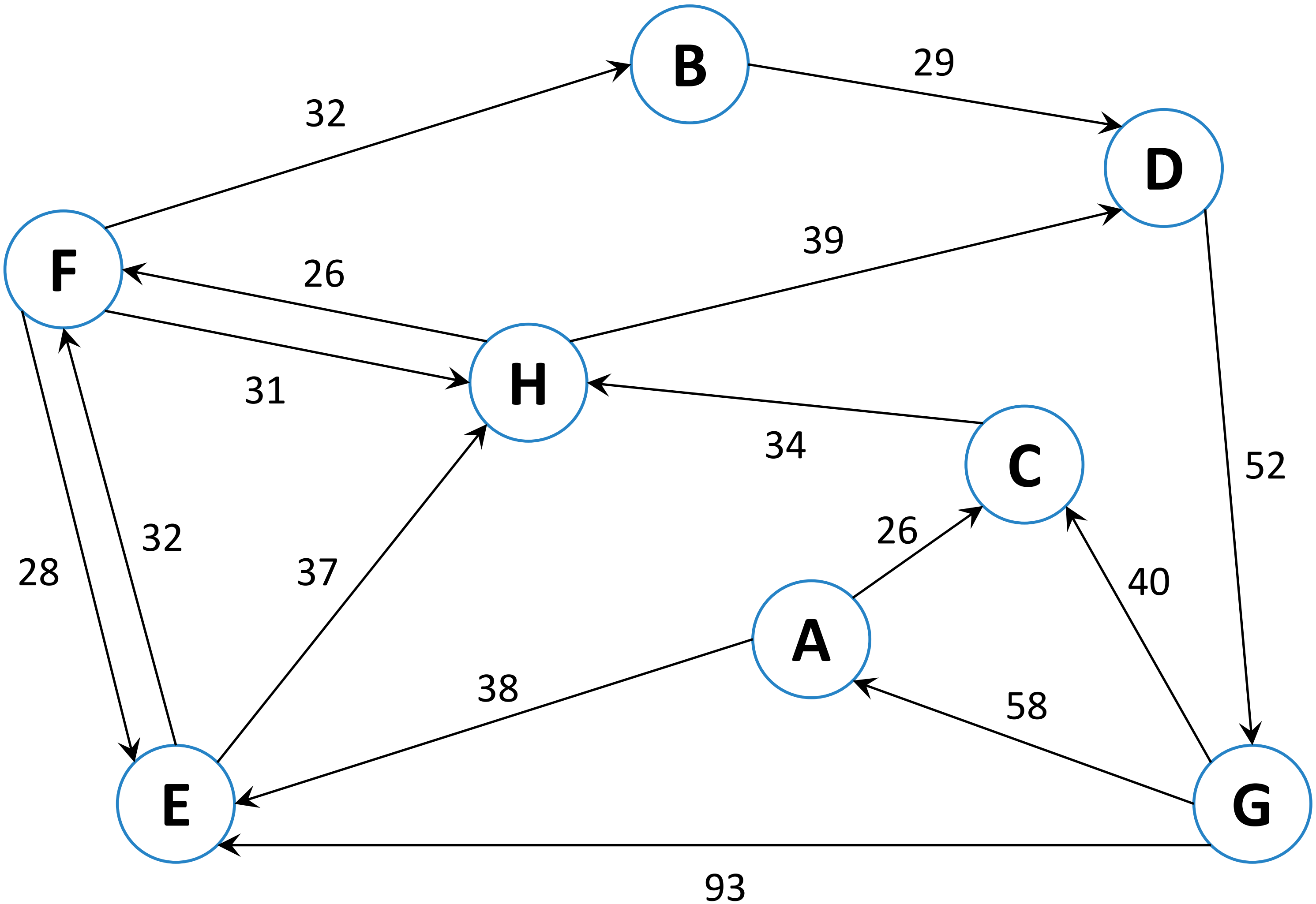
El viaje familiar

- Quieres planificar un viaje en auto desde A a B
- Los caminos tienen peajes y tiempos de recorrido

¿Cómo hacer para que el viaje te salga lo más barato posible?

¿Y lo más corto posible?

Grafo direccional con costos

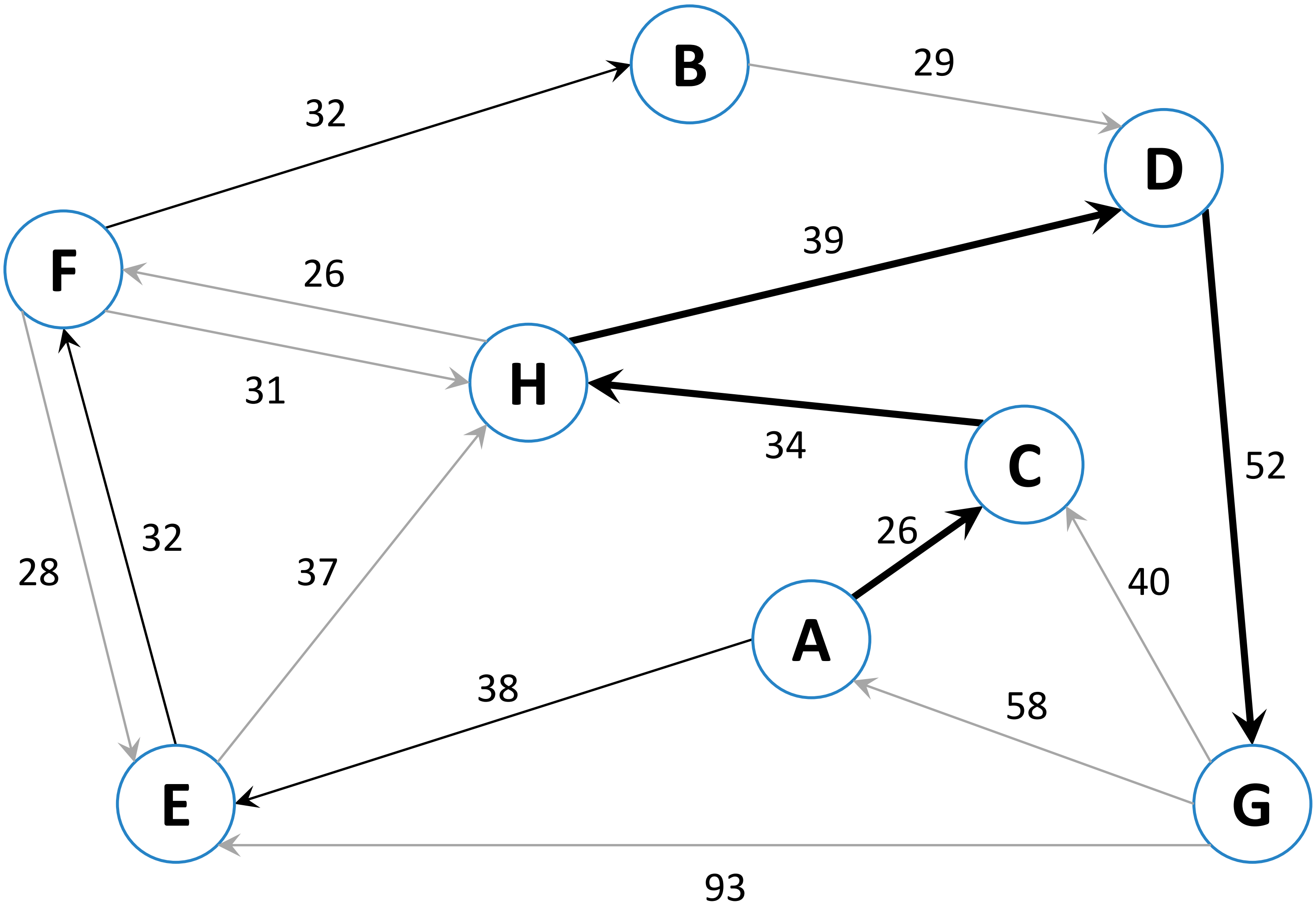


Rutas más baratas (o más cortas)

