Trabalho Prático 02

João Victor Taufner Pereira - 2017098315

[1]Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte - MG - Brasil

1. Introdução

O problema proposto nesse trabalho consiste em, a partir de uma entrada contendo aeroportos e rotas entre eles, computar o número mínimo de rotas que devem ser inseridas à essa rede aérea para que seja possível se chegar a qualquer aeroporto a partir de qualquer aeroporto de partida.

2. Implementação e Modelagem

Para a resolução do trabalho, o problema foi modelado como um grafo direcionado não ponderado G = (V, E), em que V é o conjunto de vértices que correspondem aos aeroportos e E o conjunto de arestas que correspondem às rotas entre eles. Dessa forma, encontrar o número de mínimo de rotas que devem ser adicionadas significa encontrar o número mínimo de arestas que precisam ser adicionadas ao grafo para torná-lo fortemente conexo.

Para encontrar esse valor, o algorítmo desenvolvido as seguintes etapas:

- Encontra as componentes fortementes conexas do grafo de entrada.
- Cria um novo grafo em que cada vértice corresponde a uma das componentes fortemente conexas do grafo inicial.
- Para o novo grafo, são calculados o número de vértices poço(vértices que não possuem arestas de saídas) e o número de vértices fonte(vértices que não possuem arestas de entrada).
- O máximo entre o número de fontes e poços é retornado - equivale ao número de arestas que precisam ser adicionadas para tornar o grafo fortemente conexo.

Essas etapas, bem como suas implementações e estruturas de dados, serão mais profundamente discutidas nas próximas seções.

2.1. Entradas e Saídas do programa

As entradas e saídas do programa são as padrão do sistema (stdin e stdout).

- **Entradas do programa:** A primeira linha da entrada contém dois inteiros n e m, representando o número de aeroportos e o número de rotas, respectivamente. Os aeroportos serão portanto identificados por inteiros de 1 a n. As proxímas m linhas de entrada corresponderão às rotas entre os aeroportos. Cada uma delas possui dois inteiros, sendo o primeiro o identificador do aeroporto de partida e o segundo o identificador do aeroporto de destino.

- **Saída do programa:** A saída é composta por um único inteiro: o número mínimo de rotas que devem ser inseridas para que seja possível chegar a qualquer aeroporto de destino para qualquer aeroporto de partida.

2.2. Classes implementadas

2.2.1. Graph

Classe utilizada para representar um grafo. Possui um atributo para armazenar o número de vértices e outro para o número de arestas. Além disso, há um atributo para a lista de adjacência do grafo e outro para indicar quais vértices já foram visitados por algorítmos de busca. Todos esses atributos são privados. Seus métodos são:

- **Os construtores** da classe, que devem obrigatoriamente receber o número de vértices. Um deles recebe também o número de arestas(sobrecarga).
- **Getters e Setters** para que os atributos possam ser acessados e modificados fora da classe.
- **O método *addEdge(int node1, int node2***, que adiciona ao grafo uma aresta saindo do vértice do primeiro parâmetrico para o vértice do segundo.

2.2.2. GraphUtils

Trata-se de uma classe utilitária estática e não possui atributos. Seus métodos são:

- ***reverseGraph(Graph graph***, que recebe um grafo G = (V , E) e retorna um grafo T = (V , E'), em que E' é o conjunto com todas as arestas do conjunto E em sentido contrário. Ou seja, dados vértices x e y, se a aresta u vai de x a y em E, então ela vai de y a x em E'.
- ***DFS(Graph &graph, int node, stack<int> &dfsFinishOrder)***, que é uma implementação do algorítmo de busca em profundidade que recebe também uma pilha. Dessa forma, assim que um vértice é processado pelo algorítmo DFS (ou seja, ele e todos os seus vizinhos já foram processados), ele é empilhado em *dfsFinishOrder*. Quando todos os vértices tiverem sido processados, eles estarão dispostos na pilha segundo a ordem em que seus processamentos terminaram, sendo o vértice do topo da pilha o último a ser processado.
- ***DFS(Graph &graph, int node, int counter, vector<int> &newID)***, que é uma outra implementação do algorítmo de busca em largura, dessa vez incluindo um contador e um vetor em que cada posição corresponde a um dos vértices. Para cada chamada não recursiva dessa função, os vértices processados ganharão um identificador, que será armazenado no vetor *newID*.
- ***createSimplerGraph(Graph graph, vector¡int¿ newID, int newSize)***, que cria um novo grafo em que cada vértice é uma componente de entrada do grafo de entrada. Cada um dos novos vértices terá um identificador atribuido pelo vetor *newID*. Para casos em que existem mais de uma arestas saindo de um identificador x para um y, apenas uma delas é incluida.
- ***kosaraju(Graph graph)***, que é uma implementação utilizando DFS do algorítmo de Kosaraju para computar as componentes fortemente conexas de um grafo. Atribuirá um identificador para cada componente e retornará o resultado da chamada de *createSimplerGraph* para esses novos vértices, ou seja, um novo grafo em que cada vértice representa uma componente fortemente conexa do grafo inicial.
- ***edgesToMakeSCC(Graph graph)***, que é o método "principal" do programa, onde são feitas basicamente todas as chamadas supracitadas. Assim que é computado o novo grafo, esse método também calcula o número de poços e fontes e retorna o máximo entre eles. É importante mencionar que vértices isolados (sem arestas de entrada e saída) são considerados como poço e fonte simultaneamente.

2.3. Estruturas de Dados

Para a implementação do algorítmo, foram utilizados majoritariamente estruturas da classe *vector* e *stack* da biblioteca padrão. O uso do *stack* no método *kosaraju* se justifica pela necessidade de inserir os vértices a medida em que eles são processados, pois posteriormente será necessário processar esses mesmos vértices para o grafo reverso pela ordem decrescente do tempo de processamento. Para os demais atributos, a classe *vector* foi escolhida devido a facilidade de iterar por todos os elementos, o que é feito várias vezes ao longo do algorítmo. Por fim, destaca-se a escolha de representar grafos por listas de adjacência (nesse trabalho, implementadas como um *vector* bidimensional), o que permite a iterar pelos vizinhos de todos os vértices em tempo $O(|V| + |E|)$ para um grafo G = (V , E).

2.4. O método *kosaraju*

Escolhido para encontrar as componentes fortemente conexas do grafo de entrada, o algorítmo de Kosaraju segue os seguintes passos:

- Inicializa uma pilha vazia.
- Executa DFS processando todos os vértices do grafo. Esses vértices são empilhados a medida que forem sendo terminados seus respectivos processamentos.
- Executa DFS processando todos os vértices do reverso do grafo. Nessa etapa, o vértice x escolhido para se executar a DFS é o presente no topo da pilha. Todos os vértices ”alcançados” por x que ainda não haviam sido visitados pertencem a mesma componente fortemente conexa.

Além disso, o método *kosaraju* também mantém um vetor em que cada posição equivale a um vértice. Nesse vetor, são atribuidos identificadores para cada um dos vértices, de forma que vértices de uma mesma componente possuam o mesmo identificador. A partir disso, quando o método *createSimplerGraph* for chamado, esses vértices serão ”unificados”.

Algorithm 1: método ***kosaraju***

```
input  : G = (V , E)

Inicializa pilha processados ← ∅
identificador ← 0
foreach i ∈ vértices do
    DFS(G, i, pilha) empilhando os vértices segundo ordem de
      processamento
T ← Reverte(G)
foreach j ∈ topo de pilha não visitado do
    identificador ← identificador + 1
    DFS(T, j, identificador) em que identificadores[j] ← identificador
    desempilha j
Cria novo grafo a partir de identificadores e G
```

2.5. O método *edgesToMakeSCC*

Esse método tem em seu ínicio a chamada do método *kosaraju*, descrito em detalhes acima. Portanto, passaremos a considerar seu funcionamento a partir do momento em que já obtemos o grafo "simplificado" T cujo vértices são as componentes fortemente conexas do grafo inicial G.Esse é o momento em que a saída do programa começa a ser diretamente calculada.

Inicialemente calcula-se o reverso de T, que chamaremos de T'. Dessa forma, é possível calcular o número de poços e fontes iterando pela lista de adjacência desses dois grafos: Cada vértice de T sem vizinhos será um poço, e cada vértice de T' sem vizinhos será uma fonte.

A relação desses valores com o número de arestas que precisa ser inserido para tornar G fortemente conexo é bem simples. Como todos os vértices de T são componentes fortemente conexas de G, isso significa que basta-se encontrar o número de arestas para tornar T fortemente conexo.

