MAC0323 Algoritmos e Estruturas de Dados II

Edição 2020 – 2

Fonte: ash.atozviews.com

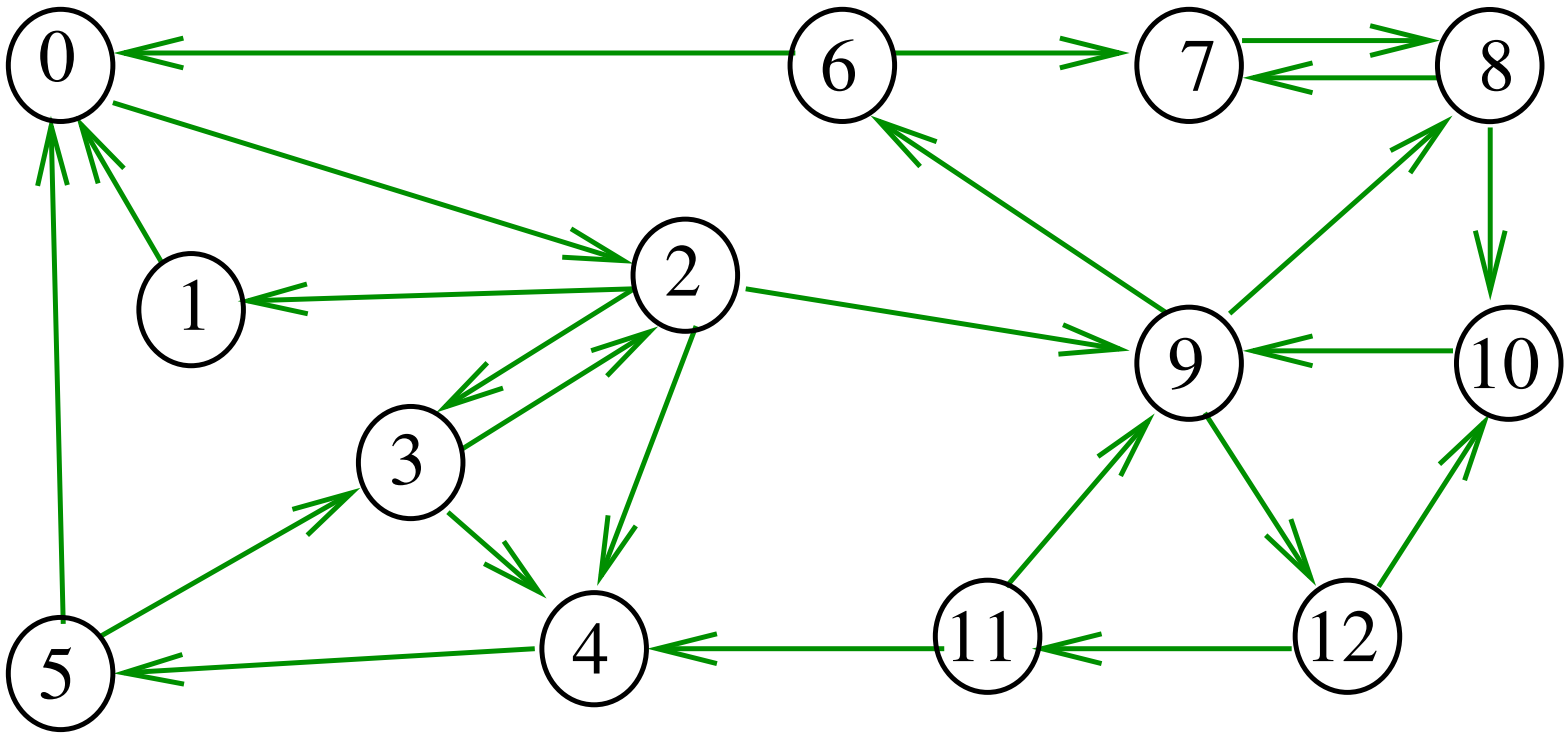
Compacto dos melhores momentos

AULA 21

Digrafos fortemente conexos

Um digrafo é **fortemente conexo** se e somente se para cada par {s,t} de seus vértices, existem caminhos de s a t e de t a s.

Exemplo: um digrafo fortemente conexo



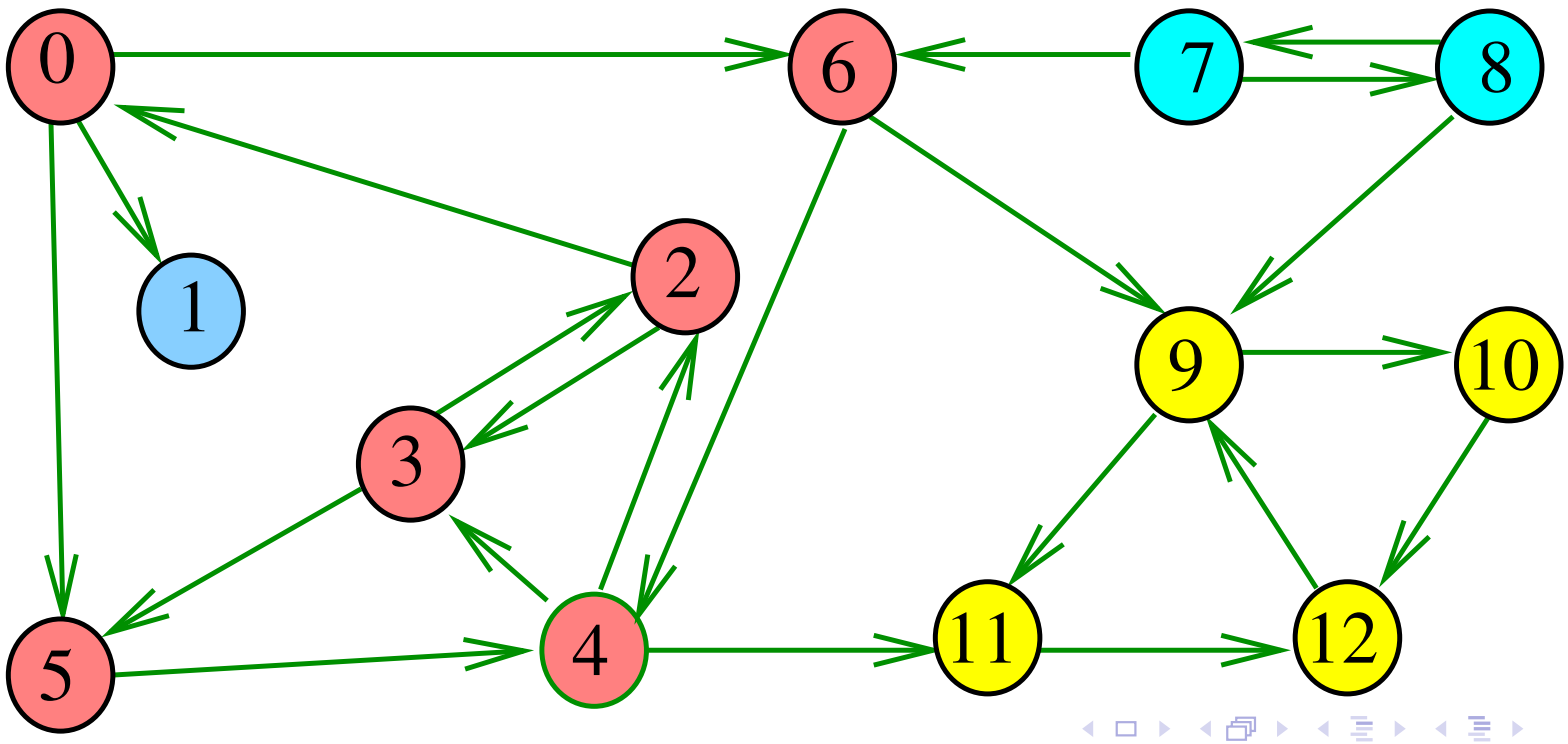
Componentes fortemente conexos

Um componente **fortemente conexo**
(= *strongly connected component* (SCC)) é
um conjunto maximal de vértices `W` tal que
o digrafo induzido por `W` é fortemente conexo.

Componentes fortemente conexos

Um componente **fortemente conexo**
(= *strongly connected component* (SCC)) é
um conjunto maximal de vértices `W` tal que
o digrafo induzido por `W` é fortemente conexo.

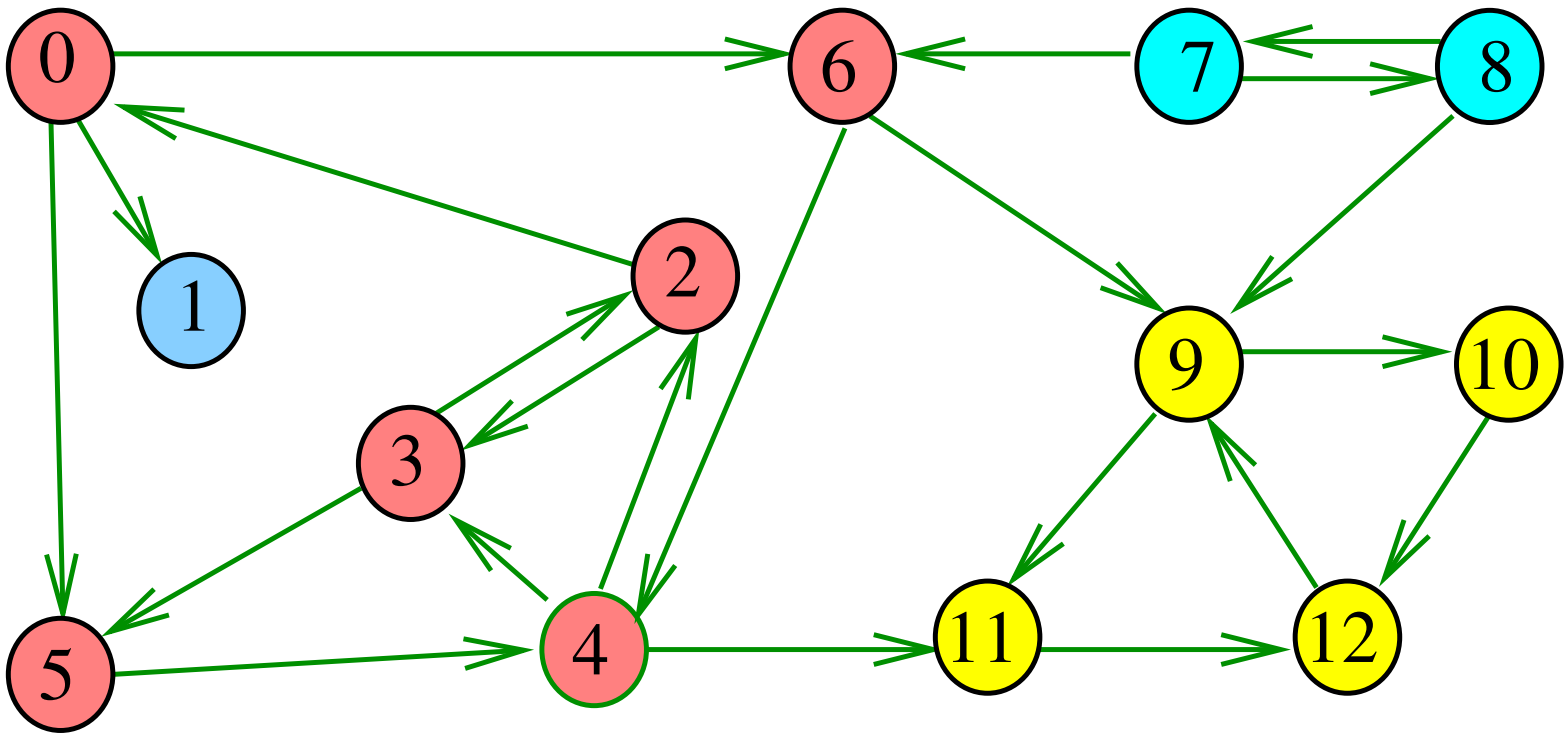
Exemplo: 4 componentes fortemente conexos



Determinando componentes f.c.

Problema: determinar os componentes fortemente conexos.

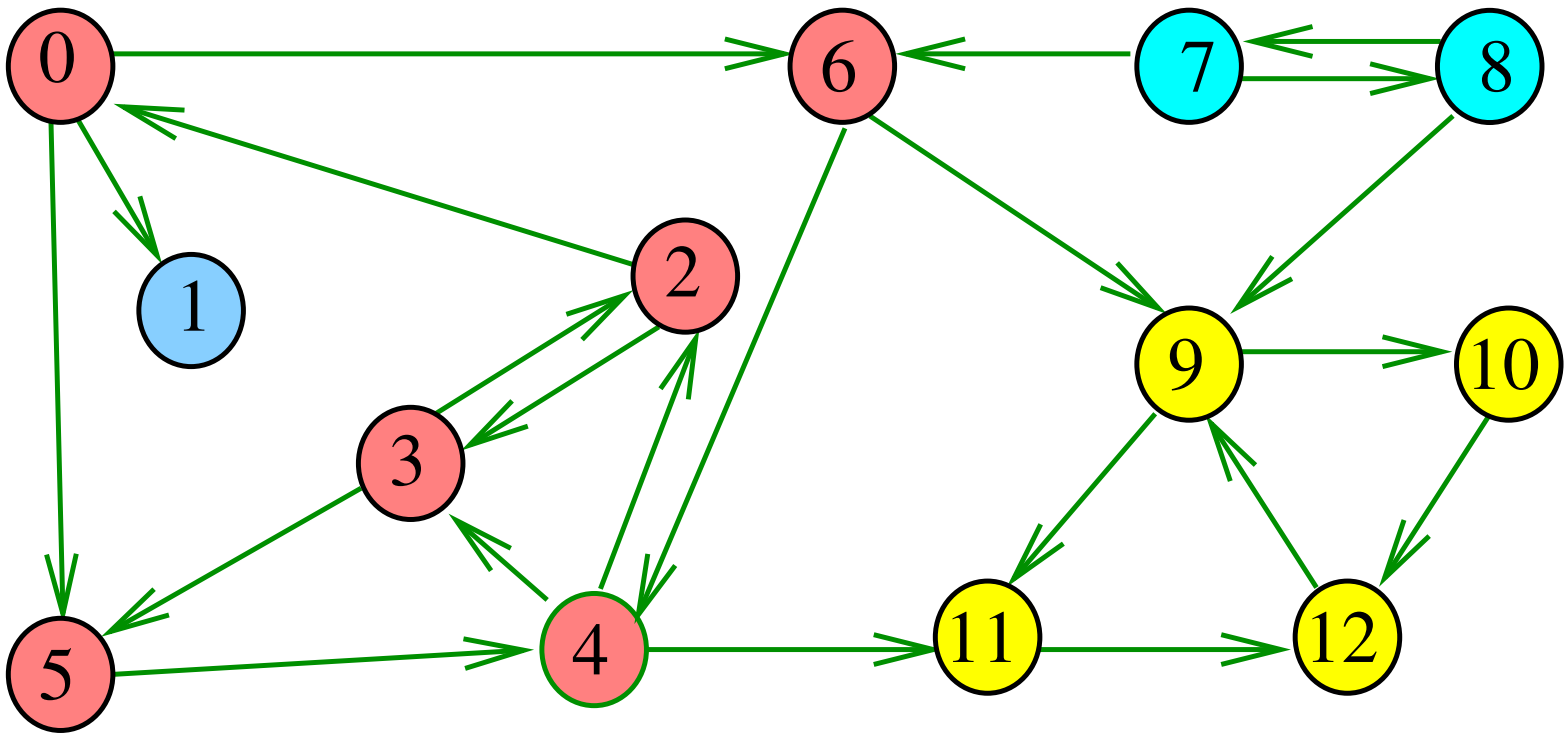
Exemplo: 4 componentes fortemente conexos



Propriedade

Um digrafo G e seu digrafo reverso R têm os mesmos componente fortemente conexos.

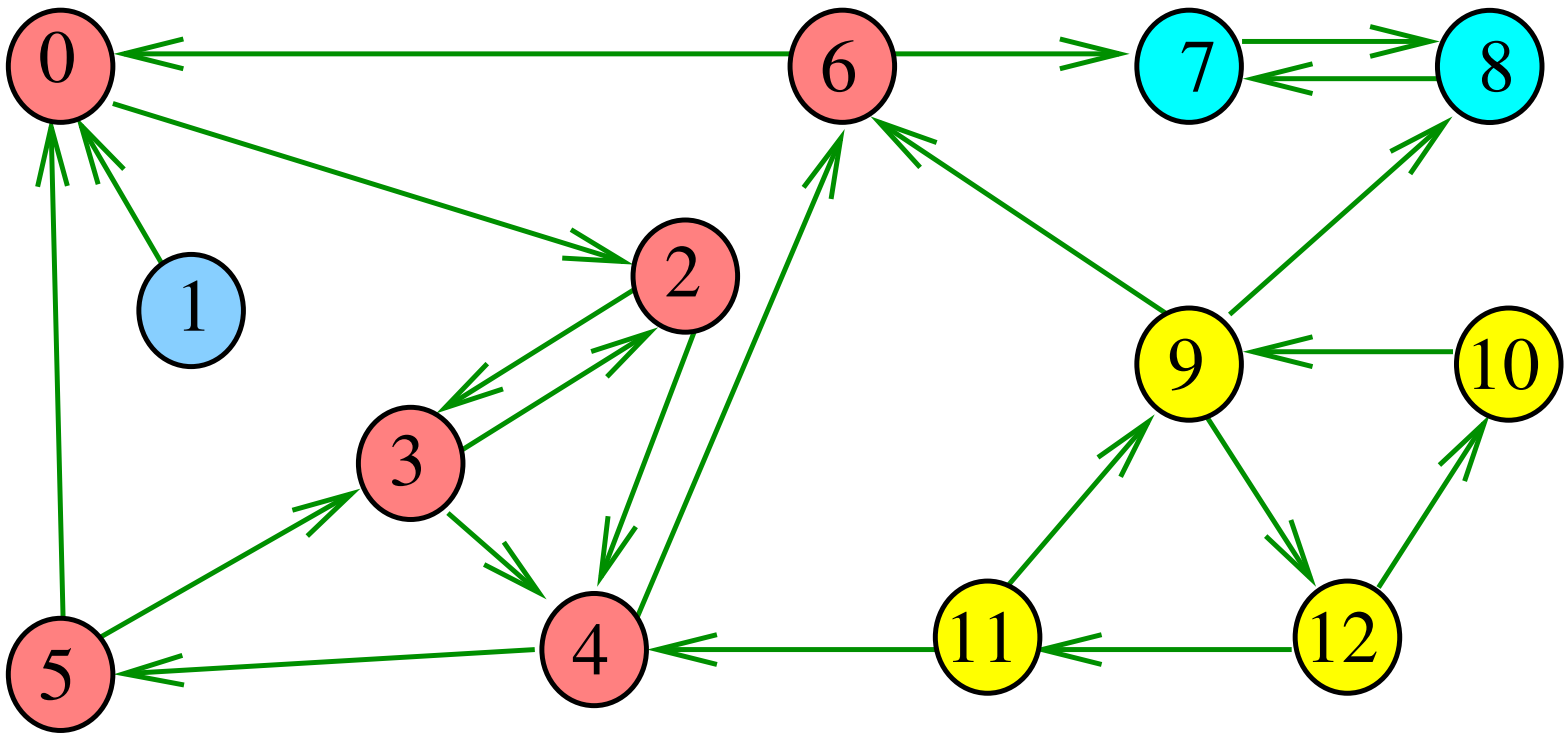
Exemplo: Digrafo G



Propriedade

Um digrafo G e seu digrafo reverso R têm os mesmos componente fortemente conexos.

Exemplo: Digrafo reverso R de G



AULA 22

G, G reverso, DFS e pós[]

pós[v] = numeração pós-ordem de v

Fato. Se pós[v] > pós[w] e existe um caminho de w a v, então existe um caminho de v a w.

G, G reverso, DFS e pós[]

pós[v] = numeração pós-ordem de v

Fato. Se pós[v] > pós[w] e existe um caminho de w a v, então existe um caminho de v a w.

Em outras palavras:

Fato. Se pós[v] > pós[w] e existe um caminho de w a v, então v e w estão em um mesmo componente fortemente conexo.

