Documentação TP2

Thiago Martin Poppe

20/05/2019

1 Uma breve descrição do problema

Motivados pela primeira fotografia de um buraco negro tirada no dia 10 de abril de 2019, várias pessoas do mundo inteiro começaram a ter interesse pela astrofotografia. Com isso, um investidor misterioso deseja construir uma rede com n telescópios tal que sempre exista comunicação entre um telescópio e outro. A sua intenção é minimizar a maior distância D entre dois telescópios, com isso, minimizando o custo da compra de um dispositivo que transimite o sinal entre eles.

2 Estruturas de Dados e Algoritmos

2.1 Estruturas de Dados utilizadas

Conseguimos modelar o seguinte problema usando grafos não direcionados, visto que a comunicação entre dois telescópios é bilateral. Sendo assim, os telescópios serão representados pelos vértices e as distâncias serão representadas pelas arestas.

Utilizei uma lista de adjacência como uma forma de representar o nosso grafo. Mesmo que inicialmente ele seja um grafo completo (sendo nesse caso mais recomendado uma modelagem através de uma matriz de adjacência), ao longo do nosso algoritmo o mesmo irá deixar de ser completo, o que torna a abordagem com lista de adjacência melhor.

Basicamente, temos 3 estruturas de dados principais:

- Uma TAD Lista, onde seus nós irão guardar o id do vértice e a distância que temos do vértice atual até esse.
- Uma TAD Grafo, que irá guardar o número de vértices e arestas, juntamente com um vetor de Lista (representação de uma lista de adjacência). Nele, também temos um vetor de estruturas Aresta que irá facilitar o acesso às arestas do grafo na forma de um vetor. Essa estrutura Aresta contém o peso daquela aresta e também os vértices aos quais ela liga.
- Uma estrutura Componente Conexo, que irá guardar o número de componentes conexos de um grafo e também os próprios componentes na forma de um vetor. Por exemplo, dado o vetor 0 0 1 0 2 3 0, sabemos que os vértices 0-1-3-6 fazem parte do componente

conexo de id 0; o vértice 2 faz parte do componente conexo de id 1; o vértice 4 faz parte do componente conexo de id 2; e por fim, o vértice 5 faz parte do componente conexo de id 3. Futuramente, essa estrutura será importante durante a criação de super vértices, sendo esse, um passo do algoritmo utilizado para resolver o problema proposto.

2.2 Algoritmos utilizados

Para resolver o problema, utilizamos uma modificação do algoritmo de Camerini que serve para acharmos uma MBST (*Minimum Bottleneck Spanning Tree*). Essencialmente, esse algoritmo retorna uma árvore geradora tal que a mesma não é necessariamente mínima. Atráves dela, conseguimos obter a menor aresta máxima que faz parte de todas as árvores geradoras possíveis, inclusive da MST (*Minimum Spanning Tree*). A modificação feita consiste em basicamente apenas retornar essa aresta, e não a MBST como no original.

Juntamente com essa modificação, utilizo alguns algoritmos auxiliares, os quais irei explicar brevemente.

2.2.1 MoM Select

O algoritmo de MoM Select consiste em basicamente selecionar o k-ésimo menor elemento de um vetor não ordenado com complexidade $O(n)$. Para tal, iremos implementar a ideia de um algoritmo quick-select, porém com a modificação de que o pivô escolhido será uma mediana aproximada do vetor de entrada. Com isso, o processo de partição garantirá que no mínimo 30% do vetor original seja desconsiderado, tornando assim $O(n)$ o pior caso do algoritmo, e não mais $O(n^2)$ como seria em um quick-select normal. Utilizaremos esse algoritmo para computar a mediana das arestas do nosso grafo em tempo linear.

2.2.2 DFS (Depth-First Search)

Utilizamos a DFS para vários subproblemas em grafos. Ela consiste em basicamente caso um vértice não seja visitado, marcamos ele como visitado e exploramos seus vizinhos.

Utilizamos a DFS para descobrir se um grafo é conexo ou não e para retornar os seus componentes conexos. Sua complexidade é de $O(V + E)$, onde V representa o número de vértices do nosso grafo e E o número de arestas.

2.2.3 Algoritmo de Camerini Modificado

Como dito anteriormente, modificamos o algoritmo de Camerini para computar a menor aresta máxima de um grafo, também conhecida como aresta *bottleneck*, ou gargalo.

O algoritmo consiste em primeiramente computar a mediana das arestas. Para tal, usaremos o algoritmo MoM Select, computando a mesma em tempo linear. Com ela em mãos, iremos gerar grafos induzidos A e B, tal que as arestas de B sejam menores que mediana, e as arestas de A o restante.

Com isso, verificamos se B é conexo. Caso for, chamamos a mesma função recursivamente agora para o grafo B. Caso contrário, criamos um grafo C que representará super vértices do grafo B. Em outras palavras, cada componente conexo de B será considerado como um

único vértice em C. As arestas do grafo C serão as arestas de A tal que ela não ligue dois componentes conexos do grafo B. Nessa parte, o nosso grafo C poderá acabar tendo arestas de pesos diferentes ligando os mesmos vértices.

