Documentação do segundo Trabalho Prático da disciplina de Algoritmos I

Luis Gabriel Caetano Diniz – Matrícula: 2019075711

Departamento de Ciência da Computação – Universidade Federal De Minas Gerais (UFMG)

Belo Horizonte – MG – Brazil

lgcaetano@ufmg.br

1. Modelagem Computacional do Problema

O problema proposto no trabalho consistia no desenvolvimento de um algoritmo que receberia as rotas de determinada companhia aérea e deveria imprimir na saída padrão quantas rotas adicionais eram necessárias para que qualquer um dos aeroportos atendidos pela companhia fosse alcançável a partir de qualquer outro.

Para resolver o problema, o modelamos usando grafos. Para cada entrada do programa, pode-se construir um grafo onde os vértices representam os aeroportos atendidos pela companhia enquanto as arestas representam as rotas oferecidas. Assim, podemos resolver o problema com o auxílio das diversas técnicas e algoritmos conhecidos sobre grafos.

2. Estruturas de Dados e explicação do algoritmo

2.1 Classes e estruturas utilizadas

Para podermos usar o conhecimento que já temos sobre grafos para solucionar o problema, precisamos de uma estrutura de dados para representá-los. Por esta razão, criamos a classe **Grafo**, que representará o conjunto de rotas e aeroportos disponibilizados. A classe possui um construtor, que inicializa uma matriz que será sua matriz de adjacência, um destrutor, uma função **contemAresta** que recebe dois números inteiros, que devem representar vértices, e retorna **true** caso exista uma aresta saindo do primeiro vértice em direção ao segundo, contém também uma função **contruirAresta** que, evidentemente, constrói uma aresta direcionada do vértice representado pelo primeiro parâmetro do método em direção ao segundo, um método denominado **dfs**, que executa o algoritmo DFS e retorna um vector de inteiros contendo os índices dos vértices que foram alcançados pelo algoritmo a partir do vértice inicial, além de receber um vector por referência (que utiliza para armazenar quais vértices já foram visitados pelo algoritmo) e uma pilha (stack) utilizada para colocar os vértices em ordem decrescente de acordo com o tempo de término no algoritmo DFS (algo essencial para a execução do algoritmo de Kosaraju). Além disso, a classe **Grafo** também possui o método **algoritmoDeKosaraju**, que executa o algoritmo de Kosaraju, retornando um vector de vectors de inteiros (algo próximo, mas não igual, a uma matriz de inteiros) contendo em cada linha os índices dos vértices que compõem um componente fortemente conectado. Há também o método **construirGrafoDeSCCs**, que constrói um grafo auxiliar onde cada vértice representa um componente fortemente conectado do grafo original. Além destes, temos também o método **arestasFaltando**, que retornará a resposta para nosso problema, calculando assim, quantas arestas precisam ser construídas para que o grafo se torne fortemente conectado.

2.2 Explicando o algoritmo

O algoritmo para resolução do problema consiste em uma série de passos: primeiro, deve se gerar o grafo auxiliar onde cada vértice representa um componente fortemente conectado do algoritmo original. Deve-se então, ligar cada vértice deste grafo auxiliar que possuía uma aresta o ligando a outro componente fortemente conectado no grafo original. Por exemplo, se A e B são componentes fortemente conectados do grafo e existe uma aresta ligando um vértice de A a um vértice de B, deve-se construir uma aresta no grafo auxiliar ligando o vértice que representa A ao vértice que representa B. Após isto, basta que se conte quantos vértices do grafo auxiliar não possuem nenhuma aresta de saída (uma aresta cuja origem seja este vértice) e quantos vértices não possuem nenhuma aresta de chegada (uma aresta cujo destino seja este vértice). O maior destes dois números calculados será a nossa resposta.

