Trabalho Prático 2

Thiago Felicíssimo César

1 Introdução

Neste trabalho prático, implementa-se um algorítimo que resolve o seguinte problema. Deseja-se conectar n cidades no globo terrestre por meio de comunicadores. Cada cidade deve receber um comunicador, que pode trocar dados com outro comunicador que esteja em uma distância máxima de d. Além disso, todos os comunicadores devem ser do mesmo tipo - isto é, ter a mesma distância máxima de comunicação - para poderem comunicar entre eles. O objetivo é então encontrar o menor valor de d que possibilite que todas as cidades sejam conectadas.

Observa-se que é possível reformular o problema enfrentado, sendo um enunciado mais sucinto o seguinte. Dado n pontos no globo terrestre, pede-se a menor distância d tal que exista uma árvore que ligue todos os pontos e cujas arestas sejam menores ou iguais a d.

A única restrição imposta para o algoritmo é que, considerando as cidades como vértices de um grafo completo, a complexidade de operações seja $O(m)$, onde m é o número de arestas (observa-se que, para um grafo completo, o tipo de grafo que modela o problema, essa complexidade é equivalente a $O(n^2)$).

2 Solução Proposta

Antes de tudo é interessante fazer algumas definições que serão úteis nesse trabalho.

Definição 2.1. Seja G um grafo conexo não-direcionado. Se S é o conjunto das árvores geradoras de G e e_k é uma aresta com peso w_k, denota-se por w_b o **gargalo** de G, definido como

$$w_b := \min_{T \in S} \max_{e_k \in T} w_k \ .$$

Em outras palavras, ao se considerar o conjunto que contém os pesos máximos de cada árvore de G, o gargalo é definido como o o menor elemento desse conjunto. Uma aresta de G que tem peso igual a w_b é chamada uma **aresta gargalo** do grafo G. Uma árvore geradora de G cuja maior aresta é uma aresta gargalo é denominada uma **árvore geradora de gargalo mínimo**.

Nota-se que achar o gargalo do grafo é precisamente o problema abordado neste trabalho. A definição acima simplesmente ajuda a encontrar em qual problema mais geral o deste trabalho se encaixa. Um algoritmo que resolve esse problema, e que portanto é um bom candidato para ser utilizado, é o proposto por Camerini, apresentado na subseção a seguir.

2.1 O Algoritmo de Camerini

Para apresentar o Algoritmo de Camerini, é interessante fazer uso de dois exemplos[1]. O primeiro exemplo do funcionamento do algoritmo é ilustrado na Figura 1.

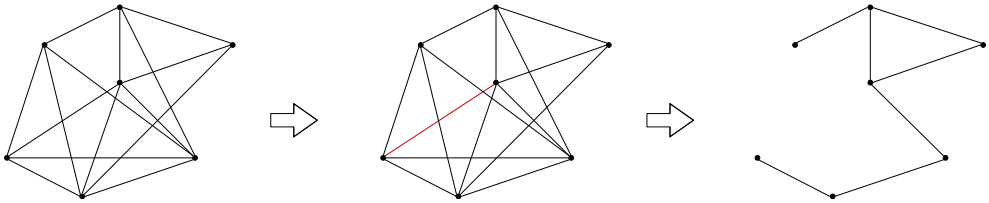

Figura 1: Exemplo 1 do funcionamento do algoritmo.

Inicialmente, o algoritmo busca qual a aresta mediana do grafo. Em seguida, todas as aresta de peso igual ou maior ao da mediana são deletadas do grafo. Como o grafo continua conexo, segue que nenhuma das arestas removidas eram arestas gargalo. Além disso, nota-se que uma aresta gargalo do grafo anterior também sera uma aresta gargalo do novo grafo.

No segundo exemplo, ilustrado na Figura 2, o mesmo procedimento é feito, mas chega-se em um grafo não conexo. Neste caso, sabe-se que a aresta (ou as arestas) gargalo foi deleta do grafo, e portanto nenhuma das arestas restantes é uma aresta gargalo - sendo possível descarta-las. O algoritmo então identifica as componentes desconexas e retorna para o grafo original.

