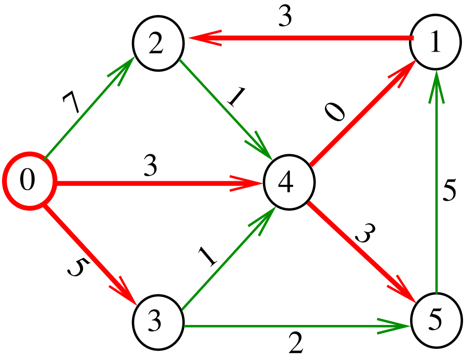
Melhores momentos

AULA 14

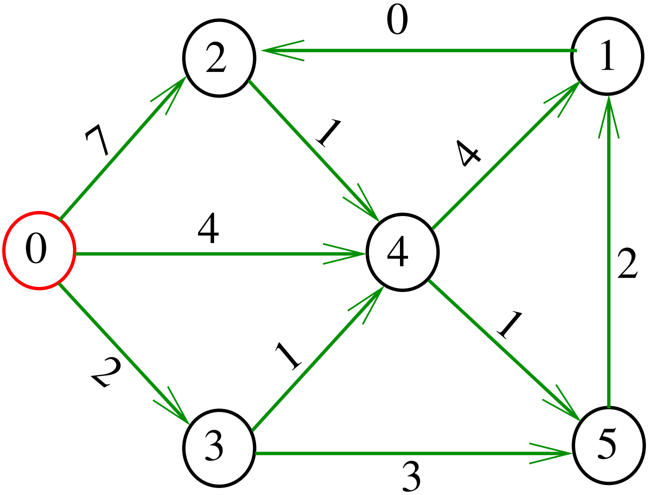
Problema

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

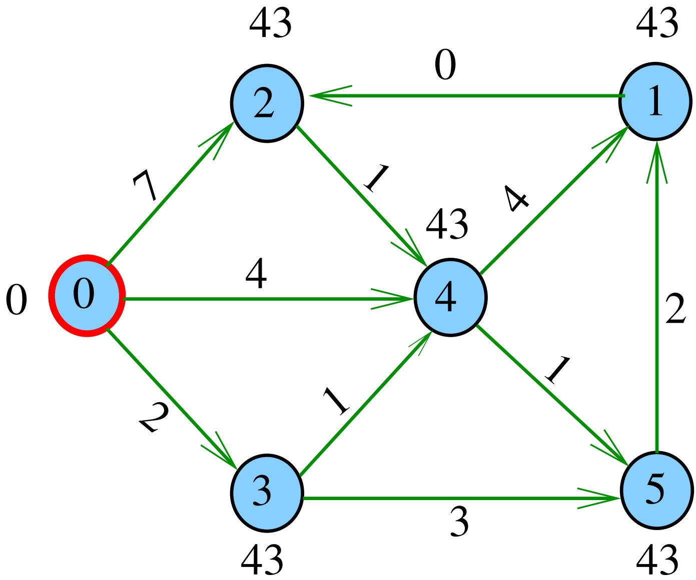
> *Dado um vértice s de um digrafo com custos não-negativos nos arcos, encontrar uma SPT com raiz s*



