
Fonte: ash.atozviews.com

# Compacto de alguns dos melhores momentos

## AULA 20

# Busca ou varredura

Um algoritimo de **busca** (ou **varredura**) examina, sistematicamente, todos os vértices e todos os arcos de um digrafo.

Cada arco é examinado **uma só vez**.
Despois de visitar sua ponta inicial o algoritmo percorre o arco e visita sua ponta final.

# Busca em largura

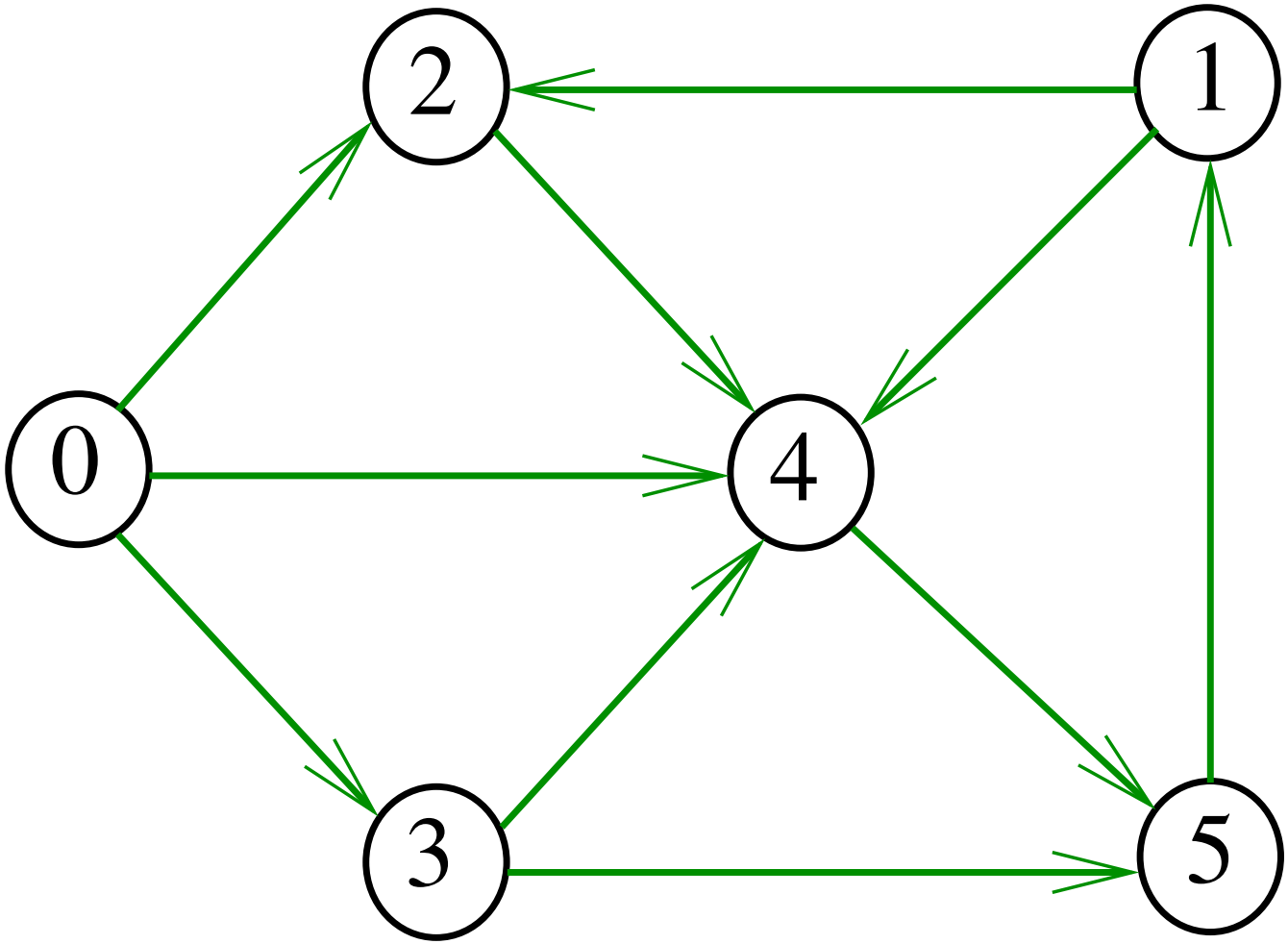
A **busca em largura** (=*breadth-first search search* = *BFS*) começa por um vértice, digamos `s`, especificado pelo usuário.

