

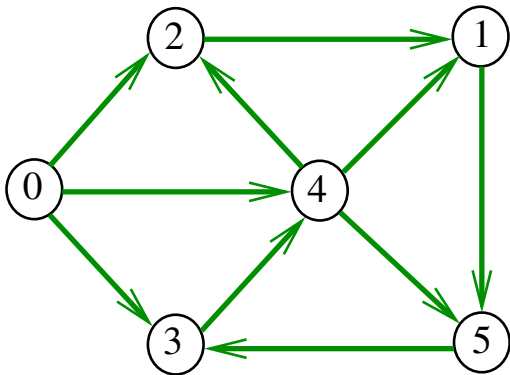
Melhores momentos

AULA 7

Procurando um ciclo

Problema: decidir se dado digrafo G possui um ciclo

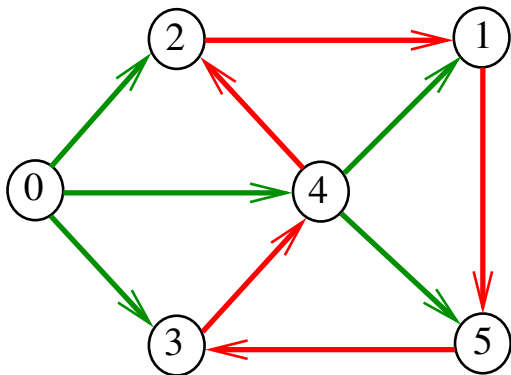
Exemplo: para o grafo a seguir a resposta é SIM



Procurando um ciclo

Problema: decidir se dado digrafo G possui um ciclo

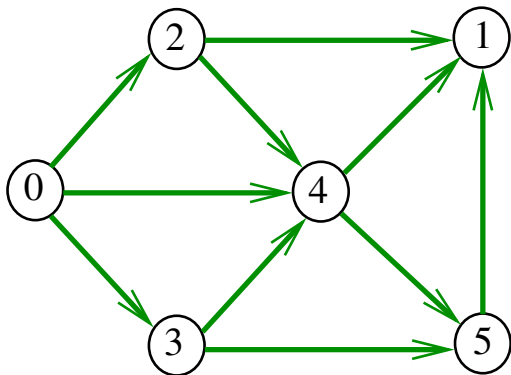
Exemplo: para o grafo a seguir a resposta é SIM



Procurando um ciclo

Problema: decidir se dado digrafo G possui um ciclo

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



digraphcycle

Recebe um digrafo **G** e devolve **1** se existe um ciclo em **G** e devolve **0** em caso contrário

```
int digraphcycle (Digraph G);
```

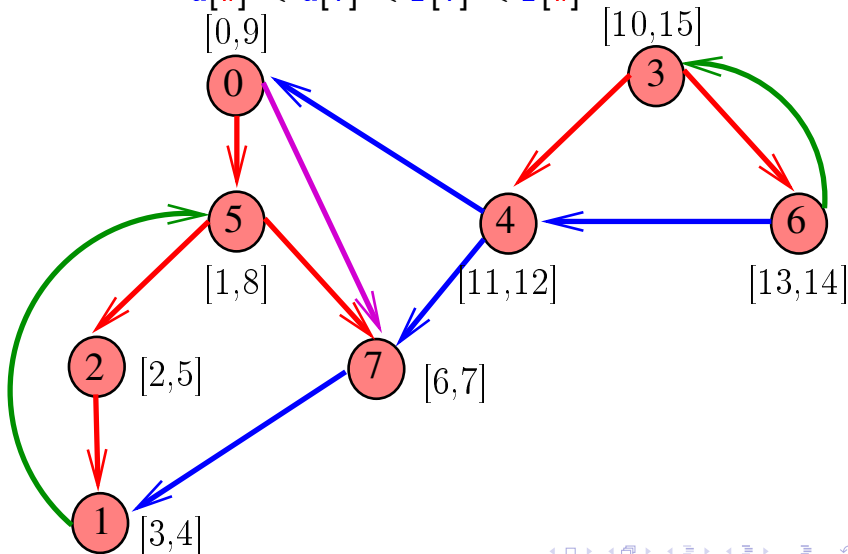
A função tem por base a seguinte observação: em relação a **qualquer** floresta de busca em profundidade,

todo arco de **retorno** pertence a um ciclo e
todo ciclo tem um arco de **retorno**

Arcos de retorno

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

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



Certificados

Como é possível 'verificar' a resposta?

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

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

Certificado de existência

Trecho de código que verifica se o arco **v-w** junto com alguns arcos da floresta DFS formam um ciclo
Supõe que o grafo está representado através de **matriz de adjacência**

```
[...]  
if (G->adj[v][w] == 0)  
    return ERRO;  
if (st_caminho(G, w, v) == 0)  
    return ERRO;  
[...]
```


st_caminho

int

```
st_caminho (Digraph G, Vertex s, Vertex t) {  
    Vertex v, w;  
1    if (parnt[t] == -1 || parnt[s] == -1)  
2        return ERRO;  
3    for (w = t; w != s; w = v) {  
4        v = parnt[w];  
5        if (G->adj[v][w] != 1) return ERRO;  
    }  
6    return OK;  
}
```

AULA 8

Digrafos acíclicos (DAGs)

