UNIVERSIDADE FEDERAL DE MINAS GERAIS

DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO

Algoritmos I
Trabalho Prático III – Minimum Vertex Cover

João Marcos Couto

Professora: Olga Nikolaevna

Belo Horizonte
22 de Junho de 2019

Sumário

1 Introdução **2**

2 Implementação **2**

3 Analise Experimental **3**
3.1 Tempo de execução 3

4 Análise de Complexidade Assintótica **5**
4.1 Análise Temporal 5
4.1.1 Leitura dos dados de entrada e criação de grafos: O(M*N) e O(M+N) . . . 5
4.1.2 Tarefa 1:DFS -> $\mathcal{O}(2M + N)$ 5
4.1.3 Tarefa 2: Iteração em lista de arestas -> $\mathcal{O}(M + N)$ 5
4.2 Análise Espacial 5
4.2.1 Tarefa 1: O(N*N) 5
4.2.2 Tarefa 2: O(M + N) 5

1 Introdução

Este trabalho prático objetivou a eleboração de um algoritmo que, dado um conjunto de arestas, nos permite encontrar o "Vertex Cover"do grafo formado pelas arestas da entrada.O vertex cover de um grafo conectado e não direcionado, é um subconjunto de vertices tal que toda aresta do grafo original tem pelo menos uma de suas extremidades dentro deste subconjunto. O minimum vertex cover, naturalmente, é um vertex cover com a menor cardinalidade possivel. O desafio foi dividido em duas tarefas. Na primeira tarefa, o conjunto das arestas formam um único caminho possivel entre quaisquer dois pontos (vertices) do grafo. Em outras palavras, nesta tarefa temos a restricão de que o grafo na entrada não contém ciclo, contituindo portanto uma árvore. Já na segunda tarefa, é possivel que existam mais de um caminho entre dois vertices do grafo, ou seja, a existência de ciclos é uma possibilidade.

2 Implementação

Uma rápida pesquisa na internet permite determinar que para a primeira tarefa é possível encontrar uma solução ótima e em tempo polinomial. Já a segunda, é diferente da primeira de tal forma que a torna um problema NP-completo. Porém, existe uma heuristica que permite encontrar uma solução que é no máximo duas vezes pior que a primeira.

Para a primeira tarefa implementei o seguinte algoritmo:

```
Root the tree at vertex 0
function dfs(u)
 foreach (children v of u) dfs(v)
 if u is not covered then
  if u has parent then take u's parent
  else take u
dfs(0)
```

Figura 1 Pseudo código tarefa 1

Para a segunda:

$C \leftarrow \emptyset$
while $E \neq \emptyset$
    pick any $\{u, v\} \in E$
    $C \leftarrow C \cup \{u, v\}$
    delete all eges incident to either u or v
return C

Figura 2 Pseudo código tarefa 2

Tarefa 1: optei pela utilização de uma lista de adjascencia para representar o grafo em memória. Isto é, criei um vetor de n posições onde cada uma delas é um ponteiro para uma lista encadeada. Em cada lista encadeada, uma celul é uma sintetizado em uma struct do tipo vértice, que armazena o índice "id"de cada um dos vizinho. Isto é, a posição i do vetor de listas de adjacência aponta para o primeiro vértice do vértice de índice i

Além disso, utilizo um vetor de n inteiros para indicar o status de cada um dos vértices. Neste vetor, -1 indica que o vertice ainda não foi visitado pela dfs, 1 indica que foi visitado mas ainda não foi coberto pelo vertex cover, 2 indica que já foi visitado e coberto, e por fim, 3 indica que um vértice faz parte da solução final do vertex cover. Assim, posteriormente basta contar o número de vezes que o 3 aparece no vetor para encontrar a resposta final.

Tarefa 2: criei uma struct "aresta"que armazena o id de cada um dos vértices que uma dada aresta conecta. Daí bastou criar um vetor de numArestas posições onde cada posição é uma struct representando uma das arestas recebidas na entrada. Além disso, para eliminar a necessidade de se deletar elementos do vetor de arestas e assim reduzir imensamente a complexidade do algoritmo, criei um vetor de inteiros auxiliar que armazena uma inteiro para cada vertice, onde 3 índica que um vértice já está na solução final e -1 indica que não. Tendo feito isso bastou iterar de aresta em aresta e então incluir na solução final seus dois vértices caso ambos ainda não tenham sido íncluidos na solução final.

