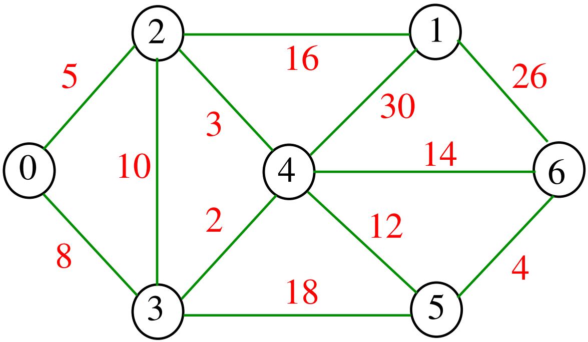
Melhores momentos

AULA 20

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

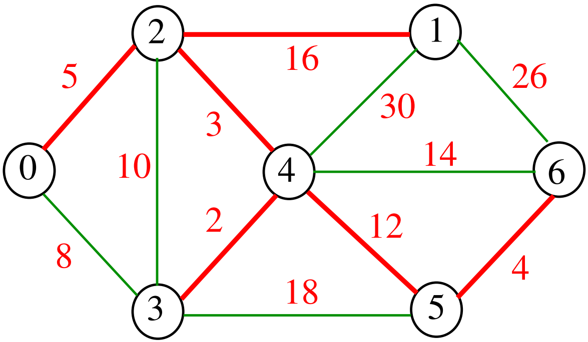
Exemplo: um grafo com custos nas aretas



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: MST de custo 42

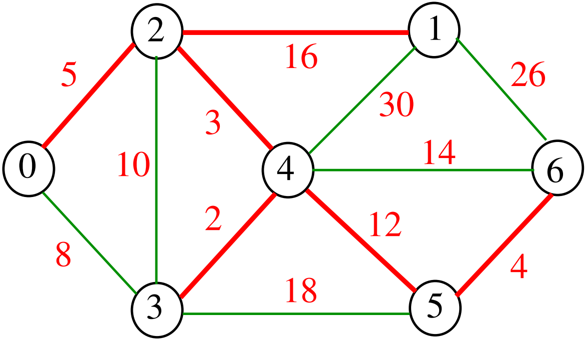


Problema MST

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

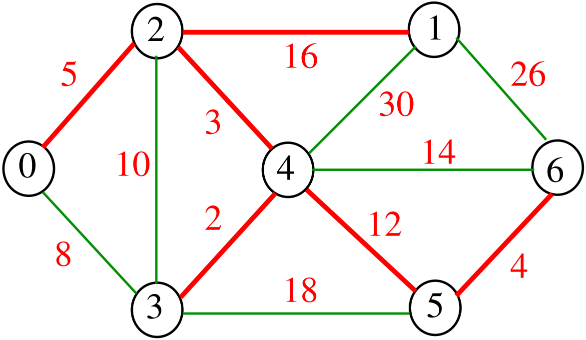
Exemplo: MST de custo 42



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 único ciclo não-trivial em T+e

