Melhores momentos

AULAS PASSADAS

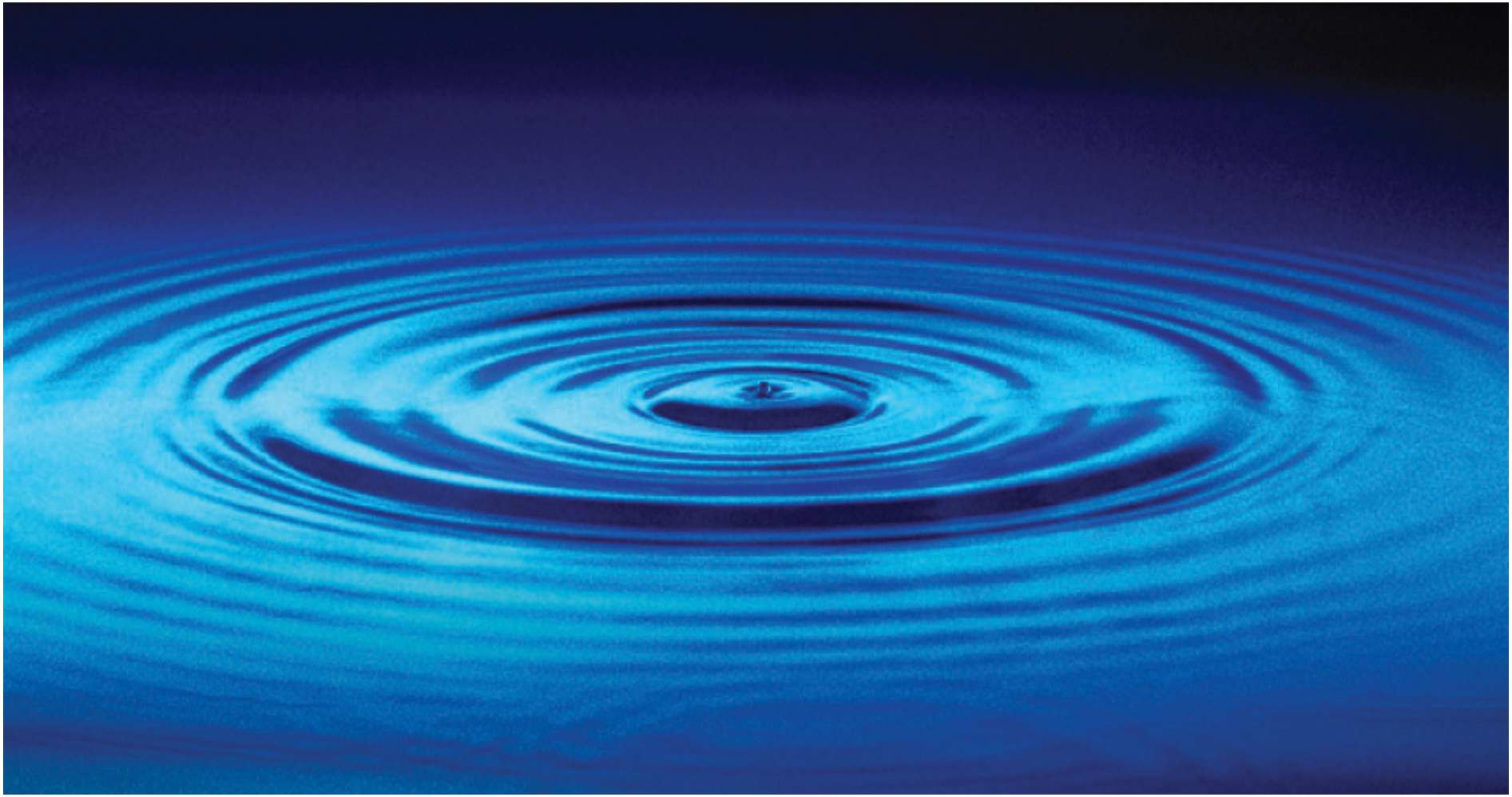
DFS

Usamos **busca em profundidade** para encontrar:

- caminho de s a tou st-corte (S, T) em que todo arco no corte tem ponta inicial em T e ponta final em S;
- ciclo em digrafos ou ordenação topologica;
- componentes de grafos;
- bipartição de grafos ou ciclo ímpar;

AULA 18

Busca em largura



Fonte: http://catalog.flatworldknowledge.com/bookhub/

Busca ou varredura

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

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

Busca em largura

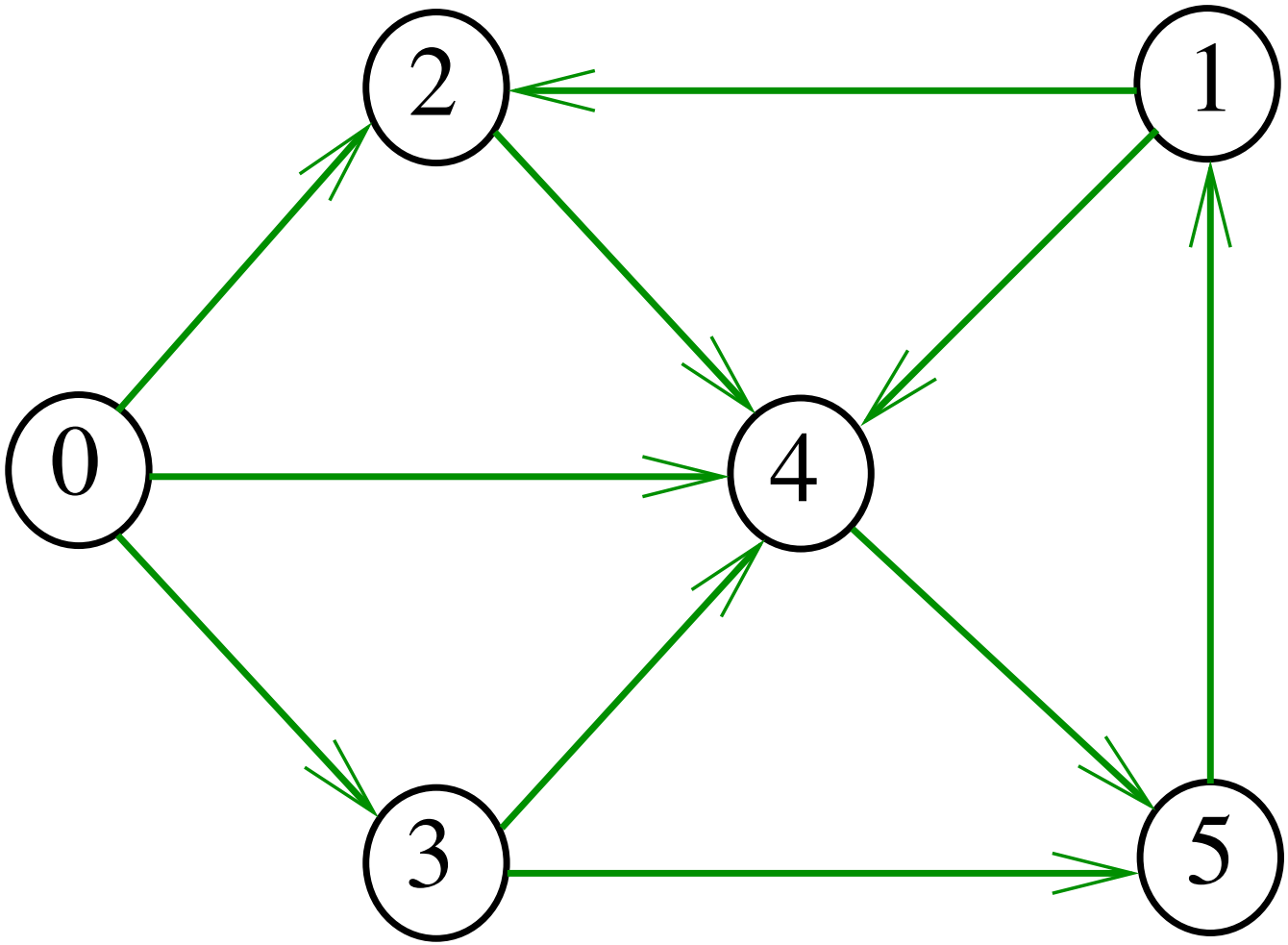
A **busca em largura** (=*breadth-first search search* = *BFS*) começa por um vértice, digamos `s`, especificado pelo usuário.

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

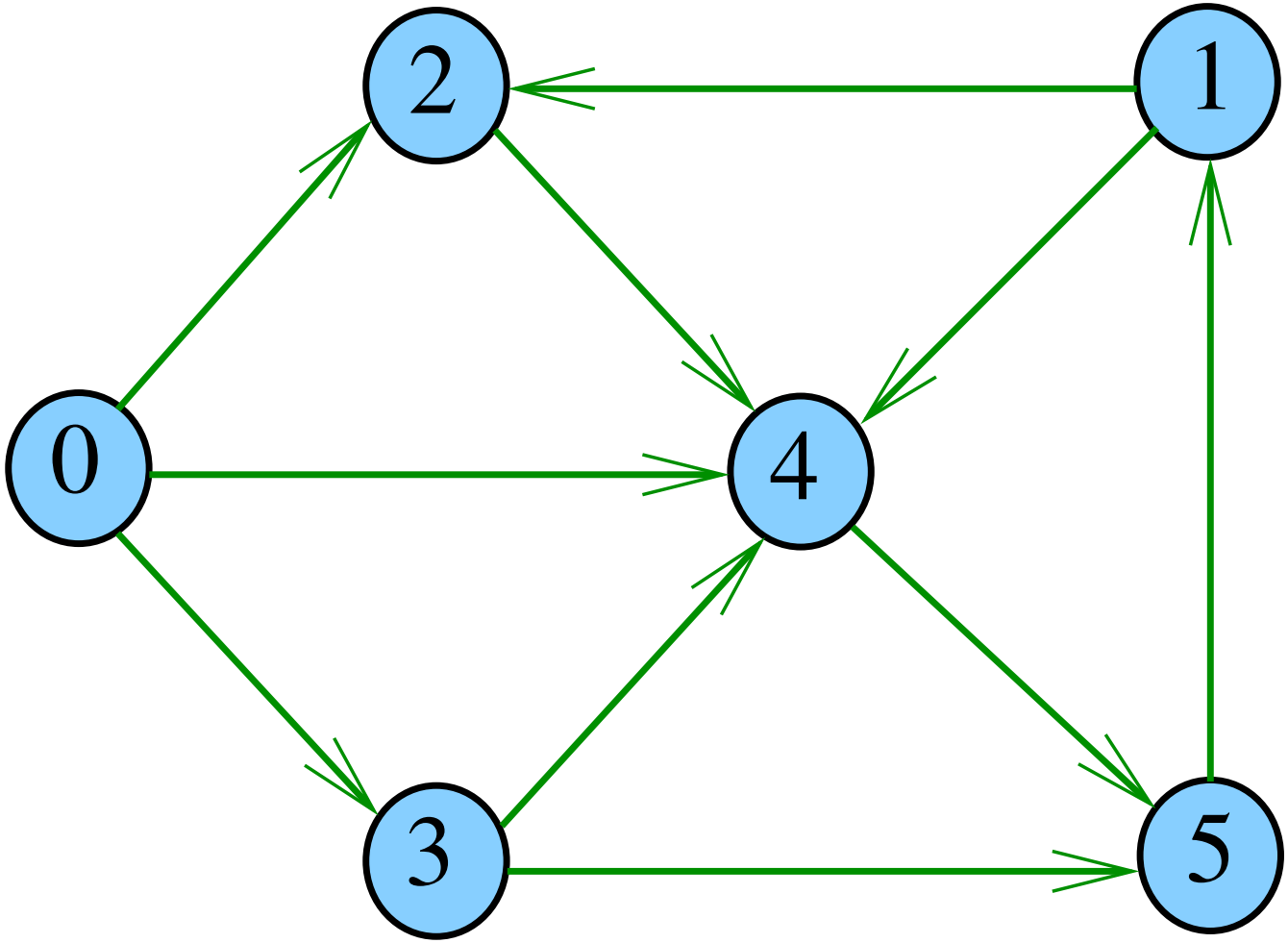
Simulação

i	0	1	2	3	4	5
q[i]						



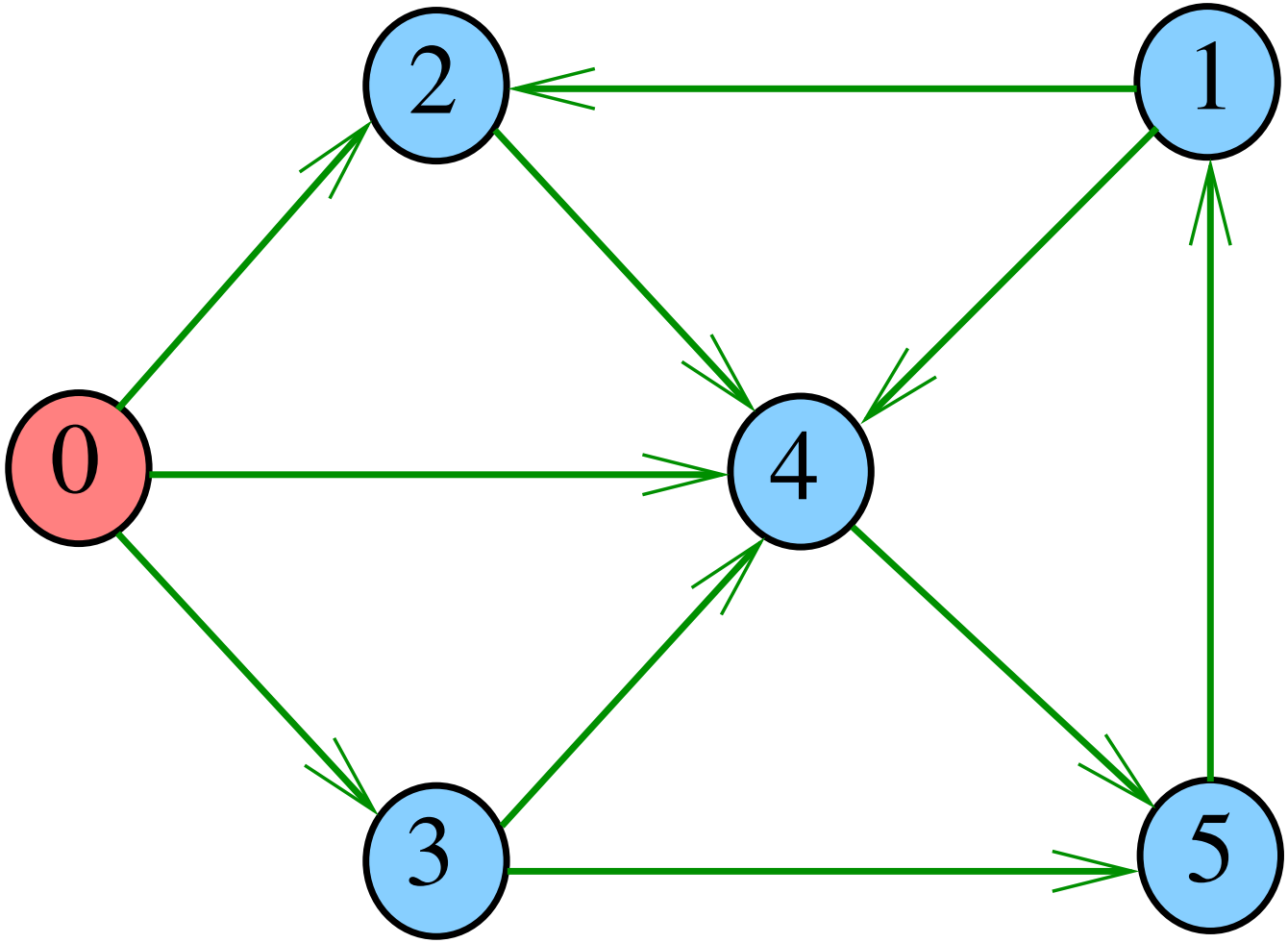
Simulação

i	0	1	2	3	4	5
q[i]						