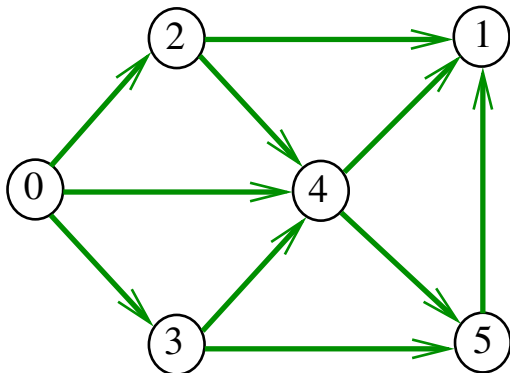
S 19.5 e 19.6

DAGs

Um digrafo é **acíclico** se não tem ciclos

Digrafos acíclicos também são conhecidos como DAGs (= *directed acyclic graphs*)

Exemplo: um digrafo acíclico

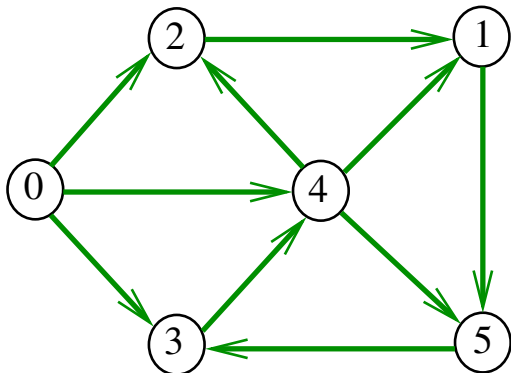


DAGs

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

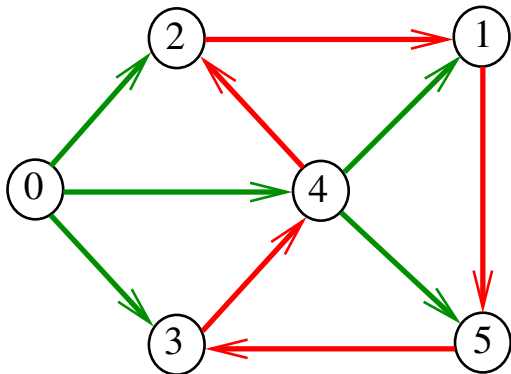


DAGs

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

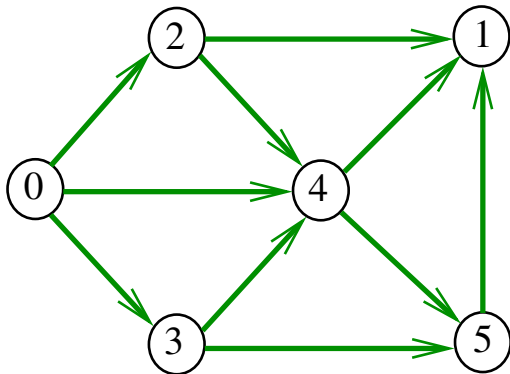
$$ts[i] - ts[j] \text{ com } i < j$$

$ts[0]$ 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

S 19.6

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

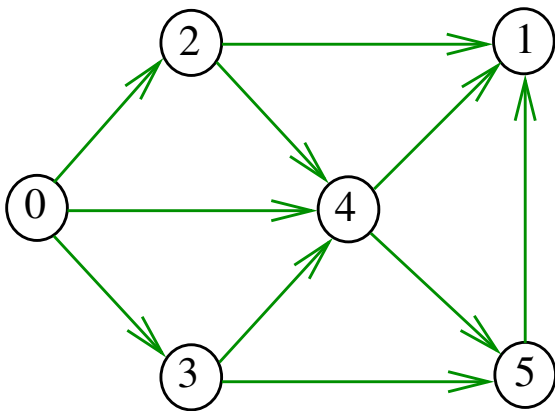
Se $i = G \rightarrow V$ então $ts[0 \dots i-1]$ é uma ordenação topológica de G .

Caso contrário, G **não** é um DAG

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

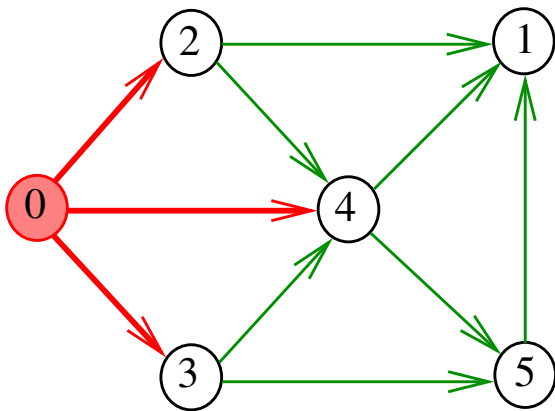
Exemplo

i	0	1	2	3	4	5
ts[i]						



