DCC
DEPARTAMENTO DE
CIÊNCIA DA COMPUTAÇÃO

UFMG
UNIVERSIDADE FEDERAL
DE MINAS GERAIS

Estruturas de Dados

Aplicações de Conjuntos disjuntos: Árvores geradoras mínimas

Professores: Wagner Meira Jr
Eder Figueiredo

Mais alguns conceitos sobre grafos

Um grafo é **conexo** quando para todo par de vértices *u* e *v*, existe um caminho que conecta *u* a *v*.

Um **ciclo** é uma sequência de vértices $v_1 v_2 \ldots v_k v_1$ de forma que termos sucessivos são adjacentes. A diferença de um ciclo para um caminho é que o ciclo começa e termina no mesmo vértice.

Uma **árvore** é um grafo que não possui ciclos.

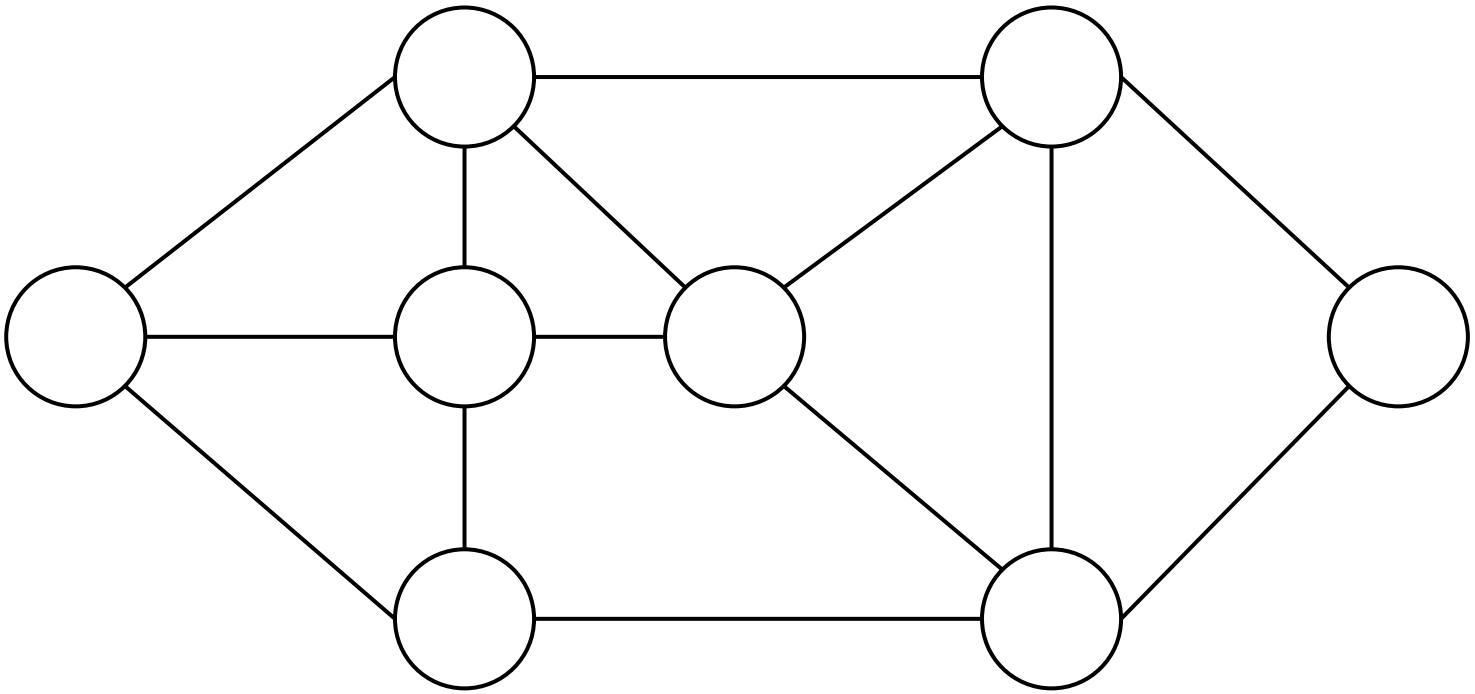
Um probleminha!

Suponha um distrito que possua apenas estradas de terra. A prefeitura local deseja pavimentar estradas de forma que as pessoas consigam se deslocar entre quaisquer dois pontos de interesse passando apenas por estradas pavimentadas (o caminho não precisa ser o menor). O custo de se pavimentar uma estrada é proporcional a seu comprimento.

O objetivo de hoje é desenvolver um algoritmo capaz de calcular o orçamento mínimo para se pavimentar essas estradas.