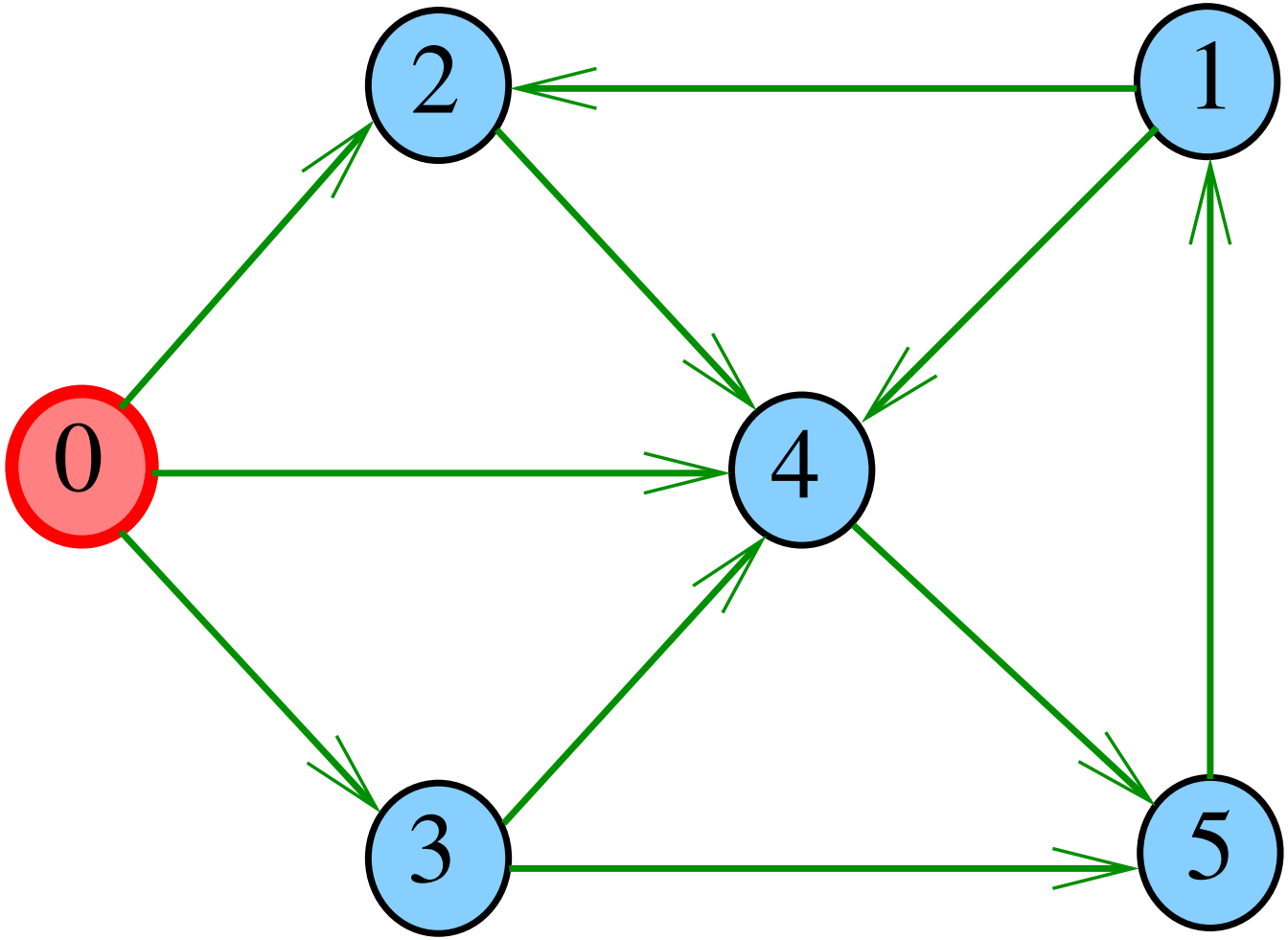
Simulação

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



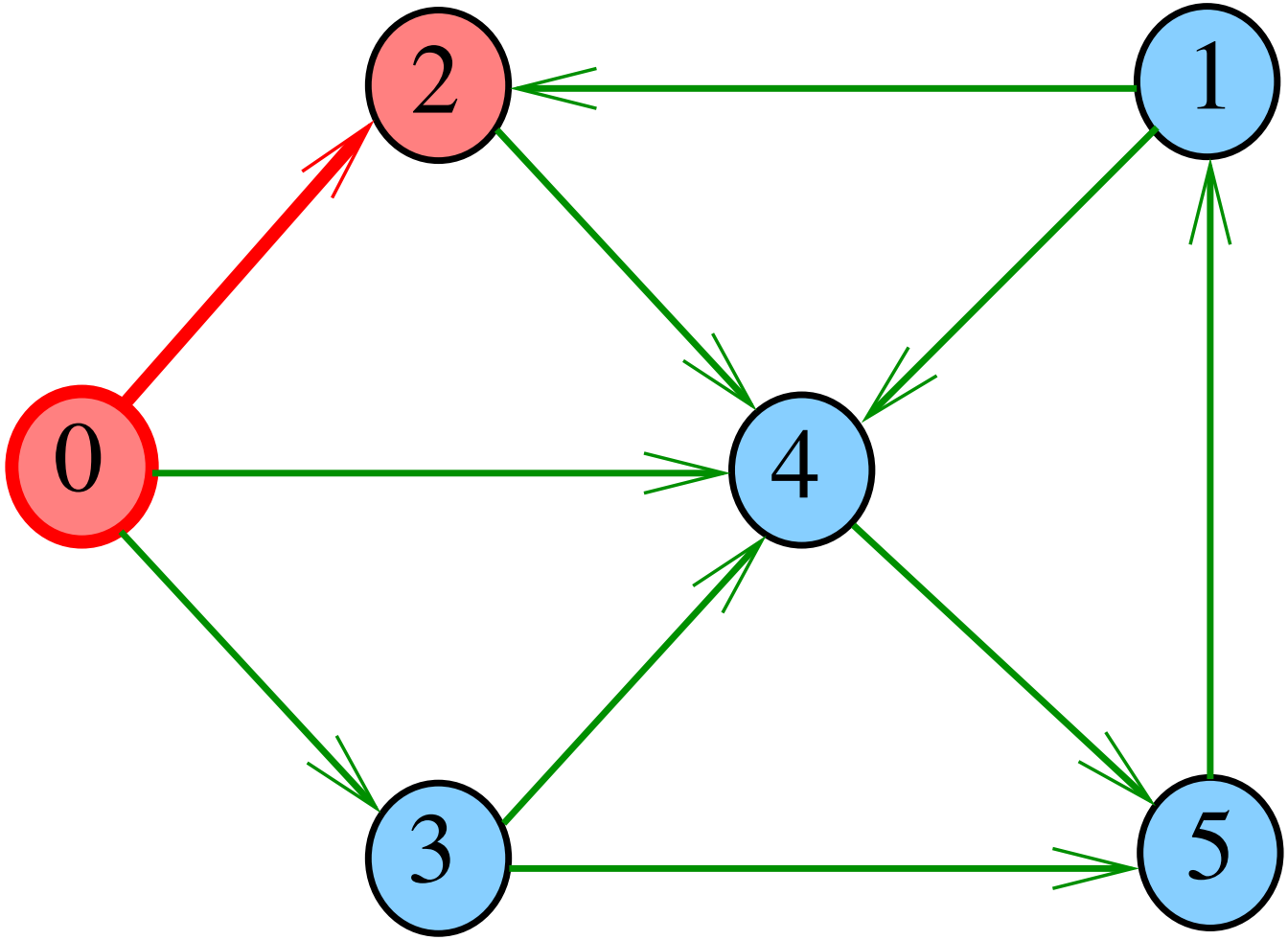
Simulação

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



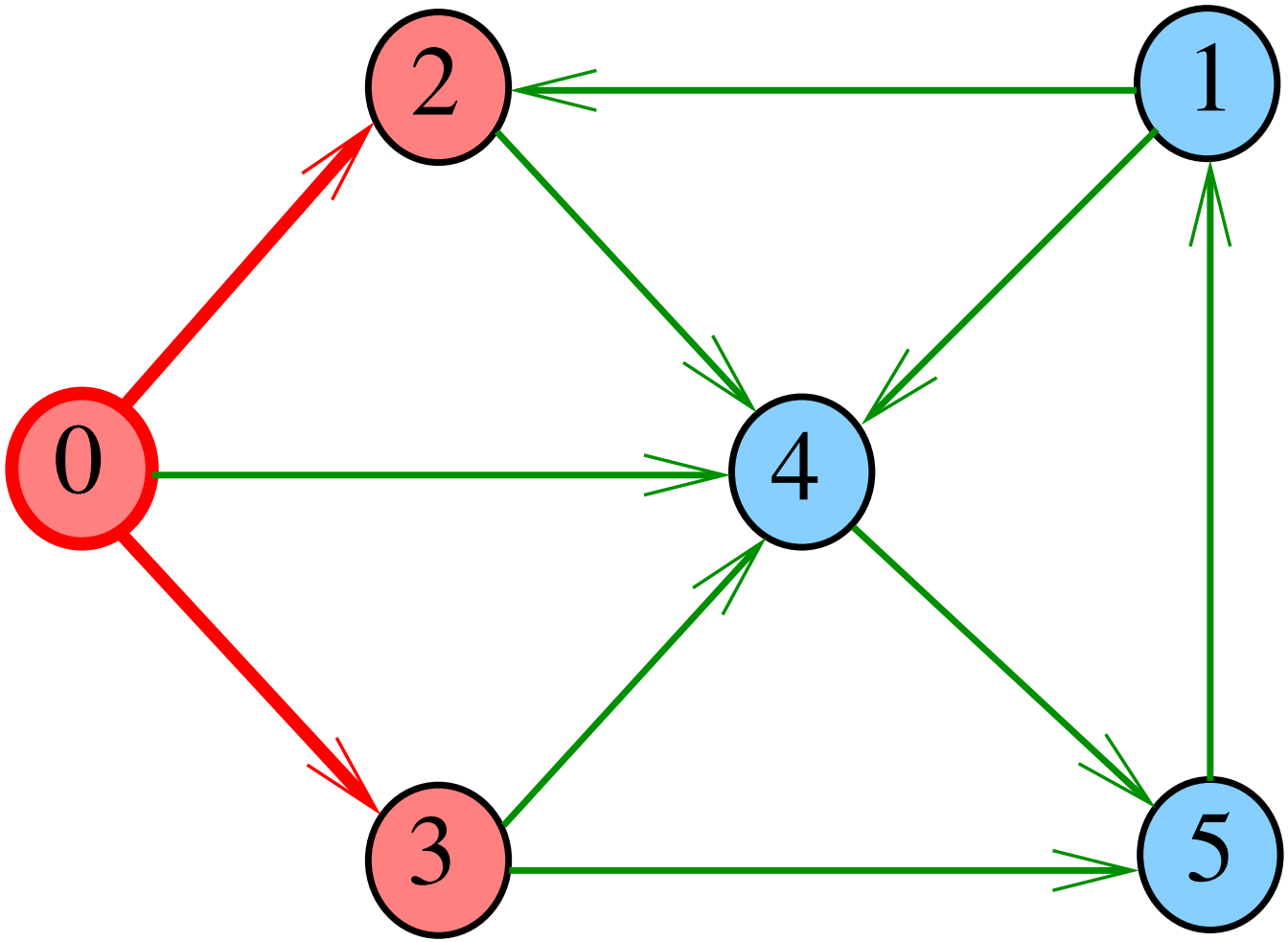
Simulação

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



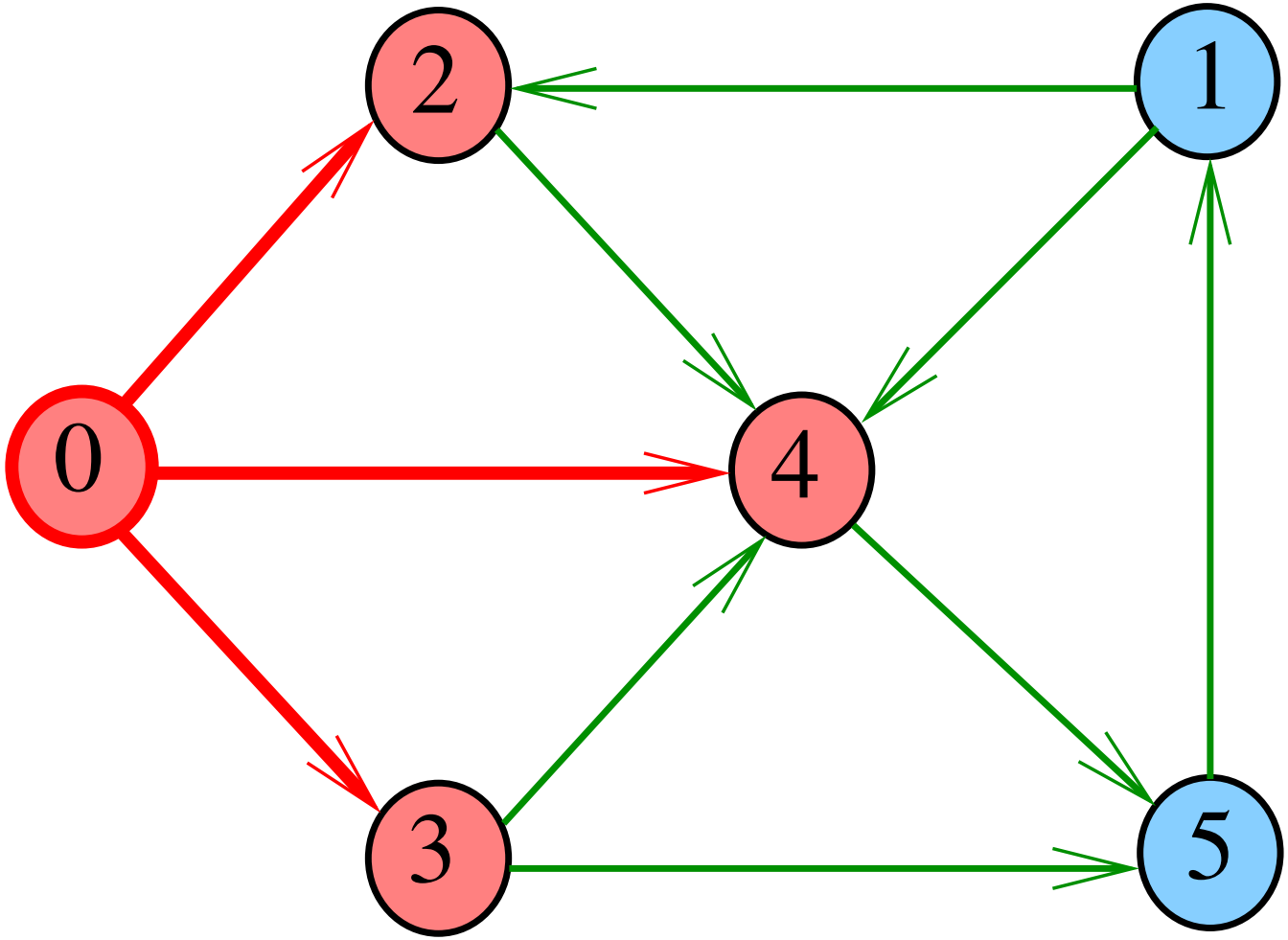
Simulação

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



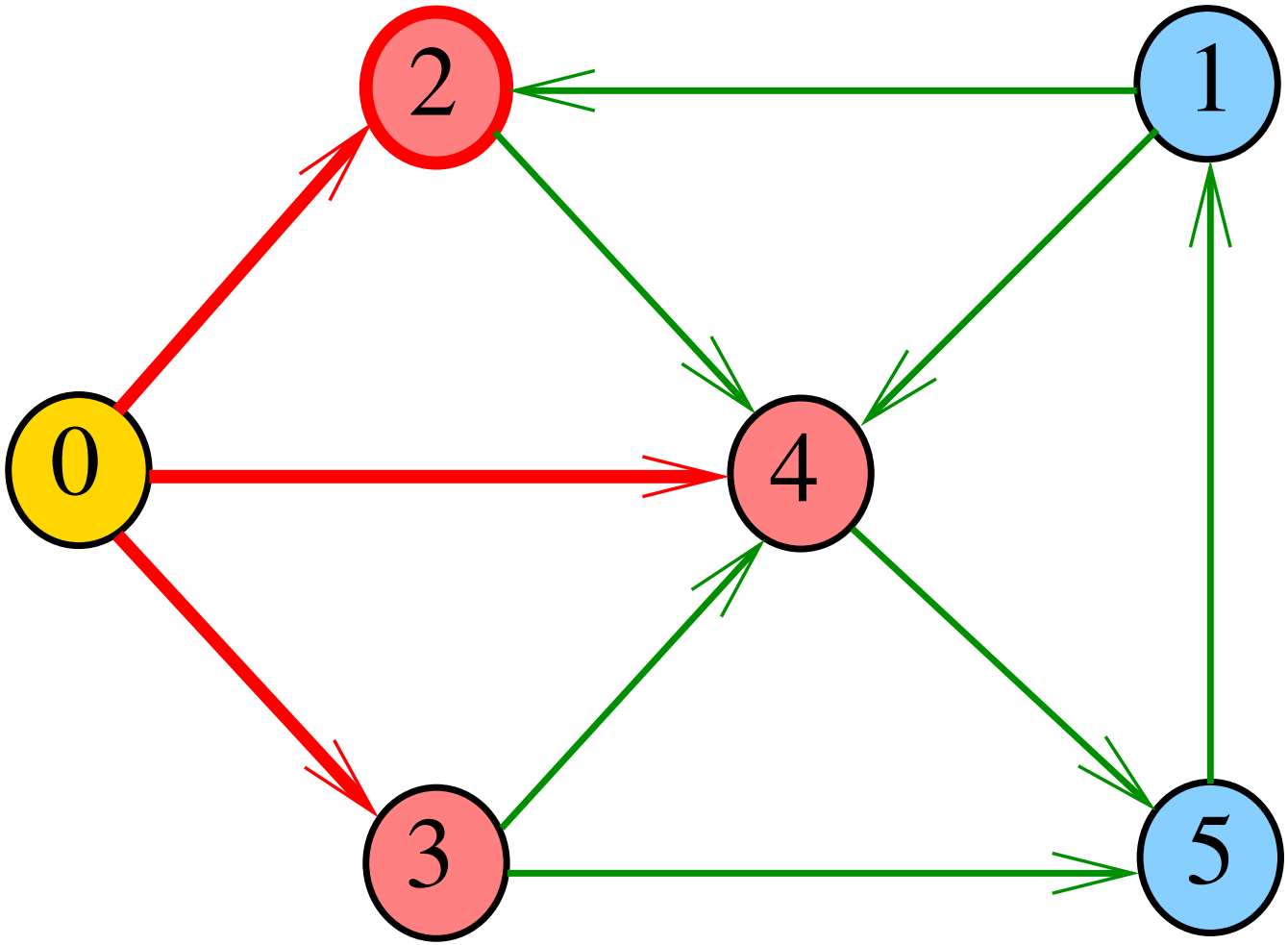
Simulação

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



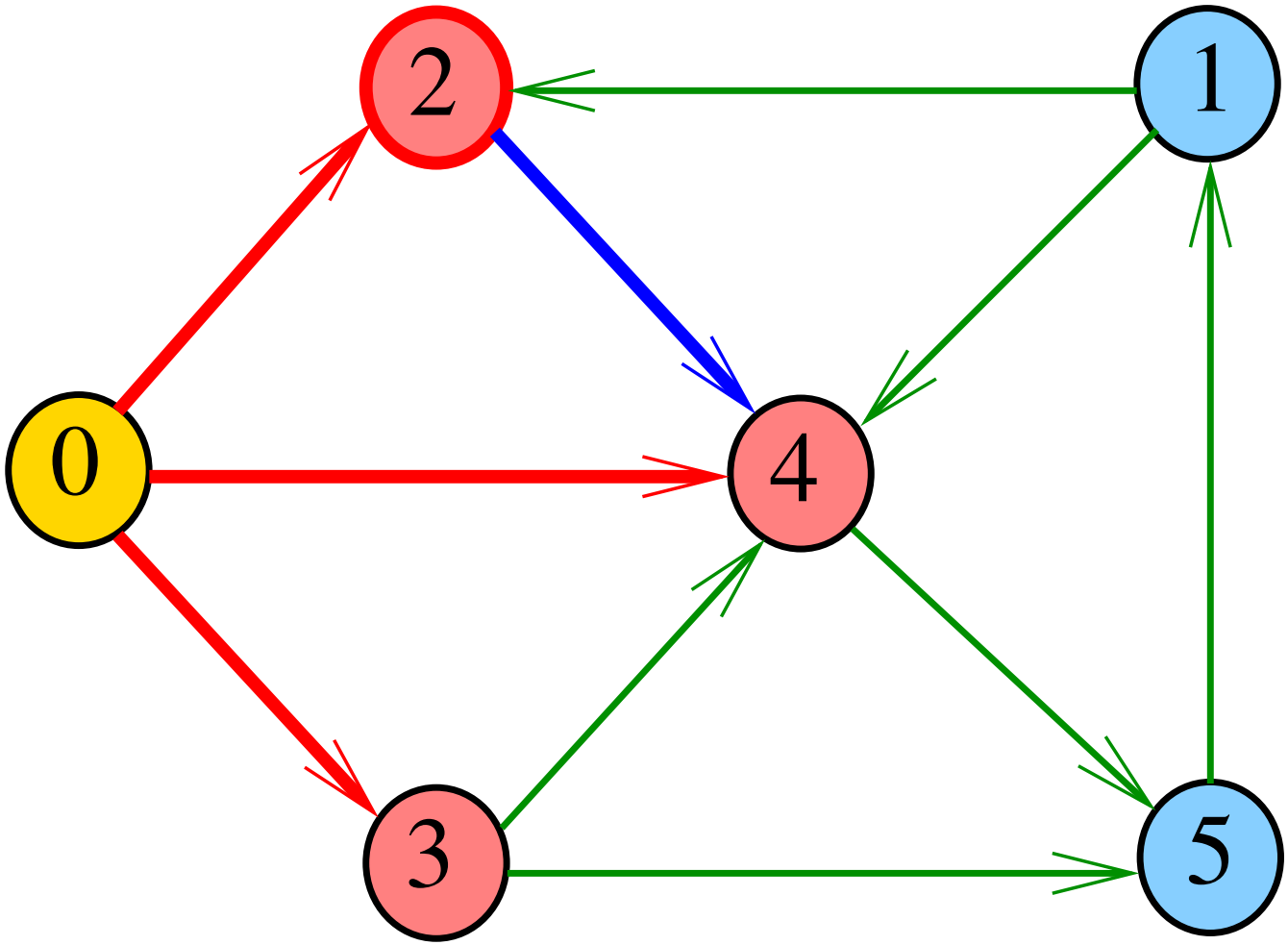
Simulação

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



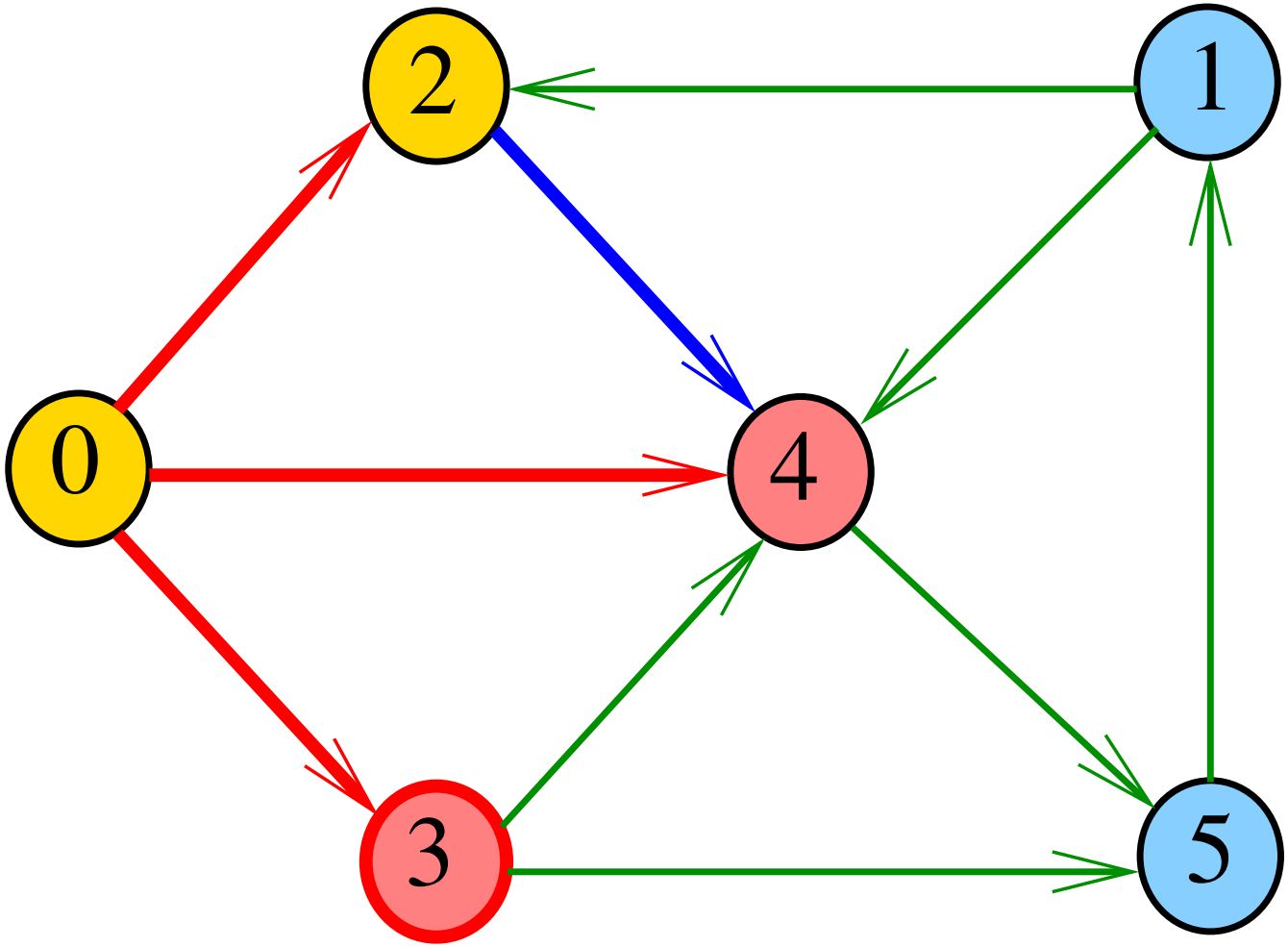
Simulação

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



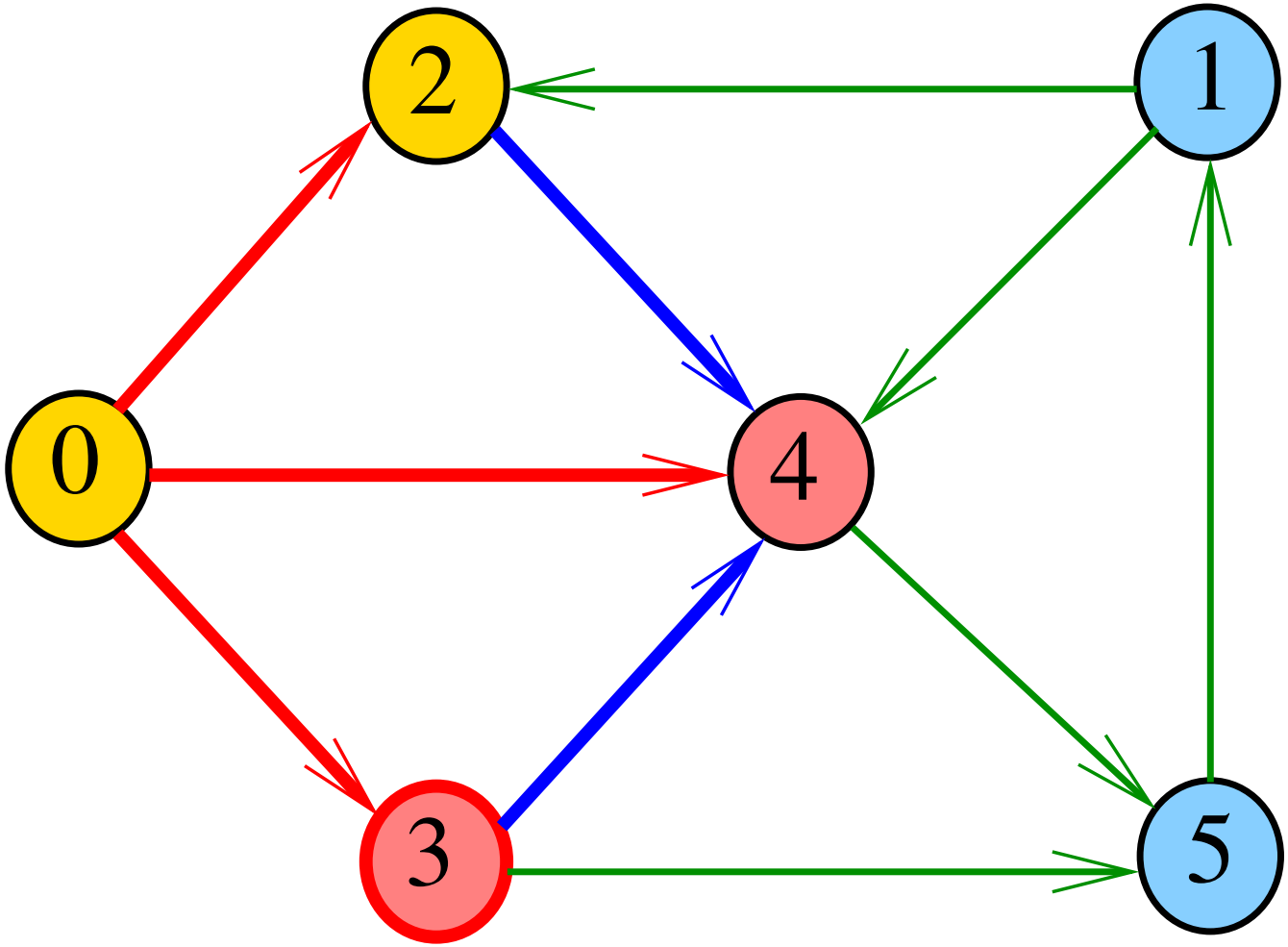
Simulação

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



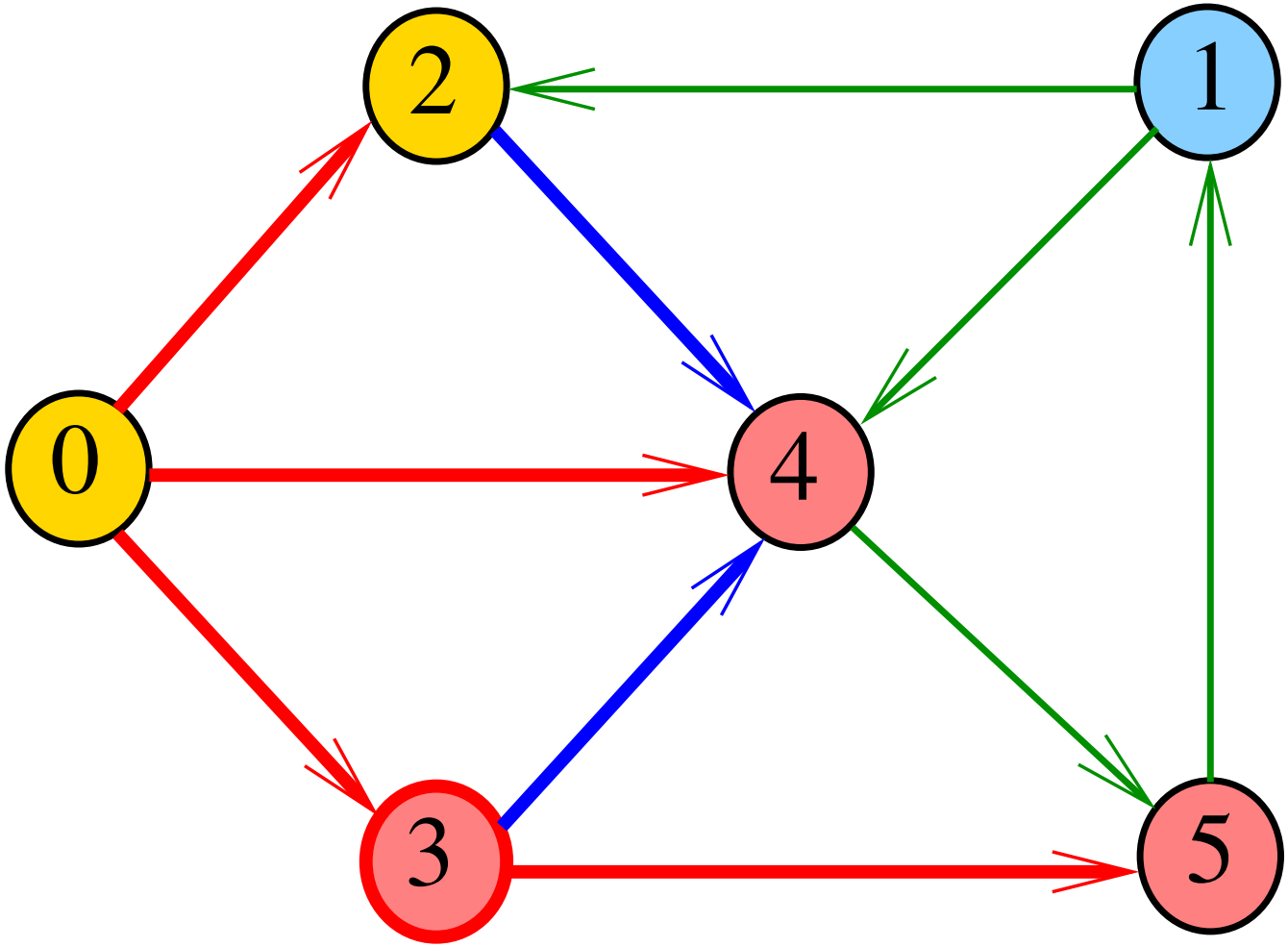
Simulação

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



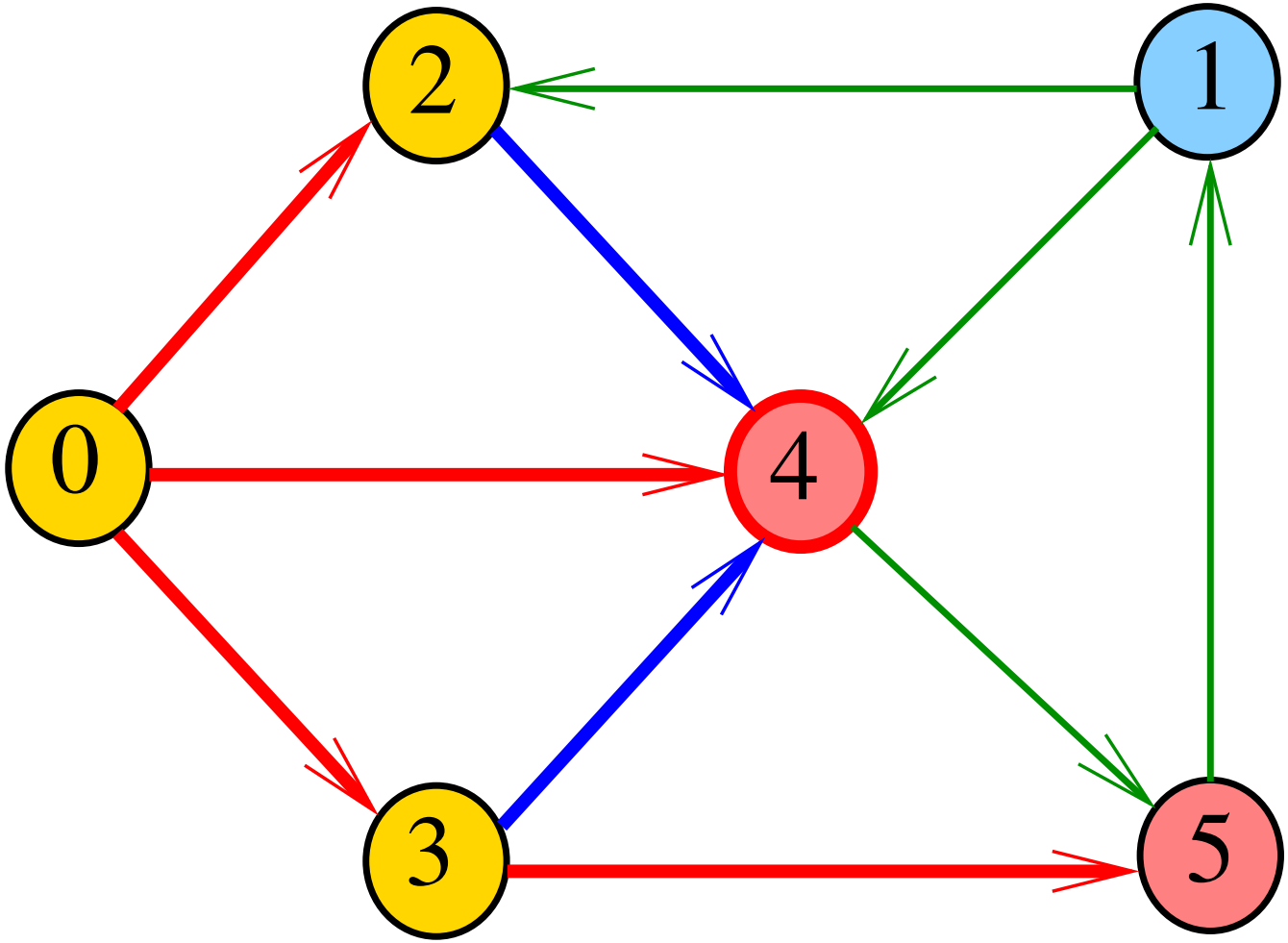
Simulação

