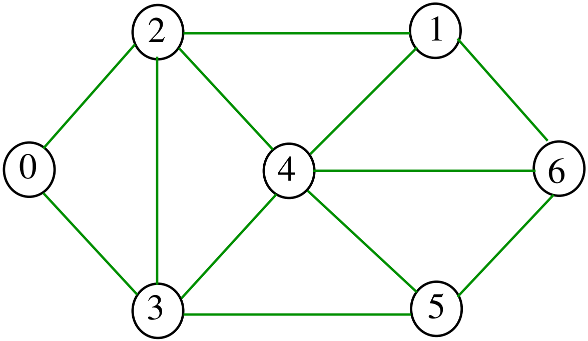
Árvores geradoras de grafos

Árvores geradoras

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

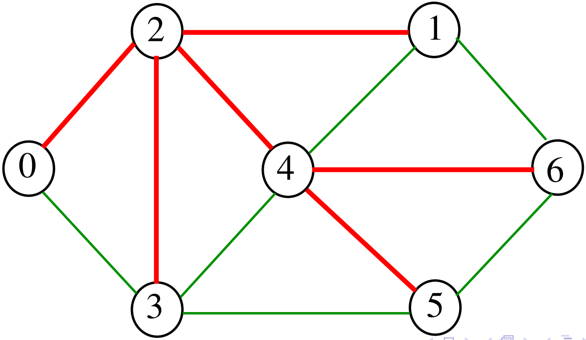
Exemplo:



Árvores geradoras

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

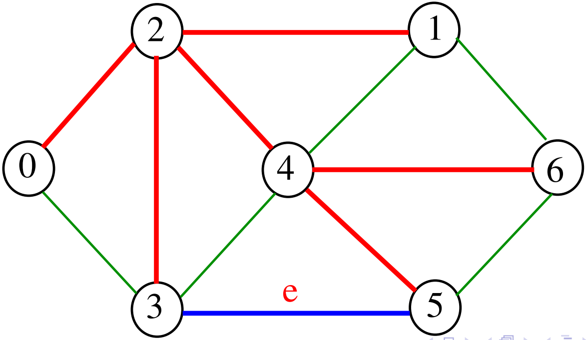
Exemplo: as arestas em vermelho formam uma árvore geradora



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** $C(\mathtt{T}, \mathtt{e})$.

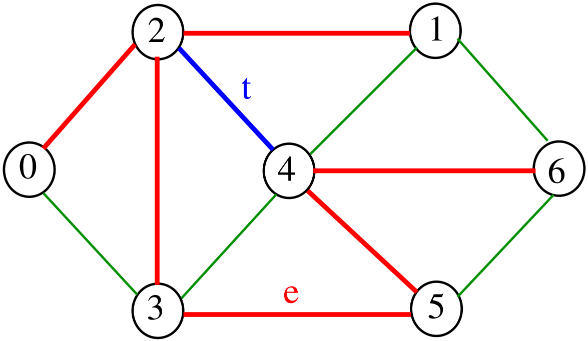
Exemplo: T+e



Primeira propriedade da troca de arestas

Seja T uma árvore geradora de um grafo G Para qualquer aresta t $\in C($T, e$)$, T+e-t é uma **árvore geradora**

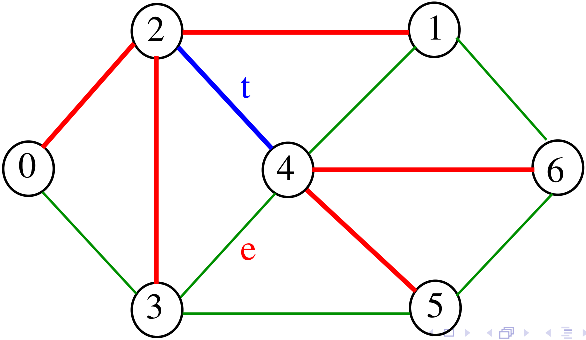
Exemplo: T+e-t



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** $D(\mathtt{T},\mathtt{t})$.

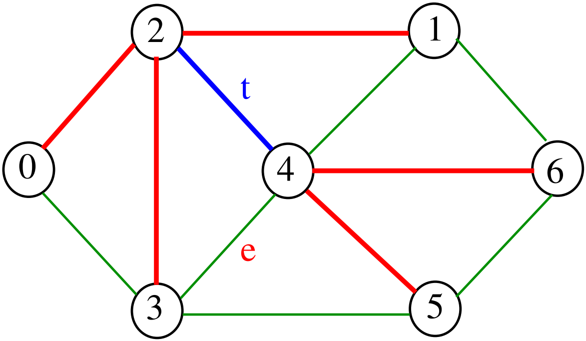
Exemplo: T-t $\{0,1,2,3\} \xrightarrow{D(\mathtt{T},\mathtt{t})} \{4,5,6\}$



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.

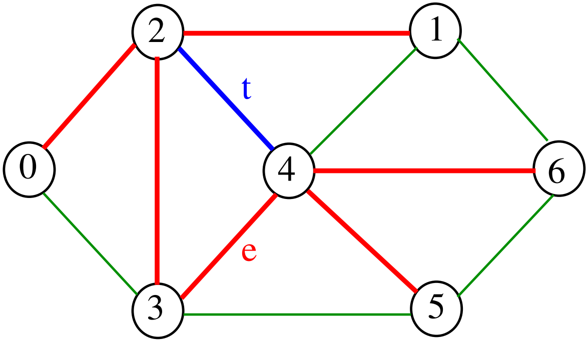
Exemplo: T-t



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+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

Propriedade fundamental

Se $\mathtt{T}$ é árvore geradora de $\mathtt{G}$, $\mathtt{t}$ é uma aresta de $\mathtt{T}$ e $\mathtt{e}$ é uma aresta fora de $\mathtt{T}$, então

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

Propriedade fundamental

Se $\texttt{T}$ é árvore geradora de $\texttt{G}$, $\texttt{t}$ é uma aresta de $\texttt{T}$ e $\texttt{e}$ é uma aresta fora de $\texttt{T}$, então

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

Dem: São equivalentes

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

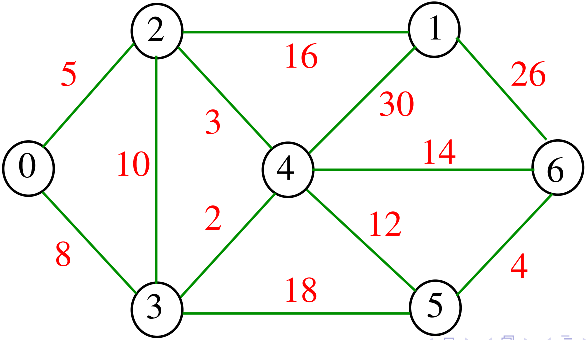
Árvores geradoras de custo mínimo

S 20.1 e 20.2

Árvores geradoras mínimas

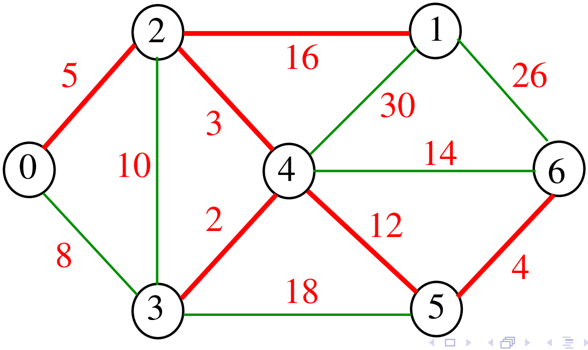
Uma **árvore geradora mínima** (= *minimum spanning tree*), ou MST, de um grafo com custos nas arestas é qualquer árvore geradora do grafo que tenha custo mínimo

Exemplo: um grafo com custos nas arestas



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: MST de custo 42

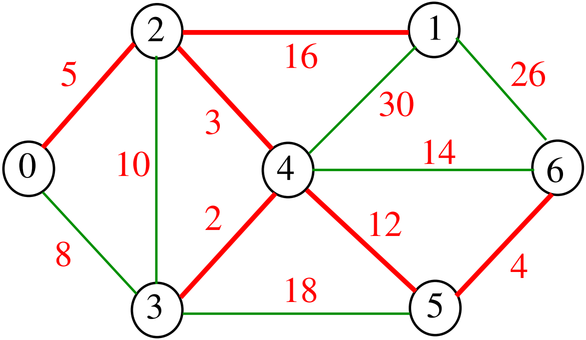


Problema MST

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

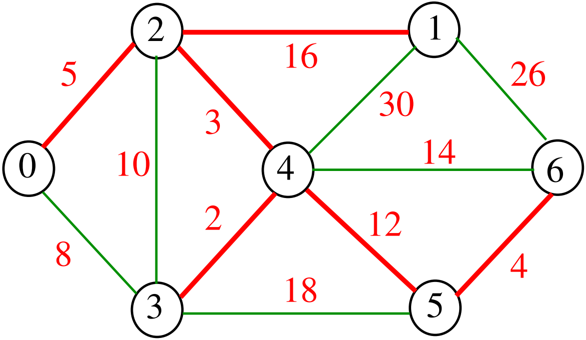
Exemplo: MST de custo 42



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 $\mathtt{T}$ uma árvore geradora satisfazendo a condição de otimalidade.

Vamos mostrar que $\mathtt{T}$ é uma MST.

Seja $\mathtt{M}$ uma MST tal que o número de arestas comuns entre $\mathtt{T}$ e $\mathtt{M}$ seja máximo.

Se $\mathtt{T} = \mathtt{M}$ não há o que demonstrar.

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

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

Demonstração da recíproca

Seja $\mathtt{T}$ uma árvore geradora satisfazendo a condição de otimalidade.

Vamos mostrar que $\mathtt{T}$ é uma MST.

Seja $\mathtt{M}$ uma MST tal que o número de arestas comuns entre $\mathtt{T}$ e $\mathtt{M}$ seja máximo.

Se $\mathtt{T} = \mathtt{M}$ não há o que demonstrar.

