Repaso I2

Estructuras de Datos y Algoritmos

2-2020

Yadran Eterovic

Temario

Hashing

Grafos: introducción

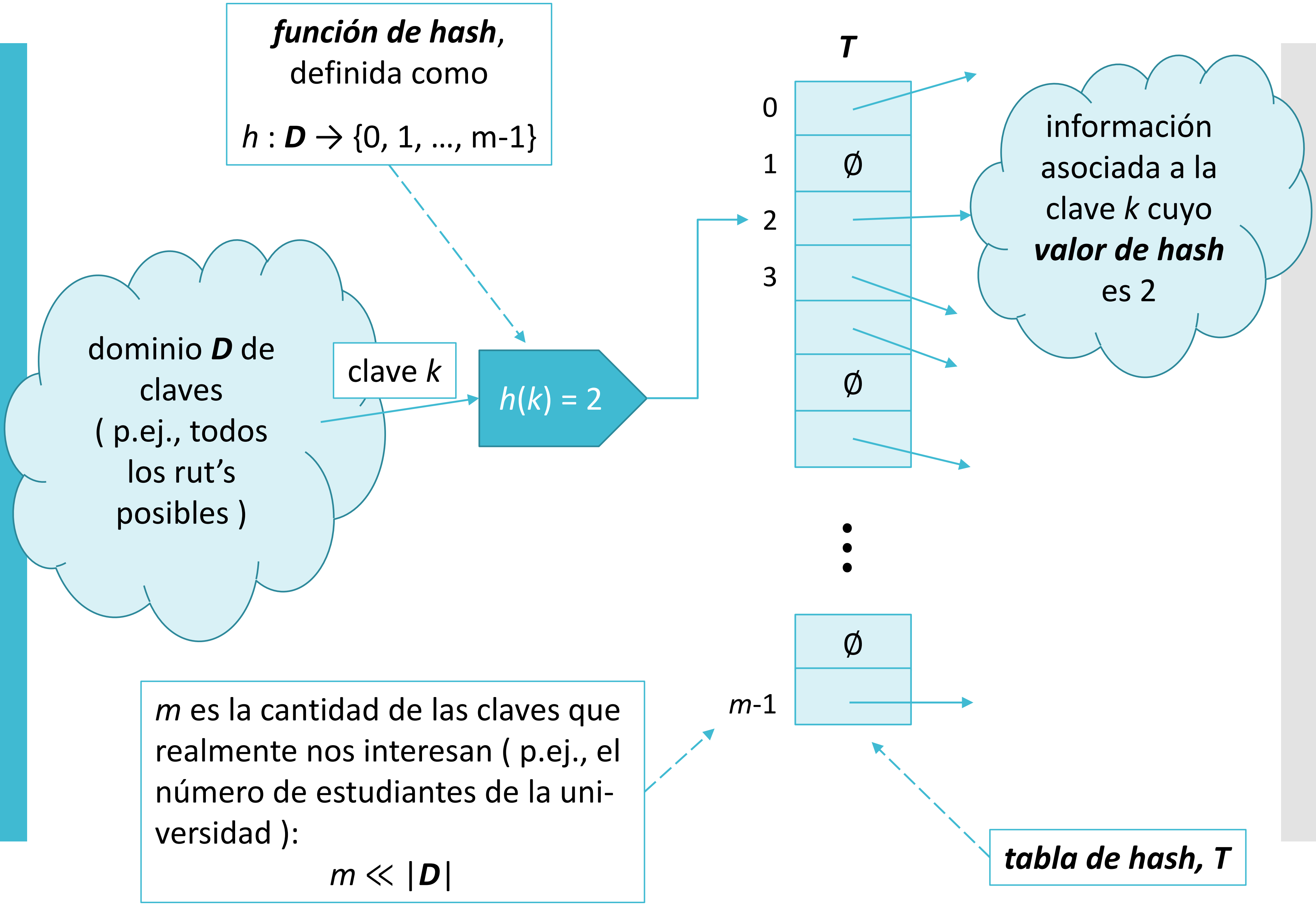
Grafos: **DFS + ordenación topológica + comp.conexas**

Grafos: **BFS + Dijkstra**

Backtracking

Tabla de *hash*:

- la clave no se usa directamente como índice en *T*
- el índice **se calcula** a partir de la clave



Colisiones

Funciones de hash, según el tipo de datos de las claves

Números enteros:

- $h(k) = k \bmod m$ —división o hashing modular
- $h(k) = \lfloor m \cdot (A \cdot k \bmod 1) \rfloor$ —multiplicación

Números reales entre 0 y 1:

- hashing modular sobre la representación binaria de la clave

Strings:

- primero hay que convertir $s = ch_0ch_1 \ldots ch_p$ a un número entero k y luego ajustar k al tamaño m de la tabla
- $k = \#(ch_p) + \#(ch_{p-1}) \times R + \#(ch_{p-2}) \times R^2 + \ldots + \#(ch_0) \times R^p$, $R = 31$ o $R = 37$
- $i = k \bmod m$

Algunas propiedades de hashing

Una tabla de hash se comporta “casi” como un arreglo:

- en un arreglo, buscar el dato con clave k consiste simplemente en mirar $T[k]$ → es O(1) (ver diap. # 6)
- en hashing, buscar el dato con clave k consiste en mirar $T[h(k)]$ → es O(1) pero sólo **en promedio**, como vamos a ver

En hashing el orden relativo de las claves **no importa**:

- comparar claves entre ellas (para determinar cuál es mayor)

  … o, dada una clave, encontrar la clave predecesora

  … **no son** operaciones de diccionario

Grafos

Representación:

- listas de adyacencias, matriz

Algoritmos de recorrido/descubrimiento:

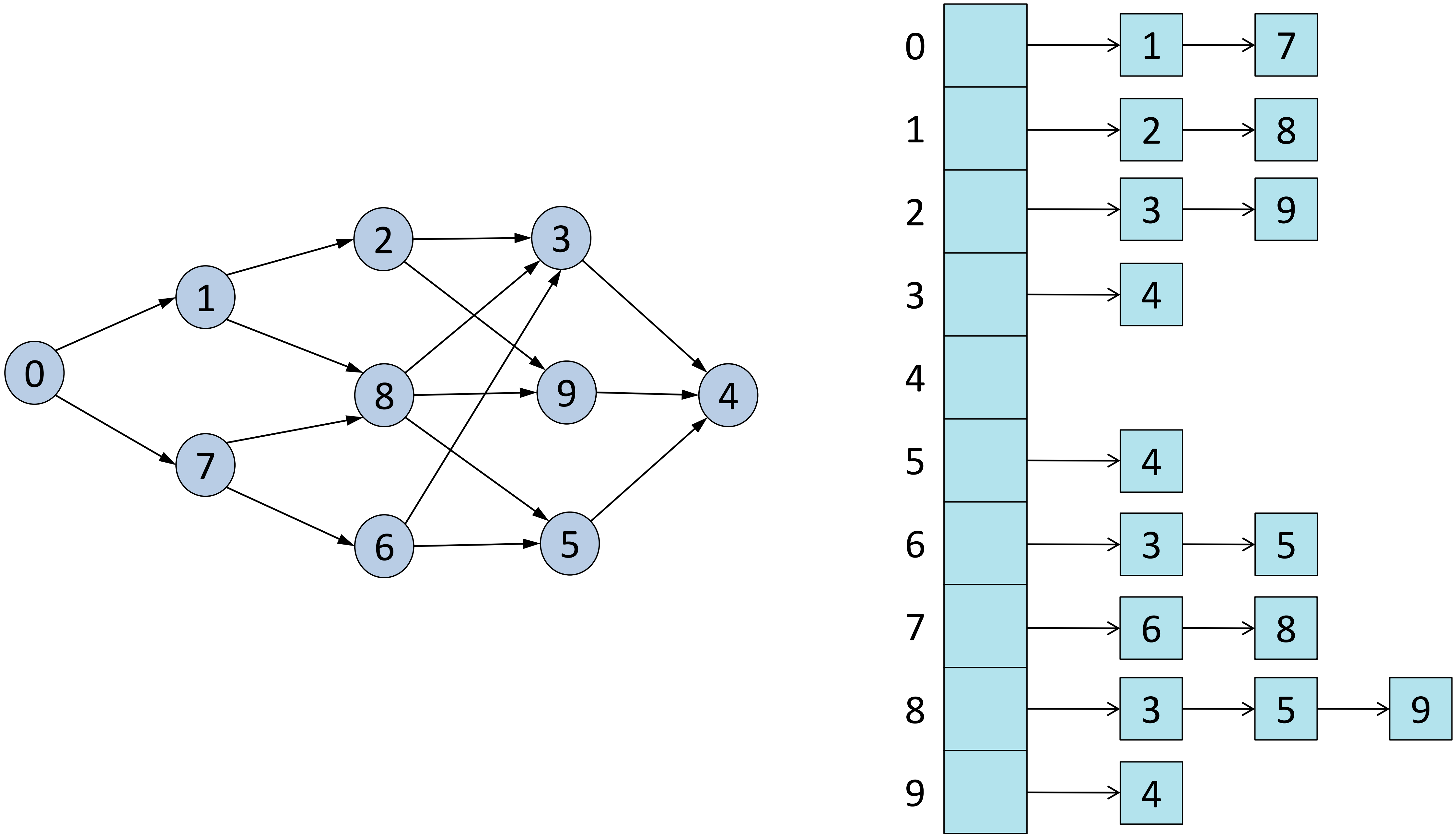
- DFS, BFS

Algoritmos de detección de propiedades:

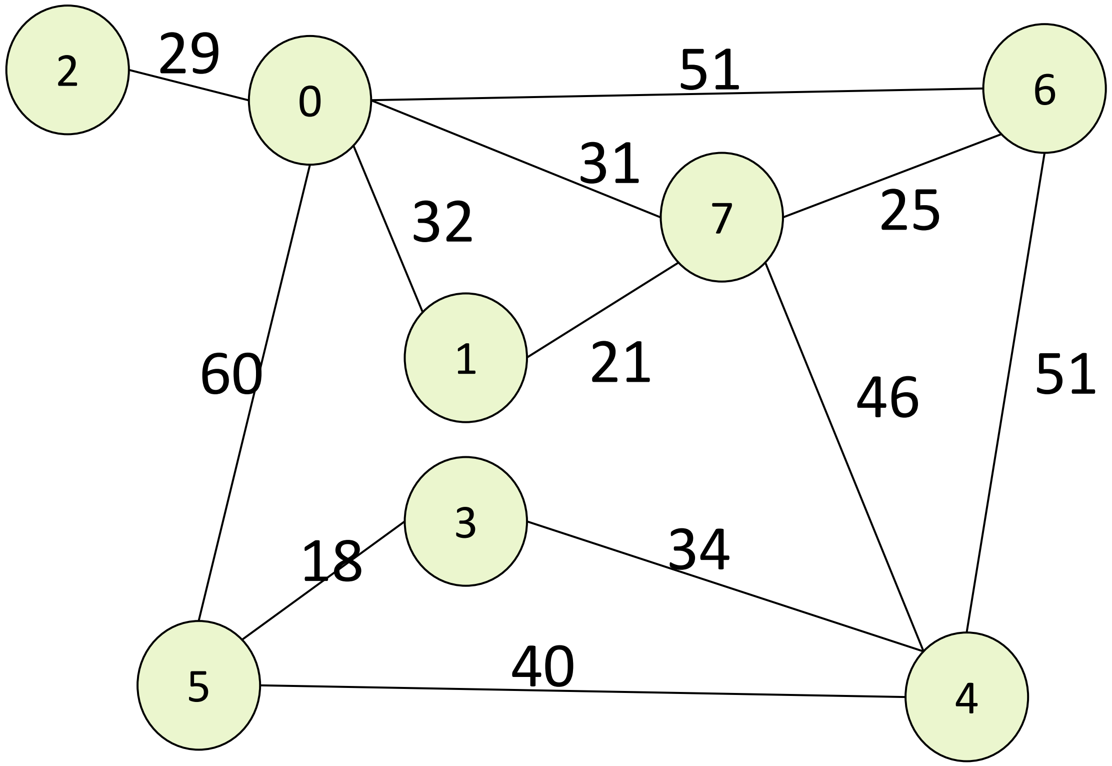
- ordenación topológica, componentes fuertemente conectadas

