

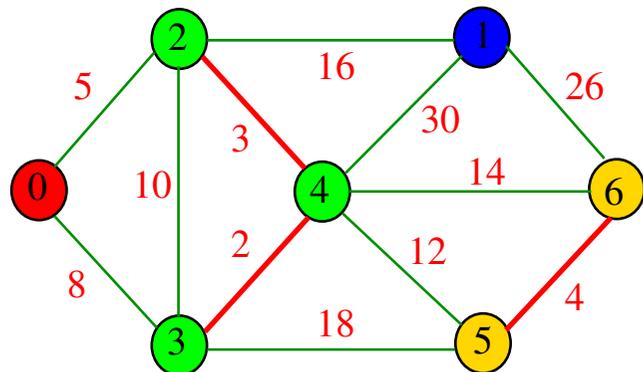
# Algoritmo de Kruskal

S 20.3

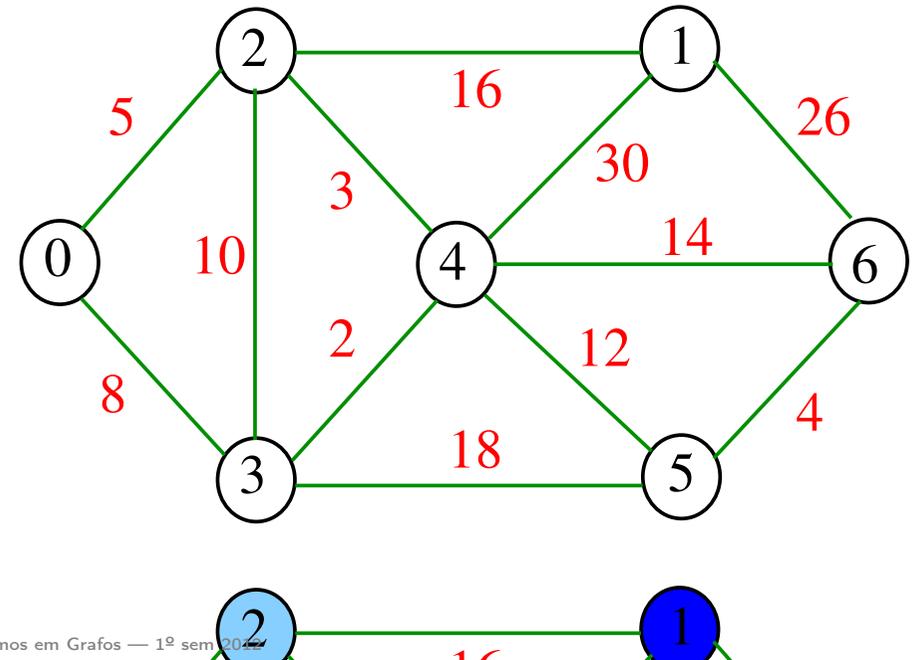
## Subfloresta

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



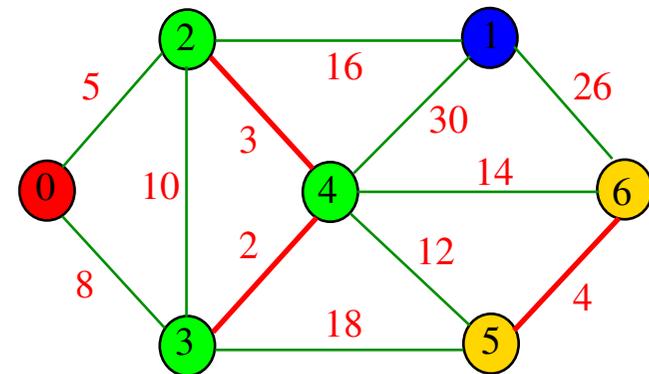
# Algoritmo de Kruskal



## Floresta geradora

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

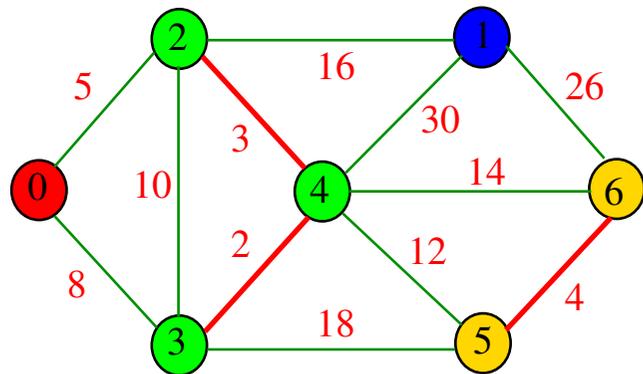
**Exemplo:** Arestas **vermelhas** ligando os vértices verdes e amarelos, junto com os vértices azul e vermelho, forma uma floresta geradora



