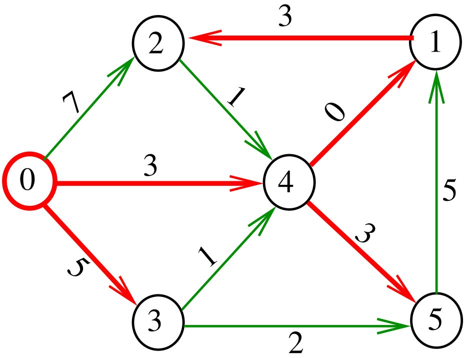
Melhores momentos

AULA 14

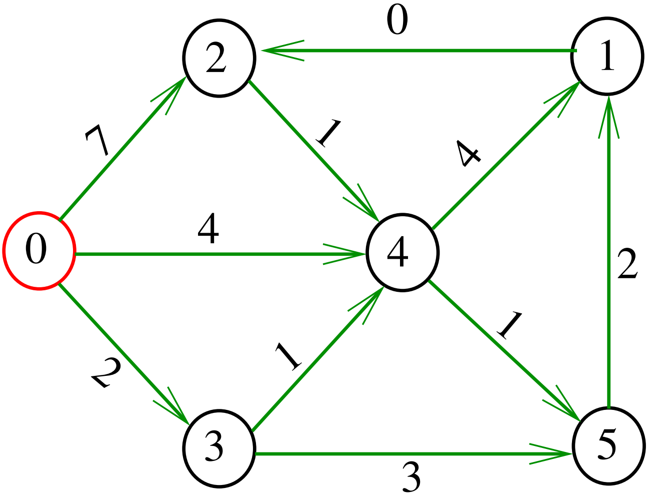
Problema

O **algoritmo de Dijkstra** resolve o problema da SPT:

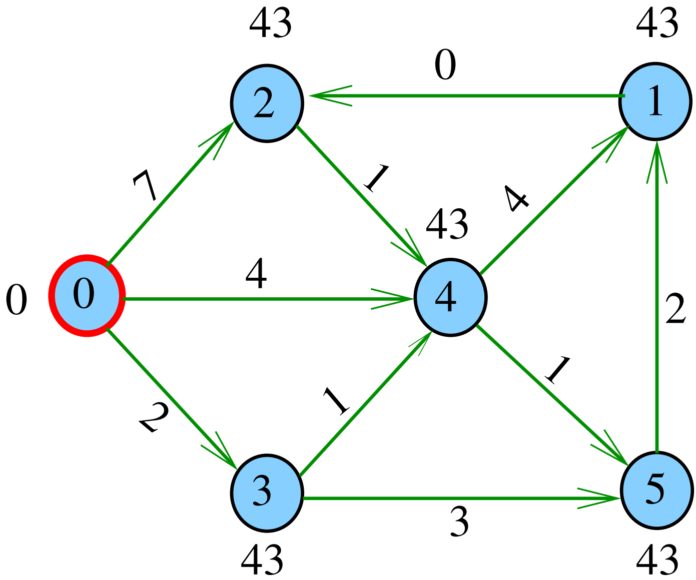
> *Dado um vértice s de um digrafo com custos não-negativos nos arcos, encontrar uma SPT com raiz s*



