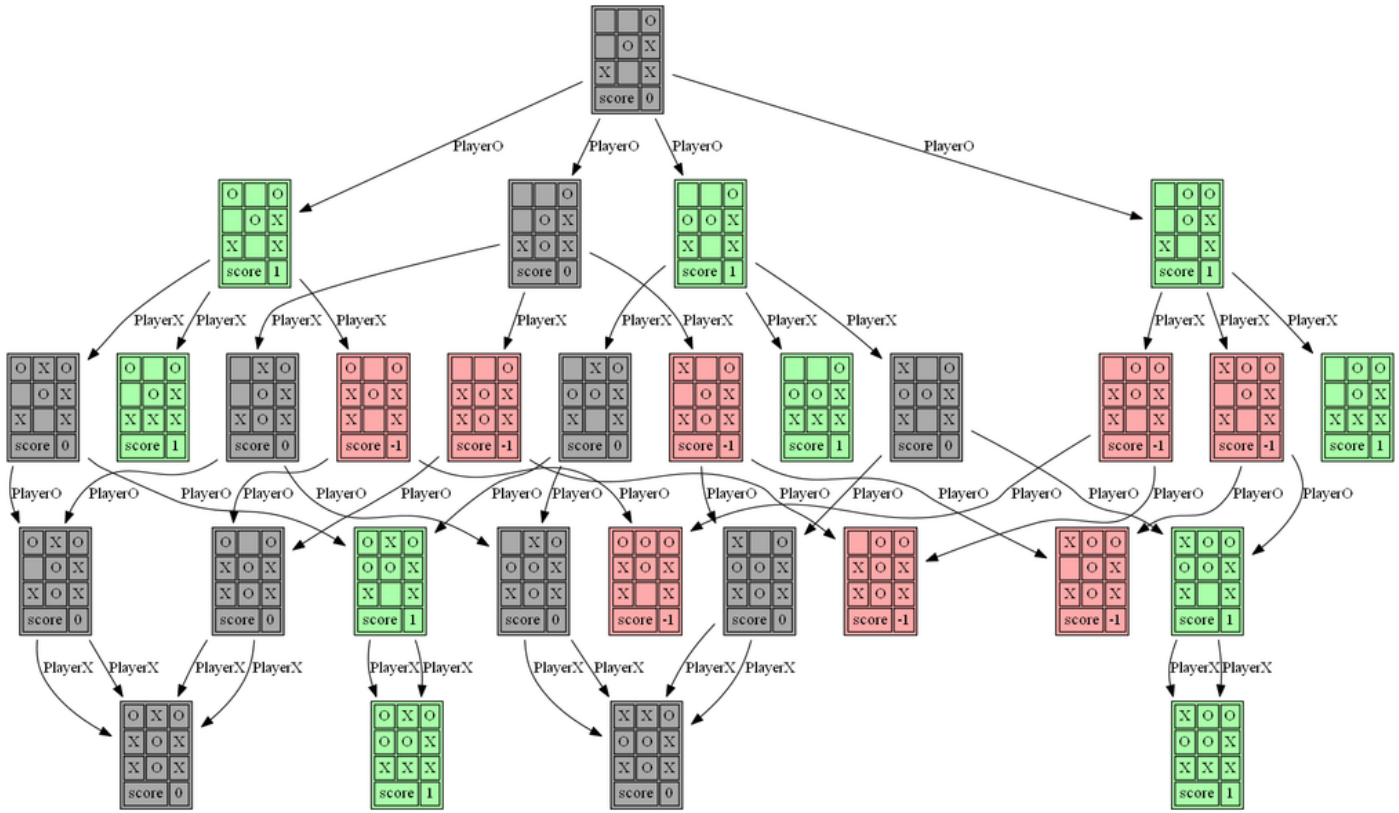
MAC0323 Algoritmos e Estruturas de Dados II

Edição 2020 – 2

AULA 20

Anatomia da busca em profundidade

Classificação dos arcos



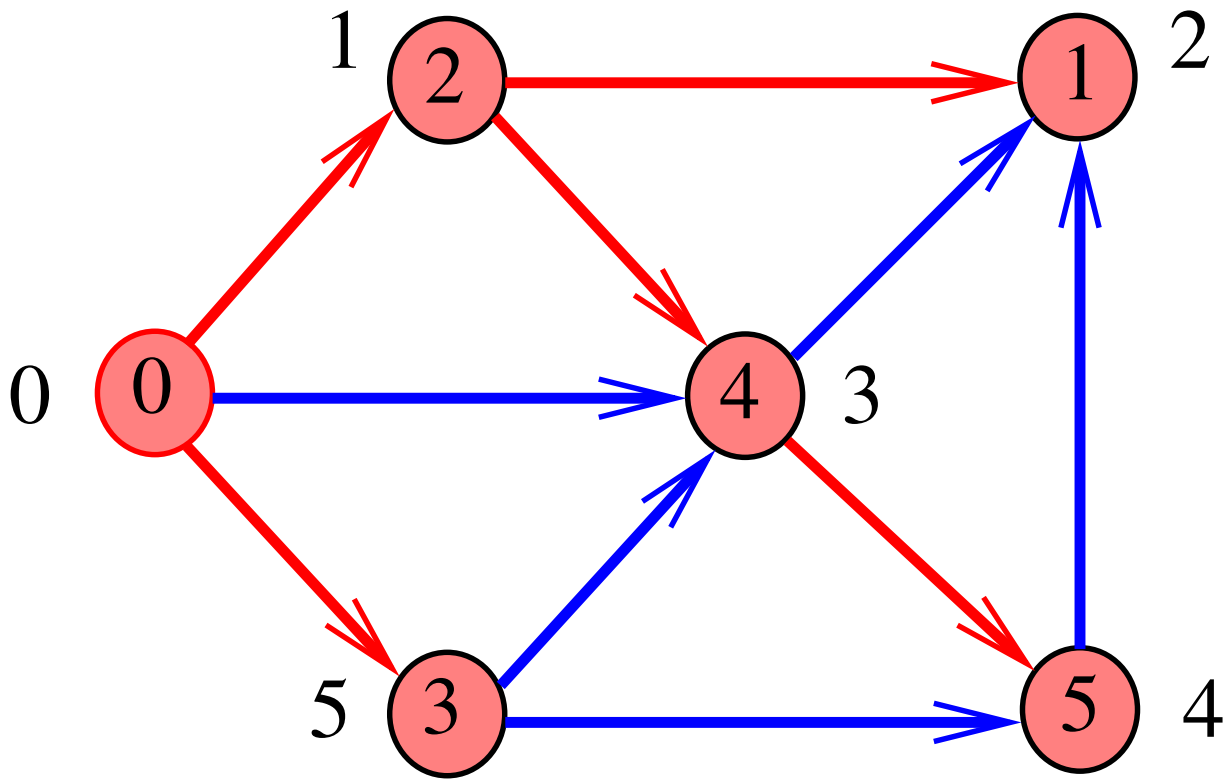
Fonte: `Using Minimax (with the full game tree) to implement the machine players ...`

Referências: CLRS 22

Arcos da arborescência

Arcos da arborescência são os arcos `v-w` que `dfs()` percorre para visitar `w` pela primeira vez.

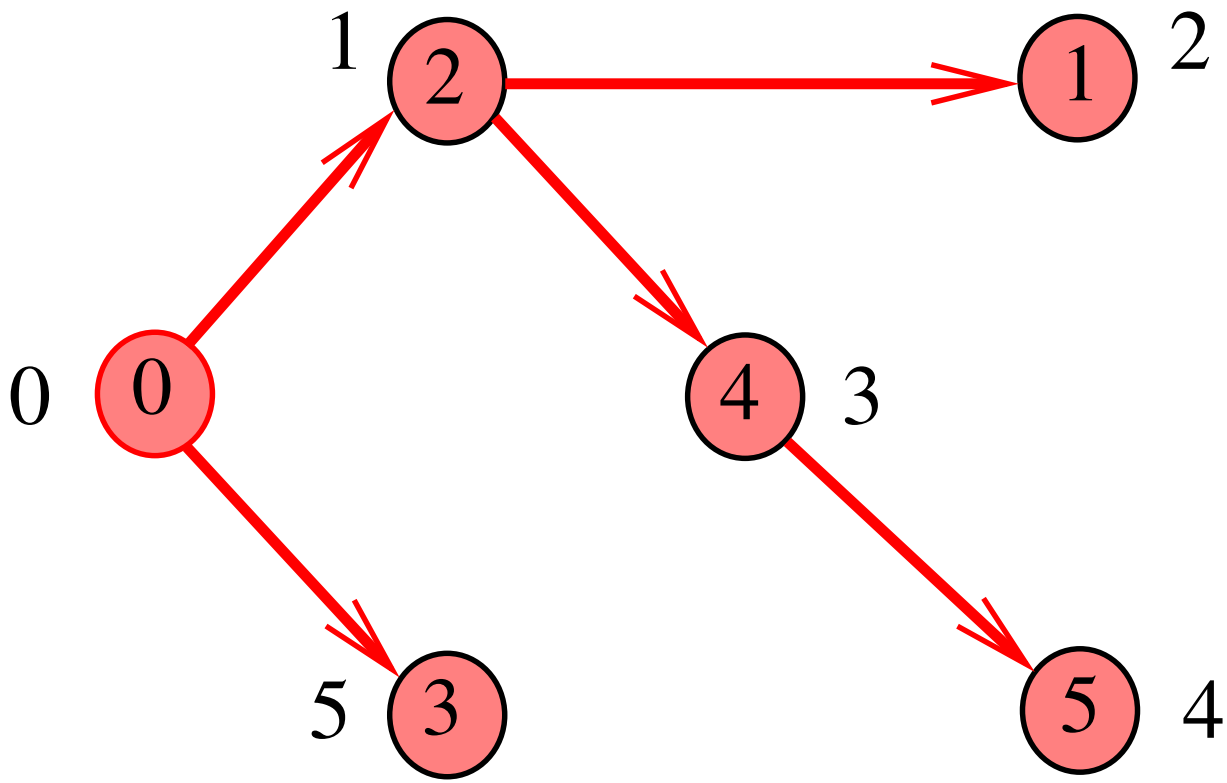
Exemplo: arcos em vermelho são arcos da arborescência



Arcos da arborescência

Arcos da arborescência são os arcos `v-w` que `dfs()` percorre para visitar `w` pela primeira vez.

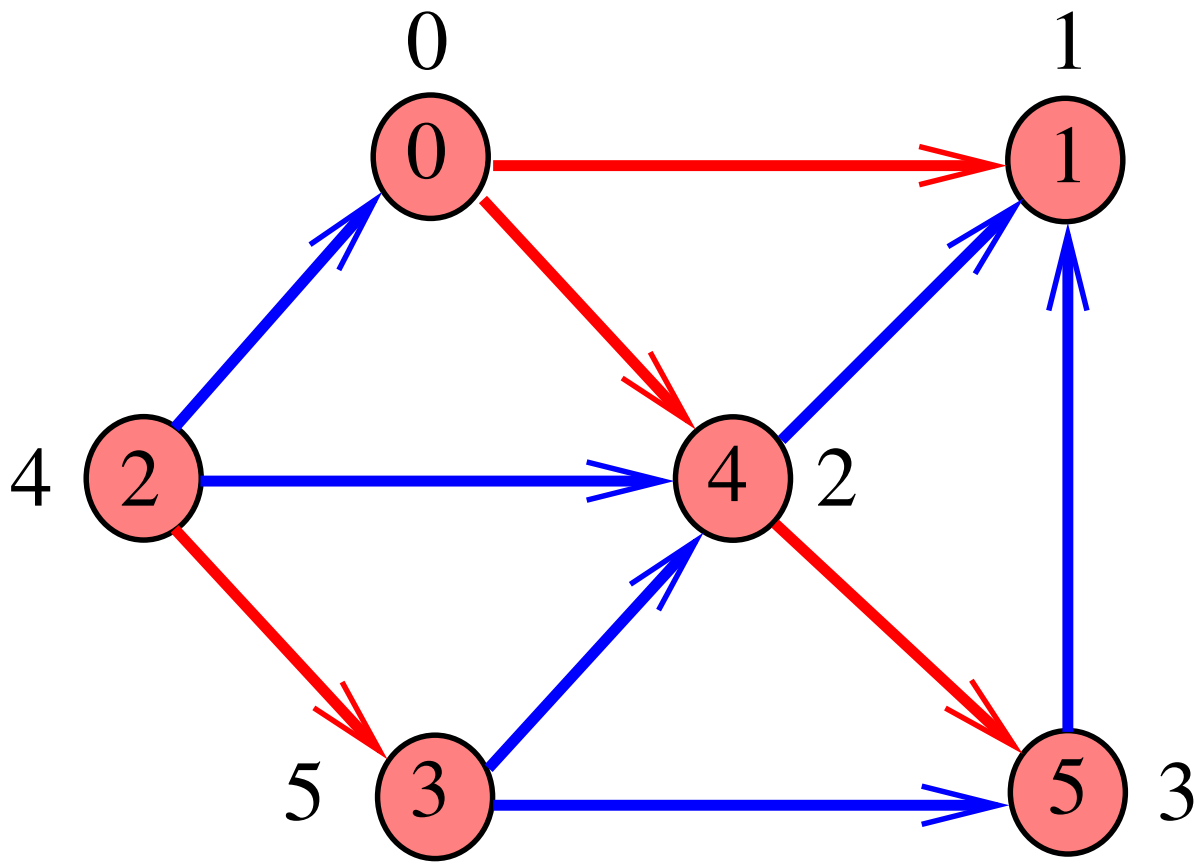
Exemplo: arcos em vermelho são arcos da arborescência



Floresta DFS

Conjunto de arborescências é a **floresta da busca em profundidade** (= *DFS forest*).

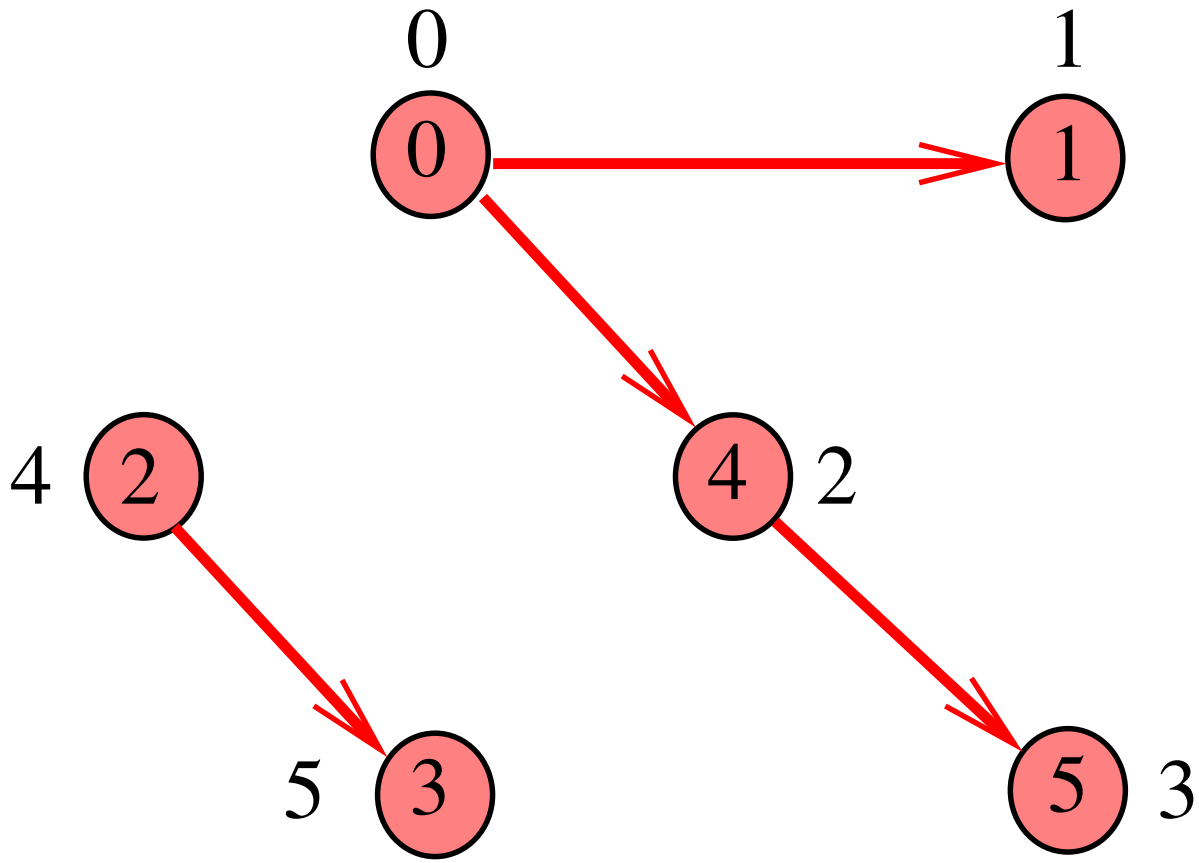
Exemplo: arcos em vermelho formam a floresta DFS



Floresta DFS

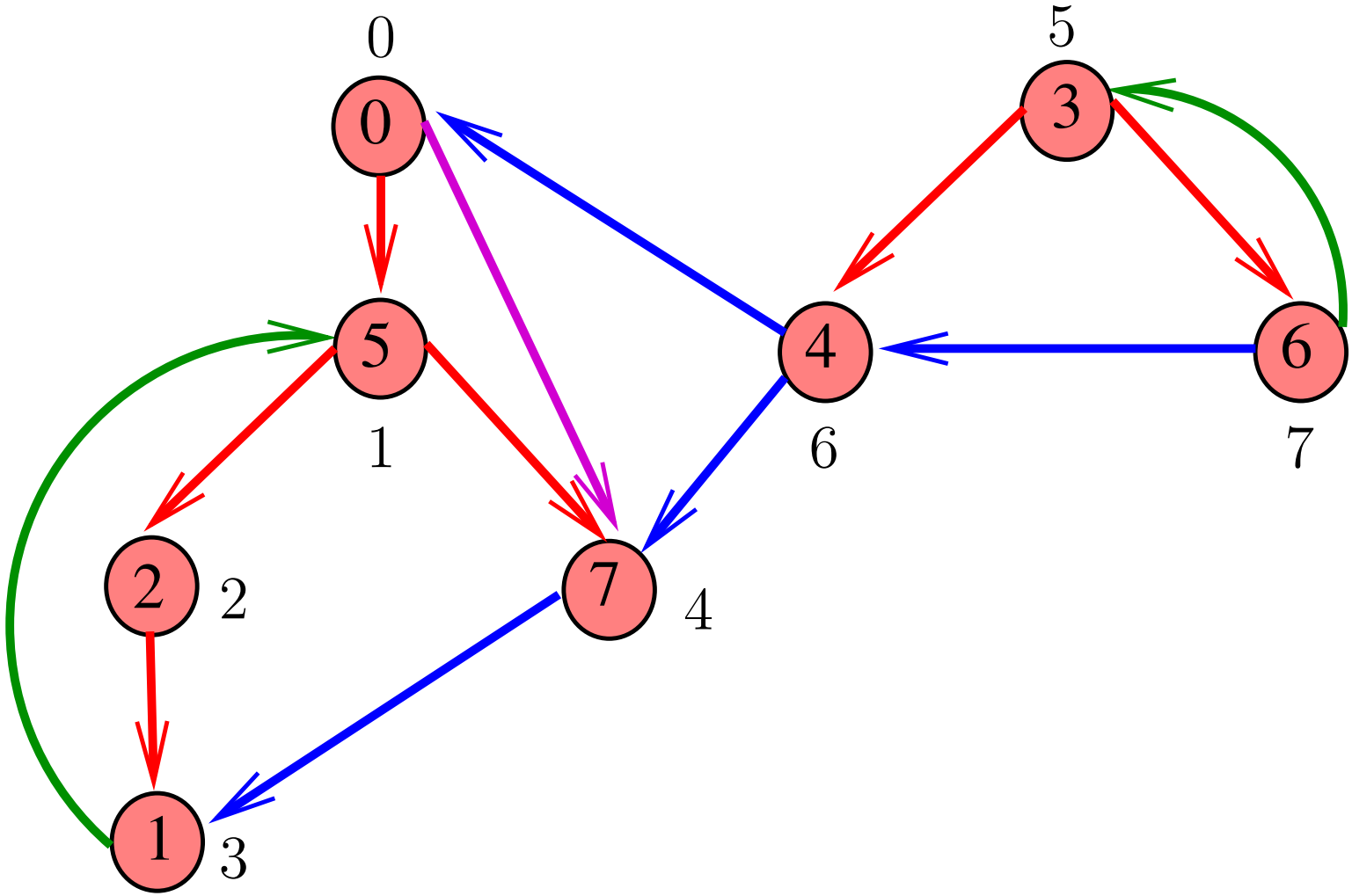
Conjunto de arborescências é a **floresta da busca em profundidade** (= *DFS forest*).

Exemplo: arcos em vermelho formam a floresta DFS



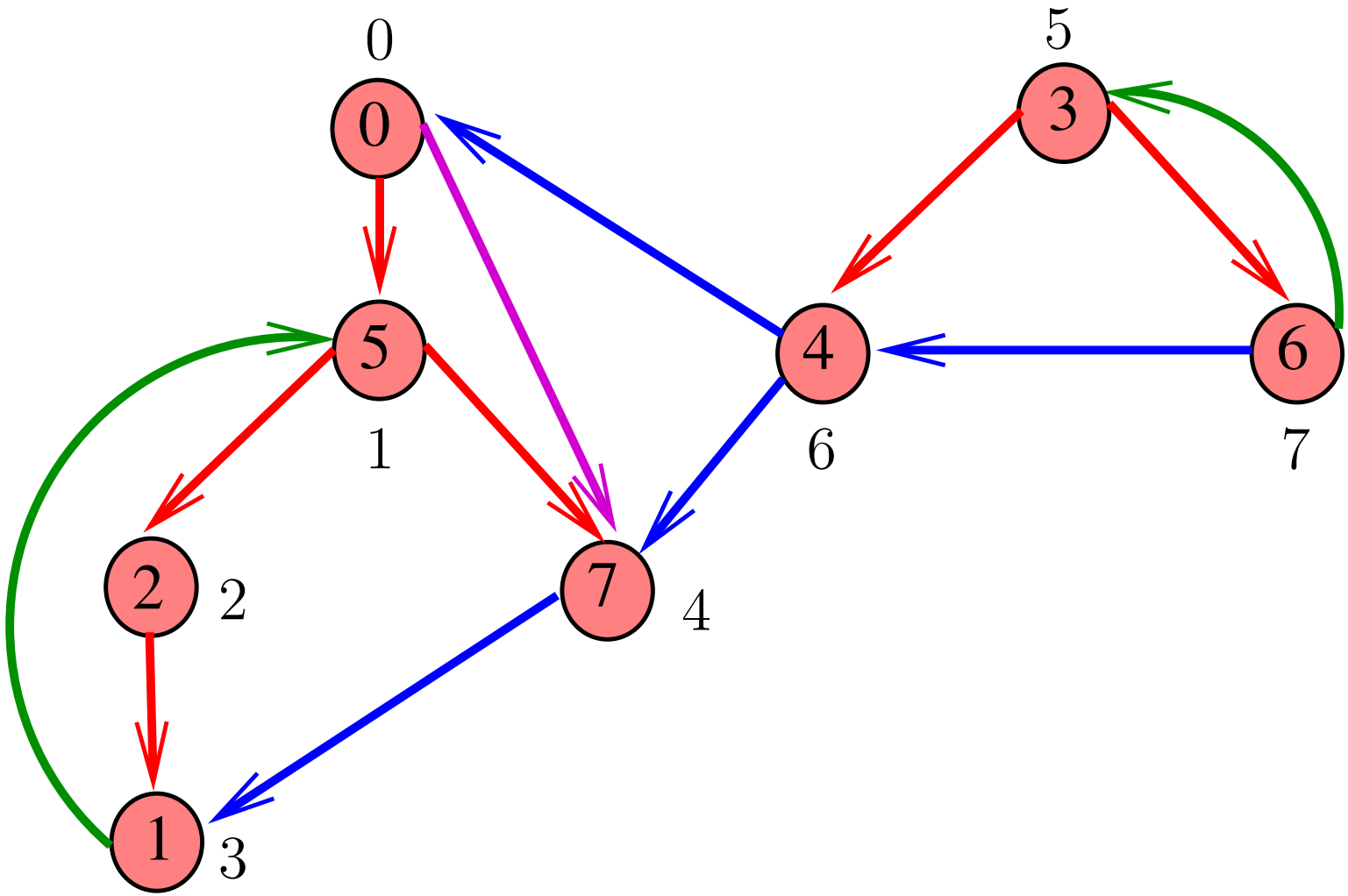
Arcos de arborescência

`v-w` é **arco de arborescência** se foi usado para visitar `w` pela primeira vez (arcos **vermelhos**).



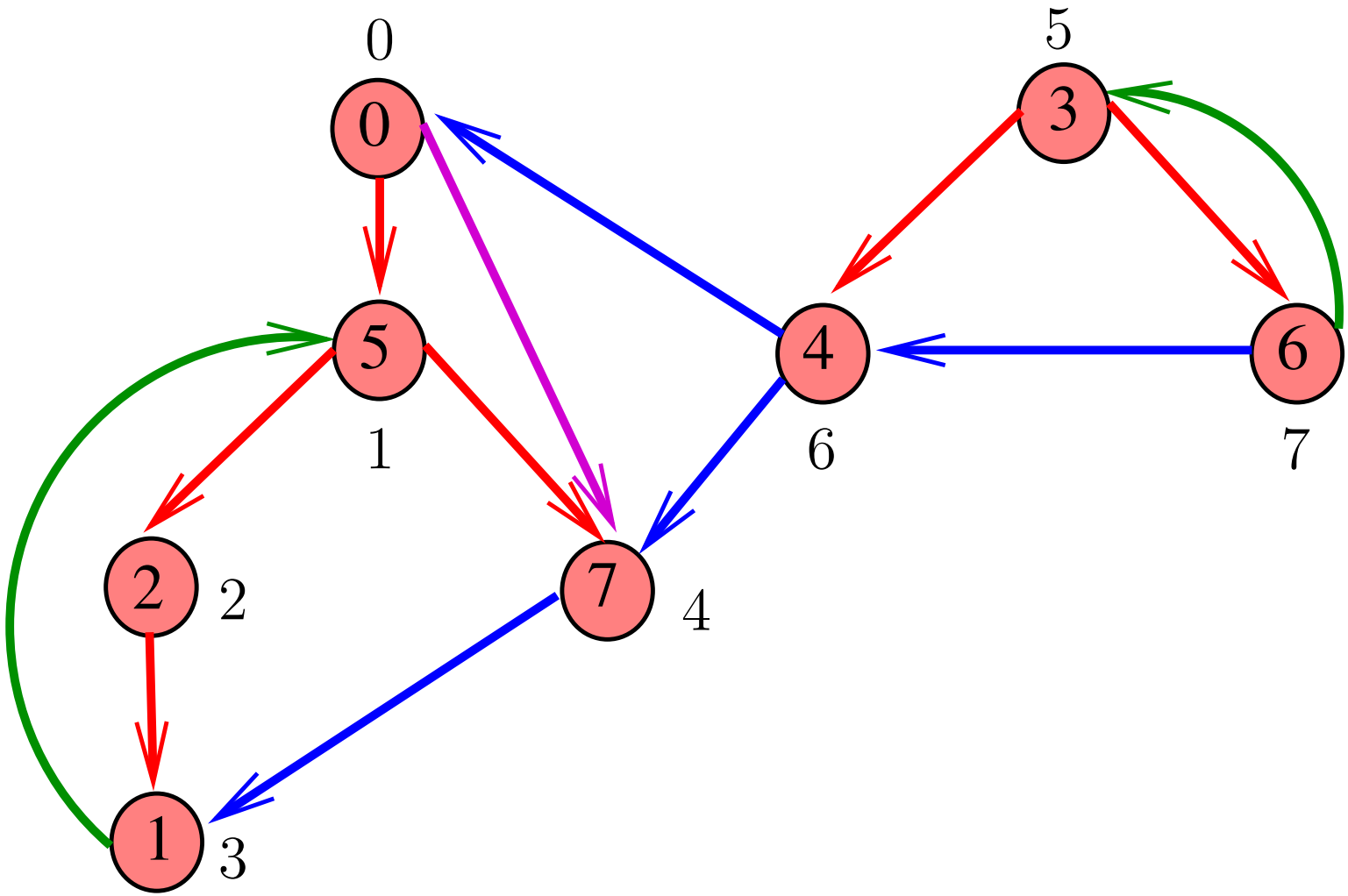
Arcos de retorno

v-w é **arco de retorno** se w é ancestral de v (arcos **verdes**).



