Melhores momentos

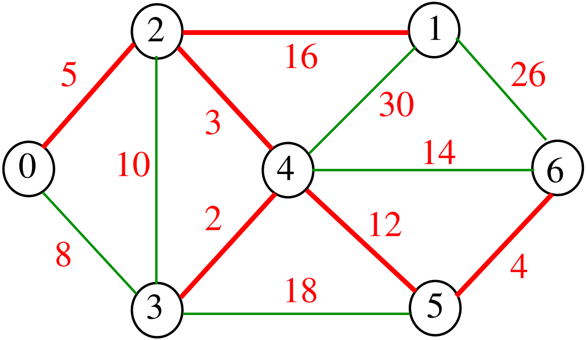
AULA 21

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

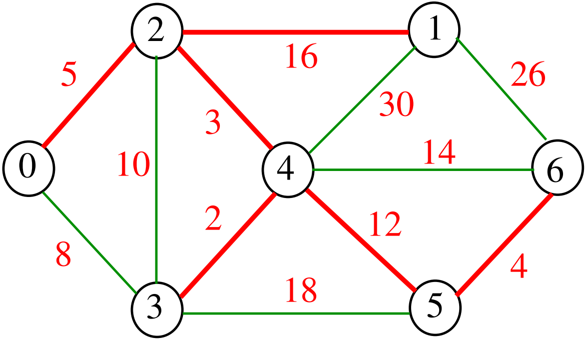
Exemplo: MST de custo 42



Propriedade dos cortes

Condição de Otimalidade: Se T é uma MST então cada aresta t de T é uma aresta mínima dentre as que atravessam o corte determinado por T-t

Exemplo: MST de custo 42



Algoritmo de PRIM

função	consumo de tempo	observação
bruteforcePrim	$O(V^3)$	matriz adjacência
GRAPHmstP1	$O(V^2)$	grafos densos matriz adjacência
GRAPHmstP1	$O(E \lg V)$	grafos esparços listas de adjacência

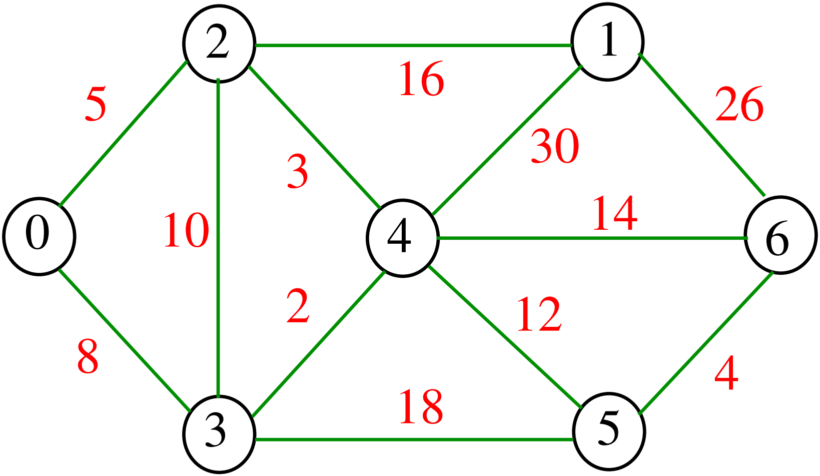
Os algoritmos funcionam para arestas com **custos quaisquer**.

