Documentação Trabalho Prático 3 - Algoritmos 1

Guilherme Luiz Lara Silva
Departamento de Ciência da Computação - Universidade Federal de Minas Gerais
Belo Horizonte - MG - Brazil
guilhermells@ufmg.br

Apresentação

O código resolve o problema da tarefa 1 e tarefa 2 apresentados no documento de especificações, que é descobrir a menor quantidade possível de depósitos de vacina dado em grafos sem ciclos e com ciclos.

Modelagem Computacional

Classes: Graph.
Estruturas de dados: std::list.
Algoritmos: Depth First Search e Aproximação em grafos cíclicos

Sobre a arquitetura e classes:
Minha ideia para esse TP era arquitetar algo mais direto com o que foi solicitado, não vi a necessidade de criar classes abstratas e métodos genéricos. Os problemas solicitados são conhecidos no meio da computação como problemas de cobertura de vértices. Depois de algum estudo, percebi que para a tarefa 1 seria necessário usar uma busca em profundidade para marcar a quantidade mínima de vértices que seriam necessários para criar uma cobertura mínima. Já para a tarefa 2, em que precisamos de uma solução até duas vezes pior, foi necessário criar um algoritmo que exclui as arestas adjacentes aos vértices inicialmente escolhidos.
A única classe necessária para esse TP foi a classe *Graph,* que possui todos os métodos necessários para percorrer, adicionar nós e realizar operações no grafo em questão.

Sobre Depth First Search:

PSEUDO-DFS

```
dfs(node v):
    visited[v] = true;
    for item in adjMatrix[v]:
        if item is not visited:
            dfs(item)
```

Sobre Aproximação em Grafos Cíclicos:

PSEUDO-Aproximação em Grafos Cíclicos

```
findApproximateAddition():
  for fromVertex in adjMatrix
    for item in adjMatrix[fromVertex]:
        visited[fromVertex] = true
        visited[item] = true
        selectedVertices[fromVertex] = true
        selectedVertices[item] = true
            for searchedVertex in adjMatrix[item]:
                if item == fromVertex:
                    visited[searcedVertex] = true
```

Sobre as estruturas de dados:

Foi utilizada uma lista de adjacência para armazenar os dados lidos do arquivo. Essa lista foi montada usando uma std::list em que seus elementos eram vetores.
Para auxiliar no algoritmo de DFS e no algoritmo de Aproximação de Grafos Cíclicos, foi utilizado um vetor de booleanos para marcar quais vértices haviam sido selecionados e outro para marcar quais vértices haviam sido visitados.

Descrição da Solução

Sobre responsabilidades:

A função main foi a responsável por ler parte da entrada, criar e adicionar os vértices do grafo e para realizar as chamadas necessárias da classe Graph. Já a classe Graph era responsável por executar a busca em profundidade, inicializar vetores, o grafo e achar a quantidade de vértices selecionados.

Sobre o fluxo de execução macro:

1. O arquivo txt deve ser lido, e a partir dele será criado o grafo que representa as trilhas entre as vilas.
2. Caso o argumento 1 do programa seja igual à “tarefa1”, a classe Graph chama a função *findMinimalWarehouseAddition*, que é responsável por rodar o algoritmo de DFS, que marca os índices dos vértices que devem ser depósitos e depois executa uma função que encontra quantos vértices foram marcados.
3. Caso o argumento 1 do programa seja igual à “tarefa2”, a classe Graph chama a função *findApproximateMinimalWarehouseAddition,* que é responsável por rodar o algoritmo de Aproximação em Grafos Cíclicos, que marca os índices dos vértices que devem ser depósitos.

Análise de complexidade de espaço e tempo assintótica

Essa análise será iniciada a partir dos métodos chamados diretamente na main e, portanto, possuirá ramificações de funções que chamam outras funções.