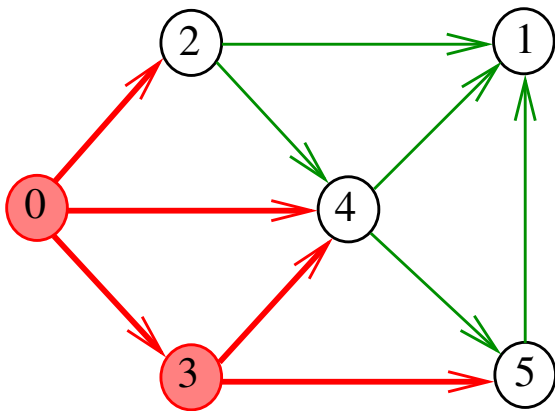
Exemplo

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



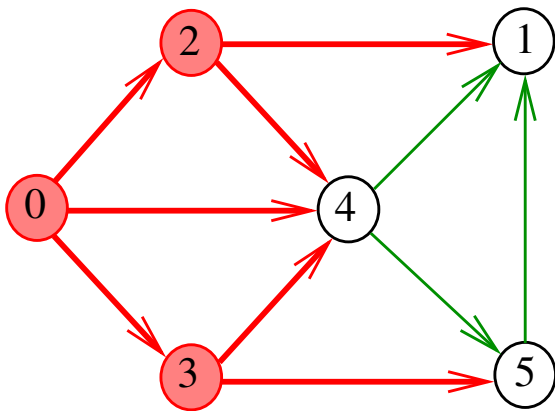
Exemplo

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



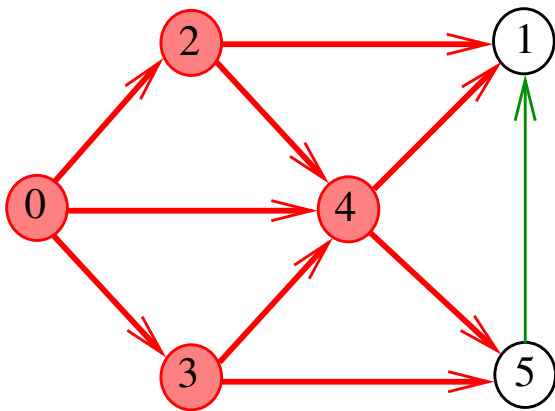
Exemplo

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



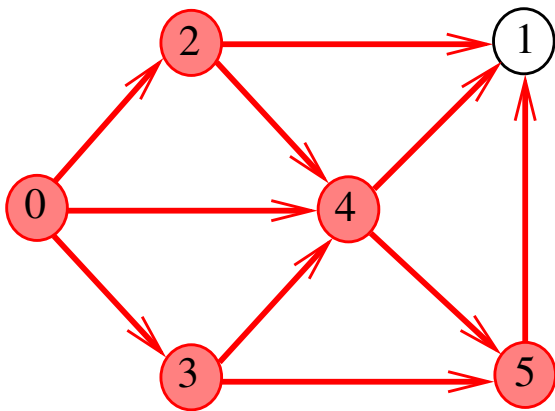
Exemplo

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



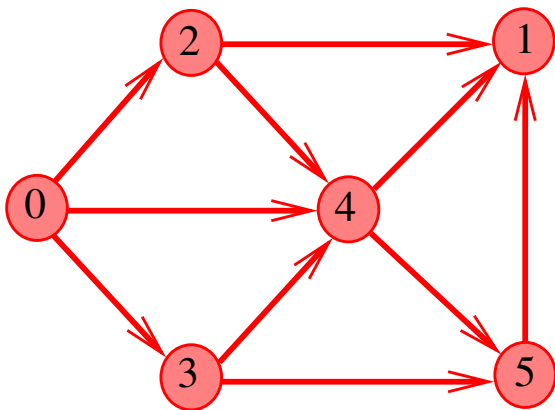
Exemplo

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



DAGts1

```
int DAGts1 (Digraph G, Vertex ts[])
{
1   int i, in[maxV]; Vertex v; link p;
2   for (v = 0; v < G->V; v++)
3       in[v] = 0;
4   for (v = 0; v < G->V; v++)
5       for (p=G->adj[v]; p!=NULL; p=p->next)
6           in[p->w] ++;
```

DAGts1

```
7  QUEUEinit(G->V);
8  for(v = 0; v < G->V; v++)
9      if (in[v] == 0)
10         QUEUEput(v);
11  for (i = 0; !QUEUEempty(); i++) {
12      ts[i] = v = QUEUEget();
13      for (p=G->adj[v]; p!=NULL; p=p->next)
14          if (--in[p->w] == 0)
15              QUEUEput(p->w);
16  }
17  QUEUEfree();
18  return i;
19 }
```

Implementação de uma fila

```
/* Item.h */  
typedef Vertex Item;  
  
/* QUEUE.h */  
void QUEUEinit(int);  
int QUEUEempty();  
void QUEUEput(Item);  
Item QUEUEget();  
void QUEUEfree();
```

QUEUEinit e QUEUEempty

```
Item *q;  
int inicio, fim;  
  
void QUEUEinit(int maxN) {  
    q = (Item*) malloc(maxN*sizeof(Item));  
    inicio = 0;  
    fim = 0;  
}  
int QUEUEempty() {  
    return inicio == fim;  
}
```

QUEUEput, QUEUEget e QUEUEfree

```
void QUEUEput(Item item){  
    q[fim++] = item;  
}
```

```
Item QUEUEget() {  
    return q[inicio++];  
}
```

```
void QUEUEfree() {  
    free(q);  
}
```

Consumo de tempo

O consumo de tempo da função `DAGts1` para
vetor de listas de adjacência é $O(V + A)$.

Algoritmos de ordenação topológica

S 19.6

Algoritmo DFS

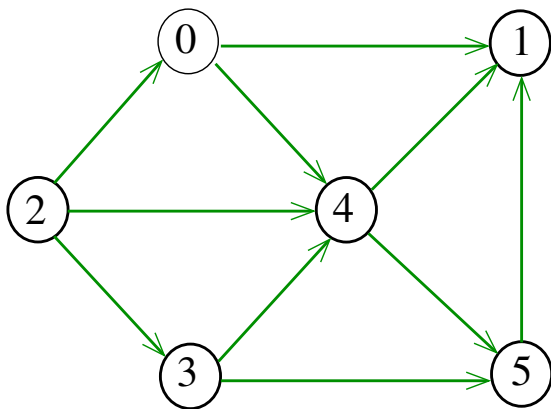
```
static int cnt;  
static int lbl[maxV];
```

Recebe um DAG G e armazena em $ts[0 \dots V-1]$
uma ordenação topológica de G

```
void DAGts2 (Digraph  $G$ , Vertex ts[]);
```

