Documentação do Segundo Trabalho Prático de Algoritmos I.

Marcelo Mascarenhas Ribeiro de Araújo
2019110053

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

mascarenhassribeiro@gmail.com

1.Introdução:

Essa documentação tem a finalidade de apresentar diretrizes a serem seguidas na execução do programa, além de fornecer uma breve explicação da implementação realizada, em concomitância com uma análise de complexidade das principais funções utilizadas para a solução do problema proposto.

O projeto trata da implementação de um sistema responsável por calcular o número mínimo de rotas que deverá ser adicionado á uma rede aérea para fazer com que um dado aeroporto de origem possa atingir qualquer aeroporto de destino, dado que muitas rotas utilizadas anteriormente foram restringidas devido à crise sanitária global. Ou seja, o problema consiste basicamente em calcular o número mínimo de arestas para transformar um grafo direcionado e não ponderado em um grafo fortemente conectado.

Nesse sentido, o projeto modela o problema computacional descrito e apresenta uma solução viável, que recebe os parâmetros através de um arquivo .txt, e realiza a leitura e a execução seguindo a lógica proposta na apresentação dos requerimentos do projeto.

Para informações mais detalhadas sobre o projeto, consulte as seções a seguir, sendo que:

- A seção 2 trata sobre as decisões tomadas para a resolução do problema proposto, bem como alguns detalhes de implementação, além do ambiente computacional que o trabalho foi desenvolvido.
- A seção 3 informa as instruções de compilação e execução.
- A seção 4 aborda uma breve análise de complexidade, de tempo e espaço, das principais funções e métodos utilizados no programa.
- A seção 5 conclui a documentação, fornecendo alguns detalhes adicionais sobre o processo de desenvolvimento e considerações em relação à solução proposta.

2.Implementação:

O programa foi desenvolvido e testado nas seguintes especificações:

-Linguagem: C++;

-Compilador: g++;

-OS e Versão do sistema: Linux Mint, versão 21.04 (Uma).

-Configuração do computador: AMD Ryzen 5 3600, 16GB RAM.

2.1. Modelagem Computacional e Técnica de Resolução

O problema requerido na proposta do projeto foi resolvido por Eswaran, K. e Tarjan, R. em 1976, e é denominado de 'Strong Connectivity Augmentation Problem'[1], ou 'Problema do Aumento de Componentes Fortemente Conectados'. Os pesquisadores tentaram achar uma solução para o problema que consistia em, dado um grafo direcionado e não ponderado, como achar e quanto é o número mínimo de arestas que deveria ser adicionadas no grafo a fim de torná-lo fortemente conectado. Eles observaram que esse número era relacionado aos componentes fortemente conectados do grafo que seguiam alguns padrões, e poderia ser calculado como **max(s+q, t+q)**, tal que **s** é um componente fortemente conectado(CFC) do grafo que tem ao menos uma aresta de saída e nenhuma aresta de entrada, **t** é um CFC que possui ao menos uma aresta de entrada e nenhuma aresta de saída, e **q** é um CFC que não possui nenhuma aresta de entrada e nenhuma aresta de saída. Note que CFC que possuem tanto ao menos uma aresta de saída e uma de entrada não são contabilizados. A solução funciona pois s+q arestas precisam ser adicionadas para conectar os CFCs de saídas(s) e os CFCs isolados(q), formando um ciclo entre os CFCs, tornando o grafo globalmente fortemente conectado. A mesma explicação vale para t+q, caso hajam mais CFCs de entrada(t) do que de saída(s). Note também que os CFCs (q) sempre são contabilizados, pois eles estão isolados e arestas necessariamente precisam ser adicionadas conectando outros CFCs a eles, a fim de tornar o grafo fortemente conectado. Para mais informações, incluindo a demonstração matemática da solução funciona, consulte a referência no rodapé da página.
Dito isso, muitas das ideias utilizadas por eles também foram utilizadas na resolução do presente problema.