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

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

Continuação

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

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

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

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

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

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

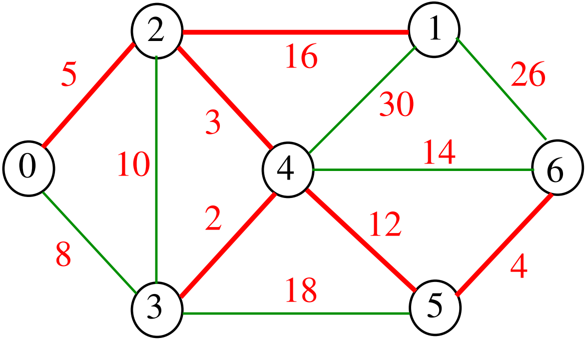
Mas então, $\texttt{M}-\texttt{f}+\texttt{d}$ é uma MST que tem o mesmo custo que $\texttt{M}$, logo é mínima. Por outro lado, tem uma aresta a mais em comum com $\texttt{T}$ do que $\texttt{M}$. Isso contradiz a escolha de $\texttt{M}$.

Portanto, $\texttt{T} = \texttt{M}$, o que mostra que $\texttt{T}$ 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`.

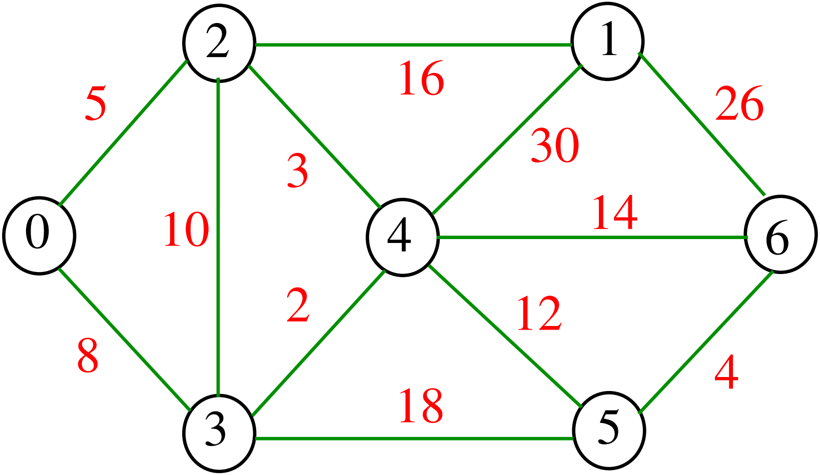
Exemplo: MST de custo 42



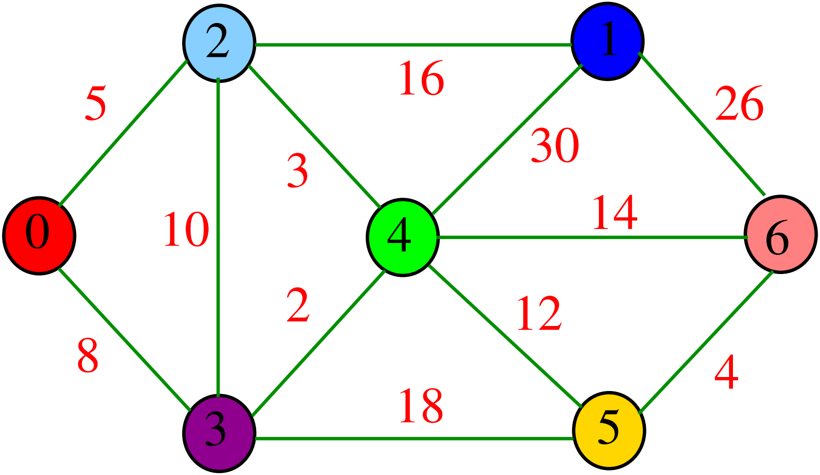
Algoritmo de Kruskal

S 20.3

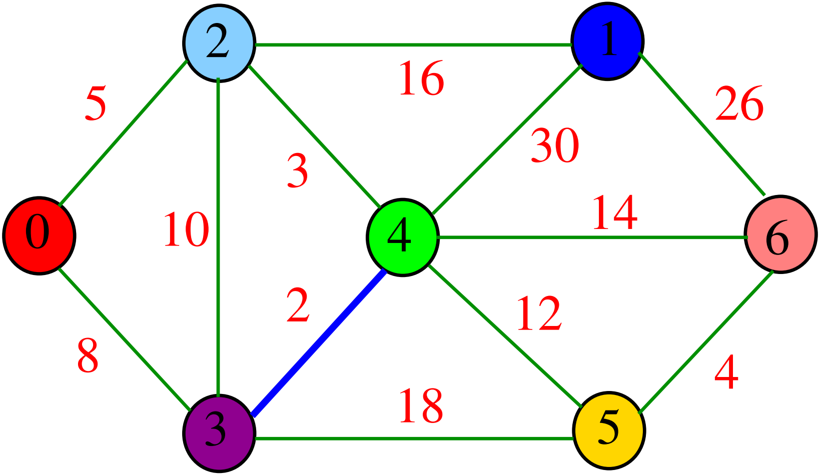
Algoritmo de Kruskal



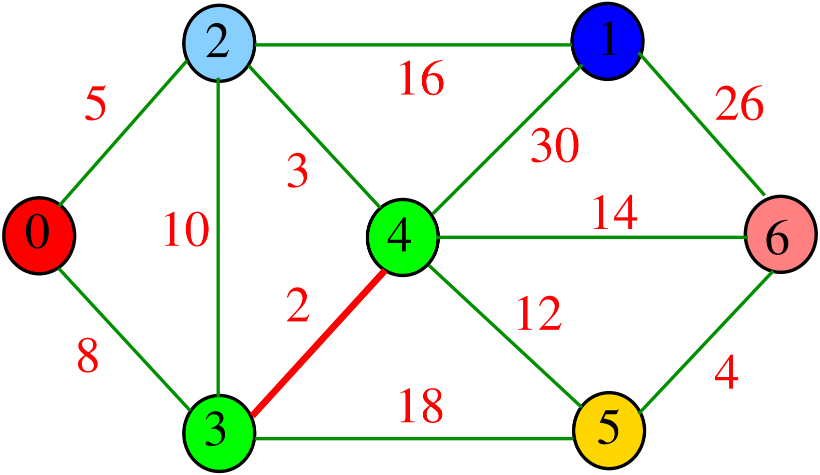
Algoritmo de Kruskal



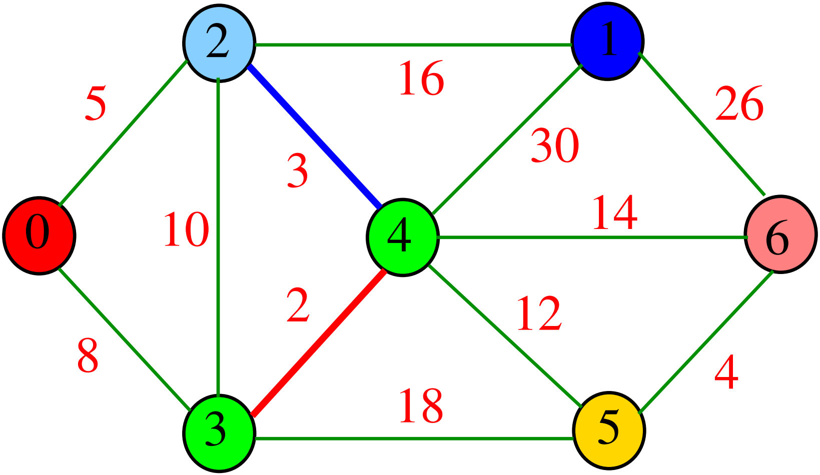
Algoritmo de Kruskal



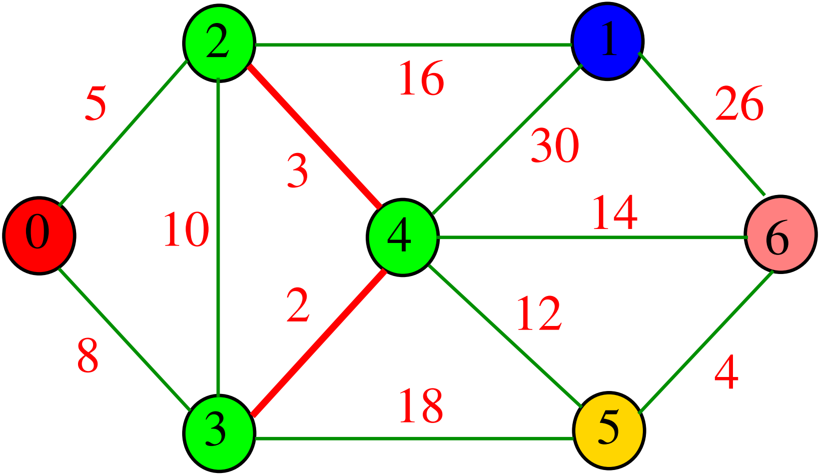
