Minimum Vertex Cover and Minimum Vertex Cover in Trees

Marcelo Mascarenhas Ribeiro de Araújo
2019110053

Departamento de Ciência da Computação – Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte – MG – Brazil

mascarenhassribeiro@gmail.com

1.Introdução:

Essa documentação tem a finalidade de apresentar diretrizes a serem seguidas na execução do programa, além de fornecer uma breve explicação da implementação realizada, em concomitância com uma análise de complexidade das principais funções utilizadas para a solução do problema proposto.

O projeto trata da implementação de um sistema responsável por calcular o número mínimo de depósitos de vacinas que devem ser construídos em um conjunto de pequenas vilas agrícolas dentro de uma grande área florestal, de modo que cada trilha, ou seja, caminho entre duas vilas seja incidente á um depósito. A solução para o problema pode ser encontrada de maneira exata caso não exista um ciclo entre os caminhos das vilas. Caso exista, um algoritmo que fornece uma solução aproximada, de maneira que seja até duas vezes pior que a solução ótima, foi implementado.

Nesse sentido, o projeto modela o problema computacional descrito e apresenta uma solução viável, que recebe os parâmetros através de um arquivo .txt, e realiza a leitura e a execução seguindo a lógica proposta na apresentação dos requerimentos do projeto.

Para informações mais detalhadas sobre o projeto, consulte as seções a seguir, sendo que:

- A seção 2 trata sobre as decisões tomadas para a resolução do problema proposto, bem como alguns detalhes de implementação, além do ambiente computacional que o trabalho foi desenvolvido.
- A seção 3 informa as instruções de compilação e execução.
- A seção 4 aborda uma breve análise de complexidade, de tempo e espaço, das principais funções e métodos utilizados no programa. A seção apresenta também provas de corretude dos algoritmos propostos.
- A seção 5 exibe algumas avaliações experimentais que foram utilizadas para avaliar o tempo de execução.
- A seção 6 conclui a documentação, fornecendo alguns detalhes adicionais sobre o processo de desenvolvimento e considerações em relação à solução proposta.

2.Implementação:

O programa foi desenvolvido e testado nas seguintes especificações:

-Linguagem: C++;

-Compilador: g++;

-OS e Versão do sistema: POP_OS!, versão 20.04.

-Configuração do computador: AMD Ryzen 5 3600, 16GB RAM.

2.1. Modelagem Computacional e Técnica de Resolução:

Para solucionar o problema proposto, a primeira observação é sobre a modelagem. As vilas foram representadas utilizando grafos. Após esse processo, pra cada tarefa, i.e, tarefa1 ou tarefa2, foram aplicados algoritmos diferentes. O problema proposto no trabalho é denominado de 'Minimum Vertex Cover'. O problema é da classe NP Completo, ou seja, a menos que P=NP, não existe um algoritmo polinomial que solucione qualquer instância do problema. Dessa forma, para a tarefa2, onde grafos com quaisquer propriedades podem ser passados, foi implementada uma solução baseada na heurística de aproximação descrito no livro 'Algoritmos: Teoria e Prática', escrito por Thomas H. Cormen. Essa heurística consiste basicamente na seleção de uma aresta aleatória do grafo, e a inclusão dos dois vértices que essa aresta incide no conjunto solução. Após esse processo, todas as arestas incidentes aos dois vértices que estão do conjunto solução são removidas, e o algoritmo executa os passos supracitados até não existirem mais arestas. Dessa forma, é encontrado uma aproximação que dá a resposta de modo que a mesma seja até duas vezes pior que a solução ótima.

Entretanto, apesar de não ser possível definir um algoritmo que encontre uma solução geral para qualquer instância do problema, em grafos sem ciclos, ou até mesmo em grafos bipartidos, é possível desenvolver um algoritmo que . Dessa forma, como na 'tarefa'1 foi garantido na proposta que não existiria ciclos nos caminhos das vilas na tarefa1, uma solução exata pôde ser implementada. A solução consiste basicamente em achar o 'minimum vertex cover' numa árvore(ou seja, um grafo sem ciclo), utilizando a metodologia da programação dinâmica. O algoritmo basicamente realiza uma recursão até as folhas da árvore utilizando um DFS, e calcula possíveis cenários(de inclusão e exclusão da solução) pra cada nó e suas subárvores. Dessa forma, caso um nó A seja excluído da solução, o número total de vertex cover se transforma na solução que inclui a soma da inclusão de todos os nós filhos de A. Caso o nó A seja incluído, a solução se transforma na soma do mínimo entre a solução de exclusão ou inclusão dos vértices filhos de A. Esse processo é calculado pra todos os vértices até chegar na raiz, dando, portanto, a solução desejada.

