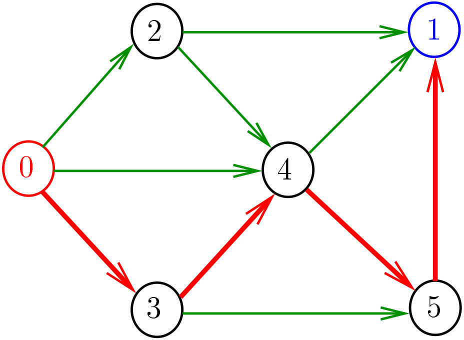
Melhores momentos

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

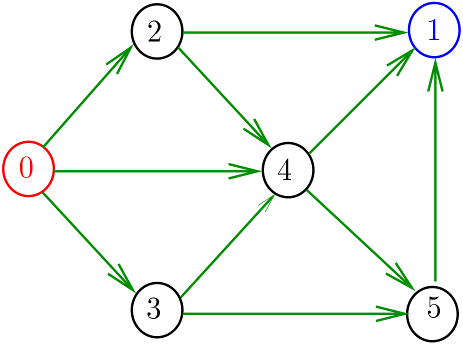
Exemplo: para s = 0 e t = 1 a resposta é SIM



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

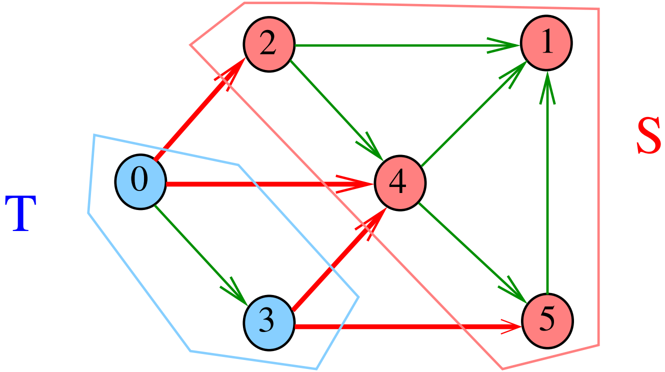
Certificado de inexistência

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

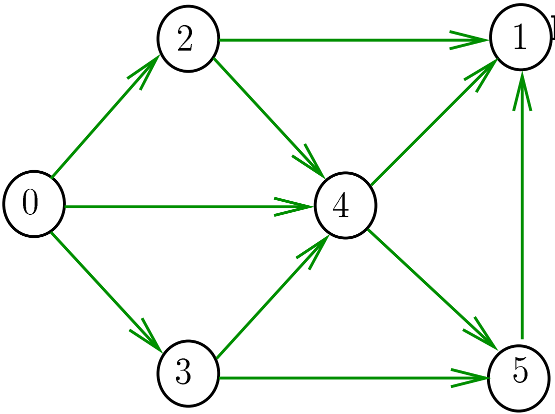
> ***todo arco*** *no corte tem ponta inicial em T e ponta final em S*