AULA 22

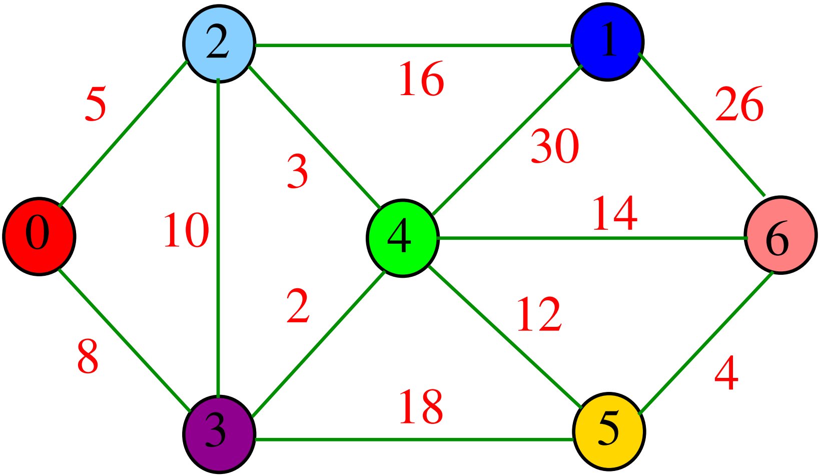
Algoritmo de Kruskal

S 20.3

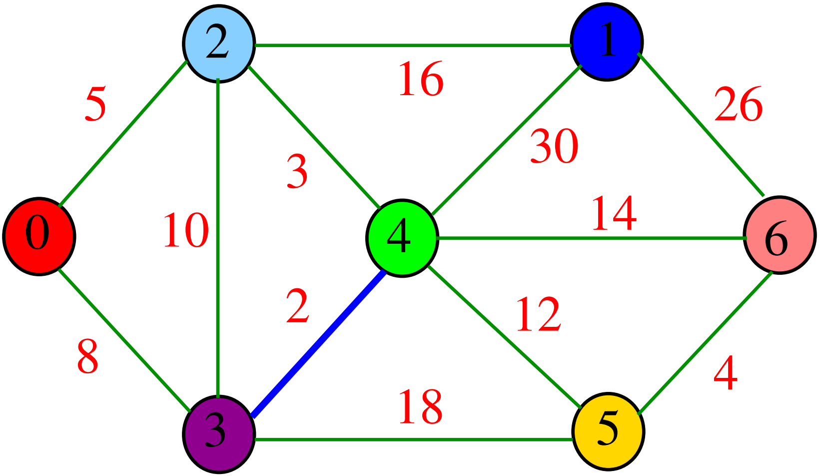
Algoritmo de Kruskal



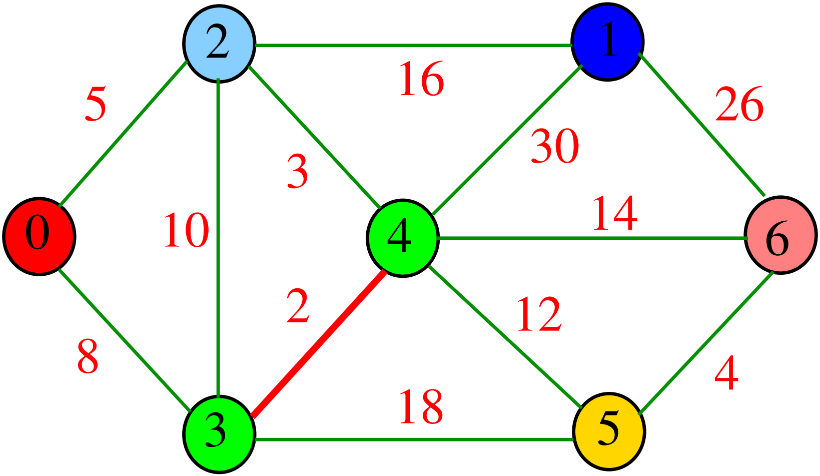
Algoritmo de Kruskal



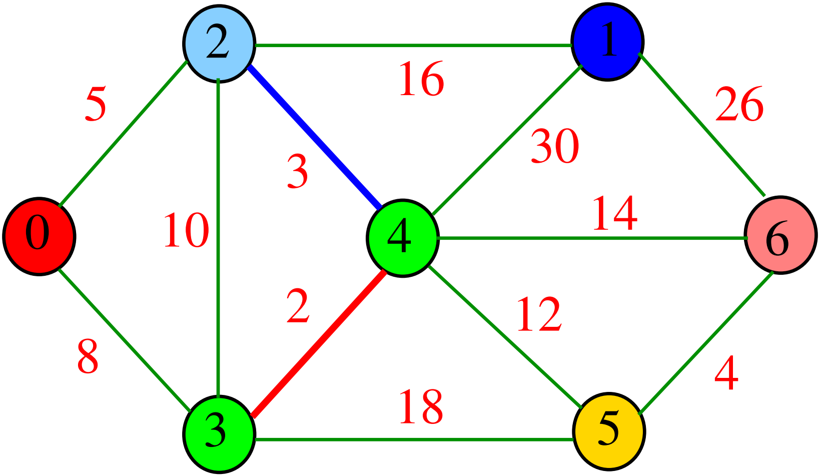
Algoritmo de Kruskal



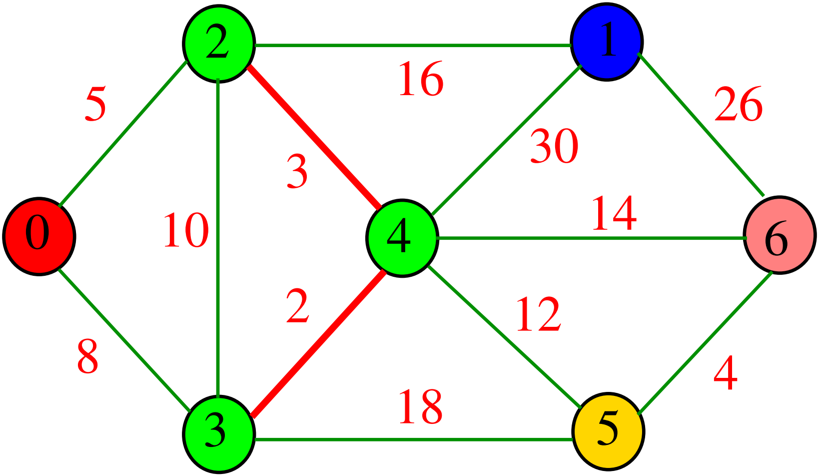
Algoritmo de Kruskal



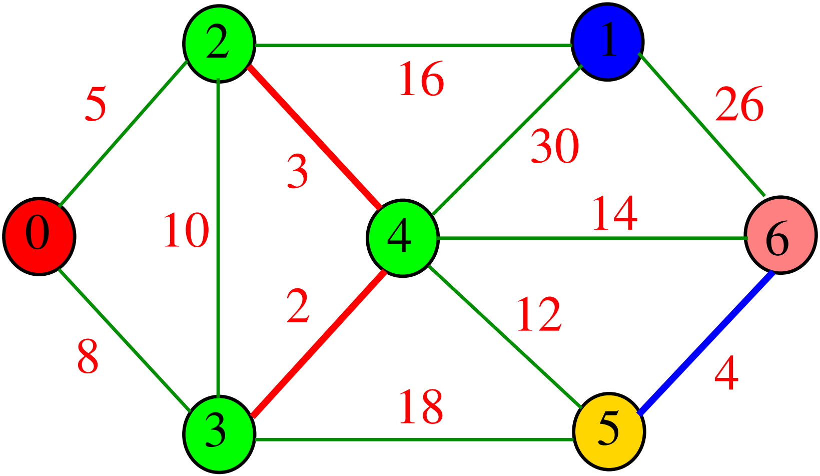
Algoritmo de Kruskal



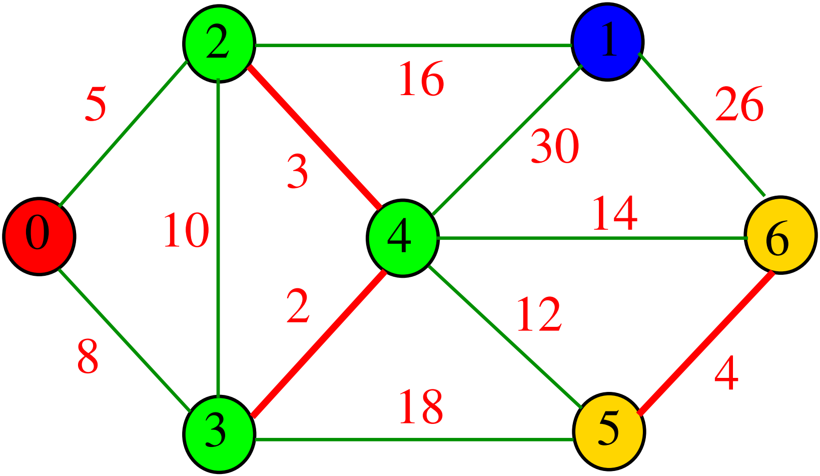
Algoritmo de Kruskal



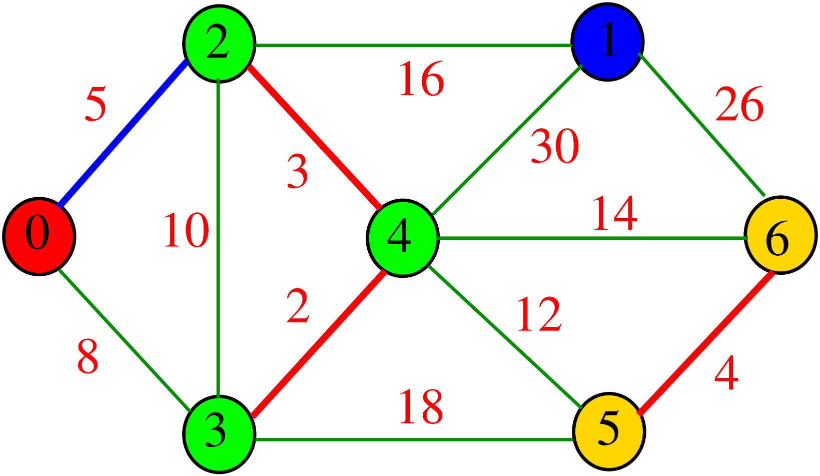
Algoritmo de Kruskal



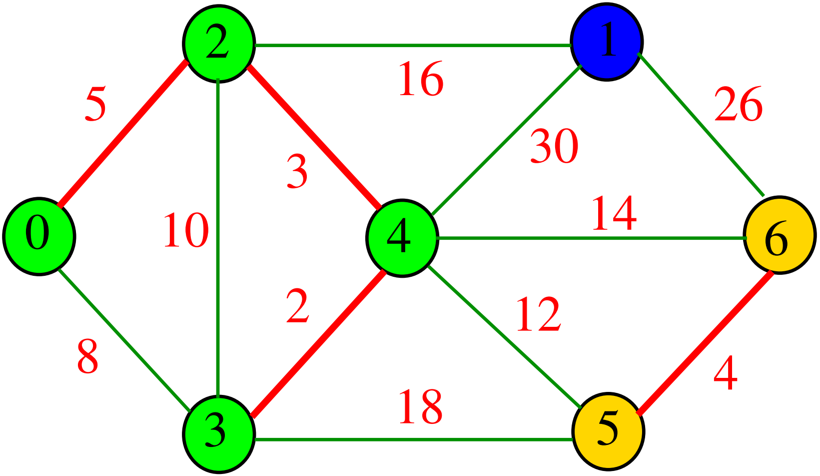
Algoritmo de Kruskal



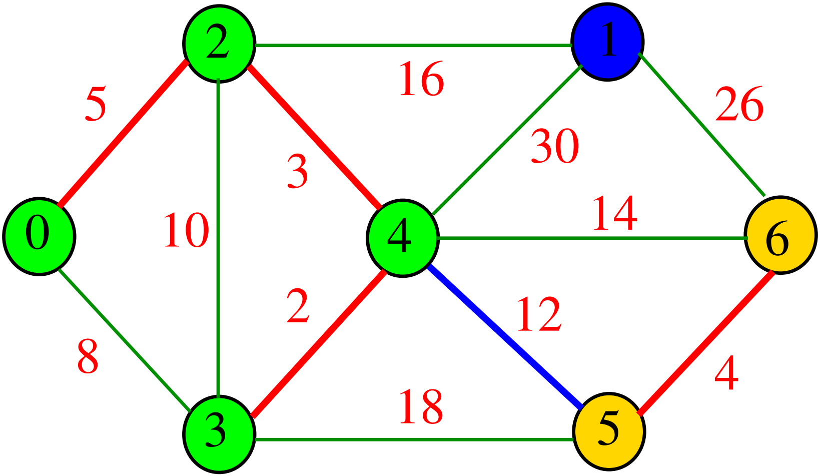
Algoritmo de Kruskal



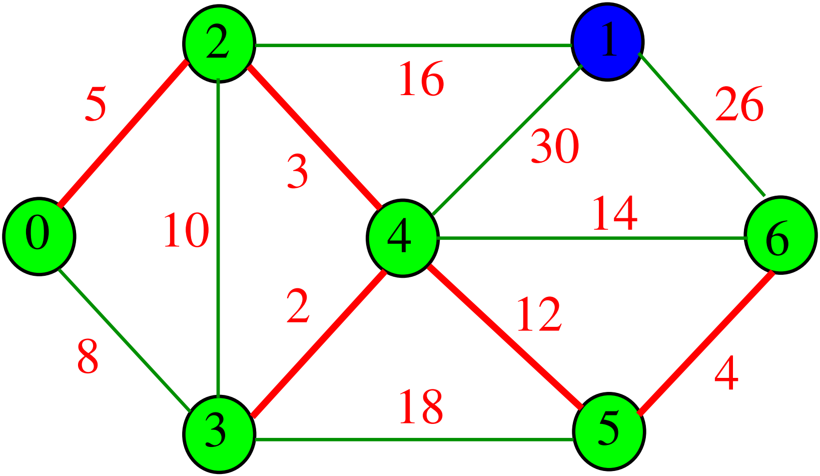
Algoritmo de Kruskal



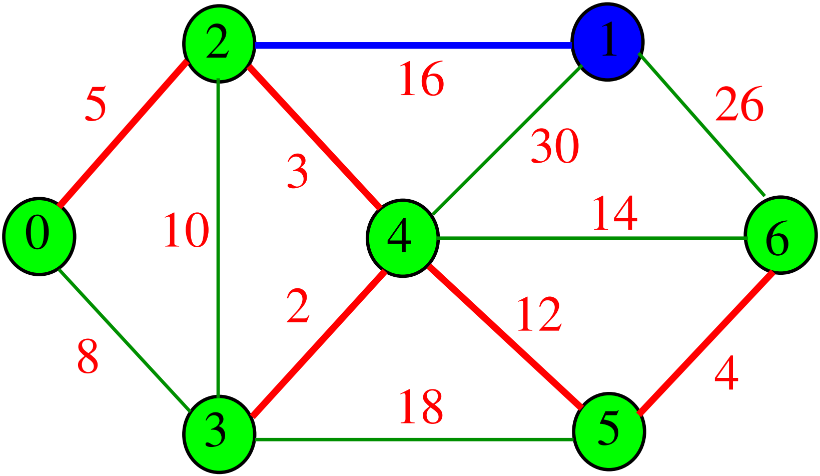
Algoritmo de Kruskal



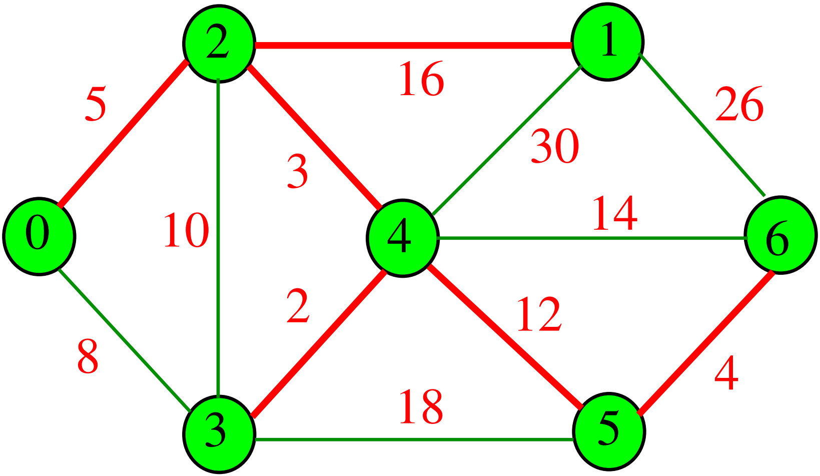
Algoritmo de Kruskal



Algoritmo de Kruskal



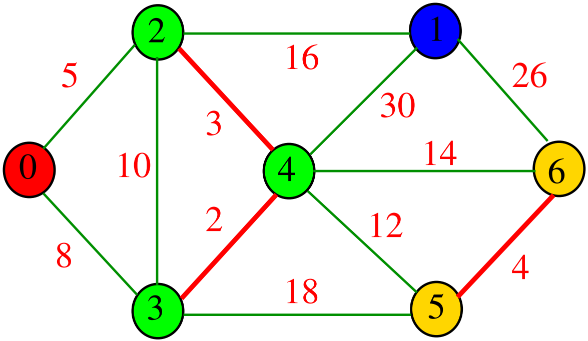
Algoritmo de Kruskal



Subfloresta

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

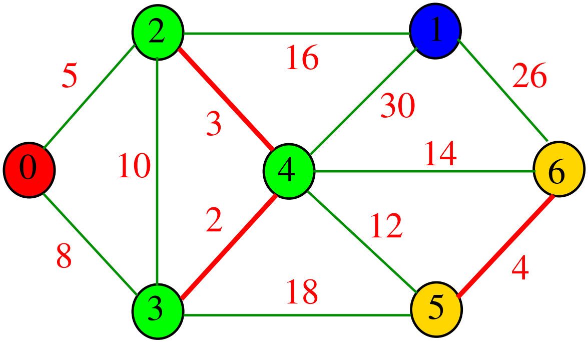
Exemplo: As arestas vermelhas que ligam os vértices verdes e amarelos e formam uma subfloresta



Floresta geradora

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

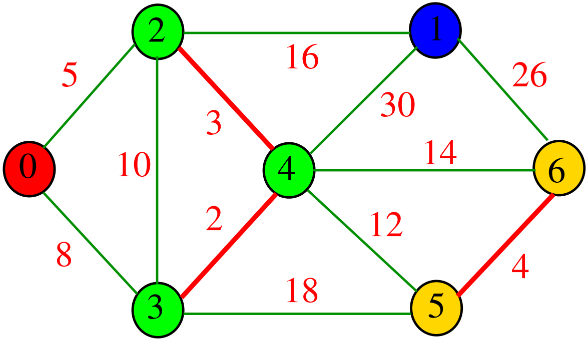
Exemplo: Arestas vermelhas ligando os vértices verdes e amarelos, junto com os vértices azul e vermelho, forma uma floresta geradora



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
Comece nova iteração com F+e no papel de F

Relação invariante chave

No início de cada iteração vale que

existe uma MST que contém as arestas em F.