Desse modo, a seguinte solução foi adotada para resolver o trabalho: primeiro, foi implementado um grafo, através de listas de adjacência, para modelar a malha aérea passada através do arquivo de entrada. Depois, foi calculado todos os componentes fortemente conectados do grafo, utilizado o algoritmo de Kosaraju[2]. Após esse processo, foi construído um grafo auxiliar, composto do condensamento do grafo original, i.e, um grafo direcionado e acíclico onde cada vértice é um componente fortemente conectado, e cada aresta representa as conexões de um CFC com outro CFC. Após esse processo, foram calculados os CFCs que correspondem a **s, t e q**, como descrito acima, e o número foi calculado.

O código está organizado em 6 diferentes arquivos, sendo eles:

Nota: Serão omitidos os parâmetros recebidos e os tipos retornados, salvo por quando forem citados na descrição, para fins de simplificação. Para mais detalhes, ou em caso de dúvidas, consulte o código que acompanha a documentação.

Main.cpp: Programa principal, responsável por declarar as classes, chamar o método de leitura do arquivos e preencher o grafo, chamar o método de kosaraju para o grafo lido, o método para calcular o número mínimo de arestas para tornar o grafo fortemente conectado, e imprimir a resposta.

Functions.hpp/Functions.cpp: Classe que possui o método de leitura de arquivos e preenchimento do grafo de acordo com a entrada fornecida. O único método da classe é:

```
void Functions::readInput(Graph &grafo)
```

Recebe um grafo como parâmetro, e basicamente aloca os vértices e as arestas lidas no grafo recebido.

Vertex.hpp/Vertex.hpp: Classe base que possui informações sobre cada nó do grafo, como o id do nó, número de arestas emissoras e receptoras, id_do_grupo(para identificar qual CFC que o vértice faz parte) e um vetor de adjacência, que representa os nós vizinhos do vértice em questão. Só possui métodos construtores, setters e getters. É considerado 'friend class' da classe graph.

Graph.hpp/Graph.cpp: Classe principal do programa. Possui métodos de inicialização do grafo, adicionar nós, arestas, dentre outros. Possui 3 atributos principais. Um deles é um responsável por armazenar o grafo

1 https://epubs.siam.org/doi/10.1137/0205044
2 https://www.ime.usp.br/~pf/algoritmos_para_grafos/aulas/kosaraju.html

em si. Os outros dois atributos são vetores, que guardam o grafo reverso e os componentes fortemente conectados do grafo, servindo como auxiliadores na execução de alguns métodos importantes que serão citados abaixo.
Segue uma descrição dos principais métodos:

```
void kosaraju();
```

Método que acha os componentes fortemente conectados de um grafo. Seu funcionamento basicamente consiste na aplicação de dois DFSs num grafo direcionado (note que não faz sentido falar de componentes fortemente conectados num grafo não-direcionado), Primeiramente é aplicado o DFS no grafo original, e conforme o algoritmo vai achando os nós, vai colocando os mesmos em uma pilha. Ao final, a função chama uma função que reverte o grafo original, e chama o DFS nos vértices que estão na pilha mas com as arestas reversas, e os nós vão sendo marcados como explorados durante o processo. Dessa forma, se existir um caminho de um vértice A até outros vértices nesse segundo DFS, isso implica que o vértice A e esses outros vértices citados anteriormente formam um componente fortemente conectado. Ao final, uma referência aos componentes fortemente conectados são salvos em um vetor, que é um atributo da classe..

```
int minAugmPath();
```