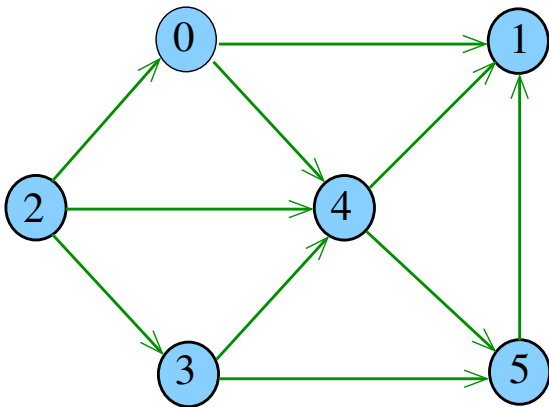
Exemplo

i	0	1	2	3	4	5
ts[i]						



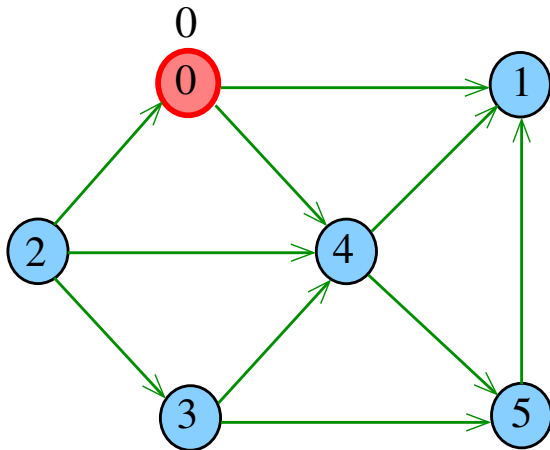
Exemplo

i	0	1	2	3	4	5
ts[i]						

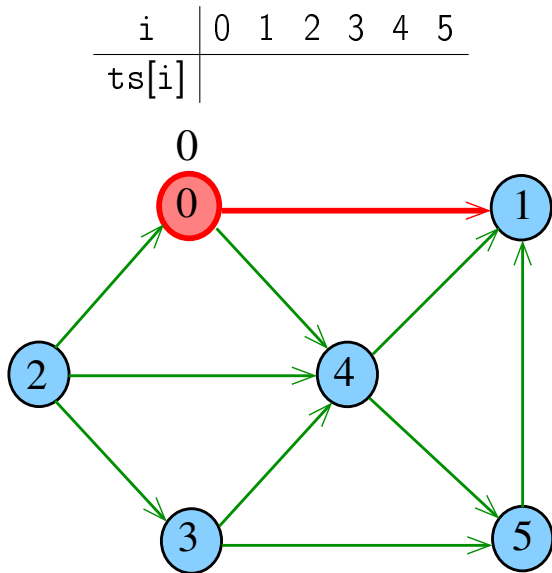


Exemplo

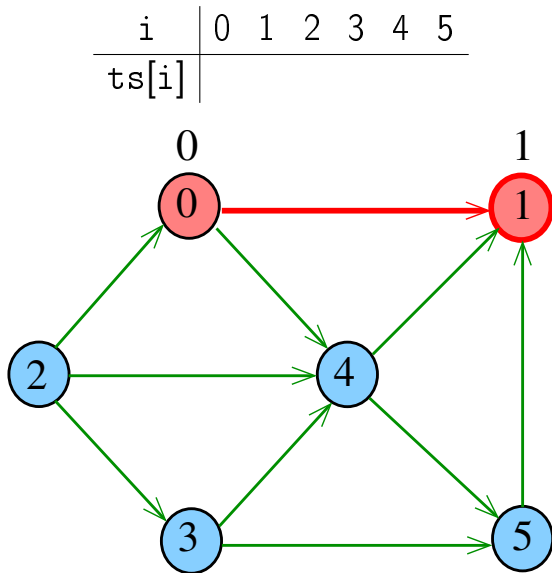
i	0	1	2	3	4	5
ts[i]						



