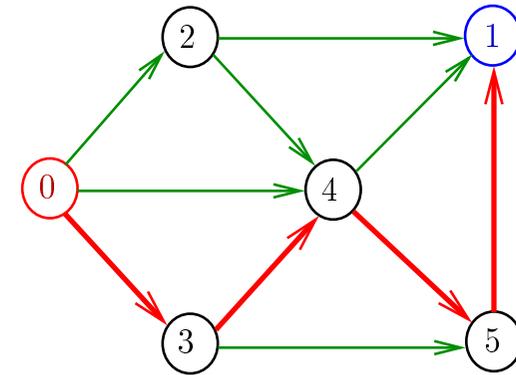


Na última aula...

Procurando um caminho

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

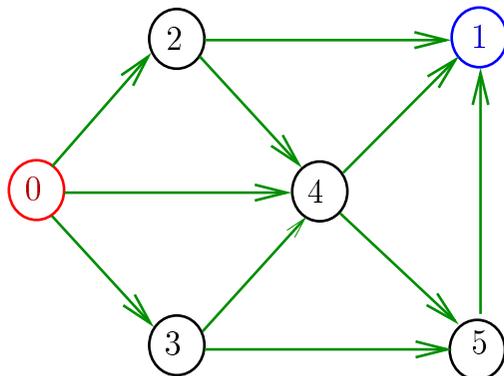
Exemplo: para $s = 0$ e $t = 1$ a resposta é **SIM**



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**



Certificados

Como é possível 'verificar' a resposta?

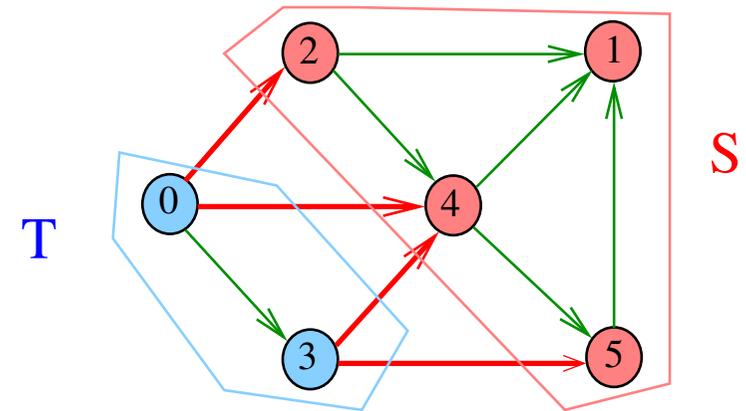
Como é possível 'verificar' que **existe** caminho?

Como é possível 'verificar' que **não existe** caminho?

Veremos questões deste tipo freqüentemente

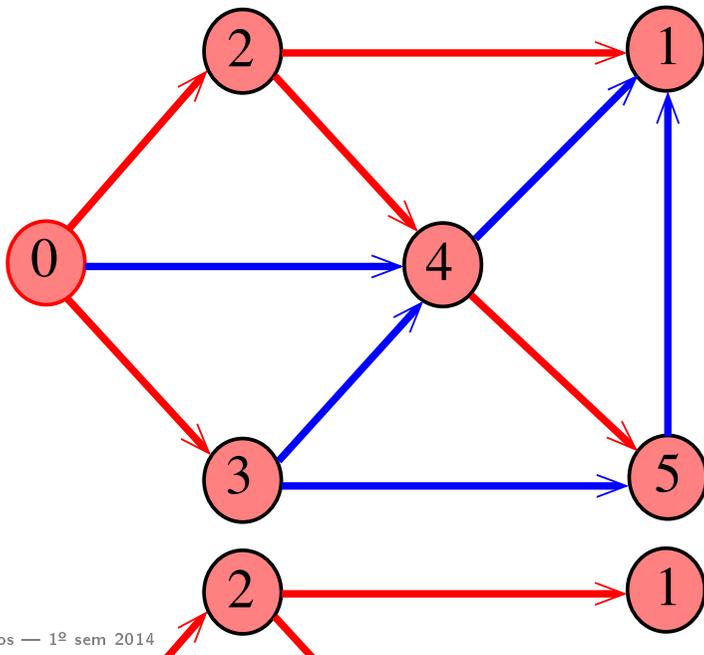
Certificado de inexistência

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



Arborescências

Exemplo: a raiz da arborescência é 0

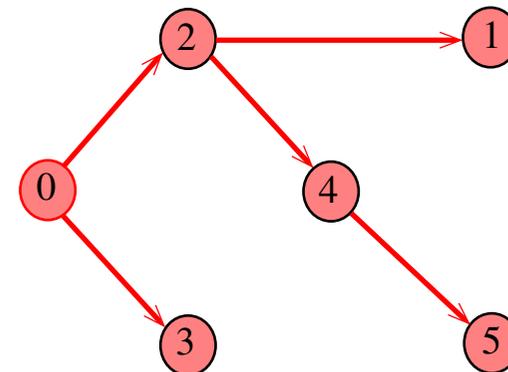


Certificado de inexistência

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

Arborescências no computador

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



vértice	parnt
0	0
1	2
2	0
3	0
4	2
5	4

Teorema

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 .

Hoje

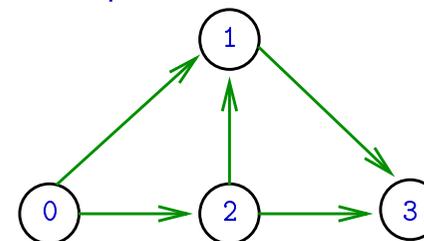
Vetor de listas de adjacência

S 17.4

Vetor de listas de adjacência de digrafos

Na representação de um digrafo através de **listas de adjacência** tem-se, para cada vértice v , uma lista dos vértices que são vizinhos v .

