MAC0323 Algoritmos e Estruturas de Dados II

Edição 2020 – 2

AULA 19

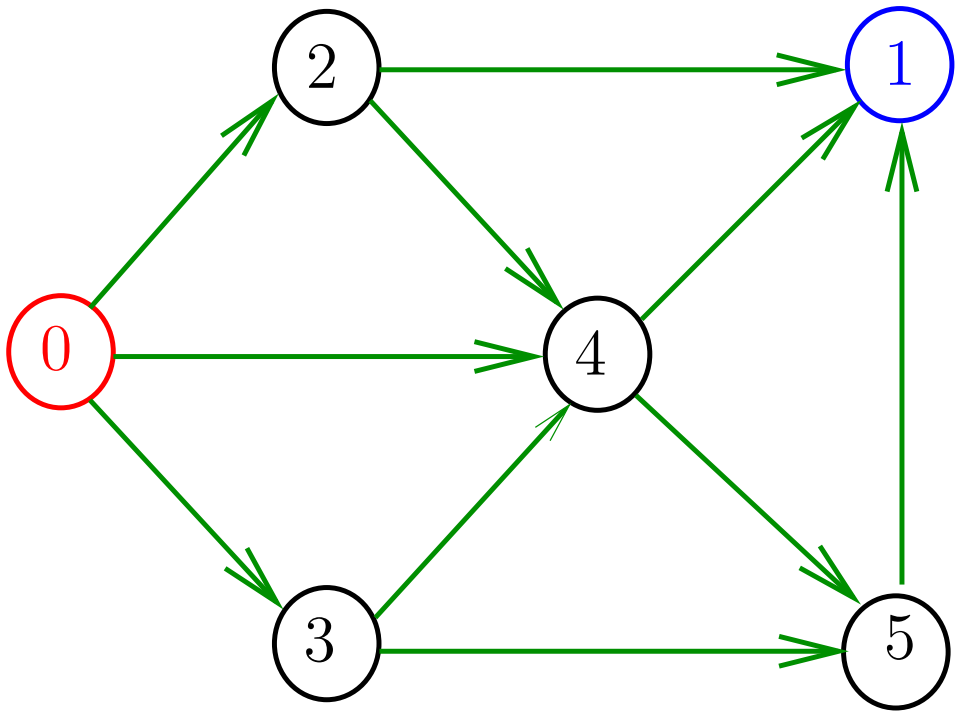
Procurando caminhos

Fonte: Mincecraft maze created by Carl Eklof (algs4)

Procurando um caminho

Problema: dados um digrafo `G` e dois vértices `s` e `t`, decidir se existe um caminho de `s` a `t`.

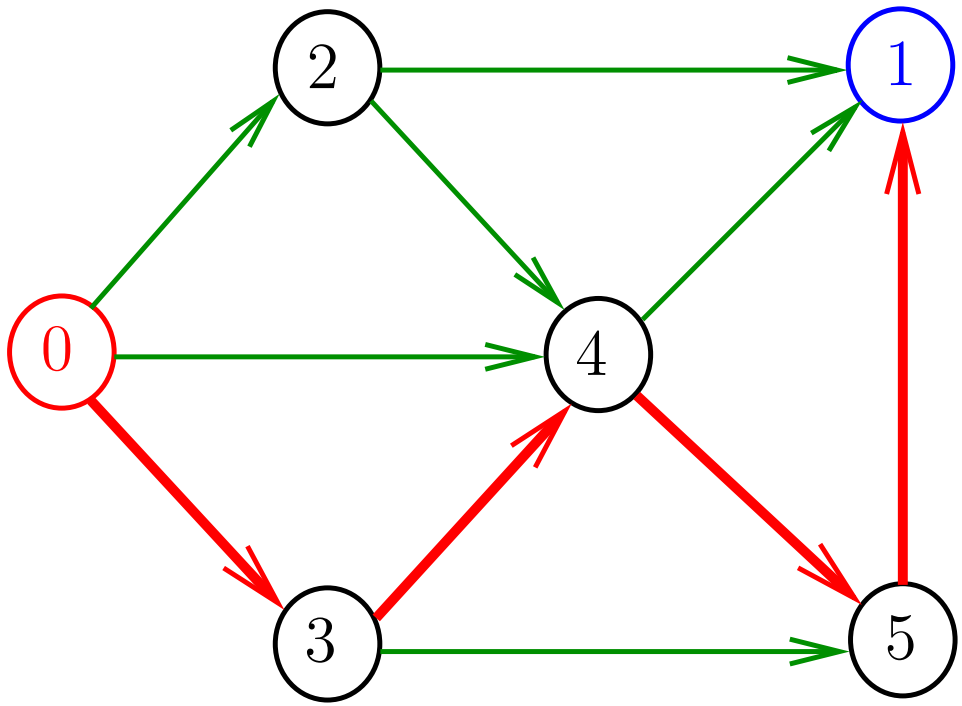
Exemplo: para `s` $= 0$ e `t` $= 1$, a resposta é SIM



Procurando um caminho

Problema: dados um digrafo `G` e dois vértices `s` e `t`, decidir se existe um caminho de `s` a `t`.

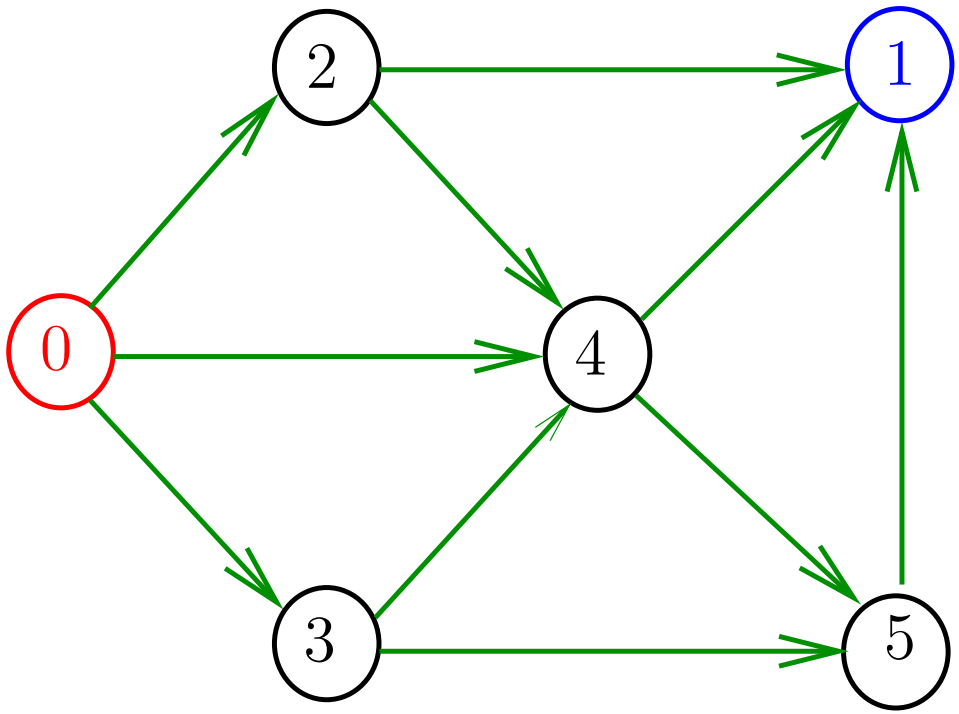
Exemplo: para `s` = 0 e `t` = 1, a resposta é SIM



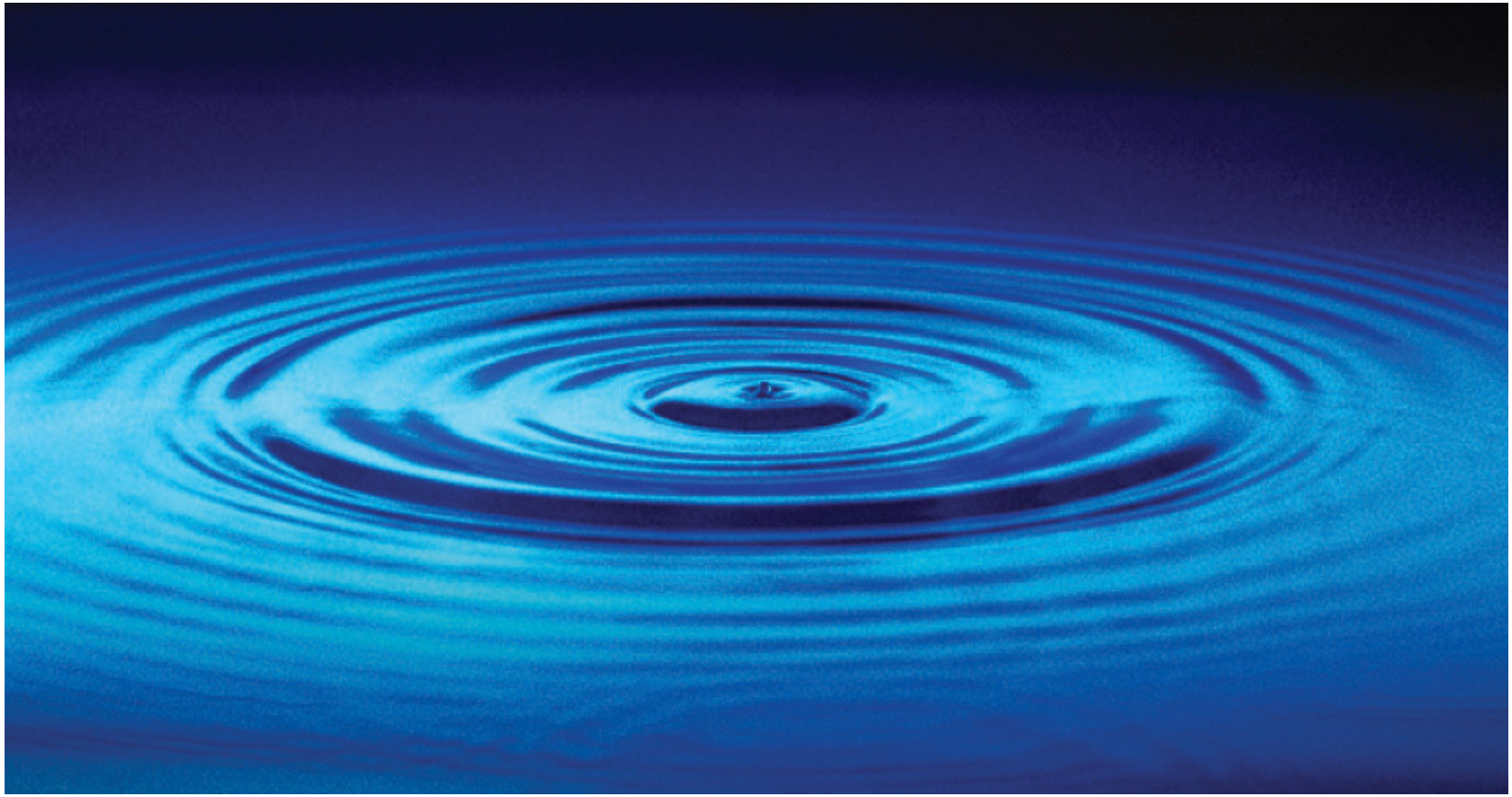
Procurando um caminho

Problema: dados um digrafo `G` e dois vértices `s` e `t`, decidir se existe um caminho de `s` a `t`.

Exemplo: para `s` = 5 e `t` = 4, a resposta é NÃO



Busca em largura



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

Busca ou varredura

Um algoritmo 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

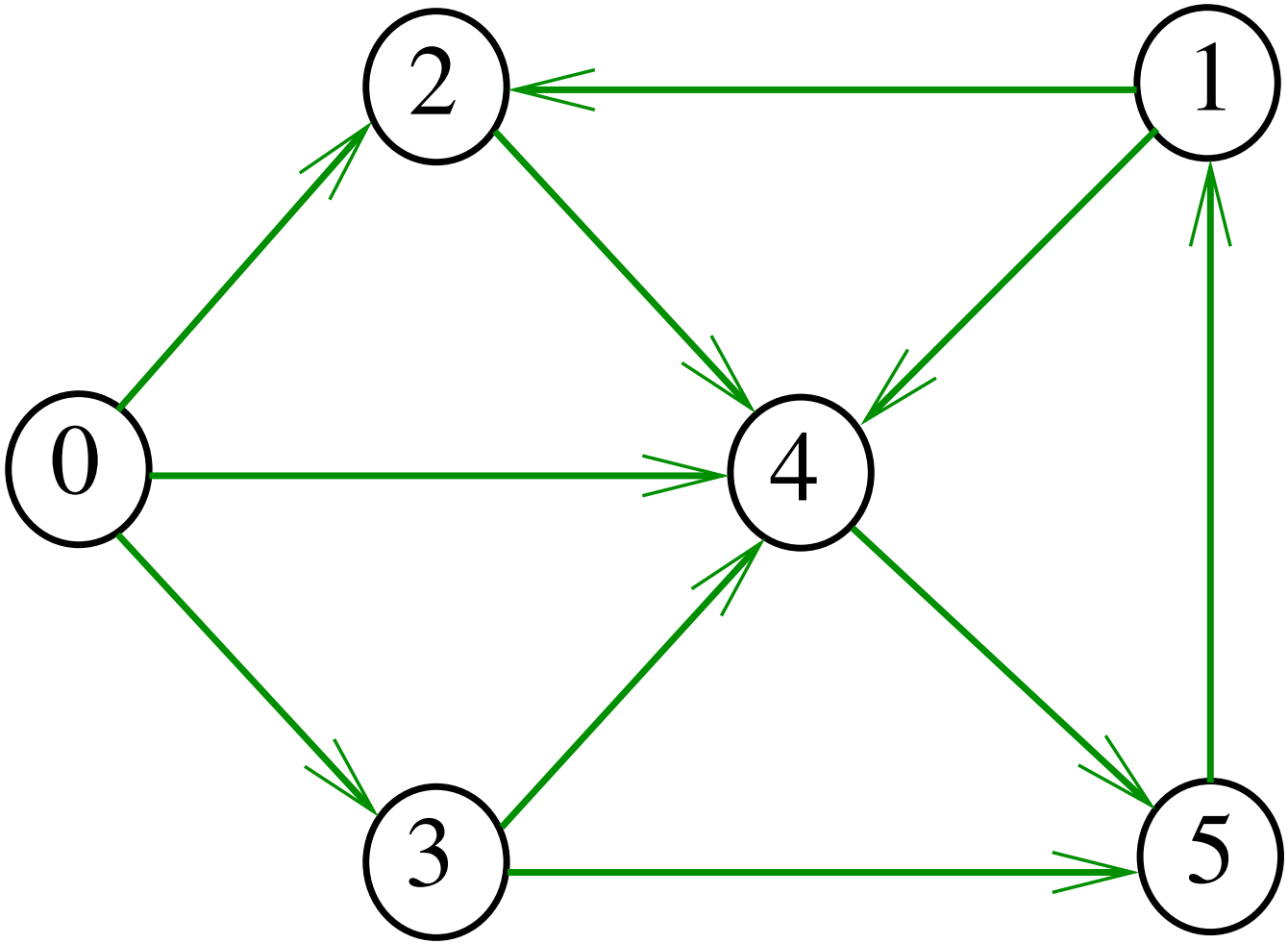
A **busca em largura** (=*breadth-first search* = *BFS*) começa por um vértice, digamos s, especificado pelo usuário.

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.

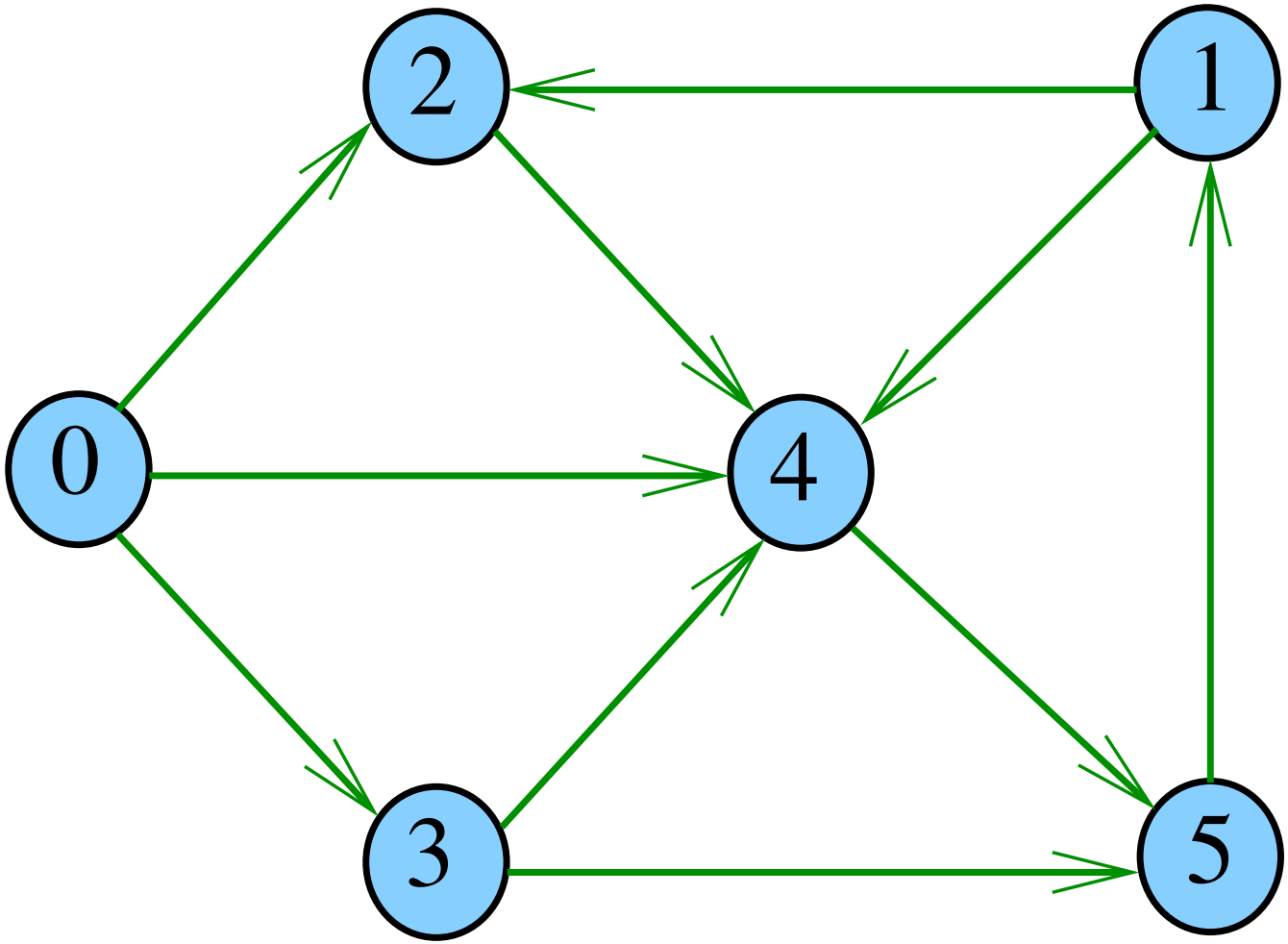
Simulação

i	0	1	2	3	4	5
q[i]						



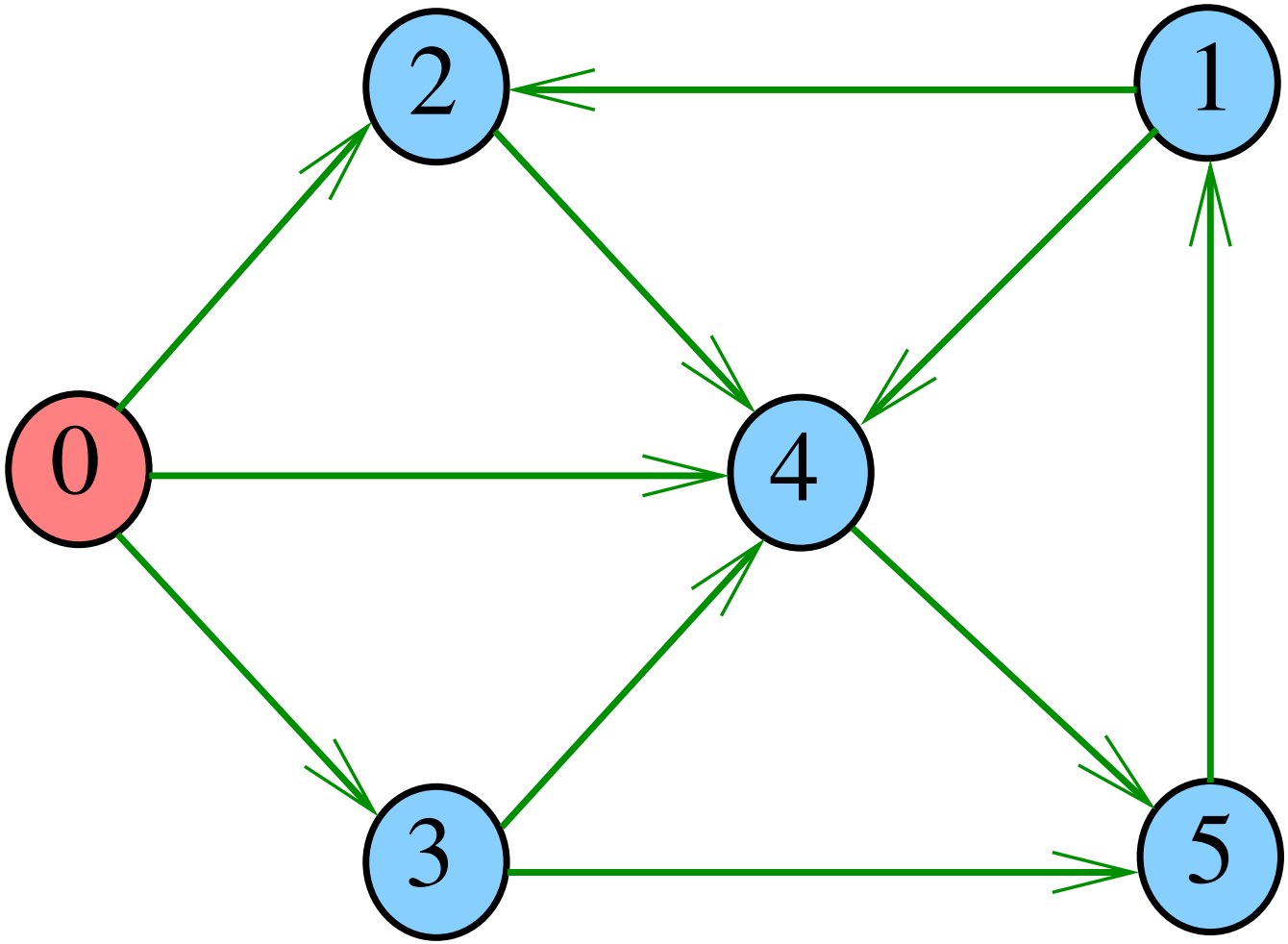
Simulação

i	0	1	2	3	4	5
q[i]						



