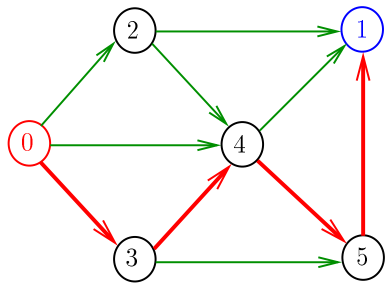
Melhores momentos

AULAS 1-8

Procurando um caminho

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

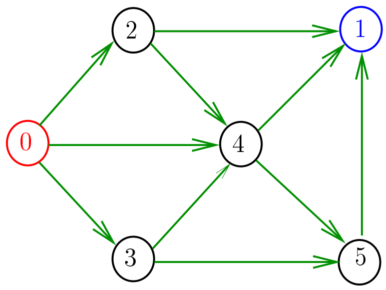
Exemplo: para s = 0 e t = 1 a resposta é SIM



Procurando um caminho

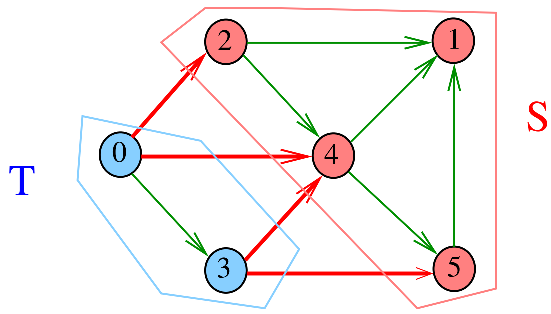
Problema: dados um digrafo G e dois vértices s e t decidir se existe um caminho de s a t

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



Certificado de inexistência

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



Conclusão

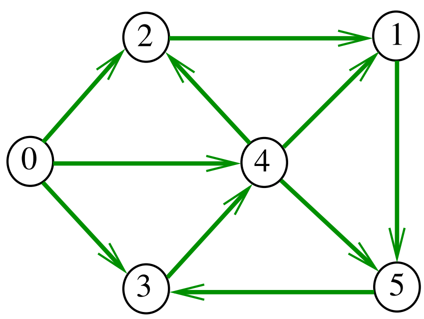
Para quaisquer vértices s e t de um digrafo, vale uma e apenas umas das seguintes afirmações:

- 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.

Procurando um ciclo

Problema: decidir se dado digrafo G possui um ciclo

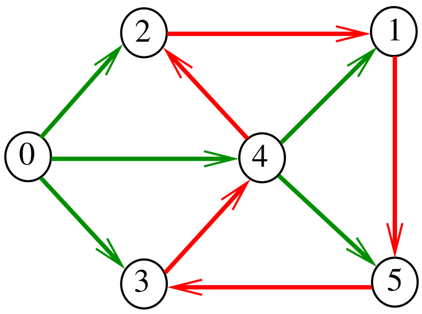
Exemplo: para o grafo a seguir a resposta é SIM



Procurando um ciclo

Problema: decidir se dado digrafo `G` possui um ciclo

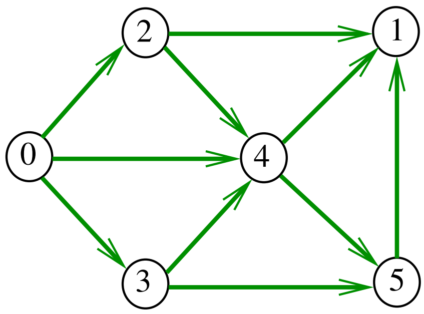
Exemplo: para o grafo a seguir a resposta é SIM



Procurando um ciclo

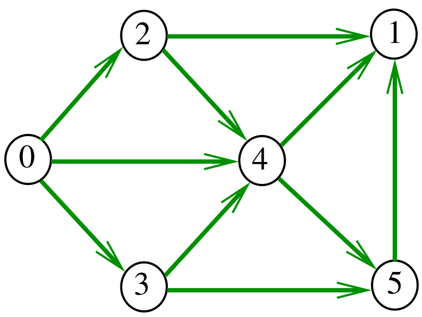
Problema: decidir se dado digrafo `G` possui um ciclo