3 Analise Experimental

3.1 Tempo de execução

Escrevi um programa simples para gerar arquivos de texto na mesma formatação dos arquivos disponibilizados para teste. O programa gera 20 conjuntos de entradas para tamanhos sordidos de grafo. O gráfico a seguir apresenta, para cada tamanho de grafo, a média dos tempos de execução dos dois algoritmos, seguido do desvio padrão destes tempos

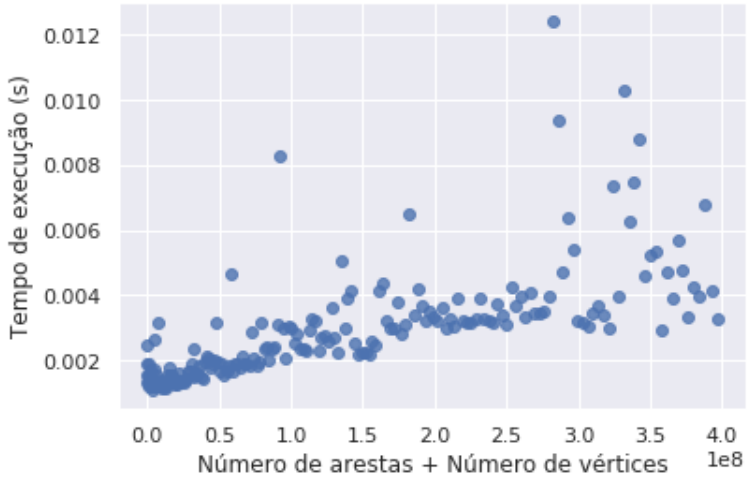

Figura 3 Tempo de execução da tarefa 1 em função do número de arestas+vértices.

A seguir, executei o programa acima 10 vezes para obter várias médias de tempos para um mesmo número de arestas. Plotei então o desvio padrão do tempo de execução do programa em cada tamanho de grafo

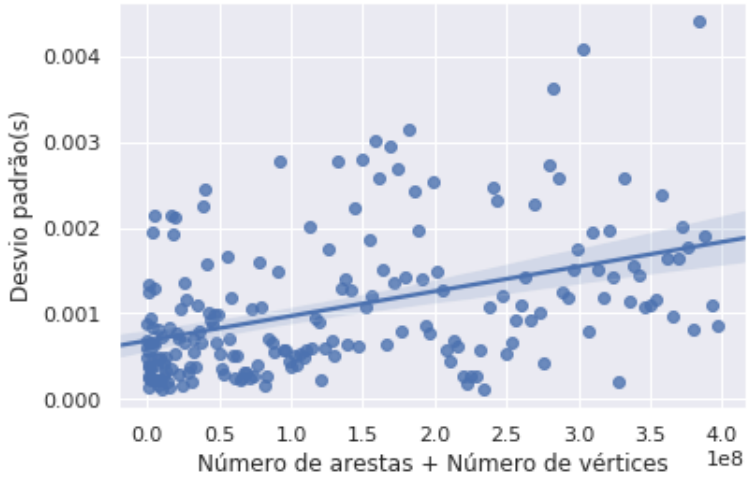

Figura 4 Desvio padrão do tempo de execução da tarefa 1 em função do número de arestas+vértices.

