

Compacto dos melhores momentos

# AULA 20

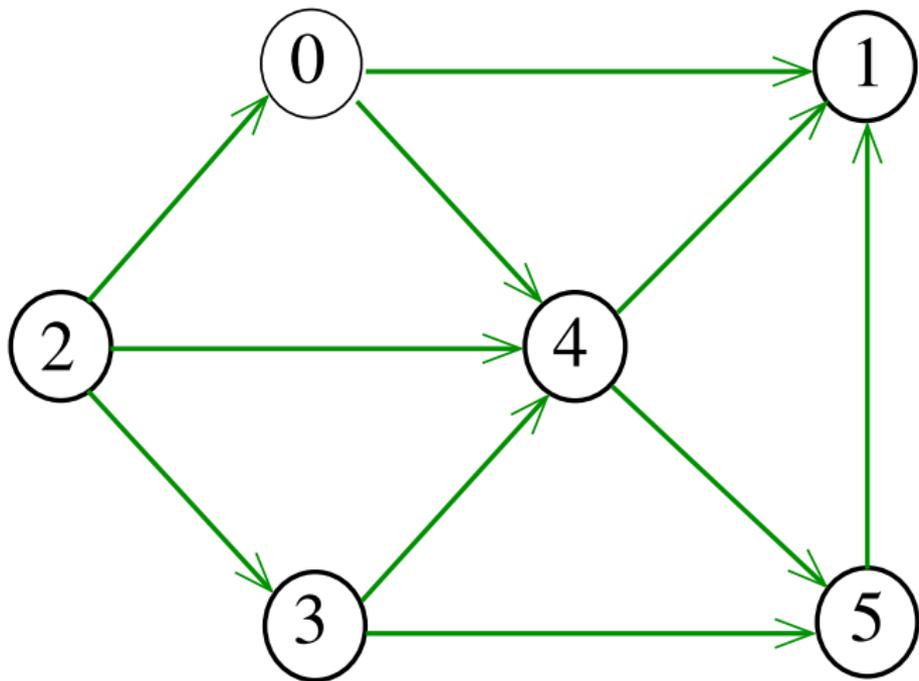
# Busca ou varredura

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

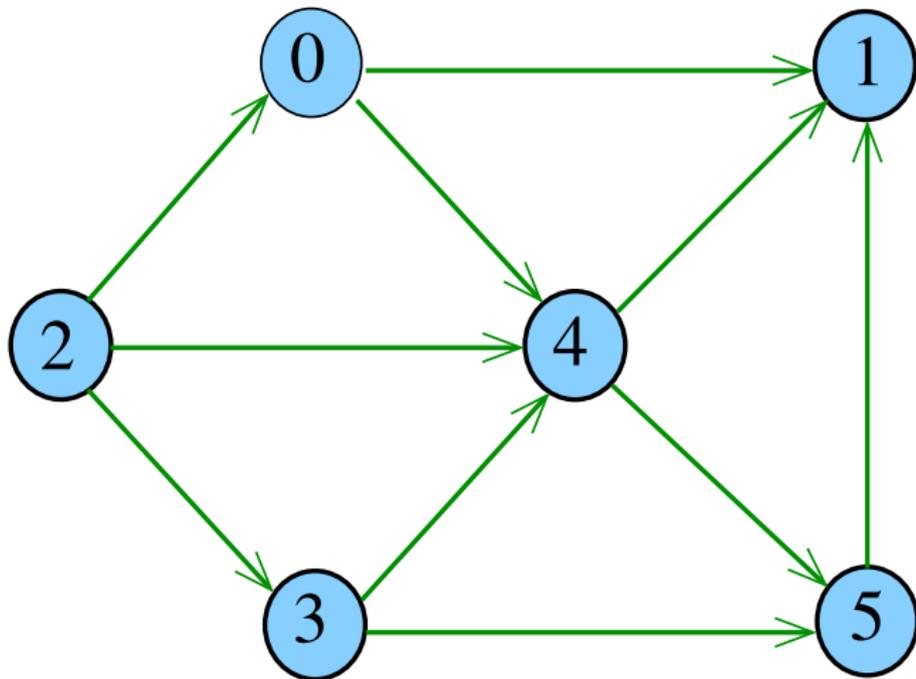
Cada arco é examinado **uma só vez**.

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

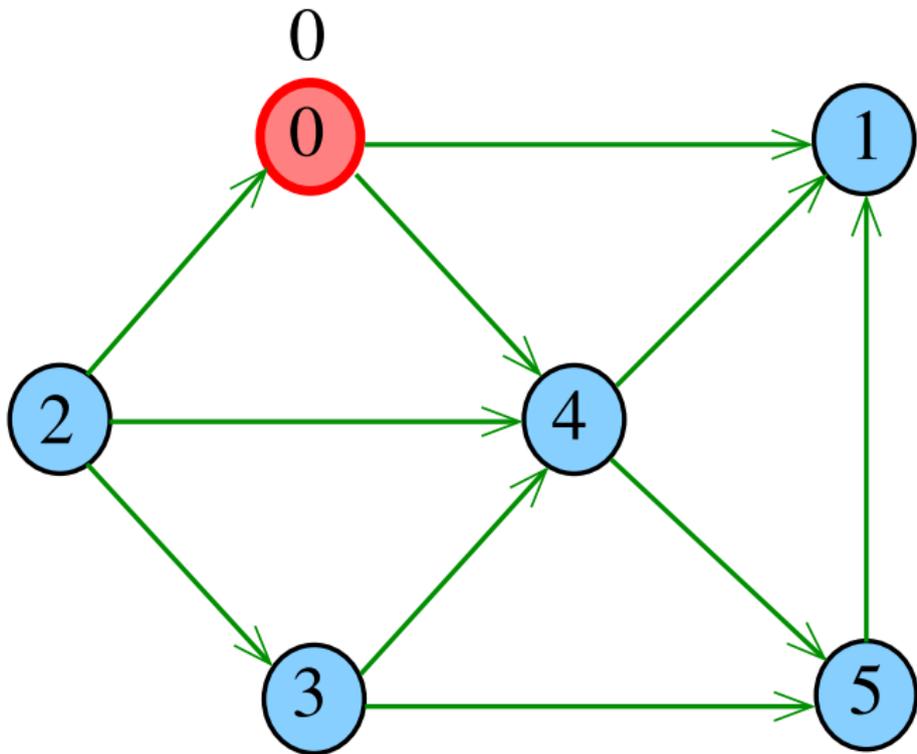
DFS(G)



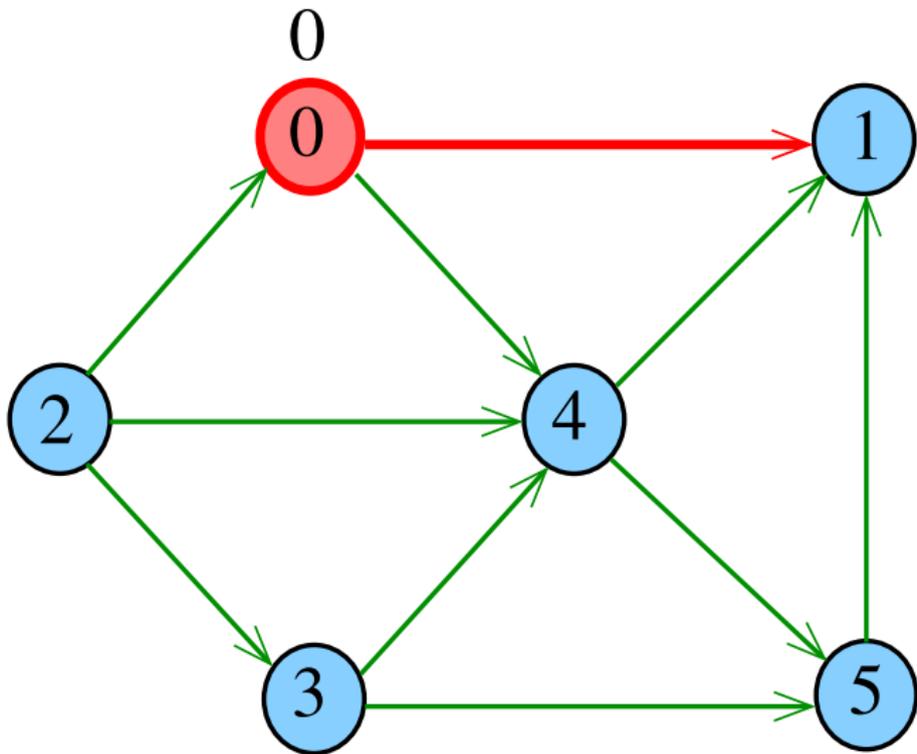
DFS(G)



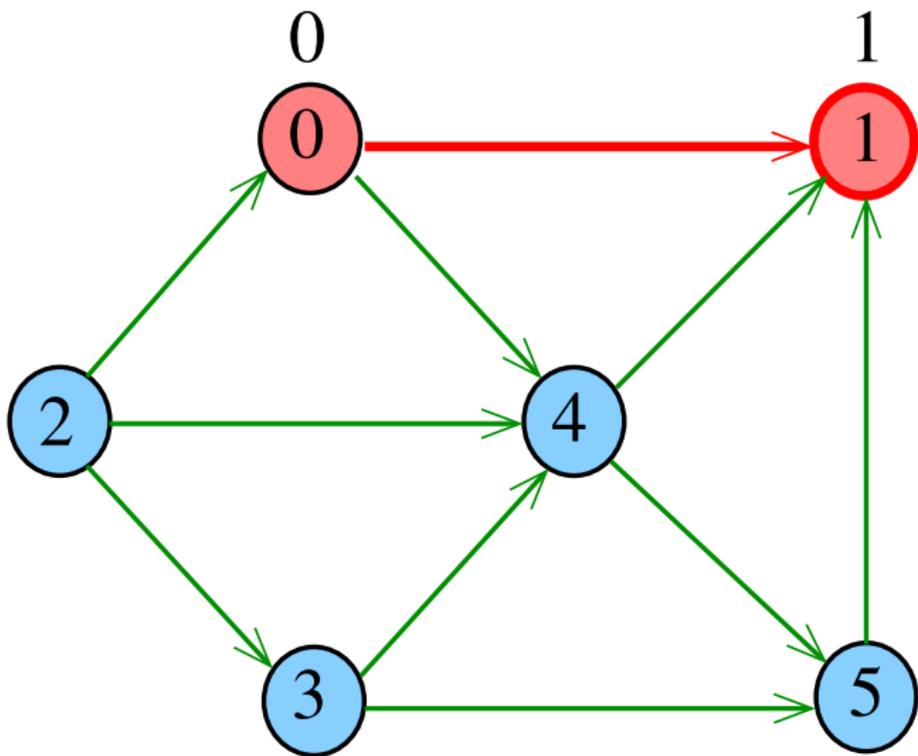
dfs(G,0)



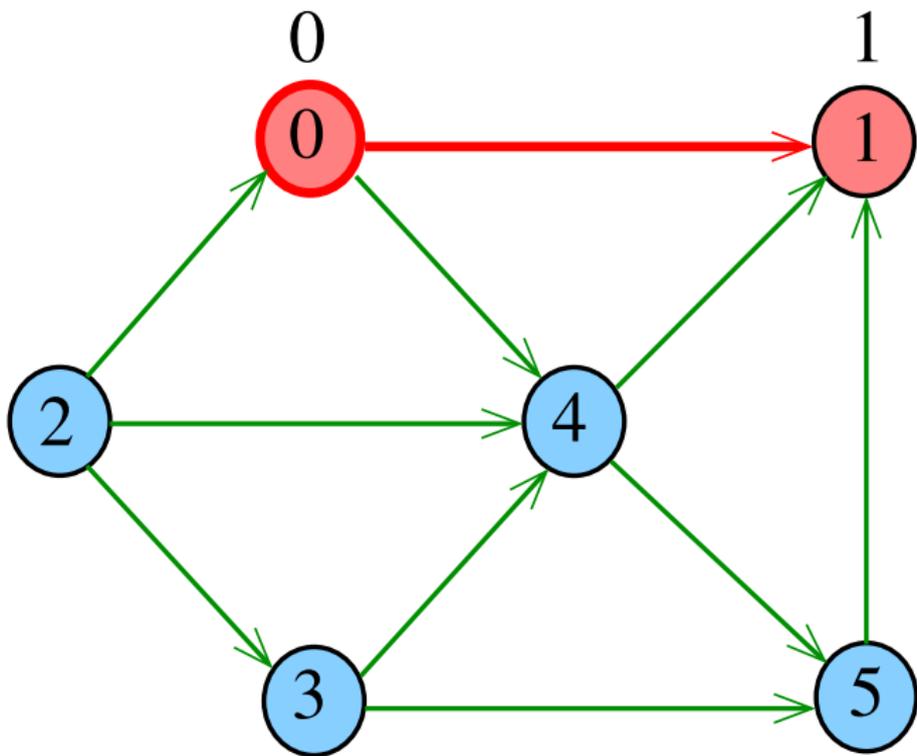
dfs(G,0)



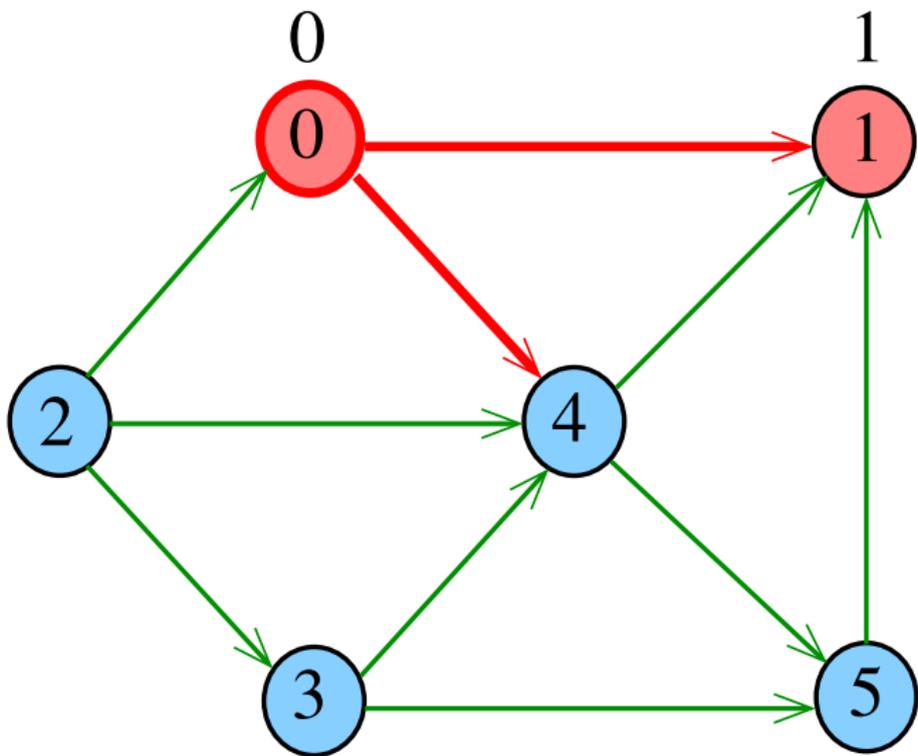
dfs(G,1)



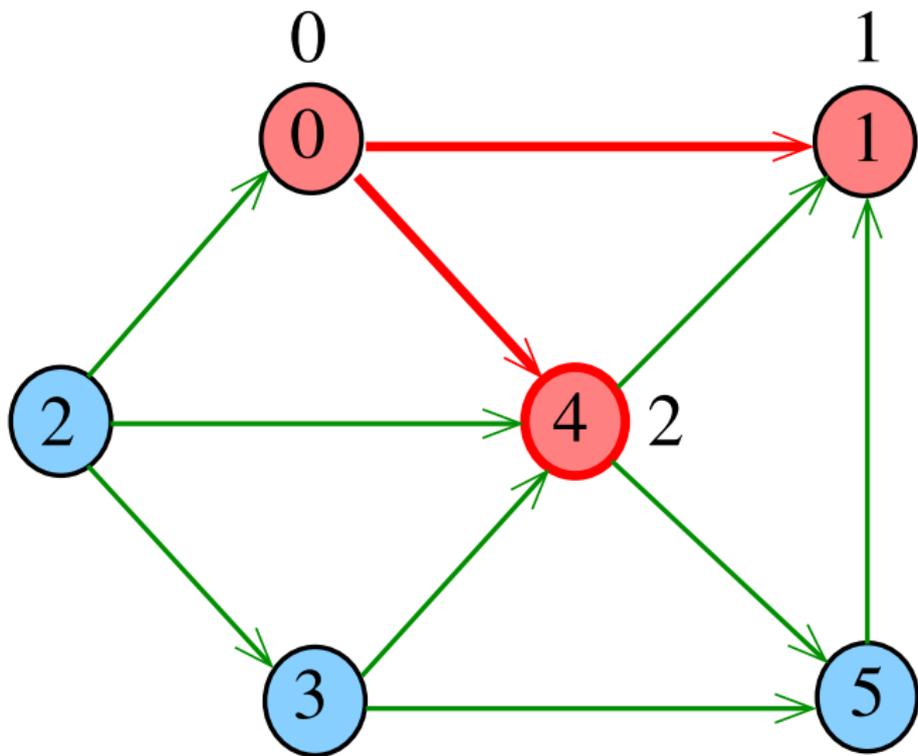
dfs(G,0)



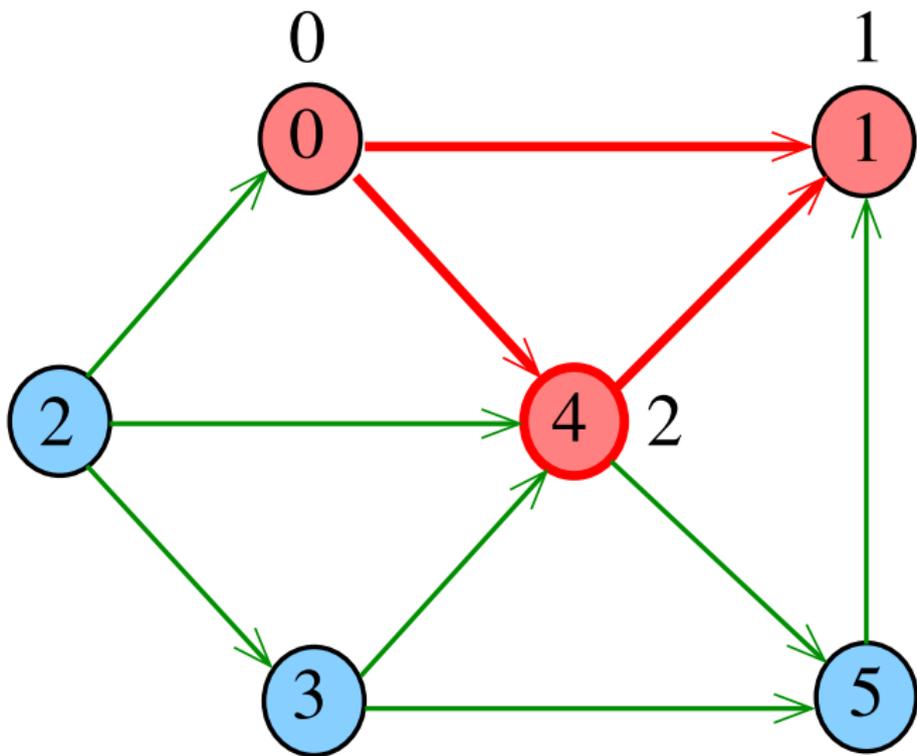
dfs(G,0)



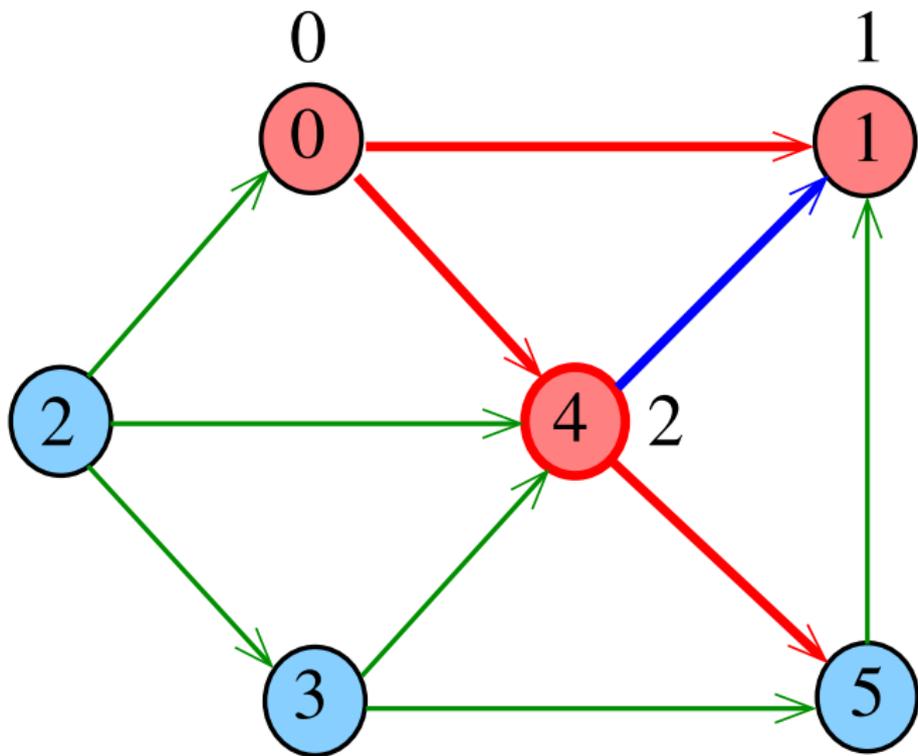
dfs(G,4)



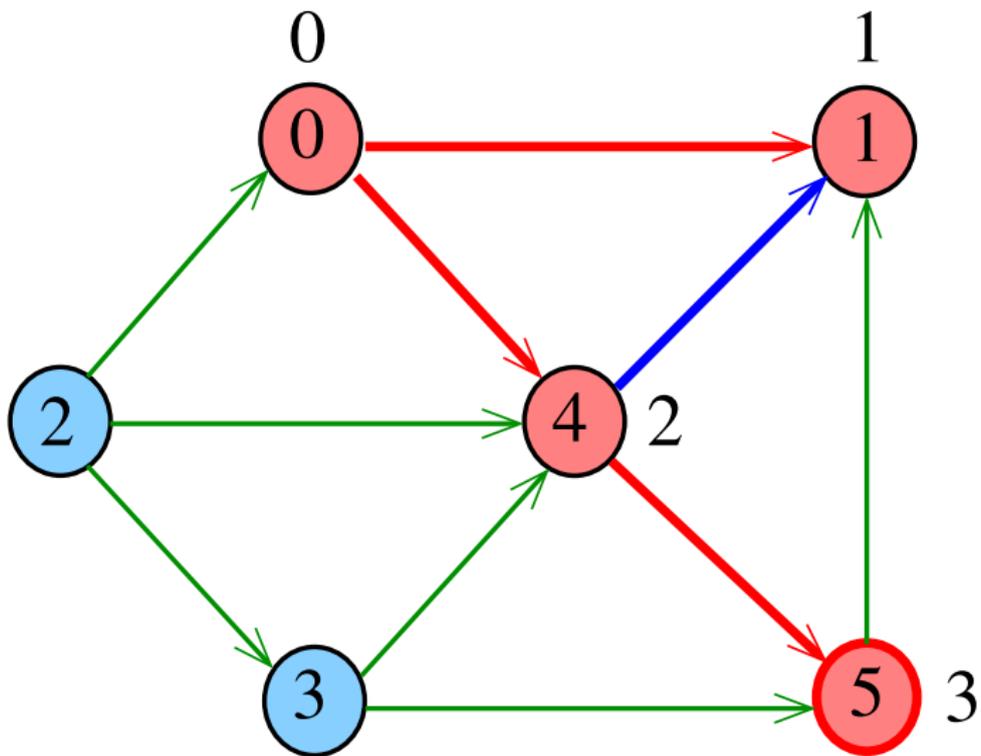
dfs(G,4)



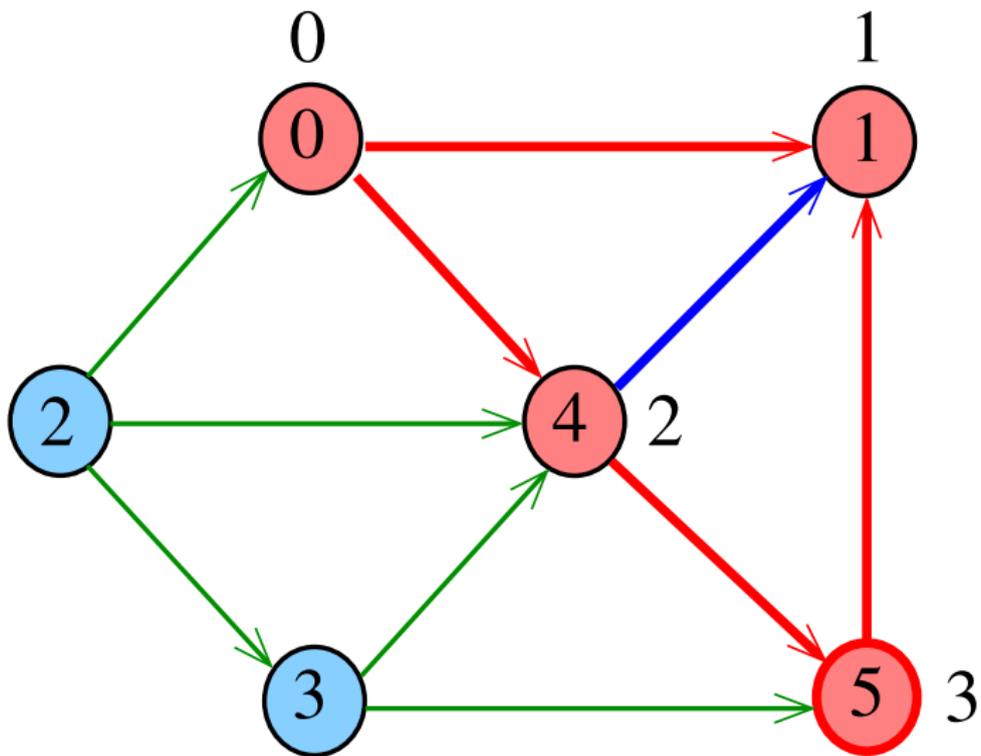
dfs(G,4)