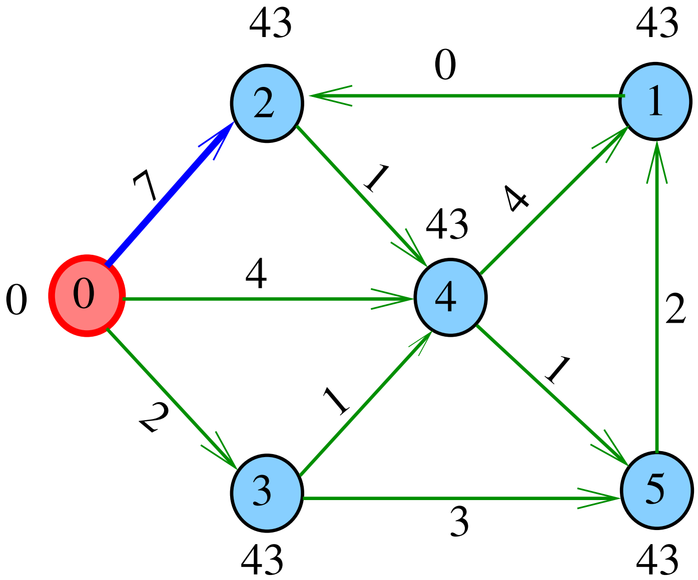
Simulação



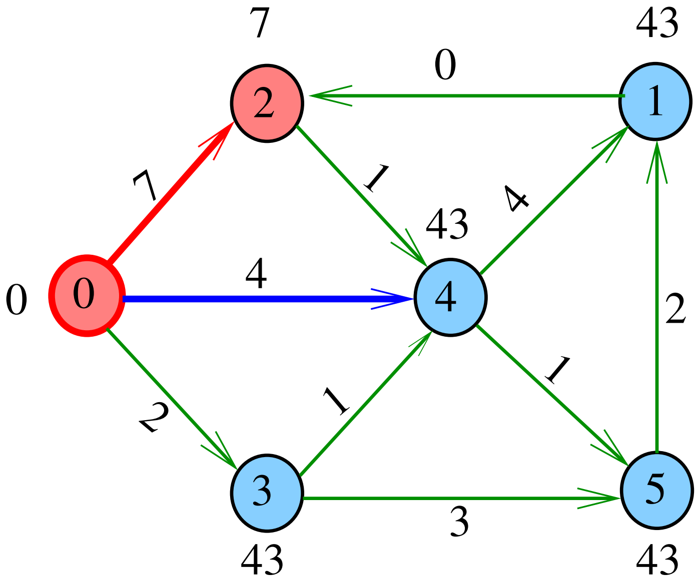
Simulação



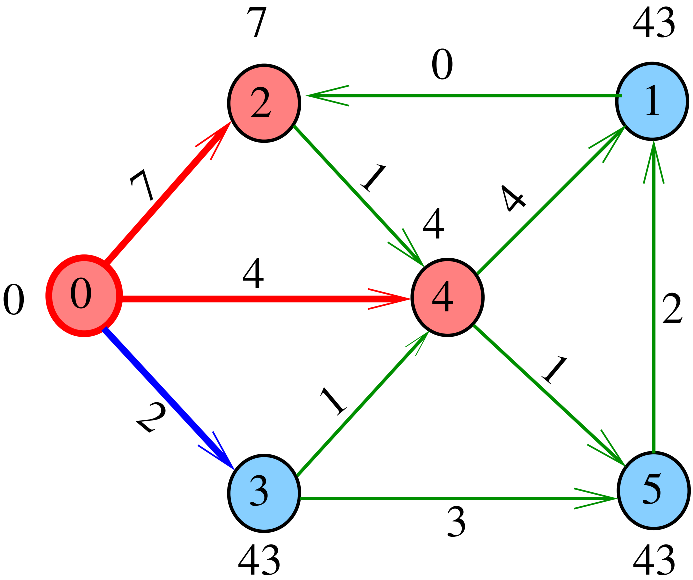
Simulação



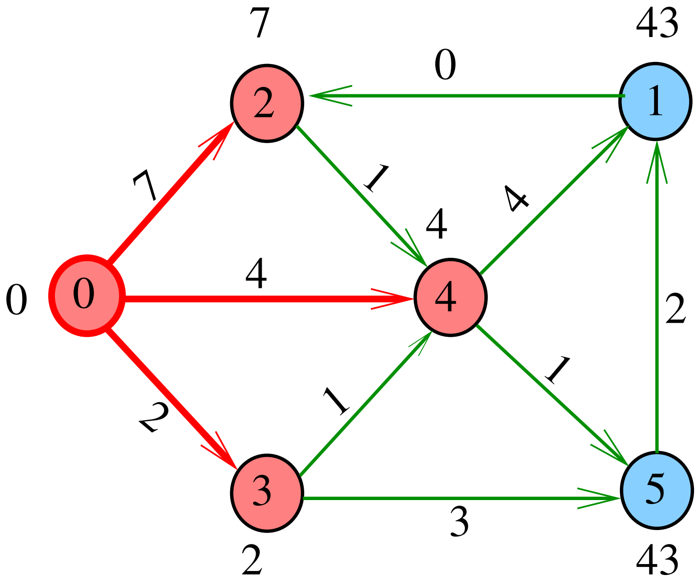
Simulação



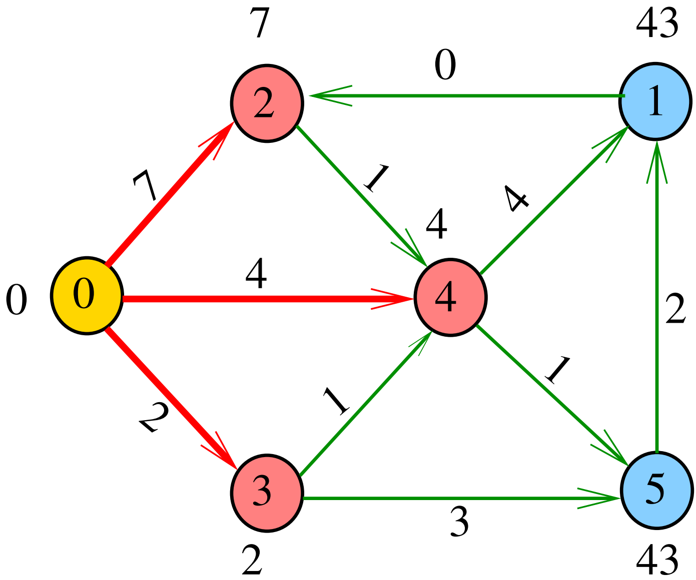
Simulação



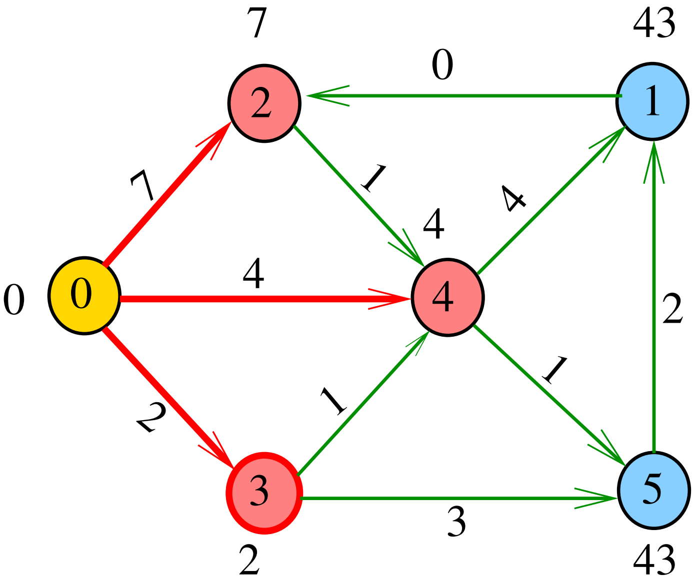
Simulação



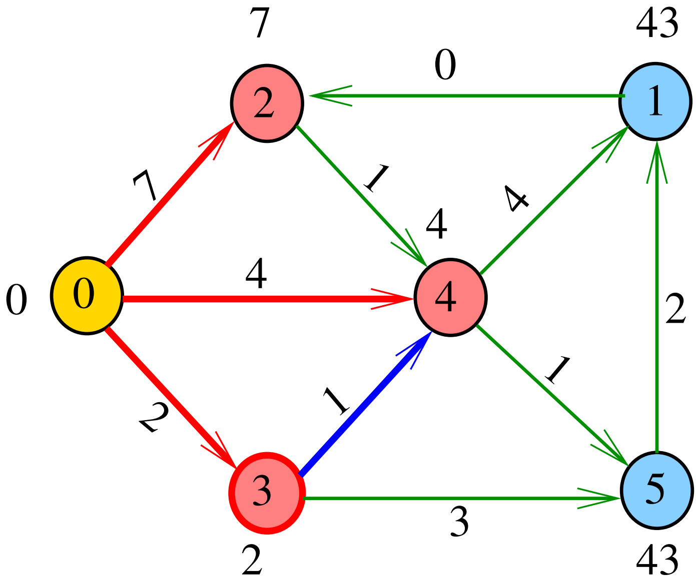
Simulação



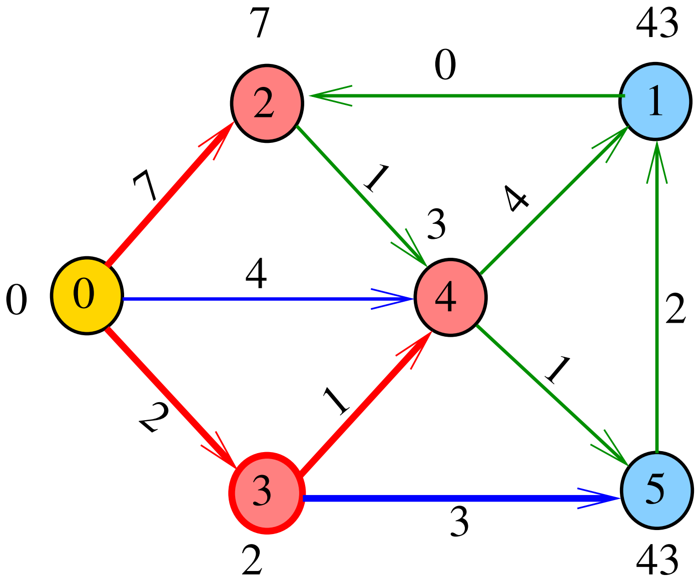