Debemos buscar la **ruta más barata** de A a B, es decir,

… la **suma de los costos** de sus aristas debe ser mínima

P.ej., ruta más cortas de *A* a *G*

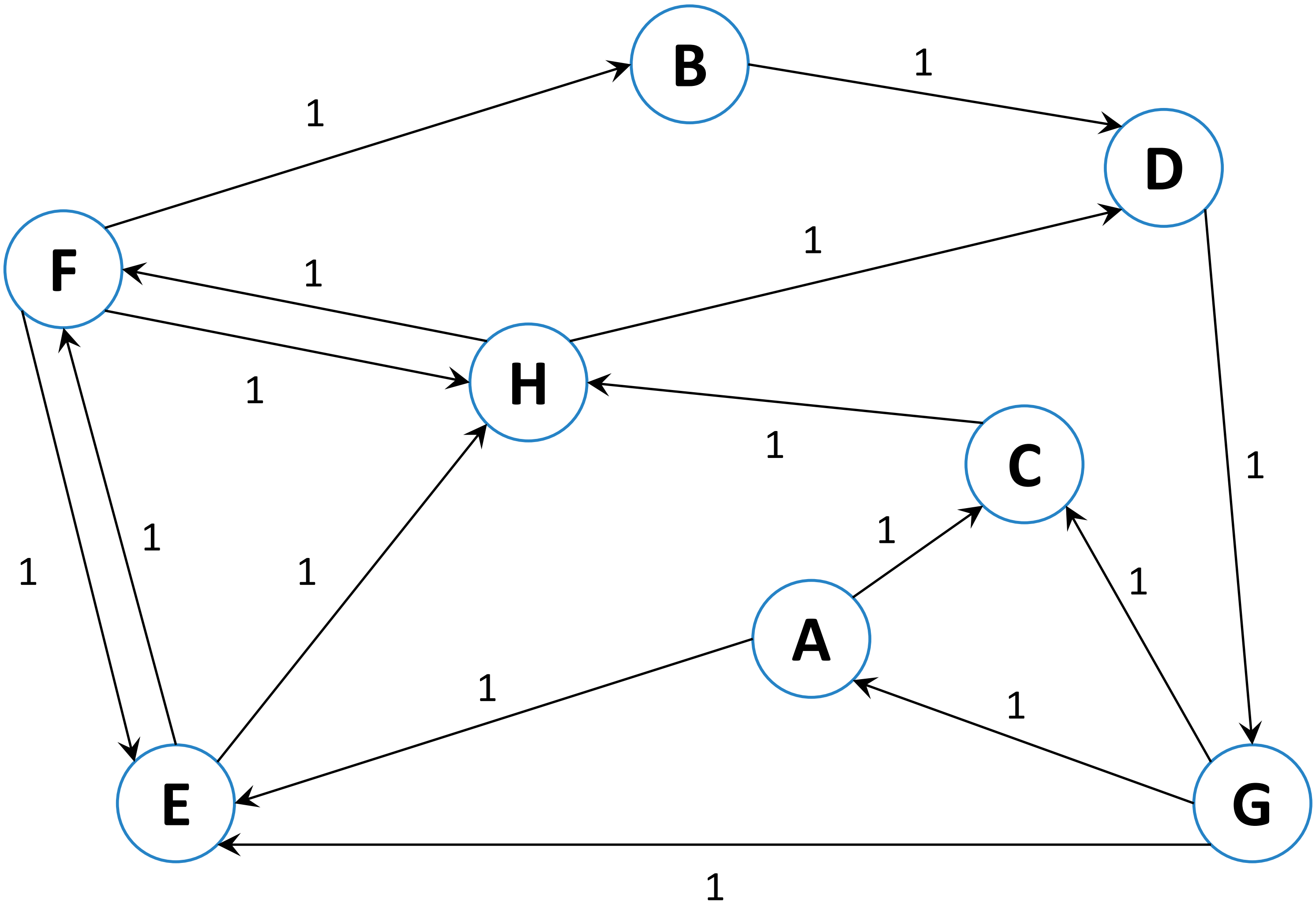


Primero, tratemos de resolver una versión simplificada del problema

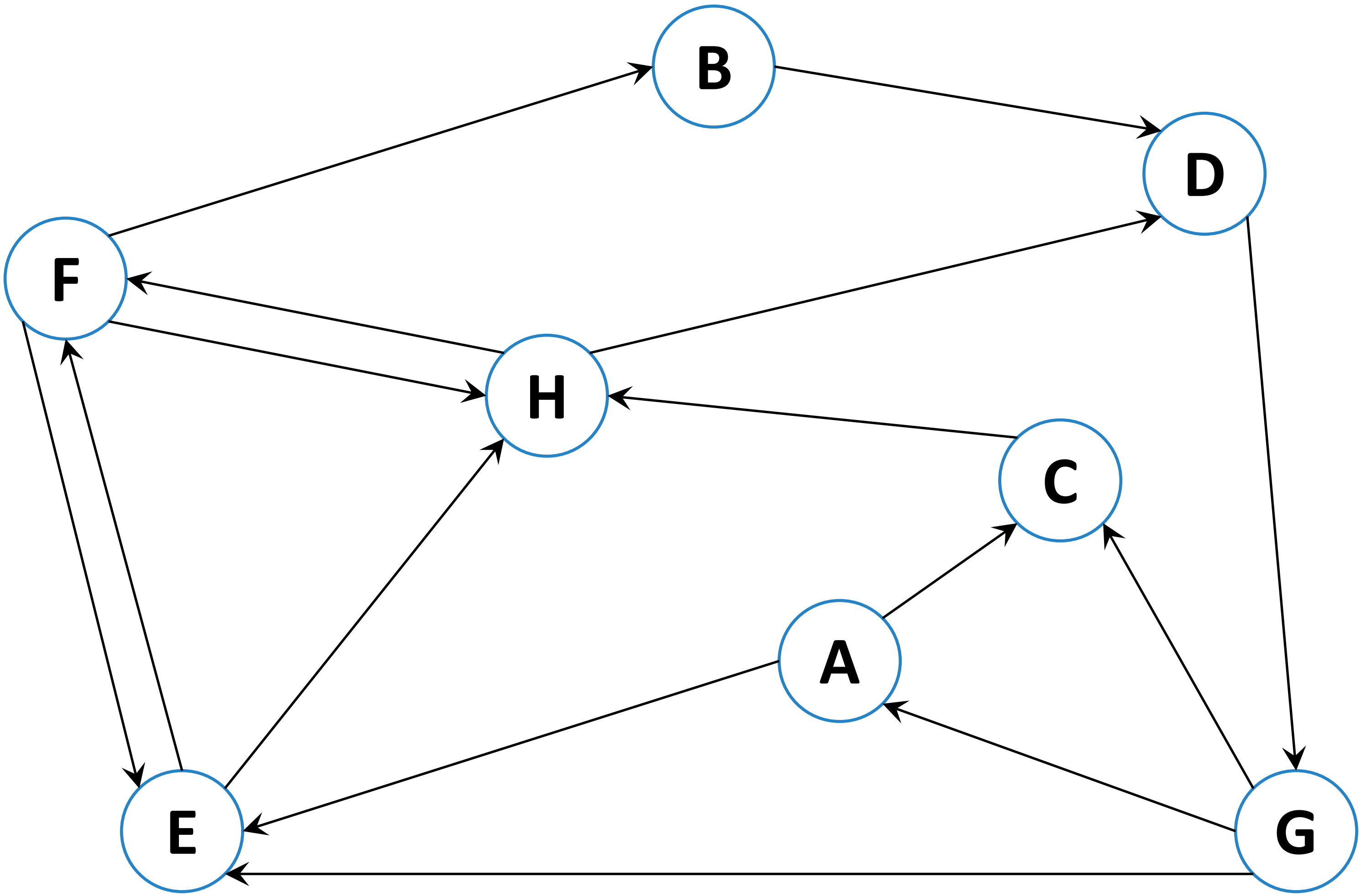
Supongamos que todas las aristas tienen el mismo costo

… entonces la ruta más corta de A a B es la ruta …

Grafo direccional en que todas las aristas tienen el mismo costo

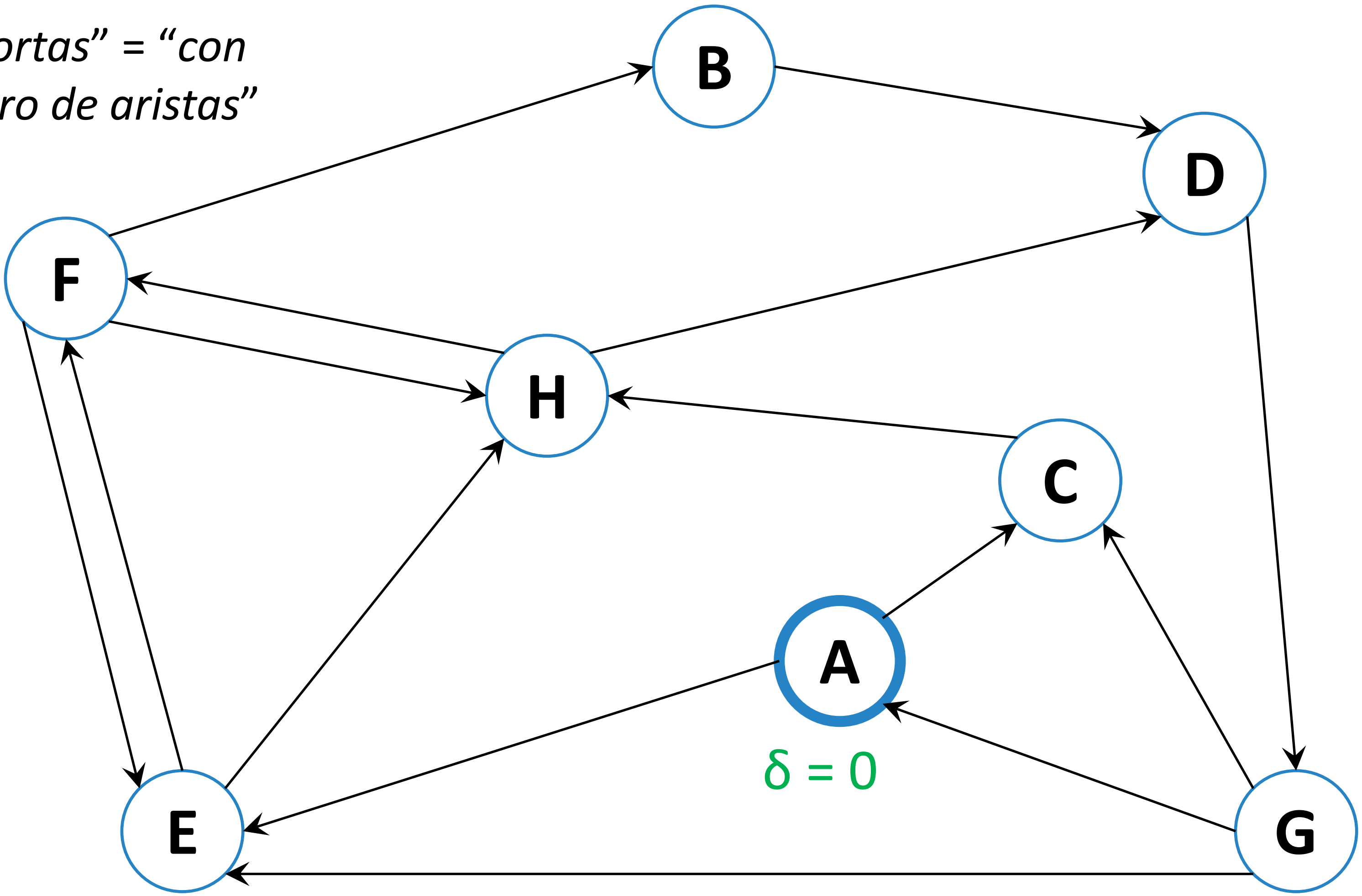


Entonces, las rutas más cortas son las rutas con **el menor número de aristas**



BFS: algoritmo para determinar las rutas más cortas a partir de un determinado vértice

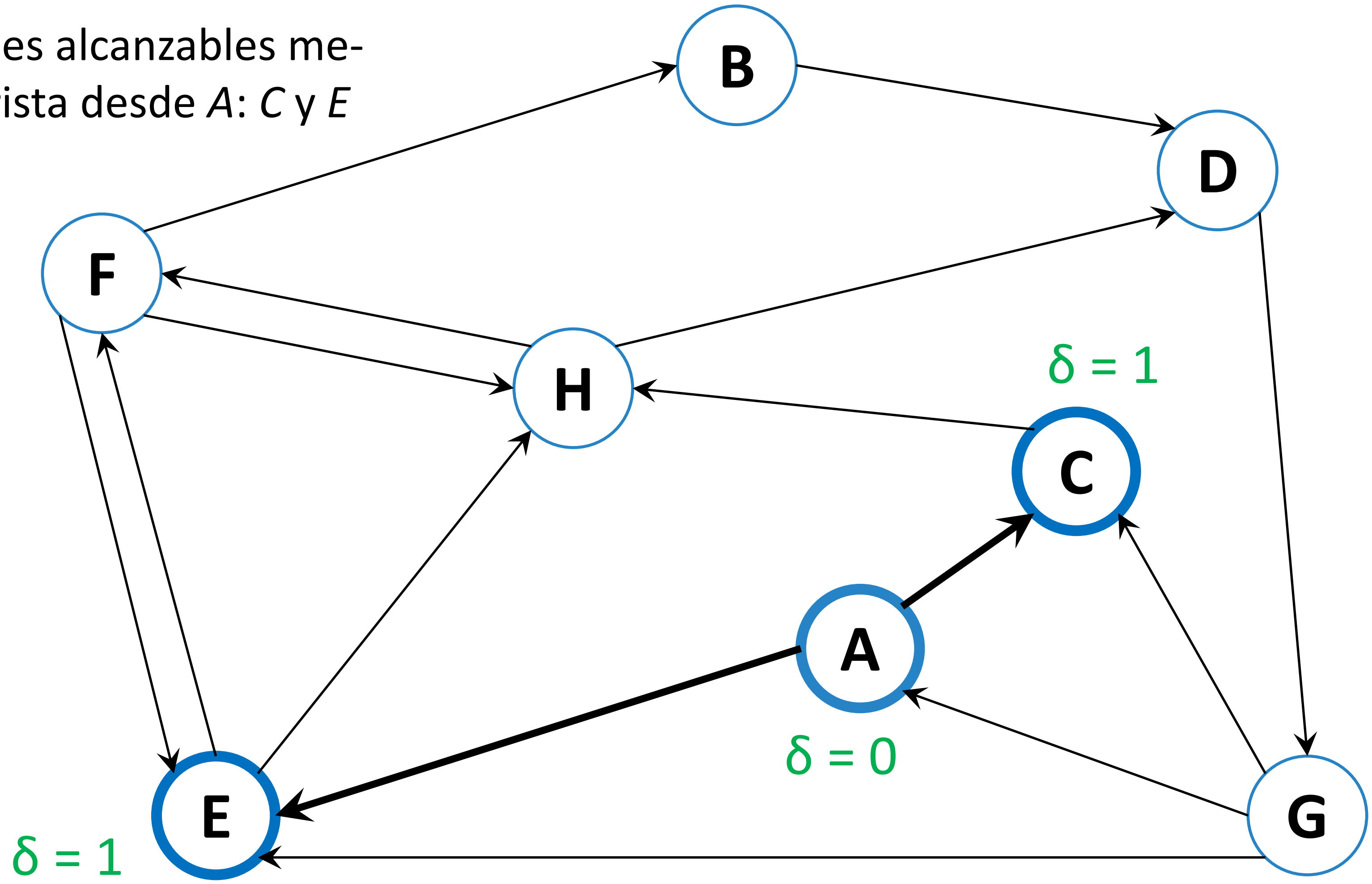
En BFS, “*más cortas*” = “*con el menor número de aristas*”