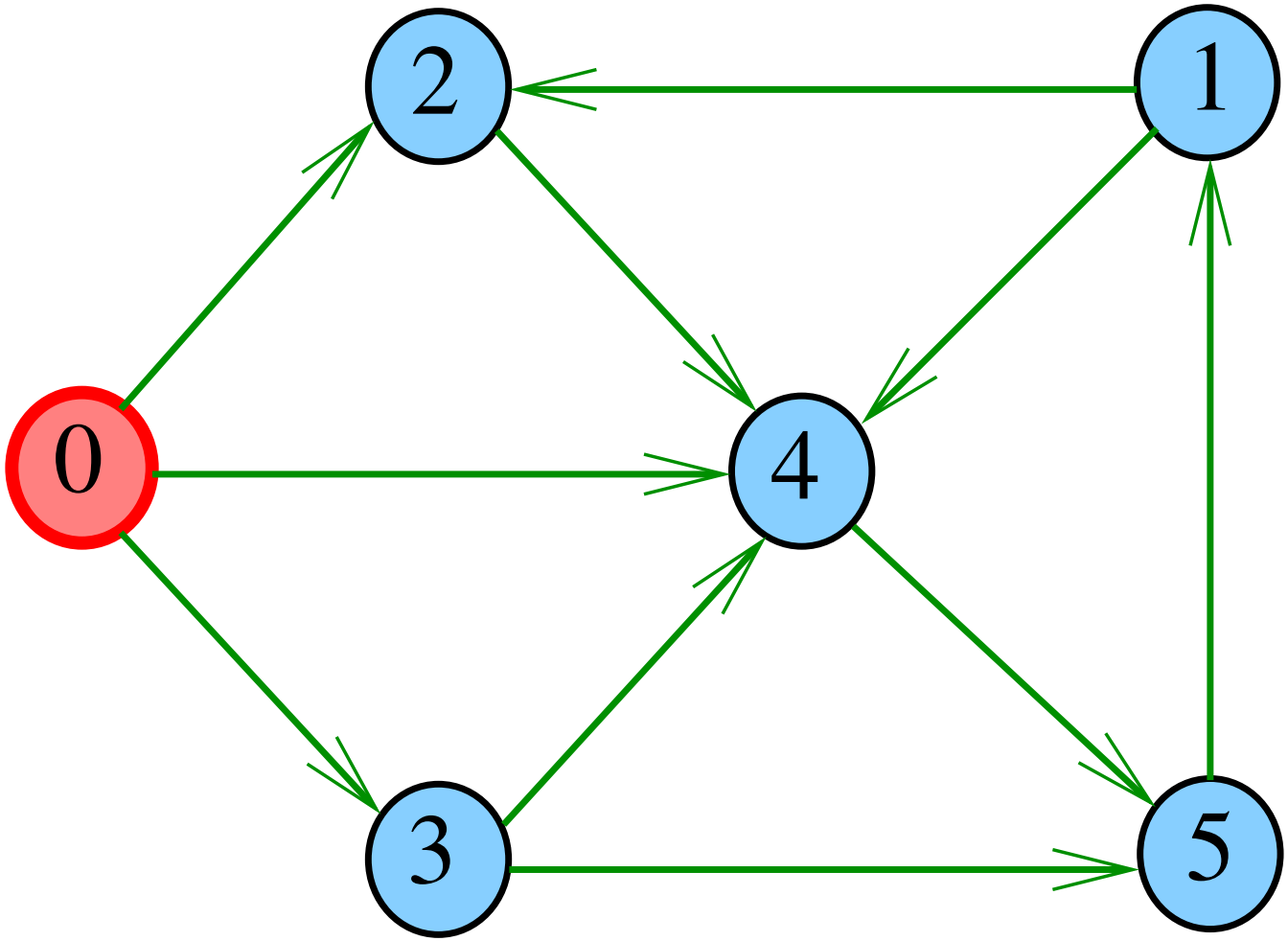
Simulação

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



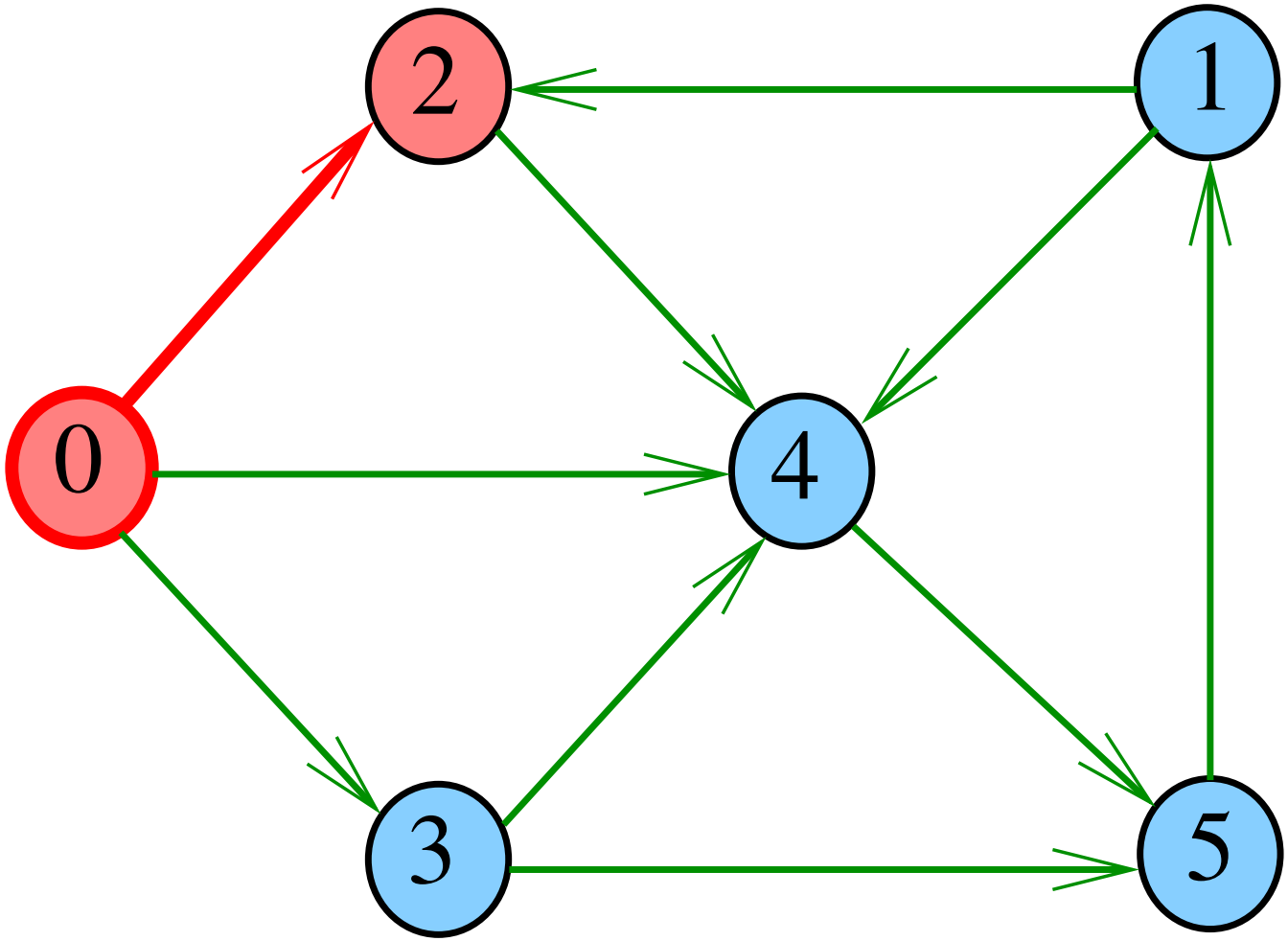
Simulação

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



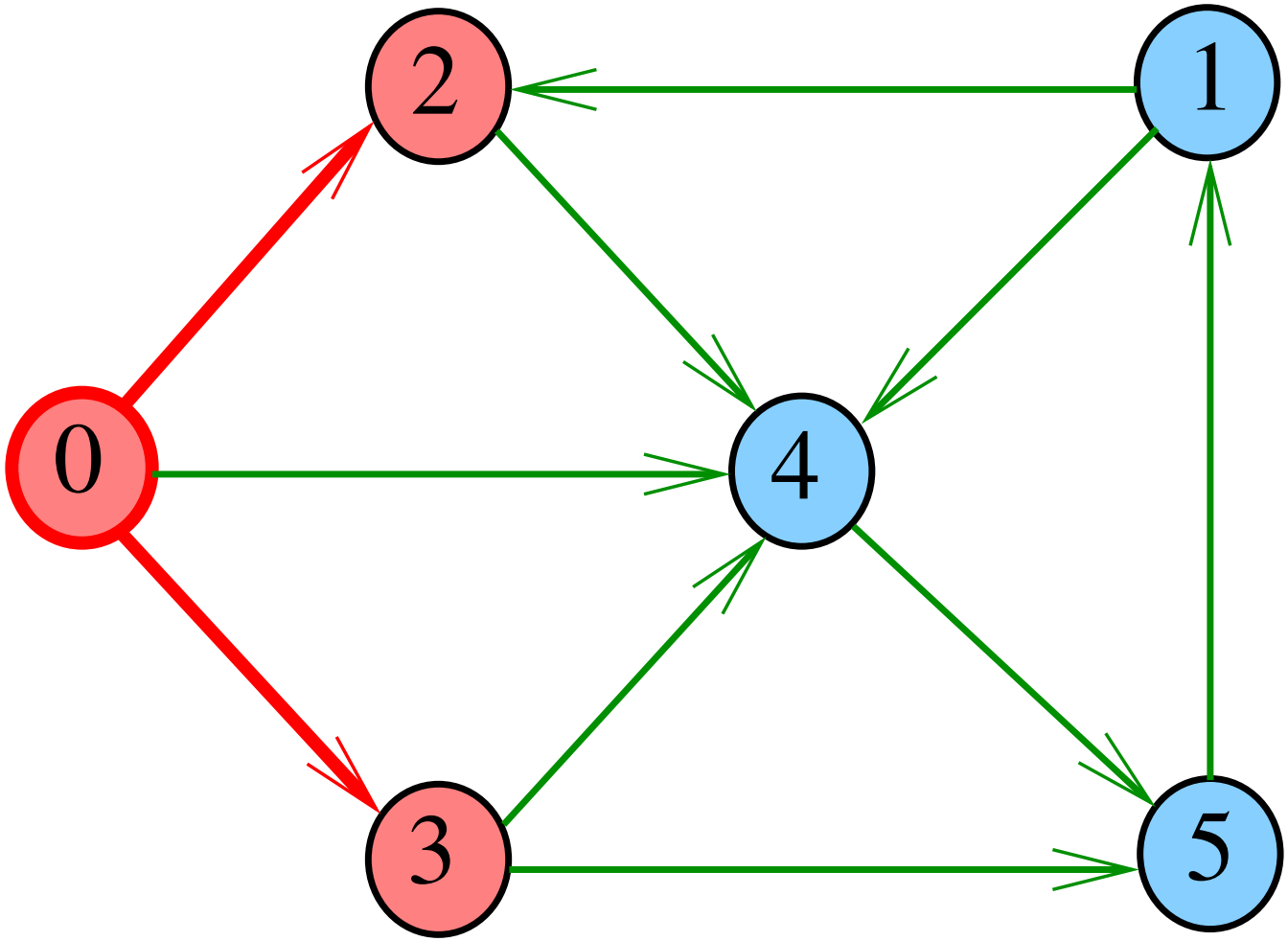
Simulação

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



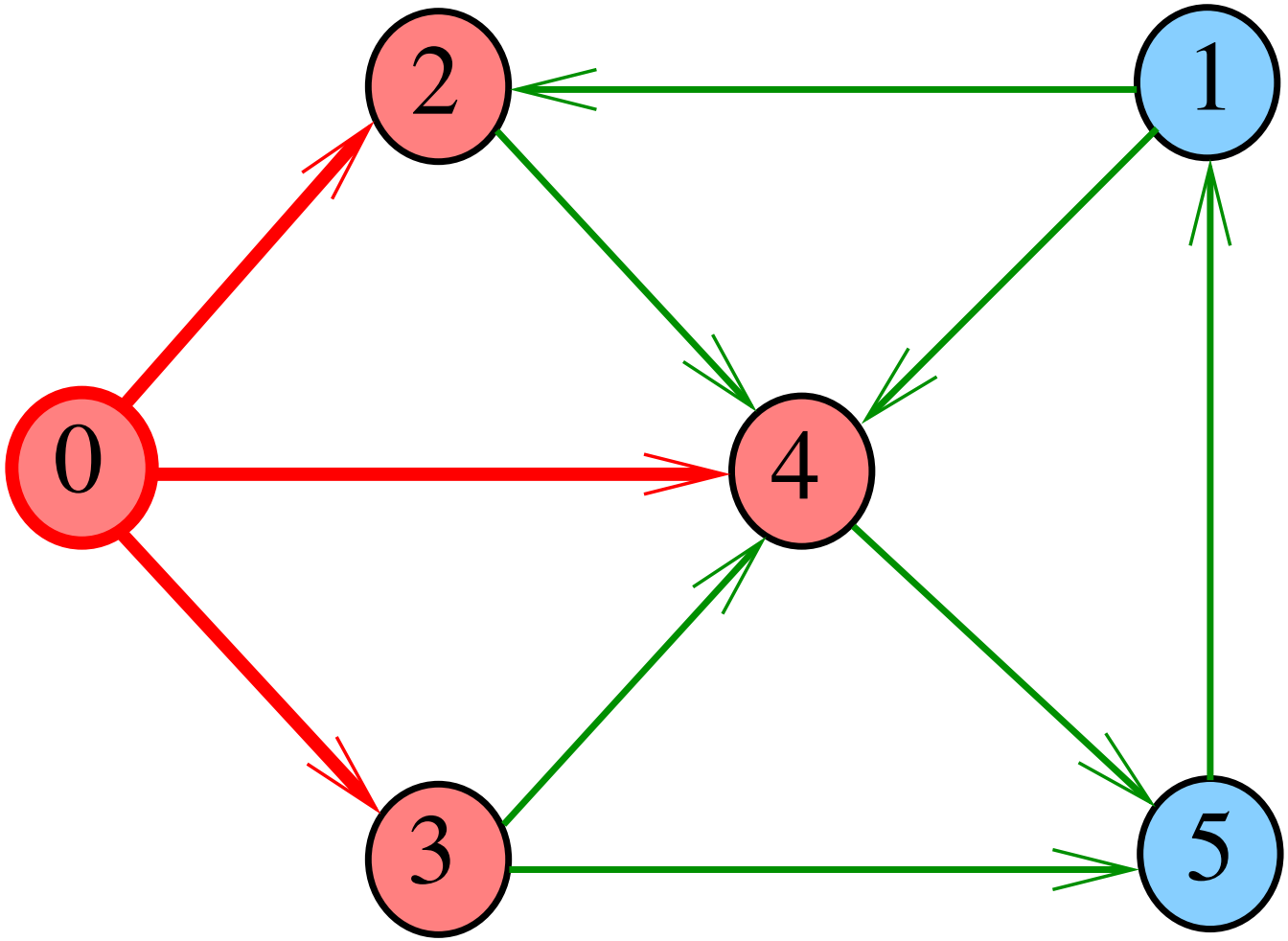
Simulação

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



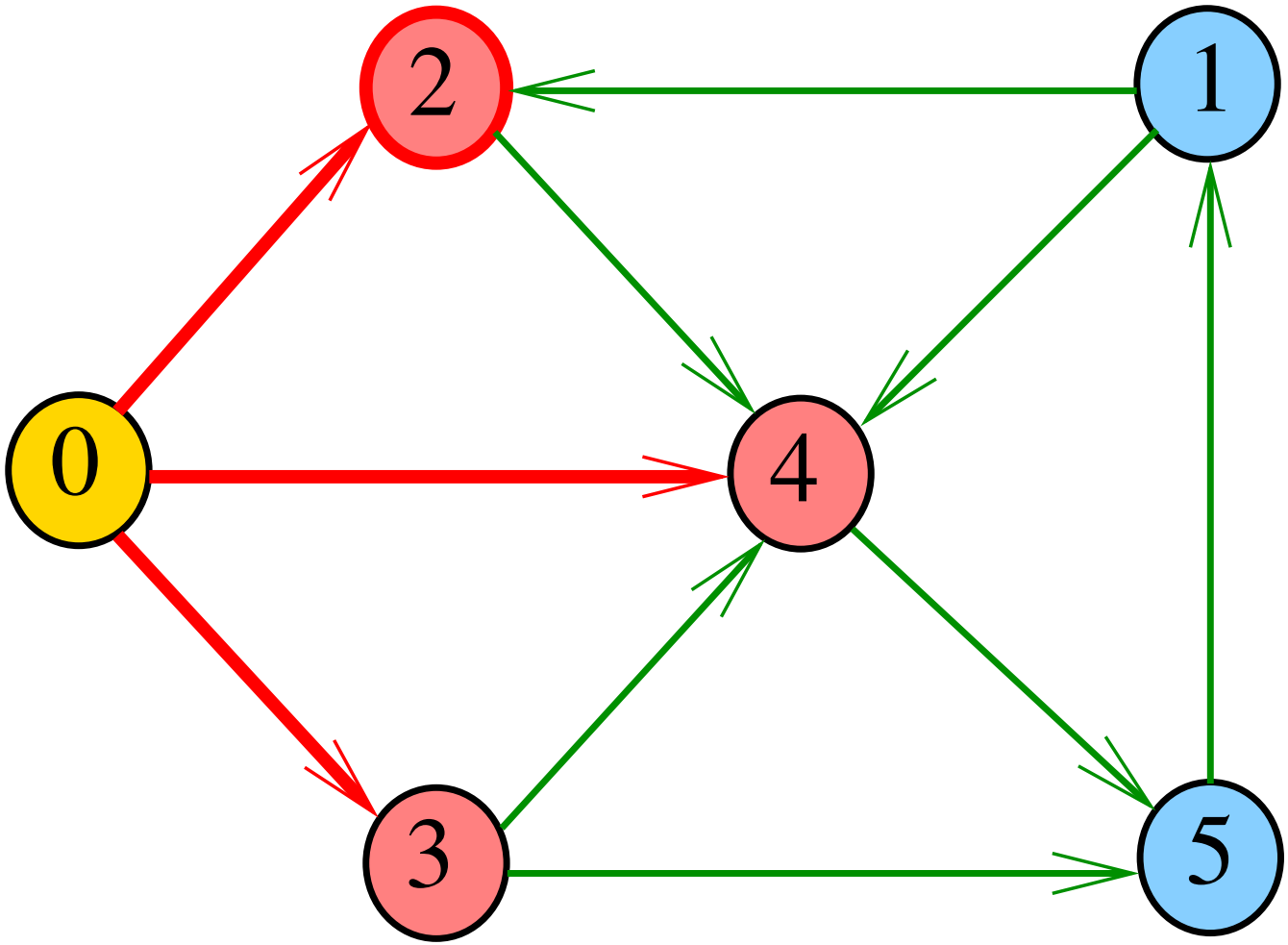
Simulação

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



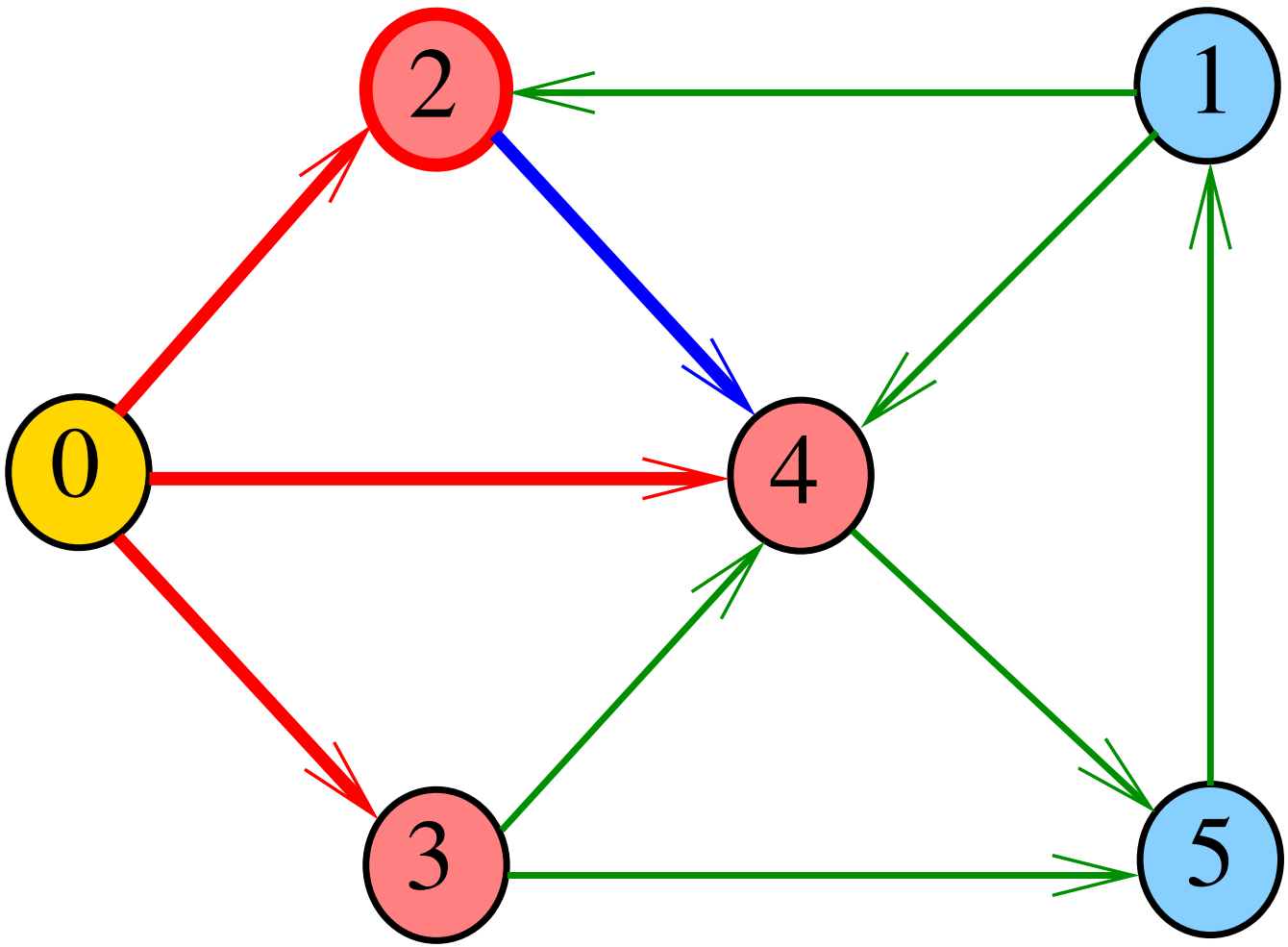
Simulação

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



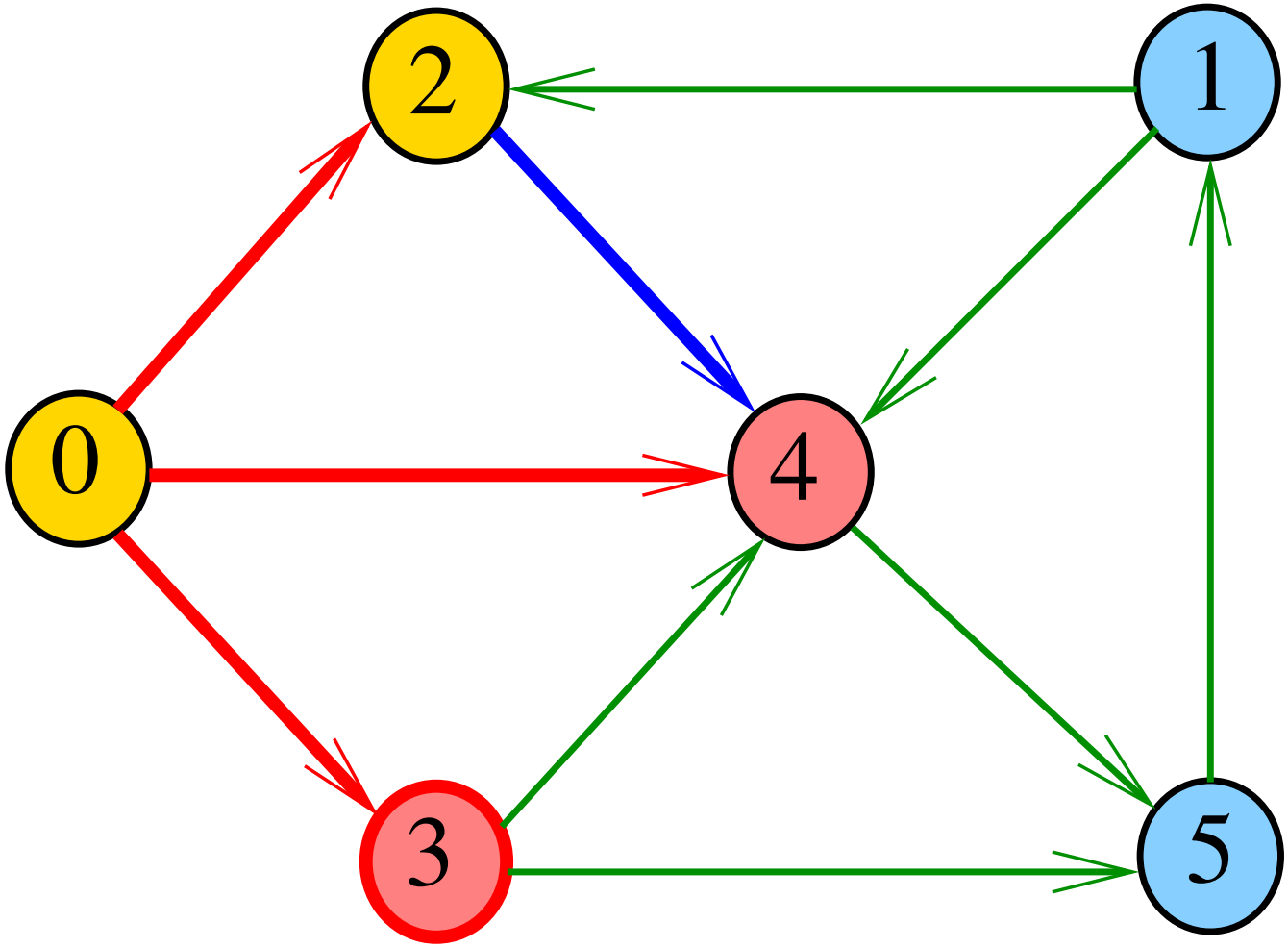
Simulação

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



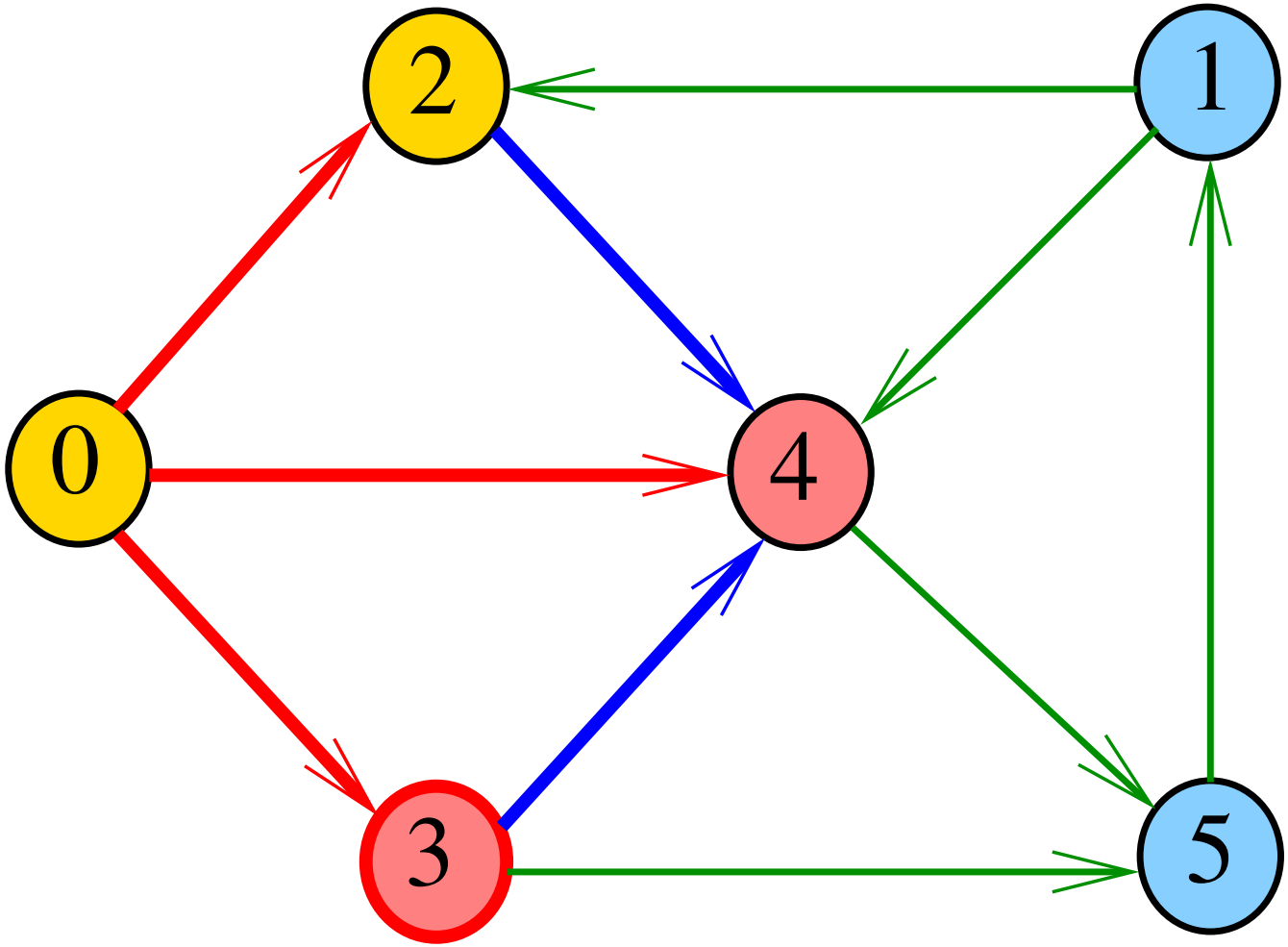
Simulação

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



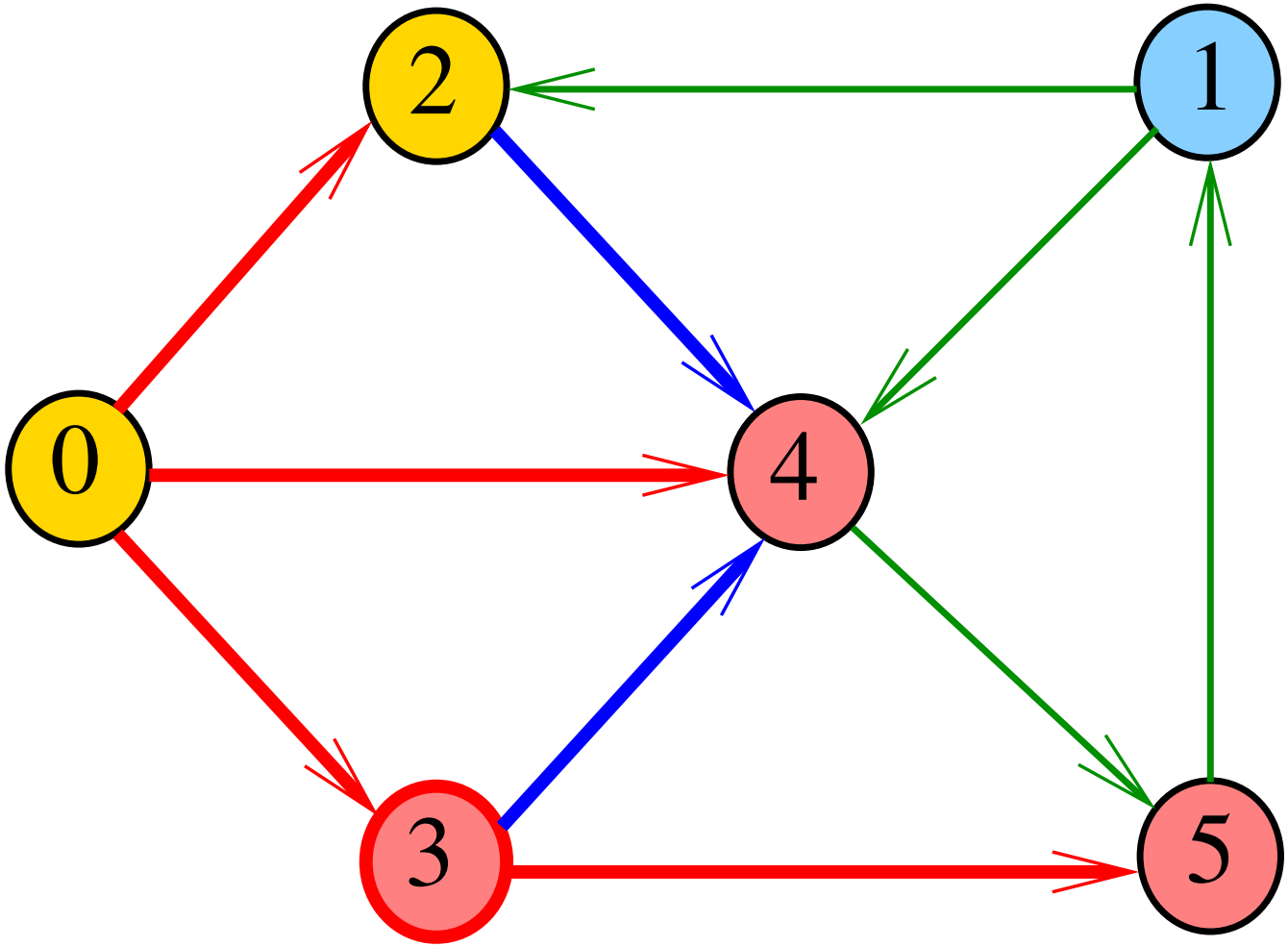
Simulação

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



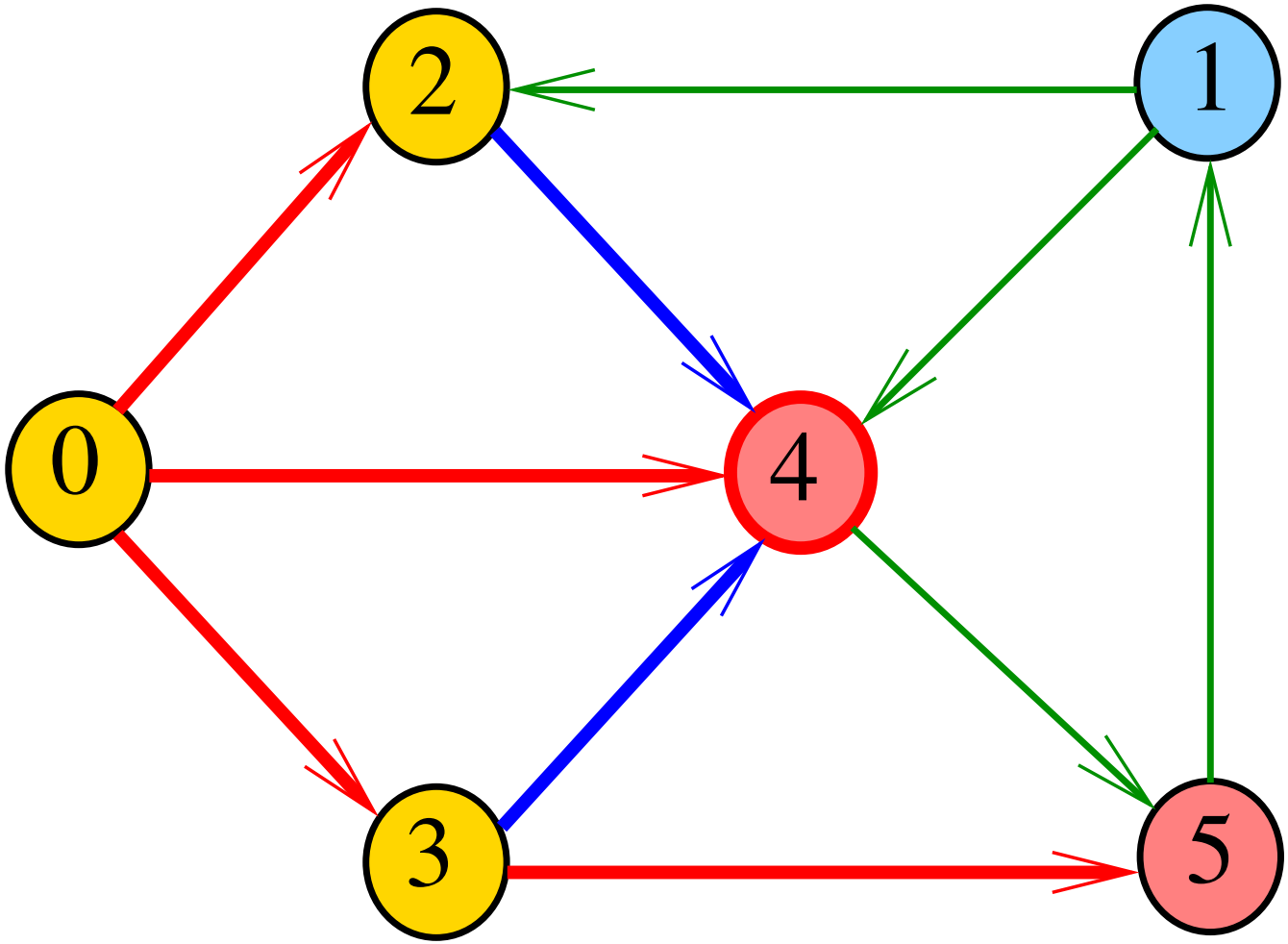
Simulação

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



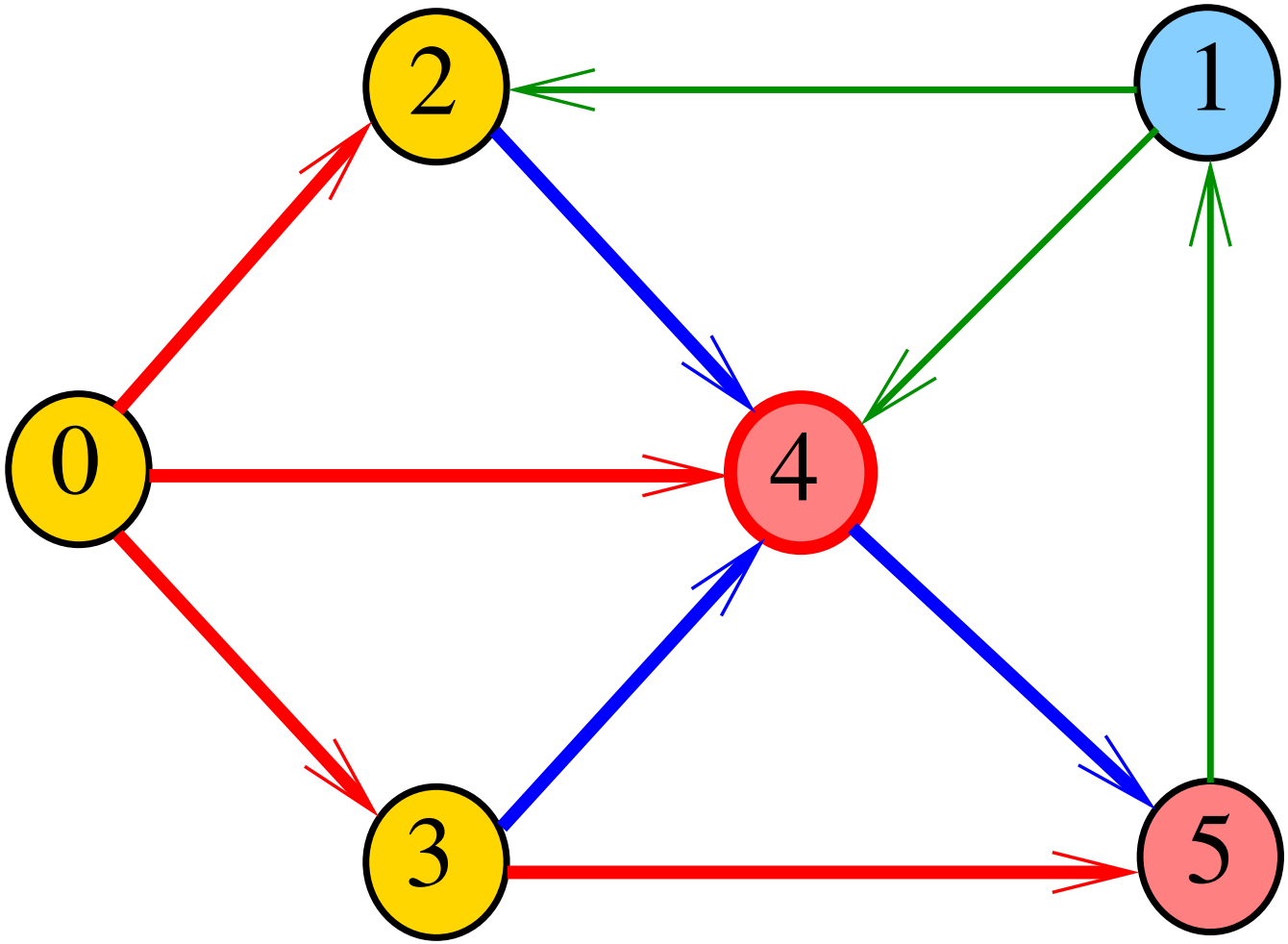
Simulação

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



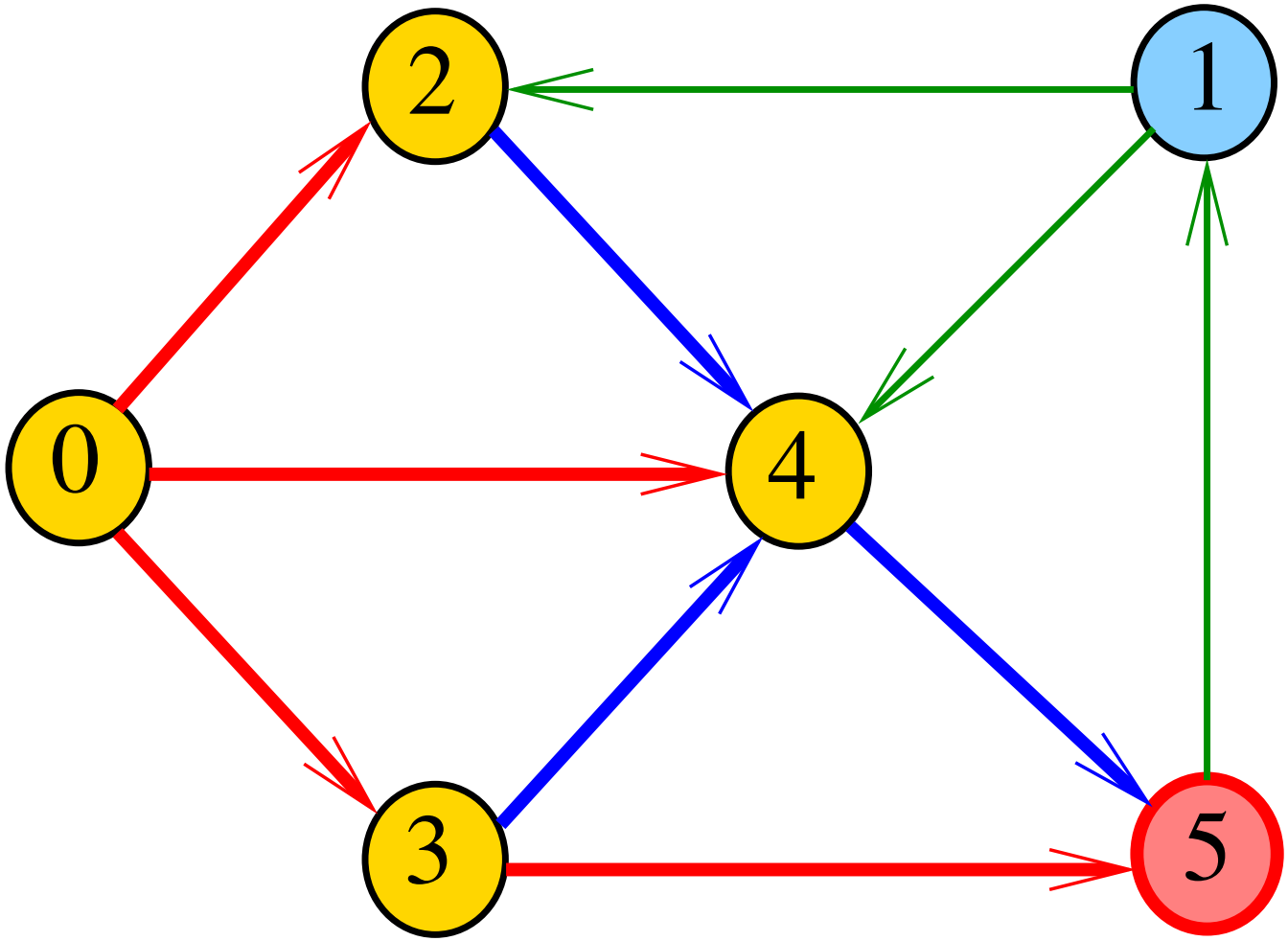
Simulação

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



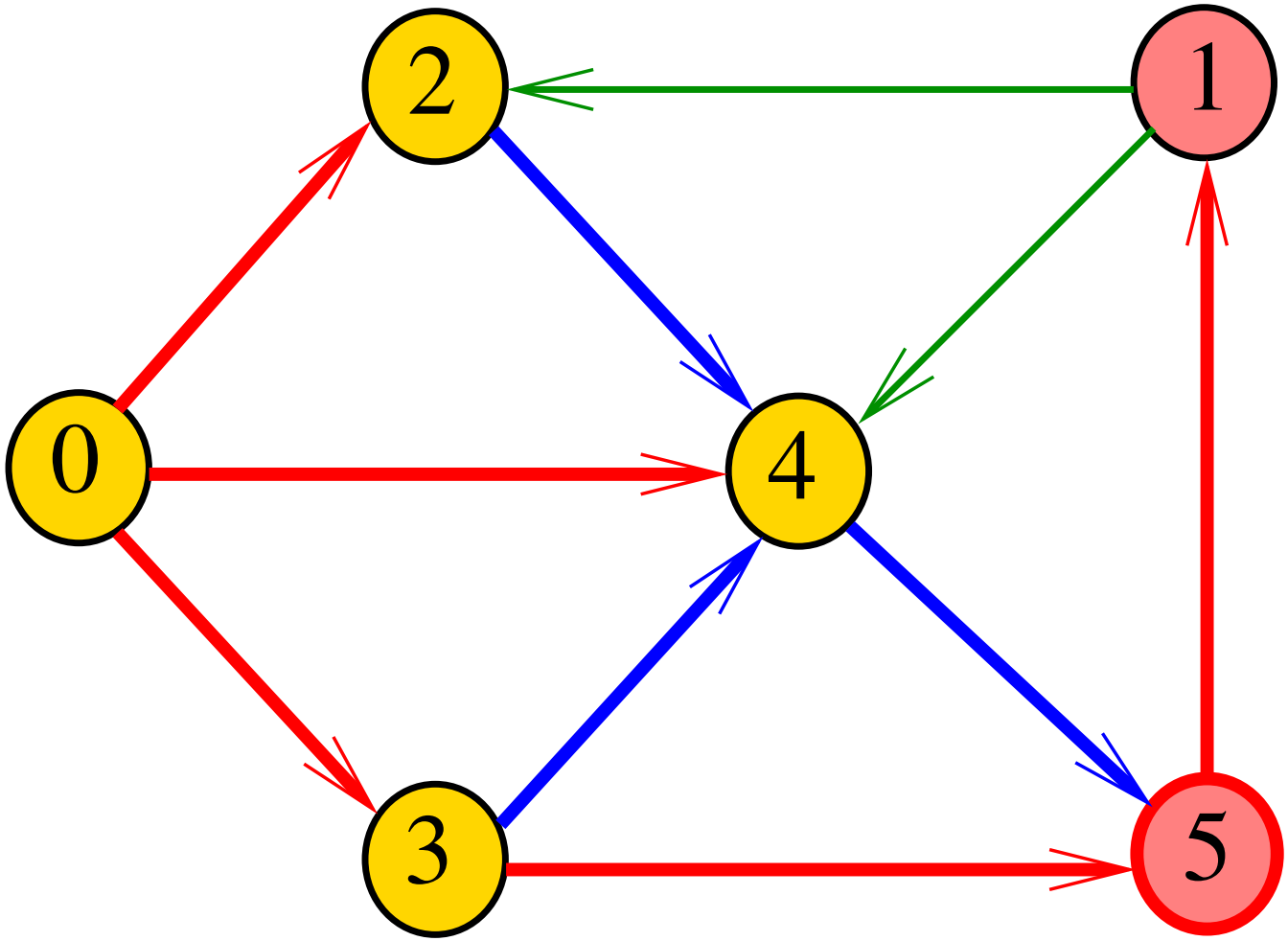
Simulação

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



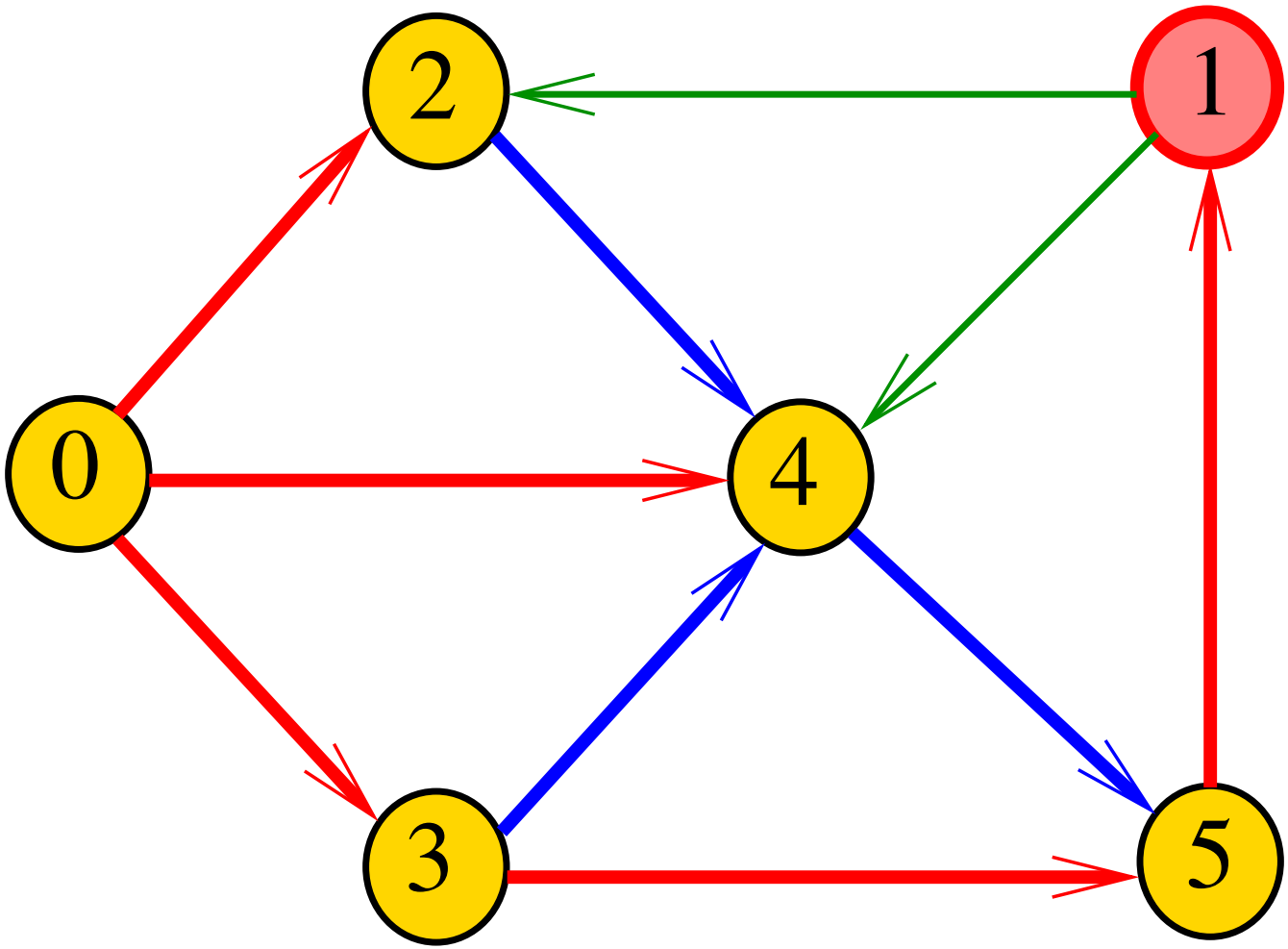
Simulação

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



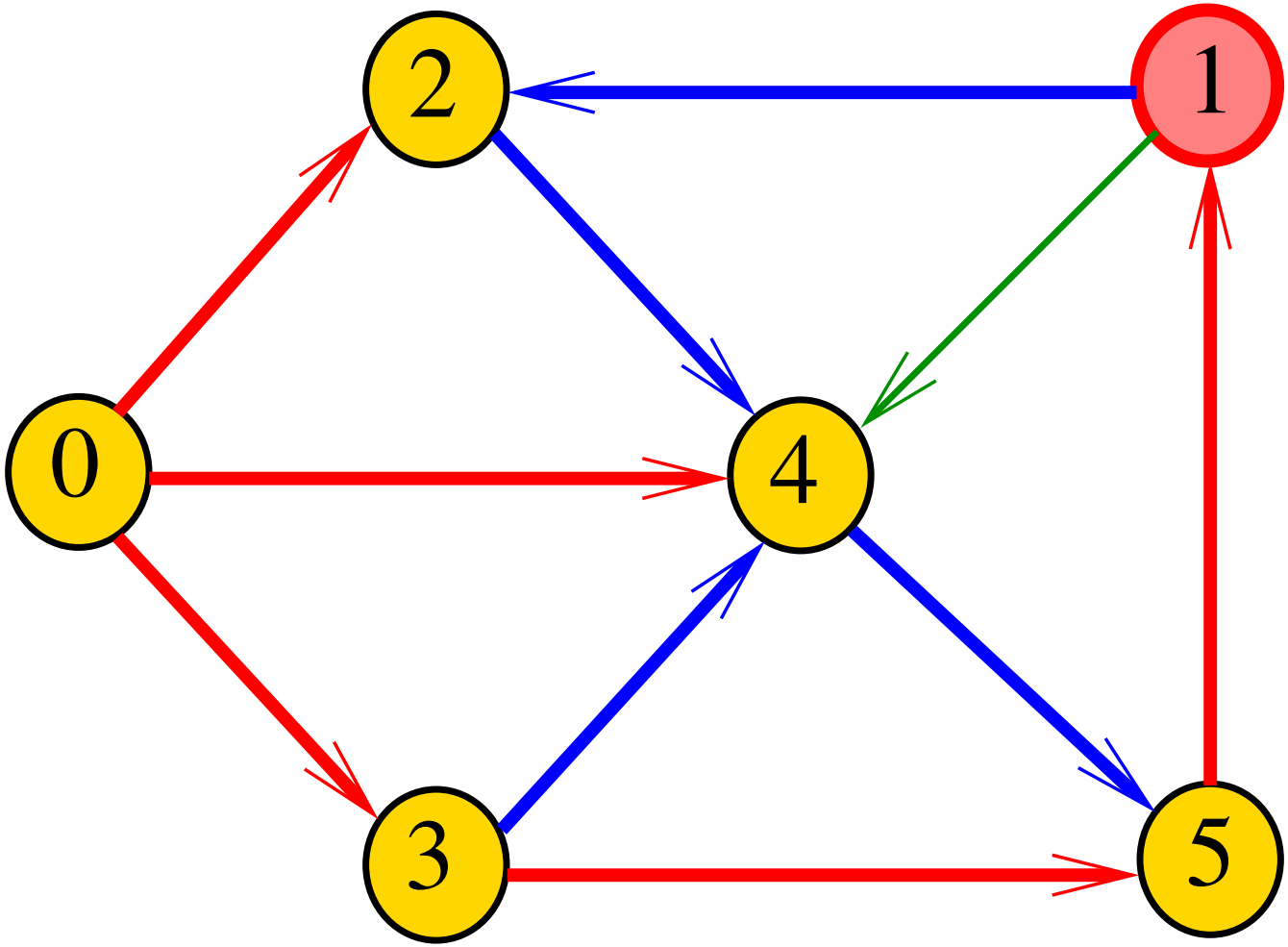
Simulação

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



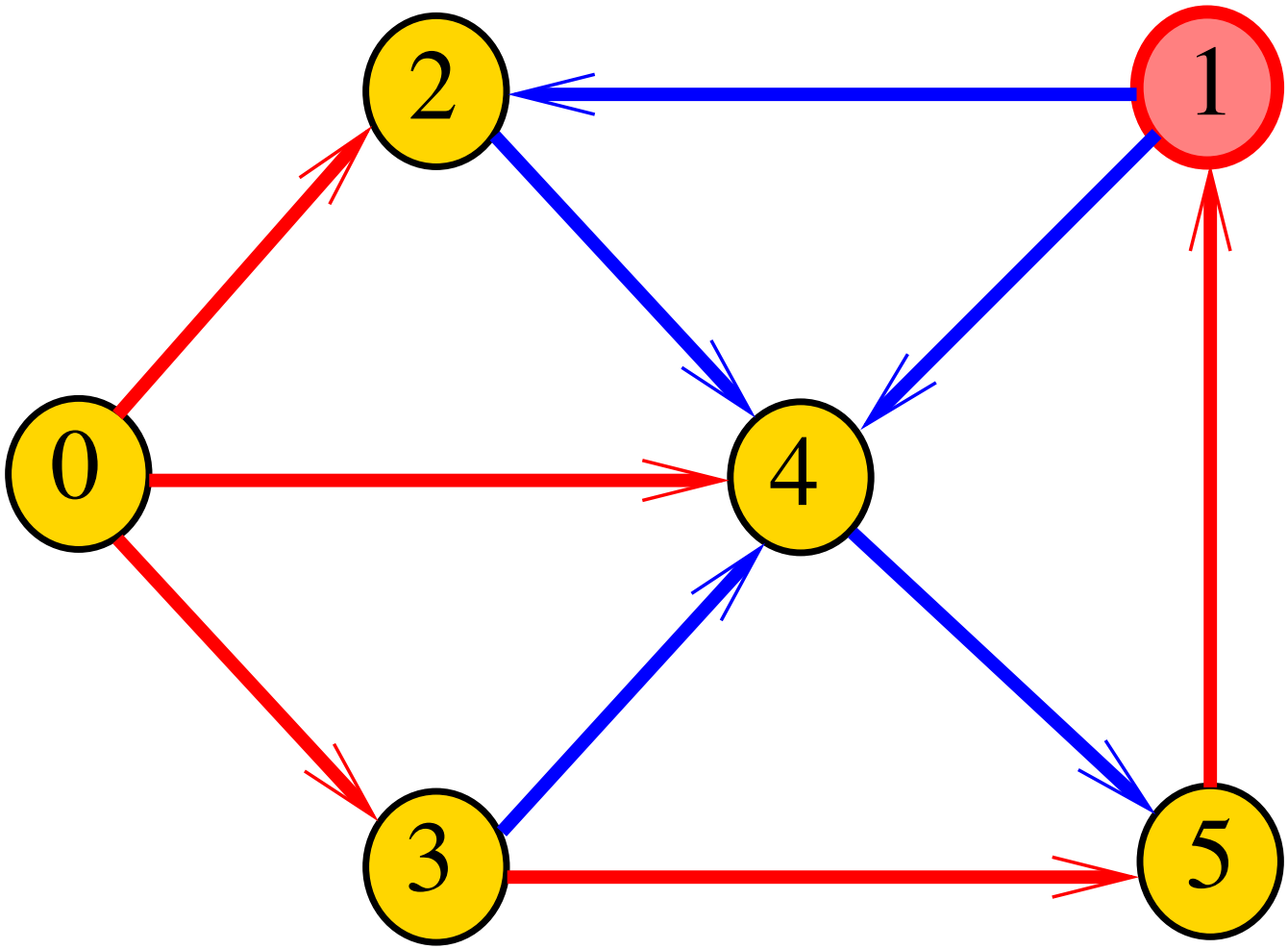
Simulação

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



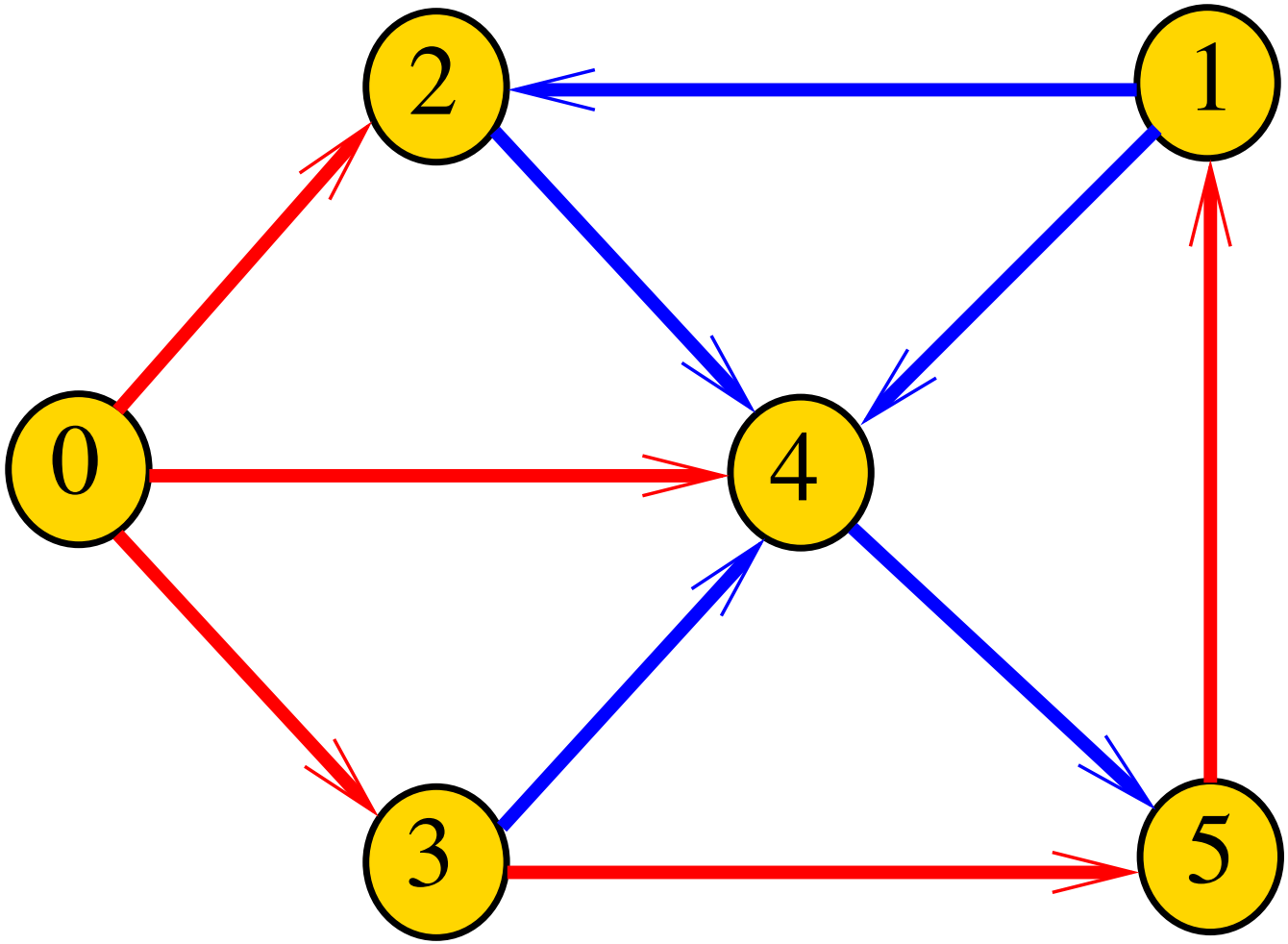
Simulação

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



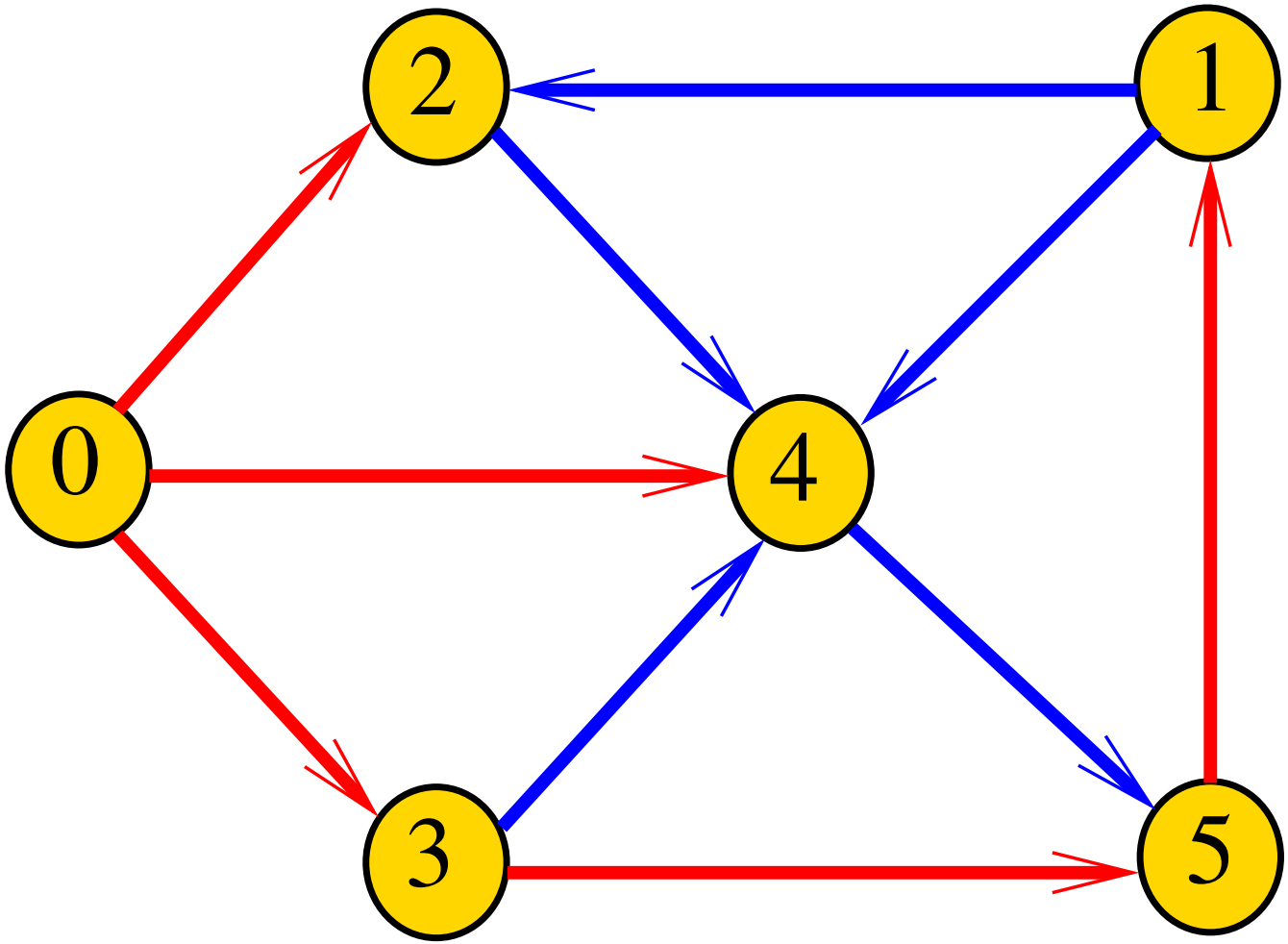
Simulação

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 uma *arborescência* com raiz `s`.



Arborescência

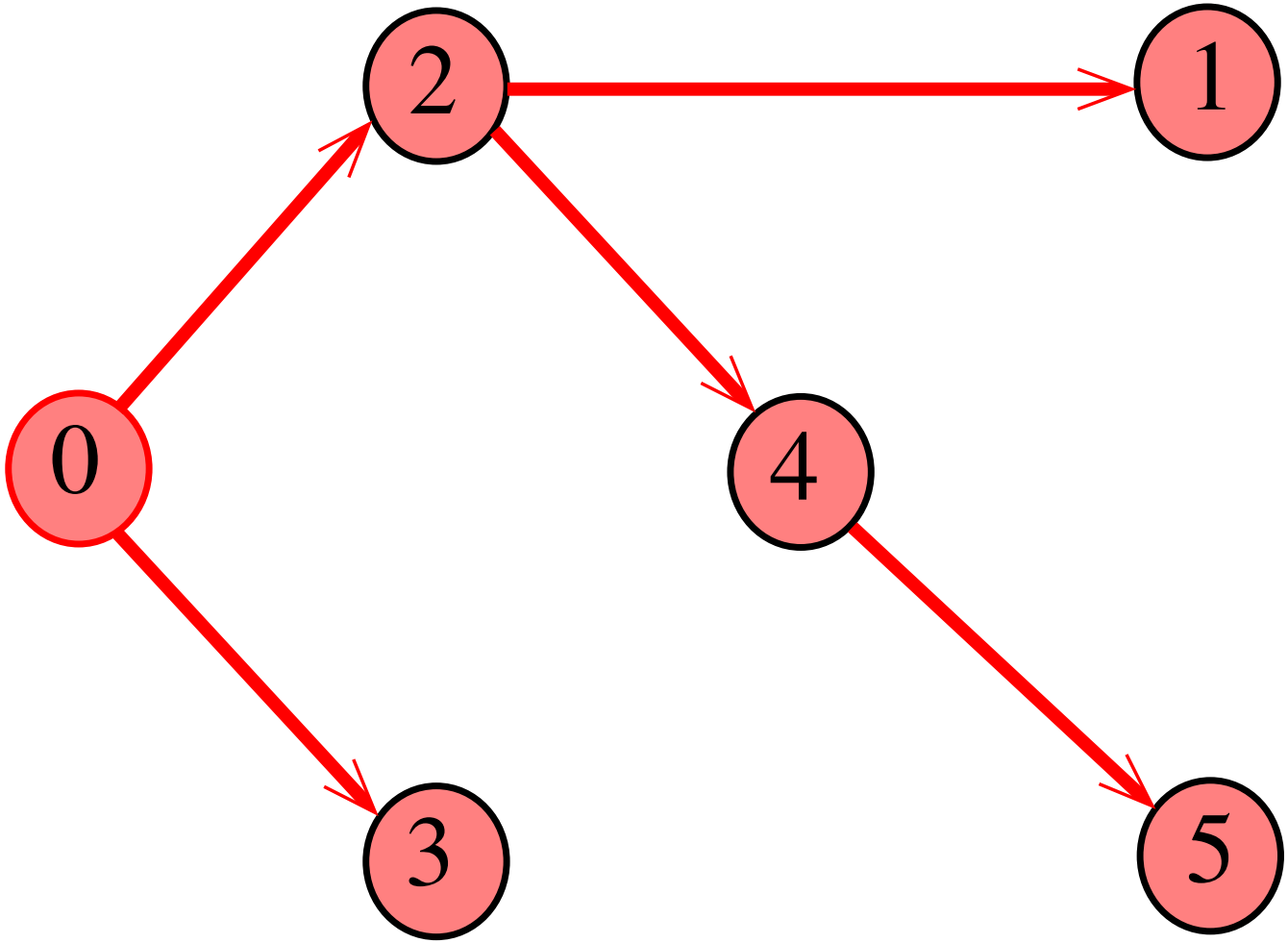
Uma **arborescência** é um digrafo em que

- existe exatamente um vértice com grau de entrada 0, a **raiz** da arborescência;
- não existem vértices com grau de entrada maior que 1;
- cada um dos vértices é término de um caminho com origem no vértice **raiz**.

É uma maneira **compacta** de representar caminhos de um vértice a outros.

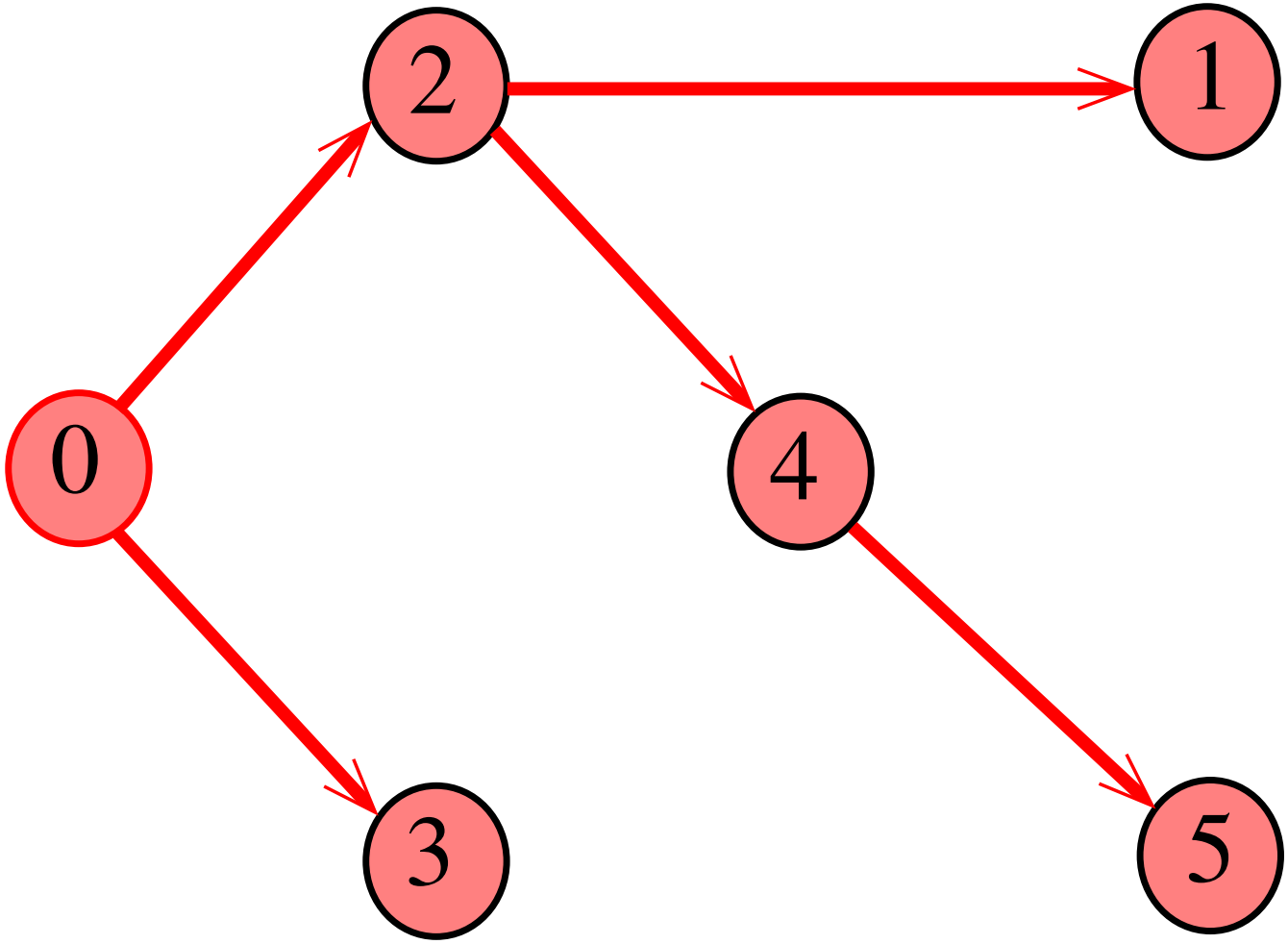
Arborescência

Propriedade: para todo vértice v,
existe exatamente um caminho da raiz a v.



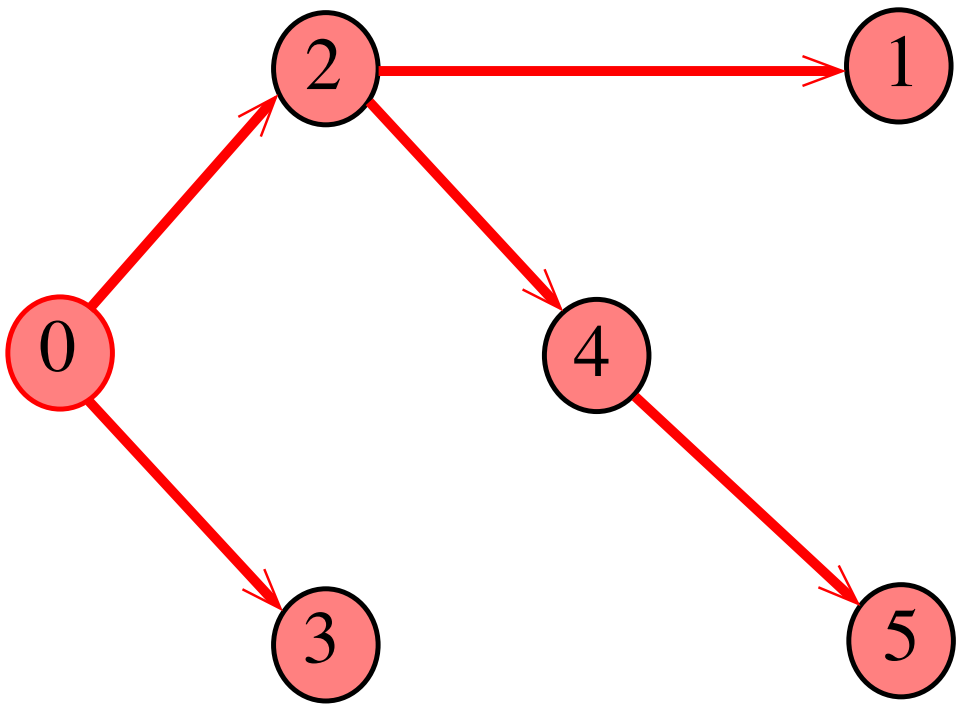
Arborescência

Todo vértice `w`, exceto a raiz, tem um **pai**:
o **único** vértice `v` tal que `v-w` é um arco.



Arborescência

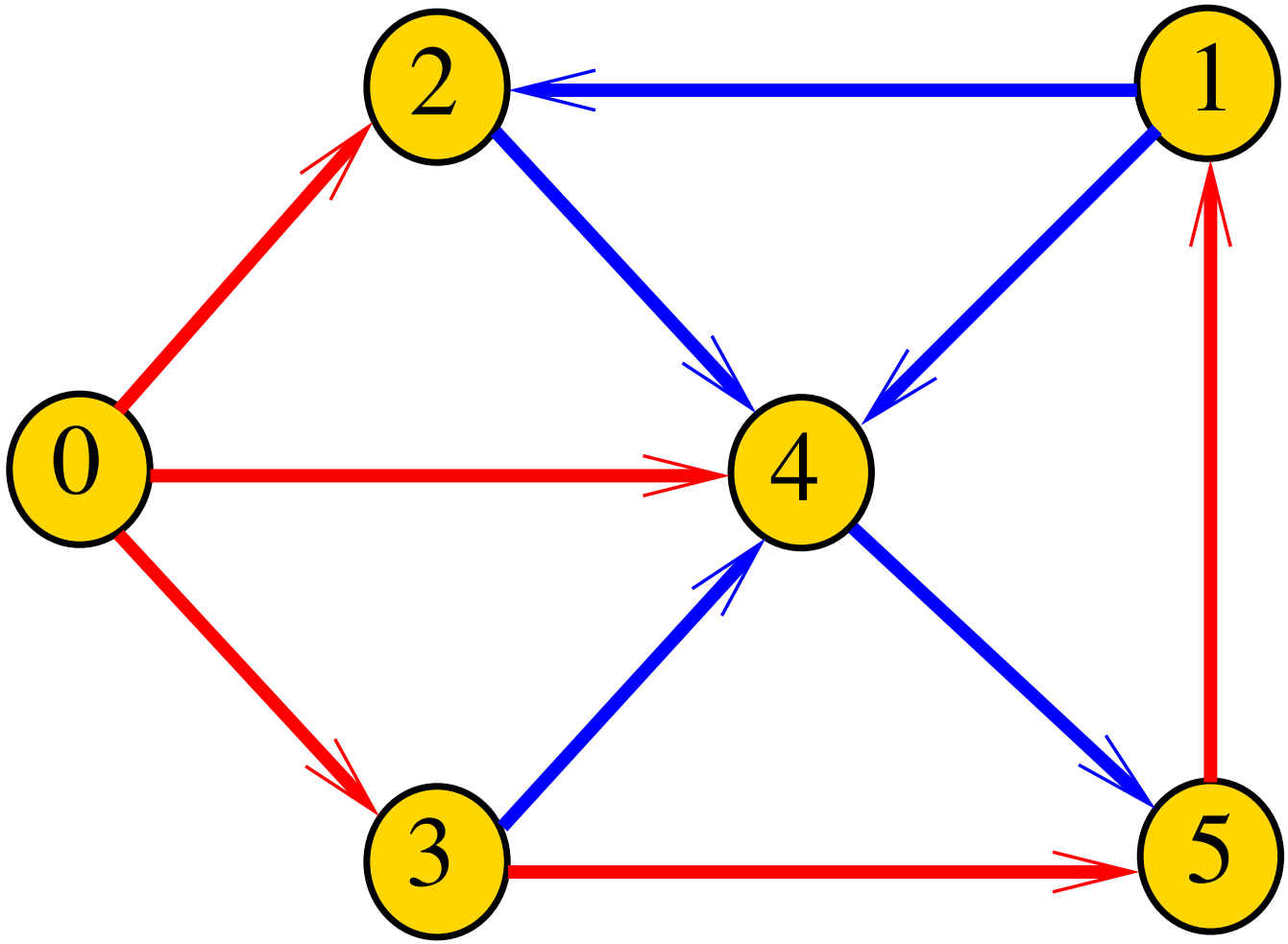
