MAC0323 Algoritmos e Estruturas de Dados II

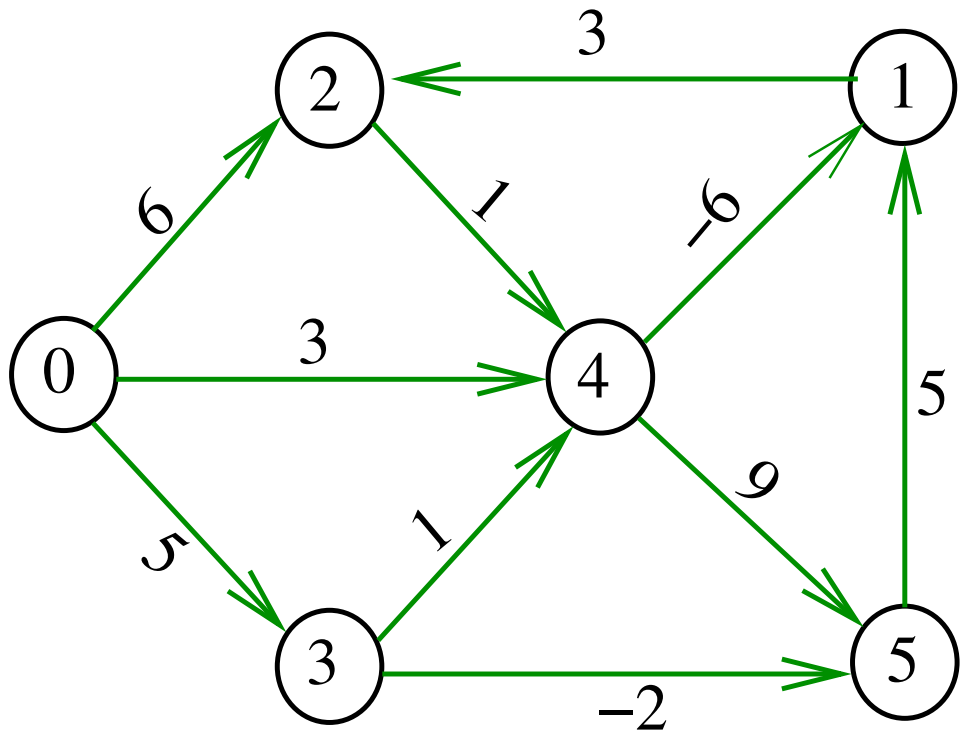
Edição 2020 – 2

Caminhos de custo mínimo

Fonte: The Shortest Distance WoW Achievement Fast

Custo de um caminho

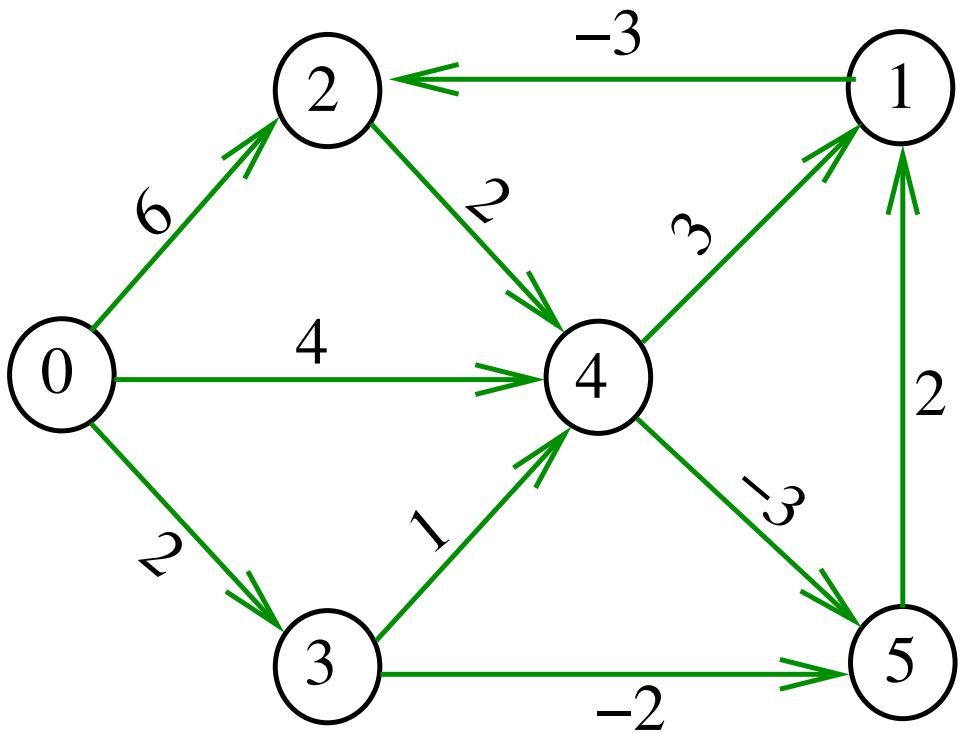
Custo de um caminho é
a soma dos custos de seus arcos.
Custo do caminho 0–2–4–5 é 16.
Custo do caminho 0–2–4–1–2–4–5 é 14.
Custo do caminho 0–2–4–1–2–4–1–2–4–5 é 12.



Caminho mínimo

Um caminho P tem **custo mínimo**
se o custo de P é menor ou igual ao custo de
todo caminho com a mesma origem e término.

O caminho 0–3–4–5–1–2 é mínimo, tem custo -1.



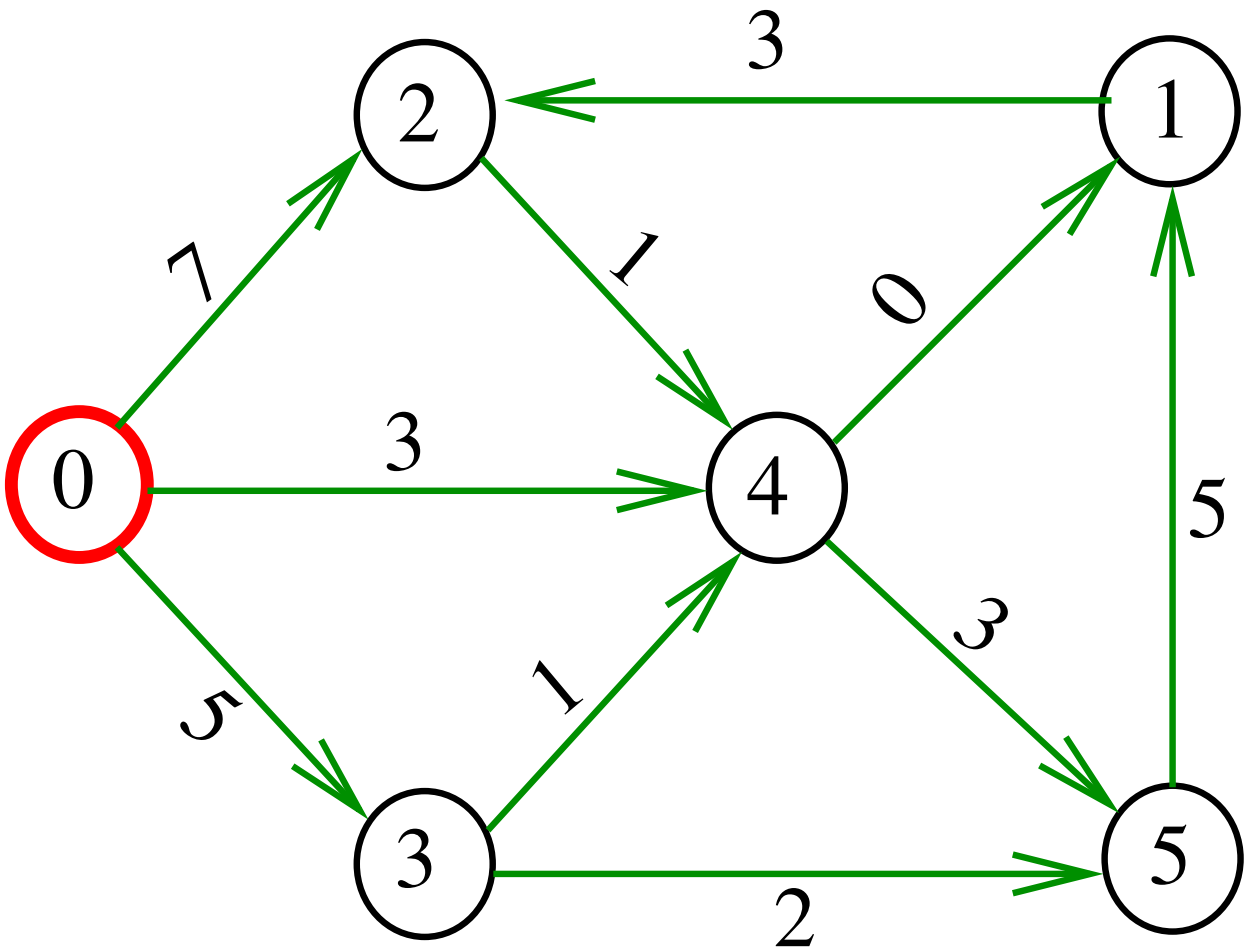
Problema

Problema dos Caminhos Mínimos com Origem Fixa (*Single-source Shortest Paths Problem*):

> *Dado um vértice s de um digrafo com custos* ***não-negativos*** *nos arcos, encontrar, para cada vértice t que pode ser alcançado a partir de s, um caminho mínimo simples de s a t.*

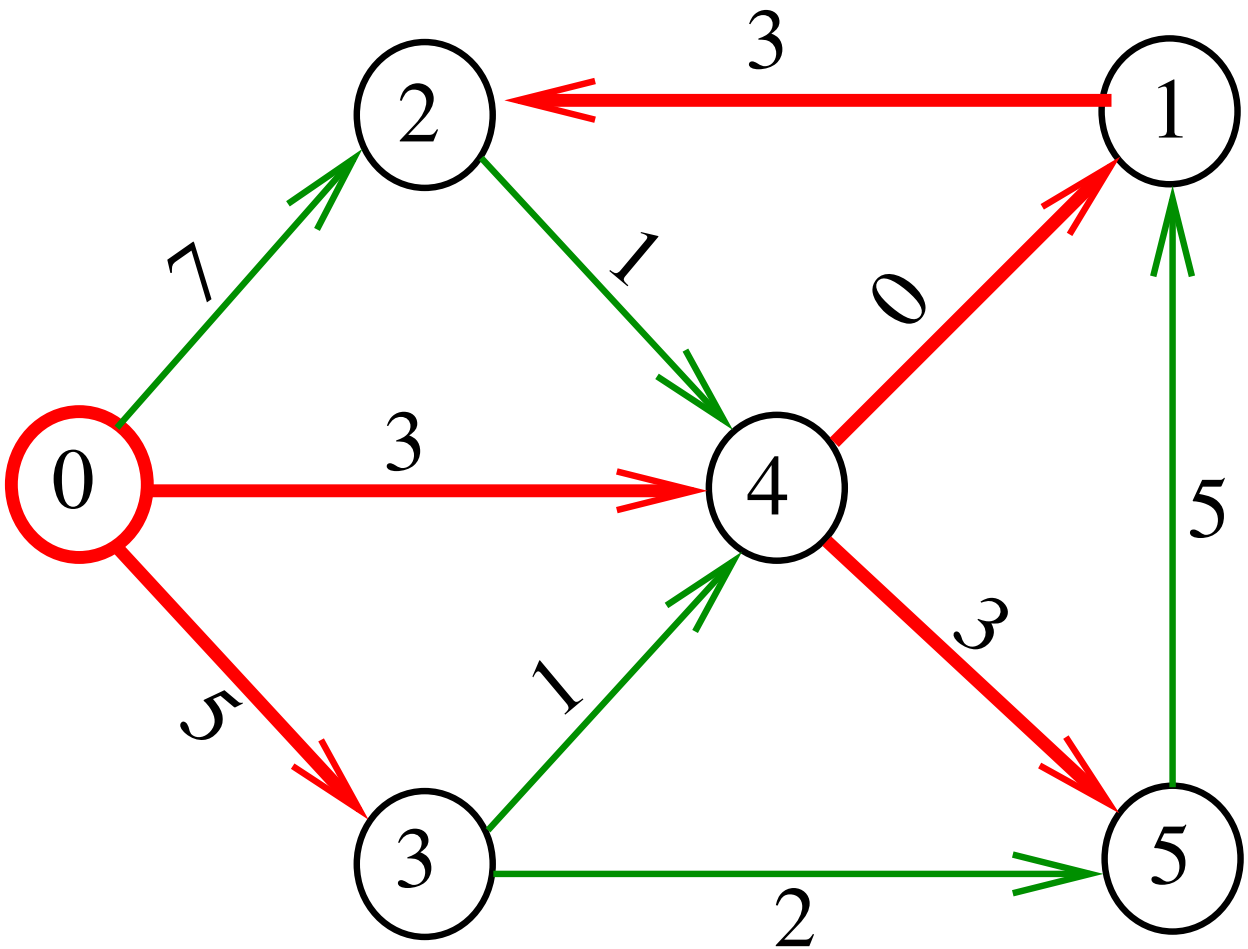
Exemplo

Entra:



Exemplo

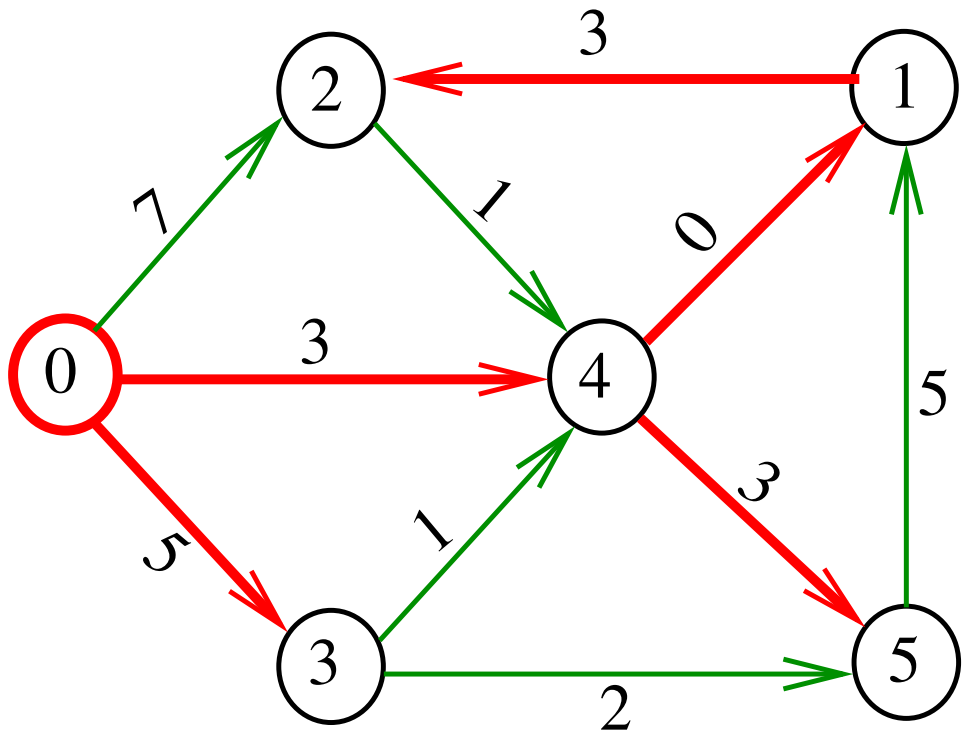
Sai:



Arborescência de caminhos mínimos

Uma arborescência com raiz s é de **caminhos mínimos** (= *shortest-paths tree* = *SPT*) se, para todo vértice t que pode ser alcançado a partir de s,

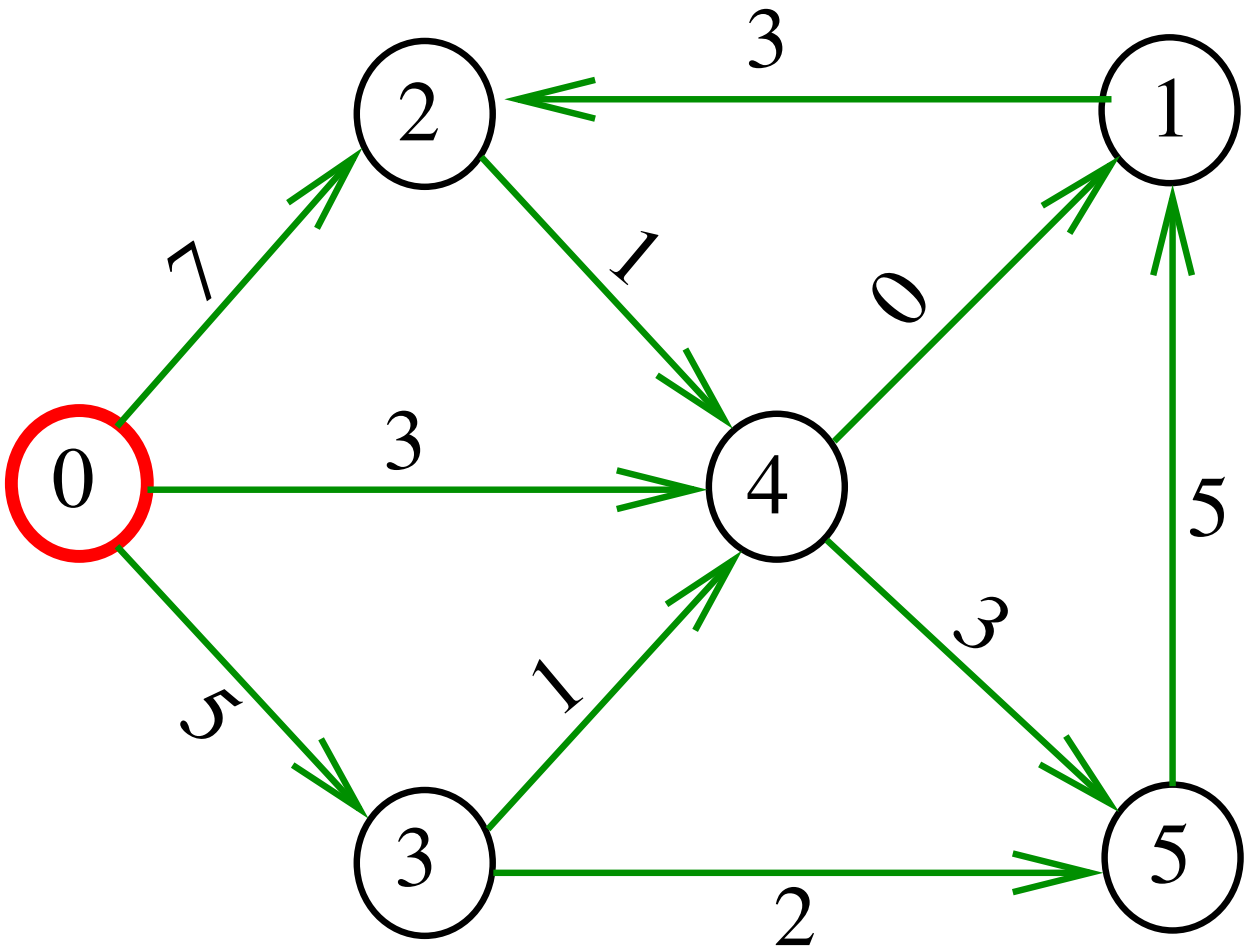
> *o único caminho de s a t na arborescência é um caminho mínimo.*



Problema da SPT

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

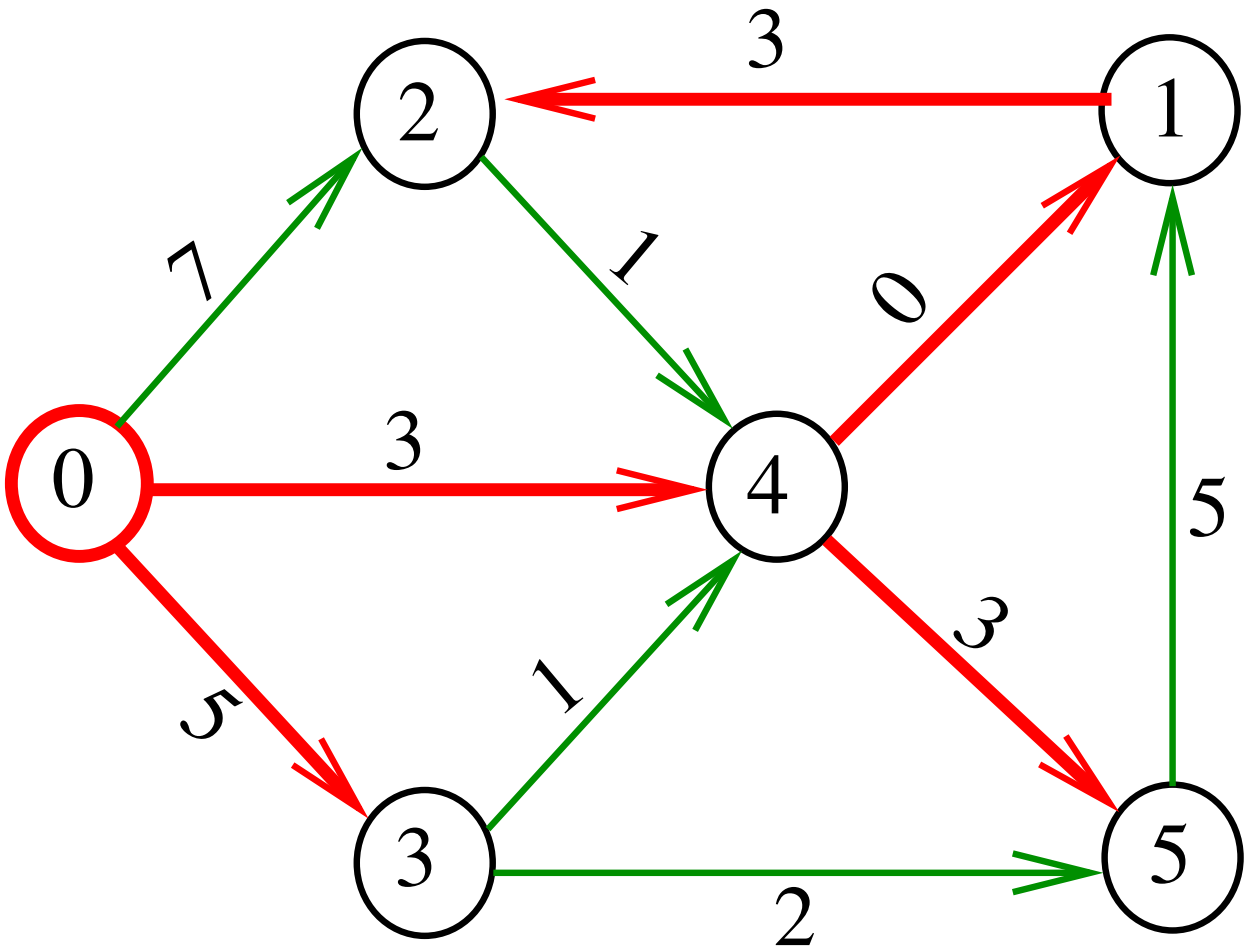
Entra:



Problema da SPT

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

Sai:

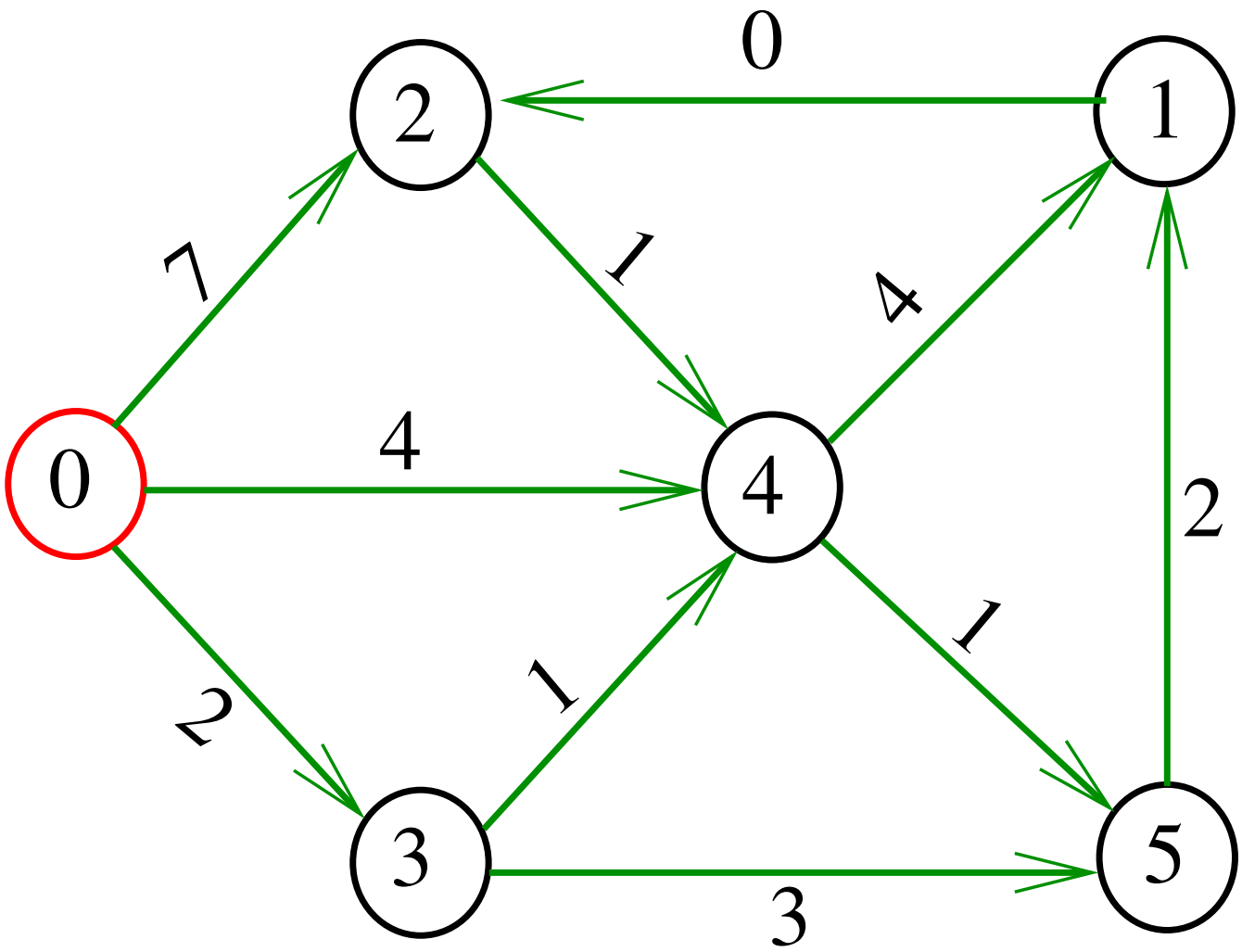


AULA 23

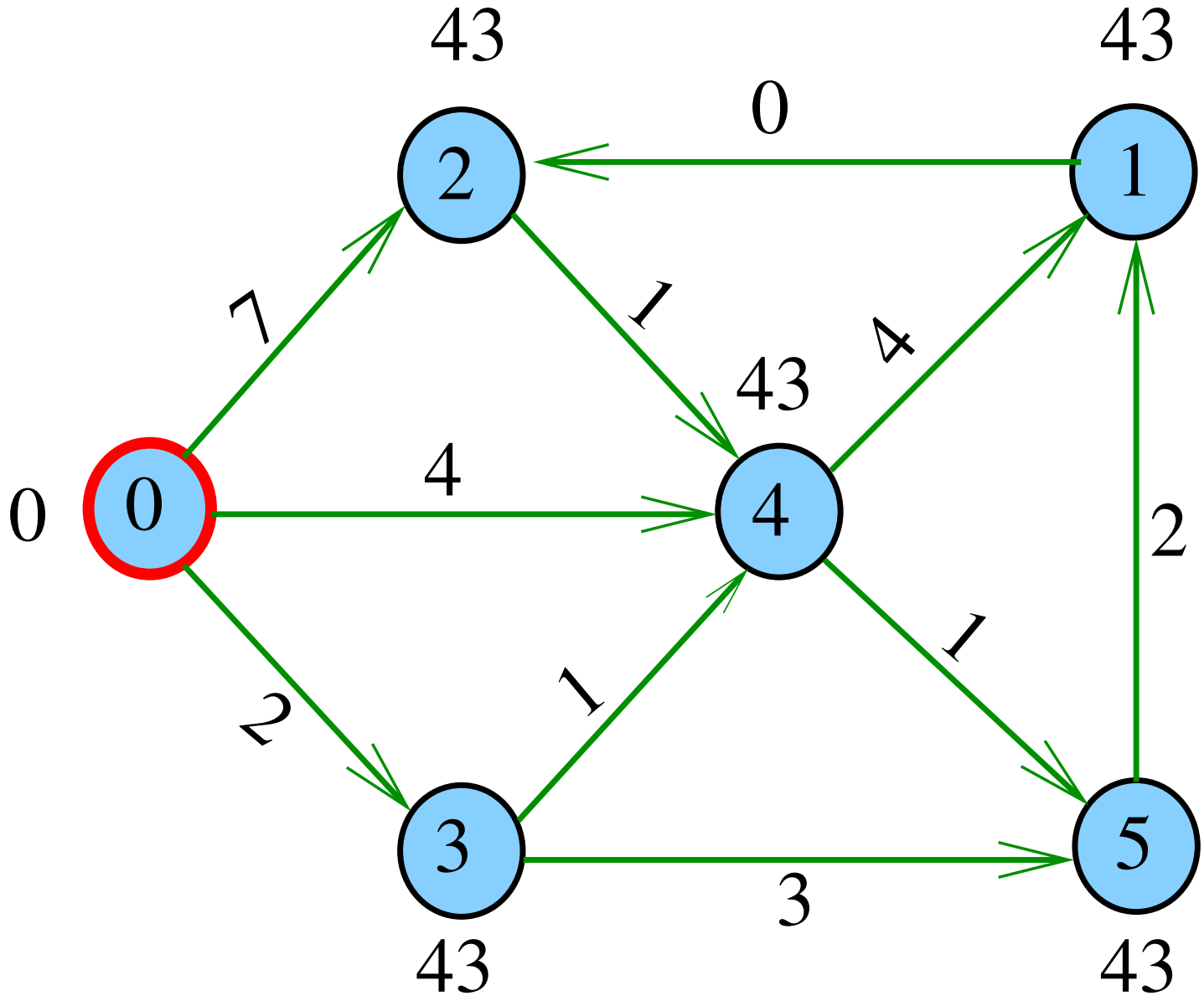
Algoritmo de Dijkstra

Fonte: What S So Great About A World Flight Paths Map Spatial Ly In

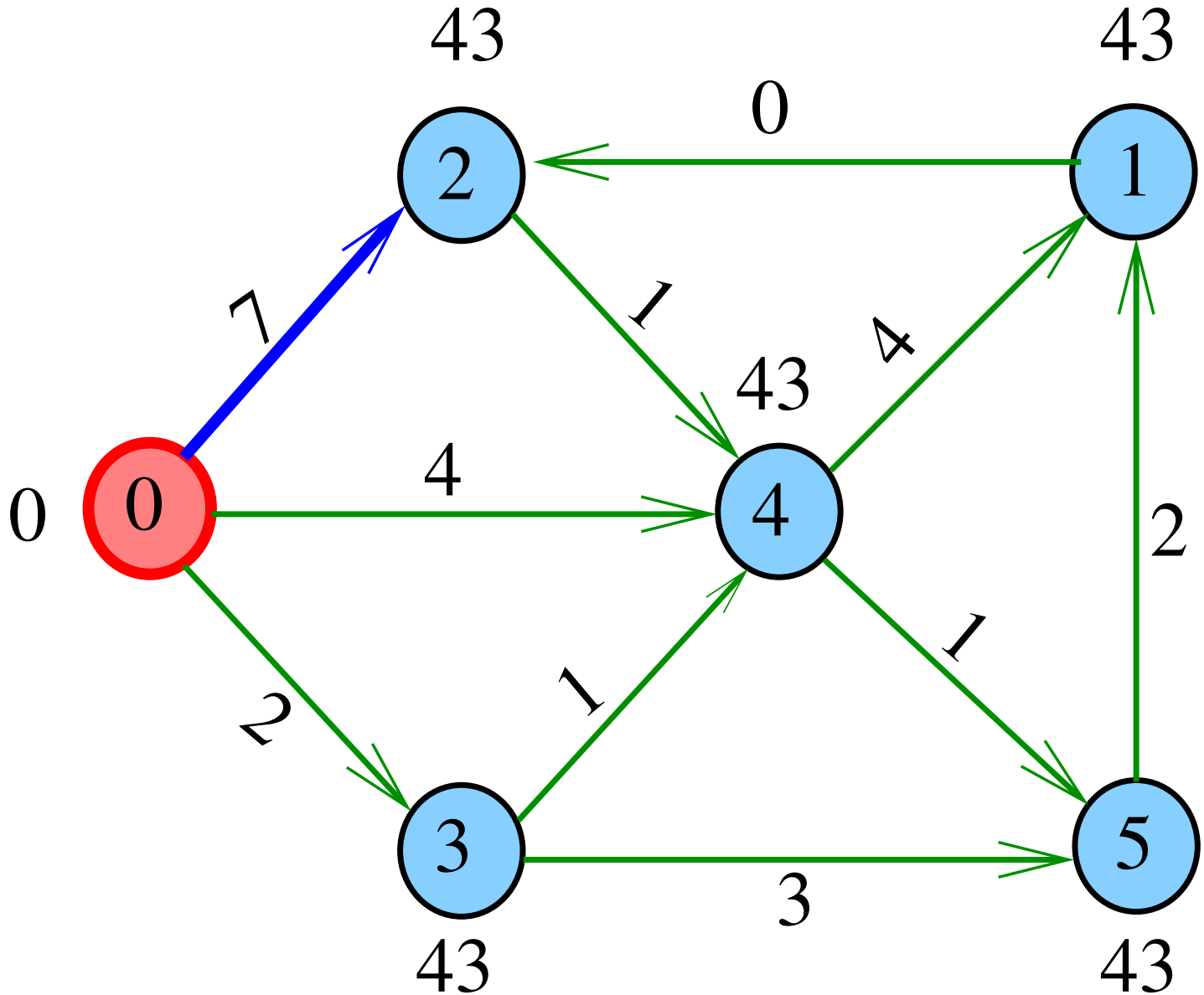
Simulação



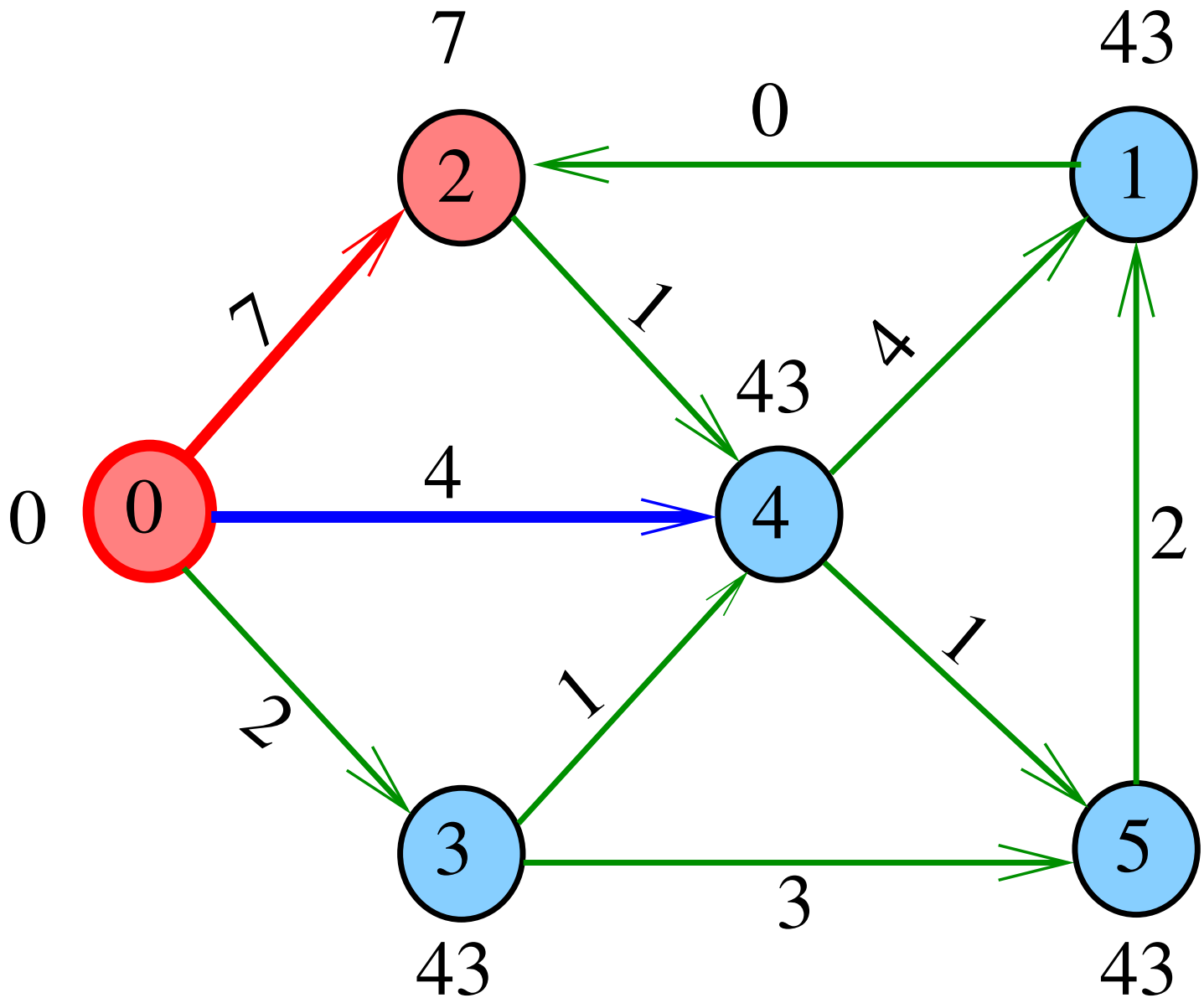
Simulação



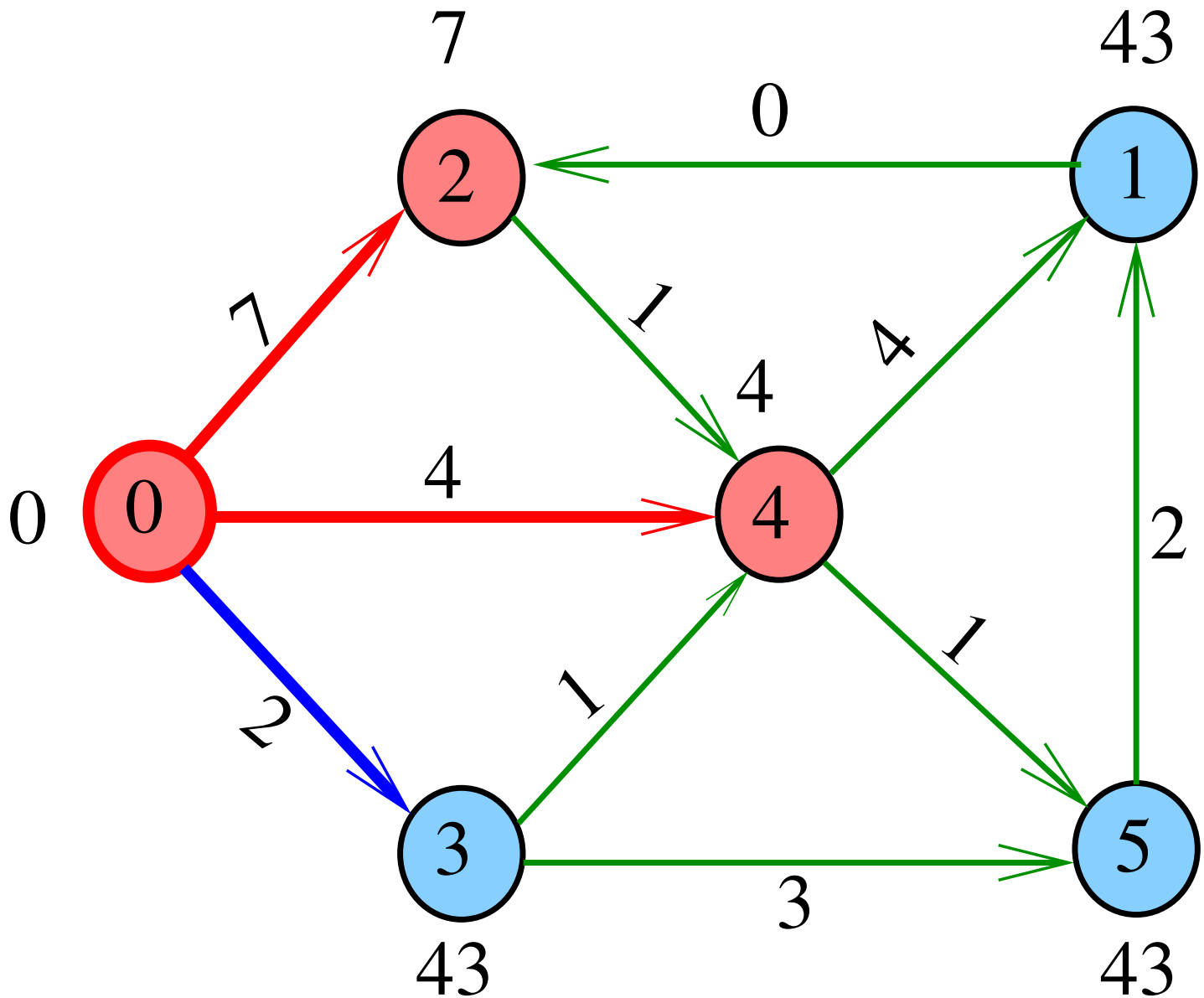
Simulação



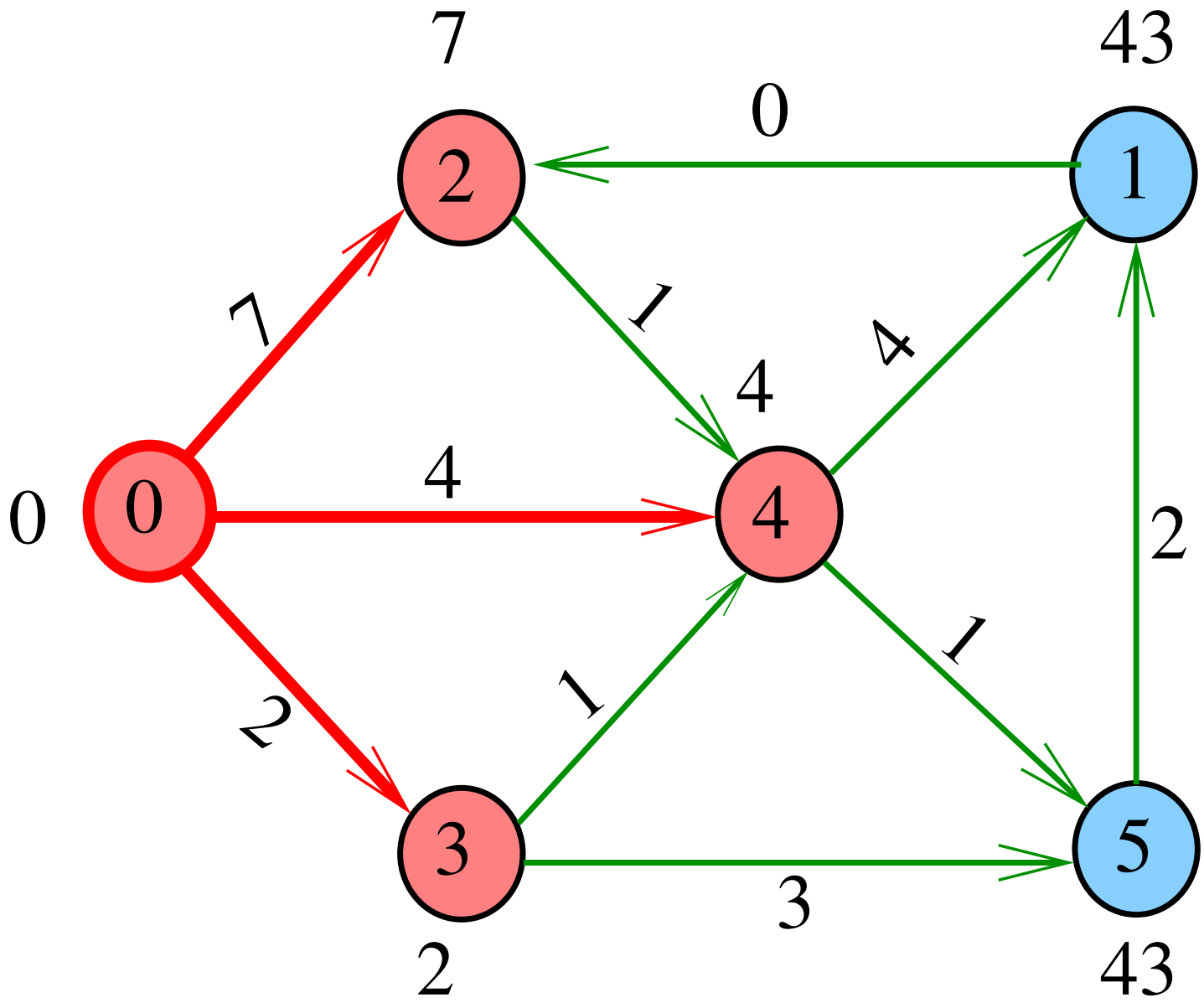
Simulação



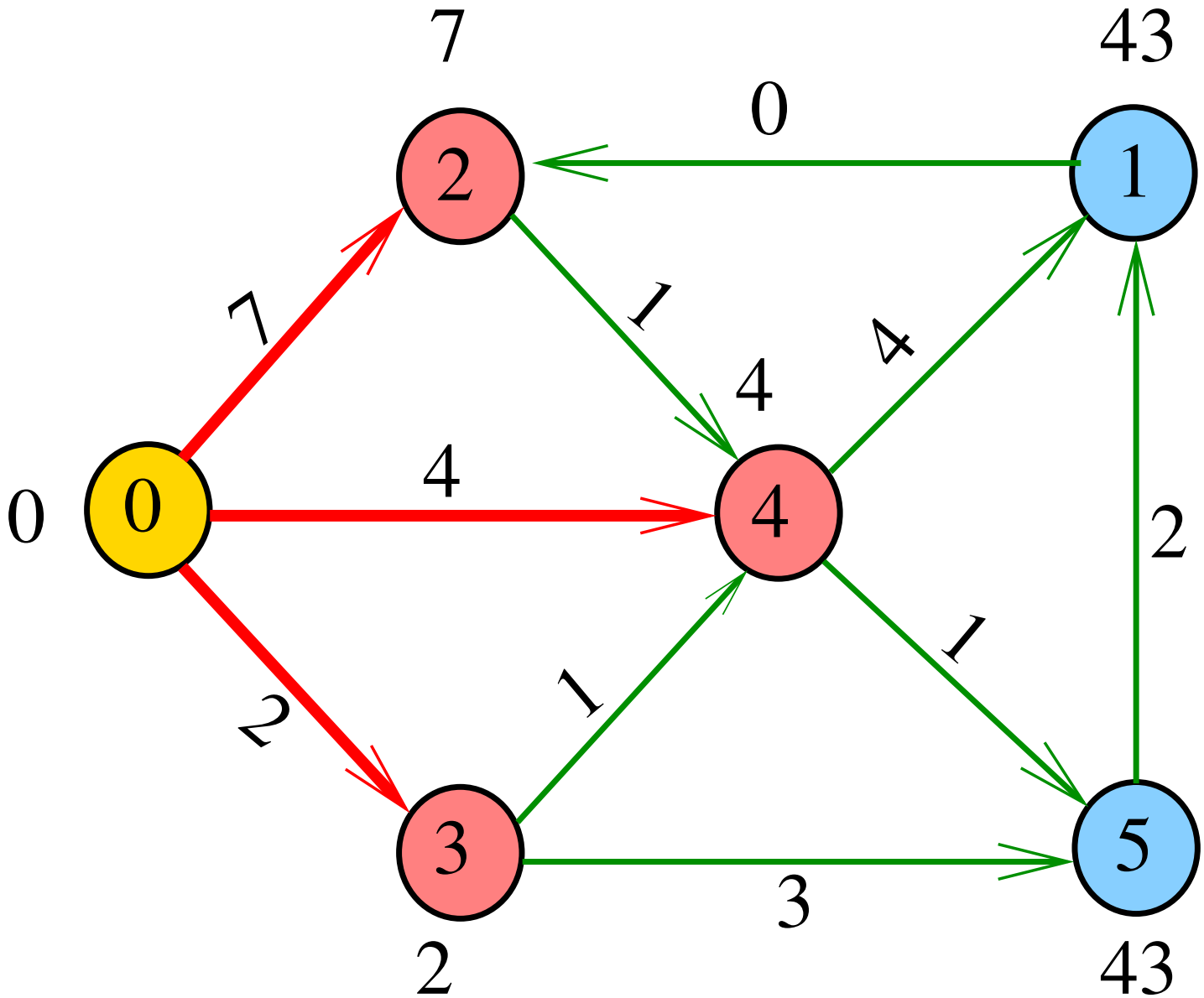
Simulação



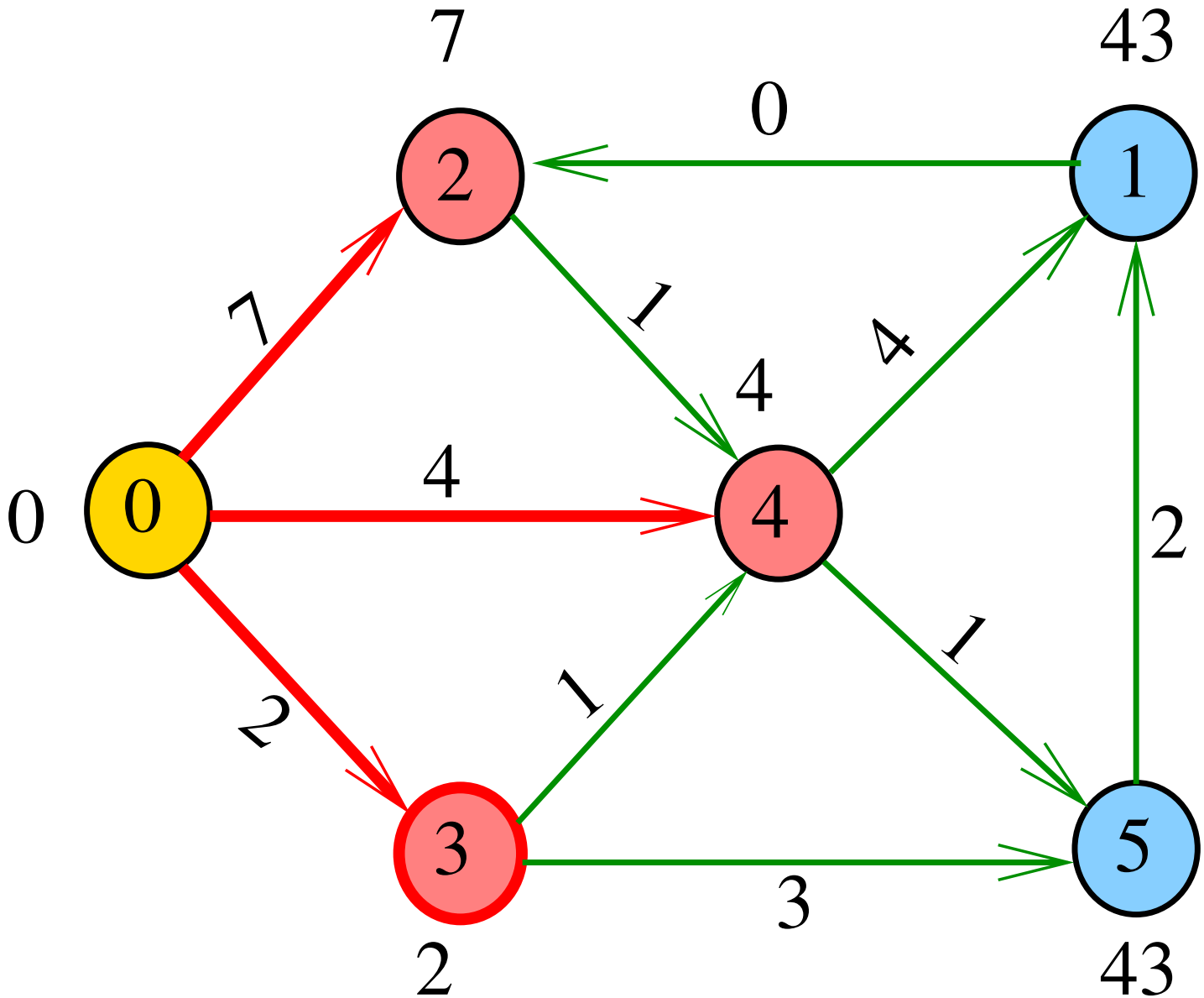
Simulação



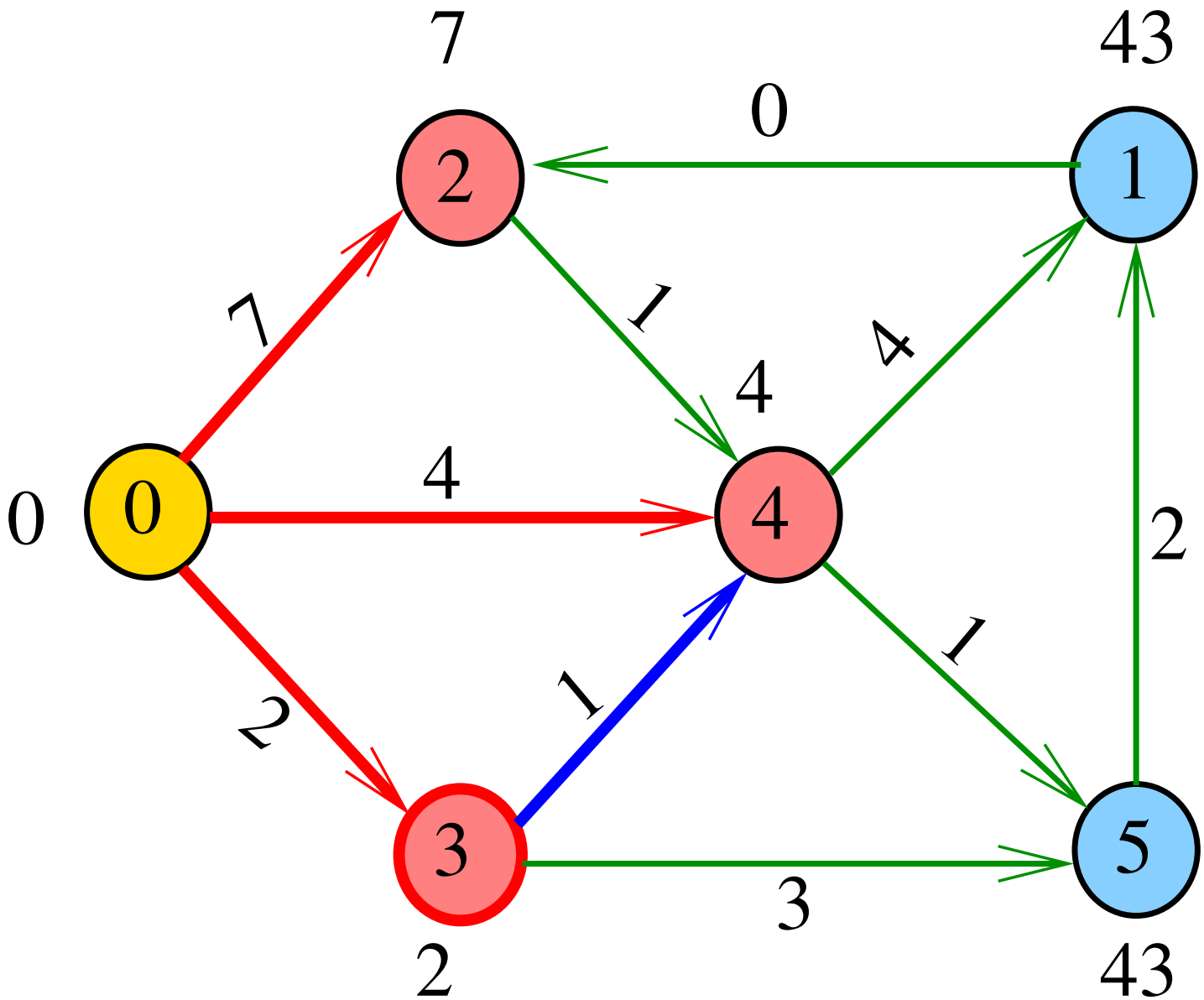
Simulação



