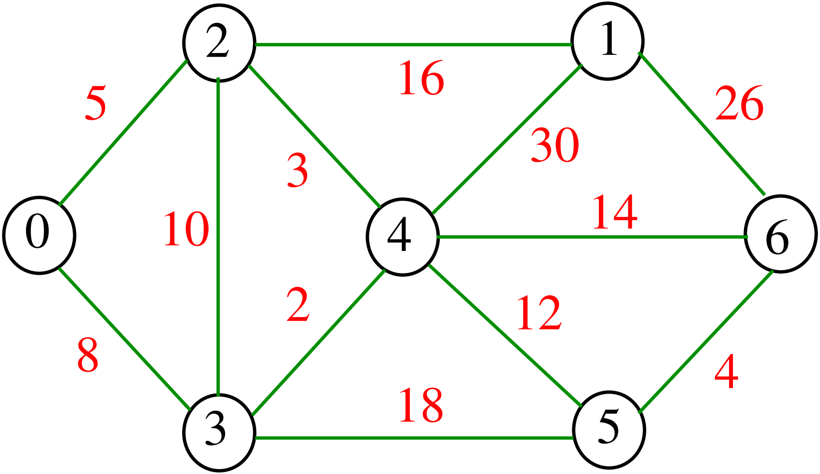
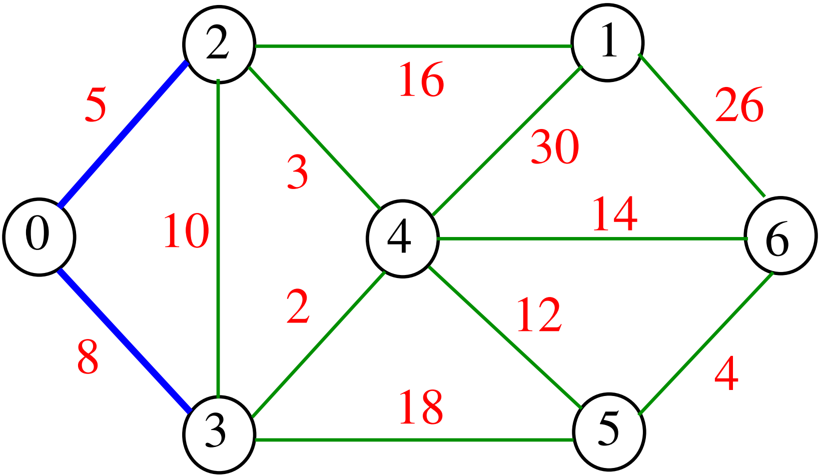
Algoritmo de Prim