Função que calcula o número mínimo de arestas que devem ser inseridas para tornar o grafo fortemente conexo e retorna esse valor. Ele faz isso primeiro atribuindo os valores ao atributo group_id dos vértices do grafo original, para identificação. Depois, cria um segundo grafo auxiliar(condensado), onde cada componente fortemente conectado se transforma em um vértice, e calcula as arestas emissoras e receptoras desses vértices(ou seja, dos CFCs). Após essa etapa, o novo grafo está condensado, e o algoritmo passa por todos os vértices do novo grafo contabilizando quantos vértices são do tipo **s,q,t.** Depois isso, desaloca esse novo grafo, e retorna o valor máximo entre **(s+q, q+t)**
OBS: Além disso, a classe grafo conta também com 3 funções dfsUtil, que são basicamente as funções alicerçantes do DFS. Entretanto, elas realizam operações um pouco diferentes, já que cada uma recebe uma estrutura de dados diferente, e portanto os métodos e onde manipulá-las são diferentes. Entretanto, todas ainda tem a mesma função base de percorrer o grafo.

2.2. Estruturas de Dados Utilizadas:

Nesse trabalho, foram utilizadas 3 estruturas de dados básicas, VECTOR, SET e STACK. Embora seja possível representar os vizinhos de um vértice da classe Vértice com uma lista, pode também ser utilizado um vector sem maiores prejuízos, especialmente pois existe a possibilidade de manipular elementos específicos a custos constantes, sem precisar percorrer de fato a lista de vértices para selecioná-lo. O Stack foi utilizado durante o algoritmo de Kosaraju, conforme foi explicado acima. E o SET foi utilizado para encontrar as arestas emissoras e receptoras de um grafo condensado, evitando de contabilizar duas ou mais arestas de um CFC para um outro mesmo CFC.

2.3.Pseudo Código:

```
Begin:
        Initialize a graph. Fullfil the graph with the entry edges and nodes.
        Apply Kosaraju's algorithm in the graph to find the strong connected components(SCC)
        Condentase the SCC in an auxiliar graph
        Apply a DFS in the condensated graph to find the incoming and outgoing edges of each node.
        Initialize s, t, q equals to 0.
        For each Node in Condensated_Graph:
                If Node_outgoing_edges == 0 and Node_incoming_edges > 1:
                        t++
                Else if  Node_outgoing_edges > 0 and Node_incoming_edges == 0:
                        s++
                Else if Node_outgoing_edges == 0 and Node_incoming_edges == 0:
                        q++
        Return max(s+q,t+q)
End
```

3. Instruções de compilação e execução:

1. Realize a extração do zip para um diretório de sua preferência.

2. Acesse o diretório onde está o makefile, abra o terminal no diretório, e execute o comando make (Caso o comando não seja reconhecido no Windows, é necessário instalá-lo, através do pacote “Chocolatey”). Um executável com o nome “tp02” irá ser criado na pasta do makefile.

3. Abra o terminal.

No Windows:

- Execute o programa digitando o nome do executável, em concomitância com o caminho completo do arquivo que possui as informações das rotas aéreas, com aspas, e o passando para a entrada padrão com o operador “<”. Veja o exemplo:

tp02 < “C:/Users/Marcelo/OneDrive/Documentos/Faculdade Stuff/TP2_ALG/input.txt”

No Linux:

- Execute o programa digitando o nome do executável precedido de um ./ , em concomitância com o caminho completo do arquivo que possui as informações das rotas aéreas, e o passando para a entrada padrão com o operador “<”. Veja o exemplo:

./tp02 < /home/mmra/Documents/Faculdade/Algoritmos_I/Trabalho/input.txt

4. Análise de Complexidade:

Segue uma breve análise de tempo e espaço das principais funçõesa utilizadas no programa:

```
void Functions::readInput(Graph &grafo)
```

Complexidade de tempo: A função realiza operações constantes, e lê um arquivo, através da entrada padrão, de n linhas. Tendo isso em vista, podemos classificar o custo assintótico de tempo dessa função como **O(n).**

Complexidade de espaço: A função basicamente preenche o grafo, que utiliza a representação através de ‘listas de adjacência’, com o número de vértices e arestas passadas na entrada. Dessa forma, podemos concluir que o custo assintótica de espaço é **O(|V|+|E|).**

```
void Graph::kosaraju()
```