Algoritmo de Kruskal



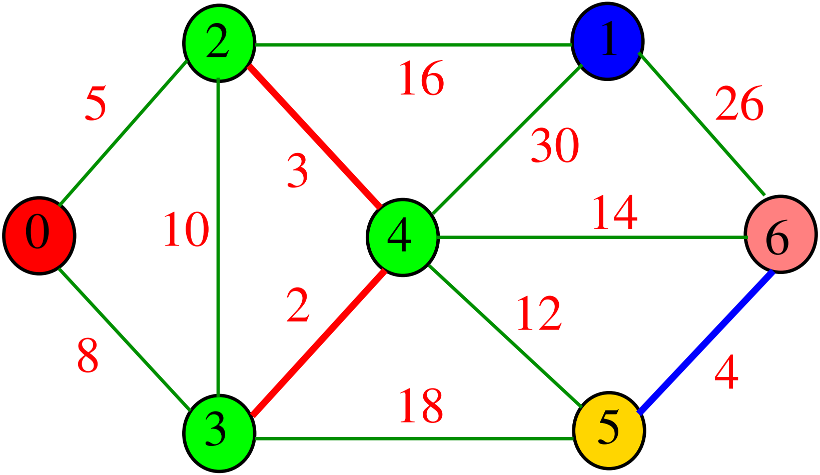
Algoritmo de Kruskal



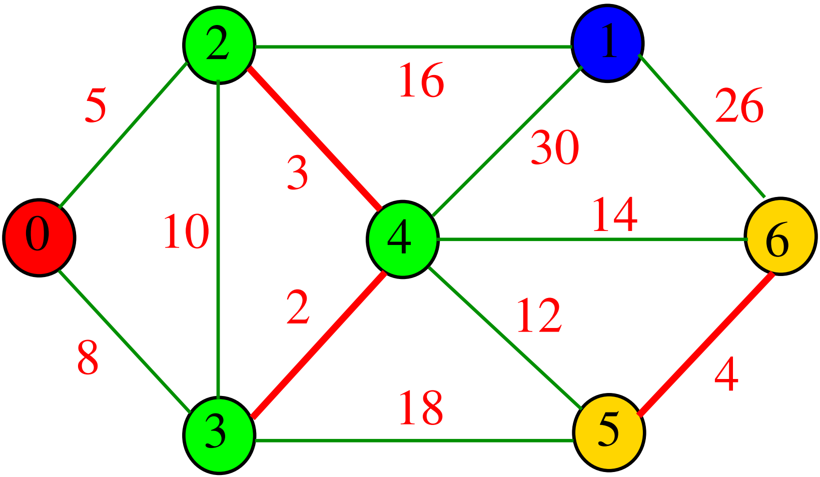
Algoritmo de Kruskal



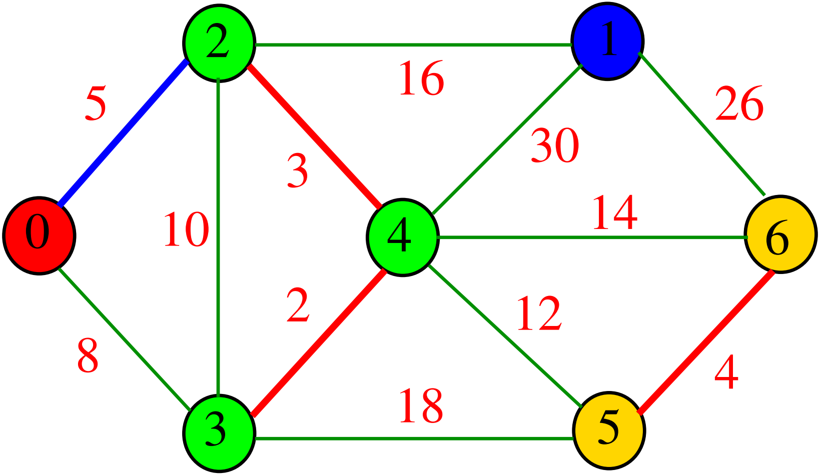
Algoritmo de Kruskal



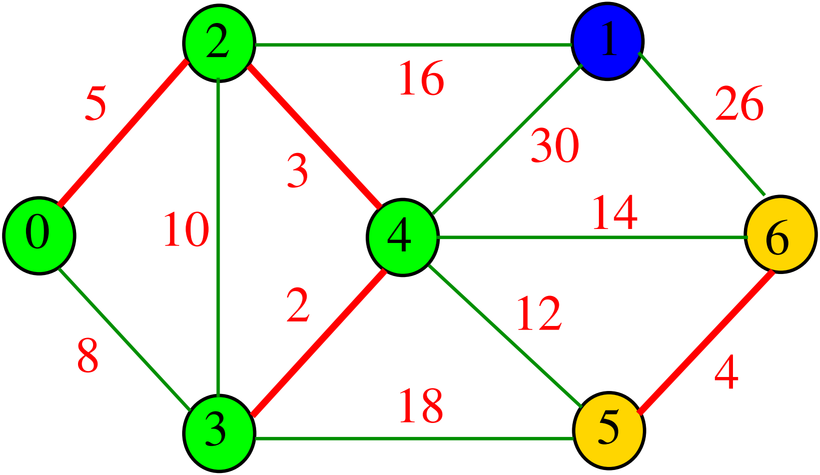
Algoritmo de Kruskal



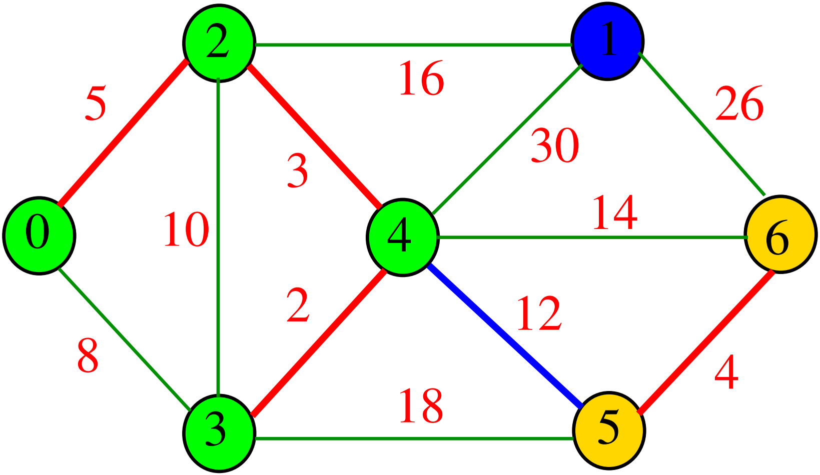
Algoritmo de Kruskal



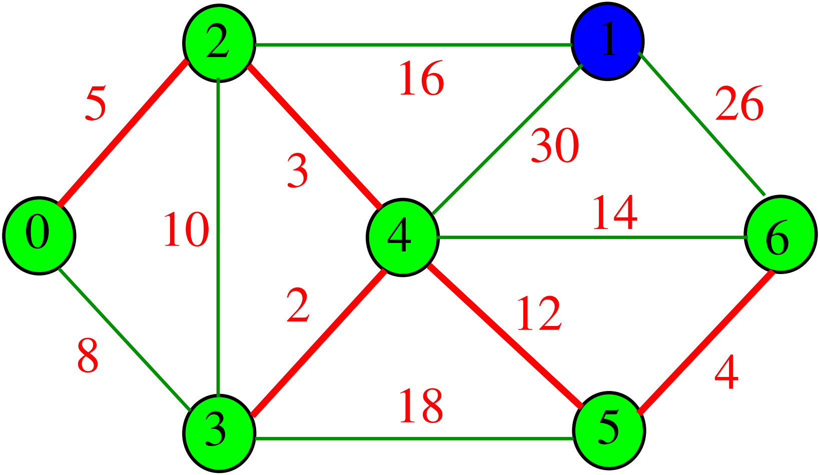
Algoritmo de Kruskal



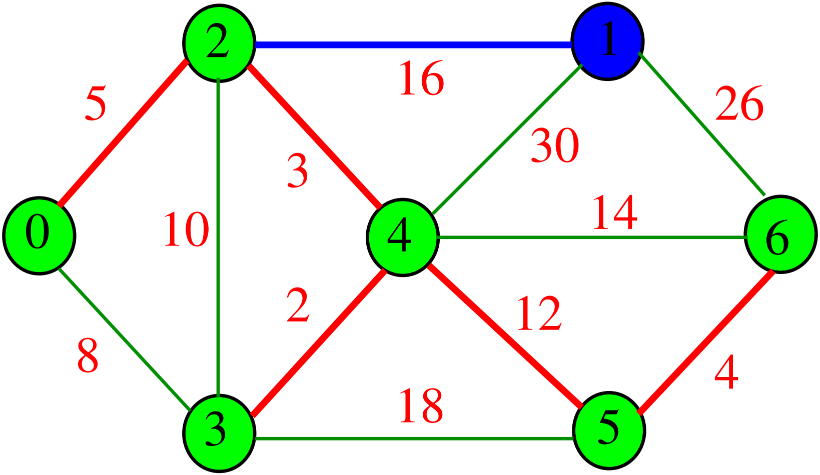
Algoritmo de Kruskal



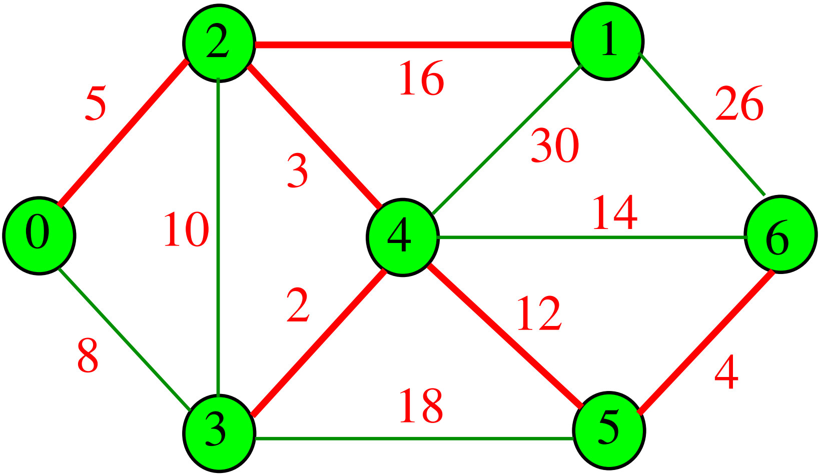
Algoritmo de Kruskal



Algoritmo de Kruskal



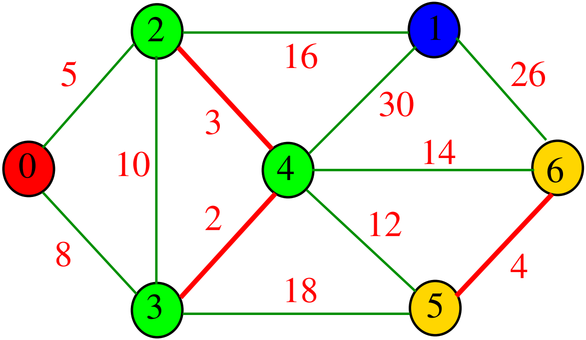
Algoritmo de Kruskal



Subfloresta

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

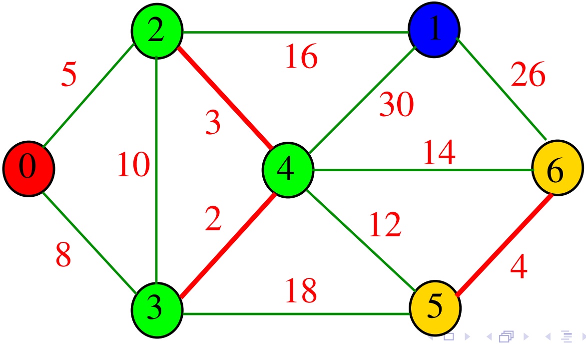
Exemplo: As arestas vermelhas que ligam os vértices verdes e amarelos e formam uma subfloresta



Floresta geradora

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

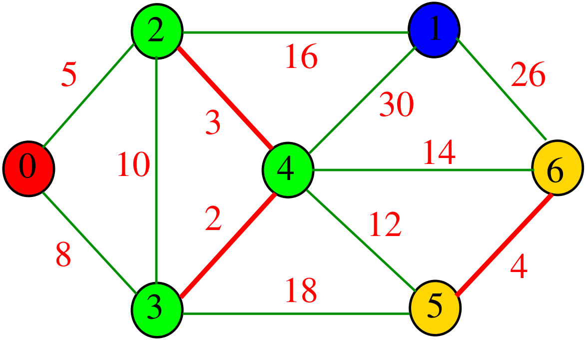
Exemplo: Arestas vermelhas ligando os vértices verdes e amarelos, junto com os vértices azul e vermelho, forma uma floresta geradora



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
Atualize: F $\leftarrow$ 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* $C(F, e)$.

