DCC
DEPARTAMENTO DE
CIÊNCIA DA COMPUTAÇÃO

UFMG
UNIVERSIDADE FEDERAL
DE MINAS GERAIS

Estruturas de Dados

Aplicações de Conjuntos disjuntos em grafos

Professores: Anisio Lacerda
Wagner Meira Jr.

Aplicações de DSU em grafos

- A estrutura DSU é comumente utilizada em problemas que requerem o particionamento de elementos de um conjunto.

Aplicações de DSU em grafos

- A estrutura DSU é comumente utilizada em problemas que requerem o particionamento de elementos de um conjunto.
- Em particular, DSU aparece em várias soluções clássicas para problemas em grafos.

Aplicações de DSU em grafos

- A estrutura DSU é comumente utilizada em problemas que requerem o particionamento de elementos de um conjunto.
- Em particular, DSU aparece em várias soluções clássicas para problemas em grafos.
- Vamos falar sobre 3 dessas aplicações:

Aplicações de DSU em grafos

- A estrutura DSU é comumente utilizada em problemas que requerem o particionamento de elementos de um conjunto.
- Em particular, DSU aparece em várias soluções clássicas para problemas em grafos.
- Vamos falar sobre 3 dessas aplicações:
  - Identificar componentes conexos.

Aplicações de DSU em grafos

- A estrutura DSU é comumente utilizada em problemas que requerem o particionamento de elementos de um conjunto.
- Em particular, DSU aparece em várias soluções clássicas para problemas em grafos.
- Vamos falar sobre 3 dessas aplicações:
  - Identificar componentes conexos.
  - Detecção de ciclos.

Aplicações de DSU em grafos

- A estrutura DSU é comumente utilizada em problemas que requerem o particionamento de elementos de um conjunto.
- Em particular, DSU aparece em várias soluções clássicas para problemas em grafos.
- Vamos falar sobre 3 dessas aplicações:
    - Identificar componentes conexos.
    - Detecção de ciclos.
    - Árvore geradora mínima.

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*.

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.

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 conexo que não possui ciclos.

Aplicações de DSU em grafos

- **Identificar componentes conexos.**
- Detecção de ciclos.
- Árvore geradora mínima.

Identificar componentes conexas

- Quando um grafo não é conexo, seus vértices podem ser particionados em **componentes conexos**.

Identificar componentes conexas

- Quando um grafo não é conexo, seus vértices podem ser particionados em **componentes conexos**.
- Um componente conexo é um subgrafo conexo.

Identificar componentes conexas

- Quando um grafo não é conexo, seus vértices podem ser particionados em **componentes conexos**.
- Um componente conexo é um subgrafo conexo.
- Vamos utilizar o DSU para detectar quantas componentes conexos um grafo possui e a qual componente cada vértice pertence.

Identificar componentes conexas

- Nosso conjunto universo serão os vértices do grafo, e nossa coleção de conjuntos serão os componentes conexos.

Identificar componentes conexas

- Nosso conjunto universo serão os vértices do grafo, e nossa coleção de conjuntos serão os componentes conexos.

O algoritmo consiste em:

- Inicialmente cada vértice *v* estará em um componente conexo que possui apenas *v*.

Identificar componentes conexas

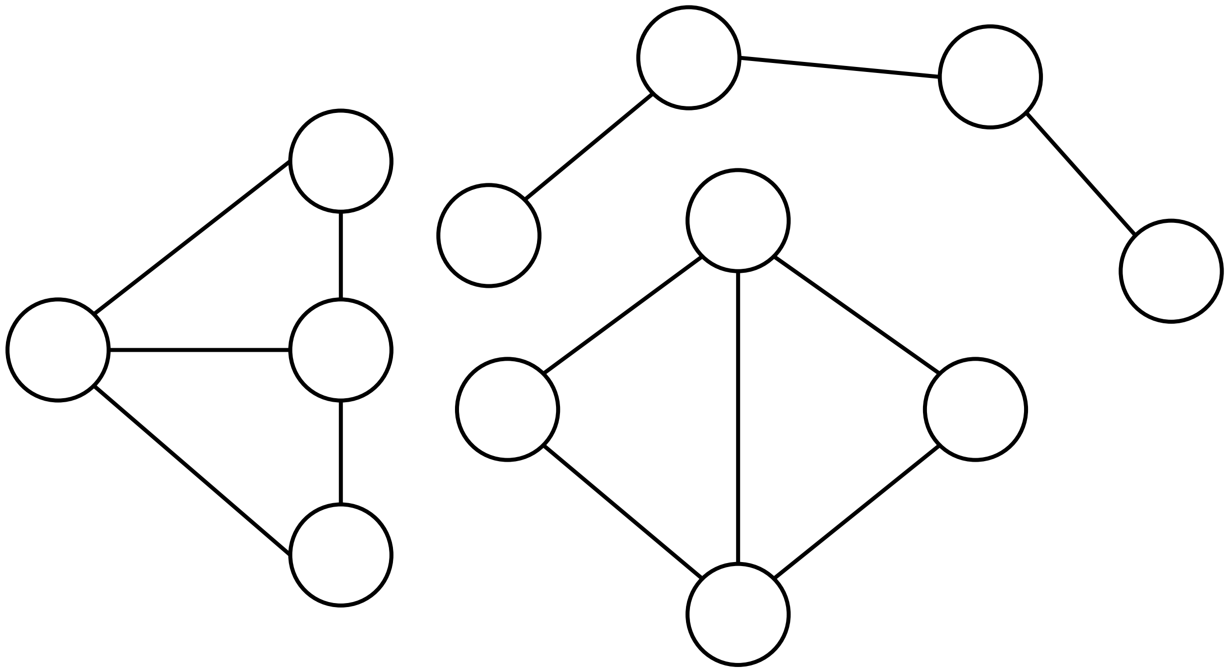
- Nosso conjunto universo serão os vértices do grafo, e nossa coleção de conjuntos serão os componentes conexos.

O algoritmo consiste em:

- Inicialmente cada vértice *v* estará em um componente conexo que possui apenas *v*.
- Itere pelas arestas e para cada aresta *e* chame *union* para os representantes dos dois vértices de *e*.

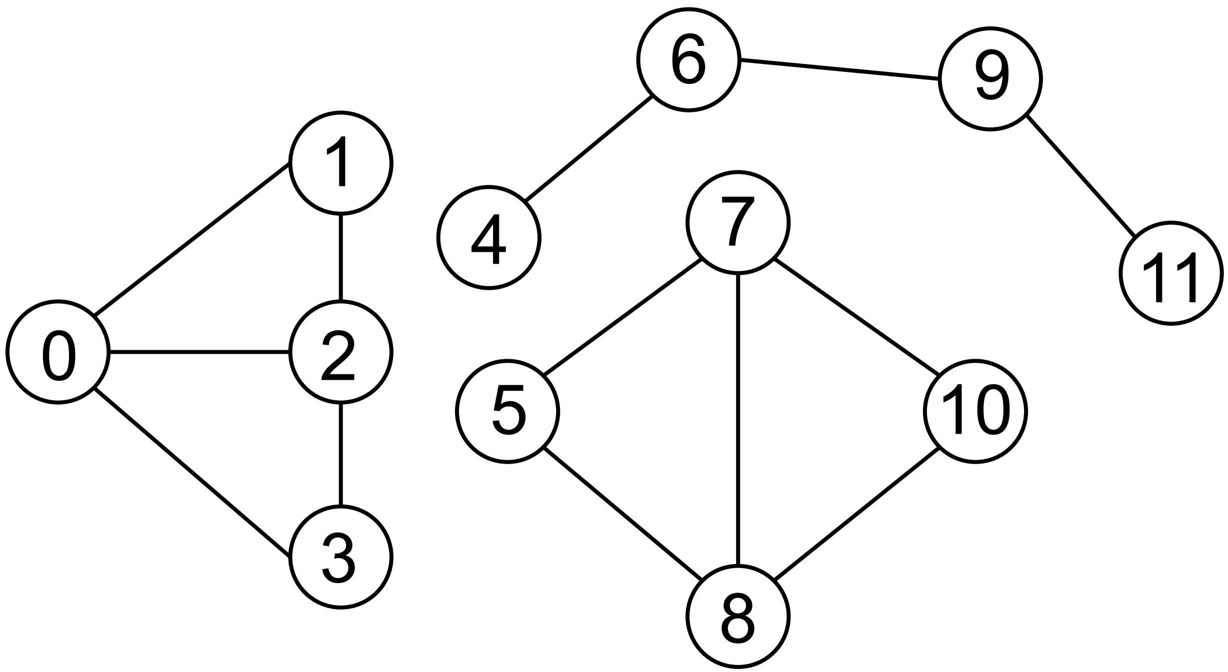
Identificar componentes conexas

Considere o grafo abaixo de exemplo:



Identificar componentes conexas

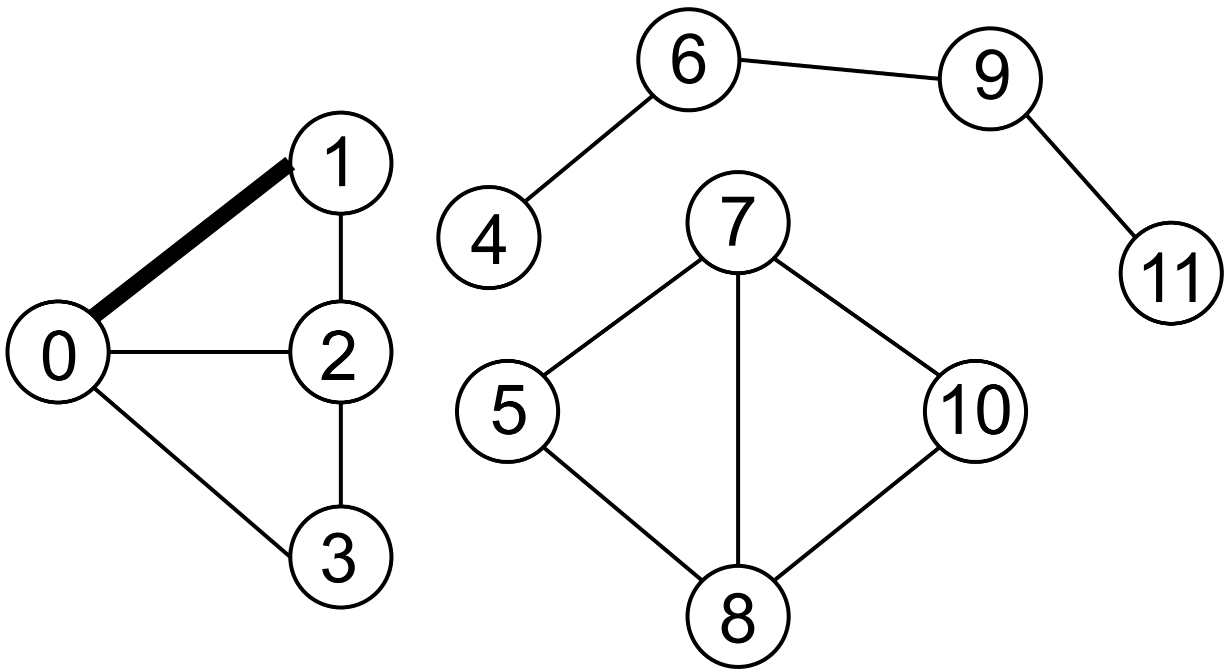
Rotulando os vértices e criando a DSU inicial:



0	1	2	3	4	5	6	7	8	9	10	11
0	1	2	3	4	5	6	7	8	9	10	11

Identificar componentes conexas

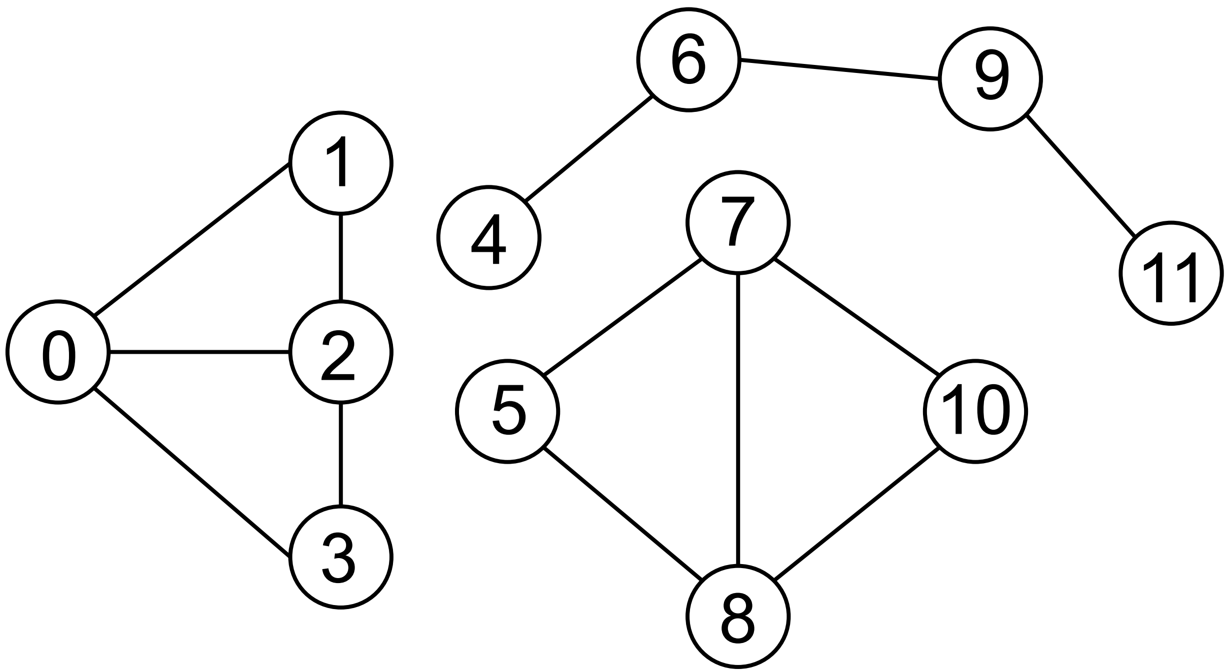
Para cada aresta, faça *union* de seus representantes:



0	1	2	3	4	5	6	7	8	9	10	11
0	1	2	3	4	5	6	7	8	9	10	11

Identificar componentes conexas

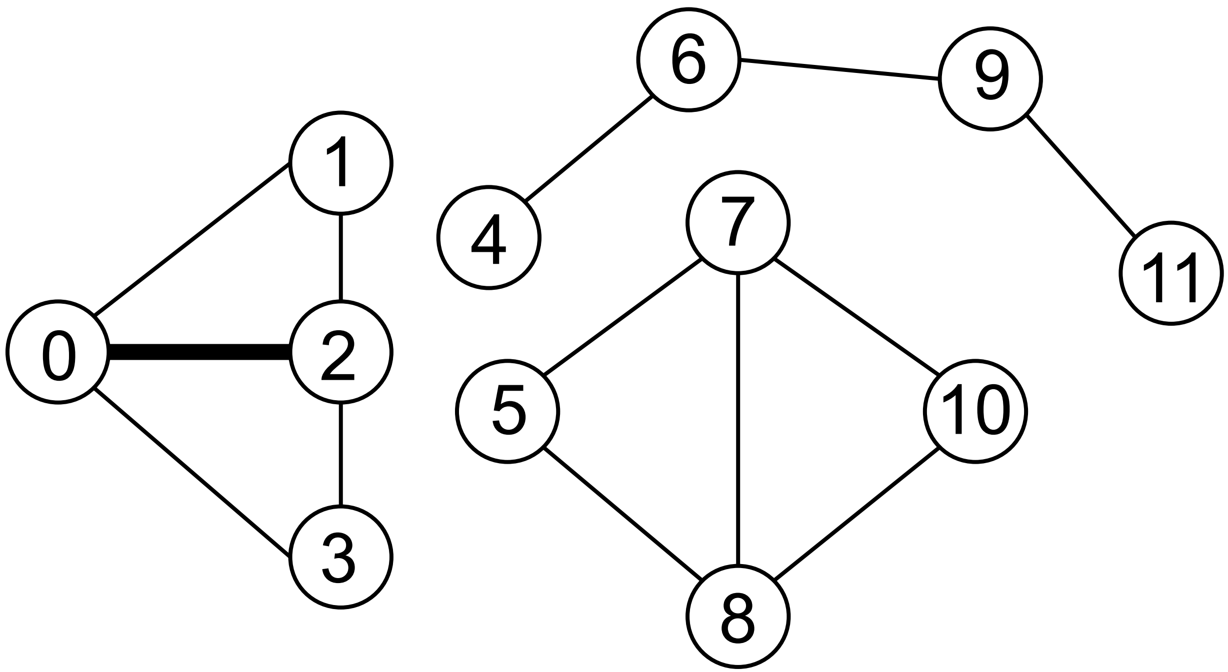
Para cada aresta, faça *union* de seus representantes:



0	1	2	3	4	5	6	7	8	9	10	11
0	0	2	3	4	5	6	7	8	9	10	11

Identificar componentes conexas

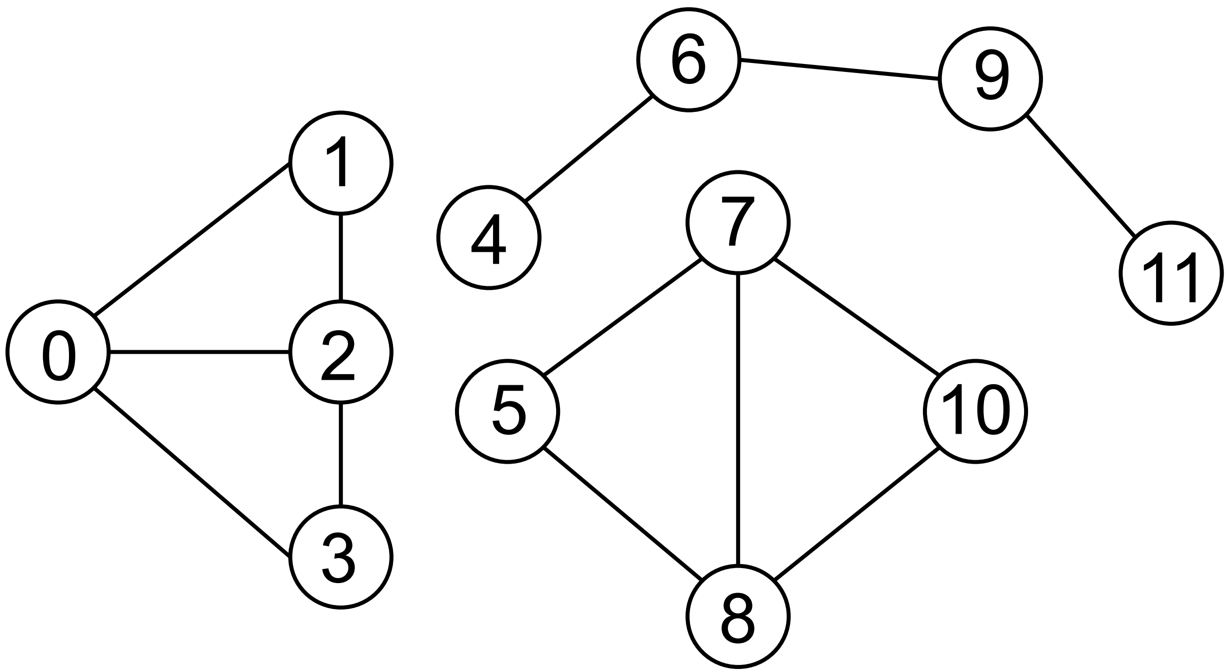
Para cada aresta, faça *union* de seus representantes:



0	1	2	3	4	5	6	7	8	9	10	11
0	0	2	3	4	5	6	7	8	9	10	11

Identificar componentes conexas

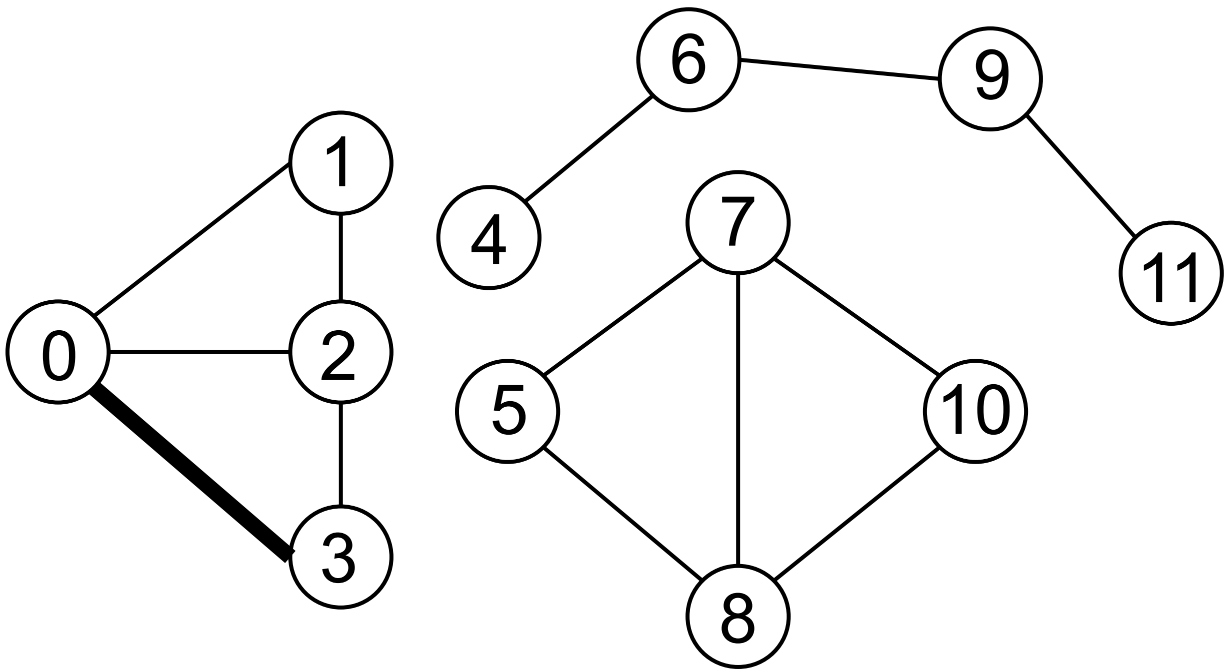
Para cada aresta, faça *union* de seus representantes:



0	1	2	3	4	5	6	7	8	9	10	11
0	0	0	3	4	5	6	7	8	9	10	11

Identificar componentes conexas

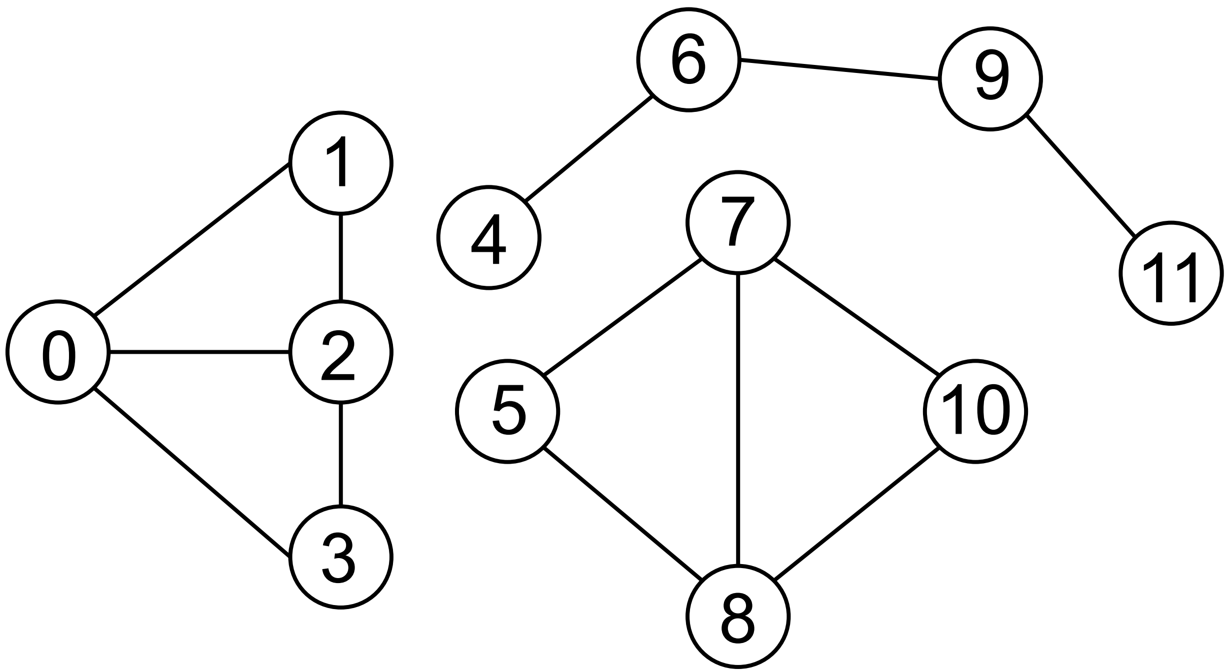
Para cada aresta, faça *union* de seus representantes:



0	1	2	3	4	5	6	7	8	9	10	11
0	0	0	3	4	5	6	7	8	9	10	11

Identificar componentes conexas

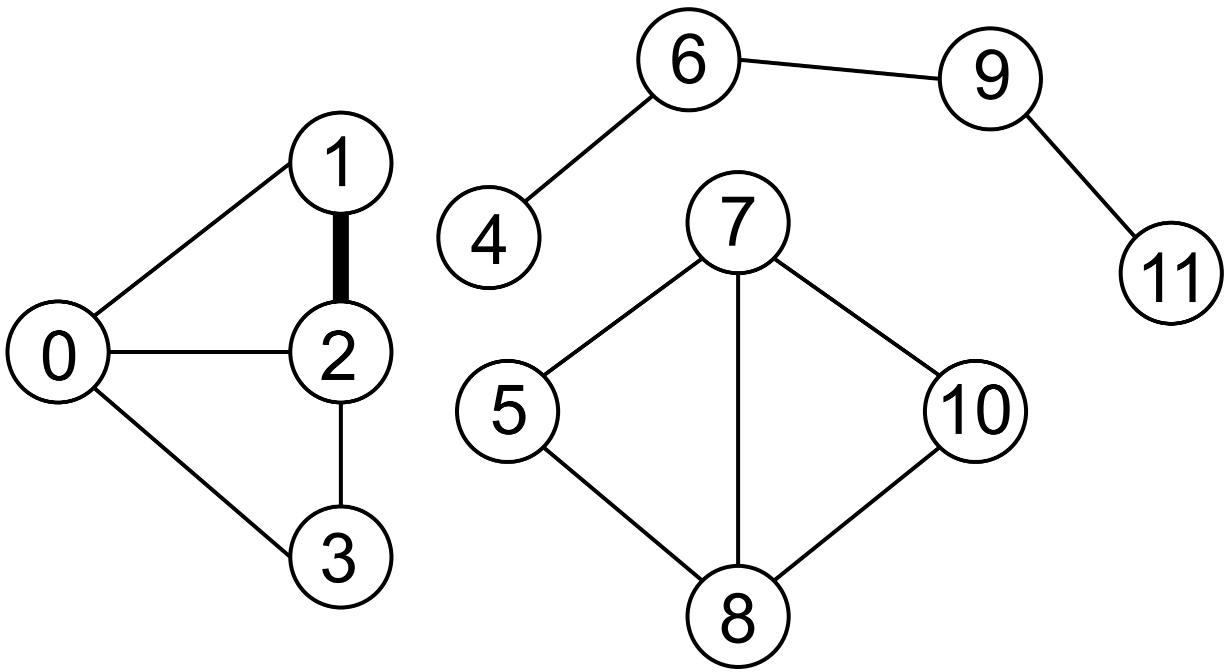
Para cada aresta, faça *union* de seus representantes:



0	1	2	3	4	5	6	7	8	9	10	11
0	0	0	0	4	5	6	7	8	9	10	11

Identificar componentes conexas

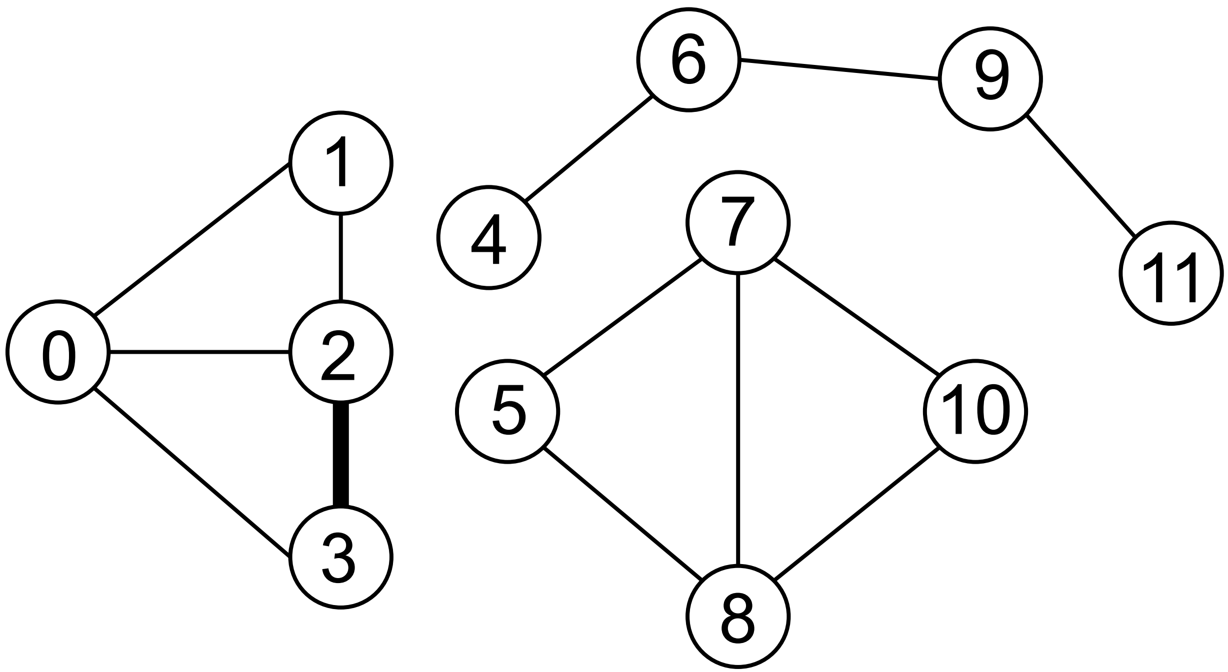
Para cada aresta, faça *union* de seus representantes:



0	1	2	3	4	5	6	7	8	9	10	11
0	0	0	0	4	5	6	7	8	9	10	11

Identificar componentes conexas

Para cada aresta, faça *union* de seus representantes:

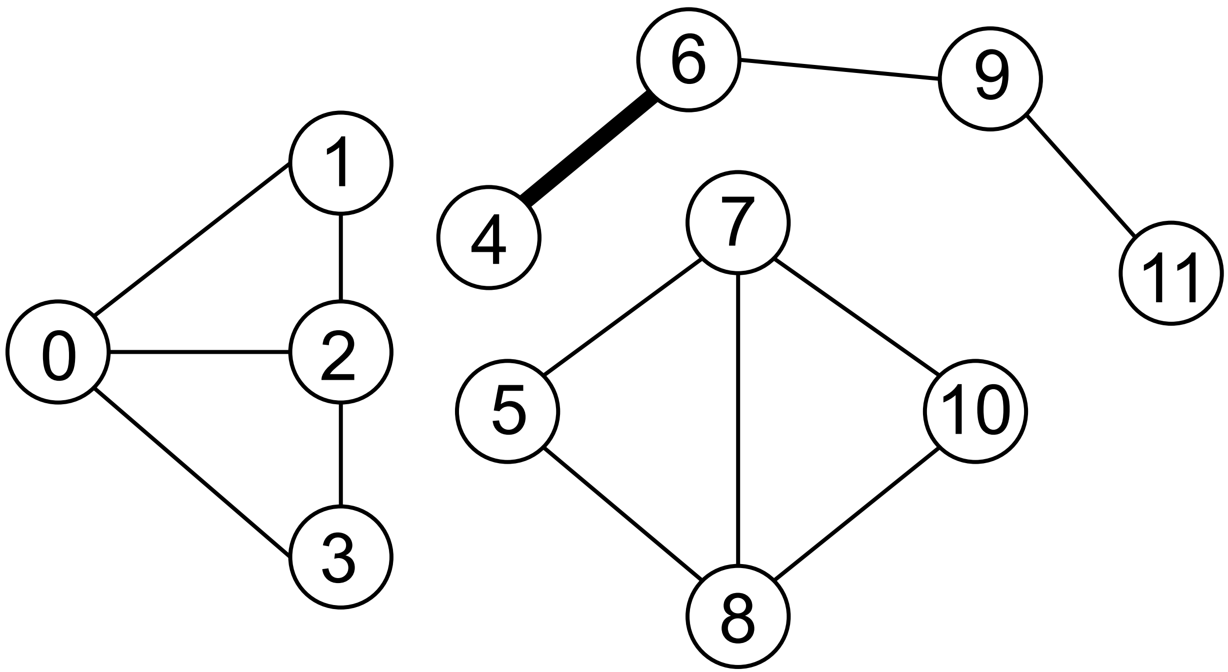


0	1	2	3	4	5	6	7	8	9	10	11
0	0	0	0	4	5	6	7	8	9	10	11

DCC

Identificar componentes conexas

Para cada aresta, faça *union* de seus representantes:

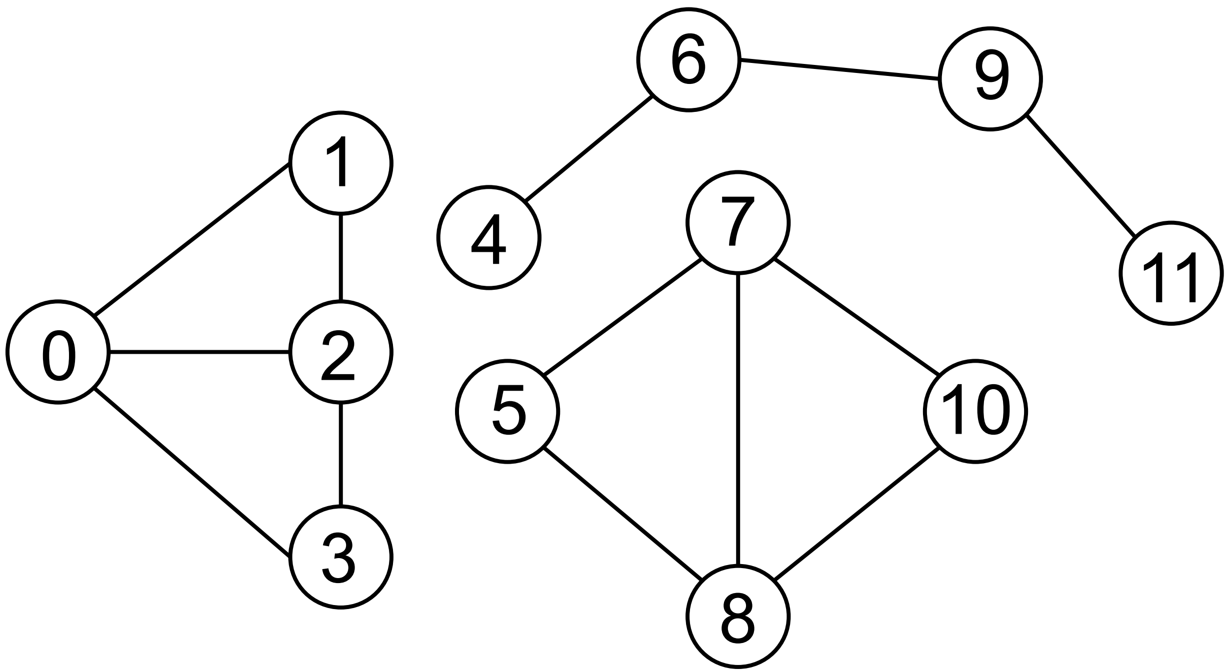


0	1	2	3	4	5	6	7	8	9	10	11
0	0	0	0	4	5	6	7	8	9	10	11

DCC

Identificar componentes conexas

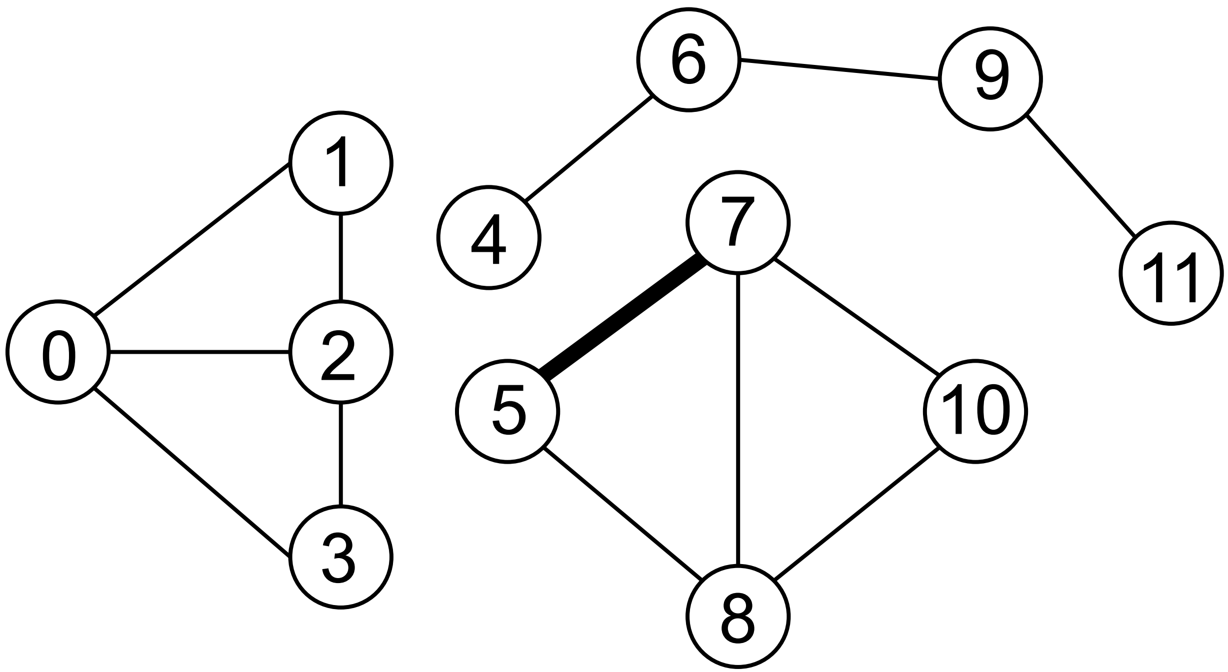
Para cada aresta, faça *union* de seus representantes:



0	1	2	3	4	5	6	7	8	9	10	11
0	0	0	0	4	5	4	7	8	9	10	11

Identificar componentes conexas

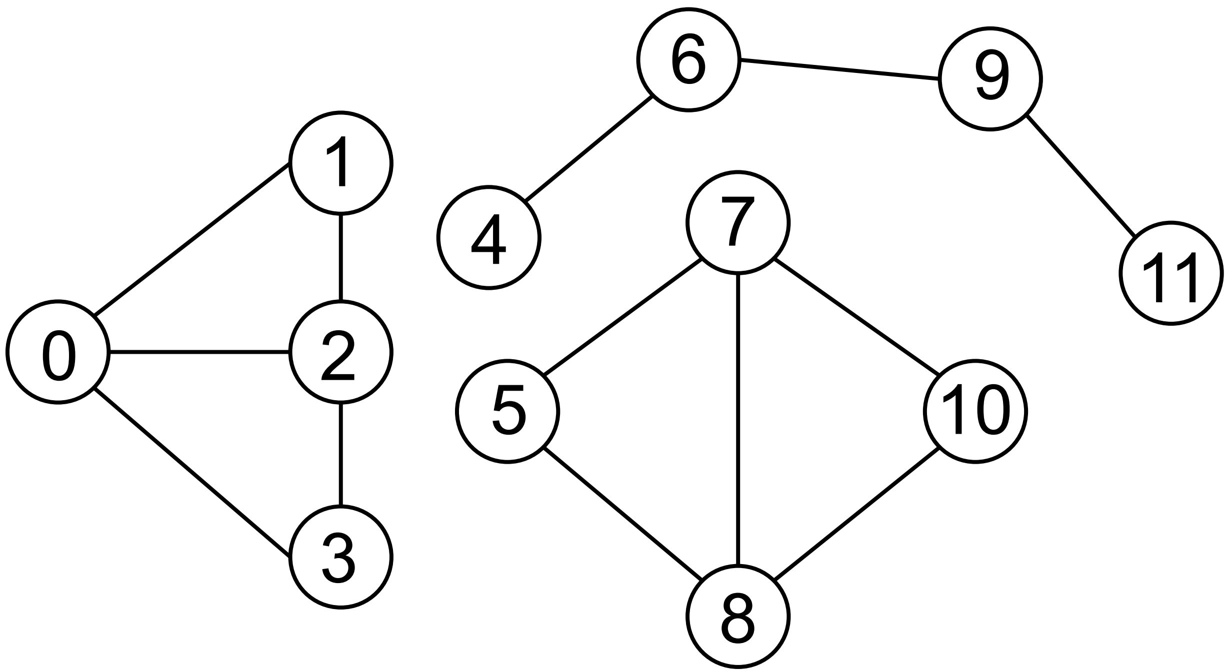
Para cada aresta, faça *union* de seus representantes:



0	1	2	3	4	5	6	7	8	9	10	11
0	0	0	0	4	5	4	7	8	9	10	11

Identificar componentes conexas

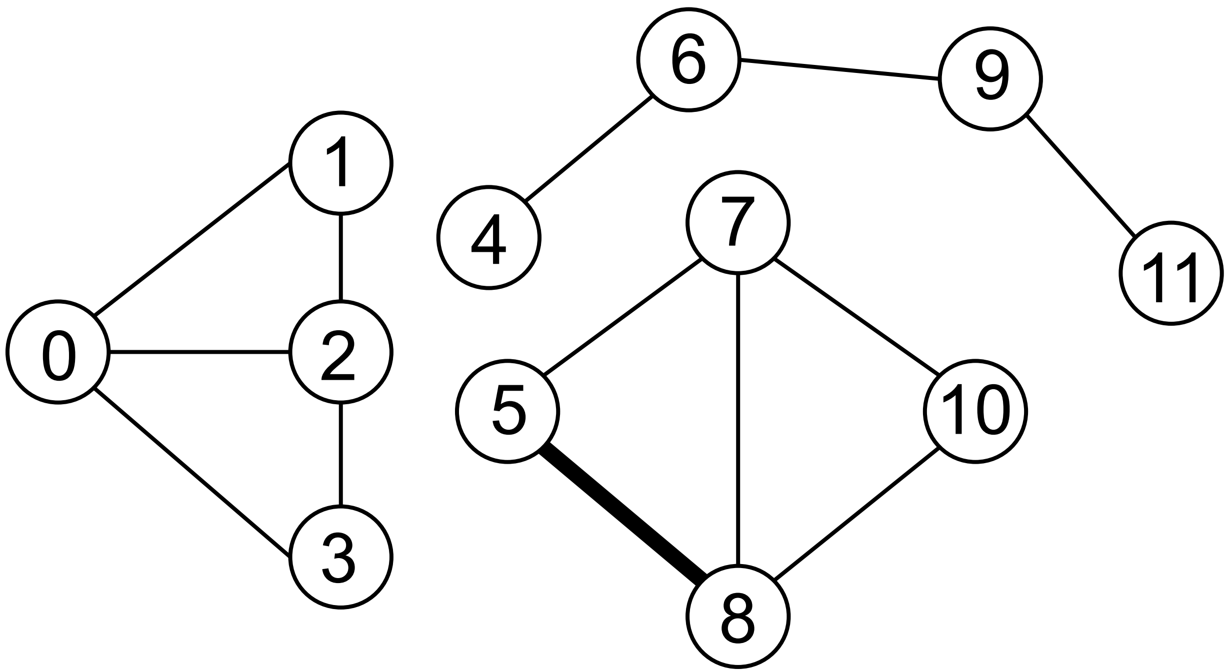
Para cada aresta, faça *union* de seus representantes:



0	1	2	3	4	5	6	7	8	9	10	11
0	0	0	0	4	5	4	5	8	9	10	11

Identificar componentes conexas

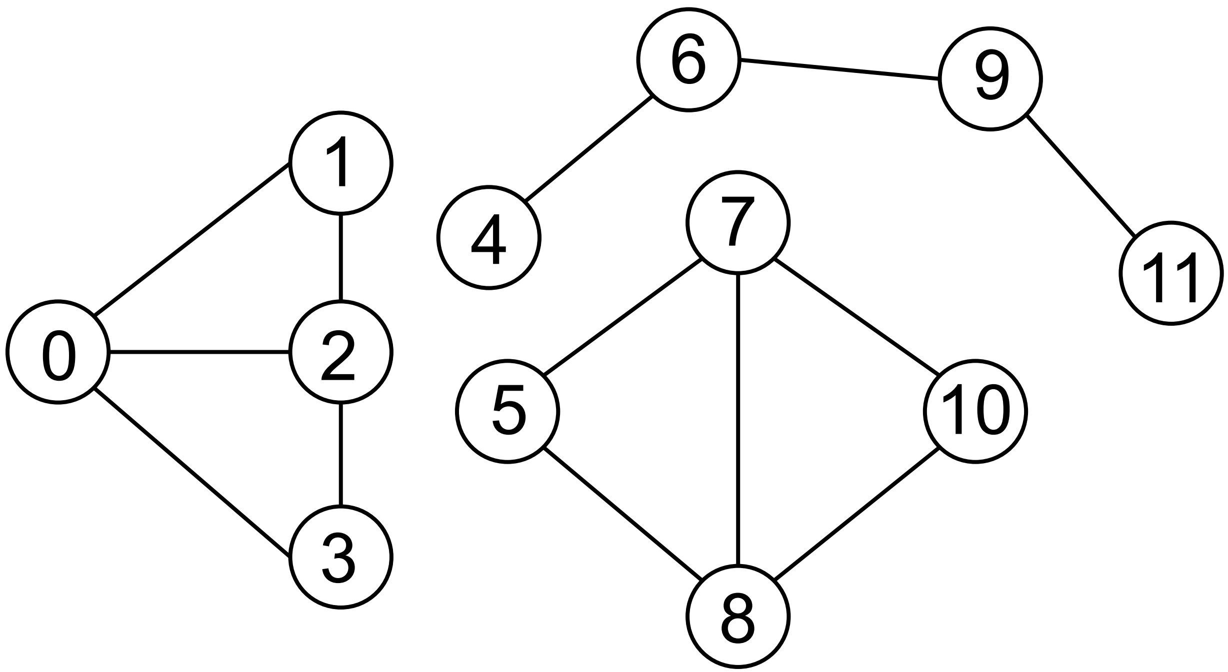
Para cada aresta, faça *union* de seus representantes:



0	1	2	3	4	5	6	7	8	9	10	11
0	0	0	0	4	5	4	5	8	9	10	11

Identificar componentes conexas

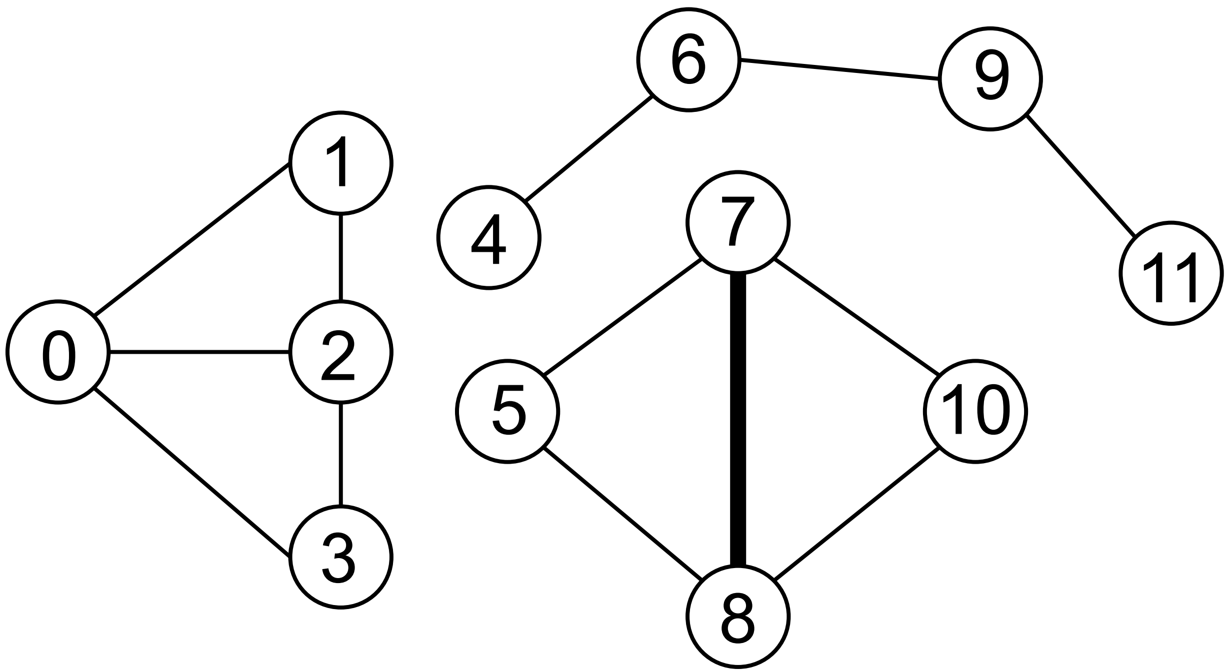
Para cada aresta, faça *union* de seus representantes:



0	1	2	3	4	5	6	7	8	9	10	11
0	0	0	0	4	5	4	5	5	9	10	11

Identificar componentes conexas

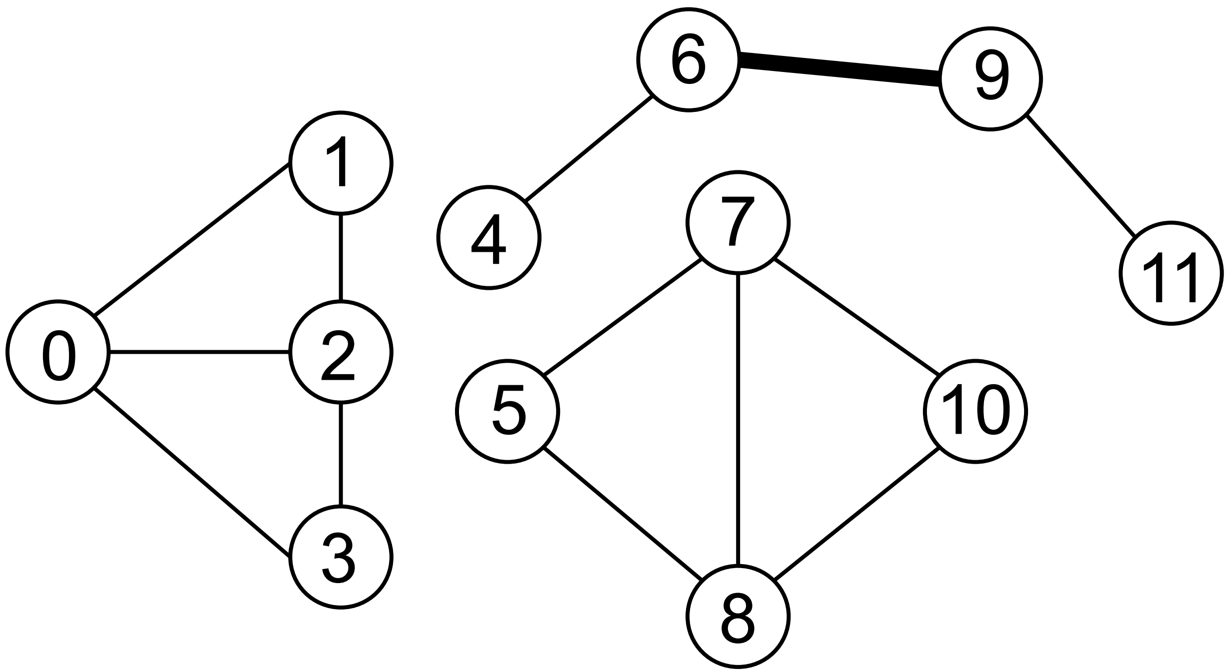
Para cada aresta, faça *union* de seus representantes:



0	1	2	3	4	5	6	7	8	9	10	11
0	0	0	0	4	5	4	5	5	9	10	11

Identificar componentes conexas

Para cada aresta, faça *union* de seus representantes:

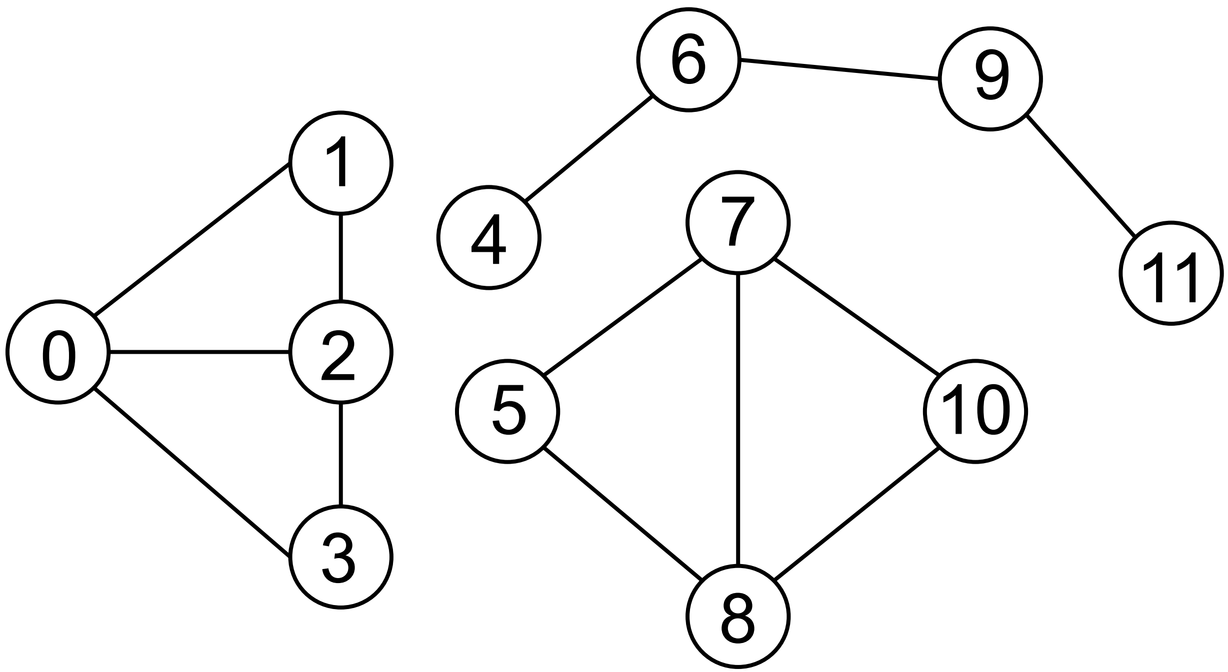


0	1	2	3	4	5	6	7	8	9	10	11
0	0	0	0	4	5	4	5	5	9	10	11

DCC

Identificar componentes conexas

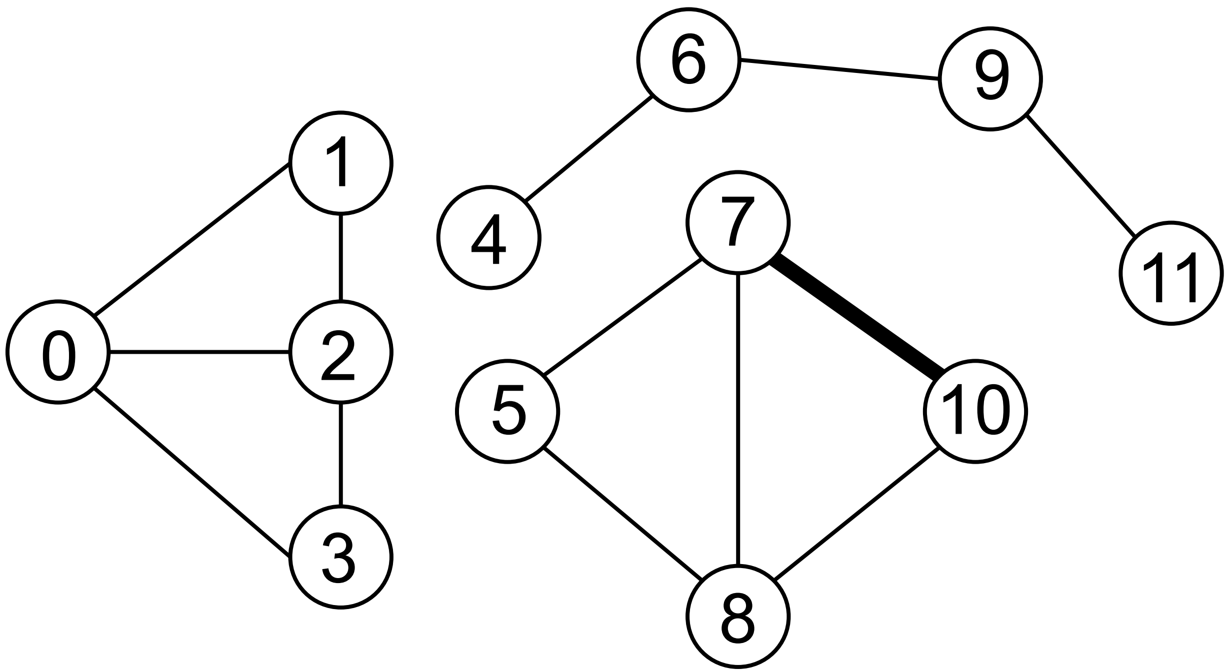
Para cada aresta, faça *union* de seus representantes:



0	1	2	3	4	5	6	7	8	9	10	11
0	0	0	0	4	5	4	5	5	4	10	11

Identificar componentes conexas

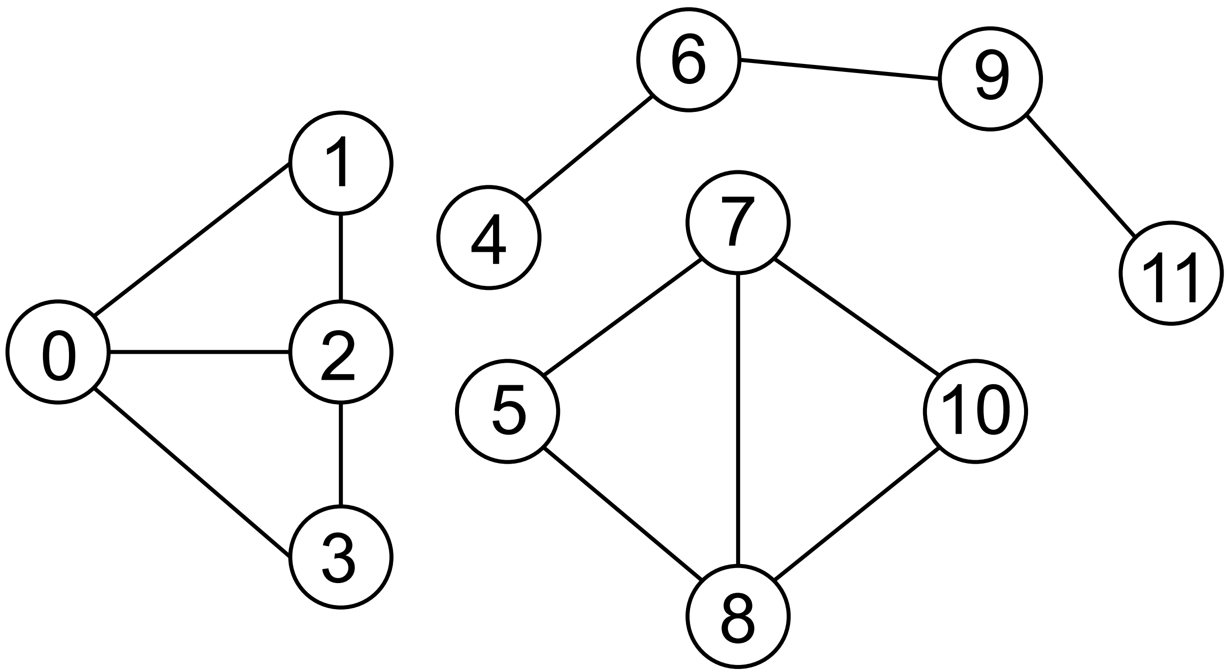
Para cada aresta, faça *union* de seus representantes:



0	1	2	3	4	5	6	7	8	9	10	11
0	0	0	0	4	5	4	5	5	4	10	11

Identificar componentes conexas

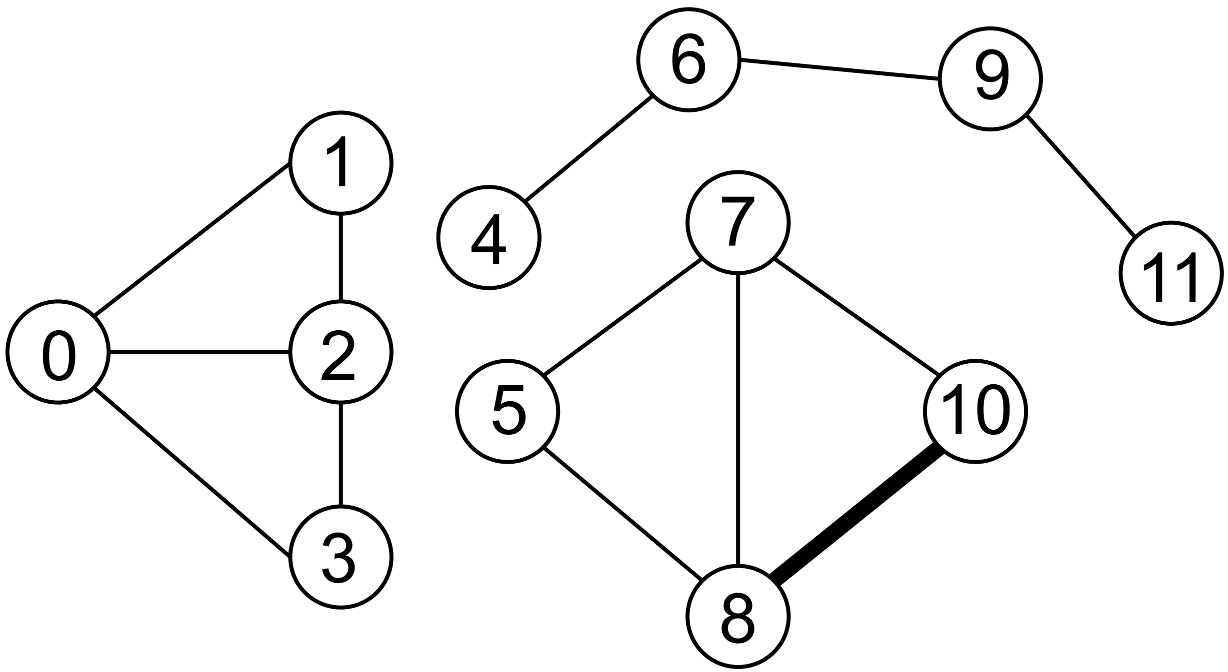
Para cada aresta, faça *union* de seus representantes:



0	1	2	3	4	5	6	7	8	9	10	11
0	0	0	0	4	5	4	5	5	4	5	11

Identificar componentes conexas

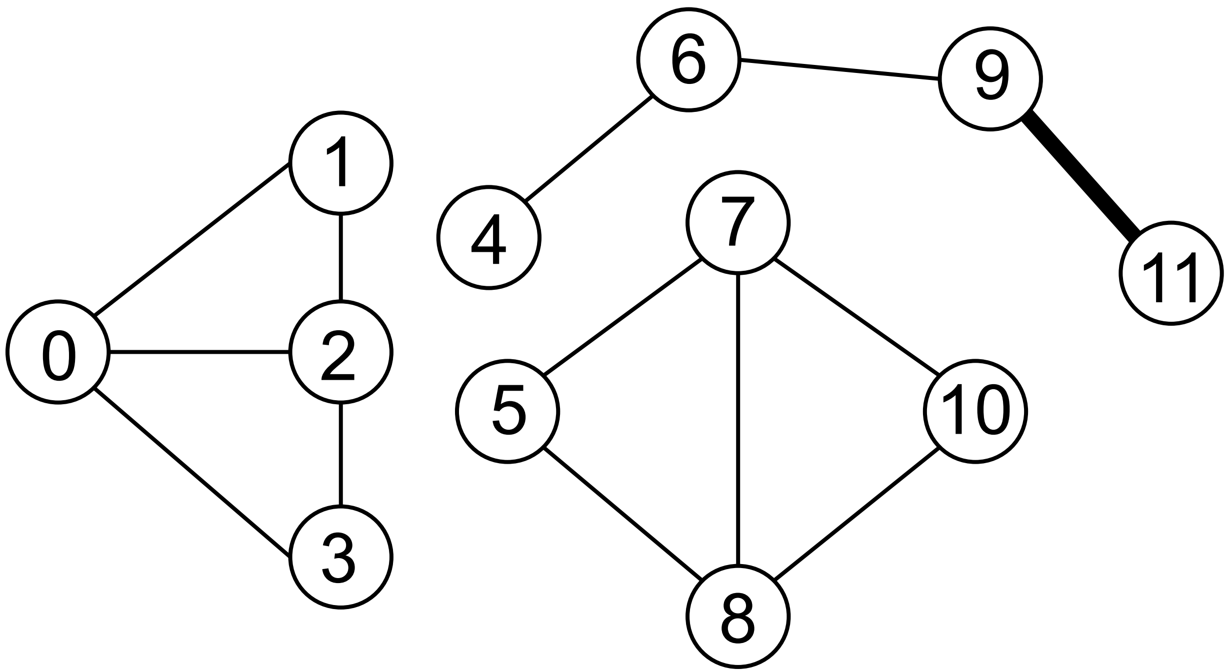
Para cada aresta, faça *union* de seus representantes:



0	1	2	3	4	5	6	7	8	9	10	11
0	0	0	0	4	5	4	5	5	4	5	11

Identificar componentes conexas

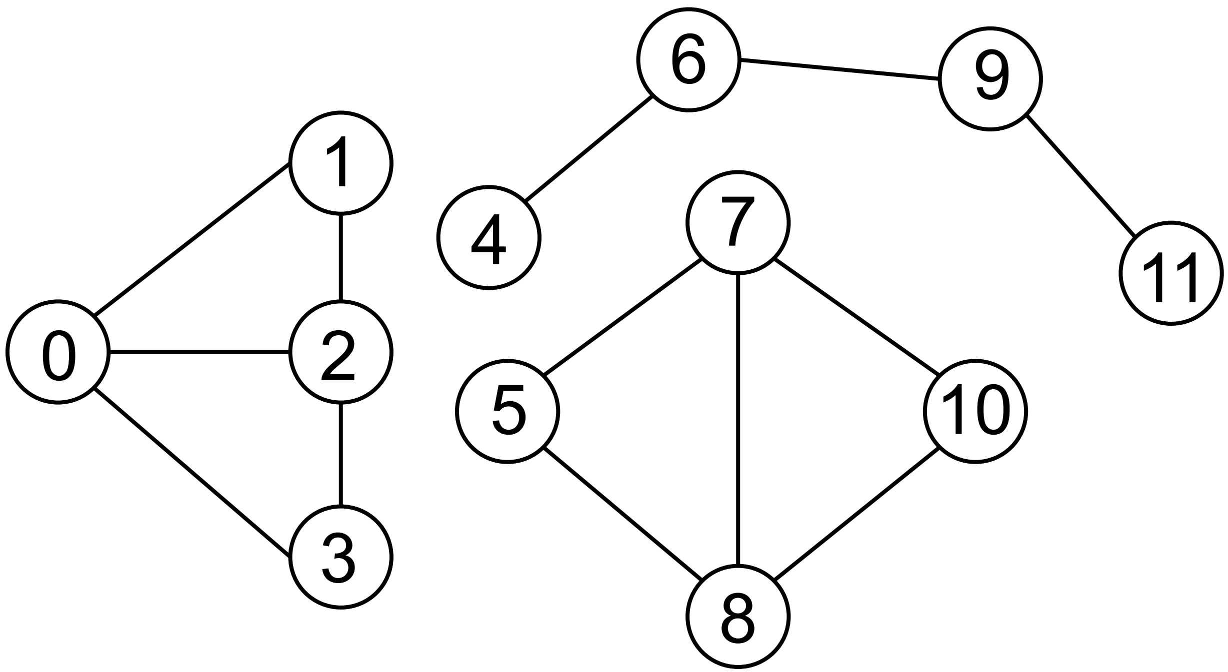
Para cada aresta, faça *union* de seus representantes:



0	1	2	3	4	5	6	7	8	9	10	11
0	0	0	0	4	5	4	5	5	4	5	11

Identificar componentes conexas

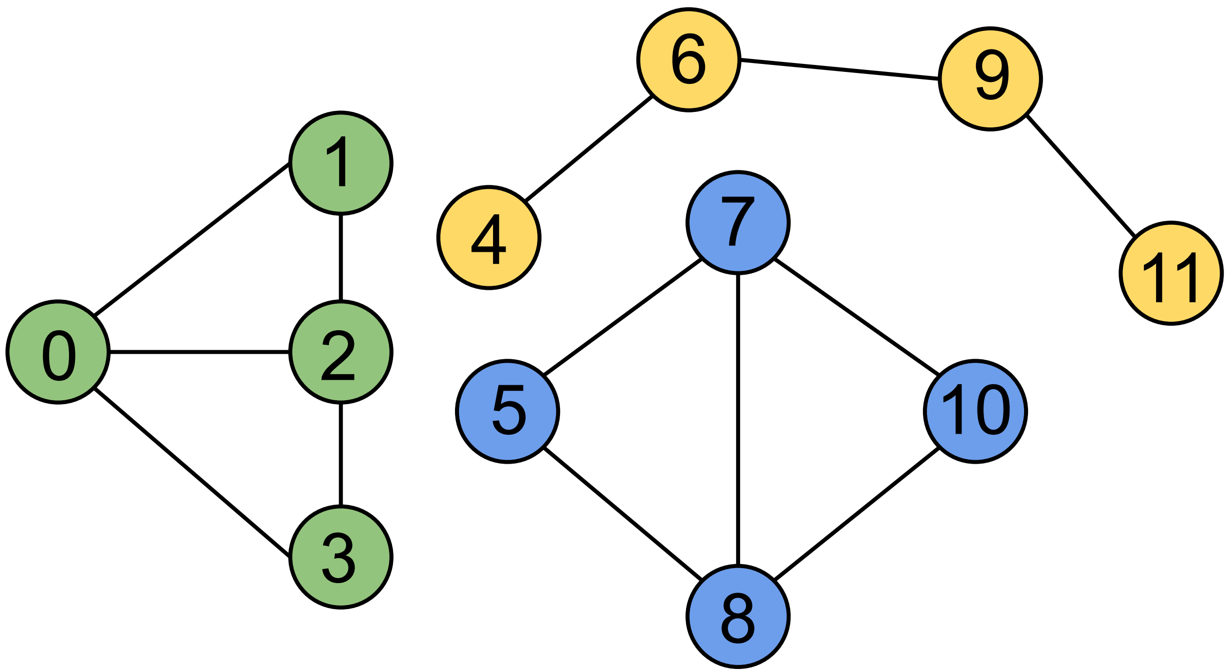
Para cada aresta, faça *union* de seus representantes:



0	1	2	3	4	5	6	7	8	9	10	11
0	0	0	0	4	5	4	5	5	4	5	4

Identificar componentes conexas

Agora podemos identificar as componentes conexas



0	1	2	3	4	5	6	7	8	9	10	11
0	0	0	0	4	5	4	5	5	4	5	4

Identificar componentes conexas

ComponentesConexas (*V,E,DSU*)

Para cada vértice *v* de *V*

DSU[v] = *v*

Para cada aresta *e(u,v)* de *E*

Se **Find**(*u*) != **Find**(*v*)

Union(*u,v*)

Para cada vértice *v* de *V*

Se *DSU[v]* == *v*

Print "v é componente"

DCC

Identificar componentes conexas

Qual a complexidade do algoritmo?

Identificar componentes conexas

Qual a complexidade do algoritmo?

- Considerando DSU implementado em árvores, com união por *rank* e *path compression* a complexidade é $O(|E|\alpha(|V|))$.

DCC

Identificar componentes conexas

Qual a complexidade do algoritmo?

- Considerando DSU implementado em árvores, com união por *rank* e *path compression* a complexidade é $O(|E|\alpha(|V|))$.
- Como a função α cresce muito devagar, podemos dizer que tempo de execução do algoritmo praticamente depende apenas da quantidade de arestas do grafo.

Aplicações de DSU em grafos

- Identificar componentes conexas.
- **Detecção de ciclos.**
- Árvore geradora mínima.

Detectar existência de ciclos

- O algoritmo funciona de forma similar ao anterior.

Detectar existência de ciclos

- O algoritmo funciona de forma similar ao anterior.
- Dado um DSU com os vértices do grafo, itere por todas as arestas.

Detectar existência de ciclos

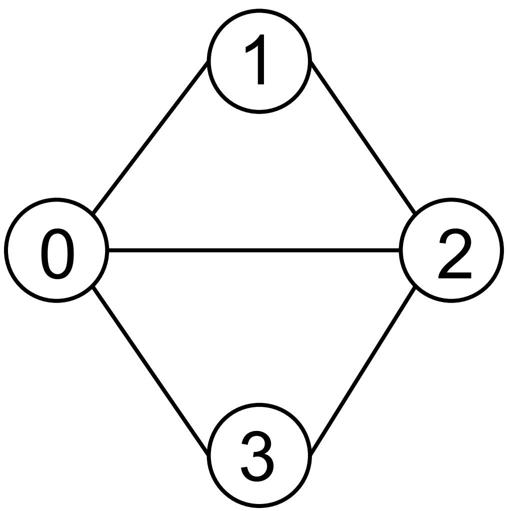
- O algoritmo funciona de forma similar ao anterior.
- Dado um DSU com os vértices do grafo, itere por todas as arestas.
    - Se em alguma aresta os vértices já pertencem à mesma componente conexa antes do *union*, retorne SIM.

Detectar existência de ciclos

- O algoritmo funciona de forma similar ao anterior.
- Dado um DSU com os vértices do grafo, itere por todas as arestas.
  - Se em alguma aresta os vértices já pertencem à mesma componente conexa antes do *union*, retorne SIM.
  - Ao fim da iteração por todas as arestas, retorne NÃO.