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



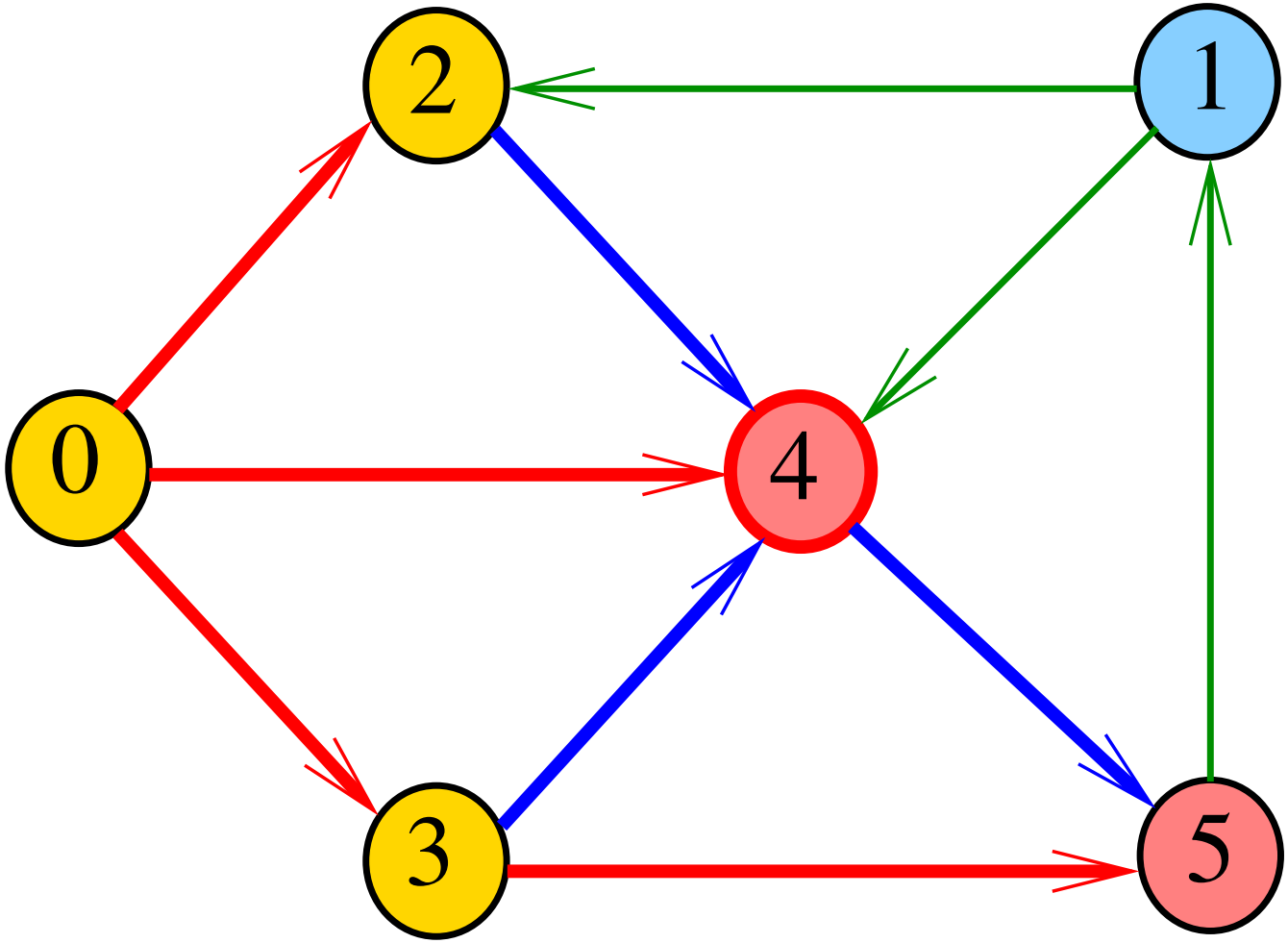
Simulação

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



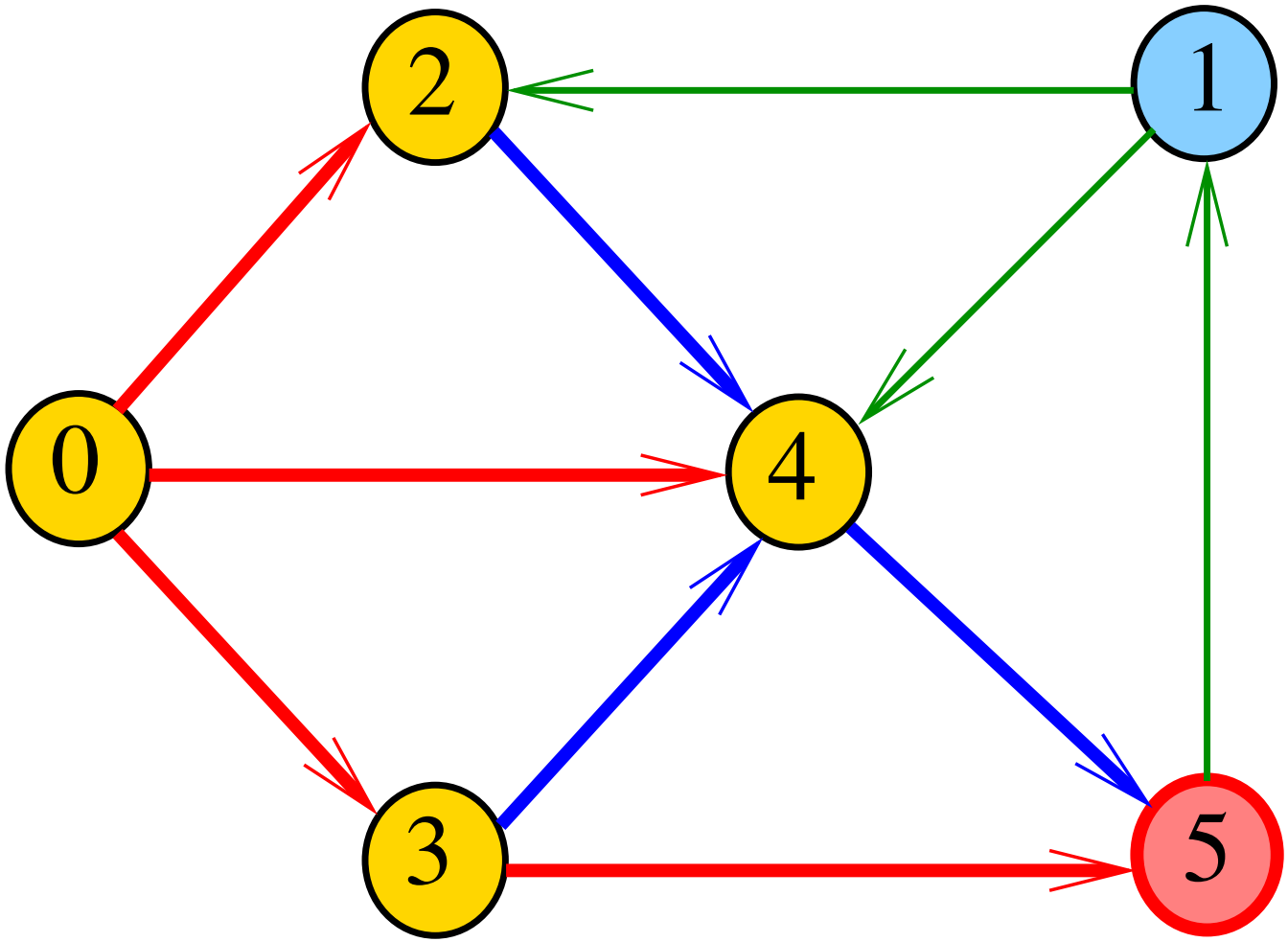
Simulação

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



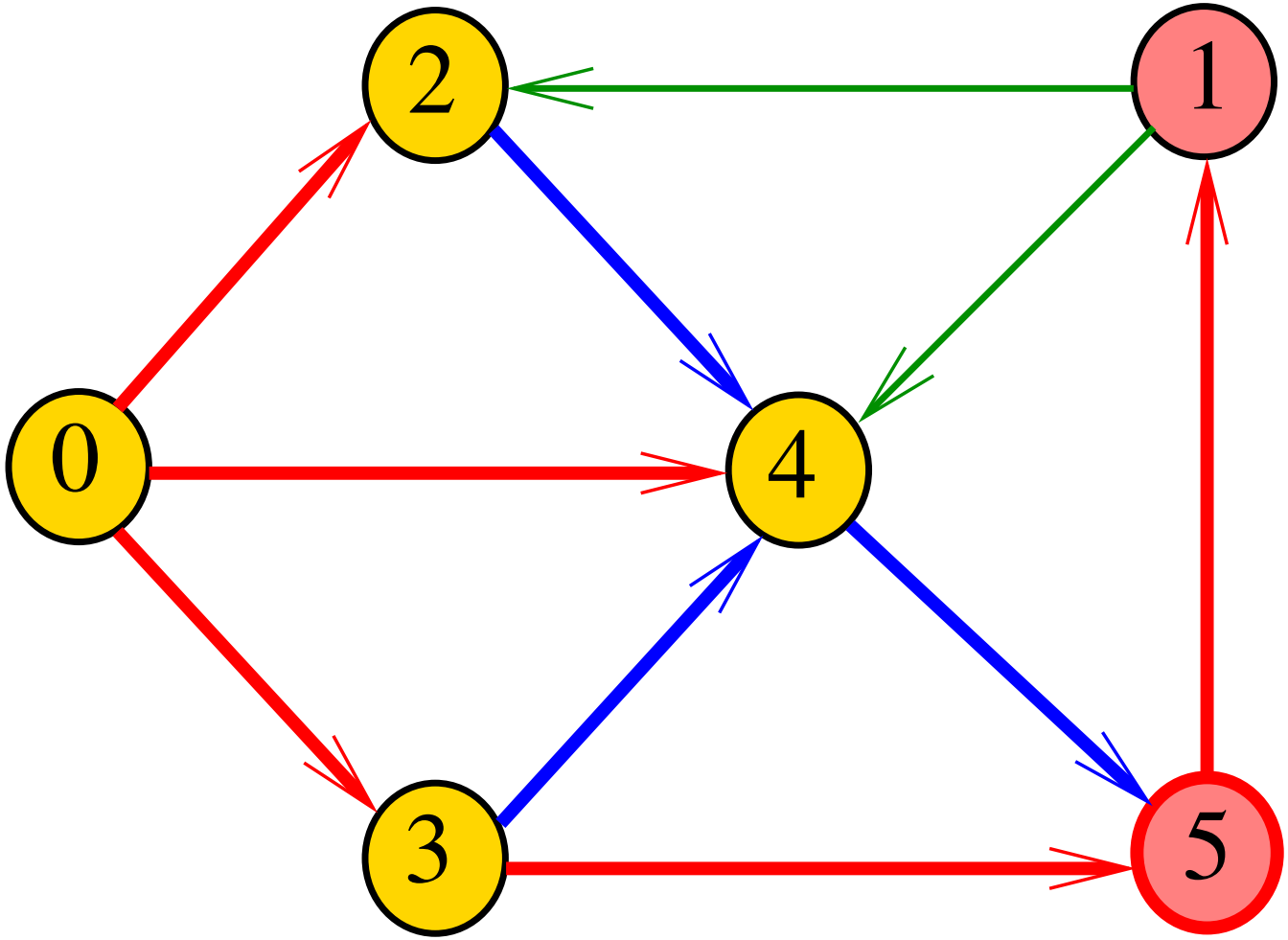
Simulação

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



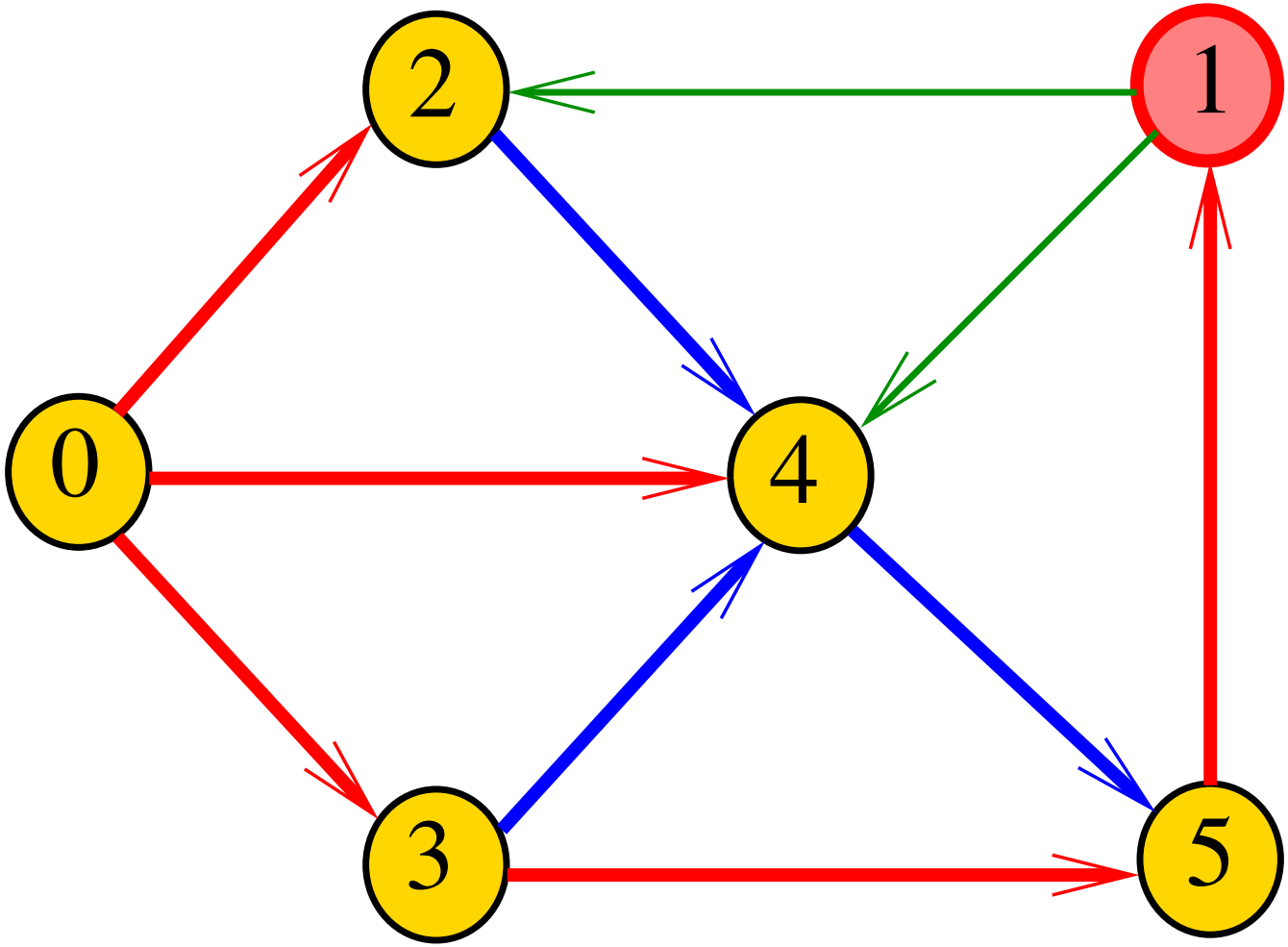
Simulação

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



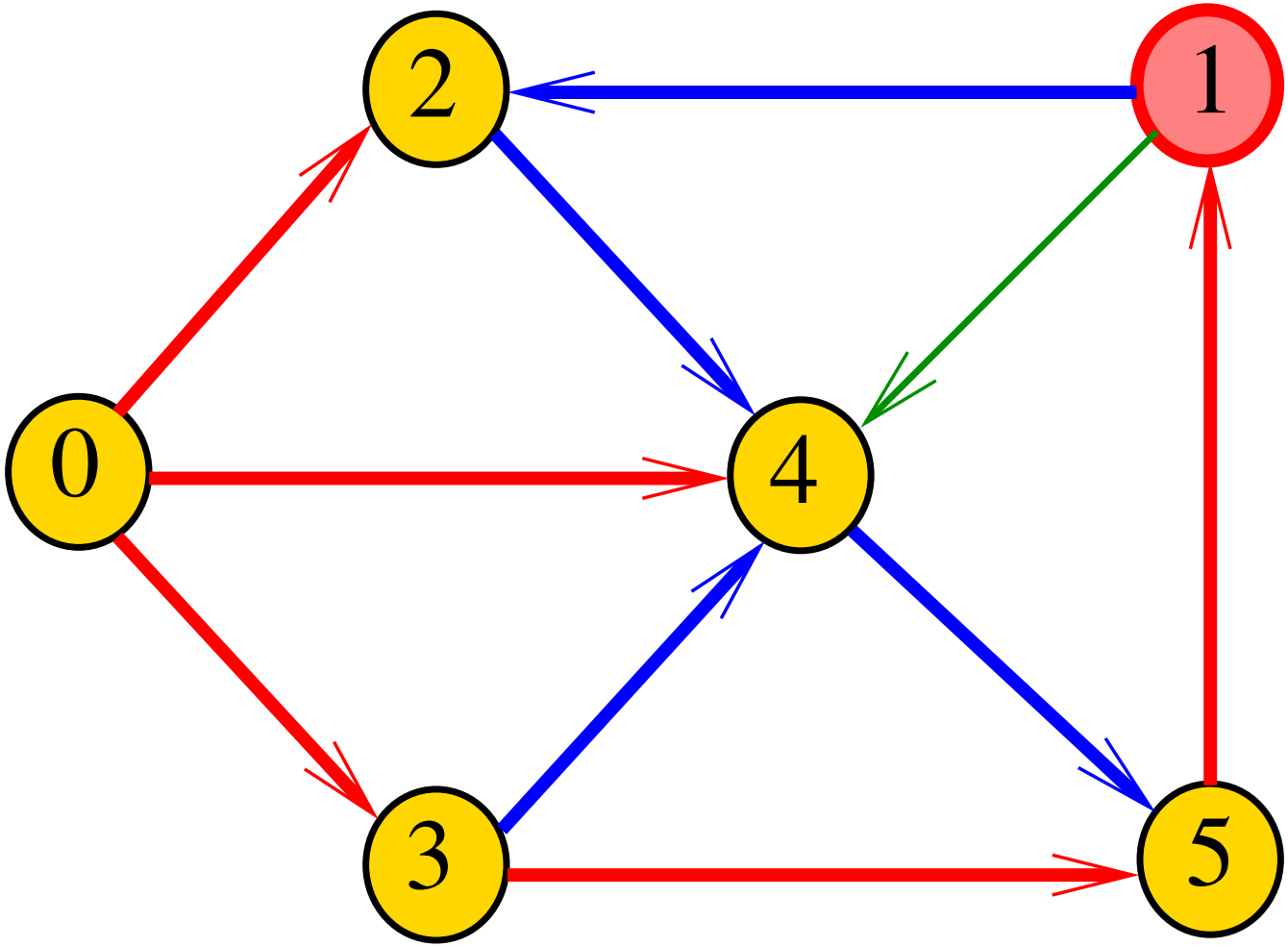
Simulação

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



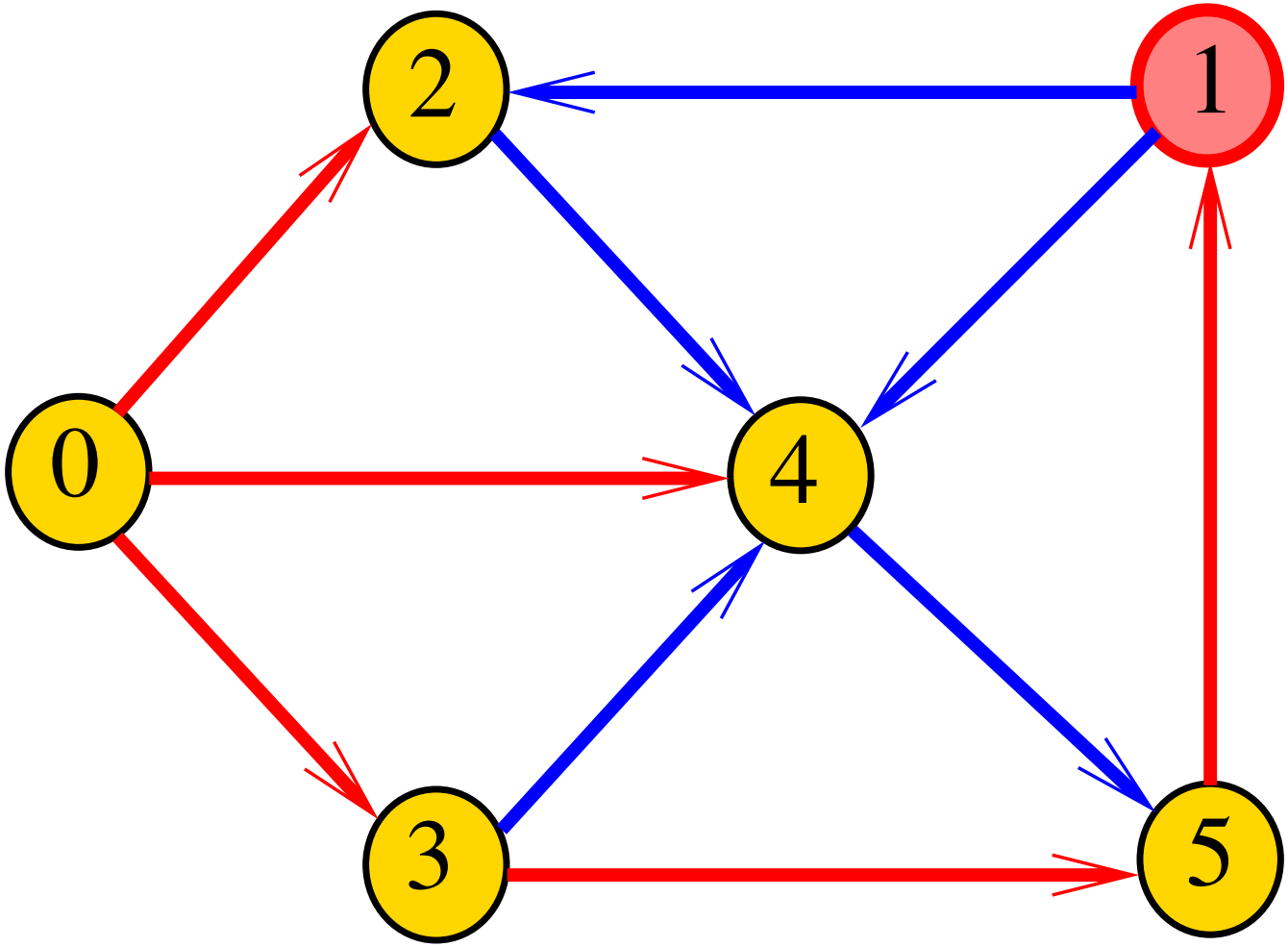
Simulação

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



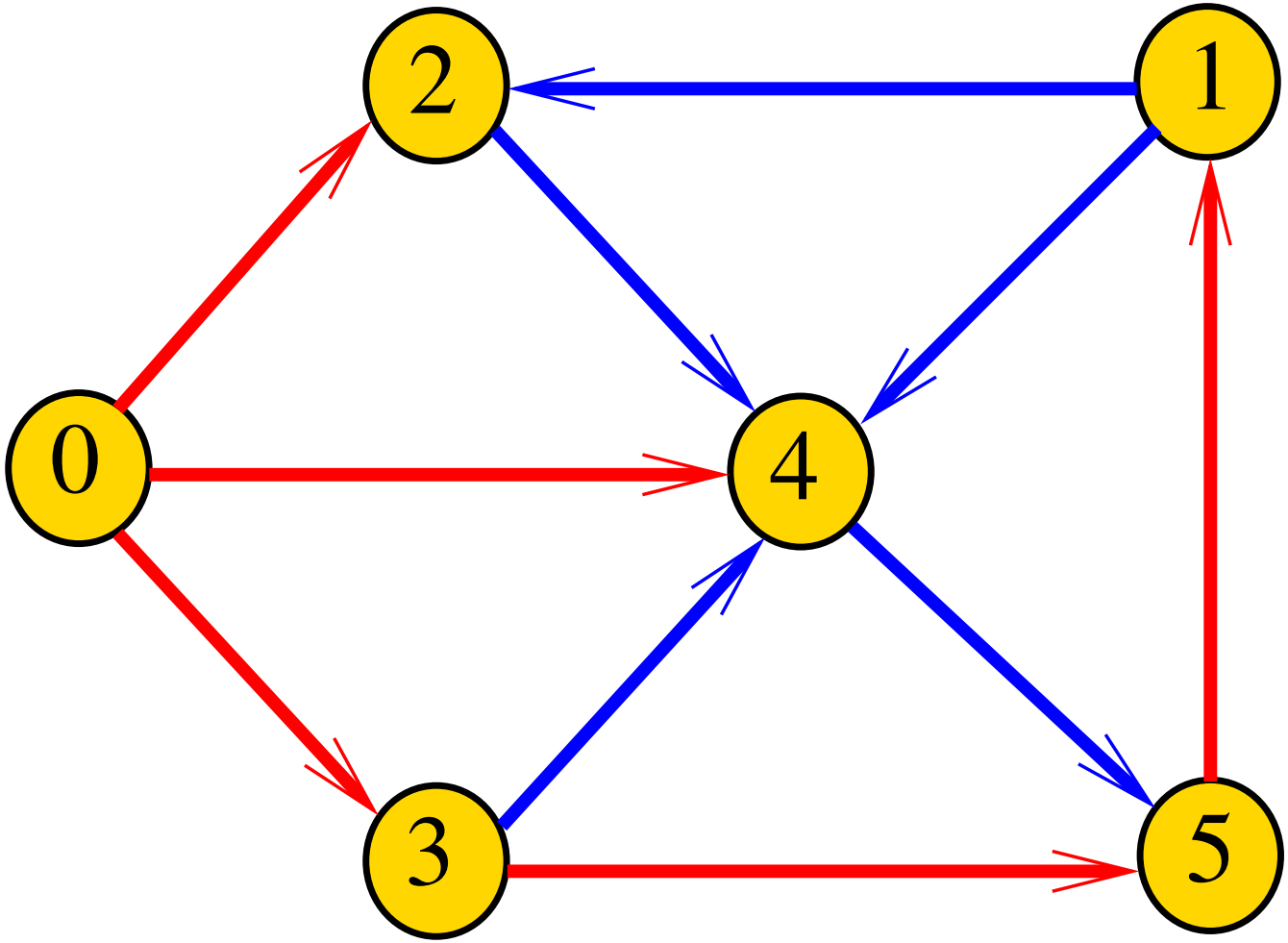
Simulação

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



BreadFirstDirectedPaths

`BreadFirstDirectedPaths` visita todos os vértices do digrafo `G` que podem ser alcançados a partir de `s`

A visita aos vértices é registrada no vetor `marked[]`. Se `v` foi então `marked[v] == true`.