Dessa maneira, o fluxograma do programa segue da seguinte maneira: o grafo é construído de acordo com o .txt passado. Após isso, o programa detecta se o primeiro argumento é 'tarefa1' ou 'tarefa2'. Caso seja tarefa1, o programa executa o algoritmo que acha o 'vertex cover mínimo' na árvore e retorna a resposta. Caso seja tarefa2, o programa executa o algoritmo de aproximação e retorna a resposta em concomitância com os vértices inclusos na solução.

O trabalho está organizado em 5 diferentes arquivos, sendo eles:

Nota: Serão omitidos os parâmetros recebidos e os tipos retornados, salvo por quando forem citados na descrição, para fins de simplificação. Para mais detalhes, ou em caso de dúvidas, consulte o código que acompanha a documentação.

Graph.cpp/Graph.hpp: Classe base que possui a lista de adjacências que é utilizada para representar o grafo. A classe possui também os dois métodos que retornam a resposta desejada para ambas as tarefas.

Functions.cpp/Functions.cpp: Classe auxiliar, que contém um método para leitura do arquivo e preenchimento do grafo. Apesar das arestas serem passadas de maneira direcionada no .txt, a função presente na classe adiciona uma aresta nos dois sentidos, tornando o grafo não direcionado.
Ex: 3 0 é interpretado como sendo uma aresta que sai do vértice 3 e incide no vértice 0, e uma aresta que sai do vértice 0 e incide no vértice 3.

Main.cpp: Arquivo responsável por criar as outras classes, chamadas de métodos das mesmas, e contemplar mensagens de erro.

2.2. Estruturas de Dados Utilizadas:

Durante o desenvolvimento do trabalho, a única estrutura de dados utilizada foram os vectors, para representar os grafos, criar vetores de checagem auxiliar e vetores de solução. A classe dos vectors é uma classe muito versátil, e pode ser utilizada em diversos cenários sem grandes drawbacks.

2.3. Pseudocódigos:

Segue abaixo o pseudocódigo de ambos os algoritmos principais implementados.

- **Algoritmo de aproximação:**

```
begin:
        Create two sets E and S.
        Insert all edges of the graph in E.

        while E is not empty:
                Pick a random element of E.
                Insert both vertices incident to the selected edge in S.
                Remove all edges in E of the vertices that were inserted in S.
        endWhile
End
```

- **Algoritmo ‘Minimum Vertex Cover in tree’**

```
begin:
        Be G the number of vertices in the graph.

        Create an auxiliary vector<vector<int>> AS(G, vector<int>(2)), to hold the results of the inclusion or
        exclusion for each node and its children of the subproblem’s solution.

        For each item in AS:
          AS[0] = 0
          AS[1] = 1
        endFor

        Be Z the ID of some node in the graph.
        dfs(AS, Z, -1)
        answer = min(AS[Z][0], AS[Z][1])
end

dfs(AS, current_node, parent_node):
        for children in current_node:
          if id_children != id_parent_node:
             dfs(AS, children, id_current_node)
             AS[id_parent_node][0] += AS[id_children][1]
             AS[id_parent_node][1] += min(AS[id_children][0], AS[id_children][1])
        endFor
end dfs
```

3. Instruções de compilação e execução:

1. Realize a extração do zip para um diretório de sua preferência.

2. Acesse o diretório onde está o makefile, abra o terminal no diretório, e execute o comando make (Caso o comando não seja reconhecido no Windows, é necessário instalá-lo, através do pacote “Chocolatey”). Um executável com o nome “tp03” irá ser criado na pasta do makefile.

