Documentação do Trabalho Prático 3 da disciplina de Algoritmos I

Lívia Delgado de Almeida Carneiro

2019054749

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

livia.delgado@dcc.ufmg.br

1. Introdução

Este trabalho visa documentar a solução para o problema descrito no Trabalho Prático 3, o qual propõe a abstração da construção de depósitos de vacinas para várias vilas agrícolas num estado brasileiro que contém uma extensa área florestal.

As próximas seções deste documento procuram, respectivamente, detalhar a implementação do programa supracitado (seção 2), fornecer instruções de compilação e execução do trabalho (seção 3), analisar a complexidade dos algoritmos da solução (seção 4), propor uma prova de corretude para o algoritmo implementado (seção 5) e apresentar os resultados da avaliação experimental (seção 6). Podem ser encontradas no final deste documento referências utilizadas durante a implementação.

2. Implementação

Iniciarei esta seção introduzindo a organização do código e explicando o fluxo geral do programa, para em seguida descrever a função de cada classe e suas interações. A implementação deste trabalho se encontra no GitHub.

2.1. Organização das pastas

A estrutura externa das pastas obedece o indicado pelas orientações do próprio trabalho prático, em que o executável *tp3* é criado no diretório raiz do projeto, a pasta *src* contém os arquivos de código, a pasta *include* contém os cabeçalhos do projeto; o *Makefile* contém comandos de compilação.

Internamente, temos a seguinte organização (que será mais profundamente detalhada):

- A pasta que contém os cabeçalhos tem a mesma forma que as subpastas da *src*. O arquivo *main.cpp* se encontra isolado das classes pois age como inicializador.

As subpastas da *src* estão organizadas da seguinte forma:

- As classes na pasta *grafo* representam a estrutura que abstrai os conceitos do problema representados como elementos do grafo: Trilha (aresta) e Vila (vértice), além da implementação dos algoritmos selecionados para resolver o problema (*Resolvedor* e *Aproximador*);
- A classe na pasta *leitura* é responsável pela leitura do arquivo de entrada e conversão das informações para os objetos conhecidos internamente.

2.2. Descrição das classes

2.2.1. Leitor

A classe *Leitor* executa a leitura do arquivo passado como parâmetro e cria um vetor de trilhas (representações das arestas, caminhos entre os vértices) e vilas (vértices), já computando as vilas adjacentes como forma de otimização de performance.

2.2.2. Vila

A classe *Vila* representa um vértice no grafo. É composta pelo índice identificador da Vila e um vetor de vilas adjacentes a ela, além de uma variável booleana que indica se foi deletada ou não - variável esta que serve também para otimização de performance.

2.2.3. Trilha

A classe *Trilha* representa um caminho entre duas vilas (uma aresta não direcionada que conecta dois vértices). Composta por identificadores das vilas que conecta.

2.2.4. Resolvedor

A classe *Resolvedor* implementa um algoritmo proposto por Sharad Singh, Gaurav Singh e Neeraj Kushwah (IJSER, 2018) para resolver o clássico problema NP-Hard de contabilizar o número mínimo de vértices numa cobertura de vértices - tal que todas as arestas não direcionadas se conectam a pelo menos 1 vértice selecionado -, o *Minimum Vertex Cover*.

A implementação é baseada em 2 etapas principais:

- Ordenar as vilas de forma ascendente pelo número de trilhas que levam a ela (número de vértices adjacentes);
- Selecionar as vilas nas quais serão construídos os depósitos.

A etapa de seleção segue o que foi descrito no artigo original [1]:

- Para cada vértice do grafo, deleta o vértice se todos os vértices adjacentes a ele ainda estiverem na lista. Se foi deletado, o número de vértices da cobertura é decrescido em 1 unidade.

No caso dessa implementação, optei por utilizar um novo vetor (de ponteiros) de Vilas, eliminando a necessidade de remover o elemento do vetor e computando as mudanças na flag "deletada" da vila. Dessa forma, é otimizada a performance do algoritmo.

2.2.5. Aproximador

A classe *Aproximador* implementa o algoritmo descrito em material da disciplina de Algoritmos da Universidade de Dartmouth [2].

Inicia-se o conjunto C (conjunto de vértices da cobertura mínima). Enquanto houver arestas a se analisar, escolhe-se qualquer aresta desse conjunto E e adiciona os vértices ao conjunto C. Então, deleta-se todas as arestas incidentes a qualquer um dos vértices adicionados. No final, retorna-se o conjunto C.

Ela não foi utilizada de fato, mas pensei ser interessante incluí-la para fins de referência de algoritmos comprovadamente eficientes em comparação com heurísticas.

2.3. Configuração do ambiente de testes

O programa foi desenvolvido em C++ 11, executado e testado no ambiente Linux 64x (distribuição Lubuntu), compilado utilizando o *g++*.

Processador: Intel(R) Core(TM) i5-5200U CPU @ 2.20GHz. Memória RAM: 8,00 GB.