Para isso `BreadFirstDirectedPaths` usa uma fila de vértices

```
Queue<Integer> q = new
Queue<Integer>();
public BreadFirstDirectedPaths (Digraph
G, Vertex s)
```

Relações invariantes

Digamos que um vértice `v` foi **visitado** se

```
marked[v] == true
```

No início de cada iteração vale que

- todo vértice que está na fila já foi visitado;
- se um vértice `v` já foi visitado mas algum de seus vizinhos ainda não foi visitado, então `v` está na fila.

Cada vértice entra na fila no máximo uma vez.

Portanto, basta que a fila tenha espaço suficiente para `G.V()` vértices.

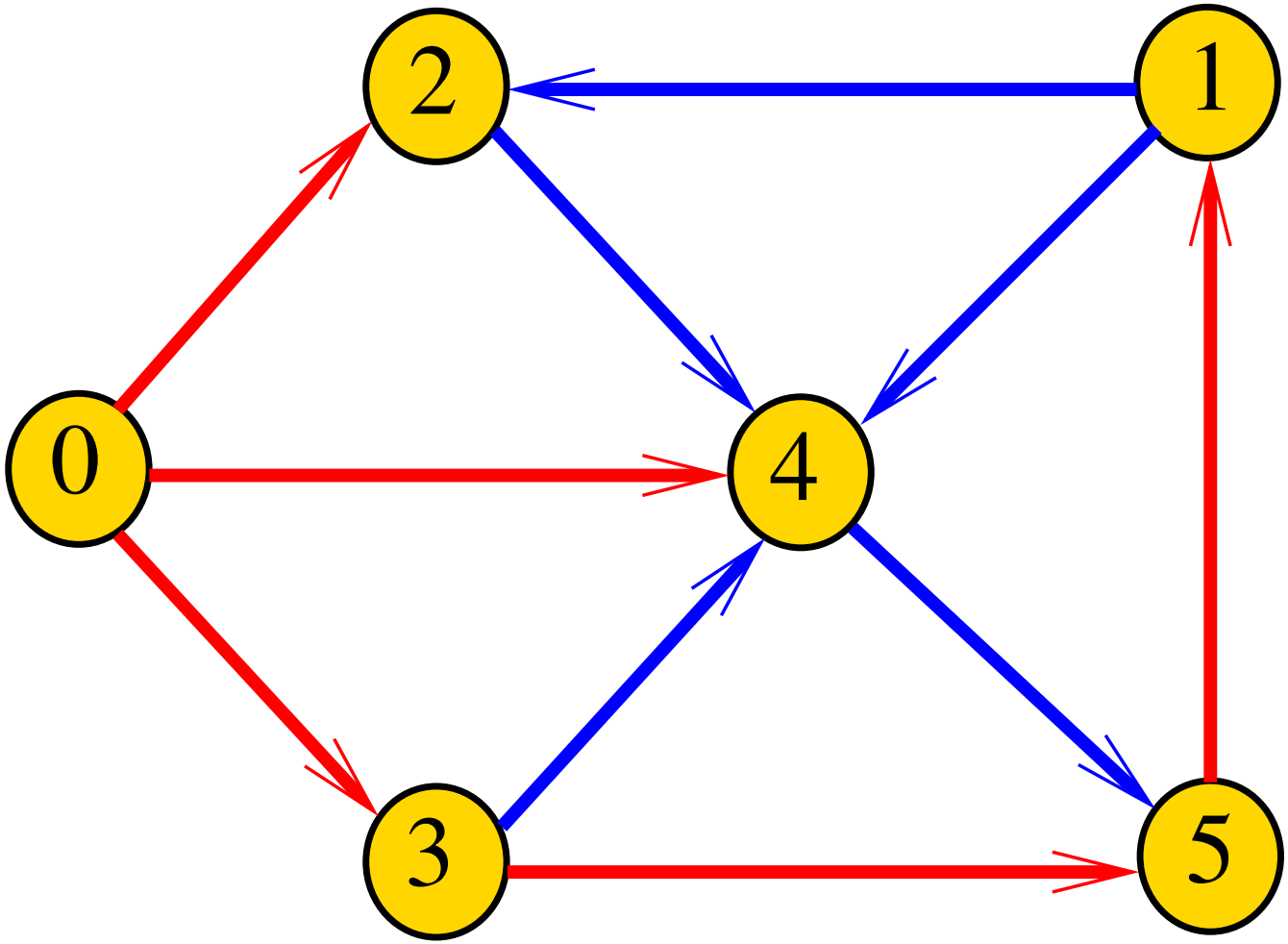
Consumo de tempo

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

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

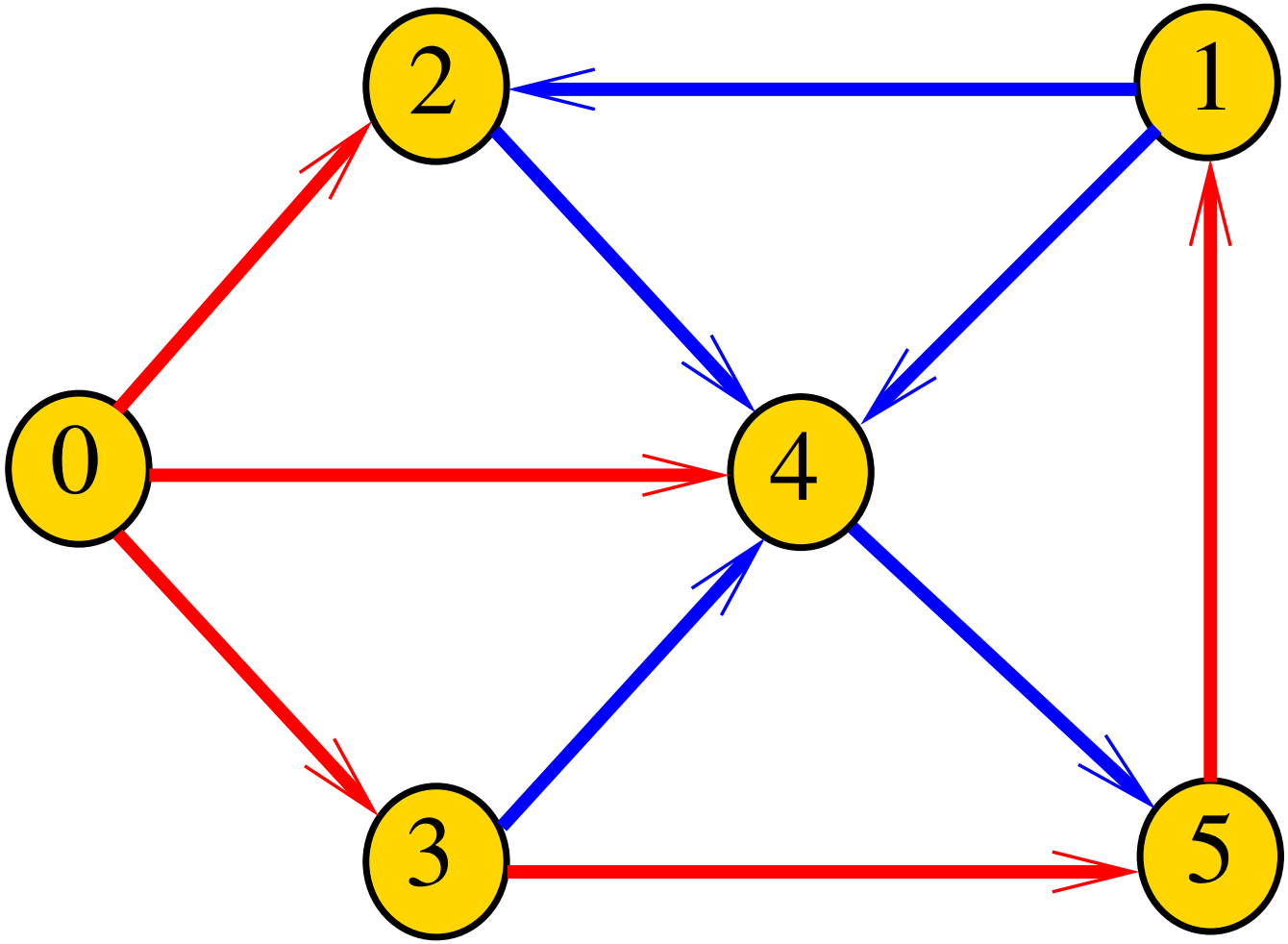
Arborescência da BFS

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



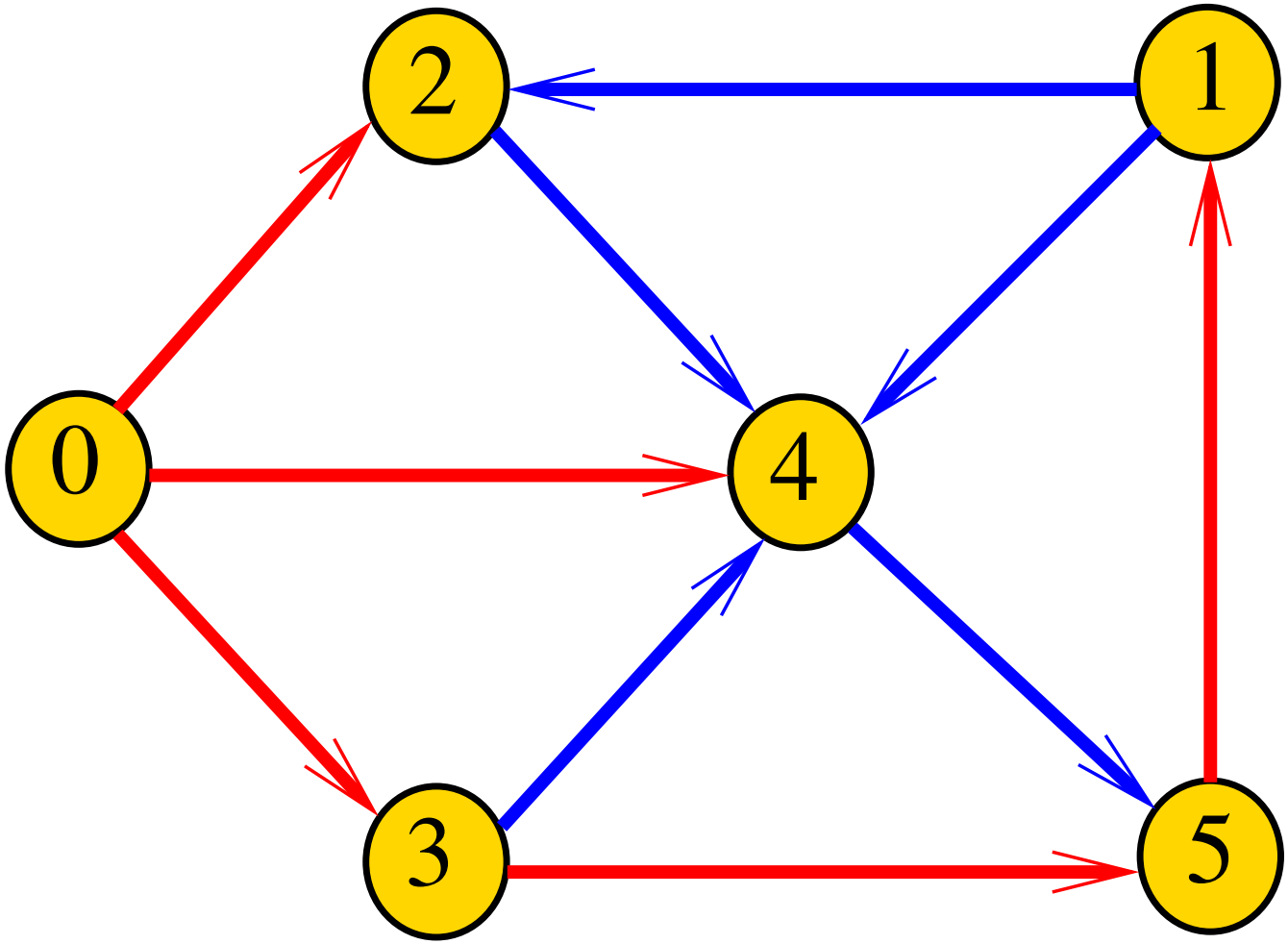
Arborescência da BFS

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



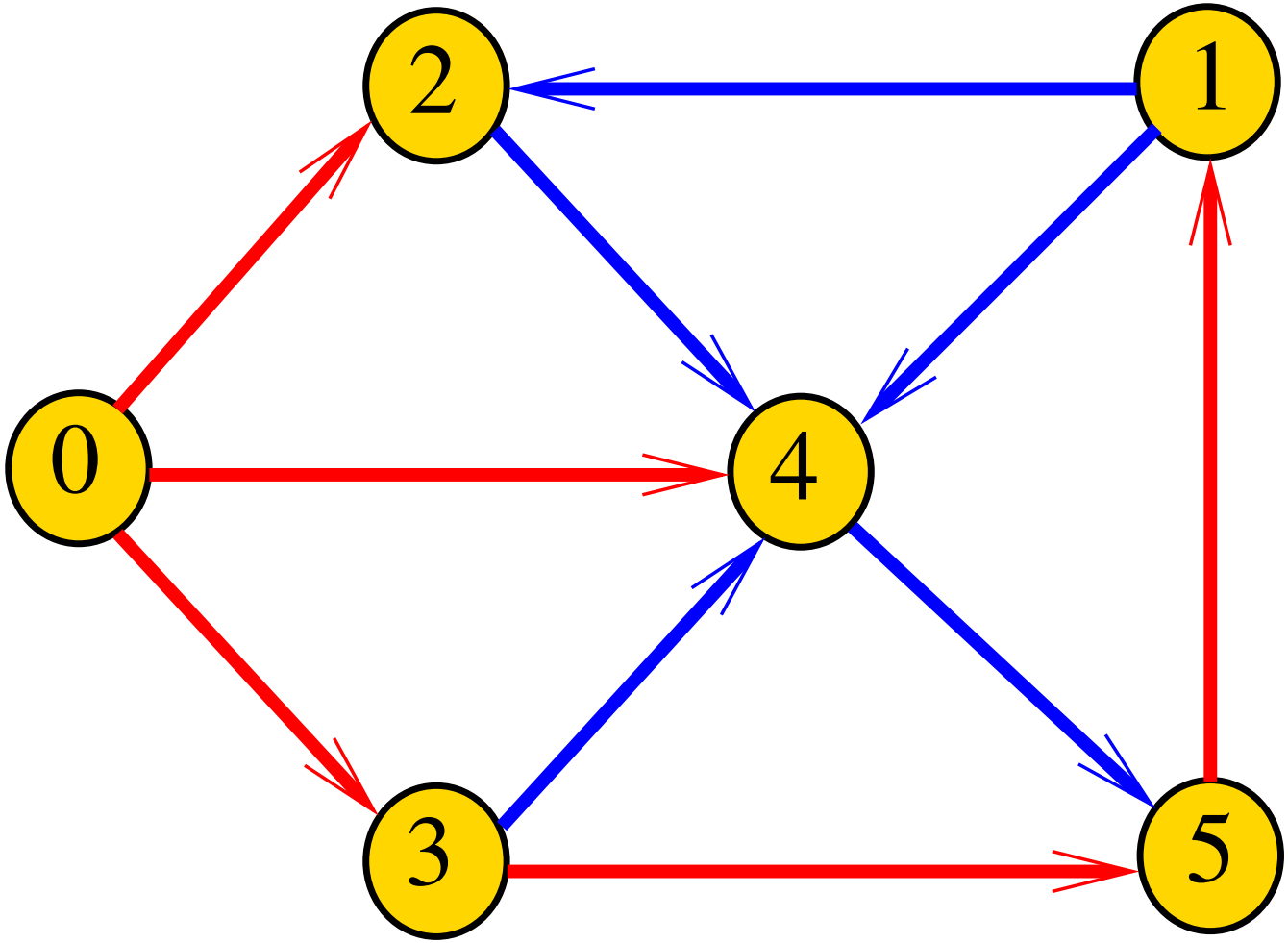
Representação da BFS

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



BFS versus DFS

- busca em largura usa **fila**, busca em profundidade usa **pilha**
- a busca em largura é descrita em **estilo iterativo**, enquanto a busca em profundidade é descrita, usualmente, em **estilo recursivo**
- busca em largura começa tipicamente num vértice especificado, a busca em profundidade, o próprio algoritmo escolhe o vértice inicial
- a busca em largura visita apenas os vértices que podem ser atingidos a partir do vértice inicial, a busca em profundidade visita, tipicamente, todos os vértices do digrafo

Caminhos mínimos

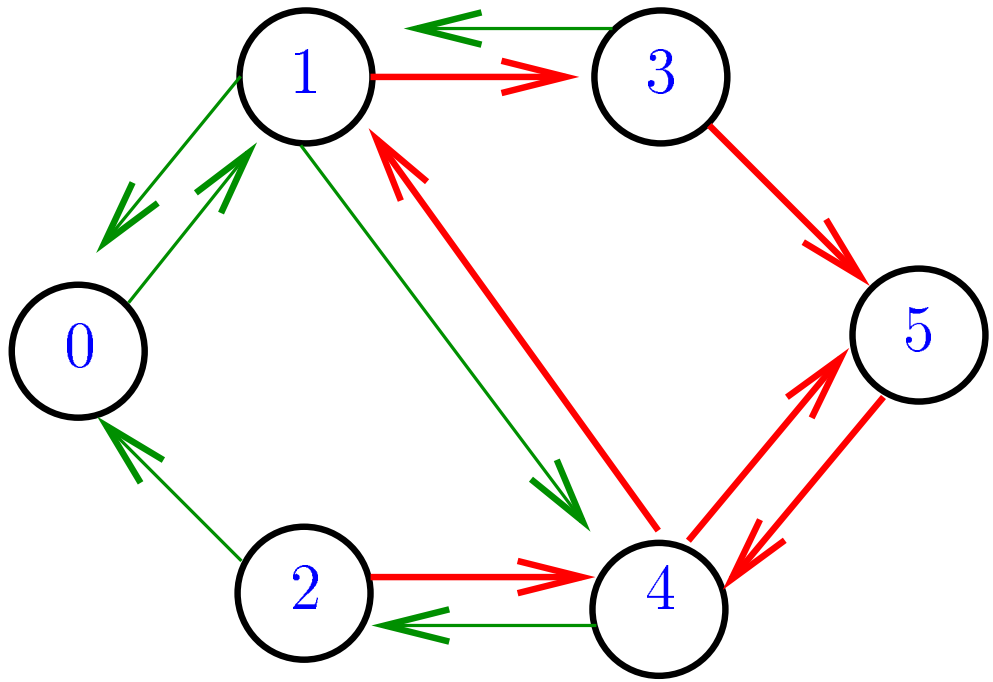
Fonte: http://vandanasanju.blogspot.com.br/

S 18.7

Comprimento

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

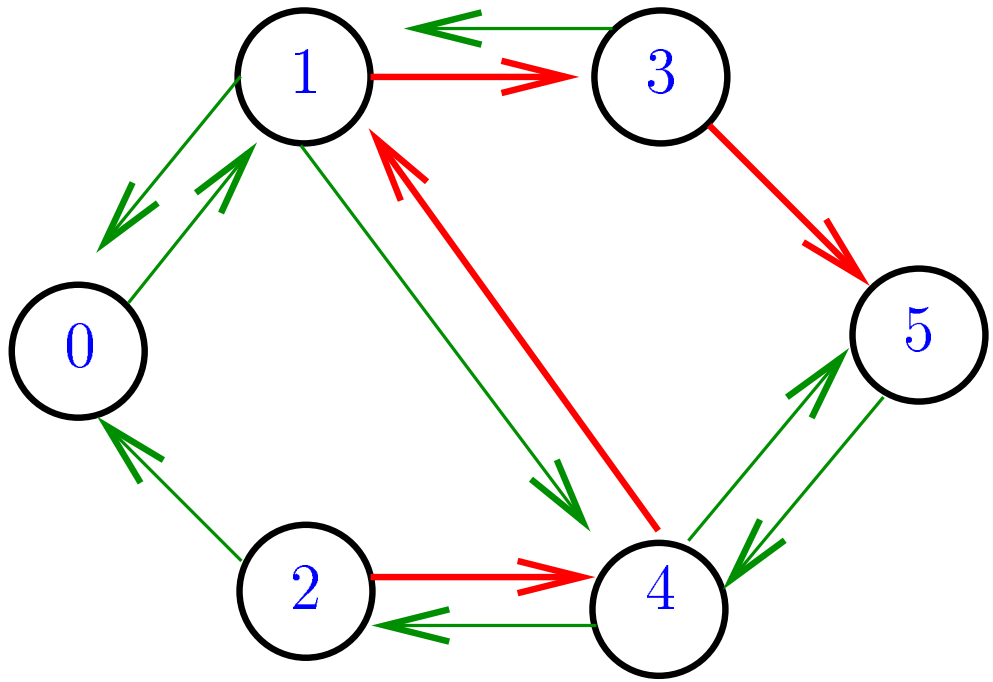
Exemplo: 2-4-1-3-5-4-5 tem comprimento 6



