Universidade Federal de Minas Gerais

Projeto e Análise de Algoritmos
Mestrado em Ciência da Computação

# Trabalho Prático - GRAFOS

*Autor:*
Daniel Martins Reis

*Matrícula:*
2019697267

# 1 Introdução

O presente trabalho consiste em implementar um algoritmo que dado um conjunto de naves, representados por grafos (vértices e arestas), temos que identificar o tipo e o tempo de vantagem de cada nave. Cada aresta (u,v) representa um teletransporte na nave, ou seja, um caminho do vértice 'u' para o vértice 'v'.

Notações utilizadas:

- V representa a quantidade de vértices da nave;
- E representa a quantidade de arestas da nave;
- GrauMin representa o menor grau entre os vértices de uma nave;
- GrauMax representa o maior grau entre os vértices de uma nave;
- Vmingrau representa a quantidade de vértices de menor grau entre os vértices de uma nave;
- Vmaxgrau representa a quantidade de vértices de maior grau entre os vértices de uma nave.

Tipos de nave e suas caractéristicas:

- T (TRANSPORTADORA): Grafo com apenas um ciclo, em que o primeiro vértice é interligado ao último, gerando o formato de um retângulo;

  V = E, GrauMin = 2, GrauMax = 2.

- R (RECONHECIMENTO): Grafo acíclico que tem o formato de uma linha;

  V-1 = E, GrauMin = 1, GrauMax = 2.

- F (FRIGATA): Grafo que apresenta apenas uma aresta a mais que a quantidade de vértices.

  V-1 = E;

- B (BOMBARDEIRO): Grafo em que todos vértices do lado esquerdo se interligam com todos vértices do lado direito, classificado como bipartido.

  V = Vmingrau + Vmaxgrau, E = Vmingrau * Vmaxgrau.

Tempo de vantagem: Refere-se a soma do tempo de cada elemento da nave voltar para o seu respectivio lugar, ou seja, cada par de vértice (u,v), u ser trocado de lugar com v. Só que isso tem um custo, que é o custo do caminho entre os vértices (u e v). Portanto, o tempo de vantagem de uma nave é a soma dos custos de cada par de vértices (u,v) voltar para o seu lugar.

Dessa maneira, o que se busca obter de informação é a quantidade de naves de cada tipo na sequência (R, F, B, T) e o menor tempo de vantagem entre elas.

Segue uma breve descrição das Estruturas de Dados utilizadas:

- Pilha -> Espécie de Lista LIFO (Last in first out) que armazena apenas um número inteiro em cada célula;
- Lista de inteiros -> Espécie de vetor alocado dinâmicamente, de forma que ao inserir é testado se ainda existe espaço em relação ao tamanho máximo. Caso não exista, aloca-se mais 10 espaços de memória e insere o elemento. Caso exista, apenas insere o elemento na lista;
- Lista de naves -> Espécie de Lista Duplamente Encadeada para armazenar os dados de uma nave (Lista de inteiros de componentes conexas; Tipo; Quantidade de vértices; Quantidade de arestas; Maior grau; Menor grau; Quantidade de vértices de maior grau; Quantidade de vértices de menor grau);
- Adjacente -> Célula utilizada para compor uma lista de adjacência, em que armazena (Vértice 'w' adjacente a 'v'; referência do próximo vértice adjacente);
- Vértice -> Célula utilizada para compor uma lista de vértices, em que armazena (Valor do vértice; Vértice no qual este vértice deve ser trocado; Cor do vértice; Distância do vértice até alguém; Pai do vértice; Grau do vértice; Nave na qual o vértice está inserido);
- Grafo -> Célula que armazenada os dados de um grafo (Total de vértices; Total de arestas; Lista de vértices (monta uma lista com cada célula 'Vértice', interligando as referências de próximo); Lista de adjacências (monta uma lista com cada célula 'Adjacente', interligando as referências de próximo); Lista de naves).

Portanto, o grafo é representado como uma lista de vértices, em que cada vértice tem a referência da sua respecitiva lista de adjacências. Ressalta-se que tanto a lista de vértices como a lista de adjacências de cada vértice são formadas adicionando uma célula utilizando a referência de próximo (prox) em cada célula. A pilha que armazena apenas um número (valor do vértice) é utilizada pelo algoritmo de busca em profundidade para controlar os vértices a serem percorridos.