3. Abra o terminal.

No Windows:

- Execute o programa digitando o nome do executável, em concomitância com o tipo de tarefa(tarefa1 ou tarefa2), junto com o caminho completo do arquivo que possui as informações das vilas, com aspas. Veja o exemplo:

tp03 tarefa1 “C:/Users/Marcelo/OneDrive/Documentos/Faculdade Stuff/TP3_ALG/input.txt”

No Linux:

- Execute o programa digitando o nome do executável precedido por ‘./’, em concomitância com o tipo de tarefa(tarefa1 ou tarefa2), junto com o caminho completo do arquivo que possui as informações das vilas, com aspas. Veja o exemplo:

./tp03 tarefa2 /home/mmra/Documents/Faculdade/Algoritmos_I/Trabalho/input.txt

4. Análise de Complexidade:

Segue uma breve análise de tempo e espaço das principais funções utilizadas no programa:

```
void Functions::readInput(Graph &grafo)
```

Complexidade de tempo: A função realiza operações constantes, e lê um arquivo, através da entrada padrão, de n linhas. Tendo isso em vista, podemos classificar o custo assintótico de tempo dessa função como **O(n).**

Complexidade de espaço: A função basicamente preenche o grafo, que utiliza a representação através de ‘listas de adjacência’, com o número de vértices e arestas passadas na entrada. Dessa forma, podemos concluir que o custo assintótica de espaço é **O(|V|+|E|).**

```
int Graph::minNumVill()
```

Complexidade de tempo: A função realiza algumas operações de custo constante, além de um DFS que percorre uma árvore. É sabido que um DFS aplicado em uma árvore tem o custo de **O(|V|),** determinando, portanto, a complexidade de tempo dessa função.

Complexidade de espaço: *A* função cria um vetor auxiliar do tamanho do número de vértices do grafo para armazenar o resultado do vertex set para cada vértice e seus filhos, além do espaço na pilha para as chamadas da função. Dessa forma, possuindo complexidade de espaço **O(|V|).**

```
std::vector<int> Graph::vertexApprox()
```

Complexidade de tempo: A função de aproximação percorre todos os vértices num loop e passa por quase todas as arestas do grafo, seguindo a lógica descrita na descrição do algoritmo. Dessa forma, a complexidade de tempo do mesmo pode ser descrita como **O(|V|+|E|).**

Complexidade de espaço: *A* função cria um vetor auxiliar que guarda o número da solução aproximada e os vértices do conjunto solução, além de um vetor que checa se um determinado vértice foi ou não checado. Assim, a complexidade de espaço pode ser descrita como **O(|V|).**

```
int main(int argc, char* argv[])
```

Complexidade de tempo: O main realiza algumas operações constantes, e como sua principal funções é criar classes e chamar outros métodos, ele herda a complexidade da maior função presente. Dessa maneira, a complexidade do mesmo pode ser descrita em **O(|V|+|E|).**

Complexidade de espaço: A complexidade de espaço também é herdada do método que possui a maior complexidade, nesse caso temos que **O(|V|+|E|)** para construir e alocar o grafo.

4.1. Prova de corretude:

Iremos provar a corretude dos dois principais algoritmos utilizados no programa.

- **Algoritmo de aproximação:**

Vamos provar que o algoritmo sempre retorna um ‘Vertex Cover’. É facil perceber pelo modo de execução do algoritmo que durante o processamento das arestas, sempre é escolhido um conjunto maximal de arestas para garantir que a solução sempre seja válida em qualquer grafo. Dessa forma, sabemos que o algoritmo retorna um conjunto de arestas M de modo que cada vértice do grafo seja incidente a alguma aresta desse conjunto. Suponha que M não contemple todas as arestas necessárias para satisfazer o enunciado. Se isso ocorre, é por que ainda existe alguma aresta que pertence ao grafo e que ainda não foi processada, e portanto, o algoritmo não finalizou. ∎

