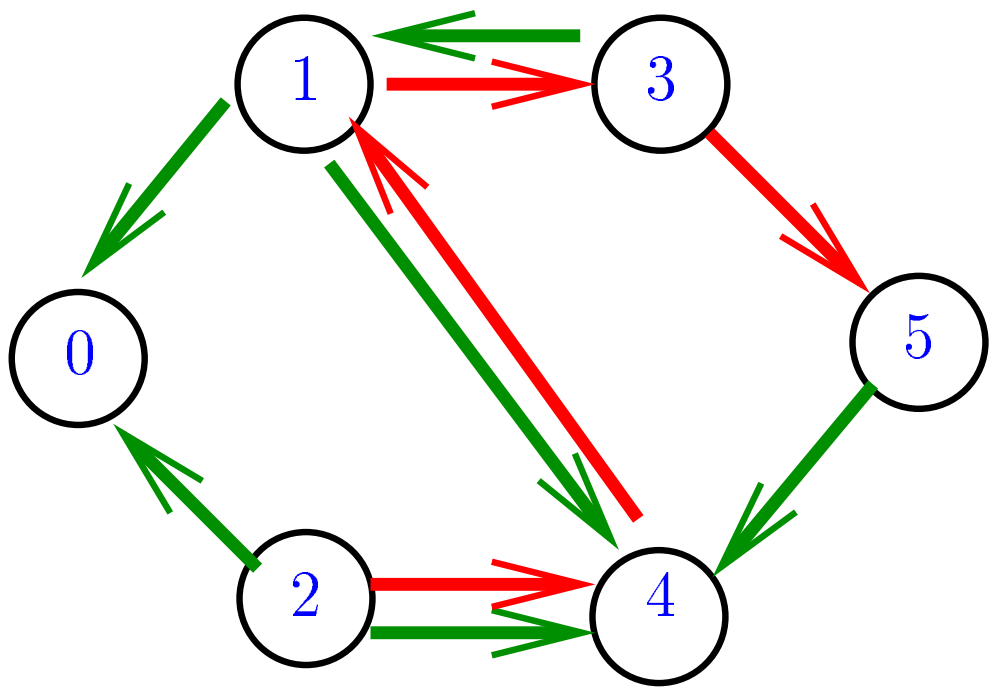
Melhores momentos

AULA 13

Distância

A **distância** de um vértice `s` a um vértice `t` é o menor comprimento de um caminho de `s` a `t`. Se não existe caminho de `s` a `t` a distância é **infinita**

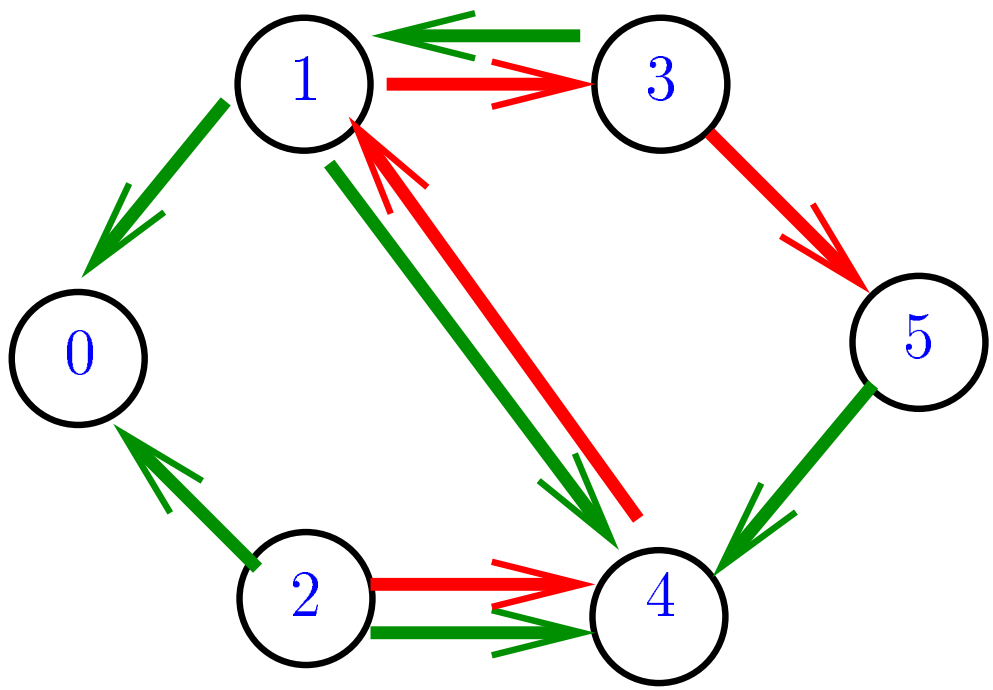
Exemplo: a distância de 2 a 5 é 4



Distância

A **distância** de um vértice `s` a um vértice `t` é o menor comprimento de um caminho de `s` a `t`. Se não existe caminho de `s` a `t` a distância é **infinita**

Exemplo: a distância de 0 a 2 é infinita

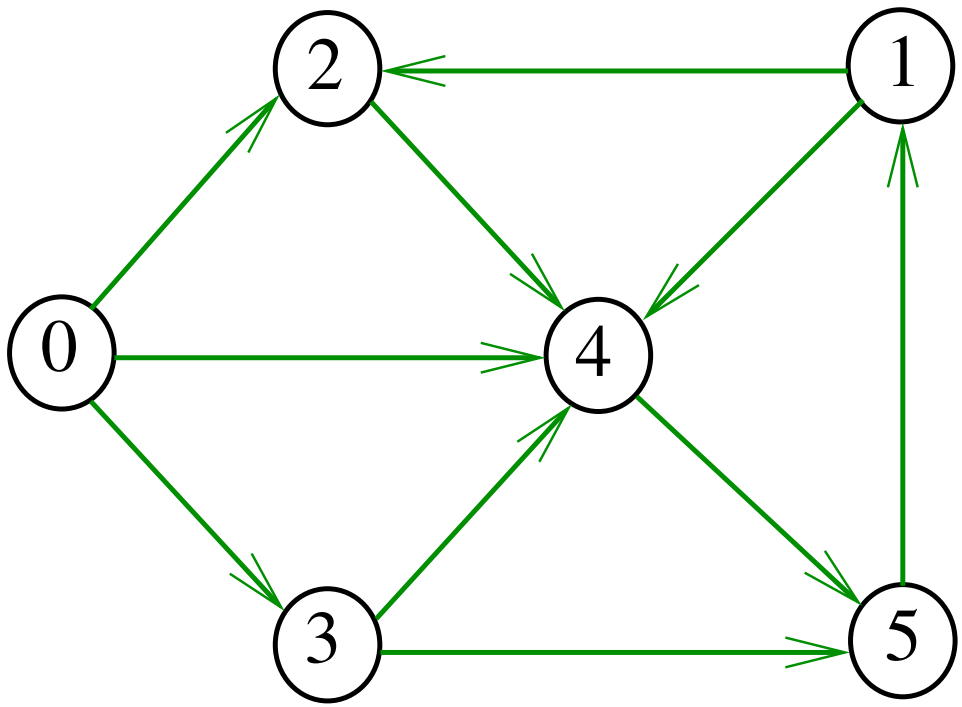


Calculando distâncias

Problema: dados um digrafo `G` e um vértice `s`, determinar a distância de `s` aos demais vértices do digrafo

Exemplo: para `s` = 0

v	0	1	2	3	4	5
dist[v]	0	3	1	1	1	2



Busca em largura

A **busca em largura** (=*breadth-first search search* = *BFS*) começa por um vértice, digamos `s`, especificado pelo usuário.

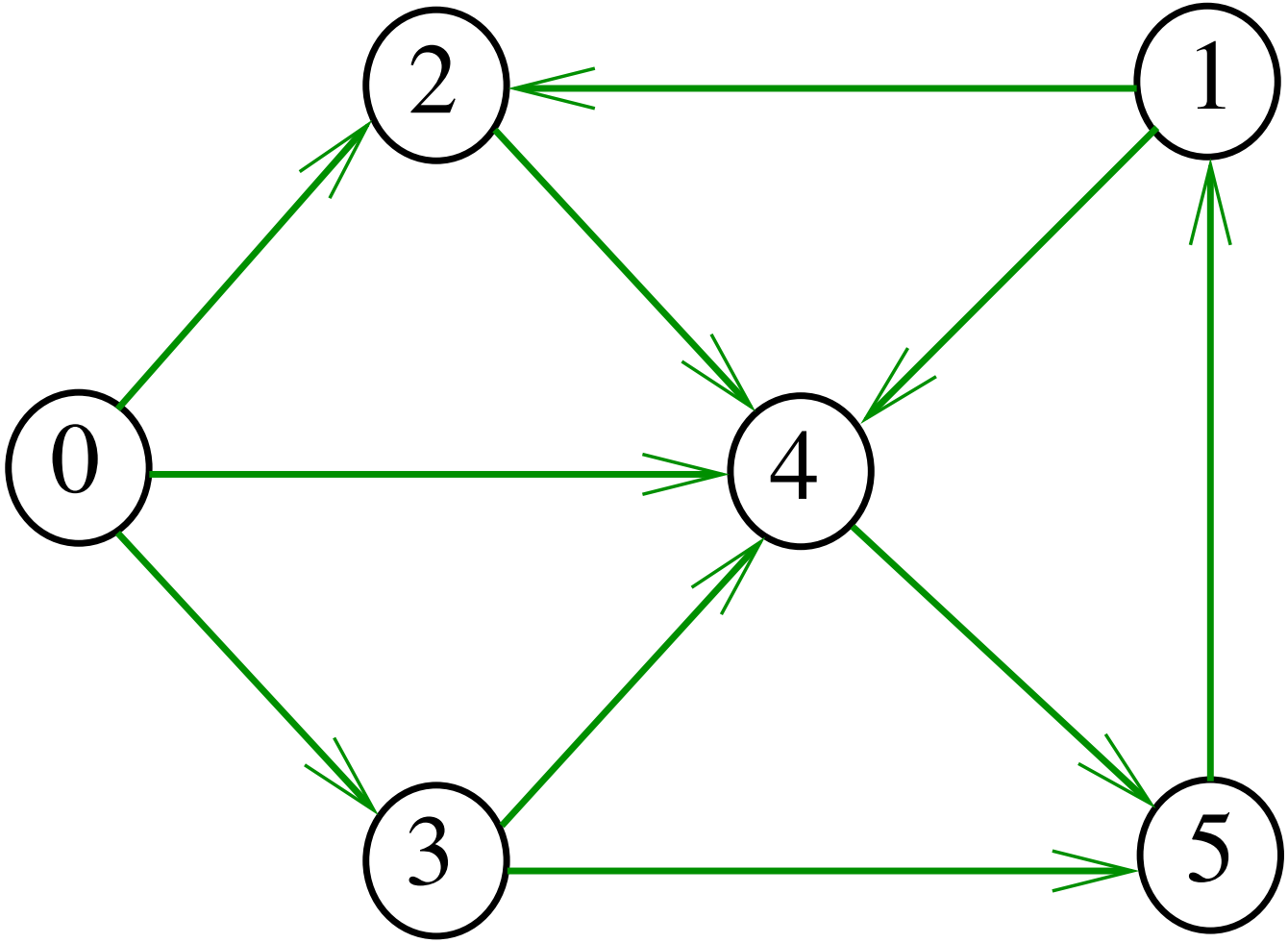
O algoritmo

visita `s`,
depois visita vértices à distância 1 de `s`,
depois visita vértices à distância 2 de `s`,
depois visita vértices à distância 3 de `s`,
e assim por diante

Simulação

i	0	1	2	3	4	5
q[i]						

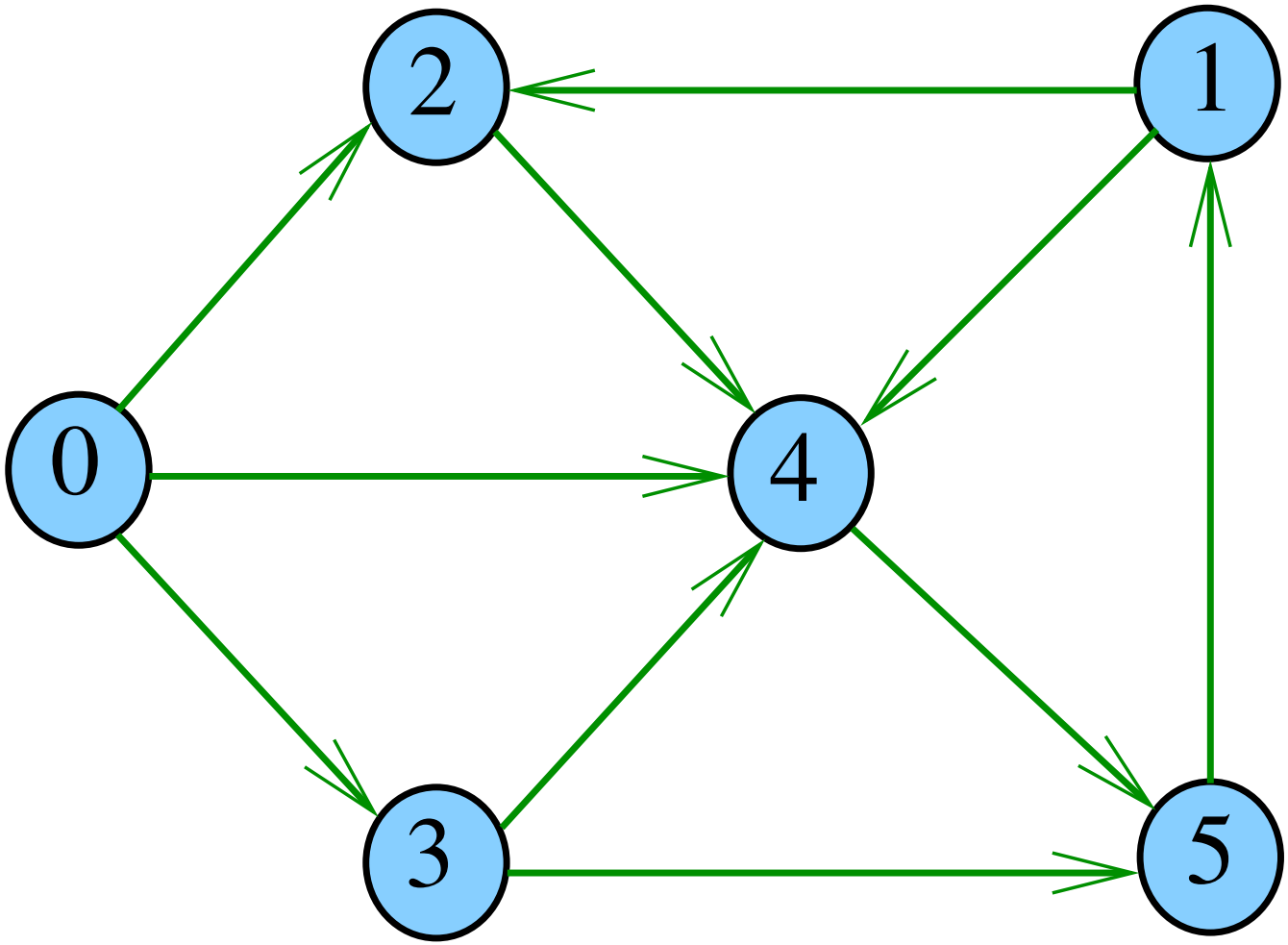
v	0	1	2	3	4	5
dist[v]						



Simulação

i	0	1	2	3	4	5
q[i]						