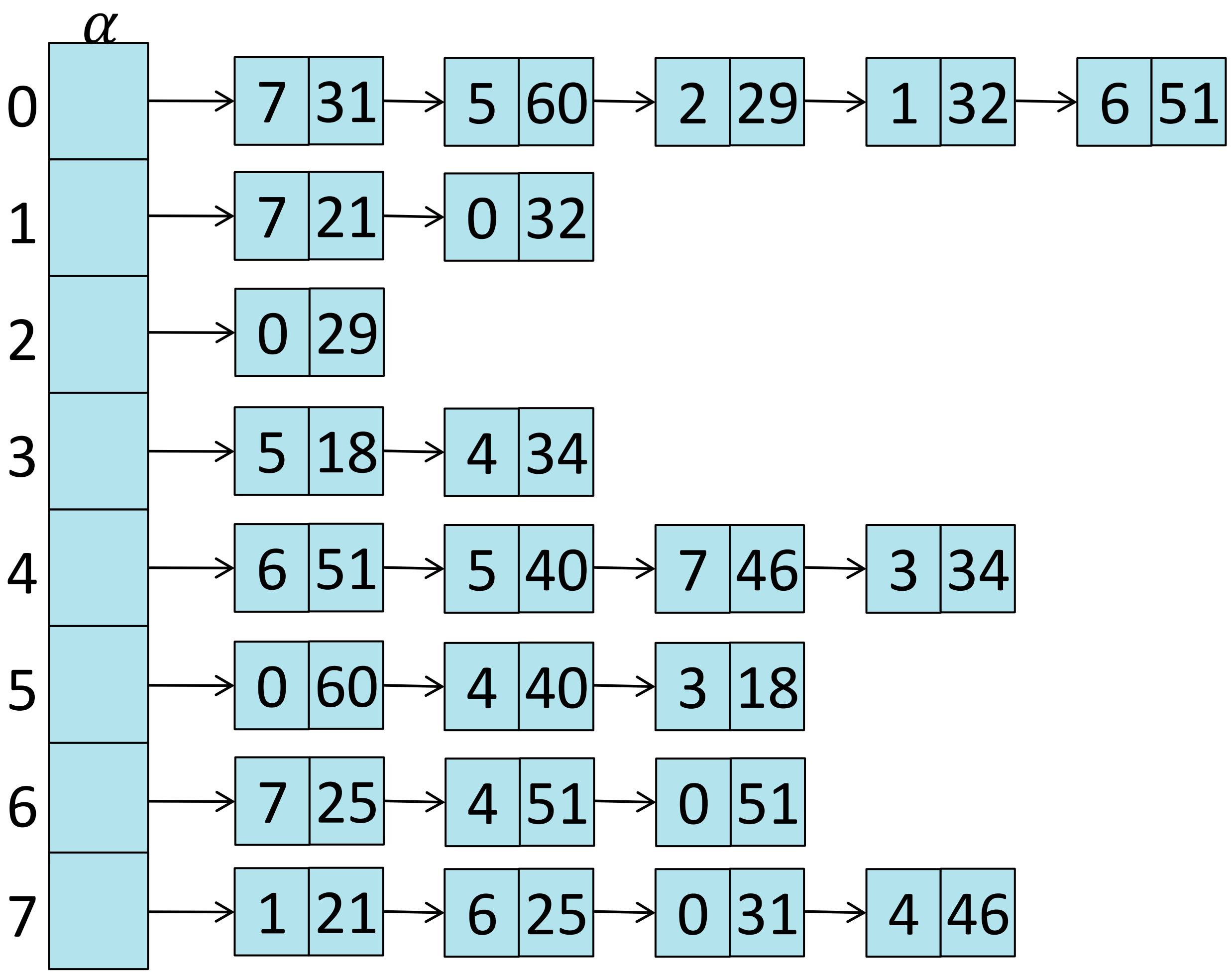
Algoritmos de optimización:

- Dijkstra



0
1
2
3
4
5
6
7
8
9
0 1 7
1 2 8
2 3 9
3 4
4
5 4
6 3 5
7 6 8
8 3 5 9
9 4


2
29
0
51
6
31
32
7
25
1
21
60
46
51
3
18
34
40
5
4


α
0
7 31
5 60
2 29
1 32
6 51
1
7 21
0 32
2
0 29
3
5 18
4 34
4
6 51
5 40
7 46
3 34
5
0 60
4 40
3 18
6
7 25
4 51
0 51
7
1 21
6 25
0 31
4 46

