

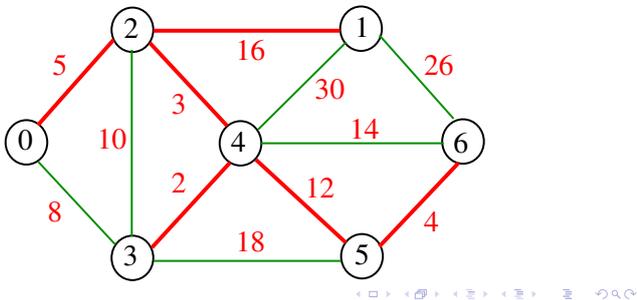
Melhores momentos

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**

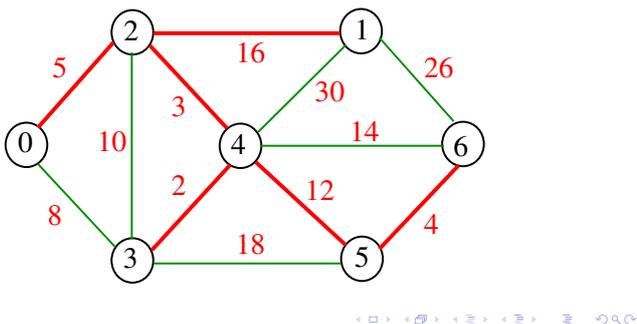
Exemplo: MST de custo 42



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 $T+e$

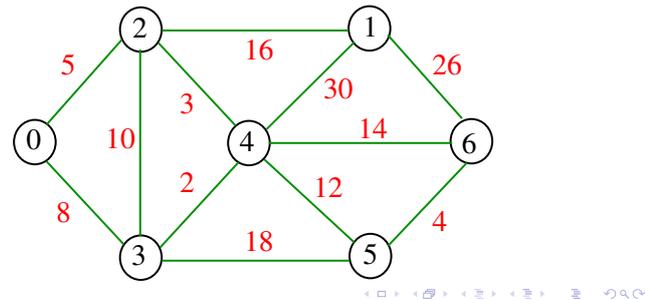
Exemplo: MST de custo 42



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

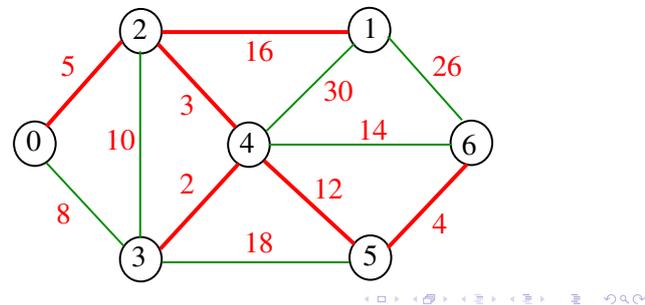


Problema MST

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

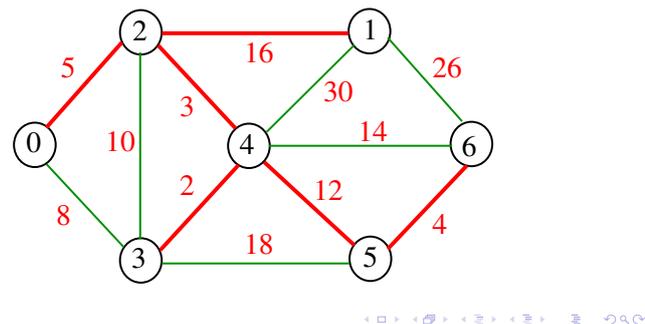