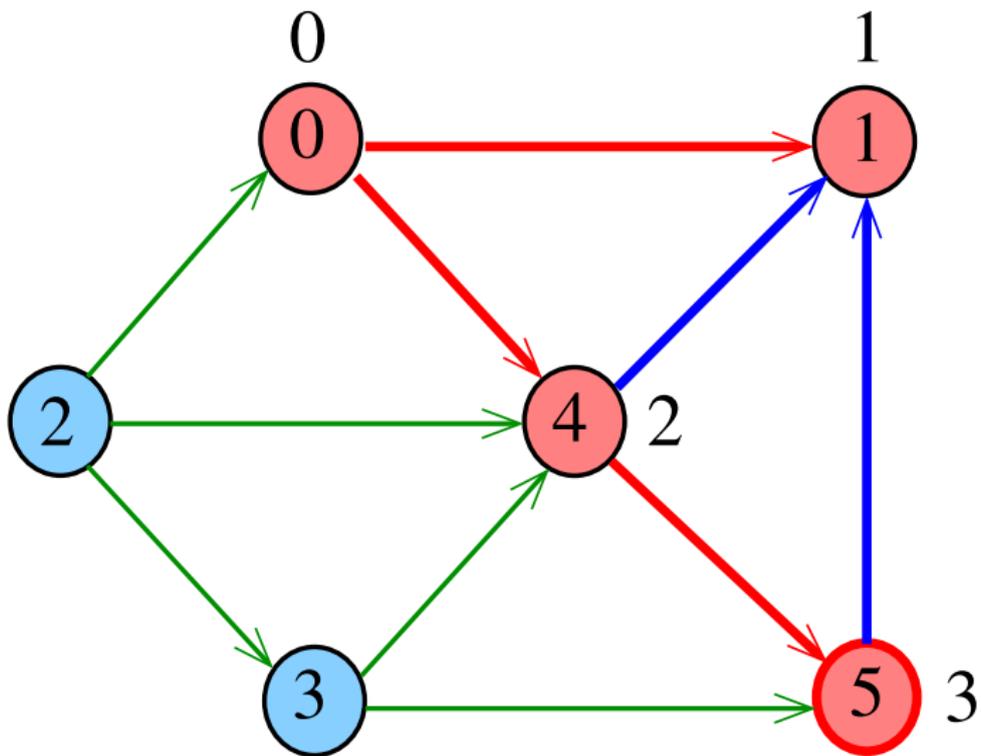
dfs(G,5)



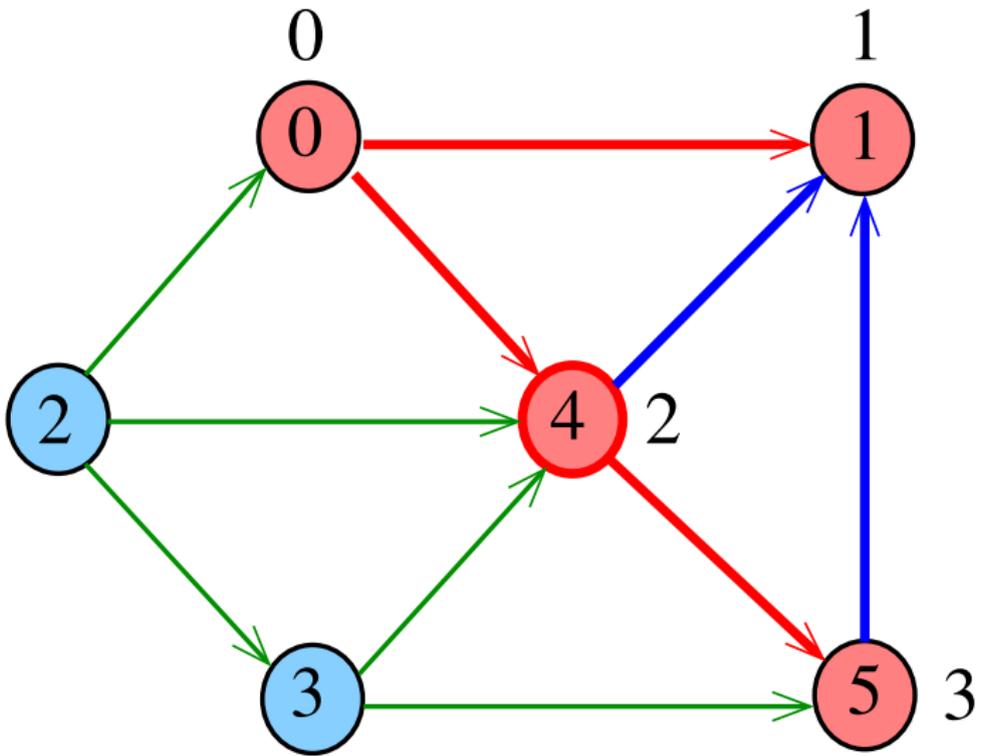
dfs(G,5)



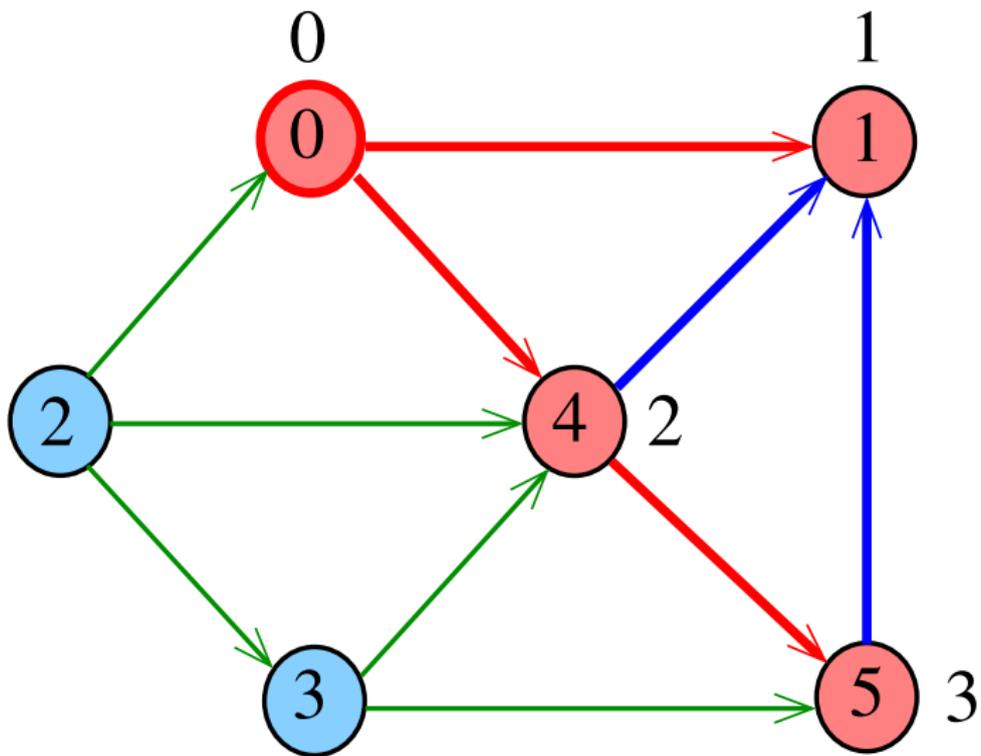
dfs(G,5)



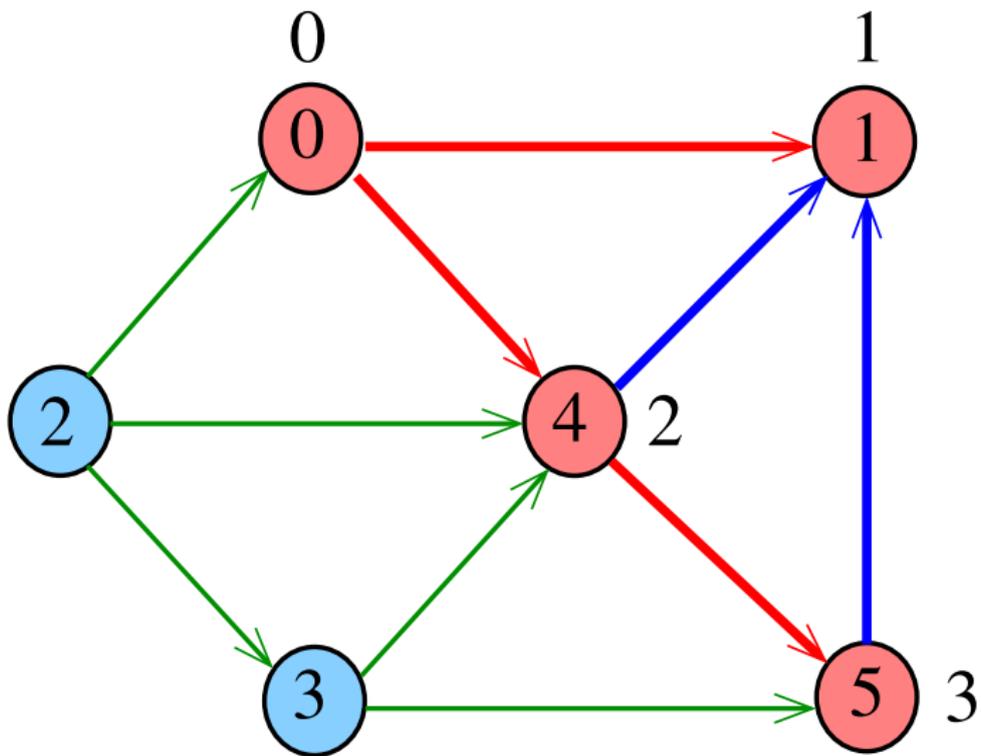
dfs(G,4)



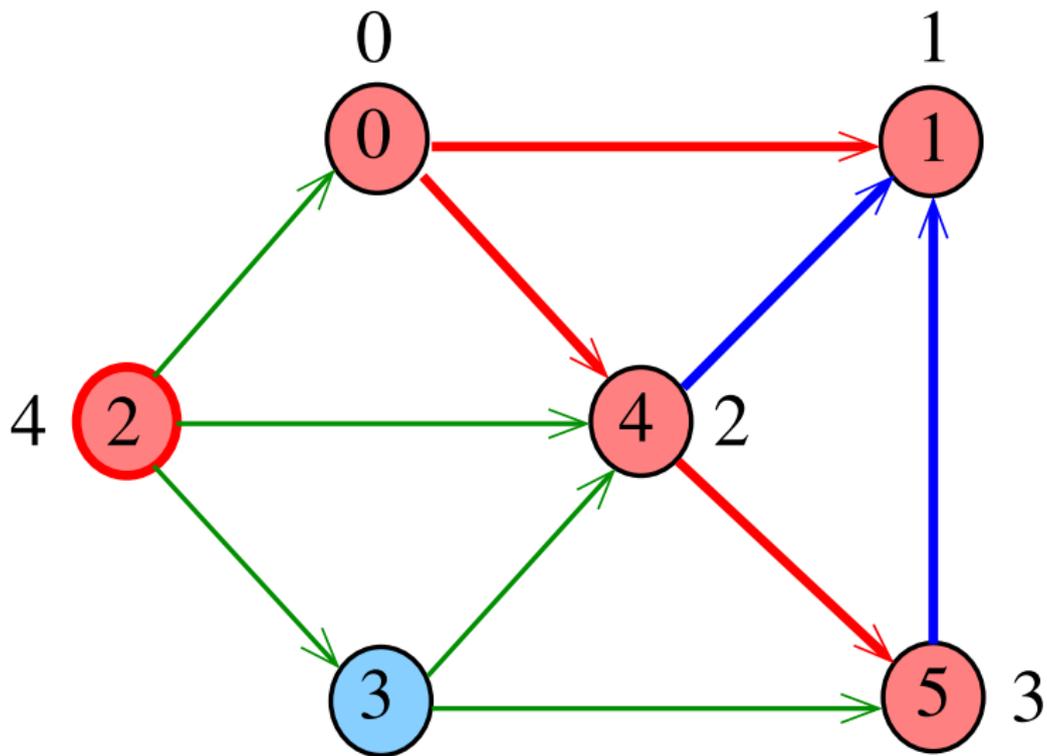
dfs(G,0)



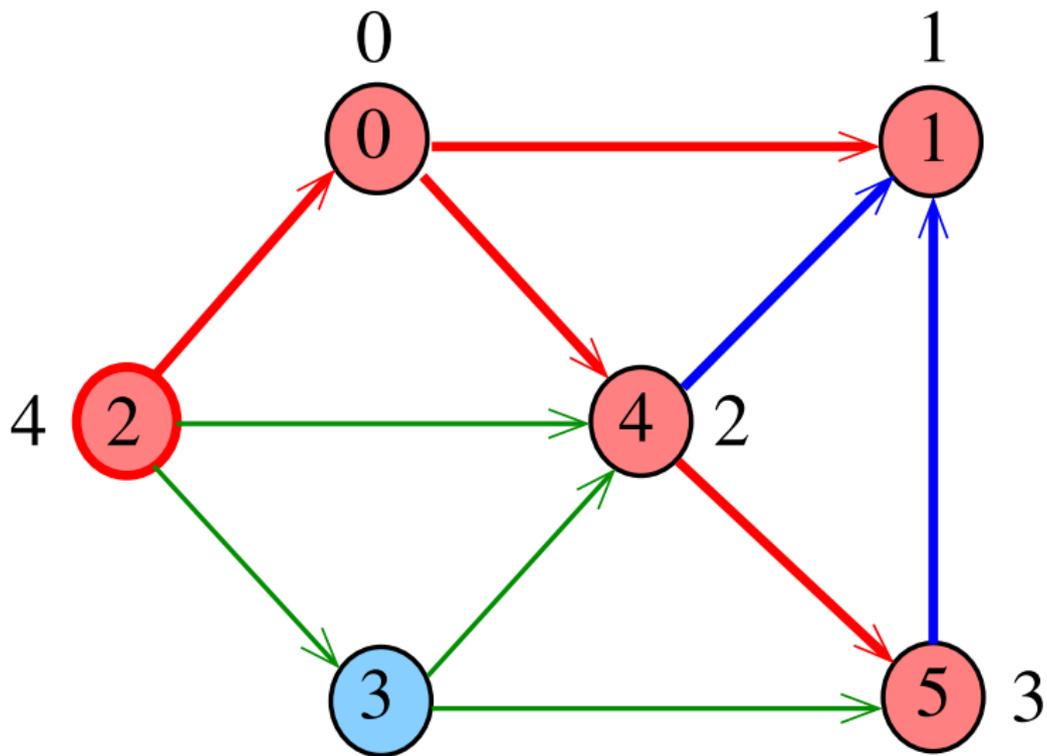
DFS(G)



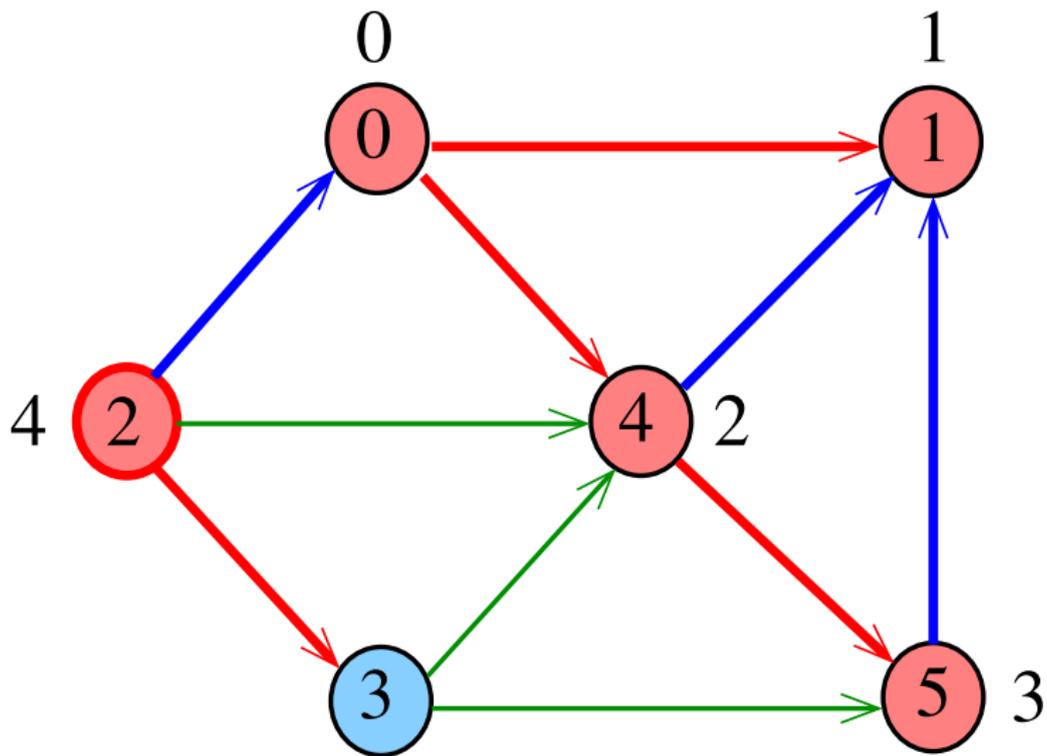
dfs(G,2)



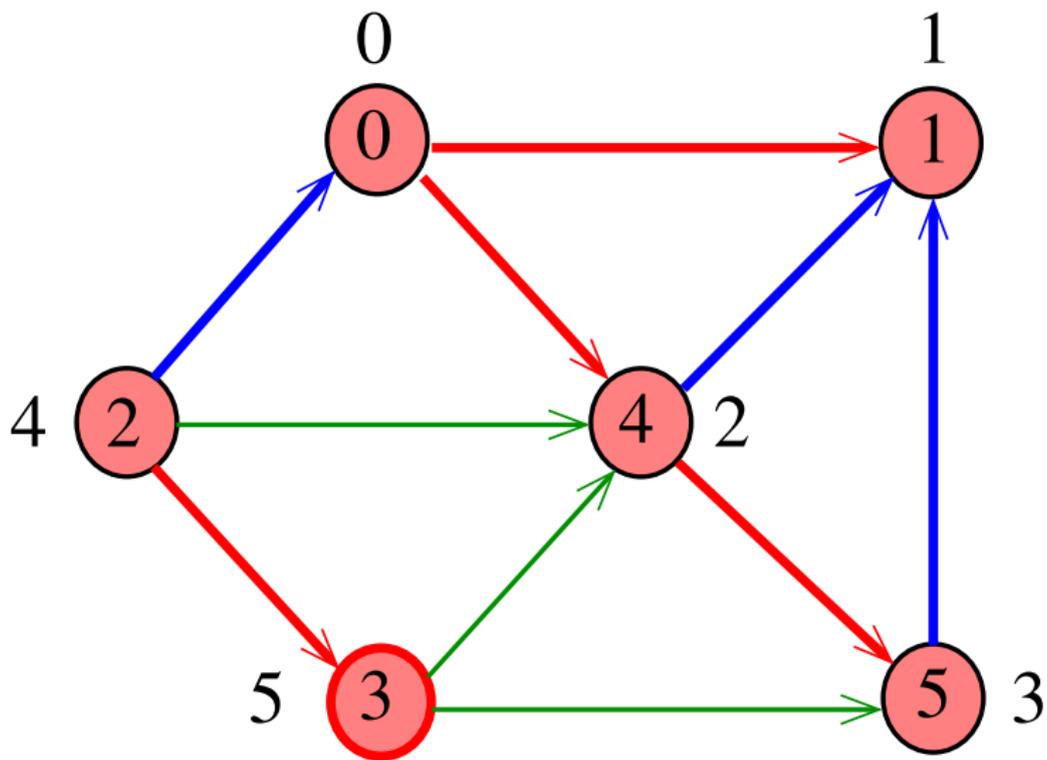
dfs(G,2)



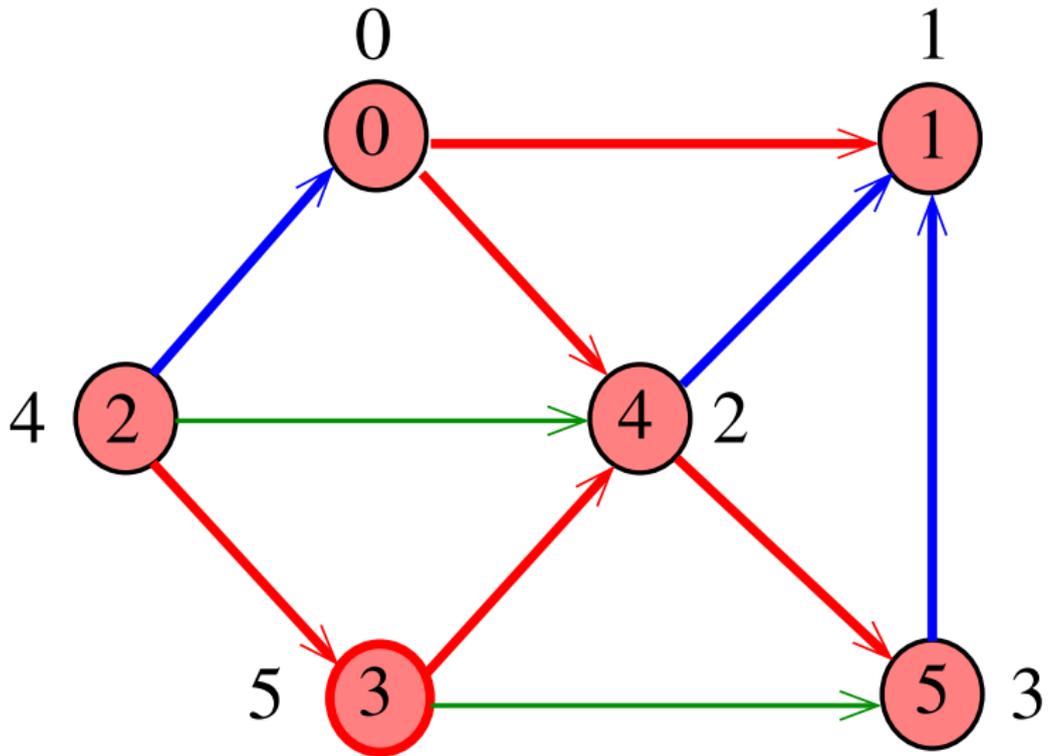
dfs(G,2)



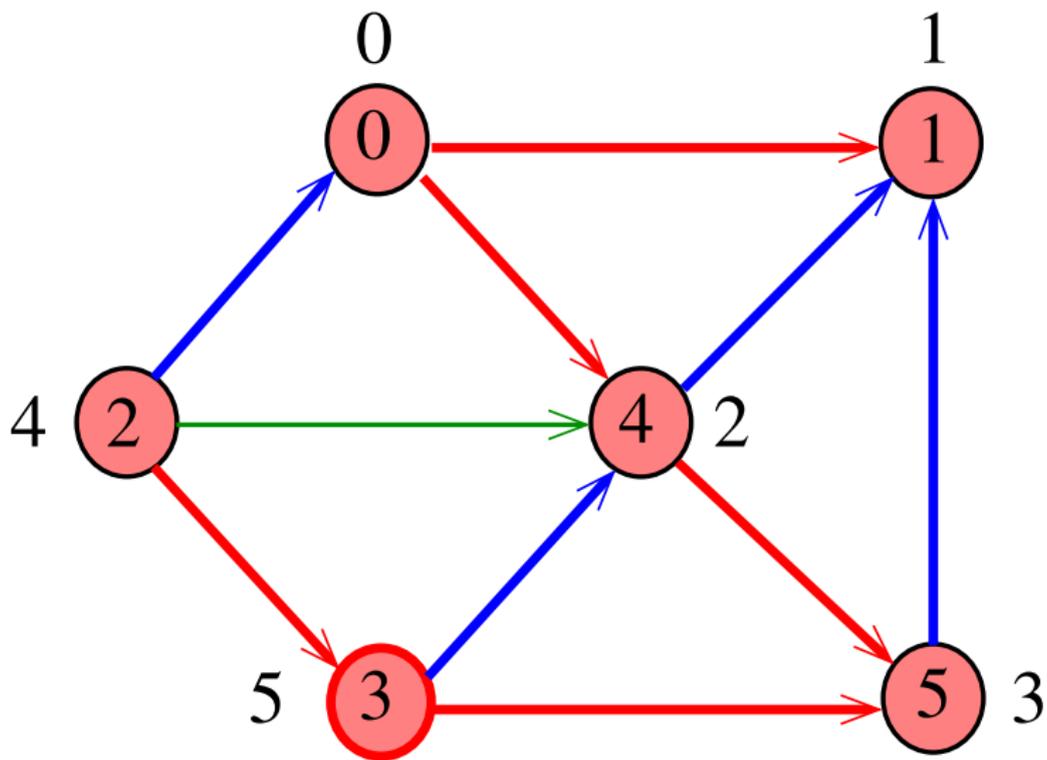
dfs(G,3)