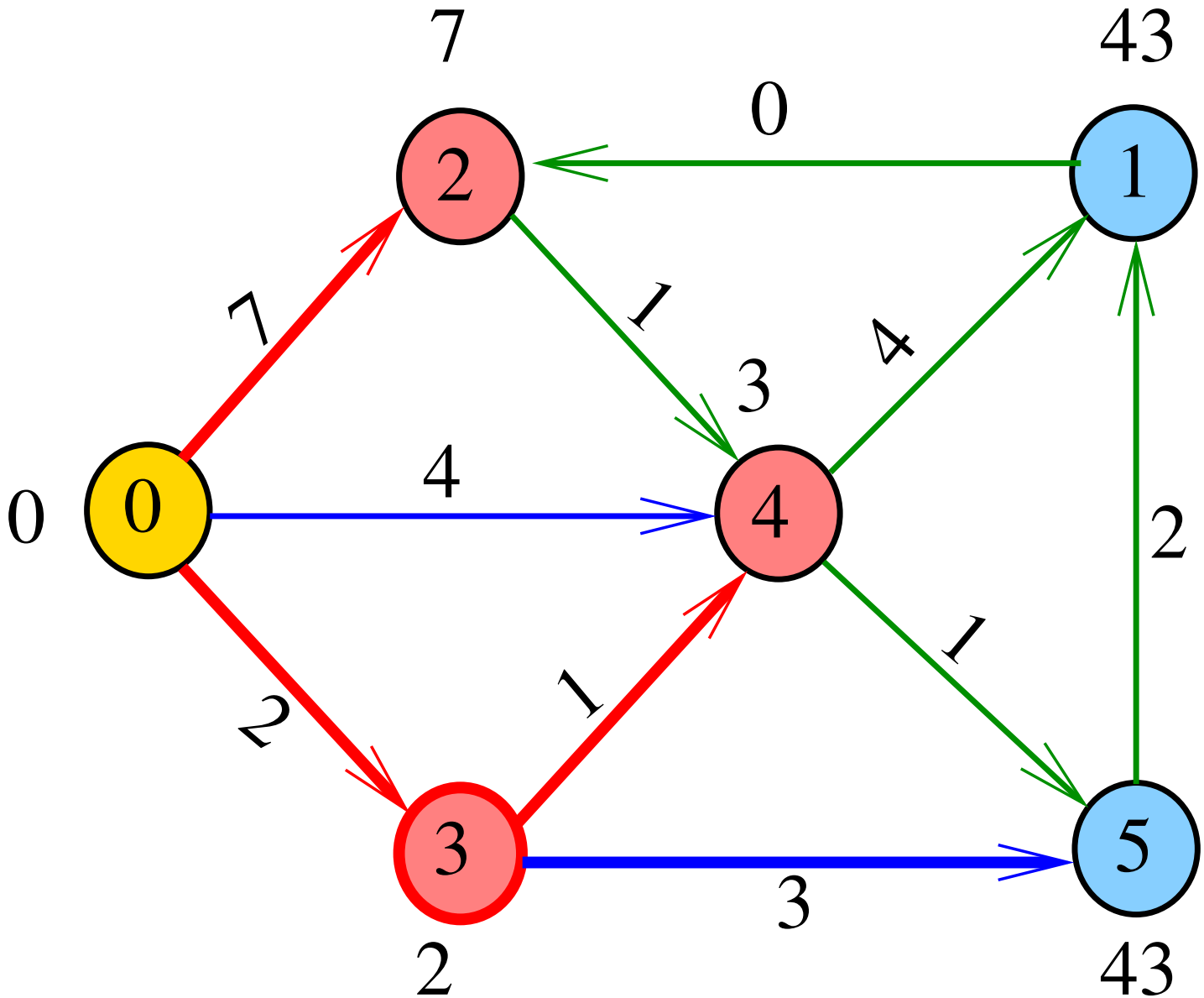
Simulação



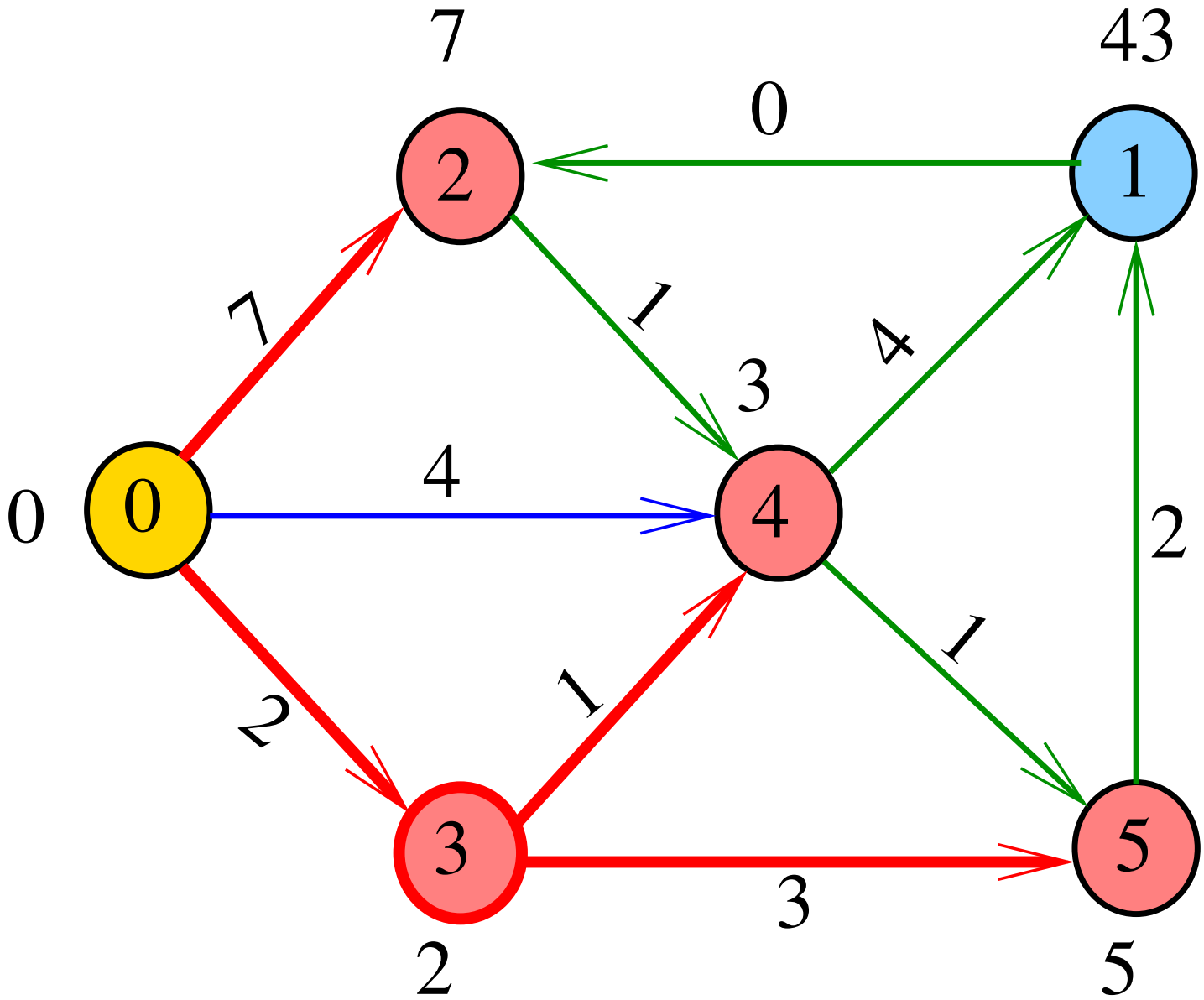
Simulação



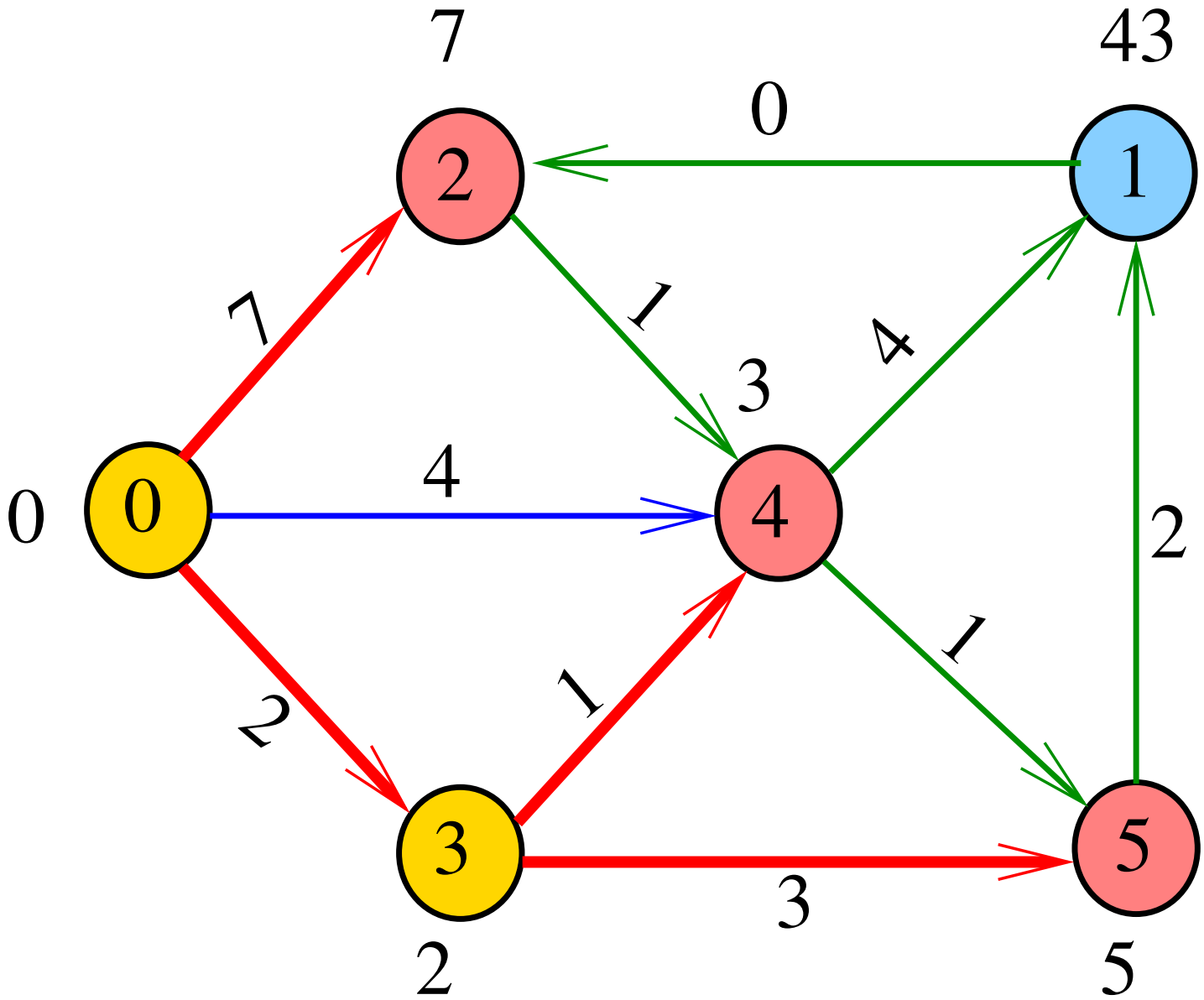
Simulação



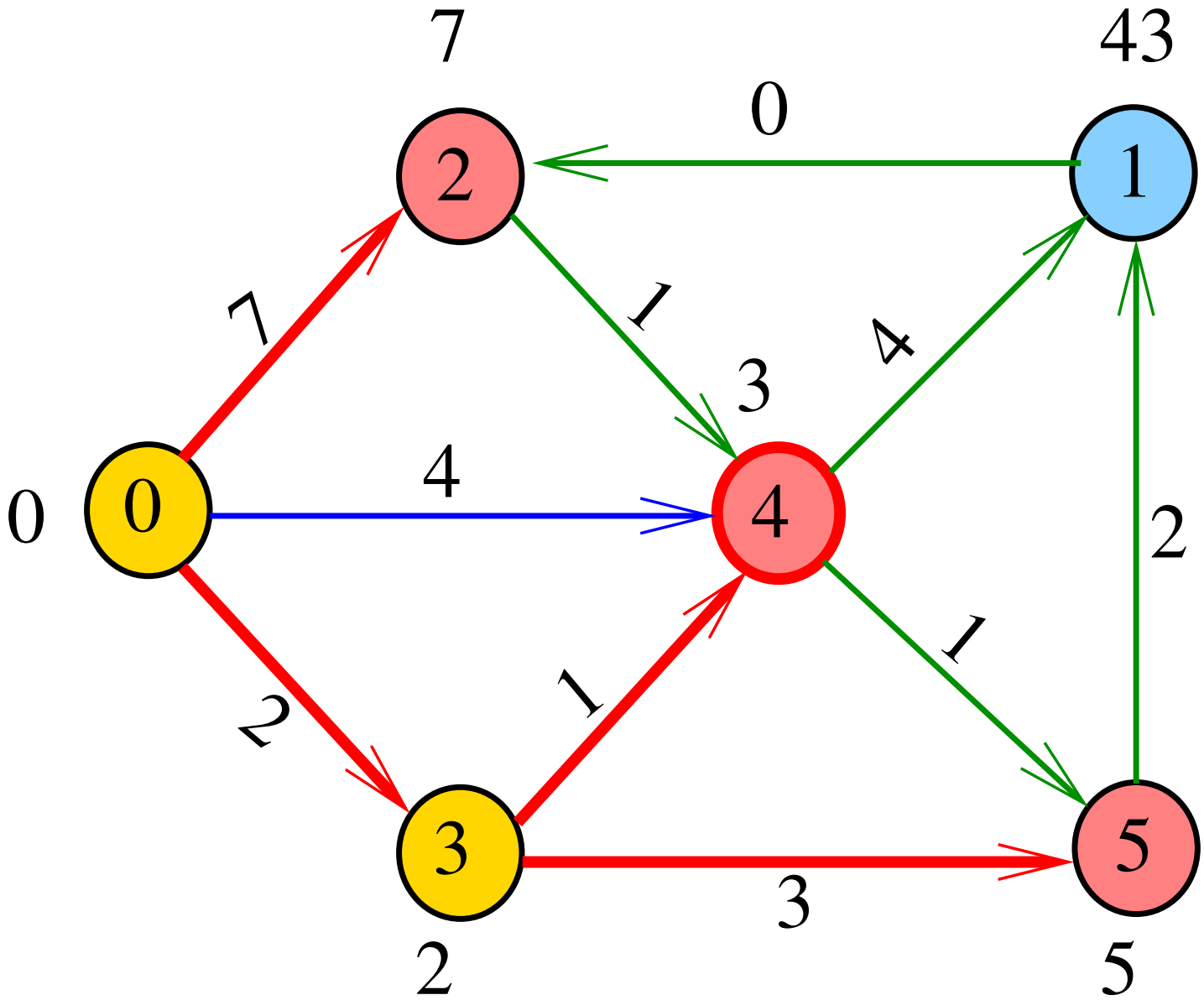
Simulação



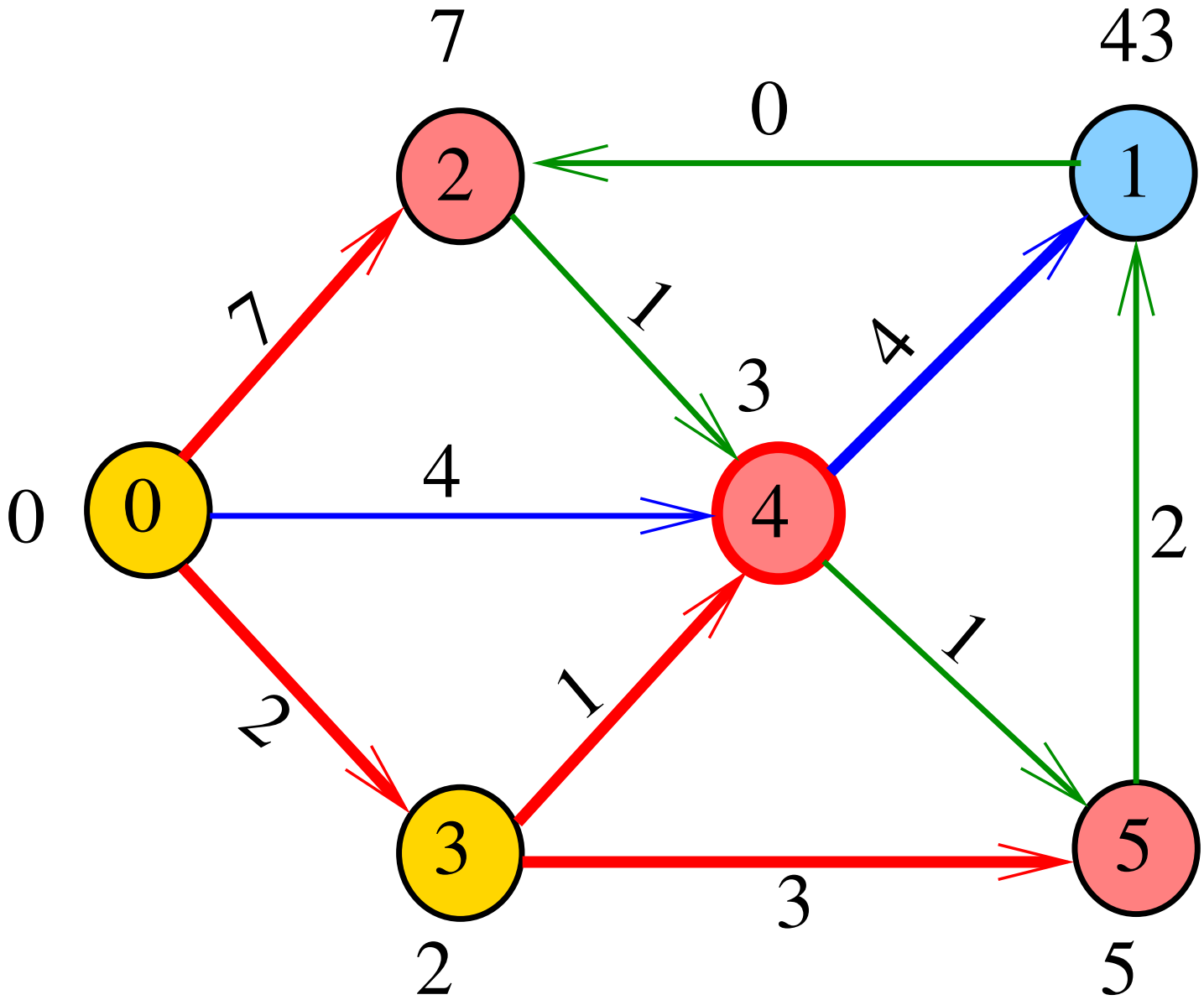
Simulação



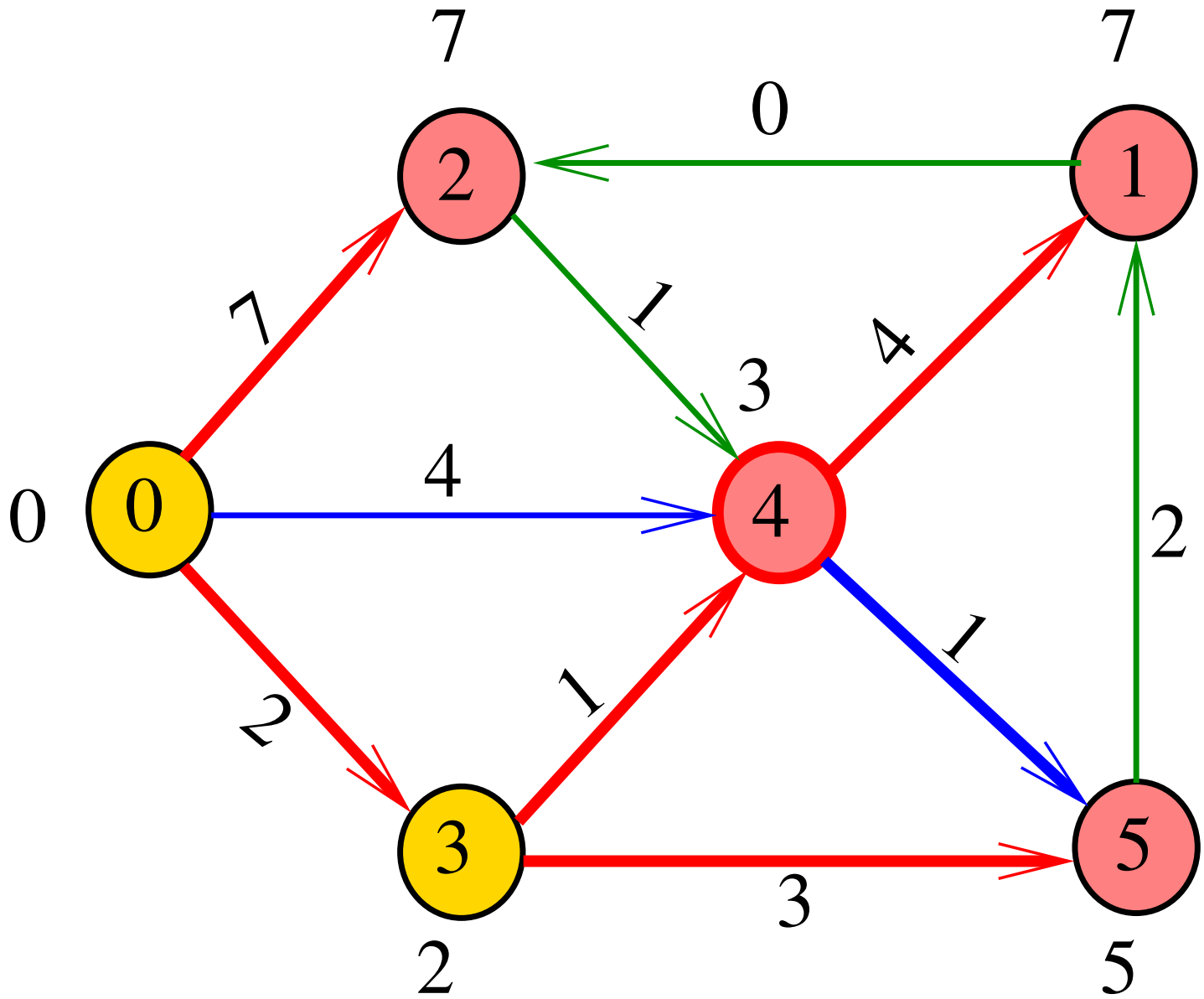
Simulação



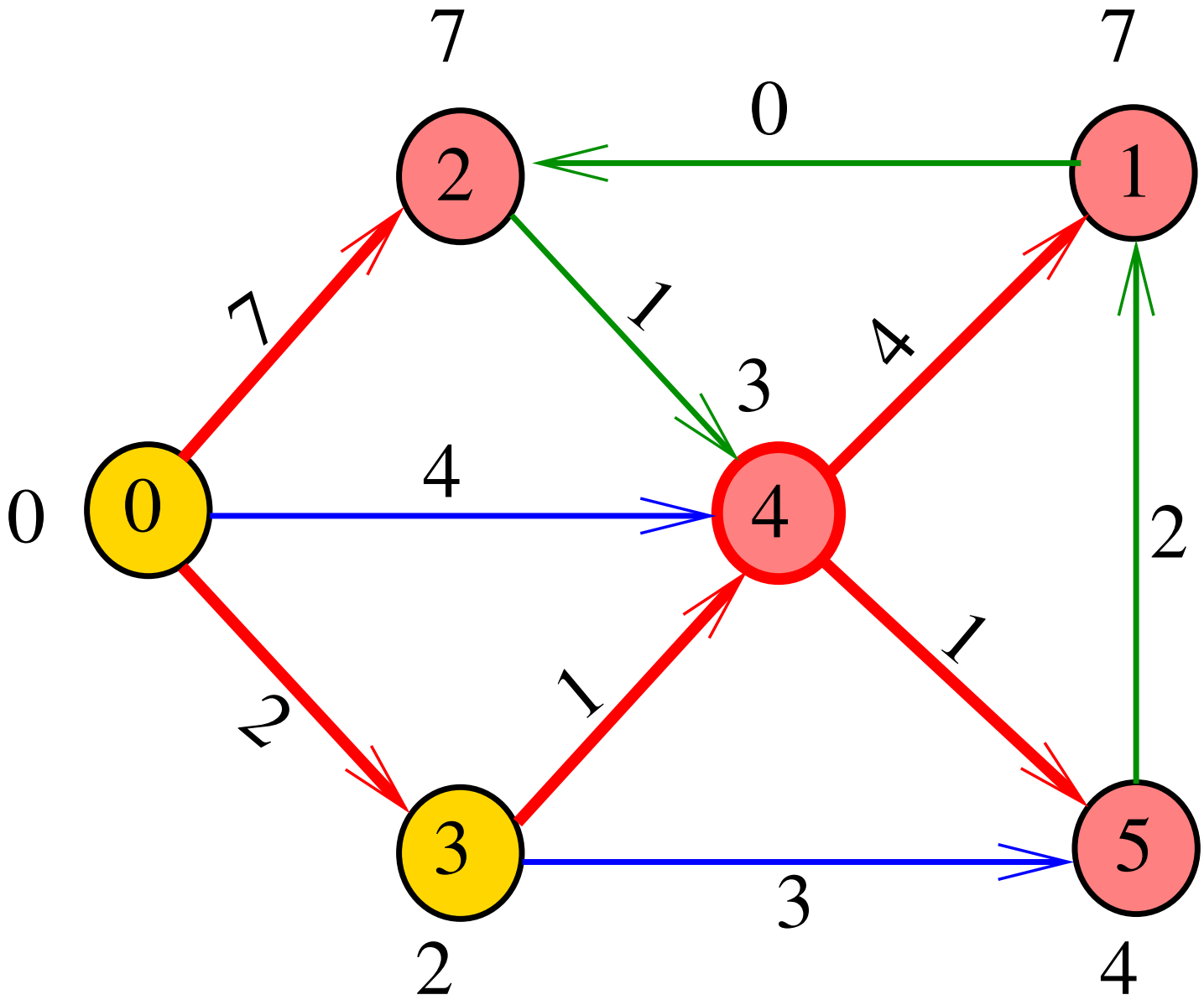
Simulação



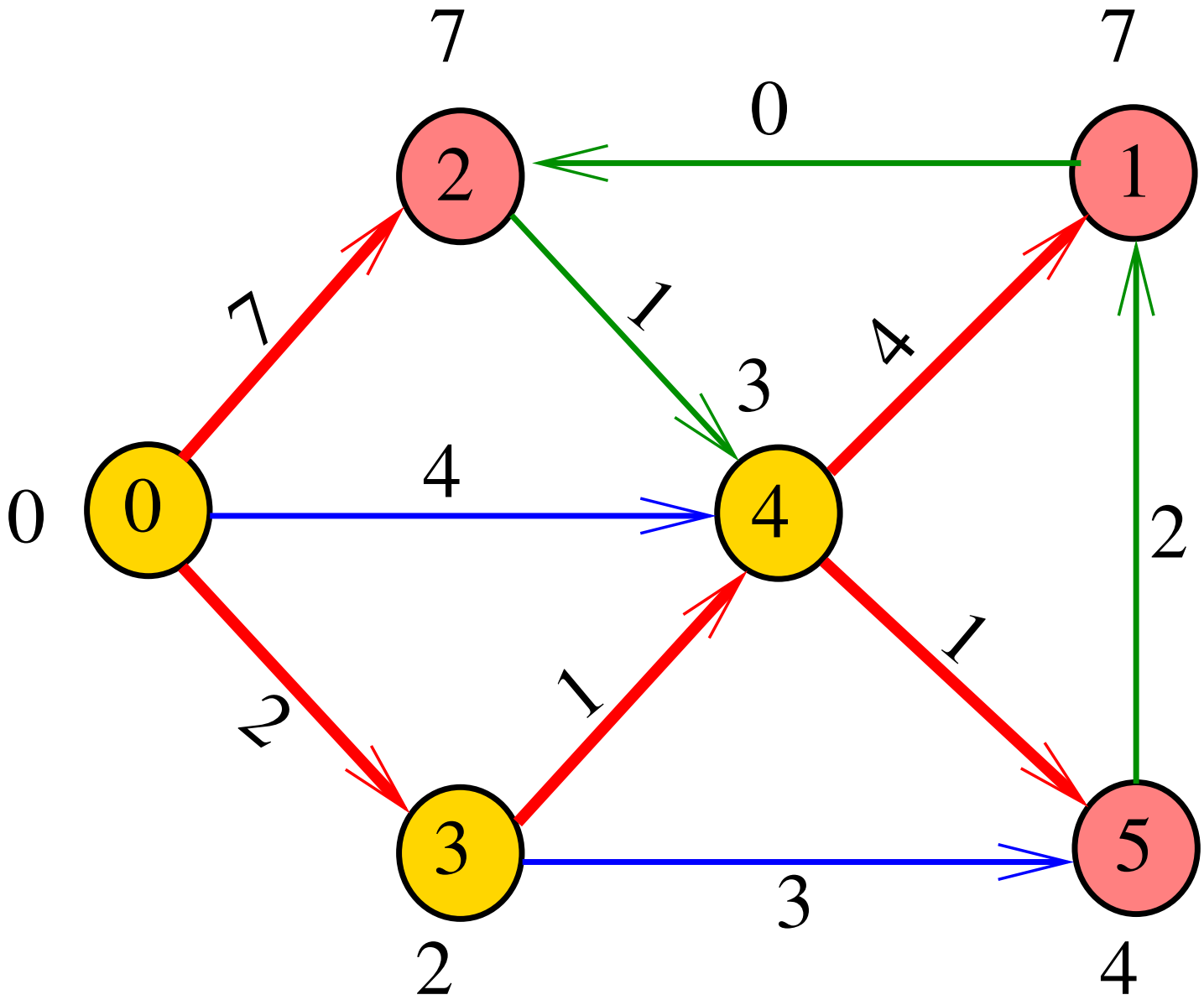
Simulação



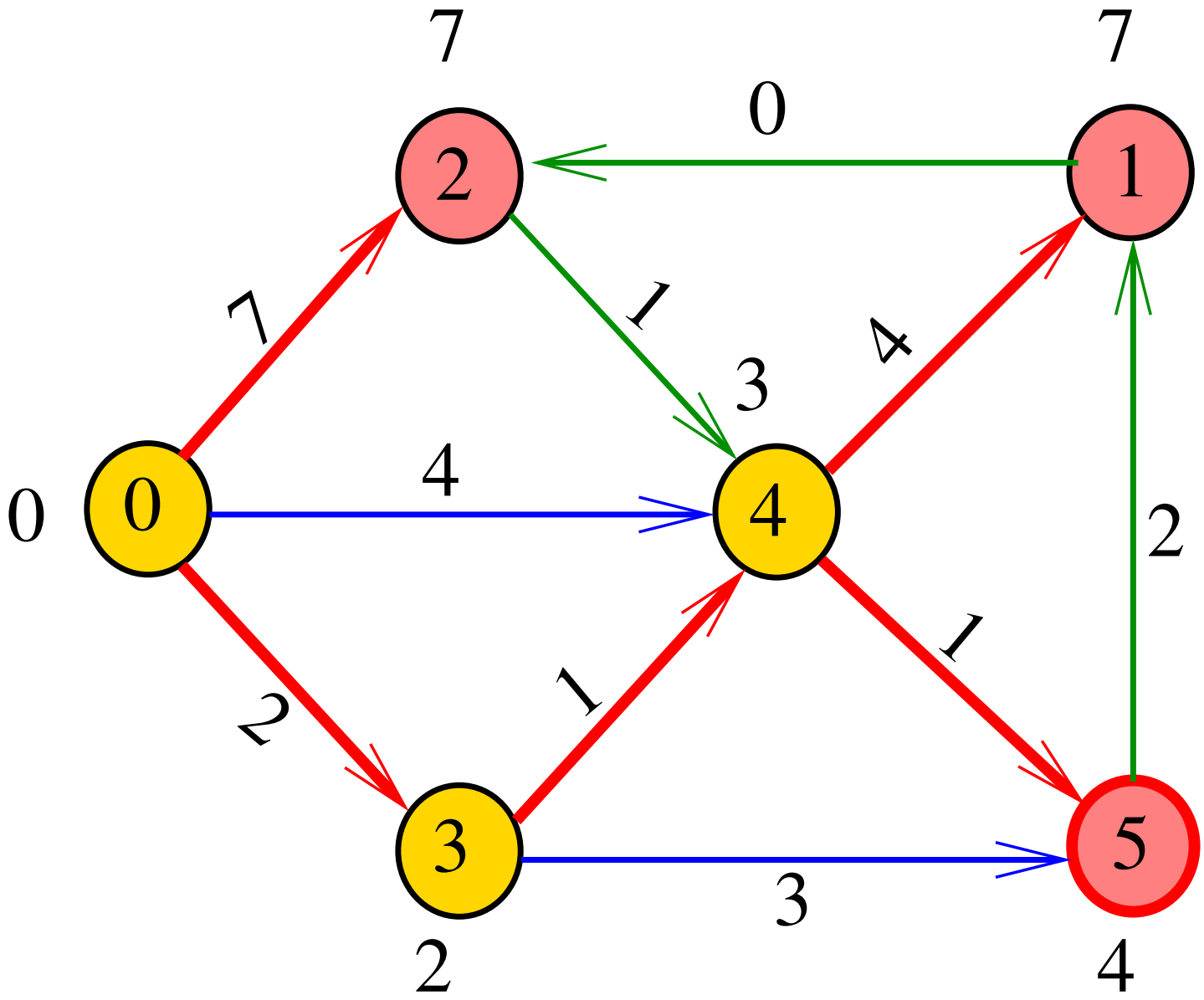
Simulação



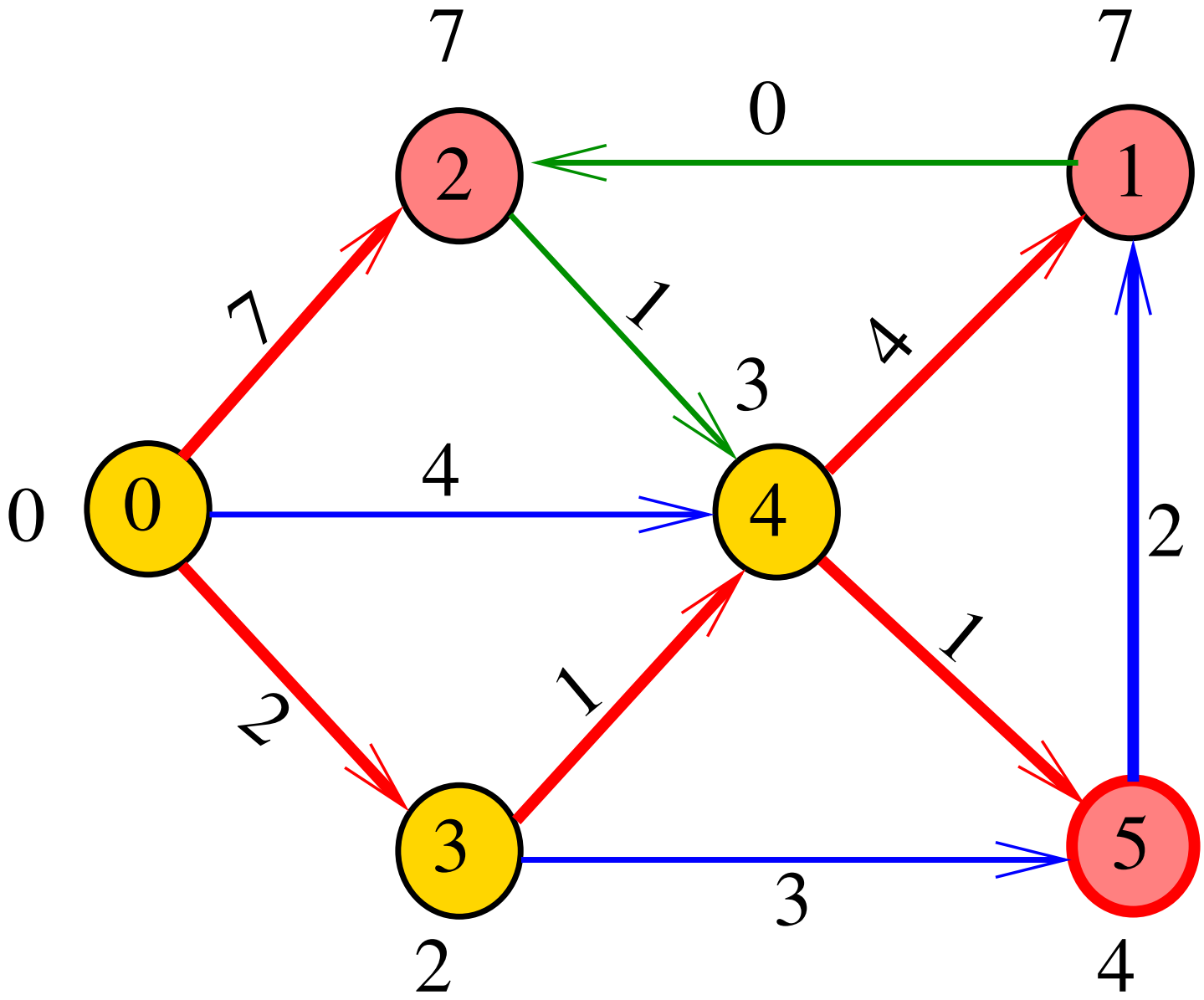
Simulação



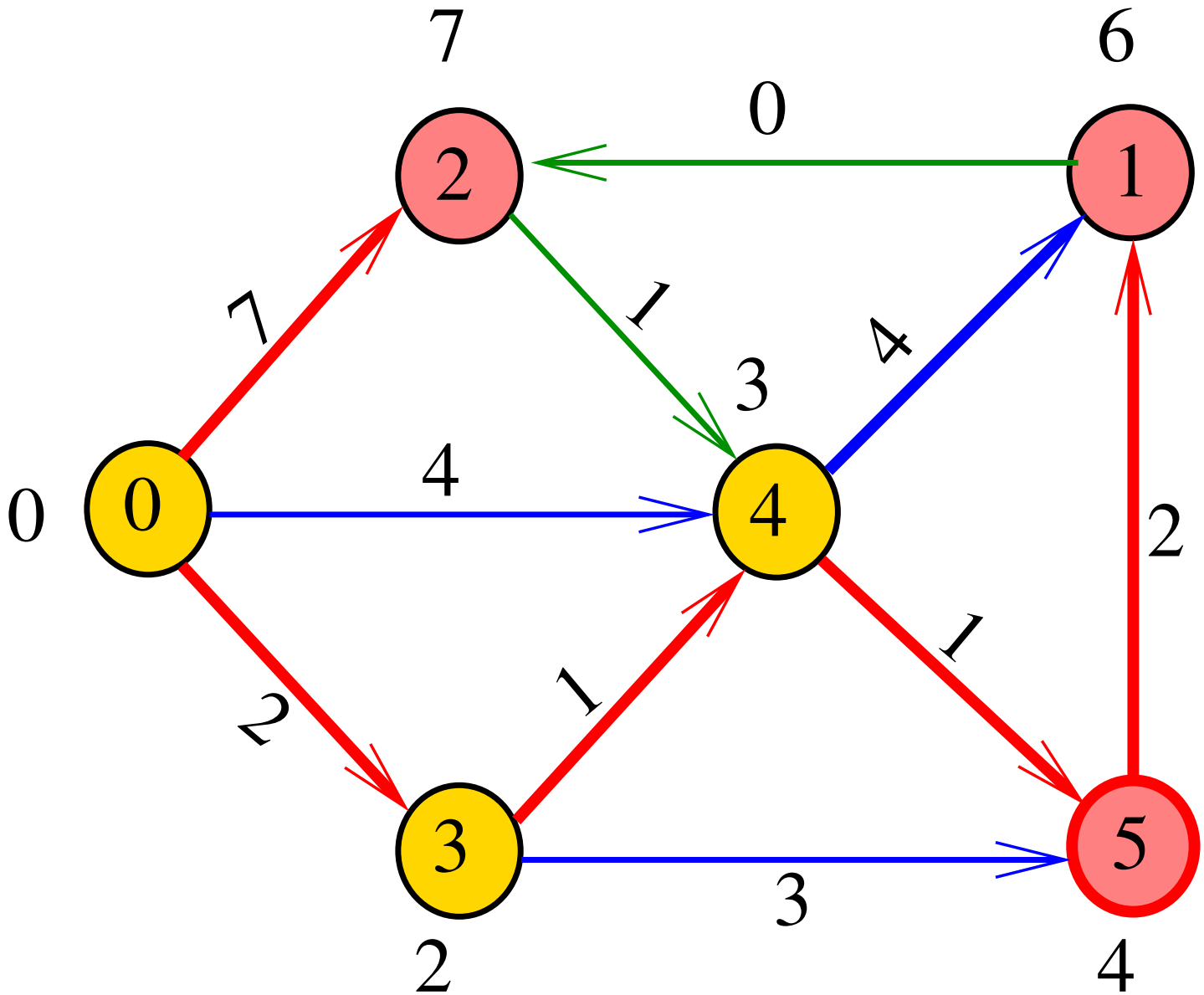