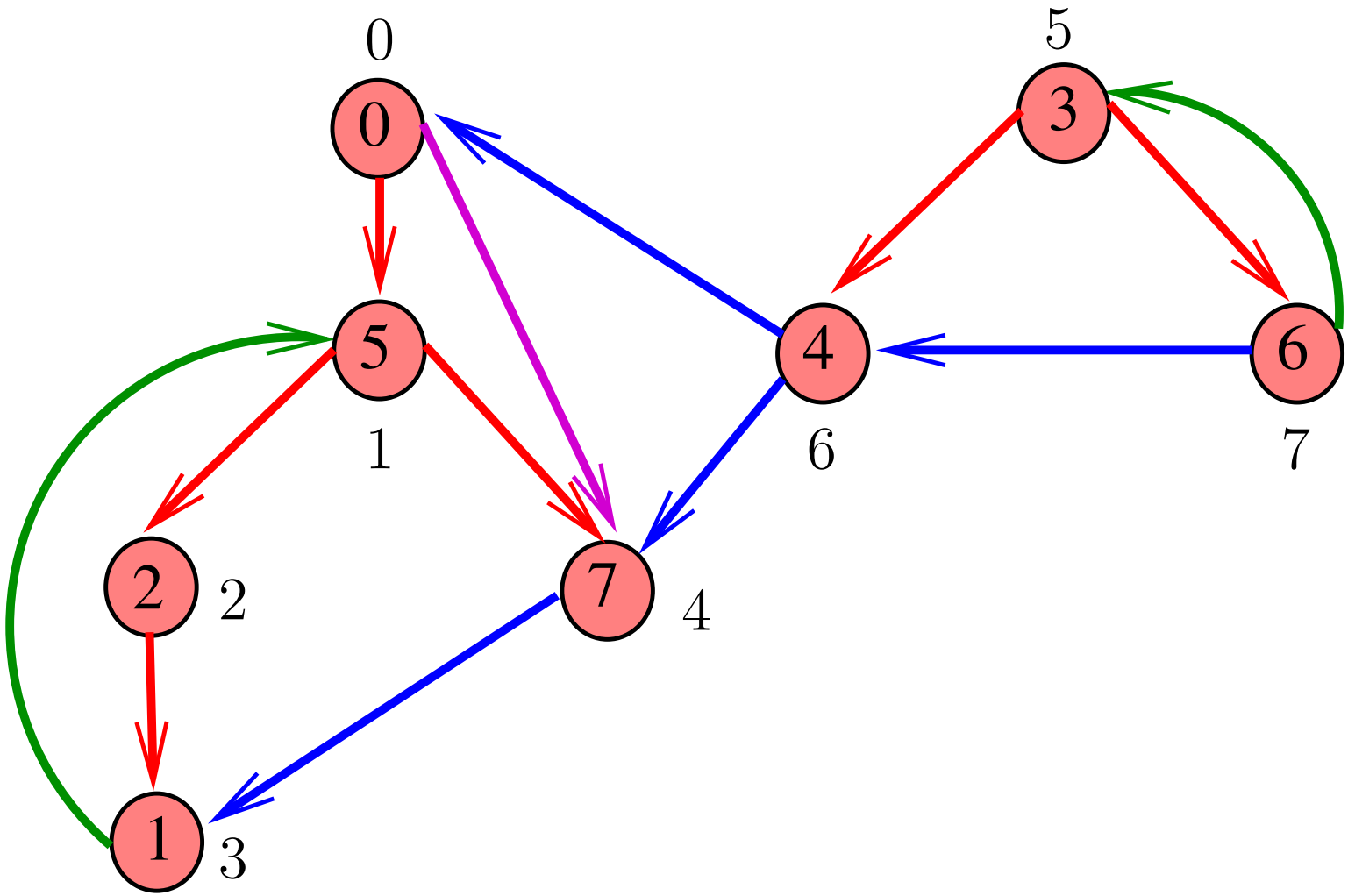
Arcos descendentes

v-w é **descendente** se w é descendente de v, mas não é filho de v (arco **roxo**).



Arcos cruzados

v-w é **arco cruzado** se w não é ancestral nem descendente de v (arcos **azuis**).



Como obter a anatomia da DFS (CLRS)

O digrafo `G` tem `G->V` vértices

Fazemos uma DFS que
visita todos os vértices e arcos do digrafo `G`.

Como obter a anatomia da DFS (CLRS)

O digrafo `G` tem `G->V` vértices

Fazemos uma DFS que
visita todos os vértices e arcos do digrafo `G`.

```
static int time;
static int *d = mallocSafe(G->V*sizeof(int));
static int *f = mallocSafe(G->V*sizeof(int));
```

A função registra em `d[v]` o 'momento' em
que `v` foi descoberto e em `f[v]` o momento em
que `v` foi completamente examinado.

DFSanatomia: estrutura

```
static struct dfsanatomia {
    bool *marked;
    int  *edgeTo;
```

DFSanatomia: estrutura

```
static struct dfsanatomia {
   bool *marked;
   int  *edgeTo;

   int  time;
   int  *d;                          /* discovered */
   int  *f;                          /* finished */
```

DFSanatomia: estrutura

```
static struct dfsanatomia {
   bool *marked;
   int  *edgeTo;

   int  time;
   int  *d;                          /* discovered */
   int  *f;                           /* finished */

   Queue pre;                         /* pré-ordem */
   Queue pos;                         /* pós-ordem */
   Stack revPos;                /* pós-ordem reversa */
}

typedef struct dfscc *dfsAnatomia;
```

DFSanatomia: construtor

```
dfsAnatomia DFSanatomiaInit(Digraph G) {
    dfsAnatomia T = mallocSafe(sizeof(*dfs));
    /* DFSPathsInit */
    ...
    T->time = 0;
    T->d = mallocSafe(G->V*sizeof(int));
    T->f = mallocSafe(G->V*sizeof(int));

    T->pre = queueInit();
    T->pos = queueInit();
    T->revPos = stackInit();
```

DFSanatomia: construtor

```
dfsAnatomia DFSanatomiaInit(Digraph G) {
    dfsAnatomia T = mallocSafe(sizeof(*dfs));
    /* DFSPathsInit */
    ...
    T->time = 0;
    T->d = mallocSafe(G->V*sizeof(int));
    T->f = mallocSafe(G->V*sizeof(int));

    T->pre = queueInit();
    T->pos = queueInit();
    T->revPos = stackInit();

    for (int v = 0; v < G->V; v++)
        if (!T->marked(v))
            dfs(G, v, T);
    return T;
}
```

dfs() original

```
static void dfs(Digraph G, int v, dfsPaths T) {
    Link w;
    T->marked[v] = true;
    for (w = G->adj[v]; w != NULL; w = w->next)
        if (!T->marked[w->vertex]) {
            T->edgeTo[w->vertex] = v;
            dfs(G, w->vertex, T);
        }
    }
}
```

Alterações na dfs()

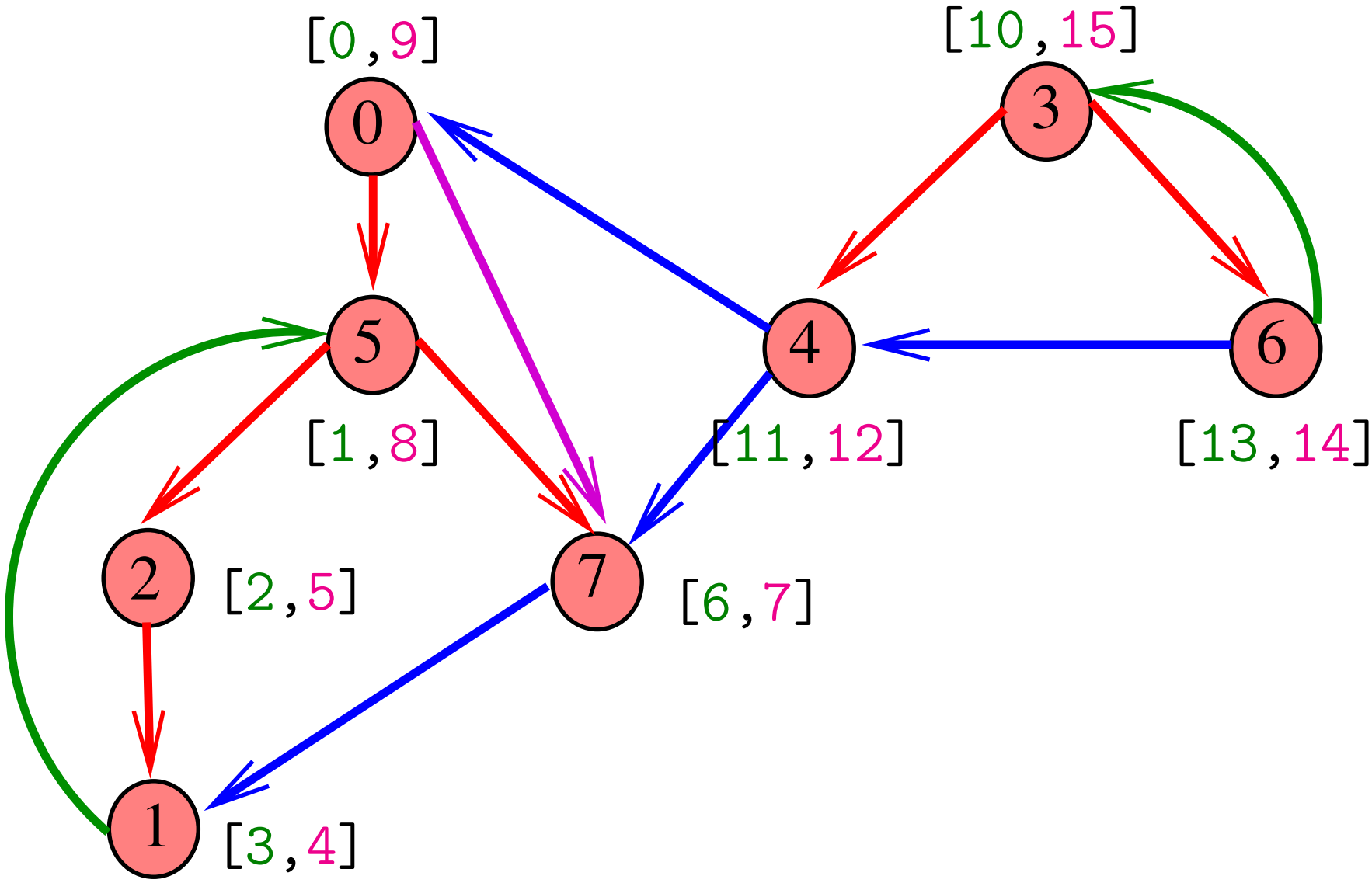
```
static void dfs(Digraph G, int v, dfsAnatomia T) {
    Link w;
    T->marked[v] = true;

    T->d[v] = (T->time)++;                    /* descoberto */
    enqueue(T->pre, v);                        /* pré-ordem */

    for (w = G->adj[v]; w != NULL; w = w->next)
        if (!T->marked[w->vertex]) {
            T->edgeTo[w->vertex] = v;
            dfs(G, w->vertex, T);
        }

    enqueue(T->pos, v);                       /* pós-ordem */
    push(T->revPos, v);                /* pós-ordem reversa */
    T->f[v] = (T->time)++;                    /* terminamos */
}
```

Consumo de tempo

Determinar a anatomia da DFS, para vetor de listas de adjacência, consome tempo $O(V + E)$.

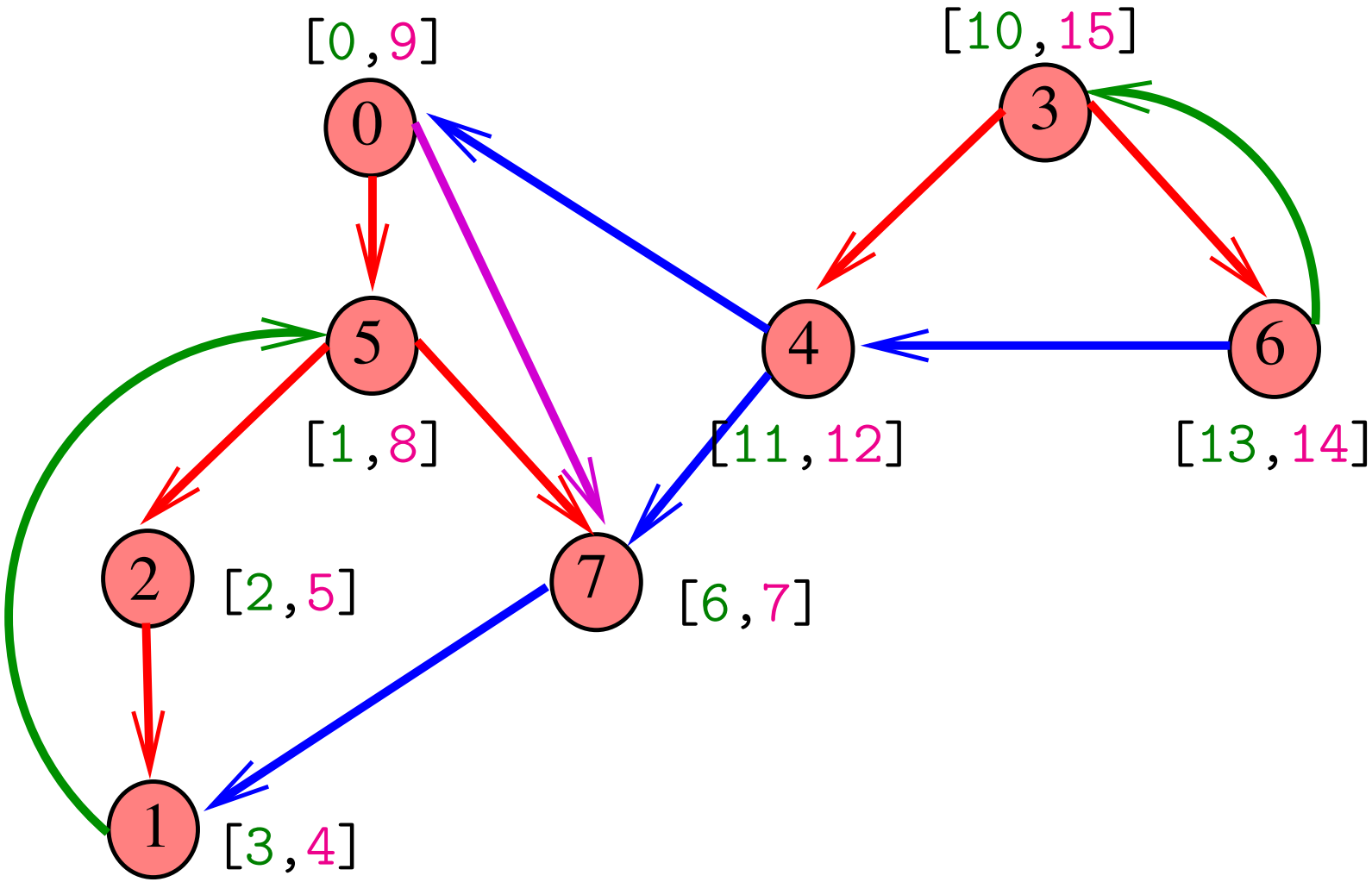
Determinar a anatomia da DFS, para matriz de adjacências, consome tempo $O(V^2)$.

Classificação dos arcos



Arcos de arborescência ou descendentes

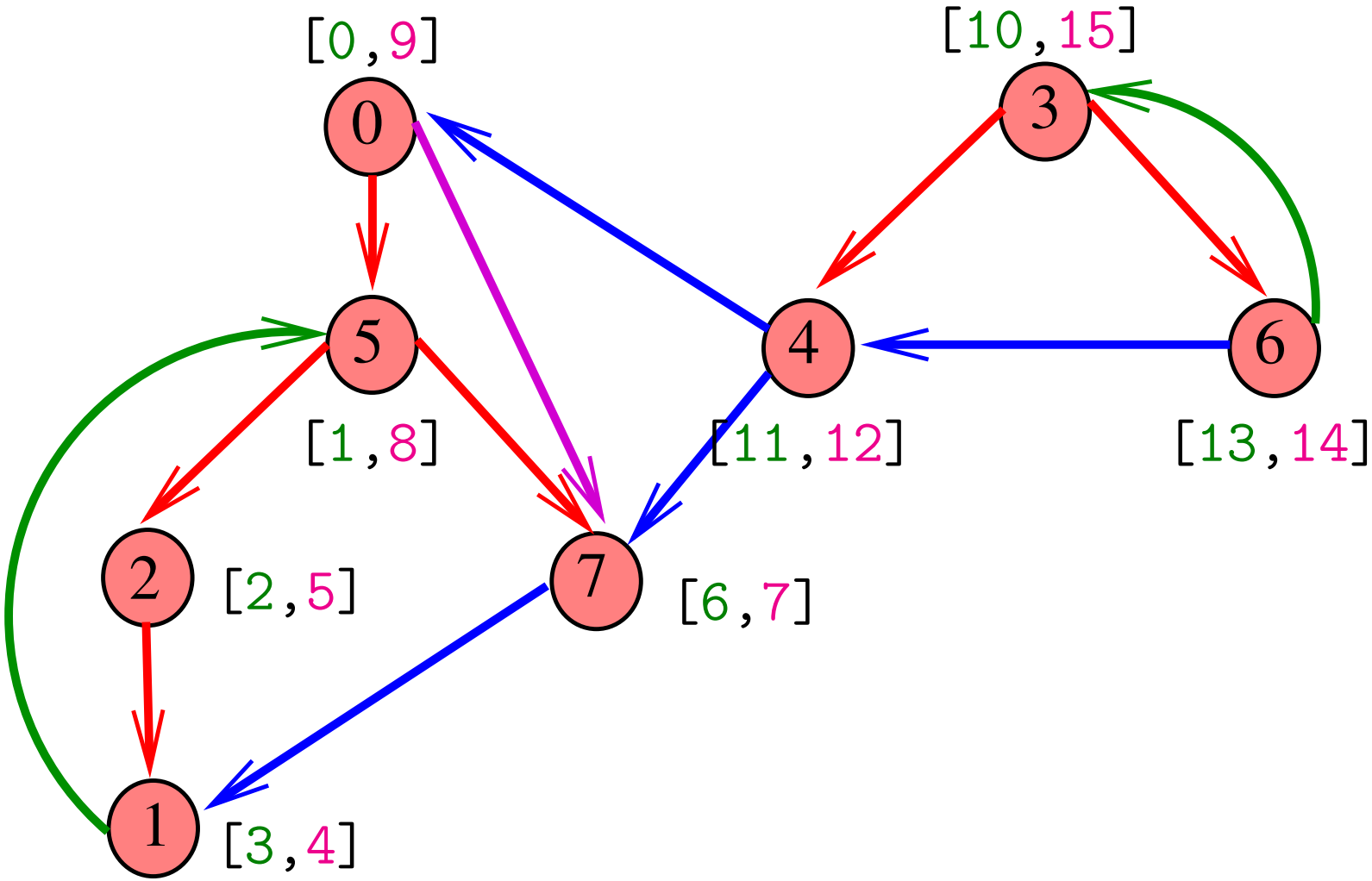
v-w é **arco de arborescência** ou **descendente** se e somente se $d[v] < d[w] < f[w] < f[v]$.



Arcos de retorno

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

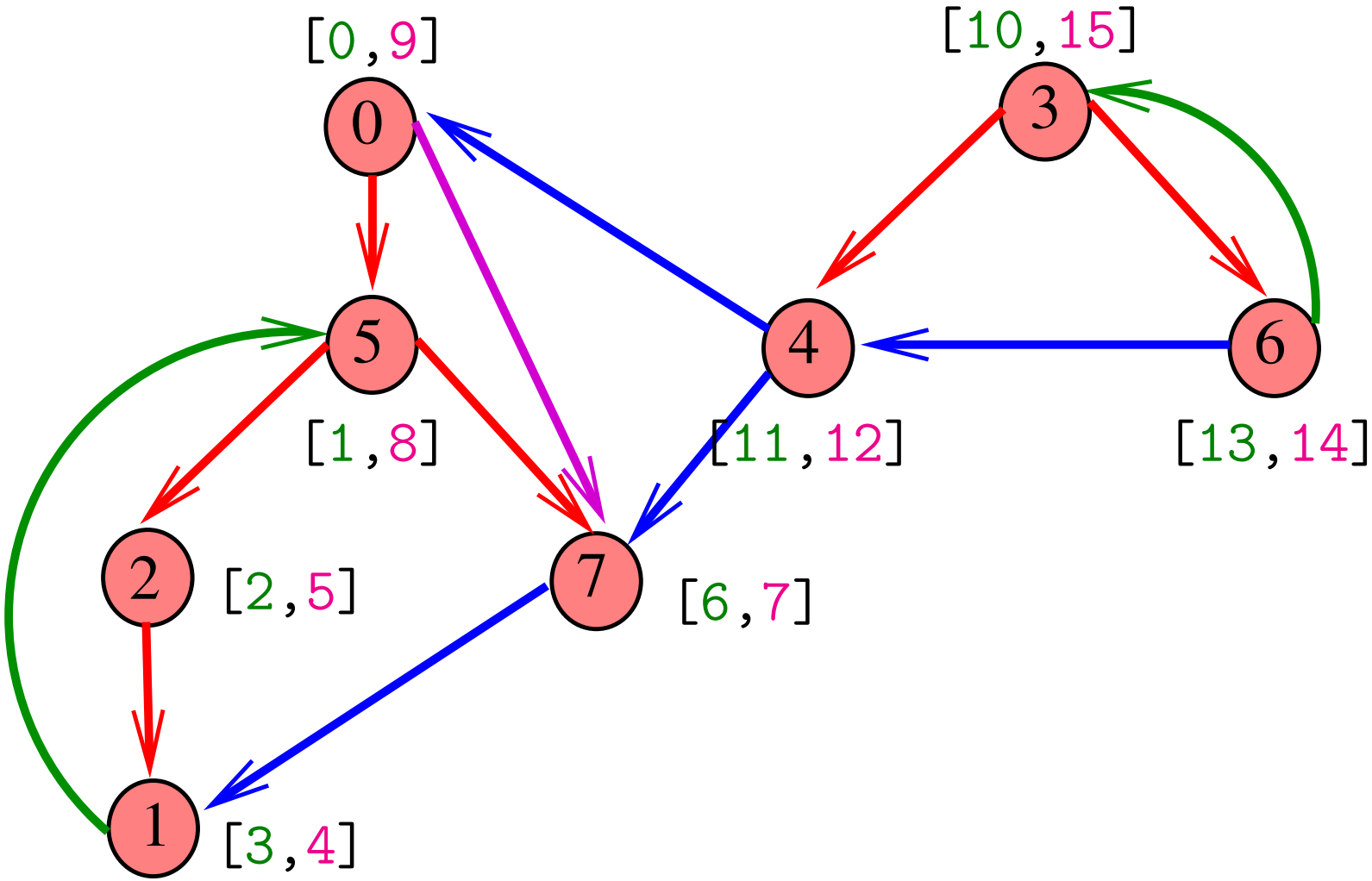
$$d[w] < d[v] < f[v] < f[w].$$



Arcos cruzados

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

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

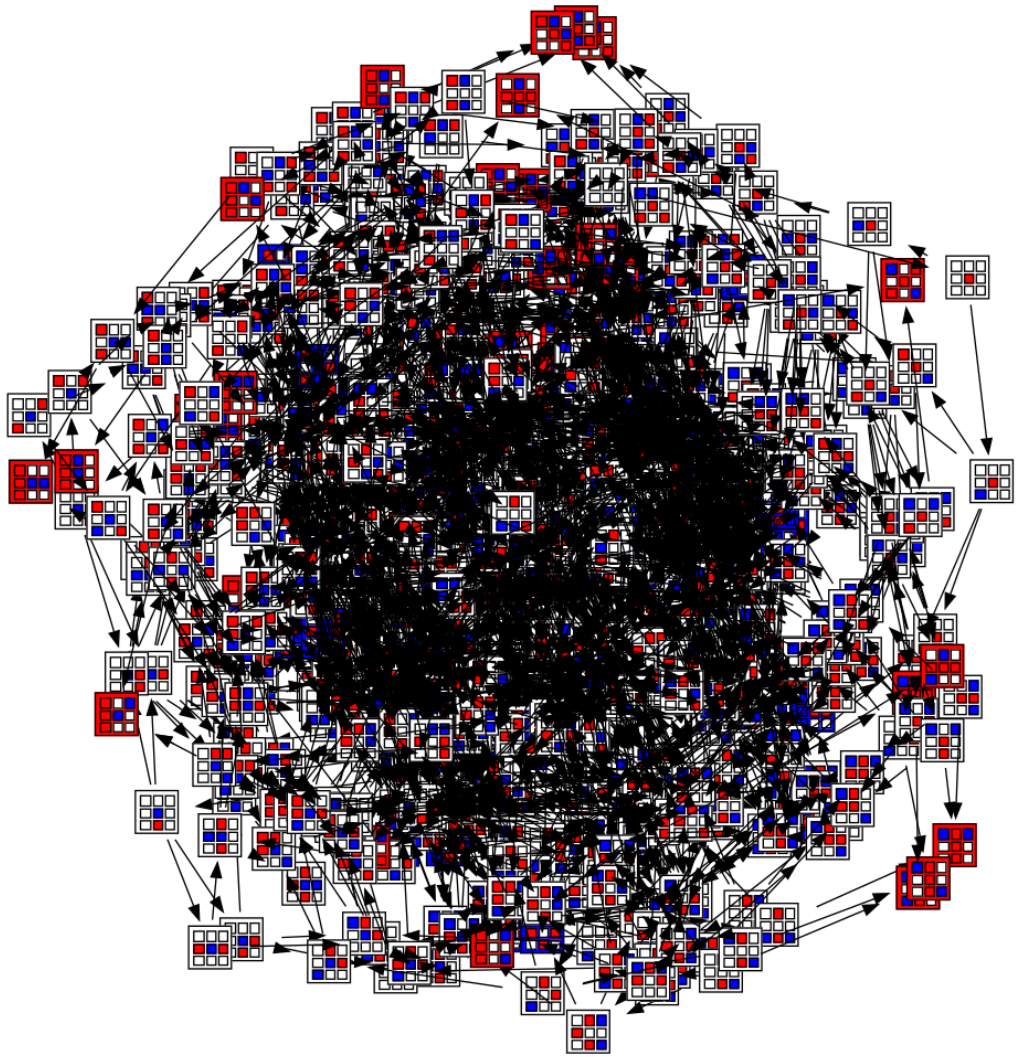


Conclusões

v-w é:

- **arco de arborescência** se e somente se
  $d[v] < d[w] < f[w] < f[v]$ e $edgeTo[w] = v$;
- **arco descendente** se e somente se
  $d[v] < d[w] < f[w] < f[v]$ e $edgeTo[w] \neq v$;
- **arco de retorno** se e somente se
  $d[w] < d[v] < f[v] < f[w]$;
- **arco cruzado** se e somente se
  $d[w] < f[w] < d[v] < f[v]$;

Ciclos em digrafos

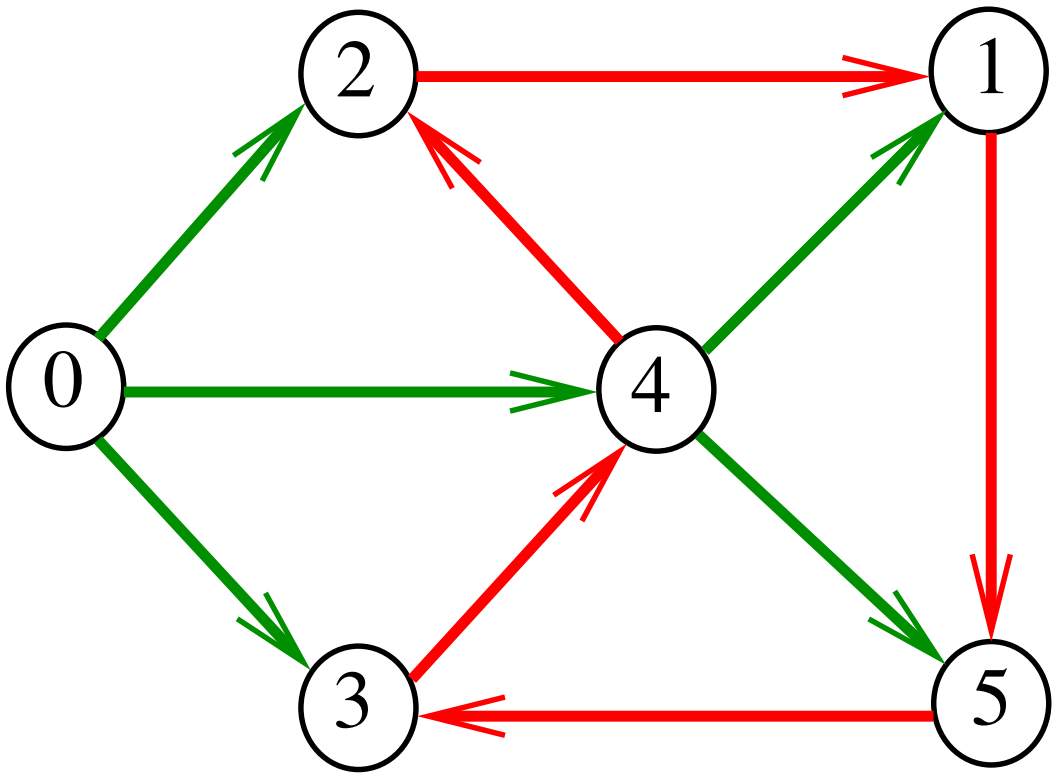


Fonte: laying out a large graph with graphviz

Ciclos

Um **ciclo** num digrafo é qualquer sequência da forma v_0–v_1–v_2– $\cdots$ –v_{k-1}–v_p, onde v_{k-1}–v_k é um arco para $k = 1, \ldots, p$ e $v_0 = v_p$.

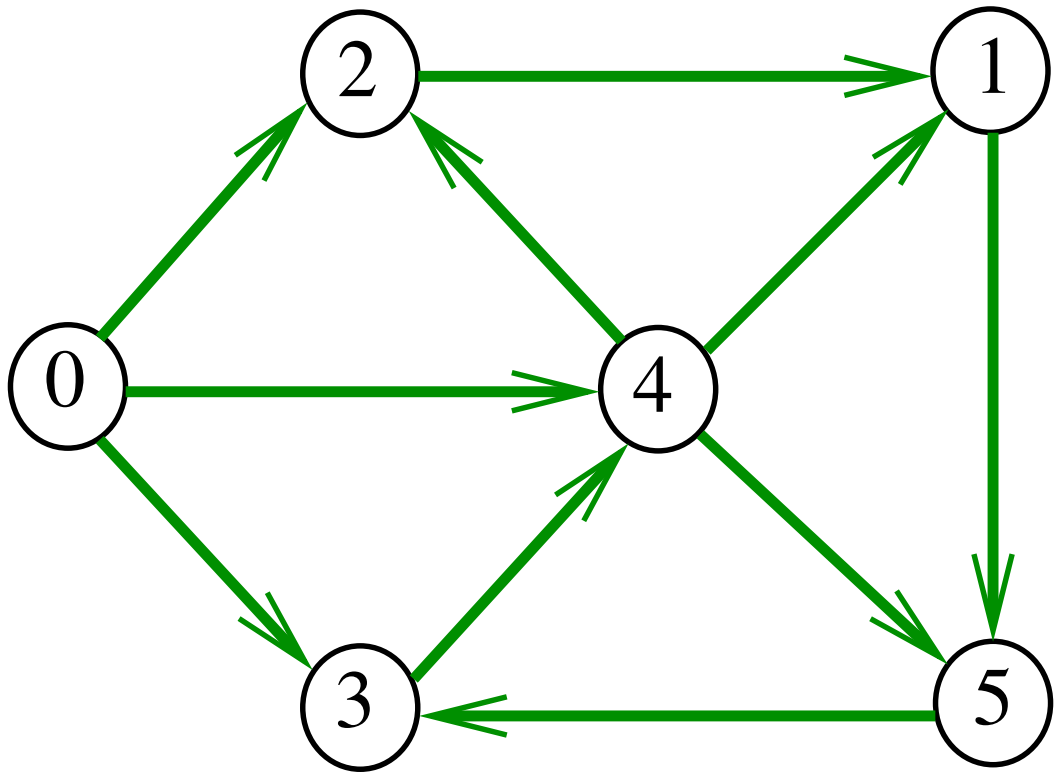
Exemplo: 2–1–5–3–4–2 é um ciclo



Procurando um ciclo

Problema: decidir se um dado digrafo possui um ciclo.

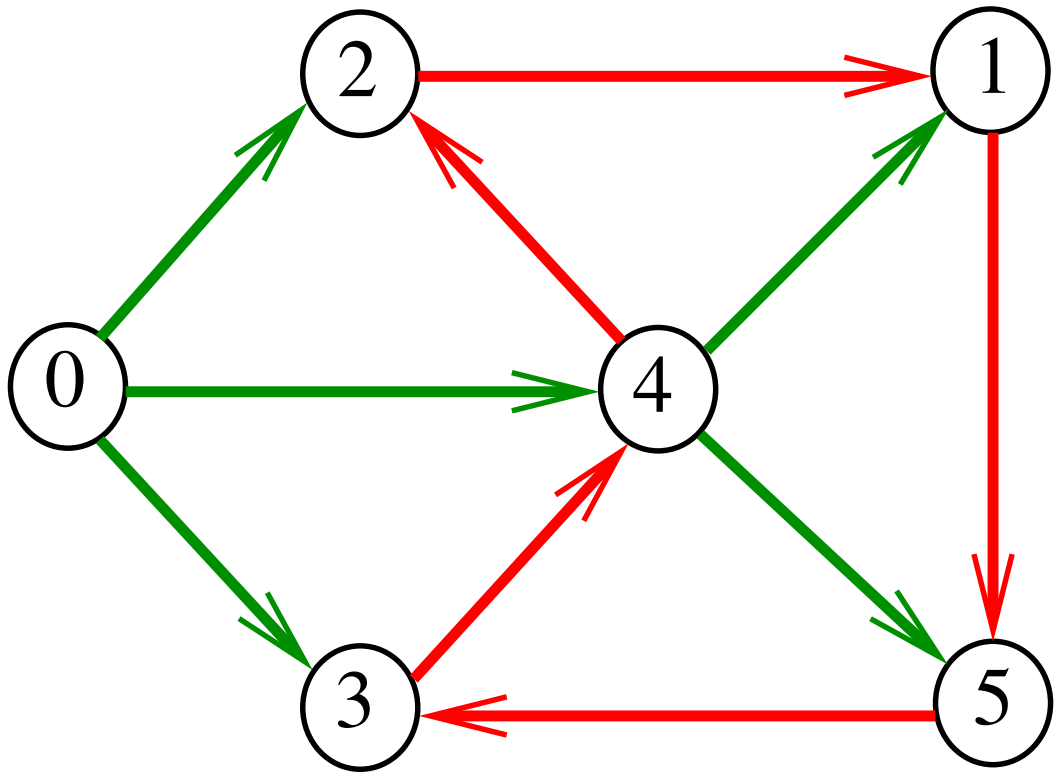
Exemplo: para o digrafo a seguir, a resposta é **SIM**



Procurando um ciclo

Problema: decidir se um dado digrafo possui um ciclo.

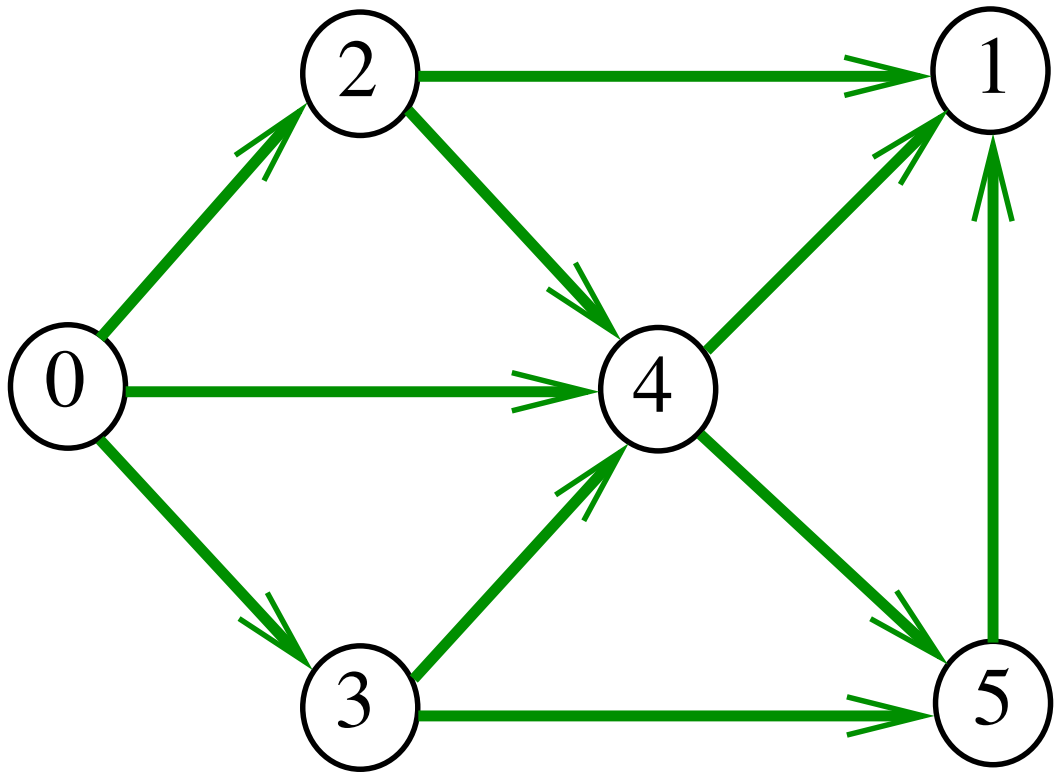
Exemplo: para o digrafo a seguir, a resposta é **SIM**



Procurando um ciclo

Problema: decidir se um dado digrafo possui um ciclo.

Exemplo: para o digrafo a seguir, a resposta é **NÃO**



`DirectedCycle` café com leite

Recebe um digrafo `G` e decide se existe um ciclo em `G`.

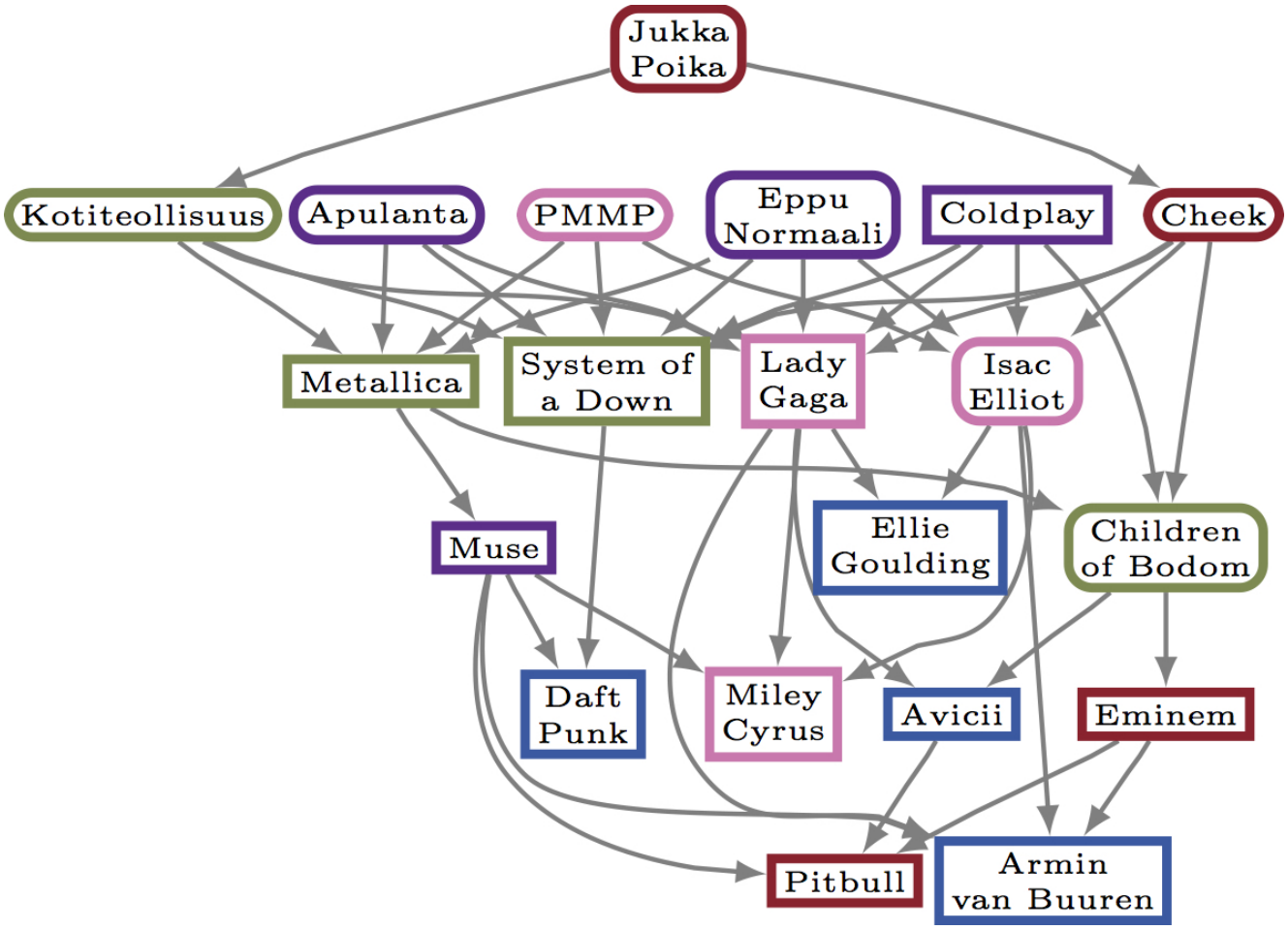
Para cada arco `u-v`, podemos fazer

```
dfsPaths dfs = DFSpaths(G, v);
```

e verificar se `hasPath(dfs, u)`.

O consumo de tempo para
vetor de listas de adjacência é $O(E\ (V + E))$.

Digrafos acíclicos (DAGs)

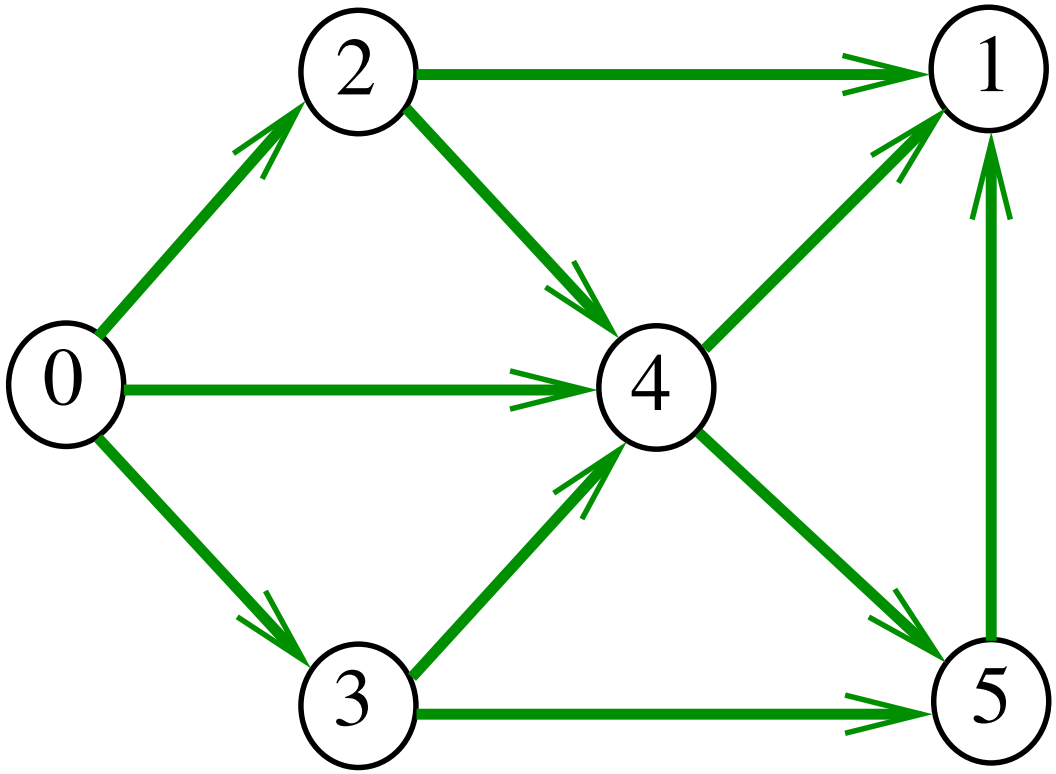


Fonte: Comparing directed acyclic graphs

DAGs

Um digrafo é **acíclico** se não tem ciclos.
Digrafos acíclicos também são conhecidos como DAGs (= *directed acyclic graphs*).

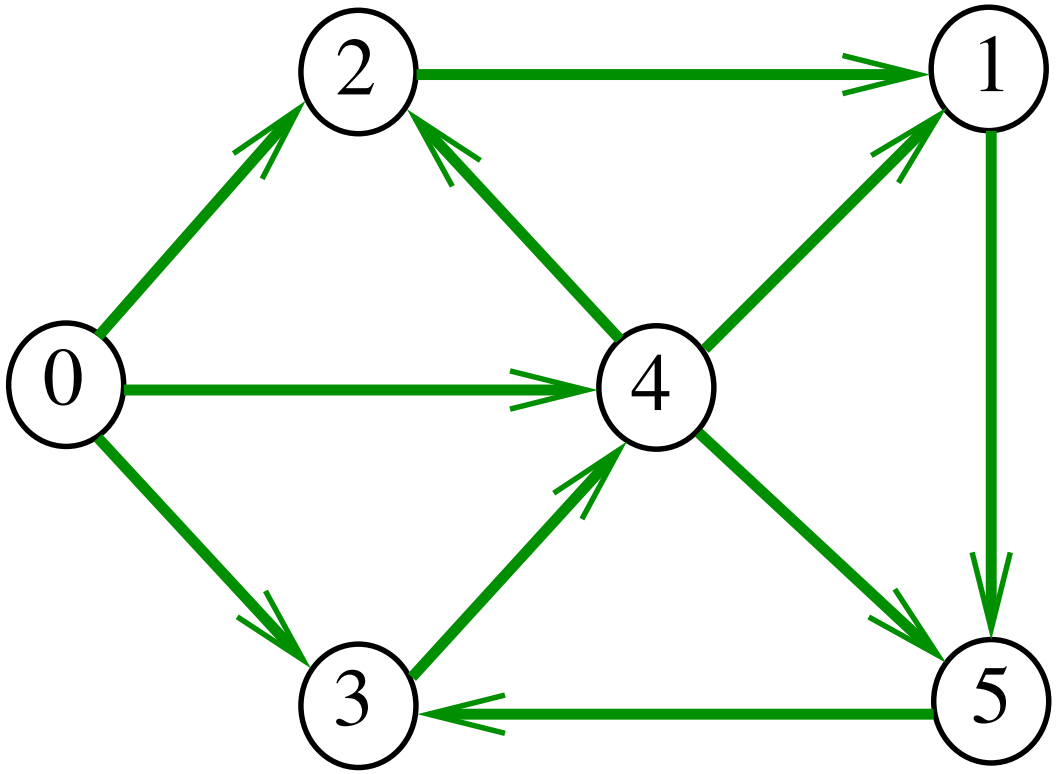
Exemplo: um digrafo acíclico



DAGs

Um digrafo é **acíclico** se não tem ciclos.
Digrafos acíclicos também são conhecidos como DAGs (= *directed acyclic graphs*).

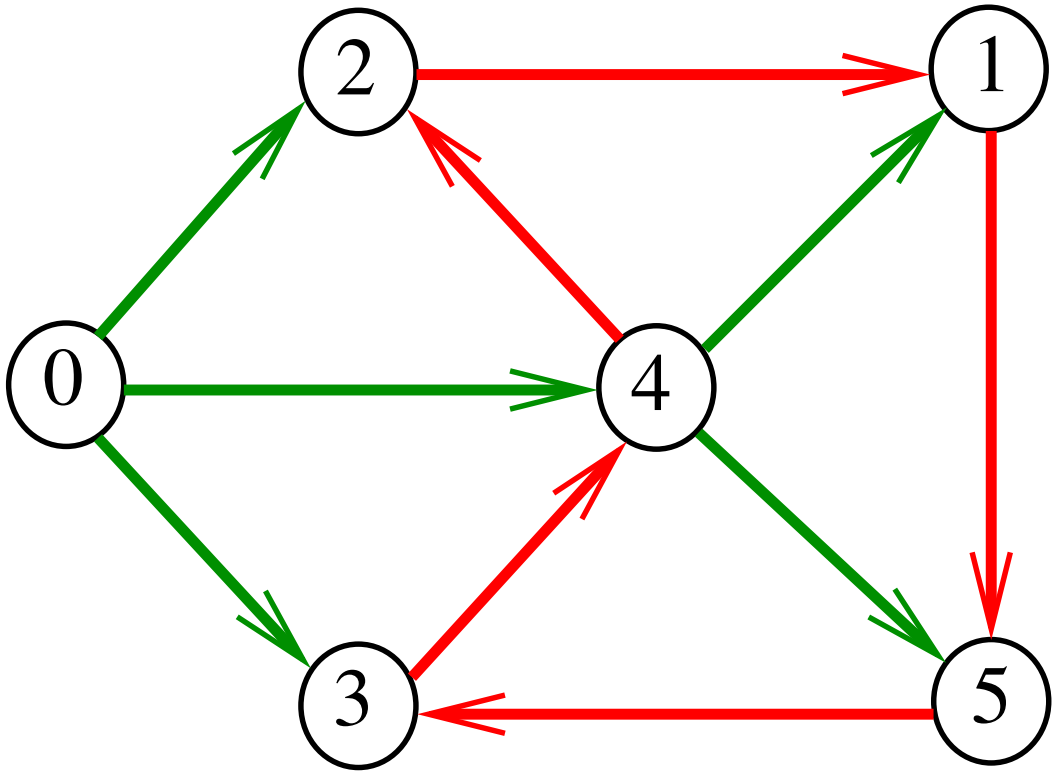
Exemplo: um digrafo que **não** é acíclico



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 sequência 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]`

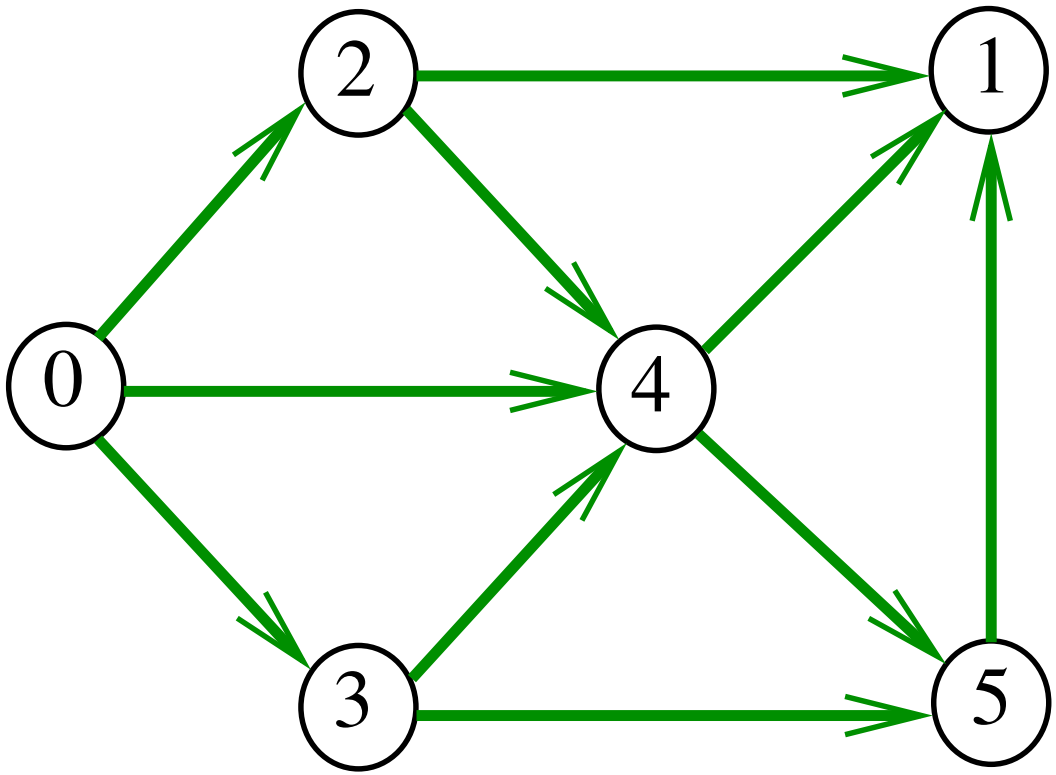
dos seus vértices tal que todo arco tem a forma

`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.