1. findMinimalWarehouseAddition: essa função inicializa os vetores de vértices visitados e de vértices selecionados**.** Ela também chama a dfs para os vértices que não foram visitados e por fim retorna a quantidade de vértices selecionados. Possui complexidade de tempo de **O(V+E)** e de espaço de O(v).
    a. initializeBool: essa função inicializa um vetor booleano recebido por parâmetro. Essa inicialização se dá através de um loop que atribui false para todas as posições do vetor. Possui complexidade de tempo de **O(n)** e de espaço de **O(1)**.

    b. dfs: Essa função é uma busca em profundidade que encontra os elementos necessários para que tenhamos uma cobertura mínima dos vértices dado o grafo em questão. Os vértices que fazem parte desse conjunto de elementos que compõem a cobertura mínima são marcados como true em um vetor de adjacência chamado vertices (custo **O(1)**). Por se tratar de uma busca em profundidade, a complexidade dela é **O(V+E)**, tal que V é número de vértices e E o número de nós. A complexidade de espaço é **O(V).**
    c. getAmountOfSelectedVertices: essa função percorre um vetor e incrementa um contador quando o valor do elemento iterado é true. Por possuir apenas um loop, tem complexidade **O(n)**. A complexidade de espaço é **O(1)**.
2. findApproximateMinimalWarehouseAddition: essa função inicializa os vetores de vértices visitados e de vértices selecionados. Ela também faz a procura na lista de adjacência para acharmos os depósitos necessários para um dado grafo. Por fim ela retorna a quantidade de depósitos. Por utilizar de 3 loops aninhados, todos para iterar sobre a lista de adjacência, possui complexidade de tempo de O(n * m * m), ou seja **O(nm²)**, tal que *n* representa a quantidade de vilas e *m* representa as vilas adjacentes. Essa função possui complexidade de espaço de **O(n),** pois cria dois vetores.

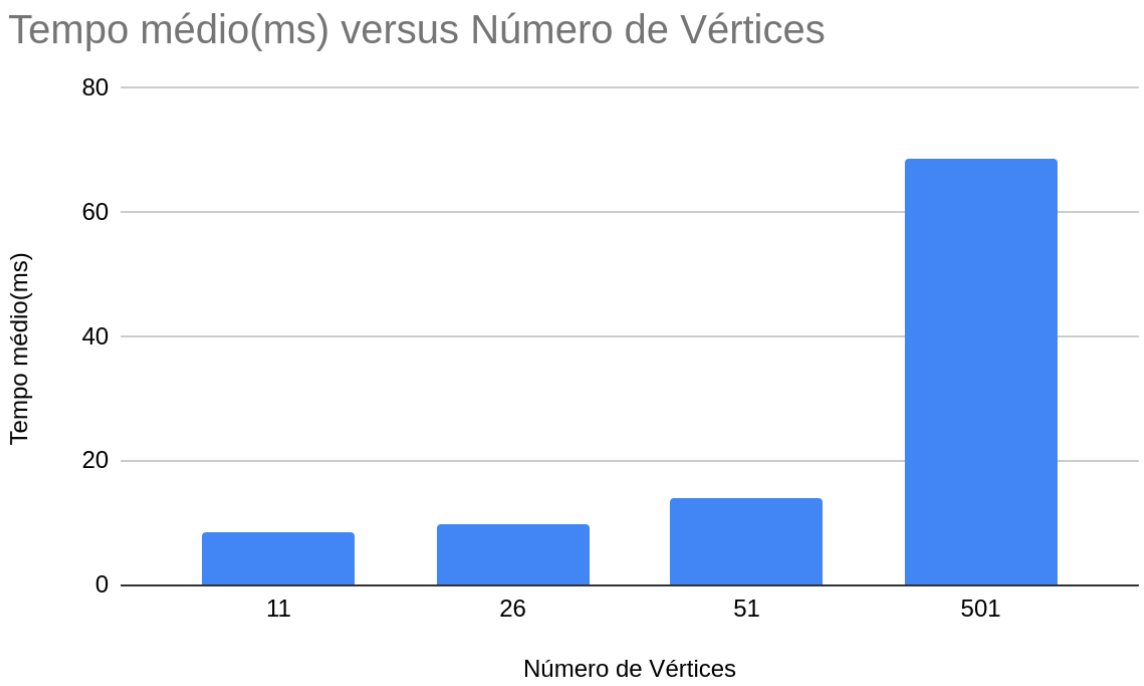
Portanto, após a análise detalhada, podemos concluir que a complexidade de tempo do código como um todo é para a tarefa 1 é de **O(V+E)** e para a tarefa 2 é de **O(nm²).** No agregado, temos que a complexidade de tempo geral do programa é **O(nm²).**