Simulação



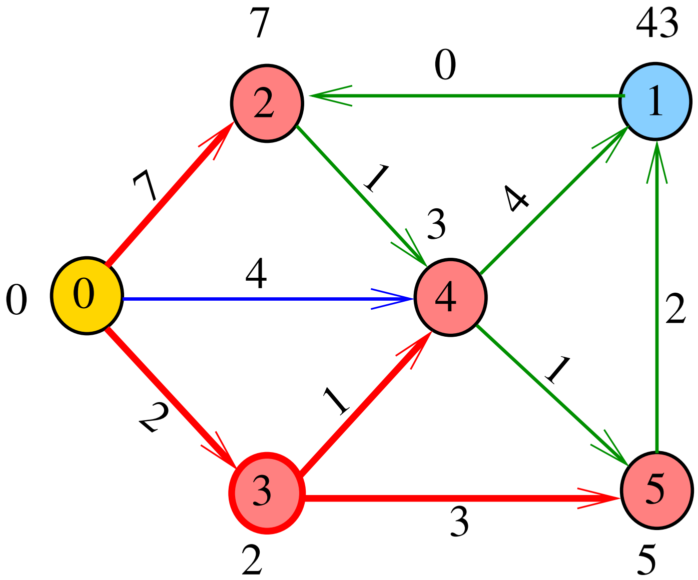
Simulação



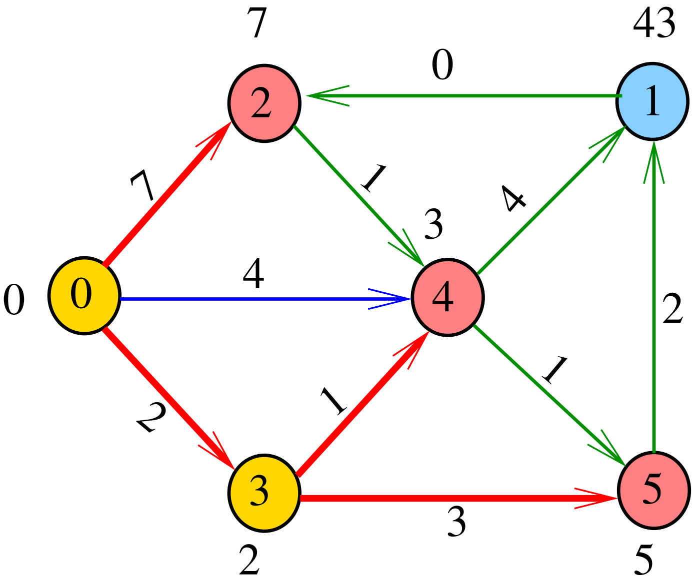
Simulação



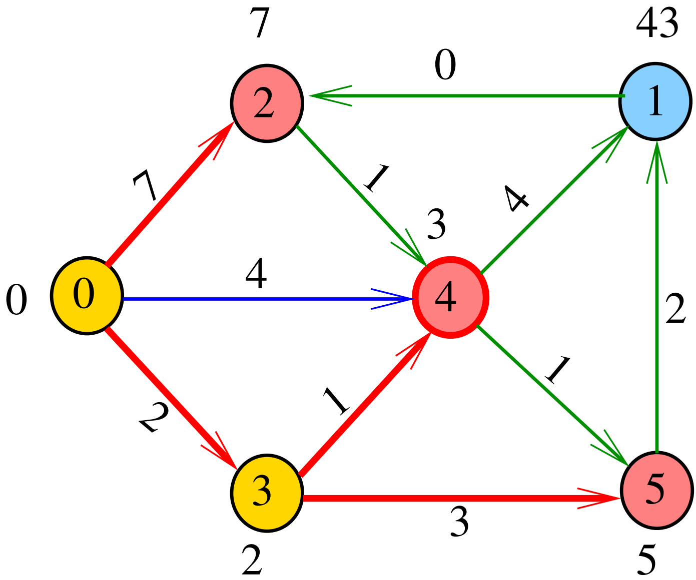
Simulação



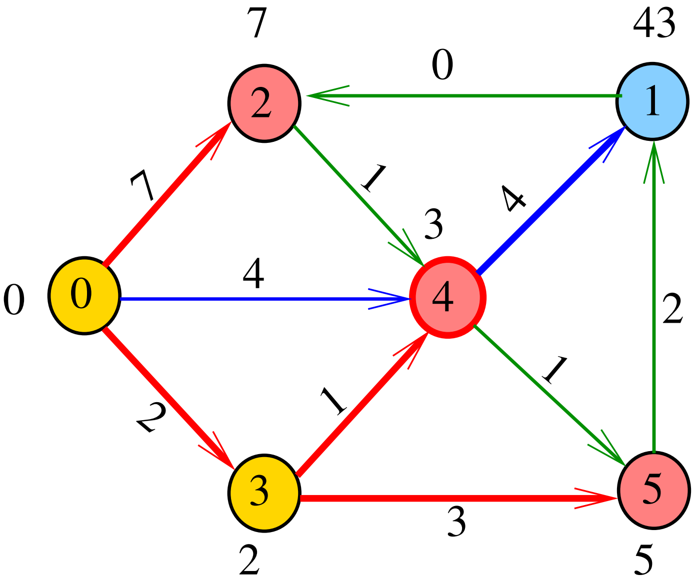
Simulação



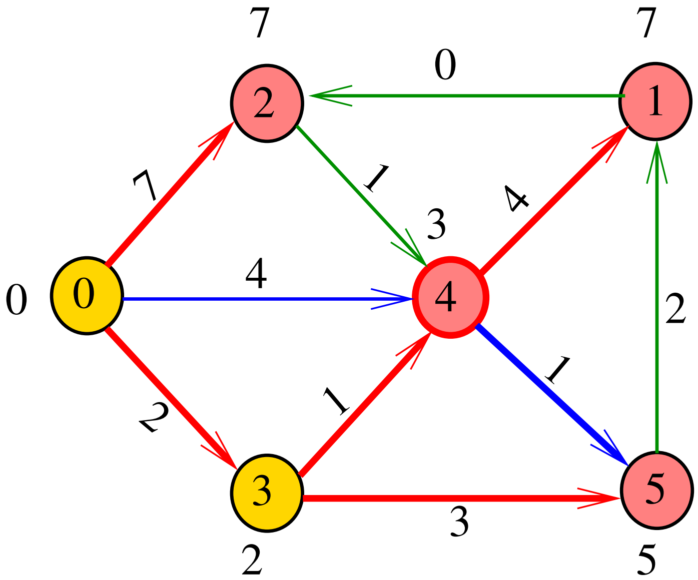
Simulação



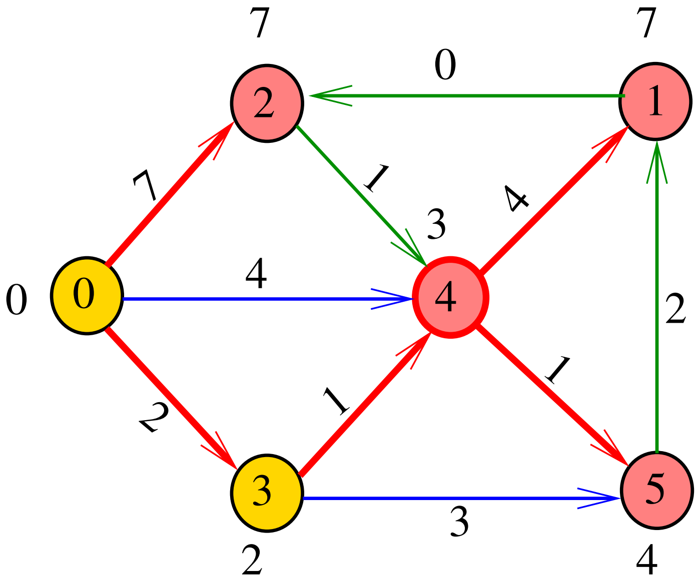
Simulação



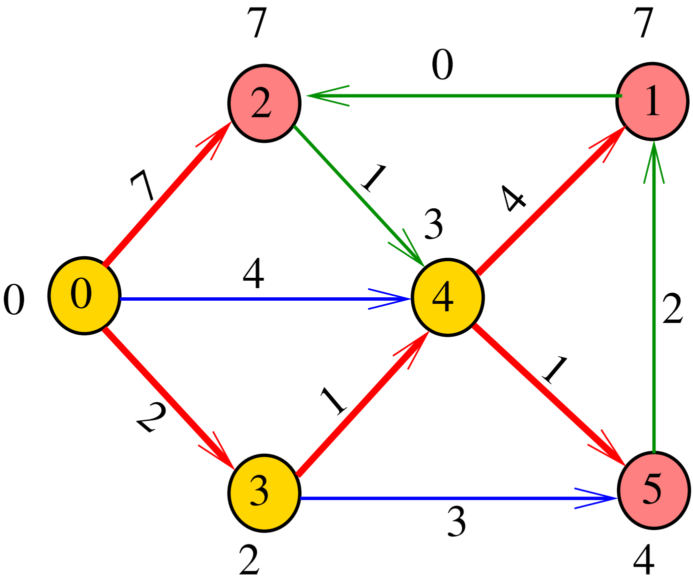
Simulação



