Digrafos acíclicos (DAGs)

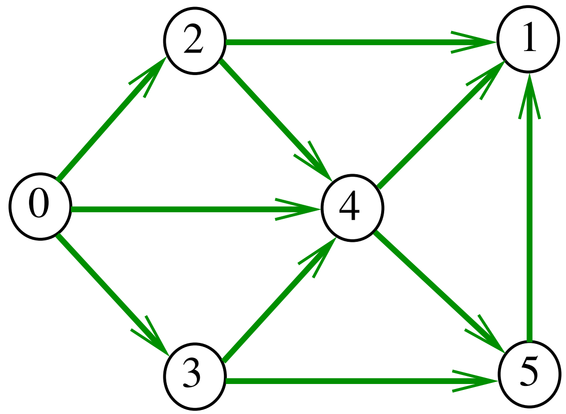
S 19.5 e 19.6

DAGs

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

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

Exemplo: um digrafo acíclico



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]`

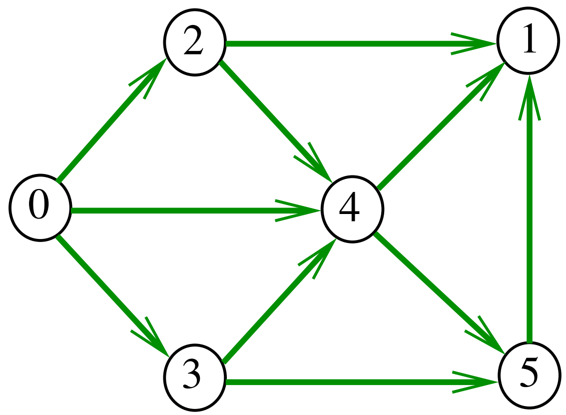
dos seus vértices tal que todo arco tem a forma

`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, ou
- ▷ uma ordenação topológica do digrafo.

Algoritmos de ordenação topológica

S 19.6

Algoritmo de eliminação de fontes

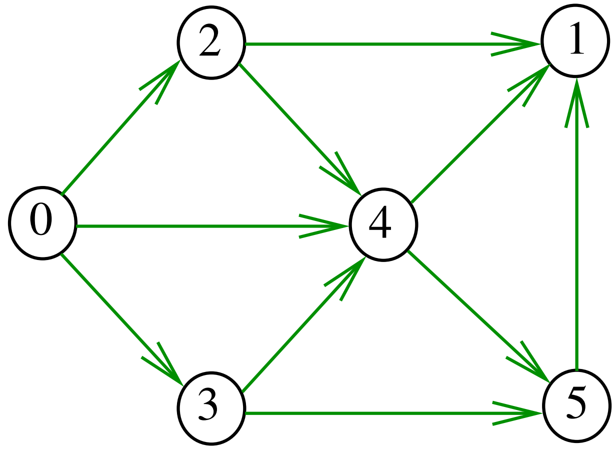
Armazena em `ts[0 . . i-1]` uma permutação de um subconjunto do conjunto de vértices de `G` e devolve o valor de `i`

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]						



i 0 1 2 3 4 5

DAGts1

```
int DAGts1 (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]++;
```


DAGts1

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

Algoritmos de ordenação topológica

S 19.6

Algoritmo DFS

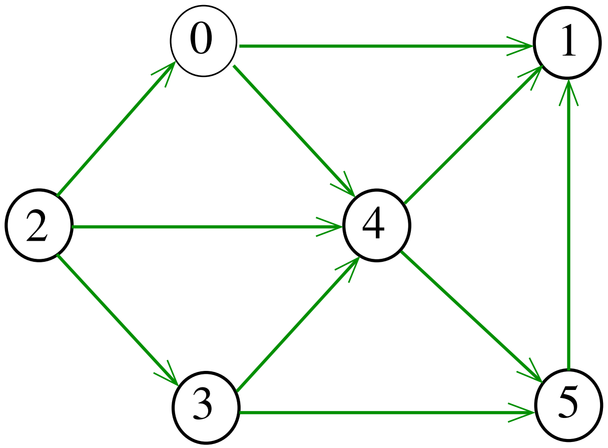
```
static int cnt;
static int lbl[maxV];
```

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

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

Exemplo

i	0	1	2	3	4	5
ts[i]						



i | 0 1 2 3 4 5

DAGts2

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


DAGts2

```
void
TSdfsR (Digraph G, Vertex v, Vertex ts[])
{
    link p;
    lbl[v] = 0;
    for (p=G->adj[v]; p!=NULL; p=p->next)
        if (lbl[p->w] == -1)
            TSdfsR(G, p->w, ts);
    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 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;
    time = 0;    cnt = G->V-1;
    for (v = 0; v < G->V; v++)
        d[v] = f[v] = parnt[v] = -1;
    for (v= 0; v < G->V;v++)
        if (d[v] == -1) {
            parnt[v] = v;
            if (cycleR(G,v,ts)) return 1;
        }
    return 0;
}
```


Adaptação de cycleR

```
int cycleR (Digraph G,Vertex v,Vertex ts[])
{
    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,ts)==1) return 1;
        }
        else if (f[w] == -1) return 1;
    f[v] = time++; ts[cnt--] = v;
    return 0;
}
```

Problema

- Adaptar essa função para devolver um ciclo, quando houver.
  Pode ser devolvida uma lista ligada de vértices, ou
  um vetor de vértices ou
  um apontador para lista ou vetor, mallocado na função.

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 umas das seguintes afirmações:

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