Projeto e Análise de Algoritmos

Aula 09 – Árvore Geradora Mínima

Edirlei Soares de Lima
<edirlei@iprj.uerj.br>

Árvore Geradora Mínima

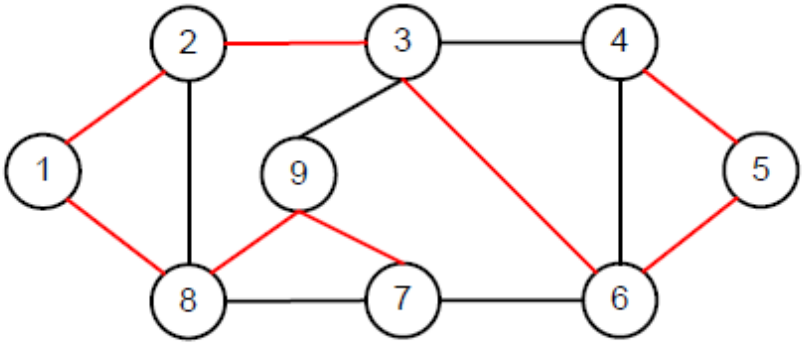
- Dado um grafo não direcionado conectado *G*, uma árvore *T* é chamada de **árvore geradora** de *G* se *T* é um subgrafo de *G* que possui todos os vértices de *G*.

- Uma **árvore geradora mínima** é uma árvore geradora com peso menor ou igual a cada uma das outras árvores geradoras possíveis.
  - Também conhecida como árvore de extensão mínima ou árvore de extensão de peso mínimo;

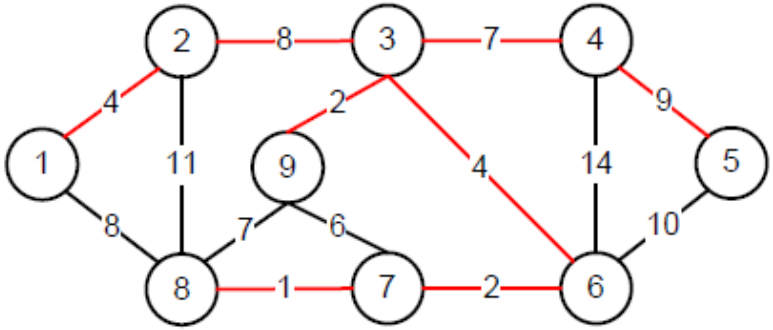
- **Aplicações:**
  - Projeto de redes de telecomunicação;
  - Projeto de rodovias, ferrovias, etc;
  - Projeto de redes de transmissão de energia;

Árvore Geradora Mínima

- **Exemplo:**



Uma árvore geradora.



Uma árvore geradora mínima.

Árvore Geradora Mínima

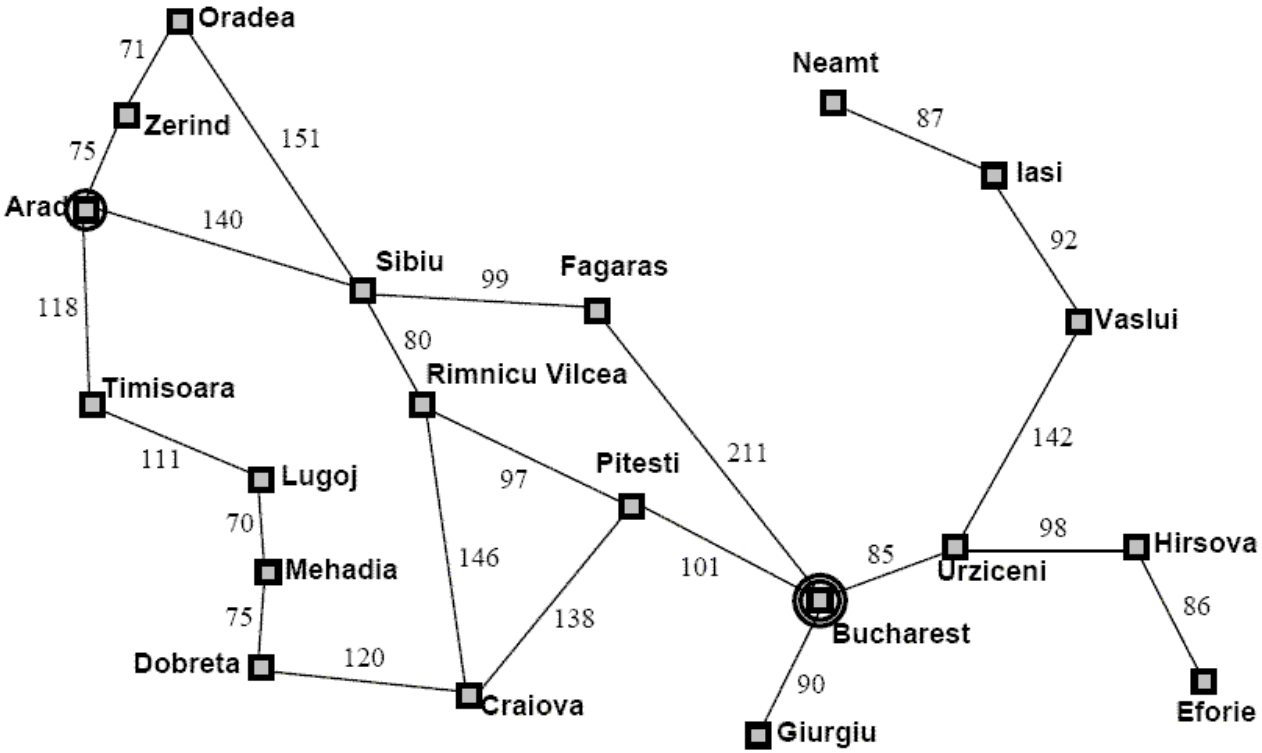
- Existem dois algoritmos clássicos para resolver o problema da árvore geradora mínima:
  - Algoritmo de Prim;
  - Algoritmo de Kruskal;

- Ambos são considerados **algoritmos gulosos**:
  - A estratégia gulosa defende que a menor escolha a cada passo deve ser feita, mesmo que tal escolha não nos leve a uma solução ótima ao final da execução.

- Diferente de alguns algoritmos gulosos, os algoritmos de árvore geradora mínima **sempre encontram a solução ótima**.

Algoritmo Guloso – Caminho Mínimo

- Como ir de Arad a Bucharest?

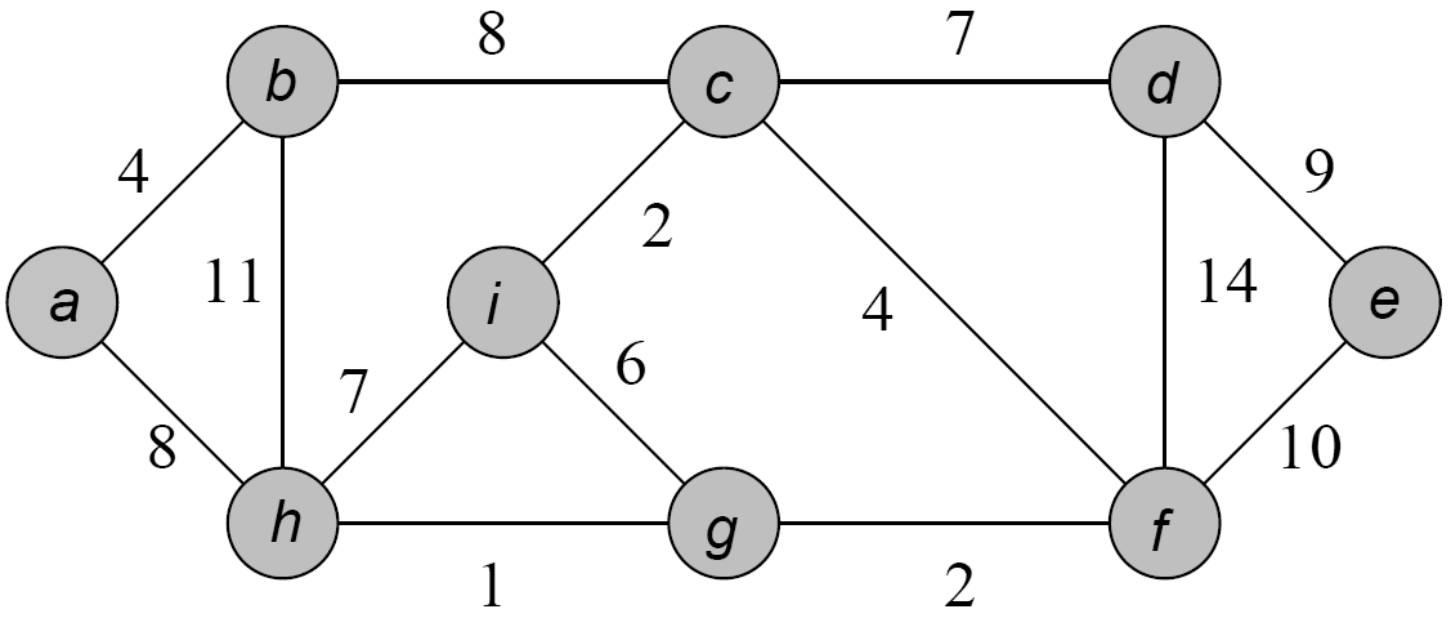


Distâncias em linha reta até Bucharest:

Arad	366	Mehadia	241
Bucharest	0	Neamt	234
Craiova	160	Oradea	380
Drobeta	242	Pitesti	100
Eforie	161	Rimnicu Vilcea	193
Fagaras	176	Sibiu	253
Giurgiu	77	Timisoara	329
Iasi	226	Vaslui	199
Lugoj	244	Zerind	374
Hirsova	151	Urziceni	80