Comprimento

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

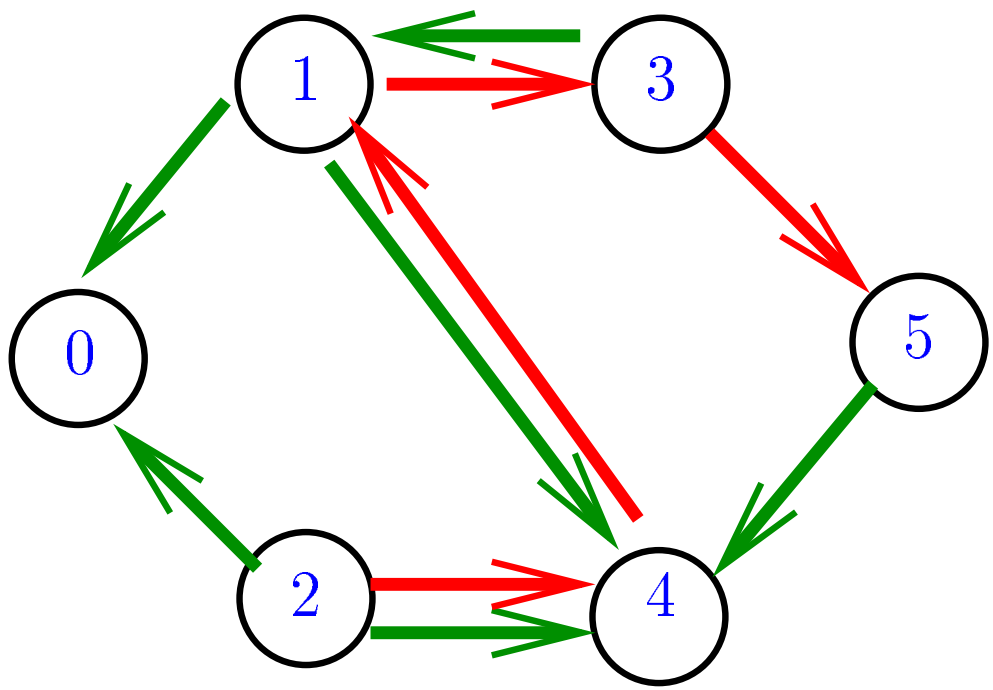
Exemplo: 2-4-1-3-5 tem comprimento 4



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**

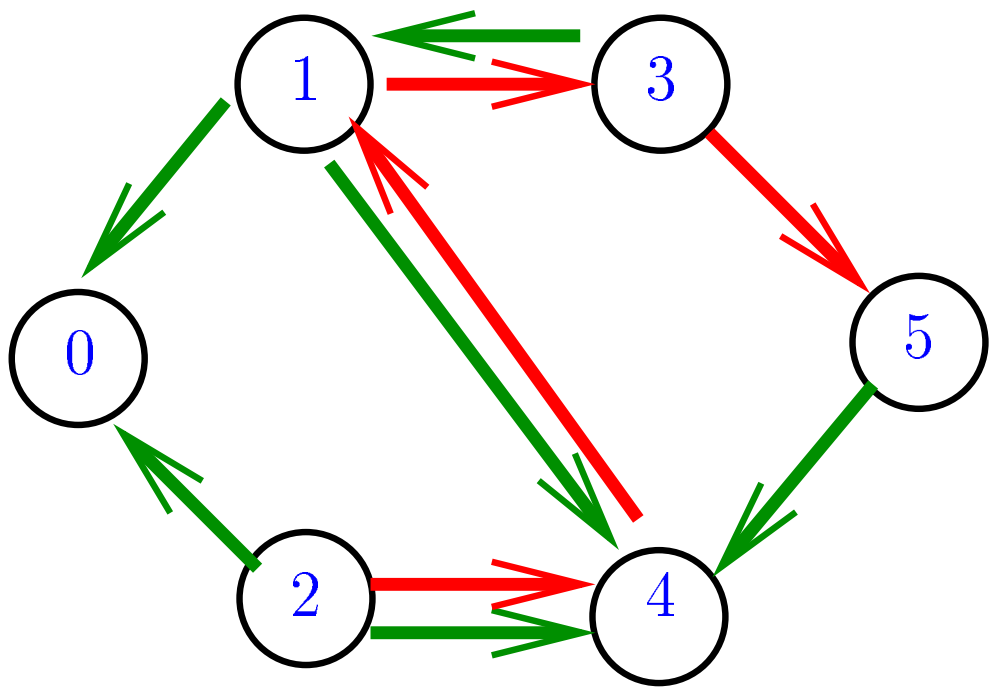
Exemplo: a distância de 2 a 5 é 4



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

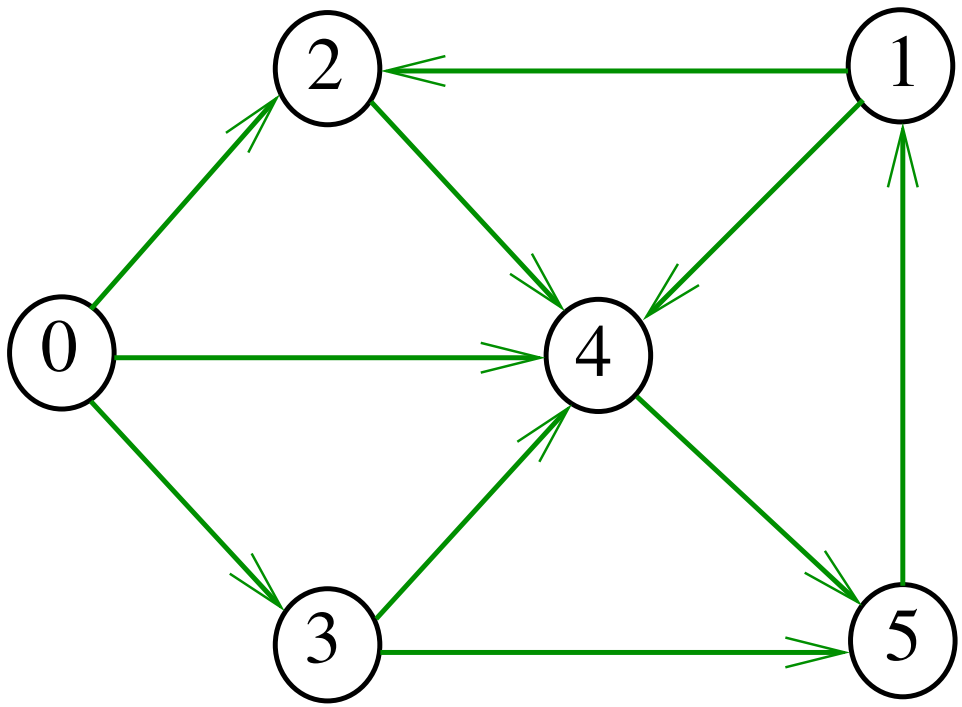


Calculando distâncias

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

Exemplo: para `s` = 0

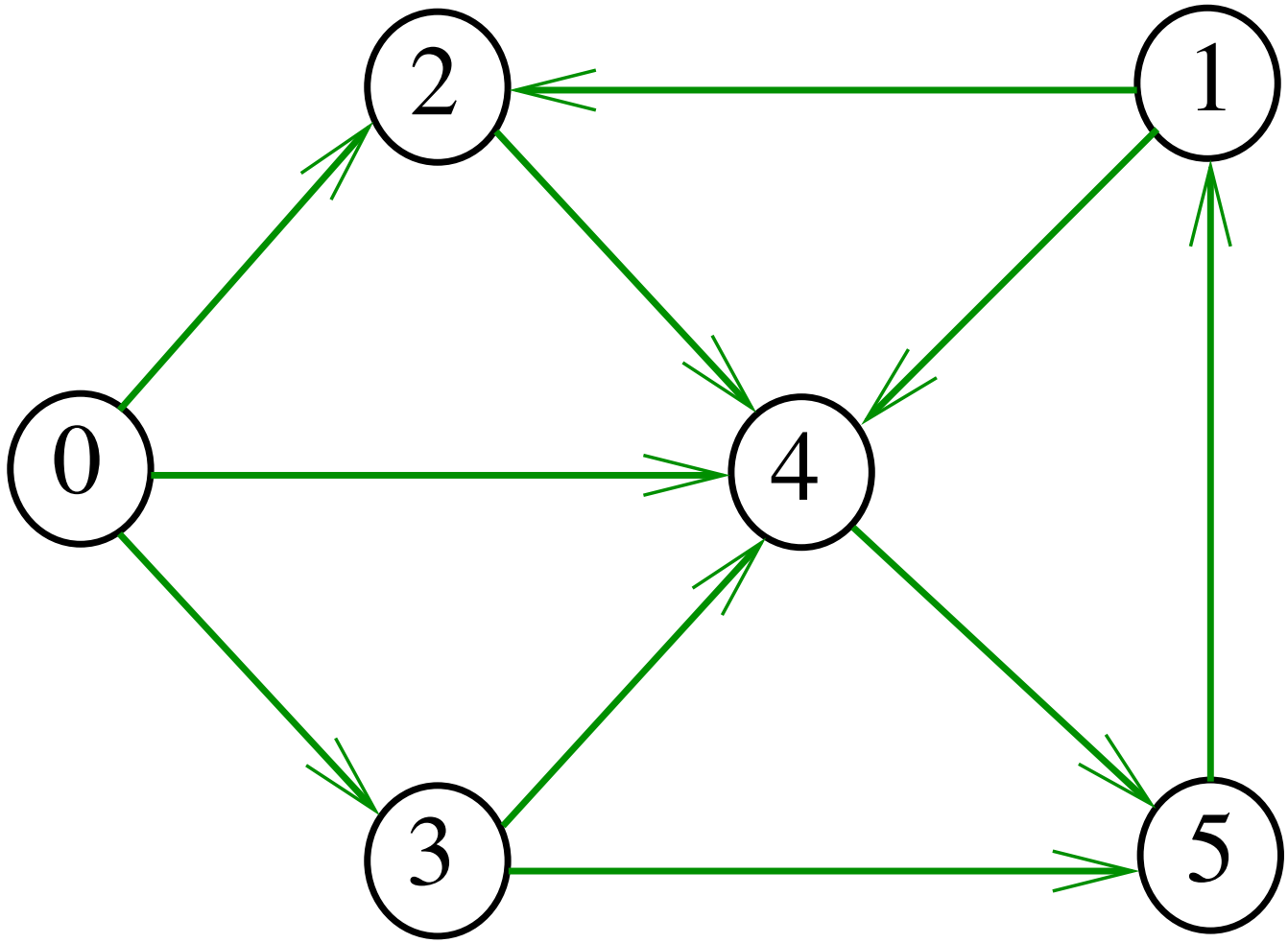
`v`	0	1	2	3	4	5
`distTo[v]`	0	3	1	1	1	2



Simulação

i	0	1	2	3	4	5
q[i]						

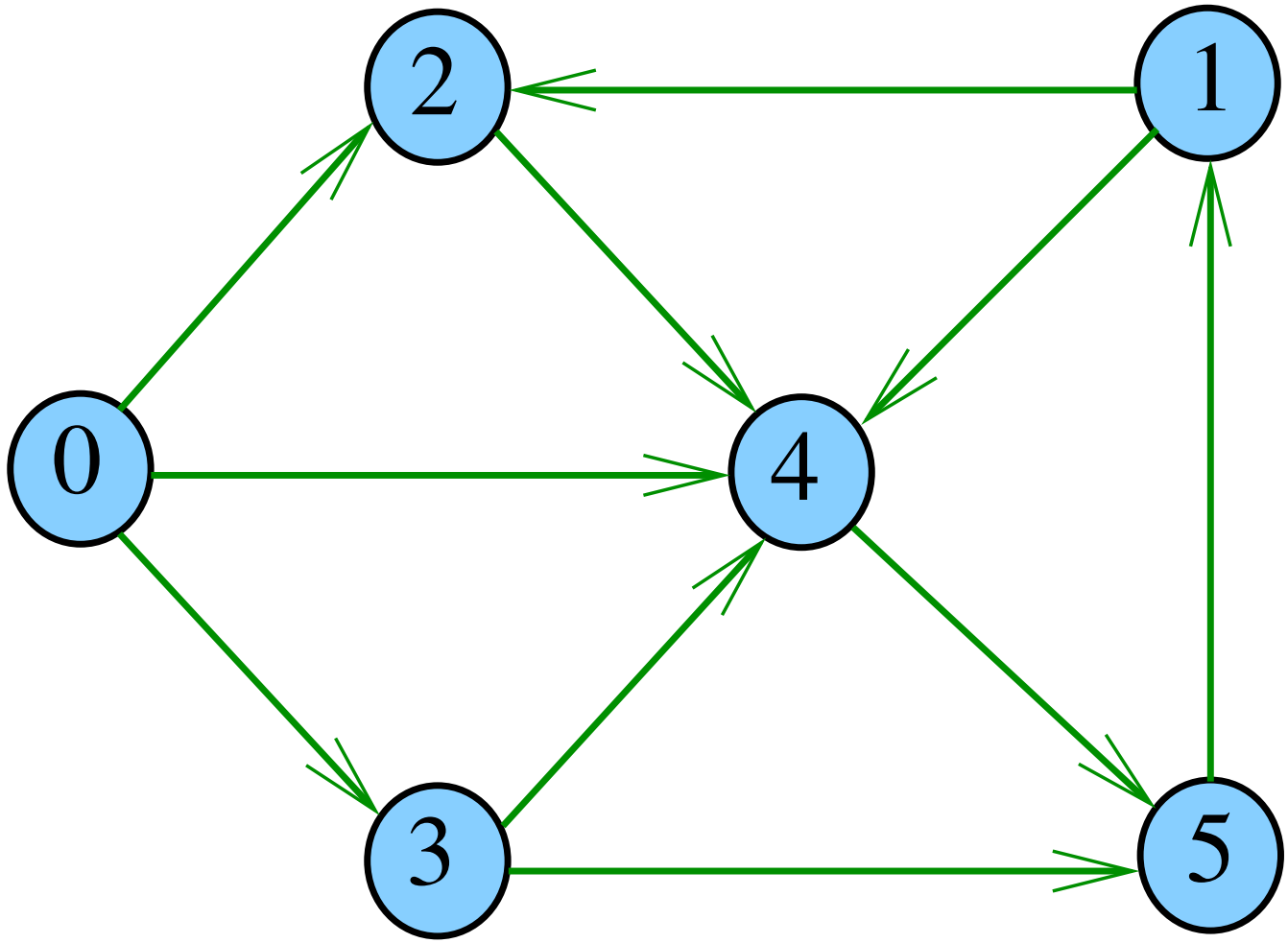
v	0	1	2	3	4	5
distTo[v]						



Simulação

i	0	1	2	3	4	5
q[i]						

