Yadran Eterovic

Rutas más cortas en grafos direccionales con costos

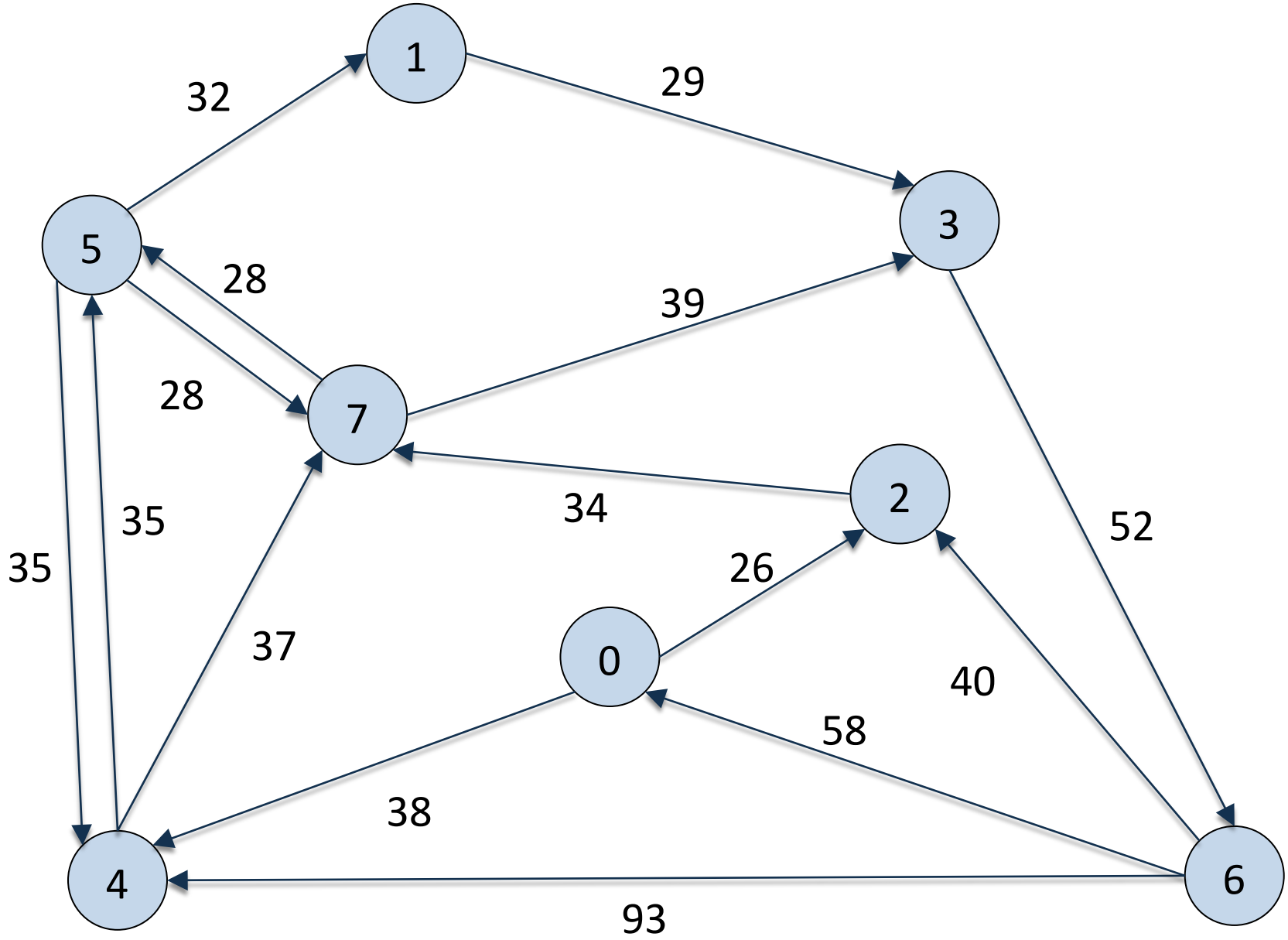
2 - 2017

Estructuras de Datos y Algoritmos - IIC2133

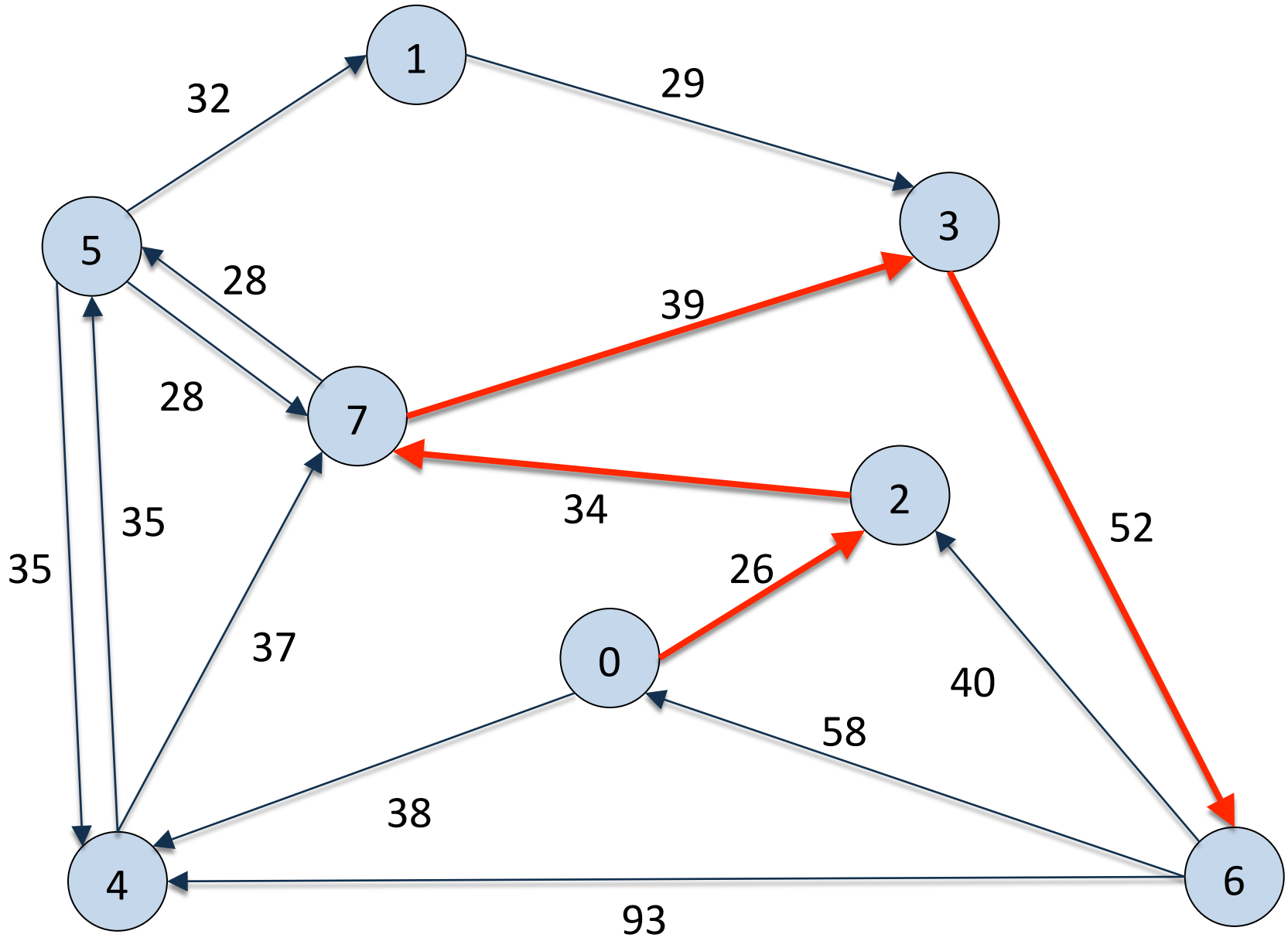
El problema de encontrar rutas más cortas es muy intuitivo en procesamiento de grafos

Una **ruta más corta** desde el vértice *s* al vértice *t* en un grafo direccional con costos

… es una ruta direccional desde *s* a *t* que tiene la propiedad de que ninguna otra ruta de *s* a *t* tiene un costo menor



¿Cuál es la ruta más corta de 0 a 6?