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*

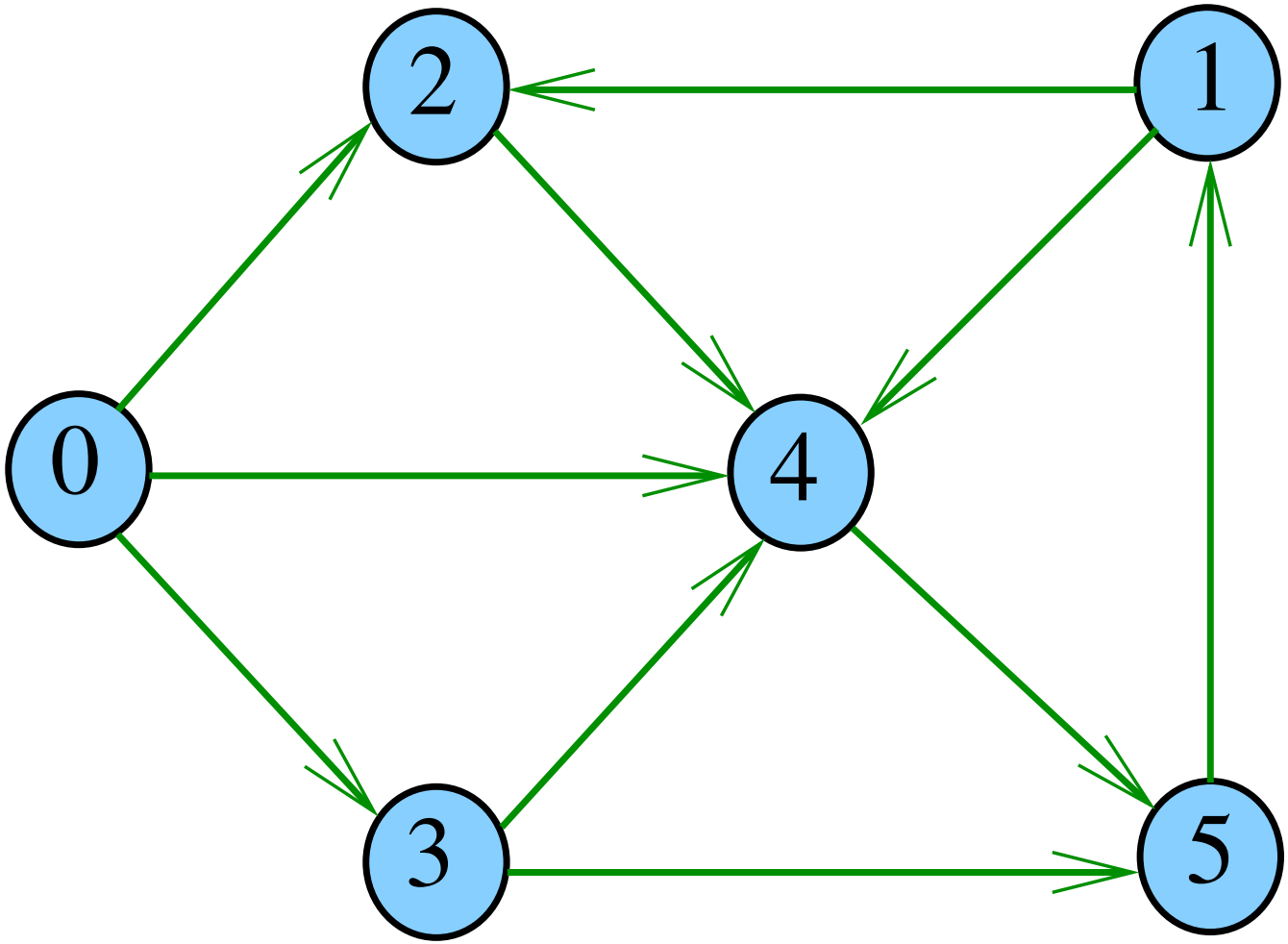
# Simulação

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



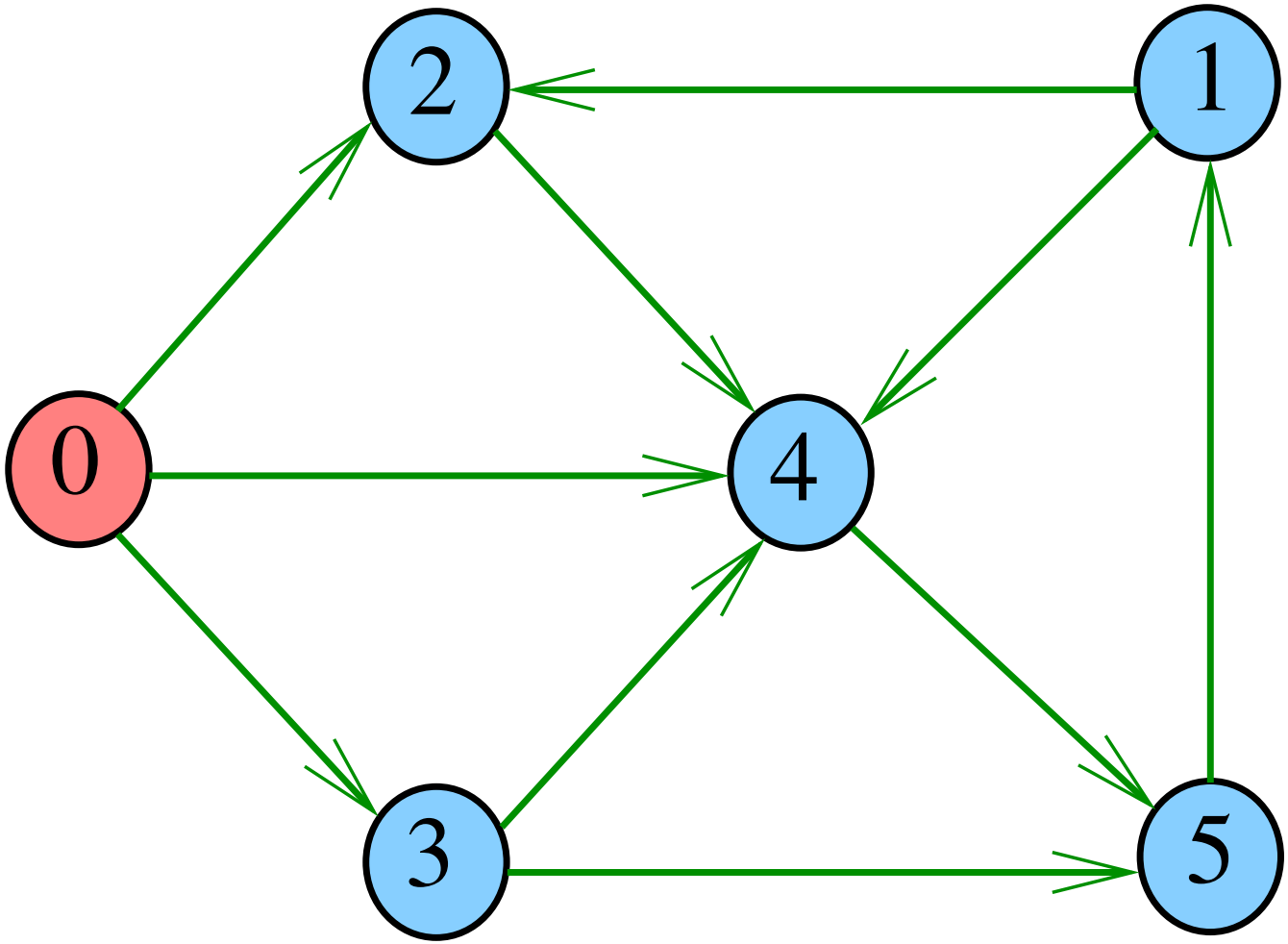
# Simulação

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



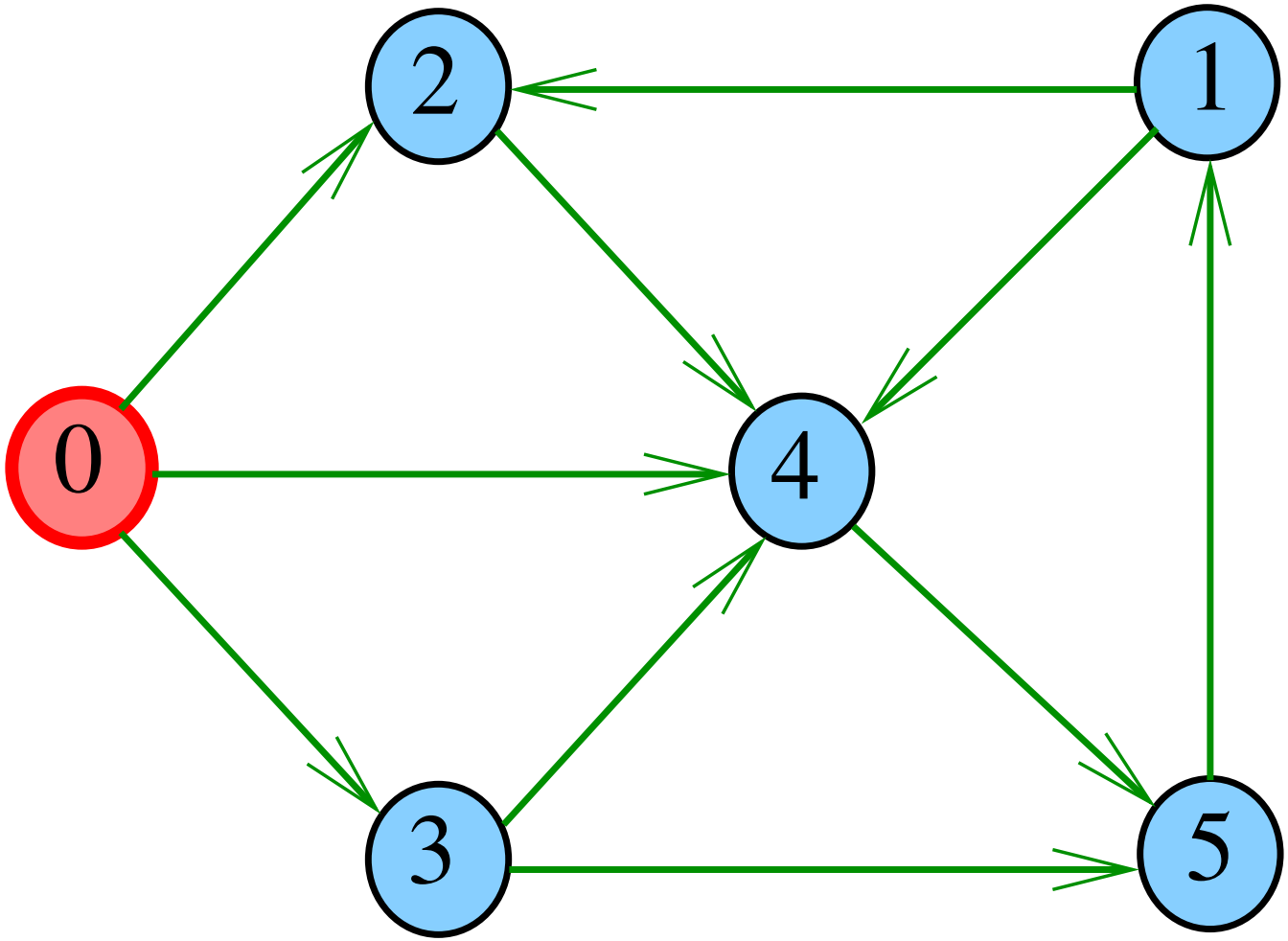
# Simulação

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



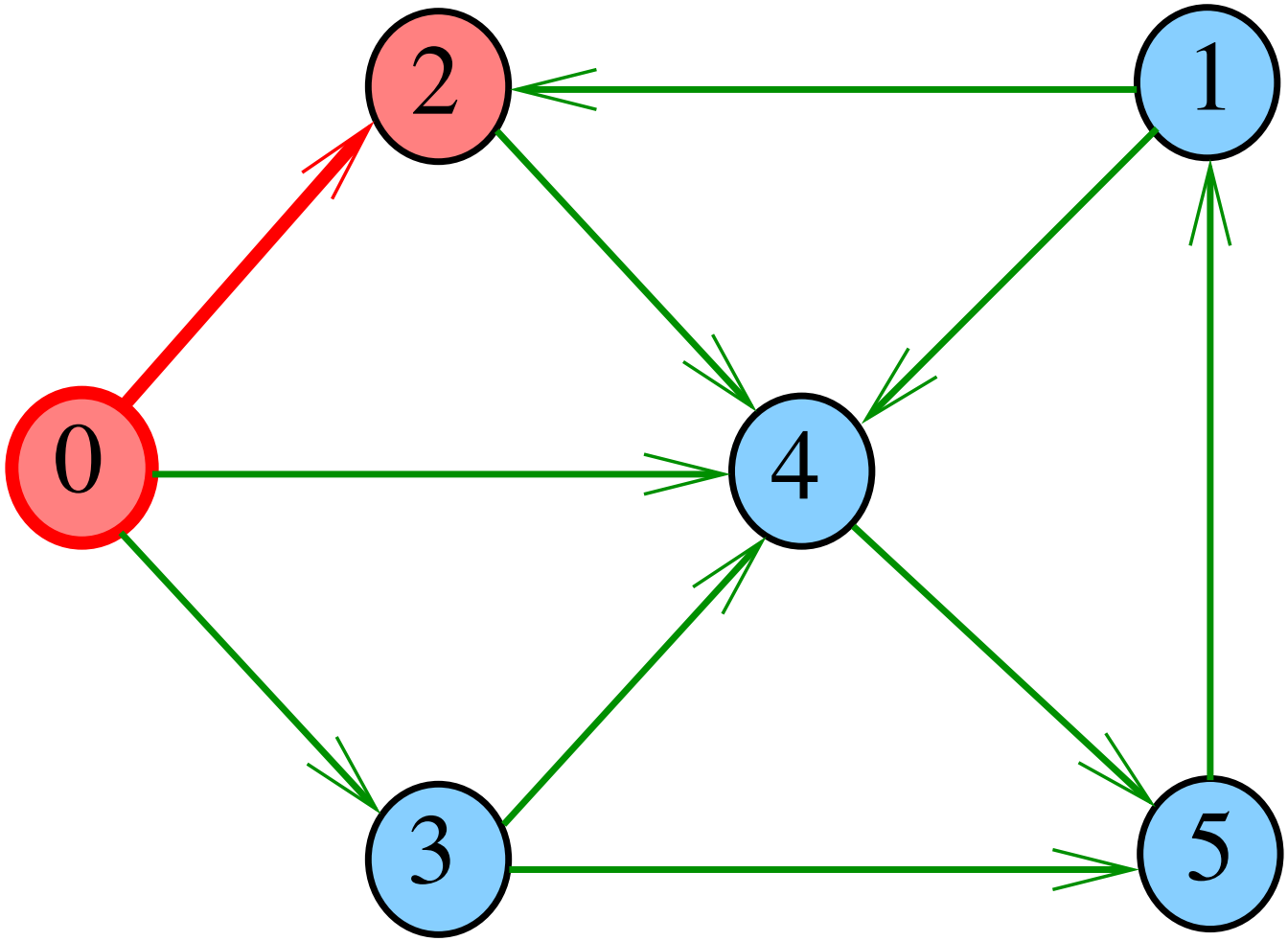
# Simulação

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



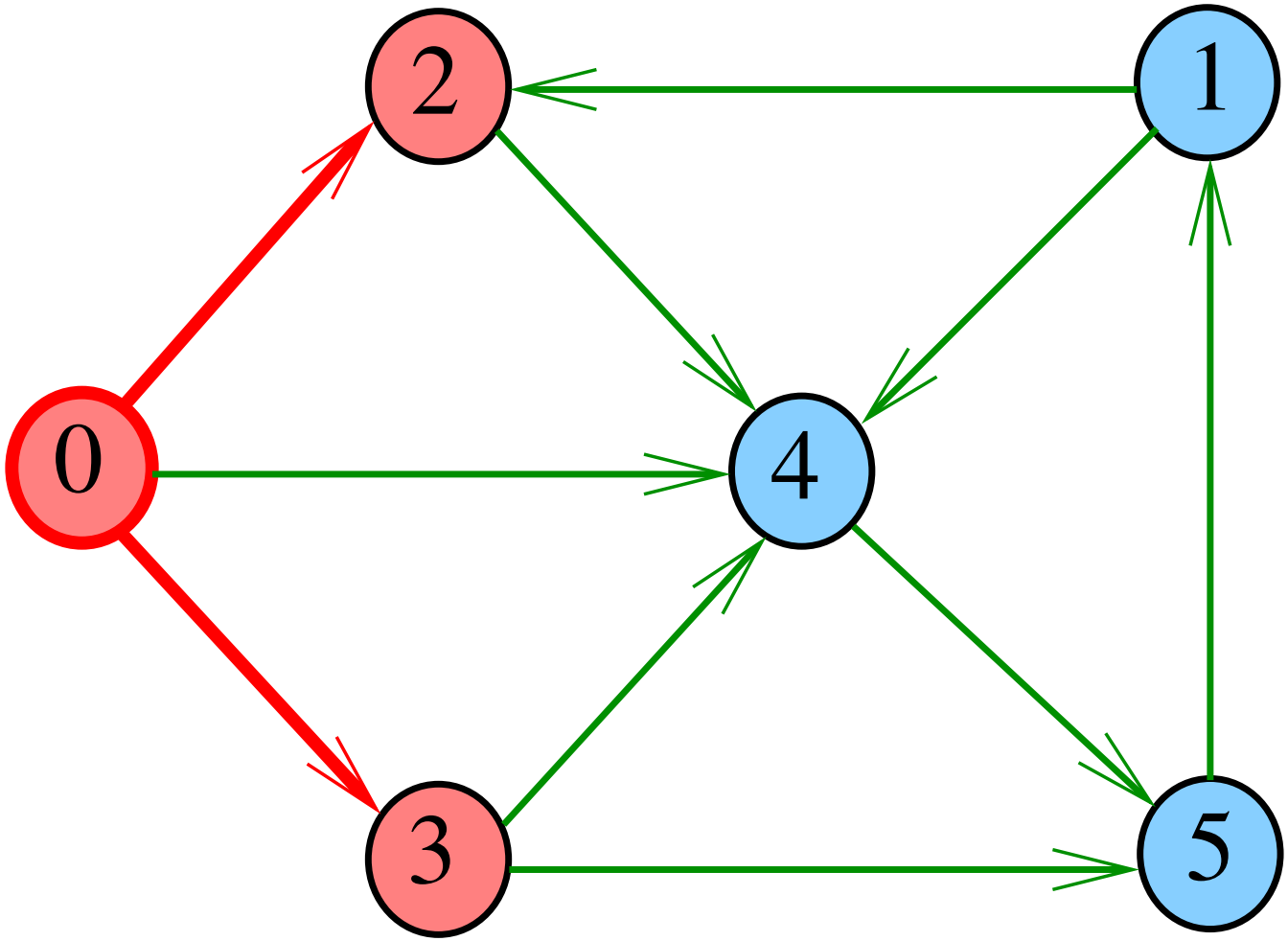
# Simulação

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



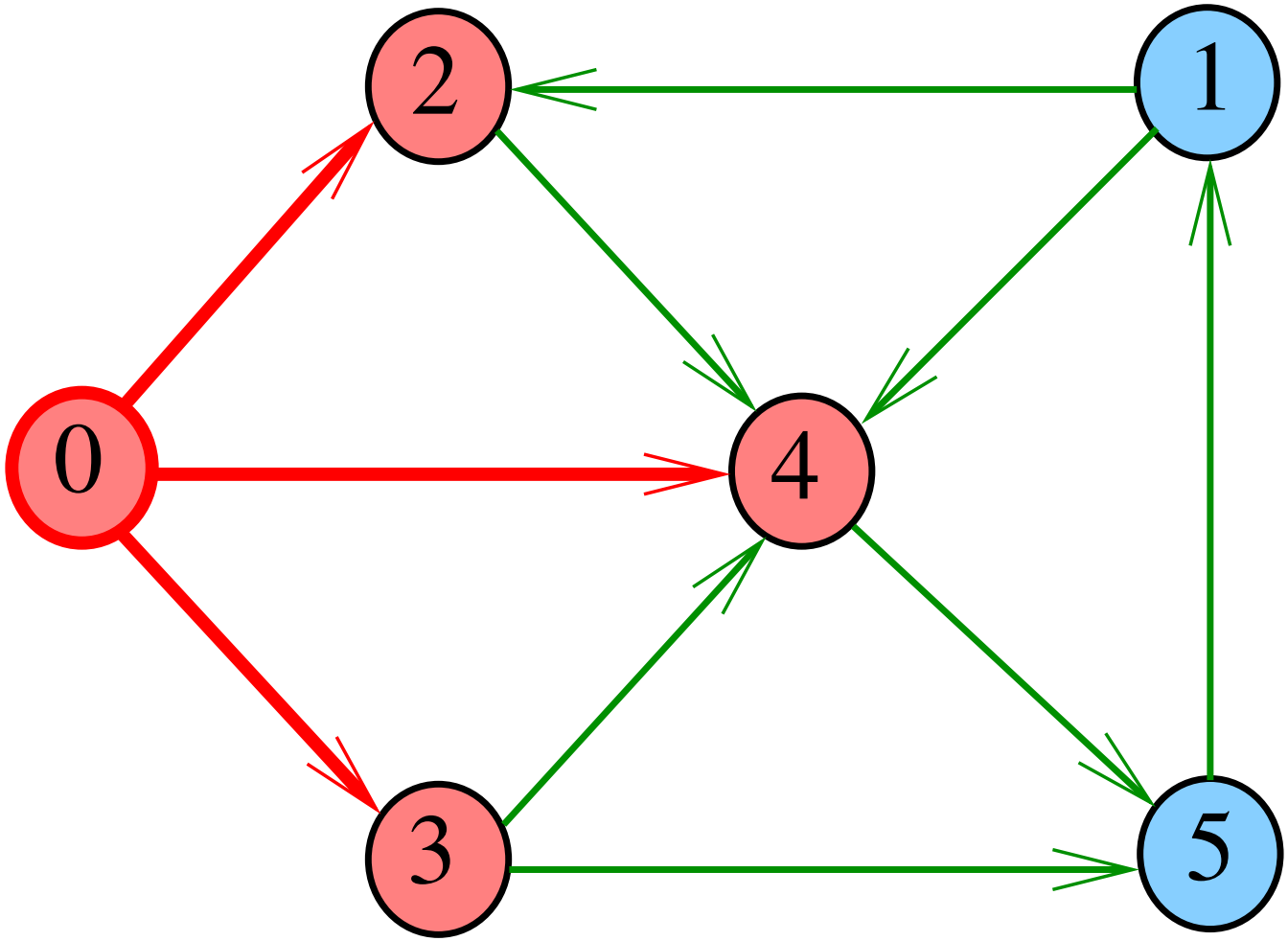
# Simulação

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



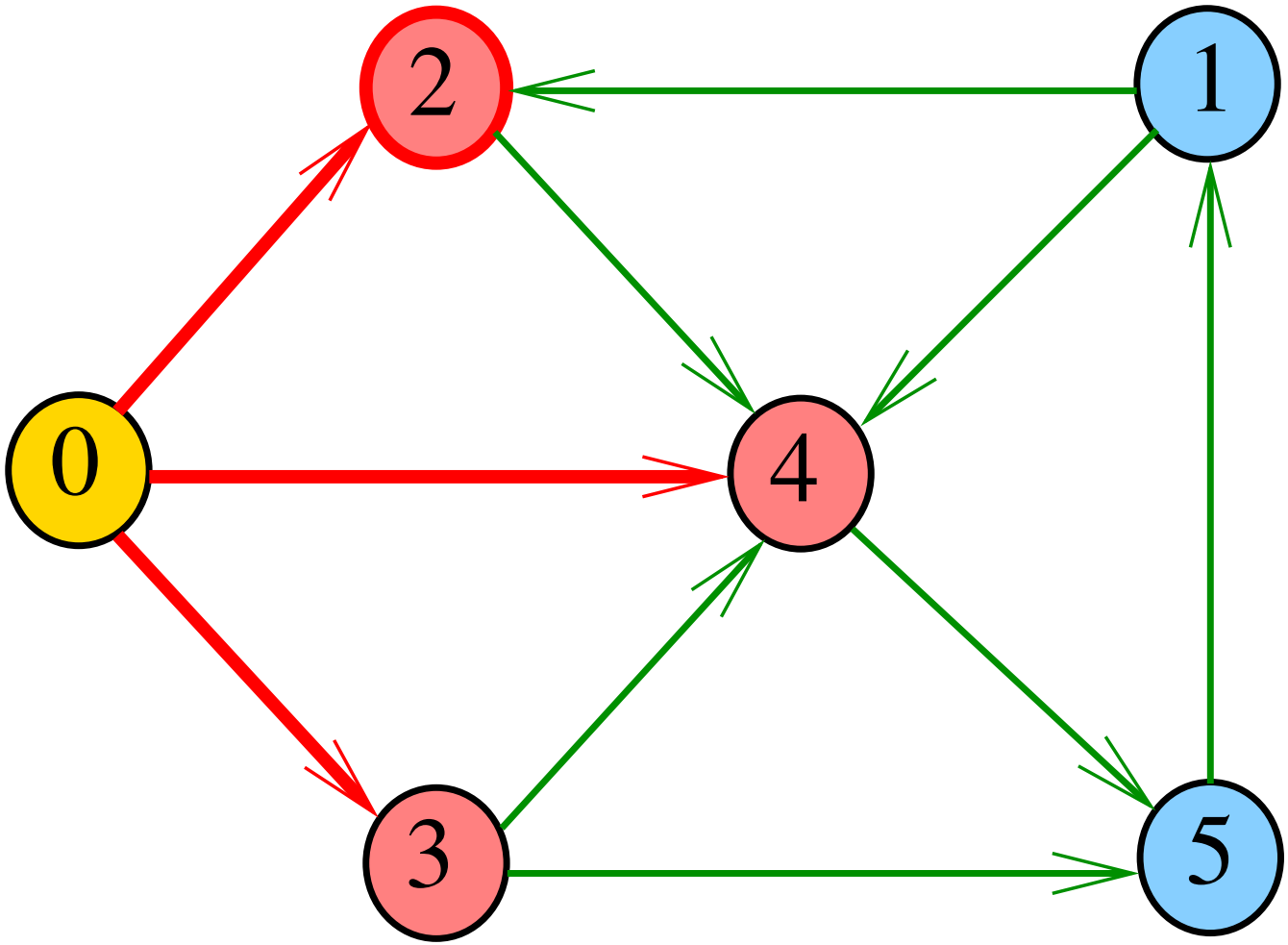
# Simulação

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



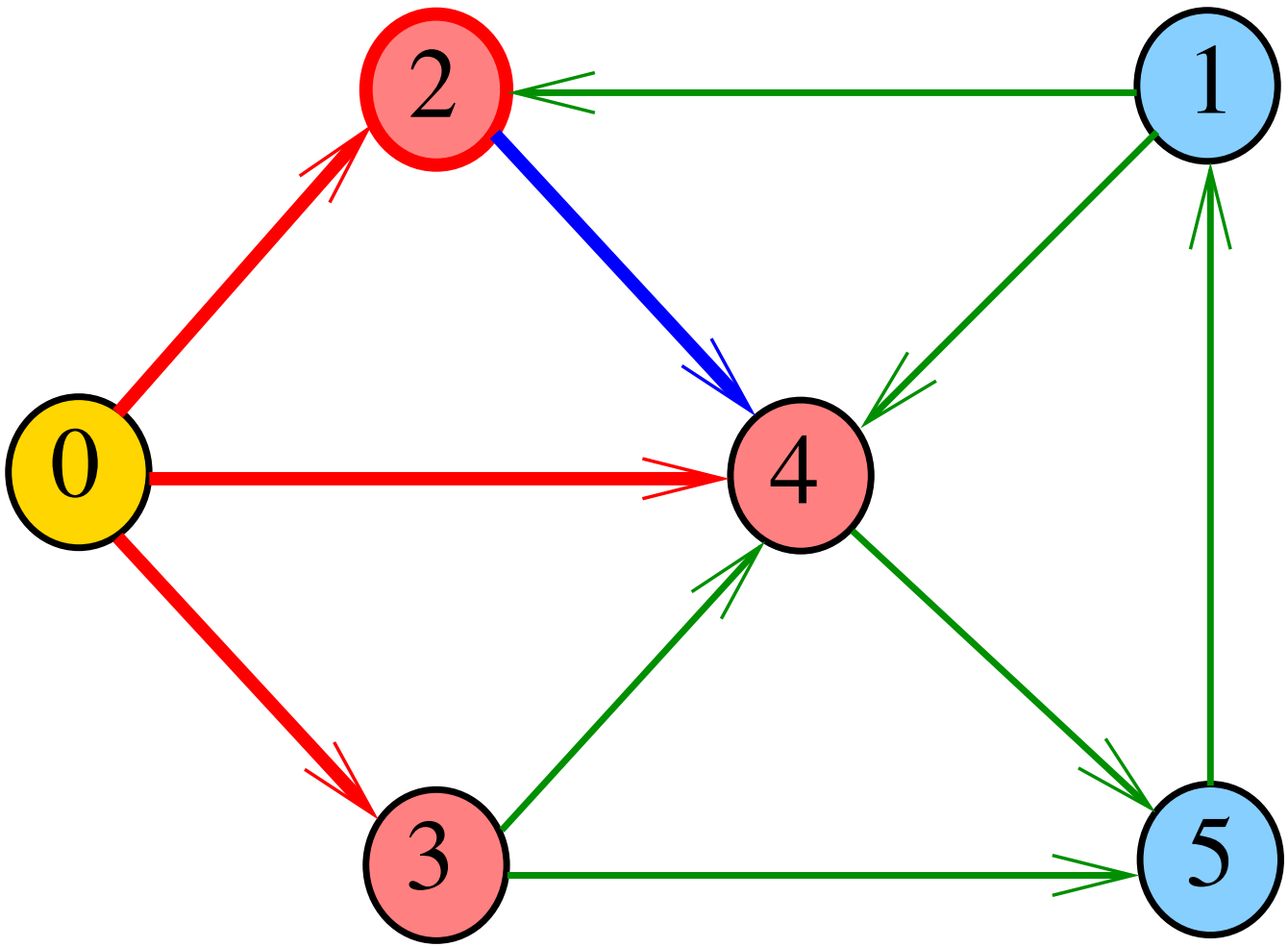
# Simulação

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



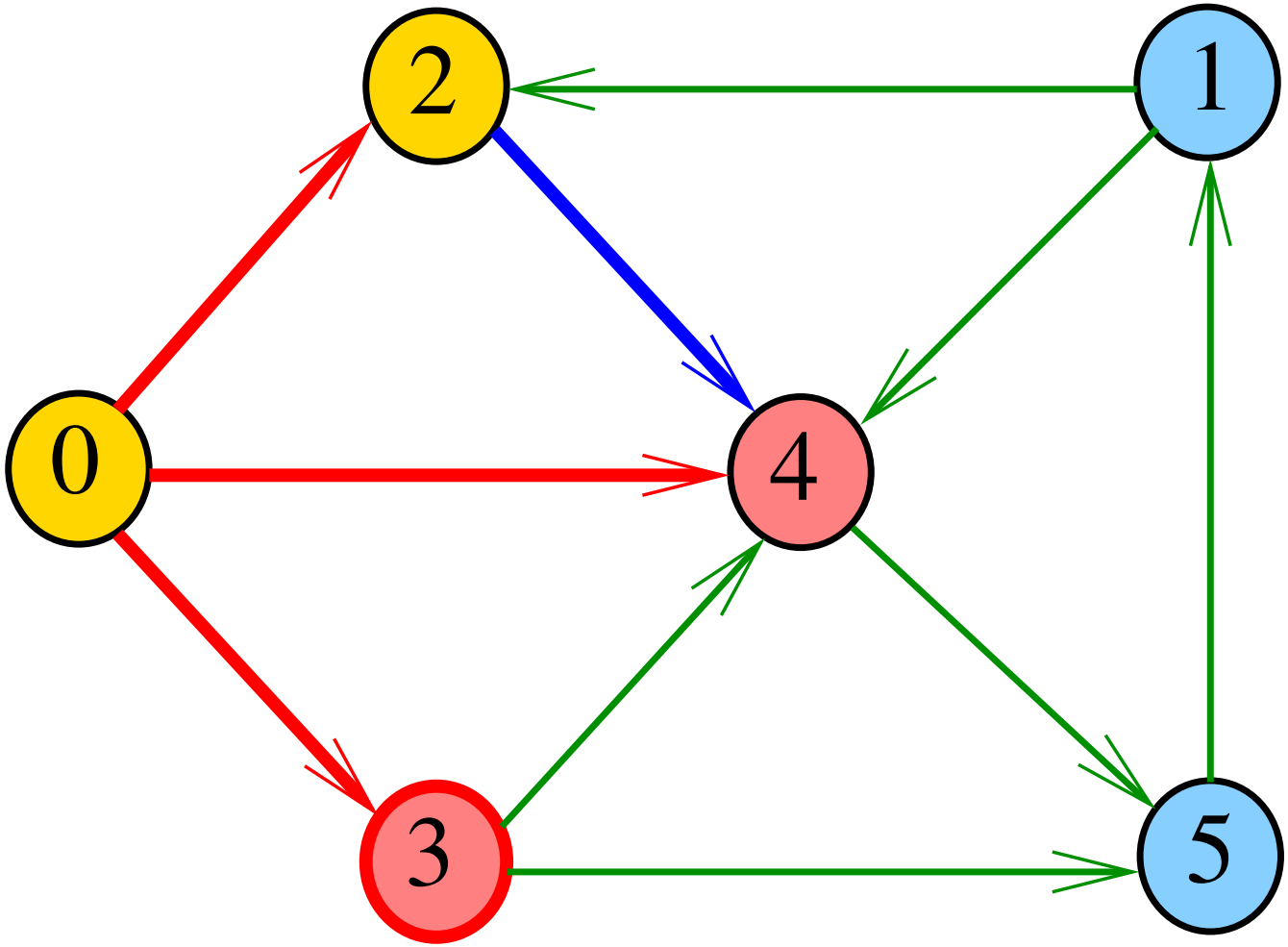
# Simulação

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



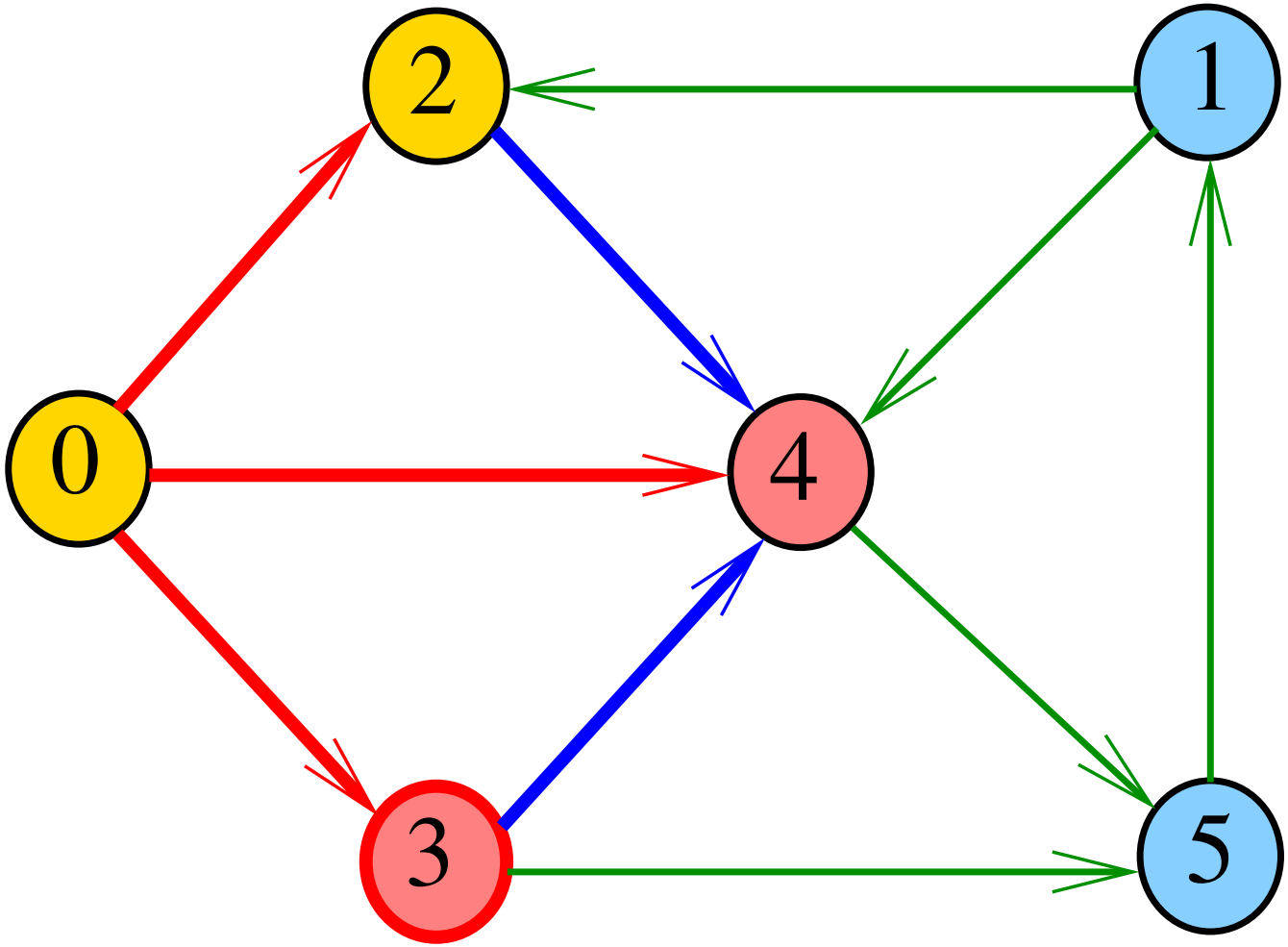
# Simulação

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



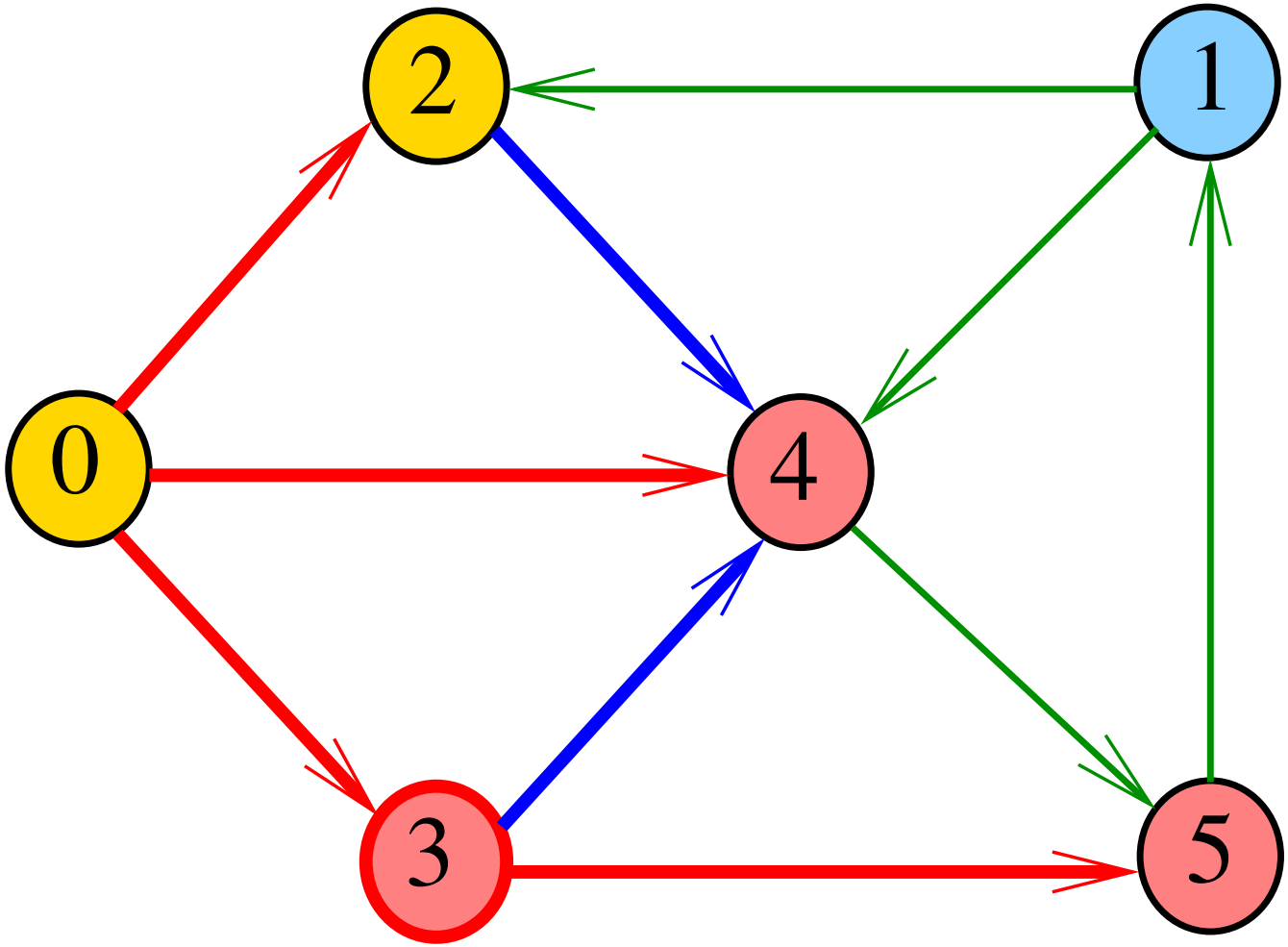
# Simulação

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



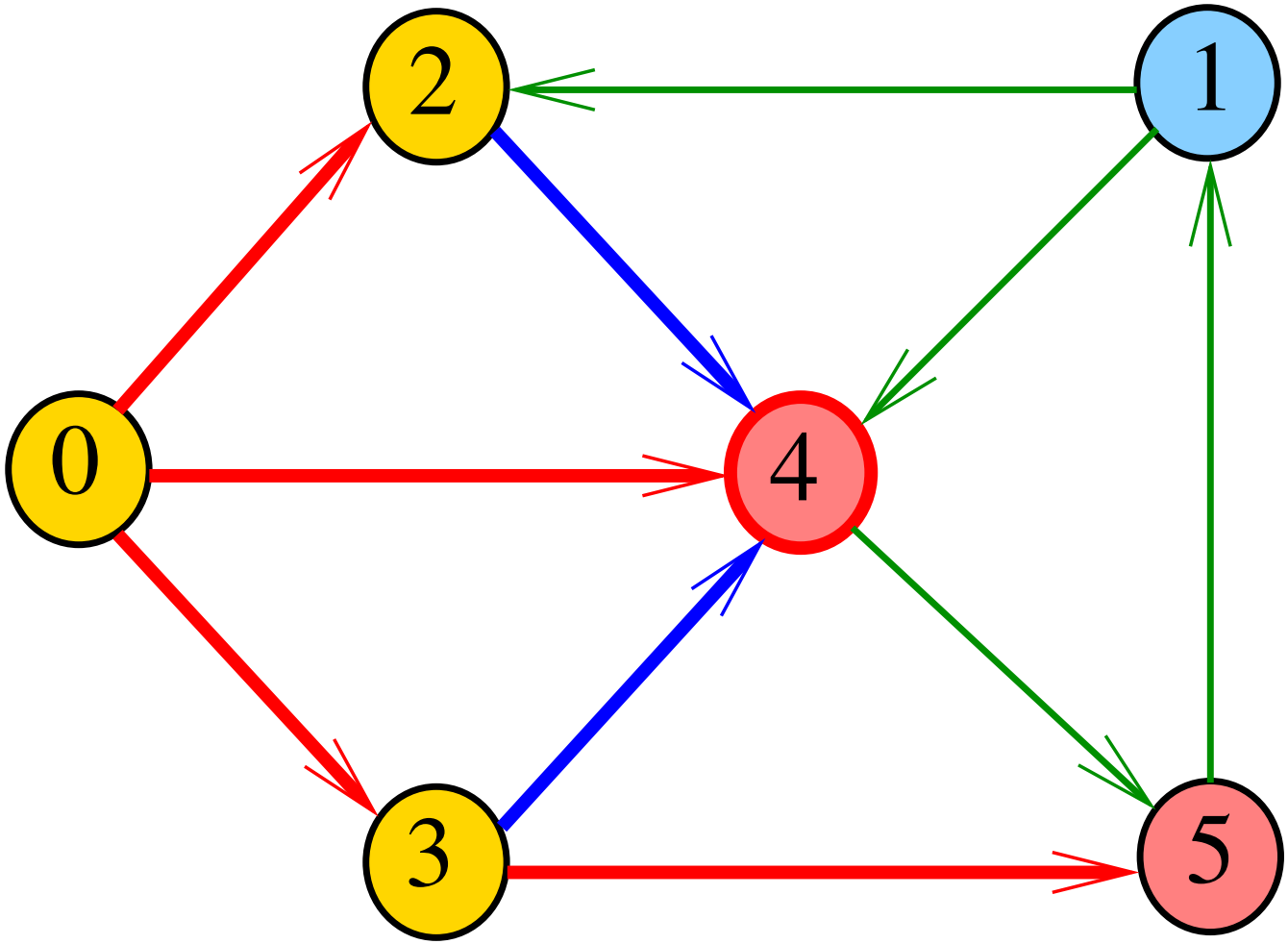
# Simulação

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



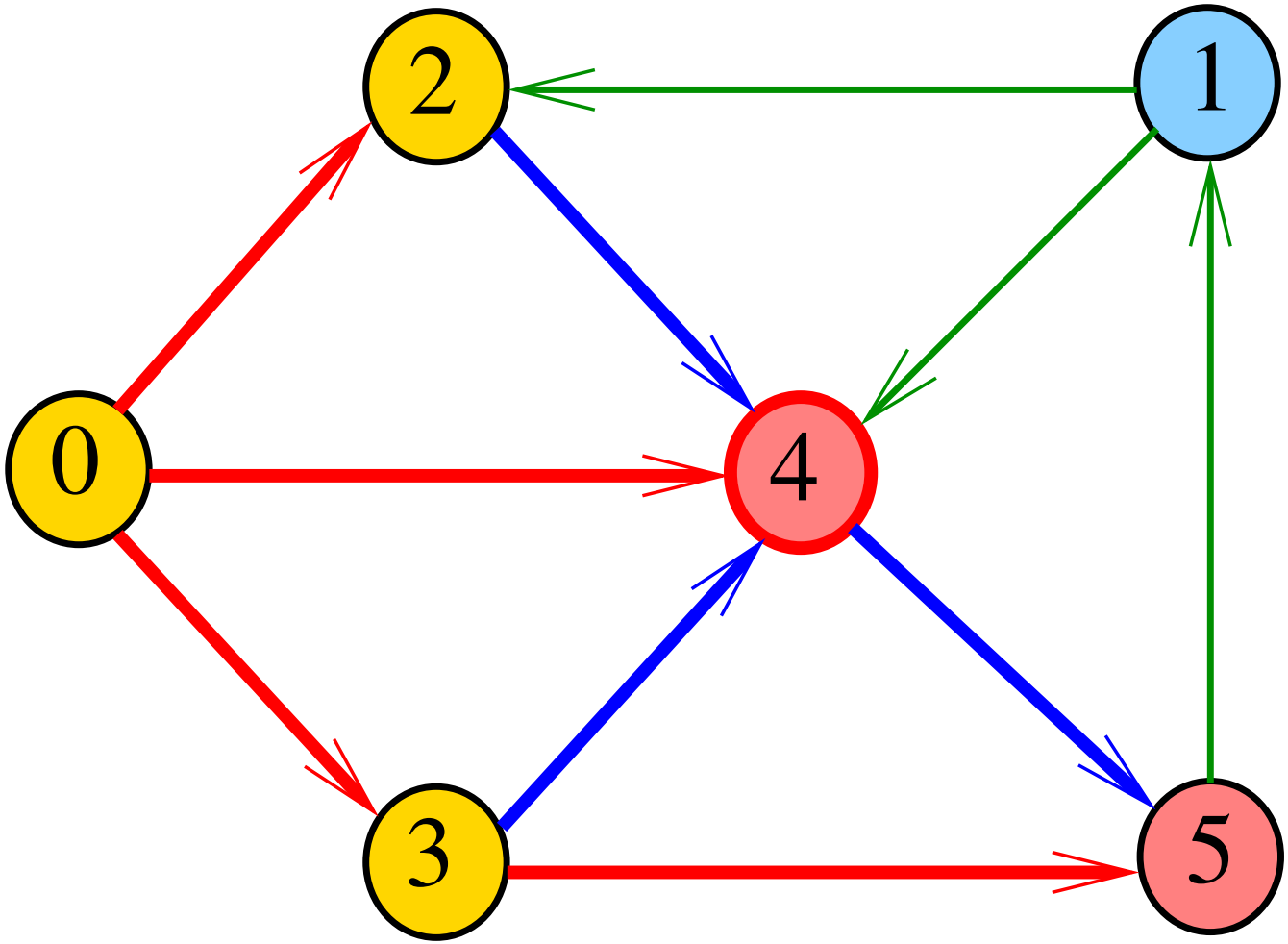
# Simulação

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



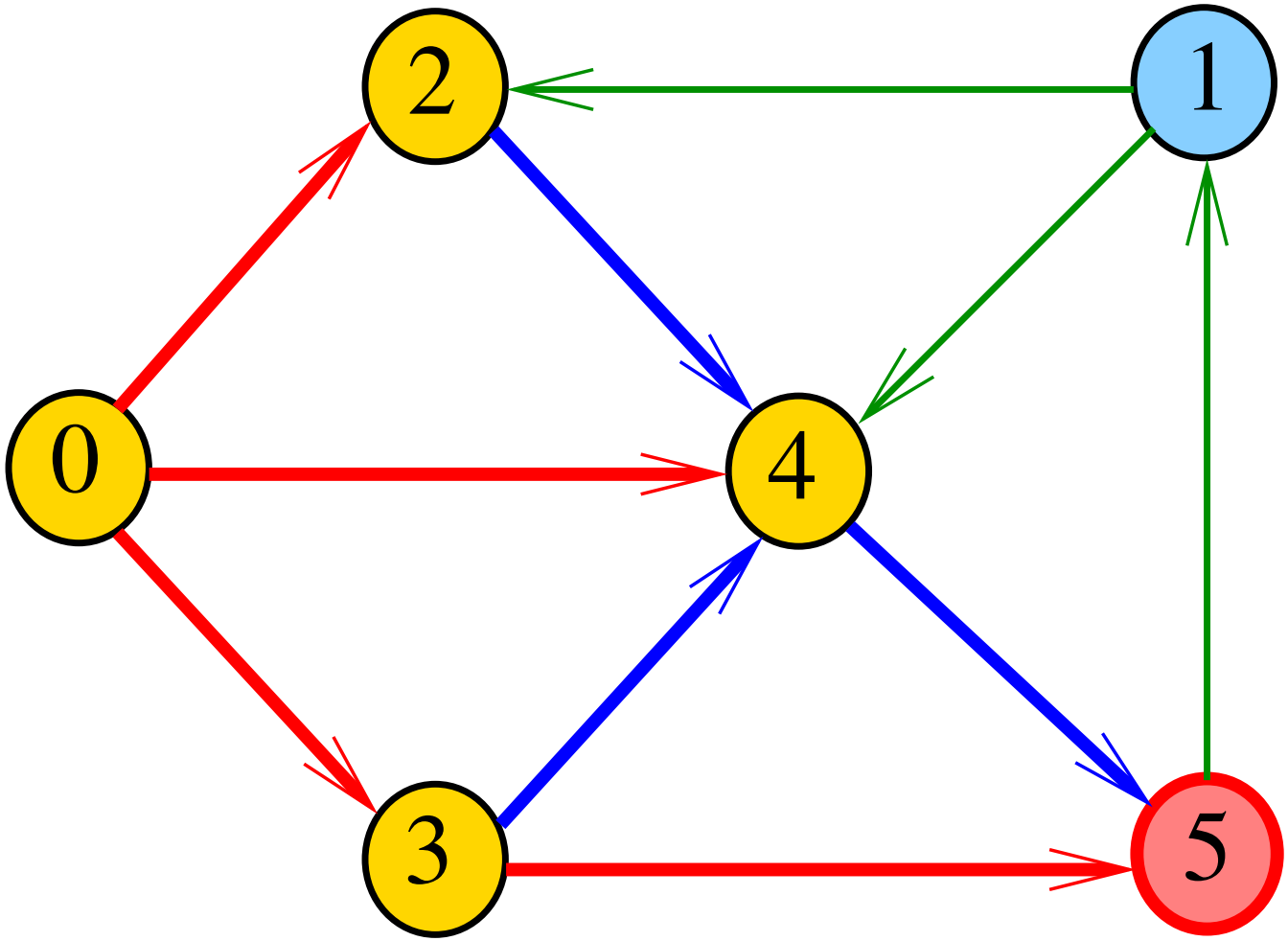
# Simulação

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



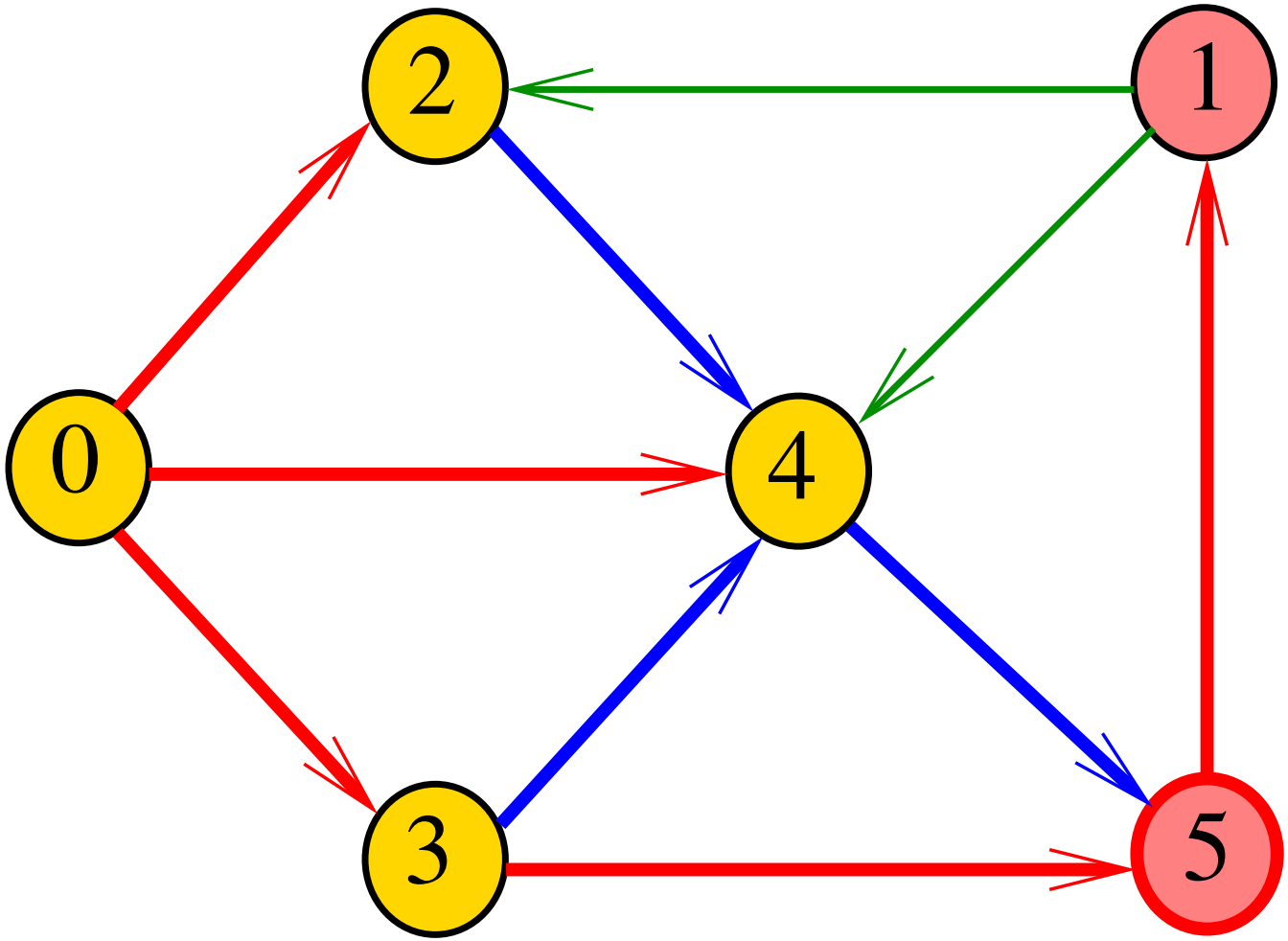
# Simulação

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



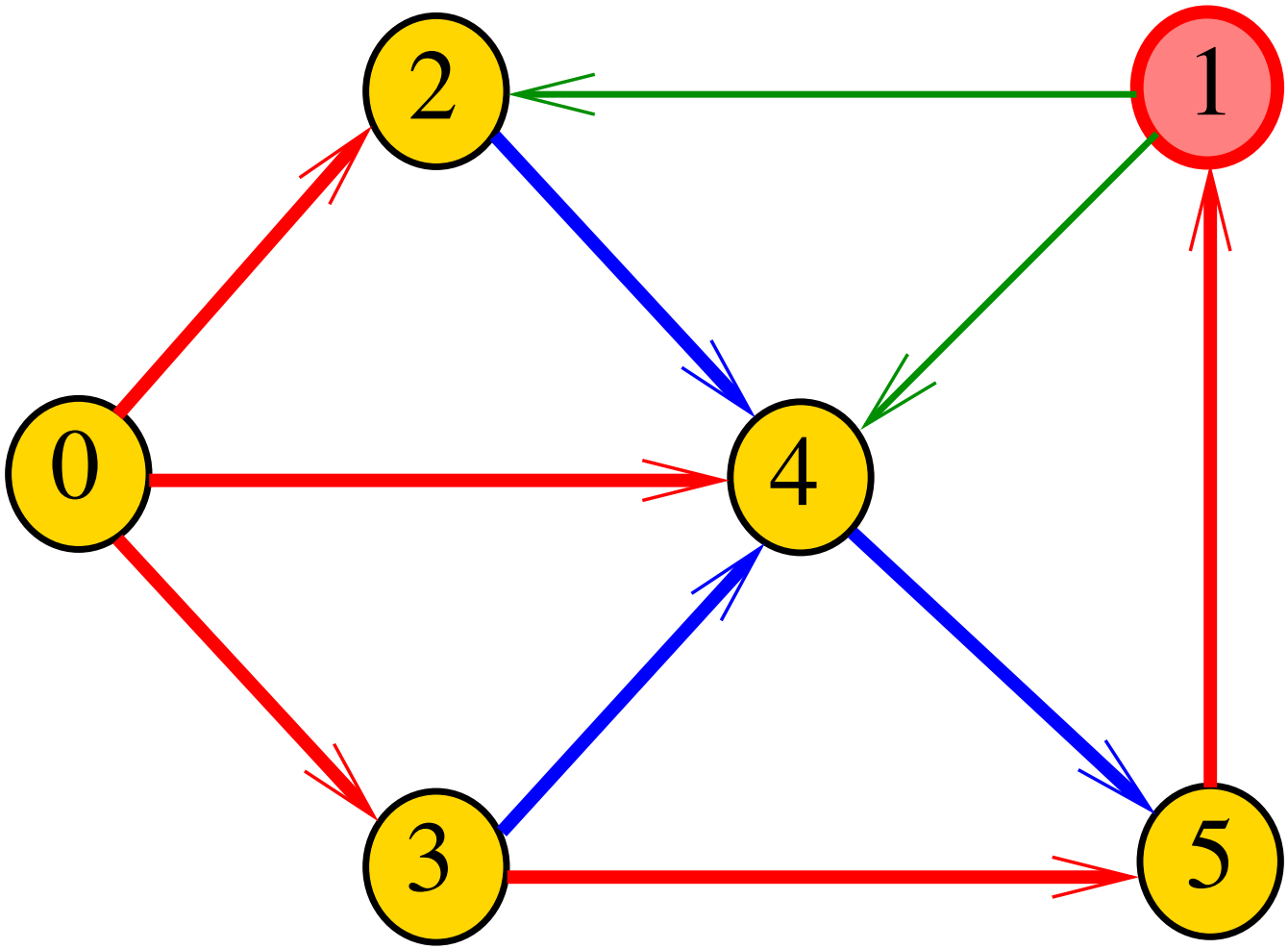
# Simulação

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



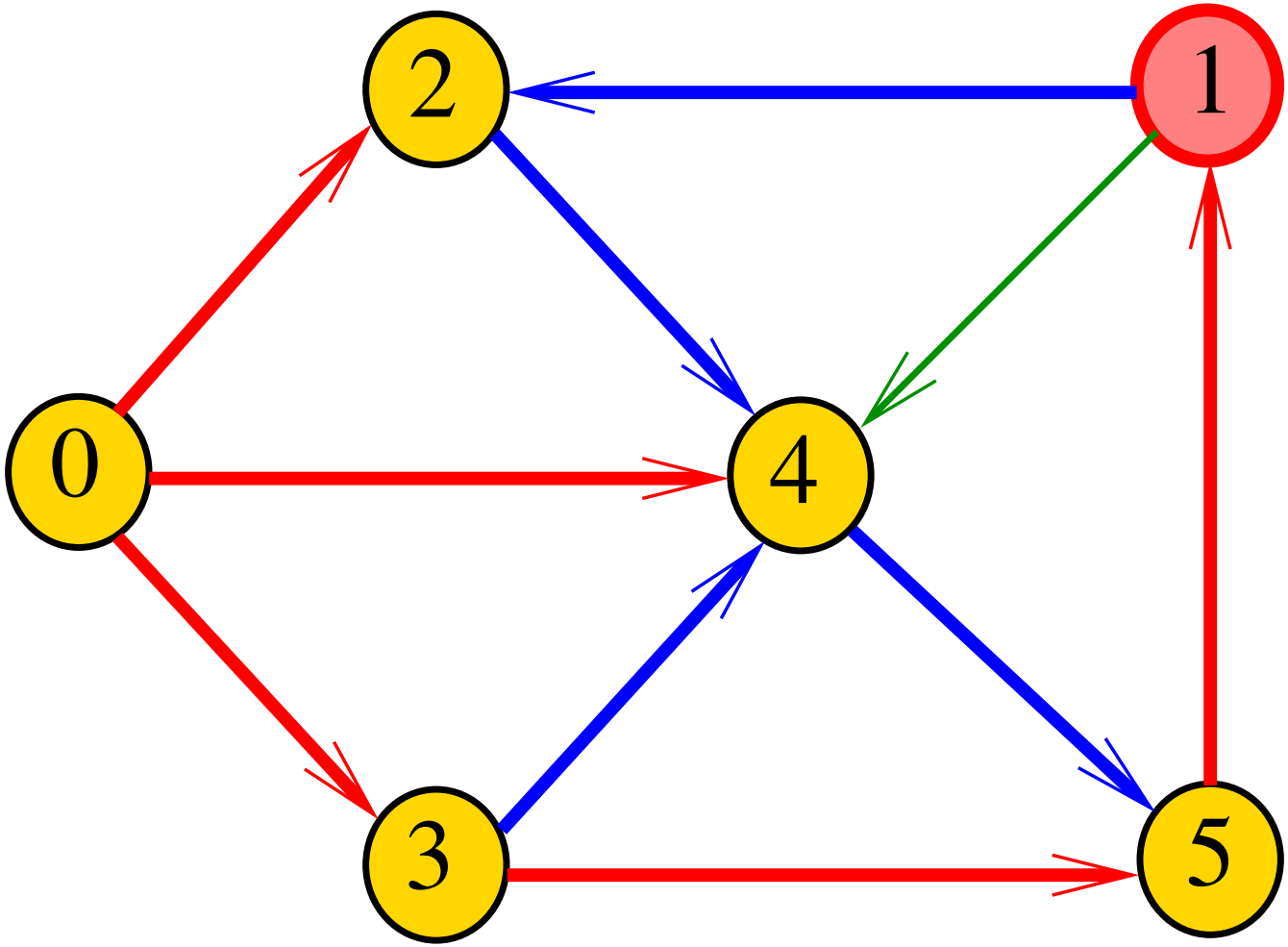
# Simulação

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



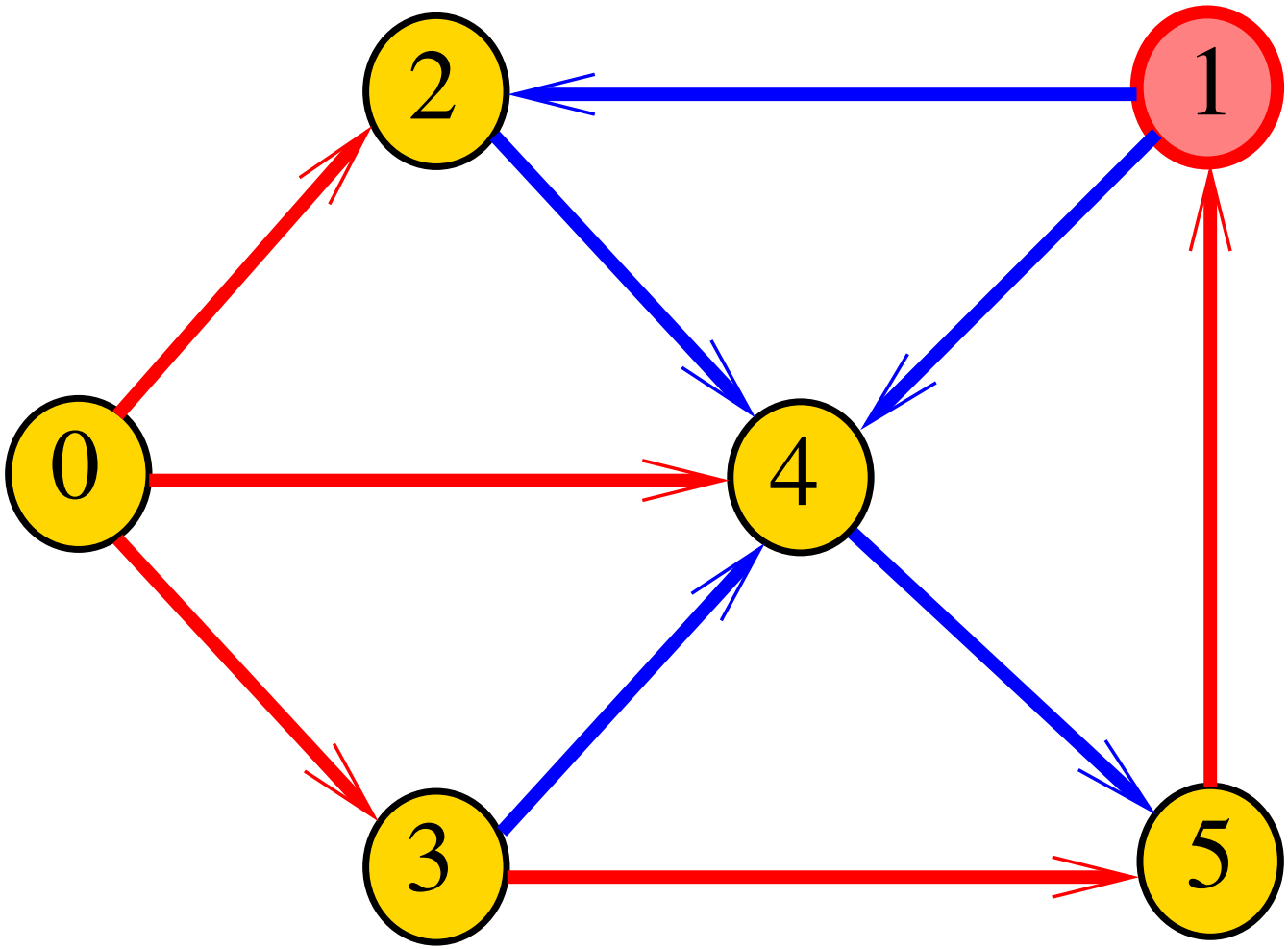
# Simulação

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



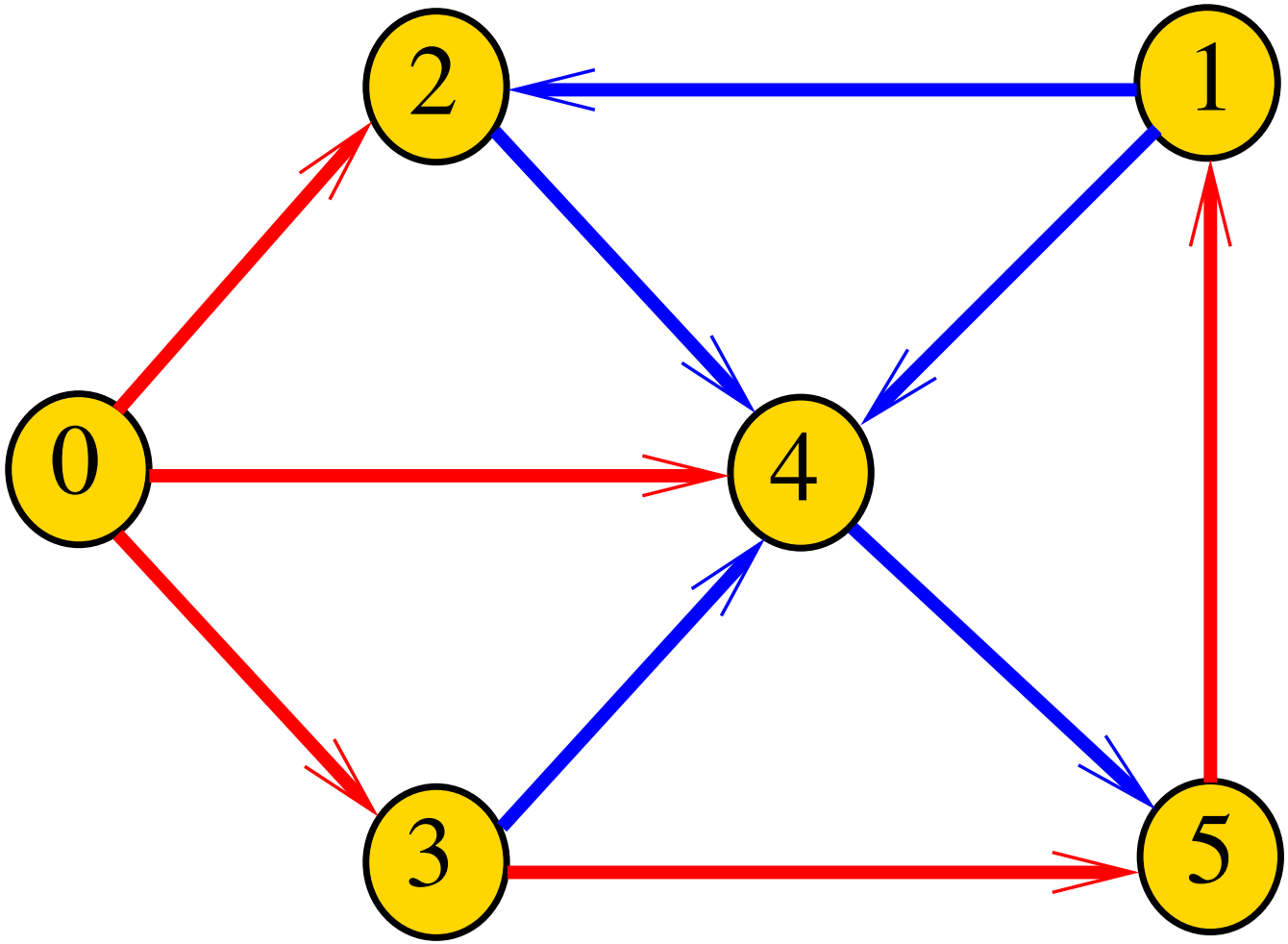
# Simulação

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



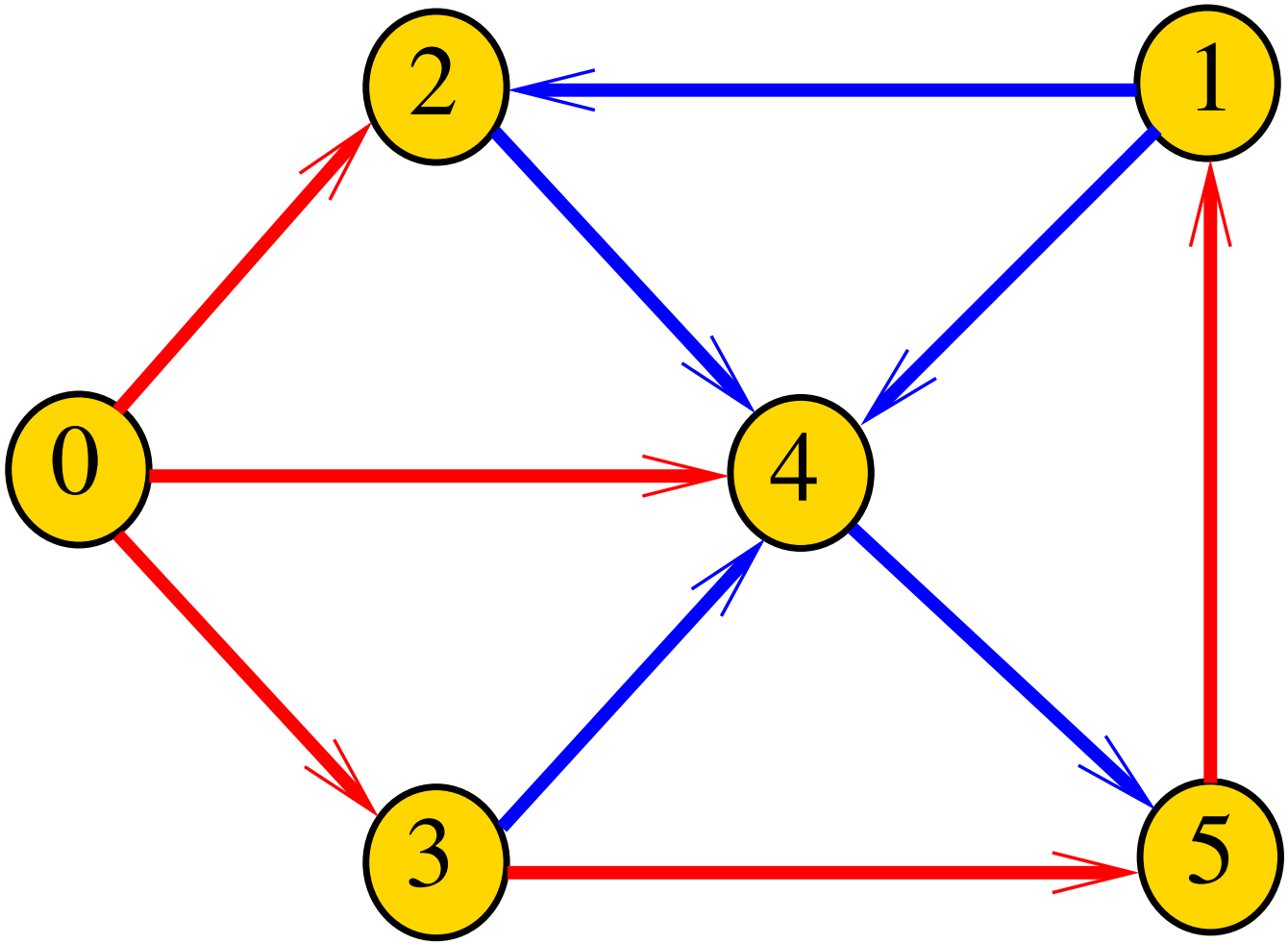
# Simulação

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



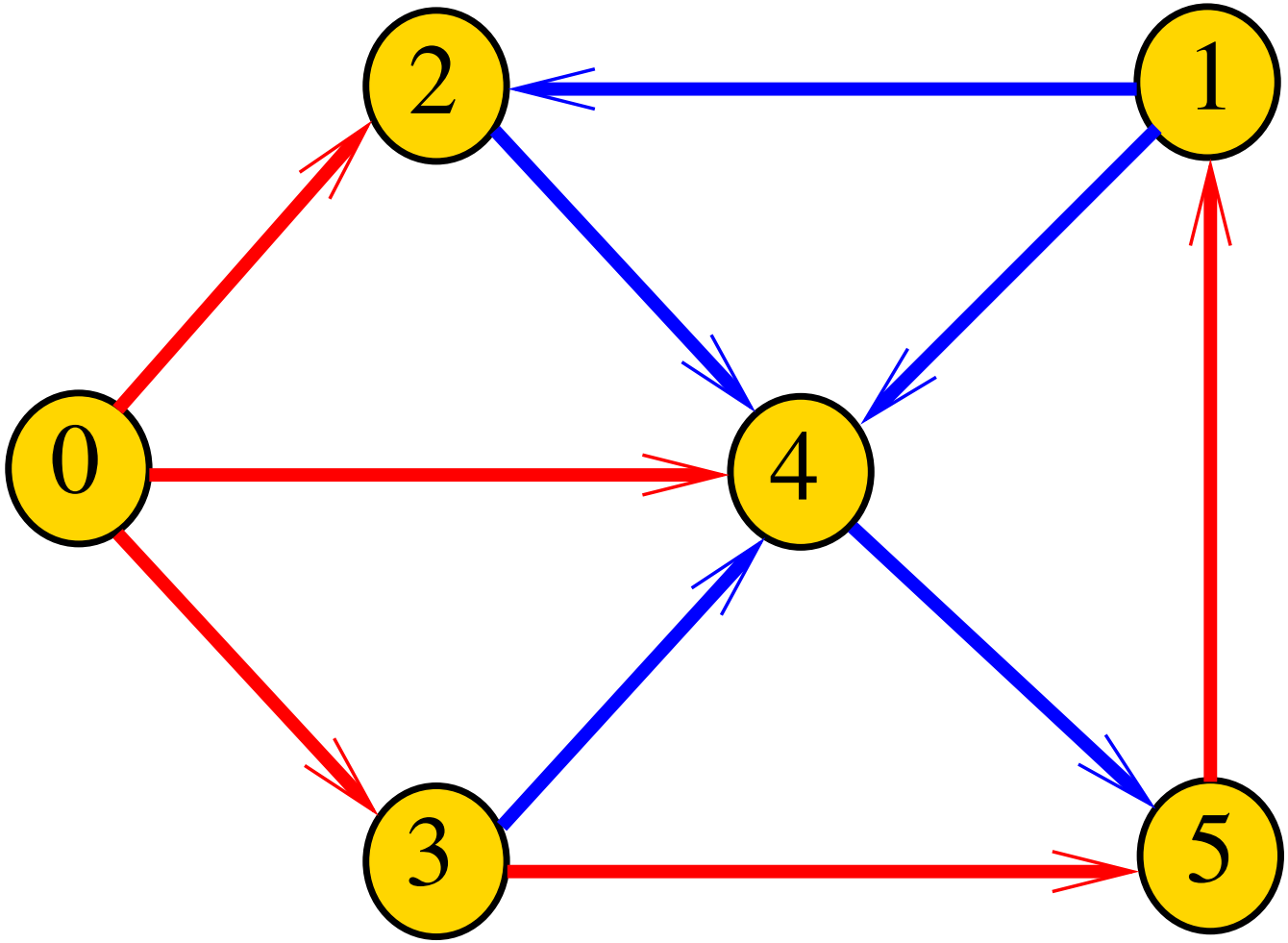
# Arborescência da BFS

A busca em largura a partir de um vértice `s` descreve a arborescência com raiz `s`



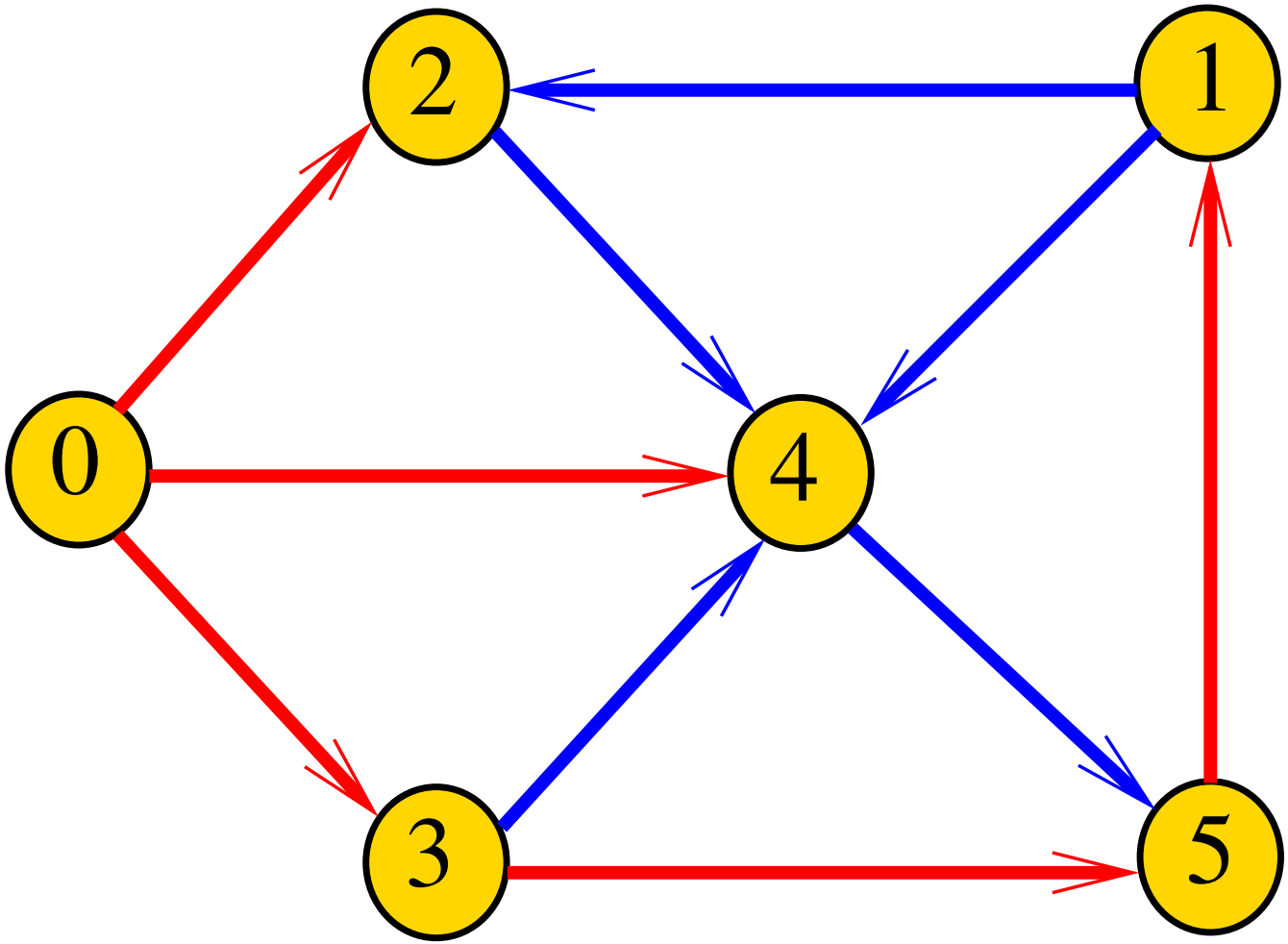
# Arborescência da BFS

Essa arborescência é conhecida como
**arborescência de busca em largura** (= *BFS tree*)



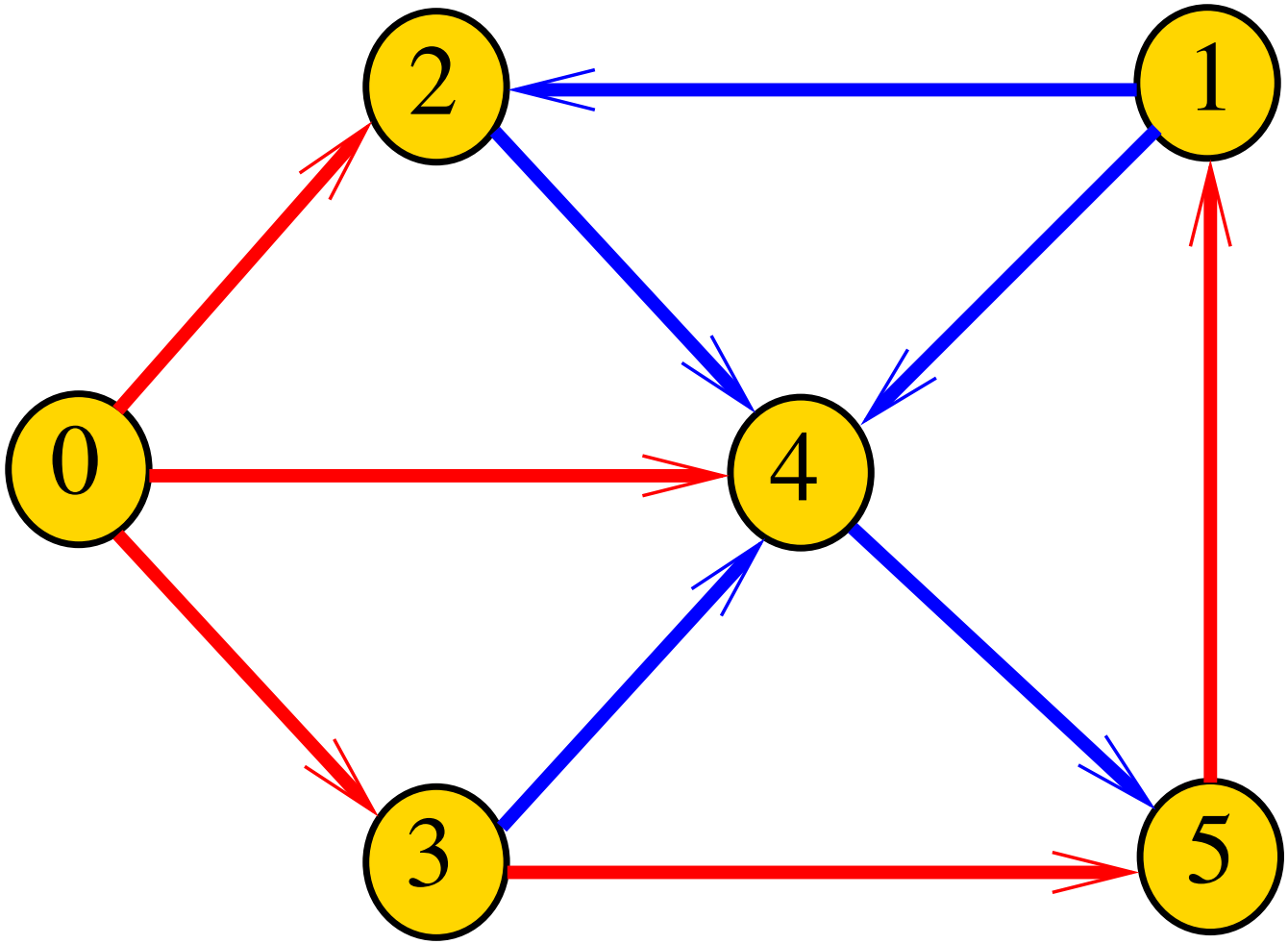
# Representação da BFS

Podemos representar essa arborescência explicitamente por um vetor de pais `edgeTo[]`



# Representação da BFS

| v | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| edgeTo | 0 | 5 | 0 | 0 | 0 | 3 |



# Consumo de tempo

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

O consumo de tempo da função `BFSpaths`para matriz de adjacência é $O(V^2)$.

# AULA 21

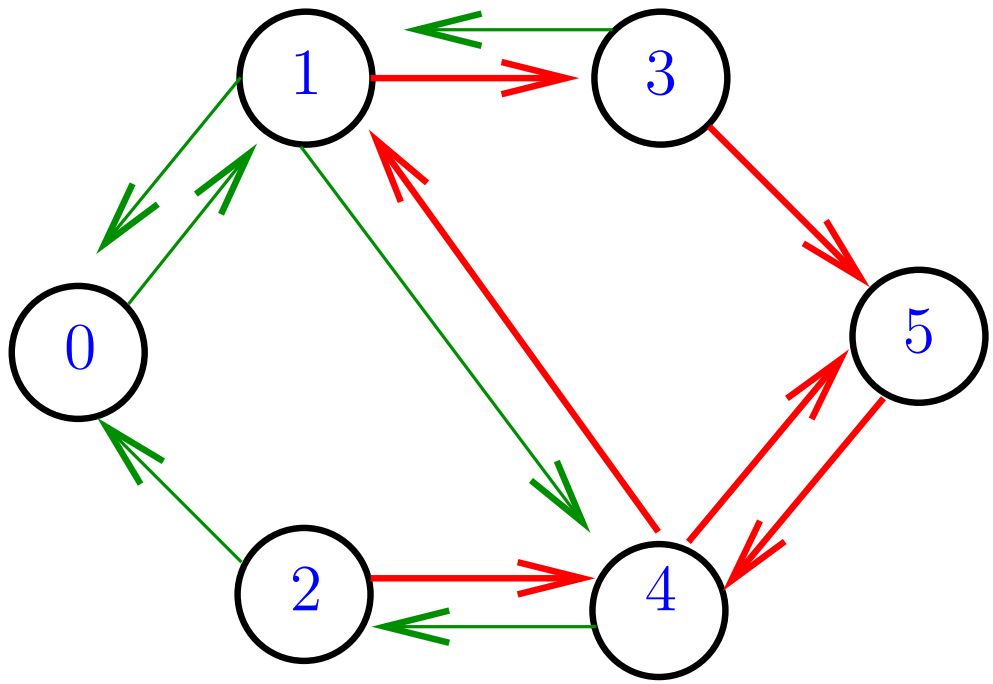
# Caminhos mínimos


Fonte: http://vandanasanju.blogspot.com.br/

# Comprimento

O **comprimento** de um caminho é o número de arcos no caminho, contanto-se as repetições

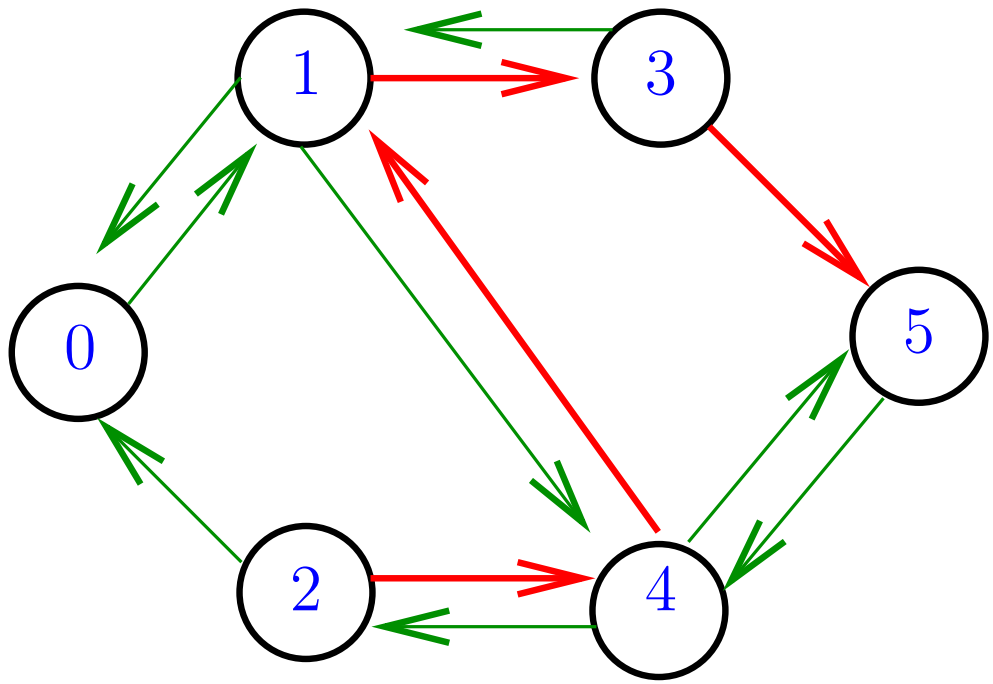
Exemplo: 2-4-1-3-5-4-5 tem comprimento 6



# Comprimento

O **comprimento** de um caminho é o número de arcos no caminho, contanto-se as repetições.

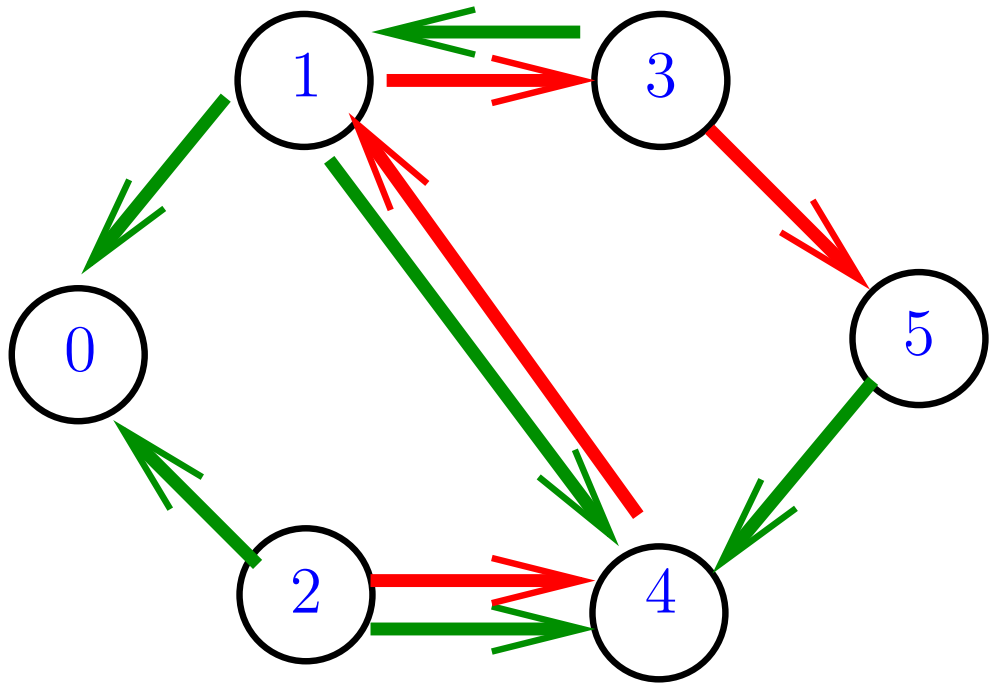
Exemplo: 2-4-1-3-5 tem comprimento 4



# 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**

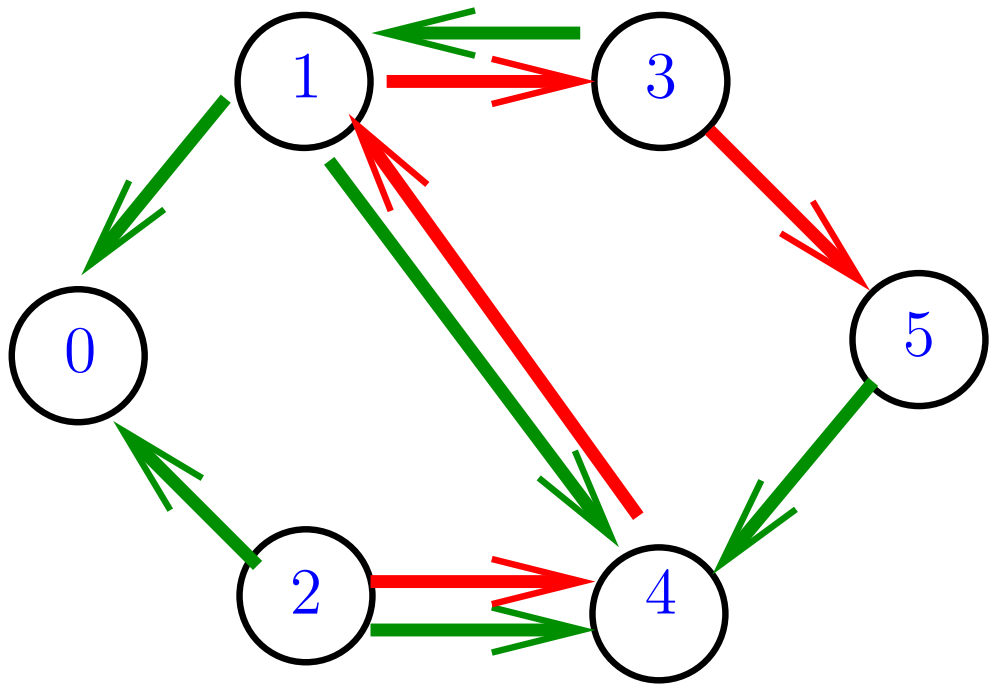
Exemplo: a distância de 2 a 5 é 4



# 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

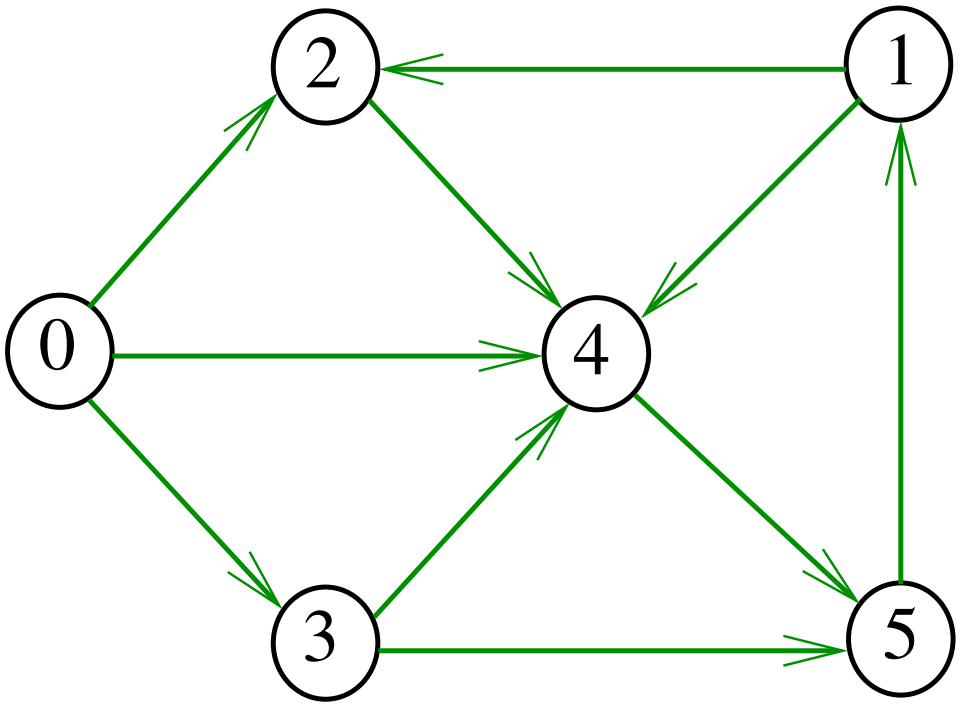


# Calculando distâncias

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

Exemplo: para `s` $= 0$

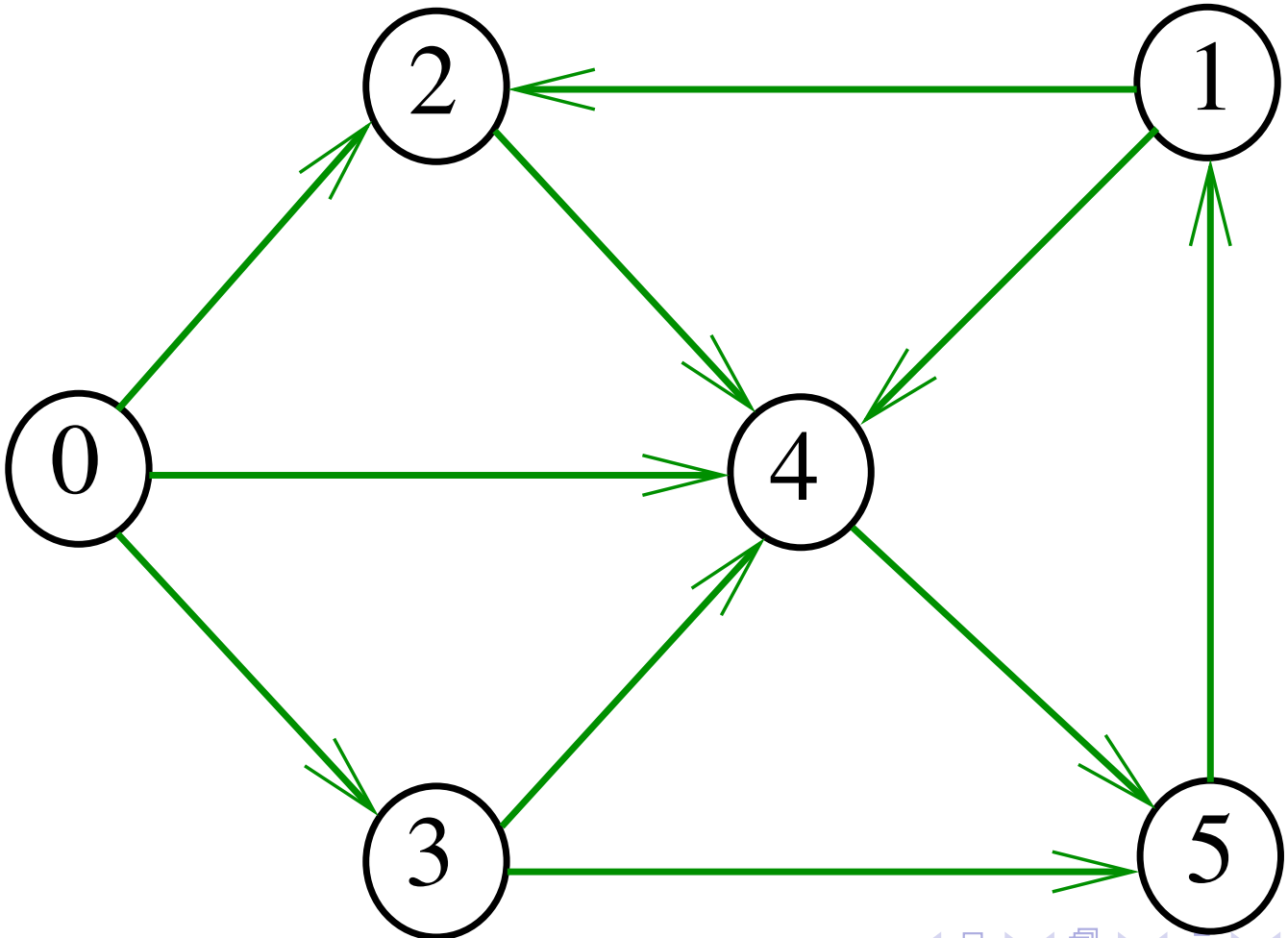
| `v` | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| `distTo[v]` | 0 | 3 | 1 | 1 | 1 | 2 |



# Simulação

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

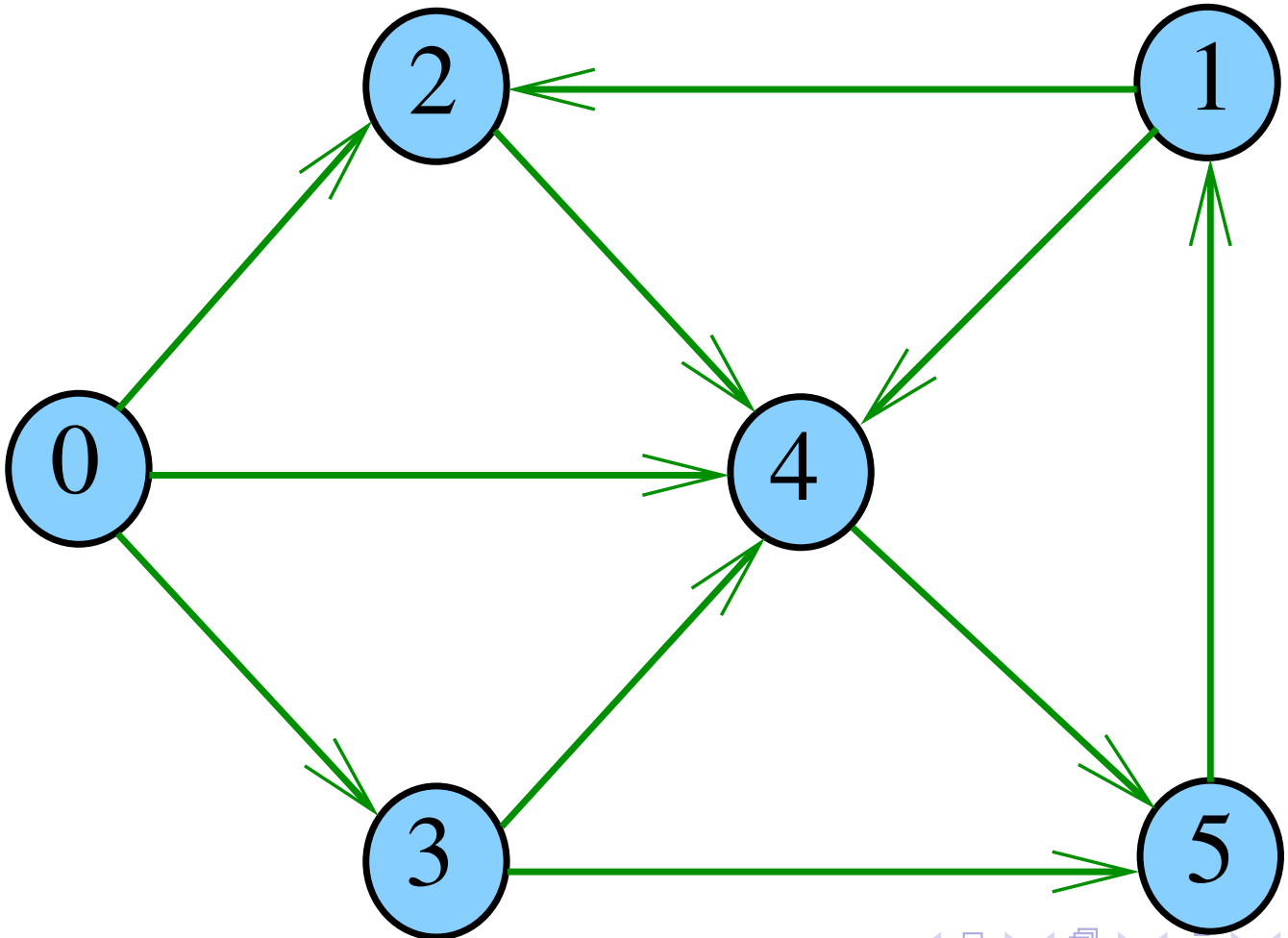
| v | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| distTo[v] | | | | | | |



# Simulação

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

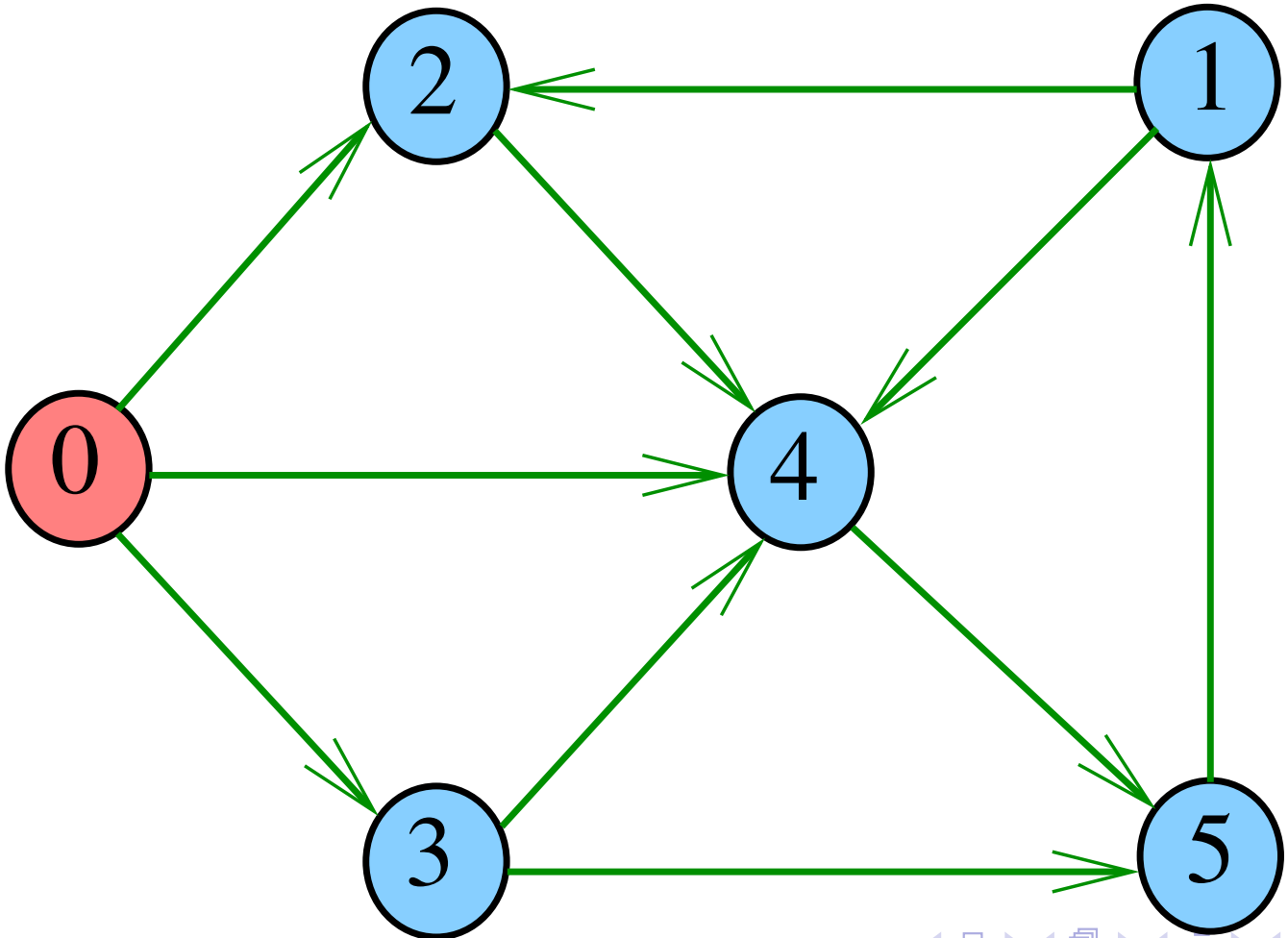
| v | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| distTo[v] | 6 | 6 | 6 | 6 | 6 | 6 |



# Simulação

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

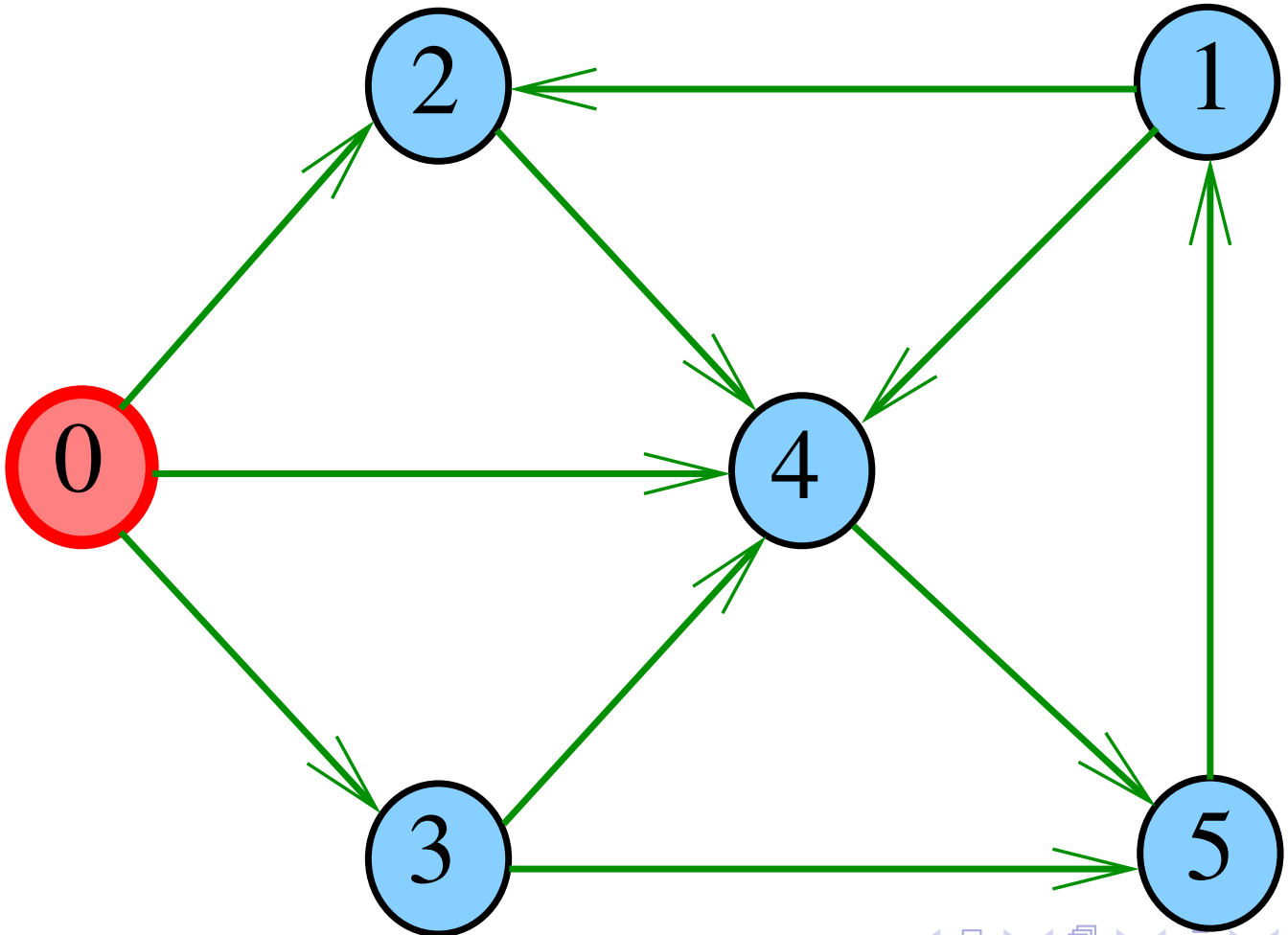
| v | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| distTo[v] | 6 | 6 | 6 | 6 | 6 | 6 |



# Simulação

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

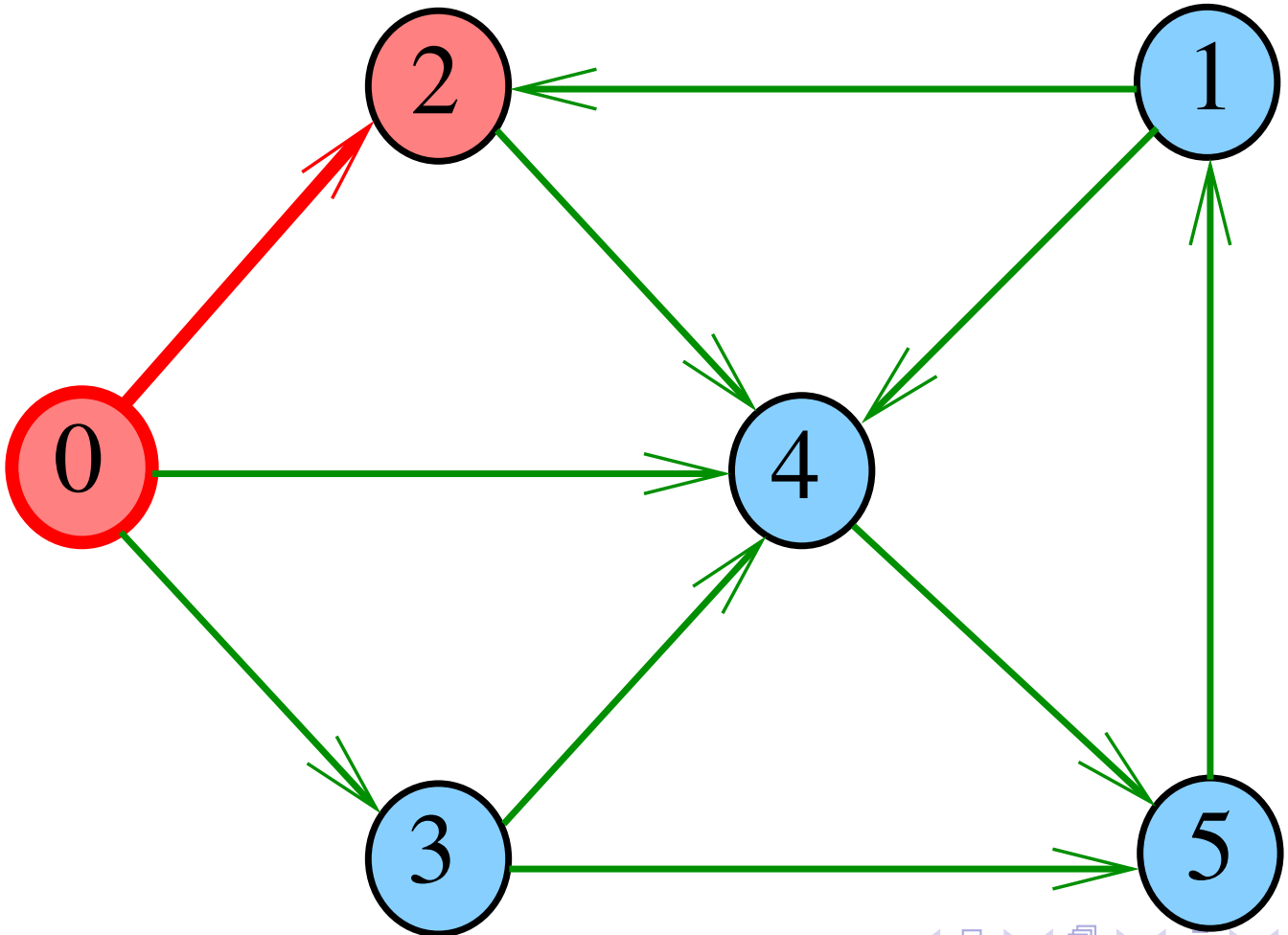
| v | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| distTo[v] | **0** | 6 | 6 | 6 | 6 | 6 |



# Simulação

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

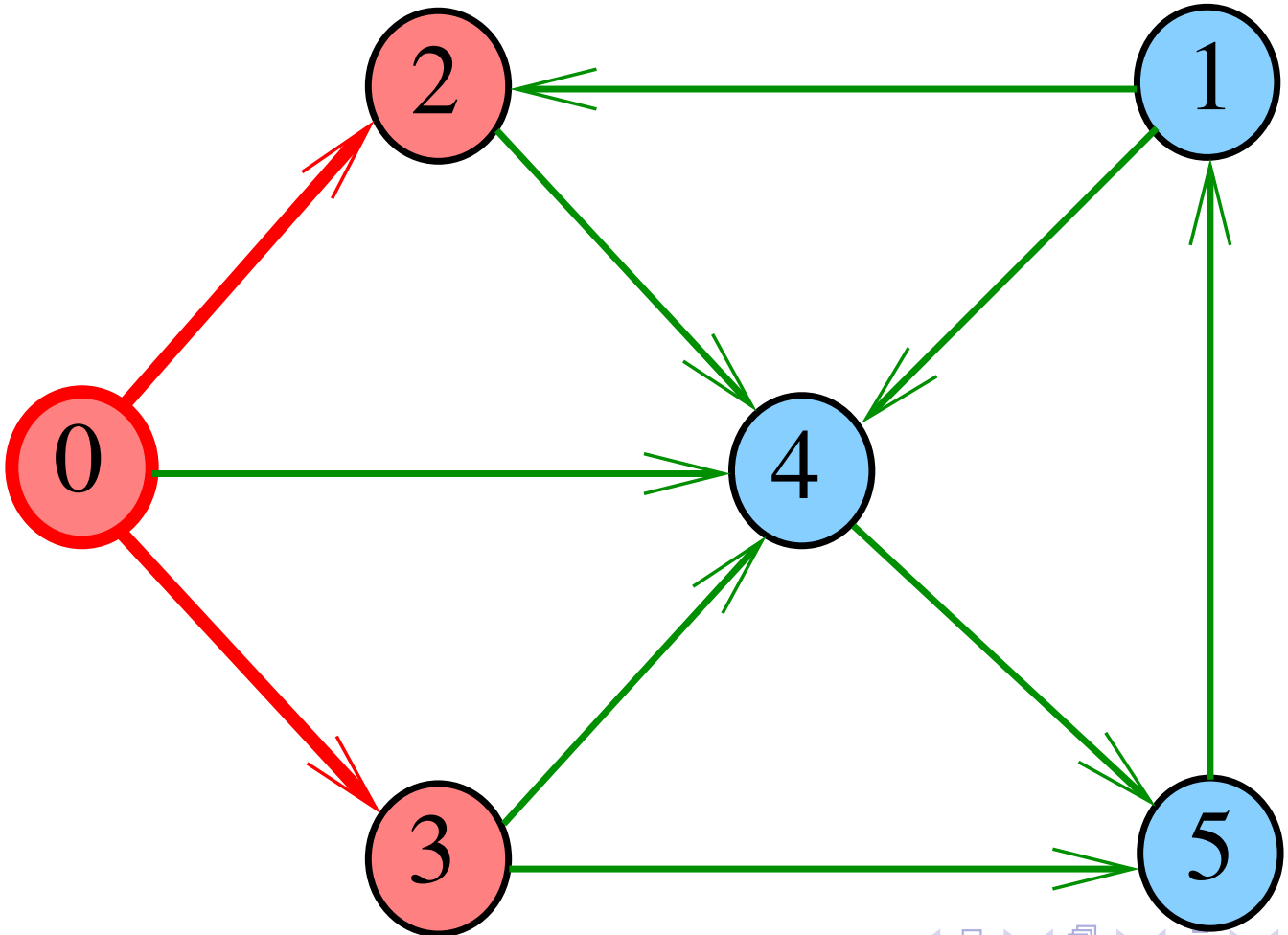
| v | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| distTo[v] | 0 | 6 | **1** | 6 | 6 | 6 |