Simulação



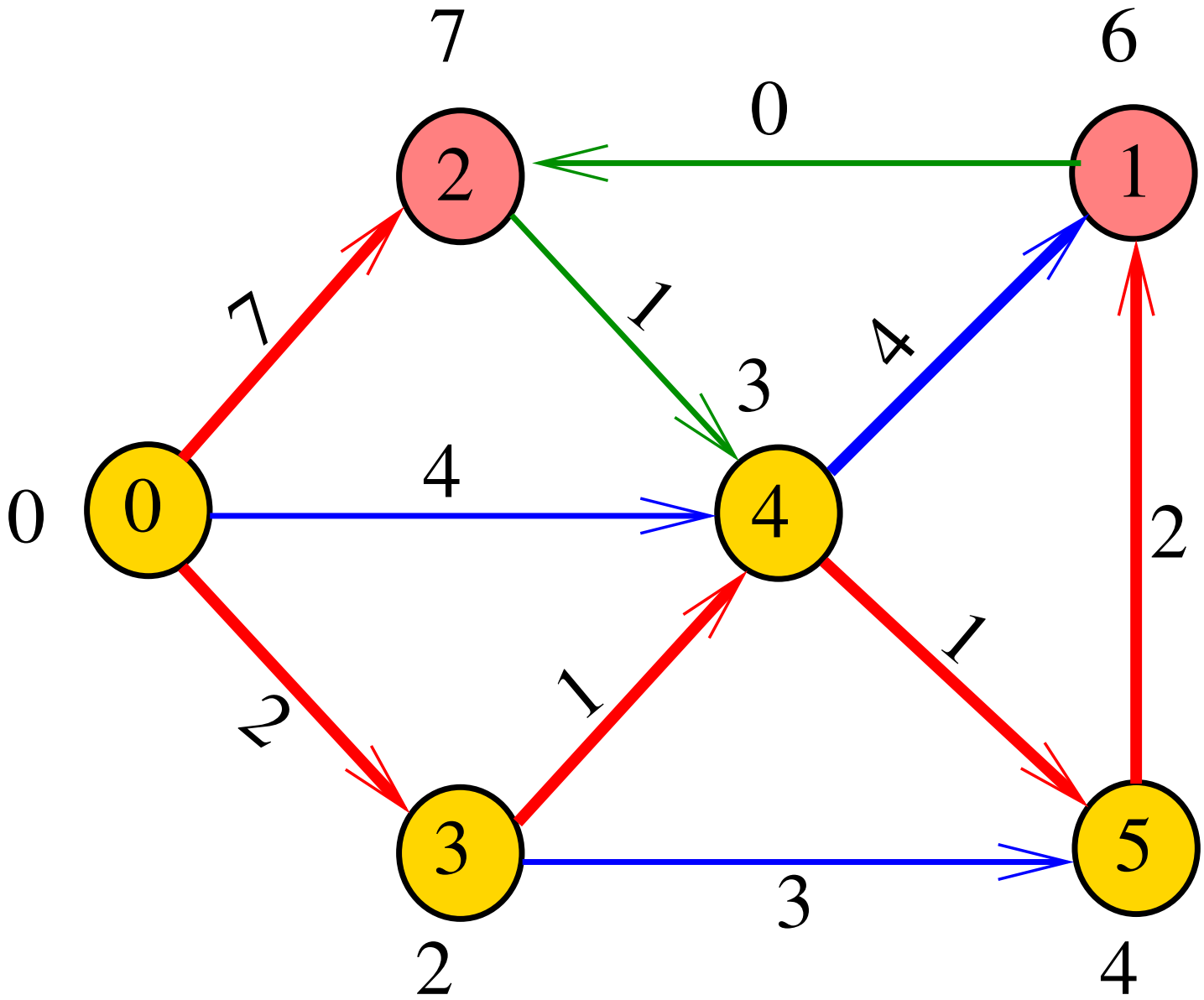
Simulação



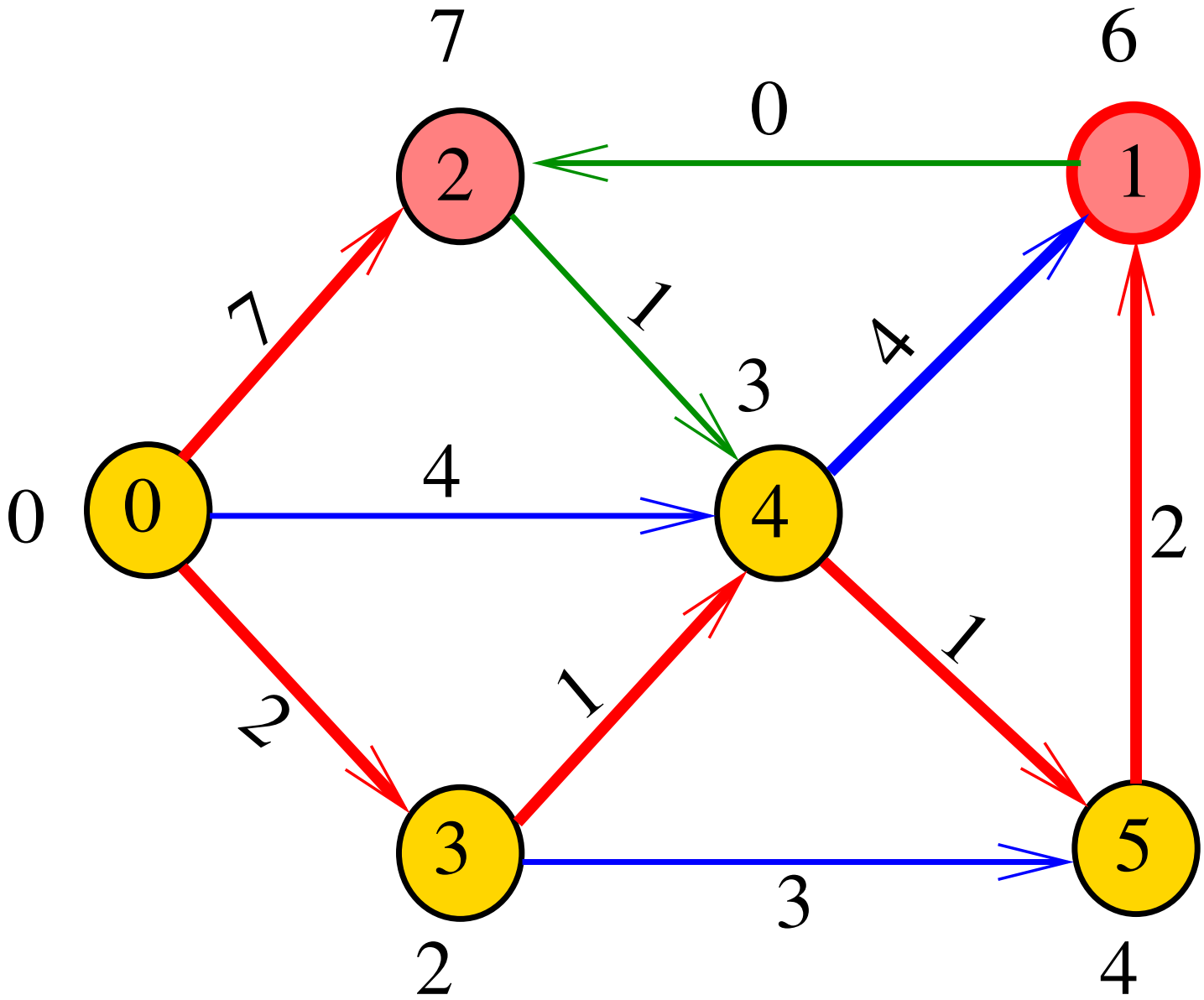
Simulação



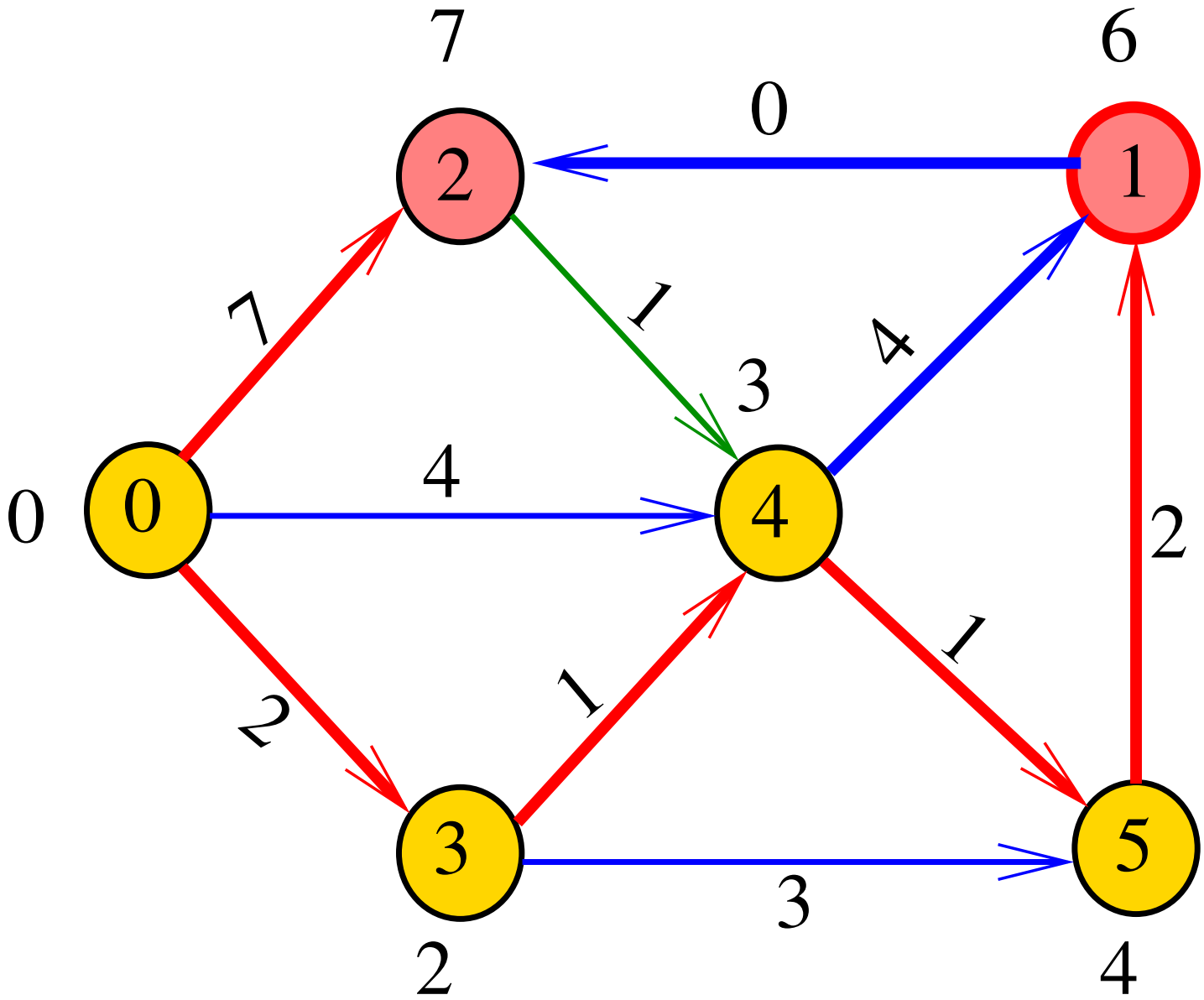
Simulação



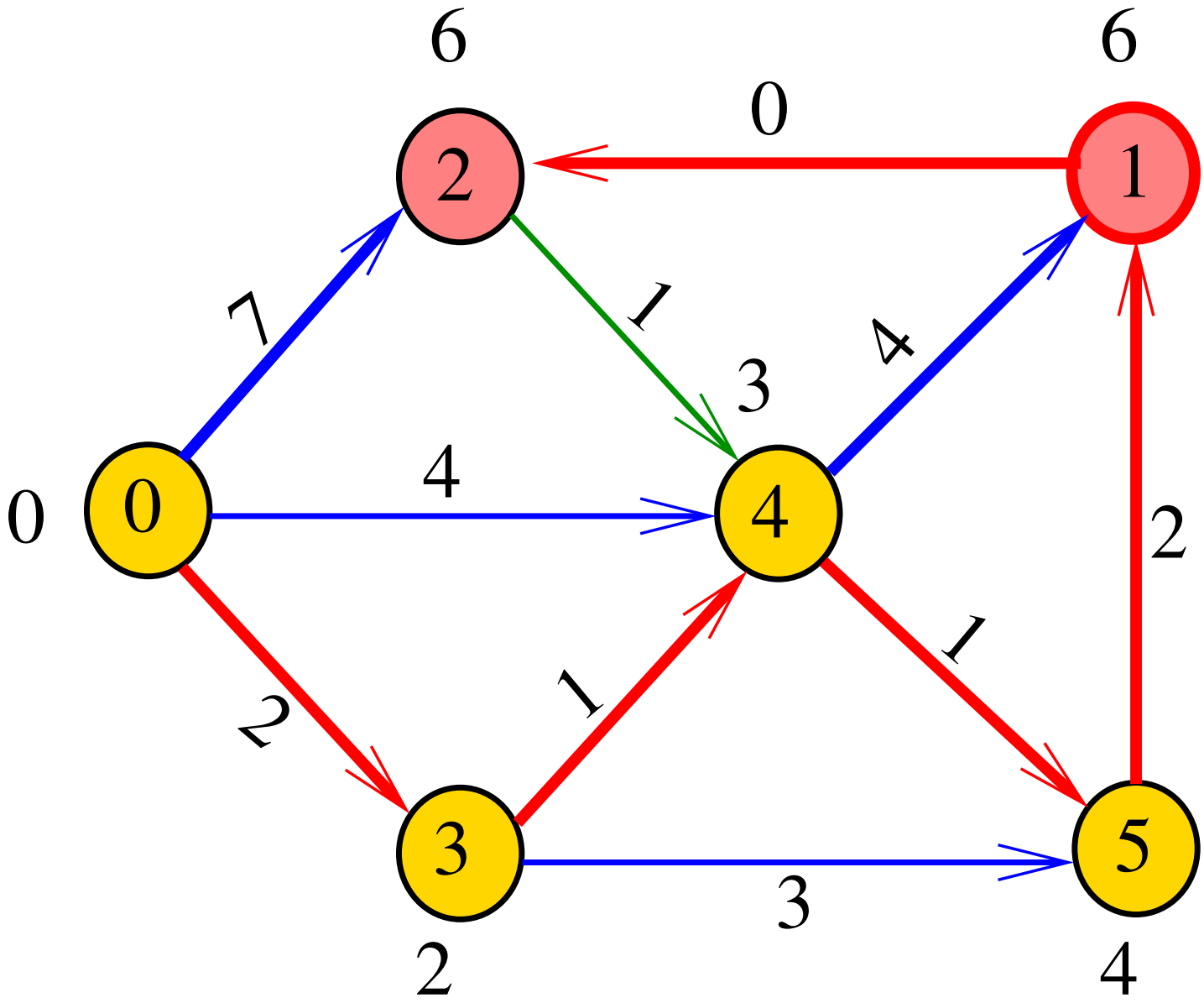
Simulação



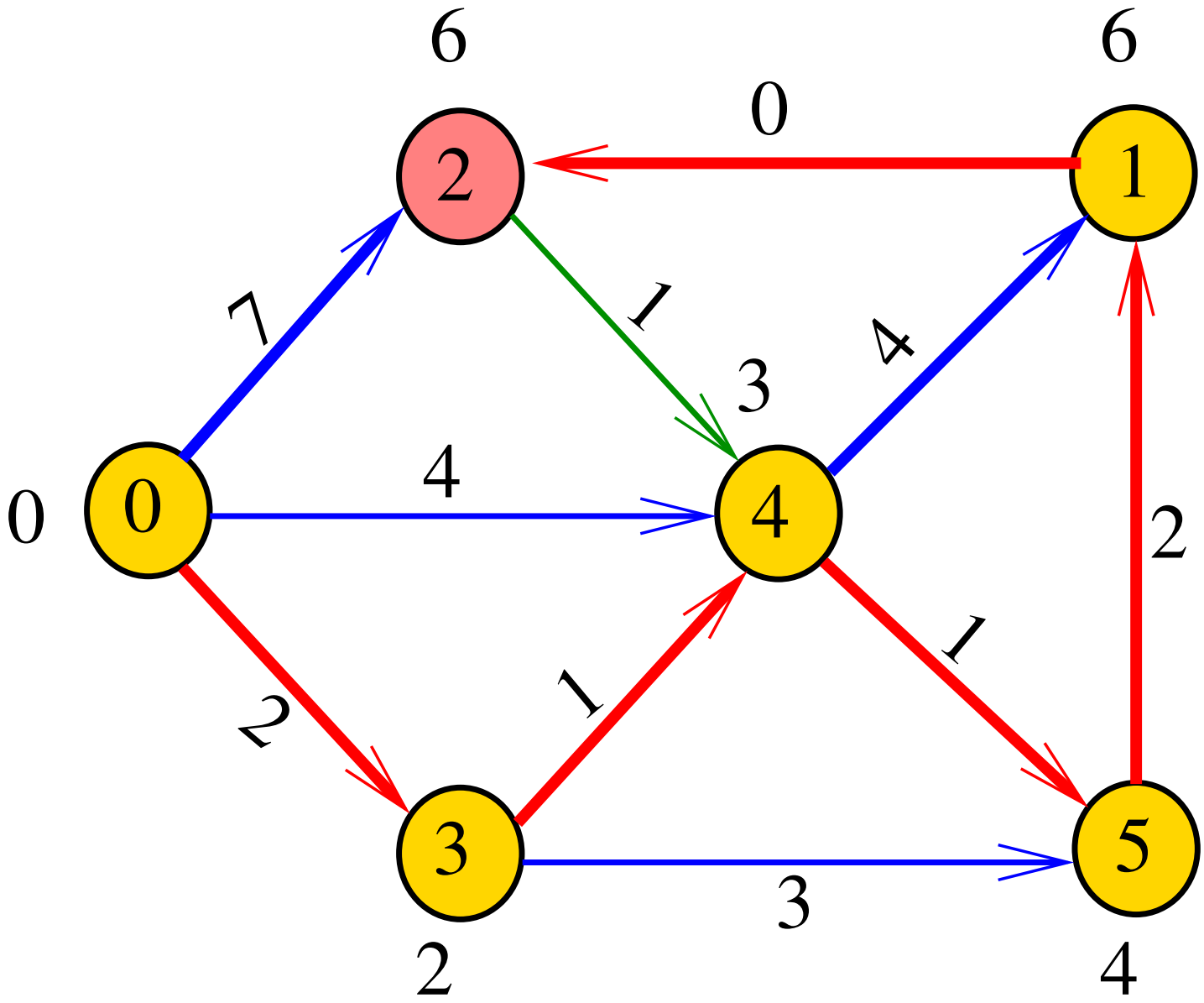
Simulação



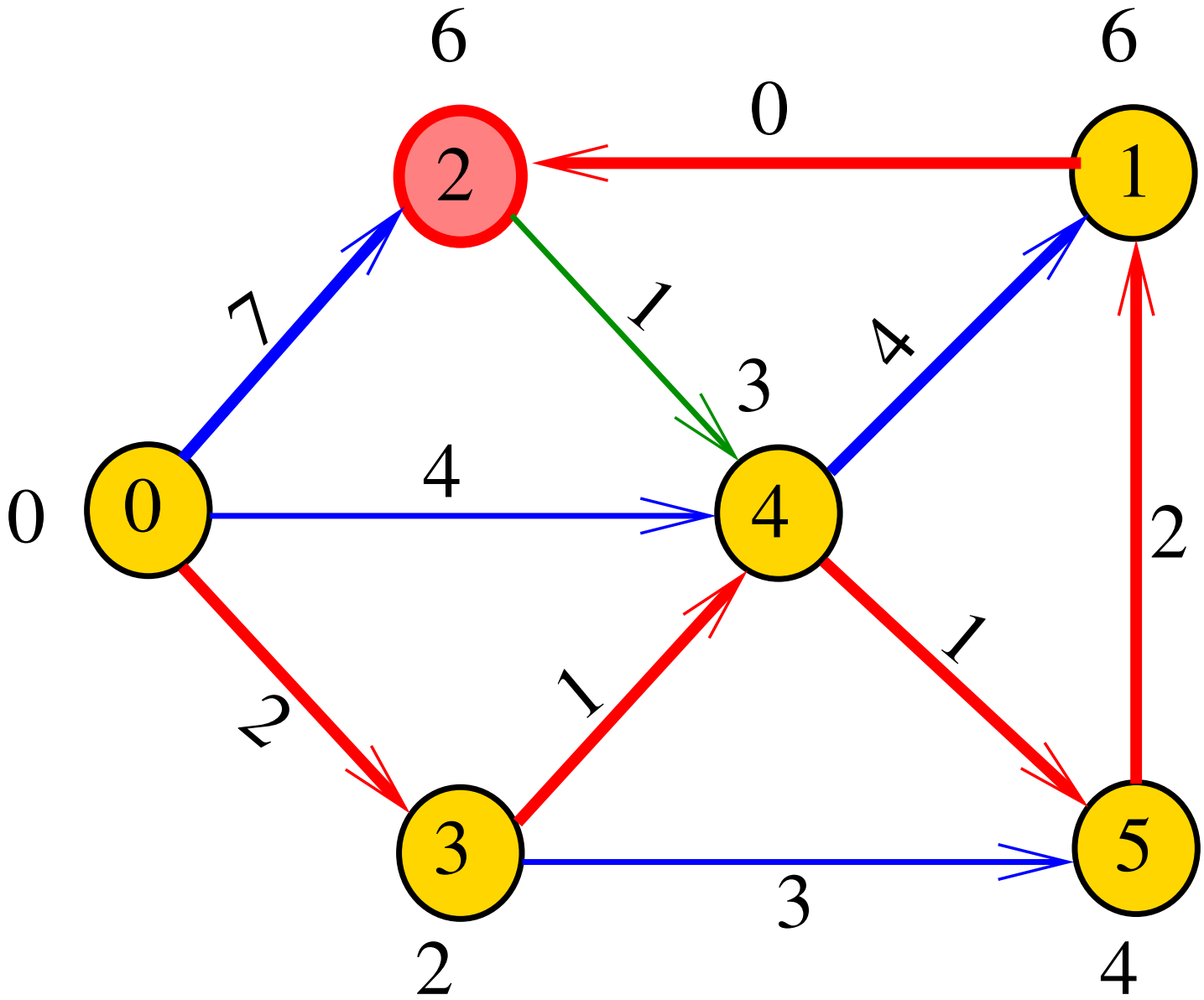
Simulação



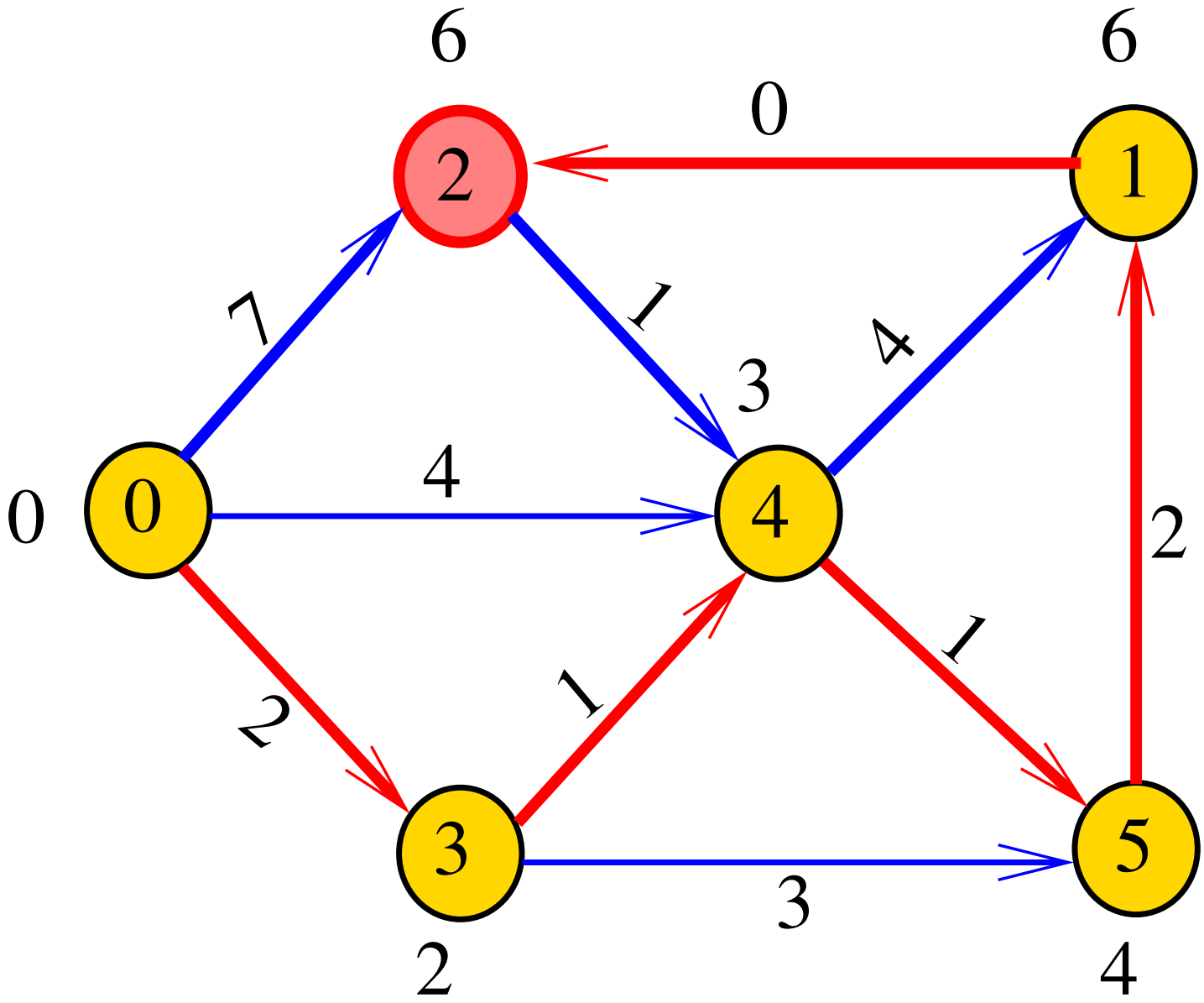
Simulação



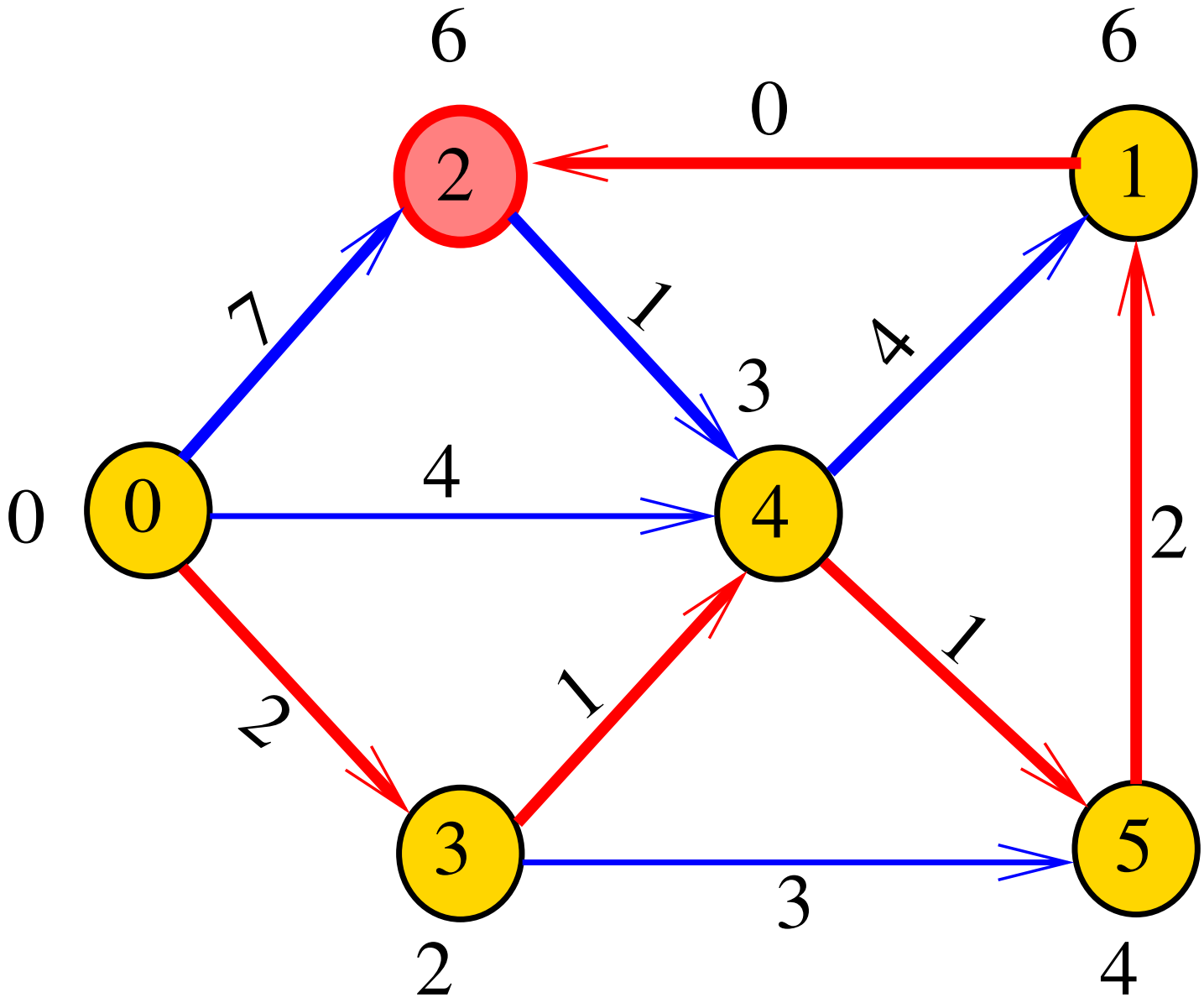
Simulação



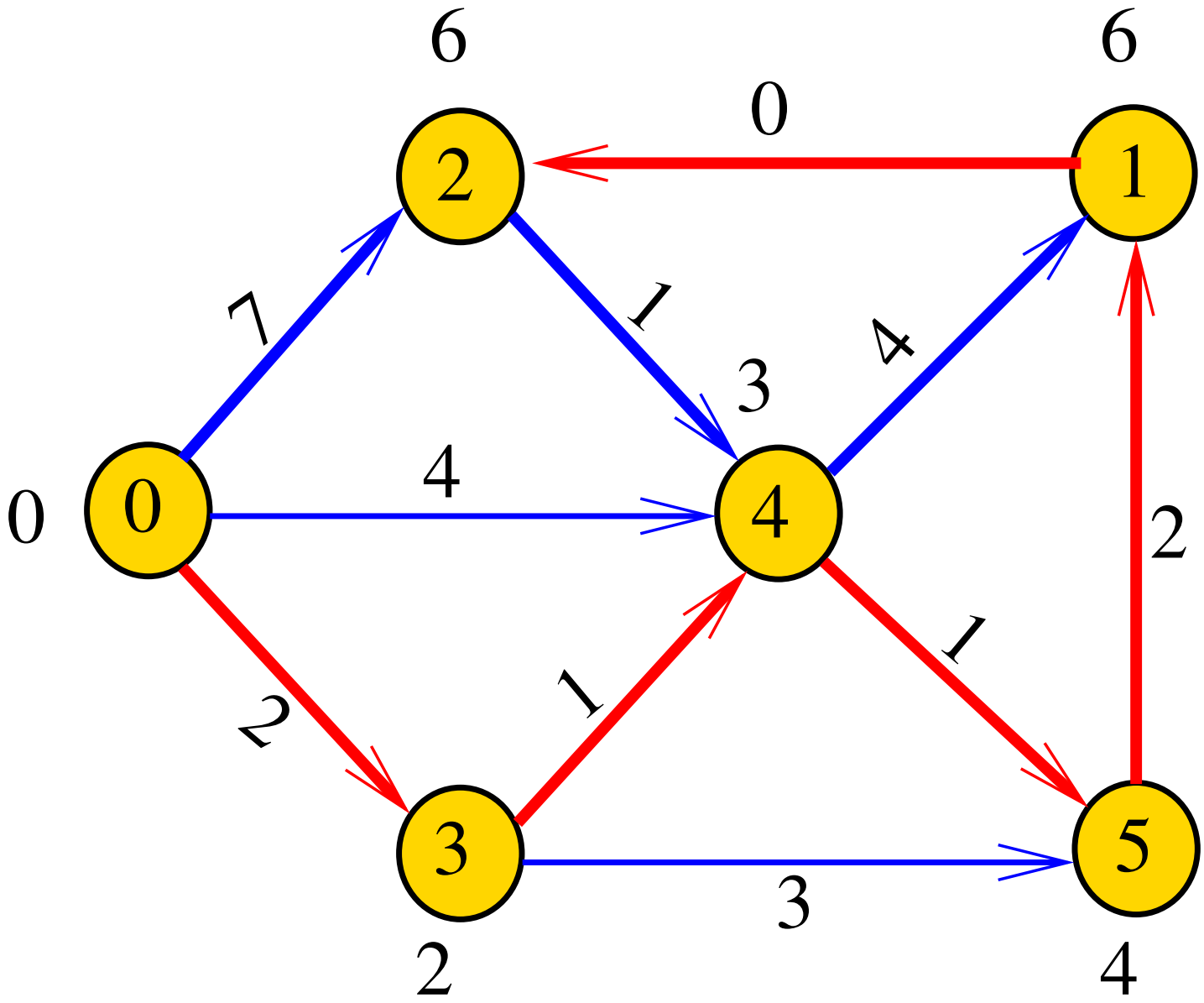
Simulação



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 `edgeTo[]`.