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

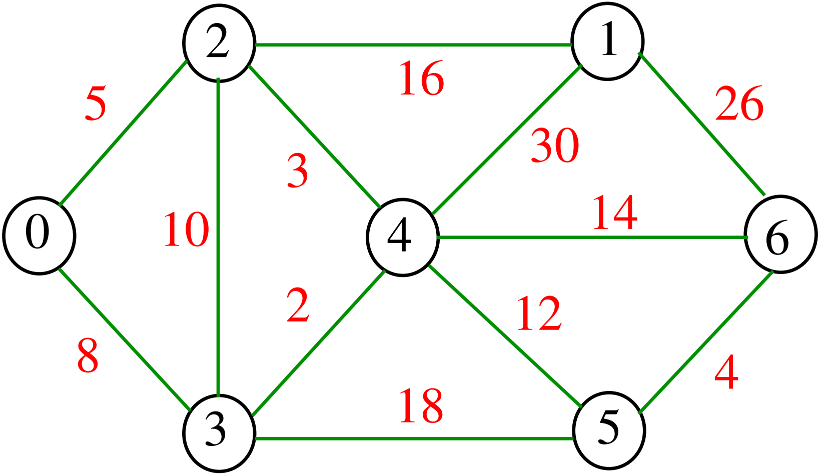
A função armazena as arestas das MSTs no vetor mst[0..k-1] e devolve k.

```
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 ;
```

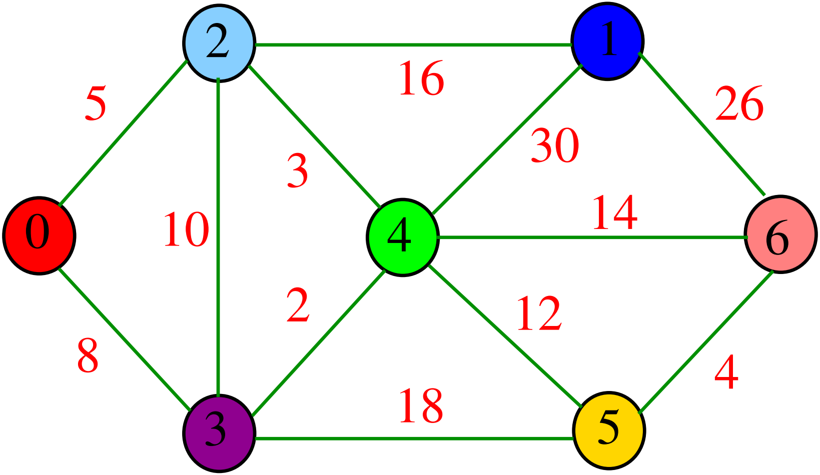
Implementação grosseira

```
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; }
```