Exemplo

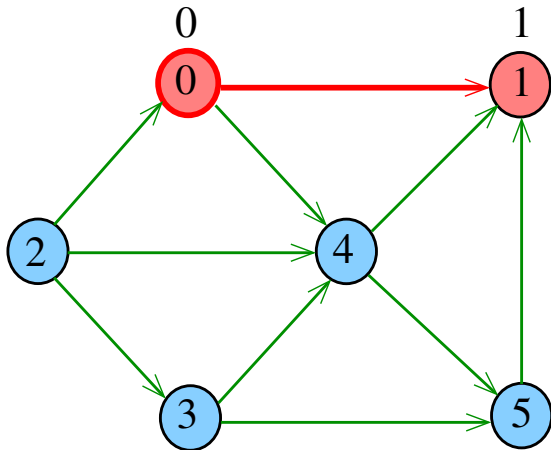


Exemplo



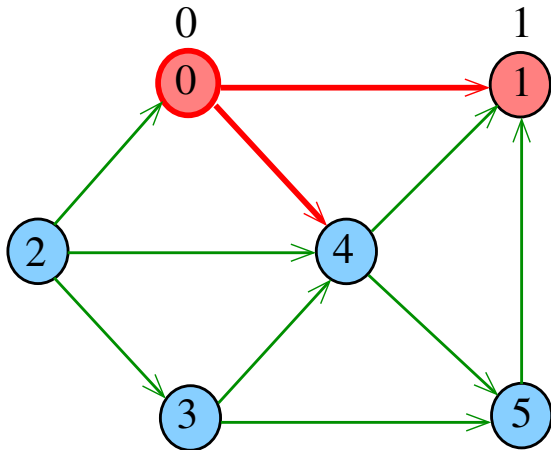
Exemplo

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



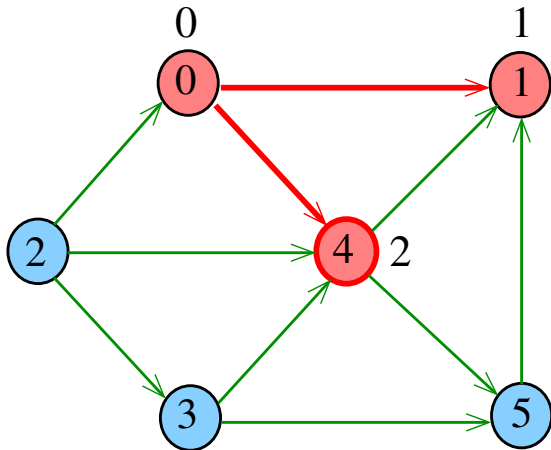
Exemplo

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



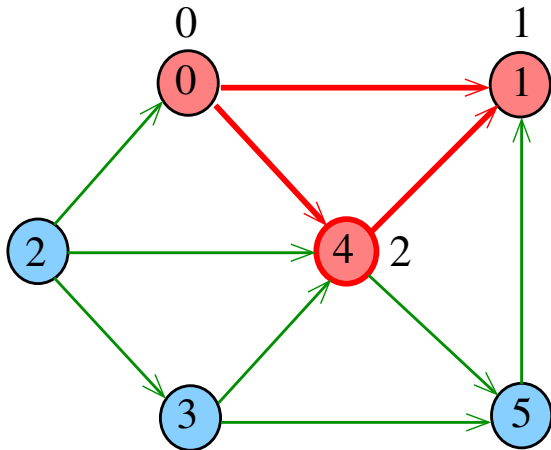
Exemplo

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



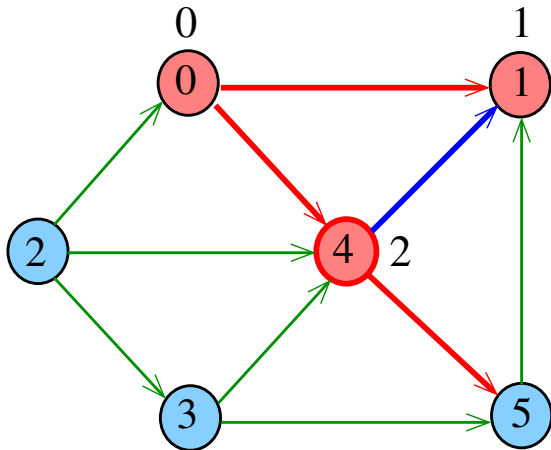
Exemplo

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



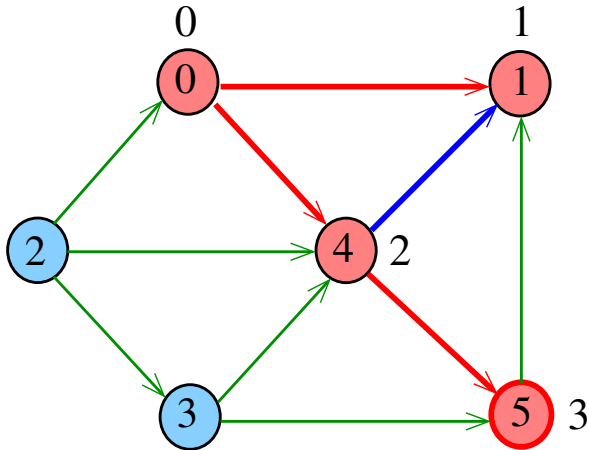
Exemplo

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



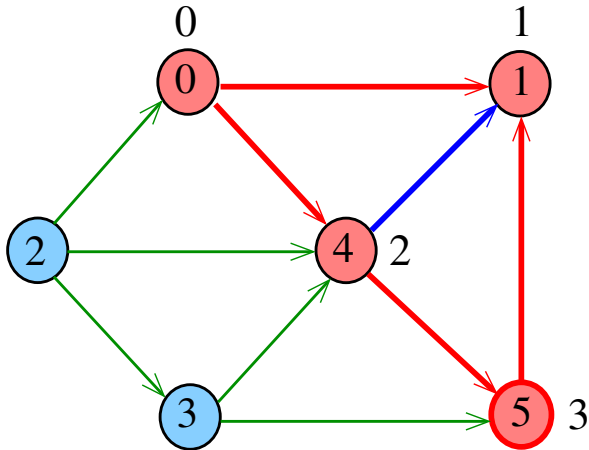
Exemplo

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



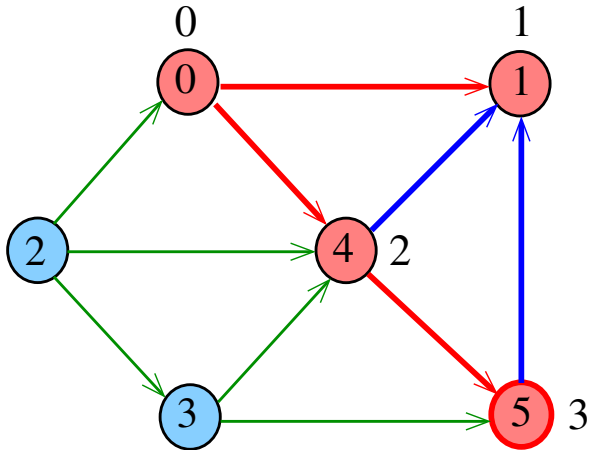
Exemplo

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



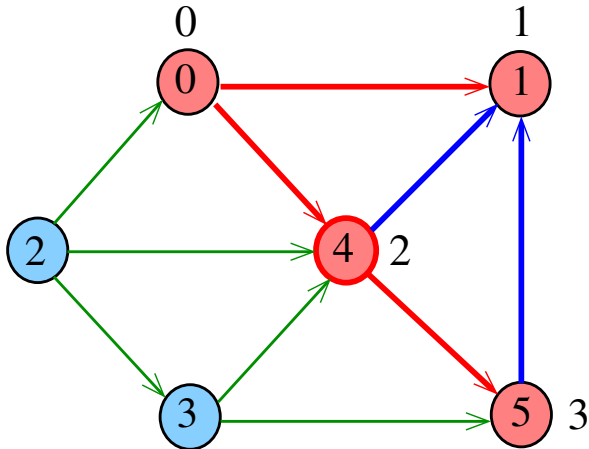
Exemplo

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



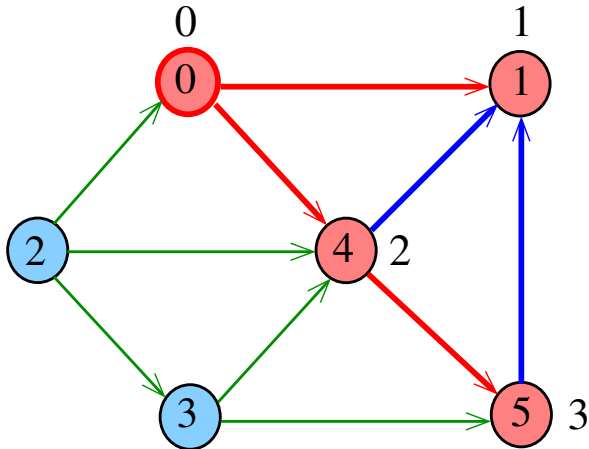
Exemplo

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



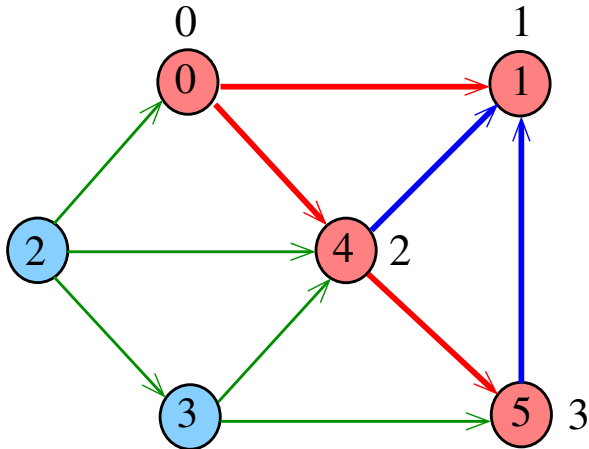
Exemplo

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



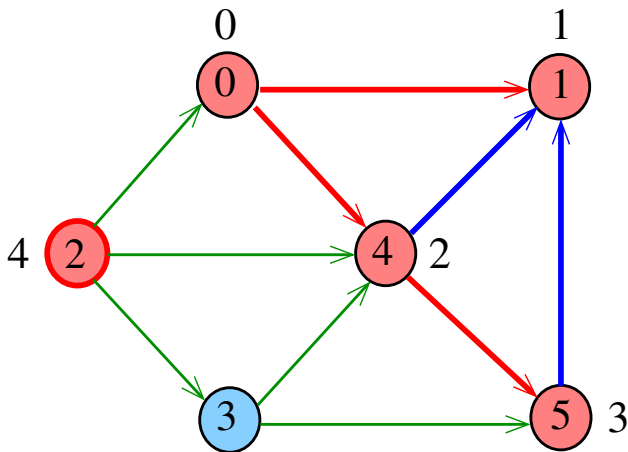
Exemplo