Algoritmo de Prim

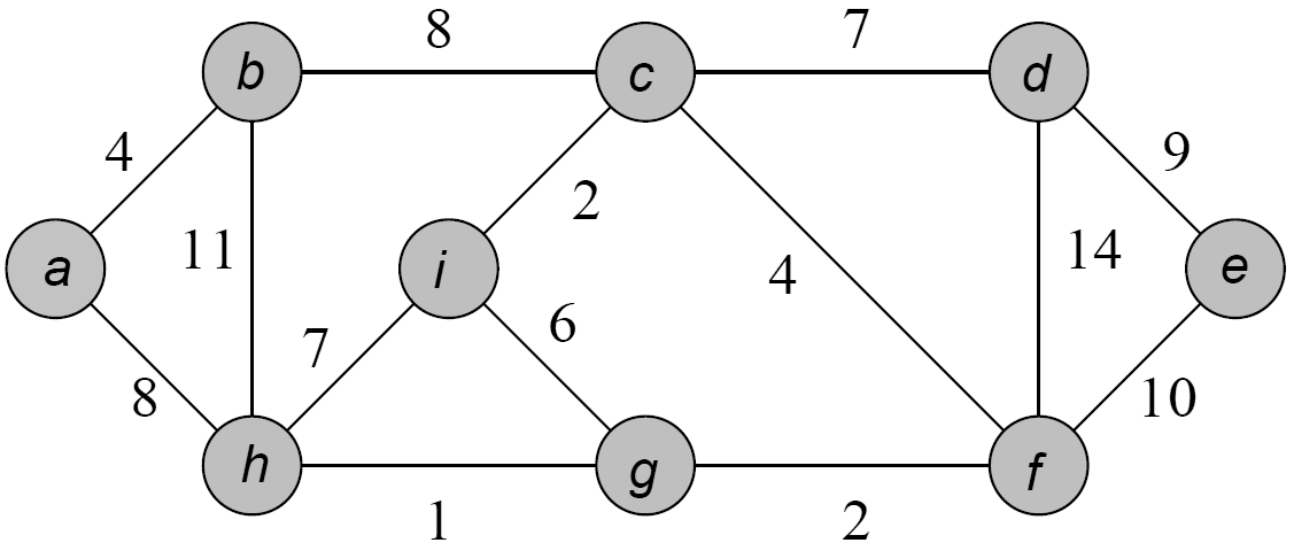
- **Ideia**: a partir de um vértice inicial, selecione as arestas de menor peso disponíveis a cada vértice não visitado.
  - Sem ciclos, pois só visita novos vertices.



Algoritmo de Prim

```
AGM-PRIM(G, w, r)
  for each u ∈ V[G]
    key[u] ← ∞;
    π[u] ← NULL;
  key[r] ← 0;
  Q ← V[G];   //heap ordenado por key[v]
  while Q ≠ ∅
    u ← POP-MIN(Q);
    for each v ∈ Adj[u]
      if (v ∈ Q) and (w(u, v) < key[v])
        π[v] ← u;
        key[v] ← w(u, v);
  return π;
```

Algoritmo de Prim

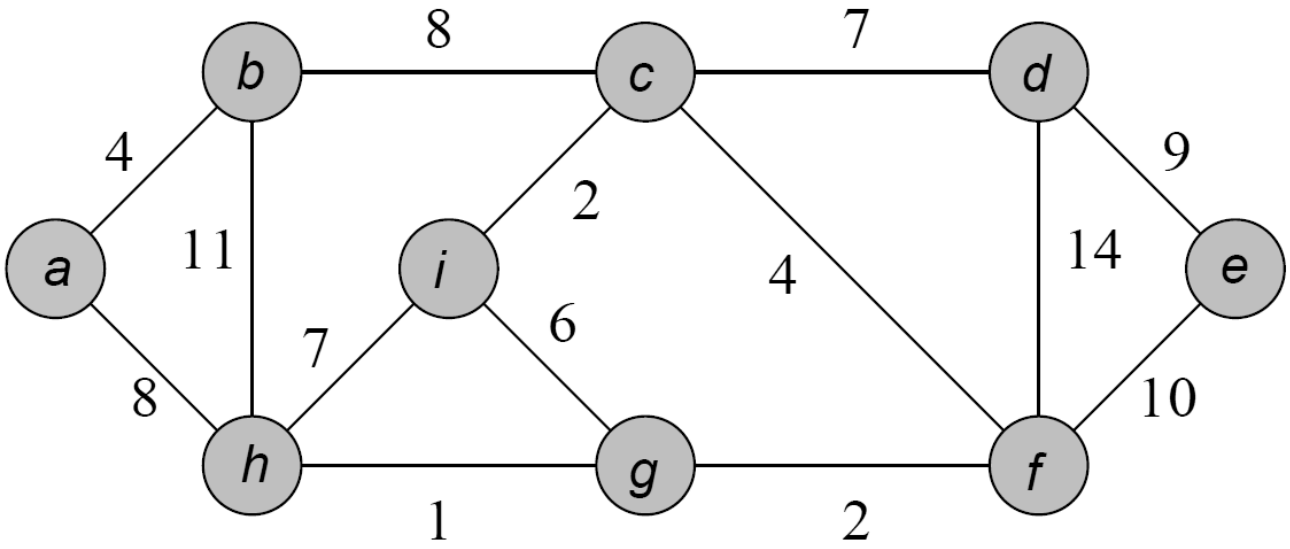



```
AGM-PRIM(G, w, r)
  for each u ∈ V[G]
    key[u] ← ∞;
    π[u] ← NULL;
  key[r] ← 0;
  Q ← V[G];
```

	a	b	c	d	e	f	g	h	i
π									
key									

Q	

Algoritmo de Prim

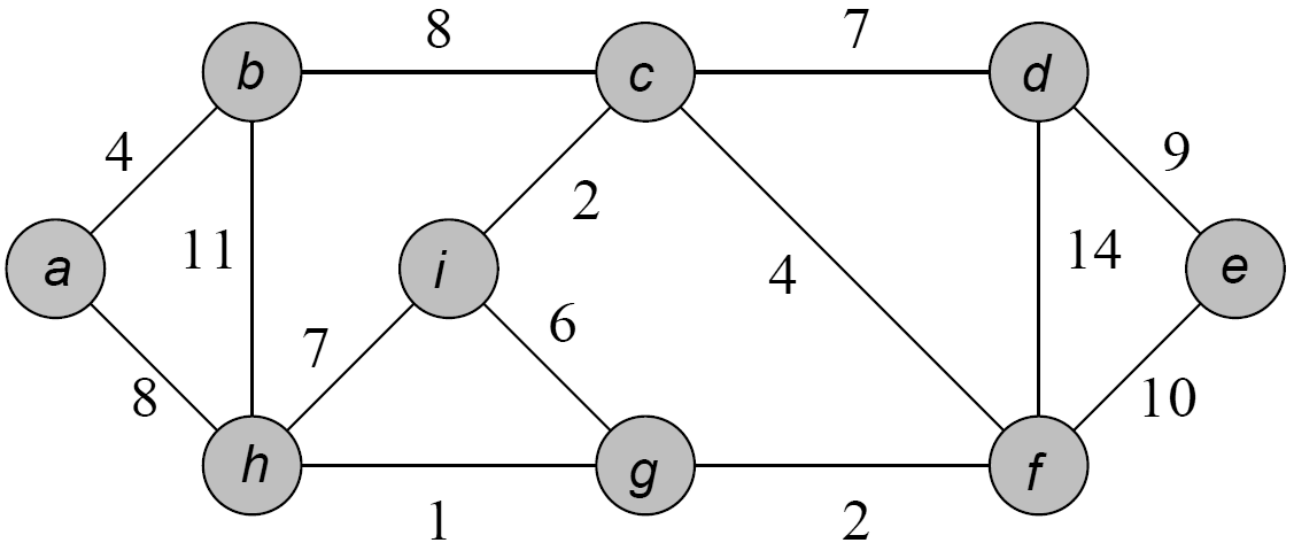



```
while Q ≠ ∅
  u ← POP-MIN(Q);
  for each v ∈ Adj[u]
    if (v ∈ Q) and (w(u, v)<key[v])
      π[v] ← u;
      key[v] ← w(u, v);
```

	a	b	c	d	e	f	g	h	i
π									
key									

Q	

Algoritmo de Prim

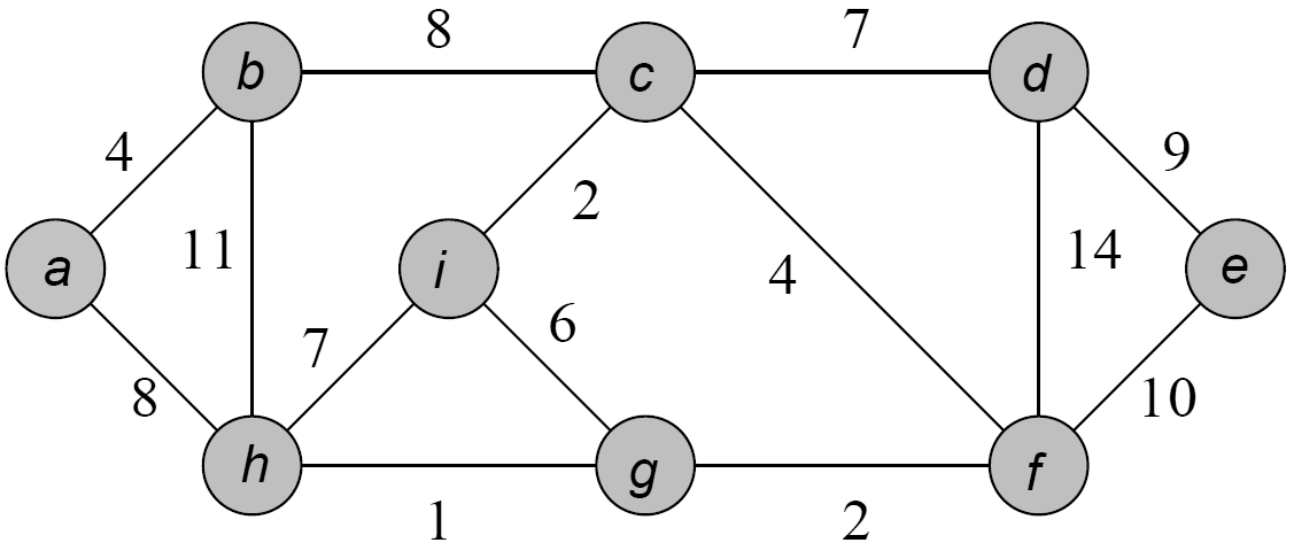



```
while Q ≠ ∅
  u ← POP-MIN(Q);
  for each v ∈ Adj[u]
    if (v ∈ Q) and (w(u, v)<key[v])
      π[v] ← u;
      key[v] ← w(u, v);
```