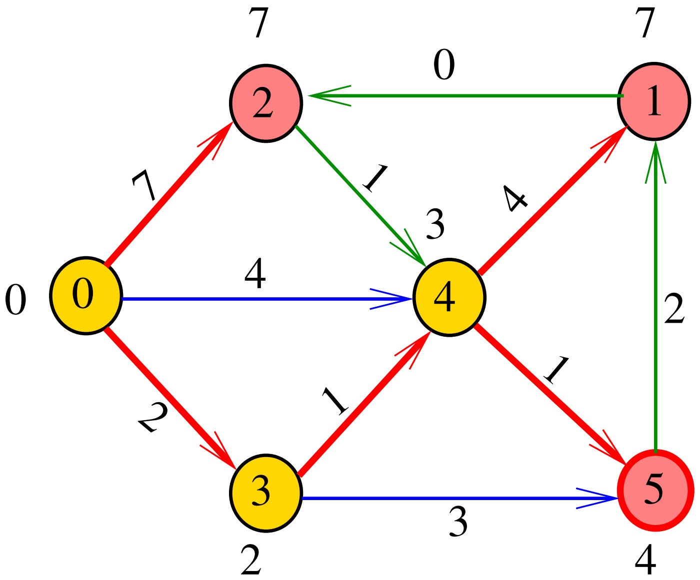
Simulação



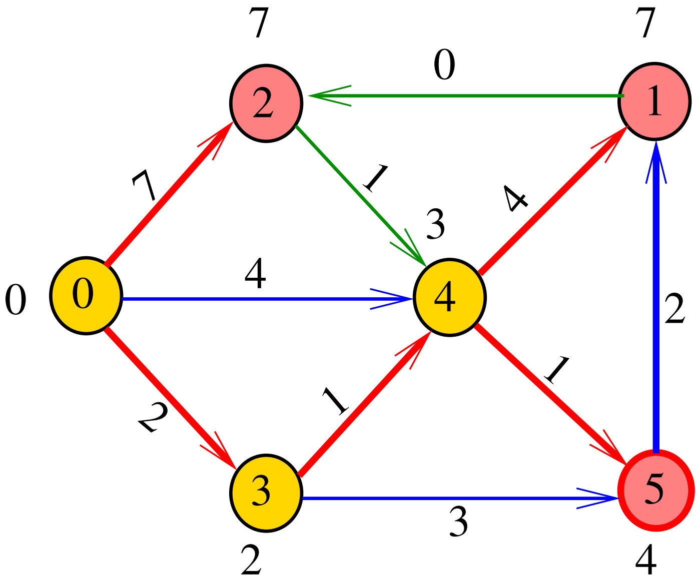
Simulação



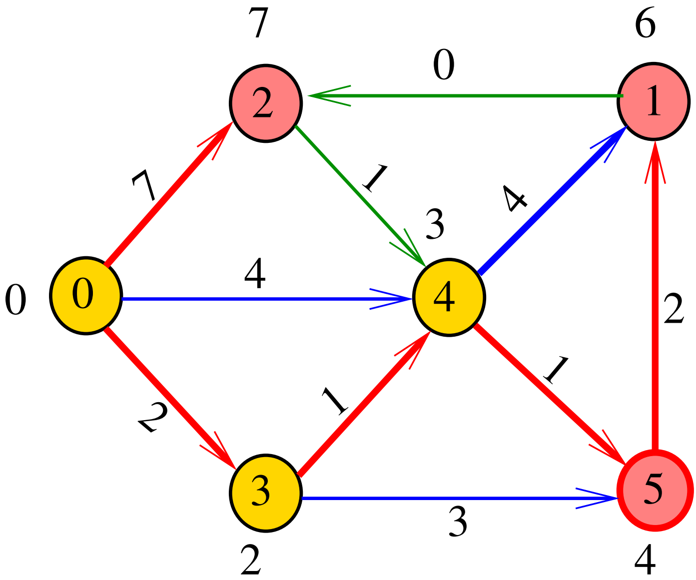
Simulação



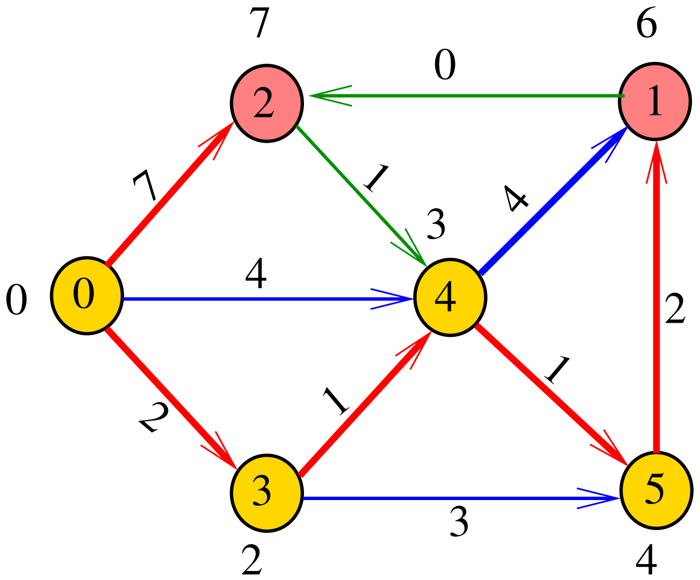
Simulação



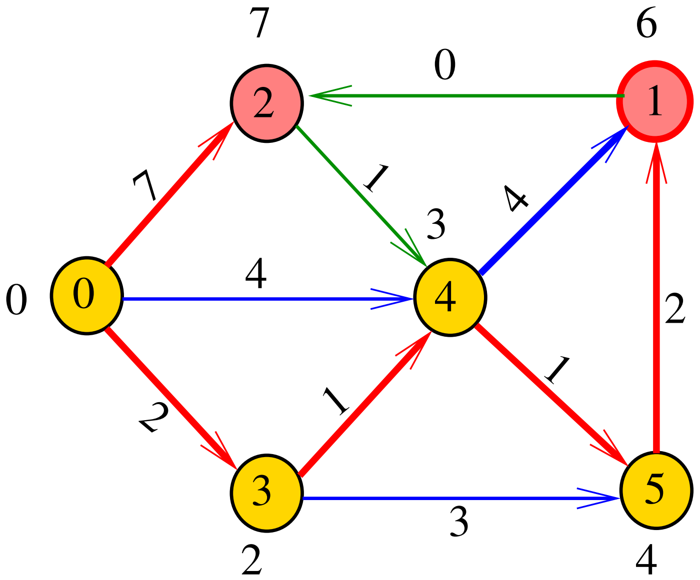
Simulação



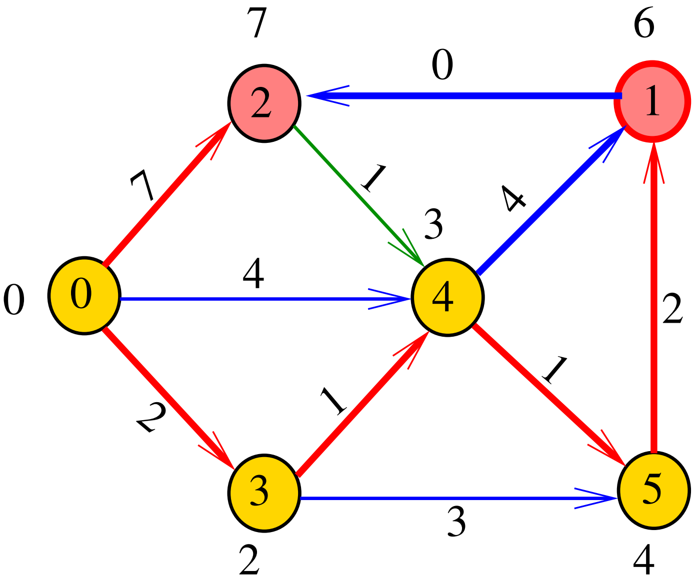
Simulação



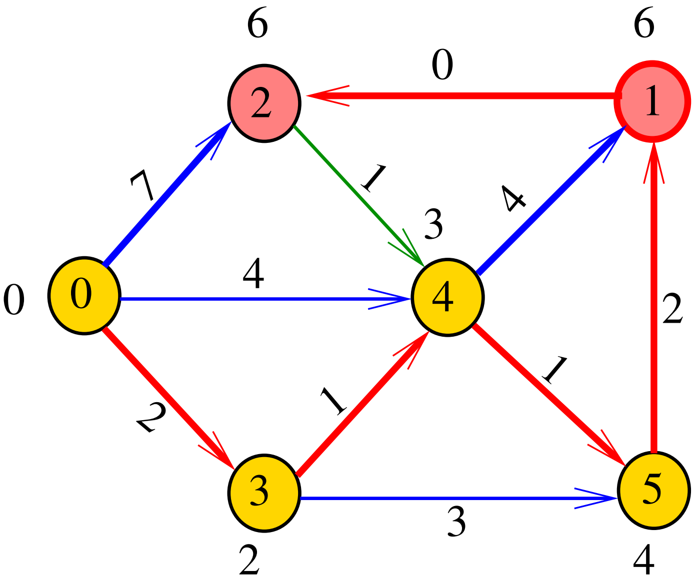
Simulação



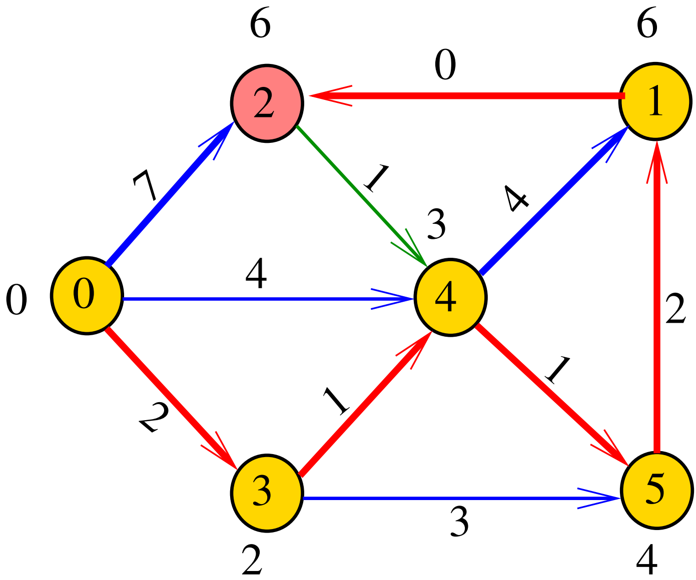