DFS

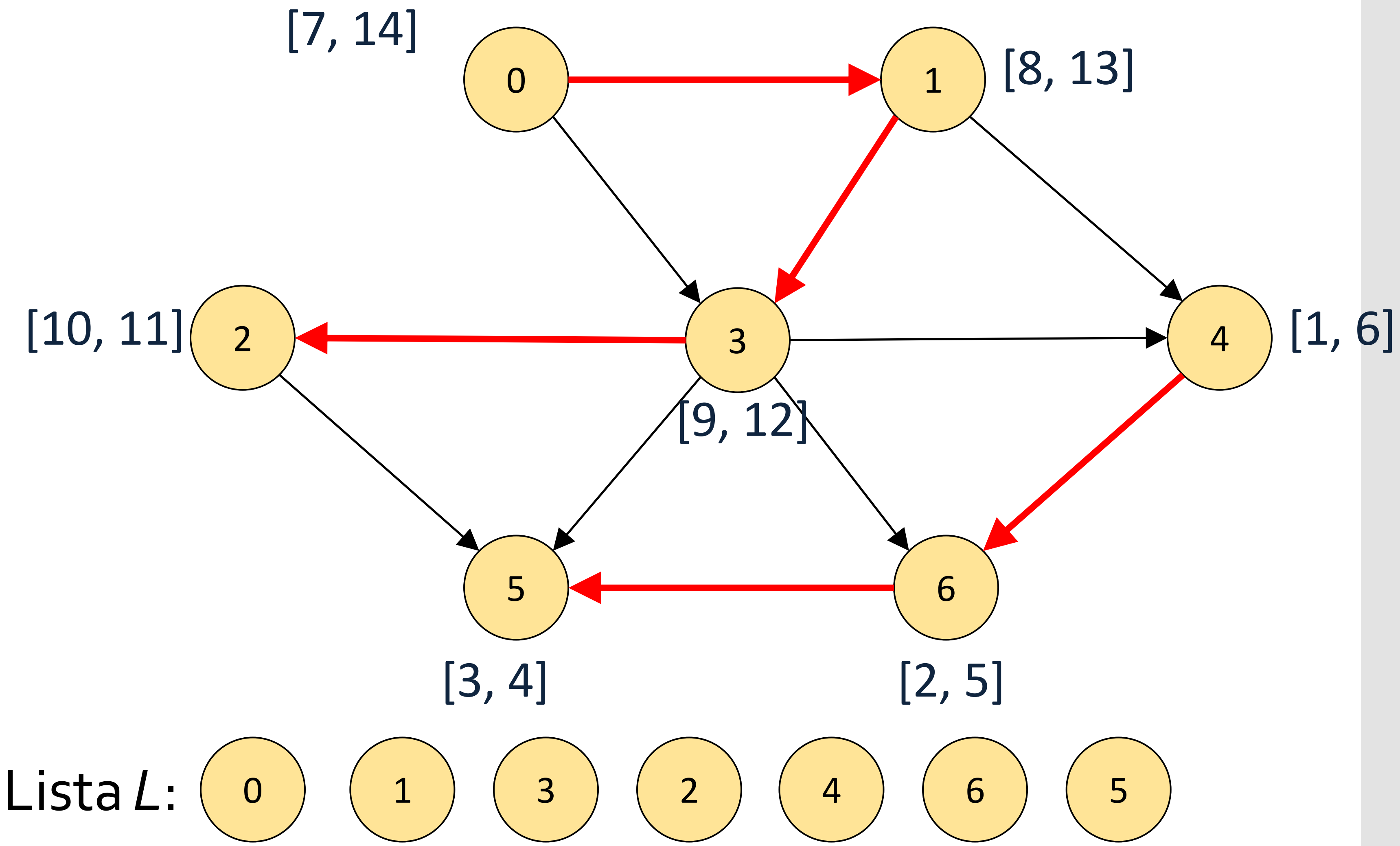
```
dfs():
    for each u in V:
        u.color = white
        π[u] = null
    time = 0
    for each u in V:
        if u.color == white:
            time = dfsVisit(u, time)

dfsVisit(u, time):
    u.color = gray
    time = time+1
    u.start = time
    for each v in α[u]:
        if v.color == white:
            π[v] = u
            time = dfsVisit(v, time)
    u.color = black
    time = time+1
    u.end = time
