

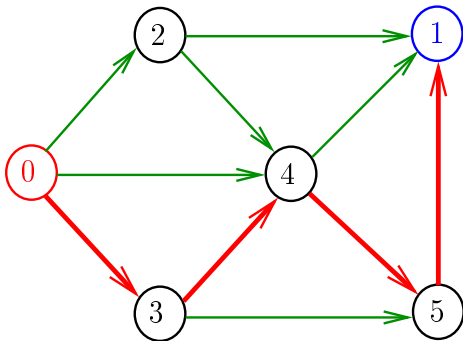
Melhores momentos

AULAS PASSADAS

Procurando um caminho

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

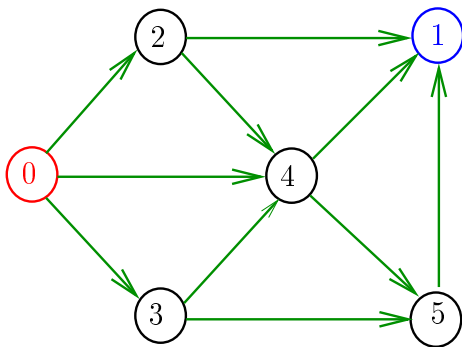
Exemplo: para $s = 0$ e $t = 1$ a resposta é **SIM**



Procurando um caminho

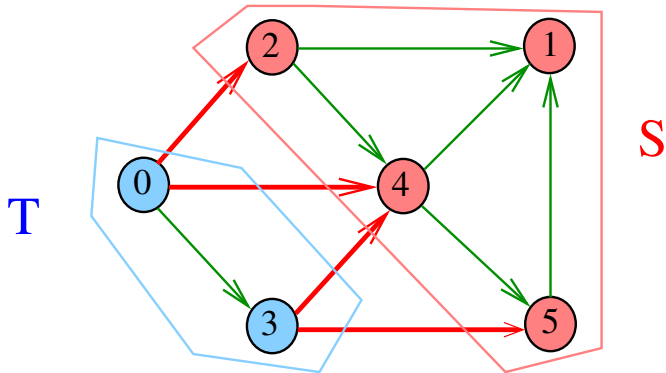
Problema: dados um digrafo G e dois vértices s e t decidir se existe um caminho de s a t

Exemplo: para $s = 5$ e $t = 4$ a resposta é **NÃO**



Certificado de inexistência

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



Consumo de tempo

A classe `DepthFirstPath`, para **vetor de listas de adjacência**, consome tempo $O(V + E)$ para encontrar caminhos a partir um dado vértice **s** a todos os vértices atingíveis a partir de **s**.

Certificados

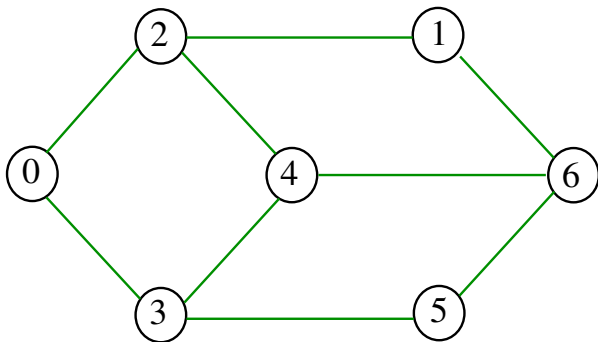
Para quaisquer vértices s e t de um digrafo, vale uma e apenas uma das seguintes afirmações:

- ▶ 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 .

Bipartição

Um grafo é **bipartido** (= *bipartite*) se existe uma bipartição do seu conjunto de vértices tal que cada aresta tem uma ponta em uma das partes da bipartição e a outra ponta na outra parte

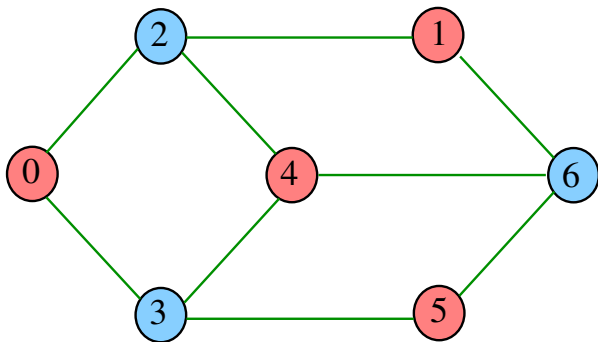
Exemplo:



Bipartição

Um grafo é **bipartido** (= *bipartite*) se existe uma bipartição do seu conjunto de vértices tal que cada aresta tem uma ponta em uma das partes da bipartição e a outra ponta na outra parte

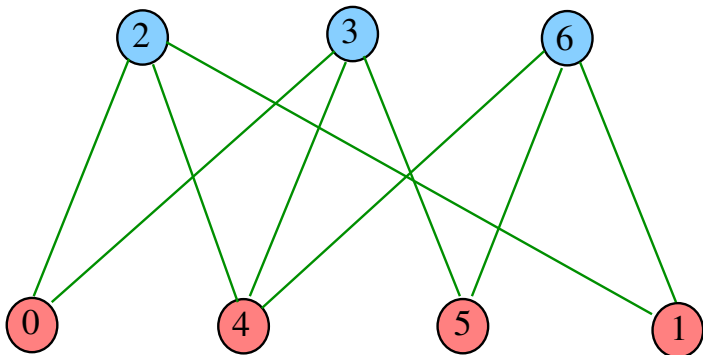
Exemplo:



Bipartição

Um grafo é **bipartido** (= *bipartite*) se existe uma bipartição do seu conjunto de vértices tal que cada aresta tem uma ponta em uma das partes da bipartição e a outra ponta na outra parte

Exemplo:



Consumo de tempo

A classe `TwoColor`, para **vetor de listas de adjacência**, consome tempo $O(V + E)$ para decidir se um grafo é bipartido.

Certificados

Para todo grafo G , vale uma e apenas uma das seguintes afirmações:

- ▶ G possui um ciclo ímpar
- ▶ G é bipartido

AULA 17

Busca DFS

S 18.1 e 18.2

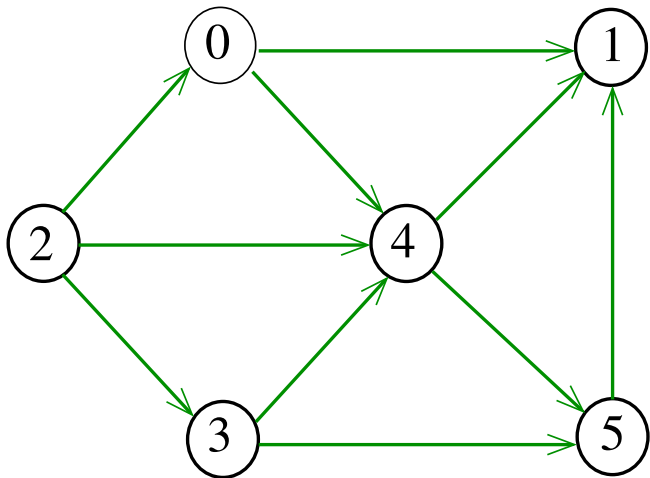
Busca ou varredura

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

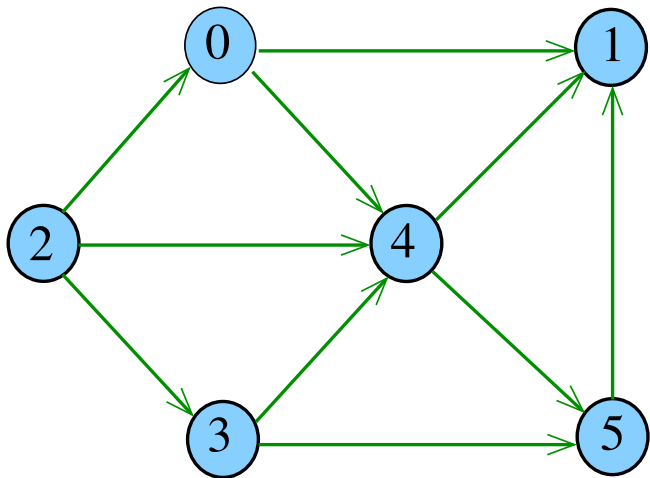
Cada arco é examinado **uma só vez**.

Depois de visitar sua ponta inicial o algoritmo percorre o arco e visita sua ponta final.

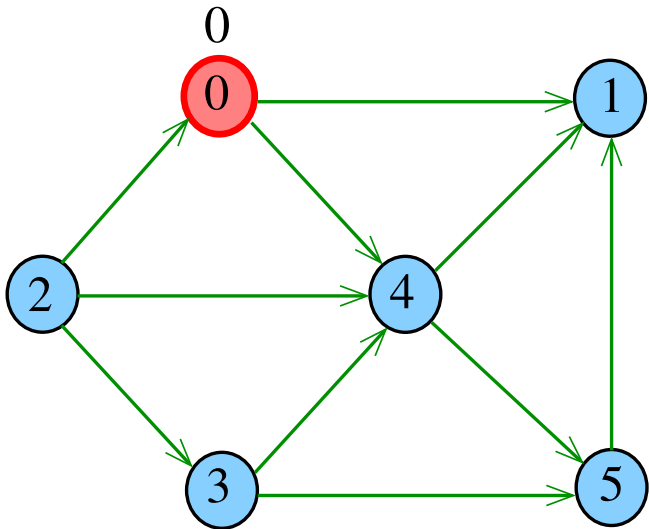
DFS(G)



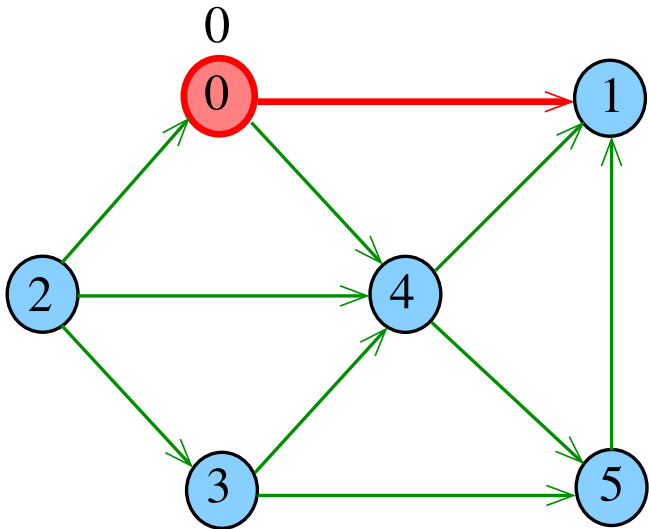
DFS(G)



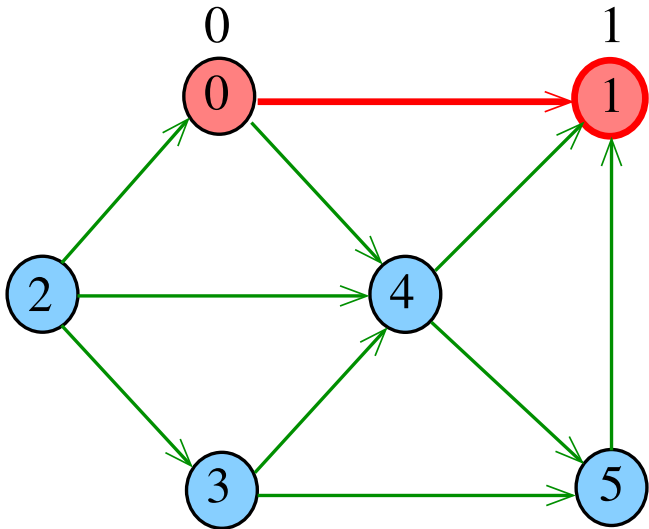
dfs(G,0)



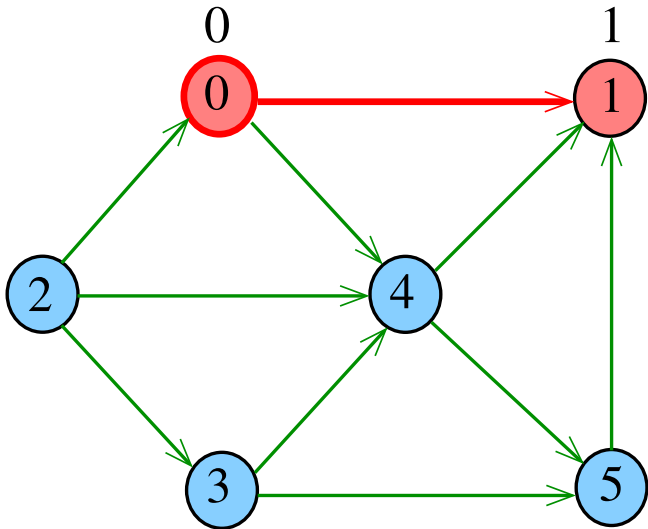
dfs(G,0)