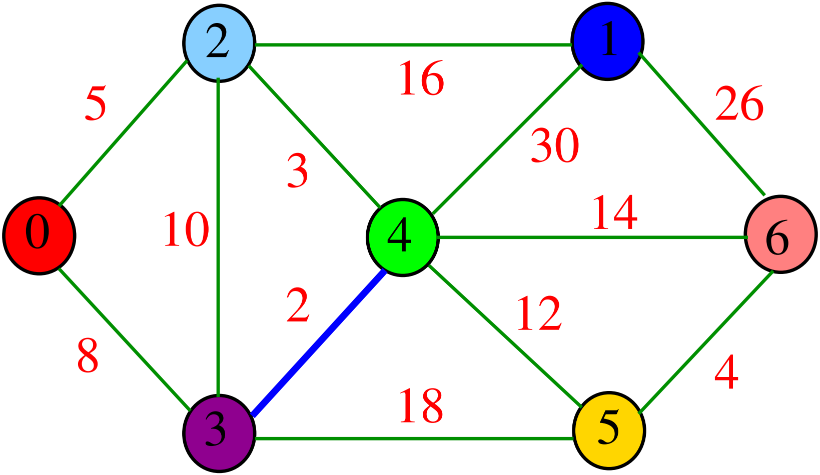
Algoritmo de Kruskal



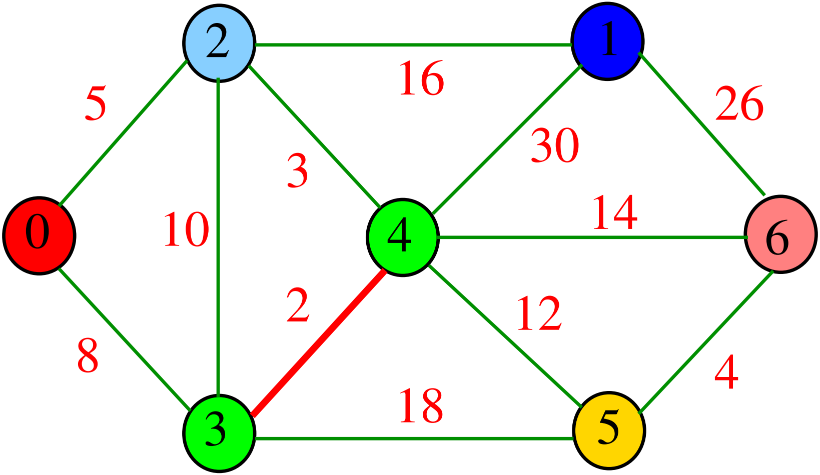
Algoritmo de Kruskal



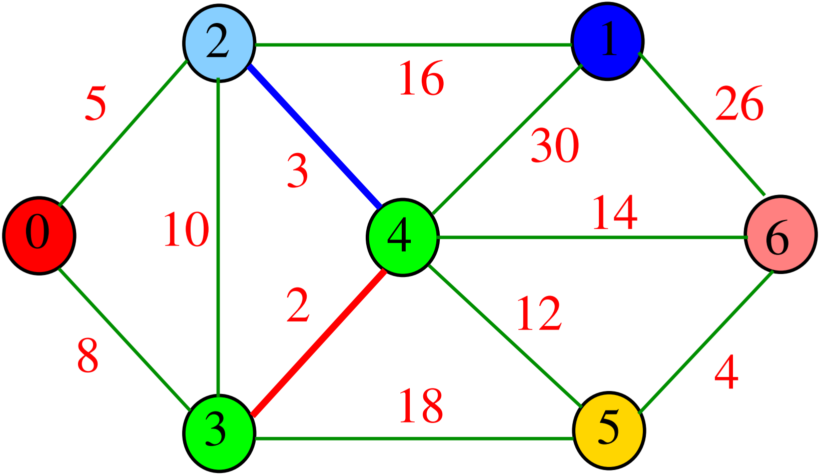
Algoritmo de Kruskal



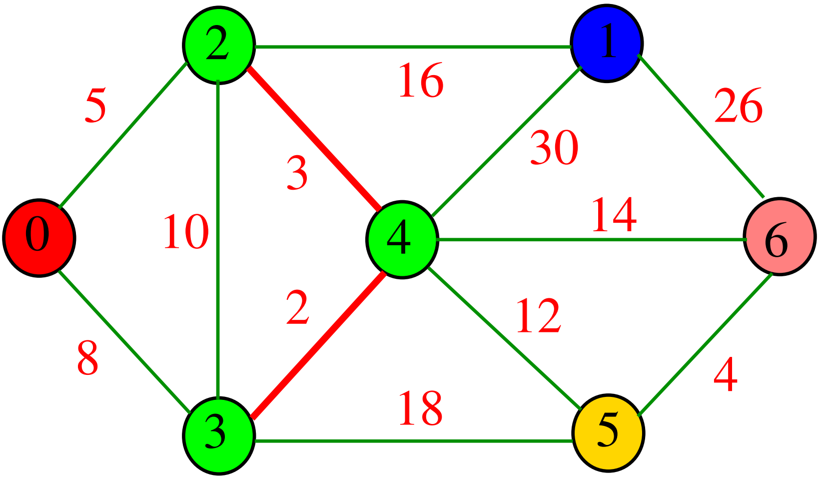
Algoritmo de Kruskal



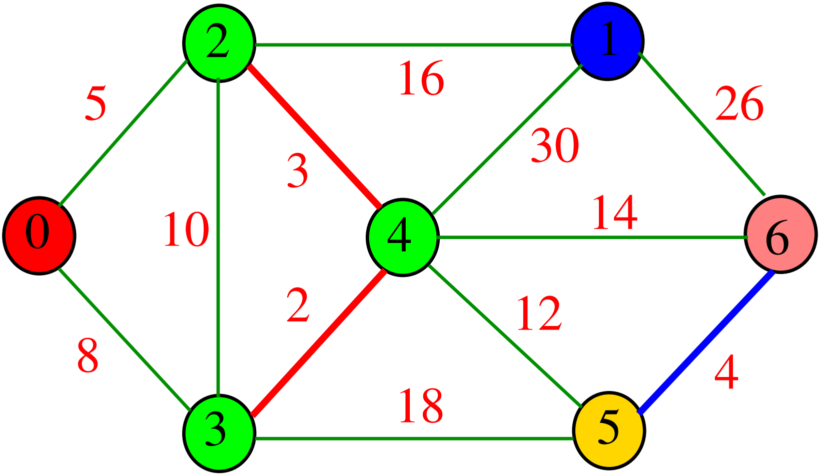
Algoritmo de Kruskal



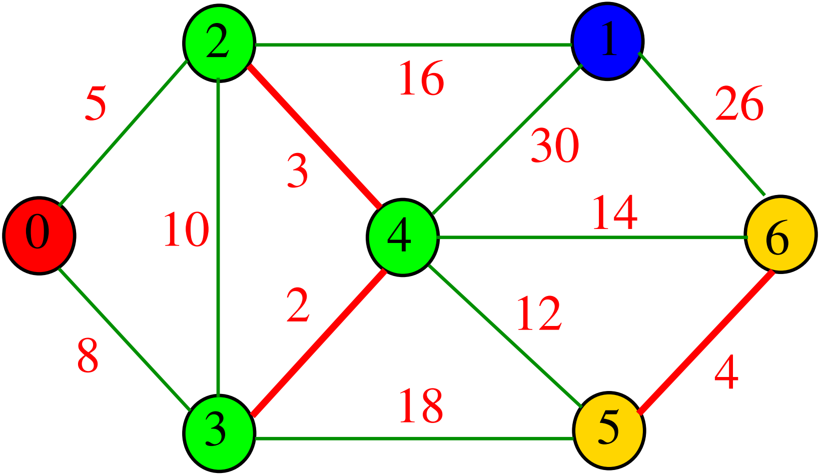
Algoritmo de Kruskal



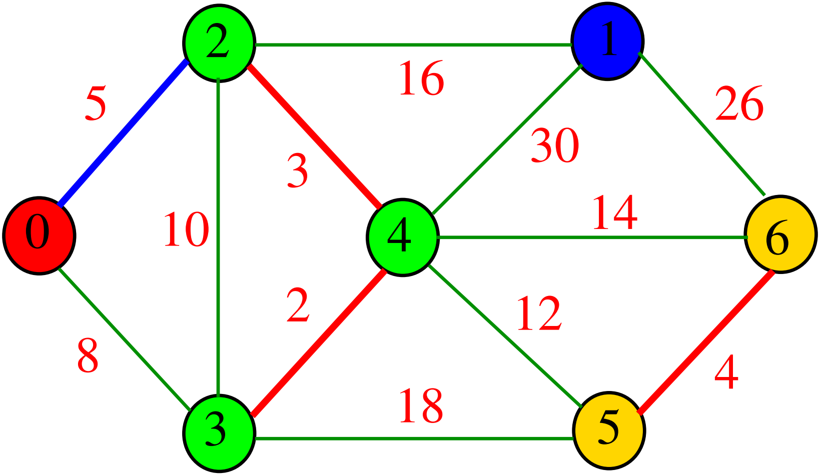
Algoritmo de Kruskal



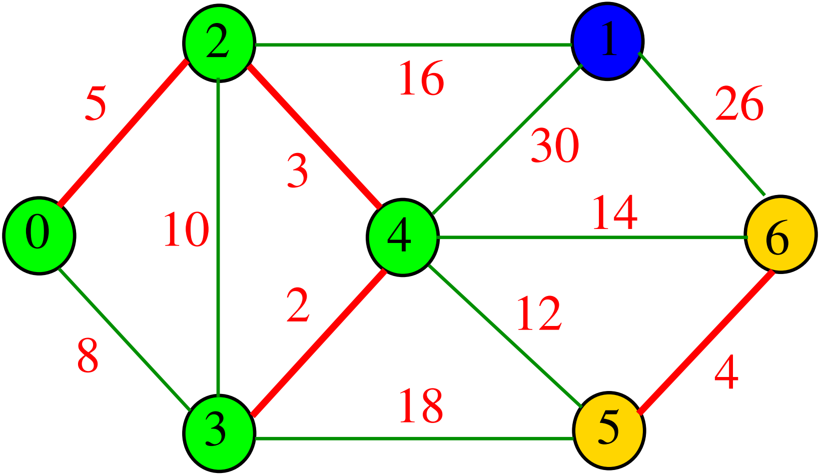
Algoritmo de Kruskal



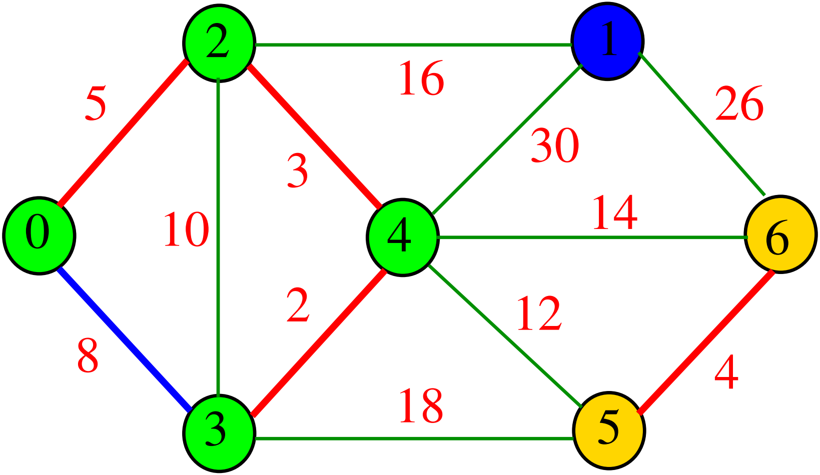
Algoritmo de Kruskal



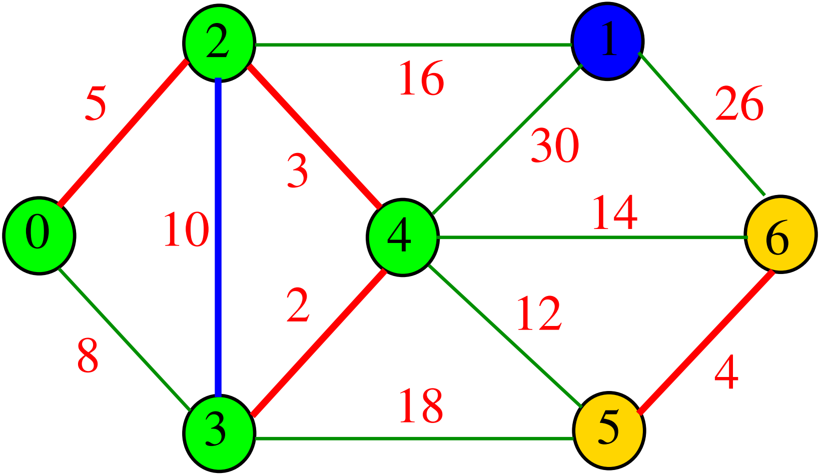
Algoritmo de Kruskal



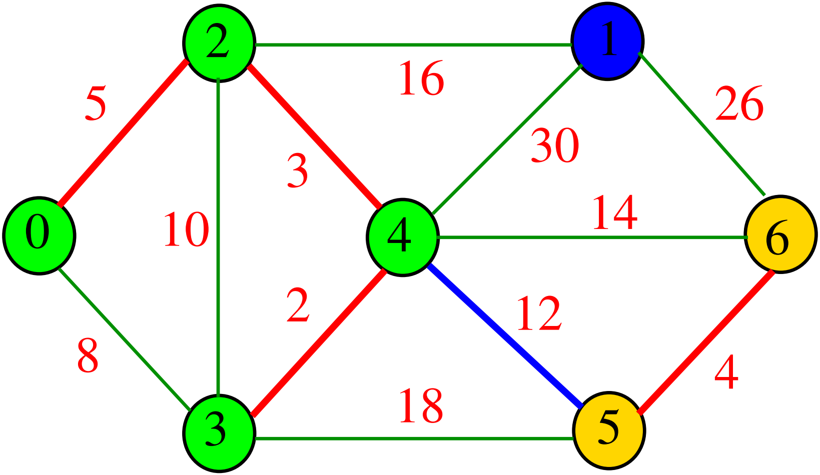
Algoritmo de Kruskal



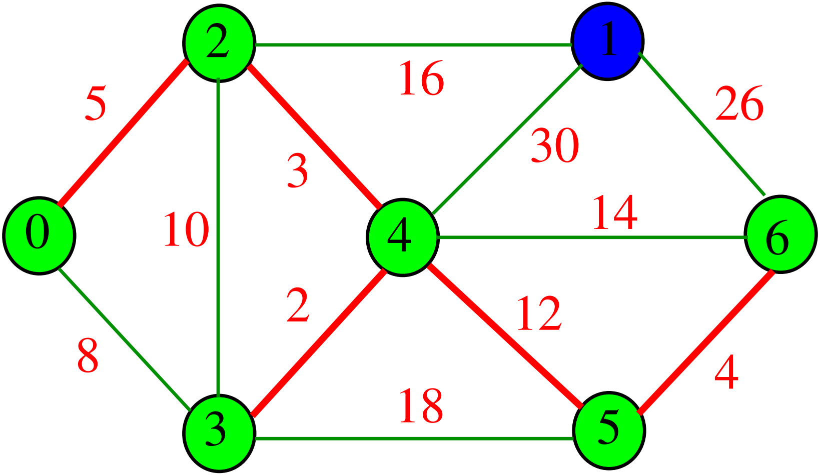
Algoritmo de Kruskal



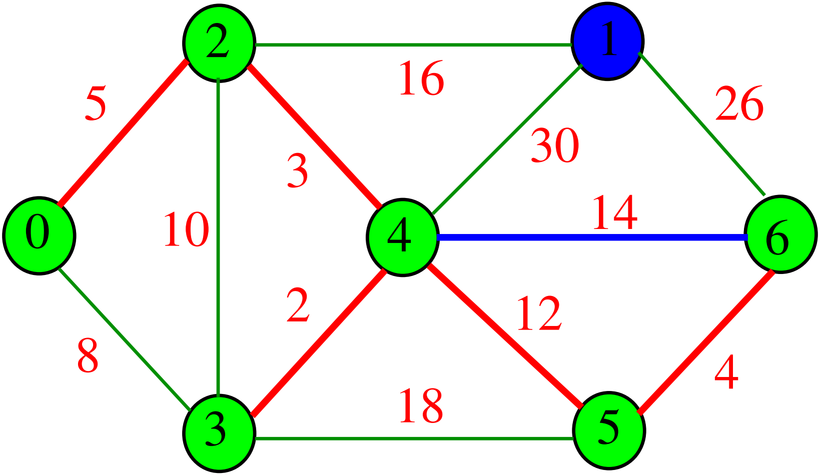
Algoritmo de Kruskal



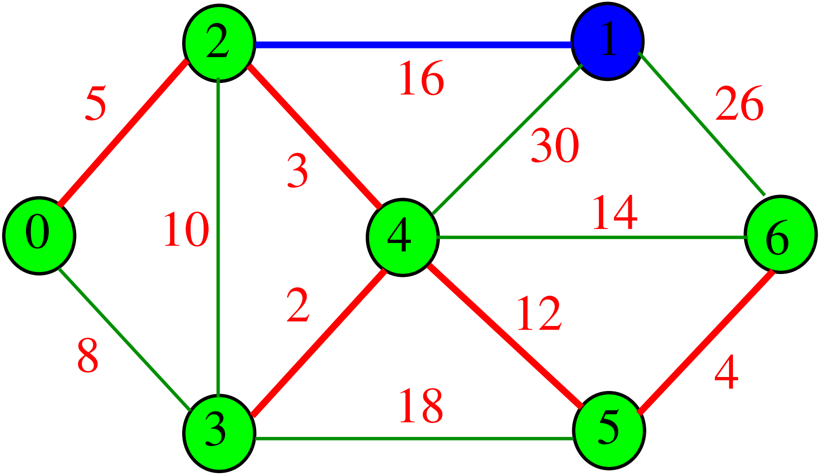
Algoritmo de Kruskal



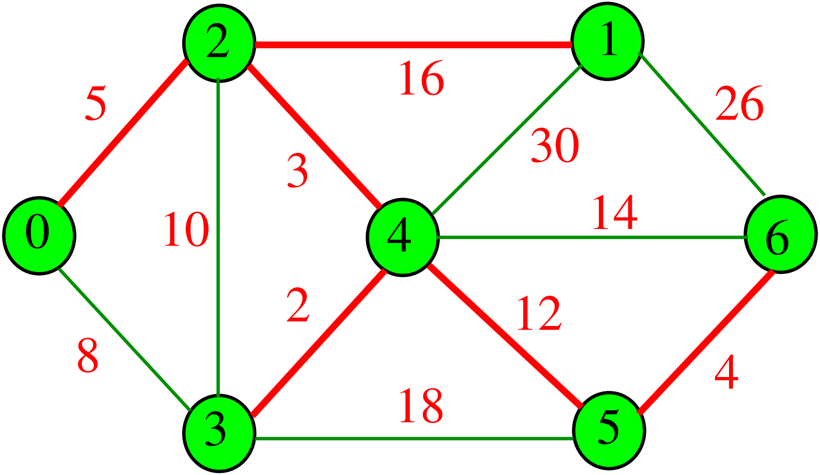
Algoritmo de Kruskal



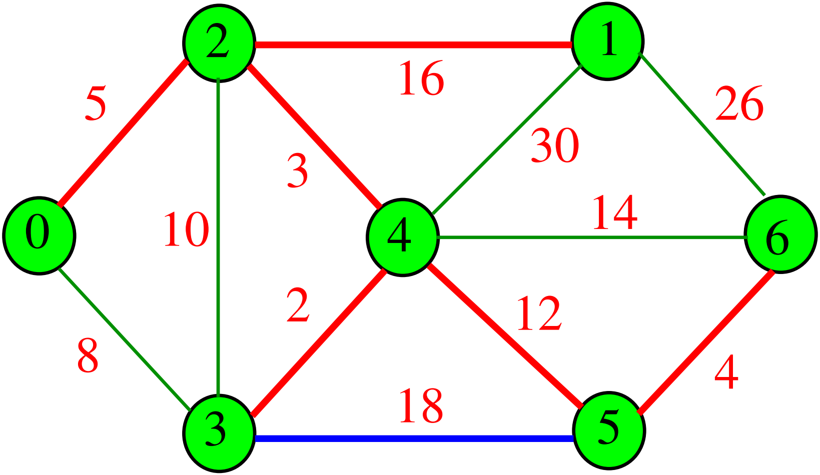
Algoritmo de Kruskal



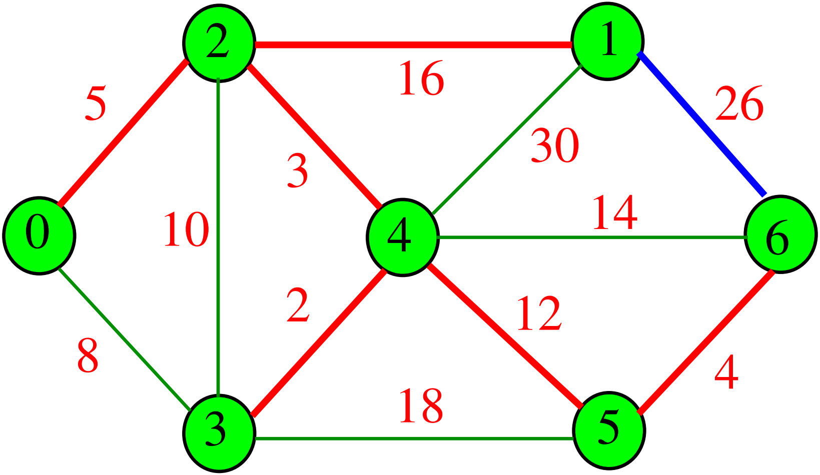
Algoritmo de Kruskal



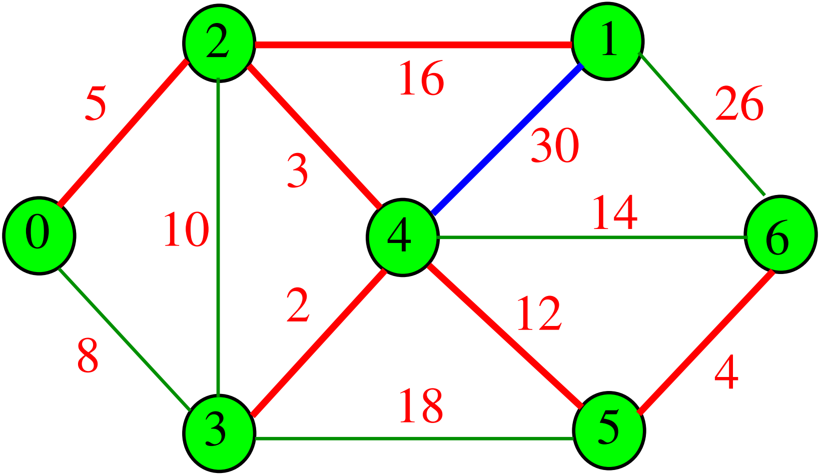
Algoritmo de Kruskal



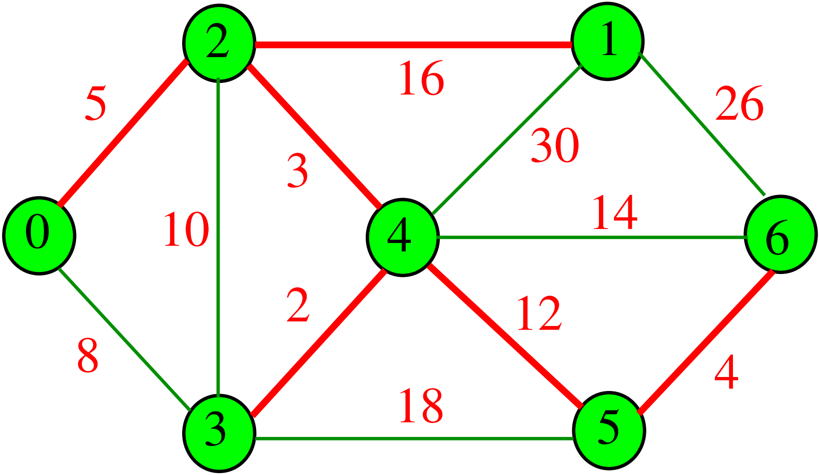
Algoritmo de Kruskal



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 ;
}
```

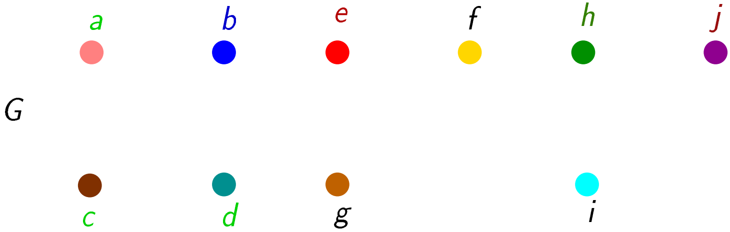
Coleção disjunta dinâmica

Conjuntos são **modificados ao longo do tempo**

Coleção disjunta dinâmica

Conjuntos são **modificados ao longo do tempo**

Exemplo: grafo dinâmico

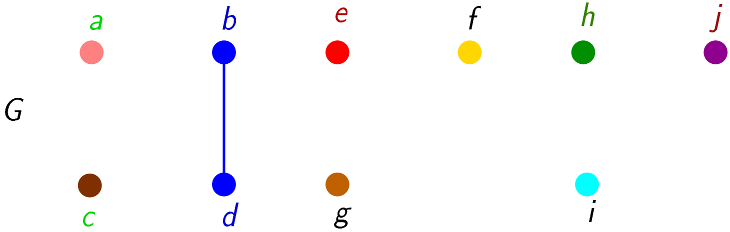