    return time
```

BFS

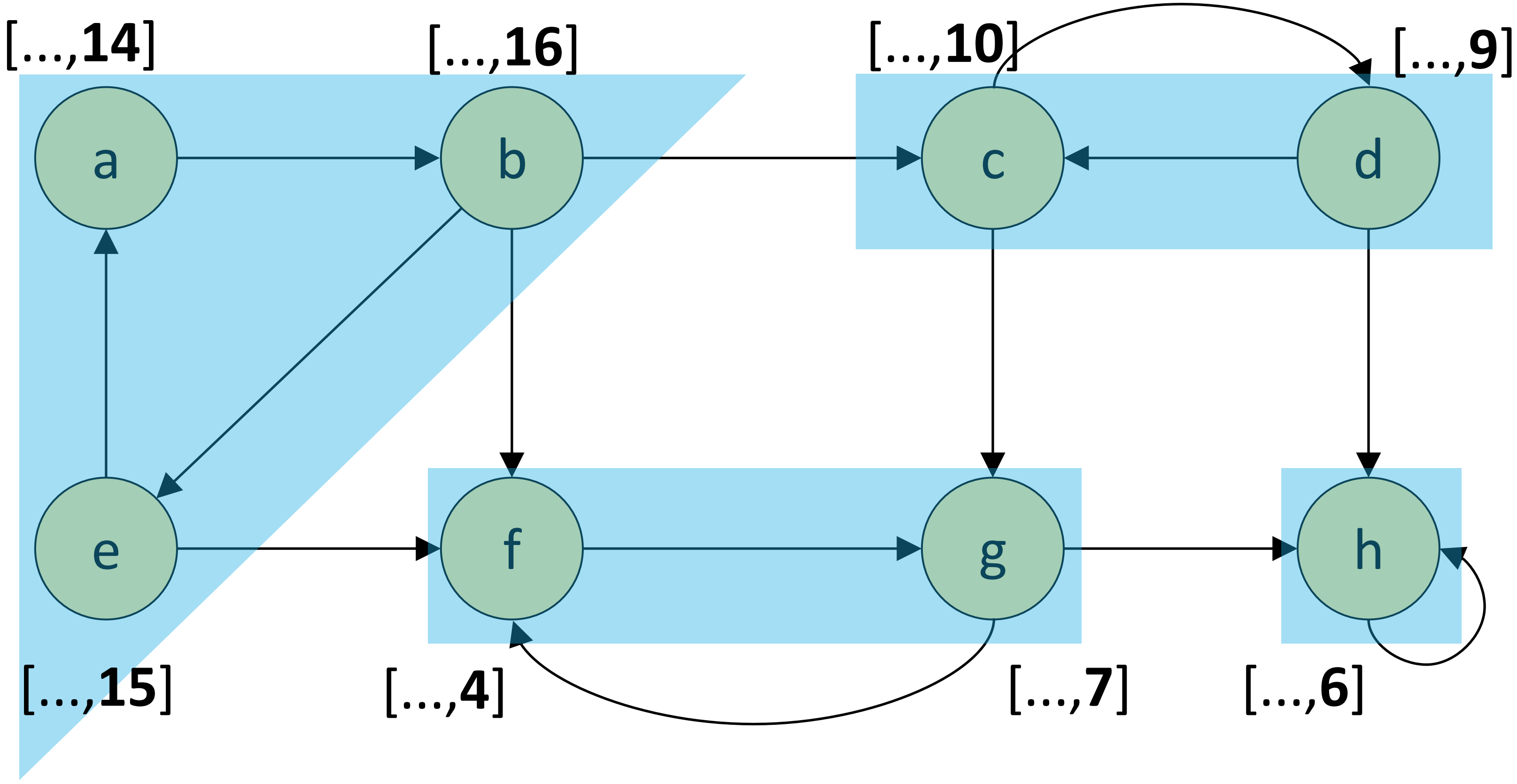
```
BFS(s): —s es el vértice de partida
    for each u in V-{s}:
        u.color ← white; u.δ ← ∞; π[u] ← null
    s.color ← gray; s.δ ← 0; π[s] ← null
    Q ← cola; Q.enqueue(s)
    while !Q.empty():
        u ← Q.dequeue()
        for each v in α[u]:
            if v.color == white:
                v.color ← gray; v.δ ← u.δ+1
                π[v] ← u; Q.enqueue(v)
        u.color ← black
```