## 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



## Otimidade

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  $C(F, e)$ .*

## 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:  $F \leftarrow F + e$

## 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;
```

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

```
#define Edge Arc
```

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

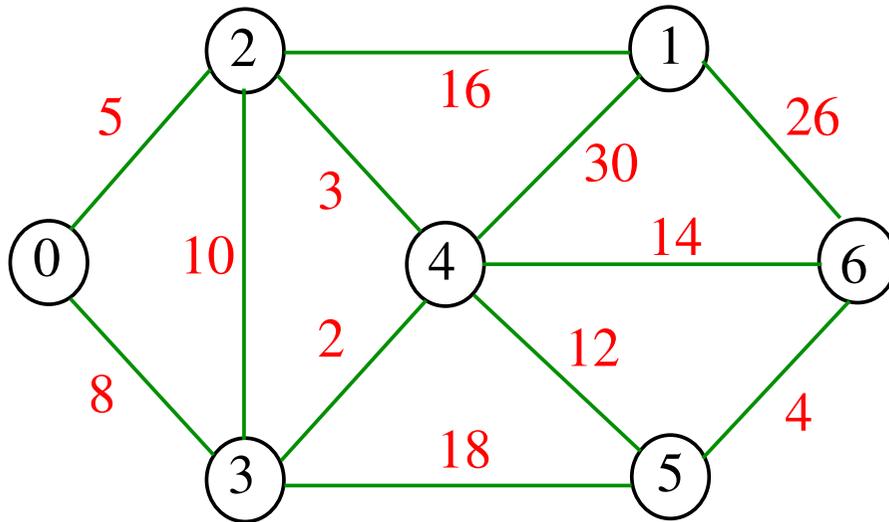
A função armazena as arestas das MSTs no vetor `mst[0..k-1]` e devolve `k`.

```
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]
9            && 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



## 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
```

## 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.

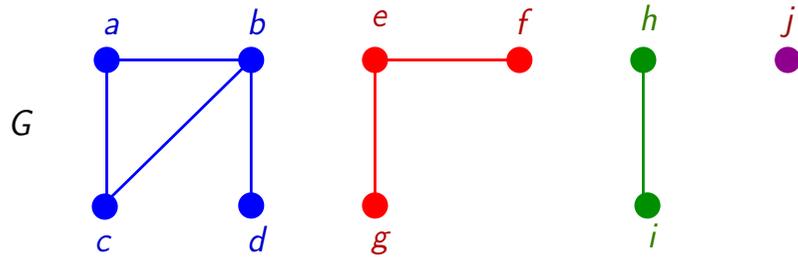
## Kruskal