Certificado de inexistência

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

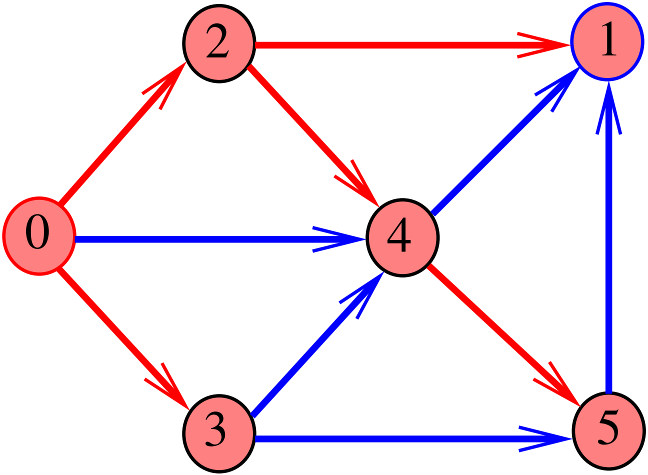


Certificado de existência



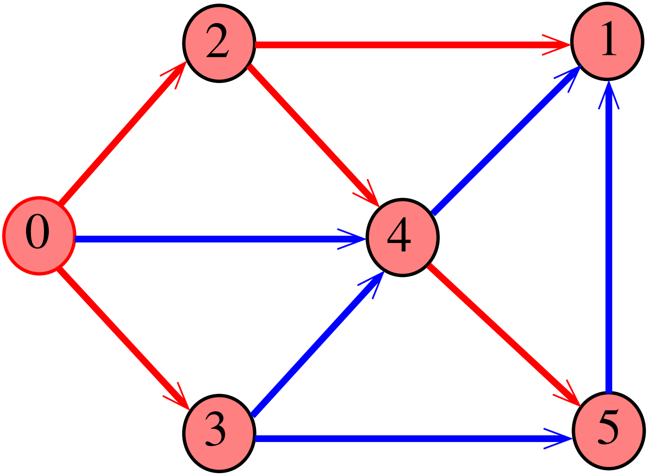

```
pathR(G, 0)
  0-2 pathR(G,2)
    2-1 pathR(G,1)
    2-4 pathR(G,4)
      4-1
      4-5 pathR(G,5)
      5-1
  0-3 pathR(G,3)
    3-4
    3-5
  0-4
existe caminho
```

DIGRAPHpath(G,0,1)



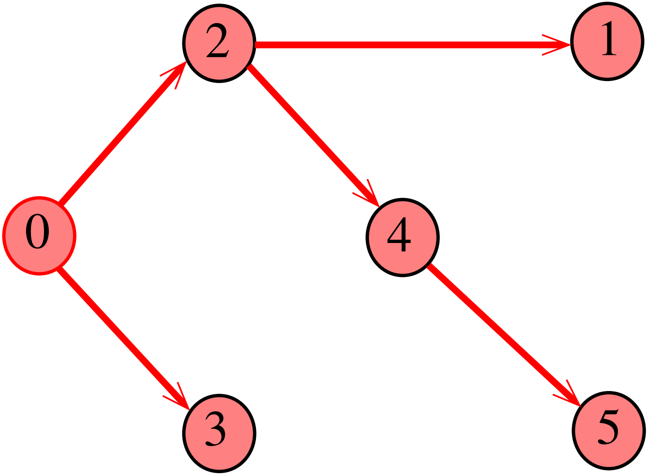
Arborescências

Exemplo: a raiz da arborescência é 0



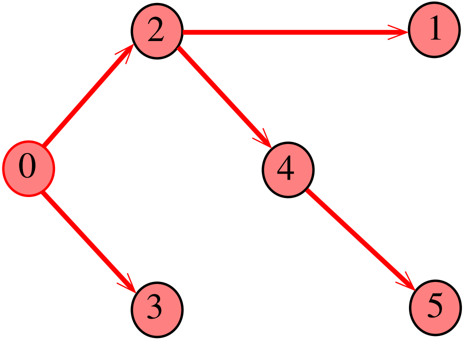
Arborescências

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 s e t de um digrafo, vale uma e apenas umas das seguintes afirmações:

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

AULA 5

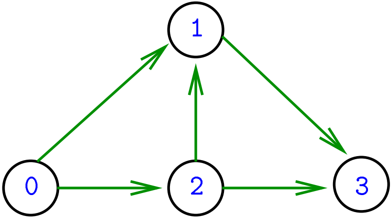
Vetor de listas de adjacência

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:
```

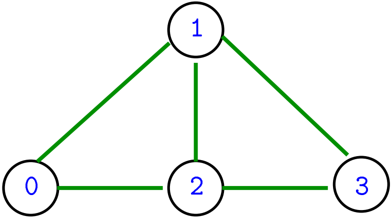
Consumo de espaço: $\Theta(V + A)$ (linear)

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:



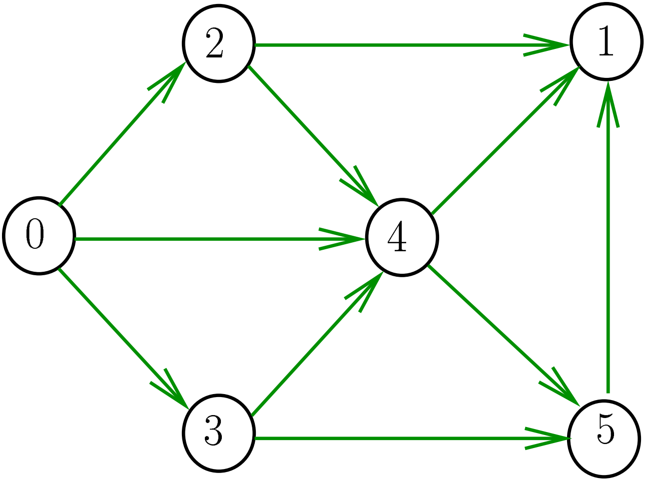

```
0:  1,  2
1:  3,  0,  2
2:  1,  3,  0
3:  1,  2
```

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

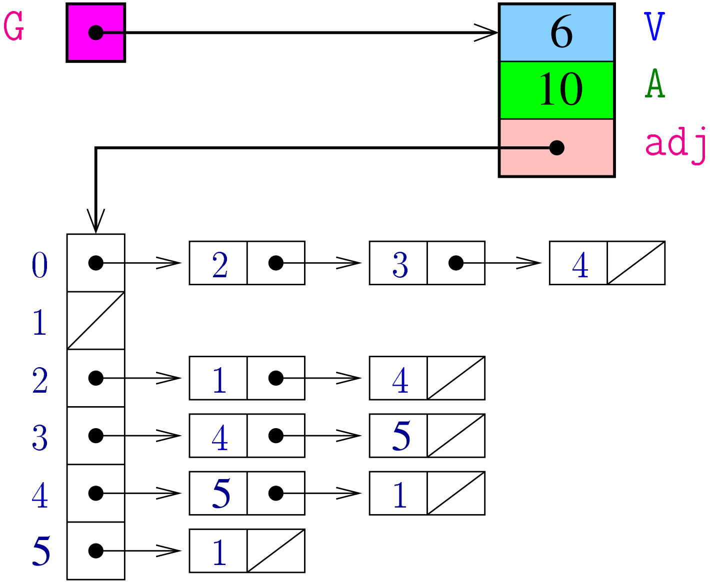
Manipulação eficiente

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

$$\texttt{x.w} = \texttt{w} \text{ e } \texttt{x.next} = \texttt{next}$$

```
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

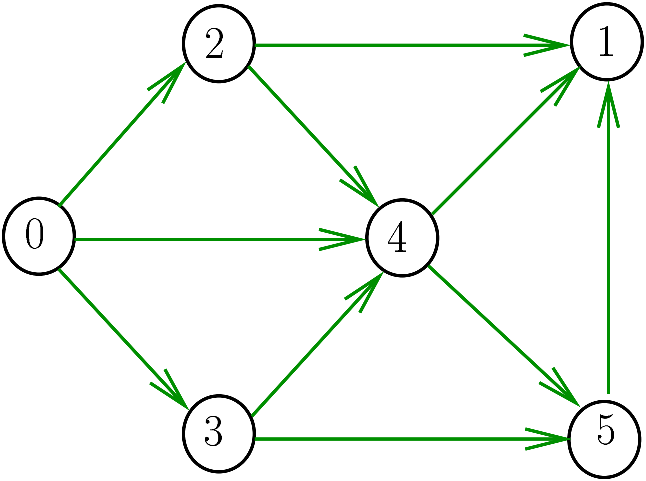
```
void
DIGRAPHinsertA (Digraph G, Vertex v, Vertex w)
{
    link p;
    if (v == w) return;
    for (p = G->adj[v]; p != NULL; p = p->next)
        if (p->w == w) return;
    G->adj[v] = NEW(w, G->adj[v]);
    G->A++;
}
```

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`