Documentação do trabalho prático 2 da disciplina de Algoritmos 1

Caio César Silva - 2019075681

Departamento de Ciência da Computação – Universidade Federal de Minas Gerais (UFMG) Belo Horizonte – MG – Brasil

caiocs2019@ufmg.br

1. Modelagem computacional

No projeto, foi criada apenas uma classe, de forma que ela tivesse sua responsabilidade. A classe de abstração e de regras de negócio é a classe **Grafo**, nela, modelei os aeroportos e as rotas de forma que cada aeroporto é um nó e a rota é um vértice no grafo, que por sinal, é um grafo direcionado. Nesta classe realizo todo o processamento e comparação de dados, todas as operações são chamadas no arquivo Main.

A organização das pastas e do Makefile estão na forma como nos foi ensinado na disciplina de Estrutura de Dados. Os headers das classes ficam na pasta “include”, a implementação de cada classe fica pasta “src”, quando as classes são compiladas, os arquivos com formato “.o” ficam na pasta “obj”. Na pasta “bin” fica o arquivo gerado para execução (formato “.out”) do programa, assim como deve ficar o arquivo de entrada.

Cada classe possui atributos públicos e privados, de forma que para acessar ou alterar um atributo privado, existe uma função getter ou setter para isso.

2. Instruções de compilação e execução

Configurações do meu computador e do compilador:

- Sistema Operacional: Windows 10 Pro x64
- Linguagem de programação utilizada: C++
- Compilador utilizado: g++ através do terminal WSL com Ubuntu instalado, utilizando o Makefile fornecido pela equipe da disciplina de Estrutura de Dados e alterado para as solicitações do trabalho atual.
- Processador/RAM: Intel Core i3-9100F 3.60GHz / 8GB RAM

Instruções de execução:

- Acesse o diretório raíz do projeto, nele, haverão as pastas **bin**, **include**, **obj**, **src** e o arquivo **Makefile**.
- Utilizando um terminal Linux, certifique-se de que você tenha o make e o g++ instalado, caso não, rode os seguintes comandos:
  - sudo apt-get install build-essential
  - sudo apt-get install make
  - sudo apt-get install g++
- Com o make instalado, rode o seguinte comando:
  - make
- Dessa forma, os arquivos com formato “.o” serão gerados na pasta **obj** e o arquivo **tp02.out** será gerado na pasta **bin**.
- Agora, navegue a até a pasta **bin** usando o comando:
  - cd bin

- Coloque seu arquivo de entrada nesta pasta, este arquivo deve estar no formato **“.txt”**.
- Com os arquivos na pasta **bin** rode o comando:
    - ./tp02.out < nomedoarquivo.txt
    - Onde o parâmetro **nomedoarquivo.txt** seria o nome do seu arquivo, acompanhado do formato dele (txt).
- Com isso, os saída do projeto será impressa no terminal,
- Caso queira utilizar outros arquivos de entrada é só substituir pelos que estão na pasta bin e rodar o comando acima, trocando o nome do arquivo no parâmetro do comando, se necessário.
- Caso ocorra qualquer problema com sua execução, entre em contato comigo pelo e-mail: caiocs2019@ufmg.br

3. Estrutura de dados e algoritmos

Explicando a classe **Grafo**, ela possui atributos como: quantidade de aeroportos, a lista de rotas identificadas pelo ID de cada aeroporto, a mesma lista de rotas transpostas, a lista de componentes conectados encontrados no mapa de aeroportos e rotas, e dois vetores indicando o grau de entrada e saída de componentes.

Iniciamos a aplicação lendo o arquivo através do método **ProcessaEntrada**, que irá chamar o método **CriaAeroportos** e **AdicionaRota**. O grafo está estruturado de maneira que seja representado por uma matriz de duas dimensões, em que cada vetor possui um vetor que representa seus nós adjacentes. Em cada posição da matriz está um número inteiro que representa o ID do aeroporto, este é lido no momento em que está sendo processada a entrada, por exemplo, se houver 10 aeroportos no arquivo, os IDs serão preenchidos na ordem de 1 a 10. O método **AdicionaRota** adiciona o aeroporto na segunda dimensão do array caso haja seja um adjacente dele.

Após isso, é executado o algoritmo de **Kosaraju** (que também é um método) para identificar os componentes conectados e preencher no atributo **ComponentesConectados**, que também é uma matriz de duas dimensões, onde cada vetor é um componente, e cada componente é um vetor de nós que compõem ele. O algoritmo de Kosaraju funciona da seguinte maneira:

Kosaraju():
Iniciar *pilha P*
Iniciar *vetor de booleanos Explorados[]*
Para *cada aeroporto i do grafo* ***faça****:*
Defina *Explorados[i] como falso*
FimPara
Para *cada aeroporto i do grafo* ***faça****:*
Se *Explorados[i] = falso* ***faça****:*
DFSTempo(*Explorados, P, i****)***
FimSe
FimPara
TransporRotas()
Para *cada aeroporto i do grafo* ***faça****:*
Defina *Explorados[i] como falso*
FimPara
Inicia *contador j como 0*

```
        Enquanto P não for vazio faça:
            Remove elemento do topo de P e atribui a AeroportoId
            Se Explorados[AeroportoId] = falso faça:
                Cria posição e atribui vetor vazio no vetor ComponentesConectados
                Cria posição e atribui 0 no vetor GrauEntradaComponentes
                Cria posição e atribui 0 no vetor GrauSaidaComponentes
                DFSComponentesContectados(Explorados, AeroportoId, j)
                j = j + 1
            FimSe
        FimEnquanto
FimKosaraju
```

A complexidade assintótica deste método deriva de ambos DFS (**DFSTempo** e **DFSComponentesConectados**) executados dentro dele, pois a complexidade destes métodos é de **O(A²)**, onde A é o número de aeroportos (nós).

