Documentação TP2

Thiago Martin Poppe

22/06/2019

1 Uma breve descrição do problema

Foi entregue duas tarefas a serem realizadas por um grupo de escoteiros. Ambas tarefas consiste em distribuir bandeiras em uma trilha de tal forma que todos os caminhos c_k existentes entre dois pontos (p_i, p_j) sempre seja incidente a pelo menos uma bandeira. Na primeira tarefa, a trilha não possui ciclos e a partir de qualquer ponto, conseguimos chegar nos demais seguindo caminhos consecutivos; já na segunda tarefa, a trilha possui ciclos, exigindo maior atenção dos escoteiros.

A partir da descrição do problema, percebemos que podemos modelar problema utilizando um grafo, onde os caminhos representarão as arestas e os pontos representarão os vértices. Com isso, o problema nada mais é do que achar o $vertex - cover$ mínimo desse grafo, ou seja, selecionar o menor grupo de vértices tal que todas as arestas do grafo sejam cobertas por eles. Esse problema em sua forma otimizada para um grafo genérico, pertence à classe de problemas $NP - Completo$, ou seja, apenas conseguimos uma solução polinomial para tal, se e somente se for provado que $P = NP$. Com isso, em grafos genéricos (com ciclos, etc) apenas conseguimos encontrar uma aproximação do que seria uma solução ótima para o problema. Porém, quando o nosso grafo é uma árvore (i.e. grafos conexos e acíclicos), conseguimos encontrar uma solução ótima para o problema em tempo polinomial.

2 Estruturas de Dados e Algoritmos

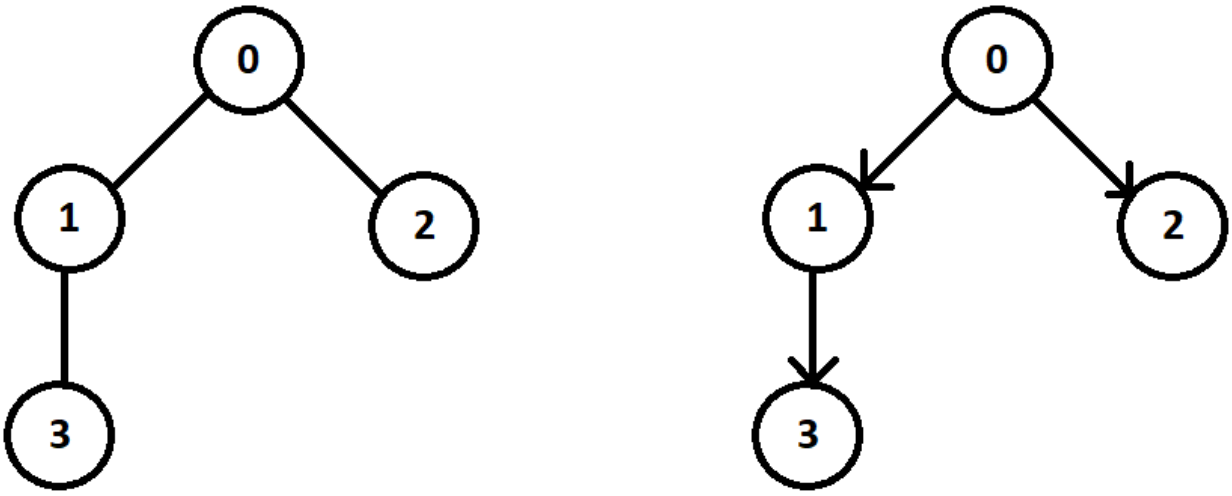
2.1 Estruturas de Dados utilizadas

Conseguimos modelar o seguinte problema usando grafos não direcionados, visto que o caminho entre dois pontos é de ida e volta. Sendo assim, os pontos da trilha serão representados pelos vértices e os caminhos serão representados pelas arestas.

Utilizei uma lista de adjacência como uma forma de representar o nosso grafo, ou seja, cada vértice possui consigo uma lista de conexões com os demais vértices, i.e. se a conexão (u, v) existir, temos uma aresta que liga os vértices u e v.