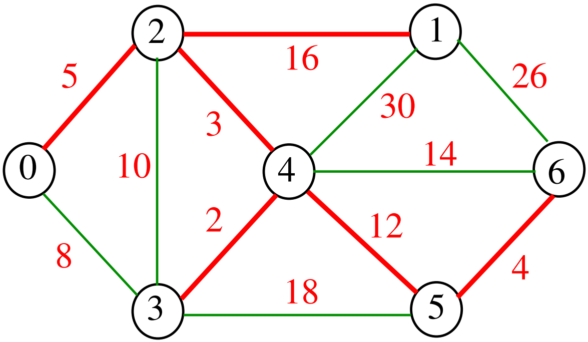
Exemplo: MST de custo 42



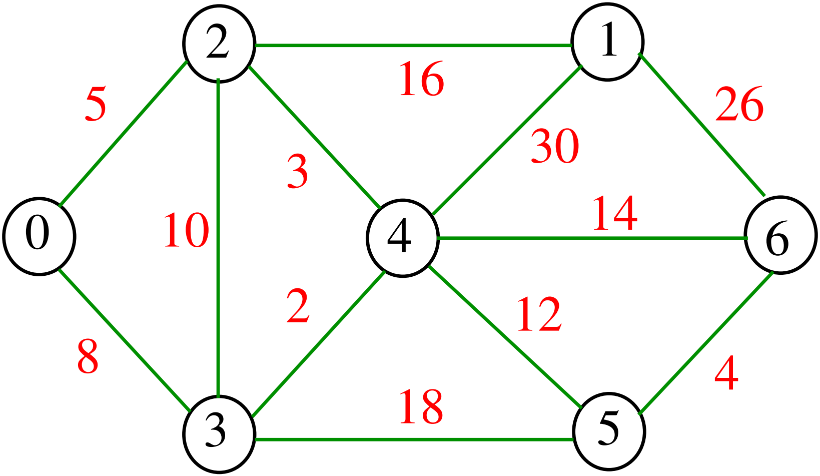
Propriedade dos cortes

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`

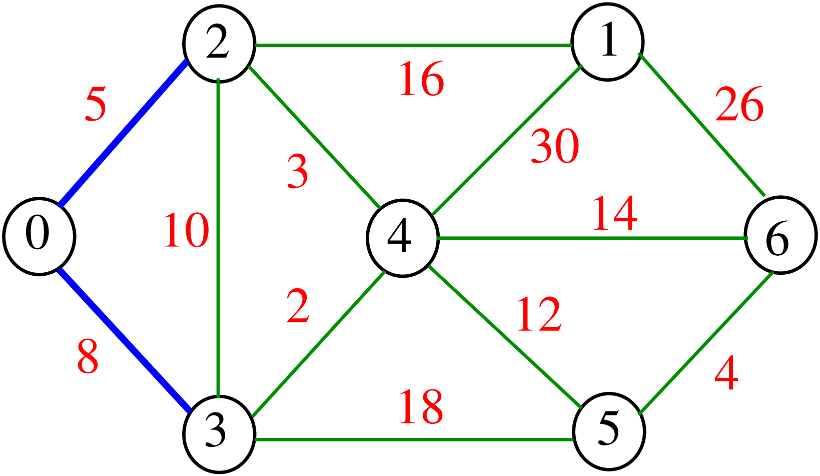
Exemplo: MST de custo 42



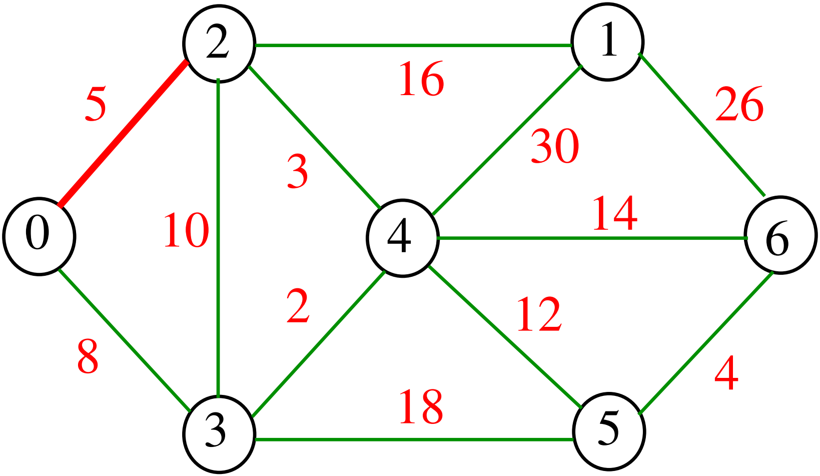