	a	b	c	d	e	f	g	h	i
π	/	a	/	/	/	/	/	a	/
key	0	4	∞	∞	∞	∞	∞	8	∞

Q	a	b	c	d	e	f	g	h	i

Algoritmo de Prim

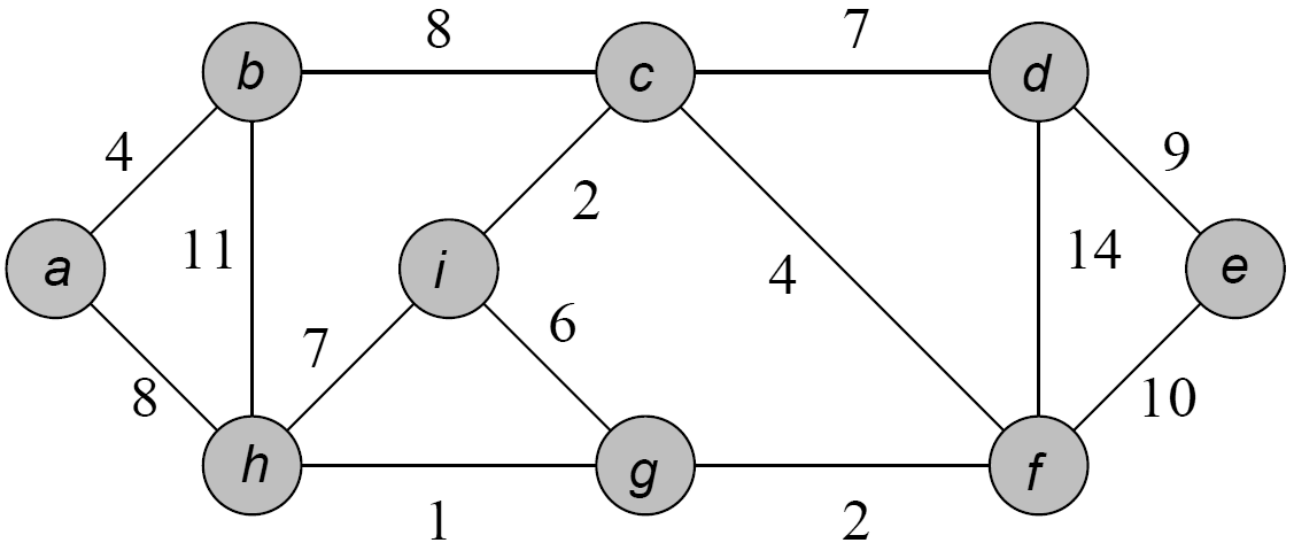



```
while Q ≠ ∅
  u ← POP-MIN(Q);
  for each v ∈ Adj[u]
    if (v ∈ Q) and (w(u, v)<key[v])
      π[v] ← u;
      key[v] ← w(u, v);
```

	a	b	c	d	e	f	g	h	i
π	/	a	b	/	/	/	/	a	/
key	0	4	8	∞	∞	∞	∞	8	∞

Q		b	h	c	d	e	f	g	i

Algoritmo de Prim

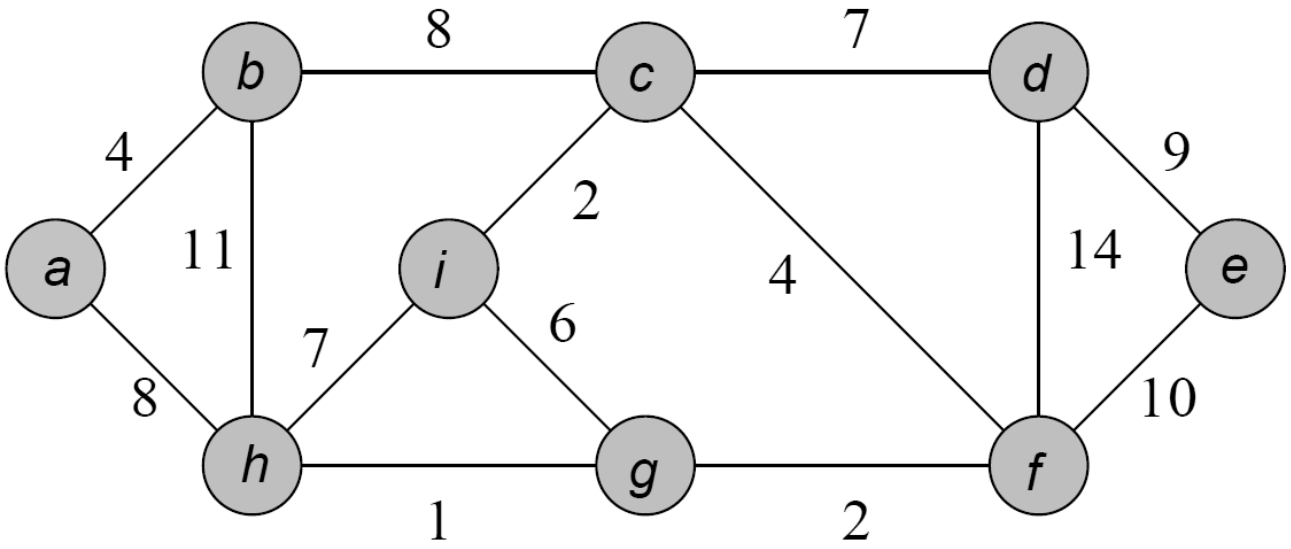



```
while Q ≠ ∅
  u ← POP-MIN(Q);
  for each v ∈ Adj[u]
    if (v ∈ Q) and (w(u, v)<key[v])
      π[v] ← u;
      key[v] ← w(u, v);
```

	a	b	c	d	e	f	g	h	i
π	/	a	b	c	/	c	/	a	c
key	0	4	8	7	∞	4	∞	8	2

Q			c	h	d	e	f	g	i

Algoritmo de Prim

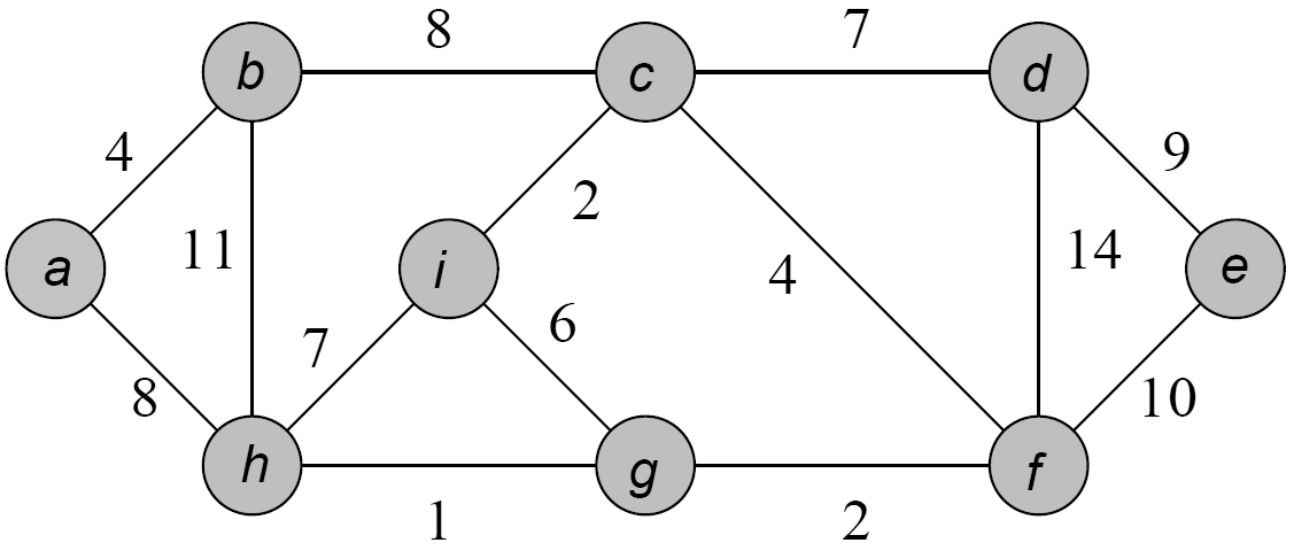



```
while Q ≠ ∅
  u ← POP-MIN(Q);
  for each v ∈ Adj[u]
    if (v ∈ Q) and (w(u, v)<key[v])
      π[v] ← u;
      key[v] ← w(u, v);
```