G, G reverso, DFS e pós[]

Algoritmo de Kosaraju: aplique DFS
no grafo reverso R de G e compute pós[].
Em seguida

G, G reverso, DFS e pós[]

Algoritmo de Kosaraju: aplique DFS
no grafo reverso R de G e compute pós[].
Em seguida

- pegue o vértice v tal que pós[v] é máximo (em ordem reversa de pós[]);

G, G reverso, DFS e pós[]

Algoritmo de Kosaraju: aplique DFS
no grafo reverso R de G e compute pós[].
Em seguida

- pegue o vértice v tal que pós[v] é máximo
  (em ordem reversa de pós[]);
- determine o conjunto
  $W = \{w : \text{existe caminho de } v \text{ a } w \text{ em } G\}$;
- para w em W, existe em R um caminho de w a v;
- **Fato** $\Rightarrow$ W forma um componente f.c.
  de R, e portanto de G;

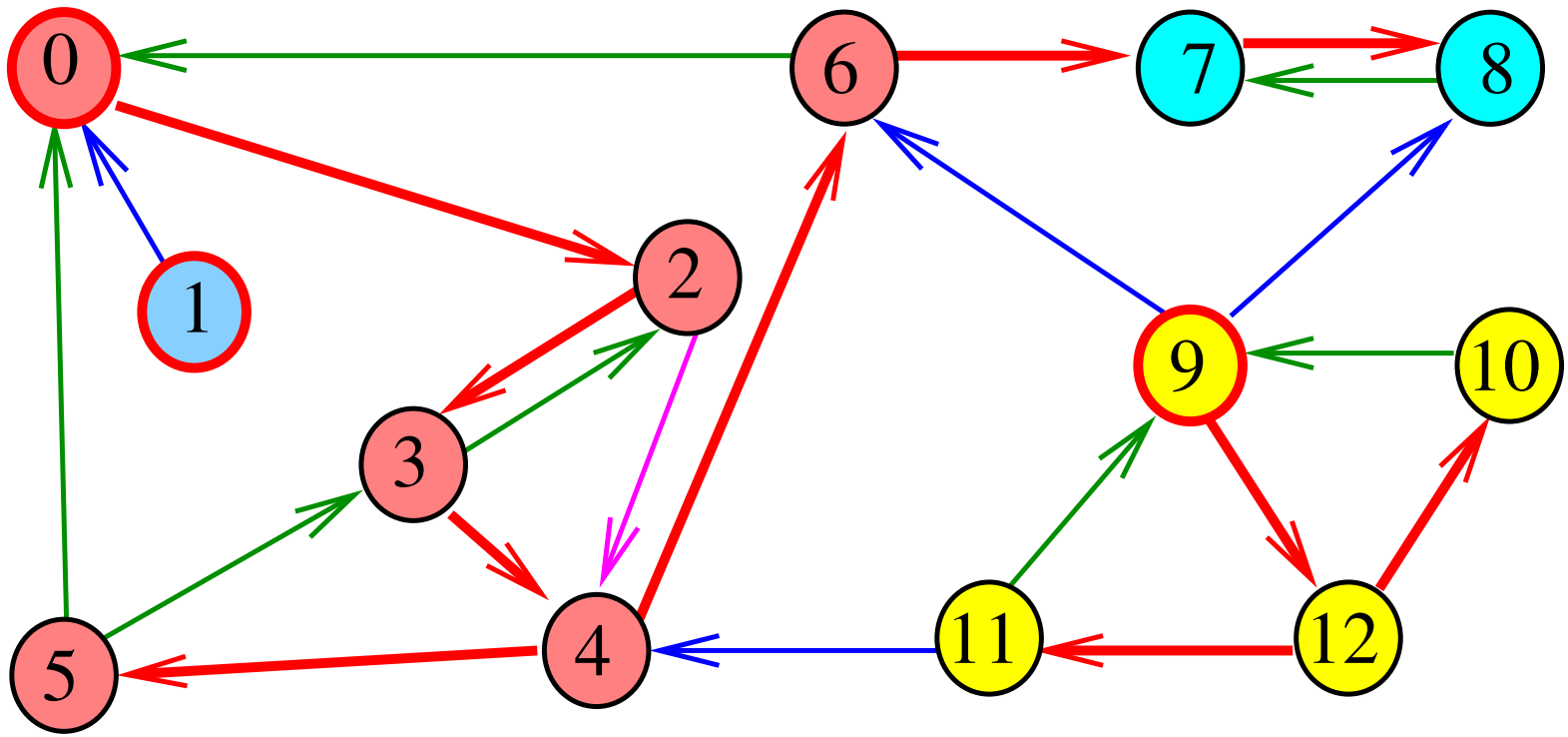
G, G reverso, DFS e pós[]

Algoritmo de Kosaraju: aplique DFS
no grafo reverso R de G e compute pós[].
Em seguida

- pegue o vértice v tal que pós[v] é máximo
  (em ordem reversa de pós[]);
- determine o conjunto
  $W = \{w : \text{existe caminho de } v \text{ a } w \text{ em } G\}$;
- para w em W, existe em R um caminho de w a v;
- **Fato** $\Rightarrow$ W forma um componente f.c.
  de R, e portanto de G;
- remova W de G e pegue o vértice v tal que ...

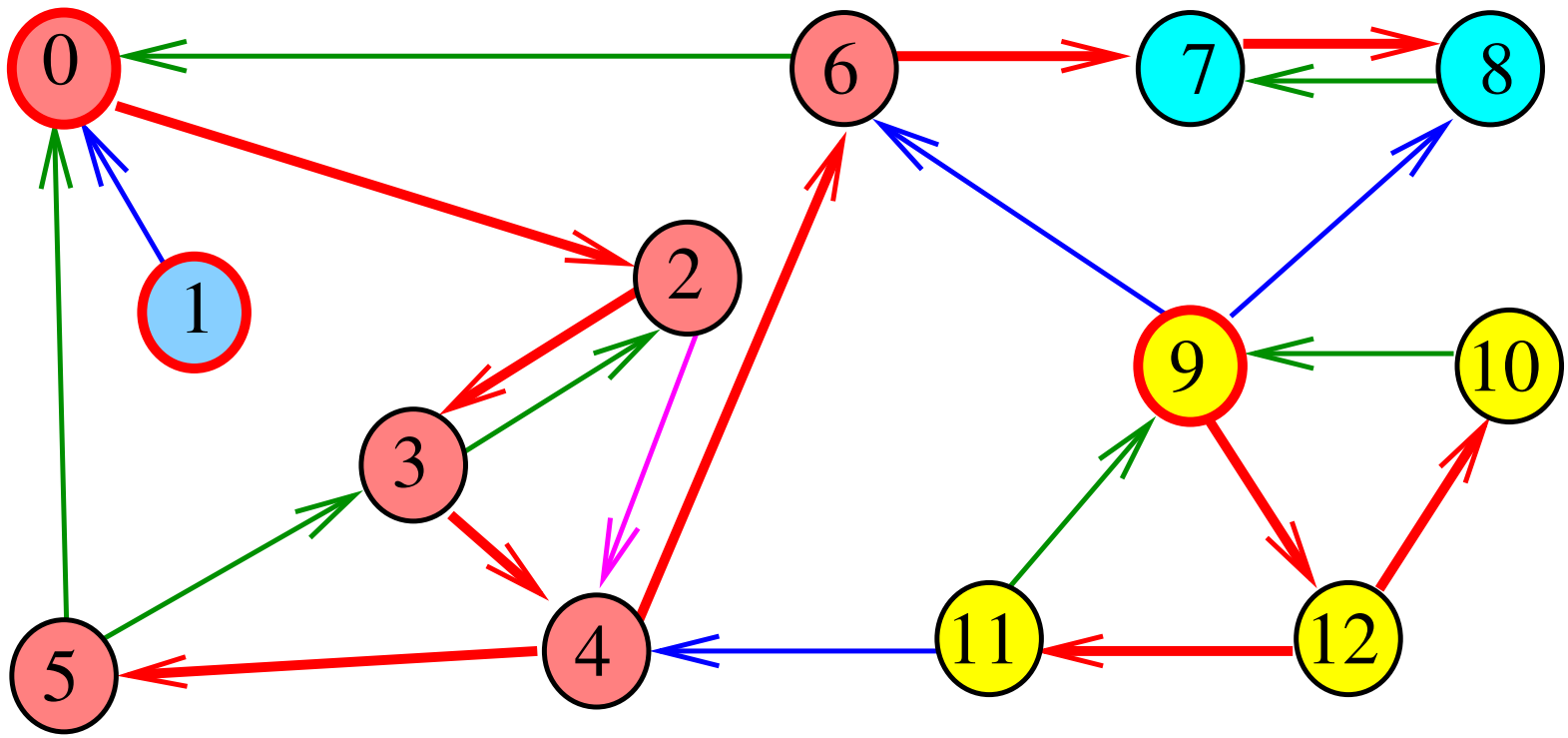
Digrafo reverso R e DFS

v	**0**	**1**	2	3	4	5	6	7	8	**9**	10	11	12
pós[v]	7	8	6	5	4	3	2	1	0	12	9	10	11



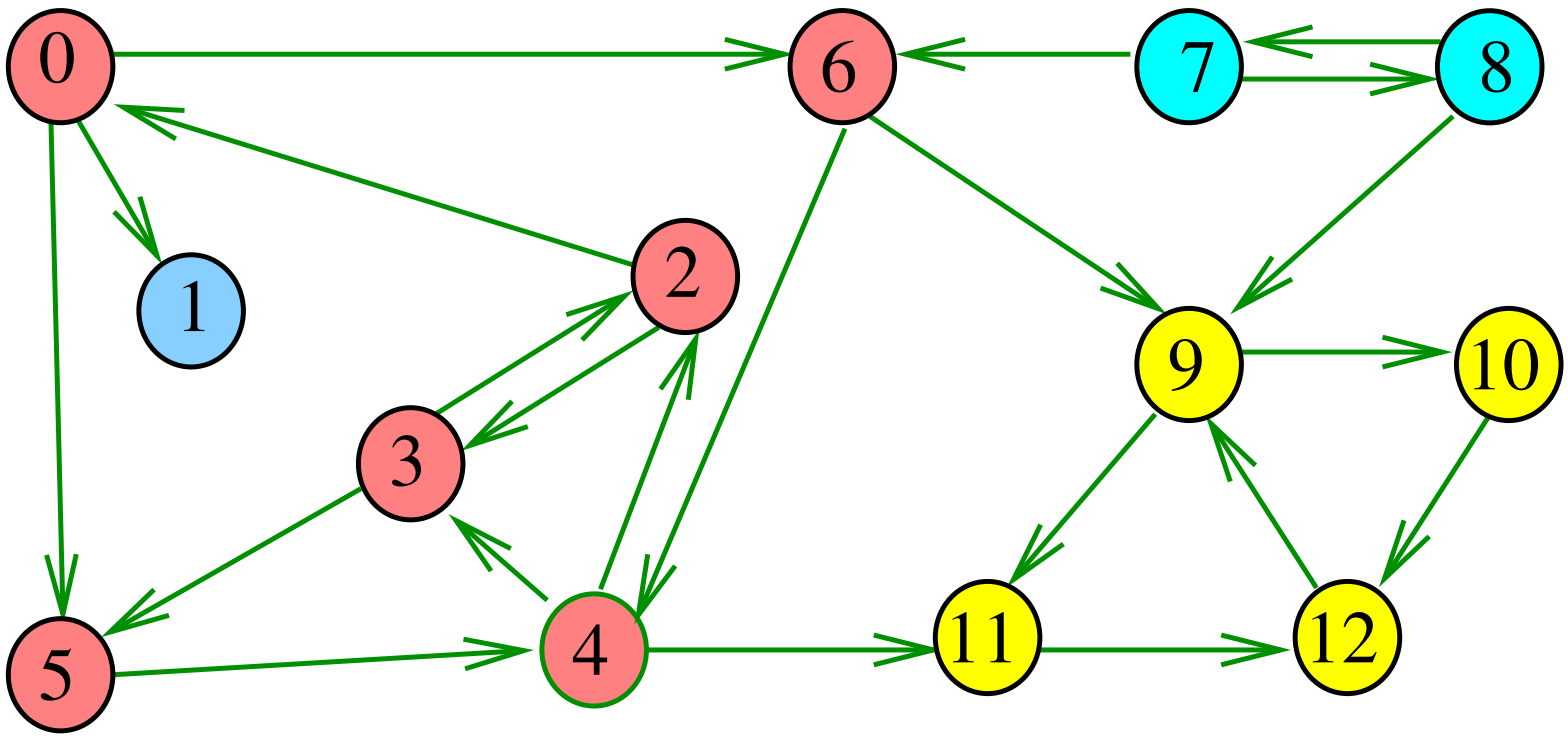
Digrafo reverso R e DFS

v	0	1	2	3	4	5	6	7	8	9	10	11	12
pós[v]	7	8	6	5	4	3	2	1	0	12	9	10	11
i	0	1	2	3	4	5	6	7	8	9	10	11	12
sóp[i]	8	7	6	5	4	3	2	0	1	10	11	12	9



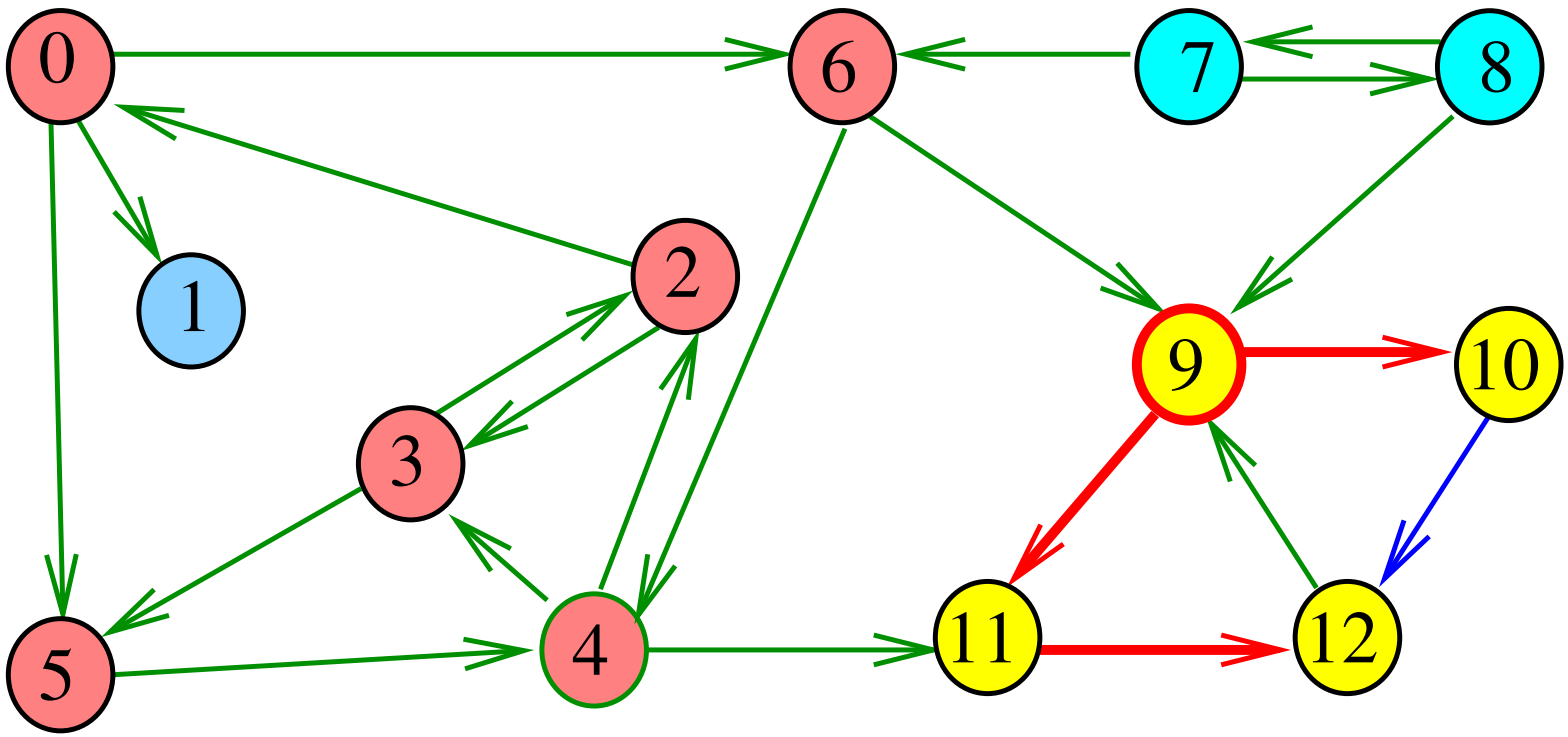
Digrafo G e DFS

i	0	1	2	3	4	5	6	7	8	9	10	11	12
sóp[i]	8	7	6	5	4	3	2	0	1	10	11	12	9



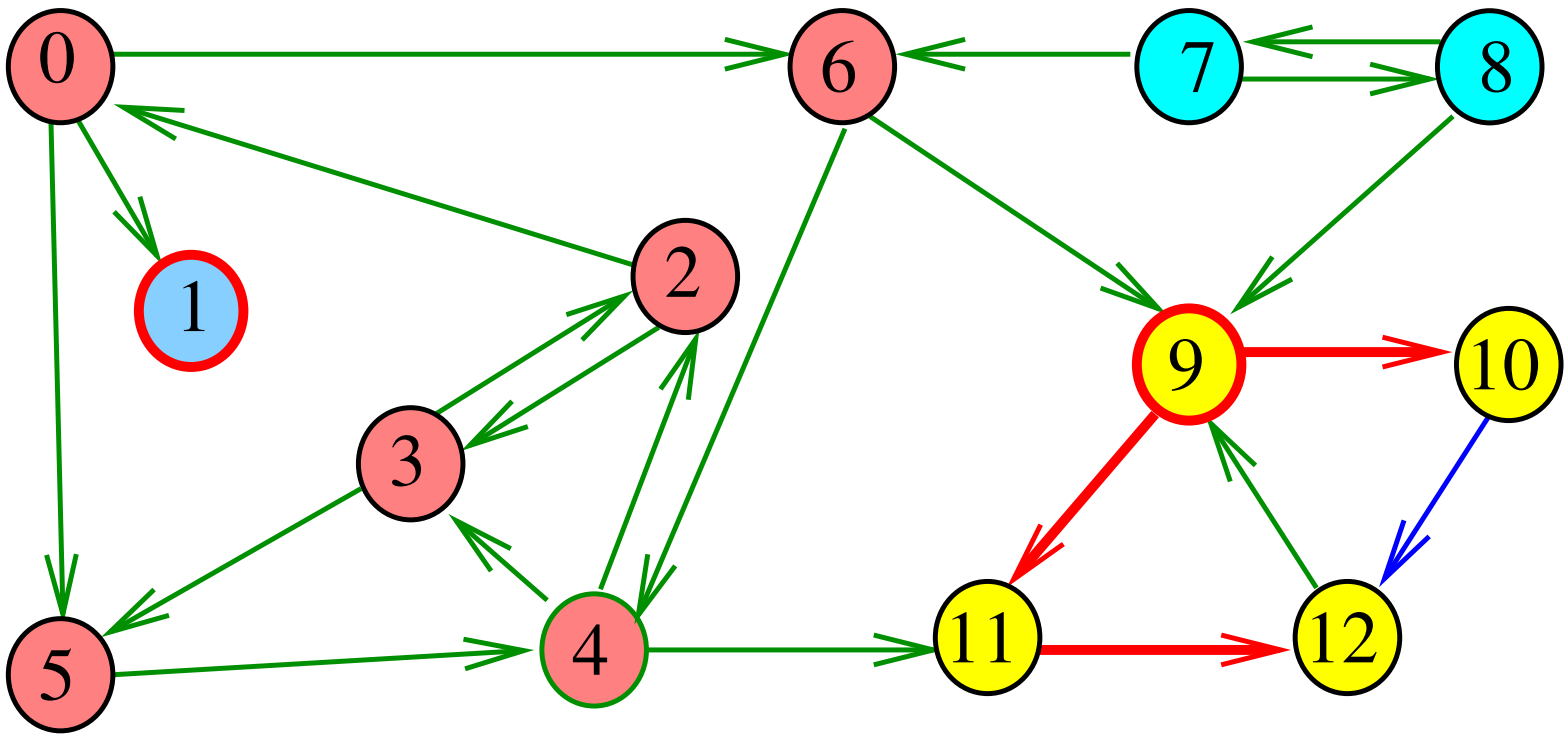
Digrafo G e DFS

i	0	1	2	3	4	5	6	7	8	9	10	11	**12**
sóp[i]	8	7	6	5	4	3	2	0	1	**10**	**11**	**12**	**9**