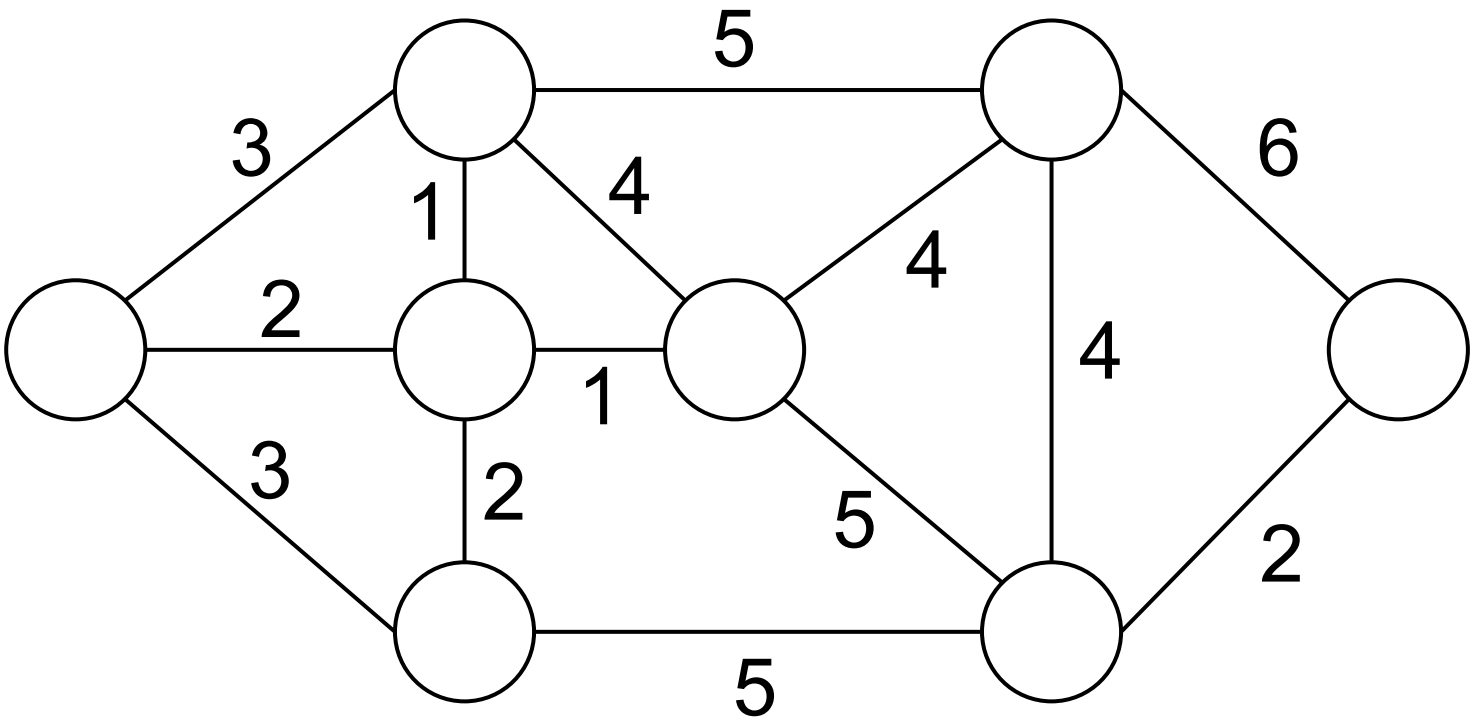
Um probleminha!

Nós podemos modelar o problema como um grafo conexo, onde os pontos de interesse são os vértices e as estradas são as arestas.



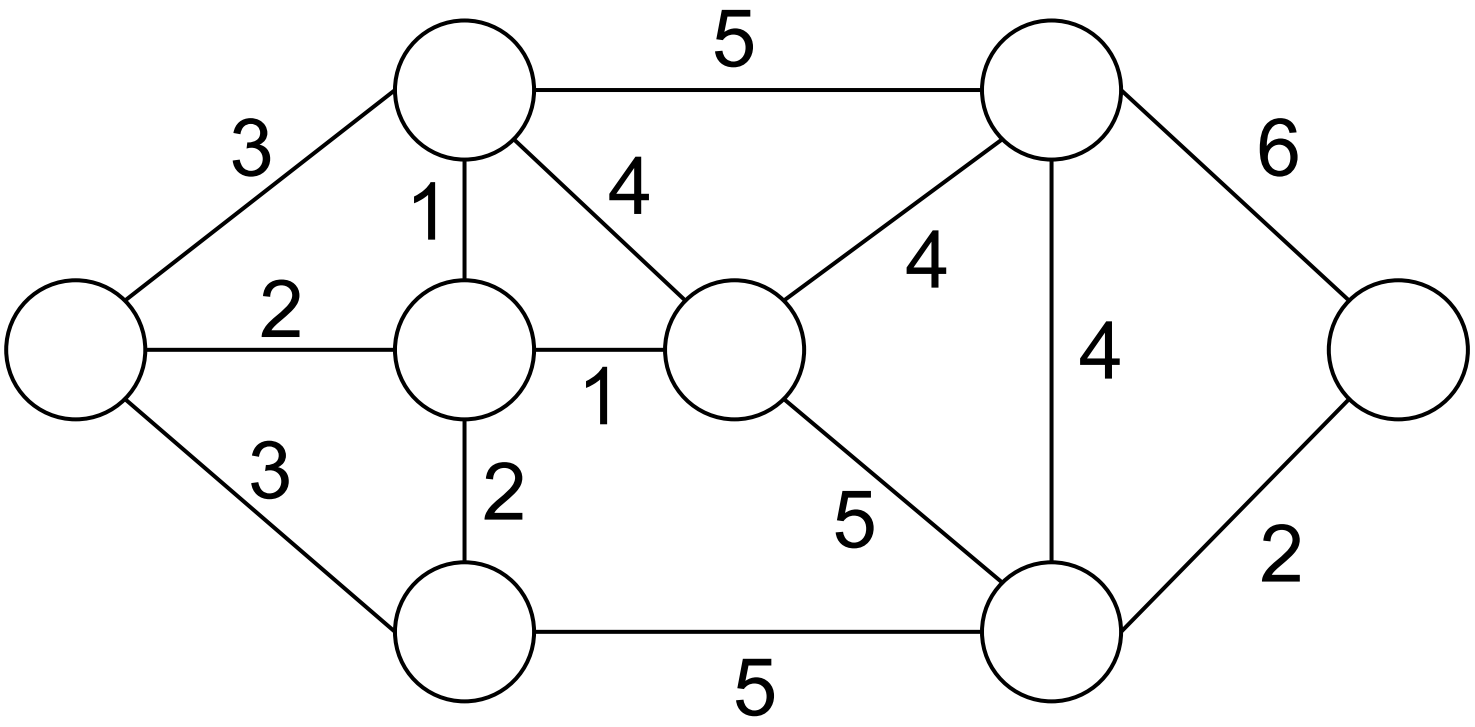
Um probleminha!

Mas agora nossas arestas são ponderadas. Ou seja, cada uma estará associada ao custo de se pavimentar a estrada que ela representa.



