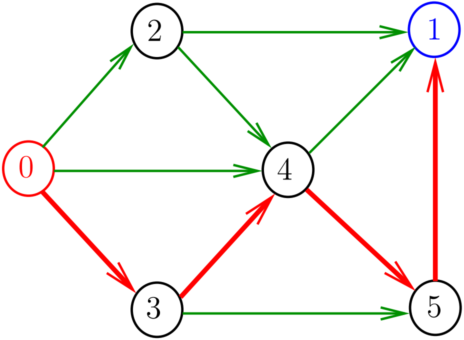
Melhores momentos

AULA 4

Procurando um caminho

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

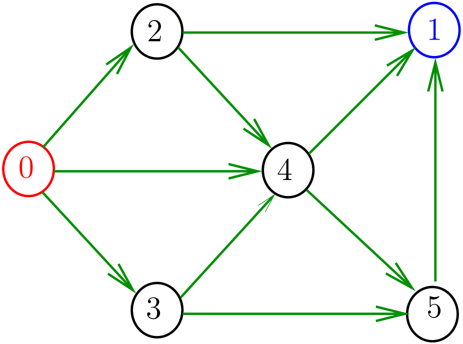
Exemplo: para s = 0 e t = 1 a resposta é SIM



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

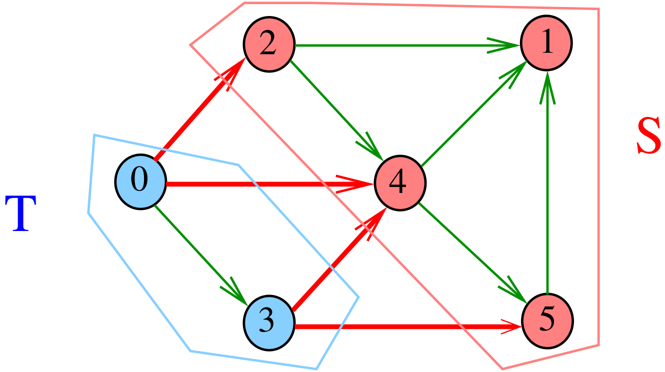
Certificado de inexistência

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

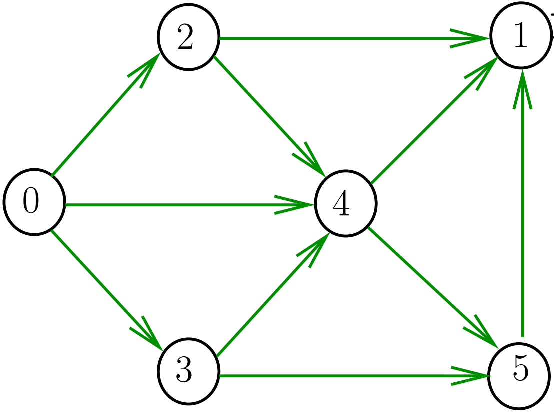
> ***todo arco*** *no corte tem ponta inicial em T e ponta final em S*

Certificado de inexistência

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

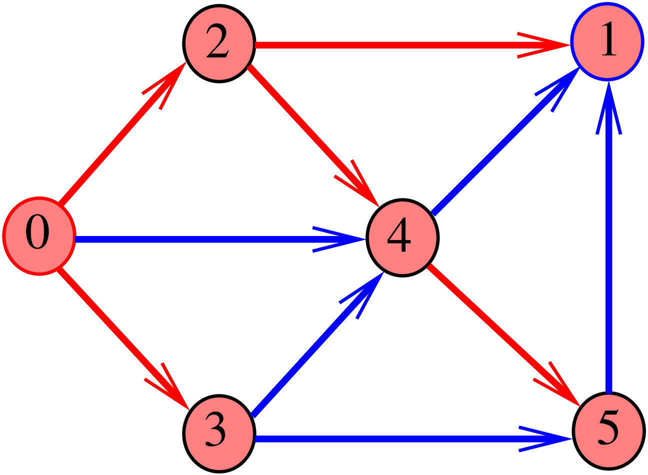


Certificado de existência



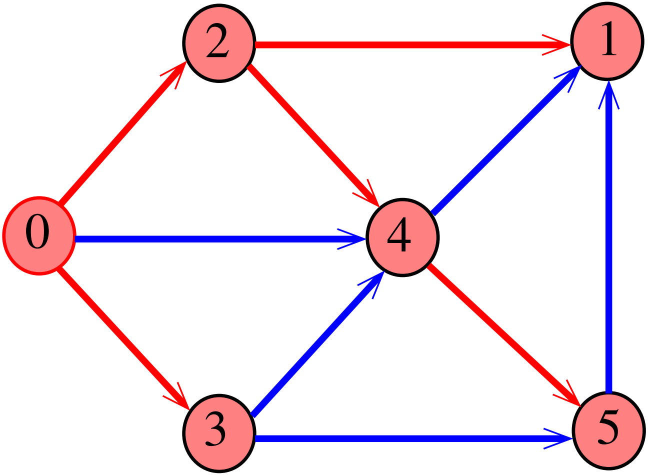

```
pathR(G, 0)
  0-2 pathR(G,2)
    2-1 pathR(G,1)
    2-4 pathR(G,4)
      4-1
      4-5 pathR(G,5)
      5-1
  0-3 pathR(G,3)
    3-4
    3-5
  0-4
existe caminho
```

DIGRAPHpath(G,0,1)



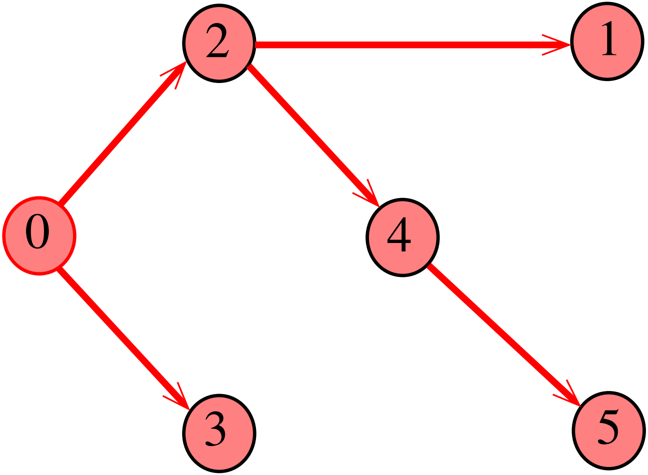
Arborescências

Exemplo: a raiz da arborescência é 0



Arborescências

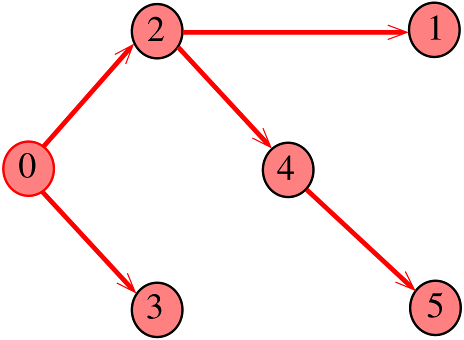
Exemplo: a raiz da arborescência é 0



Arborescências no computador

Um arborência pode ser representada através de um **vetor de pais**: `parnt[w]` é o pai de `w`

Se `r` é a raiz, então `parnt[r]=r`



vértice	parnt
0	0
1	2
2	0
3	0
4	2
5	4

Conclusão

Para quaisquer vértices $\mathtt{s}$ e $\mathtt{t}$ de um digrafo, vale uma e apenas umas das seguintes afirmações:

- existe um caminho de $\mathtt{s}$ a $\mathtt{t}$
- existe $\mathtt{st}$-corte $(\mathtt{S}, \mathtt{T})$ em que todo arco no corte tem ponta inicial em $\mathtt{T}$ e ponta final em $\mathtt{S}$.

AULA 5

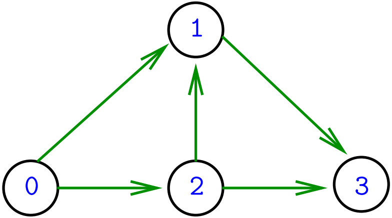
Vetor de listas de adjacência

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:
```

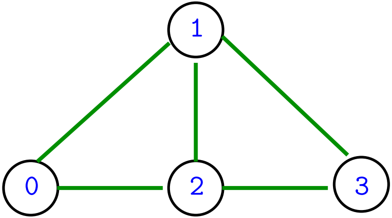
Consumo de espaço: $\Theta(V + A)$ (linear)

Manipulação eficiente

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:



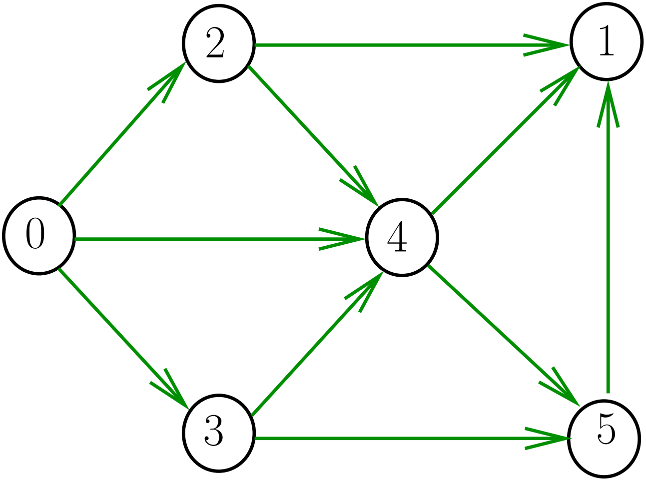

