Resolução do problema EP02

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

O Problema:

Dada uma quantidade **n** de estações ferroviárias bidirecionais que trocam informações à quem está conectada e **m** bombardeios possíveis, devemos verificar quais são as **m** melhores estações para serem bombardeadas de forma com que o “pigeon value” seja o maior possível;

“Pigeon value” é o número mínimo de pombos usados para transmitir uma informação à todas as estações não bombardeadas.

Entradas:

A entrada consiste em diversas verificações seguindo o modelo:

- Quantidade **n** estações ferroviárias ($3 < n < 10000$);
- Quantidade **m** de bombardeios ($1 \leq m \leq n$);
- Próximas linhas com um **par de inteiros (x,y)**, sendo eles a conexão entre duas estações;
- A parada da verificação se dá quando temos **x = y = -1**;

Para encerrar a entrada devemos ter **m = n = 0** na leitura de estações e bombardeios.

Saídas:

O programa deve imprimir as **m** melhores estações para o bombardeio, imprimindo um par de inteiros (x,y) na qual x é o número da estação e y é o seu “Pigeon value”;

A saída estará ordenada de forma decrescente referente ao “pigeon value” e caso tenham valores iguais, é ordenado em ordem crescente referente ao número da estação.

Exemplo:

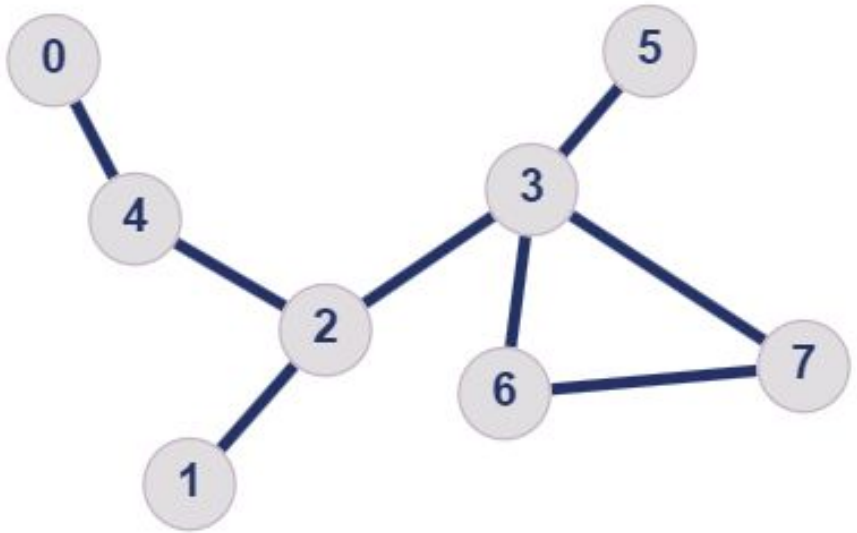
Output:

2 3

3 3

4 2

0 1



Exemplo:

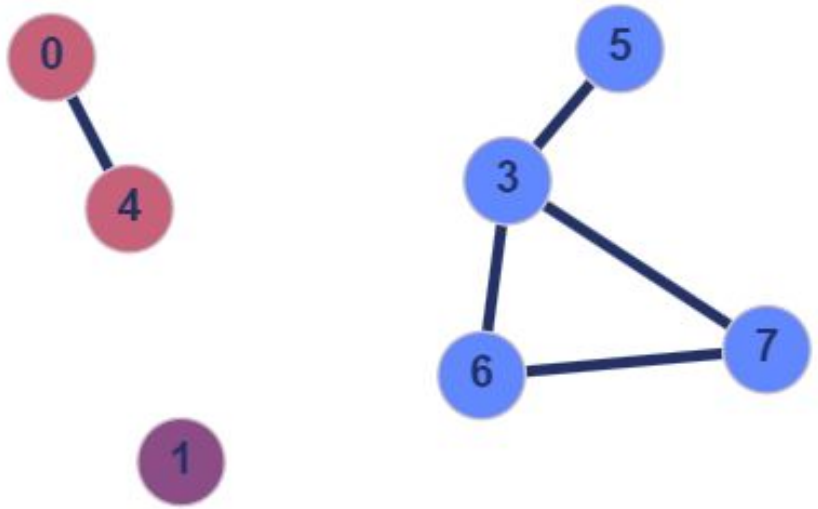
Output:

2 3

3 3

4 2

0 1



Exemplo:

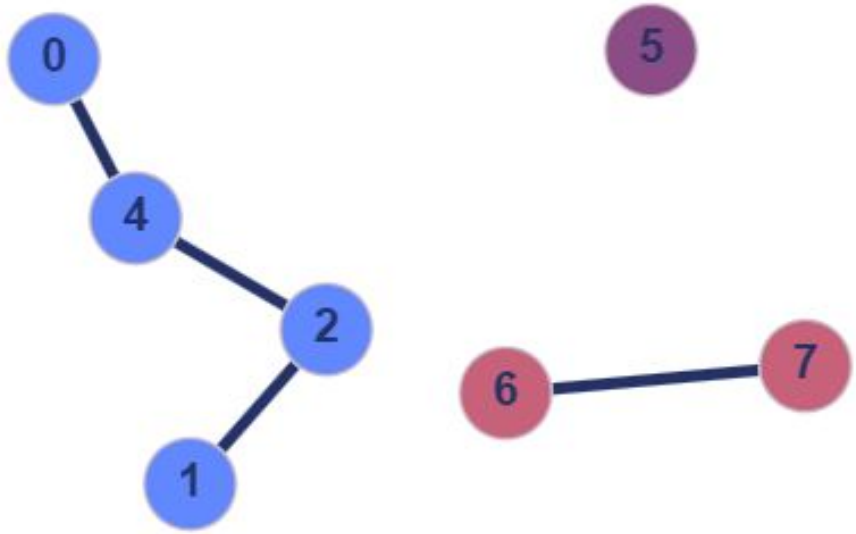
Output:

2 3

3 3

4 2

0 1



Exemplo:

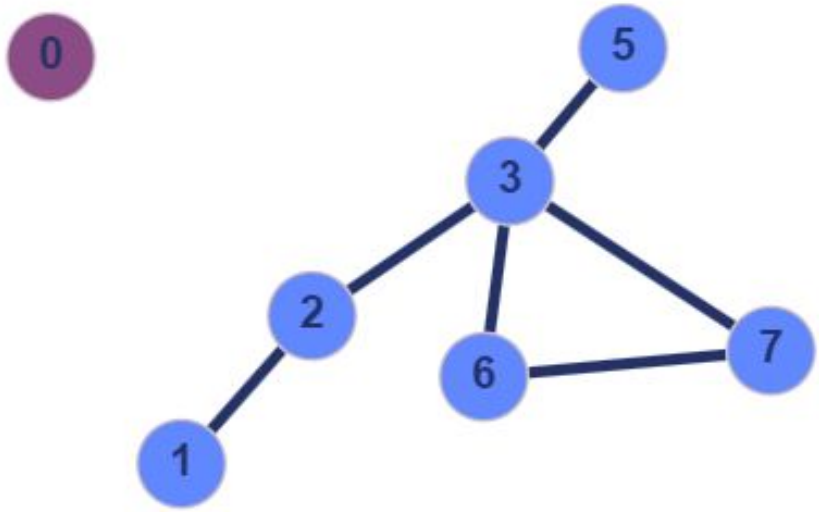
Output:

2 3

3 3

4 2

0 1



Exemplo:

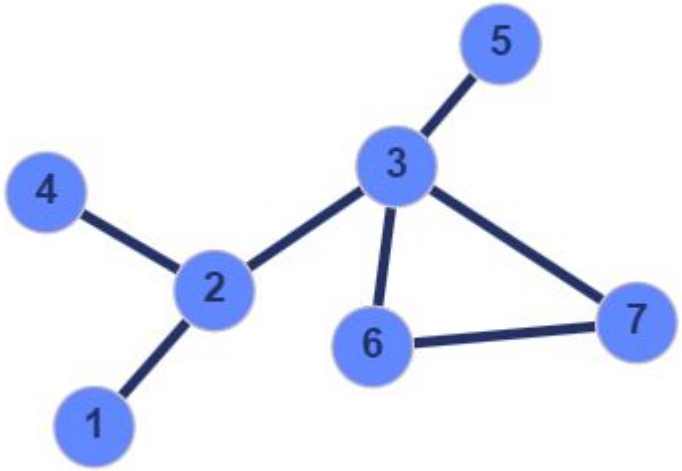
Output:

2 3

3 3

4 2

0 1



OBSERVAÇÃO: A partir desse ponto, qualquer outro vértice também traria o mesmo resultado, porém por conta da ordenação, foi imprimido o vértice 0

Nossa Resolução:

Decidimos resolver esse problema com Grafos pois podemos representar a existência de uma estação A para B como as adjacências de um grafo bidirecionado.