3. Instruções de compilação e execução

O programa deve ser executado da seguinte forma:

1. Acesse o diretório onde foi extraído o conteúdo deste trabalho
2. Entre na pasta **algoritmos-I-tp3**
3. Pelo terminal, execute a instrução **make** para compilar o programa
    a. Deverá ser gerado um arquivo **tp3** dentro da pasta **algoritmos-I-tp3**
4. Pelo terminal, execute o programa com as instruções
   ./**tp3 tipo_tarefa caminho_arquivo_entrada.txt**
    a. Sendo o tipo_tarefa *tarefa1* ou *tarefa2*
5. O programa deverá ter executado as instruções informadas no arquivo de entrada e imprimido na saída padrão (o terminal, no caso) o resultado das codificações e decodificações.
6. Para executar os testes da tarefa 1 - executa-se o comando *make test*
7. Para executar os testes da tarefa 2 - executa-se o comando *make test2*

4. Análise de Complexidade

Considerando *n* o número de vértices (vilas) entre 0 e 10^5 e *m* o número de arestas (trilhas) entre 0 e10^2.

4.1. Resolvedor

4.1.1. Análise de Tempo

Consideramos a parte de ordenar as vilas pelo número de vértices adjacentes como $O(n\,log(n))$ - pela documentação oficial da linguagem [3].

A segunda parte possui complexidade $O(n^2)$, pois percorre o vetor de vilas (com todas as *n*) e internamente a esse loop percorre todas as vilas adjacentes (limitadas superiormente por *n*).

Como uma última etapa para auxiliar na impressão, temos outra execução do algoritmo de ordenação$O(n\,log(n))$.

Assim, no final, o que limita a execução desse algoritmo superiormente é a segunda parte - de calcular em quais vilas serão construídos os depósitos. Podemos finalizar descrevendo a complexidade como $O(n^2)$.

4.1.2. Análise de Espaço

Como são armazenados 3 vetores de ponteiros vilas (que seguem os mesmos limites) e um vetor de trilhas, podemos considerar a complexidade de espaço como $O(m + n)$.

4.2. Aproximador

4.2.1. Análise de Tempo

No fluxo do algoritmo, percorremos o vetor de trilhas 2 (duas) vezes - uma no *while* e outra no *for* da remoção. Temos também a ordenação do set no final, para fins de impressão do resultado.

Assim, podemos descrever a complexidade do algoritmo como O(m²).

4.2.1. Análise de Espaço

É armazenado um vetor de trilhas (*m*) e um set de inteiros representando as vilas (*n*). Para cada trilha, a cada remoção, é criado um novo vetor de trilhas com todas aquelas que não são incidentes aos vértices da cobertura de vértices. Totalizando uma complexidade de espaço de O(n²).

5. Prova de Corretude

5.1. Resolvedor

Considerando o algoritmo como a função *selecionarVilasParaConstruirDeposito*.
Pré-condições:

- Vilas adjacentes já foram processadas;
- Vilas estão ordenadas de forma ascendente pelo número de caminhos (trilhas) que levam a ela.

Descrição simplificada do algoritmo:

> Para cada vila:
>
> Se todas as vilas adjacentes a ela existirem, deleta a vila;
> Senão, adiciona um depósito nessa vila.

Prova de que o algoritmo termina:

- O loop é em função das vilas ordenadas. Sendo assim, quando o número de vilas analisadas foi igual ao número de vilas da entrada, o algoritmo finaliza a execução.

Prova de que o algoritmo retorna a solução correta:

- Como as vilas estão ordenadas pelo número de caminho, as últimas analisadas serão aquelas com mais caminhos incidentes.
    - Sendo assim, garantimos que todas as arestas incidentes a esses vértices que possuem mais arestas de entrada têm pelo menos 1 vértice escolhido na cobertura.
- A condição para que seja adicionado um depósito à vila é que todas as vilas conectadas a ela por meio de um caminho (aresta) existam.
    - Isso garante que todas as arestas incidirão sobre pelo menos 1 vértice selecionado.