Algoritmo de Prim



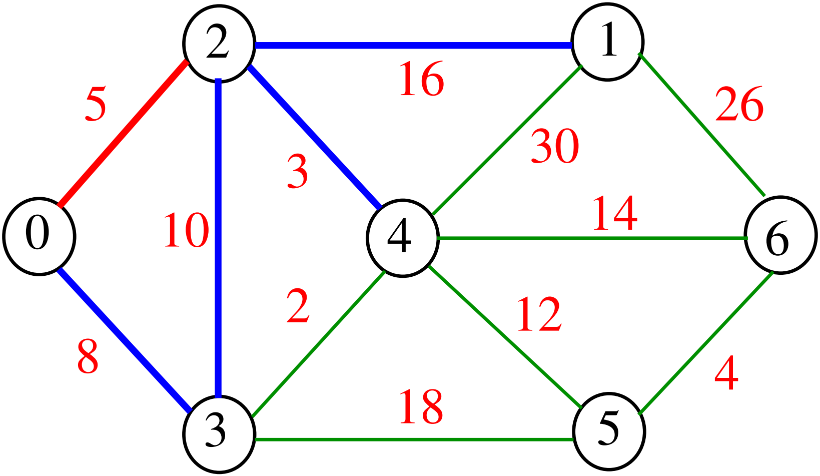
Algoritmo de Prim



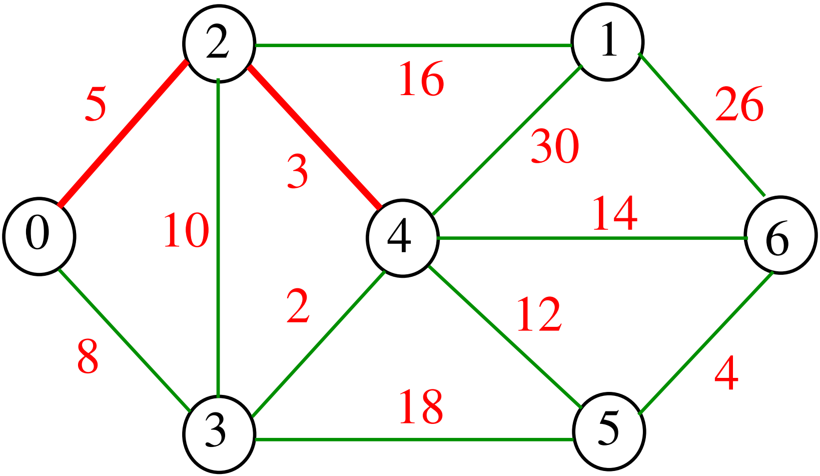
Algoritmo de Prim



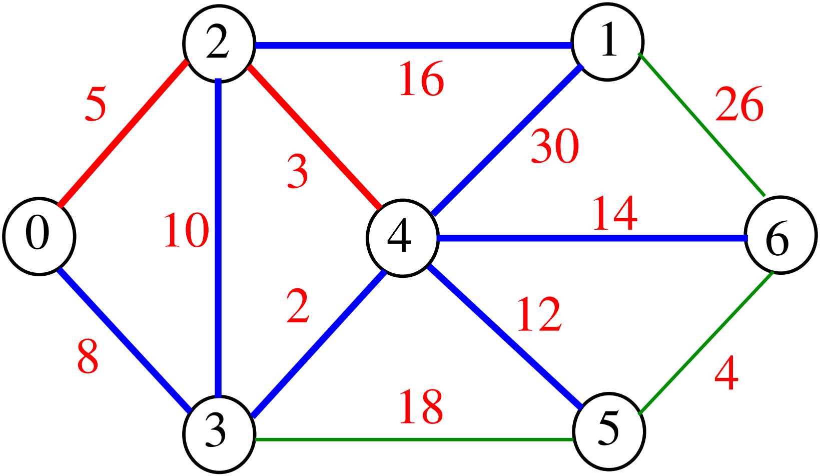
Algoritmo de Prim



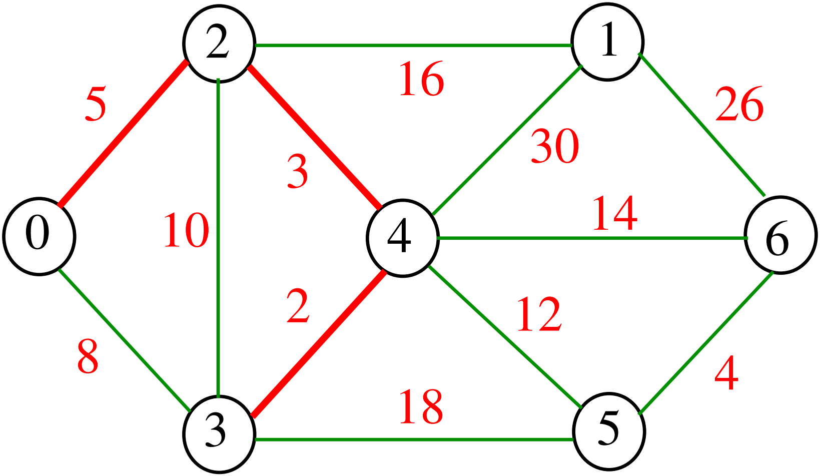
Algoritmo de Prim



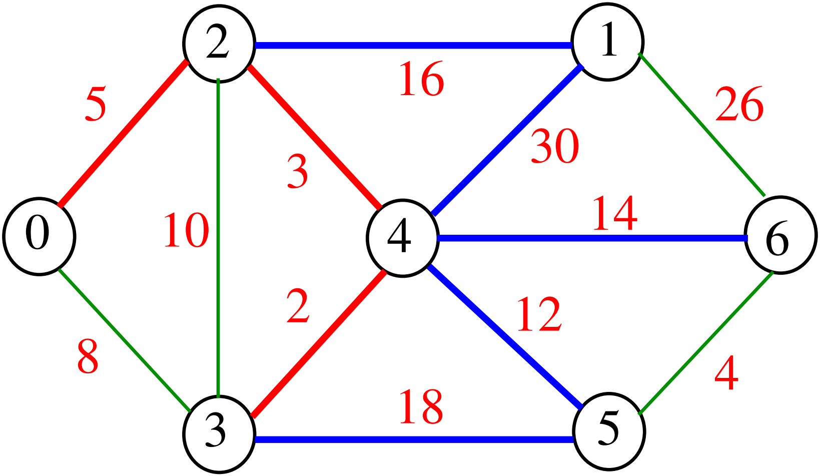