Simulação



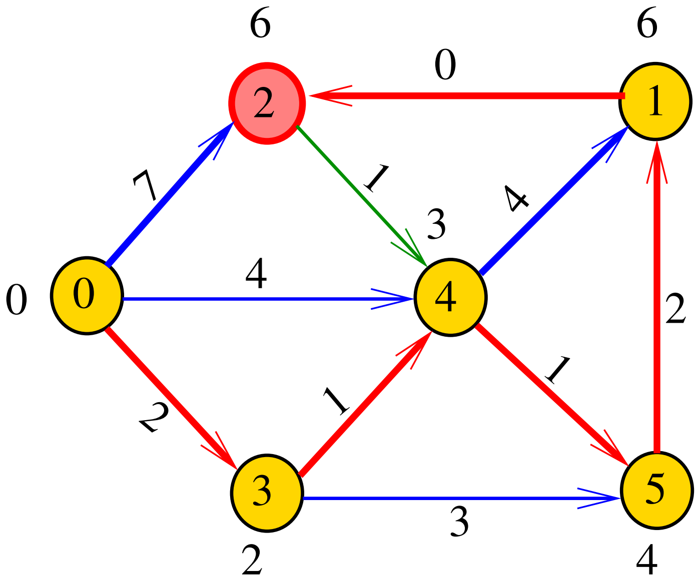
Simulação



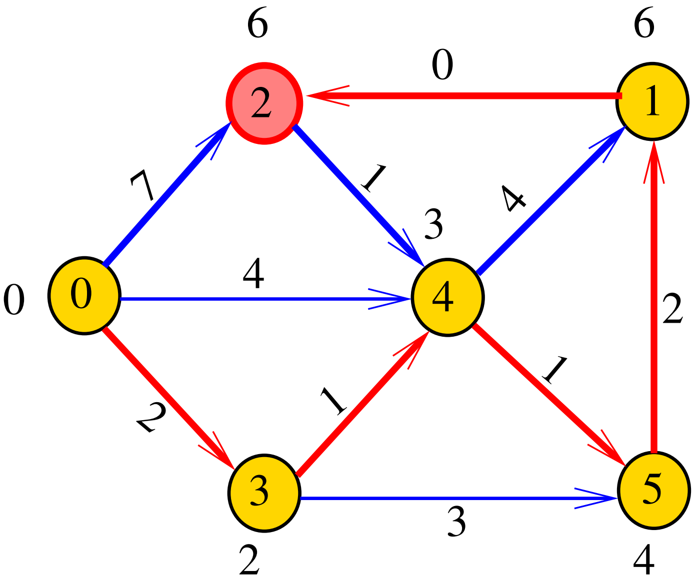
Simulação



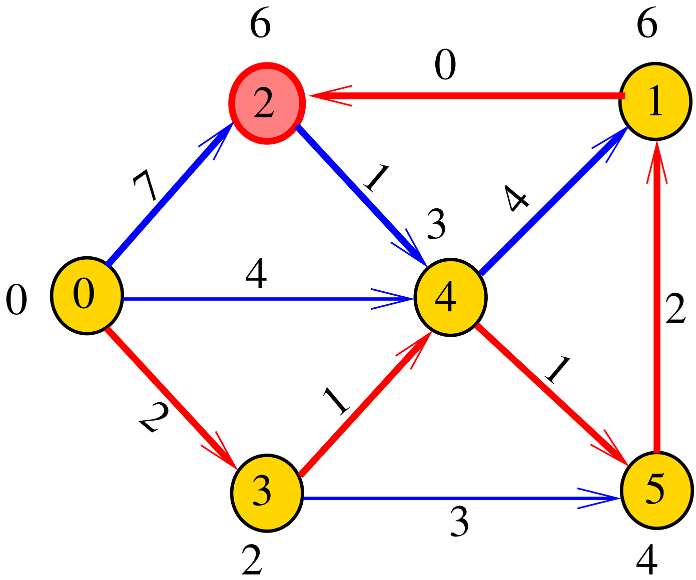
Simulação



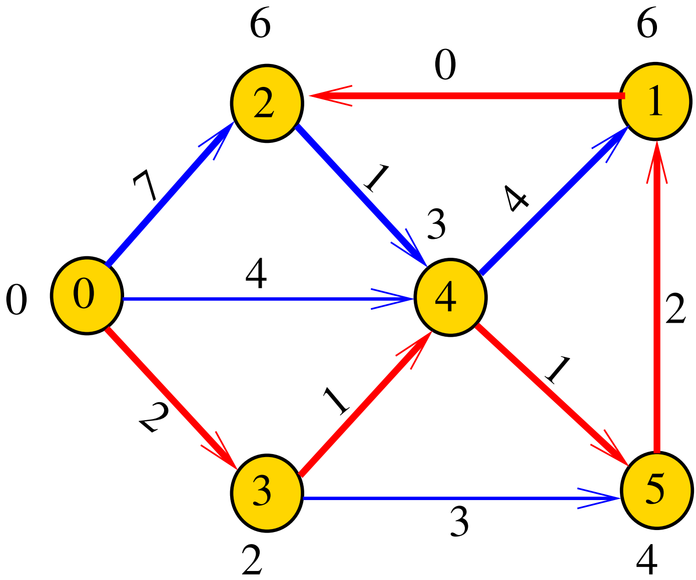
Simulação



Simulação



Simulação



`dijkstra`

Recebe digrafo `G` com custos não-negativos nos arcos e um vértice `s`

Calcula uma arborescência de caminhos mínimos com raiz `s`.