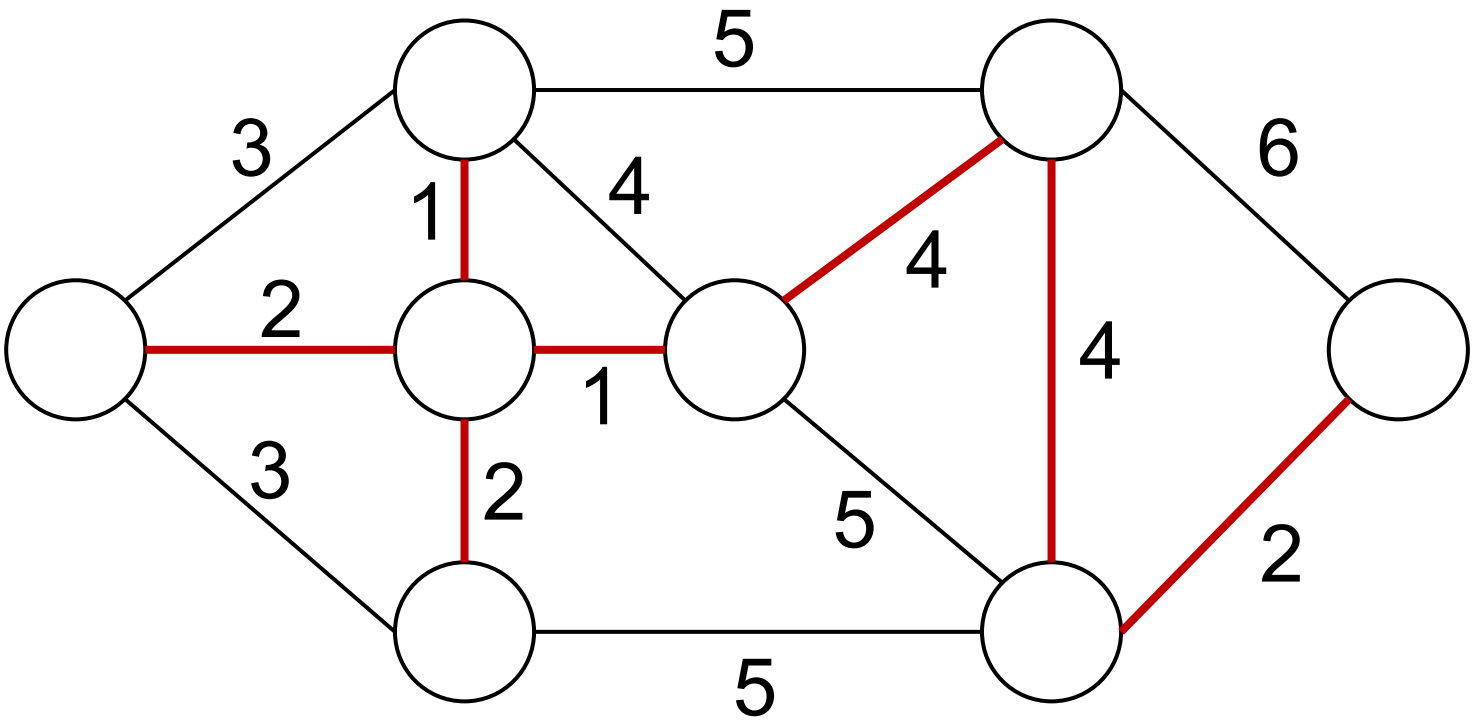
Um probleminha!

Note que arestas que fecham ciclos só aumentam o custo, então estamos interessados em encontrar uma árvore.



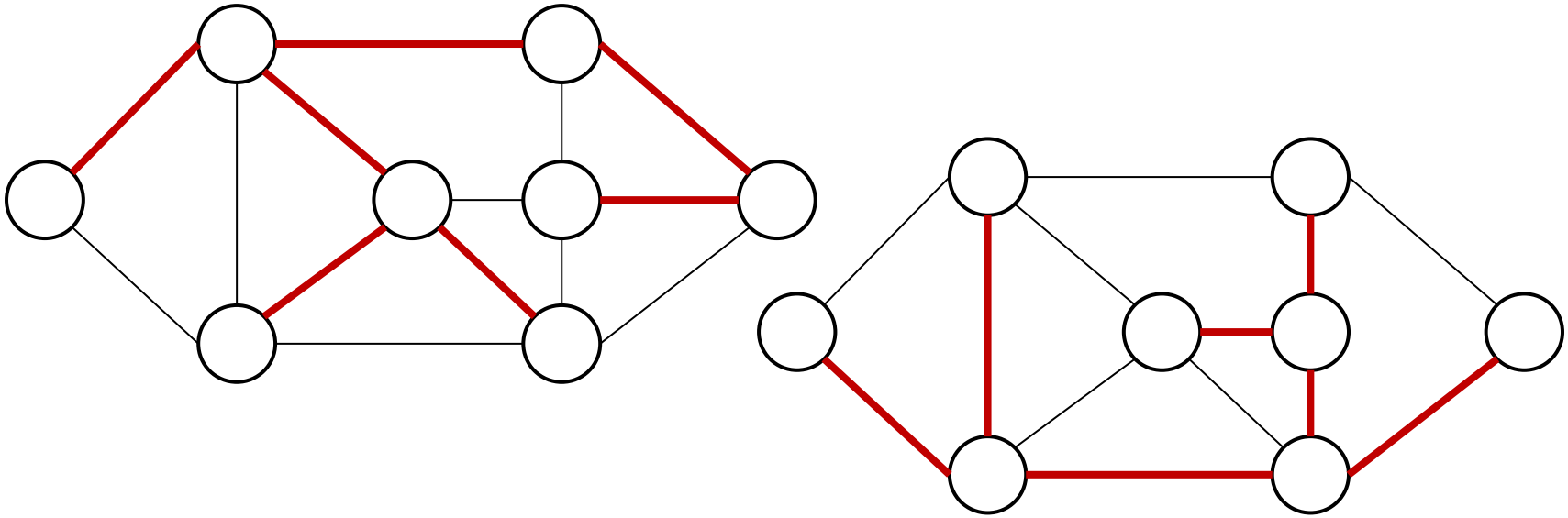
Um probleminha!

Neste exemplo a solução são as arestas ressaltadas de vermelho, com custo total de 16.



Árvores geradoras

Queremos encontrar então arestas em G que formem uma árvore que conecta todos os vértices. Chamamos esse tipo de árvore de **árvore geradora**. Observe que árvores geradoras não são únicas.



Como encontrar uma árvore geradora?