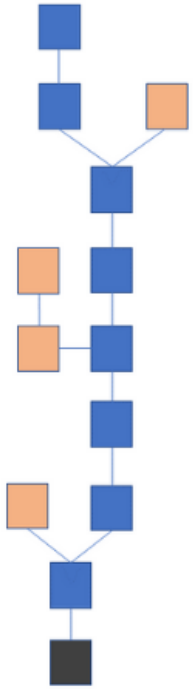
É menos evidente que **todo** DAG
admite uma ordenação topológica.

A prova desse fato é um algoritmo que
recebe um digrafo arbitrário e devolve

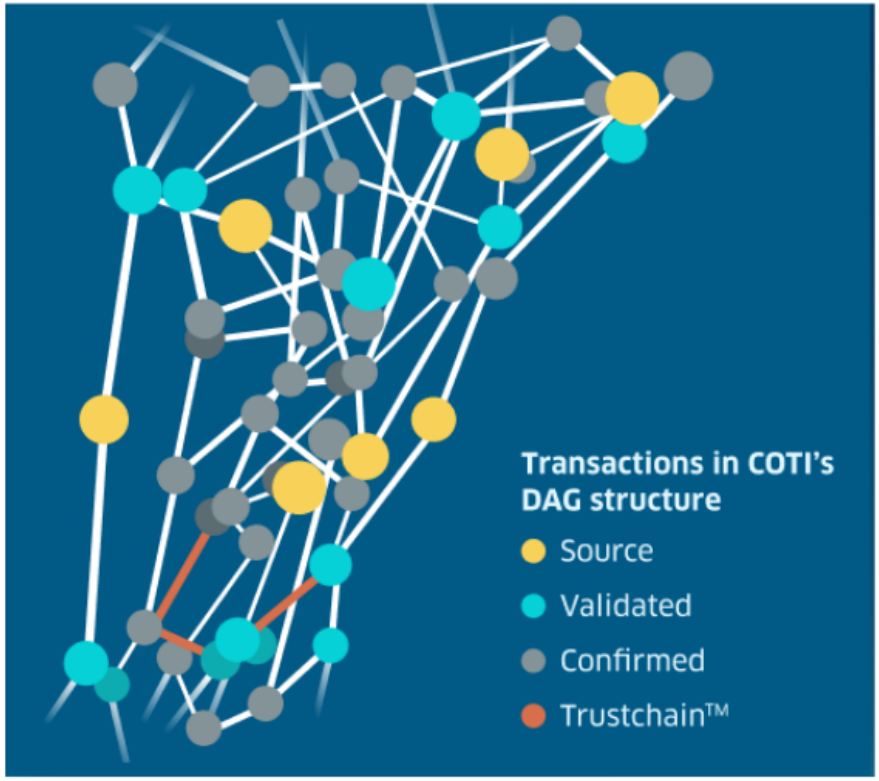
- um ciclo ou
- uma ordenação topológica do digrafo.

Algoritmos de ordenação topológica

Blockchain vs Directed Acyclic Graph (DAG)



Blockchain



COTI's DAG

Fonte: Our Exciting Partnership With COTI

Algoritmo de eliminação de fontes

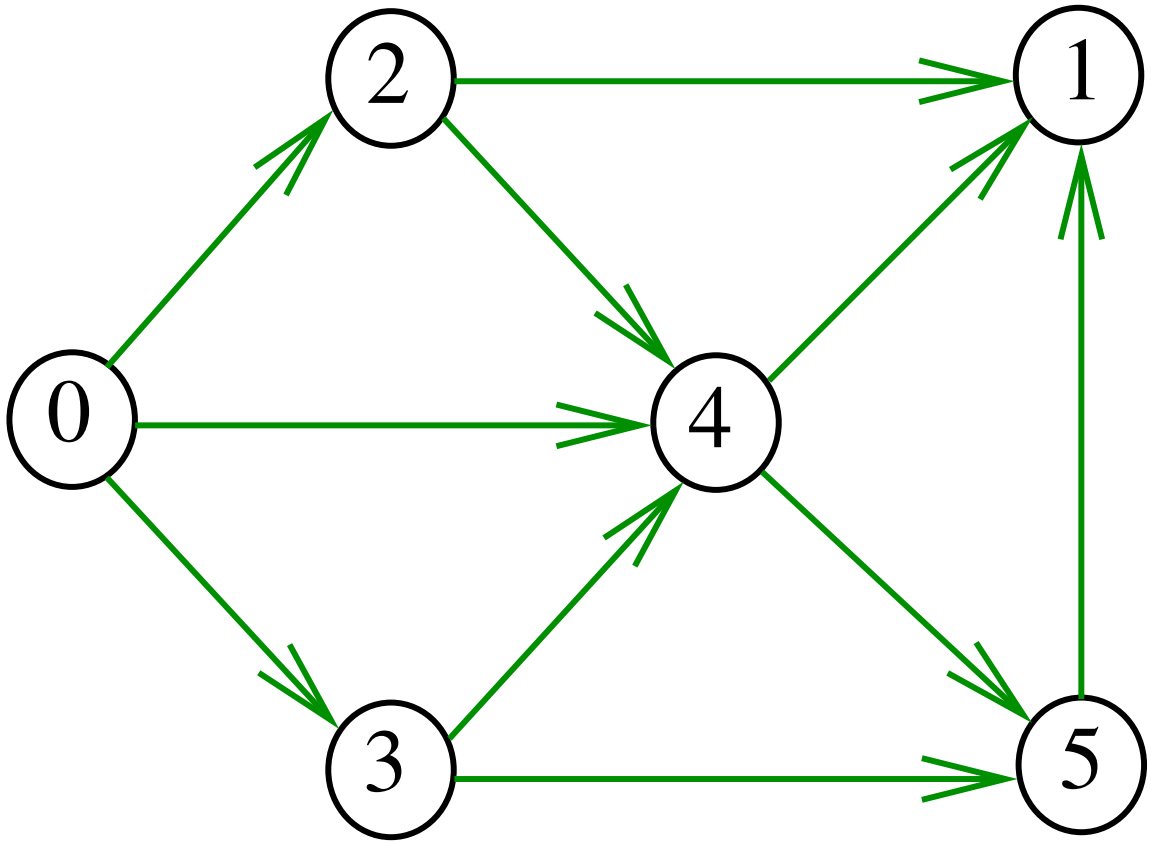
Armazena em `ts[0..i-1]` uma permutação de um subconjunto do conjunto de vértices de `G` e devolve o valor de `i`.