Uma arborescência pode ser representada através de um **vetor de pais**: `edgeTo[w]` é o pai de `w`. Se `r` é a raiz, então `edgeTo[r]=r`.



vértice	edgeTo
0	0
1	2
2	0
3	0
4	2
5	4

Arborescência da BFS

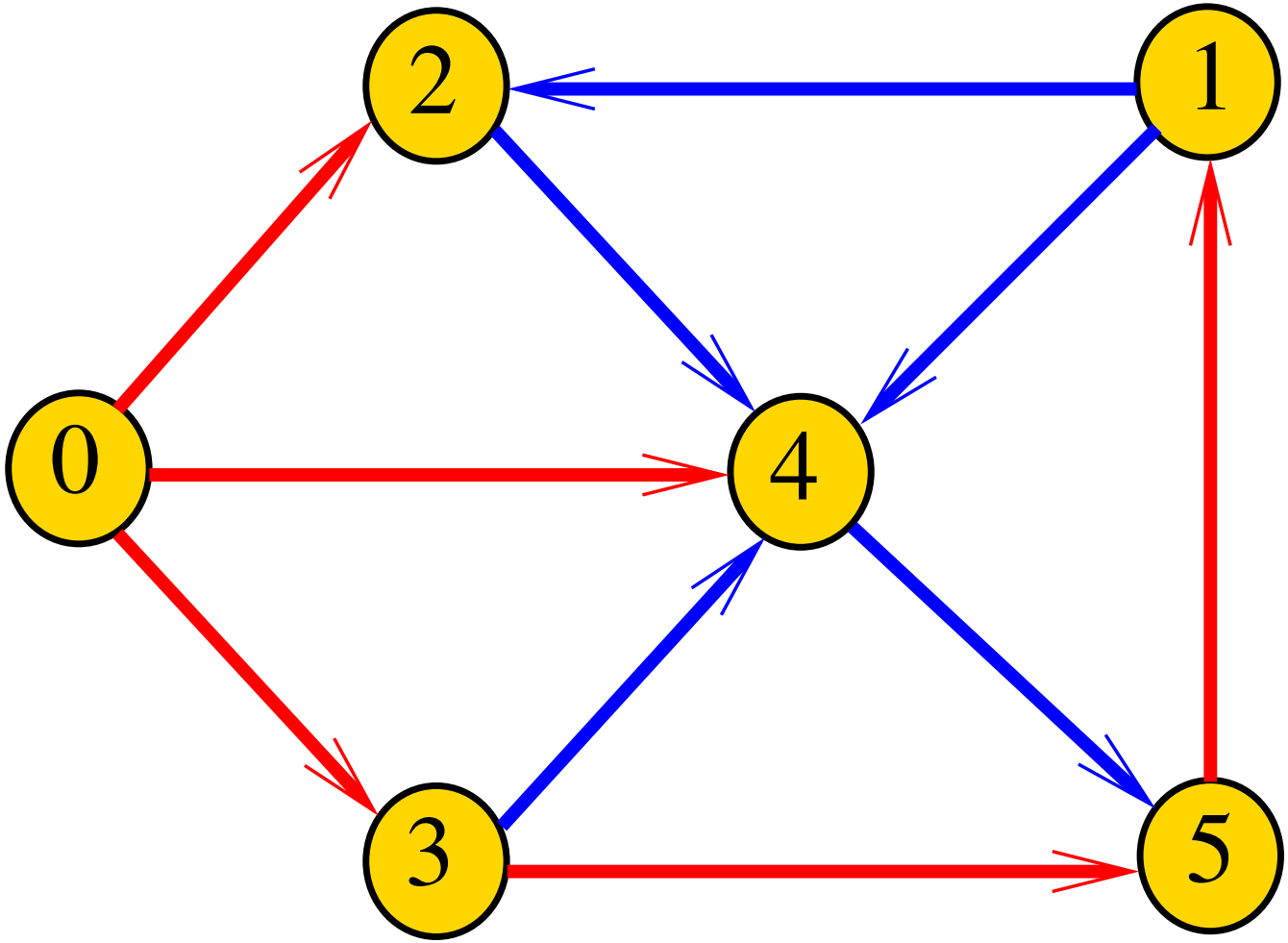
A arborescência obtida da BFS é conhecida como **arborescência de busca em largura** (= *BFS tree*).



Arborescência da BFS

Representação:

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



Caminho

Dado o vetor de pais, `edgeTo`, de uma arborescência, é fácil determinar o caminho que leva da raiz a um dado vértice `v`: basta inverter a sequência impressa pelo seguinte fragmento de código:

Caminho

Dado o vetor de pais, edgeTo, de uma arborescência, é fácil determinar o caminho que leva da raiz a um dado vértice v: basta inverter a sequência impressa pelo seguinte fragmento de código:

```
for (int x=v; edgeTo[x]!=x; x=edgeTo[x])
   printf("%d-", x);
printf("%d\n", x);                    /* raiz */
```

Implementação de BFSpaths

BFSpaths 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 visitado então marked[v] == true.

Para isso BFSpaths usará uma fila de vértices.

BFSpaths: estrutura

```
static struct bfspaths {
   int s;
   bool *marked;
   int  *edgeTo;
};
typedef struct bsfpaths *bfsPaths;
```

BFSpaths: Init

```
bfsPaths BFSpathsInit(Digraph G, int s) {
   bfsPaths T = mallocSafe(sizeof(*T));
   T->s = s;
   T->marked = mallocSafe(G->V*sizeof(bool));
   T->edgeTo = mallocSafe(G->V*sizeof(int));
   for (int v = 0; v < G->V; v++) {
      T->marked[v] = false;
      T->edgeTo[v] = -1;
   }
   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) {...}

/* pilha com o caminho requisitado */
Stack 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

bfs usa uma fila de vértices.

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

   T->edgeTo[s] = s;
   T->marked[s] = true;
   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]) {
                T->edgeTo[w->vertex] = v;
                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->marked[v];
}
```

BFSpaths

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

Versão mais simples, que devolve o caminho numa pilha.

```
Stack pathTo(bfsPaths T, int v) {
   Stack path = stackInit(); int x;

   if (!hasPath(T, v)) return path;
   for (x = v; x != T->s; x = T->edgeTo[x])
      stackPush(path, x);
   stackPush(path, T->s);

   return path;
}
```

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

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

Busca em profundidade

Agora veremos uma outra maneira de visitar os vértices de um digrafo.

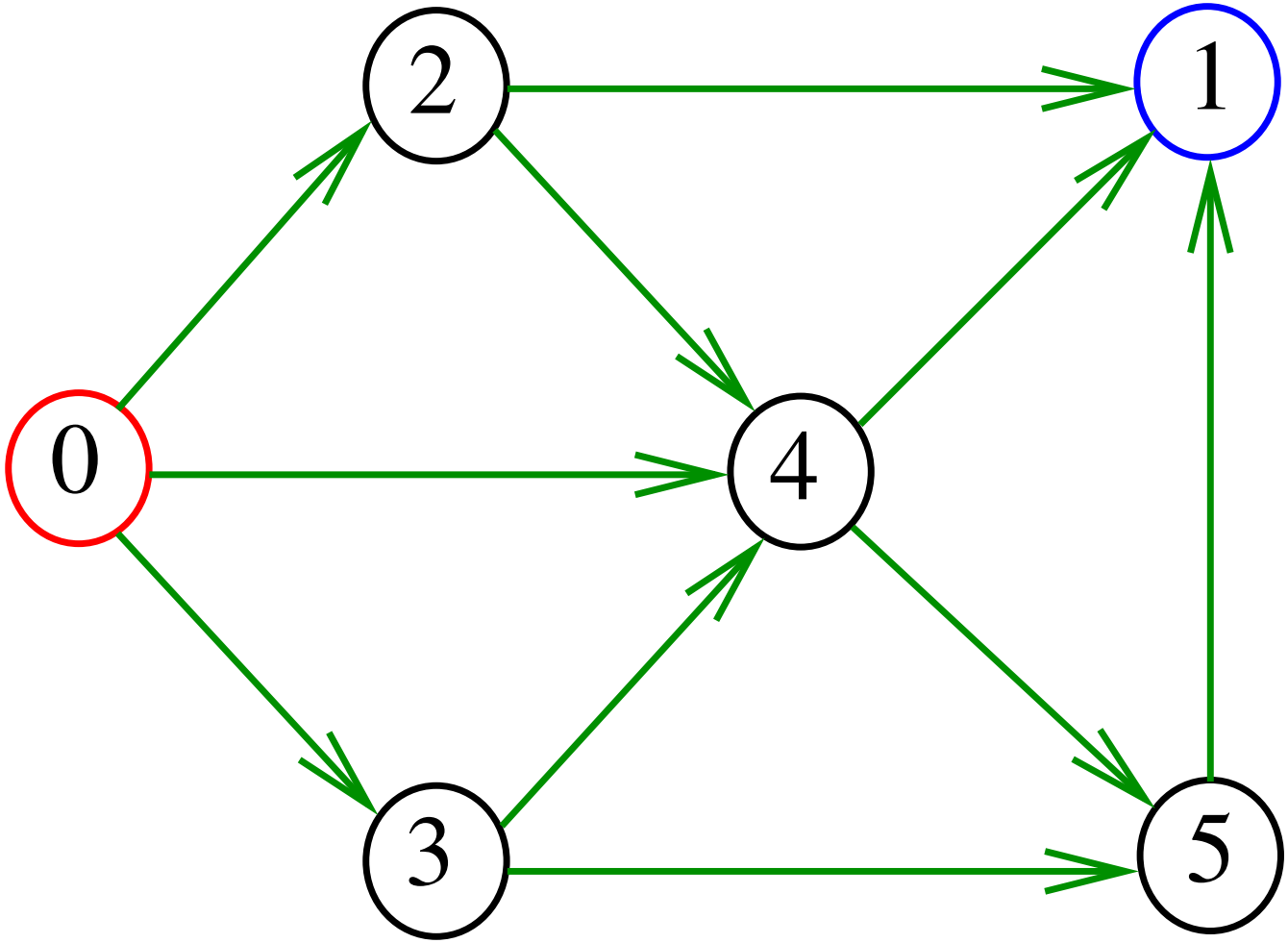
A rotina `DFSpaths` recebe um digrafo `G` e um vértice `s` e determina todos os vértices alcançáveis a partir de `s`.

A rotina implementa a técnica chamada busca em profundidade (*D*epth *F*irst *S*earch).