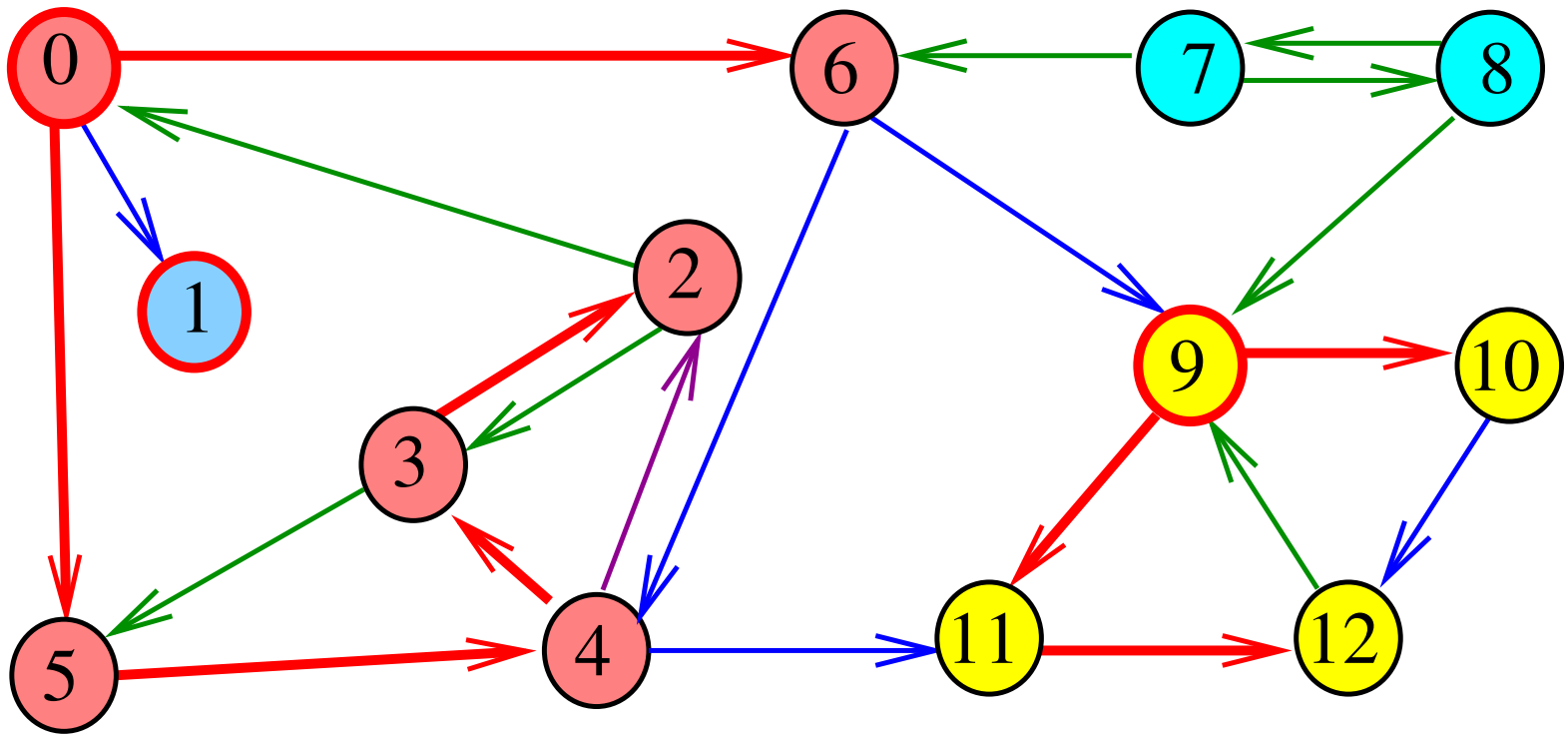
Digrafo G e DFS

i	0	1	2	3	4	5	6	7	**8**	9	10	11	**12**
sóp[i]	8	7	6	5	4	3	2	0	**1**	**10**	**11**	**12**	**9**



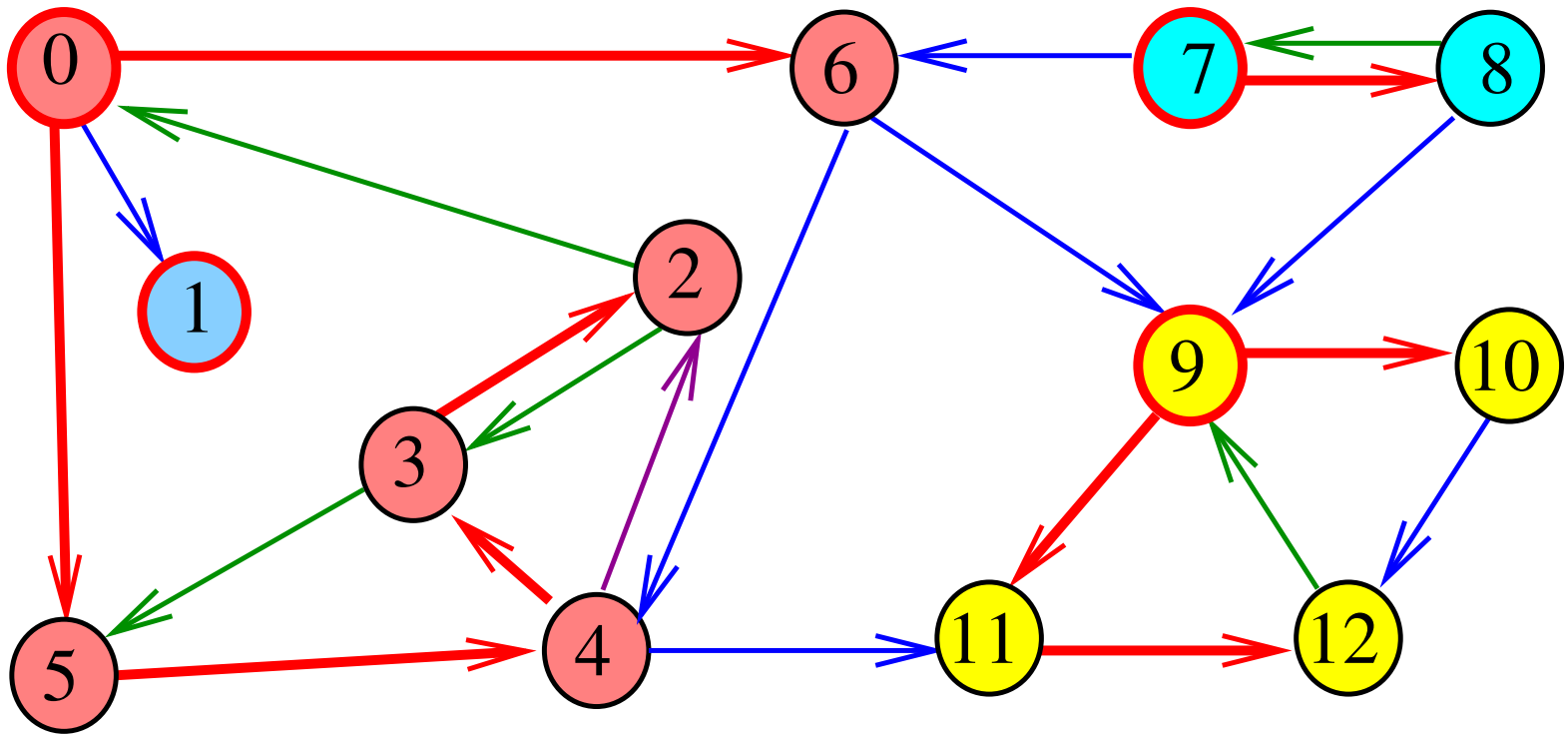
Digrafo G e DFS

i	0	1	2	3	4	5	6	7	8	9	10	11	12
sóp[i]	8	7	6	5	4	3	2	0	1	10	11	12	9



Digrafo G e DFS

i	0	1	2	3	4	5	6	7	8	9	10	11	12
sóp[i]	8	7	6	5	4	3	2	0	1	10	11	12	9



Algoritmo de Kosaraju e Sharir

A rotina DFSscc calcula
os componentes fortemente conexos do digrafo G.

```
static bool *marked;
static int  *id;
static int  count; /* no. de scc */
```

Ela armazena no vetor id[] o número
do componente a que o vértice pertence:
se o vértice v pertence ao k-ésimo componente
então id[v] == k-1.

Classe DFSscc: esqueleto

```
void DFSsccInit(Digraph G) {
   marked = mallocSafe(G->V*sizeof(bool));
   for (v = 0; v < G->V; v++)
      marked[v] = false;
   count = 0;
   id = mallocSafe(G->V*sizeof(int));
}
void DFSscc(Graph G) {...}

static void dfs(Digraph G, int v) {...}

bool sConnected(int v, int w) {...}

int SCCid(int v) {...}

int SCCcount(int v) {...}
```

DFSscc

```
void DFSscc(Digraph G) {
   /* computa uma pós-ordem reversa */
   DFSanatomia anaR;
   anaR = DFSanatomia(reverse(G));
   /* contrói floresta DFS de G */
   DFSsccInit(G);
   while (!stackEmpty(anaR->revPos)) {
      v = pop(anaR->revPos);
      if (!marked[v]) {
         dfs(G, v);
         count++;
      }
   }
}
```

DFSscc: dfs()

```
/* DFS no digrafo G */

static void dfs(Digraph G, int v) {
   Link w;
   marked[v] = true;
   id[v] = count;
   for (w = G->adj[v]; w != NULL; w = w->next)
      if(!marked[w->vertex])
         dfs(G, w->vertex);
   }
}
```

DFSscc

```
/* no. de comps fortemente conexos */
int SCCcount() {
   return count;
}

/* v e w estão no mesmo comp. f.c.? */
bool sConnected(int v, int w) {
   return id[v] == id[w];
}

/* id do comp. fort. conexo de v */
int SCCid(int v) {
   return id[v];
}
```

Digraph: reverse(G)

```
/* retorna o digrafo reverso */

Digraph reverse(Digraph G) {
   Digraph reverse = newDigraph(G->V);
   int v; Link u;
   for (v = 0; v < G->V; v++) {
      for (u = G->adj[v]; u != NULL; u = u->next)
         addEdge(reverse, u->vertex, v);
   }
   return reverse;
}
```

Consumo de tempo

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

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

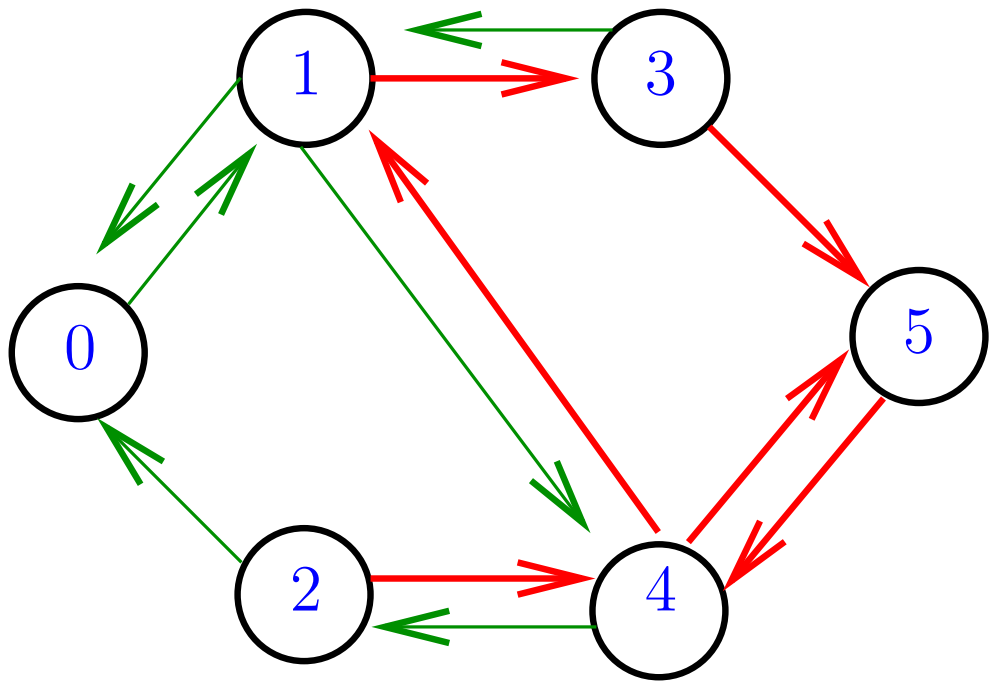
Caminhos mínimos

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

Comprimento

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

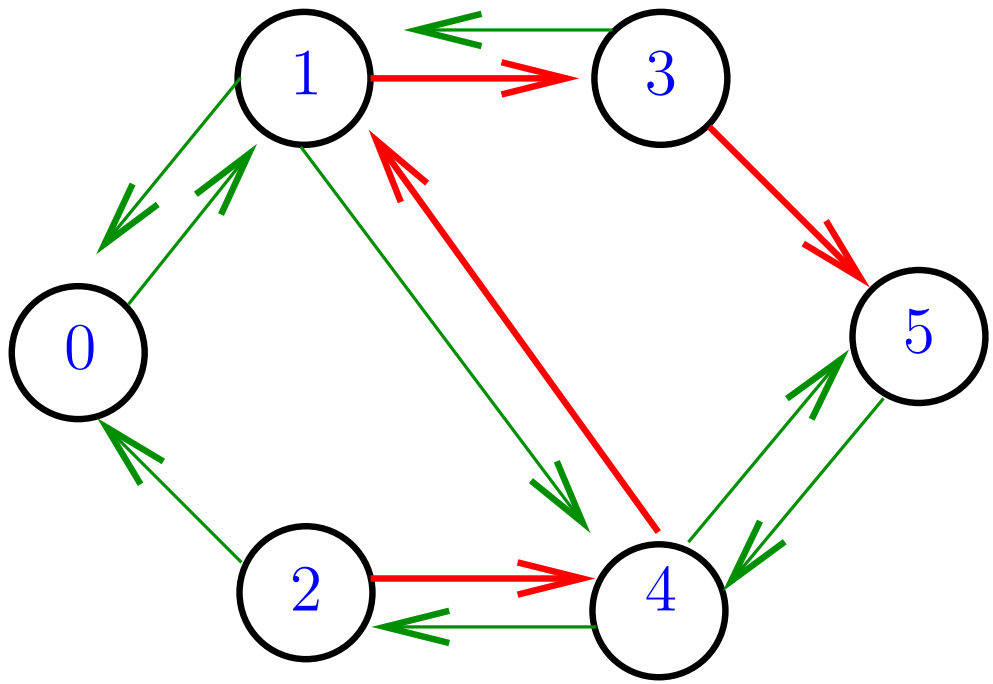
Exemplo: 2-4-1-3-5-4-5 tem comprimento 6



Comprimento

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

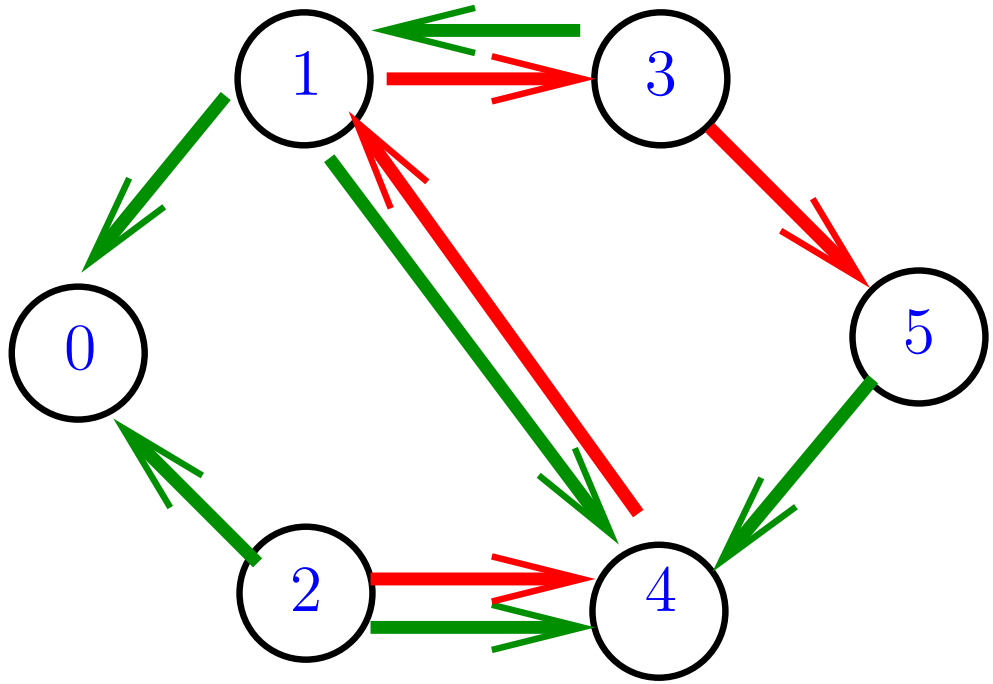
Exemplo: 2-4-1-3-5 tem comprimento 4



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**.

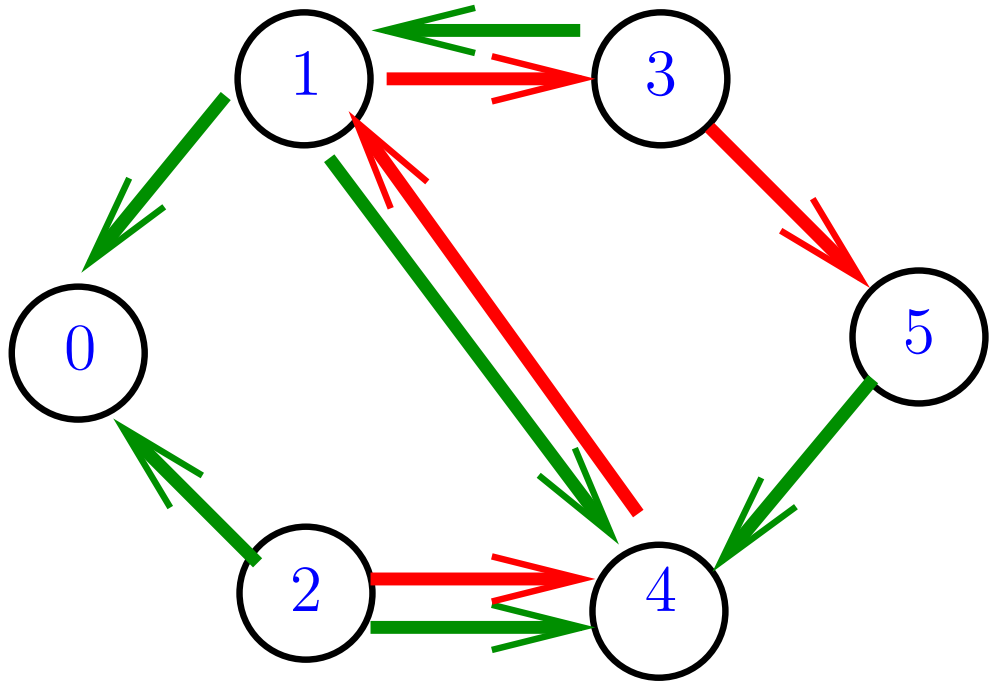
Exemplo: a distância de 2 a 5 é 4



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

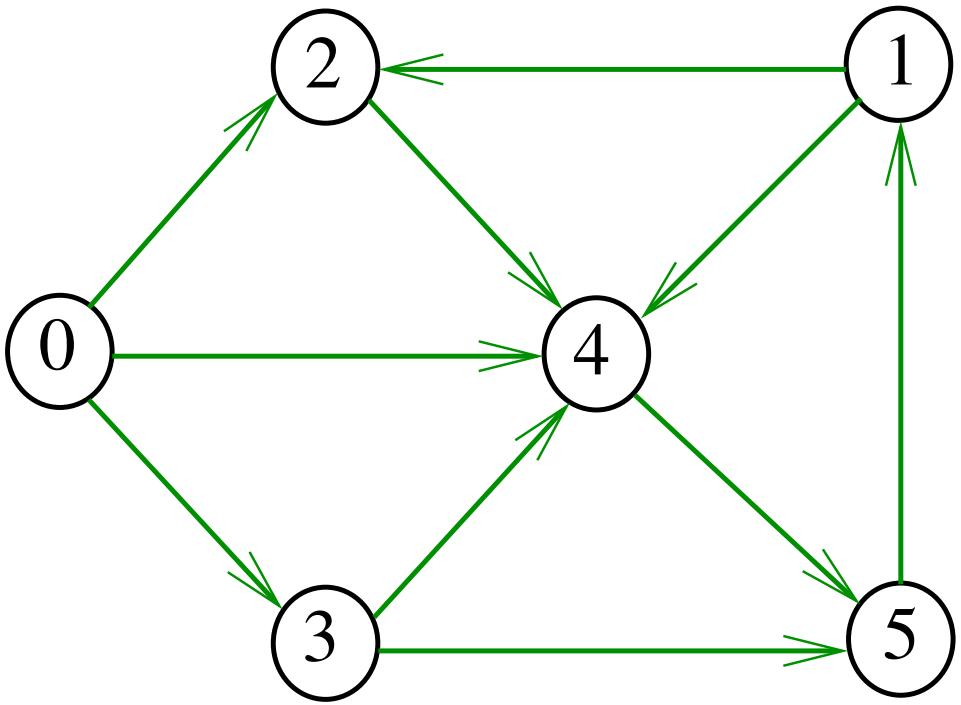


Calculando distâncias

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

Exemplo: para `s` = 0

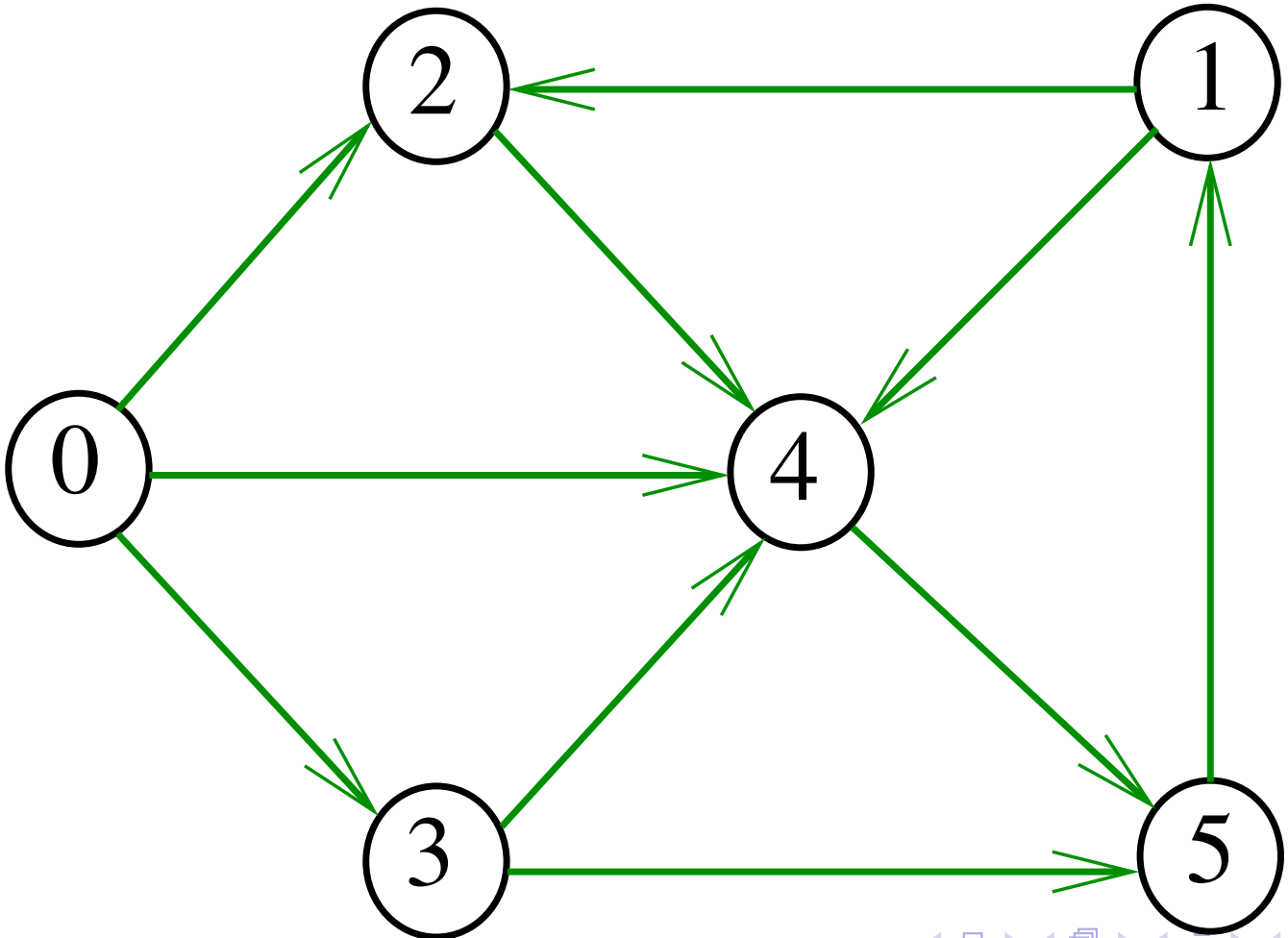
`v`	0	1	2	3	4	5
`distTo[v]`	0	3	1	1	1	2



Simulação

i	0	1	2	3	4	5
q[i]						