```
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      }
9  return k;
}
```

## Coleção disjunta dinâmica

Conjuntos são **modificados ao longo do tempo**

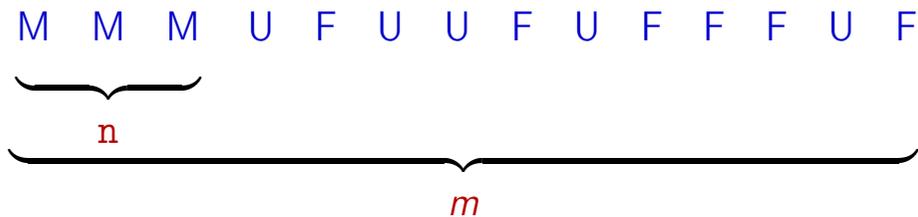
Exemplo: grafo dinâmico



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

## Conjuntos disjuntos dinâmicos

Seqüência de operações MAKESET, UNION, FINDSET



Que estrutura de dados usar?

Compromissos (*trade-offs*).

## Operações básicas

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

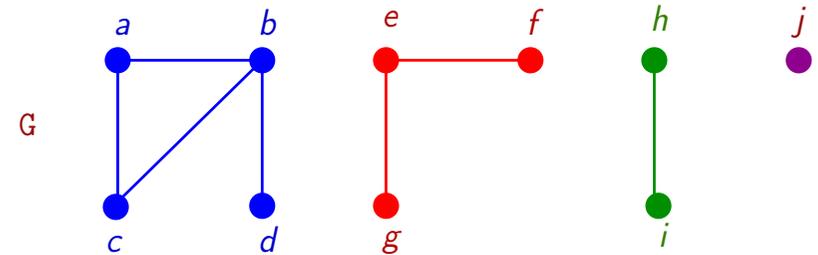
Cada conjunto tem um **representante**.

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  $S_y$

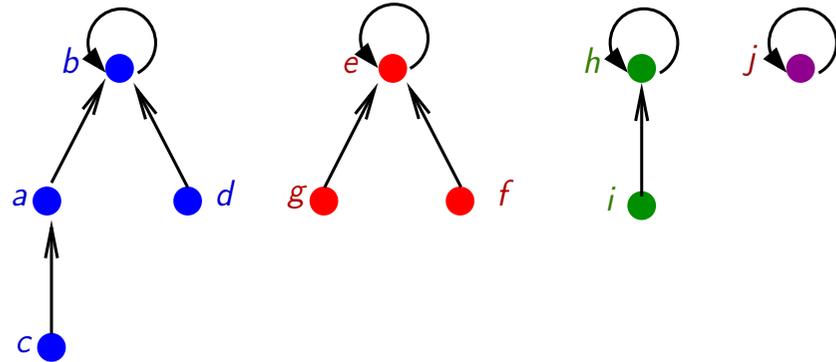
FINDSET: ( $x$ ): devolve representante do conjunto que contém  $x$

## Estrutura *disjoint-set forest*



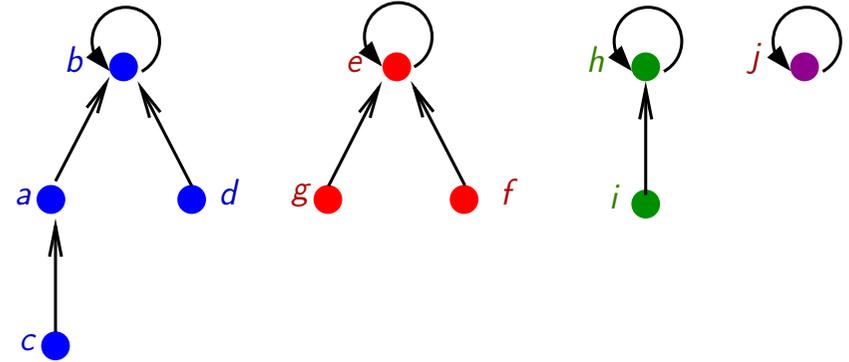
- cada conjunto tem uma *raiz*, que é o seu representante
- cada nó  $x$  tem um *pai*
- $pai[x] = x$  se e só se  $x$  é uma raiz

## Estrutura disjoint-set forest



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

## MakeSet<sub>0</sub> e FindSet<sub>0</sub>



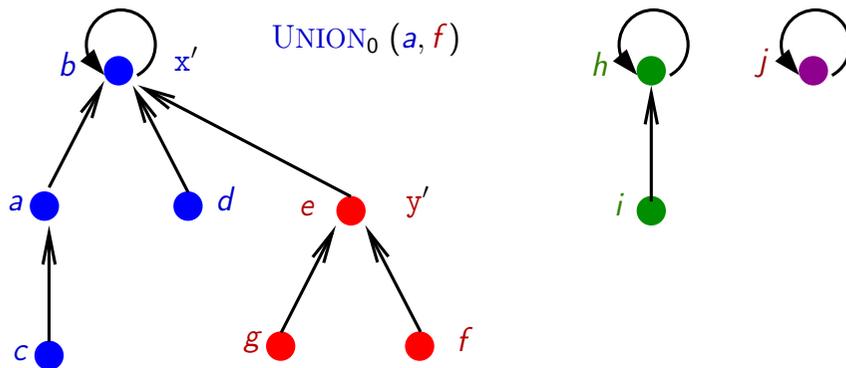
MAKESET<sub>0</sub> ( $x$ )

- 1  $pai[x] \leftarrow x$

FINDSET<sub>0</sub> ( $x$ )

- 1 enquanto  $pai[x] \neq x$  faça
- 2      $x \leftarrow pai[x]$
- 3 devolva  $x$

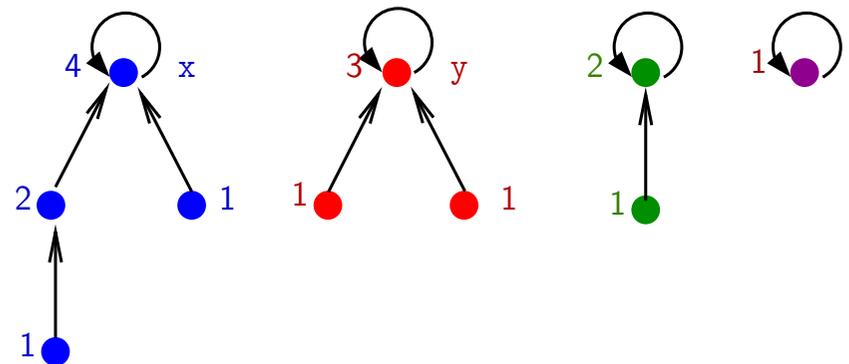
## Union<sub>0</sub>



UNION<sub>0</sub> ( $x, y$ )