Mas não estamos interessados em uma árvore geradora qualquer. Queremos uma que **minimize** a soma dos custos das arestas selecionadas. Este é um problema clássico da computação conhecido como **árvore geradora mínima** (AGM).

No momento estamos apenas interessados em praticar as estruturas de dados, então não vamos falar sobre as provas de corretude do algoritmo.

A pergunta central deste problema é: qual critério usaremos para selecionar as arestas?

Como encontrar uma árvore geradora?

Para resolver este problema iremos utilizar em conjunto duas estruturas que já estudamos:

- *Heap*
- *Union-find*

Utilizaremos um **min heap** para ditar em qual ordem analisaremos as arestas do grafo, e depois iremos usar o *union-find* para aos poucos criar uma única componente conexa sem ciclos.

Como encontrar uma árvore geradora?

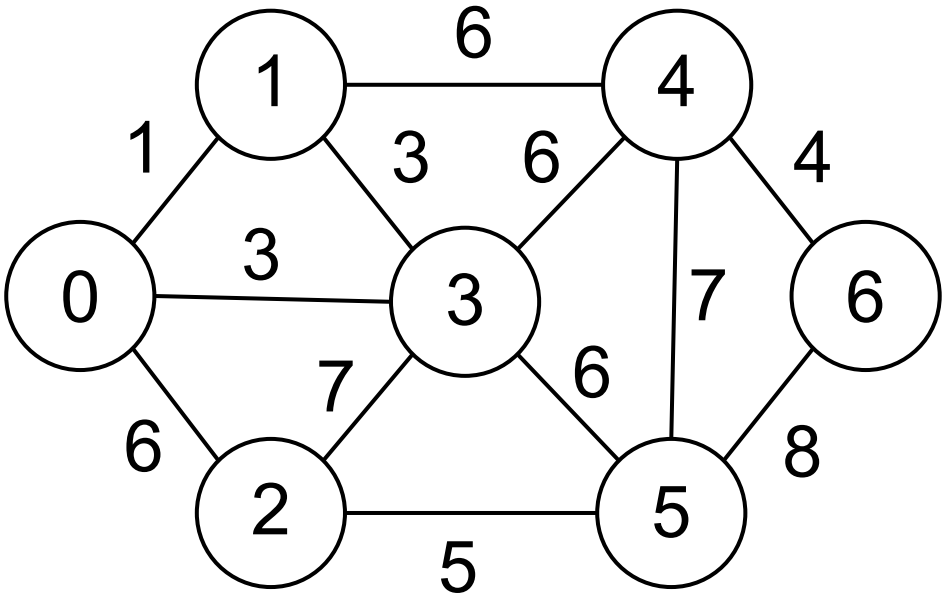
Para auxiliar nosso algoritmo, iremos nos apoiar no seguinte fato:

Toda árvore de *n* vértices possui *n-1* arestas.

O algoritmo será:

- Inicialize um contador de uniões e um de custo com 0, coloque todas as arestas no min heap e crie um subconjunto para cada vértice, sendo ele seu próprio representante
- Enquanto o heap não estiver vazio, remova a aresta do heap. Se o representante de cada ponta for diferente, faça união e incremente o contador.
- Caso o contador de uniões valha *n-1*, pare.

Árvores geradoras - exemplo



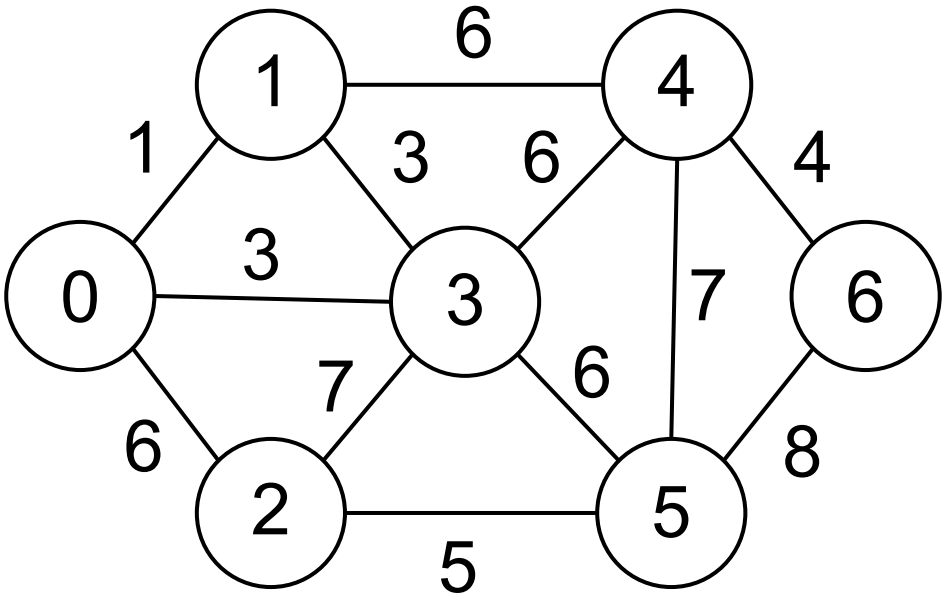
custo = 0
cont = 0

Representantes:

0	1	2	3	4	5	6
0	1	2	3	4	5	6

A variável *cont* armazena quantas uniões foram feitas até então. O vetor de representantes é inicializado de forma que cada vértice seja seu próprio representante.