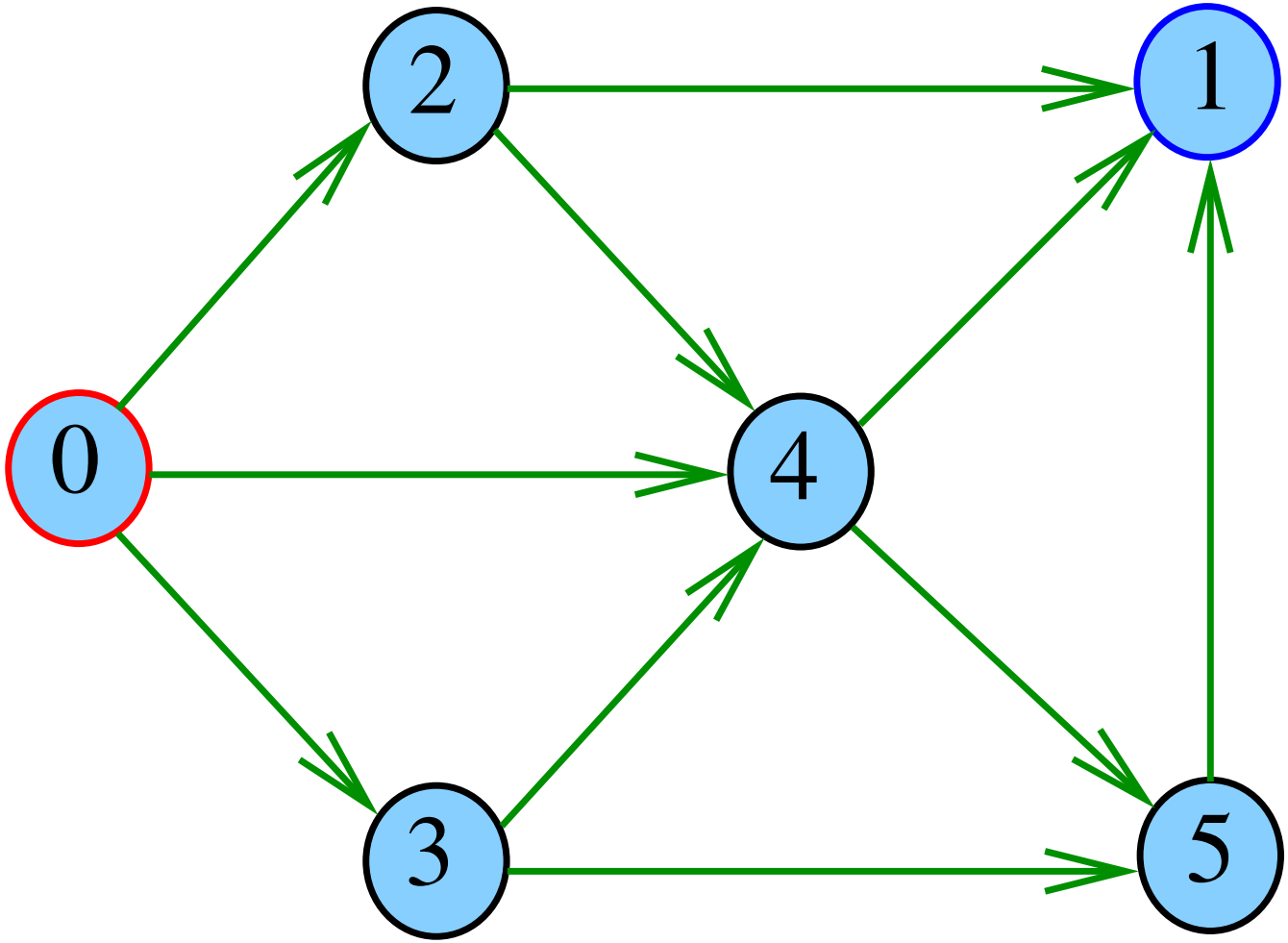
Ela utiliza uma estratégia recursiva.

DFSpaths(G, 0)

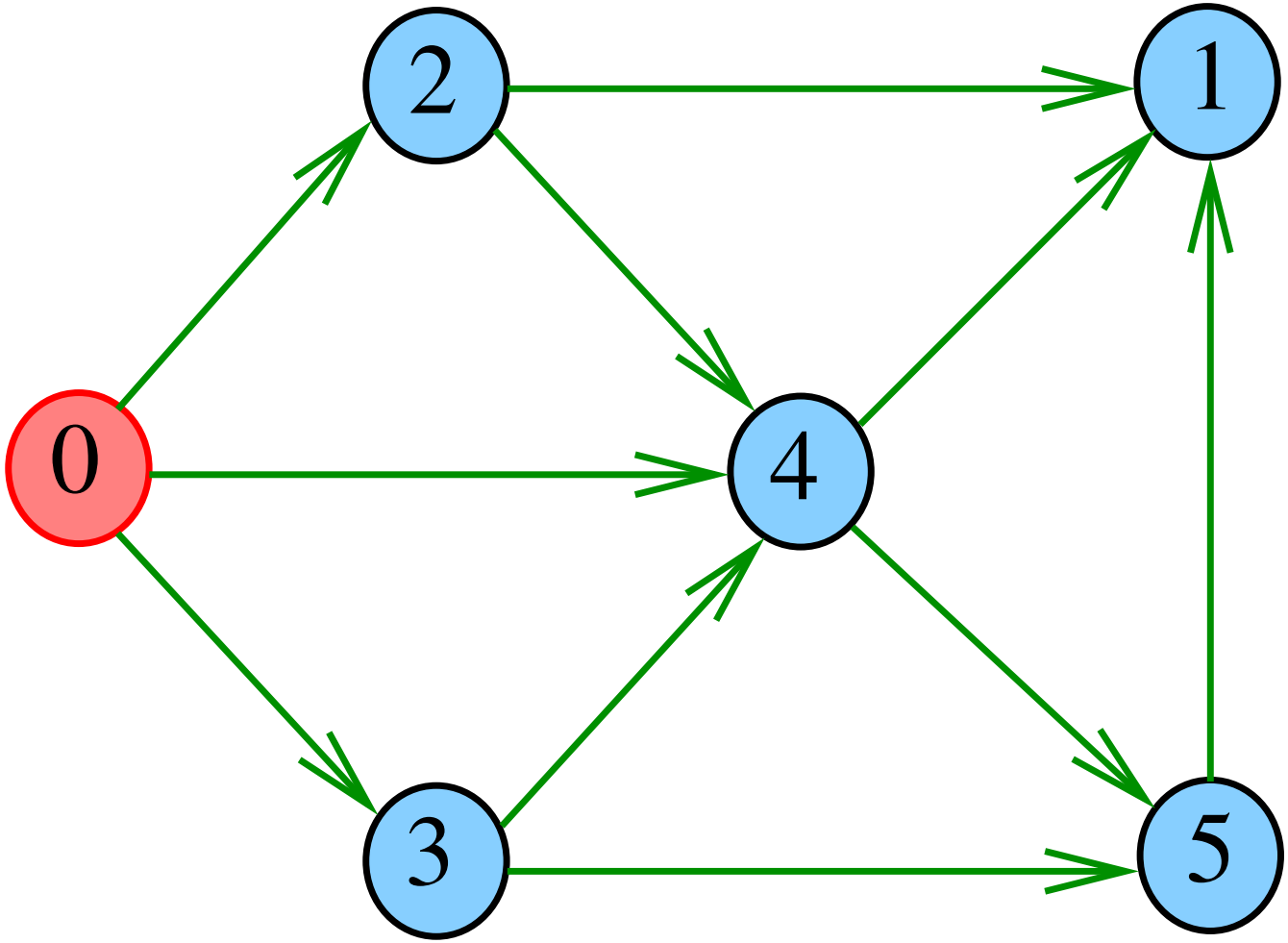
Existe caminho de 0 a 1?



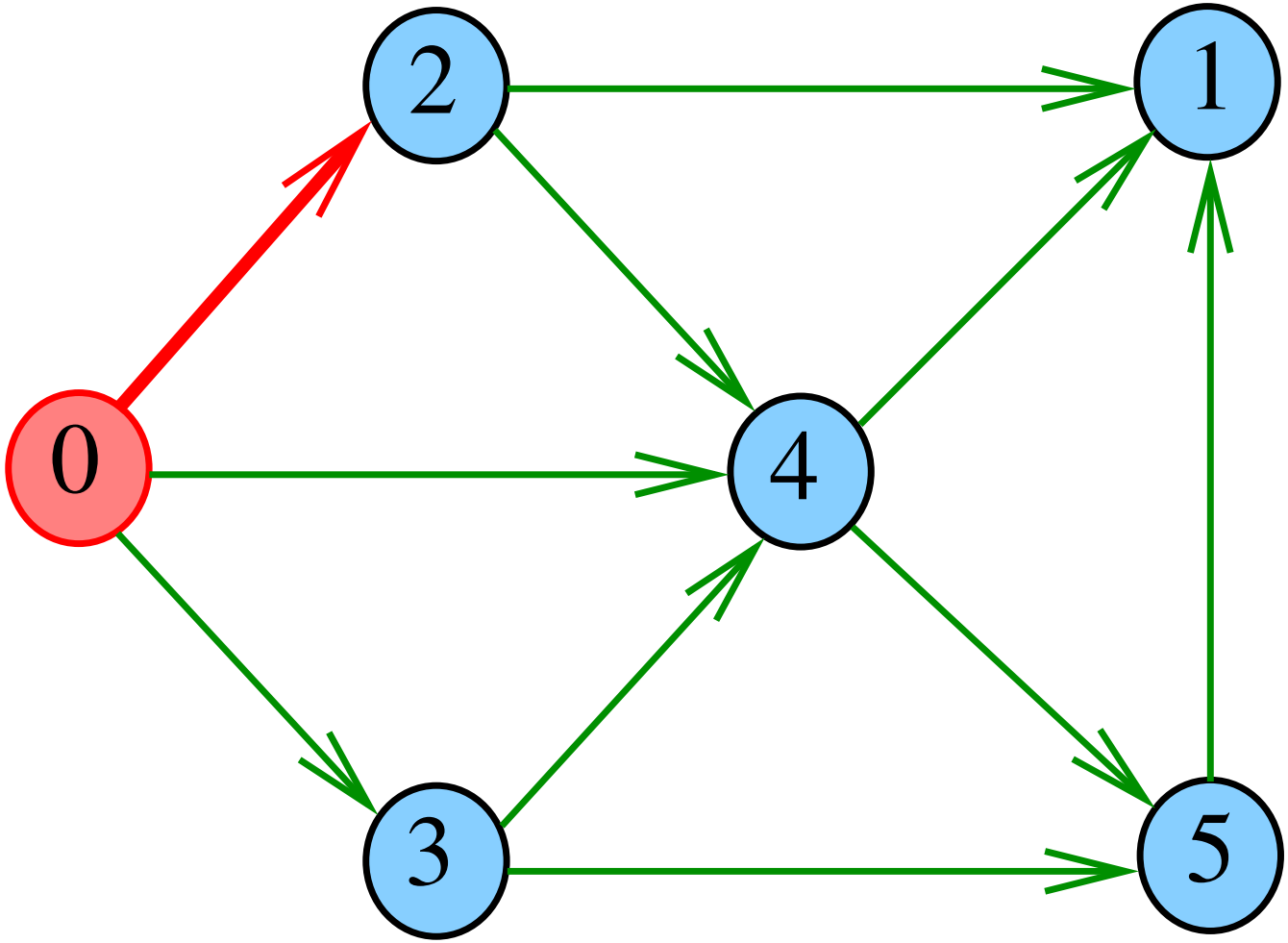
DFSpaths(G, 0)



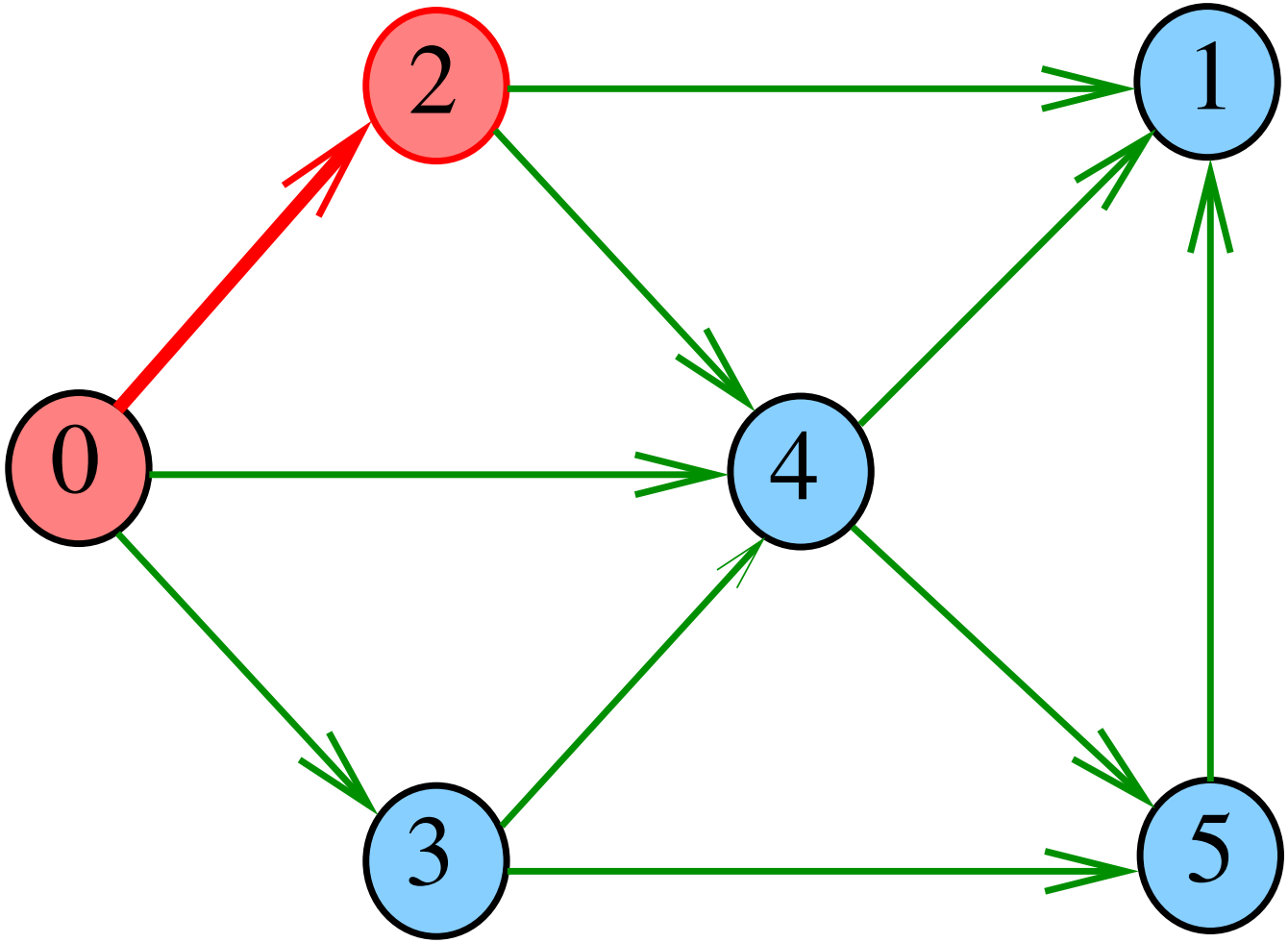
dfs(G, 0)



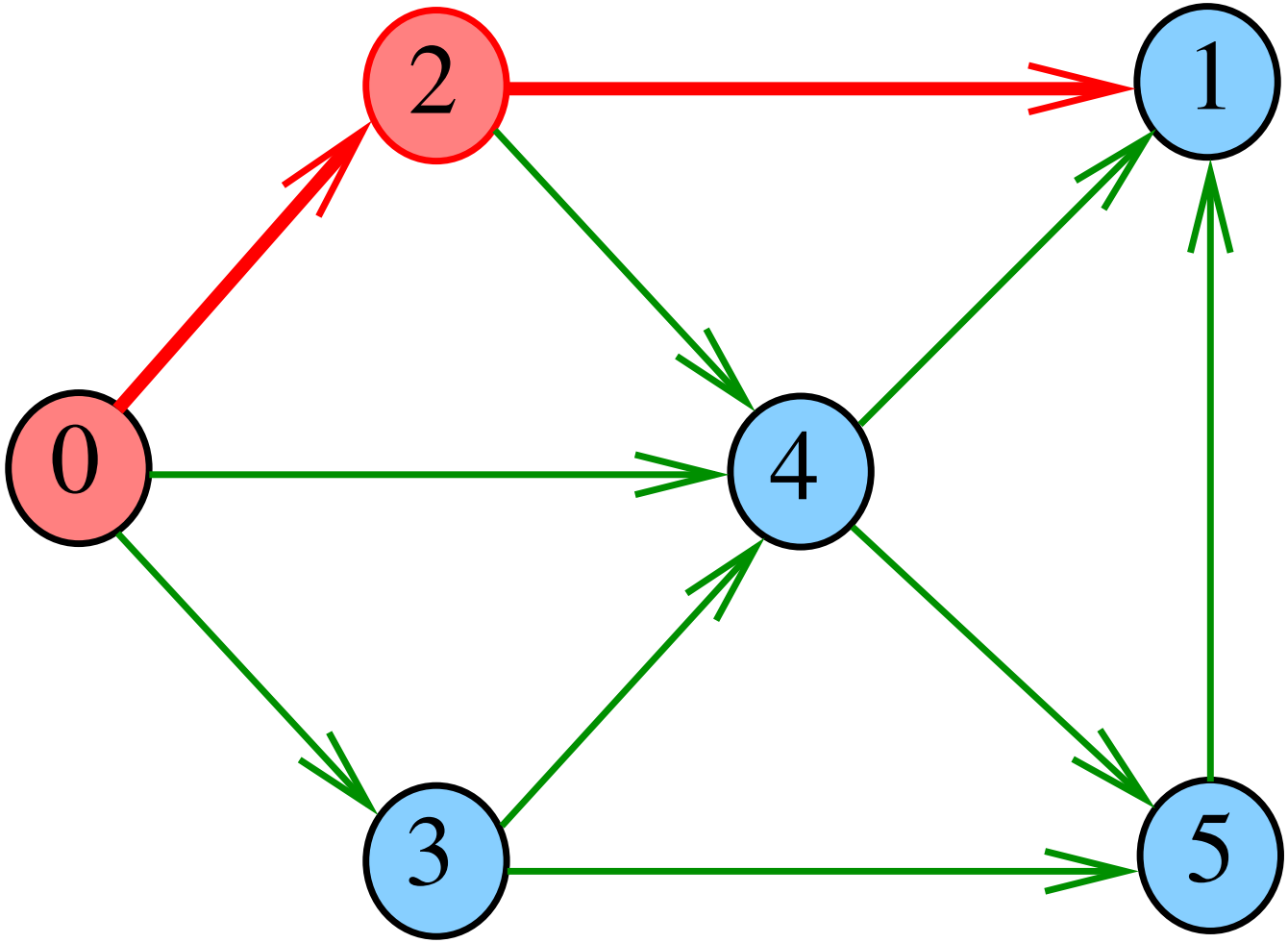
dfs(G, 0)



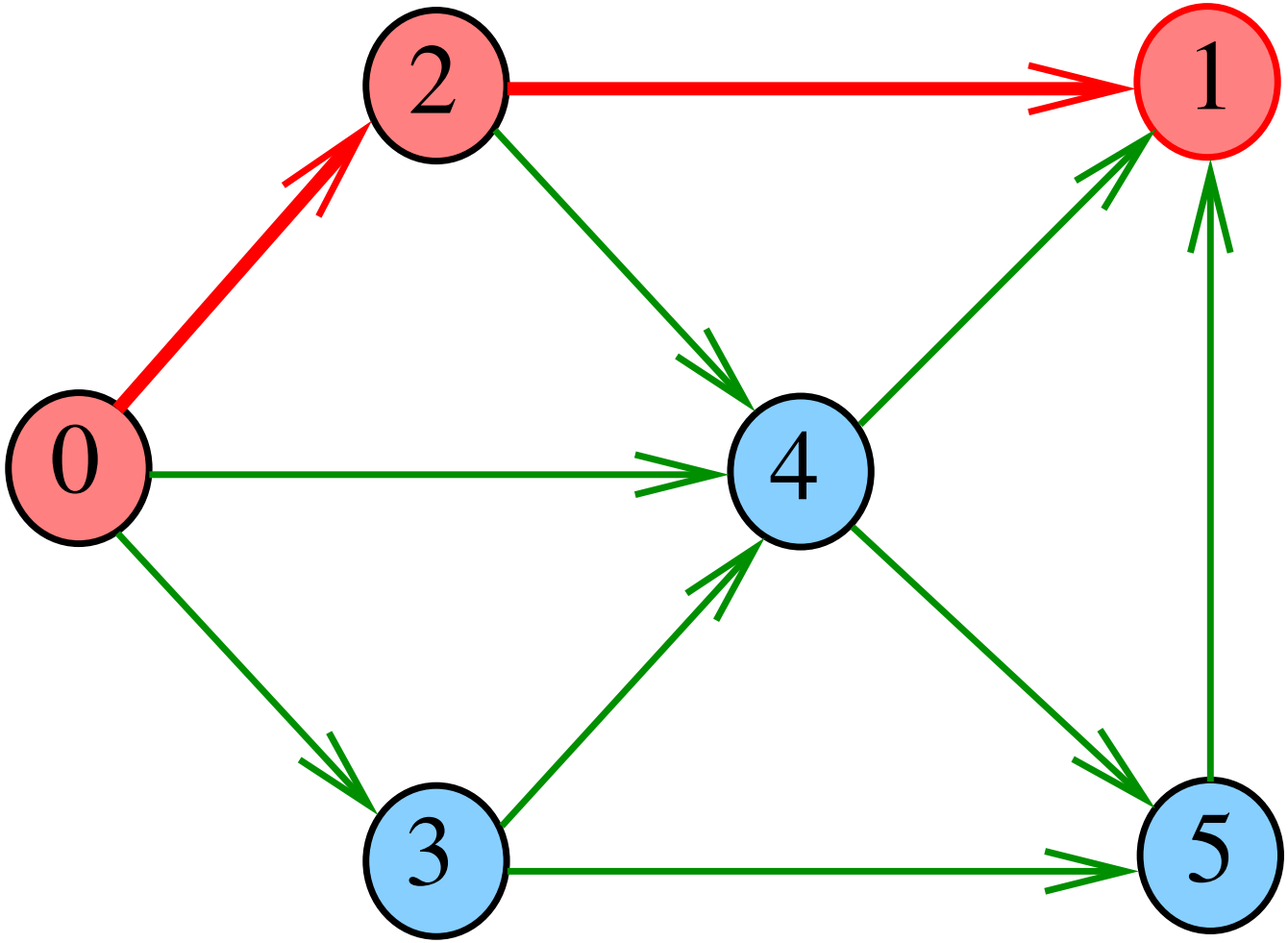
dfs(G, 2)



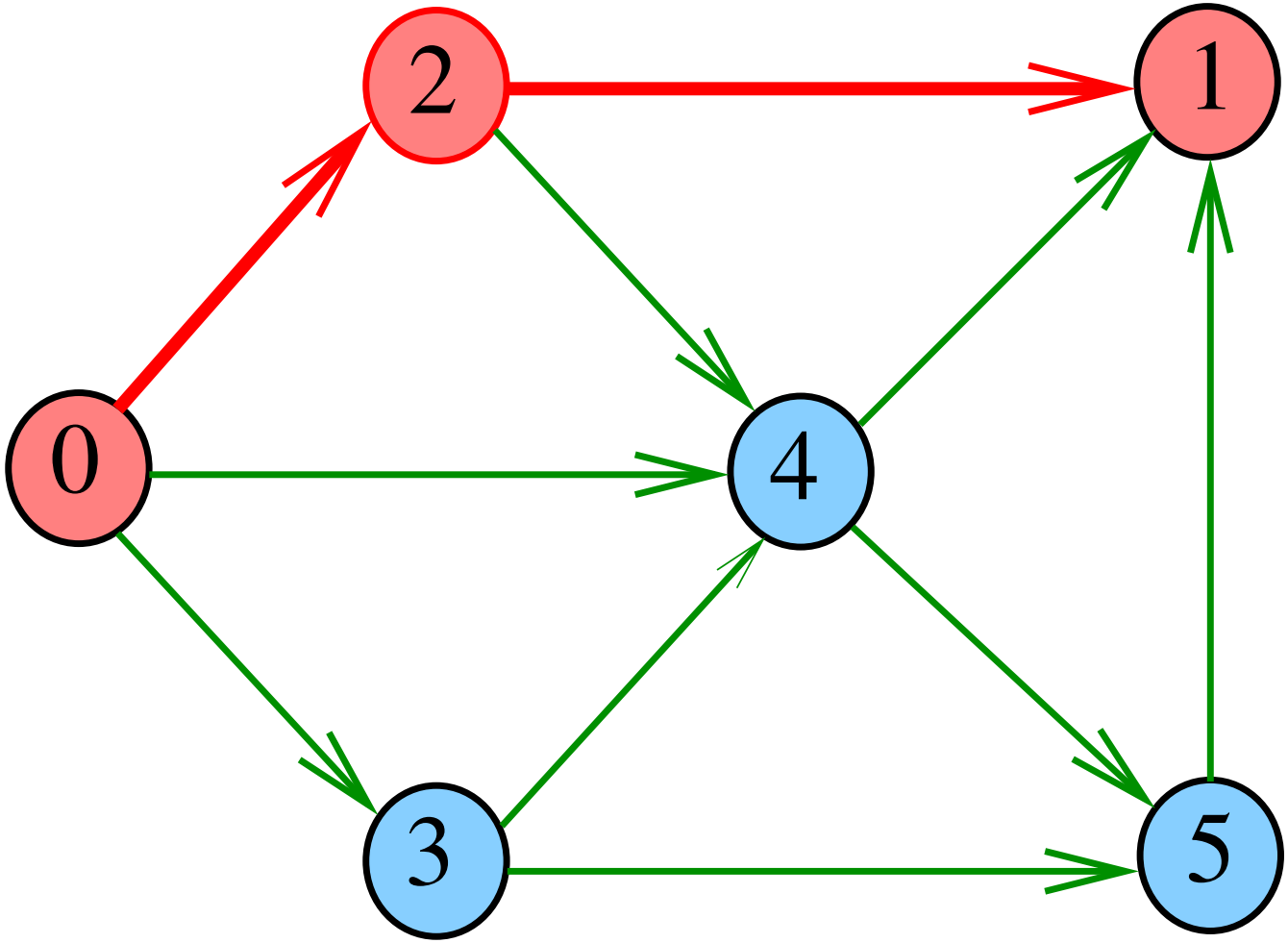
dfs(G, 2)



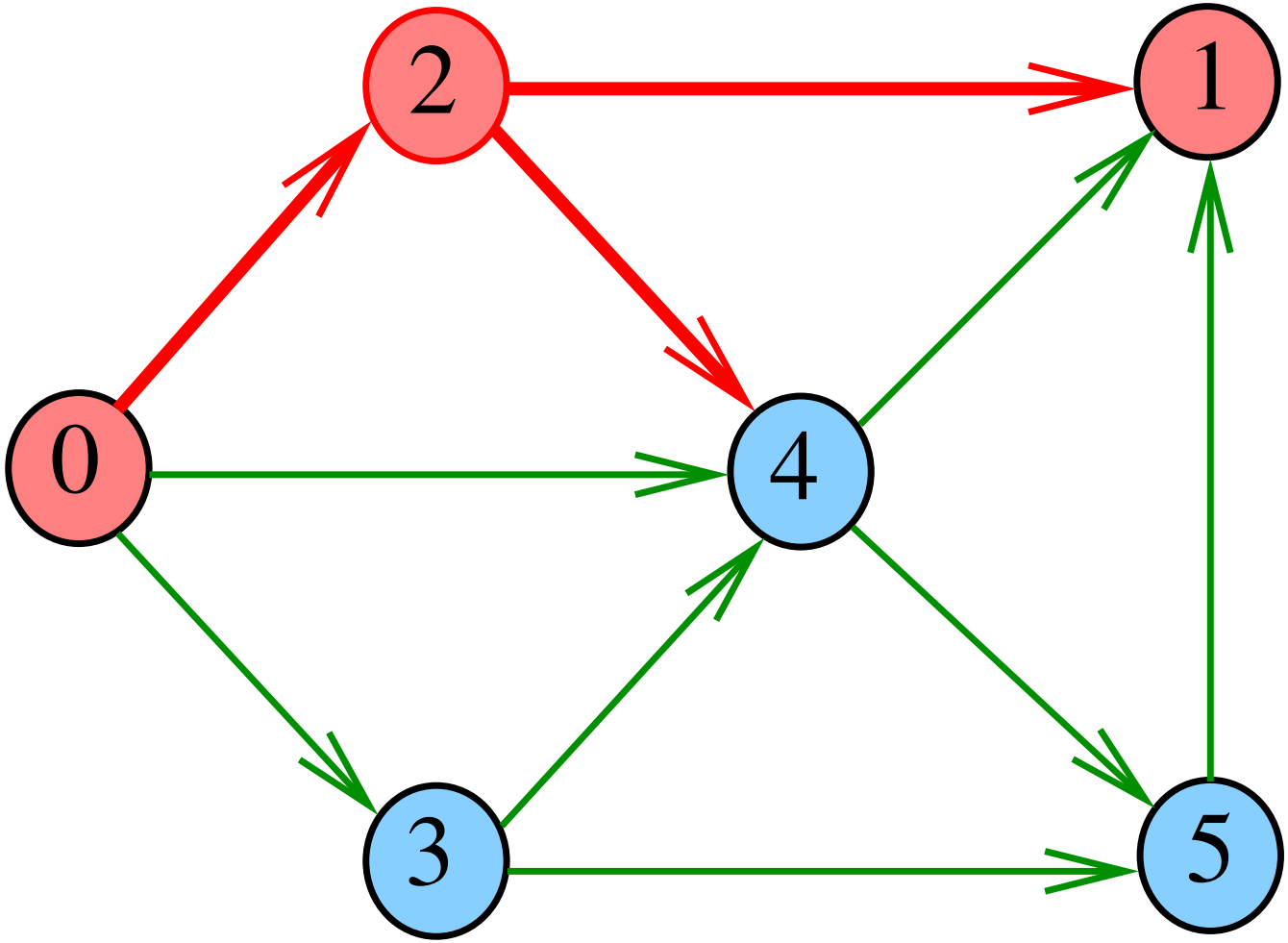
dfs(G, 1)



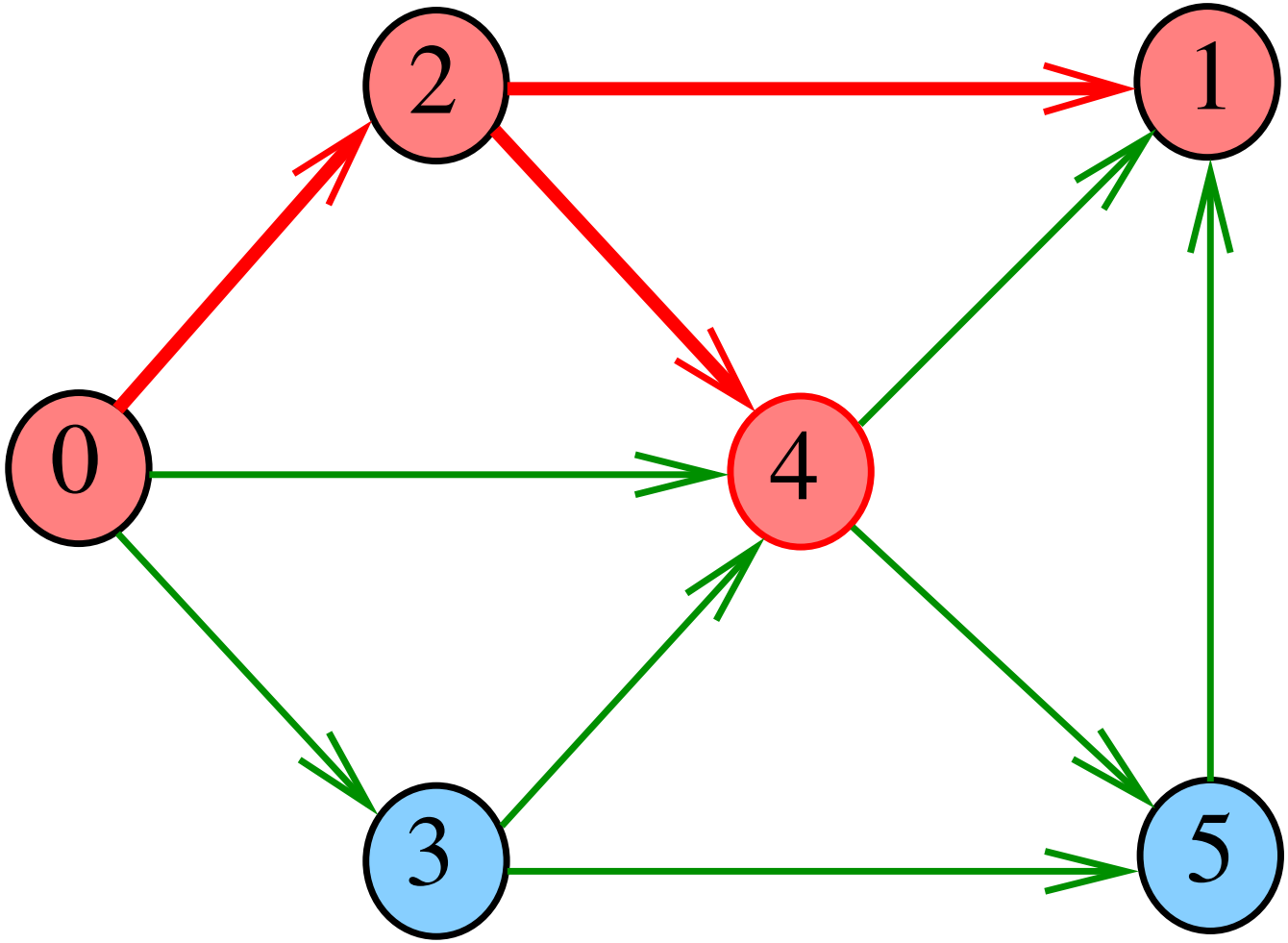
dfs(G, 2)



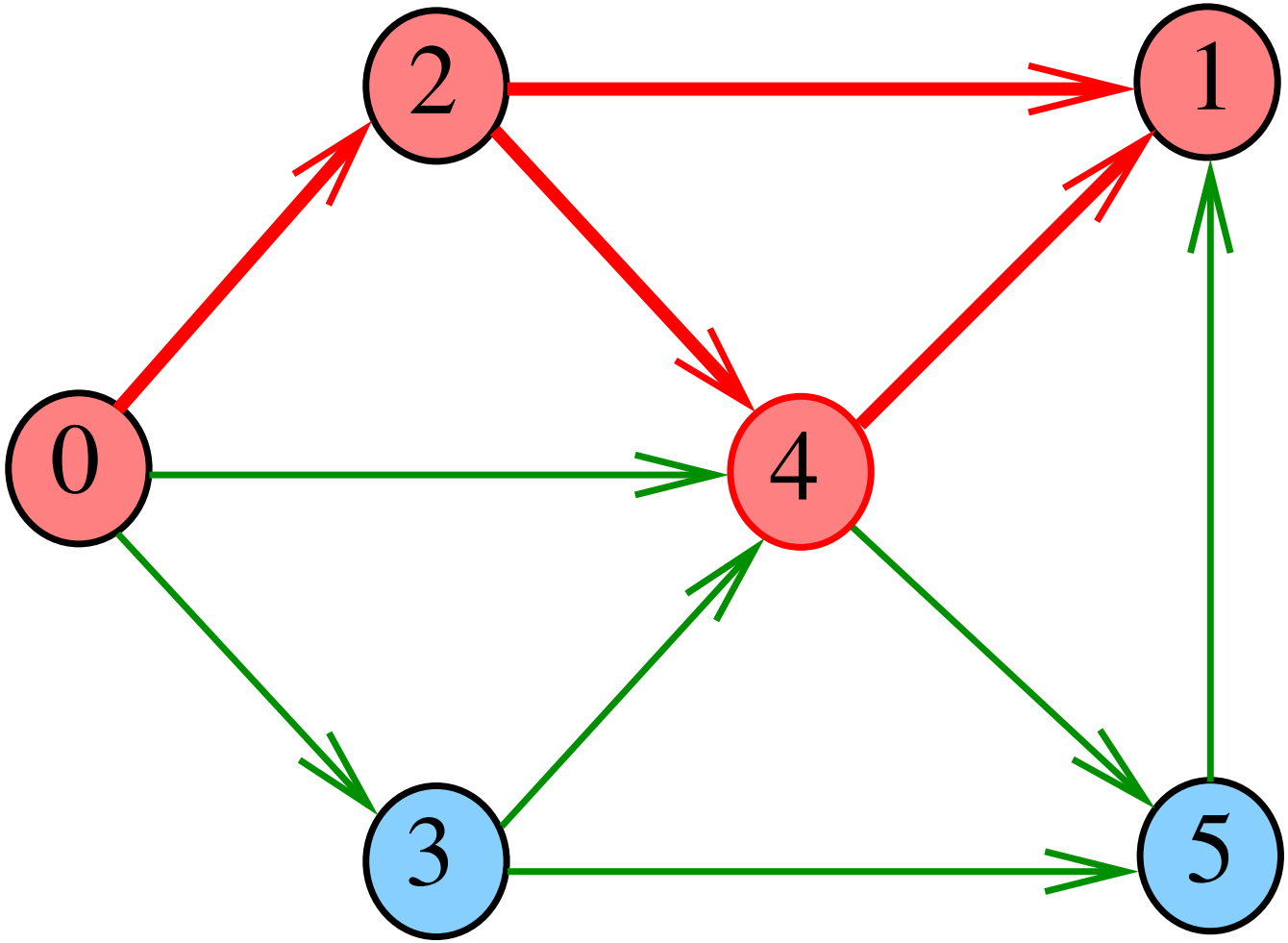
dfs(G, 2)