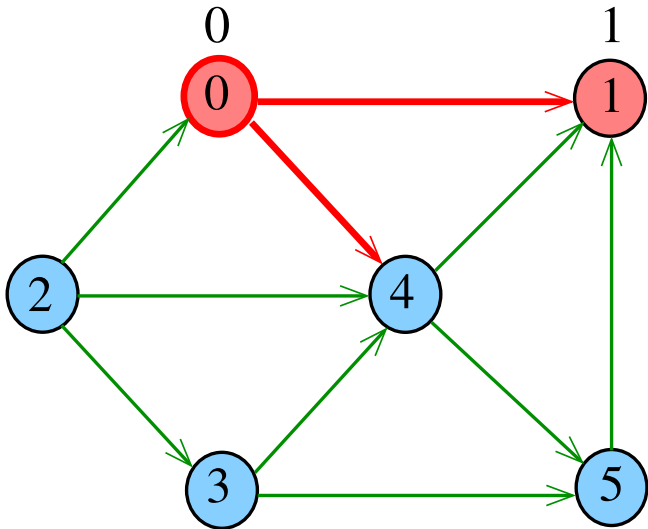
dfs(G, 1)



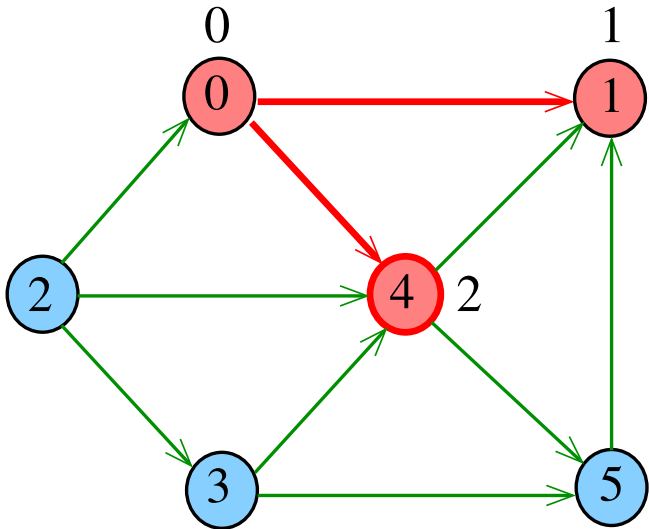
dfs(G,0)



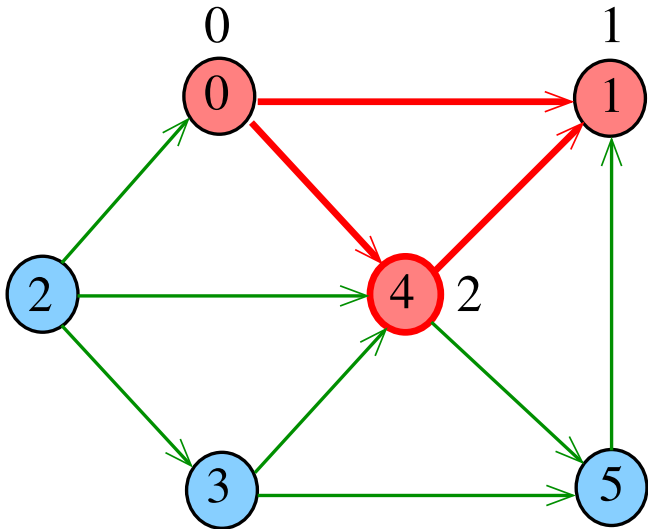
dfs(G,0)



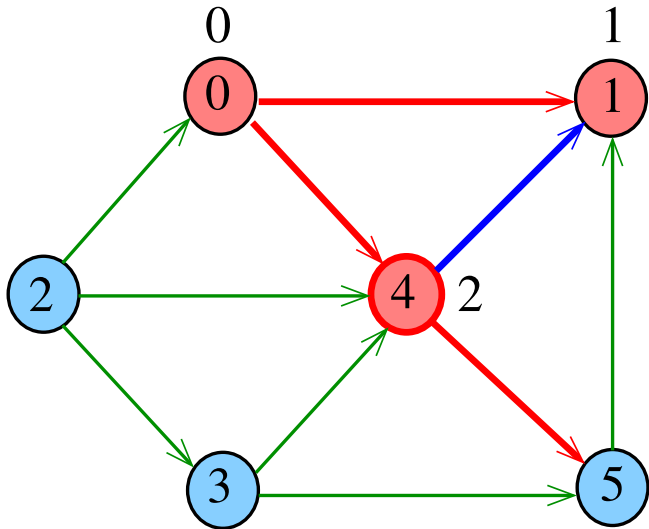
dfs(G,4)



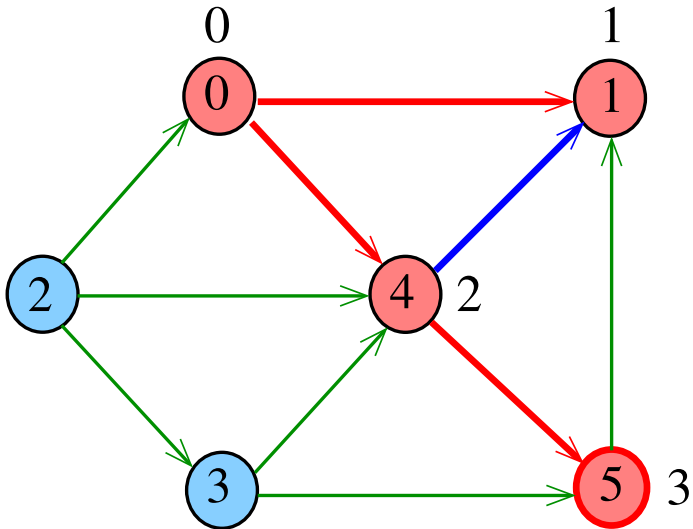
dfs(G,4)



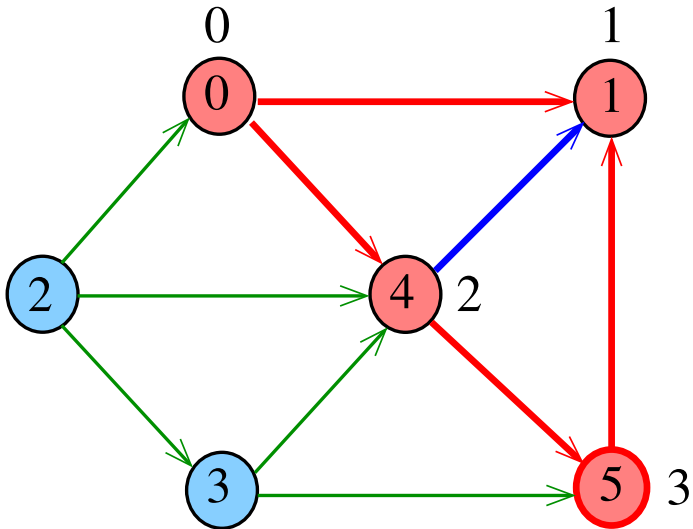
dfs(G,4)



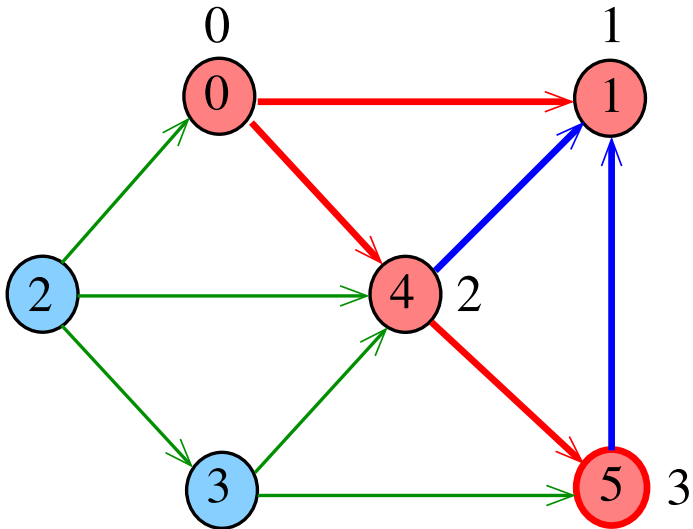
dfs(G, 5)



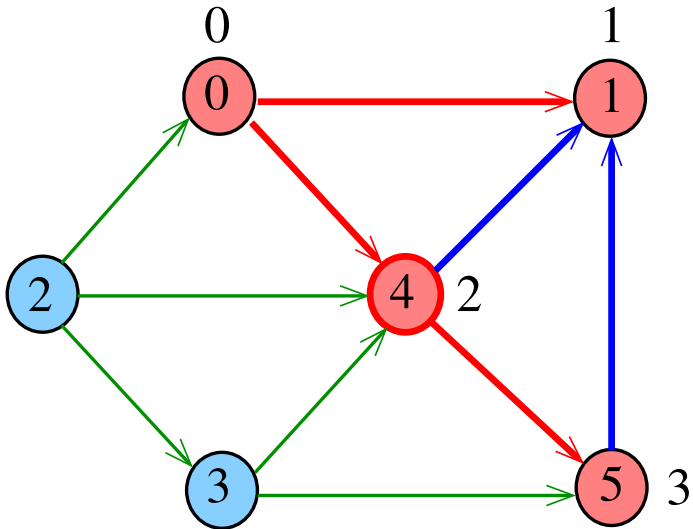
dfs(G, 5)



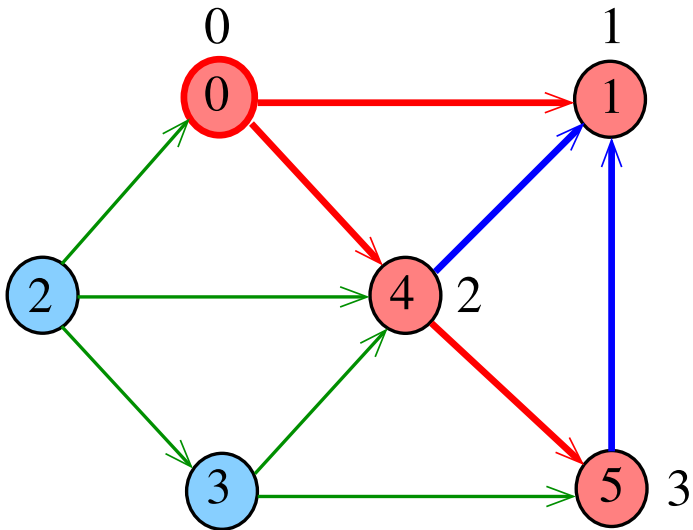
dfs(G, 5)



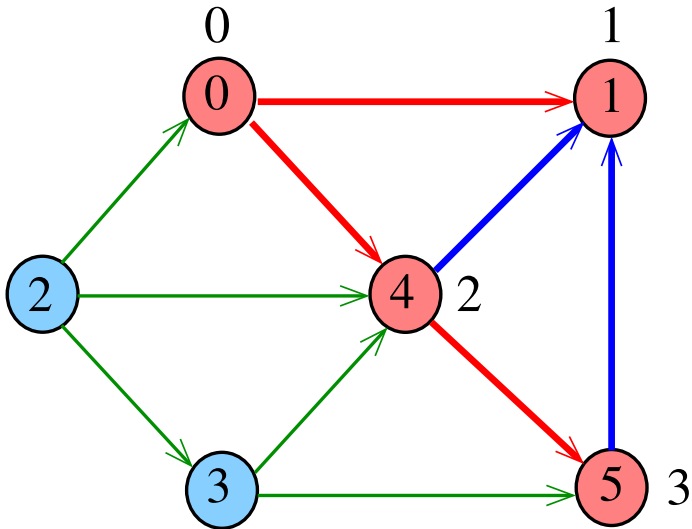
dfs(G,4)



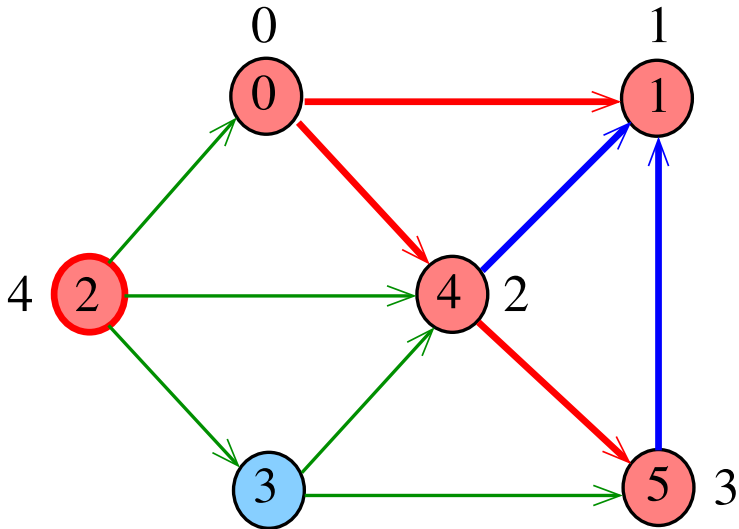
dfs(G,0)