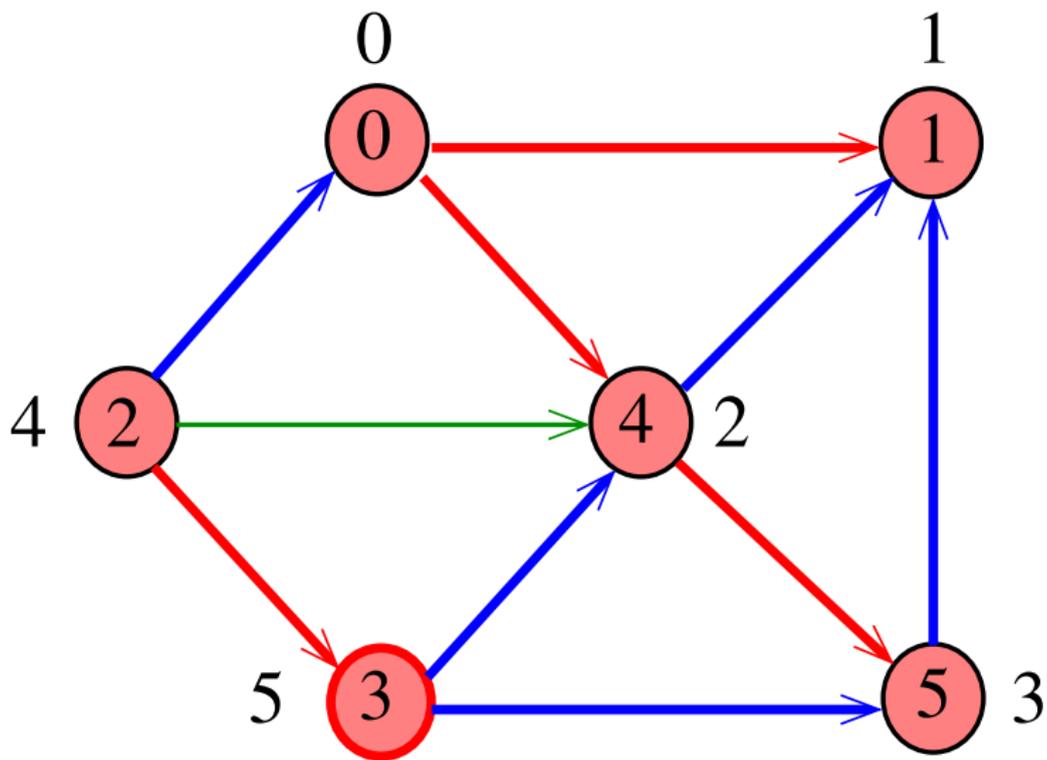
dfs(G,3)



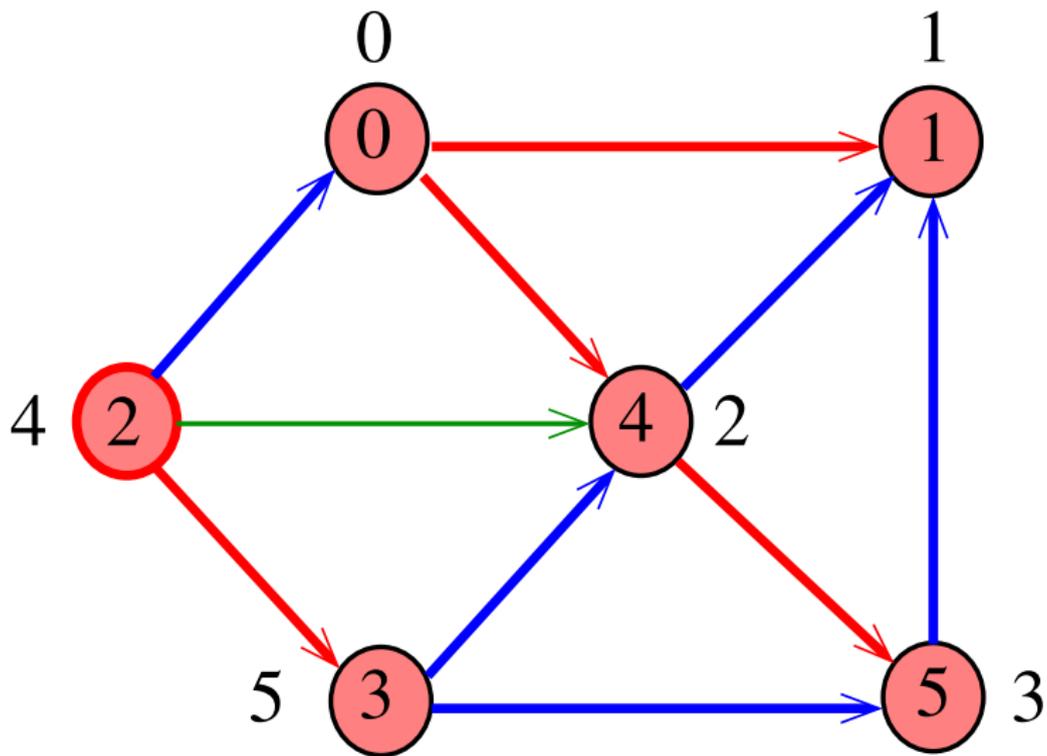
dfs(G,3)



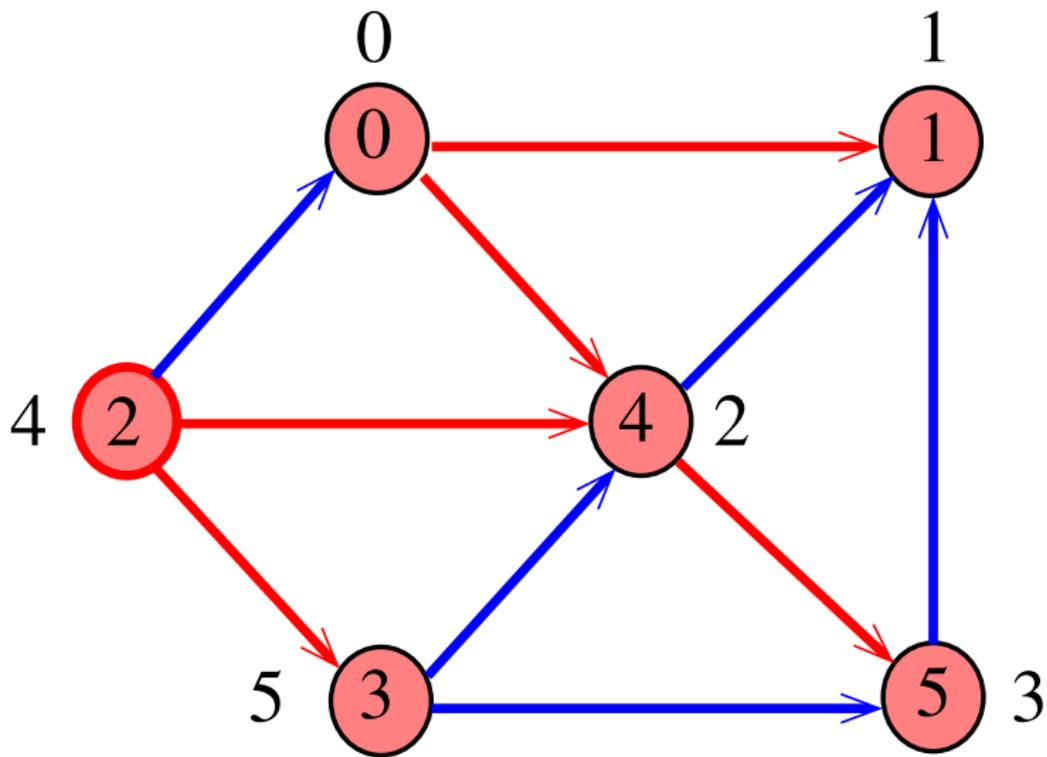
dfs(G,3)



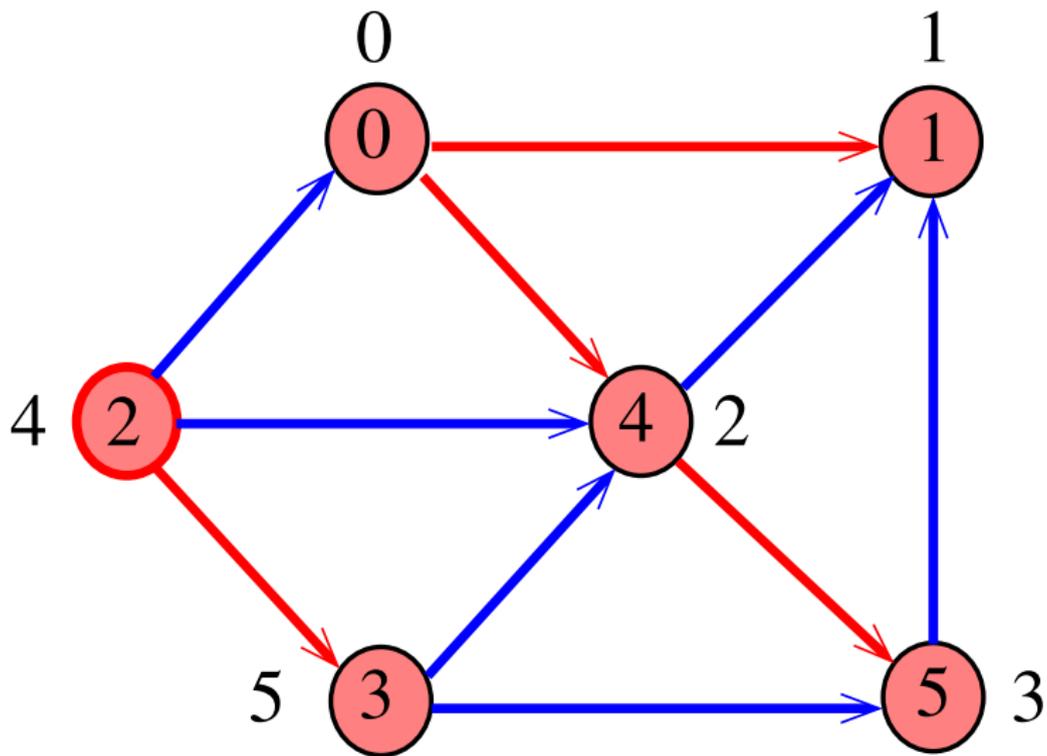
dfs(G,2)



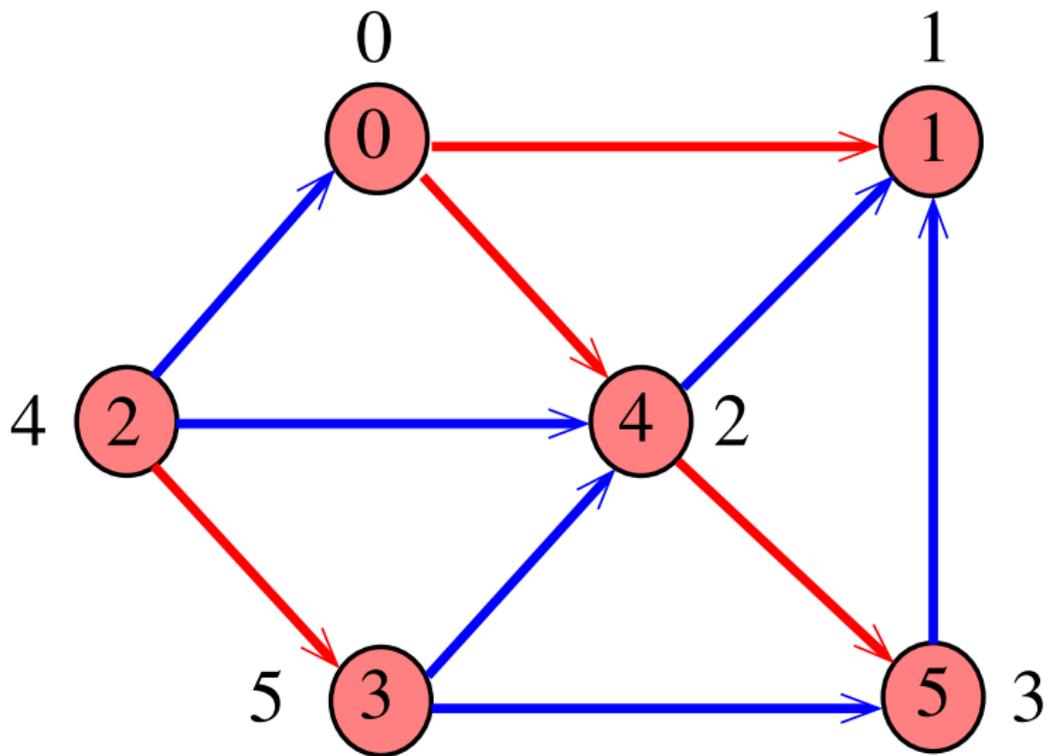
dfs(G,2)



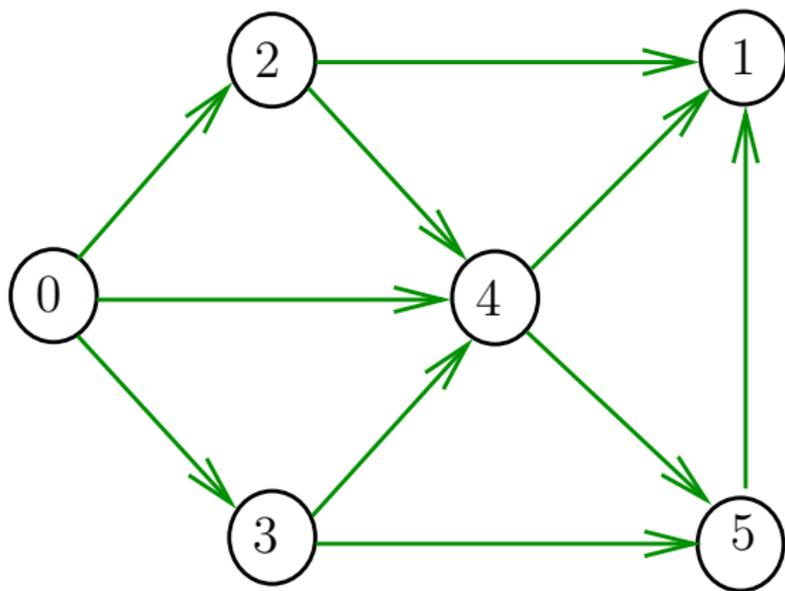
dfs(G,2)



DFS(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

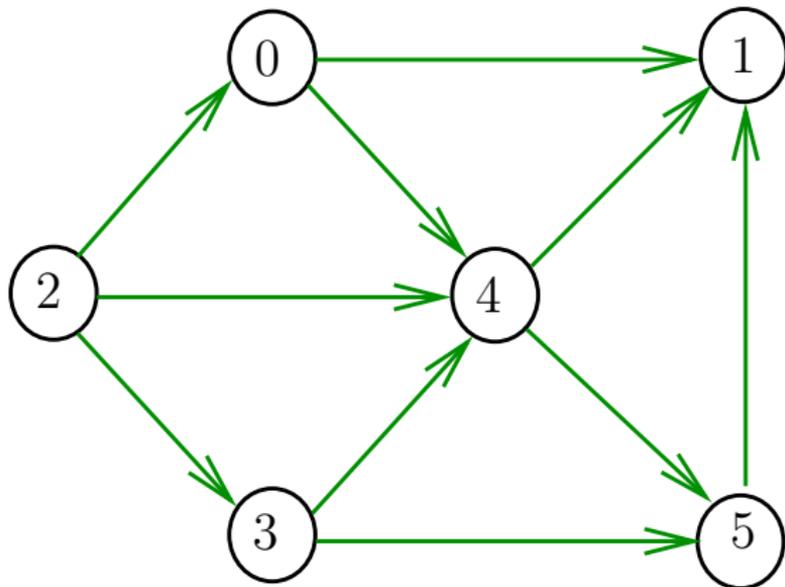
0-3 dfs(G, 3)

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  $\Theta(V^2)$ .

# AULA 21

# Caminhos no computador



Fonte: [Tron Legacy Light Cycle Riders wallpaper](#)

# Caminhos no computador

Como representar **caminhos** no computador?

# Caminhos no computador

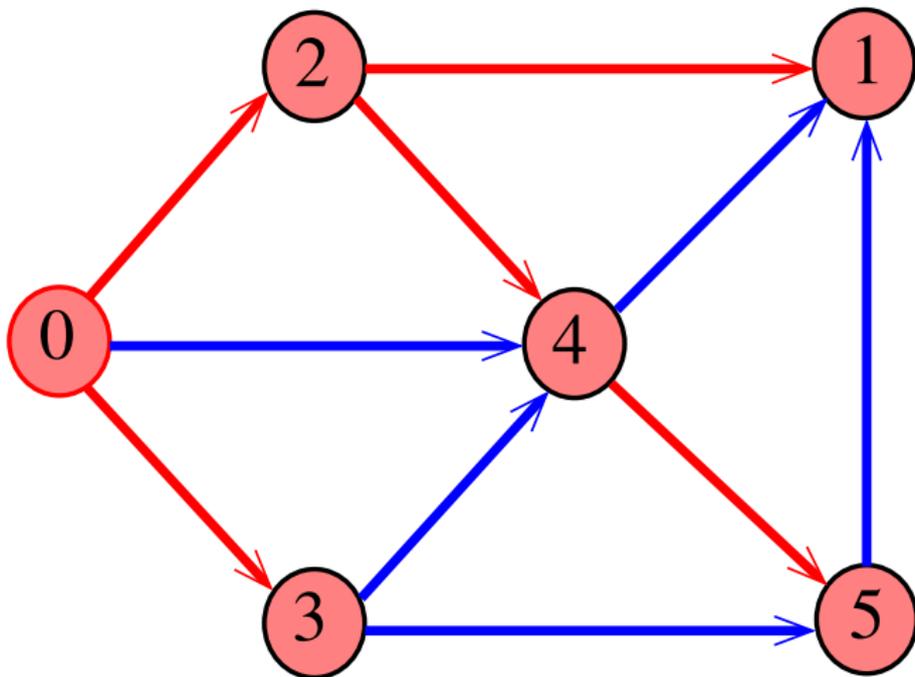
Uma maneira **compacta** de representar **caminhos** de um **vértice** a outros é uma arborescência

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**.

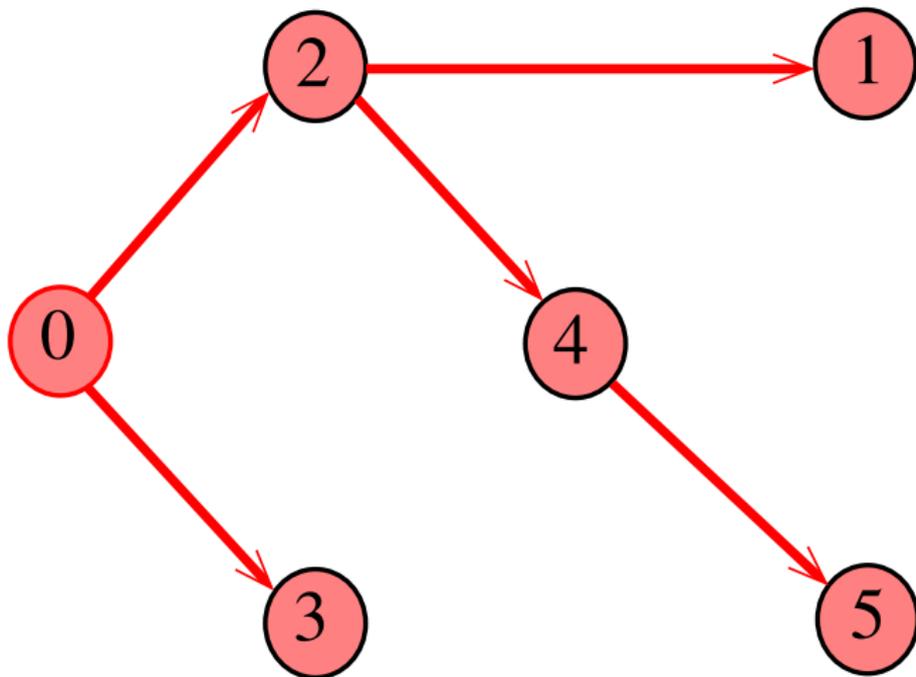
# Arborescências

Exemplo: a raiz da arborescência é 0



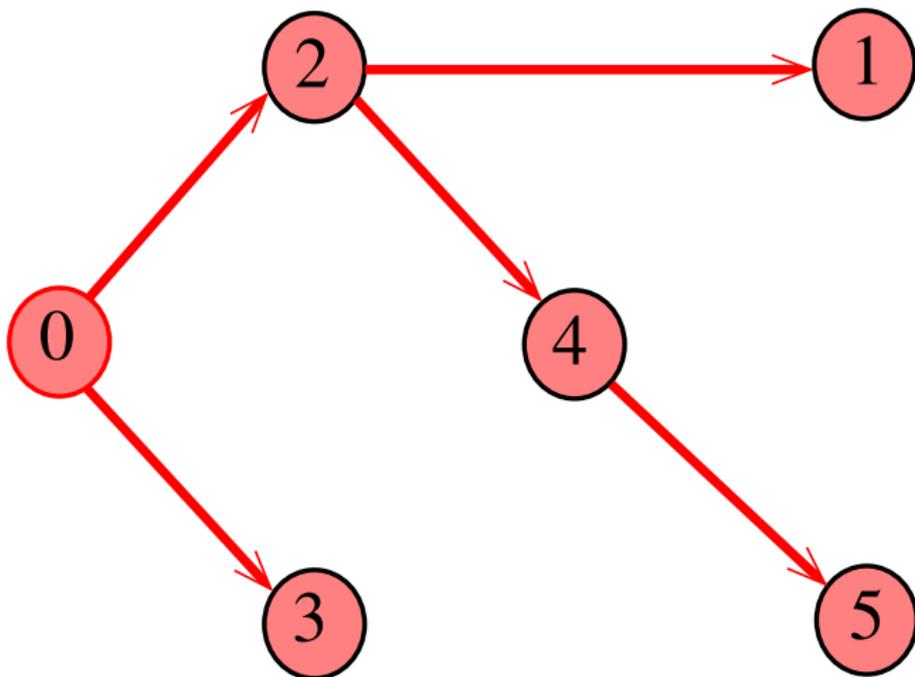
# Arborescências

Exemplo: a raiz da arborescência é 0



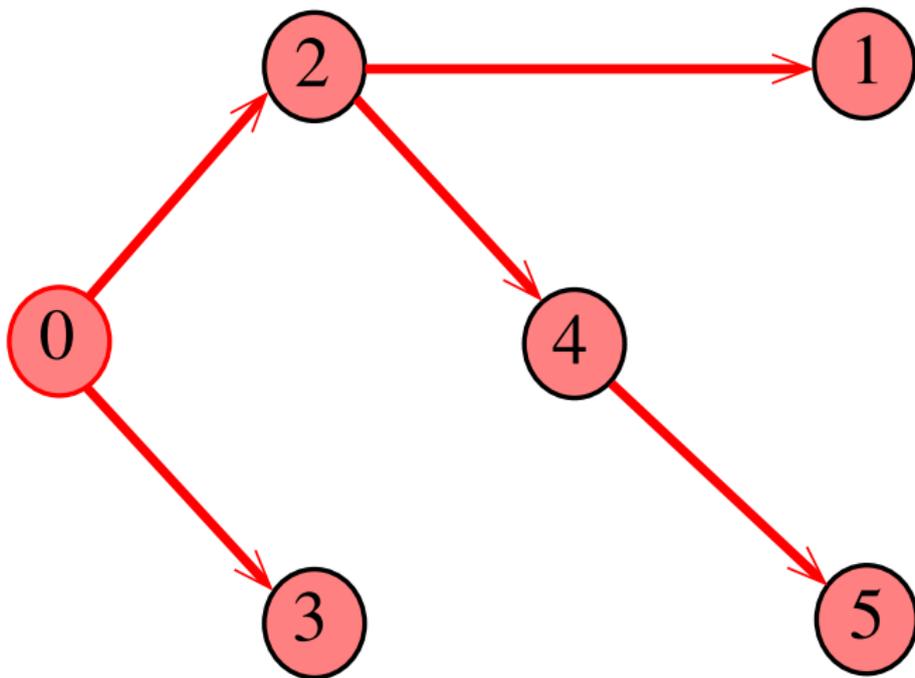
# Arborescências

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



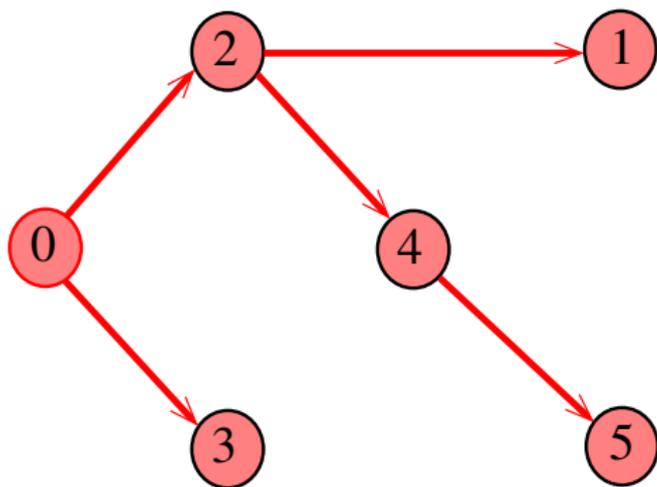
# Arborescências

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



## Arborescências no computador

Um arborescência pode ser representada através de um **vetor de pais**:  $\text{edgeTo}[w]$  é o pai de  $w$   
Se  $r$  é a raiz, então  $\text{edgeTo}[r]=r$