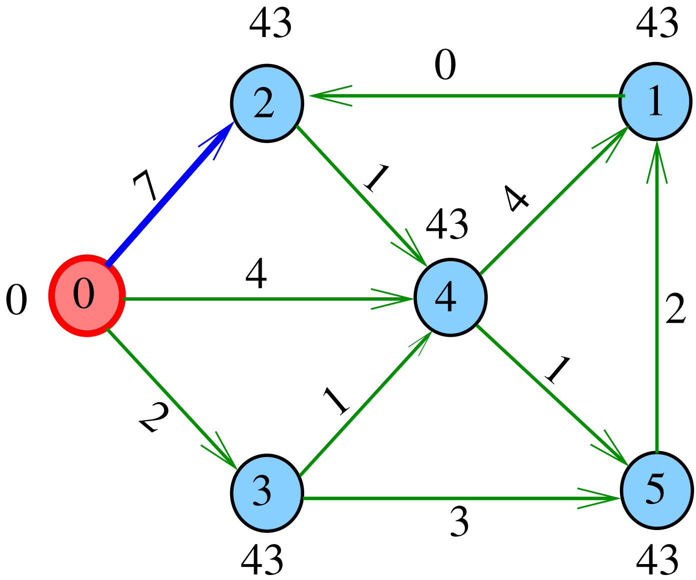
Simulação



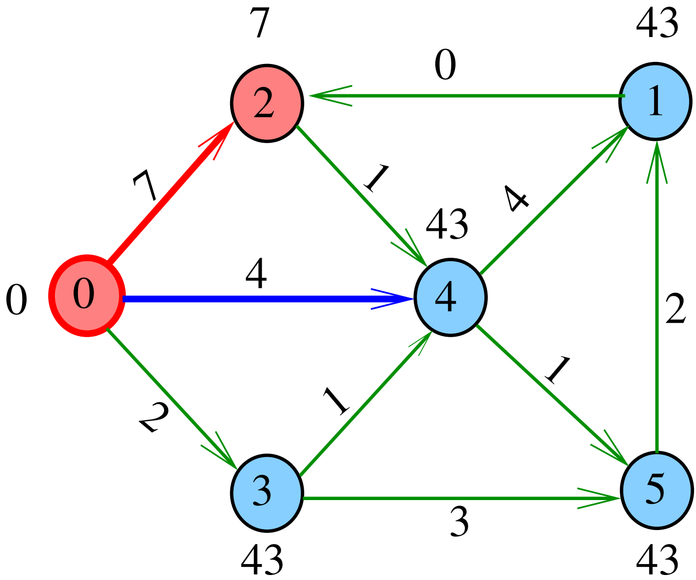
Simulação



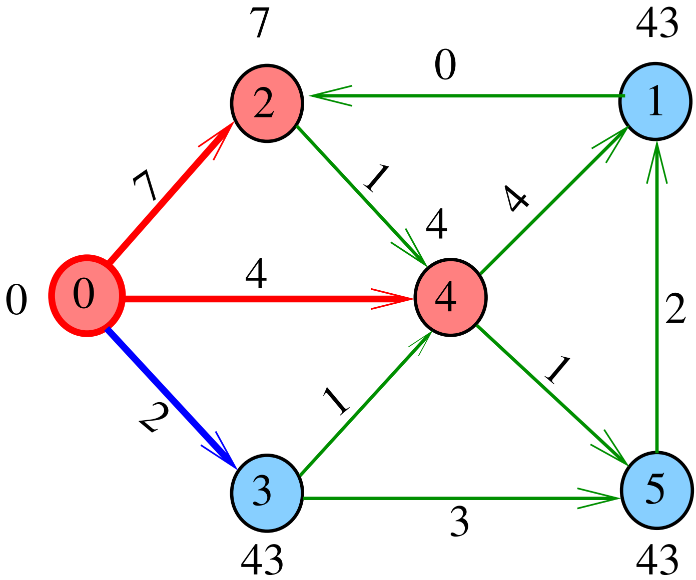
Simulação



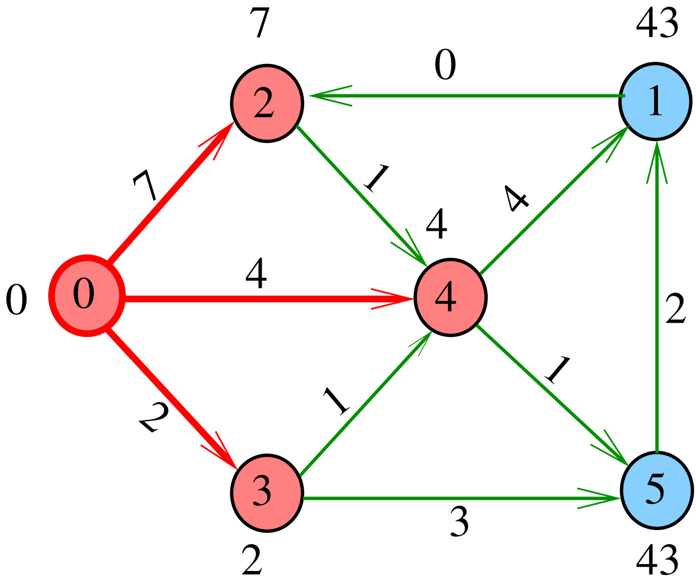
Simulação



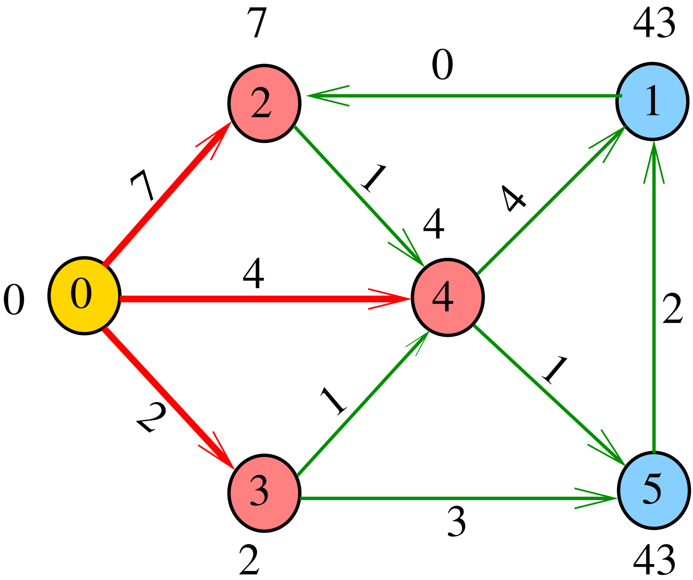
Simulação



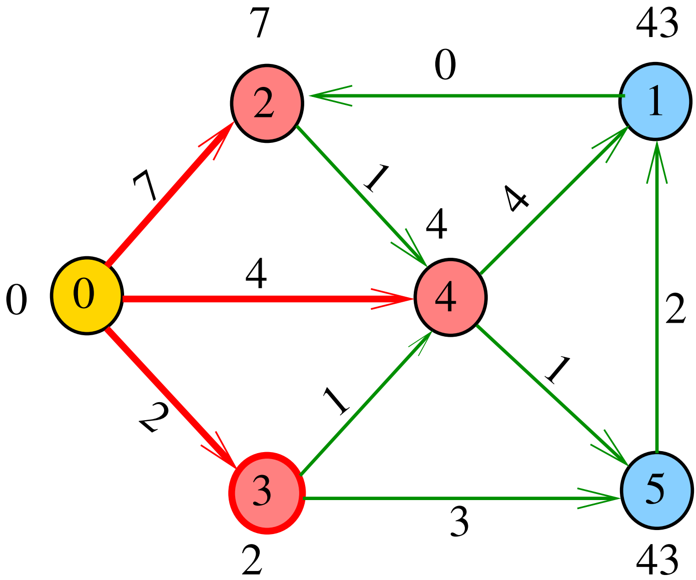
Simulação



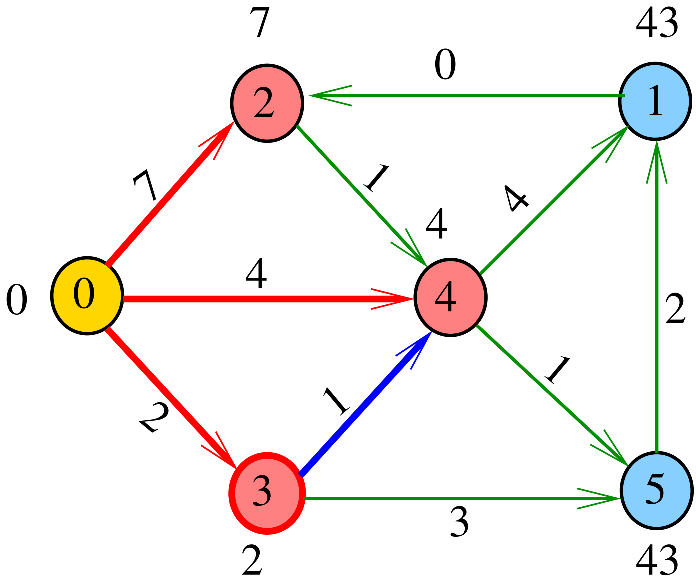
Simulação



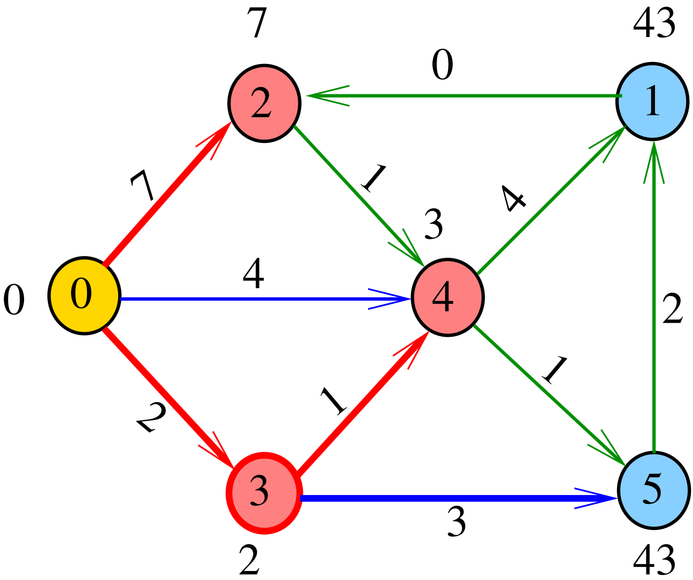
Simulação



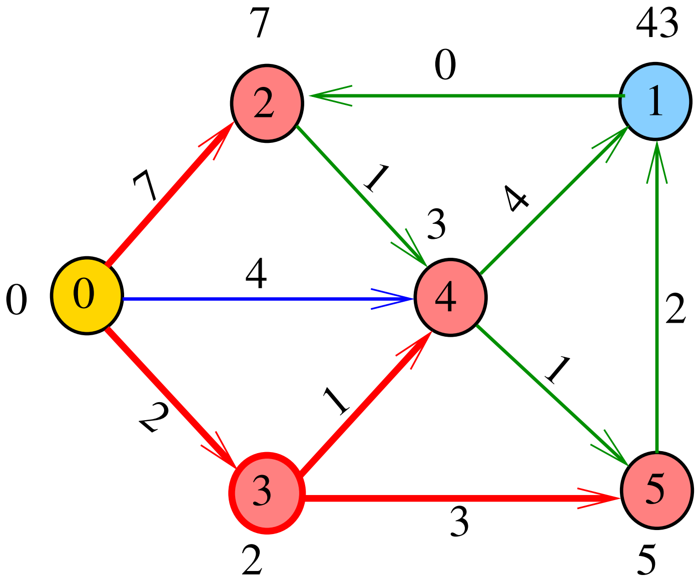
Simulação



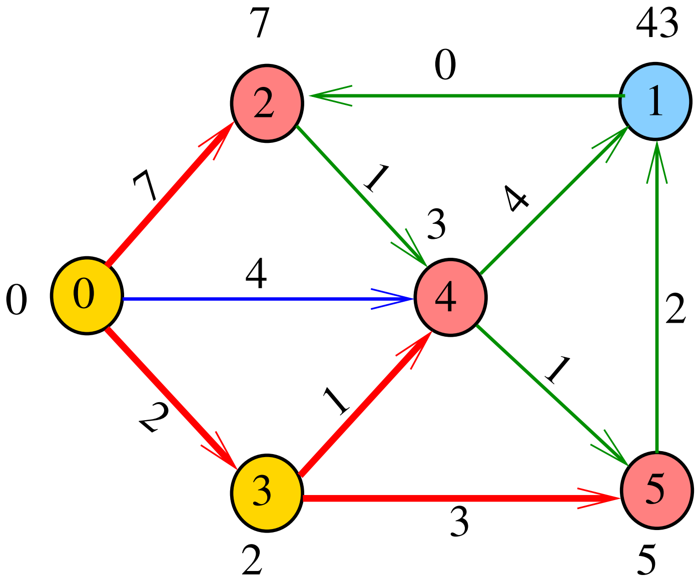
Simulação



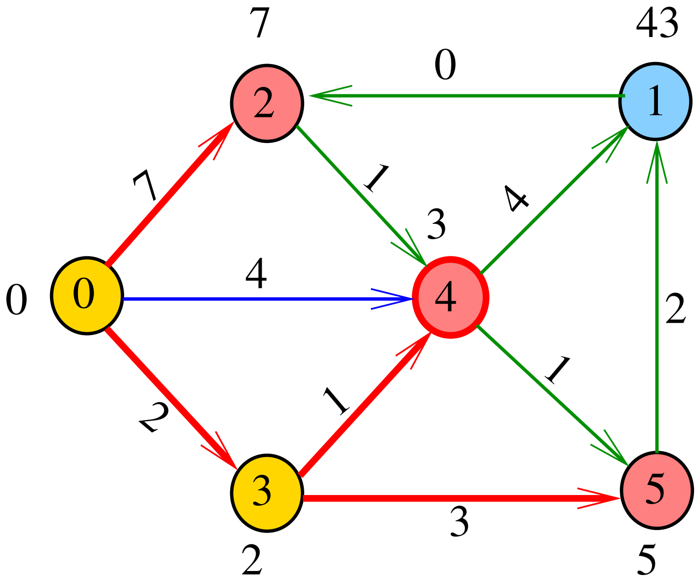
Simulação



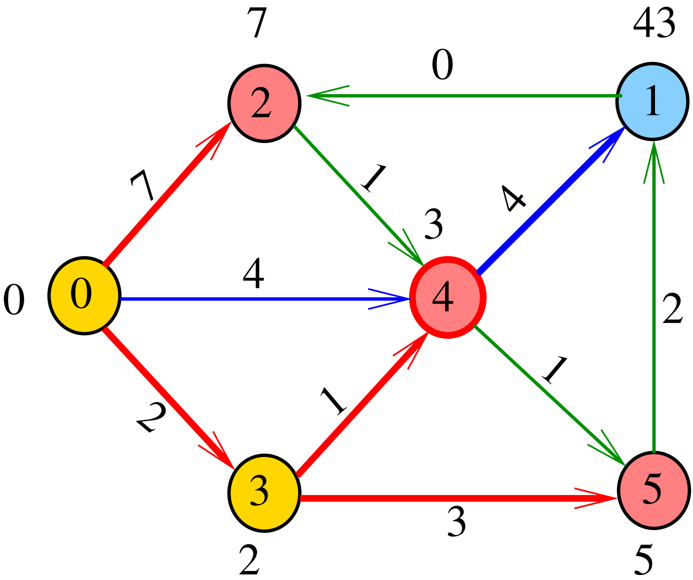
Simulação



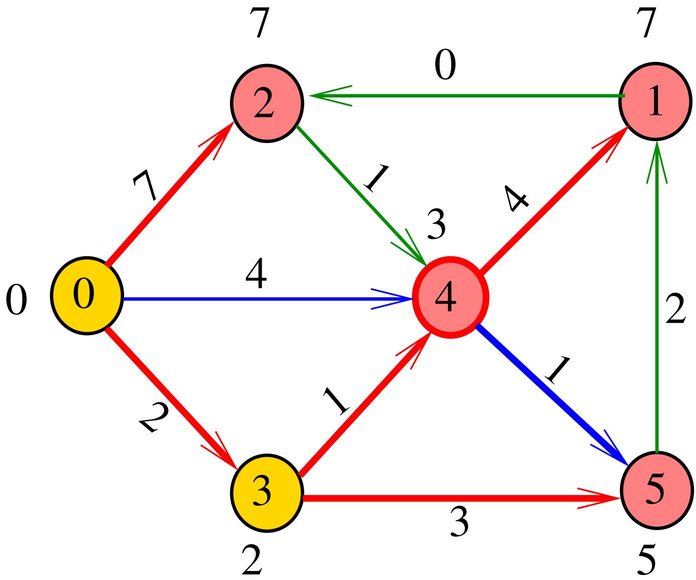
Simulação



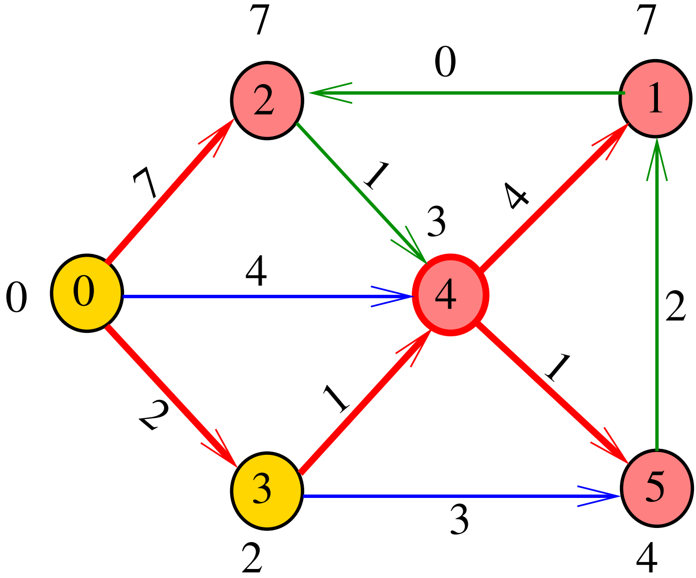
