Documentação TP2 - MBST

Isadora Alves de Salles

21/05/2019

1 Introdução

Este trabalho teve como objetivo encontrar a distância mínima para interligar vários telescópios por uma rede de comunicação, dada as posições de cada um deles na Terra, queremos que a ligação seja acíclica, ou seja, que só exista um caminho entre cada par de vértice. Portanto, o que queremos é minimizar a distância máxima necessária para conectar todos os pares de cidades, e isso foi modelado com grafos.

Traduzindo para o vocabulário de grafos, o que queremos é encontrar a menor aresta máxima para criar uma árvore geradora. A menor aresta máxima será nossa bottleneck que é também a maior aresta de uma MST. Uma MST é sempre uma MBST, ou seja, uma MST é um subconjunto da MBST. Sendo assim poderíamos implementar um algoritmo para retornar uma MST, como kruskal ou prim, e nossa bottleneck seria a maior aresta da árvore encontrada, porém a complexidade desse algoritmo seria $\mathcal{O}(m * logn)$ e desejamos encontrar a bottleneck em tempo linear para o número de arestas.

2 Solução do Problema

Para solucionar o problema da MBST em tempo tempo linear no número de arestas foi utilizada a ideia do algoritmo de Camerini, a intuição do mesmo está contida abaixo:

A intenção é encontrar a menor aresta máxima necessária para formar uma árvore geradora. Dessa forma, uma modificação pode ser feita no algoritmo de Camerini pois não precisamos retornar a árvore geradora, então não precisamos montá-la. Fora isso, o restante do algoritmo foi seguido.

O primeiro passo do algoritmo é formar um grafo induzido (que é um grafo com a mesma quantidade de vértices de G, porém com menos arestas) com metade das arestas do grafo original. Para fazer isso precisamos encontrar a mediana aproximada das arestas do grafo original, foi usado o algoritmo da mediana das medianas para garantir que a complexidade não fosse afetada. Dessa forma podemos garantir que as arestas que formarão nosso grafo induzido possuem peso inferior ao da mediana. Depois devemos verificar se esse grafo induzido ainda é conexo, para isso foi implementada uma DFS. A ideia é que se você tem um vetor em que todas as arestas menores que a mediana estão de um lado e as maiores do outro, e tudo que está do lado esquerdo da mediana conseguir formar um grafo conexo, sabemos que a bottleneck está nesse conjunto de valores, se não for conexo sabemos que a bottleneck está no conjunto acima da mediana.

Com o uso da DFS foi possível encontrar os componentes conexos do nosso grafo e colori-los da mesma cor, e mantendo um vetor com as cores de cada componente foi possível mapear os componentes conexos para supervértices. Se o vetor de cores estiver preenchido com o mesmo valor significa que o grafo é conexo. Quando temos um grafo desconexo (mais de uma cor), deixamos de criar um grafo induzido com as arestas a esquerda da mediana e passamos a criar um grafo com as arestas a direita da mediana e possuindo um conjunto de vértices em que cada vértice será na verdade um supervértice que é um componente conexo formado com as arestas inferiores a mediana. Assim todos os nós que possuem a mesma cor estão no mesmo componente e no novo grafo serão o mesmo nó. Dessa forma, no novo grafo poderemos ter mais de uma aresta

conectando dois nós, por esse motivo foi escolhido implementar o grafo com lista de adjacência e não matriz de adjacência.

Nos subgrafos o mesmo processo é executado novamente. O algoritmo é recursivo, e a cada iteração trabalhamos com um conjunto de arestas menor, o que possibilita uma complexidade $\mathcal{O}(E)$ ao final. A condição de parada é termos apenas uma aresta e essa é a que falta para manter o grafo conexo, e será nossa bottleneck. Na Figura 1 temos um pseudocódigo do algoritmo utilizado.

Algorithm 1 Compute a MBST of Graph G

```
1: procedure MBST(G, w)
2:     Let E be the set of edges of G
3:     if |E| = 1 then
4:         Return E;
5:     else
6:         A ← UH(E, w);
7:         B ← E − A
8:         F ← FOREST(G_B)
9:         Let η = {N_1, N_2, ....., N_C} where N_i(i = 1, 2, ....., c) is the set of nodes
   of the i − th component of F;
10:        if c = 1 then
11:            Return MBST(G_B, w);
12:        else
13:            Return F ∪ MBST((G_A)_η, w);
14:        end if
15:    end if
16: end procedure
```

Figura 1: Pseudocódigo para o algoritmo de Camerini.

3 Prova de corretude do algoritmo

Temos dois Teoremas:

1. Se GB é um grafo induzido conexo de G, então a MBST de G é também uma MBST para GB.

2. Se GB é um grafo induzido desconexo de G, então a MBST de G é dada pela MBST de GA, que é um grafo composto pelos componentes conexos de B e as arestas que foram descartadas.

Para o teorema 1 é fácil observar que se temos um grafo GB que contém todos os vértices do grafo G, porém com menos arestas, e esse grafo GB é conexo, isso implica que a menor aresta de peso máximo que deve estar na árvore geradora de G não é menor que a de GB, então a nossa bottleneck está em GB. Podemos induzir isso devido ao fato de que o GB foi formado apenas por arestas menores que a mediana do conjunto de arestas, dessa forma limitamos o conjunto de arestas de GB, e podemos afirmar que a bottleneck de G se encontra em GB.