```
0:  1,  2
1:  3,  0,  2
2:  1,  3,  0
3:  1,  2
```

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

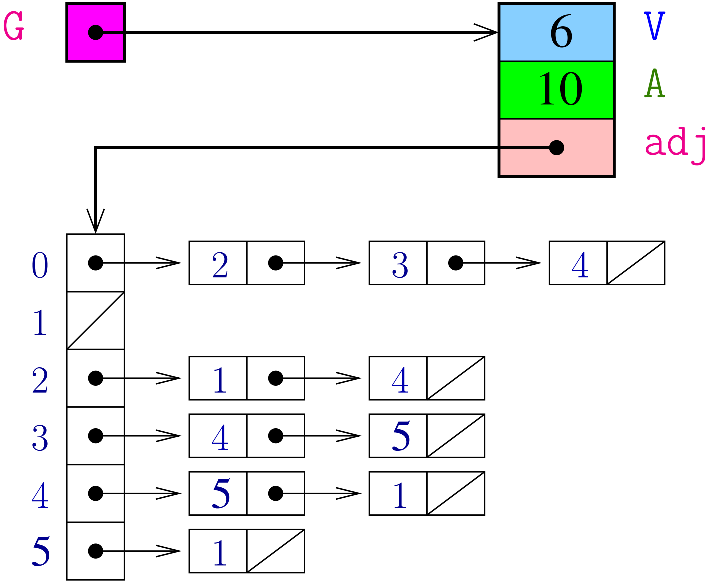
Manipulação eficiente

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;
};
```

NEW

`NEW` recebe um vértice `w` e o endereço `next` de um nó e devolve (o endereço de) um novo nó x com

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

```
link NEW (Vertex w, link next) {
```

NEW

NEW recebe um vértice w e o endereço next de um nó e devolve (o endereço de) um novo nó x com

x.w = w e x.next = next

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

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->A)/2`

DIGRAPHinit

Devolve (o endereço de) um novo digrafo com vértices `0,..,V-1` e nenhum arco

```
Digraph DIGRAPHinit (int V) {
```

DIGRAPHinit

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

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

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)
```

DIGRAPHinsertA

Insere um arco v-w no digrafo G.

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

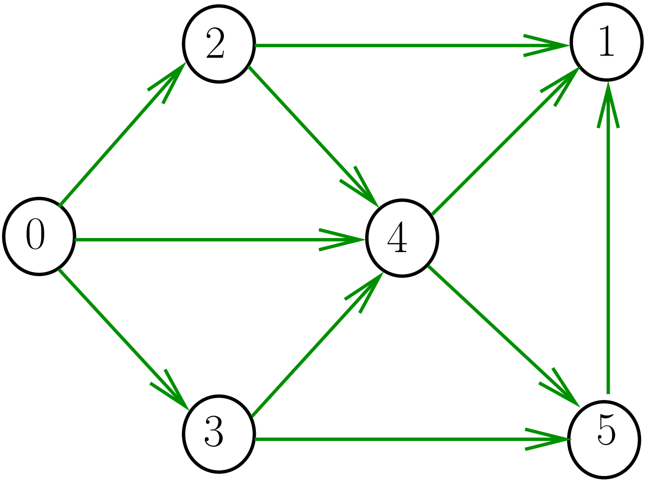
```
void
DIGRAPHinsertA (Digraph G, Vertex v, Vertex w)
{
   link p;
   if (v == w) return;
   for (p = G->adj[v]; p != NULL; p = p->next)
       if (p->w == w) return;
   G->adj[v] = NEW(w, G->adj[v]);
   G->A++;
}
```

DIGRAPHinsertA

O código abaixo transfere a responsabilidade de evitar laços e arcos paralelos ao cliente/usuário

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

DIGRAPHshow




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

DIGRAPHshow

```
void DIGRAPHshow (Digraph G) {
```

DIGRAPHshow

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

Consumo de tempo

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

Conclusão

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

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

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)
```

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

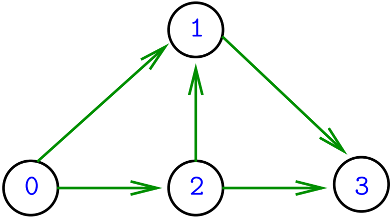
Melhores momentos

AULA 5

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:



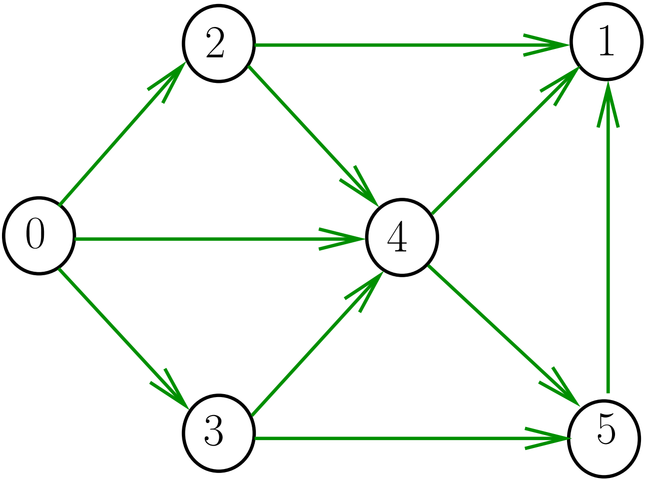

```
0:  1,  2
1:  3
2:  1,  3
3:
```

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

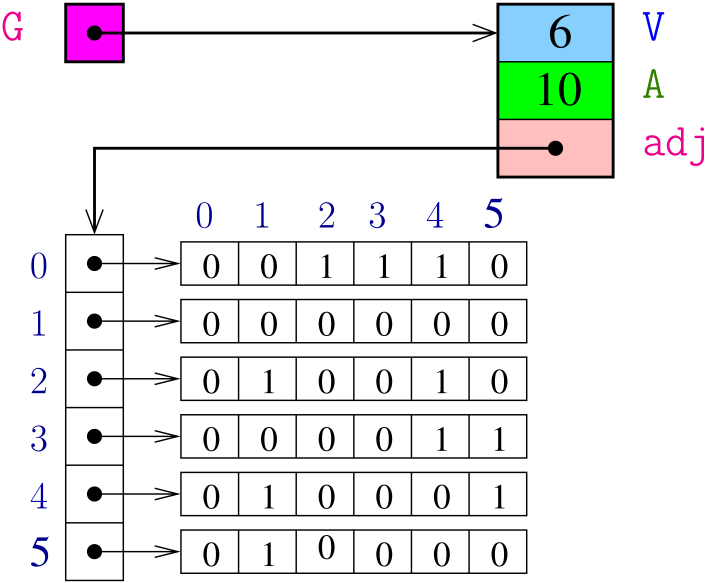
Manipulação eficiente

Digrafo

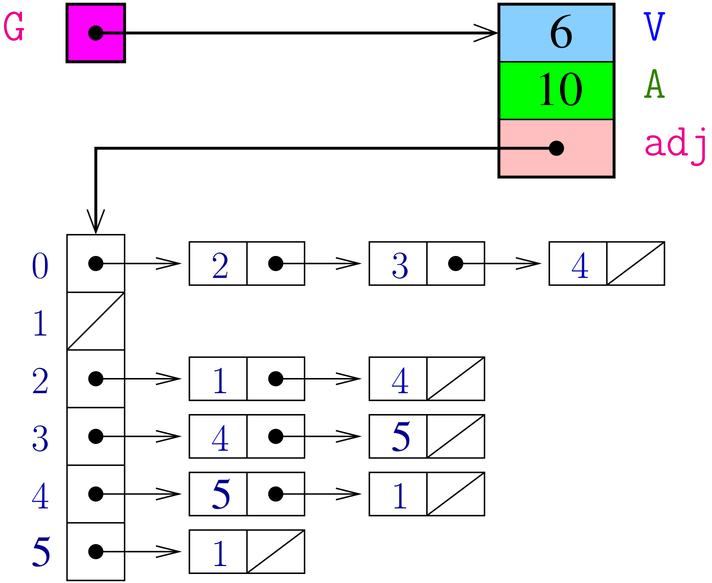
`Digraph G`



Matriz de adjacência



Listas de adjacência



AULA 6

Vetor de listas de adjacência (continuação)

S 17.4

DIGRAPHpath

Recebe um digrafo `G` e vértices `s` e `t` e devolve **1** se existe um caminho de `s` a `t` ou devolve **0** em caso contrário
Supõe que o digrafo tem no máximo `maxV` vértices.

```
int DIGRAPHpath (Digraph G, Vertex s, Vertex t)
```

DIGRAPHpath

```
static int lbl[maxV], static Vertex parnt[maxV];
int DIGRAPHpath (Digraph G,Vertex s,Vertex t)
{
    Vertex v;
    for (v = 0; v < G->V; v++) {
        lbl[v] = -1;
        parnt[v] = -1;
    }
    parnt[s] = s;
    pathR(G,s)
    if (lbl[t] == -1) return 0;
    else return 1;
}
```

pathR

```
void pathR (Digraph G, Vertex v)
{
    Vertex w;
    lbl[v] = 0;
    for (w = 0; w < G->V; w++)
        if (G->adj[v][w] == 1)
            if (lbl[w] == -1) {
                parnt[w] = v;
                pathR(G, w);
            }
}
```

pathR

```
void pathR (Digraph G,Vertex v)
{
    link p;
    lbl[v] = 0;
    for (p=G->adj[v]; p != NULL; p=p->next)
        if (lbl[p->w] == -1) {
            parnt[p->w] = v;
            pathR(G, p->w);
        }
}
```

Consumo de tempo

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

Consumo de tempo

Qual é o consumo de tempo da função DIGRAPHpath?

linha	número de execuções da linha	
1	$= \mathtt{V} + 1$	$= \Theta(\mathtt{V})$
2	$= \mathtt{V}$	$= \Theta(\mathtt{V})$
3	$= 1$	$= ????$
4	$= 1$	$= \Theta(1)$
5	$= 1$	$= \Theta(1)$

total $= 2\,\Theta(1) + 2\,\Theta(\mathtt{V}) + ???$

$= \Theta(\mathtt{V}) + ????$

Conclusão

O consumo de tempo da função `DIGRAPHpath` é $\Theta(\texttt{V})$ mais o consumo de tempo da função `PathR`.