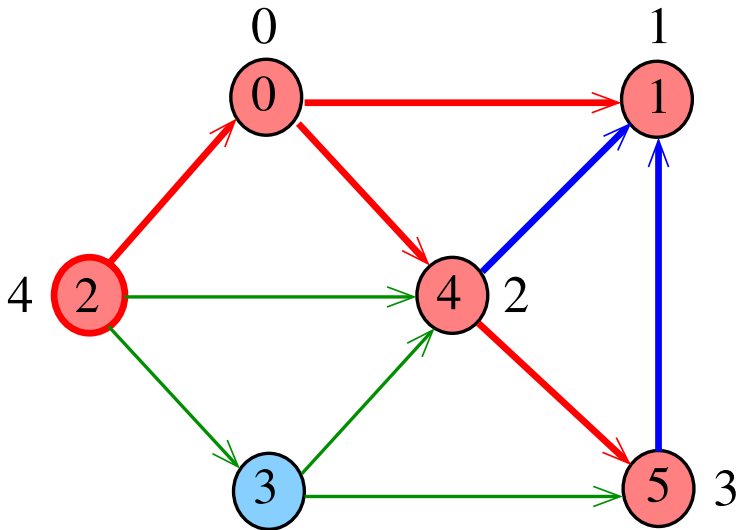
DIGRAPHdfs(G)



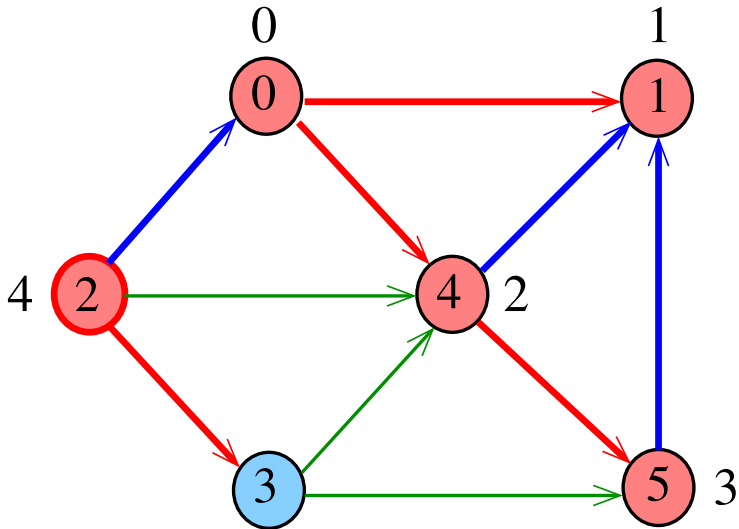
dfs(G,2)



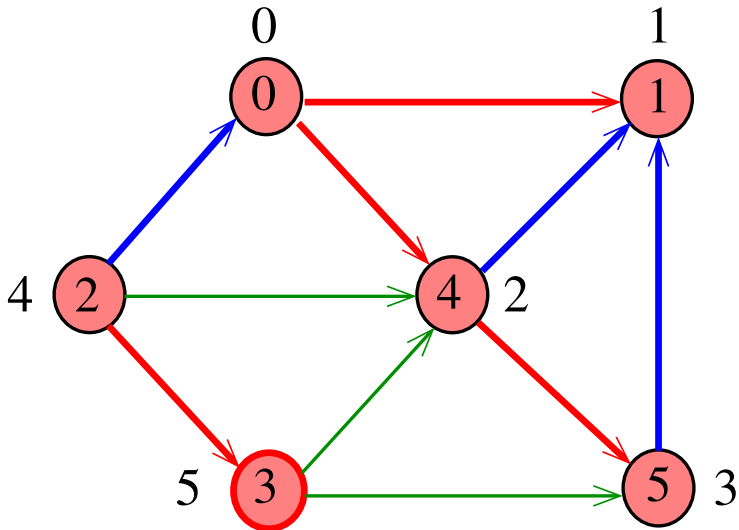
dfs(G,2)



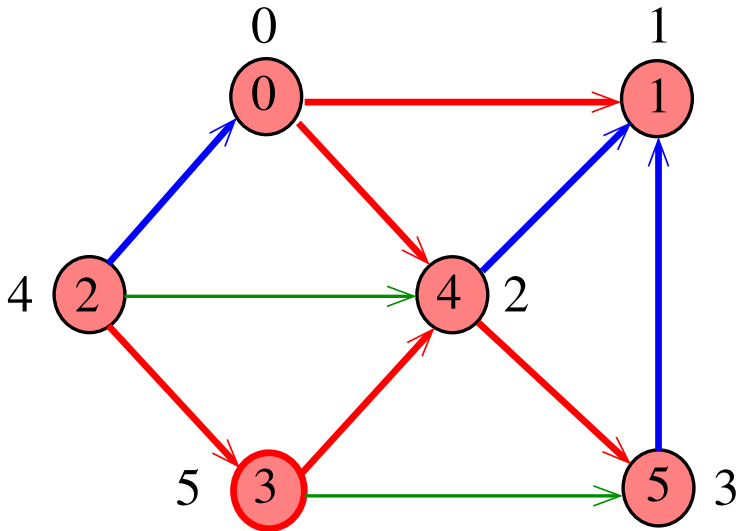
dfs(G, 2)



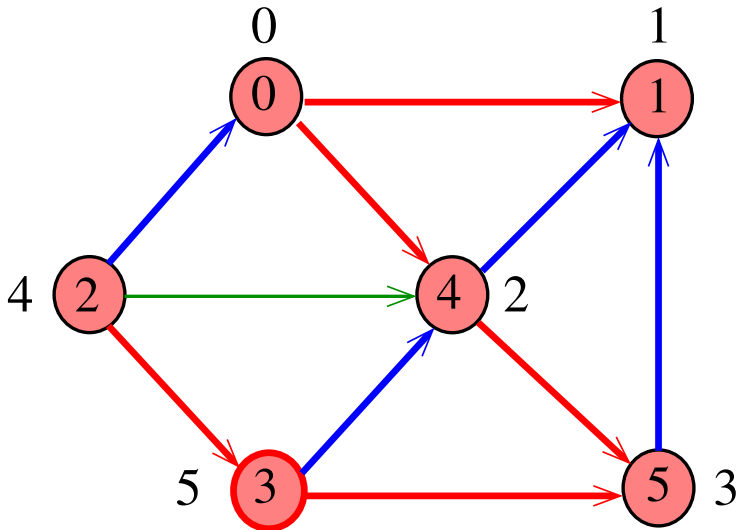
dfs(G,3)



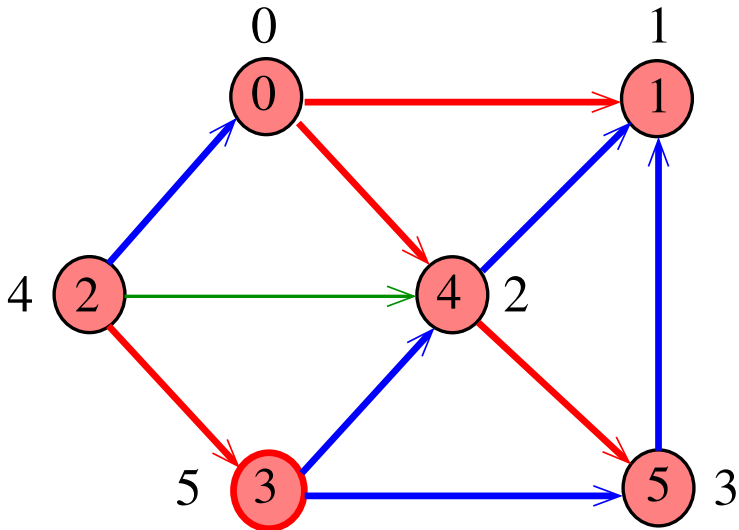
dfs(G,3)



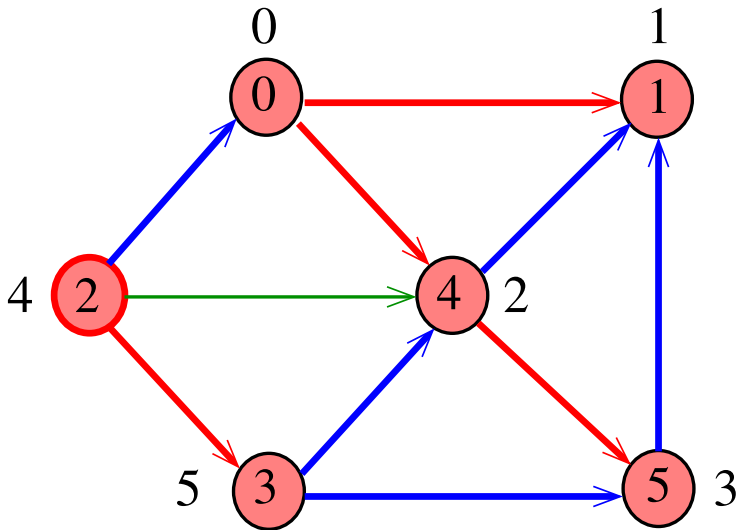
dfs(G,3)



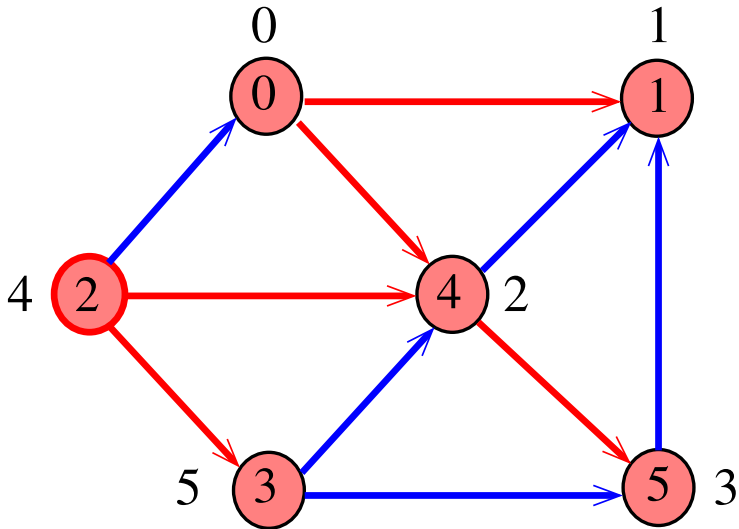
dfs(G,3)



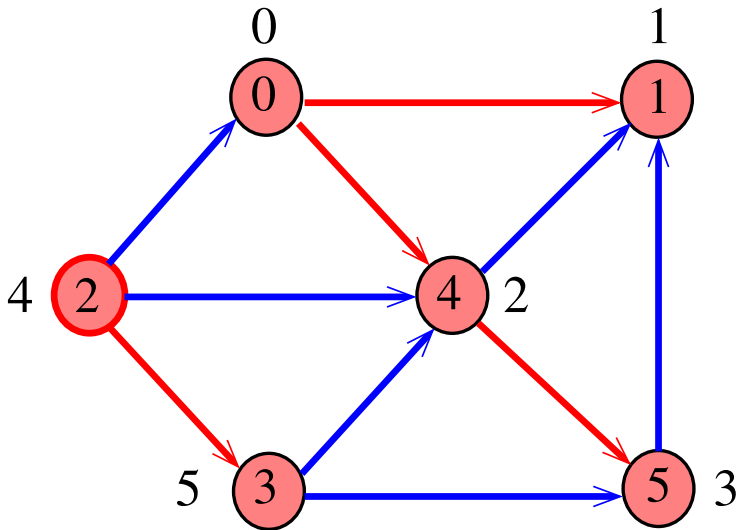
dfs(G,2)



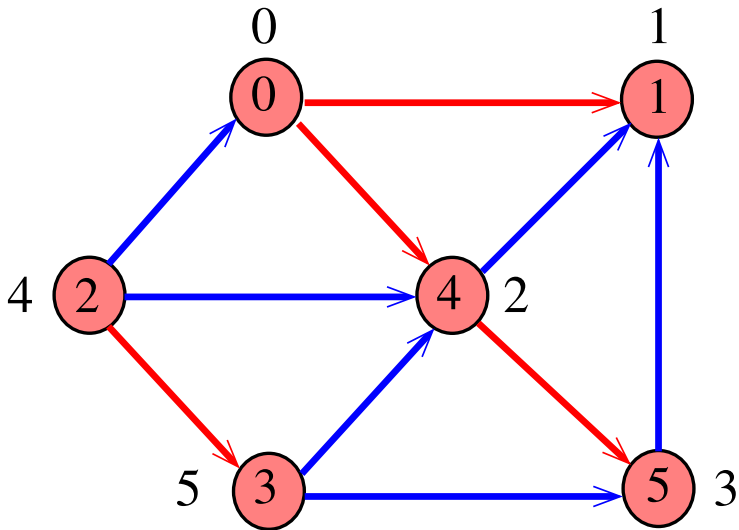
dfs(G,2)



