Documentação TP3 - Aproximação para o Vértex-Cover

Isadora Alves de Salles

23/06/2019

1 Introdução

Este trabalho teve como objetivo encontrar o mínimo de bandeiras necessárias para que cada caminho entre dois pontos tenha pelo menos uma bandeira. Traduzindo para o vocabulário de grafos, o que queremos é encontrar o minimum vértex cover, ou seja, queremos encontrar o mínimo de vértices que cobrem todas as arestas do grafo. Para isso, devemos colorir o grafo de forma que todo caminho entre dois pontos sempre seja incidente a pelo menos uma bandeira (vértice que faz parte do vértex cover). O vértex-cover é um algoritmo que faz parte do conjunto NP-completo, então não sabemos se conseguimos obter uma resposta em tempo polinomial para todas as possíveis instâncias do problema. Para a tarefa 1, temos grafos sem ciclo, ou seja, árvores, e é possível encontrar o valor exato para o MVC. Já para a tarefa 2, que são grafos com ciclos foi necessário criar uma heurística com aproximação até 2 vezes pior que o MVC.

2 Solução do Problema

Para a implementação do problema o grafo foi implementado com lista de adjacência, que era uma solução melhor para ambas as tarefas. A solução das duas etapas do problema serão explicadas separadamente a seguir:

2.1 Tarefa 1

Para a primeira tarefa é necessário usar alguma abordagem de algoritmos indutivos como: guloso, programação dinâmica ou divisão e conquista. Escolhi utilizar um algoritmo guloso e para solucionar o problema fiz uso de uma busca em profundidade. A ideia é que, para colorir os vértices de uma árvore de forma a cobrir todas as arestas com a menor quantidade de vértices possíveis nós não queremos colorir as folhas das árvores, pois isso não nos daria a solução ótima. Isso ocorre pois, dado que uma árvore não tem ciclos, colorir uma folha implica que apenas uma aresta será coberta por aquele nó. Para evitar isso, queremos colorir o pai das folhas, e teremos que colorir todos os pais de todas as folhas.

Primeiramente coloco como raiz o nó 0. E assim, rodamos uma busca em profundidade, que é um algoritmo recursivo, e quando chegamos num nó folha, a pilha de recursão é desfeita e colorimos o todos os nós que fizeram parte dessa busca de forma a fazer cor sim, cor não, começando pelo pai da folha como sendo integrante do vértex cover. Sempre é necessário verificar se o vértice já está coberto, se não estiver, colore o pai dele, se ele não possuir pai e não estiver coberto, ele deve ser incluído ao vértex cover. Entenda colorir como tornar parte do nosso conjunto de vértices que fazem parte da cobertura.

2.1.1 Prova de corretude do algoritmo com abordagem gulosa

Aplicando o paradigma guloso, fazendo uso de uma busca em profundidade, conseguimos encontrar a solução ótima para o vértex cover em árvores com um custo muito baixa. Podemos certificar que sempre encontramos a solução ótima pois sabemos que para árvores o nó folha não

deve estar na solução ótima para cobertura, pois este é o nó que cobre o menor número de arestas (apenas uma). Assim, fazendo o uso da DFS e colocando o pai de cada vértice na cobertura, garantimos que não pegaremos as folhas. Para garantir que a cobertura é total passamos em todos os vértices verificando que se eles foram cobertos quando desfazemos a pilha de recursão de cada busca em profundidade. Assim, garantimos que a solução encontrada será o minimum vértex cover.

2.2 Tarefa 2

Para a segunda tarefa temos que o vértex cover é NP-completo, e devemos criar uma heurística para aproximar da solução e obter um resultado até 2 vezes superior ao resultado real.

Para isso o algoritmo proposto é pegar uma aresta arbitrária, eu decidi fazer a escolha das arestas na ordem da minha lista de adjacência, sabemos que qualquer cobertura de vértices deve conter um dos dois vértices que se conectam a essa aresta, então pegamos os dois vértices, e excluímos todas as arestas que estão cobertas por esses dois vértices e repetimos o processo até que não sobre nenhuma aresta não coberta. A seguir o pseudocódigo do algoritmo proposto:

Algorithm 1: APPROX-VERTEX-COVER(G)

```
1 C←∅
2 while E ≠ ∅
        pick any {u, v} ∈ E
        C ← C ∪ {u, v}
        delete all eges incident to either u or v

  return C
```

Figura 1: Pseudocódigo para a heurística.