Detectar existência de ciclos

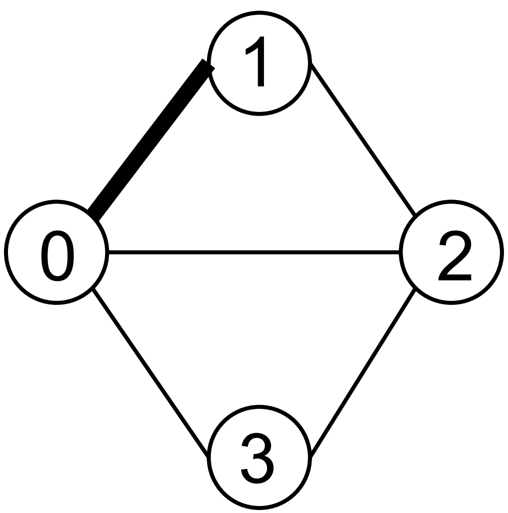
Considere o exemplo abaixo:



0	1	2	3
0	1	2	3

Detectar existência de ciclos

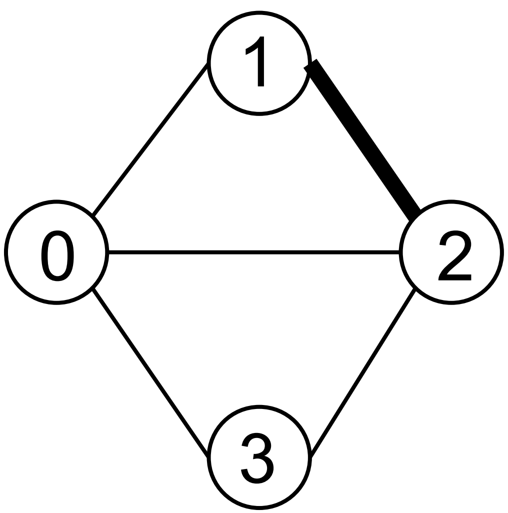
Ao analisarmos uma aresta cujos vértices estão em componentes conexos diferentes, apenas fazemos a união.



0	1	2	3
0	1	2	3

Detectar existência de ciclos

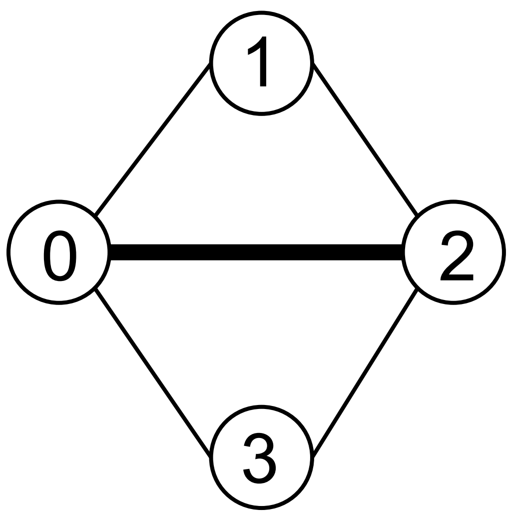
Ao analisarmos uma aresta cujos vértices estão em componentes conexos diferentes, apenas fazemos a união.



0	1	2	3
0	0	2	3

Detectar existência de ciclos

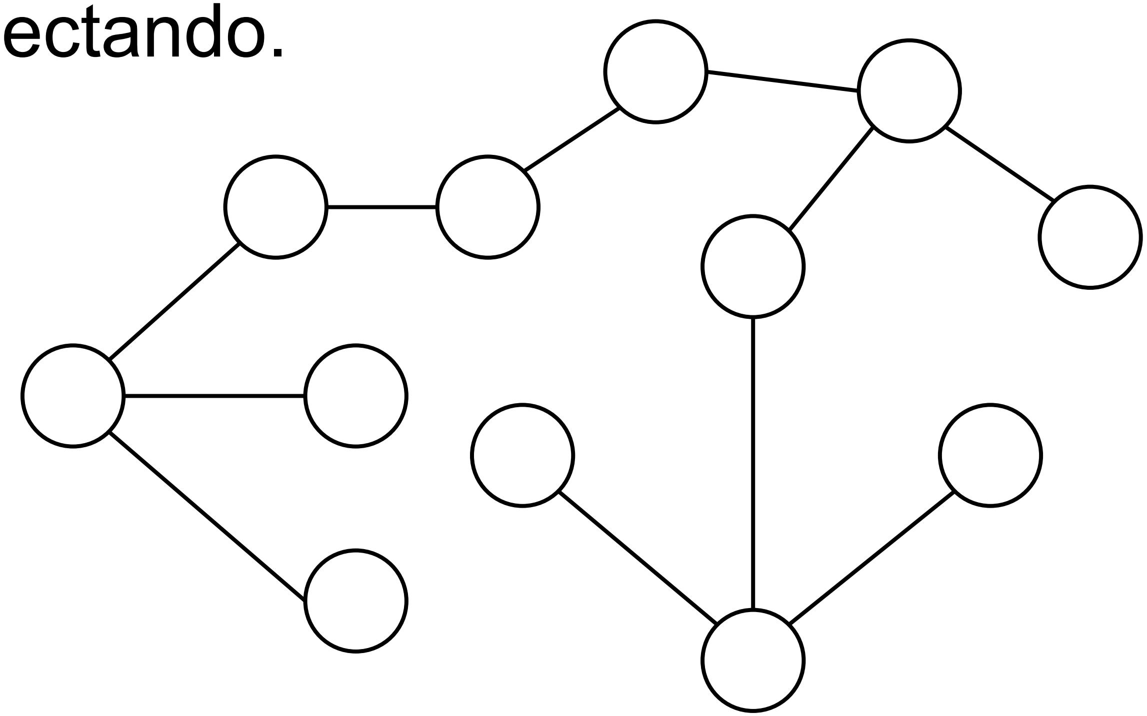
Note que quando escolhemos uma aresta cujos vértices já pertencem a uma mesma componente conexa sabemos por definição que existe um caminho que os conecta, logo, encontramos um ciclo.



0	1	2	3
0	0	0	3

Detectar existência de ciclos

Note que se um grafo não possui ciclos, então a execução do *union* sempre será em vértices de componentes conexos diferentes, afinal para cada par de vértices existe apenas um único caminho os conectando.



Detectar existência de ciclos

```
DeteccaodeCiclos (V,E,DSU)
    Para cada vértice v de V
        DSU[v] = v
    Para cada aresta e(u,v) de E
        Se Find(u) != Find(v)
            Union(u,v)
        Senão
            Retorna True
    Retorna False
```

Detectar existência de ciclos

Qual a complexidade do algoritmo?

Detectar existência de ciclos

Qual a complexidade do algoritmo?

- No caso de não haver ciclos, temos que iterar por todas arestas.

Detectar existência de ciclos

Qual a complexidade do algoritmo?

- No caso de não haver ciclos, temos que iterar por todas arestas.
- Considerando DSU implementado em árvores, com união por *rank* e *path compression* a complexidade é $O(|E|\alpha(|V|))$.

Detectar existência de ciclos

Qual a complexidade do algoritmo?

- No caso de não haver ciclos, temos que iterar por todas arestas.
- Considerando DSU implementado em árvores, com união por *rank* e *path compression* a complexidade é $O(|E|\alpha(|V|))$.

Desvantagem:

- O algoritmo detecta apenas se existe um ciclo, não retorna os vértices que o compõem.

Aplicações de DSU em grafos

- Identificar componentes conexas.
- Detecção de ciclos.
- **Árvore geradora mínima.**

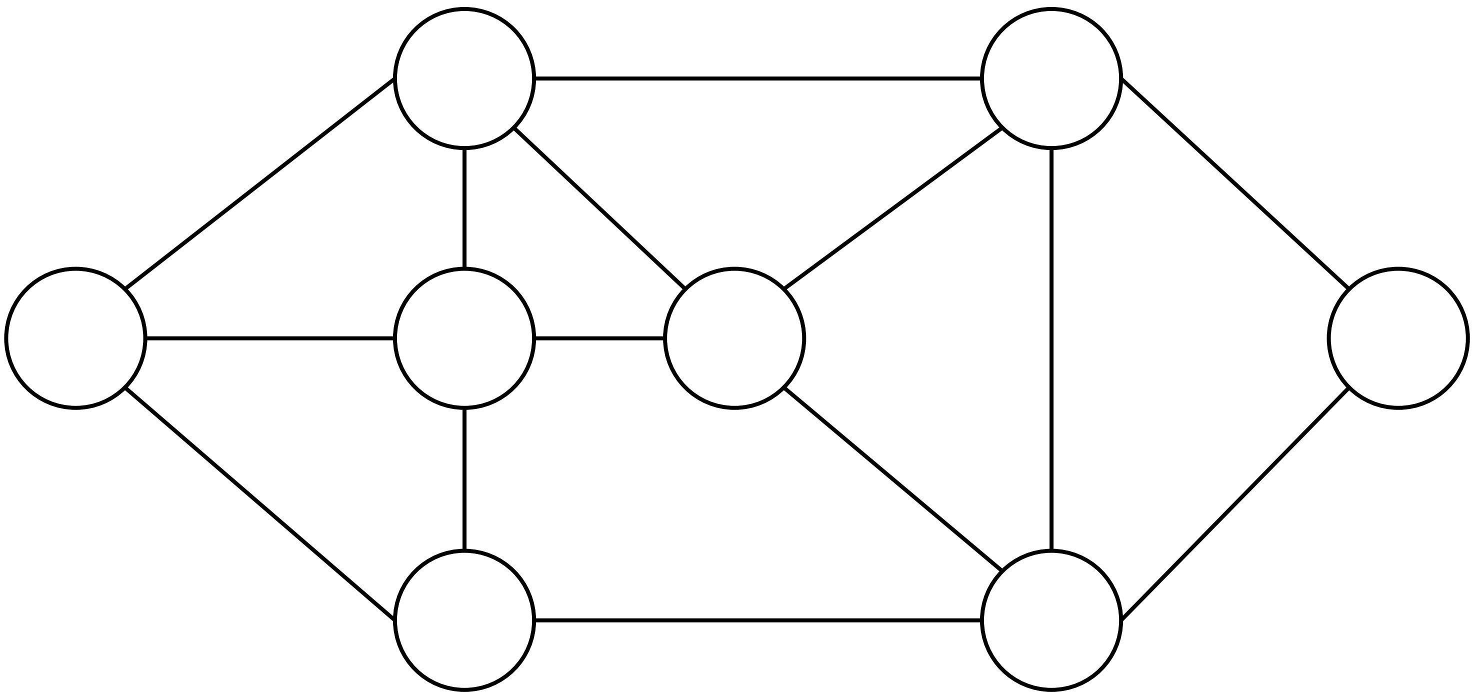
Um probleminha!

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

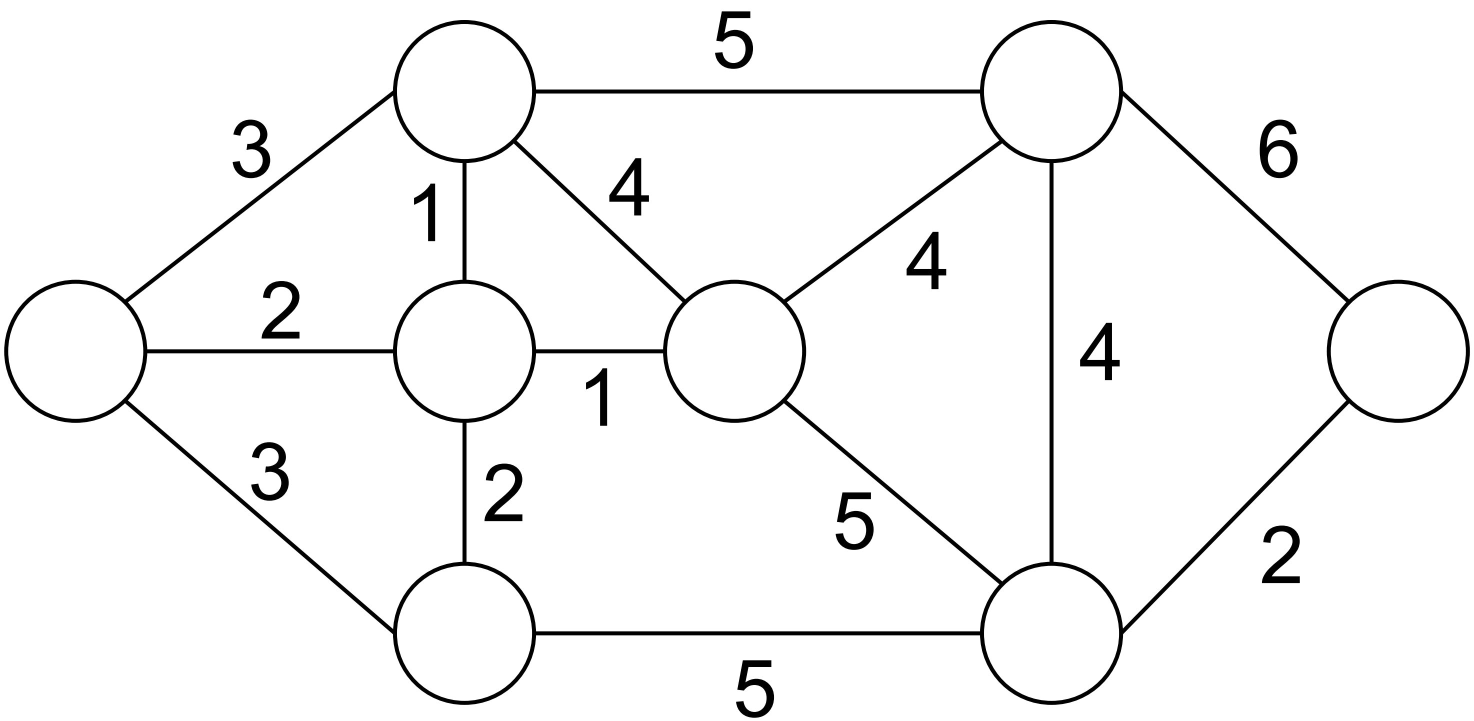
Um probleminha!

