Melhores momentos

AULA 6

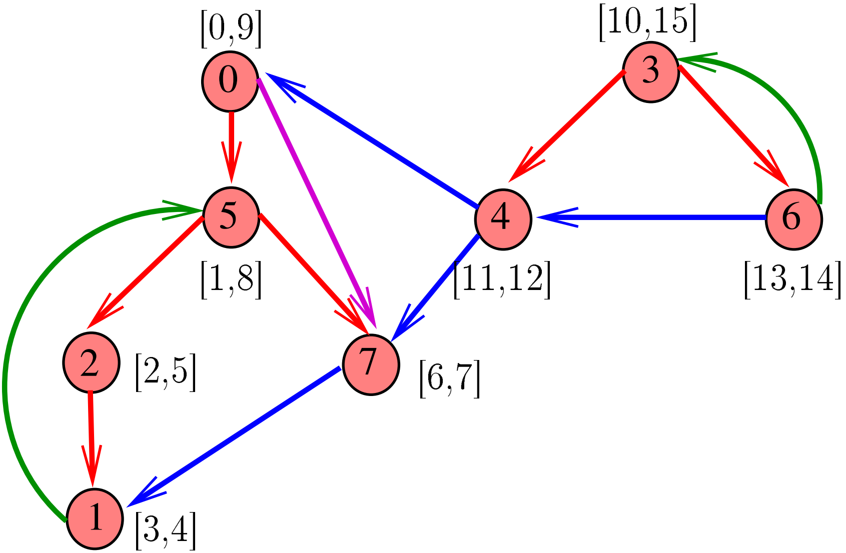
Busca DFS (CLRS)

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)



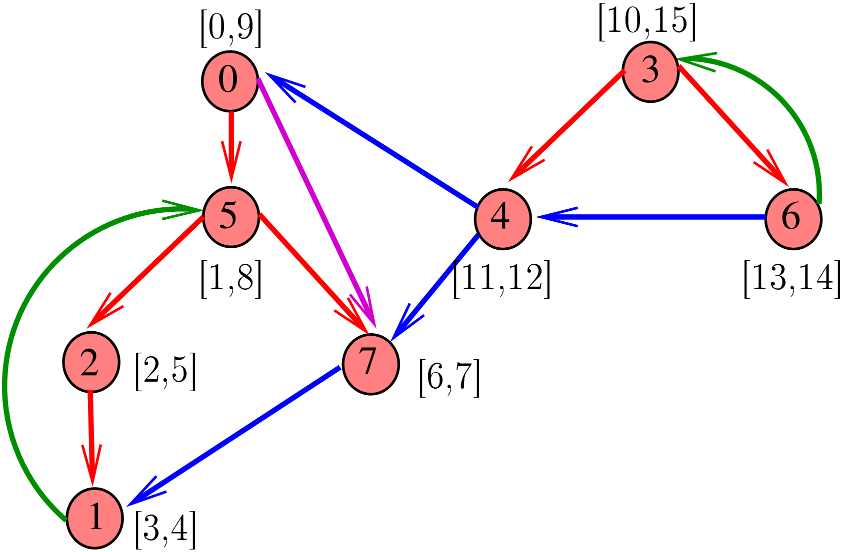
DIGRAPHdfs

```
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) {
            parnt[v] = v;
            dfsR(G, v);
        }
}
```