# Simulação

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

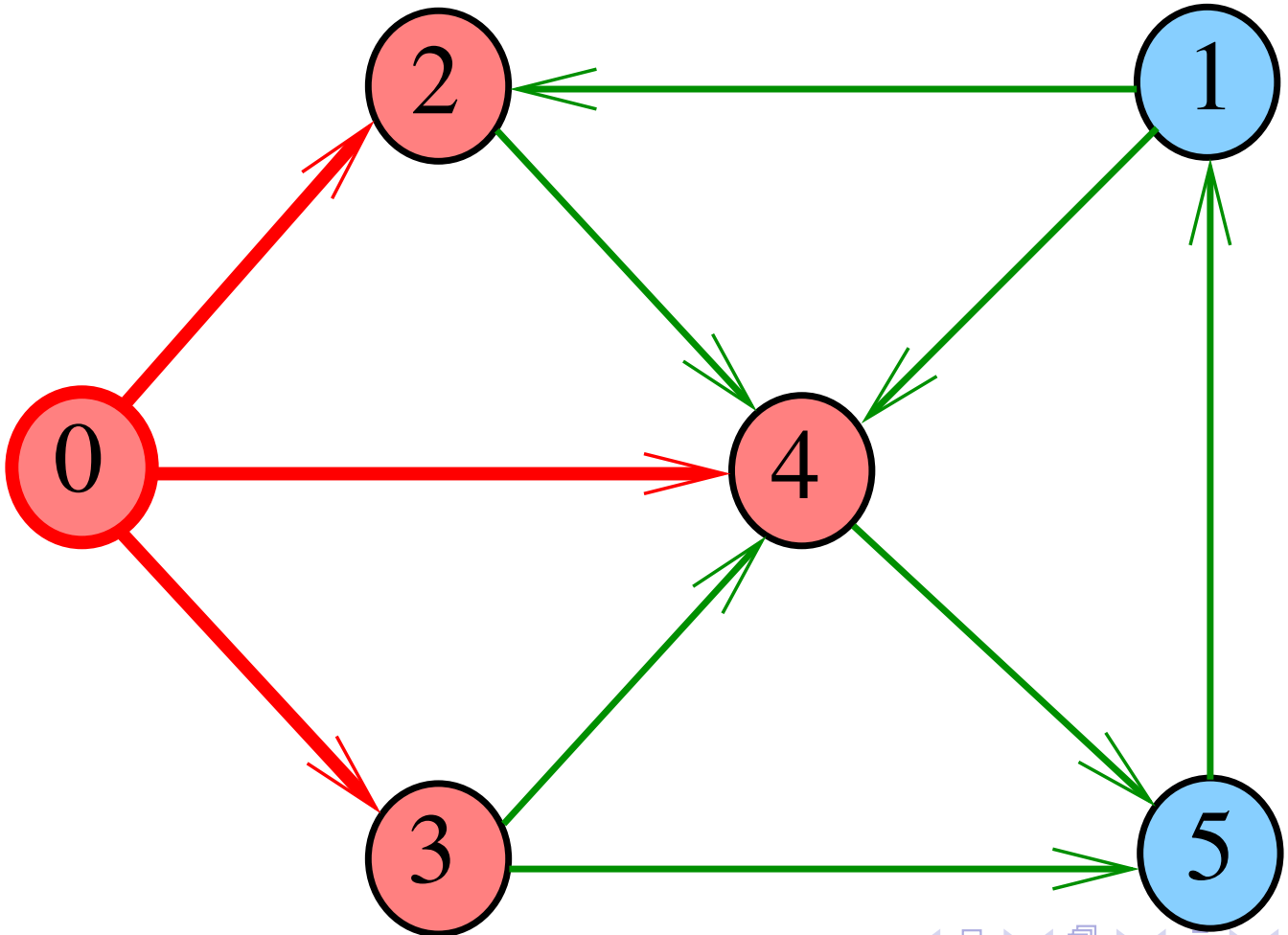
| v | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| distTo[v] | 0 | 6 | 1 | **1** | 6 | 6 |



# Simulação

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

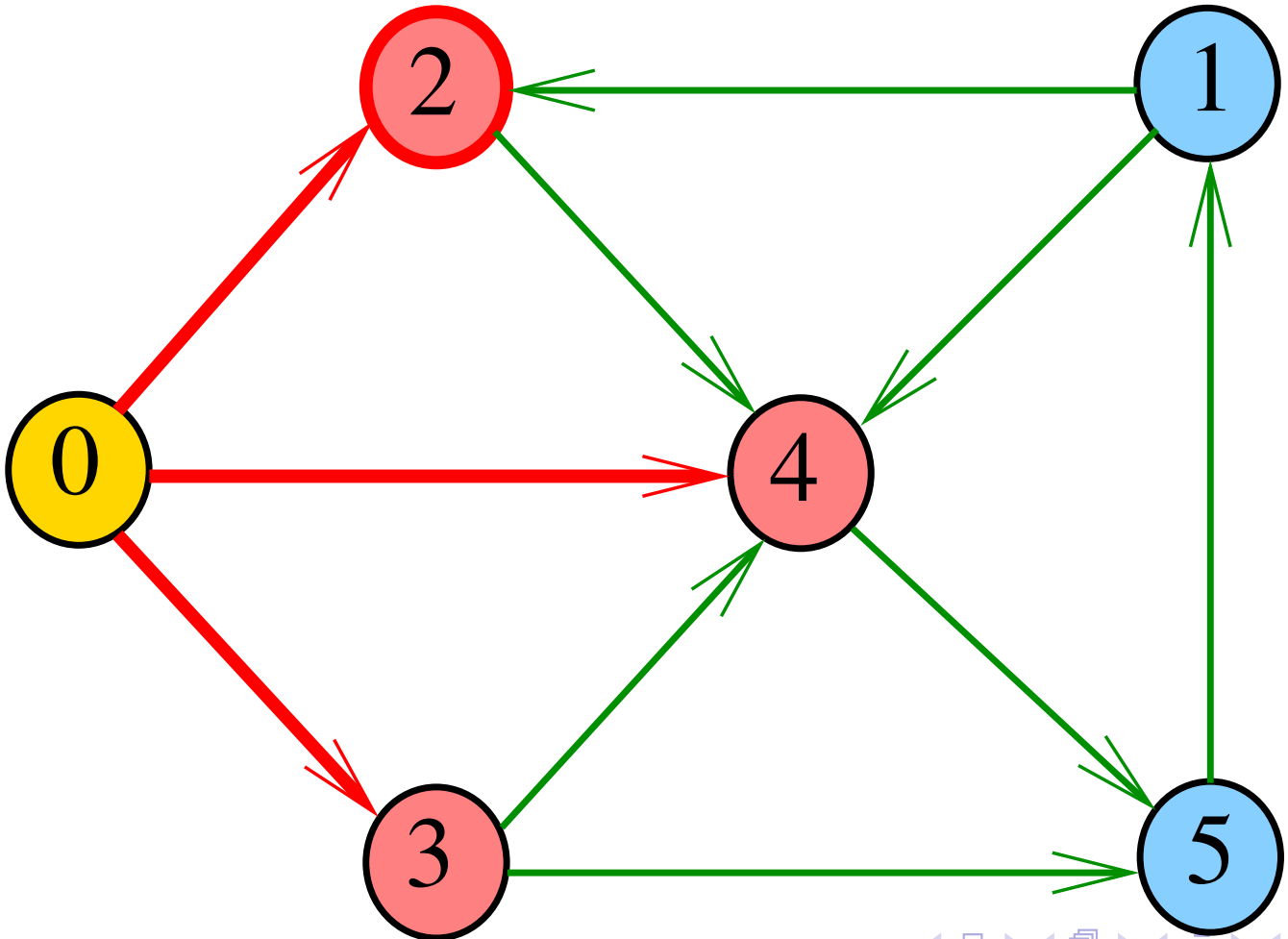
| v | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| distTo[v] | 0 | 6 | 1 | 1 | **1** | 6 |



# Simulação

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

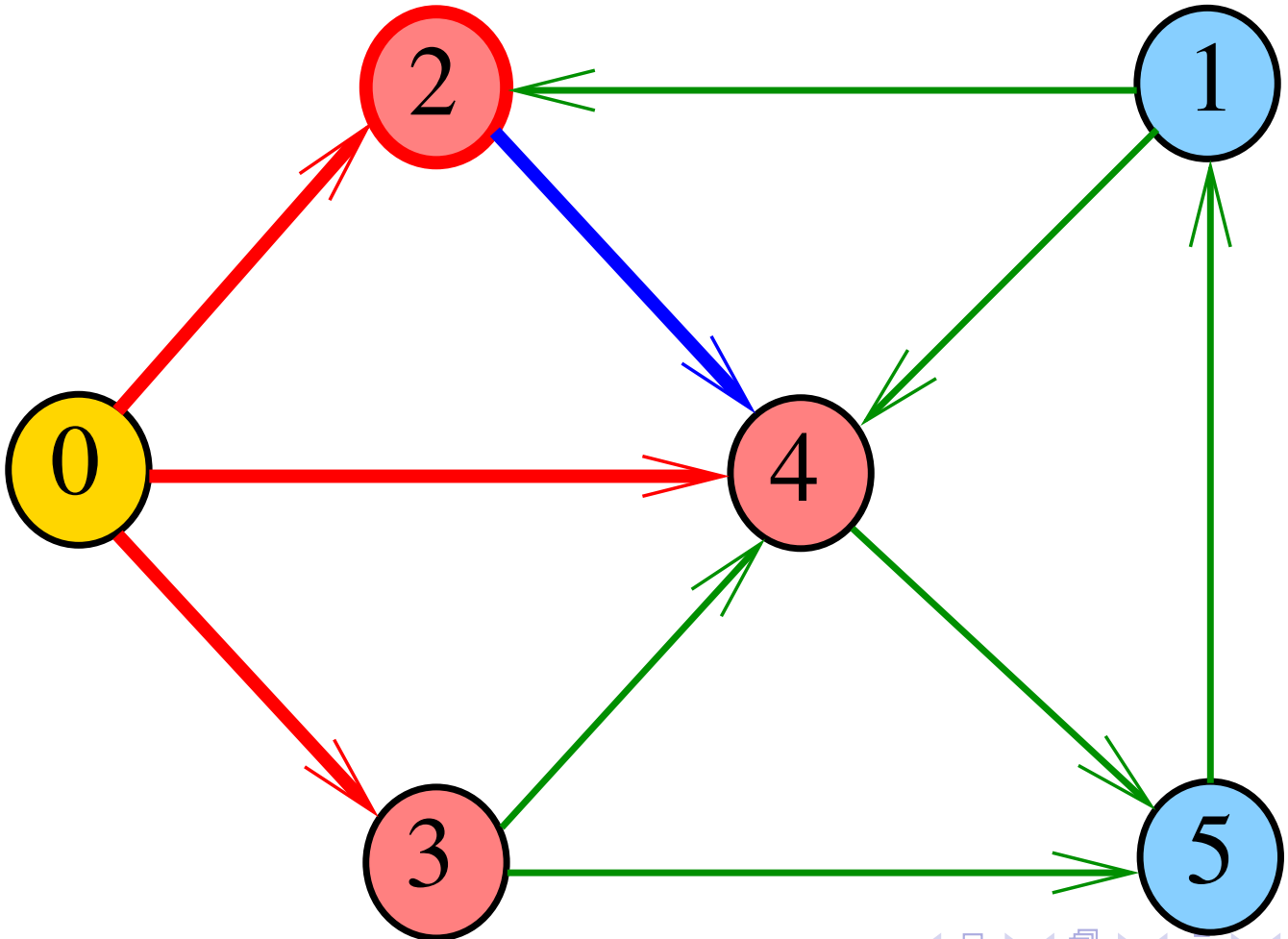
| v | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| distTo[v] | 0 | 6 | 1 | 1 | 1 | 6 |



# Simulação

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

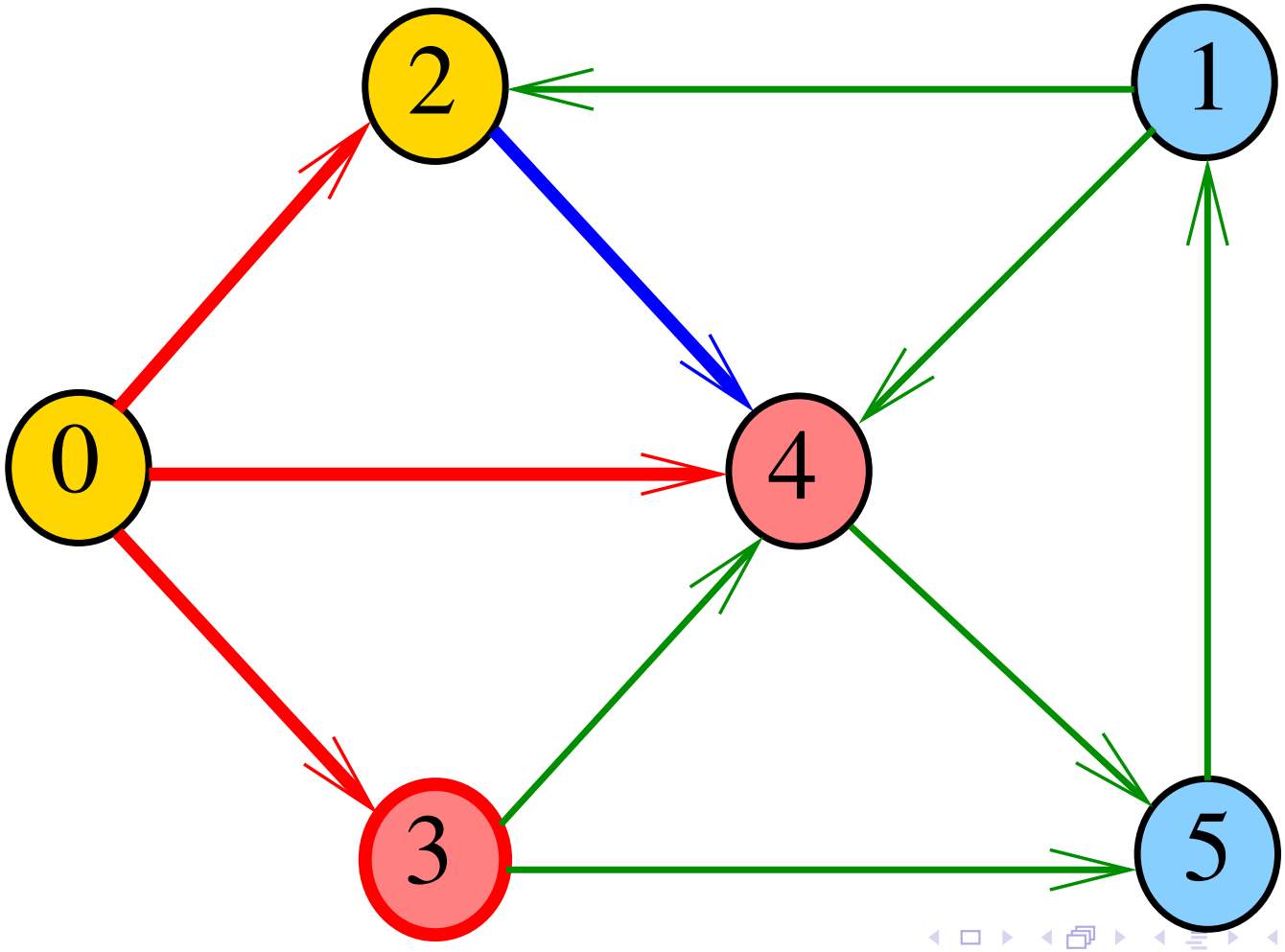
| v | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| distTo[v] | 0 | 6 | 1 | 1 | 1 | 6 |



# Simulação

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

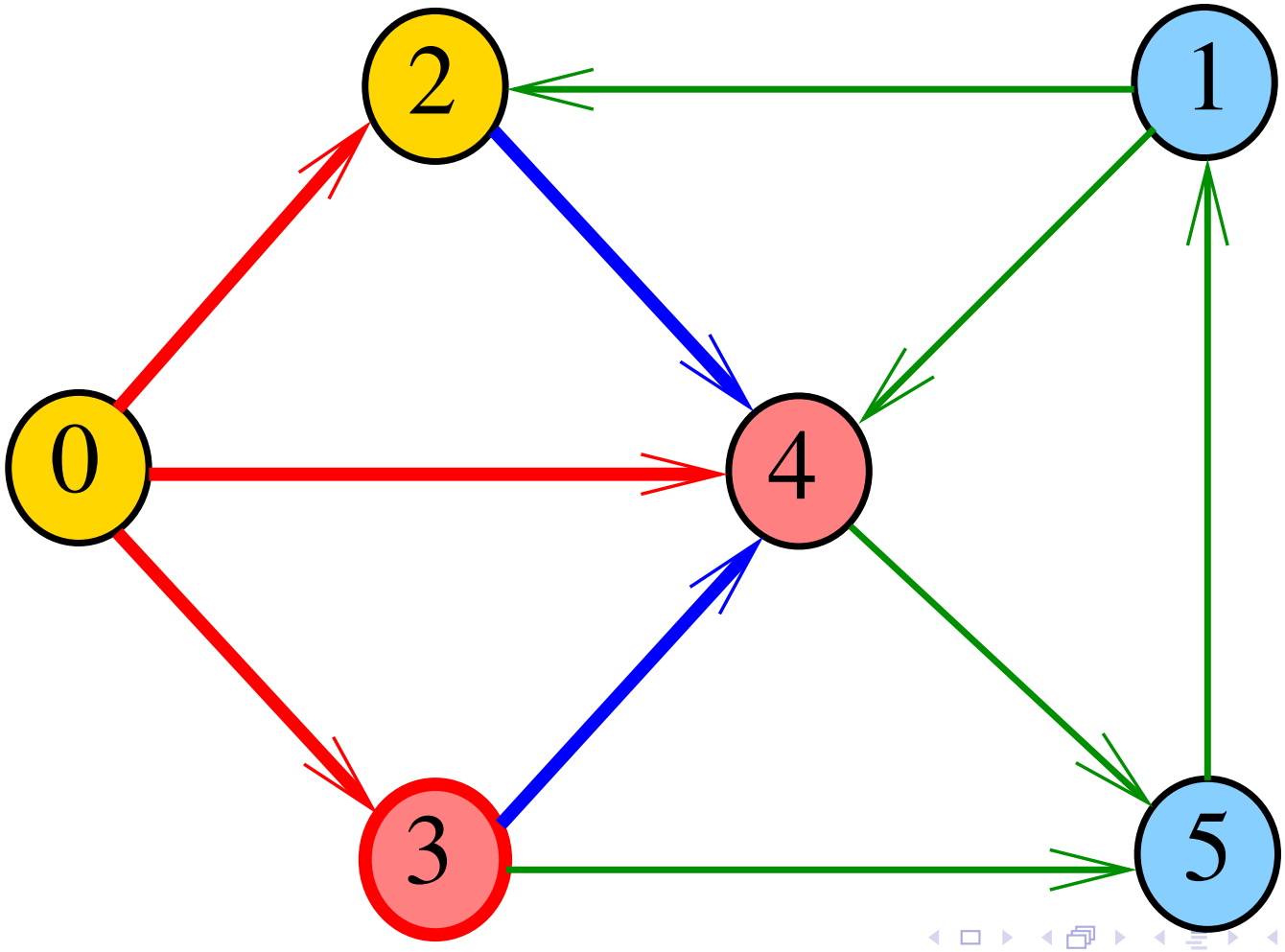
| v | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| distTo[v] | 0 | 6 | 1 | 1 | 1 | 6 |



# Simulação

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

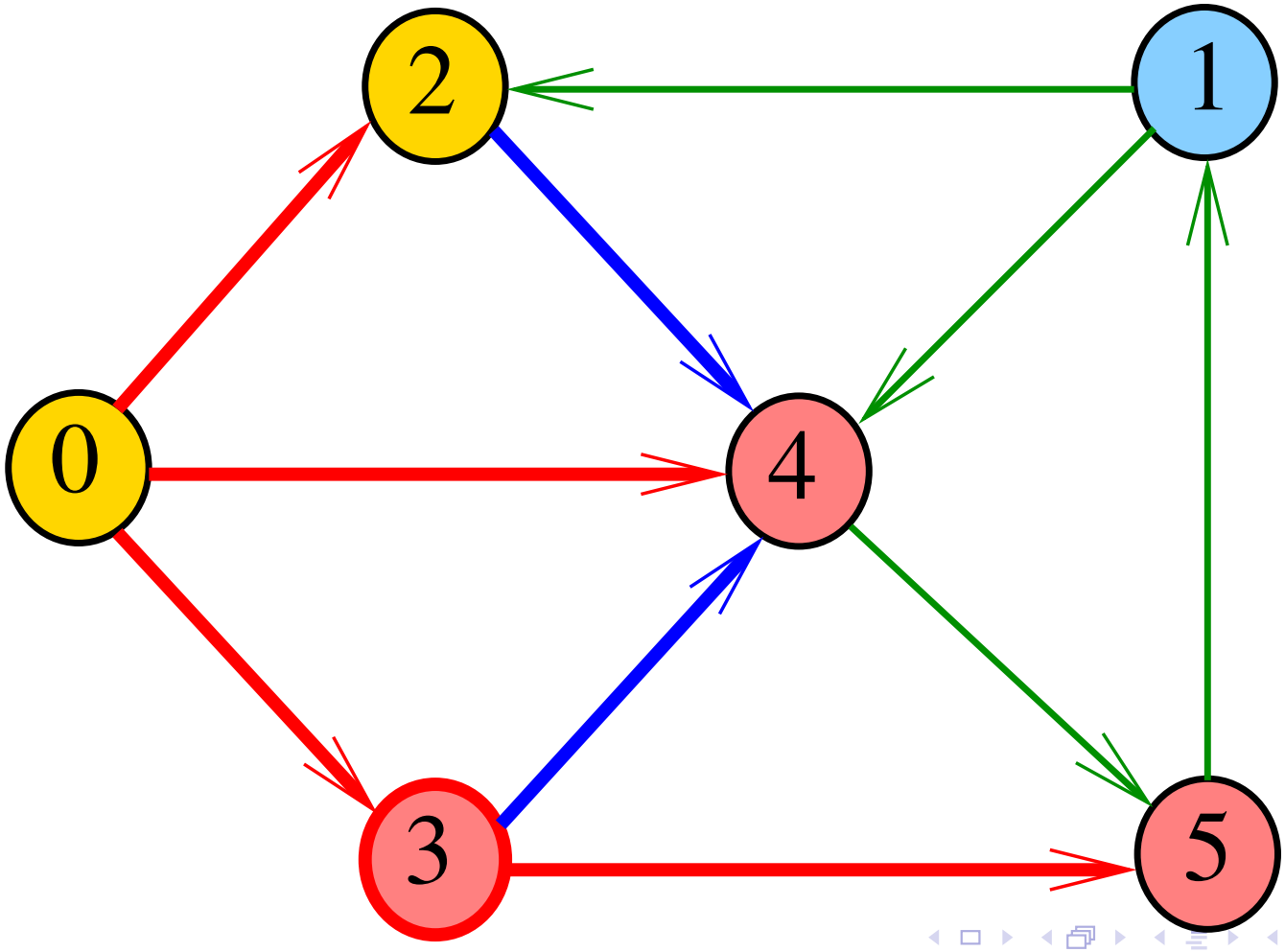
| v | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| distTo[v] | 0 | 6 | 1 | 1 | 1 | 6 |



# Simulação

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

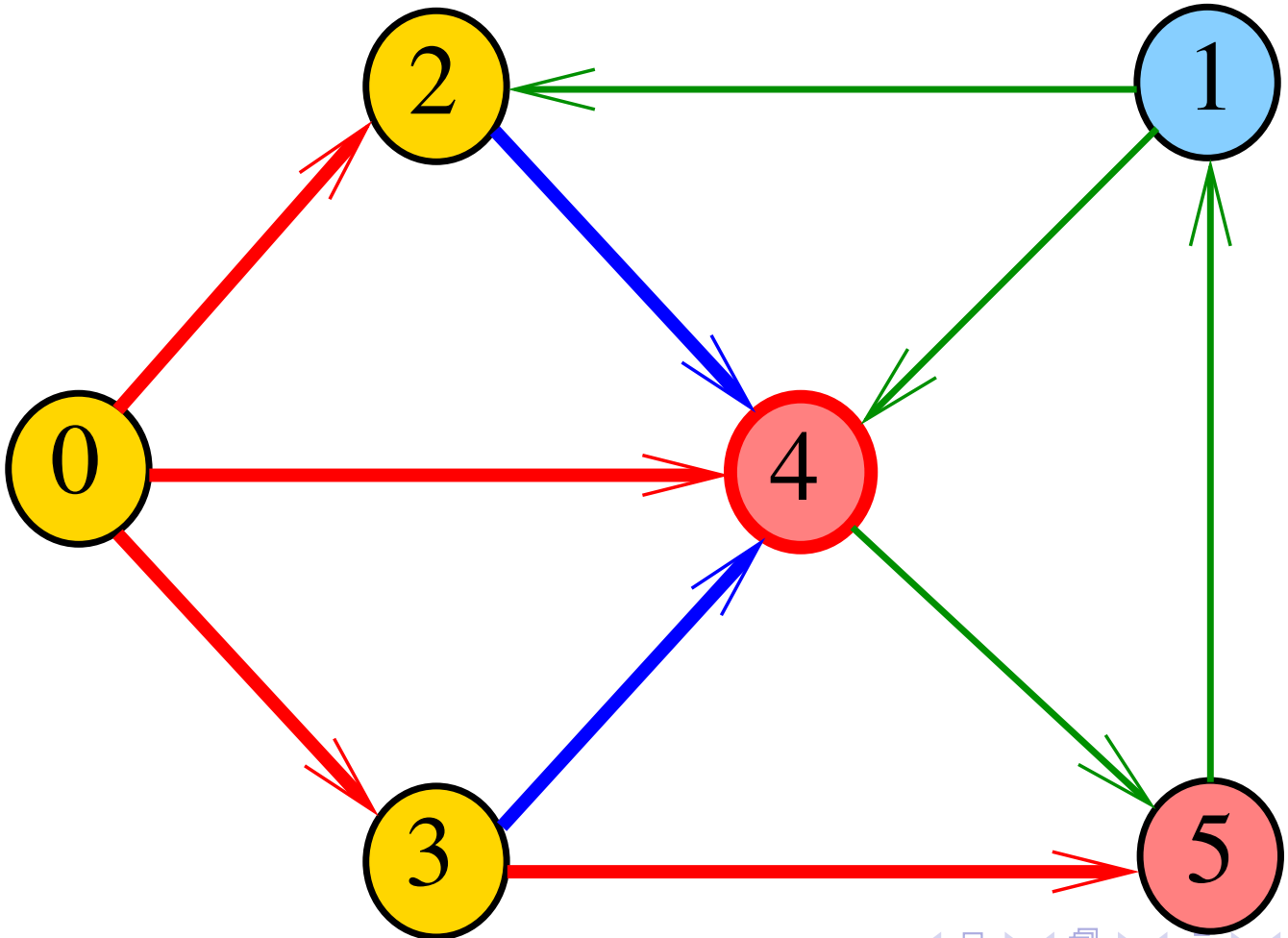
| v | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| distTo[v] | 0 | 6 | 1 | 1 | 1 | 2 |



# Simulação

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

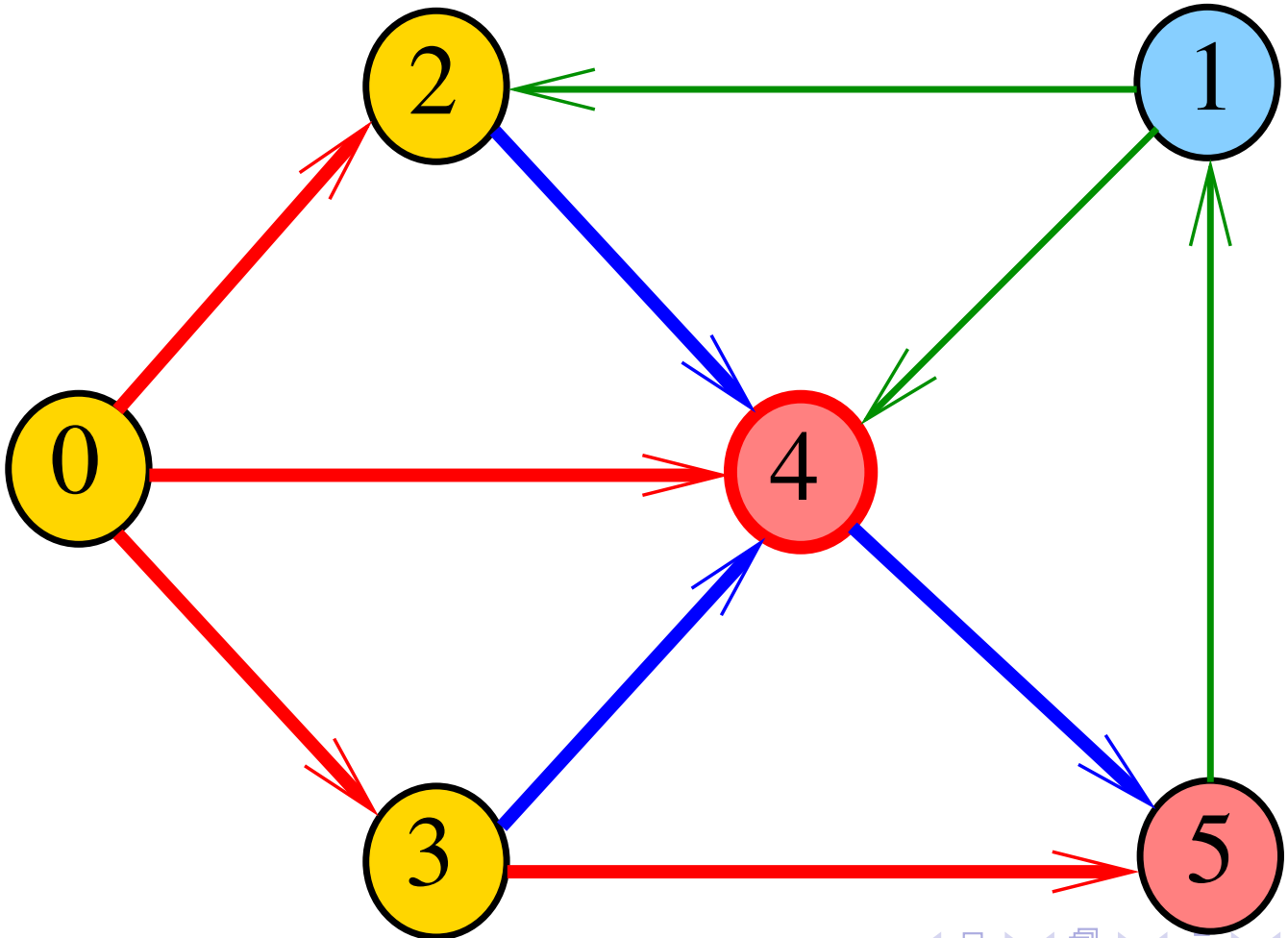
| v | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| distTo[v] | 0 | 6 | 1 | 1 | 1 | 2 |



# Simulação

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

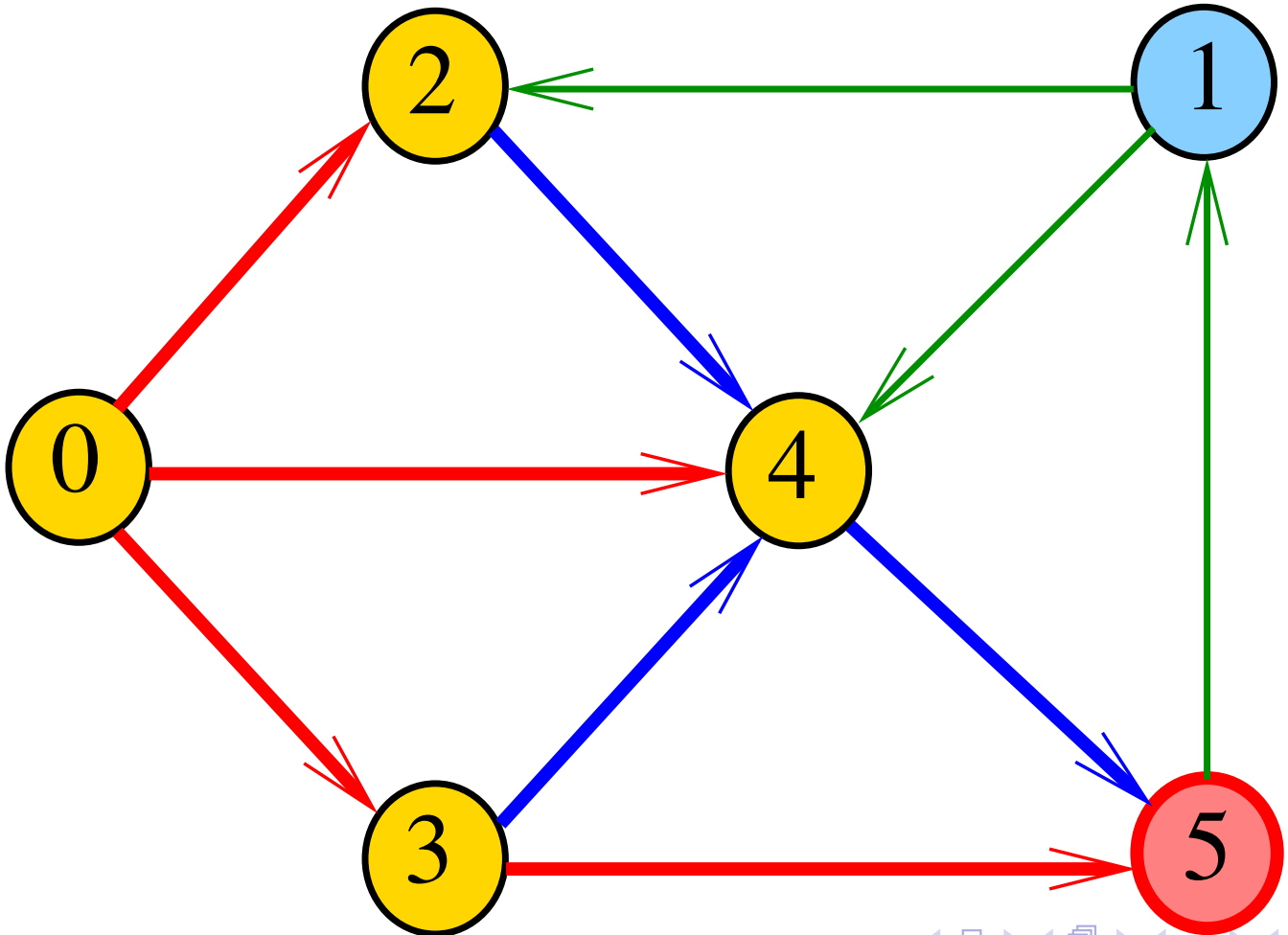
| v | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| distTo[v] | 0 | 6 | 1 | 1 | 1 | 2 |



# Simulação

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

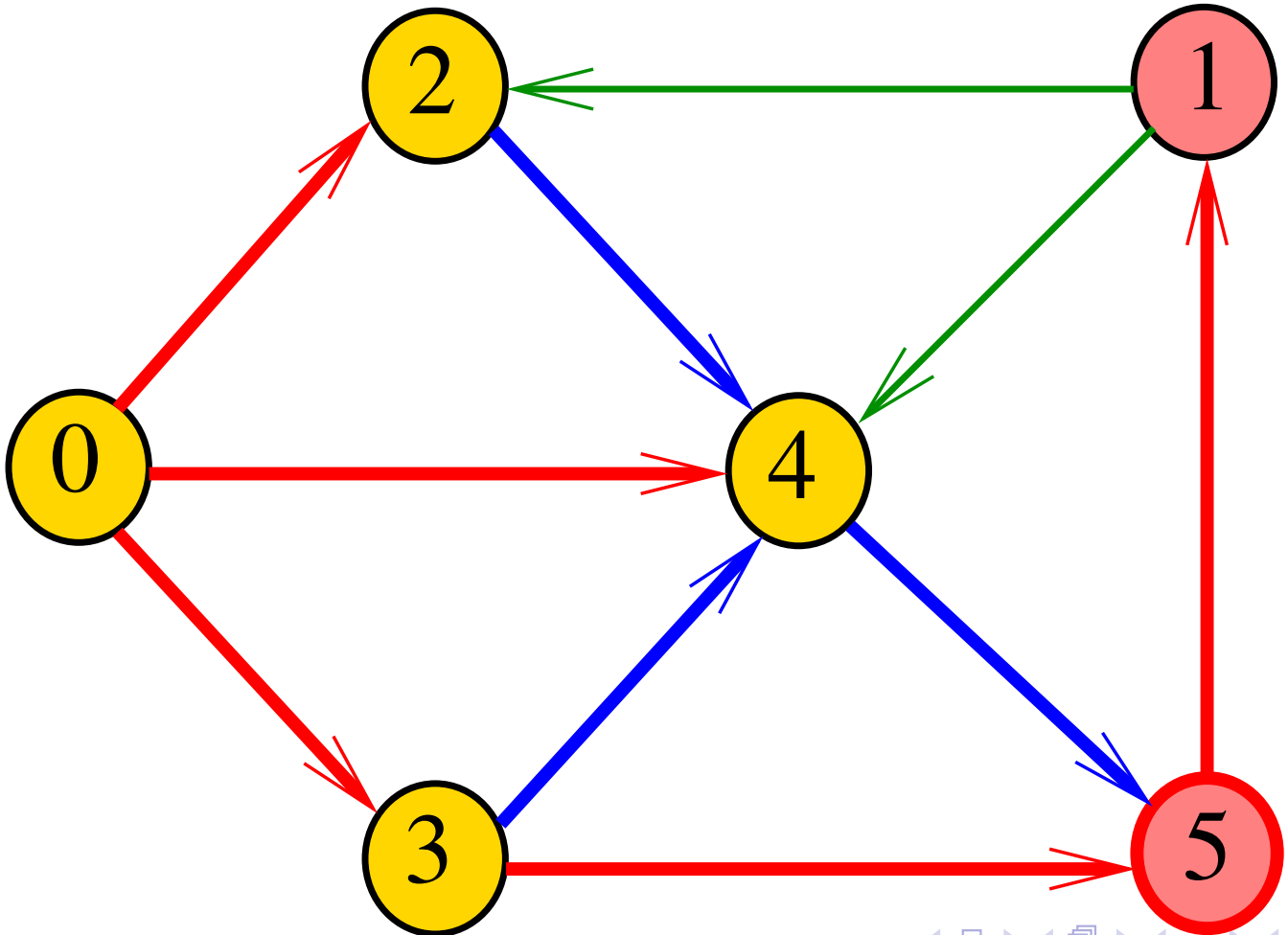
| v | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| distTo[v] | 0 | 6 | 1 | 1 | 1 | 2 |



# Simulação

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

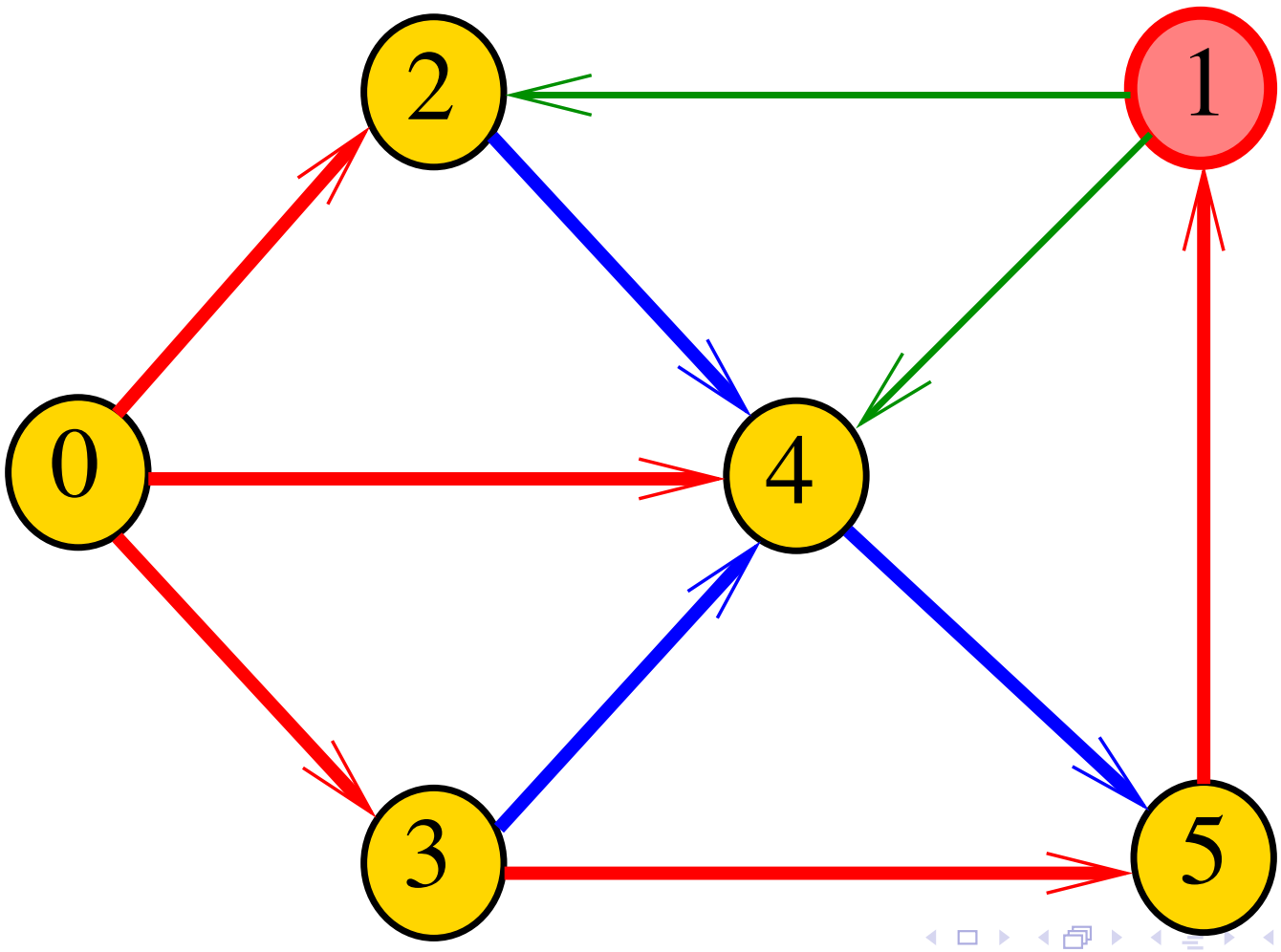
| v | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| distTo[v] | 0 | **3** | 1 | 1 | 1 | 2 |



# Simulação

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

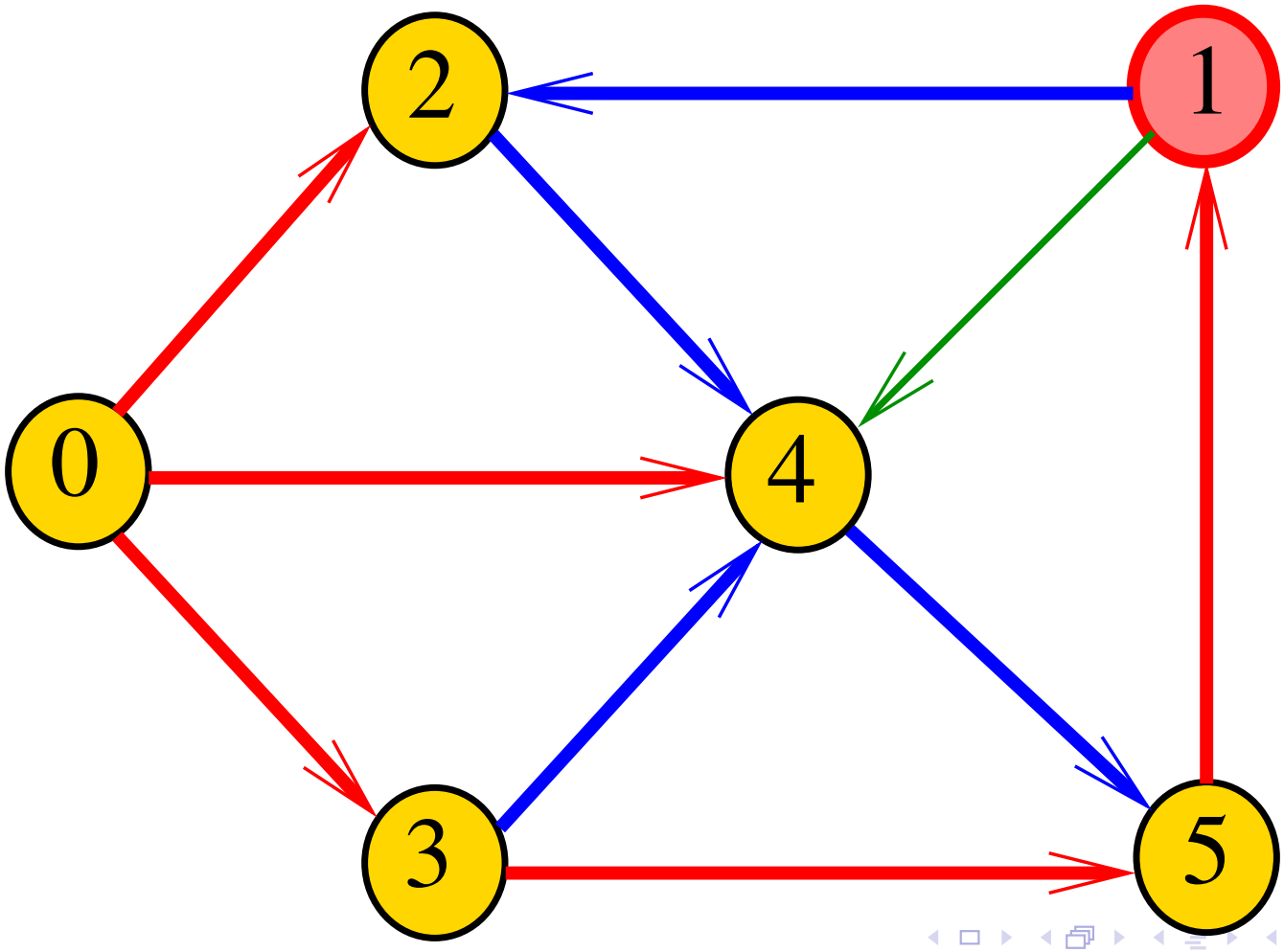
| v | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| distTo[v] | 0 | 3 | 1 | 1 | 1 | 2 |



# Simulação

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

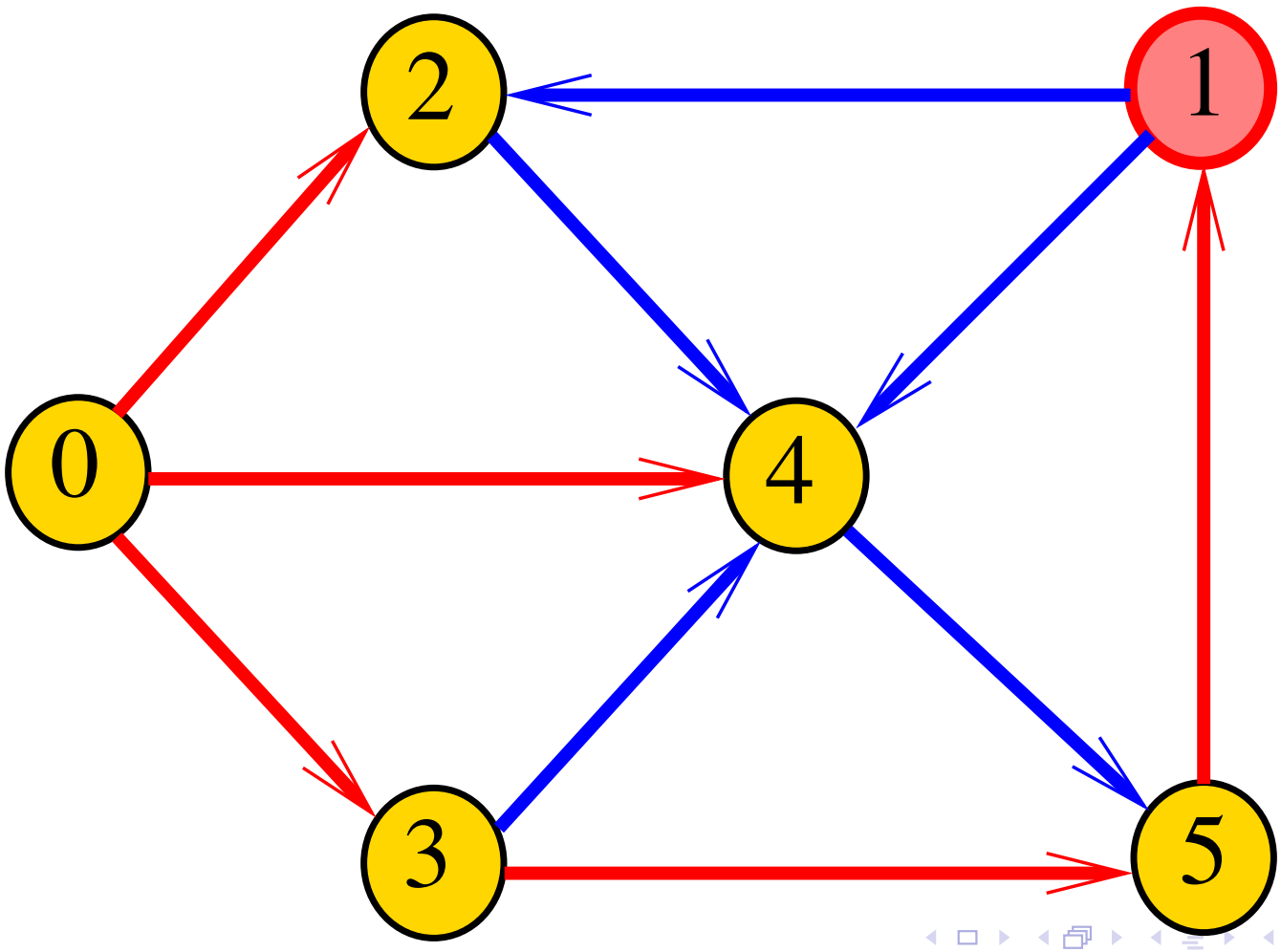
| v | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| distTo[v] | 0 | 3 | 1 | 1 | 1 | 2 |



# Simulação

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

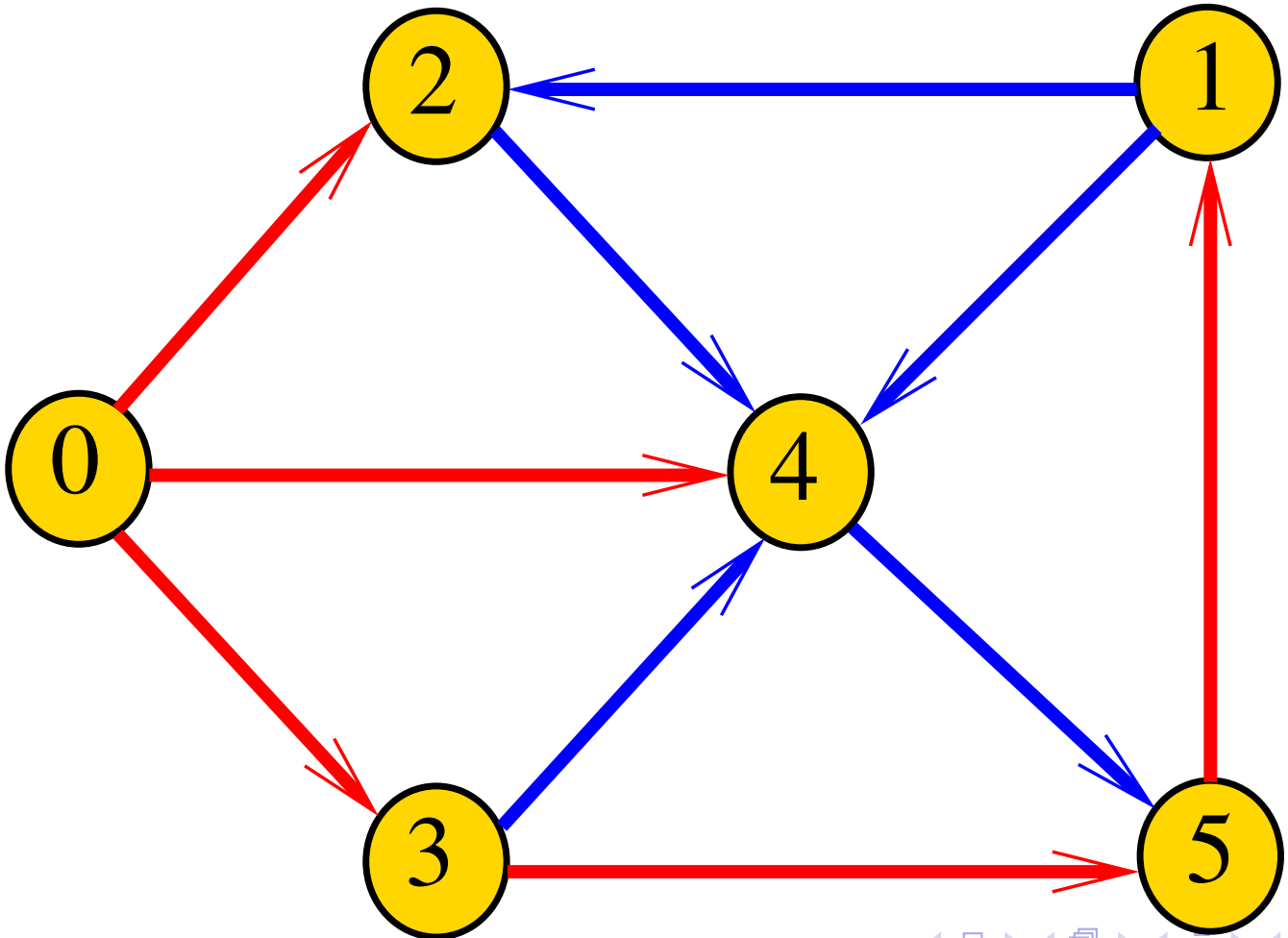
| v | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| distTo[v] | 0 | 3 | 1 | 1 | 1 | 2 |



# Simulação

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

| v | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| distTo[v] | 0 | 3 | 1 | 1 | 1 | 2 |



# Nova classe BFSpaths

BFSpaths armazena no vetor distTo[] a distância do vértice s a cada um dos vértices do digrafo G

A distância 'infinita' é representada por G.V()

```
private static final int INFINITY=G.V();
private int[] distTo = new int[G.V()];
```

# Nova BFSpaths: esqueleto

```
public class BFSpaths {
   private static final int INFINITY;
   private final int s;
   private boolean[] marked;
   private int[] distTo; // era marked
   private int[] edgeTo;
   public BFSpaths(Digraph G, int s) {}
   private void bfs(Digraph G, int s) {}
   public boolean hasPath(int v) {}
   public boolean distTo(int v) {}
   public Iterable<Integer> pathTo(int v)
}
```

# BFSpaths

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

```
public BFSpaths(Digraph G, int s) {
   INFINITY = G.V();
   marked = new boolean[G.V()];
   distTo = new int[G.V()];
   edgeTo = new int[G.V()];
   this.s = s;
   for (int v = 0; v < G.V(); v++)
       distTo[v] = INFINITY;
   bfs(G, s);
}
```

# bfs(): inicializações

```
private void bfs(Digraph G, int s) {
   Queue<Integer> q= new Queue<Integer>();
   marked[v] = true;
   distTo[s] = 0;
   q.enqueue(s);

   // aqui vem a iteração do próximo slide
```

# bfs(): iteração

```
    while (!q.isEmpty()) {
        int v = q.dequeue();
        for (int w :  G.adj(v)) {
            if (!marked[w]) {
            if (distTo[w] == INFINITY) {
                edgeTo[w] = v;
                distTo[w] = distTo[v] + 1;
                marked[w] = true;
                q.enqueue(w);
            }
        }
    }
}
```

# BFSpaths

Há um caminho de s a v?

```
// Método adaptado de DFSpaths.
public boolean hasPath(int v) {
   return distTo[v] < INFINITY;
}

// retorna o número de arcos em um
// caminho mínimo de s a t
public int distTo(int v) {
   return distTo[v];
}
```

# BFSpaths

Retorna um caminho de s a v ou null se um tal caminho não existe.

```
// Método copiado de DFSpaths.
public Iterable<Integer> pathTo(int v) {
   if (!hasPath(v)) return null;
   Stack<Integer> path =
           new Stack<Integer>();
   for (int x = v; x != s; x = edgeTo[x])
       path.push(x);
   path.push(s);
   return path;
}
```

# Relações invariantes

No início de cada iteração a fila consiste em

> *zero ou mais vértices à distância* `d` *de* `s`,
> *seguidos de zero ou mais vértices à*
> *distância* `d+1` *de* `s`,

para algum `d`

Isto permite concluir que, no início de cada iteração, para todo vértice `x`, se `distTo[x] != G.V()` então `distTo[x]` é a distância de `s` a `x`

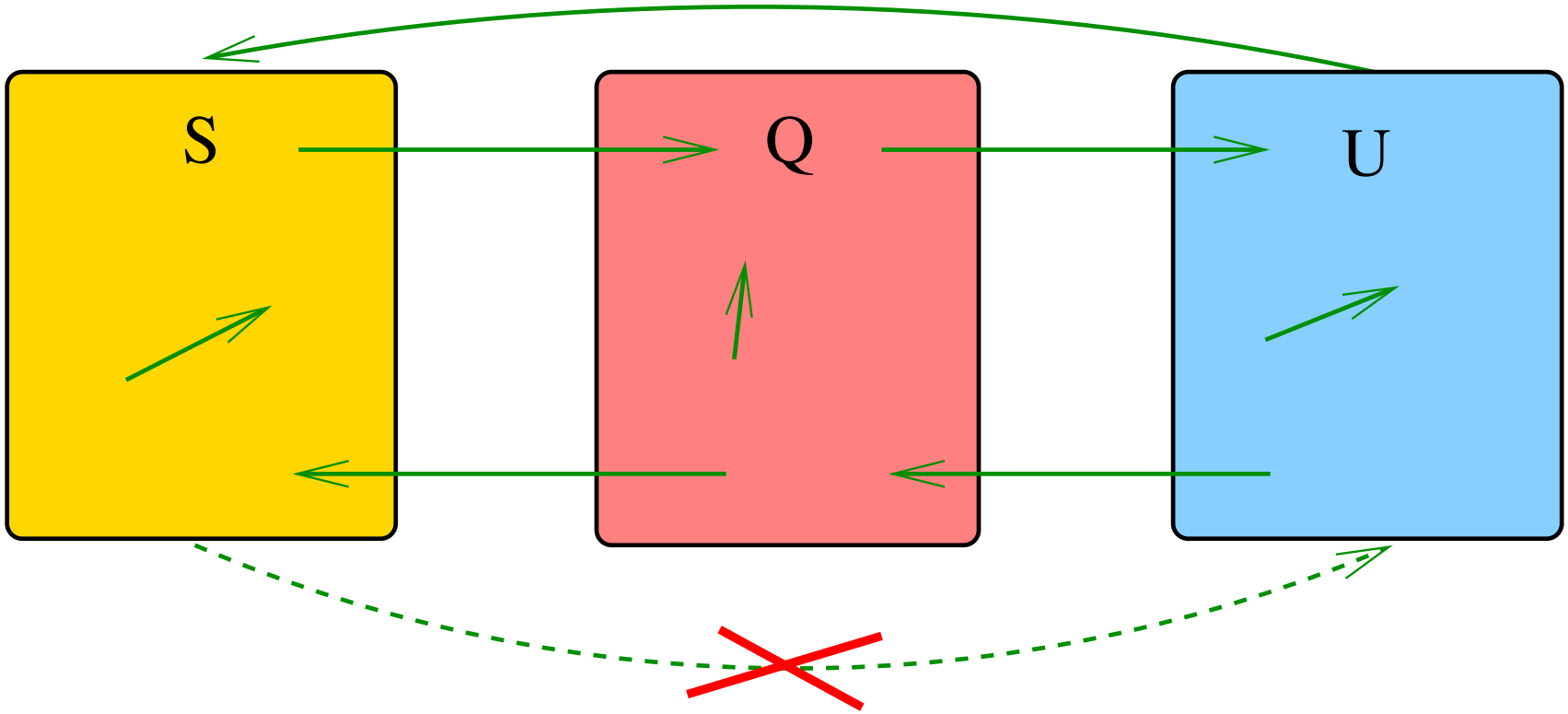
# Relações invariantes

S = vértices examinados

Q = vértices visitados = vértices na fila

U = vértices ainda não visitados

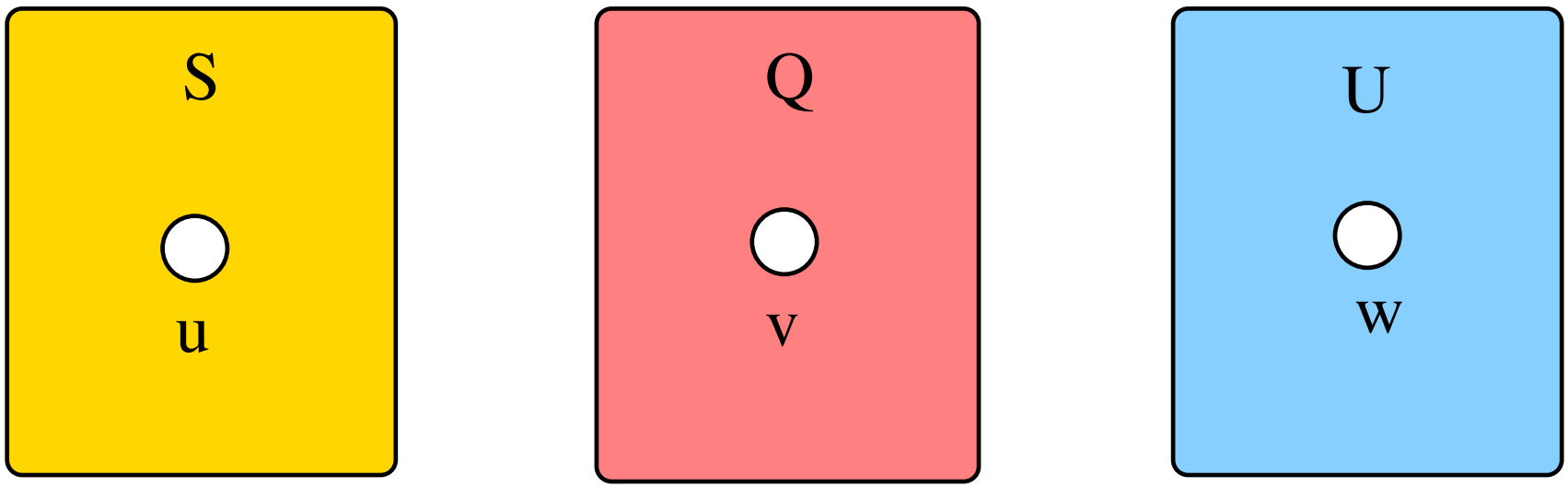
(i0) não existe arco v-w com v em S e w em U