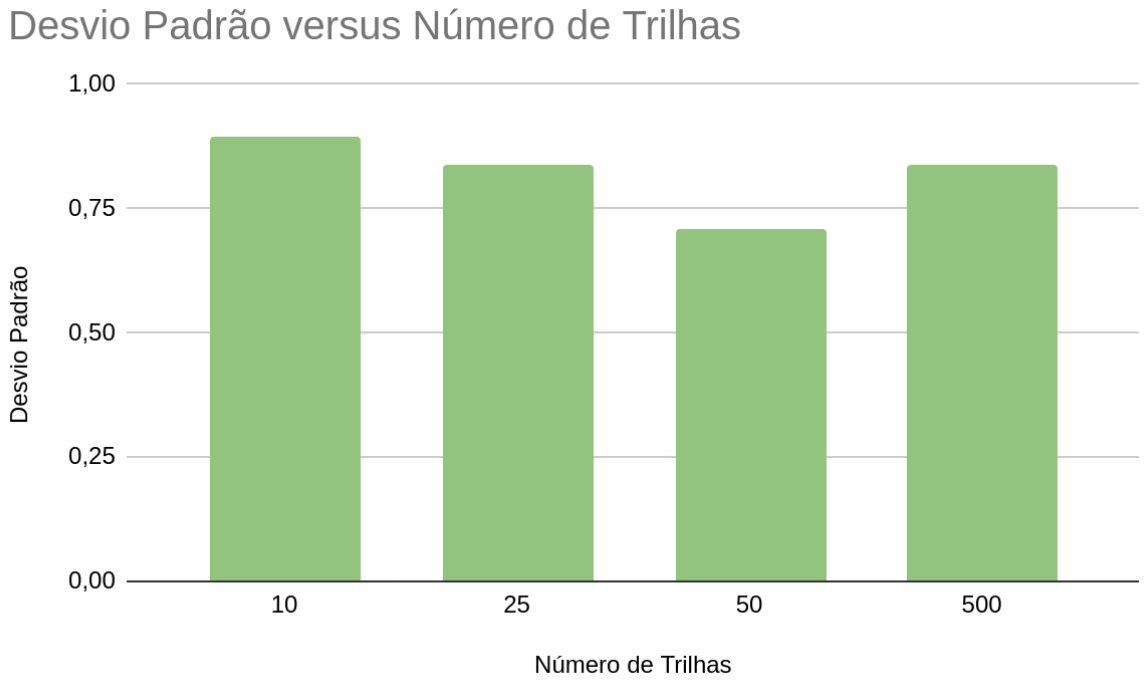
Avaliação Experimental

Nessa seção irei inserir alguns gráficos relacionados à variação do tempo de execução versus número de vértices e de trilhas. Vale ressaltar que o hardware do computador segue as seguintes especificações:

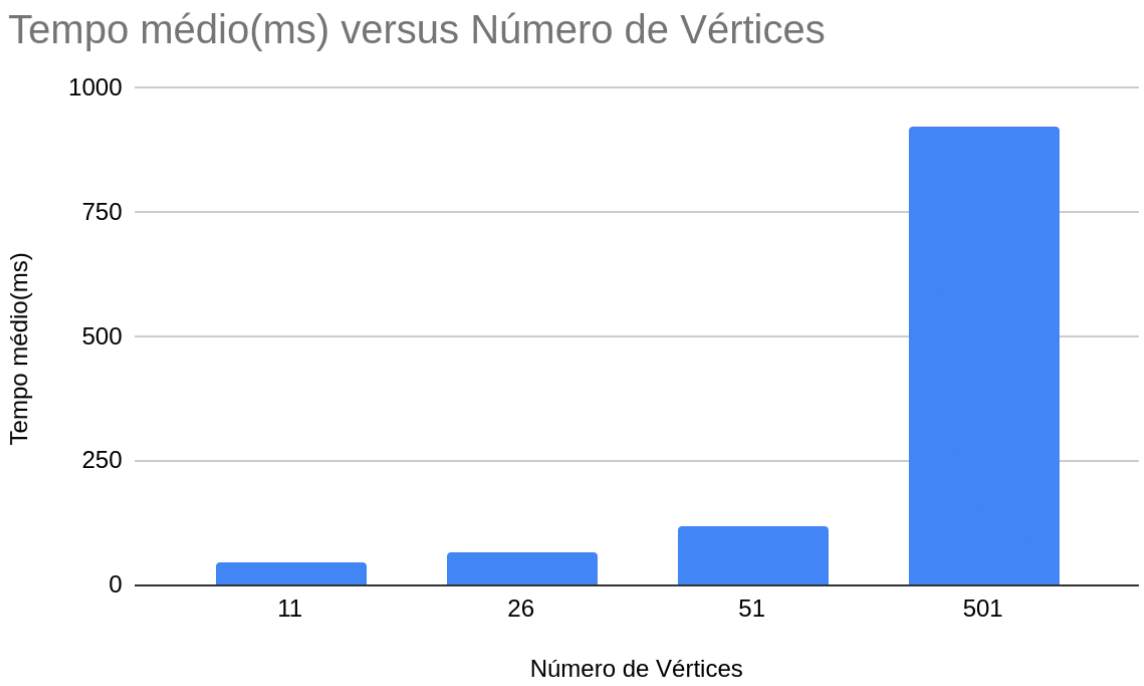
- Processador Ryzen 5 3500x
- 16GB de memória RAM
- SSD SATA 530 MB/s de leitura e 310 MB/s de gravação
- Ubuntu 20.04