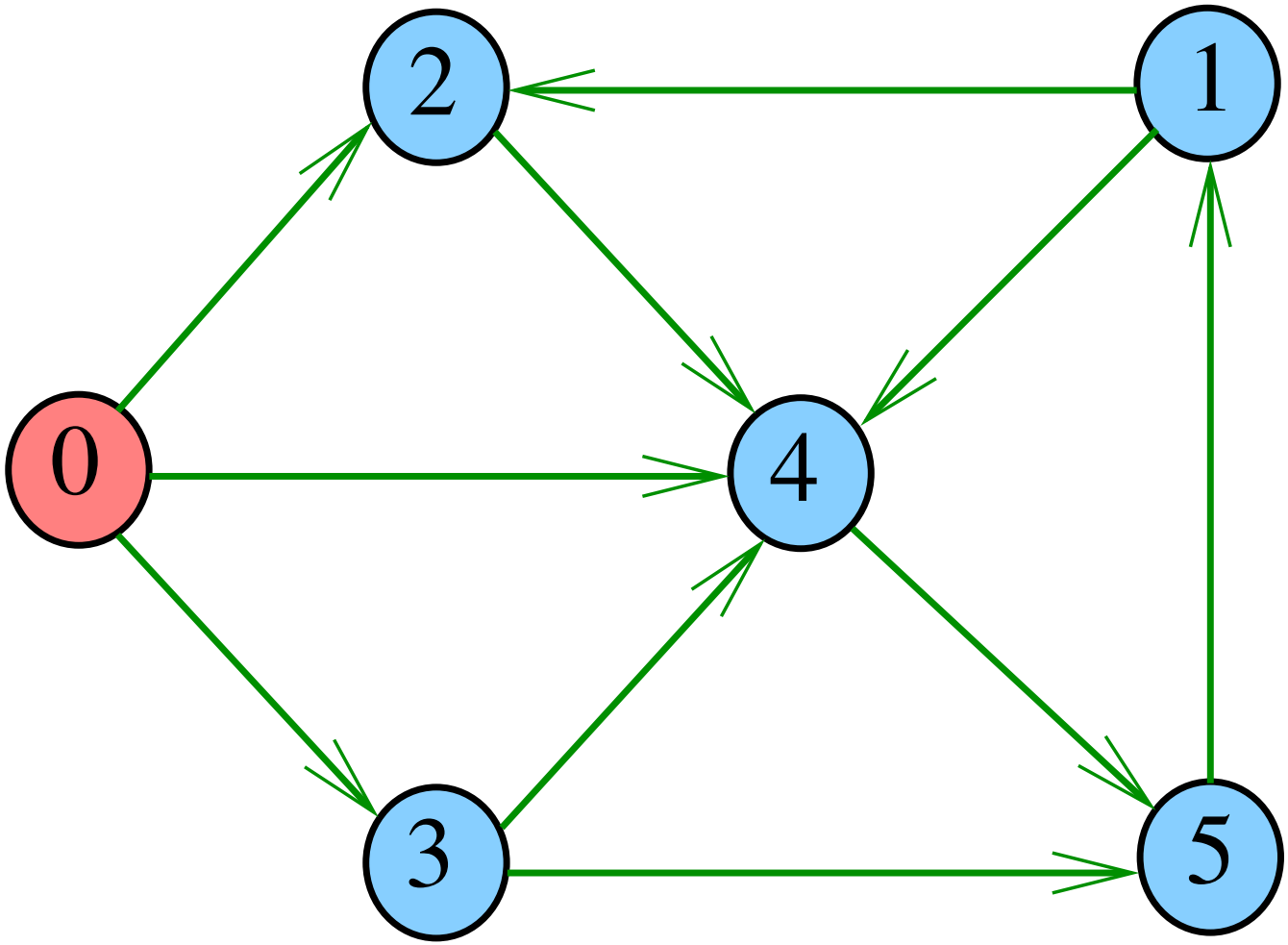
v	0	1	2	3	4	5
dist[v]	6	6	6	6	6	6



Simulação

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

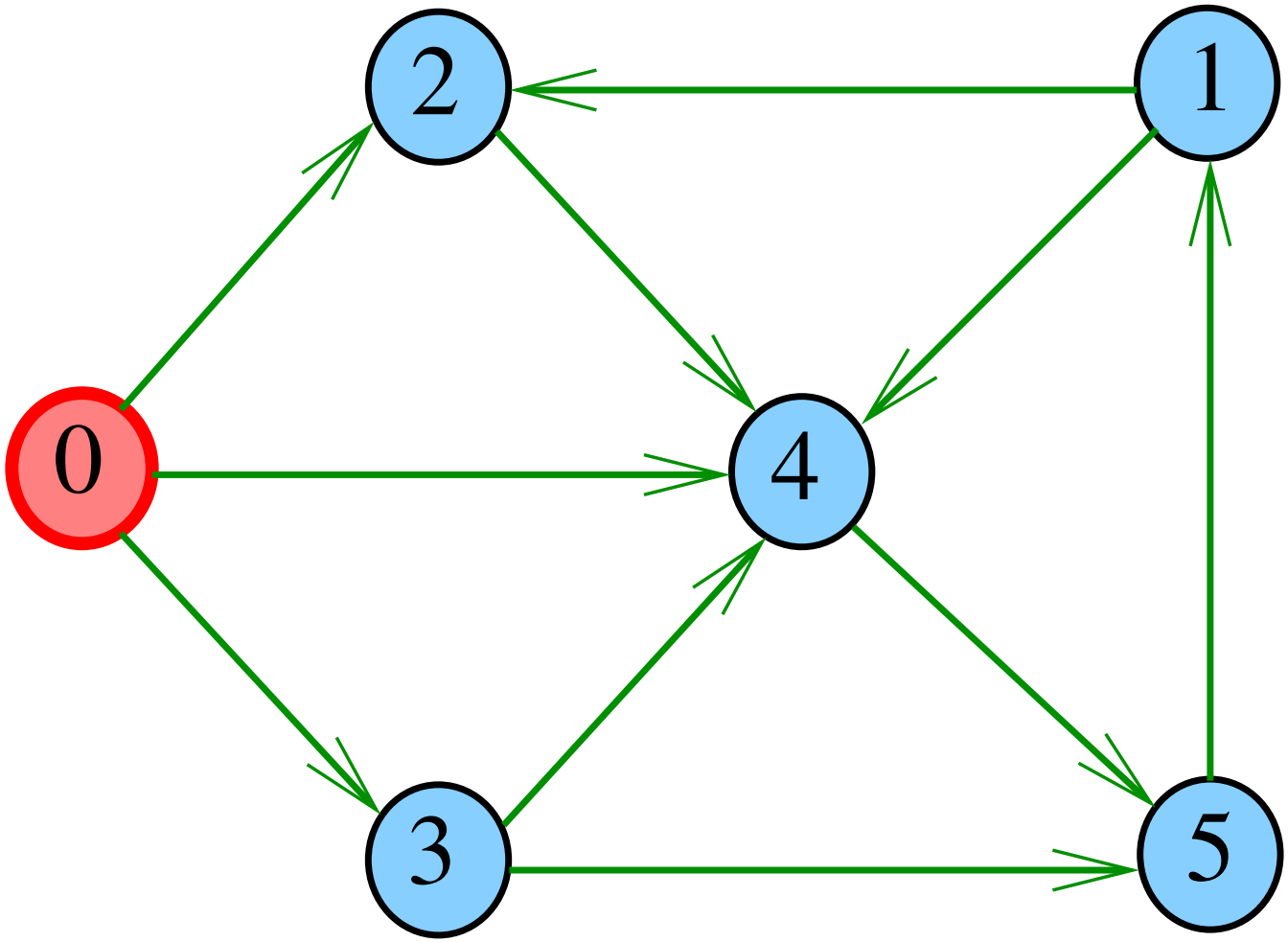
v	0	1	2	3	4	5
dist[v]	6	6	6	6	6	6



Simulação

i	0	1	2	3	4	5
q[i]	**0**					

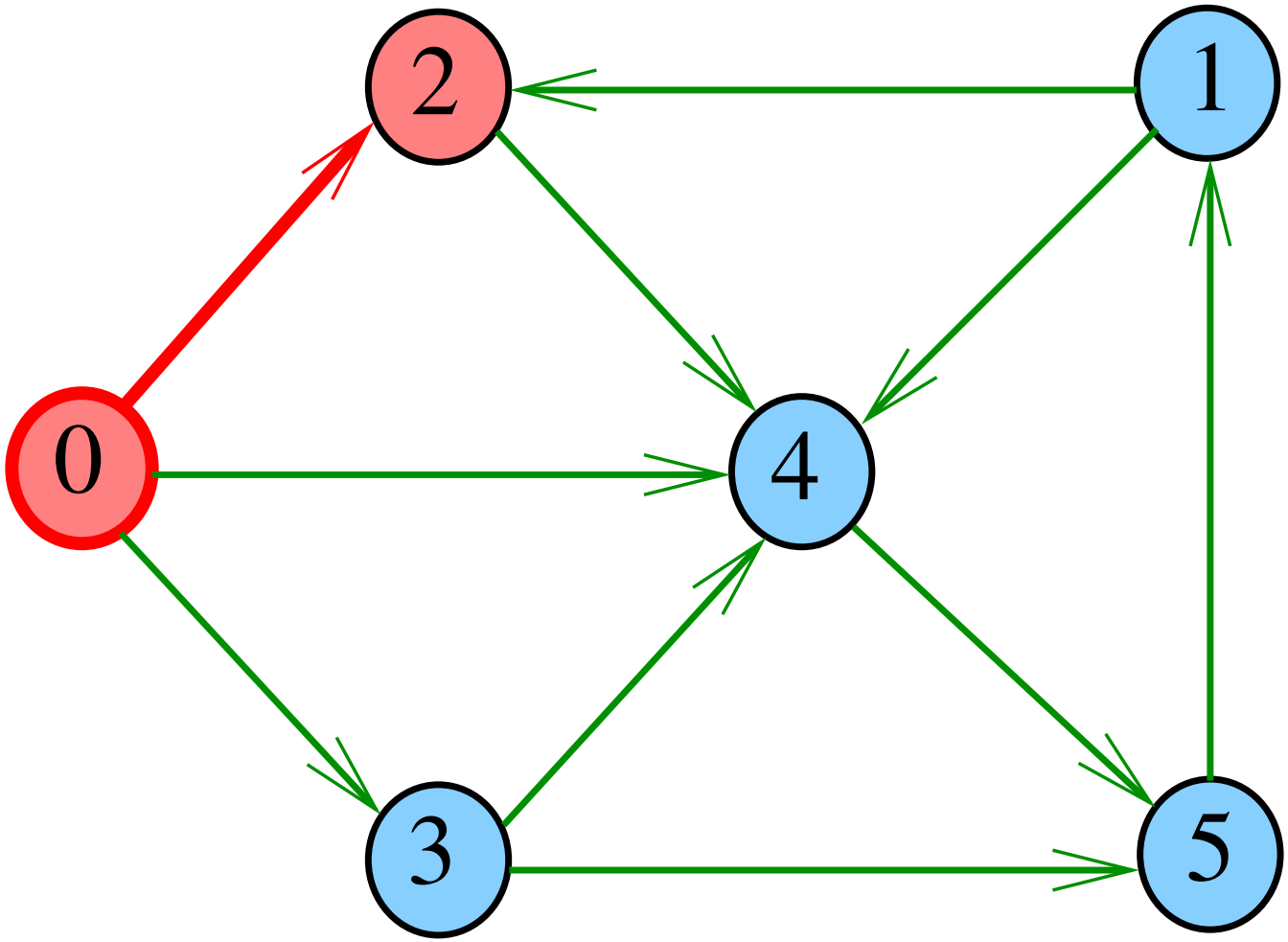
v	0	1	2	3	4	5
dist[v]	**0**	6	6	6	6	6



Simulação

i	0	1	2	3	4	5
q[i]	**0**	2				

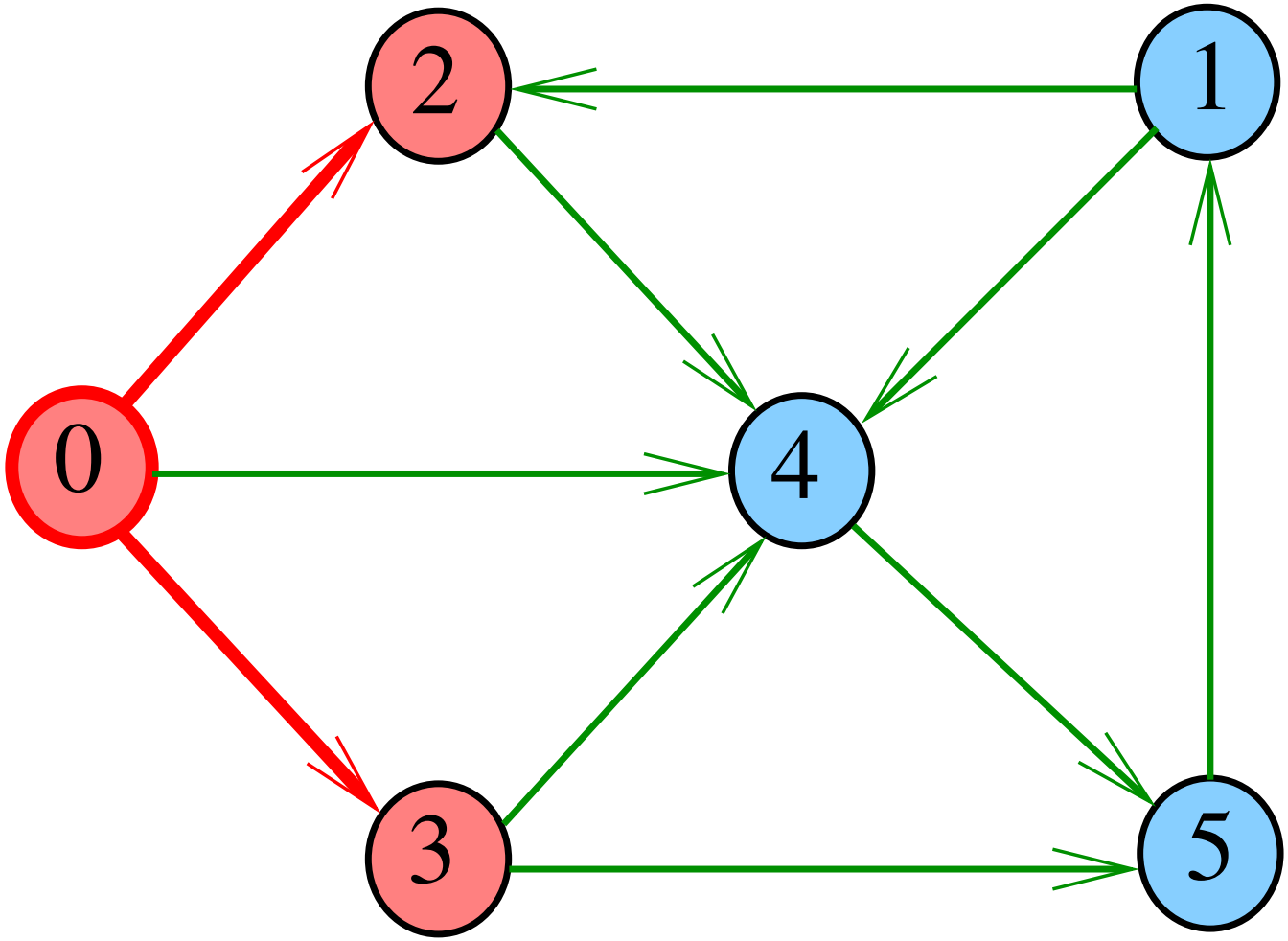
v	0	1	2	3	4	5
dist[v]	0	6	**1**	6	6	6



Simulação

i	0	1	2	3	4	5
q[i]	**0**	2	3			

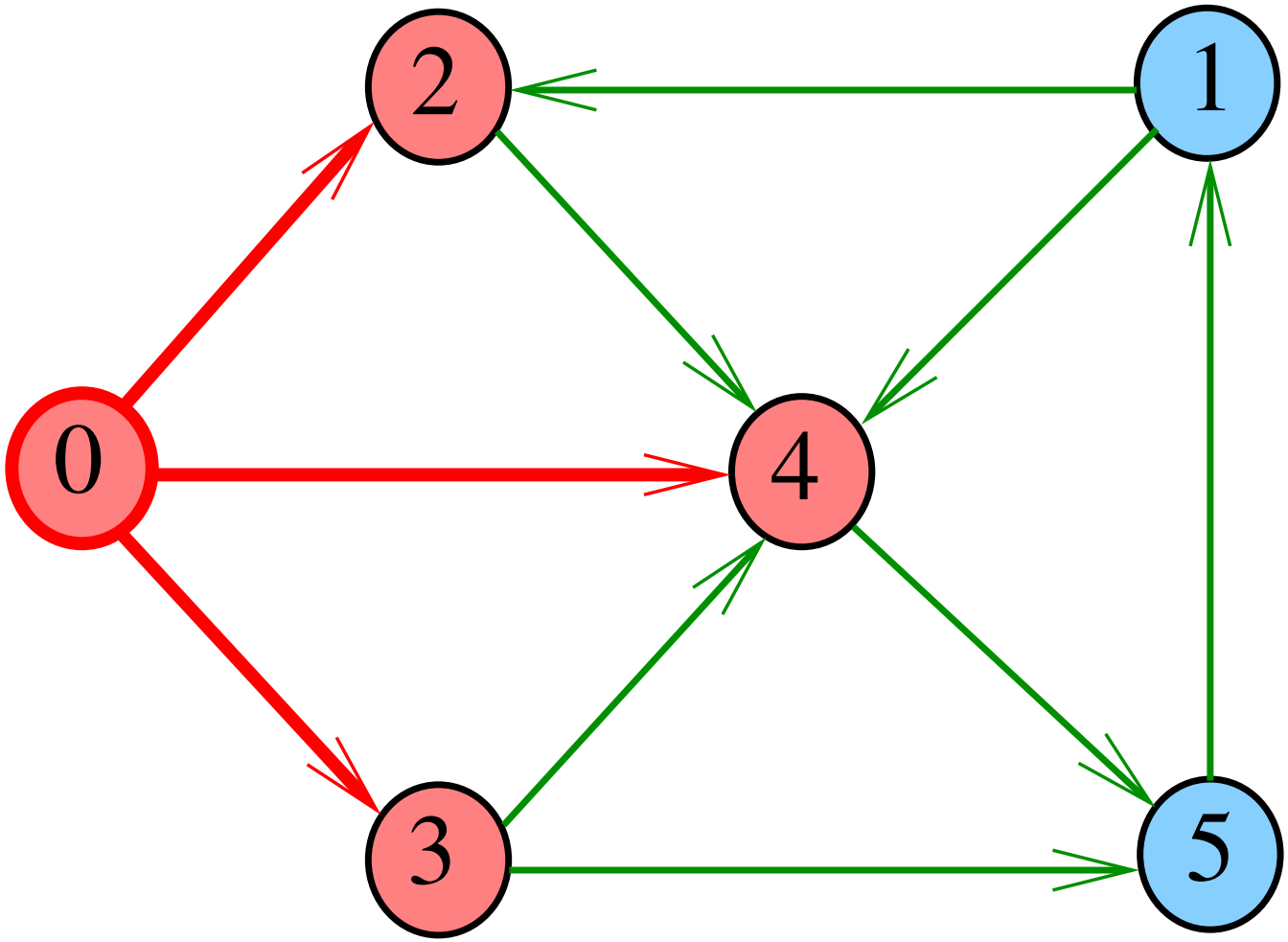
v	0	1	2	3	4	5
dist[v]	0	6	1	**1**	6	6



Simulação

i	0	1	2	3	4	5
q[i]	**0**		**2**	**3**	**4**	

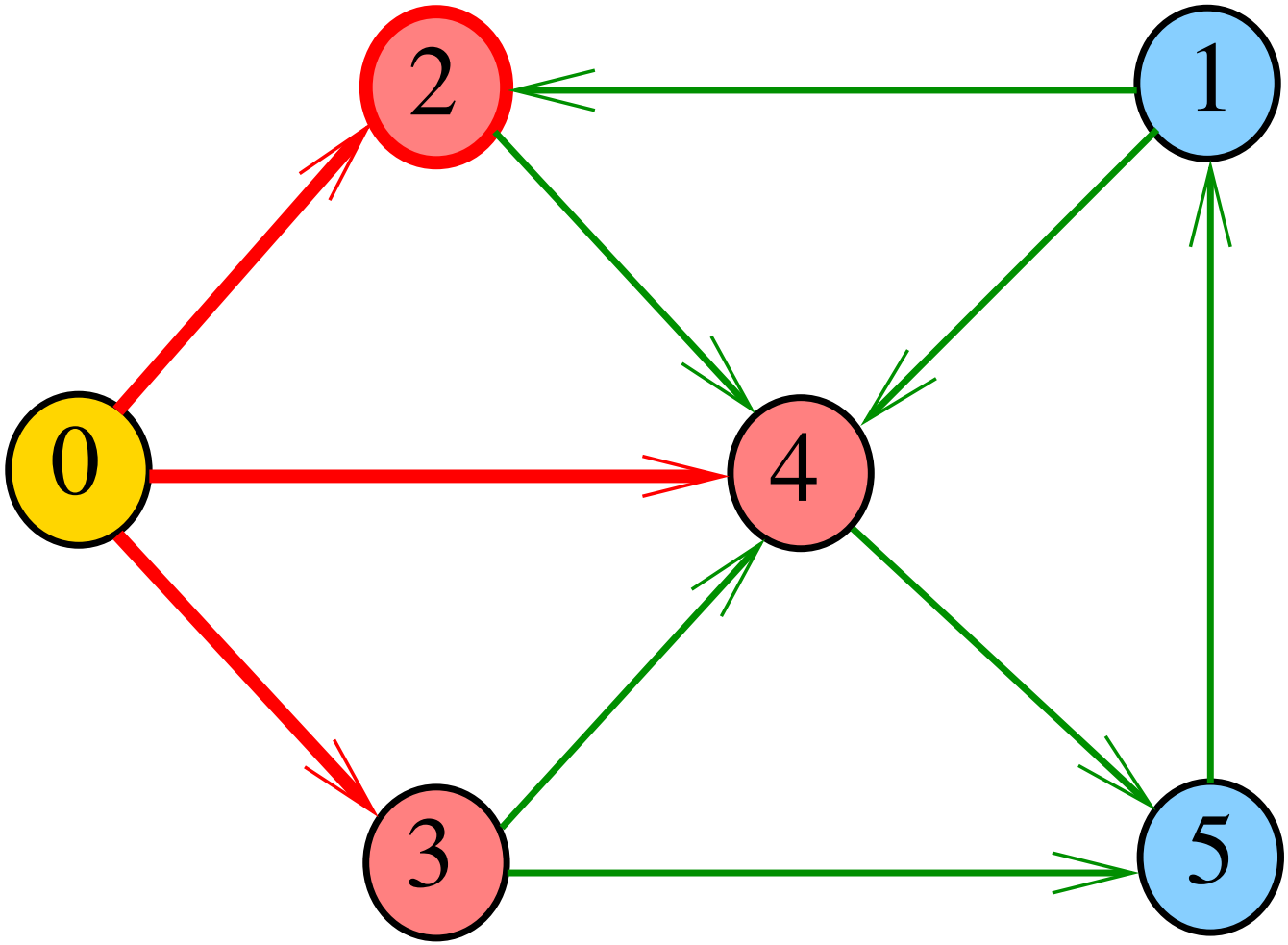
v	0	1	2	3	4	5
dist[v]	0	6	1	1	**1**	6



Simulação

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

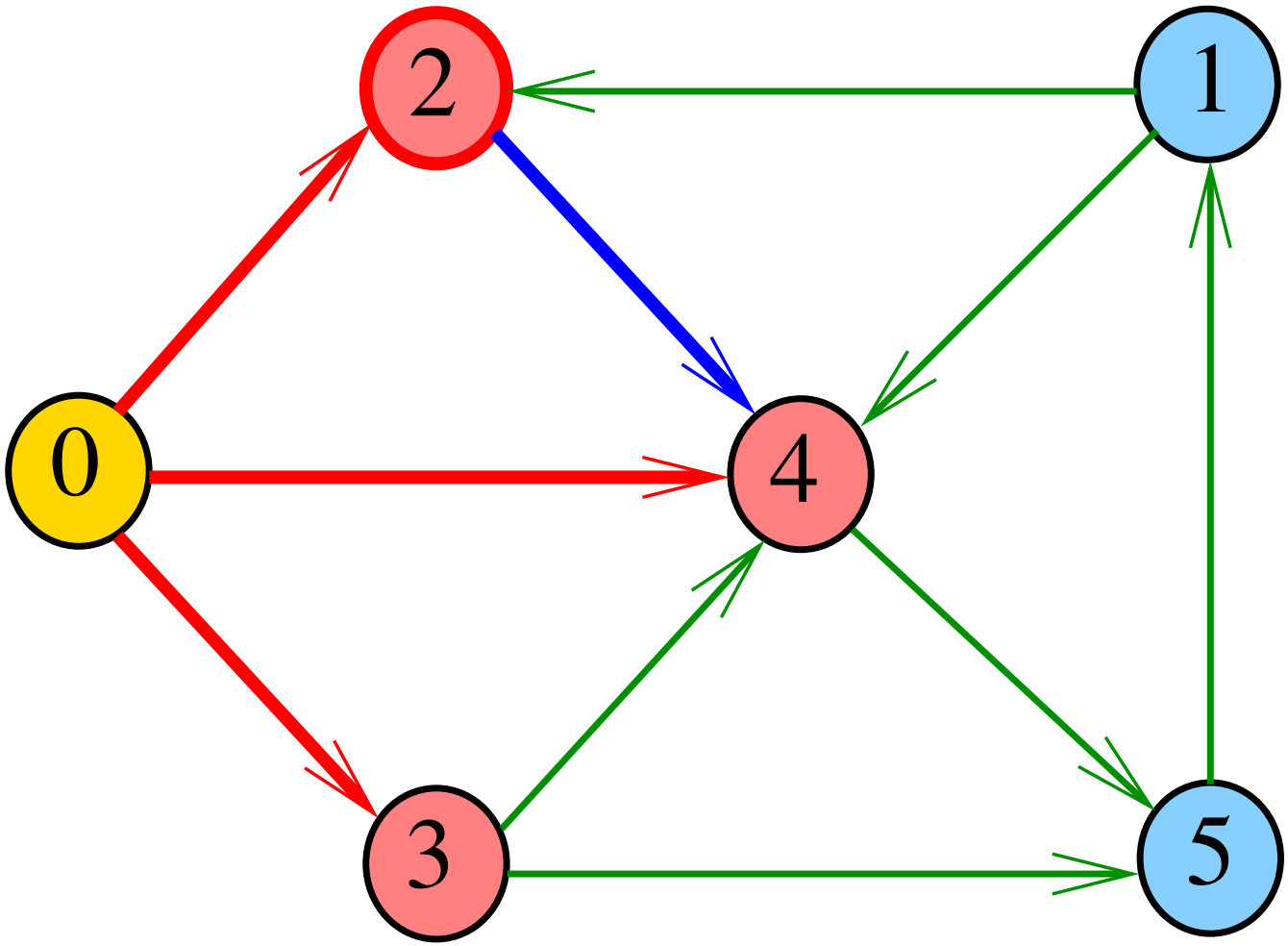
v	0	1	2	3	4	5
dist[v]	0	6	1	1	1	6



Simulação

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

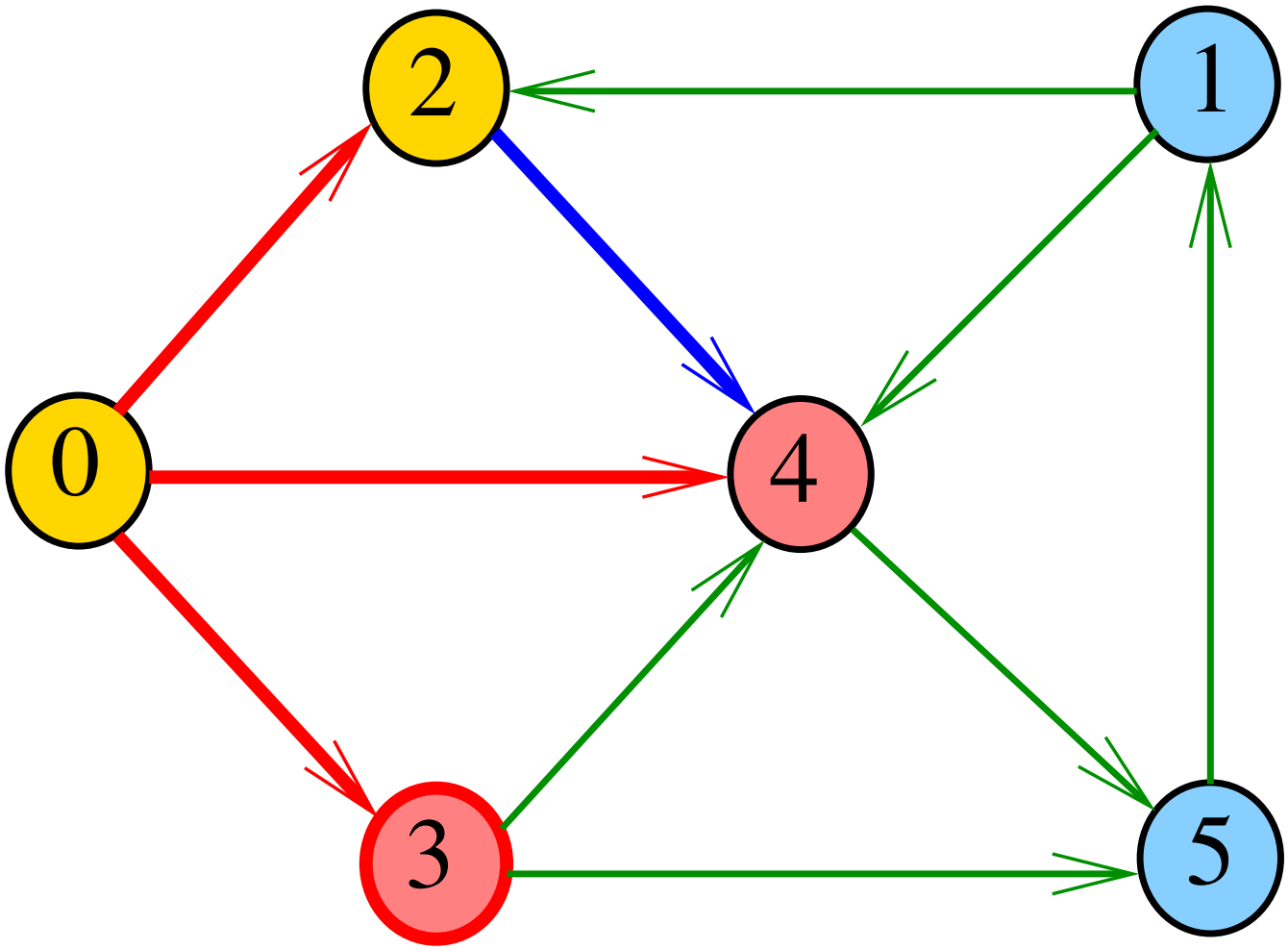
v	0	1	2	3	4	5
dist[v]	0	6	1	1	1	6



Simulação

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

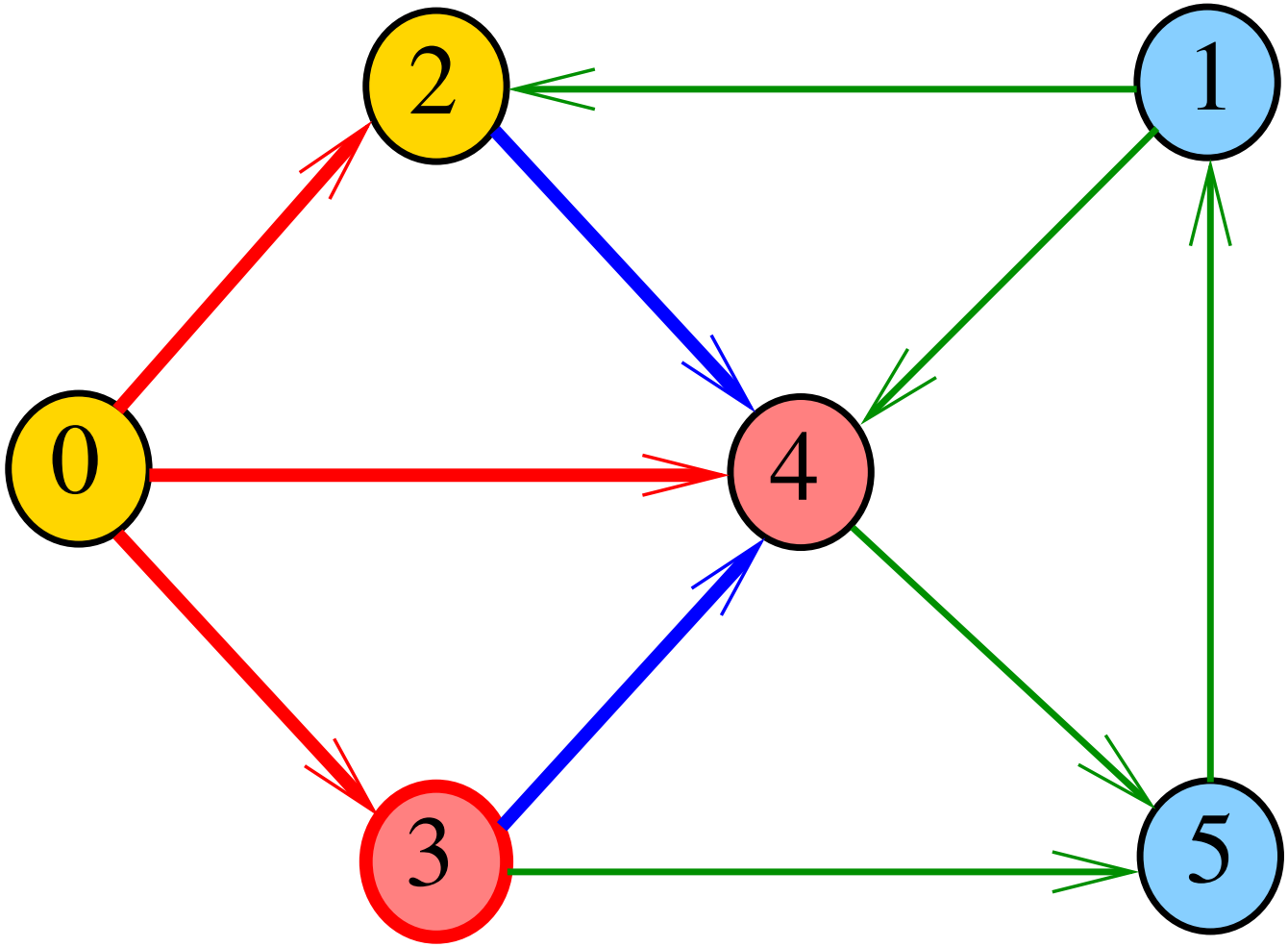
v	0	1	2	3	4	5
dist[v]	0	6	1	1	1	6



Simulação

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

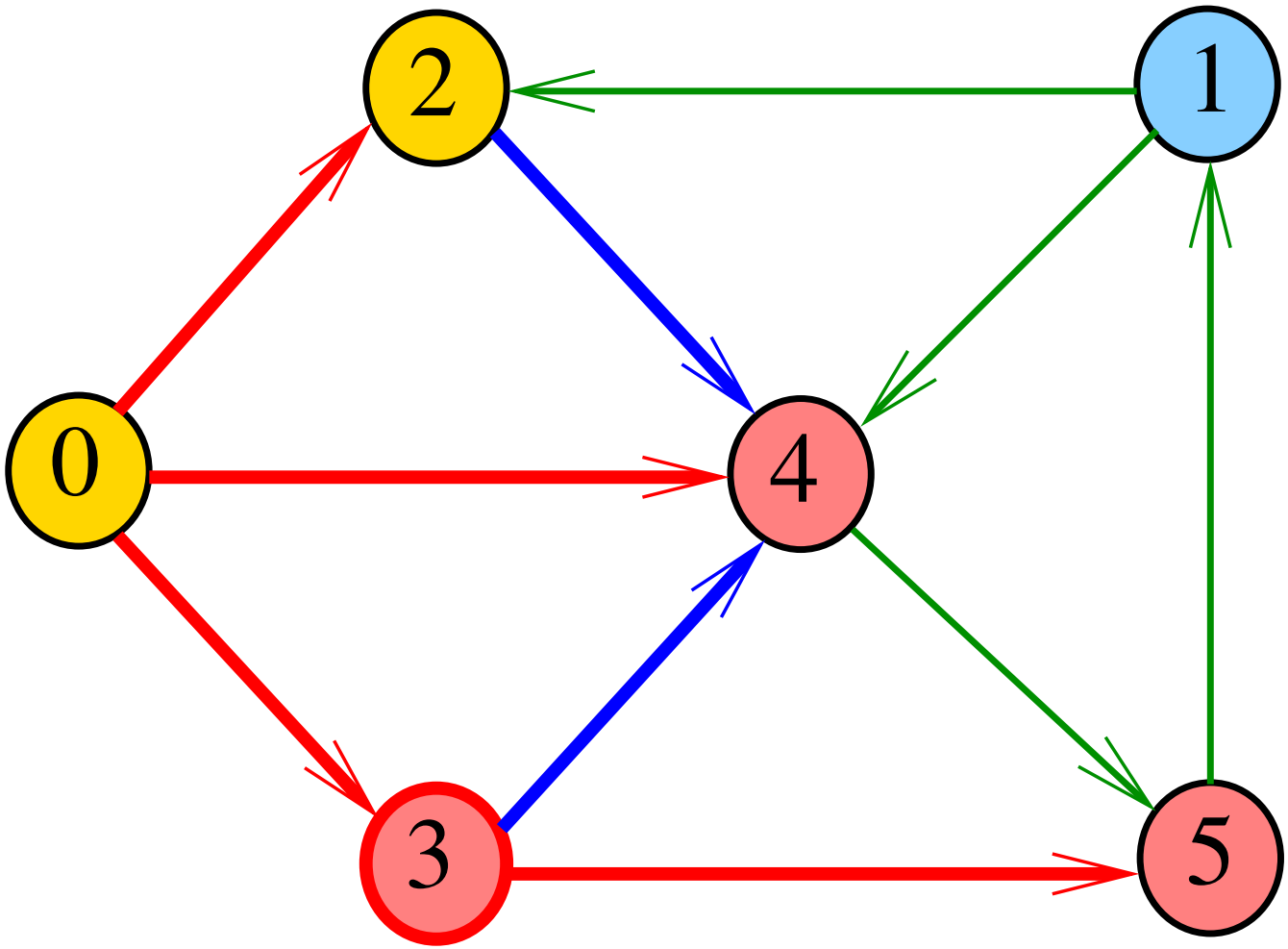
v	0	1	2	3	4	5
dist[v]	0	6	1	1	1	6



Simulação

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

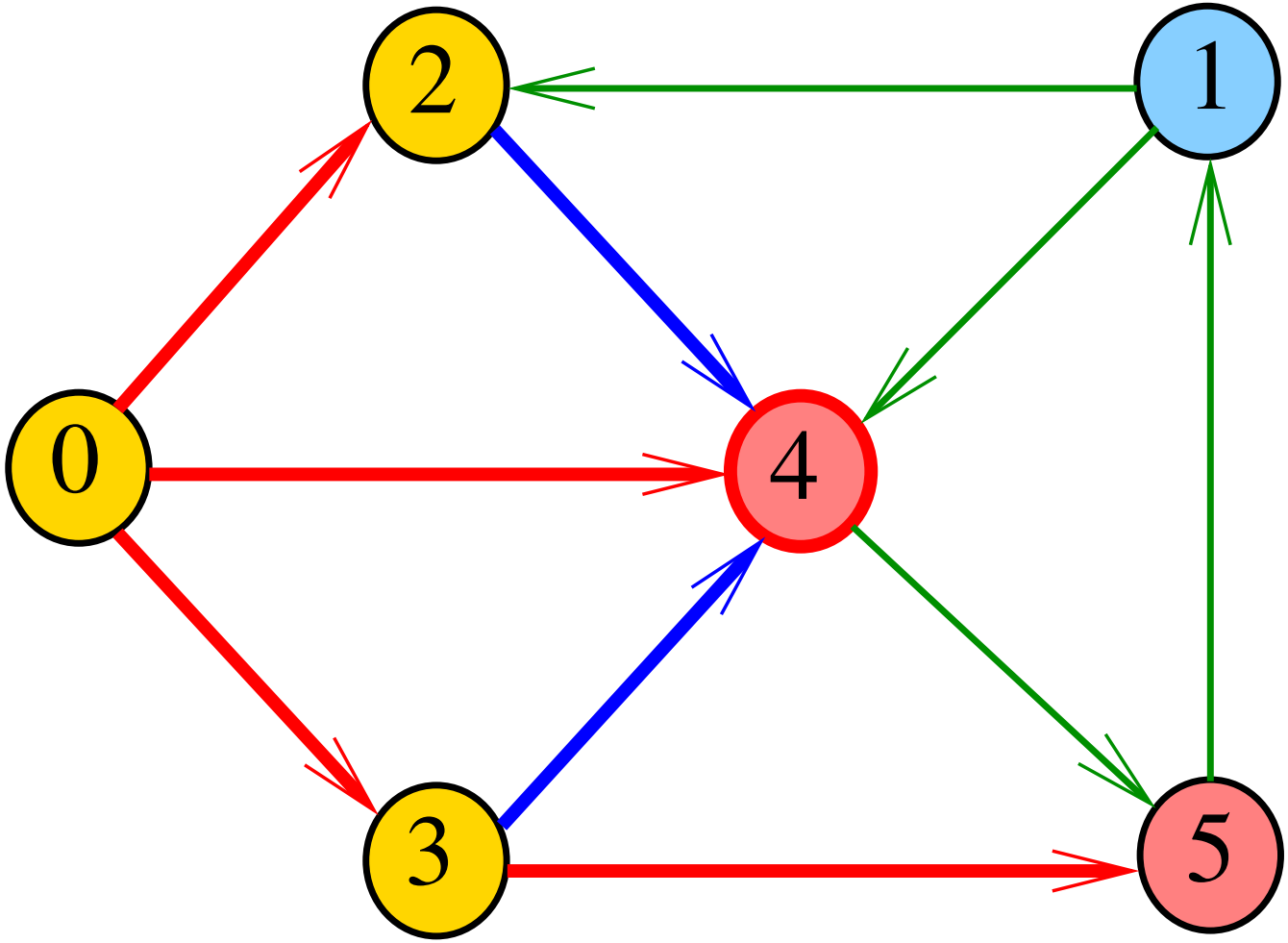
v	0	1	2	3	4	5
dist[v]	0	6	1	1	1	2



Simulação

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

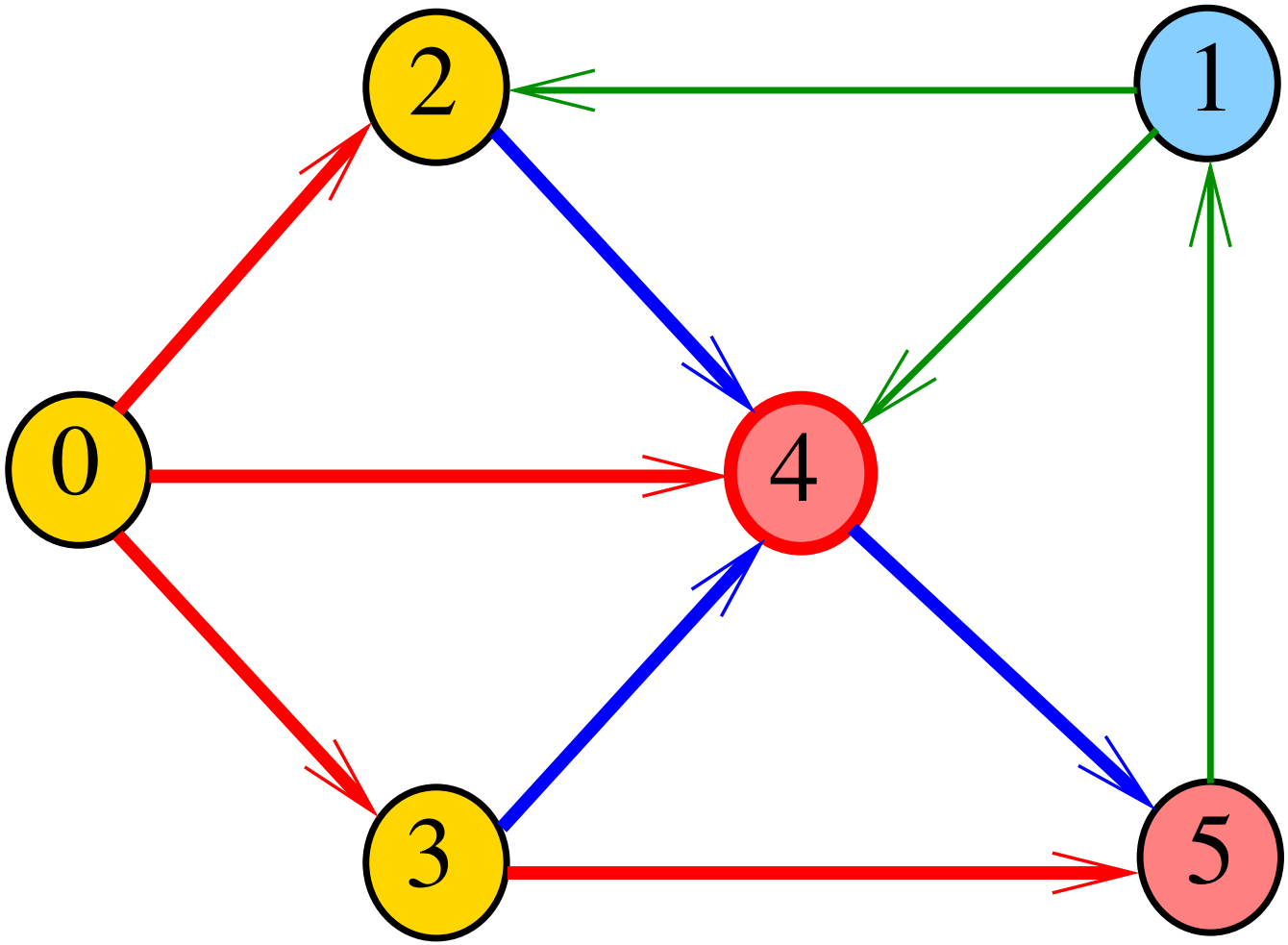
v	0	1	2	3	4	5
dist[v]	0	6	1	1	1	2



Simulação

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

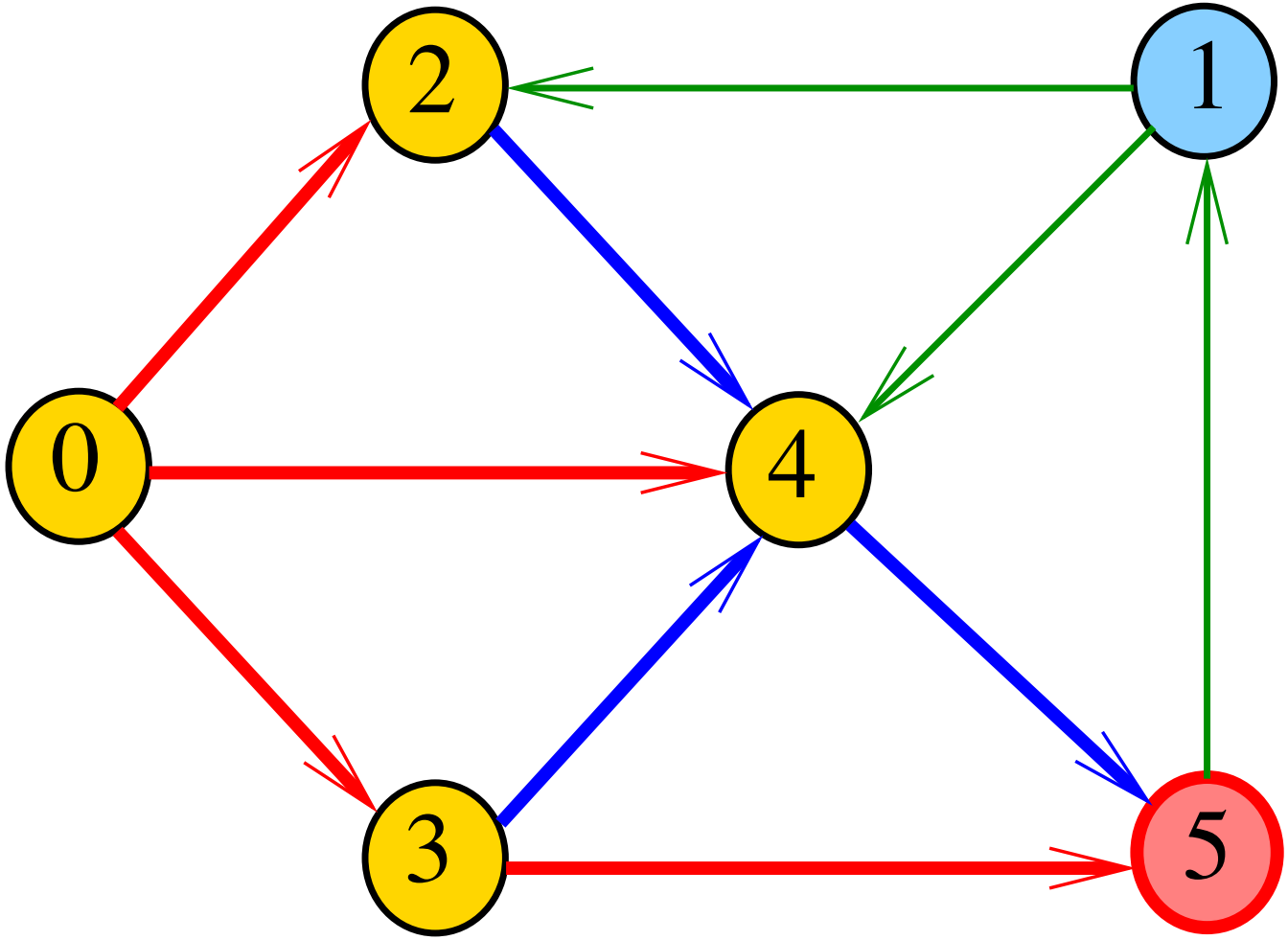
v	0	1	2	3	4	5
dist[v]	0	6	1	1	1	2



Simulação

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

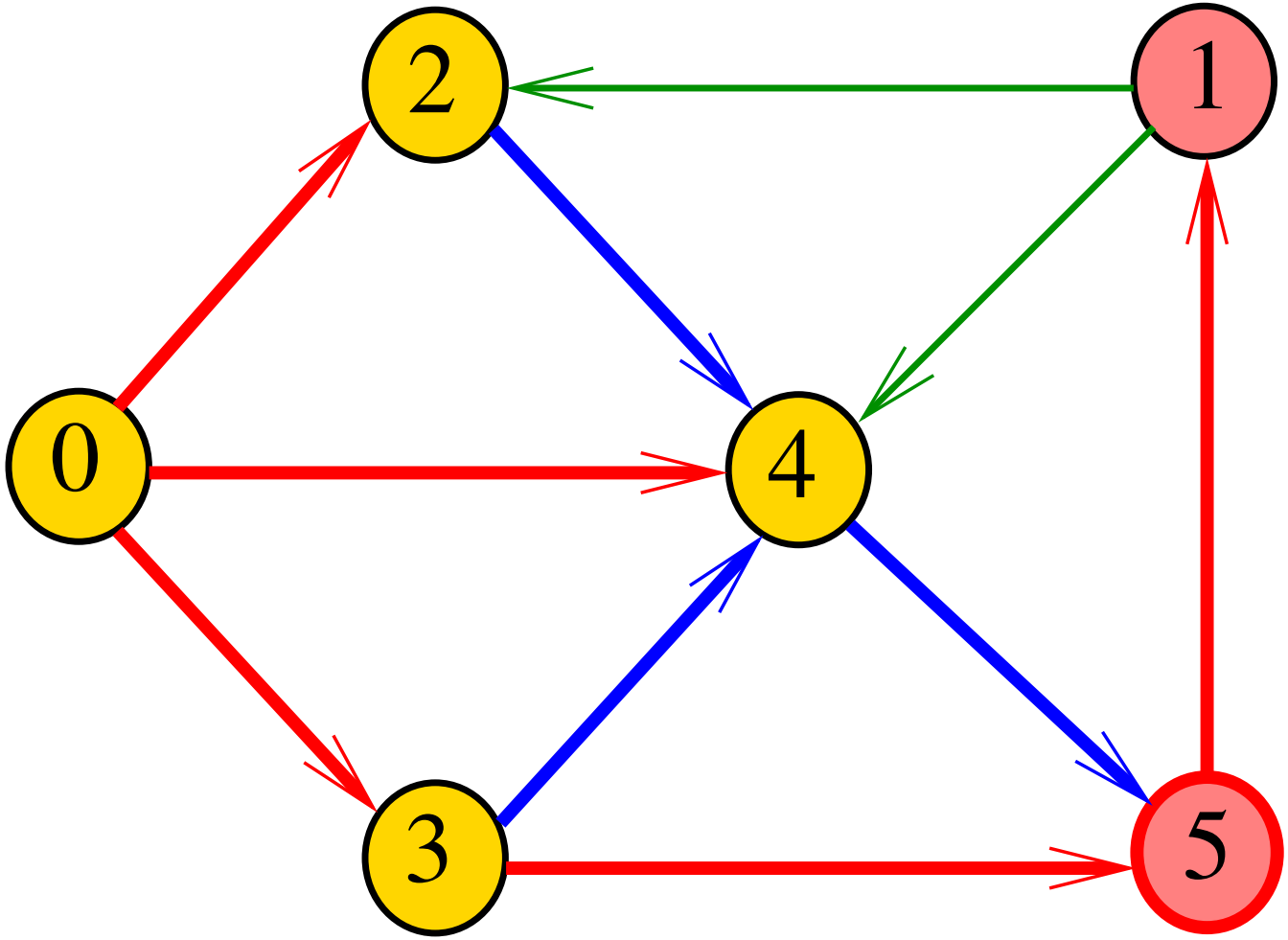