Se a relação vale no ínicio da última iteração então é evidente que, se o grafo é conexo, o algoritmo devolve uma MST.

Demonstração. Vamos mostrar que se a relação vale no início de uma iteração que não seja a última, então ela vale no fim da iteração com `F+e` no papel de `F`.

A relação invariante certamente vale no início da primeira iteração.

Demonstração

Considere o início de uma iteração qualquer que não seja a última.

Seja $\mathtt{e}$ a aresta escolhida pela iteração no caso 2.

Pela relação invariante existe uma MST $\mathtt{T}^*$ que contém $\mathtt{F}$.

Se $\mathtt{e}$ está em $\mathtt{T}^*$, então não há o que demonstrar.
Suponha, portanto, que $\mathtt{e}$ não está em $\mathtt{T}^*$.

Seja $\mathtt{e}^*$ uma aresta que está no único ciclo em $\mathtt{T}^*{+}\mathtt{e}$ que é externa a $\mathtt{F}$.

Pela escolha de $\mathtt{e}$ temos que $\mathtt{custo}(\mathtt{e}) \leq \mathtt{custo}(\mathtt{e}^*)$.

Portanto, $\mathtt{T}^*{-}\mathtt{e}^*{+}\mathtt{e}$ é uma MST que contém $\mathtt{F}{+}\mathtt{e}$.

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

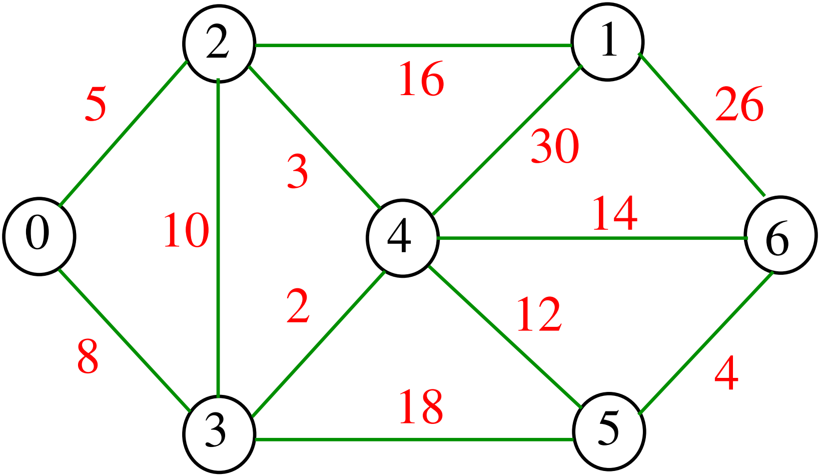
A função armazena as arestas das MSTs no vetor mst[0..k-1] e devolve k.

```
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;
```

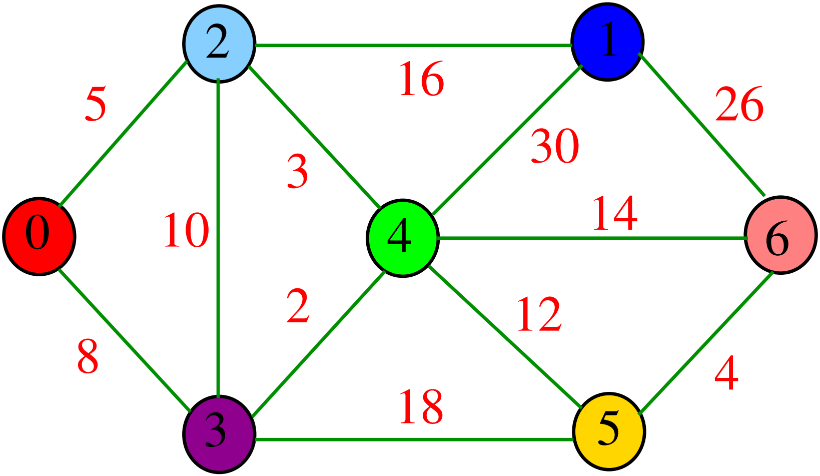
Implementação grosseira

```
while(1) {
double mincst = INFINITO;
for (v = 0; v < G->V; v++)
      for (w = 0; w < G->V; w++)
            if (cor[v] != cor[w]
                  && mincst> G->adj[v][w])
            mincst = G->adj[v0=v][w0=w];
if (mincst == INFINITO) break;
mst[k].v = v0;   mst[k].w = w0;   k++;
for (v = 0; v < G->V; v++)
      if (cor[v] == cor[w0]) cor[v] = cor[v0];
}
return k; }
```

