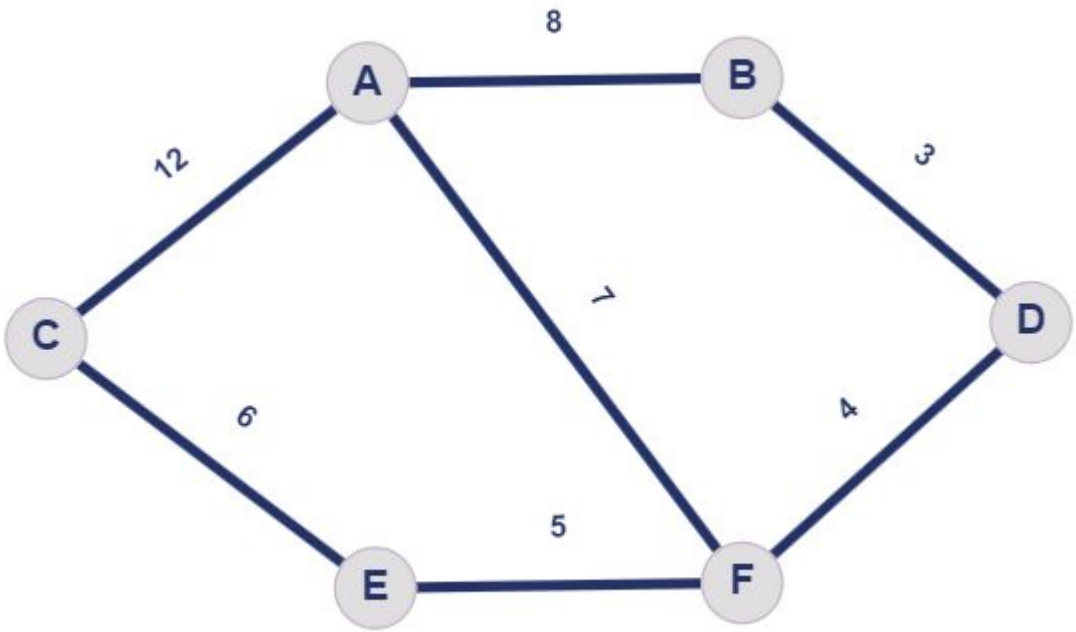
# Resolução do problema EP06

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

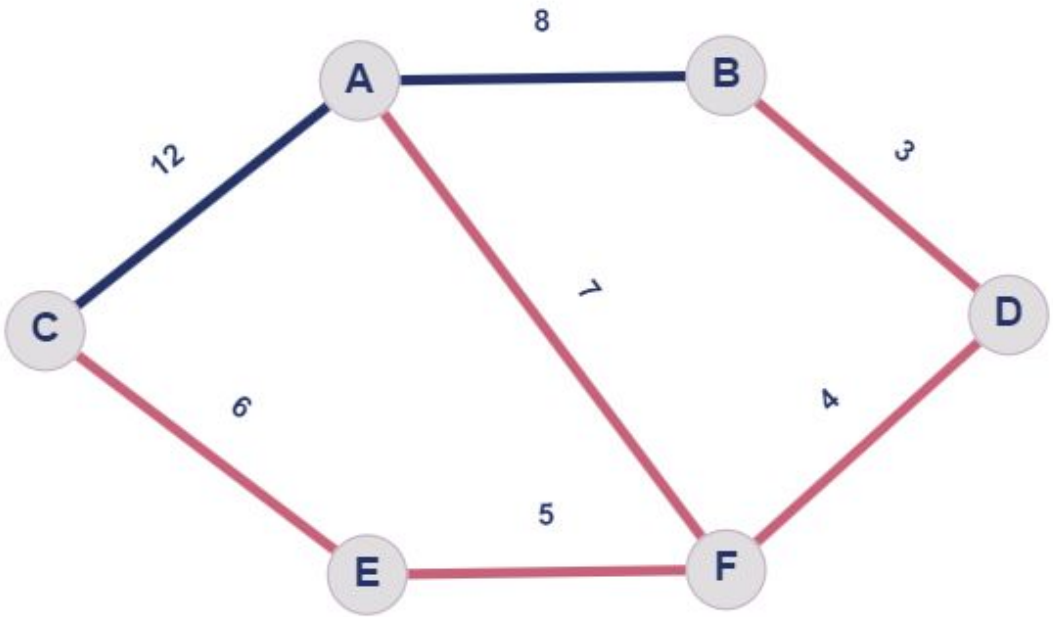
# O Problema:

Temos várias cidades protegidas por muros que se conectam por túneis subterrâneos que são protegidos por uma quantidade x de guardas. Assim para proteger as cidades, com recursos limitados, devemos determinar quais túneis serão protegidos com o menor número de guardas sem que uma cidade fique isolada.

