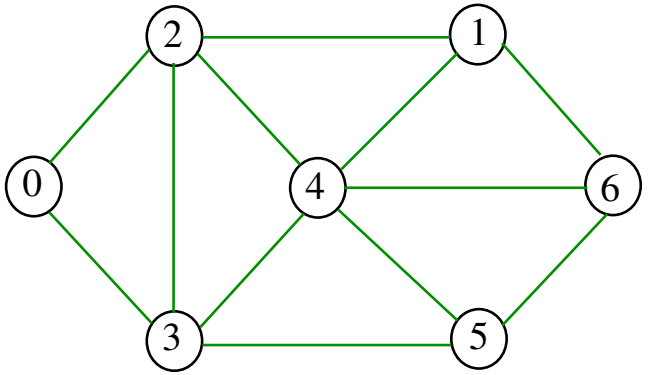
Árvores geradoras de grafos

Árvores geradoras

Uma **árvore geradora** (= *spanning tree*) de um grafo é qualquer subárvore que contenha **todos** os vértices

Exemplo:

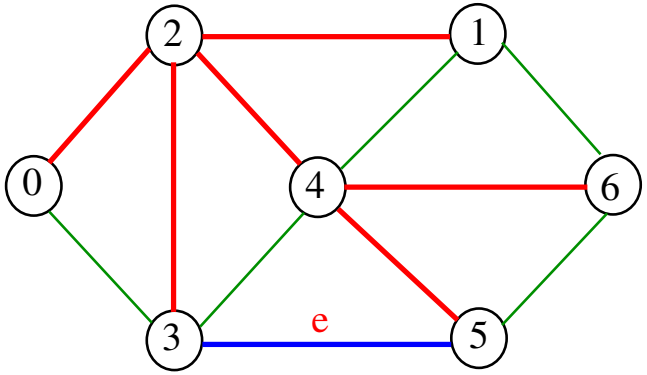


Exemplo: as arestas em vermelho formam uma árvore

Primeira propriedade da troca de arestas

Seja T uma árvore geradora de um grafo G
Para qualquer aresta e de G que não esteja em T, T+e tem um **único ciclo** não-trivial, o **ciclo fundamental** $C(\mathtt{T}, \mathtt{e})$.

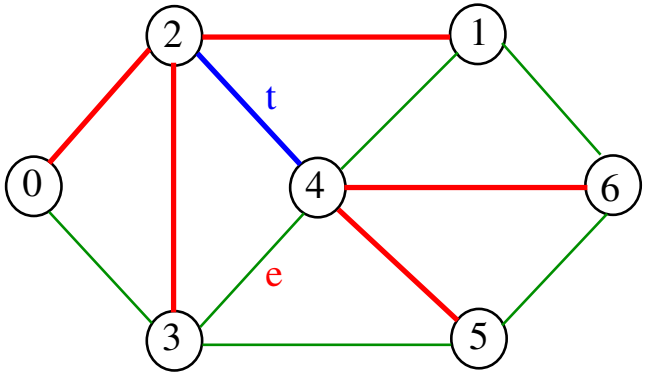
Exemplo: T+e



Segunda propriedade da troca de arestas

Seja T uma árvore geradora de um grafo G
Para qualquer aresta t de T, T-t tem duas componentes. O corte em G que separa essas componentes é o **corte fundamental** $D(\mathtt{T}, \mathtt{t})$.

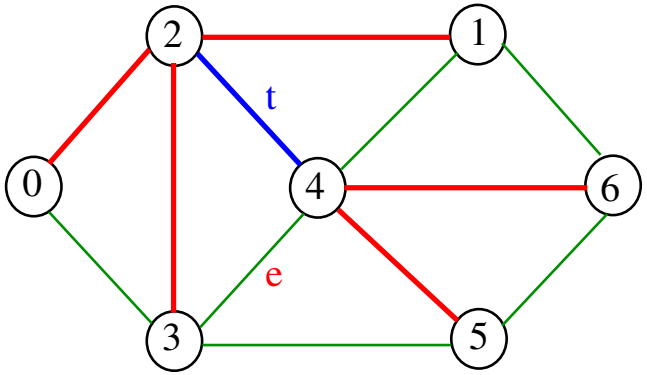
Exemplo: T-t $\{0,1,2,3\} \xrightarrow{D(\mathtt{T},\mathtt{t})} \{4,5,6\}$



Segunda propriedade da troca de arestas

Seja T uma árvore geradora de um grafo G
Para qualquer aresta t de T, se e $\in D(T,t)$ então $T - t + e$ é uma árvore geradora.