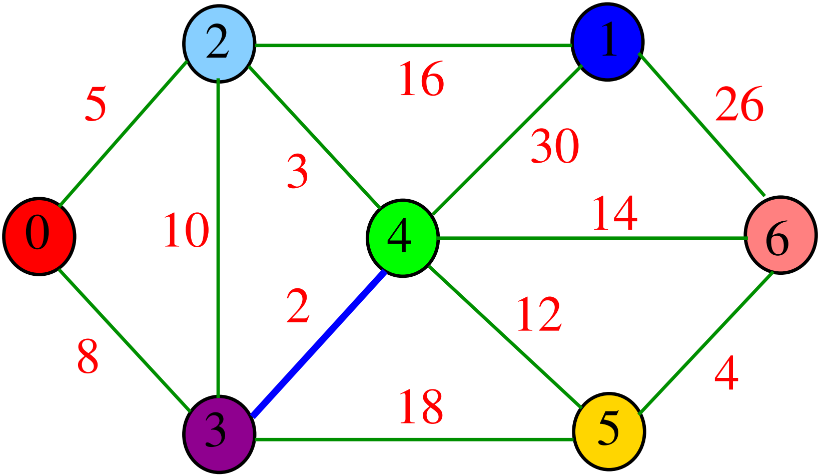
Algoritmo de Kruskal



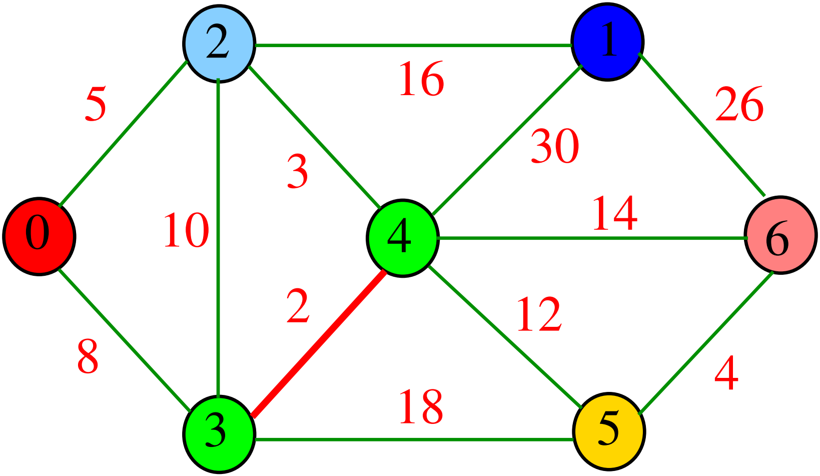
Algoritmo de Kruskal



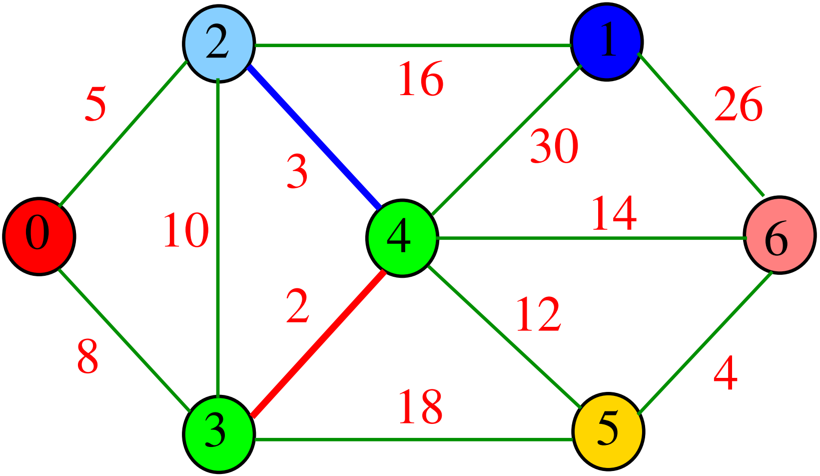
Algoritmo de Kruskal



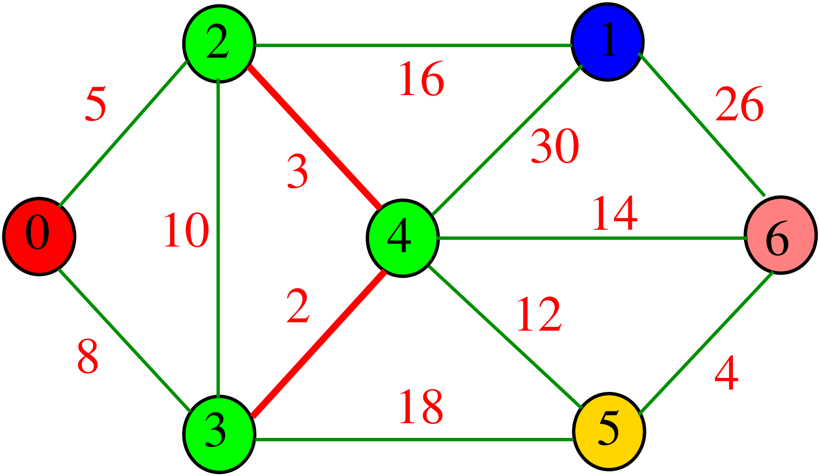
Algoritmo de Kruskal



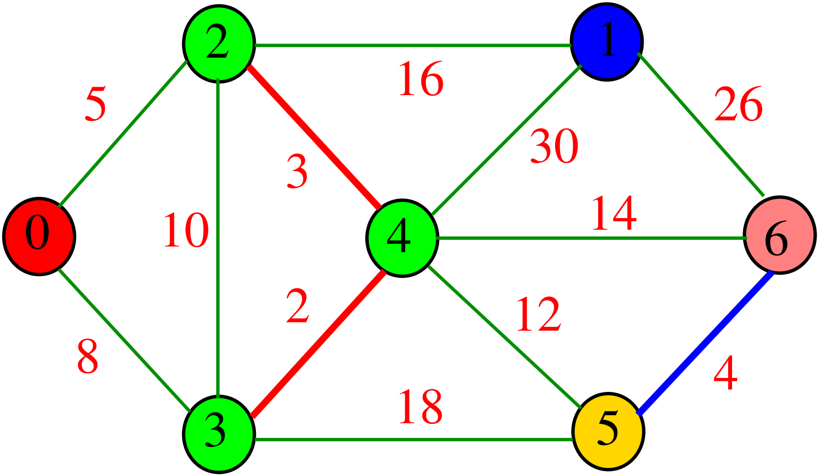
Algoritmo de Kruskal



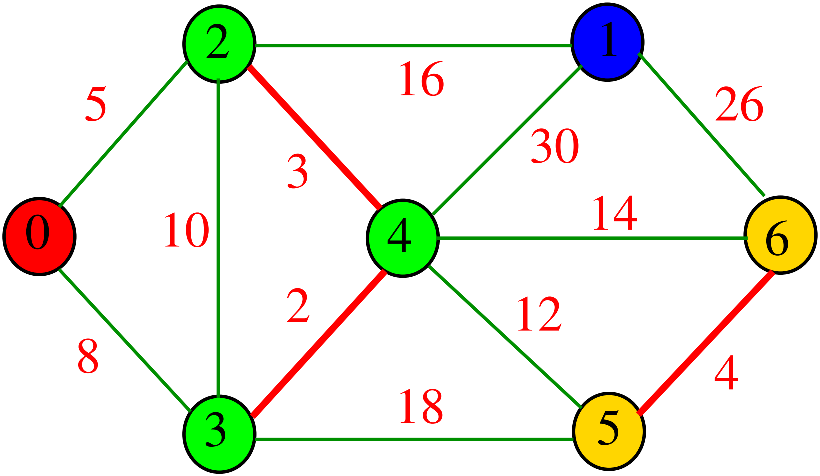
Algoritmo de Kruskal



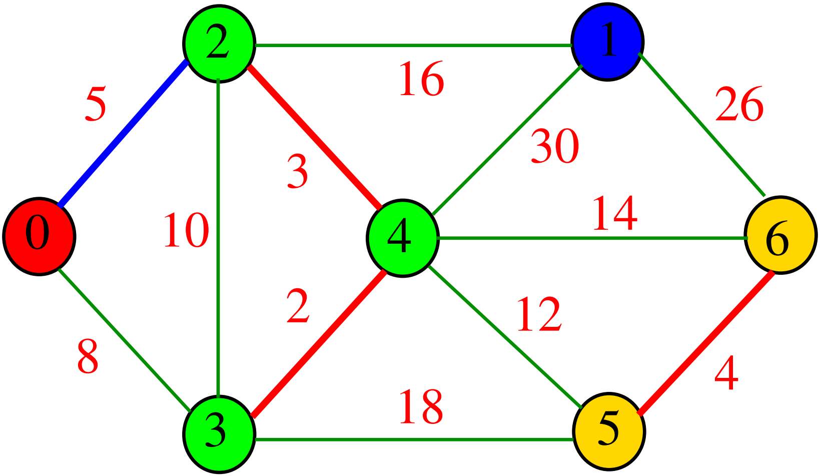
Algoritmo de Kruskal



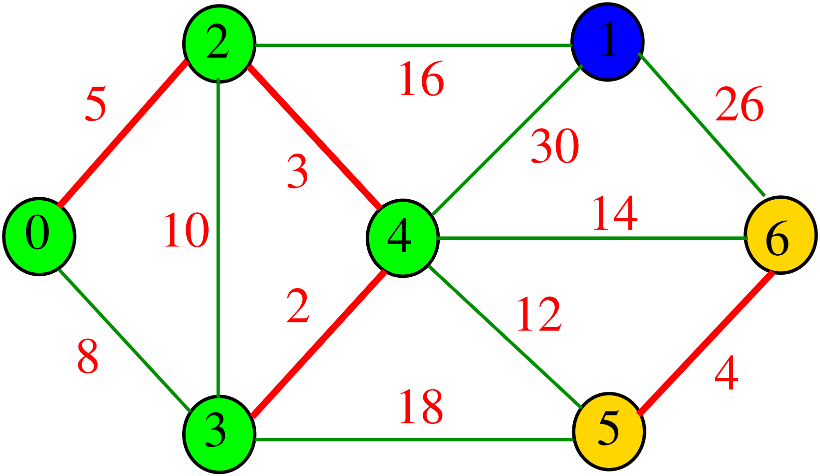
Algoritmo de Kruskal



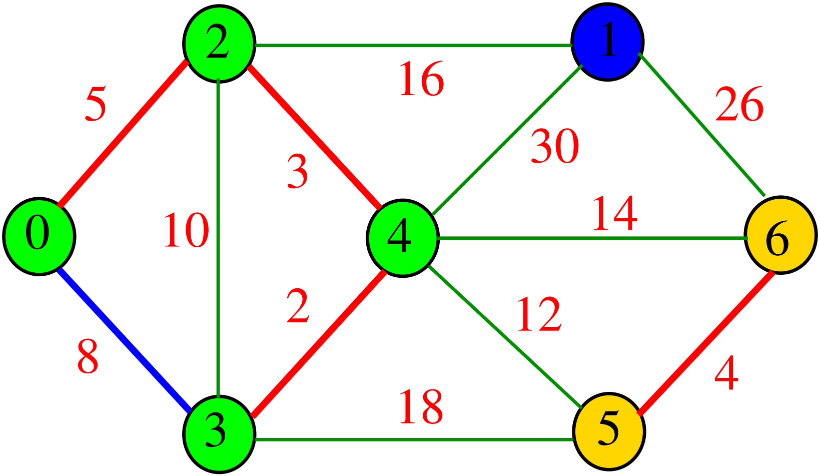
Algoritmo de Kruskal



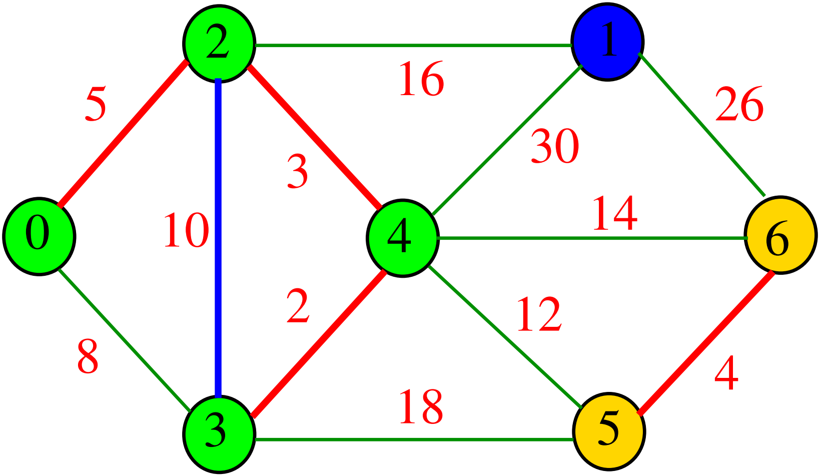
Algoritmo de Kruskal



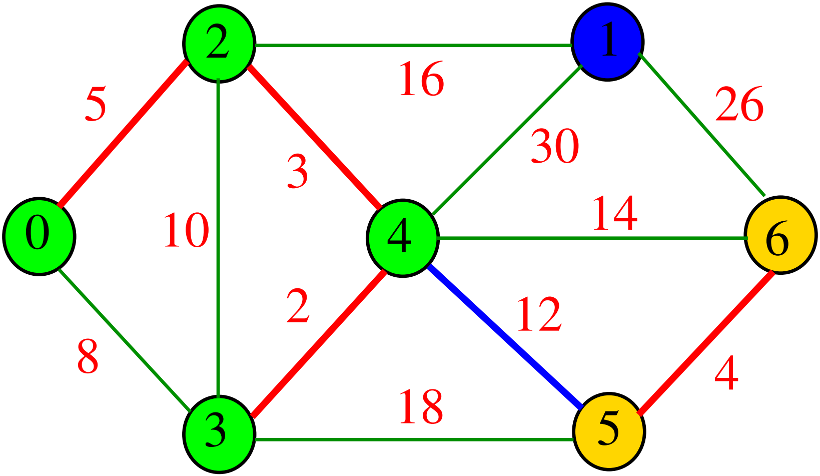
Algoritmo de Kruskal



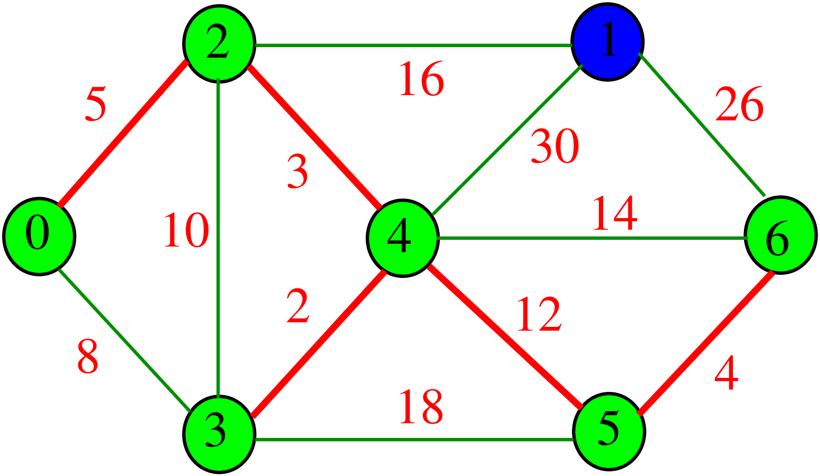
Algoritmo de Kruskal



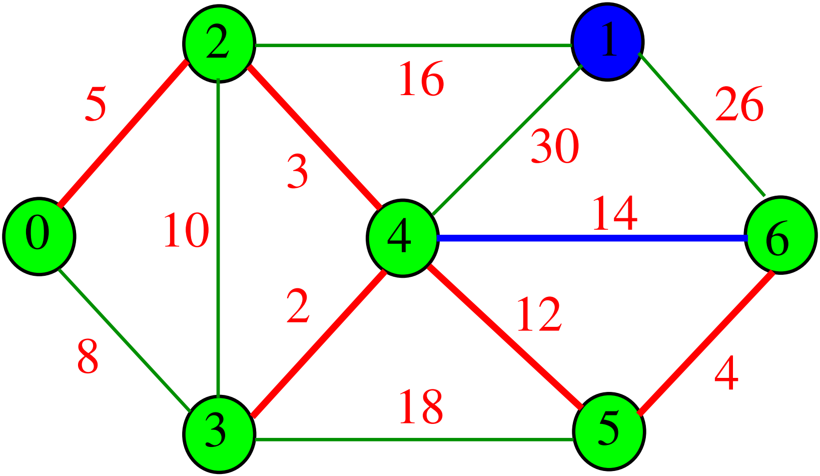
Algoritmo de Kruskal



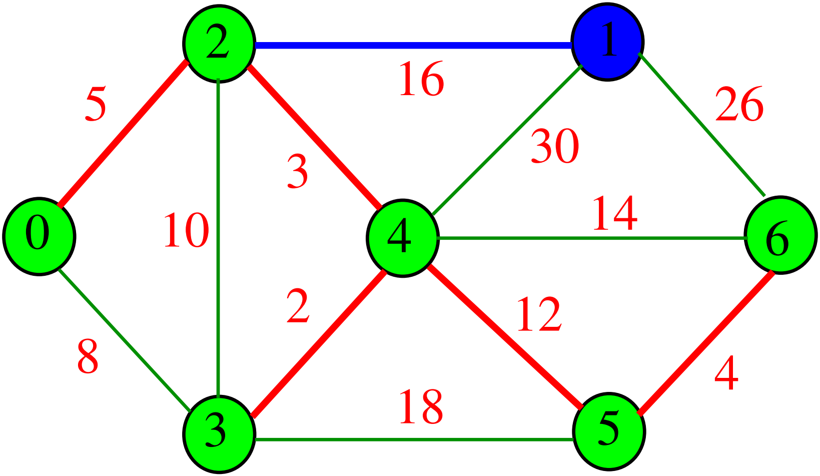
Algoritmo de Kruskal



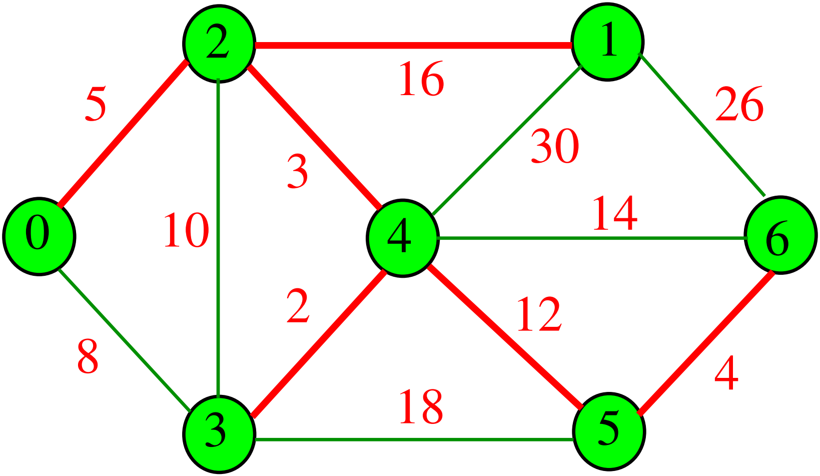
Algoritmo de Kruskal



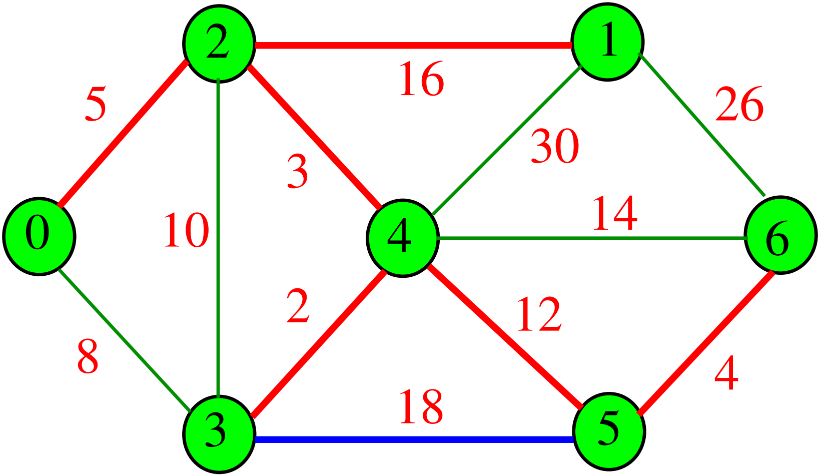
Algoritmo de Kruskal



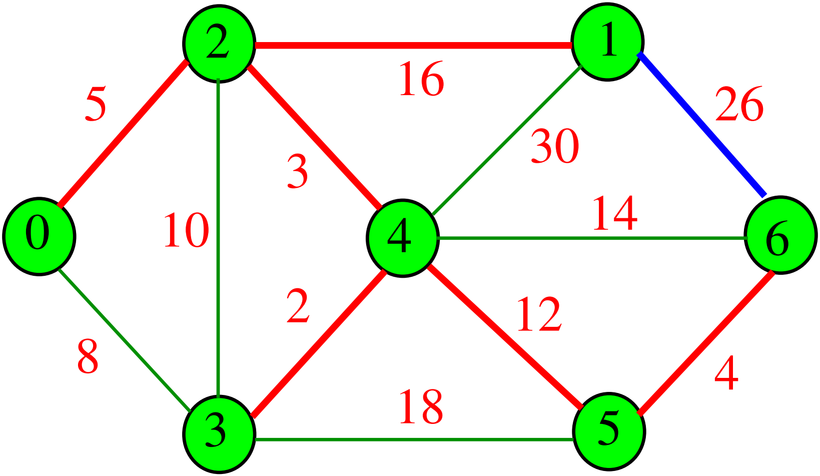