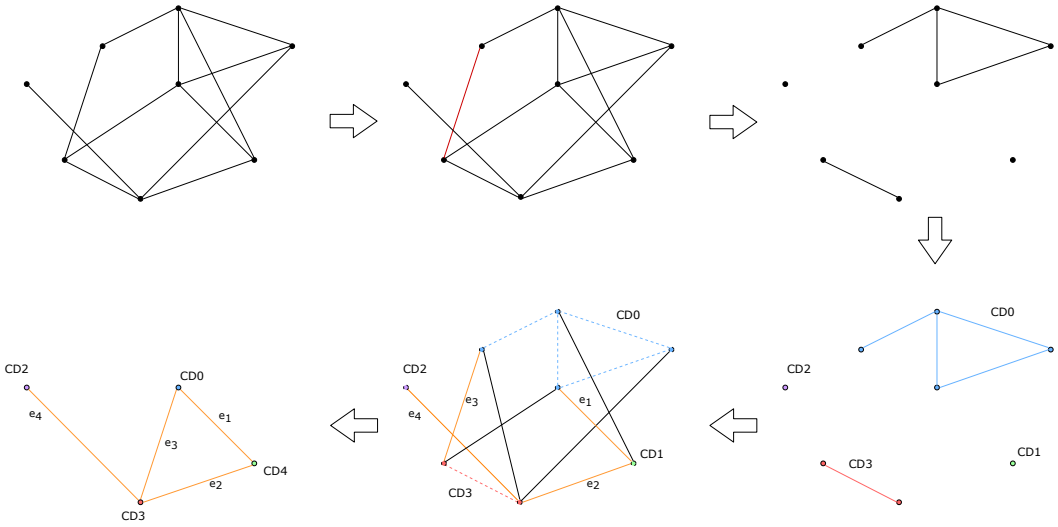

Figura 2: Exemplo 2 do funcionamento do algoritmo.

A ideia agora é considerar, para cada par de componentes desconexas, o conjunto das arestas que as unem e selecionar a aresta de menor peso (caso o conjunto seja vazio, nenhuma aresta é selecionada). É evidente que nenhuma das aresta que não foram selecionadas pode ser uma aresta gargalo - caso contrário, para cada par de vértices em que cada um se encontra em um componente, poderia-se construir um caminho que passa por uma aresta menor que o gargalo utilizando a aresta selecionada no passo anterior, um absurdo. O último passo é então colapsar cada componente em um único super-vértice e uni-los por meio da aresta selecionada

[1]Para um tratamento mais formal, recomenda-se a leitura do artigo *The min-max spanning tree problem and some extensions*, escrito por Camerini, no qual o algorítimo é apresentado de uma forma mais teórica.

no passo anterior. Observa-se que, assim como no primeiro exemplo, uma aresta gargalo do grafo anterior também será uma aresta gargalo do novo grafo.

Os dois exemplos dados ilustram como o algoritmo permite que, a cada iteração, no mínimo metade das arestas sejam descartadas. Além disso, o que é mais importante para comprovar a corretude do algoritmo é o fato do gargalo do grafo ser invariante pelo procedimento. Deste modo, percebe-se facilmente que para achar o gargalo basta executar o procedimento até que reste somente uma aresta, sendo o peso dela necessariamente o gargalo do grafo inicial.

Neste texto, também será mostrado que o custo dos procedimentos realizados a cada iteração são lineares com o número de arestas. Utilizando esse resultado, torna-se possível concluir que o custo computacional total será linear com o número de arestas, pois

$$O(m + \frac{m}{2} + \frac{m}{4} + ...) = O(m) \ .$$

3 Estruturas de Dados

Agora que o algoritmo utilizado já foi discutido de forma clara em alto nível, é necessário precisar os detalhes de como ele foi implementado. Todavia, antes de falar propriamente do algoritmo e de como as decisões são tomadas, faz-se necessário discutir as estruturas de dados implementadas.

3.1 Grafo

A Figura 3 ilustra a implementação da estrutura de dados que armazena os dados do grafo. Nela, é representado como exemplo um grafo de sete vértices.