O método **DFSTempo** irá fazer a busca em profundidade no grafo e calcular os tempos de término de cada nó, preenchendo a pilha com os nós de acordo com este tempo, a implementação escolhida é recursiva. Ela funciona da seguinte forma:

```
DFSTempo(vetor Explorados, pilha P, inteiro i):
        Explorados[i] = verdadeiro
        Para cada nó adjacente de i definido como u faça:
            Se Explorados[u] = falso faça:
                DFSTempo(Explorados, P, u)
            FimSe
        FimPara
        Empilha i em P
FimDFSTempo
```

Este método nos dá uma complexidade de **O(A²)** onde A é o número de aeroportos (nós), uma vez que o pior caso vai ser onde todo nó terá como adjacente todo outro nó e si mesmo, com isto terá que percorrer uma matriz AxA.

Após isto, realizamos a transposição das rotas usando o método **TransporRotas()**, que significa inverter a origem e o destino das rotas, preenchendo a matriz **ListaRotasAeroportosIdTransposta**, que funciona da seguinte forma:

```
TransporRotas:
        Para cada aeroporto Origem em ListaRotasAeroportosId faça:
            Para cada aeroporto Destino em ListaRotasAeroportosId[Origem] faça:
                Adiciona Origem em ListaRotasAeroportosIdTransposta[Destino]
            FimPara
        FimPara
FimTransporRotas
```

Este método possui uma complexidade assintótica de tempo de **O(A²)** onde A é o número de aeroportos (nós). Esta complexidade deriva da iteração ser aninhada, e estamos iterando em um vetor de duas dimensões.

Após isto executamos o algoritmo **DFSComponentesConectados**, que a única diferença do **DFSTempo** é que preenchemos a matriz de **ComponentesConectados**, fora da iteração neste método. A complexidade ainda é a mesma, $O(A^2)$, pelo mesmo motivo do **DFSTempo**.

Com isto, temos os componentes conectados identificados em uma matriz, portanto, podemos calcular o número mínimo de vértices a partir do momento em que definimos que estes componentes serão vértices em um novo grafo, uma vez que não faria sentido adicionar rotas dentro de componentes conectados. Isto nos dará um grafo condensado, que também será um DAG (Grafo Direcionado Acíclico), ou seja, um grafo que não há ciclos. Uma vez que este grafo não irá possuir ciclos, para encontrarmos um grafo em que de cada aeroporto (nó), conseguimos chegar em qualquer outro, devemos adicionar o número de rotas necessárias para transformar este DAG em um ciclo. A solução já foi simplificada ao criar o grafo condensado, agora, devemos encontrar o que é maior, o número de componentes com grau de entrada ou saída igual a zero, o maior dentre estes dois será a solução para nosso projeto.
Logo, temos o método **VerificaGrauSaidaEntradaComponentes** e **VerificaGrauEntradaComponentes** que realiza o cálculo destes graus de entrada e sáida, e funciona da seguinte forma:

```
VerificaGrauSaidaEntradaComponentes():
    Para cada componente c em ComponentesConectados faça:
        Para cada aeroporto a em ComponentesConectados[c] faça:
            Para cada aeroporto b em ListaRotasAeroportos[a] faça:
                Se b não existe em ComponentesConectados[c] faça:
                    //Isto significa que b é adjacente de a mas não é parte
                    //do componente, portanto, devemos incrementar o
                    //grau
                    Incrementa GrauSaidaComponente[c]
                    VerificaGrauEntradaComponentes(b)
                FimSe
            FimPara
        FimPara
    FimPara
FimVerificaGrauSaidaEntradaComponentes

VerificaGrauEntradaComponentes(inteiro destinoId):
    Para cada componente c em ComponentesConectados faça:
        Para cada aeroporto a em ComponentesConectados[c] faça:
            Se a = destinoId faça:
                Incrementa GrauEntradaComponente[c]
            FimSe
        FimPara
    FimPara
FimVerificaGrauEntradaComponentes
```

Portanto, temos que a complexidade destes dois métodos é de **O(((C * A) * A) * C)**, pois, para cada componente C, iremos iterar pela lista de Aeroportos A, e para cada um destes

Aeroportos, eu irei iterar para cada aeroporto em cada componente novamente, resultando na equação explicitada acima.

Após calcular os graus, chamamos o método **VerificaAdicaoMinima**, que basicamente irá realizar a somatória do número de componentes com grau de entrada ou saída igual a 0, e irá imprimir o maior entre eles. Sendo da seguinte forma:

```
VerificaAdicaoMinima():
    Inicia TotalEntrada = 0, TotalSaida = 0
    Para cada componente i em GrauEntradaComponentes faça:
        Se GrauEntradaComponentes[i] = 0 faça:
            TotalEntrada = TotalEntrada + 1
        FimSe
        Se GrauSaidaComponentes[i] = 0 faça:
            TotalSaida = TotalSaida + 1
        FimSe
    FimPara
    Se TotalEntrada > TotalSaida faça:
        Imprime TotalEntrada
    SeNão
        Imprime TotalSaida
    FimSe
FimVerificaAdicaoMinima
```

Com isso, temos que a complexidade assintótica geral da aplicação é de
O(((C * A) * A) * C)
pois é a função com maior complexidade da nossa solução.