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



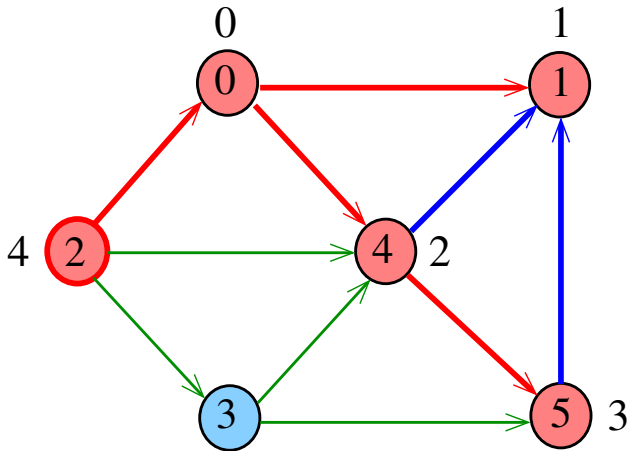
Exemplo

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



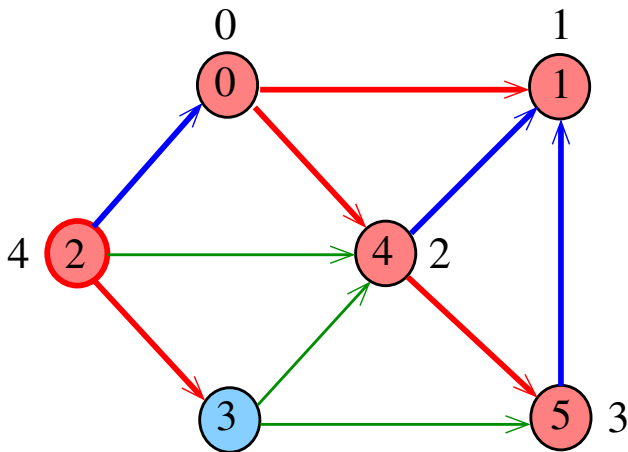
Exemplo

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



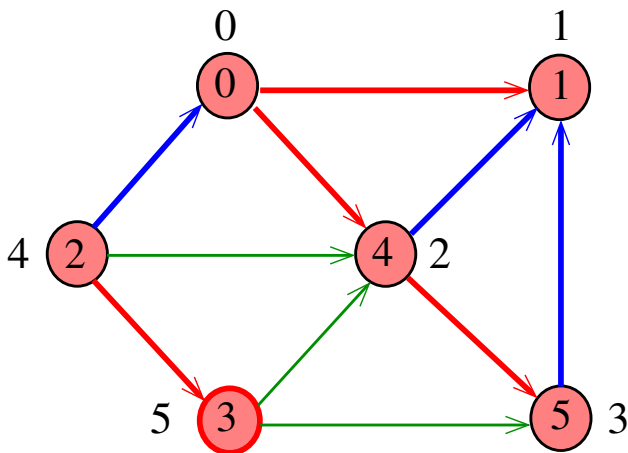
Exemplo

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



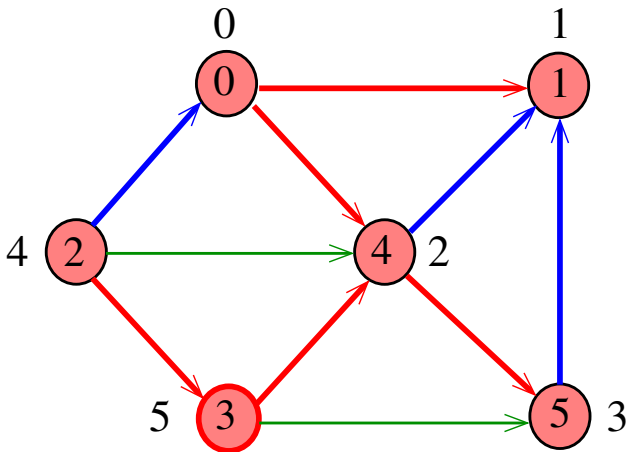
Exemplo

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



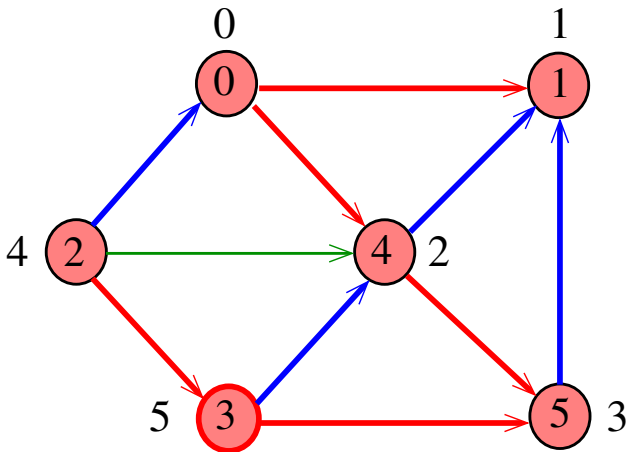
Exemplo

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



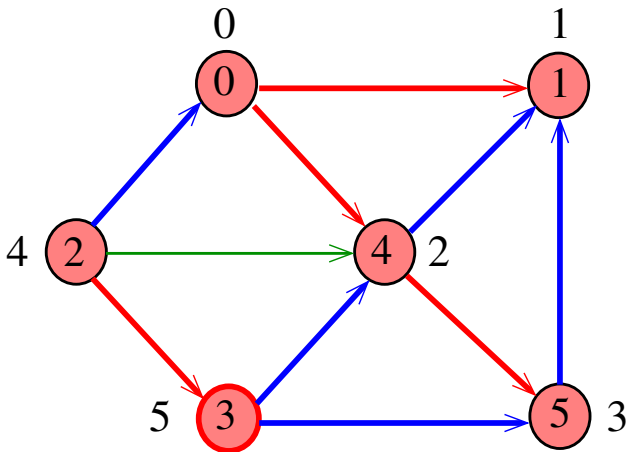
Exemplo

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



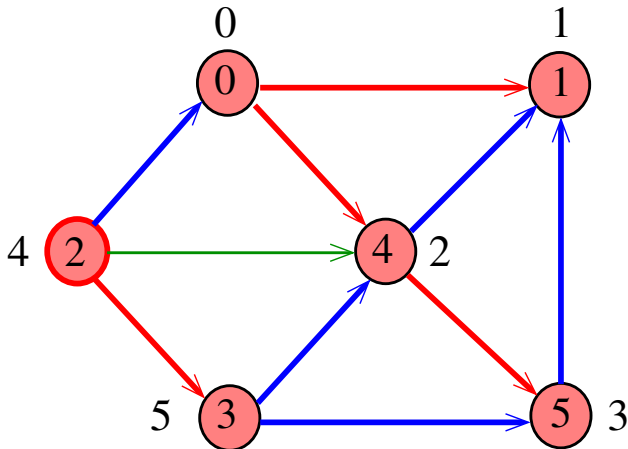
Exemplo

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



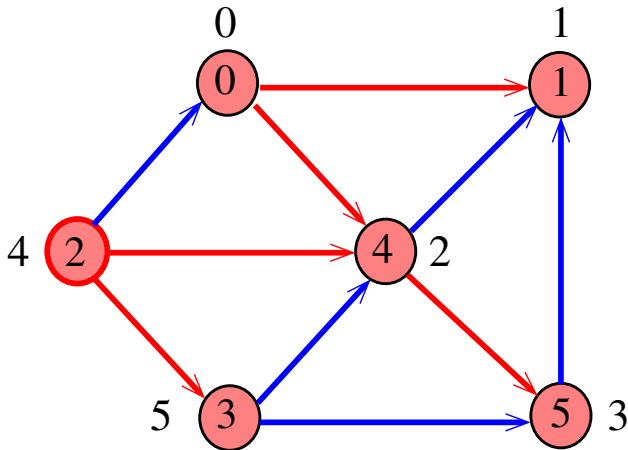
Exemplo

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



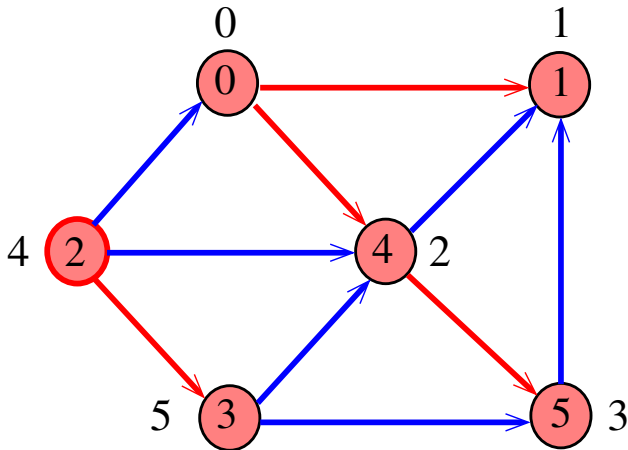
Exemplo

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



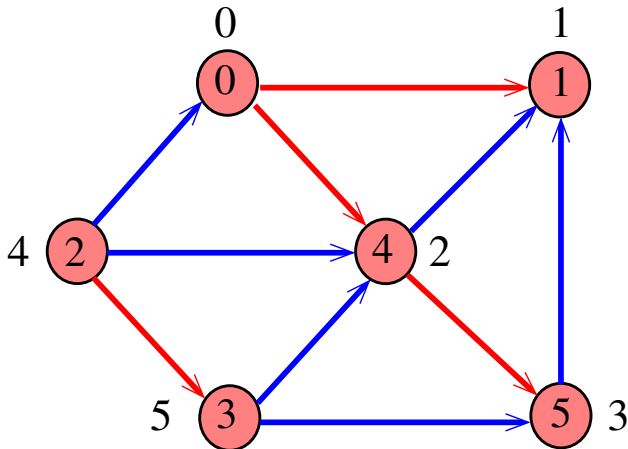
Exemplo

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



Exemplo

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



DAGts2

```