	a	b	c	d	e	f	g	h	i
π	/	a	b	c	/	c	i	i	c
key	0	4	8	7	∞	4	6	7	2

Q				i	f	d	h	e	g

Algoritmo de Prim

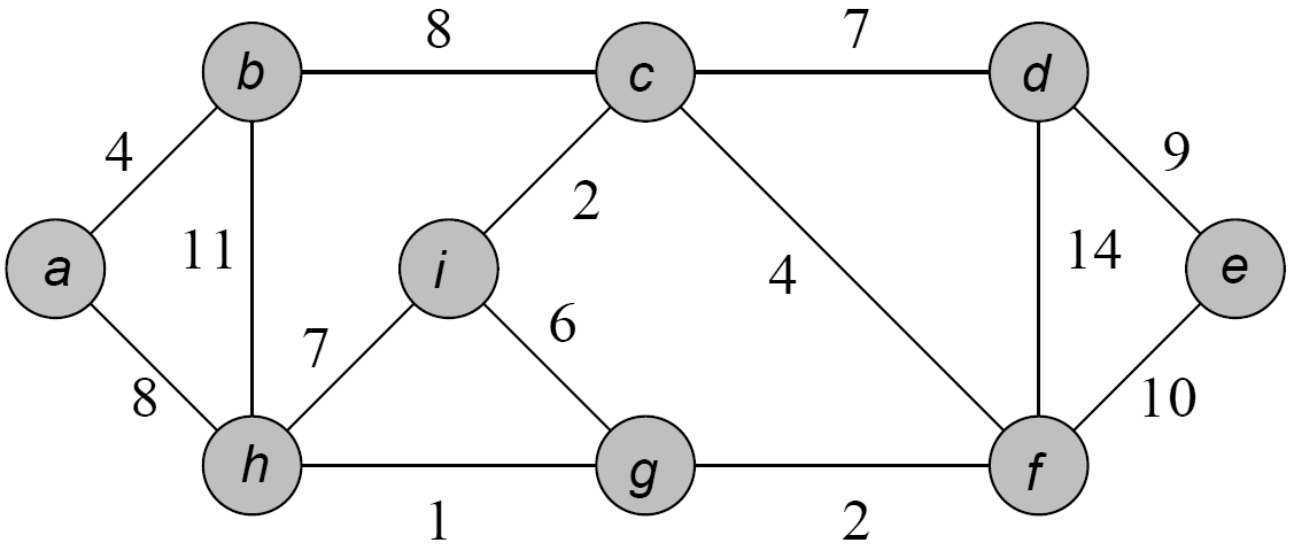



```
while Q ≠ ∅
  u ← POP-MIN(Q);
  for each v ∈ Adj[u]
    if (v ∈ Q) and (w(u, v)<key[v])
      π[v] ← u;
      key[v] ← w(u, v);
```

	a	b	c	d	e	f	g	h	i
π	/	a	b	c	f	c	f	i	c
key	0	4	8	7	10	4	2	7	2

Q					f	g	d	h	e

Algoritmo de Prim

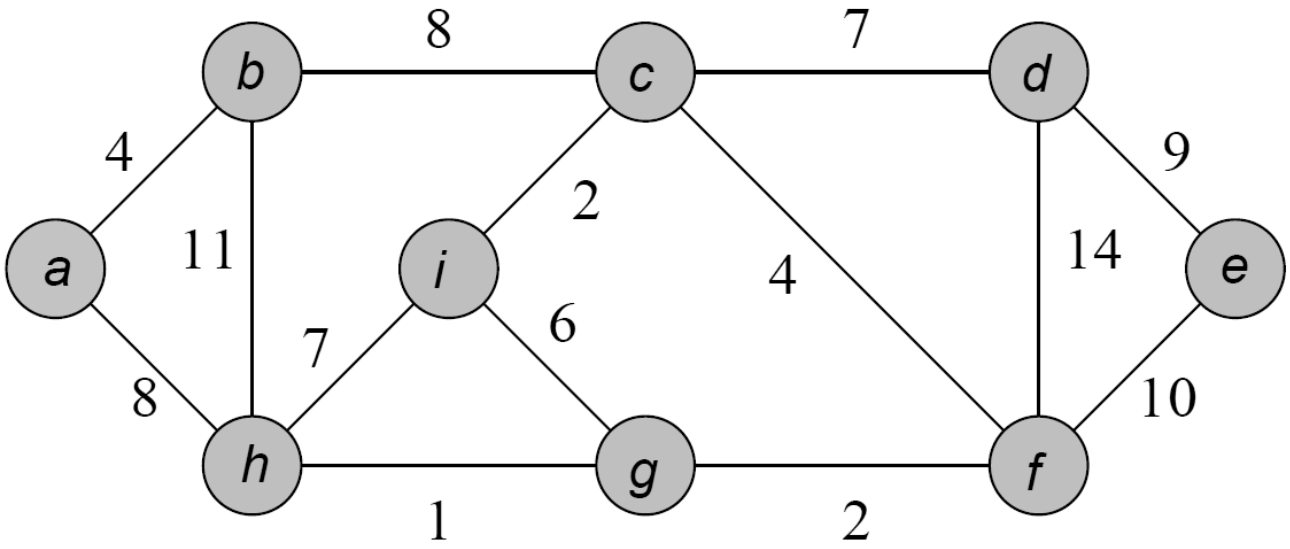



```
while Q ≠ ∅
  u ← POP-MIN(Q);
  for each v ∈ Adj[u]
    if (v ∈ Q) and (w(u, v)<key[v])
      π[v] ← u;
      key[v] ← w(u, v);
```

	a	b	c	d	e	f	g	h	i
π	/	a	b	c	f	c	f	g	c
key	0	4	8	7	10	4	2	1	2

Q						g	d	h	e

Algoritmo de Prim

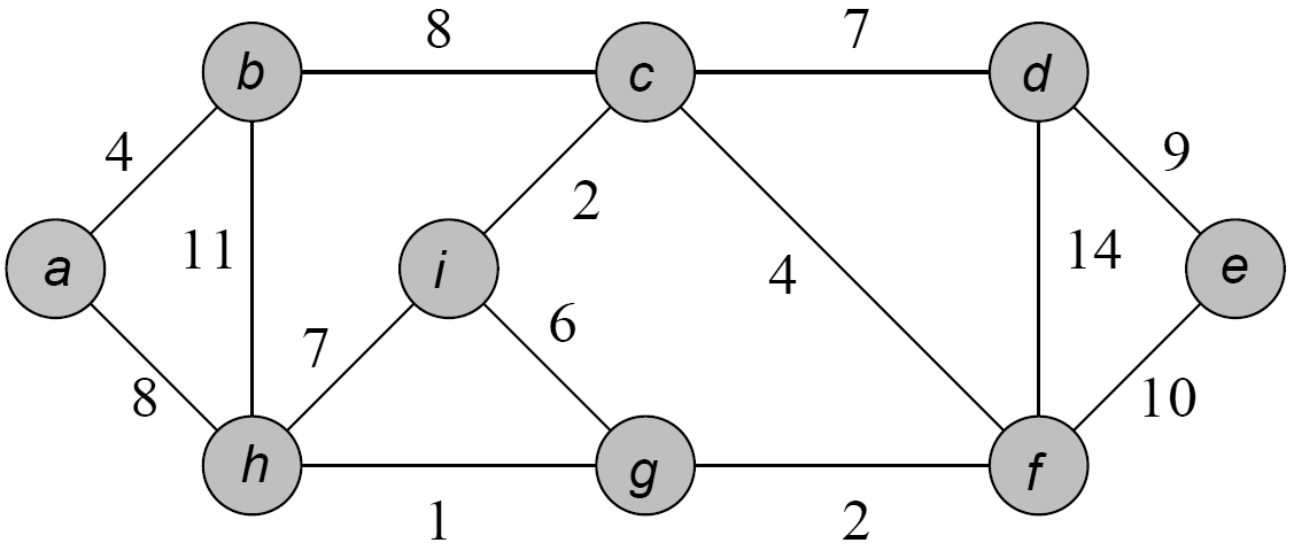



```
while Q ≠ ∅
  u ← POP-MIN(Q);
  for each v ∈ Adj[u]
    if (v ∈ Q) and (w(u, v)<key[v])
      π[v] ← u;
      key[v] ← w(u, v);
```

	a	b	c	d	e	f	g	h	i
π	/	a	b	c	f	c	f	g	c
key	0	4	8	7	10	4	2	1	2

Q							h	d	e

Algoritmo de Prim

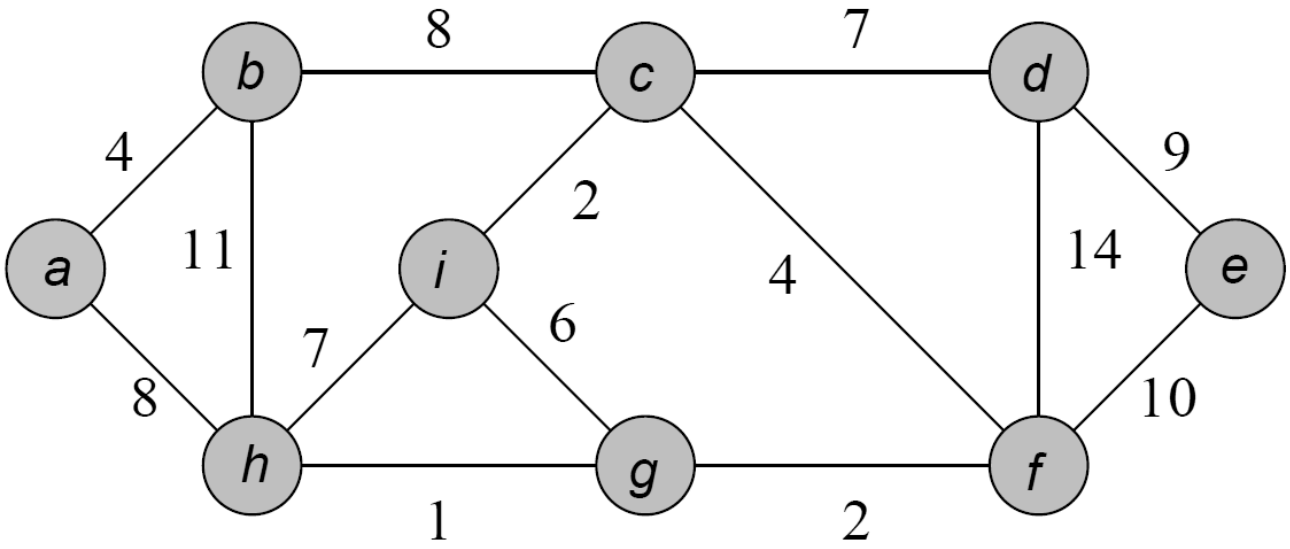



```
while Q ≠ ∅
  u ← POP-MIN(Q);
  for each v ∈ Adj[u]
    if (v ∈ Q) and (w(u, v)<key[v])
      π[v] ← u;
      key[v] ← w(u, v);
```