Exemplo: para o grafo a seguir a resposta é NÃO



Ordenação topológica

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

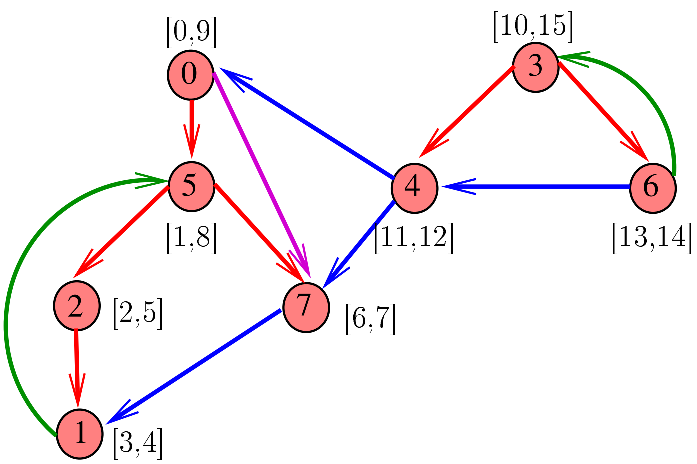


Conclusão

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

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

Floresta DFS



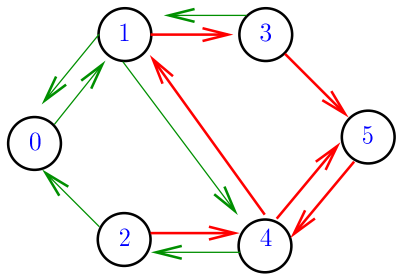
AULA 9

Ciclos em grafos

Comprimento

O **comprimento** de um caminho é o número de arcos no caminho, contanto-se as repetições.

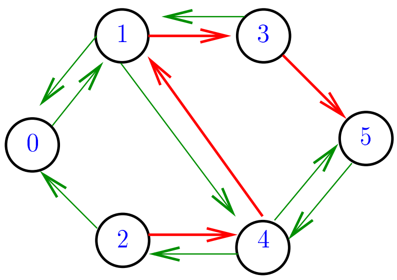
Exemplo: 2-4-1-3-5-4-5 tem comprimento 6



Comprimento

O **comprimento** de um caminho é o número de arcos no caminho, contanto-se as repetições.

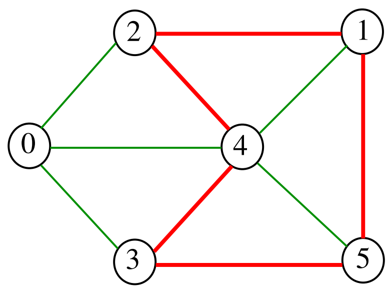
Exemplo: 2-4-1-3-5 tem comprimento 4



Ciclos

Um ciclo é **trivial** se tem comprimento 2
Num grafo, ciclos triviais são ignorados, pois usam os dois arcos de uma mesma aresta.

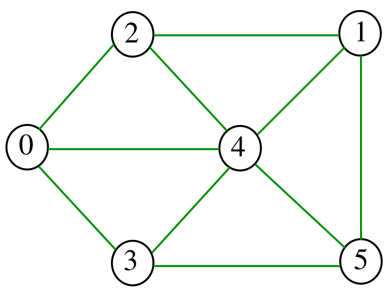
Exemplo: 2-1-5-3-4-2 é um ciclo



Procurando um ciclo

Problema: decidir se dado **grafo** G possui um ciclo (não trivial)

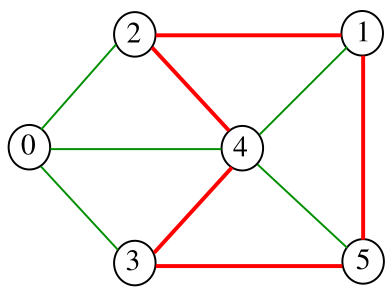
Exemplo: para o grafo a seguir a resposta é SIM



Procurando um ciclo

Problema: decidir se dado **grafo** G possui um ciclo (não trivial)

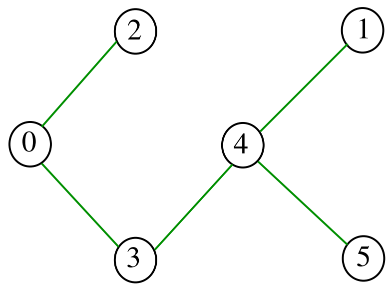
Exemplo: para o grafo a seguir a resposta é SIM



Procurando um ciclo

Problema: decidir se dado grafo G possui um ciclo

Exemplo: para o grafo a seguir a resposta é NÃO



GRAPHcycle

Recebe um grafo G e devolve **1** se existe um ciclo não-trivial em G e devolve **0** em caso contrário
Supõe que o grafo tem no máximo maxV vértices.

```
int GRAPHcycle (Graph G);
```

Primeiro algoritmo

```
int GRAPHcycle (Graph G) {
    Vertex v, w; link p; int output;
    for (v = 0; v < G->V; v++)
        for(p=G->adj[v];p!=NULL;p=p->next){
            w = p->w;
            if (v < w) {
                GRAPHremoveA(G,w,v);
                output = DIGRAPHpath(G,w,v);
                GRAPHinsertA(G,w,v);
                if (output == 1) return 1;
            }
        }
    return 0;
```

Consumo de tempo

O consumo de tempo da função GRAPHcycle é $A/2$ vezes o consumo de tempo da função DIGRAPHpath.