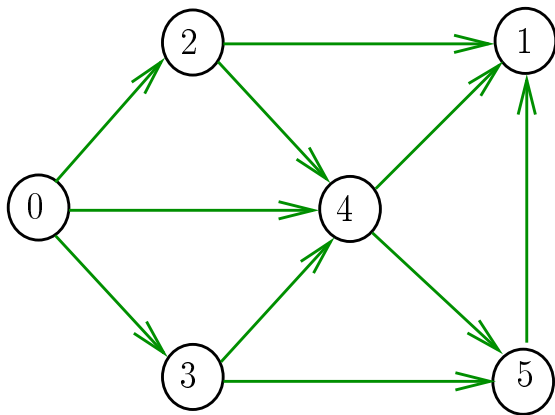
dfs(G,2)



DIGRAPHdfs(G)



DFS(G)



dfs(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

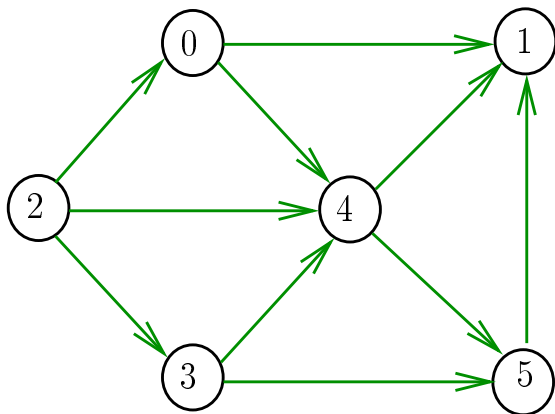
0-3 dfs(G, 3)

3-4

3-5

0-4

DFS(G)



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

Consumo de tempo

O consumo de tempo de DFS para vetor de listas de adjacência é $\Theta(V + E)$.

O consumo de tempo de DFS para matriz de adjacência é $\Theta(V^2)$.

Arborescência de busca em profundidade

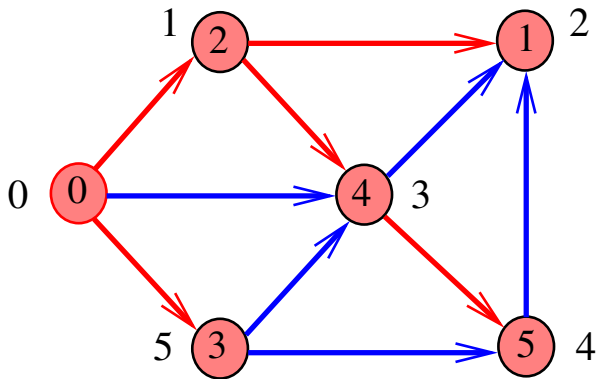
Classificação dos arcos

S 18.4 e 19.2
CLRS 22

Arcos da arborescência

Arcos da arborescência são os arcos $v-w$ que dfsR percorre para visitar w pela primeira vez

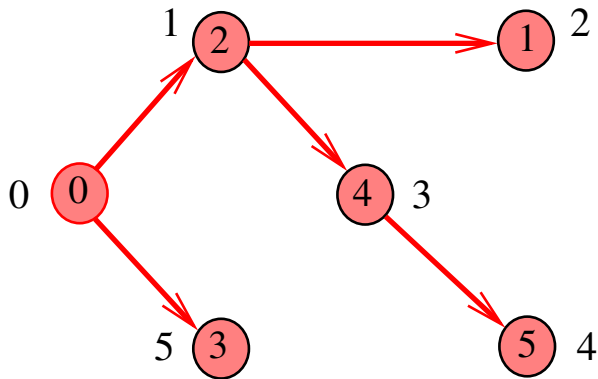
Exemplo: arcos em **vermelho** são arcos da arborescência



Arcos da arborescência

Arcos da arborescência são os arcos $v-w$ que $dfsR$ percorre para visitar w pela primeira vez

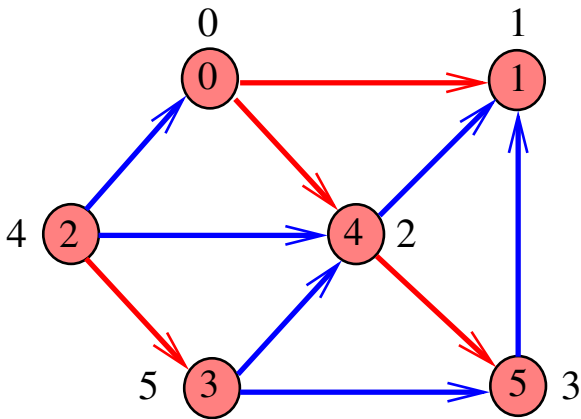
Exemplo: arcos em **vermelho** são arcos da arborescência



Floresta DFS

Conjunto de arborescências é a **floresta da busca em profundidade** (= *DFS forest*)

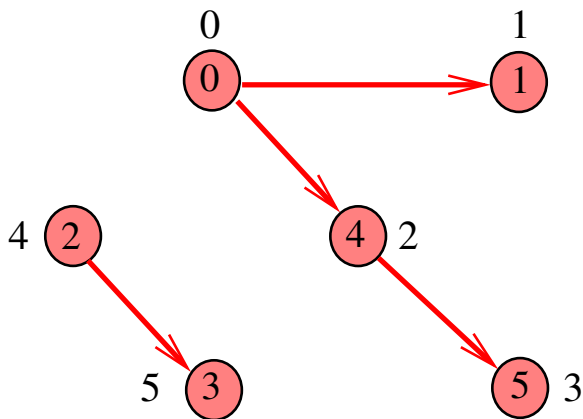
Exemplo: arcos em **vermelho** formam a floresta DFS



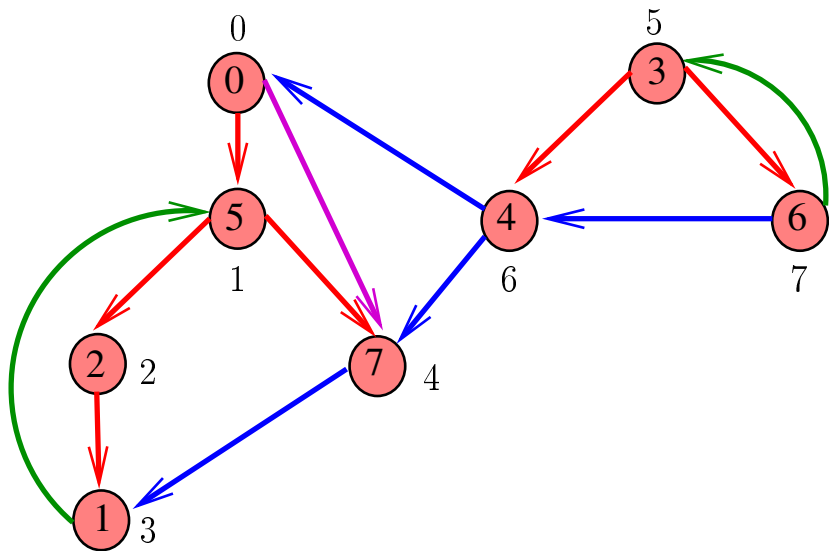
Floresta DFS

Conjunto de arborescências é a **floresta da busca em profundidade** (= *DFS forest*)

Exemplo: arcos em **vermelho** formam a floresta DFS

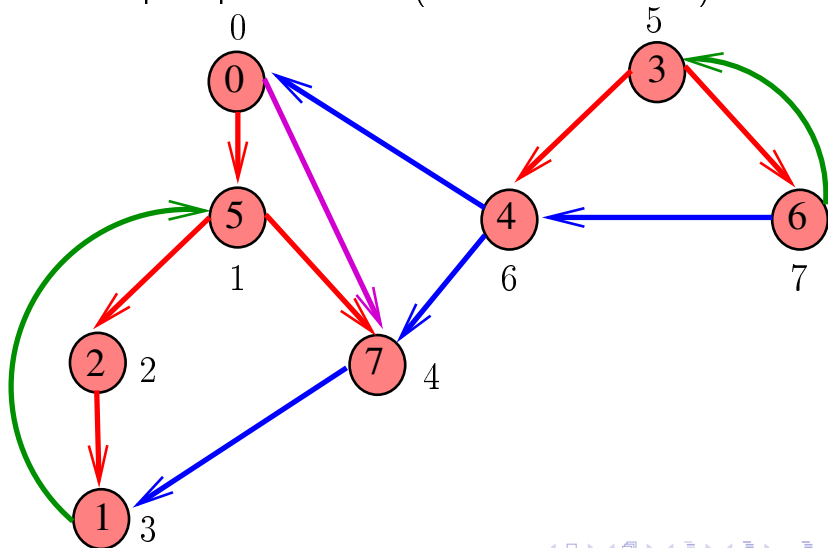


Classificação dos arcos



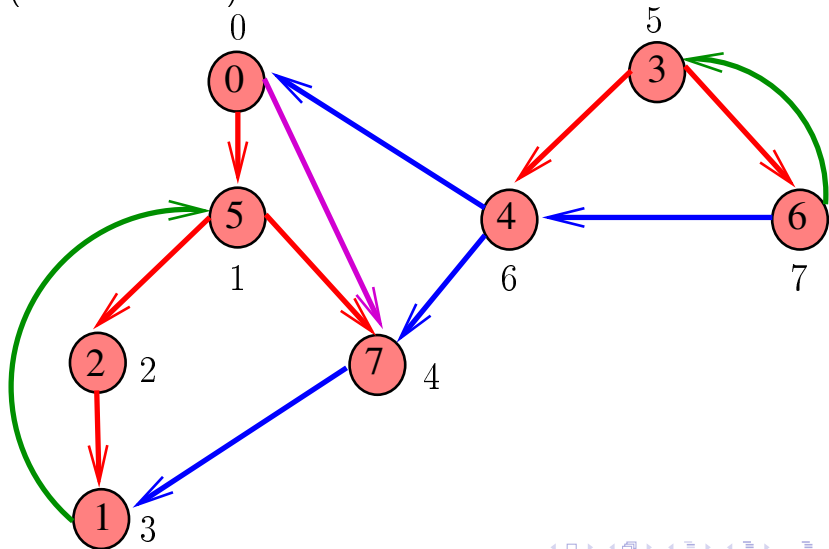
Arcos de arborescência

$v-w$ é **arco de arborescência** se foi usado para visitar w pela primeira vez (arcos **vermelhos**)



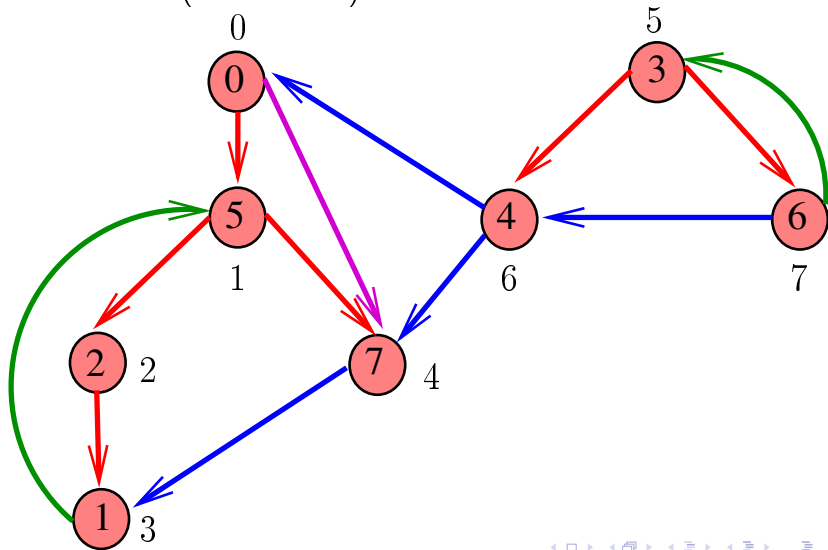
Arcos de retorno

$v-w$ é **arco de retorno** se w é ancestral de v
(arcos **verdes**)