dfsR

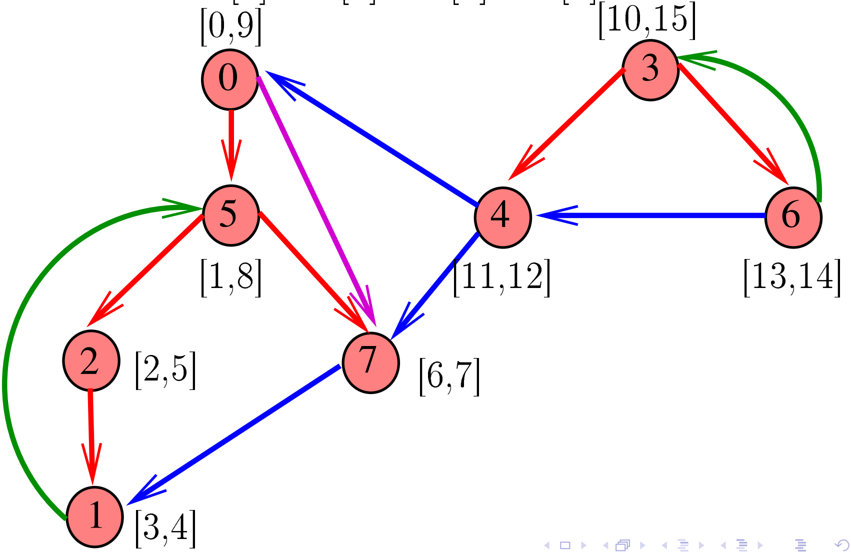
```
void dfsR (Digraph G, Vertex v) {
    link p;
    d[v] = time++;
    for (p = G->adj[v]; p != NULL; p= p->next)
        if (d[p->w] == -1) {
            parnt[p->w] = v;
            dfsR(G, p->w);
        }
    f[v] = time++;
}
```

Classificação dos arcos



Arcos de arborescência ou descendentes

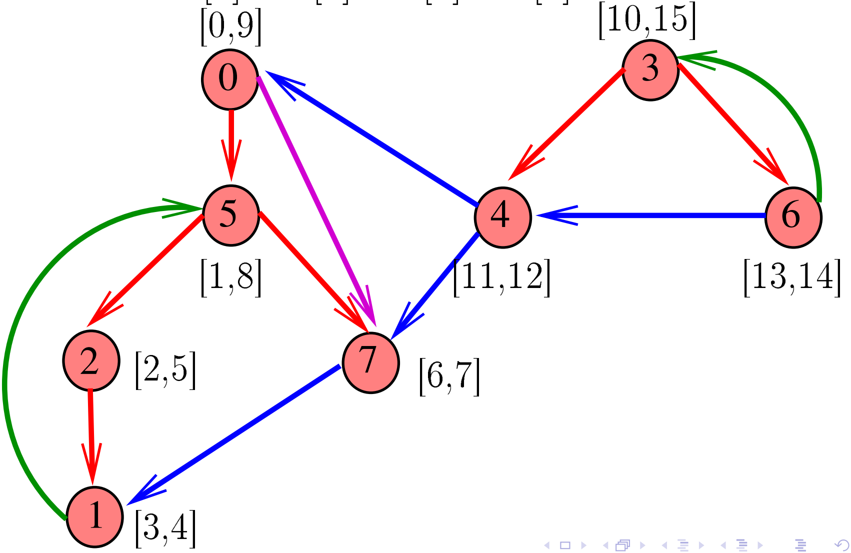
$\texttt{v-w}$ é **arco de arborescência** ou **descendente** se e somente se $\texttt{d}[\texttt{v}] < \texttt{d}[\texttt{w}] < \texttt{f}[\texttt{w}] < \texttt{f}[\texttt{v}]$



