

# Árvore Geradora Mínima

## Teoria dos Grafos

Prof. Daniel Saad Nogueira Nunes



**INSTITUTO  
FEDERAL**

Brasília

---

Campus  
Taguatinga

# Sumário

Árvore Geradora Mínima (MST)

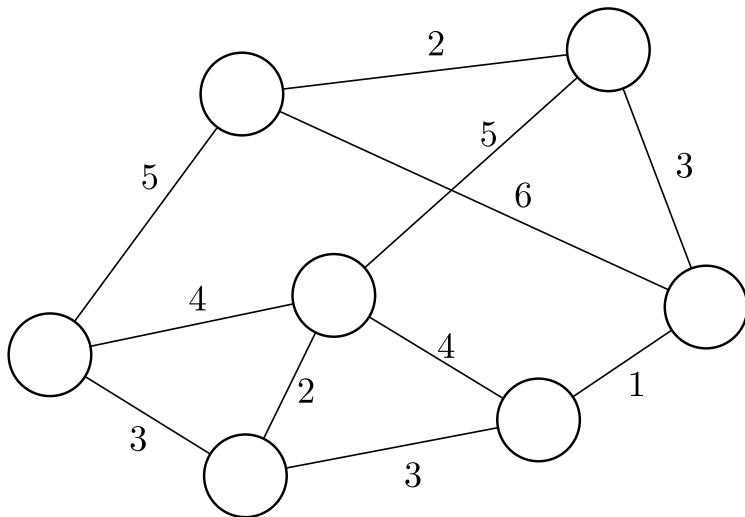
Prim

Kruskal

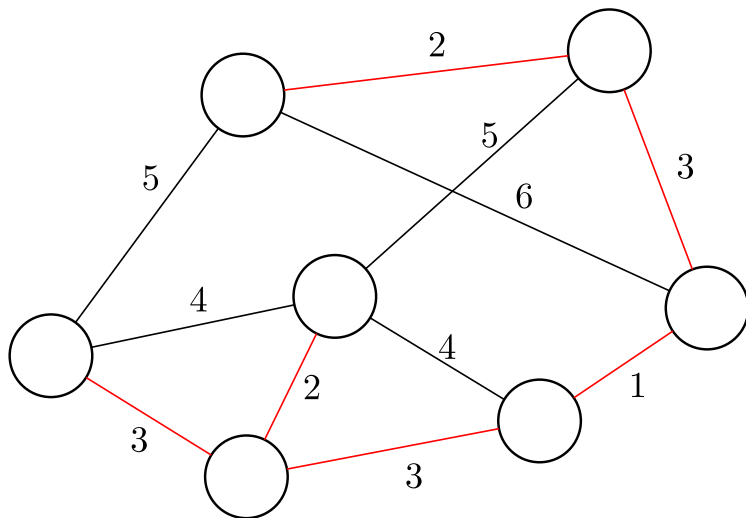
# Motivação

- ▶ Suponha que tenhamos uma infraestrutura de rede montada.
- ▶ Máquinas estão conectadas a outras através de diversos roteadores.
- ▶ Ao mesmo tempo, a economia de energia se tornou uma situação crítica.
- ▶ Como você faria para continuar permitindo a comunicação de quaisquer computadores com menor custo possível?
- ▶ Quais enlaces você desativaria?
- ▶ Qual a estrutura obtida após a remoção dos enlaces redundantes?

# Motivação



# Motivação



# Árvore geradora mínima

- ▶ Problema da **Árvore Geradora Mínima (MST – Minimum Spanning Tree)**
- ▶ Queremos um subgrafo acíclico e conexo de menor custo (árvore de menor custo).
- ▶ Existem algoritmos bem conhecidos e eficientes para resolução deste problema.
- ▶ No entanto, vamos examinar algumas definições antes de atacar o problema.

# Árvore geradora mínima

## Relembrando...

- ▶ Um subgrafo  $G' = (V', E)$  de um grafo  $G = (V, E)$  é dito **subgrafo gerador** quando  $V' = V$  e  $E' \subseteq E$ .
- ▶ Se  $G'$  é uma árvore,  $G'$  é dito uma árvore geradora.