Exemplo: T-t



Exemplo: T-t+e

Propriedade fundamental

Se T é árvore geradora de G, t é uma aresta de T e e é uma aresta fora de T, então

$$t \in C(T,e) \iff e \in D(T,t).$$

Dem: São equivalentes

- $t \in C(T,e)$
- t está no caminho em T que liga as pontas de e
- $T - t$ não tem caminho entre as pontas de e
- $e \in D(T,t)$.

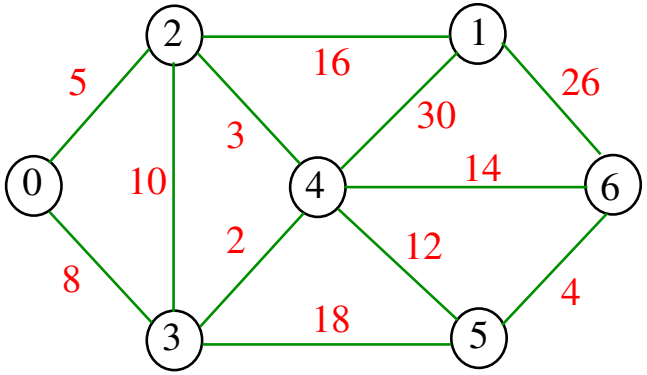
Árvores geradoras de custo mínimo

S 20.1 e 20.2

Árvores geradoras mínimas

Uma **árvore geradora mínima** (= *minimum spanning tree*), ou MST, de um grafo com custos nas arestas é qualquer árvore geradora do grafo que tenha custo mínimo

Exemplo: um grafo com custos nas arestas

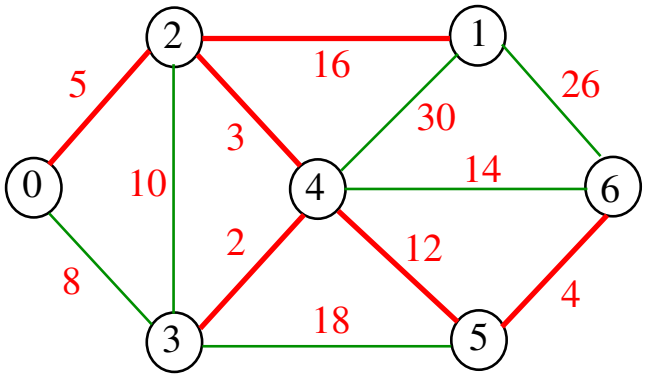


Problema MST

Problema: Encontrar uma MST de um grafo G com custos nas arestas

O problema tem solução se e somente se o grafo G é conexo

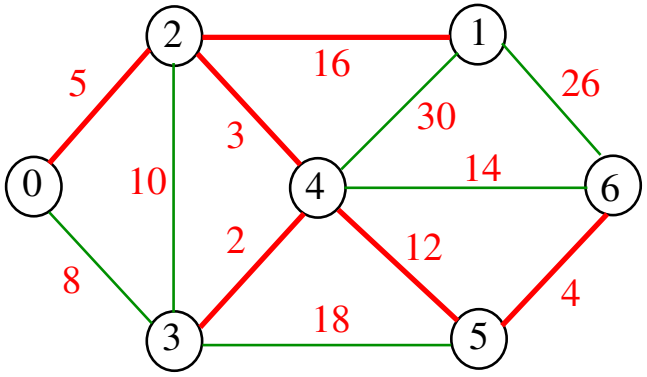
Exemplo: MST de custo 42



Propriedade dos ciclos

Condição de Otimalidade: Se T é uma MST então toda aresta e fora de T tem custo máximo dentre as arestas do ciclo fundamental T, e.

Exemplo: MST de custo 42



Demonstração da recíproca

Seja T uma árvore geradora satisfazendo a condição de otimalidade.

Vamos mostrar que T é uma MST.

Seja M uma MST tal que o número de arestas comuns entre T e M seja máximo.

Se $T = M$ não há o que demonstrar.

Suponha que $T \neq M$ e seja e uma aresta de custo mínimo dentre as arestas que estão em M mas não estão em T.

Seja d uma aresta qualquer que **não está** em M mas **está** no ciclo fundamental $C(T, e)$.

Continuação

Logo, $\texttt{custo}(d) \leq \texttt{custo}(e)$ (1).

Seja f uma aresta qualquer em $C(M, d) - T$.