Arcos de retorno

$\texttt{v-w}$ é **arco de retorno** se e somente se

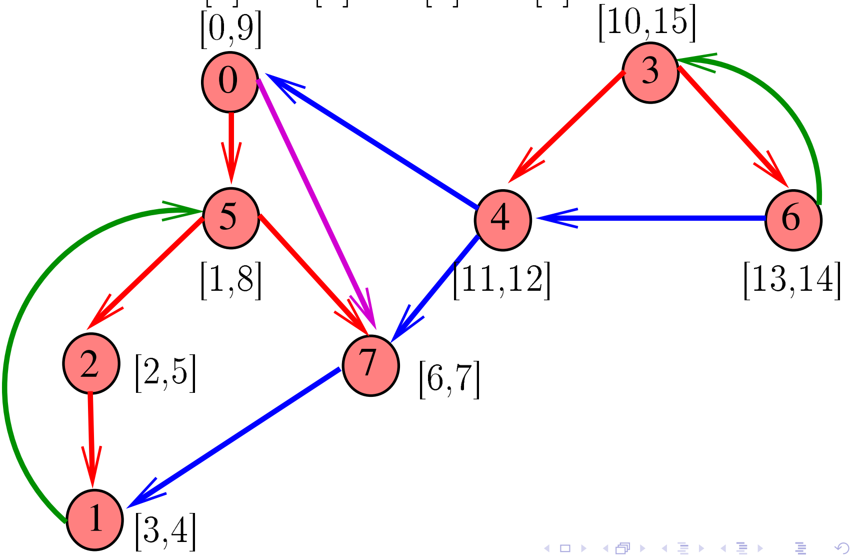
$$\texttt{d}[\texttt{w}] < \texttt{d}[\texttt{v}] < \texttt{f}[\texttt{v}] < \texttt{f}[\texttt{w}]$$



Arcos cruzados

`v-w` é arco **cruzado** se e somente se

$$\texttt{d}[\texttt{w}] < \texttt{f}[\texttt{w}] < \texttt{d}[\texttt{v}] < \texttt{f}[\texttt{v}]$$



Conclusões

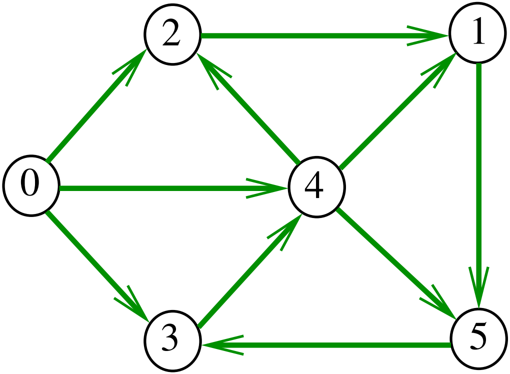
`v-w` é:

- **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
  $\mathtt{d}[\mathtt{w}] < \mathtt{f}[\mathtt{w}] < \mathtt{d}[\mathtt{v}] < \mathtt{f}[\mathtt{v}]$;

Procurando um ciclo

Problema: decidir se dado digrafo G possui um ciclo

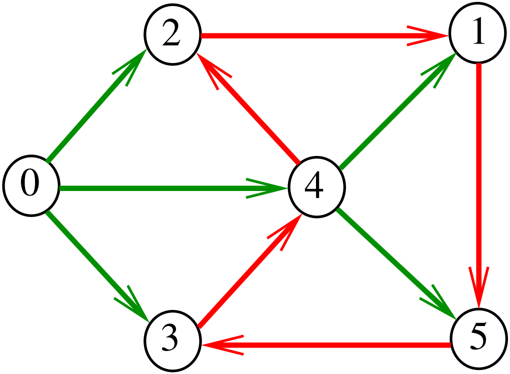
Exemplo: para o grafo a seguir a resposta é SIM



Procurando um ciclo

Problema: decidir se dado digrafo G possui um ciclo

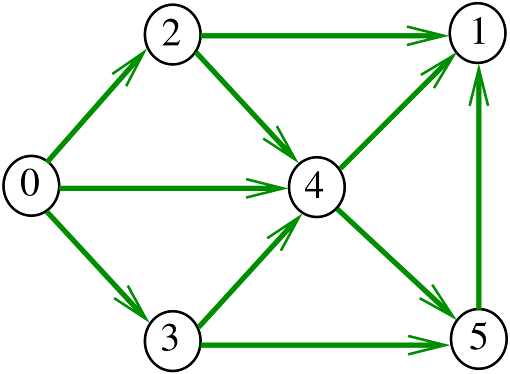
Exemplo: para o grafo a seguir a resposta é SIM



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

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

Primeiro algoritmo

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

Consumo de tempo

O consumo de tempo da função digraphcycle é A vezes o consumo de tempo da função DIGRAPHpath.