Segue abaixo algumas convenções utilizados nos pseudocódigos:

- N representa a quantidade de vértices do arquivo;
- M representa a quantidade de arestas do arquivo;
- u representa um vértice;

- v representa um vértice;
- (u,v) representa uma aresta entre os vértices u e v;
- minTV representa o menor valor encontrado até o momento para o tempo de vantagem.

Descreve-se o pseudocódigo implementado:

```
Execucao:
    Leitura da primeira linha do arquivo: N, M
    'InicializaGrafo' (N, M)
    Loop (Leitura das proximas M linhas do arquivo):
        Leitura de cada linha -> u,v
        'InsereAresta' '(u,v)' e 'InsereAresta' '(v, u)'
        Incrementa o 'grau' de v e de u
    Loop (Leitura das proximas N linhas do arquivo):
        Leitura de cada linha -> u,v
        Insere no vertice u o valor de v no campo 'troca', ou seja
            o vertice v que deve trocar de lugar com u
    'DFSComponenteConexa' que separa o grafo em 'naves' e
        'classifica' cada nave de acordo com seu respectivo 'tipo'
    'TempoDeVantagem' calcula o menor tempo de vantagem entre todas
        naves do grafo
```

O procedimento InicializaGrafo basicamente cria o espaço de memória para o vetor de vértices e para a lista de adjacência de cada vértice no vetor. Dessa forma, já realiza o procedimento de inicialização dos vértices requeridos pelo DFS => setar o pai do vértice como -1, a cor como BRANCA, o grau como ZERO e a distância como ZERO.

O procedimento InsereAresta apenas realiza a alocação de um espaço de memória na lista de adjacências de um vértice e insere nele o valor do vértice adjacente.

O procedimento DFSComponenteConexa funciona com um DFS (Busca em Profundidade) modificado de maneira a criar uma nave e inserir nela todos os vértices acessíveis a partir de um vértice origem 's'. Dessa maneira, pega um vértice BRANCO, marca de CINZA e explora a sua lista de adjacências. Nesse processo de explorar a lista de adjacências, passa os vértices adjacentes BRANCOs para CINZA e insere na pilha para poder explorá-lo, além de setar a sua distância, adicioná-lo a nave, contabilizar o menor e maior grau da nave, e se ele é um vértice de menor/maior grau da nave. Por fim, colore o vértice 'u' de PRETO. É importante ressaltar que o

procedimento de inicializar os vértice na cor BRANCA já é realizado logo na inserção do vértice, como mencionado anteriormente. Portanto, esse procedimento não é realizado de novo.

```
DFSComponenteConexa:
    Insere o vertice 's' na LIFO, ou seja, a origem na pilha
    Loop (Percorre a lista de vertices 's')
        Verifica se o vertice 's' tem cor diferente de 'PRETA'
            Cria um nave
            Insere o vertice 's' na nave, marca de 'CINZA' e insere
                o vertice 's' na pilha
            Enquanto tiver elementos na pilha
                Remove elemento do topo (vertice 'u')
                Loop (Percorre a lista de adjacencias de 'u')
                    Se vertice 'a' adjacente a 'u' e da cor 'BRANCA'
                        Marca vertice 'a' de 'CINZA'
                        Seta pai 'a' como 'u'
                        Seta distancia de 'a' como distancia de 'u'+1
                        Adiciona 'a' na componente conexa da 'nave'
                        Incrementa contador de vertices da 'nave'
                        Marca no vertice 'a' a 'nave' que ele esta
                        Contabiliza 'menor' grau da 'nave'
                        Contabiliza quantidade de vertices com o
                            'menor' grau da 'nave'
                        Contabiliza 'maior' grau da 'nave'
                        Contabiliza quantidade de vertices com o
                            'maior' grau da 'nave'
                Marca o vertice 'u' de 'PRETO'
             Divide por 2 o total de 'arestas' da 'nave' para
                considerar apenas u->v/v->u e nao ambos
    Loop (Percorre cada nave)
        Verifica quantidade de 'arestas', 'vertices', 'grau minimo',
            'grau maximo', quantidade de 'vertices de grau minimo' e
            quantidade de 'vertices de grau maximo' e classifica
            cada nave conforme o descrito na secao 'Introducao'
```