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

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 um método para a verificação da quantidade de componentes do grafo.
- Vertex.hpp: contém, de atributos, um array de vértices adjacentes, o valor do vértice (dado no input), seu id e a marcação de vertices.
  - 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.
- Sort.hpp: Não contém atributos
  - Possui quatro métodos estáticos para comparar um par de inteiros e os ordenarem da forma requisitada.

```cpp
#include <vector>

class Vertex
{
private:
  int id;
  std::vector<Vertex*> adjacency;
  bool mark;
public:
  //Construtores:
  Vertex();
  Vertex(int id);

  //Getters:
  int getId();
  bool getMark();
  std::vector<Vertex*> getAdjacency();

  //retorna o primeiro adjacente não colorido do vertice
  Vertex* getAdjacencyNotColored();

  //Setters:
  void setId(int id);
  void setMark(bool mark);
  void setSafe(bool safe);

  //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>
#include <stack>

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 n,int m);

  //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();

  //Calcula quantidade de componentes ao "retirar" um vertice
  void dfs_qtd(int u, int v);
};
```

```cpp
#include <iostream>
#include <vector>

class Sort
{
  public:
    int static comparePair(std::pair<int,int> a,std::pair<int,int> b);

    void static swapPair(std::vector<std::pair<int,int>> &v, int pivo, int i);

    int static partitionPair(std::vector<std::pair<int,int>> &v, int esq, int dir);

    void static quicksort(std::vector<std::pair<int,int>> &v, int esq, int dir);
};
```

Método para resolução do problema

Para resolver o problema, primeiro fizemos a leitura do grafo a partir da função readGraph, depois foi feita uma verificação da quantidade de componentes que o grafo teria quando “removemos” cada vértice.

Essa verificação foi feita com o uso de uma busca por profundidade em um loop condicional.

Leitura de dados:

```
24  Graph *Graph::readGraph(int n,int m) {
25    int entry1, entry2, k;
26    Graph *g;
27    Vertex *v1, *v2;
28    g = new Graph( );
29
30    // cria vertices com id baseando-se em n
31    for (int i = 0; i < n; i++) {
32      v1 = new Vertex(i);
33      g->addVertex(v1);
34    }
35    // conecta todos os vertices que deverão ser conectados
36    while(true) {
37      std::cin >> entry1;
38      std::cin >> entry2;
39      if(entry1 == -1 || entry2 == -1) break;
40      v1 = g->getVertex(entry1);
41      v2 = g->getVertex(entry2);
42      v1->addToAdjacency(v2);
43      v2->addToAdjacency(v1);
44    }
45    // retorna o grafo com todos os vertices
46    return g;
47  }
```

Verificações de quantidade de componentes:

```
41    //Verifica nº de componentes se "retirar o vertice"
42    for (int i = 0; i < n; i++) {
43      for(int j = 0; j < n; j++){
44        if(j != i && (!g->getVertex(j)->getMark())){
45          components.at(i).second++;
46          g->dfs_qtd(j,i);
47        }
48       }
49      for(int j = 0; j < n; j++){
50        g->getVertex(j)->setMark(false);
51      }
52    }
```

```
56  void Graph::dfs_qtd(int u, int v){
57    vertices.at(u)->setMark(true);
58    for(auto i : vertices.at(u)->getAdjacency())
59      {
60        //procura marcar cada vértice adjacente,
    identificando-o pelo ID
61        if((!i->getMark()) && i->getId() != v) dfs_qtd(i-
    >getId(),v);
62      }
63  }
```

FIM

Referências:

Repositório com os códigos fonte: https://replit.com/@Gezero/TG3-EP02

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