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

# 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])  
    StdOut.printf("%d-", x);  
StdOut.printf("%d", x);
```

## DFSpaths: esqueleto

```
public class DFSpaths {  
    private final int s;  
    private boolean[] marked;  
    private int[] edgeTo;  
    public DFSpaths(Digraph G, int s) {}  
    private void dfs(Digraph G, int v) {}  
    public boolean hasPath(int v) {}  
    public Iterable<Integer> pathTo(int v)  
}
```

## DFSpaths: construtor

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

```
public DFSpaths(Digraph G, int s) {  
    marked = new boolean[G.V()];  
    edgeTo = new int[G.V()];  
    this.s = s;  
    dfs(G, s);  
}
```

## DFSpaths: dfs()

```
private void dfs(Digraph G, int v) {  
    marked[v] = true;  
    for (int w : G.adj(v)) {  
        if (!marked[w]) {  
            edgeTo[w] = v;  
            dfs(G, w);  
        }  
    }  
}
```

## DFSpaths: pathTo()

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

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

# Certificados

**Achievement Certificate**

---

is presented with the 

**Computer Achievement Award**

On the \_\_\_\_\_ Day of \_\_\_\_\_ In the Year \_\_\_\_\_.

Signed,



\_\_\_\_\_

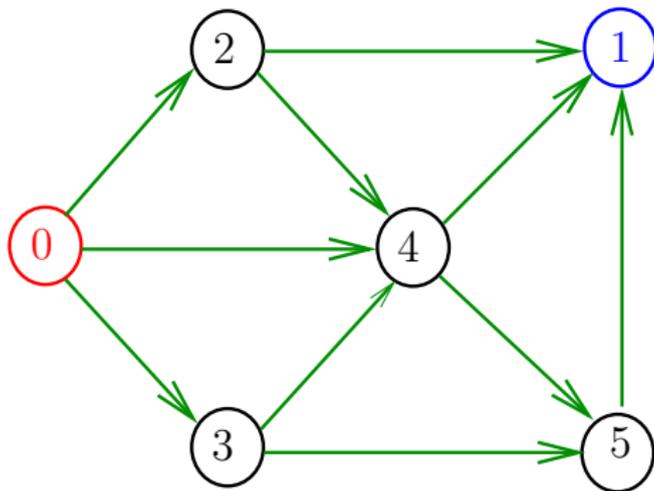
Certificate Provided by [www.hooverwebdesign.com](http://www.hooverwebdesign.com)

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?

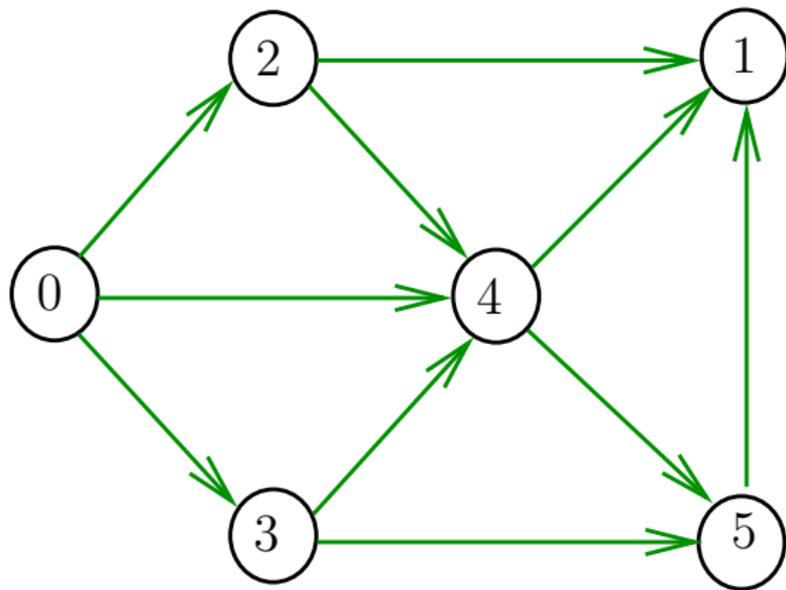
Veremos questões deste tipo freqüentemente

Elas terão um papel **suuupeer** 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**

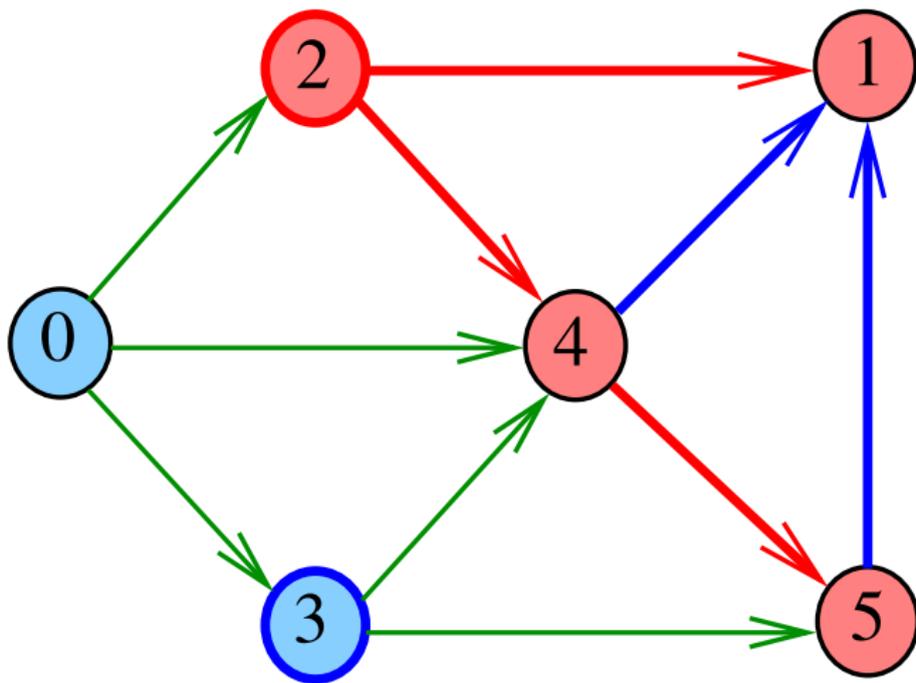
## Certificado de inexistência

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( $G, 2, 3$ )

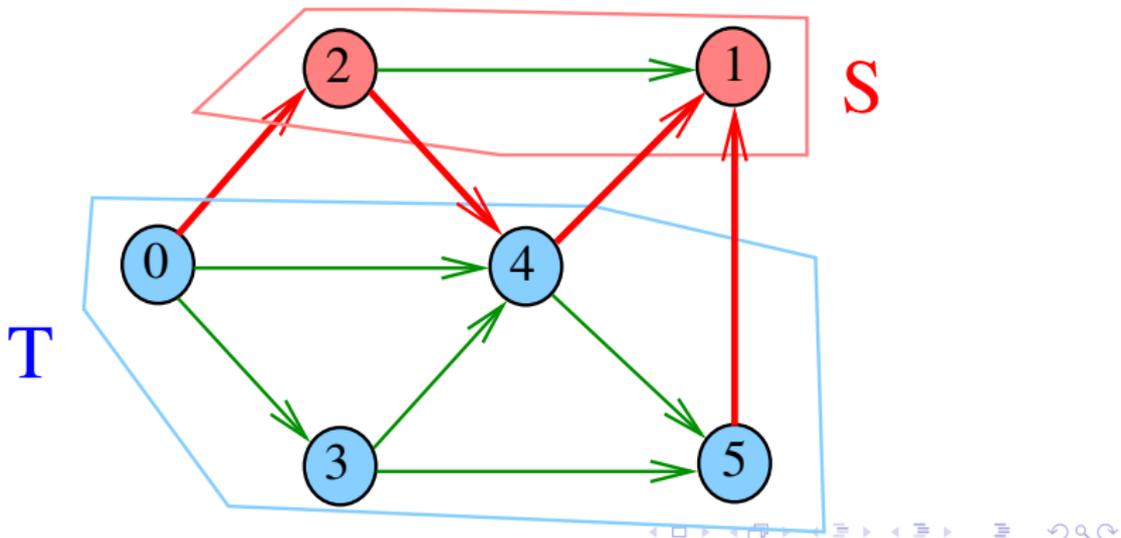


## Cortes (= cuts)

Um **corte** é uma bipartição do conjunto de vértices

Um arco **pertence** ou **atravessa** um corte  $(S, T)$  se tiver uma ponta em  $S$  e outra em  $T$

**Exemplo 1:** arcos em **vermelho** estão no corte  $(S, T)$

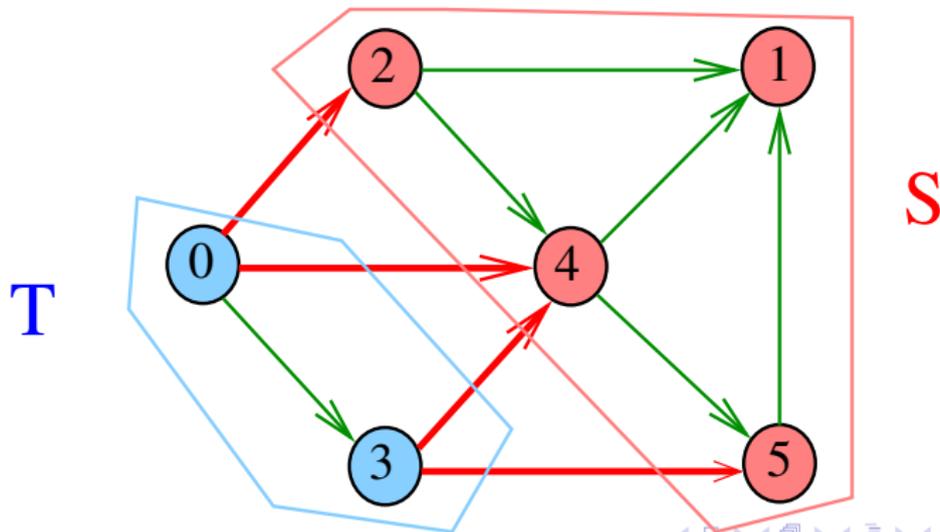


## Cortes (= cuts)

Um **corte** é uma bipartição do conjunto de vértices

Um arco **pertence** ou **atravessa** um corte  $(S, T)$  se tiver uma ponta em  $S$  e outra em  $T$

**Exemplo 2:** arcos em **vermelho** estão no corte  $(S, T)$

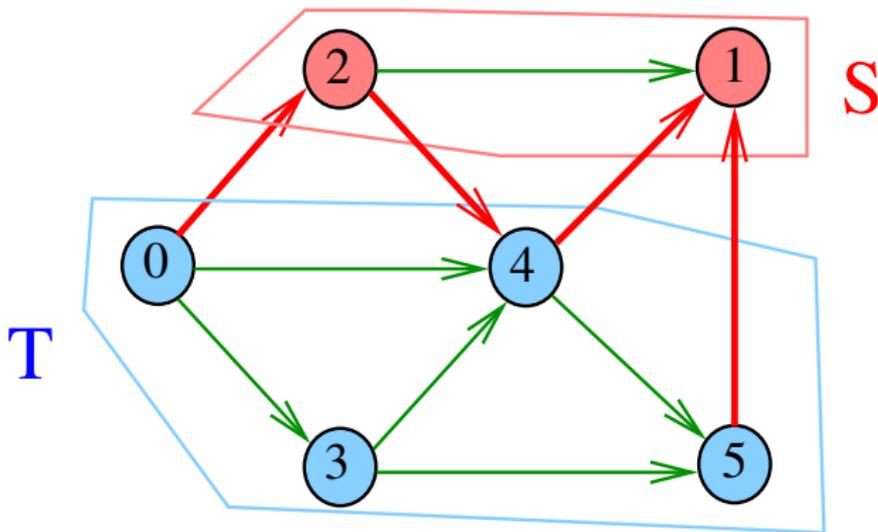


## st-Cortes (= st-cuts)

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 ...

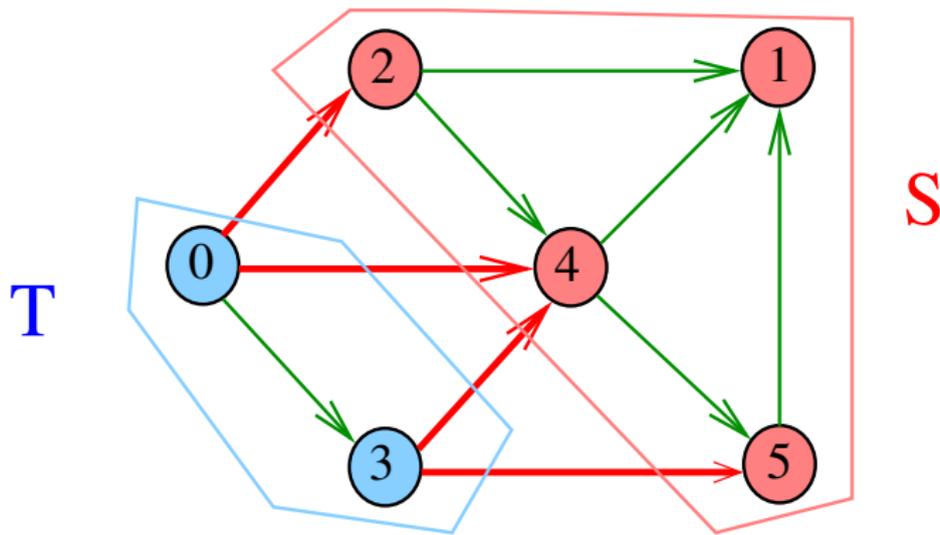


# Certificado de inexistência

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 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  $S$ .



Fonte: [Yin and yang \(Wikipedia\)](#)

## E DFSpaths com isso?

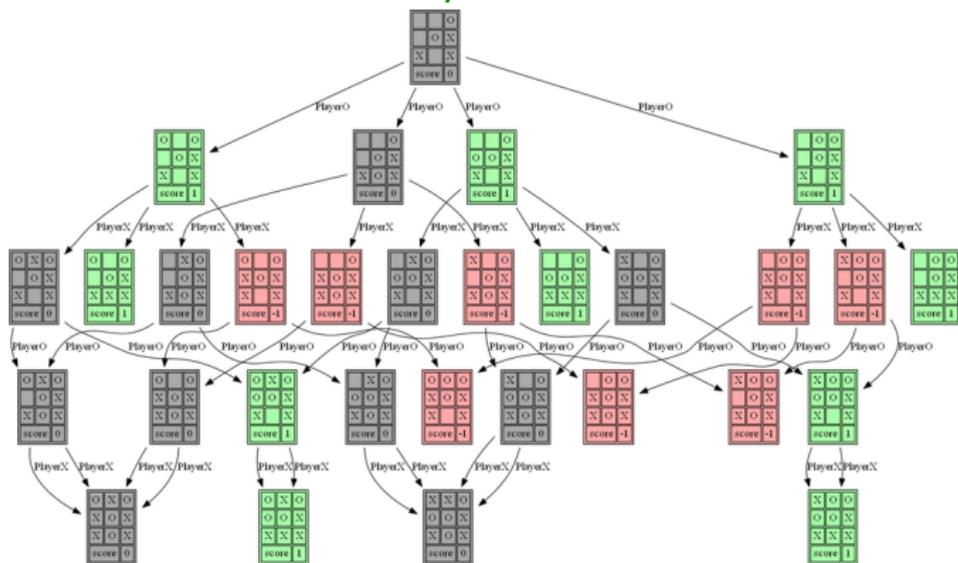
No código da classe `DFSpaths` se existe um caminho de `s` a `t` ele está representado no vetor `edgeTo[]`.

No código da classe `DFSpaths` 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  $V + E$ .

# Anatomia de busca em profundidade

## Classificação dos arcos



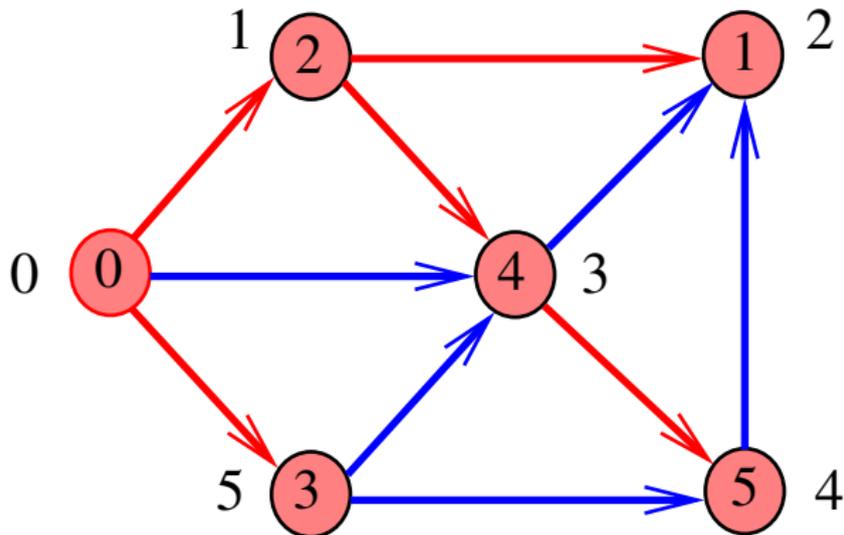
Fonte: Using Minimax (with the full game tree) to implement the machine players ...

Referências: CLRS 22

## Arcos da arborescência

**Arcos da arborescência** são os arcos  $v-w$  que  $\text{dfsR}$  percorre para visitar  $w$  pela primeira vez **Exemplo:**

arcos em **vermelho** são arcos da arborescência

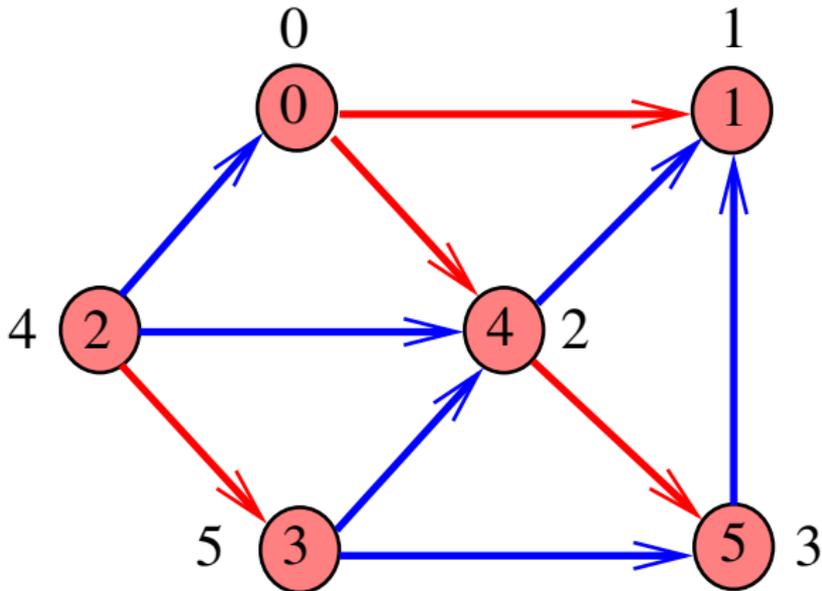




## Floresta DFS

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

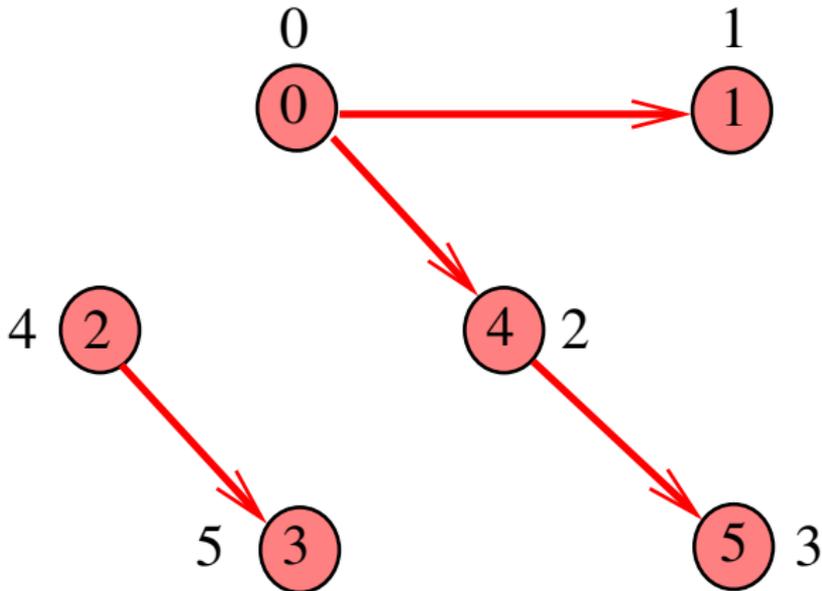
Exemplo: arcos em **vermelho** formam a floresta DFS



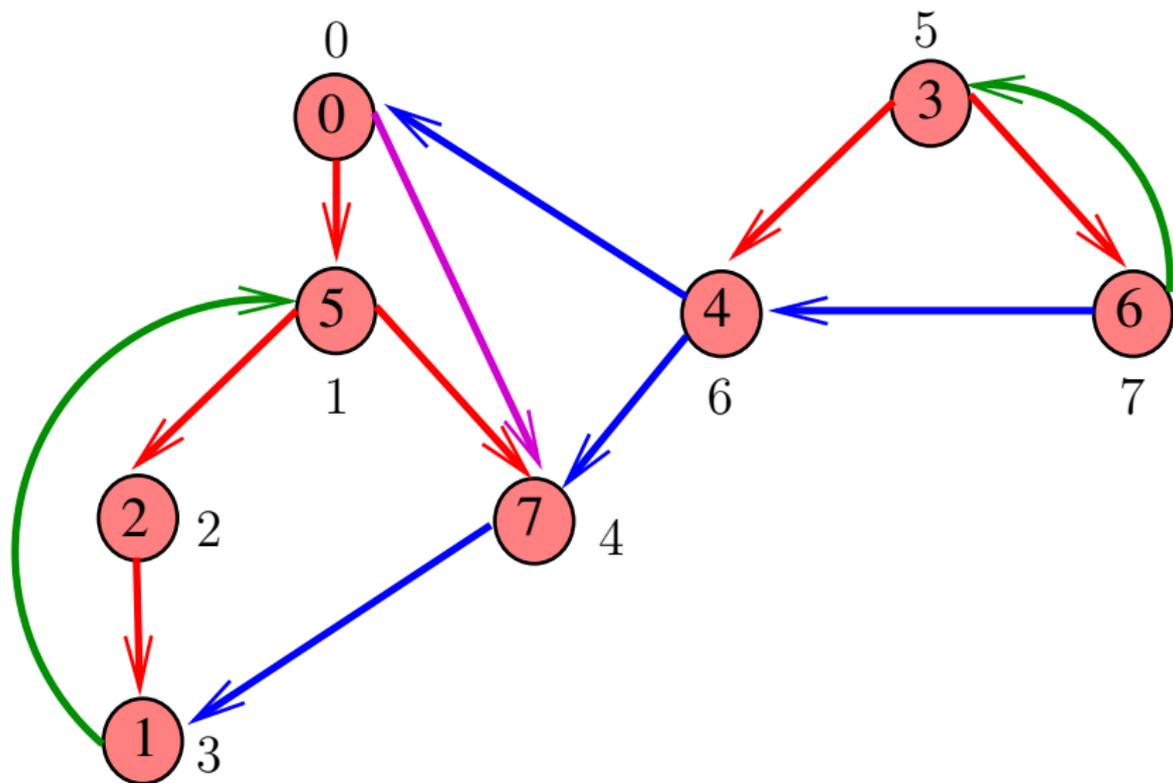
## Floresta DFS

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

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

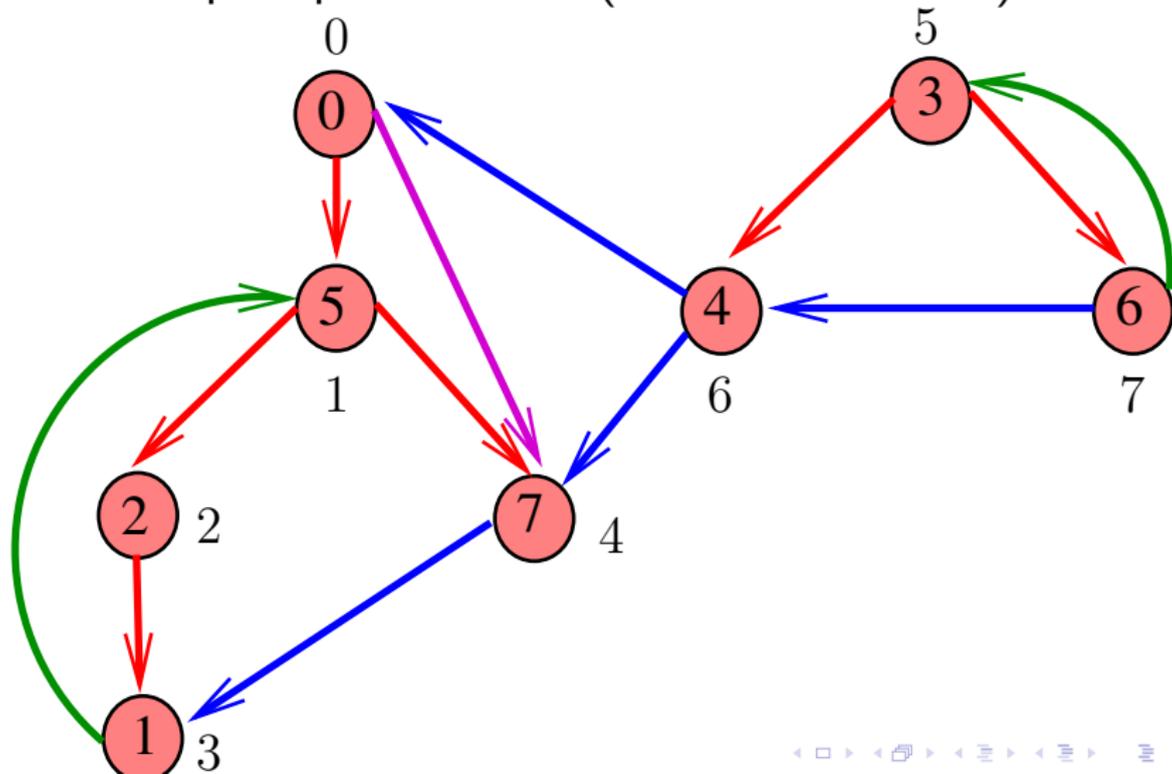


# Classificação dos arcos



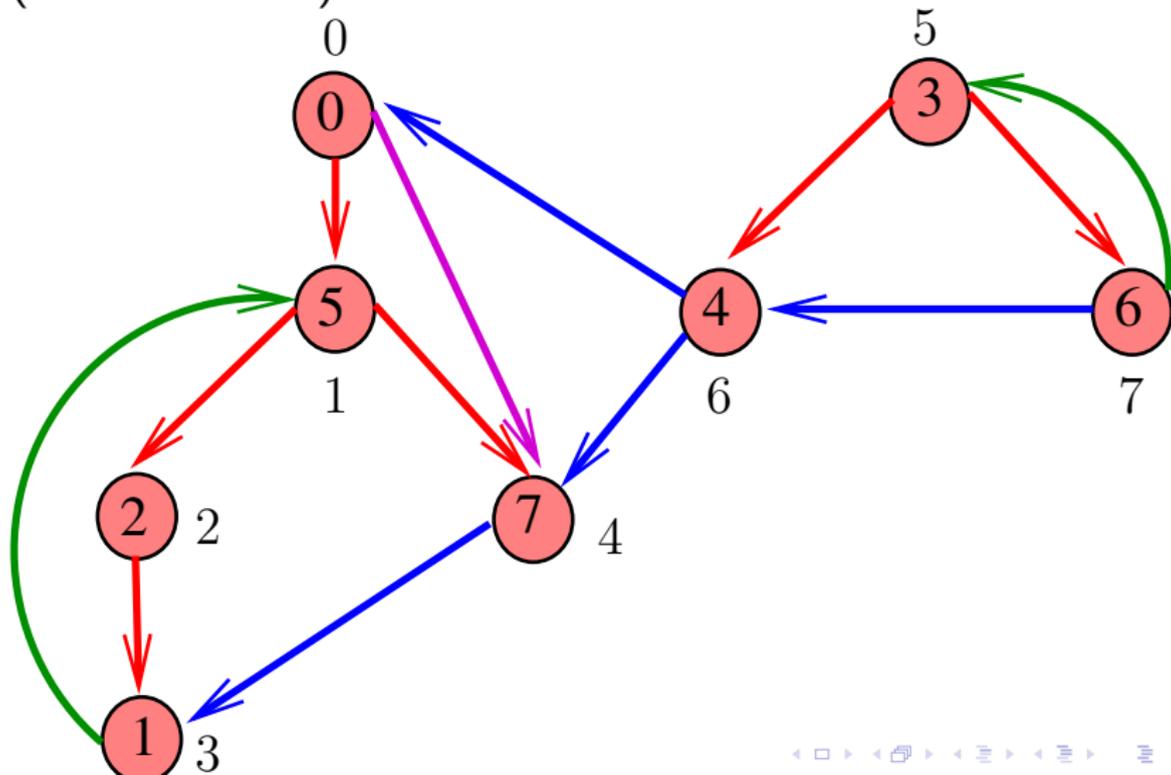
## Arcos de arborescência

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



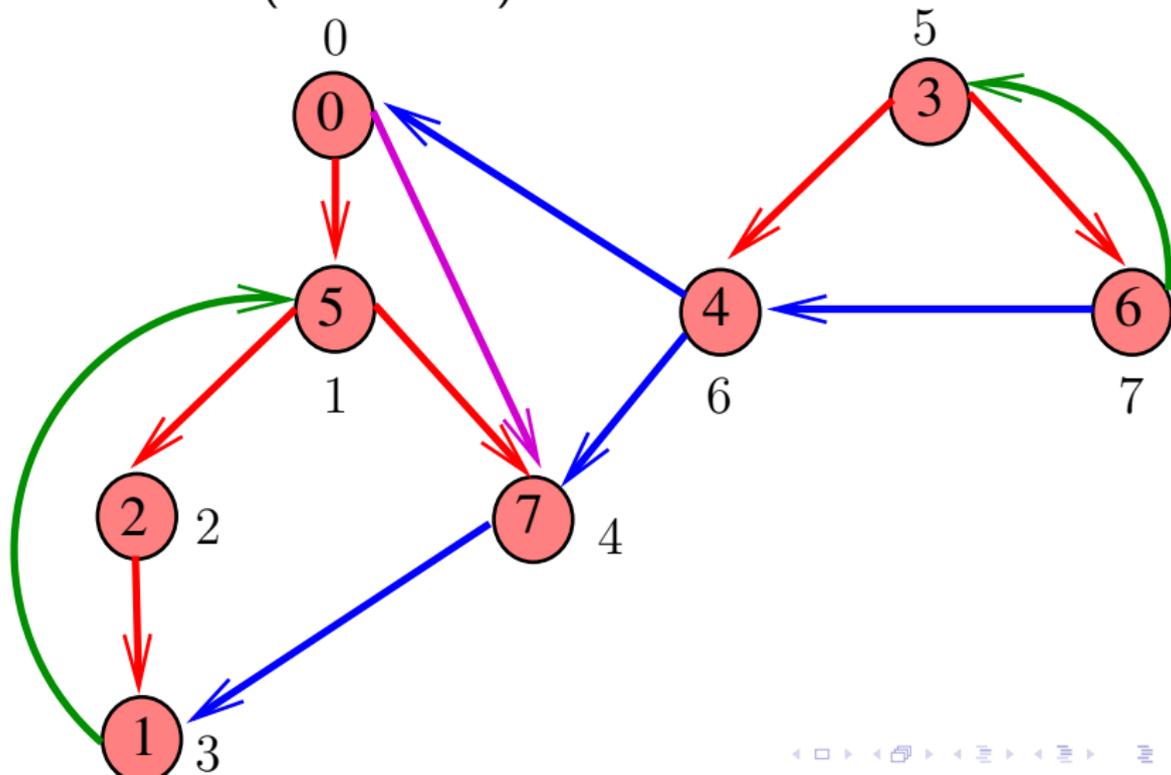
## Arcos de retorno

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



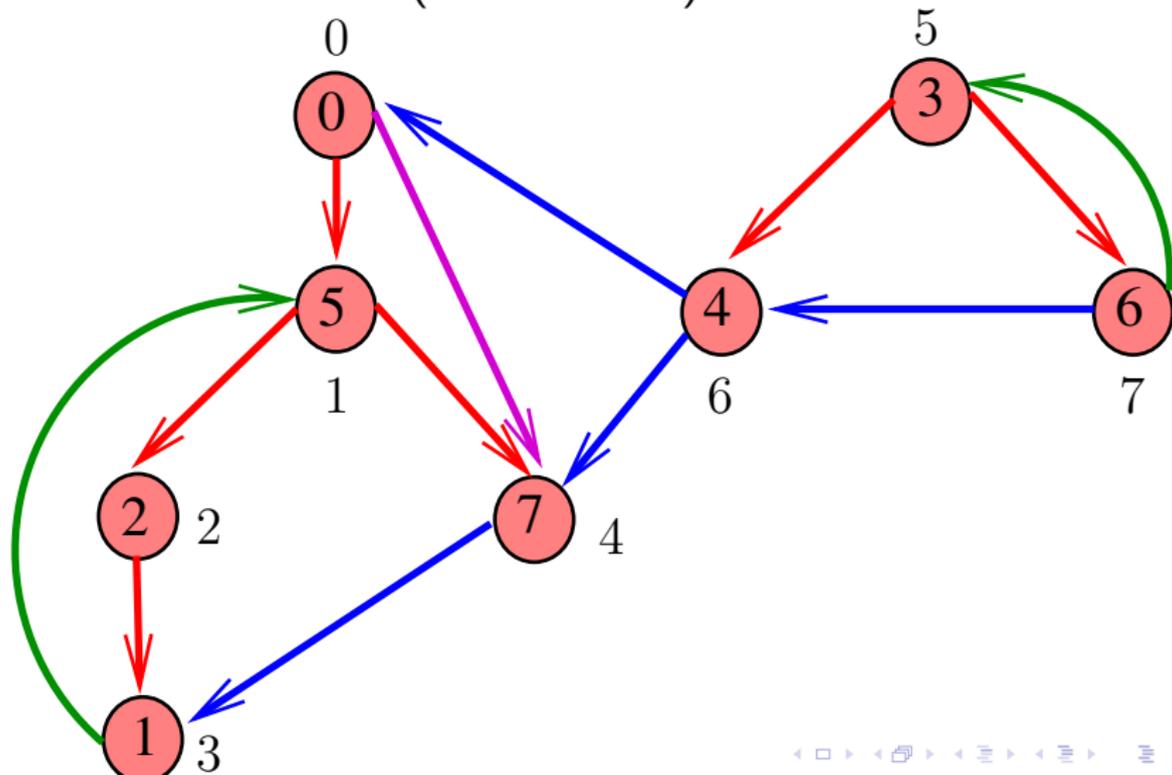
## Arcos descendentes

$v-w$  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 DFSanatomia (CLRS)

Vamos supor que nossos digrafos têm no máximo  $\text{maxV}$  vértices

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

**DFSanatomia** 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

## DFSanatomia: esqueleto