- **Algoritmo de ‘Minimum vertex cover in trees’:**

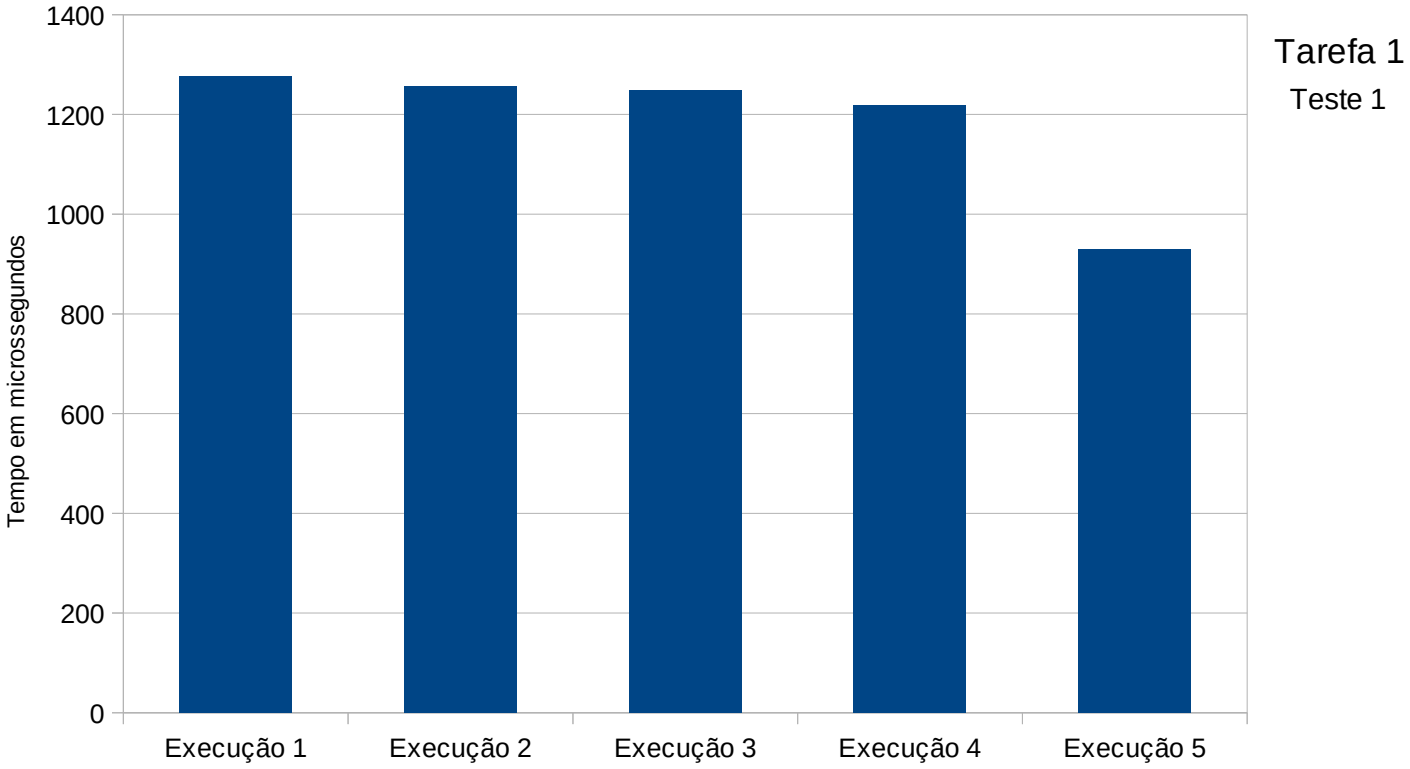
Prova por indução;

Caso base: Considere uma árvore com 2 vértices e uma aresta. Primeiro é escolhido um vértice raiz e um vértice filho. Após isso, são consideradas as duas possibilidades levadas em conta pelo algoritmo: ou o vértice filho está na solução ou ele não está na solução. Se ele não estiver, o vértice raiz está, resultando em 1 vértice como resposta. Se ele estiver, o vértice raiz não está, resultando também em 1 vértice como resposta