Portanto, sendo m o número de poços e n o número de fontes em T(se um vértice for isolado, ele será poço e fonte ao mesmo tempo), o número mínimo de arestas que precisamos inserir para tornar T conexo é o máximo entre m e n. Isso pode ser visto analisando-se os dois possíveis casos:

- Se $m >= n$, podemos inserir n arestas saindo dos m poços e indo para as n fontes e em seguida $m - n$ arestas saindo dos $m - n$ poços restantes e indo para quaisquer vértices.
- Se $m < n$, podemos inserir m arestas saindo dos m poços e indo para as n fontes e em seguida $n - m$ arestas saindo de quaisquer vértices e chegando nas $n - m$ fontes.

Dessa forma, quando esse método retorna o máximo entre os dois valores, esse valor é a saída esperada do programa, solucionando o problema em questão.

3. Análise de Complexidade

O projeto, em sua totalidade, pode ser divido em cinco etapas: a leitura da entrada, a execução do método *kosaraju*, a criação do grafo simplificado pelo método *createSimplerGraph*, o cálculo da saída pelo método *edgesToMakeSCC* e a impressão da saída. Consideremos o grafo de entrada G = (V , E).

- **Leitura da entrada:** Na função main, a entrada é lida e é criada uma instância da classe *Graph* para esses valores. Inicialmente lê-se o número de vértices e arestas, e em seguida lemos $|E|$ os vértices de origem e destino de cada uma das arestas, adicionando-as ao grafo. Portanto, essa etapa é feita em tempo $O(|E|)$.
- **Execução do método *kosaraju*:** Inicialmente, o grafo de entrada é revertido, criando G'. Para isso, iteramos por toda a lista de adjacência em tempo $O(|V| + |E|)$. Em seguida, executamos o algorítmo DFS para os vértices de G em tempo $O(|V| + |E|)$ e para os vértices de G' segundo a ordem da pilha de processados também em $O(|V| + |E|)$. Como todas essas partes acontecem independendo do grafo de entrada, essa etapa inteira é dominada por $O(|V| + |E|)$.
- **Criação do grafo simplificado:** Para criar o grafo simplificado T, percorre-se a lista de adjacência do grafo de entrada G. Portanto, essa etapa é executada em tempo $O(|V|+|E|)$.
- **Cálculo da saída pelo método *edgesToMakeSCC*:** Inicia-se através da chamada do método *kosaraju*, que já vimos que é executado em $O(|V| + |E|)$. Além disso, há o custo para calcular o grafo simplificado T(foi tratado como uma etapa diferente, mas no código ocorre

no final do método *kosaraju*), que também já foi mostrado que é $O(|V| + |E|)$. Então, é criado o gráfico reverso T' em tempo $O(|V| + |E|)$. Por fim, verifica-se o tamanho do *vector* de vizinhos de cada vértice de T e T'. Sendo X o número de vértices de T(que é o número de componentes fortemente conexas de G), iteramos pelos vértices em $O(X)$ para ambos os grafos. Como todos os passos descritos acontecem da mesma forma e em mesma quantidade para qualquer grafo de entradam essa etapa é limitada superior por $O(|V| + |E|)$.

- **Impressão da saída:** A impressão da saída é feita por um único comando *cout*, por se tratar de apenas um inteiro. Portanto sua execução é $O(1)$.

Na função main, é feita a leitura $O(|E|)$ e a chamada do método *edgesToMakeSCC*. Dentro desse método, são chamados os métodos *kosaraju*(que por sua vez chama uma vez o método de criação do grafo simplificado) e o método que reverte o grafo. A partir da análise acima, é possível perceber que a execução completa do *edgesToMakeSCC* é $O(|V| + |E|)$ e, portanto, a complexidade geral é linear e da forma $O(|V| + |E|)$.

4. Conclusões

A partir da implementação do trabalho foi possível colocar em prática os conhecimentos adquiridos na disciplina a respeito de modelagem de grafos e componentes fortemente conexos. Dessa forma, por meio da impementação do algorítmo de Kosaraju com busca em profundidade, foi possível encontrar uma solução para o problema.

5. Bibliografia

- Kleinberg, Jon; Tardos, Eva. Algorithm Design, Pearson Education India, 2006