Ordenación topológica: grafos direccionales sin ciclos



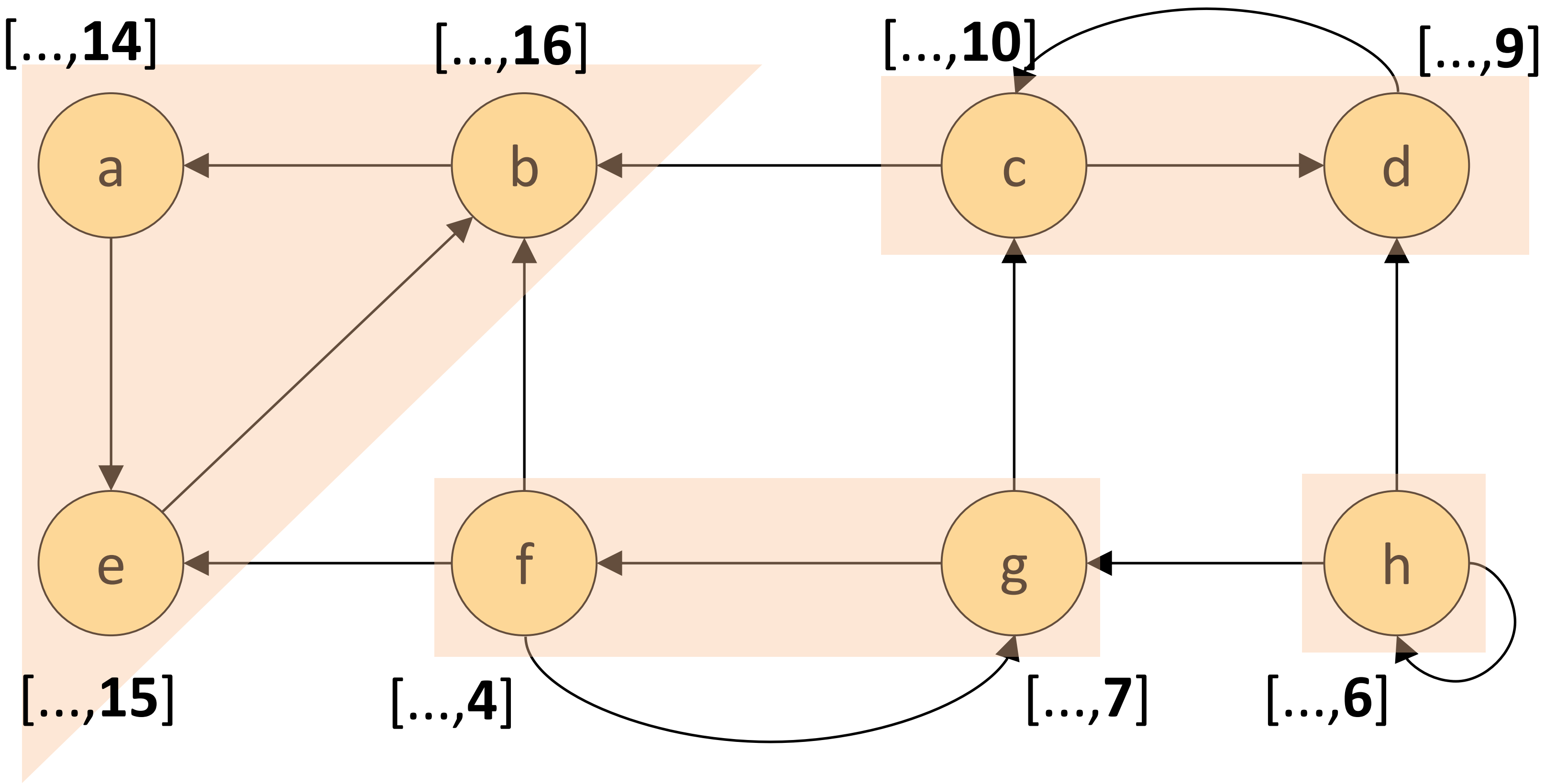
Componentes furtemente conectadas: grafos direccionales con ciclos