Como M é uma MST, $\texttt{custo}(f) \leq \texttt{custo}(d)$ (2).

Pela escolha de e, $\texttt{custo}(e) \leq \texttt{custo}(f)$ (3).

Juntando (1), (2) e (3), vem que

$$\texttt{custo}(d) = \texttt{custo}(f) = \texttt{custo}(e)$$

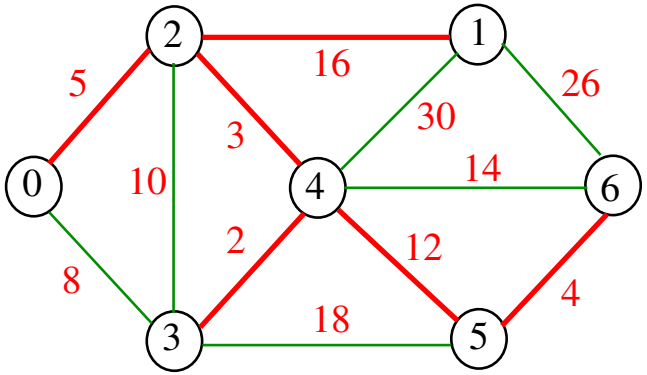
Mas então, M−f+d é uma MST que tem o mesmo custo que M, logo é mínima. Por outro lado, tem uma aresta a mais em comum com T do que M. Isso contradiz a escolha de M.

Portanto, $T = M$, o que mostra que T é uma MST.

Propriedade dos cortes

Condição de Otimalidade: T é uma MST se e somente se cada aresta t de T é uma aresta mínima no corte fundamental T, t.

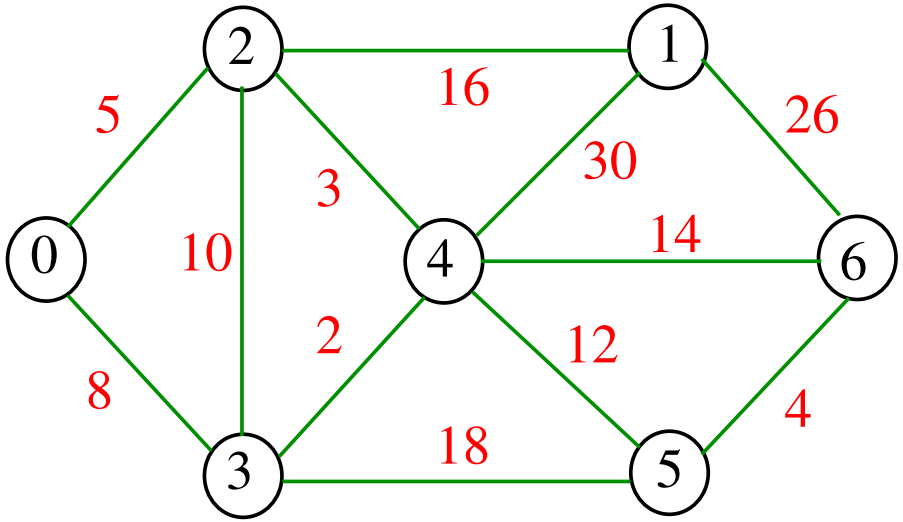
Exemplo: MST de custo 42



Algoritmo de Kruskal

S 20.3

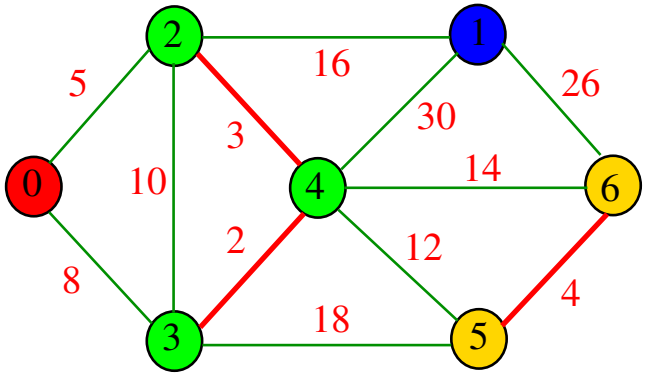
Algoritmo de Kruskal



Subfloresta

Uma **subfloresta** de G é qualquer floresta F que seja subgrafo de G.