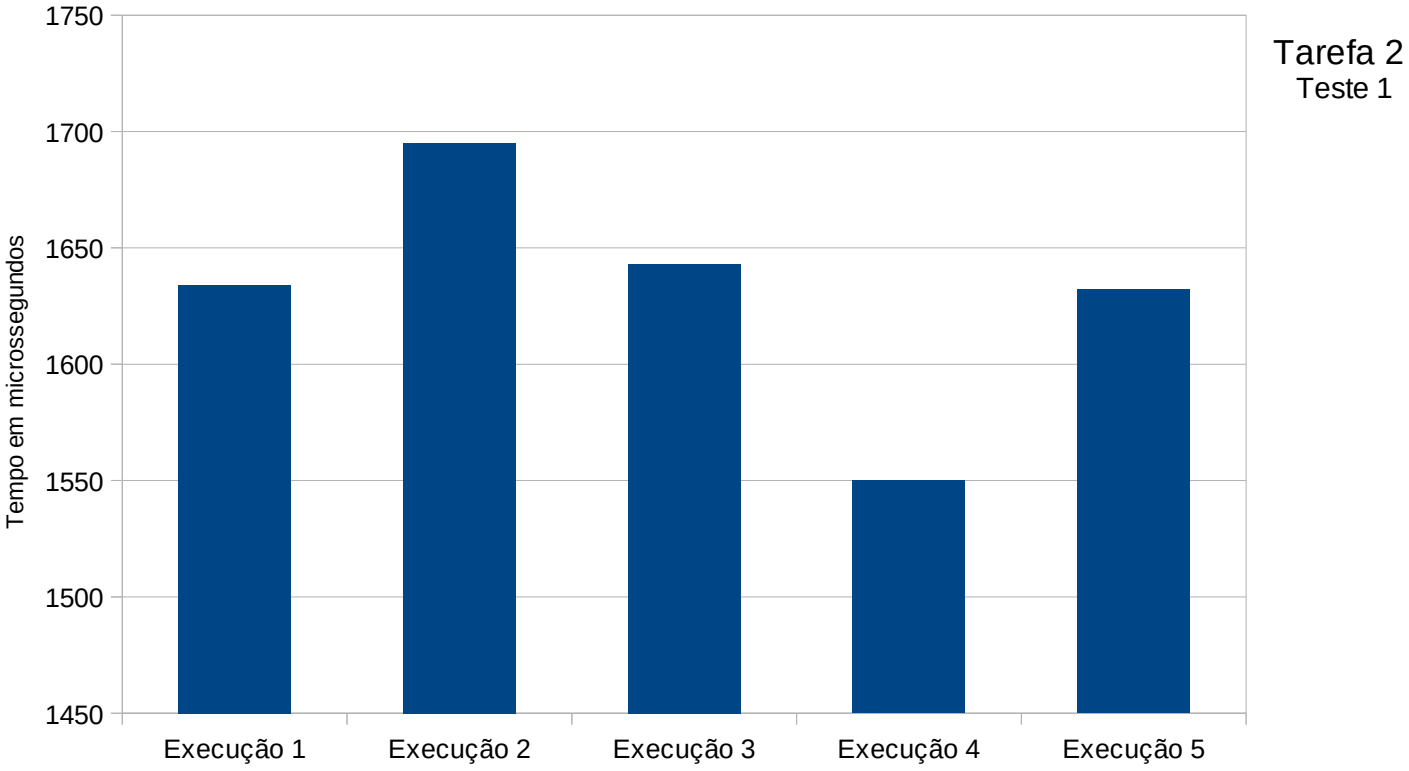
Hipótese indutiva: Considere que o problema seja solúvel para uma árvore com n vértices e n-1 arestas. Mostraremos que o problema também é solúvel para uma árvore A com n+1 vértices e n arestas. Fixe o vértice adicionado new_root, como raiz, e acrescente a aresta adicional à raiz da árvore anterior, root, que havia solução(com n vértices e n-1 arestas). Pela hipótese indutiva, A é solúvel, e portanto possui um número mínimo de ‘Vertex Cover’. Assim, podemos resolver o problema através de duas hipóteses. A primeira é que new_root não está incluso no conjunto solução da nova árvore, e portanto o número mínimo do ‘Vertex Cover’ é igual ao da resposta da árvore A. A segunda é que new_root está incluso, e a resposta passa a ser 1+min(min(root está incluso, root não está incluso)), como indica a equação de solução do problema. Como é possível decidir o min(root está incluso, root não está incluso), já que, fundamentalmente é a operação que leva a resposta, independente do cenário, ao adicionar um novo vértice, é sempre possível encontrar uma solução. ∎

5. Avaliação Experimental:

A fim de medir o tempo de execução do algoritmo, foi realizado um teste na tarefa1 com o pior caso passado pela equipe de monitoria da UFMG, que contém 500 vértices e 499 arestas. O tempo de todos os testes foi medido com o auxílio da biblioteca ‘chrono’, da linguagem C++.