A arborescência é armazenada no vetor `parnt`

As distâncias em relação a `s` são armazenadas no vetor `cst`

```
void
dijkstra(Digraph G, Vertex s,
        Vertex parnt[], double cst[]);
```

Fila com prioridades

A função `dijkstra` usa uma fila com prioridades
A fila é manipulada pelas seguintes funções:

- `PQinit()`: inicializa uma fila de vértices em que cada vértice v tem prioridade `cst[v]`
- `PQempty()`: devolve 1 se a fila estiver vazia e 0 em caso contrário
- `PQinsert(v)`: insere o vértice `v` na fila
- `PQdelmin()`: retira da fila um vértice de prioridade mínima.
- `PQdec(w)`: reorganiza a fila depois que o valor de `cst[w]` foi decrementado.

dijkstra

```
#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;
```

dijkstra

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

dijkstra

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

Conclusão

O consumo de tempo da função dijkstra é $O(V + A)$ mais o consumo de tempo de

1	execução de	PQinit e PQfree,
$\leq V$	execuções de	PQinsert,
$\leq V + 1$	execuções de	PQempty,
$\leq V$	execuções de	PQdelmin, e
$\leq A$	execuções de	PQdec.

Conclusão

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

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

AULA 15

Mais algoritmo de Dijkstra

S 21.1 e 21.2

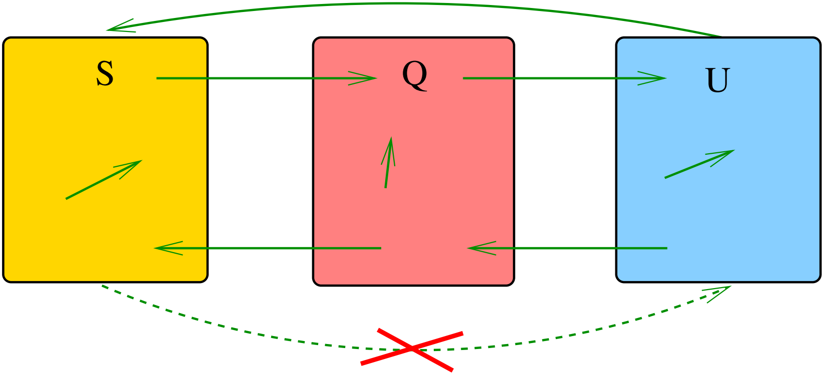
Relações invariantes

S = vértices examinados

Q = vértices visitados = vértices na fila

U = vértices ainda não visitados

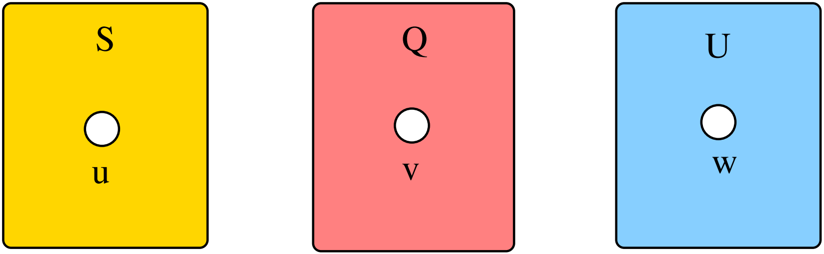
(i0) não existe arco v-w com v em S e w em U



Relações invariantes

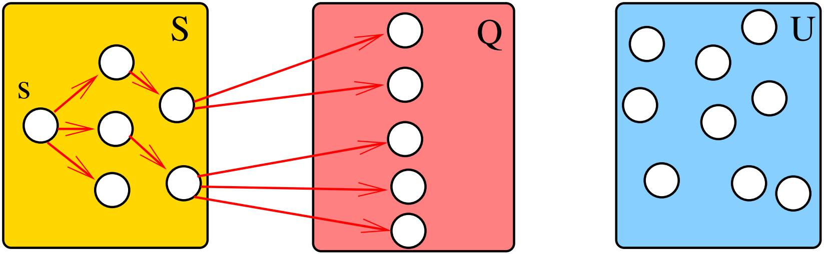
(i1) para cada `u` em `S`, `v` em `Q` e `w` em `U`

$$\texttt{cst[u]} \leq \texttt{cst[v]} \leq \texttt{cst[w]}$$



Relações invariantes

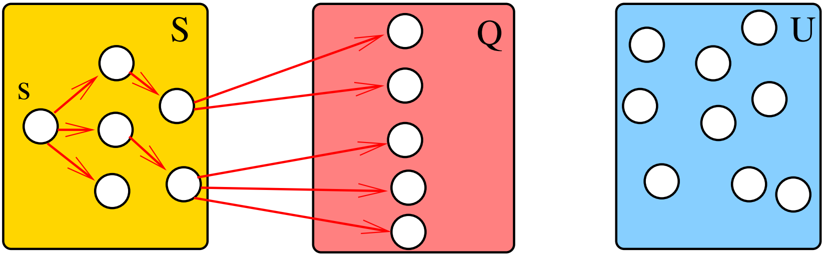
(i2) O vetor `parnt` restrito aos vértices de `S` e `Q` determina um árborescência com raiz `s`



Relações invariantes

(i3) Para arco `v-w` na arborescência vale que

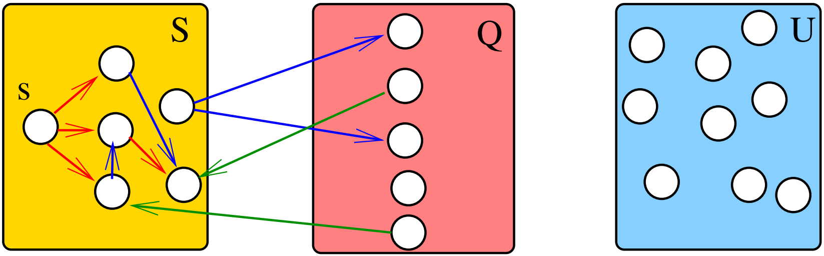
$$\texttt{cst}[\texttt{w}] = \texttt{cst}[\texttt{v}] + \text{custo do arco } \texttt{vw}$$



Relações invariantes

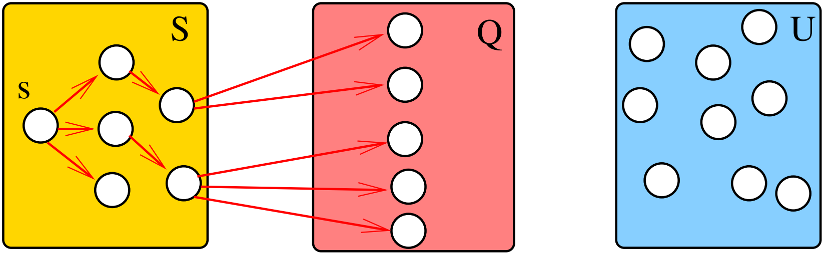
(i4) Para cada arco `v-w` com `v` **ou** `w` em `S` vale que

$$\texttt{cst}[\texttt{w}] - \texttt{cst}[\texttt{v}] \leq \text{custo do arco } \texttt{vw}$$