O consumo de tempo da função GRAPHcycle para vetor de listas de adjacência é $O(A(V+A))$.

O consumo de tempo da função GRAPHcycle para matriz de adjacência é $O(AV^2)$.

GRAPHcycle

Vamos supor que nossos digrafos têm no máximo maxV vértices

```
#define maxV 10000
static int cnt, parnt[maxV];
```

GRAPHcycle

Recebe um grafo G e devolve **1** se existe um ciclo não-trivial em G e devolve **0** em caso contrário

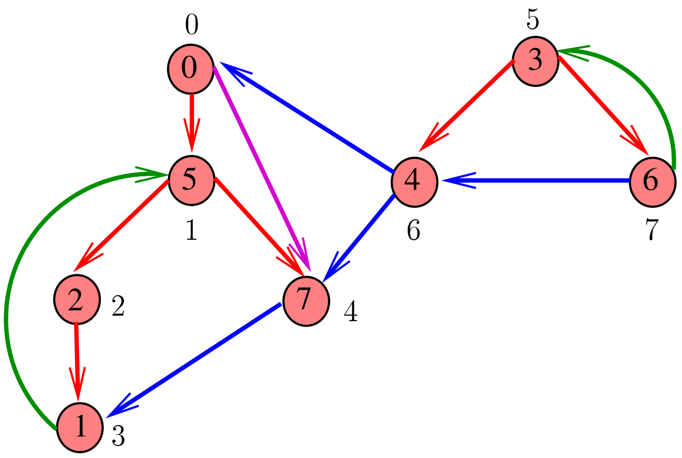
```
int GRAPHcycle (Graph G);
```

A função tem por base a seguinte observação: em relação a **qualquer** floresta DFS,

*todo arco de **retorno** que **não** é anti-paralelo a um arco da arborescência pertence a um ciclo não-trivial*

*todo ciclo não trivial tem um arco de **retorno** que **não** é anti-paralelo a um arco da arborescência*

Arcos de retorno



GRAPHcycle

```
int GRAPHcycle (Graph G) {
    Vertex v;
    for (v = 0; v < G->V; v++)
        lbl[v] = -1;
    for (v= 0; v < G->V,v++)
        if (lbl[v] == -1) {
            parnt[v] = v;
            if (cycle3R(G,v) == 1)
                return 1 ;
    return 0;
}
```

cycle3R

```
int cycle3R (Graph G, Vertex v) {
    link p;
    lbl[v] = 1;
    for (p = G->adj[v]; p != NULL; p= p->next)
        Vertex w = p->w;
        if (lbl[w] == -1) {
            parnt[w] = v;
            if (cycle3R(G,w)==1) return 1;
        }
        else if (parnt[w]!=v) return 1;
    }
    return 0;
}
```

Consumo de tempo

O consumo de tempo da função GRAPHcycle para vetor de listas de adjacência é $O(V + A)$.

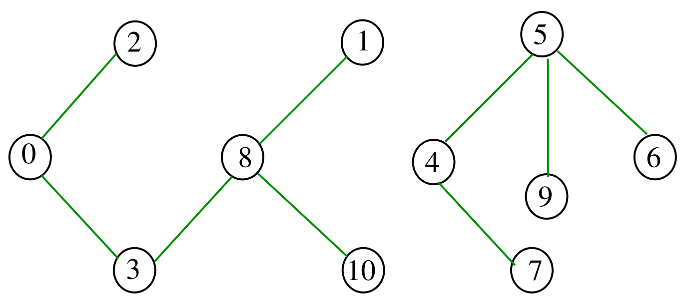
O consumo de tempo da função GRAPHcycle para matriz de adjacência é $O(V^2)$.

Florestas e árvores

Florestas

Uma **floresta** (= *forest*) é um grafo sem ciclos não-triviais

Exemplo:



Propriedades

Para cada par s,t de vértices de uma árvore existe um e um só caminho simples de s a t.

Toda árvore com V vértices tem exatamente V-1 arestas.