Partimos de *A*; notamos que *A* está a distancia δ = 0 de *A*

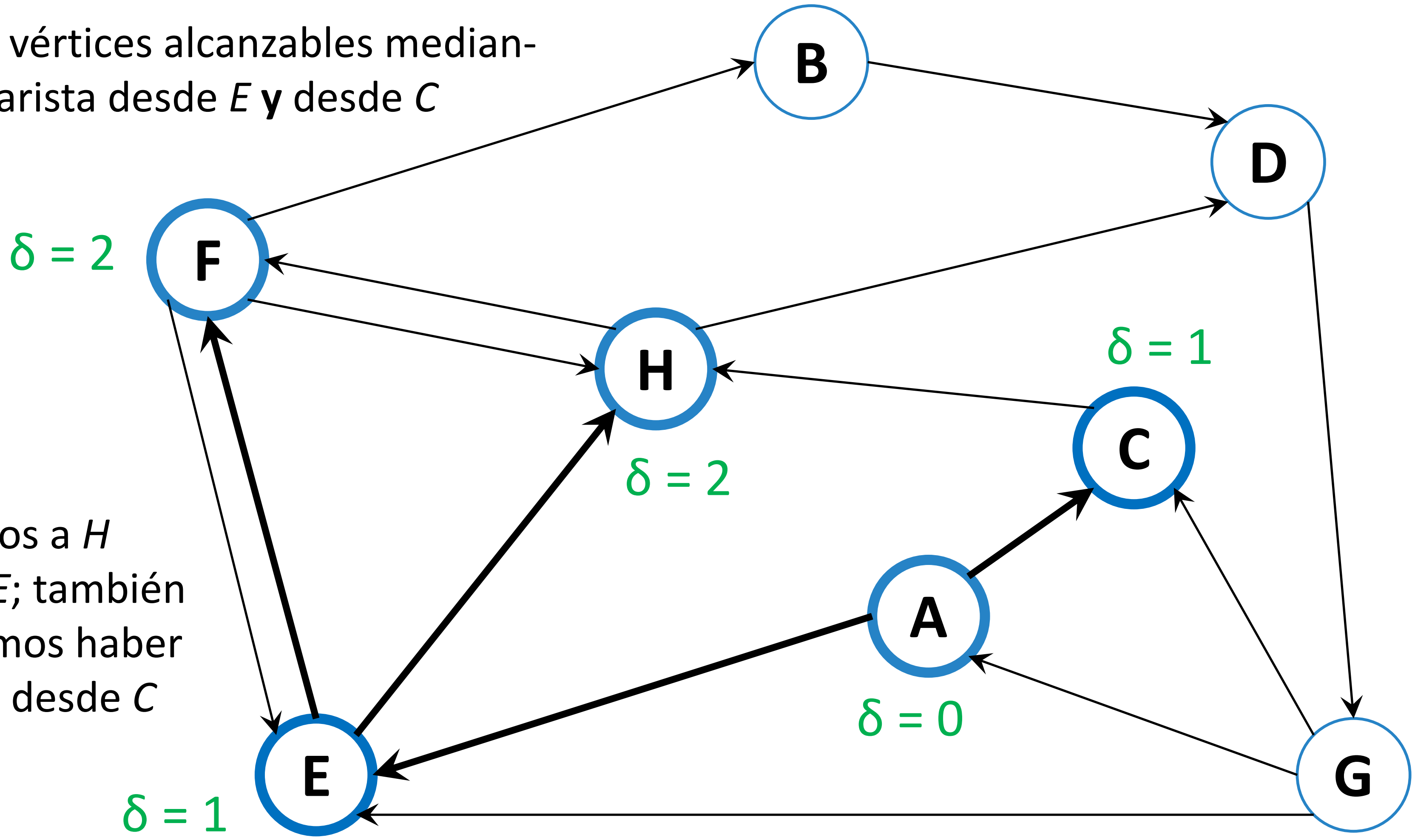
A continuación buscamos **todos** los vértices que estén a distancia δ = 1 de *A*

Son los vértices alcanzables mediante una arista desde *A*: *C* y *E*



Luego buscamos **todos** los vértices que estén a distancia δ = 2 de *A*

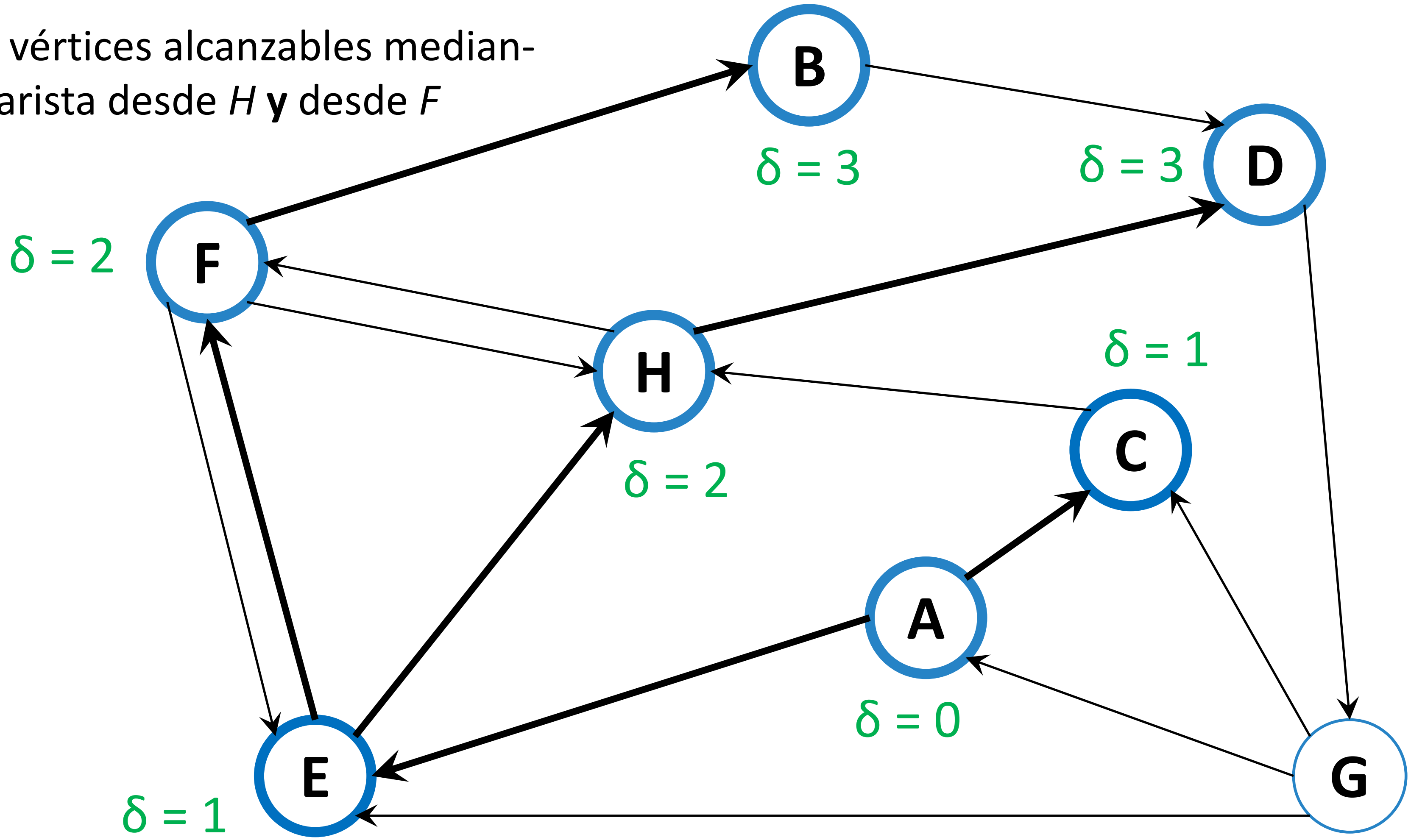
Son los vértices alcanzables median-
te una arista desde *E* **y** desde *C*



Llegamos a *H* desde *E*; también podríamos haber llegado desde *C*

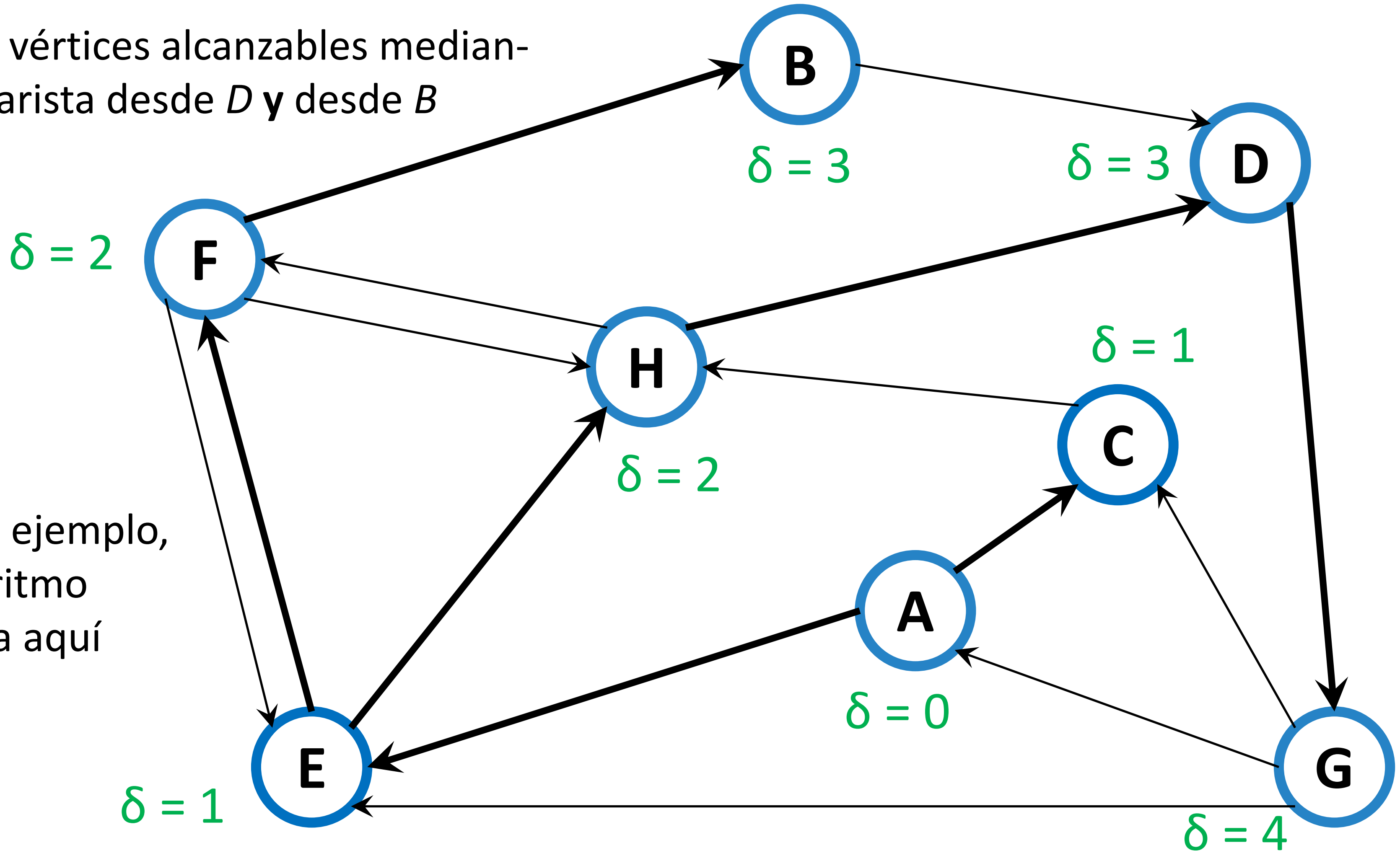
Ahora buscamos **todos** los vértices que estén a distancia δ = 3 de *A*

Son los vértices alcanzables mediante una arista desde *H* **y** desde *F*



Y ahora buscamos **todos** los vértices que estén a distancia δ = 4 de *A*

Son los vértices alcanzables median-te una arista desde *D* **y** desde *B*



En este ejemplo, el algoritmo termina aquí

Propiedad de BFS

BFS nos asegura que cuando llegamos por primera vez a (descubrimos) un vértice

… llegamos a través del **menor número de aristas**

¿Por qué?