# Árvore geradora mínima

## Definição (Custo de uma árvore)

*Seja  $T = (V, E)$  uma árvore e  $w : E \rightarrow \mathbb{R}$  uma função de custo que associa um peso a cada aresta. O custo de uma árvore, denotada por  $w(T)$ , é o somatório dos custos das arestas. Em outras palavras:*

$$w(T) = \sum_{(u,v) \in E} w((u,v))$$

# Árvore geradora mínima

## Definição (Árvore geradora mínima)

*Seja  $G = (V, E)$  um grafo e  $w : E \rightarrow \mathbb{R}$  uma função de custo que associa um peso a cada aresta.*

*A árvore geradora mínima (MST) de  $G$  é uma árvore geradora  $T = (V, E')$  de  $G$  com  $w(T)$  mínimo.*

# Árvore geradora mínima

## Unicidade da solução

A MST de um grafo não é única. Pode haver múltiplas árvores geradoras com custo mínimo.

# Árvore geradora mínima

Uma estratégia genérica para obter a MST é a seguinte.

- ▶ Inicialmente todo vértice é uma componente conexa isolada.
- ▶ Inclua arestas de menor custo possível de modo a unir duas componentes conexas distintas.
- ▶ Jamais inclua uma aresta entre dois vértices da mesma componente conexa.
- ▶ Jamais forme um ciclo!

# Árvore geradora mínima

---

**Algorithm 1:** GENERIC-MST( $(G)$ )

---

```
1  $T \leftarrow (V, \emptyset)$ 
2 while  $T$  não for uma árvore do
3   | Encontre uma aresta  $(u, v)$  segura para  $T$ 
4   | Adicione a aresta  $(u, v)$  a  $T$ 
```

---

# Árvore geradora mínima

- ▶ O que é uma aresta segura?
- ▶ Como selecionar uma aresta segura?
- ▶ Depende do algoritmo. . .

# Sumário

Árvore Geradora Mínima (MST)

Prim

Kruskal

# Algoritmo de Prim

- ▶ Começamos de um nó arbitrário  $s$  como o único nó de nossa árvore.
- ▶ Escolhemos sempre as arestas de menor custo para adicionarmos à árvore a partir dos nós previamente inseridos na árvore.
- ▶ Seja  $d[u]$  o custo para adicionar um vértice  $u$  a árvore. A cada passo, o algoritmo escolhe o vértice  $v$  com menor  $d[v]$  possível.
- ▶ Precisamos de uma estrutura eficiente para escolher o vértice com menor custo de adição!
- ▶ **Fila de prioridade.**

# Algoritmo de Prim

---

**Algorithm 2:** INITIALIZE-PRIM( $G, s$ )

---

```
1 forall  $u \in V$  do  
2    $d[u] \leftarrow \infty$   
3    $p[u] \leftarrow \perp$   
4  $d[s] \leftarrow 0$ 
```

---

# Algoritmo de Prim

---

**Algorithm 3:** PRIM( $G, s$ )

---

```
1  INITIALIZE-PRIM( $G, s$ )
2   $Q$ .INSERT( $s$ )
3   $T \leftarrow (V, \emptyset)$ 
4  while not  $Q$ .EMPTY() do
5       $u \leftarrow Q$ .EXTRACT-MIN()
6      visited[ $u$ ]  $\leftarrow true$ 
7       $T \leftarrow T \cup (p[u], u)$ 
8      forall  $(u, v)$  and not visited[ $v$ ] do
9          if  $d[v] > w((u, v))$  then
10              $d[v] \leftarrow w((u, v))$ 
11              $p[v] \leftarrow u$ 
12              $Q$ .INSERT-UPDATE( $v$ )
13 return  $T$ 
```