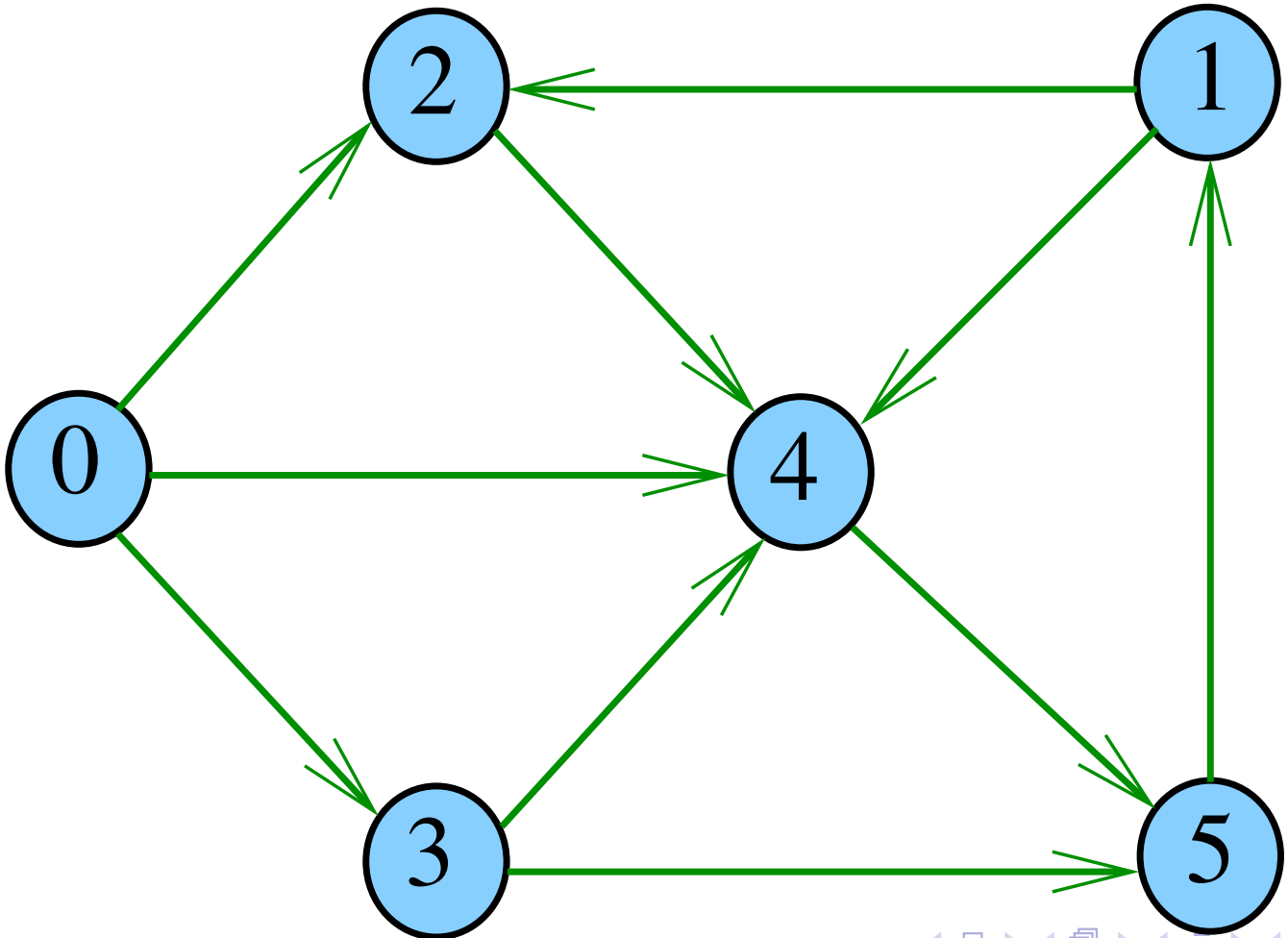
v	0	1	2	3	4	5
distTo[v]						



Simulação

i	0	1	2	3	4	5
q[i]						