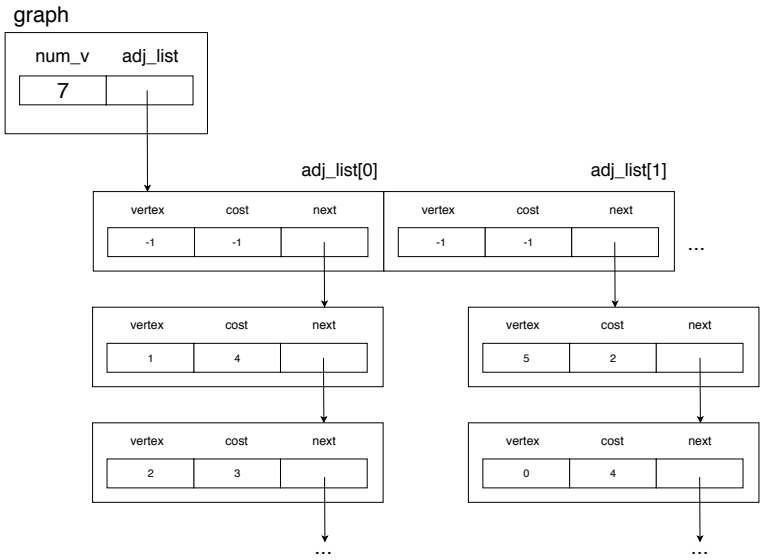

Figura 3: Estrutura de dados do grafo.

Como pode ser observado na imagem, dois tipos de *struct* são utilizados. Um deles, do tipo *graph*, representa o grafo todo e contém seu número de vértices e um ponteiro. Esse ponteiro aponta para um arranjo, alocado dinamicamente na inicialização do grafo, de estruturas do tipo *edge*, organizado de tal maneira que na posição i do arranjo se encontra a cabeça da lista que armazena as arestas incidentes ao vértice i. Cada célula guarda o peso associado a aresta, bem como o outro vértice ligado por ela (naturalmente, esses valores são inicializados como -1 para as cabeças). Ademais, como se trata de uma lista, é necessário armazenar em cada célula o ponteiro para a célula seguinte.

3.2 Cidades

A Figura 4 ilustra a implementação da estrutura de dados das cidades.

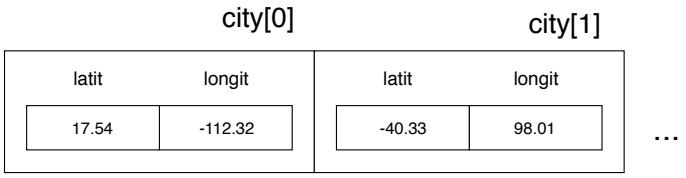

Figura 4: Estrutura de dados das cidades.

A estrutura de dados para as cidades é extremamente simples e só será útil durante a inicialização do programa, para guardar os dados de latitude e longitude das cidades. Em um segundo momento, essas cidades terão sido transformadas em vértices de um grafo e o peso associado a cada aresta terá sido calculado, sendo as informações sobre as coordenadas das cidades a partir desse ponto inúteis.

Como pode ser observado na imagem, a estrutura é formada por um arranjo, no qual cada posição é um *struct* que armazena as informações correspondentes a um determinada cidade. Essas informações dizem respeito a latitude e longetude, e são armazenadas em um tipo *double*.

4 Algoritmos

Inicialmente, é interessante discutir de forma mais aprofundada o algoritmo utilizado neste trabalho pratico, para depois discutir os procedimentos auxiliares que são utilizados.

Nesta seção, não serão discutidos trechos de código que executam a leitura da entrada e a inicialização das estruturas de dados, pois a implementação é bastante trivial. Todavia, é evidente que esses trechos serão considerado durante a análise de complexidade.

4.1 O Algoritmo Principal

O princípio de funcionamento do algoritmo principal, isto é, do Algoritmo de Camerini, já foi apresentado na Subseção 2.1. Deste modo, o foco aqui é formalizar o que é feito por meio de um pseudocódigo. O Algoritmo 1 descreve o procedimento feito.

Assim como explicado nos exemplos da Subseção 2.1, o algoritmo inicialmente procura a mediana do conjunto das arestas e em seguida faz uma copia do grafo original, retirando todas as arestas maiores que a mediana. Para verificar a conexidade do novo grafo, chama-se uma função que retorna as componentes desconexas

do novo grafo, representadas por meio de um vetor (na posição i encontra-se o vértice pai da componente desconexa que inclui o vértice i).

