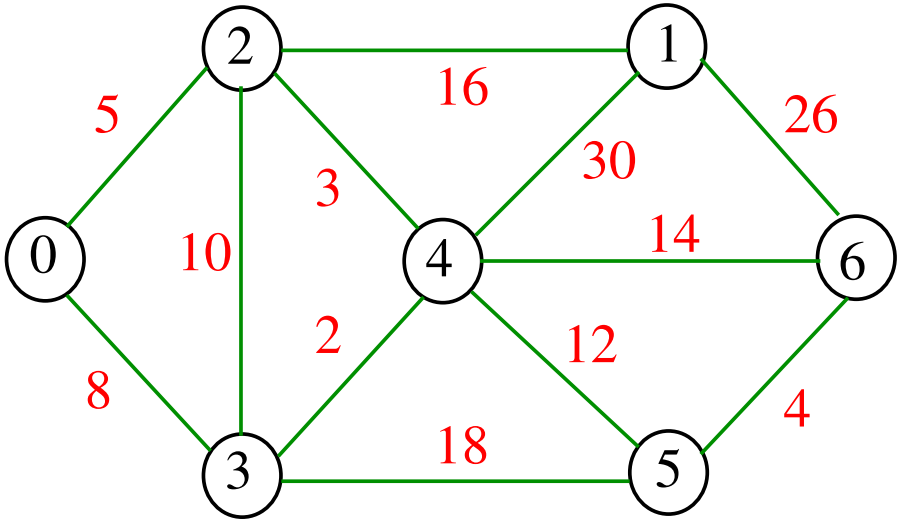
Algoritmo de Prim

S 20.3

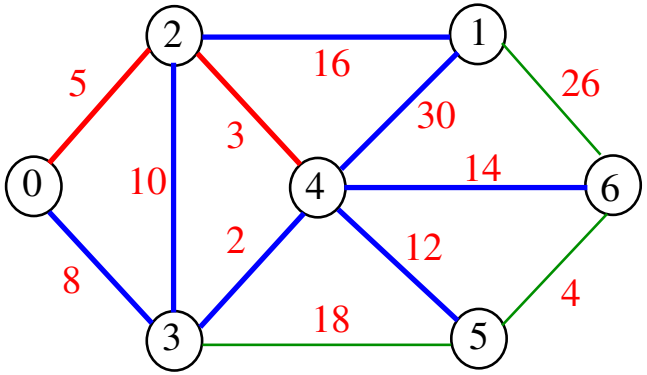
Simulação



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 e a aresta escolhida pela iteração no caso 2. Pela relação invariante existe uma MST M que contém T.

Se e está em M, então não há o que demonstrar. Suponha, portanto, que e não está em M.

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

Portanto, M−t+e é uma MST que contém T+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 bruteforcePrim(Graph G, Vertex parnt[]){
 0   Vertex v, w;
 1   for (v=0;v < G->V;v++) parnt[v] = -1;
 3   parnt[0] = 0;
```

Implementação grosseira

```
 4   while (1) {
 5   double mincst = INFINITO;
 6   Vertex v0, w0;
 7   for (w = 0; w < G->V; w++)
 8       if (parnt[w] == -1)
 9       for (v=0; v < G->V; v++)
10           if (parnt[v] != -1
                 && mincst > G->adj[v][w])
11           mincst = G->adj[v0=v][w0=w];
12   if (mincst == INFINITO) break;
13   parnt[w0] = v0;
14   }
}
```


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`.

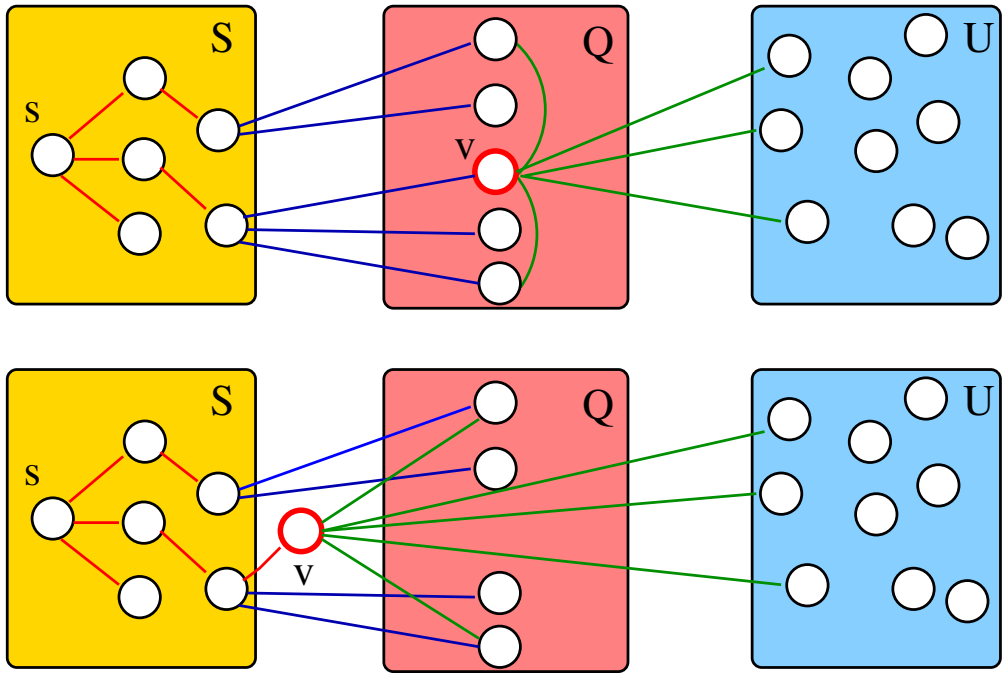
Implementações eficientes

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



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