void DAGts2 (Digraph G, Vertex ts[]) {  
    Vertex v;  
1   cnt = G->V-1;  
2   for (v = 0; v < G->V; v++)  
3       ts[v] = lbl[v] = -1;  
4   for (v = 0; v < G->V; v++)  
5       if (lbl[v] == -1)  
6           TSdfsR(G, v, ts);  
}
```

DAGts2

void

```
TSdfsR (Digraph G, Vertex v, Vertex ts[])  
{  
    link p;  
1   lbl[v] = 0;  
2   for (p=G->adj[v]; p!=NULL; p=p->next)  
3       if (lbl[p->w] == -1)  
4           TSdfsR(G, p->w, ts);  
5   ts[cnt--] = v;  
}
```

Consumo de tempo

O consumo de tempo da função `DAGts2` para
vetor de listas de adjacência é $O(V + A)$.

Certificado de inexistência

Trecho de código que verifica se um vetor `ts[]` armazena uma ordenação topológica dos vértices de um grafo G

Certificado de inexistência

Trecho de código que verifica se um vetor `ts[]` armazena uma ordenação topológica dos vértices de um grafo `G`

```
[...]  
for (v = 0; v < G->V; v++)  
    idx[ts[v]] = v;  
for (v = 0; v < G->V; v++)  
    for (p=G->adj[v]; p!=NULL; p=p->next)  
        if (idx[v] > idx[p->w])  
            return ERRO;  
[...]
```

Adaptação de DIGRAPHcycle

Recebe um digrafo **G** e devolve **1** se existe um ciclo em **G** e devolve **0** em caso contrário.