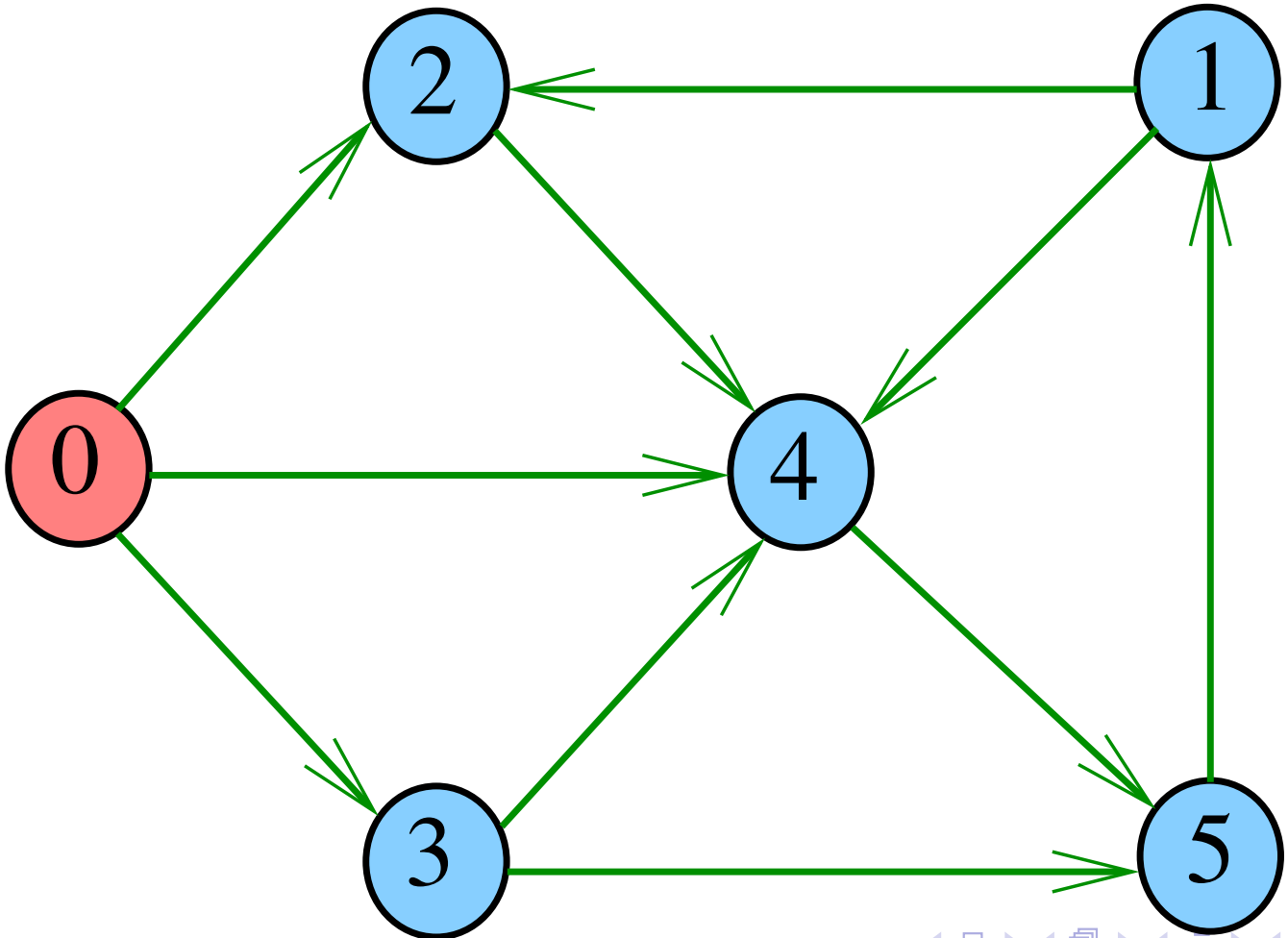
v	0	1	2	3	4	5
distTo[v]	6	6	6	6	6	6



Simulação

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

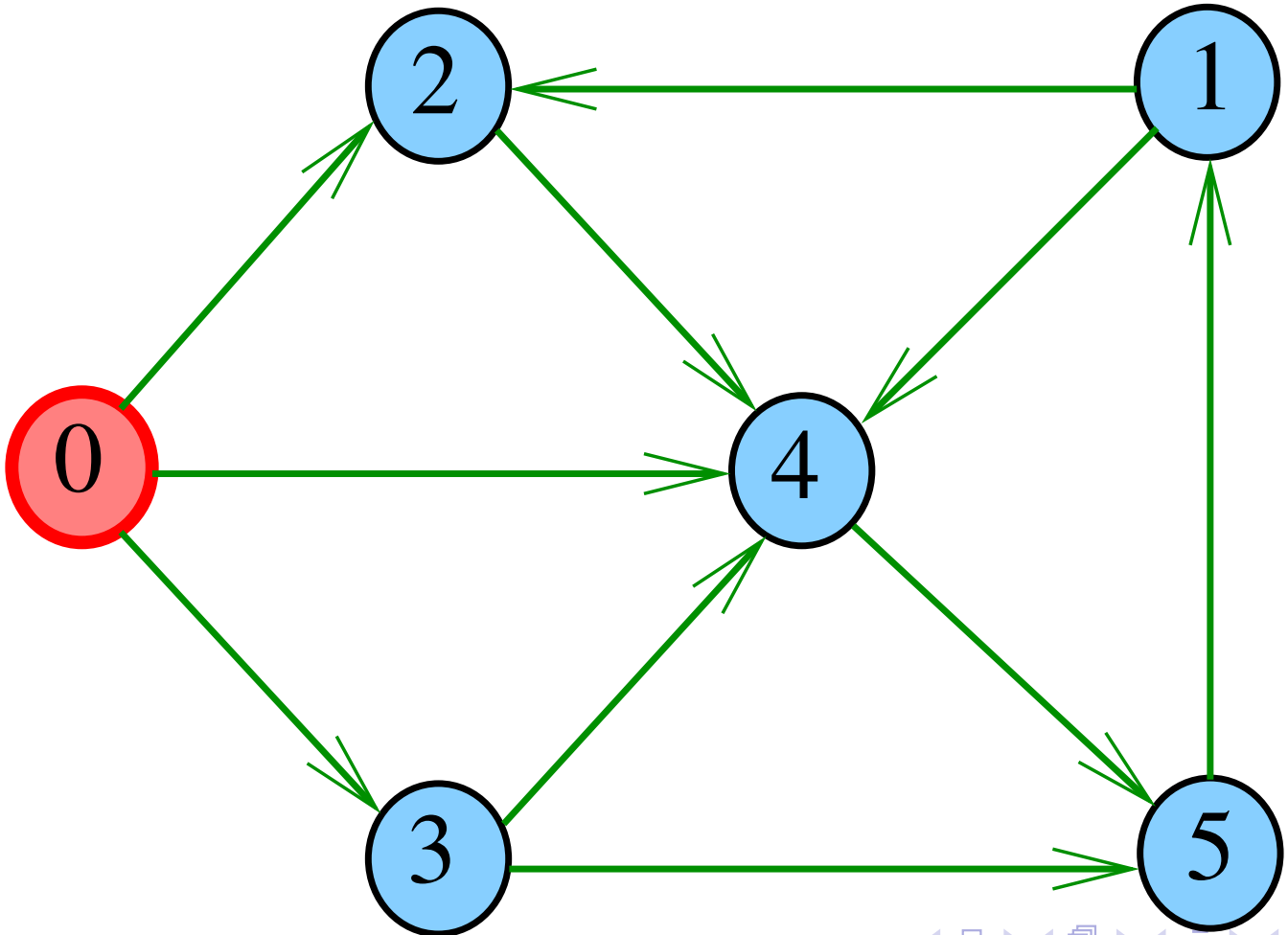
v	0	1	2	3	4	5
distTo[v]	6	6	6	6	6	6



Simulação

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

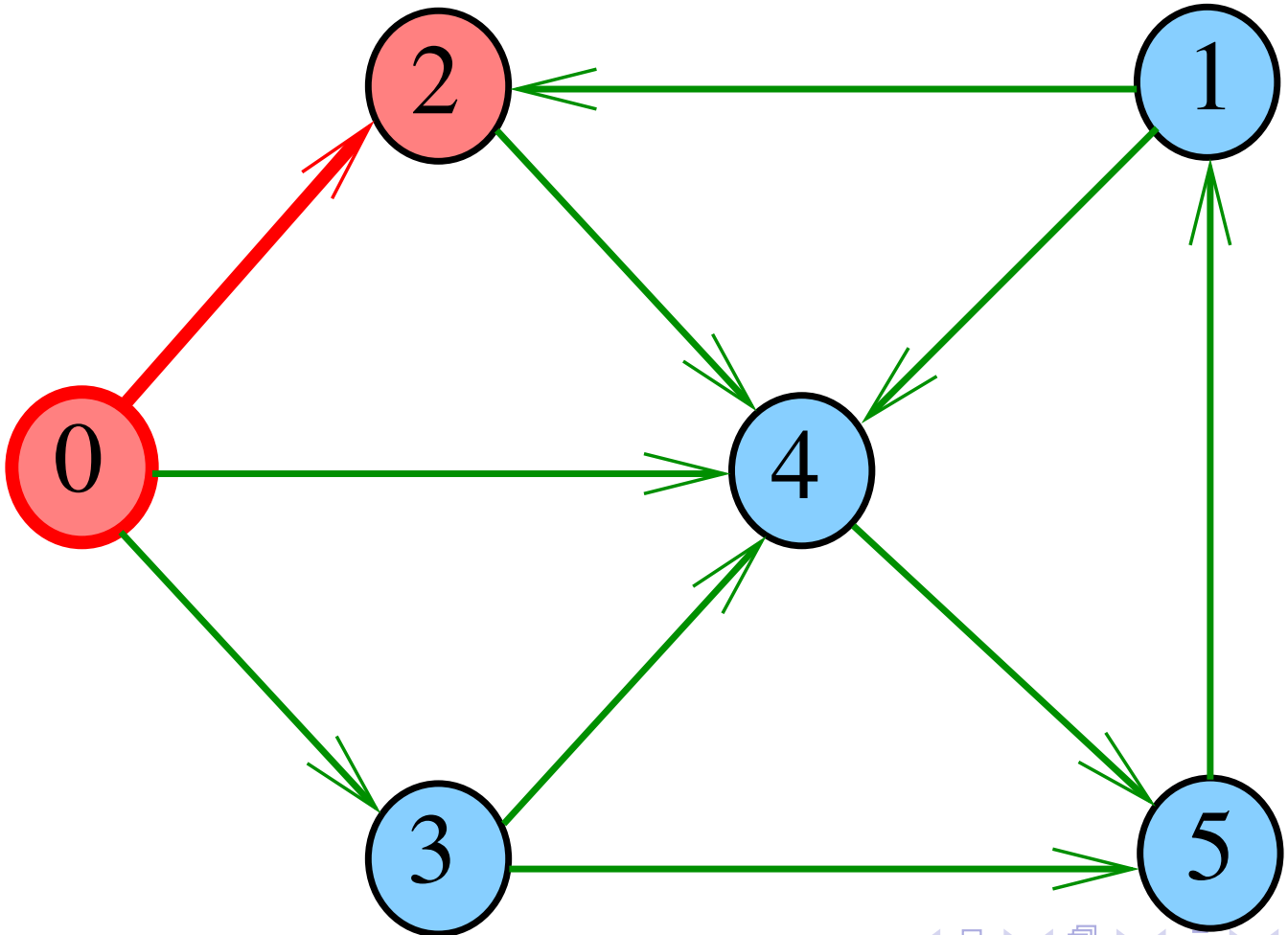
v	0	1	2	3	4	5
distTo[v]	**0**	6	6	6	6	6



Simulação

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

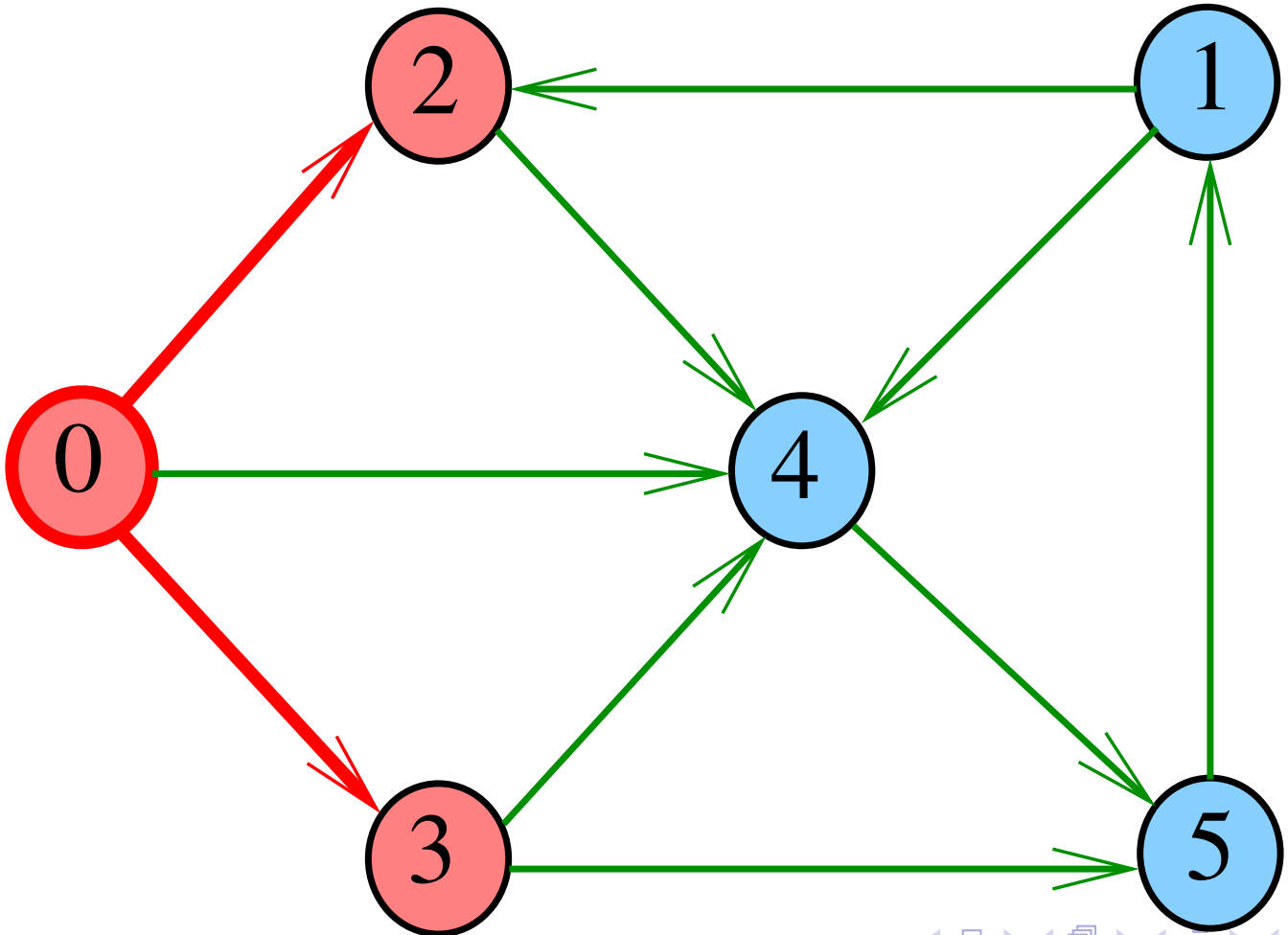
v	0	1	2	3	4	5
distTo[v]	0	6	**1**	6	6	6



Simulação

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

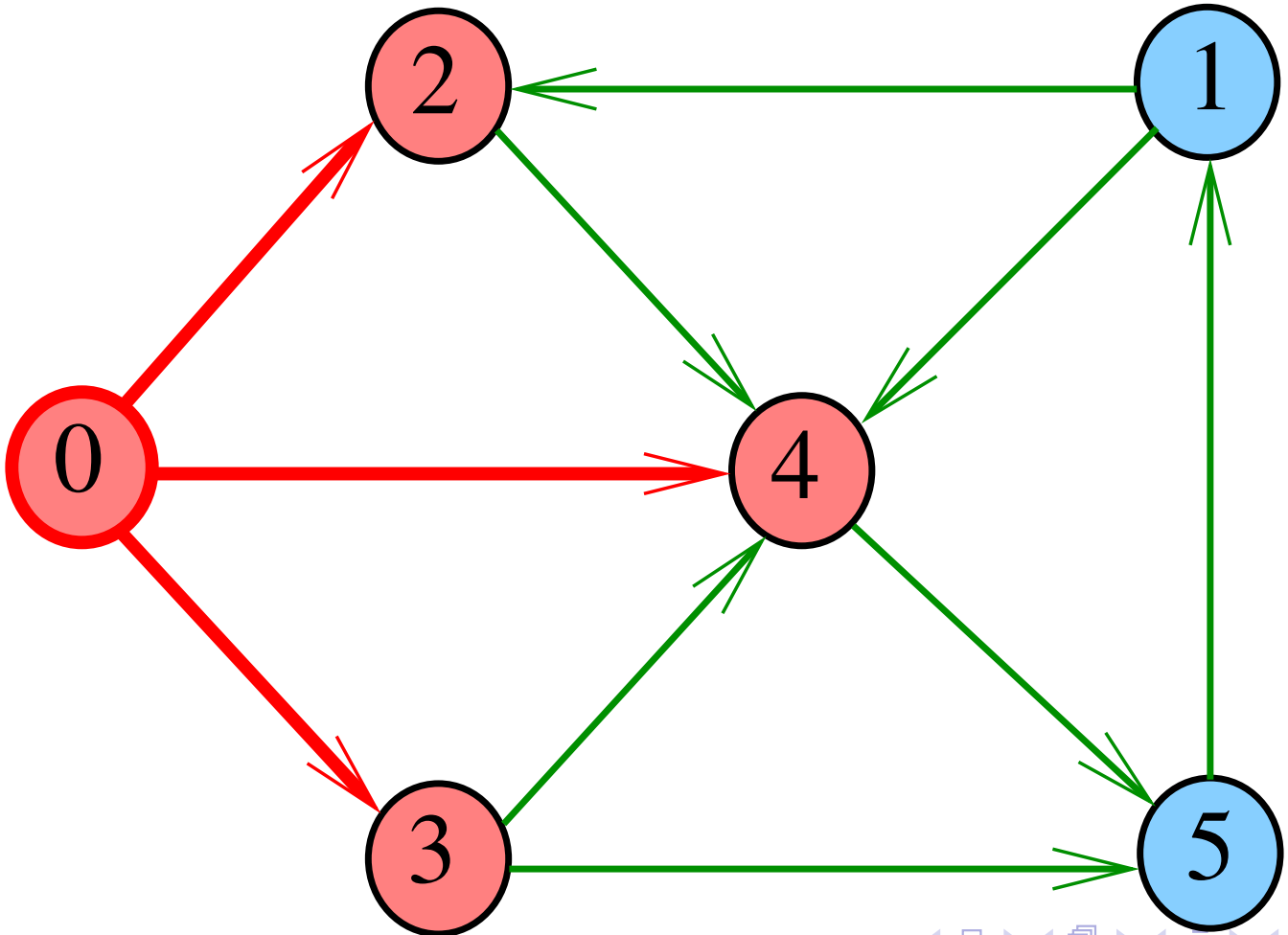
v	0	1	2	3	4	5
distTo[v]	0	6	1	**1**	6	6



Simulação

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

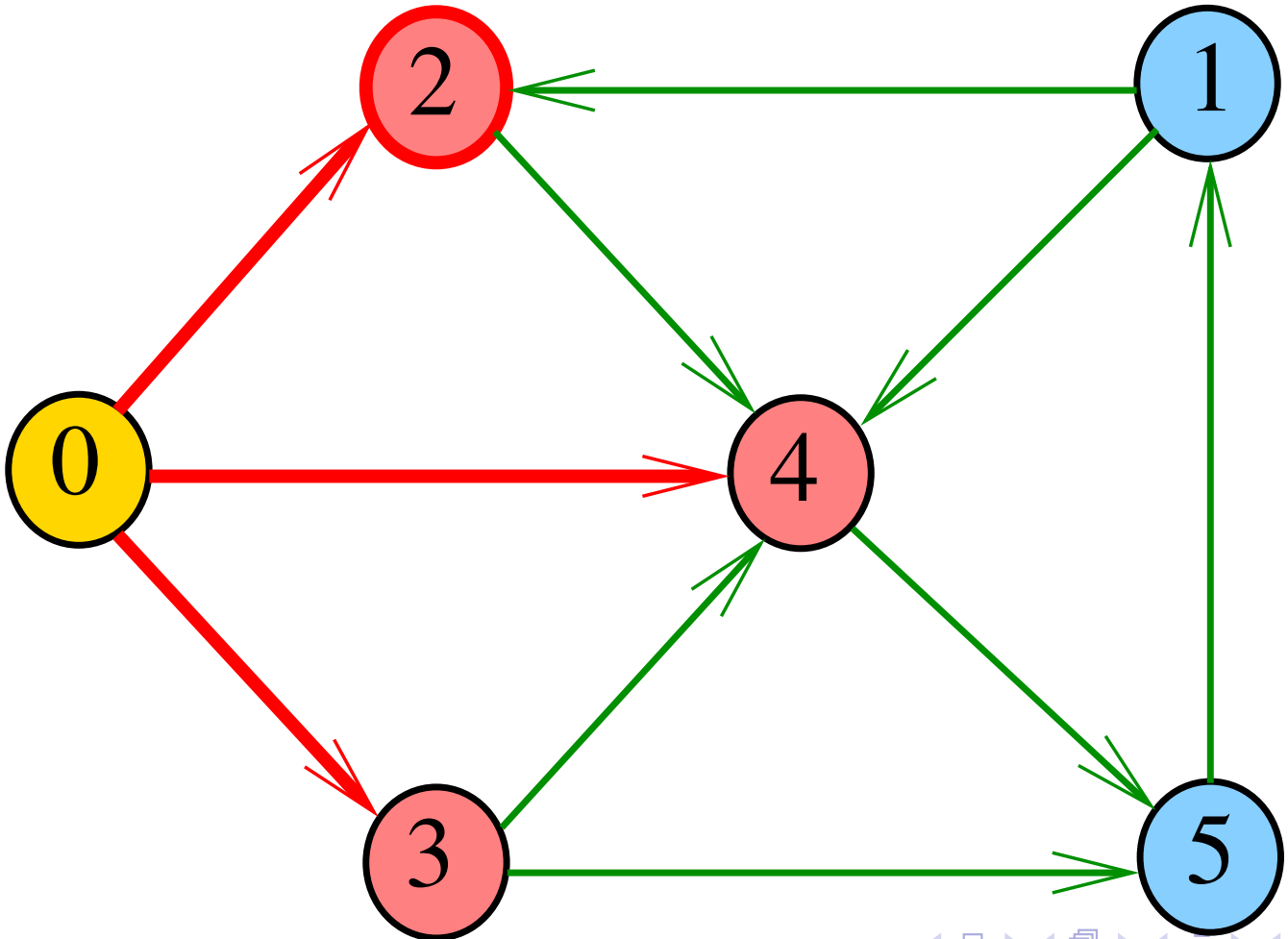
v	0	1	2	3	4	5
distTo[v]	0	6	1	1	**1**	6



Simulação

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

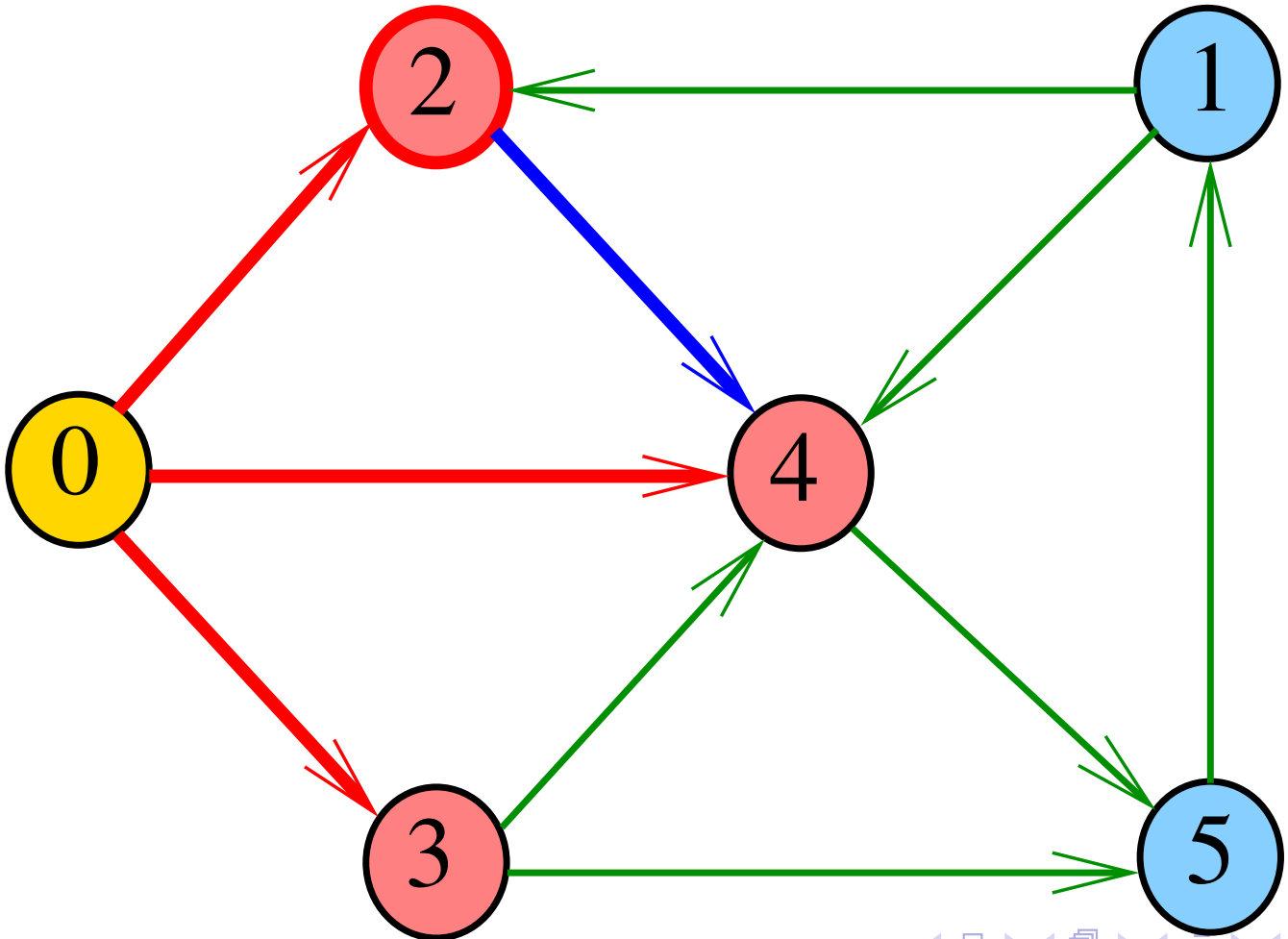
v	0	1	2	3	4	5
distTo[v]	0	6	1	1	1	6



Simulação

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

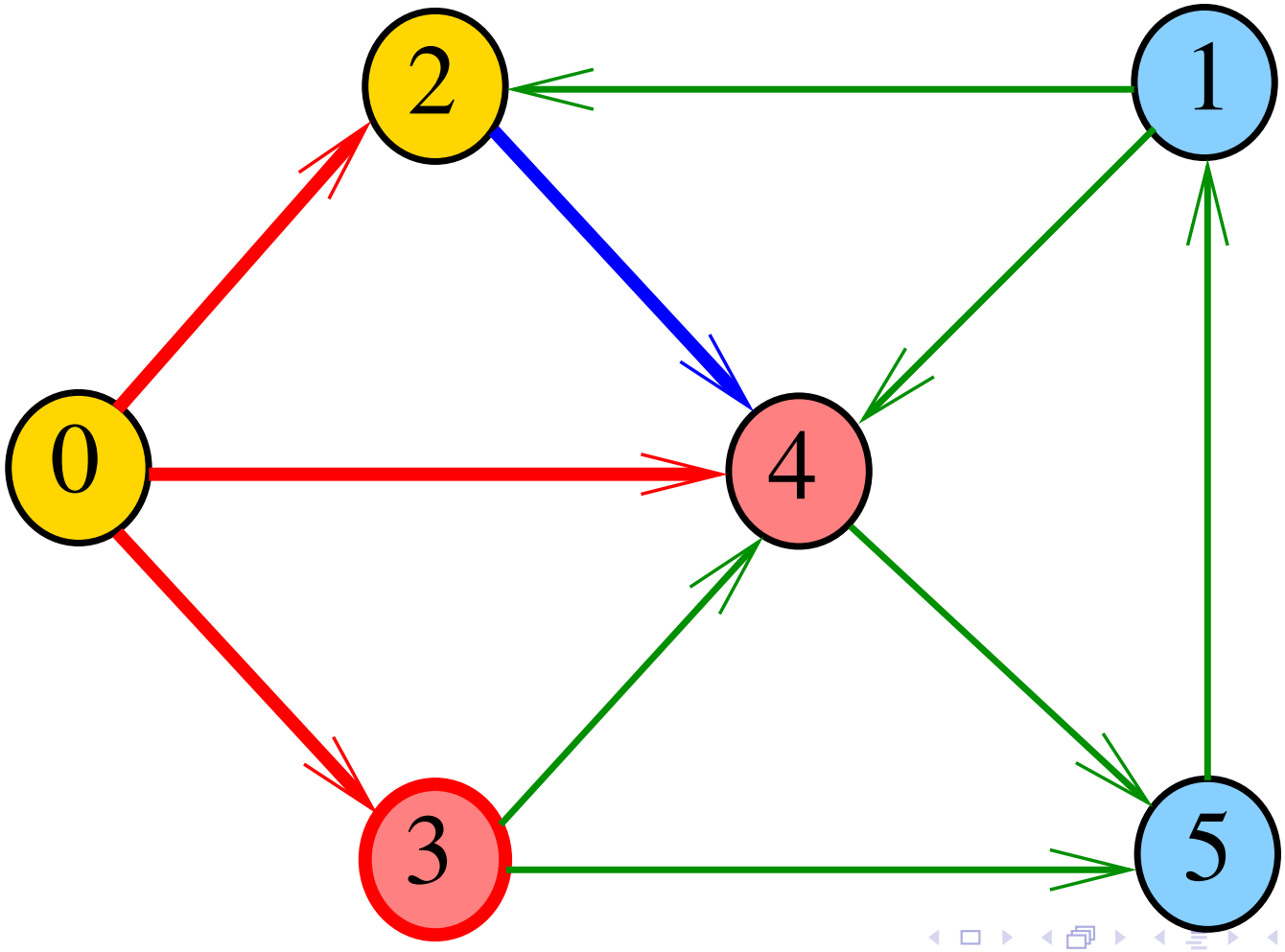
v	0	1	2	3	4	5
distTo[v]	0	6	1	1	1	6



Simulação

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

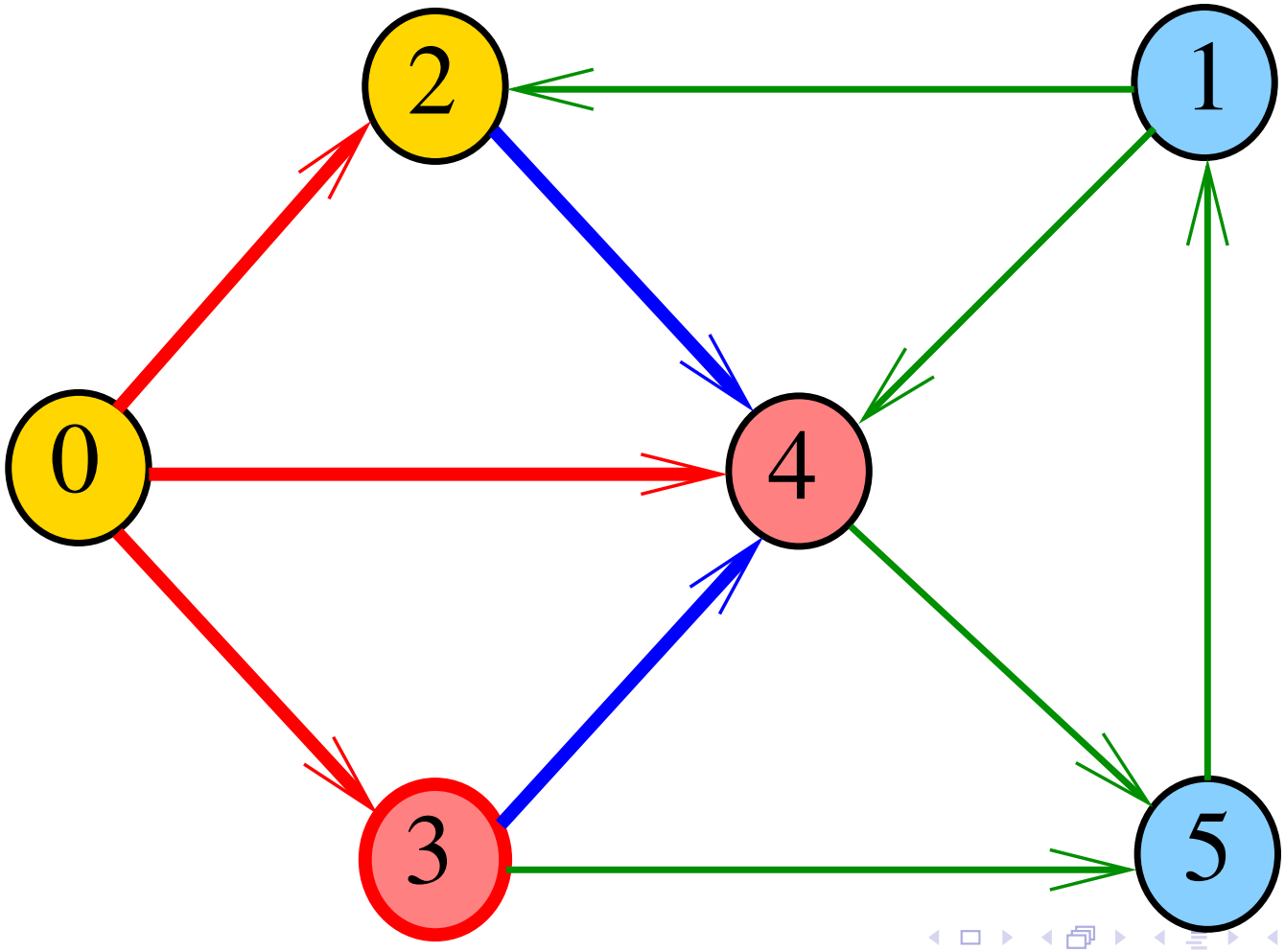
v	0	1	2	3	4	5
distTo[v]	0	6	1	1	1	6



Simulação

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

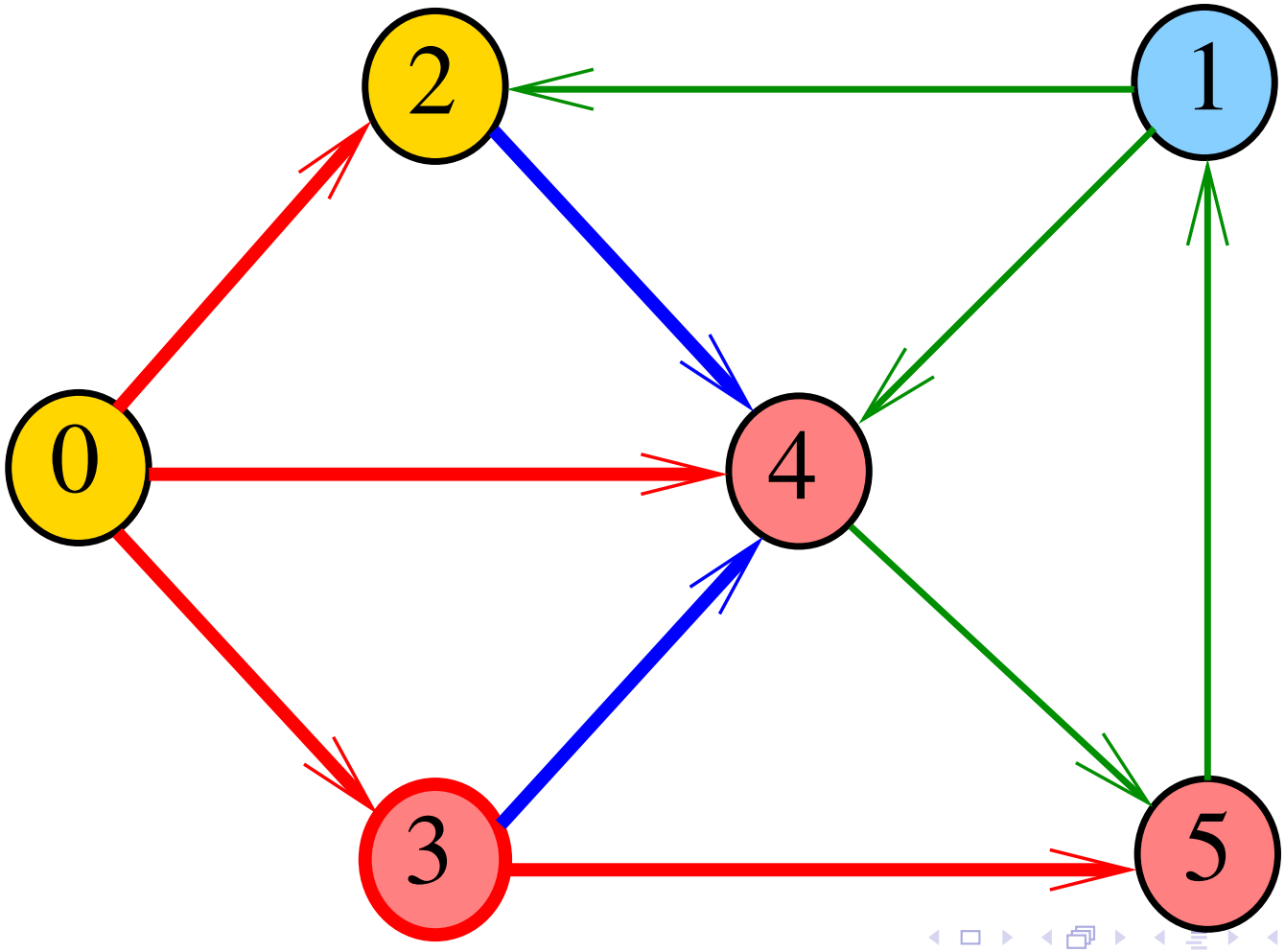
v	0	1	2	3	4	5
distTo[v]	0	6	1	1	1	6



Simulação

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