Exemplo:



0: 1, 2
1: 3
2: 1, 3
3:

Consumo de espaço: $\Theta(V + A)$

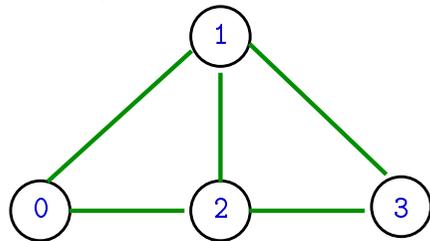
Manipulação eficiente

(linear)

Vetor de lista de adjacência de grafos

Na representação de um grafo através de **listas de adjacência** tem-se, para cada vértice v , uma lista dos vértices que são pontas de arestas incidentes a v

Exemplo:



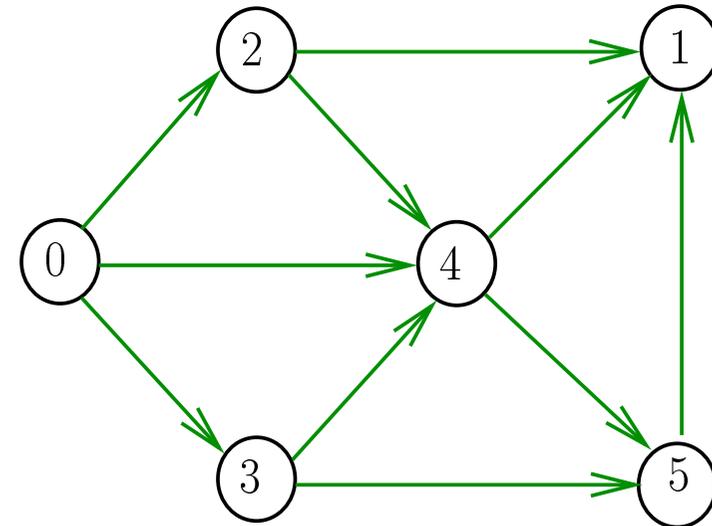
0: 1, 2
 1: 3, 0, 2
 2: 1, 3, 0
 3: 1, 2

Consumo de espaço: $\Theta(V + A)$
 Manipulação eficiente

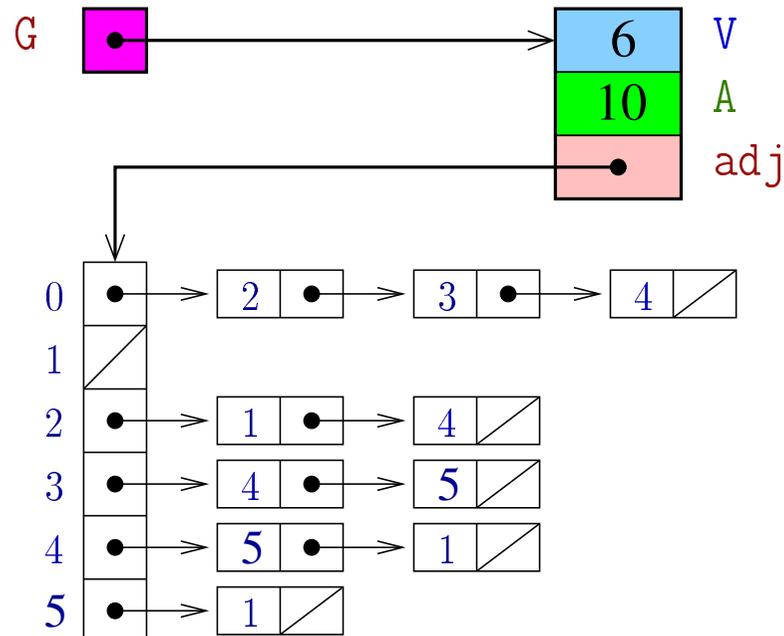
(linear)

Digrafo

Digraph G



Estruturas de dados



Estrutura digraph

A estrutura **digraph** representa um digrafo
 V contém o número de vértices
 A contém o número de arcos do digrafo
 adj é um ponteiro para vetor de listas de adjacência

```
struct digraph {
    int V;
    int A;
    link *adj;
};
```

Estrutura Digraph

Um objeto do tipo `Digraph` contém o endereço de um `digraph`

```
typedef struct digraph *Digraph;
```

Estrutura node

A lista de adjacência de um vértice `v` é composta por nós do tipo `node`

Um `link` é um ponteiro para um `node`

Cada nó da lista contém um vizinho `w` de `v` e o endereço do nó seguinte da lista

```
typedef struct node *link;
struct node {
    Vertex w;
    link next;
};
```

isArc

Decidir se $v \rightarrow u$ é um arco é feito em tempo constante se G é dado por matriz de adjacência. Já, com listas:

```
int
isArc (Digraph G, Vertex v, Vertex u)
{
    link p = G->adj[v];
    while (p && p->w != u)
        p = p->next;
    return p;
}
```

Claramente $O(\text{grau de saída}(v)) = O(V)$.

Estrutura graph e Graph

Essa mesma estrutura será usada para representar grafos

```
#define graph digraph
#define Graph Digraph
```

O número de arestas de um grafo G é

$$(G \rightarrow A) / 2$$

DIGRAPHinit

Devolve (o endereço de) um novo digrafo com vértices $0, \dots, V-1$ e nenhum arco

```
Digraph DIGRAPHinit (int V) {
0   Vertex v;
1   Digraph G = malloc(sizeof *G);
2   G->V = V;
3   G->A = 0;
4   G->adj = malloc(V * sizeof(link));
5   for (v = 0; v < V; v++)
6       G->adj[v] = NULL;
7   return G;
}
```

Algoritmos em Grafos — 1º sem 2014

NEWv

Para criar novos arcos, `NEWv` recebe um vértice `u` e o endereço `next` de um nó e devolve (o endereço de) um novo nó `x` com

`x.w = u` e `x.next = next`

```
link NEWv (Vertex u, link next) {
    link p = malloc(sizeof *p);
    p->w = u;
    p->next = next;
    return p;
}
```

Algoritmos em Grafos — 1º sem 2014

21 / 1

DIGRAPHinsertA

Insere um arco `v-u` no digrafo `G`.

A responsabilidade de evitar laços e arcos paralelos é do cliente/usuário.

void

```
DIGRAPHinsertA (Digraph G, Vertex v, Vertex
u)
{
    G->adj[v] = NEWv(u, G->adj[v]);
    G->A++;
}
```

Algoritmos em Grafos — 1º sem 2014

22 / 1

DIGRAPHinsertA

Insere um arco `v→w` no digrafo `G`.