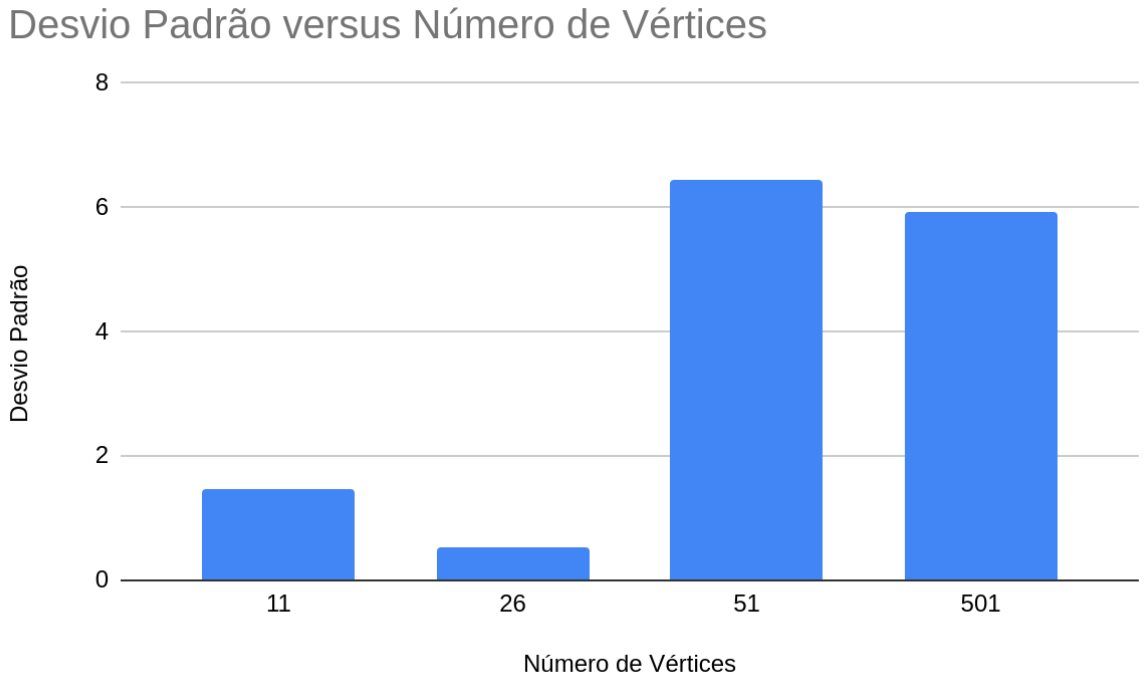
Gráficos para a tarefa 1





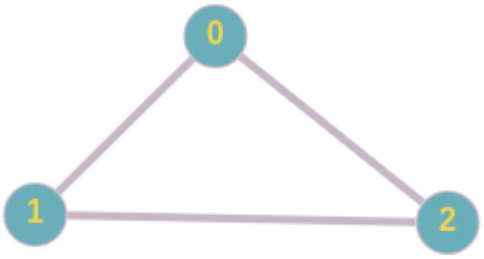
Gráficos para tarefa 2





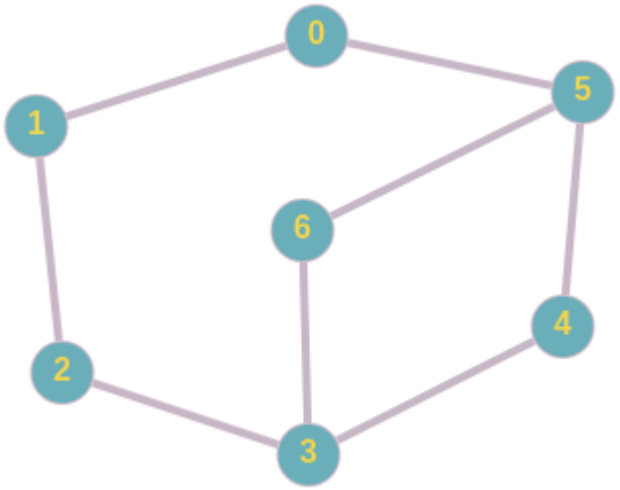
Casos de teste para grafos com ciclos

CT04



Saída esperada: 2
Saída algortimo: 2

CT05



Saída esperada: 4
Saída recebida: 4

CT06

O grafo do CT06 é igual ao CT02, porém a última linha do arquivo foi modificada. O par (45, 18) foi trocado pelo par (45, 32) para que um ciclo fosse criado. Mesmo com esse ciclo, a saída esperada se mantém igual ao do CT02, ou seja 15. Porém, nosso algoritmo retornou 24.

Saída esperada: 15
Saída recebida: 24

Heurística e Prova de Corretude Algoritmo de aproximação em grafos com ciclos

Para a solução da tarefa 2, deve-se selecionar inicialmente um vértice arbitrário e marcar seus vértices adjacentes como visitados e selecionados ao mesmo tempo. Após isso, selecionamos o próximo vértice que não foi visitado e realizamos a mesma operação. Essa operação se repete até que todos os vértices tenham sido visitados.
Em um grafo qualquer de X arestas, tomando como referência uma aresta qualquer, temos que essa aresta pode estar ligada a no máximo dois vértices. Dessa forma, o(s) vértice(s) que possuem essa aresta são adicionados ao conjunto solução.
Ao contrário da tarefa 1, que seleciona apenas um vértice a cada iteração da DFS, na tarefa 2 selecionamos dois vértices que são interligados por uma certa tarefa a cada vez que encontramos um vértices ou um par de vértices não visitados, portanto, podemos dizer que se o tamanho do conjunto solução da tarefa 1 é N, o da tarefa 2 é menor ou igual a 2N.