As distâncias em relação a `s` são armazenadas no vetor `distTo[]`.

```
double *distTo;
int    *edgeTo;
```

Fila priorizadas

A rotina `Dijkstra` usa uma fila priorizada

```
IndexMinPQ pq;
```

A fila é manipulada pelos métodos:

- `IndexMinPQInit()`: cria fila priorizada de vértices; cada vértice `v` terá prioridade `distTo[v]`.
- `isEmpty(pq)`: a fila `pq` está vazia?
- `contains(pq, v)`: `v` está na fila `pq`?
- `insert(pq, v, valor)`: insere `v` com prioridade `valor`.
- `delMin(pq)`: retorna um vértice de prioridade mínima.
- `decreaseKey(pq, w, valor)`: reorganiza a fila `pq` após decrementar o valor de `distTo[w]`.

Dijkstra: estrutura

```
static struct dijkstra {
   double INFINITY;
   int s;
   double *distTo;
   int    *edgeTo;
};
typedef struct dijkstra *DijkSt;
```

DFSpaths: Init

```
static DijkSt DijkstraInit(EWDigraph G, int s) {
    DijkSt T = mallocSafe(sizeof(*T));
    T->INFINITY = DBL_MAX;      /* double máximo */
    T->s = s;
    T->distTo = mallocSafe(G->V*sizeof(double));
    T->edgeTo = mallocSafe(G->V*sizeof(int));
    for (int v = 0; v < G->V; v++) {
        T->distTo[v] = T->INFINITY;
        T->edgeTo[v] = -1;
    }
    return T;
}
```

EWDigraph

Células das listas de adjacências:

```
typedef struct celula *Link;

static struct celula {
   double weight;                    /* peso do arco */
   int end;                    /* ponta final do arco */
   Link next;
};

void addEdge(EWDigraph, int, int, double);
```

Dijkstra: esqueleto

```
DijkSt Dijkstra(EWDigraph G, int s) {...}

static void dijkstra(EWDigraph G, int s,
                     DijsSt T) {...}

/* Métodos copiados de BFSpaths.  */

bool   hasPath(DijsSt T, int v) {...}
double distTo(DijsSt T, int v) {...}
Stack  pathTo(DijsSt T, int v) {...}
```

Dijkstra

Encontra um caminho de s a todo vértice alcançável a partir de s.

```
DijkSt Dijkstra(EWDigraph G, int s) {
   DijkSt T = DijkstraInit(G, s);
   dijkstra(G, s, T);
   return T;
}
```

dijkstra(): inicializações

```
static
void dijkstra(EWDigraph G, int s, DijkSt T) {
   IndexMinPQ pq = IndexMinPQInit(G->V);
   Link a;   int v, w;   double d;
   T->distTo[s] = 0;
   insert(pq, s, T->distTo[s]);
   /* aqui vem a iteração do próximo slide */
```

dijkstra(): iteração

```
    while (!isEmpty(pq)) {
        v = delMin(pq);
        for (a = G->adj[v]; a != NULL; a = a->next) {
            w = a->end;
            d = T->distTo[v] + a->weight;
            if (d < T->distTo[w]) {
                T->edgeTo[w] = v;
                T->distTo[w] = d;
                if (contains(pq, w))
                    decreaseKey(pq, w, d);
                else insert(pq, w, d);
            }
        }
    }
}
```

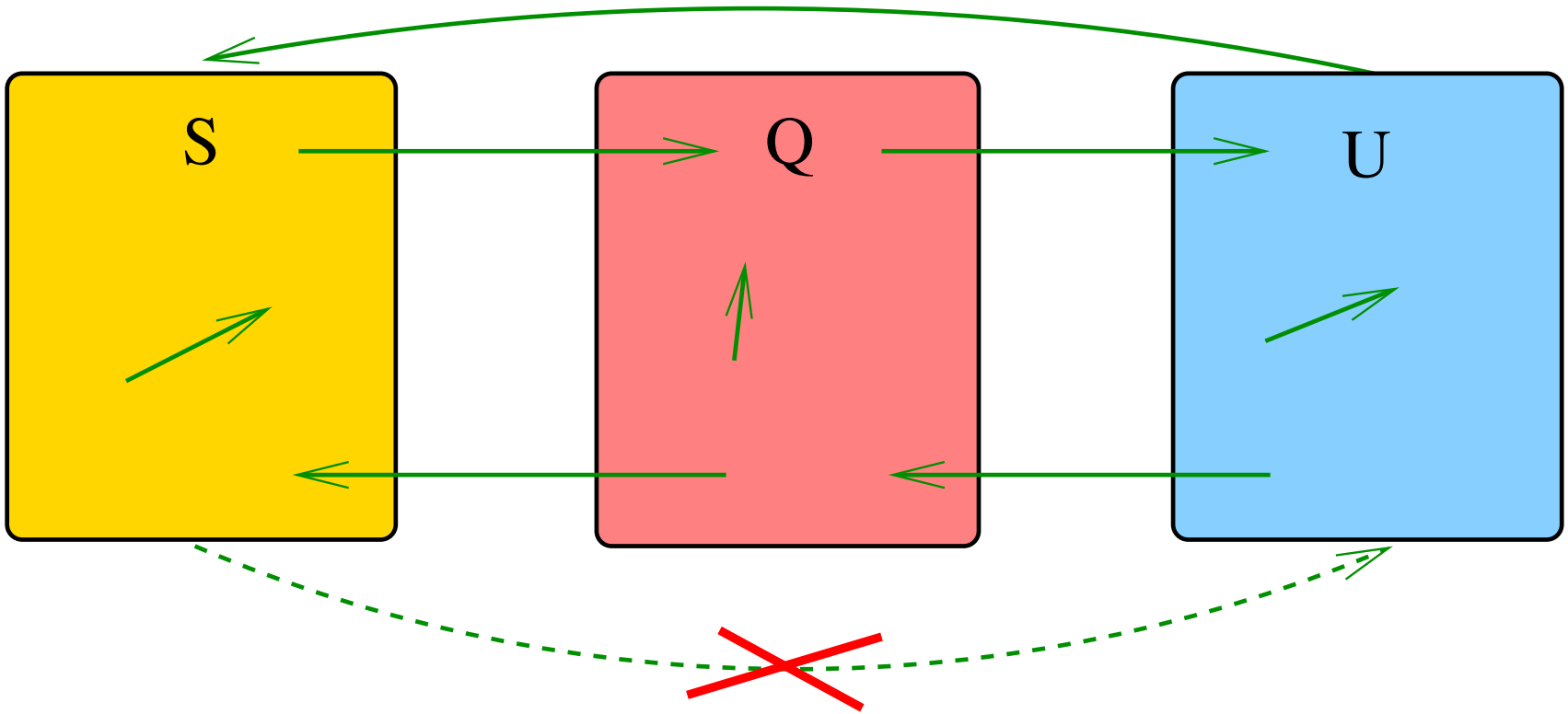
Relações invariantes

S = vértices examinados

Q = vértices visitados = vértices na fila

U = vértices ainda não visitados

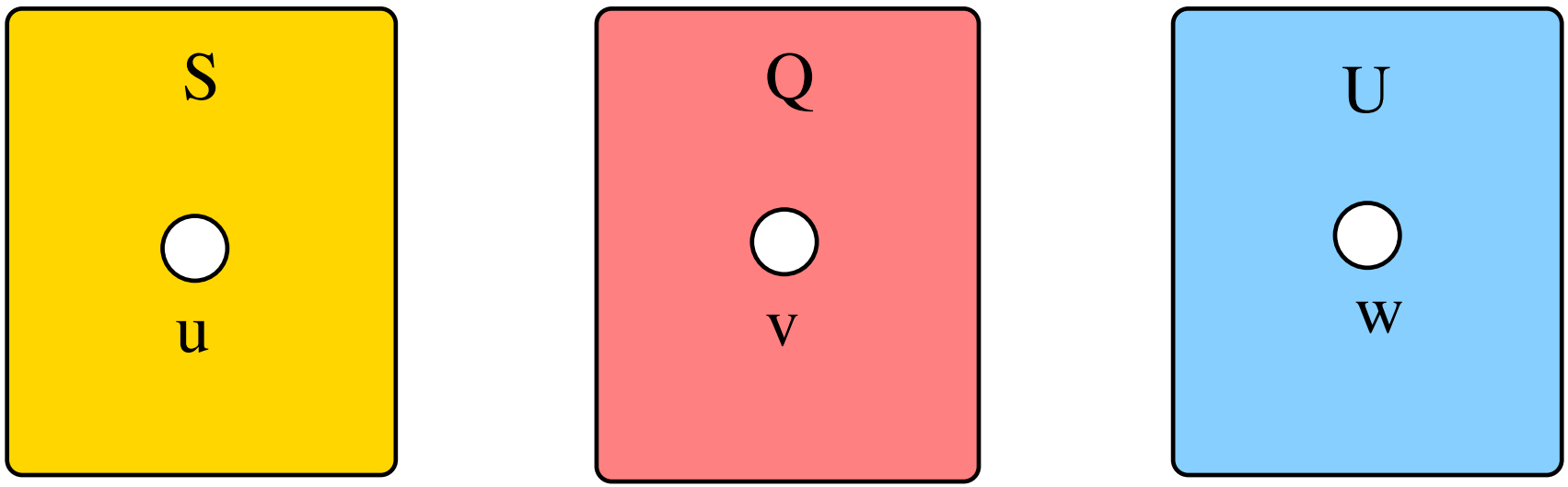
(i0) não existe arco v-w com v em S e w em U.



Relações invariantes

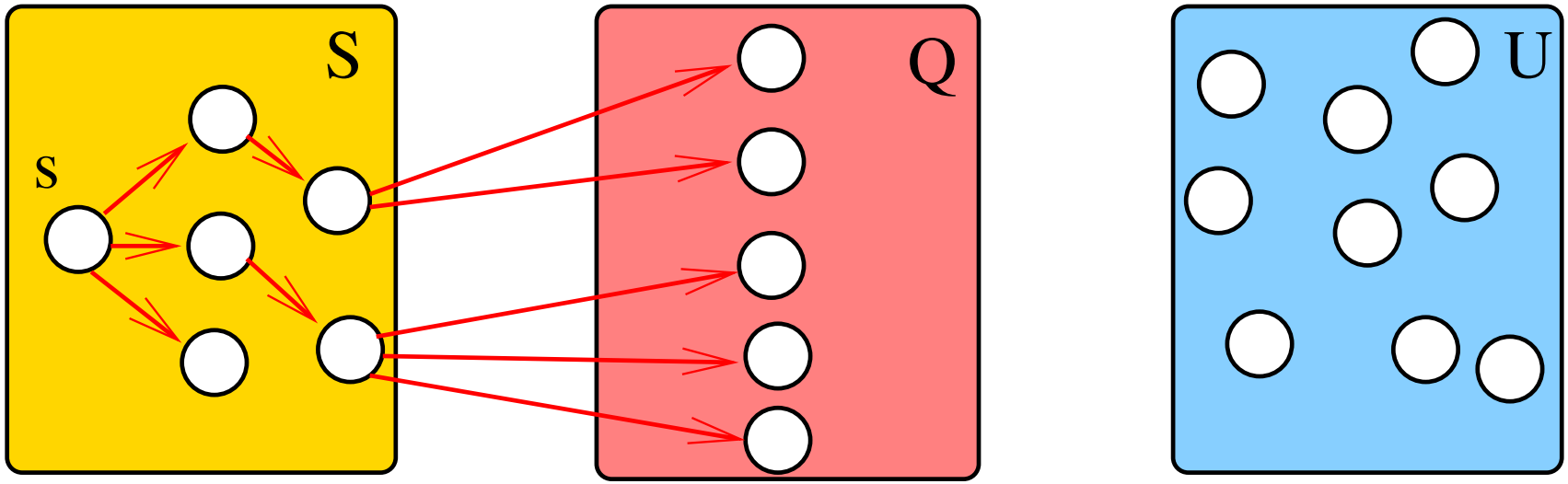
(i1) para cada u em S, v em Q e w em U

$$\texttt{distTo}[\texttt{u}] \leq \texttt{distTo}[\texttt{v}] \leq \texttt{distTo}[\texttt{w}].$$



Relações invariantes

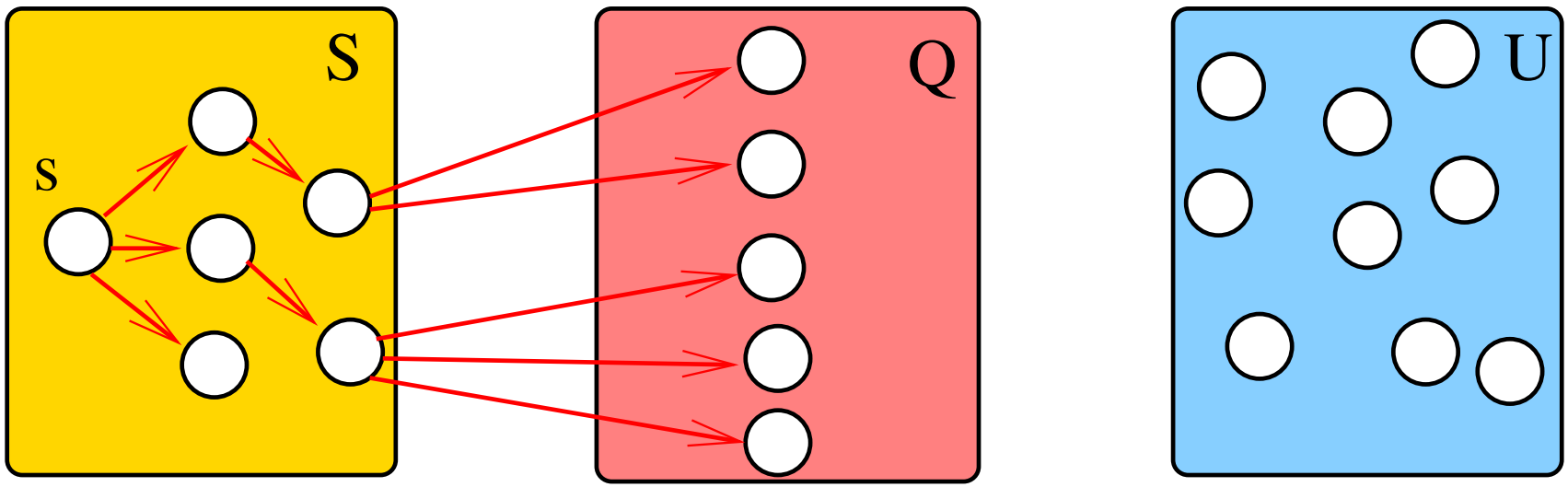
(i2) O vetor `edgeTo` restrito aos vértices de `S` e `Q` determina uma árborescência com raiz `s`.



Relações invariantes

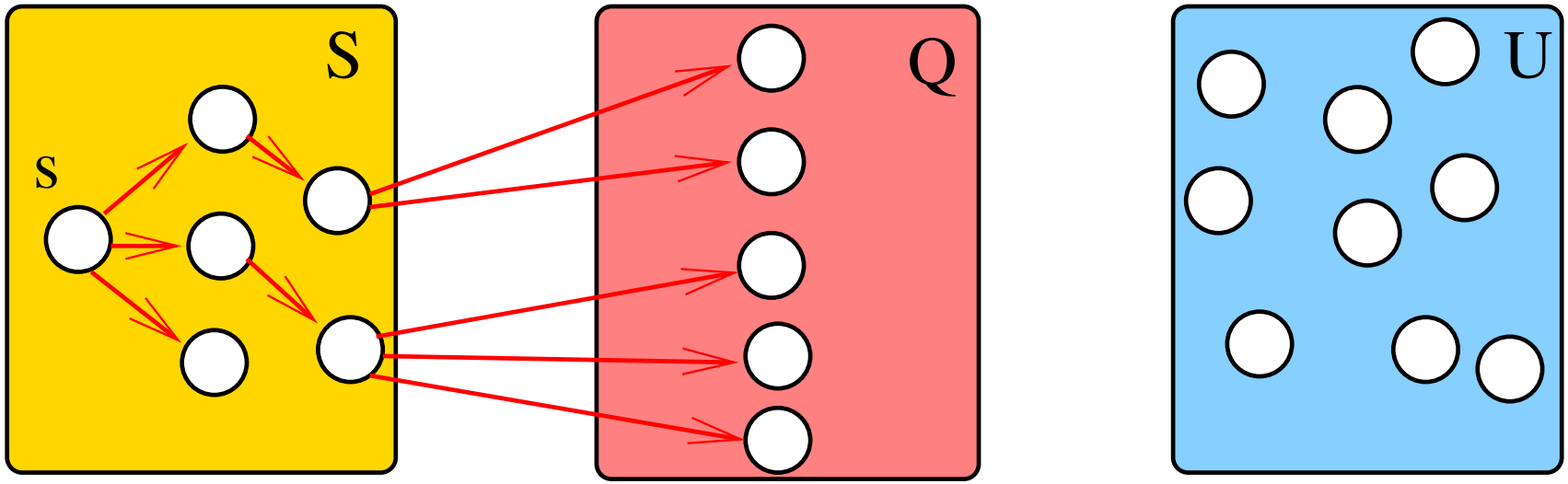
(i3) Para arco `v`-`w` na arborescência vale que

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

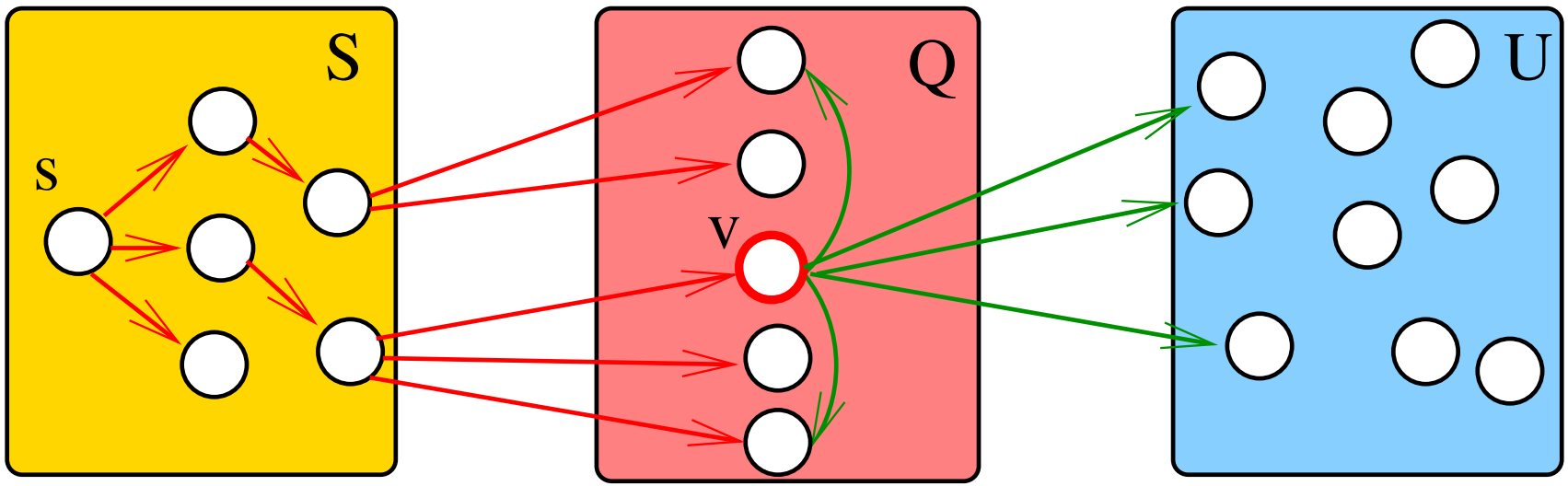


Relações invariantes

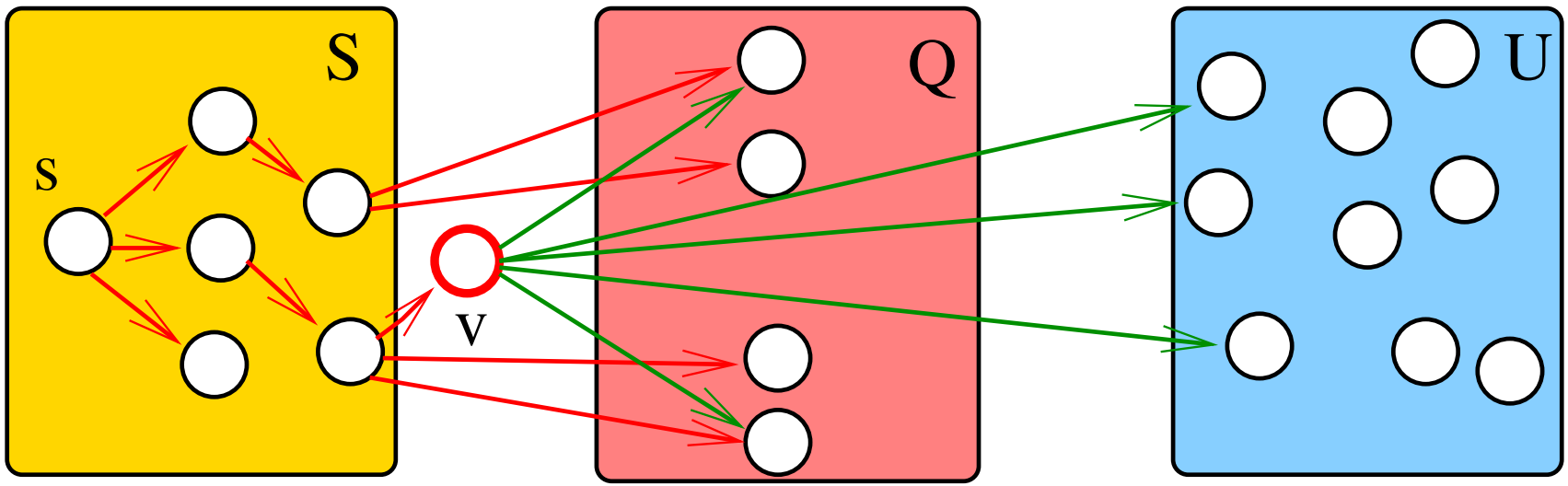
(i3) Para cada vértice `v` em `S` vale que `distTo[v]` é o custo de um caminho mínimo de `s` a `v`.



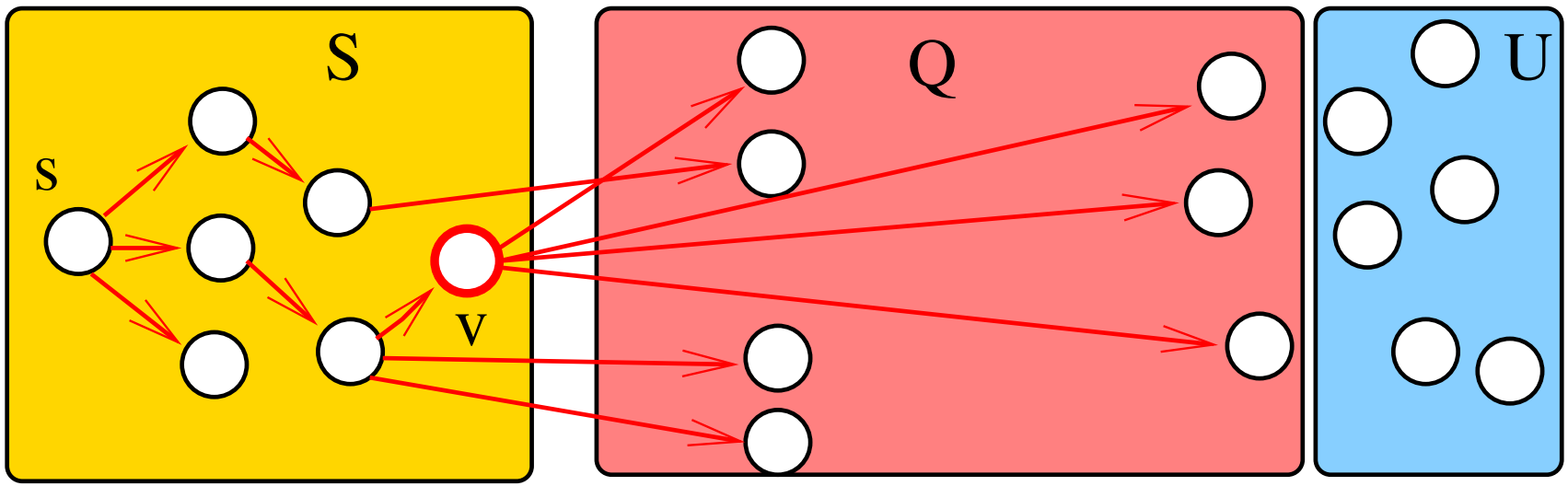
Iteração



Iteração



Iteração



Consumo de tempo

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

$= 1$ execução de `IndexMinPQInit`,
$\leq V$ execuções de `insert`,
$\leq V$ execuções de `isEmpty`,
$\leq V$ execuções de `delMin`, e
$\leq E$ execuções de `contains`,
$\leq E$ execuções de `decreaseKey`.

Consumo de tempo Min-Heap

`IndexMinPQInit`	$\Theta(V)$
`isEmpty`	$\Theta(1)$
`insert`	$\Theta(\lg\ V)$
`delMin`	$\mathrm{O}(\lg\ V)$
`decreaseKey`	$\Theta(\lg\ V)$
`contains`	$\Theta(1)$

Conclusão

O consumo de `Dijkstra` é $\mathrm{O}(\mathtt{E} \lg \mathtt{V})$.

Para **grafos densos**, podemos alcançar consumo de tempo ótimo . . .

Detalhes em **MAC0328** Algoritmos em Grafos.

Consumo de tempo Min-Heap

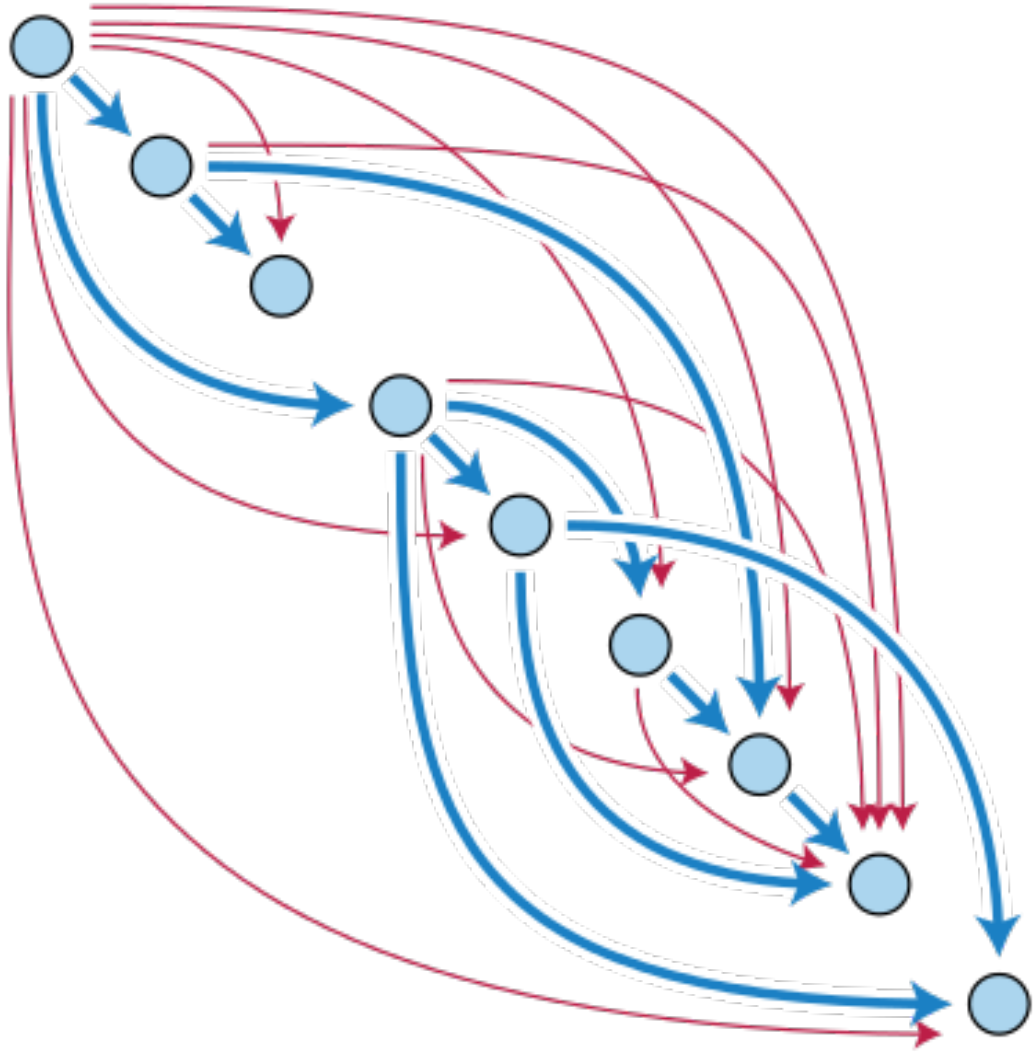
	heap	d-heap	fibonacci heap
insert	$O(\lg V)$	$O(\log_D V)$	$O(1)$
delMin	$O(\lg V)$	$O(\log_D V)$	$O(\lg V)$
decreaseKey	$O(\lg V)$	$O(\log_D V)$	$O(1)$
Dijkstra	$O(E \lg V)$	$O(E \log_D V)$	$O(E + V \lg V)$

Consumo de tempo Min-heap

	bucket heap	radix heap
insert	$O(1)$	$O(\lg(VC))$
delMin	$O(C)$	$O(\lg(VC))$
decreaseKey	$O(1)$	$O(1)$
Dijkstra	$O(E + VC)$	$O(E + V \lg(VC))$

$C =$ maior custo de um arco.