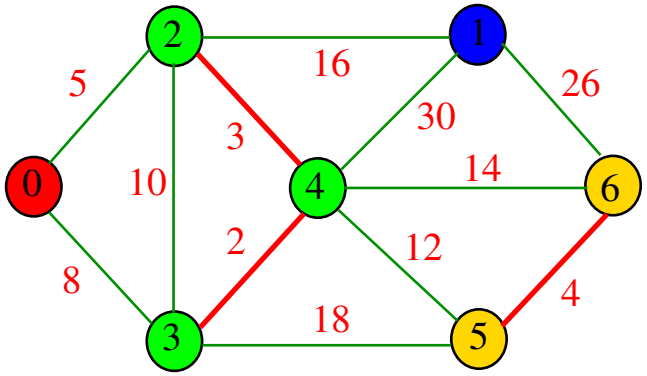
Exemplo: As arestas vermelhas que ligam os vértices verdes e amarelos e formam uma subfloresta



Floresta geradora

Uma **floresta geradora** de G é qualquer subfloresta de G que tenha o mesmo conjunto de vértices que G.

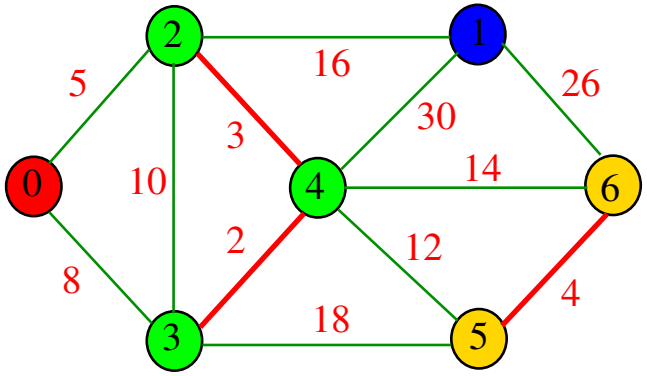
Exemplo: Arestas vermelhas ligando os vértices verdes e amarelos, junto com os vértices azul e vermelho, forma uma floresta geradora



Arestas externas

Uma aresta de G é **externa** em relação a uma subfloresta F de G se tem pontas em árvores distintas de F.

Exemplo: As aresta 0-1, 2-1, 4-6 . . . são externas



Algoritmo de Kruskal

O algoritmo de Kruskal iterativo.

Cada iteração começa com uma floresta geradora F.

No início da primeira iteração cada árvore de F tem apenas 1 vértice.

Cada iteração consiste em:

Caso 1: não existe aresta externa a F
Devolva F e pare.

Caso 2: existe aresta externa a F
Seja e uma aresta externa a F de custo mínimo
Atualize: F← F+e

Otimalidade

Duas demonstrações:

- Usando um invariante, igualzinho à prova do Prim:
  a cada iteração, existe uma MST contendo F.

- Usando o critério já provado que caracteriza MST:
  toda aresta $e \notin F$ tem custo máximo em $C(F, e)$.

Arcos

Um objeto do tipo `Arc` representa um arco com ponta inicial `v` e ponta final `w`.

```
typedef struct {
    Vertex v ;
    Vertex w ;
    double cst;
} Arc;
```


Arestas

Um objeto do tipo `Edge` representa uma aresta com ponta inicial `v` e ponta final `w`.

```
#define Edge Arc
```


Implementação grosseira

Na função abaixo para decidir se uma aresta `v-w` é externa a uma floresta geradora temos que

(1) em cada componente da floresta geradora, é eleito um dos vértices para ser o representante da componente;

(2) é mantido um vetor `cor[0..V-1]` de representantes, sendo `cor[v]` o representante da componente que contém o vértice `v`;

(3) `v-w` é externa se e somente se `cor[v]` é diferente de `cor[w]`.