O procedimento TempoDeVantagem calcula o menor tempo de vantagem entre as naves. O vértice 'troca' referido no algoritmo, se refere ao vértice lido no arquivo em que deve ser permutado de posição com o vértice 's'. Vale ressaltar, que o DFS utilizado pela função 'TempoDeVantagem', é um DFS na sua implementação

original, que finaliza a busca ao encontrar o elemento procurado.

```
TempoDeVantagem
    Inicializa 'minTV' = -1, ou seja, como uma valor invalido
    Loop (Percorre cada nave da lista de naves)
        Inicializa 'soma' = 0, sendo esse o valor total do tempo de
            vantagem da nave
        Loop (Percorre cada vertice 's' da componente conexa da nave)
            Se vertice 's' diferente do vertice 'troca' computa a
                distancia entre 's' e 'troca'
                Se nave tipo 'T' ou 'B'
                    Incrementa soma com 'DFS' entre ('s' e 'troca')
                Se nave de outro tipo, calcula o LCA
                    Se nave tipo 'R'
                        O ancestral em comum entre os dois pode ser
                            considerado o menor entre eles. Calcula
                            a distancia entre eles e incrementa em
                            soma
                    Se nave tipo 'F'
                        Busca um ancestral em comum entre os dois
                            vertices e calcula a distancia entre eles
                            e incrementa em somas
                Incrementa o contador total do tempo de vantagem da
                    nave denominado 'soma' com o valor da 'distancia'
                    obtido
                Se apos calcular o valor do tempo de vantagem entre
                    mais dois elementos da nave e encontrar um valor
                    maior que o minimo, finalizo o loop, pois essa
                    nave nao pode ser a menor
        Se apos calcular o valor do tempo de vantagem total de uma
            nave e seu valor e menor que o 'minTV' que se conhecia
            antes, atualiza o valor de 'minTV'
        Se apos calcular o valor do tempo de vantagem total de uma
            nave e seu valor igual a 'zero', finaliza a execucao,
            pois foi encontrado o menor valor possivel de tempo de
            vantagem
    Retorna 'minTV', ou seja, o menor tempo de vantagem entre todas
        naves
```

Com o intuito de tornar o cálculo do TempoDeVantagem mais rápido e menos cus-

toso, o que se faz é calcular o tempo de vantagem da primeira nave e ter ele como uma referência de 'minTV', ou seja, de menor tempo de vantagem computado até o momento. A medida que se encontra uma nave com tempo de vantagem inferior, atualiza-se o valor de 'minTV', e caso encontrar uma nave com tempo de vantagem igual a zero, finaliza-se a execução, visto que nenhuma nave poderá ter um tempo de vantagem menor do zero, isso porque um tempo de vantagem igual a 0 corresponde a todos elementos já estarem nos seus devidos lugares.

O cálculo da distância entre dois vértices é feito de maneiras diferentes para tipos de naves diferentes.

- T (TRANSPORTADORA) / B (BOMBARDEIRO): é computado um DFS (Busca em Profundidade) entre os dois vértices;
- R (RECONHECIMENTO): é calculado o LCA (Menor antecessor em comum) dos vértices. Como a nave desse tipo tem um formato de uma linha, ou seja, um vértice e adjacente a um próximo e um anterior, o LCA entre eles é o vértice de menor distância até a origem entre os dois. Então a distância entre os dois vértices corresponde a: d1 + d2 - 2 * dlca, em que:

  d1 = distância do vértice 'a' até a origem

  d2 = distância do vértice 'b' até a origem

  dlca = distância do menor vértice antecessor em comum até a origem.
- F (FRIGATA): enquanto não encontrar um ancestral em comum entre os vértices 'a' e 'b' e 'a' ou 'b' não chegar na raiz, busca o LCA (Menor antecessor em comum). Assim, se o vértice 'a' tiver numa profundidade maior que o vértice 'b', avança o vértice 'a' para o pai de 'a', e se o vértice 'b' tiver numa profundidade maior que vértice 'a', avança vértice 'b' para o pai de 'b'. Então a distância entre os dois vértices corresponde a: d1 + d2 - 2 * dlca, em que:

  d1 = distância do vértice 'a' até a origem

  d2 = distância do vértice 'b' até a origem

  dlca = distância do menor vértice antecessor em comum até a origem.

# 2 Análise de Complexidade

**InicializaGrafo** => Nesse procedimento, é realizado a leitura e inserção dos vértices e arestas, onde é feito a alocação de cada espaço de memória. Logo na inserção de cada vértice já realizamos o processo de inicialização do vértice na cor BRANCA. Portanto, tem complexidade **O(V)**, considerando V vértices e E arestas, pois o espaço alocado para cada vértice da lista de adjacência é criado no procidento InsereAresta.