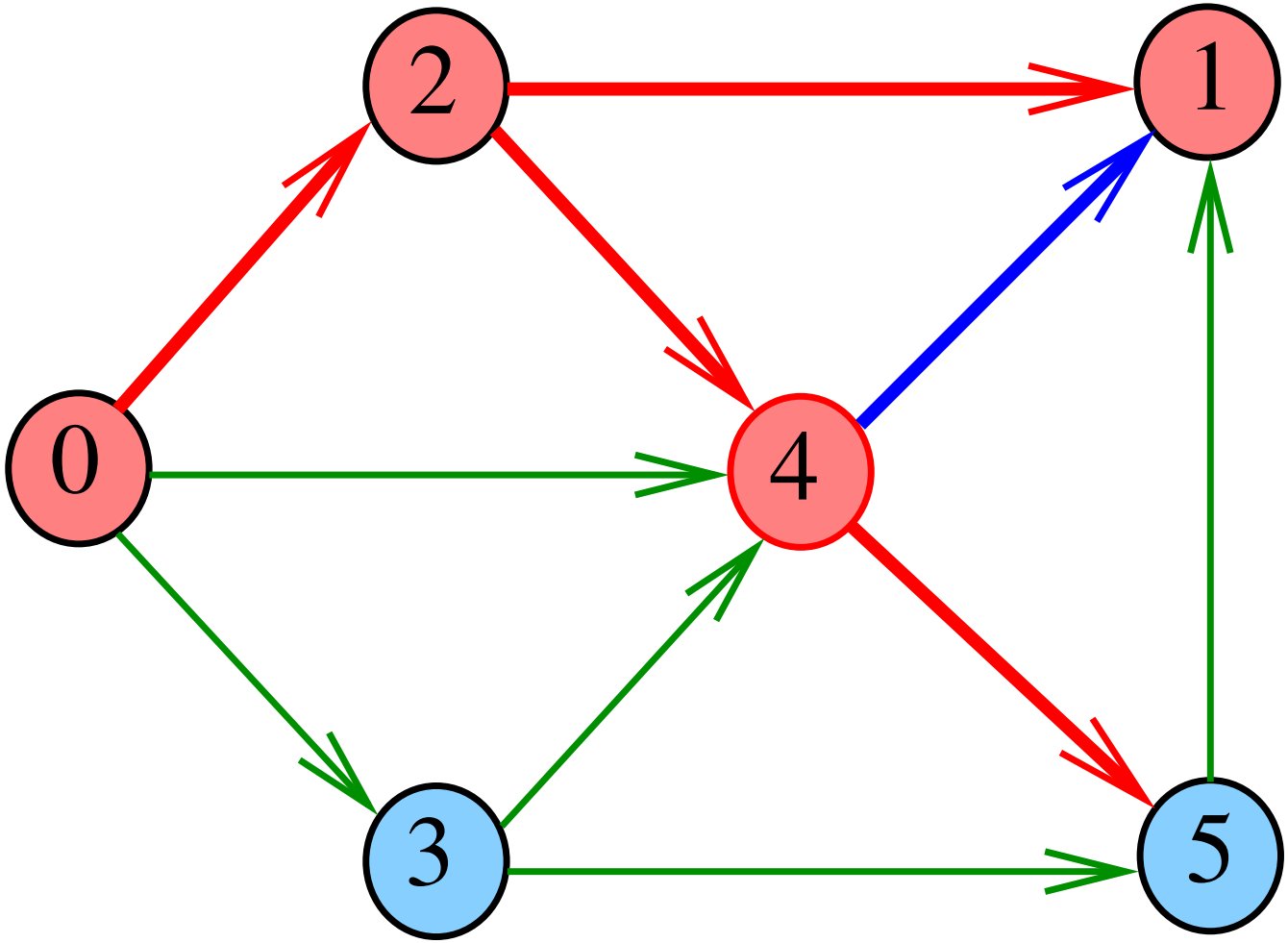
dfs(G, 4)



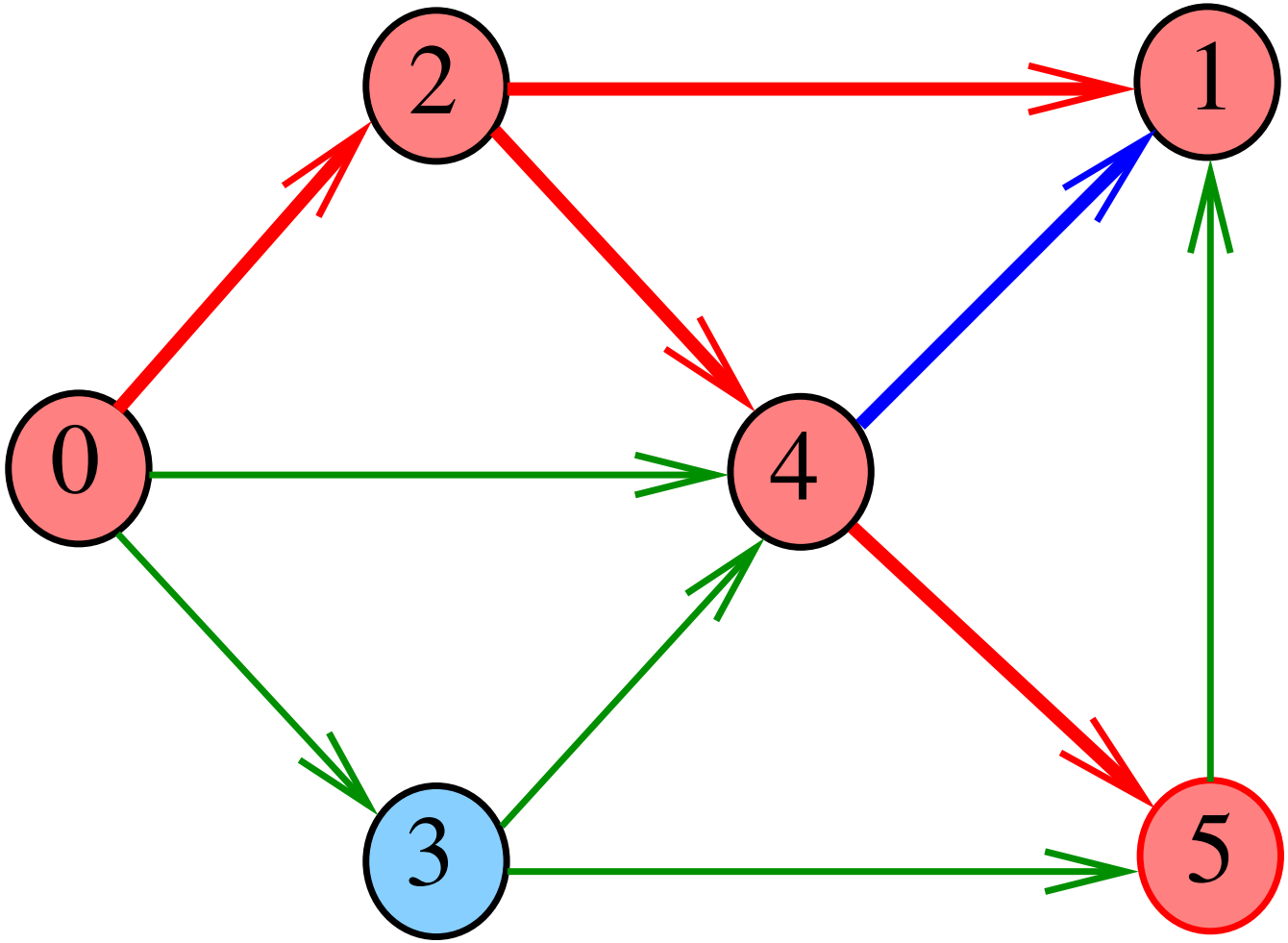
dfs(G, 4)



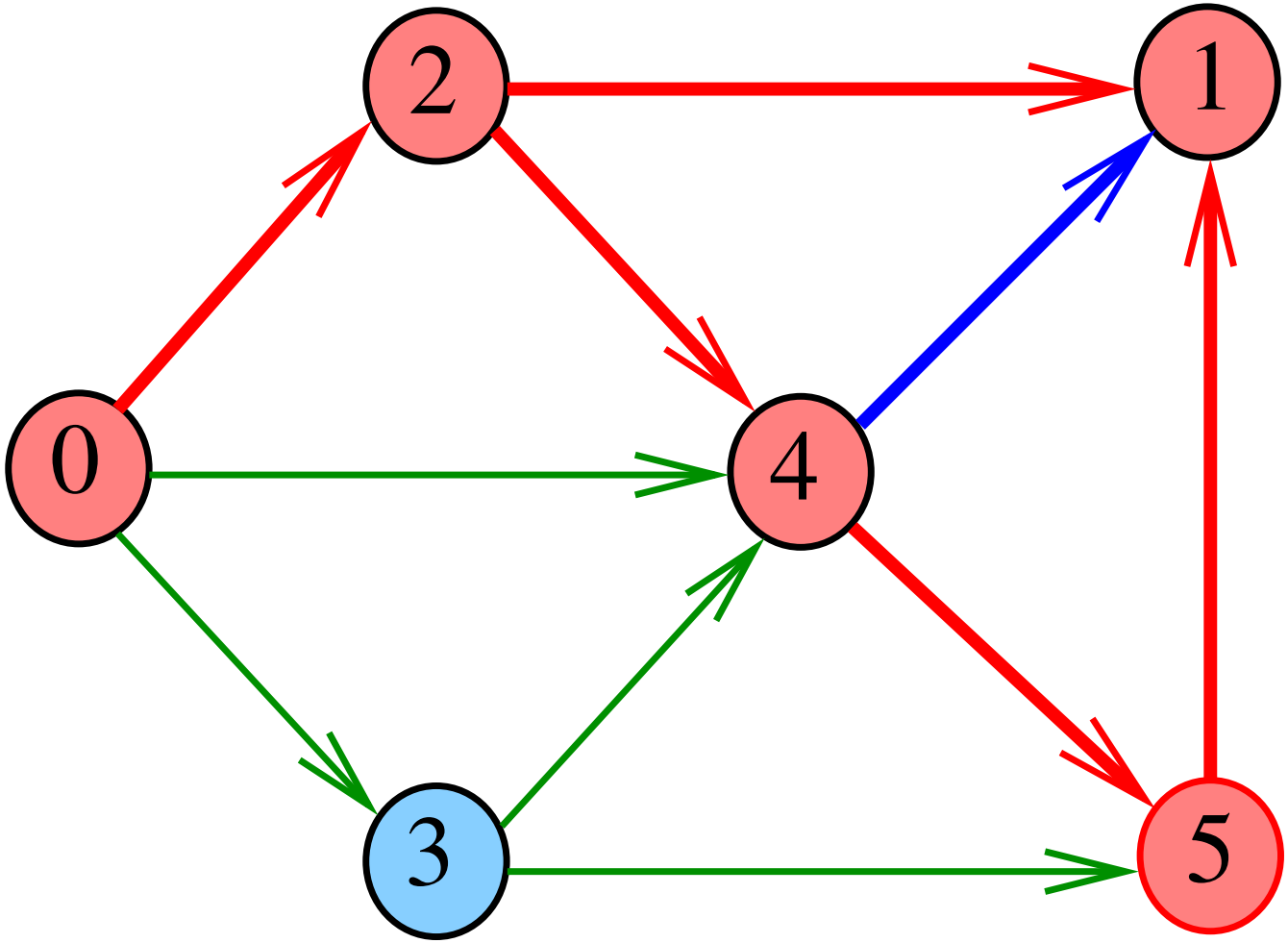
dfs(G, 4)



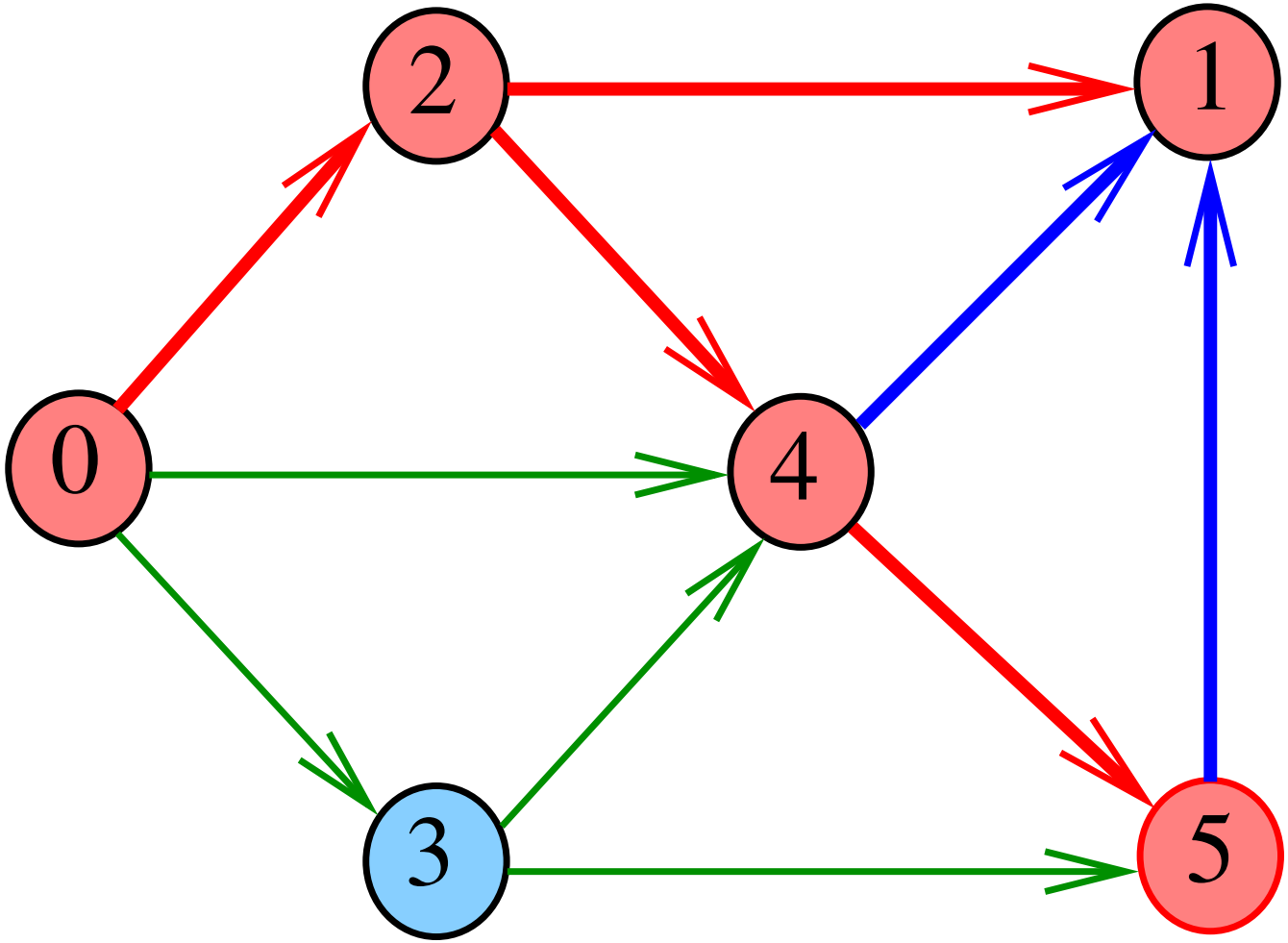
dfs(G, 5)



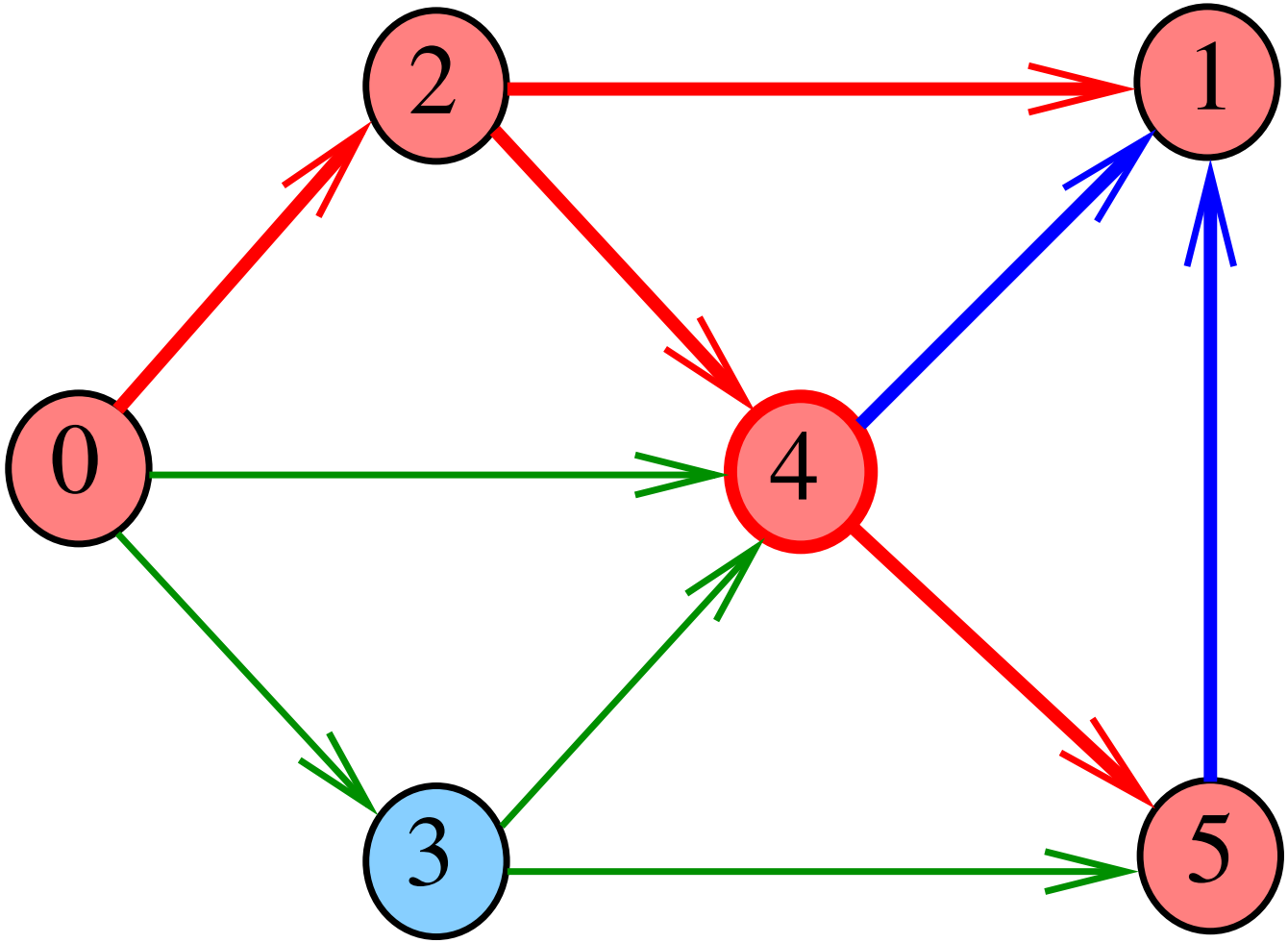
dfs(G, 5)



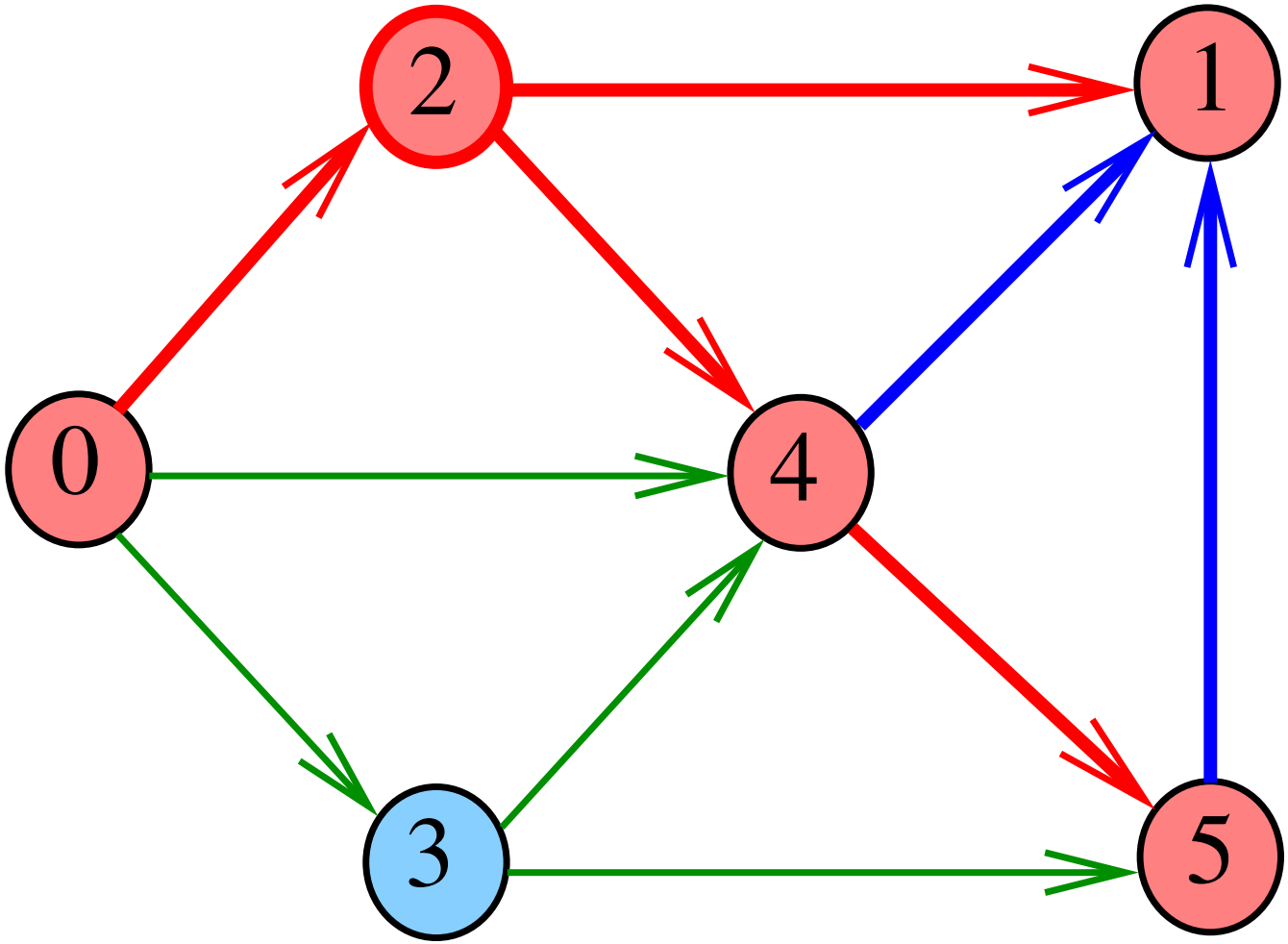
dfs(G, 5)



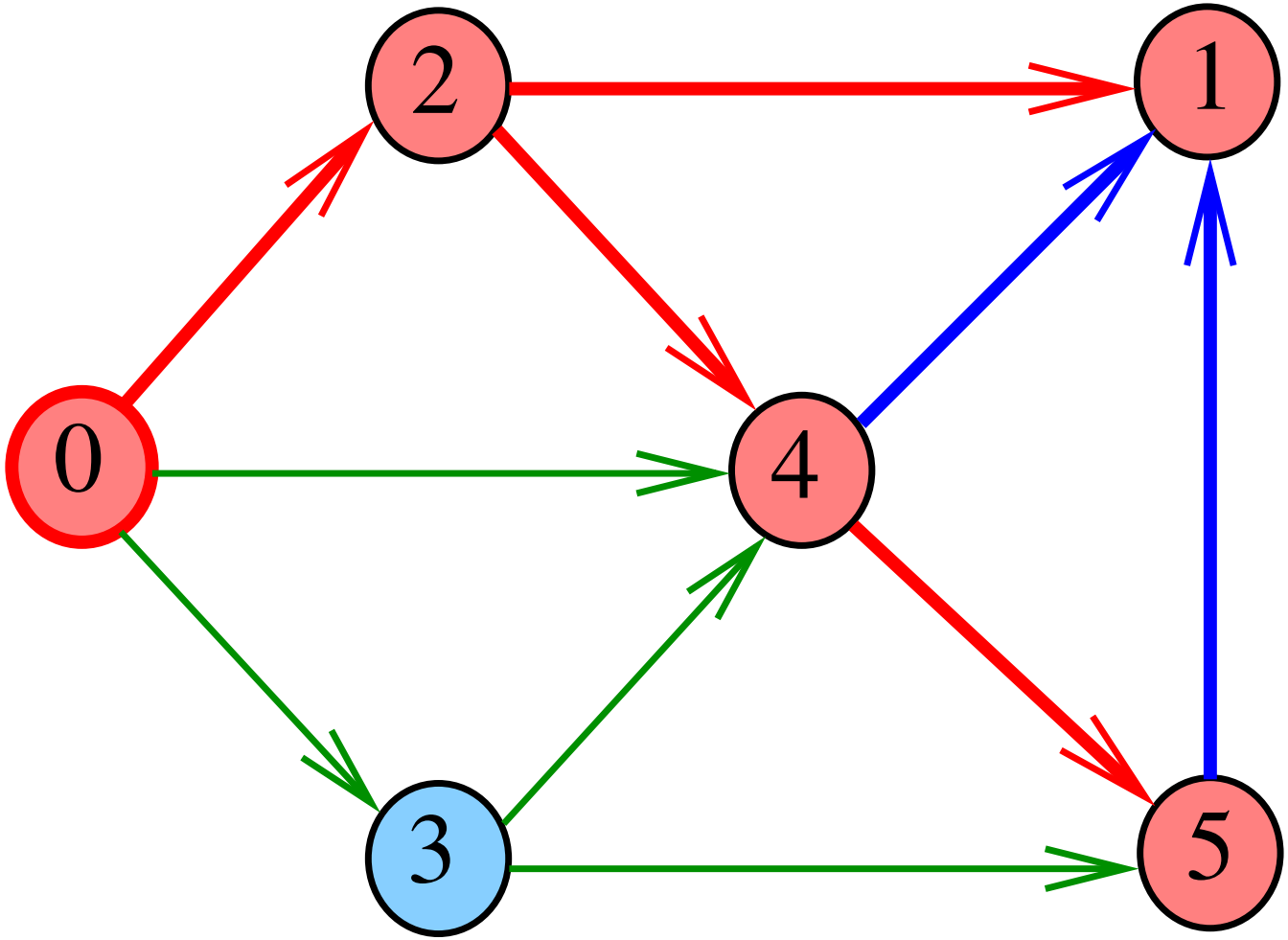
dfs(G, 4)



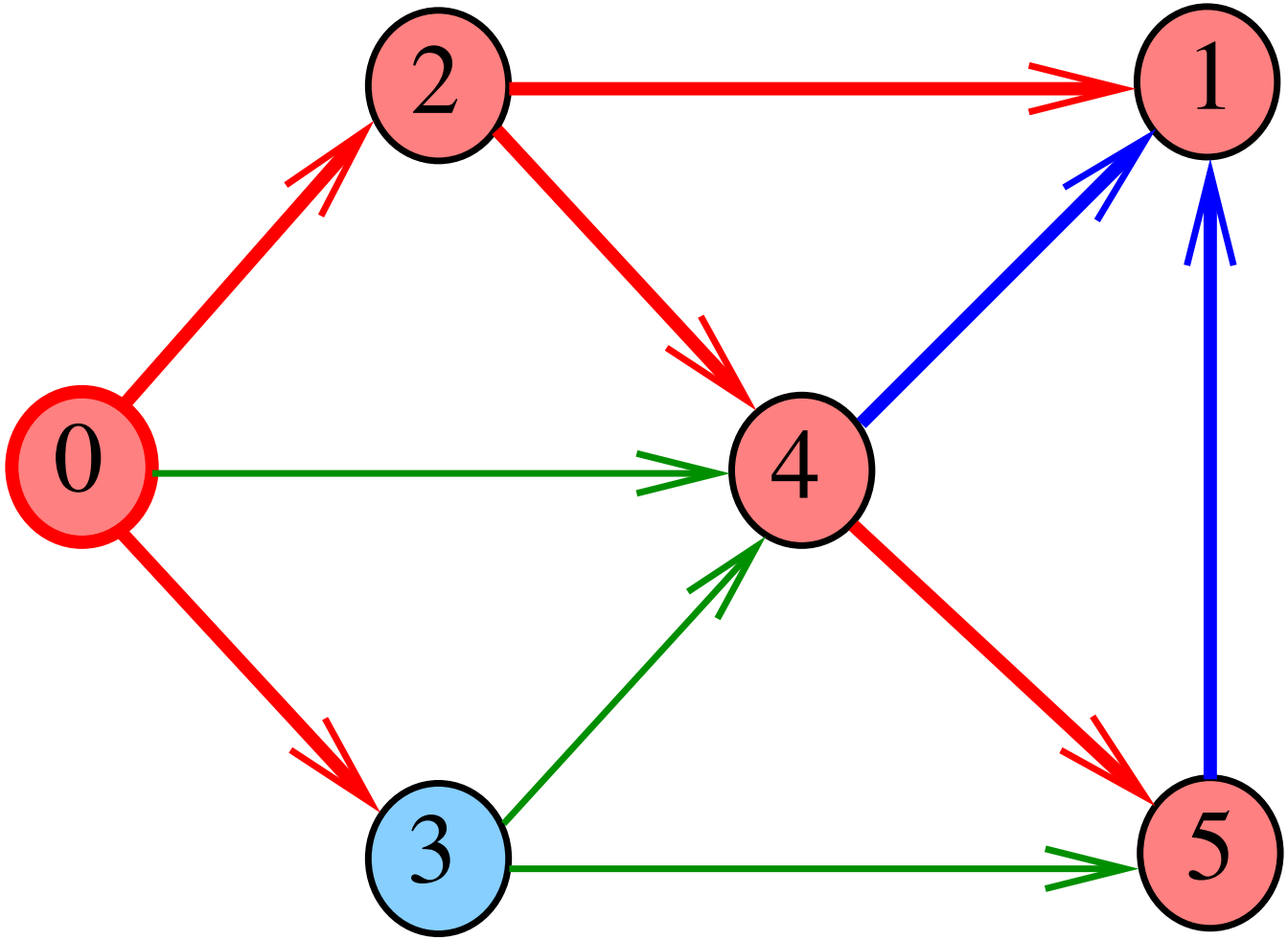
dfs(G, 2)



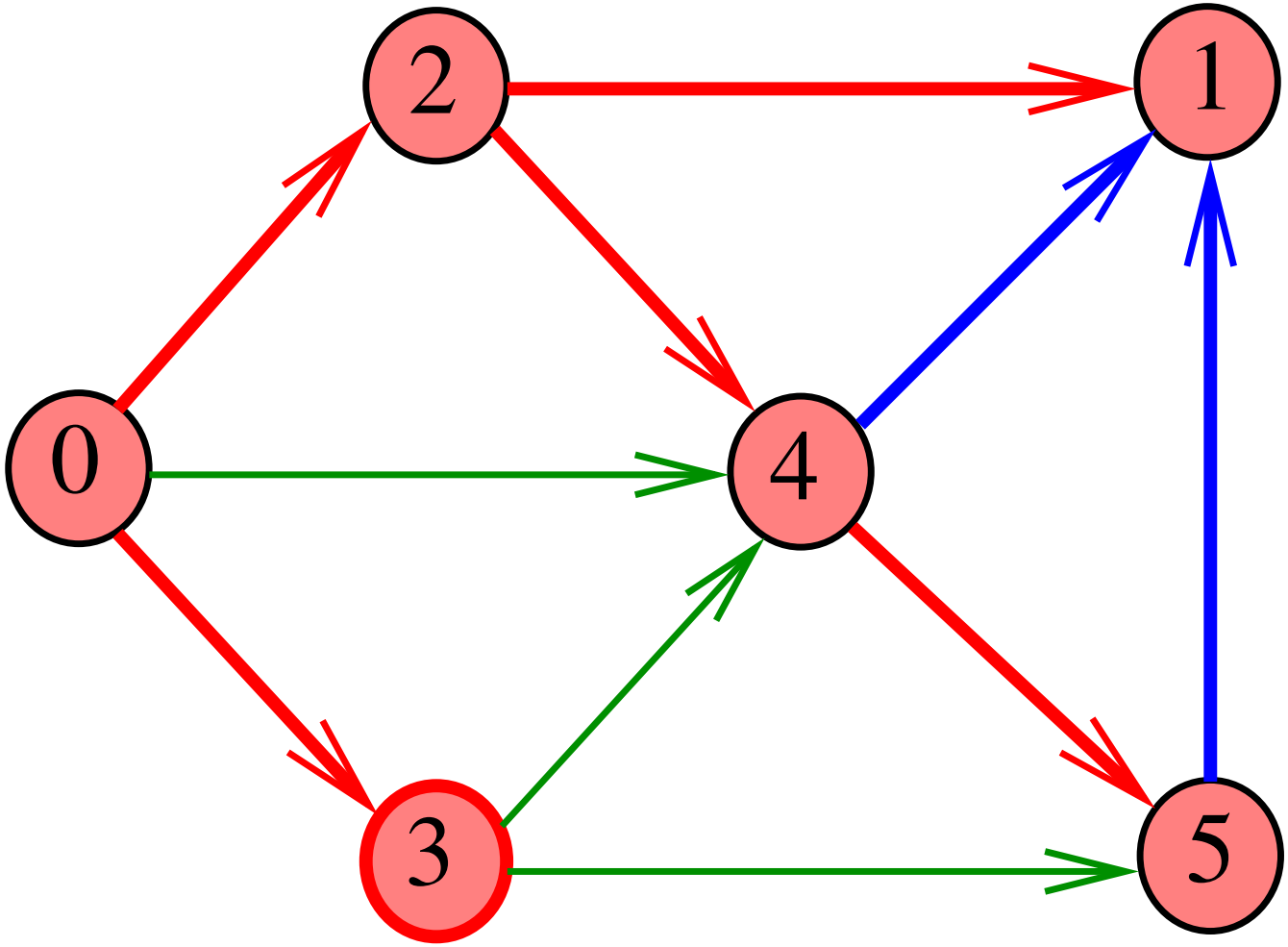
dfs(G, 0)



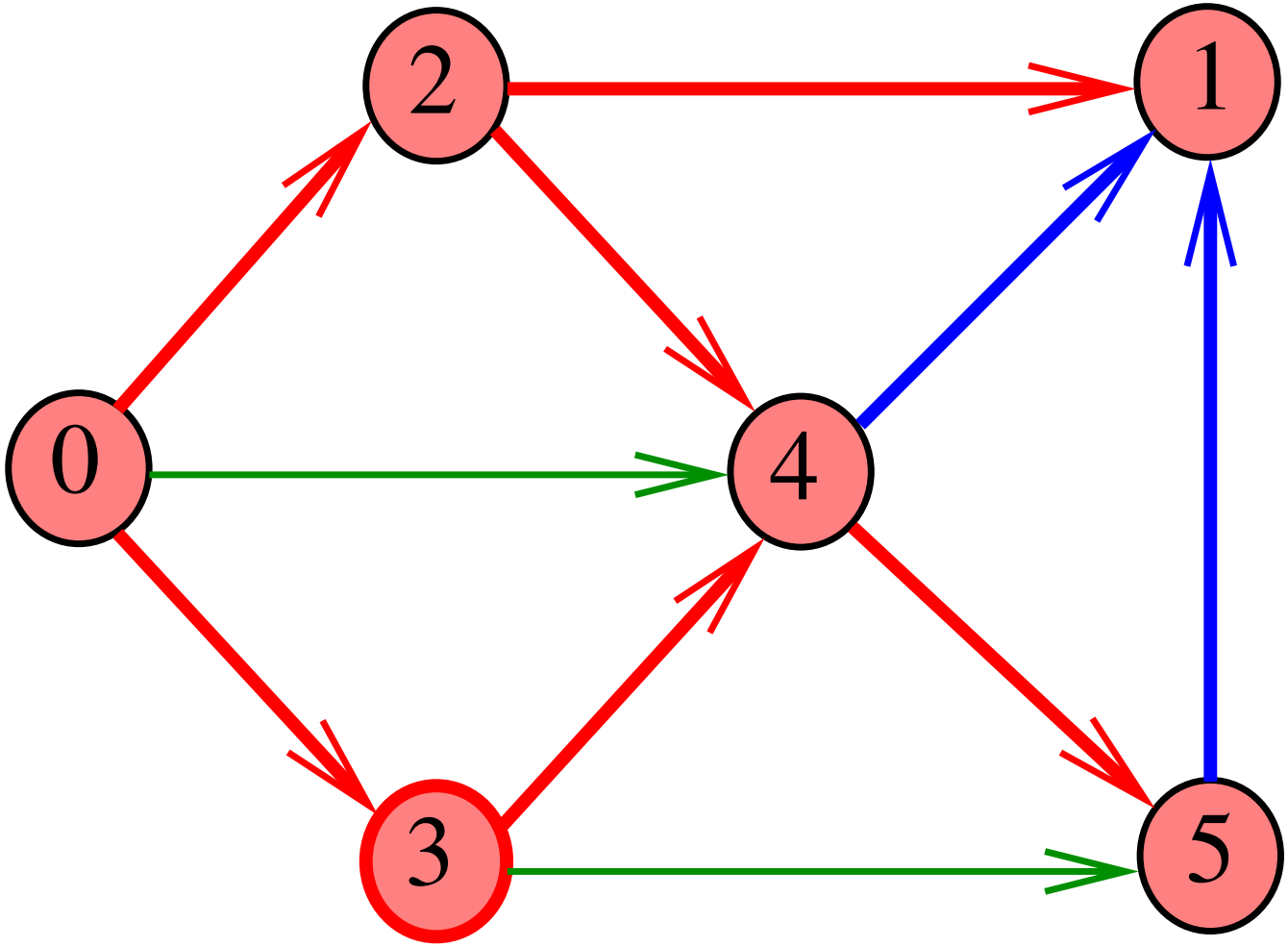
dfs(G, 0)



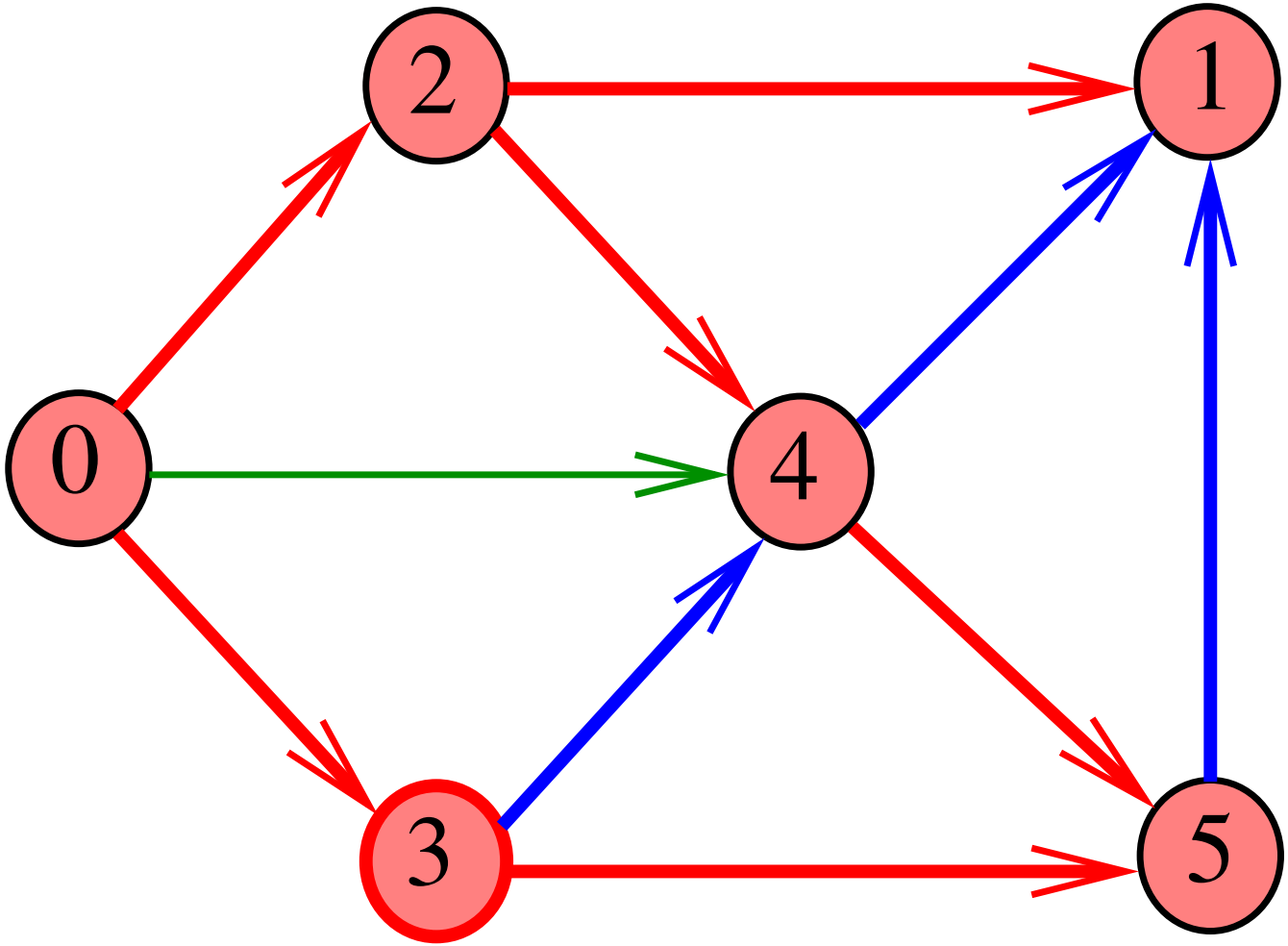
dfs(G, 3)