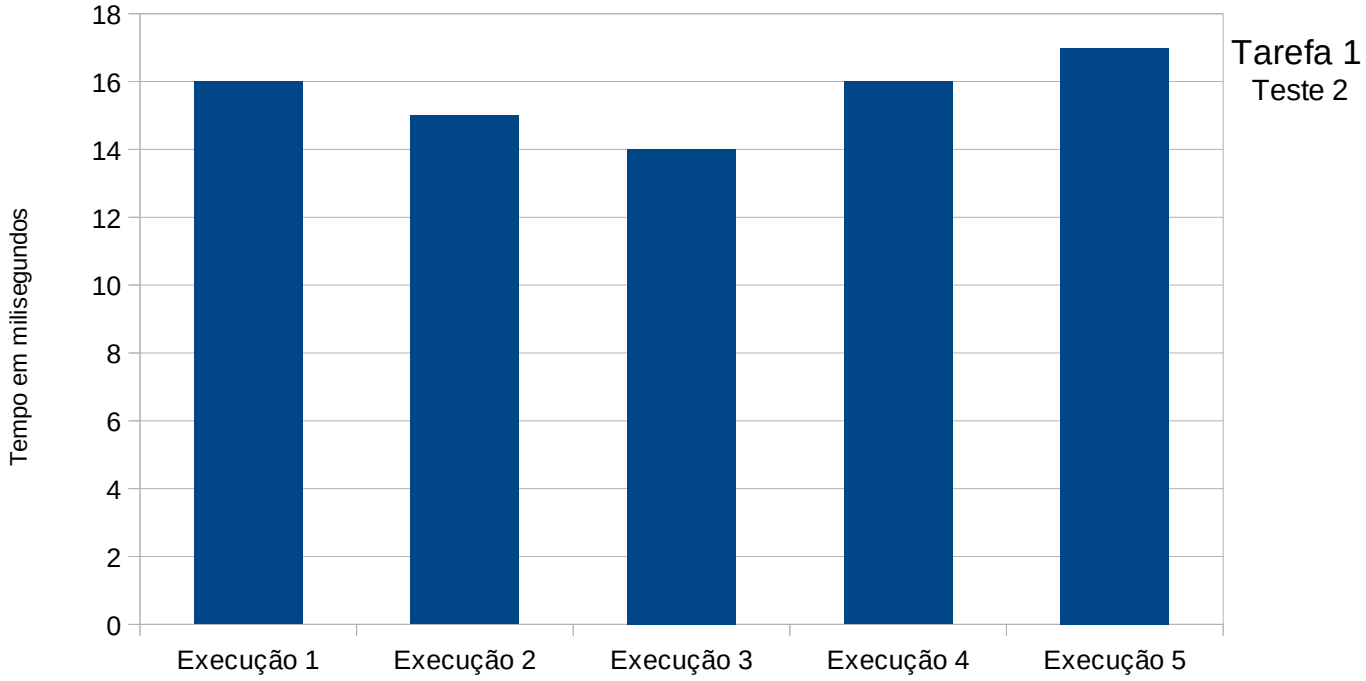
Média: 1186 microssegundos. **Desvio Padrão:** 145.127 microssegundos
O mesmo teste foi feito com a tarefa2, segue abaixo o gráfico:



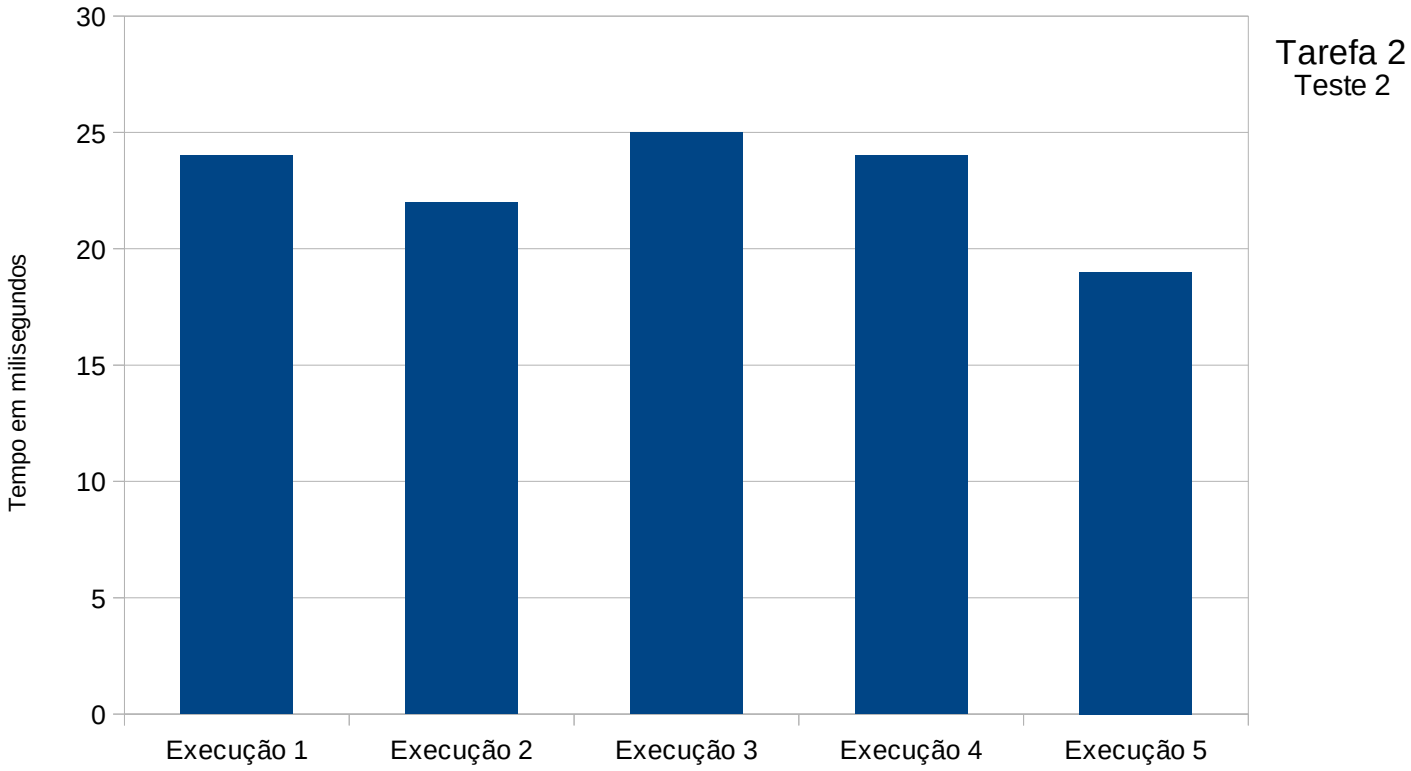
Média: 1630.8 microssegundos. **Desvio Padrão:** 51.98 microssegundos

Como podemos analisar, a tarefa2 gasta um pouco mais de tempo na execução em um mesmo arquivo, o que é esperado devido à sua complexidade ser O(|V|+|E|) contra a complexidade de O(|V|) da tarefa1.

Além disso, foi feito um teste adicional contendo um grafo sem ciclos, criados pelo próprio autor, que contém 10.000 (dez mil) vértices e 10.000 (dez mil) arestas. Segue os gráficos com o tempo de execução em ambas as tarefas.



Média: 15.6 milissegundos. **Desvio Padrão:** 1.14 milissegundos.

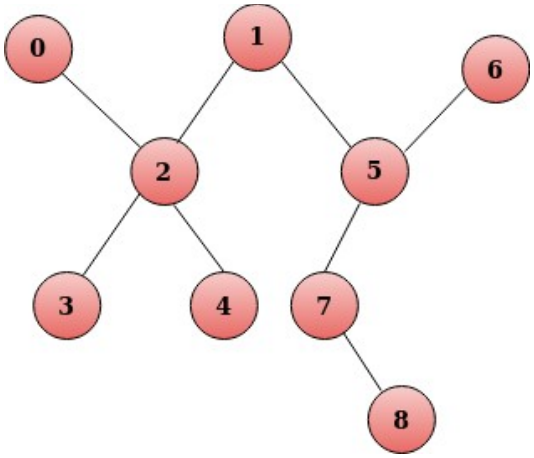


Média: 22 milissegundos. **Desvio Padrão:** 2 milissengundos.