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);
```

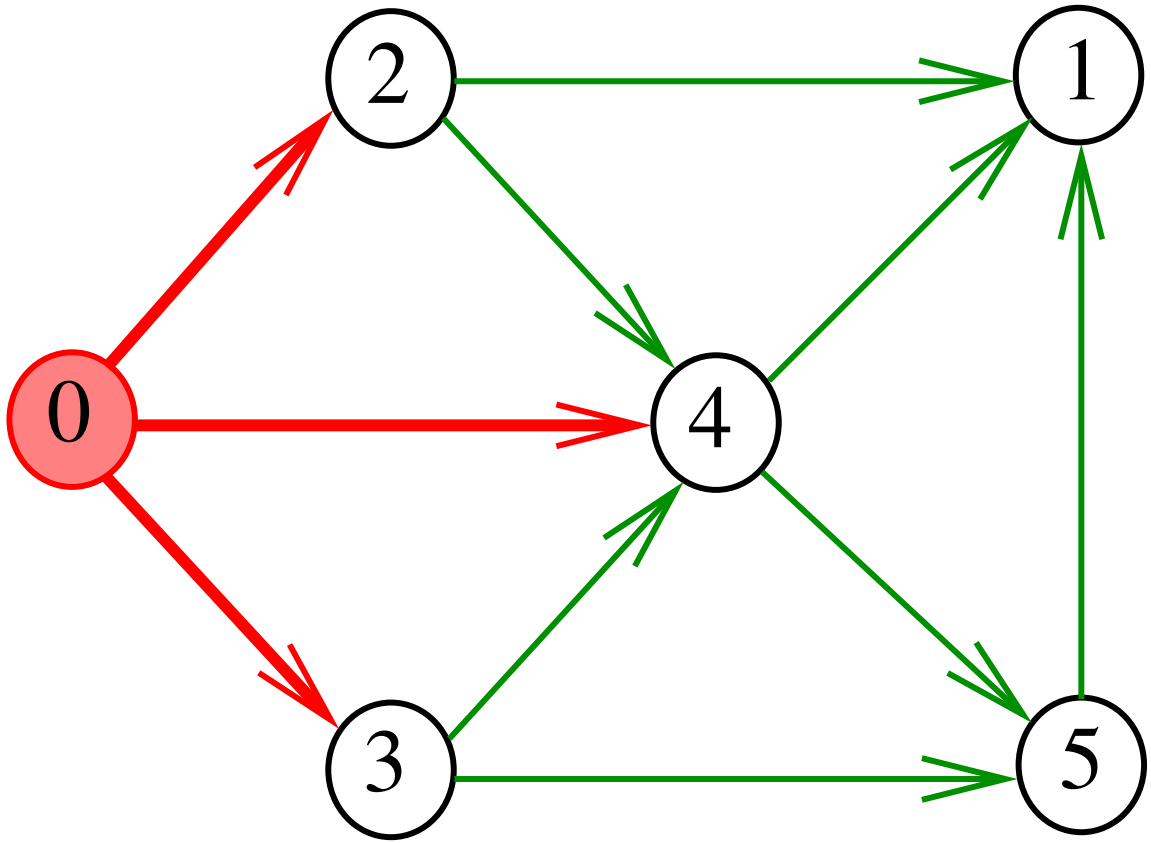
Exemplo

i	0	1	2	3	4	5
ts[i]						



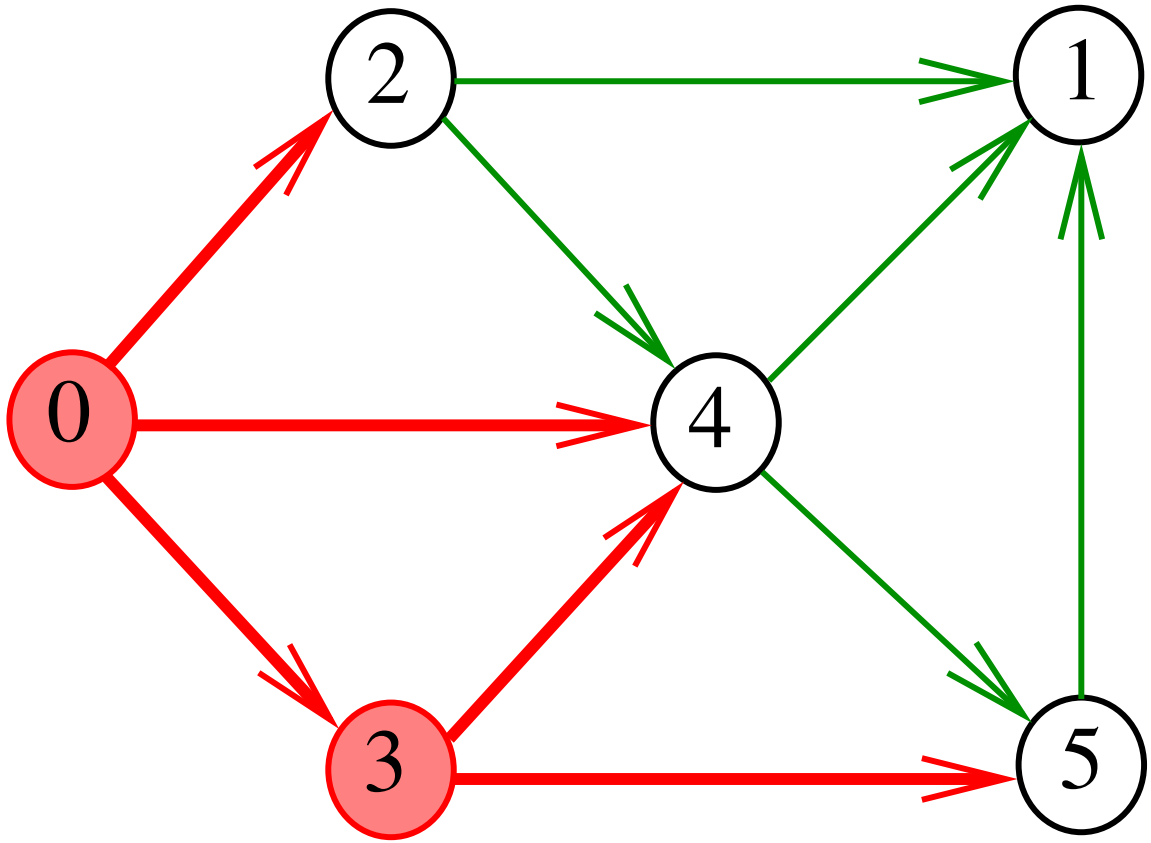
Exemplo

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



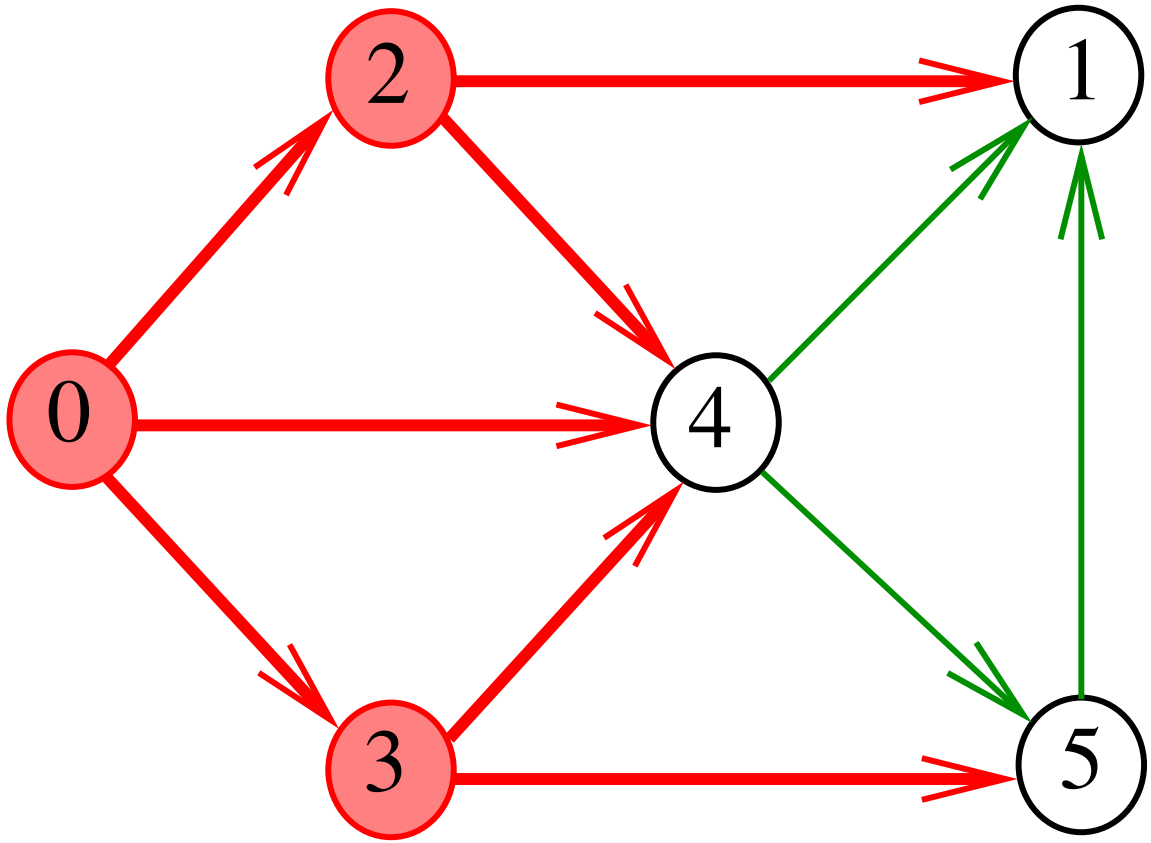
Exemplo

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



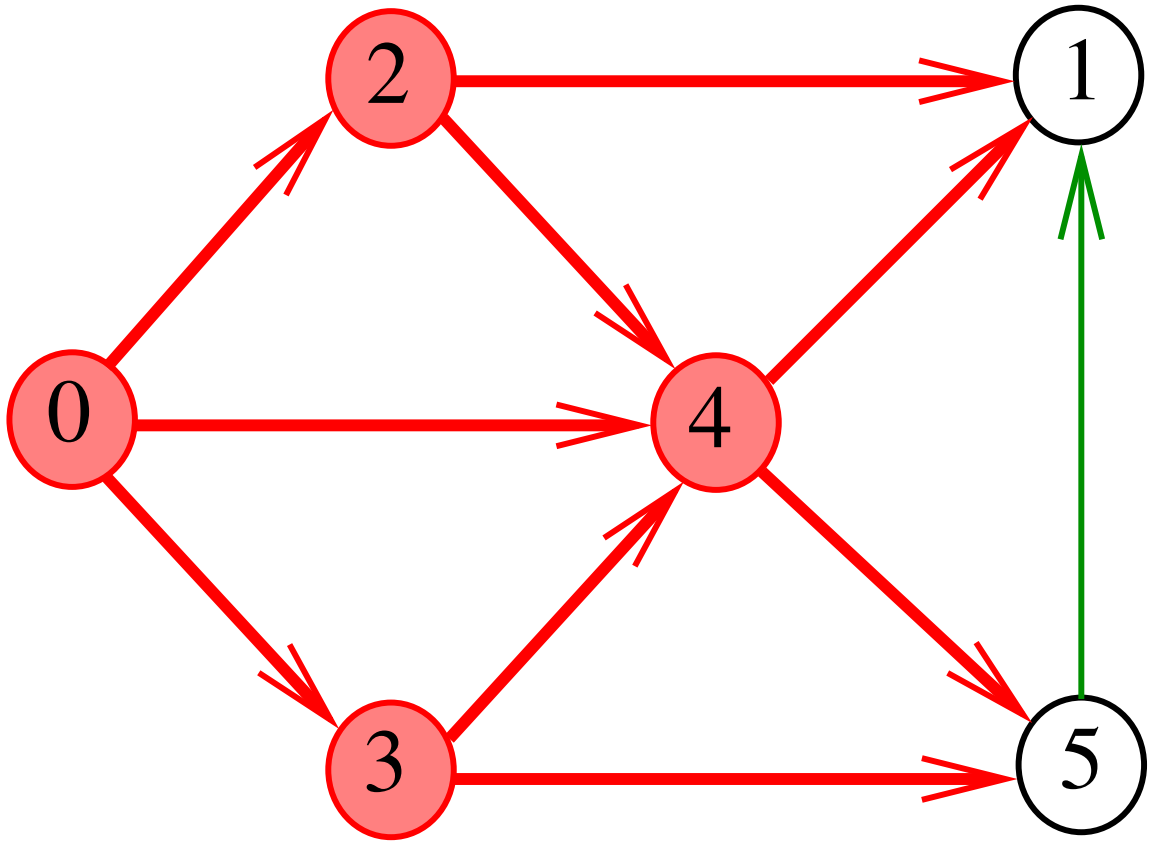
Exemplo

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



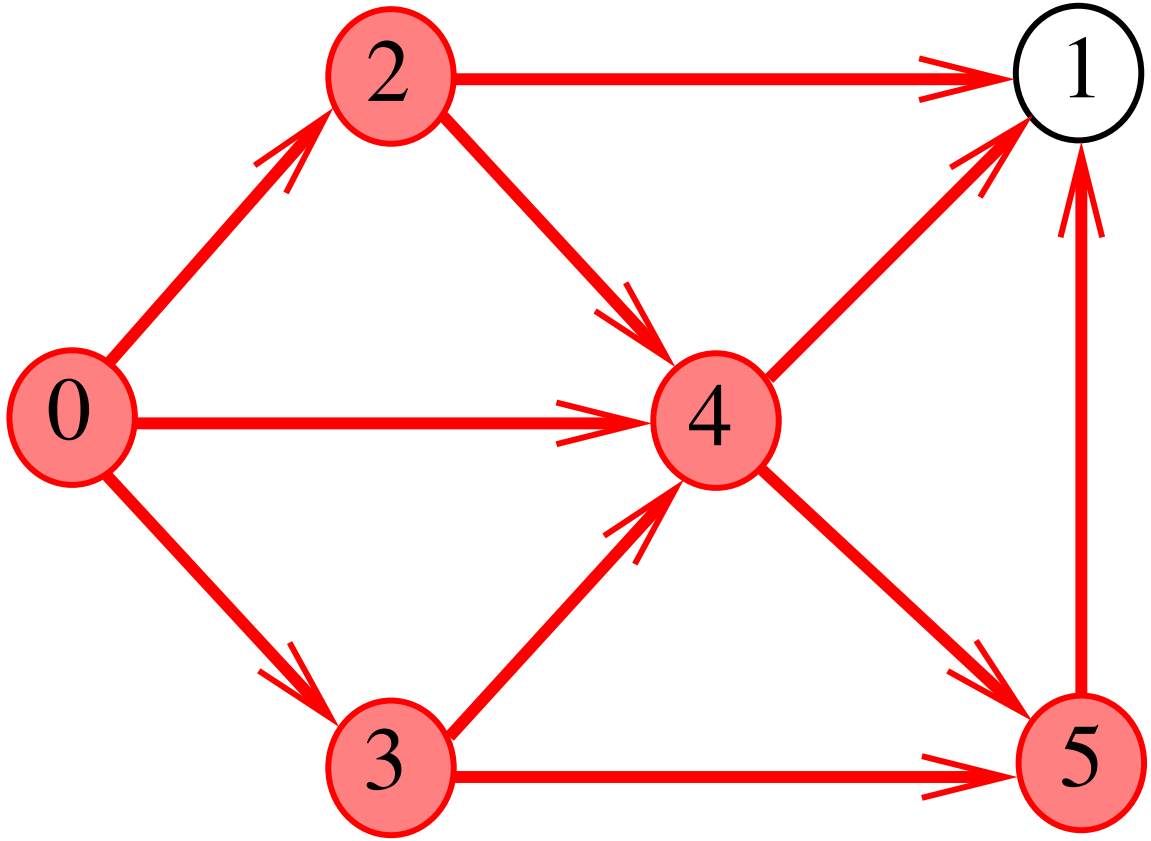
Exemplo

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



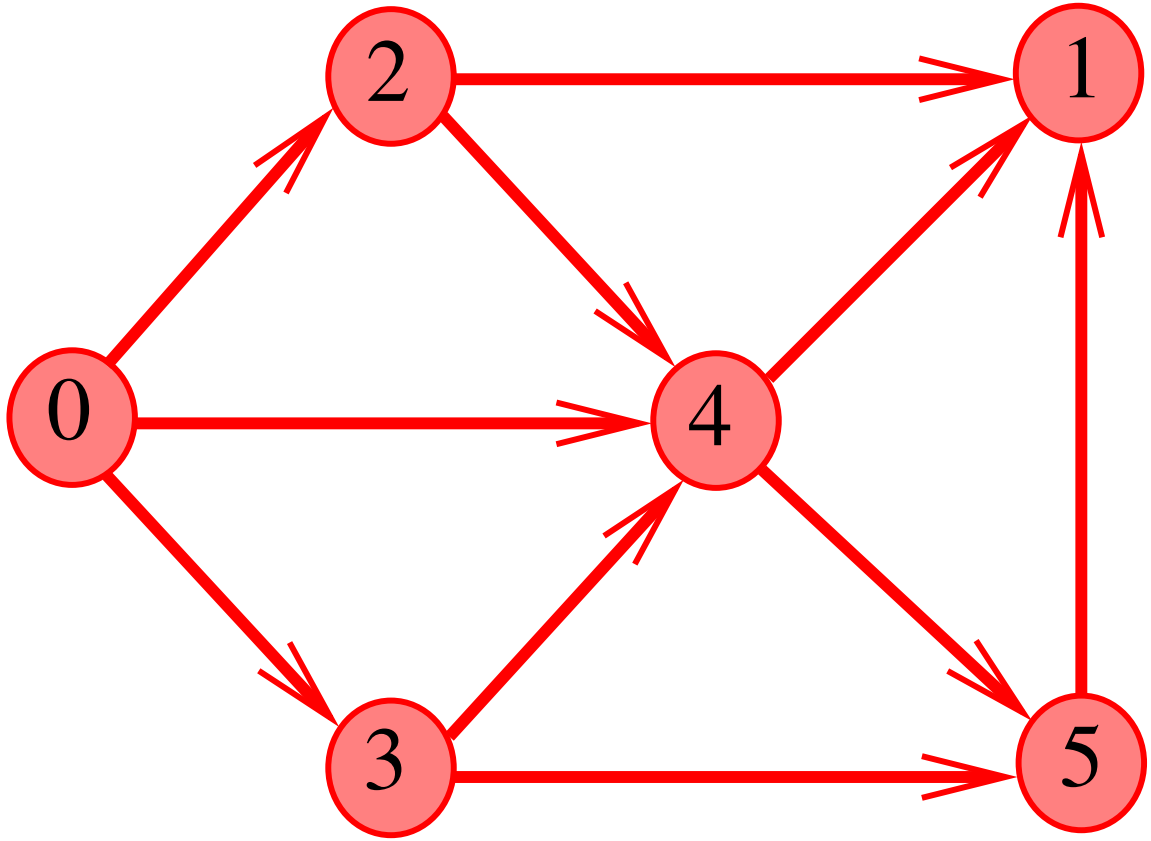
Exemplo

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



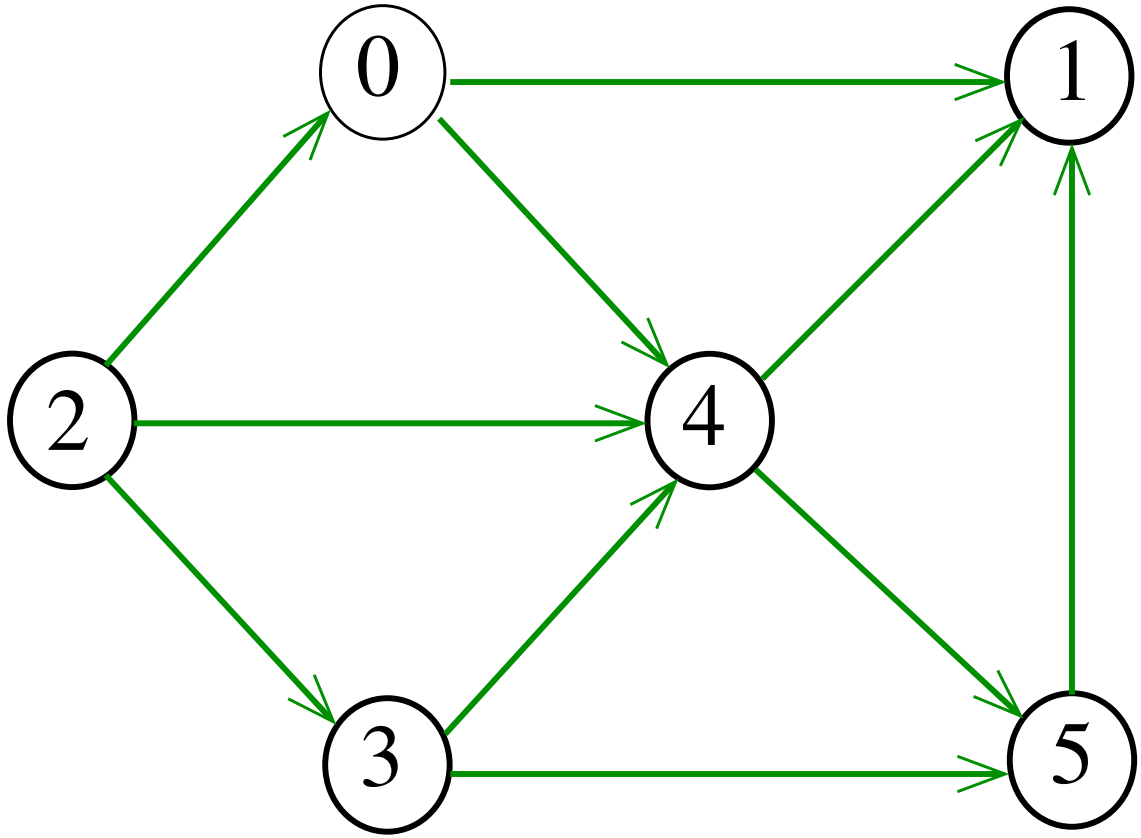
Consumo de tempo

O consumo de tempo desse algoritmo para vetor de listas de adjacência é $O(V + E)$.

O consumo de tempo desse algoritmo para matriz de adjacências é $O(V^2)$.

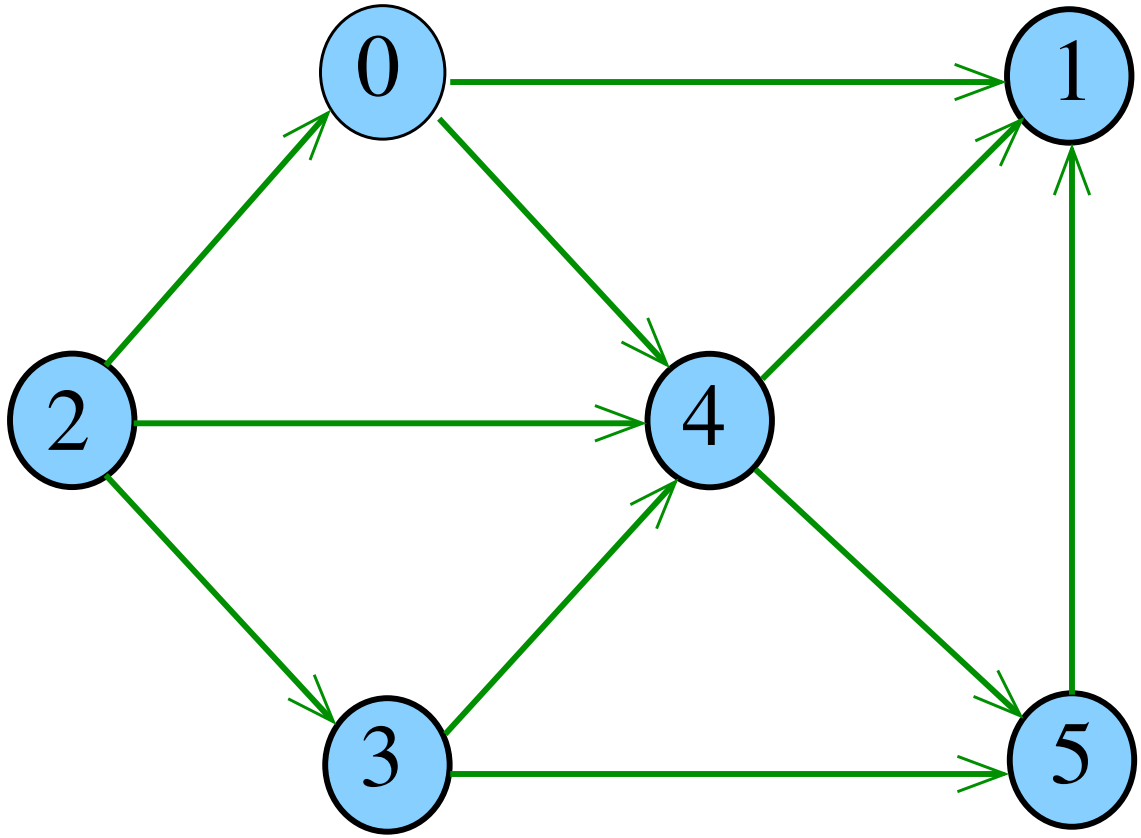
Algoritmo DFS

i	0	1	2	3	4	5
ts[i]						



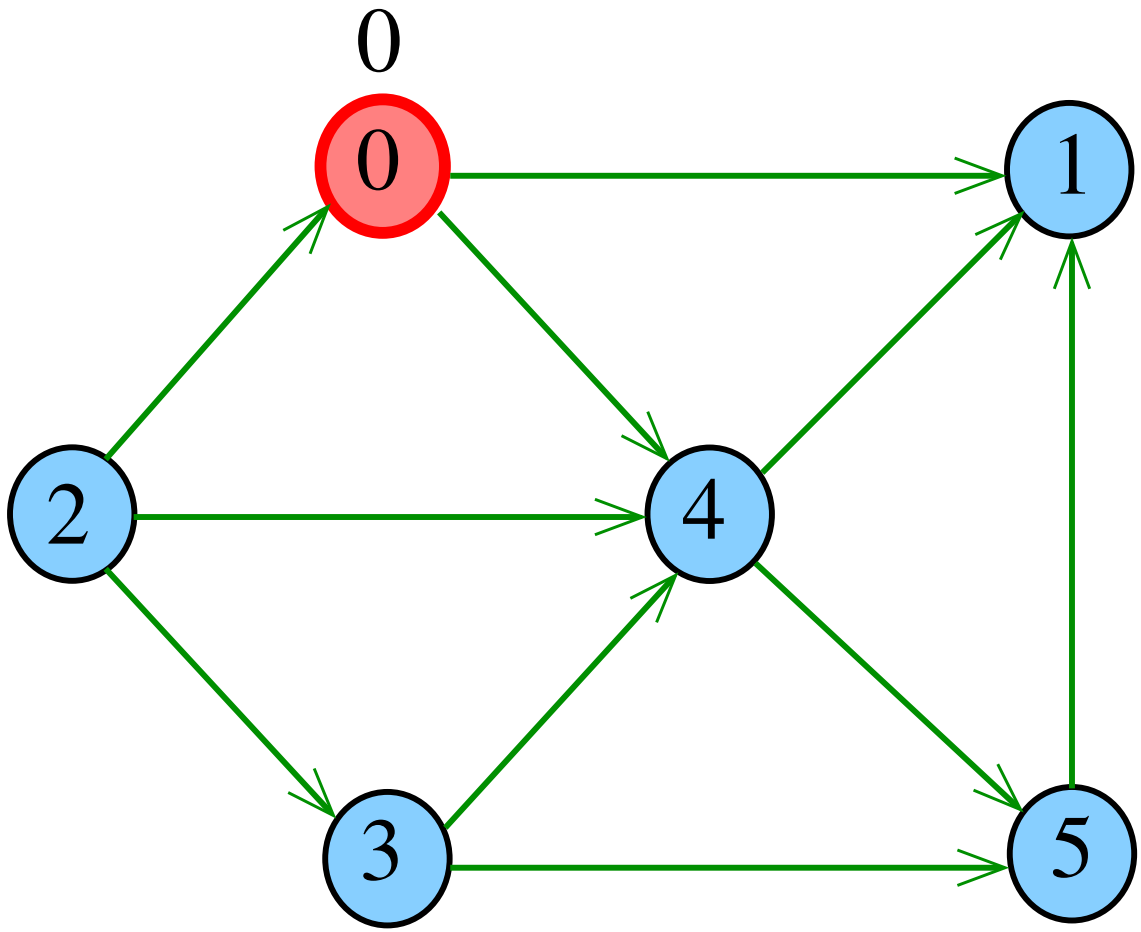
Algoritmo DFS

i	0	1	2	3	4	5
ts[i]						



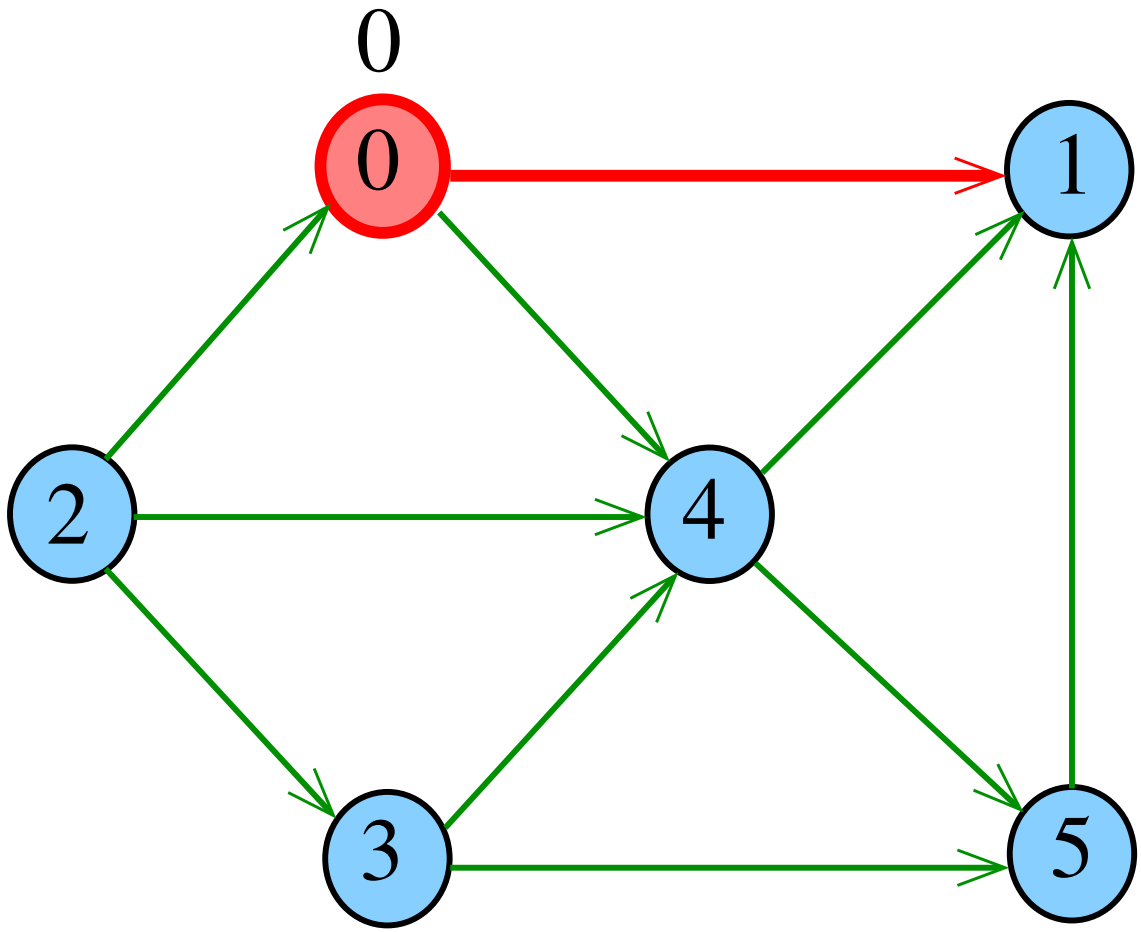
Algoritmo DFS

i	0	1	2	3	4	5
ts[i]						



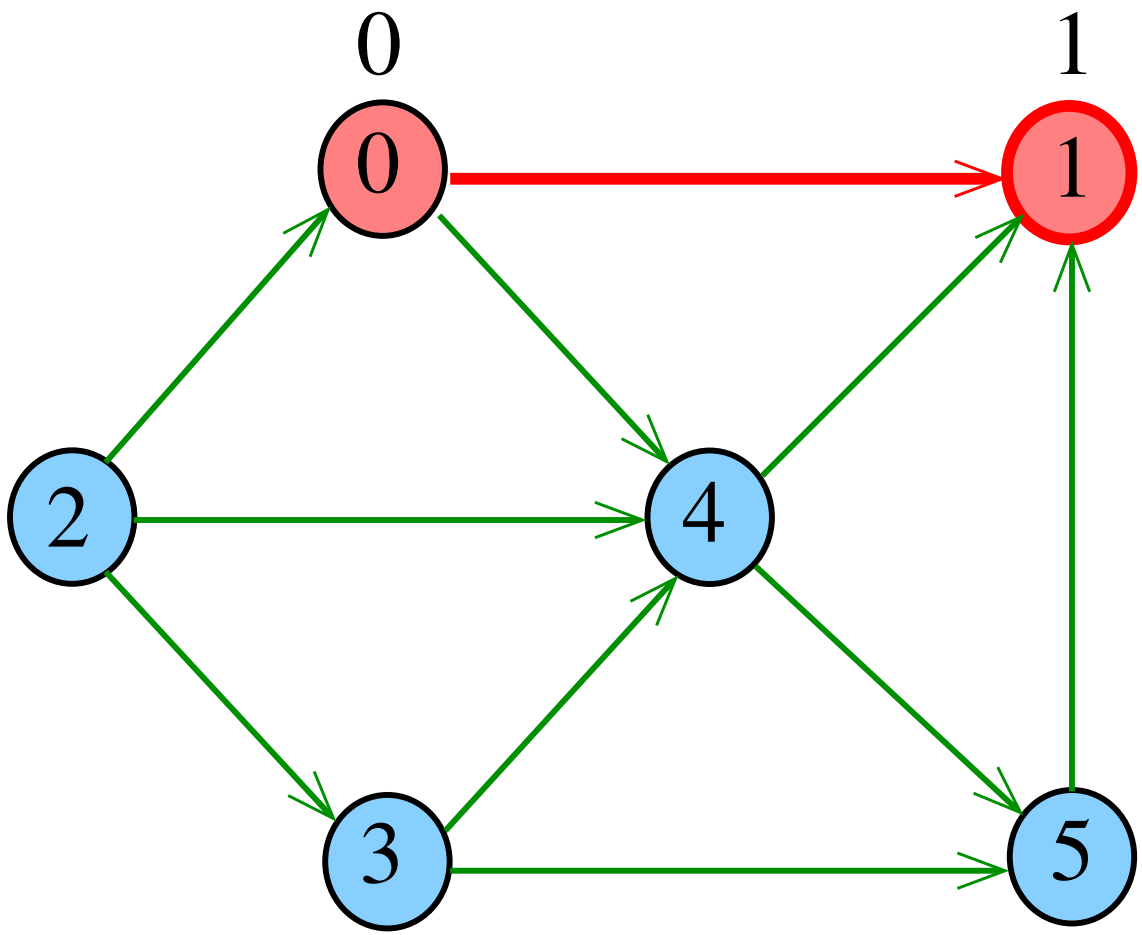
Algoritmo DFS

i	0	1	2	3	4	5
ts[i]						



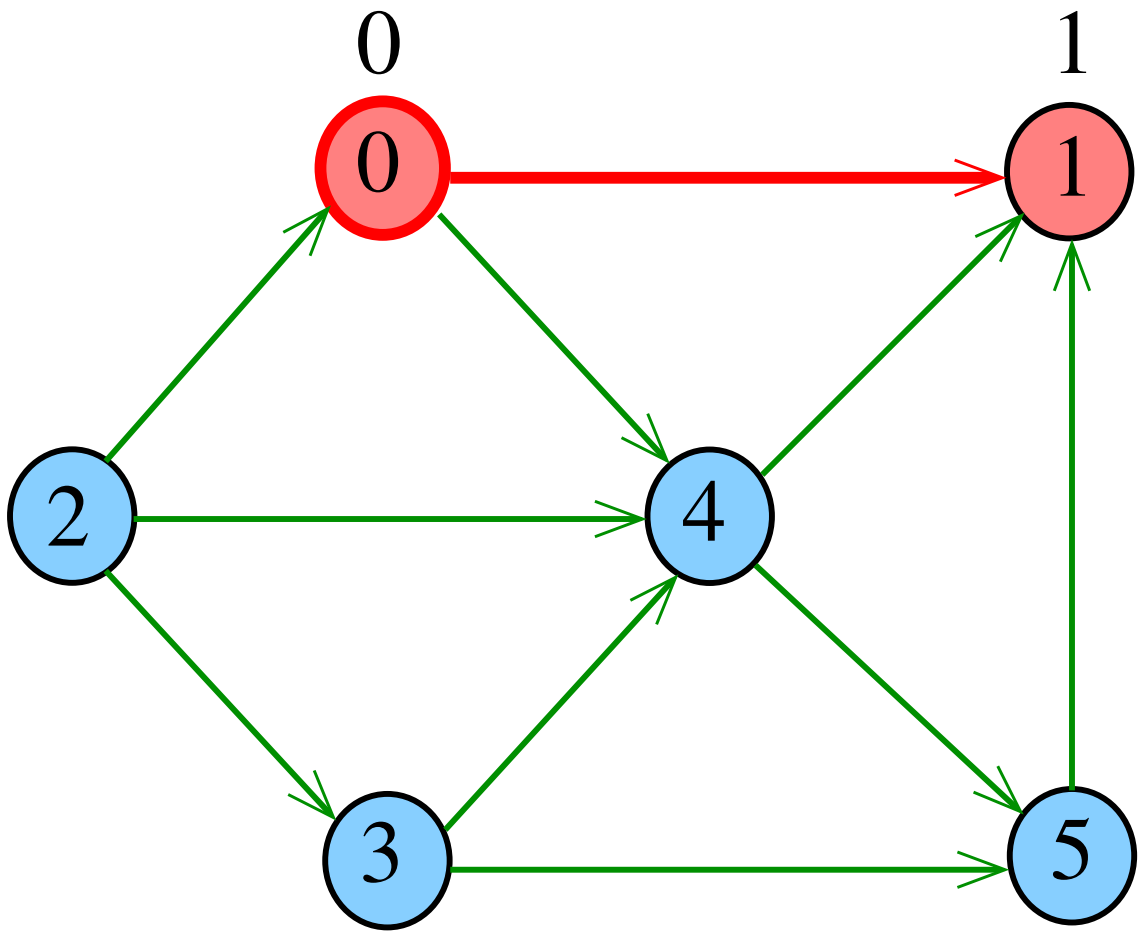