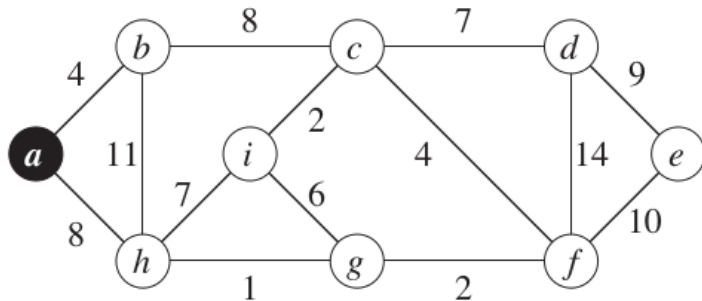
---

# Algoritmo de Prim

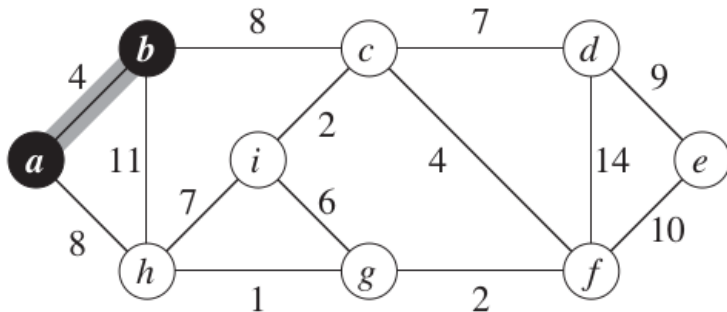
Aqui assumimos que

- ▶  $Q.\text{INSERT}(u)$  insere um nó  $u$  com custo  $d[u]$  na fila de prioridades.
- ▶  $Q.\text{EXTRACT-MIN}()$  remove o nó  $u$  com menor  $d[u]$  da fila de prioridades.
- ▶  $Q.\text{INSERT-UPDATE}(u)$  insere um nó  $u$  com custo  $d[u]$  na fila de prioridades se  $u$  não está nela, ou, atualiza a prioridade do nó  $u$  para  $d[u]$  na fila de prioridades.