Se `v == w` ou o digrafo já tem arco `v-w`, não faz nada

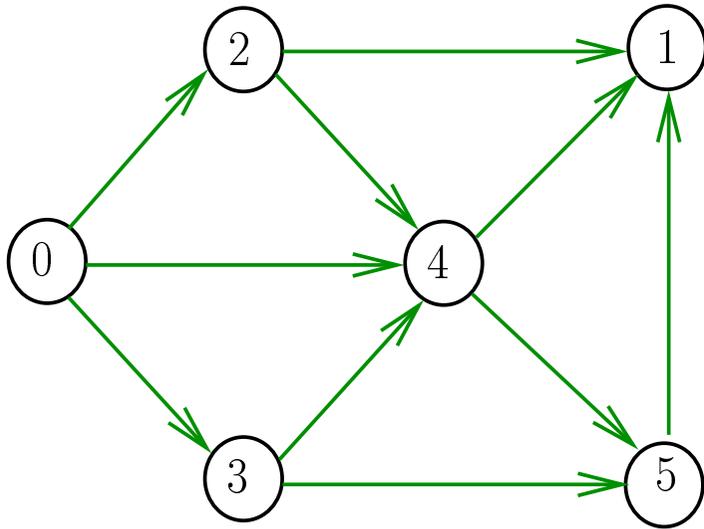
void

```
DIGRAPHinsertA (Digraph G, Vertex v, Vertex
w)
{
    link p;
    if (v == w || isArc(G, v, w)) return;
    G->adj[v] = NEWv(w, G->adj[v]);
    G->A++;
}
```

Algoritmos em Grafos — 1º sem 2014

23 / 1

24 / 1



```

0:  2  3  4
1:
2:  1  4
3:  4  5
4:  1  5
5:  1

```

```

void DIGRAPHshow (Digraph G) {
    Vertex v;
    link p;
    1 for (v = 0; v < G->V; v++) {
    2     printf("%2d:", v);
    3     for (p=G->adj[v]; p!= NULL;
    p=p->next)
    4         printf("%2d", p->w);
    5     printf("\n");
    }
}

```

Consumo de tempo

linha	número de execuções da linha
1	$= V + 1 = \Theta(V)$
2	$= V = \Theta(V)$
3	$= V + A = \Theta(V + A)$
4	$= A = \Theta(A)$
5	$= V = \Theta(V)$
total	$3\Theta(V) + \Theta(V + A) + \Theta(A) = \Theta(V + A)$

Conclusão

O consumo de tempo da função `DigraphShow` para **vetor de listas de adjacência** é $\Theta(V + A)$.

O consumo de tempo da função `DigraphShow` para **matriz adjacência** é $\Theta(V^2)$.

E para desenhar?

O pacote **Graphviz** (disponível para Linux e outros) tem programas que leem uma descrição de um grafo ou digrafo e desenham, com vários algoritmos alternativos.

Não é tópico deste curso, mas algoritmos para desenhar grafos são objeto de pesquisa atual.

Funções básicas para grafos

```
#define GRAPHinit DIGRAPHinit  
#define GRAPHshow DIGRAPHshow
```

Função que insere uma aresta **v-w** no grafo **G**

```
void  
GRAPHinsertE (Graph G, Vertex v, Vertex w)  
{  
    DIGRAPHinsertA(G, v, w);  
    DIGRAPHinsertA(G, w, v);  
}
```

Exercício. Escrever a função **GRAPHremoveE**

Busca DFS

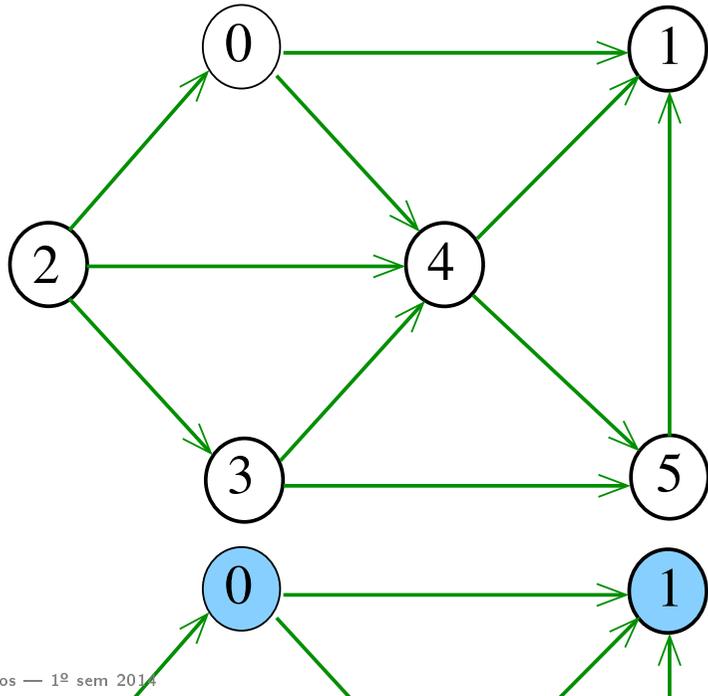
S 18.1 e 18.2

Busca ou varredura

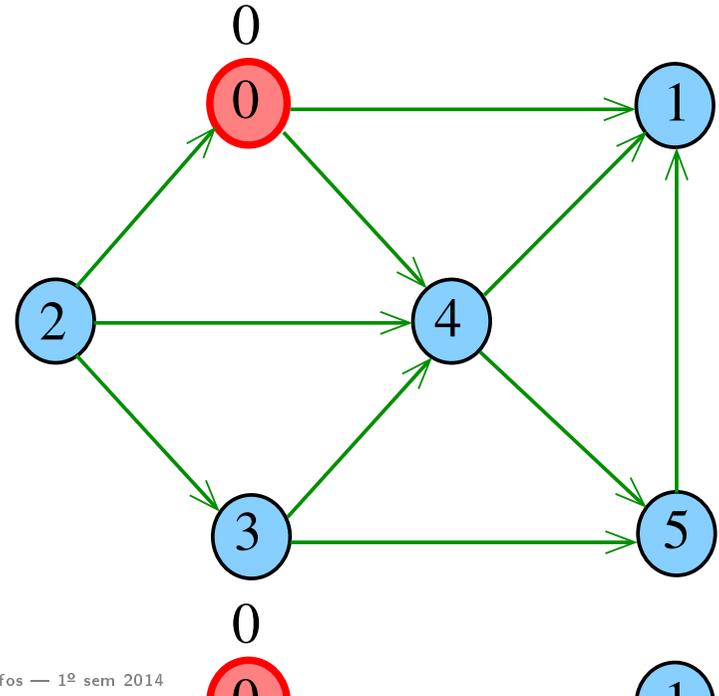
Um algoritmo de **busca** (ou **varredura**) examina, sistematicamente, todos os vértices e todos os arcos de um digrafo.