aresta	componentes
	$\{a\}$ $\{b\}$ $\{c\}$ $\{d\}$ $\{e\}$ $\{f\}$ $\{g\}$ $\{h\}$ $\{i\}$

Coleção disjunta dinâmica

Conjuntos são **modificados ao longo do tempo**

Exemplo: grafo dinâmico

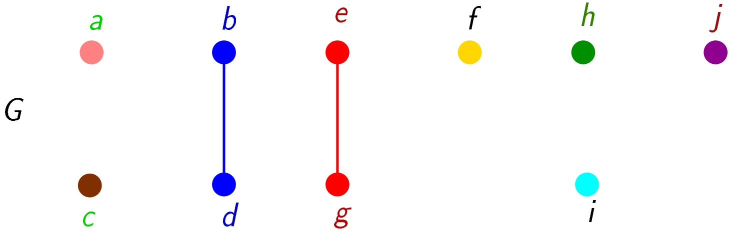


aresta	componentes
(b, d)	$\{a\}$ $\{b, d\}$ $\{c\}$ $\{e\}$ $\{f\}$ $\{g\}$ $\{h\}$ $\{i\}$ $\{j\}$

Coleção disjunta dinâmica

Conjuntos são **modificados ao longo do tempo**

Exemplo: grafo dinâmico

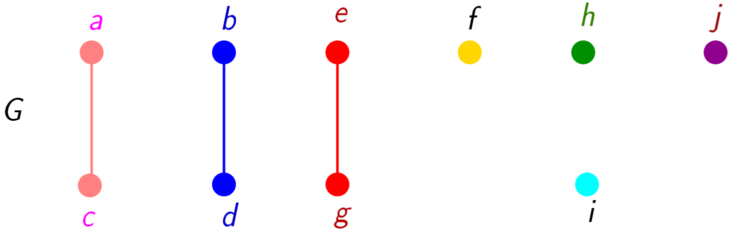


aresta	componentes							
(e, g)	$\{a\}$	$\{b, d\}$	$\{c\}$	$\{e, g\}$	$\{f\}$	$\{h\}$	$\{i\}$	$\{j\}$

Coleção disjunta dinâmica

Conjuntos são **modificados ao longo do tempo**

Exemplo: grafo dinâmico

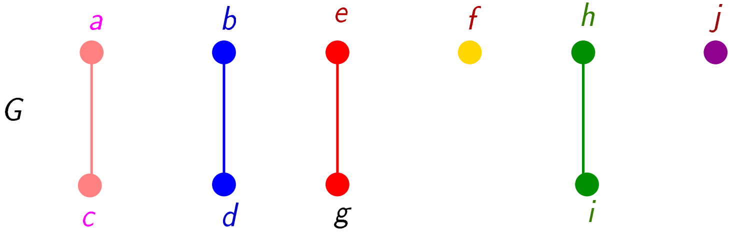


aresta	componentes
(a, c)	$\{a, c\}$ $\{b, d\}$ $\{e, g\}$ $\{f\}$ $\{h\}$ $\{i\}$ $\{j\}$

Coleção disjunta dinâmica

Conjuntos são **modificados ao longo do tempo**

Exemplo: grafo dinâmico

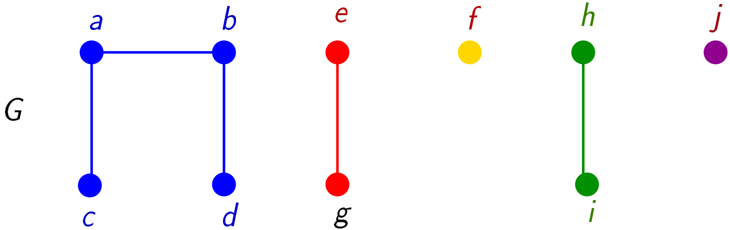


aresta	componentes
(h, i)	$\{a, c\}$ $\{b, d\}$ $\{e, g\}$ $\{f\}$ $\{h, i\}$ $\{j\}$

Coleção disjunta dinâmica

Conjuntos são **modificados ao longo do tempo**

Exemplo: grafo dinâmico

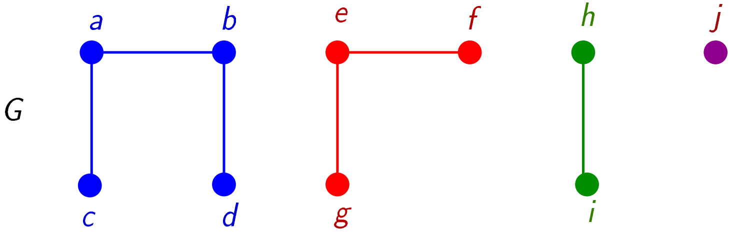


aresta	componentes
(a, b)	$\{a, b, c, d\}$ $\{e, g\}$ $\{f\}$ $\{h, i\}$ $\{j\}$

Coleção disjunta dinâmica

Conjuntos são **modificados ao longo do tempo**

Exemplo: grafo dinâmico

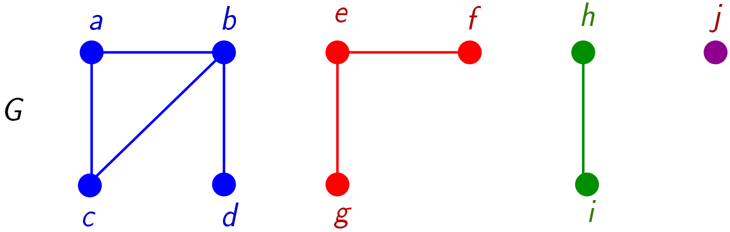


aresta	componentes
(e, f)	$\{a, b, c, d\}$ $\{e, f, g\}$ $\{h, i\}$ $\{j\}$

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 : (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