Para a tarefa 1, utilizo uma árvore para modelar o problema, visto que a trilha não possuirá ciclos e será conexa. A construção da mesma segue um algoritmo que consiste em retirar as conexões "de volta", ou seja, aquelas arestas que retornam para o vértice pai. Convertendo assim, um grafo não direcionado, para um direcionado. Para eliminar essas

arestas "de volta", passamos um vértice como a raiz da árvore (ou sub-árvore), percorremos a lista de adjacência dos seus filhos, removendo as conexões com essa raiz, e fazemos isso recursivamente. A seguir, temos um exemplo do algoritmo tornando o vértice 0 como a raiz da árvore.



Basicamente, temos 2 estruturas de dados principais:

- Uma TAD Lista, onde seus nós irão guardar o id do vértice e um ponteiro para o próximo nó.
- Uma TAD Grafo, que irá guardar o número de vértices e arestas, juntamente com um vetor de Lista (representação de uma lista de adjacência).

2.2 Algoritmos utilizados

2.2.1 Algoritmo para a tarefa 1

Mesmo que o problema de encontrar uma instância mínima de um $vertex - cover$ seja um problema da classe $NP - Completo$, quando o nosso grafo representa uma árvore, conseguimos obter uma solução polinomial para o mesmo. A seguir, apresento um pseudocódigo para esse algoritmo, seguido de uma explicação sobre o mesmo.

Algorithm 1 Retornar o tamanho mínimo de um vertex cover em um grafo árvore

```
1: procedure TREE_VERTEX_COVER(v, vertex_cover)
2:     if (vertex_cover[v] already computed) then
3:         return vertex_cover[v]
4:
5:     if (v doesn't have children) then
6:         vertex_cover[v] ← 0
7:         return 0
8:
9:     including_v ← 1
10:    for all children i of v do
11:        including_v ← including_v + tree_vertex_cover(i, vertex_cover)
12:
13:    excluding_v ← 0
14:    for all children i of v do
15:        excluding_v ← excluding_v + 1
16:        for all children j of i do
17:            excluding_v ← excluding_v + tree_vertex_cover(j, vertex_cover)
18:
19:    vertex_cover[v] ← min(including_v, excluding_v)
20:    return vertex_cover[v]
```

Percebemos que o algoritmo proposto utiliza o paradigma de programação dinâmica para resolver o problema. Nele, utilizamos um vetor chamado *vertex_cover* para guardar os valores previamente computados. A lógica do algoritmo é em pegar o resultado obtido quando consideramos o vértice atual como parte da solução e quando não consideramos como parte da solução, retornando o menor deles. Caso o vértice faça parte da solução, não podemos afirmar nada sobre seus filhos, logo chamamos a função para os mesmos. Quando o vértice atual não faz parte da solução, necessariamente seus filhos devem participar, para cumprir o requisito de cobrir todas as arestas. Com isso, chamamos a função para os seus netos dessa vez. Note que as folhas nunca farão parte da solução ótima, visto que pegar seus pais é muito mais vantajoso por cobrir mais arestas em determinadas situações.

2.2.2 Algoritmo para a tarefa 2

Em um grafo genérico, que pode conter ciclos por exemplo, o problema de achar uma instância mínima para o $vertex - cover$ torna-se $NP - Completo$. Uma saída, é tentar aproximar uma instância de tamanho no máximo 2 vezes pior que o tamanho da instância mínima, em tempo polinomial. A seguir, apresento um pseudocódigo para esse algoritmo, seguido de uma explicação sobre o mesmo.

O algoritmo consiste em basicamente, selecionar arbitrariamente arestas do nosso grafo de entrada, inserir ambos vértices que aquela aresta incide, e remover todos as arestas incidentes nos vértices. Mesmo o algoritmo sendo muito simples, ele nos retorna uma instância de tamanho no máximo 2 vezes pior do que a instância ótima, em tempo polinomial. Veremos

Algorithm 2 Retornar uma instância aproximada para o problema de vertex-cover

```
1: procedure APPROX_VERTEX_COVER(graph)
2:     result ← empty set
3:
4:     Let E be a set of edges in arbitrary order
5:     while E is not empty do
6:         (u, v) ← vertices that area connected by an edge in E
7:         Add (u, v) to the result
8:         Remove all edges from the set which are either incident to u or v
9:
10:    return result
```

com mais detalhes ambas partes nas próximas seções.