Caso o grafo G' continue conexo (ou seja, caso todas as posições de v guardem um mesmo número), deleta-se o grafo G e faz-se uma chamada recursiva em G', retornando o valor obtido. Caso contrário, ele é deletado e em seguida é chamada uma função em G que colapsa cada uma das componentes desconexas descritas pelo vetor v em um super-vértice. Em seguida, é feita uma chamada recursiva em G e seu valor é retornado. A condição de parada do algoritmo é quando o grafo analisado só tem uma aresta. Neste caso, retorna-se o peso desta aresta.

Algorithm 1 Implementação do algoritmo de Camerini.

```
E ← arestas(G)
se tamanho(E) = 1 então
    retorna E[0]
senão
    d ← mediana(E)
    G' ← retiraArestasMaiores(G, d)
    v ← pegarComponentesDesconexas(G')
    se todos os elementos de v são iguais então
        deleta(G)
        retorna gargalo(G')
    senão
        deleta(G')
        colapsaComponentes(G, v)
        retorna gargalo(G)
    fim
fim
```

4.2 Algoritmos Auxiliares

Não é o objetivo aqui explicar em detalhes o funcionamento de algoritmos auxiliares do código que já foram vistos em aula, como o algoritmo da mediana e algoritmos de busca em grafos. Também não serão explicados procedimentos cuja implementação é trivial, como o procedimento que deleta o grafo. Todavia, convém apresentar os procedimentos que são particulares para o problema atacado neste trabalho prático e cuja implementação envolve pontos mais interessantes.

4.2.1 Procedimento *retiraArestasMaiores*

Como visto, esse procedimento retorna uma copia do grafo original, retirando as arestas maiores que d. Uma maneira simples de fazer isso é inicializar um novo grafo vazio G' e iterar sobre a estrutura de dados do grafo G, copiando as arestas de G em G'. Em seguida, basta iterar agora sobre a estrutura de dados de G', retirando as arestas maiores que d.

4.2.2 Procedimento *pegarComponentesDesconexas*

Para a implementação deste procedimento, utilizou-se uma versão do algorítimo *Depth First Search (DFS)*, já visto em sala de aula. Sabe-se que o algoritmo *DFS* original retorna uma árvore formada pelos vértices

da componente desconexa correspondente ao primeiro vértice da busca. Todavia, a única informação necessária aqui é saber quais vértices estão na componente, portanto uma forma mais direta de armazenar essa informação é por meio de um arranjo, no qual cada posição corresponde a um vértice e seu conteúdo é o identificador da componente da qual ele faz parte.

Em um primeiro momento, cada vértice começa na sua própria componente. Com isso, pode-se escolher um vértice aleatório e chamar o algoritmo *DFS* nele. Se algum vértice não foi alcançado durante a busca (isto é, se algum outro vértice, diferente do escolhido no primeiro passo, ainda pertence a componente identificada por ele próprio), é feita uma outra chamada nele. Esse processo segue até que todo vértice tenha sido considerado em alguma busca.

4.2.3 Procedimento *colapsaComponentes*

Inicialmente, conta-se o número de componentes desconexas descritas pelo vetor v e inicializa-se uma matriz de arestas, de forma que cada posição nessa matriz corresponda a uma possível aresta do novo grafo. Um novo grafo $\hat{G}$ vazio é inicializado e, em seguida, passa-se a iterar sobre o grafo G.

Para cada uma das arestas de G, verifica-se, fazendo uso do vetor v, se ela conecta vértices de uma mesma componente. Caso isso se verifique, então a aresta é ignorada e passa-se para a próxima. Caso contrário, considera-se então a matriz descrita anteriormente. Caso a posição correspondente a aresta analisada ainda não tenha sido preenchida, então o valor de peso da aresta considerada é adicionado nessa posição. Caso essa posição já contenha um valor, só é adicionado se o valor da aresta considerada for menor que o valor na matriz. Dessa forma, de todas as arestas que ligam duas componentes, a menor aresta é a que vai ser selecionada para compor o novo grafo.

Para finalizar o procedimento, basta iterar sobre a matriz e, para cada posição que tenha sido preenchida, adiciona-se a aresta correspondente ao novo grafo. Em seguida, basta substituir o grafo original G por $\hat{G}$ e retornar.