Exemplo: MST de custo 42



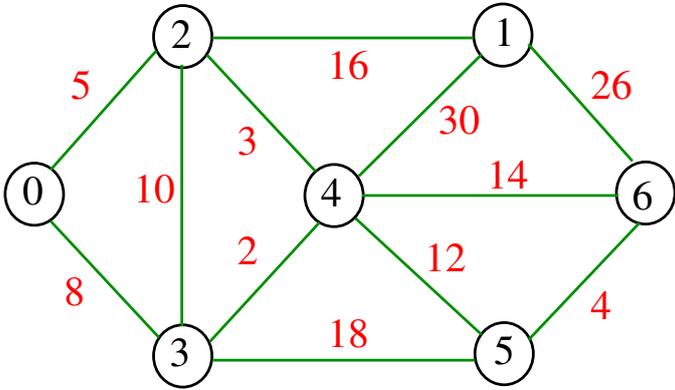
Propriedade dos cortes

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

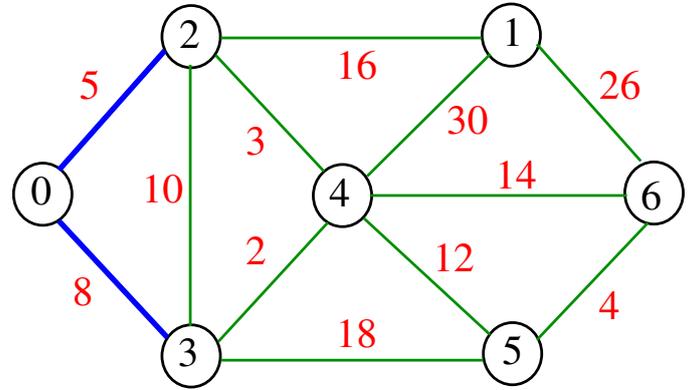


Algoritmo de Prim



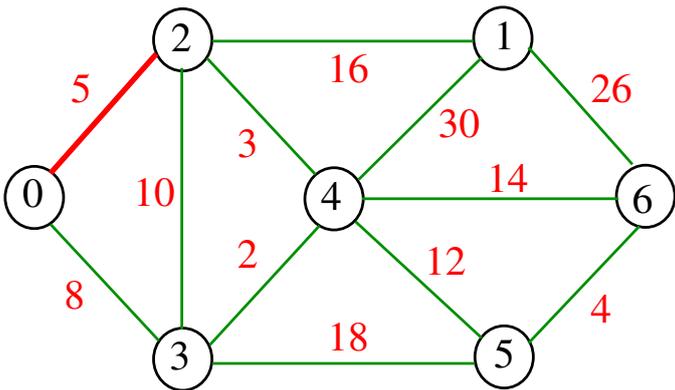
Navigation icons

Algoritmo de Prim



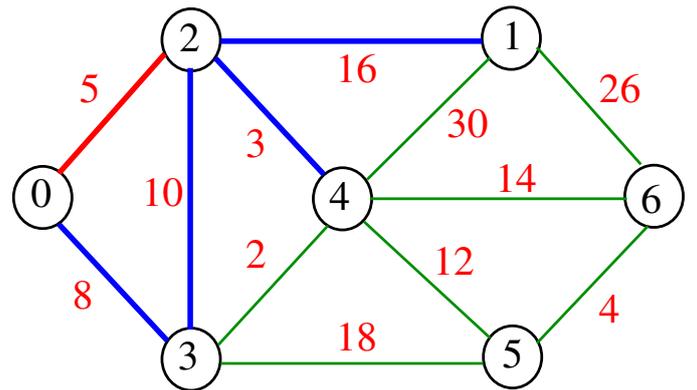
Navigation icons

Algoritmo de Prim



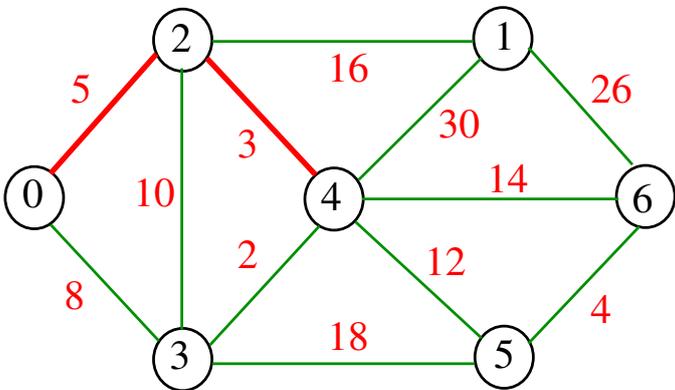
Navigation icons

Algoritmo de Prim



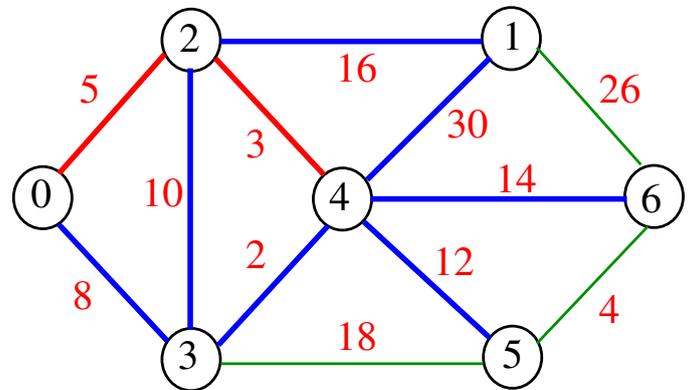
Navigation icons

Algoritmo de Prim