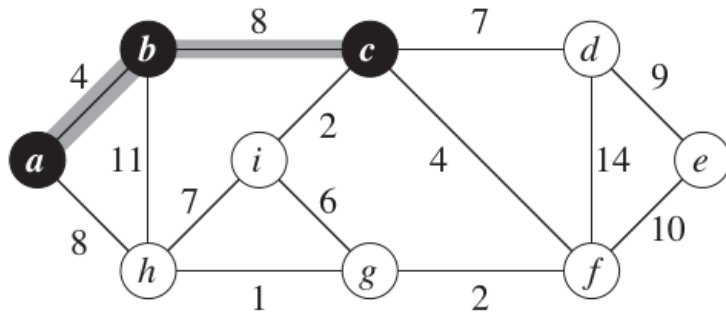
# Algoritmo de Prim



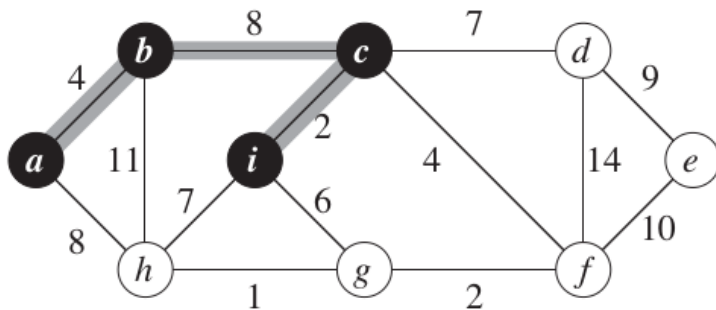
# Algoritmo de Prim



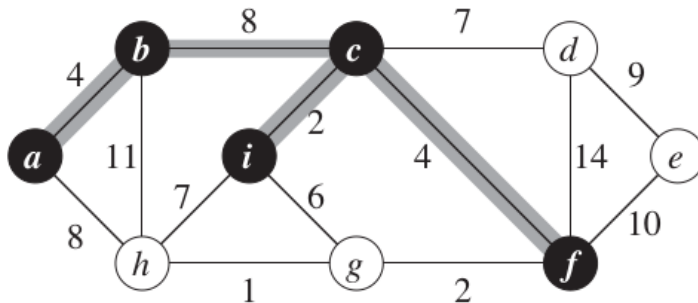
# Algoritmo de Prim



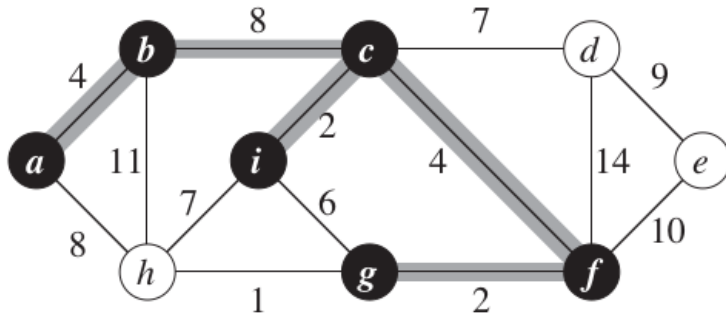
# Algoritmo de Prim



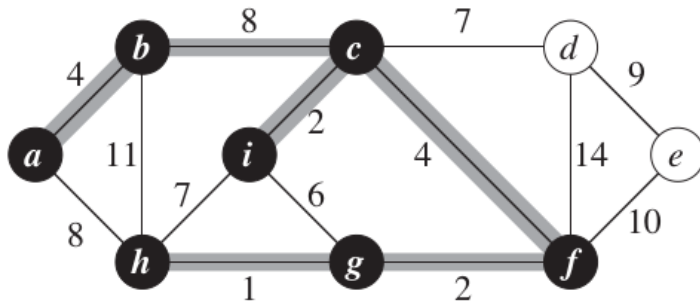
# Algoritmo de Prim



# Algoritmo de Prim



# Algoritmo de Prim



# Análise

## Análise de pior-caso

Fazemos EXTRACT-MIN  $n$  vezes e INSERT-UPDATE no máximo  $m$  vezes. Logo o custo total do algoritmo é  $\Theta(n \cdot t_e + m \cdot t_u)$  sendo  $t_e$  e  $t_u$  os custos das operações de EXTRACT-MIN e INSERT-UPDATE, respectivamente.

Em uma heap binária,  $t_e = \Theta(\lg n)$  e  $t_u = \Theta(\lg(n))$ , logo, o custo total seria  $\Theta(n \lg n + m \lg n) = \Theta(m \lg n)$

Usando uma heap de Fibonacci,  $t_e = \Theta(\lg n)$  e  $t_u = \Theta(1)$ , logo o custo seria:  $\Theta(m + n \lg n)$ .

# Sumário

Árvore Geradora Mínima (MST)

Prim

Kruskal

# Algoritmo de Kruskal

- ▶ O algoritmo de Kruskal tem uma filosofia bem simples.
- ▶ Ordenam-se as arestas em ordem crescente de custo.
- ▶ Coloca-se cada nó em uma componente conexa isolada.
- ▶ Para cada aresta  $(u, v)$  em ordem crescente, acrescenta-se a aresta  $(u, v)$  bem como os vértices  $u$  e  $v$  na árvore espalhada mínima somente se  $u$  e  $v$  não estão na mesma componente conexa.
- ▶ Isso evita a formação de ciclos.

# Algoritmo de Kruskal

Para o algoritmo de Kruskal funcionar, precisamos de uma estrutura de dados que permita:

- ▶ Verificar se dois vértices  $u$  e  $v$  estão na mesma componente conexa.
- ▶ Unir as componentes conexas de  $u$  e  $v$ .
- ▶ Como as componentes conexas são conjuntos disjuntos, podemos utilizar uma estrutura de dados para conjuntos disjuntos (DSU – Disjoin Set Union Data Structure, ou Union-Find).

# Sumário

Kruskal

DSU

O algoritmo

# DSU

## Estruturas de Dados Para Conjuntos Disjuntos

- ▶ Suponha  $\mathcal{S} = (\{s_0\}, \{s_1\}, \dots, \{s_n\})$  conjuntos *singleton*.
- ▶ Deseja-se implementar duas operações básicas:
  - ▶  $\text{UNION}(A, B)$ , une dois conjuntos  $A$  e  $B$ .
  - ▶  $\text{FIND}(x)$ , diz em qual conjunto encontra-se o elemento  $x$ .
- ▶ É interessante observar que, após um número qualquer de sucessivas aplicações de  $\text{UNION}$ , o que se tem é um único conjunto em  $\mathcal{S}$  contendo todos os elementos, ou conjuntos disjuntos.