Complexidade de tempo: Como o algoritmo de Kosaraju consiste basicamente em aplicar o DFS duas vezes, uma no grafo original, e um num grafo reverso – cujo custo para construção é **O(|V|+|E|)** - , e ainda considerando que o DFS tem custo assintótico de **O(|V|+|E|)**, o método basicamente roda diversas operações(um número constante de vezes) cujo custo é **O(|V|+|E|).** Portanto, além das operações constantes, sabemos que a complexidade assintótica dessa função é **O(|V|+|E|).**

Complexidade de espaço: O método utiliza uma pilha como estrutura auxiliar, e no primeiro DFS adiciona todos os nós para serem percorridos depois no grafo reverso. Dessa forma, o custo da pilha é **O(|V|).** Além disso, o método também constrói um grafo reverso auxiliar, cujo custo de espaço é o mesmo que o grafo original. Dessa forma, podemos dizer que a complexidade de espaço desse algoritmo também é **O(|V|+|E|).**

```
int Graph::minAugmPath(){
```

Complexidade de tempo: Esse método consiste, primeiro, na atribuição de group_ids para cada vértice no grafo original, para facilitar a identificação dos componentes fortemente conectados, a partir da referência criada e alocada no atributo auxiliar da classe 'strong_conn_comps'. Isso possui complexidade **O(|V|)**, tendo em vista que passa em cada referência e atribui no vértice seu agrupamento. Depois disso, é criado um novo grafo condensado, onde cada vértice do grafo é um CFC no original, cuja complexidade total ao decorrer da função é **O(|V|+|E|)**. Dessa forma, é feito um novo DFS no grafo original, mas partindo de cada CFC para achar as arestas emissoras e receptoras do mesmo. Esse processo também possui complexidade **O(|V|+|E|)**. Após isso, é feito o percorrimento no novo grafo condensado a fim de contabilizar os vértices do tipo 's', 'q' e 't' supracitados. Essa operação também tem complexidade **O(|V|)**, tendo em vista que o número de arestas emissoras e receptoras estão armazenadas em um atributo do vértice. Dessa forma, a complexidade de tempo total do método pode ser descrita como **O(|V|+|E|).**

Complexidade de espaço: Como a função aloca um grafo auxiliar condensado, que é representado também por listas de adjacência, podemos denotar a complexidade de espaço da função também como **O(|V|+|E|).**

```
int main(int argc, char* argv[])
```

***Complexidade de tempo*:** O main possui basicamente declaração das classes, e chamada dos métodos 'readInput', 'kosaraju' e 'minAugmPath'. Dessa forma, herda a complexidade do método que possui o maior custo assintótico, que nesse caso é linear, da forma **O(|V|+|E|).**

Complexidade de espaço: Herda a complexidade de espaço do maior método, que também é **O(|V|+|E|).**

5. Conclusão:

Considerando todas as informações supracitadas, o problema apresentado, que consistia no desenvolvimento de um sistema que achasse o número mínimo de rotas a serem adicionadas em uma malha aéreo, tal que todas as rotas ficassem interconectadas, foi resolvido de maneira satisfatória. Não foram detectados erros nos testes realizados, seja de execução – considerando tanto os criados pelo desenvolvedor, quanto os passados pela equipe de monitoria -, quanto em testes de memórias – o programa foi submetido à testes com a ferramenta 'Valgrind' antes da entrega -. Além disso, a resolução se demonstrou extremamente modularizada e não tão custosa, apesar de provavelmente existirem projetos com uma eficiência maior.

Referências:

Chaimowicz, L. and Prates, R. (2020). Modelos e Requerimentos de Documentações, fornecido na disciplina de Estruturas de Dados. Disponibilizado via Moodle. Departamento de Ciência da Computação. Universidade Federal de Minas Gerais. Belo Horizonte.

Eswaran, Kapali P.; Tarjan, R.Endre(1976), "Augmentation problems", SIAM Journal on Computing, **5** (4): 653-665, doi:10.1137/0205044, MR 0449011