1. Hacemos un recorrido DFS de *G*, anotando los tiempos ***end*** de cada nodo



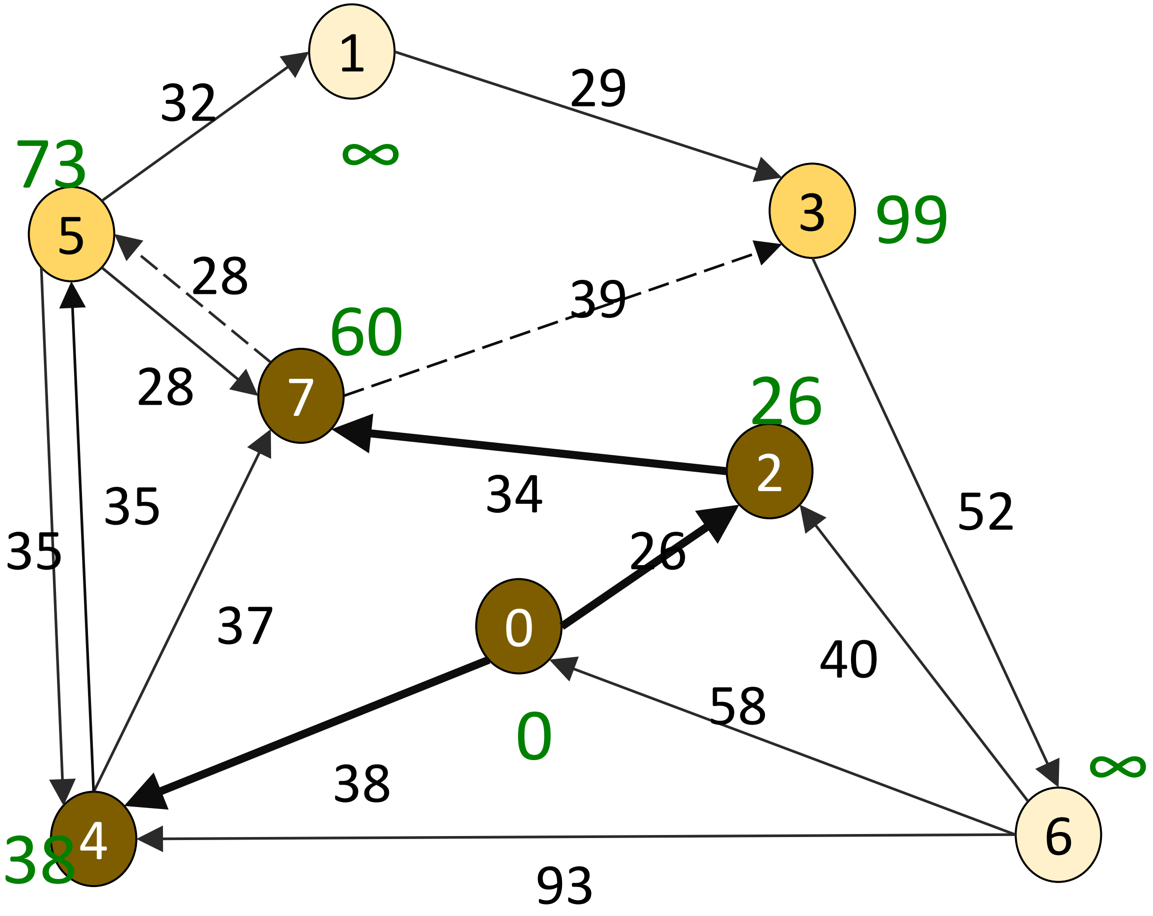
CFCs: Calculamos *G'*

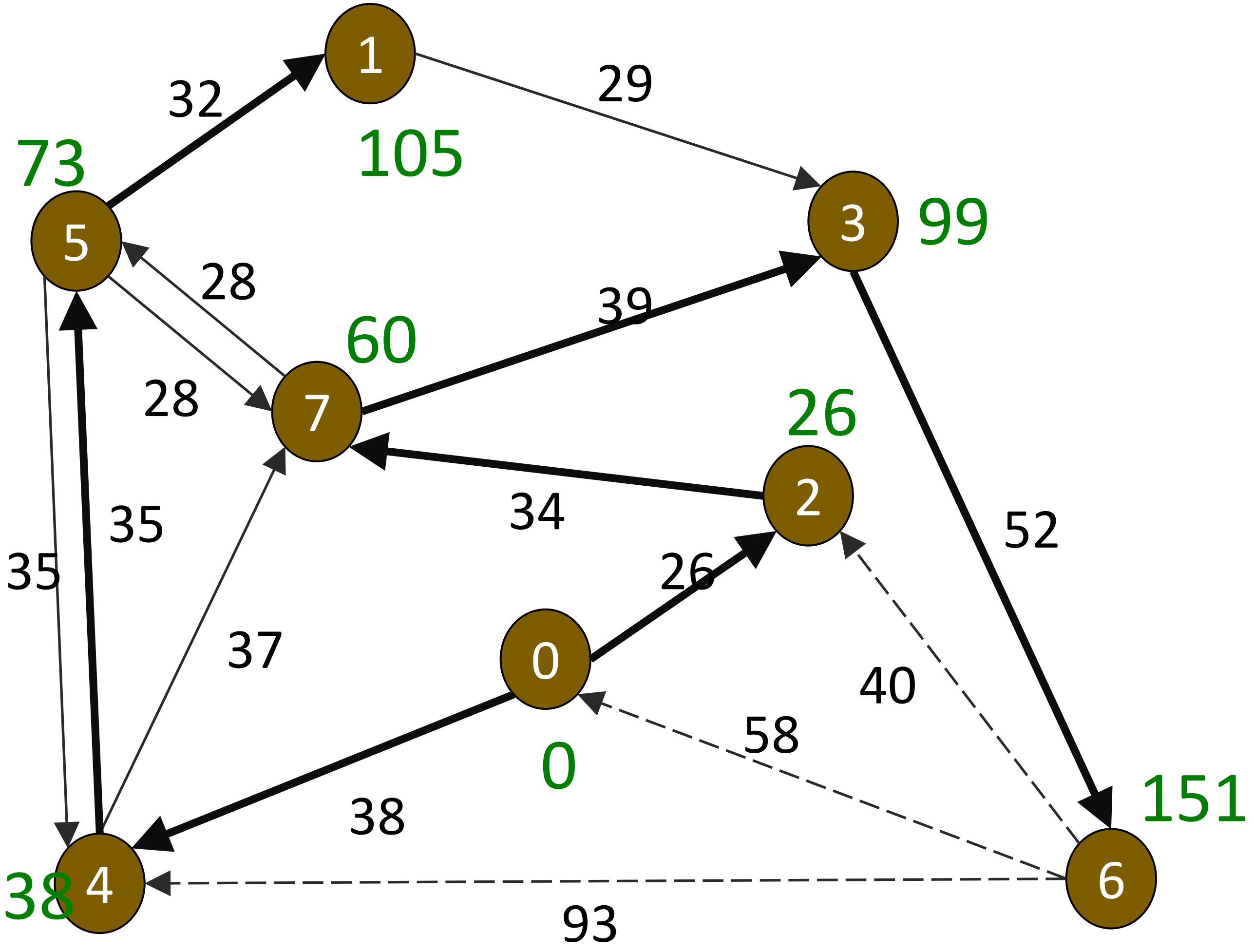
2. DFS sobre *G'*, pero en orden decreciente de tiempos ***end*** (del recorrido anterior)



Dijkstra: grafos con costos no negativos

```
Dijkstra(s): —s es el vértice de partida
    for each u in V:
         u.color ← white; d[u] ← ∞; π[u] ← null
    Q ← cola de prioridades
    s.color ← gray; d[s] ← 0; Q.enqueue(s)
    while !Q.empty():
        u ← Q.dequeue()
        for each v in α[u]:
            if v.color == white or v.color == gray:
                if d[v] > d[u] + costo(u,v):
                    d[v] ← d[u] + costo(u,v); π[v] ← u
            if v.color == white:
                    v.color ← gray; Q.enqueue(v)
        u.color ← black
```


1
∞
32
29
73
5
3
99
28
39
28
60
7
26
2
35
34
35
26
52
0
37
40
58
0
38
38
4
6
∞
93


1
32
29
105
73
5
3
99
28
39
60
28
7
26
2
34
35
35
26
52
0
37
40
58
0
38
151
38
4
6
93