S 20.3

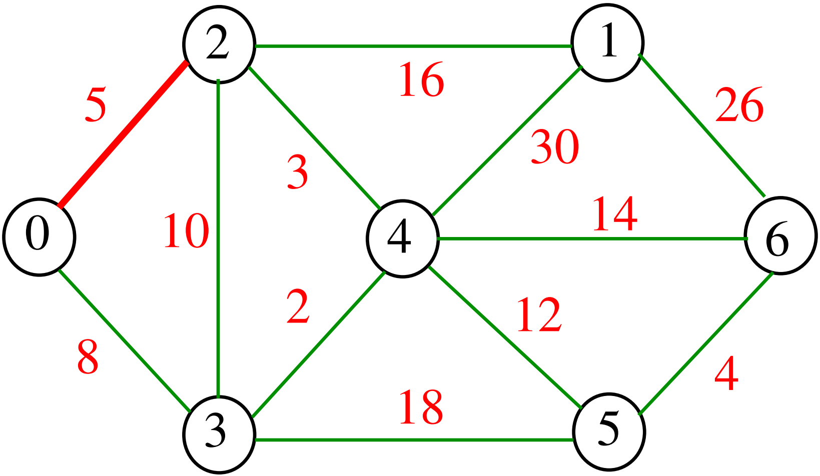
Simulação



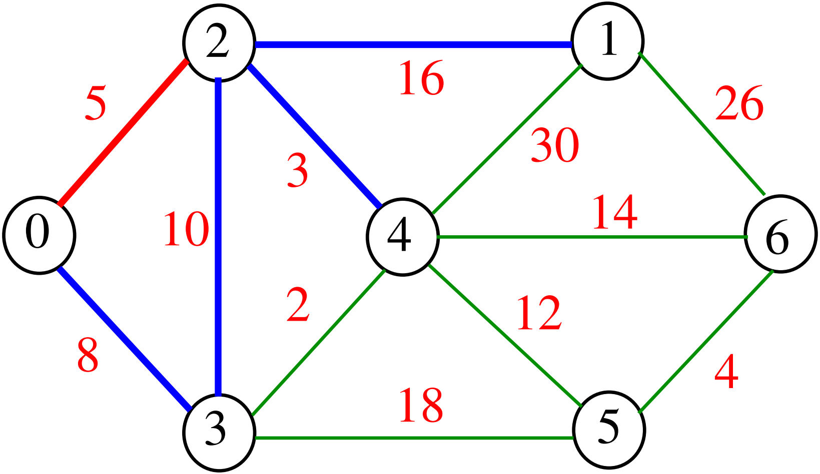
Simulação



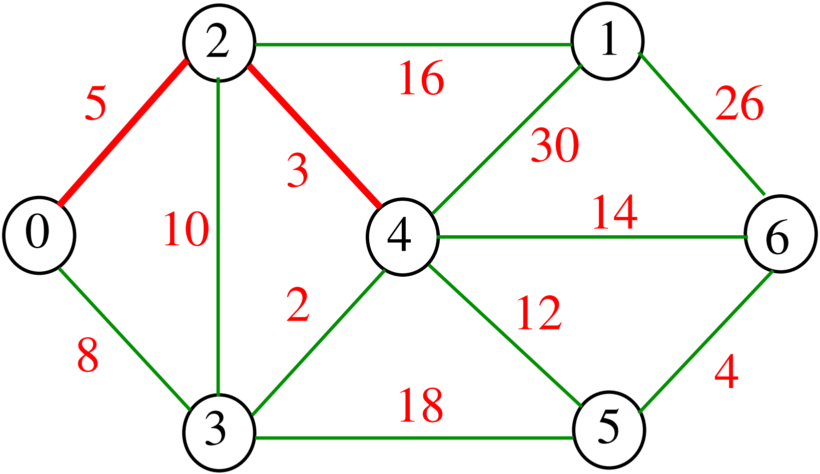
Simulação



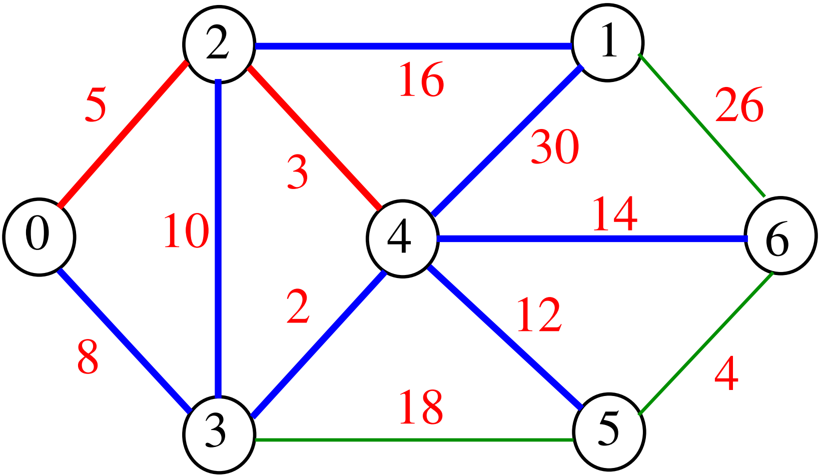
Simulação



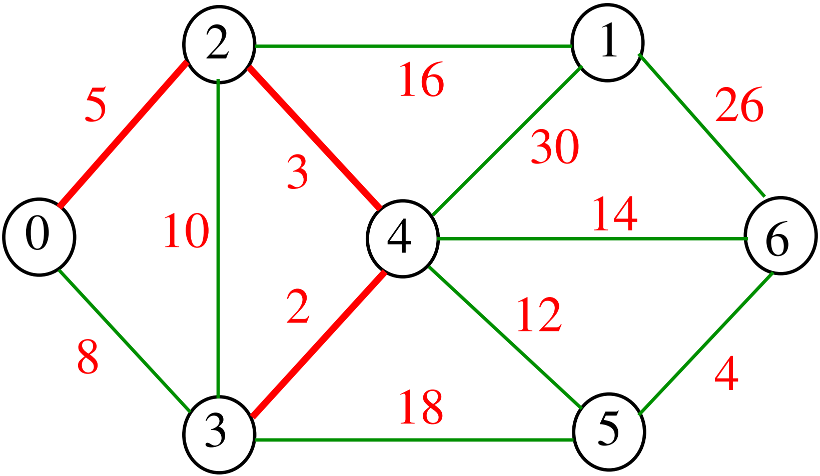
Simulação



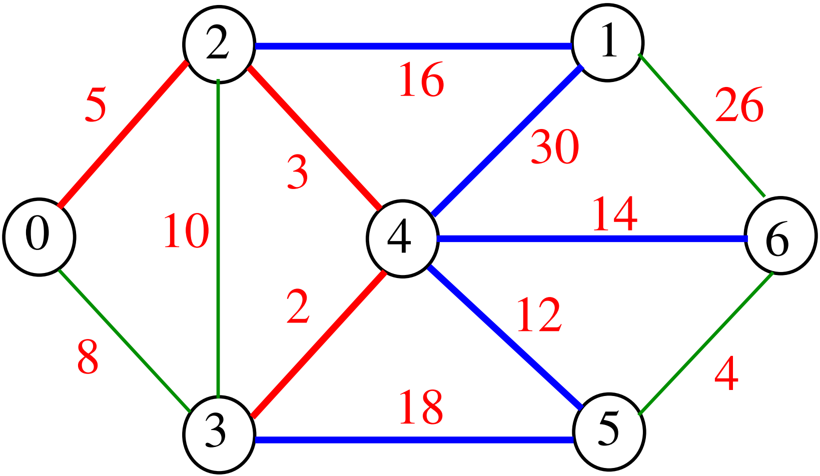
Simulação



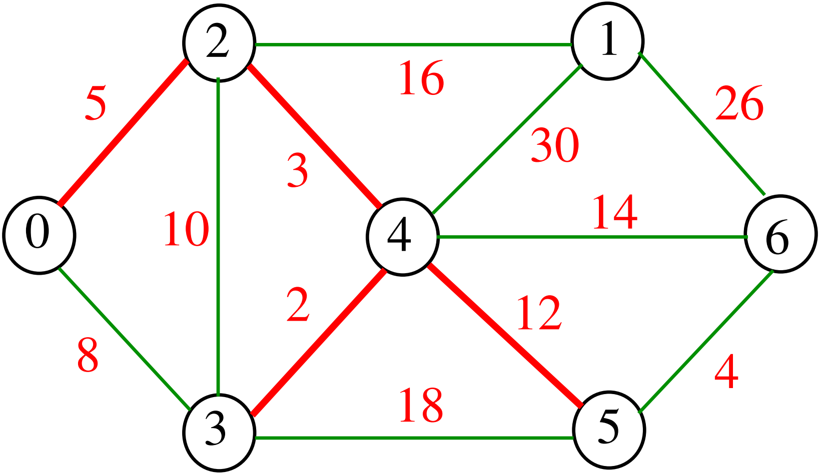
Simulação



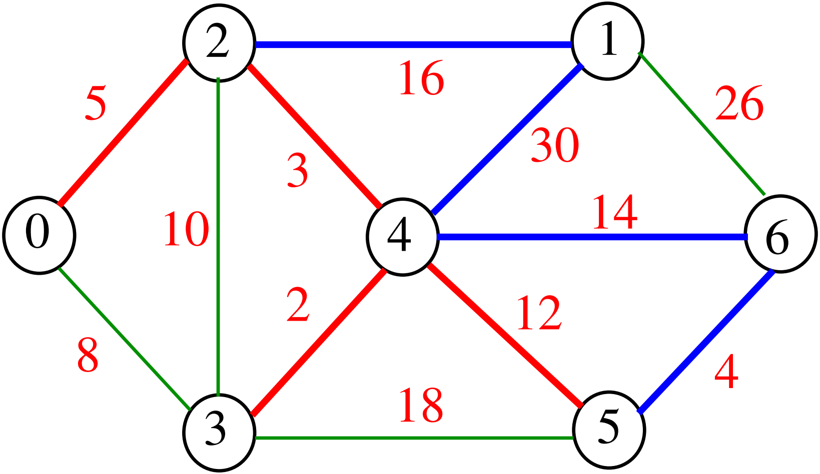
Simulação



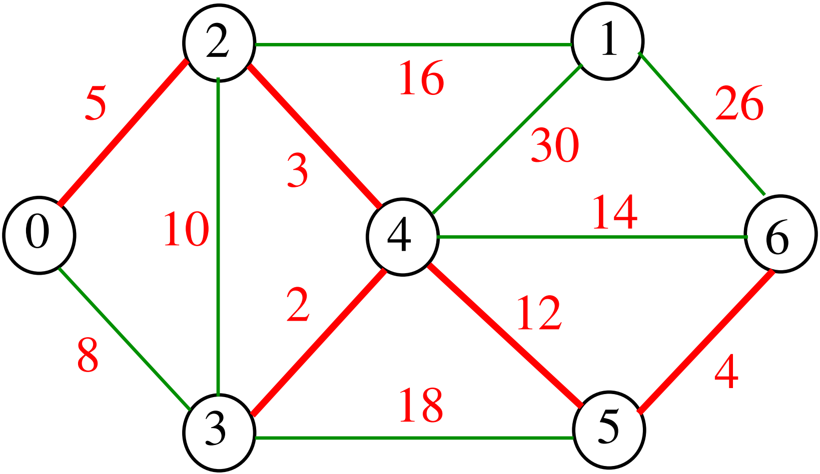
Simulação



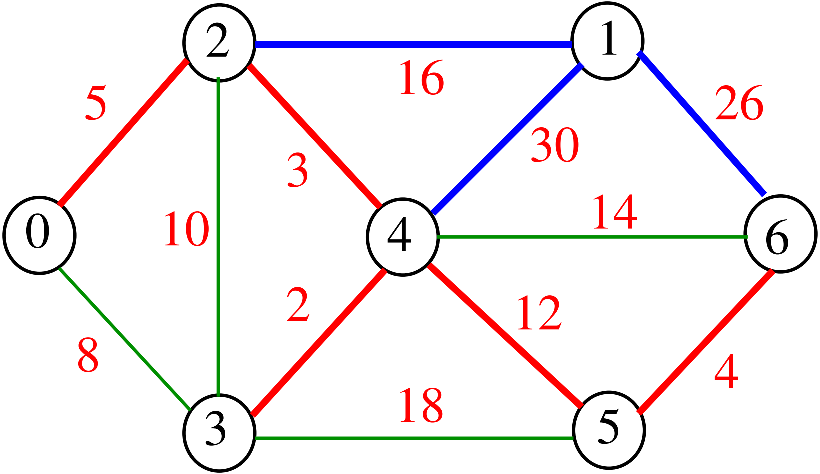
Simulação



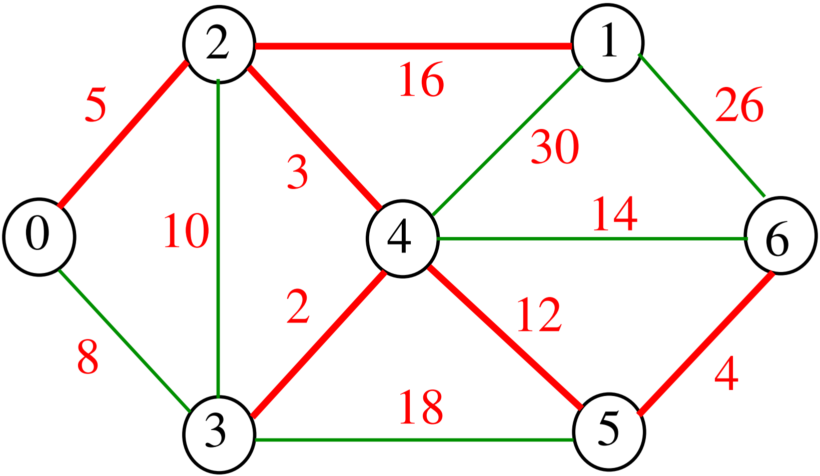
Simulação



Simulação



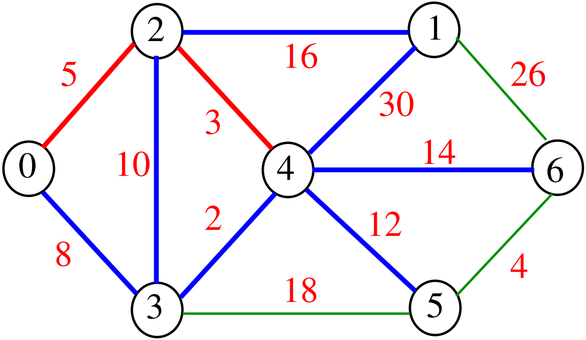
Simulação



Franja

A **franja** (= *fringe*) de uma subárvore `T` é o conjunto de todas as arestas que têm uma ponta em `T` e outra ponta fora

Exemplo: As arestas em azul formam a franja de `T`



Algoritmo de Prim

O algoritmo de Prim é iterativo.

Cada iteração começa com uma subárvore `T` de `G`.

No início da primeira iteração `T` é um árvore com apenas 1 vértice.

Cada iteração consiste em:

Caso 1: franja de `T` é vazia
Devolva `T` e pare.

Caso 2: franja de `T` não é vazia
Seja e uma aresta de custo mínimo na franja de `T`
Faça $T \leftarrow T + e$

Relação invariante chave

No início de cada iteração vale que

existe uma MST que contém as arestas em T.