Algoritmo de Prim



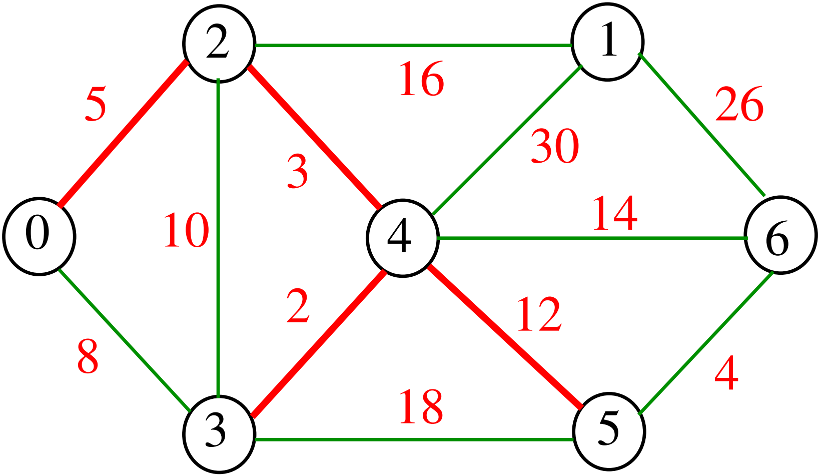
Algoritmo de Prim



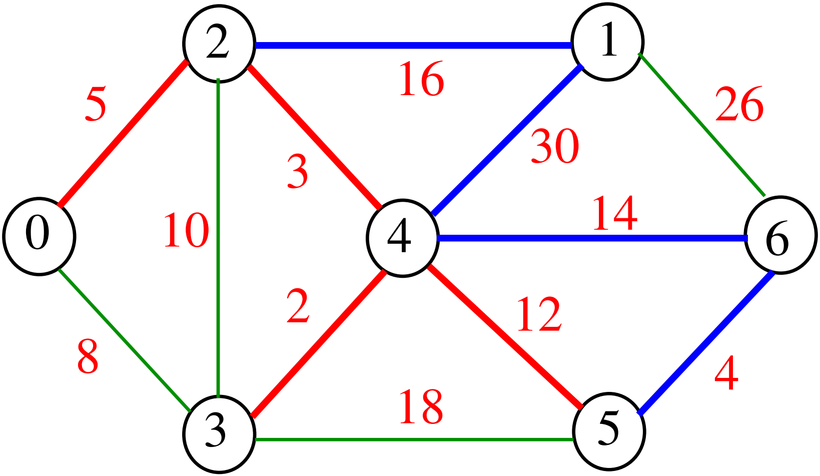
Algoritmo de Prim



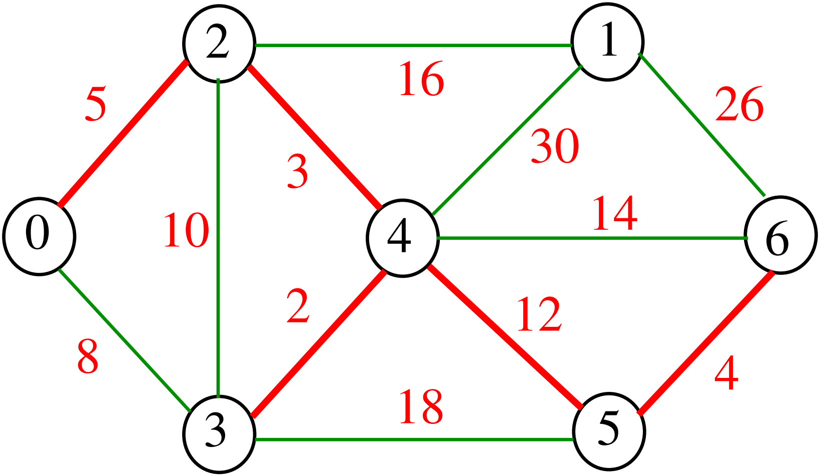
Algoritmo de Prim



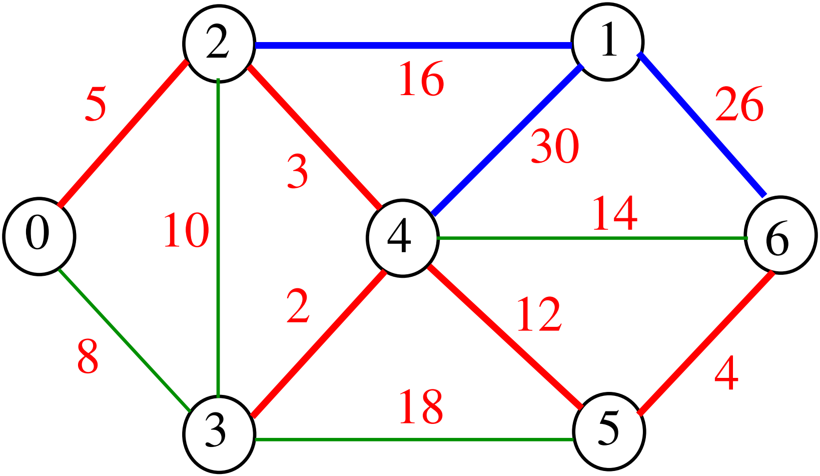
Algoritmo de Prim