Porém, percebi que seria um pouco custoso excluir as arestas, dado que temos grafos não direcionados, no pior se o grafo for completo teremos que percorrer m-1 arestas para conseguir excluir uma aresta "espelhada"da lista de um vértice. Sendo assim, eu decidi armazenar um vetor com todas as arestas e percorrer o mesmo marcando os vértices como pertencentes ou não ao vértex cover. A ideia é a mesma, só evita que necessariamente tenhamos que remover as arestas, basta conferir se os dois vértices nas pontas de uma aresta não pertencem ao vértex cover.

2.2.1 Prova de corretude da heurística

O algoritmo sempre retornará um vértex cover válido visto que ao fazer a leitura das arestas, passamos por todas elas e verificamos se está coberta.

O que encontramos no final da heurística proposta é um set de arestas máximo no qual nenhuma delas compartilha um vértice, assim não podemos adicionar nenhuma outra aresta nesse set. Isso ocorre pois sempre que escolhemos uma aresta verificamos se os vértices no final dela não estão na nossa cobertura. Se nenhum dos dois vértices estão na cobertura, sabemos que pelo menos um deve estar na cobertura, logo pegamos os dois. Como nesse algoritmo sempre iremos colocar os dois vértices que compartilham a mesma aresta, enquanto que na solução ótima apenas um desses dois vértices entraria na cobertura, temos um upper bound de ser no máximo duas vezes pior do que a solução ótima.

3 Análise de Complexidade

Denotaremos por n o número de vértices e m o número de arestas.

3.1 Complexidade Assintótica de Tempo

O programa executa quatro partes principais: lê as entradas, cria grafos, executa uma DFS ou executa a heurística para vértex cover aproximado. Vamos então analisar as complexidades separadamente:

3.1.1 Leitura dos dados: $\mathcal{O}(n)$

Nesta etapa cada um dos caminhos, ou seja, arestas, é lido em tempo $\mathcal{O}(1)$. Assim a complexidade é $\mathcal{O}(m)$.

3.1.2 Criar grafos: $\mathcal{O}(m)$

Para criar o grafo precisamos colocar cada uma das arestas nele, assim essa etapa tem custo linear no número de arestas $\mathcal{O}(m)$.

3.1.3 Busca em Profundidade (DFS): $\mathcal{O}(n+m)$

Para encontrar vértex cover para árvores é usada uma busca em profundidade, que tem o custo $\mathcal{O}(n+m)$. As operações de coloração dos vértices tem custo $\mathcal{O}(1)$ e não afetam a complexidade da DFS. Essa complexidade será provada na análise experimental.

3.1.4 Heurística: $\mathcal{O}(m)$

Para a heurística foi criado um vetor de arestas, e o que é feito no algoritmo é apenas percorrer todo este vetor colorindo vértices que farão parte da cobertura e verificando se os vértices já fazem parte, ou seja, temos apenas operações com custo constante m vezes, então o custo da heurística é linear no número de arestas, $\mathcal{O}(m)$.

3.2 Complexidade Assintótica de Espaço

Além de variáveis simples, um grafo é armazenado em memória. Na na construção do grafo 3 estruturas são armazenadas em memória. A primeira é um vetor de structs para as arestas, contendo 2 variáveis na struct, que foi criado para facilitar as operações da heurística e tem custo de espaço igual a $\mathcal{O}(2m)$. A segunda é uma lista de adjacência para cada um dos vértices, contendo todas as arestas, tendo custo no espaço de $\mathcal{O}(n*(n-1))$ pois podemos receber um grafo completo como instância do problema e nele cada vértice se conecta a todos os outros, e como o grafo é não direcionado cada aresta será listada 2 vezes. E em terceiro temos um vetor de tamanho igual ao número de vértices para armazenar a coloração dos vértices, isso será usado para a solução do vértex cover tanto da tarefa 1 quanto da 2 e tem custo no espaço igual a $\mathcal{O}(n)$.

Assim a complexidade assintótica de espaço do programa é
$\mathcal{O}(2m) + \mathcal{O}(n*(n-1)) + \mathcal{O}(n) = \mathcal{O}(n*(n-1)) = \mathcal{O}(n^2)$

4 Análise Experimental

Para realizar a análise experimental foi criado um programa em C++ para gerar casos de teste automaticamente, para um número definido de vértices e arestas. E um outro programa foi criado para executar cada um dos casos de teste gerados e marcar o tempo de execução do algoritmo.