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

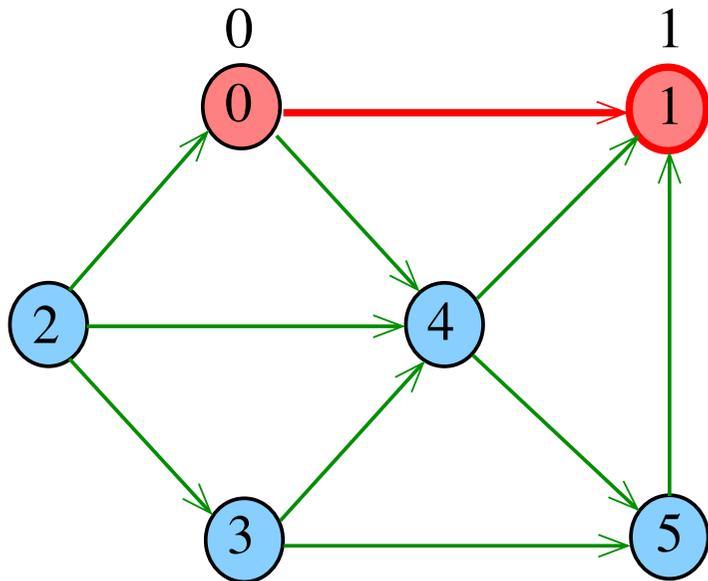
DIGRAPHdfs(G)



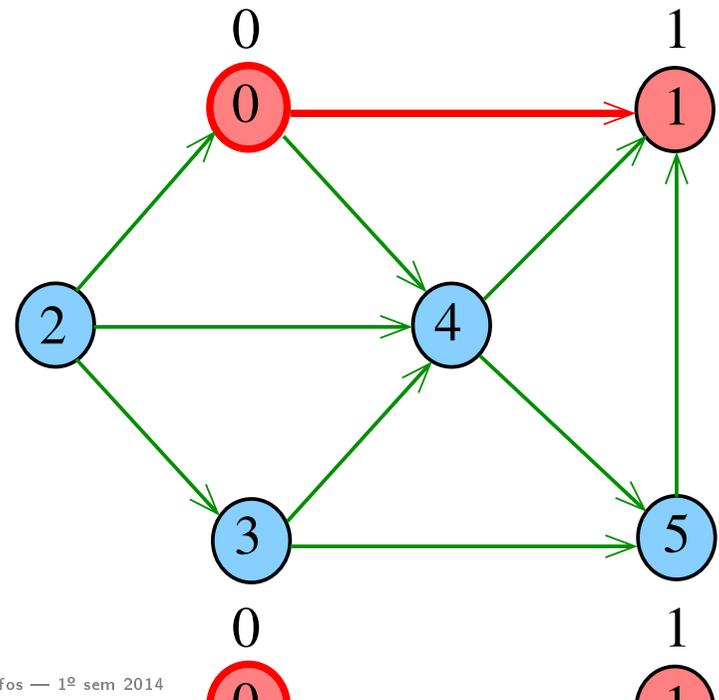
dfsR(G,0)



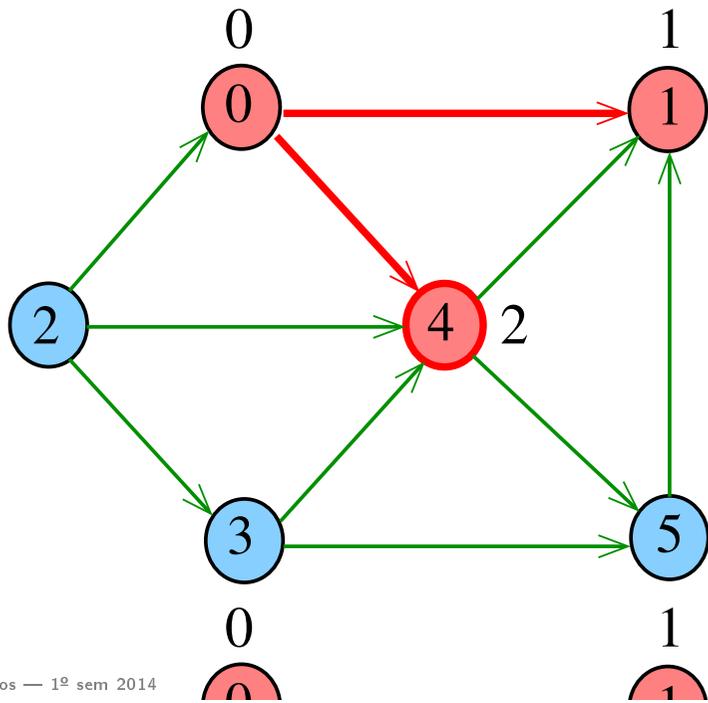
dfsR(G,1)



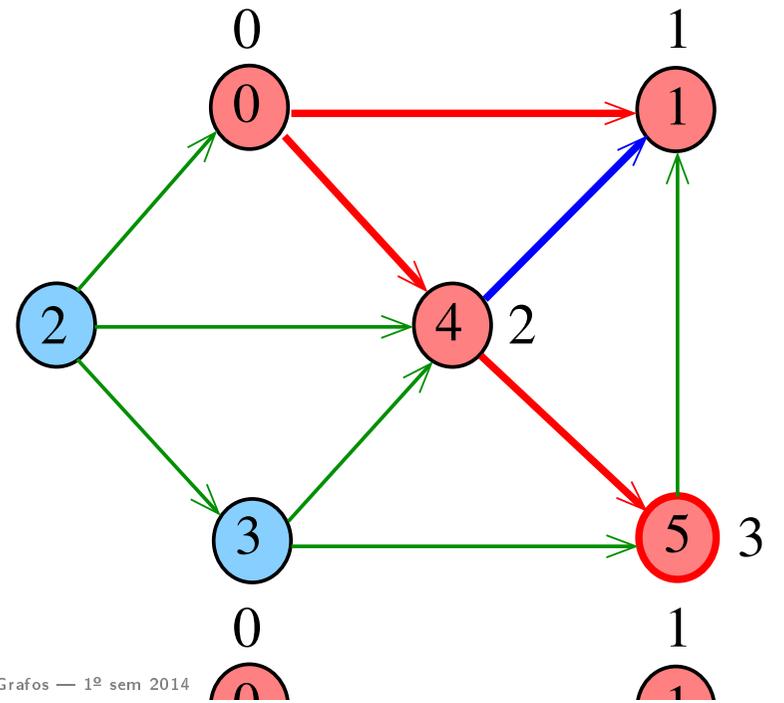
dfsR(G,0)