Caminhos mínimos em DAGs

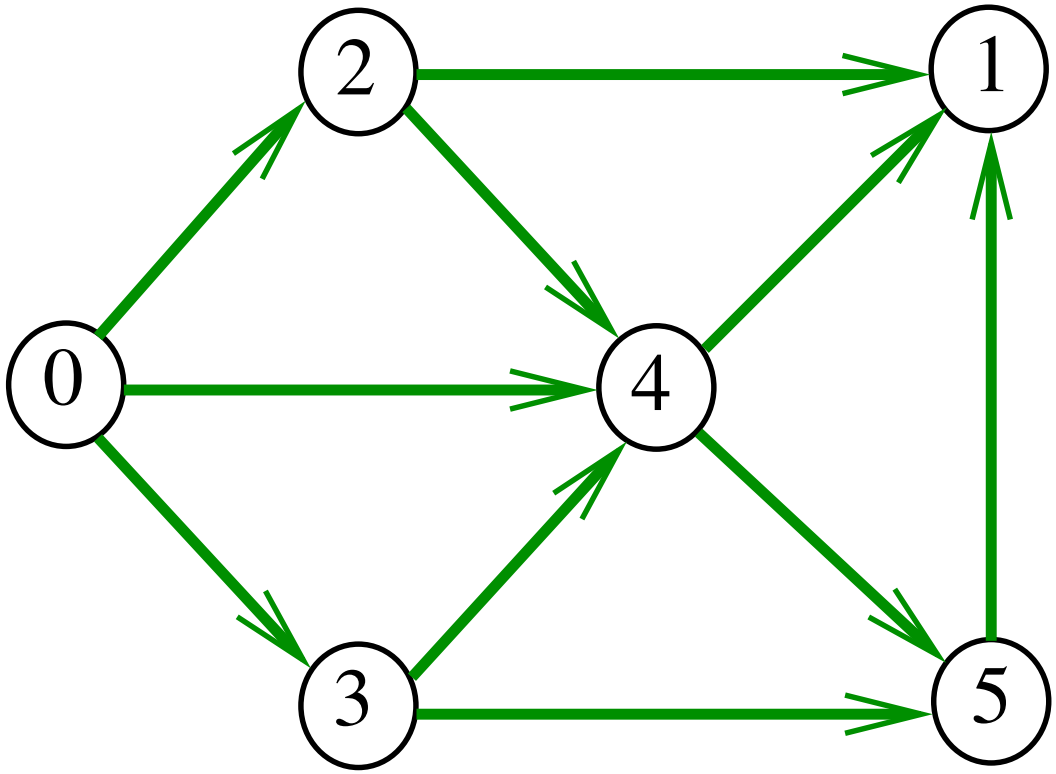


Fonte: Directed acyclic graph

DAGs

Um digrafo é **acíclico** se não tem ciclos.
Digrafos acíclicos também são conhecidos como DAGs (= *directed acyclic graphs*).

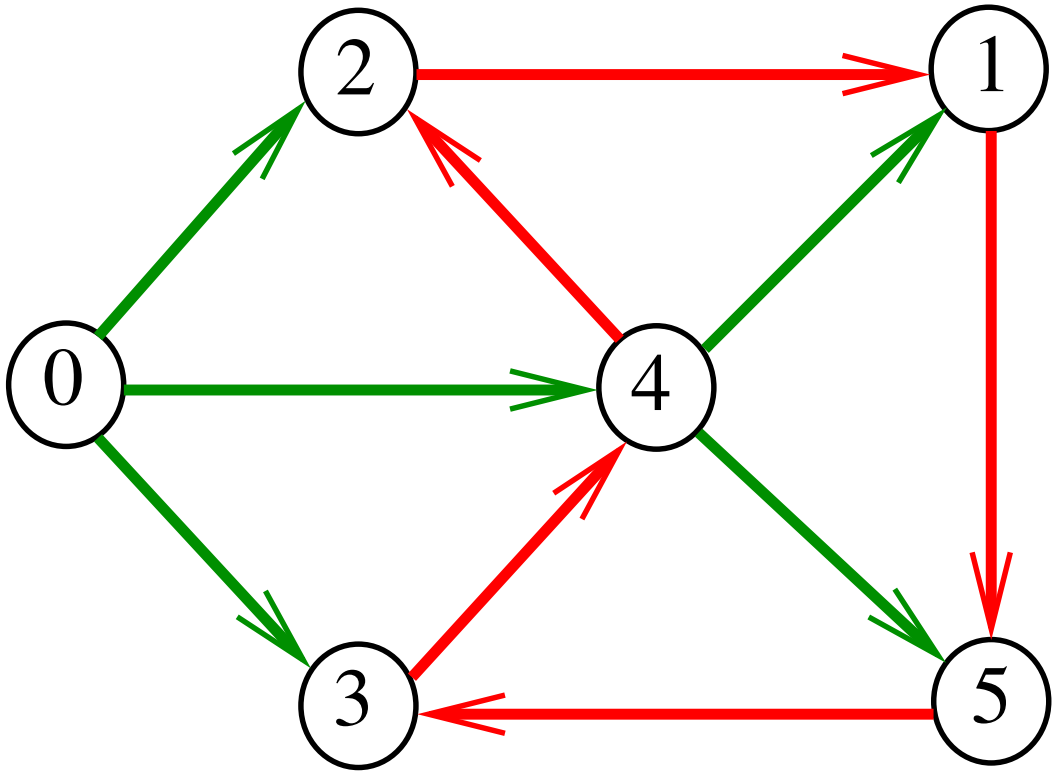
Exemplo: um digrafo acíclico



DAGs

Um digrafo é **acíclico** se não tem ciclos.
Digrafos acíclicos também são conhecidos como DAGs (= *directed acyclic graphs*).

Exemplo: um digrafo que **não** é acíclico



Ordenaçao topológica

Uma **ordenação topológica** (= *topological sorting*) de um digrafo é uma permutação

`ts[0], ts[1], ..., ts[V-1]`

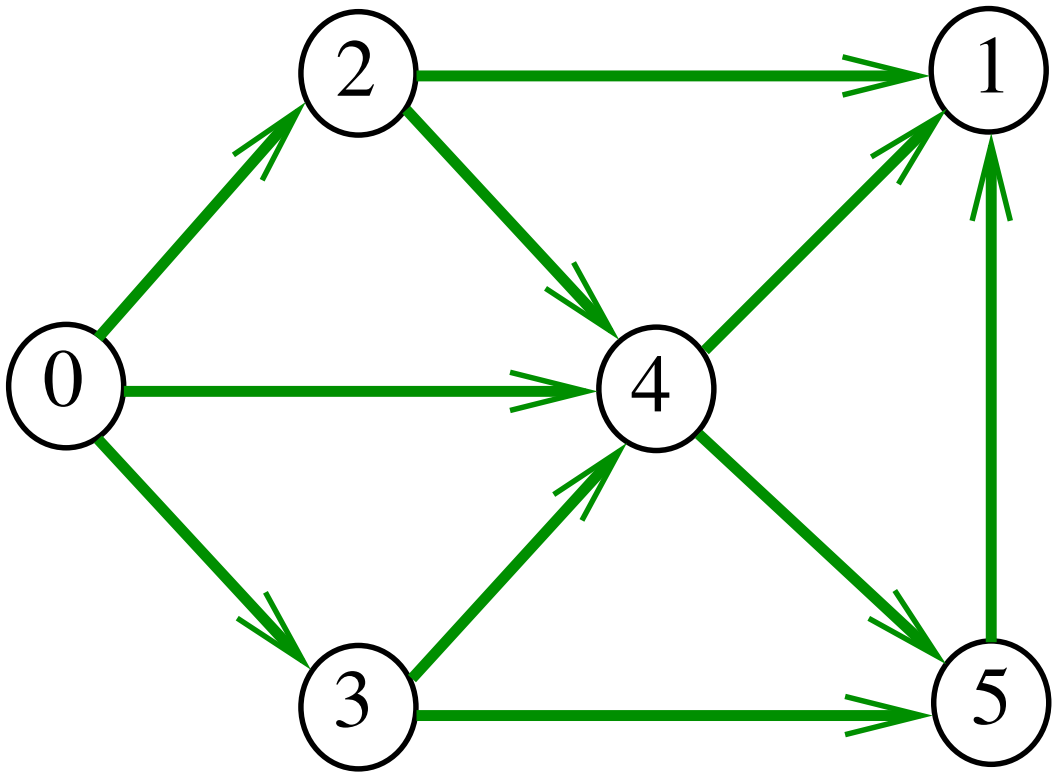
dos seus vértices tal que todo arco tem a forma

`ts[i]-ts[j]` com `i < j`.

`ts[0]` é necessariamente uma **fonte**
`ts[V-1]` é necessariamente um **sorvedouro**

Exemplo

i	0	1	2	3	4	5
ts[i]	0	3	2	4	5	1



Fato

Para todo digrafo G, vale uma e apenas umas das seguintes afirmações:

- G possui um ciclo;
- G é um DAG e, portanto, admite uma ordenação topológica.

Fonte: Well-Known Powerful Yin Yang Symbol Dates Back To Ancient China

Problema

Problema:

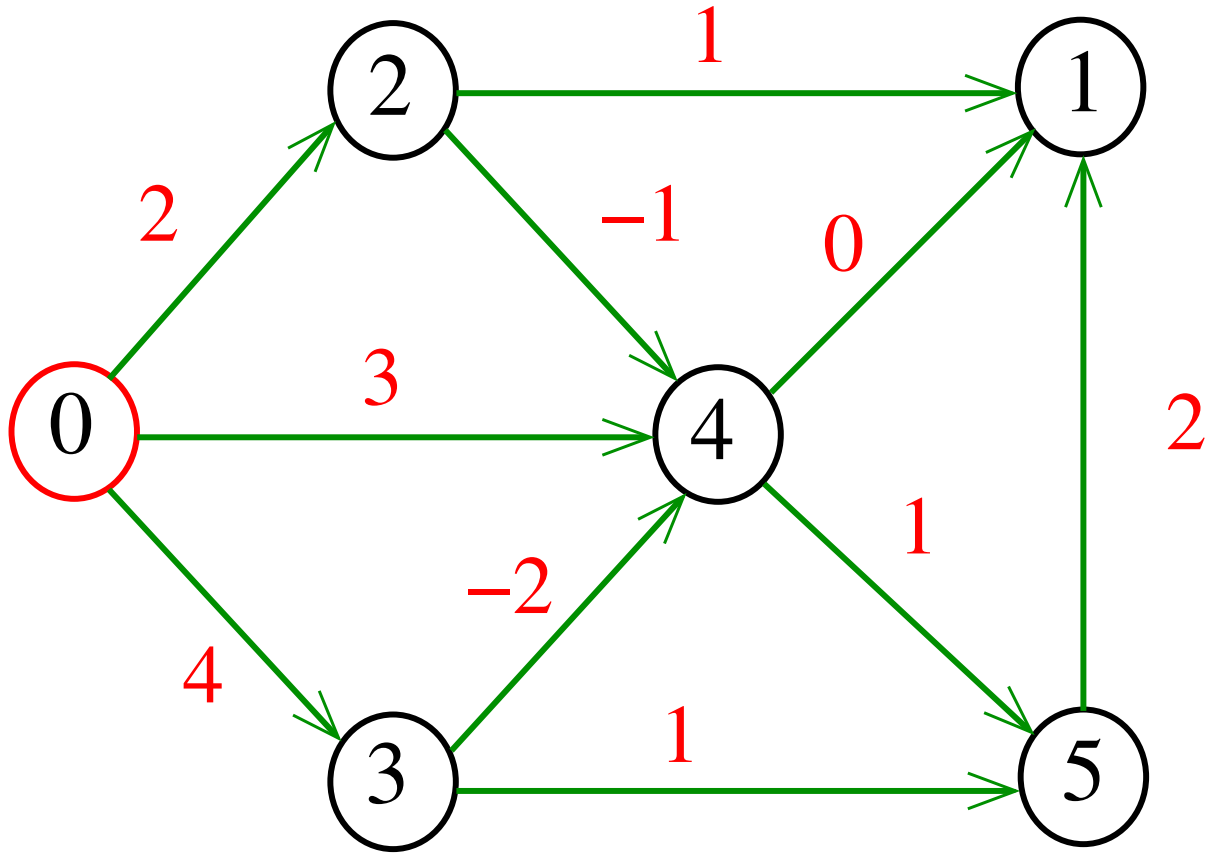
Dado um vértice s de um DAG com custos ***possivelmente negativos*** *nos arcos, encontrar, para cada vértice t que pode ser alcançado a partir de s, um caminho simples mínimo de s a t.*

Problema:

Dado um vértice s de um DAG com custos ***possivelmente negativos*** *nos arcos, encontrar uma SPT com raiz s.*

