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:

No país de Graphland existem várias cidades mas nenhuma estrada, o governo de Graphland planeja construir estradas entre as cidades do mesmo estado e ferrovias para conectar cidades de diferentes estados. Mas para economizar nos materiais eles precisam conectar todas as cidades usando a menor extensão de estradas e ferrovias.

Entrada:

O input é divido em 3 partes:

- A primeira linha possui um valor **T** que representa o número de casos teste.
- A primeira linha de cada caso possui dois inteiros
  - **N** = número de cidades.
  - **R** = distância limite para que uma estrada se torne uma ferrovia.

As **N** linhas seguintes de cada caso teste devem possuir dois valores X e Y, em que (X,Y) são as coordenadas de cada cidade no mapa**.**

Como as adjacências não nos são dadas conectamos todas as cidade diretamente.

Os pesos das arestas são decididos fazendo menor distância entre dois pontos.

Saídas:

O programa deve fornecer um output para cada um dos casos teste do input, ou seja, o programa deve ter um ou mais outputs.

O output é composto de 3 valores:

- Número de estados:
  a. Para cada ferrovia criada, considera-se um novo estado.
- Extensão mínima de estradas.
- Extensão mínima de ferrovias.

Exemplo:

Entrada:

```
3
3 100
0 0
1 0
2 0
3 1
0 0
100 0
200 0
4 20
0 0
40 30
30 30
10 10
```

Saída:

```
Case #1: 1 2 0
Case #2: 3 0 200
Case #3: 2 24 2
```

Nossa Resolução:

- Graph.hpp : Contém, de atributos, um array com os vértices, um array com as arestas e o número de vértices.
  - readGraph ( ) => Leitura e criação do grafo.
  - kruskal ( ) => Implementa algoritmo de kruskal para MST's.
- Vertex.hpp : Contém, de atributos, x, y e id;
- Edges.hpp : Vértice a, Vértice b e peso da aresta.
- UFDS.hpp : Contém, de atributo um array de inteiros que auxilia na lógica na hora da construção da MST por meio do Kruskal.
  - setRoot (int i, int j) => Configura o conjunto de j como conjunto de i.
  - findRoot (int i) => Encontra o conjunto a que i pertence.
  - isConnected(int i, int j) => Verifica se conjunto[i] == conjunto[j].

```hpp
#include "Edges.hpp"
#include <vector>
#include <utility>
#include <iostream>
class Graph
{
  public:
  //ATRIBUTOS:
  std::vector<Edges*> arestas;
  std::vector<Vertex*>vertices;
  int numVertices;

  //METODOS:
  //Construtores:
  Graph();
  Graph(int size);

  //Faz toda a leitura de entrada e cria o grafo:
  static Graph* readGraph(int n);
  Vertex* getByValue(int x, int y);
  //Adiciona novo vertice na lista de vertices:
  void addEdge(Edges* e);
  bool pertence(int x,int y);
  //Imprime todo o grafo:
  void print();
  //Kruskal:
  Graph* kruskal();
};
```

```hpp
#include "Vertex.hpp"
class Edges
{
  public:
  Vertex *a;
  Vertex *b;
  double peso;
  //Construtores:
  Edges();
  Edges(Vertex* a, Vertex* b, double peso);
  //Printa as informacoes de id e valor do vertice atual
  void print();
};
```

```hpp
#include <vector>
#include <string>
class Vertex
{
  public:
  int x;
int y;
  int id;
  //Construtores:
  Vertex();
  Vertex(int x, int y, int id);
  //Printa as informacoes de id e valor do vertice atual
  void print();
};
```