Se a relação vale no **início 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 `T+e` no papel de `T`.

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 $\mathtt{M}$ que contém $\mathtt{T}$.

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

Seja $\mathtt{t}$ uma aresta que está $C(\mathtt{M}, \mathtt{e})$ que está na franja de $\mathtt{T}$. Pela escolha de $\mathtt{e}$ feita pelo algoritmo, $\mathtt{custo}(\mathtt{e}) \leq \mathtt{custo}(\mathtt{t})$.

Portanto, $\mathtt{M}-\mathtt{t}+\mathtt{e}$ é uma MST que contém $\mathtt{T}+\mathtt{e}$.

Implementações do algoritmo de Prim

S 20.3

Implementação grosseira

A função abaixo recebe um grafo G com custos nas arestas e calcula uma MST da componente que contém o vértice **0**.

```
void brutforcePrim(Graph G, Vertex parnt[]){
    Vertex v, w;
    for (v=0;v < G->V;v++) parnt[v] = -1;
    parnt[0] = 0;
```

Implementação grosseira

```
while (1) {
    double mincst = INFINITO;
    Vertex v0, w0;
    for (w = 0; w < G->V; w++)
        if (parnt[w] == -1)
        for (v=0; v < G->V; v++)
            if (parnt[v] != -1
                && mincst > G->adj[v][w])
                mincst = G->adj[v0=v][w0=w];
    if (mincst == INFINITO) break;
    parnt[w0] = v0;
    }
}
```

Implementações eficientes

Implementações eficientes do algoritmo de Prim dependem do conceito de **custo de um vértice** em relação a uma árvore.

Dada uma árvore não-geradora do grafo, o **custo de um vértice** `w` que está fora da árvore é o custo de uma aresta mínima dentre as que incidem em `w` e estão na franja da árvore.

Se nenhuma aresta da franja incide em `w`, o custo de `w` é `INFINITO`.

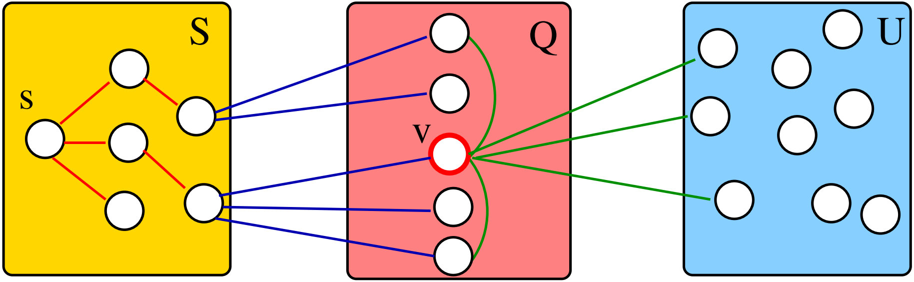
Implementações eficientes

Nas implementações que examinaremos, o custo do vértice `w` em relação à árvore é `cst[w]`.

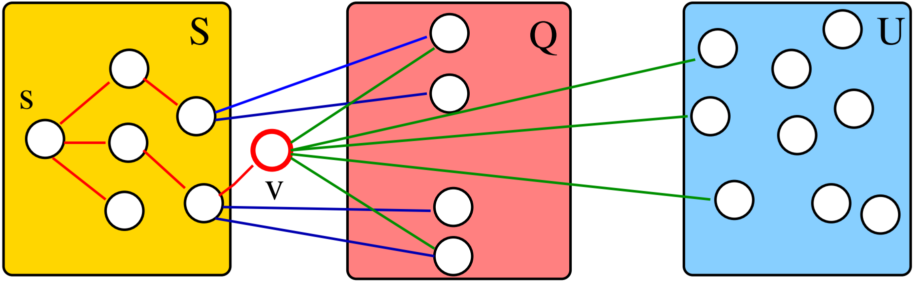
Para cada vértice `w` fora da árvore, o vértice `fr[w]` está na árvore e a aresta que liga `w` a `fr[w]` tem custo `cst[w]`.

Cada iteração do algoritmo de Prim escolhe um vértice `w` fora da árvore e adota `fr[w]` como valor de `parnt[w]`.

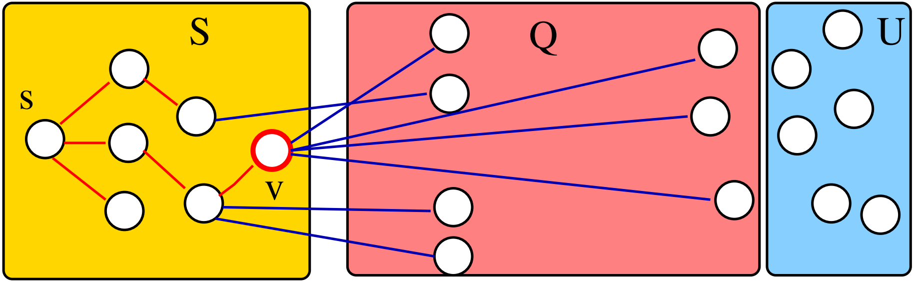
Iteração



Iteração



Iteração



Implementação eficiente para grafos densos

Recebe grafo G com custos nas arestas e calcula uma MST da componente de G que contém o vértice 0.

A função armazena a MST no vetor parnt, tratando-a como uma arborescência com raiz 0.

O grafo G é representado por sua matriz de adjacência.

```
void GRAPHmstP1 (Graph G, Vertex parnt[]){
    double cst[maxV]; Vertex v, w, fr[maxV];
    for (v= 0; v< G->V; v++) {
        parnt[v] = -1;
        cst[v] = INFINITO;
    }
    v= 0;    fr[v] = v;    cst[v] = 0;
```

```
while (1) {
double mincst= INFINITO;
for (w = 0; w < G->V; w++)
      if (parnt[w] == -1 && mincst> cst[w])
            mincst = cst[v=w];
if (mincst == INFINITO) break;
parnt[v] = fr[v];
for (w = 0; w < G->V; w++)
      if (parnt[w] == -1
            && cst[w] > G->adj[v][w]) {
            cst[w] = G->adj[v][w];
            fr[w] = v;
      }
}
}
```

Consumo de tempo

O consumo de tempo da função `GRAPHmstP1` é $\mathrm{O}(\mathtt{V}^2)$.