Consumo de tempo

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

Consumo de tempo

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

linha	número de execuções da linha	
1	$\leq \texttt{V}$	$= \mathrm{O}(\texttt{V})$
2	$\leq \texttt{V} + \texttt{A}$	$= \mathrm{O}(\texttt{V} + \texttt{A})$
3	$\leq \texttt{A}$	$= \mathrm{O}(\texttt{A})$
4	$\leq \texttt{V} - 1$	$= \mathrm{O}(\texttt{V})$
5	$\leq \texttt{V} - 1$	$= \mathrm{O}(\texttt{V})$

total $= 3\,\mathrm{O}(\texttt{V}) + \mathrm{O}(\texttt{A}) + \mathrm{O}(\texttt{V} + \texttt{A})$

$= \mathrm{O}(\texttt{V} + \texttt{A})$

Conclusão

O consumo de tempo da função `PathR` para vetor de listas de adjacência é $O(\mathtt{V} + \mathtt{A})$.

Conclusão

O consumo de tempo da função `DIGRAPHpath` para vetor de listas de adjacência é $O(\mathtt{V} + \mathtt{A})$.

O consumo de tempo da função `DIGRAPHpath` para matriz de adjacência é $O(\mathtt{V}^2)$.

Busca DFS

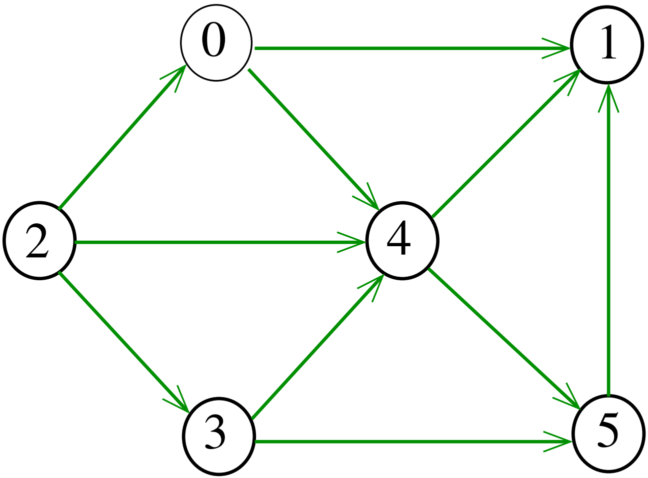
S 18.1 e 18.2

Busca ou varredura

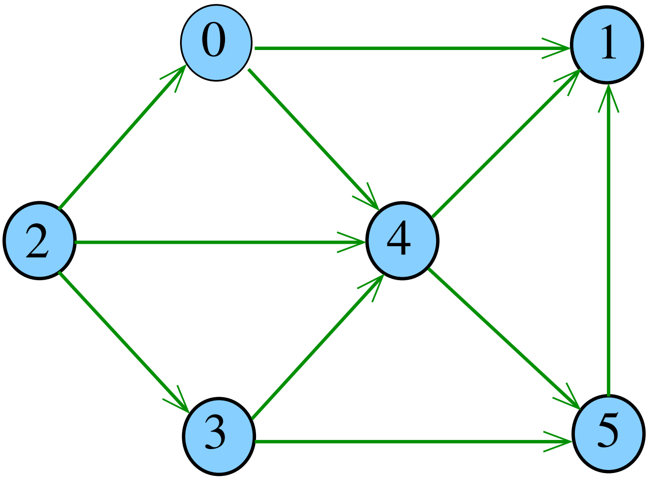
Um algoritimo de **busca** (ou **varredura**) examina, sistematicamente, todos os vértices e todos os arcos de um digrafo.

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

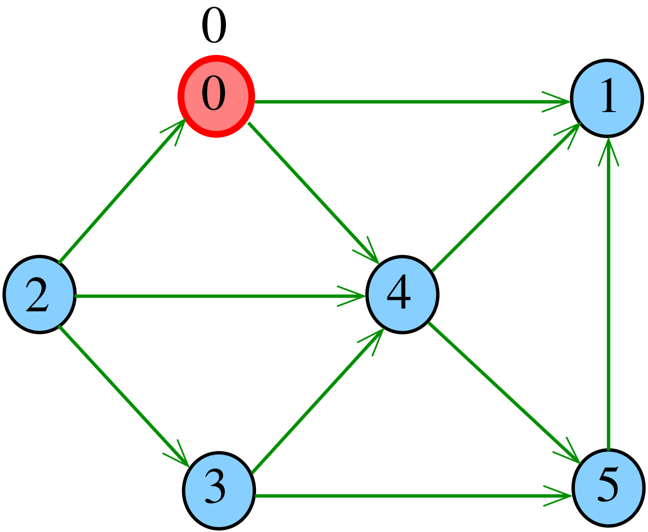
DIGRAPHdfs(G)



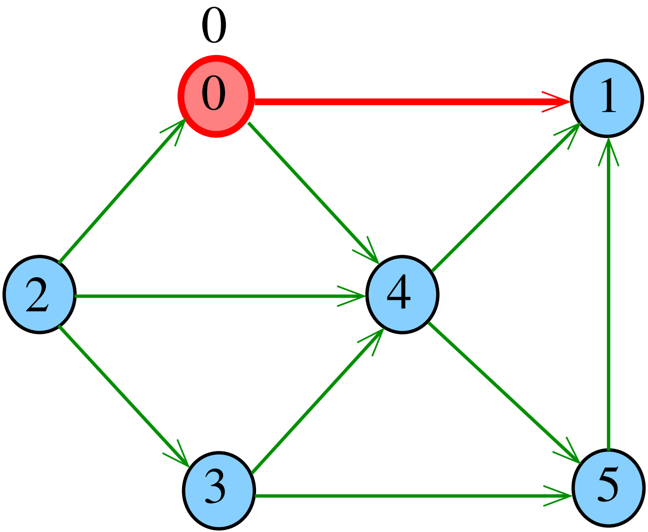
DIGRAPHdfs(G)



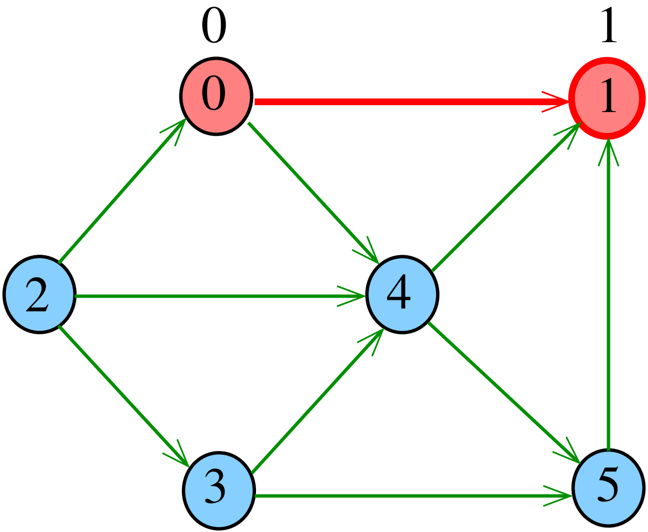
dfsR(G,0)



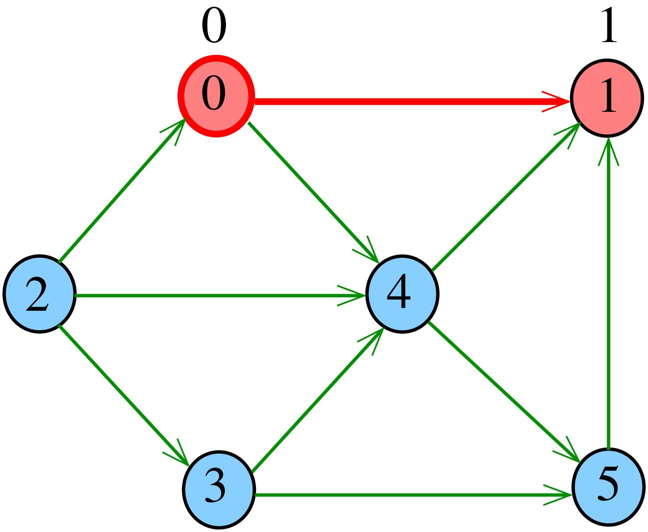
dfsR(G,0)