**InsereAresta** => Nesse procedimento, é realizado a alocação de um espaço de memória na lista de adjacências de um vértice e inserido nele o seu valor. Portanto, tem complexidade **O(1)**.

**DFSComponenteConexa** => Nesse procedimento cada vértice é percorrido apenas uma vez, visto que apenas selecionamos vértices na cor BRANCA para explorar a sua lista de adjacências, e ao explorar toda a sua lista de adjacências, colorimos o vértice de PRETO. Nesse processo, cada vértice adjacente da cor BRANCA é analisado. Portanto analisamos para todos os V vértices as suas arestas adjacentes, o que no final contabiliza uma complexidade **O(E)**, visto que analisamos todas as arestas do grafo G. Na sua versão original, o DFS tem complexidade O(V+E), sendo V devido ao processo de inicialização dos vértices com a cor BRANCA, que na verdade já foi realizado na inicialização do GRAFO.

**DFS** => Nessa função, é realizado um DFS original apenas com uma alteração: quando encontra o vértice 'v' procurado, finaliza a busca e retorna o valor da sua distância. Nesse processo, ao realizar uma busca em profundidade (u,v), ou seja, partindo de um vértice 'u' até um vértice 'v' com o intuito de encontrar o menor caminho entre ambos, o vértice objetivo pode ser exatamente o último a ser procurado. Portanto, tem complexidade **O(V+E)**.

**InsereListaDeInteiro** => Lista dinâmica utilizada para armazenar a referência dos vértices da componente conexa de uma nave. Essa função apenas verifica se a quantidade de espaços alocados é suficiente e insere o valor. Caso não seja suficiente, executa um 'realloc' utilizando 'incremento' posições de aumento, que nesse caso foi padronizado em '10' unidades. Portanto, tem complexidade **O(1)**.

**TempoDeVantagem** => Ao realizar o cálculo do tempo de vantagem de cada nave, o que se faz é calcular o tempo de vantagem de uma primeira nave, e a partir dele ter uma concepção de um valor mínimo de tempo de vantagem. Caso começar a computar o tempo de vantagem de uma nave e esse valor por algum momento ultrapassar a conecepção de mínimo que se tem, para de calcular o tempo de vantagem dessa nave, visto que o objetivo do problema é encontrar o menor tempo de vantagem entre as naves. Por outro lado, caso compute o tempo de vantagem de

uma nave e ele seja menor que a concepção de mínimo que se tinha até o momento, esse 'conecepção de mínimo' é atualizada. Então no pior caso nenhuma nave tem tempo de vantagem igual a zero, a primeira nave tem o maior tempo de vantagem e a última nave tem o menor tempo de vantagem, fazendo com que seja calcula o tempo de vantagem de todas naves, ou seja, é realizado o cálculo da distância das E arestas, portanto **O(E)**. Supondo que toda aresta (u,v) deve ser trocada, ou seja, a 'u' seja diferente de 'troca', temos que trocar E arestas de lugar. Para calcular o tempo de vantagem:

- Se a nave for do tipo T (TRANSPORTADORA) / B (BOMBARDEIRO), temos a complexidade **O(V+E)**, devido ao DFS mencionado anteriormente;
- Se a nave for do tipo R (RECONHECIMENTO), temos a complexidade **O(1)**, pois apenas consultamos os valores de d1, d2, definimos o menor como LCA e consultamos a distância de lca em dlca, para então executamos a fórmula: d1 + d2 - 2 * dlca;
- Se a nave for do tipo F (FRIGATA), enquanto não encontrar um ancestral em comum entre os vértices 'a' e 'b' e 'a' ou 'b' não chegar na raiz, busca o LCA (Menor antecessor em comum). No pior caso, percorremos as E arestas da nave. Portanto **O(E)**.

**Execução** => Na execução do algoritmo que dado várias naves, identifica a quantidade de nave de cada tipo e o menor tempo de vantagem entre elas, temos:

- InicializaGrafo => Complexidade **O(V)**
- DFSComponenteConexa => Complexidade **O(E)**;
- TempoDeVantagem => No pior caso, todas naves são do tipo T (TRANSPORTADORA) / B (BOMBARDEIRO), tornando a complexidade O(V+2E) = **O(V+E)**. No melhor caso, todas naves são do tipo R (RECONHECIMENTO), tornando a complexidade O(1+E) = **O(E)**.

Totalizando: **O(V+E)**.