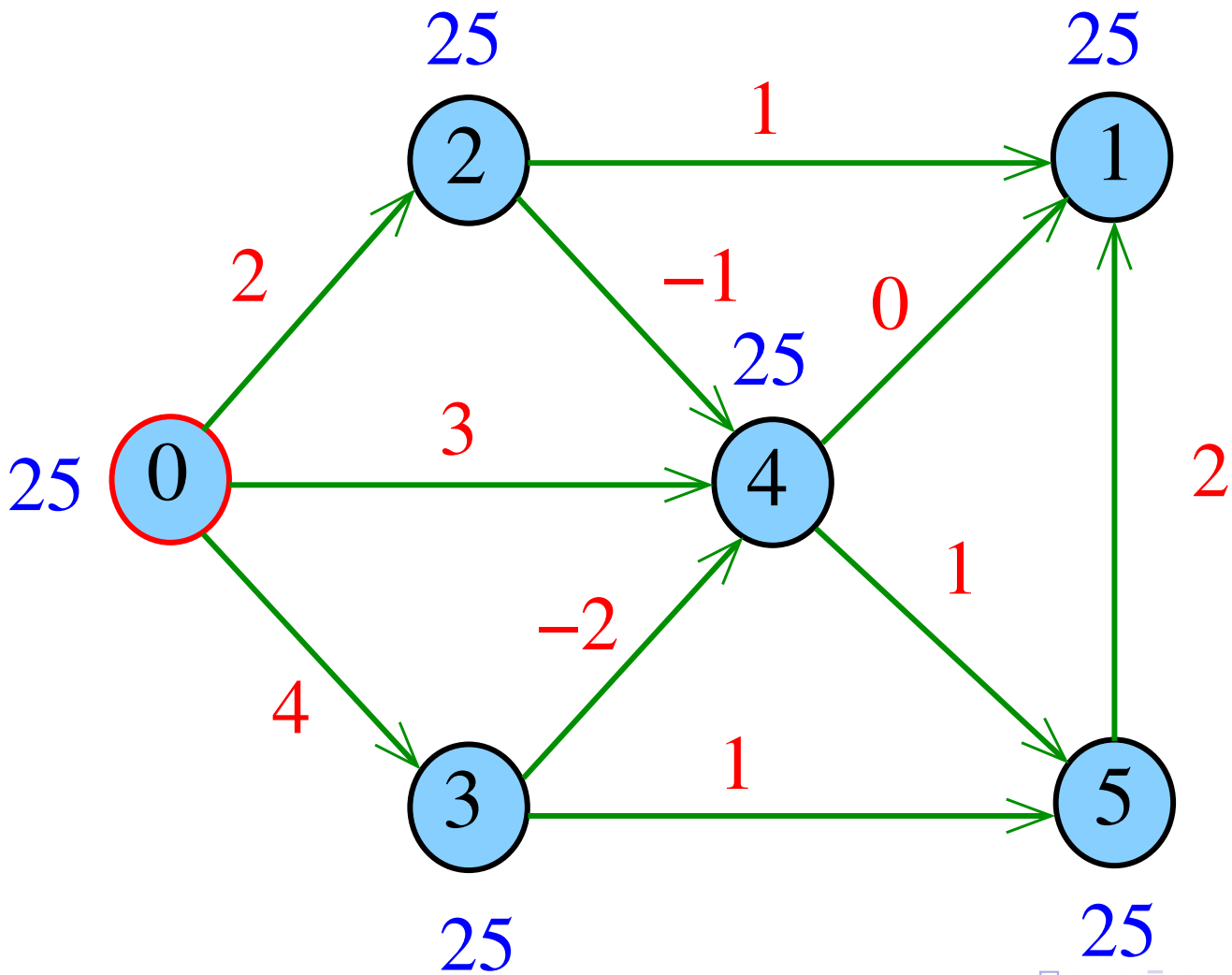
Simulação

i	0	1	2	3	4	5
ts[i]	0	3	2	4	5	1



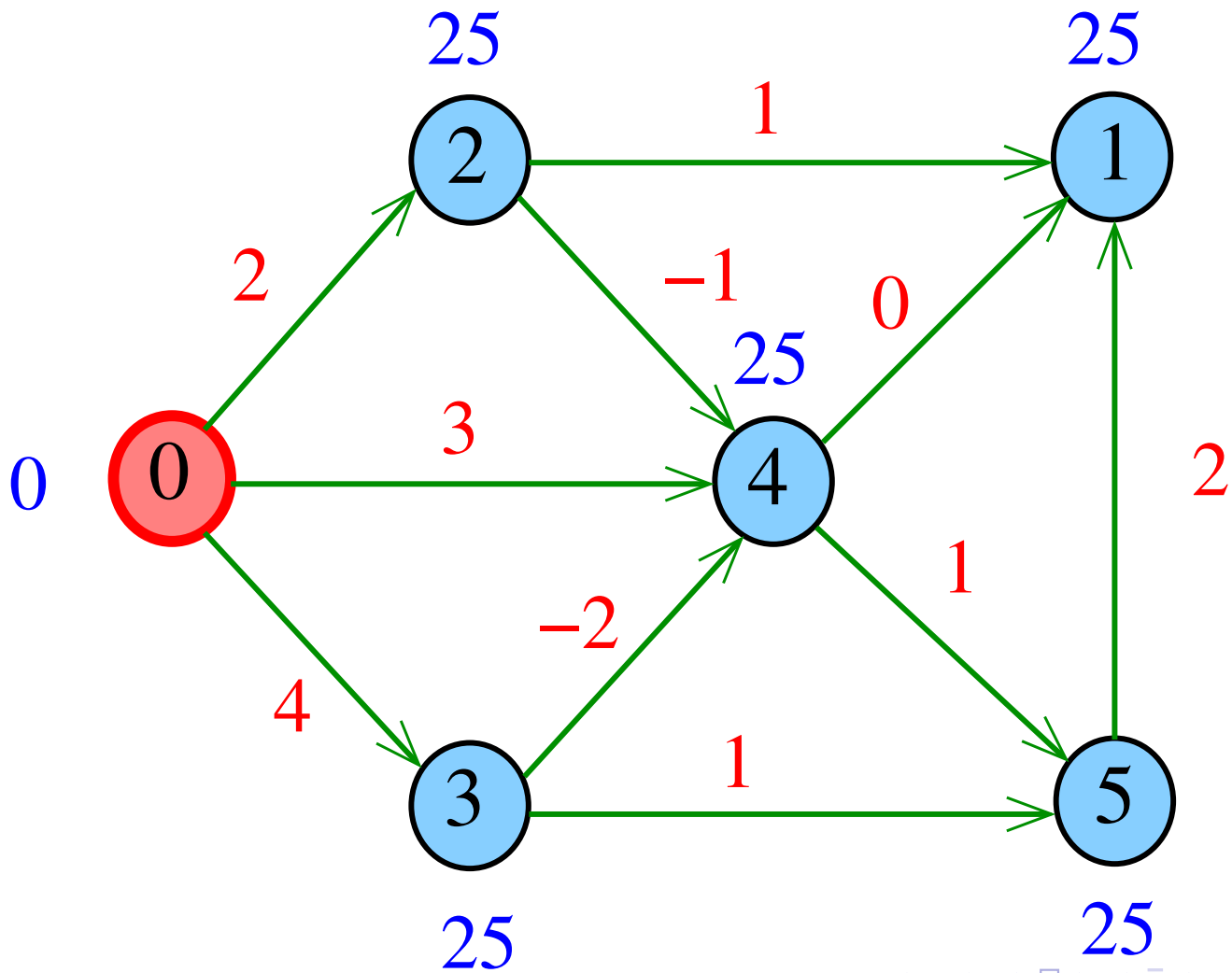
Simulação

i	0	1	2	3	4	5
ts[i]	0	3	2	4	5	1



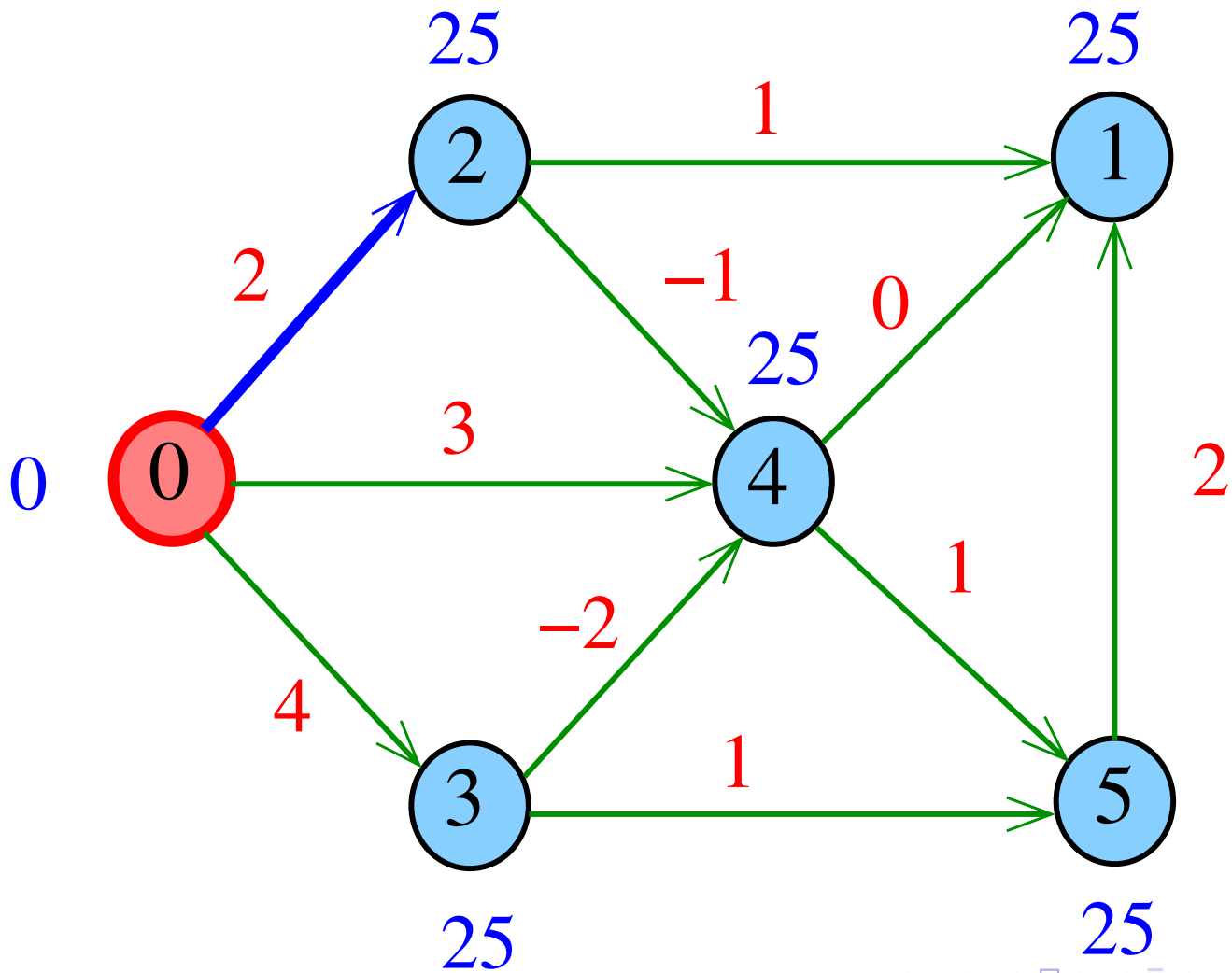
Simulação

i	0	1	2	3	4	5
ts[i]	0	3	2	4	5	1



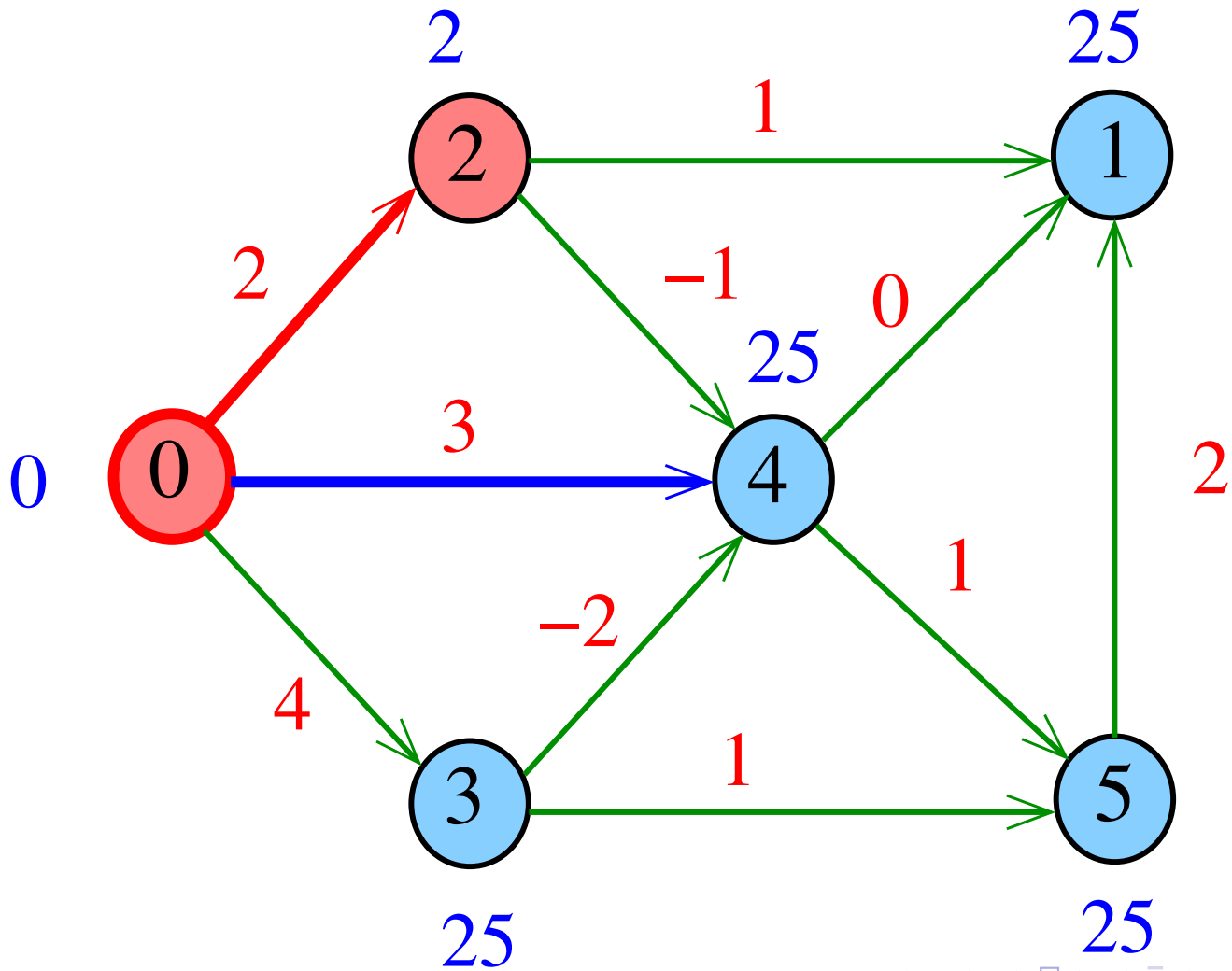
Simulação

i	0	1	2	3	4	5
ts[i]	0	3	2	4	5	1



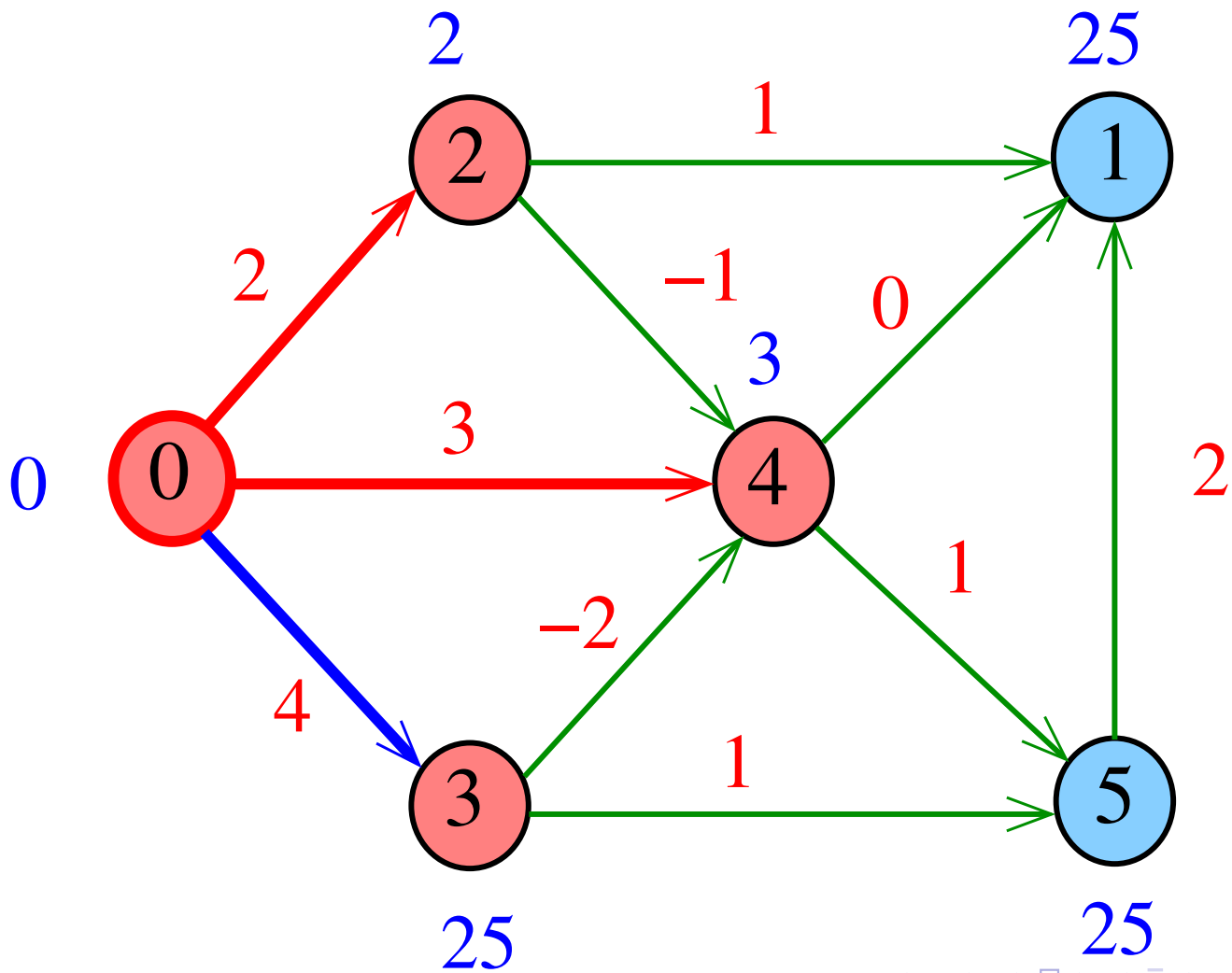
Simulação

i	0	1	2	3	4	5
ts[i]	0	3	2	4	5	1



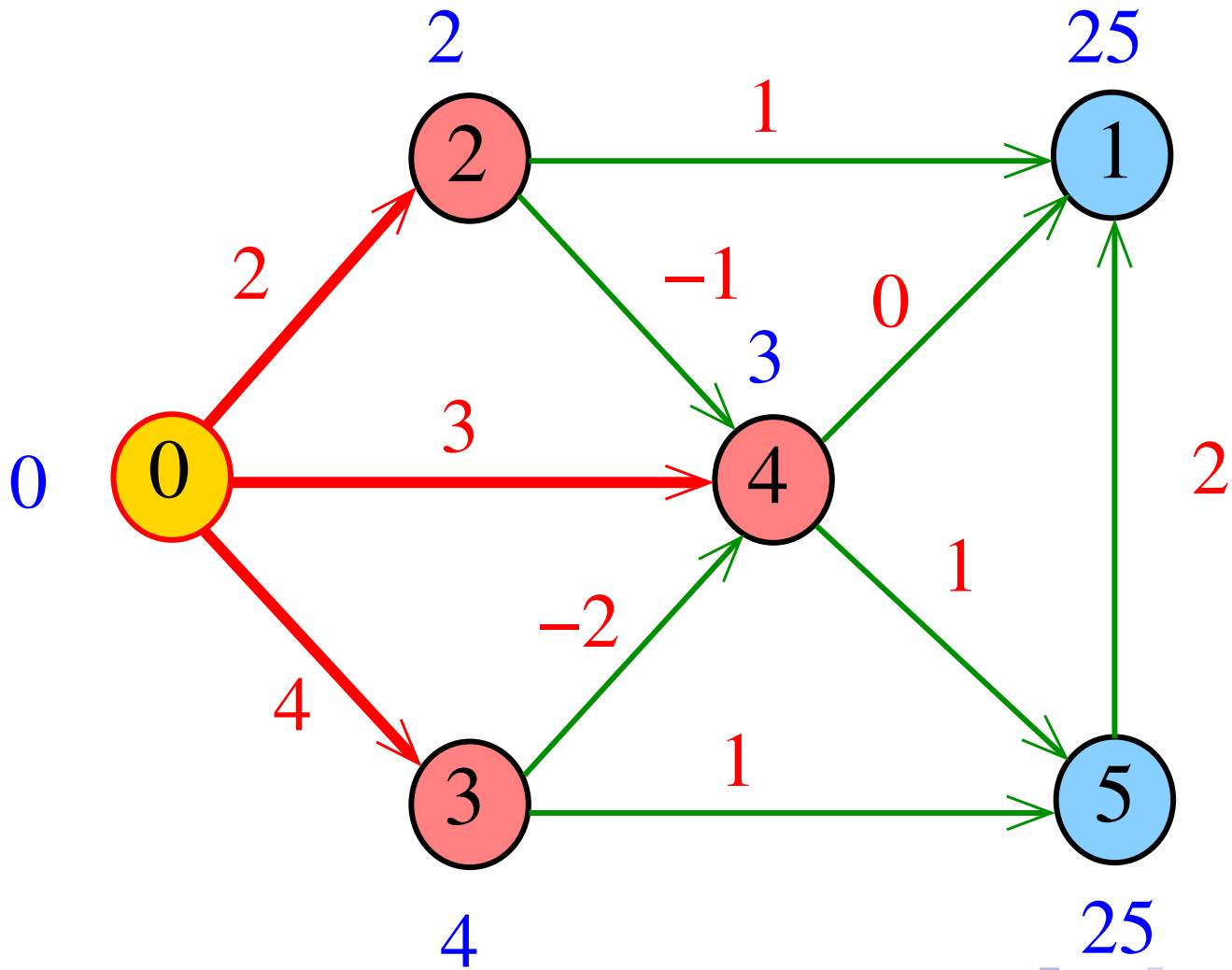
Simulação

i	0	1	2	3	4	5
ts[i]	0	3	2	4	5	1



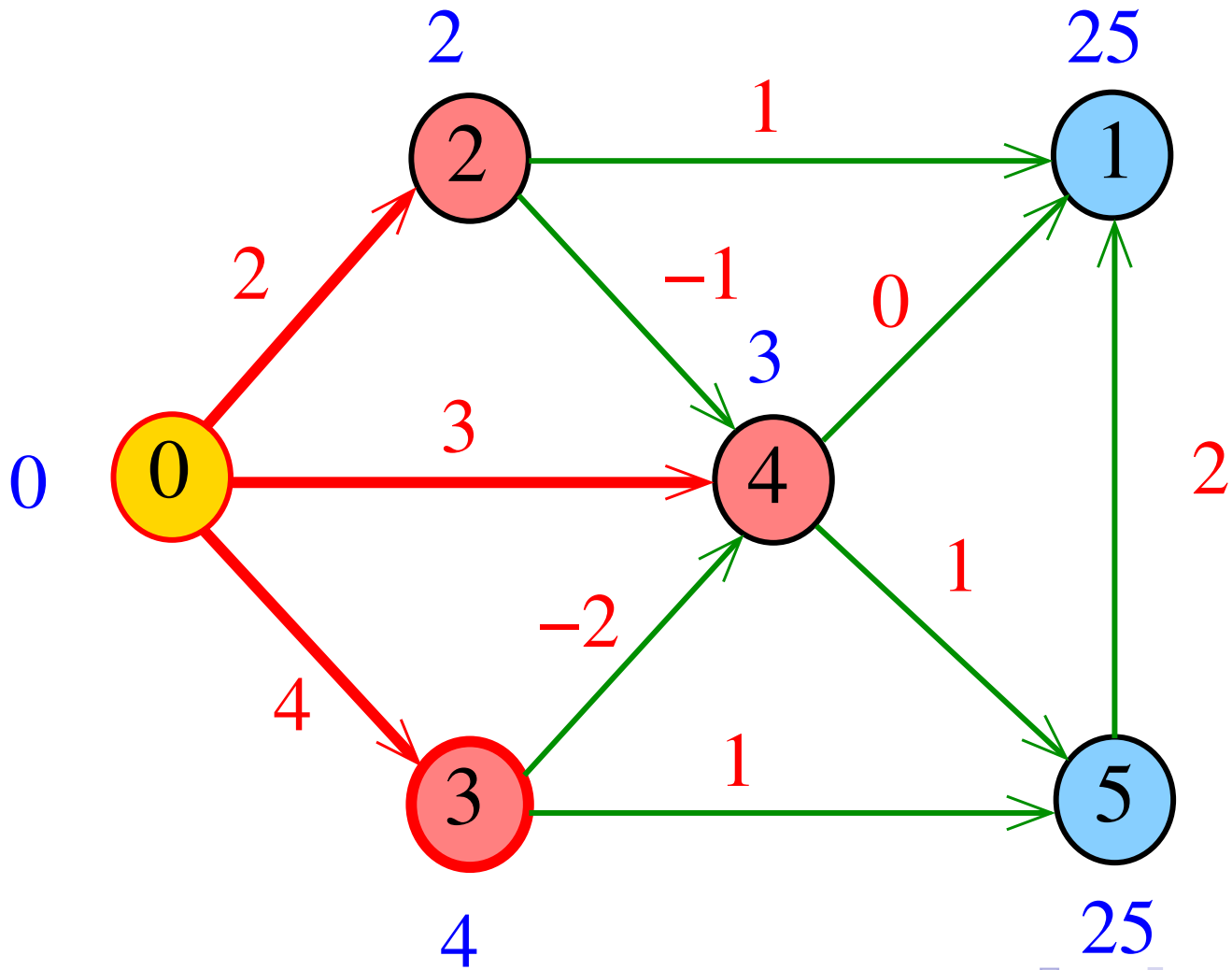
Simulação

i	0	1	2	3	4	5
ts[i]	0	3	2	4	5	1



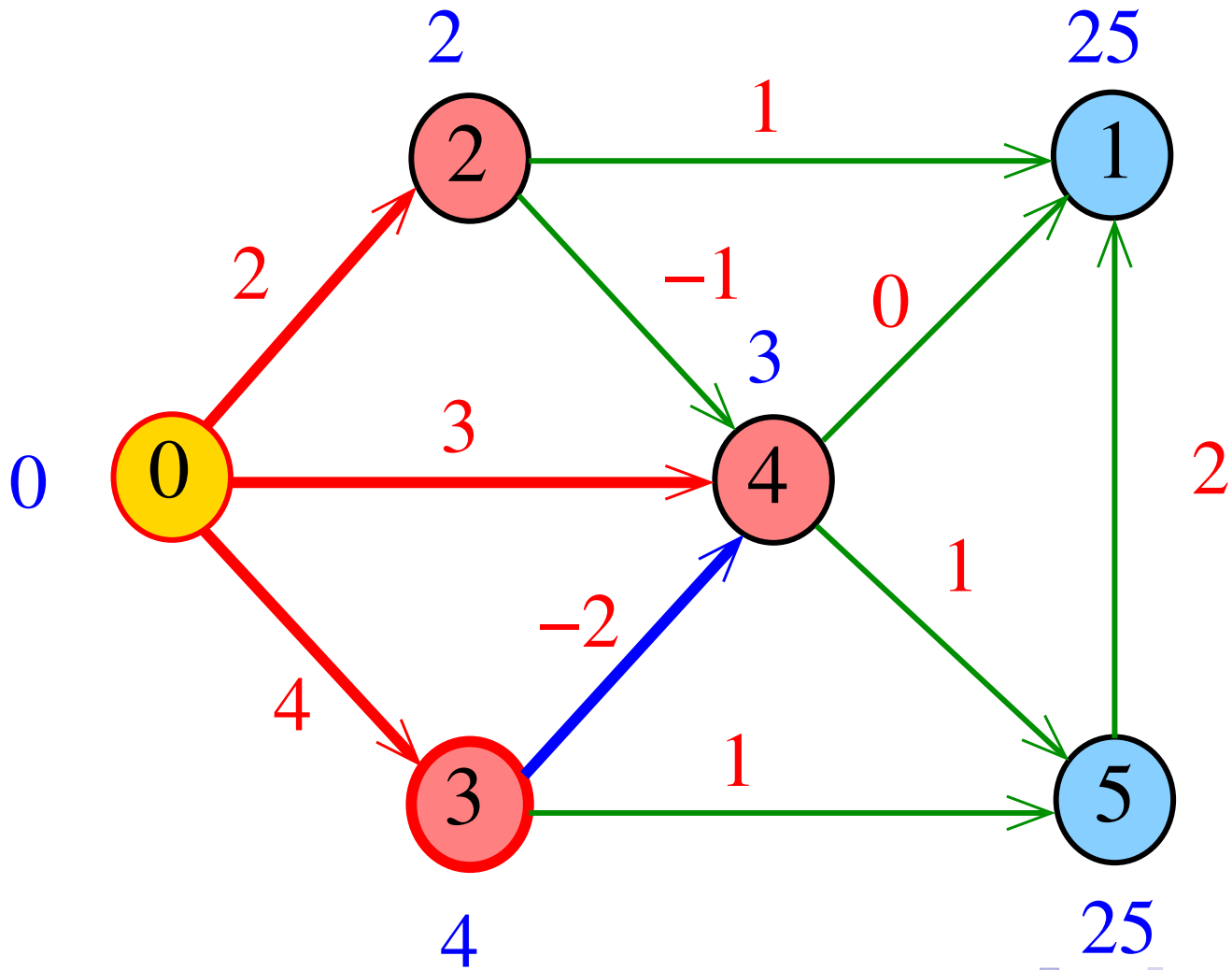
Simulação

i	0	1	2	3	4	5
ts[i]	0	3	2	4	5	1



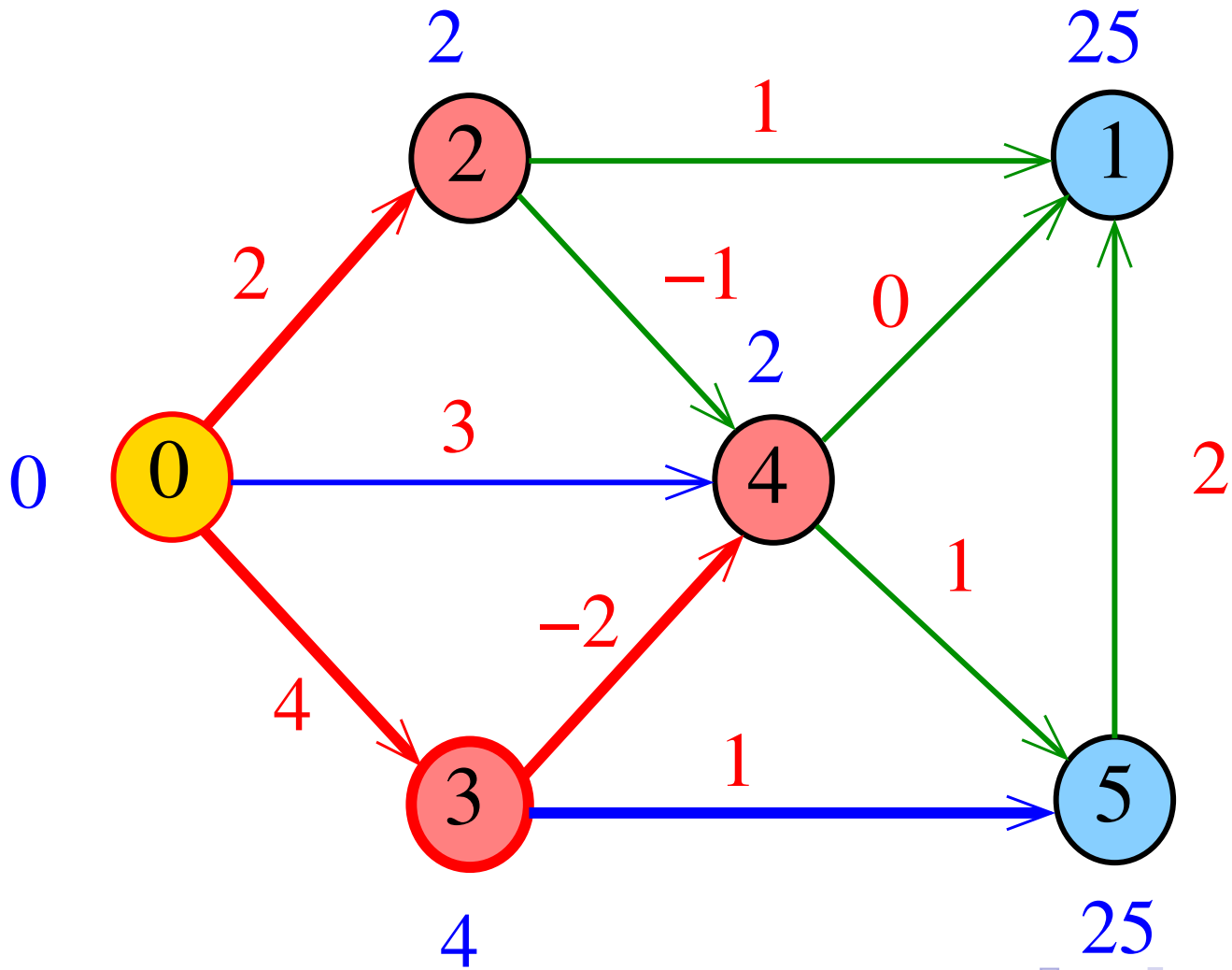
Simulação

i	0	1	2	3	4	5
ts[i]	0	3	2	4	5	1



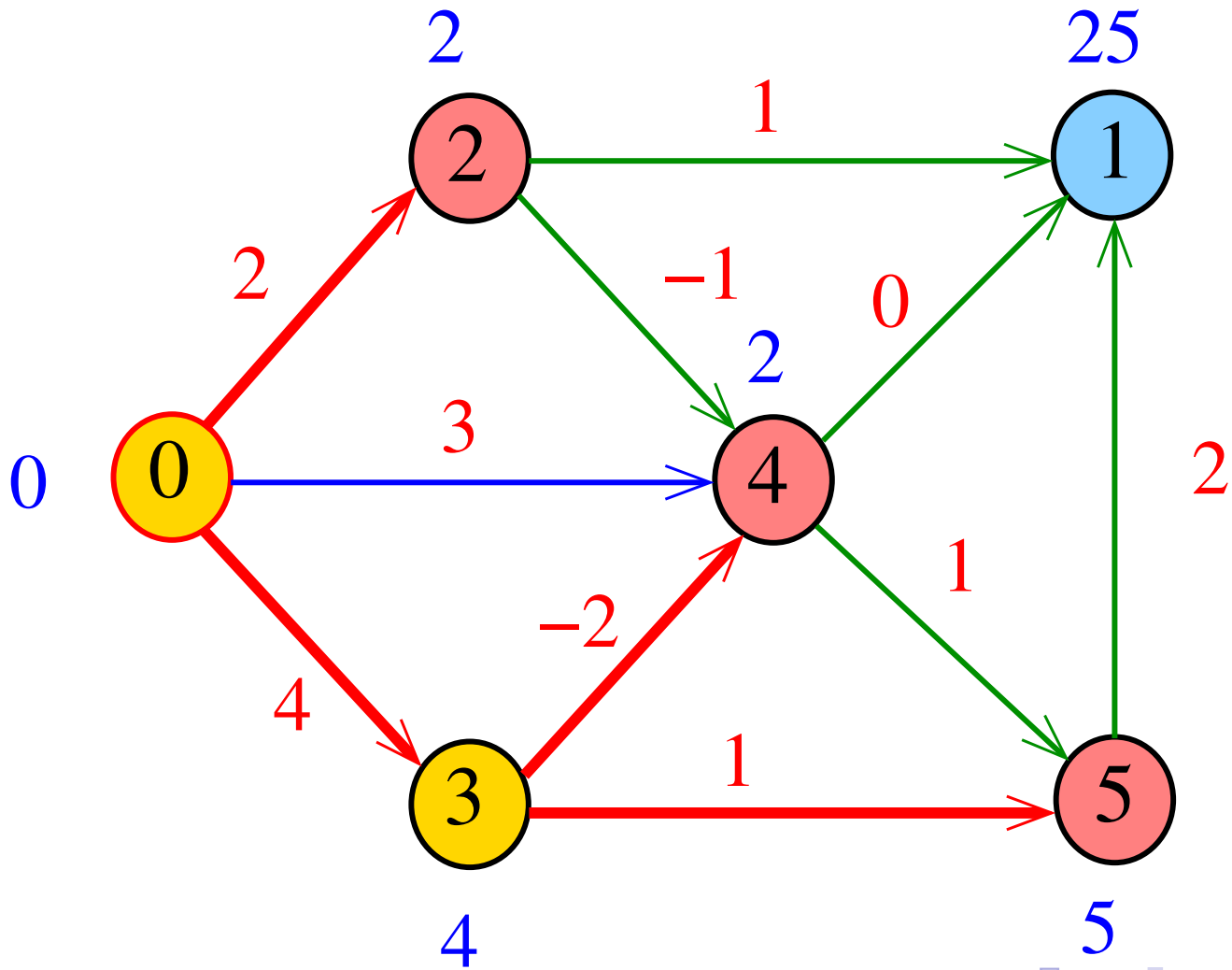
Simulação

i	0	1	2	3	4	5
ts[i]	0	3	2	4	5	1



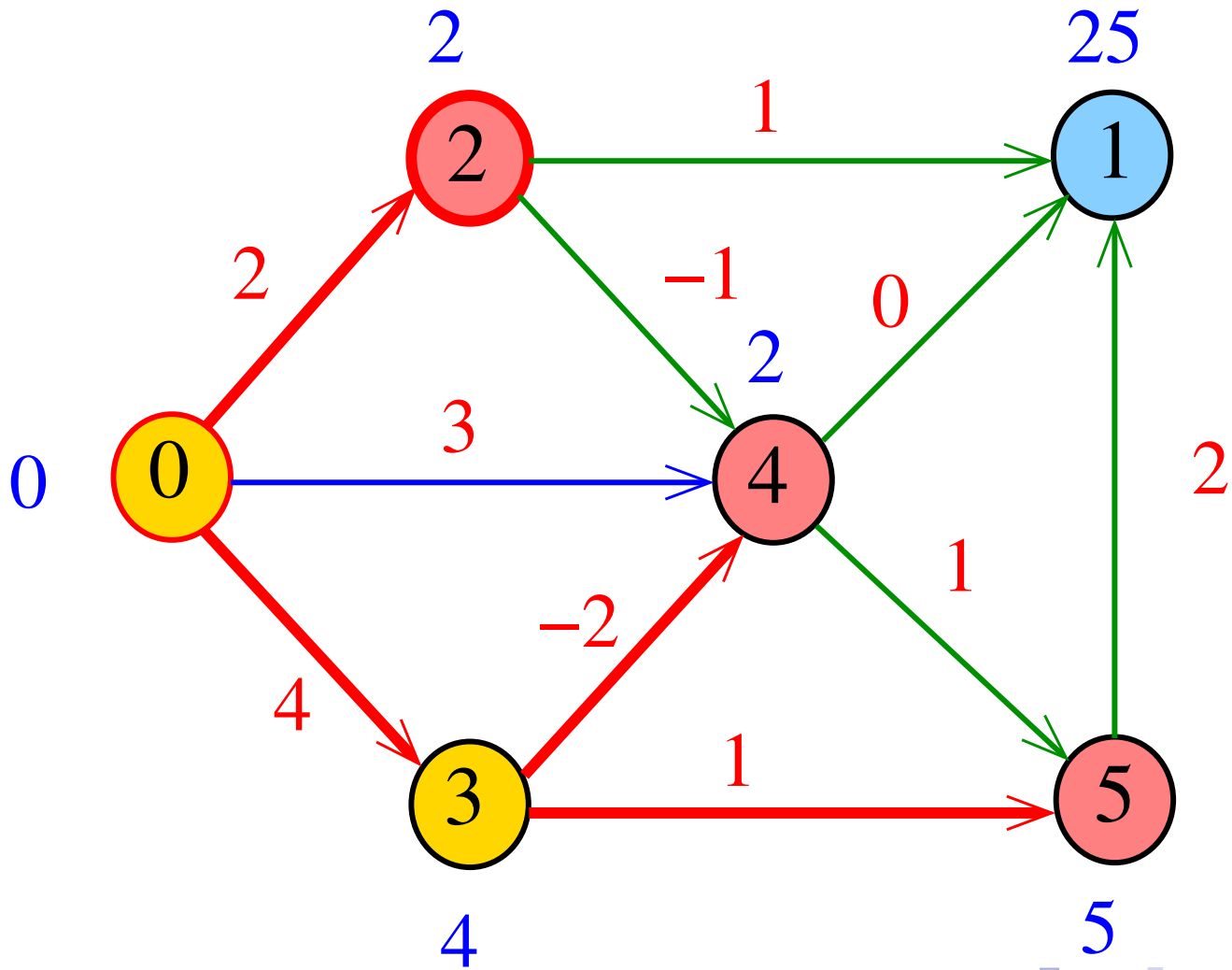
Simulação

i	0	1	2	3	4	5
ts[i]	0	3	2	4	5	1



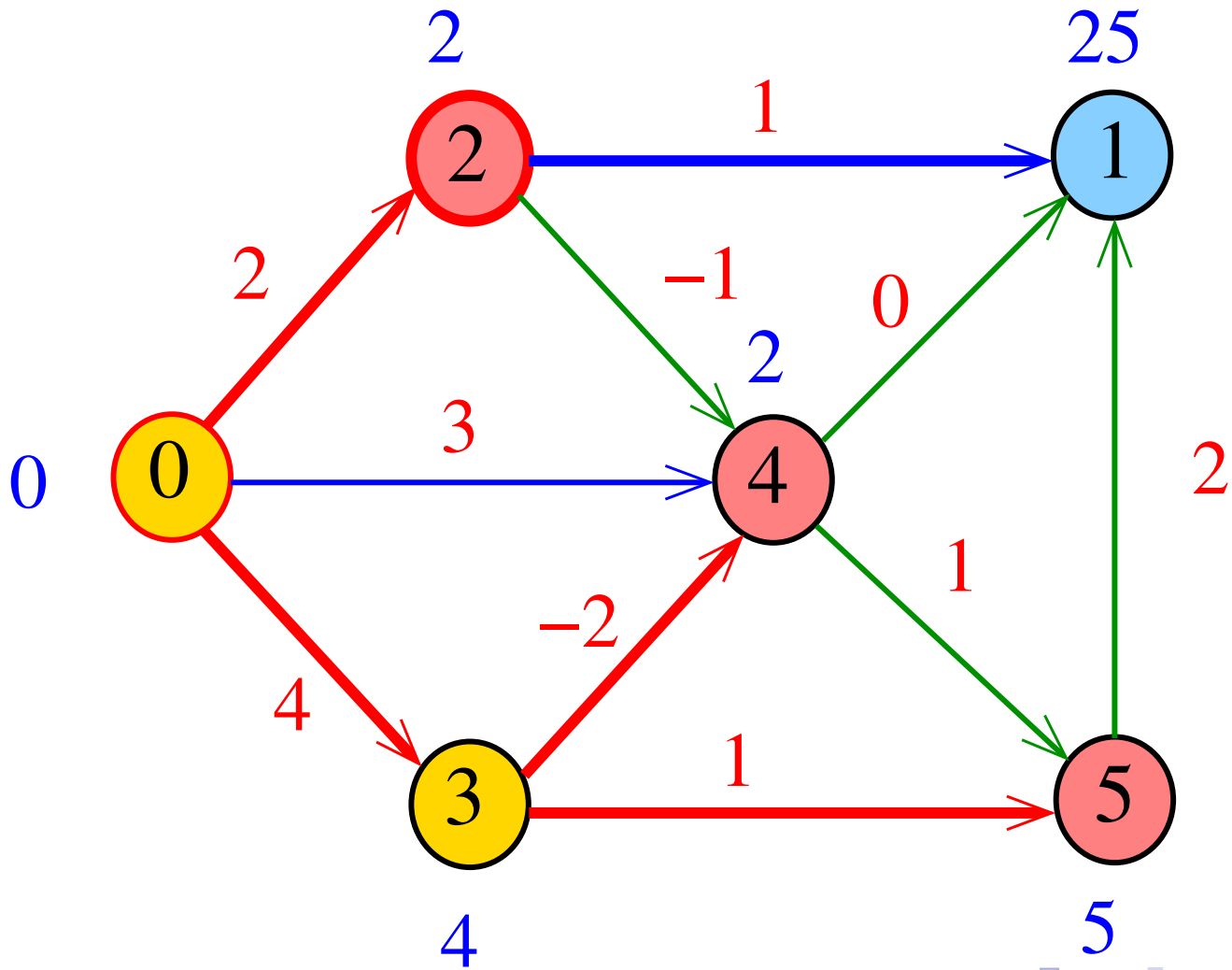
Simulação

i	0	1	2	3	4	5
ts[i]	0	3	2	4	5	1



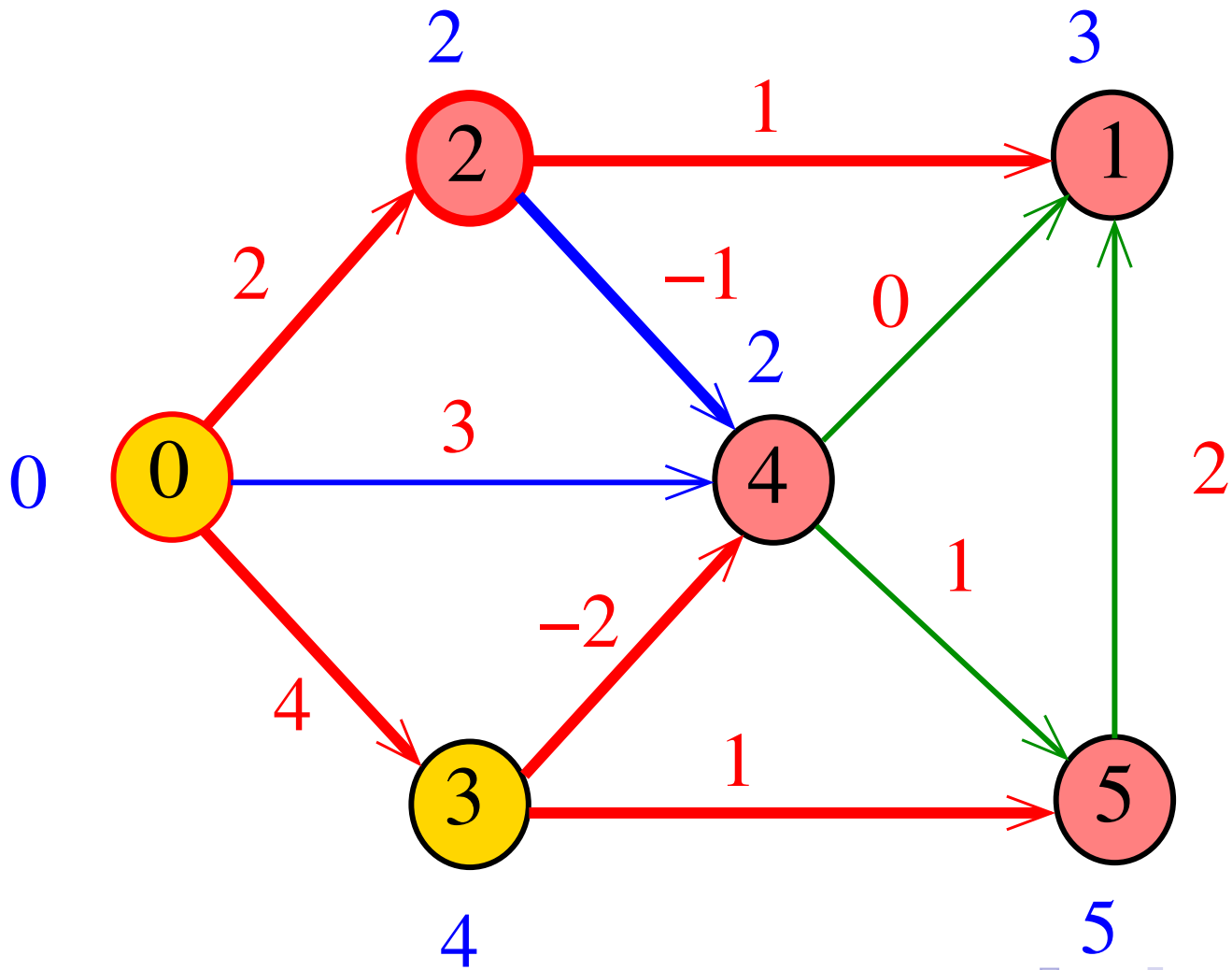
Simulação

i	0	1	2	3	4	5
ts[i]	0	3	2	4	5	1



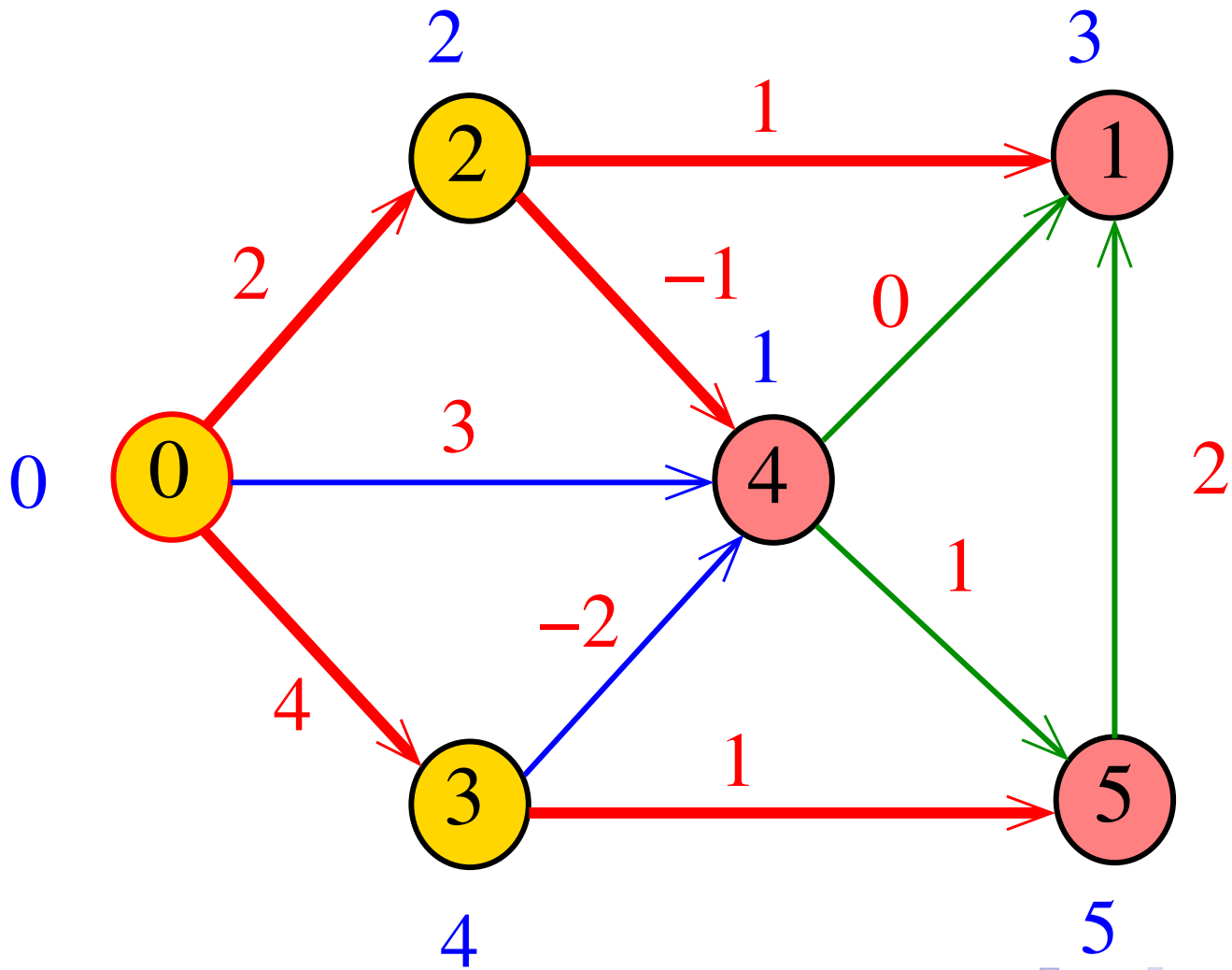
Simulação

i	0	1	2	3	4	5
ts[i]	0	3	2	4	5	1



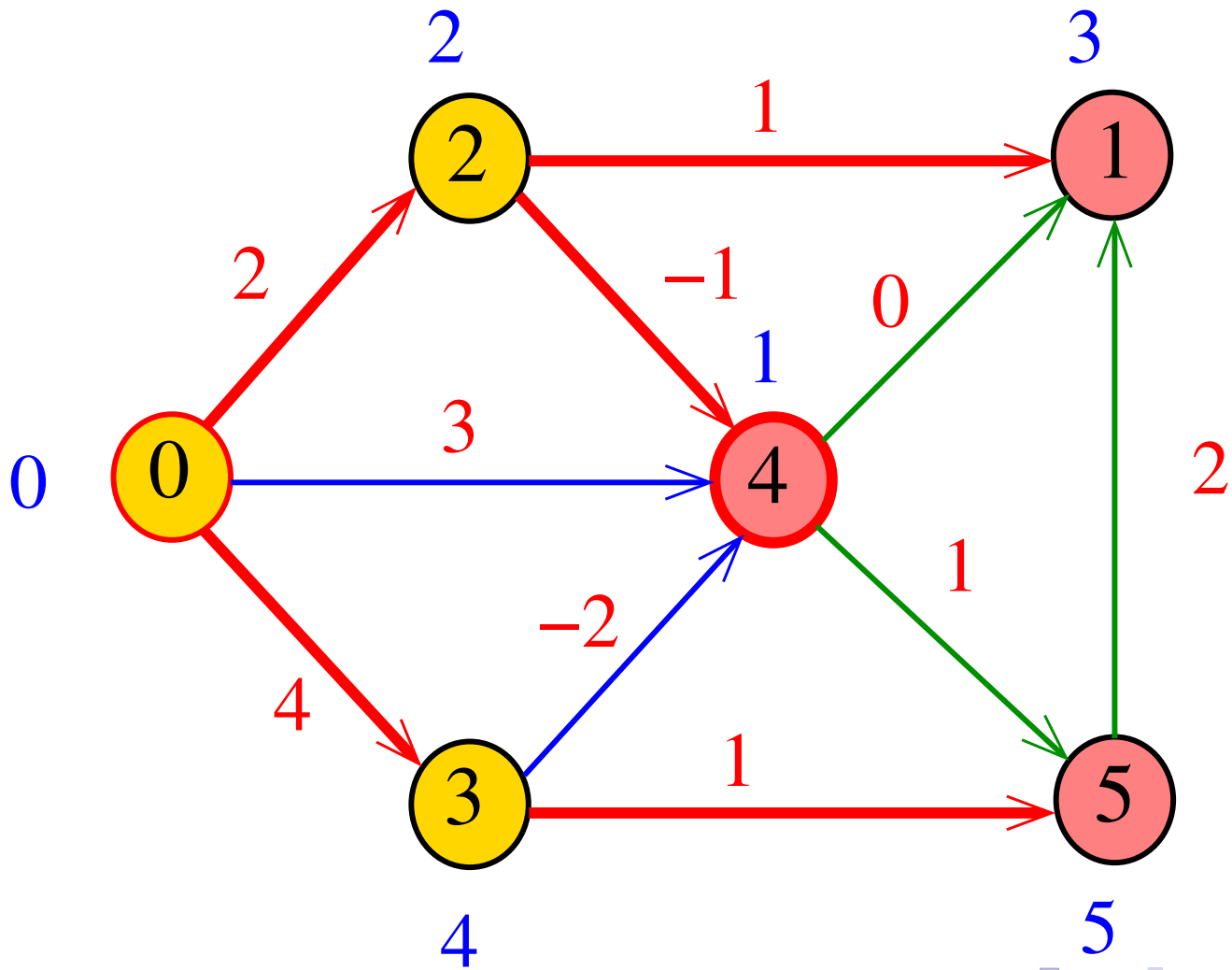
Simulação

i	0	1	2	3	4	5
ts[i]	0	3	2	4	5	1



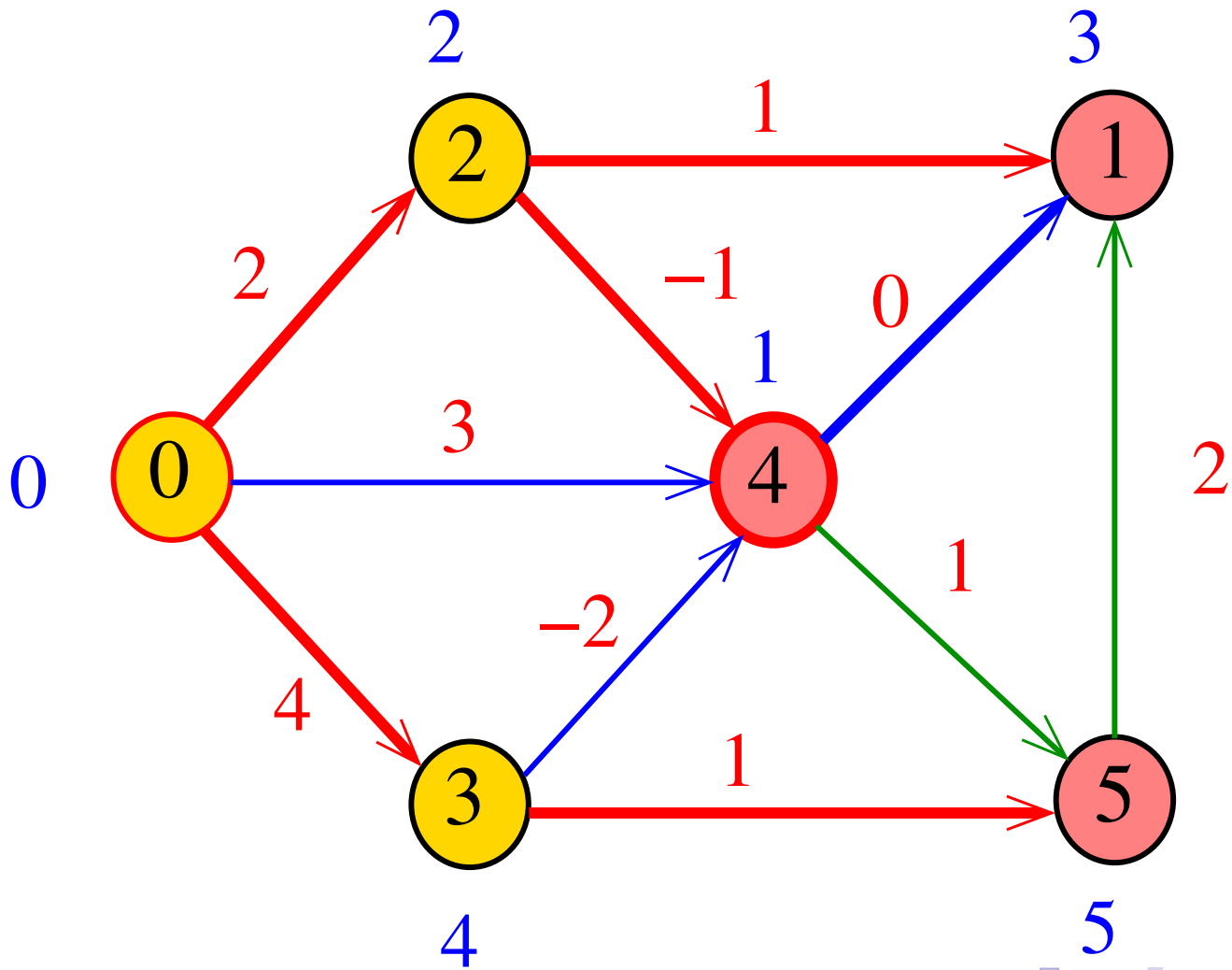
Simulação

i	0	1	2	3	4	5
ts[i]	0	3	2	4	5	1



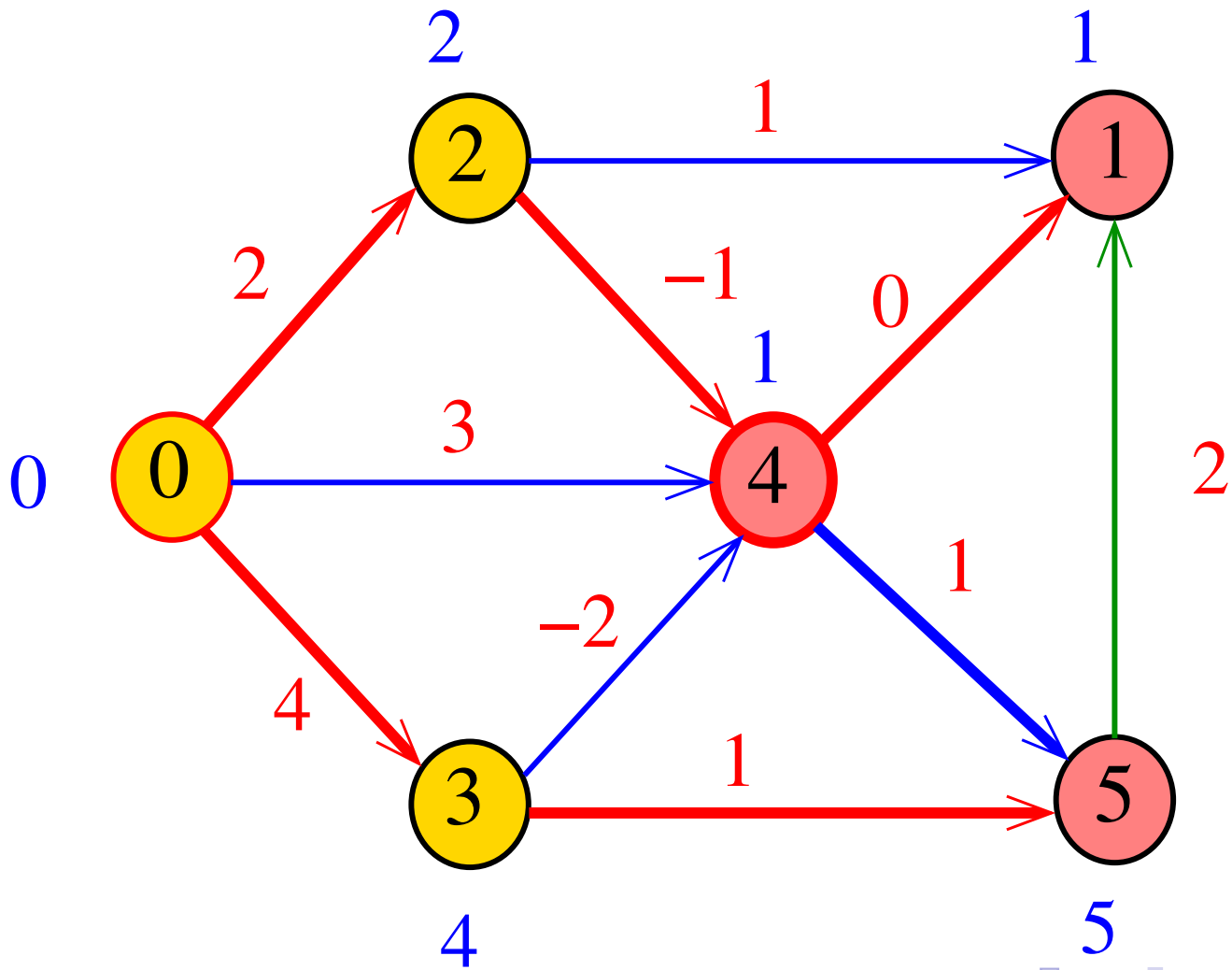
Simulação

i	0	1	2	3	4	5
ts[i]	0	3	2	4	5	1



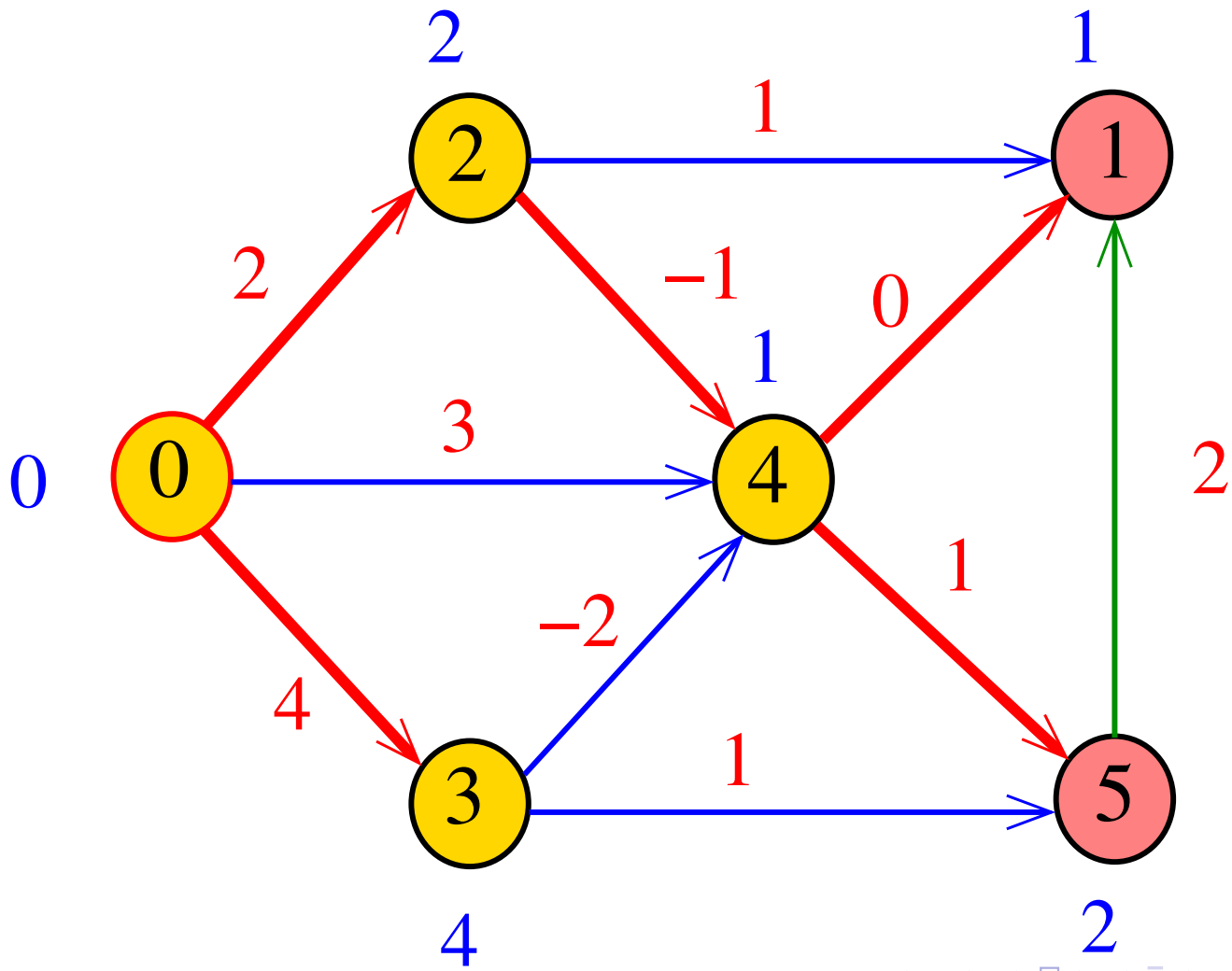
Simulação

i	0	1	2	3	4	5
ts[i]	0	3	2	4	5	1



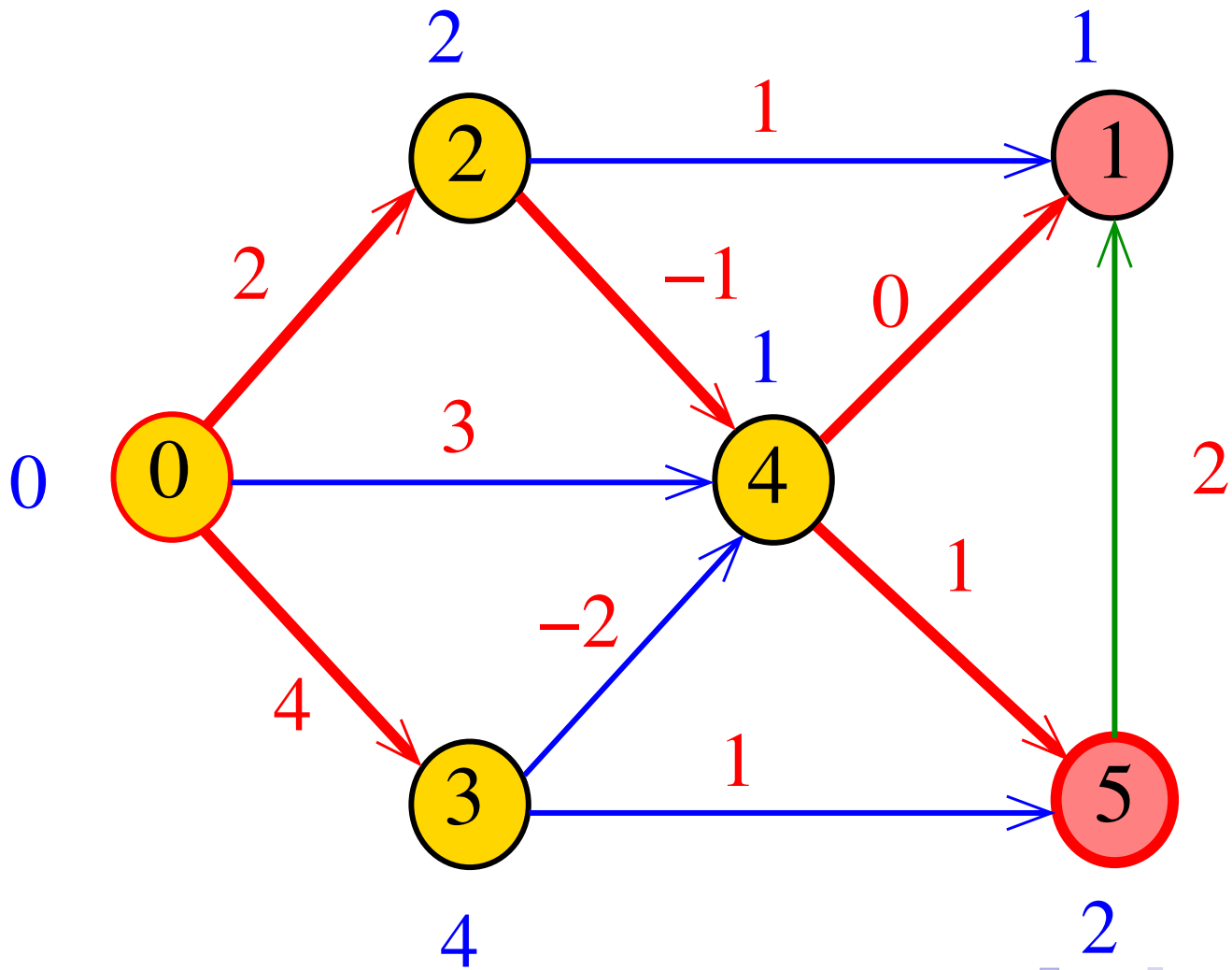
Simulação

i	0	1	2	3	4	5
ts[i]	0	3	2	4	5	1



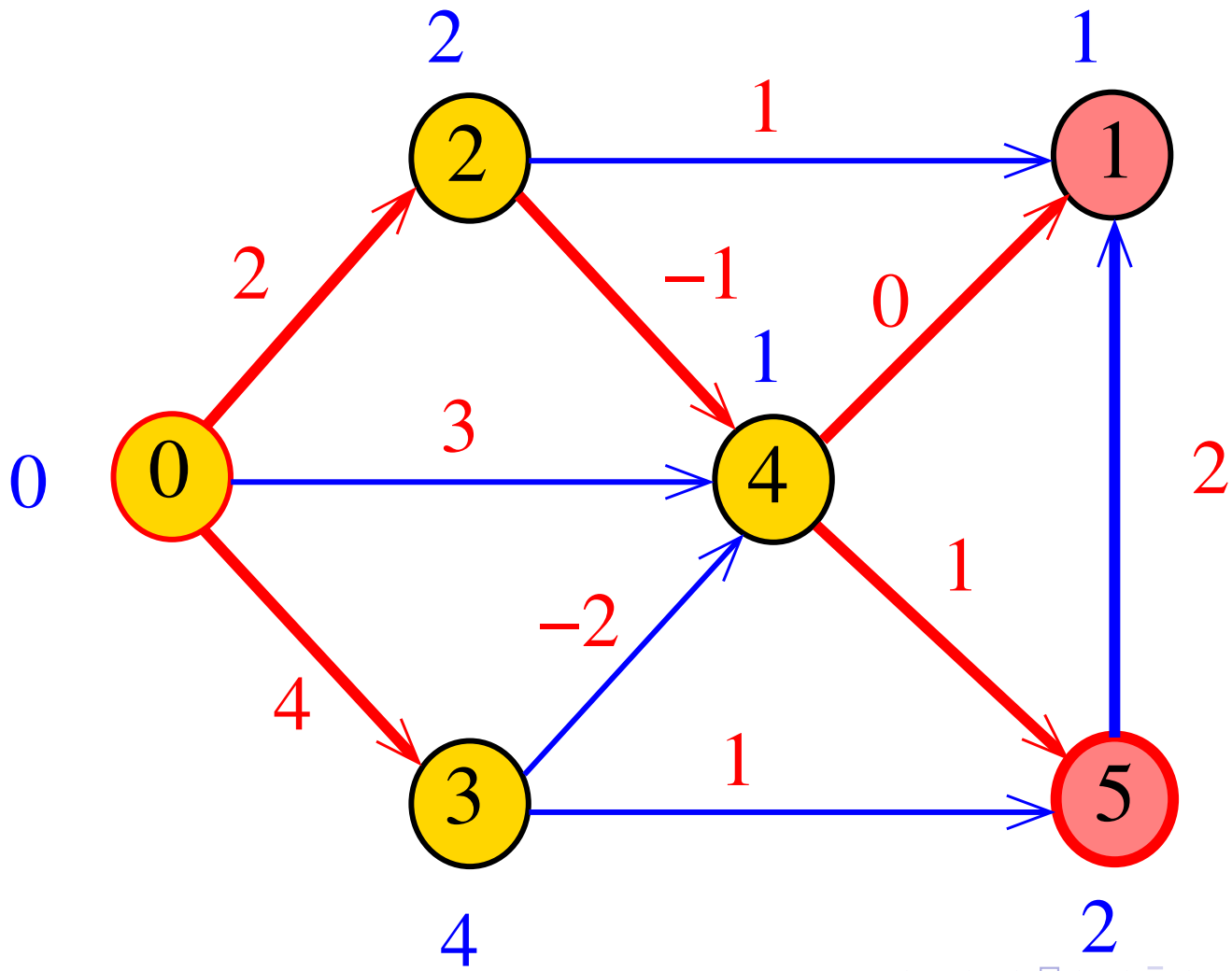
Simulação

i	0	1	2	3	4	5
ts[i]	0	3	2	4	5	1



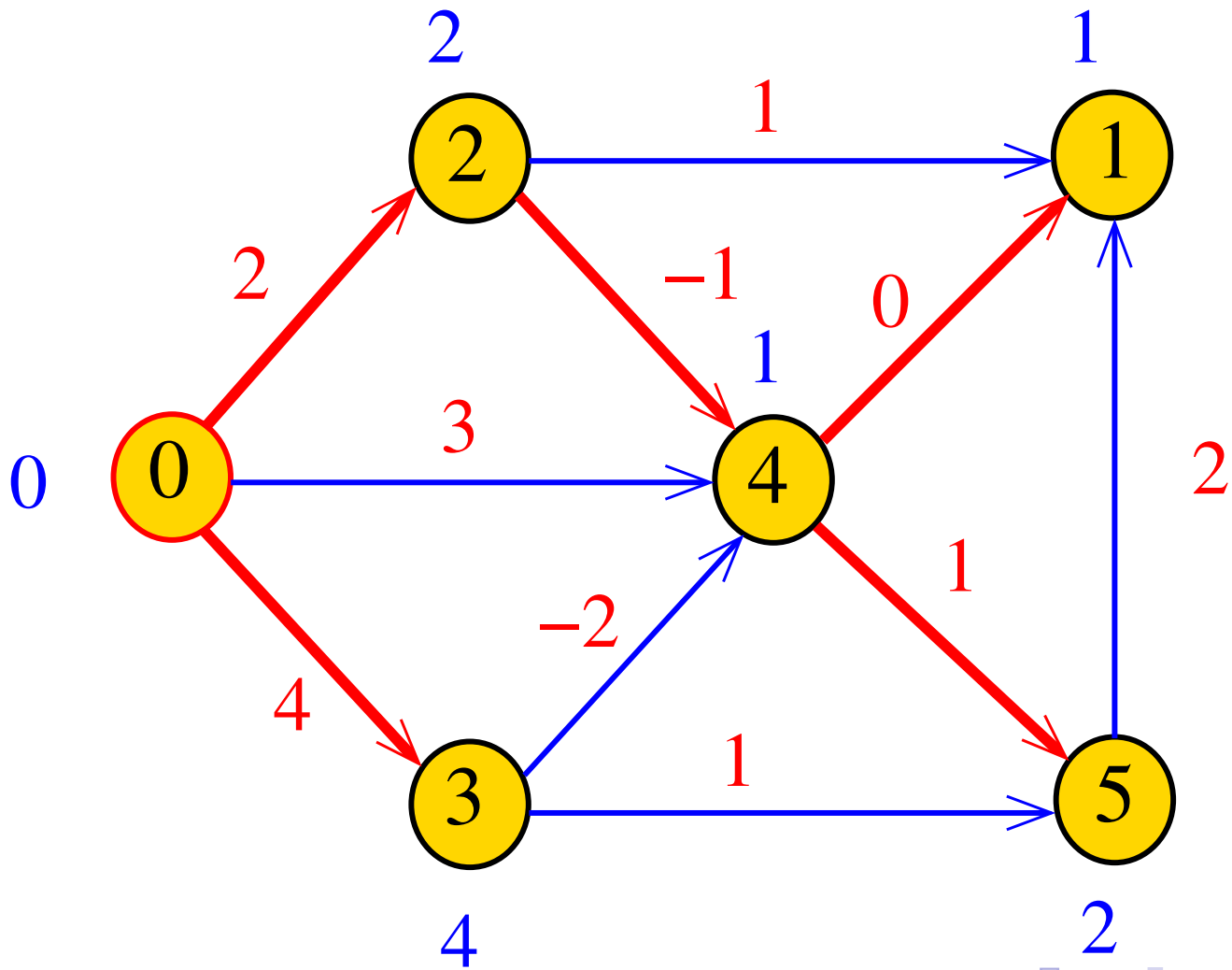
Simulação

i	0	1	2	3	4	5
ts[i]	0	3	2	4	5	1



Simulação

i	0	1	2	3	4	5