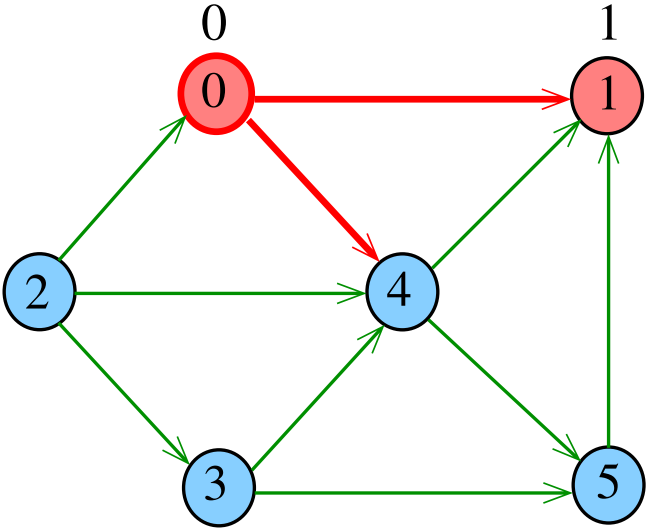
dfsR(G,1)



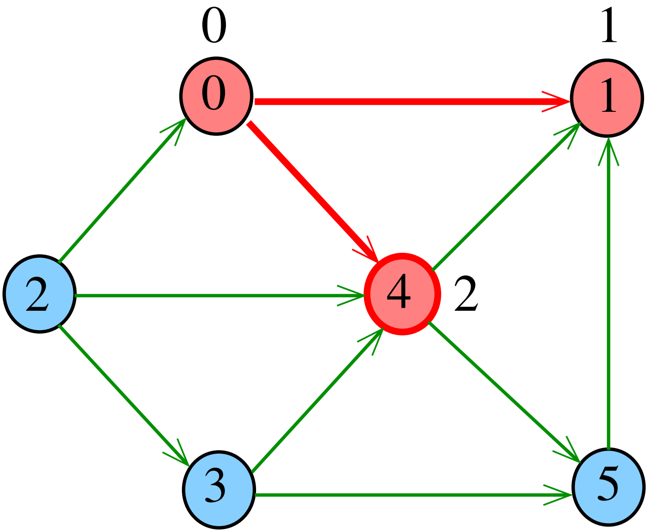
dfsR(G,0)



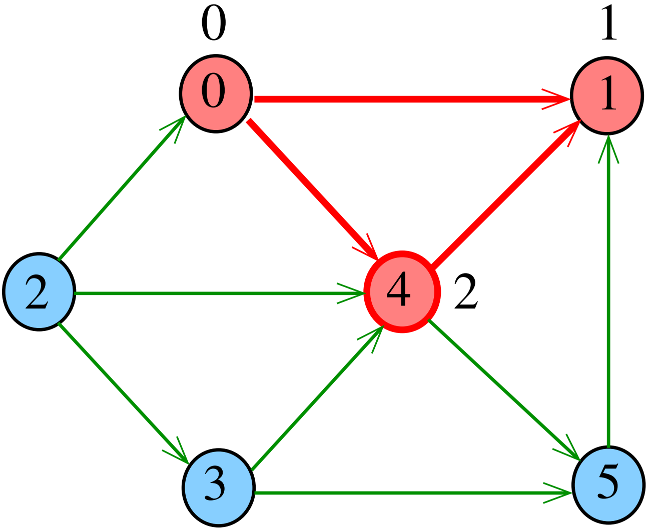
dfsR(G,0)



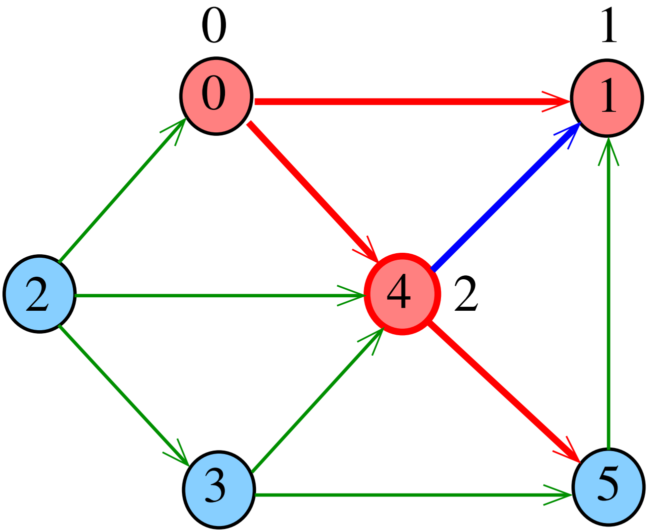
dfsR(G,4)



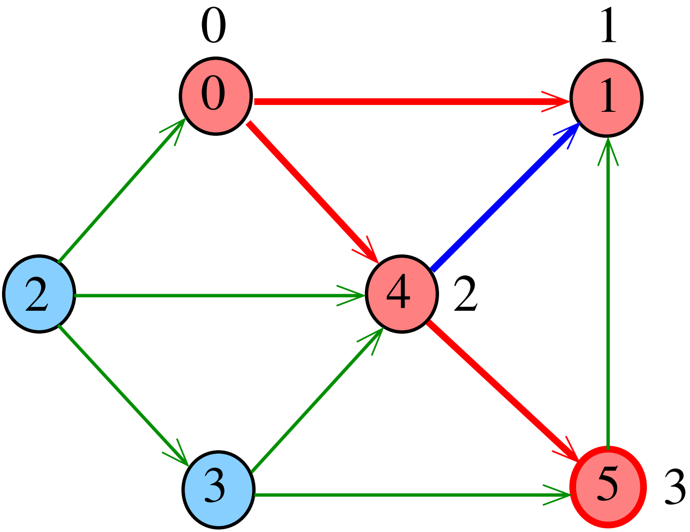
dfsR(G,4)



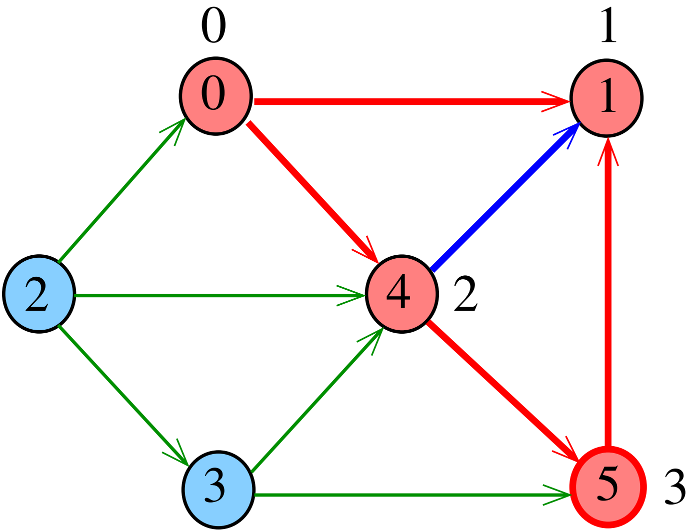
dfsR(G,4)



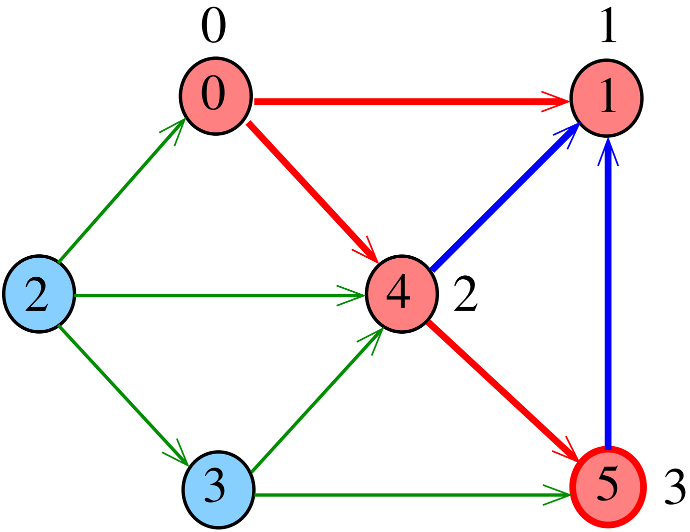
dfsR(G,5)



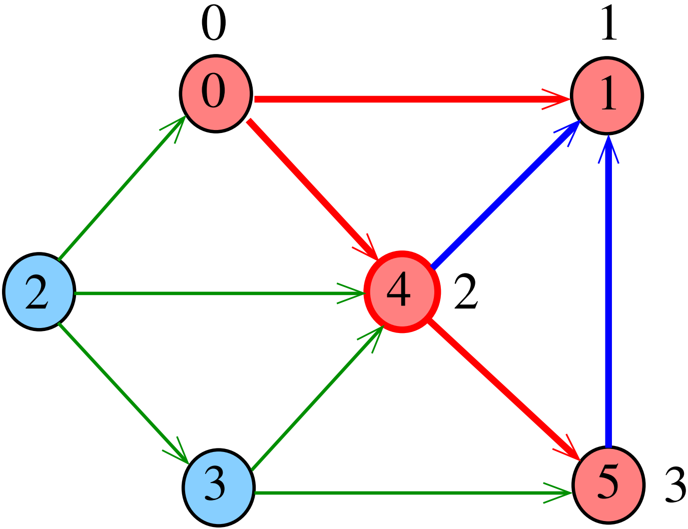
dfsR(G,5)



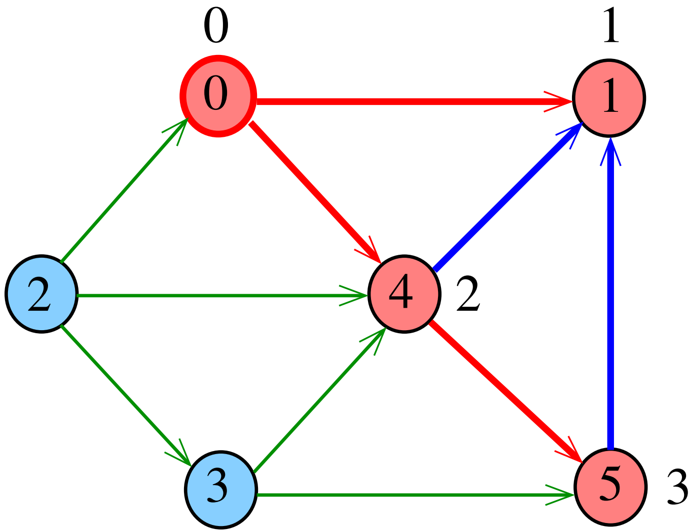
dfsR(G,5)