Kruskal na força bruta

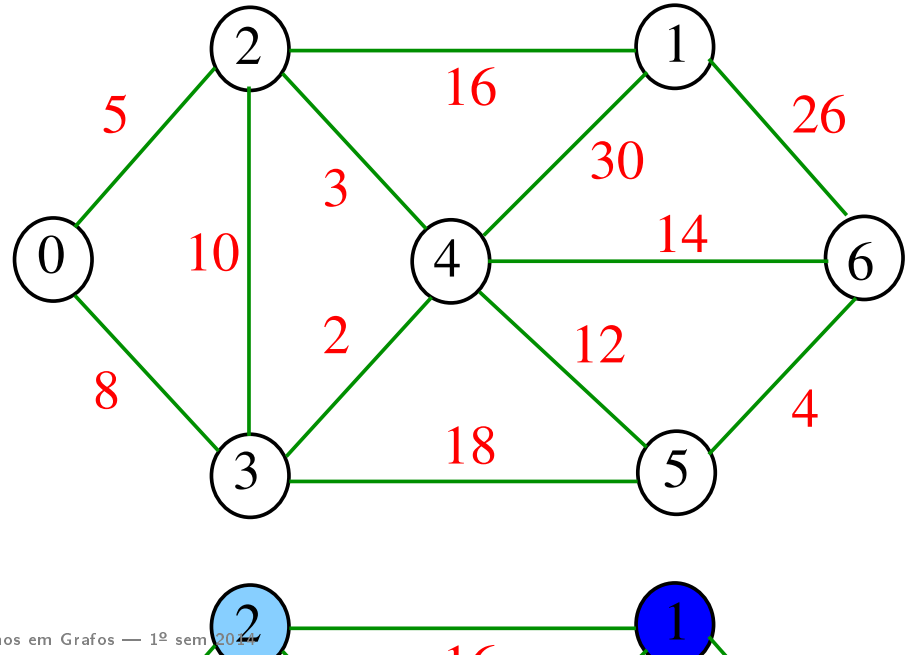
A função armazena as arestas das MSTs no vetor mst[0..k-1] e devolve `k`.

```
int bruteforceKruskal(Graph G, Edge mst[])
{
 0   Vertex cor[maxV], v , w , v0, w0;
 0   int k=0;
 1   for (v=0;v < G->V ; v++)
 3       cor[v] = v ;
```

Implementação grosseira

```
 4   while(1) {
 5   double mincst = INFINITO;
 6   for (v = 0; v < G->V ; v++)
 7       for (w = 0; w < G->V ; w++)
 8           if (cor[v] != cor[w]
                 && mincst> G->adj[v][w])
11           mincst = G->adj[v0=v][w0=w];
12   if (mincst == INFINITO) break;
13   mst[k].v = v0;    mst[k].w = w0;    k++;
14   for (v = 0; v < G->V ; v++)
15       if (cor[v] == cor[w0]) cor[v] = cor[v0];
16   }
return k; }
```


Algoritmo de Kruskal



Union-Find

As as funções `UFinit`, `UFfind` e `UFunion` têm o seguinte papel:

- `UFinit(V)` inicializa a floresta de conjuntos disjuntos com cada árvore contendo apenas 1 elemento;
- `UFfind(v, w)` tem valor 0 se e somente se `v` e `w` estão em componentes distintas da floresta;
- `UFunion(v , w)` promove a união das componentes que contêm `v` e `w` respectivamente.

Implementações eficientes

A função recebe um grafo `G` com custos nas arestas e calcula uma MST em cada componente de `G`.
A função armazena as arestas das MSTs no vetor mst[0..k-1] e devolve `k`.

#define `maxE` 10000

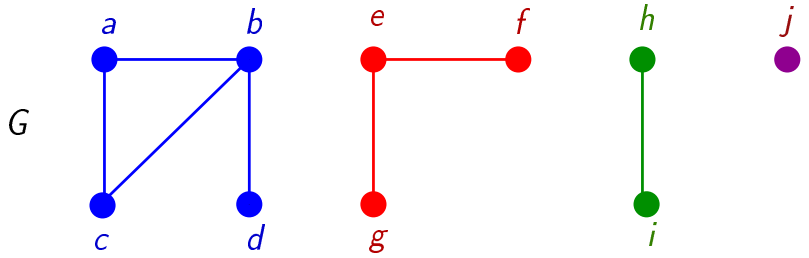
Kruskal

```
int GRAPHmstK (Graph G, Edge mst[]){
0   int i , k , E = G->A/2;
1   Edge a[maxE];
2   GRAPHedges(a, G);
3   sort(a, 0, E-1);
4   UFinit(G->V);
5   for (i = k = 0; i < E && k < G->V-1; i++)
6       if (!UFfind(a[i].v , a[i].w)) {
7           UFunion(a[i].v , a[i].w);
8           mst[k++] = a[i];
        }
9   return k ;
}
```


Coleção disjunta dinâmica

Conjuntos são **modificados ao longo do tempo**

Exemplo: grafo dinâmico



aresta	componentes
(b, c)	$\{a, b, c, d\}$ $\{e, f, g\}$ $\{h, i\}$ $\{j\}$

Operações básicas

$\mathcal{S}$ coleção de conjuntos disjuntos.

Cada conjunto tem um **representante**.