Árvores geradoras - exemplo



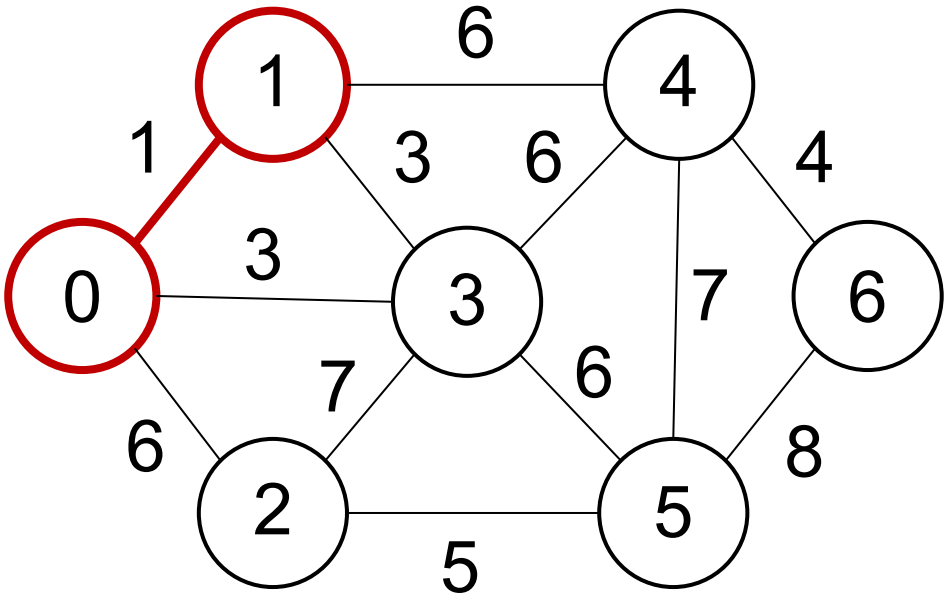
custo = 0
cont = 0

Representantes:

0	1	2	3	4	5	6
0	1	2	3	4	5	6

O arranjo de representantes vai ilustrar o que a função *find* retorna se chamada para aquele vértice.

Árvores geradoras - exemplo



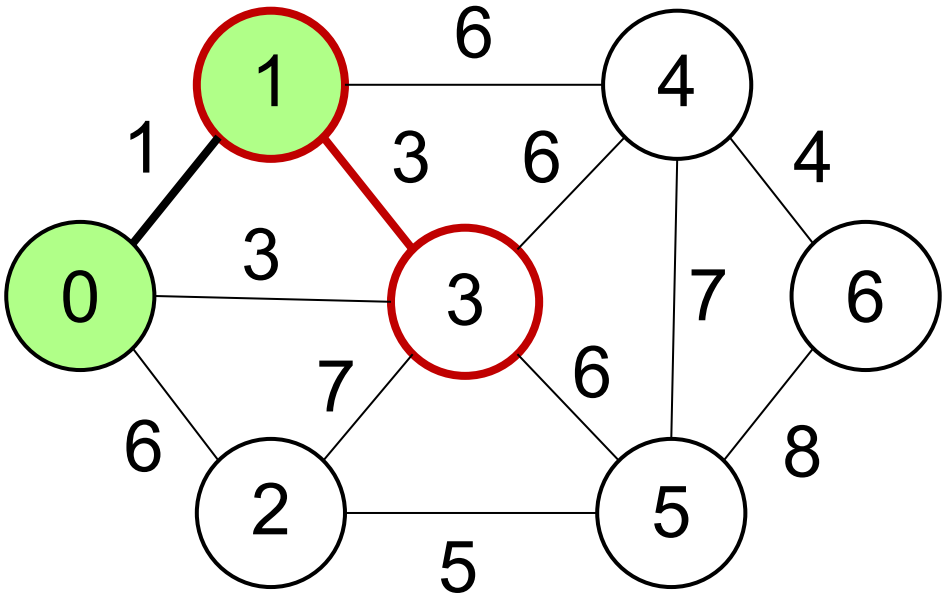
custo = 0
cont = 0

Representantes:

0	1	2	3	4	5	6
0	1	2	3	4	5	6

Agora vamos iterar pelas arestas. A aresta selecionada possui representantes diferentes em cada uma de suas pontas, então iremos chamar a função *union*. Incrementamos *cont* e o custo em 1.

Árvores geradoras - exemplo



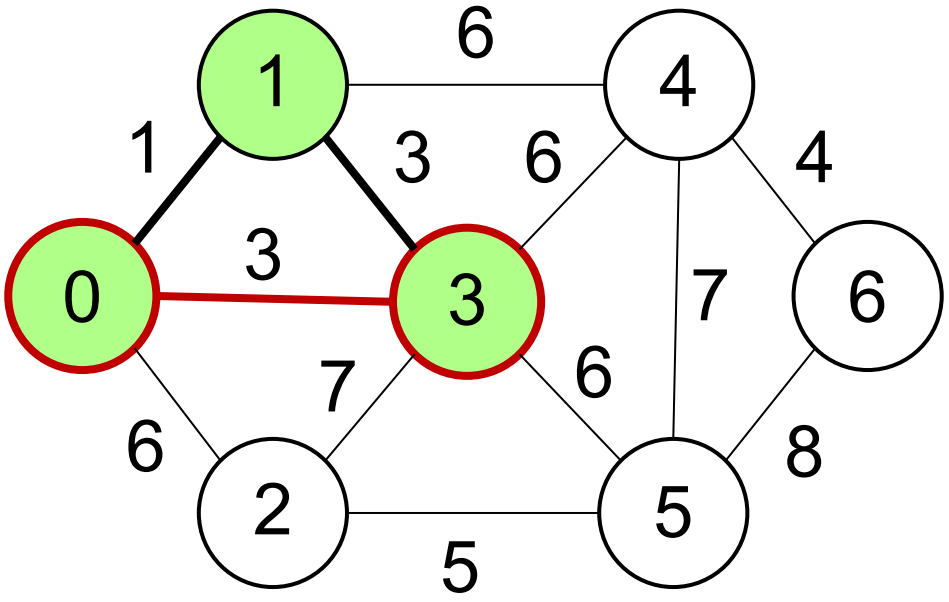
custo = 1
cont = 1

Representantes:

0	1	2	3	4	5	6
0	0	2	3	4	5	6

Mais uma vez os representantes das pontas são diferentes, então executamos o *union* e incrementamos *cont* e o custo.

