ACH2024 – Algoritmos e Estruturas de Dados II

EXERCÍCIO PROGRAMA 1 – VERSÃO 3

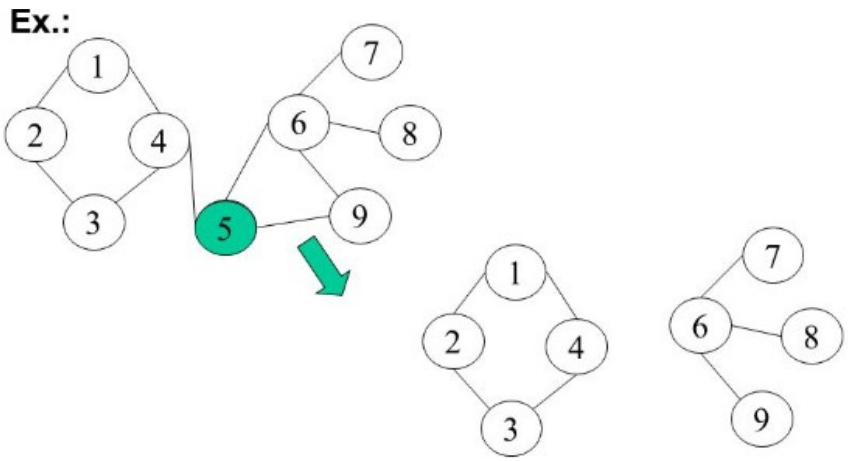
Data de entrega: 09 de maio de 2017

Evidência de plágio entre trabalhos, mesmo que parcial, implicará na nota zero no trabalho.

Responsável: Pavel Rojas
Profa: Ariane Machado Lima

Neste EP1, vocês devem resolver o problema abaixo usando busca em profundidade (DFS) e busca em largura (BFS) que aprendeu na aula. O objetivo deste exercício-programa é implementar os algoritmos básicos e usá-los em um problema de visualização.

Um vértice de articulação é um vértice que, se for removido do grafo, aumenta o número de componentes conectados do grafo em pelo menos um. Uma possível abordagem para identificar se um dado vértice é um ponto de articulação é identificar se o número de componentes conectados do grafo (não direcionado) original é menor que o número de componentes conectados do grafo sem aquele dado vértice e as arestas que saem/entram nele. De maneira similar é para grafos direcionados, considerando apenas que neste caso fala-se em "componentes fortemente conectados".



Essa abordagem simples, descrita no parágrafo anterior, será aceita no EP. Contudo essa estratégia é ineficiente O(V*(V+E)). Há um algoritmo O(V+E) que é capaz de identificar os pontos de articulação apenas analisando a árvore da busca em profundidade executada sobre o grafo original. DICA: Na árvore de busca em profundidade:

- um vértice folha …. (pode ou não pode ser um ponto de articulação?);
- o vértice raiz é um ponto de articulação se e somente …;
- um vértice *x* que não é folha nem raiz é um ponto de articulação se e somente se *x* possui um vértice filho *y* tal que … (alguma coisa a ver com a sub-árvore com raiz em *y*...)

O algoritmo eficiente para grafos direcionados é proposto no artigo publicado no periódico *Theoretical Computer Science*, vol 447, ano 2012, páginas 74–84.

Você tem que implementar um algoritmo que recebe a entrada e retorna a saída no formato especificado.
Este EP é uma simplificação do problema sugerido em SPOJ JUDGE: http://www.spoj.com/problems/GRAPHS12/

Descrição:

Neste problema, seu programa terá que ler um grafo ponderado **não direcionado** de um arquivo cujo nome (nome completo, incluindo path se houver) é informado na linha de comando[1] e executar alguns algoritmos básicos no gráfico:

- Visualização do grafo: impressão do grafo no mesmo formato da entrada

- Busca em largura (BFS Breadth First Search):
  - após a linha "BFS:" devem ser impressos os vértices na ordem em que eles foram descobertos (separados por um espaço)
  - após a linha "BFS Paths": deve ser impresso, em cada linha i, o caminho do vértice raiz em questão[2] até o vértice i durante a BFS (vértices separados por um espaço)

- Busca em profundidade (DFS Depth First Search):
  - após a linha "DFS:" devem ser impressos os vértices na ordem em que eles foram descobertos (separados por um espaço)
  - após a linha "DFS Paths": deve ser impresso, em cada linha i, o caminho do vértice raiz em questão[3] até o vértice i durante a DFS (vértices separados por um espaço)

- Visualização de componentes conectados: cada componente conectado *i* deve ser impresso em uma linha que se inicia com "C*i*: ". A impressão de um componente significa imprimir seus vértices (separados por um espaço) em ordem crescente.