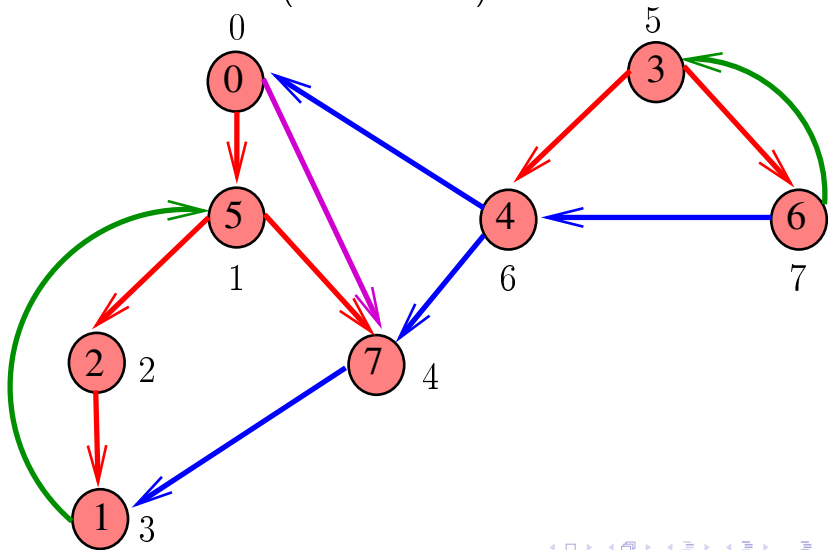
Arcos descendentes

$v-w$ é **descendente** se w é descendente de v , mas não é filho (arco **roxo**)



Arcos cruzados

$v-w$ é **arco cruzado** se w não é ancestral nem descendente de v (arcos **azuis**)



Busca DFS (CLRS)

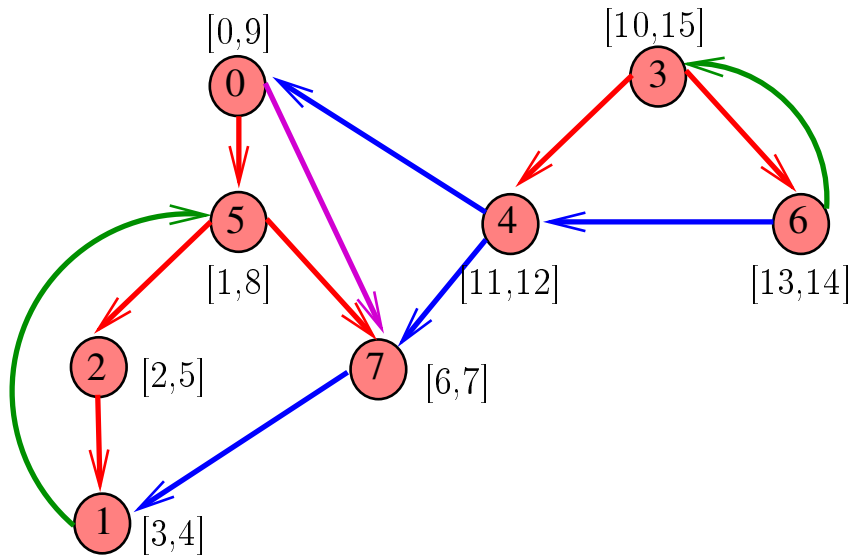
Vamos supor que nossos digrafos têm no máximo maxV vértices

```
private int time;  
private int [] d = new int[G.V()];  
private int [] f = new int[G.V()];
```

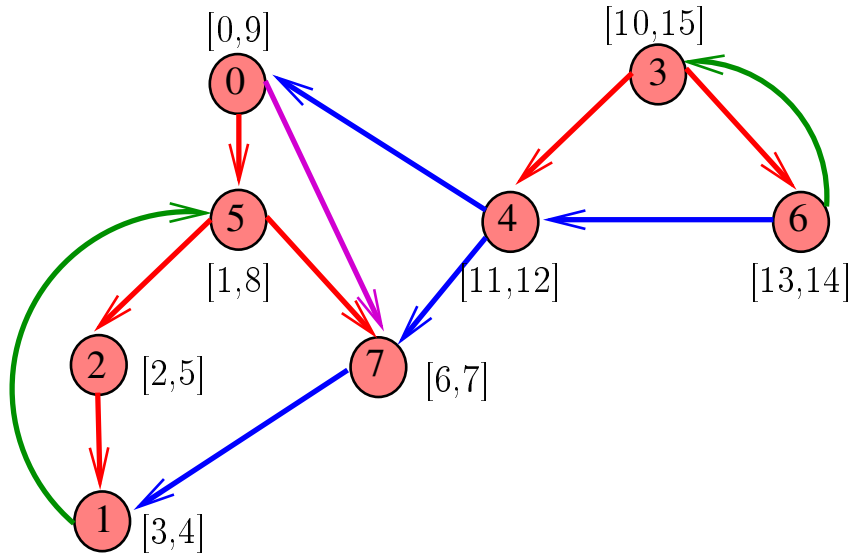
DIGRAPHdfs visita todos os vértices e arcos do digrafo G .

A função registra em $d[v]$ o 'momento' em que v foi descoberto e em $f[v]$ o momento em que ele foi completamente examinado

Busca DFS (CLRS)

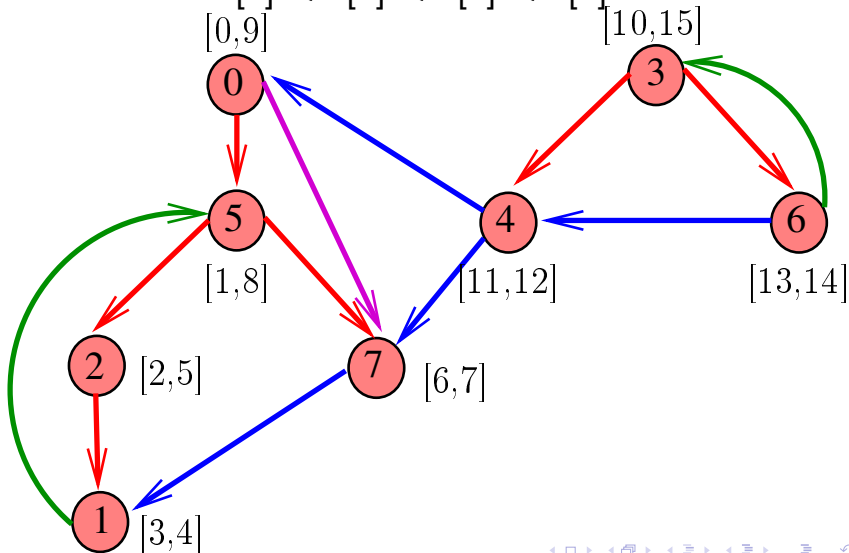


Classificação dos arcos



Arcos de arborescência ou descendentes

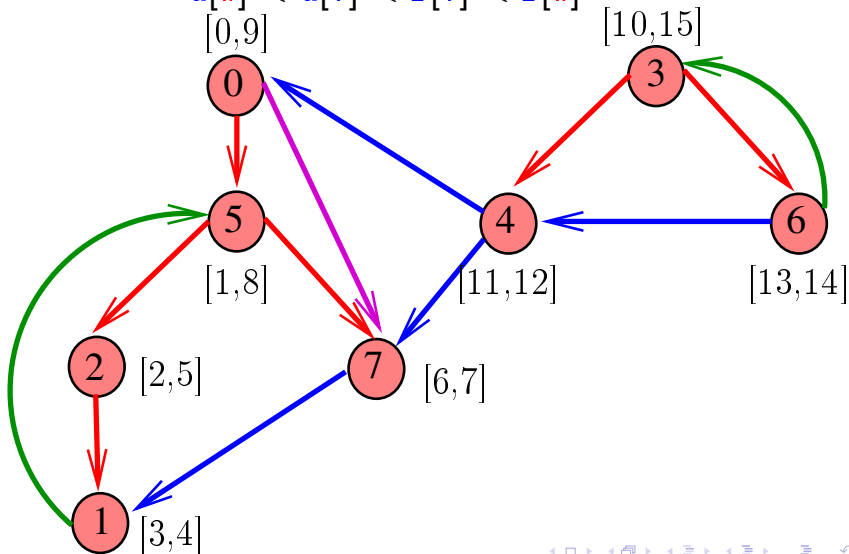
$v-w$ é **arco de arborescência** ou **descendente** se e somente se $d[v] < d[w] < f[w] < f[v]$



Arcos de retorno

$v-w$ é **arco de retorno** se e somente se

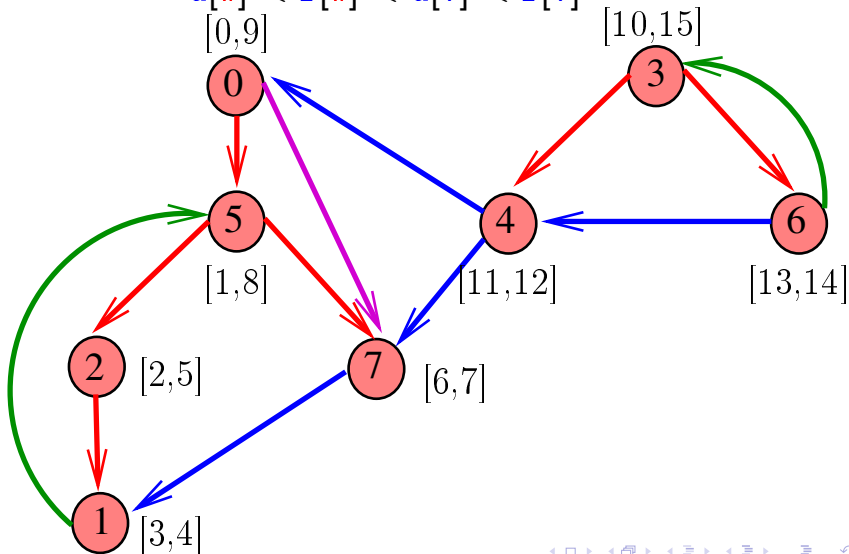
$$d[w] < d[v] < f[v] < f[w]$$



Arcos cruzados

$v-w$ é arco **cruzado** se e somente se

$$d[w] < f[w] < d[v] < f[v]$$



Conclusões

$v-w$ é:

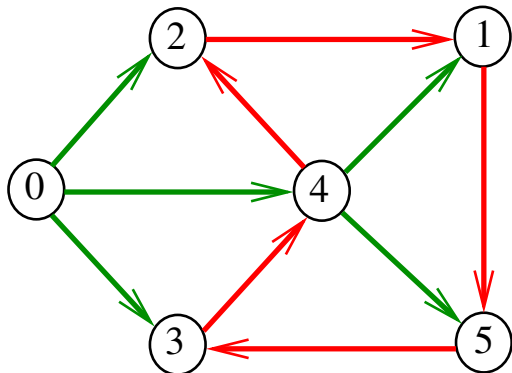
- ▶ **arco de arborescência** se e somente se $d[v] < d[w] < f[w] < f[v]$ e $\text{edgeTo}[w] = v$;
- ▶ **arco descendente** se e somente se $d[v] < d[w] < f[w] < f[v]$ e $\text{edgeTo}[w] \neq v$;
- ▶ **arco de retorno** se e somente se $d[w] < d[v] < f[v] < f[w]$;
- ▶ **arco cruzado** se e somente se $d[w] < f[w] < d[v] < f[v]$;

Ciclos em digrafos

Ciclos

Um **ciclo** num digrafo é qualquer seqüência da forma $v_0-v_1-v_2-\dots-v_{k-1}-v_p$, onde $v_{k-1}-v_k$ é um arco para $k = 1, \dots, p$ e $v_0 = v_p$.

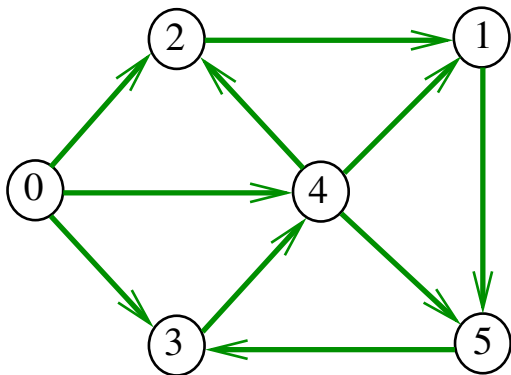
Exemplo: 2-1-5-3-4-2 é um ciclo



Procurando um ciclo

Problema: decidir se dado digrafo G possui um ciclo

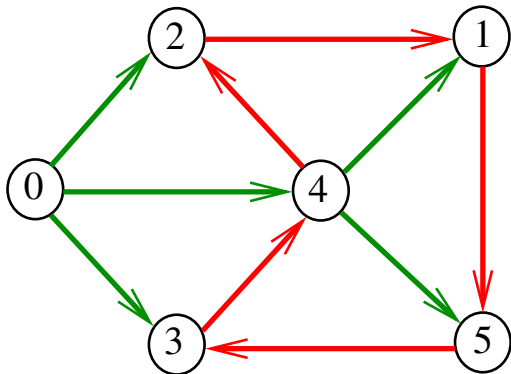
Exemplo: para o grafo a seguir a resposta é **SIM**



Procurando um ciclo

Problema: decidir se dado digrafo G possui um ciclo

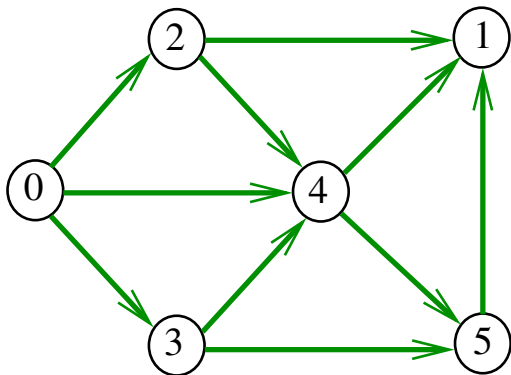
Exemplo: para o grafo a seguir a resposta é **SIM**



Procurando um ciclo

Problema: decidir se dado digrafo G possui um ciclo

Exemplo: para o grafo a seguir a resposta é **NÃO**



DirectedCycle café com leite

Recebe um digrafo G e decide se existe um ciclo.