Simulação



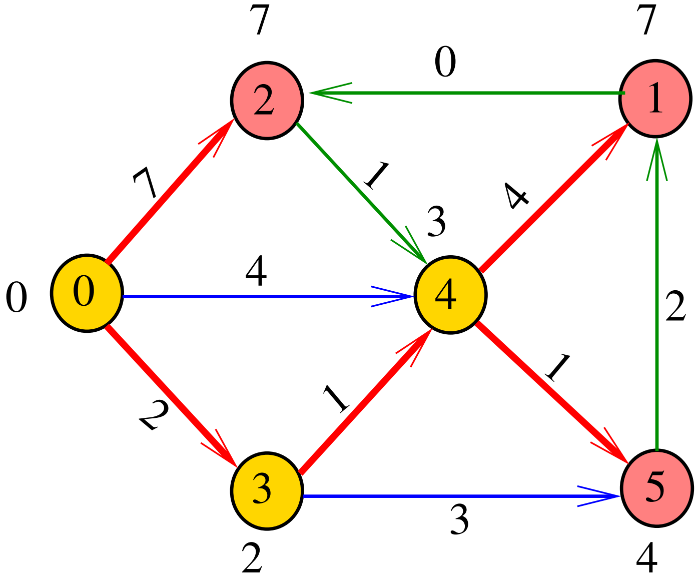
Simulação



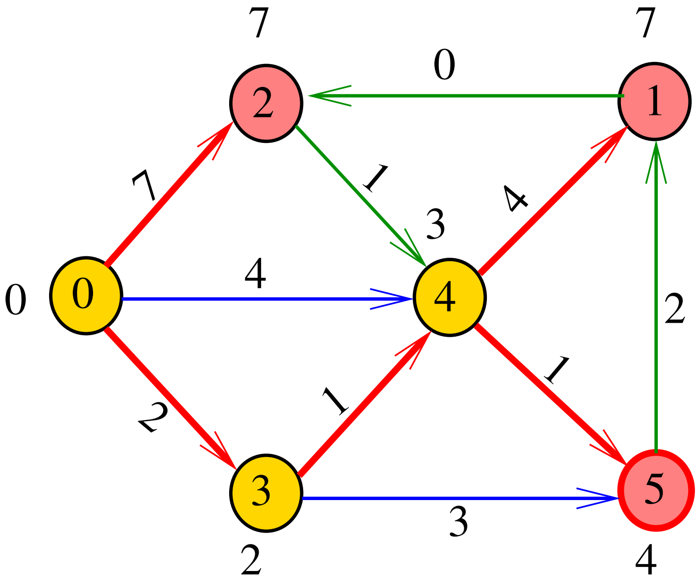
Simulação



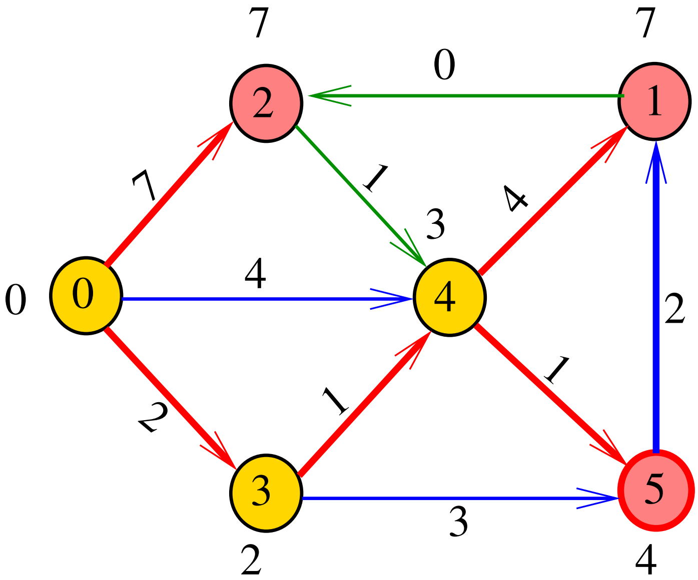
Simulação



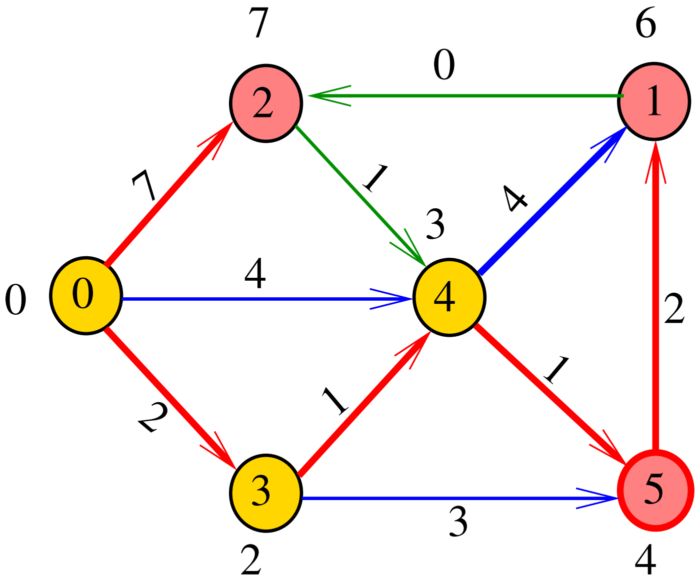
Simulação



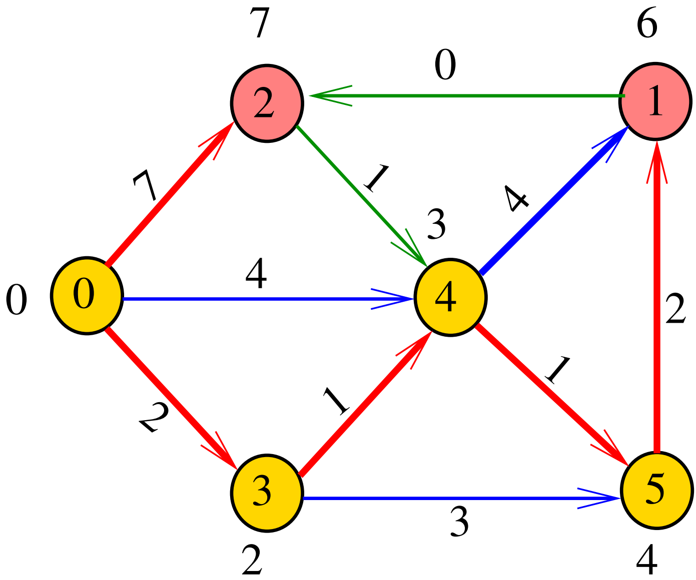
Simulação



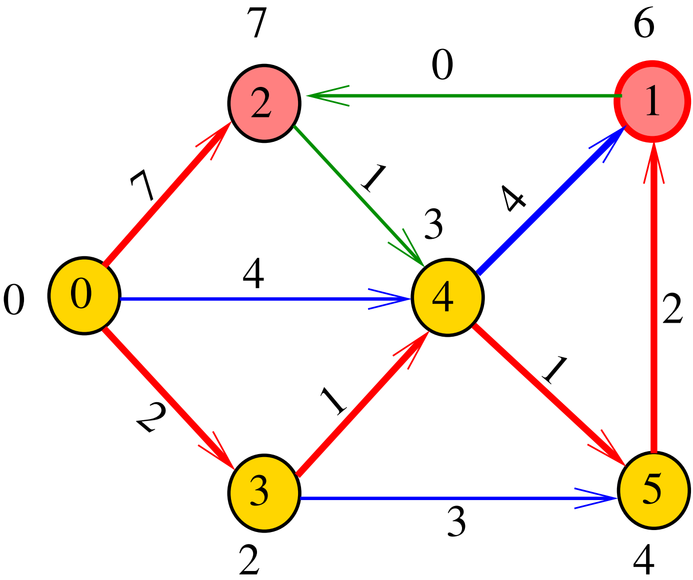
Simulação



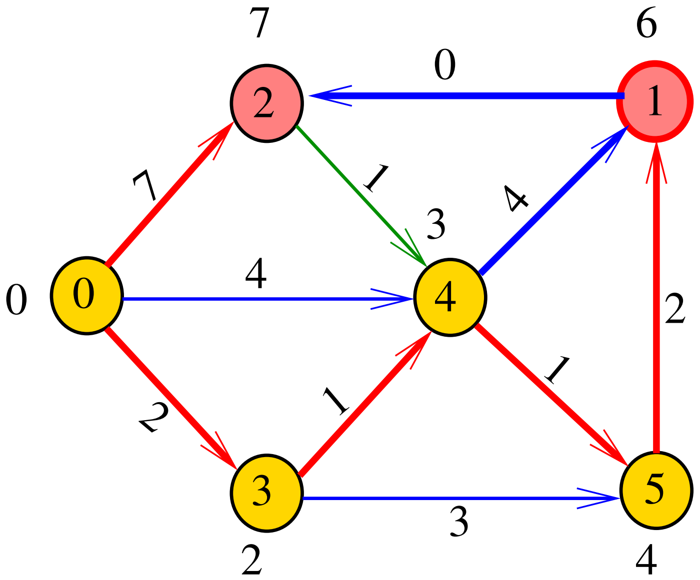
Simulação



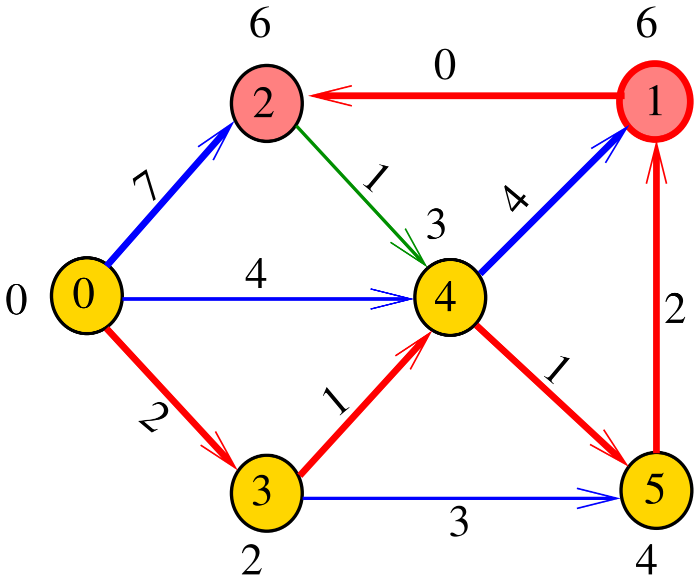
Simulação



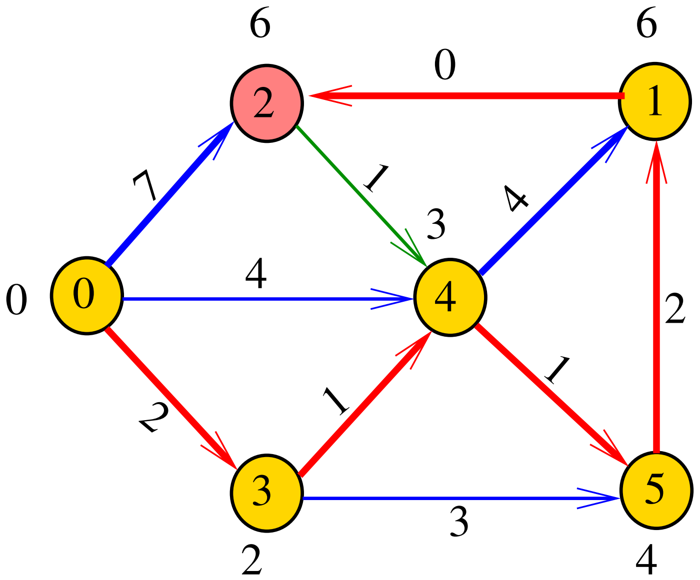
Simulação



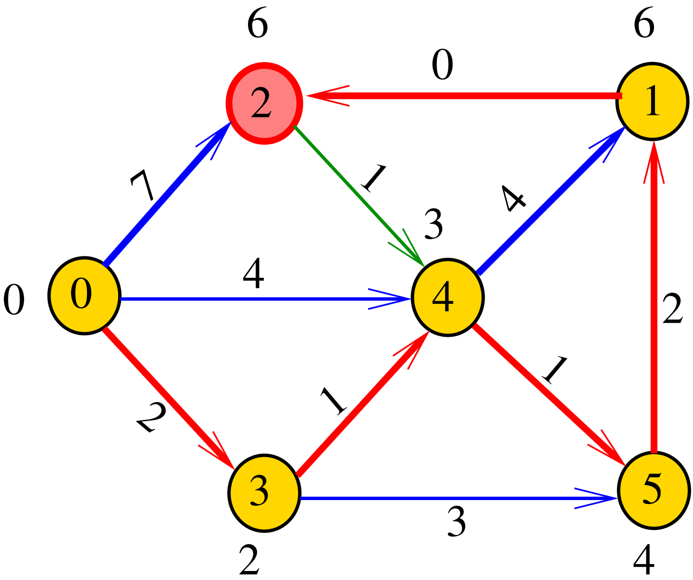
Simulação



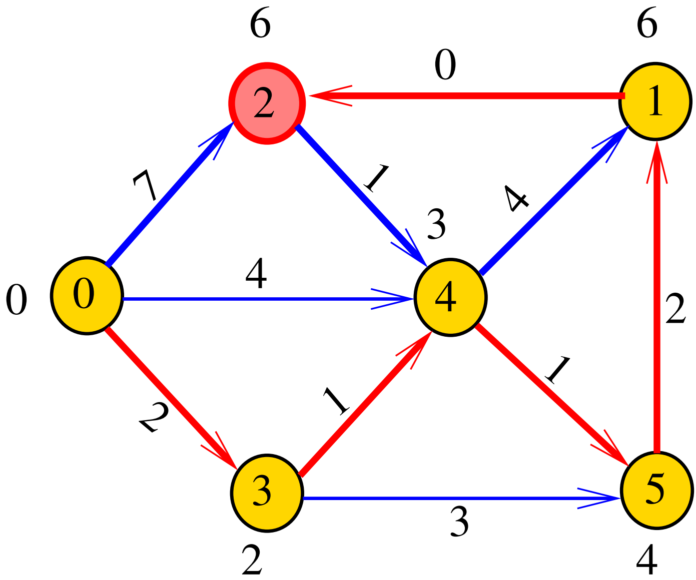
Simulação



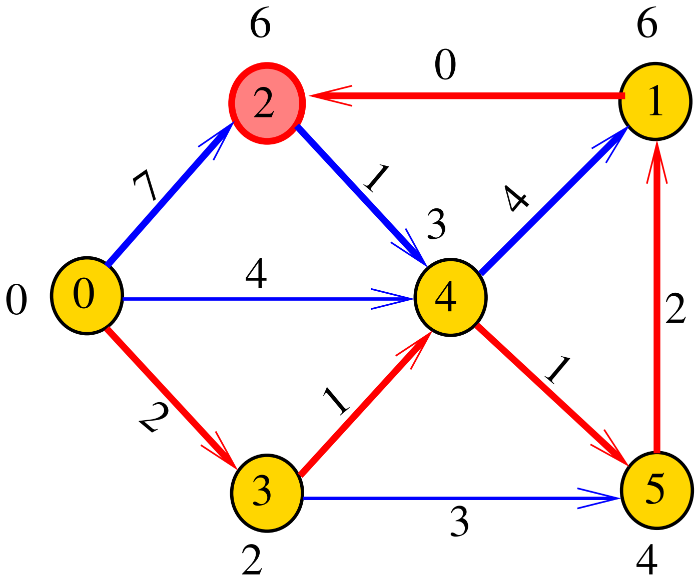
Simulação



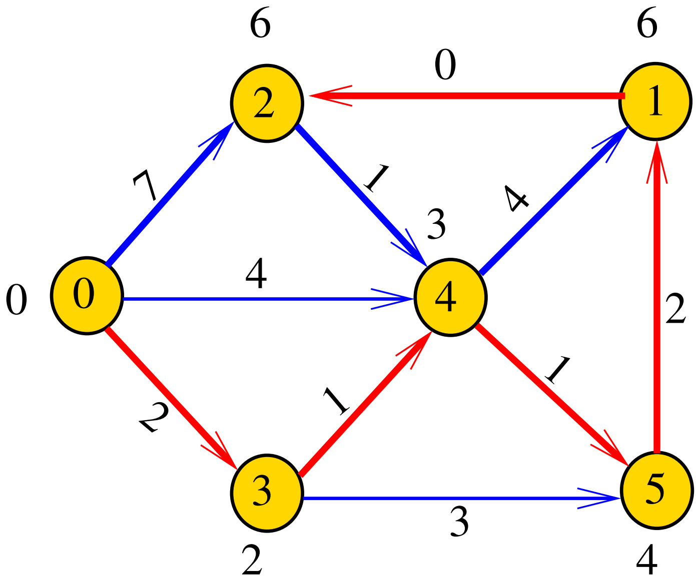
Simulação