Primero llegamos a **todos** los vértices que están a una distancia $\delta = k$ aristas

… antes de llegar a cualquier vértice que esté a una distancia $\delta = k+1$ aristas

Implementación de BFS

Hay que distinguir los vértices descubiertos de los vértices aún no descubiertos:

- usamos dos colores
- si además queremos distinguir los vértices descubiertos que aún tienen vértices vecinos por descubrir, de aquellos para los cuales ya descubrimos a todos sus vecinos, entonces usamos tres colores

Hay que almacenar los vértices recién descubiertos de manera de revisar todas las aristas que salen de un vértice a distancia $\delta = k$ antes de revisar cualquier arista que salga de un vértice a distancia $\delta = k+1$:

- cuando descubrimos un vértice, lo ponemos en una cola
- cuando tomamos un vértice para revisar sus aristas, lo sacamos de la cola

BFS en pseudo código

```
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
```

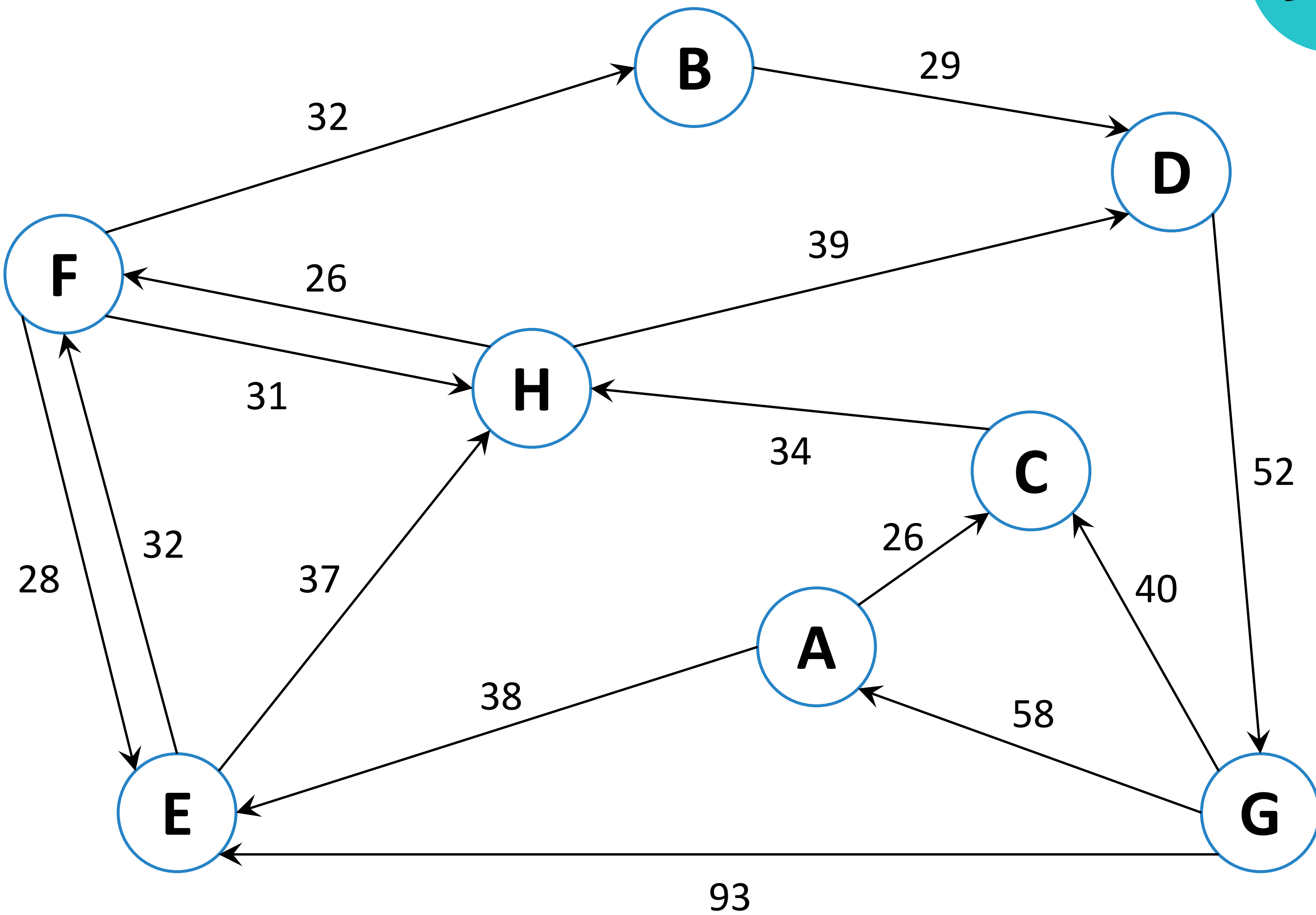
Propiedades de BFS

¿Cuál es la propiedad de **BFS**?

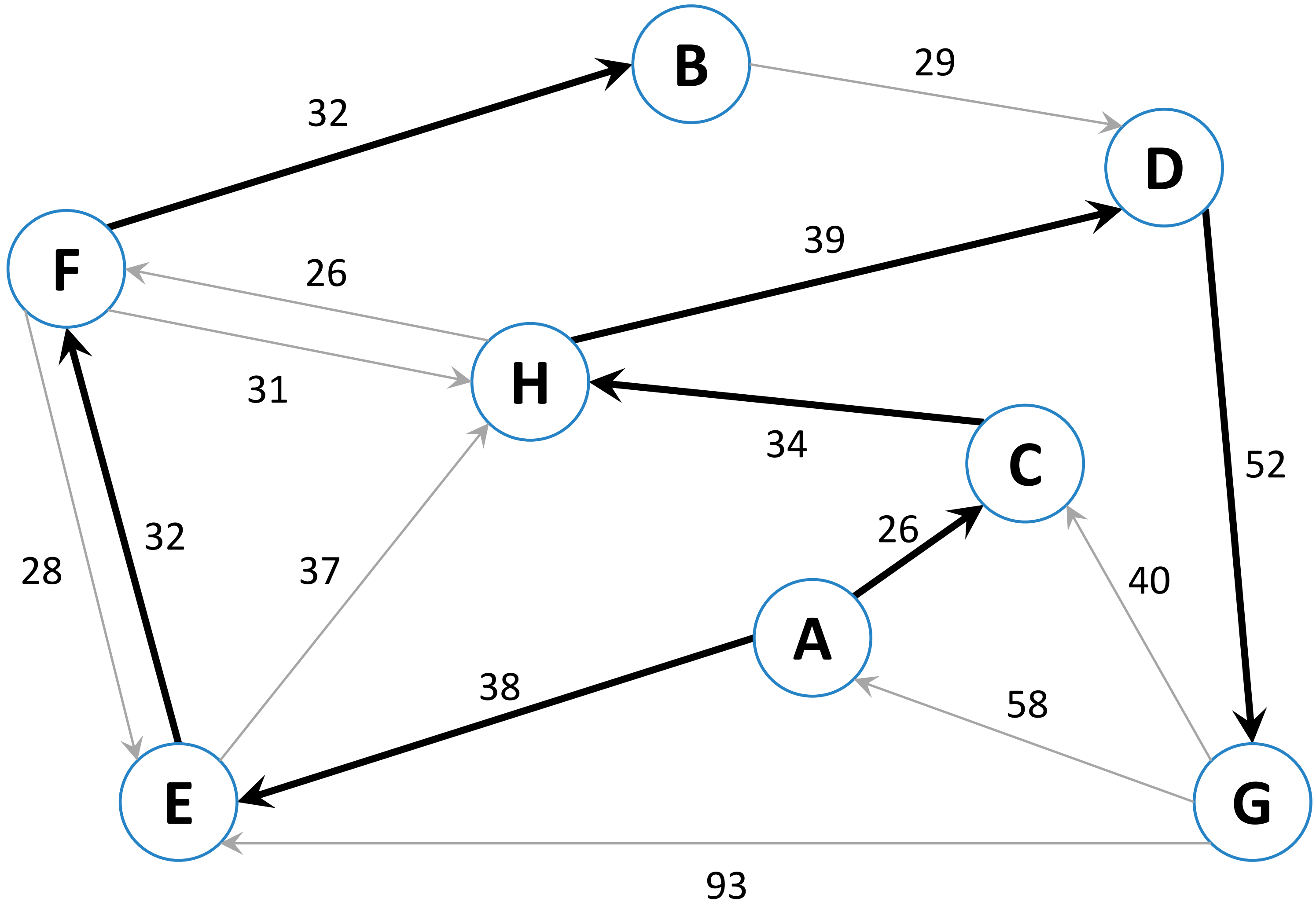
Si **marcamos** cada arista que queda como padre de un nodo,

… ¿qué **representa** el conjunto de aristas marcadas?

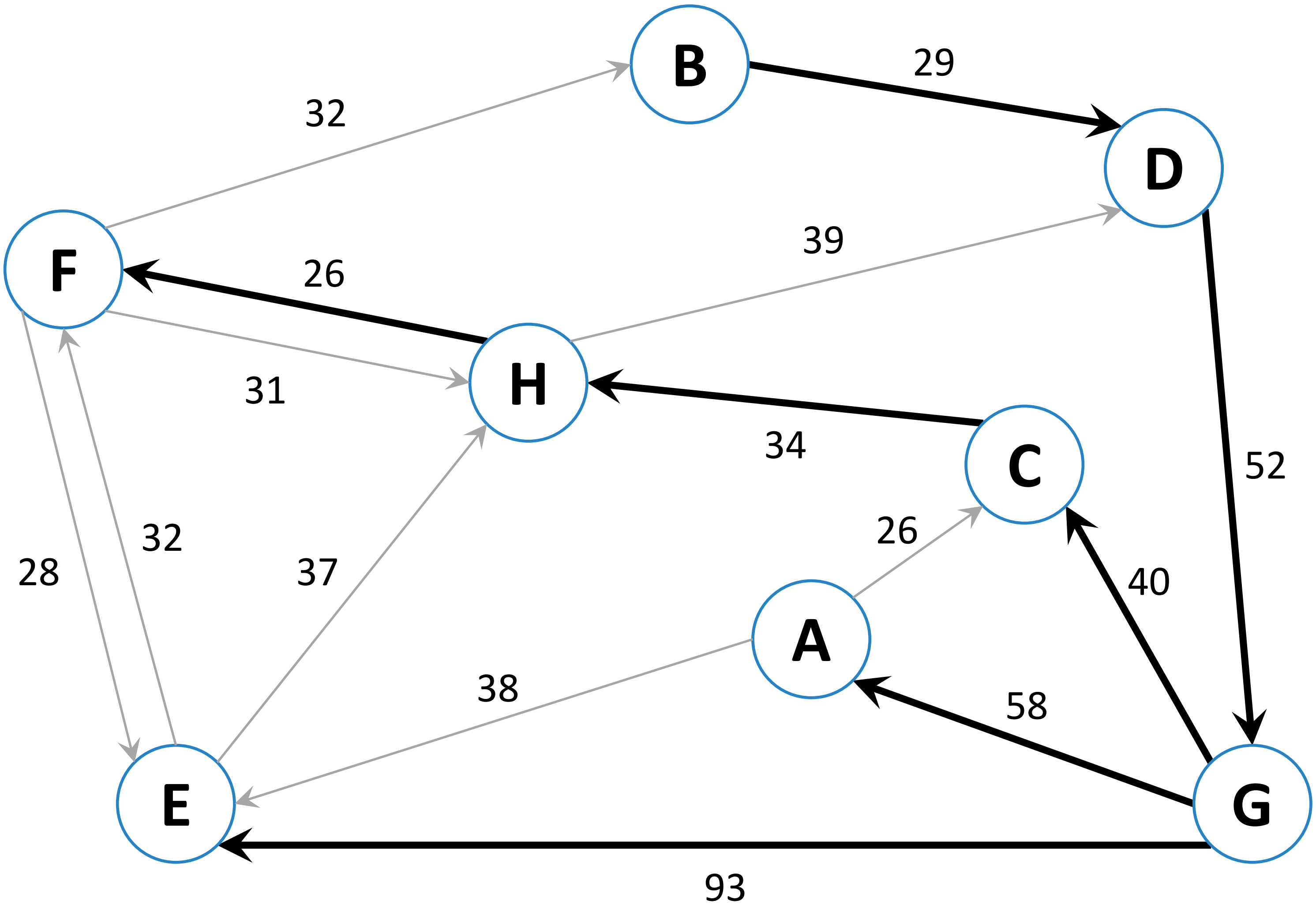
Volvamos a nuestro problema original: rutas más cortas en un grafo con costos



Árbol de rutas más cortas desde *A*



Árbol de rutas más cortas desde *B*



Propiedades del problema de rutas más cortas

Las rutas son direccionales

Los costos pueden representar distancias, tiempos de viajes, consumo de combustible, costos de peajes, etc.