Conclusão

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

- G possui um ciclo não trivial
- G é uma floresta

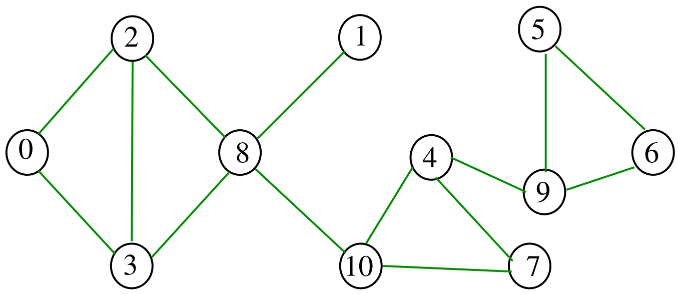
Componentes de grafos

S 18.5

Grafos conexos

Um grafo é **conexo** se e somente se, para cada par (s,t) de seus vértices, existe um caminho com origem s e término t

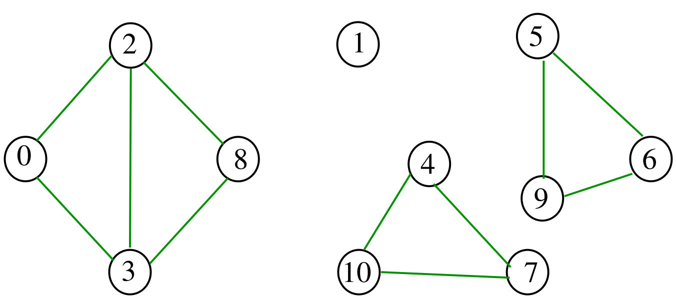
Exemplo: um grafo conexo



Componentes de grafos

Uma **componente** (= *component*) de um grafo é o subgrafo conexo maximal

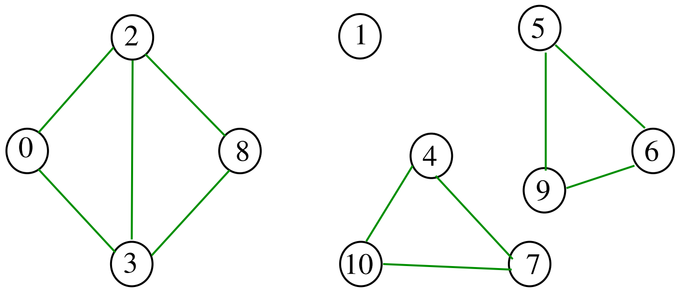
Exemplo: grafo com 4 componentes (conexos)



Contando componentes

Problema: calcular o número de componente

Exemplo: grafo com 4 componentes



Cálculo das componentes de grafos

A função abaixo devolve o número de componentes do grafo G.

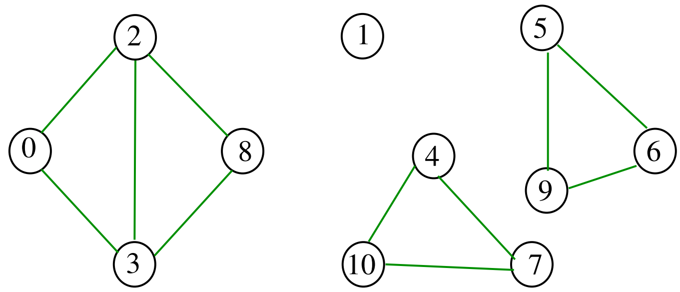
```
#define maxV 10000
static int cc[maxV];
```

Além disso, ela armazena no vetor cc o número do componente a que o vértice pertence: se o vértice v pertence ao k-ésimo componente então cc[v] == k-1

```
int GRAPHcc (Graph G)
```

Exemplo

v	0	1	2	3	4	5	6	7	8	9	10
cc[v]	0	1	0	0	2	3	3	2	0	3	2



GRAPHcc

```
int GRAPHcc (Graph G) {
 Vertex v; int id = 0;
 for (v = 0; v < G->V; v++) cc[v] = -1;
 for (v = 0; v < G->V; v++)
     if (cc[v] == -1)
         dfsRcc(G, v, id++);
 return id;
}
```

dfsRcc

```
void dfsRcc (Graph G, Vertex v, int id){
 link p;
 cc[v] = id;
 for (p=G->adj[v];p!=NULL;p=p->next)
     if (cc[p->w] == -1)
         dfsRcc(G, p->w, id);
}
```

Consumo de tempo

O consumo de tempo da função GRAPHcc é $O(V + A)$.