```
public class DFSanatomia {
    private boolean[] marked;
    private int[] edgeTo;
    private int time;
    private int[] d; // discovered
    private int[] f; // finished
    private Queue<Integer> pre; pré-ordem
    private Queue<Integer> pos; pós-ordem
    // pós-ordem reversa
    private Stack<Integer> revPos;
```

## DFSanatomia: esqueleto

```
// métodos
```

```
public DFSanatomia(Graph G) {...}  
private void dfs(Digraph G, int v){...}  
public Iterable<Integer> pre() {...}  
public Iterable<Integer> pos() {...}  
public Iterable<Integer> revPos() {...}  
}
```

## DFSanatomia: construtor

```
public DFSanatomia(Graph G) {  
    marked = new boolean[G.V()];  
    edgeTo = new int[G.V()];  
    d = new int[G.V()];  
    f = new int[G.V()];  
    pre = new Queue<Integer>;  
    pos = new Queue<Integer>;  
    for (int v = 0; v < G.V(); v++)  
        if (!marked(v)) {  
            dfs(G,v);  
        }  
}
```

## DFSanatomia: dfs()

```
private void dfs(Digraph G, int v) {
    marked[v] = true;
    d[v] = time++; // descoberto
    pre.enqueue(v); // pré-ordem
    for (int w : G.adj[v]) {
        if (!marked(w)) {
            edgeTo[w] = v;
            dfs(G, w);
        }
    }
    pos.enqueue(v); // pós-ordem
    revPos.push(v); // pós-ordem reversa
    f[v] = time++; // terminamos
}
}
```

DFSanatomia: pre(), pos() e  
revPos()