# Relações invariantes

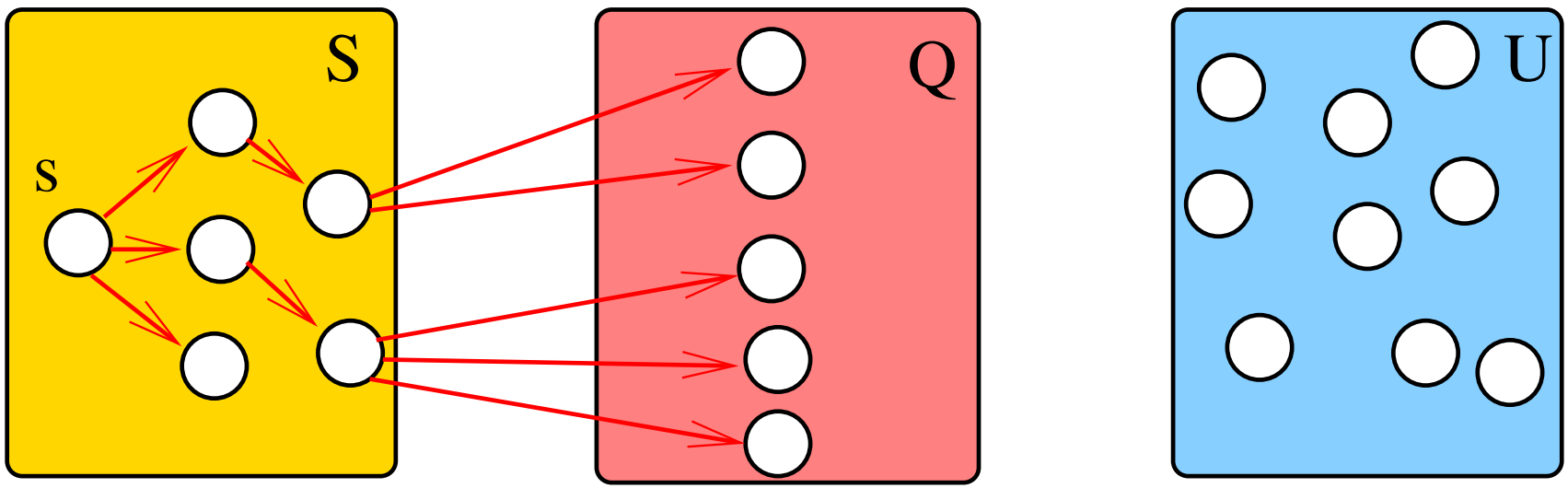
(i1) para cada `u` em `S`, `v` em `Q` e `w` em `U`

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



# Relações invariantes

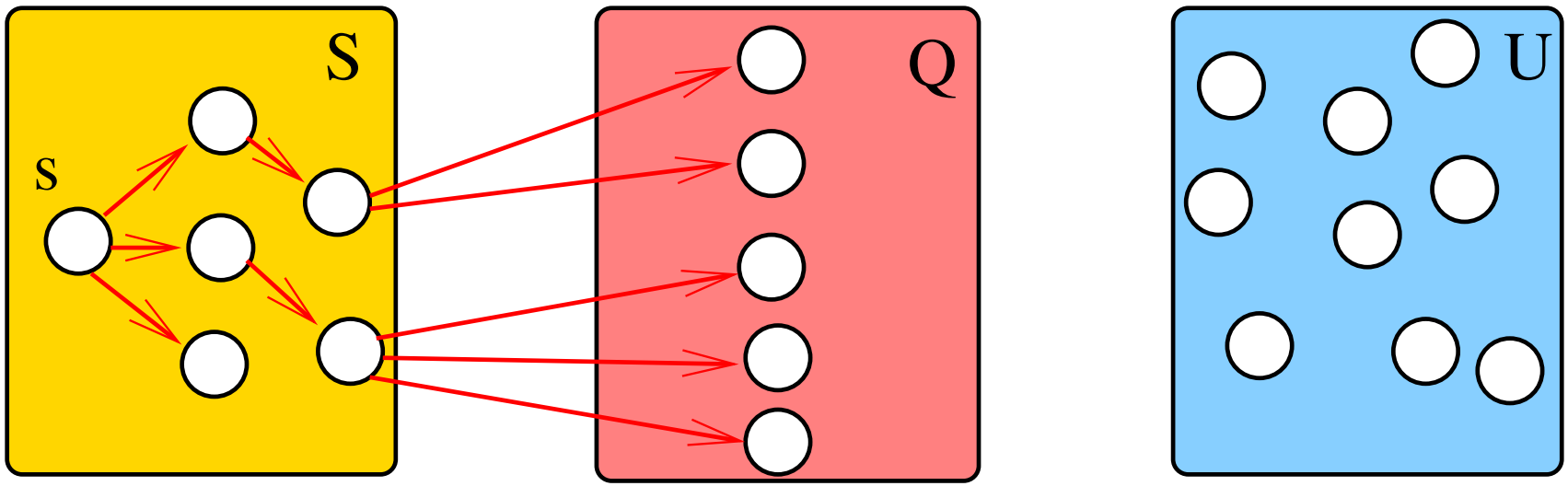
(i2) O vetor `edgeTo` restrito aos vértices de `S` e `Q` determina um árborescência com raiz `s`



# Relações invariantes

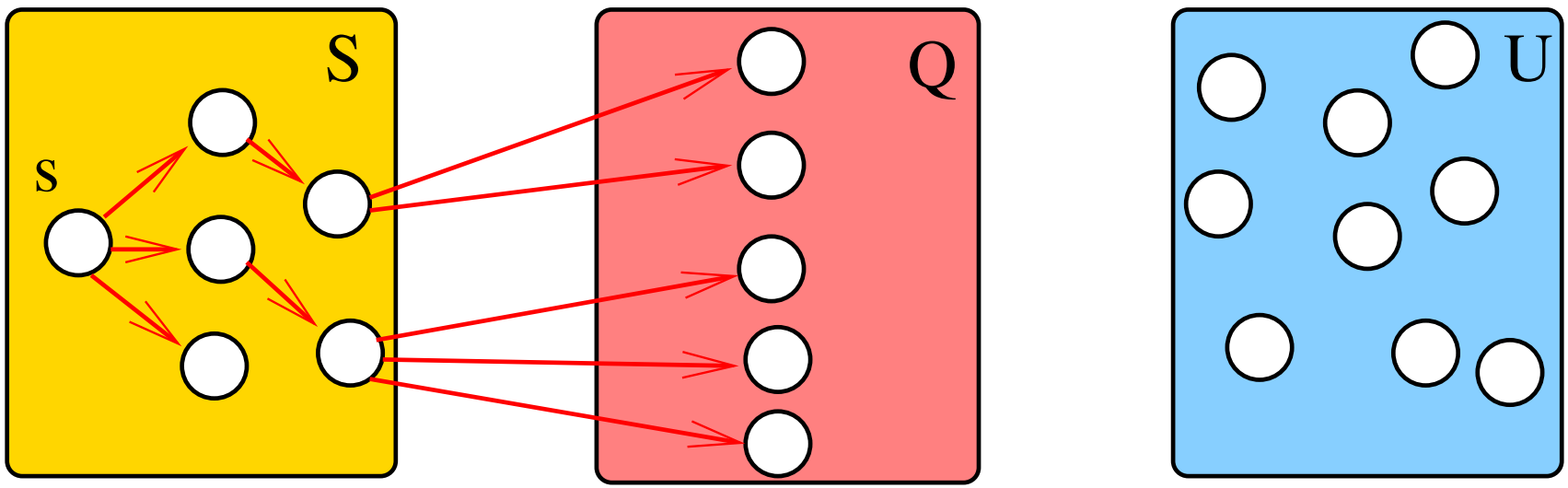
(i3) Para arco `v-w` na arborescência vale que

$$\texttt{distTo}[\texttt{w}] = \texttt{distTo}[\texttt{v}] + 1$$

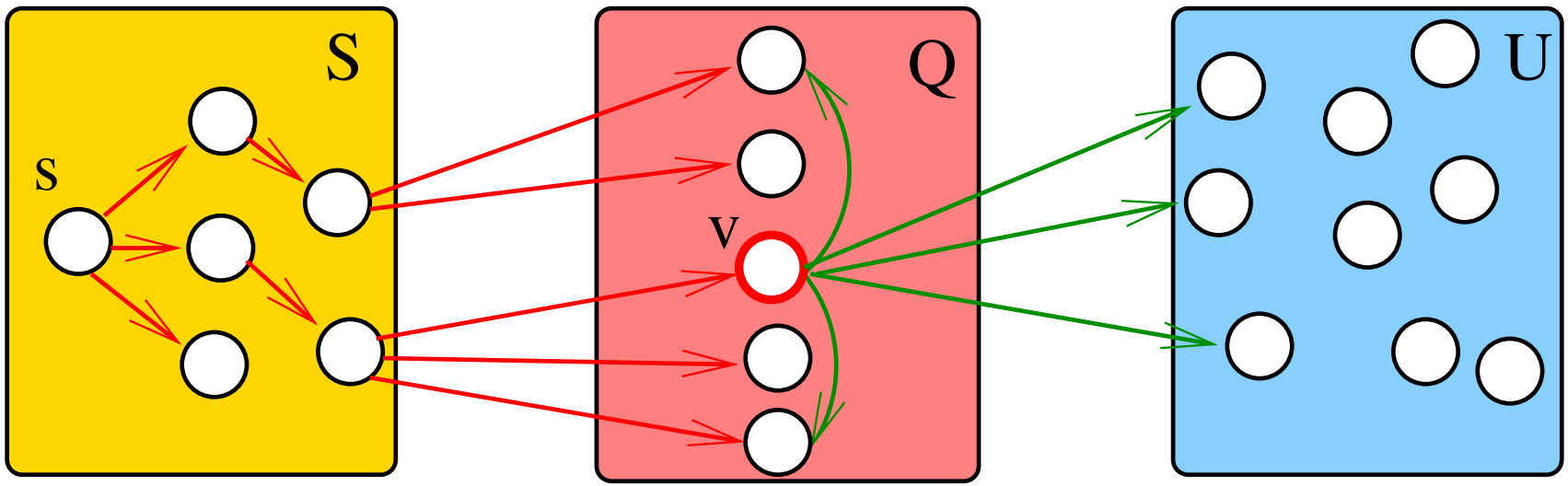


# Relações invariantes

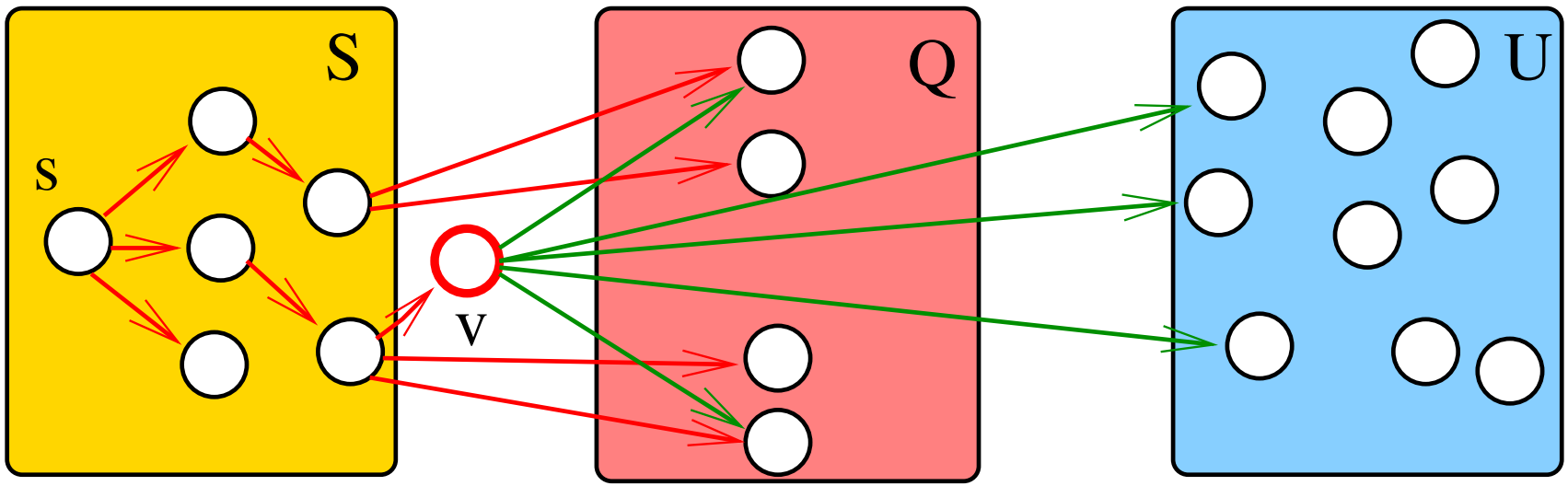
(i3) Para cada vértice `v` em `S` vale que `distTo`[`v`] é o custo de um caminho mínimo de `s` a `v`.



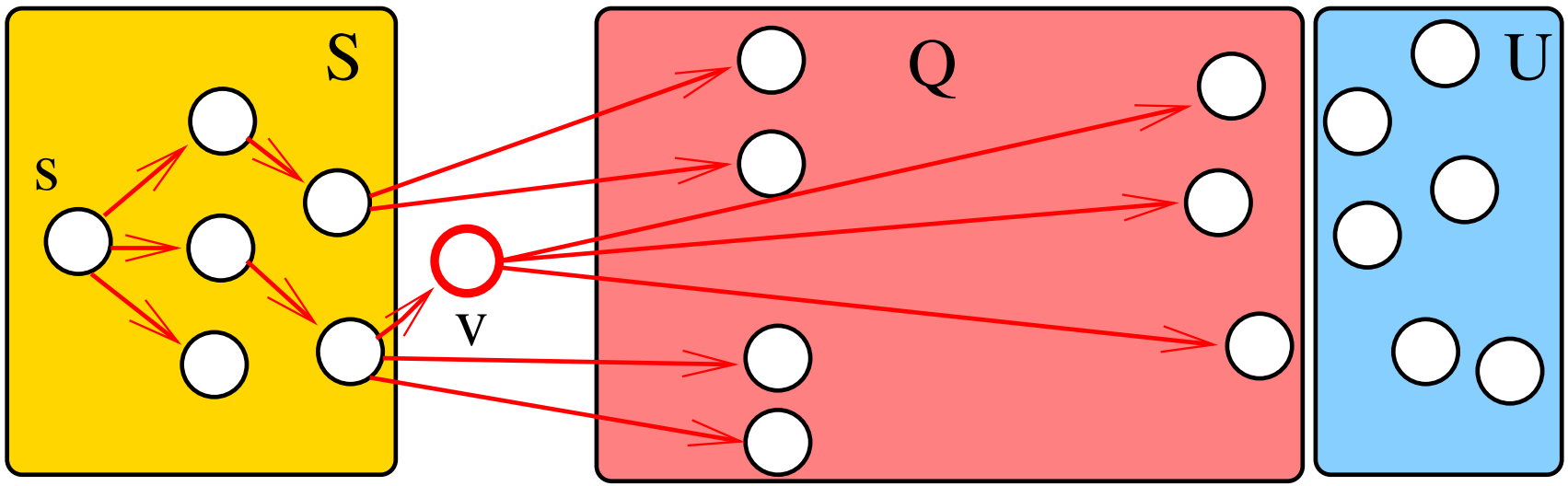
# Iteração



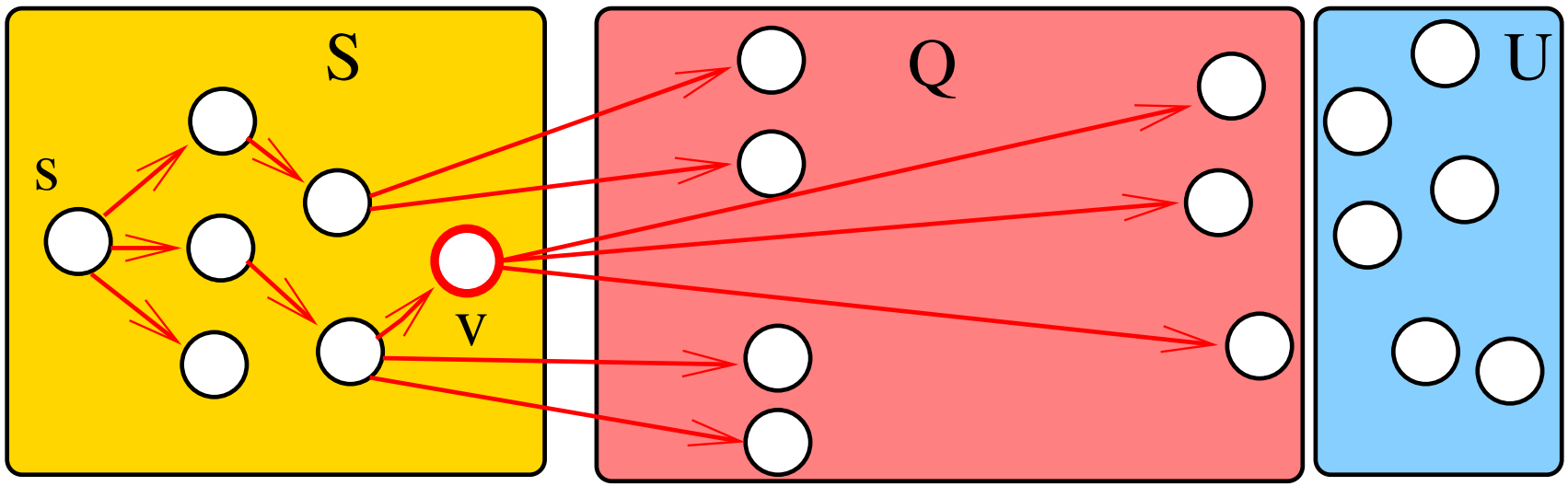
# Iteração



# Iteração



# Iteração



# Consumo de tempo

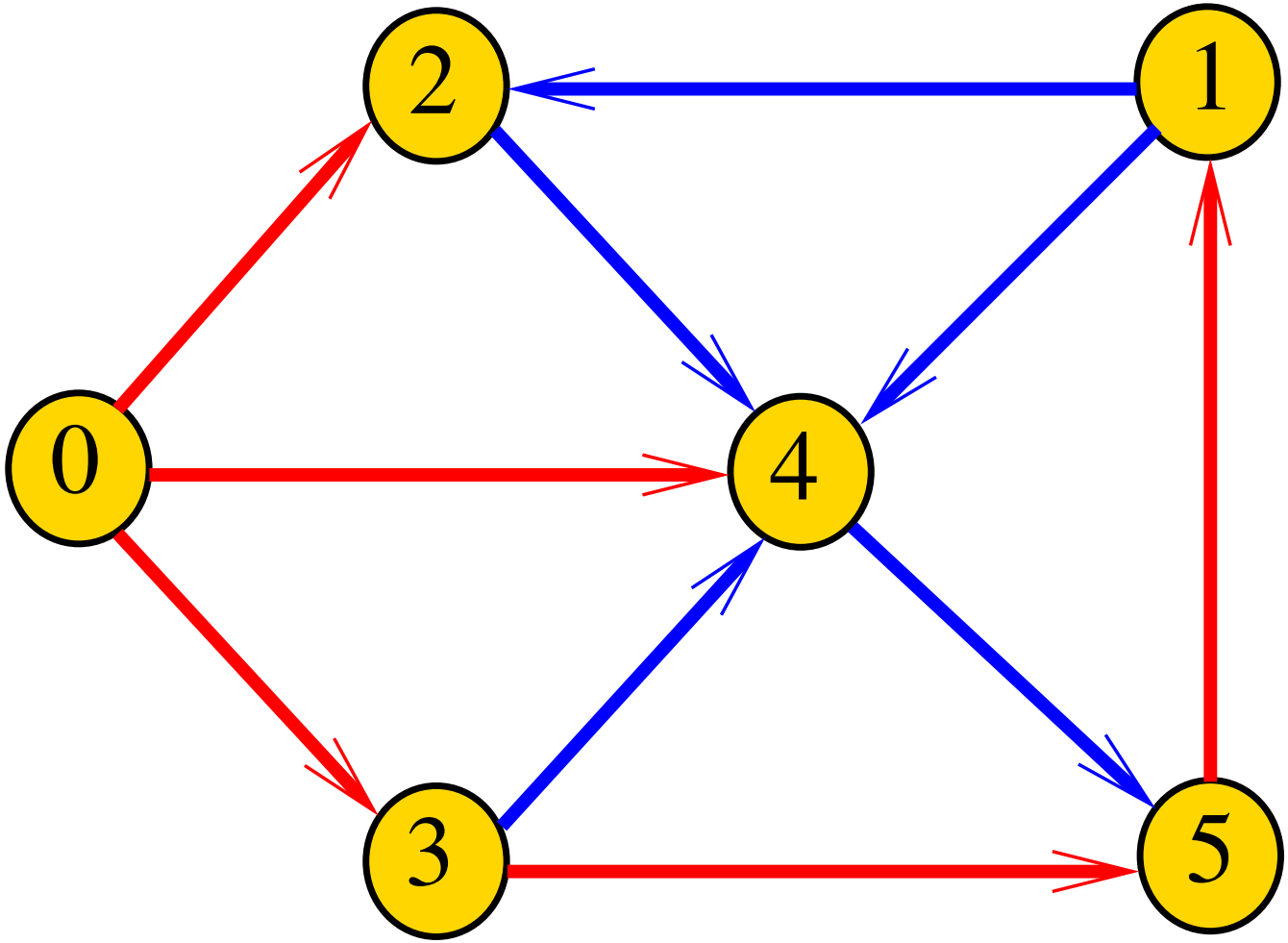
O consumo de tempo de `BFSpaths` para vetor de listas de adjacência é $O(V + E)$.

O consumo de tempo de `BFSpaths` para matriz de adjacência é $O(V^2)$.

# Arborescência da BFS

| v | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| edgeTo | 0 | 5 | 0 | 0 | 0 | 3 |

| v | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| distTo | 0 | 3 | 1 | 1 | 1 | 2 |



# Condição de inexistência

Se `distTo[t]==INFINITO` para algum vértice `t`, então

$$\texttt{S} = \{\texttt{v} : \texttt{distTo}[\texttt{v}] < \texttt{INFINITO}\}$$
$$\texttt{T} = \{\texttt{v} : \texttt{distTo}[\texttt{v}] == \texttt{INFINITO}\}$$

formam um `st`-corte $(\texttt{S}, \texttt{T})$ em que todo arco no corte tem ponta inicial em `T` e ponta final em `S`

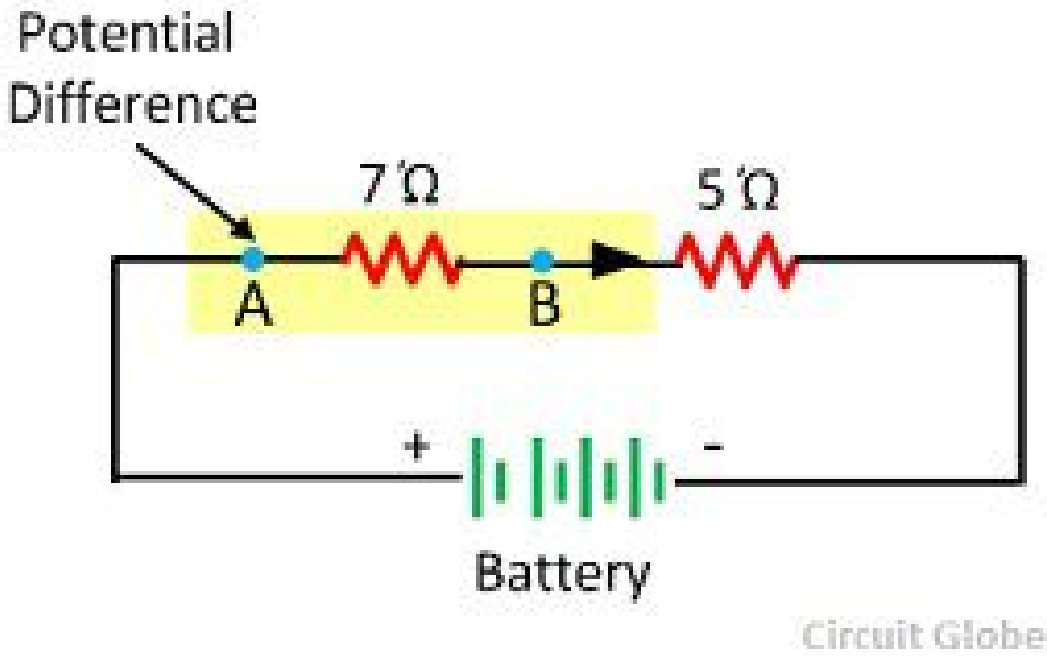
# Conclusão

Para quaisquer vértices $s$ e $t$ de um digrafo, vale uma e apenas umas das seguintes afirmações:

- existe um caminho de $s$ a $t$
- existe $st$-corte $(S, T)$ em que todo arco no corte tem ponta inicial em $T$ e ponta final em $S$.


Fonte: Yin Yang Bonsai vector image

# Apêndice: 1-potenciais




Fonte: Difference Between Electromotive Force & Potential Difference

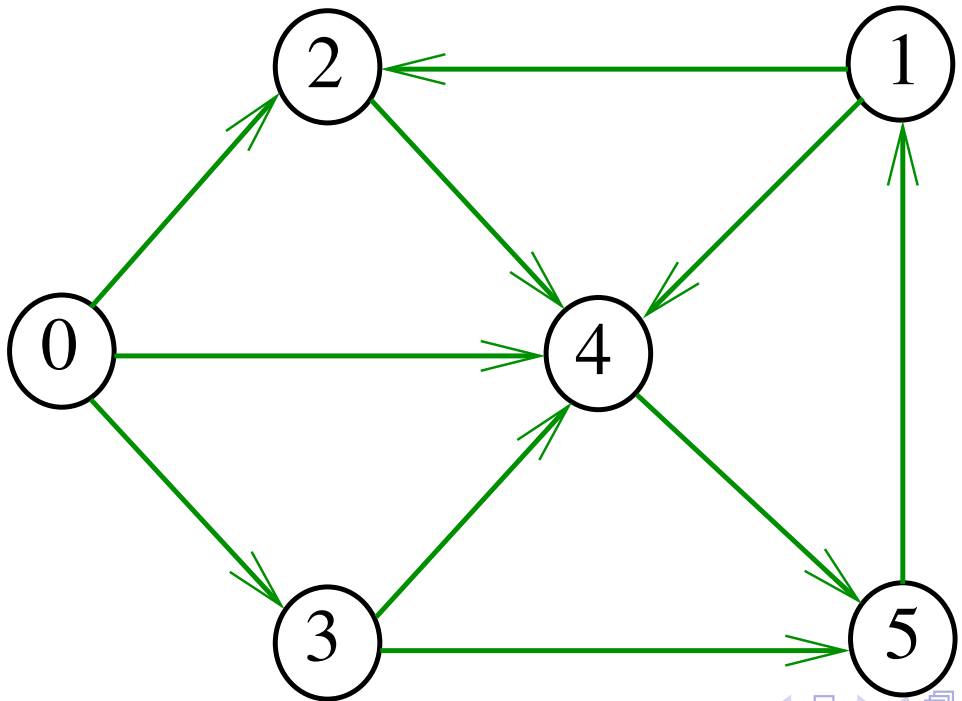
# 1-potenciais

Um **1-potencial** é um vetor $y$ indexado pelos vértices do digrafo tal que

$$y[w] - y[v] \leq 1 \text{ para todo arco } v\text{-}w$$

Exemplo:

| $v$ | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| $y[v]$ | 1 | 1 | 1 | 1 | 1 | 1 |



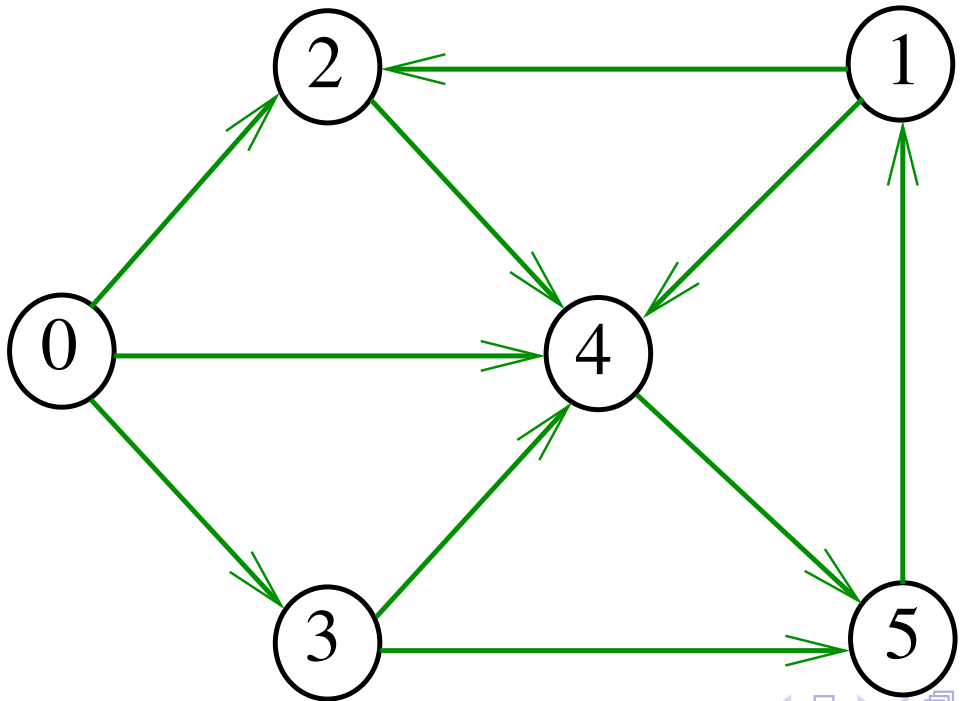
# 1-potenciais

Um **1-potencial** é um vetor $y$ indexado pelos vértices do digrafo tal que

$$y[w] - y[v] \leq 1 \text{ para todo arco } v\text{-}w$$

Exemplo:

| $v$ | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| $y[v]$ | 1 | 2 | 2 | 1 | 1 | 2 |



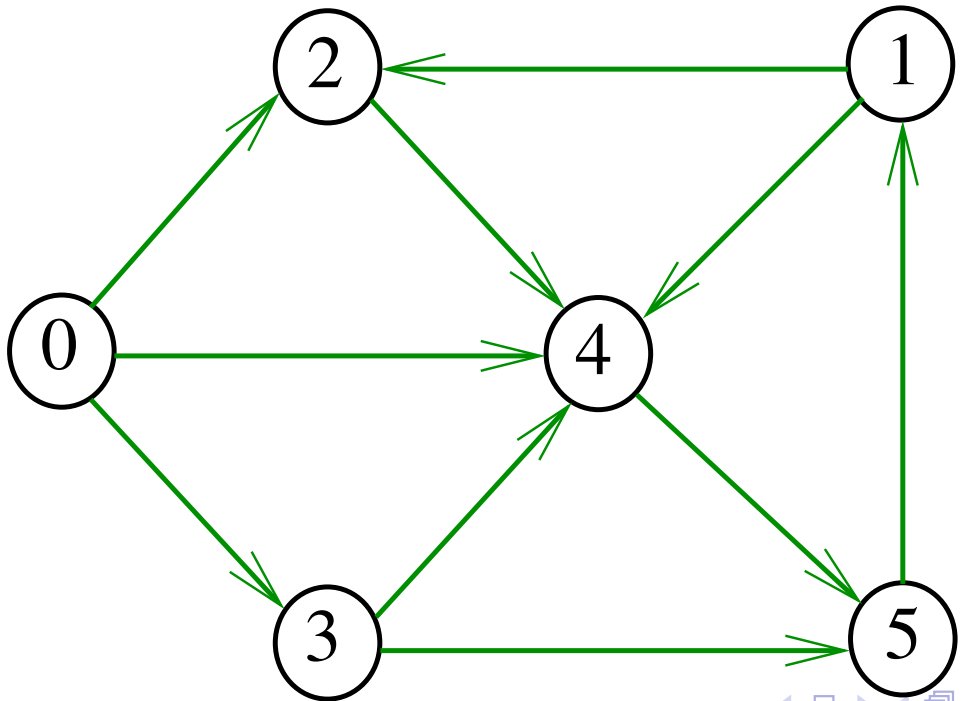
# Propriedade dos 1-potencias

Lema da dualidade. Se $y$ é um 1-potencial e $P$ é um caminho de $s$ a $t$, então

$$y[t] - y[s] \leq |P|$$

Exemplo:

| $v$ | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| $y[v]$ | 6 | 6 | 6 | 7 | 7 | 7 |



# Conseqüência

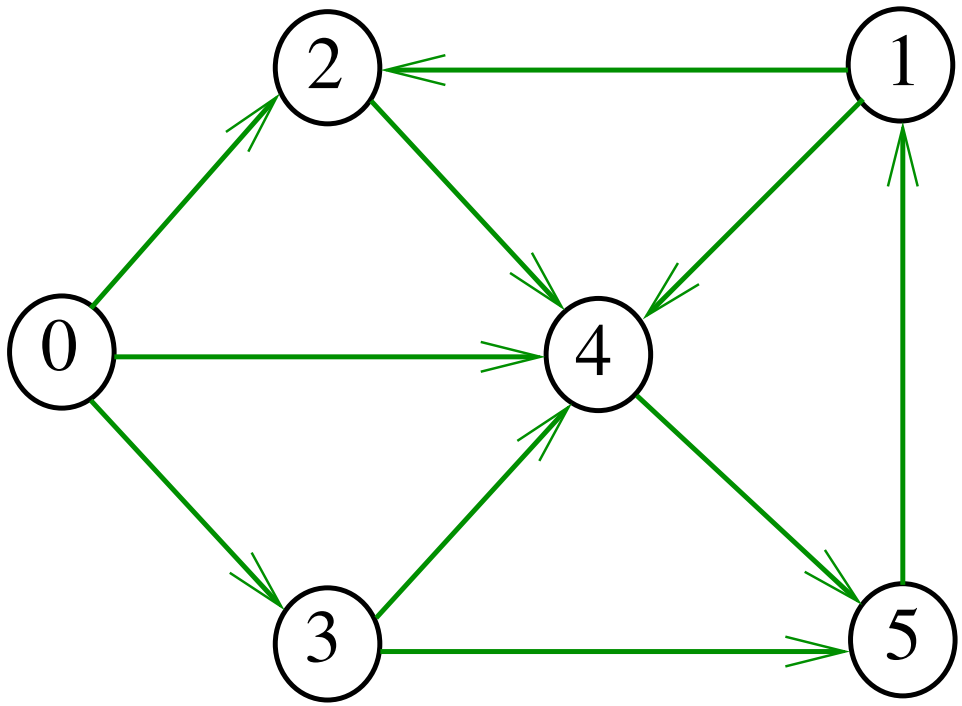
Se $\mathtt{P}$ é um caminho de $\mathtt{s}$ a $\mathtt{t}$ e $\mathtt{y}$ é um 1-potencial tais que

$$|\mathtt{P}| = \mathtt{y}[\mathtt{t}] - \mathtt{y}[\mathtt{s}],$$

então $\mathtt{P}$ é um caminho **mínimo** e $\mathtt{y}$ é um 1-potencial tal que $\mathtt{y}[\mathtt{t}] - \mathtt{y}[\mathtt{s}]$ é **máximo**

# Exemplo

| v | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| y[v] | 0 | 3 | 1 | 1 | 1 | 2 |



# Invariantes

Abaixo está escrito `y` no papel de `distTo[]`
Na classe `BFSpaths`, no início `while` do método `bfs()` valem as seguintes invariantes:

(i0) para cada arco `v-w` na arborescência `BFS` tem-se que $y[w] - y[v] = 1$;

(i1) $edgeTo[s] = s$ e $y[s] = 0$;

(i2) para cada vértice `v`, $y[v] \neq G.V() \Leftrightarrow edgeTo[v] \neq -1$;

(i3) para cada vértice `v`, se $y[v] \neq G.V()$ então existe um caminho de `s` a `v` na arborescência `BFS`.

# Invariantes (continuação)

Abaixo está escrito `y` no papel de `distTo[]`
Na linha

```
int v = q.dequeue();
```

do método `bfs()` vale a seguinte relação invariante:

(i4) para cada arco `v-w` se

$$y[w] - y[v] > 1$$

então `v` está na fila.

# Correção de BFSpaths

Início da última iteração:

- y é um 1-potencial, por (i4)
- se y[t] ≠ G.V(), então edgeTo[t] ≠ −1 [(i2)]. Logo, de (i3), segue que existe um st-caminho P na arborescência BFS. (i0) e (i1) implicam que

$$|P| = y[t] - y[s] = y[t].$$

  Da propriedade dos 1-potencias, concluímos que P é um st-caminho de comprimento mínimo
- se y[t] = G.V(), então (i1) implica que y[t] − y[s] = G.V() e da propriedade dos 1-potencias concluímos que não existe caminho de s a t no grafo

# Tipo teorema da dualidade

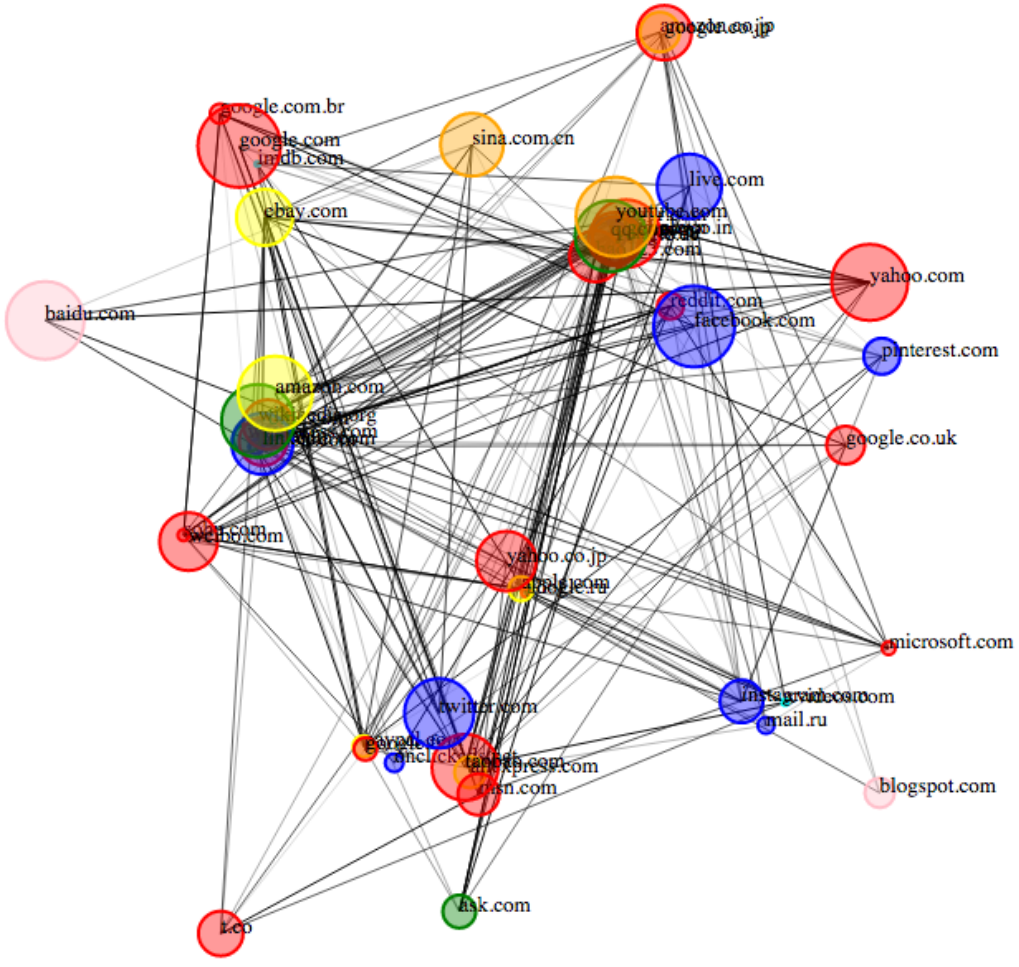
Da propriedade dos 1-potenciais (lema da dualidade) e da correção de `bfs()` concluímos o seguinte:

Se `s` e `t` são vértices de um digrafo e `t` está ao alcance de `s` então

$$\min\{|\mathtt{P}| : \mathtt{P} \text{ é um } \mathtt{st}\text{-caminho}\}$$
$$= \max\{\mathtt{y}[\mathtt{t}] - \mathtt{y}[\mathtt{s}] : \mathtt{y} \text{ é um } 1\text{-potencial}\}.$$

Fonte: Yin Yang Meaning

# Custos nos arcos




Fonte: Force-directed graph drawing

# Digrafos com custos nos arcos

Muitas aplicações associam um número a cada arco de um digrafo
Diremos que esse número é o **custo** ou **peso** do arco
Vamos supor que esses números são do tipo `double` na classe `Arc`.

```
private final double weight;
```

# Classe Arc: esqueleto

```
public class Arc {
   private final int v;
   private final int w;
   private final double weight;

   public Arc(int v, int w,
              double weight) {...}
   public int from() {...}
   public int to() {...}
   public double weight() {...}
   public String toString() {...}
}
```

# Arc

O construtor `Arc` recebe dois vértices `v` e `w` e um valor `weight` e produz a representação de um arco com ponta inicial `v` e ponta final `w` e peso `weight`.

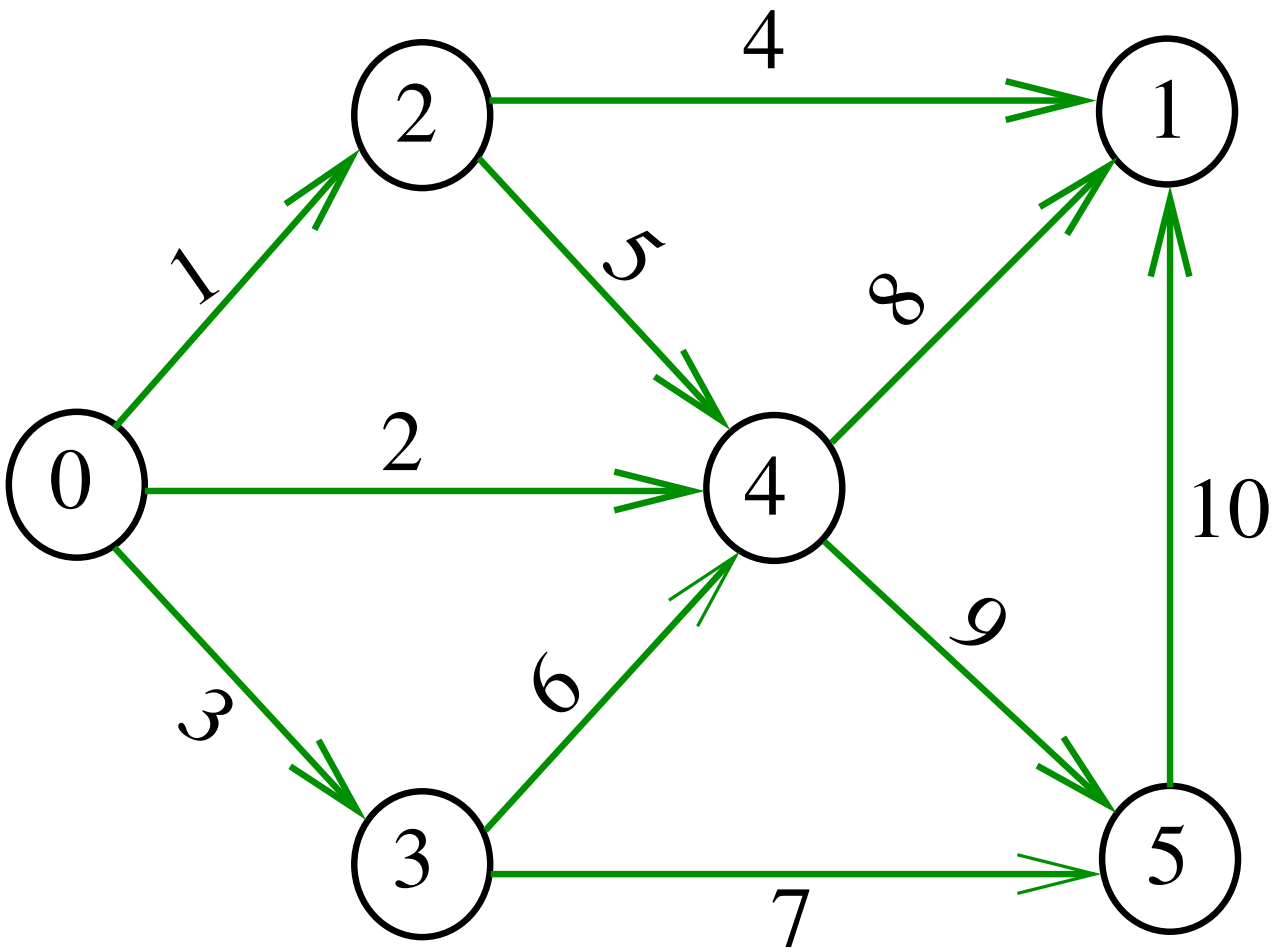
```
public Arc(int v, int w,
           double weight) {
   this.v = v;
   this.w = w;
   this.weight = weight;
}
```

# Arc

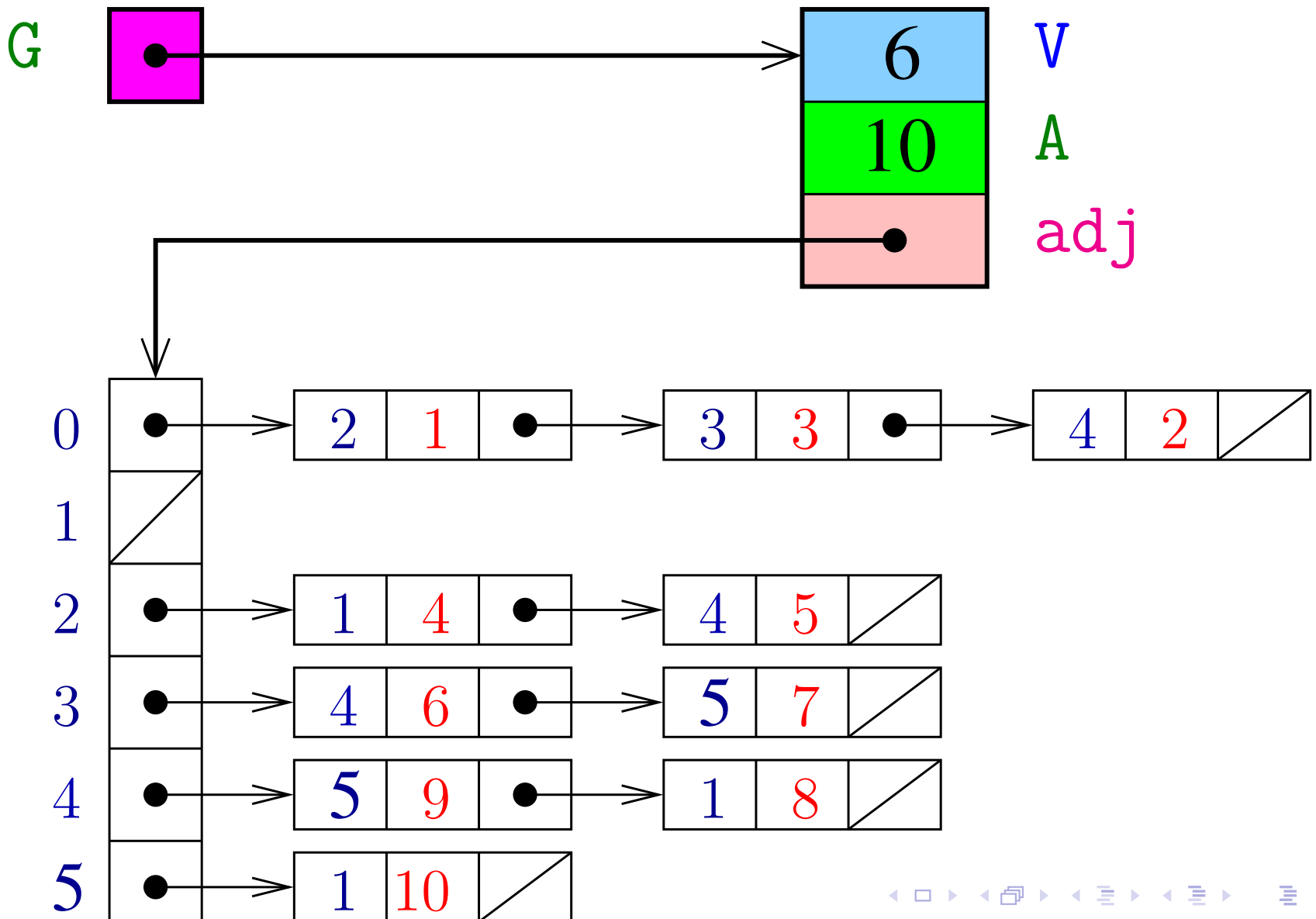
```
// retorna a ponta inicial do arco
public int from() {
   return v;
}
// retorna a ponta final do arco
public int to() {
   return w;
}
// retorna o custo/peso do arco
public double weight() {
   return weight;
}
```

# Digrafo com pesos

EdgeWeightedDigraph G



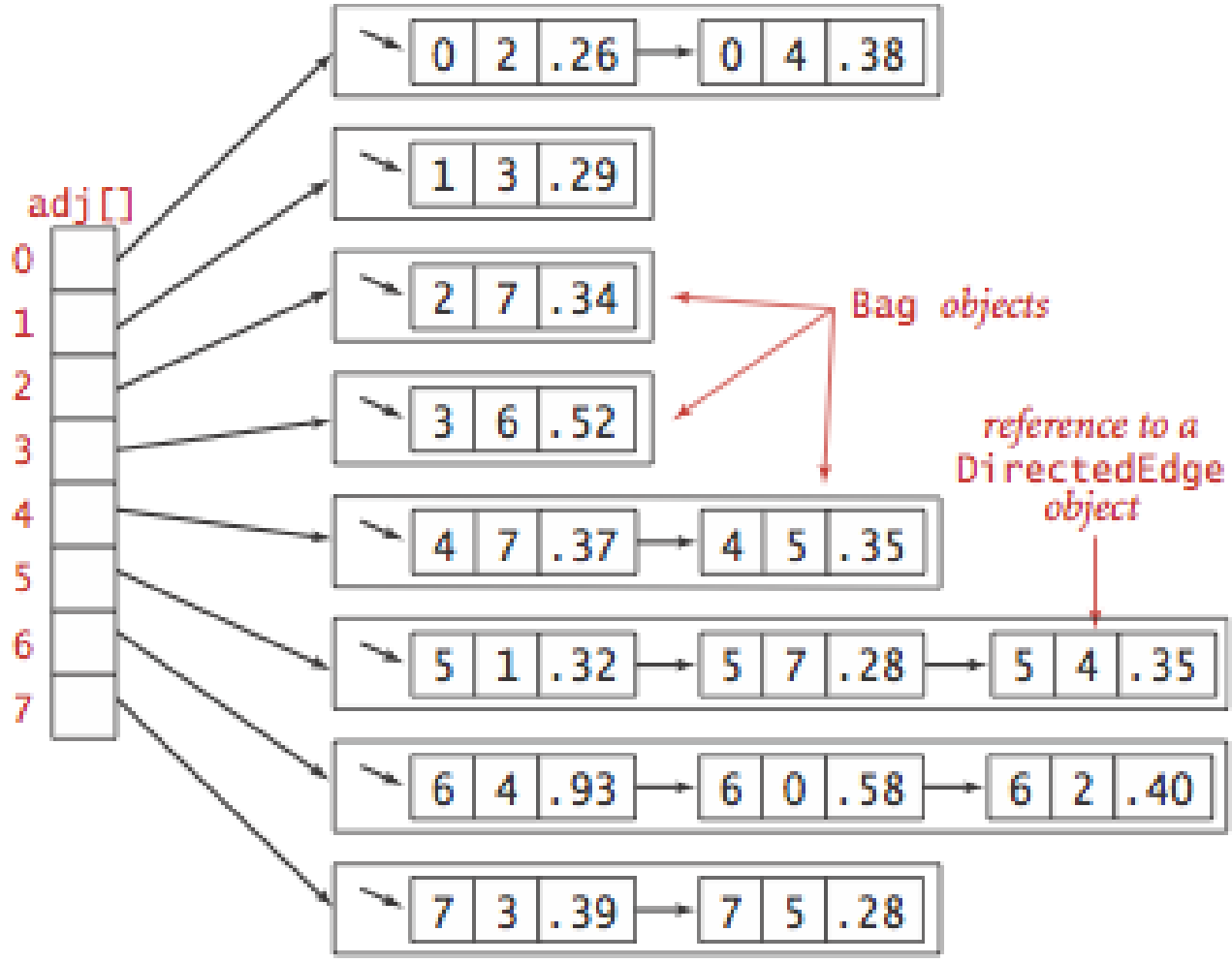
# Estruturas de dados



# Enquanto isso. . . no algs4

tinyEWD.txt

```
V
8
E
15
4 5  0.35
5 4  0.35
4 7  0.37
5 7  0.28
7 5  0.28
5 1  0.32
0 4  0.38
0 2  0.26
7 3  0.39
1 3  0.29
2 7  0.34
6 2  0.40
3 6  0.52
6 0  0.58
6 4  0.93
```




Edge-weighted digraph representation

# Classe

A estrutura `EdgeWeightedDigraph` do `algs4` representa um digrafo com pesos nos arcos.

- `V` é o número de vértices
- `E` é o número de arcos do digrafo
- `adj` é uma referência para vetor de listas de adjacência

A lista de adjacência de um vértice `v` é composta por nós do tipo `Arc`.

Um `next` de `Bag` é uma referência para um `Arc`

Cada nó da lista contém `v`, um vizinho `w` de `v`, **o custo** do arco `v-w` e o endereço do nó seguinte da lista

# Classe EWDigraph: esqueleto

```
public class EWDigraph{
   private final int V;
   private int E;
   private Bag<Arc>[] adj;
   private int[] indegree;
   public EWDigraph(int V) {...}
   public int V() { return V; }
   public int E() { return E; }
   public void addEdge(int v, int w) { }
   public Iterable<Integer> adj(int v) { }
   public int outdegree(int v) {...}
   public int indegree(int v) {...}
}
```

# Construtor de EWDigraph

Constrói um digrafo com V vértice e zero arcos.

```
public EWDigraph(int V) {
   this.V = V;
   this.E = 0;
   adj= (Bag<Arc>[]) new Bag[V];
   for (int v = 0; v < V; v++)
       adj[v] = new Bag<Arc>();
}
```

# V() e E()

```
public int V() {
      return V;
}
public int E() {
      return E;
}
```

# addEdge() e adj()

Insere um arco e no digrafo G.

```
public addEdge(Arc e) {
   int v = e.from();
   int w = e.to();
   adj[v].add(e);
   E++;
}

public Iterable<Arc> adj(int v) {
   return adj[v];
}
```

# edges()

O código abaixo retorna os arcos em G como uma coleção iterável.

```
public Iterable<Arc> edges() {
   Bag<Arc> bag;
   bag = new Bag<Arc>();
   for (int v = 0; v < V; v++)
       for (Arc e:  adj[v])
           bag.add(e);
   return bag;
}
```

# Caminhos de custo mínimo


Fonte: The Shortest Distance WoW Achievement Fast

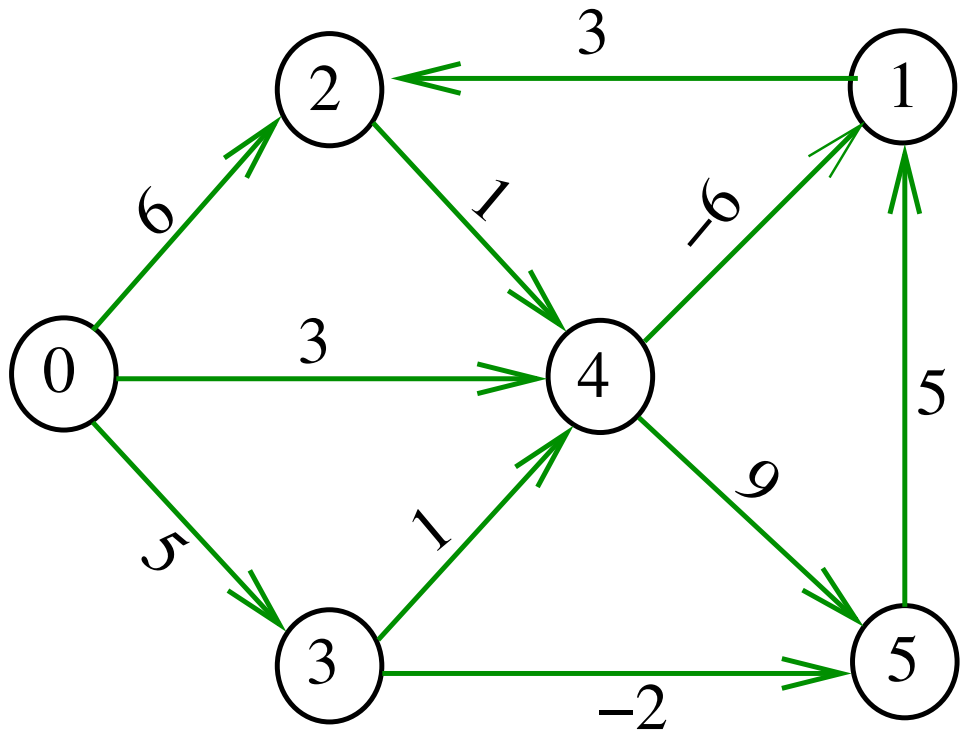
# Custo de um caminho

**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.

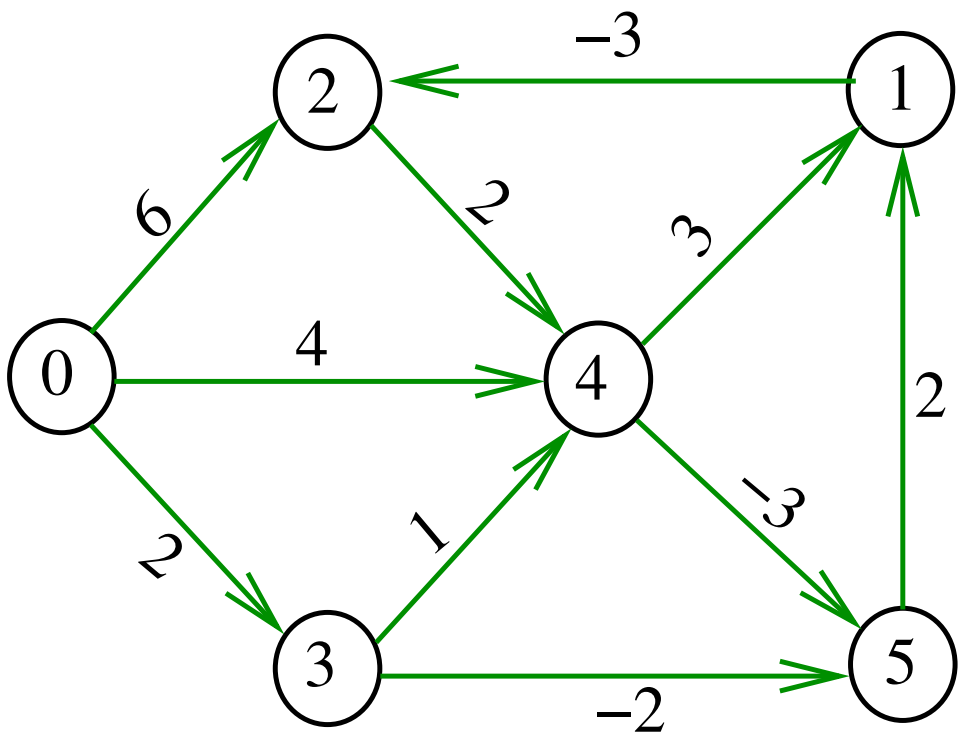
Custo do caminho 0–2–4–1–2–4–1–2–4–5 é 12.