Ruta más corta de 0 a 6

Propiedades del problema de las rutas más cortas

Las rutas son direccionales

Los costos no son necesariamente distancias

No es necesario que todos los vértices sean alcanzables

Los costos negativos introducen complicaciones

Las rutas más cortas son (normalmente) acíclicas

Las rutas más cortas no son necesariamente únicas

Veamos primero el *problema de las rutas más cortas desde un vértice s*

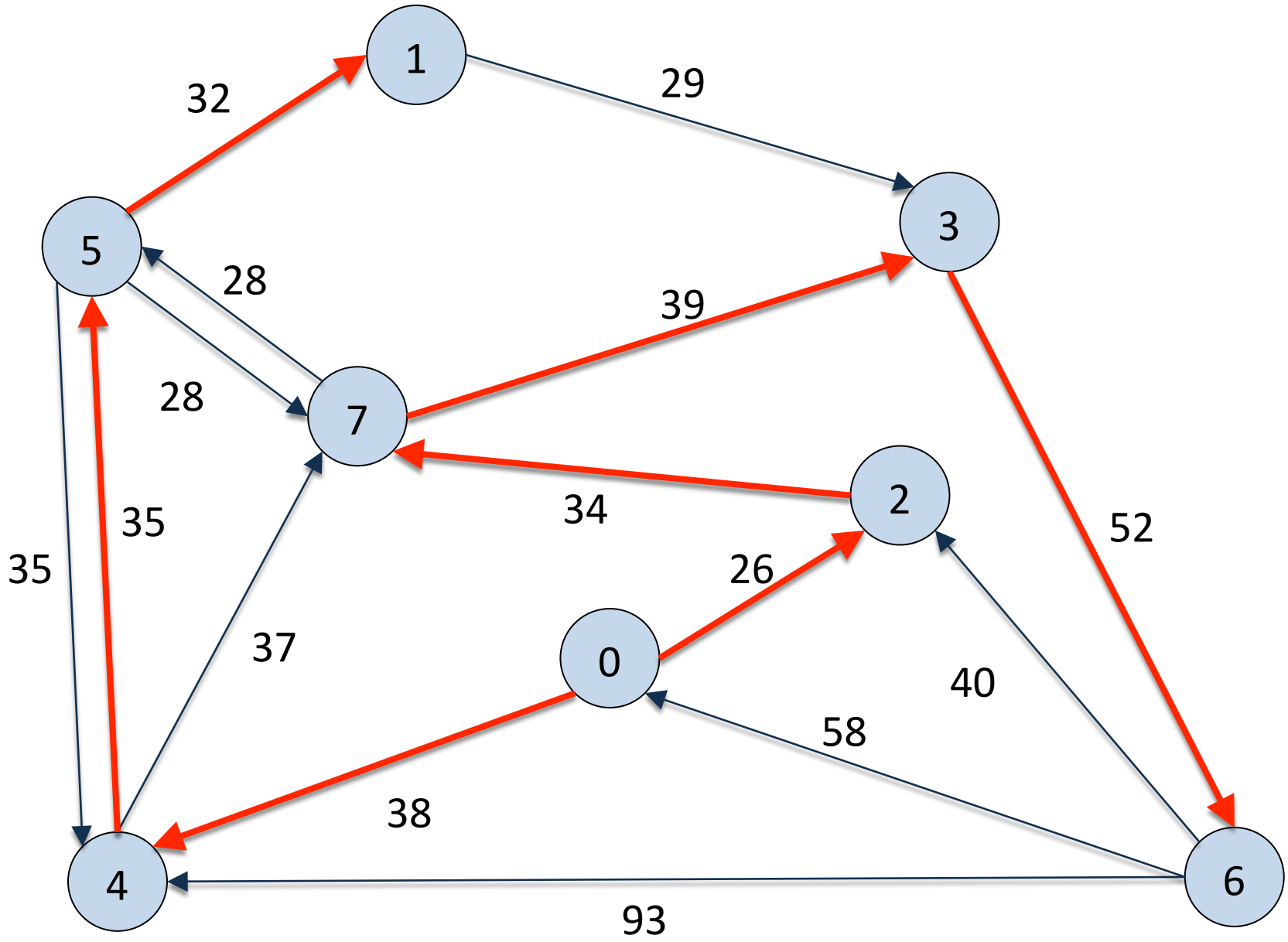
El resultado es un **árbol de rutas más cortas**

… que tiene una ruta más corta desde *s* a todo otro vértice alcanzable desde *s*

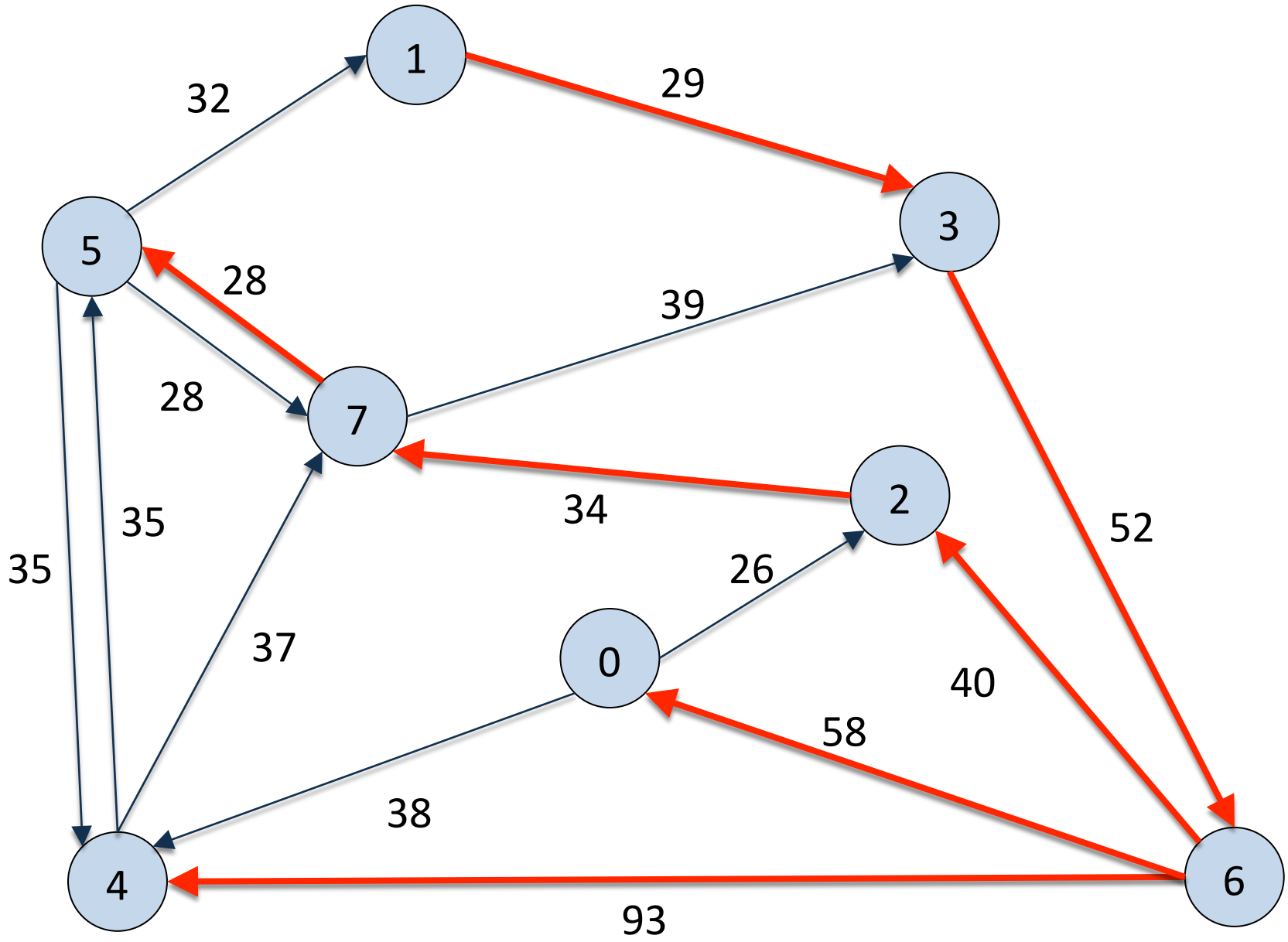
El árbol es direccional, con raíz *s*,

… es un subgrafo que contiene a *s* y a todos los vértices alcanzables desde *s*

… y toda ruta direccional en el árbol es una ruta más corta en el grafo original



Árbol de rutas más cortas con raíz en (desde el) vértice 0



Árbol de rutas más cortas con raíz en (desde el) vértice 1

Estructuras de datos necesarias para representar rutas más cortas

Aristas en el árbol de rutas más cortas:

Como en DFS, BFS y el algoritmo de Prim, usamos un arreglo π[] de vértices, indexado por los vértices, en que π(*v*) es el padre de *v* en el árbol

La arista (π(*v*), *v*) es la última arista en una ruta más corta de *s* a *v*

Por convención, π[*s*] es *null*

Distancia al vértice fuente *s*:

Usamos un arreglo *d*[] de distancias, indexado por los vértices, en que *d*[*v*] es la longitud de la ruta más corta conocida (hasta el momento) de *s* a *v*

Por convención, *d*[*s*] es 0

Inicializamos estos arreglos con `init()` y luego los actualizamos con `reduce()`


```
init(vertex s):
    for each v ∈ V:
        d[v] = ∞
        π[v] = null
    d[s] = 0

reduce(vertex u, vertex v):
    if d[v] > d[u]+ω(u,v):
        d[v] = d[u]+ω(u,v)
        π[v] = u
```

reducir es probar si pasando por *u* podemos mejorar la ruta más corta a *v* encontrada hasta ahora, y, si podemos, actualizamos $d[u]$ y $\pi[u]$

Una ruta más corta tiene 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 $p = \langle v_0, v_1, \ldots, v_k \rangle$

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

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

La propiedad de *desigualdad triangular*

Sea δ(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 $(r,v) \in E$

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

La propiedad de *convergencia*

Así, justo después de reducir una arista (u,v),

… tenemos $d[v] \leq d[u] + \omega(u,v)$

Además, si

hay una ruta más corta de s a v cuyo penúltimo vértice es u

$(u,v) \in E$

$d[u] = \delta(s,u)$ antes de llamar a **`reduce`**

… entonces $d[v] = \delta(s,v)$ depués de la llamada (por la propiedad de subestructura óptima)

El *algoritmo de Dijkstra* …

… calcula las rutas más cortas desde un vértice (fuente) $s \in V$ a todos los otros vértices de G (alcanzables desde s)

Es similar al algoritmo de Prim para encontrar el MST:

en cada paso, agregamos una nueva arista a un único árbol que va creciendo a partir del vértice fuente s

Es aplicable sólo cuando los costos de todas las aristas son ≥ 0

La idea básica del algoritmo de Dijkstra

Mantiene un conjunto S de vértices cuyas rutas más cortas desde s ya han sido calculadas:

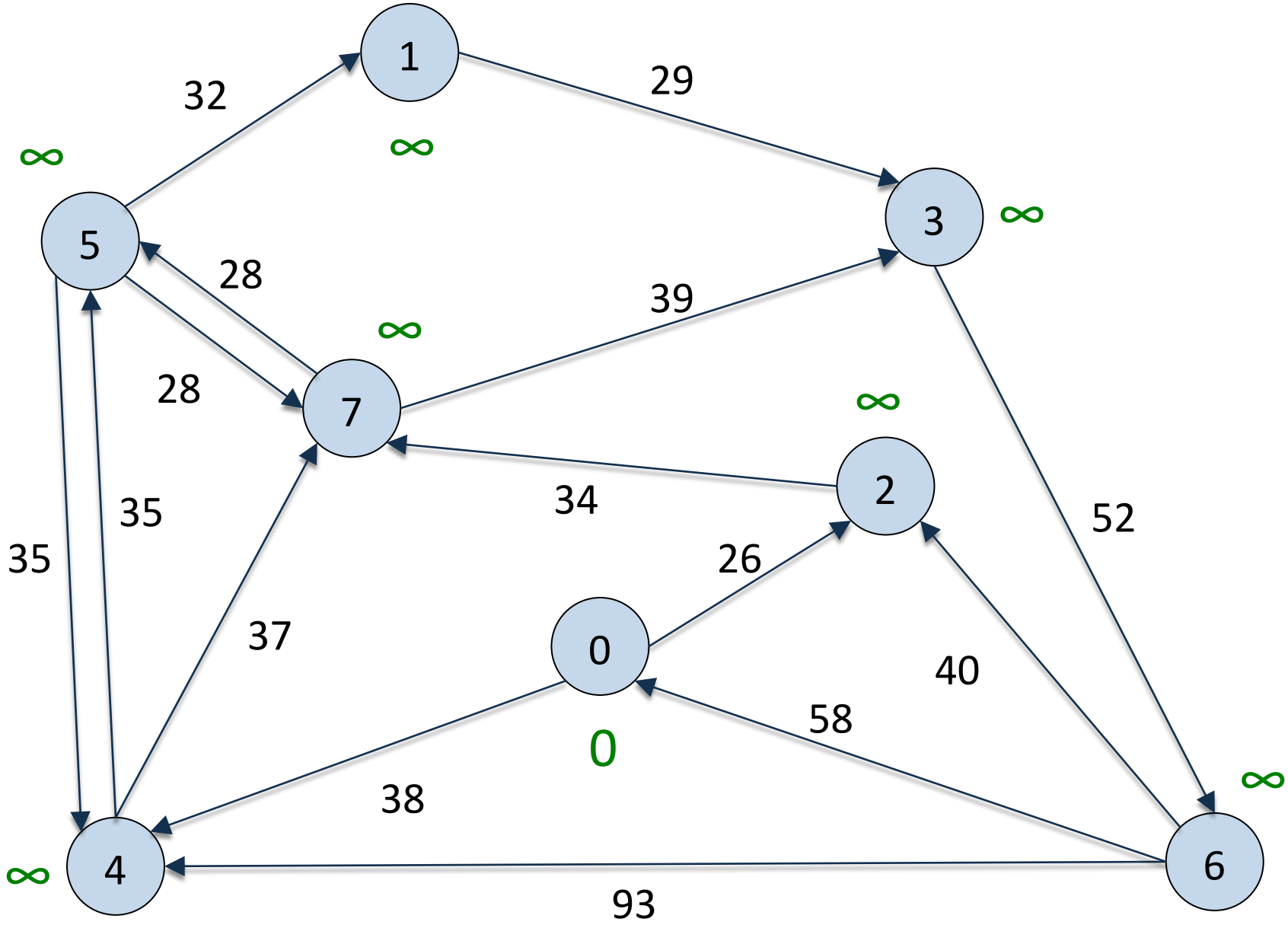
$\forall\ v \in S,\ d[v] = \delta(s,v)$

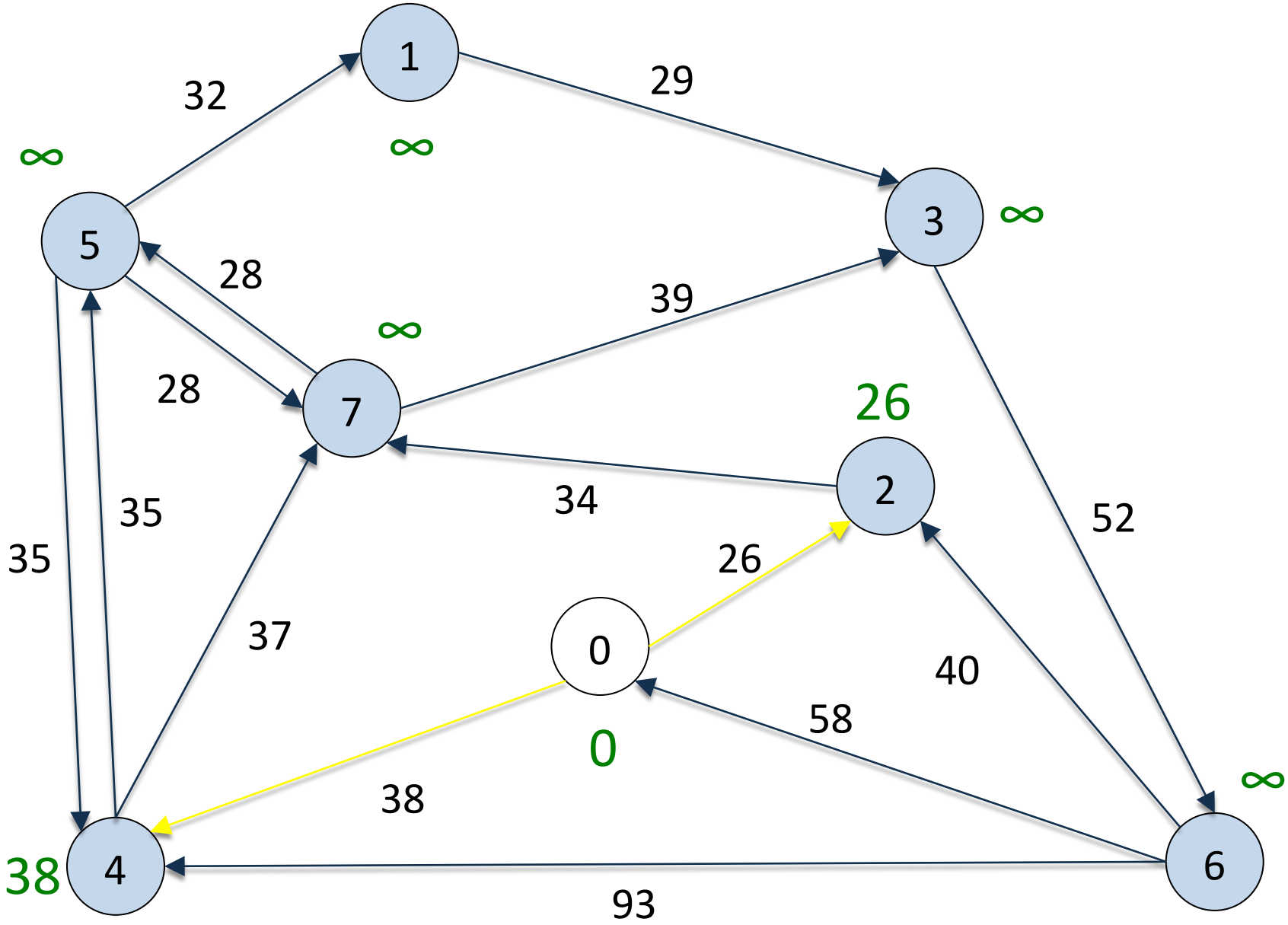
Repetidamente,

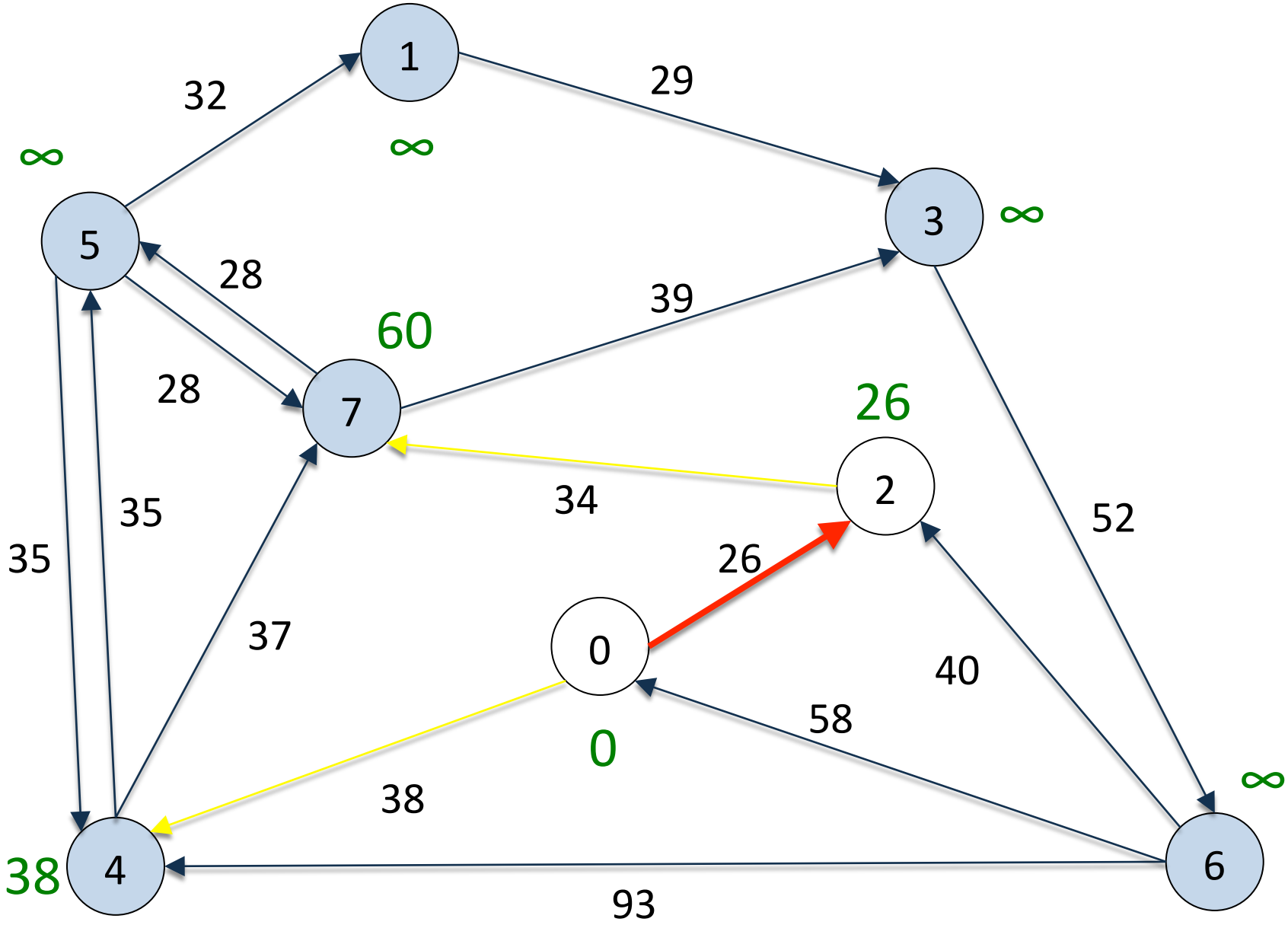
selecciona el vértice $u \in V–S$ tal que $d[u]$ es mínimo