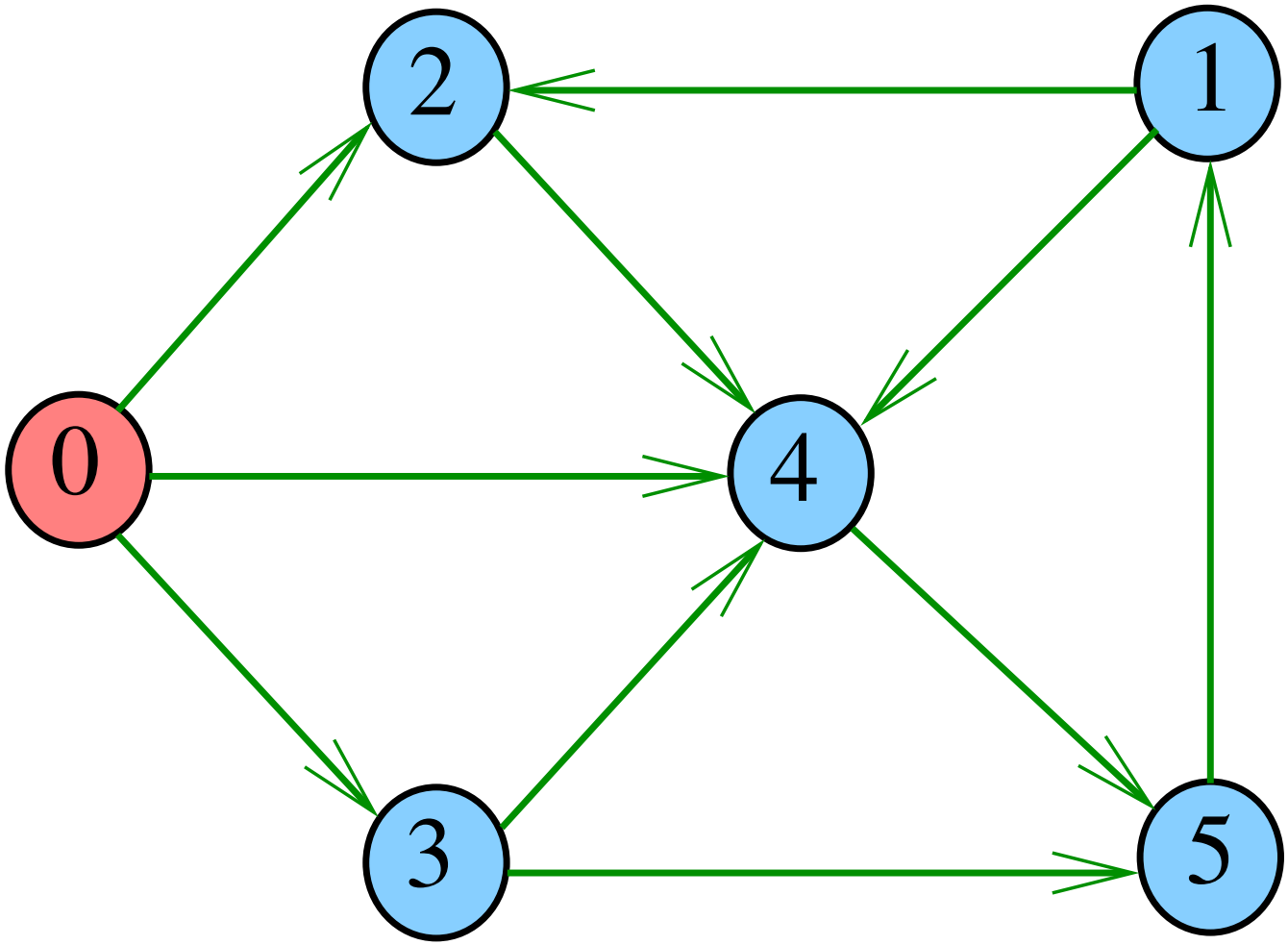
v	0	1	2	3	4	5
distTo[v]	6	6	6	6	6	6



Simulação

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

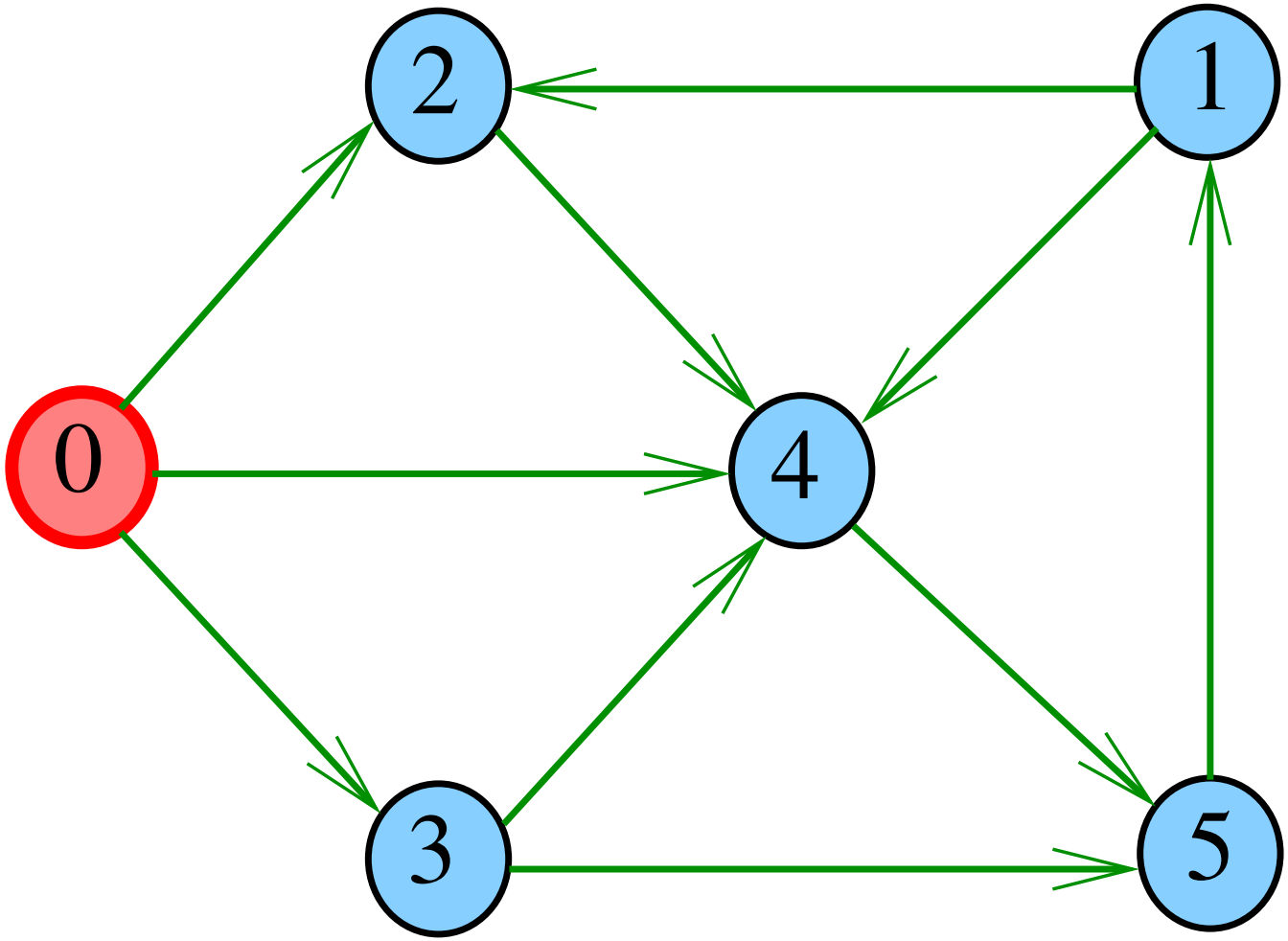
v	0	1	2	3	4	5
distTo[v]	6	6	6	6	6	6



Simulação

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

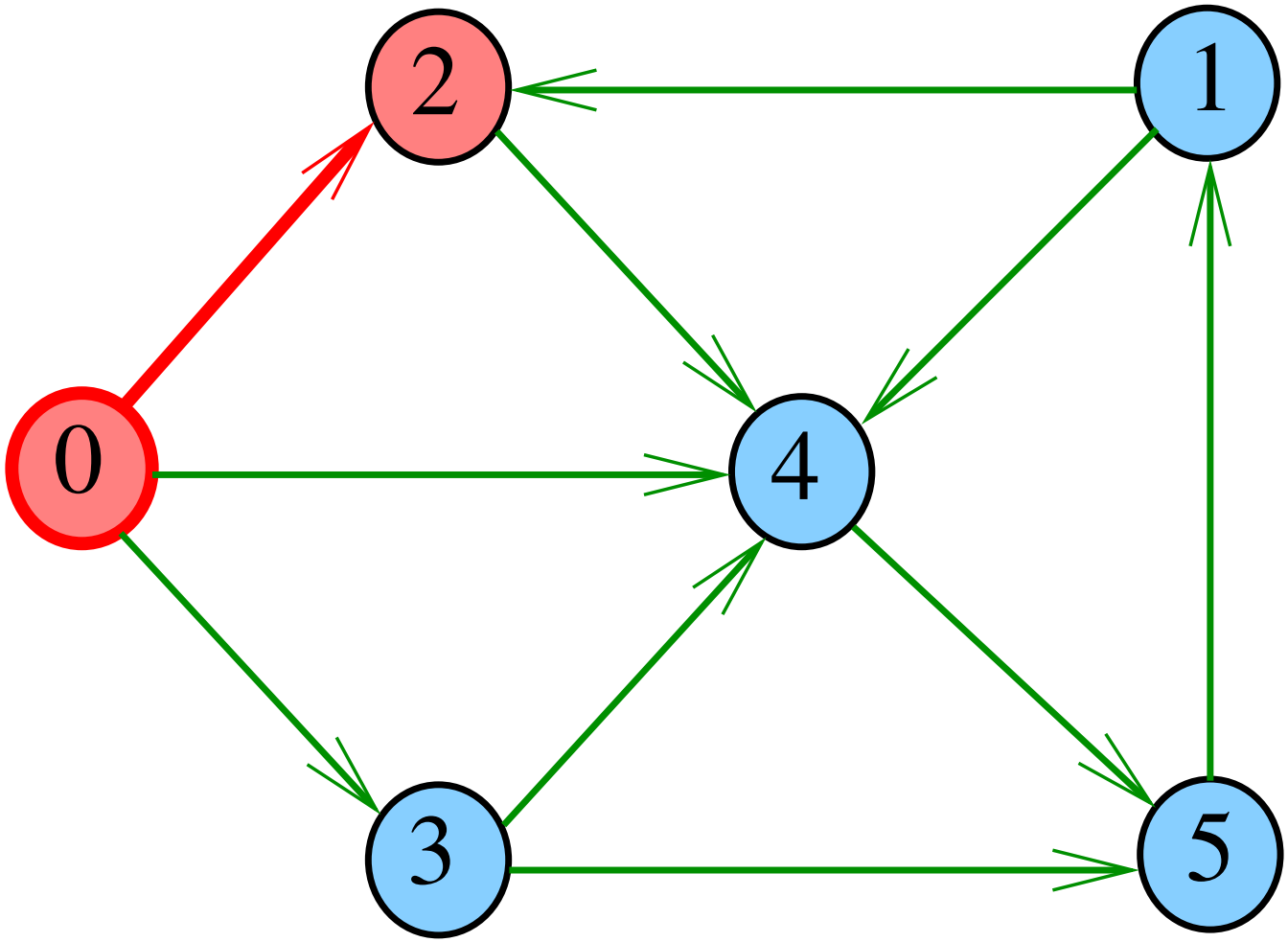
v	0	1	2	3	4	5
distTo[v]	**0**	6	6	6	6	6



Simulação

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

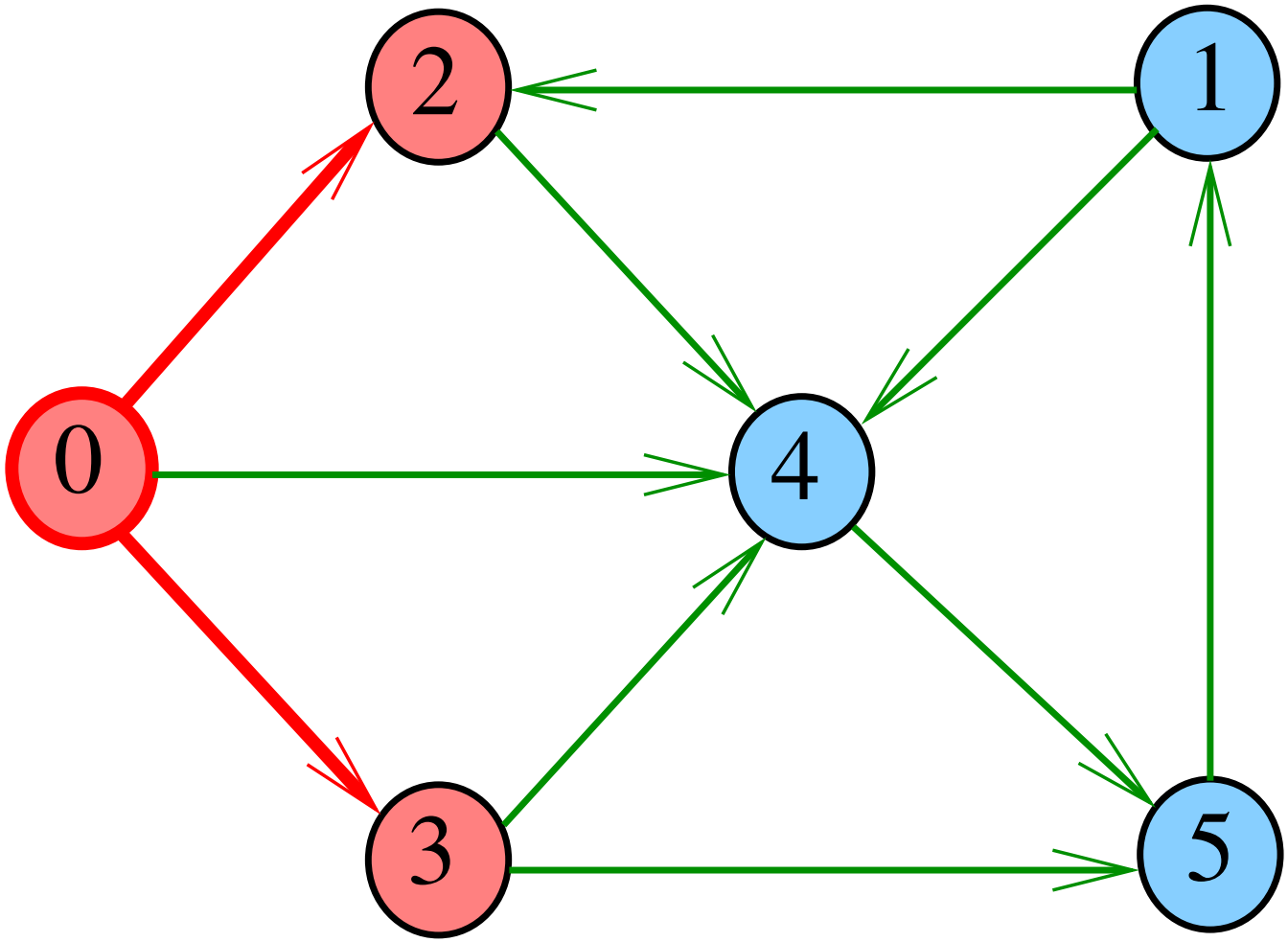
v	0	1	2	3	4	5
distTo[v]	0	6	**1**	6	6	6



Simulação

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

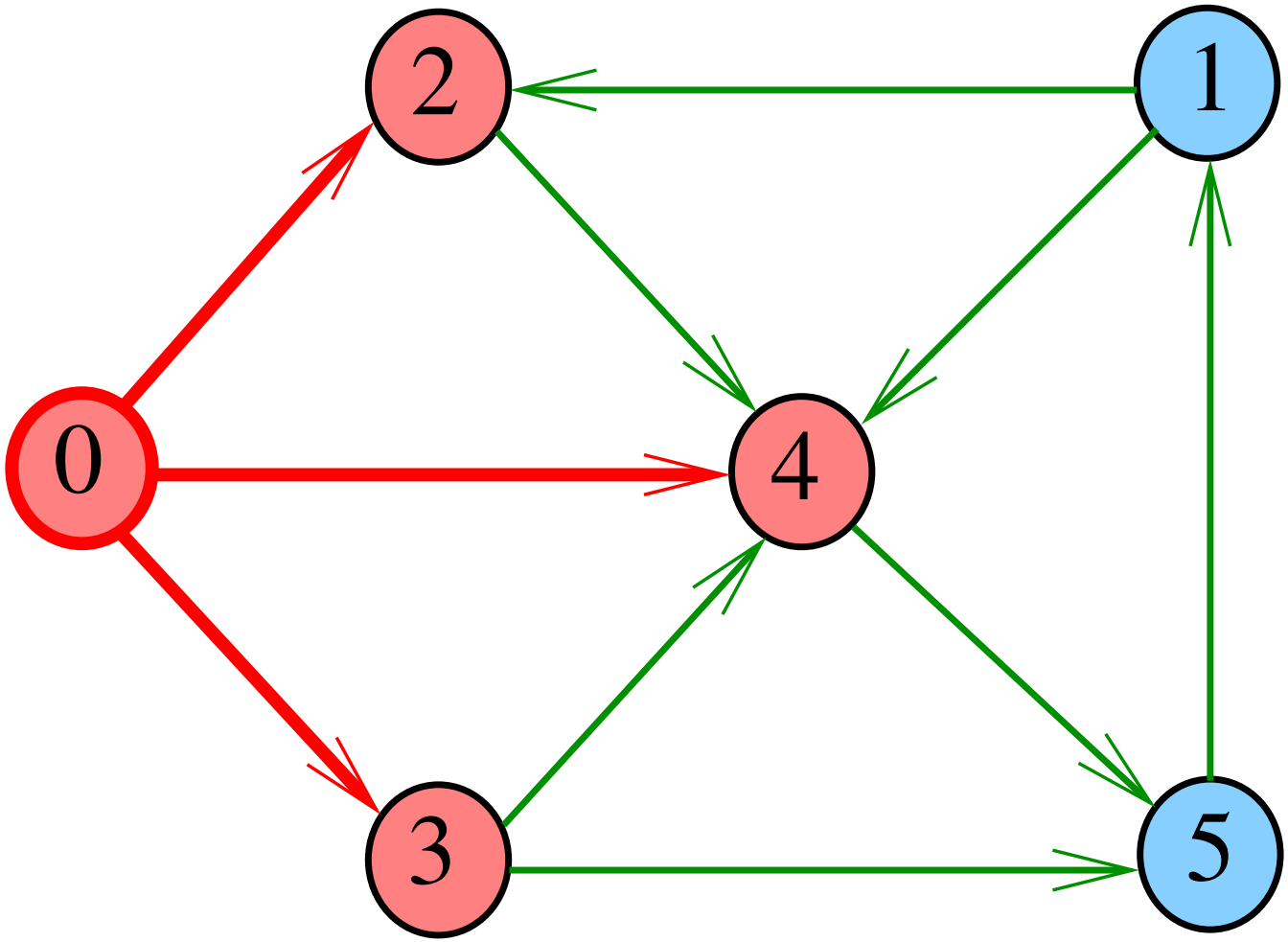
v	0	1	2	3	4	5
distTo[v]	0	6	1	**1**	6	6



Simulação

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

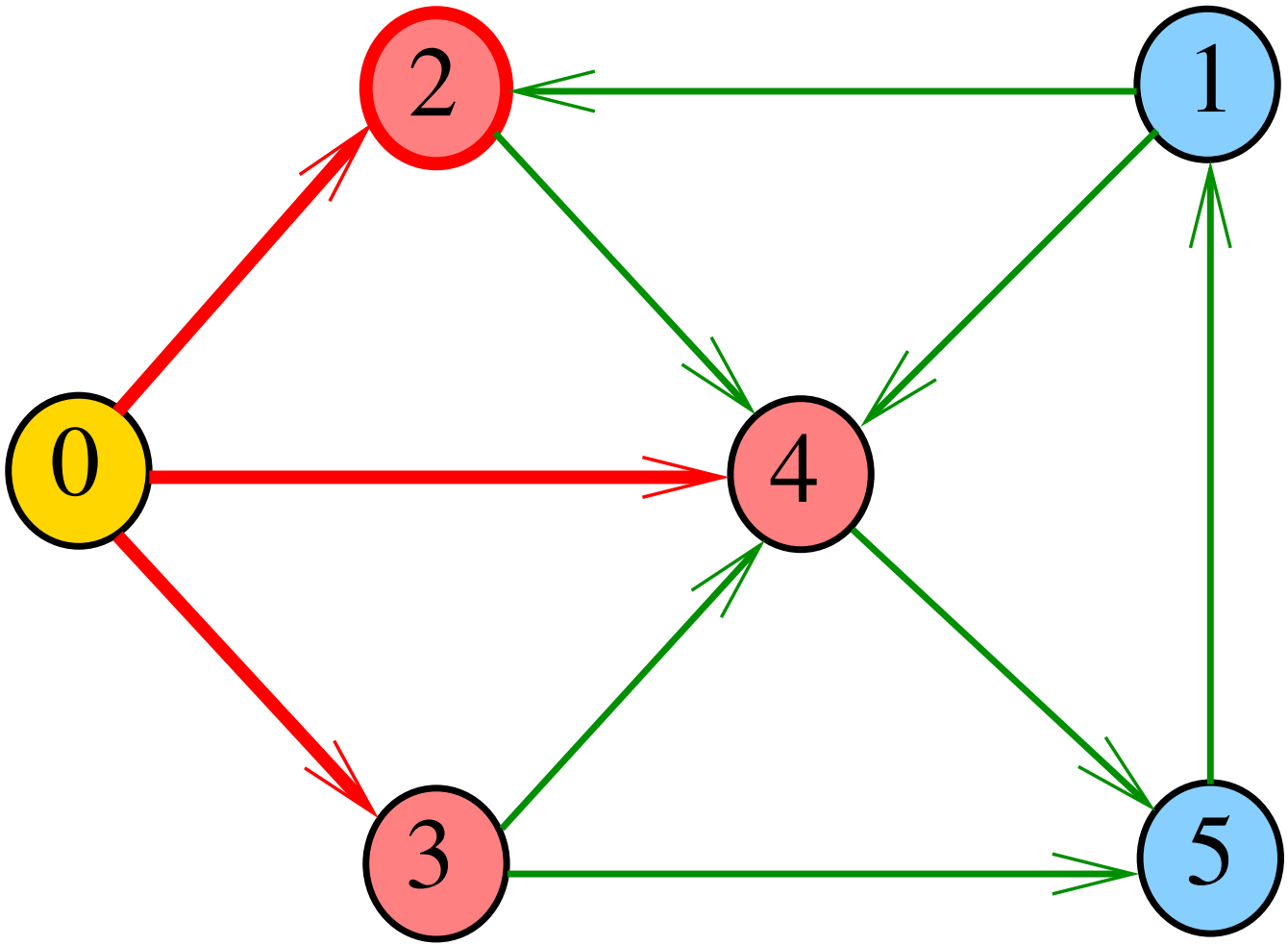
v	0	1	2	3	4	5
distTo[v]	0	6	1	1	**1**	6



Simulação

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

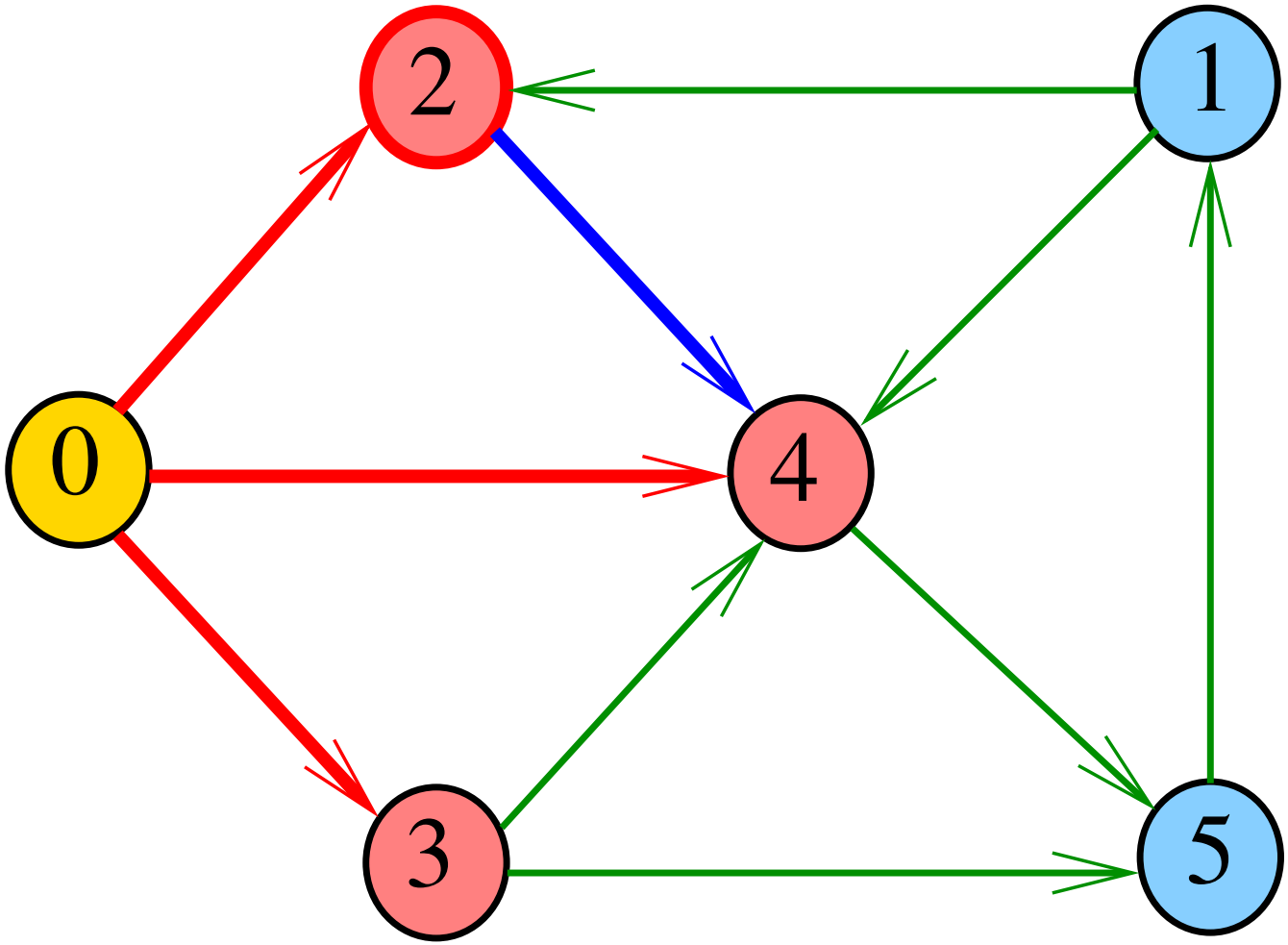
v	0	1	2	3	4	5
distTo[v]	0	6	1	1	1	6



Simulação

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

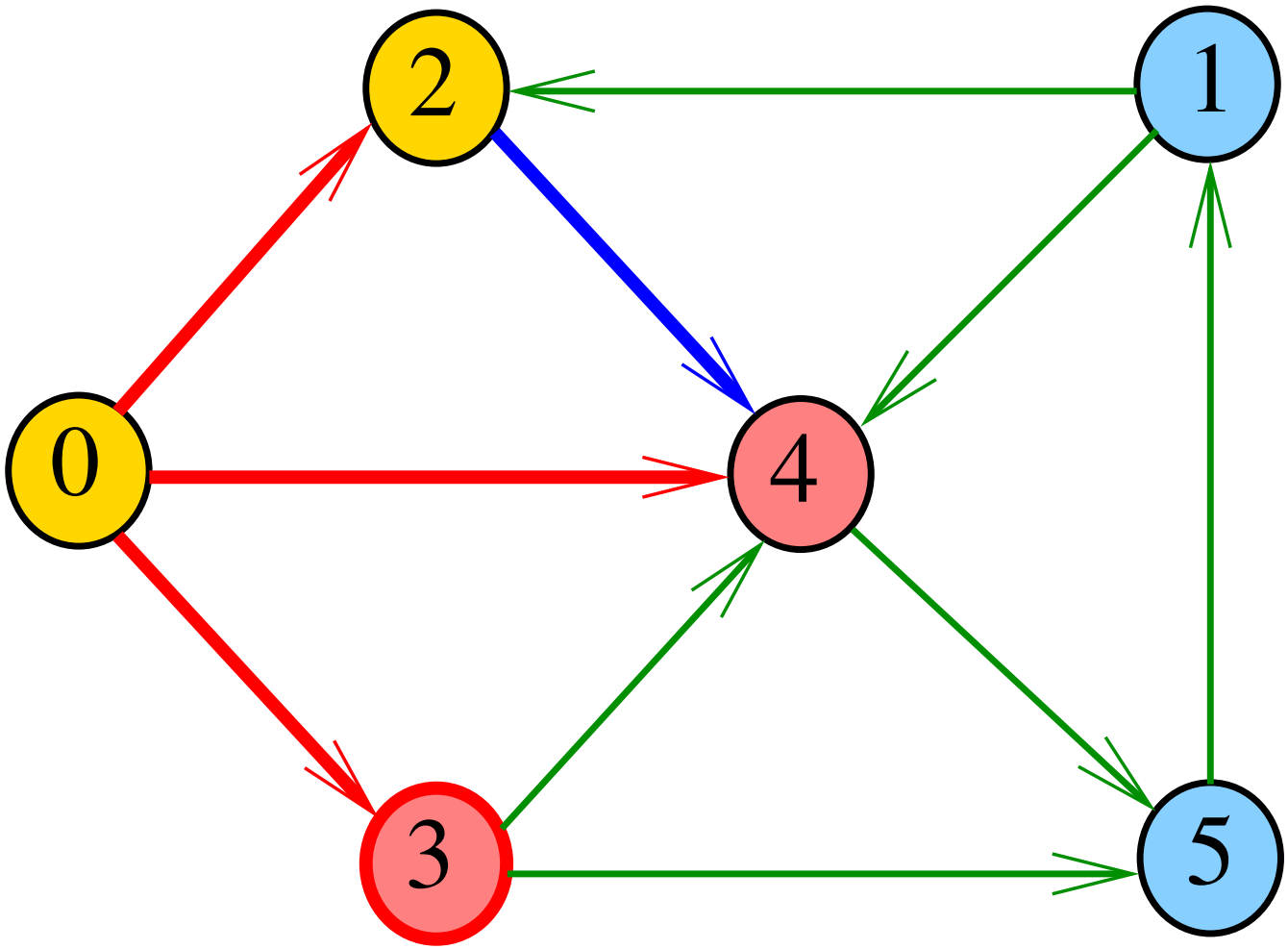
v	0	1	2	3	4	5
distTo[v]	0	6	1	1	1	6



Simulação

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

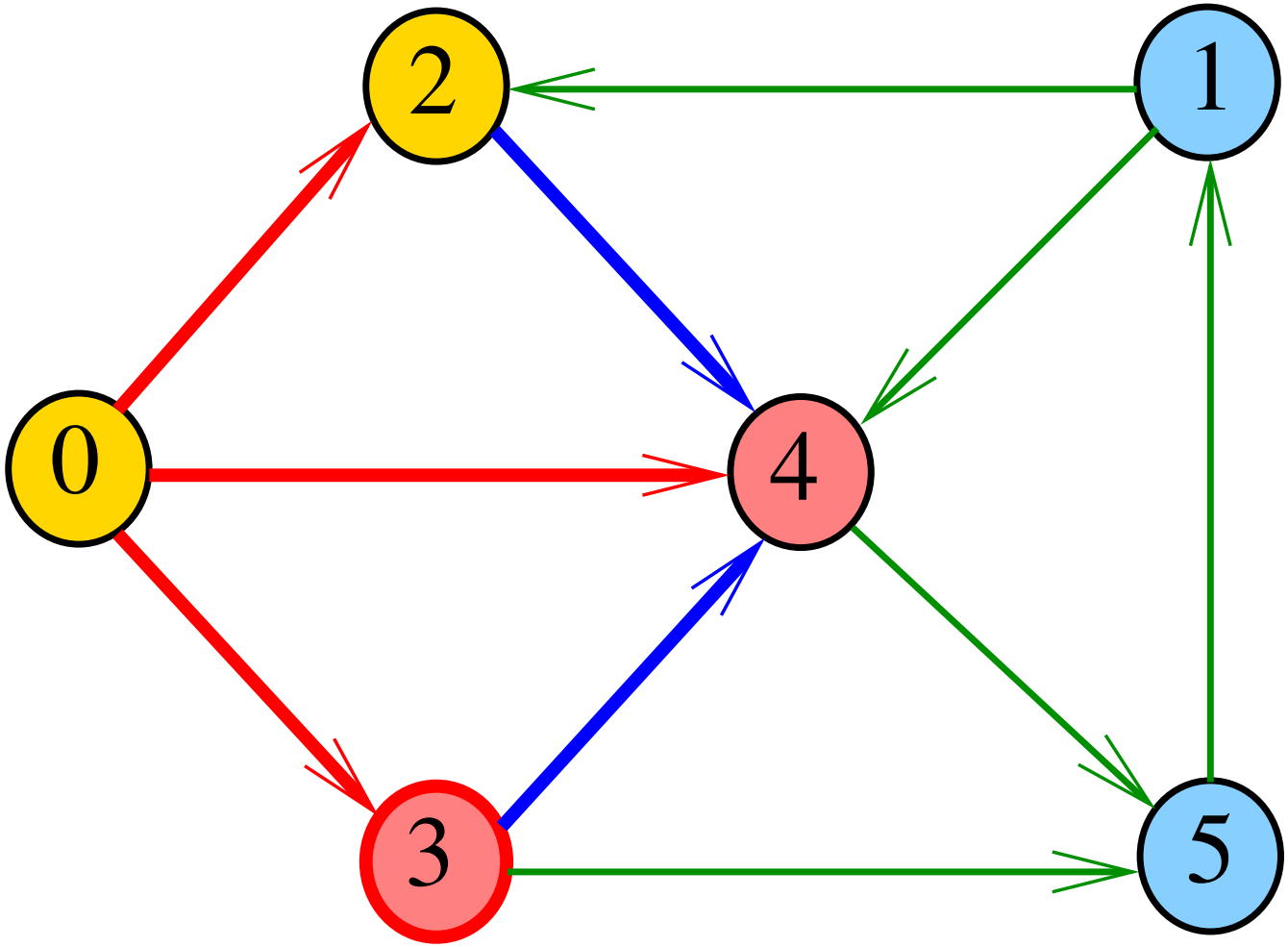
v	0	1	2	3	4	5
distTo[v]	0	6	1	1	1	6



Simulação

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

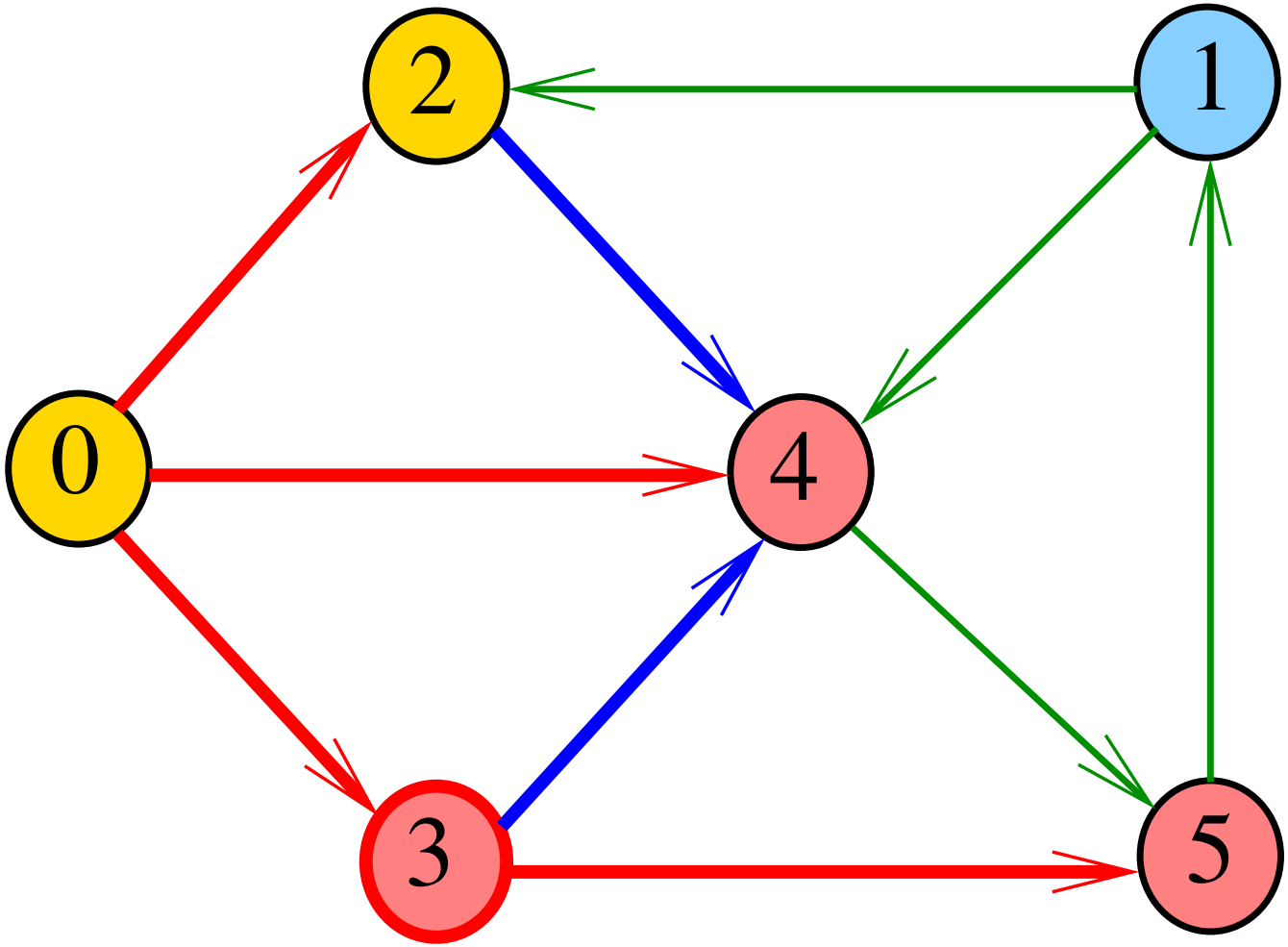
v	0	1	2	3	4	5