# DSU

- ▶ Para simplificar, para cada conjunto, vamos eleger um representante.
- ▶ Assim, dois elementos estão na mesma coleção se e somente se eles possuem o mesmo representante.
- ▶ Inicialmente, cada *singleton* tem como representante ele mesmo.
- ▶ Desta forma, a operação de  $\text{FIND}(x)$  pode simplesmente retornar o representante do conjunto no qual  $x$  está incluído.
- ▶ Assim, dados dois elementos,  $x$  e  $y$ , conseguimos saber se eles estão no mesmo conjunto verificando se  $\text{FIND}(x) = \text{FIND}(y)$ .

# DSU

- ▶ Podemos modelar isso computacionalmente através de uma floresta.
- ▶ Cada *singleton*  $x$  inicialmente faz parte de uma árvore e possui dois valores:
  - ▶  $p[x]$ : diz quem é o pai de  $x$  (inicialmente é ele mesmo).
  - ▶  $r[x]$  diz o rank de  $x$ , que é uma cota superior da altura da árvore com raiz em  $x$ .

# Algoritmo de Kruskal

## Estruturas de Dados Para Conjuntos Disjuntos

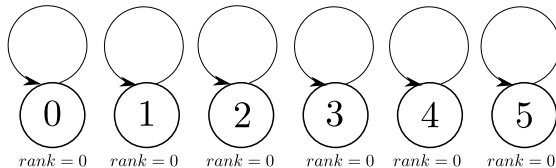


Figura: Estado inicial da DSU.

# DSU

- ▶ Para achar o representante do conjunto que  $x$  está incluso, basta seguir os ponteiros de pai até chegar na raiz.
- ▶ Logo, já sabemos implementar FIND.

# Algoritmo de Kruskal

## Estruturas de Dados Para Conjuntos Disjuntos

---

**Algorithm 4:**  $\text{FIND}(x)$ 

---

**Input:**  $x$

**Output:** representante do conjunto de  $x$

```
1 if  $p[x] \neq x$  then  
2   | return  $\text{FIND}(p[x])$   
3 return  $x$ 
```

---

# Algoritmo de Kruskal

## Estruturas de Dados Para Conjuntos Disjuntos

- ▶ Para implementar a operação de  $\text{UNION}(x, y)$ , achamos o representante dos conjuntos que  $x$  e  $y$  se encontram, chame-os de  $x'$  e  $y'$ . Se  $x' \neq y'$ , significa que  $x$  e  $y$  estão em conjuntos diferentes. Logo só é necessário unir as duas árvores fazendo com que  $y'$  vire filho de  $x'$  ou  $x'$  vire filho de  $y'$ .
- ▶ Qual opção escolher?
- ▶ Escolhemos a árvore de maior *rank* para abrigar a de menor *rank*.

# Algoritmo de Kruskal

## Estruturas de Dados Para Conjuntos Disjuntos

---

**Algorithm 5:** UNION( $x, y$ )

---

**Input:**  $x, y$

```
1  $x' \leftarrow \text{FIND}(x)$ 
2  $y' \leftarrow \text{FIND}(y)$ 
3 if  $x' \neq y'$  then
4   if  $r[x'] > r[y']$  then
5      $p[y'] \leftarrow p[x']$ 
6   else
7      $p[x'] \leftarrow y'$ 
8     if  $r[x'] = r[y']$  then
9        $r[y'] \leftarrow r[y'] + 1$ 
10  return true
11 return false
```

---

# Estruturas de Dados para Conjuntos Disjuntos

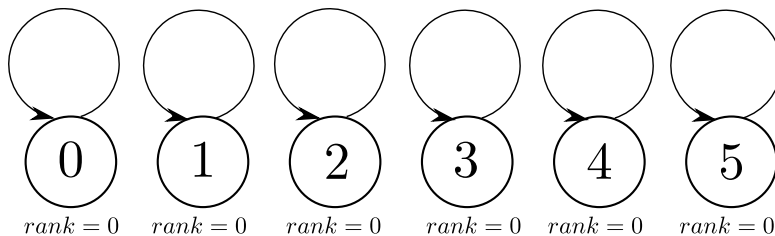


Figura: Estado inicial.