MakeSet: (x): x é elemento novo
$\mathcal{S} \leftarrow \mathcal{S} \cup \{\{x\}\}$

Union: (x, y): x e y em conjuntos diferentes
$\mathcal{S} \leftarrow \mathcal{S} - \{S_x, S_y\} \cup \{S_x \cup S_y\}$
x está em S_x e y está em Sy

FindSet: (x): devolve representante do conjunto que contém x

Conjuntos disjuntos dinâmicos

Seqüência de operações MakeSet, Union, FindSet

M M M U F U U F U F F F U F

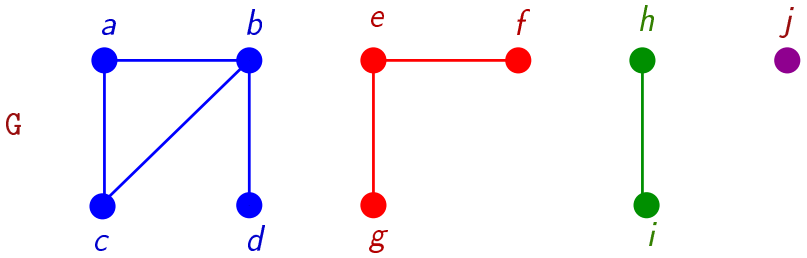
n

m

Que estrutura de dados usar?

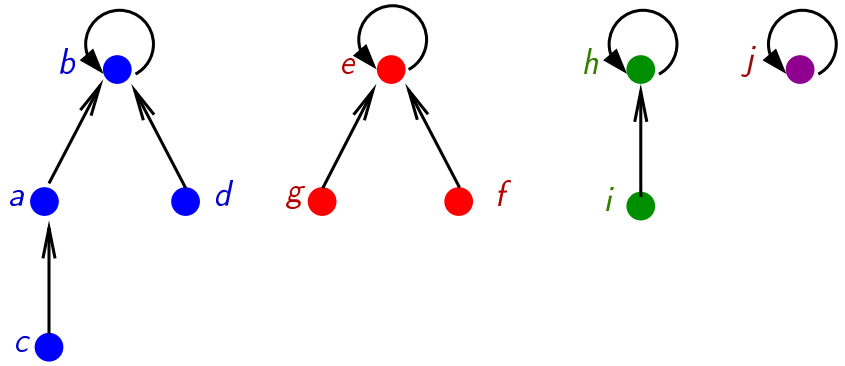
Compromissos (*trade-off*s).

Estrutura *disjoint-set forest*



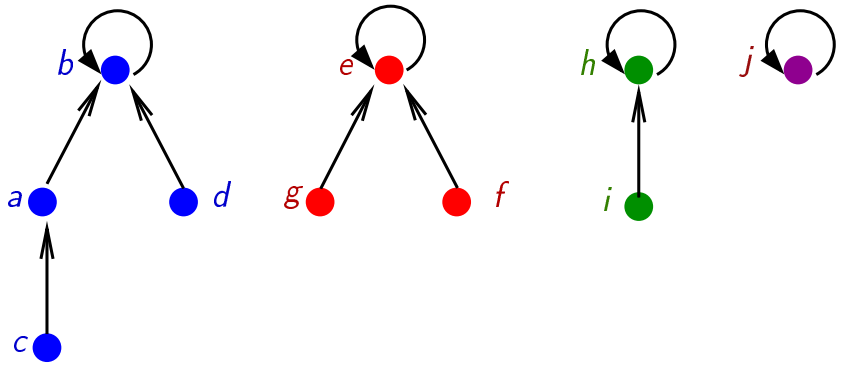
- cada conjunto tem uma *raiz*, que é o seu representate
- cada nó $\mathtt{x}$ tem um *pai*
- $pai[\mathtt{x}] = \mathtt{x}$ se e só se $\mathtt{x}$ é uma raiz

Estrutura *disjoint-set forest*



- cada conjunto tem uma *raiz*
- cada nó $\mathtt{x}$ tem um *pai*
- $pai[\mathtt{x}] = \mathtt{x}$ se e só se $\mathtt{x}$ é uma raiz