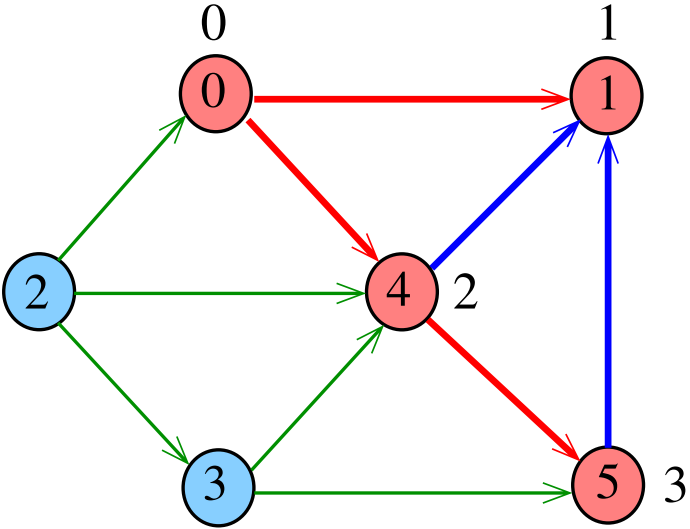
dfsR(G,4)



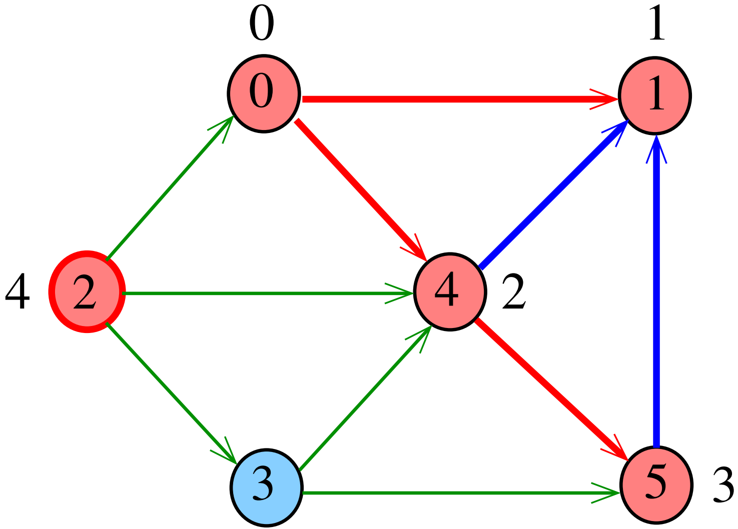
dfsR(G,0)



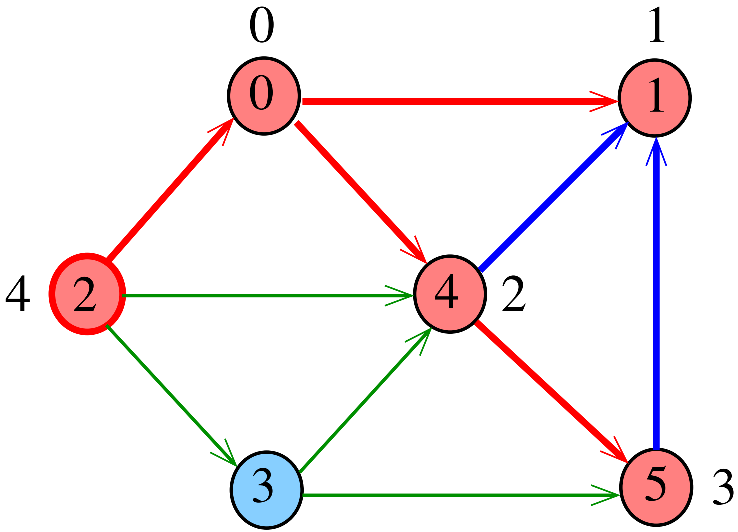
DIGRAPHdfs(G)



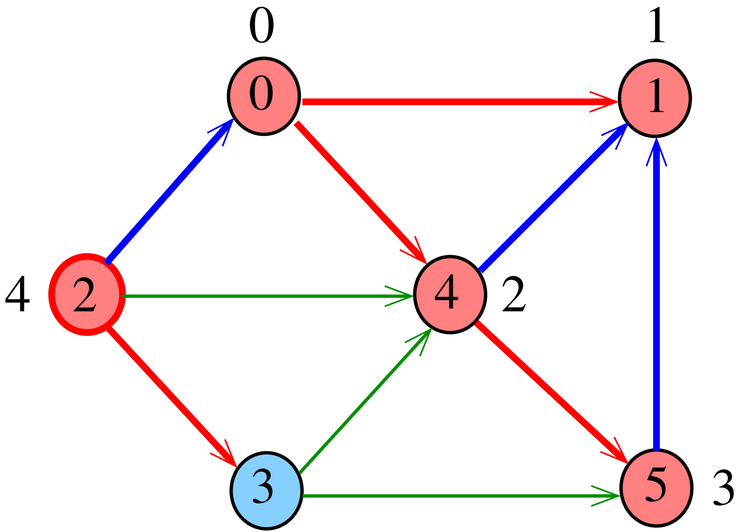
dfsR(G,2)



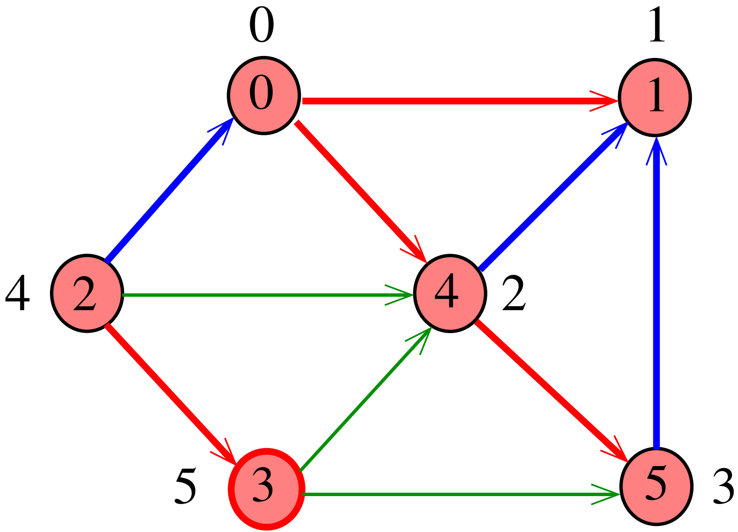
dfsR(G,2)



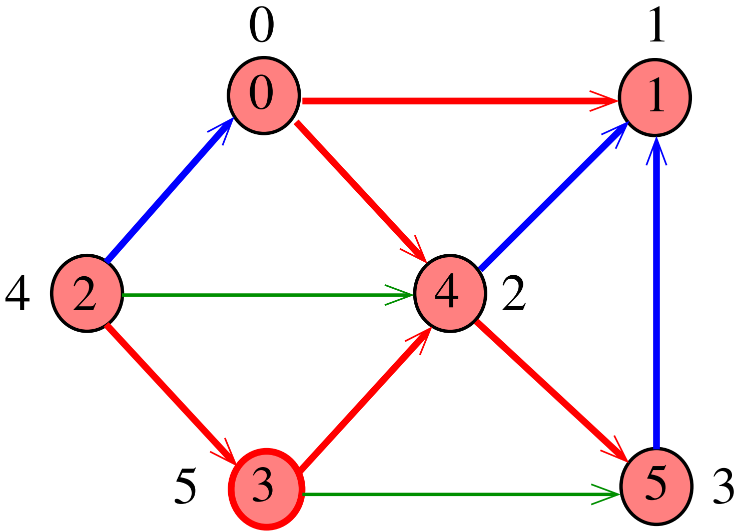
dfsR(G,2)



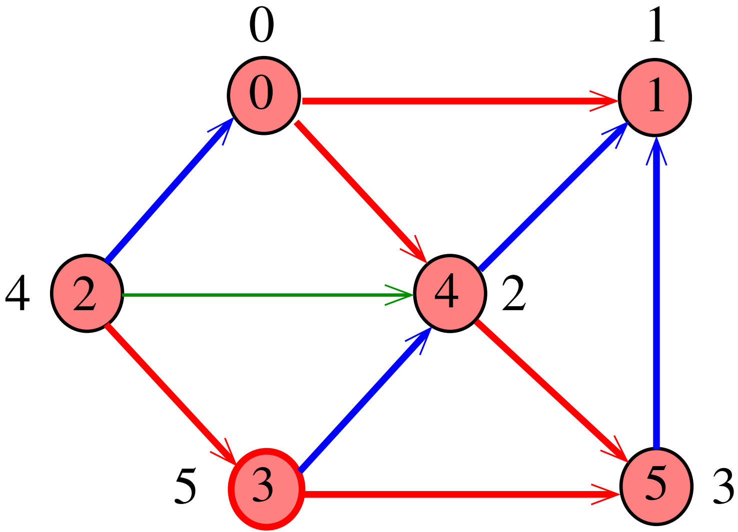
dfsR(G,3)