# Estruturas de Dados para Conjuntos Disjuntos

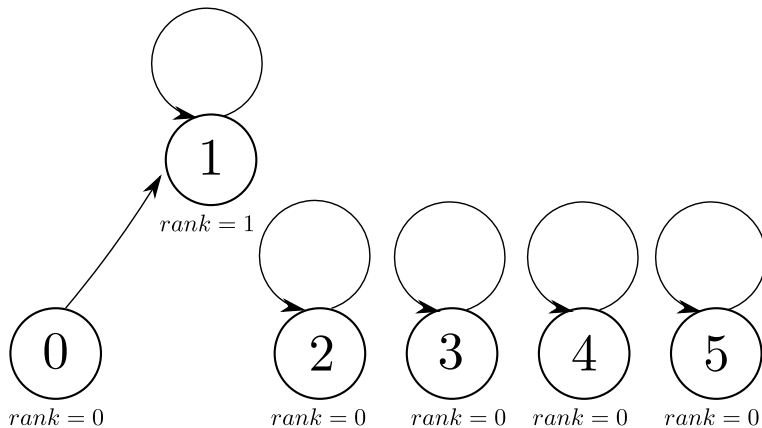


Figura:  $\text{UNION}(0, 1)$

# Estruturas de Dados para Conjuntos Disjuntos

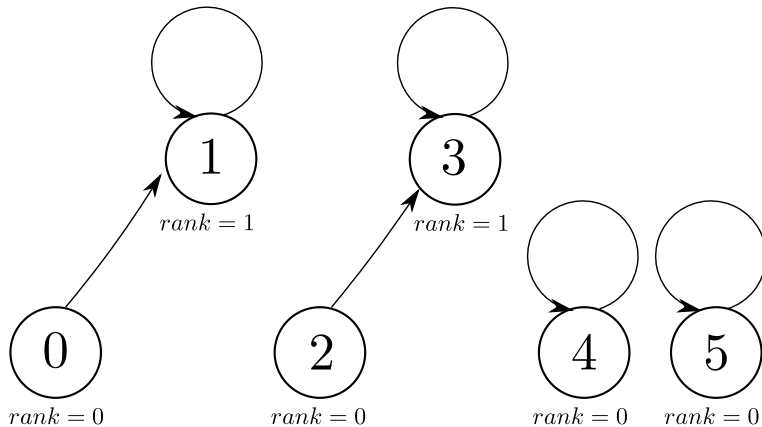


Figura:  $\text{UNION}(2, 3)$

# Estruturas de Dados para Conjuntos Disjuntos

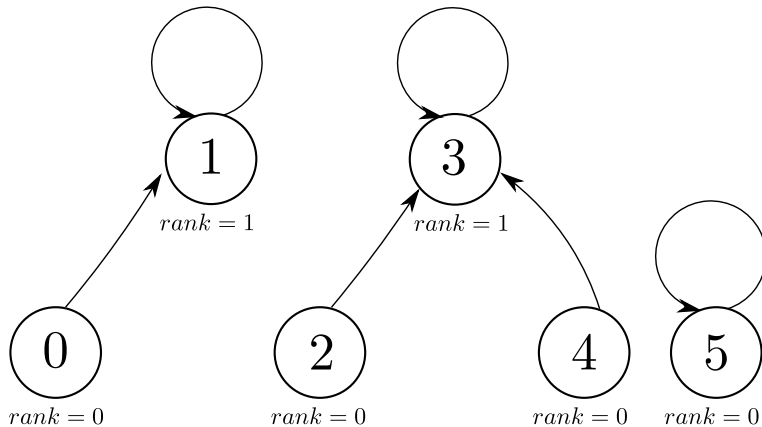


Figura:  $\text{UNION}(2, 4)$

# Estruturas de Dados para Conjuntos Disjuntos

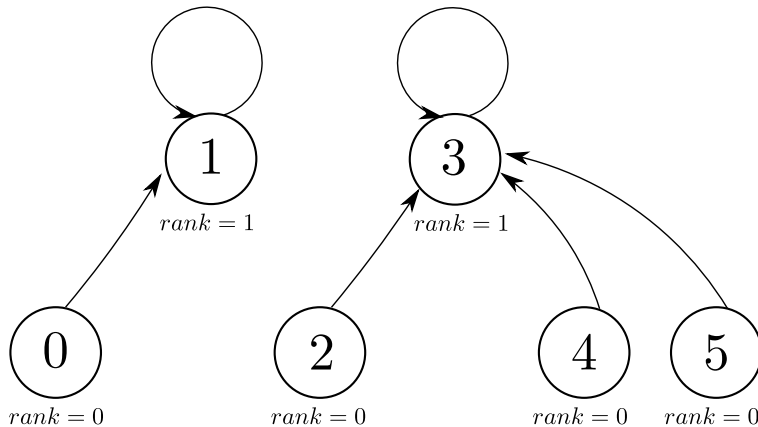


Figura: UNION(2, 5)

# Estruturas de Dados para Conjuntos Disjuntos

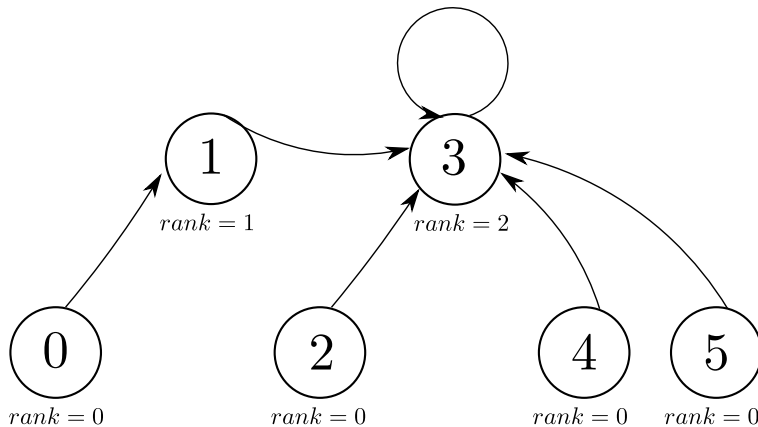


Figura: UNION(0, 3)

# Algoritmo de Kruskal

## Estruturas de Dados Para Conjuntos Disjuntos

- ▶ Da forma como estão implementados `UNION` e `FIND`, a árvore final pode ter altura  $\Theta(n)$ , o que ocasionaria um tempo linear para as consultas de `FIND`.