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) {
            parnt[w]=v;
            cst[w]=cst[v]+p->cst;
            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);
            }
    }
}
```

Consumo de tempo

linha	número de execuções da linha
2–4	$\Theta(\mathtt{V})$
5	$= 1$ `PQinit`
6–7	$= 1$
8	$= 1$ `PQinsert`
9–10	$\leq \mathtt{V} + 1$ `PQempty` e `PQdelmin`
11	$O(\mathtt{A})$
12–14	$O(\mathtt{V})$
15	$\leq \mathtt{V} - 1$ `PQinsert`
16–19	$O(\mathtt{A})$
20	$\leq \mathtt{A}$ `PQdec`
21	$= 1$ `PQfree`
total	$= O(\mathtt{V} + \mathtt{A}) + ???$

Conclusão

O consumo de tempo da função `dijkstra` é $\mathrm{O}(\mathtt{V} + \mathtt{A})$ mais o consumo de tempo de

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

Conclusão

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

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

AULA 15

Mais algoritmo de Dijkstra

S 21.1 e 21.2

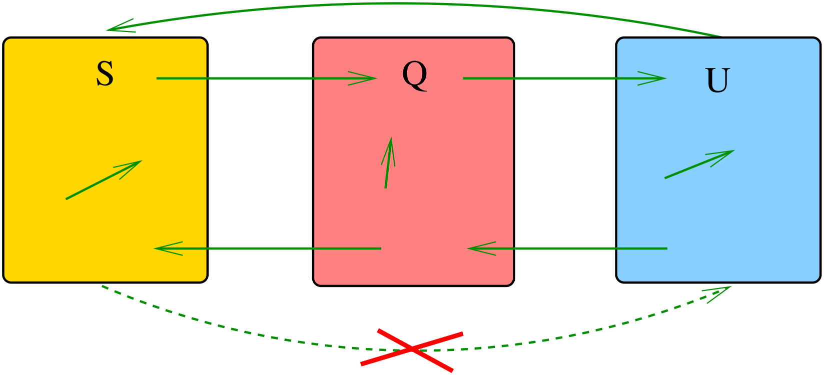
Relações invariantes

S = vértices examinados

Q = vértices visitados = vértices na fila

U = vértices ainda não visitados

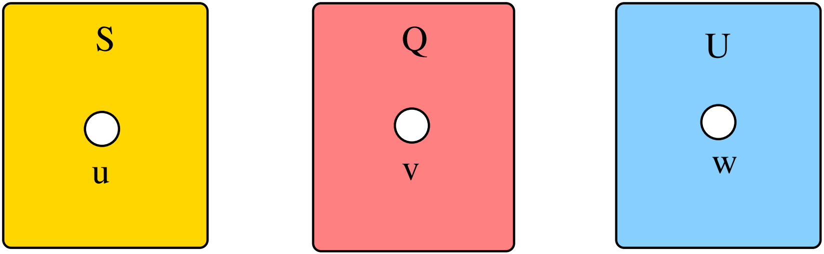
(i0) não existe arco `v-w` com `v` em S e `w` em U



Relações invariantes

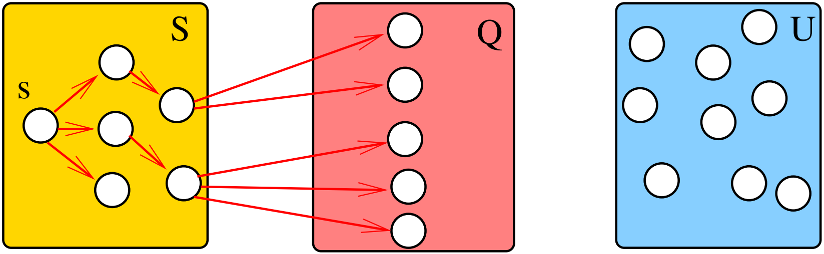
(i1) para cada u em S, v em Q e w em U

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



Relações invariantes

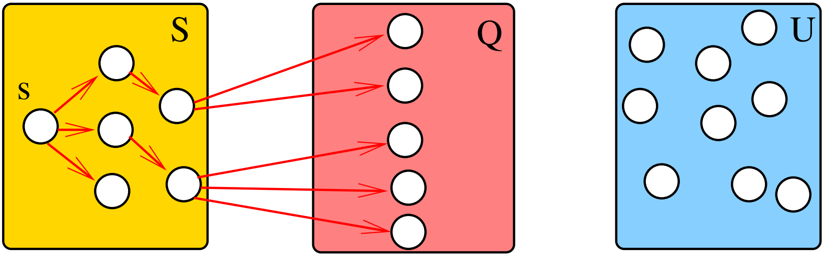
(i2) O vetor `parnt` restrito aos vértices de S e Q determina um árborescência com raiz s



Relações invariantes