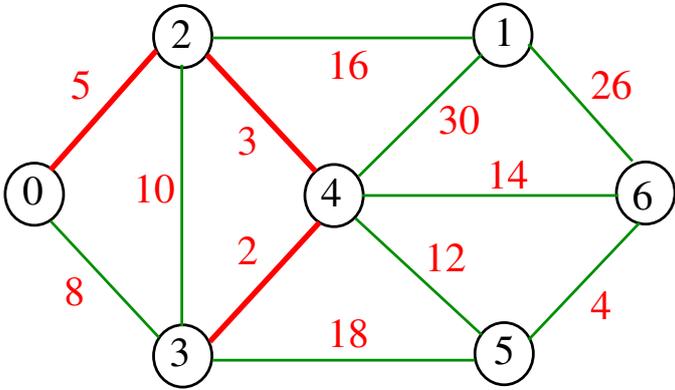
Navigation icons

Algoritmo de Prim



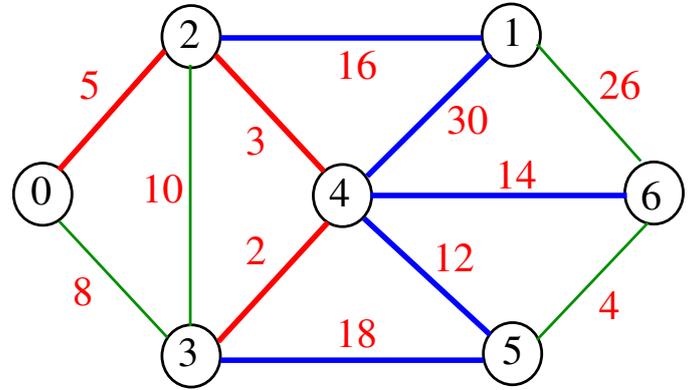
Navigation icons

Algoritmo de Prim



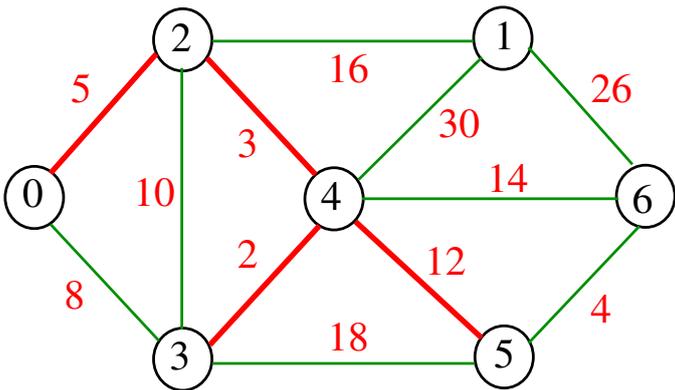
Navigation icons

Algoritmo de Prim



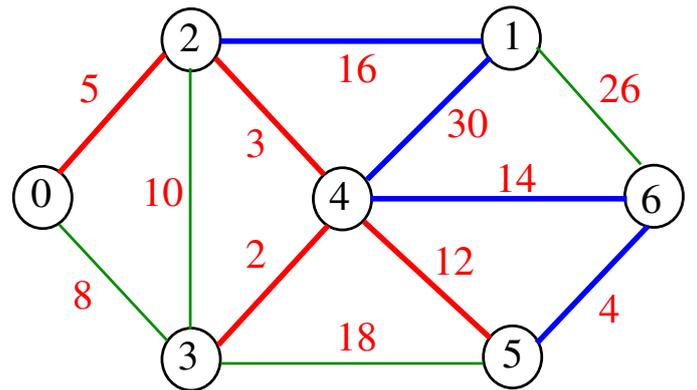
Navigation icons

Algoritmo de Prim



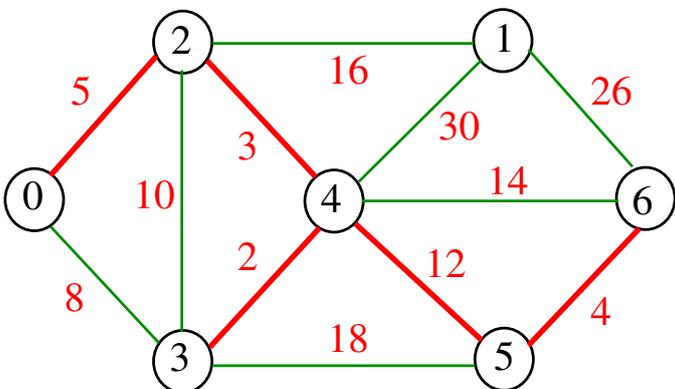
Navigation icons

Algoritmo de Prim



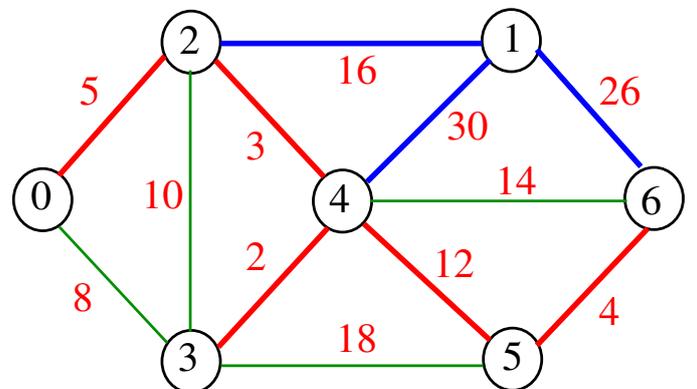
Navigation icons

Algoritmo de Prim



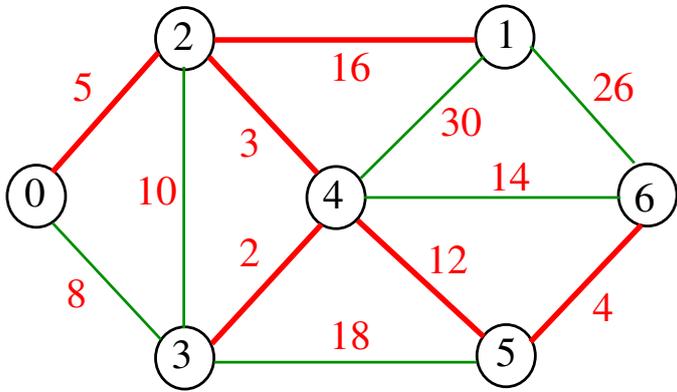
Navigation icons

Algoritmo de Prim



Navigation icons

Algoritmo de Prim



< > < > < > < > < > < >

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.

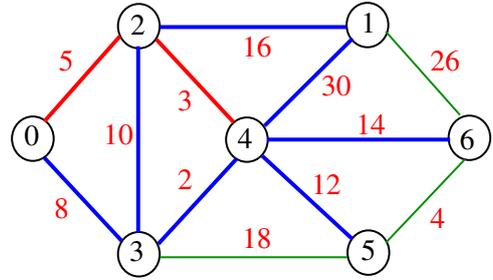
Caso 2: franja de T não é vazia
Seja e uma aresta de custo mínimo na franja de T
Comece nova iteração com $T+e$ no papel de T

< > < > < > < > < > < >

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



< > < > < > < > < > < >

Relação invariante chave

No início de cada iteração vale que
existe uma MST que contém as arestas em T .