```
public Iterable<Integer> pre() {  
    return pre;  
}
```

```
public Iterable<Integer> pos() {  
    return pos;  
}
```

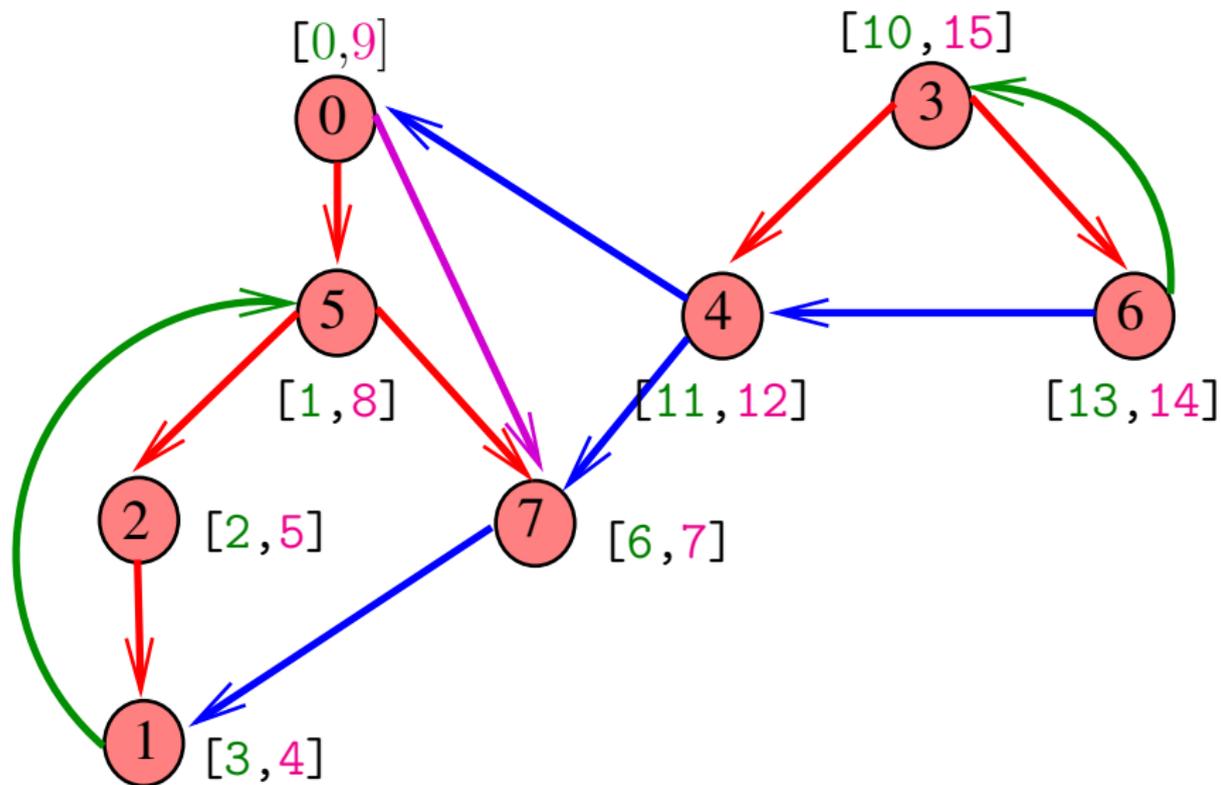
```
public Iterable<Integer> revPos() {  
    return revPos;  
}
```

## Consumo de tempo

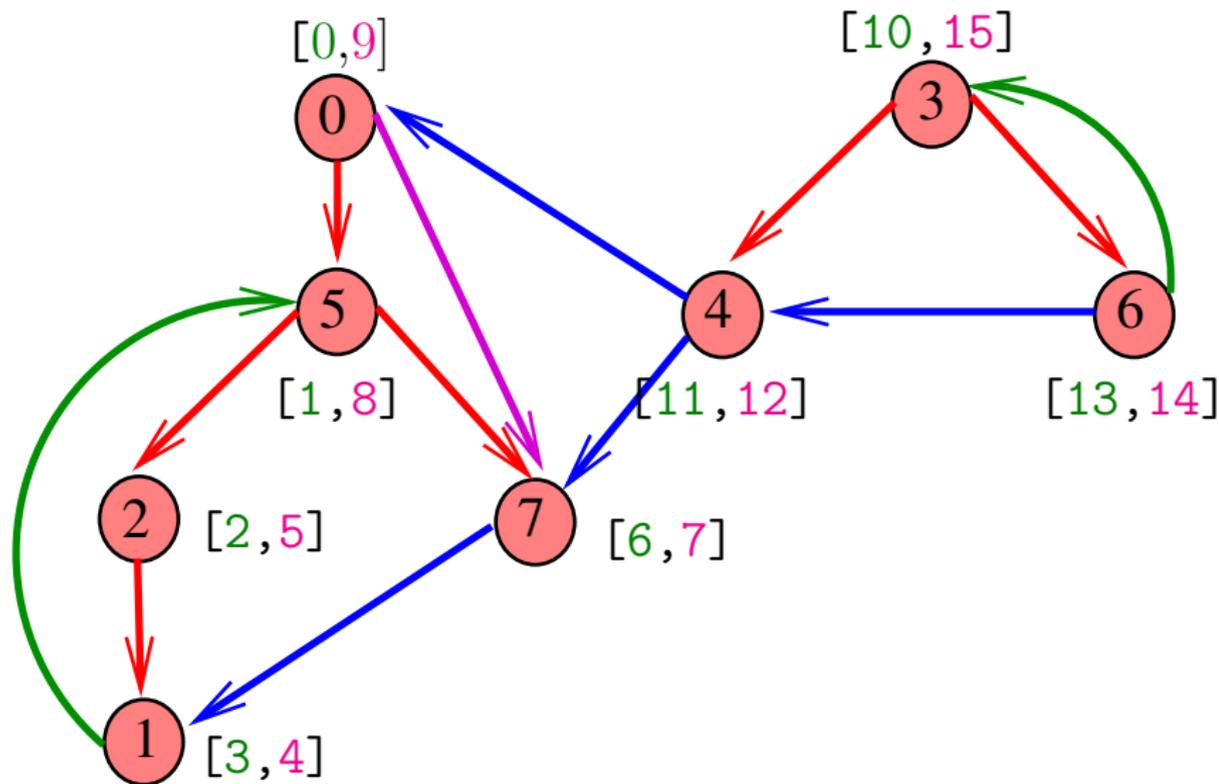
A classe `DFSanatomia`, para vetor de listas de adjacência, consome tempo  $O(V + E)$ .

A classe `DFSanatomia`, para matriz de adjacências, consome tempo  $O(V^2)$ .

# Busca DFS (CLRS)

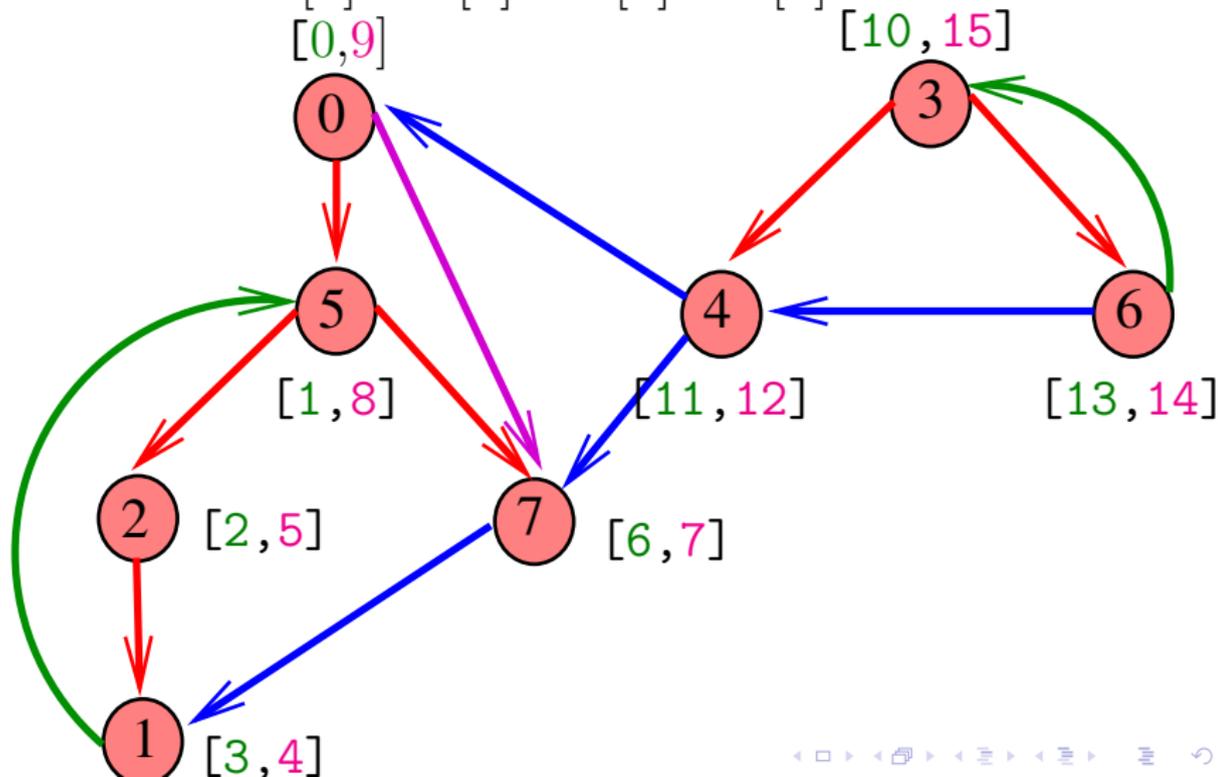


# Classificação dos arcos



## Arcos de arborescência ou descendentes

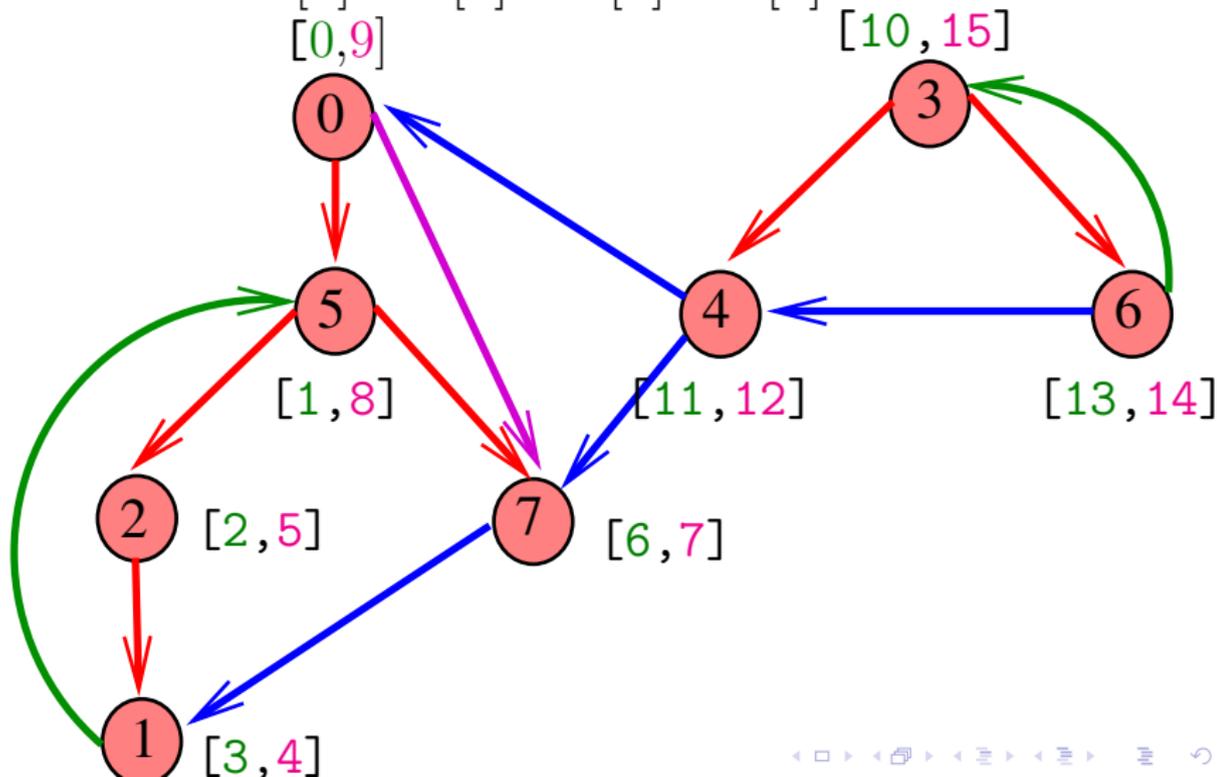
$v-w$  é **arco de arborescência** ou **descendente** se e somente se  $d[v] < d[w] < f[w] < f[v]$



## Arcos de retorno

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

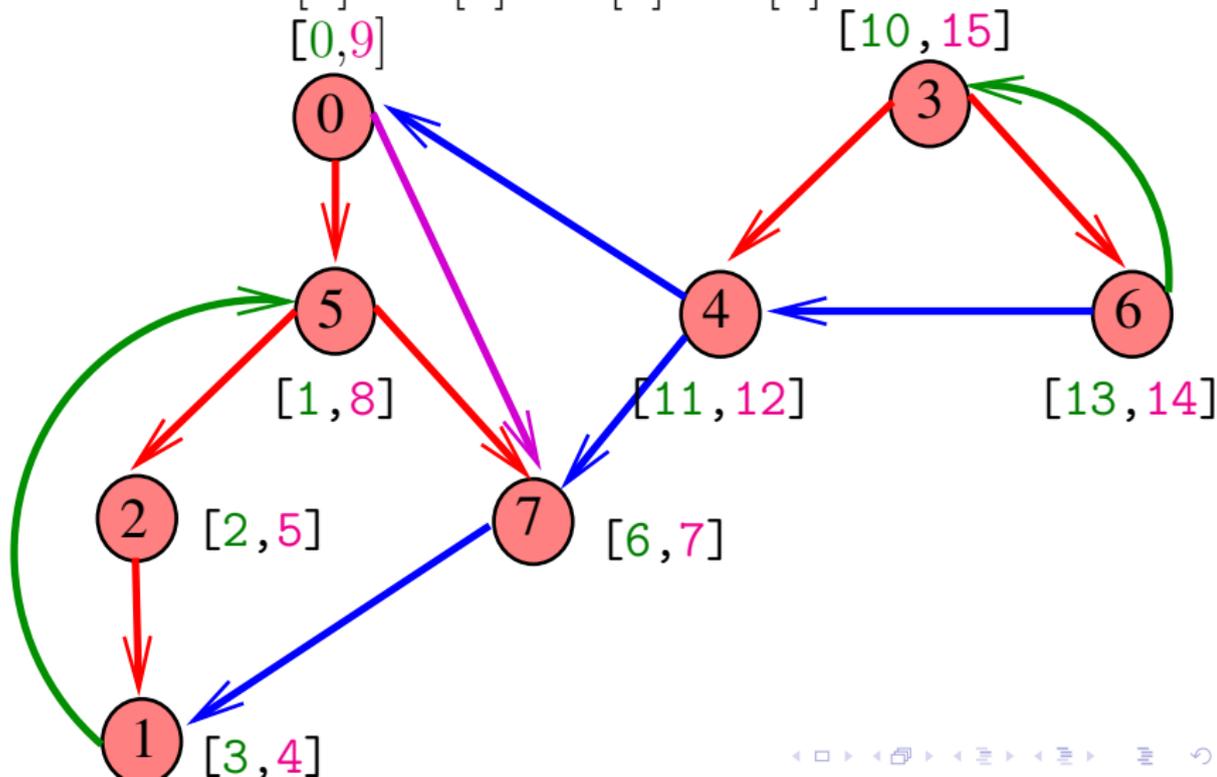
$$d[w] < d[v] < f[v] < f[w]$$



# Arcos cruzados

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

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

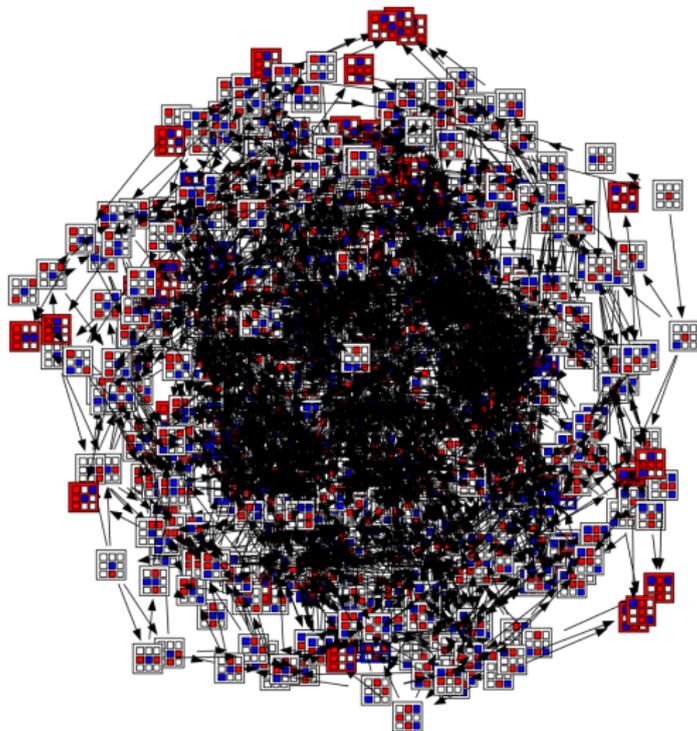


# Conclusões

$v-w$  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



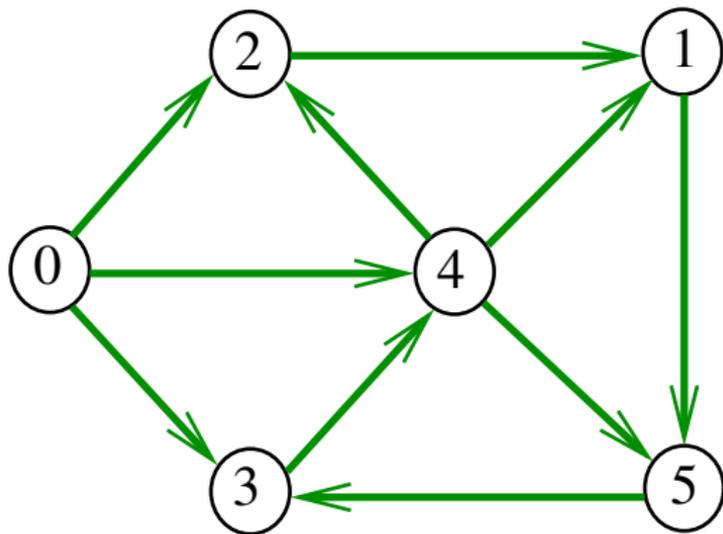
Fonte: [laying out a large graph with graphviz](#)



## Procurando um ciclo

**Problema:** decidir se dado digrafo  $G$  possui um ciclo

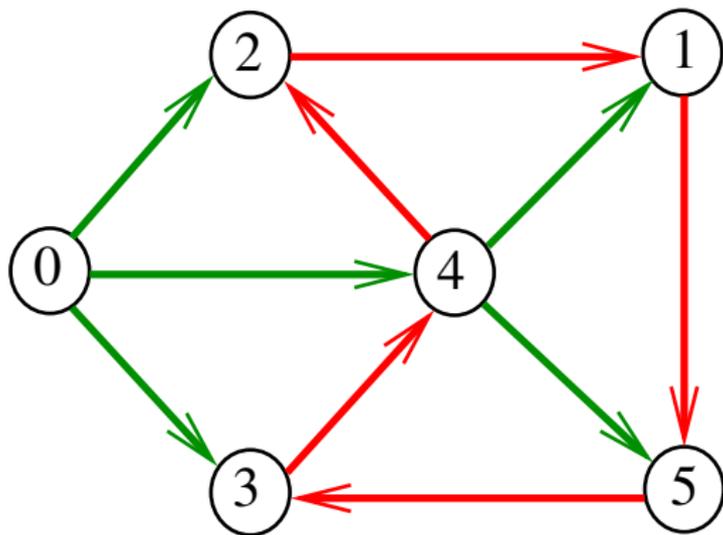
**Exemplo:** para o grafo a seguir a resposta é **SIM**



## Procurando um ciclo

**Problema:** decidir se dado digrafo  $G$  possui um ciclo

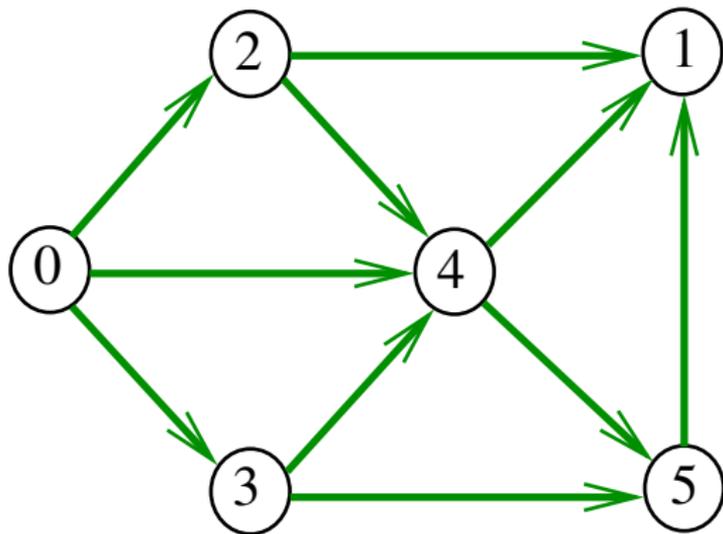
**Exemplo:** para o grafo a seguir a resposta é **SIM**



## 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.

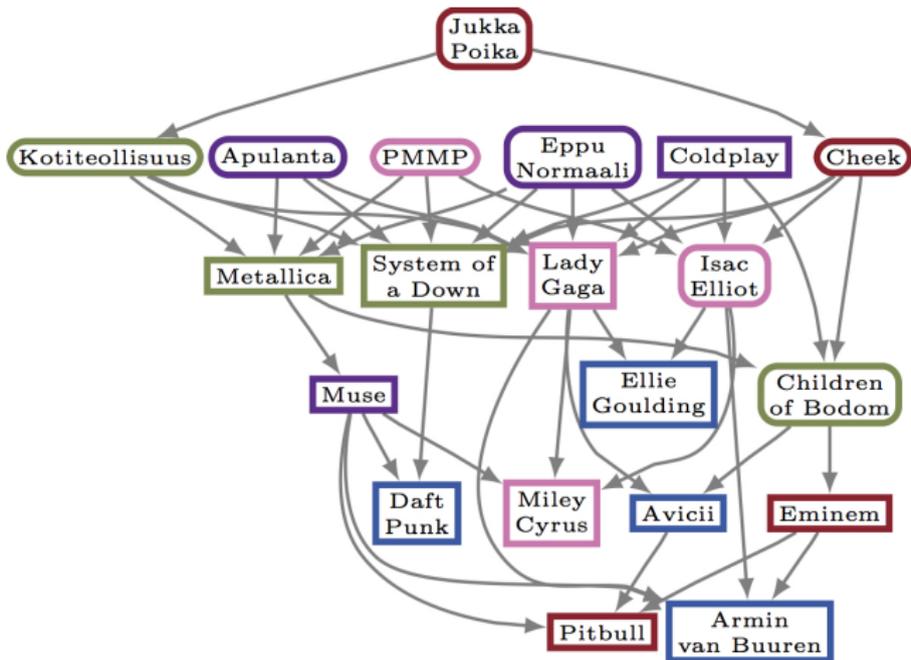
Para cada arco  $u-v$  podemos fazer

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

e verificar se `dfs.hasPath(u)`

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

# Digrafos acíclicos (DAGs)

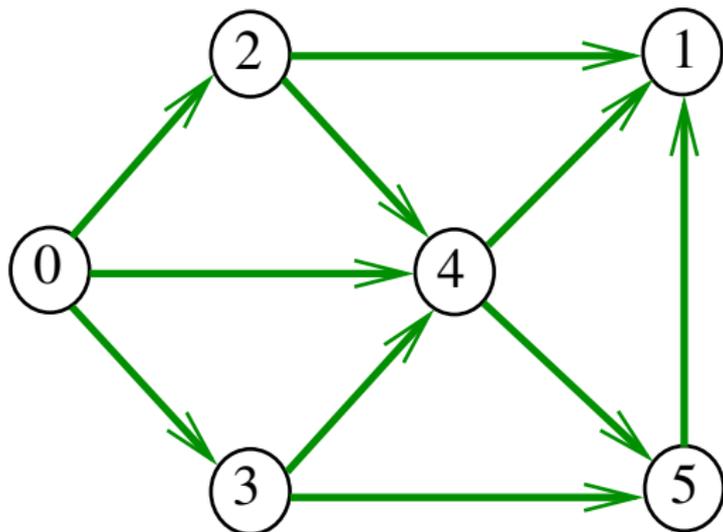


Fonte: Comparing directed acyclic graphs

## DAGs

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

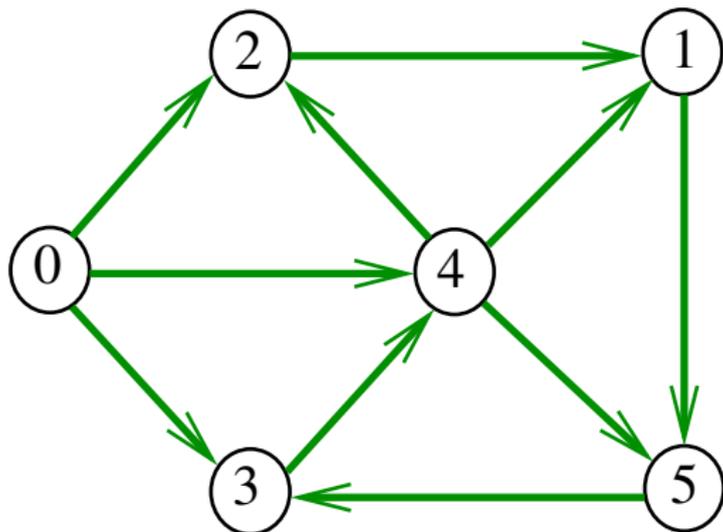
**Exemplo:** um digrafo acíclico



## DAGs

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

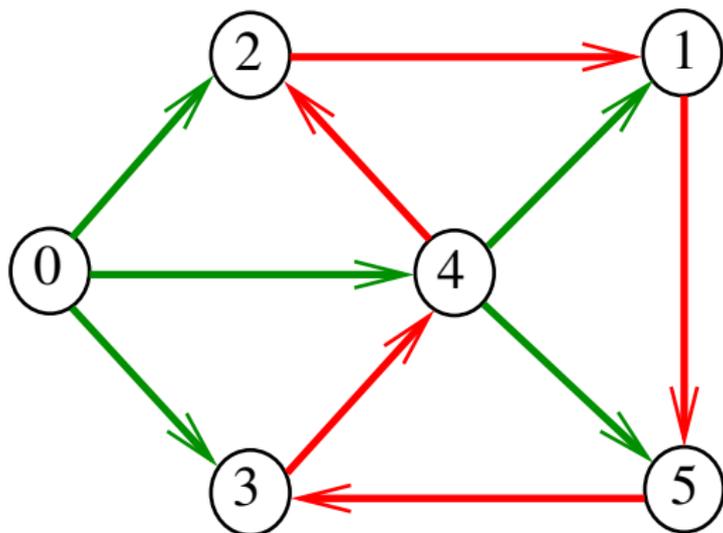
**Exemplo:** um digrafo que **não** é acíclico



## 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

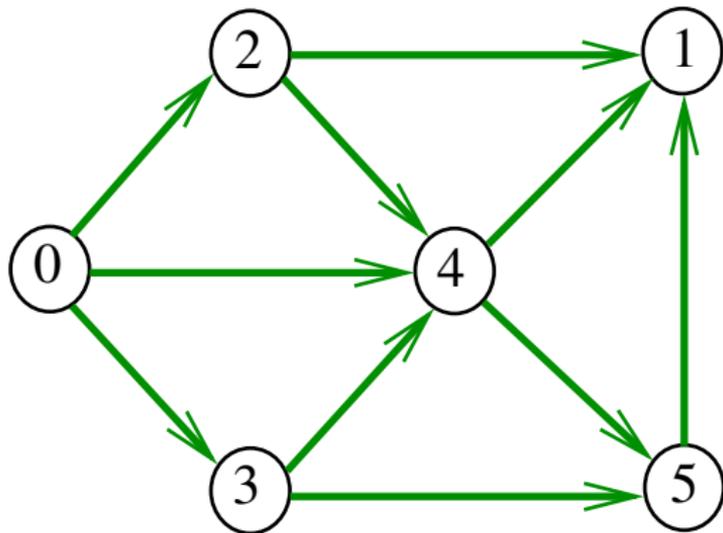
$$ts[i] - ts[j] \text{ com } i < j$$

$ts[0]$  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

## 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  $i$

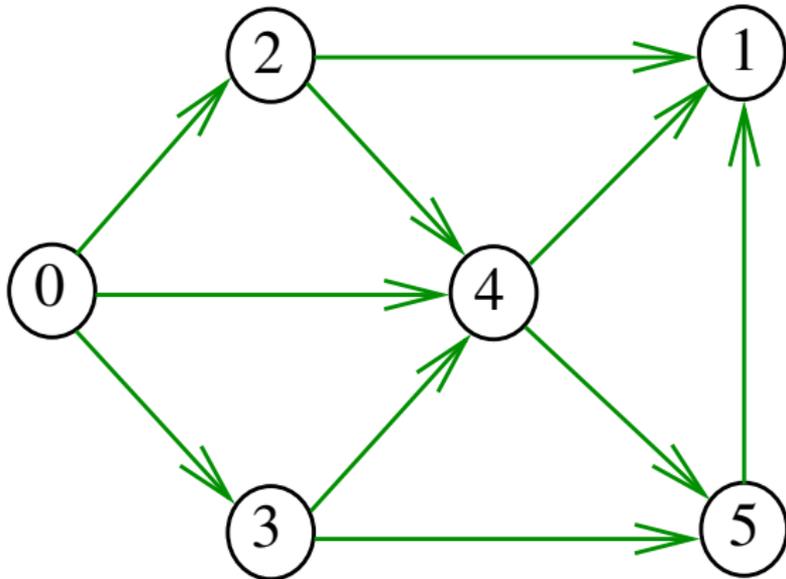
Se  $i = G \rightarrow V$  então  $ts[0 \dots i-1]$  é uma ordenação topológica de  $G$ .

Caso contrário,  $G$  **não** é um DAG

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

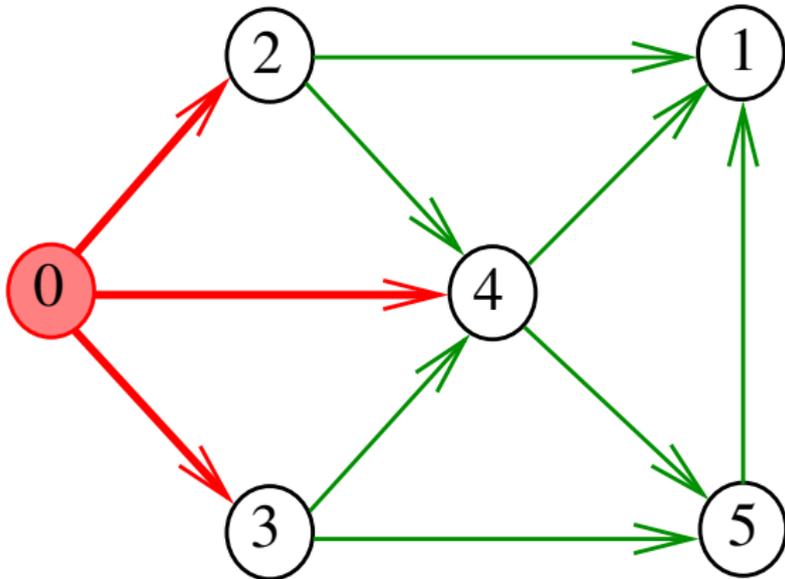
# Exemplo

i	0	1	2	3	4	5
ts[i]						



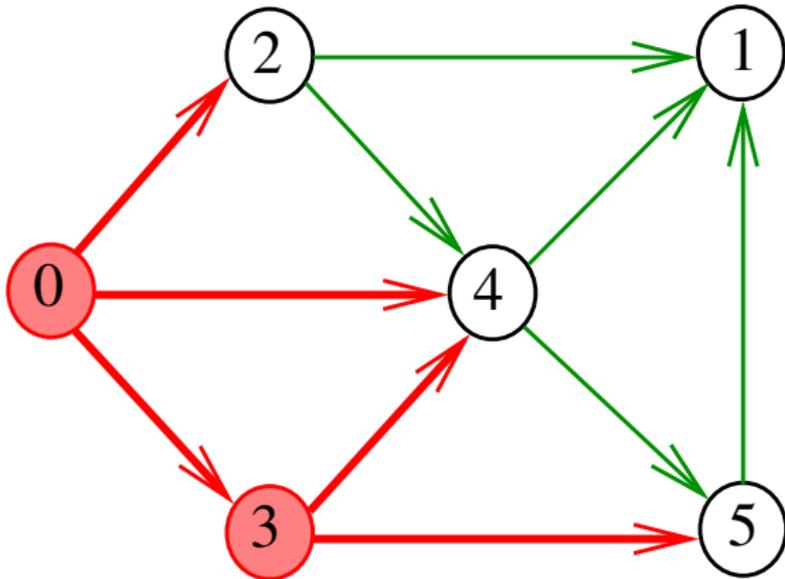
# Exemplo

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



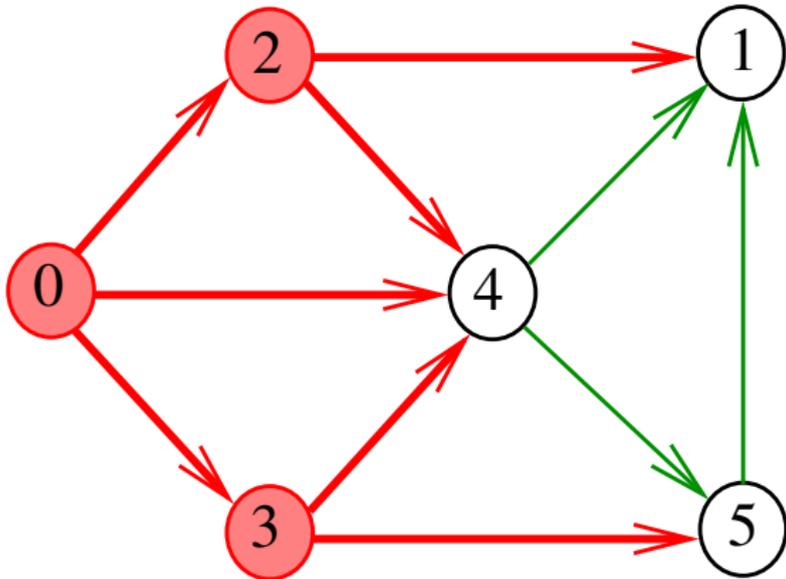
# Exemplo

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



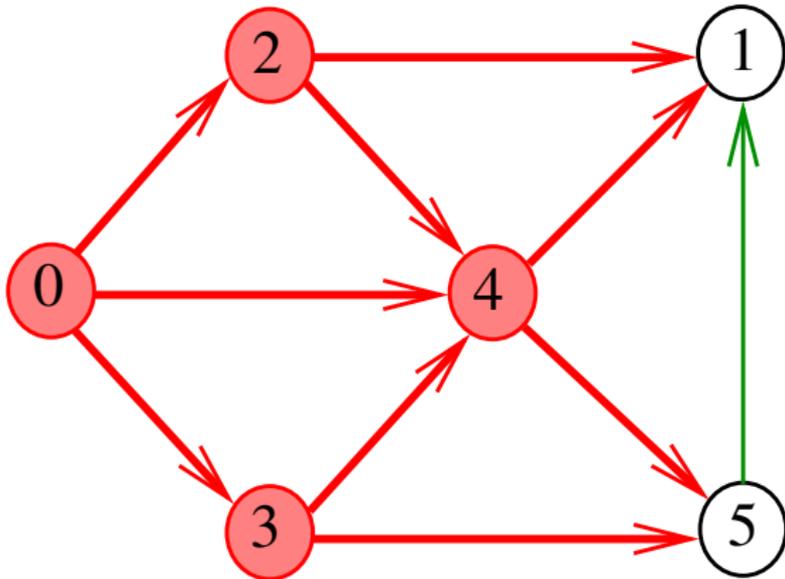
# Exemplo

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



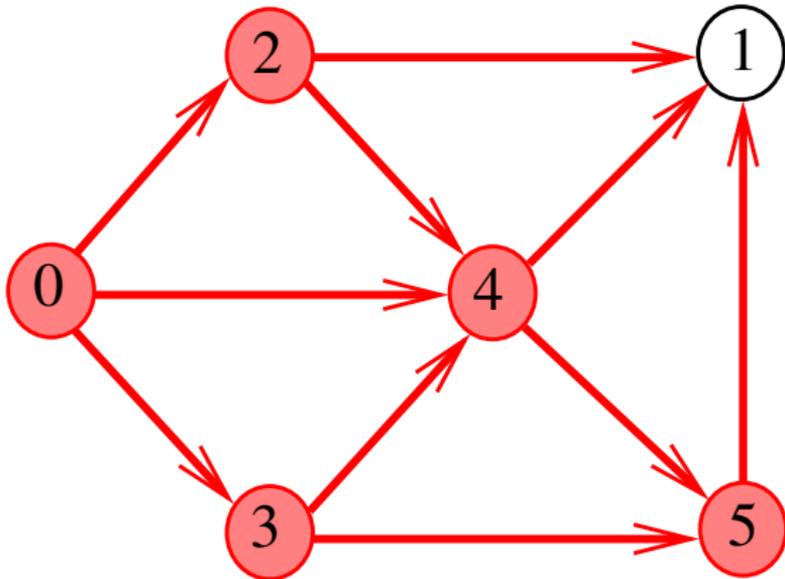
# Exemplo

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