Para cada arco $u-v$ podemos fazer

```
DirectedDFS dfs = new DirectedDFS(G,  
v);
```

e verificar se `dfs.hasPath(u)`

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

DirectedCycle

Recebe um digrafo G e decide se existe um ciclo em G .

```
public DirectedCycle (Digraph G);
```

O algoritmo tem por base a seguinte observação: em relação a **qualquer** floresta de busca em profundidade,

***todo** arco de **retorno** pertence a um ciclo e
todo ciclo tem um arco de **retorno***

DirectedCycle

vértices no caminho ativo

```
private boolean[] onPath;
```

ciclo

```
private Stack<Integer> cycle;
```

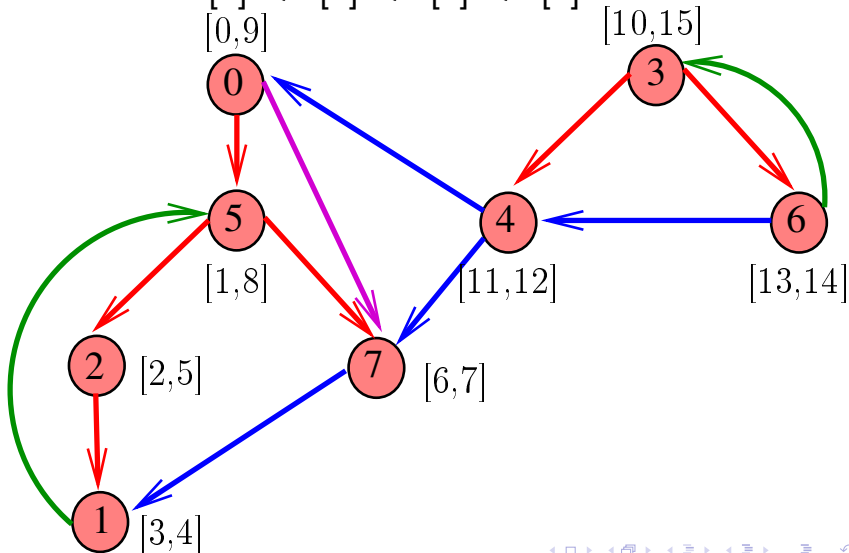
vértice no caminho

```
private int onCycle = -1;
```

Arcos de retorno

$v-w$ é **arco de retorno** se e somente se

$$d[w] < d[v] < f[v] < f[w]$$



Consumo de tempo

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

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

Digrafos acíclicos (DAGs)

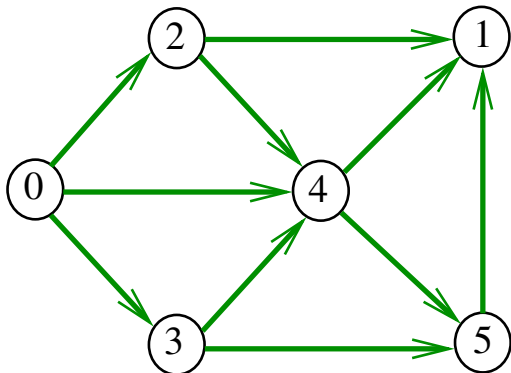
S 19.5 e 19.6

DAGs

Um digrafo é **acíclico** se não tem ciclos

Digrafos acíclicos também são conhecidos como DAGs (= *directed acyclic graphs*)

Exemplo: um digrafo acíclico

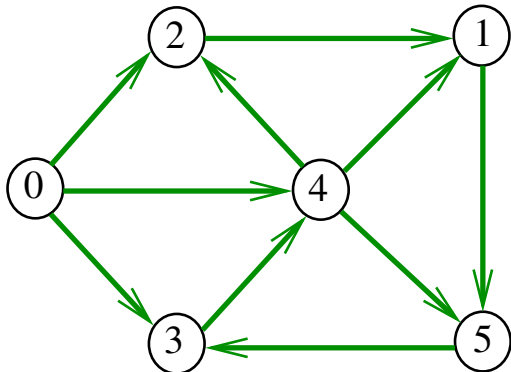


DAGs

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

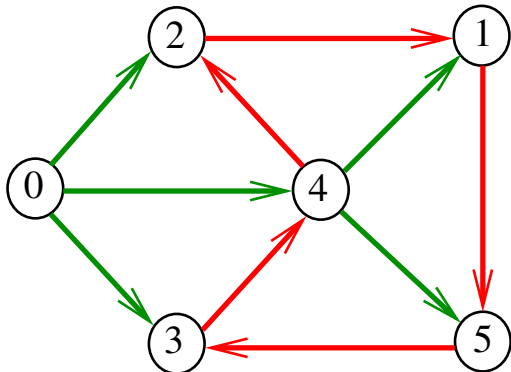


DAGs

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



Ordenação 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

$$ts[0], ts[1], \dots, ts[V-1]$$

dos seus vértices tal que todo arco tem a forma

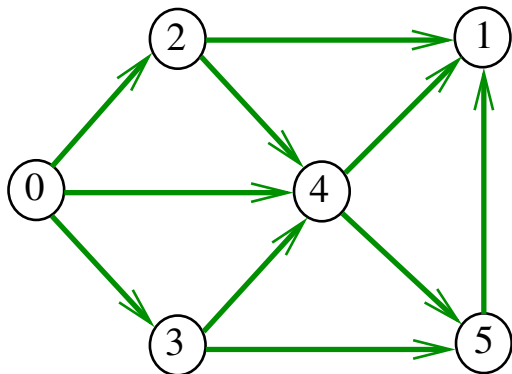
$$ts[i] - ts[j] \text{ com } i < j$$

$ts[0]$ 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



DAGs versus ordenação topológica

É evidente que digrafos com ciclos **não** admitem ordenação topológica.

É menos evidente que **todo** DAG admite ordenação topológica.

A prova desse fato é um algoritmo que recebe qualquer digrafo e devolve

- ▶ um **ciclo**;
- ▶ uma **ordenação topológica** do digrafo.

Algoritmos de ordenação topológica

S 19.6

Algoritmo de eliminação de fontes

Armazena em $ts[0 \dots i-1]$ uma permutação de um subconjunto do conjunto de vértices de G e devolve o valor de i

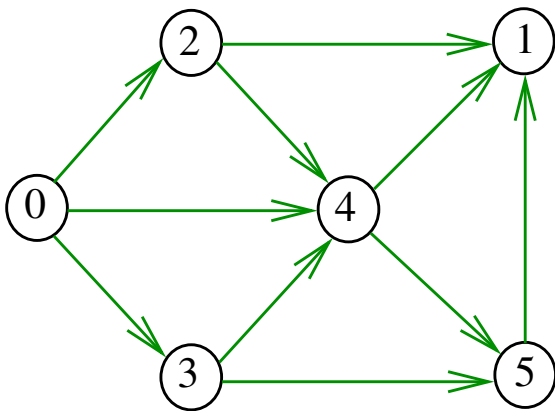
Se $i = G \rightarrow V$ então $ts[0 \dots i-1]$ é uma ordenação topológica de G .

Caso contrário, G **não** é um DAG

```
int DAGts1 (Digraph G, Vertex ts[]);
```

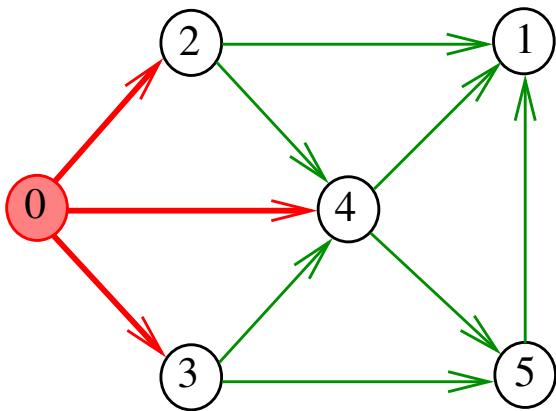

Exemplo

i	0	1	2	3	4	5
ts[i]						



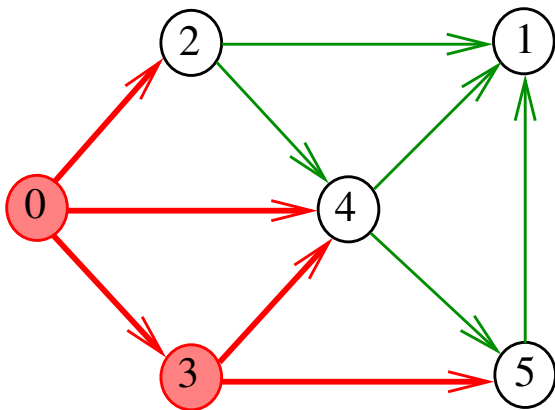
Exemplo

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



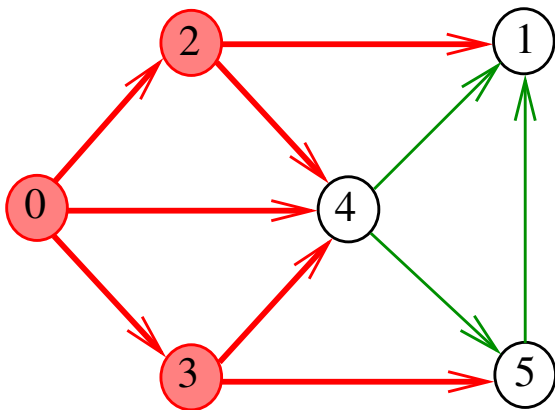
Exemplo

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



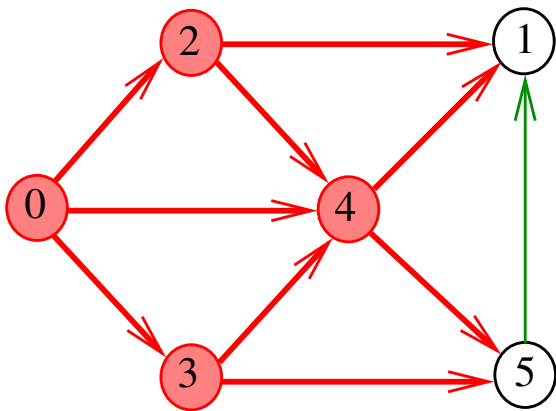
Exemplo

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



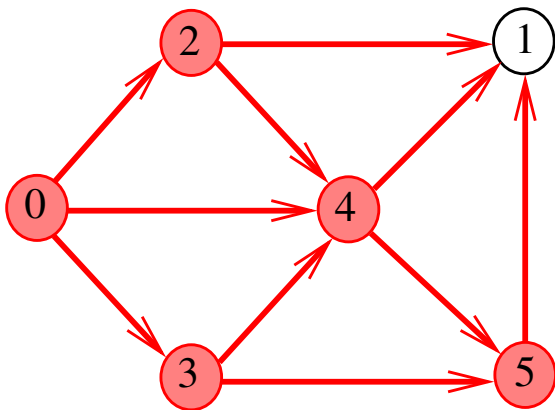
Exemplo

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



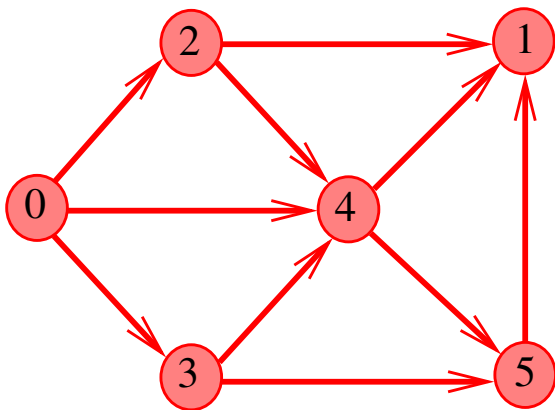
Exemplo

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



Exemplo

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



Consumo de tempo

O consumo de tempo dessa algoritmo para **vetor de listas de adjacência** é $O(V + A)$.