	a	b	c	d	e	f	g	h	i
π	/	a	b	c	**d**	c	f	g	c
key	0	4	8	7	**9**	4	2	1	2

Q								**d**	e

Algoritmo de Prim

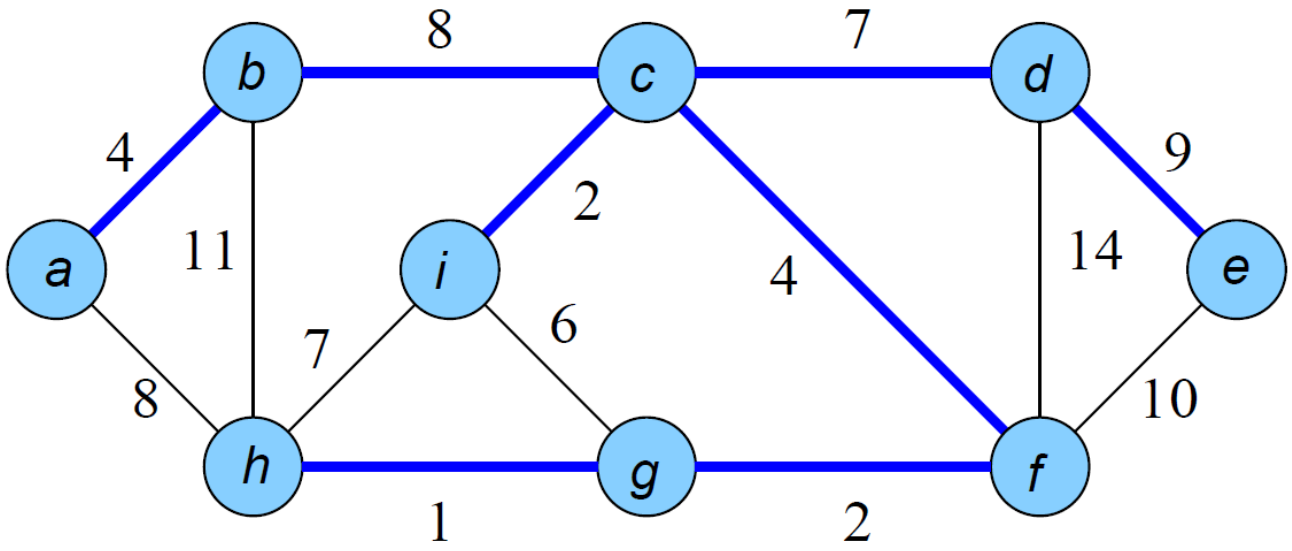



```
while Q ≠ ∅
  u ← POP-MIN(Q);
  for each v ∈ Adj[u]
    if (v ∈ Q) and (w(u, v)<key[v])
      π[v] ← u;
      key[v] ← w(u, v);
```

	a	b	c	d	e	f	g	h	i
π	/	a	b	c	d	c	f	g	c
key	0	4	8	7	9	4	2	1	2

Q									e

Algoritmo de Prim




```
return π;
```

	a	b	c	d	e	f	g	h	i
π	/	a	b	c	d	c	f	g	c
key	0	4	8	7	9	4	2	1	2

Q									

Algoritmo de Prim – Análise

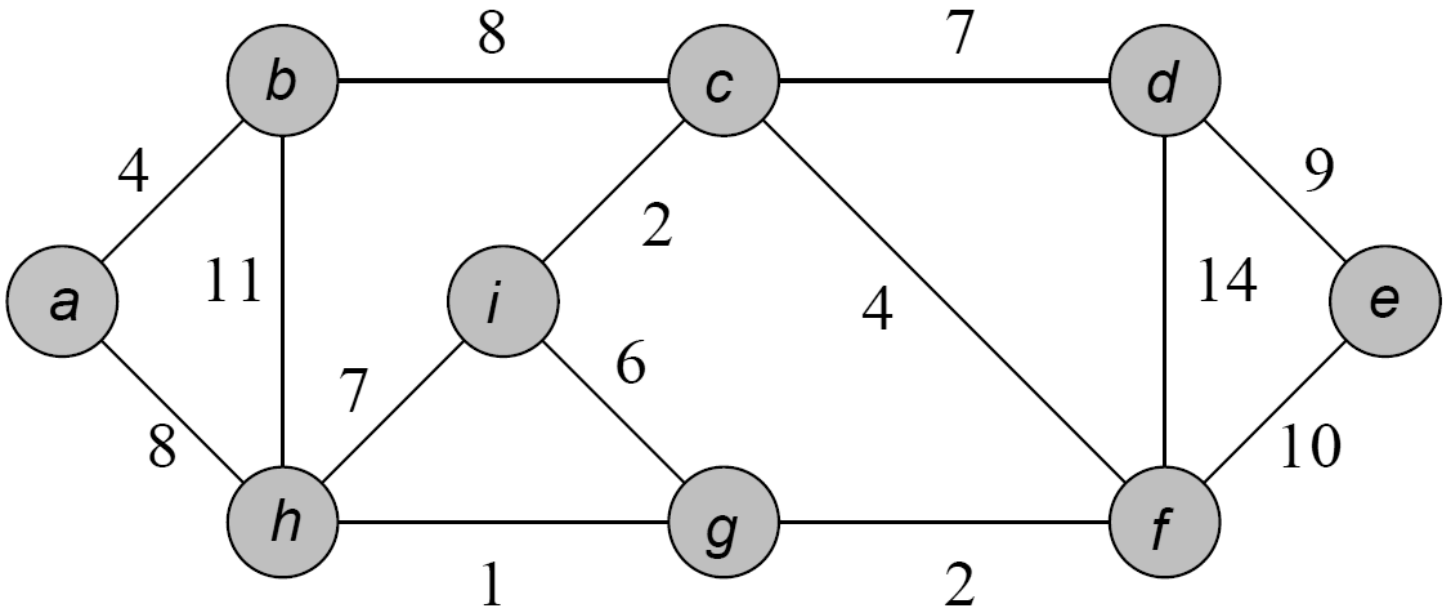
```
AGM-PRIM(G, w, r)
  for each u ∈ V[G]
    key[u] ← ∞;
    π[u] ← NULL;
  key[r] ← 0;
  Q ← V[G];  //heap ordenado por key[v]
  while Q ≠ ∅
    u ← POP-MIN(Q);
    for each v ∈ Adj[u]
      if (v ∈ Q) and (w(u, v) < key[v])
        π[v] ← u;
        key[v] ← w(u, v);
  return π;
```

- Inicializar key e π: O(V)
- Inicializar heap Q: O(V)
- Percorrer vértices em Q: O(V)
- POP-MIN: O(log V)
- Percorrer vértices adjacentes: O(A)
- Atualizar key e heap Q: O(log V)
- Complexidade:

O((V + A) log V)

Algoritmo de Kruskal

- **Ideia**: selecionar arestas de menor peso sucessivamente até que uma árvore geradora seja obtida
  - Como as arestas são as de menor peso, é seguro adicioná-las à arvore geradora mínima.



Algoritmo de Kruskal

- Versão Inicial:

```
KRUSKAL(G)
  A = ∅;

  sort G[E] by weight(u, v);

  for each (u, v) ∈ G[E]
    if (u, v) estão em componentes distintos de G_A
      A ← A ∪ {(u, v)};

  return A
```

- **Problema:** como verificar eficientemente se u e v estão no mesmo componente da floresta $G_A = (V, A)$?

Algoritmo de Kruskal

- Inicialmente $G_A = (V, \emptyset)$, ou seja, G_A corresponde à floresta onde cada componente é um vértice isolado.

- Ao longo do algoritmo, esses componentes são modificados pela inclusão de arestas em A.

- Uma estrutura de dados para representar $G_A = (V, A)$ deve ser capaz de executar eficientemente as seguintes operações:
  - Dado um vértice u, **determinar** o componente de G_A que contém u;
  - Dados dois vértices u e v em componentes distintos C e C', fazer a **união** desses em um novo componente.

Conjuntos Disjuntos

- Uma estrutura de dados para conjuntos disjuntos mantém uma coleção $\{S_1, S_2, \ldots, S_k\}$ de **conjuntos disjuntos dinâmicos** (isto é, eles mudam ao longo do tempo).

- Cada conjunto é identificado por um **representante** que é um elemento do conjunto.
  - Quem é o representante é irrelevante, mas se o conjunto não for modificado, então o representante não pode ser alterado.

Conjuntos Disjuntos

- Uma estrutura de dados para conjuntos disjuntos deve ser capaz de executar as seguintes operações:
  - **MAKE-SET(x):** cria um novo conjunto contendo somente o elmento {x}.
  - **UNION(x, y):** une os conjuntos (disjuntos) que contém x e y, digamos S_x e S_y, em um novo conjunto $S_x \cup S_y$. Os conjuntos S_x e S_y são descartados da coleção.
  - **FIND-SET(x):** devolve um apontador para o representante do (único) conjunto que contém x.