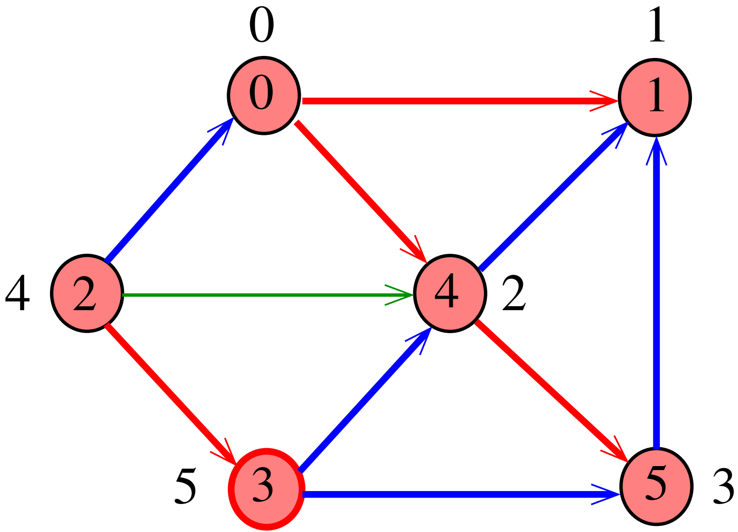
dfsR(G,3)



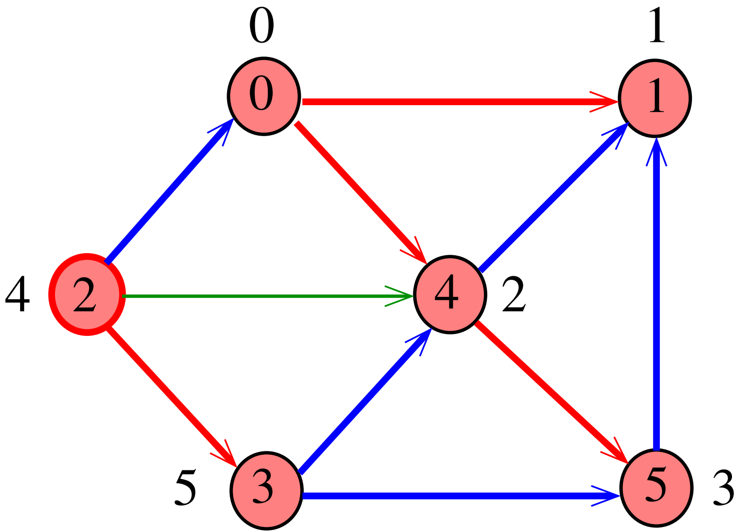
dfsR(G,3)



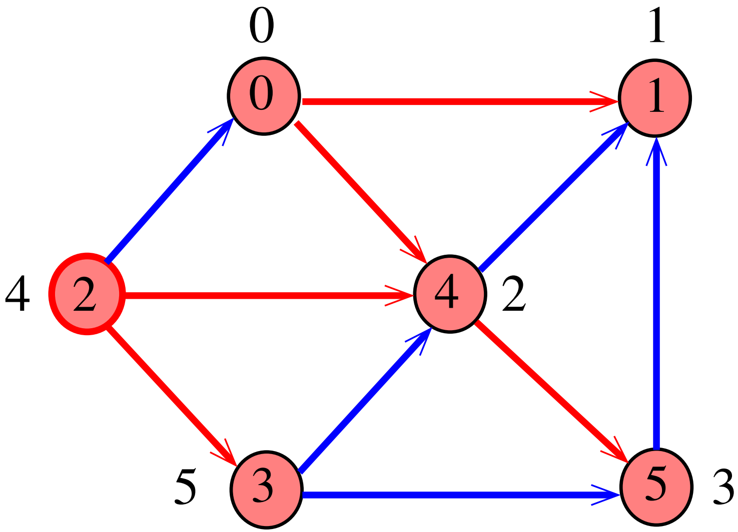
dfsR(G,3)



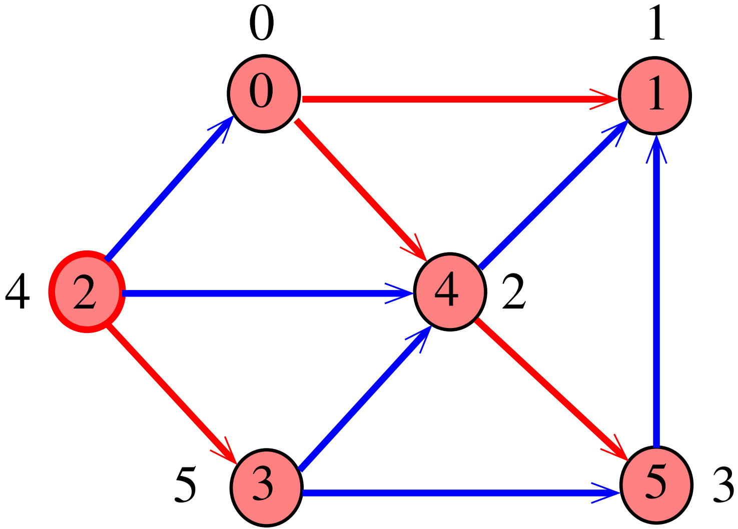
dfsR(G,2)



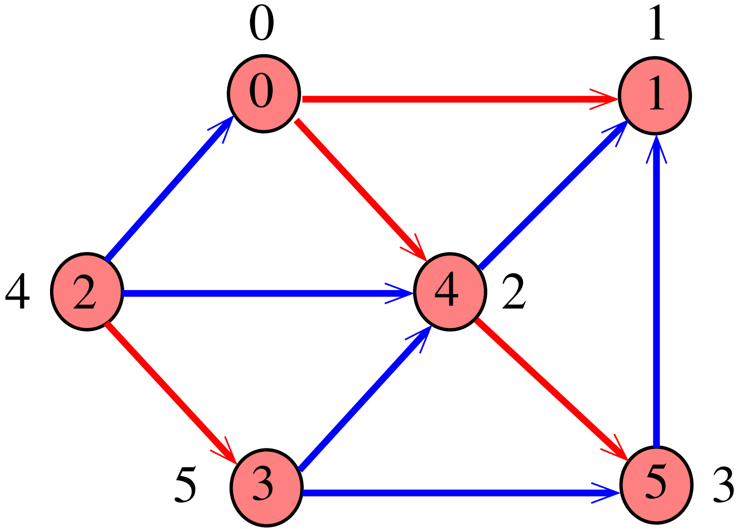
dfsR(G,2)



dfsR(G,2)



DIGRAPHdfs(G)



DIGRAPHdfs

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

dfsR

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

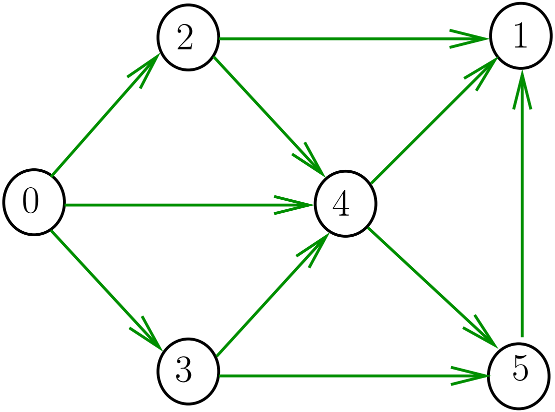
```
void dfsR (Digraph G, Vertex v) {
    Vertex w;
    lbl[v] = cnt++;
    for (w = 0; w < G->V; w++)
        if (G->adj[v][w] != 0)
            if (lbl[w] == -1)
                dfsR(G, w);
}
```

dfsR

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

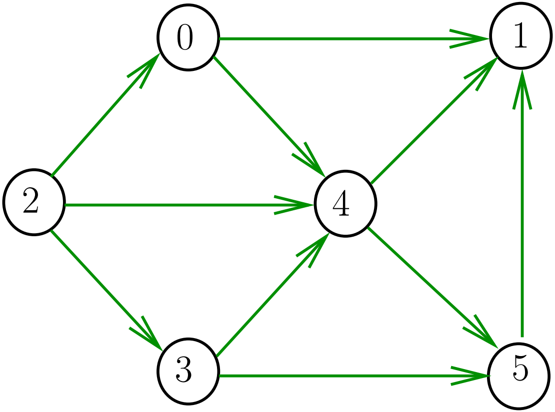
```
void dfsR (Digraph G, Vertex v) {
    link p;
    lbl[v] = cnt++;
    for (p = G->adj[v]; p != NULL; p= p->next)
        if (lbl[p->w] == -1)
            dfsR(G, p->w);
}
```

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 $\Theta(\mathtt{V} + \mathtt{A})$.