Essa linguagem foi escolhida porque ela possui formas mais precisas de medir o tempo do que a biblioteca $< time.h >$.

Para observar o comportamento assintótico dos dois algoritmos (para a tarefa 1 e 2) devemos aumentar o número de vértices, e consequentemente o número de arestas dado que sempre serão gerados grafos conexos para esta análise. Sendo assim, foram gerados 2000 casos de tese, variando o número de vértices do grafo, aumentando de 50 em 50, até 10000. Cada caso de teste foi executado 10 vezes, e a média do tempo de execução foi reportada nos gráficos a seguir. Os valores de desvio padrão entre o tempo de execução de cada um dos 10 testes com o mesmo número de vértices não foram reportados neste relatório, pois os tempos de execução deram muito próximos, ou seja, tiveram pouca variância, fazendo com que o desvio padrão fosse irrelevante, principalmente para testes que formam grafos com poucas arestas.

Ao final da execução do programa de testes, foi utilizada a linguagem R para plotar dois gráficos, um para comprovar a complexidade do algoritmo para solução da tarefa 1 e outro para comprovar a heurística da tarefa 2. O primeiro gráfico foi gerado em função de arestas + vértices para mostrar que a DFS tem custo $\mathcal{O}(n + m)$. Já o segundo gráfico nos mostra que a heurística tem complexidade $\mathcal{O}(m)$.

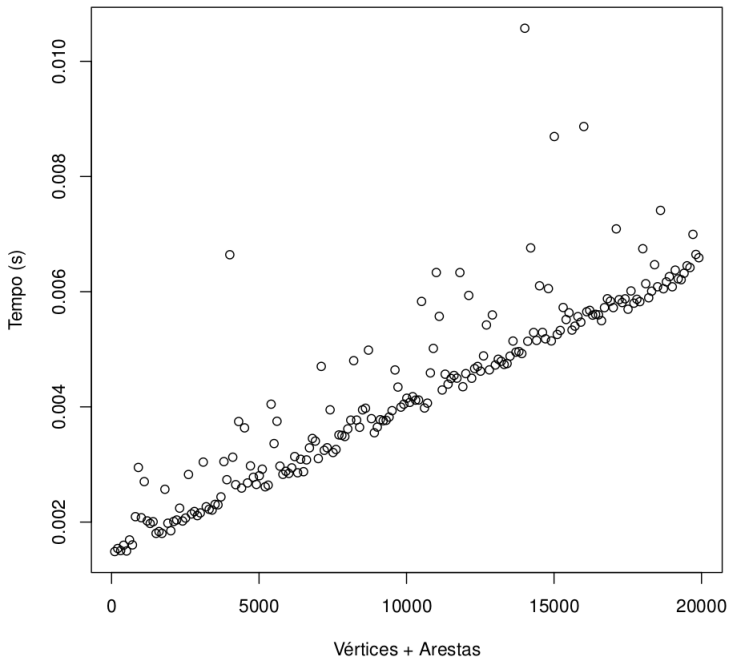

Figura 2: Tempo de execução do programa para a tarefa 1.

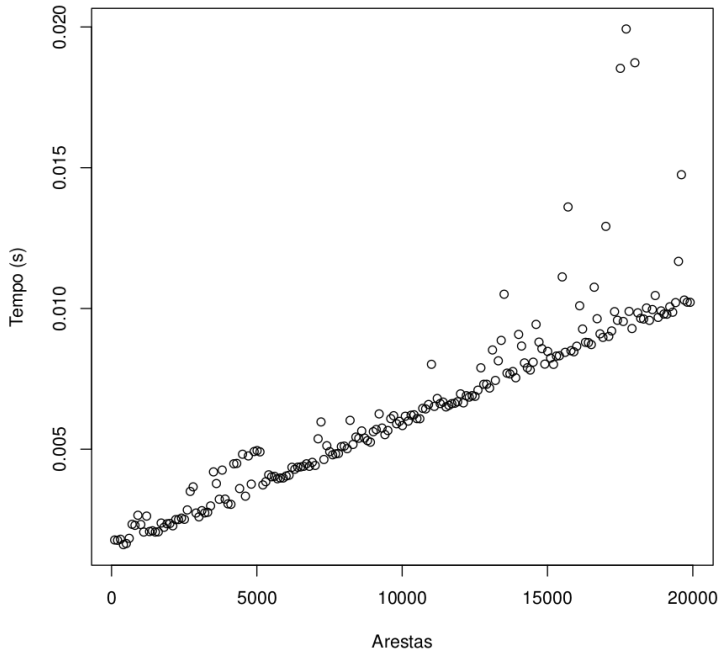

Figura 3: Tempo de execução do programa para a tarefa 2.