# 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 $-1$



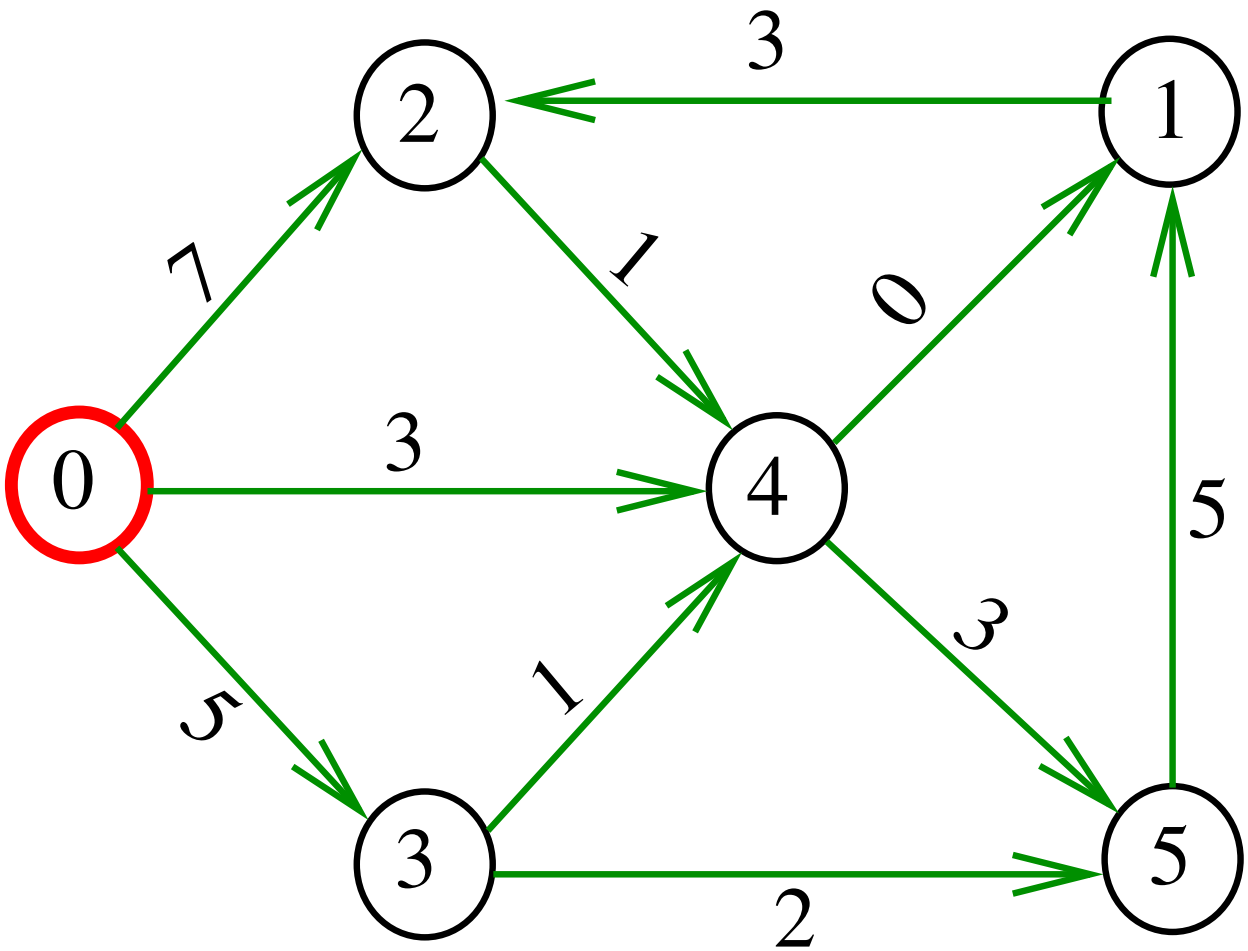
# Problema

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 $t$.*

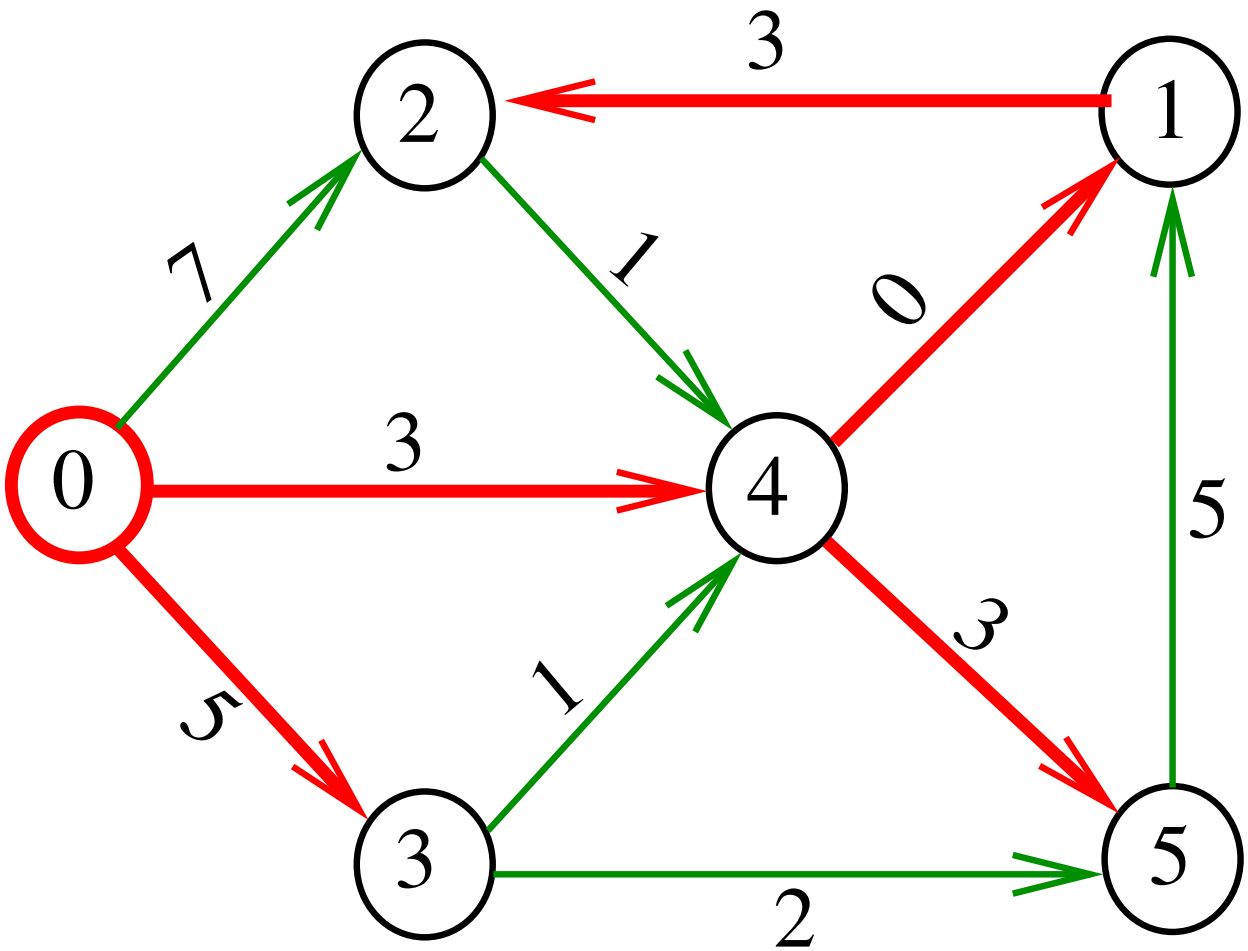
# Exemplo

Entra:



# Exemplo

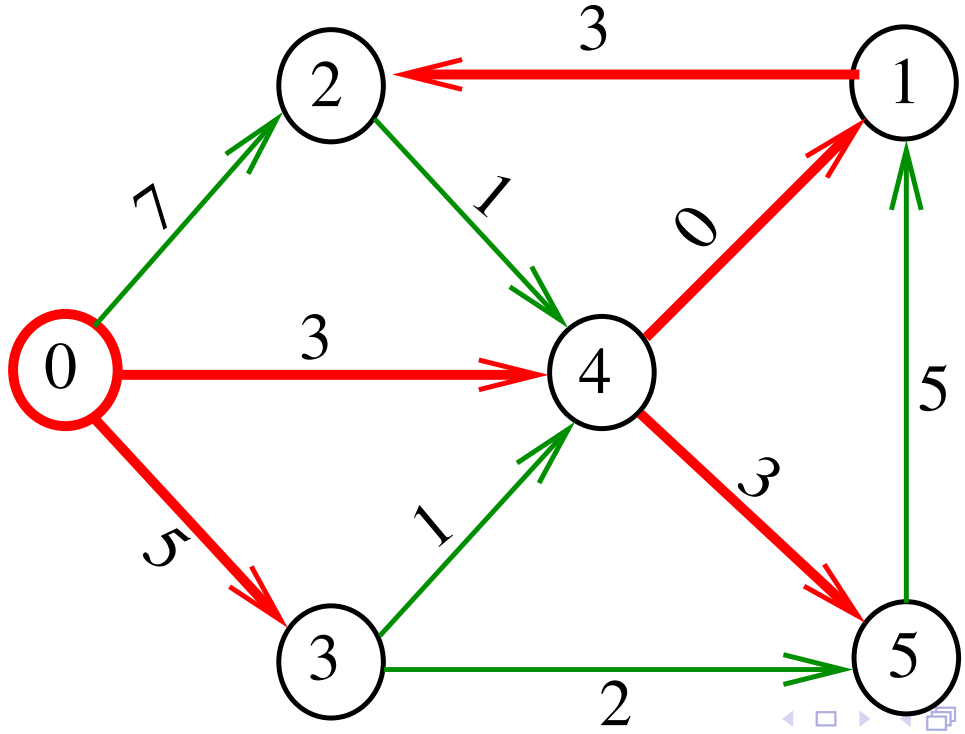
Sai:



# 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,

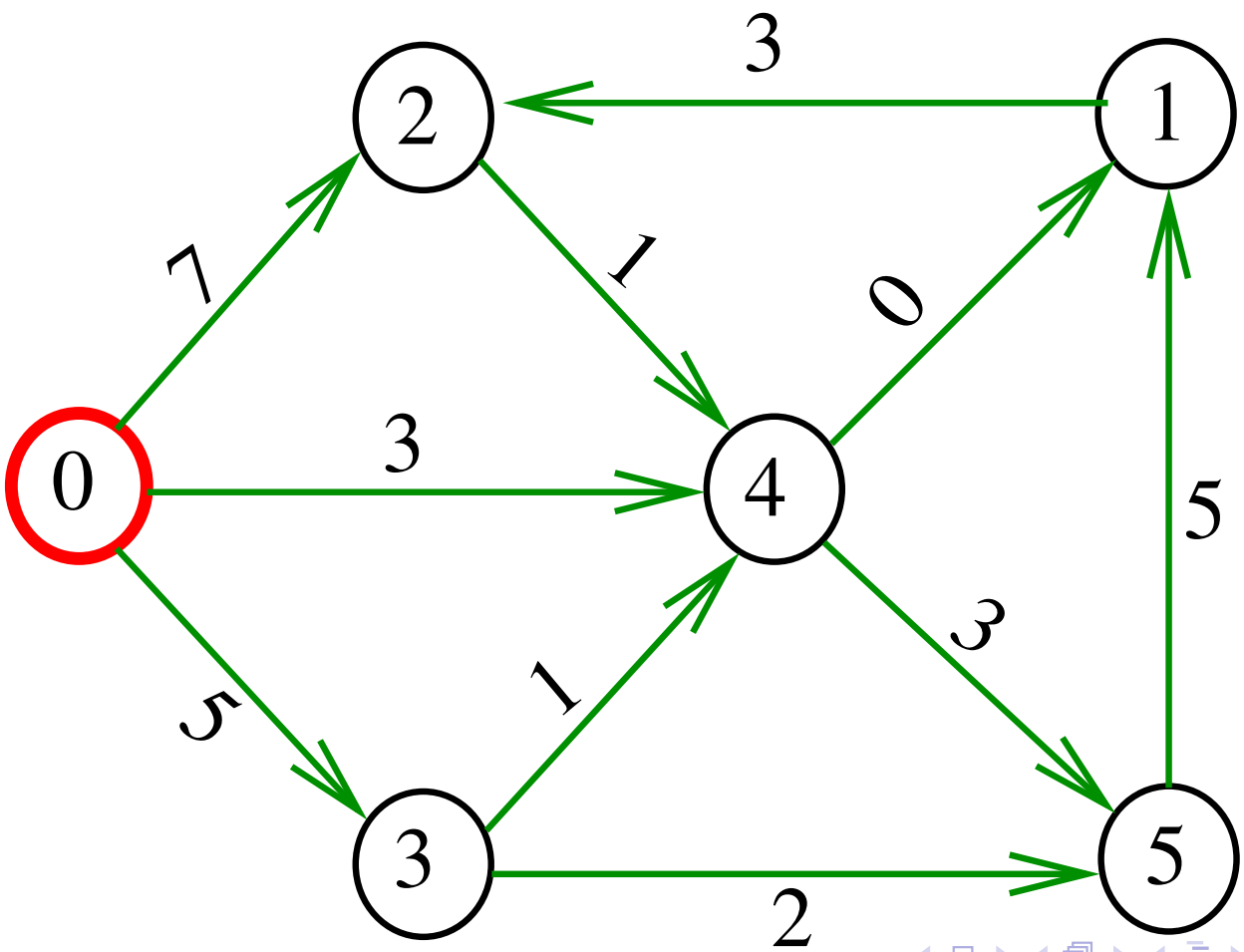
> *o único caminho de s a t na arborescência é um caminho mínimo*



# Problema da SPT

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

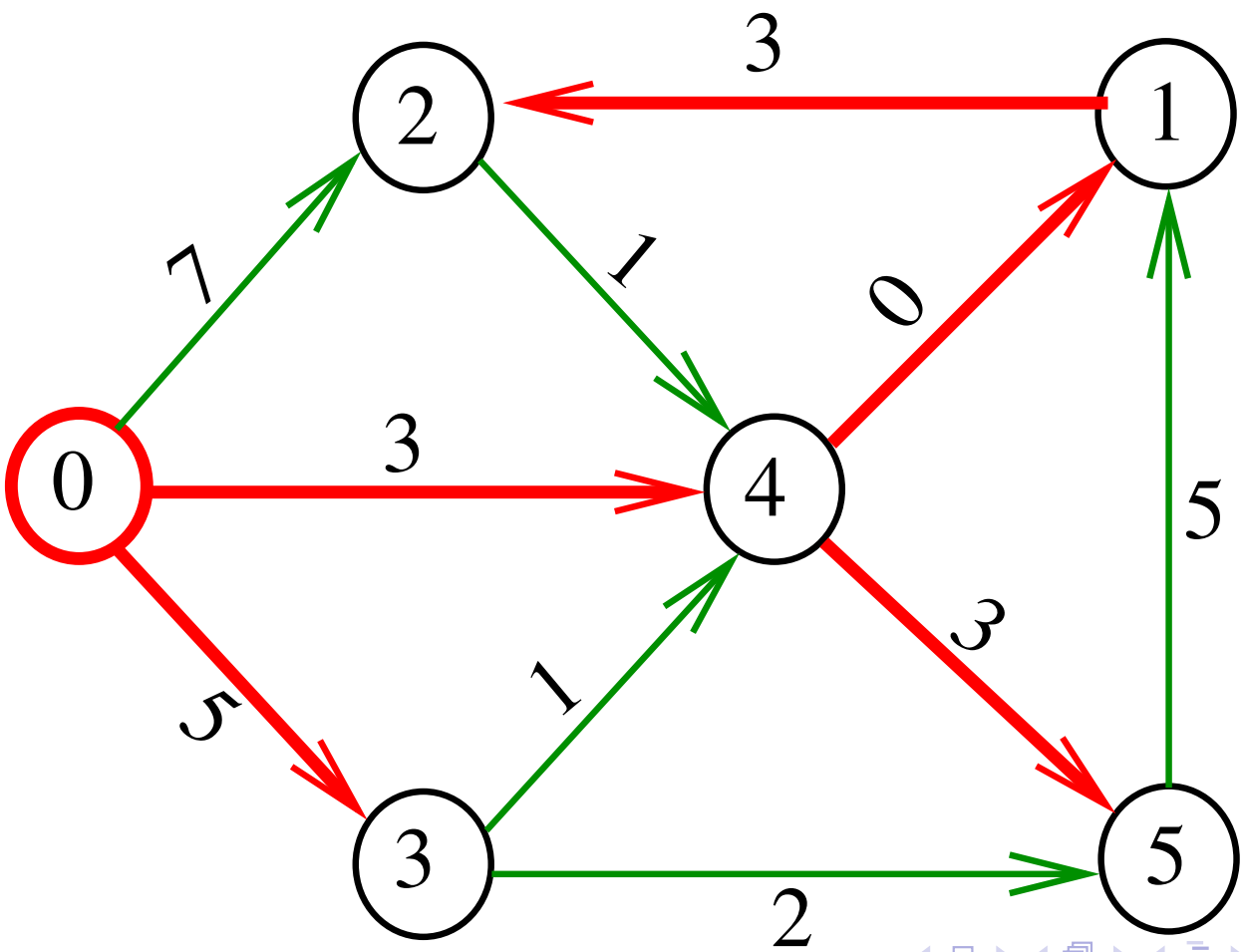
Entra:



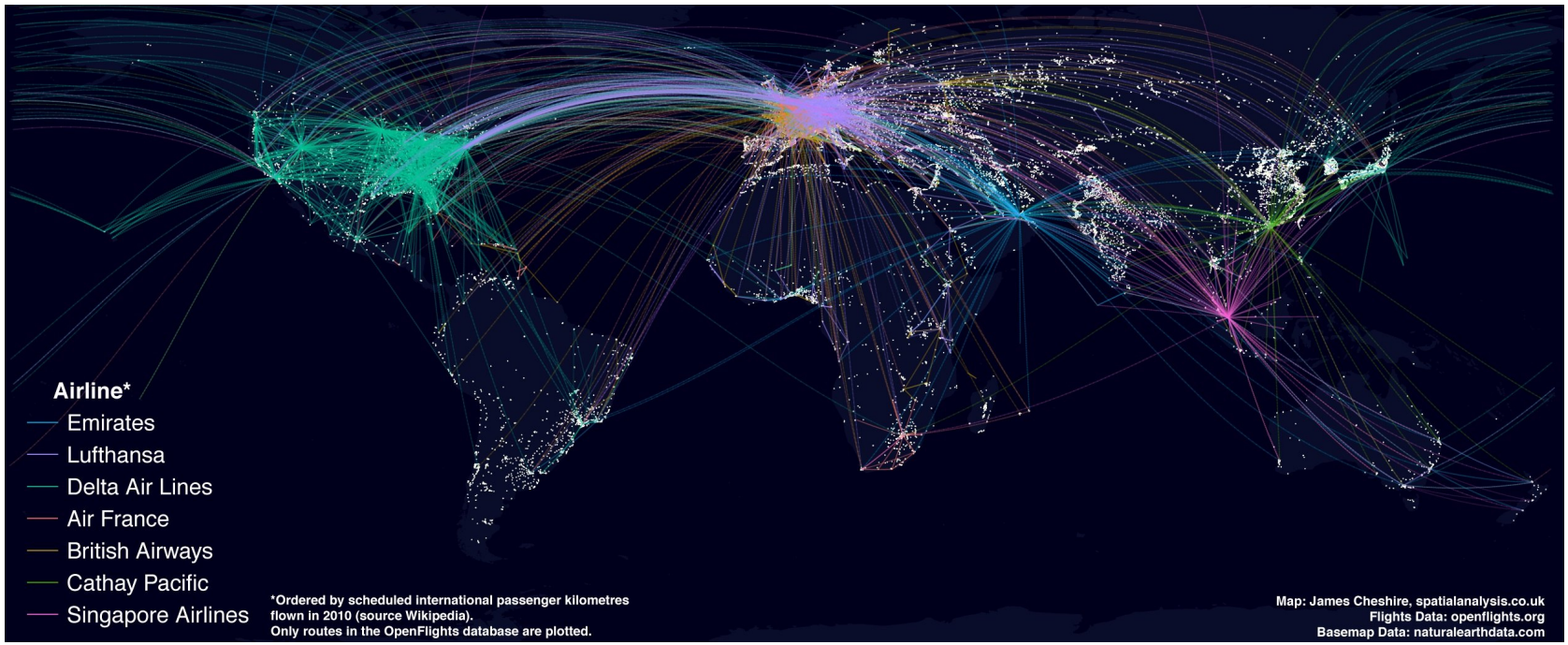
# Problema da SPT

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

Sai:



# Algoritmo de Dijkstra

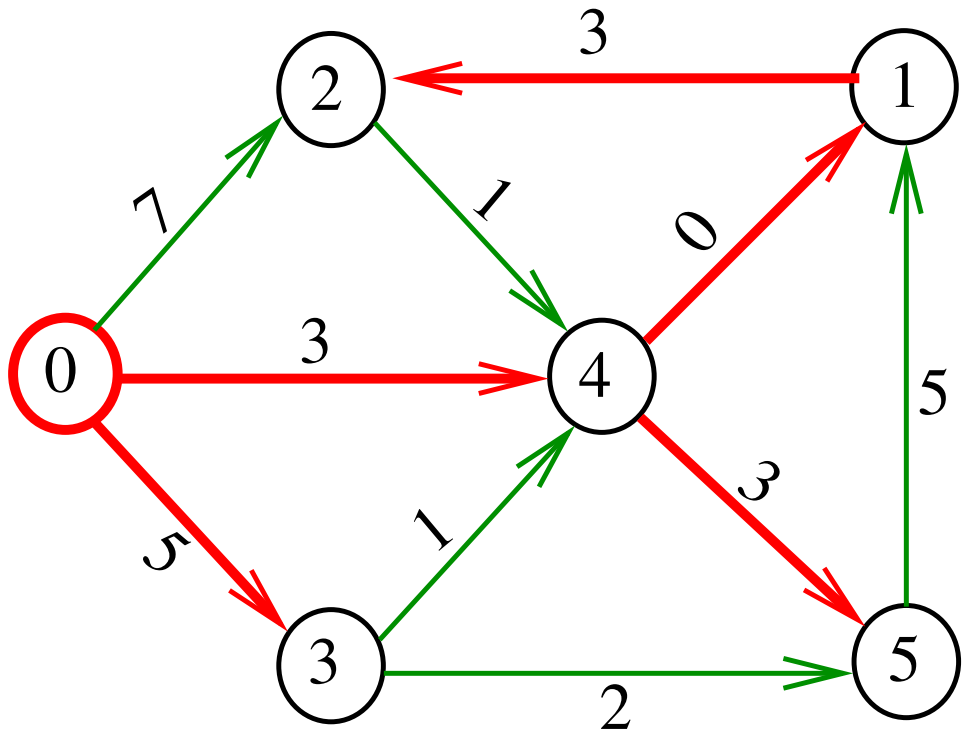



Fonte: What S So Great About A World Flight Paths Map Spatial Ly In

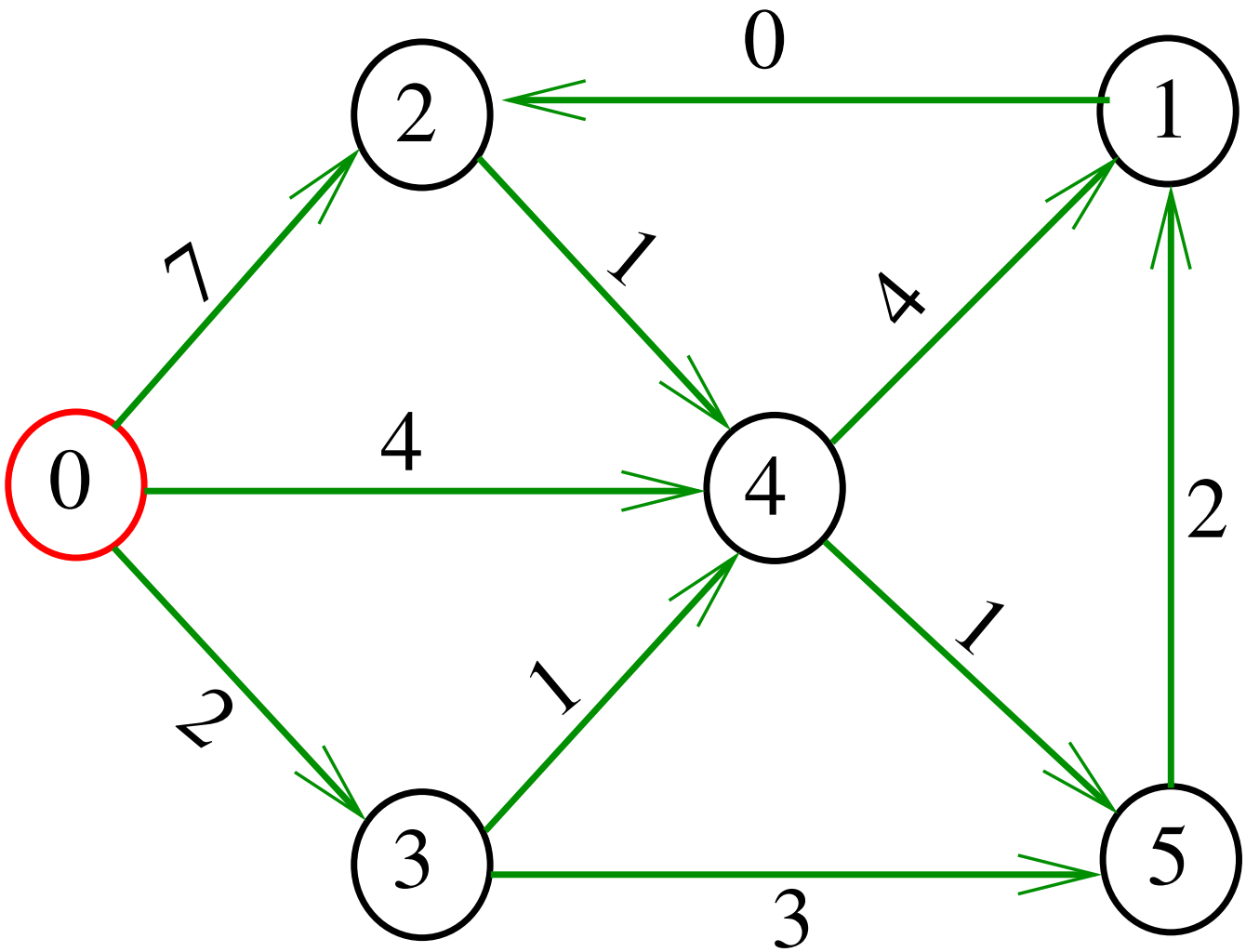
# Problema

O algoritmo de Dijkstra resolve o problema da SPT:

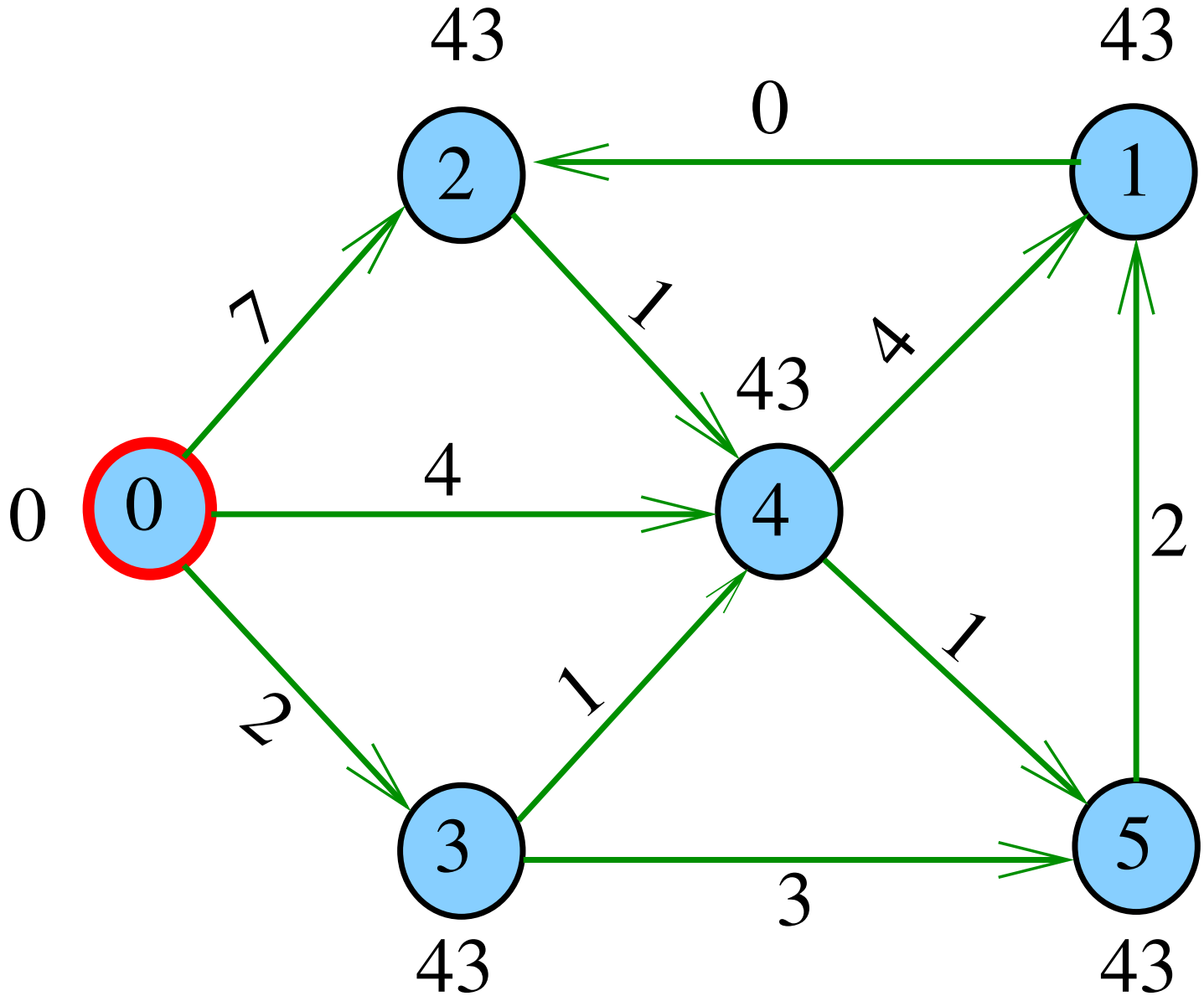
*Dado um vértice s de um digrafo com custos não-negativos nos arcos, encontrar uma SPT com raiz s*