MakeSet$_0$ e FindSet$_0$



MAKESET$_0$ $(\mathtt{x})$

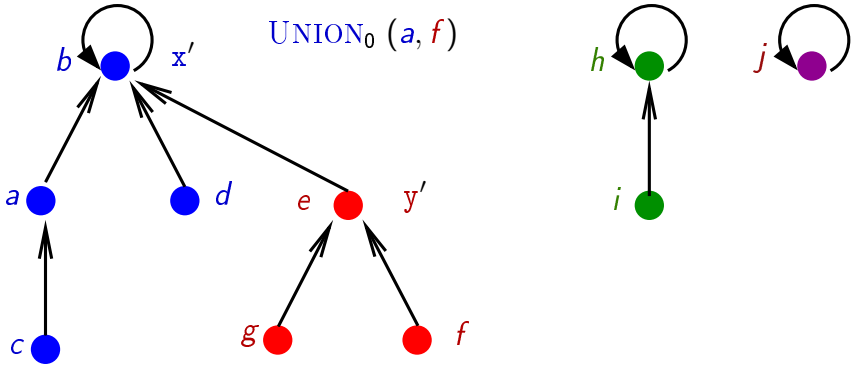
1 $pai[\mathtt{x}] \leftarrow \mathtt{x}$

FINDSET$_0$ $(\mathtt{x})$

1 **enquanto** $pai[\mathtt{x}] \neq \mathtt{x}$ **faça**
2 $\quad \mathtt{x} \leftarrow pai[\mathtt{x}]$
3 **devolva** $\mathtt{x}$

Union$_0$

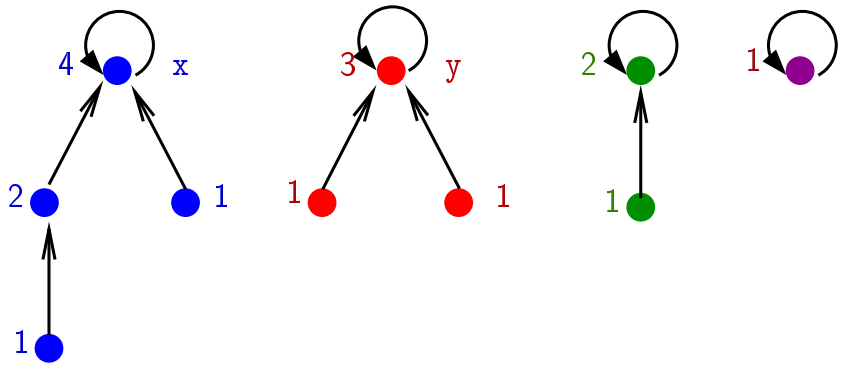
UNION$_0$ (a, f)



UNION$_0$ $(\mathtt{x}, \mathtt{y})$

1 $\mathtt{x}' \leftarrow$ FINDSET$_0$ $(\mathtt{x})$
2 $\mathtt{y}' \leftarrow$ FINDSET$_0$ $(\mathtt{y})$
3 $pai[\mathtt{y}'] \leftarrow \mathtt{x}'$

Melhoramento 1: *union by rank*



rank[x] = no. de nós de x

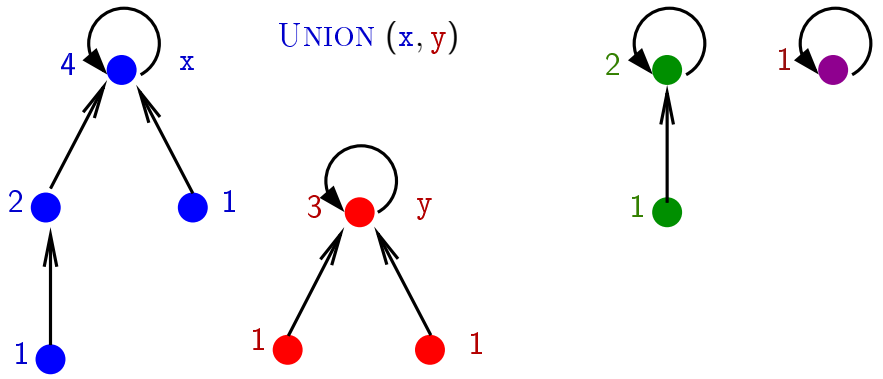
MakeSet (x)

1 $pai[x] \leftarrow x$

2 $\text{rank}[x] \leftarrow 1$

Melhoramento 1: *union by rank*

Union (x, y)



rank[x] = posto de nós de x

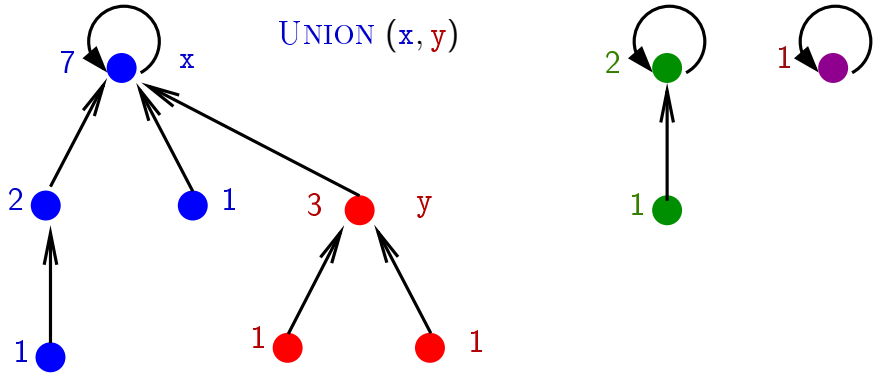
MakeSet (x)

