Trabalho Prático 2

Breno Cláudio de Sena Pimenta

[1]Departamento de Ciência da Computação
Universidade Federal de Minas Gerais
DCC216 - Algoritmos I

1. Introdução

O trabalho prático consiste em calcular o mínimo de novas rotas devem ser acrescentadas à atual malha de uma determinada empresa aérea para que, partindo de qualquer aeroporto, um passageiro possa alcançar qualquer outro aeroporto operado pela companhia. Esse percurso pode demandar diversas escalas para grandes malhas.

2. Modelagem Computacional

O problema foi modelado como um grafo $G(V, E)$, no qual os vértices representam os aeroportos e as arestas representam as rotas direcionadas e não ponderadas existentes na malha da companhia aérea. O número mínimo de rotas que devem ser acrescentadas para possibilitar que todos os aeroportos sejam alcançáveis a partir de qualquer outro é o mesmo número de arestas que devem ser acrescentadas para que G seja um único componente fortemente conectado. Assim, para solucionar o problema proposto foi utilizado da ideia do algoritmo de Kosaraju que permite identificar os componentes fortemente conectados (SCC) de um grafo por meio da busca em profundidade (DFS). O algoritmo está descrito a seguir.

Algorithm 1: Identificar os Componentes Fortemente Conectados

input : Grafo G e número de vértices V
output: Vetor scc

visitados $\leftarrow \emptyset$, stack $\leftarrow \emptyset$, scc $\leftarrow \emptyset$
foreach $V \in G$ **do**
    if *V não foi visitado* **then**
        Faça uma DFS em G a partir de V e quando a DFS terminar, coloque o vértice de origem em stack
G^{rev} $\leftarrow$ reverso do grafo G
visitados $\leftarrow \emptyset$
foreach $V \in$ *stack* **do**
    if *V não foi visitado* **then**
        Faça uma DFS em G^{rev} a partir de V, quando o DFS terminar todos os nós visitados formarão um scc e receberão o mesmo id
return scc

Com os SCCs do grafo G identificados, podemos analisar quantas arestas precisamos adicionar para que o grafo seja um único SCC. Para identificar essas arestas, foi

criado um novo grafo $G^{\text{SCC}}(V^{\text{SCC}}, E^{\text{SCC}})$ no qual cada SCC identificado anteriormente foi inserido como um vértice e as arestas foram adicionadas conforme o grafo original G. O algoritmo está descrito a seguir.

Algorithm 2: Criar grafo G^{SCC}

input : Vetor scc, Grafo G e número de vértices V
output: Grafo G^{SCC}

foreach $V \in G$ **do**
    foreach *vizinho V' de* $V \in G$ **do**
        if *scc[V]* $\neq$ *scc[V']* ***and*** *aresta (scc[V], scc[V'])* $\notin G^{SCC}$ **then**
            $G^{\text{SCC}} \leftarrow$ aresta (scc[V], scc[V'])
return G^{SCC}

Para que G^{SCC} seja fortemente conectado é preciso inserir n arestas faltantes para que haja um ciclo. Pra isso, é preciso contar o número de vértices que não possuem nenhuma aresta saindo deles (sinks) e o número de vértices que não possuem nenhuma aresta entrando (sources). O maior desses dois números representa o mínimo de arestas que devem ser adicionadas para que o grafo seja fortemente conectado, conforme demonstrado por Eswaran e Tarjan (1976).

Para encontrar o número de sinks basta verificar no grafo G^{SCC} quantos vértices não possui nenhum vizinho. Para encontrar o número de sources, podemos verificar no grafo reverso de G^{SCC} quantos vértices não possui nenhum vizinho. O algoritmo está descrito a seguir.