Puede que haya vértices inalcanzables desde el vértice de partida

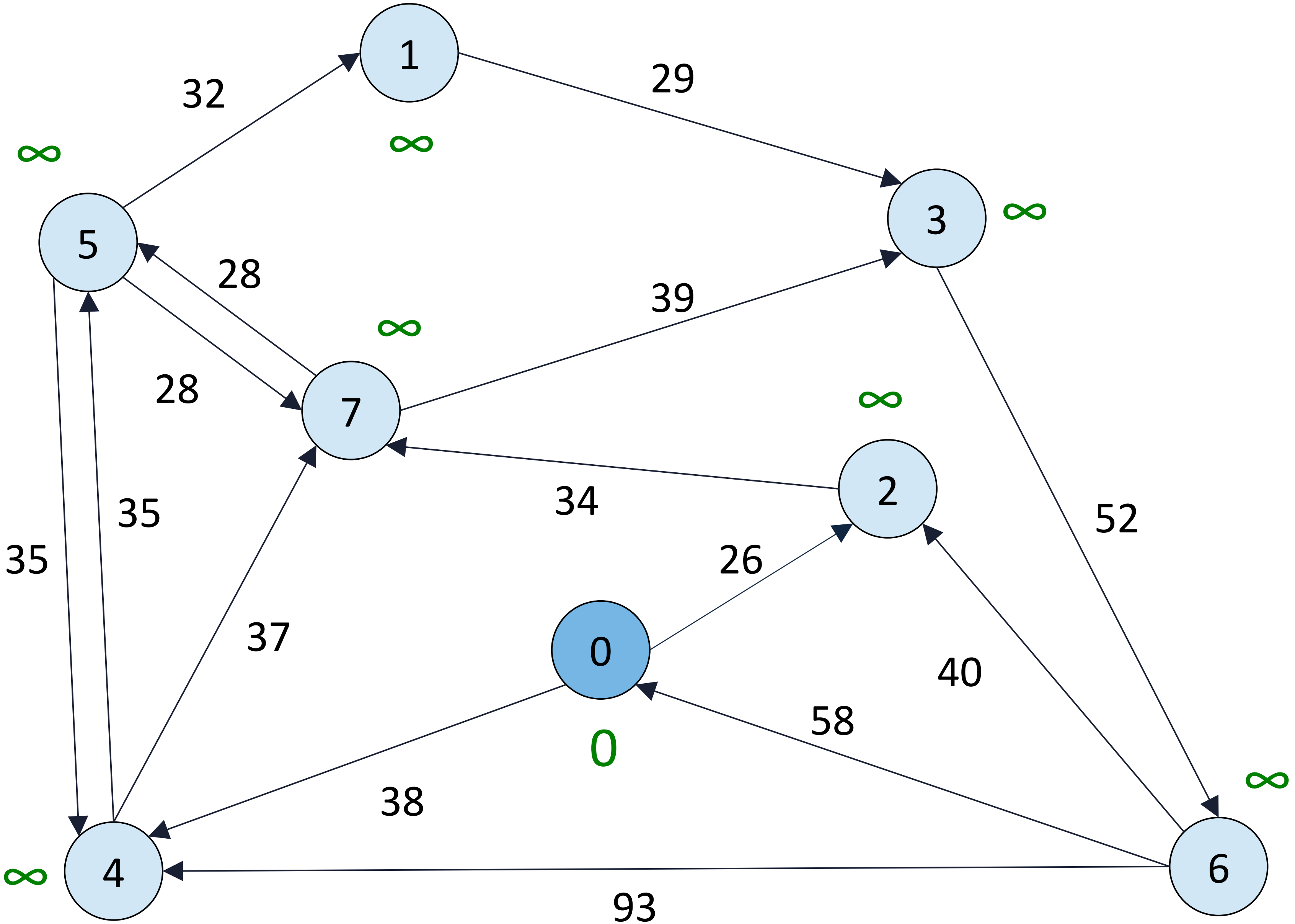
Si hay costos negativos, es más complicado resolver el problema

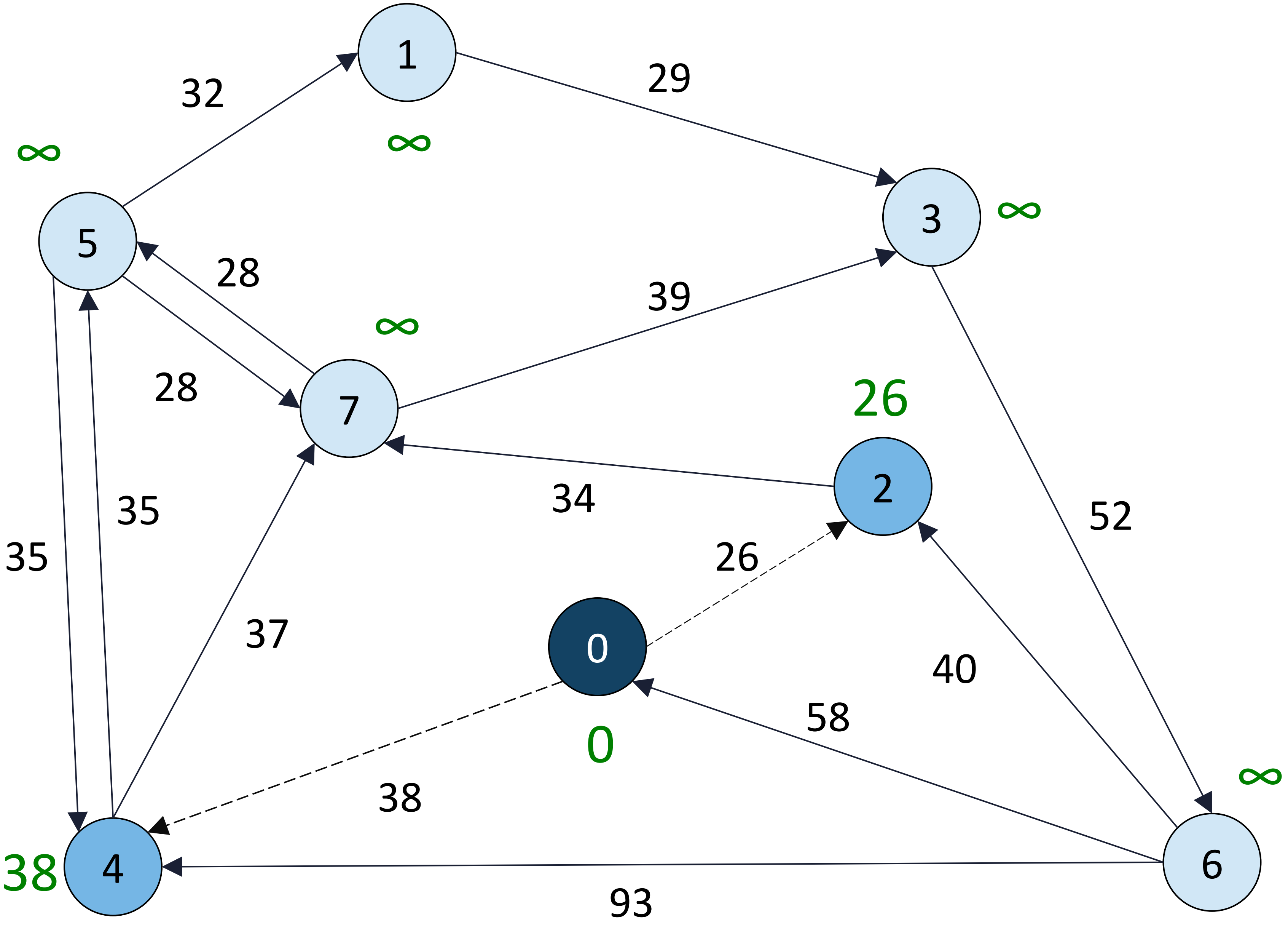
Las rutas más cortas pueden no ser únicas

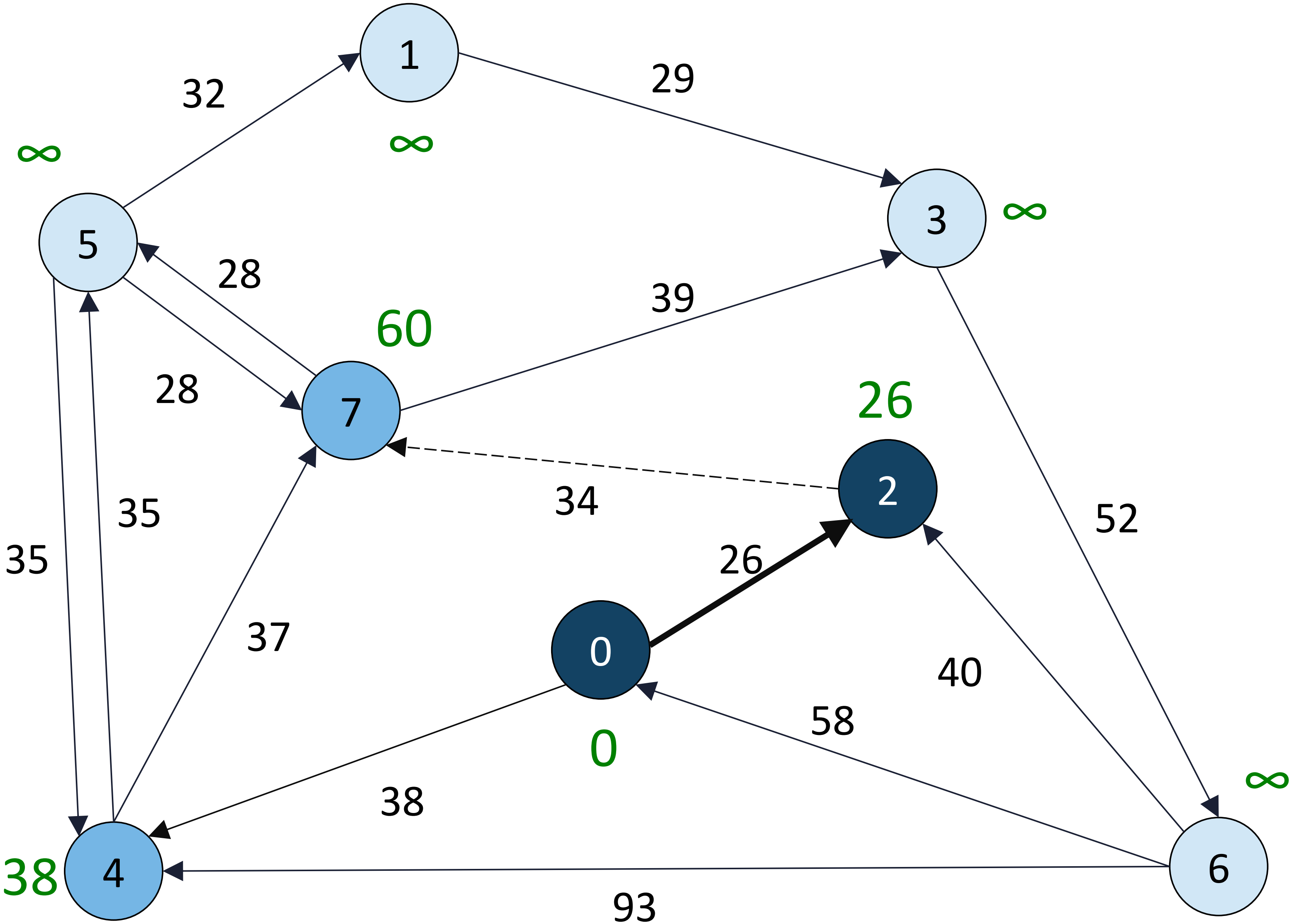
"BFS++"