Algoritmo de Kruskal

```
KRUSKAL(G)
  A = ∅;
  for each v ∈ G[V]
    MAKE-SET(v);

  sort G[A] by weight(u, v);

  for each (u, v) ∈ G[A]
    if FIND-SET(u) ≠ FIND-SET(v)
      A ← A ∪ {(u, v)};
      UNION(u, v);

  return A
```

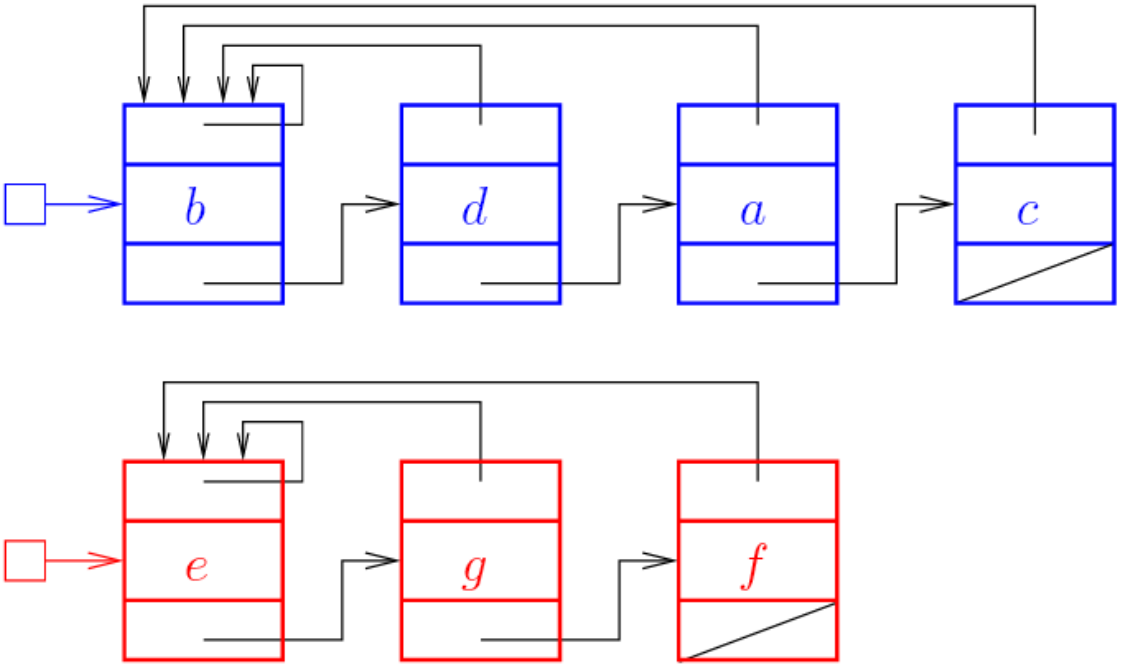
Algoritmo de Kruskal – Análise

```
KRUSKAL(G)
  A = ∅;
  for each v ∈ G[V]
    MAKE-SET(v);

  sort G[A] by weight(u, v);

  for each (u, v) ∈ G[A]
    if FIND-SET(u) ≠ FIND-SET(v)
      A ← A ∪ {(u, v)};
      UNION(u, v);

  return A
```

- Ordenação: O(A log A)
- V chamadas a MAKE-SET
- 2 A chamadas a FIND-SET
- V – 1 chamadas a UNION
- **A complexidade depende de como as operações são implementadas.**

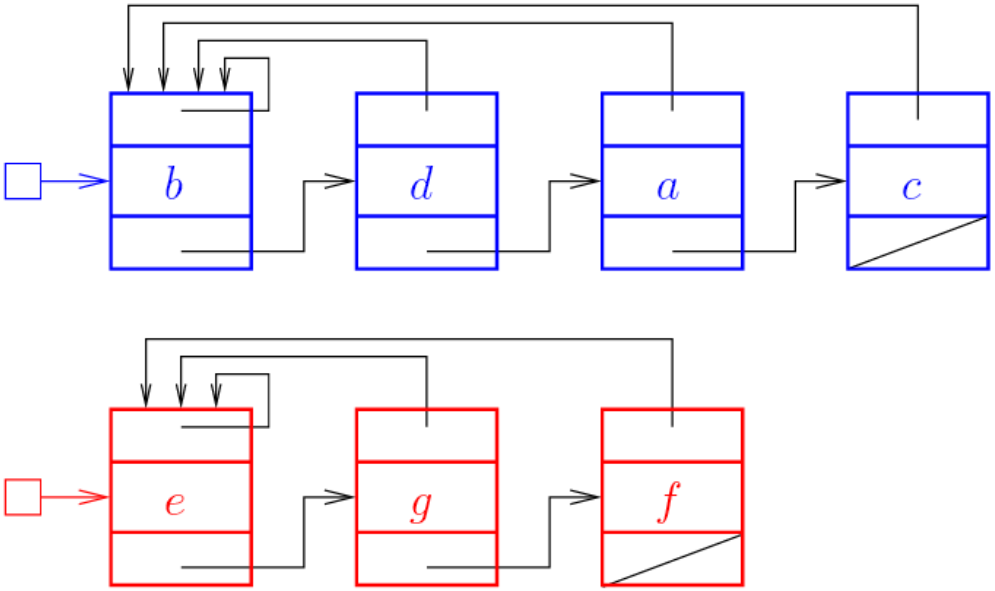
Conjuntos Disjuntos com Listas Encadeadas

- Cada conjunto tem um representante (início da lista);
- Cada nó tem um campo que aponta para o representante;



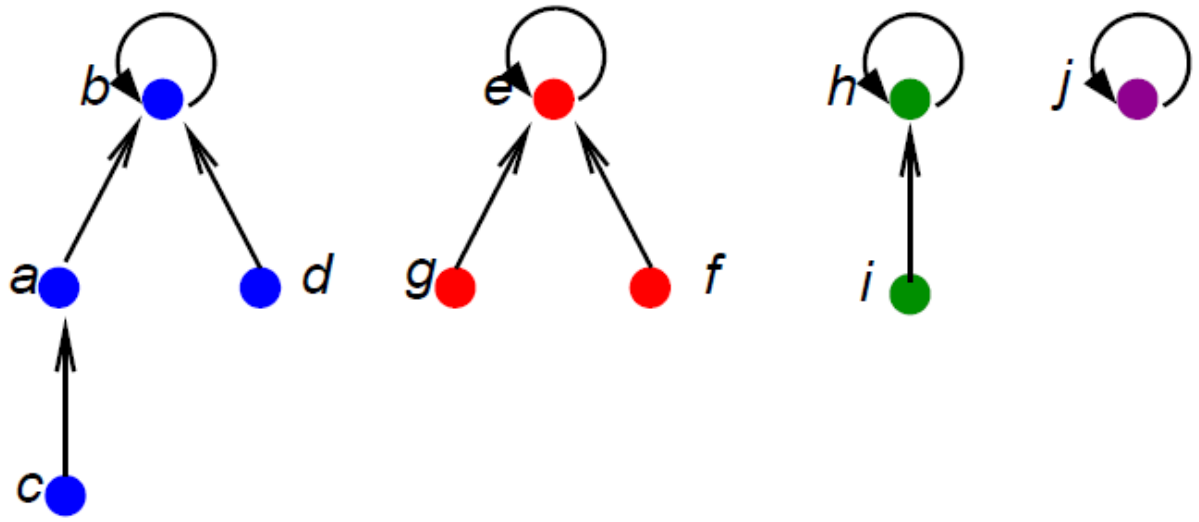
Conjuntos Disjuntos com Listas Encadeadas

- Complexidade:
  - **MAKE-SET(x):** O(1)
  - **FIND-SET(x):** O(1)
  - **UNION(x, y):** O(n)

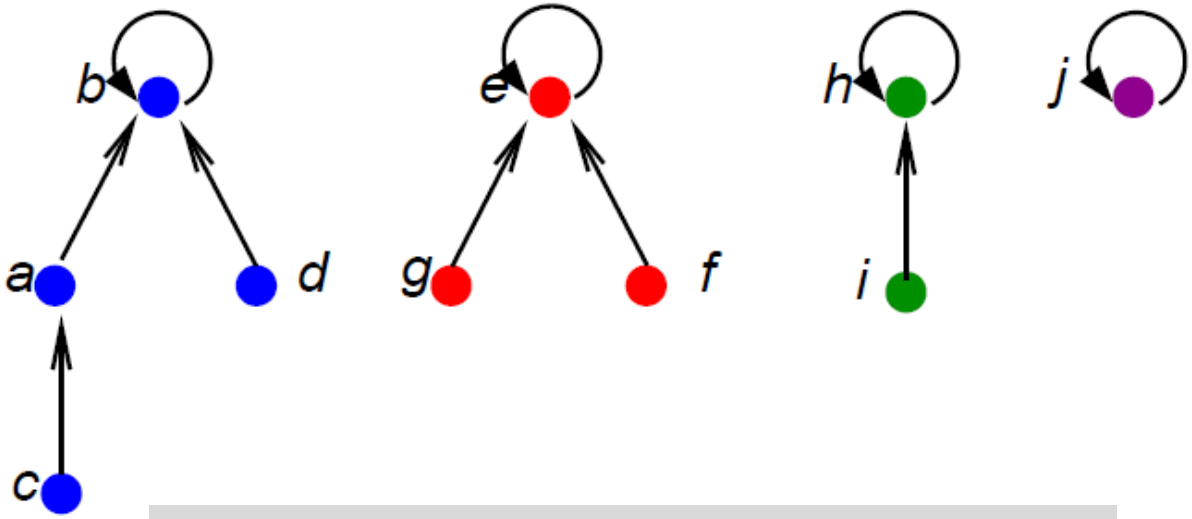


Conjuntos Disjuntos com Florestas

- Cada conjunto corresponde a uma árvore;
- Cada elemento aponta para seu pai;
- A raiz é o representante do conjunto e aponta para si mesma.



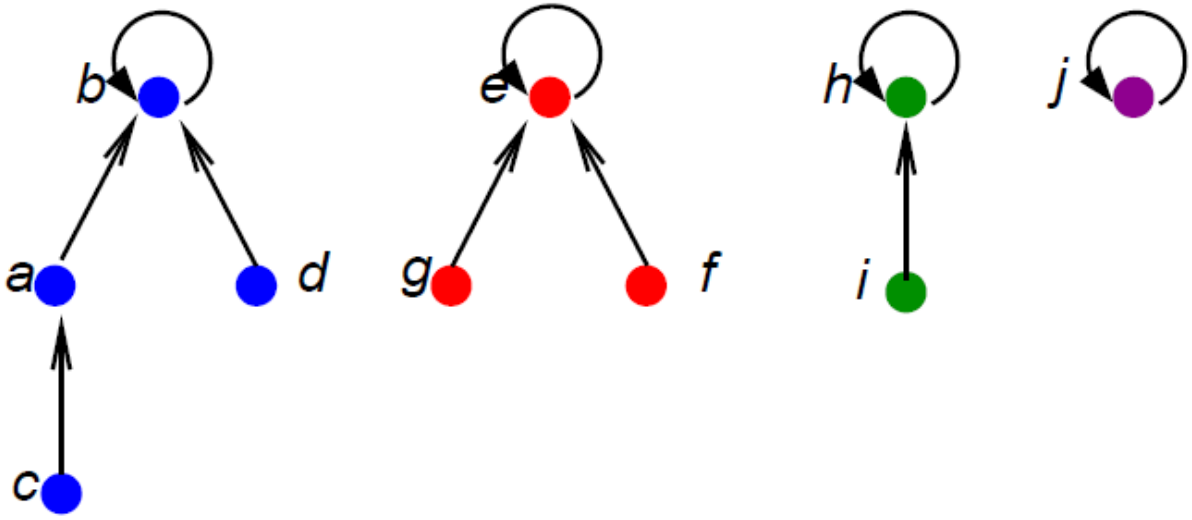
Conjuntos Disjuntos com Florestas




```
MAKE-SET(x)
  pai[x] ← x;
```