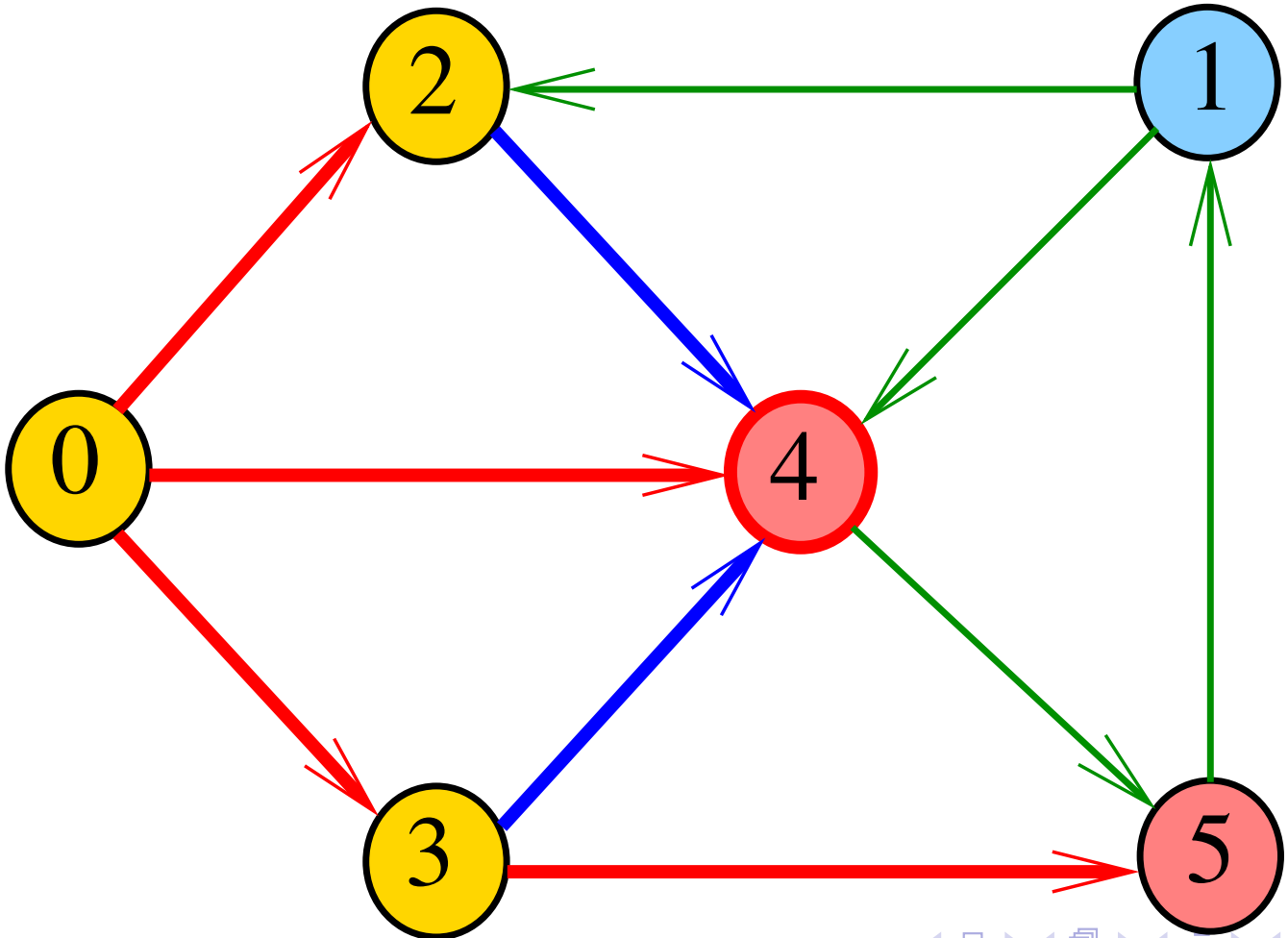
v	0	1	2	3	4	5
distTo[v]	0	6	1	1	1	2



Simulação

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

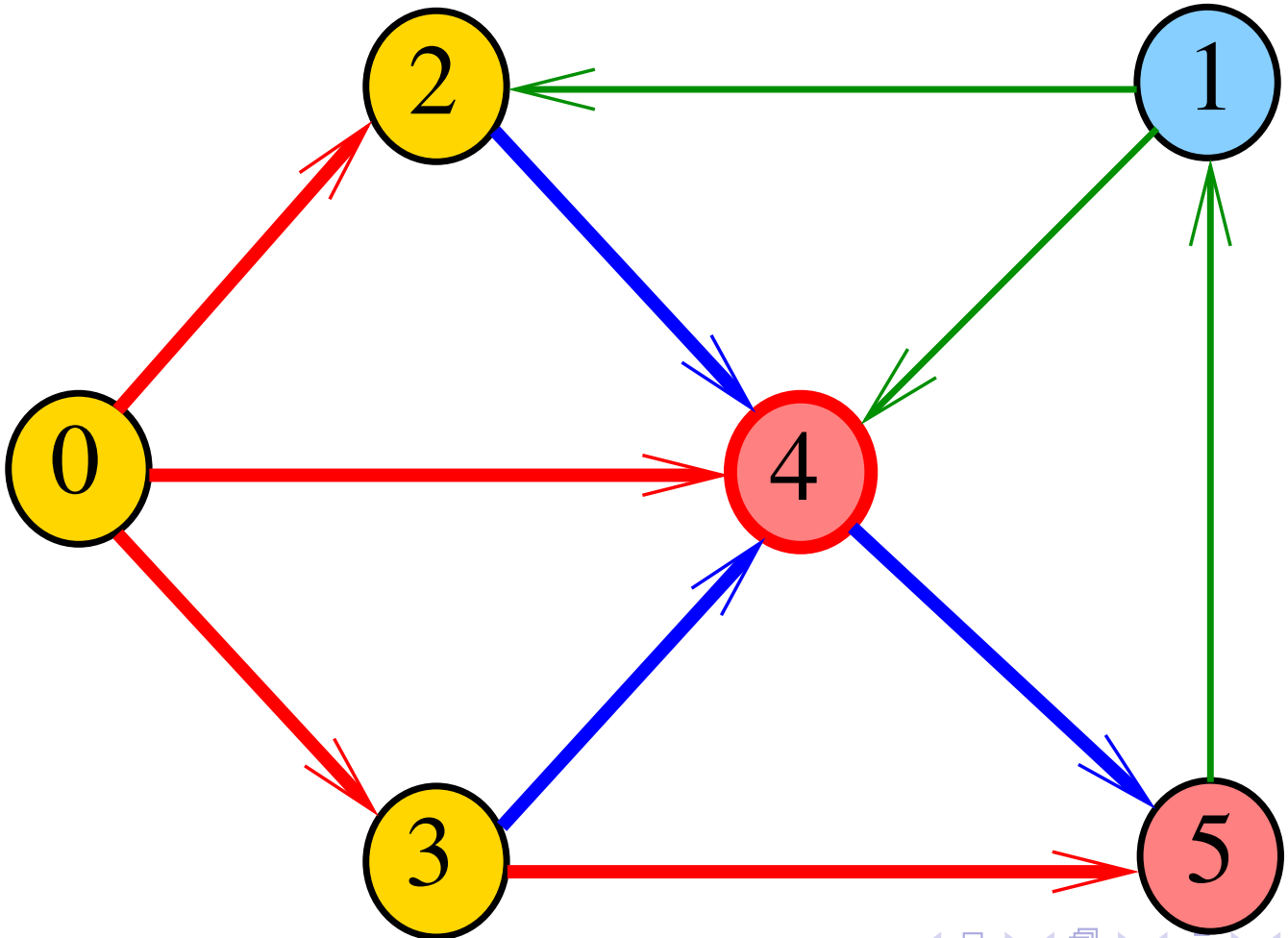
v	0	1	2	3	4	5
distTo[v]	0	6	1	1	1	2



Simulação

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

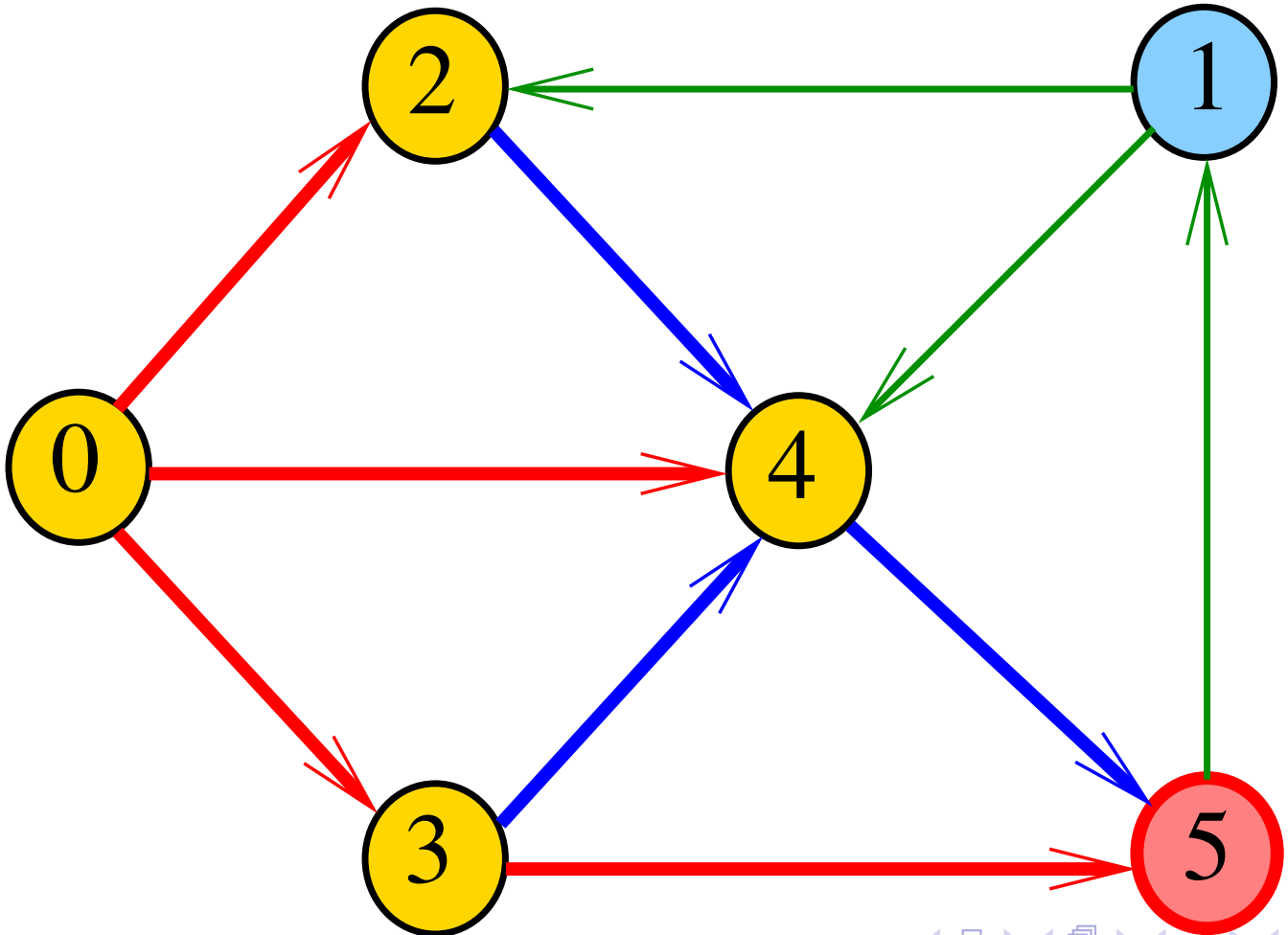
v	0	1	2	3	4	5
distTo[v]	0	6	1	1	1	2



Simulação

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

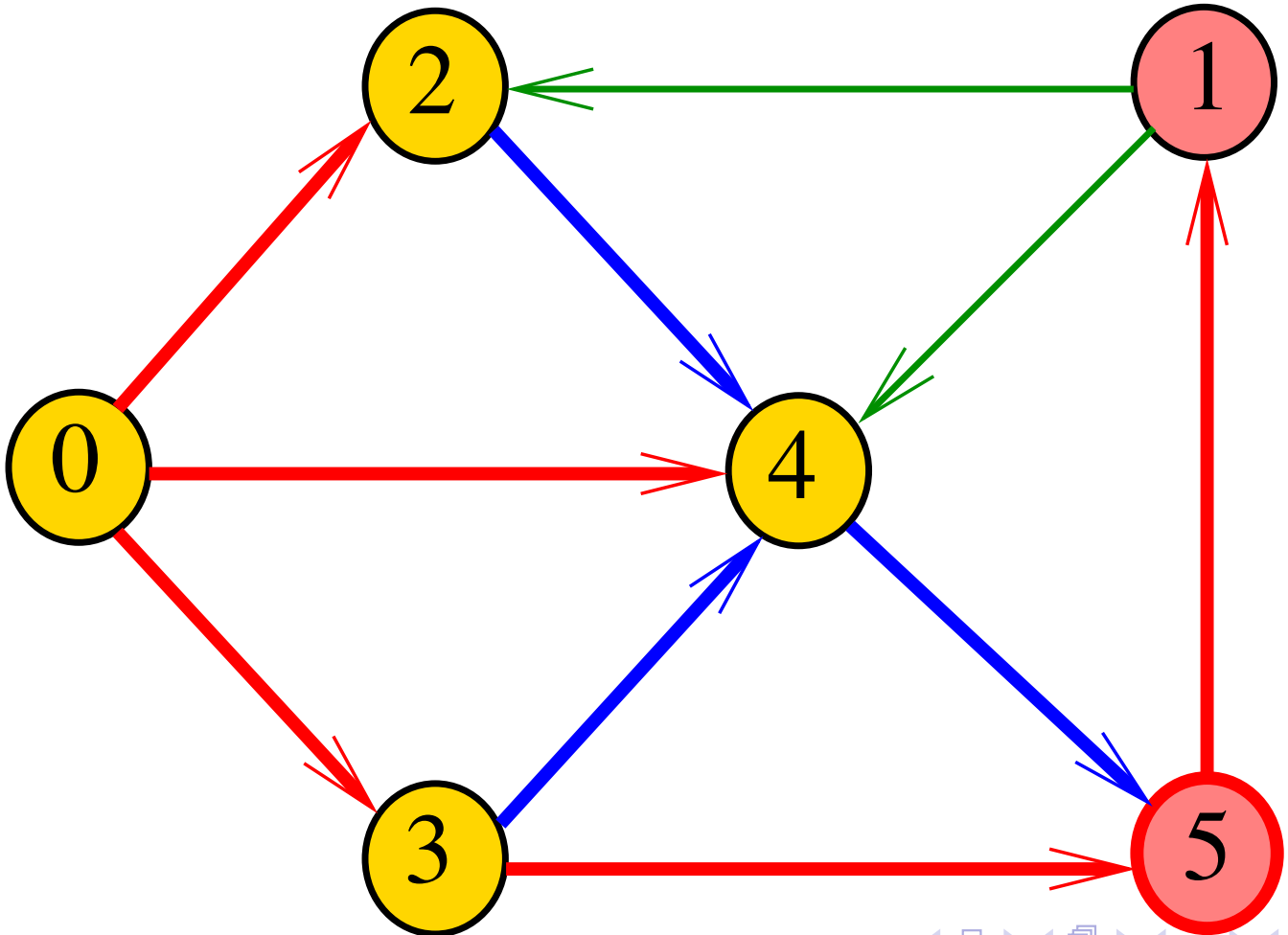
v	0	1	2	3	4	5
distTo[v]	0	6	1	1	1	2



Simulação

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

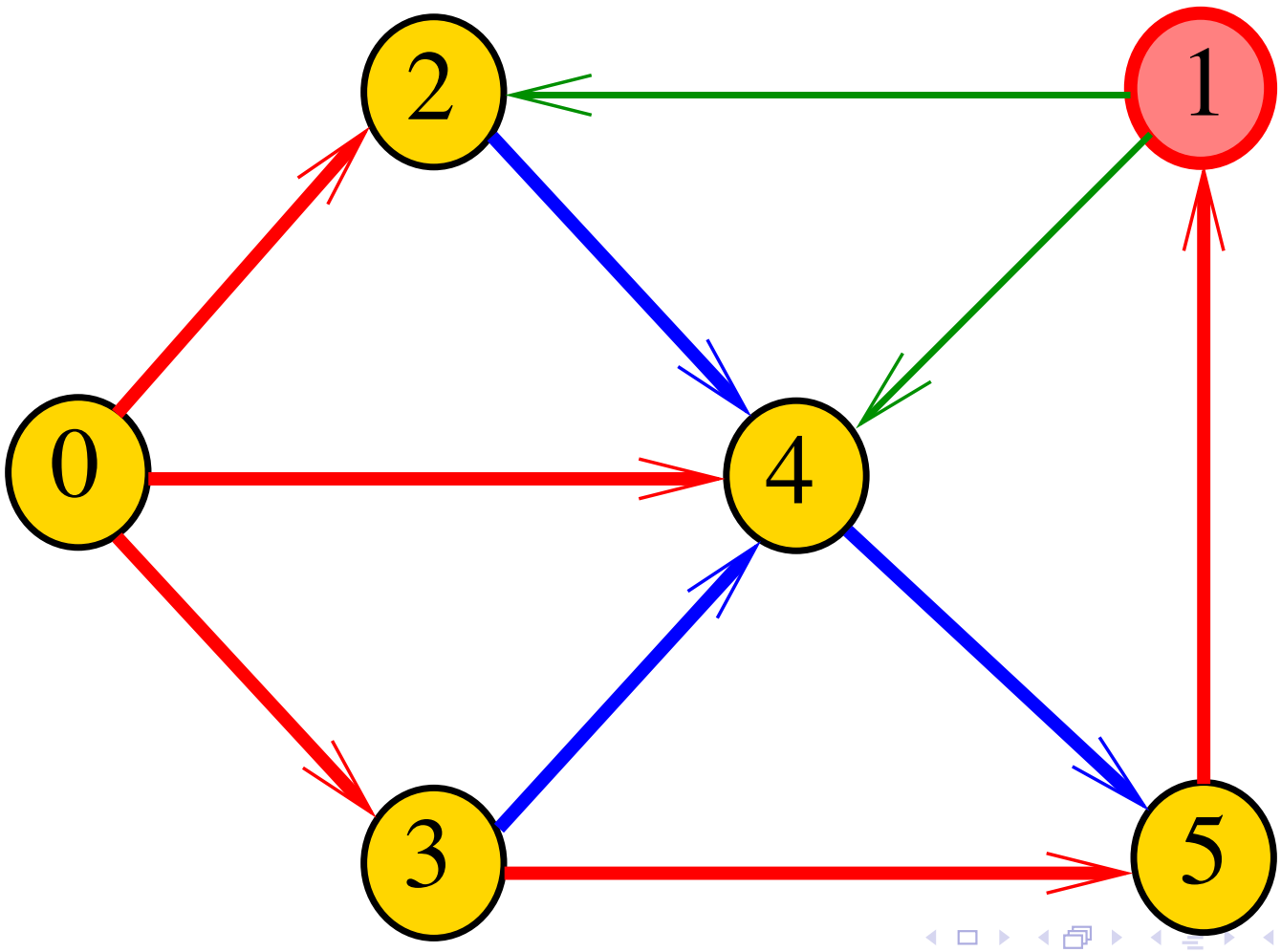
v	0	1	2	3	4	5
distTo[v]	0	**3**	1	1	1	2



Simulação

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

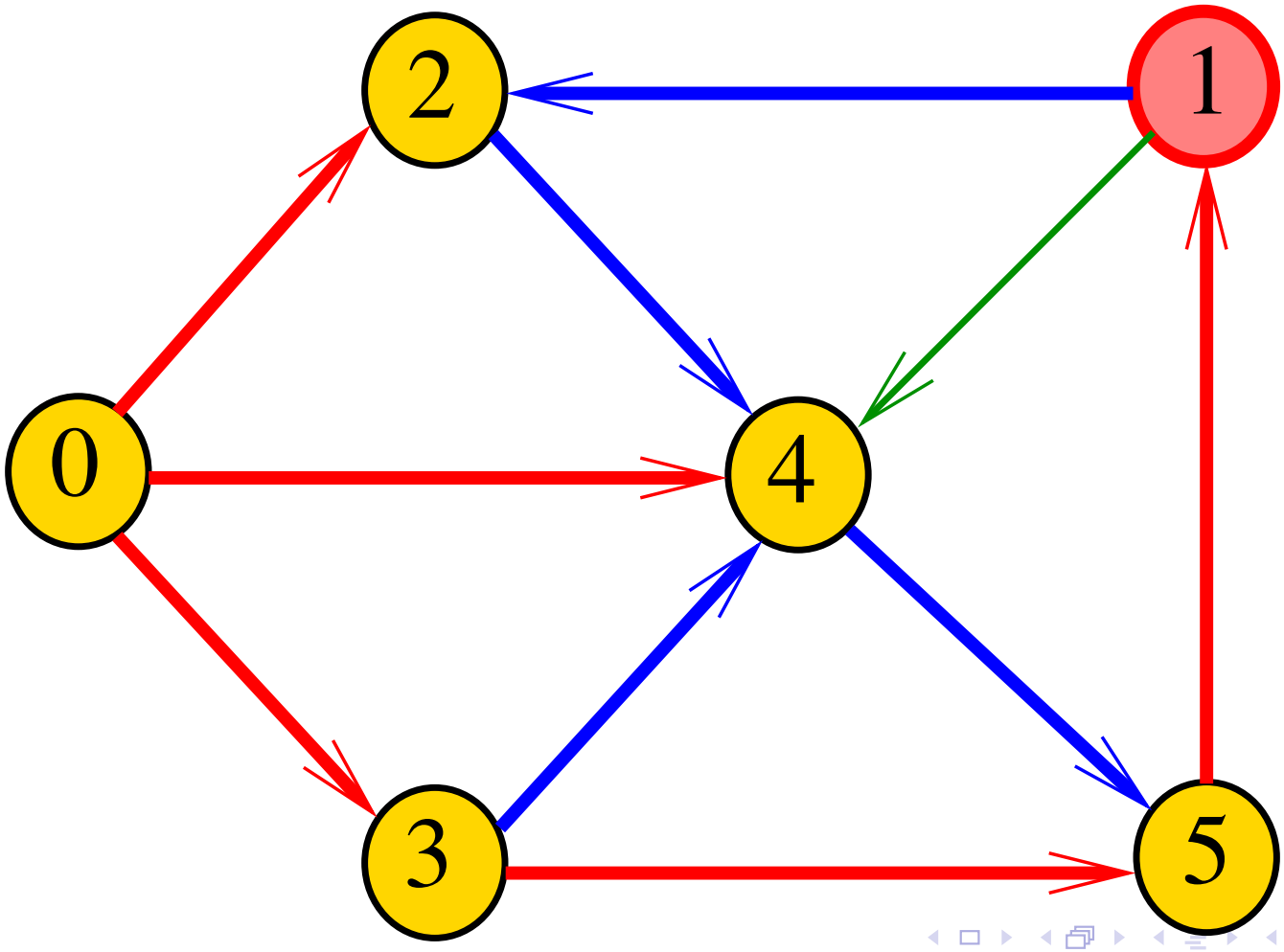
v	0	1	2	3	4	5
distTo[v]	0	3	1	1	1	2



Simulação

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

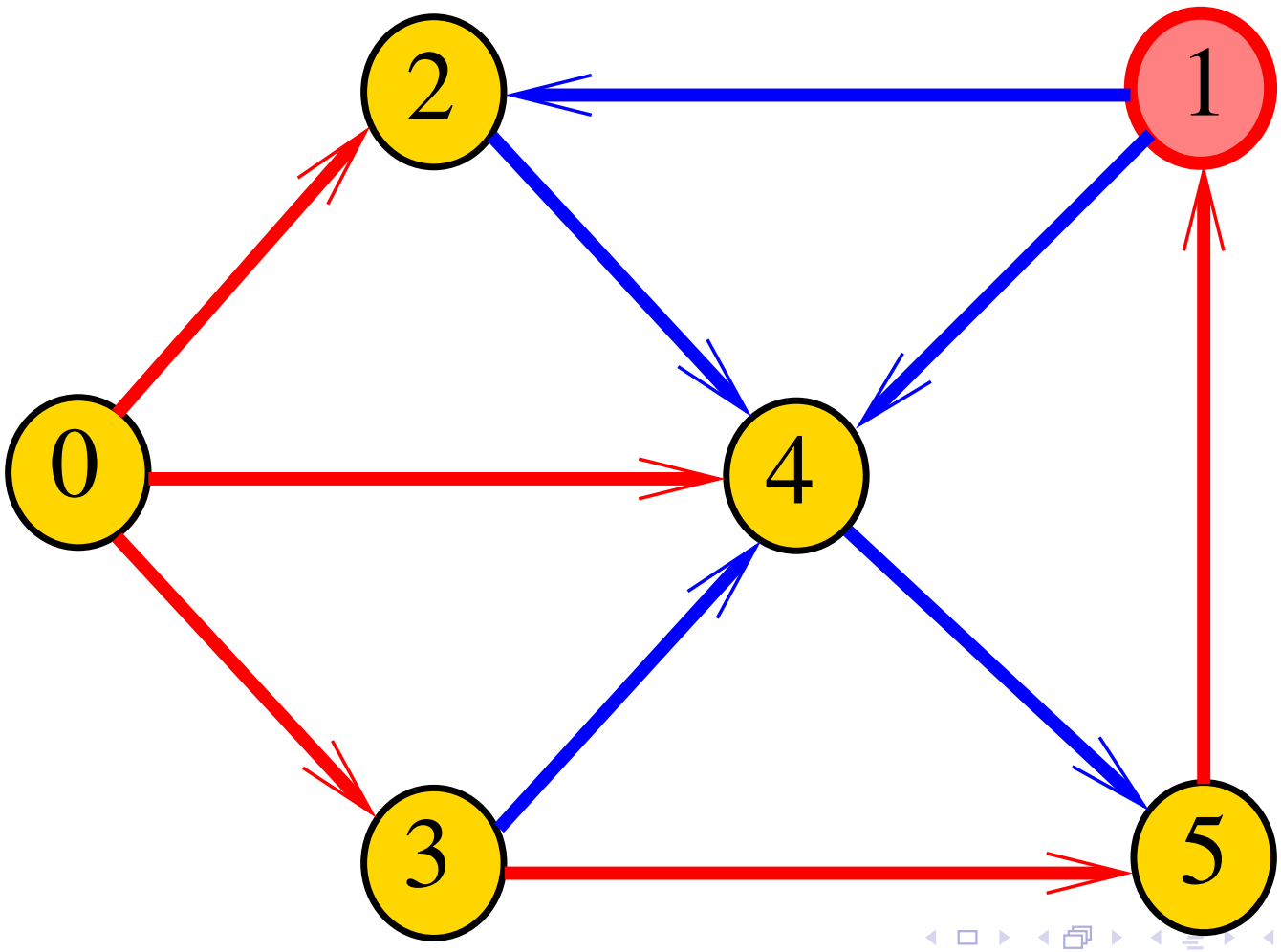
v	0	1	2	3	4	5
distTo[v]	0	3	1	1	1	2



Simulação

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

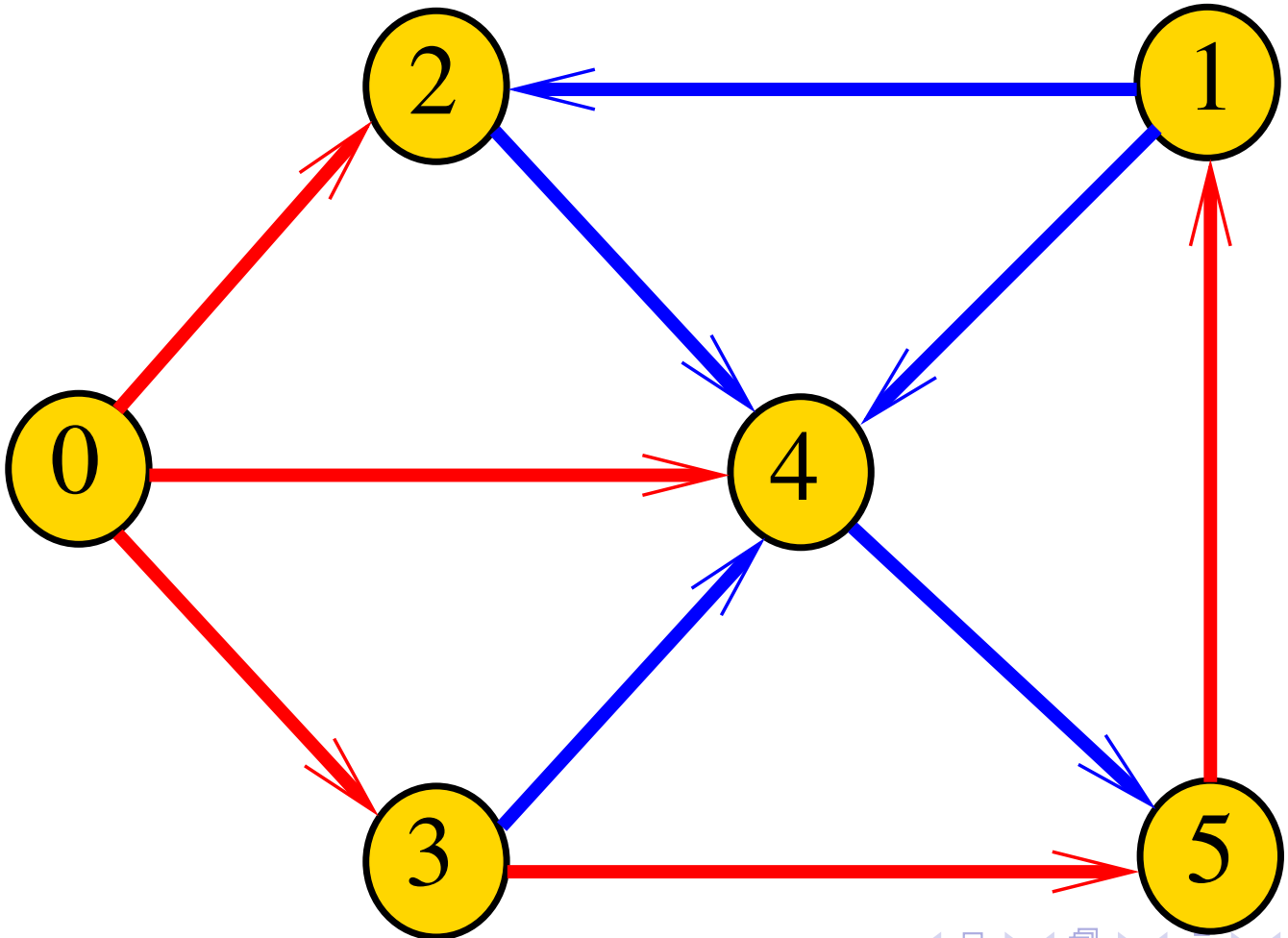
v	0	1	2	3	4	5
distTo[v]	0	3	1	1	1	2



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.

```
int INFINITY = G->V;
int *distTo = mallocSafe(G->V*sizeof(int));
```

Nova BFSpaths: estrutura

```
static struct bfspaths {
   int INFINITY;
   int s;
   bool *marked;
   int *distTo;          /* no lugar do marked */
   int *edgeTo;
};

typedef struct bsfpaths *bfsPaths;
```

DFSpaths: construtor

```