< > < > < > < > < > < >

Implementações do algoritmo de Prim

< > < > < > < > < > < >

< > < > < > < > < > < >

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;
```

Navigation icons

Implementações eficientes

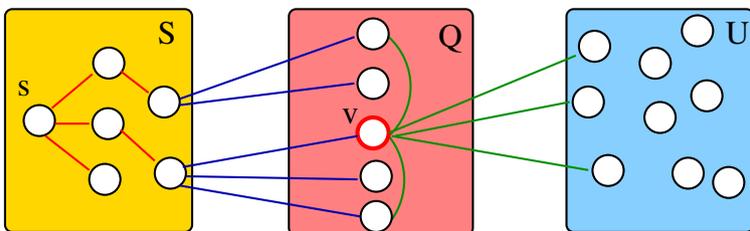
Implementações eficientes do algoritmo de Prim dependem do conceito de **custo de um vértice** em relação a uma árvore.

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.

Navigation icons

Iteração



Navigation icons

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
11                 && mincst > G->adj[v][w])
12                 mincst = G->adj[v0=v][w0=w];
12  if (mincst == INFINITO) break;
13  parnt[w0] = v0;
14  }
```

Navigation icons

Implementações eficientes

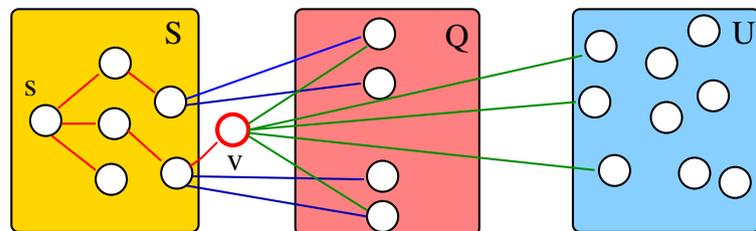
Nas implementações que examinaremos, o custo do vértice w em relação à árvore é $cst[w]$.

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 $parnt[w]$.

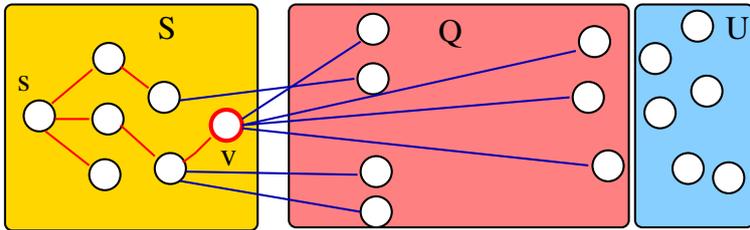
Navigation icons

Iteração



Navigation icons

Iteração



◀ ▶ ↺ ↻ 🔍

```

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
15          && cst[w] > G->adj[v][w]) {
16          cst[w] = G->adj[v][w];
17          fr[w] = v;
18      }
19  }
20 }
```

◀ ▶ ↺ ↻ 🔍

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;
```

◀ ▶ ↺ ↻ 🔍

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

◀ ▶ ↺ ↻ 🔍

Consumo de tempo

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

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

◀ ▶ ↺ ↻ 🔍

```

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;
19      }
20 }
```

◀ ▶ ↺ ↻ 🔍

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**.

```
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         }
20     }
21 }
```

Consumo de tempo

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

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 }
6 PQinit(G->V);
7 cst[0] = 0;
8 fr[0] = 0;
9 PQinsert(0);
```

```
19 else if (cst[w] > p->cst){
20     cst[w] = p->cst;
21     fr[w] = v;
22     PQdec(w);
23 }
24 } /* if (parnt[w] ...*/
25 } /* for (p...*/
26 } /* while ...
27 }
```

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 }
6 PQinit(G->V);
7 cst[s] = 0;
8 parnt[s] = s;
9 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();
    }

```