5 Análise de Complexidade

5.1 Análise de Tempo

Inicialmente, o programa precisa ler os dados de cada cidade e inicializar a sua respectiva estrutura de dados, o que no total tem um custo computacional de $O(n)$, onde n é o número de cidades. Para construir o grafo é necessário uma operação para cada par de cidades, pois como o grafo é completo, é necessário calcular a distância entre cada par de cidades e em seguida inserir a aresta correspondente no grafo. Desta forma, a construção do grafo utiliza $O(n^2)$ operações.

Uma vez que todas as estruturas de dados já foram inicializadas, o passo seguinte é chamar o algoritmo principal, que calcula o gargalo do grafo. Como já mostrado na Subseção 2.1, se os procedimentos executados em uma iteração do algoritmo - com exceção, obviamente, da chamada recursiva - tiverem custo linear com o número de arestas em G, então o custo total de executar o algorítimo também será linear. Desta forma, basta provar que os procedimentos intermediários utilizados tem custo linear com o número de arestas em G que segue diretamente que o custo computacional é $O(m)$, sendo m o número de arestas.

O procedimento de obter as arestas do grafo e armazena-las em um vetor tem custo claramente linear com o número de arestas em G. Para o cálculo da mediana, foi utilizado o algoritmo visto em sala, que é linear

com o número de elementos do conjunto - nesse caso, o número de arestas em G. Para o procedimento *retiraArestasMaiores*, como só é necessário iterar sobre cada aresta de G uma vez, o custo computacional é claramente linear com o número de arestas.

No procedimento *pegarComponentesDesconexas*, primeiramente inicializa-se o vetor v, de forma que cada vértice começa sendo o único membro de sua componente, o que claramente tem custo $O(n')$, sendo n' o número de arestas em G'. Para o procedimento completo, pode-se afirmar que cada vértice do grafo G' é considerado apenas uma vez e que, para ele, o número de operações feitas é proporcional ao seu grau - uma vez que é necessário considerar cada aresta incidente a ele. Como o somatório dos graus é igual a duas vezes o número de arestas, a complexidade total do procedimento pode ser expressa como $O(m'+n')$, onde as m' é o número de arestas em G'. Aqui é importante notar que, como $n' = n$ e $m' < m$, então $O(m'+n') \subset O(m+n)$. Como G é conexo, vale que $m \geq n-1$, e portanto pode-se reescrever a complexidade do procedimento como sendo $O(m)$.

Deletar G claramente tem custo $O(m)$, uma vez que é necessário desalocar cada uma das arestas do grafo. Como o número de arestas em G' é menor que o número de arestas em G, segue que o deletar G' também é $O(m)$.

No procedimento *colapsaComponentes*, o primeiro passo é contar o número de componentes em v, que será o número de arestas no novo grafo. Como v tem uma posição para cada vértice de G, o custo é $O(n)$, mas como $m \geq n-1$, pode-se expressar o custo também como sendo $O(m)$. Para inicializar as posições da matriz de arestas como inválidas, o custo é claramente $O(m)$, uma vez que o novo grafo não pode ter mais arestas que G. Em seguida, itera-se sobre cada aresta de G para possivelmente adiciona-la na matriz, o que tem custo $O(m)$. Por fim, itera-se sobre a matriz de arestas para adiciona-las no novo grafo, com um custo de $O(m)$. Segue que o custo total deste procedimento é $O(m)$.

Desta forma, conclui-se que obter o gargalo tem custo linear com o número de arestas em G. Para finalizar o código, basta deletar o grafo, o que tem custo $O(m)$, e o código termina (seria necessário desalocar as cidades também, o que não aumentaria a complexidade pois seria o mesmo custo de inicializa-las, mas no código implementado elas são alocadas estaticamente, e portanto isso não é necessário).

Observa-se então que os procedimentos realizados durante o código tiveram custo de $O(m + n + n^2) = O(m + n^2)$. Como o grafo inicial é completo - isto é, todas as cidades começam interligadas - então vale que $m = \frac{n^2-n}{2} = O(n^2)$. Deste modo, a complexidade de tempo total do código desenvolvido neste trabalho pode ser expressa tanto como $O(m)$ ou como $O(n^2)$.