Algorithm 3: Encontrar o maior número entre sources e sinks

input : Grafo G^{SCC}, número de vértices V^{SCC}
output: n

sinks $\leftarrow 0$
foreach $V^{SCC} \in G^{SCC}$ **do**
    if V^{SCC} *não possui vizinhos* **then**
        sinks++
$G^{\text{SCCrev}} \leftarrow$ reverso(G^{SCC})
sources $\leftarrow 0$
foreach $V^{SCC} \in G^{SCCrev}$ **do**
    if V^{SCC} *não possui vizinhos* **then**
        sources++
n $\leftarrow$ maior(sinks, sources)
return n

3. Implementação da Solução

A solução do problema proposto foi desenvolvida em C++ utilizando o paradigma de orientação a objetos.

3.1. Estruturas de dados

A representação de um grafo em todo o prolema foi feita por meio de um vetor de vetores de inteiros da biblioteca STL (Standard Template Library) por meio da declaração *vector<vector<int>>*. O seu uso se assemelha às listas de adjacência, logo se a aresta $a \rightarrow b$ existe em G, ela é representada em alguma posição (a, n) do vetor de vetores g, ou seja, $g[a][n] = b$. Para implementar o algoritmo modelado anteriormente foram criadas duas classes: Airline e SCC.

3.2. Classe Airline

A classe Airline representa uma empresa aérea. Foi criada apenas com o propósito de organizar os dados de entrada. Ela possui funções para ler dados da entrada e de retornar o grafo G que representa a sua malha aérea.

3.3. Classe SCC

A classe SCC contém os métodos para representar e operar sobre componentes fortemente conectados. Ela identifica os SCC's de um dado grafo e calcula o mínimo número de arestas necessárias para tornar o grafo um único componente fortemente conectado. Como essa classe contém a parte principal da solução desenvolvida, seus atributos e métodos são detalhados a seguir.

3.3.1. Atributos

A classe SCC contém apenas os seguintes atributos privados:

- **_numOfVertexes:** Esse valor inteiro contém o número de vértices do grafo de entrada. No problema que estamos solucionando, representa o número de vértices V do grafo G, ou seja, o número de aeroportos operado pela companhia aérea.
- **_numSCC:** Esse valor inteiro representa o número de componentes fortemente conectados que o grafo de entrada G possui.
- **_graph:** Esse grafo representa o grafo de entrada G. Cada vértice representa um aeroporto e cada aresta é uma rota existente na malha aérea da companhia.
- **_vertexesSCC:** Vetor no qual cada vértice possui um identificador que representa a qual SCC ele faz parte. Logo, ele possui tamanho igual a _numOfVertexes. Desse modo, se _vertexesSCC[n] = m, significa que o vértice n está contido no componente fortemente conectado identificado por m.
- **_SCCgraph:** Esse grafo representa o grafo G^{SCC}, ou seja o grafo em que cada SCC do grafo original G é inserido como um único vértice. As arestas são mantidas conforme as rotas do grafo original.

3.3.2. Métodos

A classe SCC contém os seguintes métodos:

- **Construtor SCC**: O construtor da classe recebe o grafo já lido e construído pela classe Airline e o número de vértices. Além disso, ele define o número inicial de componentes fortemente conectados como zero.
- **fillOrderDFS**: Essa função executa uma DFS a partir de um vértice como primeira busca para encontrar os SCC. Ela salva em uma pilha o vértice de origem ao fim da execução.
- **DFS**: Essa função executa uma DFS a partir de um vértice como segunda busca para encontrar os SCC. Ela opera sobre o grafo G^{rev} e a cada fim de execução, os vértices que possuem o mesmo componente recebem o mesmo id.
- **reverseGraph**: Essa função recebe um grafo e o número de vértices e retorna o grafo reverso.
- **identifyVertexesSCC**: Essa função realiza a chamada de fillOrderDFS, DFS e reverseGraph para retornar o vetor no qual cada posição representa o id do SCC que cada vértice faz parte.
- **generateSCCgraph**: Essa função recebe o vetor scc e gera um grafo no qual cada vértice é um componente fortemente conectado do grafo G. As arestas são inseridas conforme as conexões do garfo original. A função salva paralelamente em uma matriz cada aresta existente para possibilitar essa checagem em O(1).
- **getNumSinks**: Essa função identifica quantos vértices de um dado grafo não possuem arestas de saída.
- **getNumSources**: Essa função identifica quantos vértices de um dado grafo não possuem arestas de entrada.
- **printNumMinimalEdges**: Essa função retorna o maior número de sources e sinks do grafo, ou seja, o menor número de arestas que devem ser adicionadas para que G seja fortemente conectado.