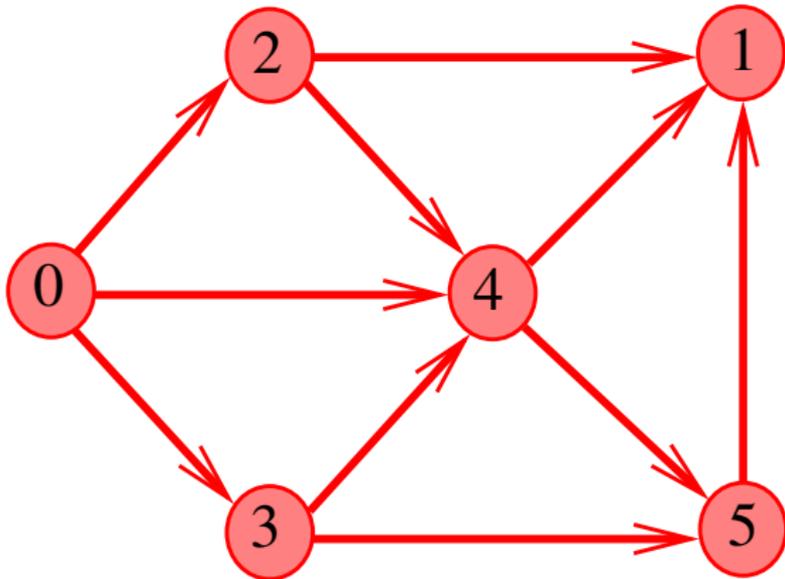
# Exemplo

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



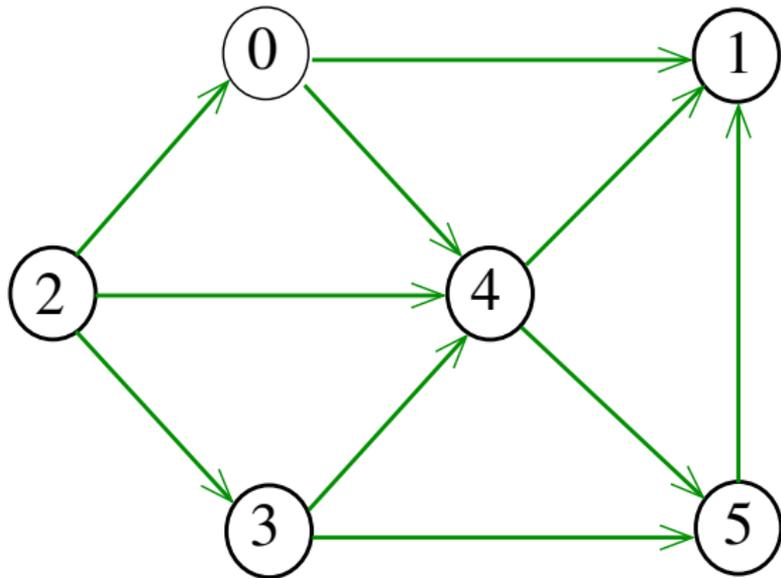
## Consumo de tempo

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

O consumo de tempo desse algoritmo para **matriz de adjacências** é  $O(V^2)$ .

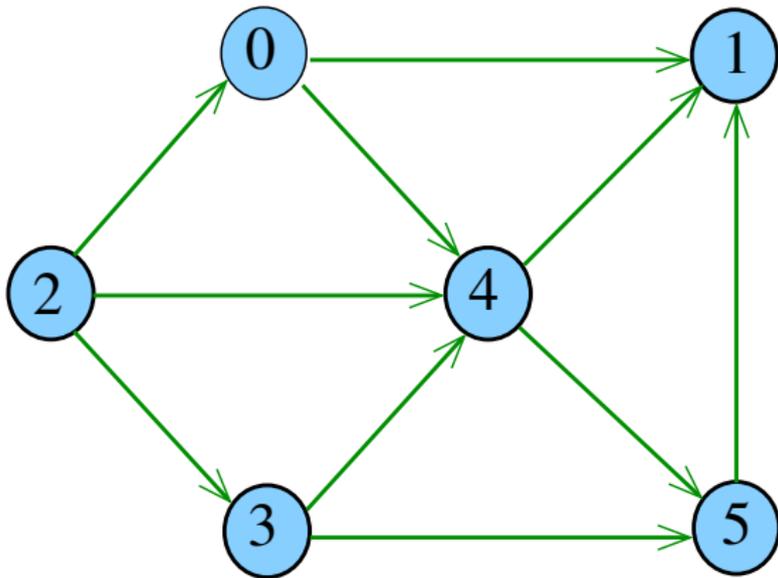
# Algoritmo DFS

i	0	1	2	3	4	5
ts[i]						



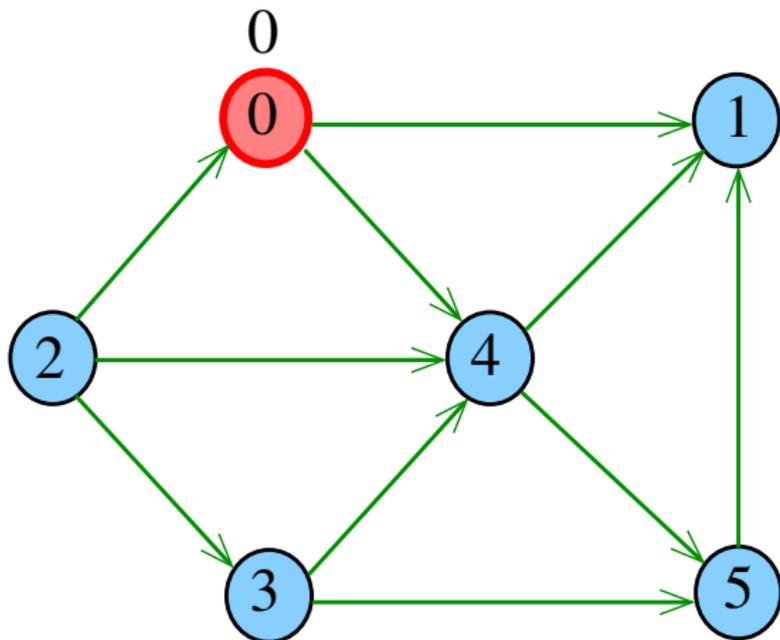
# Algoritmo DFS

i	0	1	2	3	4	5
ts[i]						



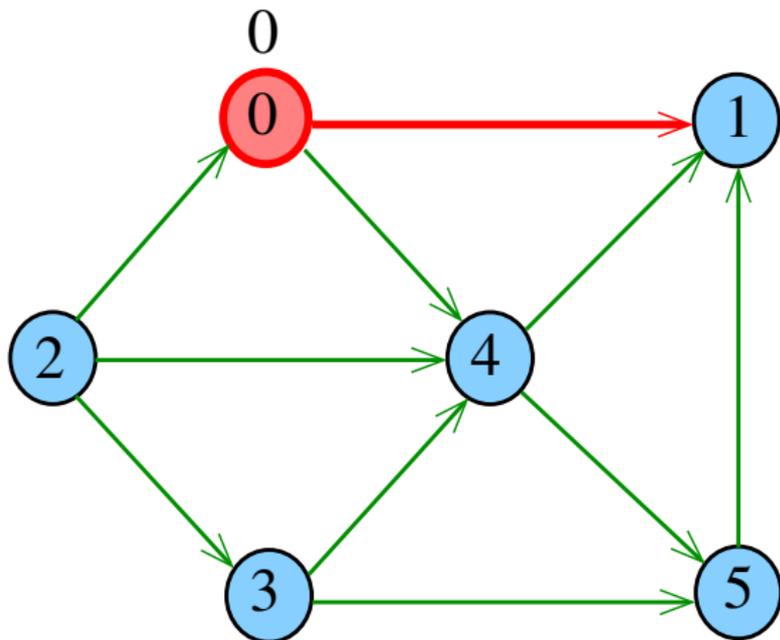
# Algoritmo DFS

i	0	1	2	3	4	5
ts[i]						



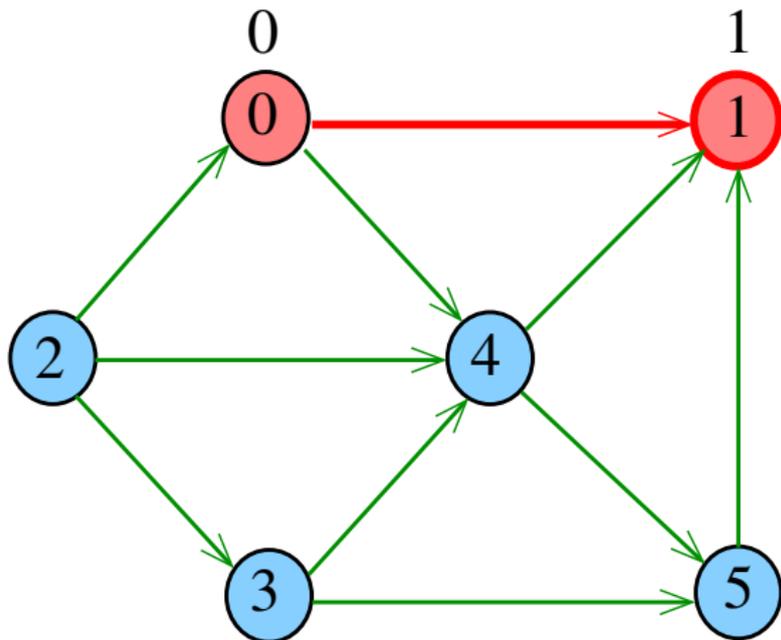
# Algoritmo DFS

i	0	1	2	3	4	5
ts[i]						



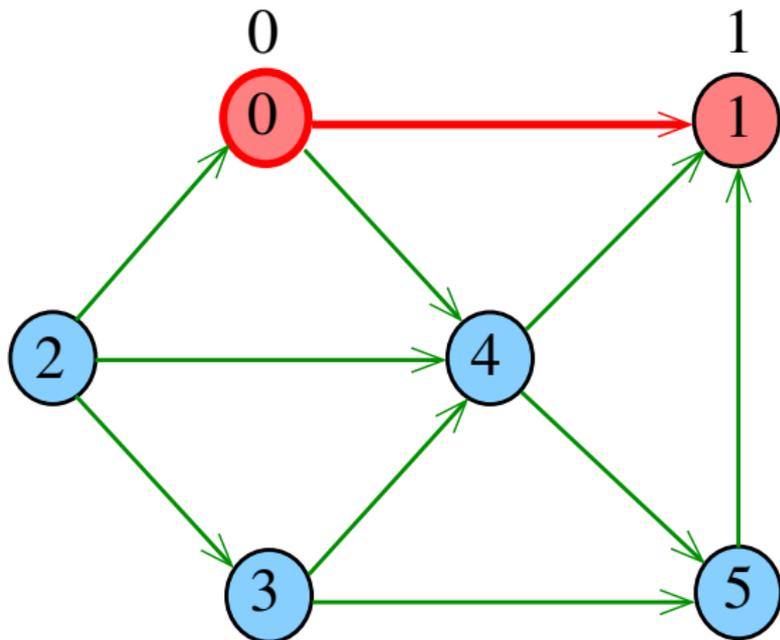
# Algoritmo DFS

i	0	1	2	3	4	5
ts[i]						



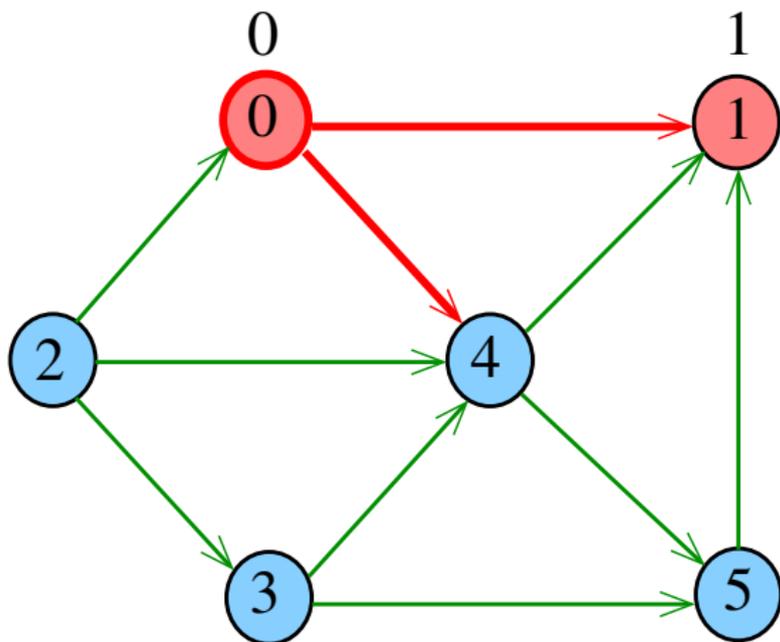
# Algoritmo DFS

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



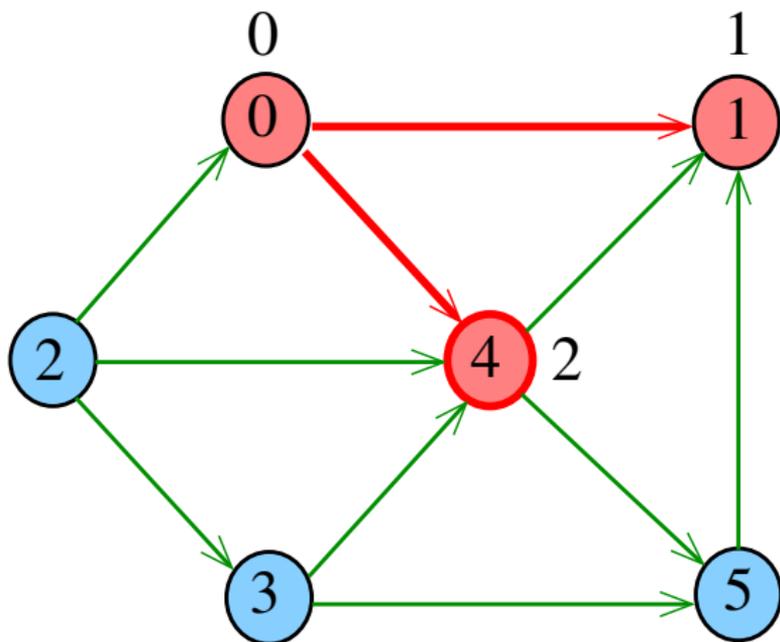
# Algoritmo DFS

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



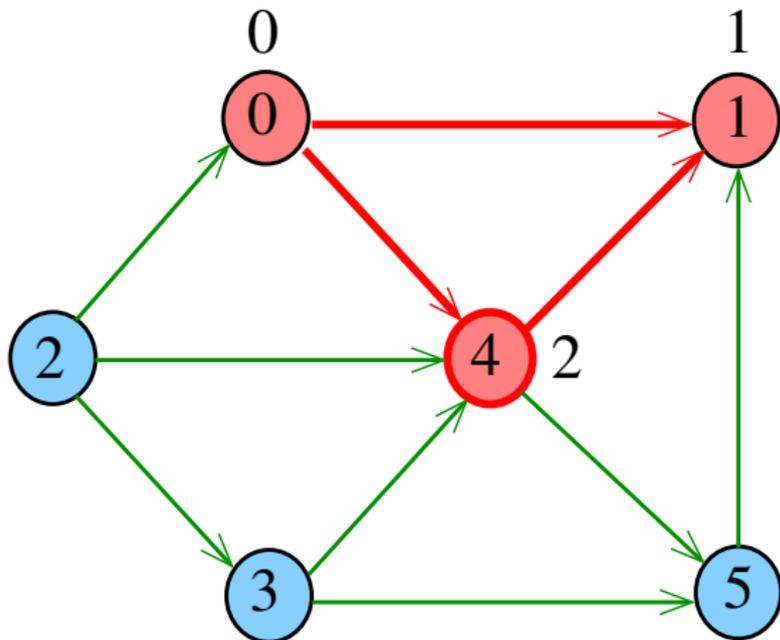
# Algoritmo DFS

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



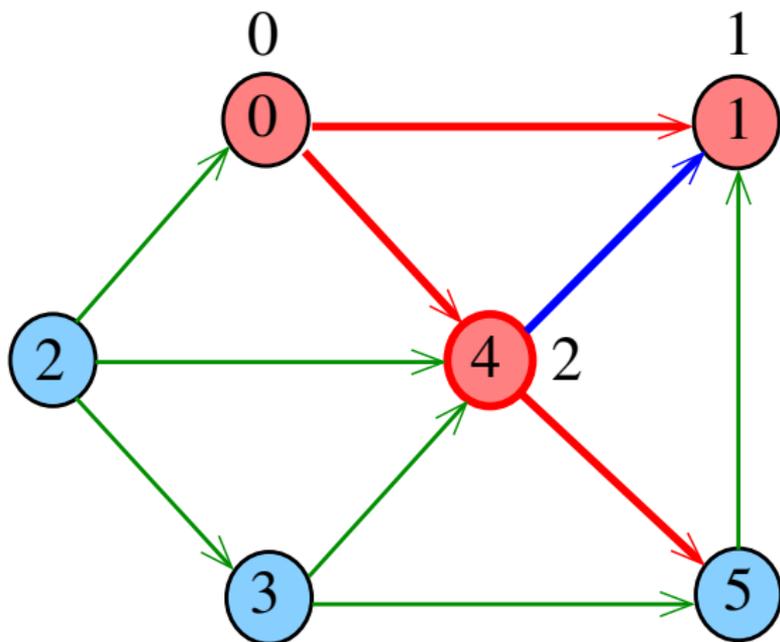
# Algoritmo DFS

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



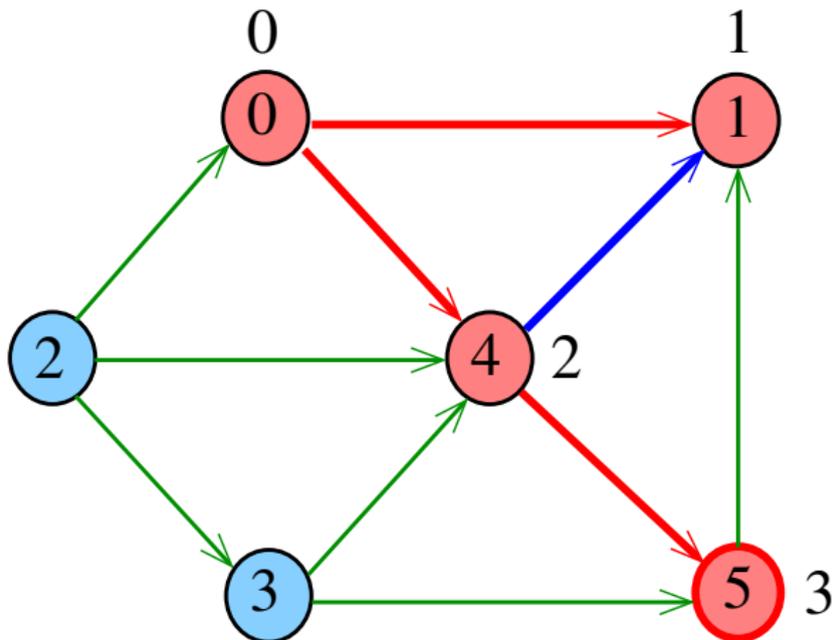
# Algoritmo DFS

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



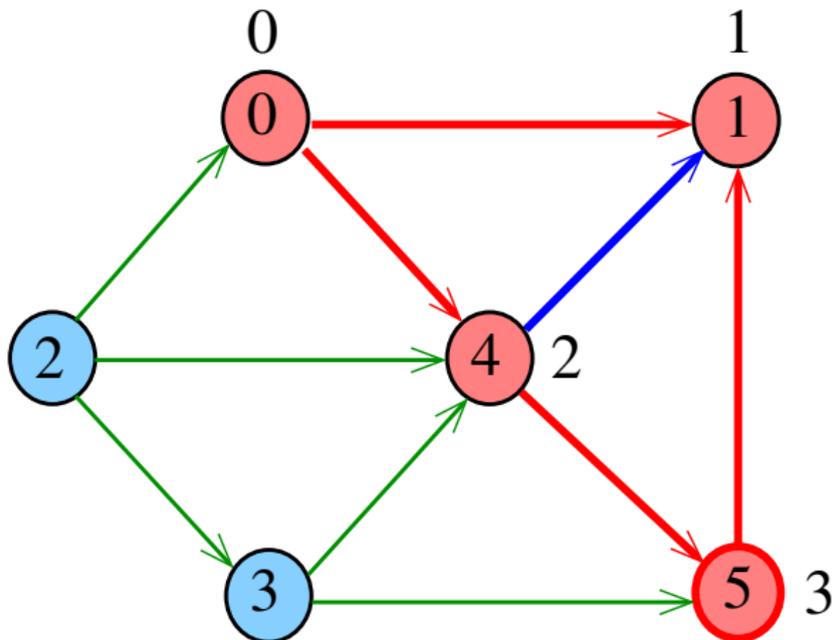
# Algoritmo DFS

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



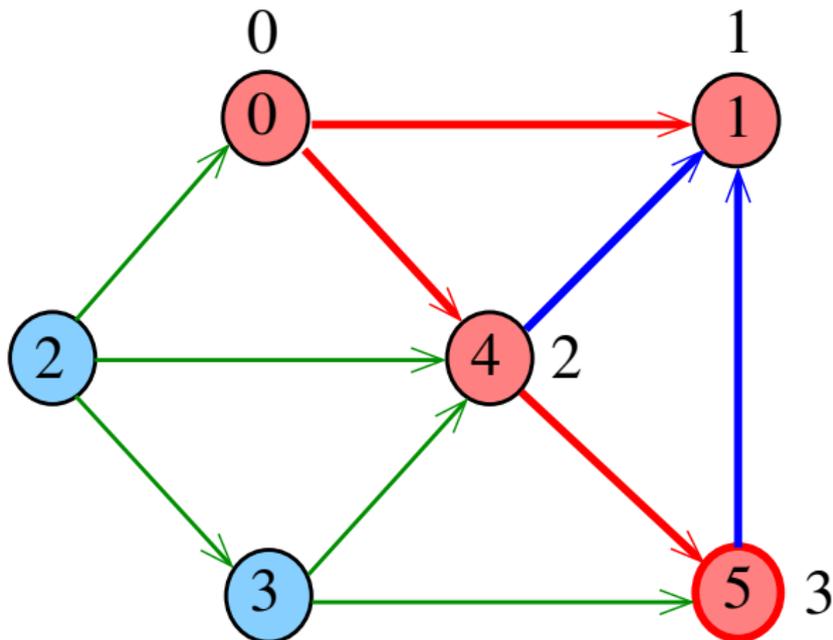
# Algoritmo DFS

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



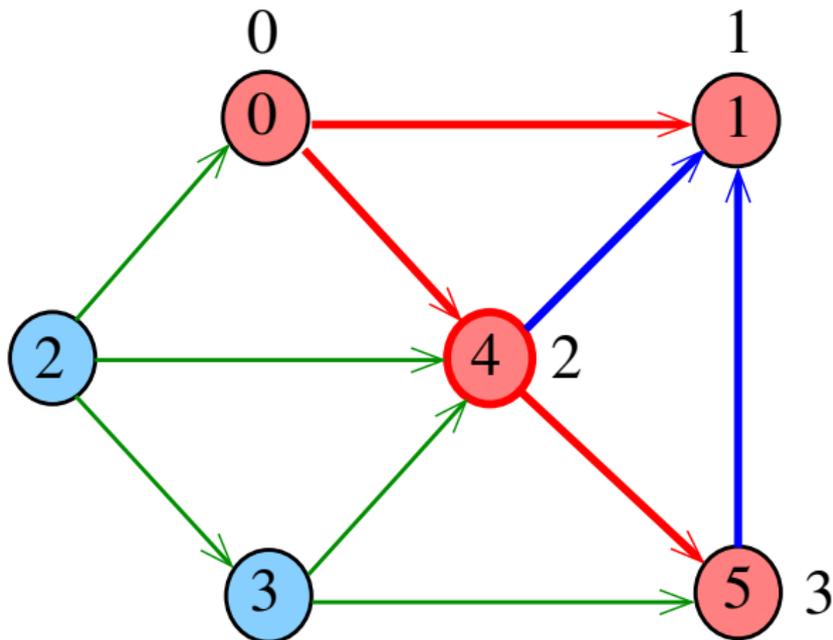
# Algoritmo DFS

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



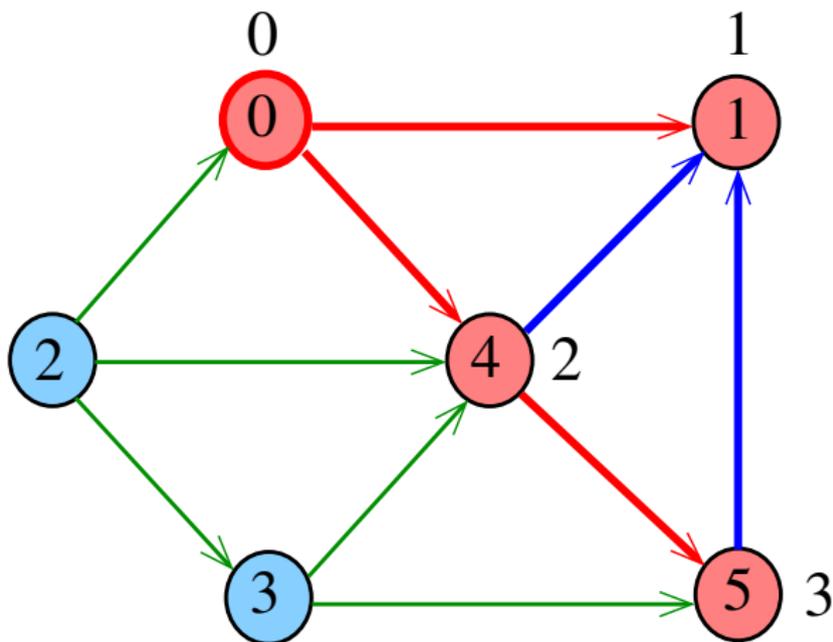
# Algoritmo DFS

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



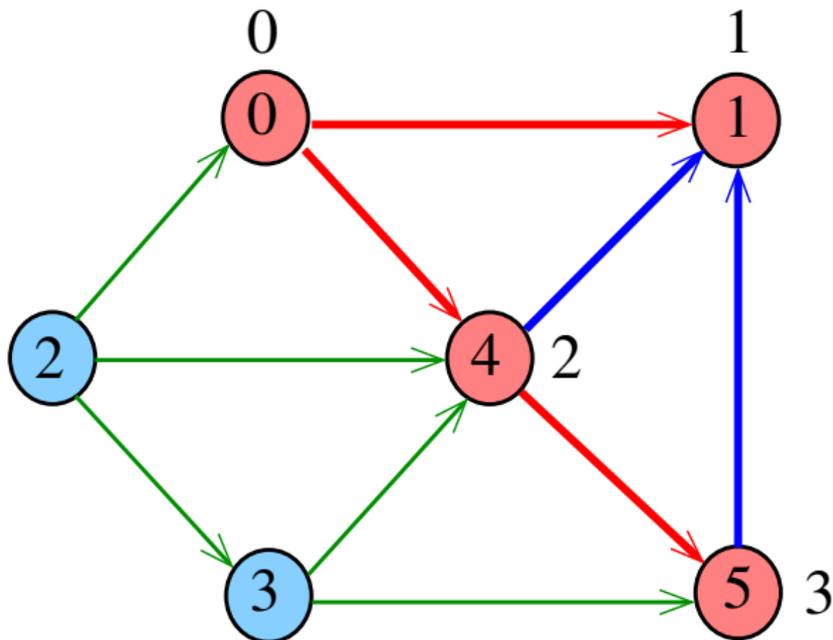
# Algoritmo DFS

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



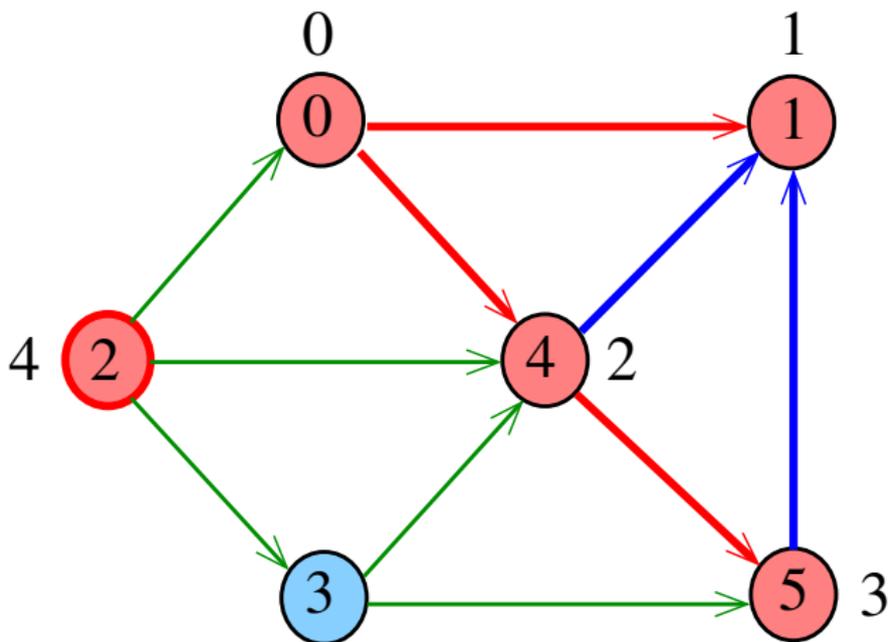
# Algoritmo DFS

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



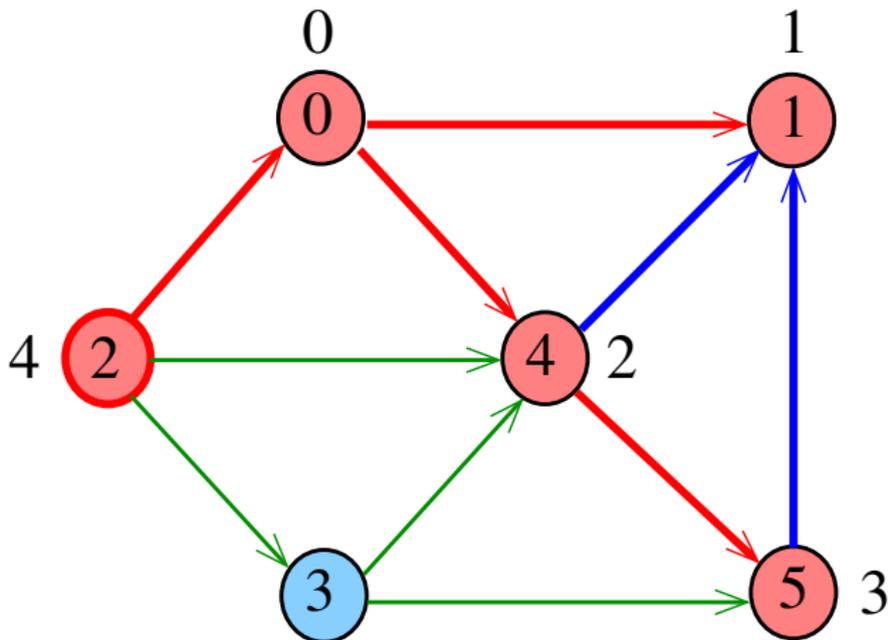
# Algoritmo DFS

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



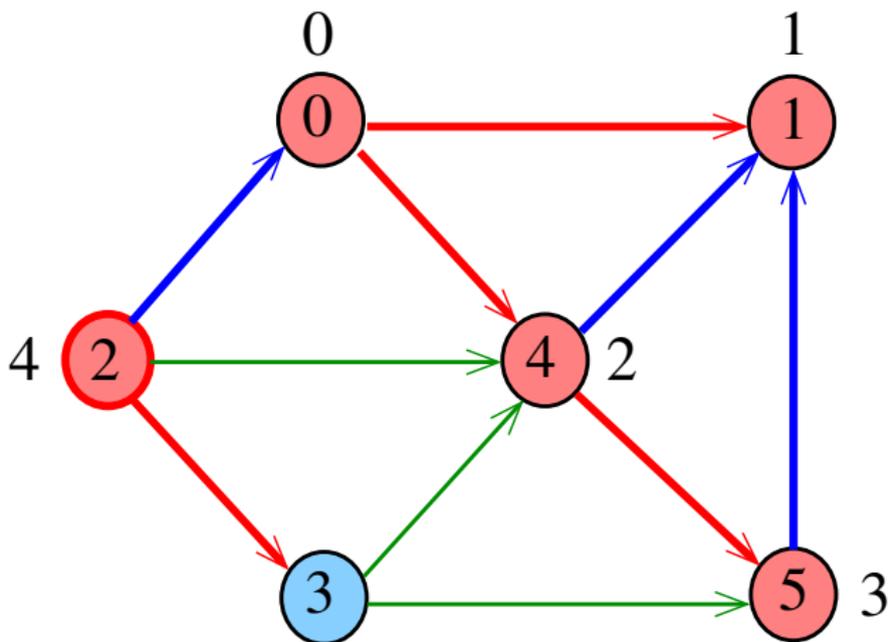
# Algoritmo DFS

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



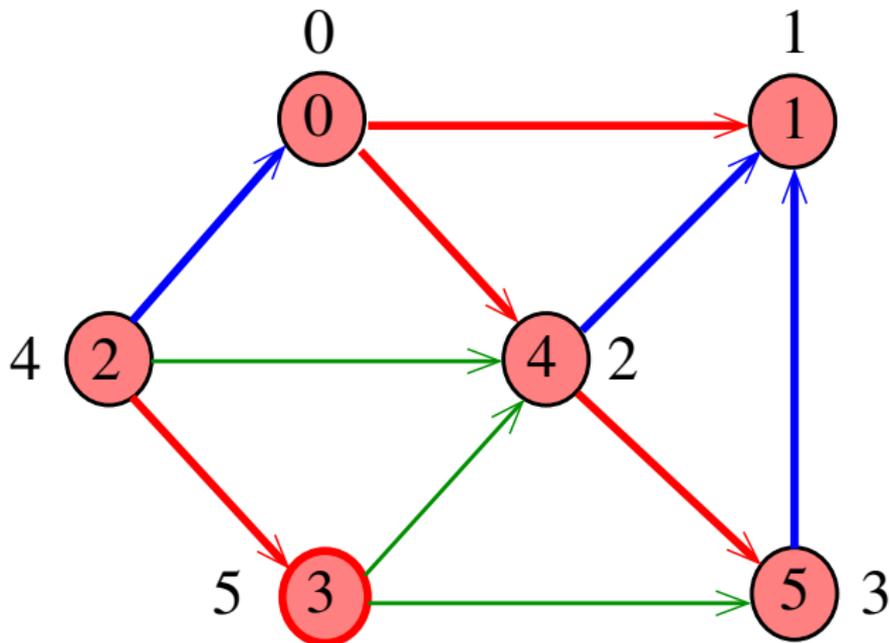
# Algoritmo DFS

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



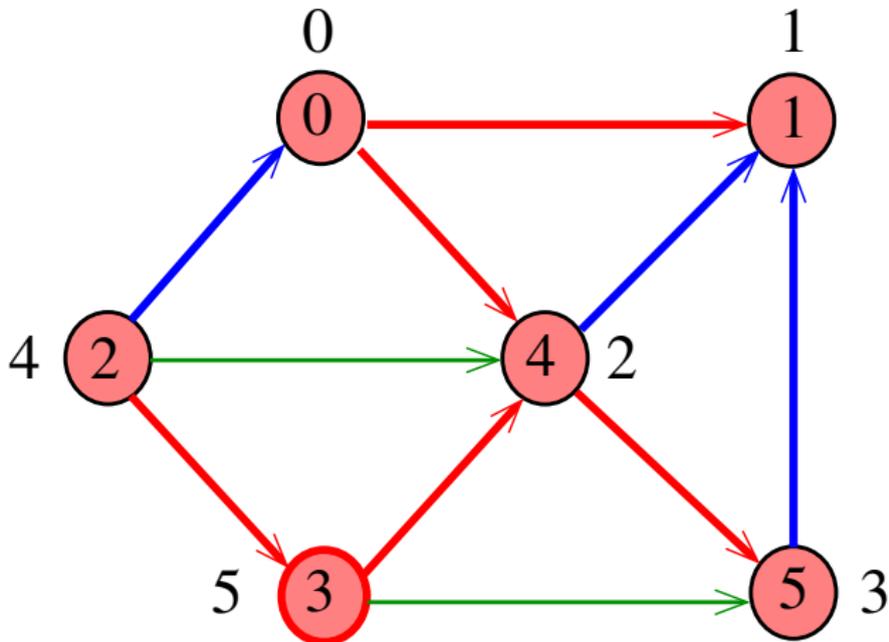
# Algoritmo DFS

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



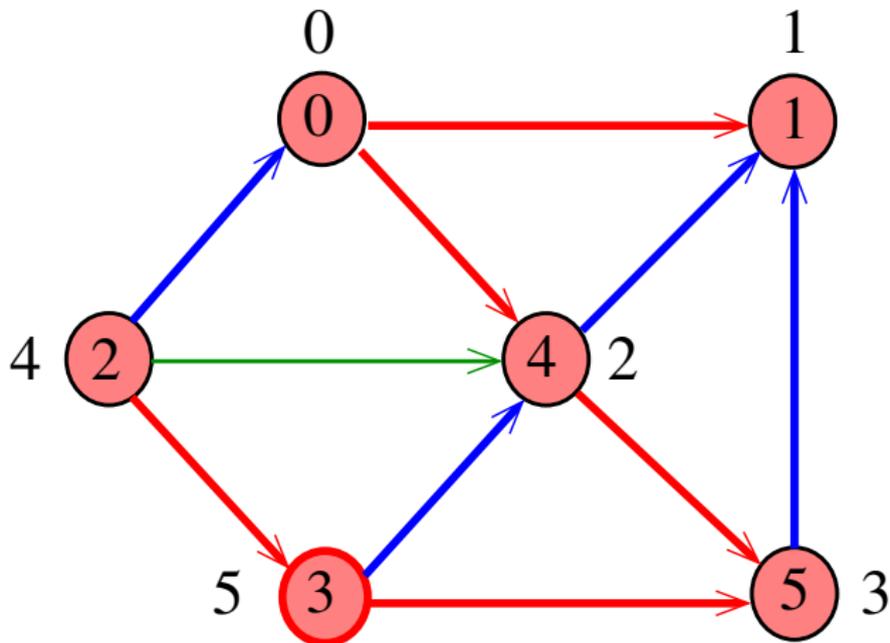
# Algoritmo DFS

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



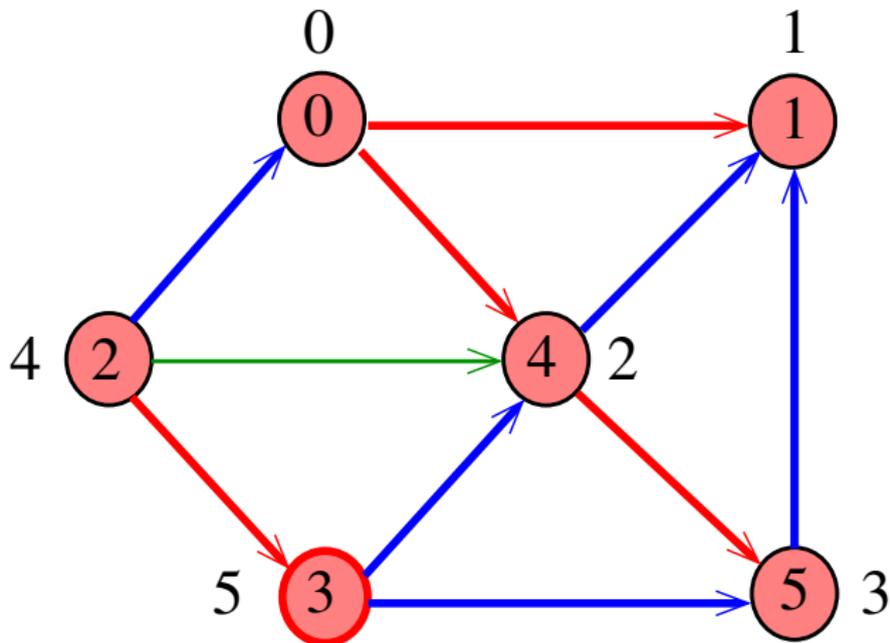
# Algoritmo DFS

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



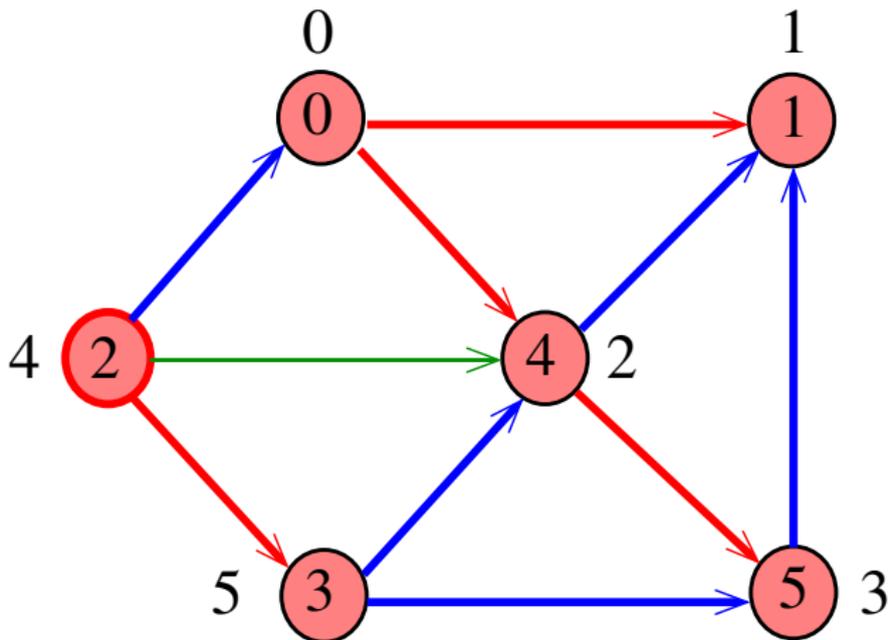
# Algoritmo DFS

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



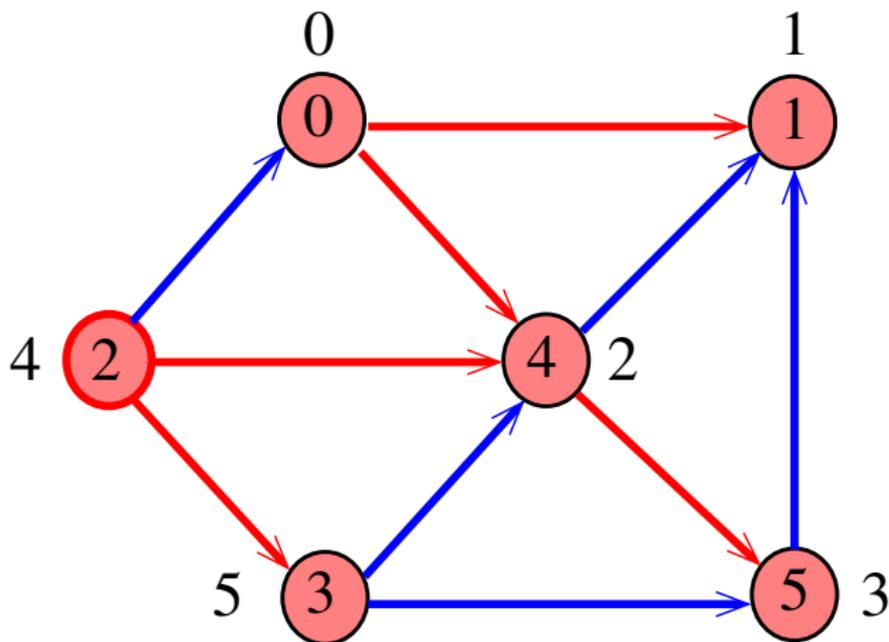
# Algoritmo DFS

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



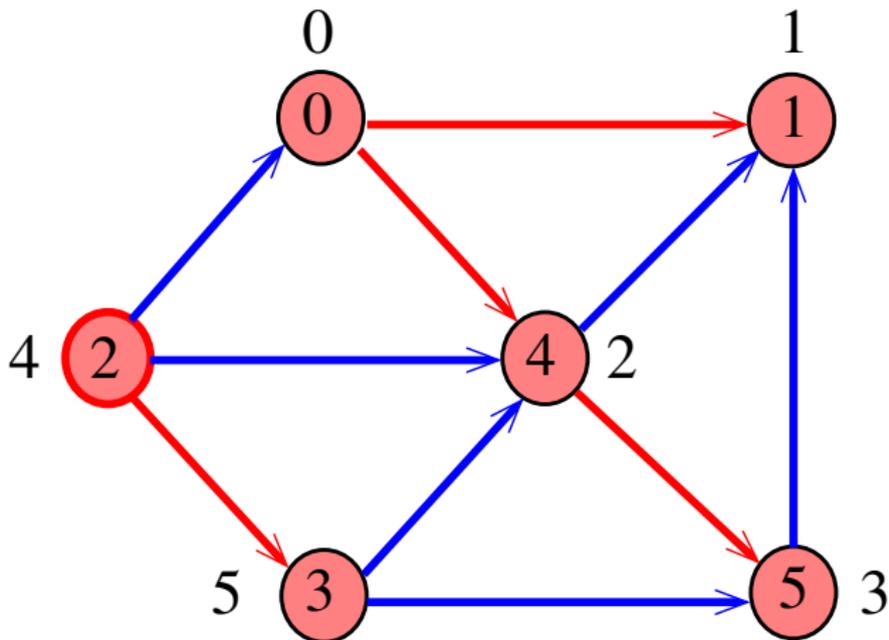
# Algoritmo DFS

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



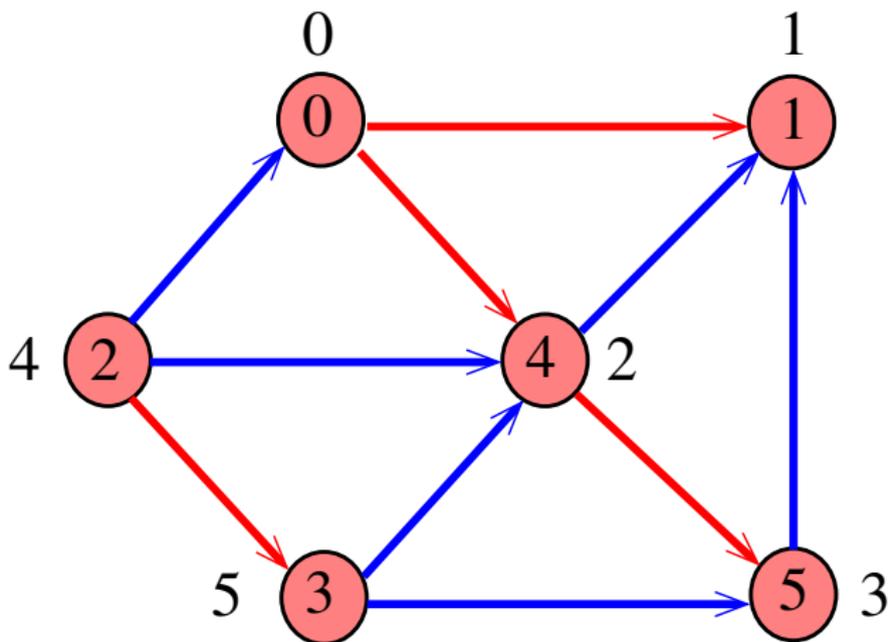
# Algoritmo DFS

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



# Algoritmo DFS

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



## Classe DFSTopological

A classe `DFSTopological` decide se um dado digrafo `G` é um DAG.