Algoritmo DFS

i	0	1	2	3	4	5
ts[i]						



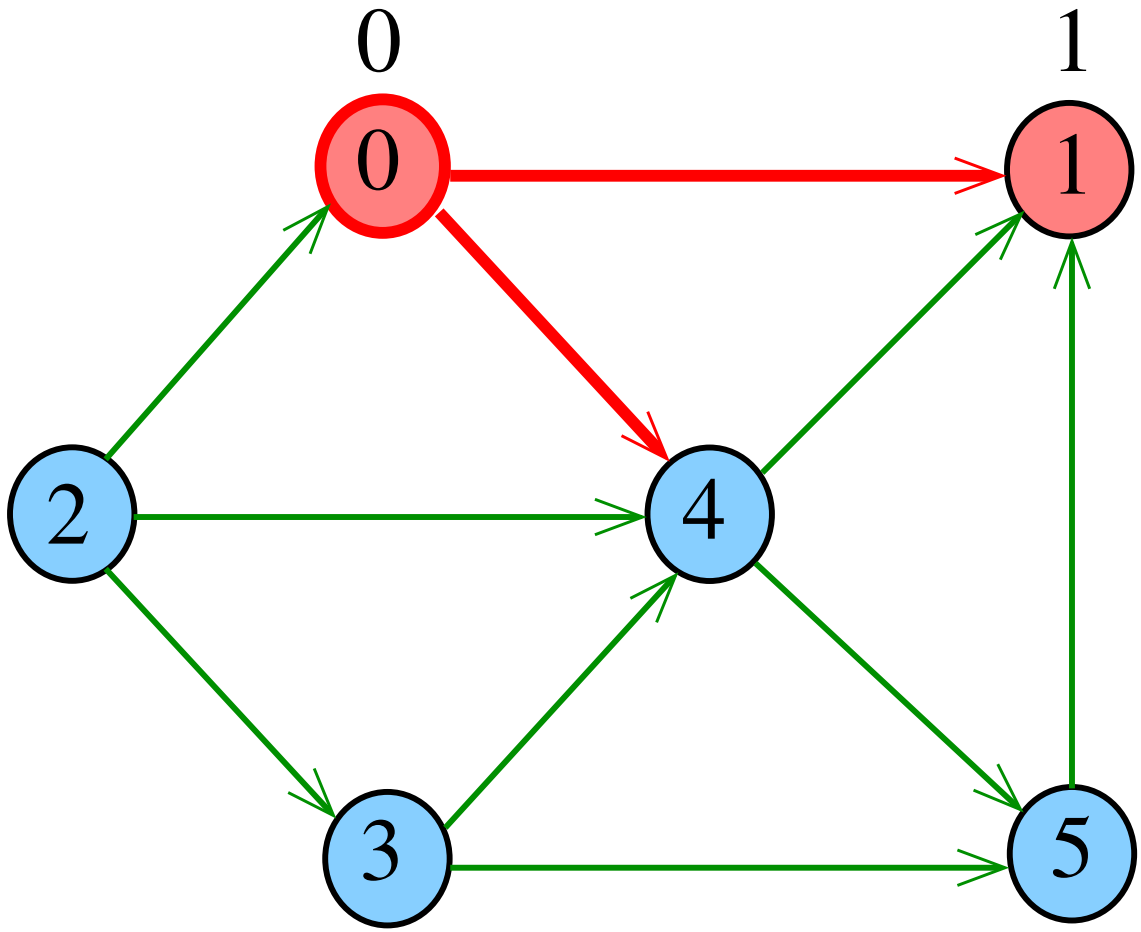
Algoritmo DFS

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



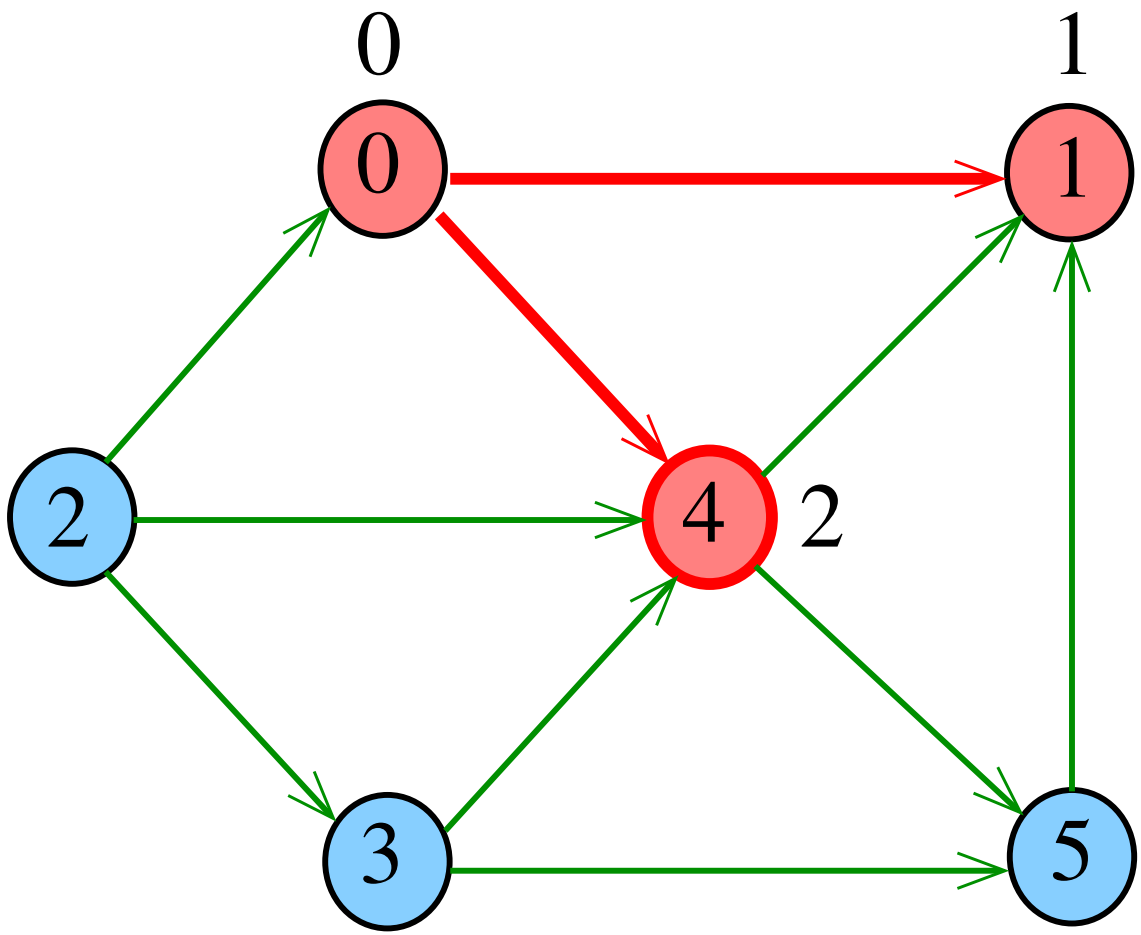
Algoritmo DFS

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



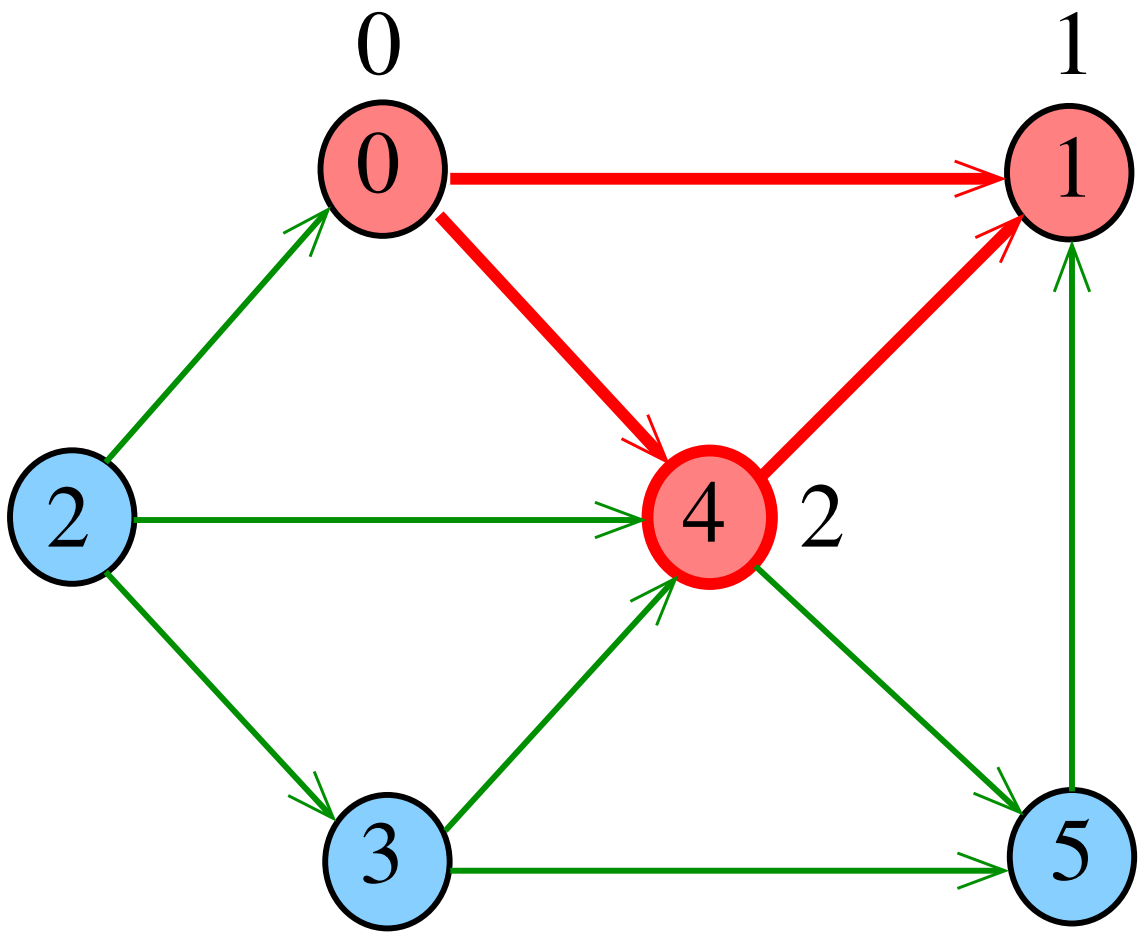
Algoritmo DFS

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



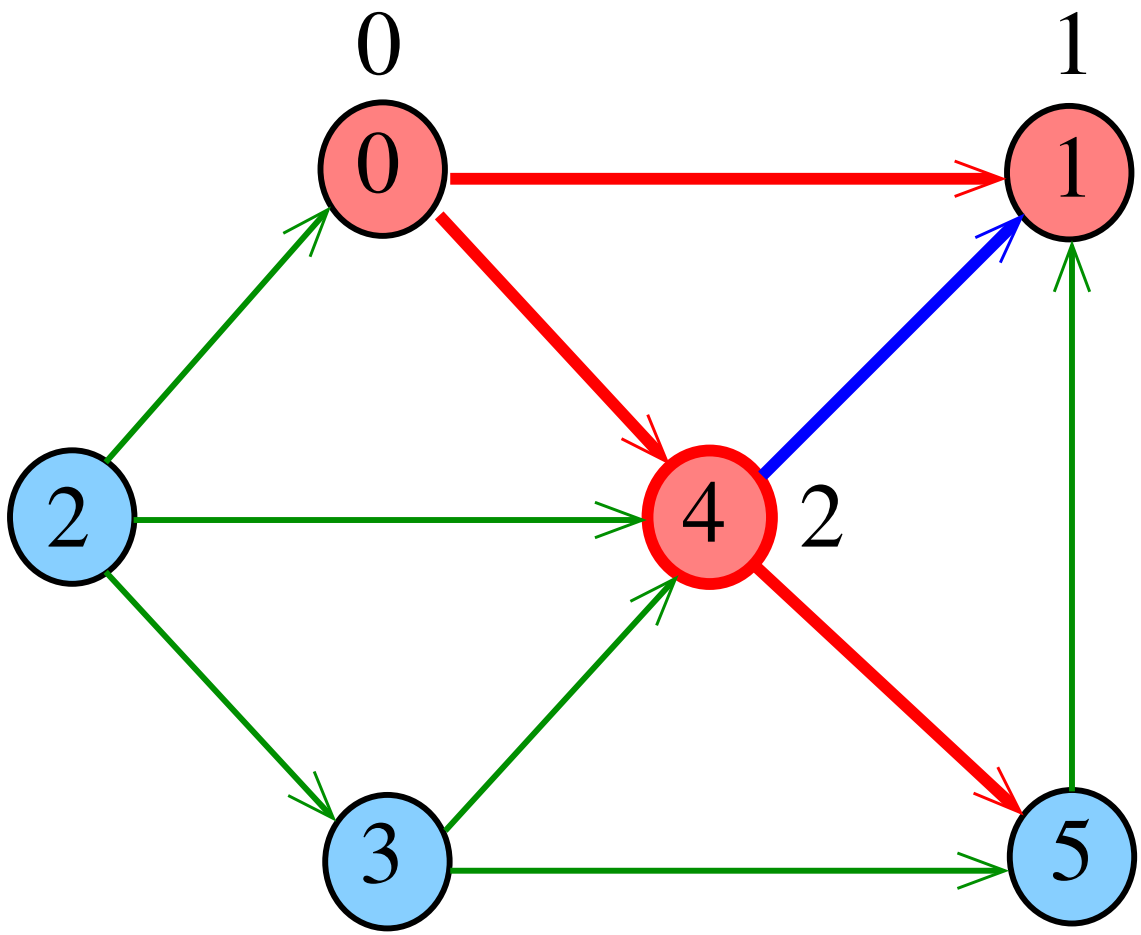
Algoritmo DFS

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



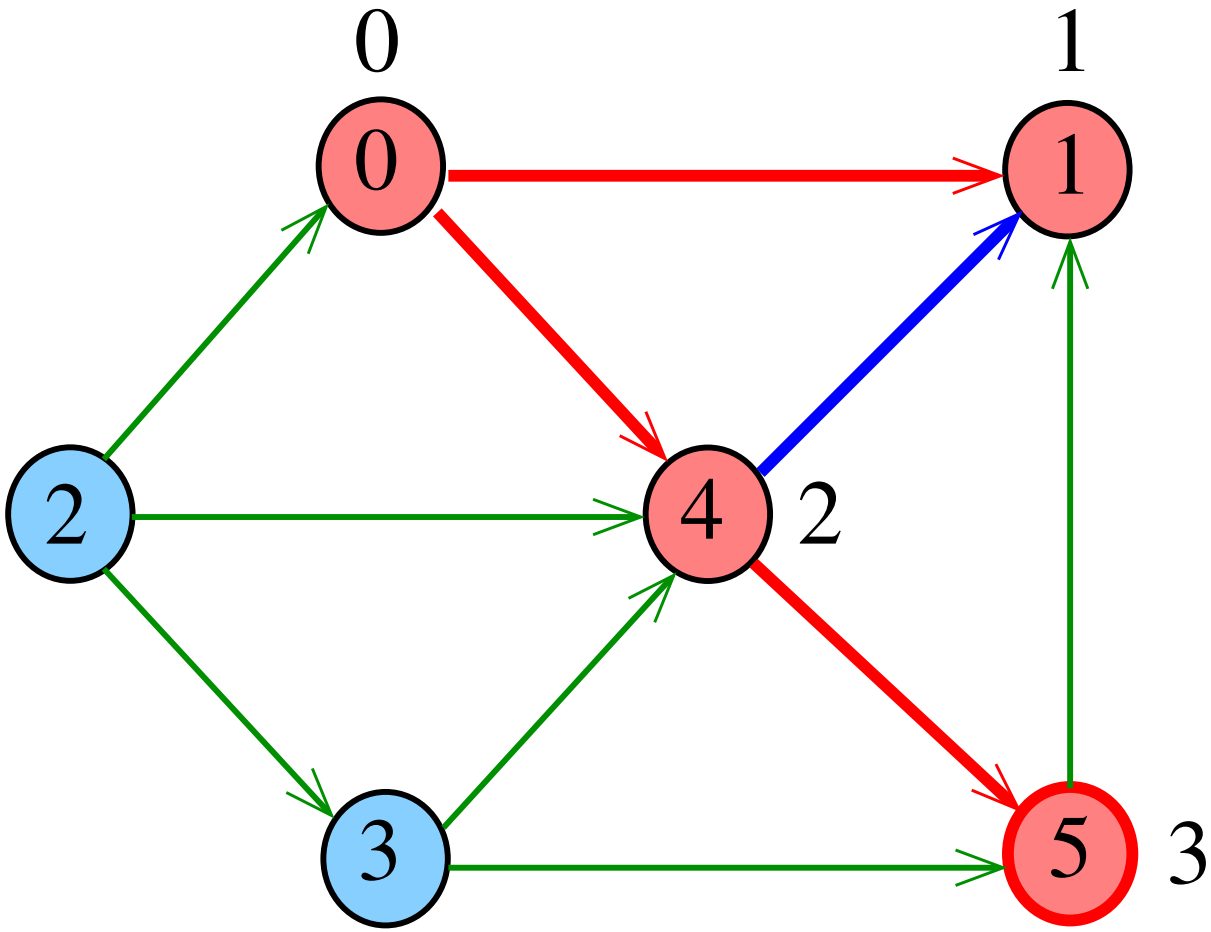
Algoritmo DFS

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



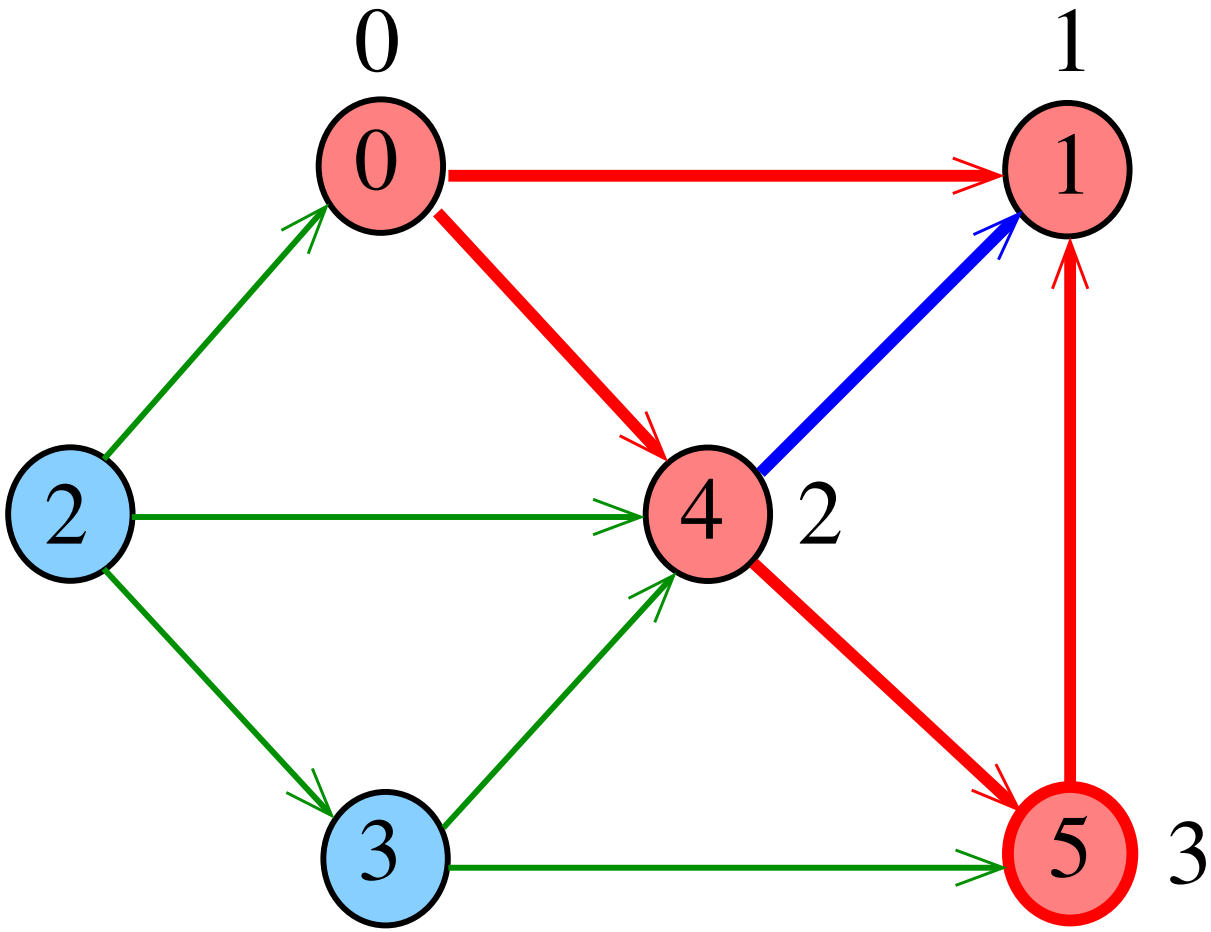
Algoritmo DFS

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



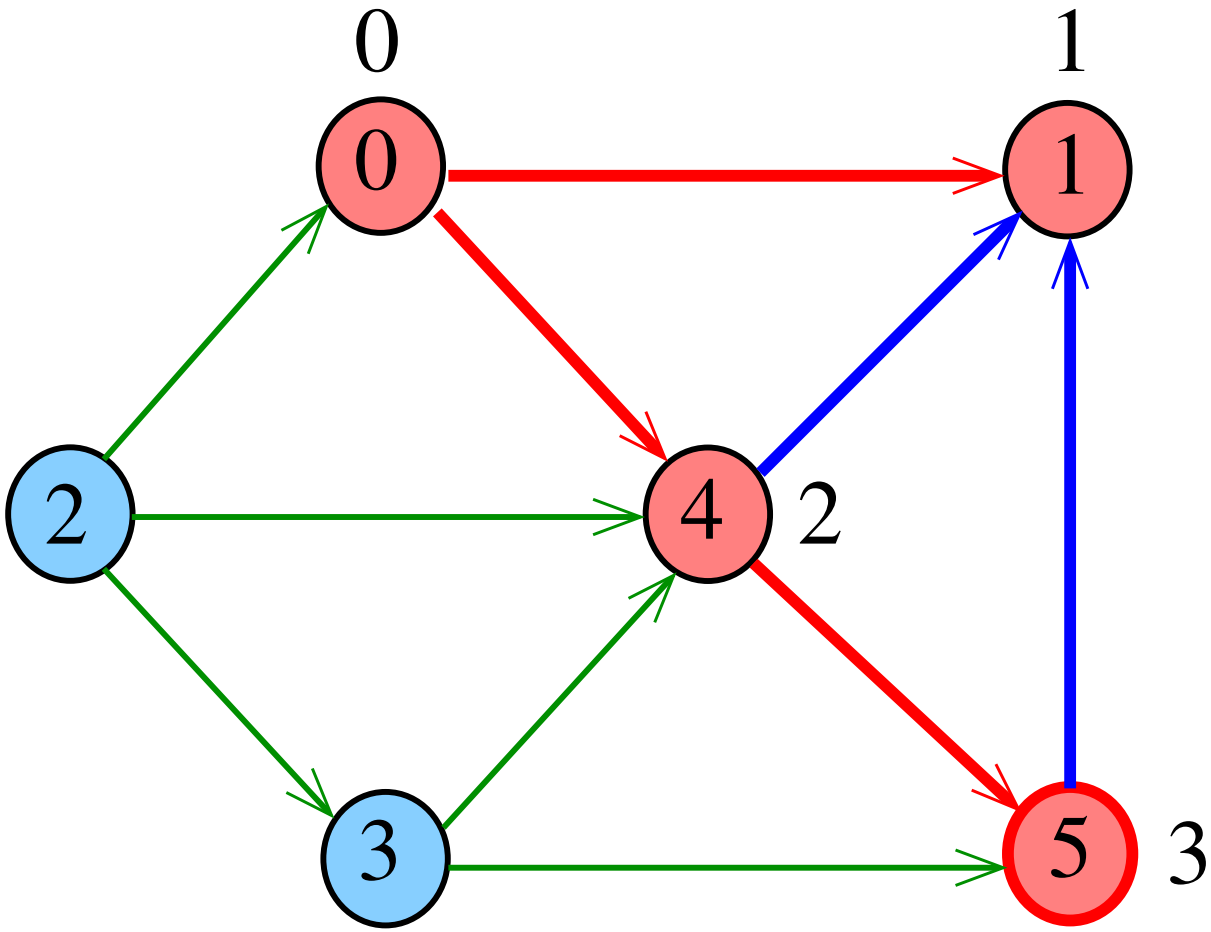
Algoritmo DFS

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



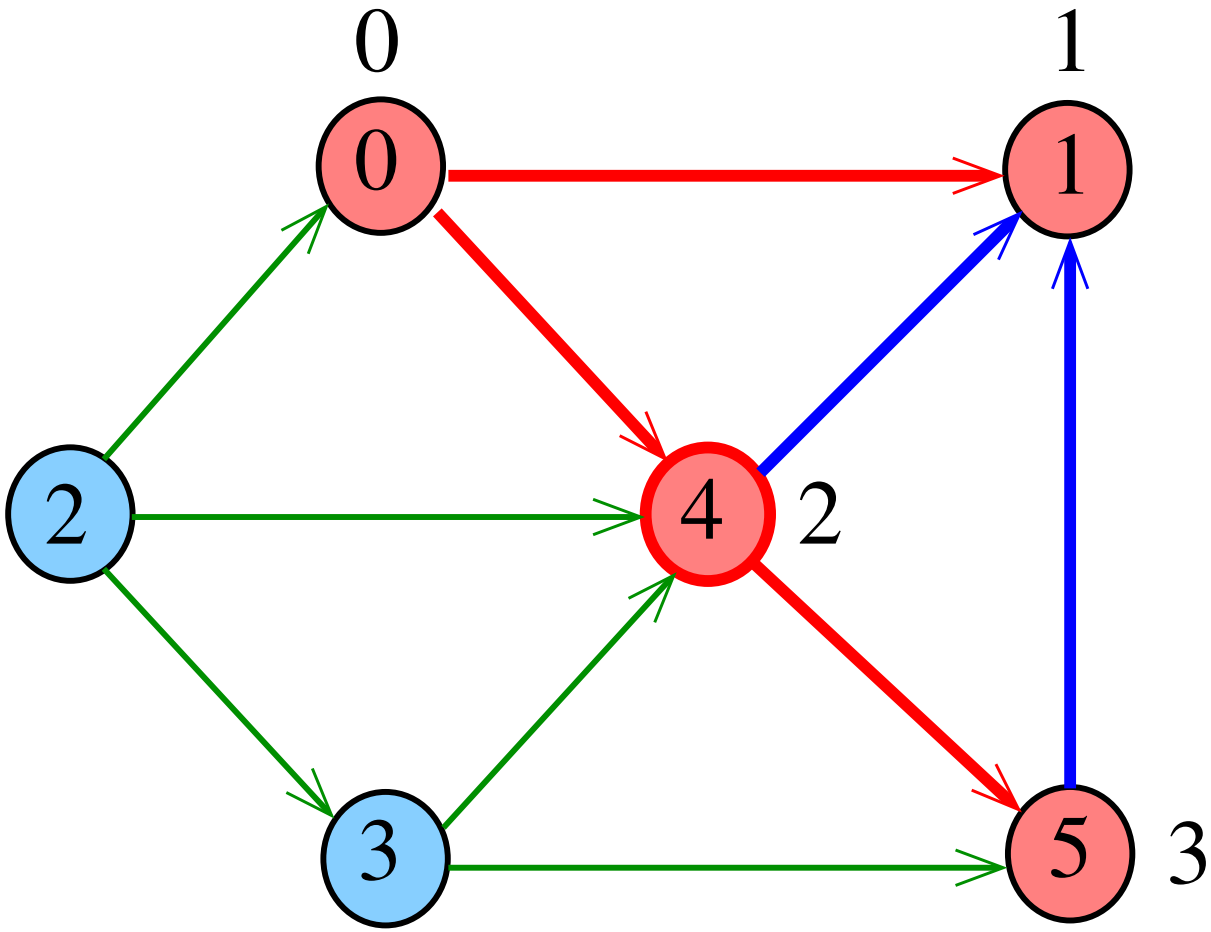
Algoritmo DFS

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



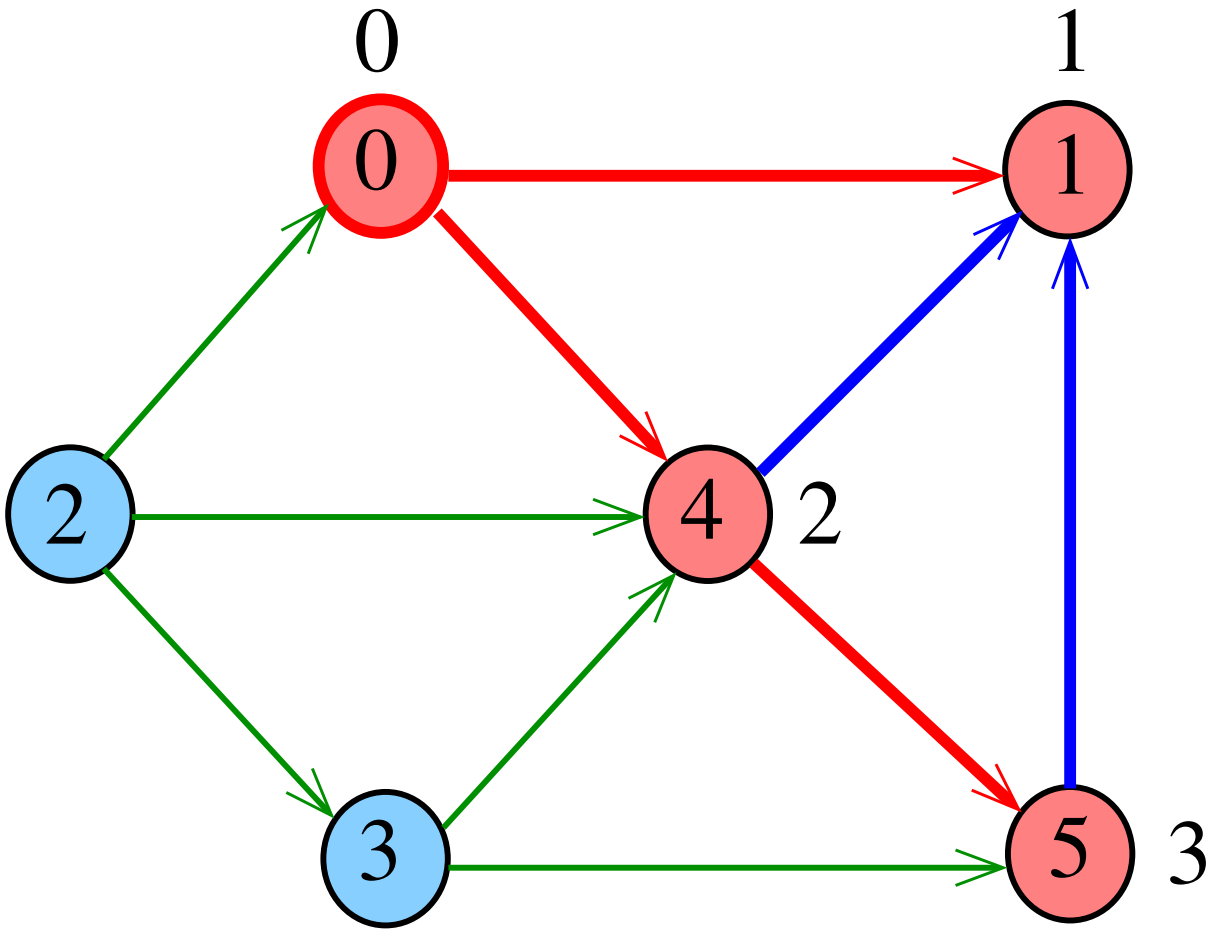
Algoritmo DFS

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



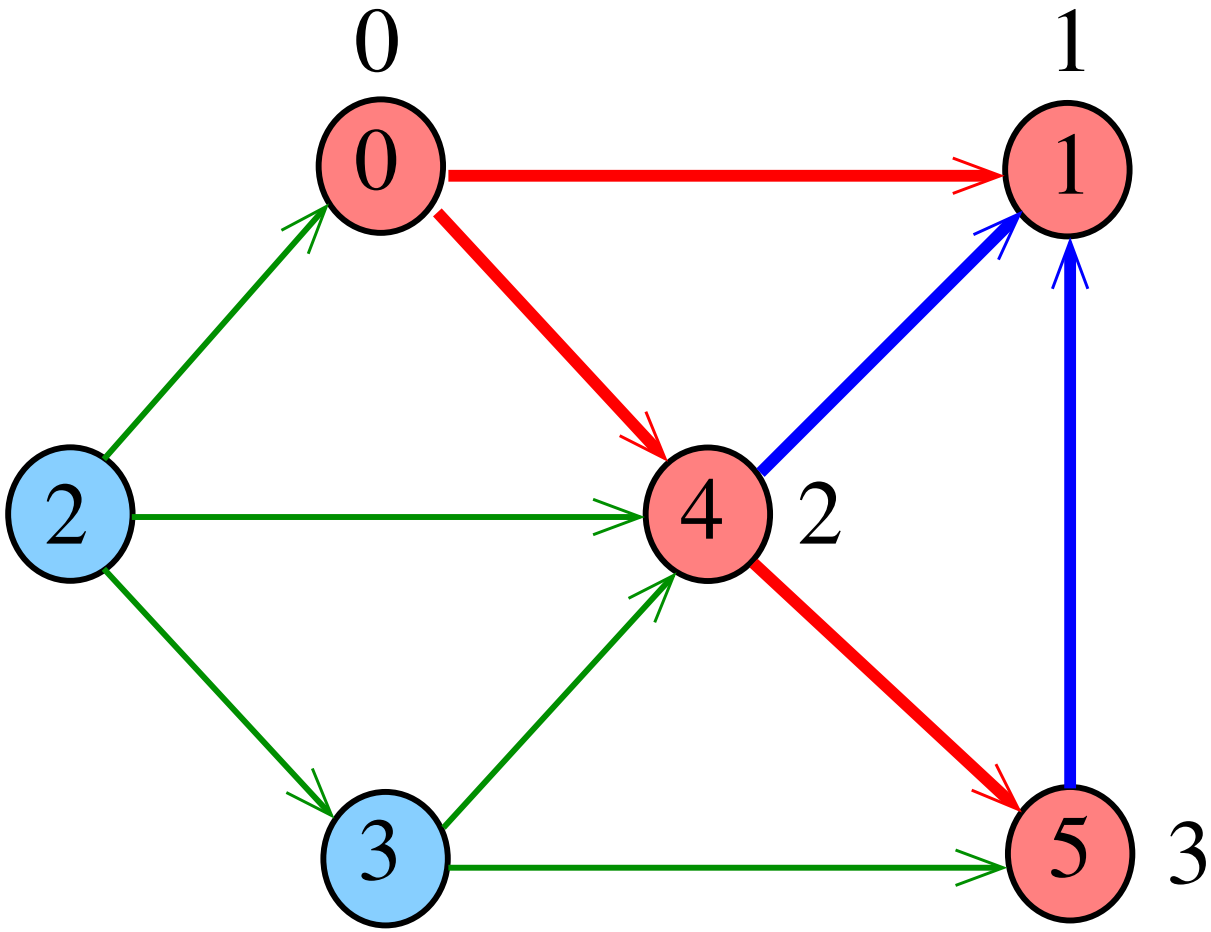
Algoritmo DFS

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



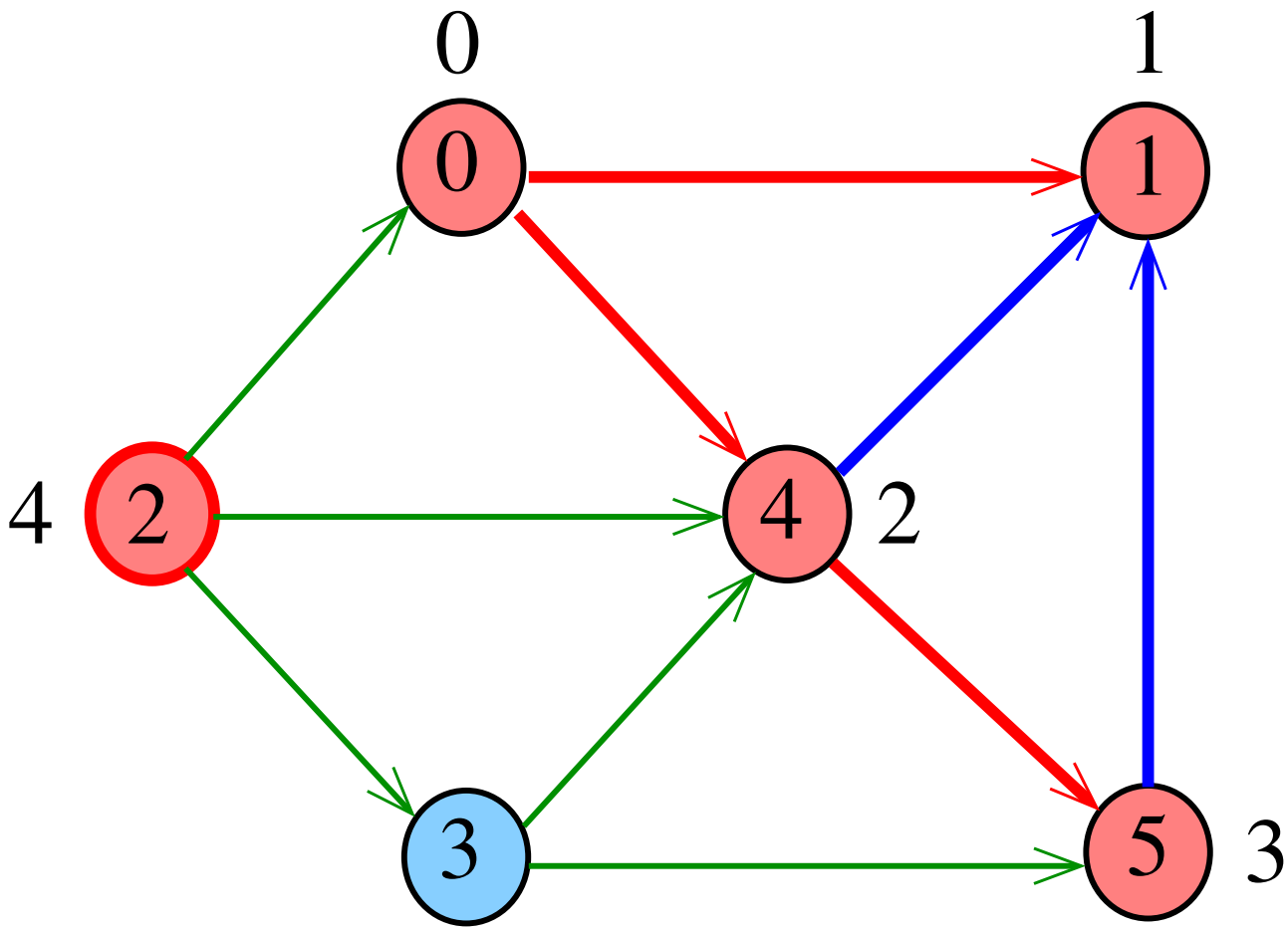
Algoritmo DFS

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