distTo[v]	0	6	1	1	1	6



Simulação

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

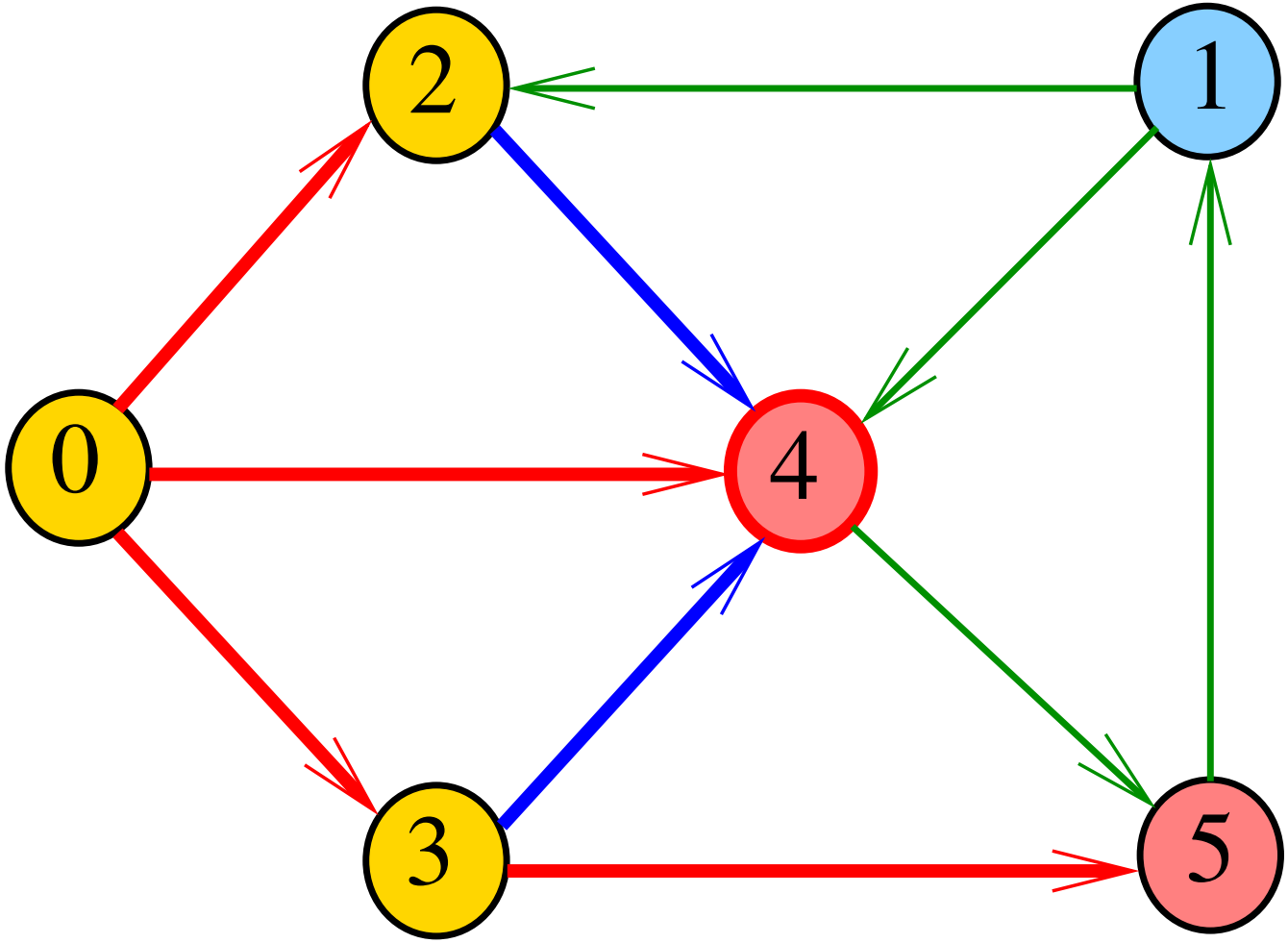
v	0	1	2	3	4	5
distTo[v]	0	6	1	1	1	2



Simulação

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

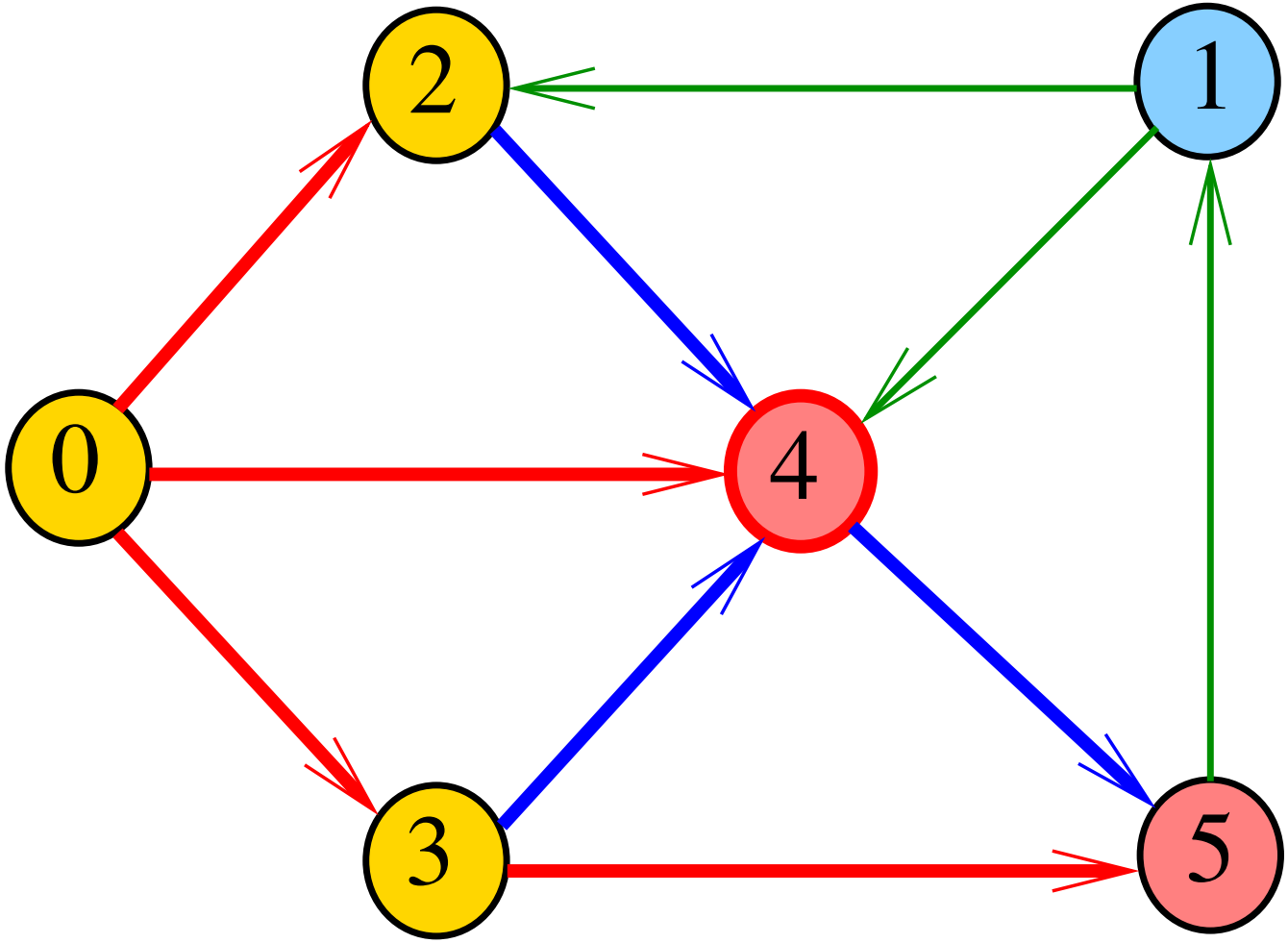
v	0	1	2	3	4	5
distTo[v]	0	6	1	1	1	2



Simulação

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

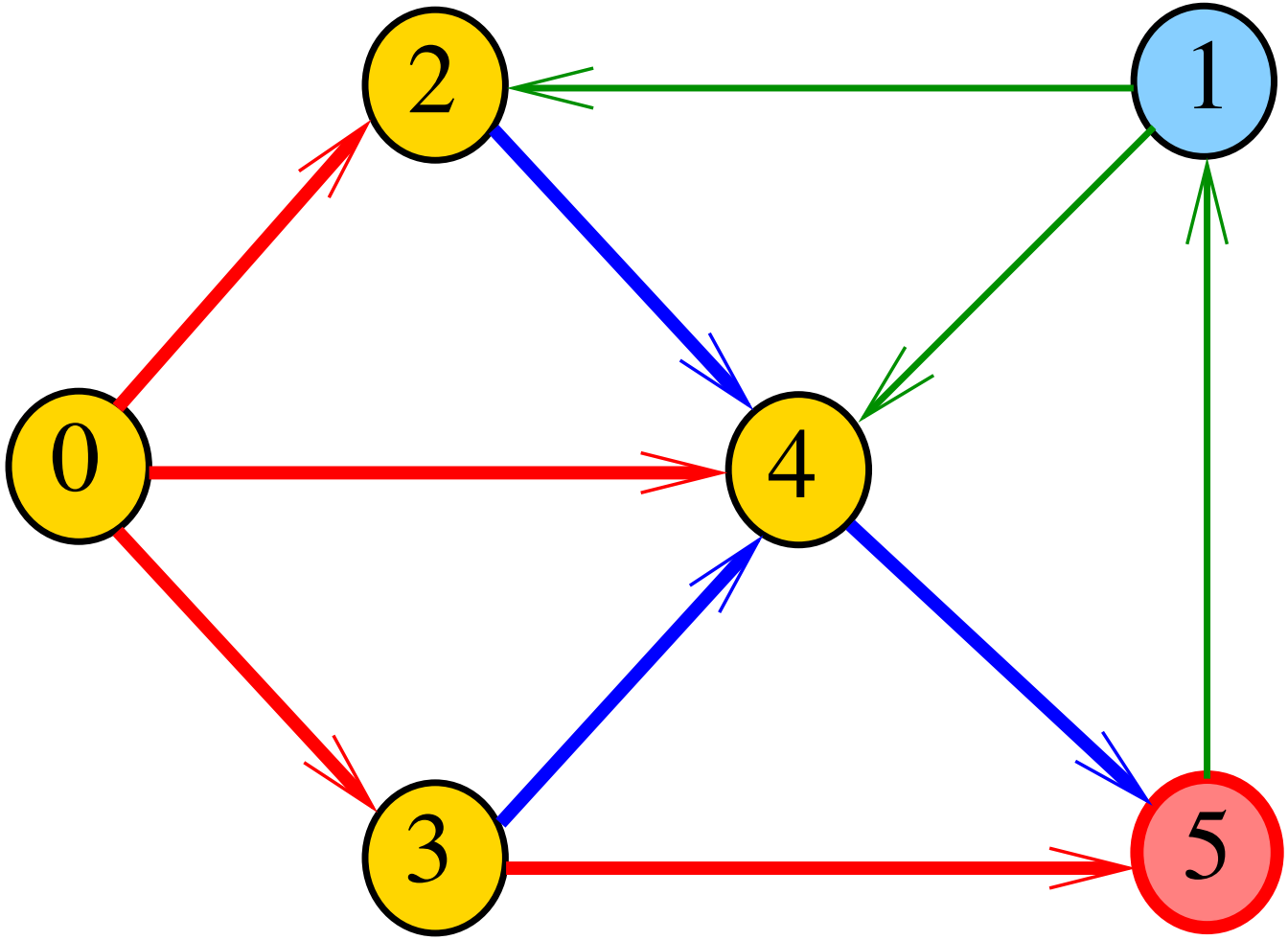
v	0	1	2	3	4	5
distTo[v]	0	6	1	1	1	2



Simulação

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

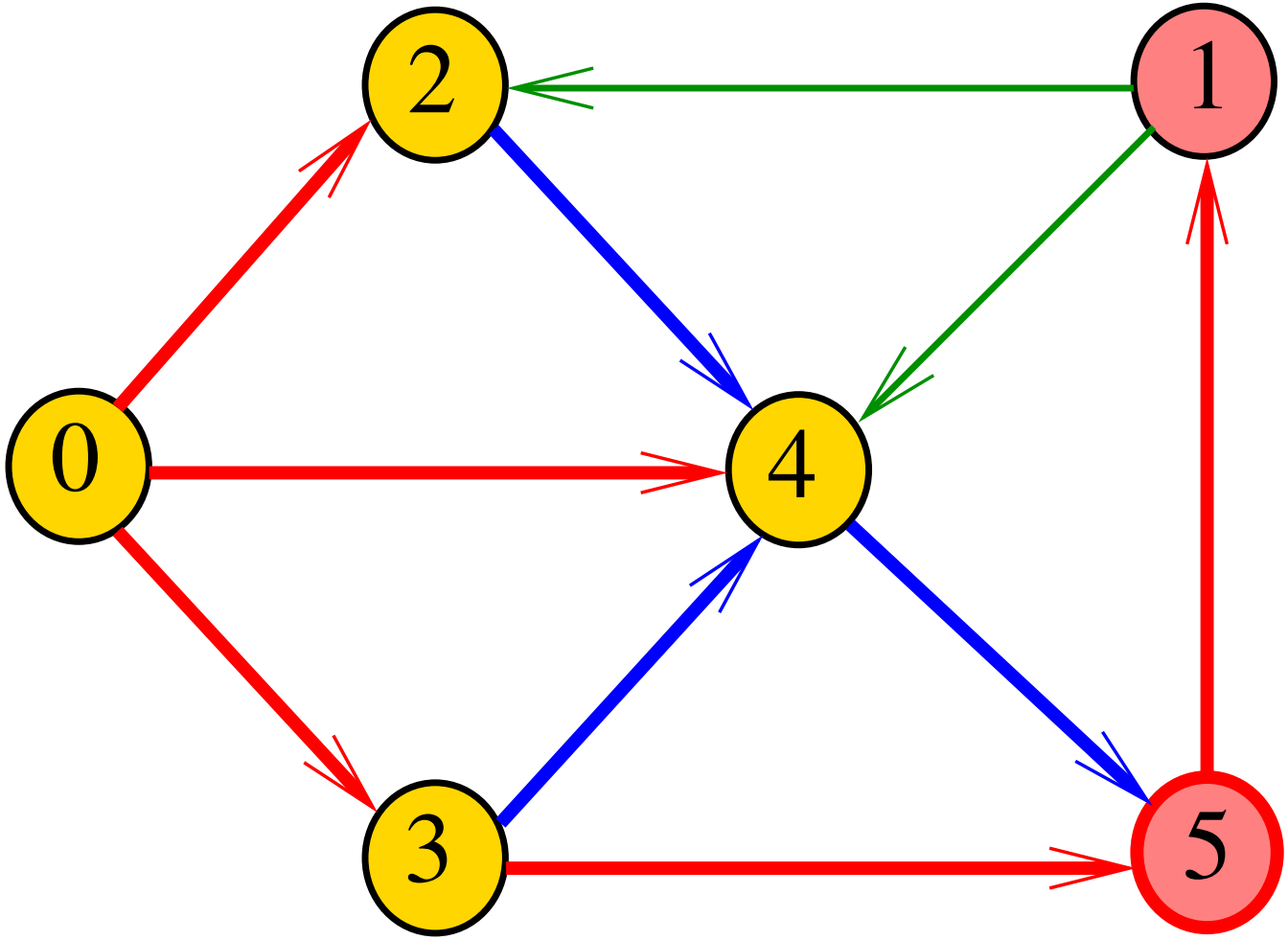
v	0	1	2	3	4	5
distTo[v]	0	6	1	1	1	2



Simulação

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

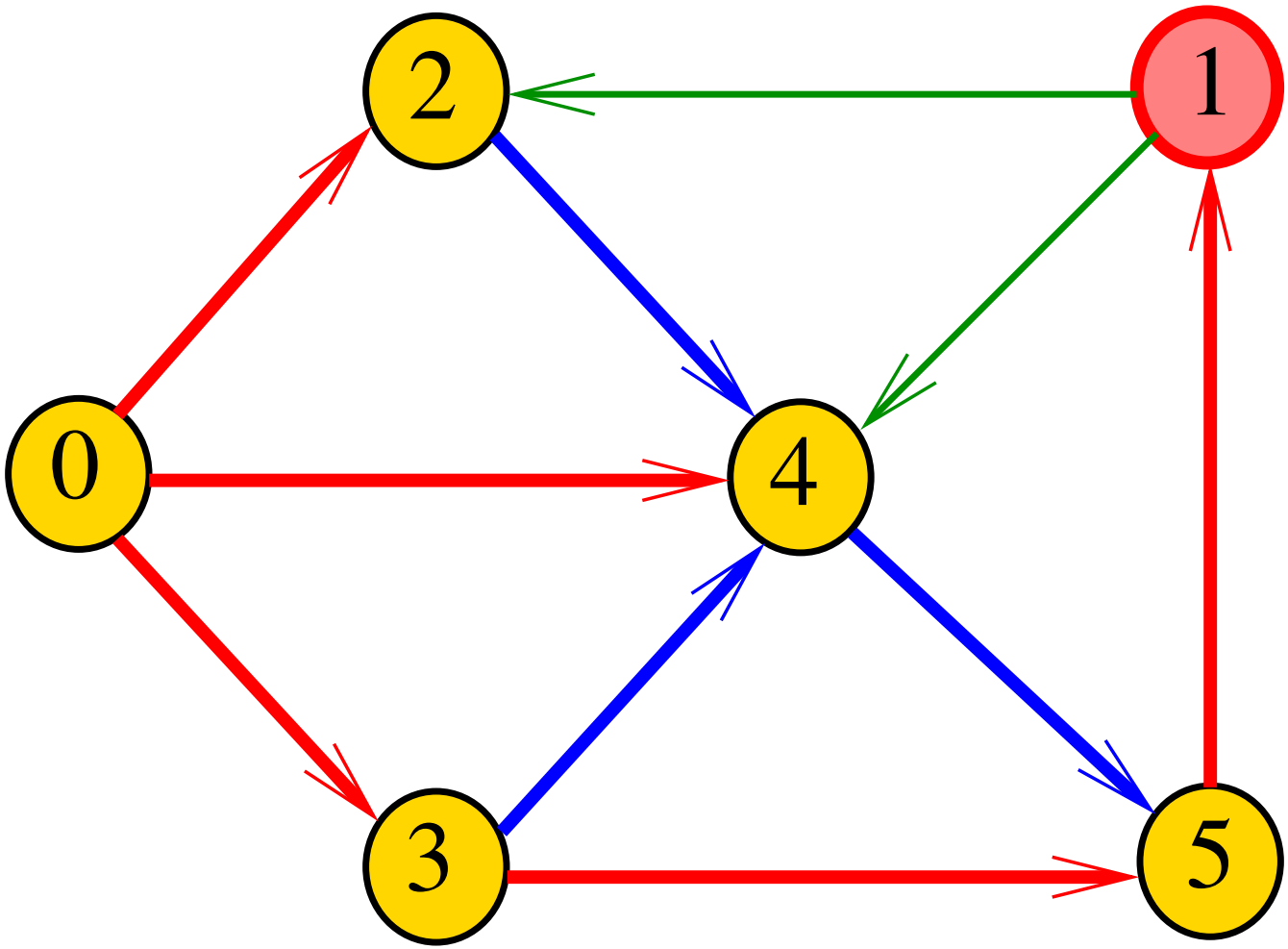
v	0	1	2	3	4	5
distTo[v]	0	3	1	1	1	2



Simulação

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

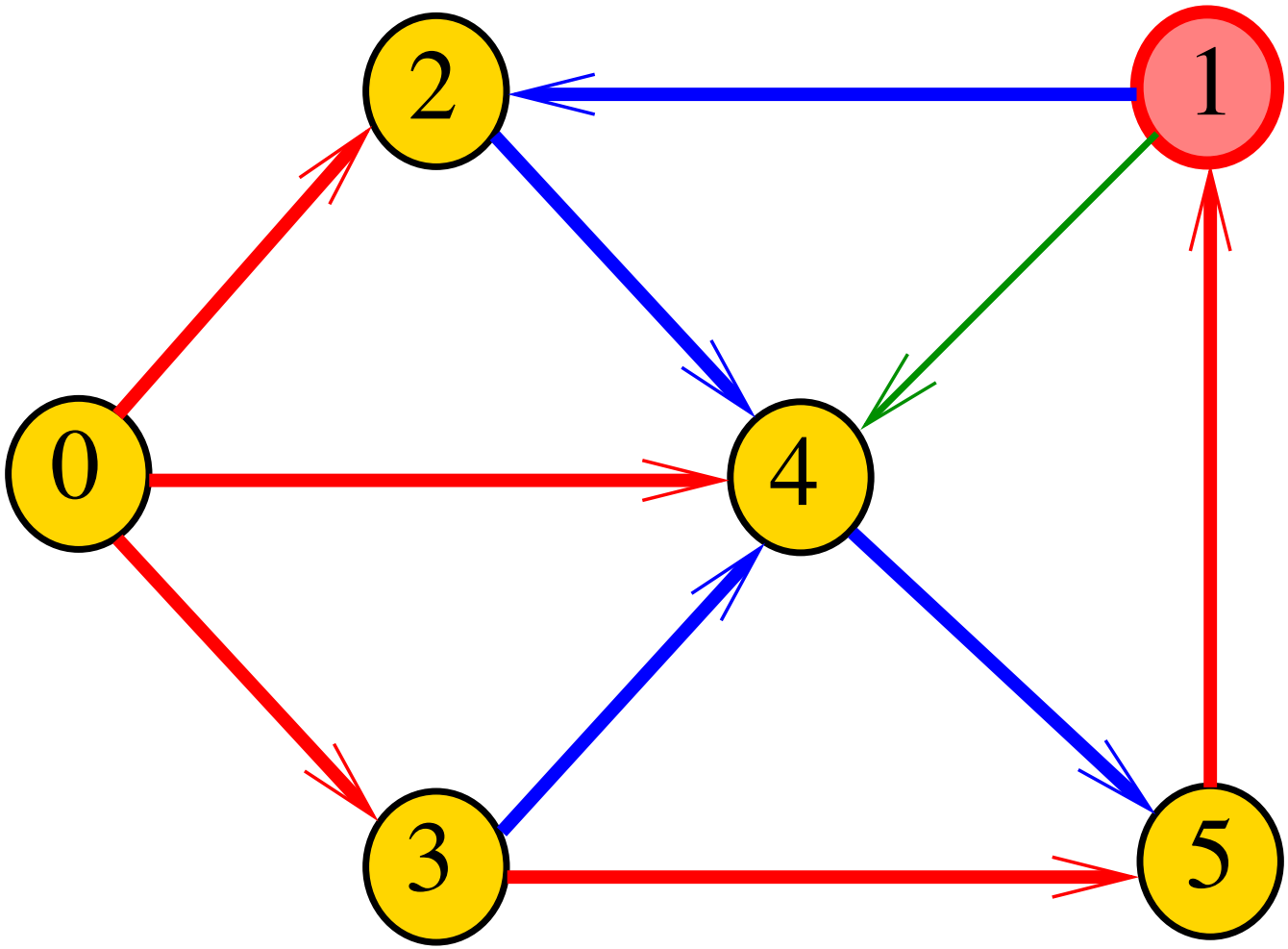
v	0	1	2	3	4	5
distTo[v]	0	3	1	1	1	2



Simulação

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

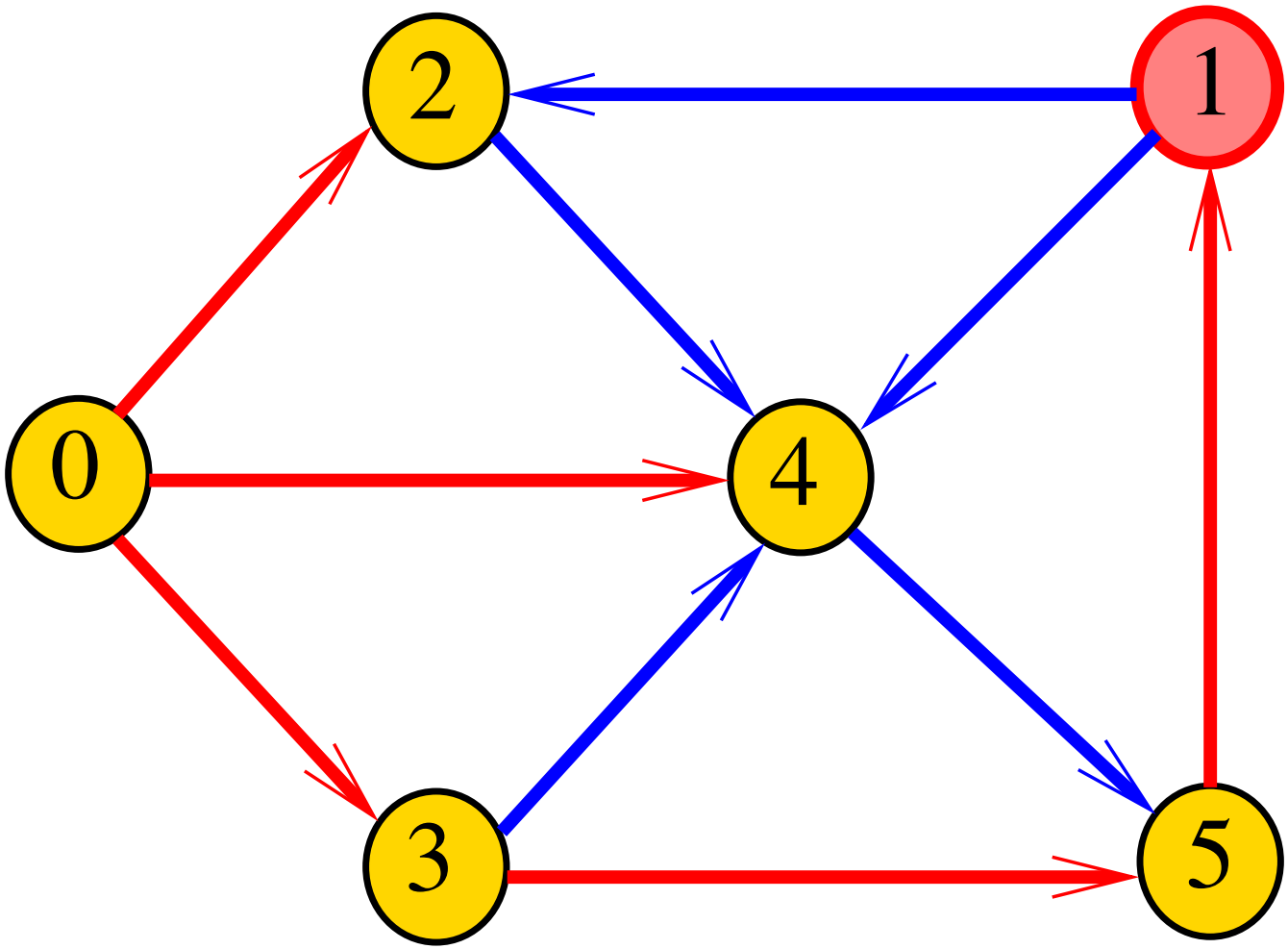
v	0	1	2	3	4	5
distTo[v]	0	3	1	1	1	2



Simulação

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

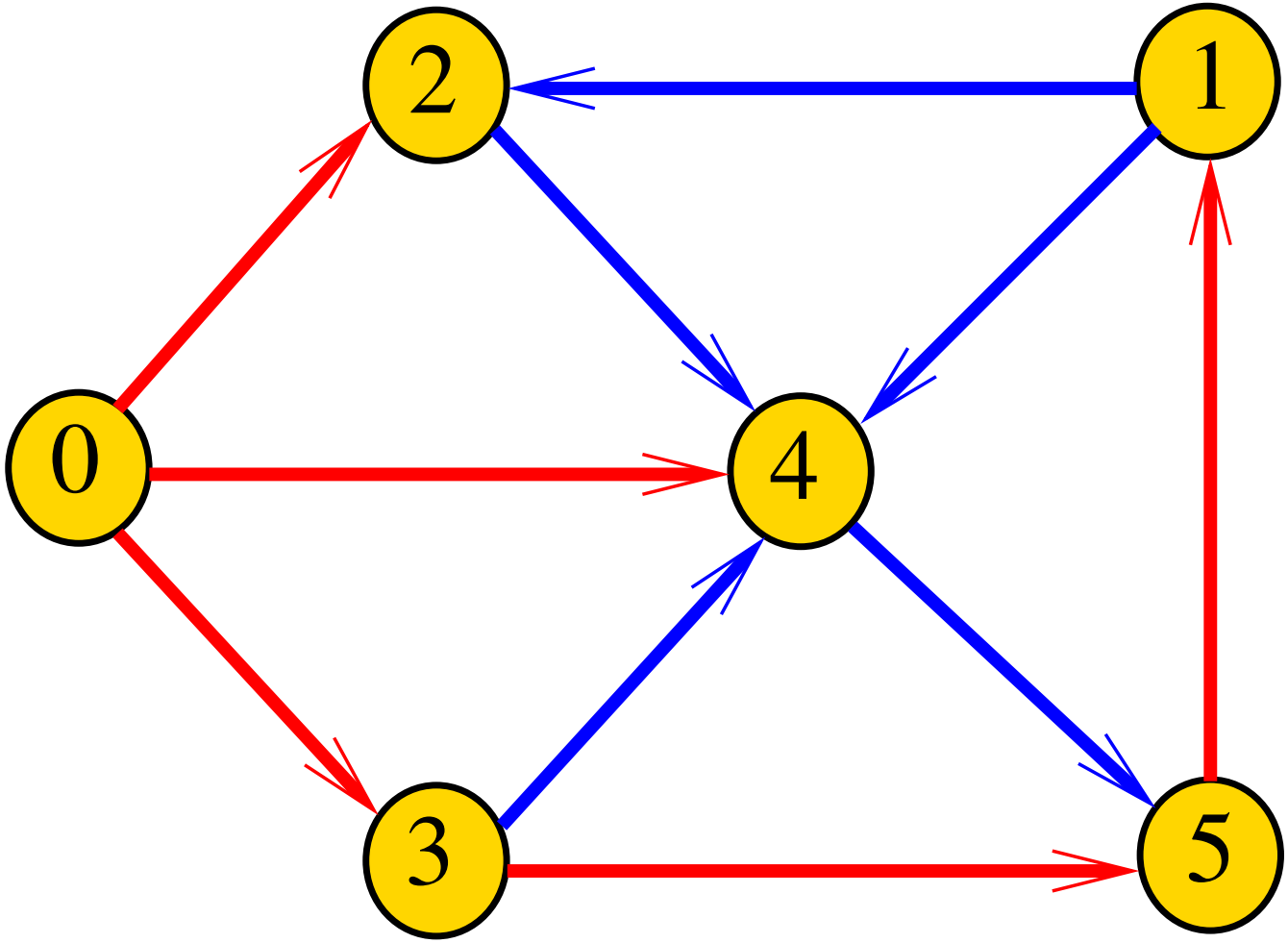
v	0	1	2	3	4	5
distTo[v]	0	3	1	1	1	2



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



BreadFirstDirectedPaths

`BreadFirstDirectedPaths` 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()];
private int[] edgeTo = new int[G.V()];