Este consumo de tempo é ótimo para **digrafos densos**.

Recordando Dijkstra para digrafos densos

```
#define INFINITO maxCST

void
DIGRAPHsptD1 (Digraph G, Vertex s,
              Vertex parnt[], double cst[]) {
    Vertex w, w0, fr[maxV];
    for (v = 0; v < G->V; v++) {
        parnt[v] = -1;
        cst[v] = INFINITO;
    }
    fr[s] = s;
    cst[s] = 0;
```

```
while (1) {
   double mincst = INFINITO;
   for (w = 0; w < G->V; w++)
        if (parnt[w]==-1 && mincst>cst[w])
             mincst = cst[v=w];
   if (mincst == INFINITO) break;
   parnt[v] = fr[v];
   for (w = 0; w < G->V; w++)
        if(cst[w]>cst[v]+G->adj[v][w]){
             cst[w] = cst[v]+G->adj[v][w];
             fr[w] = v;
        }
   }
}
```

Implementação para grafos esparsos

Recebe grafo `G` com custos nas arestas e calcula uma MST da componente de `G` que contém o vértice 0.

A função armazena a MST no vetor `parnt`, tratando-a como uma arborescência com raiz 0.

O grafo `G` é representado por listas de adjacência.

GRAPHmstP2

```
#define INFINITO maxCST
void GRAPHmstP2 (Graph G, Vertex parnt[]){
    Vertex v, w, fr[maxV]; link p;
    for (v = 0; v < G->V; v++) {
        cst[v] = INFINITO;
        parnt[v] = -1;
    }
    PQinit(G->V);
    cst[0] = 0;
    fr[0] = 0;
    PQinsert(0);
```

```
while (!PQempty()) {
v = PQdelmin();
parnt[v] = fr[v];
for (p=G->adj[v];p!=NULL;p=p->next){
    w = p->w;
    if (parnt[w] == -1){
        if (cst[w] == INFINITO){
            cst[w] = p->cst;
            fr[w] = v;
            PQinsert(w);
        }
```

```
            else if (cst[w] > p->cst){
                cst[w] = p->cst;
                fr[w] = v;
                PQdec(w);
            }
        } /* if (parnt[w] ...*/
    } /* for (p...*/
    } /* while ...
}
```

Consumo de tempo

O consumo de tempo da função `GRAPHmstP2` implementada com um min-heap é $O(A \lg V)$.

Recordando Dijkstra para digrafos esparsos

```
#define INFINITO maxCST
void dijkstra(Digraph G, Vertex s,
        Vertex parnt[], double cst[]);
{
    Vertex v, w; link p;
    for (v = 0; v < G->V; v++) {
        cst[v] = INFINITO;
        parnt[v] = -1;
    }
    PQinit(G->V);
    cst[s] = 0;
    parnt[s] = s;
    PQinsert(s);
```

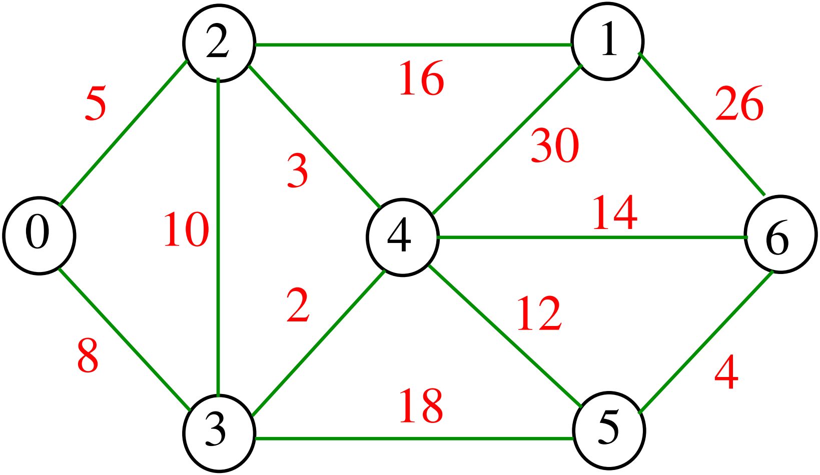
```
while (!PQempty()) {
    v = PQdelmin();
    for(p=G->adj[v];p!=NULL;p=p->next)
        w = p->w;
        if (cst[w]==INFINITO) {
            cst[w]=cst[v]+p->cst;
            parnt[w]=v;
            PQinsert(w);
        }
```

```
            else
            if(cst[w]>cst[v]+p->cst)
                cst[w]=cst[v]+p->cst
                parnt[w] = v;
                PQdec(w);
            }
    PQfree();
    }
}
```