Com o grafo C em mãos, chamamos a função recursivamente para ele. O algoritmo termina quando o número de arestas do grafo for igual a 1. Nesse caso, retornaremos o peso dessa aresta.

Obs.: Para um grafo onde temos todas as arestas iguais, percebemos que o código descrito acima entrará em loop. Para evitar tal problema, criei uma outra condição que verifica se o número de componentes conexos de B é o mesmo que o número de vértices do grafo de entrada. Ou seja, se nenhuma aresta foi inserida em B, temos que todas as arestas são iguais, visto que seria impossível termos arestas maiores ou iguais que a mediana mas não menores. Logo, retornaremos o peso de qualquer aresta do grafo de entrada.

3 Análise de Complexidade

3.1 Complexidade Temporal

Teremos que a complexidade temporal do algoritmo de Camerini será $O(E)$, onde E representa o número de arestas do nosso grafo inicial. Visto que é possível computar a mediana em tempo $O(E)$ usando o algoritmo de MoM Select; que a DFS terá complexidade $O(E)$ pois $E >> V$; e que, a cada chamada recursiva do nosso algoritmo para computar a aresta *bottleneck* estamos dividindo o número de arestas em pelo menos metade, concluimos que o algoritmo terá complexidade $O(E+\frac{E}{2}+\frac{E}{4}+\cdots+1) = O(E)$. Note que a complexidade para montar os grafos utilizados dentro da função será $O(\frac{E}{2^i})$, onde i representa a i-ésima chamada recursiva.

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 o número de vértices de E o número de arestas. No restante do programa, alocamos vetores e outras estruturas mas apenas levando em conta o número de arestas ou vértices apenas, resultando em complexidades $O(E)$ e $O(V)$. Logo, a primeira domina a complexidade espacial do nosso código. Como inicialmente o grafo é completo e não direcionado, isto é: $E = V*(V-1)$ (cada vértice está ligado a $V-1$ vértices), temos que a complexidade espacial do nosso código será na verdade: $O(V+E) = O(V+V*(V-1)) = O(V+(V^2-V)) = O(V^2)$.

4 Prova de Corretude

A prova de corretude consiste em basicamente 2 teoremas.

- Se B for uma árvore geradora de G, então a aresta *bottleneck* de G é dada pela aresta *bottleneck* de B.

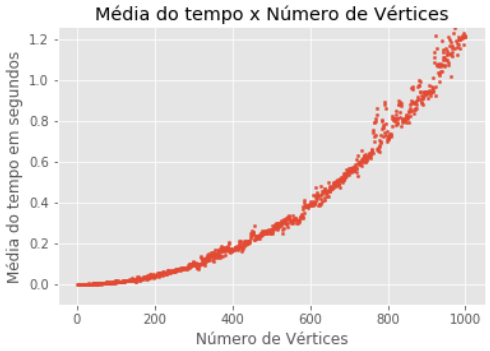
- Se B não for uma árvore geradora de G, então a aresta *bottleneck* de G é dada pela aresta *bottleneck* do grafo C, onde C representa as arestas de A ligadas aos super vértices de B.

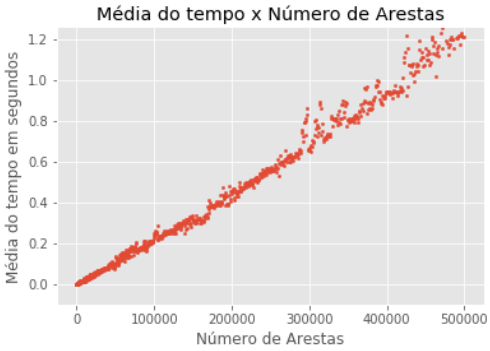
Esses teoremas se baseam no fato de que a aresta *bottleneck* faz parte de todas as possíveis árvores geradoras de um grafo, inclusive de uma MST.

O primeiro teorema fica claro que, caso B seja uma árvore geradora de G, ele deverá conter a aresta de interesse, já que ele contém assim o conjunto das menores arestas de G. Logo, basta executarmos o mesmo algoritmo dessa vez para o grafo induzido B.

Para provar o segundo teorema, suponha que a aresta de interesse esteja no grafo B, que por sua vez não representa uma árvore geradora de G, em outras palavras, é um grafo induzido desconexo. Por contradição, temos que se a aresta *bottleneck* faz parte do grafo induzido B, este por sua vez deveria ser uma árvore geradora de G. Com isso, teremos que inserir arestas (de peso maior ou igual a mediana) em B que conectam dois componentes conexos distintos, com o intuito de formarmos uma árvore geradora. Assim, teremos inserido nossa aresta *bottleneck* nesse novo grafo formado, no qual chamaremos a função recursiva até que sobre apenas 1 aresta no grafo.

5 Avaliação Experimental





Com base nos gráficos acima, temos que o nosso algoritmo se comporta de maneira quadrática no número de vértices, em outras palavras, tem uma complexidade assintótica de $O(V^2)$, onde V representa o número de vértices do nosso grafo; e possui um comportamento linear no número de arestas, em outras palavras, tem uma complexidade assintótica de $O(E)$, onde E representa o número de arestas do nosso grafo.

Para cada caso de teste, gerados aleatoriamente, foram executados 10 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 a aresta *bottleneck*.