Árvores geradoras - exemplo



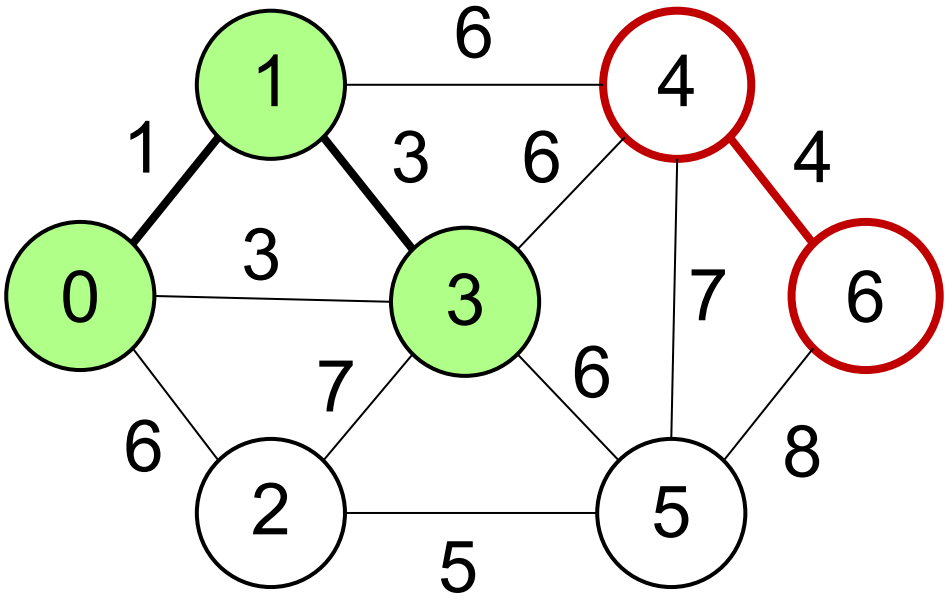
custo = 4
cont = 2

Representantes:

0	1	2	3	4	5	6
0	0	2	0	4	5	6

Dessa vez as duas pontas possuem o mesmo representante. Então não fazemos nada e passamos para a próxima aresta.

Árvores geradoras - exemplo



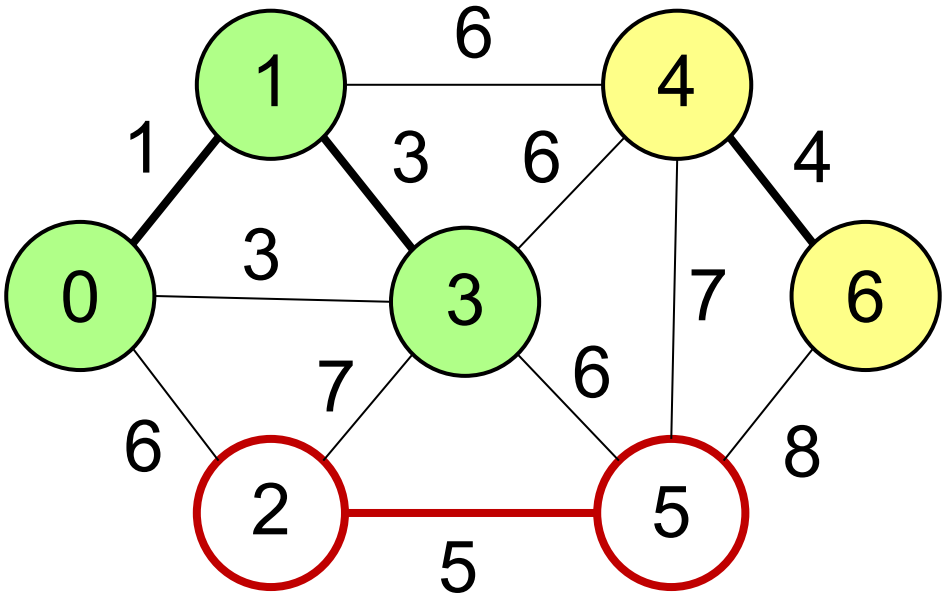
custo = 4
cont = 2

Representantes:

0	1	2	3	4	5	6
0	0	2	0	4	5	6

Mais uma vez os representantes das pontas são diferentes, então executamos o *union* e incrementamos *cont* e o custo.

Árvores geradoras - exemplo



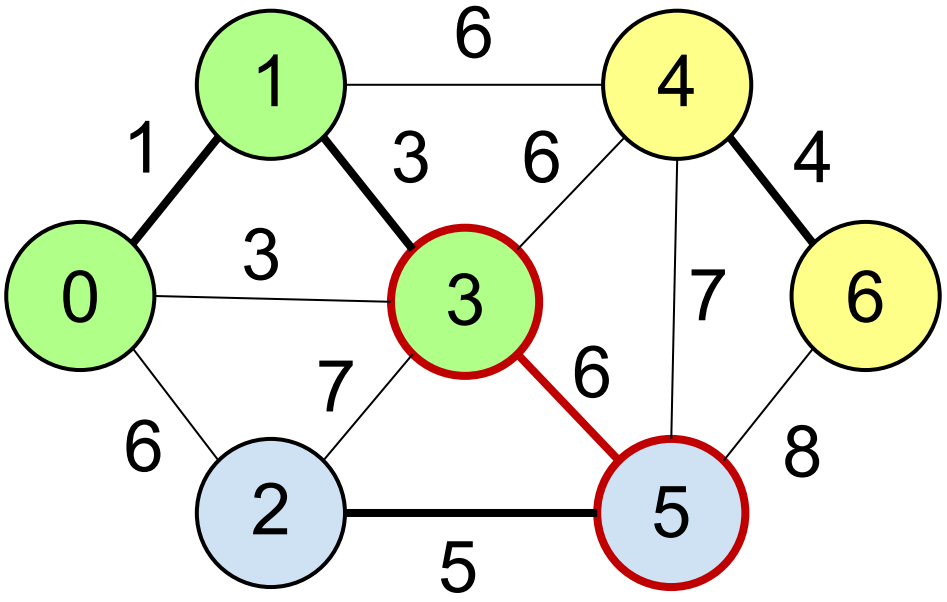
custo = 8
cont = 3

Representantes:

0	1	2	3	4	5	6
0	0	2	0	4	5	4

Mais uma vez os representantes das pontas são diferentes, então executamos o *union* e incrementamos *cont* e o custo.