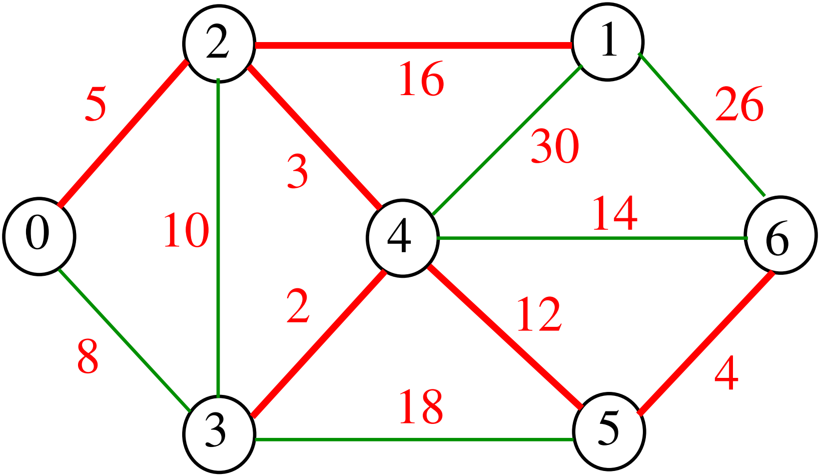
Algoritmo de Prim



Algoritmo de Prim



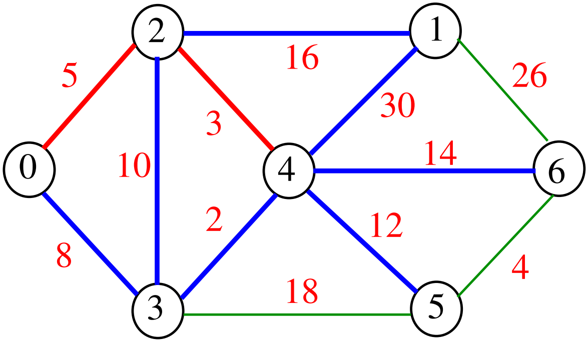
Algoritmo de Prim



Franja

A **franja** (= *fringe*) de uma subárvore não-geradora `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`
Comece nova iteração com `T+e` no papel de `T`