3 Análise de Complexidade

3.1 Complexidade Temporal

Tanto para a tarefa 1 quanto para a tarefa 2, precisamos passar por todos os vértices e por todas as arestas, seja para encontrar a solução ótima (tarefa 1) ou para excluir as arestas incidentes no vértices (tarefa 2). Note que a solução para a tarefa 1, flerta com uma complexidade quase exponencial. Mas devido ao uso do paradigma da programação dinâmica, evitamos recalcular valores previamente computados. Com isso, temos que a complexidade do algoritmo para a tarefa 1 é $O(|V| + |E|)$, onde V representa os vértices do nosso grafo de entrada e E suas arestas. A complexidade da solução para a tarefa 2 também será $O(|V| + |E|)$, visto que, mais uma vez, devemos passar por todos os vértices seja inserindo os mesmos na solução ou removendo suas arestas. Na tarefa 2, podemos conter grafos com ciclo, mais especificamente, grafos completos, visto que $0 < |E| \leq |V|^2$. Com isso, podemos ir além e dizer que se o grafo de entrada é um grafo completo, nossa complexidade temporal será da ordem de $O(V^2)$.

→ **OBS.:** A complexidade do algoritmo auxiliar que transforma o grafo em um grafo direcionado para a tarefa 1 também possui complexidade $O(|V| + |E|)$, visto que, devemos percorrer todos os vértices e arestas do mesmo.

3.2 Complexidade Espacial

Para guardar o grafo na memória, utilizamos a abordagem de lista de adjacências. Com isso, para a estrutura do grafo temos uma complexidade espacial $O(|V| + |E|)$, onde V representa os vértices do nosso grafo de entrada e E suas arestas. Na tarefa 2, podemos conter grafos com ciclo, mais especificamente, grafos completos, visto que $0 < |E| \leq |V|^2$. Com isso, podemos ir além e dizer que se o grafo de entrada é um grafo completo, nossa complexidade espacial será da ordem de $O(V^2)$.

4 Prova de Corretude

4.1 Tarefa 1

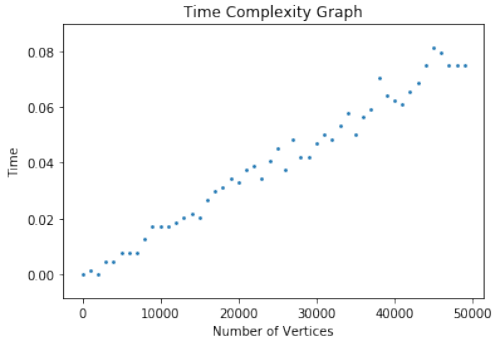
O algoritmo para a tarefa 1 tem um caráter de ser quase uma força bruta, ou seja, analisamos todas as possíveis soluções e escolhemos a melhor entre elas. Aplicando o paradigma de programação dinâmica, teremos uma melhoria na complexidade, mas ainda analisaremos todas as possíveis soluções. Por esses motivos, temos que a resposta para tal algoritmo sempre será a menor possível.

4.2 Tarefa 2

O algoritmo para a tarefa 2 sempre entregará como solução um vertex-cover válido, visto que, o algoritmo consiste em retirarmos as arestas dos vértices que fazem parte da solução, e terminar quando não temos mais arestas no nosso grafo.

Percebemos que o nosso algoritmo sempre irá pegar vértices em pares, ou seja, sempre colocará na solução os 2 vértices que compartilham aquela aresta, enquanto que na solução ótima teriamos adicionado um dos dois vértices, visto que estariamos cobrindo a mesma aresta 2 vezes. Com isso, temos que o nosso algoritmo tem um "*upper bound*" de ser no máximo duas vezes pior do que a solução ótima encontrada para o problema. Em outras palavras, seja S o conjunto de vértices da nossa solução aproximada, e S^* a solução ótima, temos que $\frac{S}{S^*} \leq 2$.

5 Avaliação Experimental



Com base no gráfico acima, temos que o nosso algoritmo se comporta de maneira linear no número de vértices, em outras palavras, tem uma complexidade assintótica de $O(|V|)$, onde V representa os vértices do nosso grafo.

Para cada caso de teste, gerados aleatoriamente, foram executados 20 vezes o programa e tiramos a média dos tempos de execução para plotar o gráfico. Uma observação é de que o tempo de execução corresponde ao programa inteiro, ou seja, criação do grafo completo mais a execução do algoritmo para computar o $Vertex - cover$.

A seguir, vemos uma imagem da resposta ótima de um problema de vertex-cover, e a resposta obtida pelo nosso algoritmo aproximativo, respectivamente, onde os vértices alaranjados representam a solução obtida.