Algoritmos de ordenação topológica

S 19.6

Algoritmo DFS

Recebe um DAG G e armazena em $ts[0 \dots V-1]$
uma ordenação topológica de G

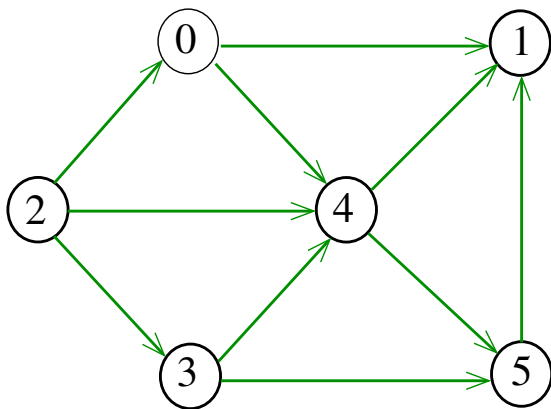
```
public Topological (Digraph  $G$ );
```

ordem topológica dos vértices

```
private Stack<Integer> reversedPost;
```

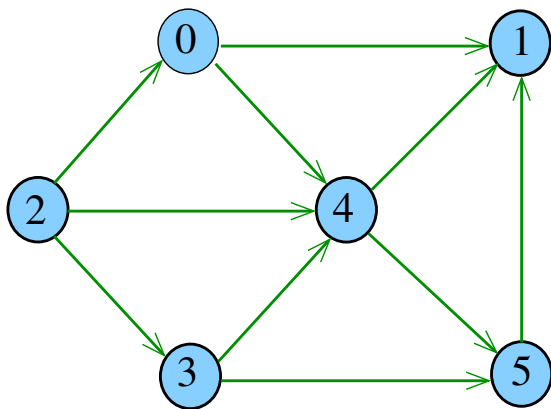
Exemplo

i	0	1	2	3	4	5
ts[i]						



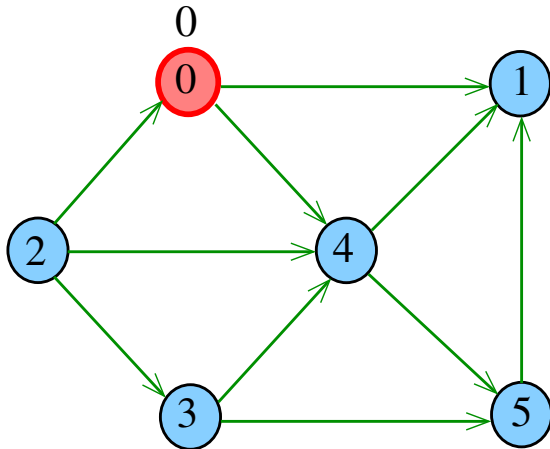
Exemplo

i	0	1	2	3	4	5
ts[i]						



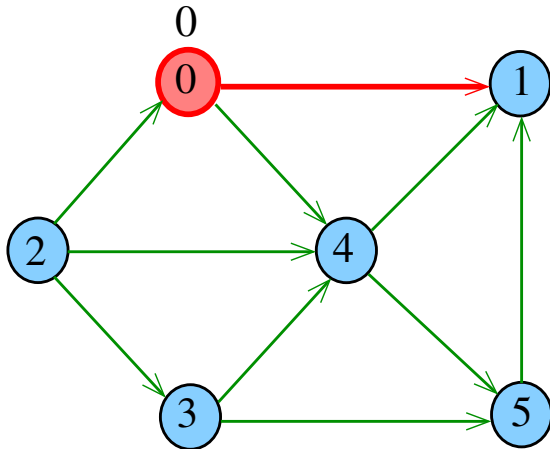
Exemplo

i	0	1	2	3	4	5
ts[i]						



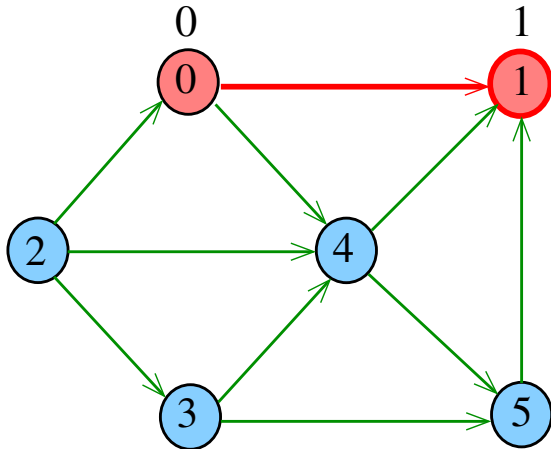
Exemplo

i	0	1	2	3	4	5
ts[i]						



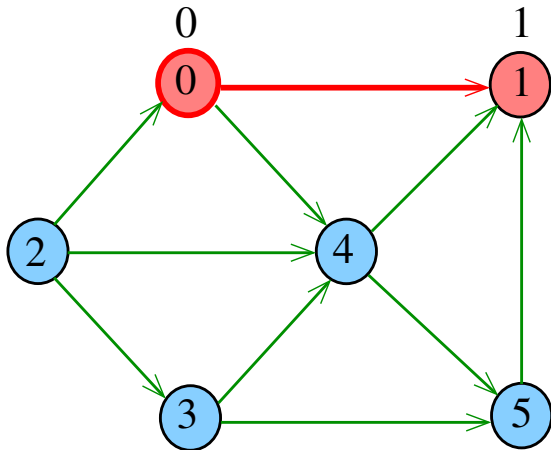
Exemplo

i	0	1	2	3	4	5
ts[i]						