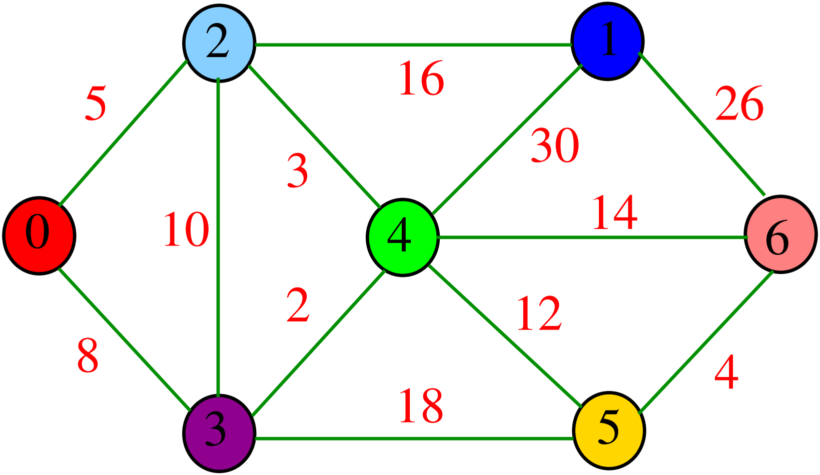
Algoritmo de Kruskal

S 20.3

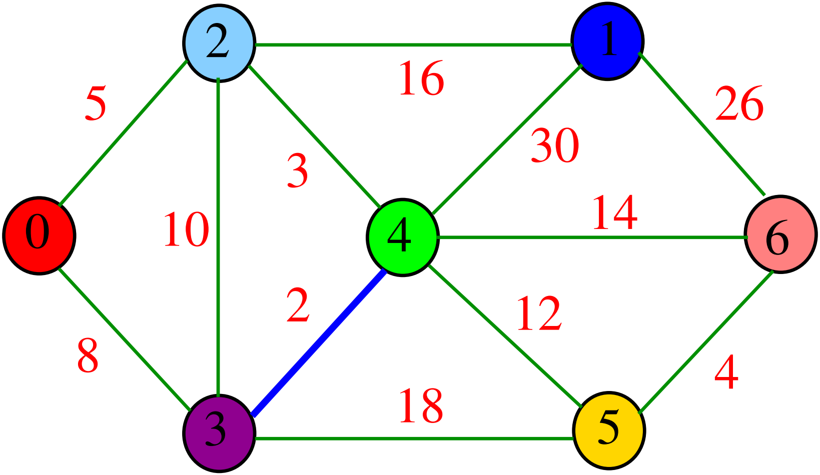
Algoritmo de Kruskal



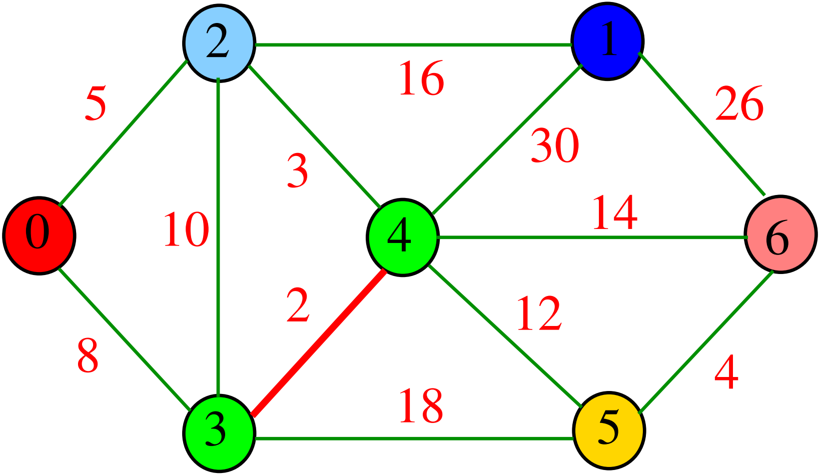
Algoritmo de Kruskal



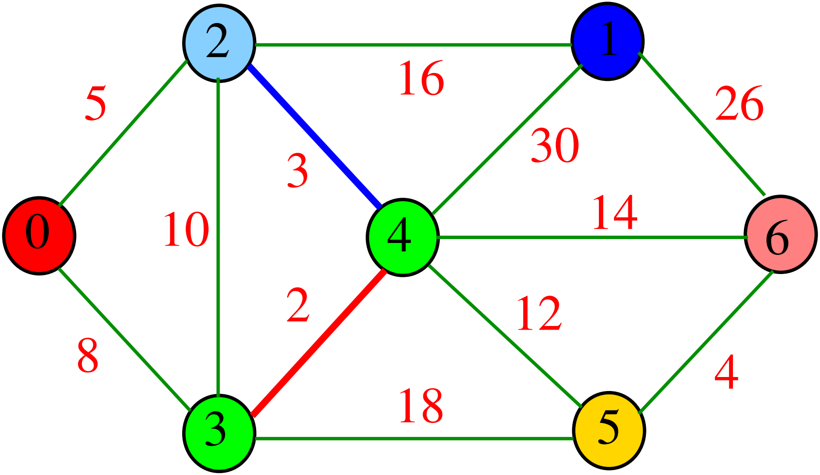
Algoritmo de Kruskal



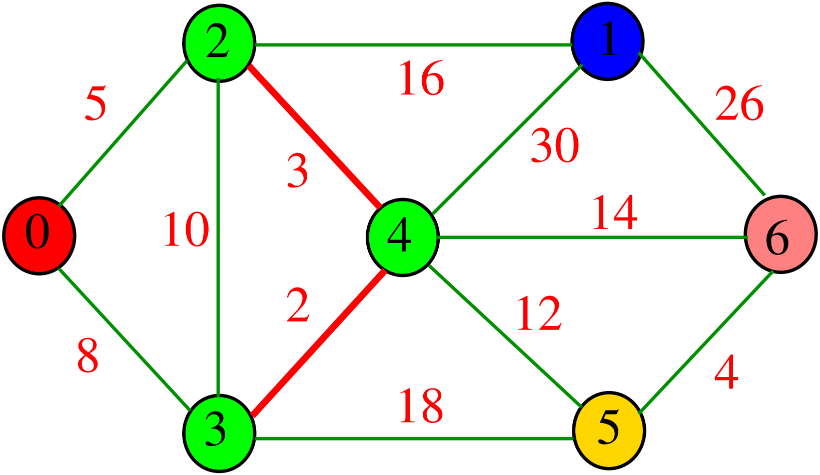
Algoritmo de Kruskal



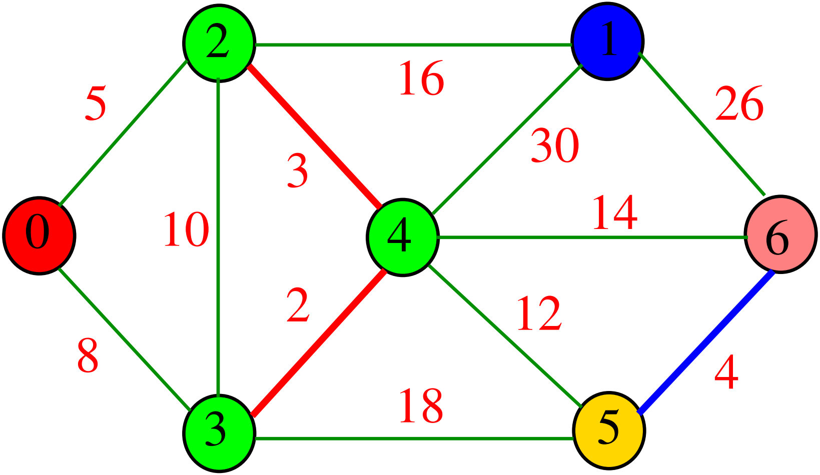
Algoritmo de Kruskal



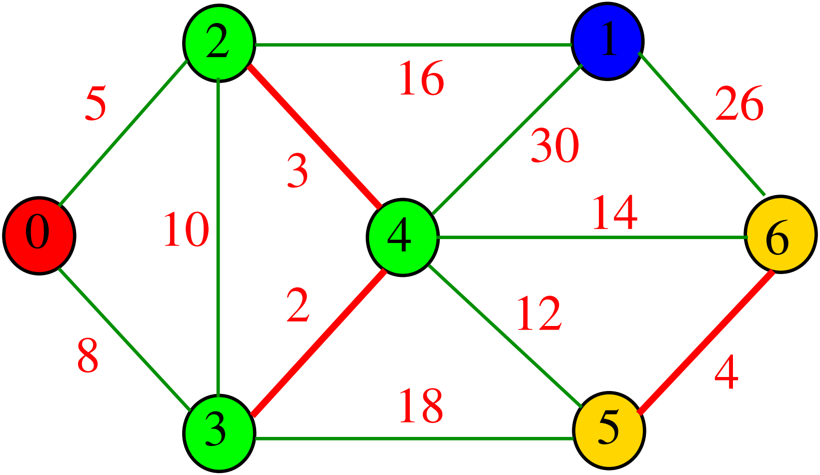
Algoritmo de Kruskal



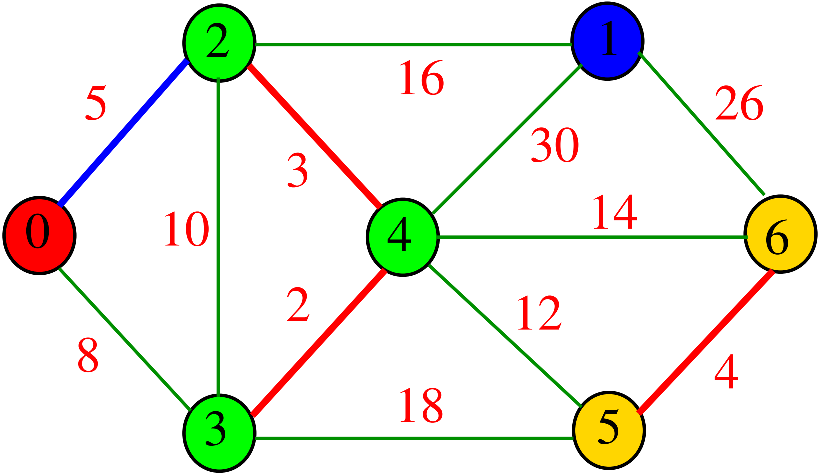
Algoritmo de Kruskal