bfsPaths BFSpathsInit(Digraph G, int s) {
   bfsPaths T = mallocSafe(sizeof(*T));
   INFINITY = G->Vcor;
   T->s = s;
   T->marked = mallocSafe(G->V*sizeof(bool));
   T->distTo = mallocSafe(G->V*sizeof(int));
   T->edgeTo = mallocSafe(G->V*sizeof(int));
   for (v = 0; v < G->V; v++)
      T->distTo[v] = INFINITY;
   return T;
}
```

BFSpaths: esqueleto

```
bfsPaths BFSpaths(Digraph G, int s) {...}

static void bfs(Digraph G, int s,
                bfsPaths T) {...}

bool hasPath(bfsPaths T, int v) {...}

bool distTo(bfsPaths T, int v) {...}

/* fila com o caminho requisitado */
Queue pathTo(bfsPaths T, int v) {...}
```

BFSpaths

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

```
bfsPaths BFSpaths(Digraph G, int s) {
   bfsPaths T = BFSpathsInit(G, s);
   bfs(G, s, T);
   return T;
}
```

bfs(): inicializações

```
static void bfs(Digraph G, int s,
                bfsPaths T) {
   Queue q = queueInit(); int v; Link w;

   marked[v] = true;
   distTo[s] = 0;
   enqueue(q, s);

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

bfs(): iteração

```
   while (!isEmpty(q)) {
      v = dequeue(q);
      for (w = G->adj[v]; w != NULL; w = w->next)
         if (!T->marked[w->vertex]) {
         if (T->distTo[w->vertex] == T->INFINITY) {
            T->edgeTo[w->vertex] = v;
            T->distTo[w->vertex] = distTo[v] + 1;
            T->marked[w->vertex] = true;
            enqueue(q, w->vertex);
         }
      }
   }
}
```

BFSpaths

Há um caminho de s a v?