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

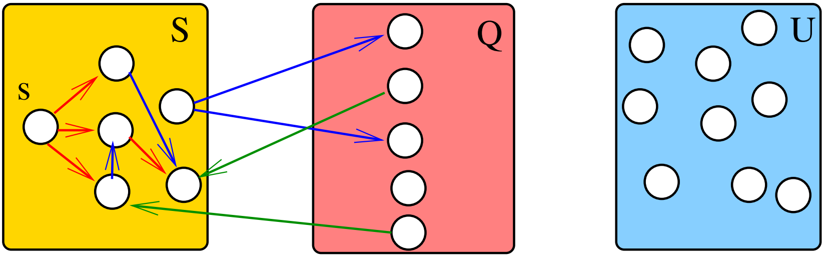
$$\texttt{cst}[\texttt{w}] = \texttt{cst}[\texttt{v}] + \text{custo do arco } \texttt{vw}$$



Relações invariantes

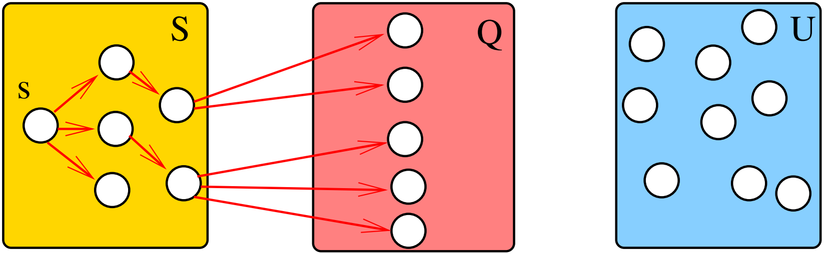
(i4) Para cada arco `v-w` com `v` **ou** `w` em `S` vale que

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

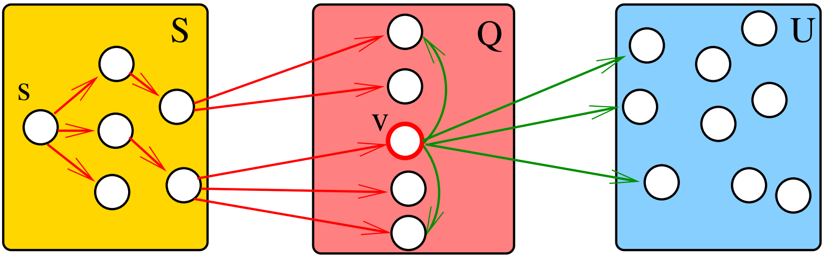


Relações invariantes

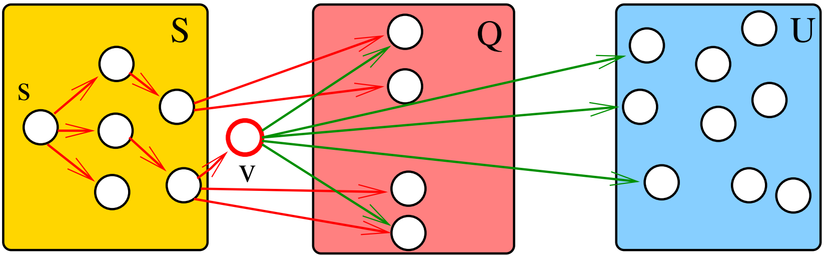
(i5) Para cada vértice `v` em S vale que `cst[v]` é o custo de um caminho mínimo de `s` a `v`.



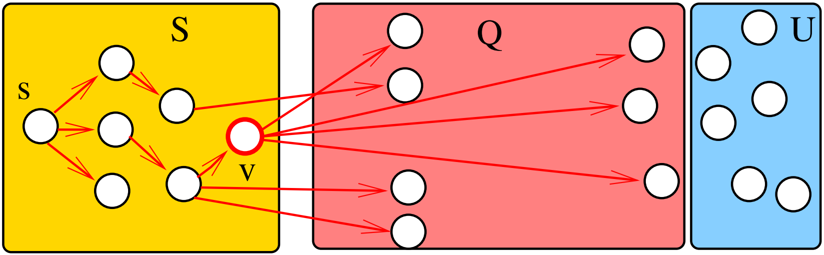
Iteração



Iteração



Iteração



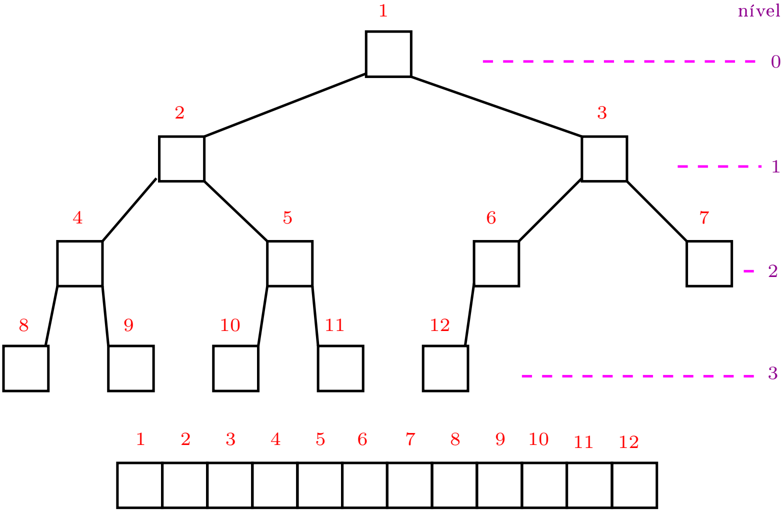
Outra implementação para digrafos densos