Algoritmo de Kruskal



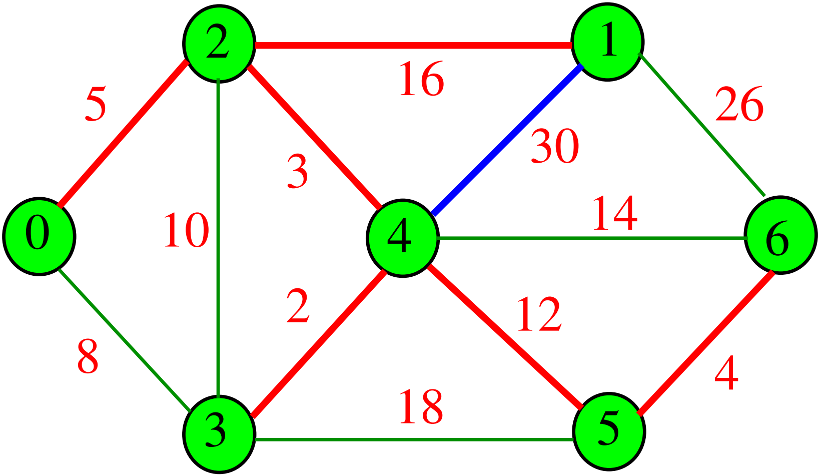
Algoritmo de Kruskal



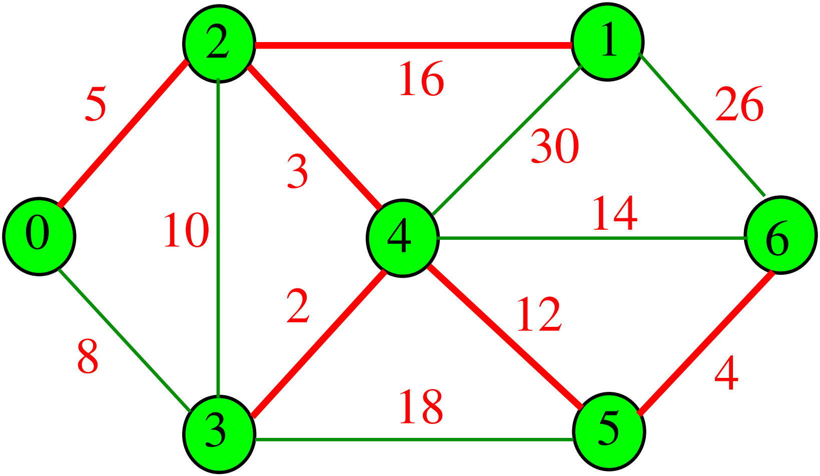
Algoritmo de Kruskal



Algoritmo de Kruskal



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

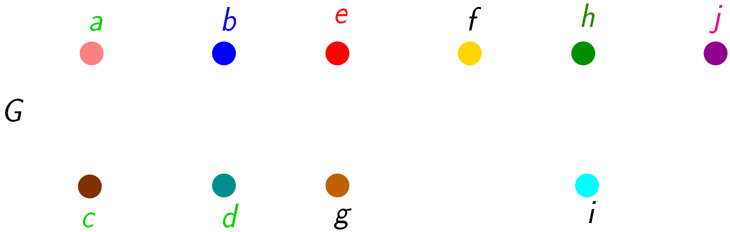
Coleção disjunta dinâmica

Conjuntos são **modificados ao longo do tempo**

Coleção disjunta dinâmica

Conjuntos são **modificados ao longo do tempo**

Exemplo: grafo dinâmico

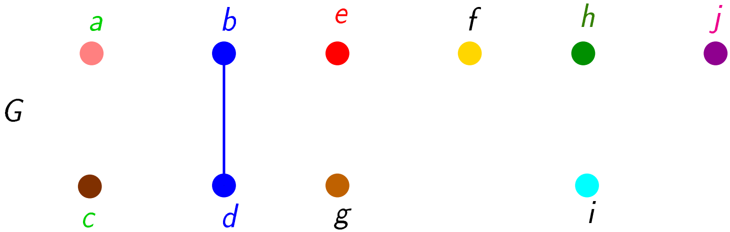


aresta	componentes
	$\{a\}$ $\{b\}$ $\{c\}$ $\{d\}$ $\{e\}$ $\{f\}$ $\{g\}$ $\{h\}$ $\{i\}$

Coleção disjunta dinâmica

Conjuntos são **modificados ao longo do tempo**

Exemplo: grafo dinâmico

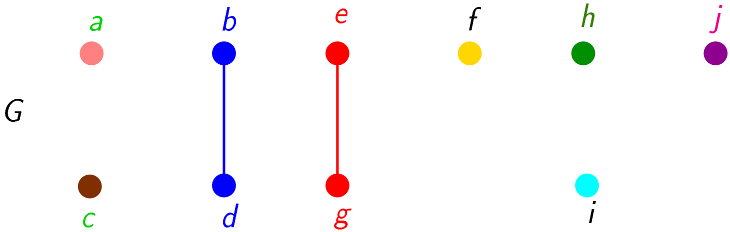


aresta	componentes
(b, d)	$\{a\}$ $\{b, d\}$ $\{c\}$ $\{e\}$ $\{f\}$ $\{g\}$ $\{h\}$ $\{i\}$ $\{j\}$

Coleção disjunta dinâmica

Conjuntos são **modificados ao longo do tempo**

Exemplo: grafo dinâmico

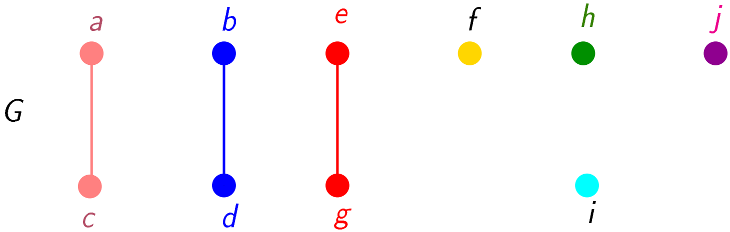


aresta	componentes
(e, g)	$\{a\}$ $\{b, d\}$ $\{c\}$ $\{e, g\}$ $\{f\}$ $\{h\}$ $\{i\}$ $\{j\}$

Coleção disjunta dinâmica