```
6   while (1) {
7   double mincst= INFINITO;
8   for (w = 0; w < G->V; w++)
9       if (parnt[w] == -1 && mincst> cst[w])
10          mincst = cst[v=w];
11  if (mincst == INFINITO) break;
12  parnt[v] = fr[v];
13  for (w = 0; w < G->V; w++)
14      if (parnt[w] == -1
            && cst[w] > G->adj[v][w]) {
15          cst[w] = G->adj[v][w];
16          fr[w] = v;
        }
    }
}
```


Consumo de tempo

O consumo de tempo da função GRAPHmstP1 é $O(V^2)$.

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[]) {
1   Vertex w, w0, fr[maxV];
2   for (v = 0; v < G->V; v++) {
3       parnt[v] = -1;
4       cst[v] = INFINITO;
5   }
6   fr[s] = s;
7   cst[s] = 0;
```

```
 8 while (1) {
 9   double mincst = INFINITO;
10   for (w = 0; w < G->V; w++)
11       if (parnt[w]==-1 && mincst>cst[w])
12           mincst = cst[v=w];
13   if (mincst == INFINITO) break;
14   parnt[v] = fr[v];
15   for (w = 0; w < G->V; w++)
16       if(cst[w]>cst[v]+G->adj[v][w]){
17           cst[w] = cst[v]+G->adj[v][w];
18           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[]){
1   Vertex v, w, fr[maxV]; link p;
2   for (v = 0; v < G->V; v++) {
3       cst[v] = INFINITO;
4       parnt[v] = -1;
    }
5   PQinit(G->V);
6   cst[0] = 0;
7   fr[0] = 0;
8   PQinsert(0);
```



```
 9   while (!PQempty()) {
10   v = PQdelmin();
11   parnt[v] = fr[v];
12   for (p=G->adj[v];p!=NULL;p=p->next){
13       w = p->w;
14       if (parnt[w] == -1){
15           if (cst[w] == INFINITO){
16               cst[w] = p->cst;
17               fr[w] = v;
18               PQinsert(w);
             }
```

```
19          else if (cst[w] > p->cst){
20              cst[w] = p->cst;
21              fr[w] = v;
22              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[]);
{
1   Vertex v, w; link p;
2   for (v = 0; v < G->V; v++) {
3       cst[v] = INFINITO;
4       parnt[v] = -1;
    }
5   PQinit(G->V);
6   cst[s] = 0;
7   parnt[s] = s;
8   PQinsert(s);
```