```
#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;
        }
   }
}
```

Representação de árvores em vetores



Pais e filhos

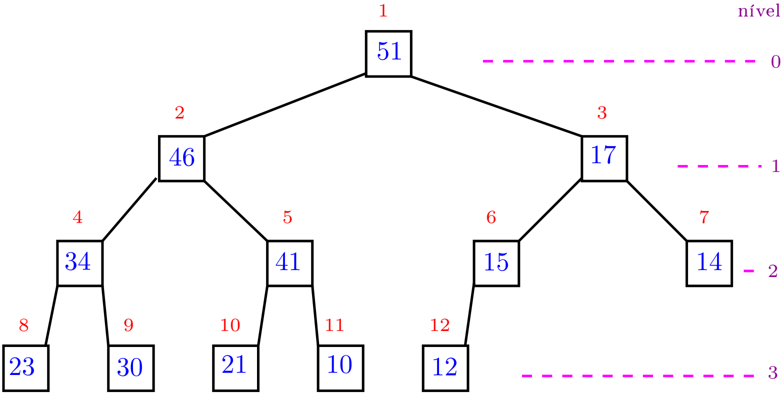
$A[1 \mathinner{..} m]$ é um vetor representando uma árvore.
Diremos que para qualquer índice ou **nó** i,

- $\lfloor i/2 \rfloor$ 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

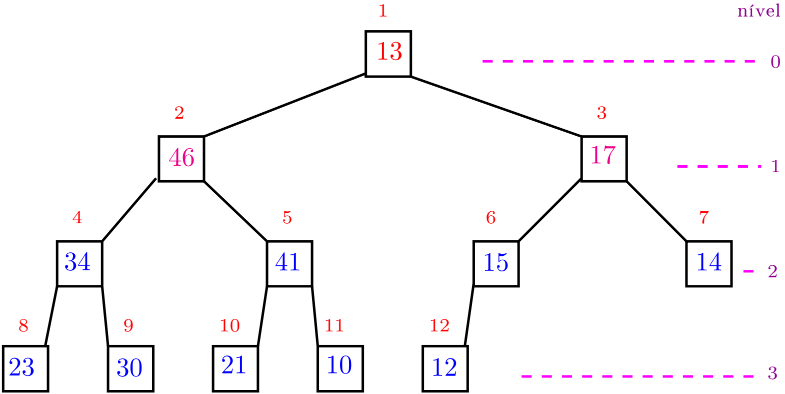
$$A[i, 2i, 2i+1, 4i, 4i+1, 4i+2, 4i+3, 8i, \ldots, 8i+7, \ldots]$$

Max-heap



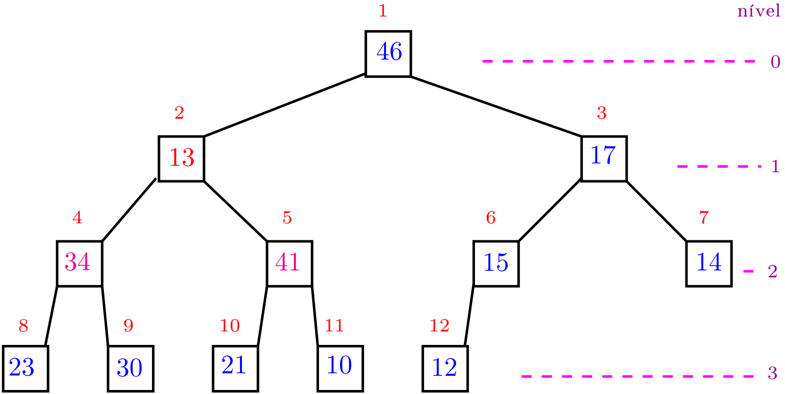
1	2	3	4	5	6	7	8	9	10	11	12
51	46	17	34	41	15	14	23	30	21	10	12

Rotina básica de manipulação de max-heap



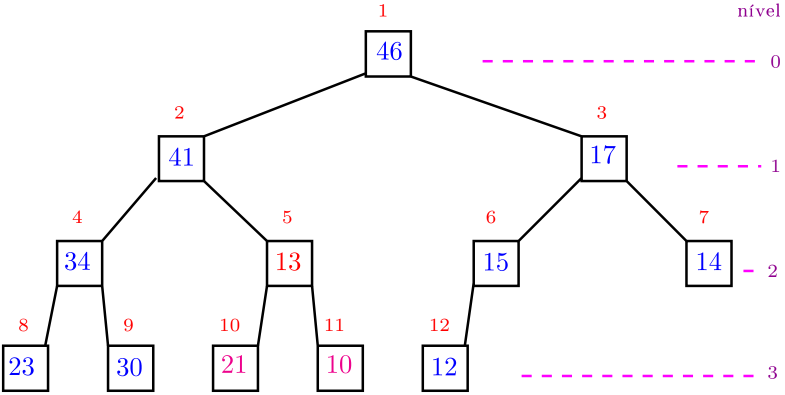
1	2	3	4	5	6	7	8	9	10	11	12
13	46	17	34	41	15	14	23	30	21	10	12

Rotina básica de manipulação de max-heap



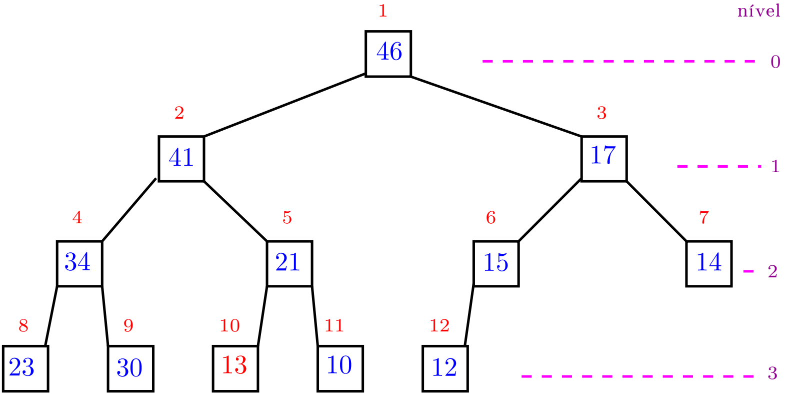
1	2	3	4	5	6	7	8	9	10	11	12
46	13	17	34	41	15	14	23	30	21	10	12

Rotina básica de manipulação de max-heap



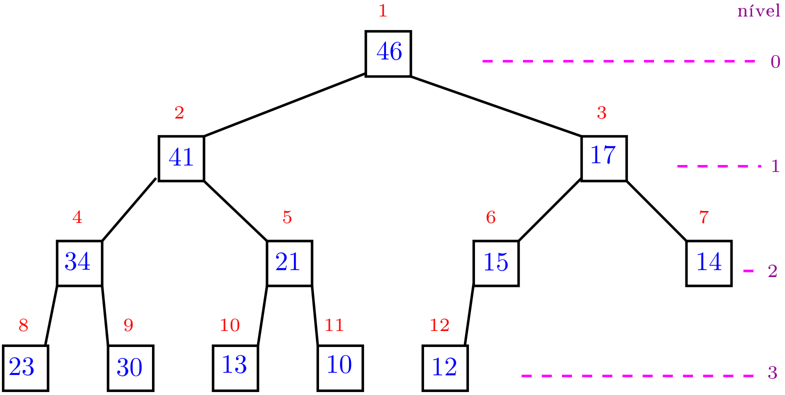
1	2	3	4	5	6	7	8	9	10	11	12
46	41	17	34	13	15	14	23	30	21	10	12

Rotina básica de manipulação de max-heap



1	2	3	4	5	6	7	8	9	10	11	12
46	41	17	34	21	15	14	23	30	13	10	12

Rotina básica de manipulação de max-heap



1	2	3	4	5	6	7	8	9	10	11	12
46	41	17	34	21	15	14	23	30	13	10	12

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 max-heaps e rearranja A de modo que subárvore com raiz i seja max-heap.