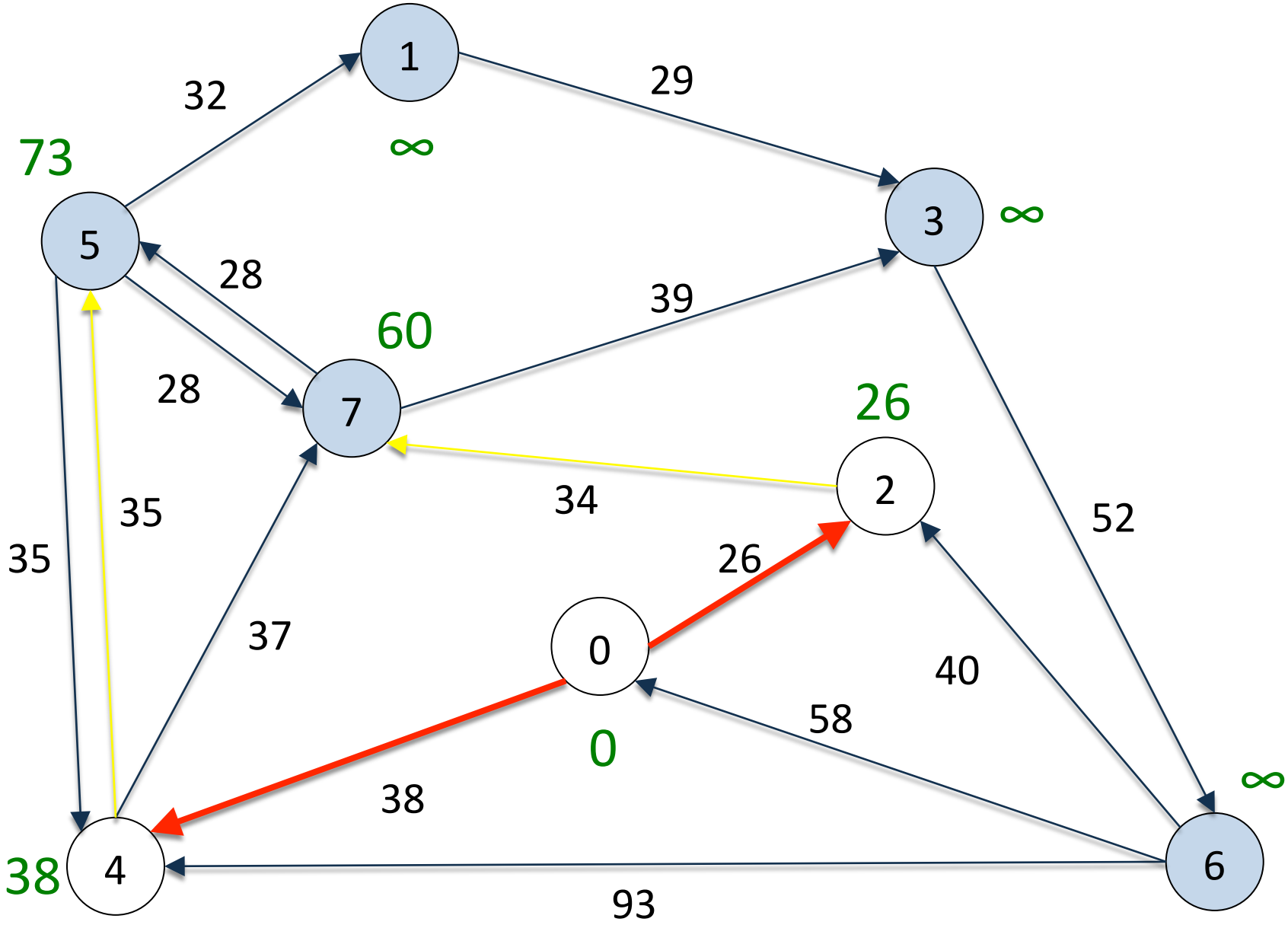
agrega u a S

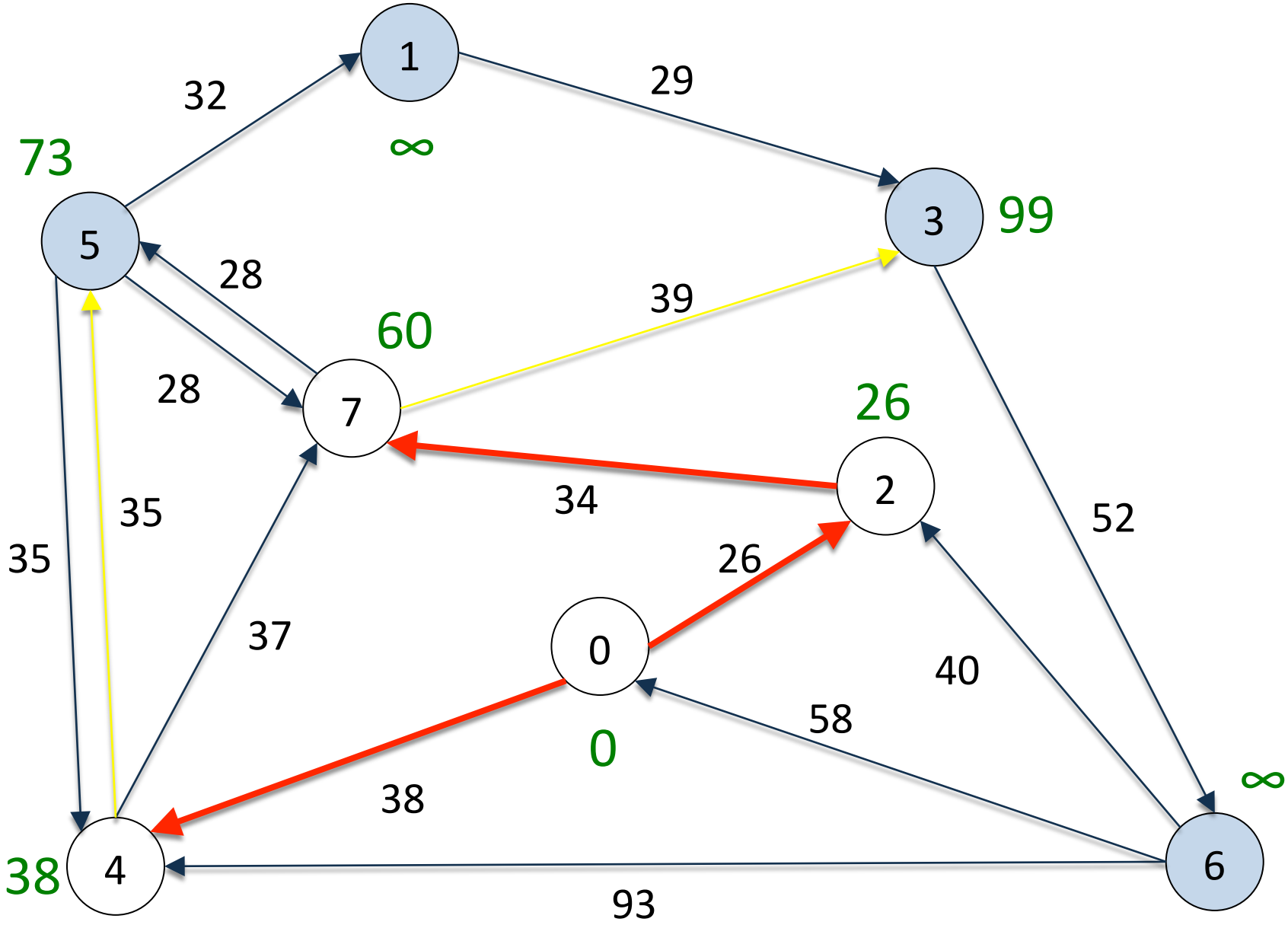
reduce las aristas que salen de u

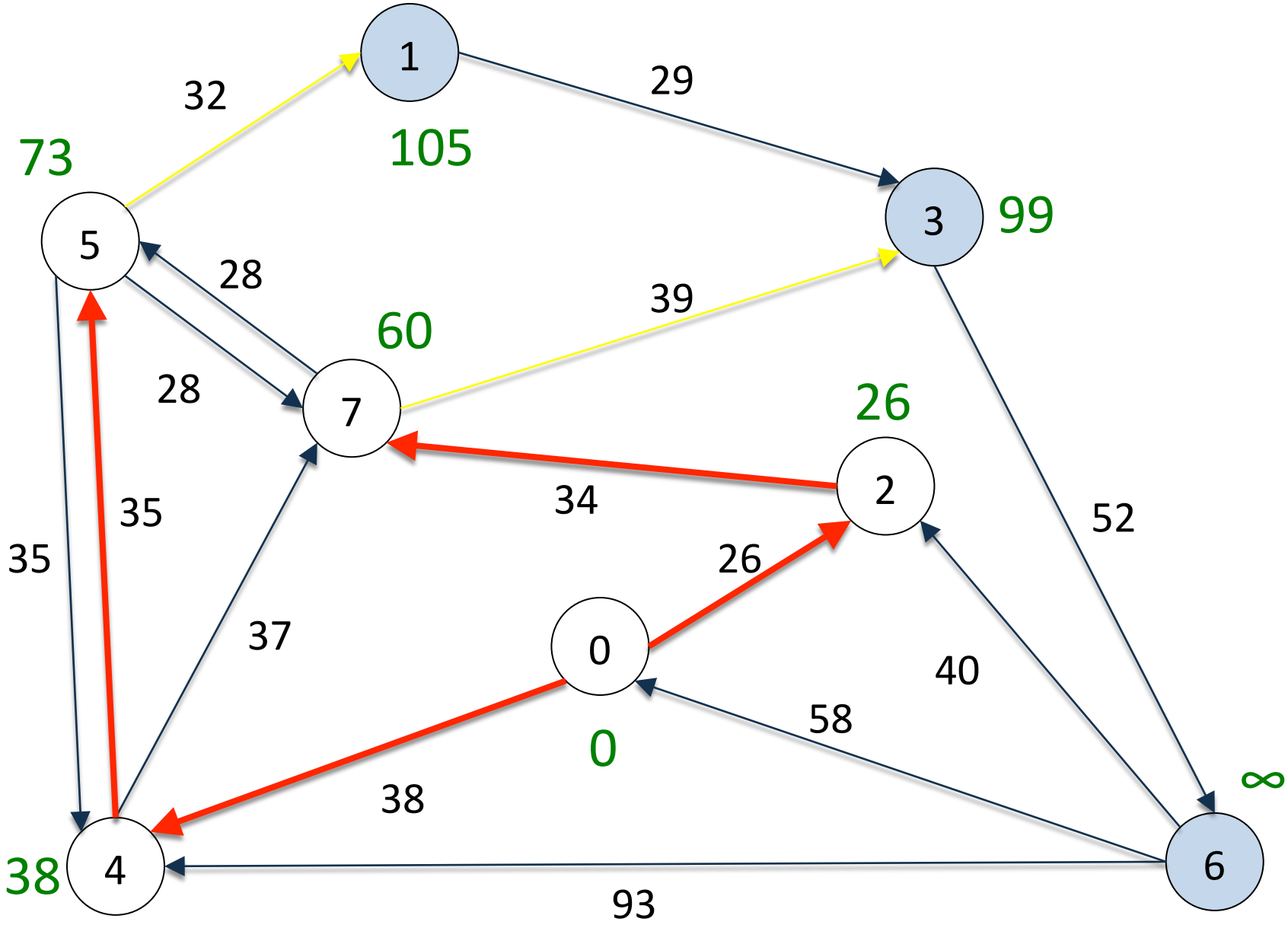

1
32
29
∞
∞
5
3
∞
28
39
∞
28
7
∞
2
34
35
52
35
26
37
0
40
58
0
38
∞
4
6
93

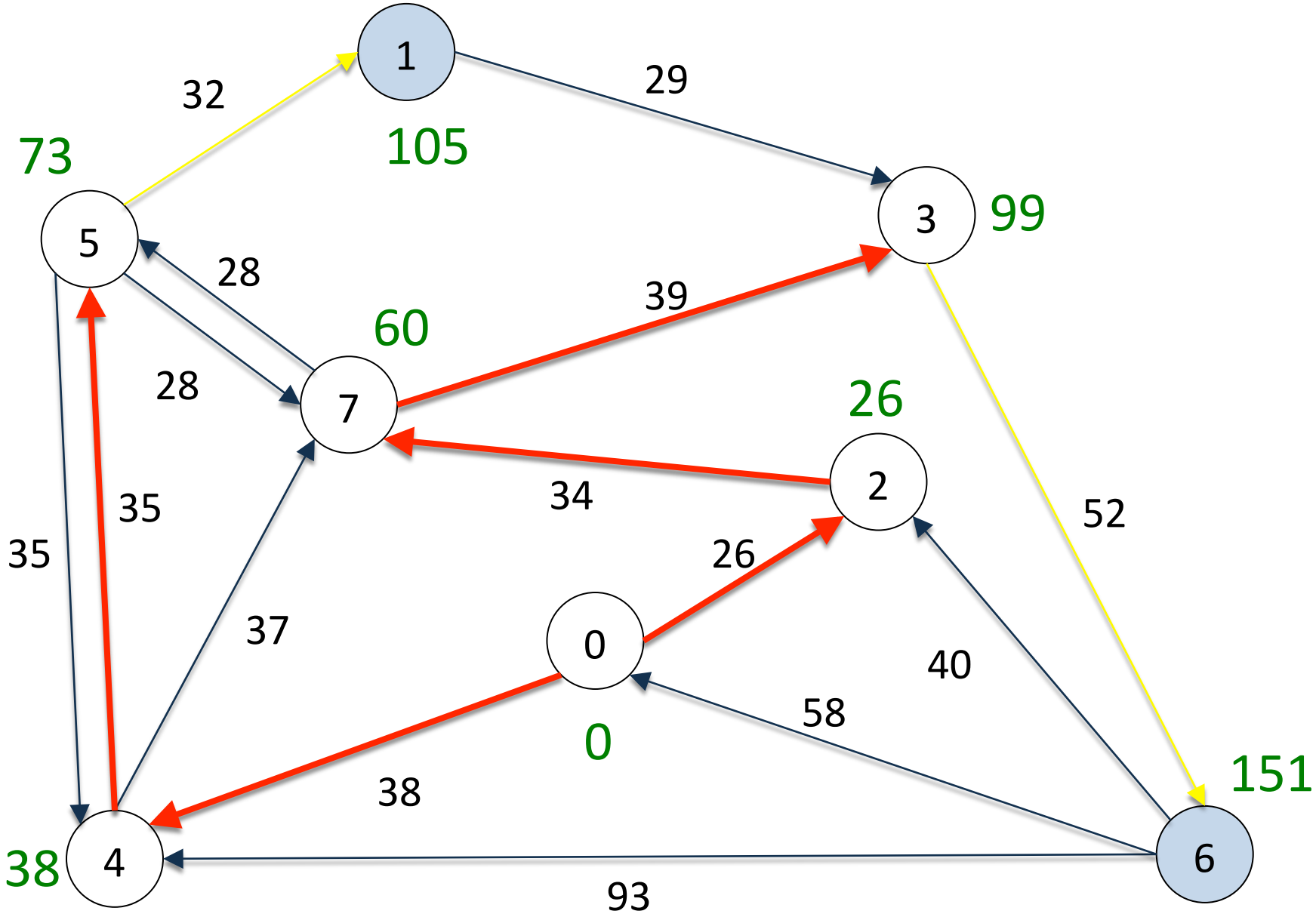

1
32
29
∞
∞
5
3
∞
28
39
∞
28
7
26
2
34
35
52
35
26
37
0
40
58
0
38
∞
38
4
6
93

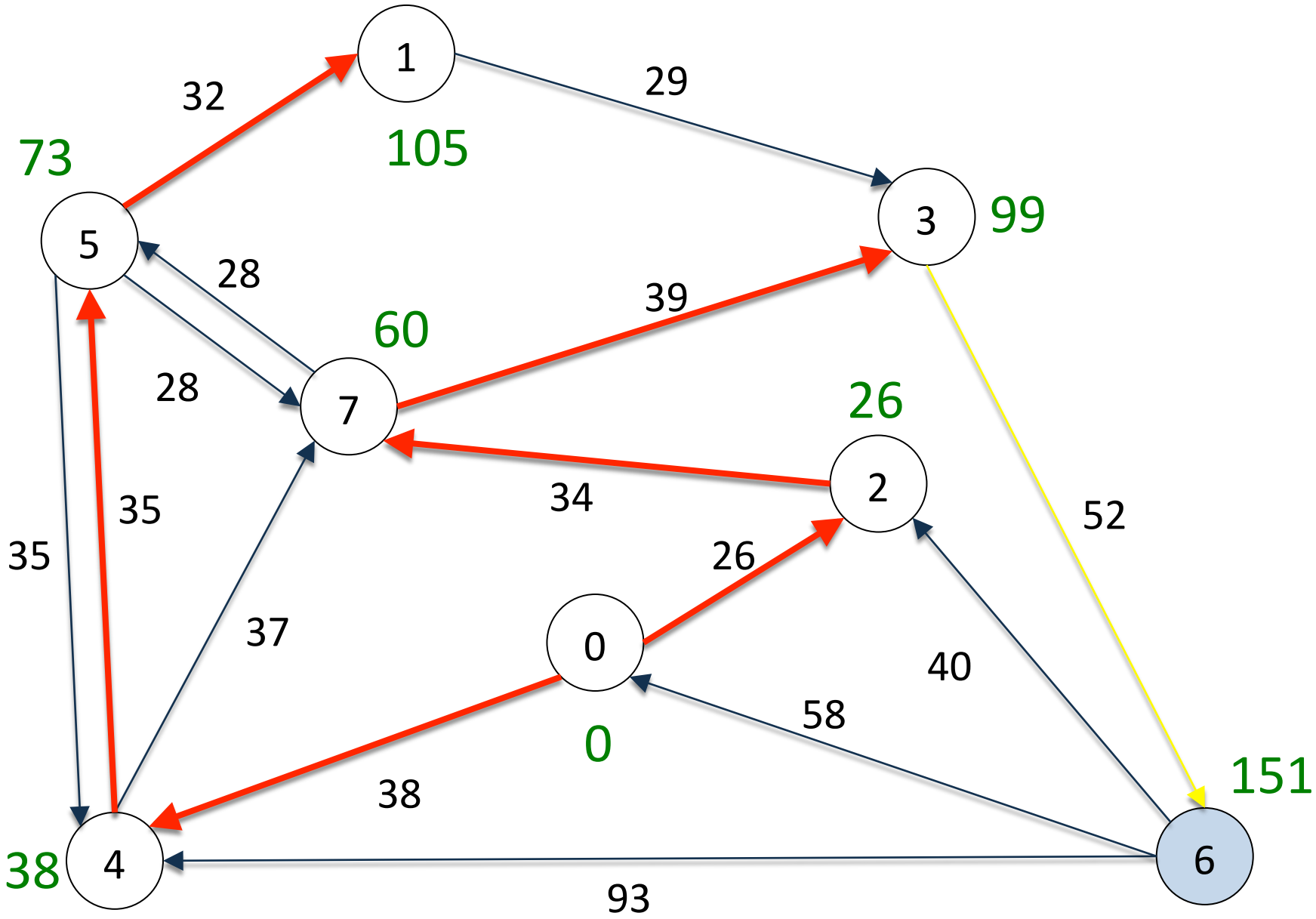

1
32
29
∞
∞
5
3
∞
28
39
60
28
7
26
2
34
35
35
52
26
37
0
40
58
0
38
∞
38
4
6
93

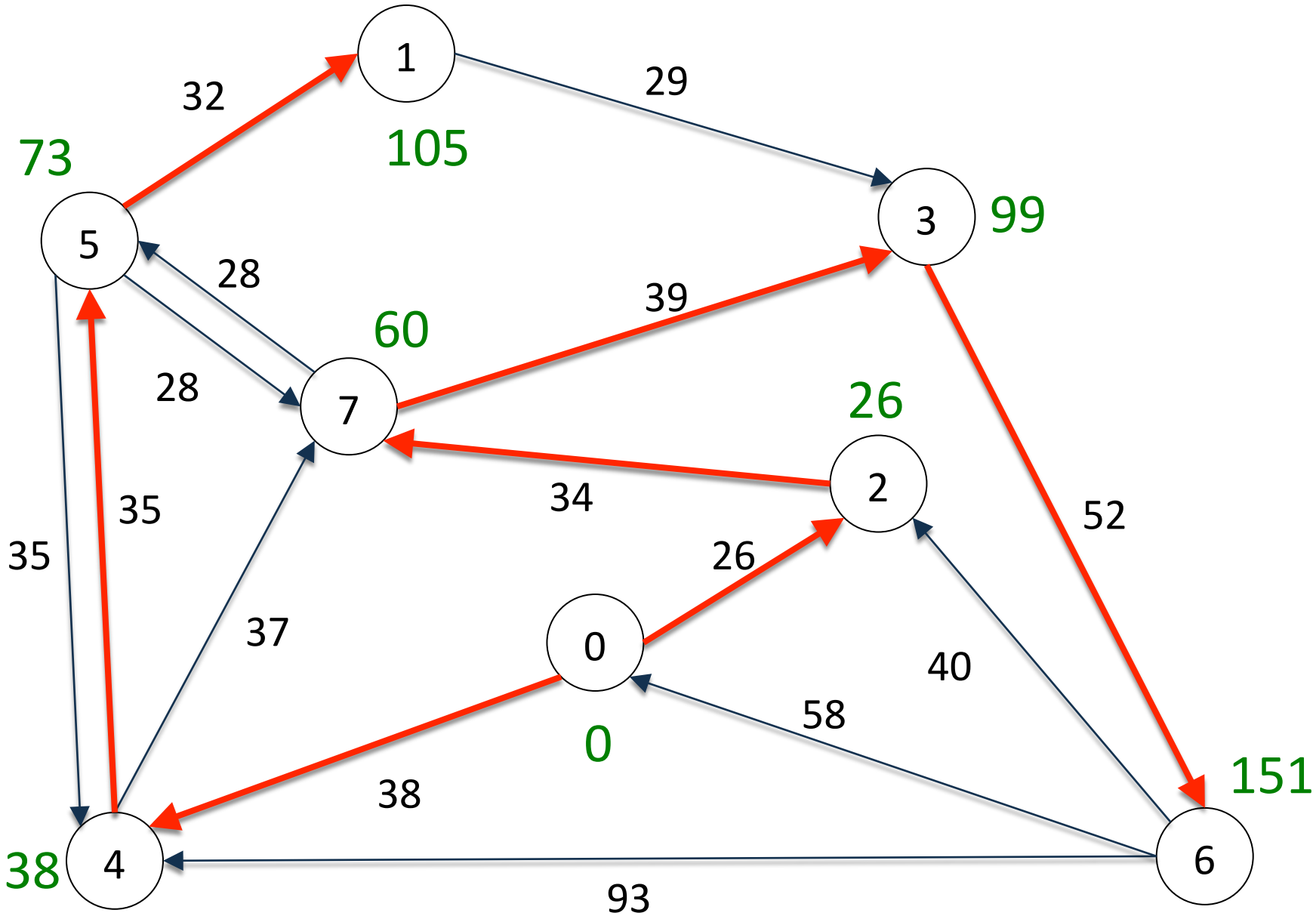

1
2
3
4
5
6
7
0
32
29
73
∞
∞
28
39
60
28
26
35
34
52
35
26
37
40
58
0
38
∞
38
93


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


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


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


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


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

Dijkstra


```
void Dijkstra(Vertex s):
   Init(s)
   S = ∅
   Queue q = new Queue(V)
   while !q.empty():
      Vertex u = q.xMin()
      S = S ∪ {u}
      for each v in α[u]:
         reduce(u,v)
```

vértice fuente

u es el siguiente vértice más cercano a *s* (la primera vez es el propio *s*)

lista de vértices adyacentes a *u*

1. Sea u el primer vértice tal que $d[u] \neq \delta(s, u)$ al ingresar a S

2. Sean p la ruta más corta de s a u

   y el primer vértice en p tal que $y \notin S$

   $x \in S$ el predecesor de y : $d[x] = \delta(s, x)$

3. Como la arista (x, y) fue reducida al ingresar x a S, entonces $d[y] = \delta(s, y)$ al ingresar u a S

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 ingresar a S, por lo que deducimos que $d[u] \leq d[y]$

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

¿Cuál es el desempeño de `Dijkstra`?