Além disso, testes foram realizados com os arquivos passados pela equipe de monitoria da UFMG para garantir que a tarefa 2 retornava uma solução até duas vezes pior em relação á solução ótima. Observe a tabela:

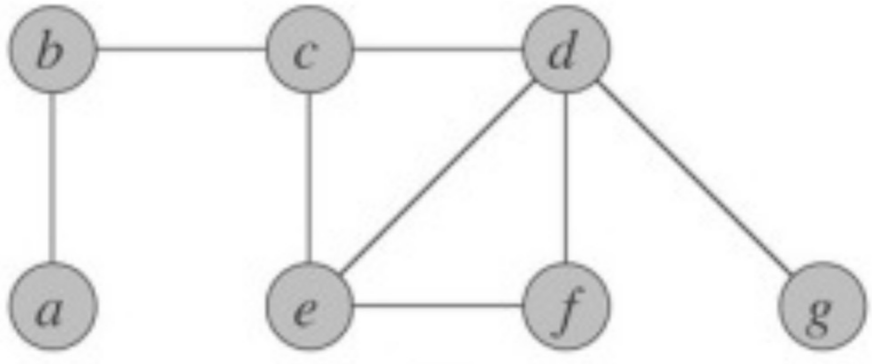
Arquivos de Entrada	Tarefa 1	Tarefa 2
CT00.txt	5	10
CT01.txt	8	14
CT02.txt	15	30
CT03.txt	165	320

Além disso, alguns grafos adicionais foram criados a fim de testar o programa desenvolvido. Segue alguns exemplos:

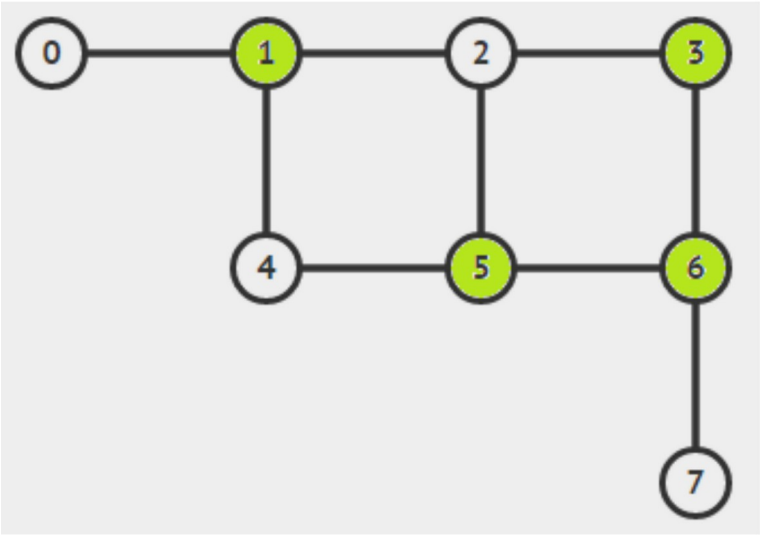


A solução ótima pra esse problema seriam 3 vértices (2, 5, 7). Quando rodados na tarefa2, a solução retorna 6, sendo os vértices(‘0,1,2,5,7,8’) parte do conjunto solução.

Outro exemplo é o seguinte grafo retirado do livro ‘Introduction to Algorithms’, escrito pelo Cormen. Durante a execução, a resposta do programa desenvolvido foi a mesma que estava prevista no livro, que eram 6 vértices (‘a, b, c, d, e, f’). A resposta ótima consiste em apenas 3 vértices, sendo eles ‘b, d, e’.



Segue um outro exemplo de grafo na tarefa 2:



Colorido de verde está a solução ótima do problema. Quando rodamos o algoritmo, temos o resultado como sendo 6 (vértices ‘0,1,2,3,5,6’).

6. Conclusão:

Considerando todas as informações supracitadas, o problema apresentado foi resolvido de maneira satisfatória. Não foram detectados erros nos testes realizados, seja de execução – considerando tanto os criados pelo desenvolvedor, quanto os passados pela equipe de monitoria -, quanto em testes de memórias – o programa foi submetido à testes com a ferramenta ‘Valgrind’ antes da entrega -. Além disso, a resolução se demonstrou extremamente modularizada e não tão custosa, apesar de provavelmente existirem projetos com uma eficiência maior.

Referências:

Chaimowicz, L. and Prates, R. (2020). Modelos e Requerimentos de Documentações, fornecido na disciplina de Estruturas de Dados. Disponibilizado via Moodle. Departamento de Ciência da Computação. Universidade Federal de Minas Gerais. Belo Horizonte.

Cormen; Leiserson; Riverst; Stein (2009). Introduction to Algorithms (3 ed.). Cambridge, ISBN.