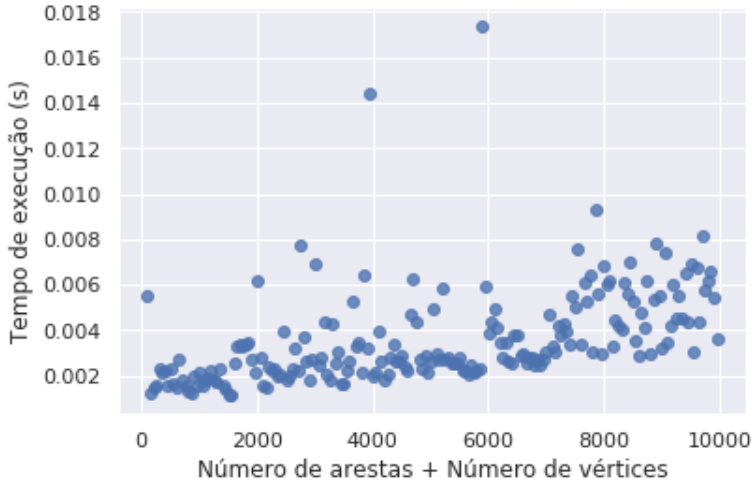

Figura 5 Tempo de execução da tarefa 2 em função do número de arestas+vértices.

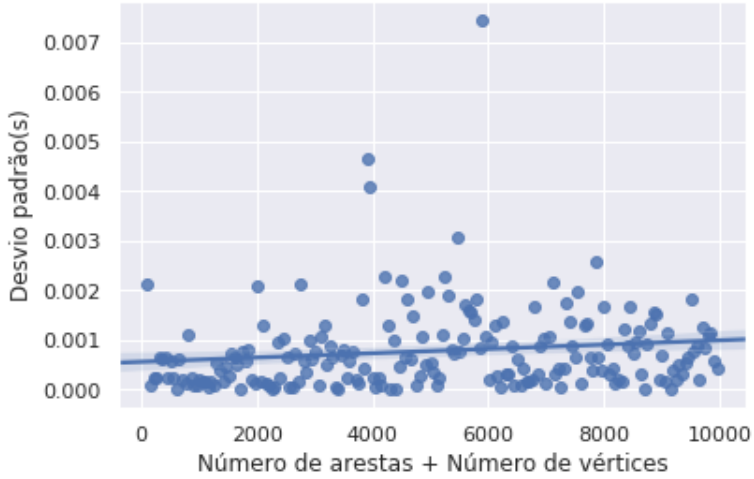

Figura 6 Desvio padrão do tempo de execução da tarefa 2 em função do número de arestas+vértices.

4 Análise de Complexidade Assintótica

M representa o número de arestas, e N o número de vértices

4.1 Análise Temporal

4.1.1 Leitura dos dados de entrada e criação de grafos: O(M*N) e O(M+N)

Na tarefa 1 faço uso de um vetor de listas de adjacencia, assim, na pior das hipóteses temos um grafo completo, portanto precisamos de fazer M*N inserções em listas, que individualmente têm custo constante.

Na tarefa 2 faço uso de um vetor com dimensão proporcional ao número de arestas e outro proporcional ao número de vértices, portanto a leitura e criação do grafo aqui, tem custo 0(M*N)

4.1.2 Tarefa 1:DFS -> $\mathcal{O}(2M + N)$

Utilizada apenas na tarefa 1, tem complexidade pois implementei o algortimo utilizando listas de adjacência, onde cada nó mantém uma lista de todos os seus vizinhos e então, para cada vertice podemos descobrir todos seus vizinhos pulando de um para o próximo na lista. Por estarmos trabalhando com um grafo não direcionado cada aresta vai aparecer duas vezes, uma vez para cada uma de suas duas extremidades, portanto temos complexidade O(2M+N)

4.1.3 Tarefa 2: Iteração em lista de arestas -> $\mathcal{O}(M + N)$

O algoritmo tem complexidade O(M+N) pois em primeiro momento é necessario inicializar o status de cobertura de todos os vértices com custo O(N). Então itera-se pela lista de arestas, de uma em uma, fazendo um número constante de operações para cada, com custo O(M)

Referências:

-> http://tandy.cs.illinois.edu/dartmouth-cs-approx.pdf -> https://visualgo.net/en/mvc

4.2 Análise Espacial

4.2.1 Tarefa 1: O(N*N)

Na pior das hipóteses, temos uma situação onde o grafo é completo, isto é, todos os vertices tem uma aresta com todos os outros. Em vista de que implementei um vetor de listas de adjacência temos então uma complexidade O(N*N)

4.2.2 Tarefa 2: O(M + N)

Aqui, utiliza-se um vetor de arestas, com complexidade O(M), e um vetor indicador de status de cada um dos vértices, de complexidade O(N)