1 $pai[x] \leftarrow x$

2 $\text{rank}[x] \leftarrow 1$

Melhoramento 1: *union by rank*

Union (x, y)



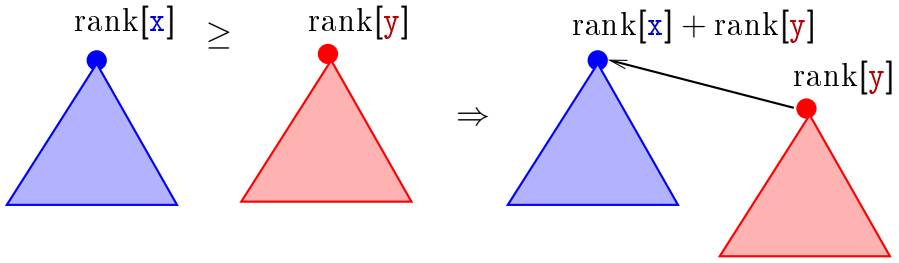
rank[x] = posto de nós de x

MakeSet (x)

1 $pai[x] \leftarrow x$

2 $\text{rank}[x] \leftarrow 1$

Melhoramento 1: *union by rank*



Melhoramento 1: *union by rank*

```
UNION (x, y)  ▷ com "union by rank"
1  x' ← FINDSET (x)
2  y' ← FINDSET (y)    ▷ supõe que x' ≠ y'
3  se rank[x'] > rank[y']
4       então pai[y'] ← x'
5             rank[y'] ← rank[y'] + rank[x']
6       senão pai[x'] ← y'
7             rank[x'] = rank[x'] + rank[y']
```


Consumo de tempo

Se conjuntos disjuntos são representados através de disjoint-set forest com *union by rank*, então uma seqüência de m operações MAKESET, UNION e FINDSET, sendo que n são MAKESET, consome tempo $O(m \lg n)$.

UFinit e UFfind

```
static Vertex cor[maxV];
static int sz[maxV];
void UFinit (int N) {
   Vertex v ;
   for (v = 0; v < N ; v++) {
        cor[v] = v ;
        sz[v] = 1;
   }
}
int UFfind (Vertex v , Vertex w) {
   return (find(v) == find(w));
}
```


find

```
static Vertex find(Vertex v) {
   while (v != cor[v])
        v = cor[v];
   return v ;
}
```

UFunion

```
void UFunion (Vertex v0, Vertex w0) {
   Vertex v = find(v0), w = find(w0);
   if (v == w) return;
   if (sz[v] < sz[w]) {
        cor[v] = w ;
        sz[w] += sz[v];
   }
   else {
        cor[w] = v ;
        sz[v] += sz[w];
   }
}
```


Consumo de tempo

Graças à maneira com duas *union-find trees* são unidas por `UFunion`, a altura de cada union-find tree é limitada por $\lg V$.

Assim, `UFfind` e `UFunion` consomem tempo $O(\lg V)$.

Podemos supor que a função `sort` consome tempo proporcional a $\Theta(E \lg E)$.

O restante do código de `GRAPHmstK` consome tempo proporcional a $O(E \lg V)$.

Conclusão

O consumo de tempo da função `GRAPHmstK` é $O(E \lg V)$.

Algoritmos

função	consumo de tempo	observação
`bruteforcePrim`	$O(V^3)$	alg. de Prim
`GRAPHmstP1`	$O(V^2)$	grafos densos matriz adjacência
`GRAPHmstP1`	$O(E \lg V)$	grafos esparsos listas de adjacência
`bruteforceKruskal`	$O(V^3)$	alg. de Kruskal
`GRAPHmstK`	$O(E \lg V)$	alg. de Kruskal *disjoint-set forest*

Os algoritmos funcionam para arestas com **custos quaisquer**.

Algoritmo de Boruvka

(Interesse histórico: 1926)

S 20.3

Algoritmo de Boruvka