5.2 Análise de Espaço

Para a análise de espaço, primeiramente são consideradas as estruturas de dados alocadas em um primeiro momento, isto é, a estrutura das cidades e o grafo construído a partir dela. É evidente que o custo da estrutura de dados para as cidades é $O(n)$, enquanto que o grafo inicial tem custo de $O(m + n) = O(m)$. Agora basta considerar as estruturas alocadas durante a chamada do algoritmo de Camerini.

Para isso, é interessante observar que a cada chamada recursiva o espaço ocupado diminui, pois as estruturas de dados alocadas em uma iteração são sempre desalocadas antes da chamada recursiva, e também porque a cada iteração o número de arestas e/ou o número de vértices diminui. De fato, como o espaço alocado pelos procedimentos auxiliares é função dos números de vértices e arestas (ainda não foi mostrado a relação exata, mas é evidentemente uma função dessas variáveis), então o espaço alocado sempre diminuirá. A única exceção é o tamanho ocupado pelo *call stack*, que aumenta a cada chamada recursiva até a última - e cuja complexidade espacial é claramente $O(\log m)$. Com isso, basta analisar a complexidade espacial da primeira chamada recursiva que essa será a complexidade total do procedimento de achar o gargalo.

A alocação do vetor E tem um custo espacial $O(m)$, uma vez que o vetor tem uma posição para cada aresta. O procedimento *retiraArestasMaiores* tem custo espacial constante - uma vez que nenhuma variável nova é alocada, com exceção das variáveis utilizadas para controle de laços. O procedimento *pegarComponentes-Desconexas* tem custo de $O(n)$, uma vez que só é alocado um vetor, de tamanho igual a n. O procedimento *colapsaComponetes* tem custo $O(\hat{m})$ para a alocação da matriz, mais um custo de $O(\hat{n}+\hat{m})$ para a alocação do novo grafo, em que $\hat{n}$ e $\hat{m}$ são, respectivamente, o número de vértices e de arestas no grafo obtido ao colapsar os vértices de G. Como $n \geq \hat{n}$ e $m \geq \hat{m}$, pode-se reescrever a complexidade do procedimento como sendo $O(m+n)$.

Para analisar a complexidade espacial do algoritmo que calcula a mediana é necessário entrar em alguns detalhes da implementação. O primeiro deles é que quando o algoritmo termina de dividir o vetor, antes de fazer a chamada recursiva na parte que contém a mediana, as outras partes e os outros vetores auxiliares são deletados, portanto o espaço não vai se enchendo ao longo das recursões que dividem o vetor inicial. Sobre as recursões que acham a mediana das medianas, o código inicialmente precisa alocar um vetor de tamanho $\frac{m}{5}$ para armazenar as medianas e fazer a chamada recursiva. Só que para achar a mediana do vetor das medianas, também será feito o mesmo processo de pegar as medianas e jogar em um novo vetor, e assim será feito a cada recursão. Portanto, para achar a mediana das medianas de um vetor, as chamadas recursivas realizam uma alocação total de

$$O(\frac{m}{5}+\frac{m}{25}+\frac{m}{125}+...) = O(m)\ .$$

Segue que a complexidade espacial do procedimento que calcula a mediana é $O(m)$. Desta forma, a complexidade espacial total da primeira recursão do algoritmo de Camerini será $O(n+m)$. Pelo argumento feito anteriormente, a complexidade total de achar o gargalo será a complexidade da primeira recursão mais o custo do *call stack*, totalizando $O(m+n+\log m)$. Adicionando os custos iniciais de alocação das estruturas de dados, a complexidade não muda. Como $O(\log m) \subset O(m)$ e $m \geq n-1$, pode-se reescrever a complexidade como sendo $O(m)$.

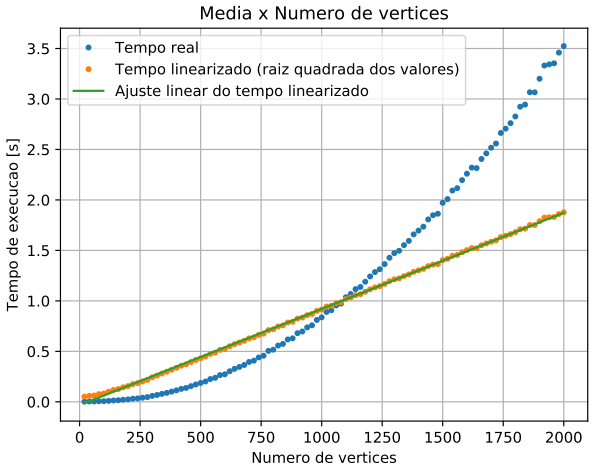

Figura 5: Médias dos tempo de execução.