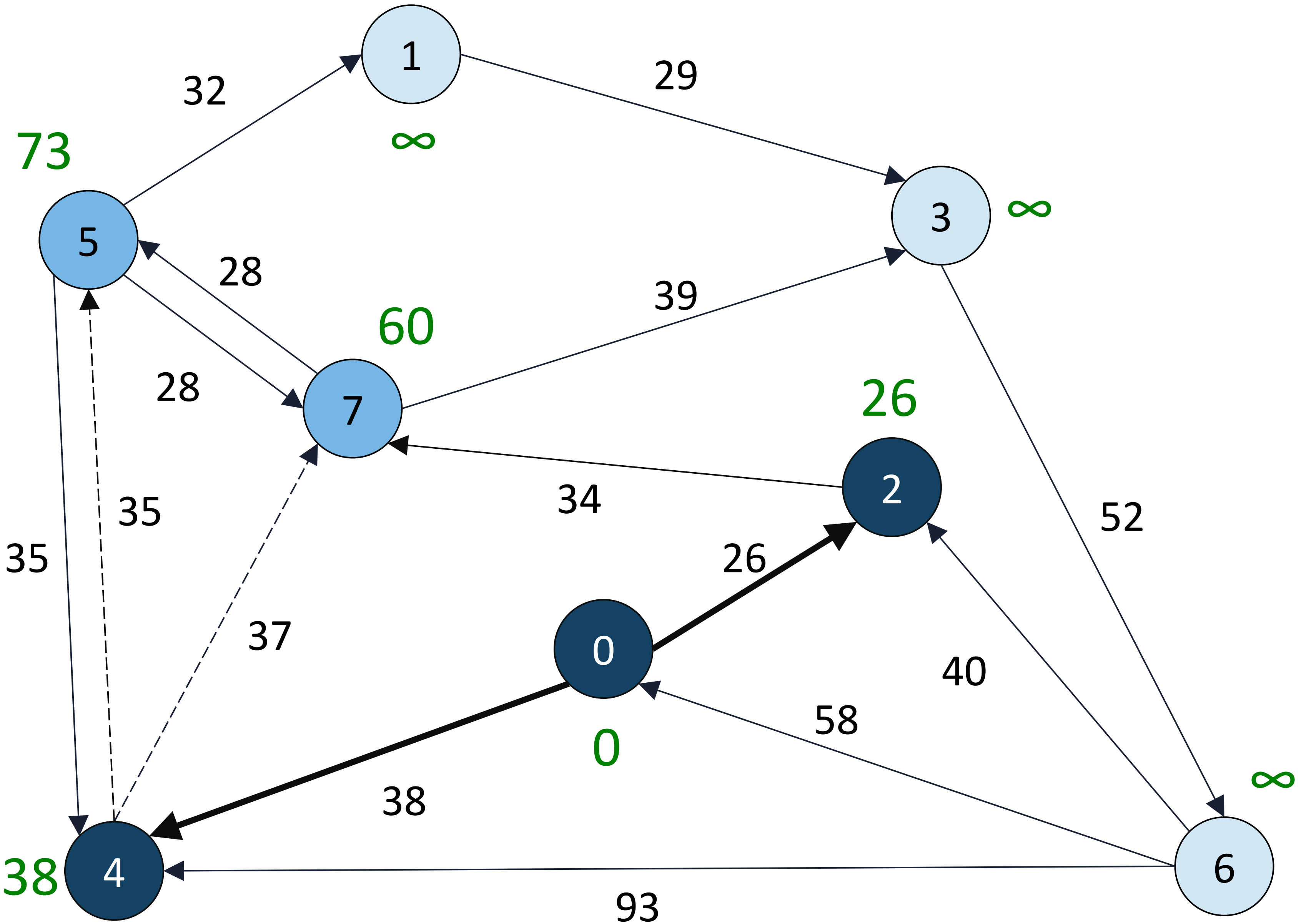
¿Cómo podemos extender **BFS** para este problema?

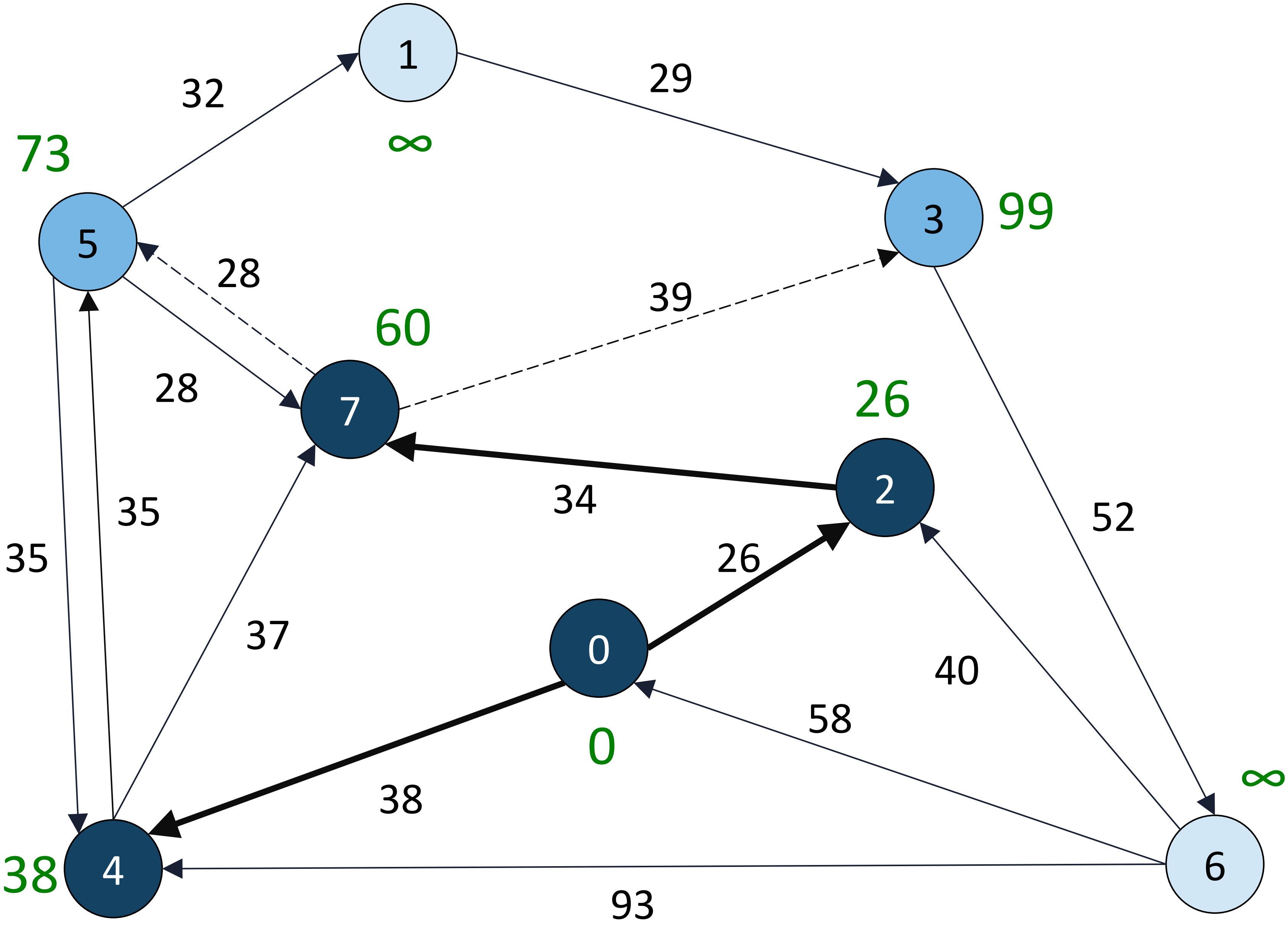
Queremos **garantizar** que al sacar un nodo de la cola $\mathbf{Q}$**,** hemos encontrado ese nodo por la ruta más corta: de menor costo acumulado (y no necesariamente con el menor número de aristas)