Relação invariante chave

No início de cada iteração vale que

> *existe uma MST que contém as arestas em T.*

AULA 21

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

Implementação grosseira

```
while (1) {
double mincst = maxCST;
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 == maxCST) 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 `maxCST`.

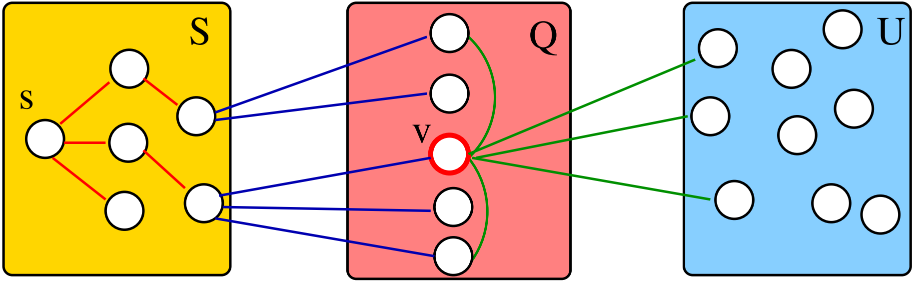
Implementações eficientes

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

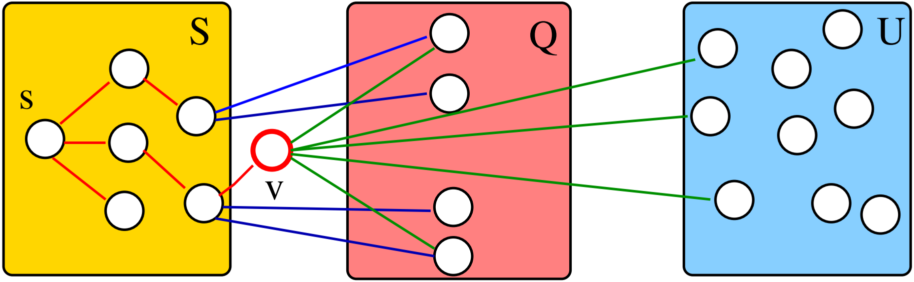
Para cada vértice `w` fora da árvore, o vértice `fr[w]` está na árvore e a aresta que liga `w` a `fr[w]` tem custo `cst[w]`.

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

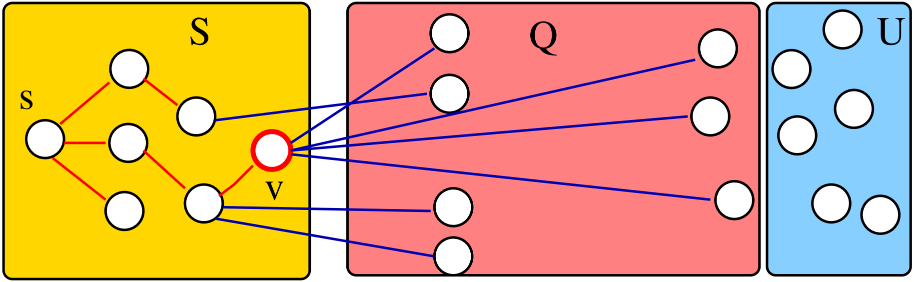
Iteração



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] = maxCST;
    }
    v= 0;   fr[v] = v;   cst[v] = 0;
```

```
while (1) {
double mincst= maxCST;
for (w = 0; w < G->V; w++)
    if (parnt[w] == -1 && mincst> cst[w])
        mincst = cst[v=w];
if (mincst == maxCST) 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 $O(V^2)$.

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

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.

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

```
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 (fr[w] == -1){
            cst[w] = p->cst;    fr[w] = v;
            PQinsert(w);
        } else if (cst[w] > p->cst){
            cst[w] = p->cst;    fr[w] = v;
            PQdec(w);
        }
    }
}
```

Consumo de tempo

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