6 Avaliação Estatística

Para a avaliação estatística, objetivou-se verificar o crescimento do tempo de execução com o valor do número de vértices na entrada. Essa análise permite também verificar a complexidade temporal do algoritmo.

Para isso, foi programado um gerador de entradas aleatórias para que o código deste trabalho pudesse ser rodado várias vezes e os tempos comparados. Para número de vértices de $20, 40, 60, ..., 2000$, foram geradas 10 entradas para cada tamanho. Em seguida, o código do trabalho foi posto para rodar com cada entrada e o tempo de execução foi cronometrado.

As Figuras 5 e 6 exibem, respectivamente, os valores da média e do desvio padrão quando se considera cada conjunto de tempos de execução associadas a um dado número de vértices na entrada.

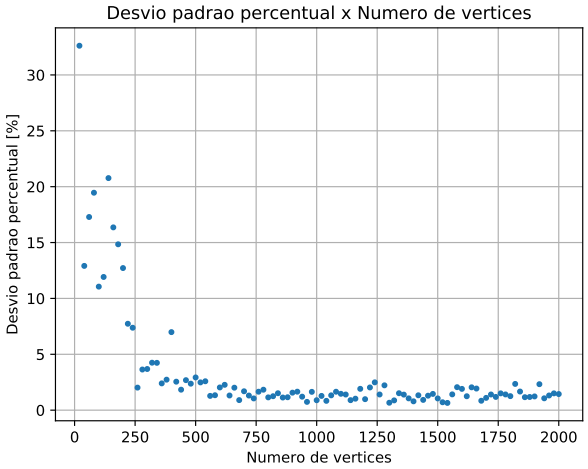

Figura 6: Desvios padrões dos tempos de execução.

Na Figura 5, exibe-se também o tempo linearizado (isso é, o gráfico das raízes quadradas de cada valor de tempo médio em função do número de vértices) bem como seu ajuste linear. Como é possível de se aferir visualmente, o tempo linearizado se comporta de forma extremamente linear, assim como esperado - uma vez que o código tem complexidade temporal de $O(n^2)$.

A Figura 6 confirma a validade das medições, principalmente quando o número de vértices é alto, uma vez que o desvio padrão percentual da grande maioria dos pontos exibidos no gráfico anterior é menor que 5%.

Nota-se nos gráficos que, quanto maior o número de vértices, mais os valores tendem a se estabilizar na curva esperado e menor tende a ser o desvio padrão. É possível especular que uma causa para isso viria do fato de que, quando a entrada é muito pequena, procedimentos constantes e possíveis ruídos na medição impactariam mais esta, uma vez que o tempo medido já é muito pequeno. Quando o tempo medido tem um valor alto, a medição tende a ser menos impactada por esses fatores, e portanto os pontos se comportam de forma mais previsível.

Caso deseje-se obter todos os dados gerados durante o procedimento de avaliação experimental, bem como os *scripts* utilizados, basta acessar o repositório `https://github.com/Thiagofelis/TP2_alg1`[2].

[2]Para garantir a impossibilidade de um outro aluno acessar o repositório e copiar o trabalho, ele só será aberto ao público uma semana após a data de entrega.

7 Conclusão

A utilização do Algoritmo de Carmerini para a resolução do problema deste trabalho prático mostra como vários problemas do mundo real podem ser modelados por meio de grafos, o que é mais uma evidência da importância da teoria dos grafos para um cientista da computação. Esse trabalho permitiu também entender como um grafo pode ser expresso por meio de estruturas de grafo e como algumas operações básicas podem ser implementadas.

A avaliação estatística também foi de grande interesse pois permitiu que se comprovasse a corretude da análise de complexidade temporal, além de confirmar que o algoritmo está de acordo com os requisitos de complexidade.