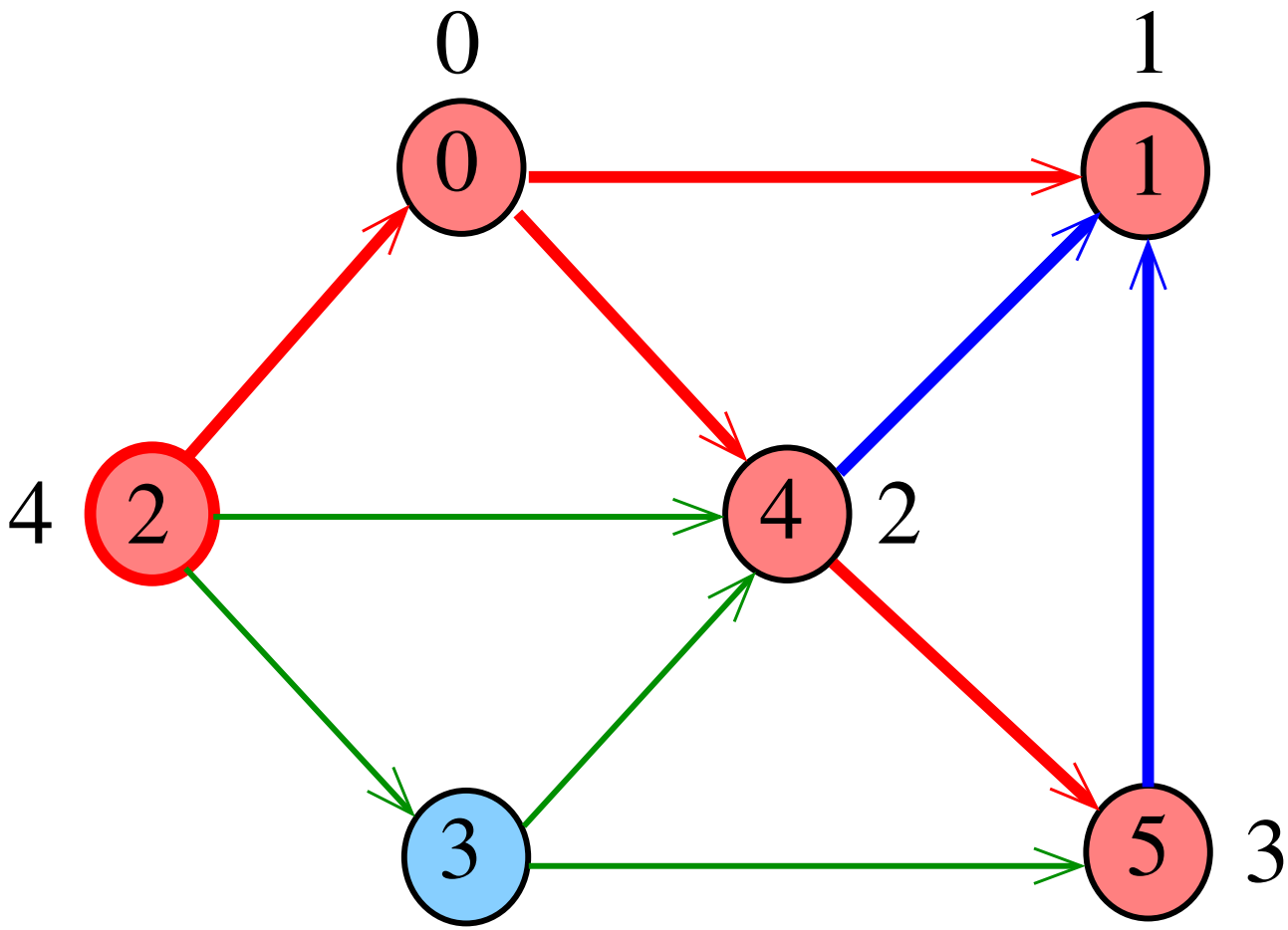
Algoritmo DFS

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



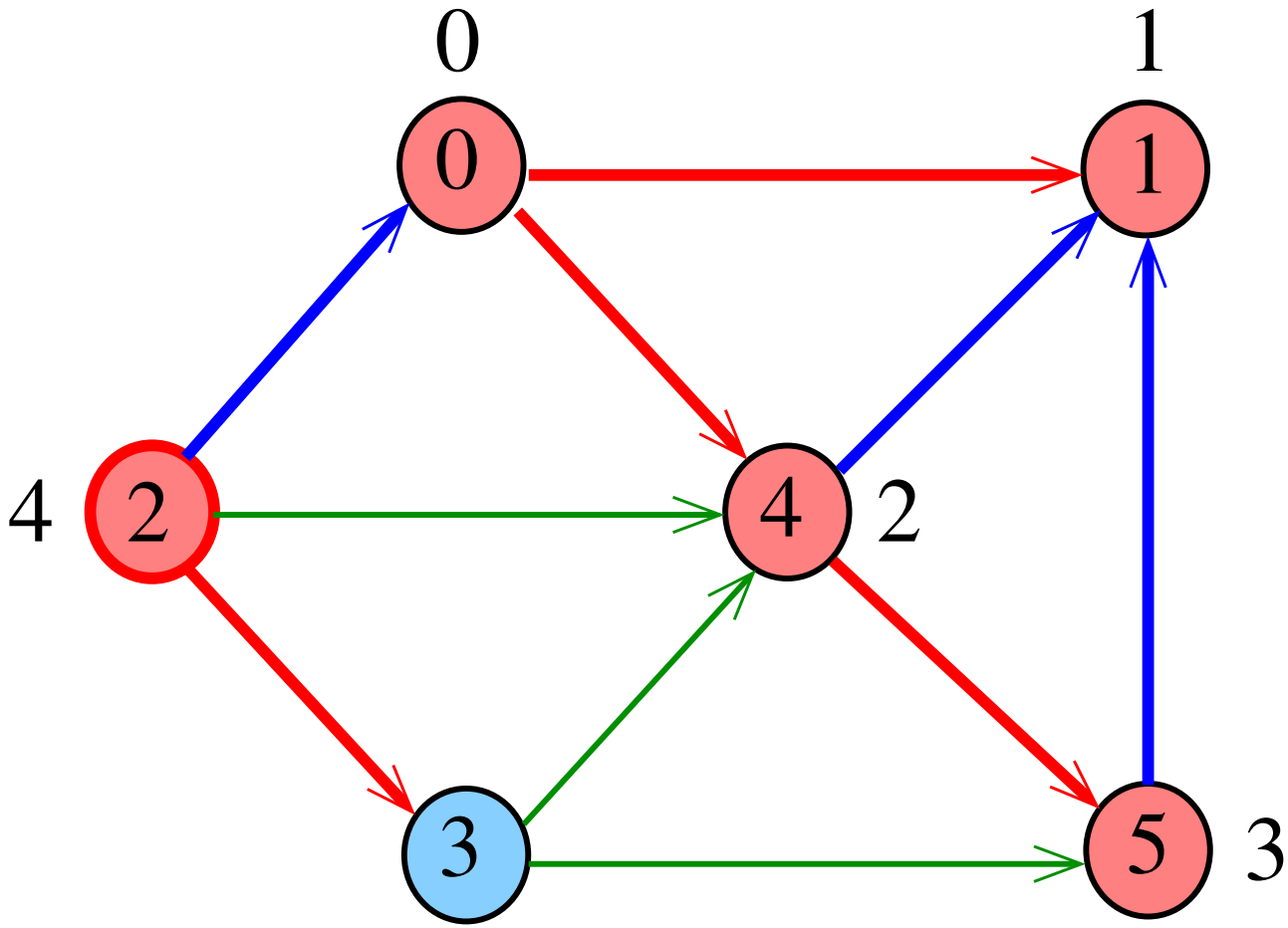
Algoritmo DFS

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



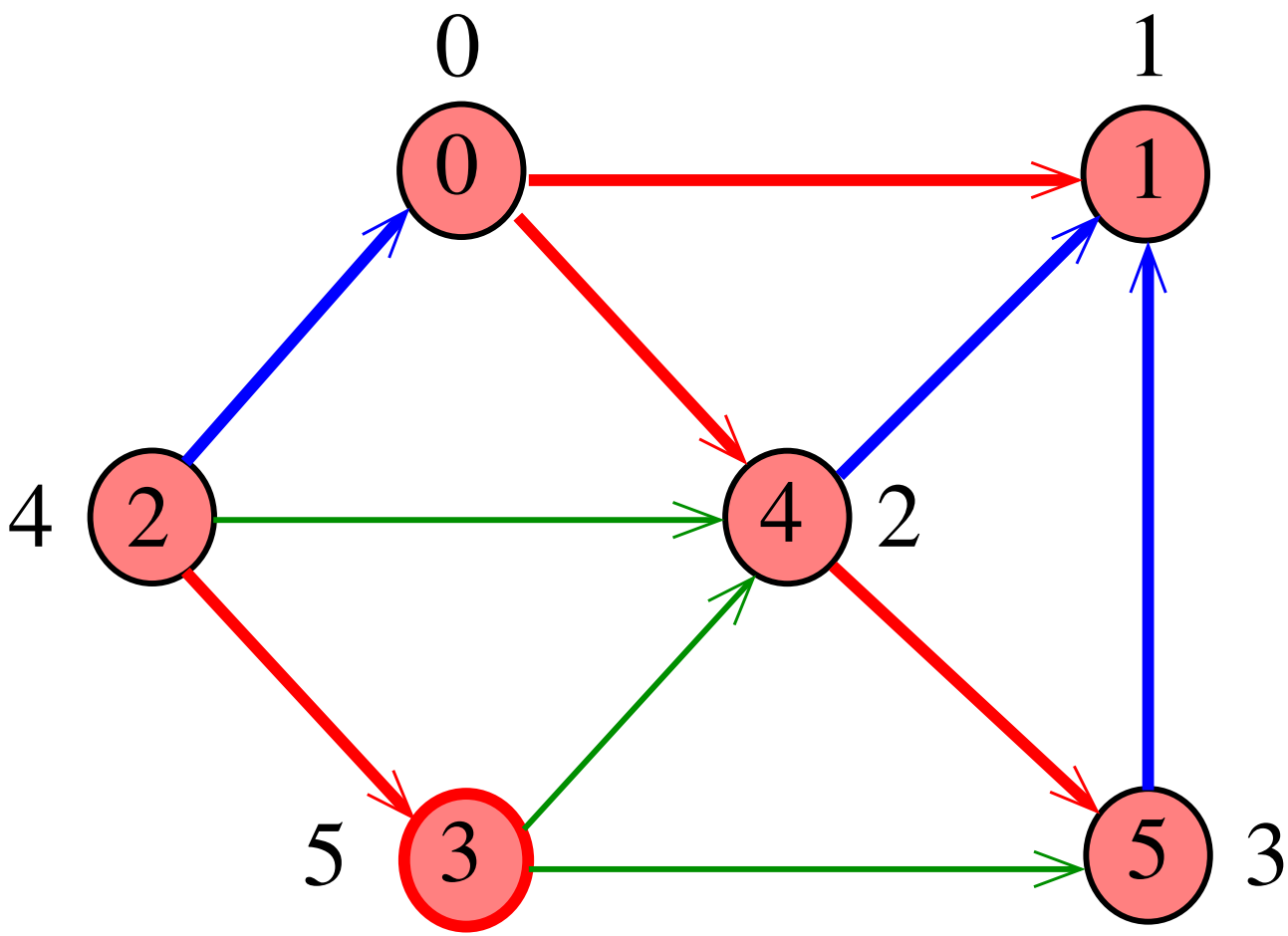
Algoritmo DFS

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



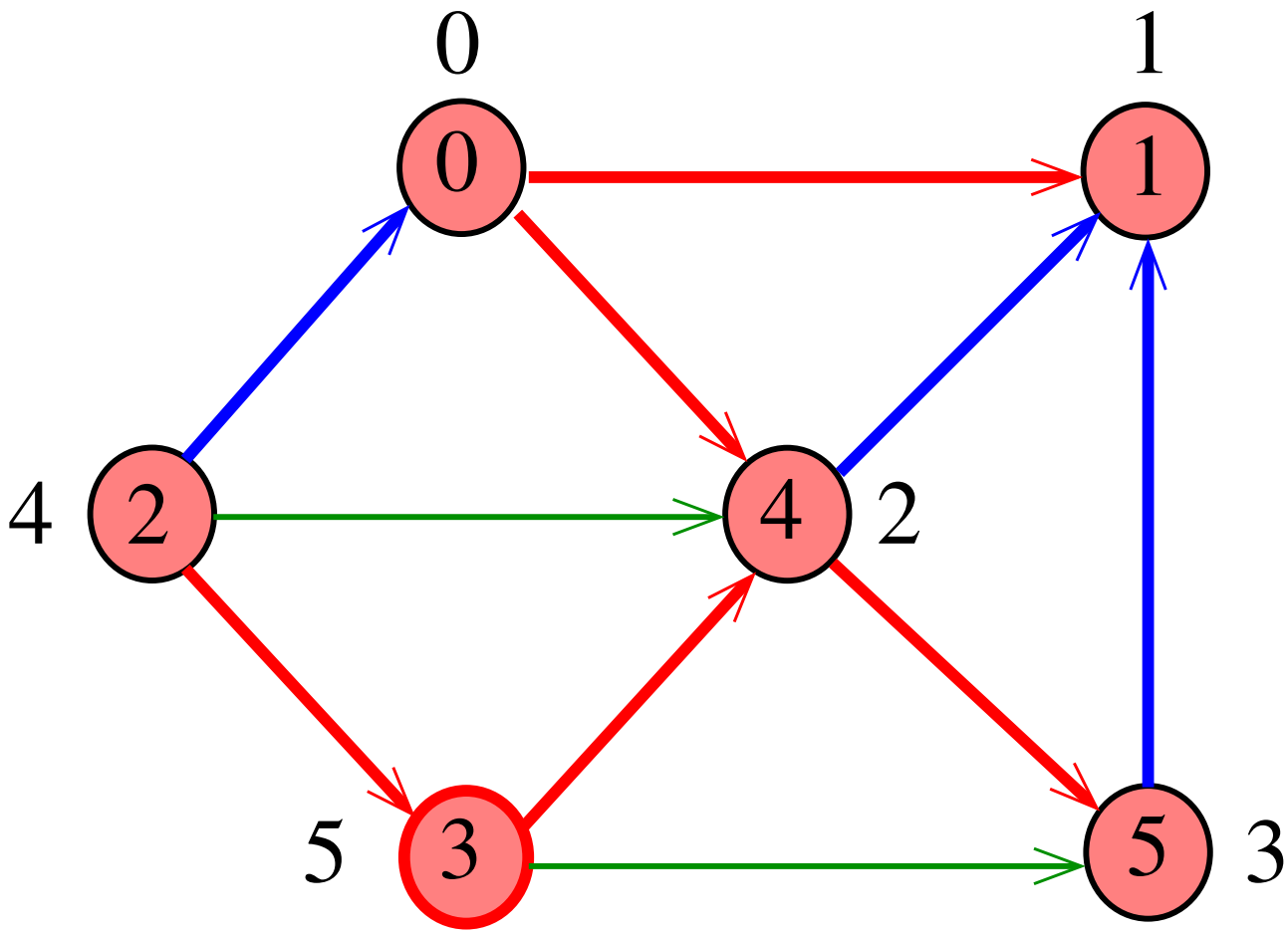
Algoritmo DFS

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



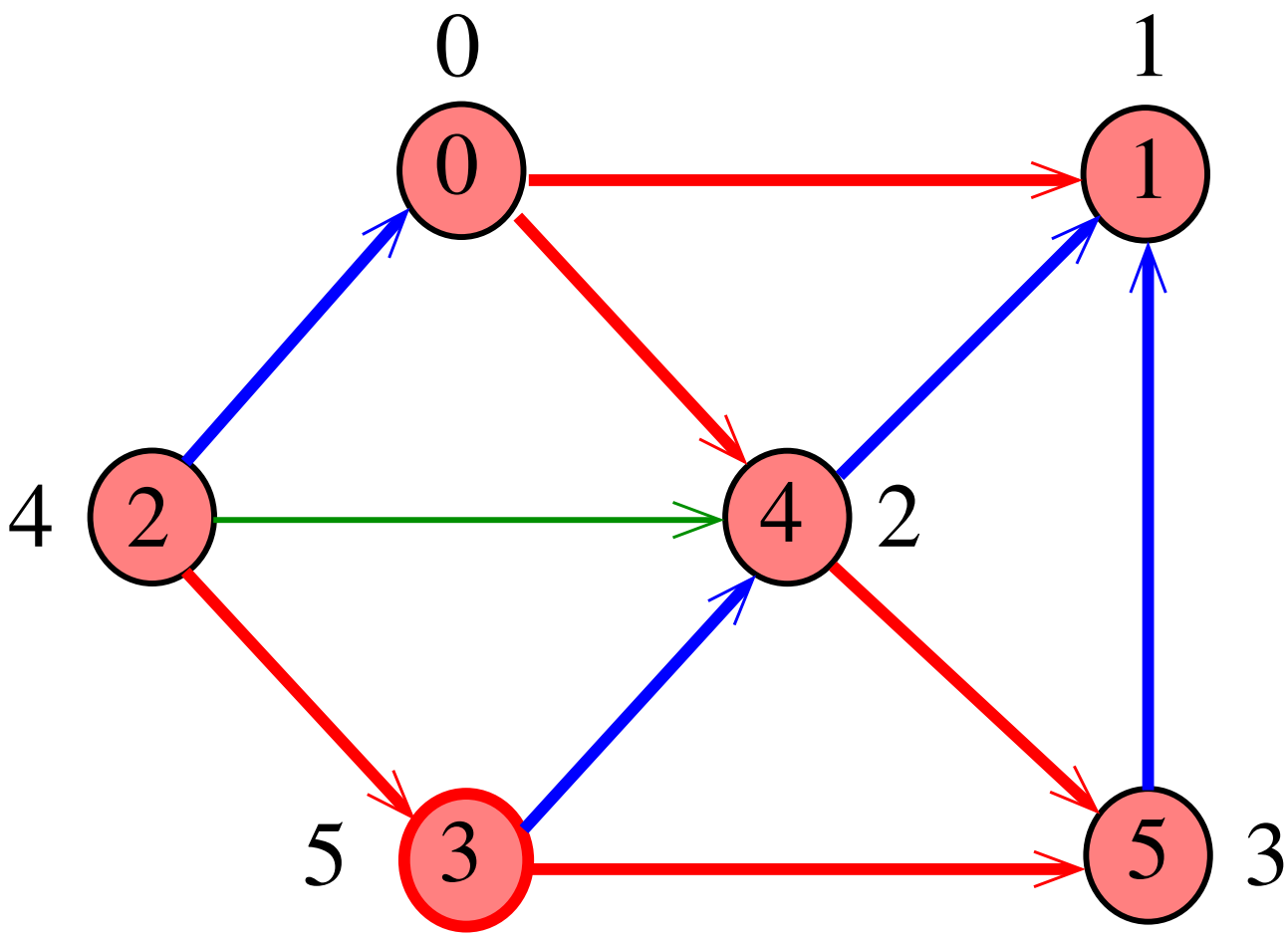
Algoritmo DFS

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



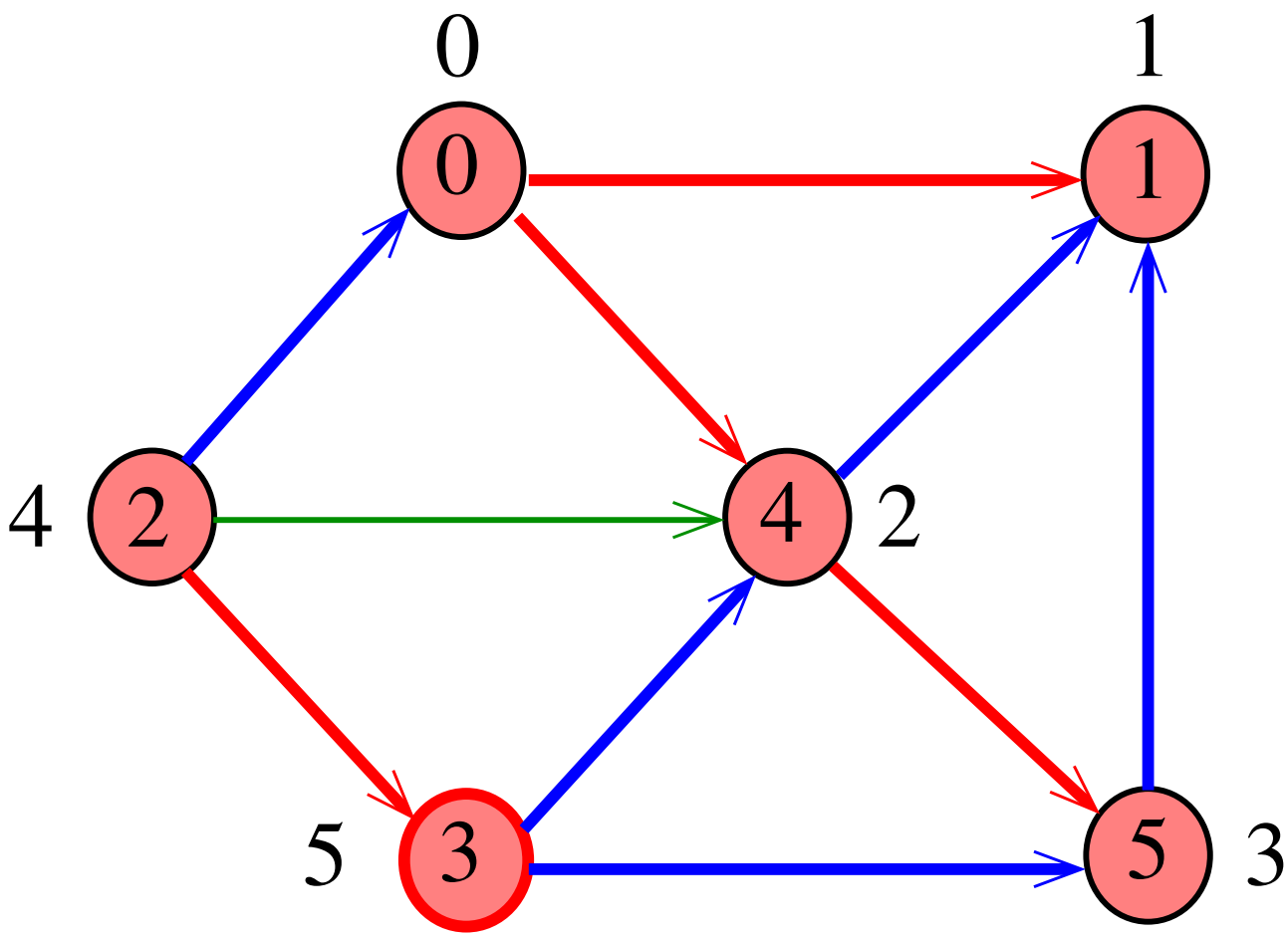
Algoritmo DFS

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



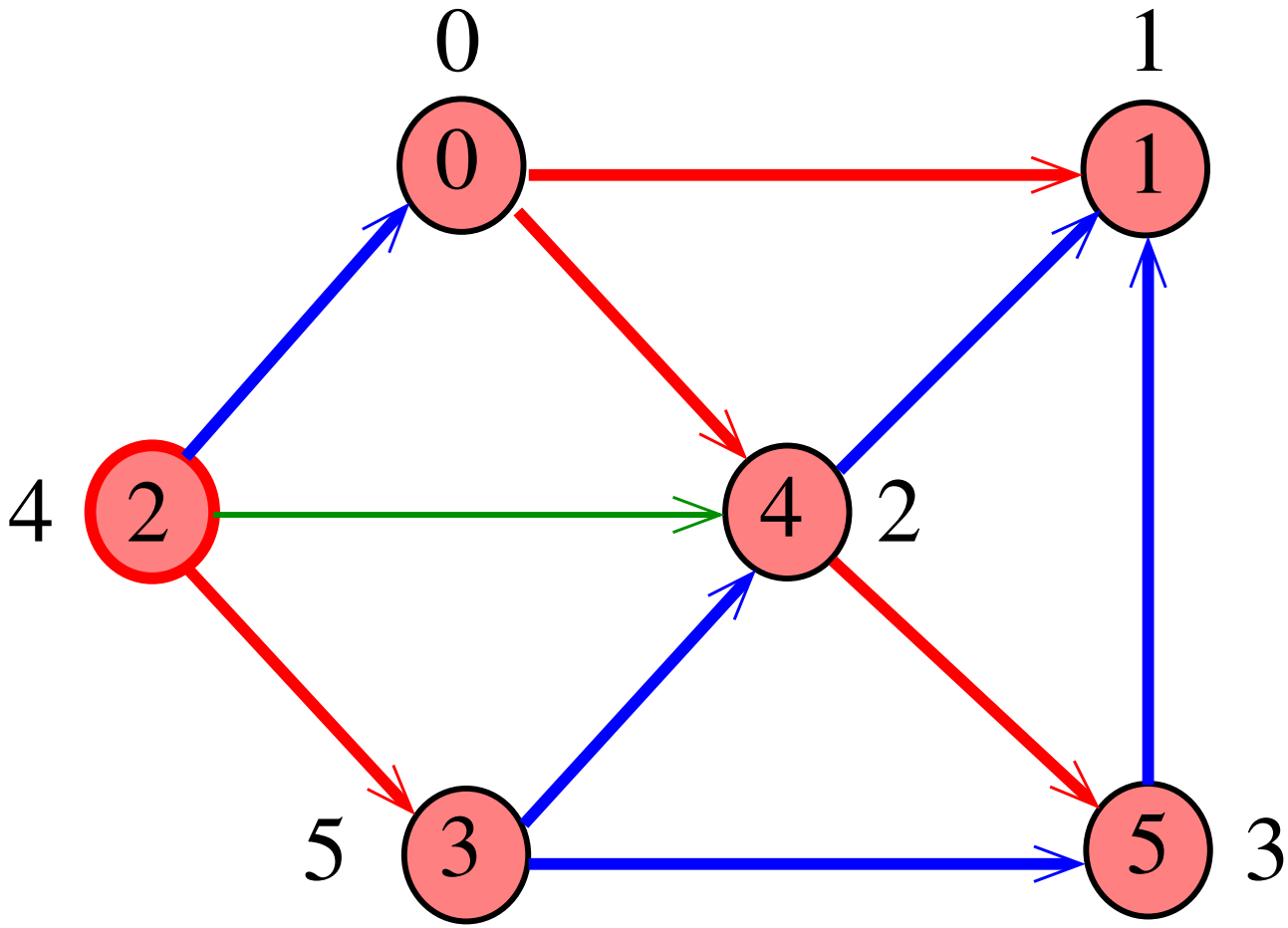
Algoritmo DFS

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



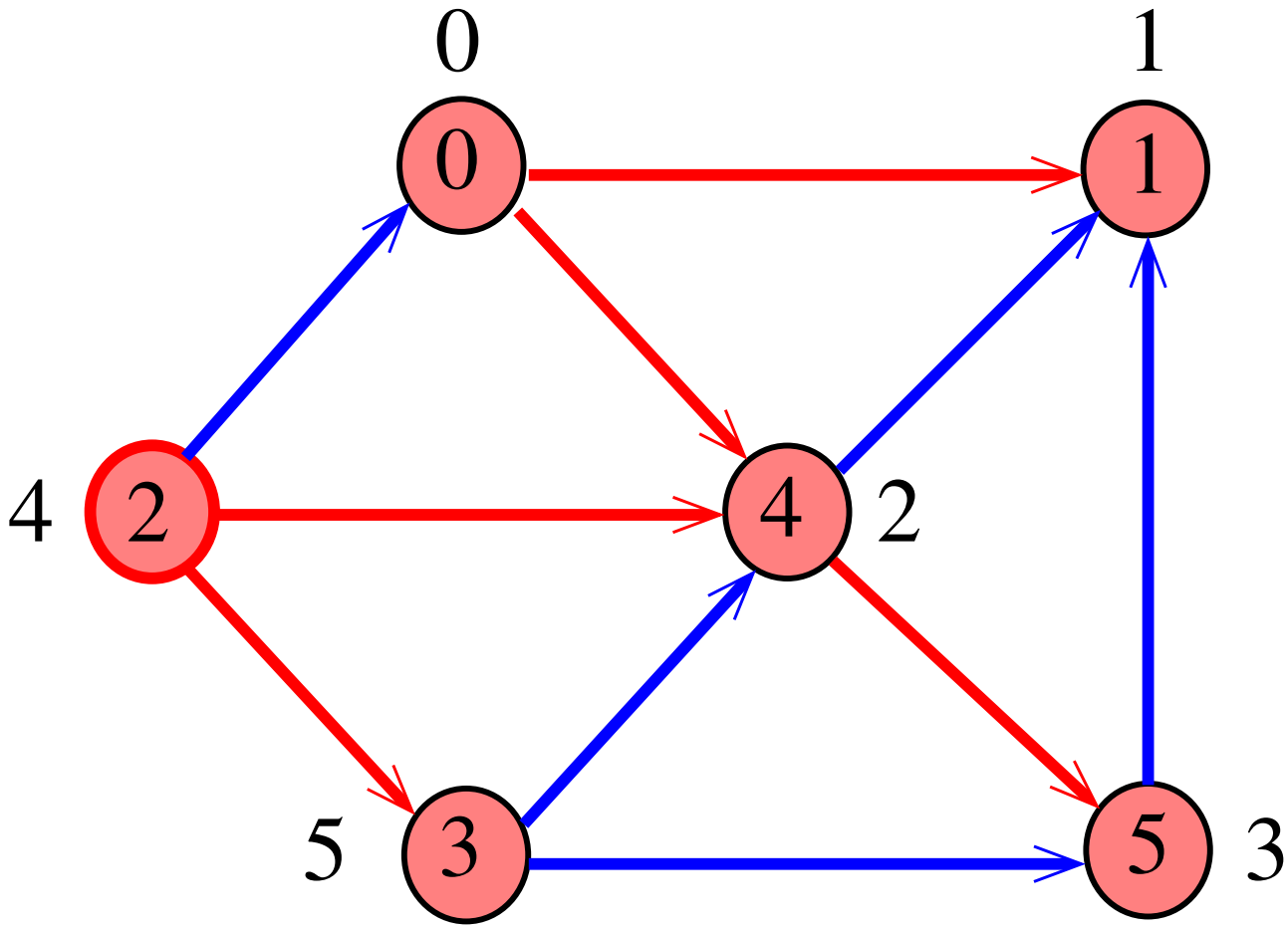
Algoritmo DFS

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



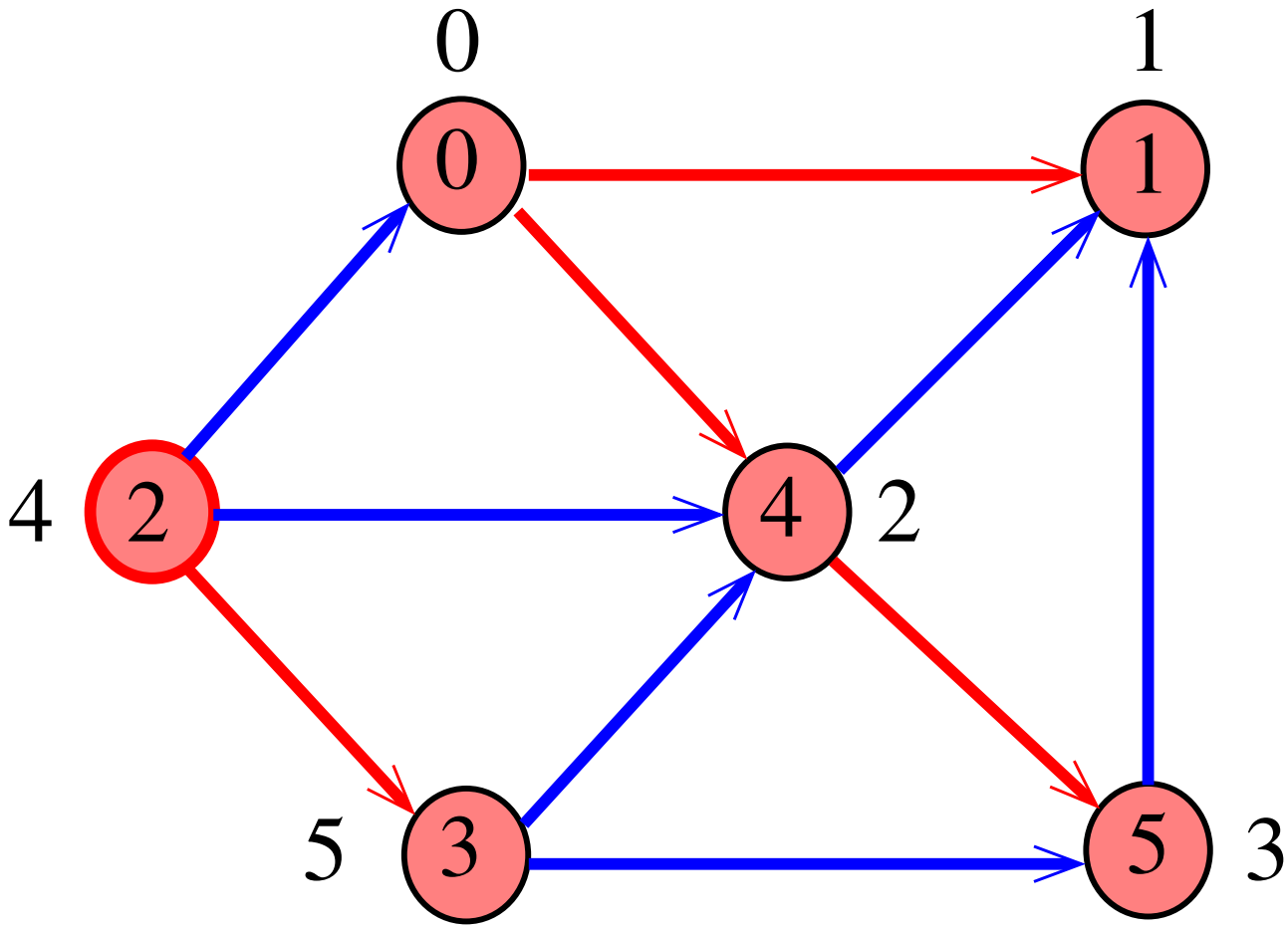
Algoritmo DFS

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



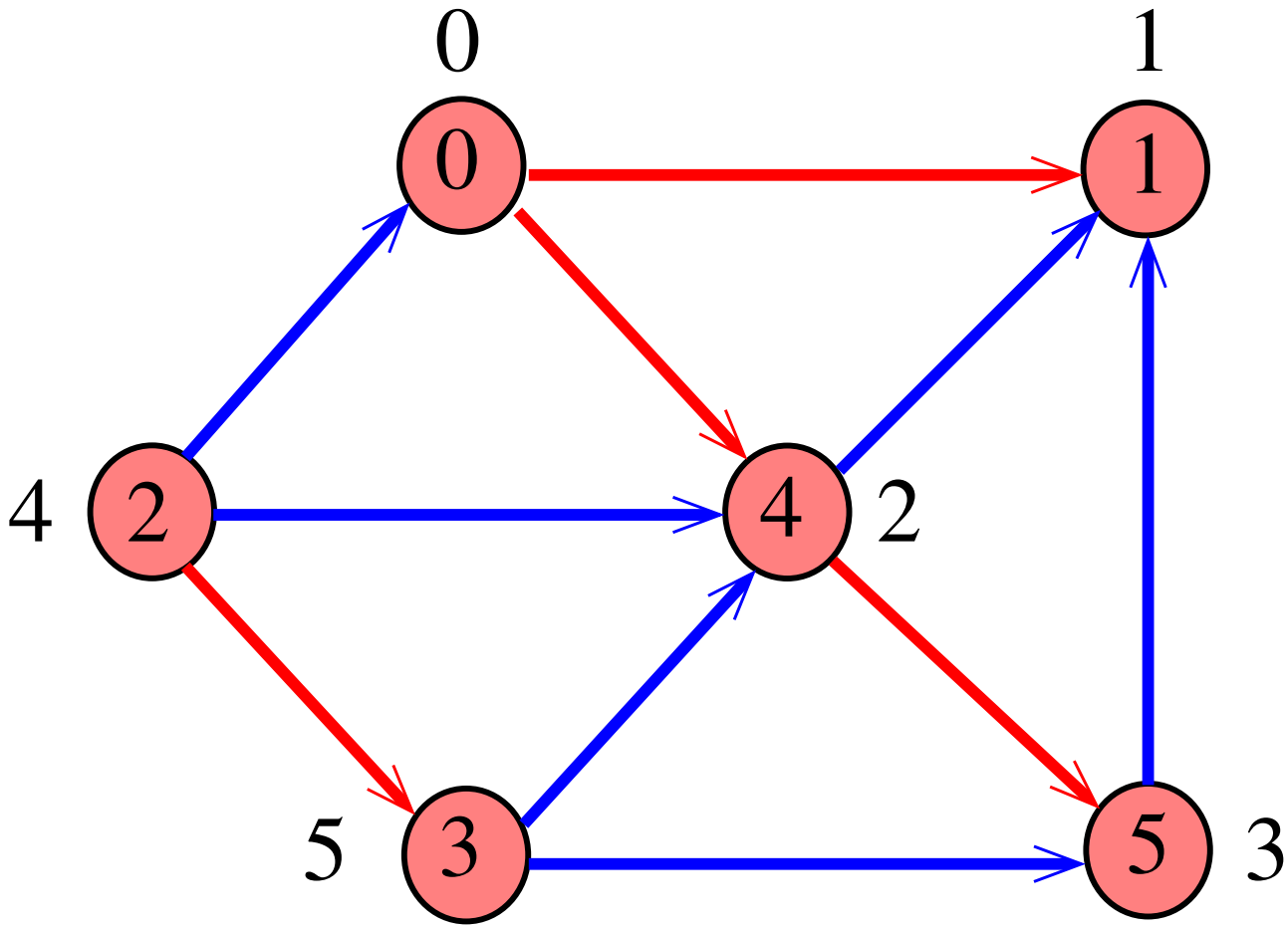
Algoritmo DFS

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



Algoritmo DFS

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



Ordenação topológica

Decidir se um dado digrafo `G` é um DAG.