BreadFirstDirectedPaths(Digraph G, int
s);
```

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`

Consumo de tempo

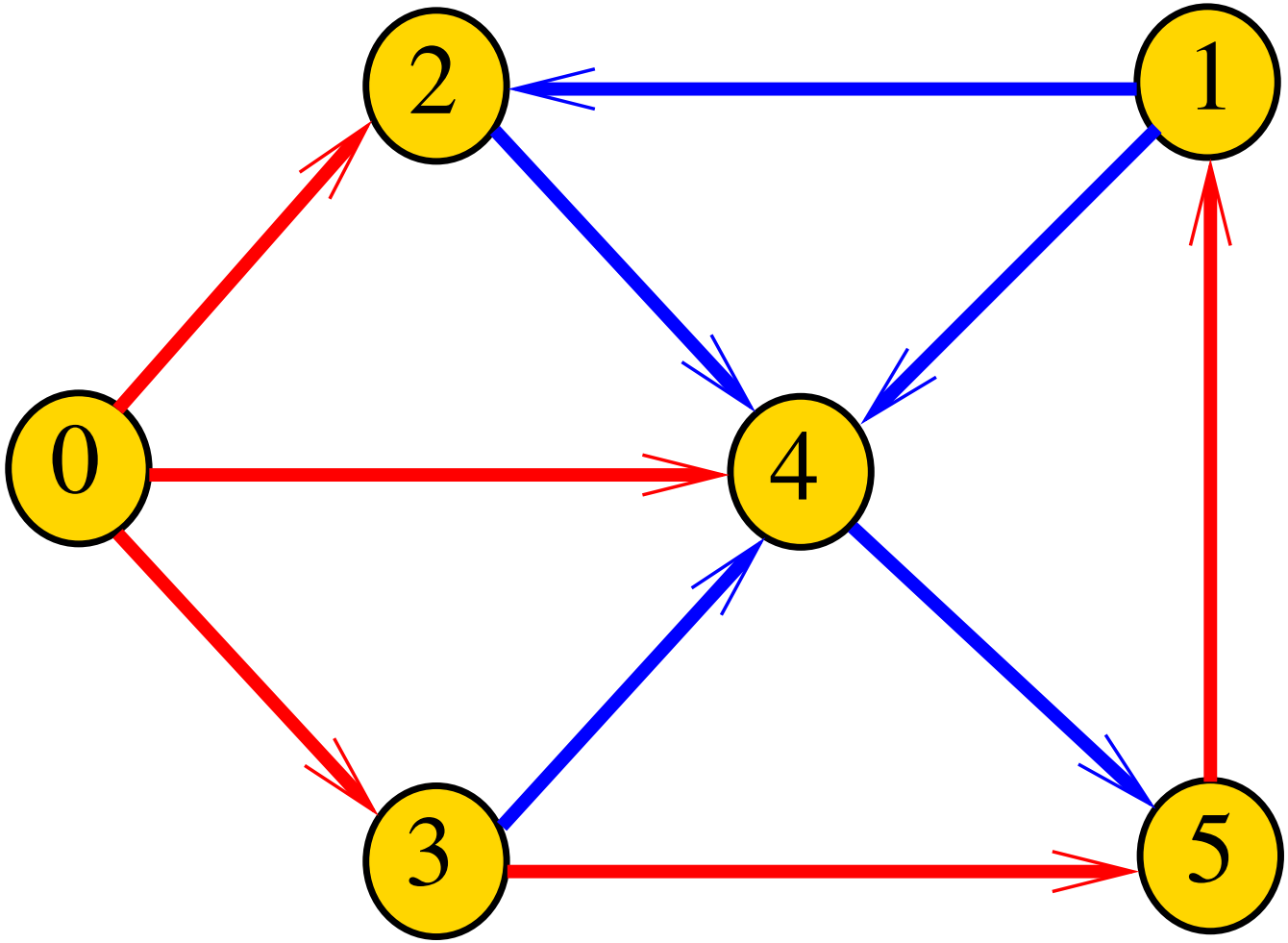
O consumo de tempo de `BreadFirstDirectedPaths` para vetor de listas de adjacência é $O(V + A)$.

O consumo de tempo de `BreadFirstDirectedPaths` 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

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

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

1-Potenciais

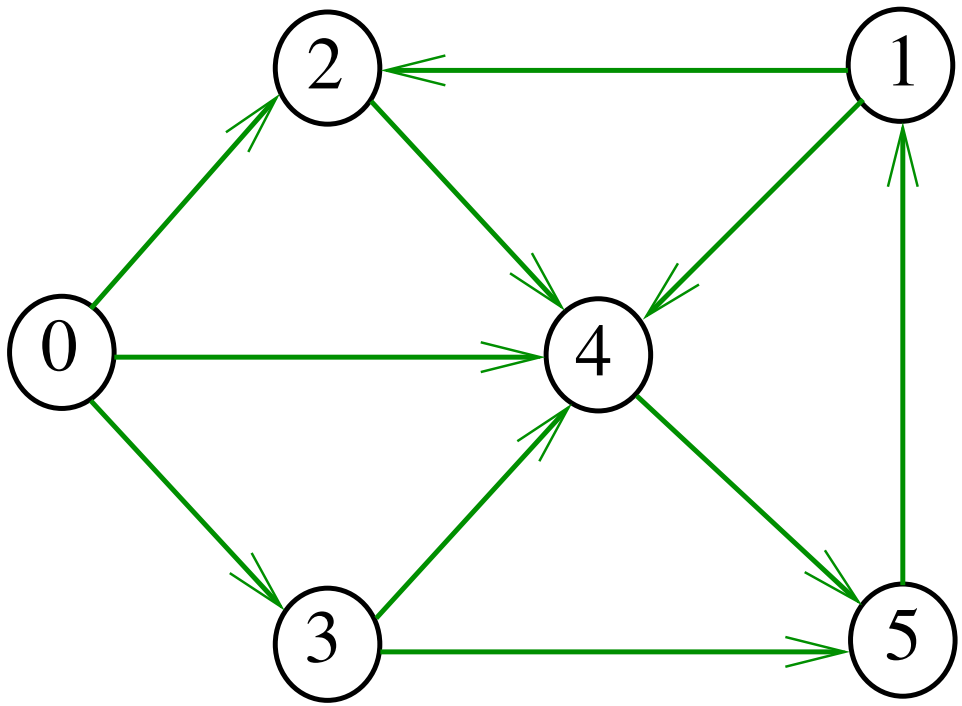
1-potenciais

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



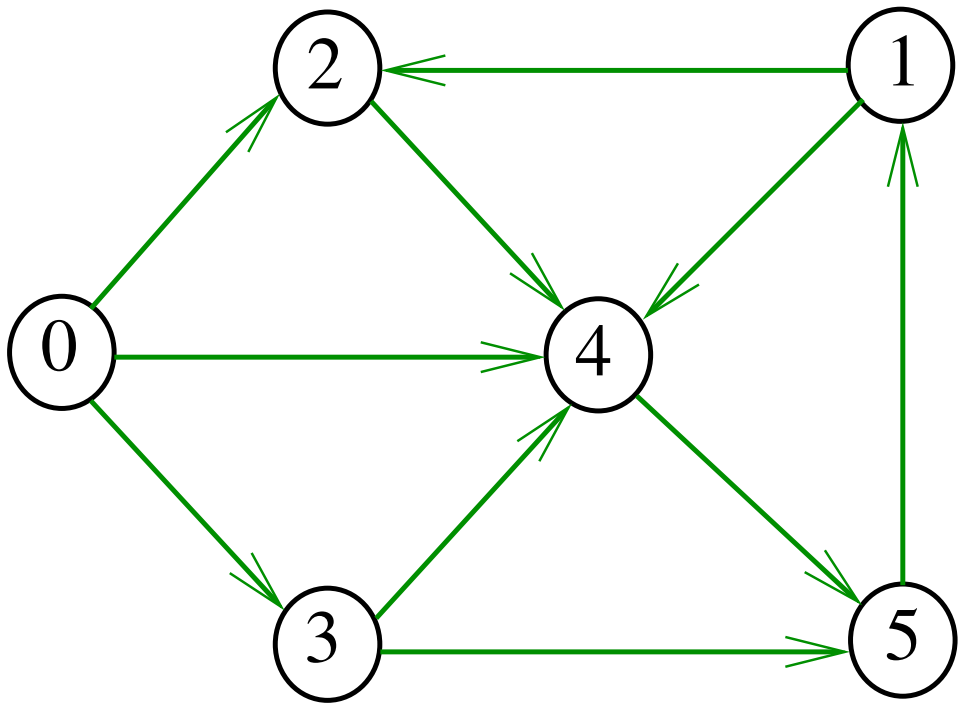
1-potenciais

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



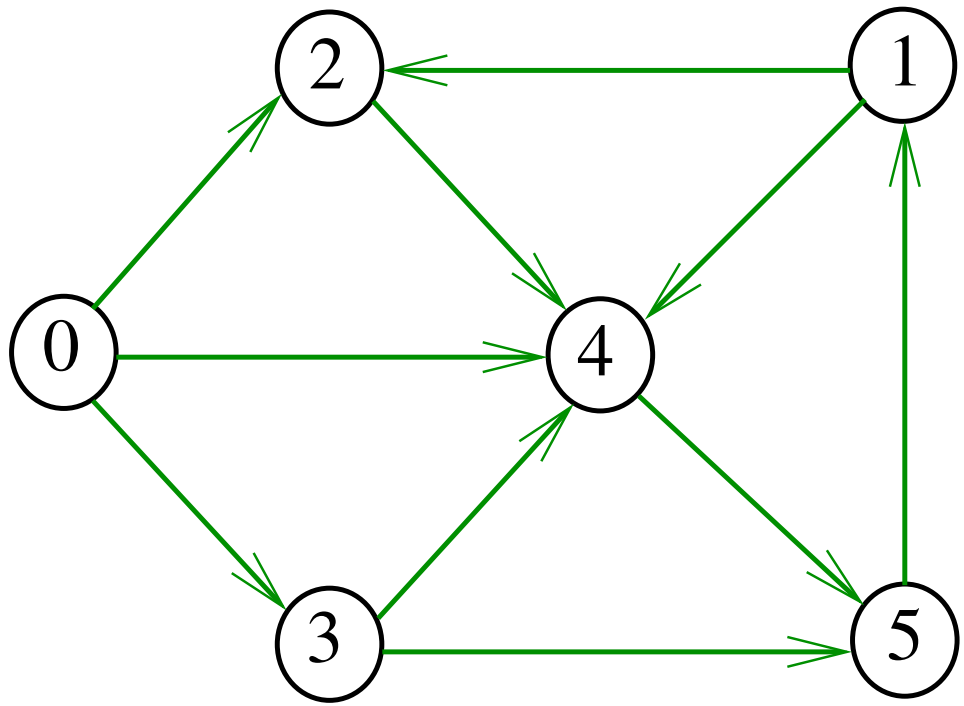
Propriedade dos 1-potencias

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

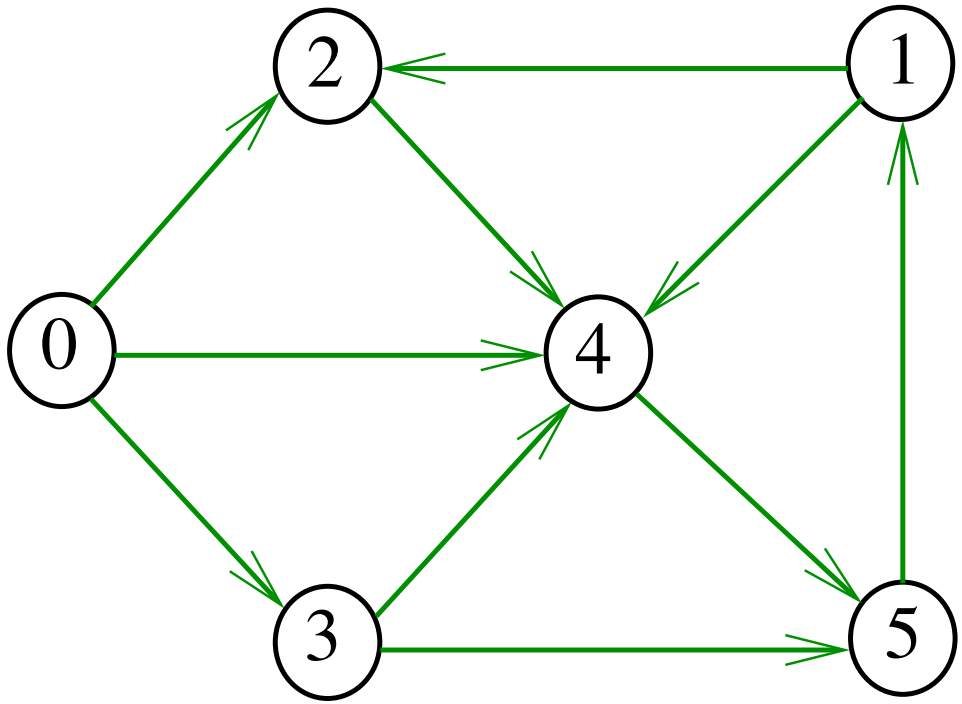
Se $\mathtt{P}$ é um caminho de $\mathtt{s}$ a $\mathtt{t}$ e $\mathtt{y}$ é um 1-potencial tais que

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

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 `BreadFirstDirectedPaths`, no início `while` do método `bfs()` valem as seguintes invariantes:

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

(i1) $\texttt{edgeTo[s]} = \texttt{s}$ e $\texttt{y[s]} = 0$;

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

(i3) para cada vértice `v`, se $\texttt{y[v]} \neq \texttt{G.V()}$ 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 DIGRAPHdist

Início da última iteração:

- y é um 1-potencial, por (i4)
- se y[t] $\neq$ G->V, então edgeTo[t] $\neq -1$ [(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

Conclusão: o algoritmo faz o que promete.

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

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

Custos nos arcos

S 20.1

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 `DirectedEdge`.

```
DirectedEdge(int v, int w, double weight)
    private final int v;
    private final int w;
    private final double weight;

    double weight()...

    int from()...
```

DirectedEdge

O construtor `DirectedEdge` 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`

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

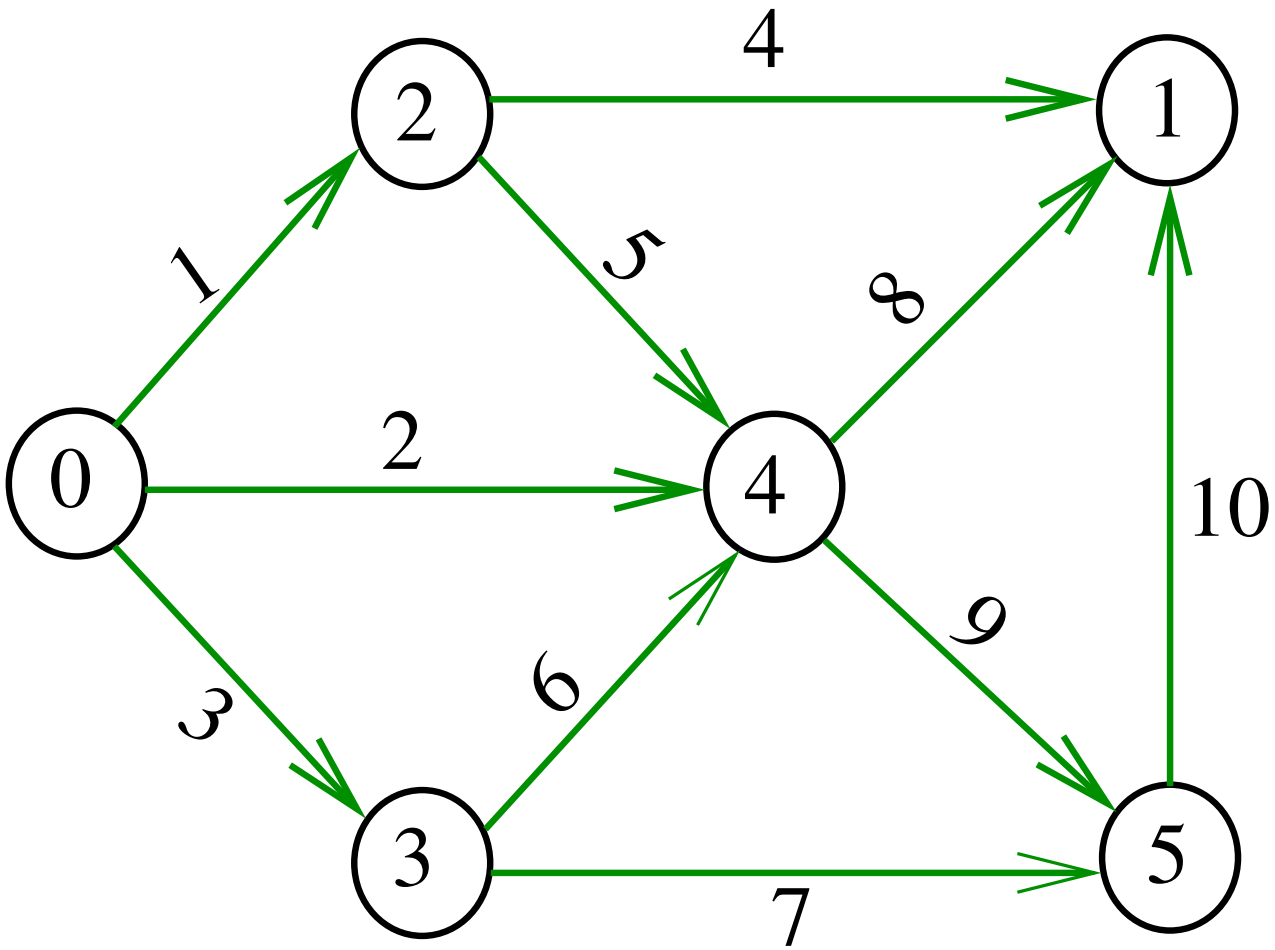
Vetor de listas de adjacência

A lista de adjacência de um vértice `v` é composta por nós do tipo `DirectedEdge[]`
Um `next` de `Bag` é um ponteiro para um `DirectedEdge[]`
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

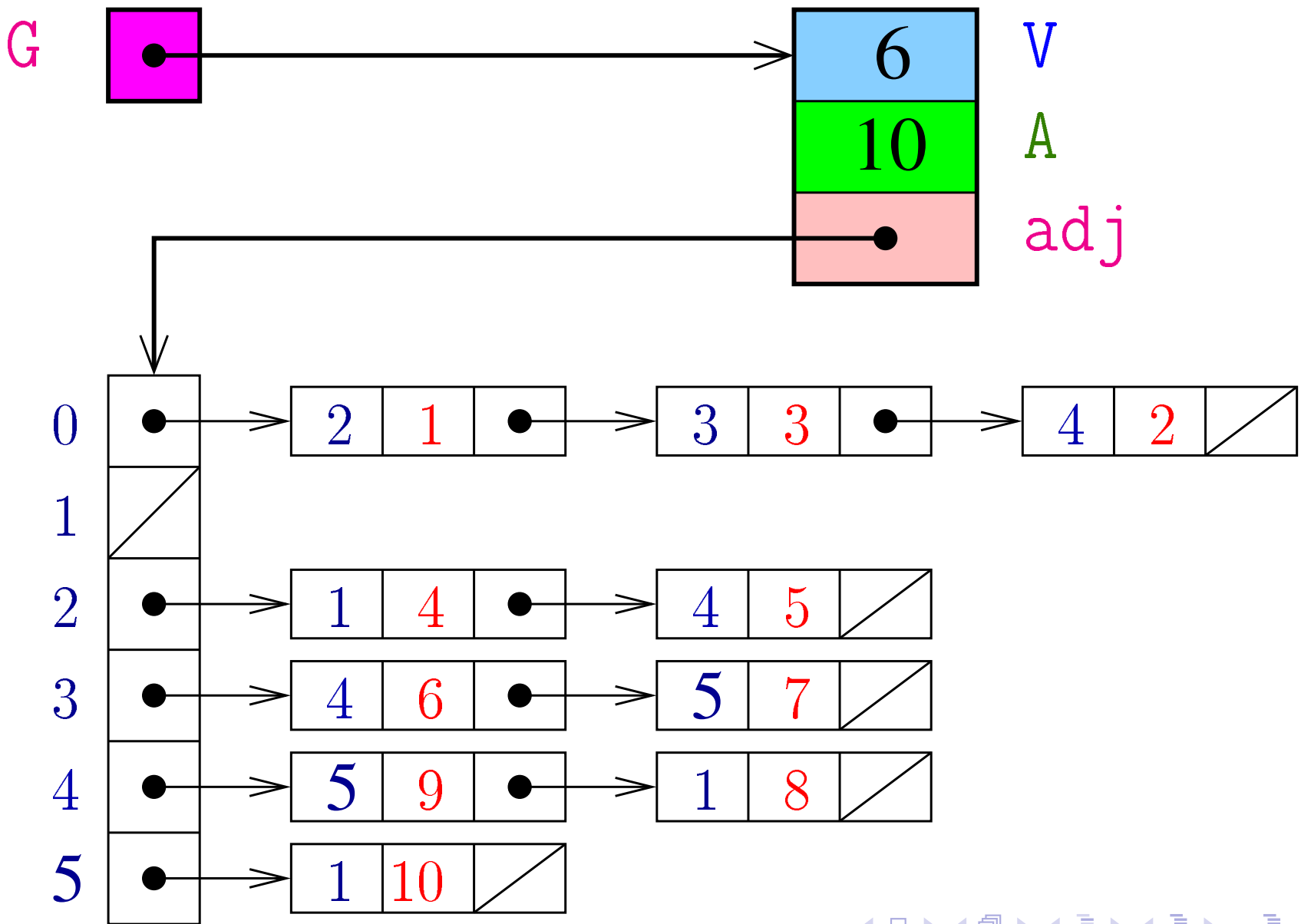
```
public class EdgeWeightDigraph {
        private final int V;
        private final int E;
        private Bag<DirectedEdge>[] adj;
```

Digrafo

EdgeWeightedDigraph G



Estruturas de dados



Classe EdgeWeightedDigraph

A estrutura **digraph** representa um digrafo

V contém o número de vértices

E contém o número de arcos do digrafo

adj é uma referência para vetor de listas de adjacência

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

V(), E()

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

addEdge() e adj()

Insere um arco e no digrafo G.

```
public addEdge (DirectedEdge e) {
        adj[e.from()].add(e);
        E++;
}
public Iterable<DirectedEdge> adj(int
v) {
        return adj[v];
}
```

edges()

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

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