Resolução do problema EP01

Geraldo Rodrigues de Melo Neto
Gustavo Duarte Ventino
Maria Luisa Gabriel Domingues
Pedro de Araújo Ribeiro
Lucas Marques Pinho Tiago

O Problema:

Uma rede de telefone é representada pelas conexões entre casas, uma casa está sujeita a problemas de conexão ou manutenção portanto é necessário verificar se ao cortar a conexão para uma casa as demais continuarão conectadas a rede. Uma casa está conectada se é possível chegar até ela a partir de qualquer outra casa, diretamente ou indiretamente. Devemos contar a quantidade de “pontos críticos” ((articulações)) na rede.

Entradas:

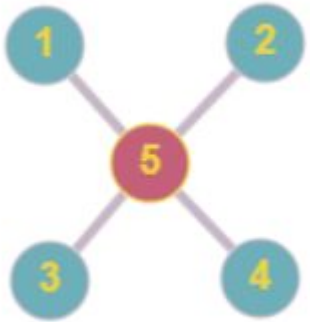
A entrada consiste em, nessa ordem:

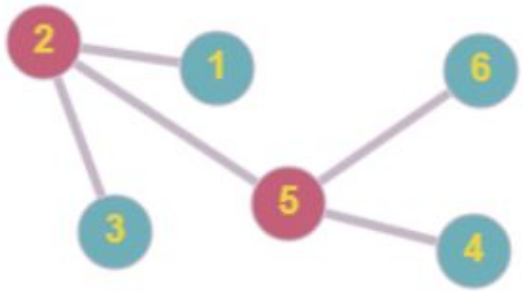
- Quantidade n de casas;
- 1 a n linhas contendo o número de uma casa seguido de todas as casas que conectam a ela;
- Um 0 para terminar de registrar essa rede, seguido de outro 0 para encerrar o programa ou a quantidade n de casas da próxima rede.

Saídas:

O programa deve imprimir a quantidade de pontos críticos ((articulações)) de cada rede.

Exemplo:





Nossa Resolução:

Decidimos resolver esse problema com Grafos pois podemos representar a existência de uma conexão entre duas casas como as adjacências de um grafo não direcionado.

Definimos dois TADs nos quais estão, respectivamente, classes presentes: uma para o grafo e outra para os vértices do grafo.

Nossa Resolução:

- Graph.hpp : contém, de atributos, um array com os vértices do grafo, além do número de vértices e arestas.
  - De métodos, contém Getters e Setters básicos para cada atributo, um método para a geração do grafo e três métodos referentes à verificação da existência de articulações no grafo, sendo um deles um DFS modificado.
- Vertex.hpp: contém, de atributos, um array de vértices adjacentes, o valor do vértice (dado no input), seu id e se o vértice é seguro.
  - De métodos, contém Getters e Setters básicos para cada atributo, além de métodos como addToAdjacency(), que inserem vértices no array de adjacência, gerando novas arestas.

```cpp
#include <vector>

class Vertex
{
private:
  int id;
  bool marked;
  std::vector<Vertex*> adjacency;
public:
  //Construtores:
  Vertex();
  Vertex(int id);
  //Getters:
  int getId();
  bool isMarked();
  std::vector<Vertex*> getAdjacency();
  //retorna o primeiro adjacente não colorido do vertice
  Vertex* getAdjacencyNotColored();
  //Setters:
  void setId(int id);
  void mark();
  void unmark();
  //Adiciona o vertice v na lista de adjacencia
  void addToAdjacency(Vertex *v);
  //Printa as informacoes de id e valor do vertice atual
  void print();
  //Printa a lista de adjacencia do vertice atual
  void printAdjacency();};
```

```cpp
#include "Vertex.hpp"
#include <iostream>
#include <fstream>

class Graph
{
private:
  std::vector<Vertex*> vertices;
  int size;
public:
  //Construtores:
  Graph();
  Graph(int size);
  //Getters e Setters
  int getSize();
  void setSize(int size);
  //Faz toda a leitura de entrada e cria o grafo:
  static Graph* readGraph(int j);
  //Adiciona novo vertice na lista de vertices:
  void addVertex(Vertex* v);
  //Dado um id retorna o vertice naquela posição:
  Vertex* getVertex(int id);
  //Imprime todo o grafo:
  void print();
  void printVertices();
  void dfs(int u, int *dfs_numbercounter, std::vector<int>& dfs_num,std::vector<int>&
dfs_low,std::vector<int>& dfs_parent,std::vector<int>& articulacao,int *dfsRoot, int *rootChildren);
  int contaCritico();
};
```

Método para resolução do problema:

Optamos por dividir o problema em duas etapas: a leitura do grafo, que implica em ler a quantidade de vértices, criá-los no grafo e em seguida ler linha por linha e realizar as atribuições necessárias; e a aplicação do algoritmo de contagem de articulações visto em aula.