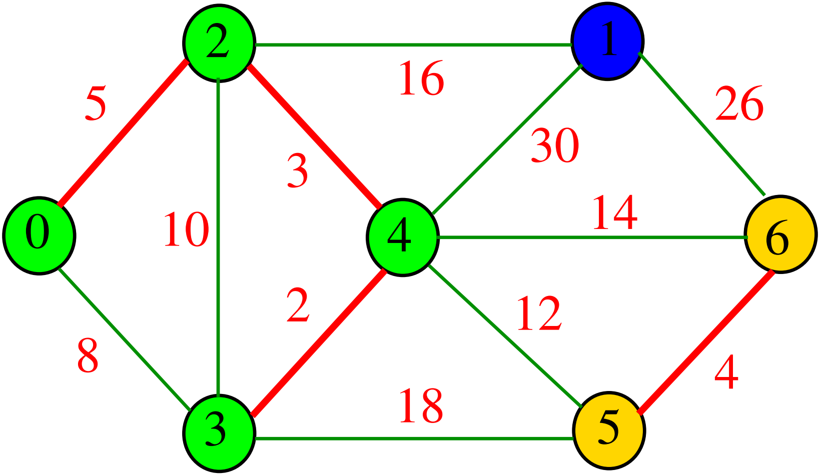
Algoritmo de Kruskal



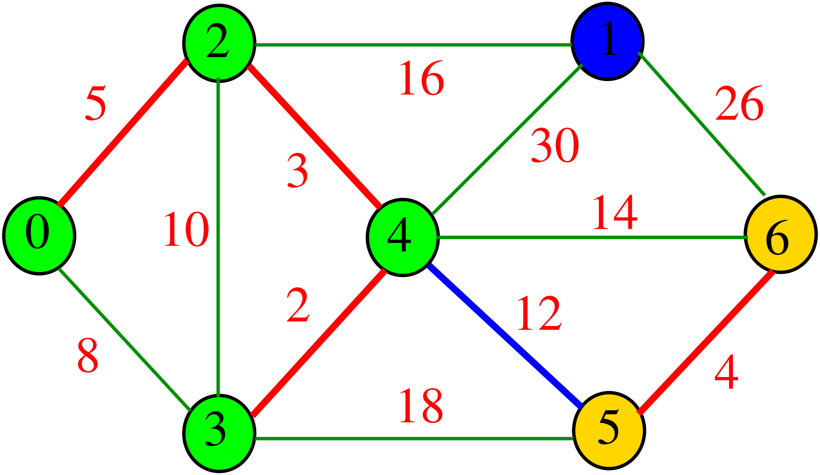
Algoritmo de Kruskal



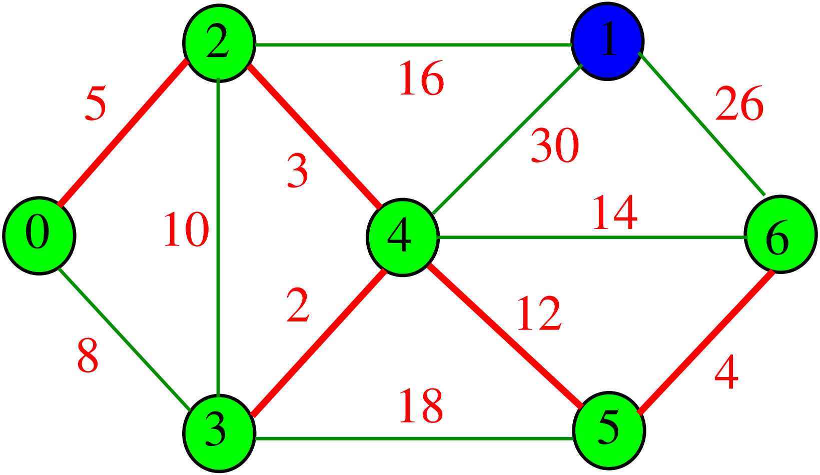
Algoritmo de Kruskal



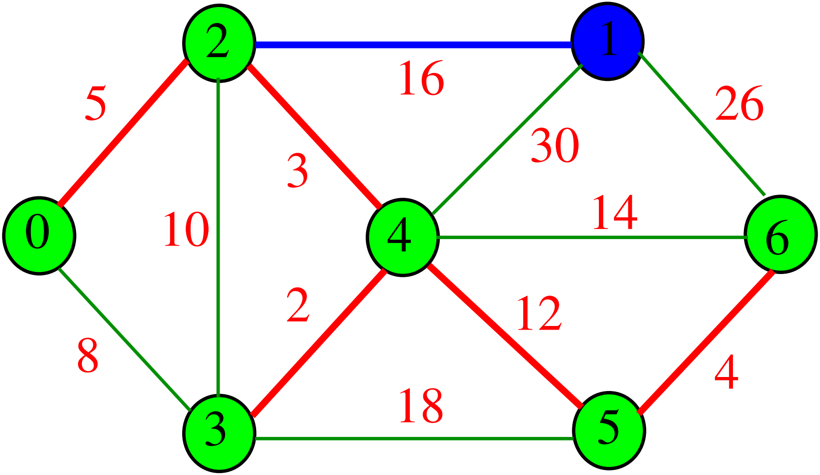
Algoritmo de Kruskal



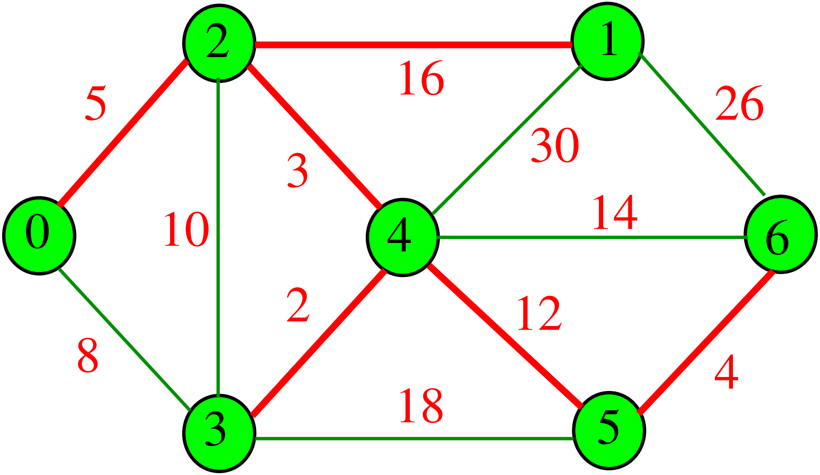
Algoritmo de Kruskal



Algoritmo de Kruskal



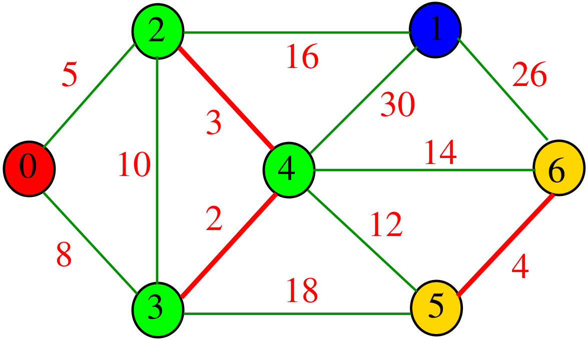
Algoritmo de Kruskal



Subfloresta

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

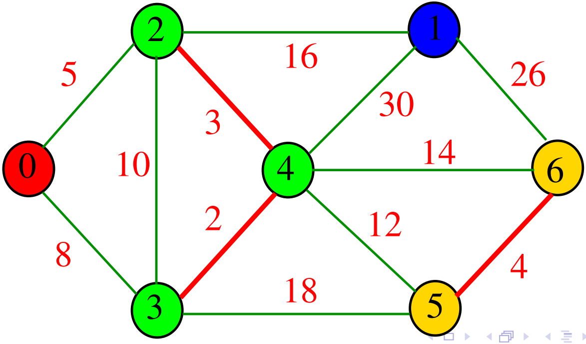
Exemplo: As arestas vermelhas que ligam os vértices verdes e amarelos e formam uma subfloresta



Floresta geradora

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

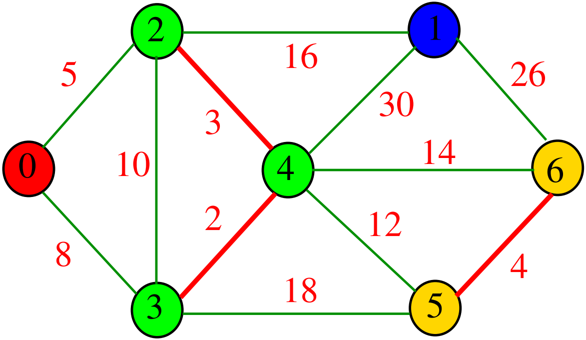
Exemplo: Arestas vermelhas ligando os vértices verdes e amarelos, junto com os vértices azul e vermelho, forma uma floresta geradora



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 $\mathtt{F}$.

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

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* $C(F, e)$.