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

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

AULA 7

Ciclos em digrafos (continuação)

digraphcycle

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

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

digraphcycle

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

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

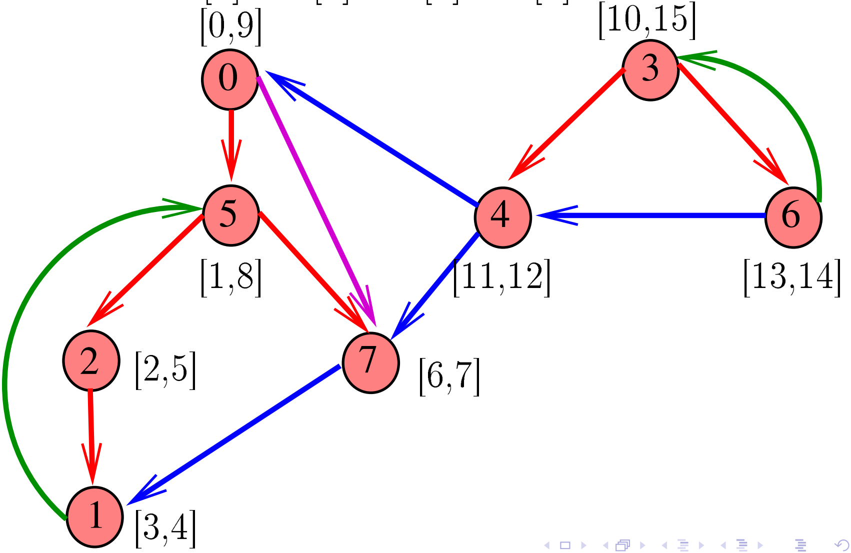
A função tem por base a seguinte observação: em relação a **qualquer** floresta de busca em profundidade,

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

Arcos de retorno

$\texttt{v-w}$ é **arco de retorno** se e somente se

$$\texttt{d}[\texttt{w}] < \texttt{d}[\texttt{v}] < \texttt{f}[\texttt{v}] < \texttt{f}[\texttt{w}]$$



digraphcycle

```
int digraphcycle (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) {
            parnt[v] = v;
            if (cycleR(G,v) == 1) return 1;
        }
    return 0;
}
```

cycleR

```
int cycleR (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[p->w] = v;
            if (cycleR(G, p->w)==1) return 1;
        }
        else if (f[w] == -1) return 1;
    f[v] = time++;
    return 0;
}
```

Consumo de tempo

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

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

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

Recebe um digrafo G e vértices s e t, além do vetor parnt computado pela chamada

```
digraphcycle(G);
```

A função devolve 1 se

```
t-parnt[t]-parnt[parnt[t]]-...
```

é o reverso de um caminho de s a t em G ou devolve 0 em caso contrário

```
int st_caminho (Digraph G, Vertex s, Vertex t);
```

st_caminho

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

Digrafos acíclicos (DAGs)