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

Arborescência de busca em profundidade

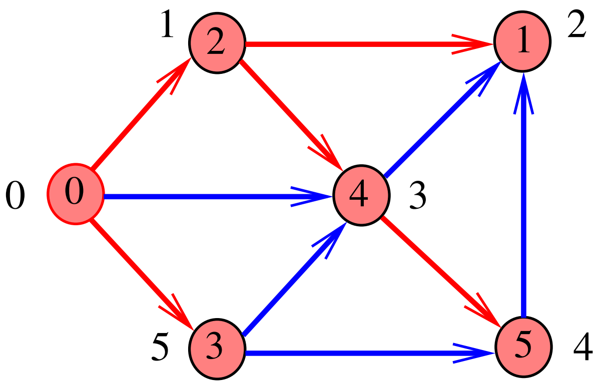
Classificação dos arcos

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

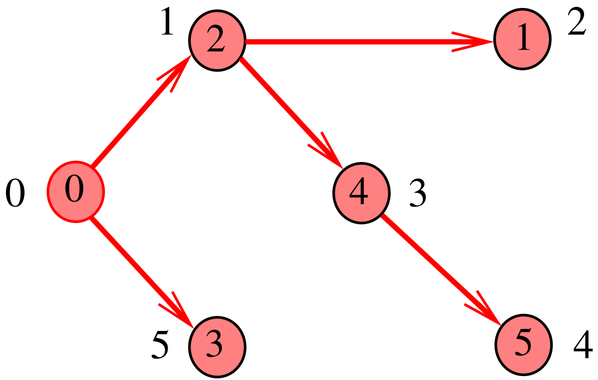
Exemplo: arcos em vermelho são arcos da arborescência



Arcos da arborescência

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

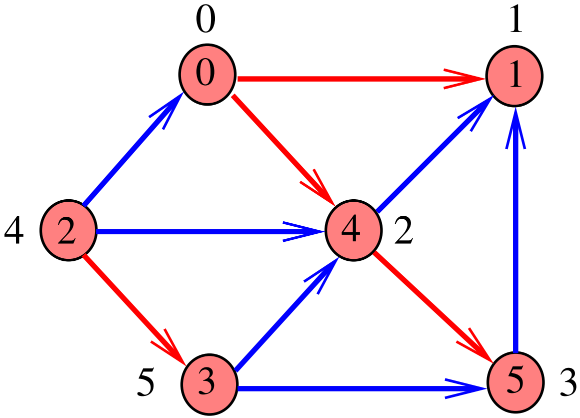
Exemplo: arcos em vermelho são arcos da arborescência



Floresta DFS

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

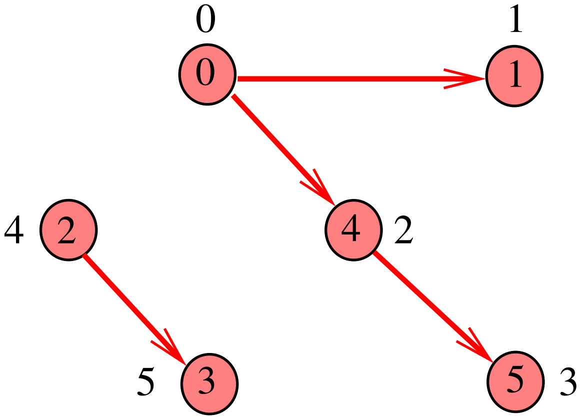
Exemplo: arcos em vermelho formam a floresta DFS