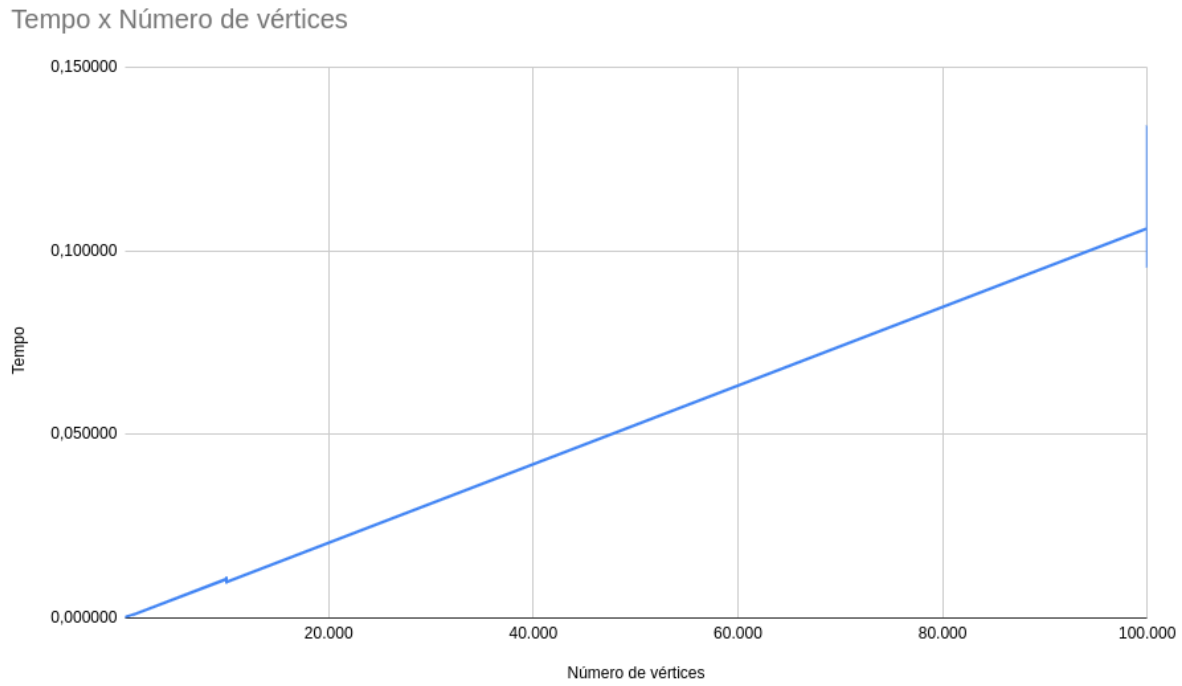
6. Avaliação Experimental

6.1. Desempenho

Para executar os testes de performance - essenciais para que fossem percebidos pontos de melhoria no algoritmo - criei um notebook Python, no qual utilizei a biblioteca *networkx* para criar os grafos automaticamente. Não verifiquei a corretude dos valores, focando na performance. Procurei manter todos os testes na mesma faixa em relação ao número de arestas - sendo o número de vértices fixo.

Número de Vértices	Valores obtidos	Média	Desvio Padrão
10^2	0,000248 0,000259 0,000264 0,000260 0,000293 0,000317	0,000262	0,00002609789264
10^3	0,001133 0,001058 0,000877 0,001280 0,001073 0,001004	0,001066	0,0001340319614
10^4	0,010688 0,011299 0,009469 0,010667 0,009550 0,009780	0,010224	0,0007465422739
10^5	0,106031 0,134006 0,097616 0,095588 0,095427 0,096159	0,096888	0,01517145885

Relação obtida entre tempo de execução e número de vértices:



O que aparenta ser uma complexidade linear - e pode, então, não representar o pior caso do algoritmo. Como a limitação superior quadrática atende ao caso de $\Omega(n)$, a análise de complexidade de tempo parece correta - ainda que não precisa nos casos coletados.

Os casos coletados estão no branch performance-tarefa-1 do repositório no Github.

6.2. Tarefa 2 - Aproximação do resultado

As imagens dos grafos gerados se encontram no arquivo **GrafosTarefa2.pdf.**

Caso de teste	Resultado ótimo	Máximo resultado esperado	Resultado obtido	Correto?
00	4	8	4	Sim
01	4	8	4	Sim
02	6	12	6	Sim
03	4	8	4	Sim
04	8	16	8	Sim
05	4	8	4	Sim

Referências

C PLUS PLUS REFERENCE. **Reference.** Disponível em: https://www.cplusplus.com/reference/. Acesso em: 02 set. 2021.

C PLUS PLUS REFERENCE. **Reference.** Disponível em: https://dl.acm.org/doi/10.1145/1597036.1597045. Acesso em 31 ago. 2021.

IJSER. **Optimal Algorithm for Solving Vertex Cover Problem in Polynomial Time**. Disponível em: https://www.ijser.org/researchpaper/Optimal-Algorithm-for-Solving-Vertex-Cover-Problem-in-Polynomial-Time.pdf. Acesso em 31 ago. 2021.

STACK EXCHANGE. **Theoretical Computer Science: Is the current best approximation ratio for Vertex Cover problem also a lower bound?**. Disponível em: https://cstheory.stackexchange.com/questions/37713/is-the-current-best-approximation-ratio-for-vertex-cover-problem-also-a-lower-bo. Acesso em 01 set. 2021.

WEIZMANN INSTITUTE OF SCIENCE. **A better approximation ratio for the Vertex Cover problem**. Disponível em: https://eccc.weizmann.ac.il/report/2004/084/. Acesso em 01 set. 2021.

DARTMOUTH. **Approximation Algorithms: Vertex Cover**. Disponível em: https://www.cs.dartmouth.edu/~ac/Teach/CS105-Winter05/Notes/wan-ba-scribe.pdf. Acesso em: 02 set. 2021.

NETWORKX. **Graph Generators**. Disponível em: https://networkx.org/documentation/stable/reference/generators.html. Acesso em: 04 set. 2021.