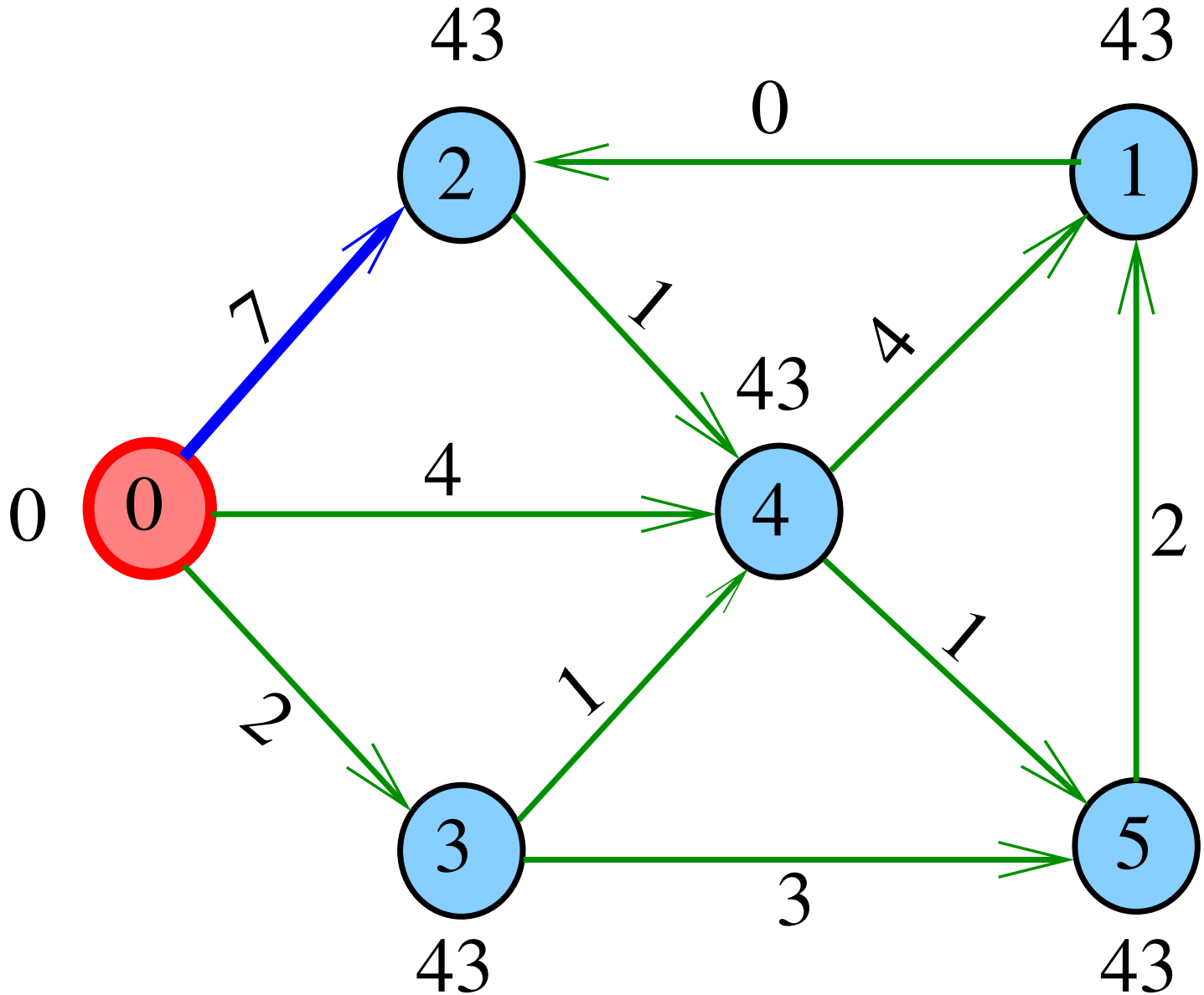
# Simulação



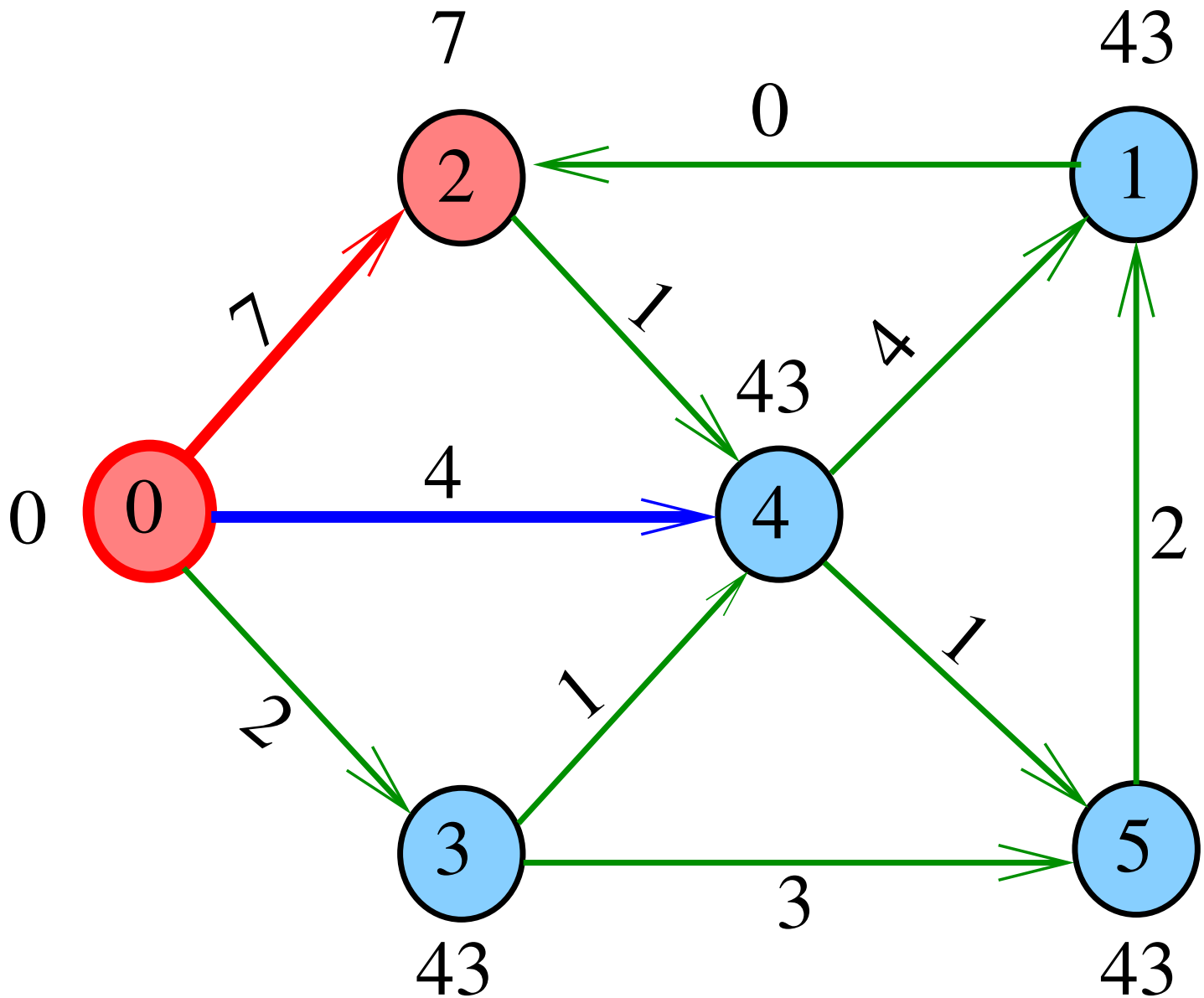
# Simulação



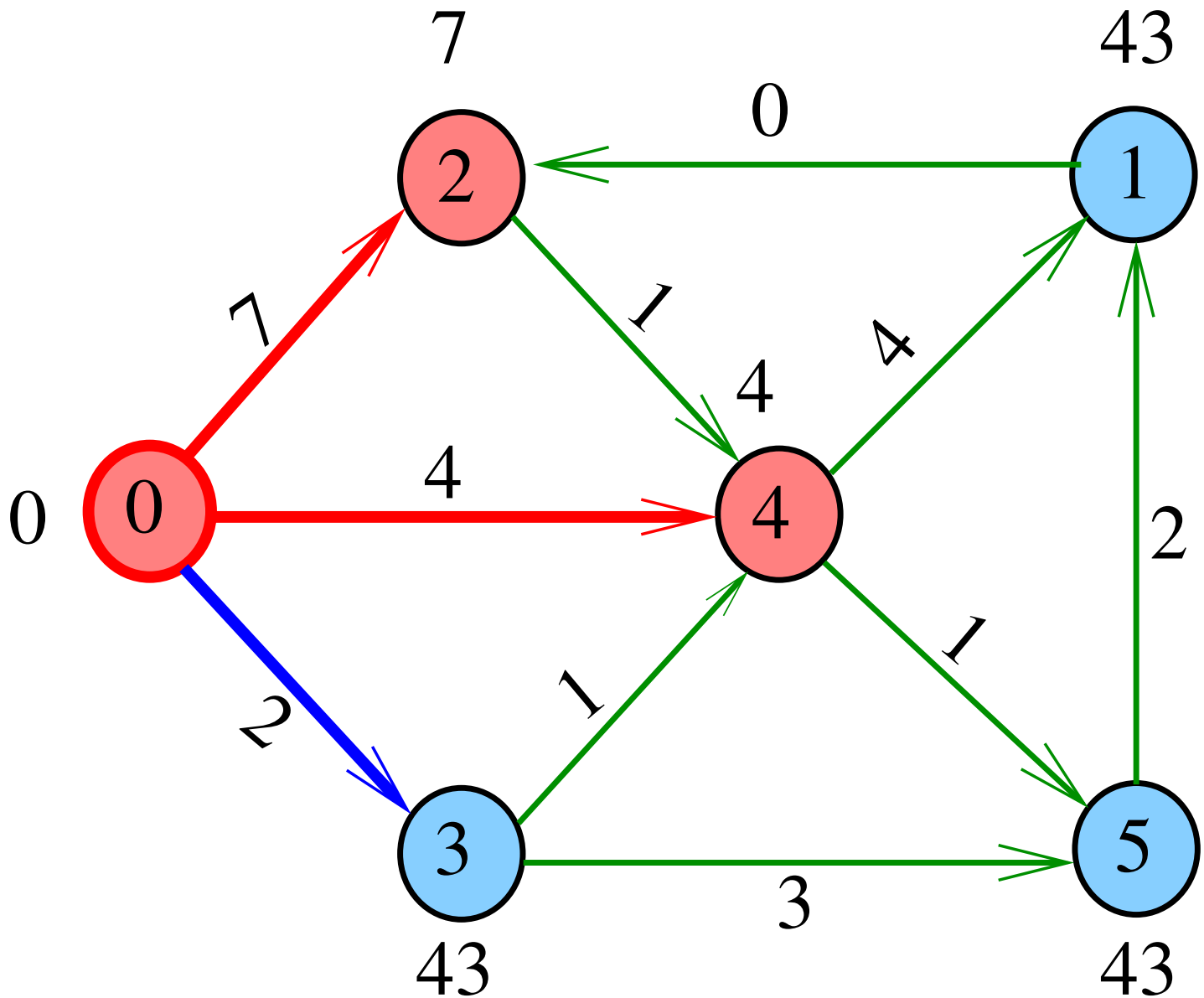
# Simulação



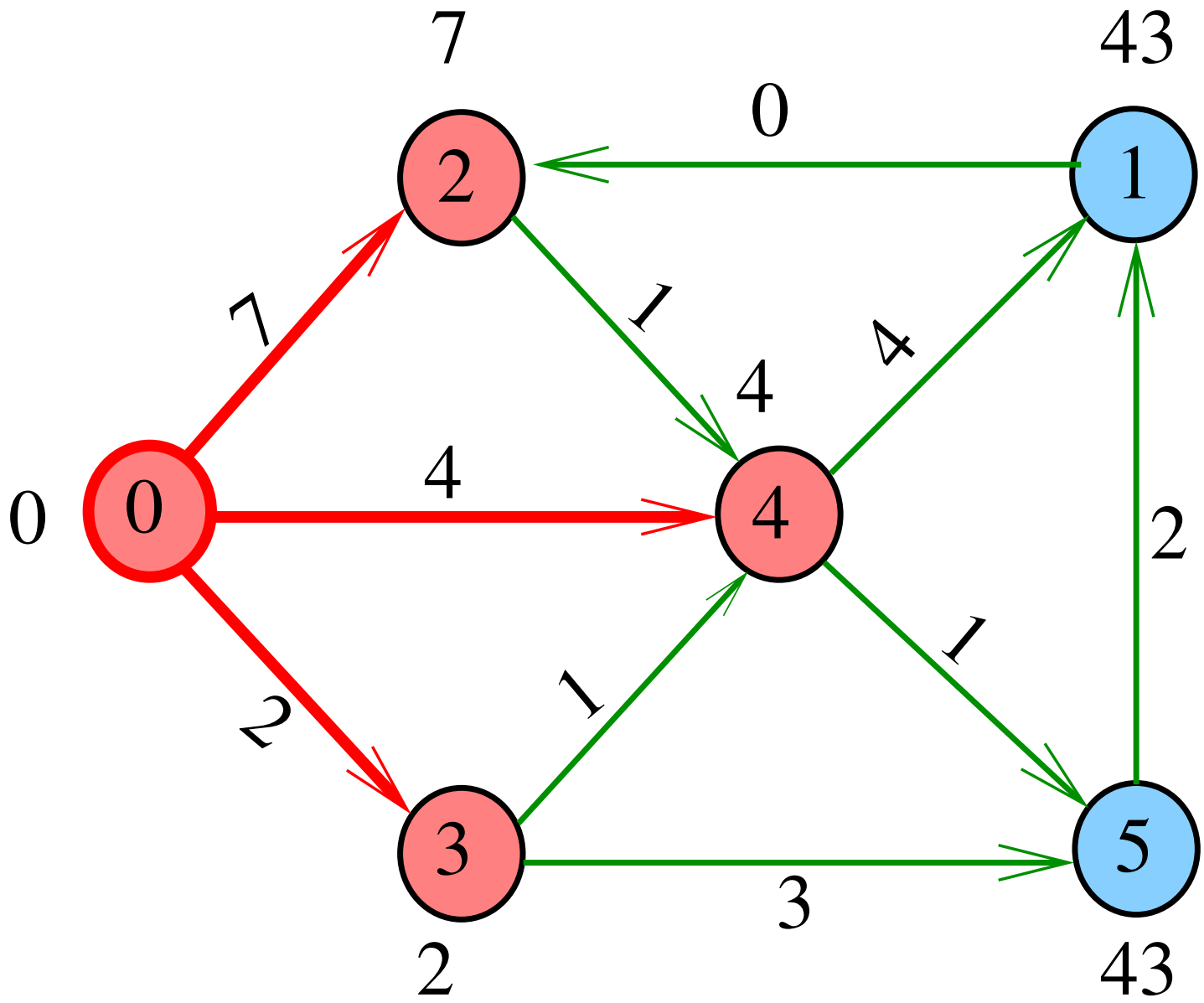
# Simulação



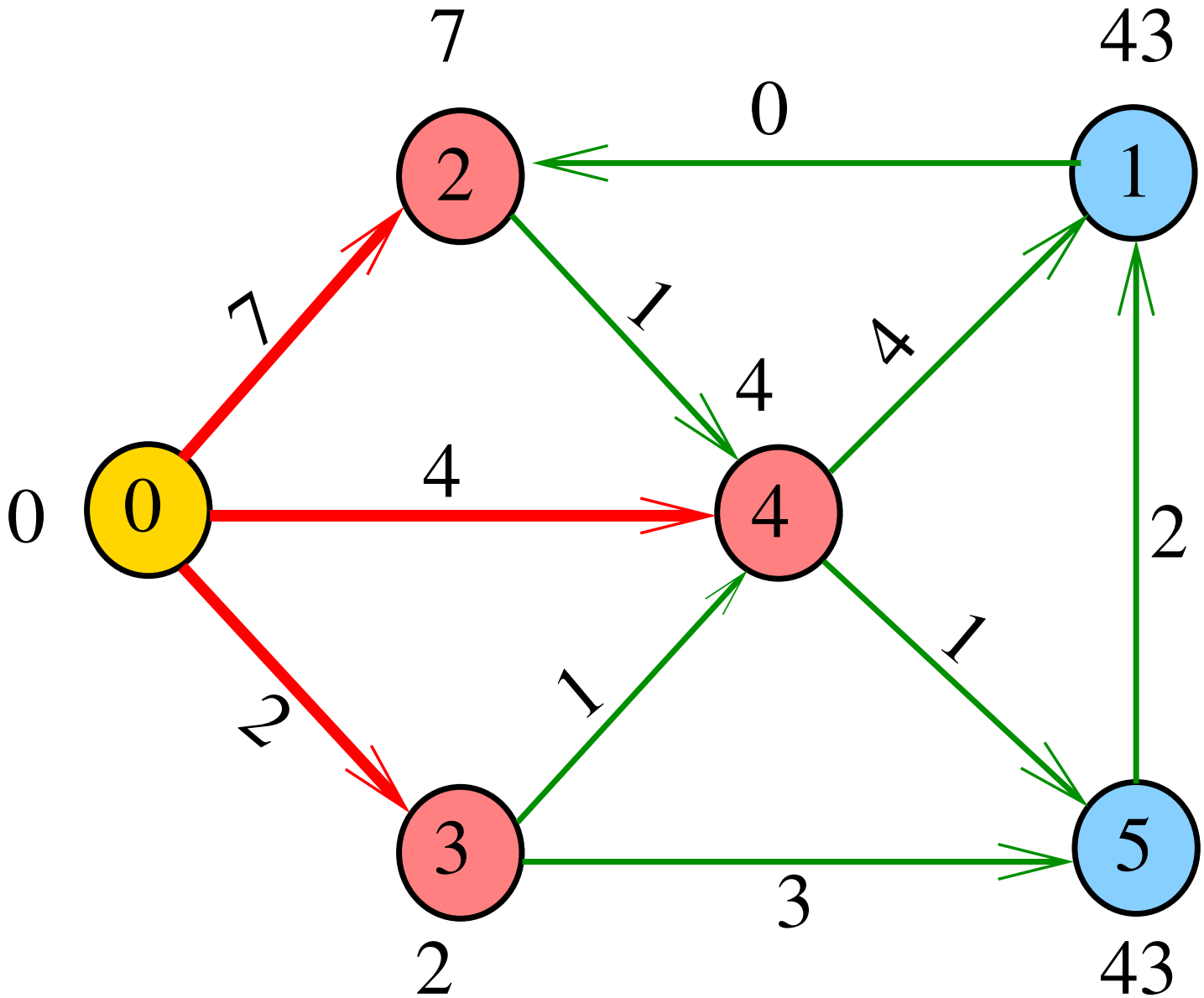
# Simulação



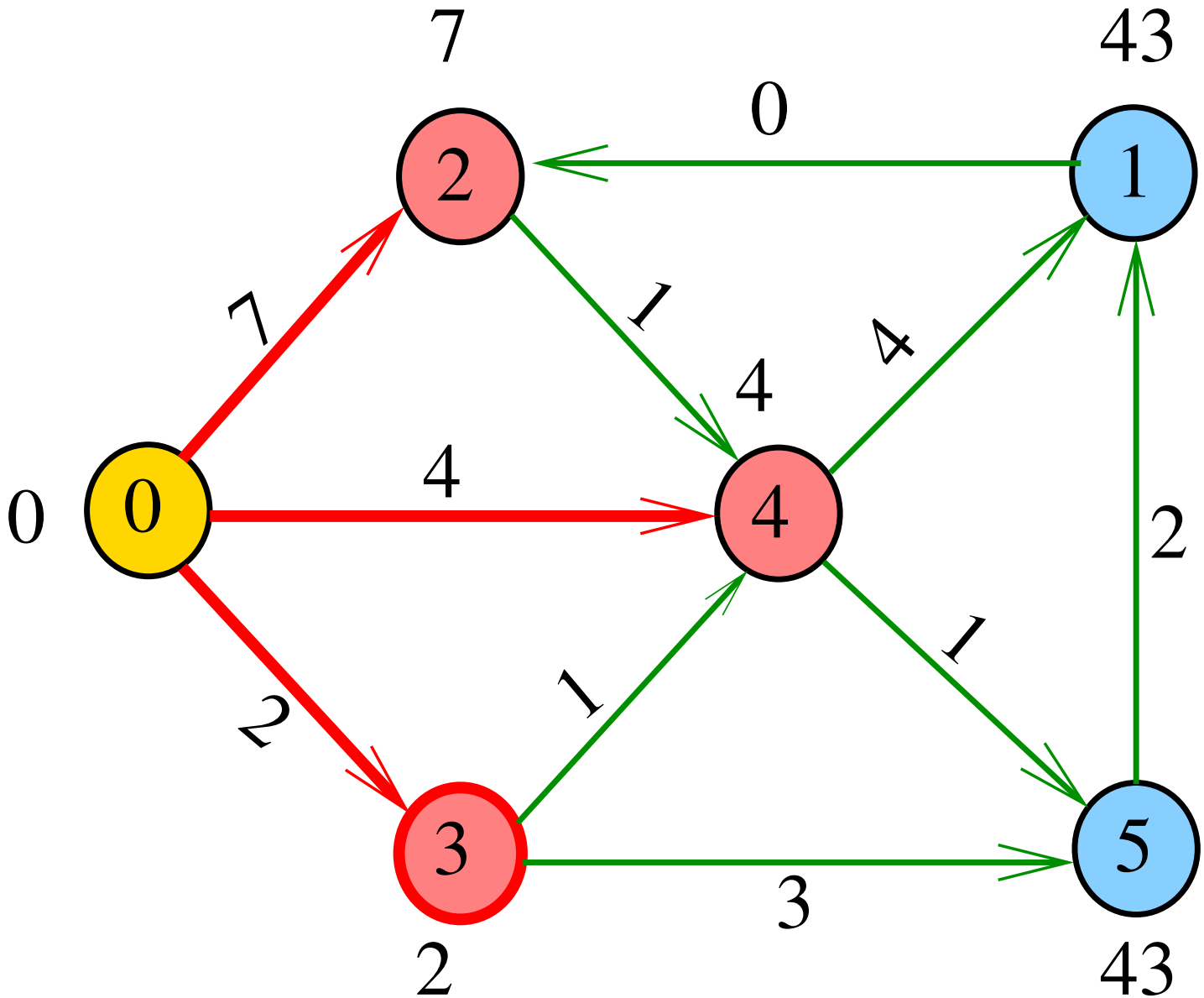
# Simulação



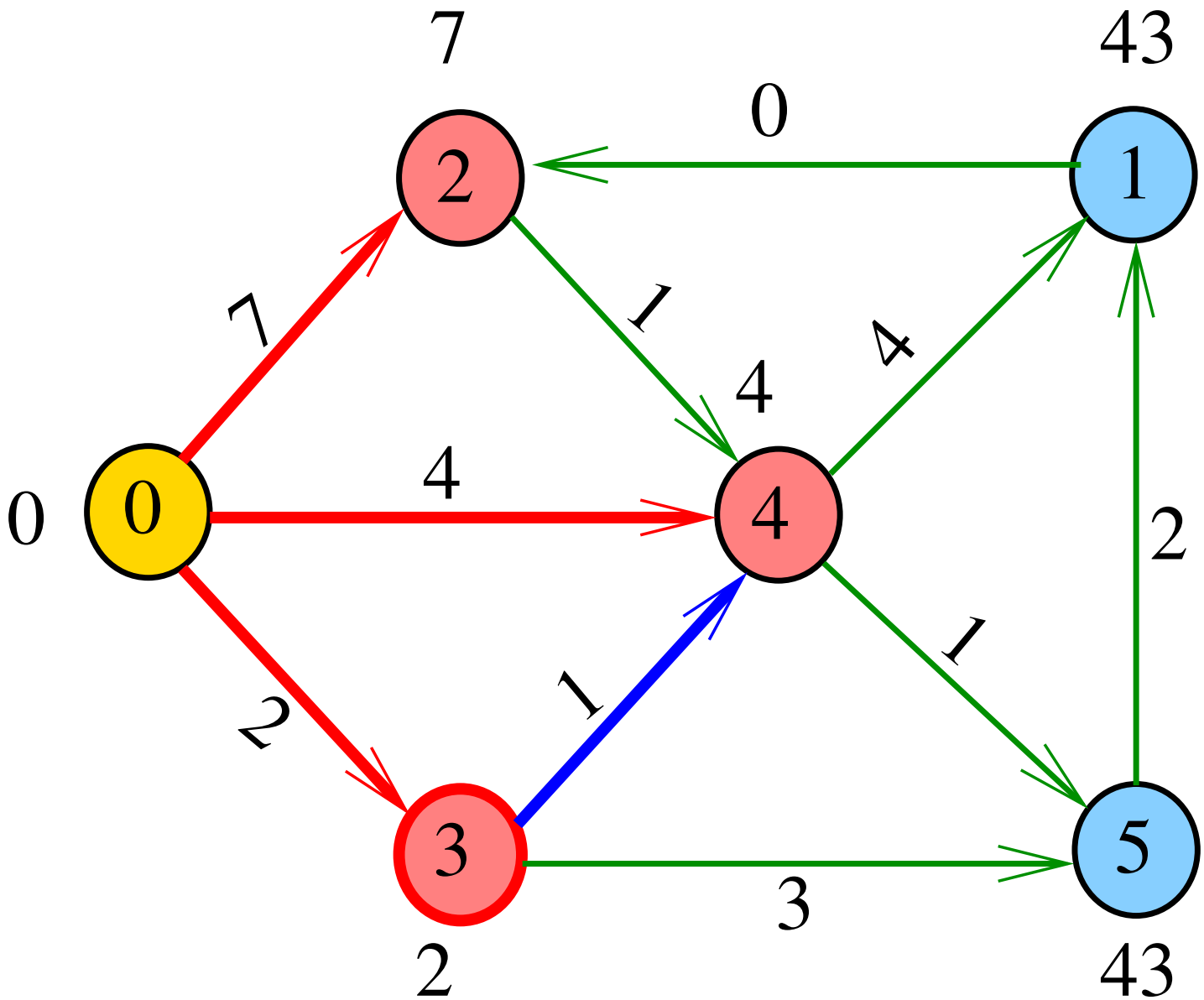
# Simulação



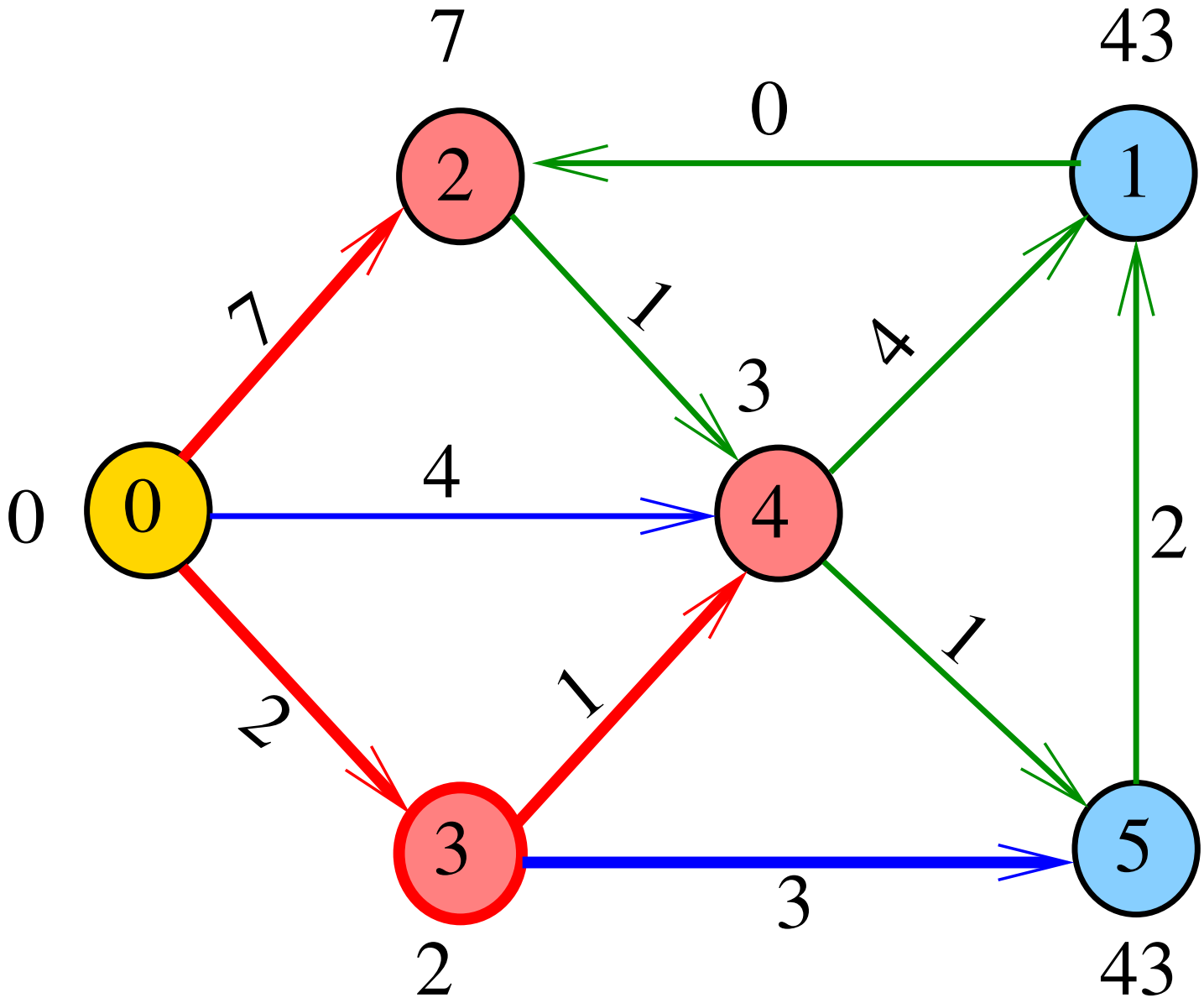
# Simulação



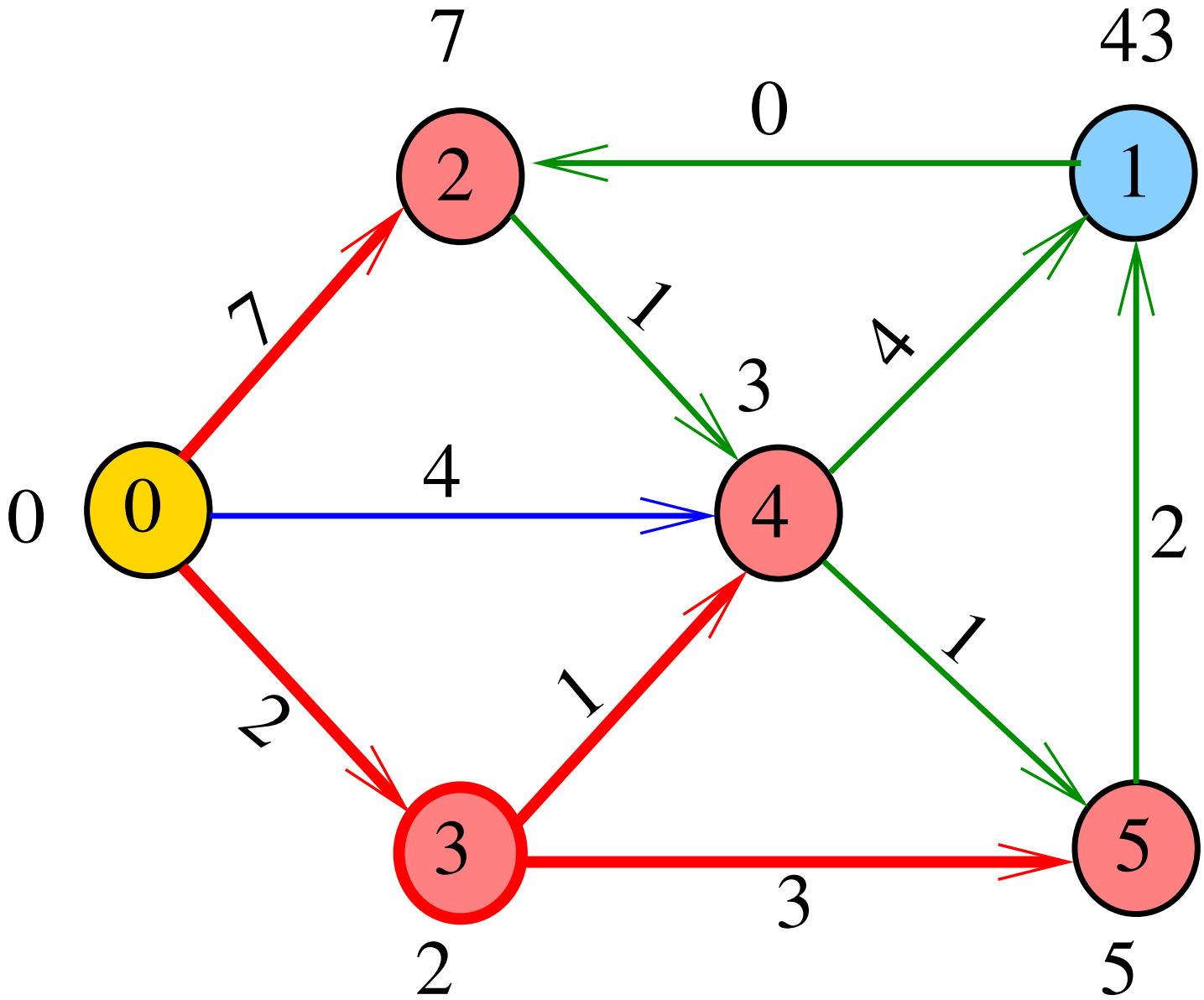
# Simulação



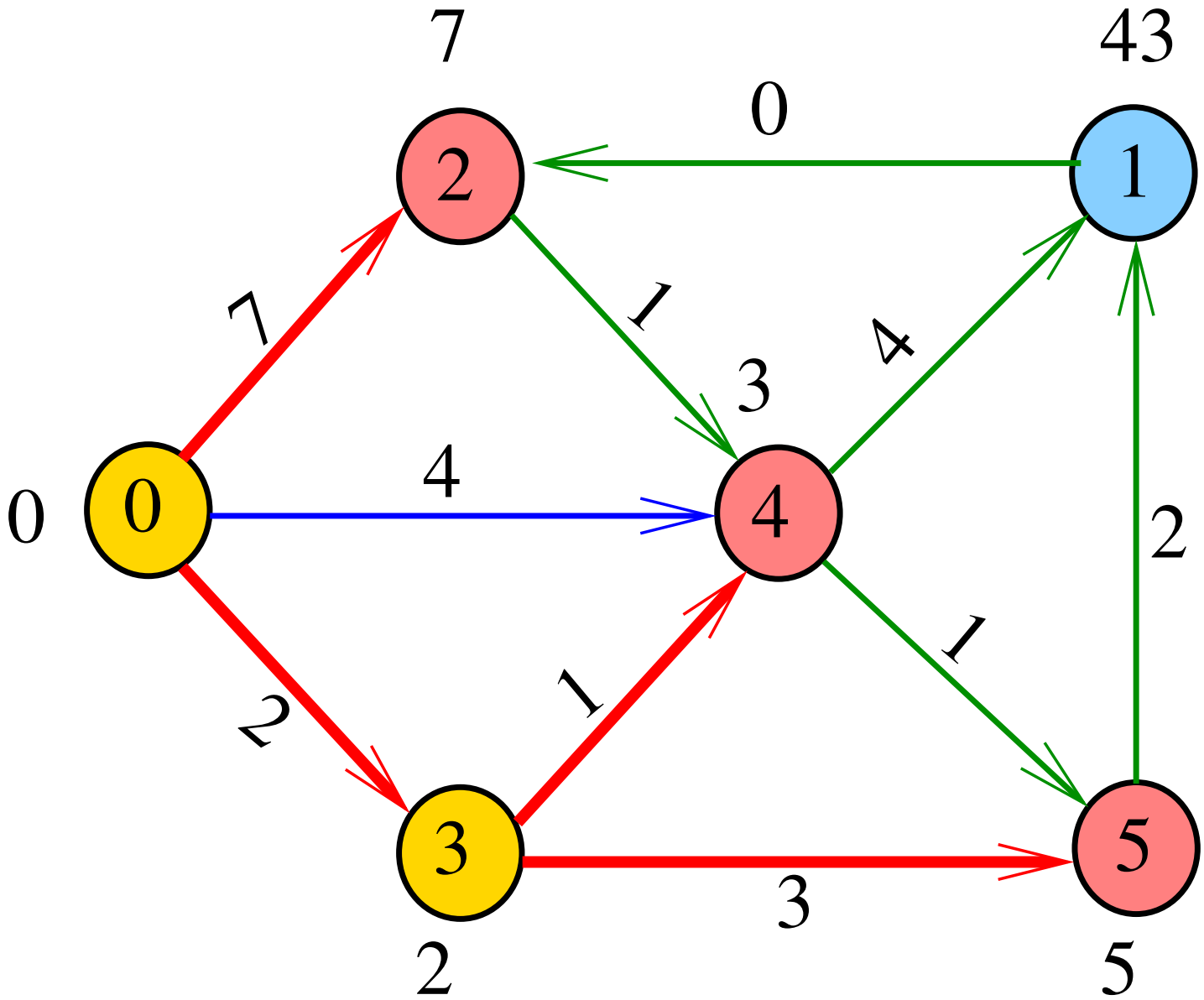
# Simulação



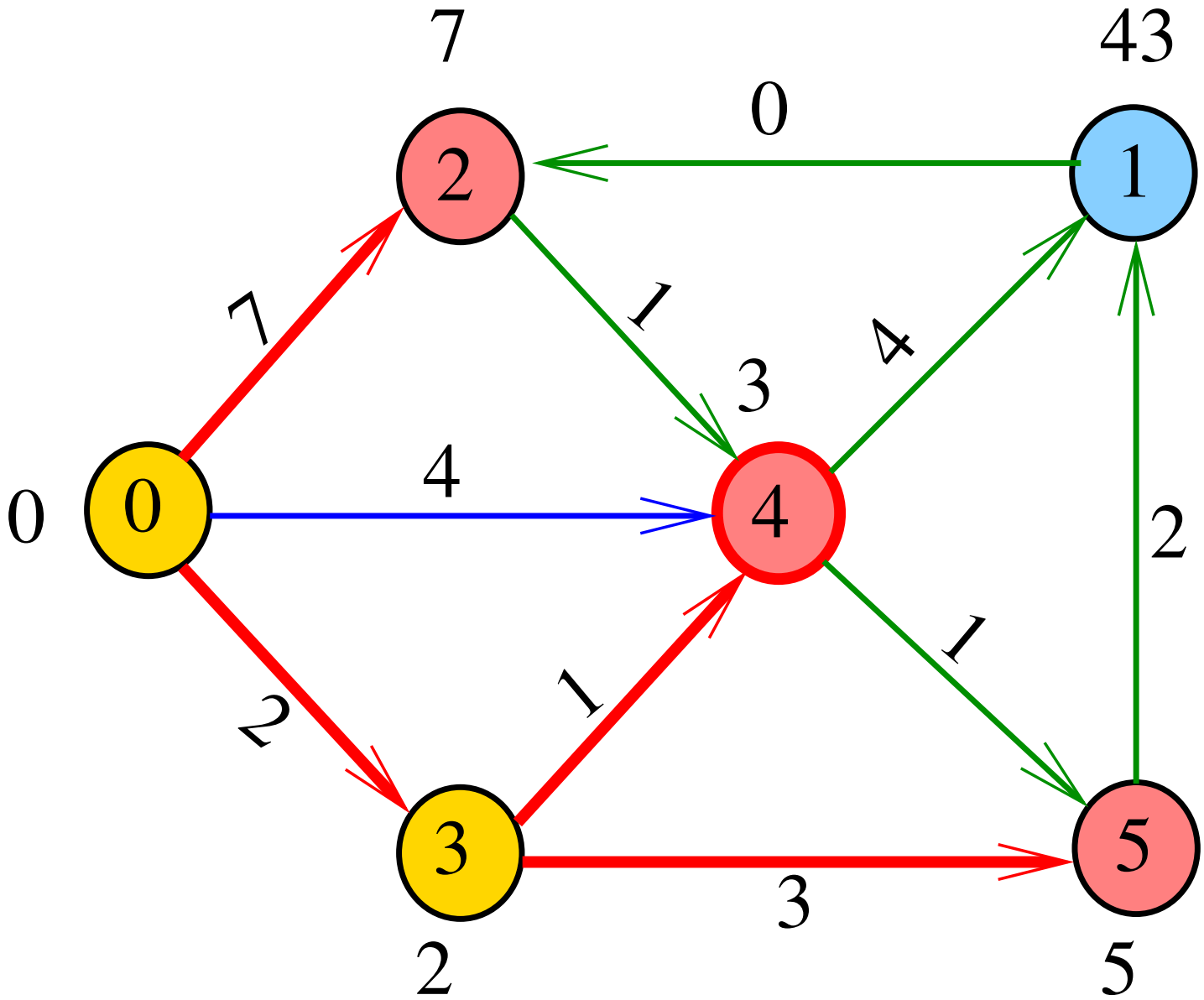
# Simulação



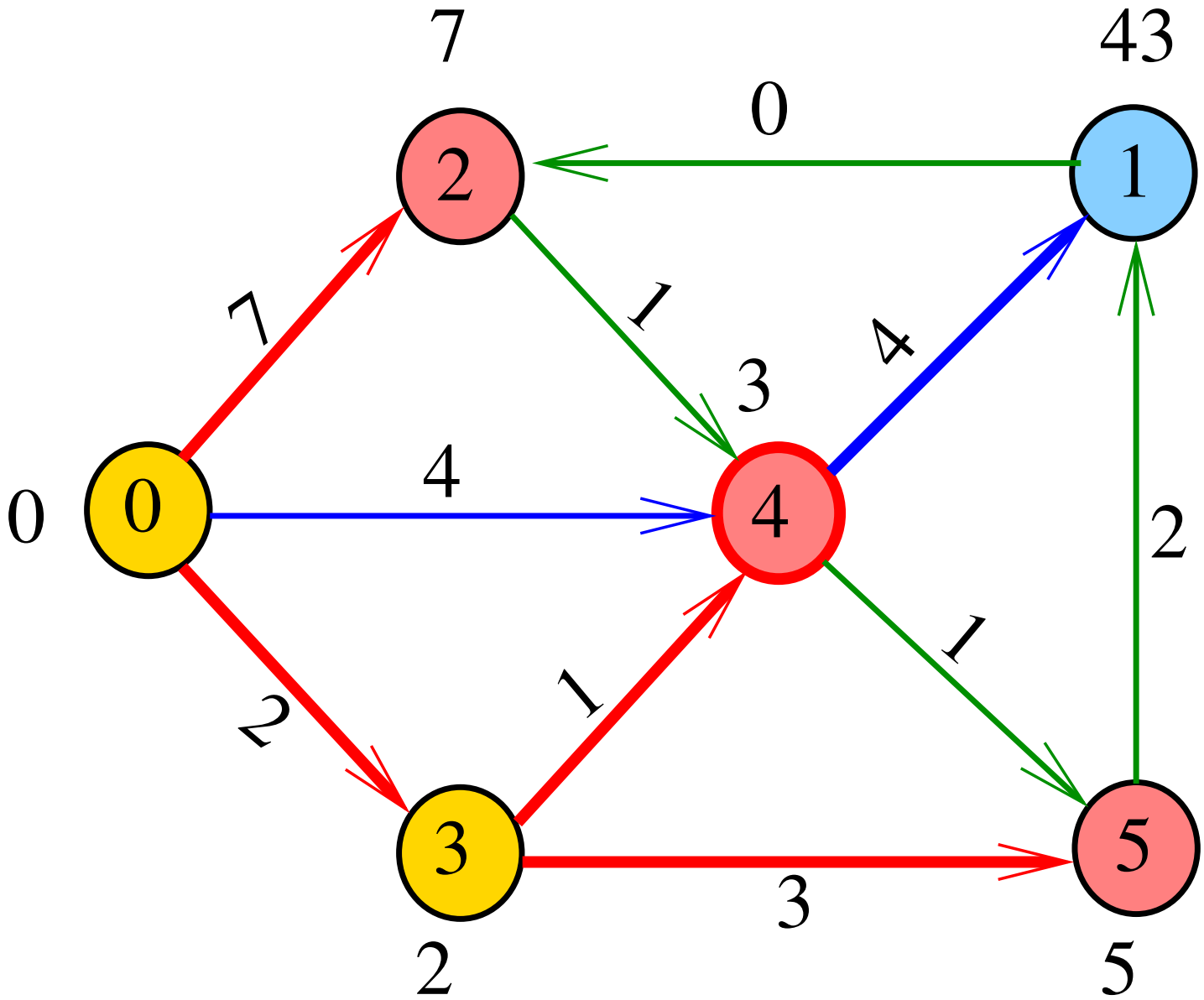
# Simulação



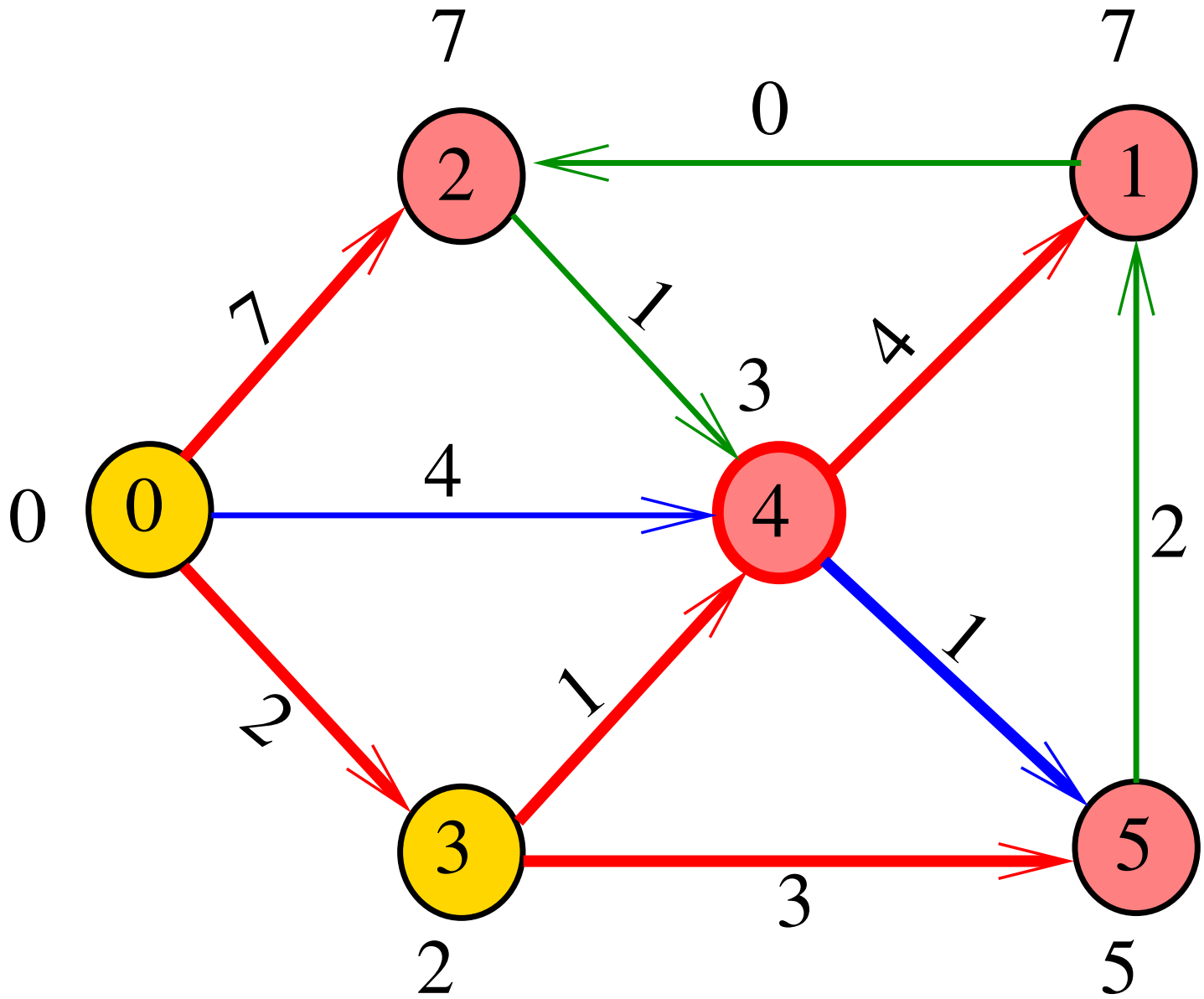
# Simulação



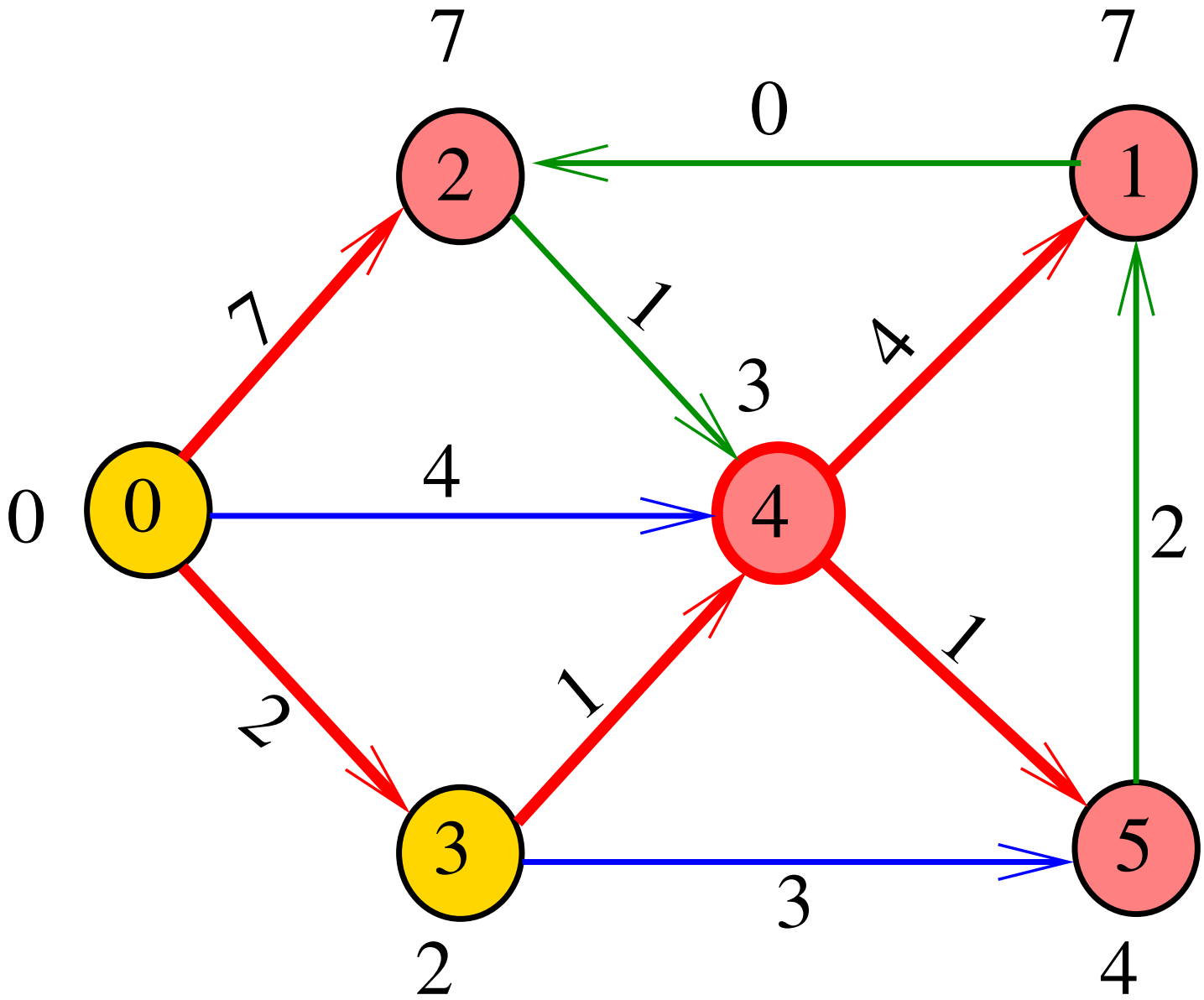
# Simulação



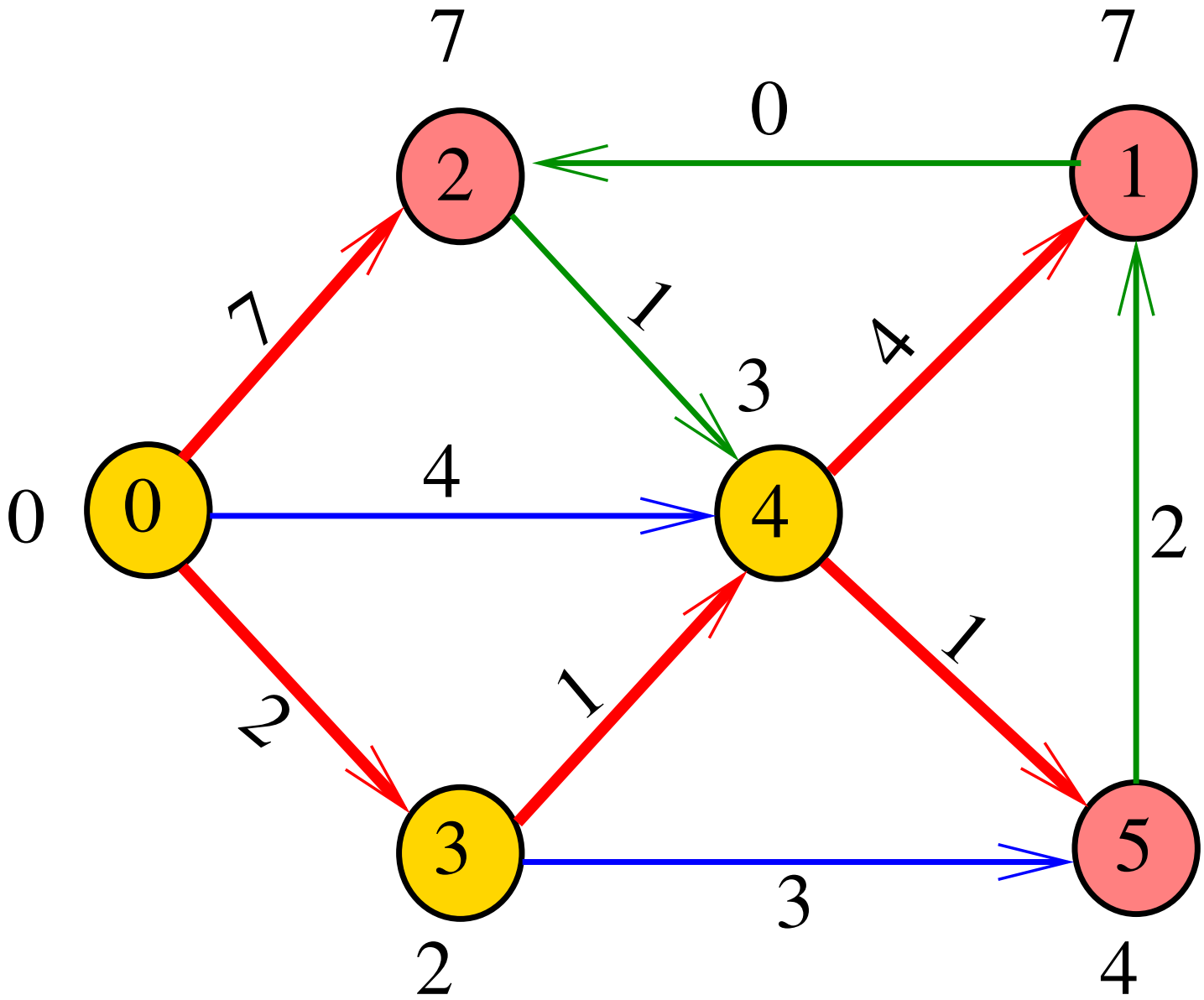
# Simulação



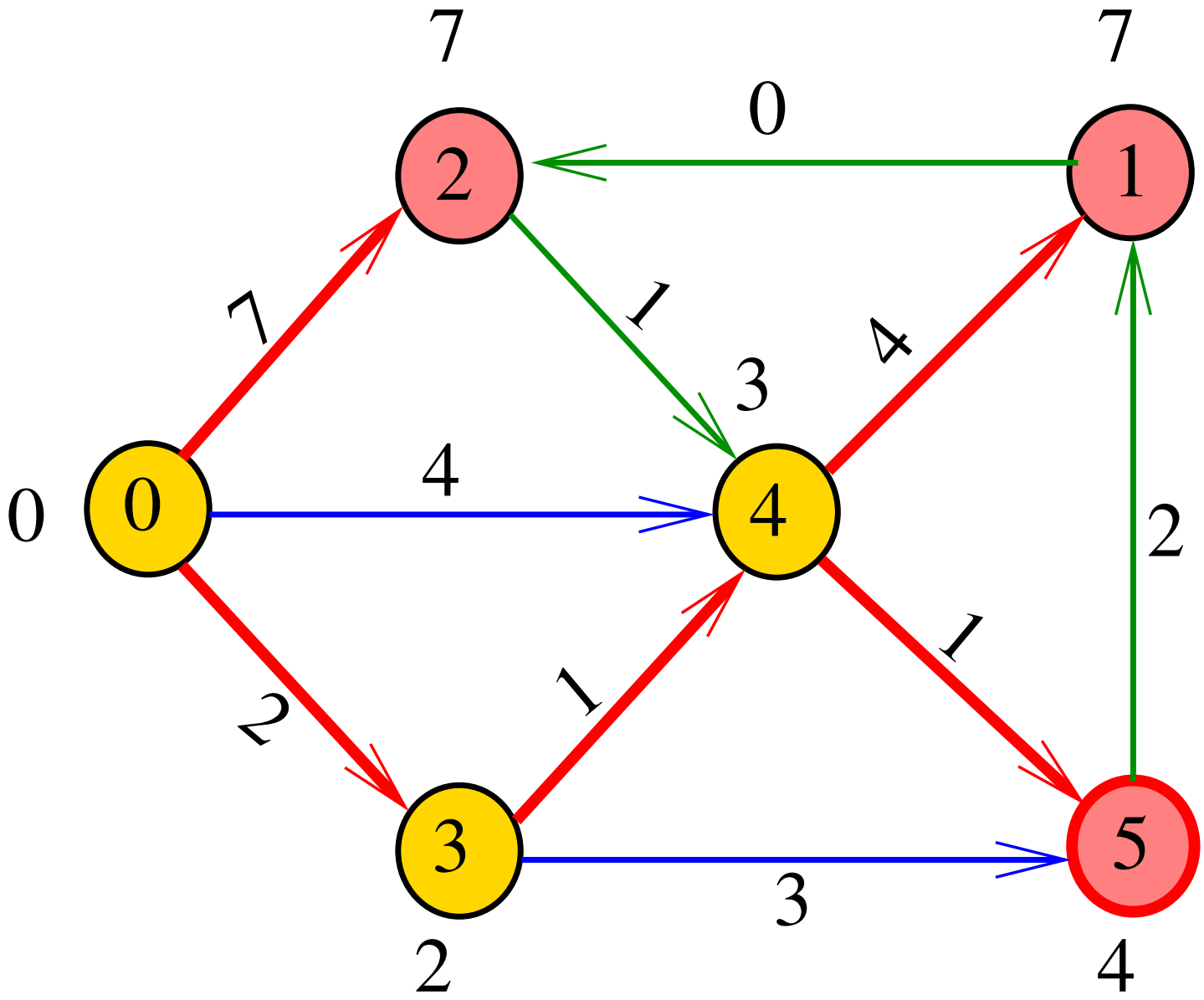
# Simulação



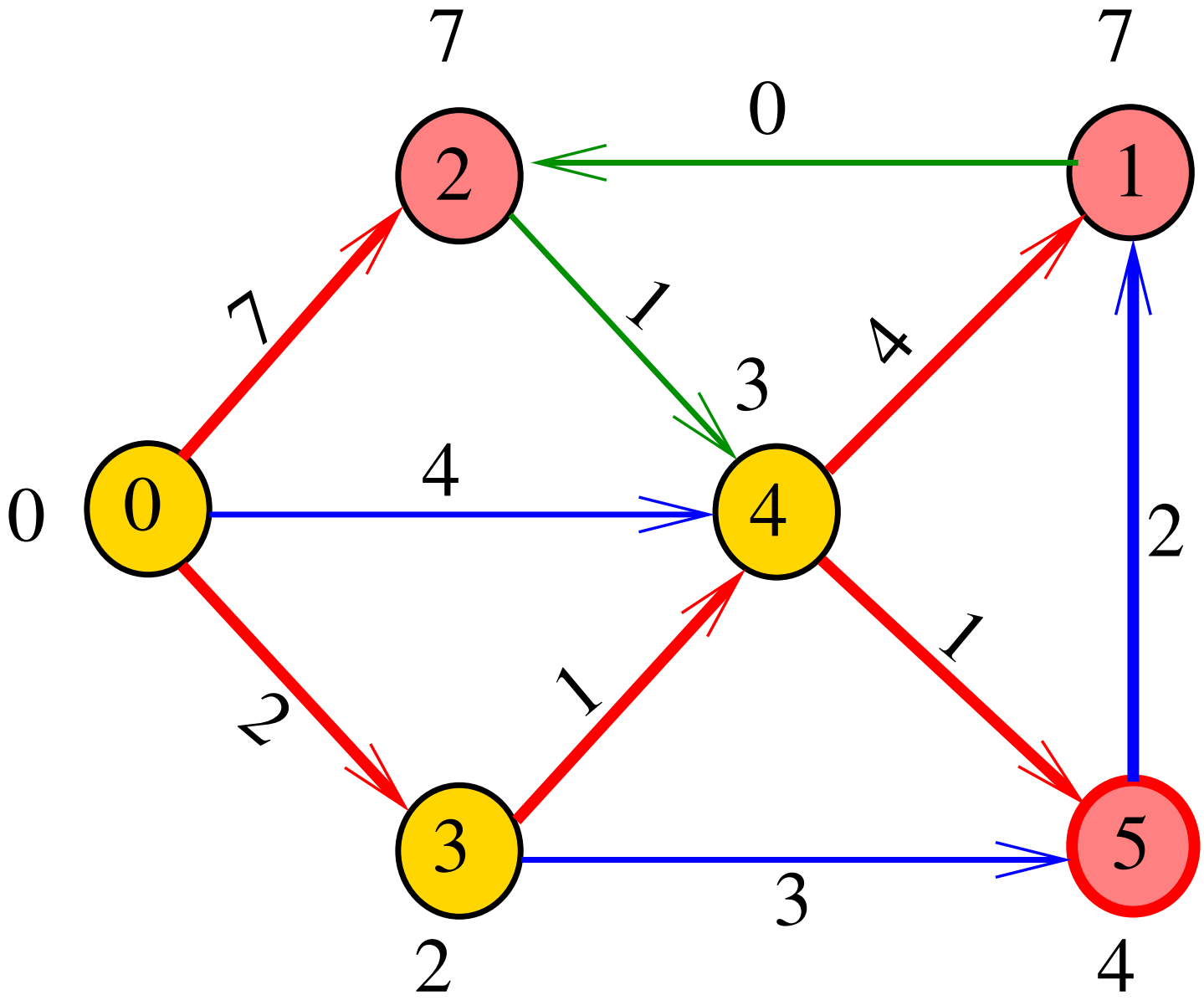
# Simulação



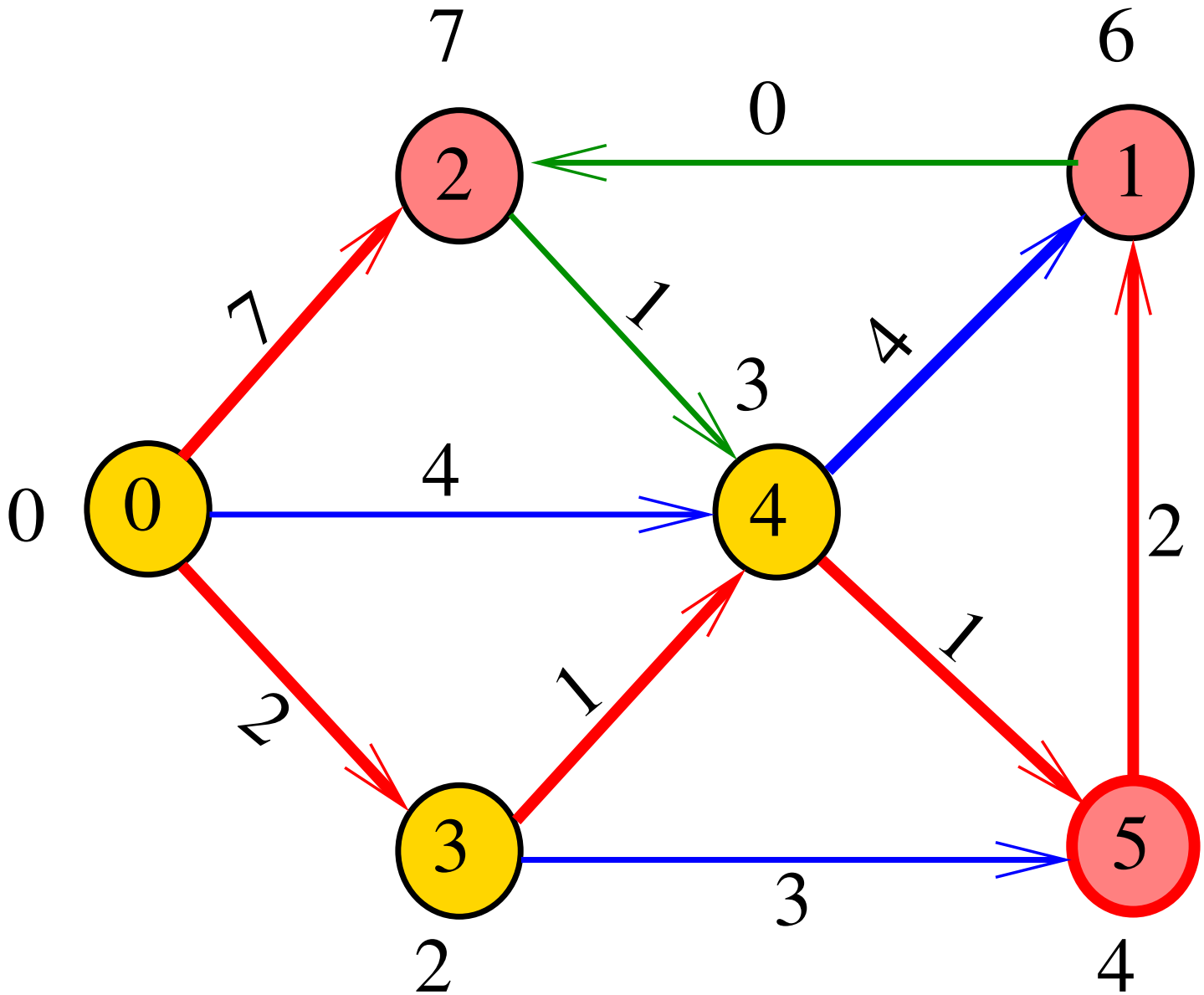
# Simulação



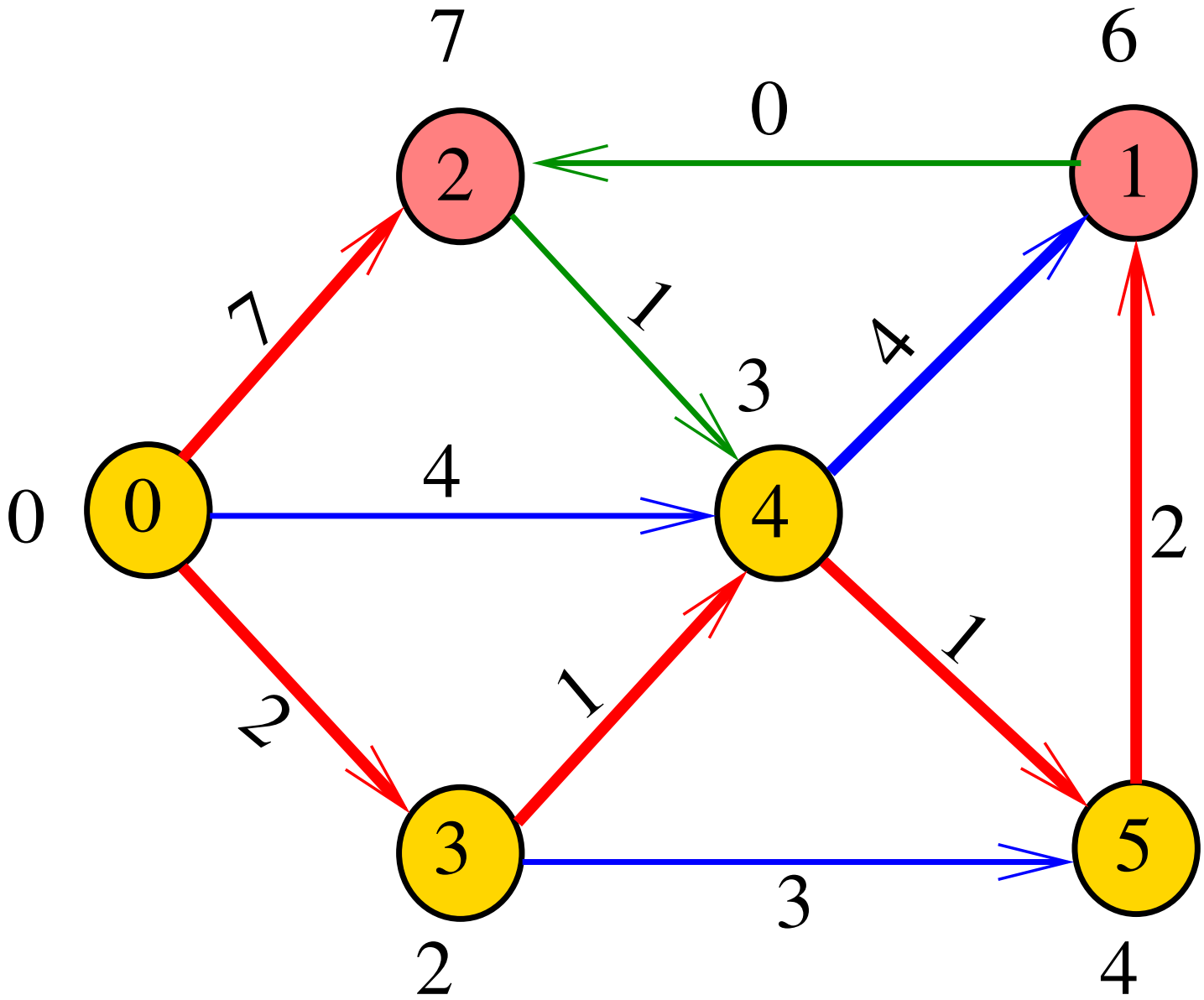
# Simulação



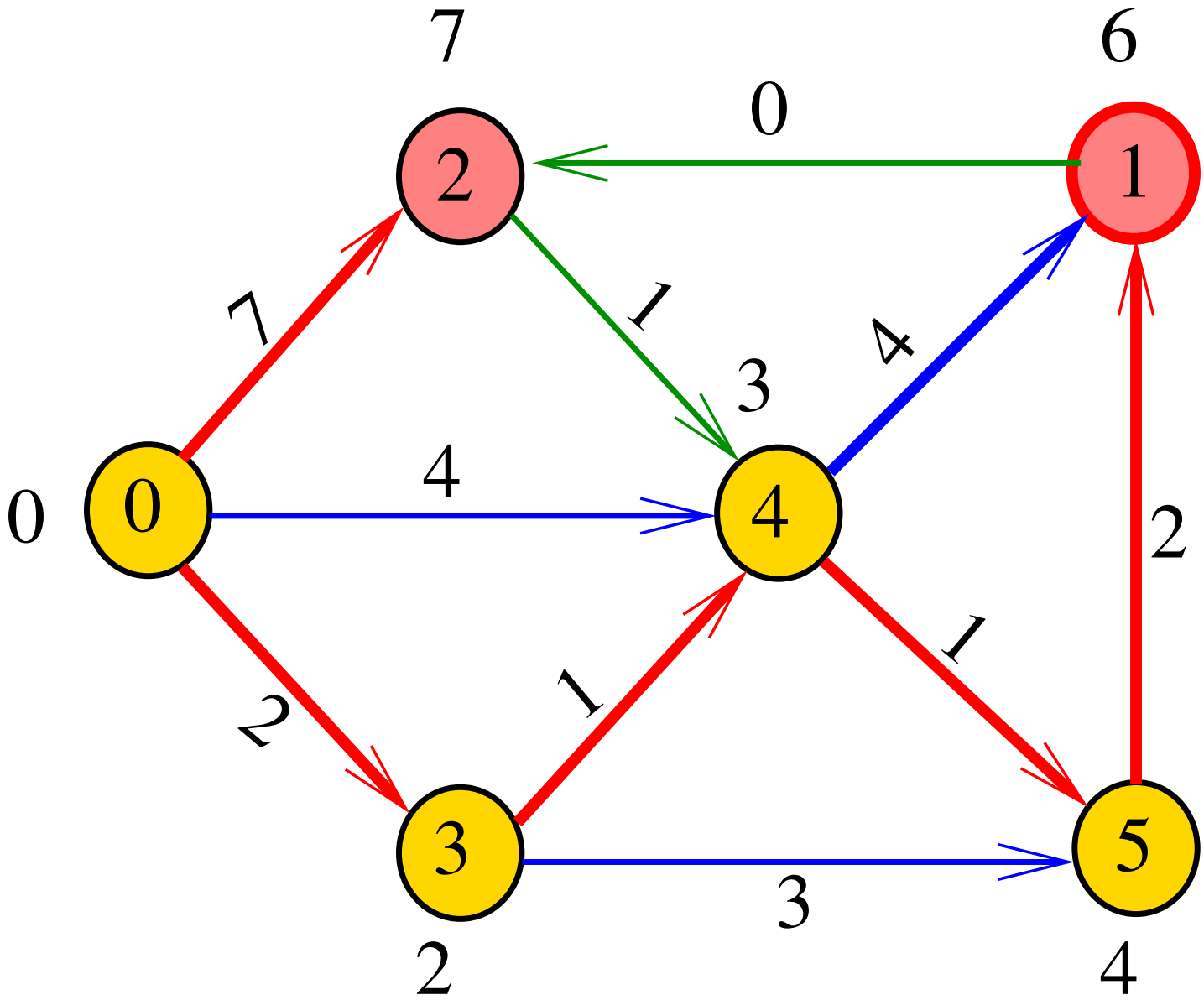
# Simulação



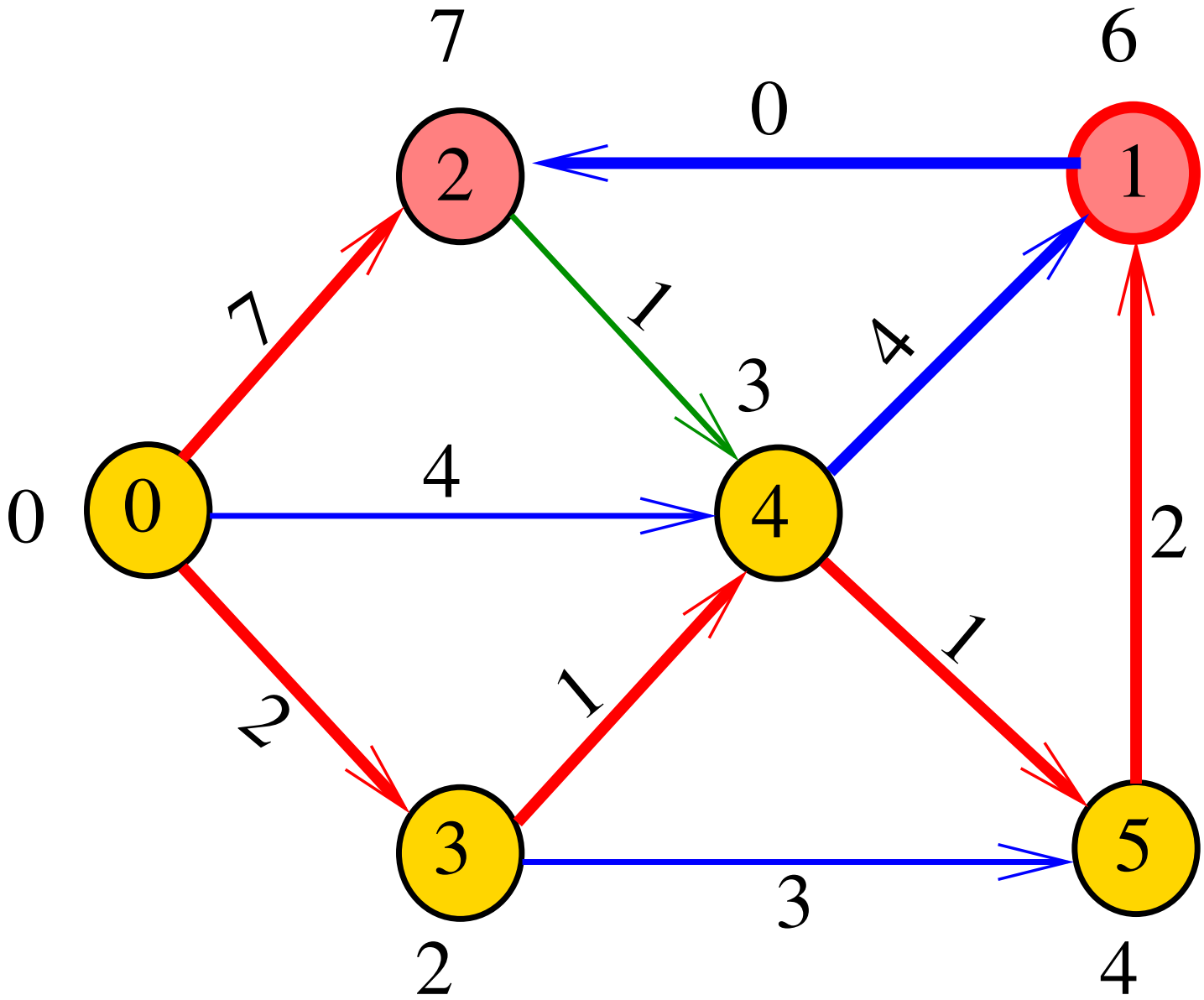
# Simulação



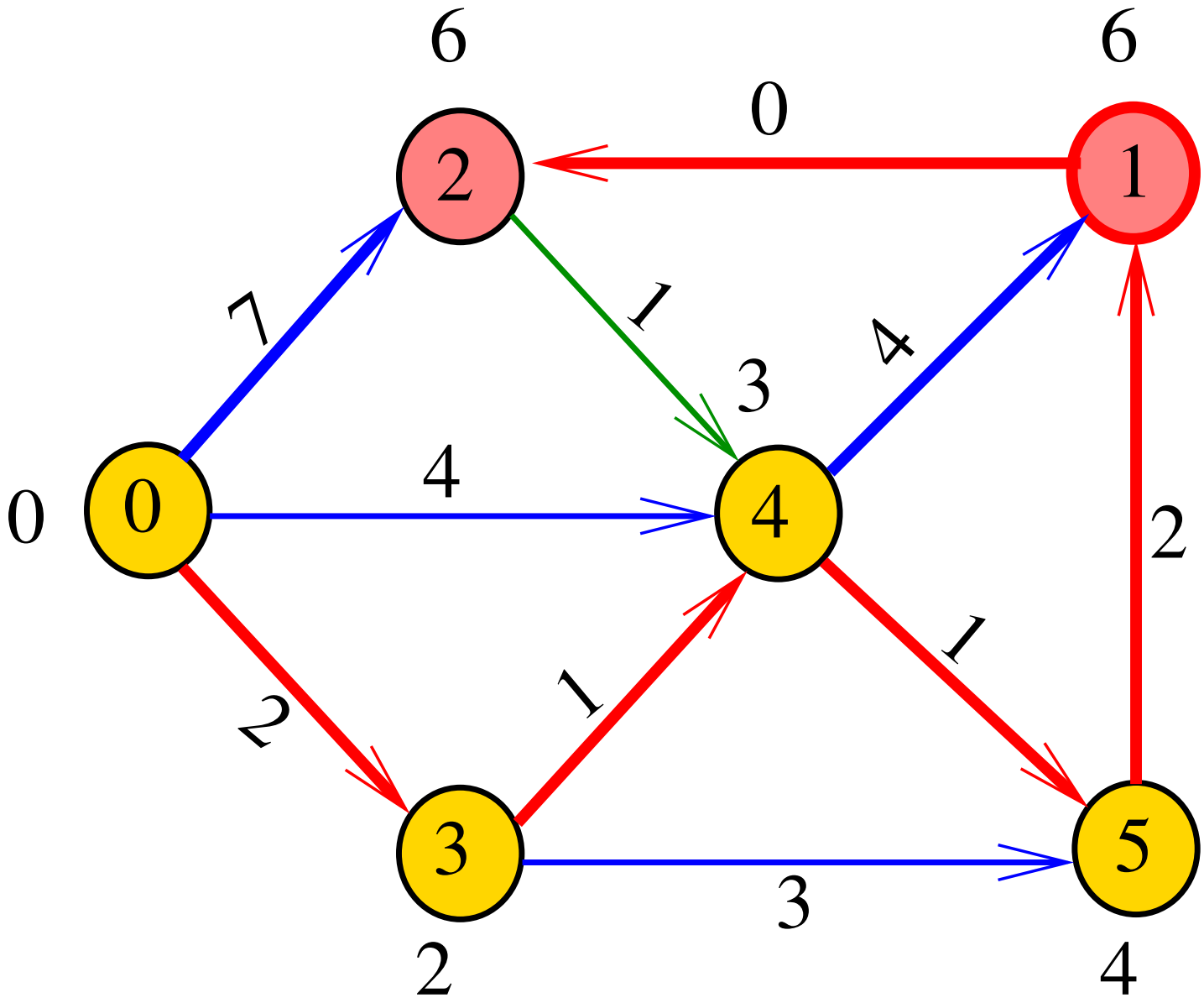
# Simulação



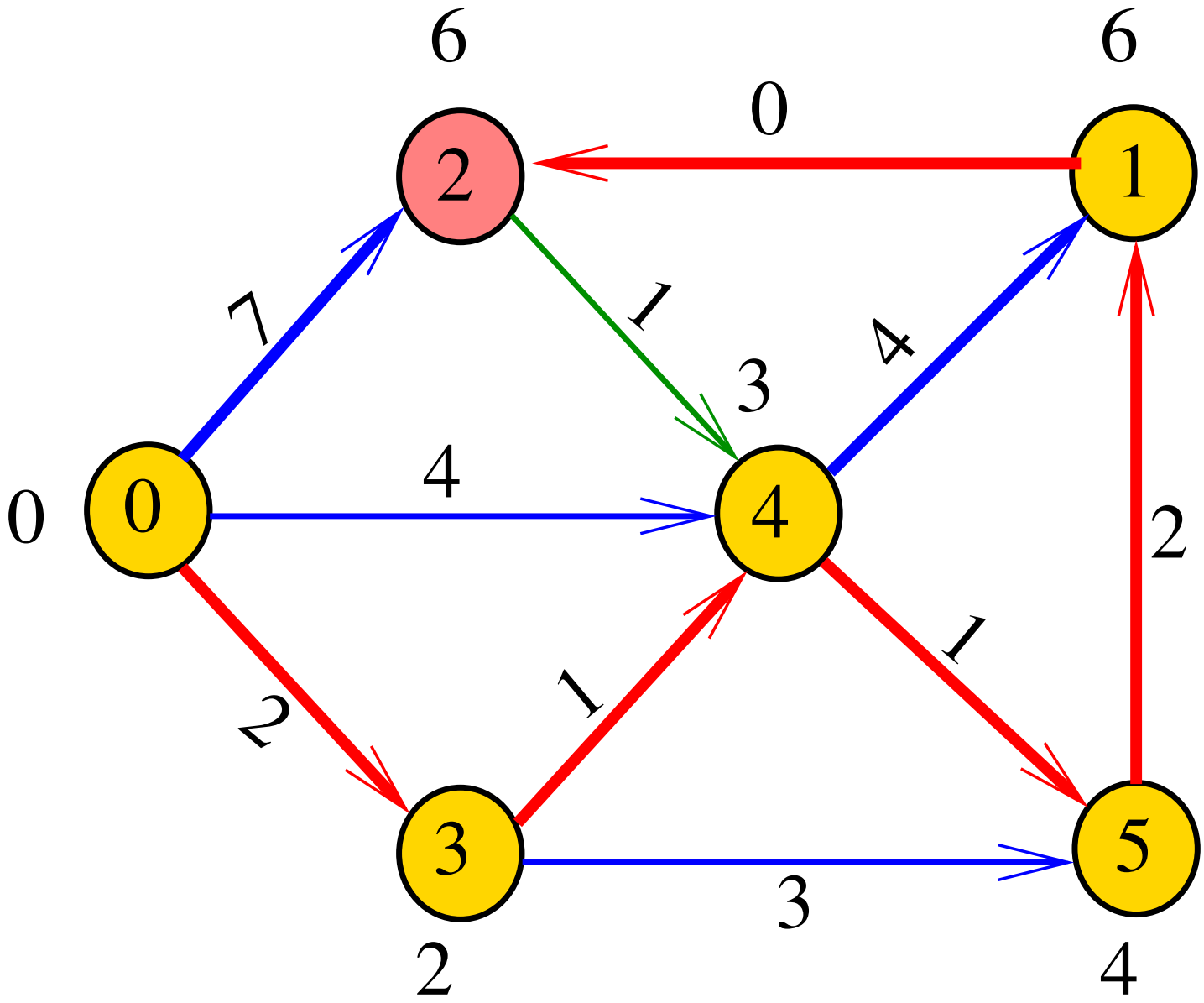
# Simulação



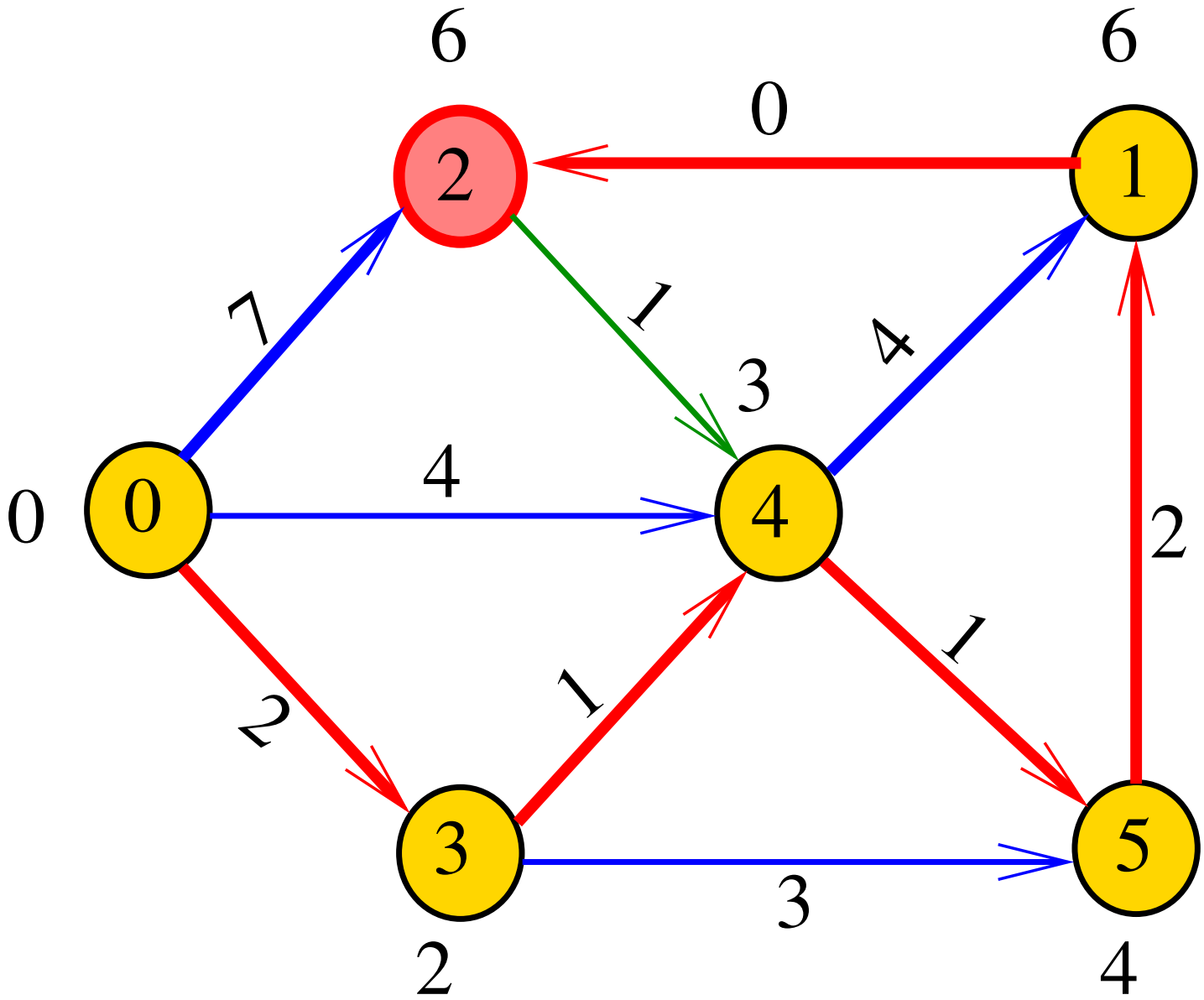
# Simulação



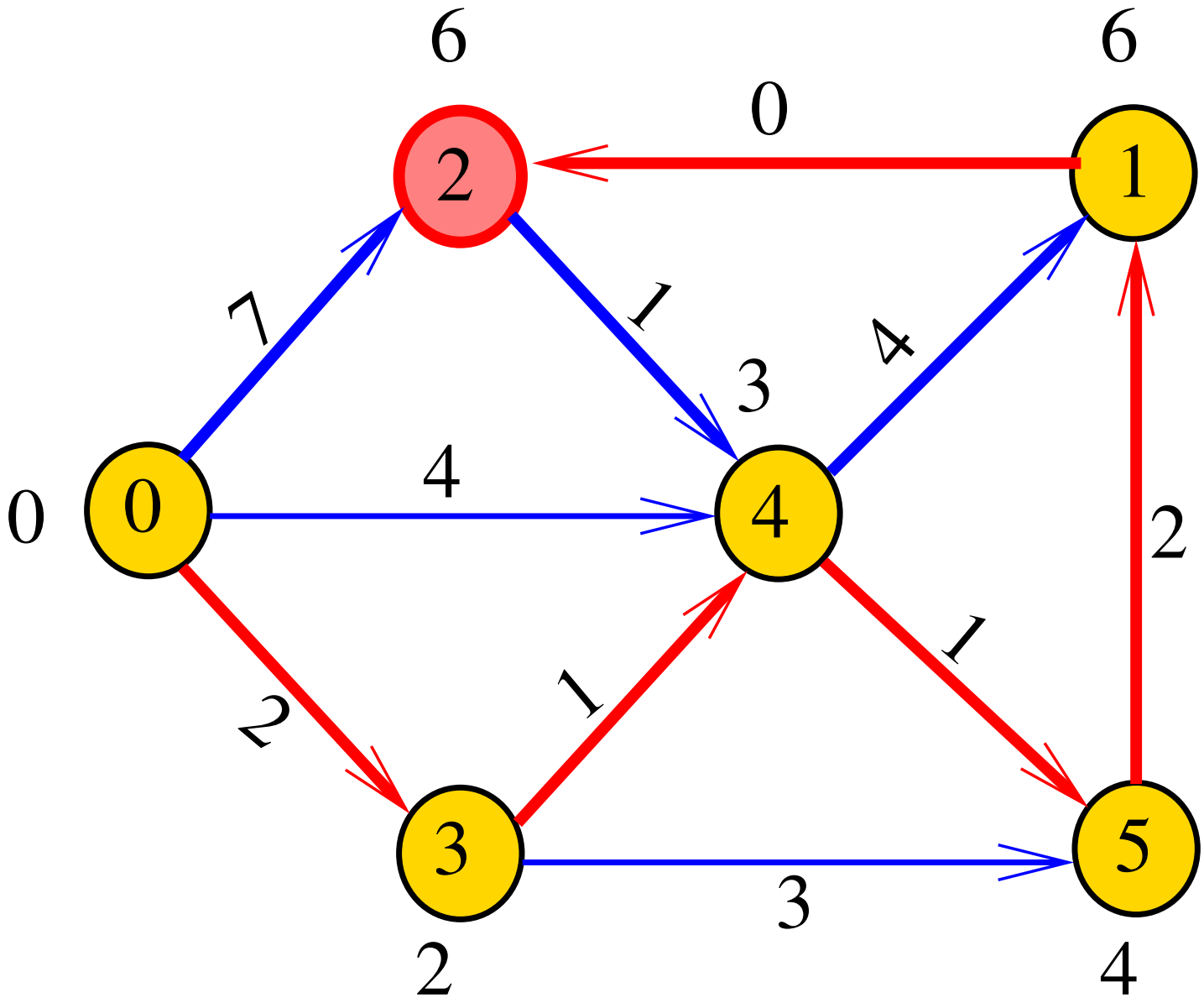
# Simulação



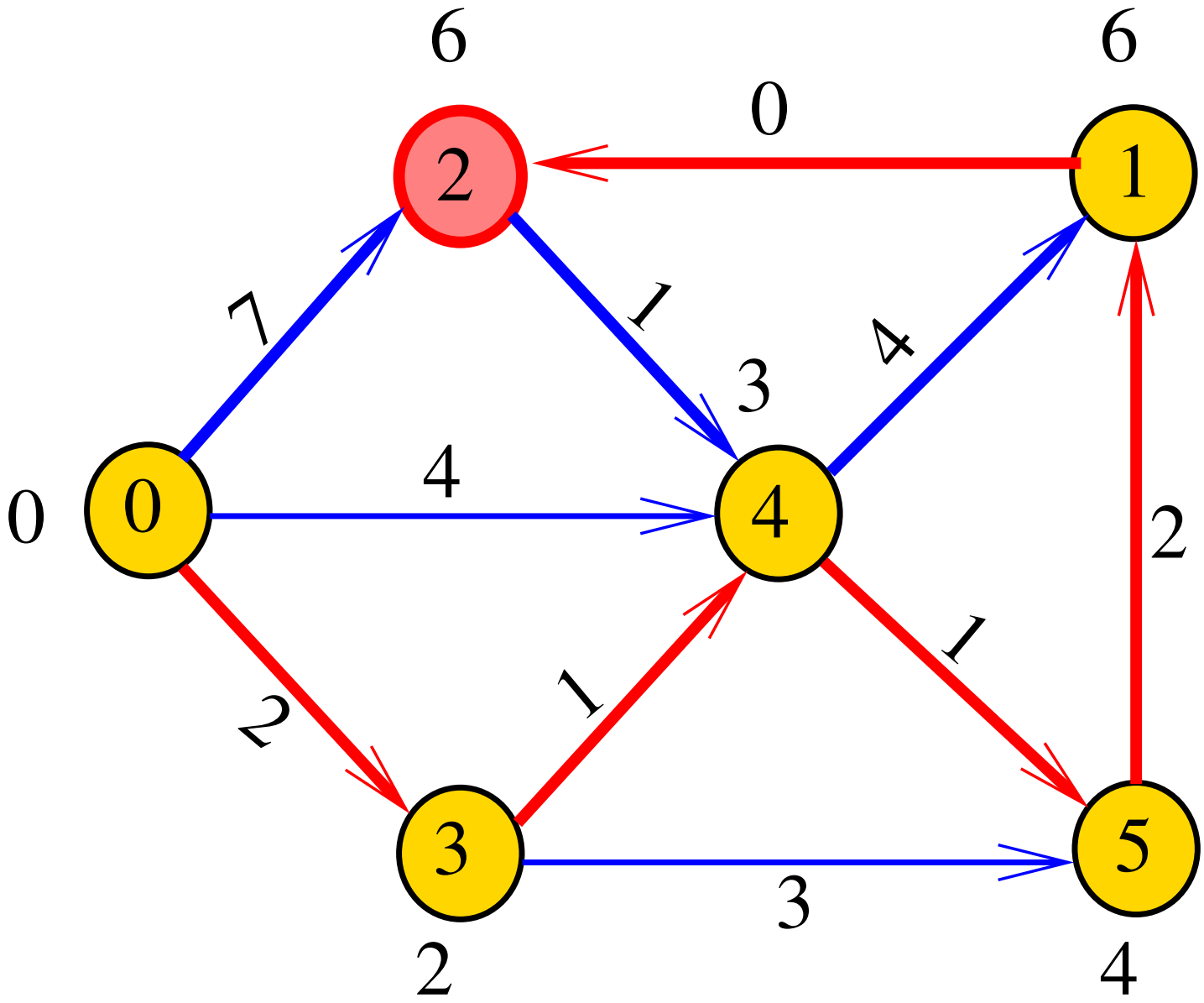
# Simulação



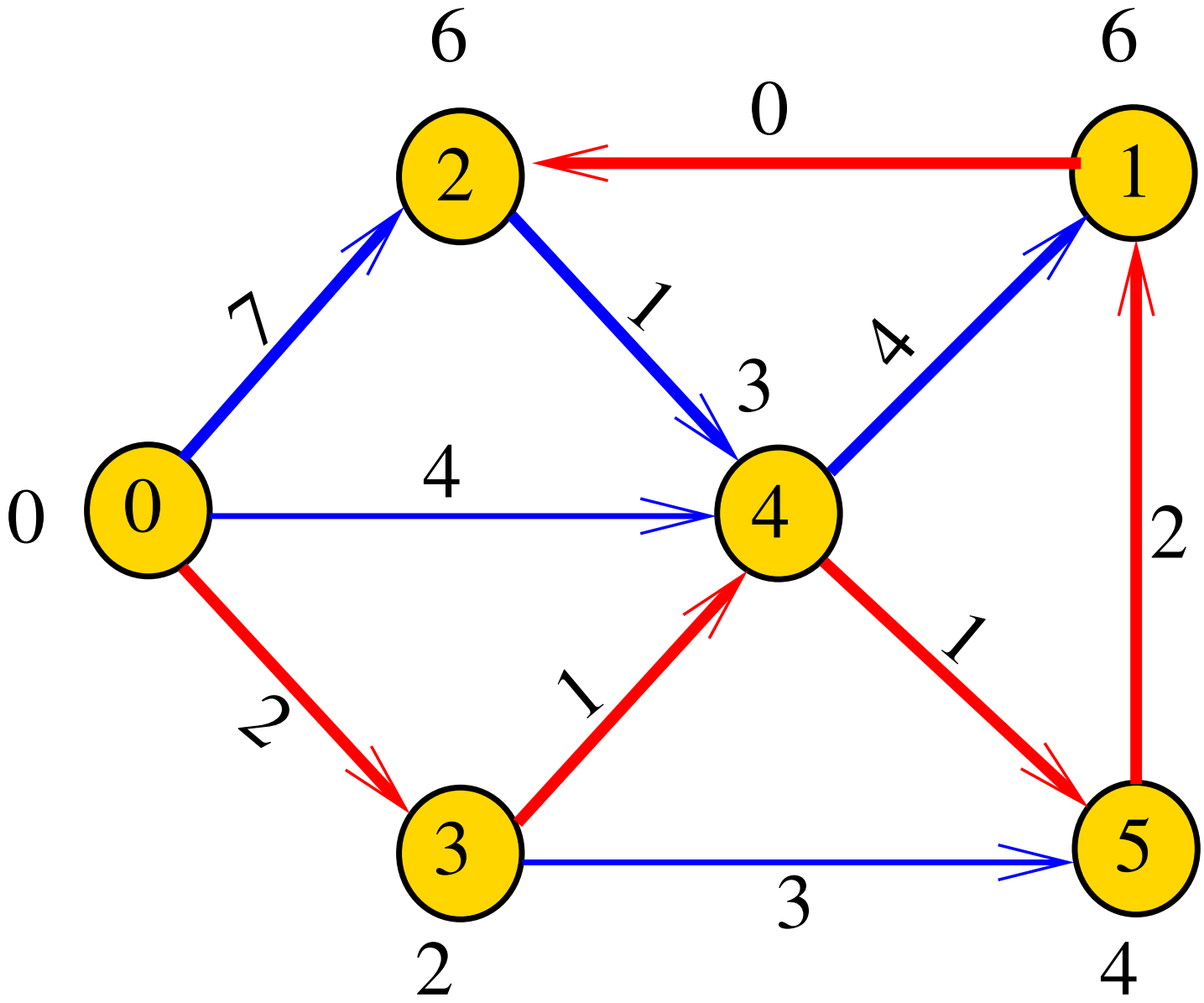
# Simulação



# 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 edgeTo[].

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

```
private double[] distTo;
private Arc[] edgeTo;
```

# Fila priorizadas

A classe `Dijkstra` usa uma fila priorizada

```
private IndexMinPQ<Double> pq;
```

A fila é manipulada pelos métodos:

- `IndexMinPQ<Double>()`: fila de vértices em que cada vértice `v` tem prioridade `distTo[v]`
- `isEmpty()`: a fila está vazia?
- `contains(v)`: `v` está na fila?
- `insert(v,valor)`: insere `v` com prior. `valor`
- `delMin()`: retorna um vértice de prioridade mínima.
- `decreaseKey(w,valor)`: reorganiza a fila depois que o valor de `distTo[w]` foi decrementado.