v	0	1	2	3	4	5
dist[v]	0	6	1	1	1	2



Simulação

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

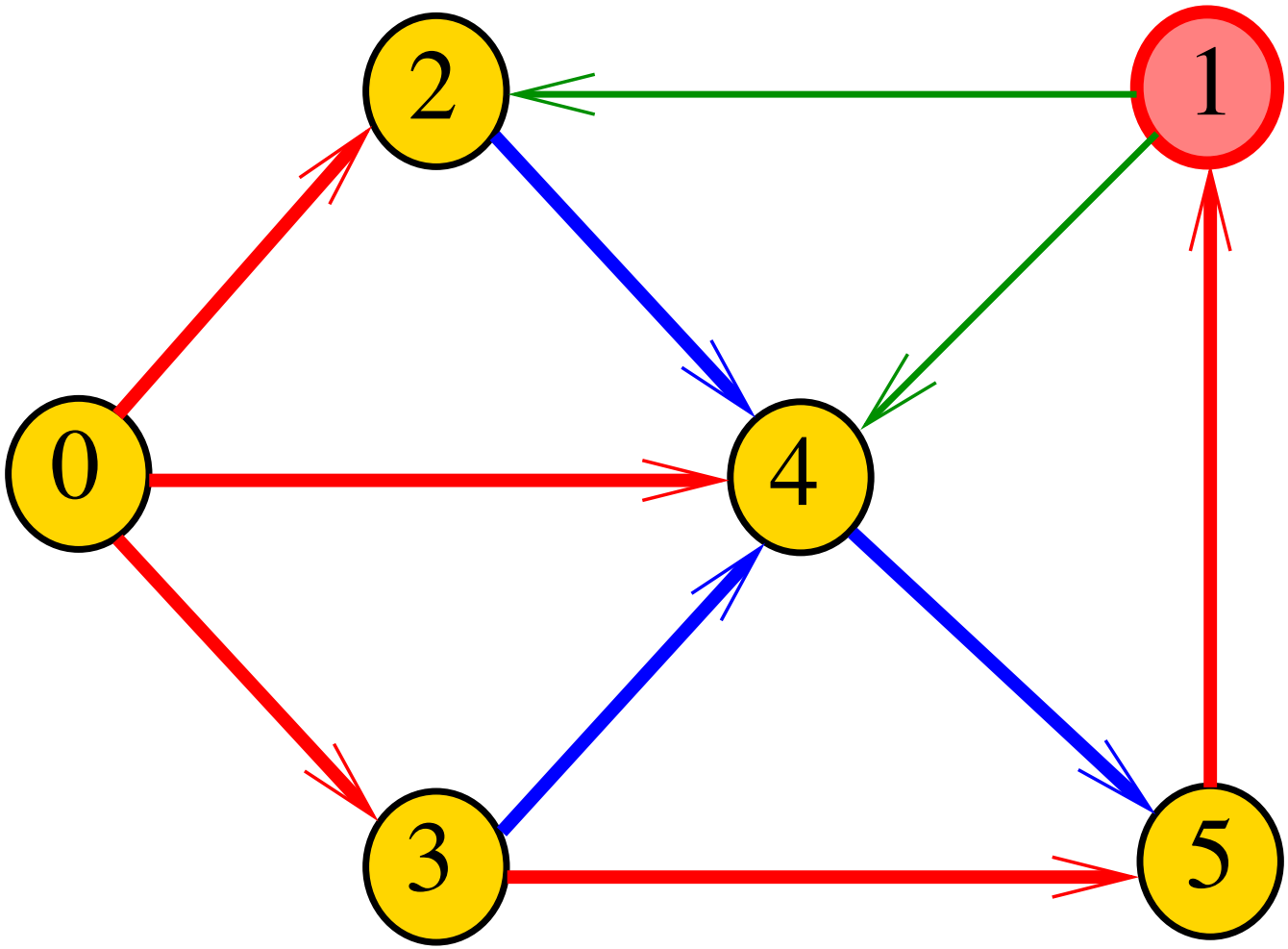
v	0	1	2	3	4	5
dist[v]	0	3	1	1	1	2



Simulação

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

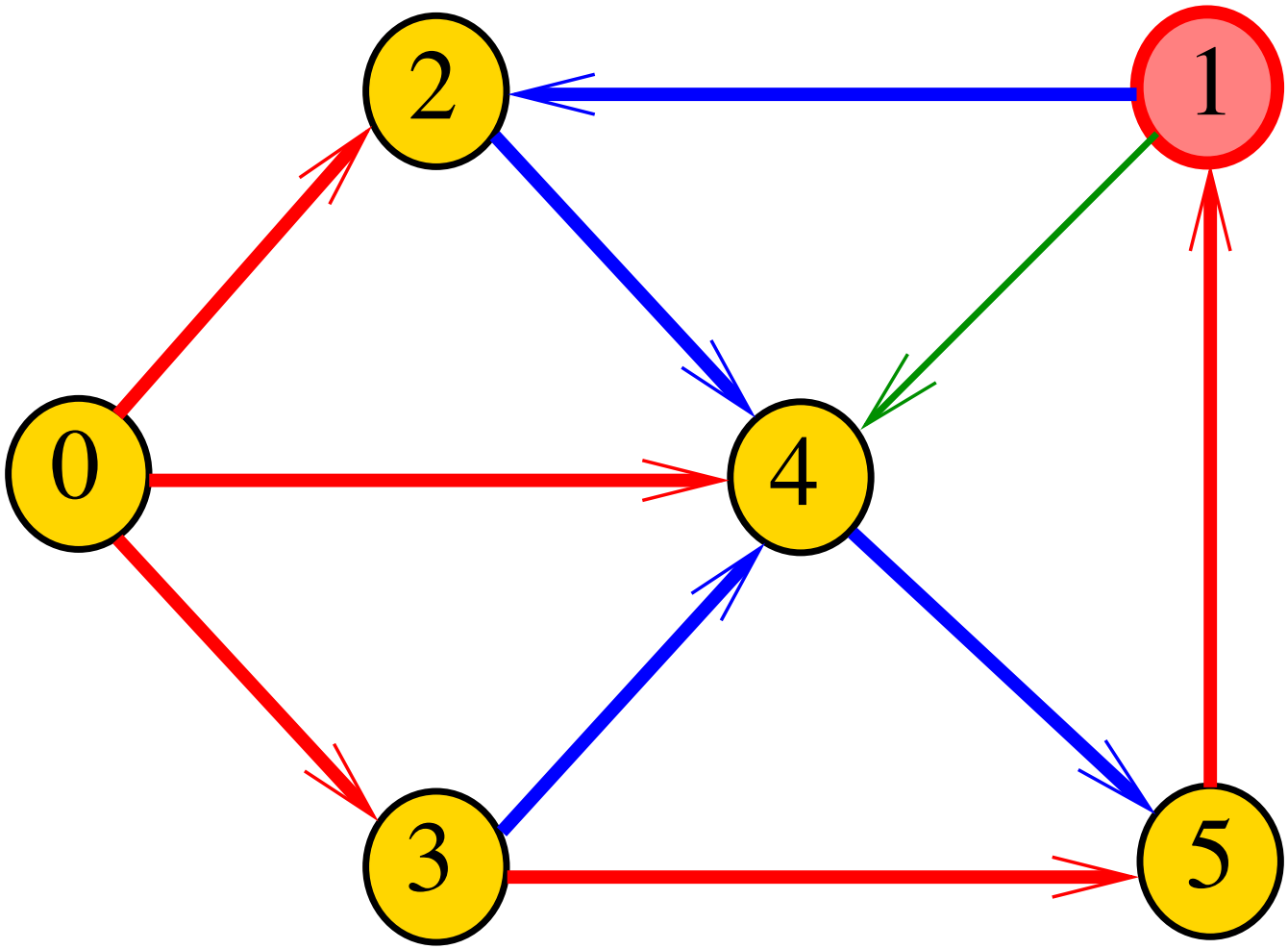
v	0	1	2	3	4	5
dist[v]	0	3	1	1	1	2



Simulação

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

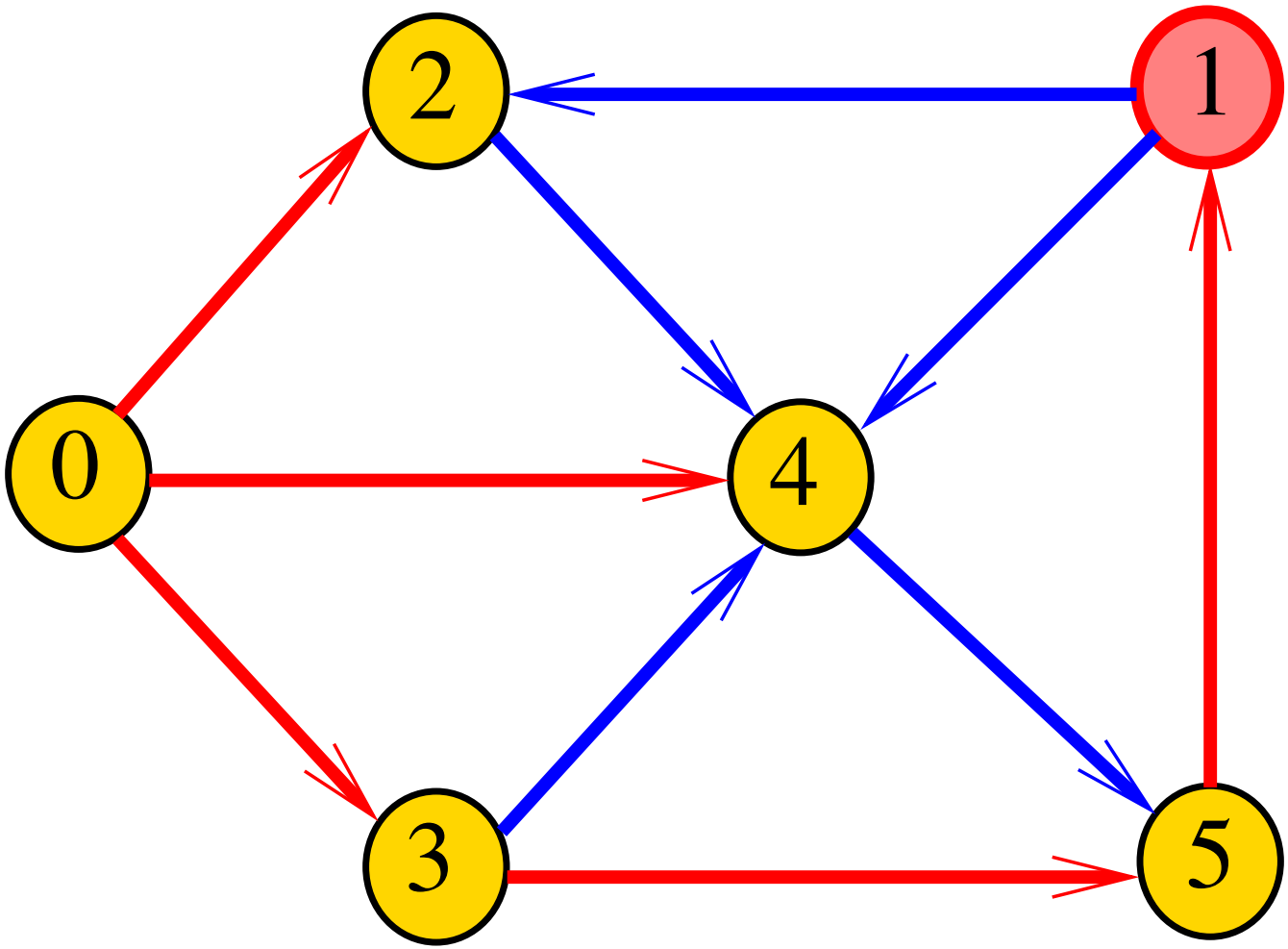
v	0	1	2	3	4	5
dist[v]	0	3	1	1	1	2



Simulação

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

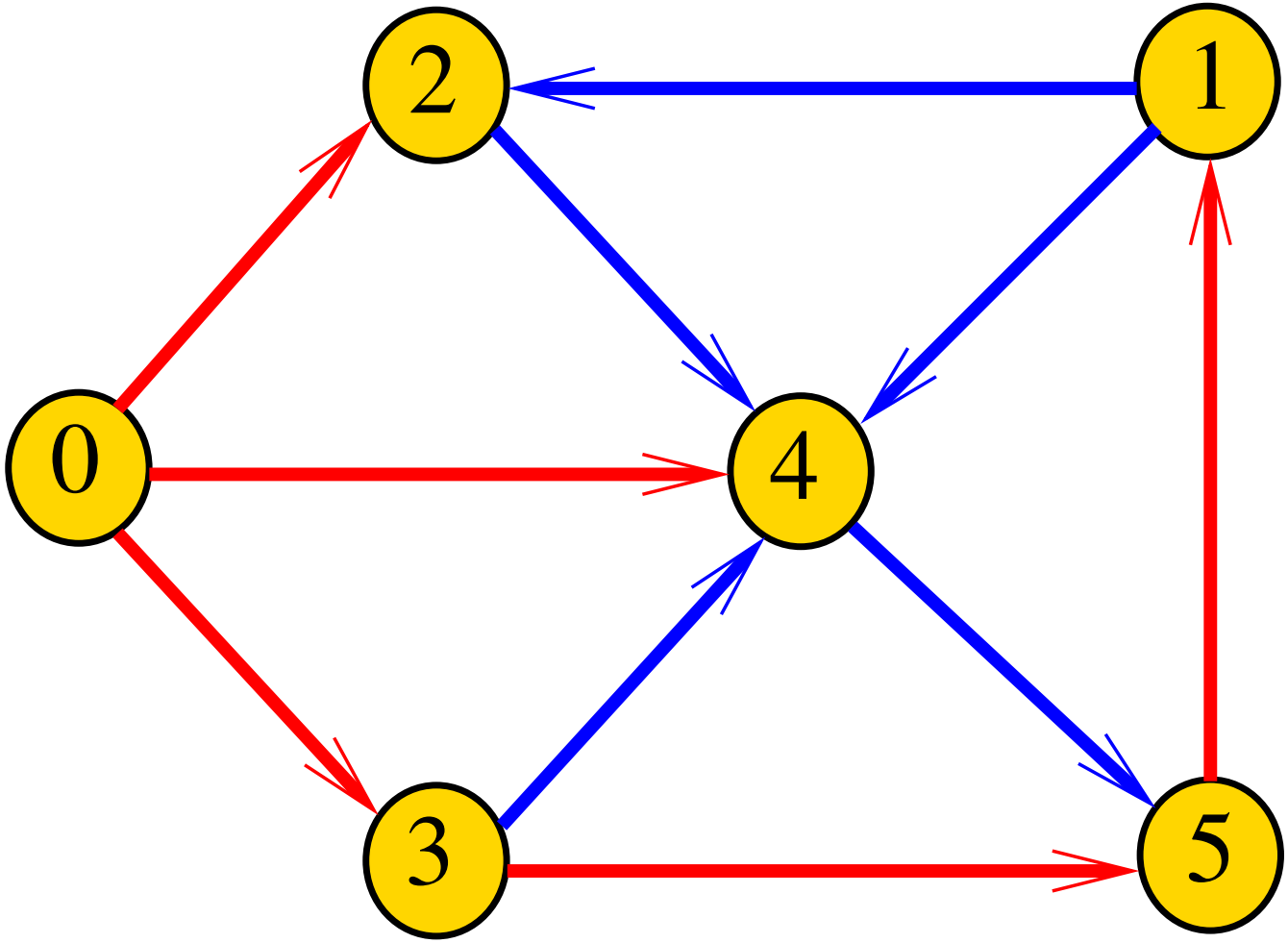
v	0	1	2	3	4	5
dist[v]	0	3	1	1	1	2



Simulação

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

v	0	1	2	3	4	5
dist[v]	0	3	1	1	1	2



DIGRAPHdist

```
#define INFINITO maxV
static int dist[maxV];
static Vertex parnt[maxV];
void DIGRAPHbfs (Digraph G, Vertex s) {
1   Vertex v, w; link p;
2   for (v = 0; v < G->V; v++)
3       dist[v] = INFINITO;
4       parnt[v] = -1;
    }
5   QUEUEinit(G->V);
6   dist[s] = 0;
7   parnt[s] = s;
```

DIGRAPHdist

```
 8   QUEUEput(s);
 9   while (!QUEUEempty()) {
10       v = QUEUEget();
11       for(p=G->adj[v];p!=NULL;p=p->next)
12           if (dist[w=p->w] == INFINITO) {

13               dist[w] = dist[v] + 1;
14               parnt[w] = v;
15               QUEUEput(w);
             }
     }
16   QUEUEfree();
}
```

Digrafos com custos nos arcos

Muitas aplicações associam um número a cada arco de um digrafo

Diremos que esse número é o custo da arco

Vamos supor que esses números são do tipo **double**

```
typedef struct {
    Vertex v;
    Vertex w;
    double cst;
} Arc;
```

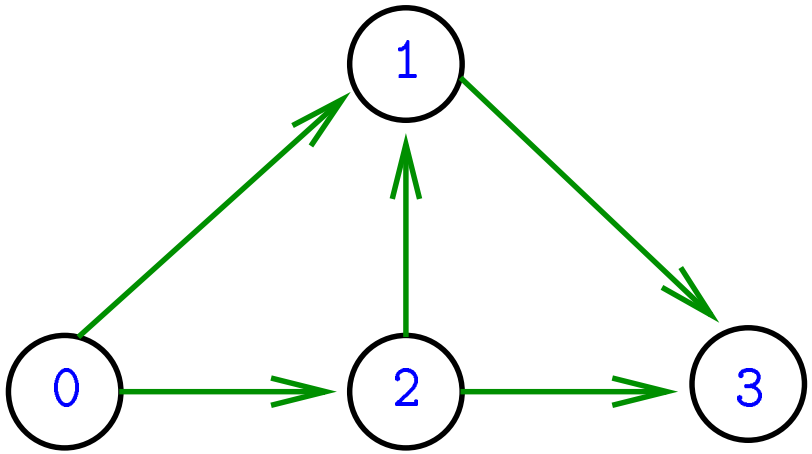
Matriz de adjacência

Matriz de adjacência indica a presença ausência e custo dos arcos:

se `v`-`w` é um arco, `adj[v][w]` é seu custo

se `v`-`w` não é arco, `adj[v][w] = maxCST`

Exemplo:

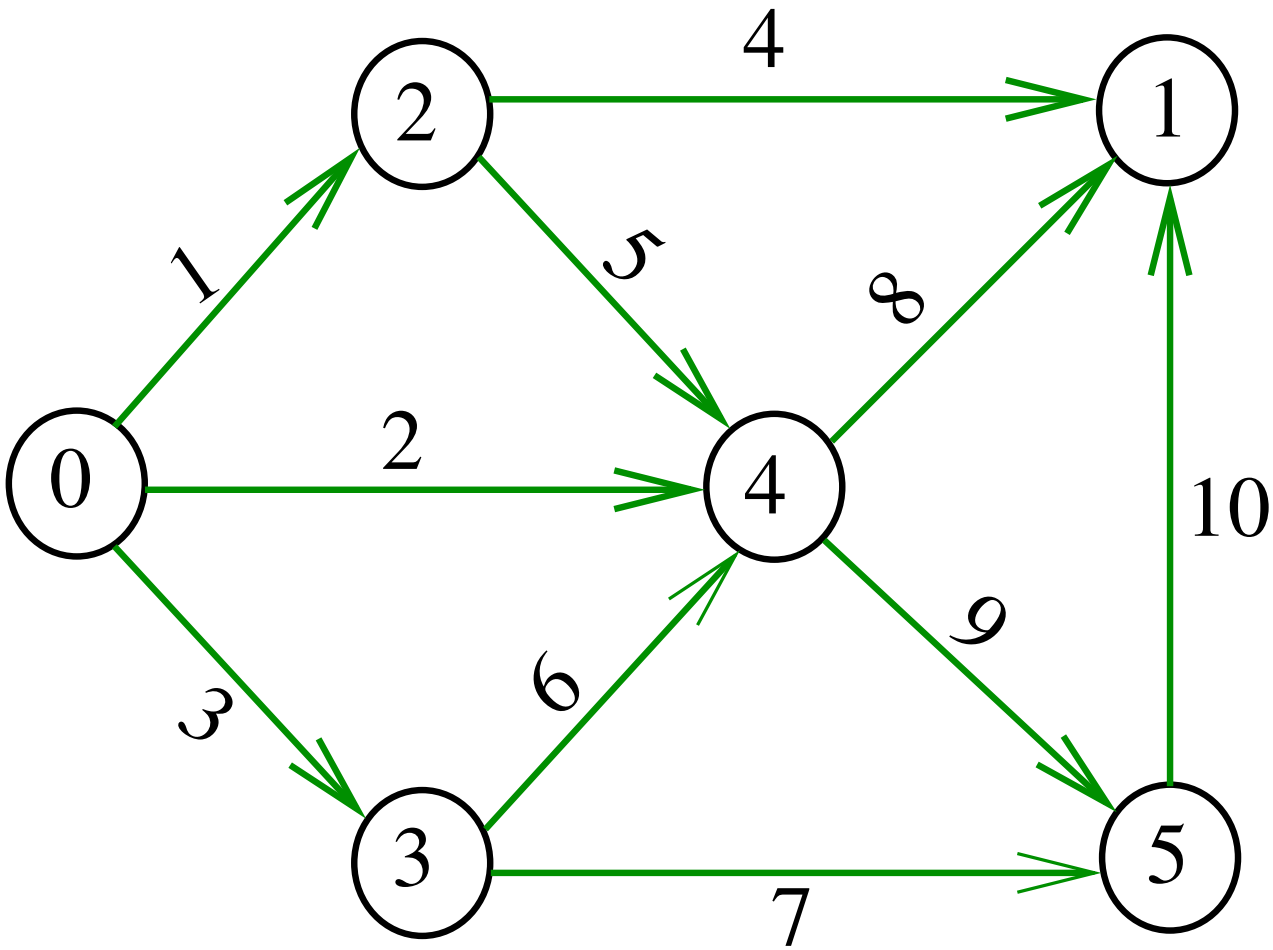


	0	1	2	3
0	*	.3	2	*
1	*	*	*	.42
2	*	1	*	.12
3	*	*	*	*

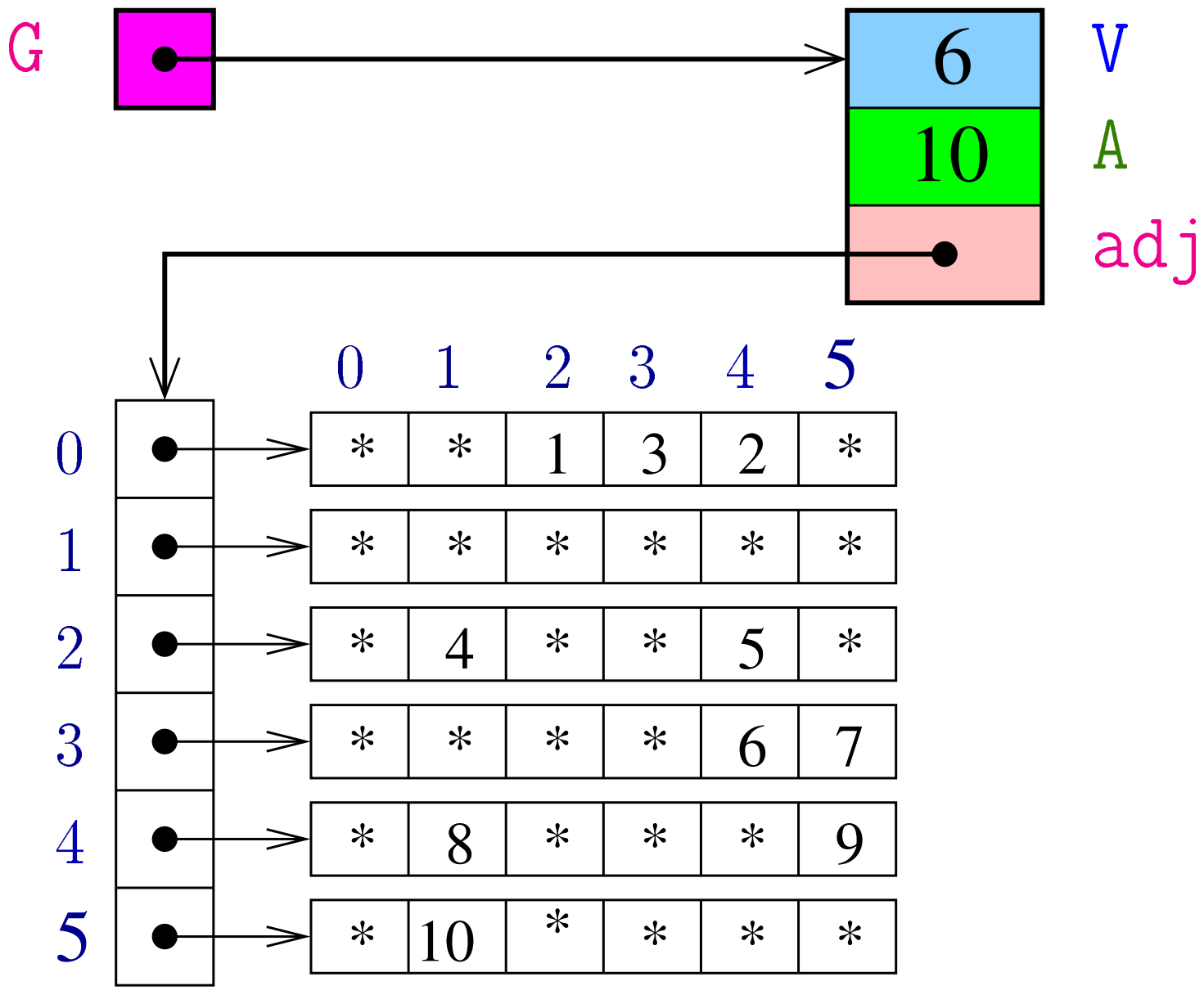
* indica `maxCST` = `INFINITO`

Digrafo

Digraph G



Estruturas de dados



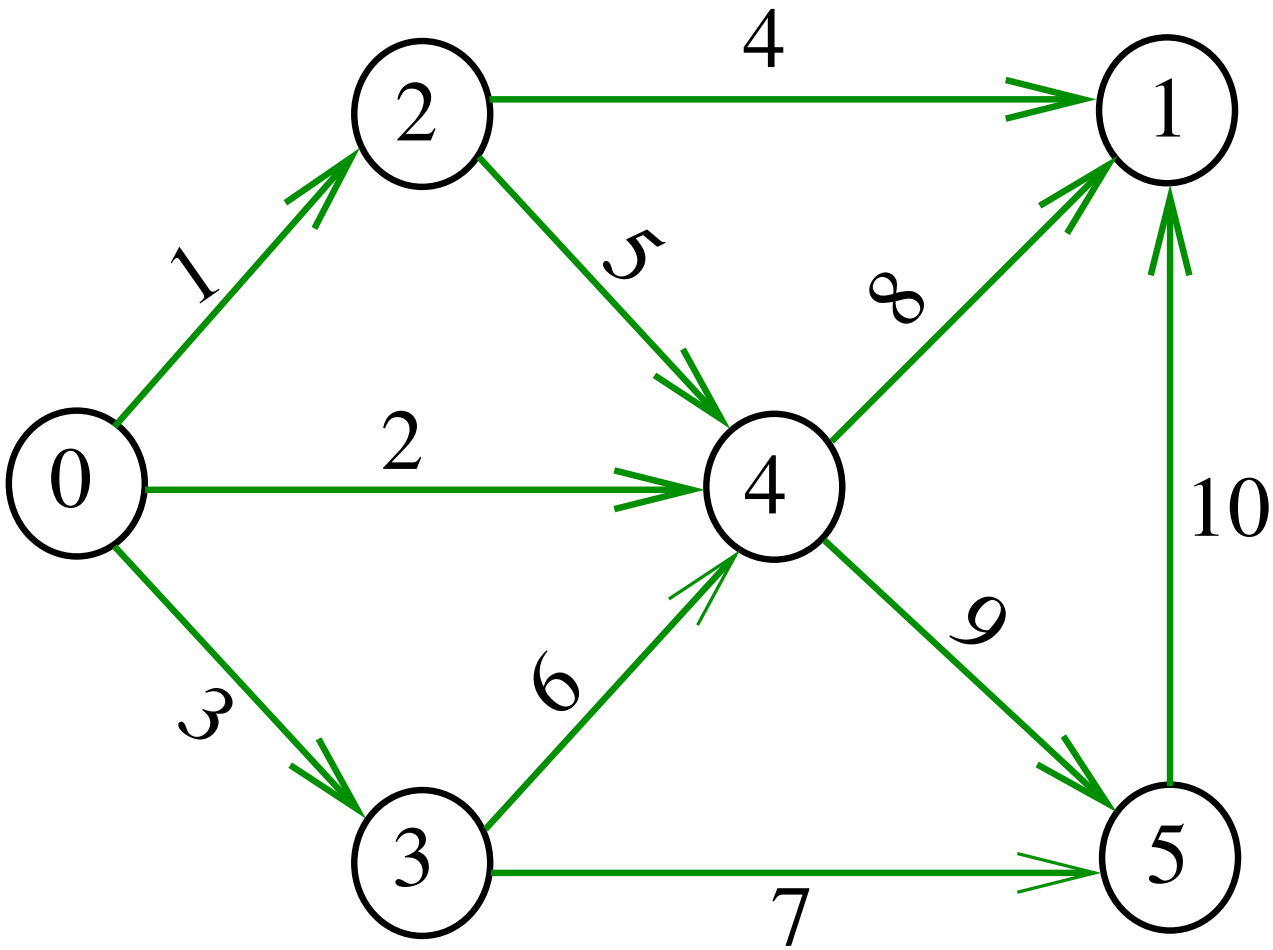
Vetor de listas de adjacência

A lista de adjacência de um vértice `v` é composta por nós do tipo `node`
Um `link` é um ponteiro para um `node`
Cada nó da lista contém um vizinho `w` de `v`, **o custo** do arco `v-w` e o endereço do nó seguinte da lista

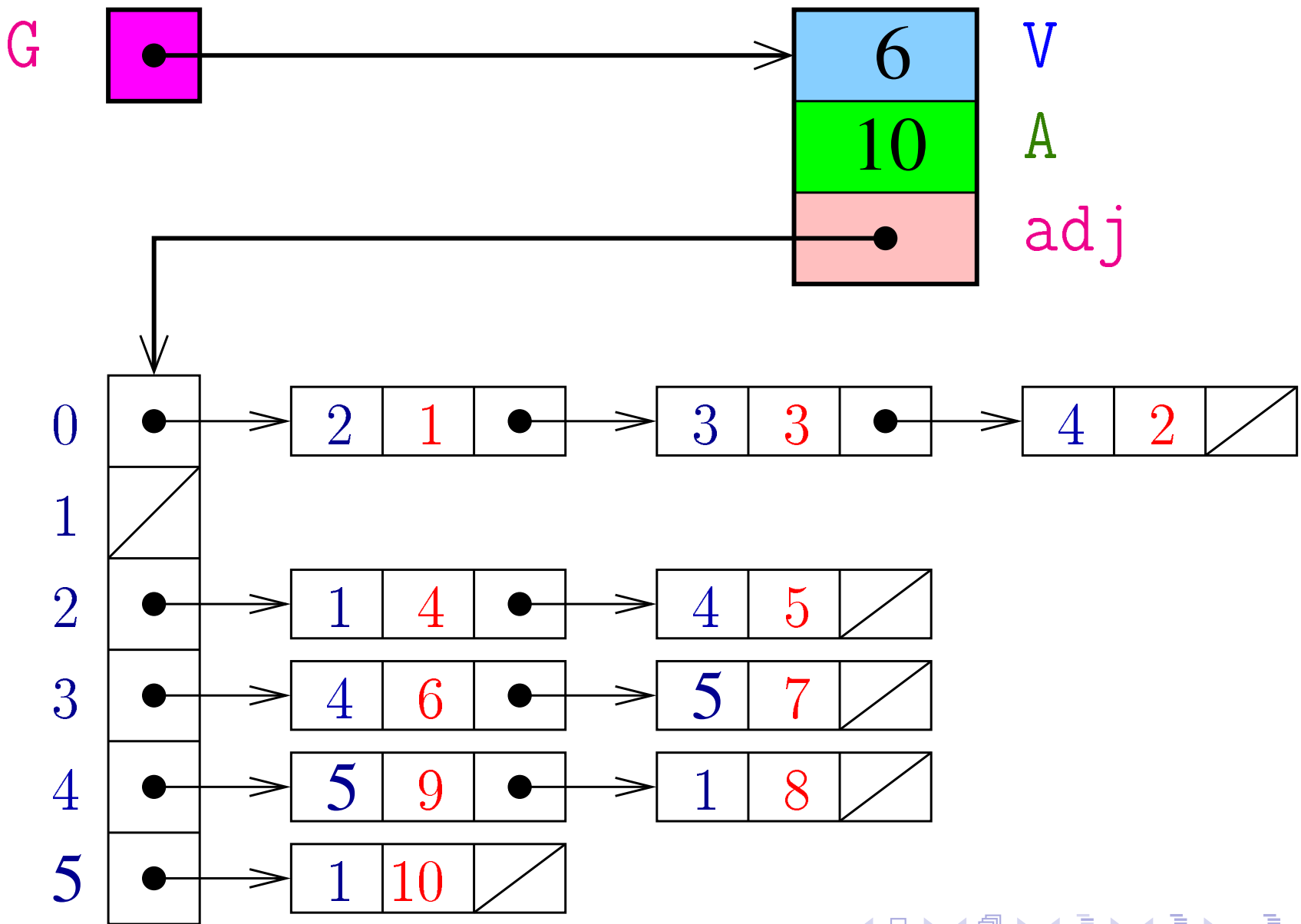
```
typedef struct node *link;
struct node {
    Vertex w;
    double cst;
    link next;
};
```

Digrafo

Digraph G



Estruturas de dados



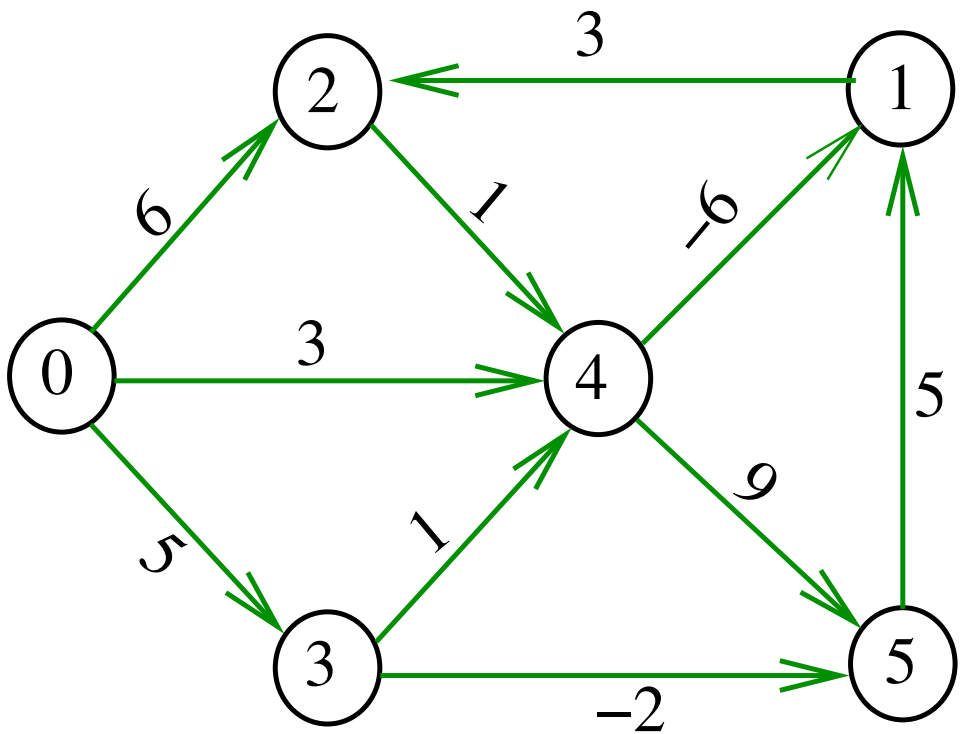
Custo de um caminho

Custo de um caminho é 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.

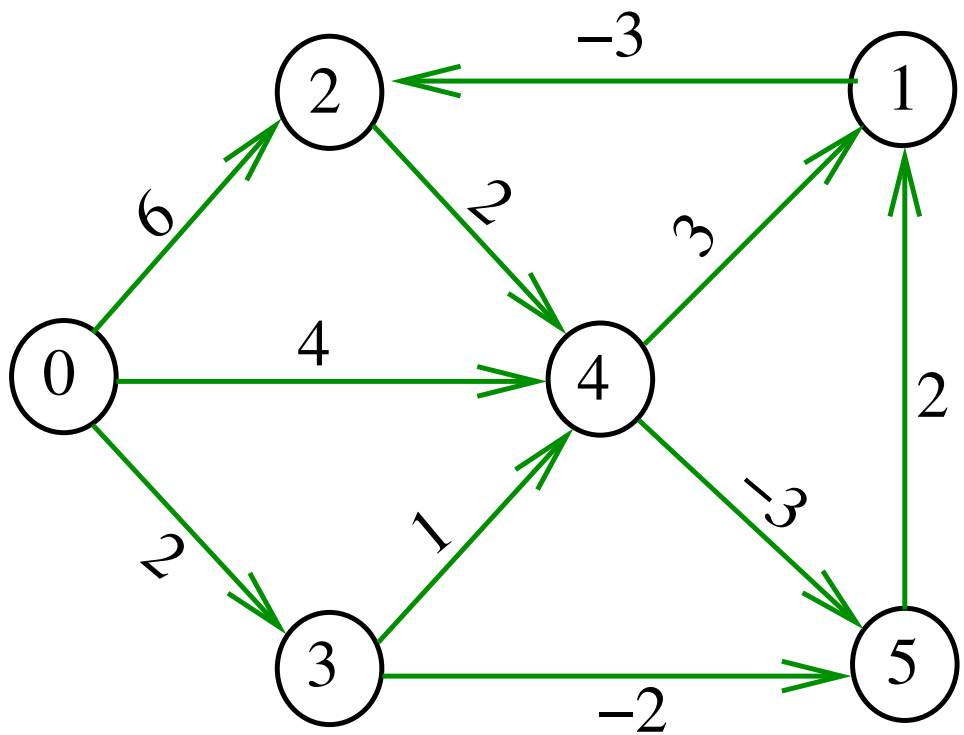
Custo do caminho 0-2-4-1-2-4-1-2-4-5 é 12.



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



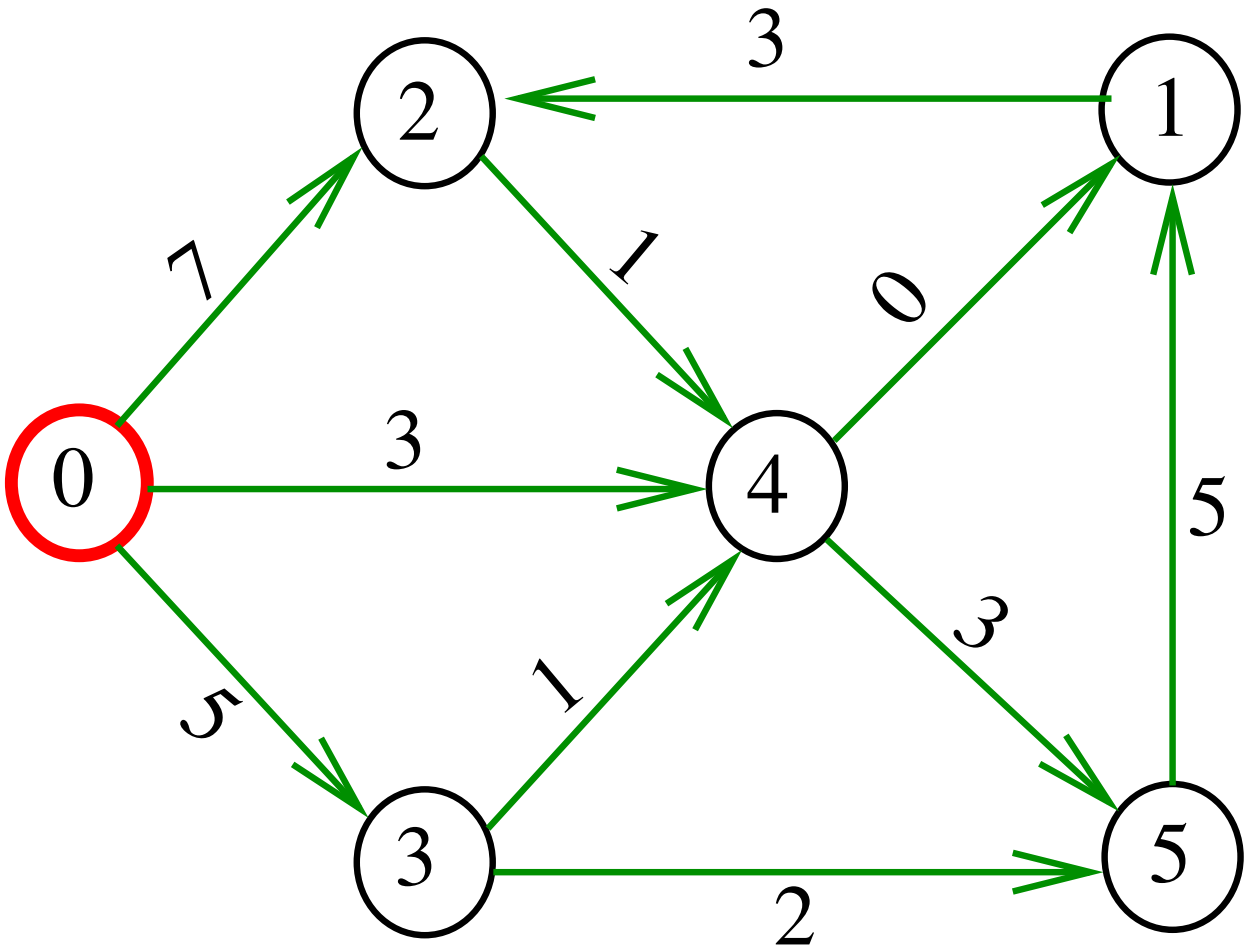
Problema

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.*

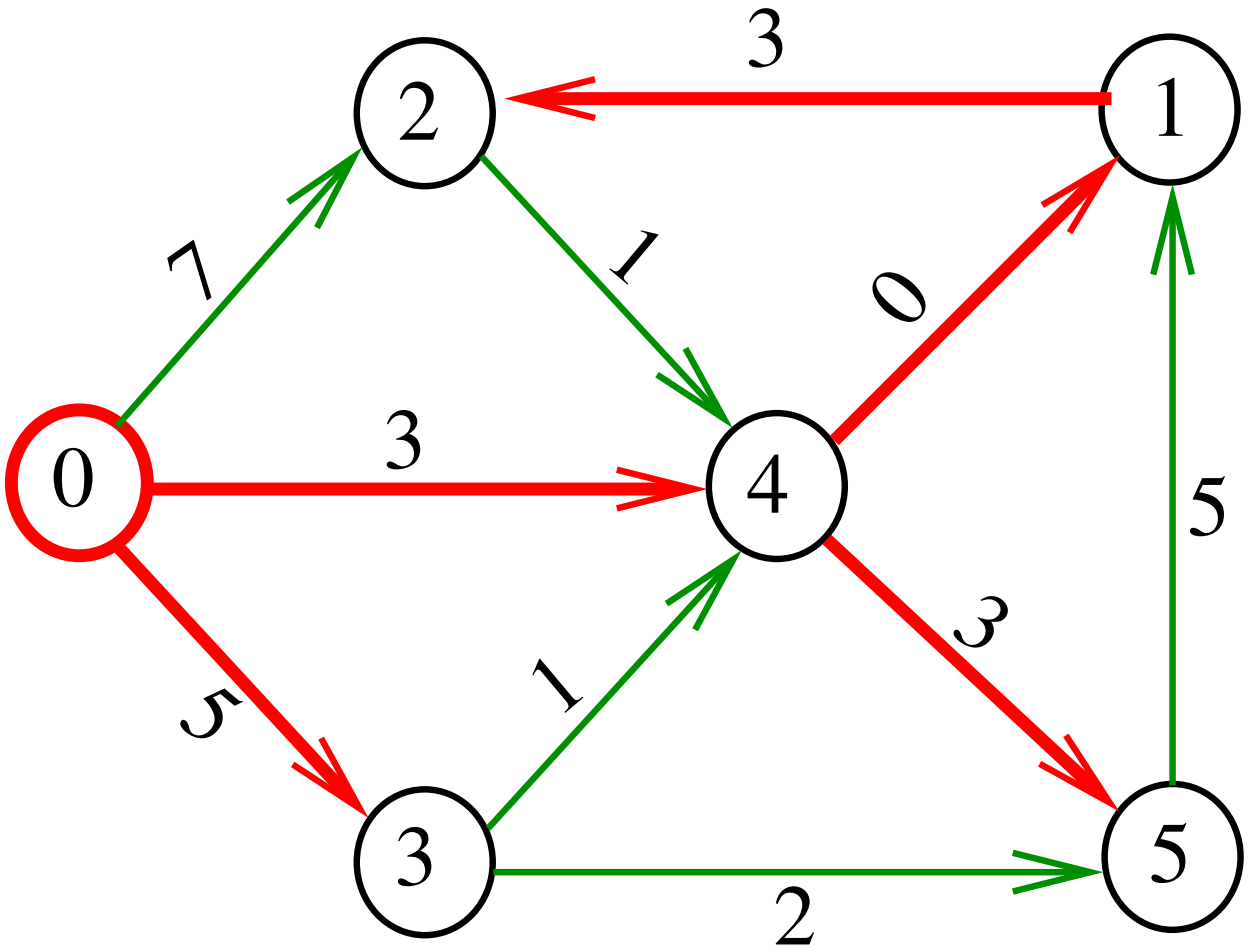
Exemplo

Entra:



Exemplo

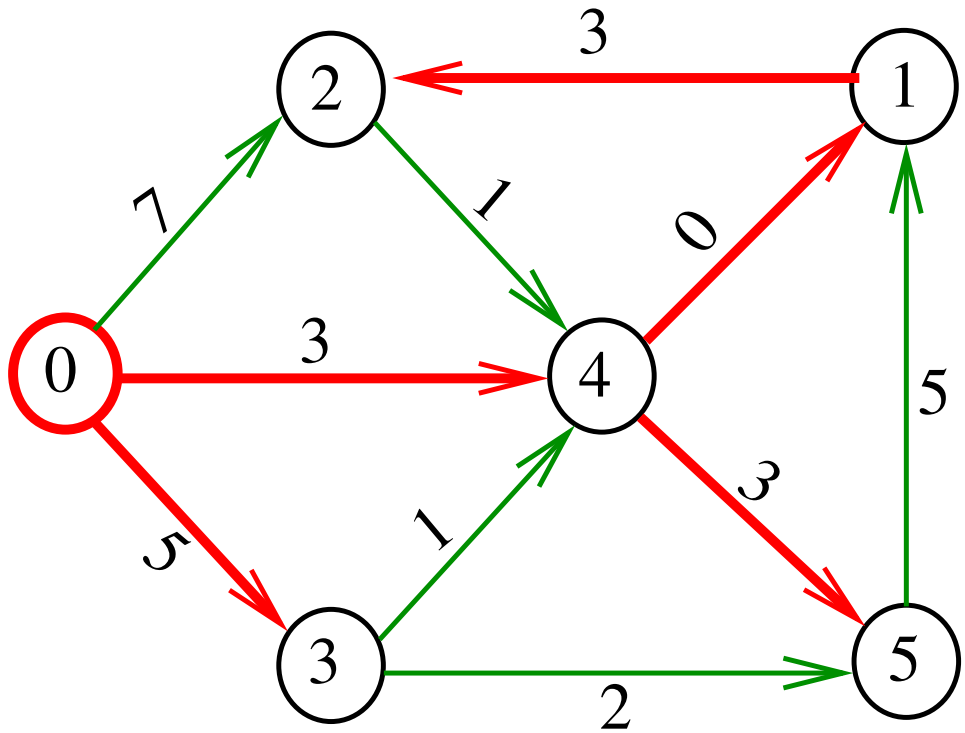
Sai:



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,

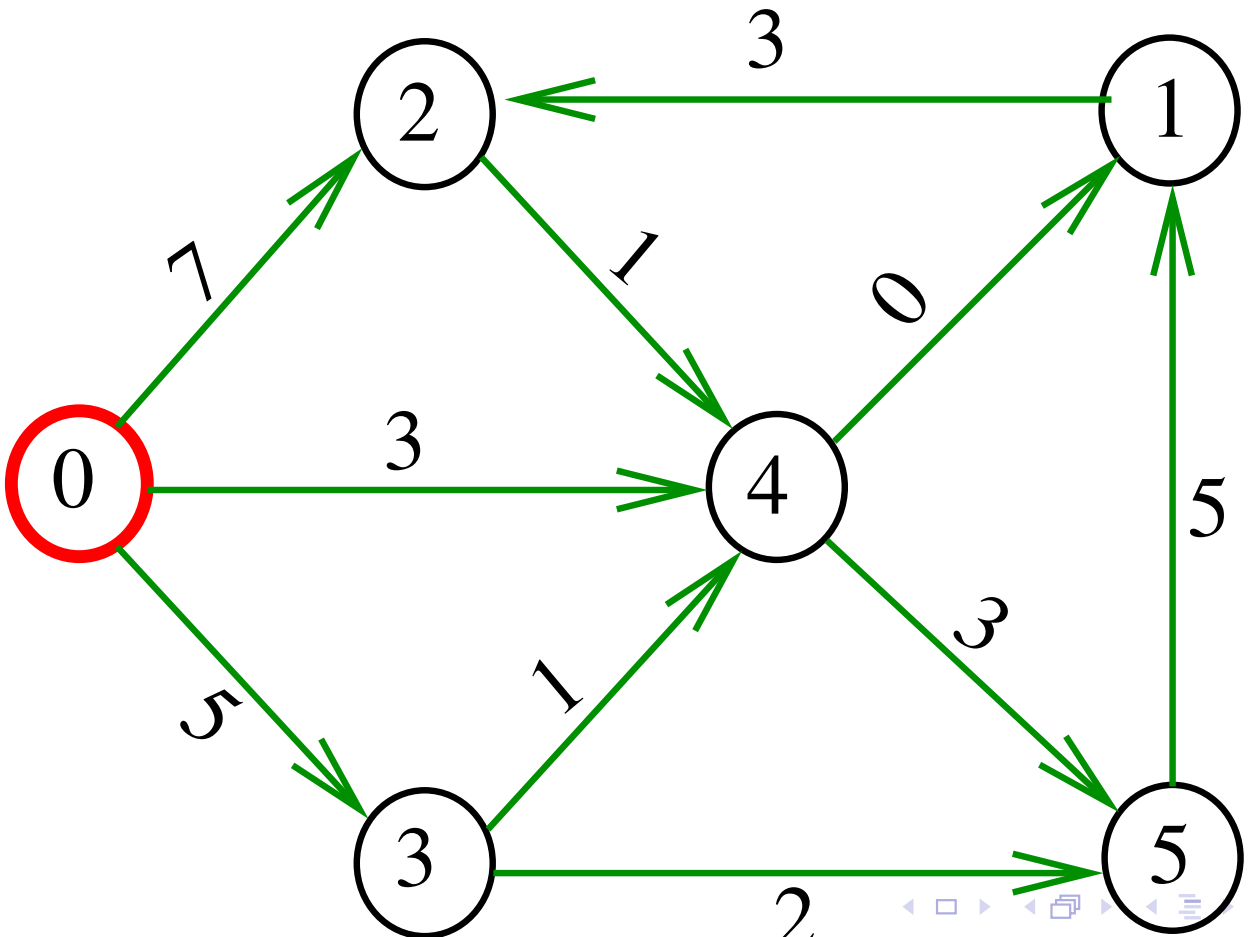
> *o único caminho de s a t na arborescência é um caminho mínimo*



Problema da SPT

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

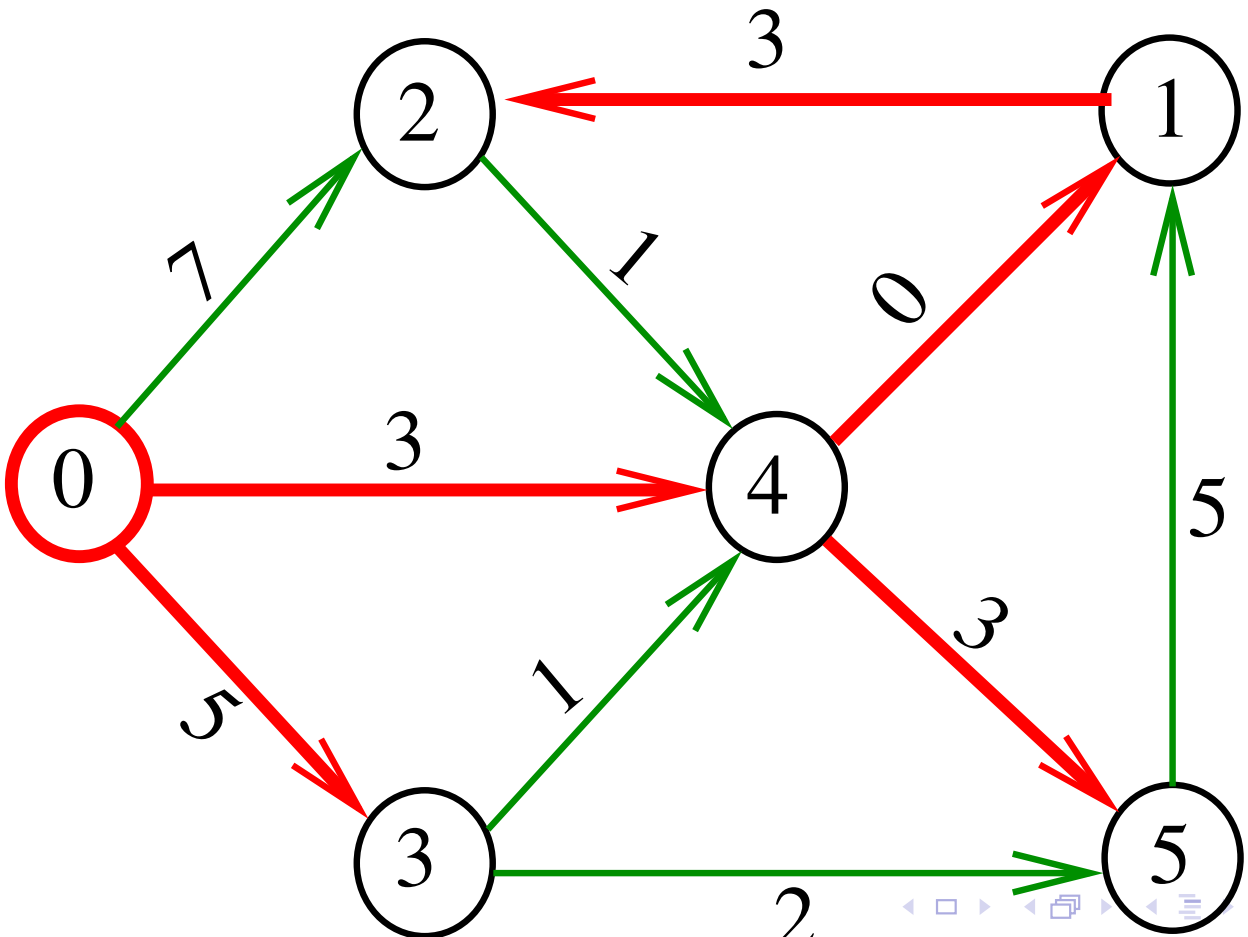
Entra:



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 14

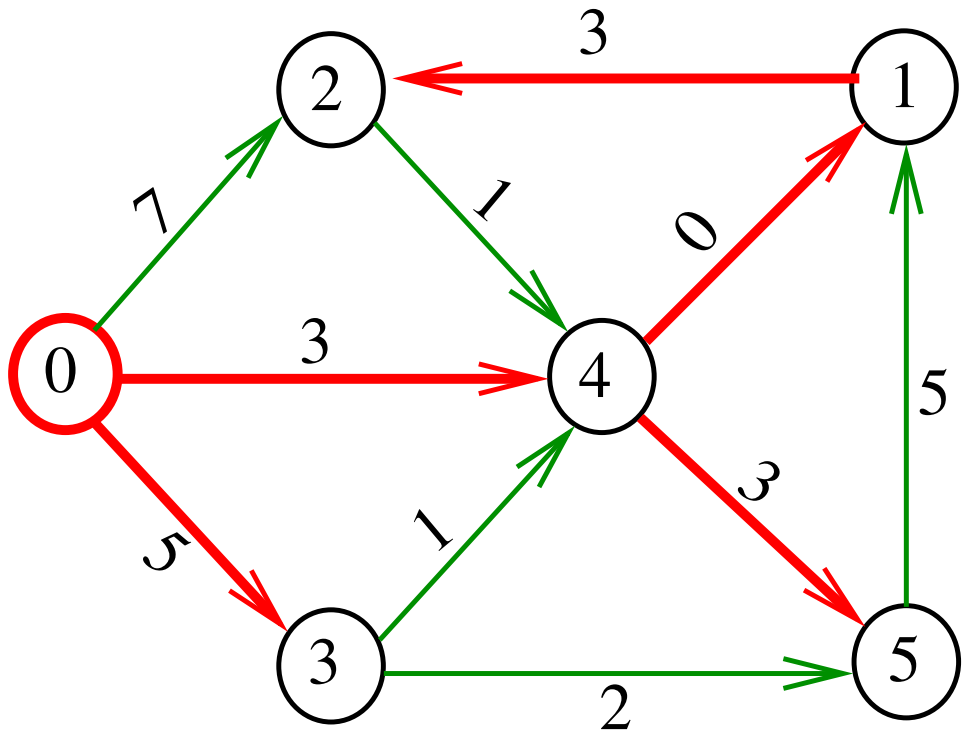
Algoritmo de Dijkstra

S 21.1 e 21.2

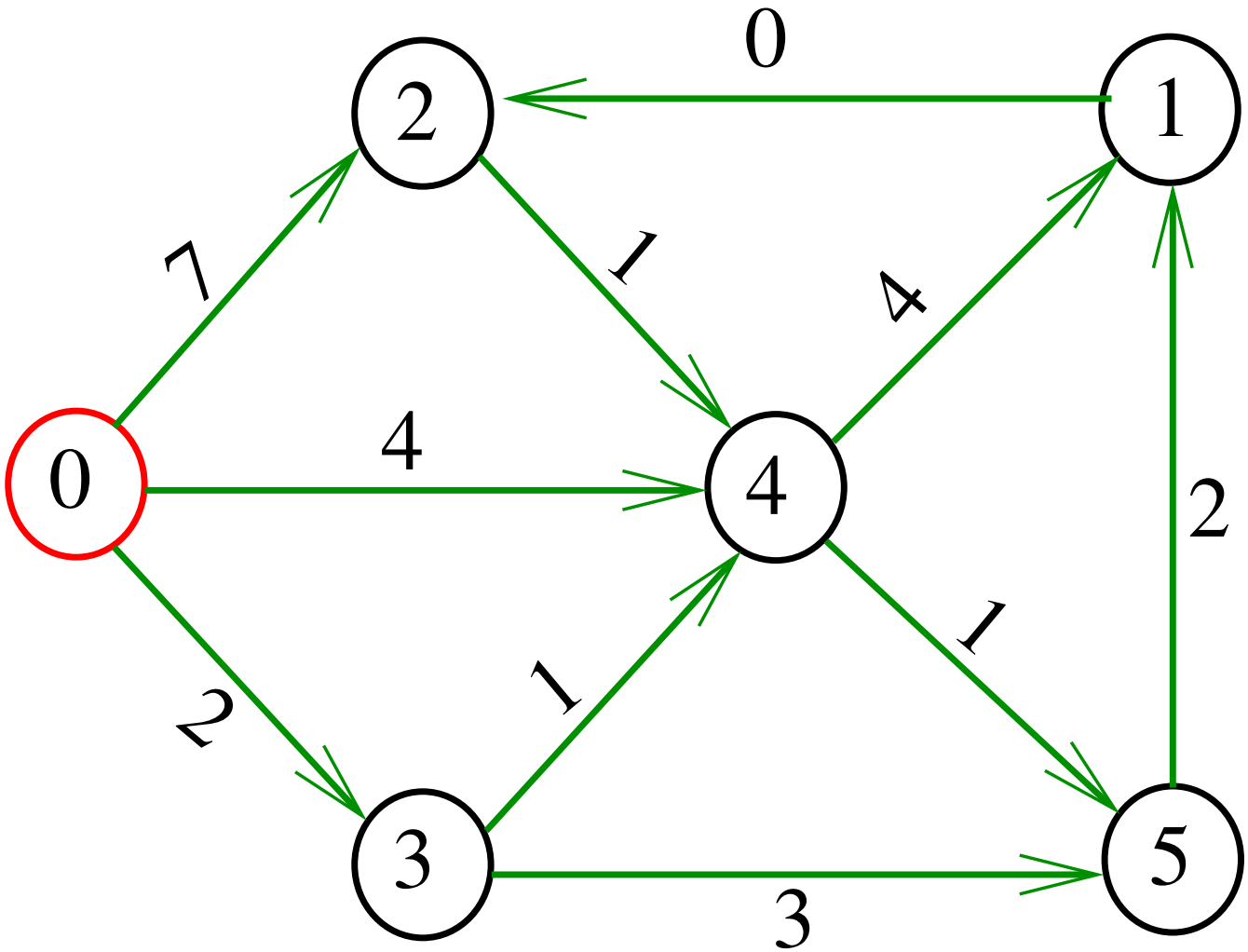
Problema

O algoritmo de Dijkstra resolve o problema da SPT:

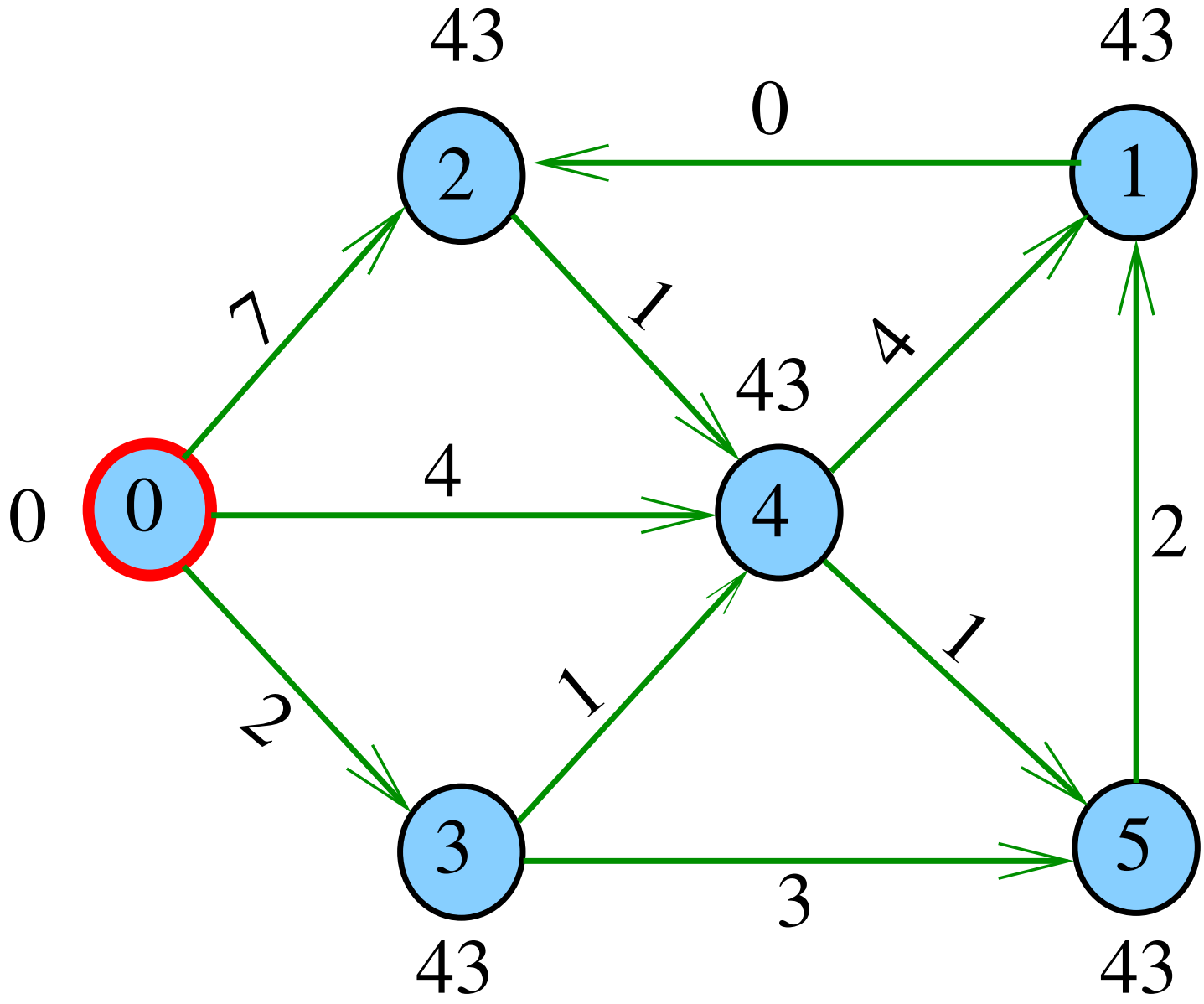
Dado um vértice s de um digrafo com custos não-negativos nos arcos, encontrar uma SPT com raiz s