Árvores geradoras - exemplo



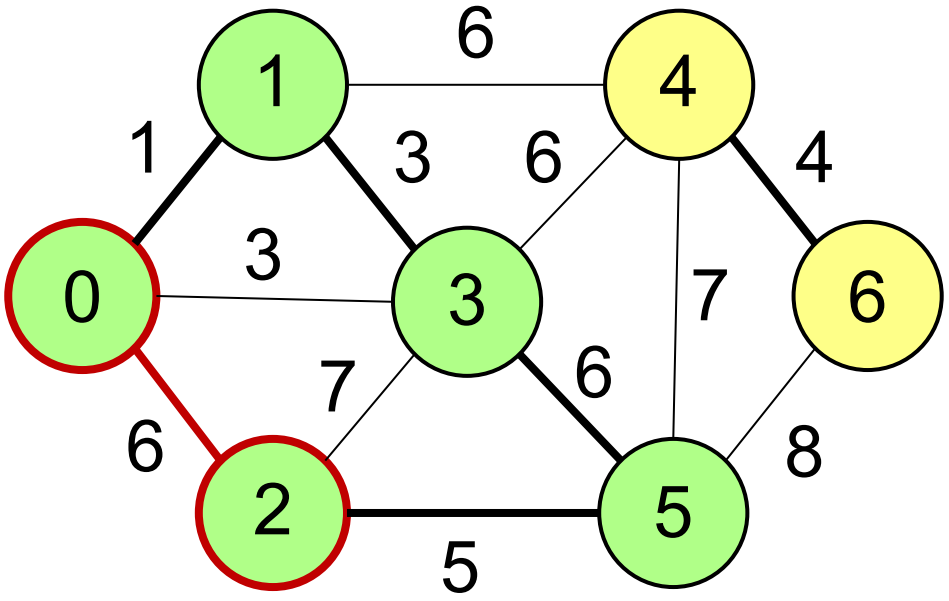
custo = 13
cont = 4

Representantes:

0	1	2	3	4	5	6
0	0	2	0	4	2	4

Mais uma vez os representantes das pontas são diferentes, então executamos o *union* e incrementamos *cont* e o custo.

Árvores geradoras - exemplo



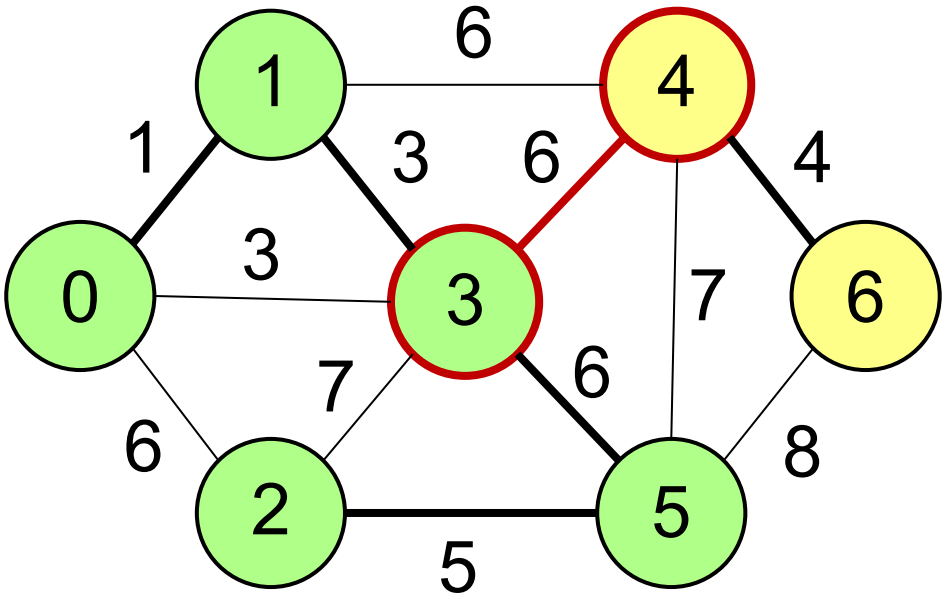
custo = 13
cont = 4

Representantes:

0	1	2	3	4	5	6
0	0	0	0	4	0	4

As duas pontas possuem o mesmo representante. Então não fazemos nada e passamos para a próxima aresta.

Árvores geradoras - exemplo



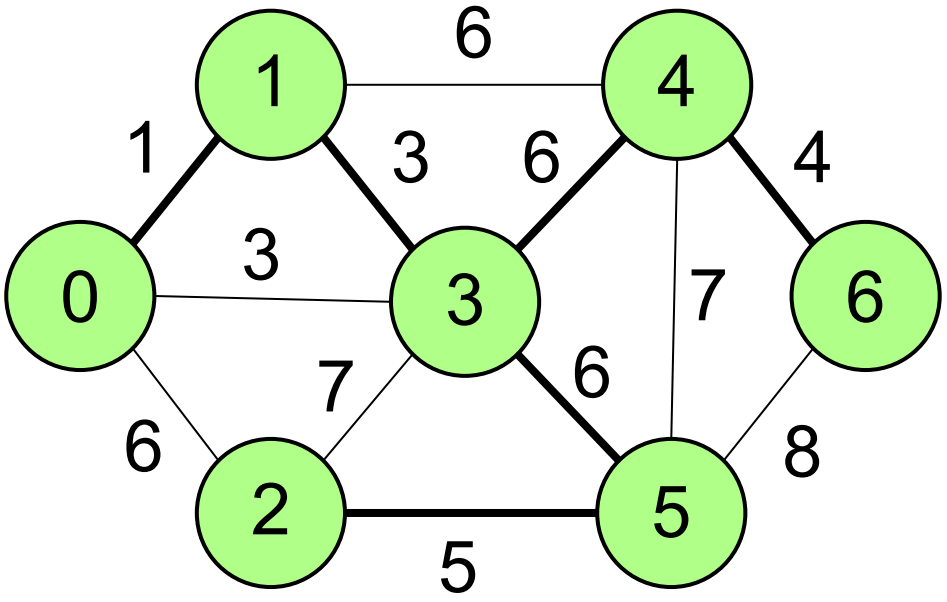
custo = 19
cont = 5

Representantes:

0	1	2	3	4	5	6
0	0	0	0	4	0	4

Mais uma vez os representantes das pontas são diferentes, então executamos o *union* e incrementamos *cont* e o custo.