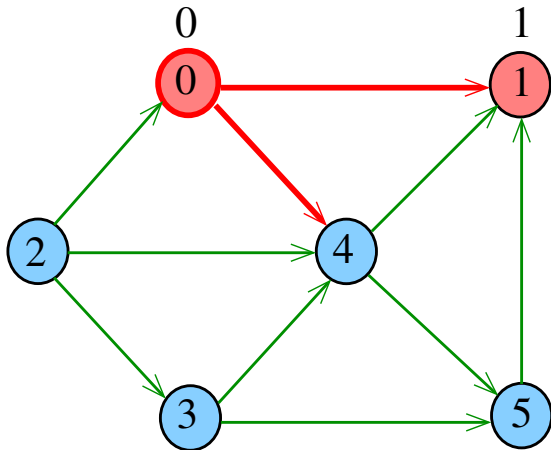
Exemplo

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



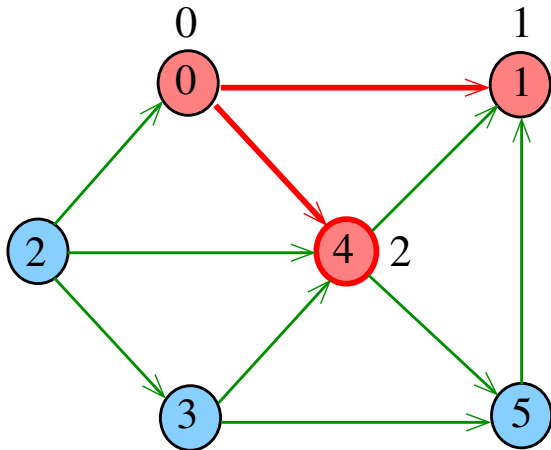
Exemplo

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



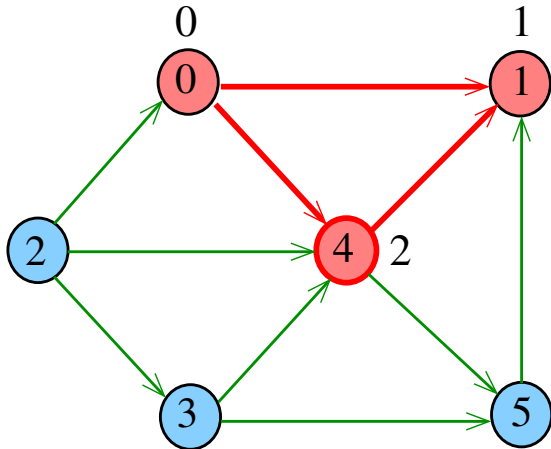
Exemplo

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



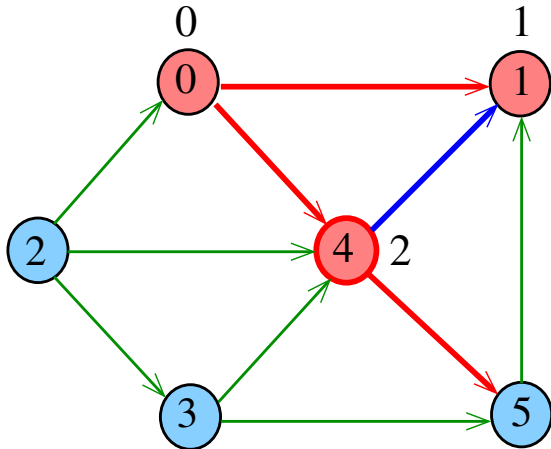
Exemplo

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



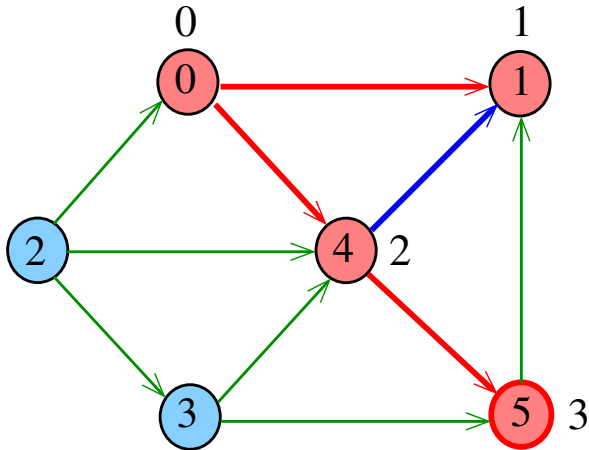
Exemplo

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



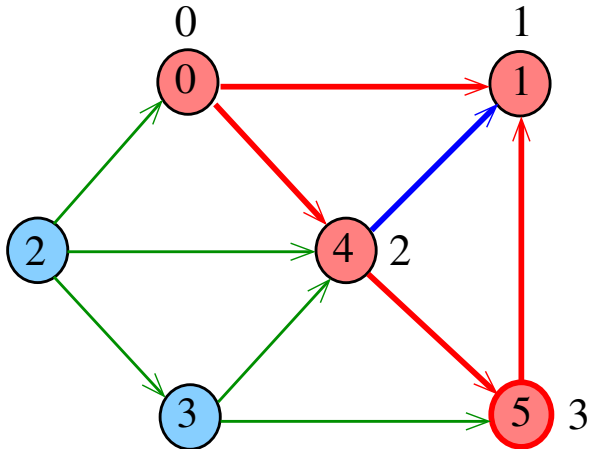
Exemplo

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



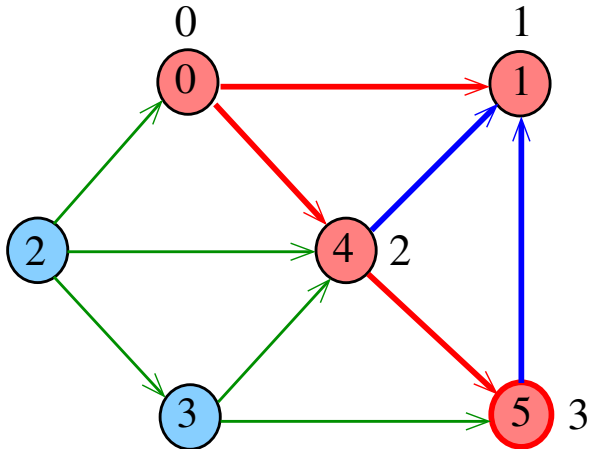
Exemplo

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



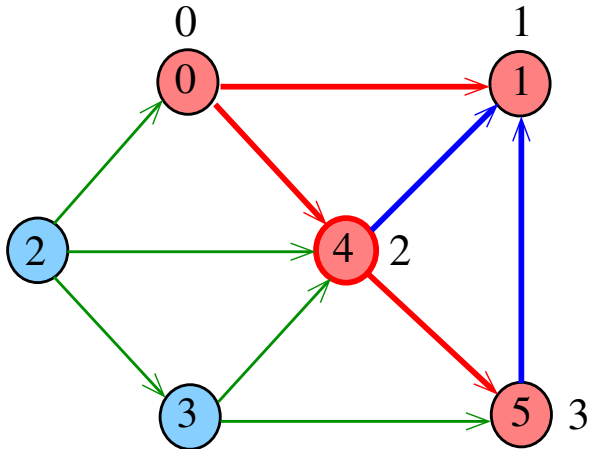
Exemplo

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



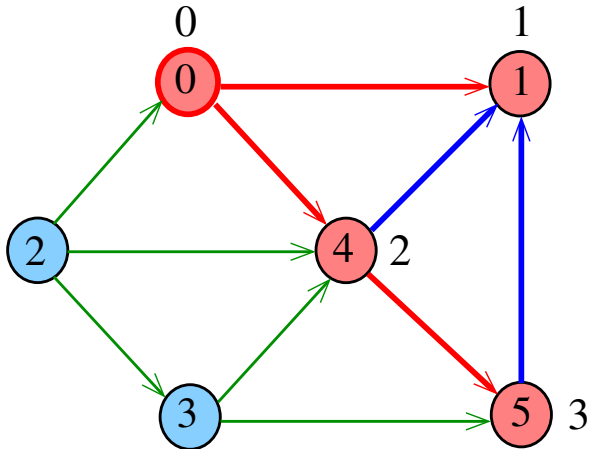
Exemplo

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



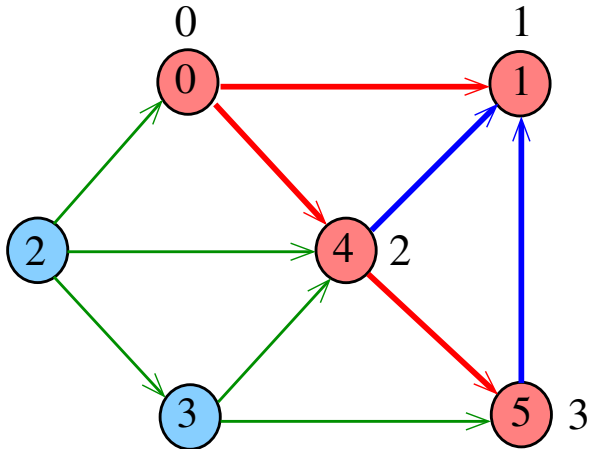
Exemplo

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



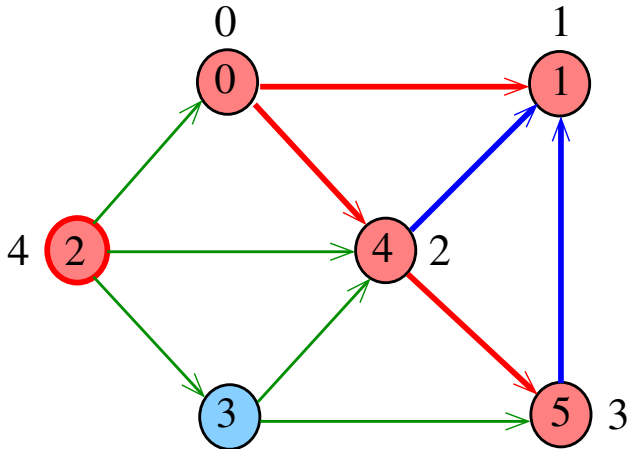
Exemplo

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



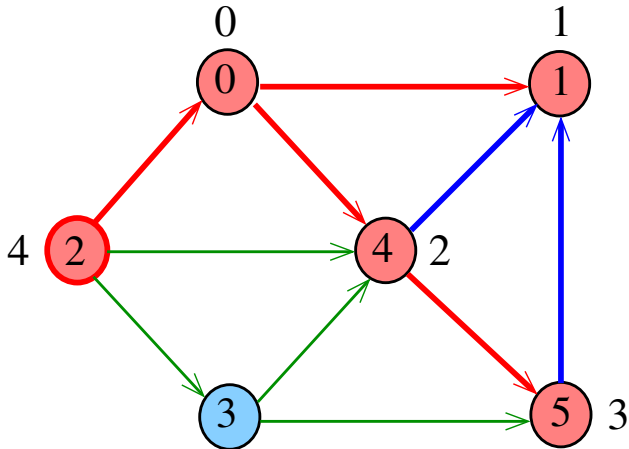
Exemplo

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



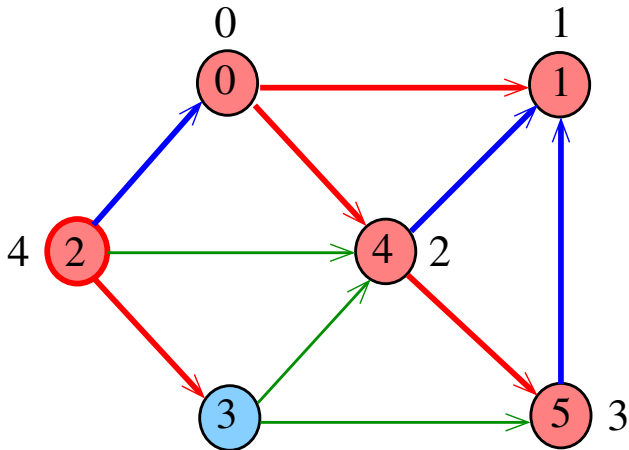
Exemplo

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



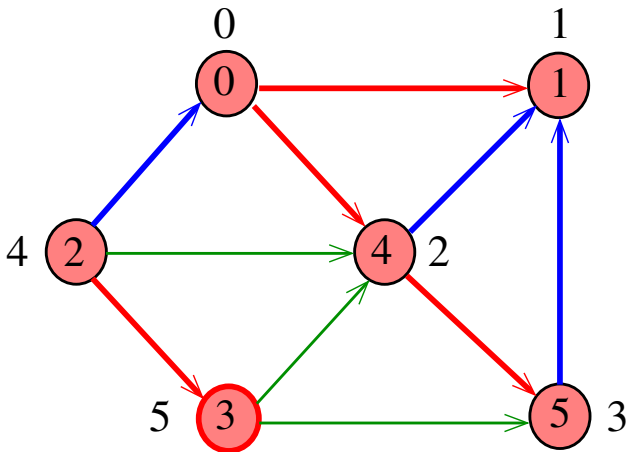
Exemplo

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



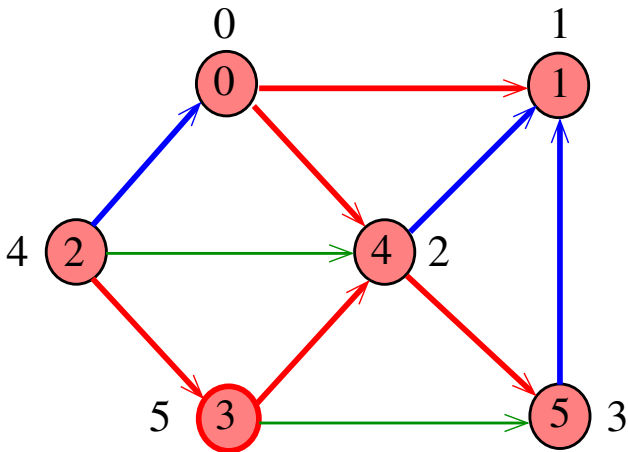
Exemplo

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



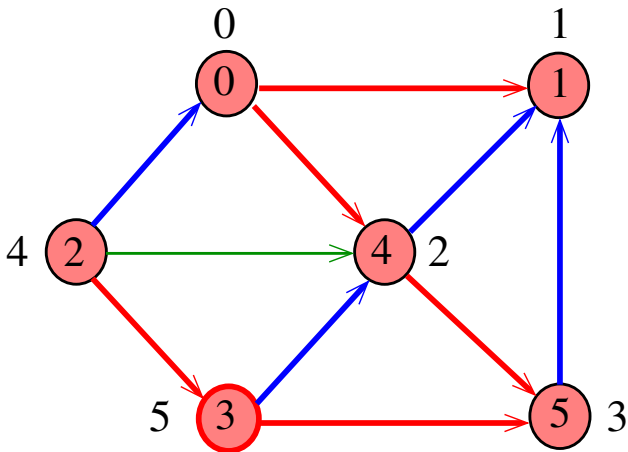
Exemplo

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



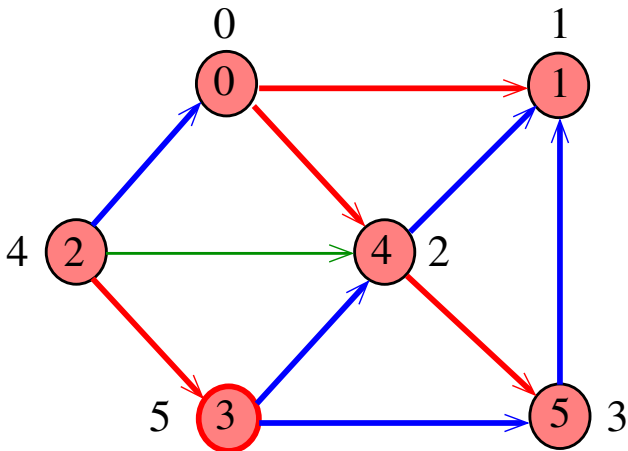
Exemplo

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



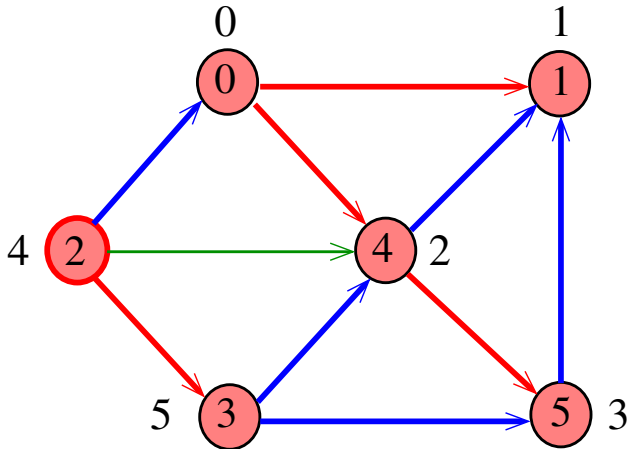
Exemplo

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



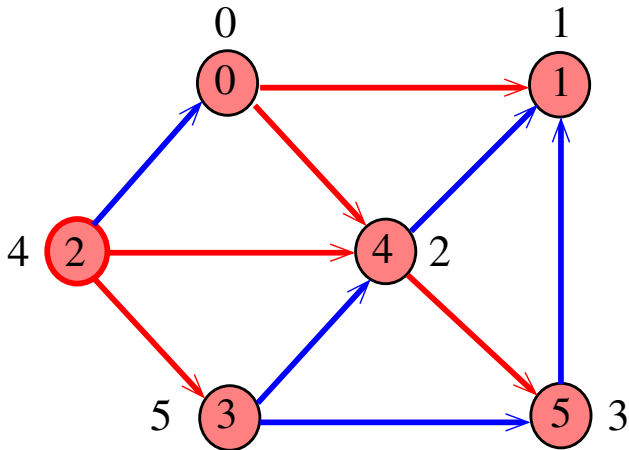
Exemplo

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



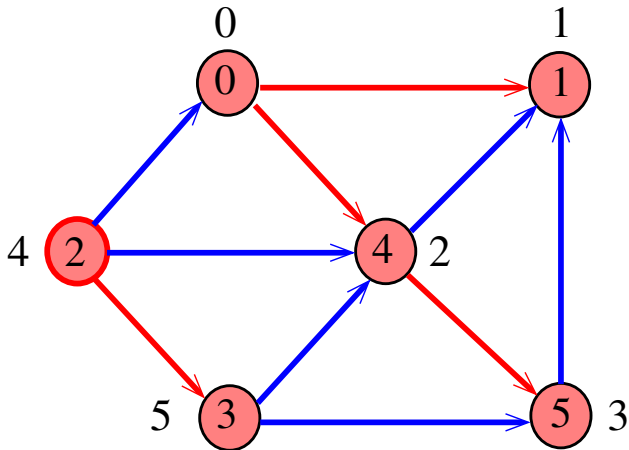
Exemplo

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



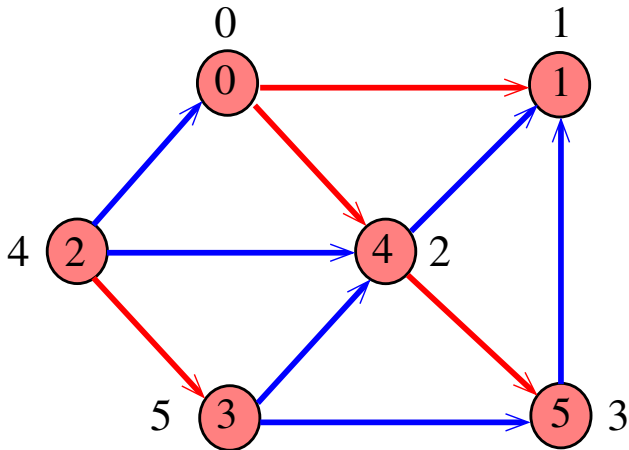
Exemplo

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



Exemplo

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



Consumo de tempo

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

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

Conclusão

Para todo digrafo G , vale uma e apenas uma das seguintes afirmações:

- ▶ G possui um ciclo
- ▶ G é um DAG e, portanto, admite uma ordenação topológica