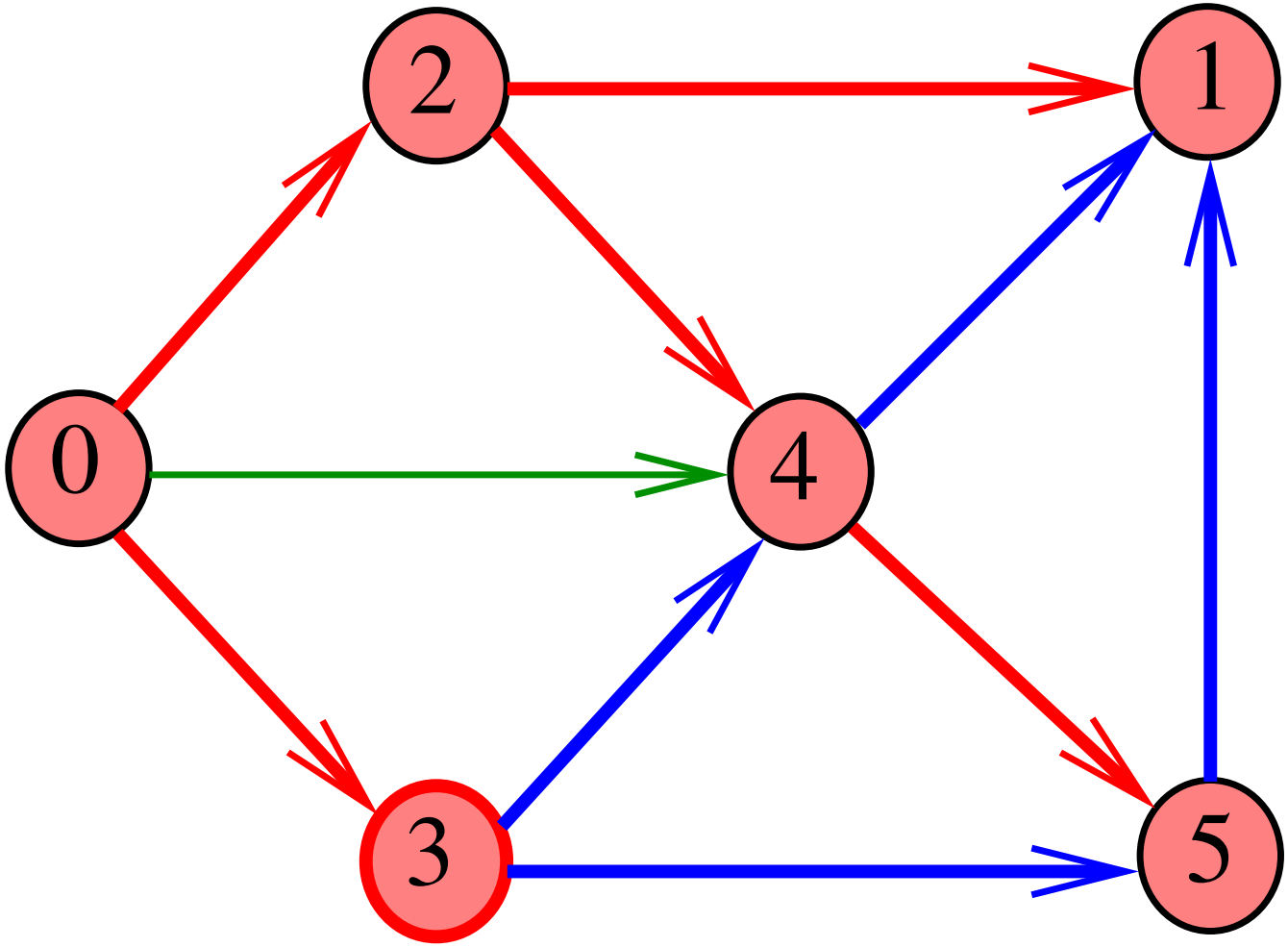
dfs(G, 3)



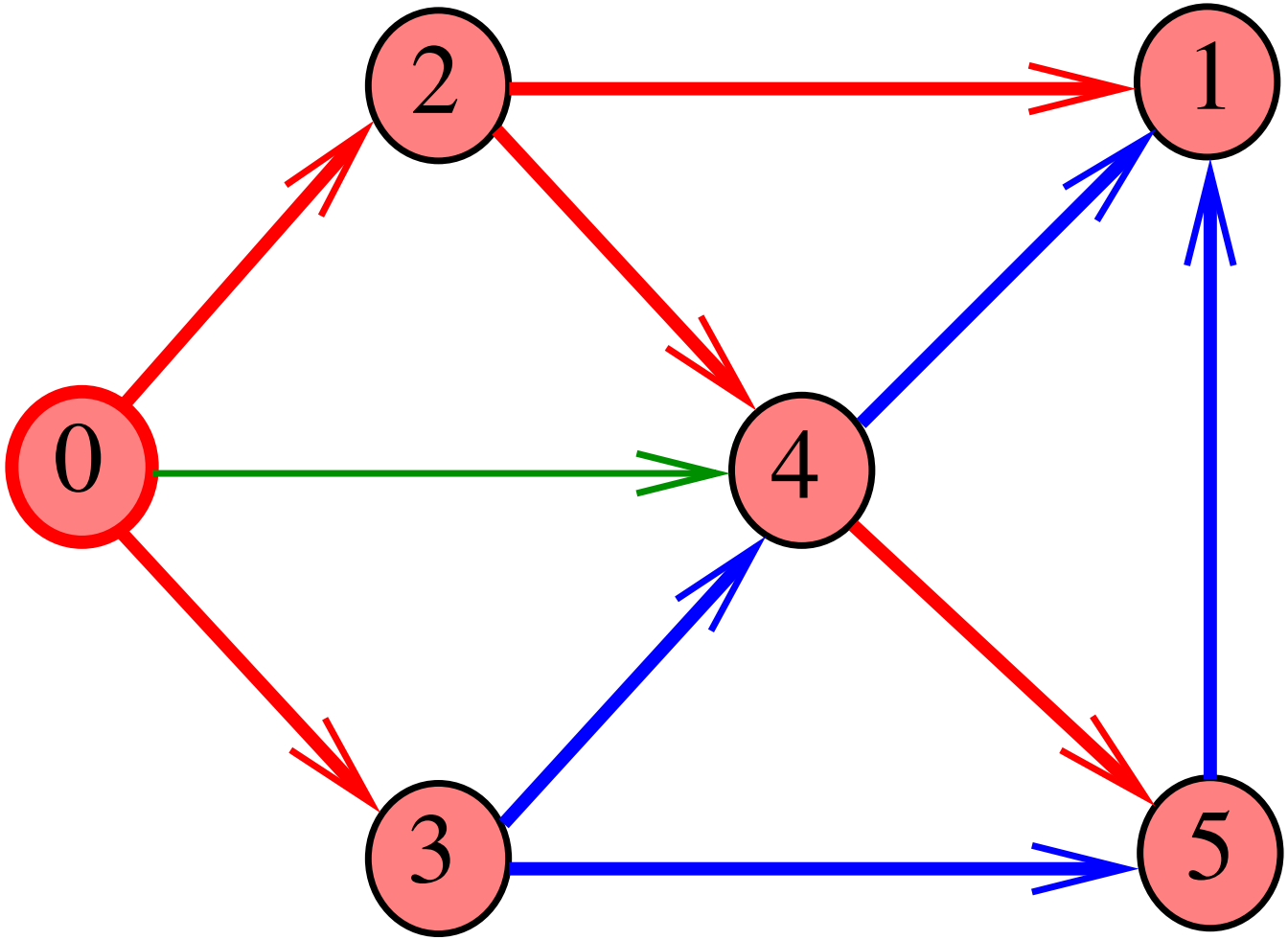
dfs(G, 3)



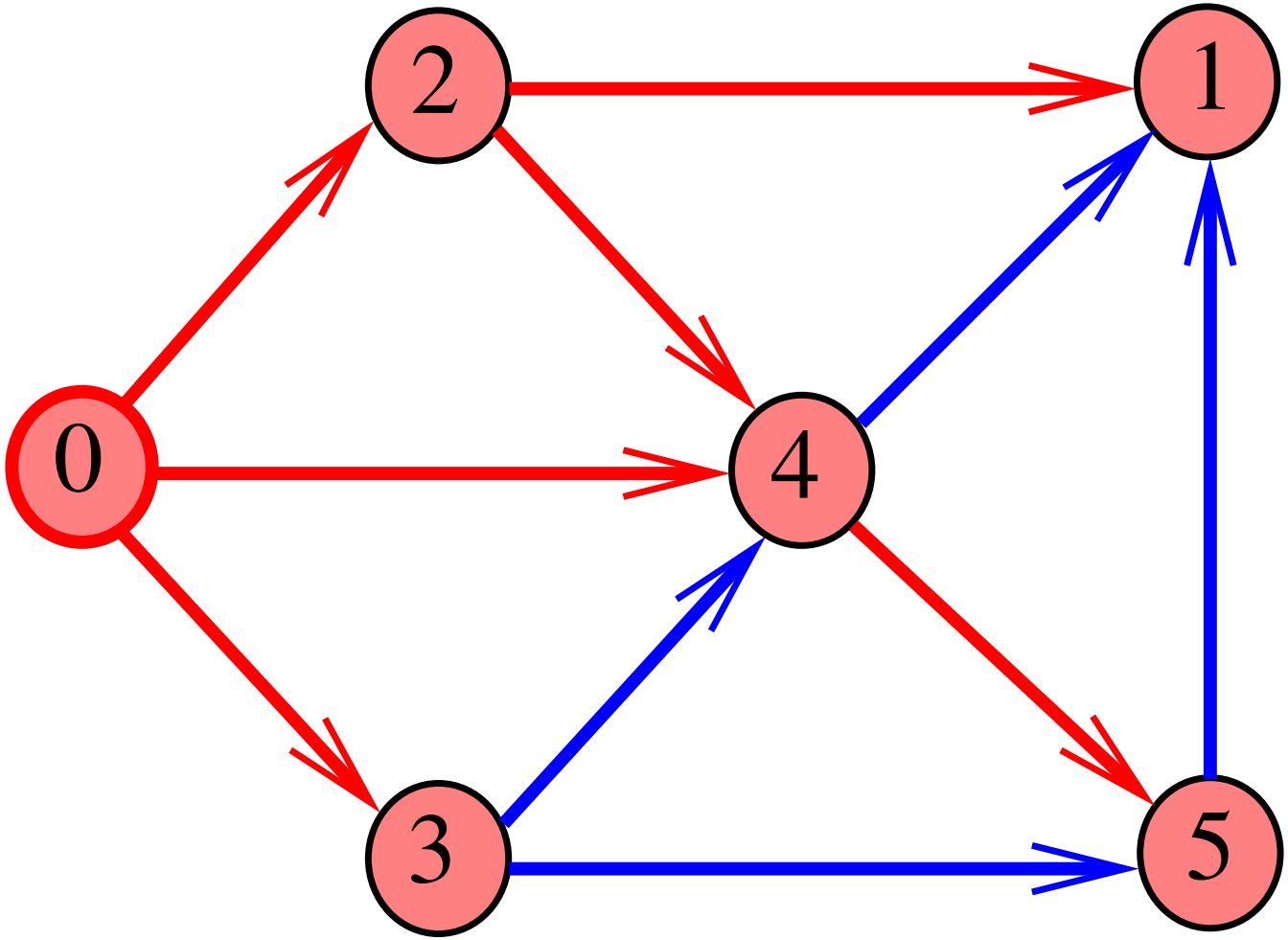
dfs(G, 3)



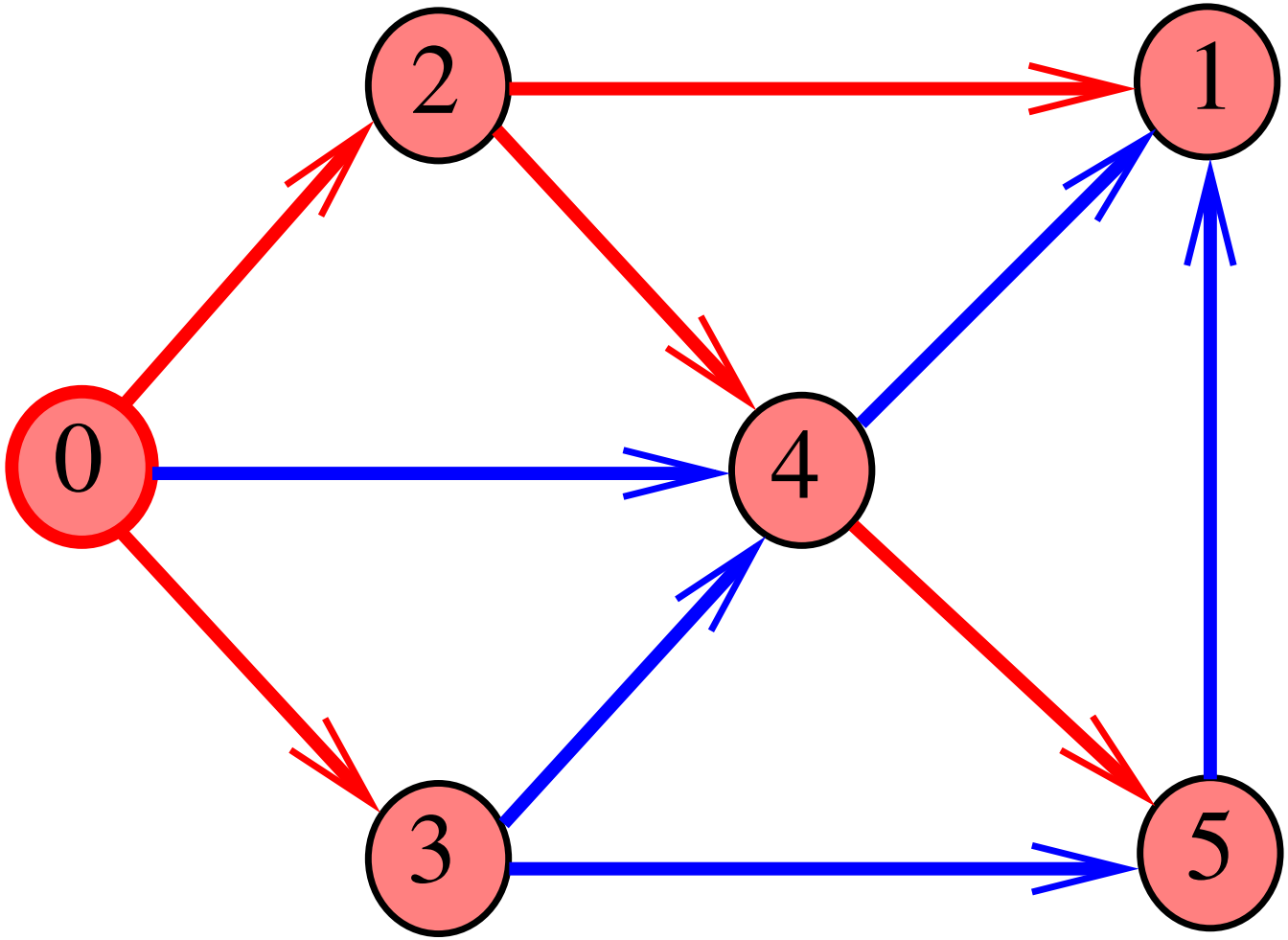
dfs(G, 0)



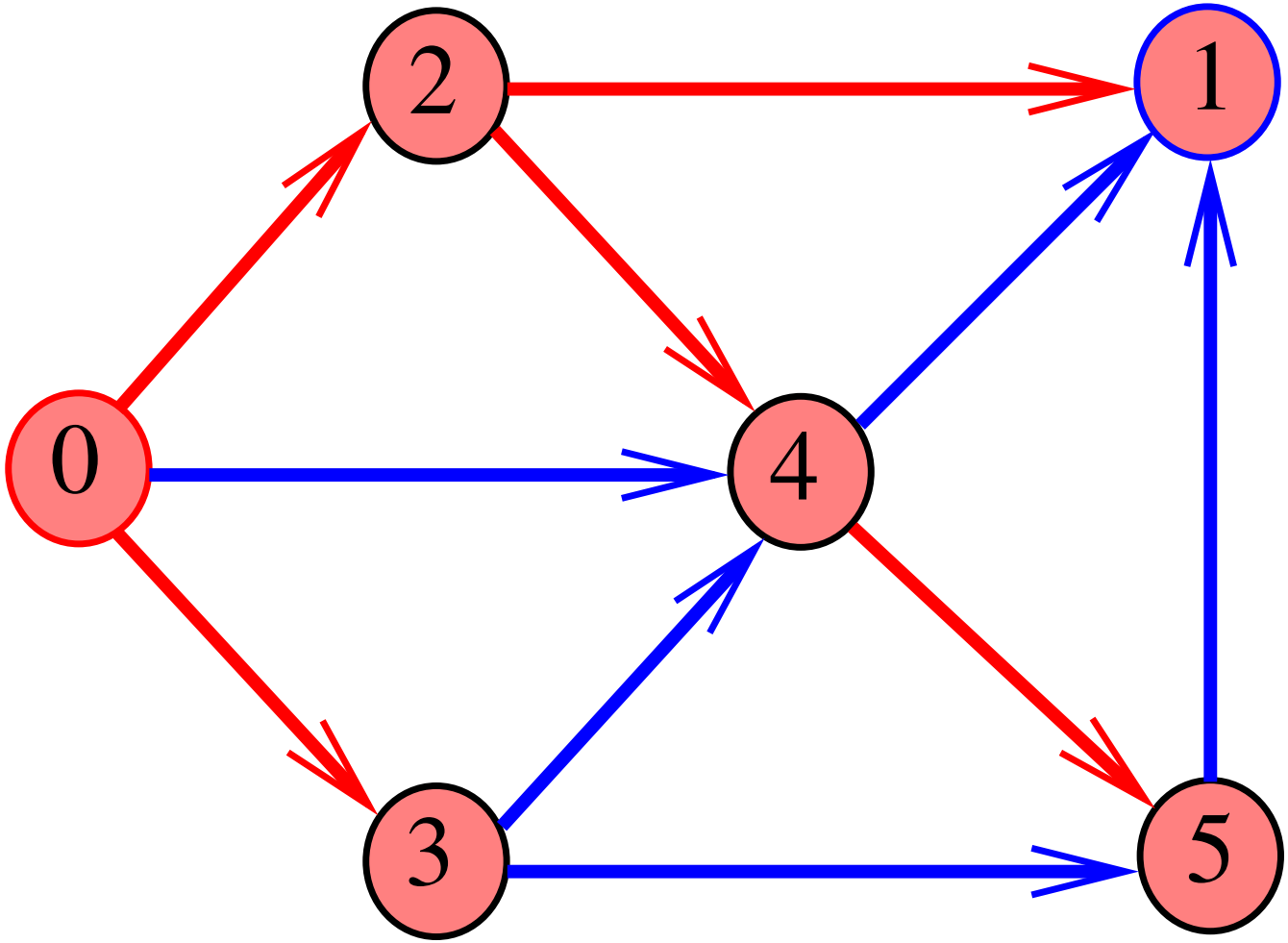
dfs(G, 0)



dfs(G, 0)

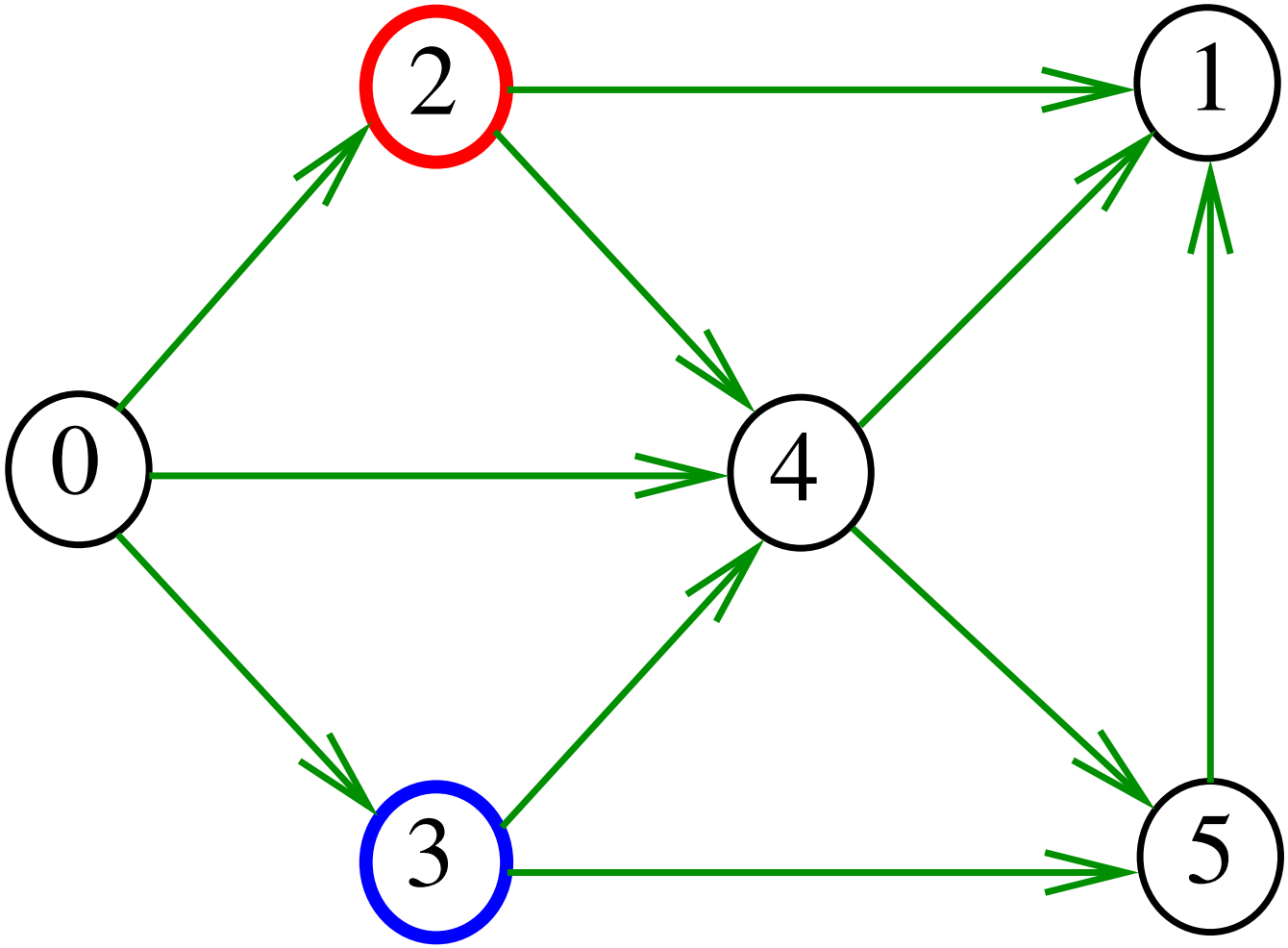


DFSpaths(G, 0)

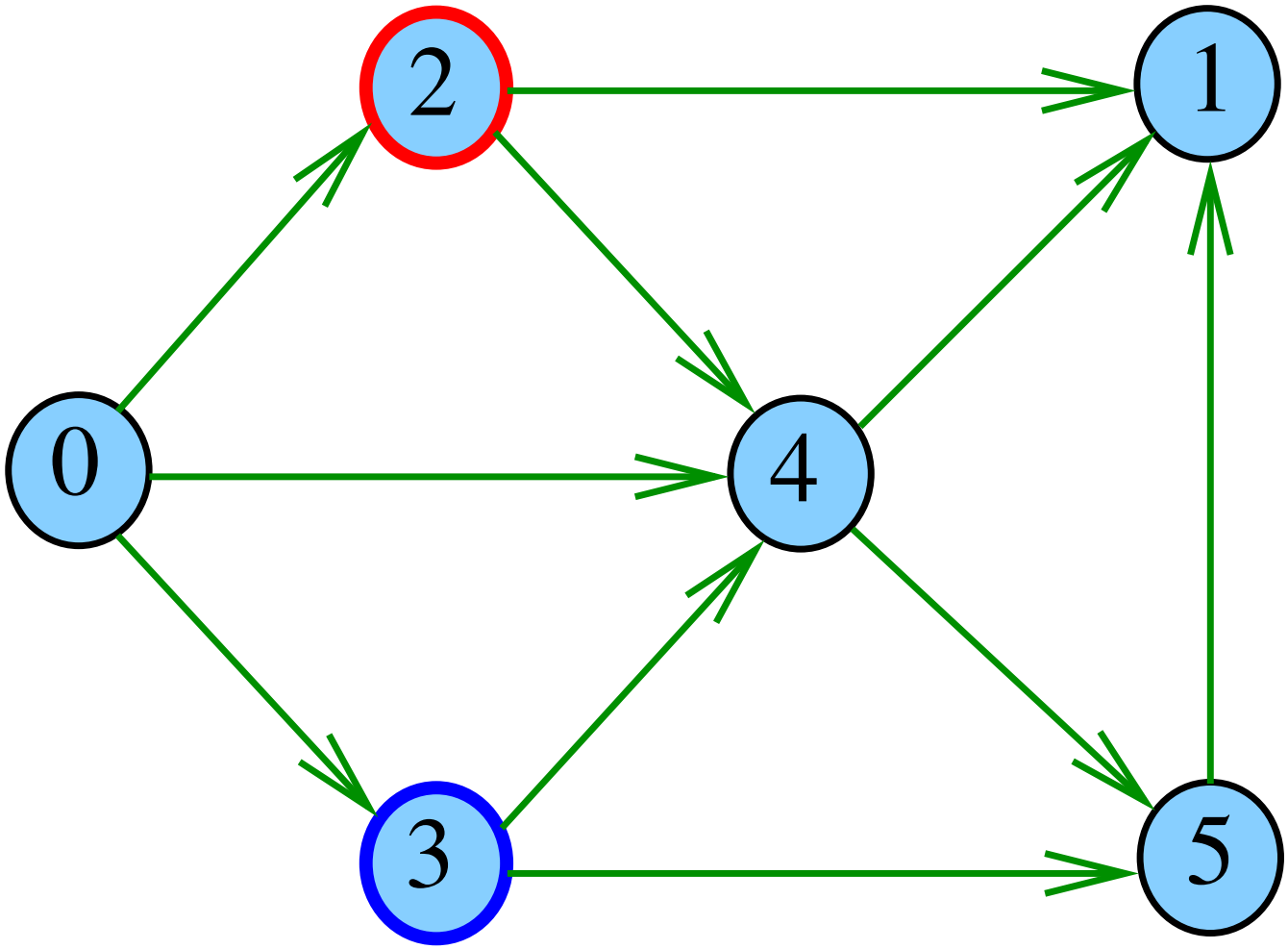


DFSpaths(G, 2)

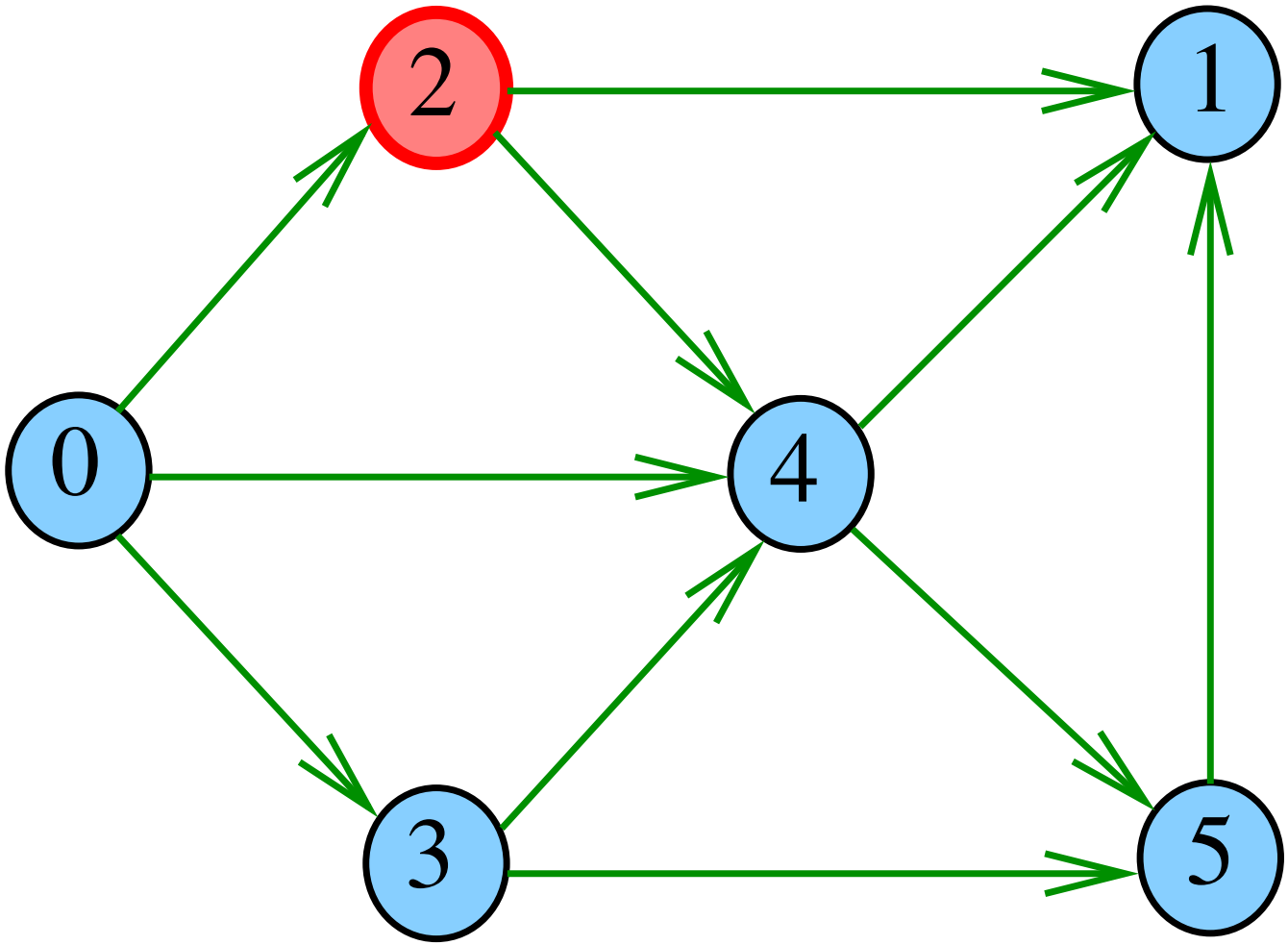
Existe caminho de 2 a 3?



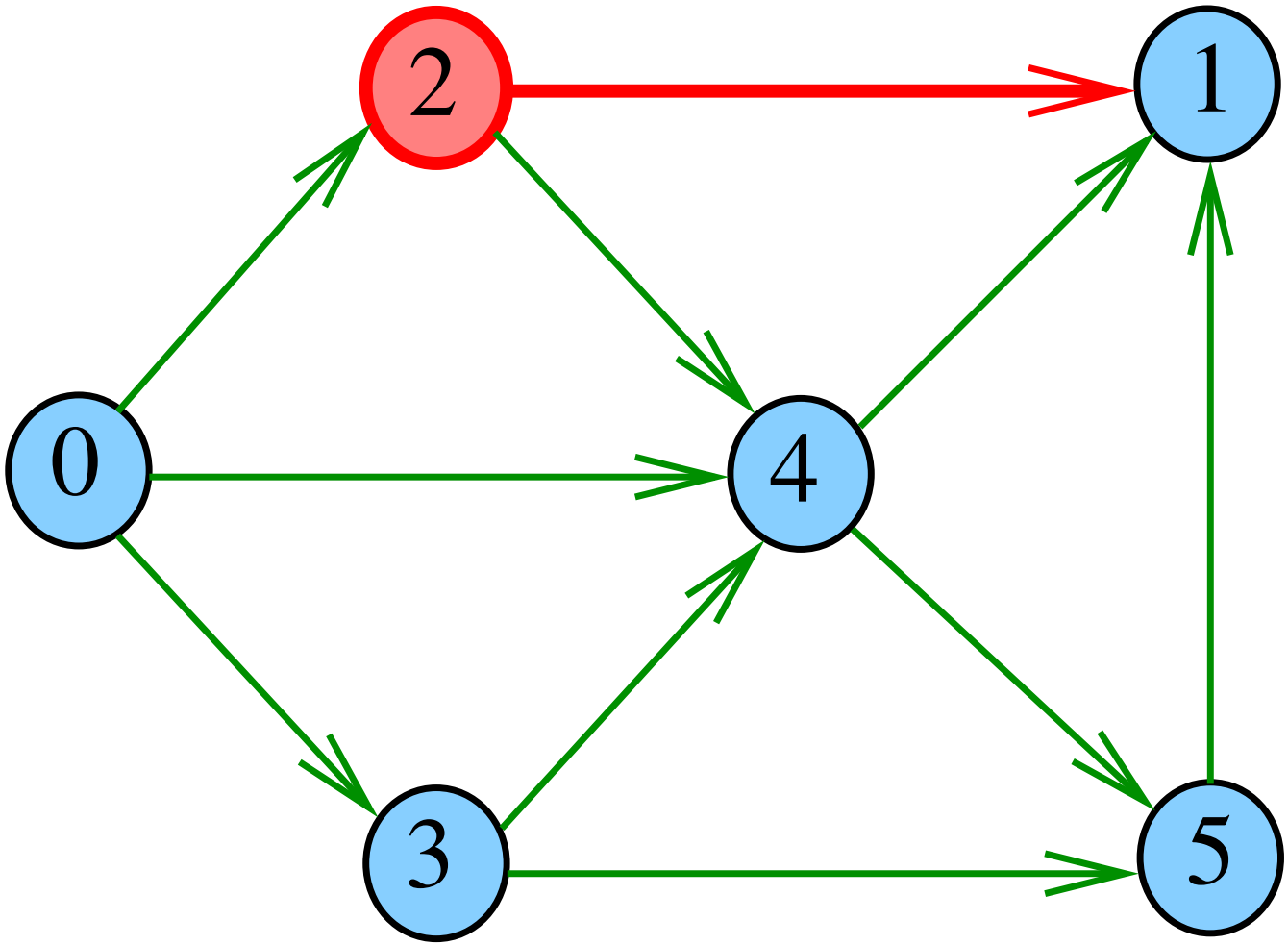
DFSpaths(G, 2)



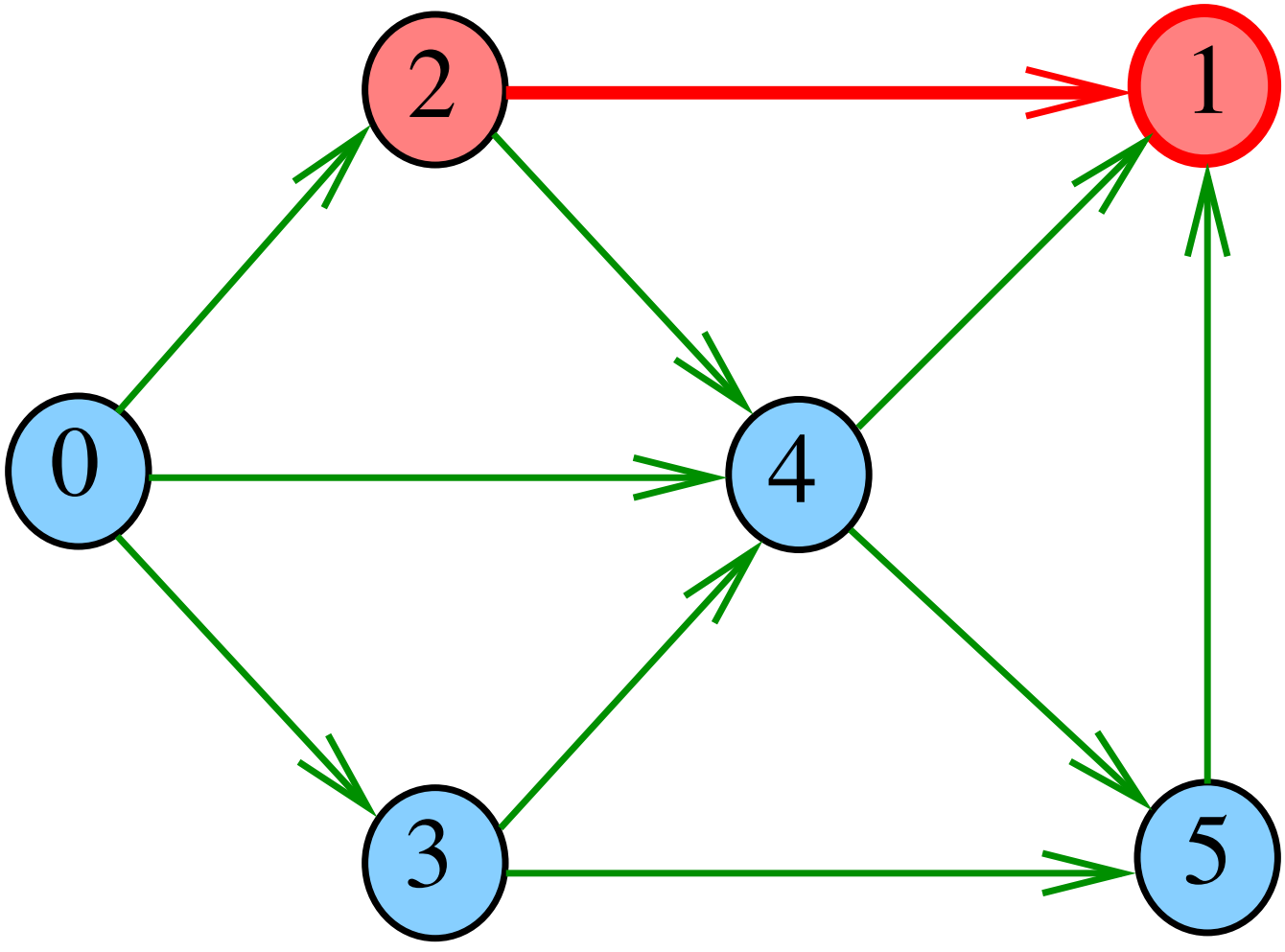
dfs(G, 2)