Ademais, se a função devolve **0**, então a função devolve no vetor **ts[]** contém uma ordenação topológica dos vértices de **G**.

Supõe que o digrafo tem no máximo **maxV** vértices.

```
int DIGRAPHcycle (Digraph G);
```

Adaptação de DIGRAPHcycle

```
int DIGRAPHcycle (Digraph G, Vertex ts[]) {  
    Vertex v;  
1   time = 0;   cnt = G->V-1;  
2   for (v = 0; v < G->V; v++)  
3       d[v] = f[v] = parnt[v] = -1;  
4   for (v = 0; v < G->V, v++)  
5       if (d[v] == -1) {  
6           parnt[v] = v;  
7           if (cycleR(G, v, ts) == 1) return 1;  
        }  
8   return 0;  
}
```

Adaptação de cycleR

```
int cycleR (Digraph G, Vertex v, Vertex ts[])
{
    link p;
1   d[v] = time++;
2   for (p = G->adj[v]; p != NULL; p = p->next)
3       if (d[p->w] == -1) {
4           parnt[p->w] = v;
5           if(cycleR(G, p->w, ts) == 1) return 1;
        }
6       else if (f[w] == -1) return 1;
7   f[v] = time++; ts[cnt--] = v;
8   return 0;
}
```

Consumo de tempo

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

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

Conclusão

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

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