Já para o teorema 2, tomamos o 1 como verdadeiro, e temos que se o grafo formado pelas arestas menores que a mediana for conexo nossa bottleneck também está contida nesse grafo. Porém, se esse grafo for desconexo, então nossa bottleneck deve estar no conjunto de arestas acima da mediana, temos os componentes conexos de GB e falta uma aresta para conectá-los, devemos escolher a aresta de menor peso dentre as que estão no corte entre os dois componentes conexos, porém como essa aresta tem peso acima das que foram adicionadas anteriormente, ela será a bottleneck.

- Prova(por contradição):

Vamos supor que o grafo induzido de GB não é conexo, sendo assim estamos no Teorema 2 e sabemos que a bottleneck não está em GB. Porém, suponhamos que esteja.

As arestas que compõem o grafo GB são arestas com peso menor ou igual a mediana das arestas. A bottleneck de uma árvore é a aresta de maior peso, se nosso grafo não é conexo significa que não conseguimos formar uma árvore geradora com as arestas dele e precisamos inserir mais arestas. As arestas que podemos inserir possuem peso superior as que já estão no grafo. Sendo assim, chegamos em uma contradição e provamos que a bottleneck, neste caso (Teorema 2), não está em GB.

4 Análise de Complexidade

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

4.1 Complexidade Assintótica de Tempo

O programa executa cinco partes principais: lê as entradas, cria grafos, calcula a mediana das medianas, executa uma DFS e executa o algoritmo de Camerini que chama algumas dessas funções já citadas. Vamos então analisar as complexidades separadamente:

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

Nesta etapa cada uma das coordenadas dos n vértices é lida em tempo $\mathcal{O}(1)$. Assim a complexidade é $\mathcal{O}(n)$.

4.1.2 Criar grafos: $\mathcal{O}(E)$

Para preencher o grafo precisamos calcular a distância entre cada vértice, utilizando a função "distanceEarthKm" que está em distance.c. Como o grafo é completo, o número de arestas é igual a n*(n-1)/2. Precisamos colocar cada uma das arestas no grafo, assim essa etapa tem custo linear no número de arestas $\mathcal{O}(E)$.

O mesmo ocorre na criação dos subgrafos no algoritmo de Camerini, a cada criação de grafos teremos custo linear no número de arestas do novo grafo sendo criado, porém neste caso os grafos terão cada vez menos arestas.

4.1.3 Mediana das medianas: $\mathcal{O}(E)$

O algoritmo da mediana das medianas retorna um valor aproximado da mediana, a cada chamada recursiva do algoritmo dividimos nosso vetor em vários grupos de 5 elementos, e para cada grupo de 5 elementos rodamos um mergeSorte, que tem complexidade $\mathcal{O}(m * logm)$, como temos poucos elementos sabemos que rodar um algoritmo de ordenação terá baixo custo. Assim, tiramos a mediana de cada grupo e fazemos uma chamada recursiva para tirar a mediana das medianas, paramos quando temos menos de 5 elementos do vetor e retornamos a mediana aproximada. Dessa forma, conseguimos obter a mediana de um vetor com custo linear no tamanho do vetor, que será igual a quantidade de arestas.

Para separar o vetor com o pivot, que será a mediana, no seu lugar certo, precisamos apenas de percorrer o vetor uma vez, tomando alguns cuidados quando temos mais de um elemento igual a mediana, dessa forma também teremos custo $\mathcal{O}(E)$ nessa etapa.

4.1.4 Busca em Profundidade (DFS): $\mathcal{O}(n + E)$

Para verificar se temos um grafo conexo é usada uma busca em profundidade, que tem o custo $\mathcal{O}(n+E)$, mas como sempre rodamos a DFS apenas em subgrafos do nosso grafo original, contendo

número de arestas igual a $E/2^i$ e o mesmo número de vértices do grafo original no máximo, as chamadas da DFS não causam problemas ao fato de que a implementação deve ter custo linear no número de arestas.

4.1.5 Camerini's algorithm: $\mathcal{O}(E)$

Todas as chamadas do algoritmo de Camerini requerem $\mathcal{O}(E/2^i)$ na iteração i, pois a cada iteração estamos partindo nosso vetor de arestas aproximadamente ao meio (na nossa mediana). Assim, a cada recursão as funções são chamadas para um conjunto de arestas que é a metade do anterior, e assim temos:
$\mathcal{O}(E + E/2 + E/4 + ... + 1) = \mathcal{O}(E)$
Essa complexidade é comprovada na análise experimental.

Assim a complexidade assintótica de tempo do programa é O(E).

4.2 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 3 variáveis na struct, que foi criada para facilitar as operações de encontrar a mediana das arestas e tem custo de espaço igual a $\mathcal{O}(3E)$. 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 em um grafo completo 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 se o vértice já foi visitado ou não, isso será usado para a busca em profundidade e tem custo no espaço igual a $\mathcal{O}(n)$.