Conjuntos são **modificados ao longo do tempo**

Exemplo: grafo dinâmico

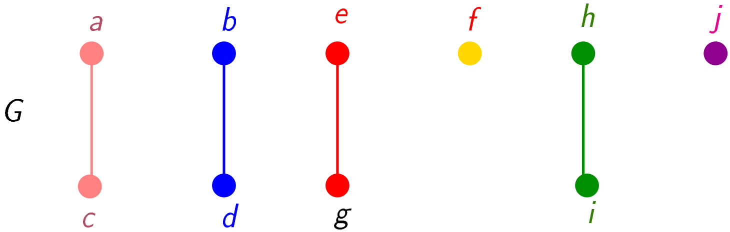


aresta	componentes
(a, c)	$\{a, c\}$ $\{b, d\}$ $\{e, g\}$ $\{f\}$ $\{h\}$ $\{i\}$ $\{j\}$

Coleção disjunta dinâmica

Conjuntos são **modificados ao longo do tempo**

Exemplo: grafo dinâmico

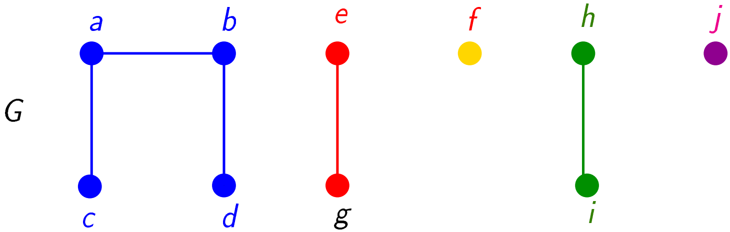


aresta	componentes
(h,i)	$\{a,c\}$ $\{b,d\}$ $\{e,g\}$ $\{f\}$ $\{h,i\}$ $\{j\}$

Coleção disjunta dinâmica

Conjuntos são **modificados ao longo do tempo**

Exemplo: grafo dinâmico

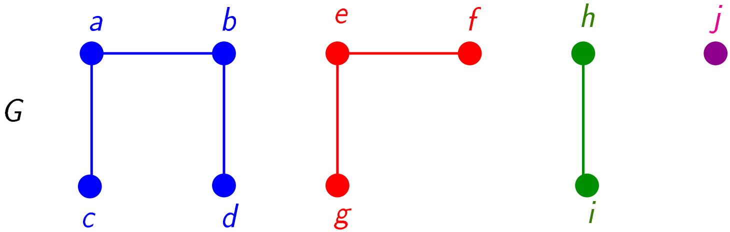


aresta	componentes
(a, b)	$\{a, b, c, d\}$ $\{e, g\}$ $\{f\}$ $\{h, i\}$ $\{j\}$

Coleção disjunta dinâmica

Conjuntos são **modificados ao longo do tempo**

Exemplo: grafo dinâmico

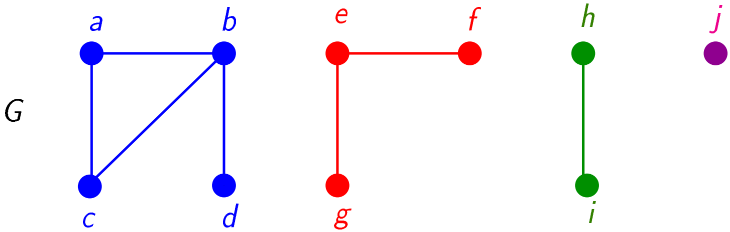


aresta	componentes
(e, f)	$\{a, b, c, d\}$ $\{e, f, g\}$ $\{h, i\}$ $\{j\}$

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

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

FindSet $(\mathtt{x})$: devolve representante do conjunto que contém $\mathtt{x}$