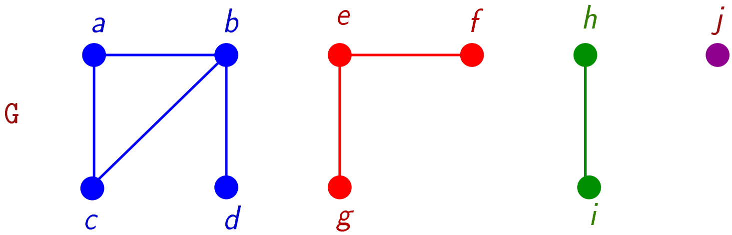
Seqüência de operações MakeSet, Union, FindSet

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

Que estrutura de dados usar?

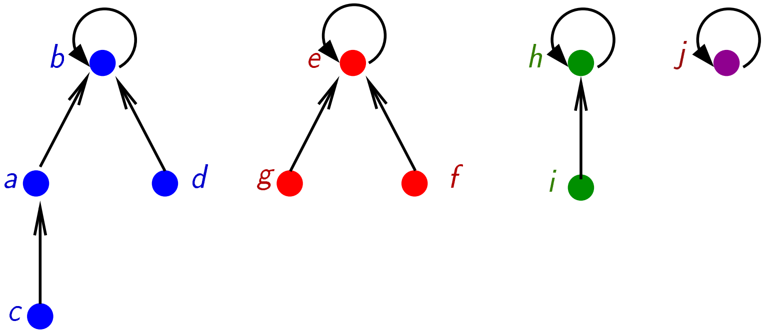
Compromissos (*trade-off* s).

Estrutura *disjoint-set forest*



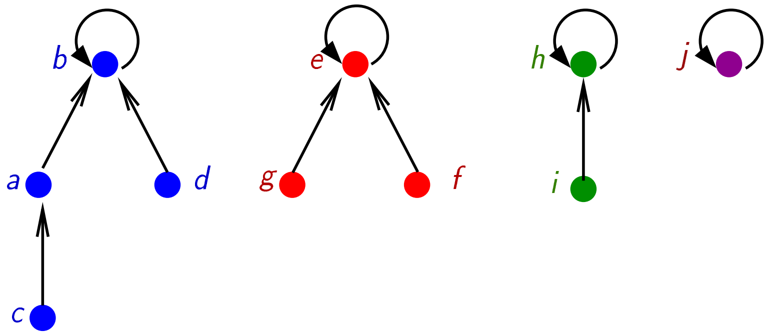
- cada conjunto tem uma *raiz*, que é o seu representate
- cada nó x tem um *pai*
- $pai[\mathtt{x}] = \mathtt{x}$ se e só se x é uma raiz

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$_0$ e FindSet$_0$



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

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

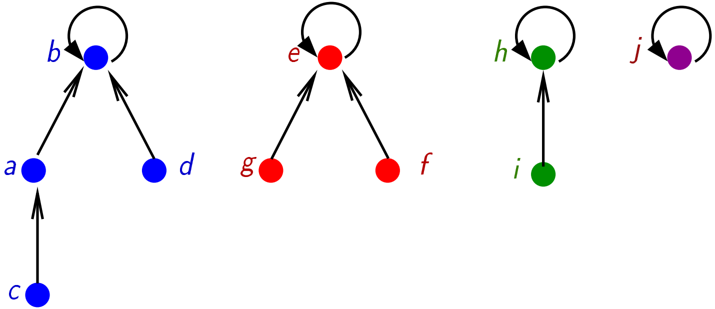
FINDSET$_0$ $(\mathtt{x})$

1 $\quad$ **enquanto** $pai[\mathtt{x}] \neq \mathtt{x}$ **faça**

2 $\qquad \mathtt{x} \leftarrow pai[\mathtt{x}]$

3 $\quad$ **devolva** $\mathtt{x}$

Union$_0$



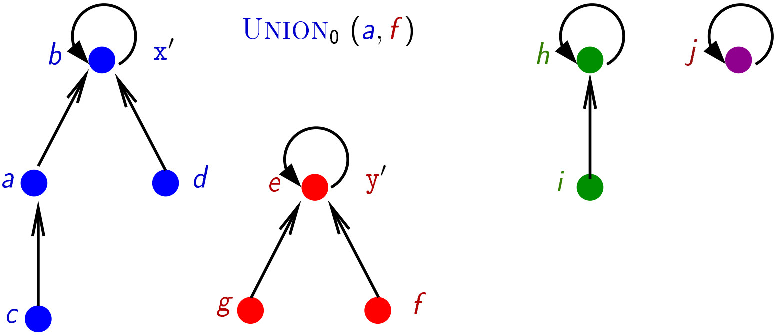
$\text{UNION}_0\,(x, y)$

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

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

$pai[y'] \leftarrow x'$

Union$_0$



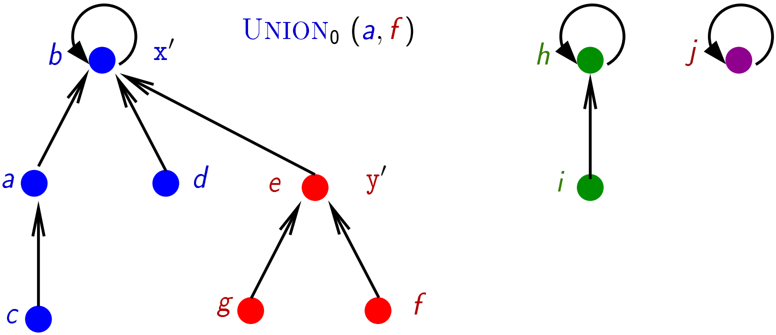
UNION$_0$ (x, y)

1 $x' \leftarrow$ FINDSET$_0$ (x)

2 $y' \leftarrow$ FINDSET$_0$ (y)

3 $pai[y'] \leftarrow x'$

Union$_0$



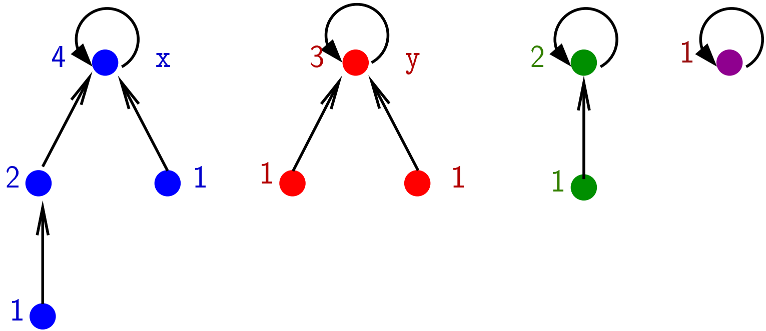
UNION$_0$ (x, y)

1 $\quad x' \leftarrow$ FINDSET$_0$ (x)

2 $\quad y' \leftarrow$ FINDSET$_0$ (y)

3 $\quad pai[y'] \leftarrow x'$

Melhoramento 1: *union by rank*



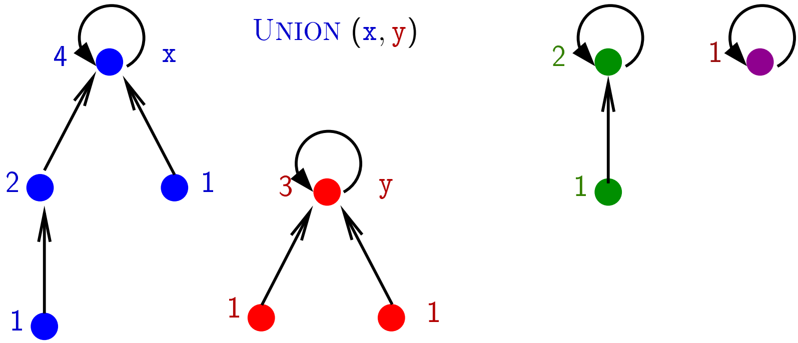