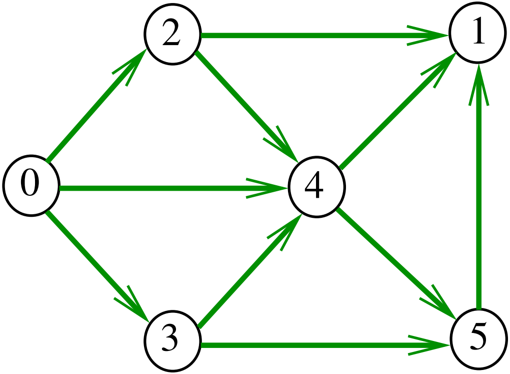
S 19.5 e 19.6

DAGs

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

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

Exemplo: um digrafo acíclico

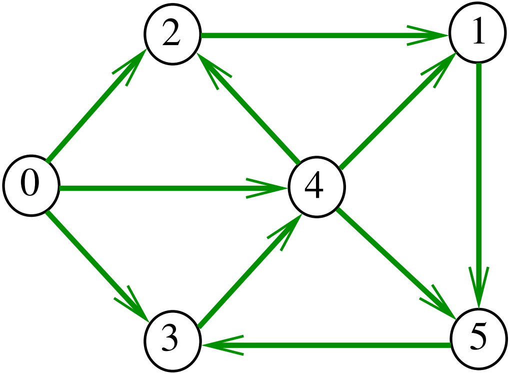


DAGs

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

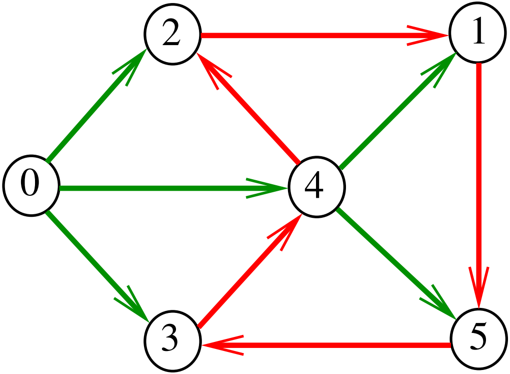


DAGs

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



Ordenaçao 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],..., ts[V-1]`

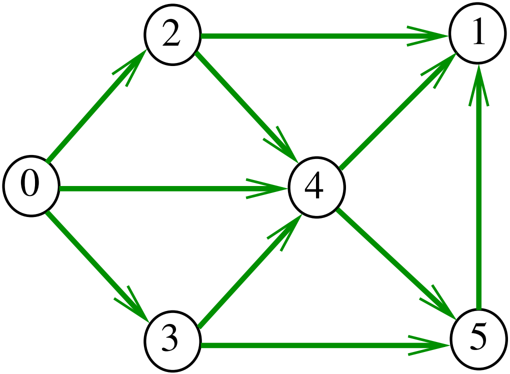
dos seus vértices tal que todo arco tem a forma

`ts[i]-ts[j]` 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

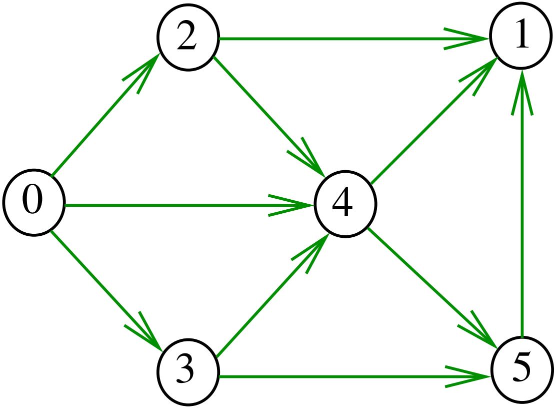
Armazena em `ts[0 . . i-1]` uma permutação de um subconjunto do conjunto de vértices de `G`e devolve o valor de `i`

Se `i` $=$ `G`$->$ `V` então `ts[0 . . i-1]` é uma ordenação topológica de G.
Caso contrário, `G` **não** é um DAG

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

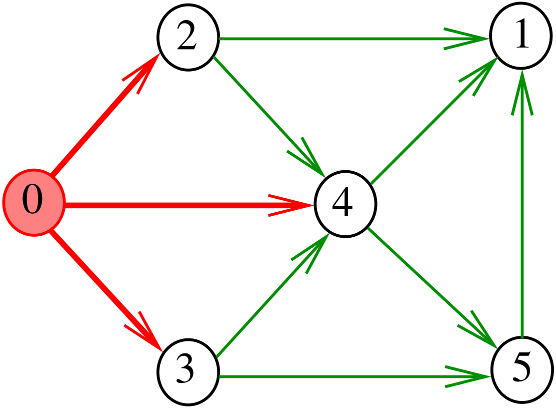
Exemplo

i	0	1	2	3	4	5
ts[i]						