Conjuntos disjuntos dinâmicos

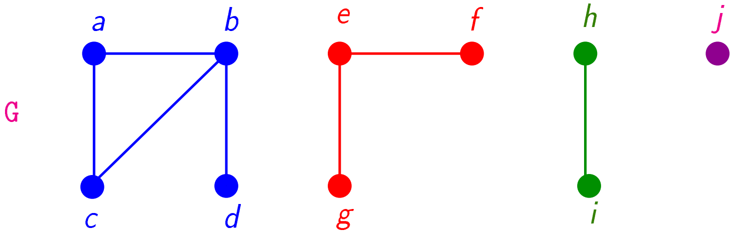
Seqüência de operações MakeSet, Union, FindSet

$$\underbrace{\underbrace{\text{M M M}}_{\texttt{n}}\ \text{U F U U F U F F F U F}}_{m}$$

Que estrutura de dados usar?

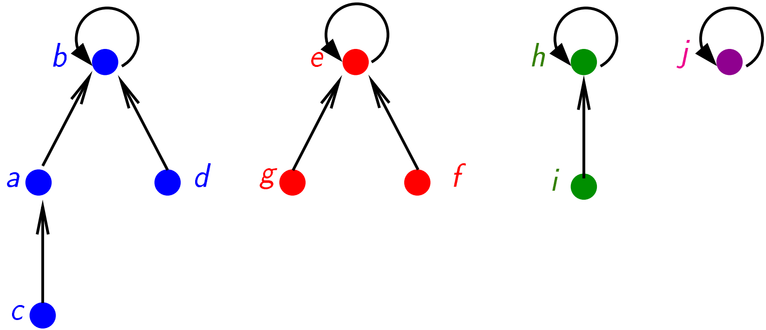
Compromissos (*trade-off* s).

Estrutura *disjoint-set forest*



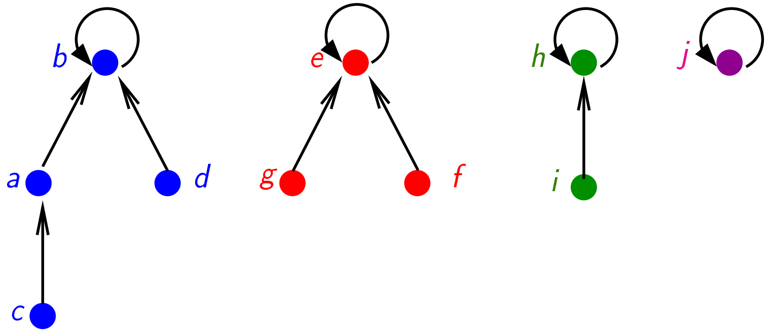
- cada conjunto tem uma *raiz*, que é o seu representate
- cada nó $\mathtt{x}$ tem um *pai*
- $pai[\mathtt{x}] = \mathtt{x}$ se e só se $\mathtt{x}$ é uma raiz

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

$\quad$ *pai*$[\mathtt{x}] \leftarrow \mathtt{x}$

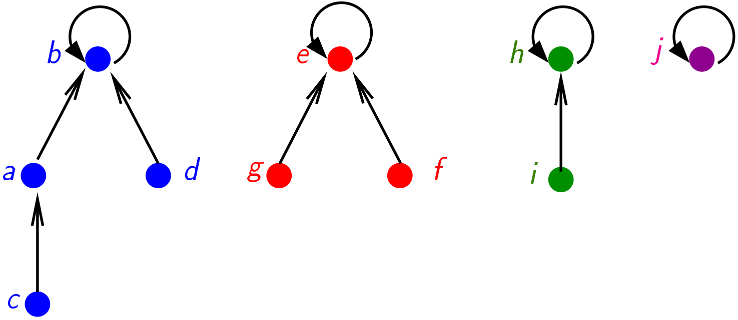
FINDSET$_0$ $(\mathtt{x})$

$\quad$ **enquanto** *pai*$[\mathtt{x}] \neq \mathtt{x}$ **faça**

$\quad\quad$ $\mathtt{x} \leftarrow$ *pai*$[\mathtt{x}]$

$\quad$ **devolva** $\mathtt{x}$

Union_0



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

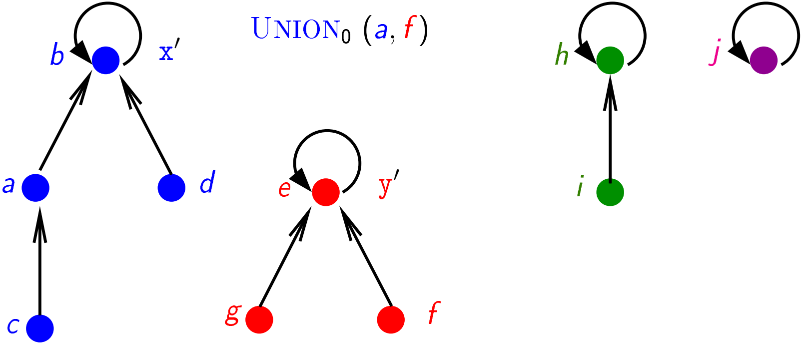
1 $\quad x' \leftarrow \text{FINDSET}_0\,(x)$

2 $\quad y' \leftarrow \text{FINDSET}_0\,(y)$

3 $\quad \mathit{pai}[y'] \leftarrow x'$

Union_0

$\text{UNION}_0\,(a, f)$



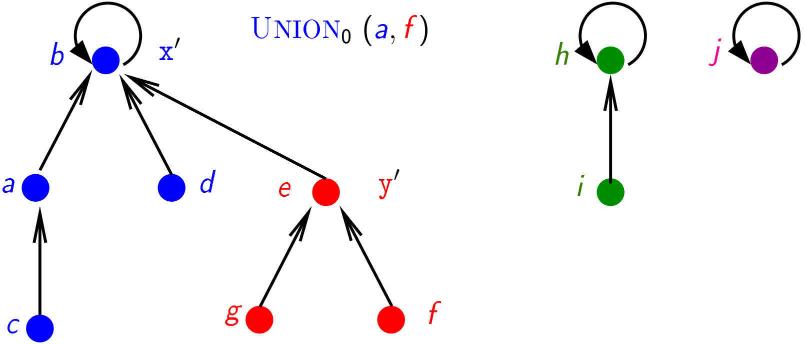
$\text{UNION}_0\,(\mathrm{x}, \mathrm{y})$

$\mathrm{x}' \leftarrow \text{FINDSET}_0\,(\mathrm{x})$

$\mathrm{y}' \leftarrow \text{FINDSET}_0\,(\mathrm{y})$

$pai[\mathrm{y}'] \leftarrow \mathrm{x}'$

Union$_0$



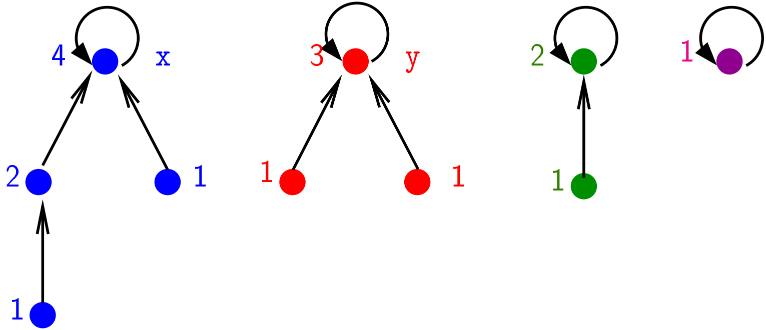
UNION$_0$ (x, y)

$x' \leftarrow$ FINDSET$_0$ (x)

$y' \leftarrow$ FINDSET$_0$ (y)

$pai[y'] \leftarrow x'$

Melhoramento 1: *union by rank*



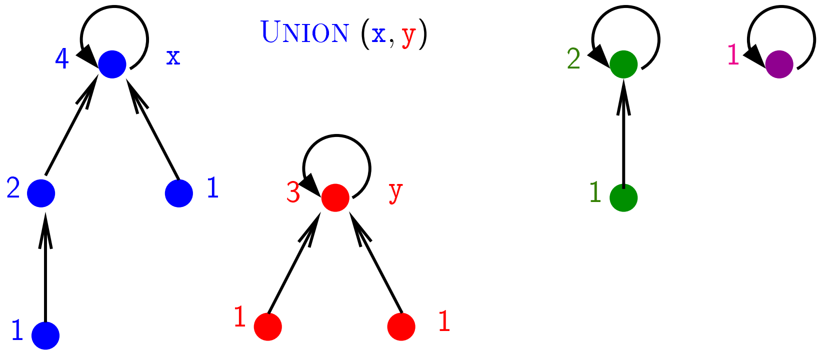
sz[x] = no. de nós de x

MAKESET (x)

