DOCUMENTAÇÃO – TRABALHO PRÁTICO 02

Nome: Lucca Silva Medeiros

Matrícula: 2019054773

Disciplina: Algoritmos I

Introdução:

O trabalho prático consiste em desenvolver um programa, capaz de processar um conjunto numérico de entrada, que representa a lista de rotas áreas de uma companhia de aviação, e dar como saída o número mínimo de rotas adicionais capaz de permitir viagens entre quaisquer aeroportos.

Modelagem computacional do problema:

Para resolver o problema proposto utilizei a modelagem em grafos, mais especificamente grafos direcionados e não ponderados, onde cada vértice representa um aeroporto, as arestas as rotas aéreas e a direção da aresta indica o sentido da rota aérea existente. Para resolver o problema não foi necessário ponderar as arestas.

Pensando agora no objetivo do problema, que é identificar o número mínimo de rotas que precisam ser adicionadas em uma malha aérea para permitir que um determinado aeroporto de origem consiga atingir todos os demais aeroportos, o que preciso fazer é encontrar o número mínimo de arestas adicionais para que meu grafo fique fortemente conectado. Para isso, dividi a solução em três passos:

1 - Identificar todos os componentes fortemente conectados do grafo;

2 - Considerar cada componente fortemente conectado como um novo vértice e representar em um novo grafo apenas os vértices que ligam componentes distintos;

3 - Verificar nesse novo grafo quantos vértices possuem grau de saída e grau de entrada igual a 0, ou seja, quantos vértices possuem nenhuma aresta entrando e quantos vértices possuem nenhuma aresta saindo.

Com isso o número mínimo de arestas adicionais para que meu grafo fique fortemente conectado será igual ao maior entre: o número de vértices possuem grau de saída igual a zero e o número de vértices que possuem grau de entrada igual a 0.

Estruturas de Dados:

Foi utilizada a seguinte estrutura de dado para solução do problema:

Graph: representa a malha aérea da companhia, modelada em grafo.

Atributos: int V , list <int> adj*

Obs: O grafo foi representado por filas onde V é o número de vértices do grafo e cada vértice possui um fila para representar os vértices adjacentes, ou seja, as arestas existentes.

Métodos:

Construtor: cria um grafo com o número de vértices passado por parâmetro;

Edge: adiciona no grafo a aresta passada por parâmetro;

StackDFS: primeira DFS realizada pelo método de Kosaraju, armazenando o caminhamento em uma pilha;

DFS: segunda DFS realizada pelo método de Kosaraju, realizada no grafo transposto, armazenando os componentes fortemente conectados (SCC) em um vetor;

Kosaraju: algoritmo de Kosaraju, responsável por encontrar todos os SCC do grafo;

DAG: cria um grafo acíclico dirigido (DAG) como representação do grafo original, onde os vértices de um SCC são representados como um vértice apenas e adiciona no novo grafo apenas as arestas entre SCC’s diferentes;

getGroup: retorna qual o grupo de SCC o vértice passado por parâmetro se encontra;

getMaxDegree0: contabiliza quantos vértices no DAG tem grau de saída e de entrada iguais a zero e retorna o maior entre eles, ou seja, o número de arestas mínimas para que o grafo original seja fortemente conectado.

Algoritmos utilizados e Complexidade Assintótica de Tempo:

O programa começa com a leitura do arquivo de entrada. Haja vista que **m** é o número de arestas e **n** o número de vértices a complexidade assintótica de tempo dessa parte do programa é **O (m)**.

Após a construção do grafo com os dados de entrada, complexidade **O (1)** , se inicia o processo de encontrar os SCC's. Os algoritmos clássicos utilizados para solução desse problema foram DFS e Kosaraju. A DFS é a busca em profundidade, algoritmo que visita todos os vértices do grafo começando pelo mais profundo, complexidade **O (m + n)** . Já o Kosaraju é o algoritmo responsável por encontrar todos os componentes fortemente conectados do grafo, seguindo a lógica do pseudo-código abaixo:

```
Escolhe qualquer vértice V do Grafo;
Roda DFS a partir do vértice V;
Acha o grafo reverso ( basta reverter a direção de cada aresta do grafo original );
Roda DFS a partir do vértice V no grafo reverso;
Adiciona no vetor todos os SCC, que foram alcançados em ambas as execuções DFS;
```

Após a execução do Kosaraju, que possui complexidade **O (m + n)**, temos todos os SCC armazenados em um vetor único, onde todos os números adjacentes representam um conjunto fortemente conectado e o número " -1 " representa um separador entre os grupos. Agora é possível criar o DAG seguindo a seguinte lógica:

```
Conta-se quantos componentes fortemente conectados X tem o grafo original;
Cria um novo grafo com X vértices;
Para cada ligação presente no grafo original:
        Se a ligação for entre o mesmo grupo fortemente conectado:
                Não representa a ligação no novo grafo;
        Caso for em entre grupos diferentes:
                Representa a ligação no novo grafo;
```

A complexidade assintótica de tempo no pior caso é **O (m*x)**, sendo **x** igual ao número de elementos no meu vetor de SCC. Isso acontece pois para cada aresta existente no meu grafo original eu verifico em qual grupo de SCC o vértice se encontra.

Com o DAG pronto falta somente identificar quantos vértices tem grau de saída e de entrada iguais a zero e retorna o maior entre eles. Para isso estruturei a seguinte lógica, com complexidade assintótica de tempo no pior caso de **O (n*n)**:

```
//Para identificar grau de saída 0
Para cada vértice do DAG:
        Se sua lista de adjacência estiver vazia:
                Contar +1 em grau de saída igual a 0;
```

//Para identificar grau de entrada 0
Para cada ligação do DAG:
    Identificar os vértices de destino;

Para cada vértice do DAG:
    Para cada número de vértices de destino:
        Se o vértice não for identificado:
            Contar +1 em grau de entrada igual a 0;

Retornar maior número entre grau de entrada ou saída igual a 0;

Haja vista que **m** é o número de arestas e **x** o número de elementos no meu vetor de SCC, a complexidade assintótica de tempo geral do programa é igual a **O(m*x).**