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

## Melhoramento 1: union by rank

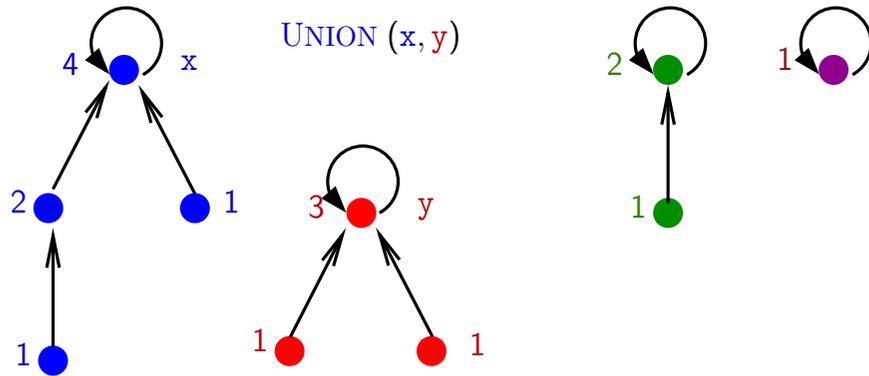


$\text{rank}[x] = \text{no. de nós de } x$

MAKESET ( $x$ )

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

## Melhoramento 1: *union by rank*

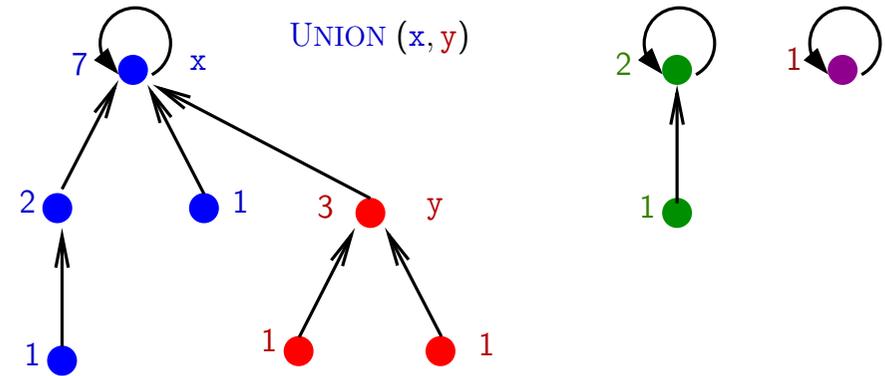


rank[x] = posto de nós  
de x

MAKESET (x)

- 1 pai[x] ← x
- 2 rank[x] ← 1

## Melhoramento 1: *union by rank*

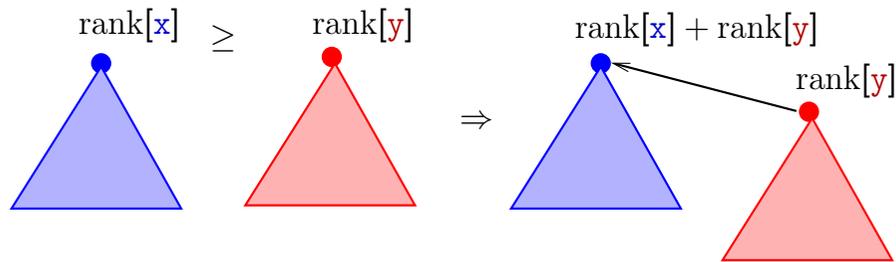


rank[x] = posto de nós  
de x

MAKESET (x)

- 1 pai[x] ← x
- 2 rank[x] ← 1

## Melhoramento 1: *union by rank*



## Melhoramento 1: *union by rank*

UNION (x, y) ▷ com “union by rank”

- 1  $x' \leftarrow \text{FINDSET}(x)$
- 2  $y' \leftarrow \text{FINDSET}(y)$  ▷ supõe que  $x' \neq y'$
- 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']

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  $O(m \lg n)$ .

```
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 UFind (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];
    }
}
```

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

Assim, `UFind` e `UFunction` 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  $O(E \lg V)$ .

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

## Algoritmos

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

Os algoritmos funcionam para arestas com **custos**

## Algoritmo de Boruvka

S 20.3

# Algoritmo de Boruvka