- ▶ Contudo é possível melhorar esse tempo significativamente ao utilizar uma técnica chamada de **path-compression**.
- ▶ Ao realizar a consulta de `FIND( $x$ )`, atualizamos  $x$  e os demais nós no caminho até a raiz para apontarem imediatamente para o representante do conjunto, isto é, a raiz.

# Algoritmo de Kruskal

## Estruturas de Dados Para Conjuntos Disjuntos

- ▶ Dadas  $M$  operações de UNION ou FIND complexidade final amortizada utilizando a técnica de **path-compression** é:

$$\Theta(M\alpha(n)) \subsetneq \Theta(M \lg^* n) \subsetneq \Theta(M \lg n)$$

em que  $\alpha$  é a função de Ackermann inversa.

- ▶ É uma função que cresce **muito** devagar. Para fins práticos, é praticamente constante.

## DSU

---

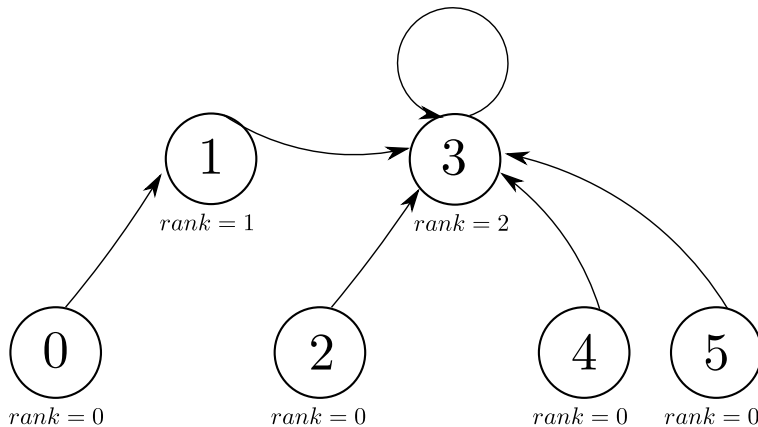
**Algorithm 6:**  $\text{FIND}(x)$  com path-compression