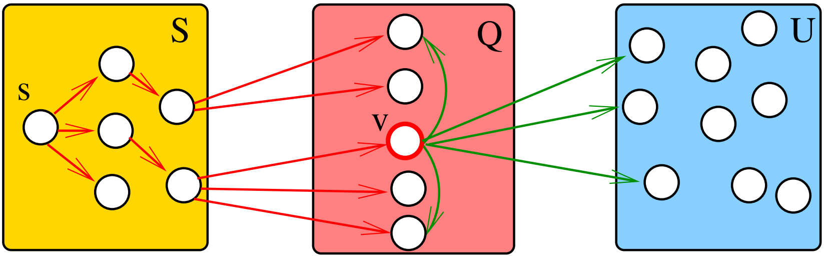


Relações invariantes

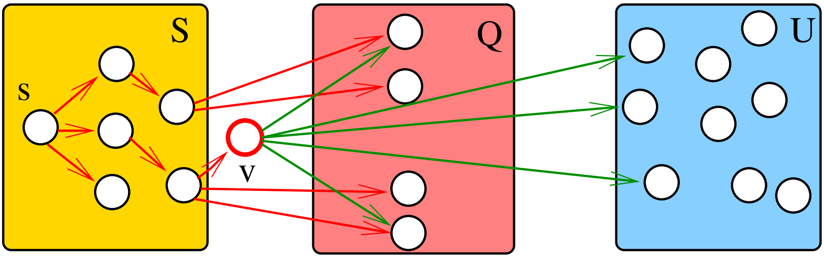
(i5) Para cada vértice `v` em `S` vale que `cst[v]` é o custo de um caminho mínimo de `s` a `v`.



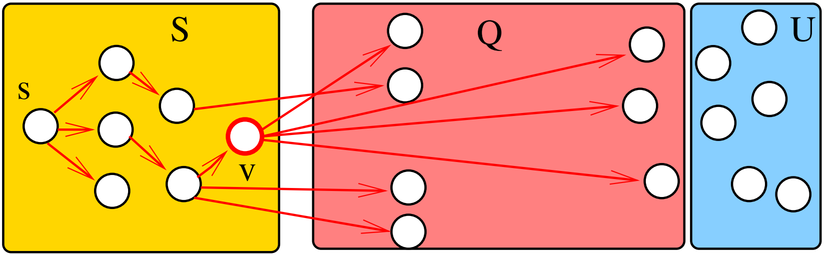
Iteração



Iteração



Iteração



Outra implementação para digrafos densos

```
#define INFINITO maxCST

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

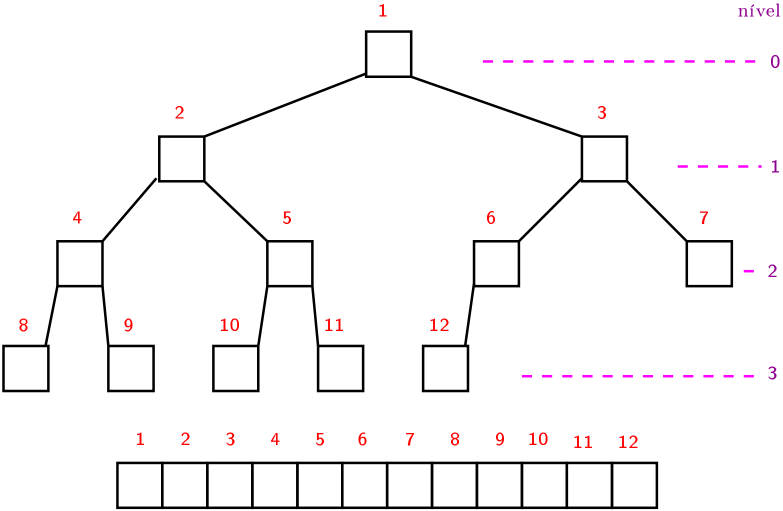
```
while (1) {
  double mincst = INFINITO;
  for (w = 0; w < G->V; w++)
      if (parnt[w]==-1 && mincst>cst[w])
          mincst = cst[w0=w];
  if (mincst == INFINITO) break;
  parnt[w0] = fr[w0];
  for (w = 0; w < G->V; w++)
      if(cst[w]>cst[w0]+G->adj[w0][w]) {