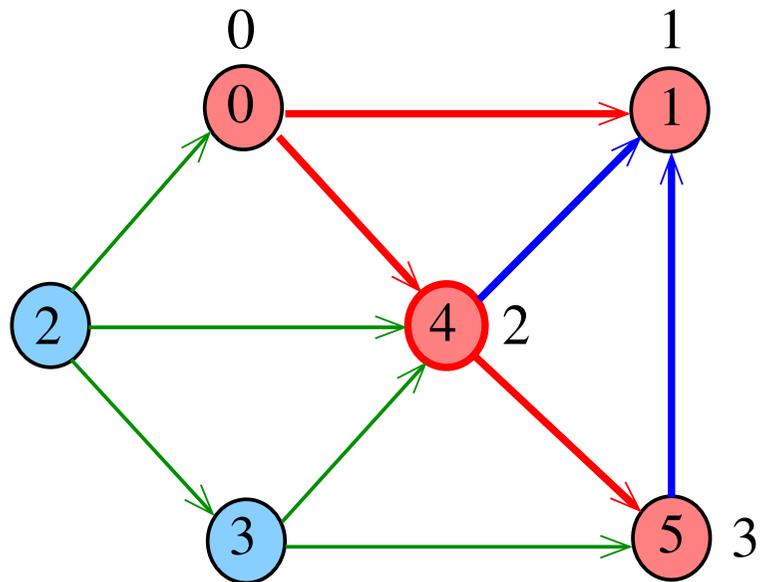
dfsR(G,4)



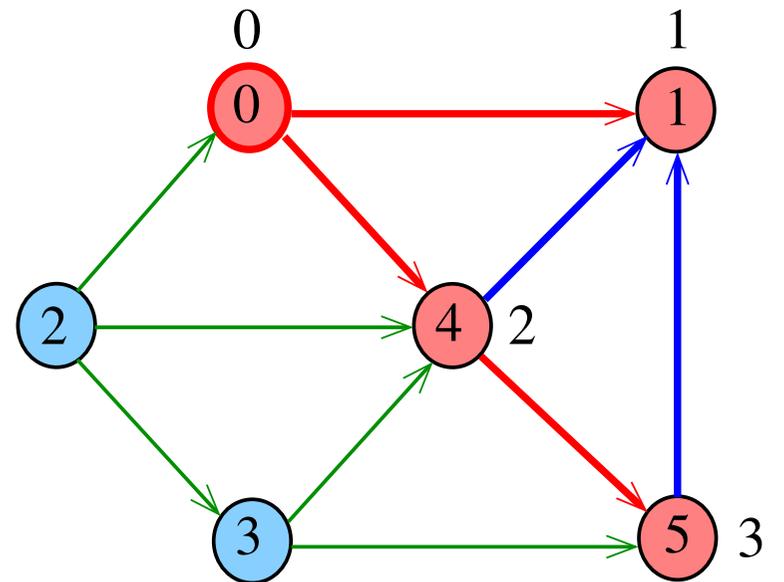
dfsR(G,5)



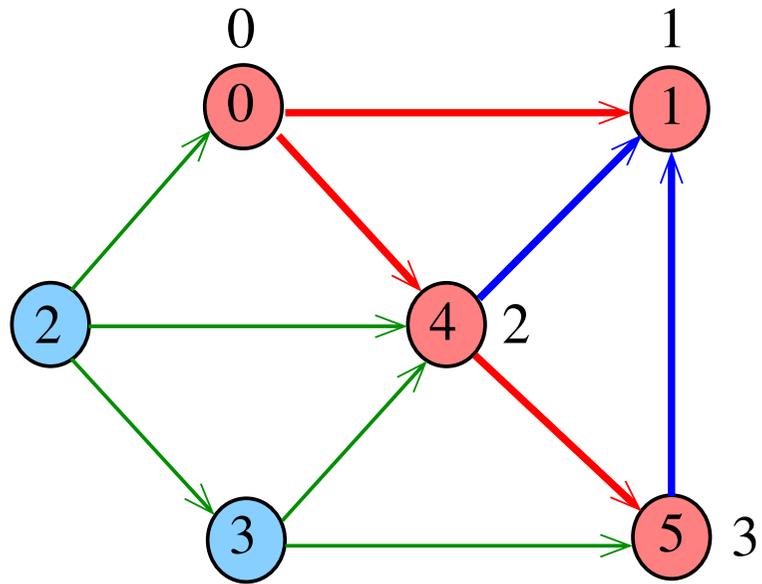
dfsR(G,4)



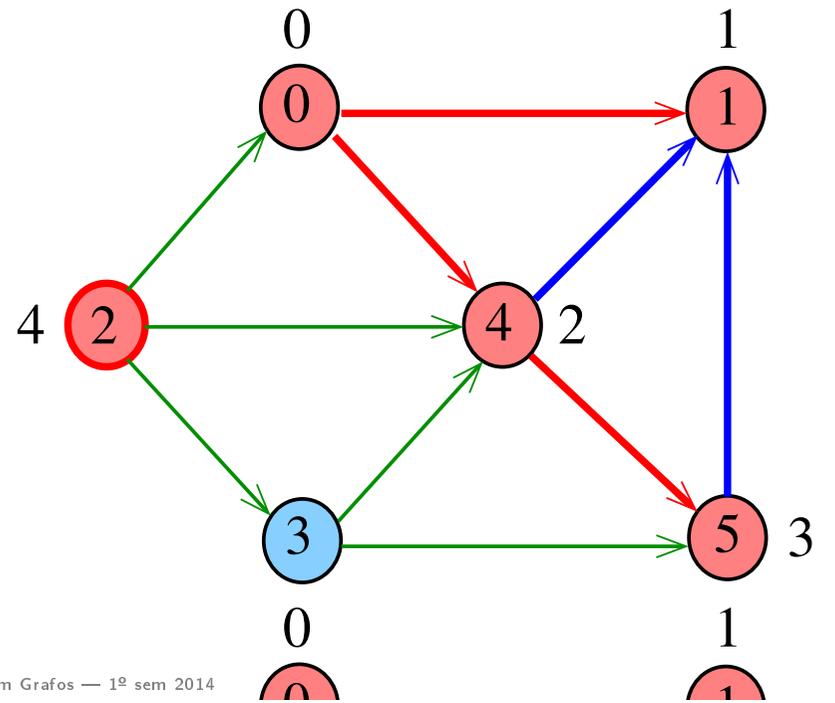
dfsR(G,0)