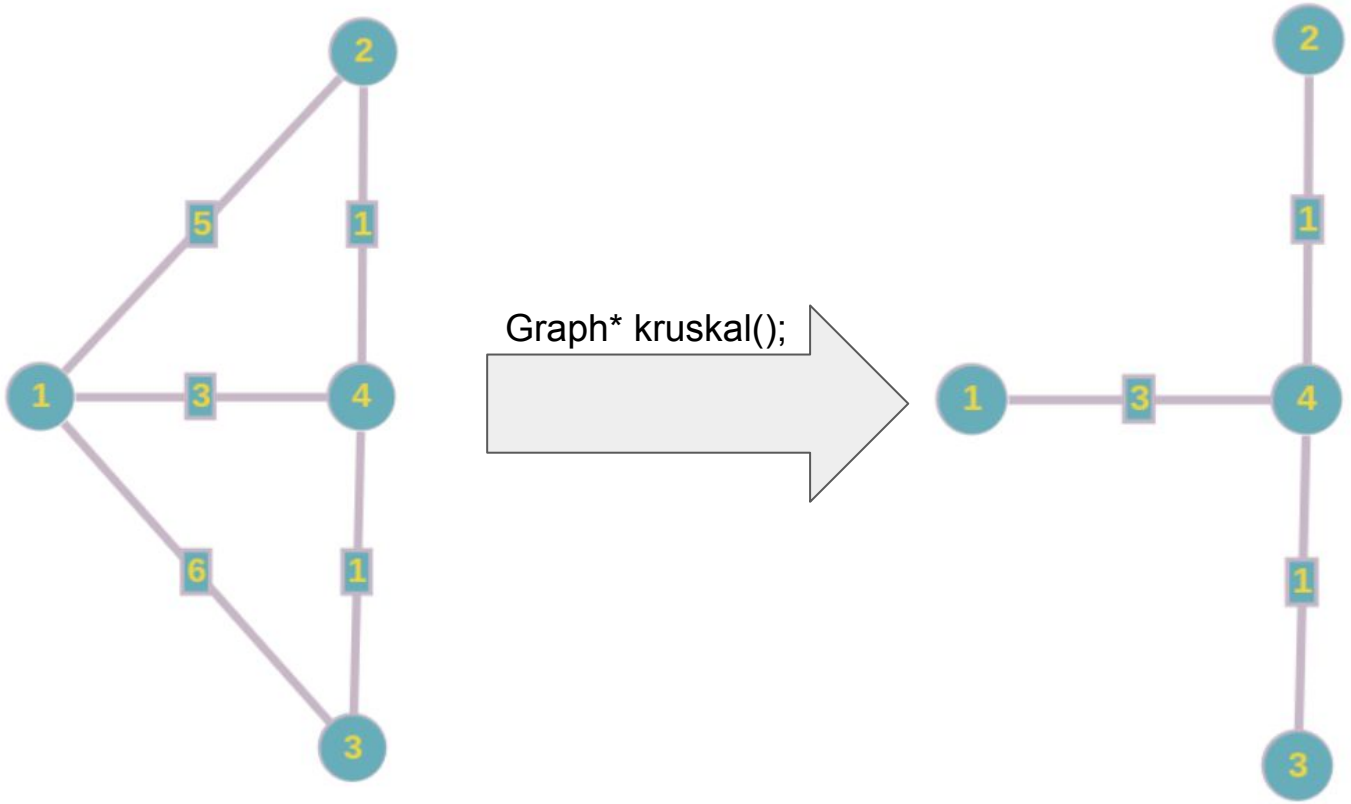
```hpp
#include <iostream>
#include <vector>

class UFDS{
public:
  //ATRIBUTOS:
  std::vector<int> marked;
  //METODOS:
  UFDS();
  UFDS(int size);
  bool isConnected(int i, int j);
  void setRoot(int i, int j);
  int findRoot(int i);
};
```

Método para resolução do problema:

- Leitura dos vértices do grafo e construção deste:
  - Usamos para isso a função Graph* readGraph( ) da nossa classe de Grafo descrita em Graph.hpp e Graph.cpp, essa função é responsável por realizar o cálculo das distâncias e conectar diretamente todos os vértices do grafo.
- Encontrando menor caminho:
  - A nossa função Graph* kruskal( ) monta a árvore geradora mínima do nosso grafo e a retorna, isso implica em dizer que, todos os vértices estão conectados, sem ciclos e com menor peso total de arestas possível.
  - Na main( ) realizamos a contagem de estados, comprimento de estradas e comprimento de ferrovias da árvore geradora mínima obtida.