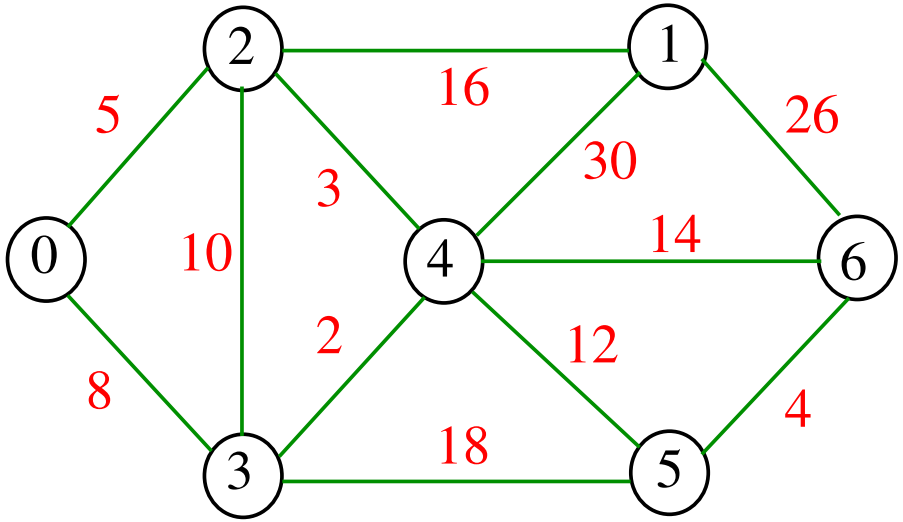
```
 9  while (!PQempty()) {
10      v = PQdelmin();
11      for(p=G->adj[v];p!=NULL;p=p->next)
12          w = p->w;
12          if (cst[w]==INFINITO) {
13              cst[w]=cst[v]+p->cst;
14              parnt[w]=v;
15              PQinsert(w);
            }
```

```
16          else
17          if(cst[w]>cst[v]+p->cst)
18              cst[w]=cst[v]+p->cst
19              parnt[w] = v;
20              PQdec(w);
            }
21  PQfree();
    }
}
```


Algoritmo de Kruskal

S 20.3

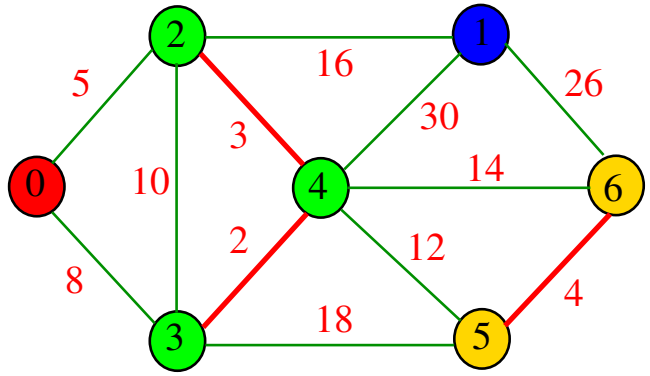
Algoritmo de Kruskal



Subfloresta

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

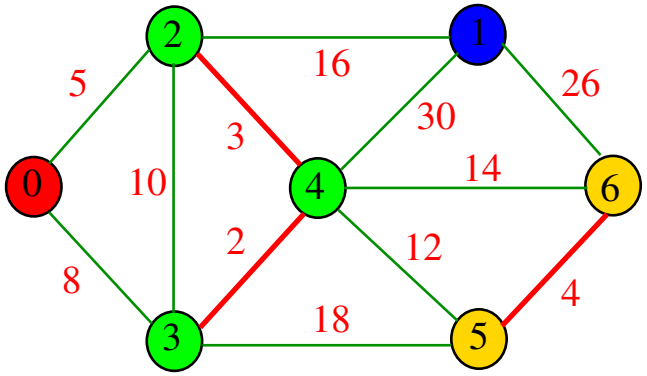
Exemplo: As arestas vermelhas que ligam os vértices verdes e amarelos e formam uma subfloresta



Floresta geradora

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

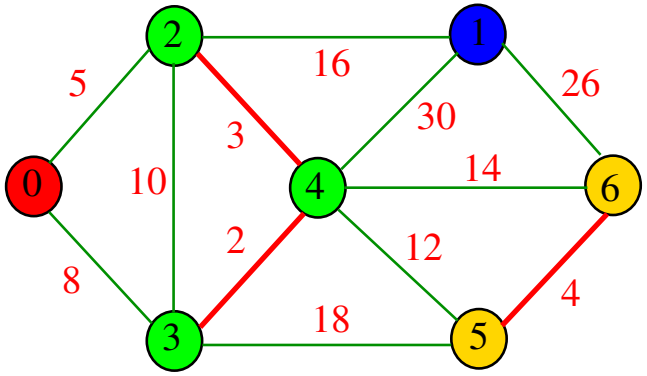
Exemplo: Arestas vermelhas ligando os vértices verdes e amarelos, junto com os vértices azul e vermelho, forma uma floresta geradora



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 $C(F, e)$.