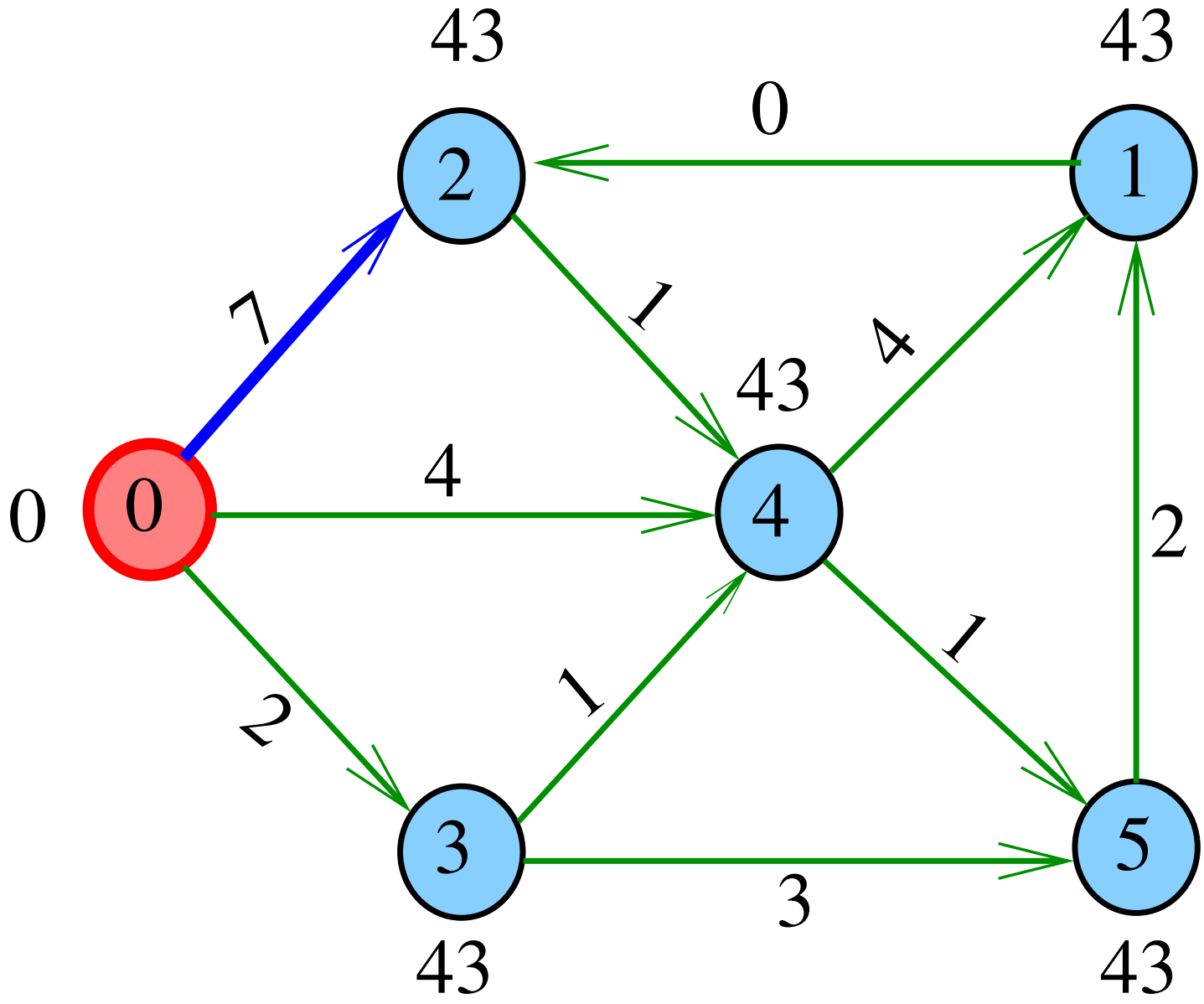
Simulação



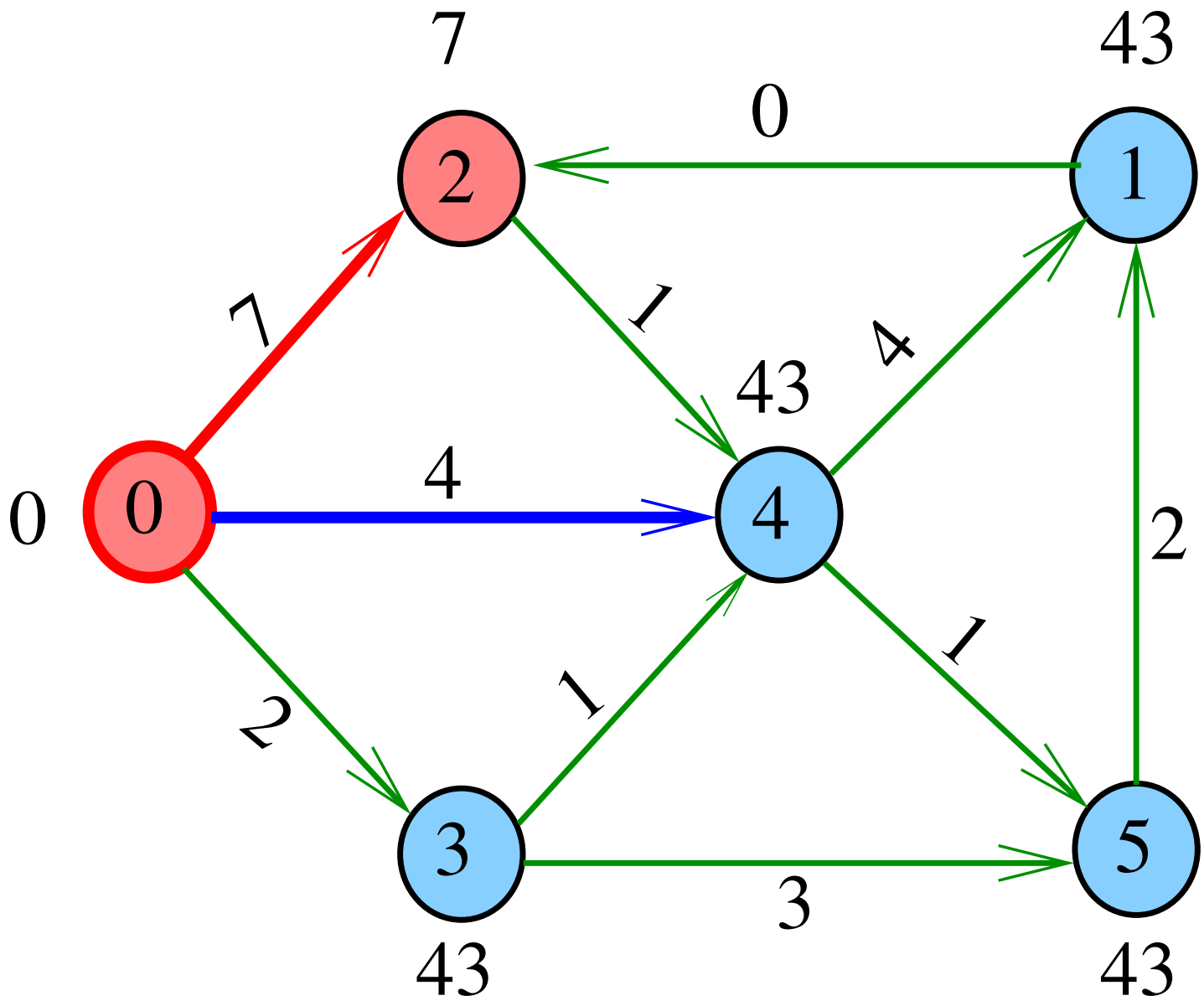
Simulação



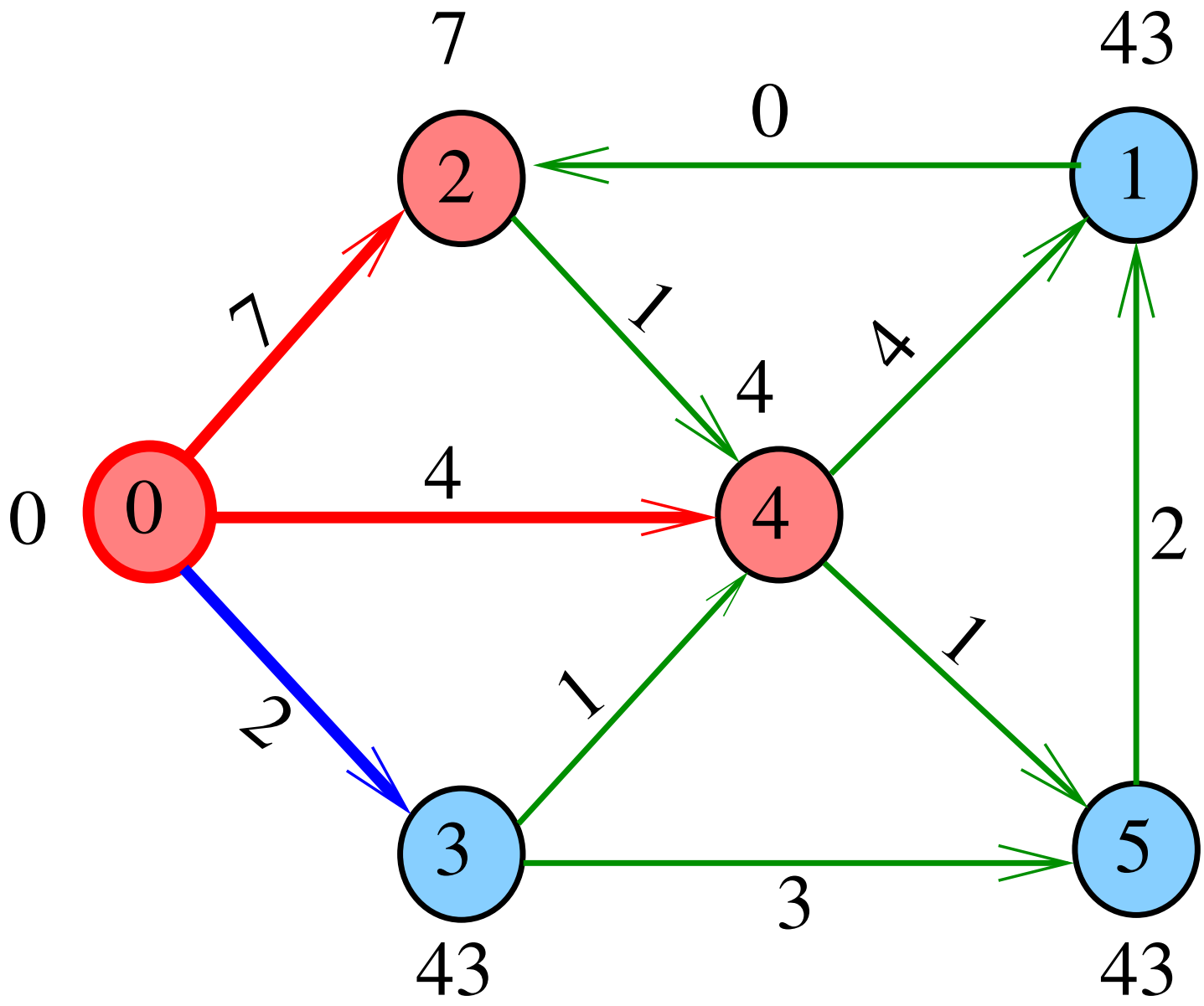
Simulação



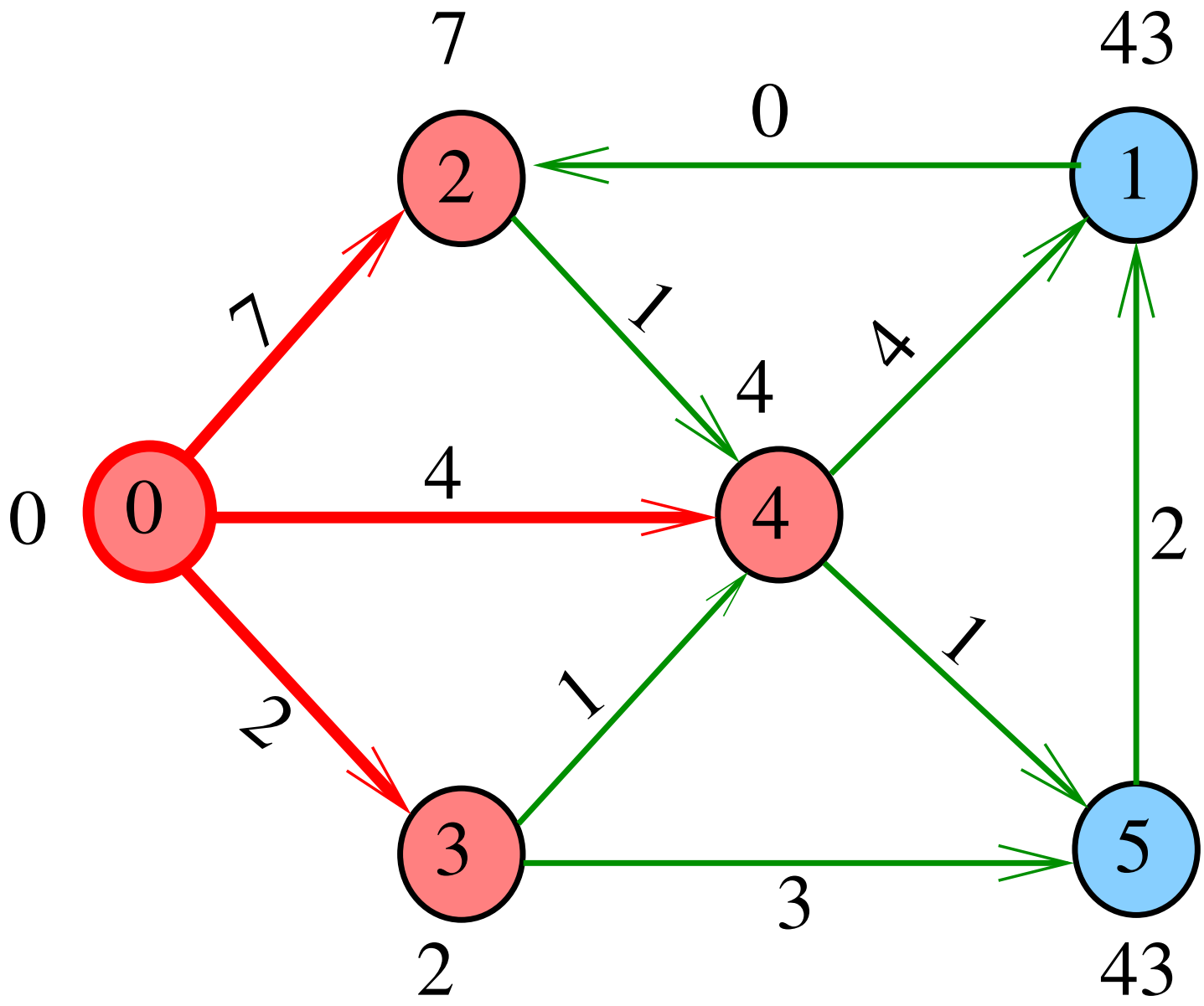
Simulação



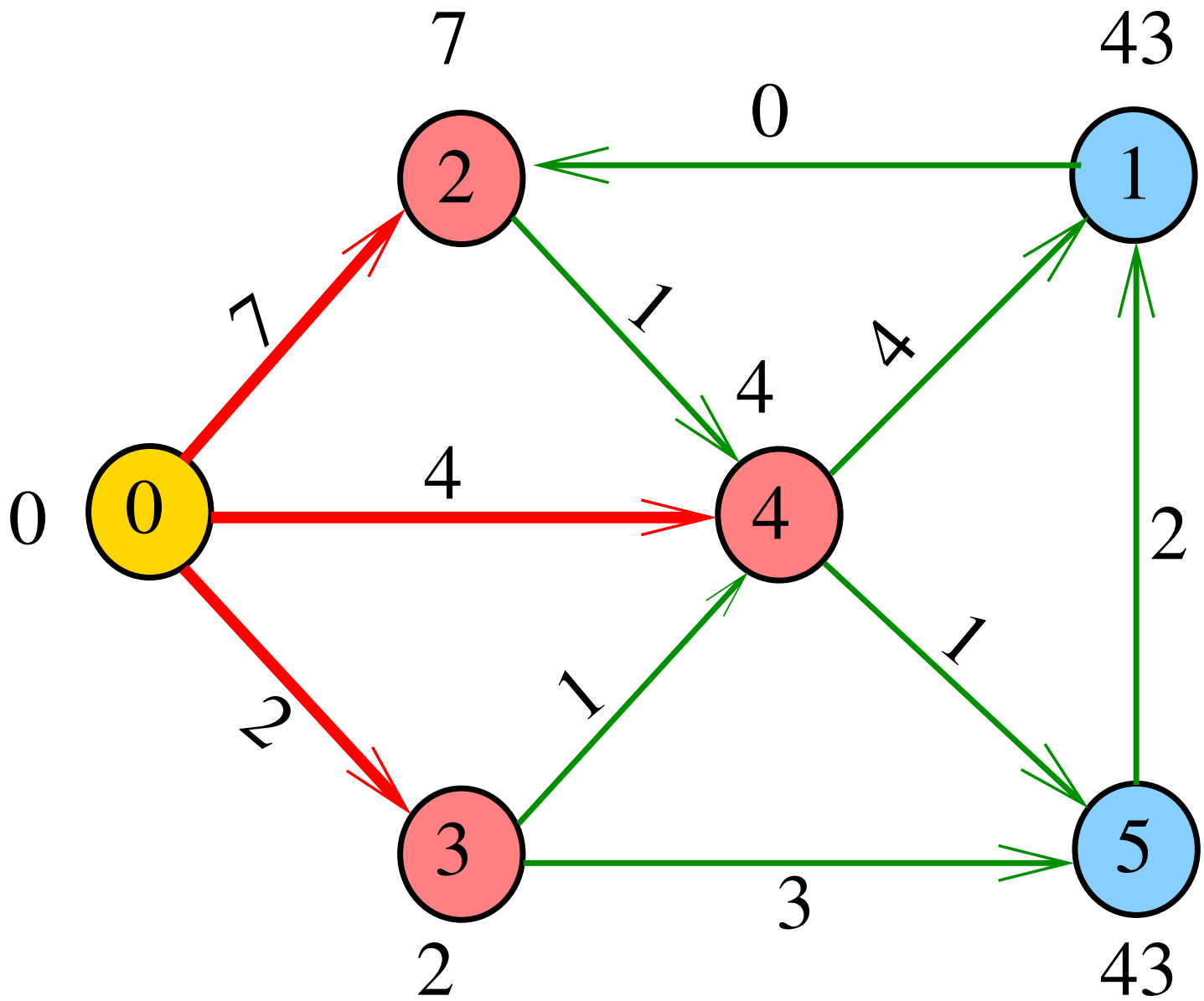
Simulação



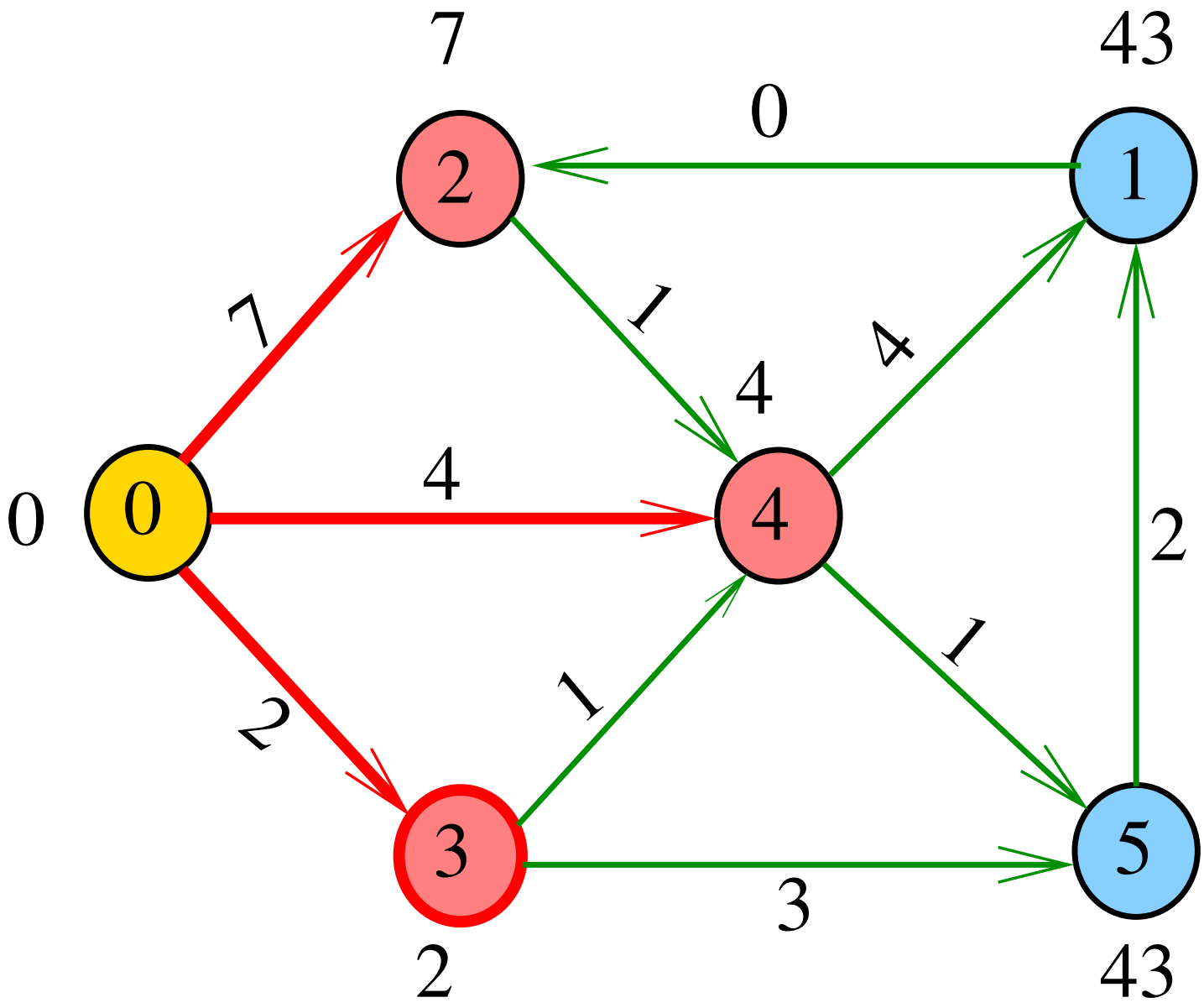
Simulação



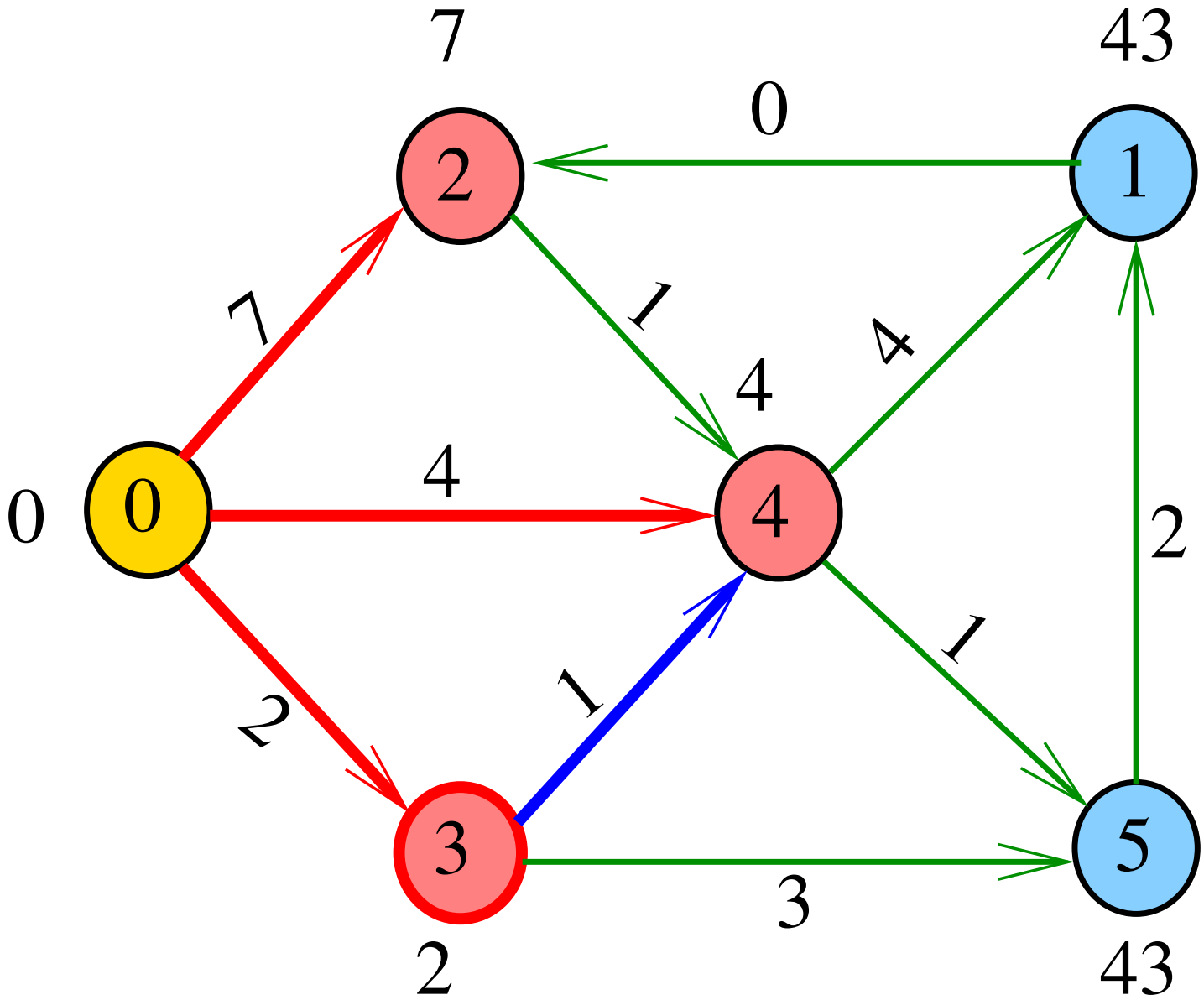
Simulação



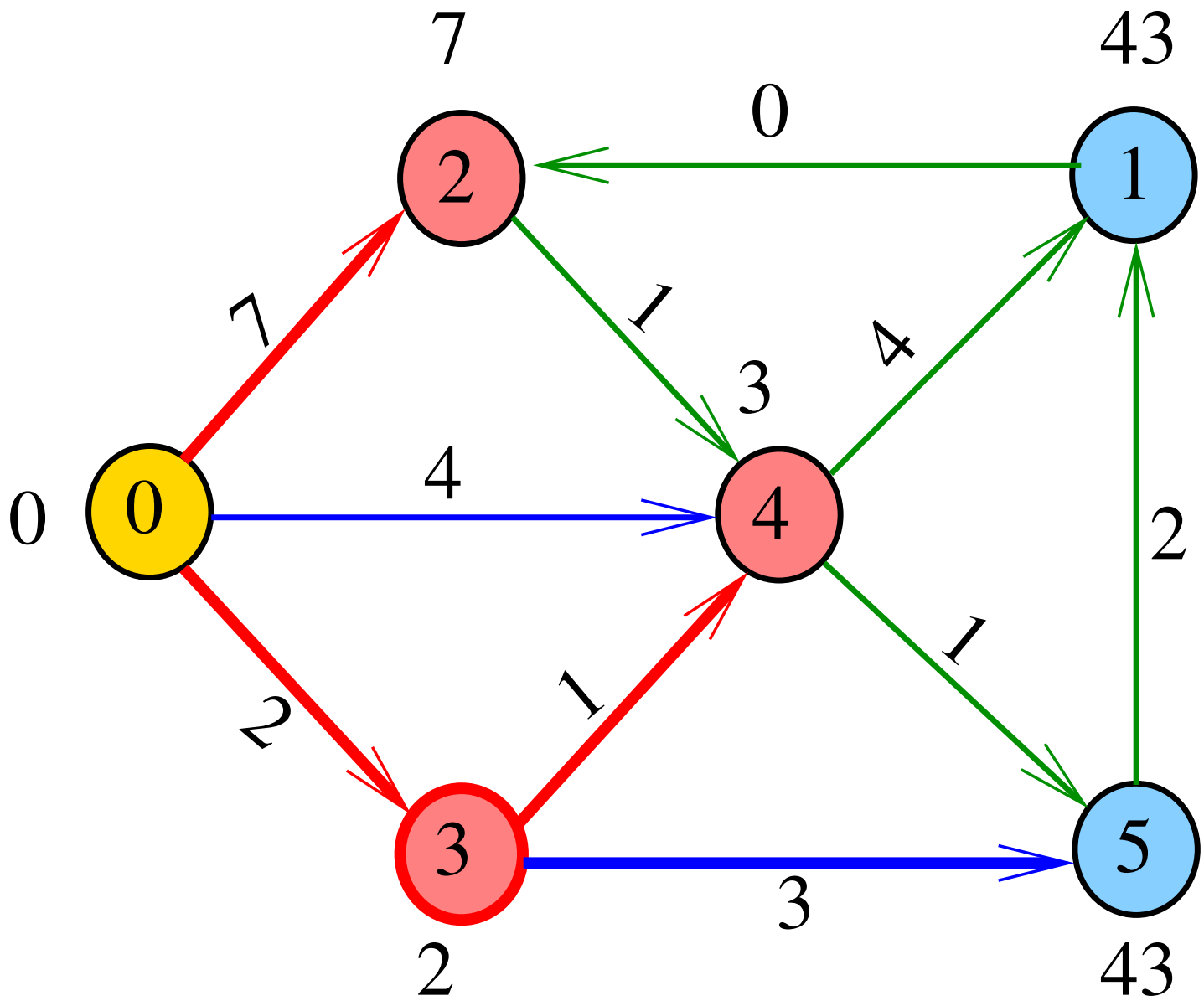
Simulação



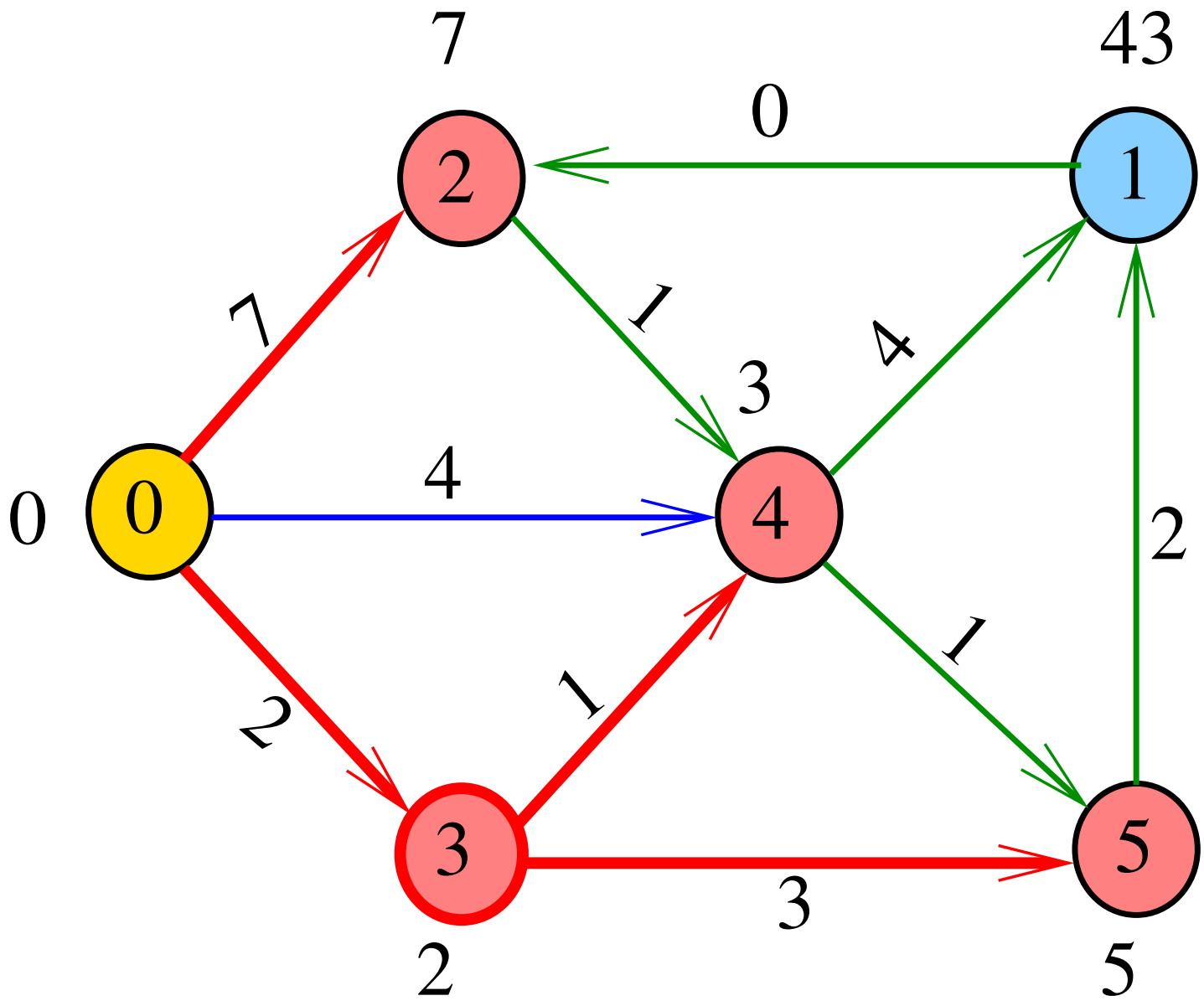
Simulação



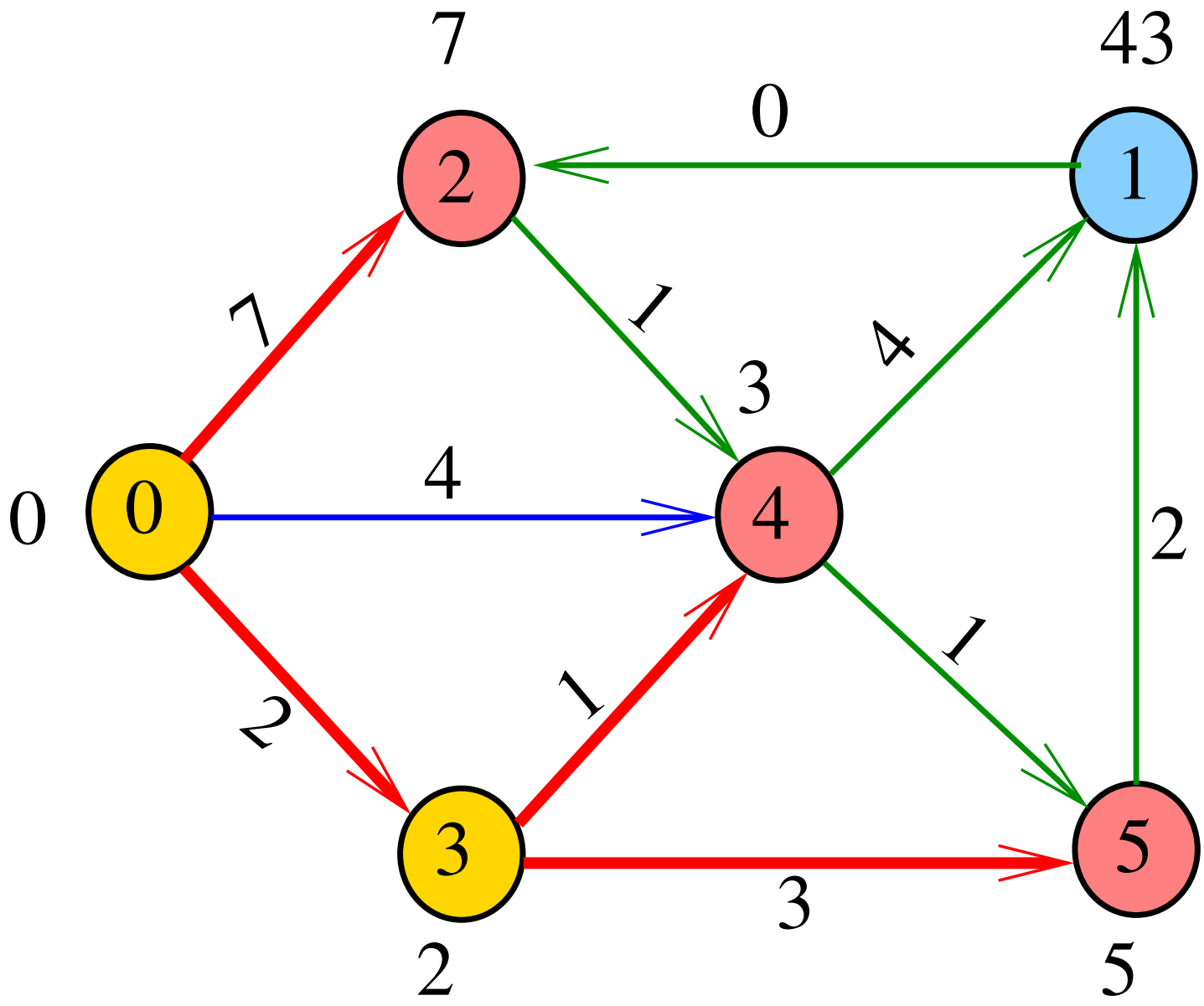
Simulação



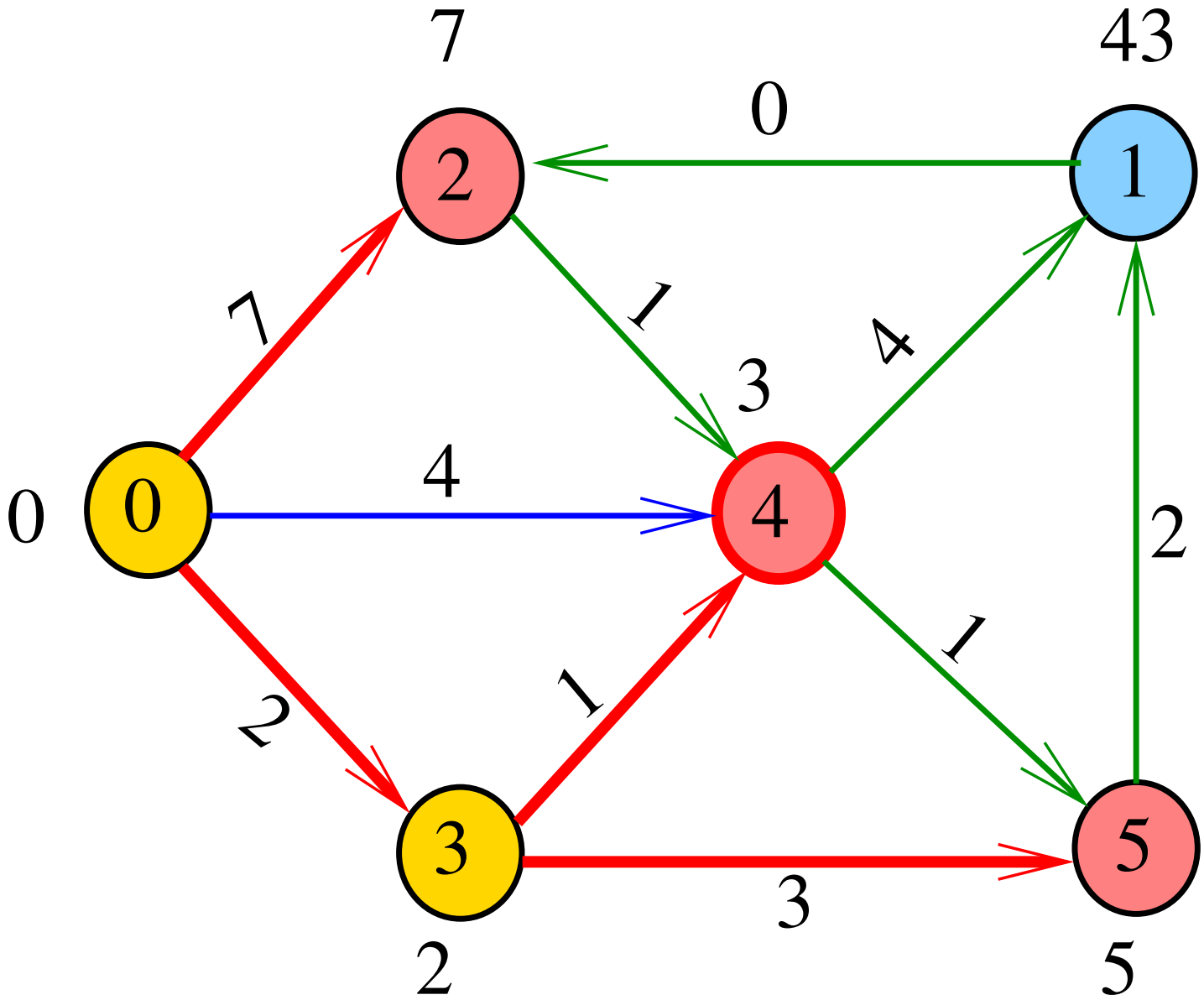
Simulação



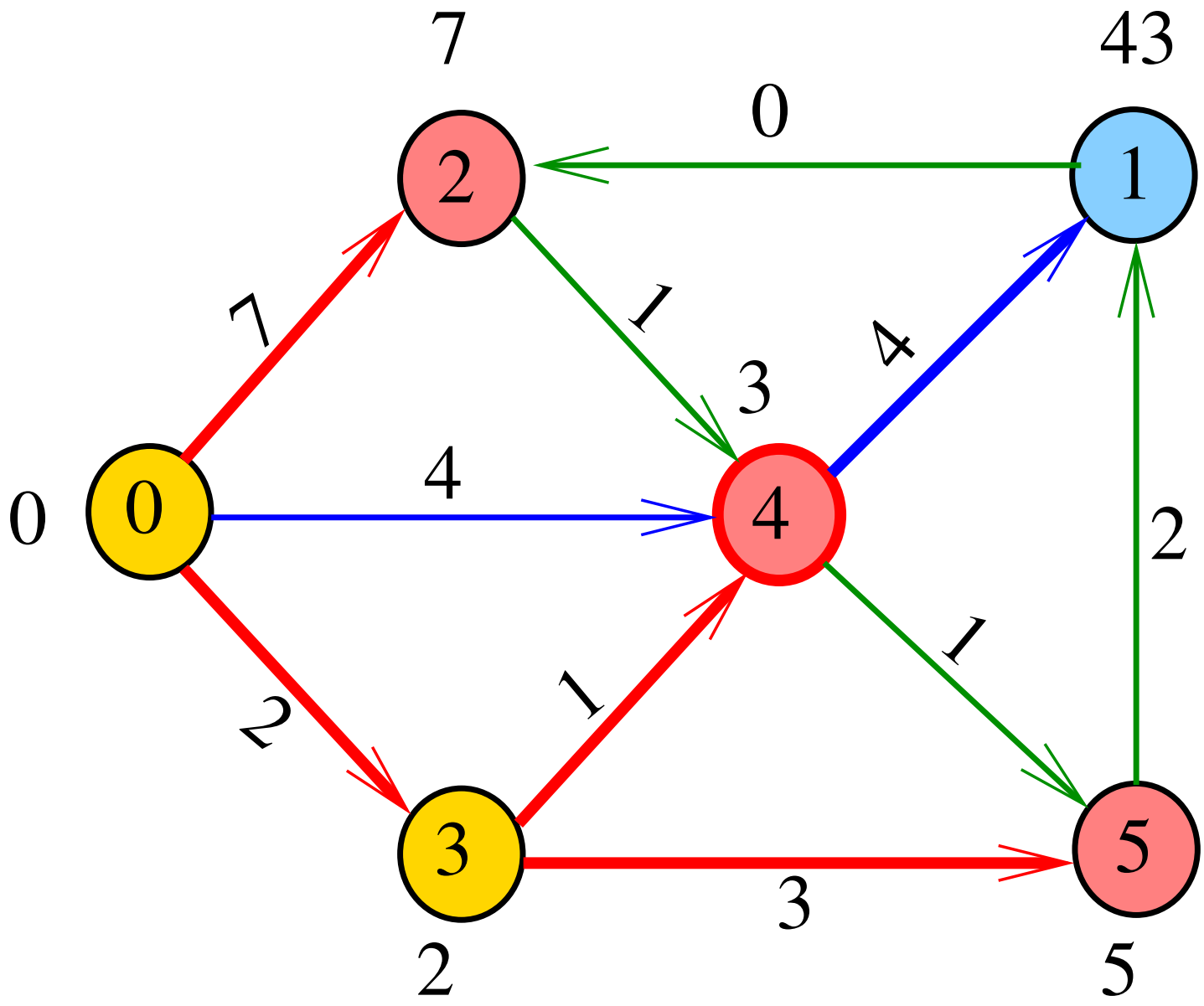
Simulação



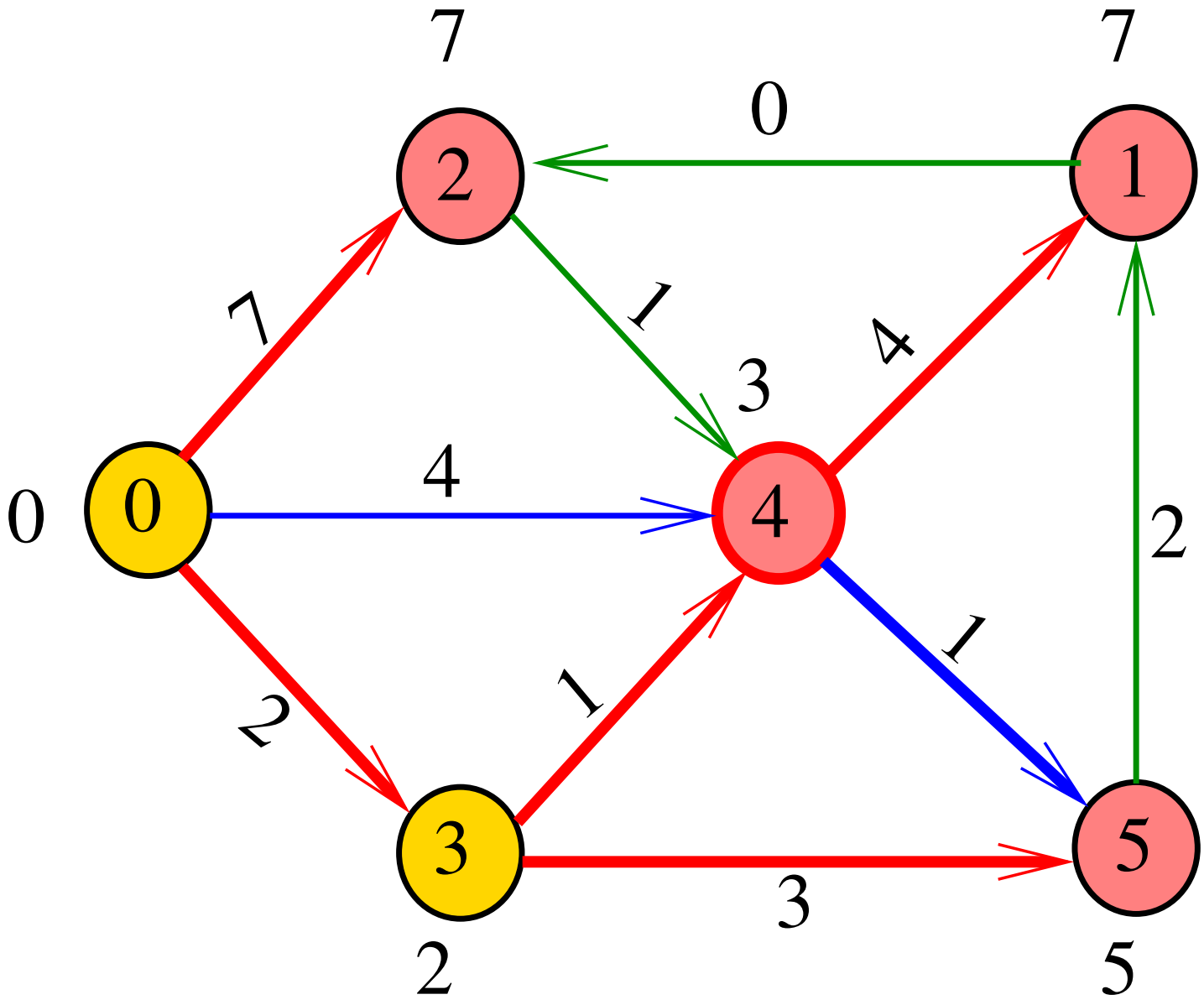
Simulação



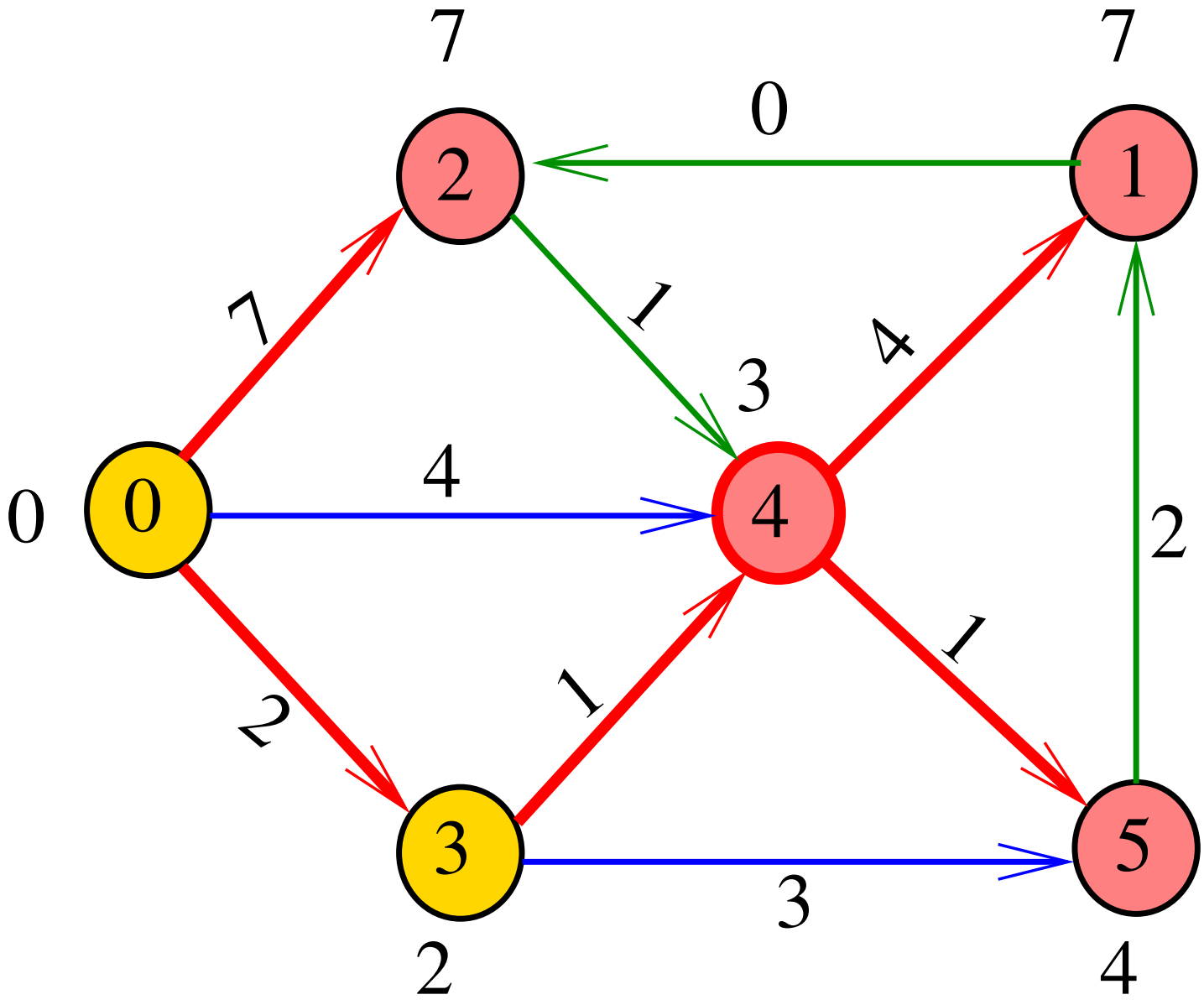
Simulação



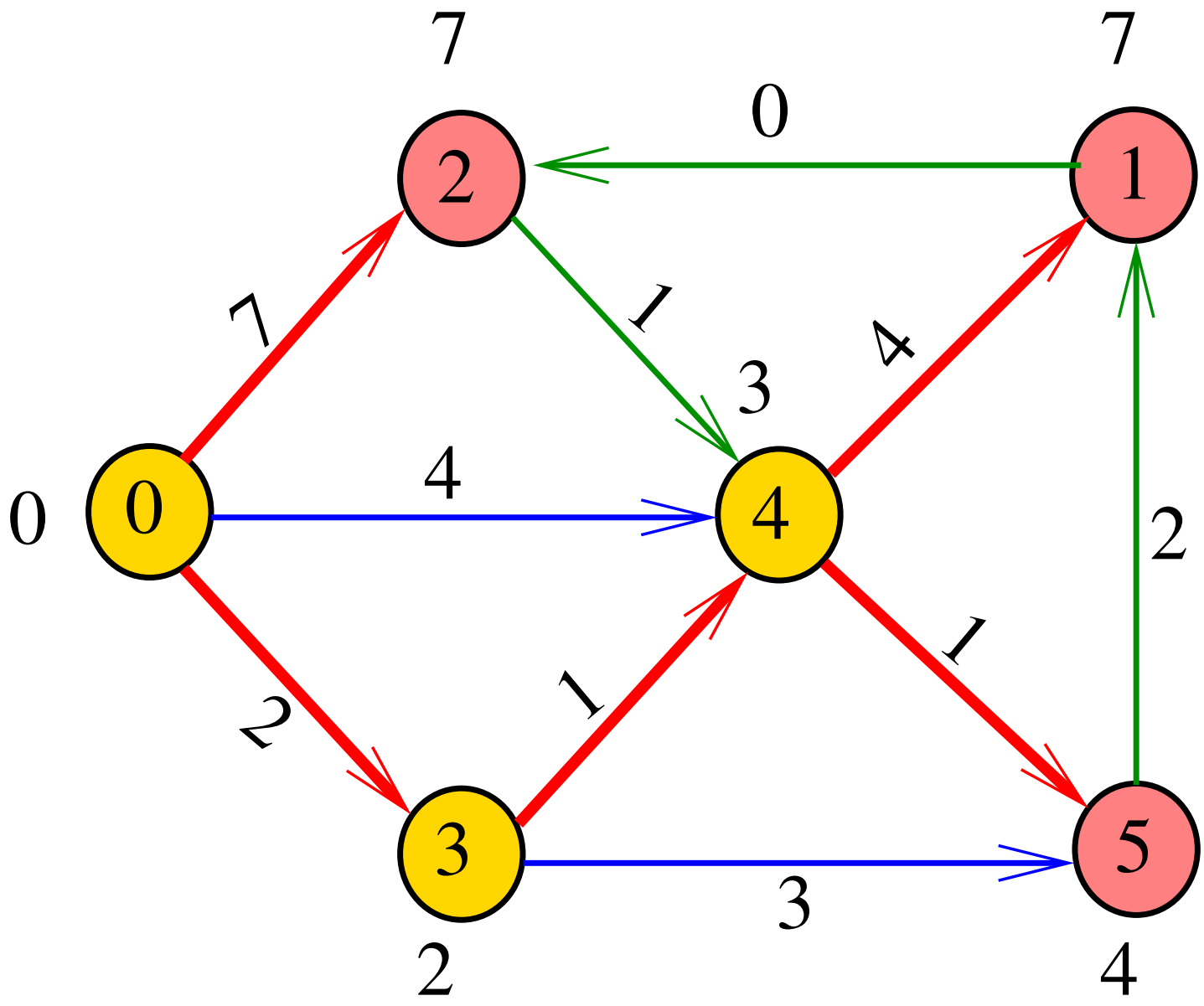
Simulação



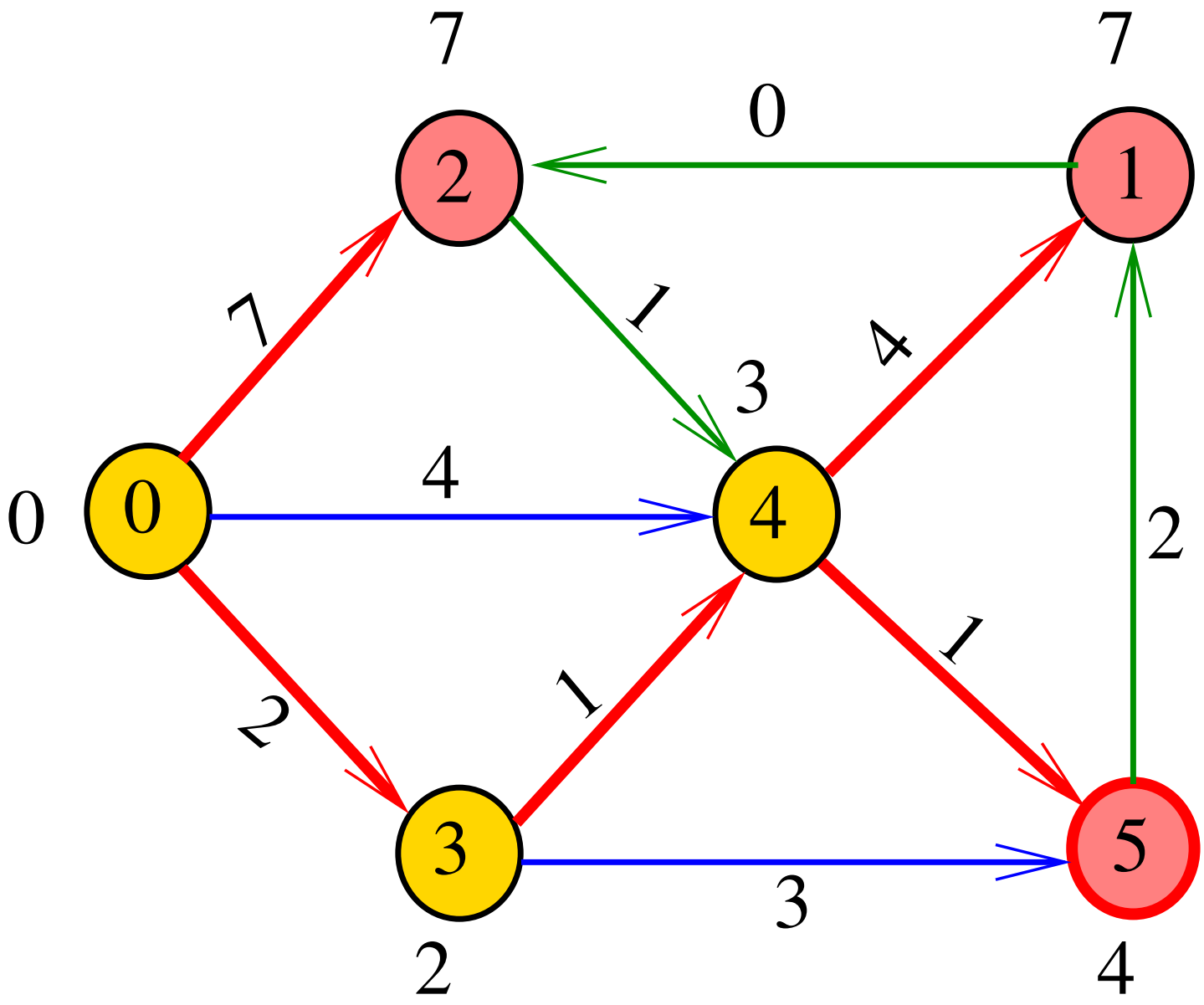
Simulação



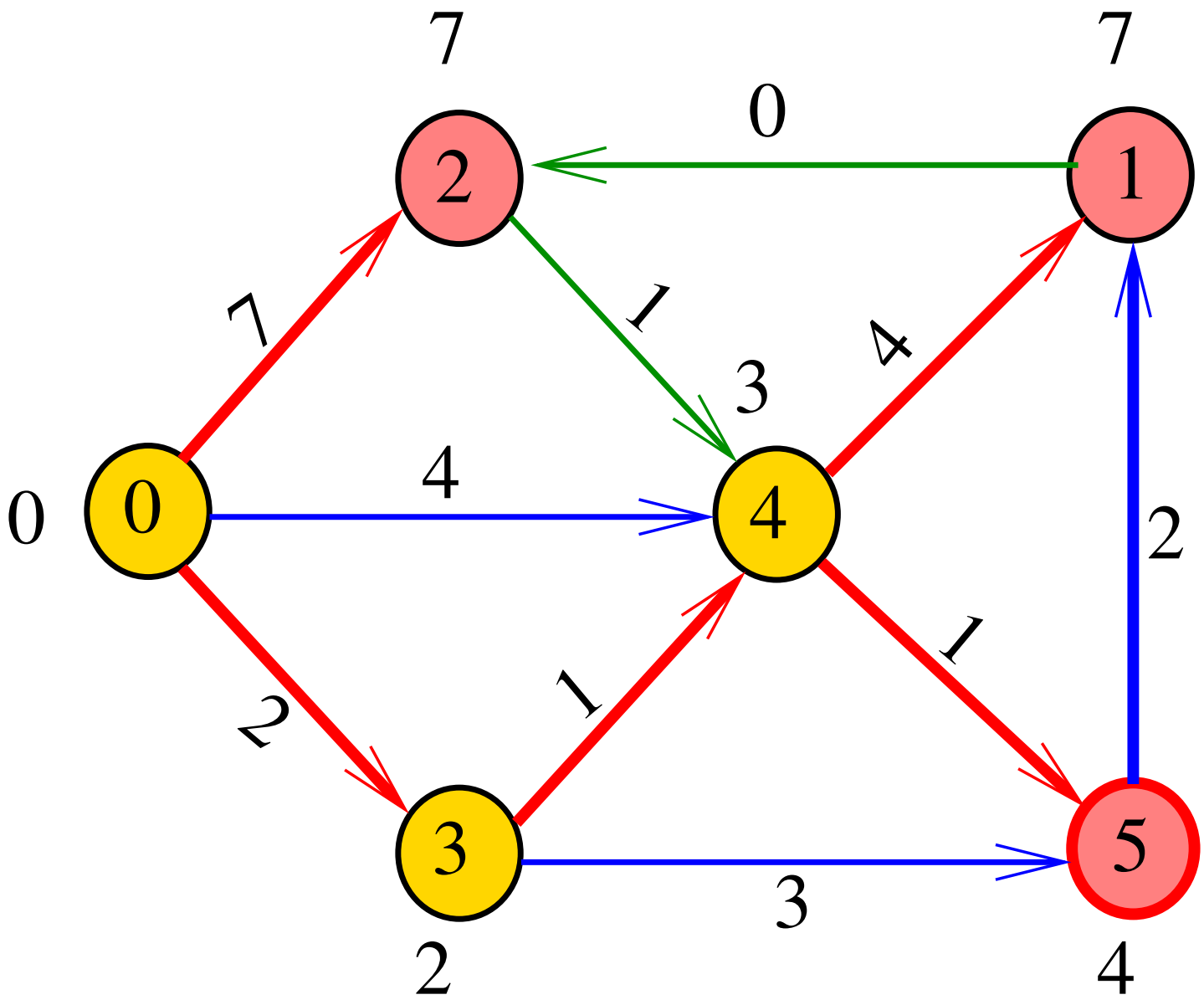
Simulação



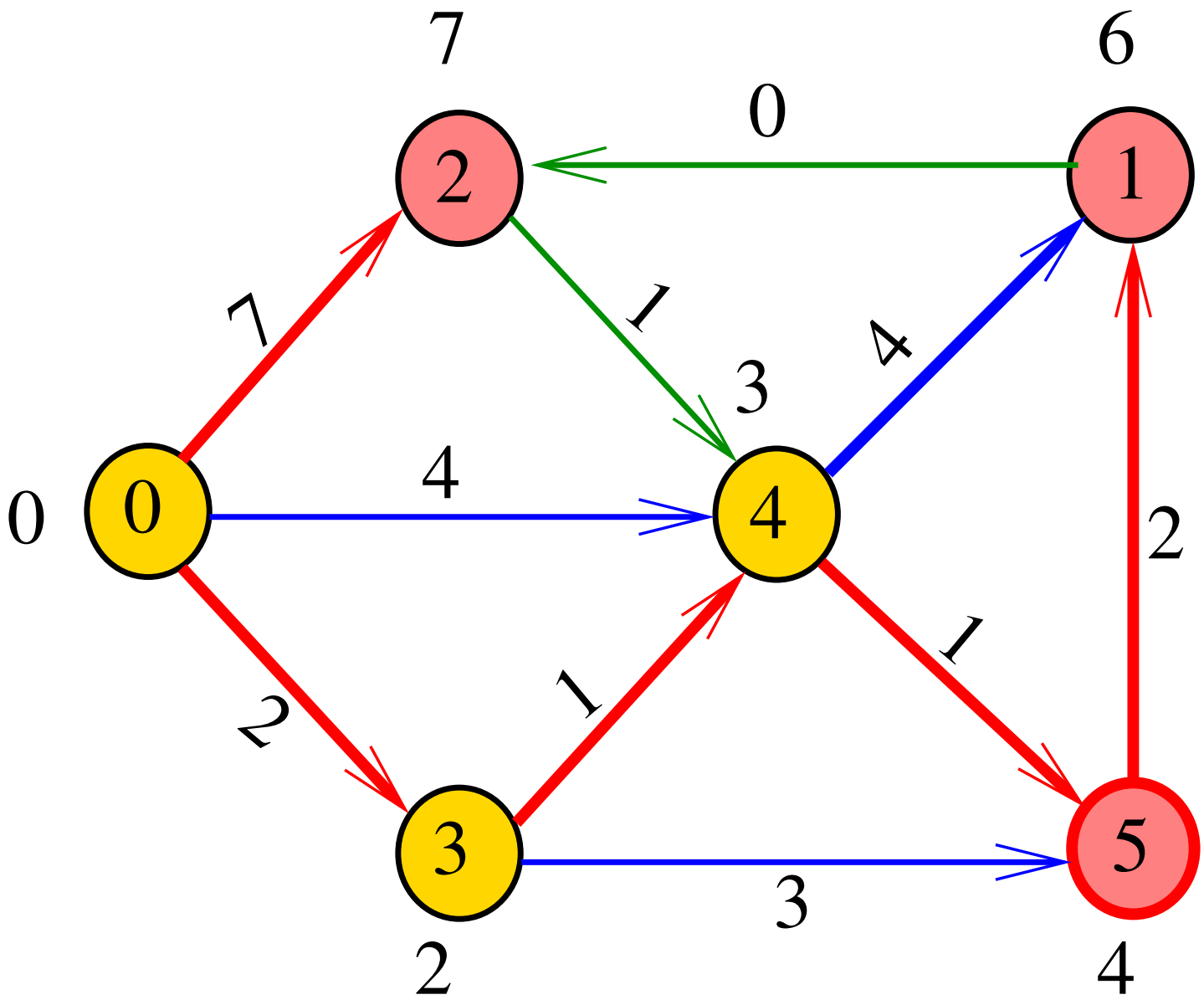
Simulação



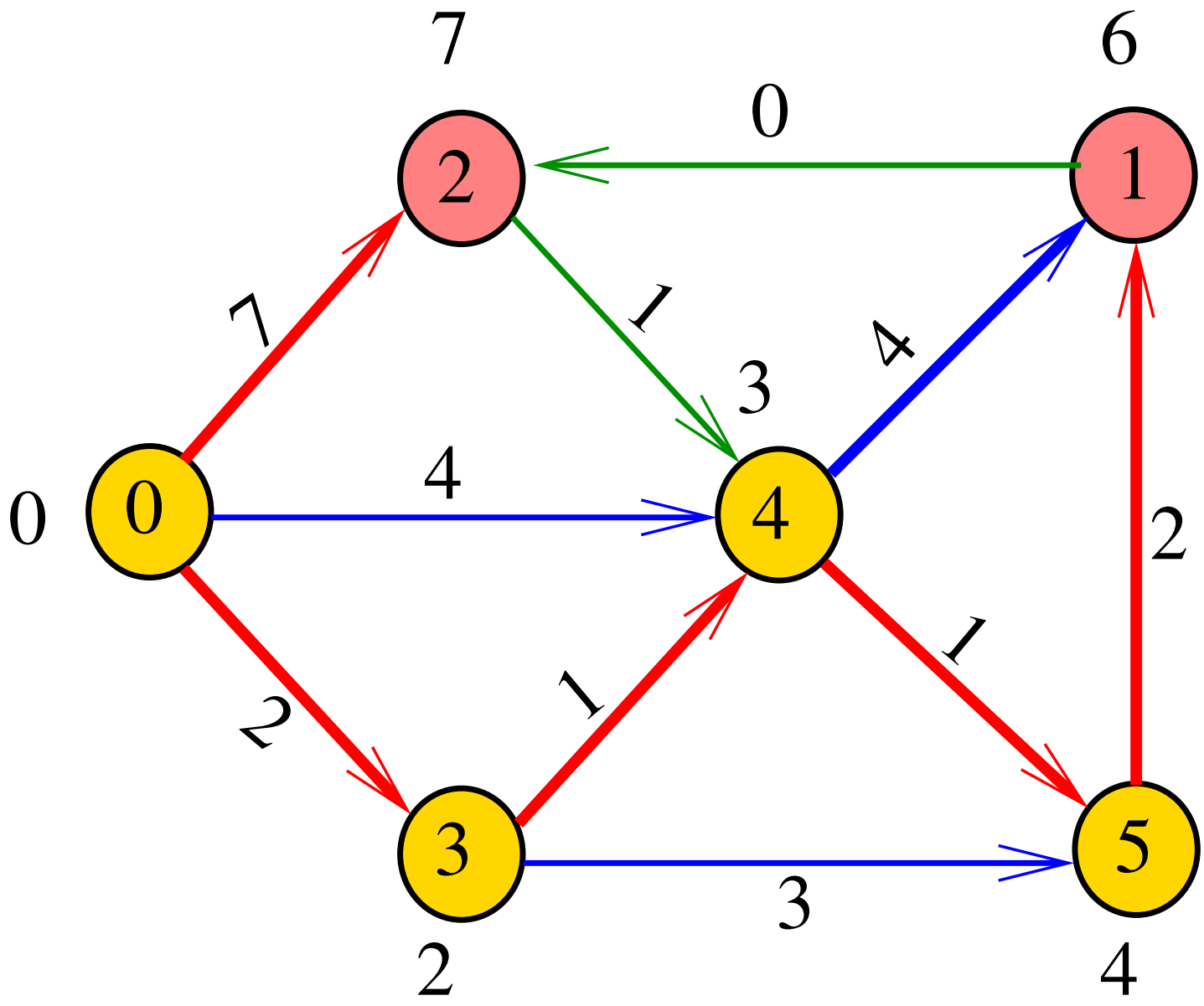
Simulação



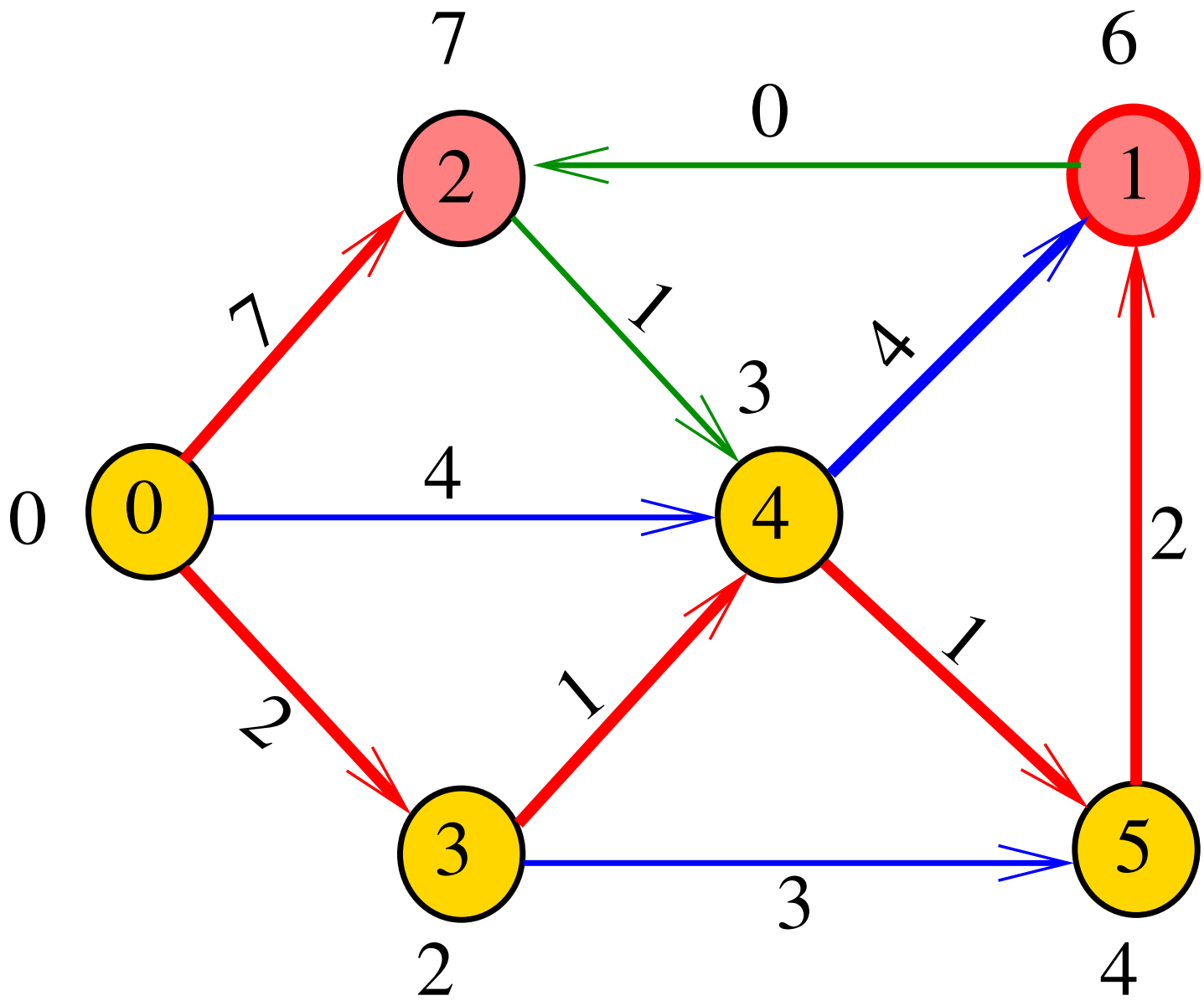
Simulação



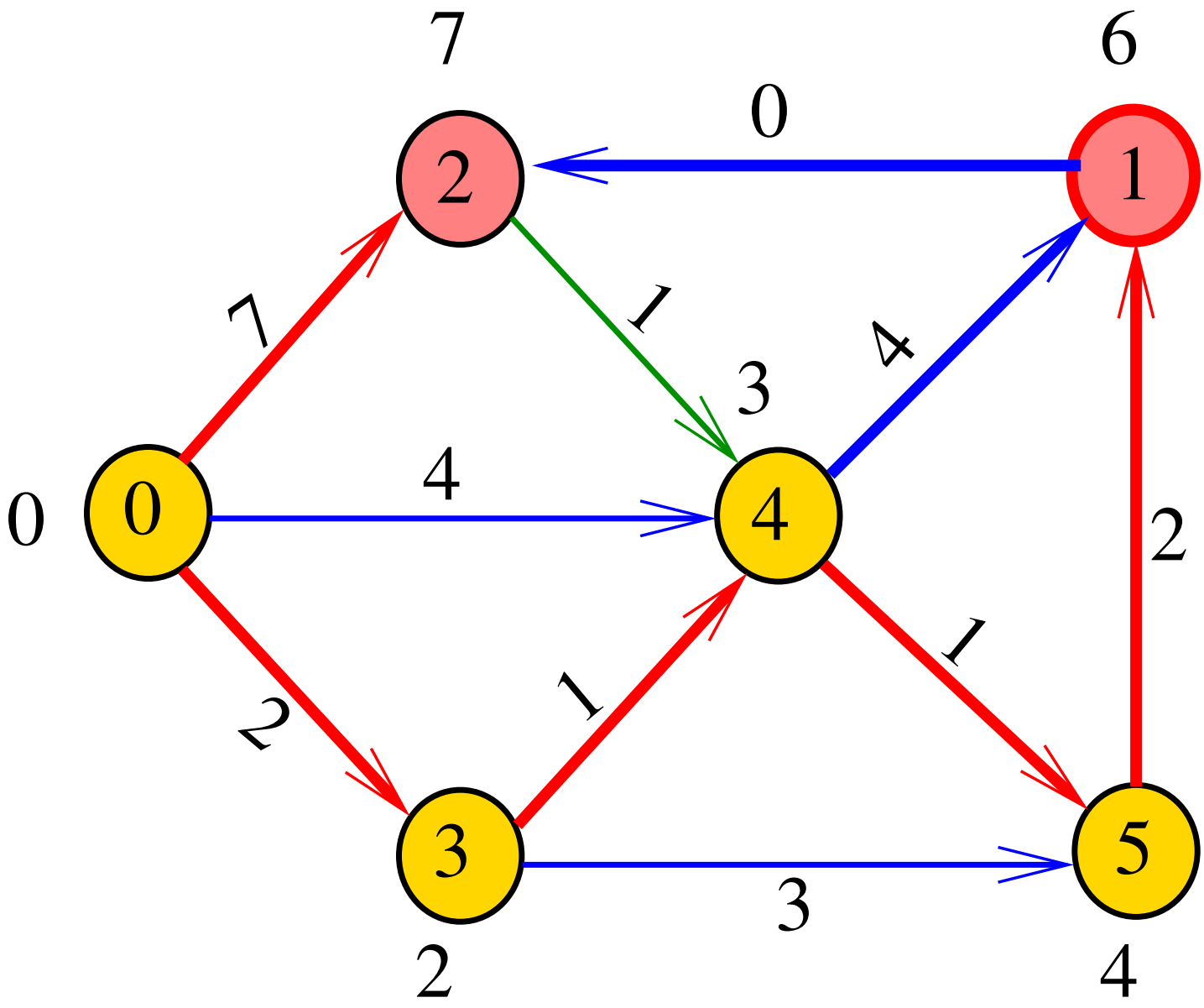
Simulação



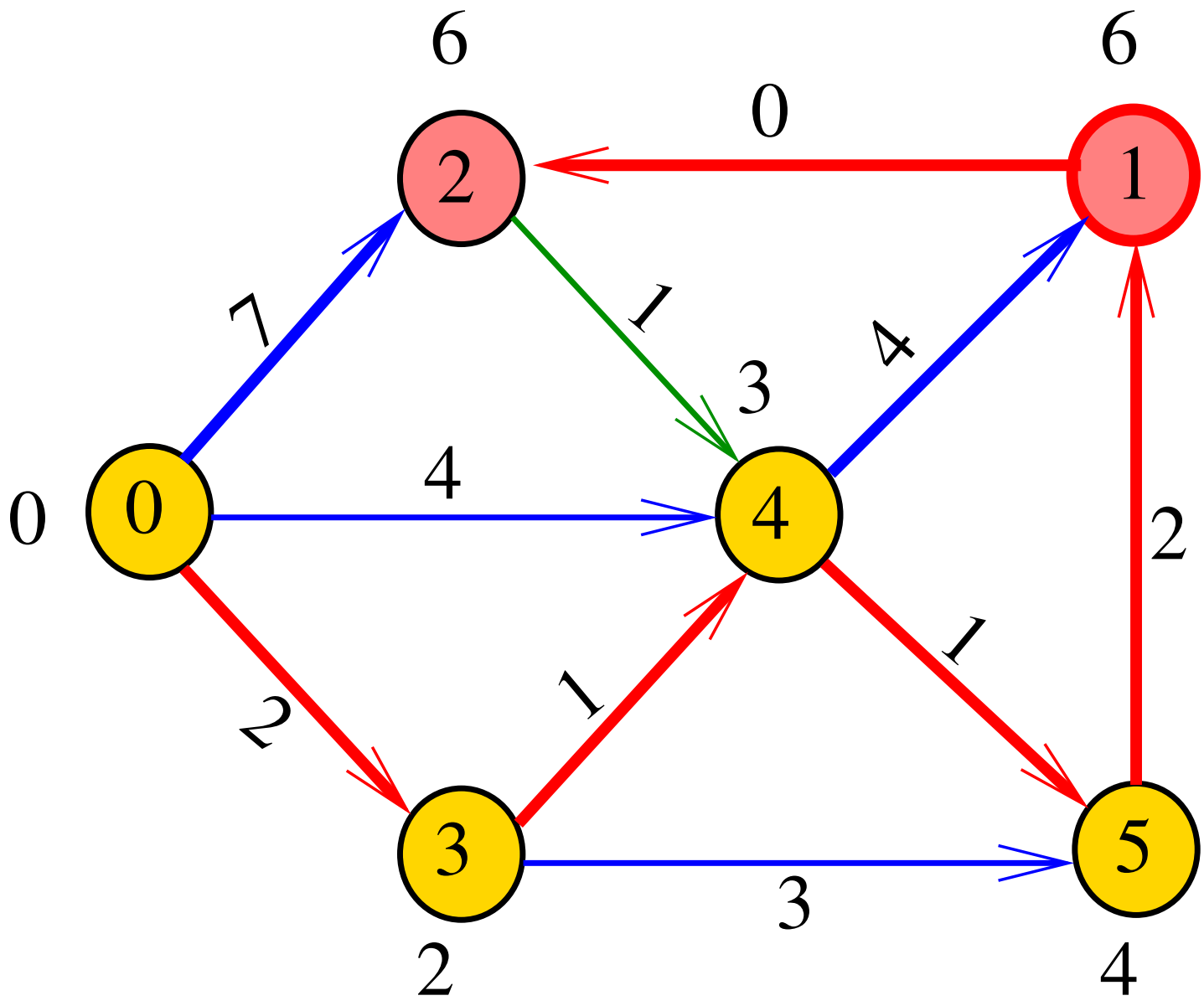