- Visualização dos vértices de articulação (os números dos vértices que são pontos de articulação): uma única linha contendo todos os vértices que são vértices de articulação (separados por um espaço)

---

1 Por exemplo, se seu código for compilado para gerar um executável com o nome "ep1.exe", o seu programa será executado via linha de comando da seguinte forma:
```
ep1.exe entrada_teste.txt
```

2 Se o grafo não for conexo, haverá mais de uma árvore de busca em largura. Assim, cada vértice pertence a uma única árvore com uma raiz específica.

3 Se o grafo não for conexo, haverá mais de uma árvore de busca em profundidade. Assim, cada vértice pertence a uma única árvore com uma raiz específica.

Especificação da entrada:

Na primeira linha você terá um par de números: *n* e *e*, sendo o primeiro (*n*) a quantidade de vértices no grafo, e o segundo (*e*) é a quantidade de arestas no grafo.

As próximas "*e*" linhas conterão uma tripla em cada linha representando uma aresta: *o*, *t* e *w*, na qual *o* é o vértice origem de uma aresta, *t* é o vértice alvo (target) e *w* é o peso. As arestas serão sempre descritas de forma que o vértice origem será o de menor número que o vértice alvo (o mesmo vale para a impressão do grafo na saída).

As triplas (arestas) são apresentadas ordenadas pelo vértice origem de forma crescente. Além disso, se houver mais de uma aresta saindo de um mesmo vértice origem, elas serão ordenadas pelo vértice destino.

Os números dos vértice são consecutivos: 1, 2, …, n.

Saída

Você terá que mostrar todas as informações na lista mostrada na seção acima (na descrição). Dê uma olhada no Exemplo abaixo para ver o formato a seguir. Toda a saída deve ser impressa na saída padrão.

Exemplo

Entrada:

~~0~~ ←- não teremos mais essa primeira linha

7 8

1 2 1

1 3 1

2 4 1

3 4 1

4 5 1

4 6 1

5 7 1

6 7 1

Saída:

~~0~~←- não teremos mais essa primeira linha

7 8

1 2 1

1 3 1

2 4 1

3 4 1

4 5 1
4 6 1
5 7 1
6 7 1

BFS:
1 2 3 4 5 6 7

BFS Paths:
1
1 2
1 3
1 2 4
1 2 4 5
1 2 4 6
1 2 4 5 7

DFS:
1 2 4 3 5 7 6

DFS Paths:
1
1 2
1 2 4 3
1 2 4
1 2 4 5
1 2 4 5 7 6
1 2 4 5 7

Connected Components:
C1: 1 2 3 4 5 6 7

Articulation Vertices:
4

Informações adicionais:

- Você deve fazer um relatório explicando seu algoritmo **didaticamente** (pdf de só uma página).
- O código deve ser implementado na linguagem C .
- Assuma que o número máximo de vértices é 100

- Vocês poderão optar por usar a implementação de grafos por matriz ou listas de adjacência.[4] Haverá um gabarito (saída esperada) para os dois casos: matriz e lista de adjacência. O exemplo de saída acima se refere apenas à implementação por matriz de adjacência. No TIDIA serão disponibilizados exemplos de saída para as duas implementações.
- O trabalho deverá ser postado no tidia – 3 arquivos:
  - NomeCompletoTudoJuntoCadaNomeIniciadoComLetraMaiúscula.c (contendo o código .c)
  - NomeCompletoTudoJuntoCadaNomeIniciadoComLetraMaiúscula.h (contendo o código .h)
  - NomeCompletoTudoJuntoCadaNomeIniciadoComLetraMaiúscula.pdf (contendo o relatório).

Evidência de plágio entre trabalhos, mesmo que parcial, implicará na nota zero no trabalho.

4 Particularmente eu acho que é mais interessante para vocês implementarem usando listas de adjacências, pois além de ser a que mais tem a ver com o EP (já que usa buscas em largura e profundidade), é o tipo de implementação mais indicada na maioria dos casos, e portanto com mais chances de vocês precisarem utilizar em breve na vida profissional e/ou acadêmica de vocês. Lembrem-se que se usarem as interfaces sugeridas em aula, muito do EP será independende da implementação por matrizes ou listas de adjacência.