No decorrer da execução do algoritmo recursivo serão alocados subgrafos, contendo as mesmas especificações acima, porém com custo espacial menor visto que o número de arestas e/ou de vértices dos subgrafos criados serão sempre menores que o do original.

Assim a complexidade assintótica de espaço do programa é
$\mathcal{O}(3E) + \mathcal{O}(n * (n-1)) + \mathcal{O}(n) = \mathcal{O}(n * (n-1)) = \mathcal{O}(n^2)$

5 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 com coordenadas sendo valores aleatórios. 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 devemos aumentar o número de arestas, como temos sempre um grafo completo como entrada do nosso algoritmo, uma forma de variar o número de arestas é variar o número de vértices. Sendo assim, foram gerados 500 casos de tese, variando o número de vértices do grafo, aumentando de 2 em 2, até 1000. 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 de quantidade de arestas por tempo de execução e outro de quantidade de vértices por tempo de execução, a fim de comprovar a complexidade do algoritmo.

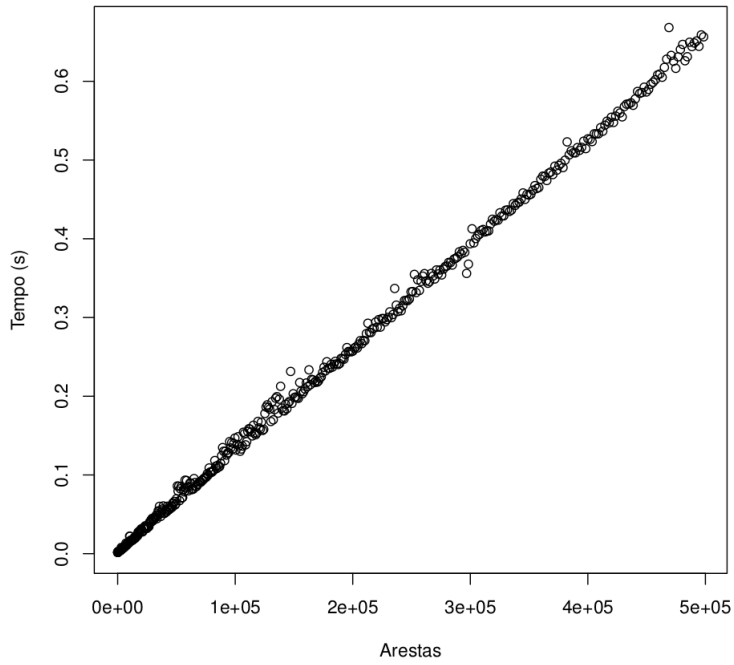

Figura 2: Tempo de execução do programa em número de arestas.

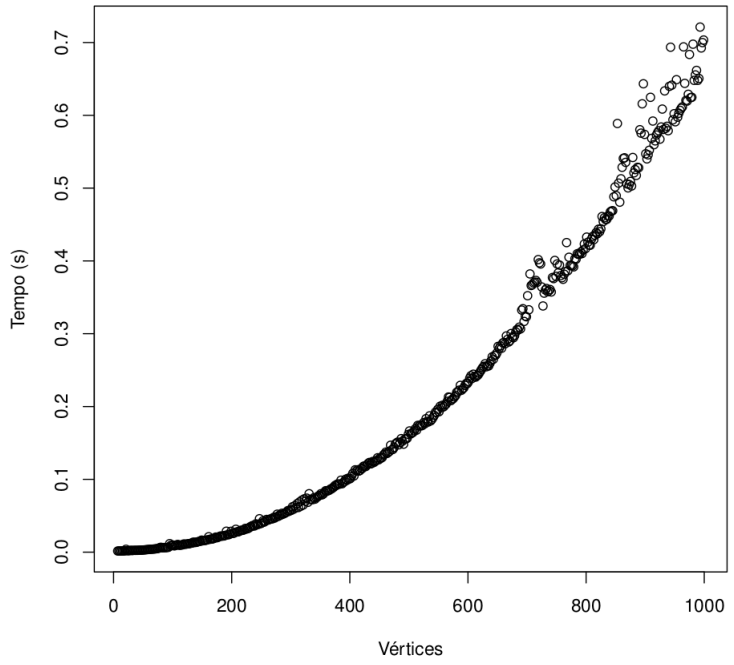

Figura 3: Tempo de execução do programa em número de vértices.

6 Conclusão

Neste trabalho o problema de encontrar a menor aresta máxima para uma árvore geradora foi resolvido em tempo linear no número de arestas ou quadrático no número de vértices, sendo modelado por um grafo e fazendo o uso do algoritmo de Camerini. A utilização de uma lista de adjacência foi escolhida, ao invés da matriz de adjacência, para economizar memória, dado que em determinado momento não teremos mais um grafo completo, além de que na solução implementada precisamos criar grafos com mais de uma aresta entre dois nós, o que faz com que o uso de uma lista de adjacência seja uma opção melhor.

Por fim, a complexidade foi ilustrada e verificada por meio da análise experimental, na qual verificou-se que o algoritmo tem um comportamento, de fato, linear no número de arestas.