```
MAX-HEAPIFY (A, m, i)
    e ← 2i
    d ← 2i + 1
    se e ≤ m e A[e] > A[i]
        então maior ← e
        senão maior ← i
    se d ≤ m e A[d] > A[maior]
        então maior ← d
    se maior ≠ i
        então A[i] ↔ A[maior]
              MAX-HEAPIFY (A, m, maior)
```

Filas com prioridades

Operações que iremos considerar na fila com prioridades:

Maximum(S): devolve o elemento de S com a maior prioridade;

Extract-MAX(S): remove e devolve o elemento em S com a maior prioridade;

Increase-Key(S, s, p): aumenta o valor da prioridade do elemento s para p; e

Insert(S, s, p): insere o elemento s em S com prioridade p.

Implementação com max-heap

Heap-Max (A, m)

Implementação com max-heap

Heap-Max (A, m)

1 **devolva** $A[1]$

Consome tempo ????.

Implementação com max-heap

Heap-Max (A, m)
1 **devolva** $A[1]$

Consome tempo $\Theta(1)$.

Heap-Extract-Max (A, m) $\triangleright$ $m \geq 1$

Implementação com max-heap

Heap-Max (A, m)
1 **devolva** $A[1]$

Consome tempo $\Theta(1)$.

Heap-Extract-Max (A, m) $\triangleright$ $m \geq 1$
1 $max \leftarrow A[1]$
2 $A[1] \leftarrow A[m]$
3 $m \leftarrow m - 1$
4 Max-Heapify $(A, m, 1)$
5 **devolva** max

Consome tempo $????$.

Implementação com max-heap

```
Heap-Max (A, m)
    devolva A[1]
```

Consome tempo $\Theta(1)$.

```
Heap-Extract-Max (A, m)   ▷ m ≥ 1
    max ← A[1]
    A[1] ← A[m]
    m ← m − 1
    Max-Heapify (A, m, 1)
    devolva max
```

Consome tempo $O(\lg m)$.

Implementação com max-heap

Heap-Increase-Key $(A, \texttt{i}, prior)$ $\triangleright$ $prior \geq A[i]$

Implementação com max-heap

Heap-Increase-Key $(A, \texttt{i}, prior)$ $\triangleright$ $prior \geq A[i]$

1 $A[\texttt{i}] \leftarrow prior$
2 **enquanto** $\texttt{i} > 1$ e $A[\lfloor \texttt{i}/2 \rfloor] < A[\texttt{i}]$ **faça**
3 $A[\texttt{i}] \leftrightarrow A[\lfloor \texttt{i}/2 \rfloor]$
4 $\texttt{i} \leftarrow \lfloor \texttt{i}/2 \rfloor$

Consome tempo ?????.

Implementação com max-heap

Heap-Increase-Key $(A, \mathtt{i}, prior)$ $\quad \triangleright\ prior \geq A[i]$

```
A[i] ← prior
enquanto i > 1 e A[⌊i/2⌋] < A[i] faça
      A[i] ↔ A[⌊i/2⌋]
      i ← ⌊i/2⌋
```

Consome tempo $\mathrm{O}(\lg m)$.

Max-Heap-Insert $(A, m, prior)$

Implementação com max-heap

Heap-Increase-Key $(A, \mathtt{i}, prior)$ $\triangleright$ $prior \geq A[i]$

```
A[i] ← prior
enquanto i > 1 e A[⌊i/2⌋] < A[i] faça
        A[i] ↔ A[⌊i/2⌋]
        i ← ⌊i/2⌋
```

Consome tempo $O(\lg m)$.

Max-Heap-Insert $(A, m, prior)$

```
m ← m + 1
A[m] ← −∞
Heap-Increase-Key (A, m, prior)
```

Consome tempo ?????.

Implementação com max-heap

Heap-Increase-Key $(A, \texttt{i}, prior)$ $\triangleright$ $prior \geq A[i]$

$A[\texttt{i}] \leftarrow prior$

enquanto $\texttt{i} > 1$ e $A[\lfloor \texttt{i}/2 \rfloor] < A[\texttt{i}]$ **faça**

  $A[\texttt{i}] \leftrightarrow A[\lfloor \texttt{i}/2 \rfloor]$

  $\texttt{i} \leftarrow \lfloor \texttt{i}/2 \rfloor$

Consome tempo $\mathrm{O}(\lg m)$.

Max-Heap-Insert $(A, m, prior)$

$m \leftarrow m + 1$

$A[m] \leftarrow -\infty$

Heap-Increase-Key $(A, m, prior)$

Consome tempo $\mathrm{O}(\lg m)$.

Consumo de tempo Min-Heap

	heap	d-heap	fibonacci heap
INSERT	$O(\lg V)$	$O(\log_D V)$	$O(1)$
EXTRACT-MIN	$O(\lg V)$	$O(\log_D V)$	$O(\lg V)$
DECREASE-KEY	$O(\lg V)$	$O(\log_D V)$	$O(1)$
dijkstra	$O(A \lg V)$	$O(A \log_D V)$	$O(A + V \lg V)$

Consumo de tempo Min-Heap

	bucket heap	radix heap
INSERT	$O(1)$	$O(\lg(VC)R$
EXTRACT-MIN	$O(C)$	$O(\lg(VC)$
DECREASE-KEY	$O(1)$	$O(A + V \lg(VC))$
dijkstra	$O(A + VC)$	$O(A + V \lg(VC))$

C = maior custo de um arco.