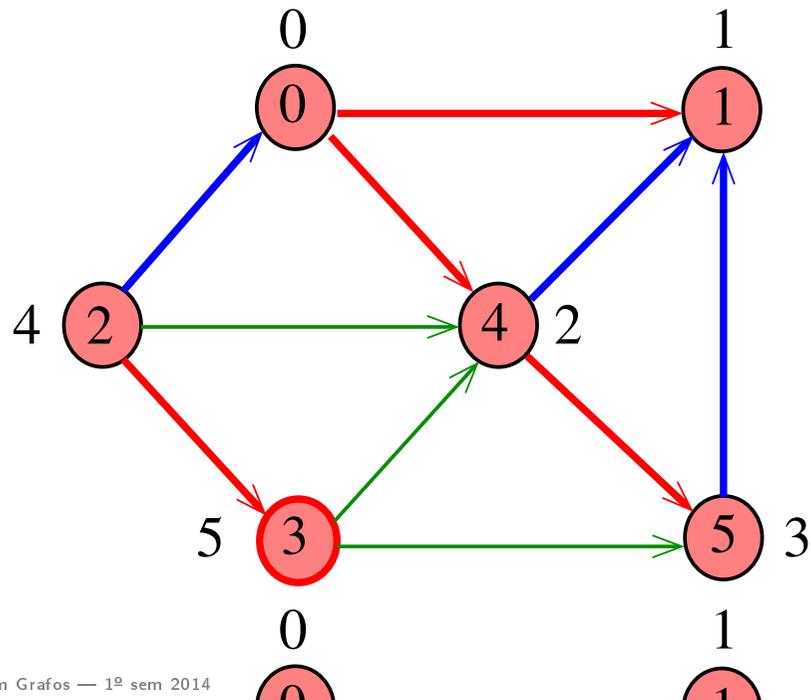
DIGRAPHdfs(G)



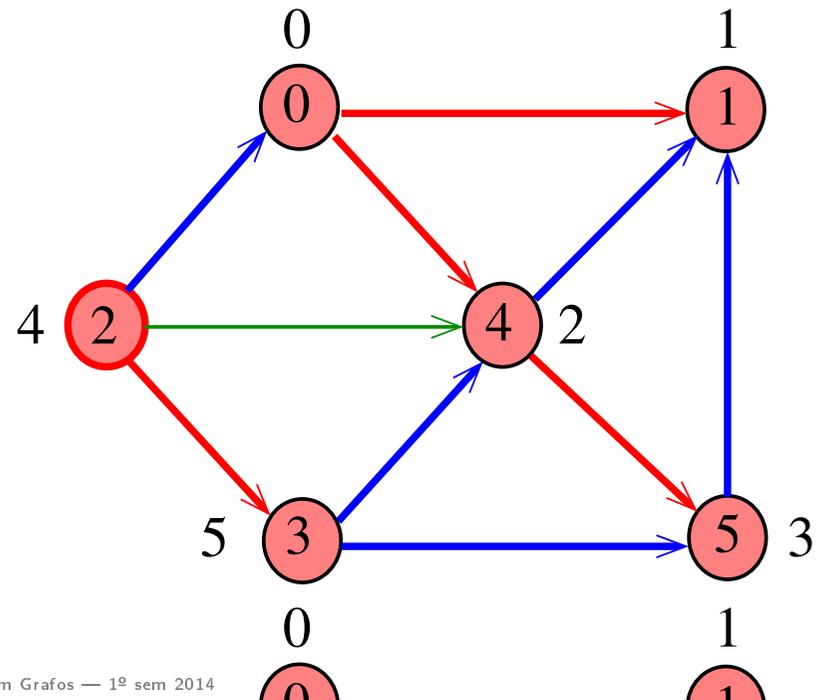
dfsR(G,2)



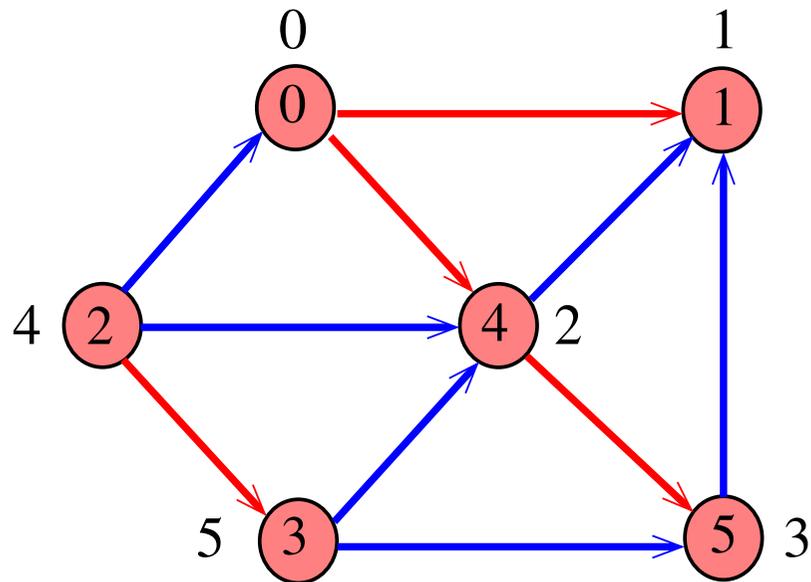
dfsR(G,3)



dfsR(G,2)



DIGRAPHdfs(G)



dfsR

dfsR supõe que o digrafo G é representado por uma matriz de adjacência

```
void dfsR (DigraphG, Vertex v) {
    Vertex w;
    1 lbl[v] = cnt++;
    2 for (w = 0; w < G->V; w++)
    3     if (G->adj[v][w])
    4         if (lbl[w] == -1)
    5             dfsR(G, w);
}
```