Simulação



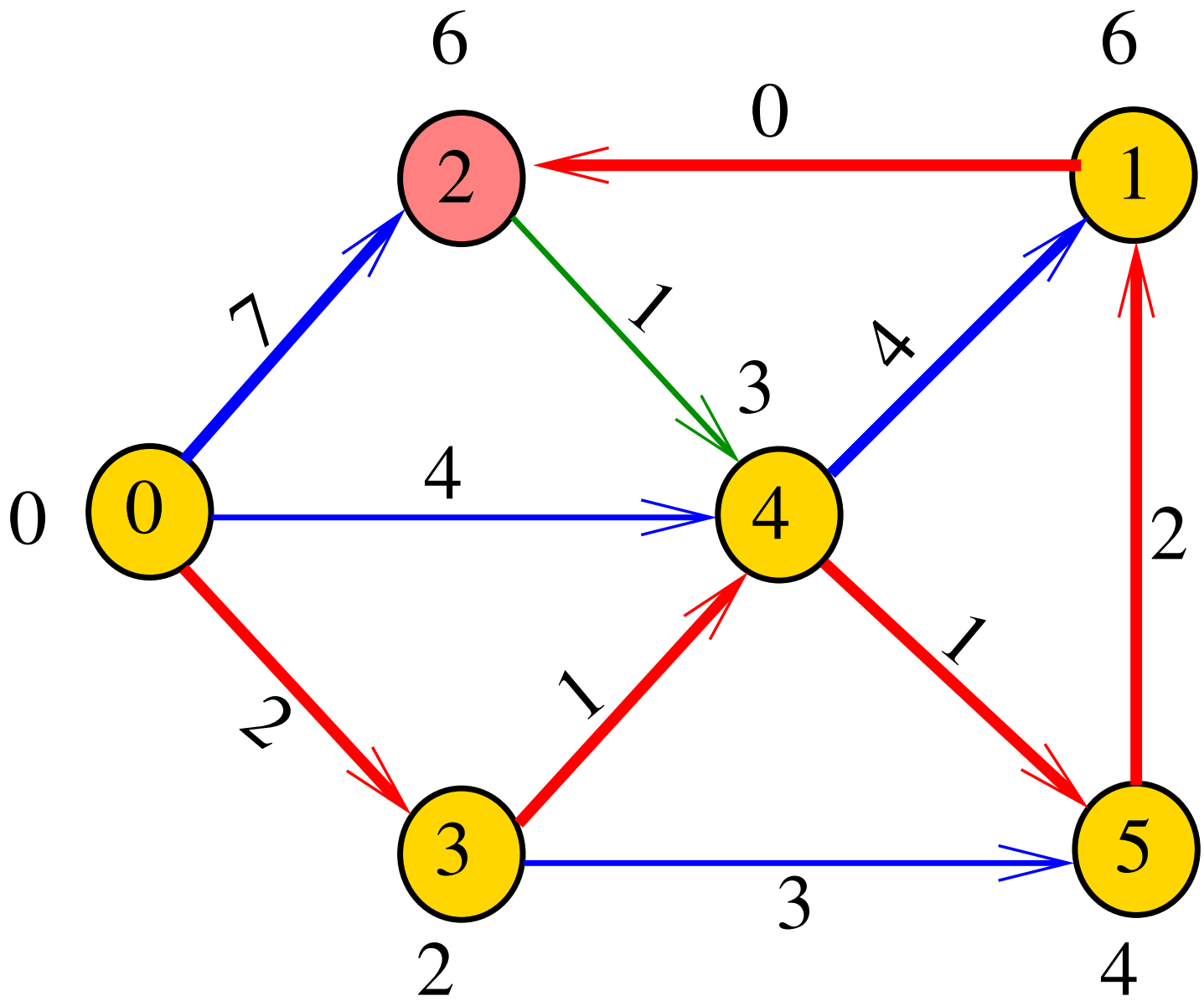
Simulação



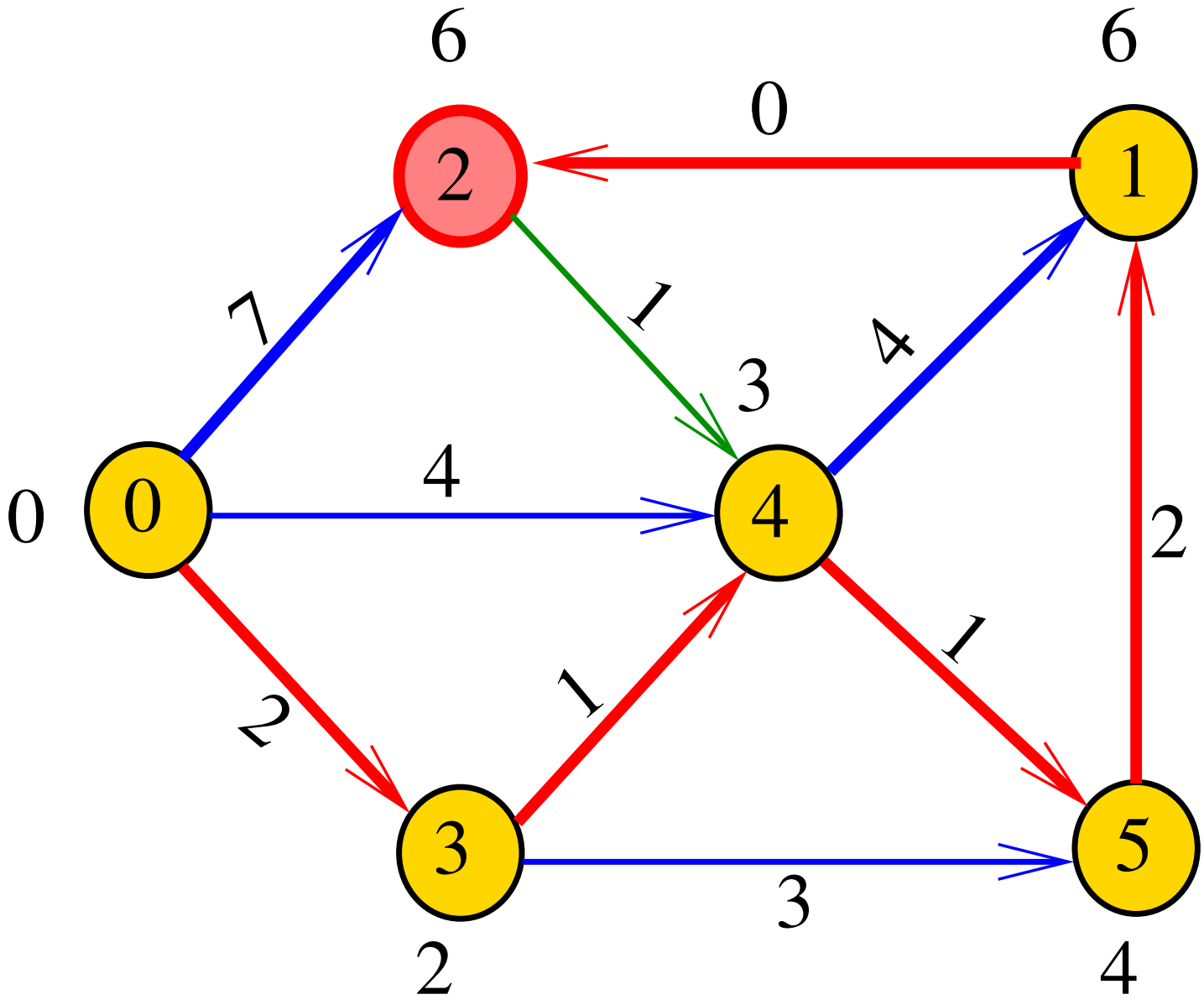
Simulação



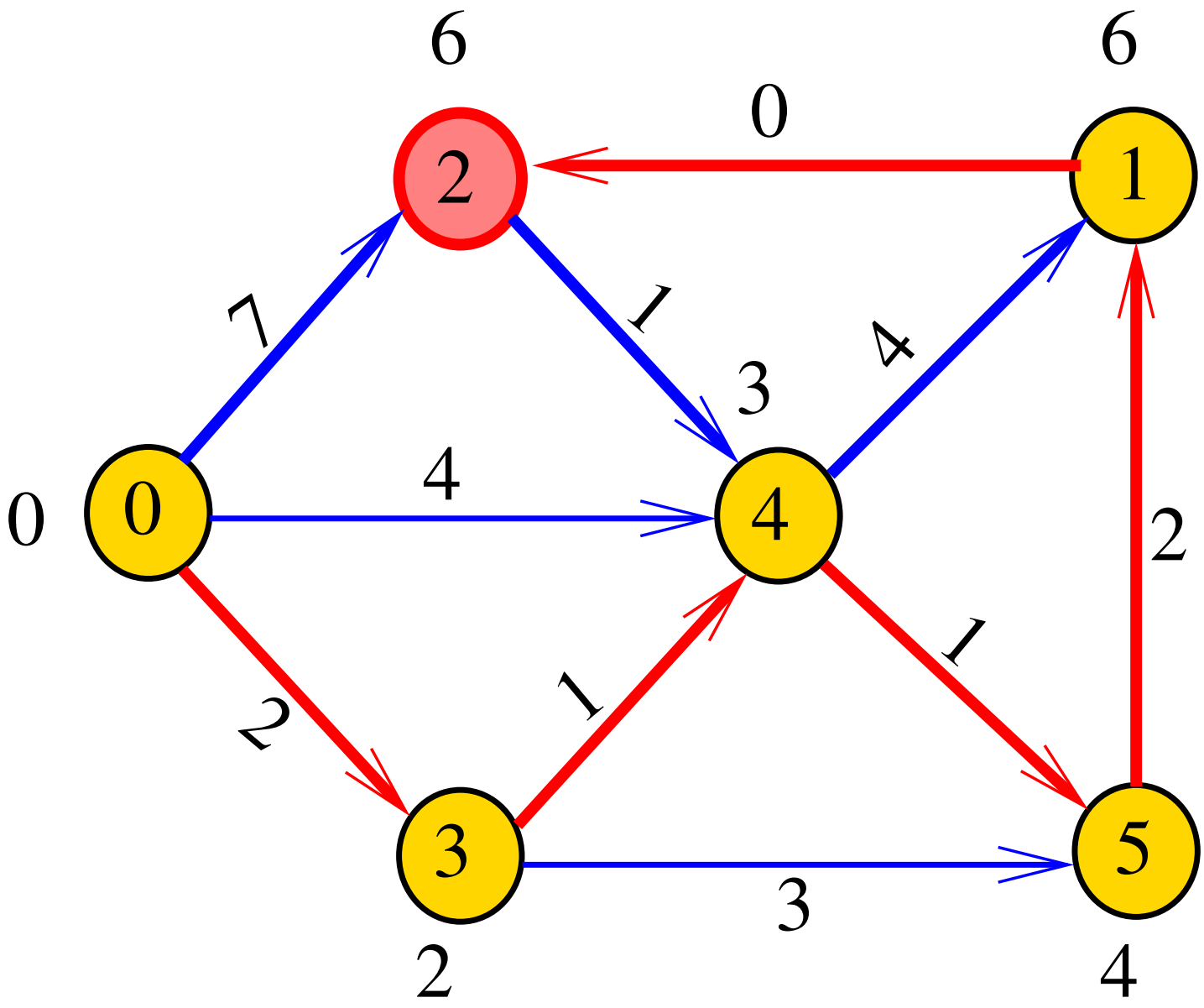
Simulação



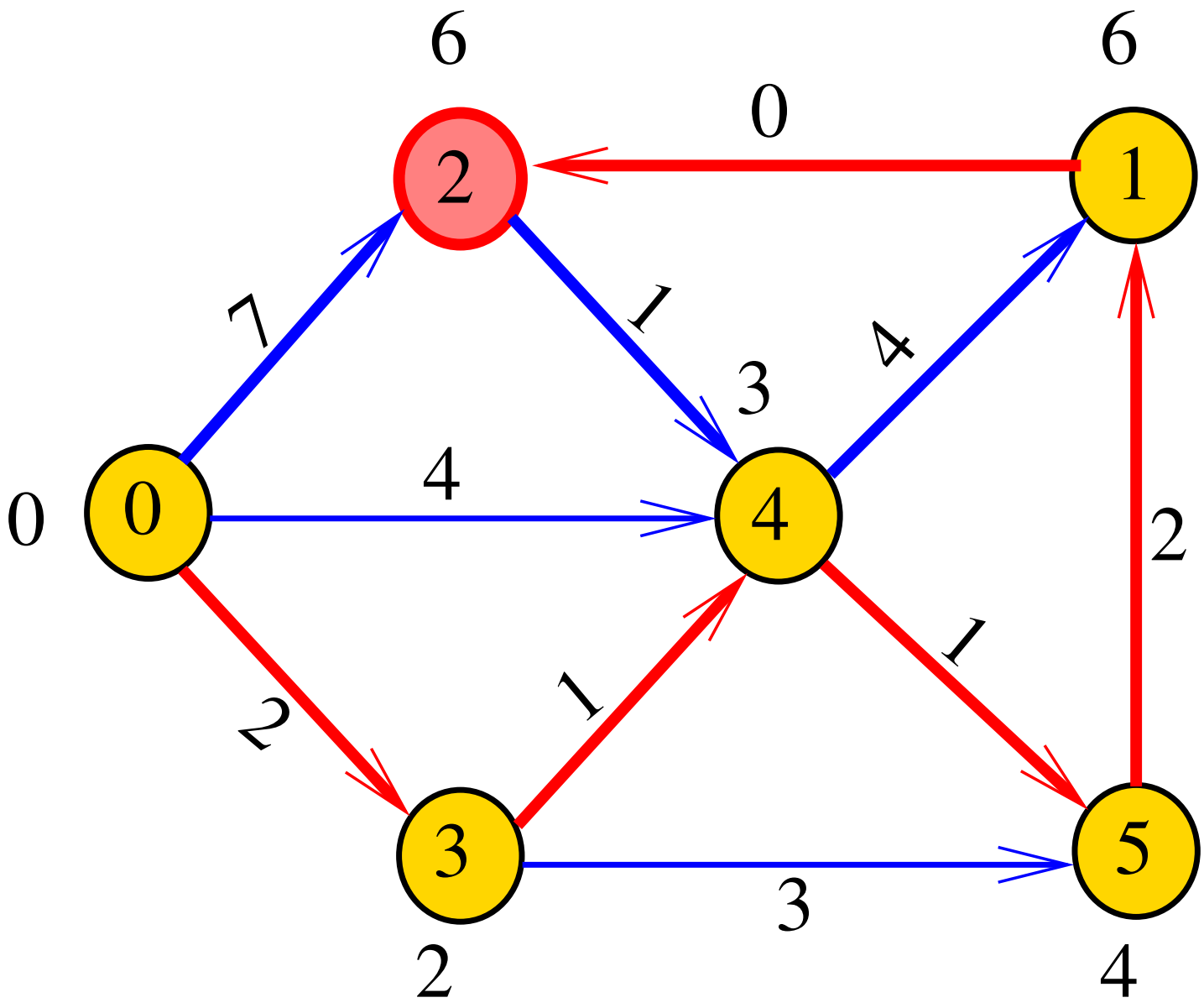
Simulação



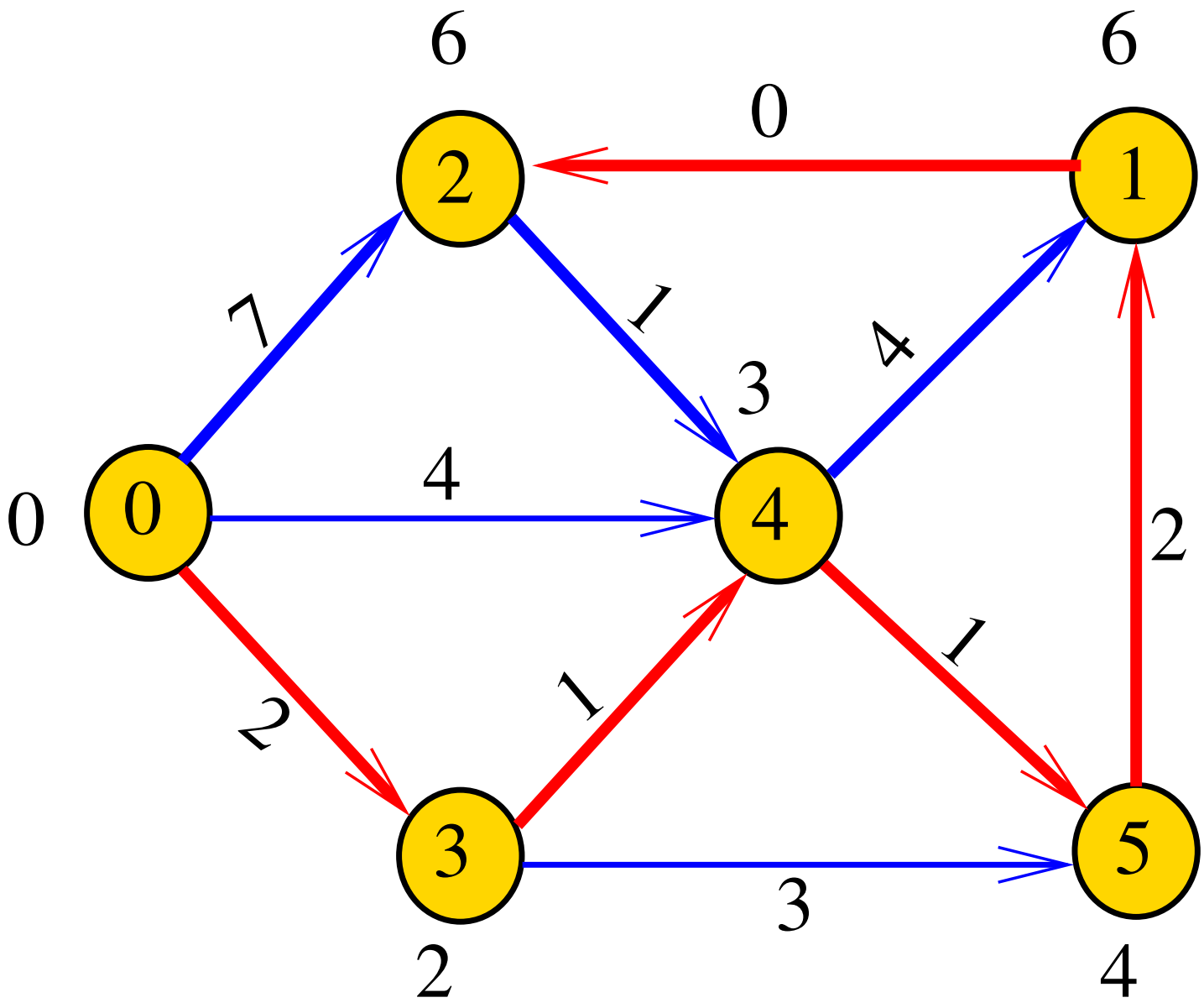
Simulação



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[]);
{
1   Vertex v, w; link p;
2   for (v = 0; v < G->V; v++) {
3       cst[v] = INFINITO;
4       parnt[v] = -1;
    }
5   PQinit(G->V);
6   cst[s] = 0;
7   parnt[s] = s;
```

dijkstra

```
 8   PQinsert(s);
 9   while (!PQempty()) {
10       v = PQdelmin();
11       for(p=G->adj[v];p!=NULL;p=p->next)
12           if (cst[w=p->w]==INFINITO) {
13               cst[w]=cst[v]+p->cst;
14               parnt[w]=v;
15               PQinsert(w);
             }
```

dijkstra

```
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();
        }
}
```

Consumo de tempo

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

Conclusão

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

1	execução de	`PQinit` e `PQfree`,
$O(V)$	execuções de	`PQinsert`,
$O(V)$	execuções de	`PQempty`,
$O(V)$	execuções de	`PQdelmin`, e