Exemplo de criação e alteração do grafo:



```cpp
// Faz toda a leitura de entrada e cria o grafo:
Graph *Graph::readGraph(int n) {
  Graph *g;
  Edges *edge;
  Vertex *v1 = NULL;
  Vertex *v2 = NULL;
  int entry1, entry2;
  int id = 0;
  int i;

  g = new Graph(n);
  for (i = 0; i < n; i++) {
    std::cin >> entry1;
    std::cin >> entry2;
    v1 = new Vertex(entry1, entry2, id);
    id++;

    if (g->pertence(entry1, entry2)) {
      v1 = g->getByValue(entry1, entry2);
      id--;
    } else
      g->vertices.push_back(v1);

    // conecta
    for (int j = i - 1; j >= 0; j--) {
      v2 = g->vertices[j];
      // calculando menor distancia entre dois pontos
      // pontos (v1x,v1y) e (v2x,v2y)
      double a = v2->x - v1->x;
      double b = v2->y - v1->y;
      a = a * a;
      b = b * b;
      edge = new Edges(v1, v2, sqrt(a + b));
      g->addEdge(edge);
    }
  }
  return g;
}
```

```cpp
Graph *Graph::kruskal() {
  // ordenar as arestas do grafo por peso;
  std::stable_sort(arestas.begin(), arestas.end(),
                   [](Edges *e1, Edges *e2) //funcao lambda
                   { return e1->peso < e2->peso; });

  Graph *g = new Graph();
  // Criar grafo vazio;

  UFDS *ufds = new UFDS(numVertices);
  for (Edges *it : arestas) {
    //pega o id de cada vertice da aresta
    int u = it->a->id;
    int v = it->b->id;
    //pega a raiz dos vertices
    int uRep = ufds->findRoot(u);
    int vRep = ufds->findRoot(v);
    //atualiza a raiz e adicionar a aresta no grafo
    if (uRep != vRep) {
      ufds->setRoot(u, v);
      g->addEdge(it);
    }
  }
  return g;
}
```

```cpp
int main() {
  Graph *g, *k;
  std::vector<Graph *> grafos;
  std::vector<Graph *> kruskal;
  std::vector<int> rr;
  // usados para o input:
  int casos, n, r;
  // usados para o output:
  double roads = 0, rails = 0;
  double states = 1;
  std::queue<std::tuple<int, int, int>> saida;

  // captura:
  std::cin >> casos;
  for (int i = 0; i < casos; i++) {
    std::cin >> n;
    std::cin >> r;
    rr.push_back(r);
    g = Graph::readGraph(n);
    grafos.push_back(g);

    // montando a saida:
    int i = 0;
    for (Graph *g : kruskal) {
      std::tuple<double, double, double> tripla;
      for (int j = 0; j < g->arestas.size(); j++) {
        Edges *e = g->arestas[j];
        if (e->peso <= rr[i])
          roads += e->peso;
        else {
          states++;
          rails += e->peso;
        }
      }
      // montando saida:
      tripla = {states, (int)round(roads), (int)round(rails)};
      saida.push(tripla);
      // resetando valores:
      states = 1;
      roads = 0;
      rails = 0;
      i++;
    }
    // printando saida:
    for (int i = 1; !saida.empty(); i++) {
      std::cout << "Case #" << i << ": ";
      std::cout << std::get<0>(saida.front()) << " ";
      std::cout << std::get<1>(saida.front()) << " ";
      std::cout << std::get<2>(saida.front()) << std::endl;
      saida.pop();
    }
```

FIM

Referências:

Repositório com os códigos fonte: https://replit.com/@Ventinos/S05EP01?v=1

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