Em ambos os gráficos observamos alguns outliers, que fogem a reta, porém isso não compromete a complexidade dos algoritmos, visto que o tempo de execução deles é extremamente baixo.

A seguir temos um exemplo para demonstrar o resultado encontrado pela heurística. Dado o grafo da Figura 4, sabemos que o mínimo vértex cover deve conter 4 vértices, que seriam 1, 2, 4 e 6 por exemplo.

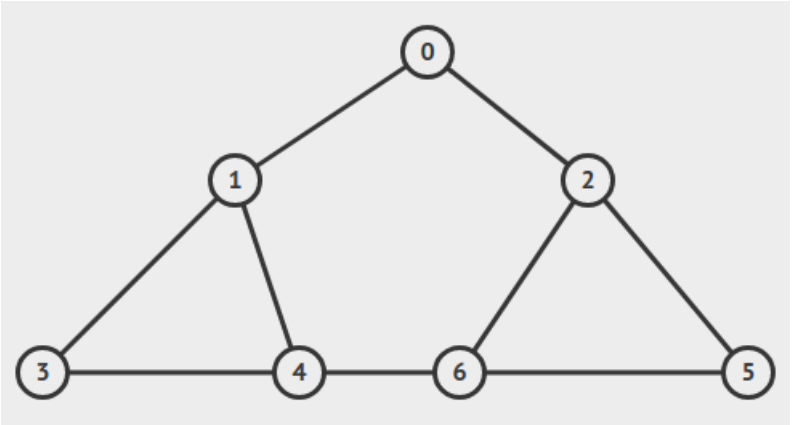

Figura 4: Grafo antes de rodar o algoritmo para MVC.

Na Figura 5 temos a solução encontrada pela heurística, os vértices pintados de laranja são os que fazem parte da nossa cobertura, então temos 6 vértices no vértex cover, isso não ultrapassa o limite de 2 vezes pior do que a solução ótima que seria 4 vértices.

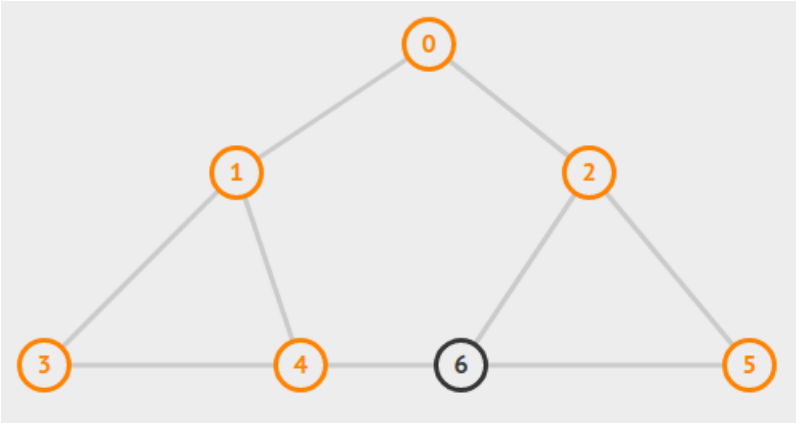

Figura 5: Grafo após executar a heurística.

5 Conclusão

Neste trabalho o problema de encontrar o vértex cover para árvores foi resolvido em tempo linear no número de arestas + vértices, e o aproximativo do vértex cover para grafos com ciclo foi resolvido em tempo linear no número de arestas. O problema foi modelado por um grafo e fez uso de uma busca em profundidade além do conceito de algoritmos aproximativos.

O problema do vértex cover é um problema NP-completo, e o desafio deste trabalho era conseguir uma heurística que obtenha um resultado aproximado ao do vértex cover, no máximo 2 vezes pior, e que rode em tempo polinomial para um grafo qualquer. Na análise experimental, foi mostrado um exemplo de instância para o algoritmo e a resposta do mesmo, dada a resposta esperada para comparação.

Por fim, a complexidade foi ilustrada e verificada por meio da análise experimental, na qual verificou-se que os algoritmos tem um comportamento, de fato, polinomial.