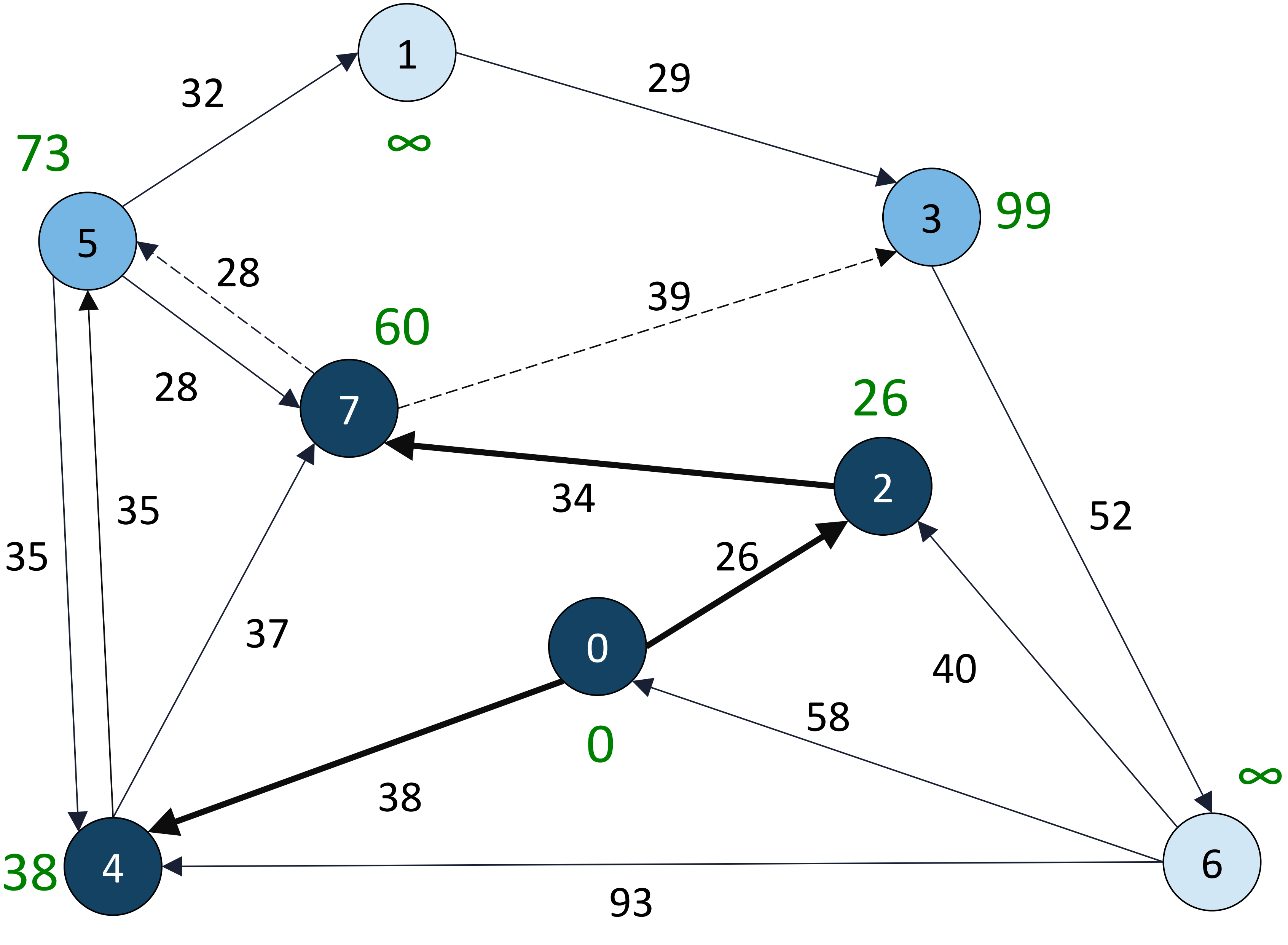

1
2
3
4
5
6
7
0
32
29
28
28
39
34
35
35
37
26
40
52
58
38
93
∞
0

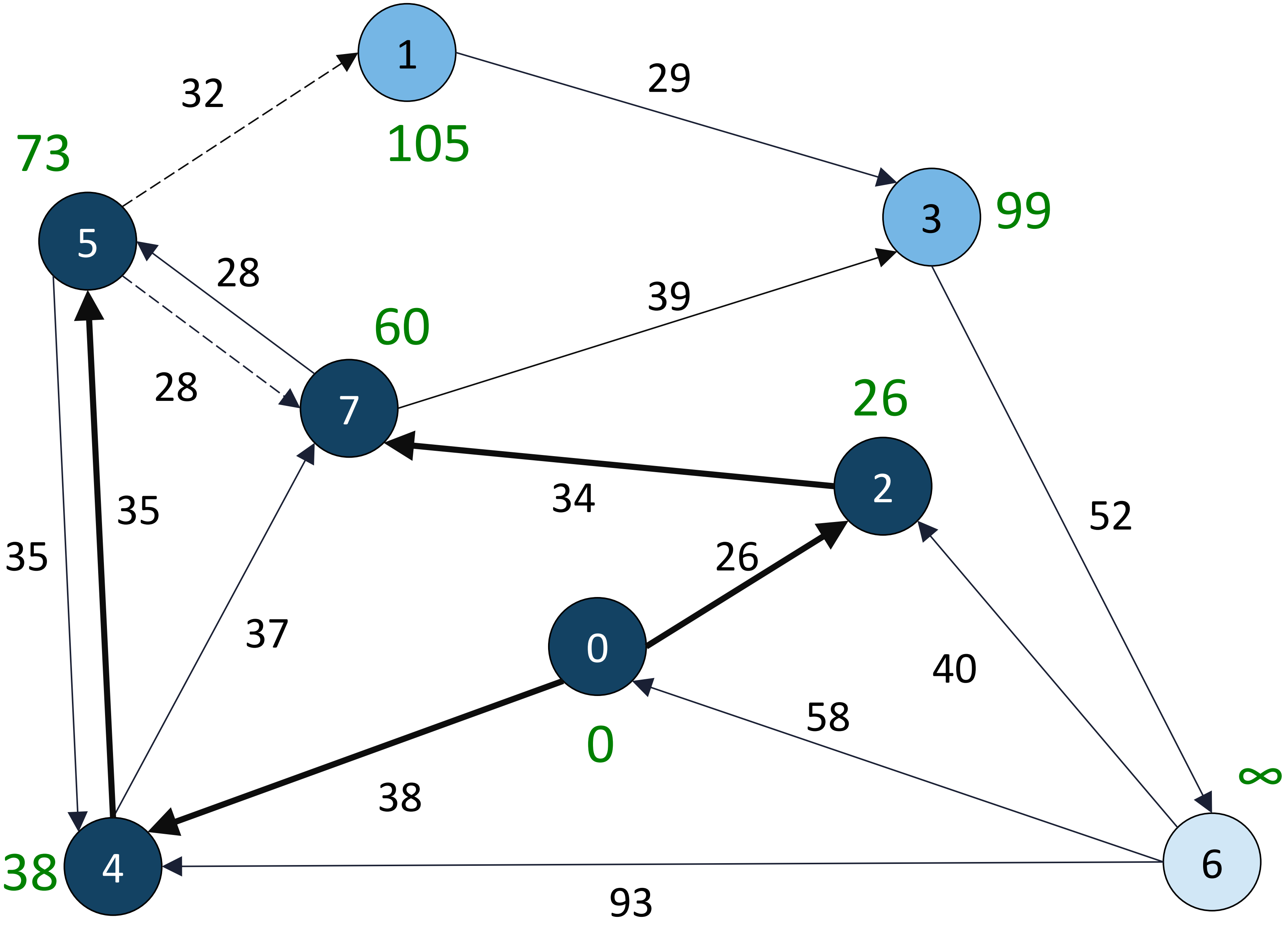

1
29
32
∞
∞
5
3
∞
28
39
∞
28
7
26
2
34
35
52
35
26
37
0
40
58
0
∞
38
38
4
6
93


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

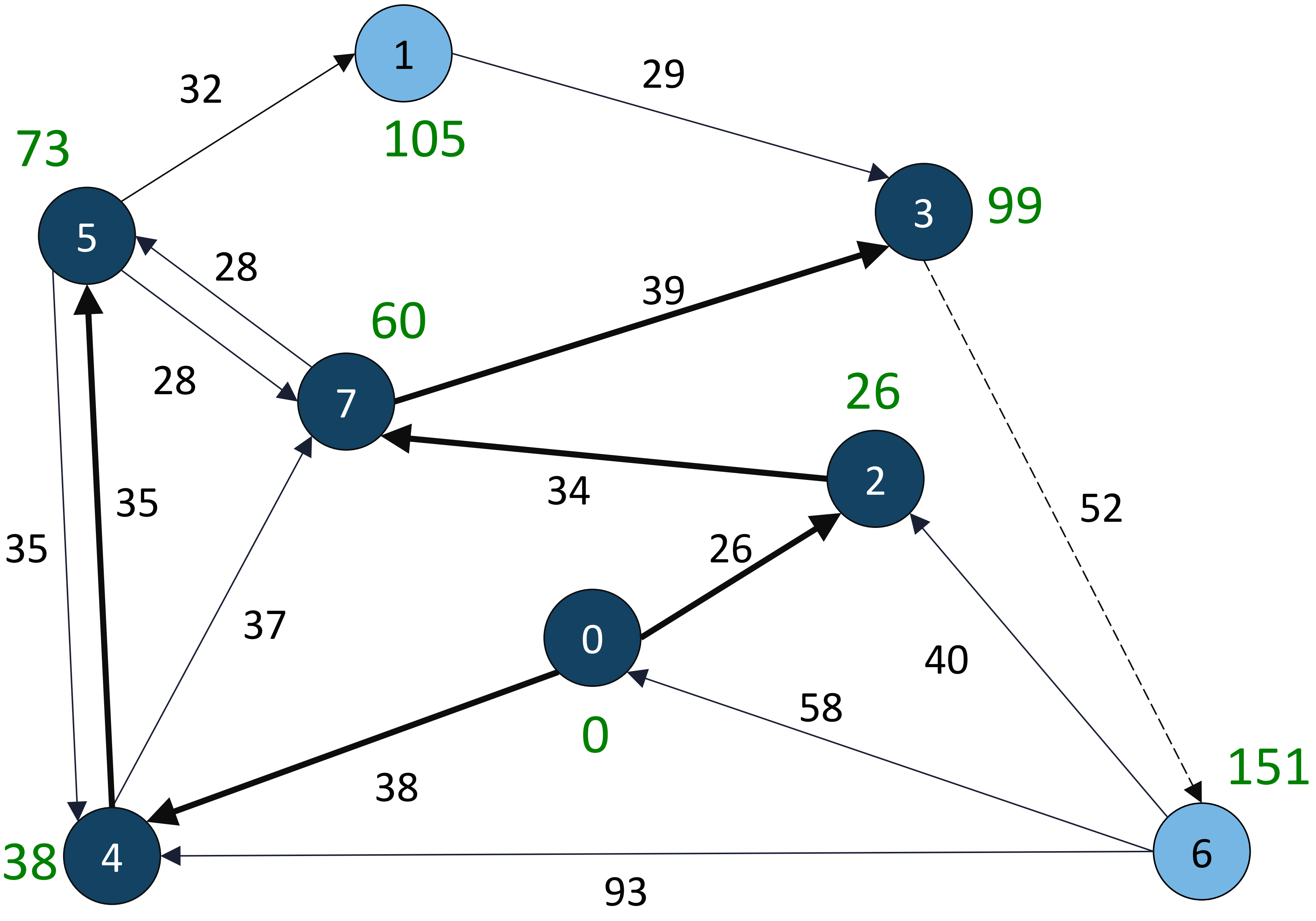

1
29
32
∞
73
5
3
∞
28
39
60
28
7
26
2
34
35
52
35
26
37
0
40
58
0
∞
38
38
4
6
93

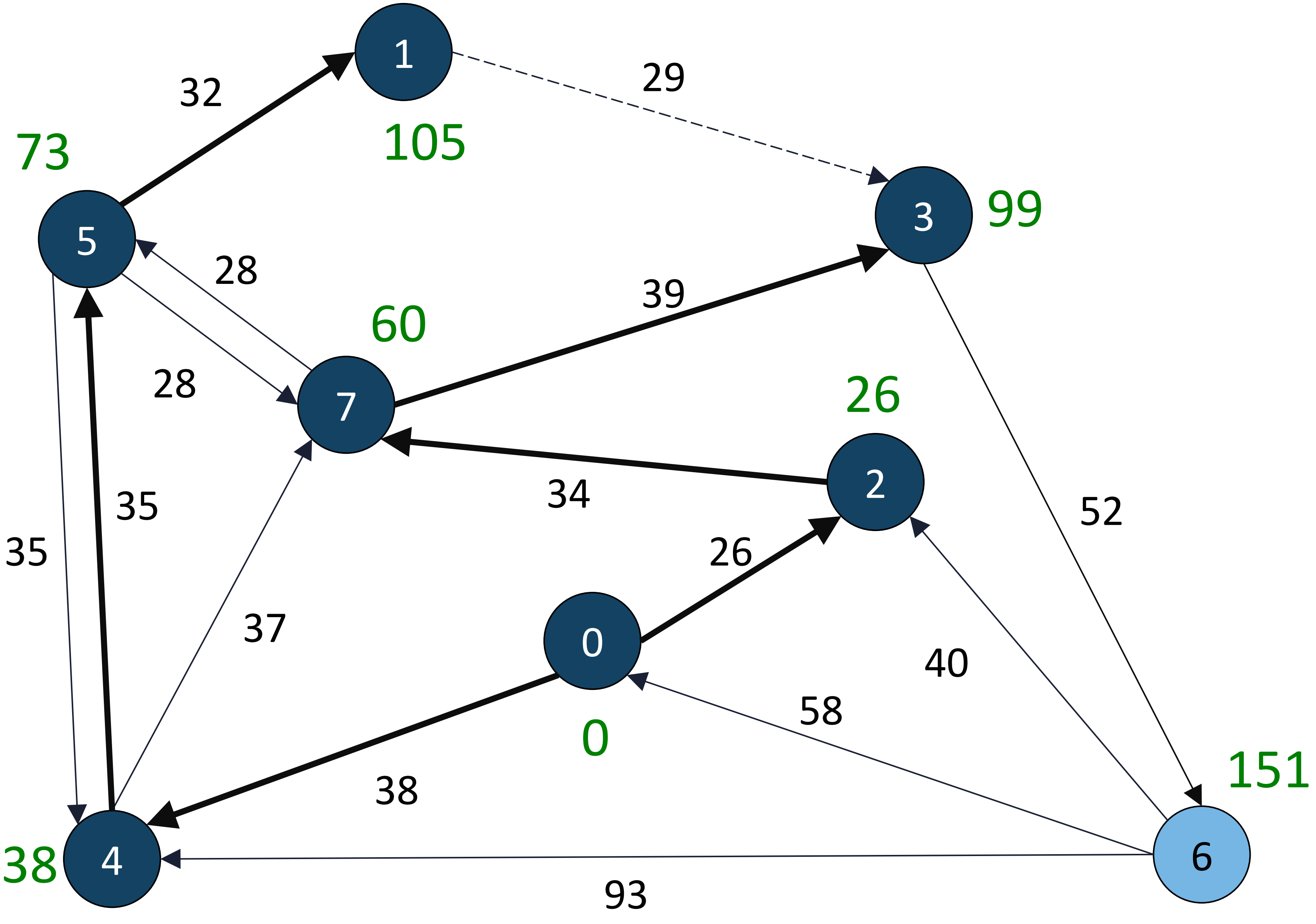

1
29
32
∞
73
5
3
99
28
39
60
7
28
26
2
34
35
35
26
37
0
40
58
0
∞
38
38
4
6
93


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

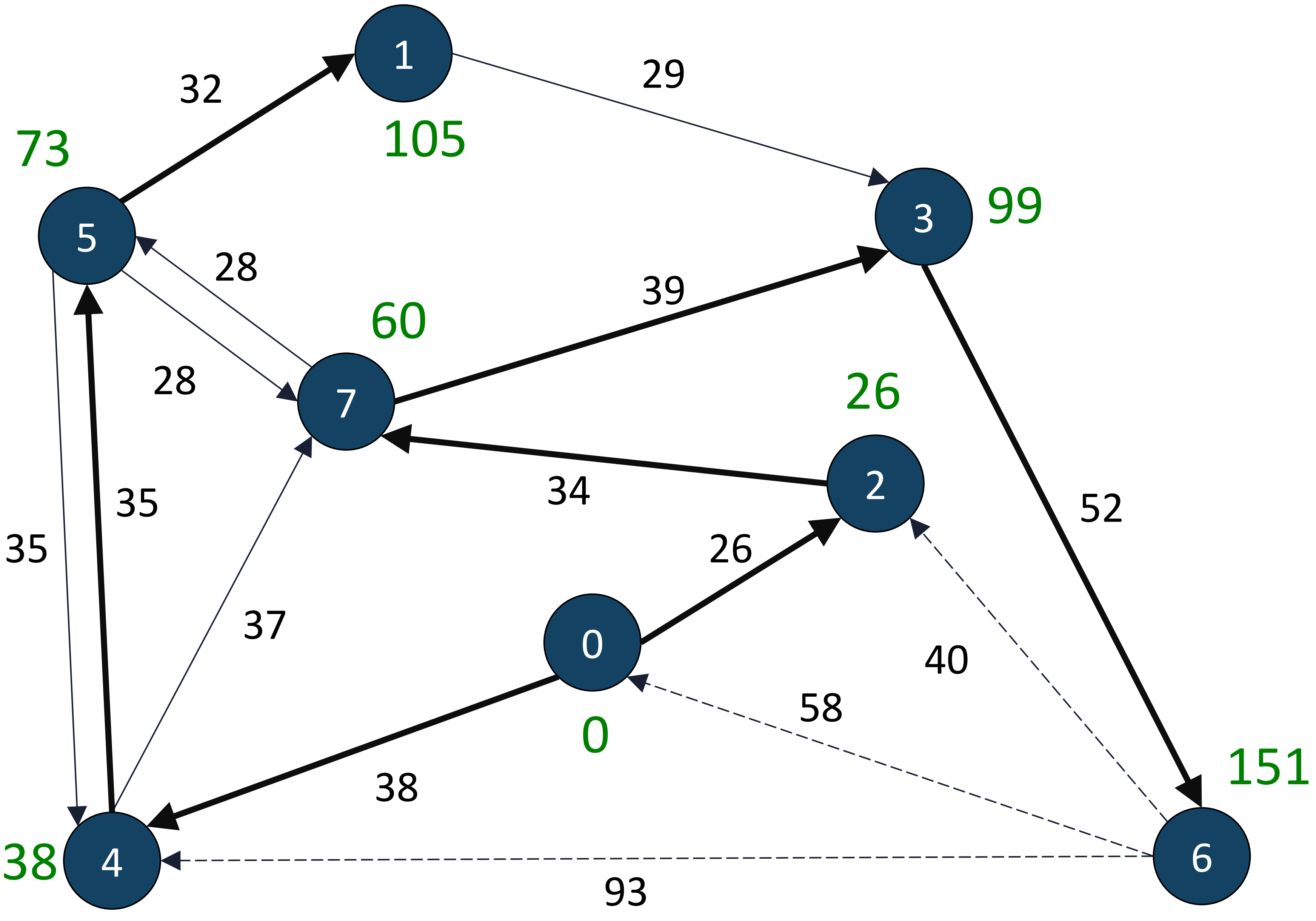


1
29
32
105
73
5
3
99
28
39
60
28
7
26
2
34
35
35
52
26
37
0
40
58
0
∞
38
38
4
6
93