```
static Stack ts
static Stack cycle;
static bool *onPath;
```

Se `G` é um DAG, uma ordenação topológica de seus vértices é armazenada em `ts`.

Se `G` não é um DAG, `cycle` armazenará um ciclo de `G`.

Ordenação topológica

Decidir se um dado digrafo `G` é um DAG.

```
static Stack ts
static Stack cycle;
static bool *onPath;
```

Se `G` é um DAG, uma ordenação topológica de seus vértices é armazenada em `ts`.

Se `G` não é um DAG, `cycle` armazenará um ciclo de `G`.

`onPath[v]` é `true` se o vértice `v` está no *caminho ativo*.

Ordenação topológica da DFS

```
static bool *marked;
static int  *edgeTo;

static bool *onPath;
static Stack ts;
static Stack cycle;
```

Ordenação topológica da DFS

```
static bool *marked;
static int  *edgeTo;

static bool *onPath;
static Stack ts;
static Stack cycle;

/* indicador de ciclo */
static int onCycle = -1;

bool hasCycle() {      /* G contém um ciclo? */
   return onCycle != -1;
}
bool isDag() {                /* G é um DAG? */
   return onCycle == -1;
}
```

Ordenação topológica da DFS

Determina se um digrafo G é acíclico, e portanto seus vértices têm uma ordem topológica, ou tem um ciclo.

```
marked = mallocSafe(G->V*sizeof(bool));
edgeTo = mallocSafe(G->V*sizeof(int));
onPath = mallocSafe(G->V*sizeof(bool));
for (int v = 0; v < G->V; v++)
   marked[v] = onPath[v] = false;
ts     = stackInit();
cycle = stackInit();
```

Ordenação topológica da DFS

Determina se um digrafo G é acíclico, e portanto seus vértices têm uma ordem topológica, ou tem um ciclo.

```
marked = mallocSafe(G->V*sizeof(bool));
edgeTo = mallocSafe(G->V*sizeof(int));
onPath = mallocSafe(G->V*sizeof(bool));
for (int v = 0; v < G->V; v++)
   marked[v] = onPath[v] = false;
ts     = stackInit();
cycle = stackInit();

for (int v = 0; v < G->V; v++)
   if (!marked[v] && onCycle == -1)
      dfs(G, v);
```

Alterações na dfs()

```
static void dfs(Digraph G, int v) {
    Link w;
    marked[v] = true;    onPath[v] = true;
    for (w = G->adj[v]; w != NULL; w = w->next) {
        if (hasCycle()) return;
        if (!marked[w->vertex]) {
            edgeTo[w->vertex] = v;
            dfs(G, w->vertex);
        }
```

Alterações na dfs()

```
static void dfs(Digraph G, int v) {
    Link w;
    marked[v] = true;    onPath[v] = true;
    for (w = G->adj[v]; w != NULL; w = w->next) {
        if (hasCycle()) return;
        if (!marked[w->vertex]) {
            edgeTo[w->vertex] = v;
            dfs(G, w->vertex);
        } else if (onPath[w->vertex]) {
            onCycle = v;
            edgeTo[w->vertex] = v; /* fecha o ciclo */
        }
    }
    onPath[v] = false;    push(ts, v);
}
```

Rotina Cycle()

Retorna uma pilha com o ciclo se G possui um ciclo, ou NULL em caso contrário.

```
Stack Cycle() {
   if (!hasCycle()) return NULL;
   if (!stackEmpty(cycle)) return cycle;
   for (int x = edgeTo[onCycle];
         x != onCycle; x = edgeTo[x])
      push(cycle, x);
   push(cycle, onCycle);
   return cycle;
}
```

Rotina `order()`

Retorna uma pilha com a ordem topológica
dos vértices de `G` se `G` é um DAG,
ou `NULL` em caso contrário.

```
Stack order() {
   if (!isDag()) return NULL;
   return ts;
}
```

Consumo de tempo

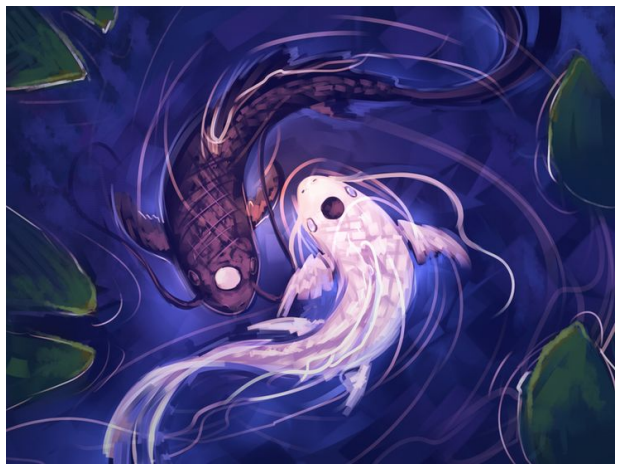
O consumo de tempo para obter
uma ordenação topológica com DFS para
vetor de listas de adjacência é $O(V + E)$.

Ordenação topológica com DFS, para
matriz de adjacências, consome tempo $O(V^2)$.

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.

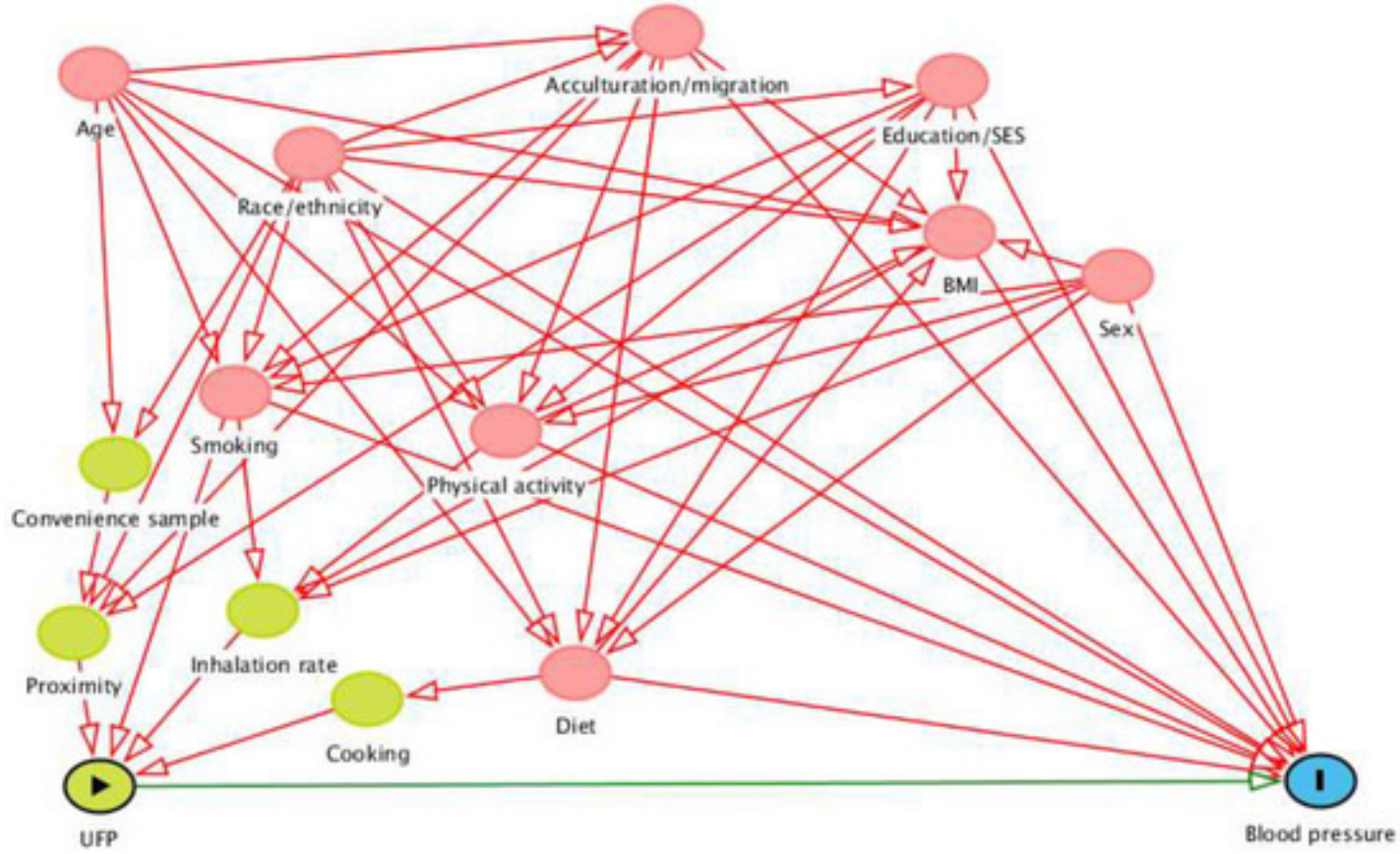


Fonte: Avatar: The Last Airbender

Exercício

Adapte a implementação apresentada da ordenação topológica para que devolva um struct `DFStopological`, como fizemos com a `DFSanatomia`.

Usaremos isso algumas aulas adiante.



Fonte: Relationship of Time-Activity-Adjusted Particle Number Concentration with Blood Pressure