Este algoritmo pode parecer bem confuso e é pouco intuitivo quando não se demonstra a lógica dele. Porém, é possível deixá-lo mais claro se analisarmos a razão do porquê fazer cada passo: construímos o grafo auxiliar para que modelemos o problema de uma forma diferente, queremos tornar o grafo original um grafo conexo, mas ora, se conseguirmos conectar os componentes fortemente conectados do grafo então conectamos o grafo como um todo. Temos então **n** SCCs (componentes fortemente conectados) e **n** arestas no nosso grafo auxiliar. Como nosso grafo é direcionado, a solução mais trivial para o problema de conectar o grafo auxiliar, seria desenhar um ciclo de n arestas, passando por todos os vértices, ou seja, sabemos que conseguimos conectar o grafo com no máximo **n** arestas. Contudo, se houverem ligações no grafo original entre vértices de dois componentes fortemente conectados distintos, é possível que o grafo possa ser conectado com menos do que **n** arestas. Para então estabelecermos um valor mínimo, podemos observar o que é absolutamente necessário para que o grafo auxiliar inteiro possa ser fortemente conectado. Uma das condições que deve ser satisfeita é que todo vértice deve possuir uma aresta chegando a ele e uma aresta saindo dele, já que seria impossível alcançá-lo ou alcançar outro vértice a partir deste caso esta condição não fosse satisfeita. É por isso que realizamos o terceiro passo, caso T seja o número de vértices sem saída e U o número de vértices sem entrada, teremos que construir max(U, T) no mínimo para conectar o grafo. Contudo, como provar que uma solução que constrói apenas max(U, T) arestas existe? Para isso, existe um algoritmo desenvolvido por **Eswaran e Tarjan (1976)** (referência ao final da documentação) que sempre encontra um grupo de max(U, T) arestas que conecta o grafo. O algoritmo consiste nos seguintes passos: rode DFS para cada vértice que possui uma aresta de saída, mas nenhuma aresta de chegada e forme um par entre ele e um dos nós folha da sua árvore DFS (como este nó é folha, ele será possuirá aresta de chegada, mas nenhuma aresta de saída). É possível que não seja possível parear todos estes vértices em certos casos. Após esta etapa, conecte todos os vértices isolados (que não possuem arestas nem de saída nem de chegada) e todos os pares obtidos na última etapa formando um grande ciclo. Após isso, conecte os vértices que não possuem aresta de saída (mas possuem de chegada) a vértices que não possuem arestas de chegada (mas possuem de saída), até a exaustão. Então, ao final deste processo, se ainda restarem vértices sem arestas de saída ou chegada, conecte-os ao ciclo com as arestas adequadas. Este processo construirá apenas max(U, T) arestas.

2.3 Pseudocódigo do Algoritmo

```
ComponentesConectados <- Grafo.Kosaraju();

PARA1 i de 0 até ComponentesConectados.size:

    PARA2 j de 0 até ComponentesConectados.size:

        SE i != j e
ComponentesConectados[i].possuiArestaAté(ComponentesConecta
dos[j]):

            GrafoDeSCCs.desenharArestaEntre(i, j);

    FIM PARA2;

FIM PARA1;


numVerticesSemSaida <- 0;

numVerticesSemChegada <- 0;


PARA1 i de 0 até ComponentesConectados.size:

    SE GrafoDeSCCs.vertice(i).naoPossuiArestaDeSaida:

        numVerticesSemSaida++;

    SE GrafoDeSCCs.vertice(i).naoPossuiArestaDeChegada:

        numVerticesSemChegada++;

FIM PARA1;


return max(numVerticesSemChegada, numVerticesSemSaida);
```

3. Complexidade assintótica do algoritmo

Com relação à quantidade de memória utilizada pelo algoritmo, temos complexidade $O(n^2)$, já que usamos de uma matriz para representar as conexões entre os vértices de nosso grafo.

Quanto ao tempo, as coisas ficam mais complicadas, pois há vários passos a serem realizados no código. Temos de executar Kosaraju, que possui complexidade $O(n^2)$ no nosso caso (já que utilizamos uma matriz de adjacência e não uma lista). Além disso, para construir as arestas do grafo auxiliar, comparamos todos os vértices do grafo original com todos os demais vértices, (como se tivéssemos dois loops **for** de 0 a n aninhados, apesar de no próprio código serem 4 loops, mas que se comportam como se fossem 2) processo que possui complexidade $O(n^2)$. Finalmente, percorremos a matriz do grafo auxiliar (que possui m vértices, com m sendo o número de SCCs do grafo original) para verificar quantos vértices não possuem saída ou entrada, processo que possui complexidade $O(m^2)$. Como $m < n$, a complexidade final do nosso algoritmo é $O(n^2)$, já que esta é a maior complexidade encontrada durante a execução.

Referência: https://epubs.siam.org/doi/10.1137/0205044