# 3 Análise Experimental

Como executar?

```
./compilar.sh entrada saida
```

Especificações da máquina utilizada:

- Intel(R) Core(TM) i7-6700 CPU @ 3.40GHz;
- 16 GB de RAM;
- 64 bits;
- Linux grande 4.4.0-146-generic

  #172-Ubuntu SMP x86_64 x86_64 x86_64 GNU/Linux

Tabela 1: Dados da execução

| Arquivo | Vértices | Arestas | R | F | B | T | Tempo de vantagem |
|---|---|---|---|---|---|---|---|
| pdf1 | 15 | 14 | 1 | 1 | 1 | 0 | 4 |
| pdf2 | 19 | 18 | 1 | 0 | 0 | 2 | 0 |
| 1 | 165 | 546 | 1 | 2 | 1 | 1 | 21 |
| 2 | 21 | 20 | 1 | 1 | 1 | 1 | 3 |
| 3 | 999 | 992 | 4 | 5 | 1 | 0 | 4 |
| 4 | 2435 | 10000 | 11 | 21 | 12 | 5 | 3 |
| 5 | 10000 | 9524 | 554 | 578 | 379 | 282 | 0 |
| 6 | 11823 | 100000 | 26 | 11 | 13 | 8 | 19 |
| 7 | 42758 | 100000 | 435 | 475 | 411 | 233 | 0 |
| 8 | 22436 | 100000 | 129 | 109 | 126 | 68 | 2 |
| 9 | 36407 | 1000000 | 17 | 23 | 20 | 14 | 5 |
| 10 | 100000 | 95626 | 5475 | 5620 | 3953 | 2847 | 0 |
| 25 | 25 | 20 | 5 | 0 | 0 | 0 | 1 |
| 250 | 250 | 245 | 5 | 0 | 0 | 0 | 381 |
| 2500 | 2500 | 2495 | 5 | 0 | 0 | 0 | 40782 |
| 25000 | 25000 | 24995 | 5 | 0 | 0 | 0 | 4084373 |
| tree | 100000 | 99999 | 1 | 0 | 0 | 0 | 49999 |

Tabela 2: Dados da execução

| Arquivo | Real | Usuário | Sistema | Alocação | free (bytes) | Alocados (bytes) |
|---|---|---|---|---|---|---|
| pdf1 | 0,009 | 0,000 | 0,000 | 125 | 16 | 103232 |
| pdf2 | 0,006 | 0,000 | 0,000 | 128 | 18 | 103424 |
| 1 | 0,007 | 0,000 | 0,000 | 2312 | 208 | 154360 |
| 2 | 0,006 | 0,000 | 0,000 | 169 | 24 | 104472 |
| 3 | 0,008 | 0,000 | 0,000 | 5154 | 370 | 308544 |
| 4 | 0,019 | 0,004 | 0,000 | 38459 | 2596 | 994408 |
| 5 | 0,024 | 0,004 | 0,000 | 56406 | 5597 | 1801208 |
| 6 | 0,118 | 0,064 | 0,000 | 287784 | 9185 | 6609320 |
| 7 | 0,170 | 0,080 | 0,012 | 358719 | 35480 | 9219760 |
| 8 | 0,144 | 0,092 | 0,004 | 300117 | 20215 | 6933000 |
| 9 | 0,816 | 0,656 | 0,032 | 3188737 | 595755 | 67077192 |
| 10 | 0,296 | 0,128 | 0,016 | 562847 | 55915 | 17080160 |
| 25 | 0,006 | 0,000 | 0,000 | 150 | 10 | 104288 |
| 250 | 0,006 | 0,000 | 0,000 | 1295 | 30 | 139488 |
| 2500 | 0,011 | 0,000 | 0,000 | 12770 | 255 | 715488 |
| 25000 | 0,065 | 0,032 | 0,000 | 127520 | 2505 | 28750488 |
| tree | 0,252 | 0,124 | 0,012 | 510016 | 60005 | 2014700168 |

Nota-se que todas as instâncias utilizadas nos experimentos executaram num tempo inferior a 1 segundo. Dentre as instâncias utilizadas, temos desde valores de tempo de vantagem 0 até valores muito grandes, 0 ou mais naves de cada tipo e uma quantia bem pequena de vértices e arestas até uma quantia próxima do limite estipulado no trabalho. Os dados de consumo de memória foram coletados utilizando o 'valgrind'. É importante ressaltar que o algoritmo atende ao limite de 2 segundos de execução especificado para as referidas instâncias, de maneira que as naves e o menor de tempo de vantagem entre as naves foram classificadas corretamente se comparado as respostas do problema.