# Exemplo:



# Exemplo:



# Entradas:

A entrada consiste em diversas verificações seguindo o modelo:

- Número de verificações **n**;
- Quantidade **m** de cidades;
- Próximas **m** linhas que formam uma “matriz” de túneis;
  - A linha **x** coluna **y** representam a o túnel entre a cidade x com a cidade y;
  - O **número presente** em **(x,y)** representa a quantidade de guardas para o túnel;
  - Caso esse número seja 0, significa que não existe conexão entre as cidades;

**Observação:** As cidades são nomeadas por letras do alfabeto.

# Saídas:

A saída do programa mostra os túneis de conexão e a quantidade de guardas necessárias para protegê-lo.

**Exemplo:**

```
Case 1:
B-D 3
D-F 4
E-F 5
C-E 6
A-F 7
```

# Nossa Resolução:

Decidimos resolver esse problema com Grafos pois podemos representar cada cidade como um grafo e os seus túneis como arestas, sendo o peso de cada aresta a quantidade de guardas no túnel.

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.
  - De métodos temos um método para a geração do grafo, um método de solução e impressão da solução.
- Vertex.hpp: contém, de atributos, um par de id de vértices (gerado após input) e o tamanho da aresta.
  - De métodos temos apenas print, que imprime o vértice dado.

```cpp
#include "Vertex.hpp"
#include <fstream>
#include <queue>
class Graph
{
public:
  std::vector<Vertex*> vertices;
  int size;

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

//Faz toda a leitura de entrada e cria o grafo:
  static Graph* readGraph(int n);

//soluçaõ do problema, retorna o novo grafo
  Graph * solve();

//Imprime todo o grafo:
  void print();

//Imprime a resposta
  void printResul();
};
```

```cpp
#include <vector>
#include <string>
class Vertex
{
public:
  //Atributos
  std::pair<int,int> aresta;
  int size;

  //Construtores:
  Vertex();
  Vertex(int id1, int id2);
  Vertex(int id1, int id2, int size);

  //Printa as informacoes de id e valor do vertice atual
  void print();
};
```

# Método para resolução do problema

Para resolver o problema, primeiro fizemos a leitura do grafo a partir da função readGraph, depois foi realizada o algoritmo de árvore geradora de Kruskal.

# Leitura de dados:

```
11  // Faz toda a leitura de entrada e cria o grafo:
12  Graph * Graph::readGraph(int n) {
13    int size = 0;
14    std::string input;
15    Vertex *v;
16    Graph *g;
17    g = new Graph(n);
18    for (int i = 0; i < n; i++) {
19      for(int j = 0; j < n; j++){
20        std::cin >> input;
21        size = stoi(input);
22        if(size != 0){
23          if(i > j){
24          v = new Vertex(j,i,size);
25          g->vertices.push_back(v);
26          }
27        }
28      }
29    }
30    //Ordena os vertices
31    std::sort(g->vertices.begin(), g->vertices.end(),
32              [](Vertex *e1, Vertex *e2)
33              { return e1->aresta.second < e2->aresta.second; });
34    std::stable_sort(g->vertices.begin(), g->vertices.end(),
35              [](Vertex *e1, Vertex *e2)
36              { return e1->aresta.first < e2->aresta.first; });
37    std::stable_sort(g->vertices.begin(), g->vertices.end(),
38              [](Vertex *e1, Vertex *e2)
39              { return e1->size < e2->size; });
40    return g;
41  }
```

# Resolução:

```
50  void Graph::printResul() {
51    for (auto i : vertices) {
52      char R1, R2;
53        R1 = i->aresta.first + 65;
54        R2 = i->aresta.second + 65;
55      std::cout << R1 << "-" << R2 << " " << i->size <<std::endl;
56    }
57  }
```

```
60  Graph * Graph::solve() {
61    int vet[size];
62    Graph *g;
63    g = new Graph(size);
64    //inicionalizamos o vetor de "pais" com -1
65    //-1 indica que o vértice não foi inserido no grafo
66    for(int i = 0 ;i<size;i++)
67      vet[i]=-1;
68    for(auto i : vertices){
69      if((vet[i->aresta.first]==-1)&&(vet[i->aresta.second]==-1)){
70        vet[i->aresta.first]=i->aresta.first;
71        vet[i->aresta.second]=i->aresta.first;
72        g->vertices.push_back(i);
73      }
74      else if(vet[i->aresta.first]==vet[i->aresta.second]){}
75      else if(vet[i->aresta.first]==-1){
76        vet[i->aresta.first]=vet[i->aresta.second];
77        g->vertices.push_back(i);
78      }
79      else if(vet[i->aresta.second]==-1){
80        vet[i->aresta.second]=vet[i->aresta.first];
81        g->vertices.push_back(i);
82      }
83      else{
84        int a,b;
85        if(vet[i->aresta.second]<vet[i->aresta.first]){
86          a=vet[i->aresta.first];
87          b=vet[i->aresta.second];
88        }
89        else{
90          a=vet[i->aresta.second];
91          b=vet[i->aresta.first];
92        }
93        for(int j = 0;j<size;j++)
94          if(vet[j]==a)
95            vet[j]=b;
96        g->vertices.push_back(i);
97      }
98    }
99    return g;
100 }
```

FIM

# Referências:

Repositório com os códigos fonte: https://replit.com/@Gezero/S05EP06

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