1 *pai*[x] ← x

2 sz[x] ← 1

Melhoramento 1: *union by rank*



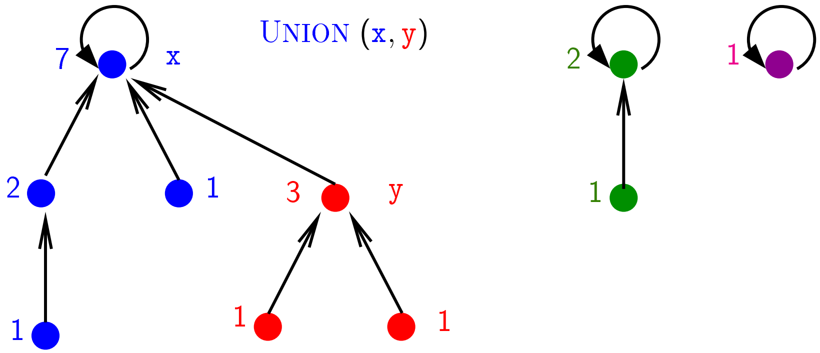
$\mathtt{sz}[\mathtt{x}]$ = posto de nós de $\mathtt{x}$

MAKESET ($\mathtt{x}$)

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

2 $\quad \mathtt{sz}[\mathtt{x}] \leftarrow 1$

Melhoramento 1: *union by rank*



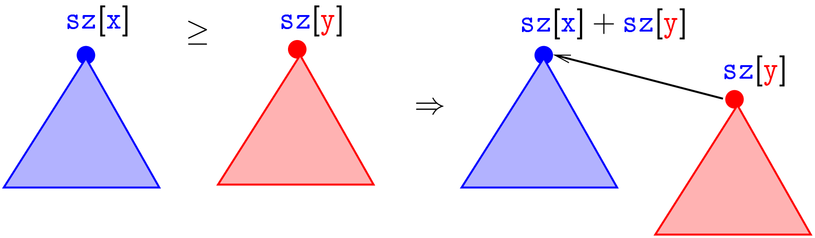
$\text{sz}[\text{x}]$ = posto de nós de x

MAKESET (x)

1 $\quad pai[\text{x}] \leftarrow \text{x}$

2 $\quad \text{sz}[\text{x}] \leftarrow 1$

Melhoramento 1: *union by rank*



Melhoramento 1: *union by rank*

UNION (x, y) $\triangleright$ com “union by rank”

$x' \leftarrow$ FINDSET (x)

$y' \leftarrow$ FINDSET (y) $\triangleright$ supõe que $x' \neq y'$

se $sz[x'] > sz[y']$

  então $pai[y'] \leftarrow x'$

    $sz[y'] \leftarrow sz[y'] + sz[x']$

  senão $pai[x'] \leftarrow y'$

    $sz[x'] = sz[x'] + sz[y']$

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 $\mathtt{n}$ são MAKESET, consome tempo $O(m \lg \mathtt{n})$.

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 $\lg \mathtt{V}$.

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

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

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

Conclusão

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

Algoritmos

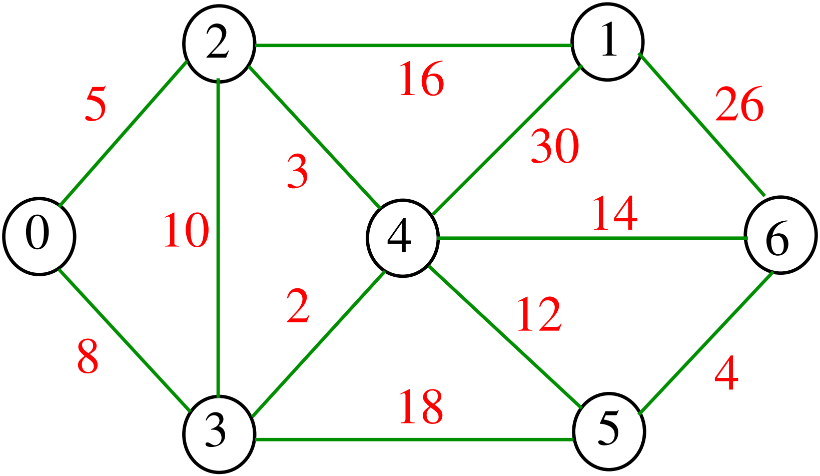
função	consumo de tempo	observação
bruteforcePrim	$O(V^3)$	alg. de Prim
GRAPHmstP1	$O(V^2)$	grafos densos matriz adjacência
GRAPHmstP1	$O(E \lg V)$	grafos esparços listas de adjacência
bruteforceKruskal	$O(V^3)$	alg. de Kruskal
GRAPHmstK	$O(E \lg V)$	alg. de Kruskal *disjoint-set forest*

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

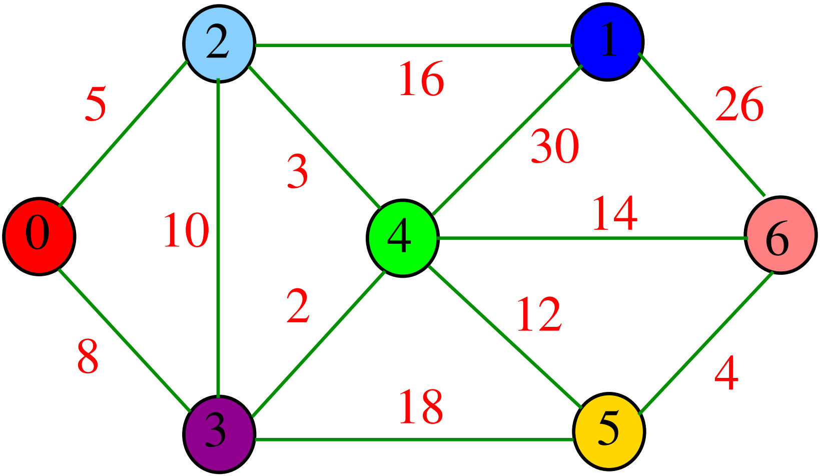
Algoritmo de Boruvka

S 20.3

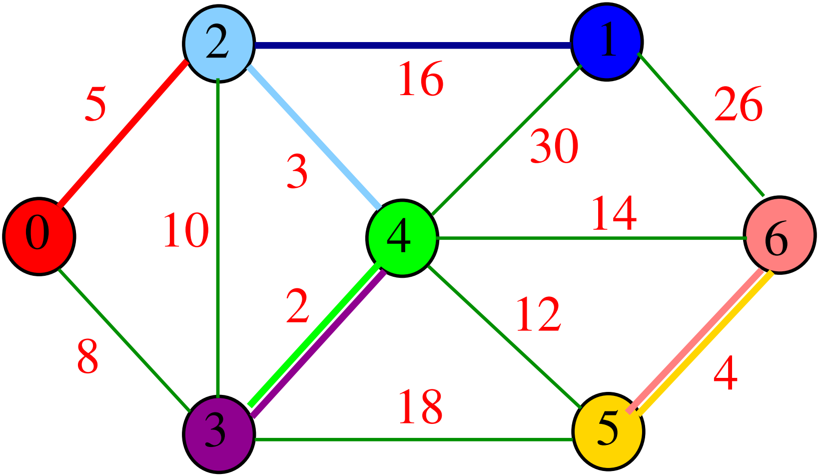
Algoritmo de Boruvka



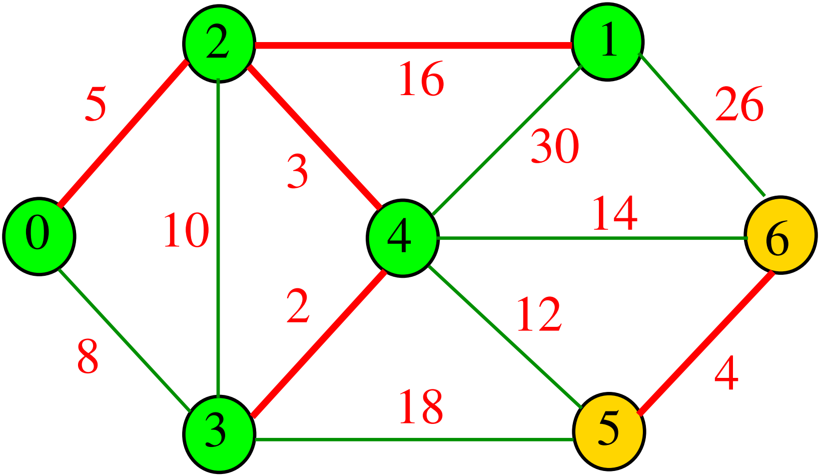
Algoritmo de Boruvka



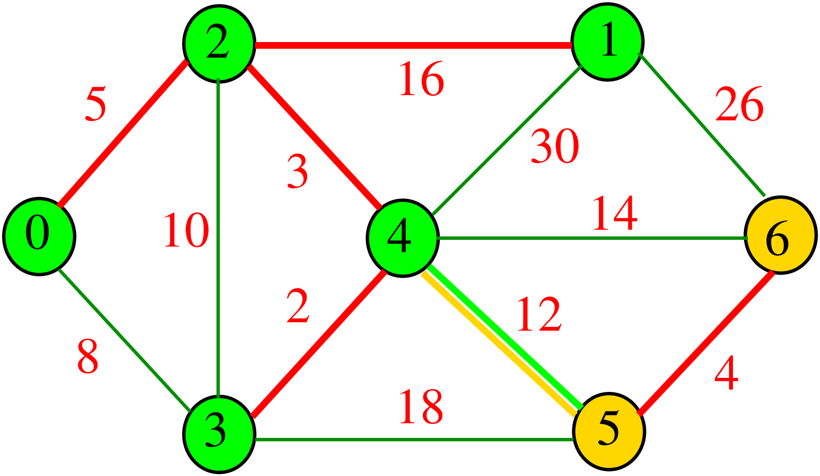
Algoritmo de Boruvka



Algoritmo de Boruvka



Algoritmo de Boruvka



Algoritmo de Boruvka