DIGRAPHdfs

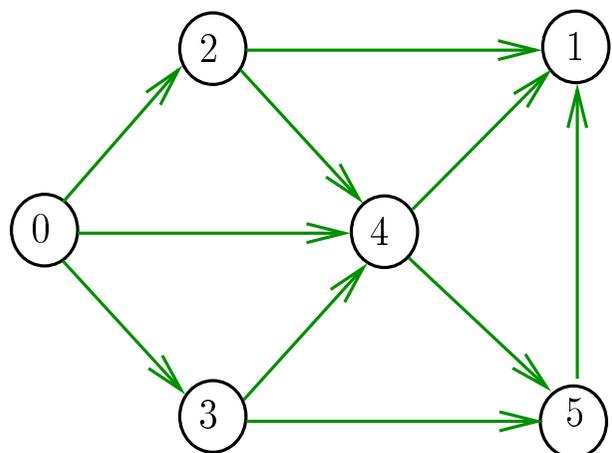
```
static int cnt, lbl[maxV];
void DIGRAPHdfs (Digraph G) {
    Vertex v;
    1 cnt = 0;
    2 for (v = 0; v < G->V; v++)
    3     lbl[v] = -1;
    4 for (v = 0; v < G->V; v++)
    5     if (lbl[v] == -1)
    6         dfsR(G, v);
}
```

dfsR

dfsR supõe que o digrafo G é representado por listas de adjacência

```
void dfsR (Digraph G, Vertex v) {
    link p;
    1 lbl[v] = cnt++;
    2 for (p = G->adj[v]; p != NULL; p = p->next)
    3     if (lbl[p->w] == -1)
    4         dfsR(G, p->w);
}
```

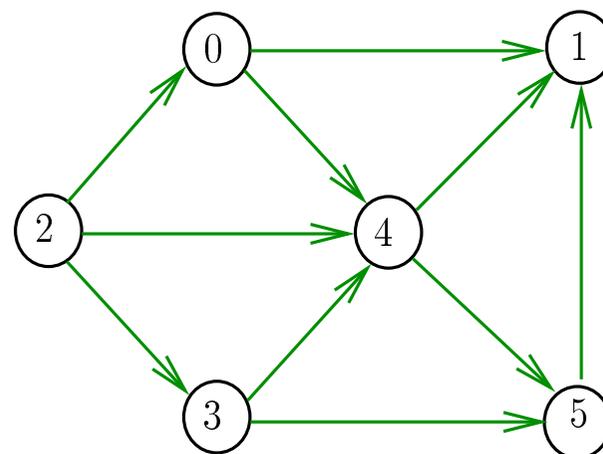
DIGRAPHdfs(G)



```

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

DIGRAPHdfs(G)



```

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

Consumo de tempo

O consumo de tempo da função **DIGRAPHdfs** para **vetor de listas de adjacência** é $\Theta(V + A)$.

O consumo de tempo da função **DIGRAPHdfs** para **matriz de adjacência** é $\Theta(V^2)$.

Arborescência de busca em profundidade

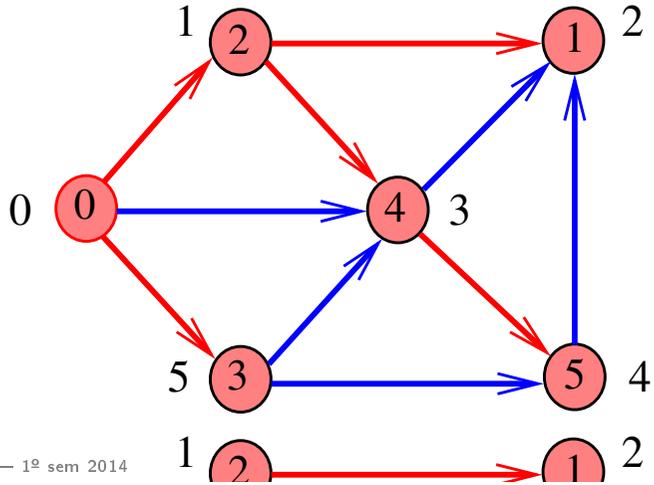
Classificação dos arcos

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

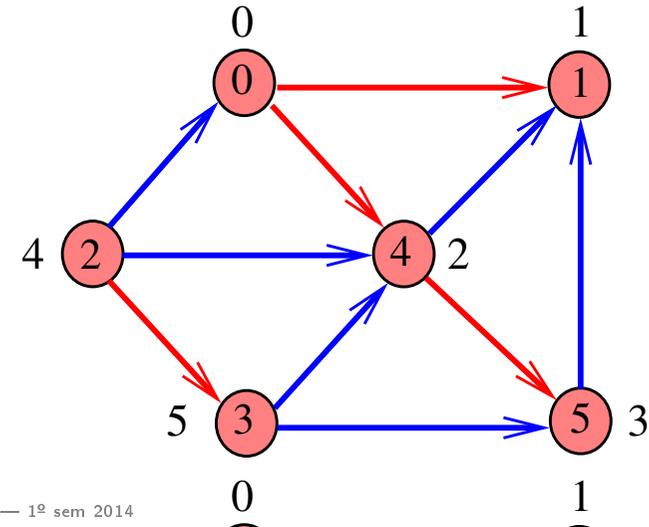
Exemplo: arcos em **vermelho** são arcos da arborescência