4. Análise de Complexidade Assintótica de Tempo

Existem três passos para solucionar o problema proposto, esses passos foram apresentados anteriormente pelos algoritmos 1: Identificar os Componentes Fortemente Conectados, 2: Criar grafo G^{SCC} e 3: Encontrar o maior número entre sources e sinks. A complexidade assintótica de cada um desses algoritmos será analisada a seguir.

1. **Identificar os Componentes Fortemente Conectados:** Esse algoritmo é uma implementação do algoritmo de Kosaraju, que depende do número de vértices V e de arestas E do grafo de entrada G, com uma modificação simples que não impacta na complexidade original. Assim, esse algoritmo tem complexidade $O(V + E)$.
2. **Criar grafo G^{SCC}:** Esse algoritmo precisa percorrer o grafo G e adicionar cada aresta que liga componentes diferentes em G^{SCC}. Logo, esse caminhamento na lista de adjacências depende do número de vértices V e arestas E do grafo original,

ou seja ocorre em $O(V + E)$. A checagem da existência de uma aresta ocorre em $O(1)$, visto que é usada uma matriz de adjacências apenas para essa checagem. Assim, esse algoritmo tem complexidade $O(V + E)$.

3. **Encontrar o maior número entre sources e sinks:** Esse algoritmo precisa checar quantos vértices de G^{SCC} e de $G^{\text{SCC}}{}_{\text{rev}}$ não possui vizinhos, essa checagem ocorre em $O(V)$. Porém, para gerar o reverso de G^{SCC}, é necessário realizar o caminhamento no grafo G^{SCC}. Esse caminhamento depende do número de vértices V^{SCC} e de arestas E^{SCC} do grafo G^{SCC}. Assim, esse caminhamento ocorre em $O(V^{\text{SCC}} + E^{\text{SCC}})$. Portanto, como V^{SCC} é menor que V e E^{SCC} é menor que E, esse algoritmo tem complexidade $O(V + E)$.

Assim, podemos concluir que a complexidade assintótica de tempo geral do código desenvolvido é $O(V + E)$.

5. Considerações finais

Ao fim do desenvolvimento, pode-se dizer que a maior dificuldade encontrada foi em como encontrar o mínimo de arestas necessárias para criar um único SCC a partir dos componentes fortemente conectados já identificados. Para solucionar esse problema foram feitas diversas pesquisas no livro e na internet, além de tentativas práticas para chegar ao algoritmo de solução.

A partir do trabalho foi possível entender mais sobre o grafos fortemente conectados, algoritmos de caminhamento e busca em grafos na prática. Com a solução desenvolvida foi possível encontrar o mínimo de rotas extras necessárias na malha aérea para que, a partir de qualquer aeroporto, um passageiro possa alcançar qualquer outro aeroporto operado pela companhia. Assim pode-se concluir que foi possível cumprir com os objetivos do trabalho prático proposto.

Referências

Jon Kleinberg, E. T. *Algorithm Design*. Pearson Education.

K. P. Eswaran, R. E. T. (1976). *Augmentation problem*. SIAM Joumal on Computing.

Raghavan, S. *A Note on Eswaran and Tarjan's Algorithm for the Strong Connectivity Augmentation Problem*. University of Maryland.

Ziviani, N. *Projetos de Algoritmos com Implementações em Java e C++*. Cengage Learning.