$O(A)$	execuções de	`PQdec`.

Implementação para digrafos densos

```
/* Item.h */
typedef Vertex Item;

/* QUEUE.h */
void PQinit(int);
int PQempty();
void PQinsert(Item);
Item PQdelmin();
void PQdec(Item);
void PQfree();
```

PQinit e PQempty

```
Item *q;
int  inicio, fim;

void PQinit(int maxN) {
   q=(Item*)malloc(maxN*sizeof(Item));
   inicio = 0;
   fim = 0;
}

int PQempty() {
   return inicio==fim;
}
```

PQinsert e PQdelmin

```
void PQinsert(Item item){
   q[fim++] = item;
}

Item PQdelmin() {
   int i, j; Item x;
   i= inicio;
   for (j=i+1; j < fim; j++)
       if (cst[q[i]] > cst[q[j]]) i = j;
   x = q[i];
   q[i] = q[--fim];
   return x;
}
```

PQdec e PQfree

```
void PQdec(Vertex v) {
/* faz nada */
}

void PQfree() {
   free(q);
}
```

Consumo de tempo Min-Heap

`PQinit`	$\Theta(1)$
`PQempty`	$\Theta(1)$
`PQinsert`	$\Theta(1)$
`PQdelmin`	$\mathrm{O}(V)$
`PQdec`	$\Theta(1)$
`PQfree`	$\Theta(1)$

Conclusão

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

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

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.

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.