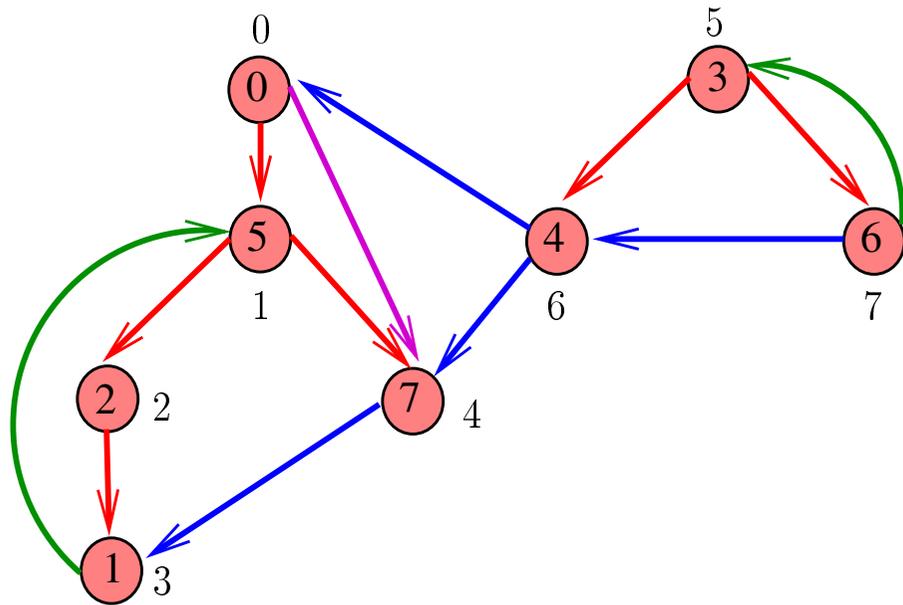
Floresta DFS

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

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

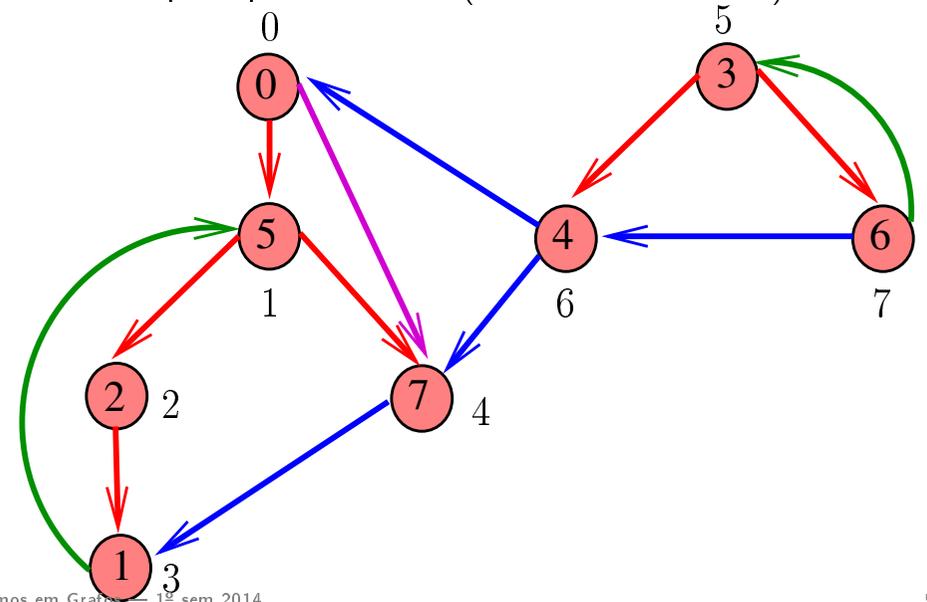


Classificação dos arcos



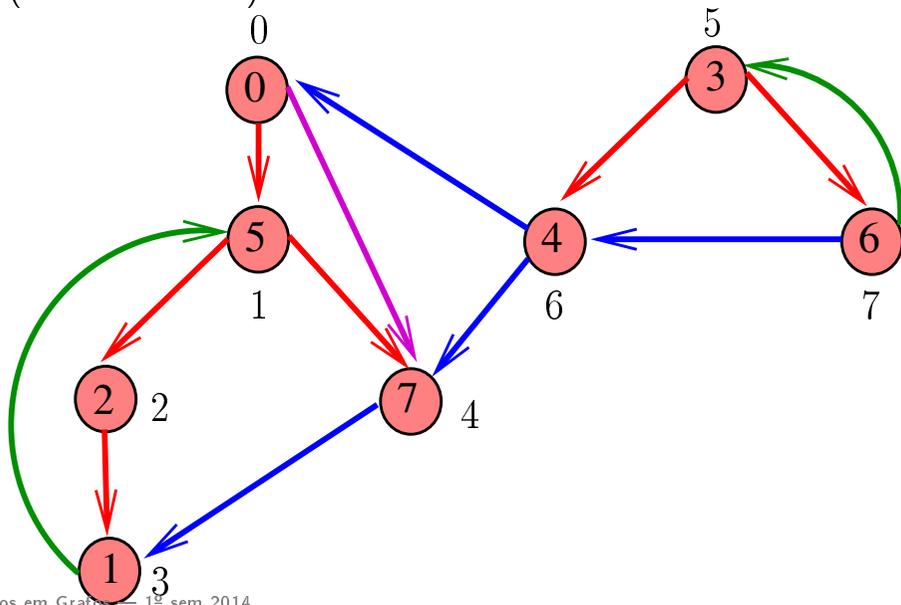
Arcos de arborescência

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



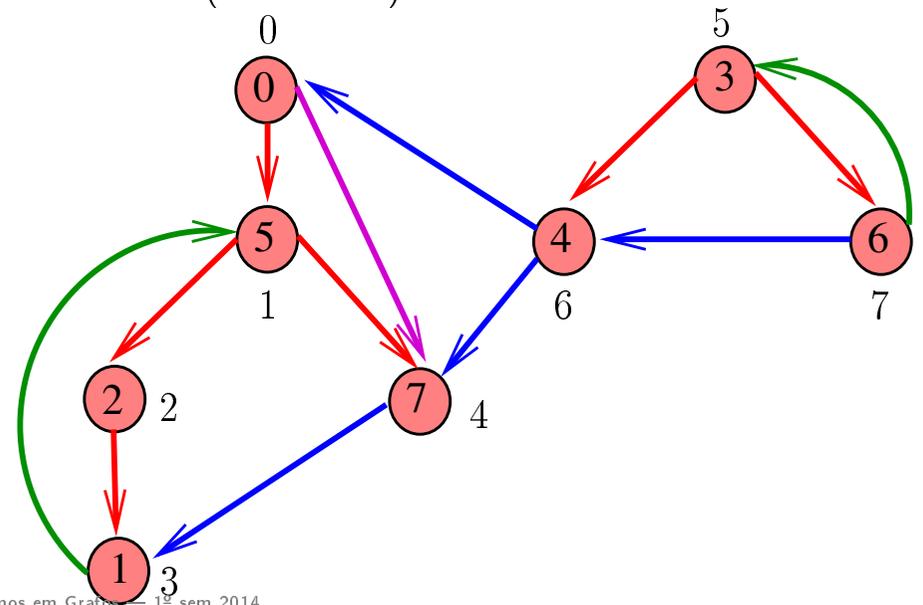
Arcos de retorno

$v-w$ 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**)