Árvores geradoras - exemplo



custo = 25
cont = 6

Representantes:

0	1	2	3	4	5	6
0	0	0	0	0	0	0

Como *cont* agora vale *n-1*, o algoritmo para. Note que as arestas ressaltadas formam uma árvore geradora de custo total 25.

Roteiro da atividade

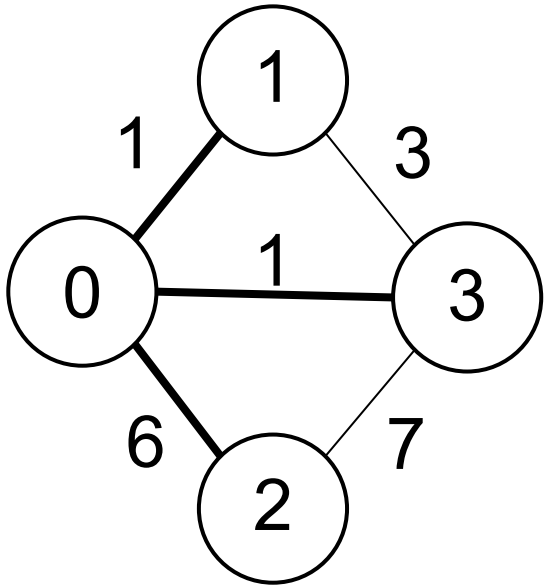
Nesta atividade você deverá implementar um programa que:

1. Lê da entrada padrão:
   - Inteiros *n* e *m* indicando respectivamente a quantidade de vértices e arestas.
   - Seguido de *m* linhas contendo os dados das arestas. Cada uma dessas linhas possui três inteiros *u, v, c,* indicando os vértices *u* e *v* que compões a aresta de custo *c*.
2. Calcule utilizando o algoritmo visto o valor da árvore geradora mínima e o imprima na tela.

Exemplo

Entrada:

```
4 5
0 1 1
0 2 6
0 3 1
1 3 3
2 3 7
```



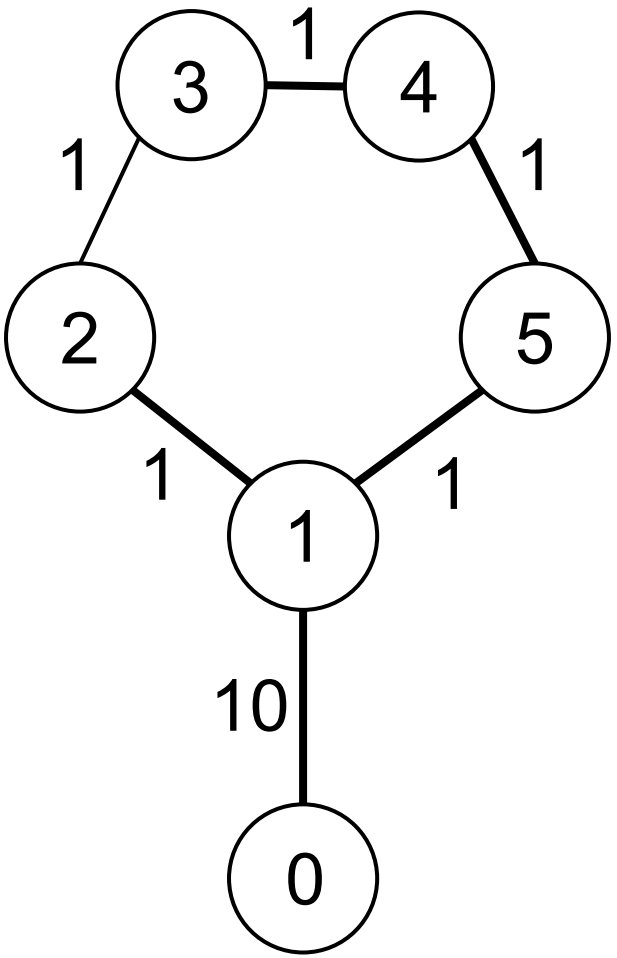
Saída esperada:

```
8
```

Exemplo

Entrada:

```
6 6
0 1 10
1 2 1
1 5 1
2 3 1
3 4 1
4 5 1
```



Saída esperada:

```
14
```

Roteiro da atividade

Observações sobre a implementação:

- **Em C:** Você deve implementar o TAD UnionFind bem como as funções propostas no arquivo unionFind.h
- **Em C++:** Você deve implementar a classe do arquivo unionFind.hpp.
- Em ambos os casos você não deve alterar a interface das funções ou métodos(você não pode adicionar campos ou métodos a classe, nem modificar as assinaturas dos métodos ou funções), mas pode adicionar os includes de seus TADs.

unionFind.h - Interface do TAD e suas funções

```
typedef struct s_edge{
    int u;
    int v;
    int custo;
} Aresta;
typedef struct s_subset{
    int representante;
    int rank;
} Subconjunto;
typedef struct s_dsu{
    int representante;
    int rank;
} UnionFind;
UnionFind* NovoUnionFind( int quantidade_subconjuntos);
void DeletaUnionFind(UnionFind* dsu);
void Make(UnionFind* dsu, int x);
int Find(UnionFind* dsu, int x);
void Union(UnionFind* dsu,int x, int y);
```

unionFind.hpp - Interface da classe

```
typedef struct s_edge{
    int u;
    int v;
    int custo;
} Aresta;
typedef struct s_subset{
    int representante;
    int rank;
} Subconjunto;
class UnionFind{
    public:
        UnionFind(int quantidade_subconjuntos);
        ~UnionFind();
        void Make(int x);
        int Find(int x);
        void Union(int x, int y);
    private:
        int tamanho;
        Subconjunto* subconjuntos;
};
```

Submissão

- A submissão será feita por **VPL**. Certifique-se de seguir as instruções do tutorial disponibilizado no moodle.
- O seu arquivo executável **DEVE** se chamar **pa8.out** e deve estar localizado na pasta **bin**.
- Seu código será compilado com o comando:

  make all
- Você **DEVE** utilizar a estrutura de projeto abaixo junto ao Makefile :

```
- PA8
   |- src
   |- bin
   |- obj
   |- include
   Makefile
```