---

**Input:**  $x$ **Output:** representante do conjunto de  $x$ 

- 1 **if**  $p[x] \neq x$  **then**
  - 2      $p[x] \leftarrow \text{FIND}(p[x])$
  - 3 **return**  $x$
-

# Estruturas de Dados para Conjuntos Disjuntos



# Estruturas de Dados para Conjuntos Disjuntos

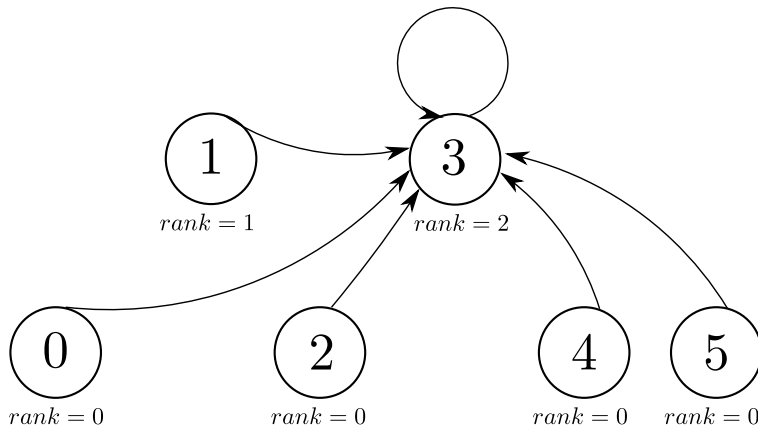


Figura: FIND(0)

# Algoritmo de Kruskal

## Estruturas de Dados Para Conjuntos Disjuntos

- ▶ Todas as operações que sucedem uma compressão de caminho se beneficiarão do tamanho reduzido da altura.
- ▶ Note que, após a compressão de caminho, o *rank* da raiz não corresponde mais a altura da árvore, então *rank* deve ser sempre interpretado como uma cota superior para a altura.

# Sumário

Kruskal

DSU

O algoritmo

# Algoritmo de Kruskal

- ▶ Suponha agora a existência de uma estrutura de dados UF que provê as operações de  $\text{UNION}(x, y)$ ,  $\text{FIND}(x)$  como discutido anteriormente.
- ▶ O algoritmo de Kruskal inicializa os vértices, numerados de 0 a  $n - 1$  em  $n$  singletons.
- ▶ Esta inicialização pode ser feita facilmente por uma função  $\text{MAKE-SET}$ .

# Algoritmo de Kruskal

- ▶ Em seguida, as arestas são ordenadas em ordem crescente do peso.
- ▶ Por fim, para cada aresta  $(u, v)$  na ordem dada, pegamos ela para a árvore espalhada desde que  $\text{UF.UNION}(u, v)$  seja verdadeiro, isto é, desde que  $u$  e  $v$  estejam em componentes conexas distintas. Após a união,  $u$  e  $v$  estarão na mesma componente conexa.

# Algoritmo de Kruskal

---

**Algorithm 7:** KRUSKAL( $G$ )

---

**Input:**  $G$

**Output:**  $T$ , a árvore espalhada mínima

```
1   $UF.MAKESET(n)$ 
2   $T \leftarrow (V, \emptyset)$ 
3   $SORT(G.E)$  // ordena-se as arestas pelo custo
4  forall  $(u, v) \in E$  do
5    if  $UF.UNION(u, v)$  then
6       $T.E \leftarrow T.E \cup (u, v)$ 
7  return  $T$ 
```

---

# Algoritmo de Kruskal

## Análise de pior-caso

- ▶ Tempo de ordenação  $\Theta(m \lg m)$ .
- ▶ Custo das operações de união:  $\Theta(m\alpha(n))$ .
- ▶ Tempo total:  $\Theta(m \lg m + m\alpha(n)) = \Theta(m \lg m) = \Theta(m \lg n)$ .
- ▶ Se as arestas tiverem custo inteiro e pequeno em relação ao número de vértices e arestas, o tempo é dominado pelas  $m$  uniões, uma vez que podemos usar um método pseudolinear de ordenação:  $\Theta(m + m\alpha(n)) = \Theta(m\alpha(n))$ .