ts[i]	0	3	2	4	5	1



AcyclicSP

A rotina AcyclicSP recebe um DAG G
com custos possivelmente negativos.
Recebe também um vértice s.

A rotina determina uma ordenação topológica dos vértices de G através da rotina DFStopological modificada para trabalhar com EWDigraphs.

Para cada vértice t, a função calcula o custo de um caminho de custo mínimo de s a t.
Esse número é depositado em distTo[t].

AcyclicSP: estrutura e rotinas

```
static struct spdag {
    double INFINITY;
    int s;
    double *distTo;
    int    *edgeTo;
};
typedef struct spdag *spDAG;

spDAG AcyclicSP(EWDigraph, int);
static void acyclic(EWDigraph, int);

/* Métodos copiados de BFSpaths.  */
bool hasPath(int);
bool distTo(int);
Stack pathTo(int);
```

AcyclicSP

Encontra um caminho de s a
todo vértice alcançável a partir de s.

```
spDAG AcyclicSP(EWDigraph G, int s) {
    spDAG T = mallocSafe(sizeof(*T));

    T->INFINITY = DBL_MAX;       /* double máximo */
    T->s = s;
    T->distTo = mallocSafe(G->V*sizeof(double));
    T->edgeTo = mallocSafe(G->V*sizeof(int));
    for (int v = 0; v < G->V; v++) {
        T->distTo[v] = T->INFINITY;
        T->edgeTo[v] = -1;
    }
    acyclic(T, G, s);
    return T
}
```

acyclic()

```
static void acyclic(spDAG T, EWDigraph G, int s) {
    DFStopological ts = DFStopological(G);
    Link a;   int v, w;   double d;

    T->distTo[s] = 0;
    while(!stackEmpty(ts->order)) {
        v = stackPop(ts->order);
        for (a = G->adj[v]; a != NULL; a = a->next) {
            w = a->end;
            d = T->distTo[v] + a->weight;
            if (T->distTo[w] > d) {
                T->edgeTo[w] = v;
                T->distTo[w] = d;
            }
        }
    }
}
```

Consumo de tempo

O consumo de tempo de `AcyclicSP` para vetor de listas de adjacência é $O(V + E)$.

O consumo de tempo de `AcyclicSP` para matriz de adjacências é $O(V^2)$.