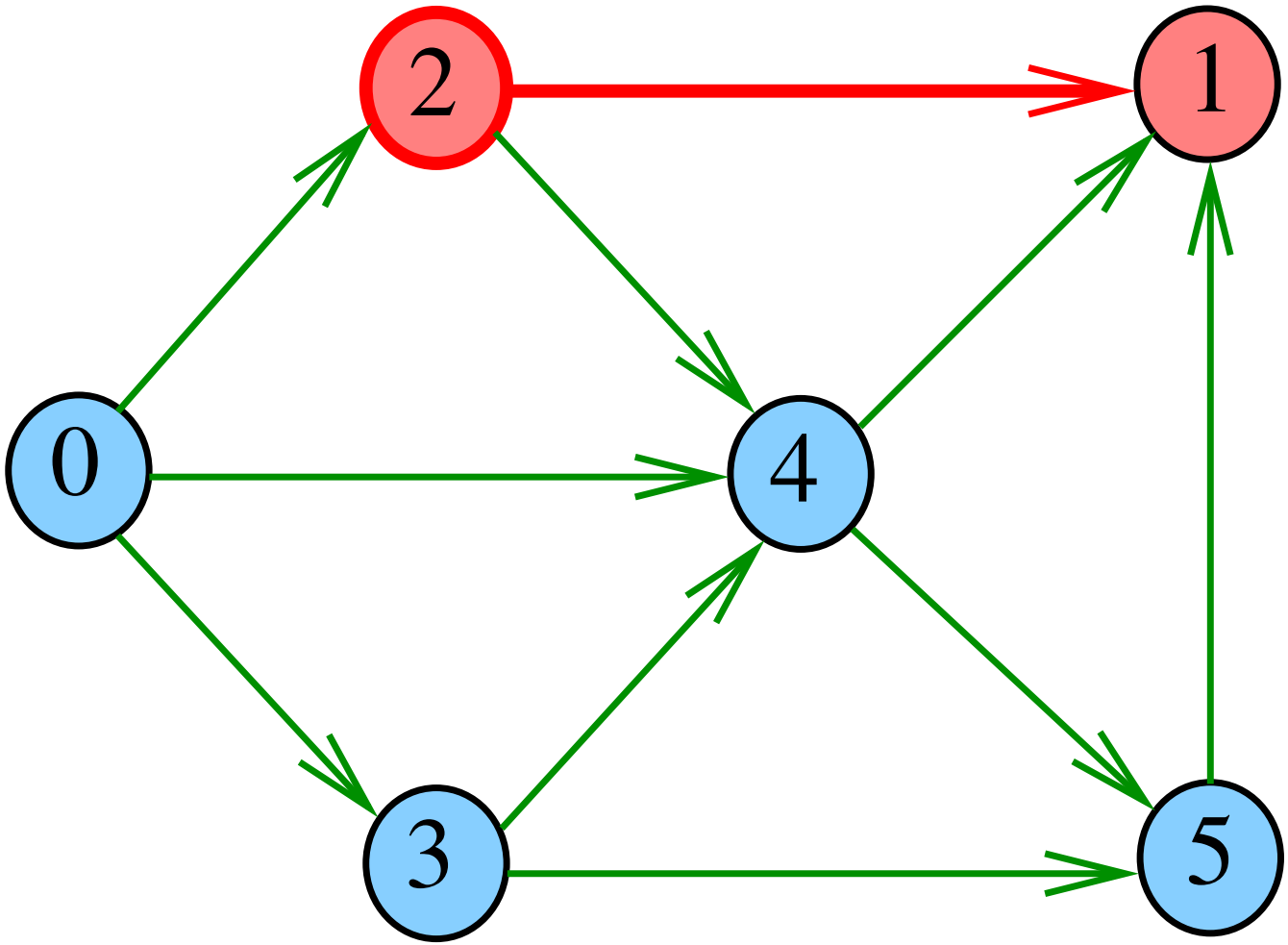
dfs(G, 2)



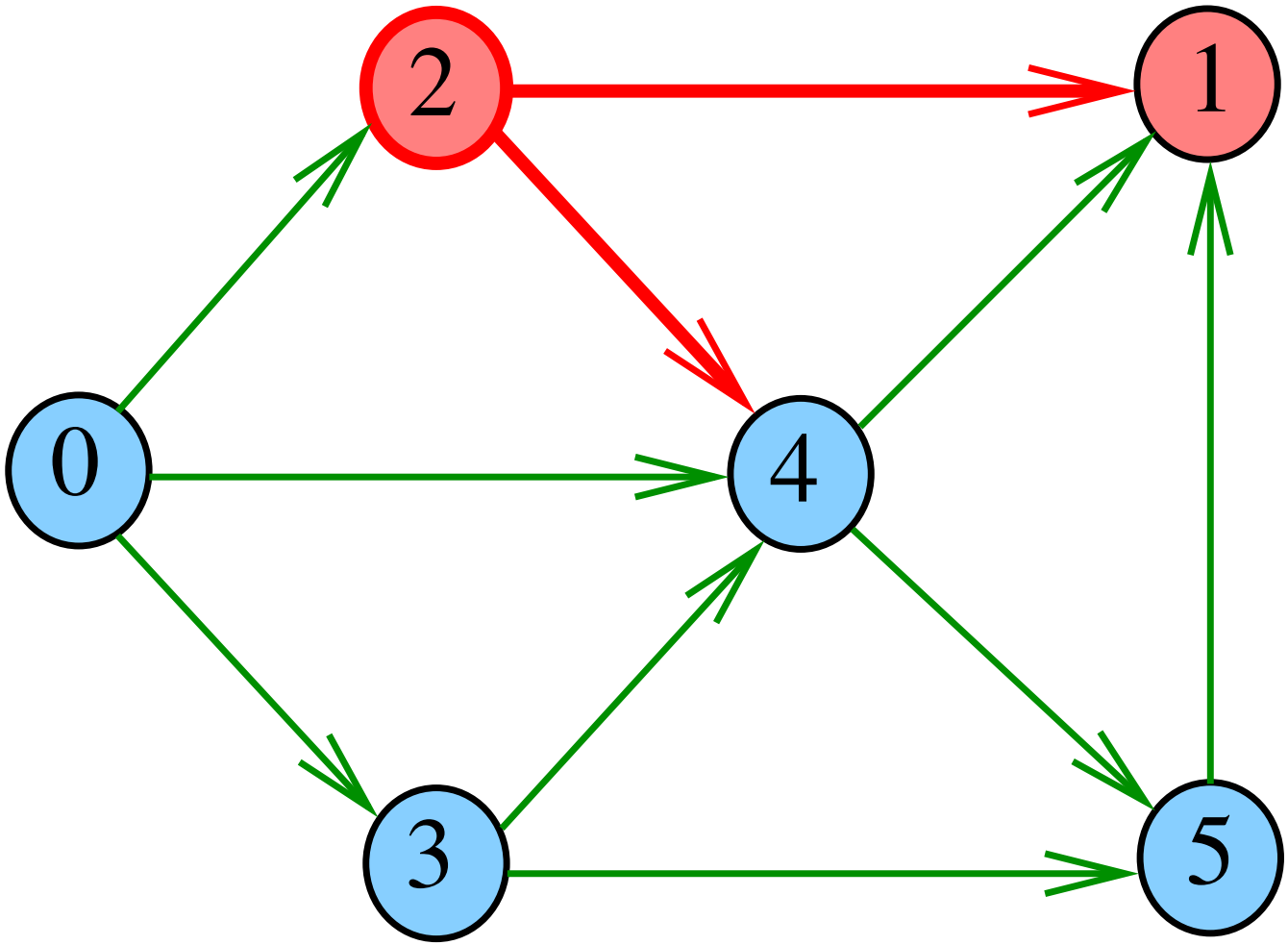
dfs(G, 1)



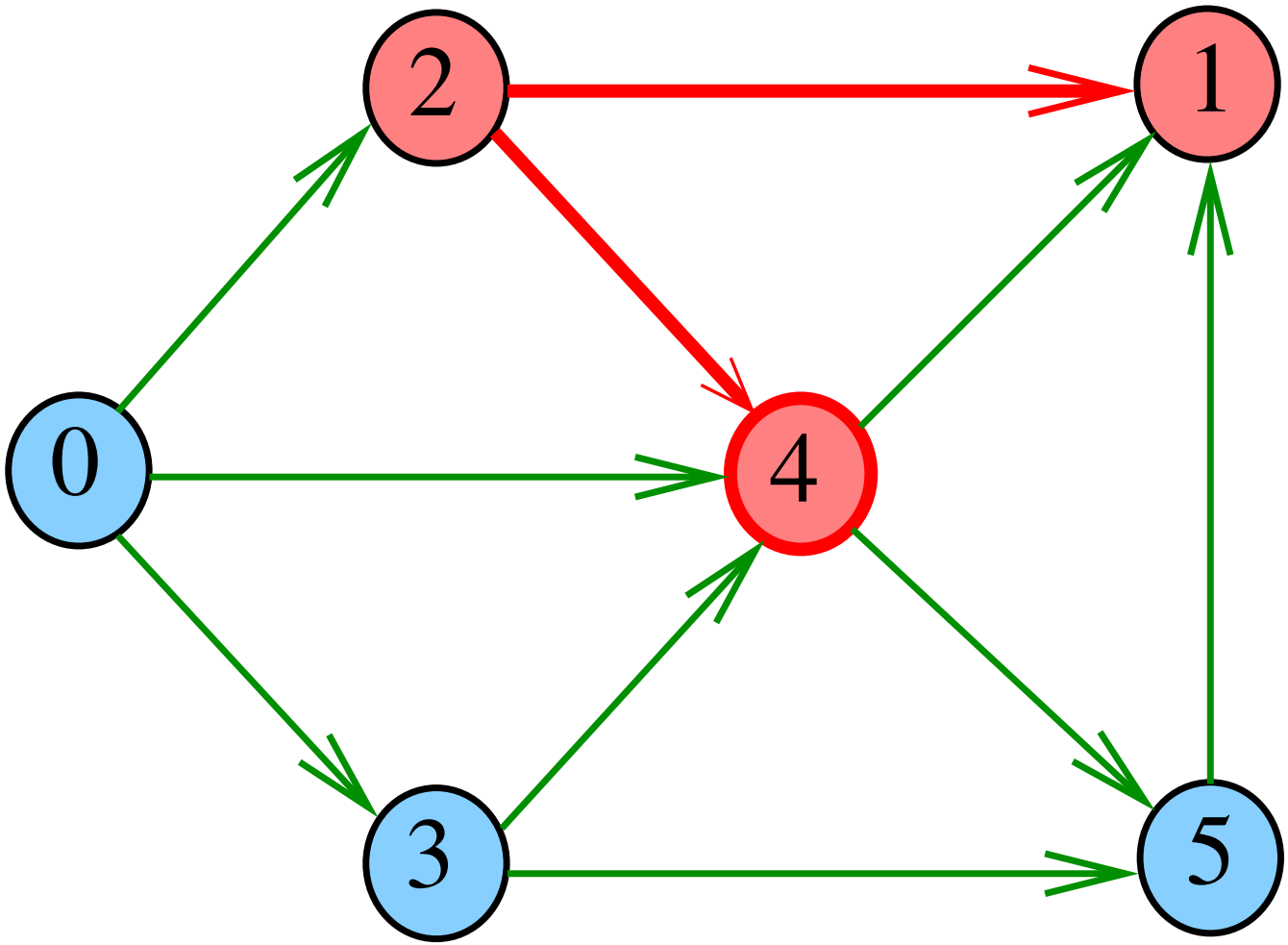
dfs(G, 2)



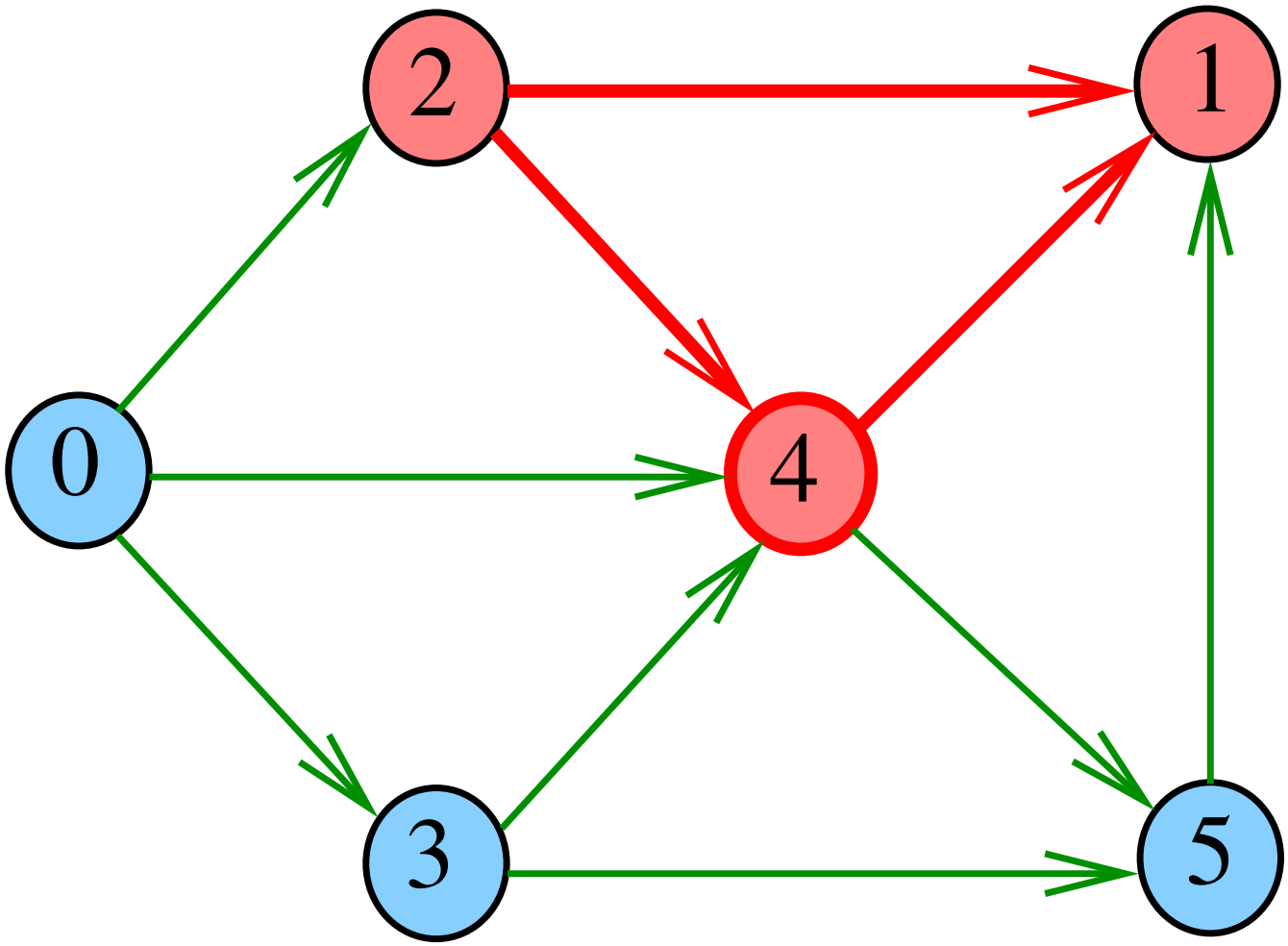
dfs(G, 2)



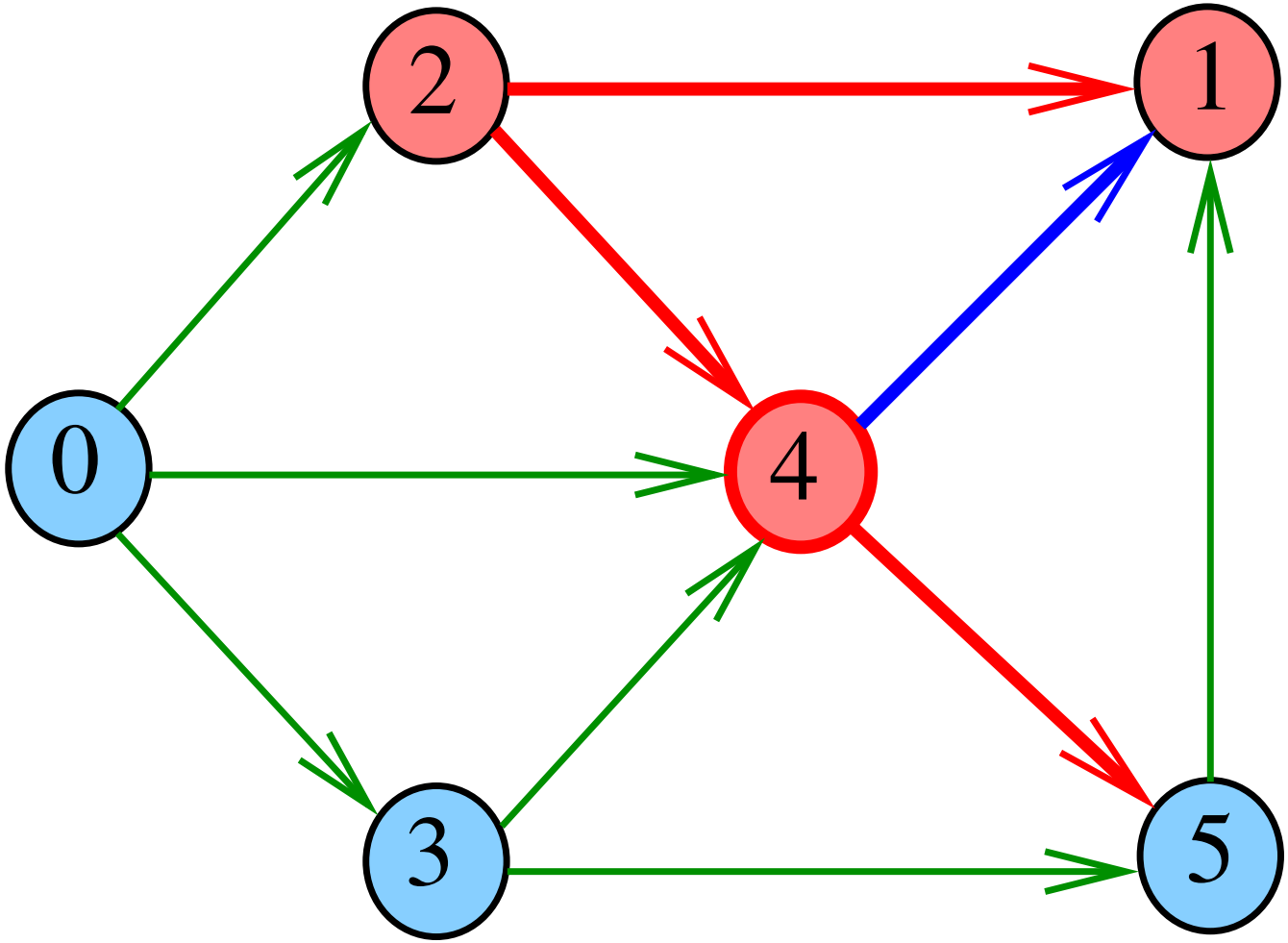
dfs(G, 4)



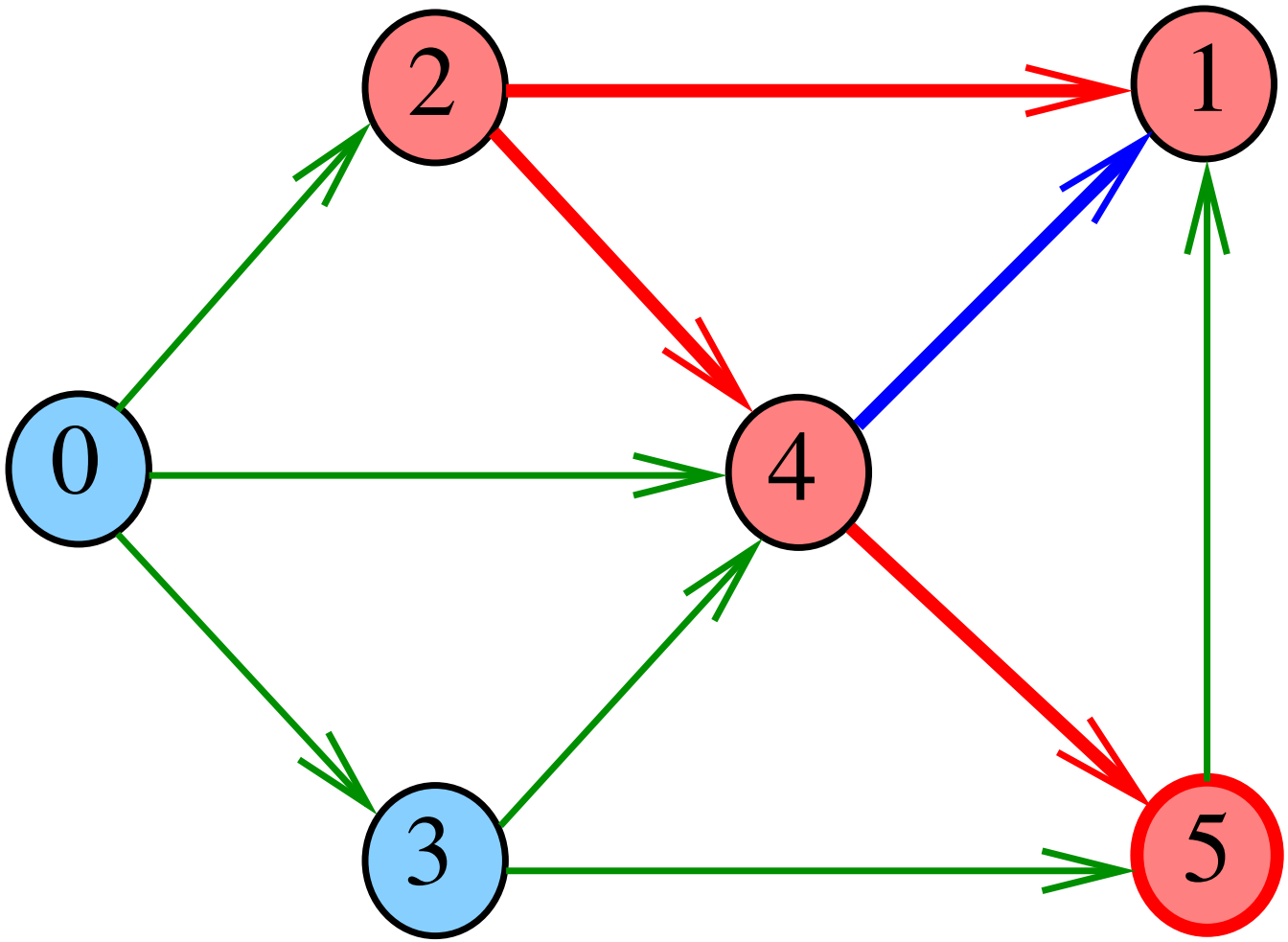
dfs(G, 4)



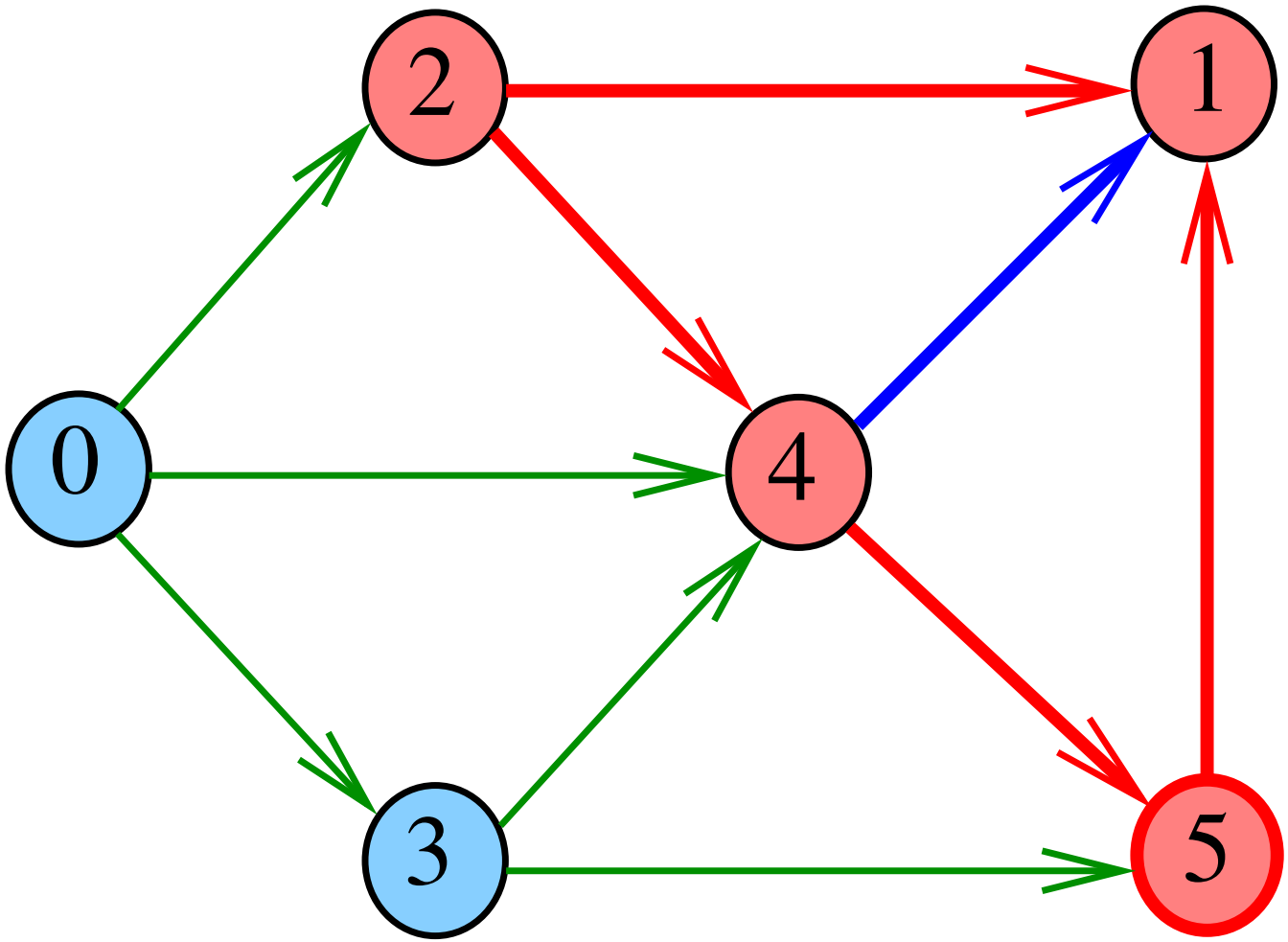
dfs(G, 4)



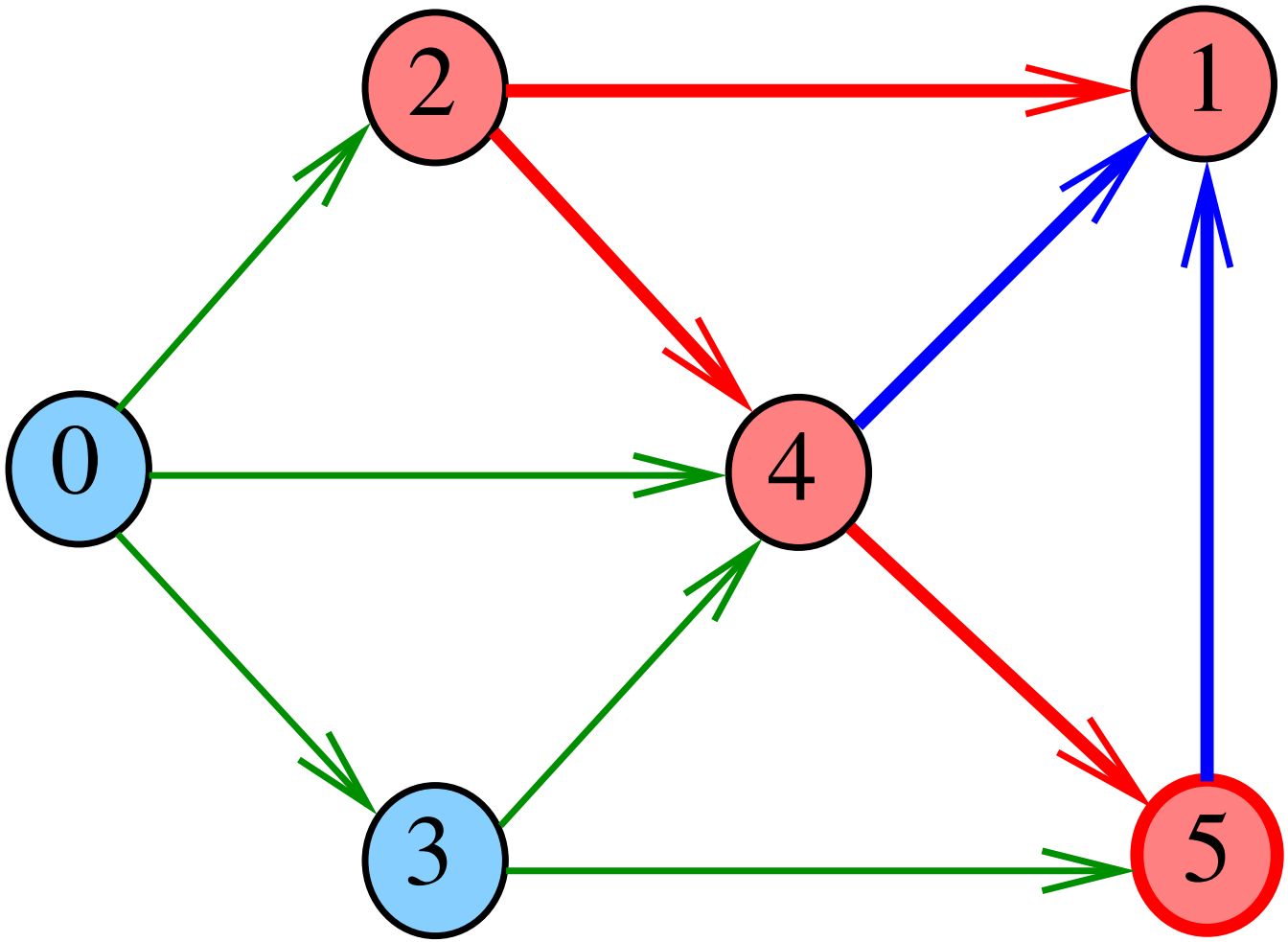
dfs(G, 5)



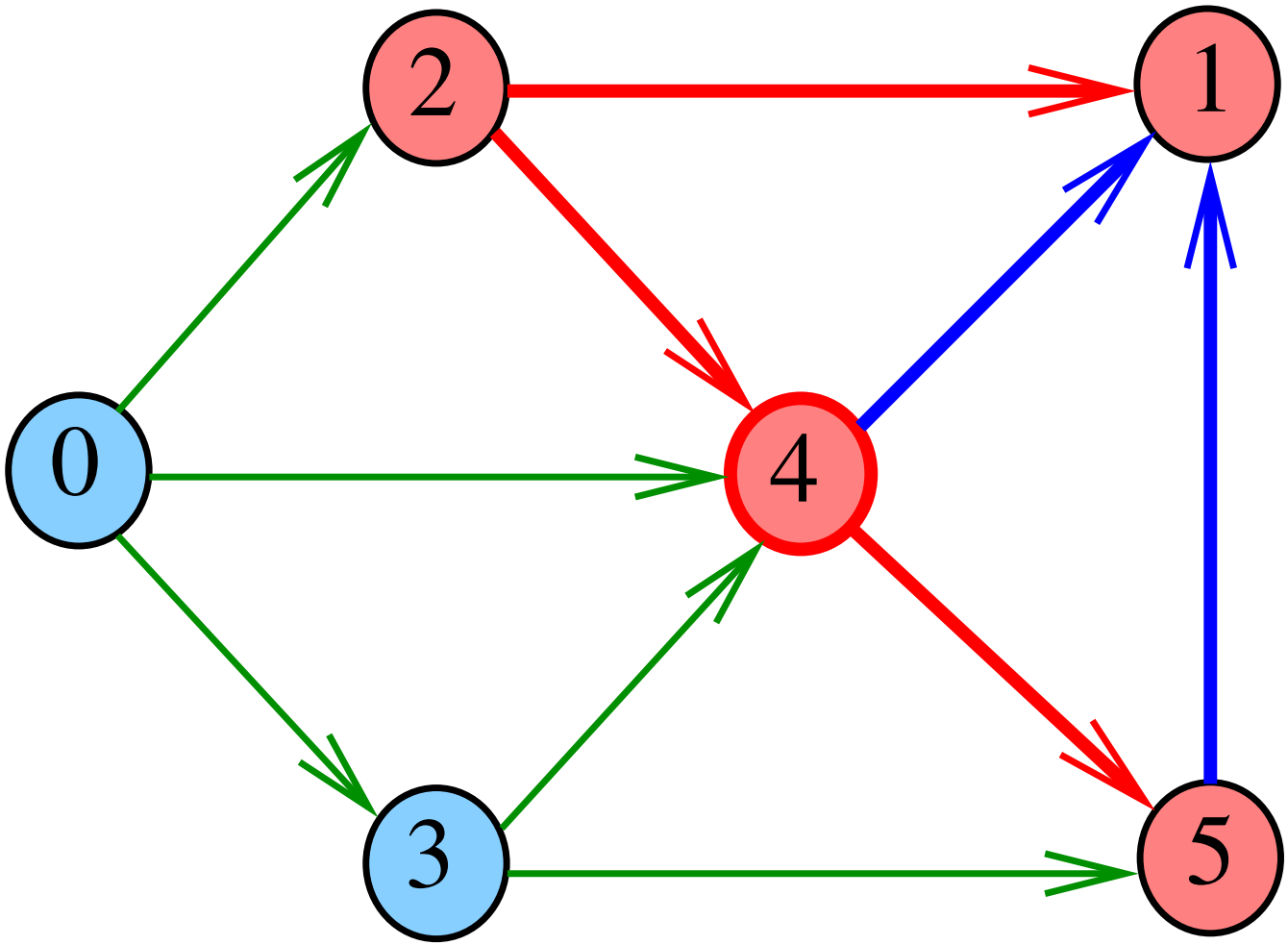
dfs(G, 5)



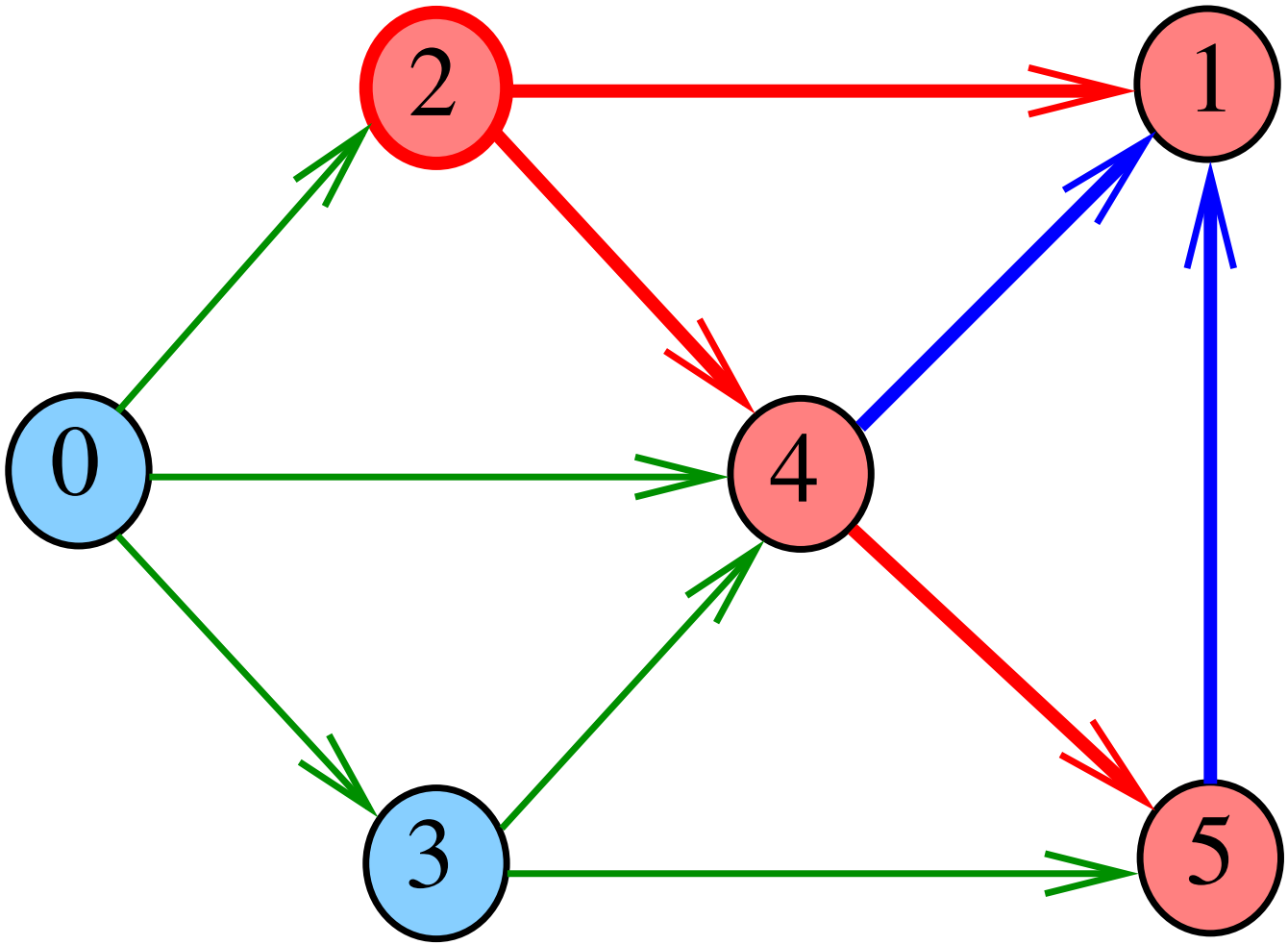
dfs(G, 5)