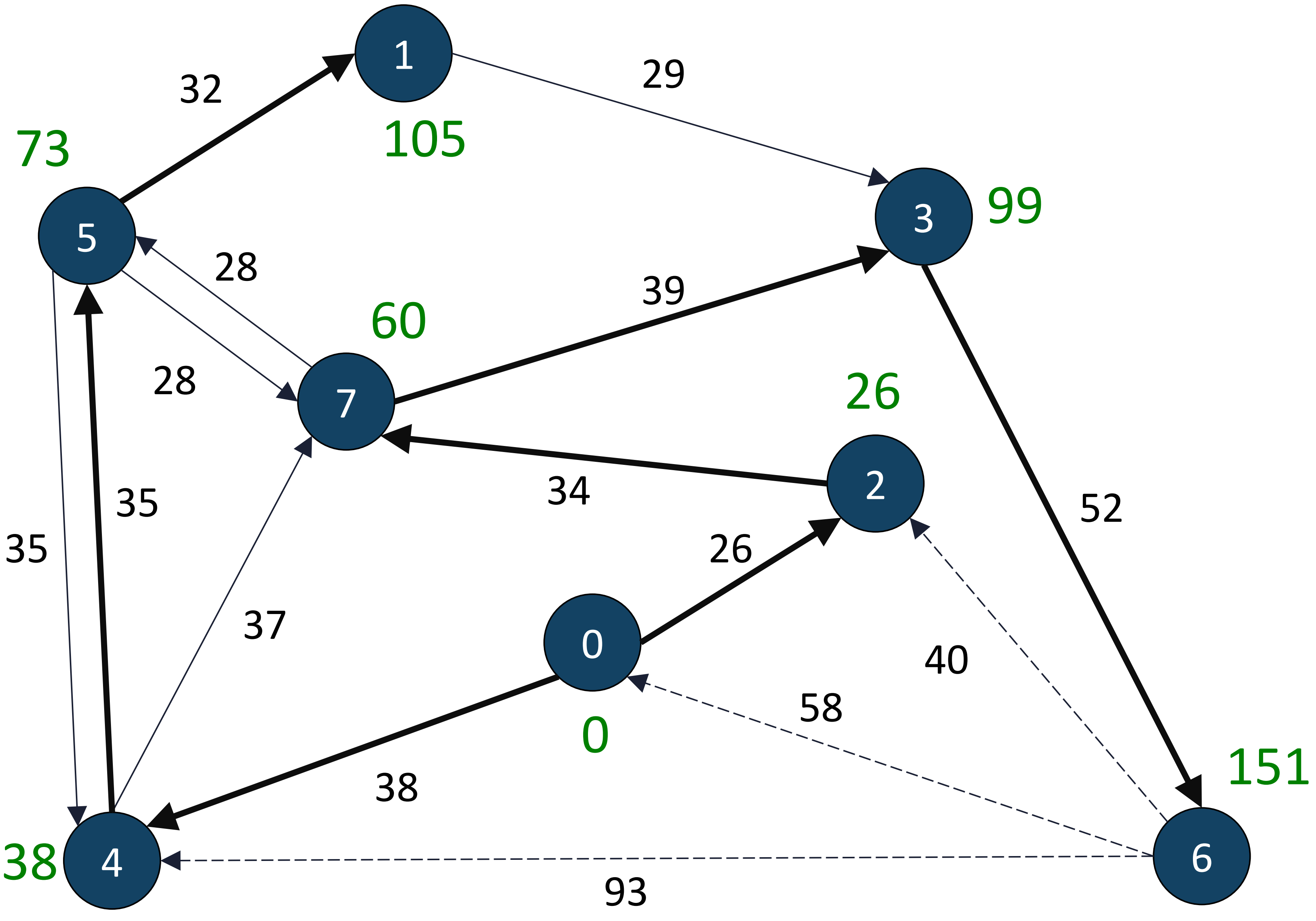

1
105
29
32
73
5
3
99
28
39
60
28
7
26
2
35
34
35
52
26
37
0
40
58
0
151
38
38
4
6
93



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



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



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

Dijkstra en pseudo código

```
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
```

Propiedad de Dijkstra

Dijkstra encuentra las rutas más cortas desde el vértice de partida, *s*, a todos los vértices del grafo (alcanzables desde *s*):

- en el ej. anterior, la ruta 0, 2, 7, 3, 6 es la ruta más corta para ir de 0 a 2, de 0 a 7, de 0 a 3 y de 0 a 6
- … y análogamente para la ruta 0, 4, 5, 1

Una ruta más corta cumple la propiedad de *subestructura óptima*

Los algoritmos para encontrar rutas más cortas usan la siguiente propiedad:

Todas las subrutas en una ruta más corta p entre dos vértices v_0 y v_k son también rutas más cortas

Si $p = \langle v_0, v_1, \dots, v_k \rangle$

… sea $p_{ij} = \langle v_i, \dots, v_j \rangle$, $0 \le i \le j \le k$

… entonces p_{ij} es una ruta más corta de v_i a v_j

La propiedad de *desigualdad triangular*

Sea $\delta(s,v)$ el costo de la ruta más corta de *s* a *v*

Si la (una) ruta más corta de *s* a *v* puede descomponerse en una ruta de *s* a *u* seguida de la arista (u,v)

… entonces $\delta(s,v) = \delta(s,u) + \omega(u,v)$

… y para todas las aristas $(r,v) \in E$

… $\delta(s,v) \leq \delta(s,r) + \omega(r,v)$

Dijkstra en pseudo código

```
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
```

Demostración de la corrección de Dijkstra

Afirmamos que cuando un vértice u sale de la cola Q, implica que hemos llegado hasta u por la ruta más corta —la demostración es por inducción / contradicción

Representemos por $\delta(s, v)$ el costo de la ruta más corta de s a v

0. El primer vértice que sale de Q es s con $\delta(s, s) = 0$: la afirmación se cumple

1. Sea u el primer vértice que sale de Q tal que $d[u] \neq \delta(s, u)$

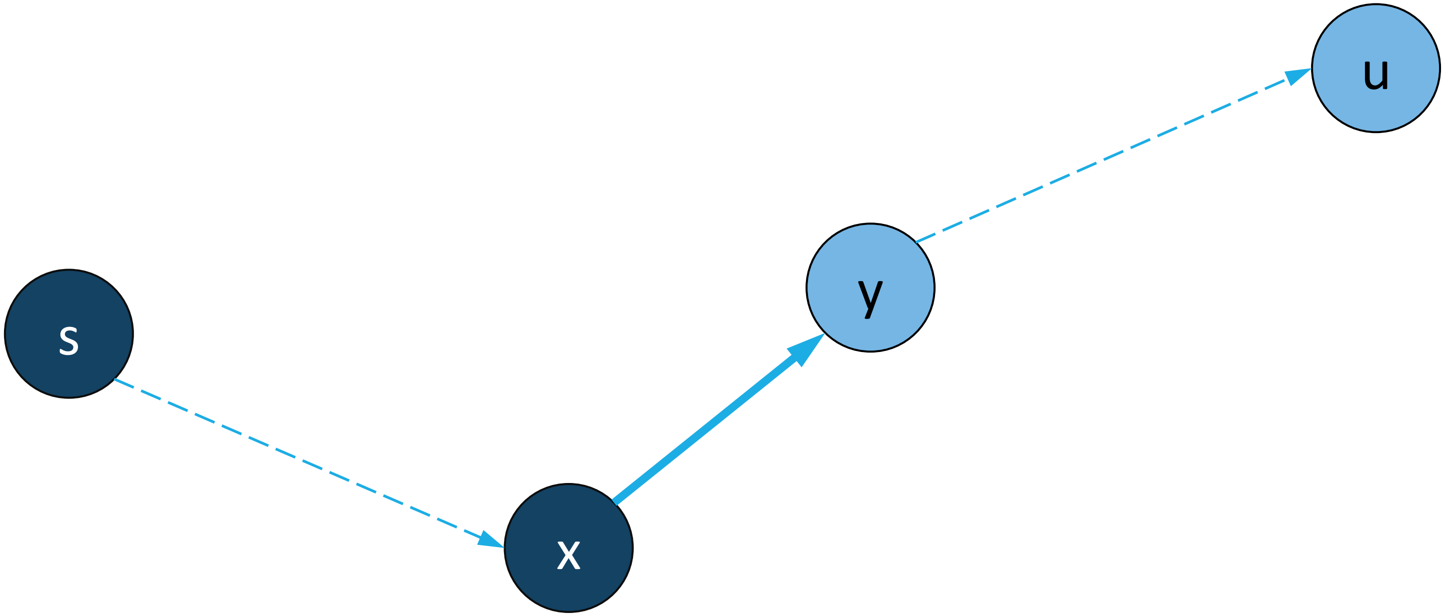
2. Sean p la ruta más corta de s a u … y sea (x, y) una arista en esta ruta tal que:
   y es el primer vértice en la ruta p tal que *y.color* ≠ *black* (es decir, y aún no sale de Q)
   x es el predecesor de y en la ruta p (*x.color* = *black*): $d[x] = \delta(s, x)$ (x ya salió de Q)

3. Como la arista (x, y) fue actualizada al sacar x de Q, entonces $d[y] = \delta(s, y)$ al sacar u de Q

4. Como y aparece antes que u en p y todos los costos son ≥ 0, entonces $\delta(s, y) \leq \delta(s, u)$
   … y como $d[y] = \delta(s, y)$ y $d[u] \geq \delta(s, u)$, entonces $d[y] \leq d[u]$

5. Pero u fue elegido antes que y para sacarlo de Q, por lo que deducimos que $d[u] \leq d[y]$

6. Estas dos desigualdades implican que $d[u] = \delta(s, u)$


s
x
y
u

¿Cuál es la complejidad del algoritmo?

`Dijkstra` realiza $|V|$ **dequeue**'s

… y $|E|$ actualizaciones $\boldsymbol{d}[\boldsymbol{v}] = \boldsymbol{d}[\boldsymbol{u}] + \boldsymbol{costo}(\boldsymbol{u}, \boldsymbol{v})$

Si la cola Q es implementada como un heap binario,

… entonces cada extracción de $\boldsymbol{u}$ y cada actualización de $\boldsymbol{d}[\boldsymbol{v}]$ toma tiempo $O(\log V)$

Así, `Dijkstra` toma tiempo $O((V+E) \log V)$

Variantes

Rutas más cortas en grafos acíclicos

Rutas más cortas de un vértice a otro

Rutas más cortas entre todos los pares de vértices

Rutas más cortas en grafos euclideanos

Algoritmos codiciosos

El algoritmo de **Dijkstra** es **codicioso**

Estos algoritmos no necesariamente producen soluciones **óptimas**

¿Por qué funciona el enfoque codicioso en este caso?