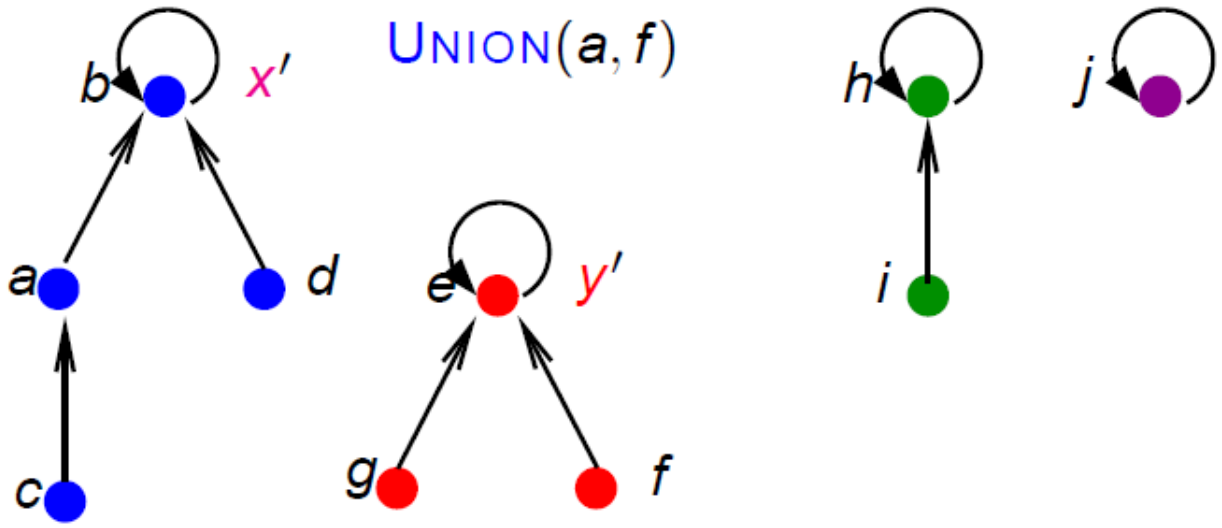
```
FIND-SET(x)
  if x = pai[x]
    return x;
  else
    return FIND-SET(pai[x]);
```

Conjuntos Disjuntos com Florestas



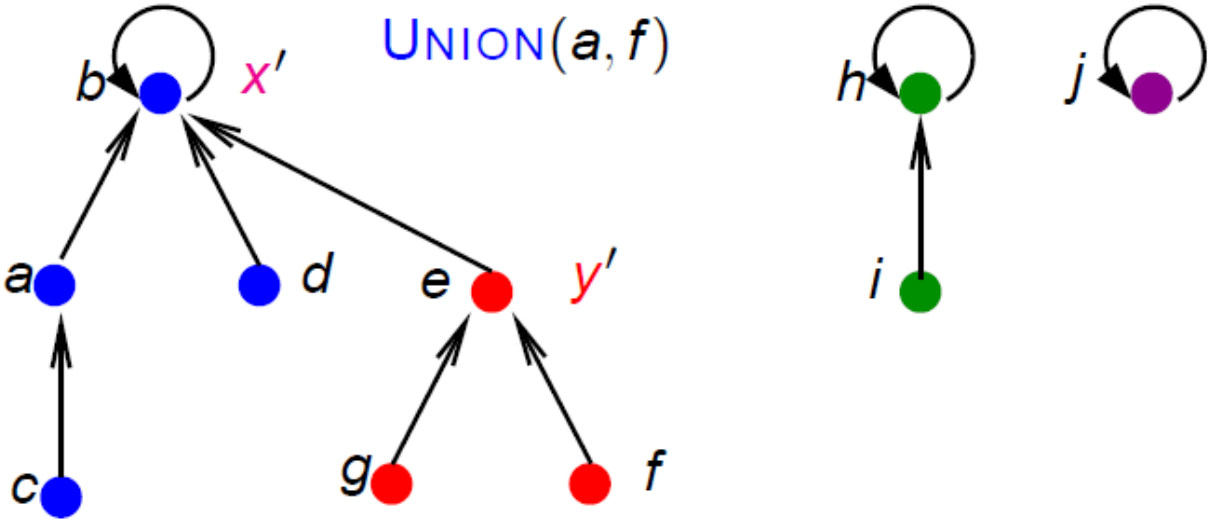

```
UNION(x, y)
  x' ← FIND-SET(x);
  y' ← FIND-SET(y);
  pai[y'] ← x'
```

Conjuntos Disjuntos com Florestas




```
UNION(x, y)
  x' ← FIND-SET(x);
  y' ← FIND-SET(y);
  pai[y'] ← x'
```

Conjuntos Disjuntos com Florestas




```
UNION(x, y)
  x' ← FIND-SET(x);
  y' ← FIND-SET(y);
  pai[y'] ← x'
```

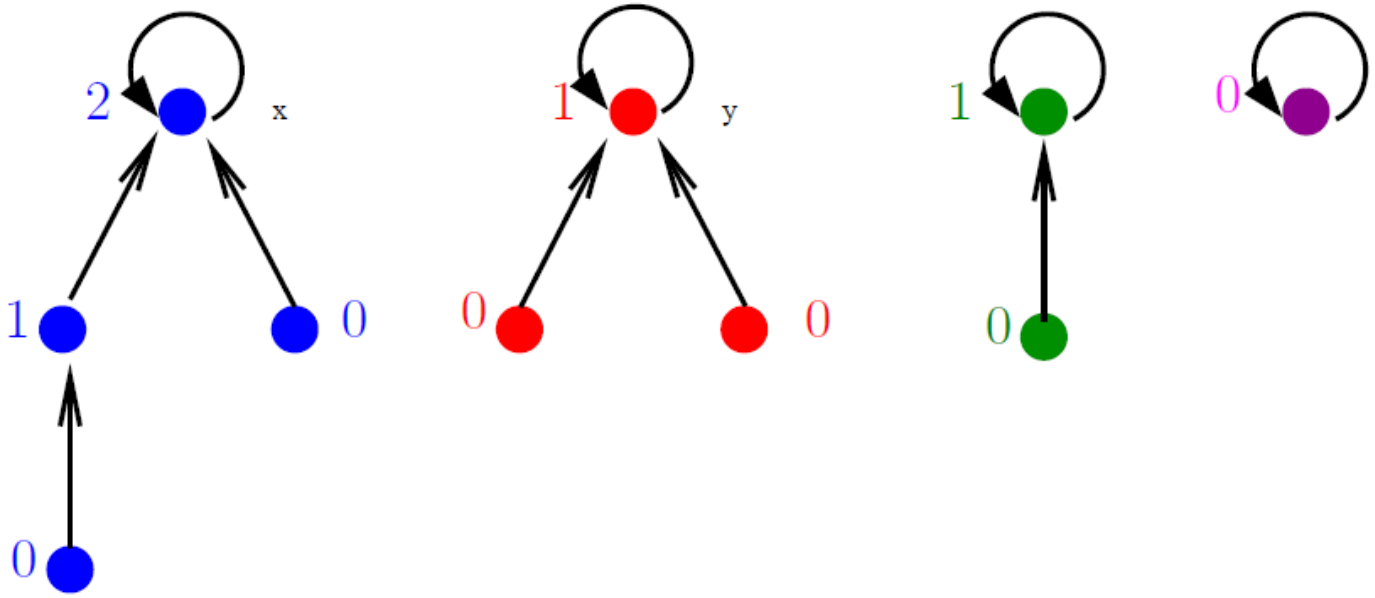
Conjuntos Disjuntos com Florestas

- Complexidade?
  - **MAKESET:** (1)
  - **UNION:** O(n)
  - **FINDSET:** O(n)

- Melhorias?
  - Union by rank;
  - Path compression;

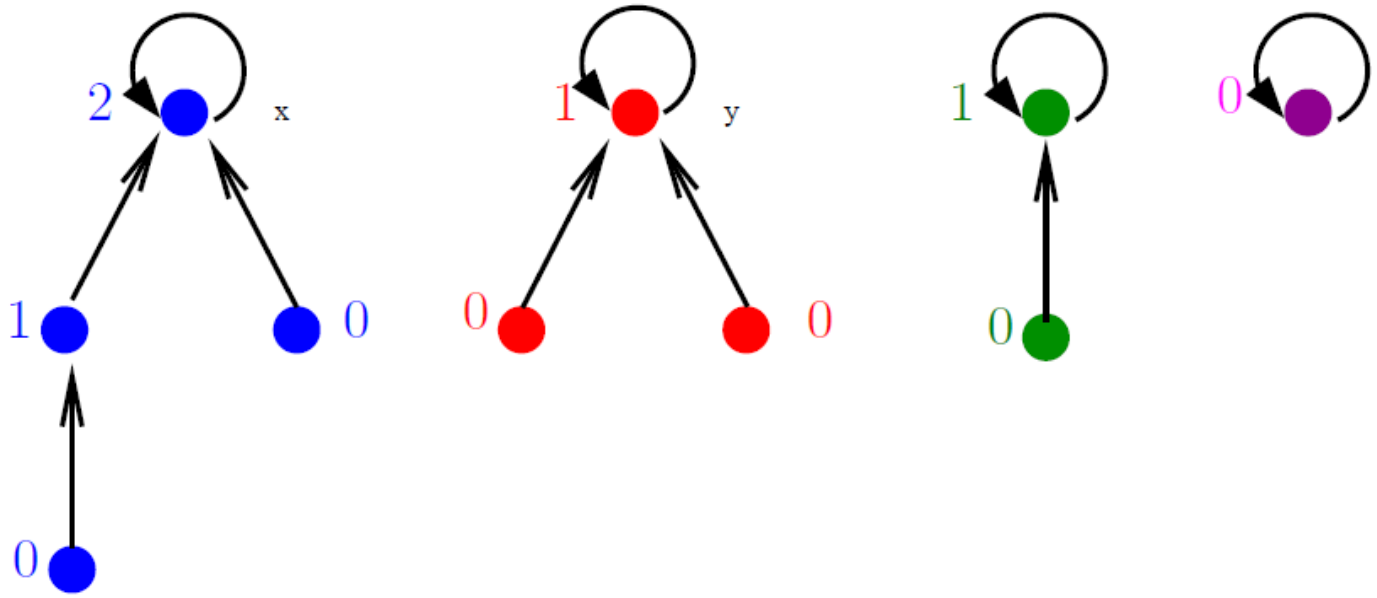
Conjuntos Disjuntos com Florestas

- **Union by rank:**
  - **Objetivo:** sempre unir a menor árvore a raiz da maior;
  - Cada nó x possui um "posto" rank[x] que é um limitante superior para a altura de x;
  - A raiz com menor rank aponta para a raiz com maior rank;