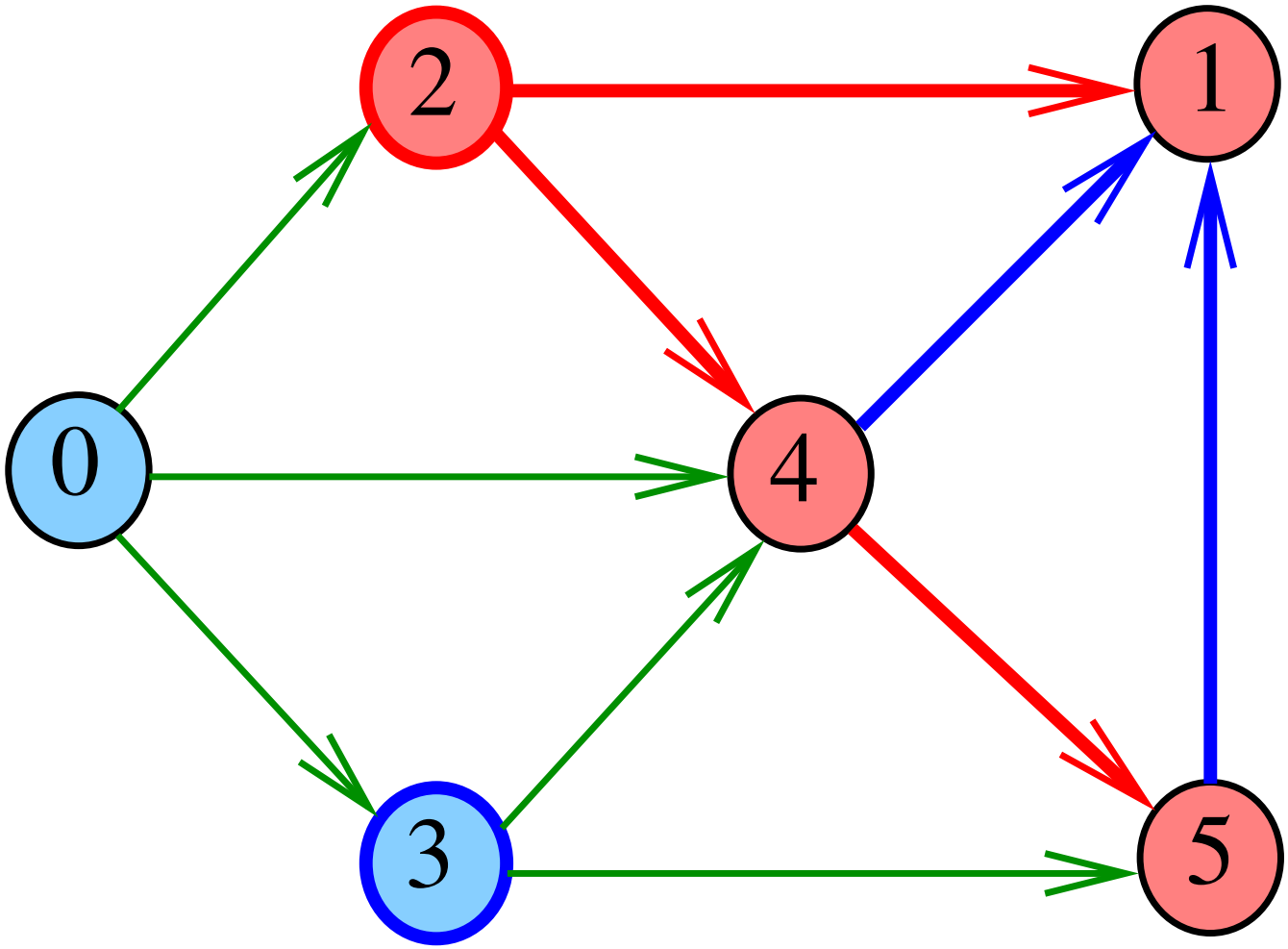
dfs(G, 4)



dfs(G, 2)



DFSpaths(G, 2)



DFSpaths: estrutura

```
static struct dfspaths {
    bool *marked;
    int  *edgeTo;
};
typedef struct dsfpaths *dfsPaths;
```

DFSpaths: Init

```
static dfsPaths DFSpathsInit(Digraph G) {
   dfsPaths T = mallocSafe(sizeof(*T));
   T->marked = mallocSafe(G->V*sizeof(bool));
   T->edgeTo = mallocSafe(G->V*sizeof(int));
   for (int v = 0; v < G->V; v++) {
      T->marked[v] = false;
      T->edgeTo[v] = -1;
   }
   return T;
}
```

DFSpaths: esqueleto

```
dfsPaths DFSpaths(Digraph G, int s) {...}

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

/* Métodos copiados de BFSpaths. */
bool hasPath(dfsPaths T, int v) {...}

/* pilha com o caminho requisitado */
Stack pathTo(dfsPaths T, int v) {...}
```

DFSpaths

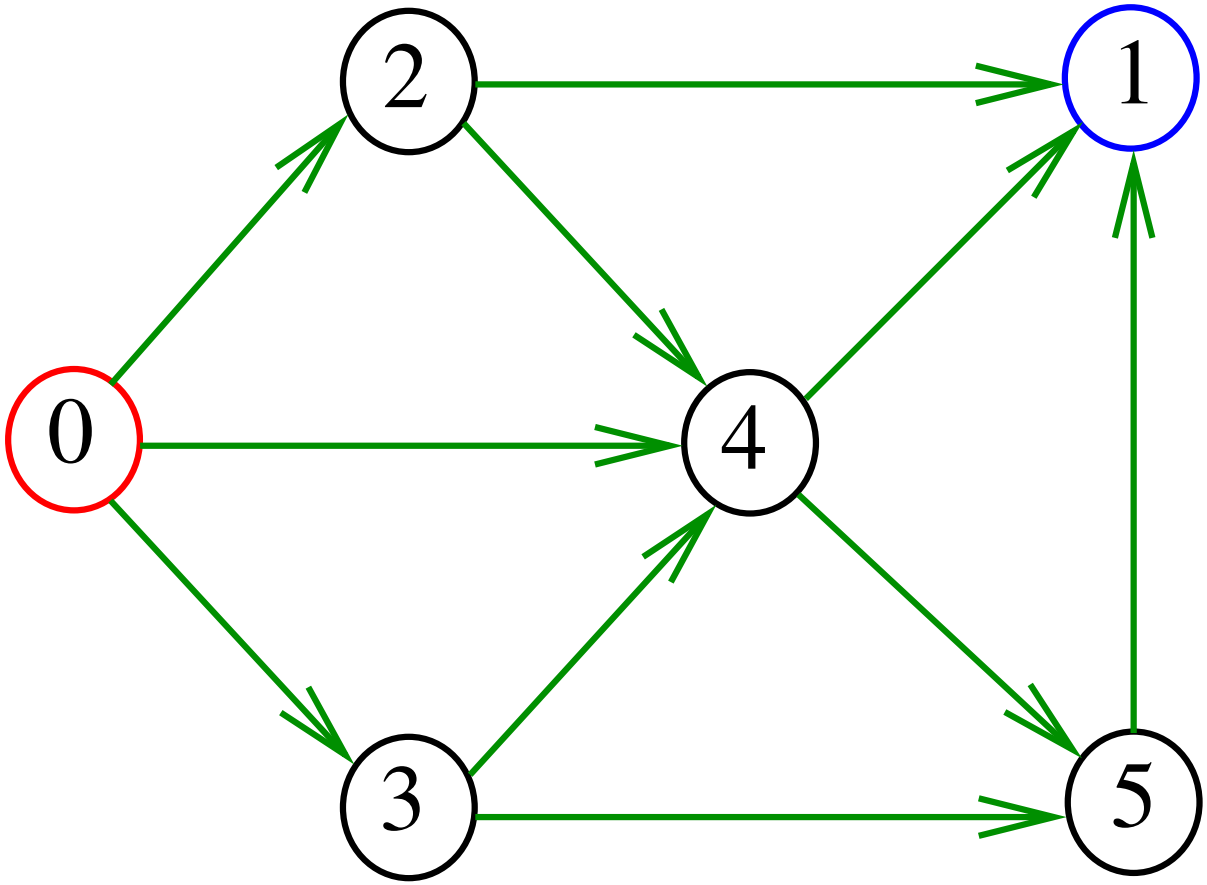
Encontra um caminho de `s` a todo vértice alcançável a partir de `s`.

```
dfsPaths DFSpaths(Digraph G, int s) {
   dfsPaths T = DFSpathsInit(G);
   T->edgeTo[s] = s;
   dfs(G, s, T);
   return T;
}
```

DFSpaths: dfs()

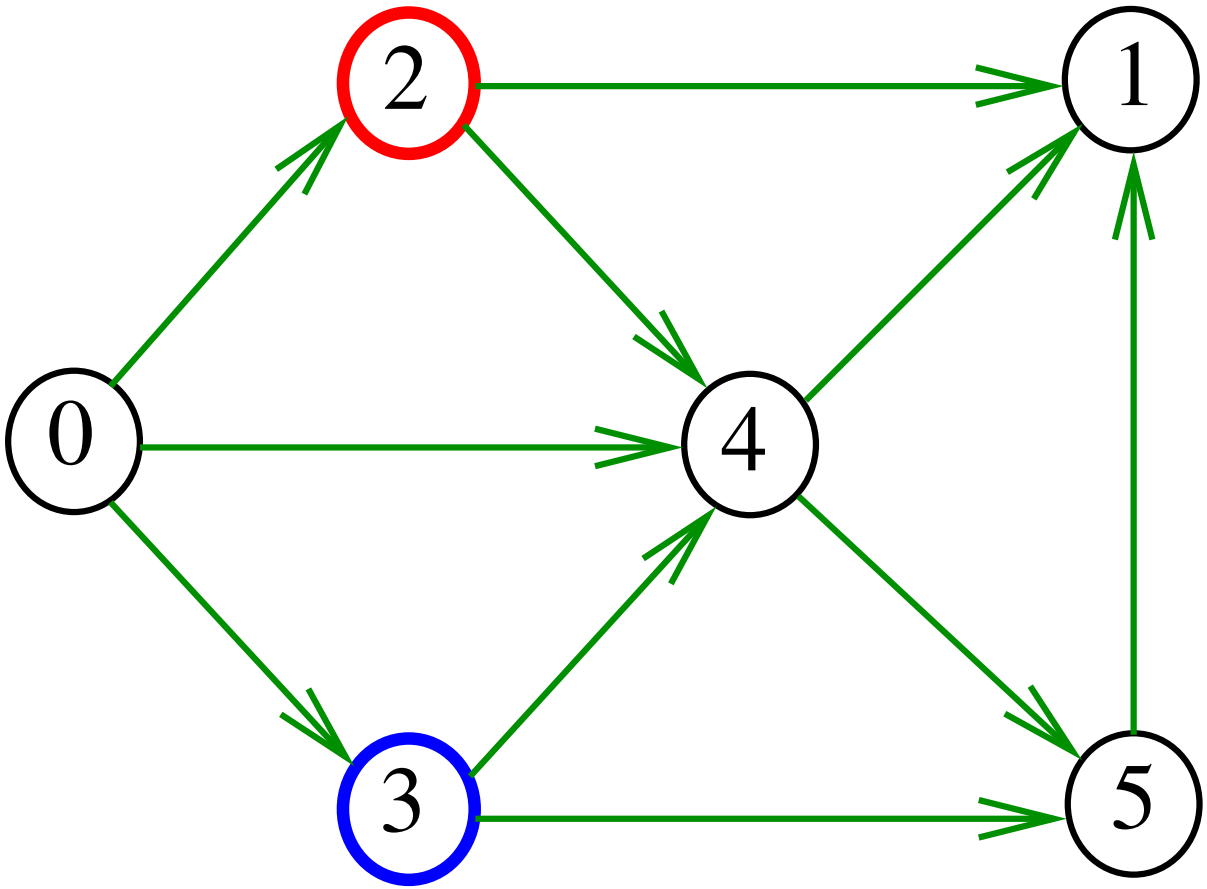
```
static void dfs(Digraph G, int v, dfsPaths T){
   Link w;
   T->marked[v] = true;
   for (w = G->adj[v]; w != NULL; w = w->next)
      if (!T->marked[w->vertex]) {
         T->edgeTo[w->vertex] = v;
         dfs(G, w->vertex, T);
      }
   }
}
```

DFSpaths(G, 0)




```
0-2 dfs(G,2)
  2-1 dfs(G,1)
  2-4 dfs(G,4)
    4-1
    4-5 dfs(G,5)
      5-1
0-3 dfs(G,3)
  3-4
  3-5
0-4
existe caminho
```

DFSpaths(G, 2)




```
2-1 dfs(G,1)
2-4 dfs(G,4)
   4-1
   4-5 dfs(G,5)
      5-1
nao existe caminho
```

Consumo de tempo

Qual é o consumo de tempo de DFSpaths?

```
dfsPaths DFSpaths(Digraph G, int s) {
   dfsPaths T = DFSpathsInit(G);
   T->edgeTo[s] = s;
   dfs(G, s, T);
   return T;
}
```

Consumo de tempo

Qual é o consumo de tempo de DFSpaths?

```
dfsPaths DFSpaths(Digraph G, int s) {
   dfsPaths T = DFSpathsInit(G);
   T->edgeTo[s] = s;
   dfs(G, s, T);
   return T;
}
```

O consumo de tempo de DFSpaths é $\Theta(V)$ mais o consumo de tempo da função dfs().

Consumo de tempo

Qual é o consumo de tempo da função `dfs`?

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

Conclusão

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

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

Conclusão

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

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

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 apenas **visita os vértices que podem ser atingidos** a partir do vértice inicial; a busca em profundidade, tipicamente, **visita todos os vértices** do digrafo.

Certificados

Achievement Certificate

_____

is presented with the

Computer Achievement Award

On the _____ Day of _____ In the Year _____.

Signed,

_____

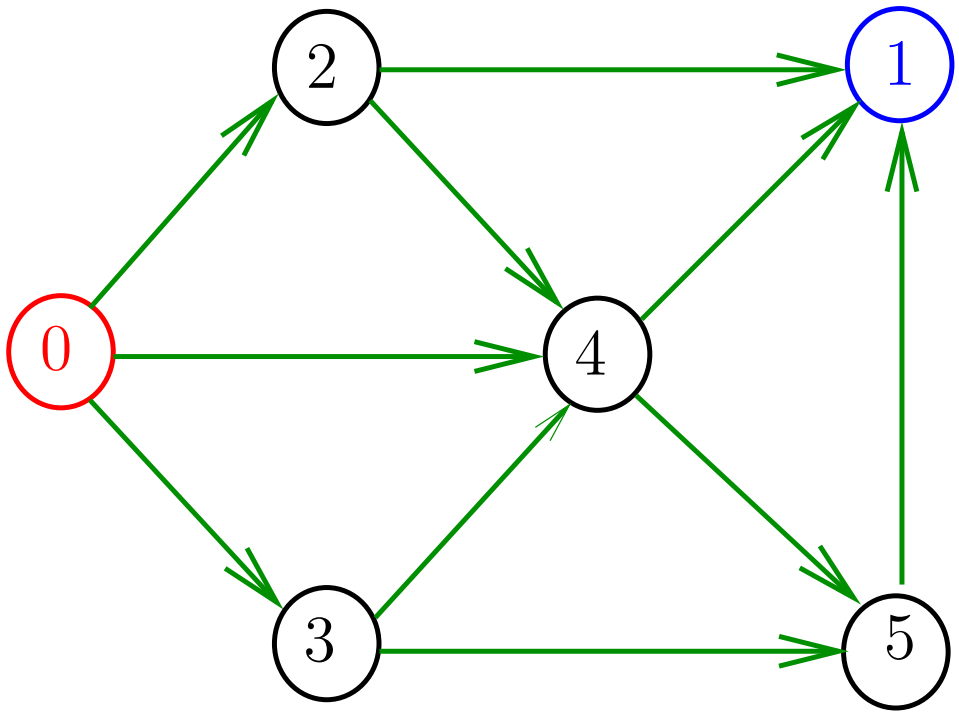
Certificate Provided by www.hooverwebdesign.com

Fonte: Free Printable Computer Achievement Award Certificates

Procurando um caminho

Problema: dados um digrafo `G` e dois vértices `s` e `t`, decidir se existe um caminho de `s` a `t`

Exemplo: para `s` $= 0$ e `t` $= 1$, a resposta é SIM



Certificados

Como é possível 'verificar' a resposta?

Como é possível 'verificar' que existe caminho?

Como é possível 'verificar' que não existe caminho?

Certificados

Como é possível 'verificar' a resposta?

Como é possível 'verificar' que existe caminho?

Como é possível 'verificar' que não existe caminho?

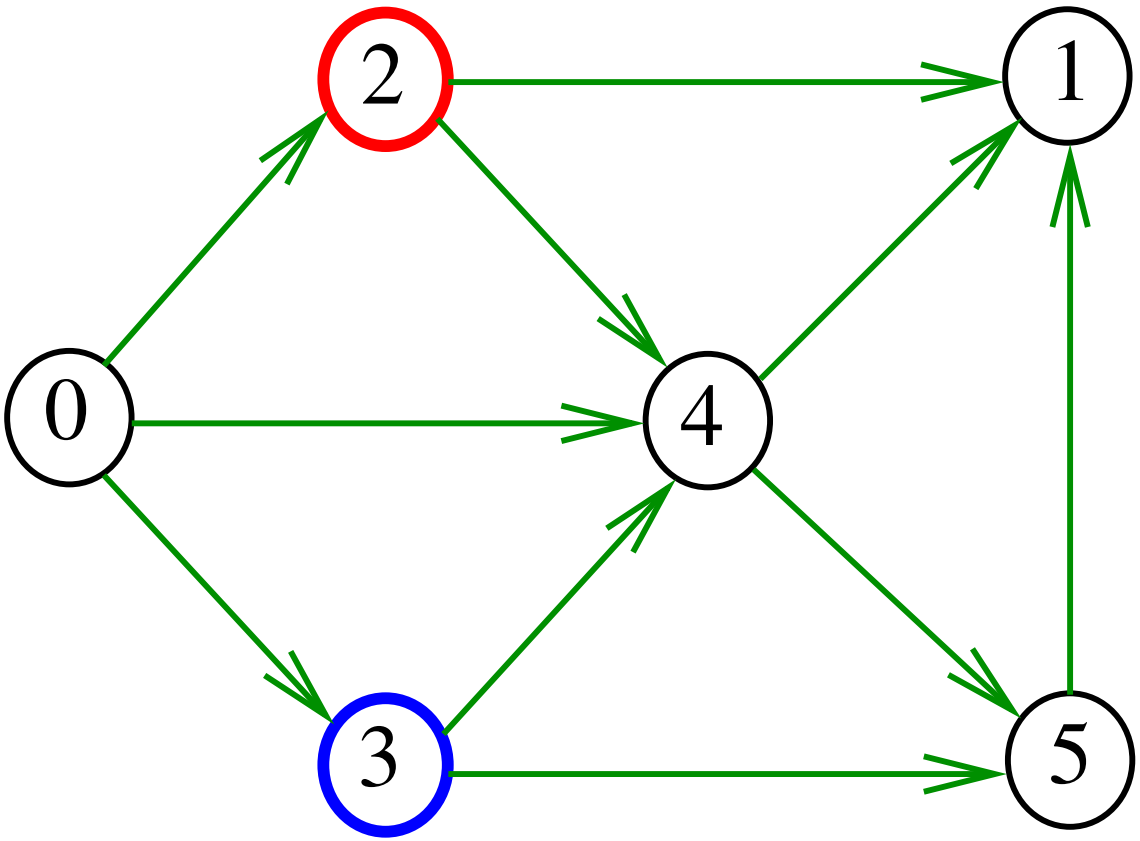
Veremos questões deste tipo frequentemente

Elas terão um papel **suuupeeer** importante no final de MAC0338 Análise de Algoritmos e em MAC0414 Autômatos, Computabilidade e Complexidade.

Elas estão relacionadas com o Teorema da Dualidade visto em MAC0315 Otimização Linear.

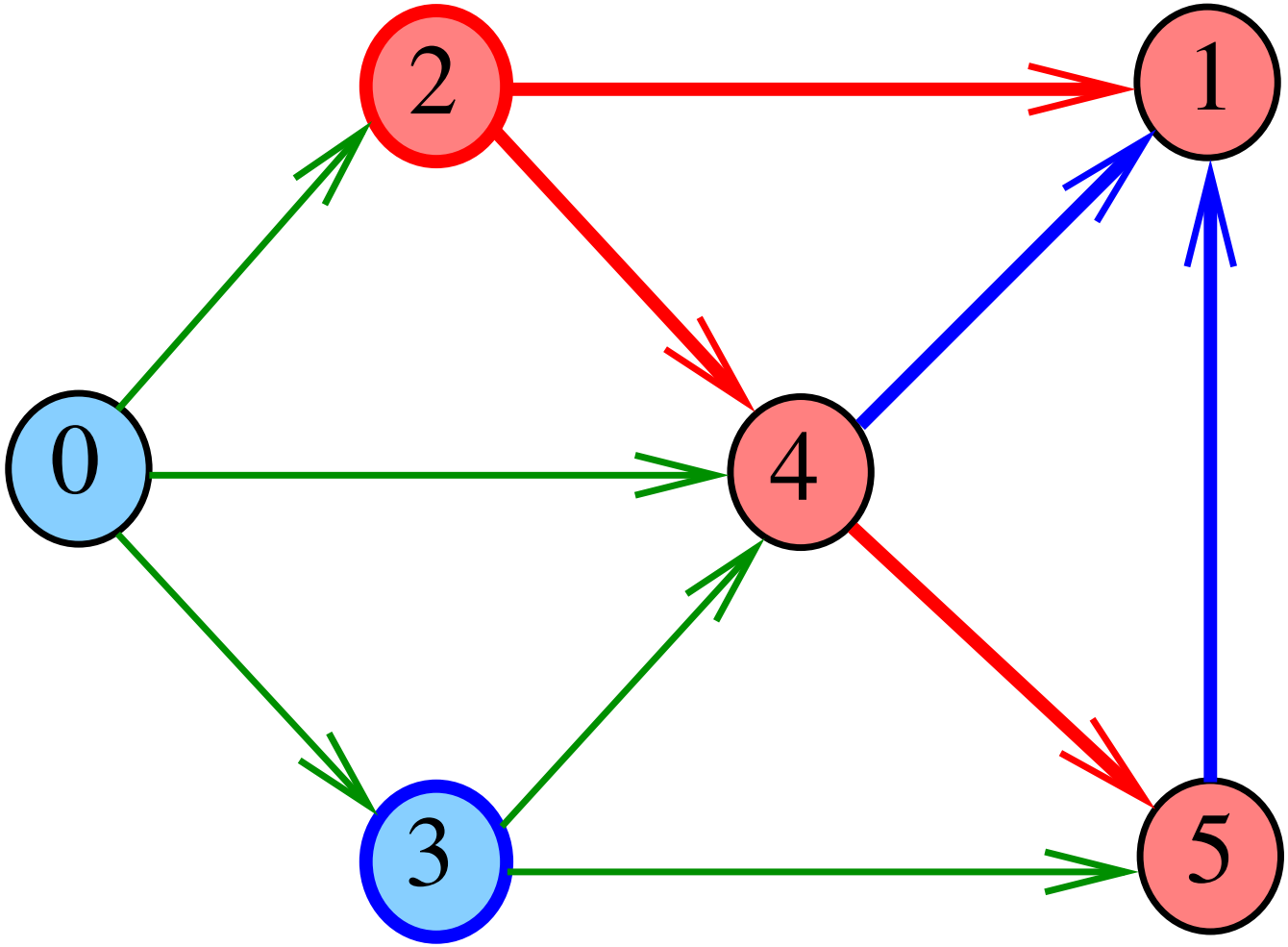
Certificado de inexistência

Como é possível demonstrar que o problema não tem solução?




```
dfs(G,2)
  2-1 dfs(G,1)
  2-4 dfs(G,4)
    4-1
    4-5 dfs(G,5)
      5-1
nao existe caminho
```

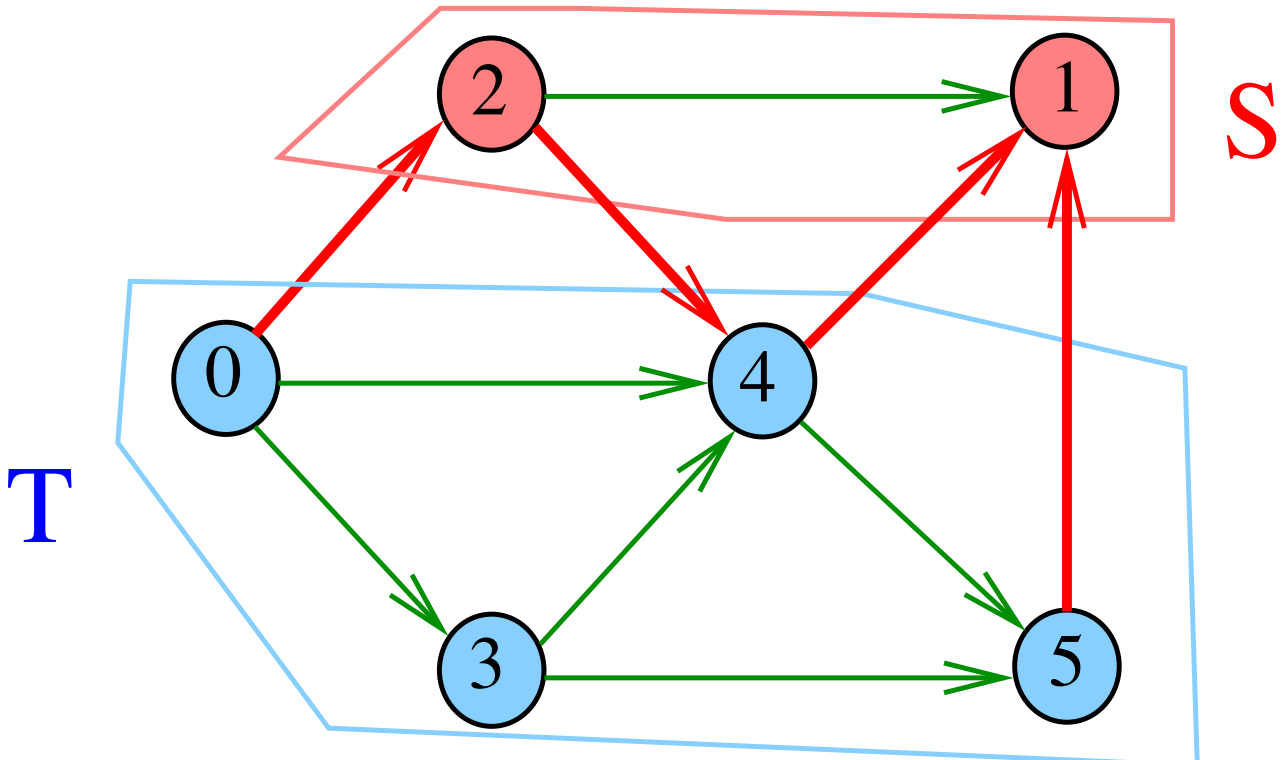
DFSpath(G,2,3)



Cortes (= *cuts*)

Um **corte** é uma bipartição do conjunto de vértices.
Um arco **pertence** ou **atravessa** um corte $(\mathtt{S}, \mathtt{T})$ se tiver uma ponta em $\mathtt{S}$ e outra em $\mathtt{T}$.

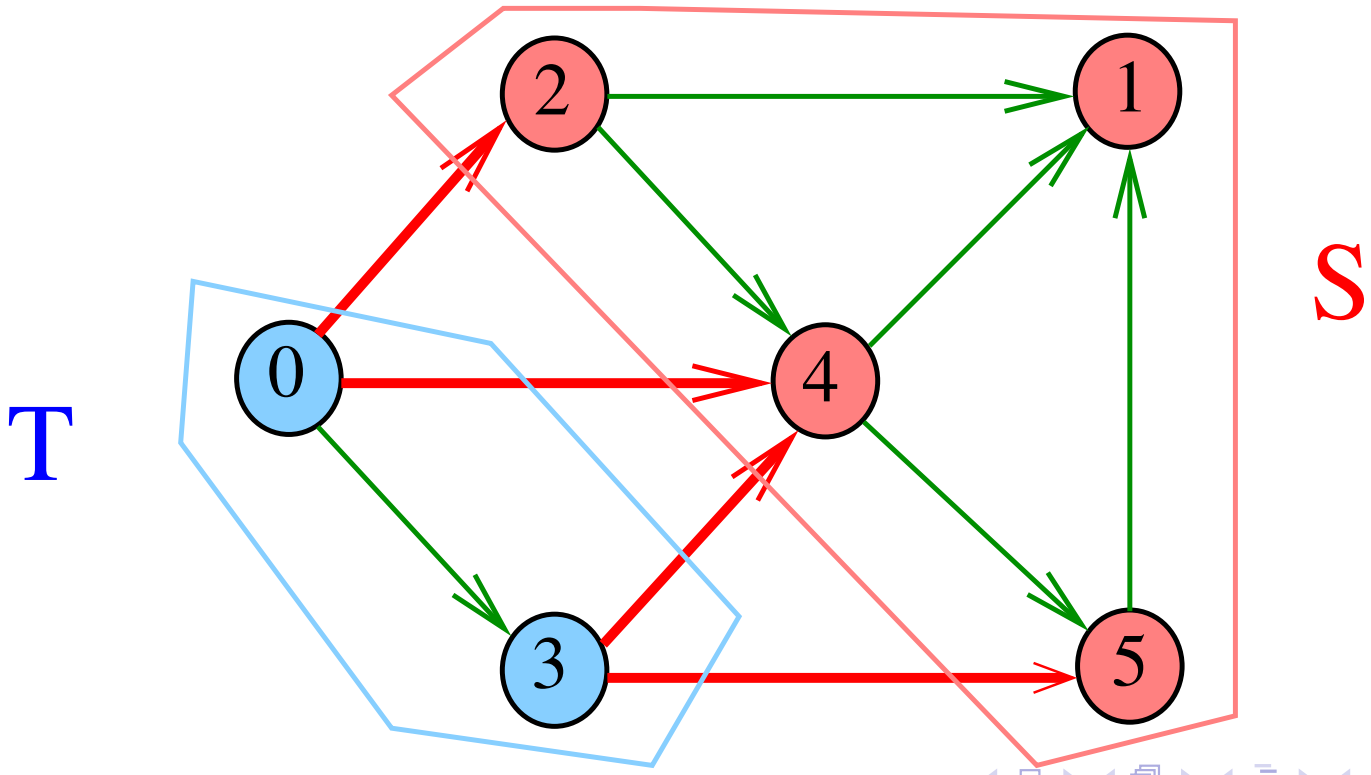
Exemplo 1: arcos em vermelho estão no corte $(\mathtt{S}, \mathtt{T})$



Cortes (= *cuts*)

Um **corte** é uma bipartição do conjunto de vértices.
Um arco **pertence** ou **atravessa** um corte $(\mathtt{S}, \mathtt{T})$ se tiver uma ponta em $\mathtt{S}$ e outra em $\mathtt{T}$.

Exemplo 2: arcos em vermelho estão no corte $(\mathtt{S}, \mathtt{T})$

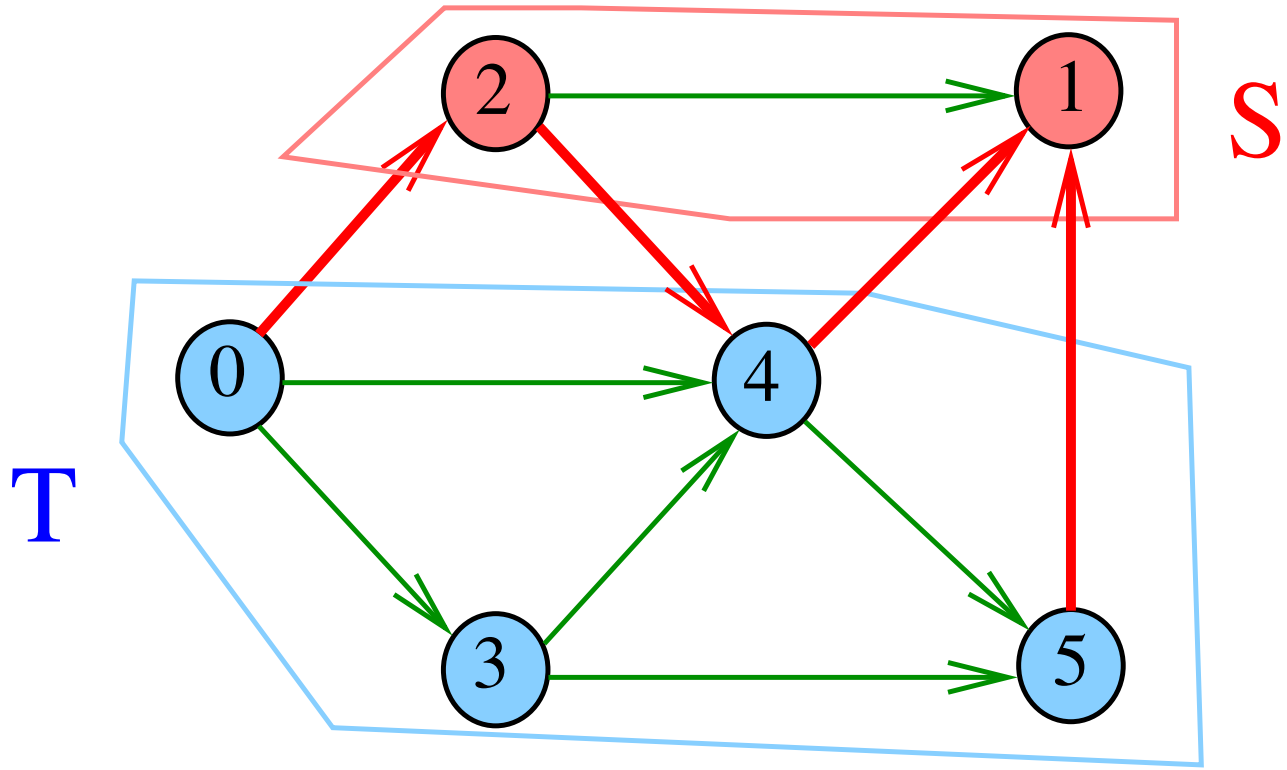


st-Cortes (= st-*cuts*)

Um corte (S, T) é um **st-corte** se

s está em S e t está em T.

Exemplo: (S, T) é um 1-3-corte, um 2-5-corte ...



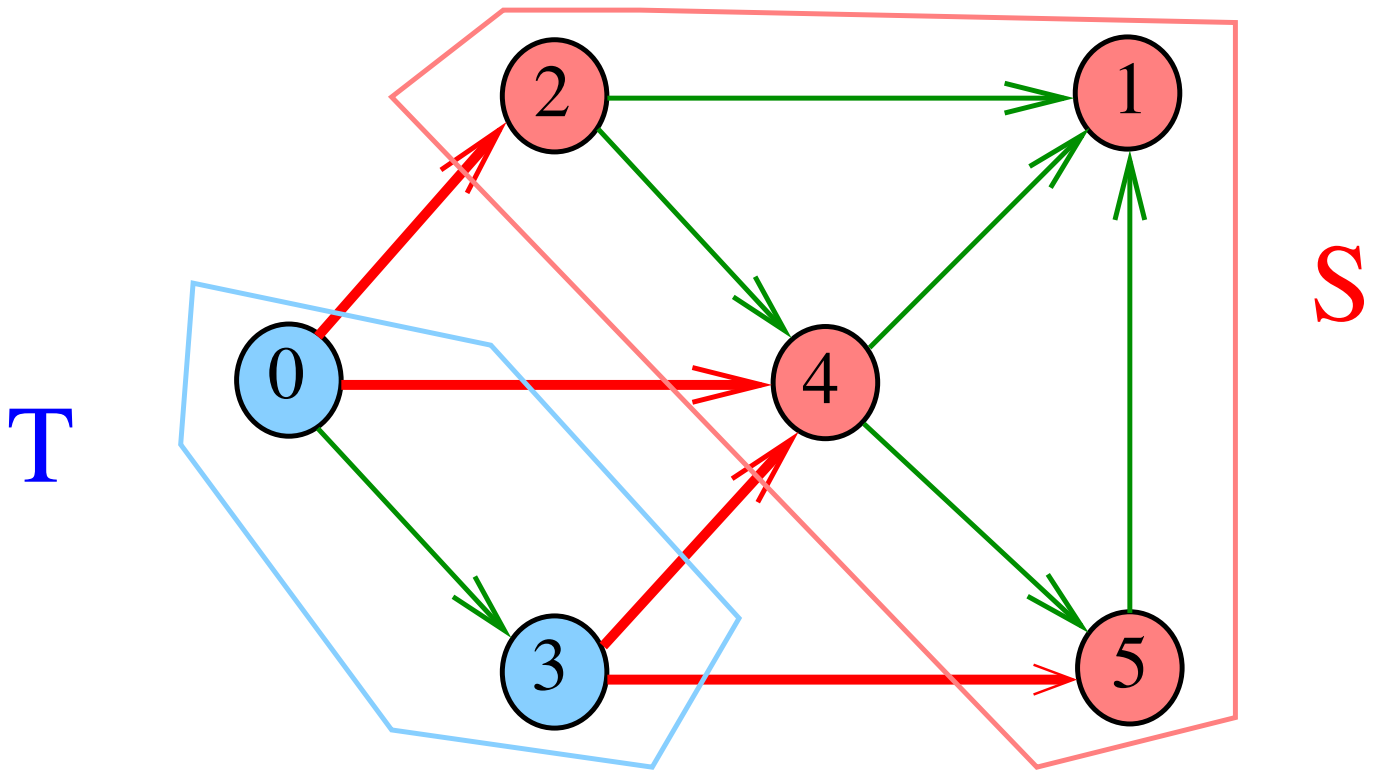
Certificado de inexistência

Para demonstrarmos que **não existe** um caminho de s a t, basta exibirmos um st-corte (S, T) em que

> ***todo arco*** *no corte tem*
> *ponta inicial em* T *e ponta final em* S.

Certificado de inexistência

Exemplo:
certificado de que não há caminho de 2 a 3



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 and yang (Wikipedia)

E DFSpaths e BFSpaths com isso?

Nos códigos de DFSpaths e BFSpaths,
se existe um caminho de s a t,
ele está representado no vetor edgeTo[].

Nos códigos de DFSpaths e BFSpaths,
se não existe um caminho de s a t,
um st-corte separando s de t está
representado no vetor marked[].

Em ambos os casos, podemos fazer
um trecho de código que verifica a resposta
em tempo proporcional a V + E.