# Classe Dijkstra: esqueleto

```
public class Dijkstra {
   private static final double INFINITY;
   private final int s;
   private double[] distTo;
   private Arc[] edgeTo;
   public Dijkstra(EWDigraph G, int s) {}
   private void dijkstra(EWDigraph G,
                         int s) {}
   public boolean hasPath(int v) {}
   public boolean distTo(int v) {}
   public Iterable<Arc> pathTo(int v) {}
}
```

# Dijkstra

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

```
public Dijkstra(EWDigraph G, int s) {
   INFINITY = Double.POSITIVE_INFINITY;
   distTo = new double[G.V()];
   edgeTo = new Arc[G.V()];
   this.s = s;
   for (int v = 0; v < G.V(); v++)
       distTo[v] = INFINITY;
   dijkstra(G, s);
}
```

# dijkstra(): inicializações

```
private void dijkstra(EWDigraph G, int s)
{
   IndexMinPQ<Double> pq =
            new IndexMinPQ<Double>(G.V());
   distTo[s] = 0;
   pq.insert(s, distTo[s]);

   // aqui vem a iteração do próximo slide
```

# dijkstra(): iteração

```
while (!pq.isEmpty()) {
    int v = pq.delMin();
    for (Arc e :  G.adj(v)) {
        int w = e.to();
        double d = distTo[v]+e.weight();
        if (distTo[w] > d)
            edgeTo[w] = e;
            distTo[w] = d;
            if (pq.contains(w))
                pq.decreaseKey(w, d);
            else pq.insert(w, d);
        }
    }
}
```

# Dijkstra

Há um caminho de s a v?

```
// Método copiado de BFSpaths.
public boolean hasPath(int v) {
   return distTo[v] < INFINITY;
}

// retorna o comprimento de um
// caminho mínimo de s a t
public int distTo(int v) {
   return distTo[v];
}
```

# Dijkstra

Retorna um caminho de s a v ou null se um tal caminho não existe.

```
// Método adaptado de DFSpaths.
public Iterable<Arc> pathTo(int v) {
   if (!hasPath(v)) return null;
   Stack<Arc> path = new Stack<Arc>();
   for (Arc e = edgeTo[v]; e != null;
                  e = edgeTo[e.from()])
      path.push(e);
   return path;
}
```

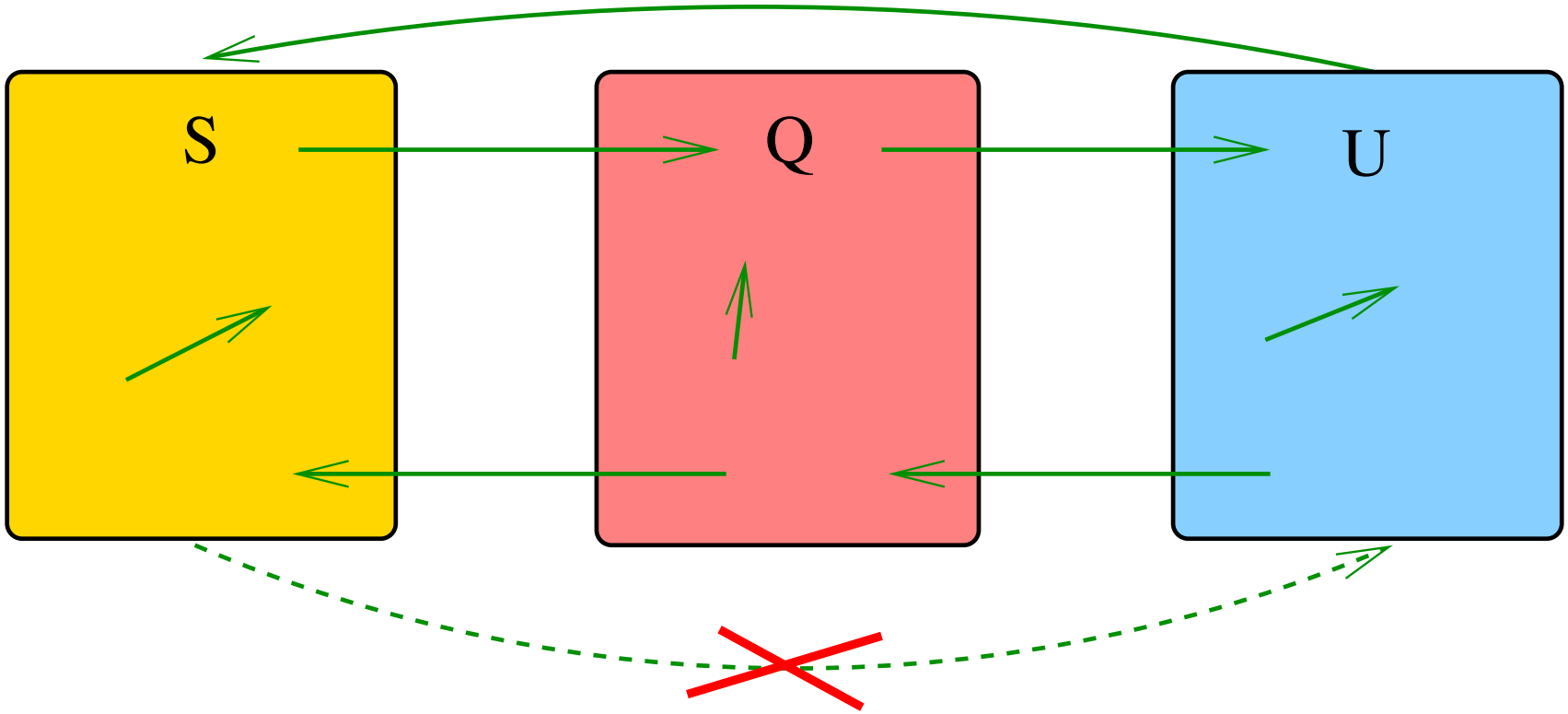
# Relações invariantes

S = vértices examinados

Q = vértices visitados = vértices na fila

U = vértices ainda não visitados

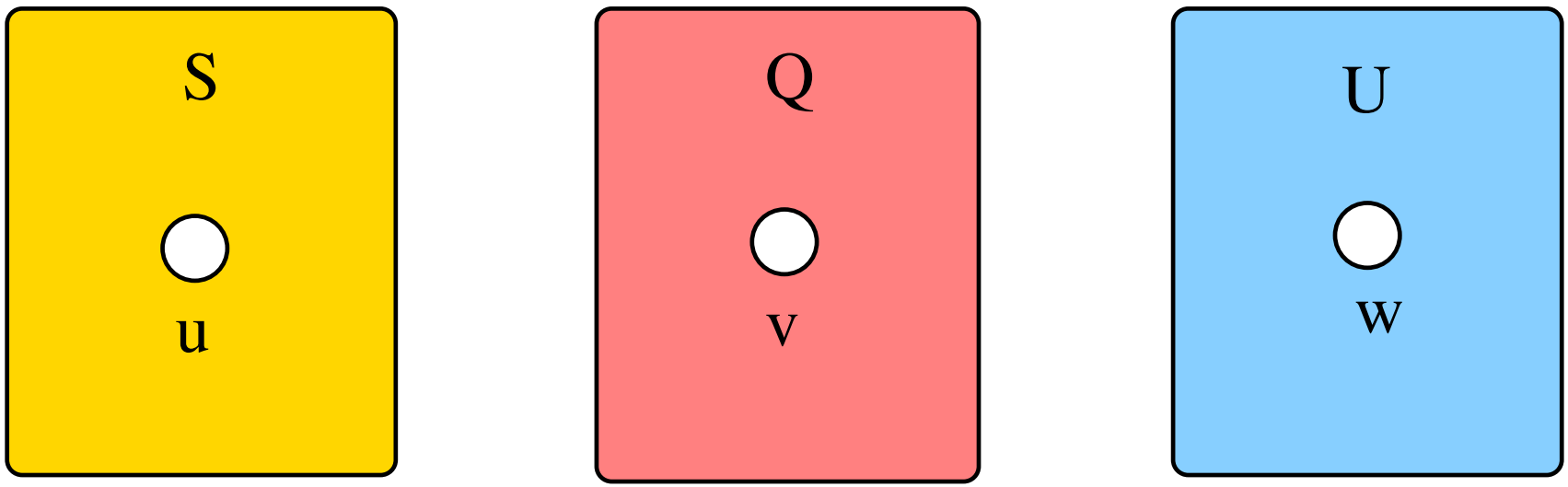
(i0) não existe arco v-w com v em S e w em U



# Relações invariantes

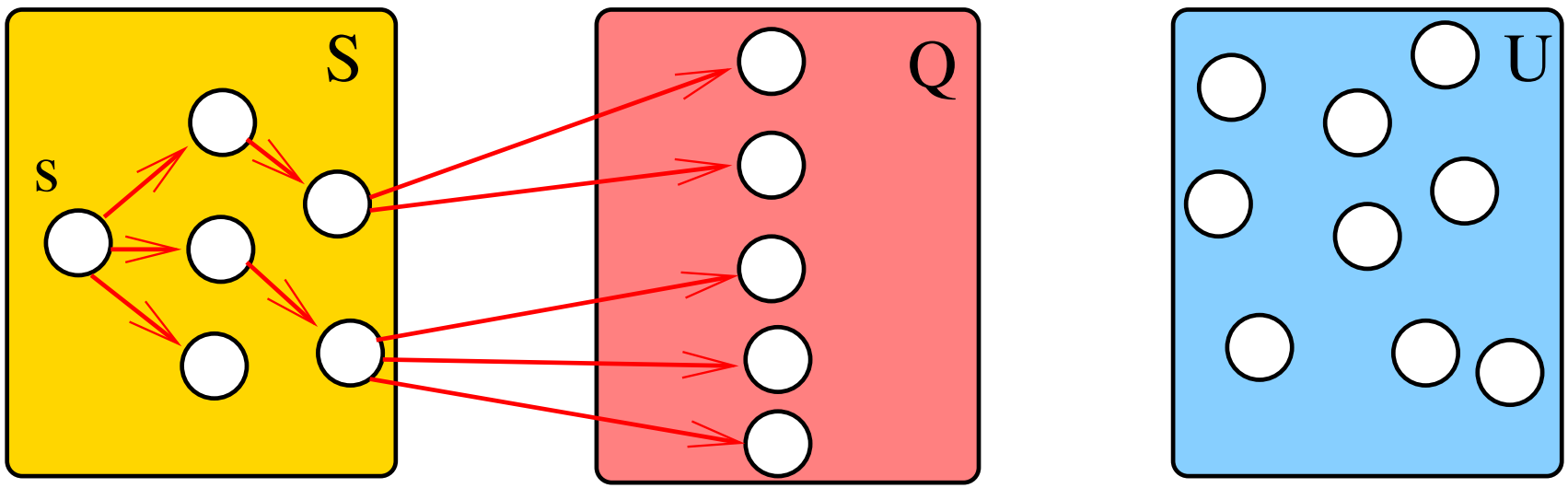
(i1) para cada `u` em `S`, `v` em `Q` e `w` em `U`

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



# Relações invariantes

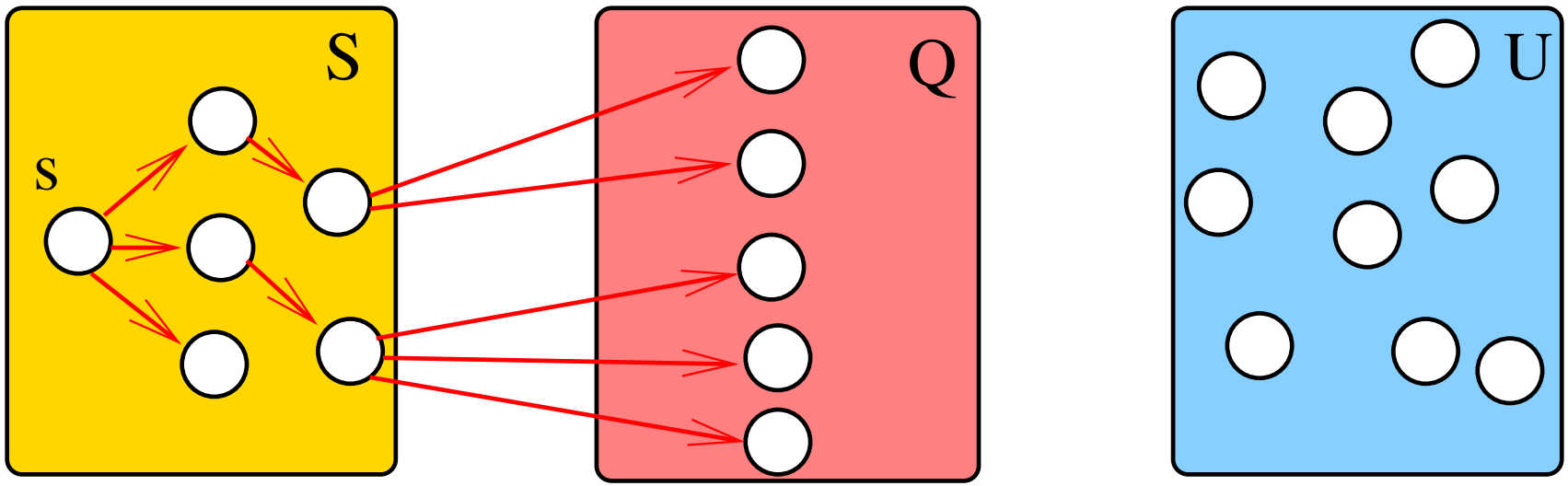
(i2) O vetor `edgeTo` restrito aos vértices de `S` e `Q` determina um árborescência com raiz `s`



# Relações invariantes

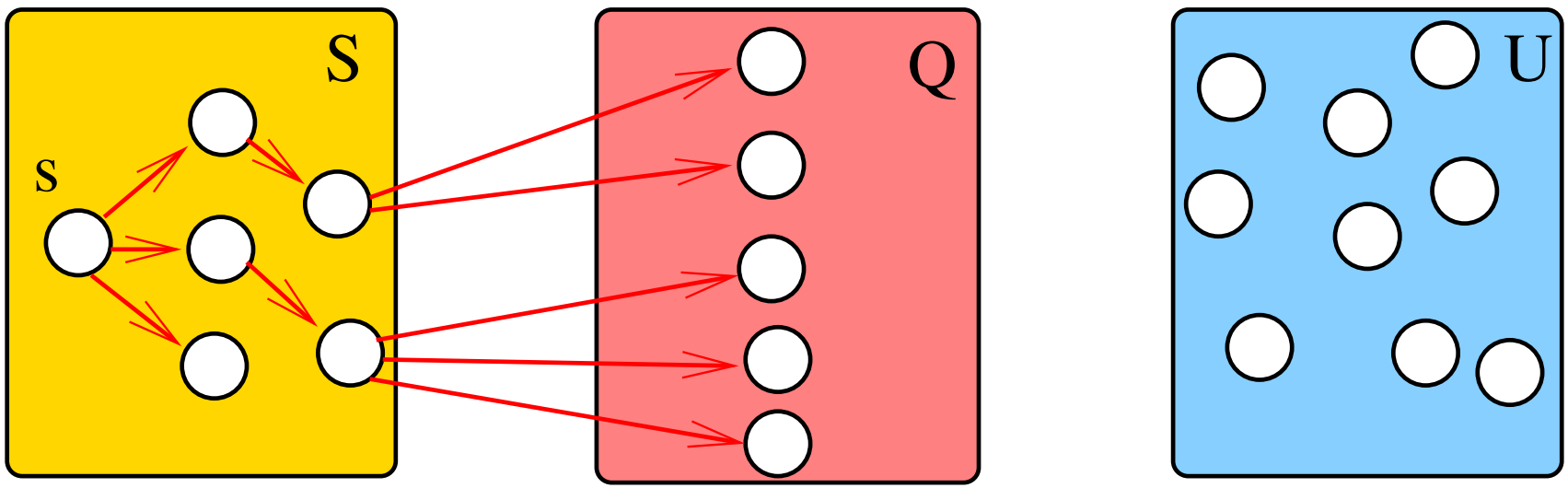
(i3) Para arco `v-w` na arborescência vale que

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

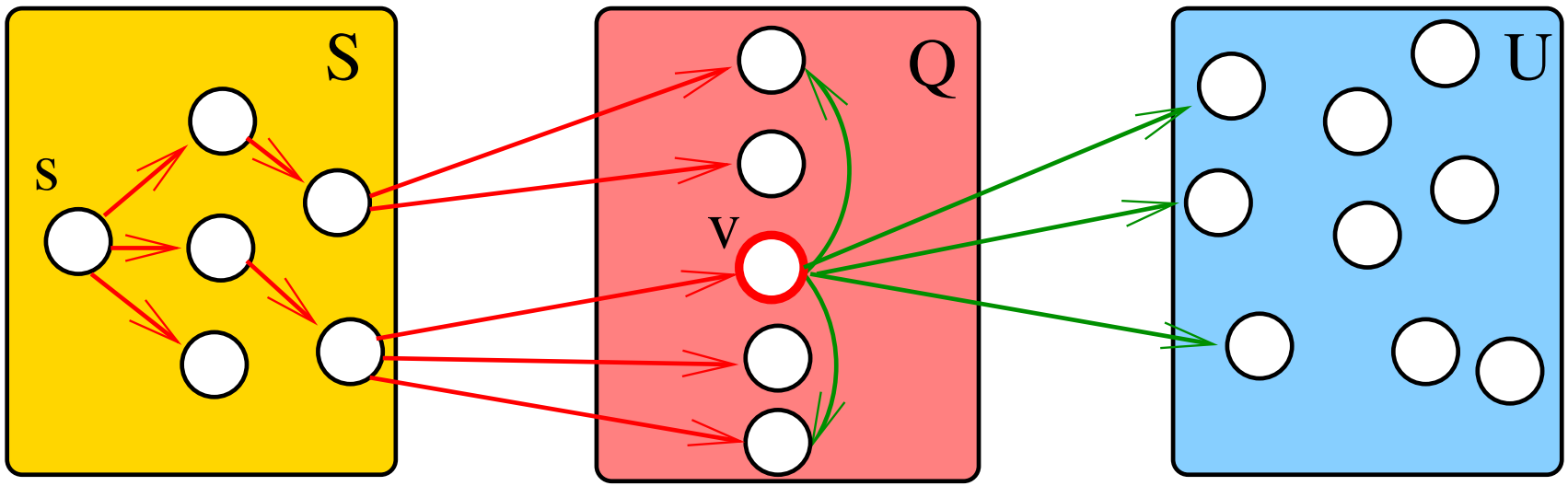


# Relações invariantes

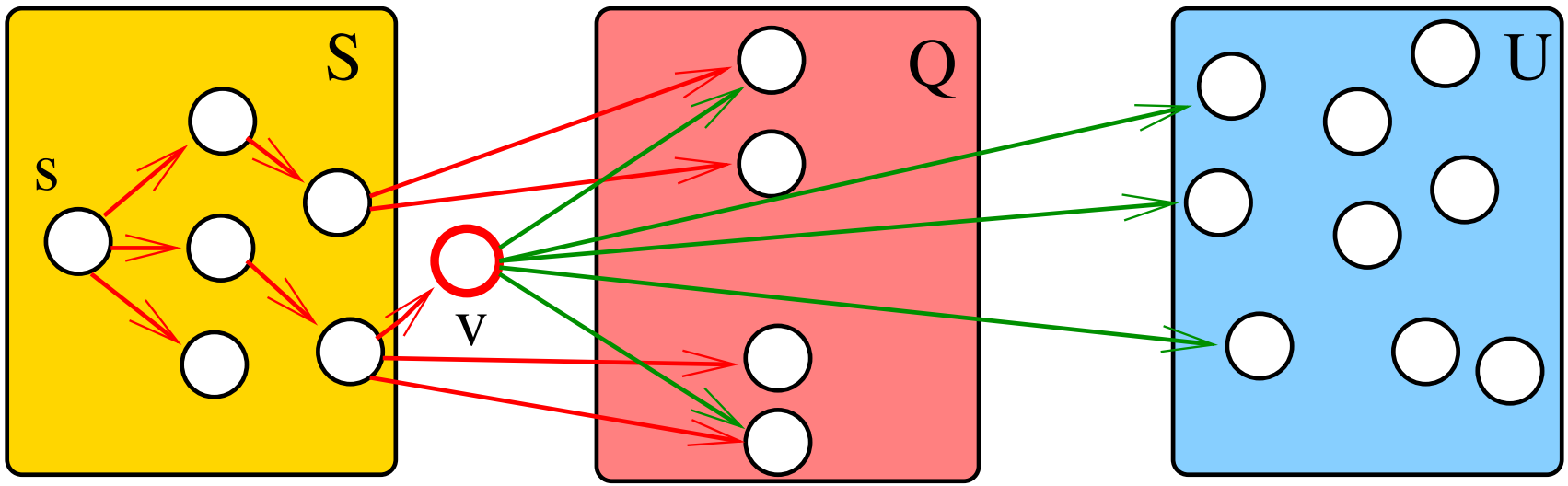
(i3) Para cada vértice `v` em `S` vale que `distTo`[`v`] é o custo de um caminho mínimo de `s` a `v`.



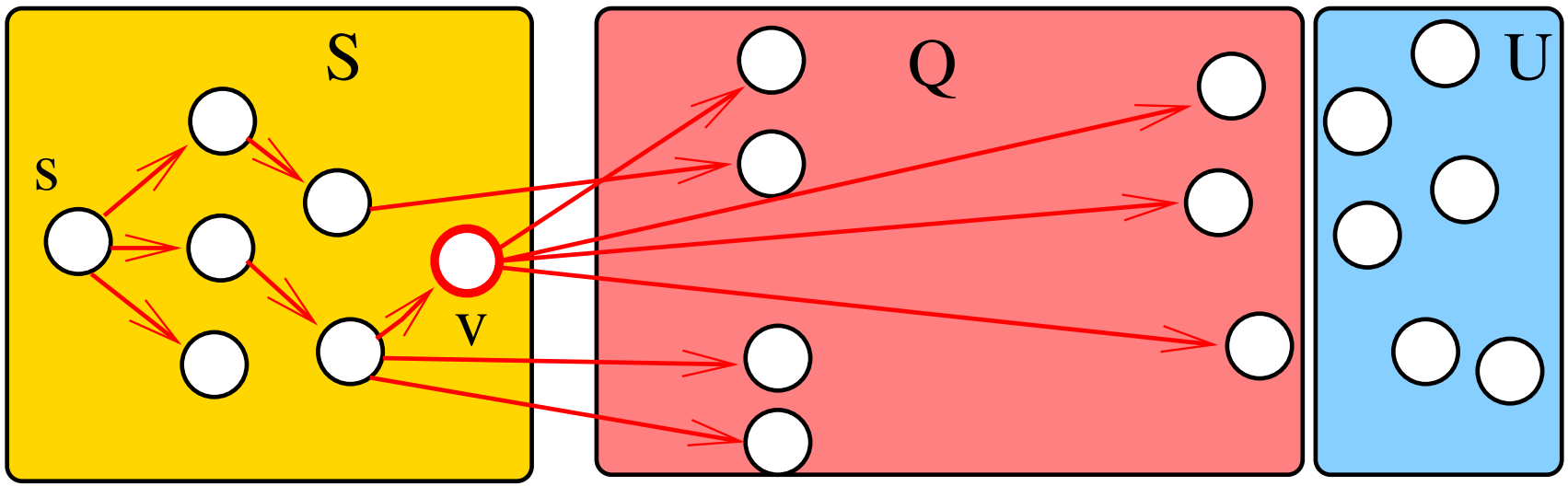
# Iteração



# 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 `IndexMinPQ<Double>`,
$\leq V$ 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

| | |
|---|---|
| `IndexMinPQ` | $\Theta(V)$ |
| `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 **MAC0328** Algoritmos em Grafos.

# Consumo de tempo Min-Heap

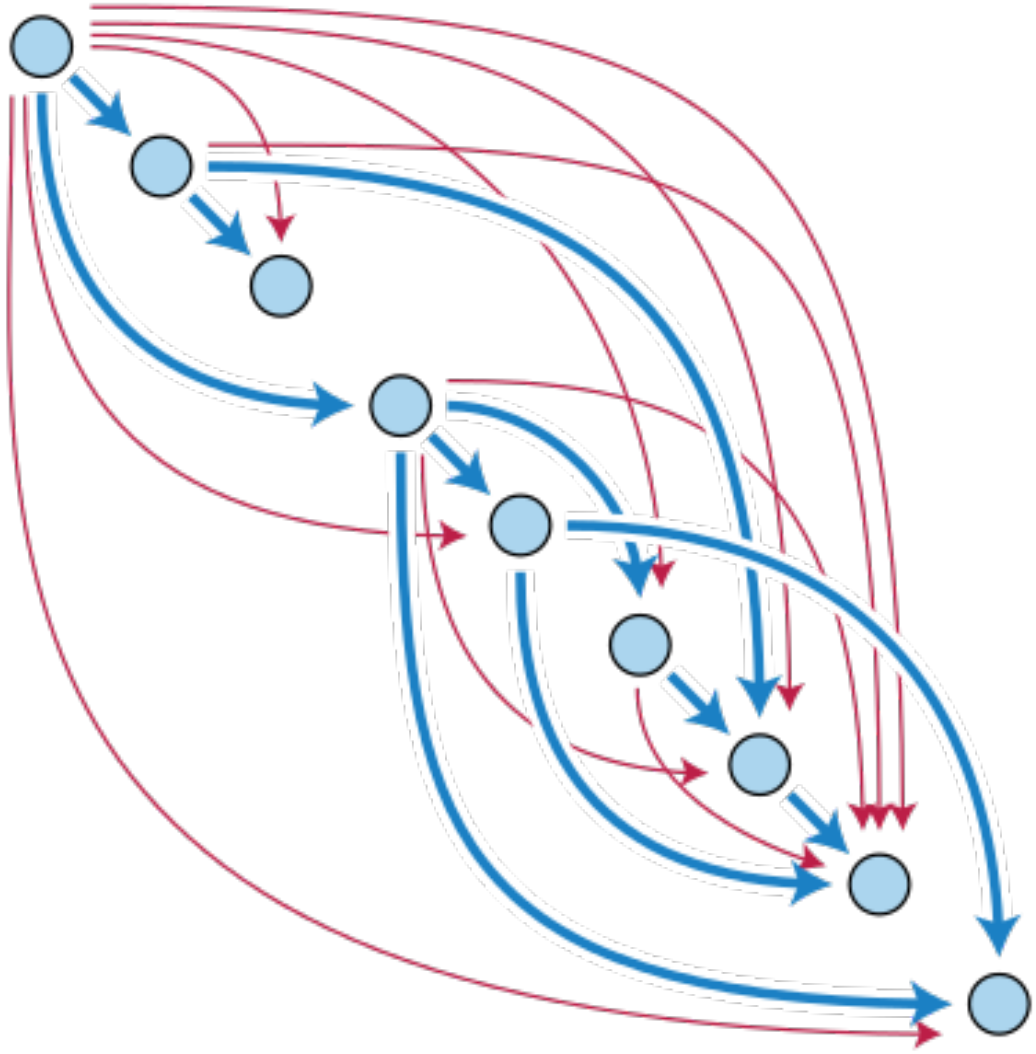
| | heap | $d$-heap | fibonacci heap |
|---|---|---|---|
| `insert` | $\mathrm{O}(\lg \mathtt{V})$ | $\mathrm{O}(\log_D \mathtt{V})$ | $\mathrm{O}(1)$ |
| `delMin` | $\mathrm{O}(\lg \mathtt{V})$ | $\mathrm{O}(\log_D \mathtt{V})$ | $\mathrm{O}(\lg \mathtt{V})$ |
| `decreaseKey` | $\mathrm{O}(\lg \mathtt{V})$ | $\mathrm{O}(\log_D \mathtt{V})$ | $\mathrm{O}(1)$ |
| `Dijkstra` | $\mathrm{O}(\mathtt{E} \lg \mathtt{V})$ | $\mathrm{O}(\mathtt{E} \log_D \mathtt{V})$ | $\mathrm{O}(\mathtt{E} + \mathtt{V} \lg \mathtt{V})$ |

# 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

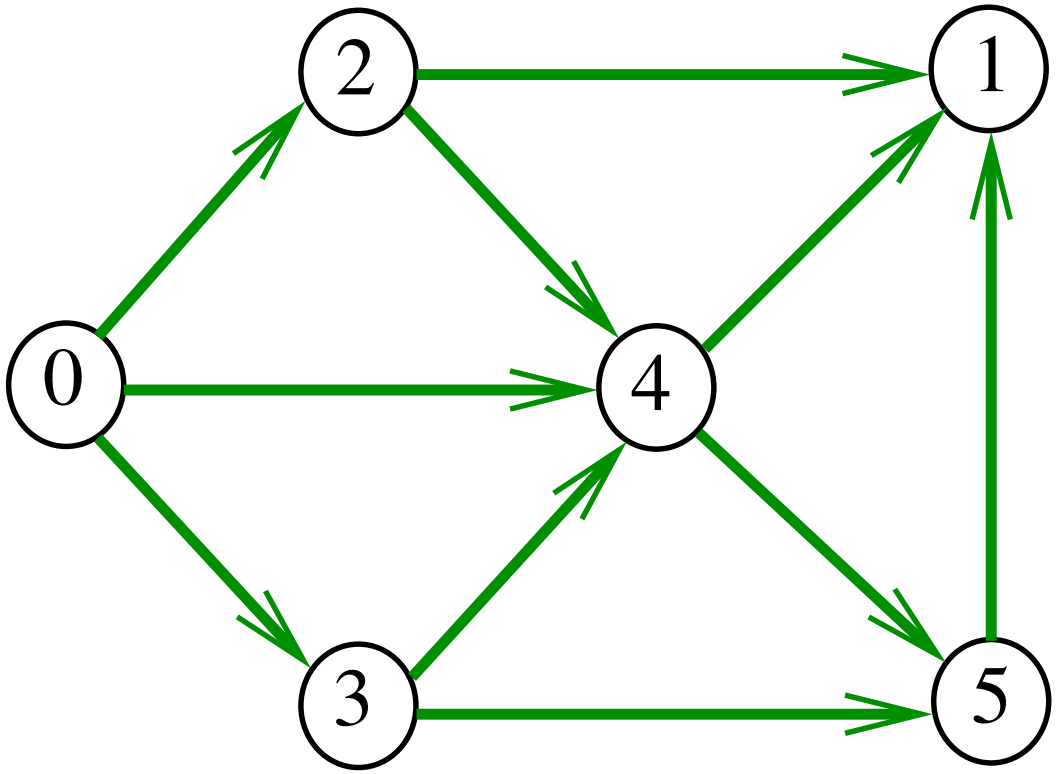


Fonte: Directed acyclic graph

# DAGs

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

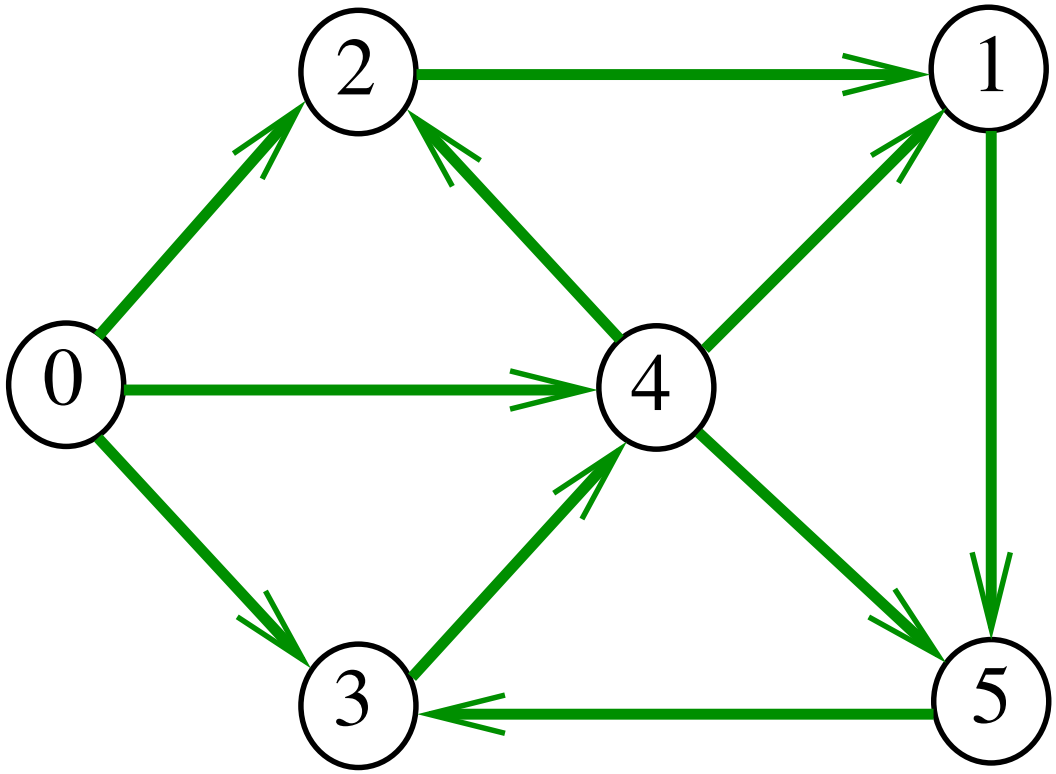
Exemplo: um digrafo acíclico



# DAGs

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

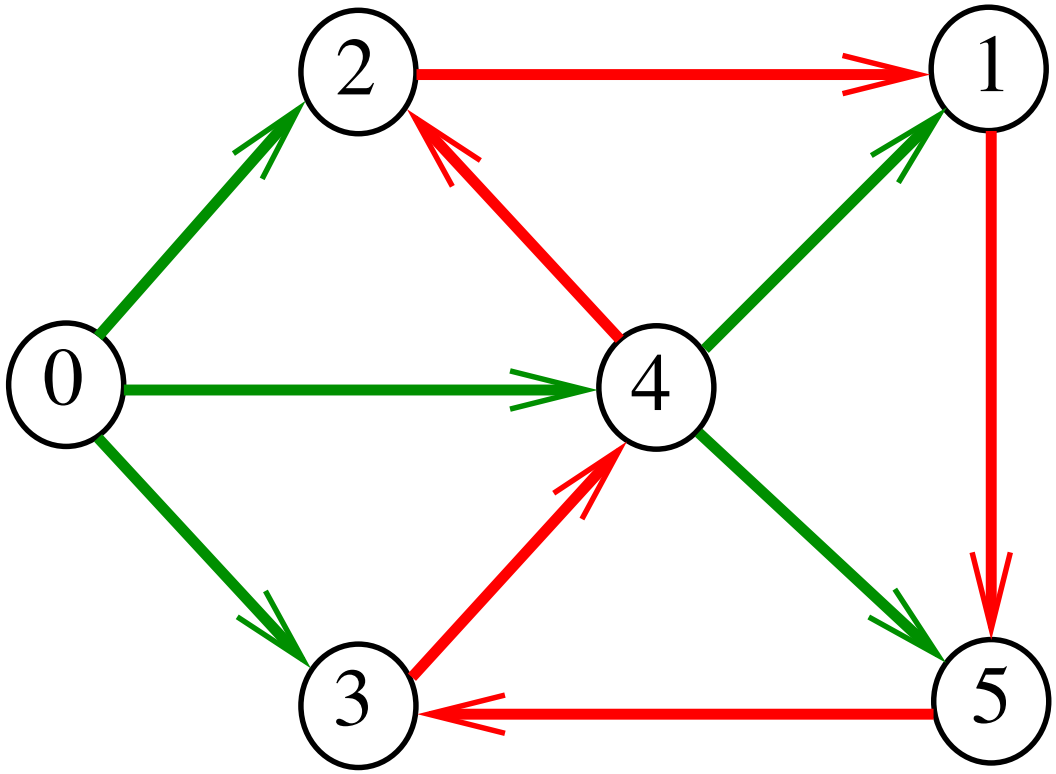
Exemplo: um digrafo que **não** é acíclico



# 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 **permutação** dos vértices de um digrafo é uma seqüência em que cada vértice aparece uma e uma só vez

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

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

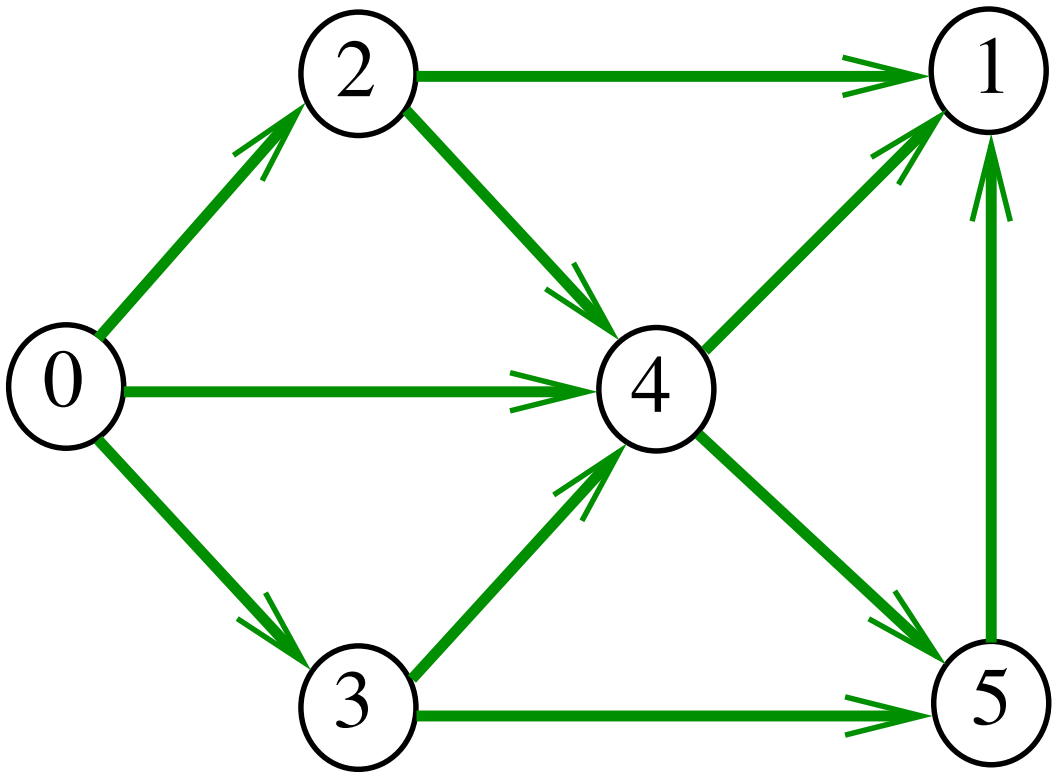
dos seus vértices tal que todo arco tem a forma

`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:

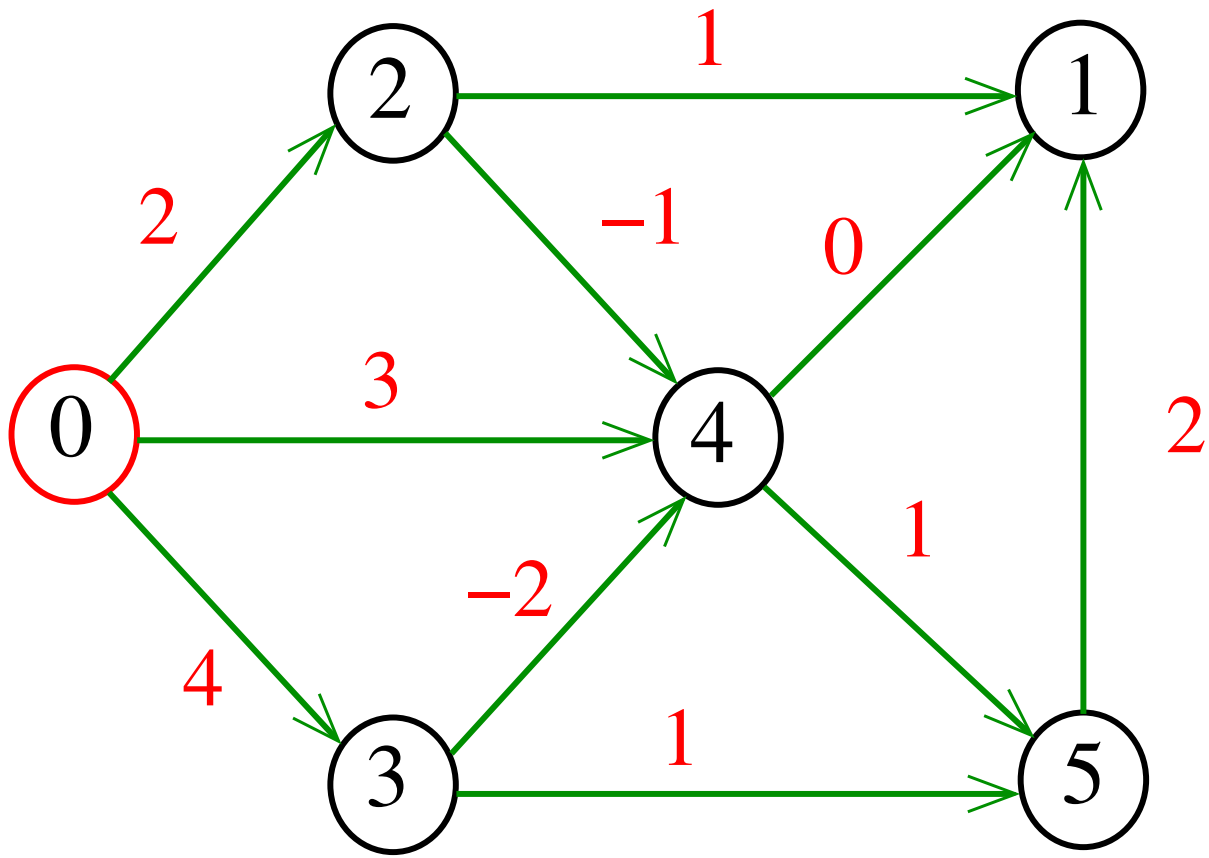
*Dado um vértice $s$ de um DAG com custos* ***possivelmente negativos*** *nos arcos, encontrar, para cada vértice $t$ que pode ser alcançado a partir de $s$, um caminho simples mínimo de $s$ a $t$*

Problema:

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

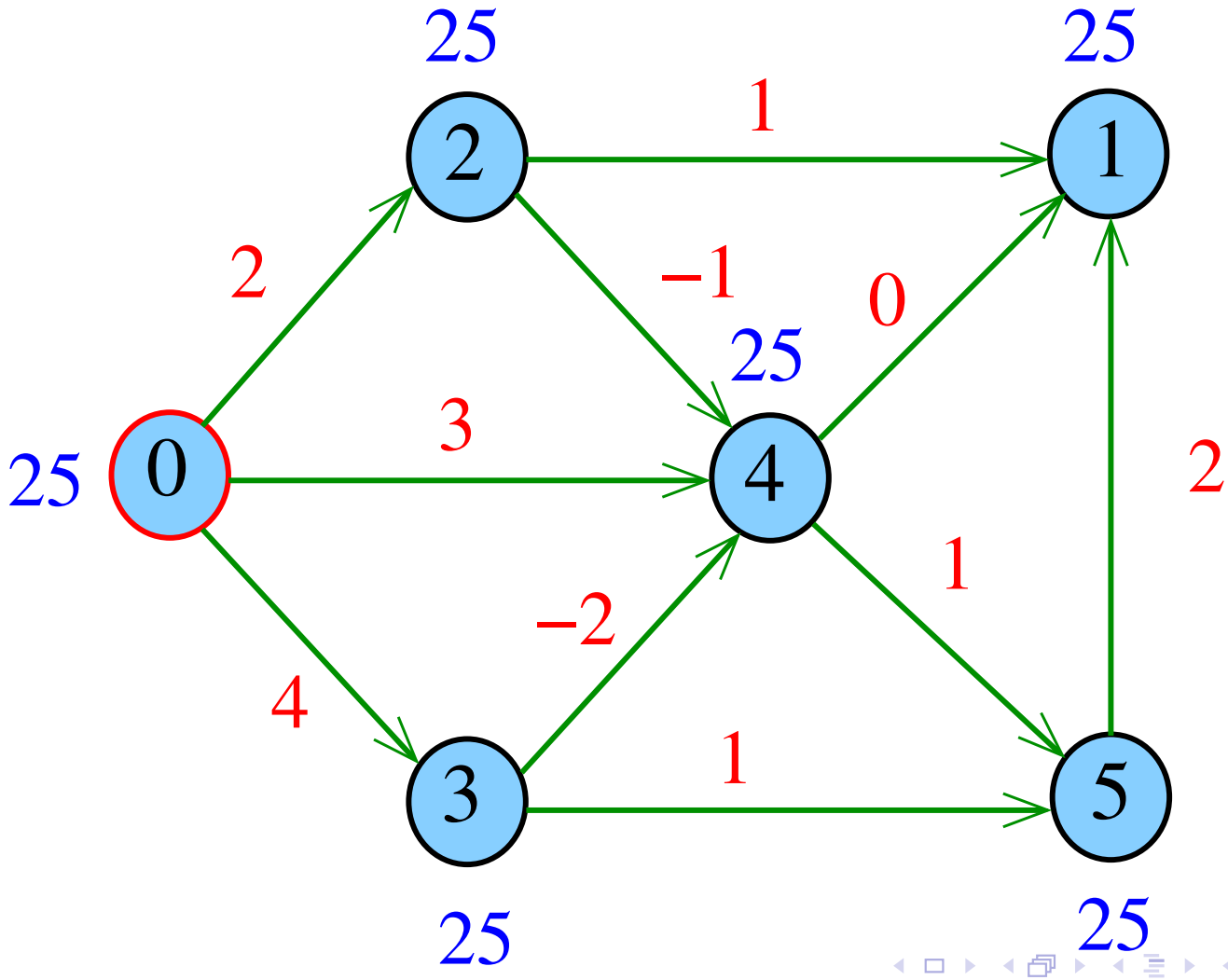
# Simulação

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



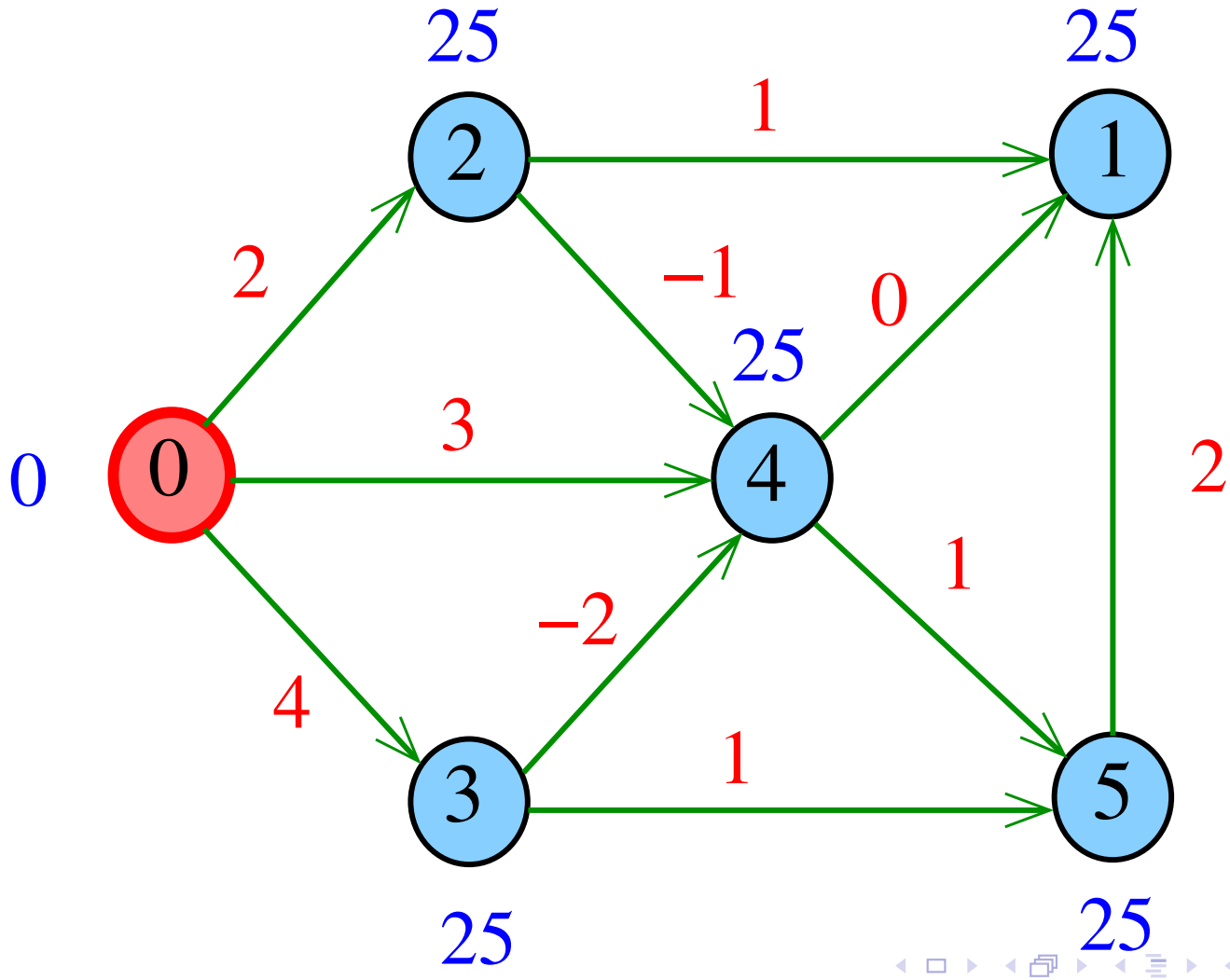
# Simulação

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



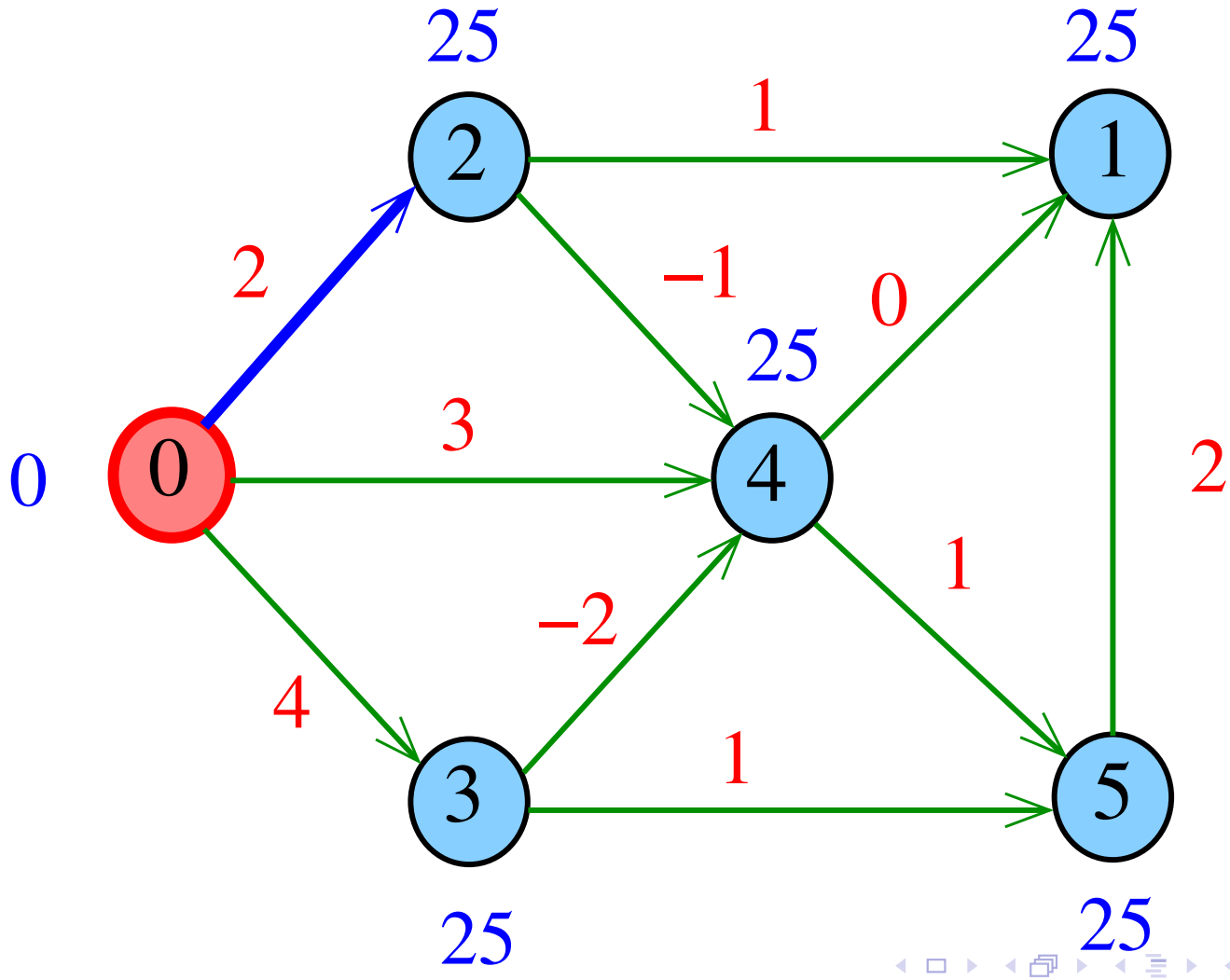
# Simulação

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



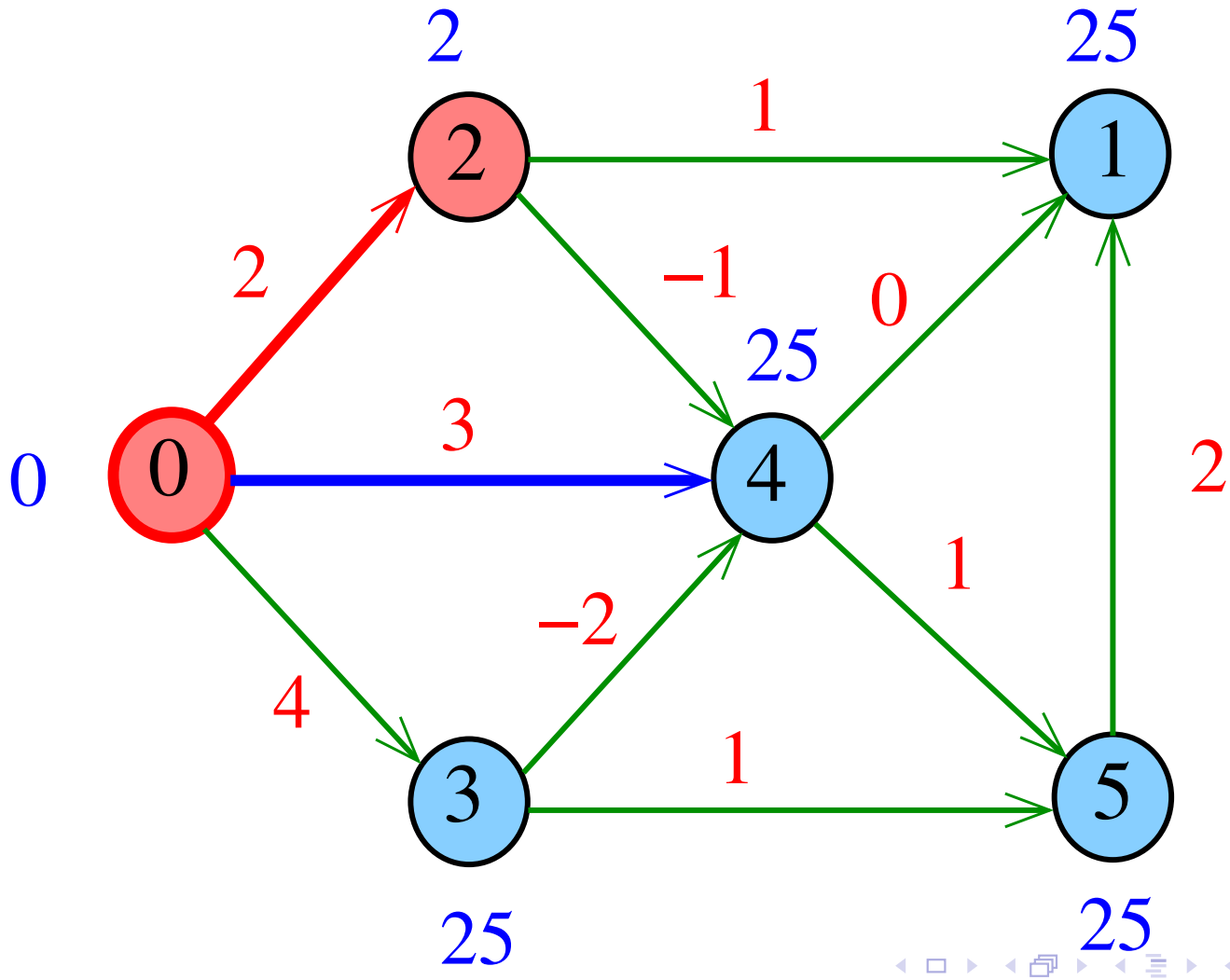
# Simulação

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



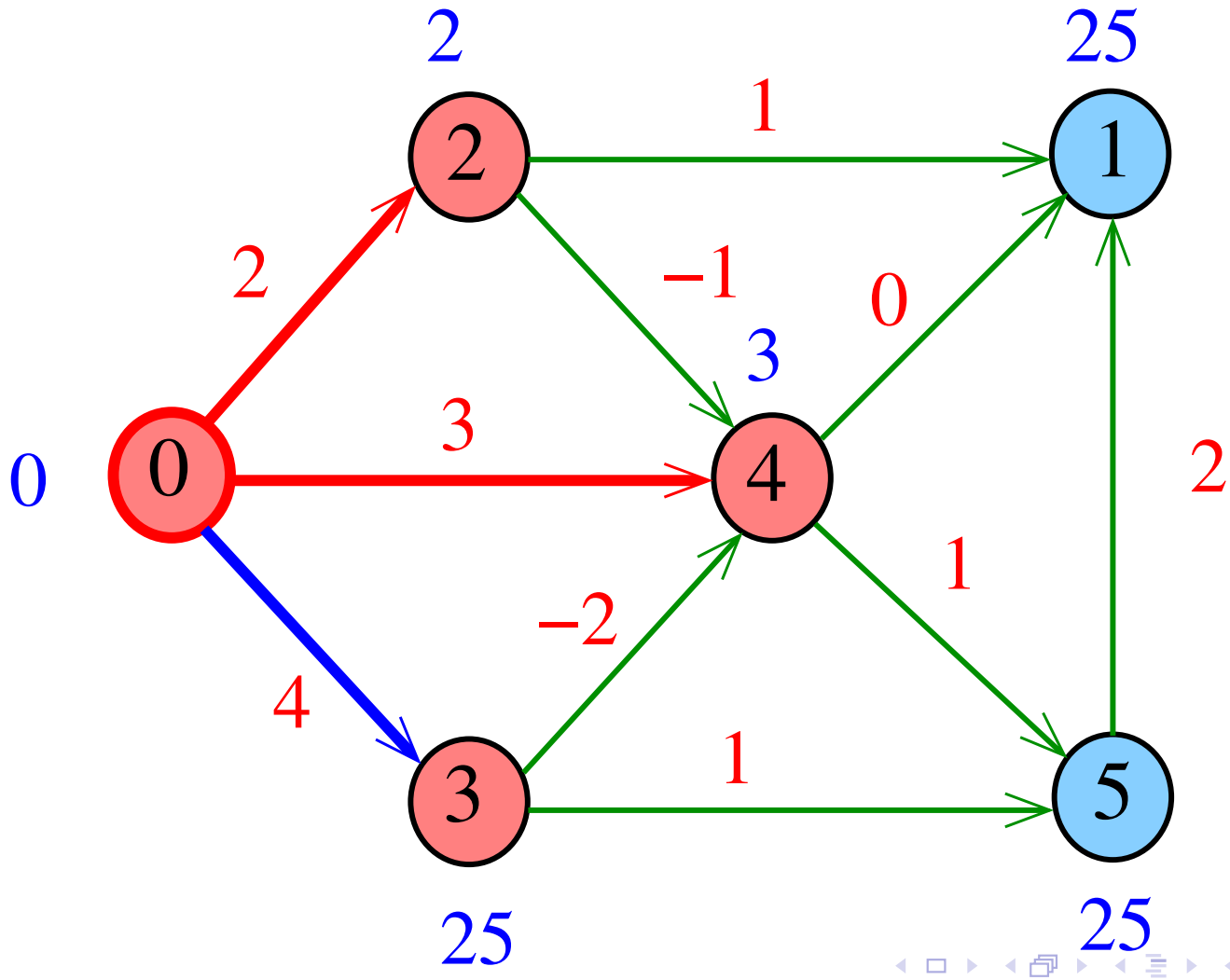
# Simulação

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



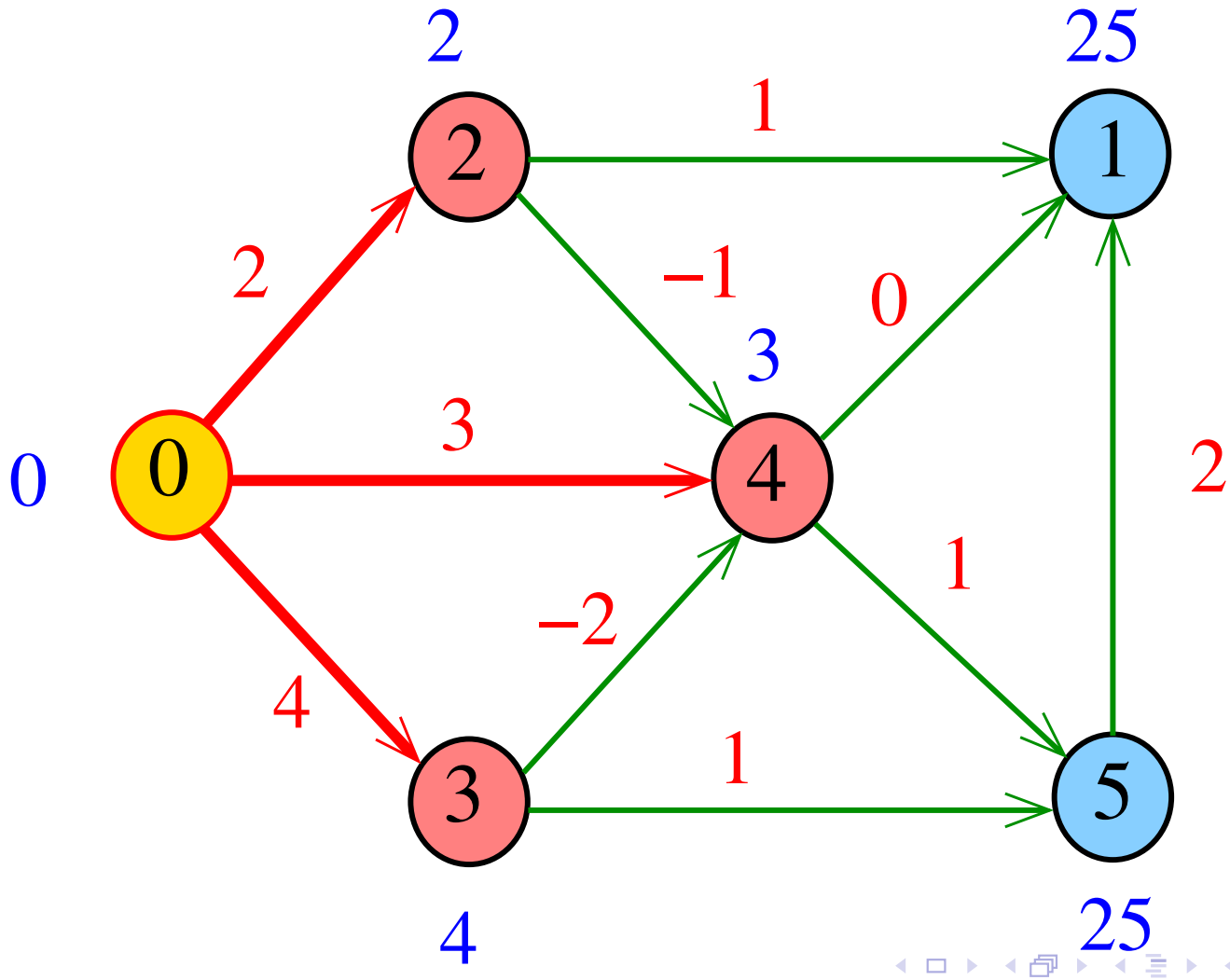
# Simulação

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



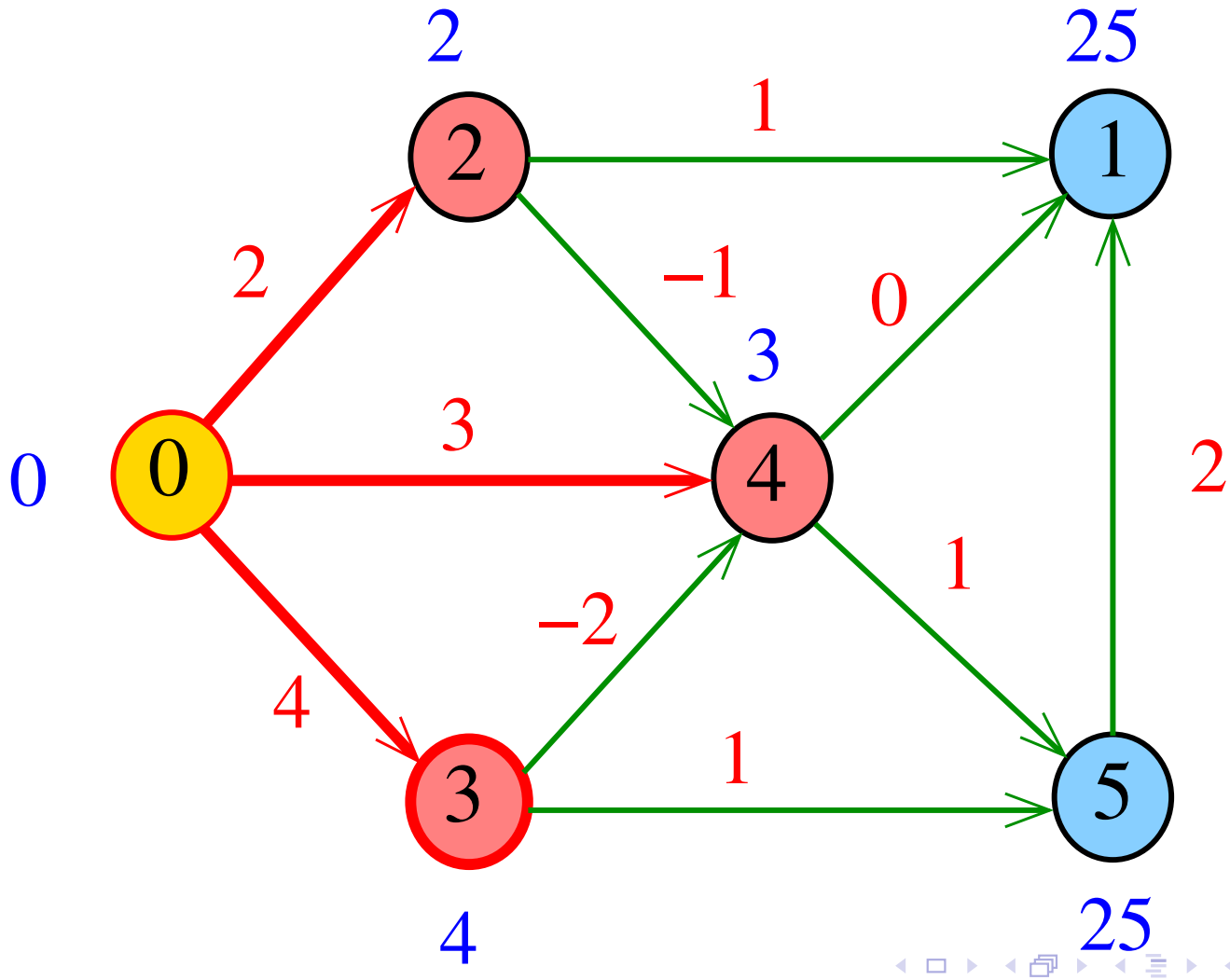
# Simulação

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



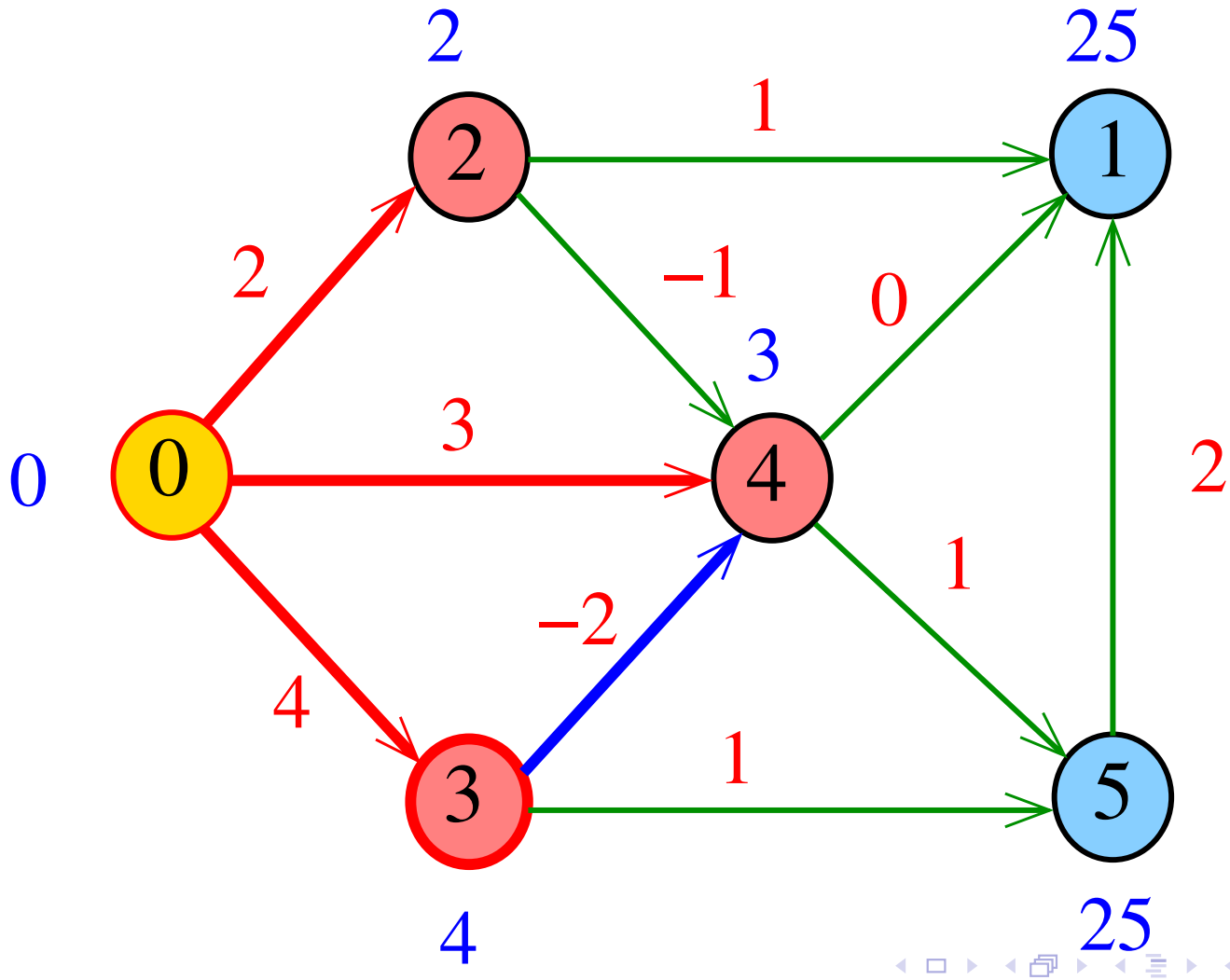
# Simulação

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



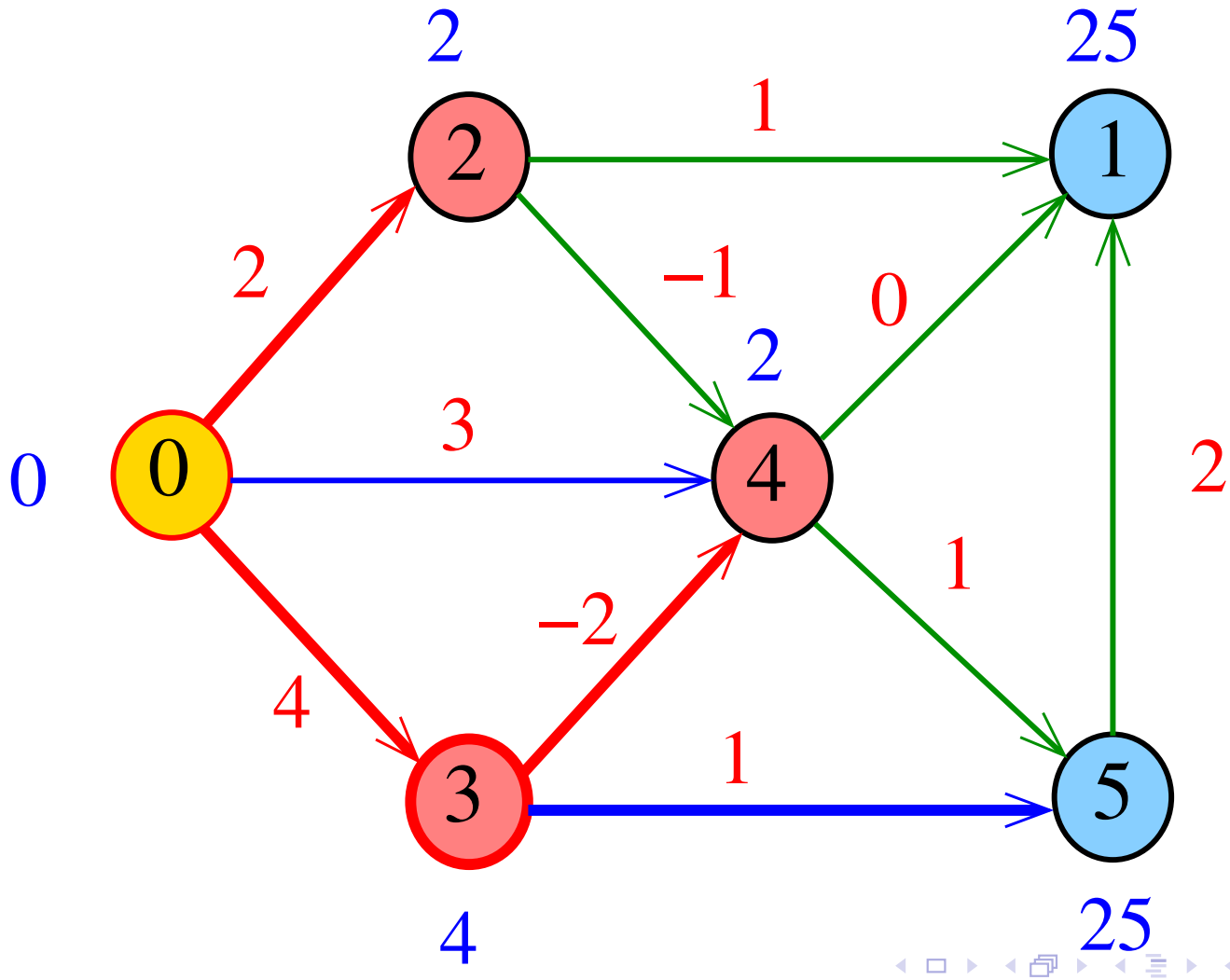
# Simulação

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



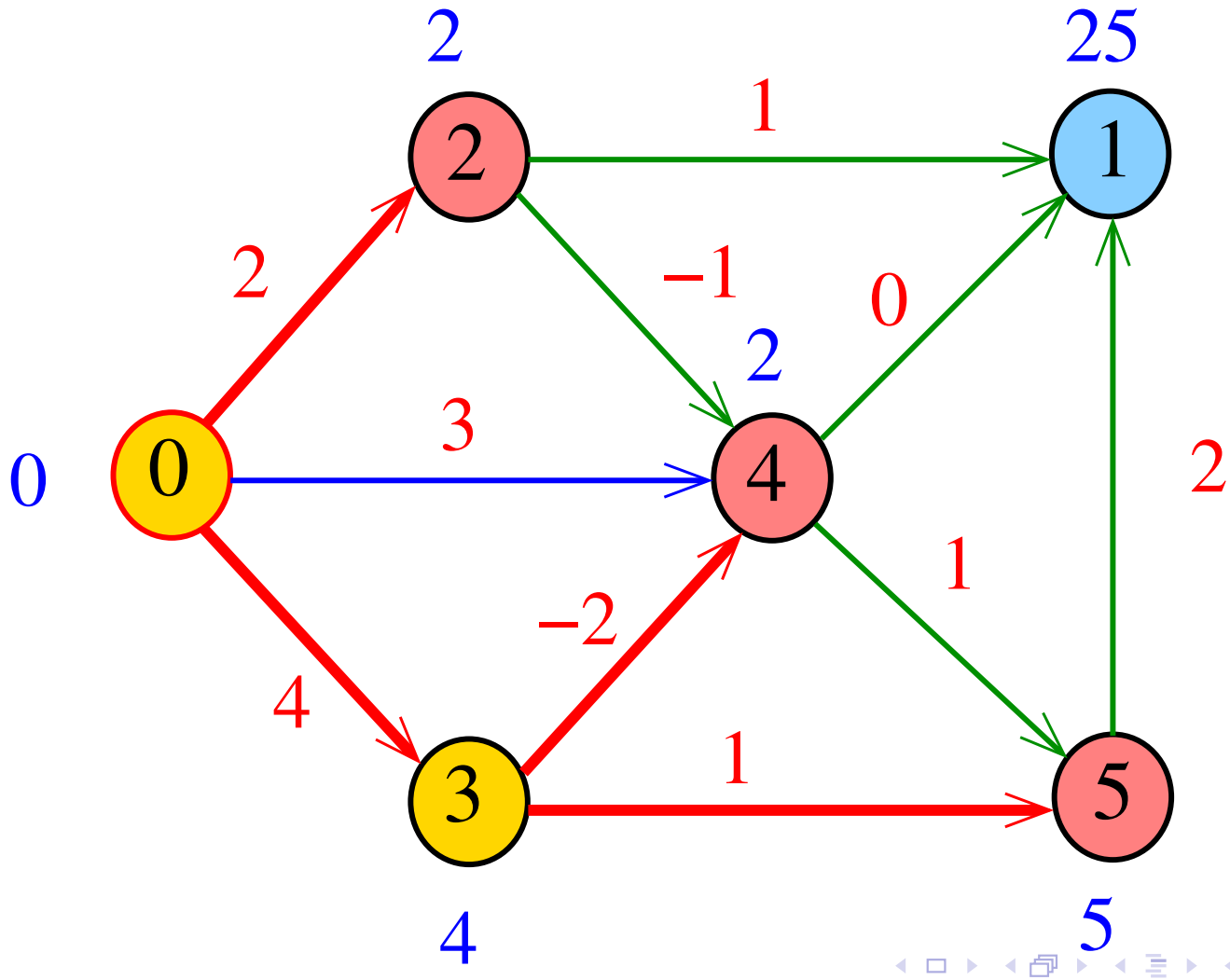
# Simulação

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



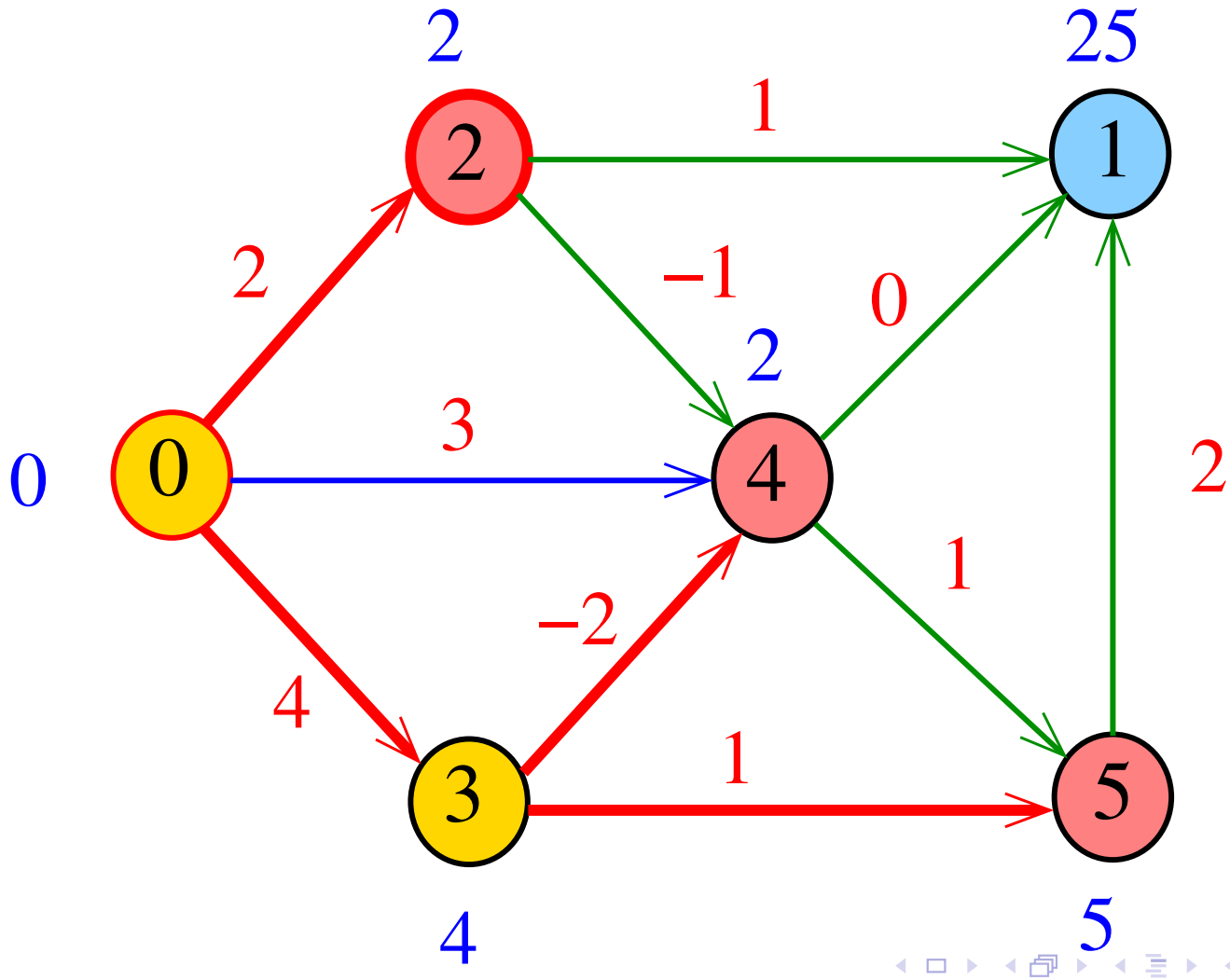
# Simulação

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



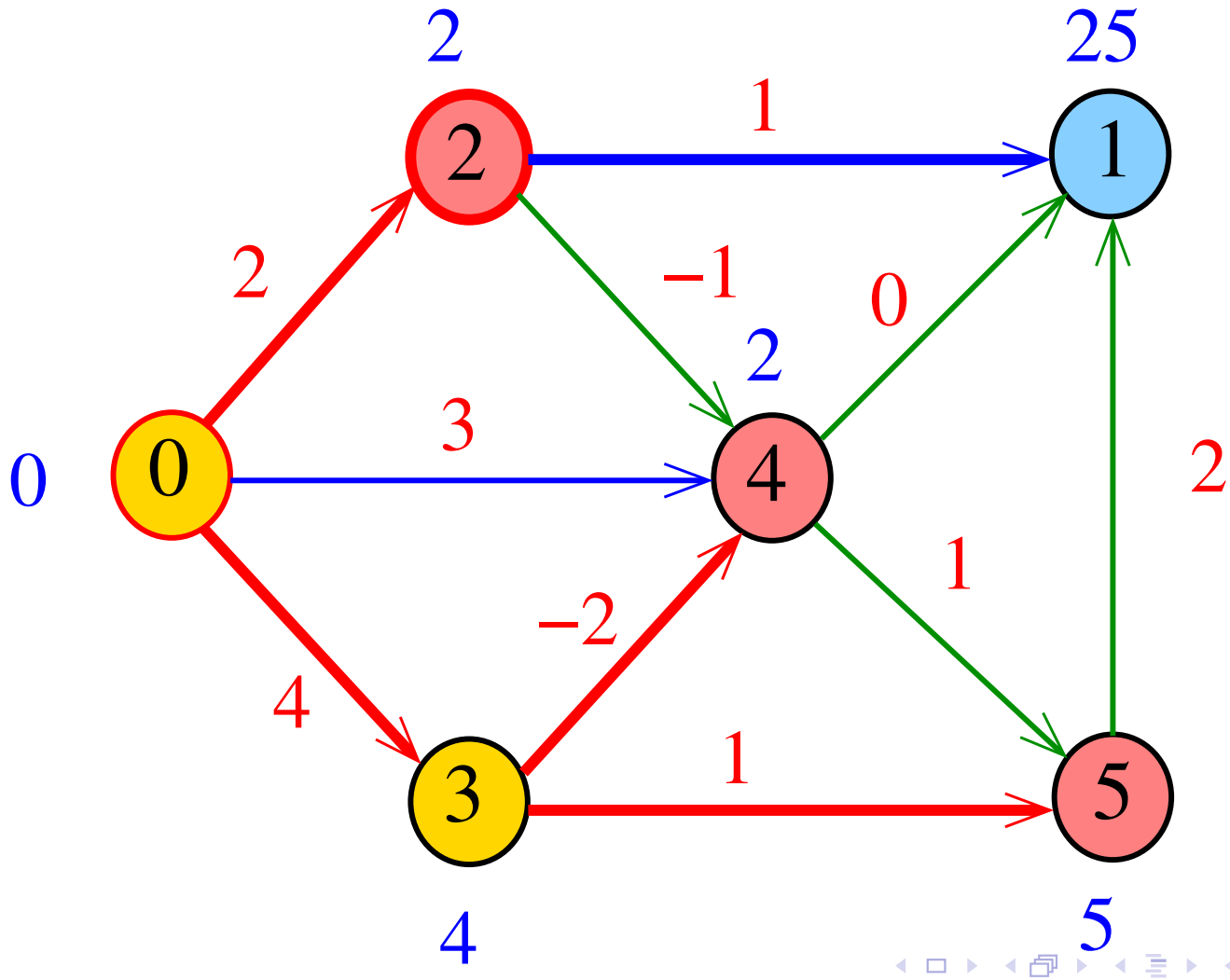
# Simulação

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



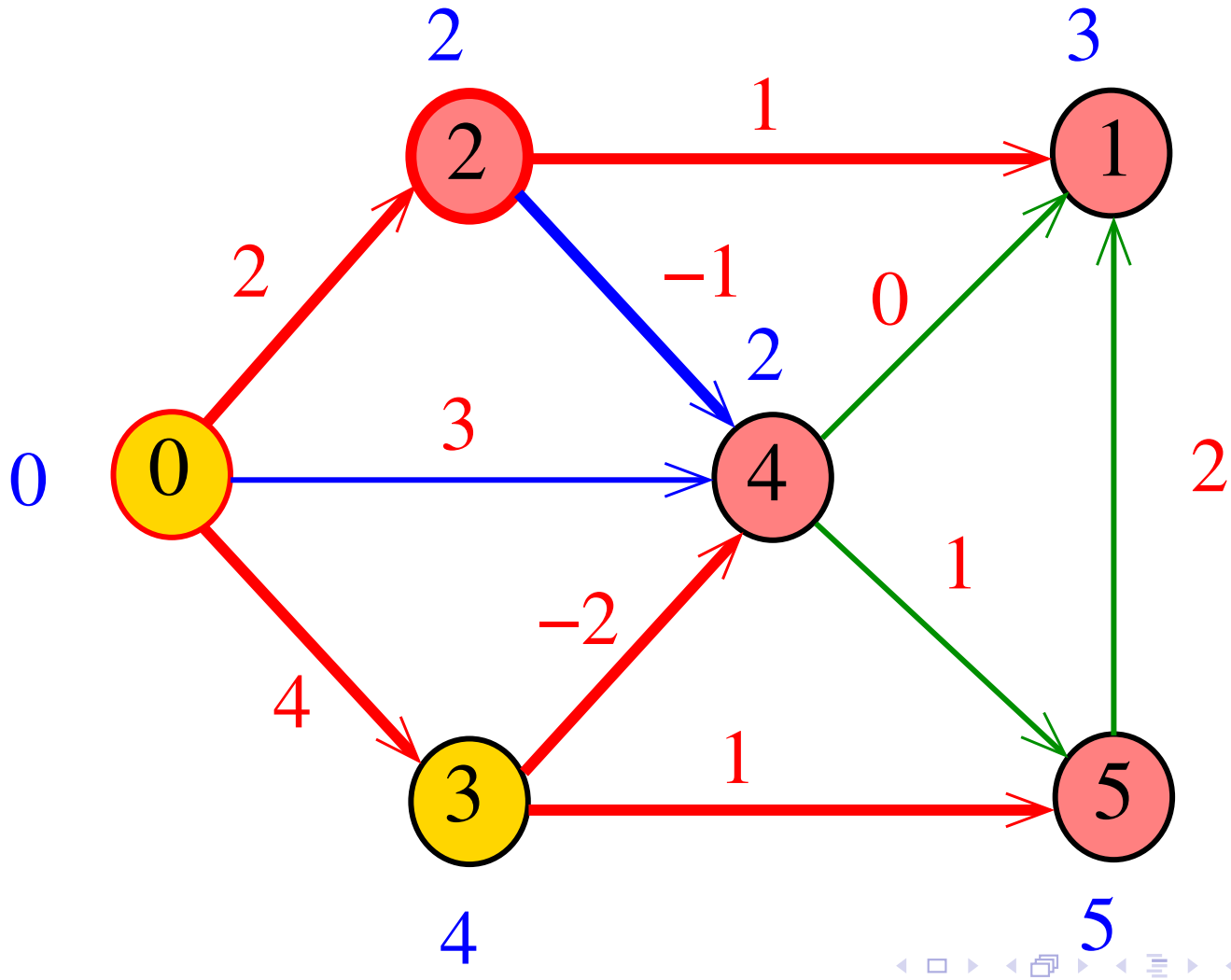
# Simulação

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



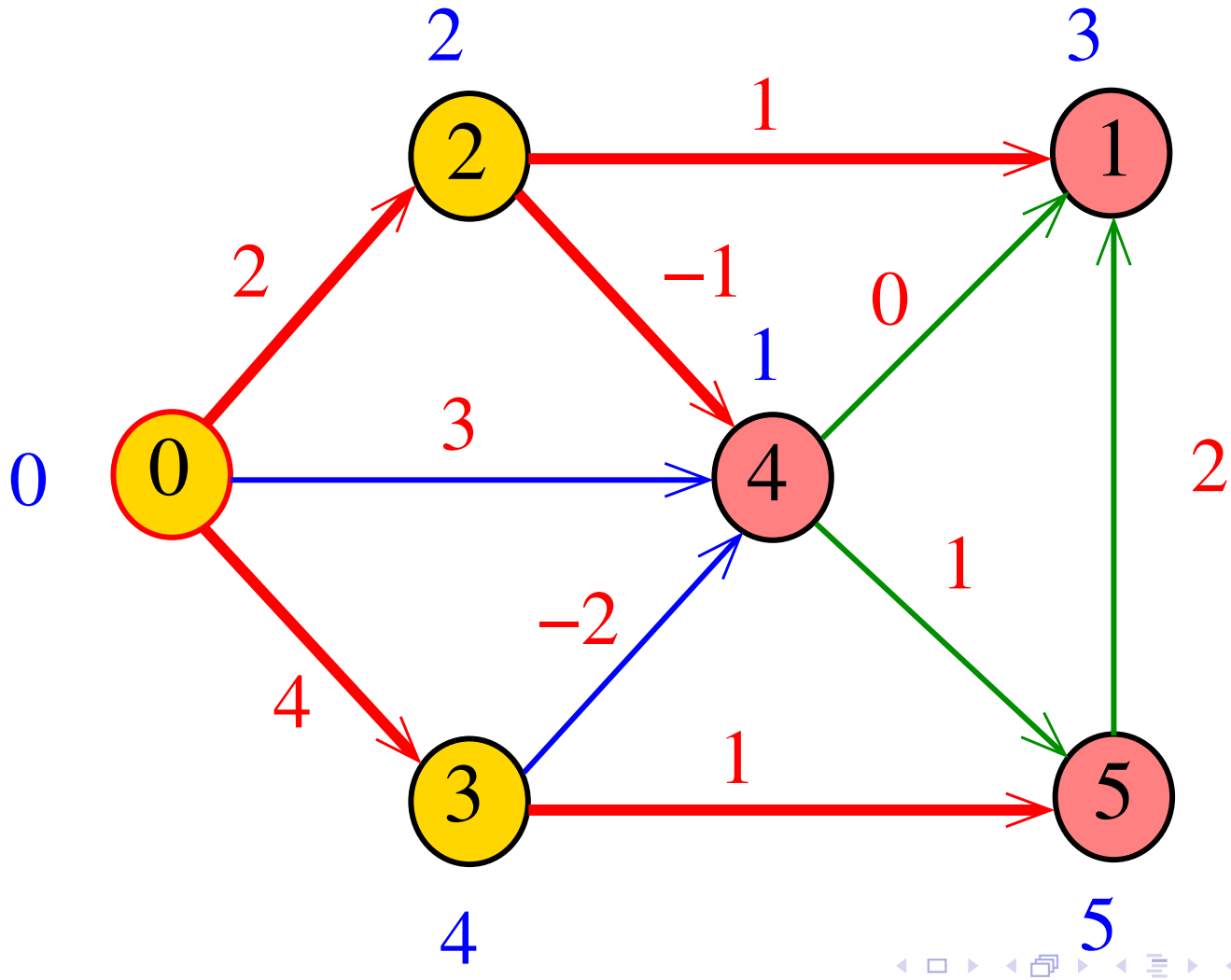
# Simulação

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



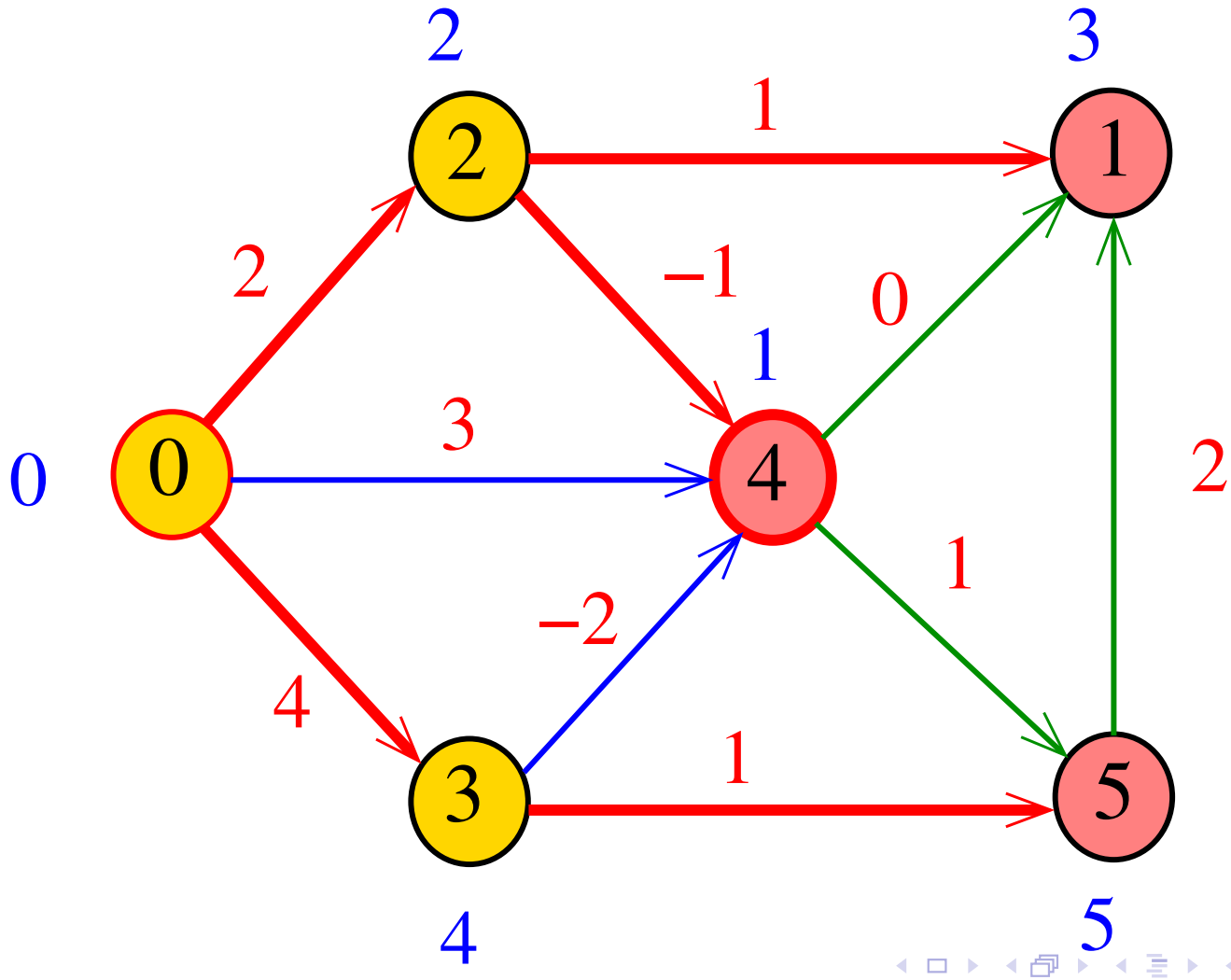
# Simulação

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



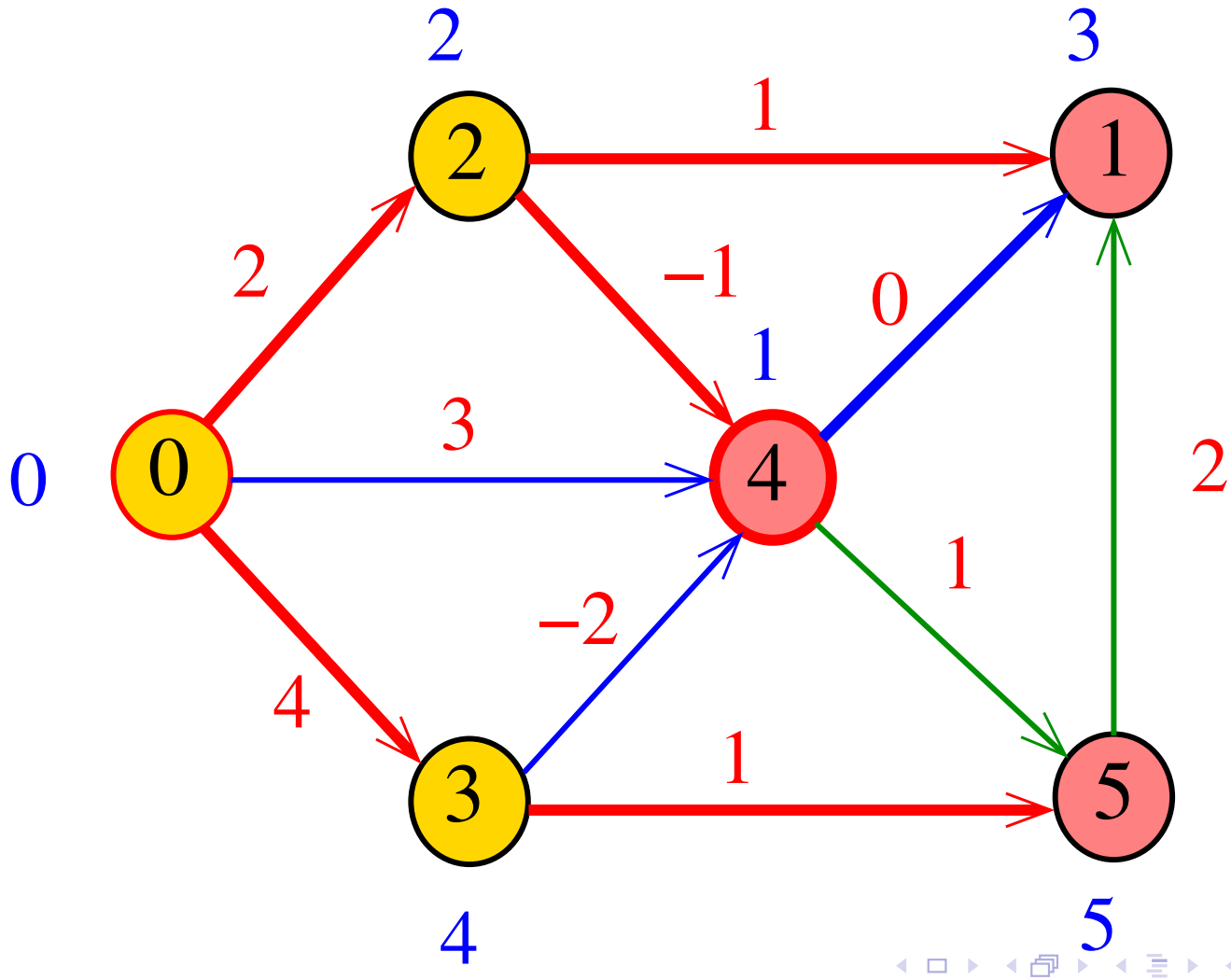
# Simulação

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



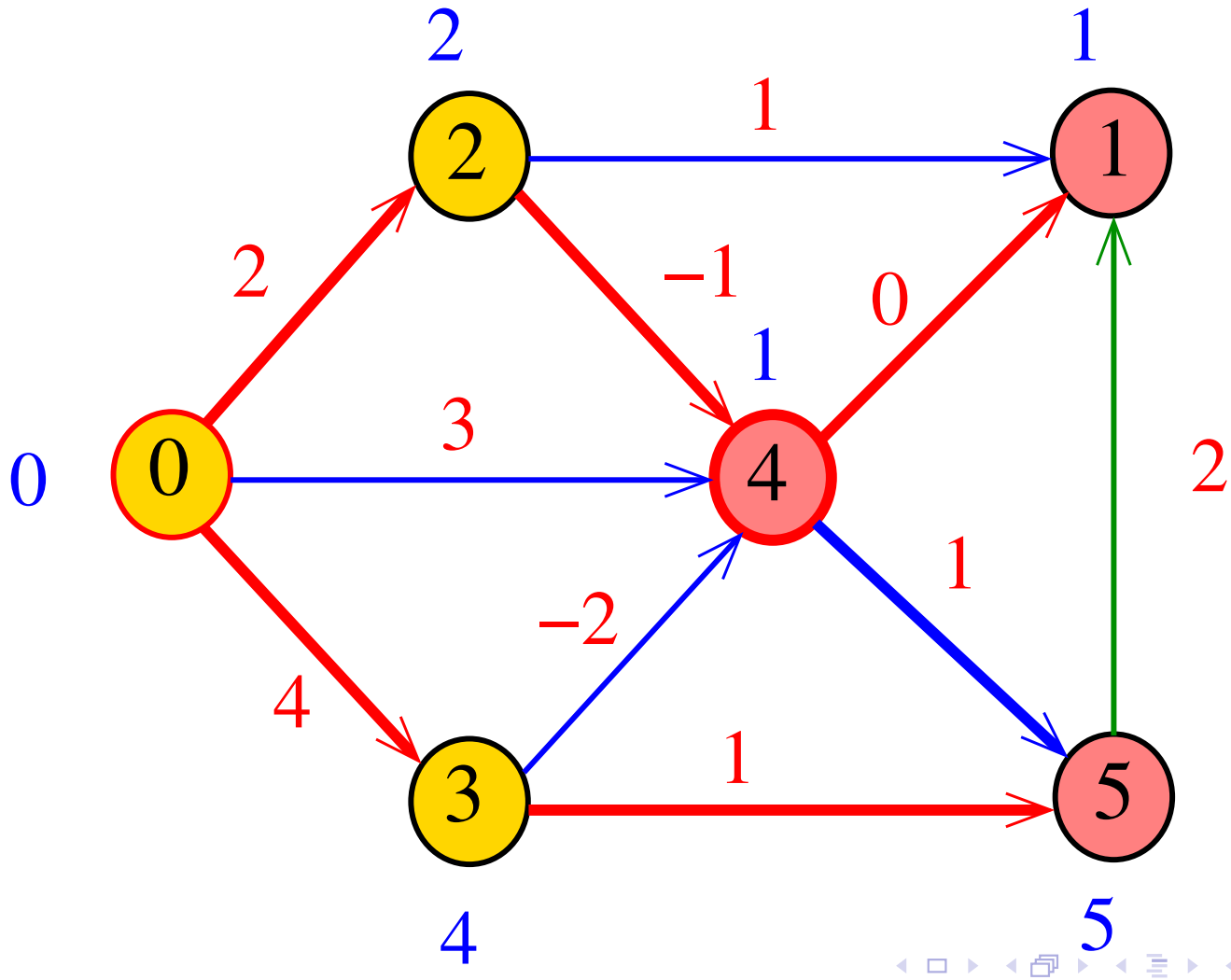
# Simulação

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



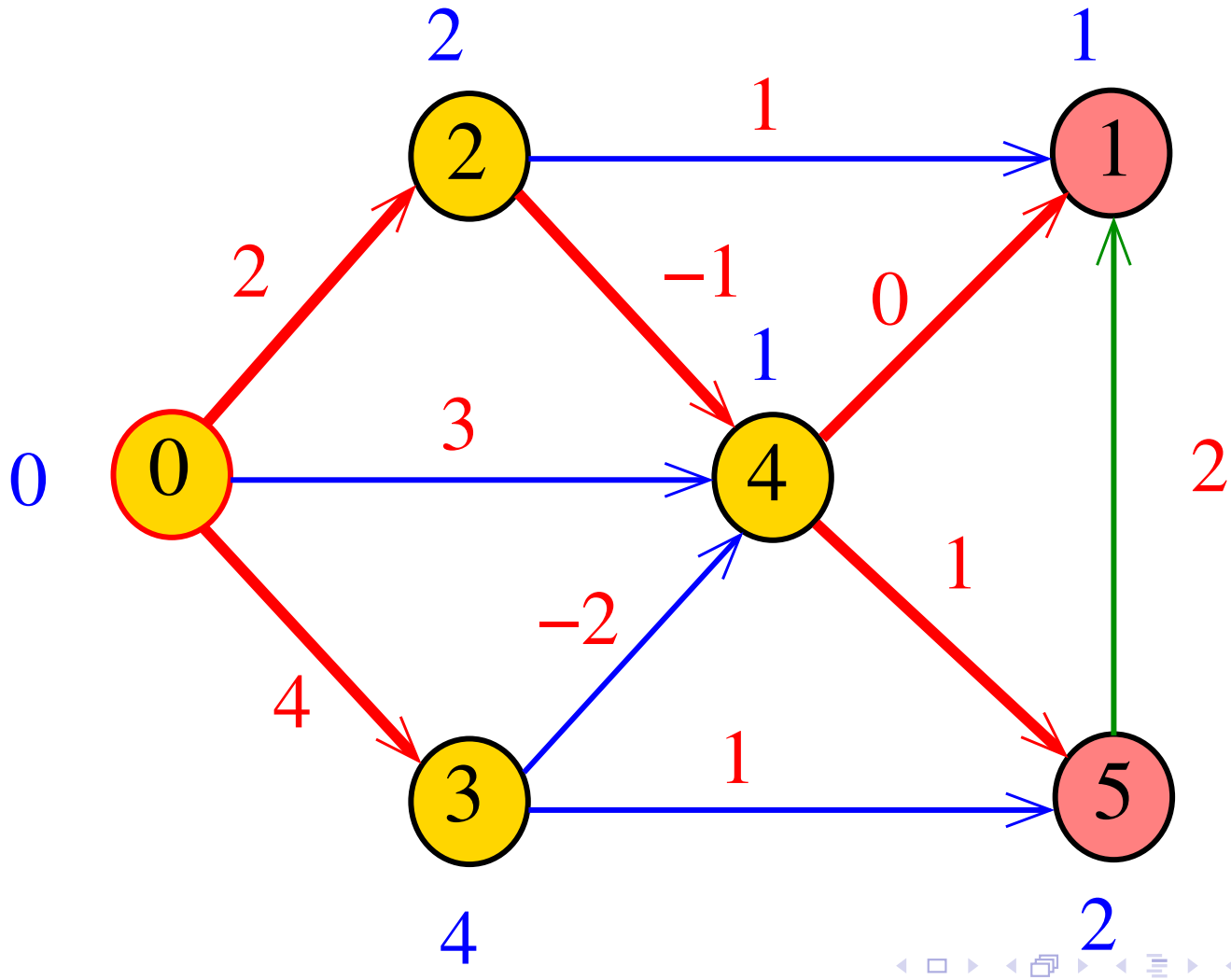
# Simulação

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



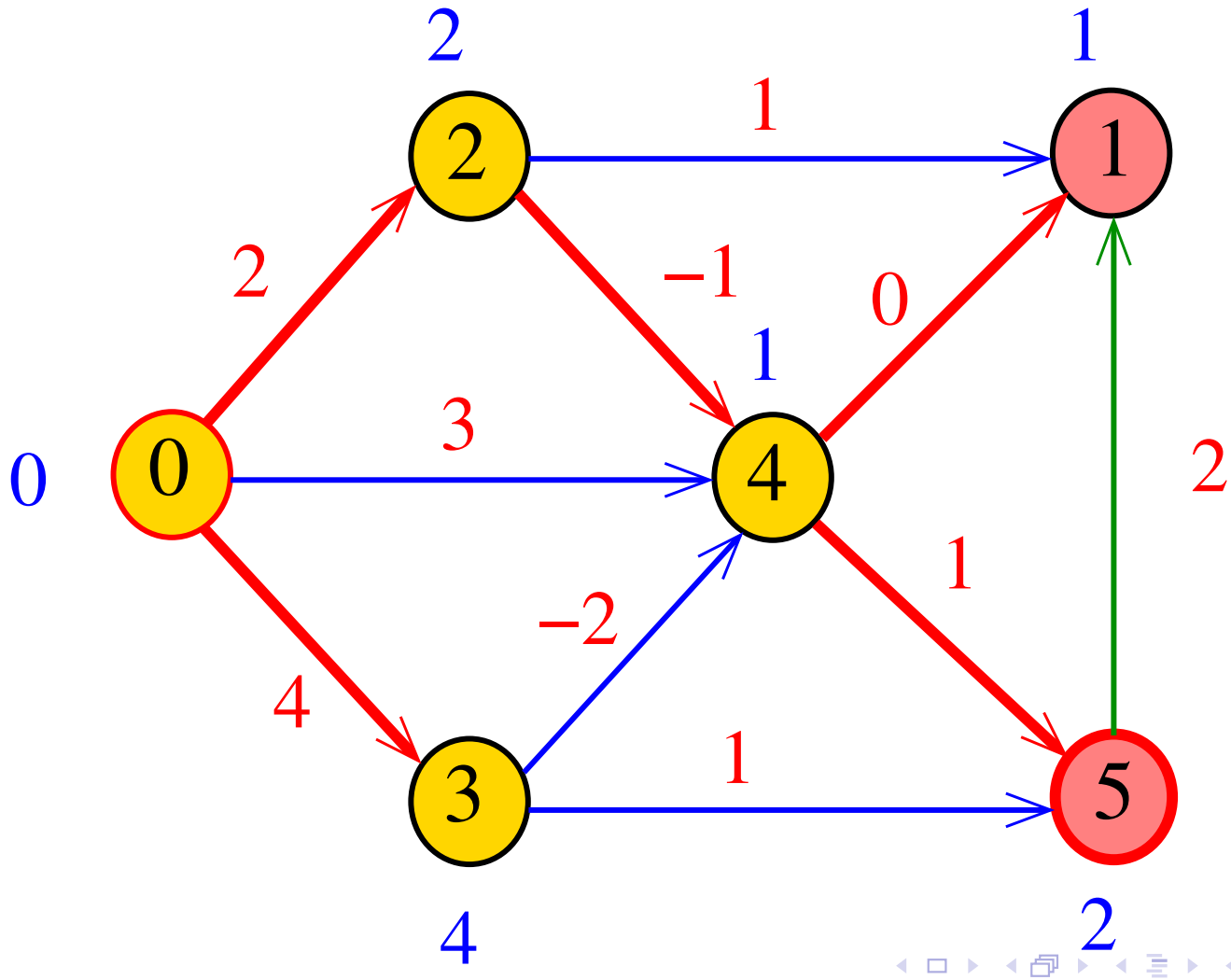
# Simulação

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



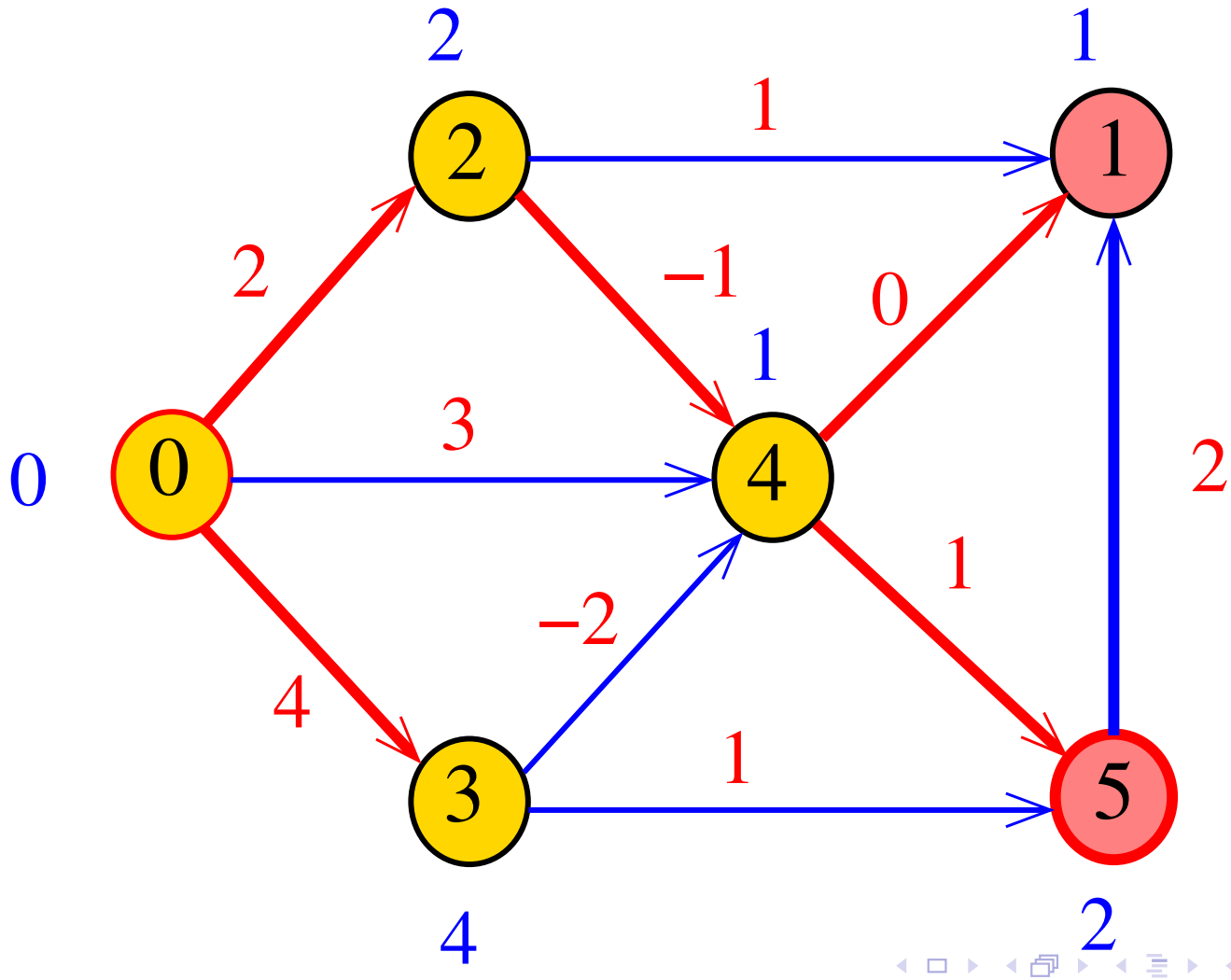
# Simulação

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



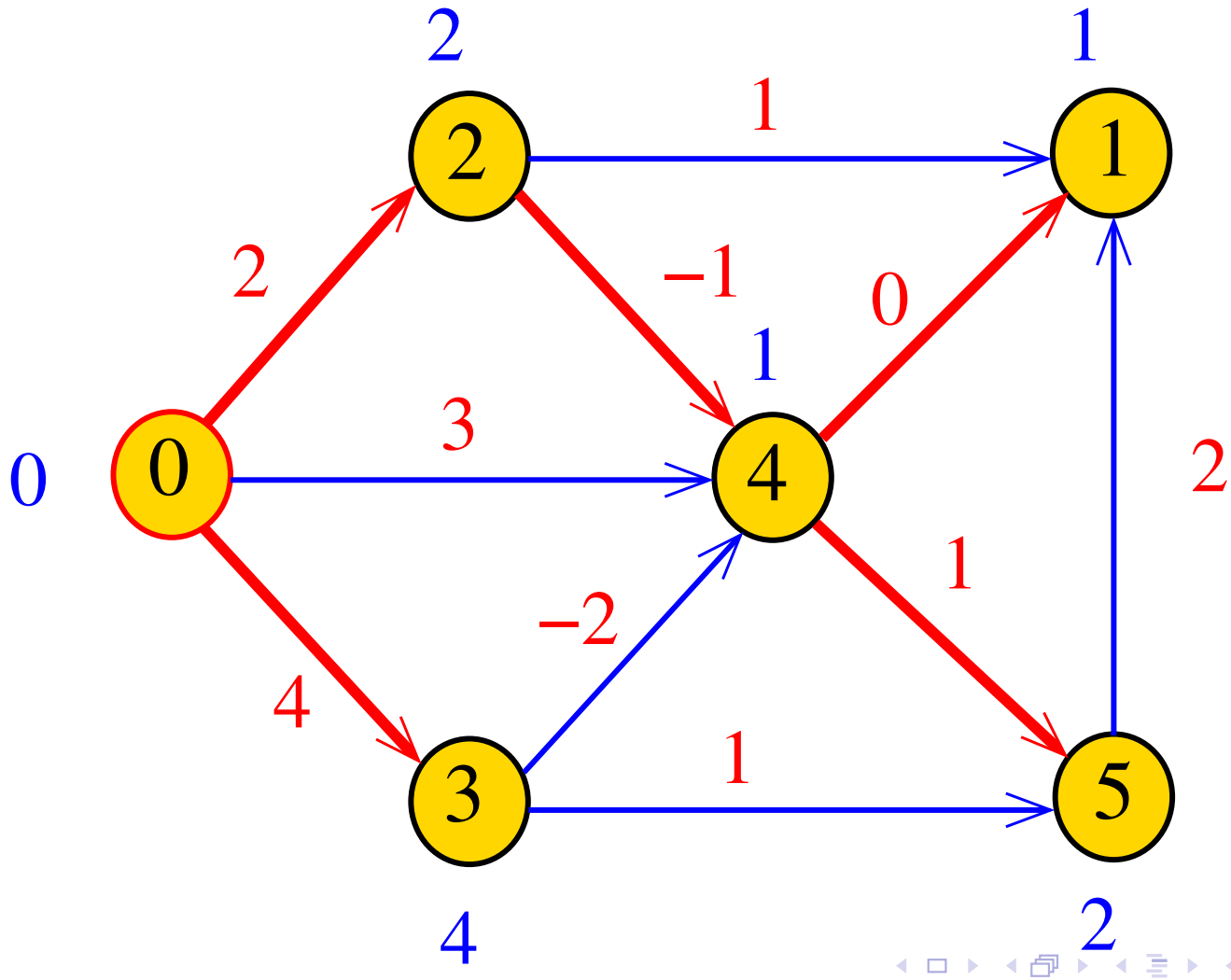
# Simulação

| 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 classe AcyclicSP recebe um DAG G com custos possivelmente negativos. Recebe também um vértice s.

A classe determina uma ordenação topológica dos vértices de Gatravés da classe DFStopological modificada para trabalhar com EWDigraphs.

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

```
public class AcyclicSP(EWDigraph G,
                       int s);
```

# Classe AcyclicSP: esqueleto

```
public class AcyclicSP {
   private static final double INFINITY;
   private final int s;
   private double[] distTo;
   private Arc[] edgeTo;
   public AcyclicSP(EWDigraph G, int s) {}
   private void acyclic(EWDigraph G,
                        int s) {}
   public boolean hasPath(int v) {}
   public boolean distTo(int v) {}
   public Iterable<Arc> pathTo(int v)
}
```

# AcyclicSP

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

```
public AcyclicSP(EWDigraph G, int s) {
   INFINITY = Double.POSITIVE_INFINITY;
   distTo = new double[G.V()];
   edgeTo = new Arc[G.V()];
   this.s = s;
   for (int v = 0; v < G.V(); v++)
       distTo[v] = INFINITY;
   acyclic(G, s);
}
```

# acyclic()

```
private void acyclic(EWDigraph G, int s){
   DFStopological ts= new DFStopological(G);
   distTo[s] = 0;
   for(int v:  ts.order()) {
       for (Arc e :  G.adj(v)) {
           int w = e.to();
           double d = distTo[v]+e.weight();
           if (distTo[w] > d) {
              edgeTo[w] = e;
              distTo[w] = d;
           }
       }
   }
```

# AcyclicSP

Há um caminho de s a v?

```
// Método copiado de BFSpaths.
public boolean hasPath(int v) {
   return distTo[v] < INFINITY;
}

// retorna o comprimento de um
// caminho mínimo de s a t
public int distTo(int v) {
   return distTo[v];
}
```

# AcyclicSP

Retorna um caminho de s a v ou null se um tal caminho não existe.

```
// Método adaptado de DFSpaths.
public Iterable<Arc> pathTo(int v) {
   if (!hasPath(v)) return null;
   Stack<Arc> path = new Stack<Arc>();
   for (Arc e = edgeTo[v]; e != null;
                  e = edgeTo[e.from()])
      path.push(e);
   return path;
}
```

# 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 $O(V^2)$.