```
private Stack<Integer> ts
private Stack<Integer> cycle;
private boolean[] onPath;
```

Se `G` é um DAG uma ordenação topológica de seus vértices é armazenada em `ts`.

Se `G` não é um DAG, `cycle` armazenará um ciclo de `G`.

`onPath[v]` é true se o vértice `v` está no *caminho ativo*.

## DFStopological: esqueleto

```
public class DFStopological {
    private boolean[] marked;
    private int[] edgeTo;
    private boolean[] onPath;
    private Stack<Integer> ts;
    private Stack<Integer> cycle;
    private int onCycle= -1;
    public DFStopological(Graph G) {...}
    private void dfs(Digraph G, int v){...}
    public boolean isDag() {...}
    public boolean hasCycle() {...}
    public Iterable<Integer> order() {...}
    public Iterable<Integer> cycle() {...}
}
```

## DFStopological: construtor

**Determina** se um digrafo **G** é acíclico, e portanto seu vértices tem uma **ordem topológica**, ou tem um **ciclo**.

```
public DFStopological(Graph G) {
    marked = new boolean[G.V()];
    edgeTo = new int[G.V()];
    ts = new Stack<Integer>();
    onPath = new boolean[G.V()];
    for (int v = 0; v < G.V(); v++)
        if (!marked[v] && onCycle == -1) {
            dfs(G,v);
        }
}
```

## DFStopological: dfs()

```
private void dfs(Digraph G, int v) {
    marked[v] = true; onPath[v] = true;
    for (int w : G.adj(v)) {
        if (onCycle != -1) return;
        if (!marked[w]) {
            edgeTo[w] = v;
            dfs(G, w);
        } else if (onPath[w]) {
            onCycle = v;
            edgeTo[v] = w; // fecha o ciclo
        }
    }
    onPath[v] = false; ts.push(v);
}
```

DFStopological: hasCycle(), isDag()

```
// G contém um ciclo ?  
public boolean hasCycle() {  
    return onCycle != -1;  
}  
  
// G é um DAG ?  
public boolean isDag() {  
    return onCycle == -1;  
}
```

## DFStopological: cycle()

Retorna um **ciclo** como iterável se **G** possui um ciclo ou **null** em caso contrário.

```
public Iterable<Integer> cycle() {
    if (onCycle == -1) return null;
    if (cycle != null) return cycle;
    cycle = new Stack<Integer>();
    for (int x=edgeTo[onCycle]; x!=onCycle;
         x = edgeTo[x]) {
        cycle.push(x);
    }
    cycle.push(onCycle);
    return cycle;
}
```

## DFStopological: order()

Retorna uma **ordem topológica** dos vértices de **G** como iterável, se **G** é um DAG.

```
public Iterable<Integer> order() {  
    if (onCycle != -1) return null;  
    return ts;  
}
```

## Consumo de tempo

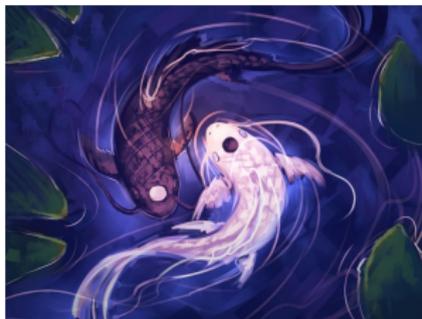
O consumo de tempo da função `DFStopological` para vetor de listas de adjacência é  $O(V + E)$ .

A classe `DFStopological`, para matriz de adjacências, consome tempo  $O(V^2)$ .

## Conclusão

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

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



Fonte: Avatar: The Last Airbender