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



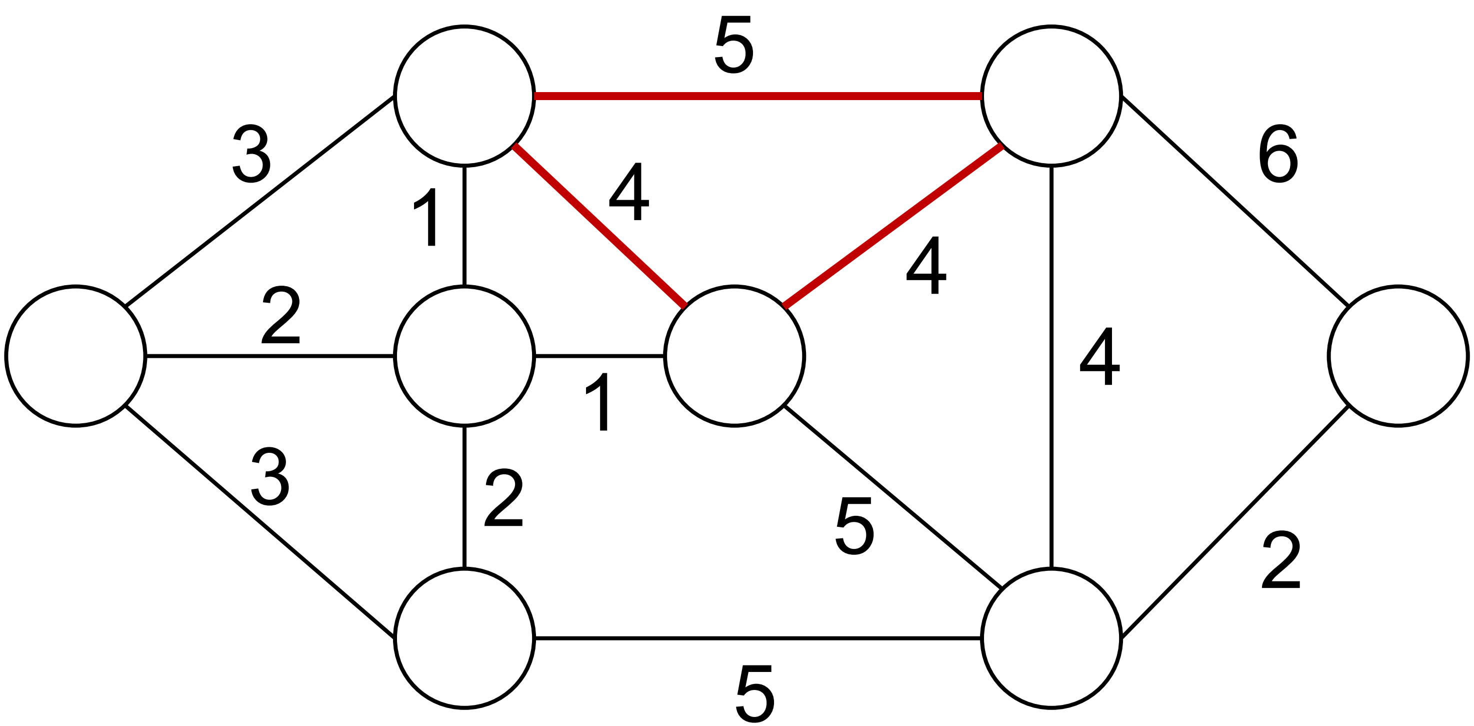
Um probleminha!

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



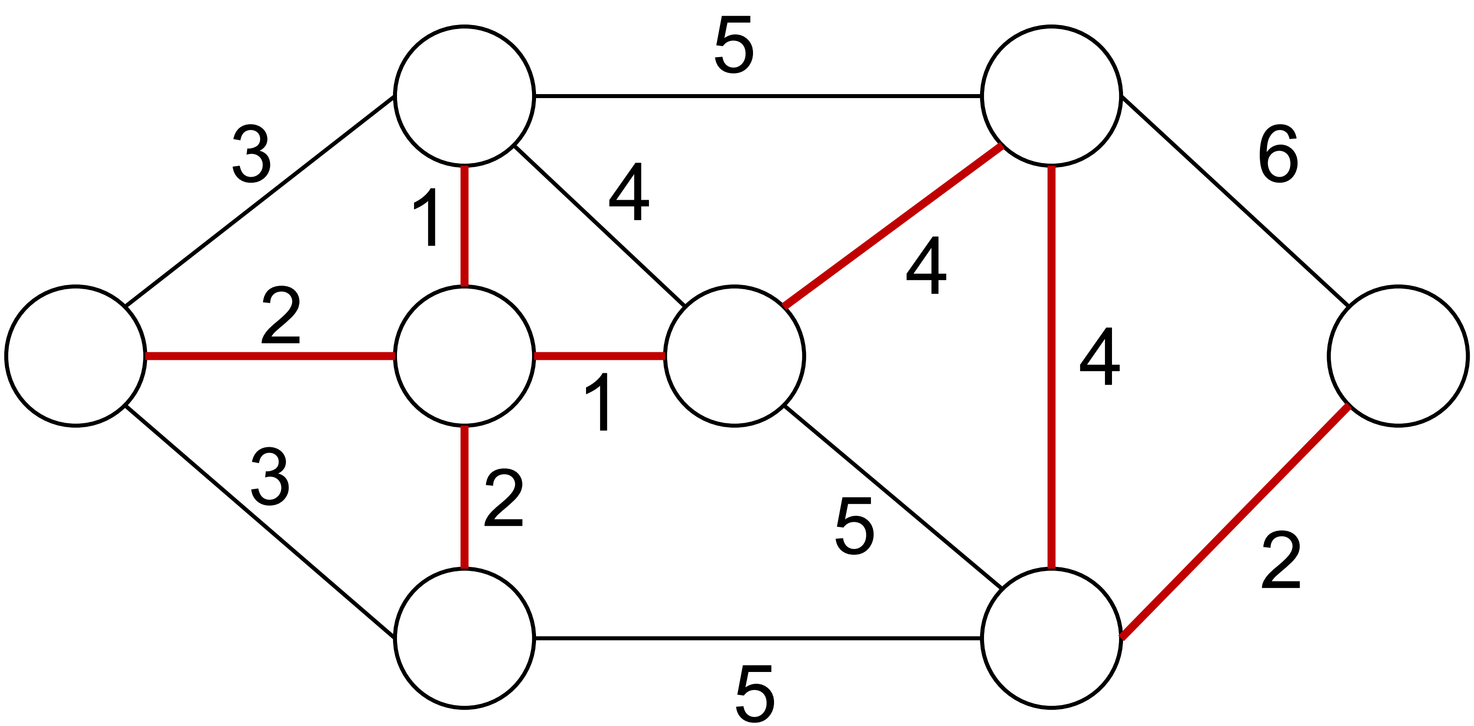
Um probleminha!

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



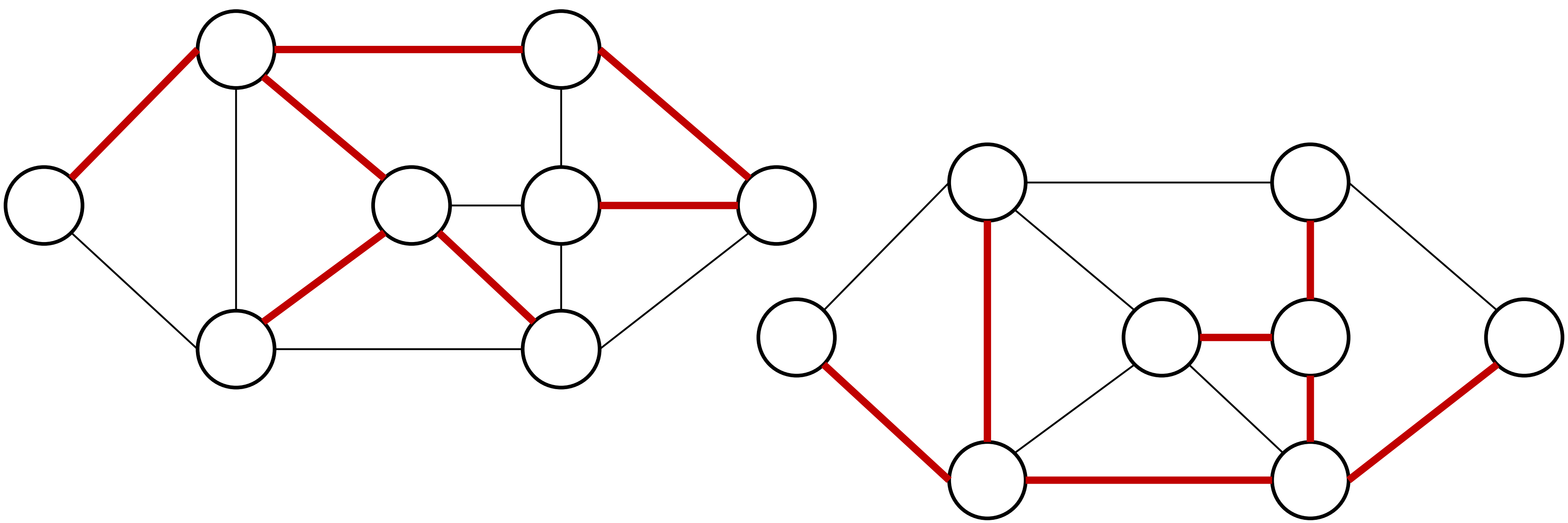
Um probleminha!

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).

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).
- Sabemos utilizar o DSU para detectar ciclos, logo podemos utilizá-lo para encontrar uma árvore geradora.

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).
- Sabemos utilizar o DSU para detectar ciclos, logo podemos utilizá-lo para encontrar uma árvore geradora.
- Então agora basta estabelecer um critério para a ordem que iremos escolher as arestas.

Como encontrar uma árvore geradora?

- Para resolver este problema iremos utilizar uma outra estrutura que já estudamos!

Como encontrar uma árvore geradora?

- Para resolver este problema iremos utilizar uma outra estrutura que já estudamos!
  - *Heap*

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.

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 consiste em:

- 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.

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 consiste em:

- 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.

Como encontrar uma árvore geradora?

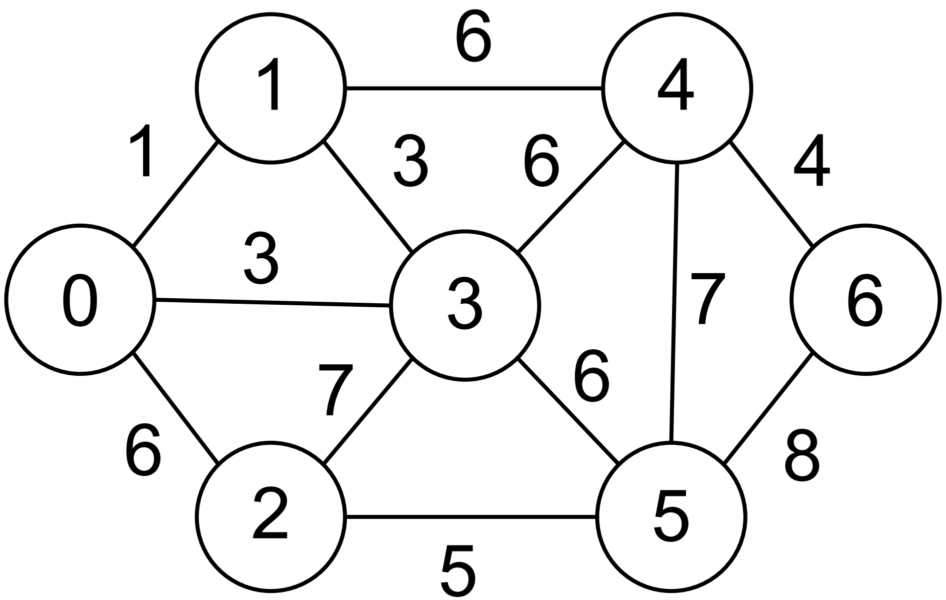
Para auxiliar nosso algoritmo, iremos nos apoiar no seguinte fato:

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

O algoritmo consiste em:

- 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



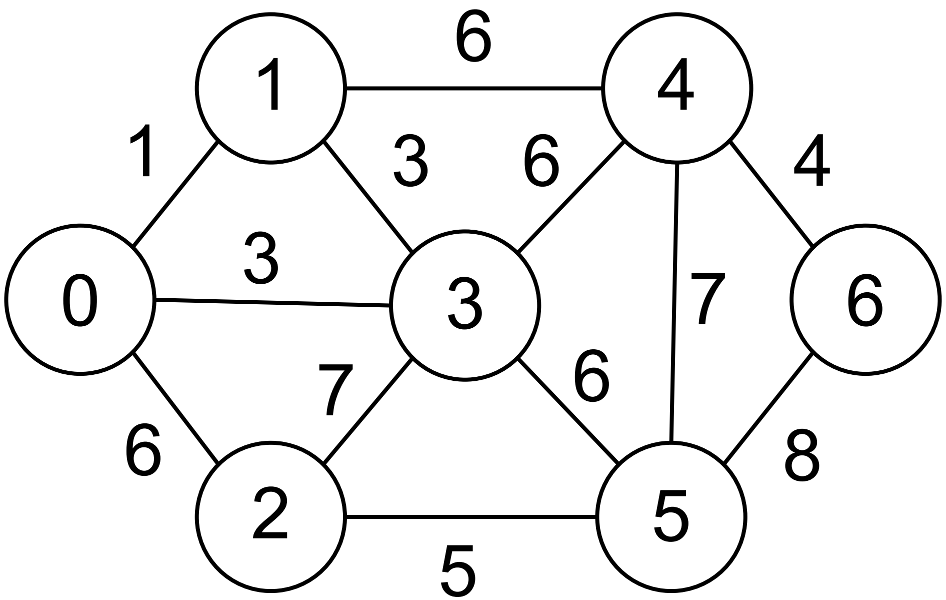
custo = 0
cont = 0

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



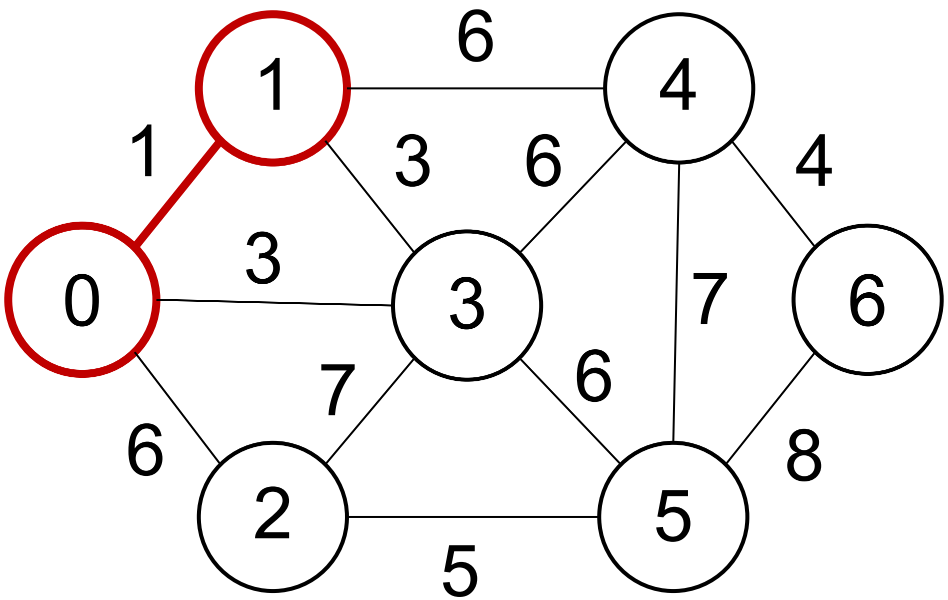
custo = 0
cont = 0

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



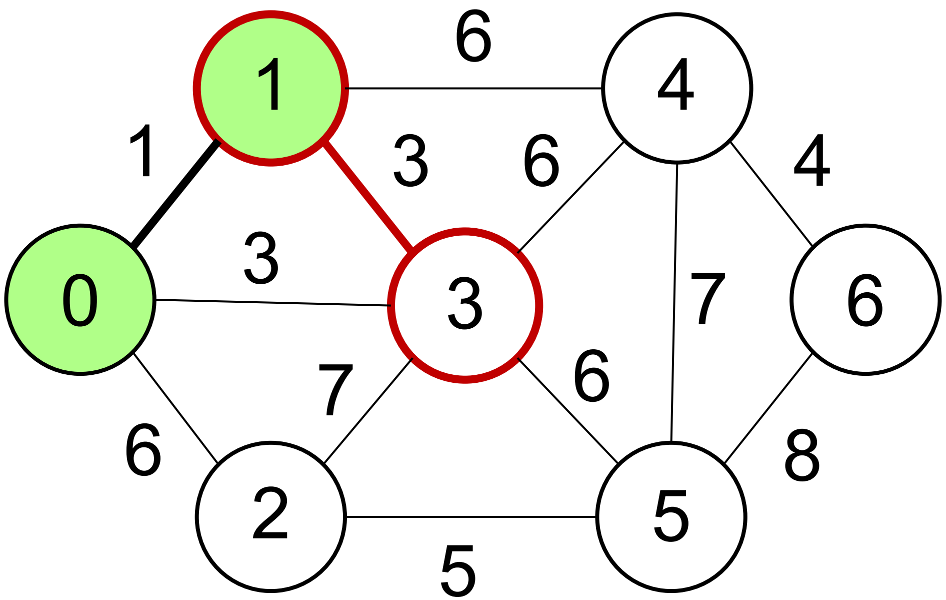
custo = 0
cont = 0

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



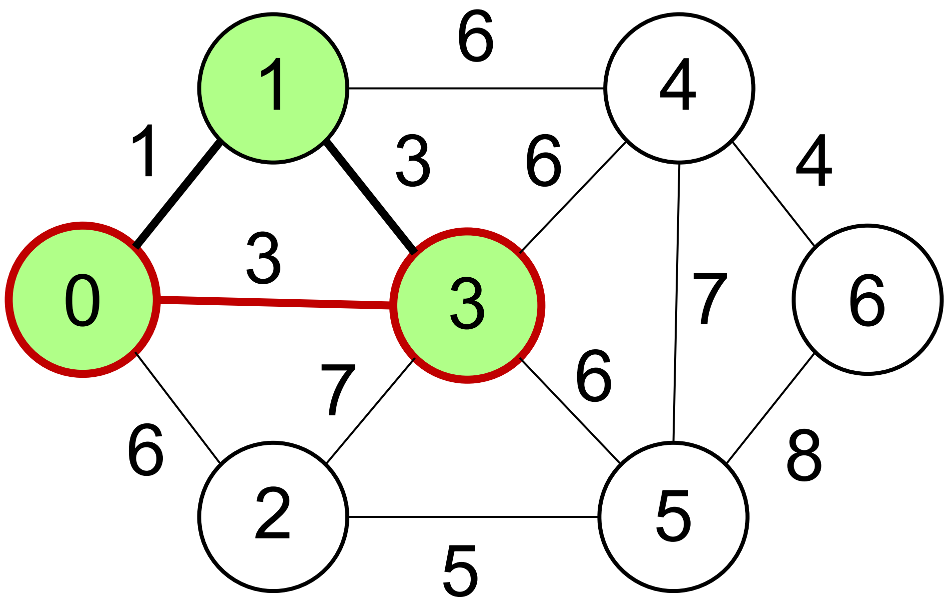
custo = 1
cont = 1

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



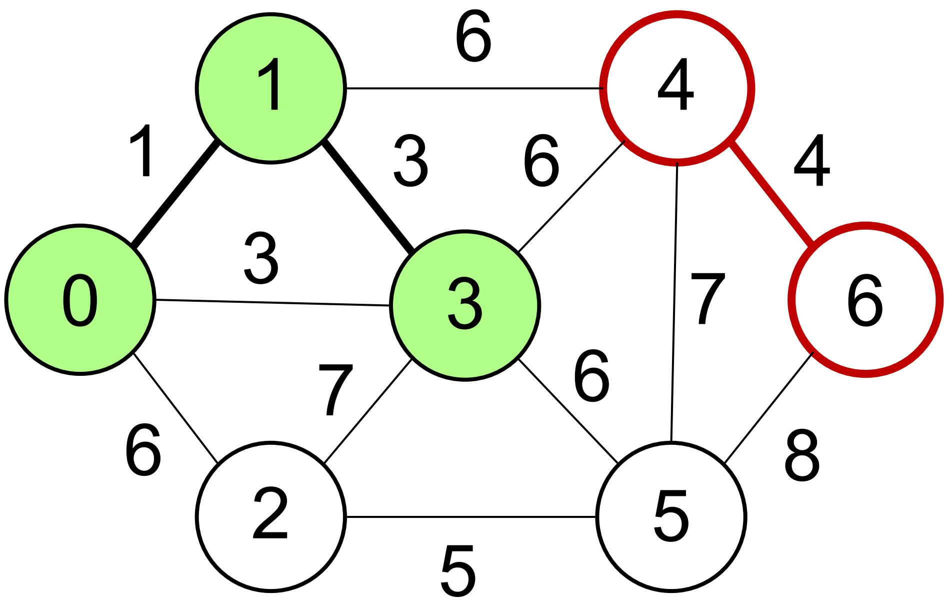
custo = 4
cont = 2

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



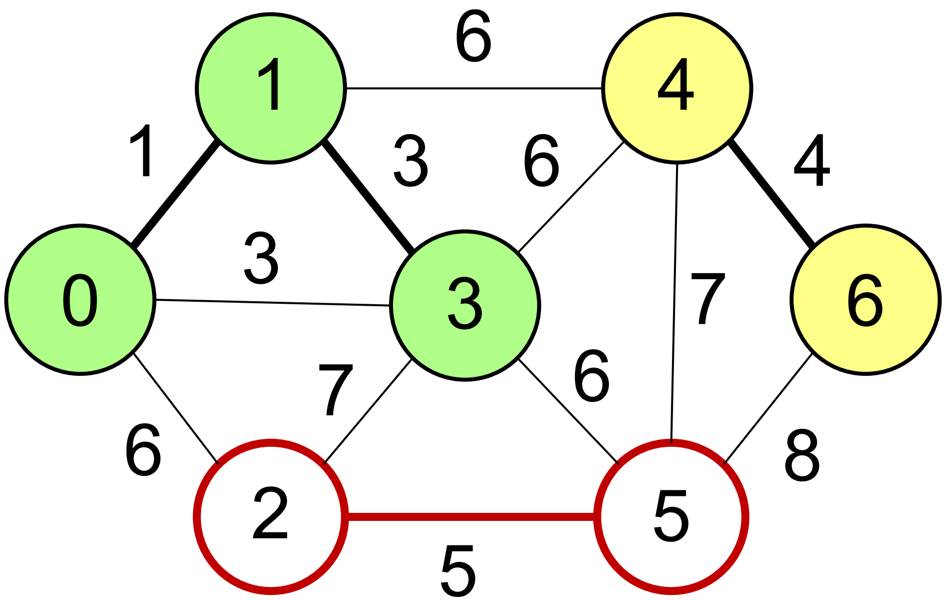
custo = 4
cont = 2

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



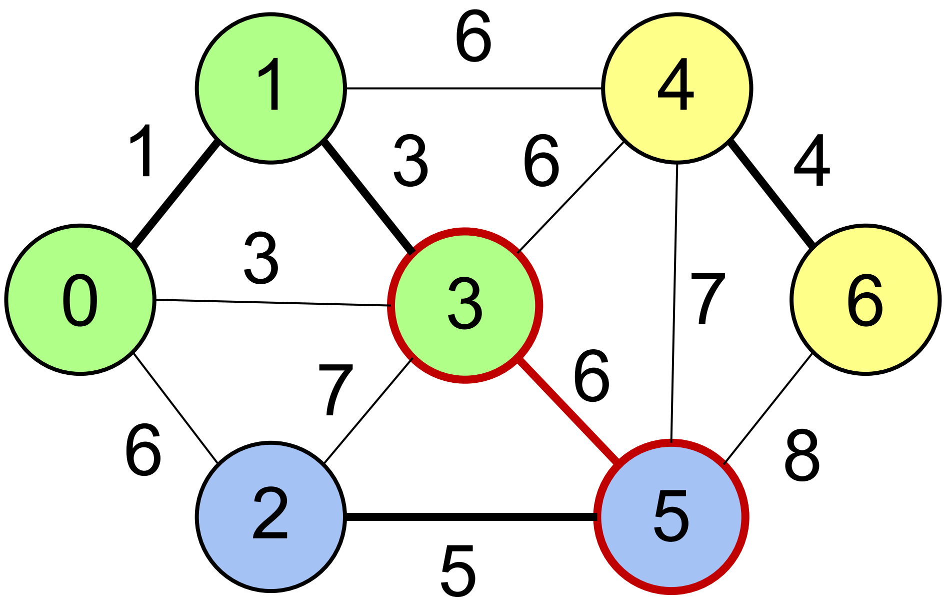
custo = 8
cont = 3

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



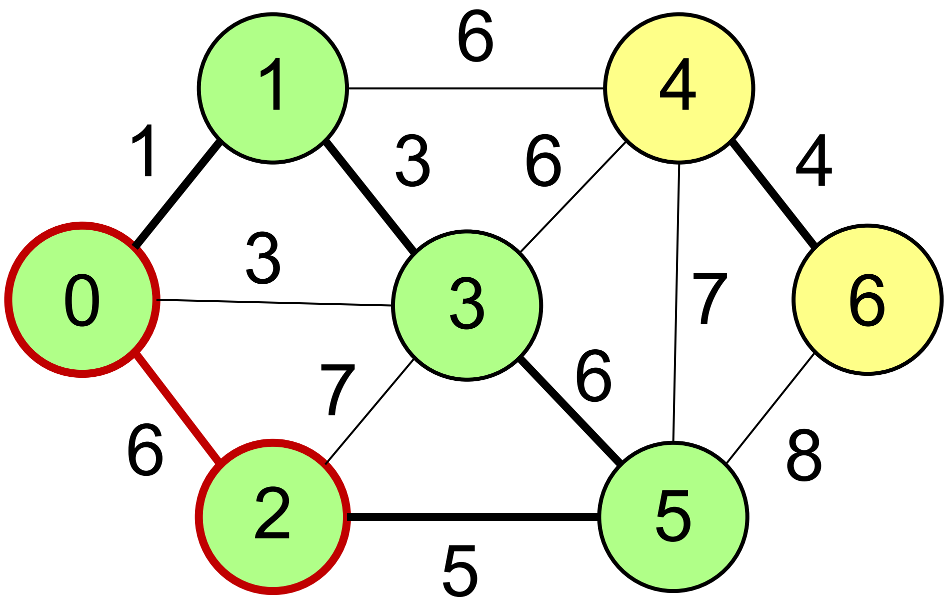
custo = 13
cont = 4

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



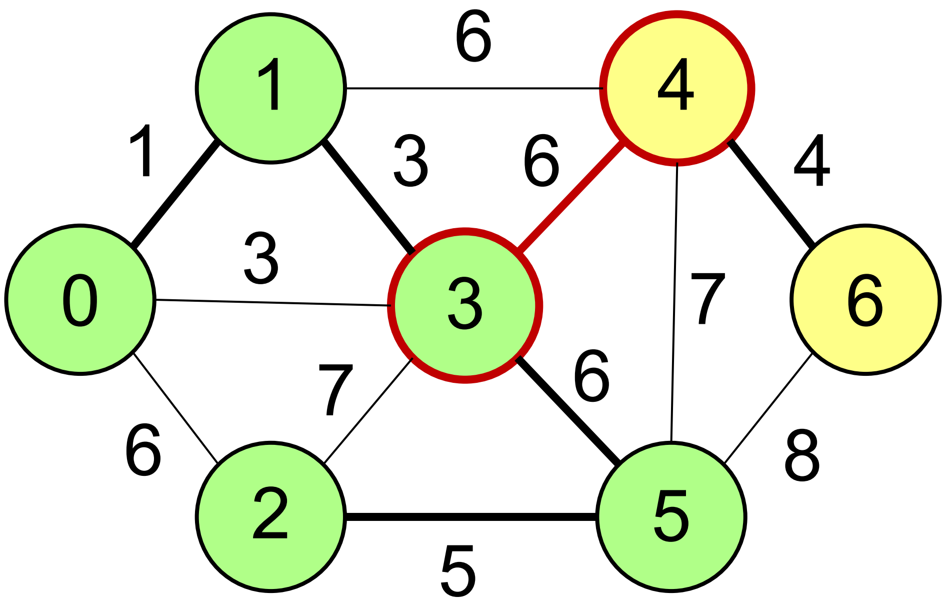
custo = 13
cont = 4

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



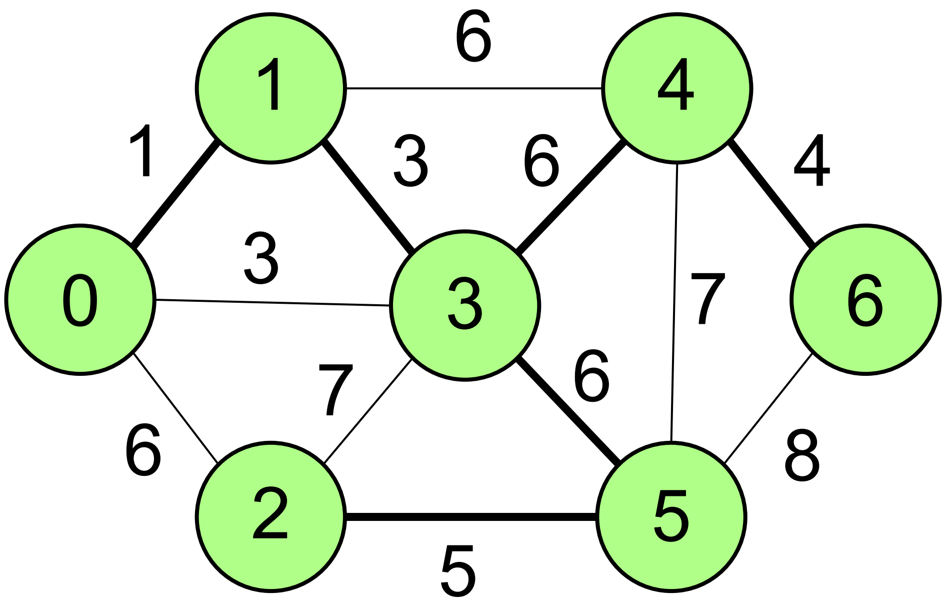
custo = 19
cont = 5

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.

Árvore Geradora Mínima

ÁrvoreGeradoraMínima (*V,E,DSU*)

    Custo=0; *Arestas*=0;

    Para cada vértice *v* de *V*

        DSU[v] = v

    H=Heap(E)

    Para cada aresta *e(u,v)*=Removemin(H)

        Se **Find**(*u*) != **Find**(*v*)

            Union(*u,v*)

            Custo+=*e(u,v)*

            Arestas++

Árvore geradora Mínima

Qual a complexidade do algoritmo?

Árvore geradora Mínima

Qual a complexidade do algoritmo?

- O algoritmo utiliza um *heap* e todas as arestas são inseridas.

Árvore geradora Mínima

Qual a complexidade do algoritmo?

- O algoritmo utiliza um *heap* e todas as arestas são inseridas.
- Se o grafo de entrada for uma árvore, o algoritmo irá parar apenas quando todas as arestas forem removidas e analisadas.

Árvore geradora Mínima

Qual a complexidade do algoritmo?

- O algoritmo utiliza um *heap* e todas as arestas são inseridas.
- Se o grafo de entrada for uma árvore, o algoritmo irá parar apenas quando todas as arestas forem removidas e analisadas.
- Portanto, mesmo implementando o DSU em árvores, com união por *rank* e *path compression* a complexidade é $O(|E|\log(|E|))$.