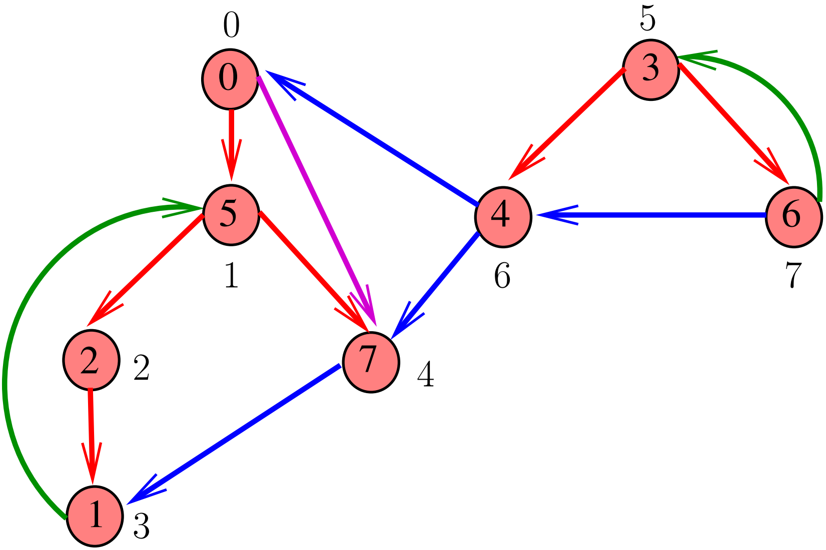
Floresta DFS

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

Exemplo: arcos em vermelho formam a floresta DFS

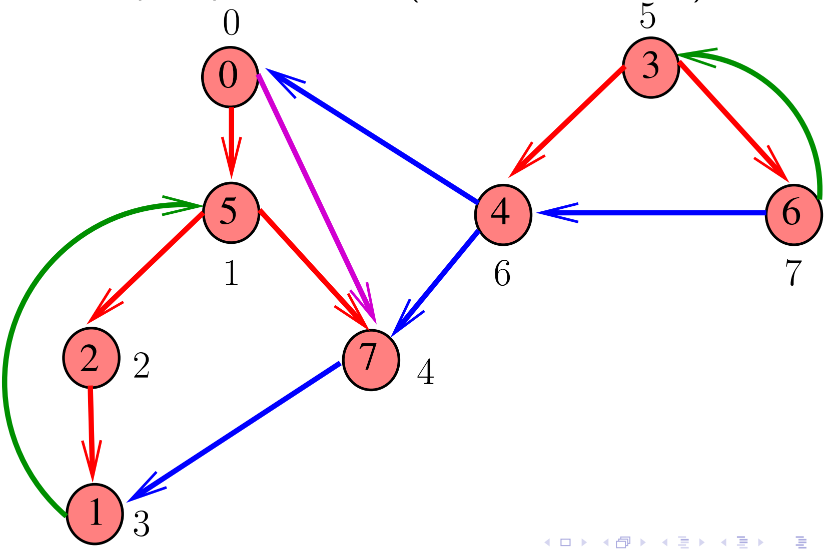


Classificação dos arcos



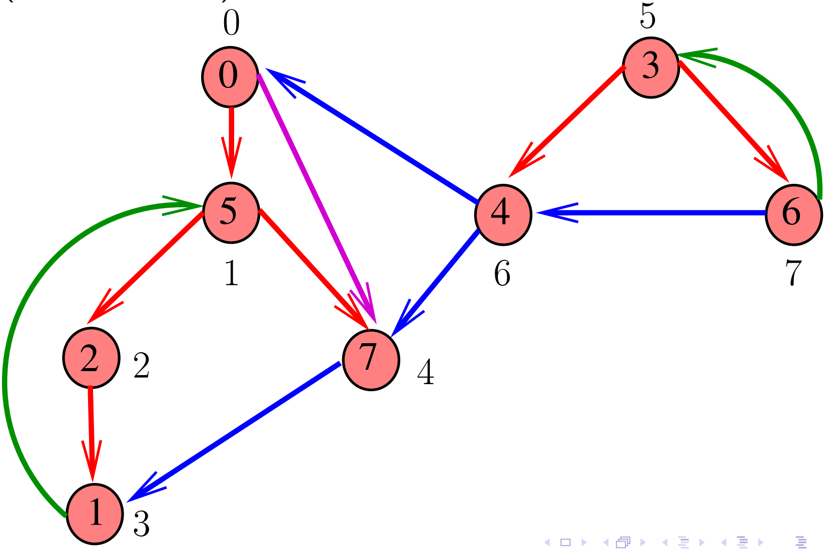
Arcos de arborescência

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



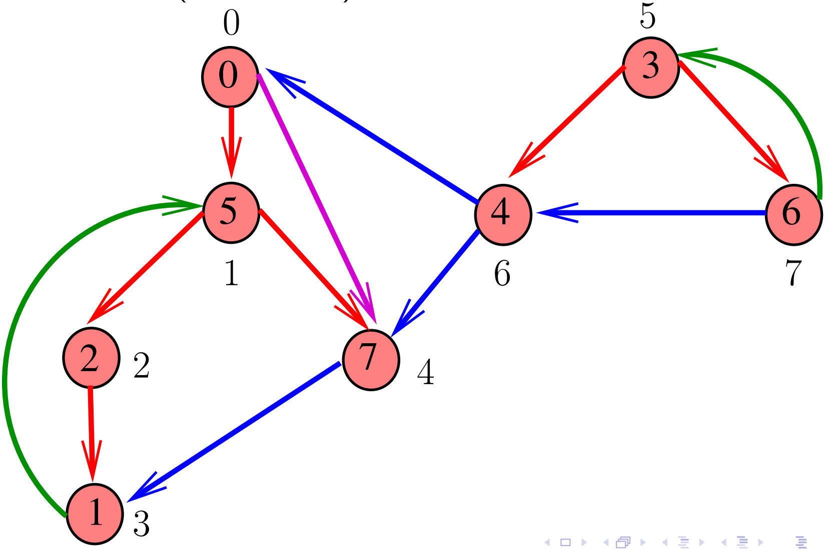
Arcos de retorno

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



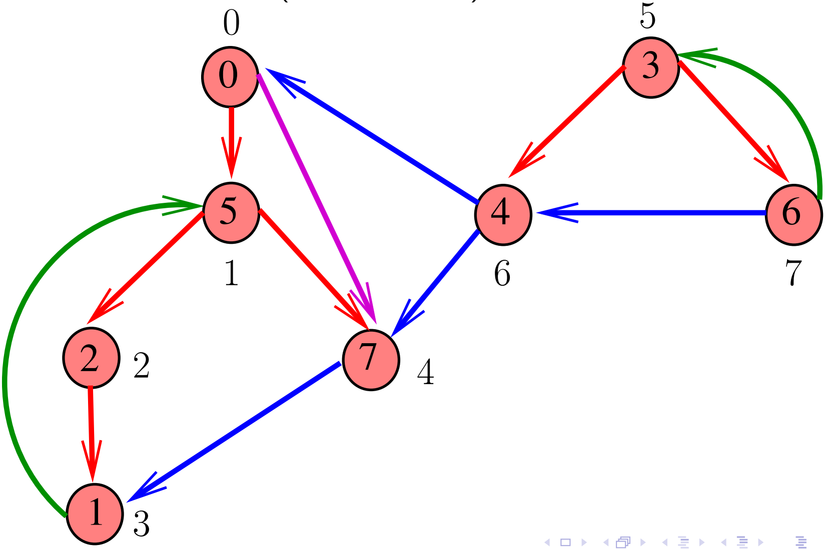
Arcos descendentes

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



Arcos cruzados

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



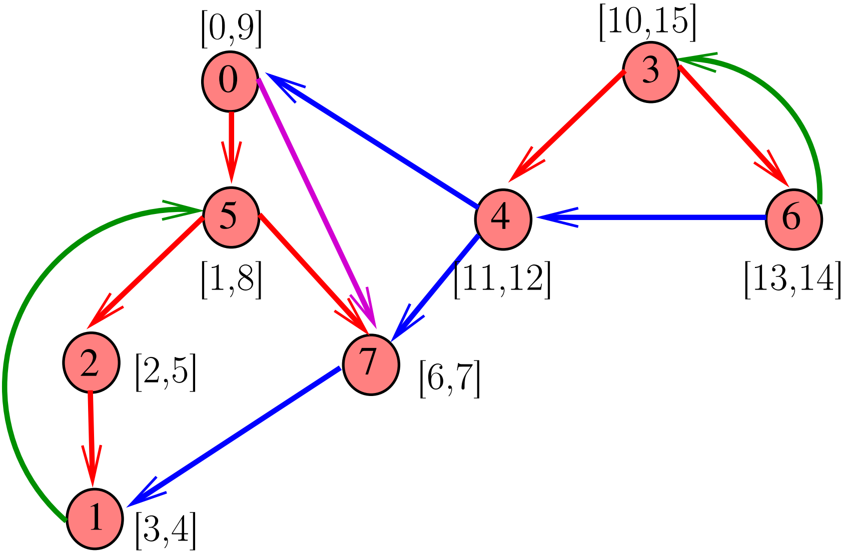
Busca DFS (CLRS)

Vamos supor que nossos digrafos têm no máximo `maxV` vértices

```
#define maxV 10000
static int time, parnt[maxV], d[maxV], f[maxV];
```

`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)



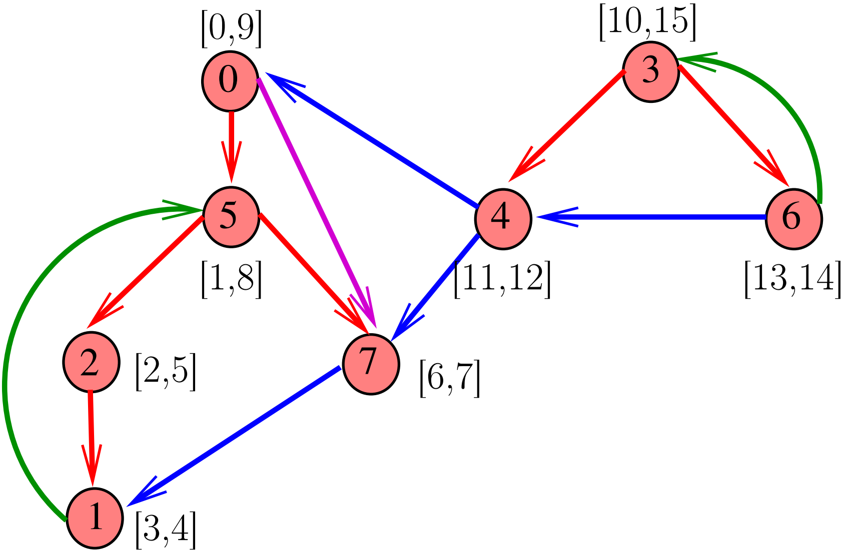
DIGRAPHdfs

```
void DIGRAPHdfs (Digraph G) {
    Vertex v;
    time = 0;
    for (v = 0; v < G->V; v++) {
        d[v] = f[v] = -1;
        parnt[v] = -1;
    }
    for (v= 0; v < G->V; v++)
        if (d[v] == -1)
            dfsR(G, v);
}
```

dfsR

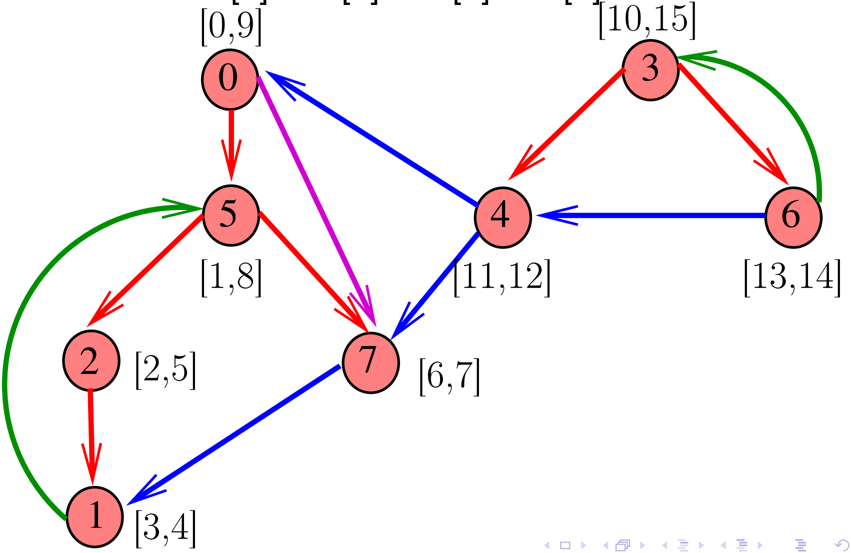
```
void dfsR (Digraph G, Vertex v) {
    link p;
    d[v] = time++;
    for (p = G->adj[v]; p != NULL; p= p->next)
        if (d[p->w] == -1) {
            parnt[w] = p->w;
            dfsR(G, p->w);
        }
    f[v] = time++;
}
```

Classificação dos arcos



Arcos de arborescência ou descendentes

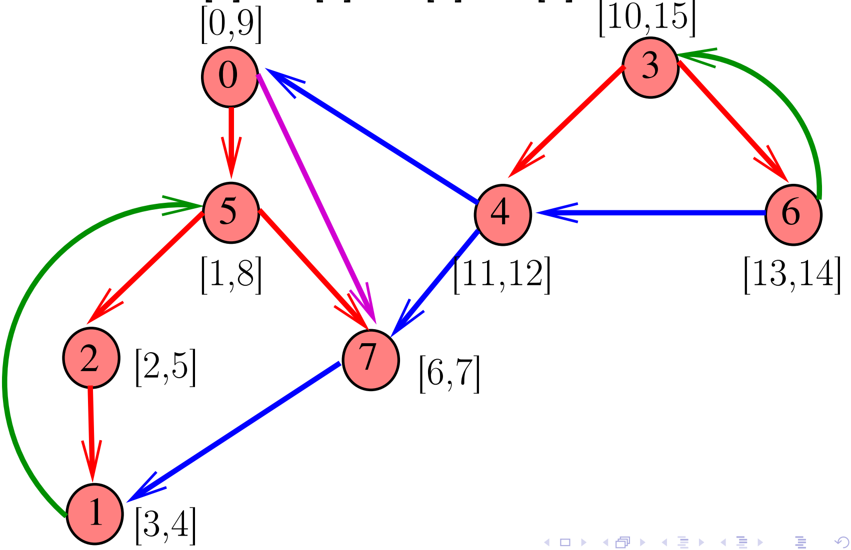
v-w é **arco de arborescência** ou **descendente** se e somente se d[v] $<$ d[w] $<$ f[w] $<$ f[v]



Arcos de retorno

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

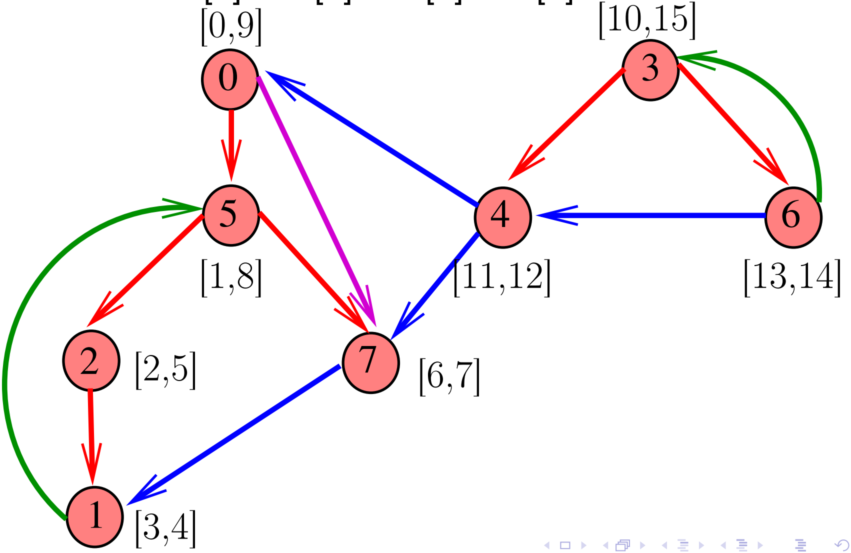
$$d[w] < d[v] < f[v] < f[w]$$



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