          cst[w] = cst[w0]+G->adj[w0][w];
          fr[w] = w0;
      }
  }
}
```

Dijkstra para digrafos esparços

S 21.1 e 21.2

Representação de árvores em vetores



Pais e filhos

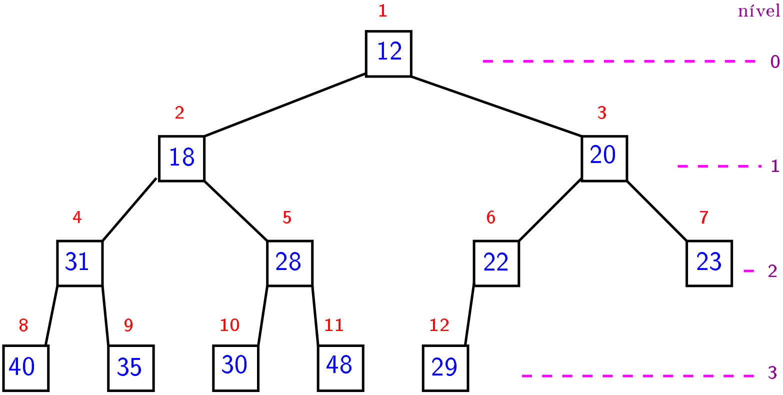
$A[1\,..\,m]$ é um vetor representando uma árvore.
Diremos que para qualquer índice ou **nó** i,

- $\lfloor i/2 \rfloor$ é o **pai** de i;
- $2\,i$ é o **filho esquerdo** de i;
- $2\,i+1$ é o **filho direito**.

Todo nó i é raiz da subárvore formada por

$$A[i, 2i, 2i+1, 4i, 4i+1, 4i+2, 4i+3, 8i, \ldots, 8i+7, \ldots]$$

Min-heap

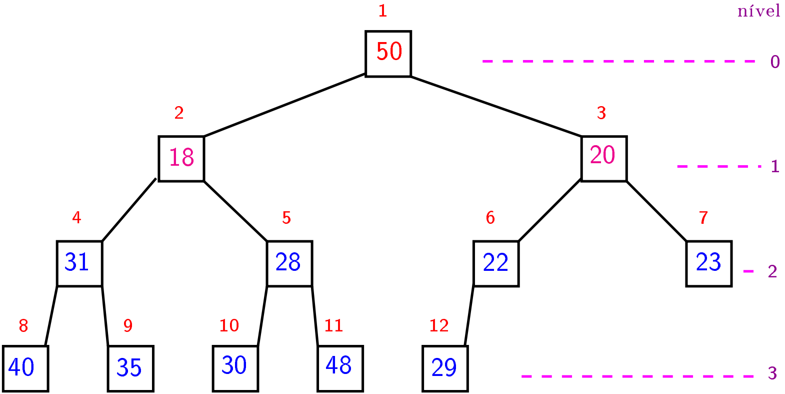


1	2	3	4	5	6	7	8	9	10	11	12
12	18	20	31	28	22	23	40	35	30	48	29

Rotina básica de manipulação de max-heap

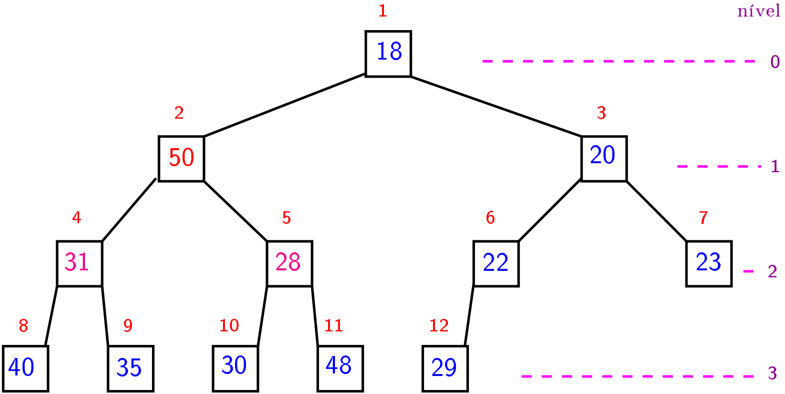
Recebe $A[1\,..\,m]$ e $i \geq 1$ tais que subárvores com raiz $2i$ e $2i+1$ são min-heaps e rearranja A de modo que subárvore com raiz i seja min-heap.

Rotina básica de manipulação de min-heap



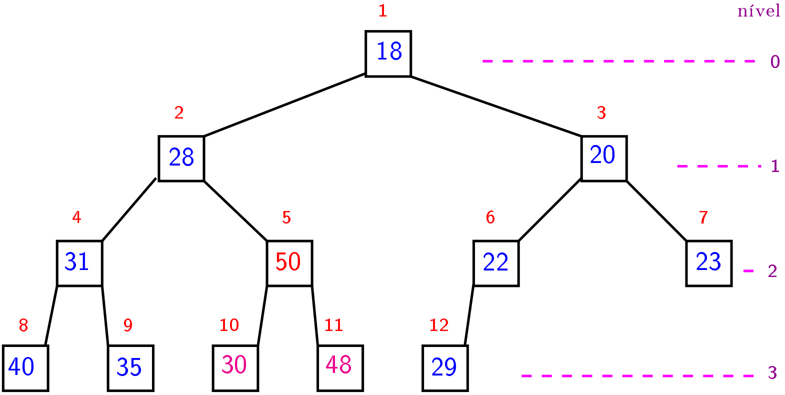
1	2	3	4	5	6	7	8	9	10	11	12
50	18	20	31	28	22	23	40	35	30	48	29

Rotina básica de manipulação de min-heap



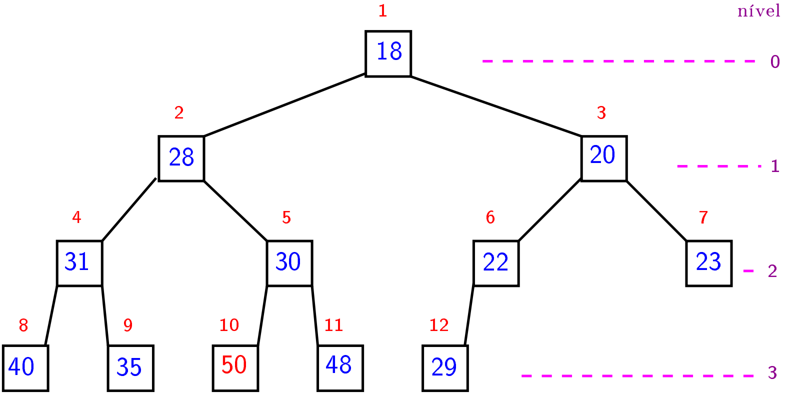
1	2	3	4	5	6	7	8	9	10	11	12
18	50	20	31	28	22	23	40	35	30	48	29

Rotina básica de manipulação de min-heap



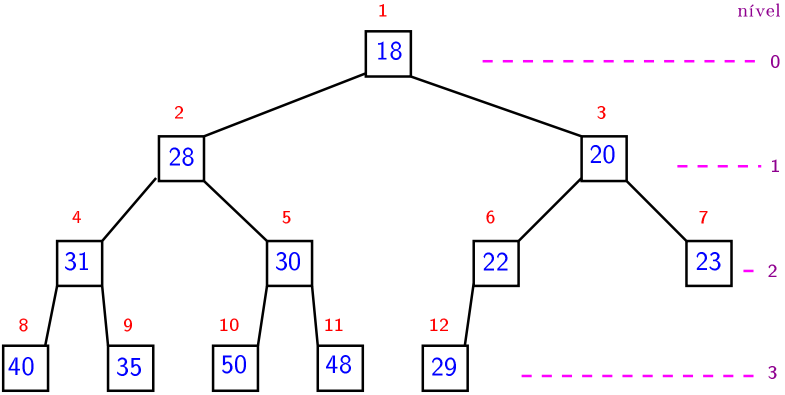
1	2	3	4	5	6	7	8	9	10	11	12
18	28	20	31	50	22	23	40	35	30	48	29

Rotina básica de manipulação de min-heap



1	2	3	4	5	6	7	8	9	10	11	12
18	28	20	31	30	22	23	40	35	50	48	29

Rotina básica de manipulação de min-heap



1	2	3	4	5	6	7	8	9	10	11	12
18	28	20	31	30	22	23	40	35	50	48	29

Implementação clássica Min-Heap

O vetor qp é o "inverso" de pq:

para cada vértice v, qp[v] é o único índice tal que pq[qp[v]] == v.

É claro que qp[pq[i]]==i para todo i.

```
static Vertex pq[maxV+1];
static int N;
static int qp[maxV];
```

PQinit, PQempty, PQinsert

```
void PQinit(void) {
   N = 0;
}
int PQempty(void) {
   return N == 0;
}
void PQinsert(Vertex v) {
   qp[v] = ++N;
   pq[N] = v;
   fixUp(N);
}
```

PQdelmin e PQdec

```
Vertex PQdelmin(void) {
   exch(1, N);
   --N;
   fixDown(1);
   return pq[N+1];
}
void PQdec(Vertex w) {
   fixUp(qp[w]);
}
```

exch e fixUp

```
static void exch(int i, int j) {
    Vertex t;
    t = pq[i]; pq[i] = pq[j]; pq[j] = t;
    qp[pq[i]] = i;
    qp[pq[j]] = j;
}
static void fixUp(int i) {
    while (i>1 && cst[pq[i/2]]>cst[pq[i]]){
        exch(i/2, i);
        i = i/2;
    }
}
```

fixDown

```
static void fixDown(int i) {
   int j;
   while (2*i <= N) {
      j = 2*i;
      if (j< N && cst[pq[j]] > cst[pq[j+1]])
         j++;
      if (cst[pq[i]] <= cst[pq[j]]) break;
      exch(i, j);
      i = j;
   }
}
```

Consumo de tempo Min-Heap

	heap	d-heap	fibonacci heap
PQinsert	$\mathrm{O}(\lg \mathtt{V})$	$\mathrm{O}(\log_D \mathtt{V})$	$\mathrm{O}(1)$
PQdelmin	$\mathrm{O}(\lg \mathtt{V})$	$\mathrm{O}(\log_D \mathtt{V})$	$\mathrm{O}(\lg \mathtt{V})$
PQdec	$\mathrm{O}(\lg \mathtt{V})$	$\mathrm{O}(\log_D \mathtt{V})$	$\mathrm{O}(1)$
dijkstra	$\mathrm{O}(\mathtt{A} \lg \mathtt{V})$	$\mathrm{O}(\mathtt{A} \log_D \mathtt{V})$	$\mathrm{O}(\mathtt{A} + \mathtt{V} \lg \mathtt{V})$

Consumo de tempo Min-Heap

	bucket heap	radix heap
`PQinsert`	$O(1)$	$O(\lg(VC)R$
`PQdelmin`	$O(C)$	$O(\lg(VC))$
`PQdec`	$O(1)$	$O(A + V \lg(VC))$
`dijkstra`	$O(A + VC)$	$O(A + V \lg(VC))$

C = maior custo de um arco.

Conclusão

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

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