depósitos nesses pontos para que toda trilha esteja coberta.

Com a solução adotada, pode-se verificar que a tentativa utilizada foi a criação de uma classe chamada Graph, e essa classe desempenha um papel especial como vimos no desenvolvimento da documentação.

Por meio da resolução desse trabalho, foi possível praticar os conceitos relacionados a grafos, com a necessidade de criação de algoritmos para resolução aproximada de um problema NP-Completo. Pela estratégia de resolução escolhida, além do algoritmo de DFS, pratiquei principalmente a estruturação de um algoritmo complexo e o trabalho com vetores.

Referências

Slides virtuais da disciplina de Algoritmos 1. Disponibilizado via moodle. Departamento de Ciência da Computação. Universidade Federal de Minas Gerais. Belo Horizonte.