```
bool hasPath(bfsPaths T, int v) {
   return T->distTo[v] < T->INFINITY;
}

/* retorna o número de arcos em */
/* um caminho mínimo de s a t   */
int distTo(bfsPaths T, int v) {
   return T->distTo[v];
}
```

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`.

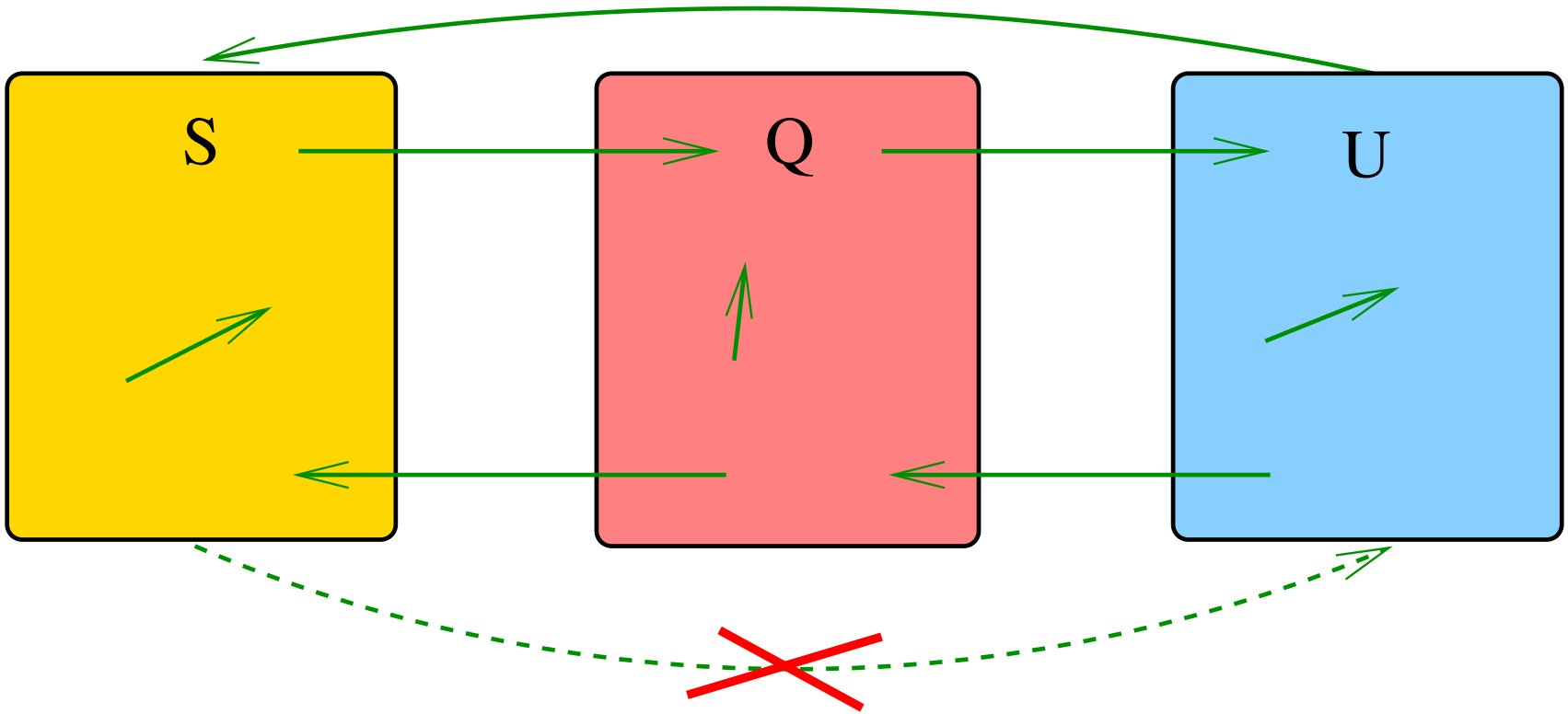
Relações invariantes

S = vértices examinados

Q = vértices visitados = vértices na fila

U = vértices ainda não visitados

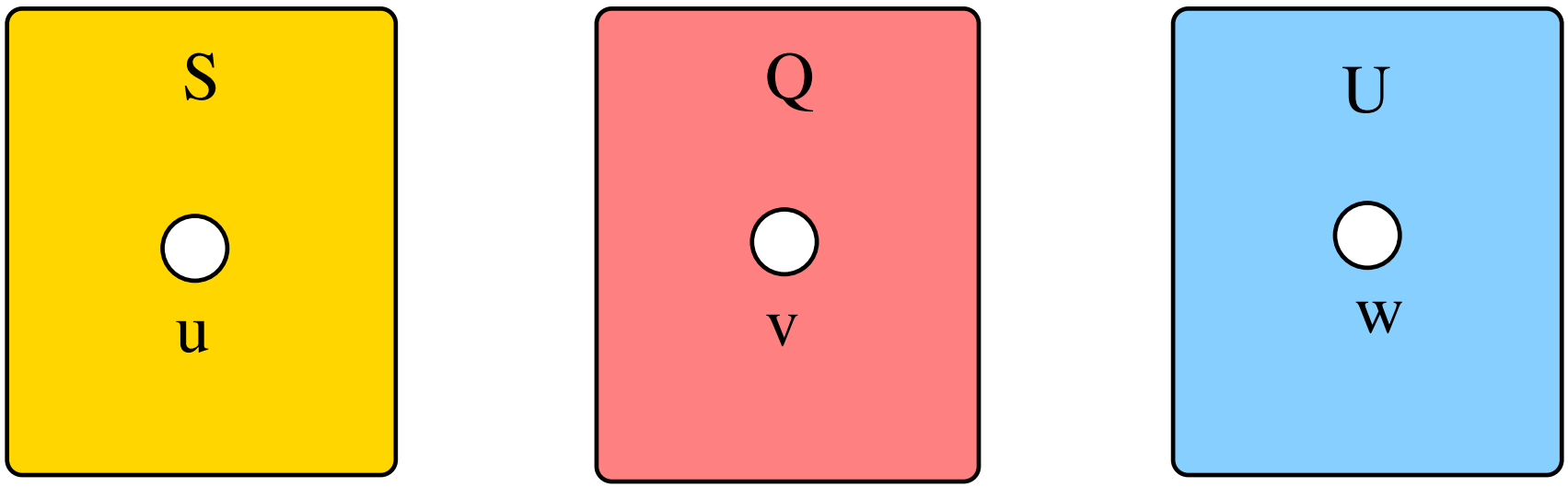
(i0) não existe arco v-w com v em S e w em U.



Relações invariantes

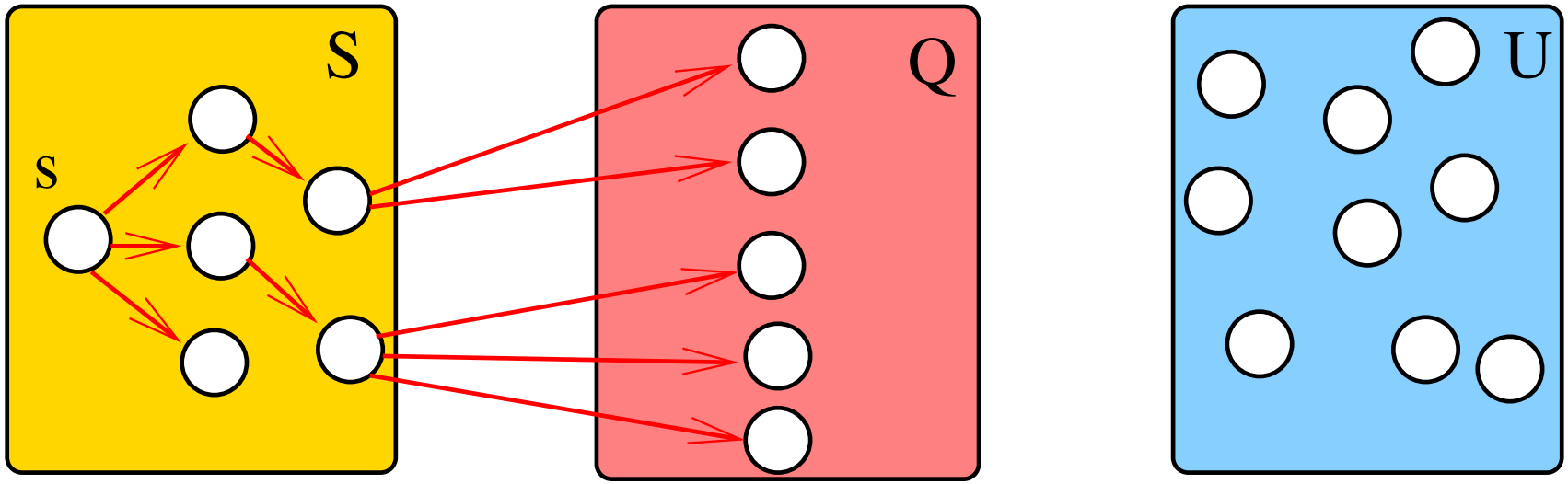
(i1) para cada u em S, v em Q e w em U

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



Relações invariantes

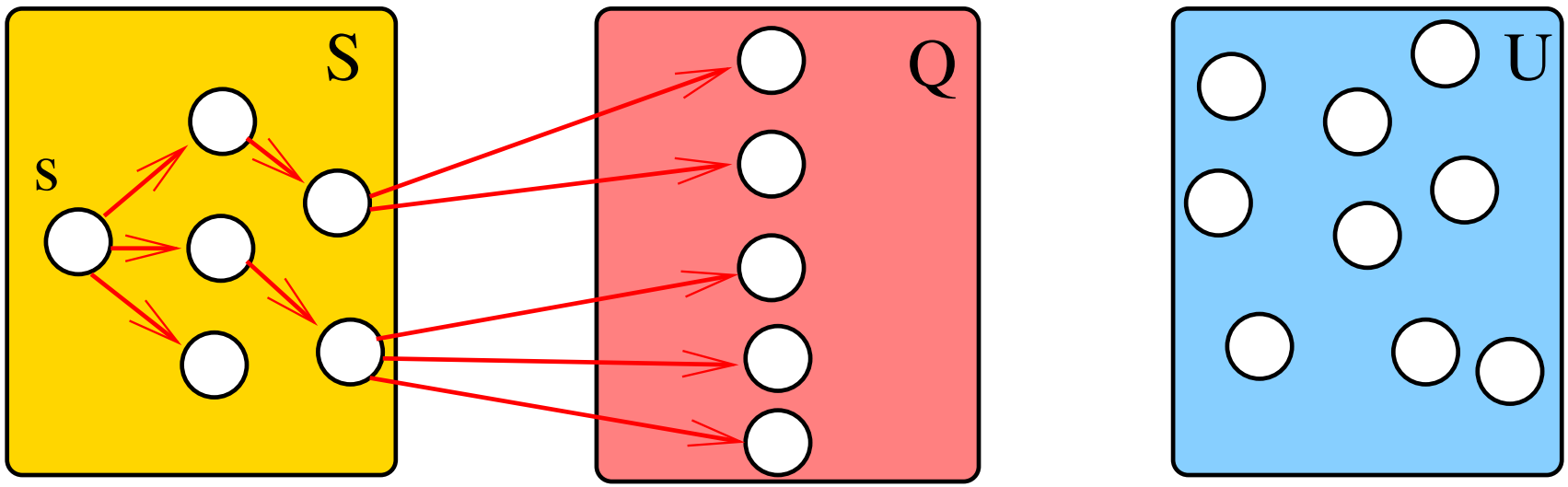
(i2) O vetor `edgeTo` restrito aos vértices de `S` e `Q` determina uma arborescência com raiz `s`.



Relações invariantes

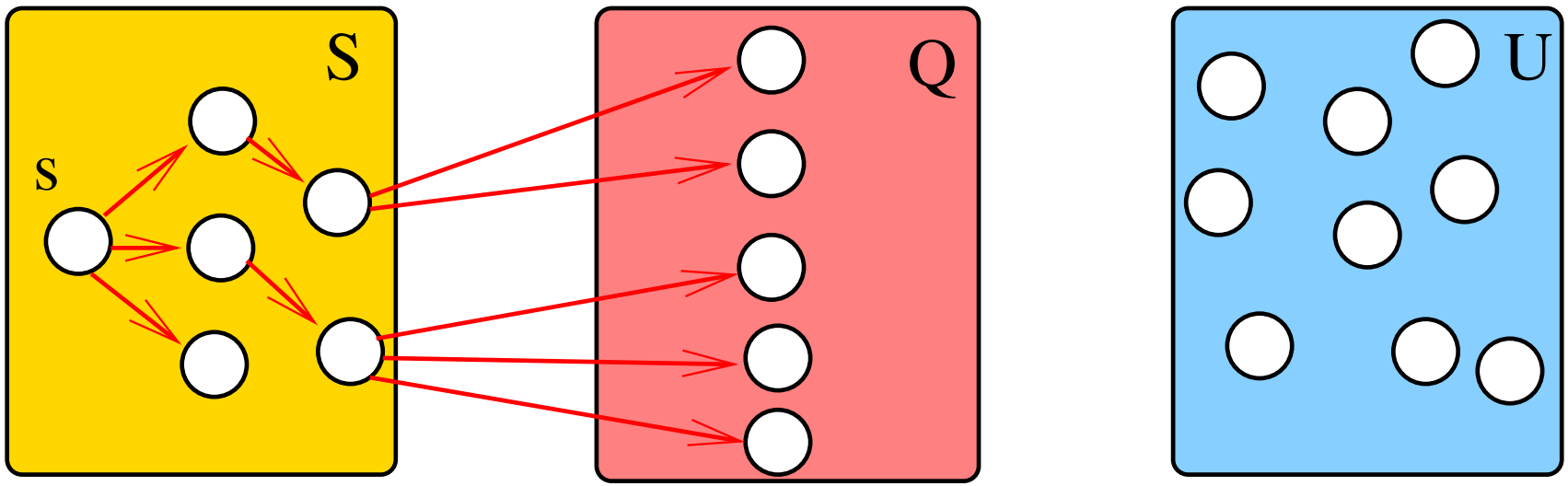
(i3) Para arco `v-w` na arborescência, vale que

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

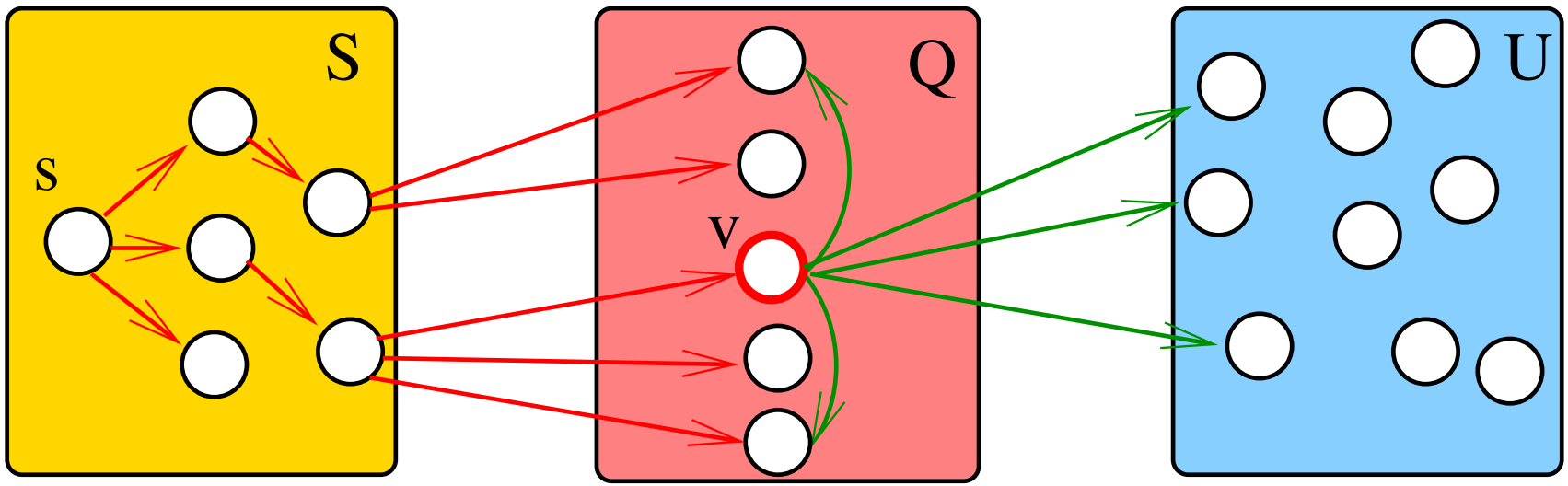


Relações invariantes

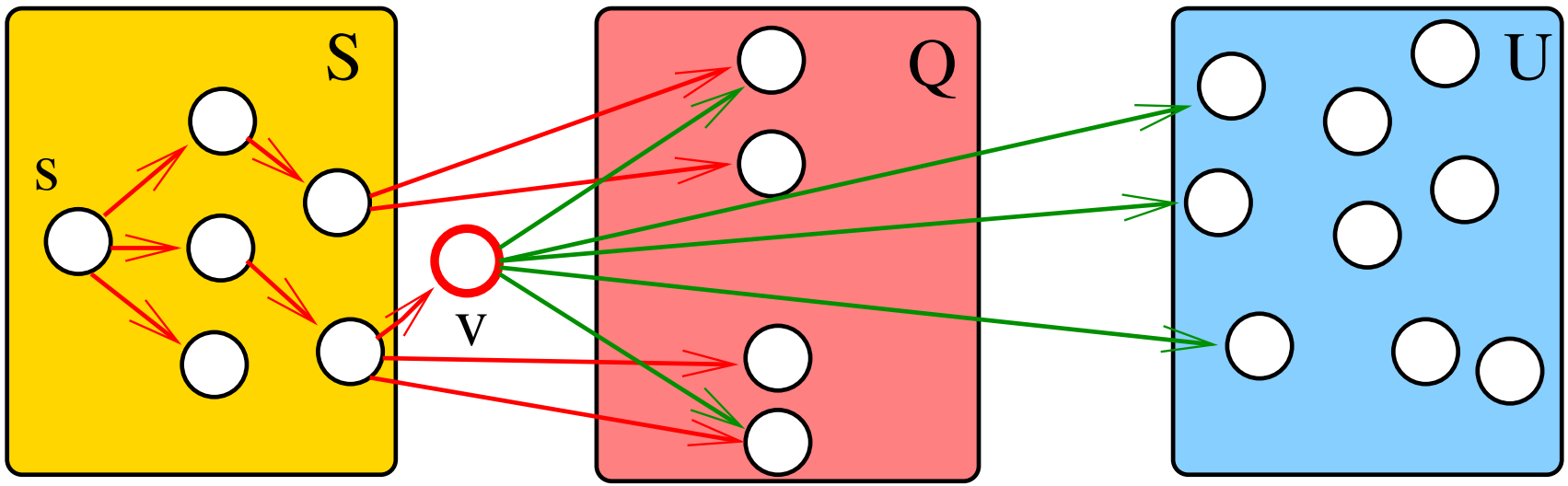
(i3) Para cada vértice `v` em `S`, vale que `distTo[v]` é o custo de um caminho mínimo de `s` a `v`.



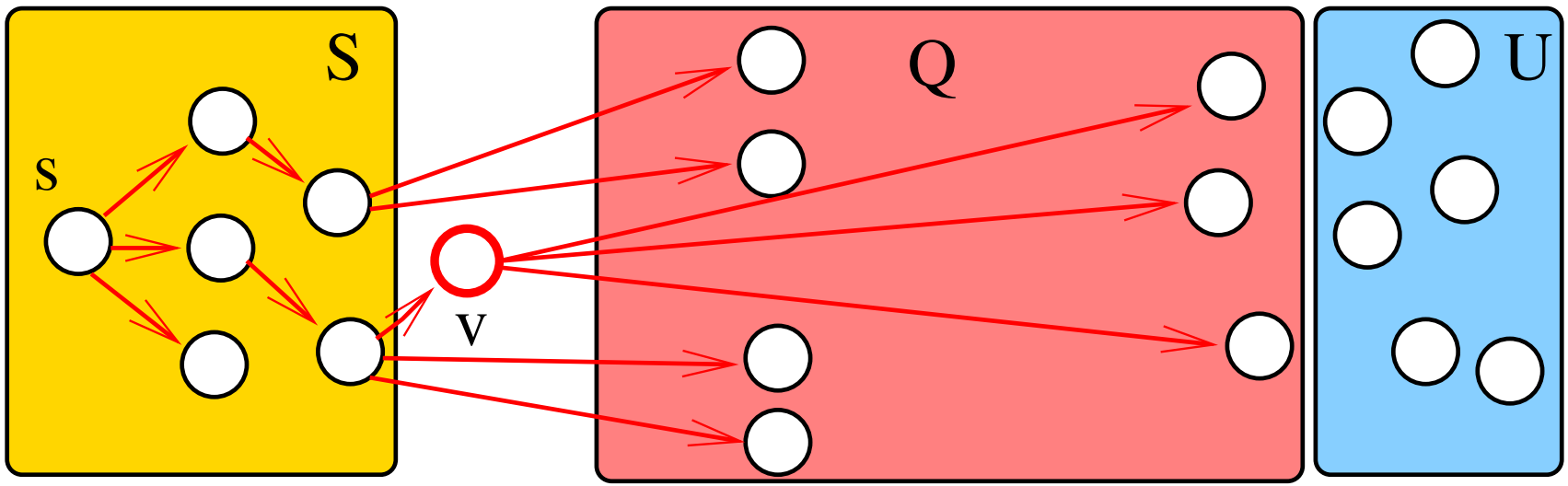
Iteração



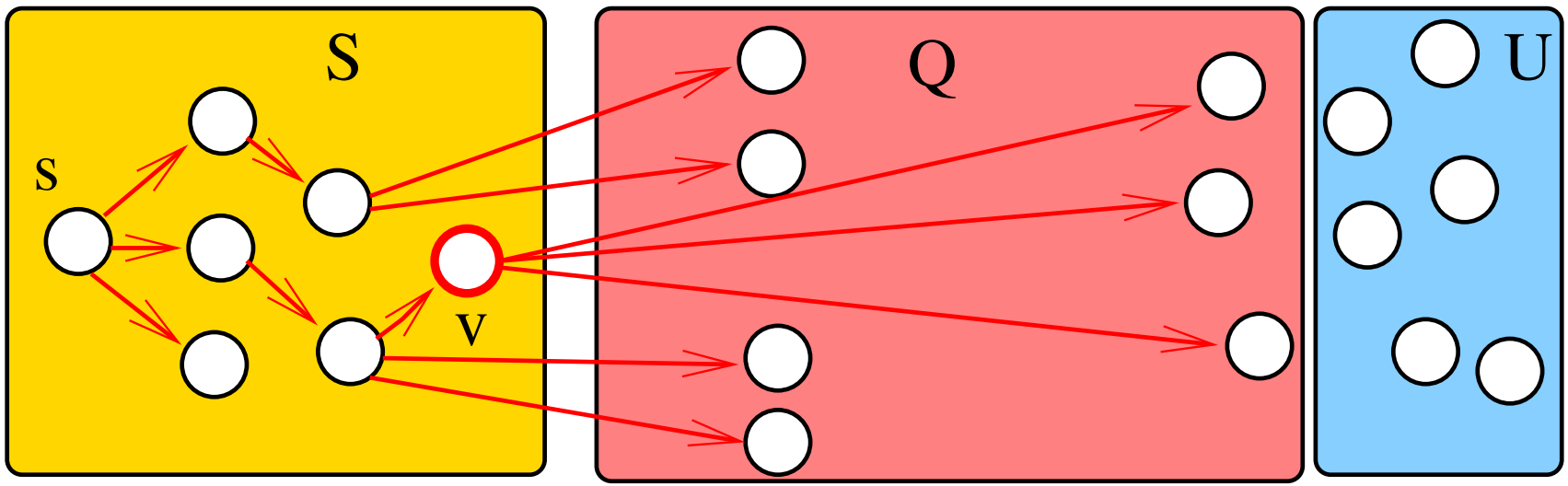
Iteração



Iteração



Iteração



Consumo de tempo

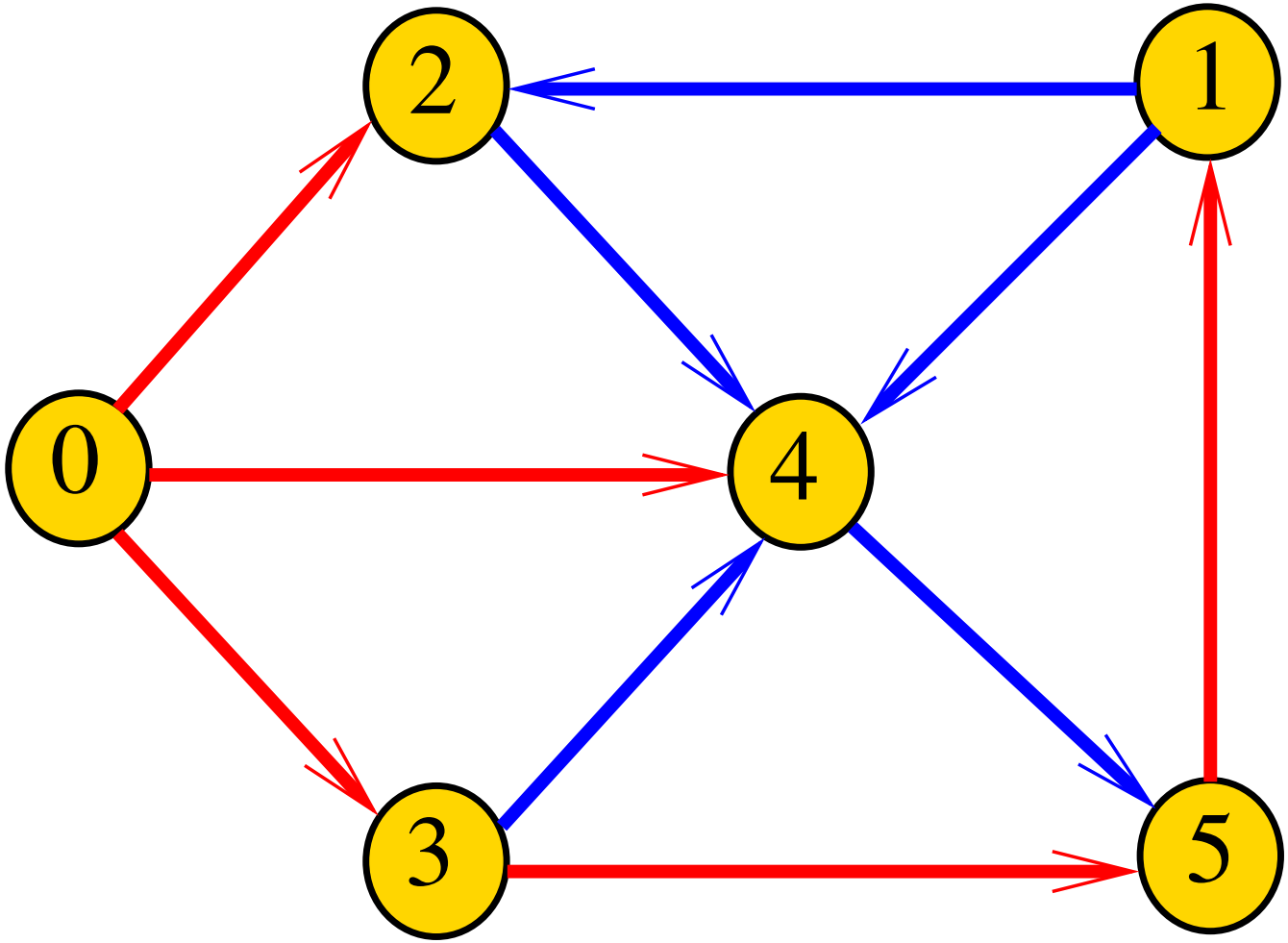
O consumo de tempo de `BFSpaths` para vetor de listas de adjacência é $O(V + E)$.

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