Leitura dos Dados:

```cpp
#include "Graph.hpp"
#include <iostream>

int main() {
  int n;
  std::string input;
  std::vector<int> cnt;;
  std::cin >> n;
  while(n!=0){
  std::getline(std::cin, input);
  Graph *g = Graph::readGraph(n);
  n = g->contaCritico();
  cnt.push_back(n);
  std::cin >> n;
  }
  for(auto i : cnt)
    std::cout << i << std::endl;
  return 0;
}
```

```cpp
Graph *Graph::readGraph(int j) {
  std::string input;
  std::vector<int> numbers;
  int entry1, entry2,m,n;
  Graph *g;
  Vertex *v1, *v2;
  g = new Graph();

  g->setSize(j);
  for(int i=0; i<j;i++){
    v1 = new Vertex(i);
    g->addVertex(v1);
  }

  while(1){
    std::getline(std::cin, input);
    std::istringstream iss(input);
    int num;
    while (iss >> num) {
        numbers.push_back(num);
    }
    if(numbers[0]==0)
      break;
    v1 = g->getVertex(numbers[0]-1);
    for(int i=1;i<numbers.size();i++){
      v2 = g->getVertex(numbers[i]-1);
      v1->addToAdjacency(v2);
      v2->addToAdjacency(v1);
    }
    input.clear();
    numbers.clear();
  }
```

Método para resolução do problema:

```
70  void Graph::dfs(int u, int *dfs_numbercounter, std::vector<int> &dfs_num,
71                  std::vector<int> &dfs_low, std::vector<int> &dfs_parent,
72                  std::vector<int> &articulacao, int *dfsRoot,
73                  int *rootChildren) {
74    *dfs_numbercounter = *dfs_numbercounter + 1;
75    dfs_num.at(u) = *dfs_numbercounter;
76    dfs_low.at(u) = dfs_num.at(u);
77    for (auto v : getVertex(u)->getAdjacency()) {
78      //std::cout << "flag " << u <<std::endl;
79      if (dfs_num.at(v->getId()) == -1 /*unvisited*/) {
80        dfs_parent.at(v->getId()) = u;
81
82        if (u == *dfsRoot)
83          *rootChildren = *rootChildren + 1;
84
85        dfs(v->getId(), dfs_numbercounter, dfs_num, dfs_low, dfs_parent,
86            articulacao, dfsRoot, rootChildren);
87
88        if (dfs_low.at(v->getId()) >= dfs_num.at(u)){
89          articulacao.at(u) = 1;
90        }
91
92        dfs_low.at(u) = std::min(dfs_low.at(u), dfs_low.at(v->getId()));
93      } else if (v->getId() != dfs_parent.at(u)) {
94        dfs_low.at(u) = std::min(dfs_low.at(u), dfs_low.at(v->getId()));
95      }
96    }
97  }
```

```
99  int Graph::contaCritico(){
100   int n = getSize(),u = 0, dfs_numbercounter = -1, dfsRoot = 0,
    rootChildren = 0,accumulator = 0;
101   std::vector<int> dfs_num, dfs_low, dfs_parent, articulacao;
102
103   // preenchendo os vetores com seus valores iniciais:
104   for (int i = 0; i < n; i++) {
105     dfs_num.push_back(-1);
106     dfs_low.push_back(0);
107     dfs_parent.push_back(-1);
108     articulacao.push_back(0);
109   }
110
111   for (int u = 0; u < n; u++) {
112     if (dfs_num.at(u) == -1) {
113       dfsRoot = u;
114       rootChildren = 0;
115       dfs(u, &dfs_numbercounter, dfs_num, dfs_low,dfs_parent,
    articulacao,&dfsRoot, &rootChildren);
116       articulacao.at(u) = (rootChildren > 1);
117     }
118   }
119   //total de vertices articulacao:
120   for (auto i : articulacao) {
121     accumulator += i;
122   }
123   return accumulator;
124 }
```

FIM

Referências:

Repositório com os códigos fonte: https://replit.com/@PedroRibeiroA12/S03EP01

Imagens de grafos tiradas de: https://graphonline.ru/en/#