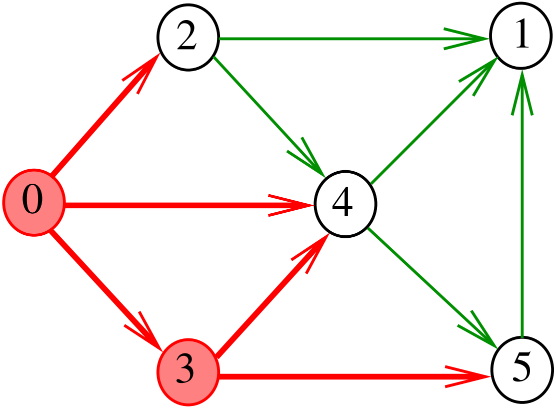
Exemplo

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



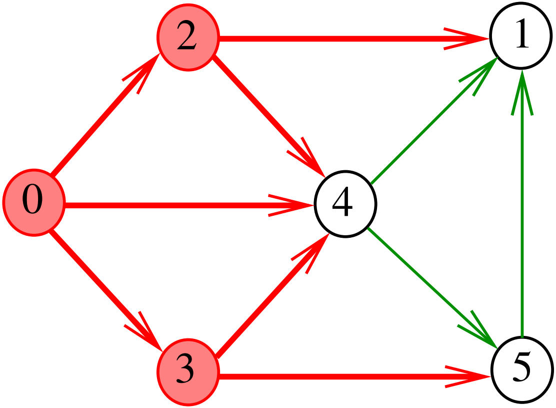
Exemplo

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



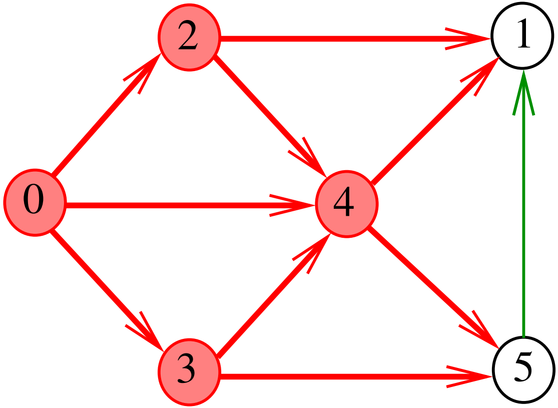
Exemplo

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



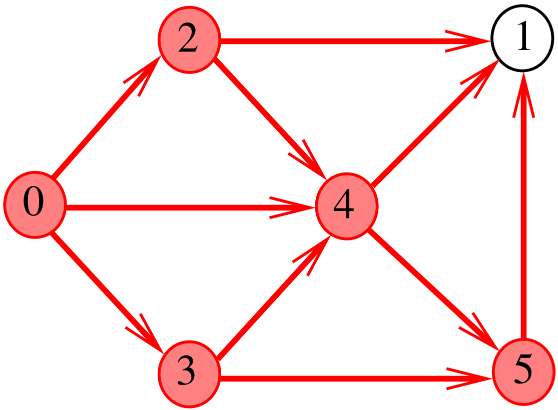
Exemplo

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



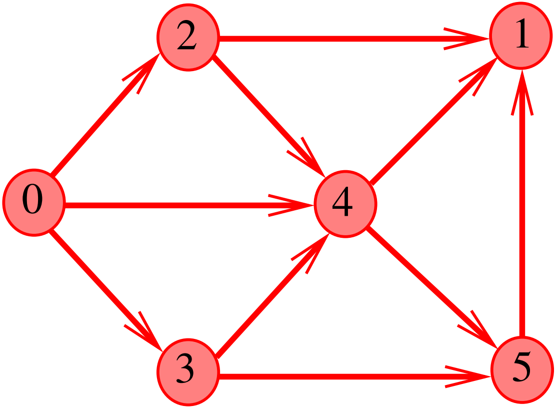
Exemplo

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



DAGtsf

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

DAGtsf

```
QUEUEinit(G->V);
for(v = 0; v < G->V; v++)
    if (in[v] == 0)
        QUEUEput(v);
for (i = 0; !QUEUEempty(); i++) {
    ts[i] = v = QUEUEget();
    for (p=G->adj[v];p!=NULL;p=p->next)
        if (--in[p->w] == 0)
            QUEUEput(p->w);
}
QUEUEfree();
return i;
}
```

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 `DAGtsf` para vetor de listas de adjacência é $O(\mathtt{V} + \mathtt{A})$.