$\text{rank}[x]$ = no. de nós de x

MakeSet (x)

1 $\quad pai[x] \leftarrow x$

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

Melhoramento 1: *union by rank*



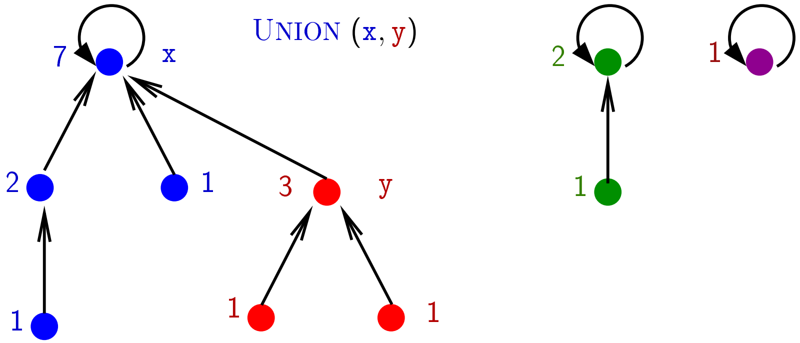
rank[x] = posto de nós de x

MAKESET (x)

1 *pai*[x] ← x

2 rank[x] ← 1

Melhoramento 1: *union by rank*



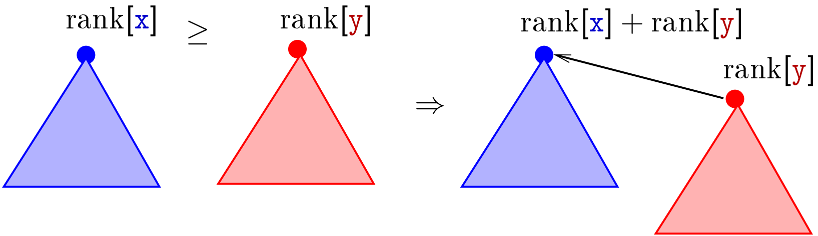
rank[x] = posto de nós de x

MAKESET (x)

1 *pai*[x] ← x

2 rank[x] ← 1

Melhoramento 1: *union by rank*



Melhoramento 1: *union by rank*

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

$x' \leftarrow$ FINDSET (x)
$y' \leftarrow$ FINDSET (y) $\triangleright$ supõe que $x' \neq y'$
se $\mathrm{rank}[x'] > \mathrm{rank}[y']$
  então *pai*$[y'] \leftarrow x'$
    $\mathrm{rank}[y'] \leftarrow \mathrm{rank}[y'] + \mathrm{rank}[x']$
  senão *pai*$[x'] \leftarrow y'$
    $\mathrm{rank}[x'] = \mathrm{rank}[x'] + \mathrm{rank}[y']$

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

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 $O(\mathtt{E} \lg \mathtt{V})$.

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*

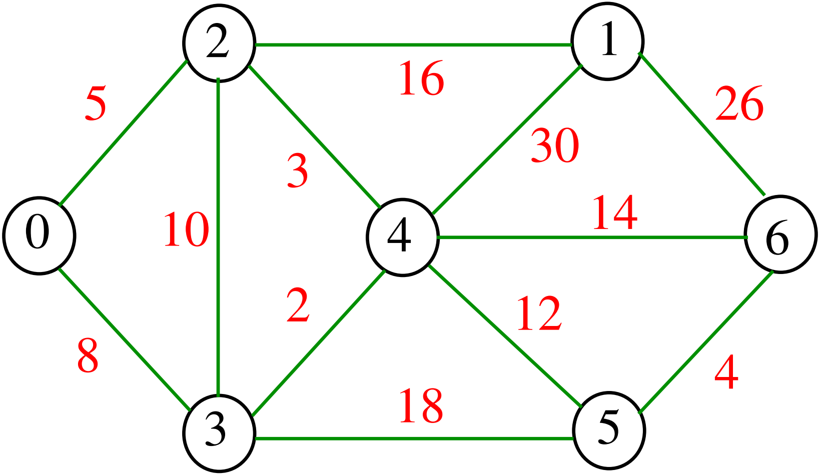
Os algoritmos funcionam para arestas com **custos quaisquer**.

Algoritmo de Boruvka

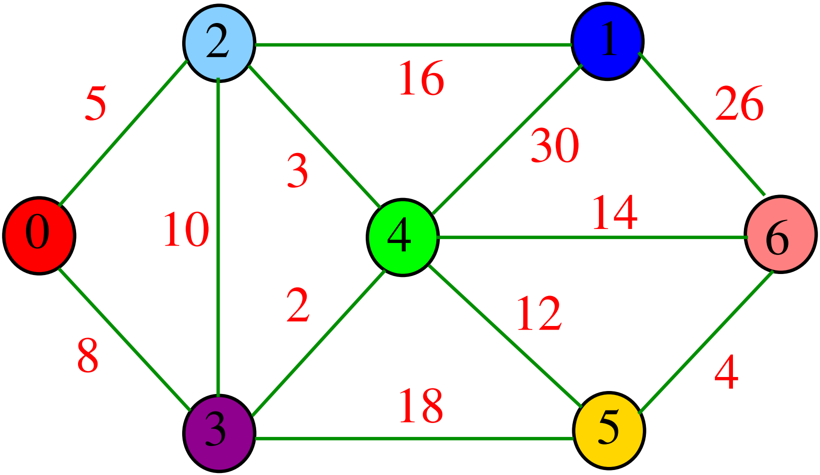
(Interesse histórico: 1926)

S 20.3

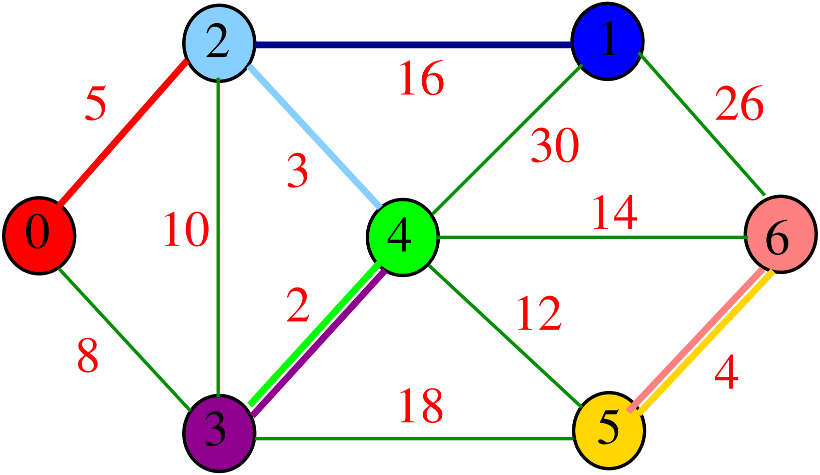
Algoritmo de Boruvka



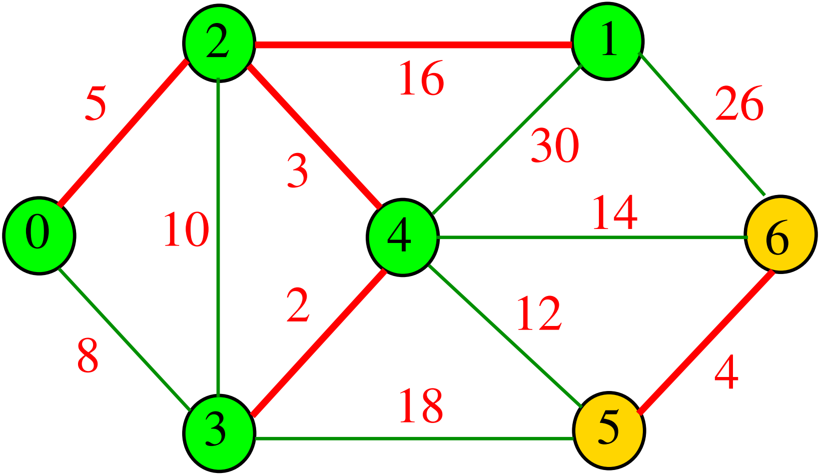
Algoritmo de Boruvka



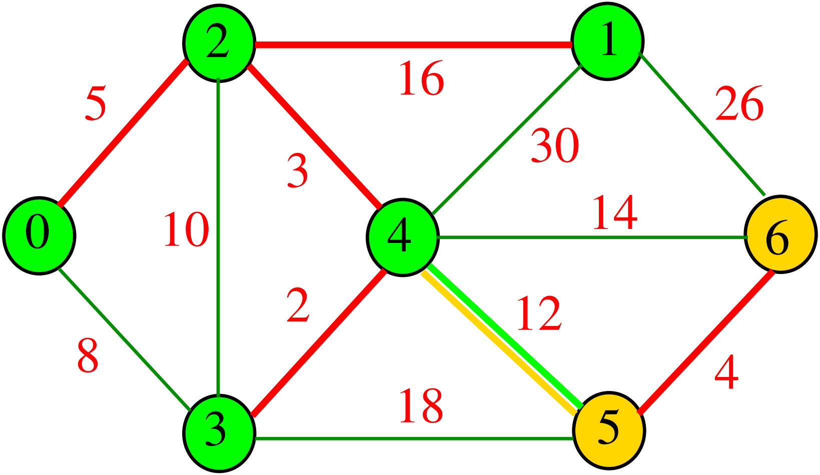
Algoritmo de Boruvka



Algoritmo de Boruvka



Algoritmo de Boruvka



Algoritmo de Boruvka