Conjuntos Disjuntos com Florestas

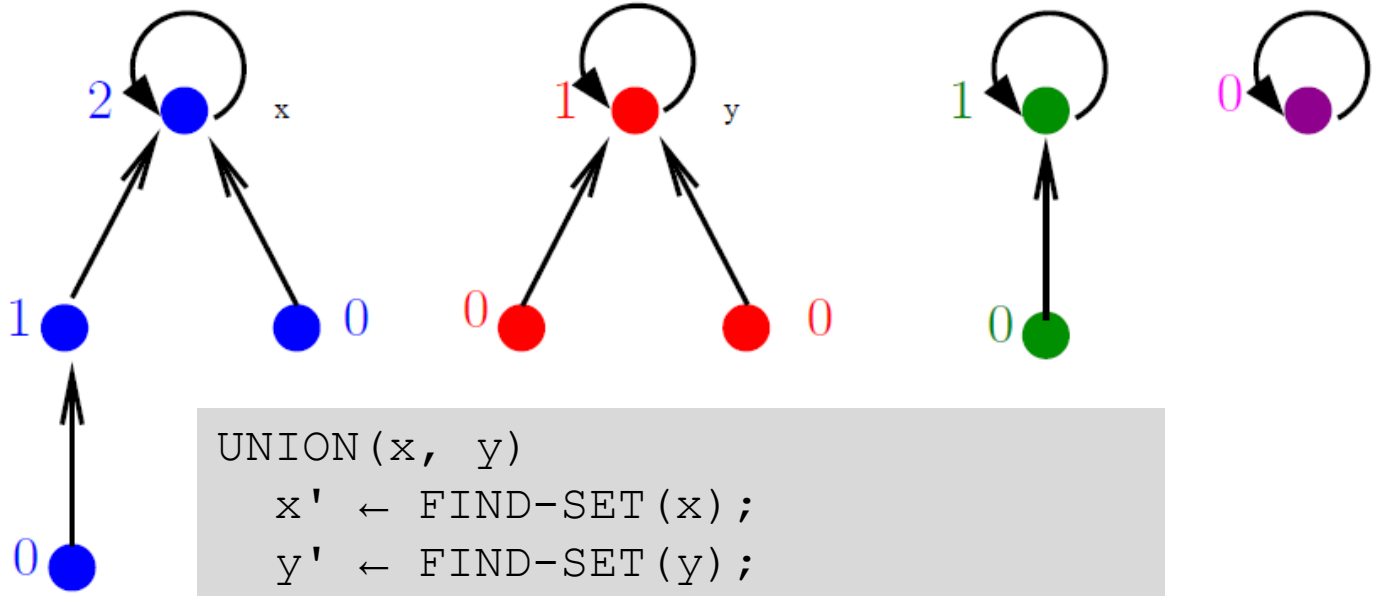
- **Union by rank:**




```
MAKE-SET(x)
  pai[x] ← x;
  rank[x] ← 0;
```

Conjuntos Disjuntos com Florestas

- **Union by rank:**



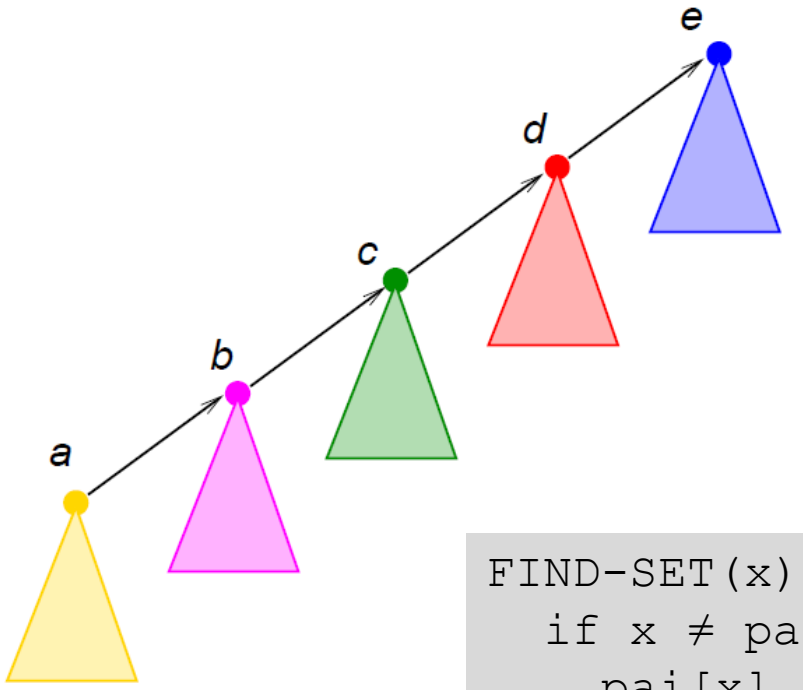

```
UNION(x, y)
  x' ← FIND-SET(x);
  y' ← FIND-SET(y);
  if rank[x'] < rank[y']
    pai[y'] ← x'
  else
    pai[x'] ← y'
  if rank[x'] = rank[y']
    rank[y'] ← rank[y'] + 1
```

Conjuntos Disjuntos com Florestas

- Complexidade com o union by rank:
  - **MAKESET:** (1)
  - **UNION:** O(log n)
  - **FINDSET:** O(log n)

Conjuntos Disjuntos com Florestas

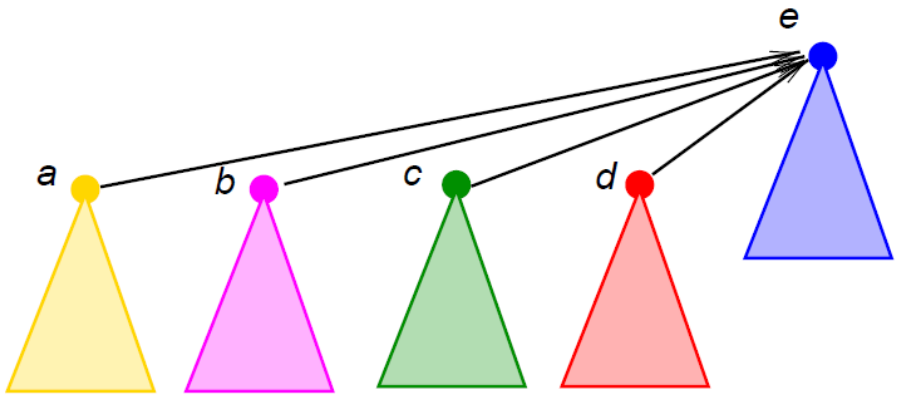
- **Path compression:**
  - Ao tentar determinar o representante (raiz da árvore) de um nó fazemos com que todos os nós no caminho apontem para a raiz.



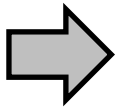

```
FIND-SET(x)
  if x ≠ pai[x]
    pai[x] ← FIND-SET(pai[x])
  return pai[x]
```

Conjuntos Disjuntos com Florestas

- **Path compression:**
  - Ao tentar determinar o representante (raiz da árvore) de um nó fazemos com que todos os nós no caminho apontem para a raiz.




```
FIND-SET(x)
  if x = pai[x]
    return x;
  return FIND-SET(pai[x]);
```



```
FIND-SET(x)
  if x ≠ pai[x]
    pai[x] ← FIND-SET(pai[x])
  return pai[x]
```

Conjuntos Disjuntos com Florestas

- Complexidade com o union by rank e Path compression:
  - **MAKESET:** (1)
  - **UNION:** O(α(n))
  - **FINDSET:** O(α(n))

Conjuntos Disjuntos com Florestas

- Função de Ackermann:

$$A(m,n) = \begin{cases} n+1 & \text{if } m = 0 \\ A(m-1,1) & \text{if } m > 0 \text{ and } n = 0 \\ A(m-1, A(m,n-1)) & \text{if } m > 0 \text{ and } n > 0. \end{cases}$$

- A complexidade cresce muito rapidamente:
  - A(1, 0) = 2
  - A(1, 1) = 3
  - A(2, 2) = 7
  - A(3, 2) = 29
  - A(4, 2) = $2^{65536} - 3$ (mais que o número estimado de atomos do universo!)

- α(n) → Inverso da função de Ackermann → (α(n) ≤ 4)

De Volta ao Algoritmo de Kruskal

```
KRUSKAL(G)
  A = ∅;
  for each v ∈ G[V]
    MAKE-SET(v);

  sort G[E] by weight(u, v);

  for each (u, v) ∈ G[E]
    if FIND-SET(u) ≠ FIND-SET(v)
      A ← A ∪ {(u, v)};
      UNION(u, v);

  return A
```

- Ordenação: O(A log A)
- Chamadas a MAKE-SET: O(V)
- Chamadas a FIND-SET e UNION: O(A)
- **Complexidade:**

  O(A log A)

Exercícios

Lista de Exercícios 09 – Árvore Geradora Mínima

http://www.inf.puc-rio.br/~elima/paa/