`Dijkstra` realiza $|V|$ **`xMin`**'s y $|E|$ **`reduce`**'s

Si la cola `q` es implementada como un heap binario,

… entonces cada una de las **`xMin`**'s y cada una de las actualizaciones de $d[v]$ en **`reduce`** toma tiempo $O(V)$

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

Variantes

Rutas más cortas en grafos direccionales acíclicos

Rutas más cortas en grafos no direccionales

Rutas más cortas entre una fuente y un destino

Rutas más cortas entre todos los pares de vértices

Rutas más cortas en grafos euclideanos

Una versión tratable y bien planteada del problema de rutas más cortas

Requerimos algoritmos que

… asignen un costo de $+\infty$ a los vértices que no son alcanzables desde la fuente

… asignen un costo de $-\infty$ a los vértices que están en una ruta desde la fuente que tiene un vértice que está en un ciclo (con costo acumulado) negativo

… calculen el costo de la ruta más corta (y el árbol de rutas más cortas) para todos los otros vértices

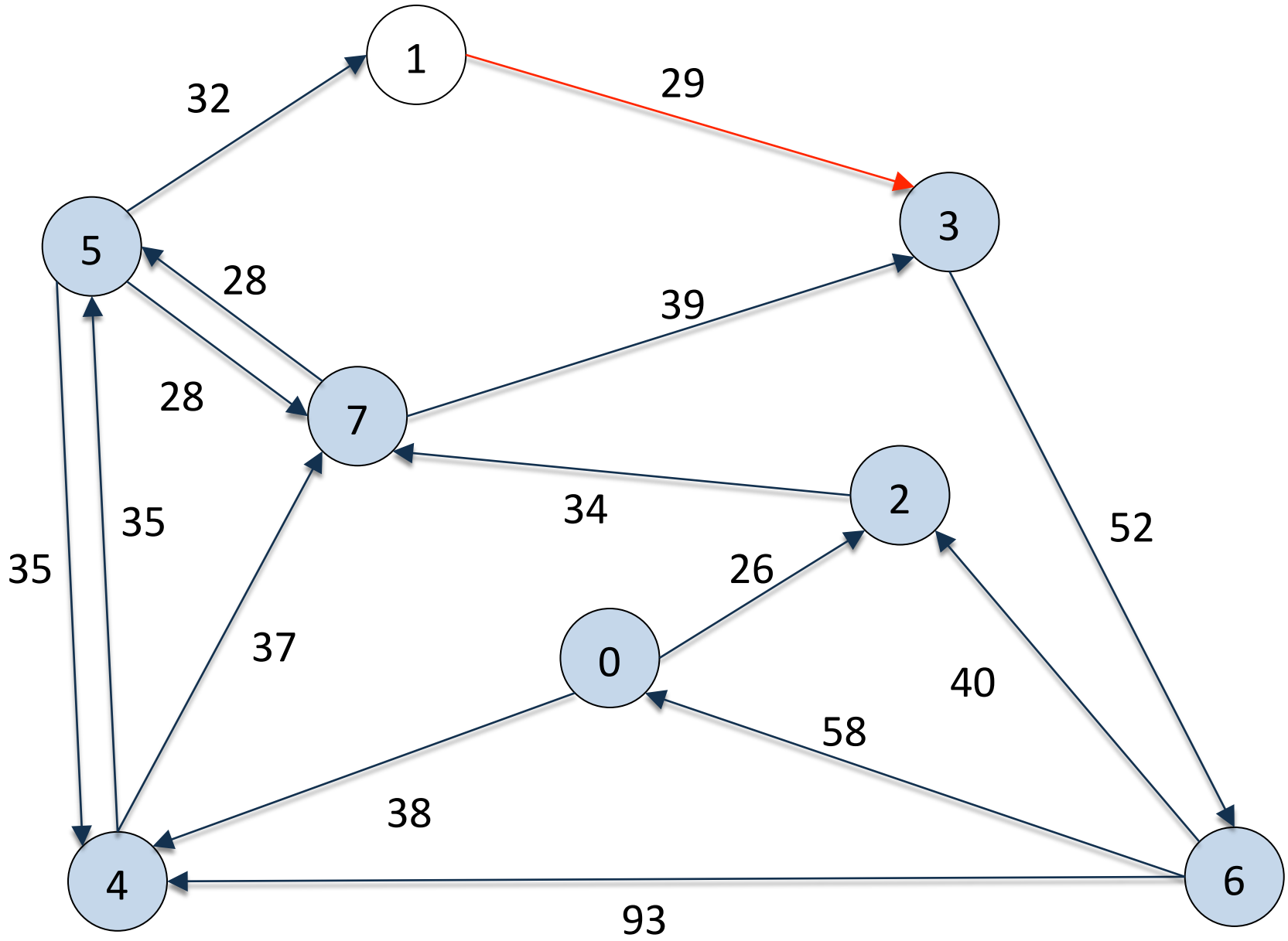
El *algoritmo de Bellman-Ford* …

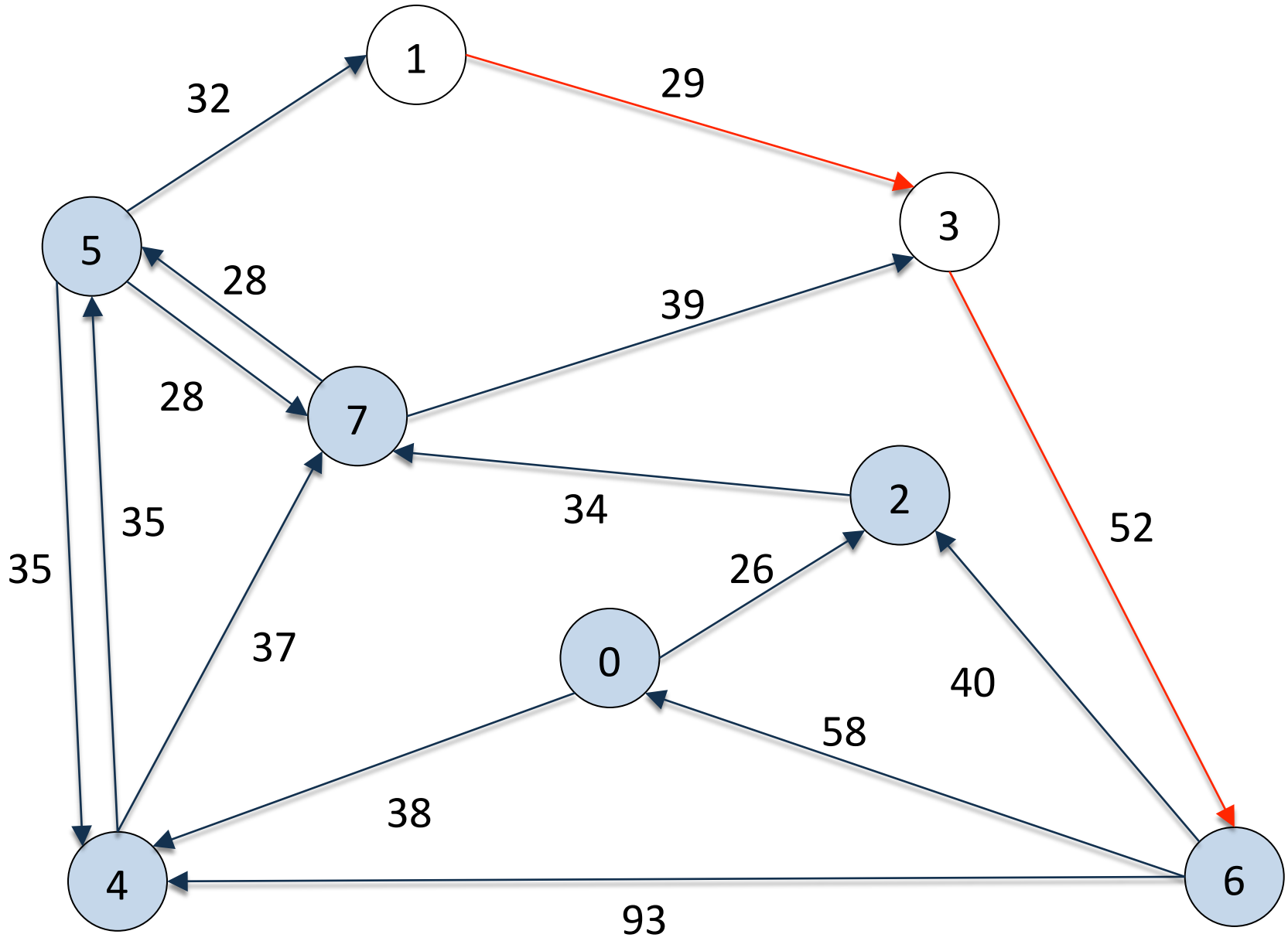
… resuelve el problema de rutas más cortas desde una fuente *s* para cualquier grafo *G* = (*V*, *E*) direccional con costos

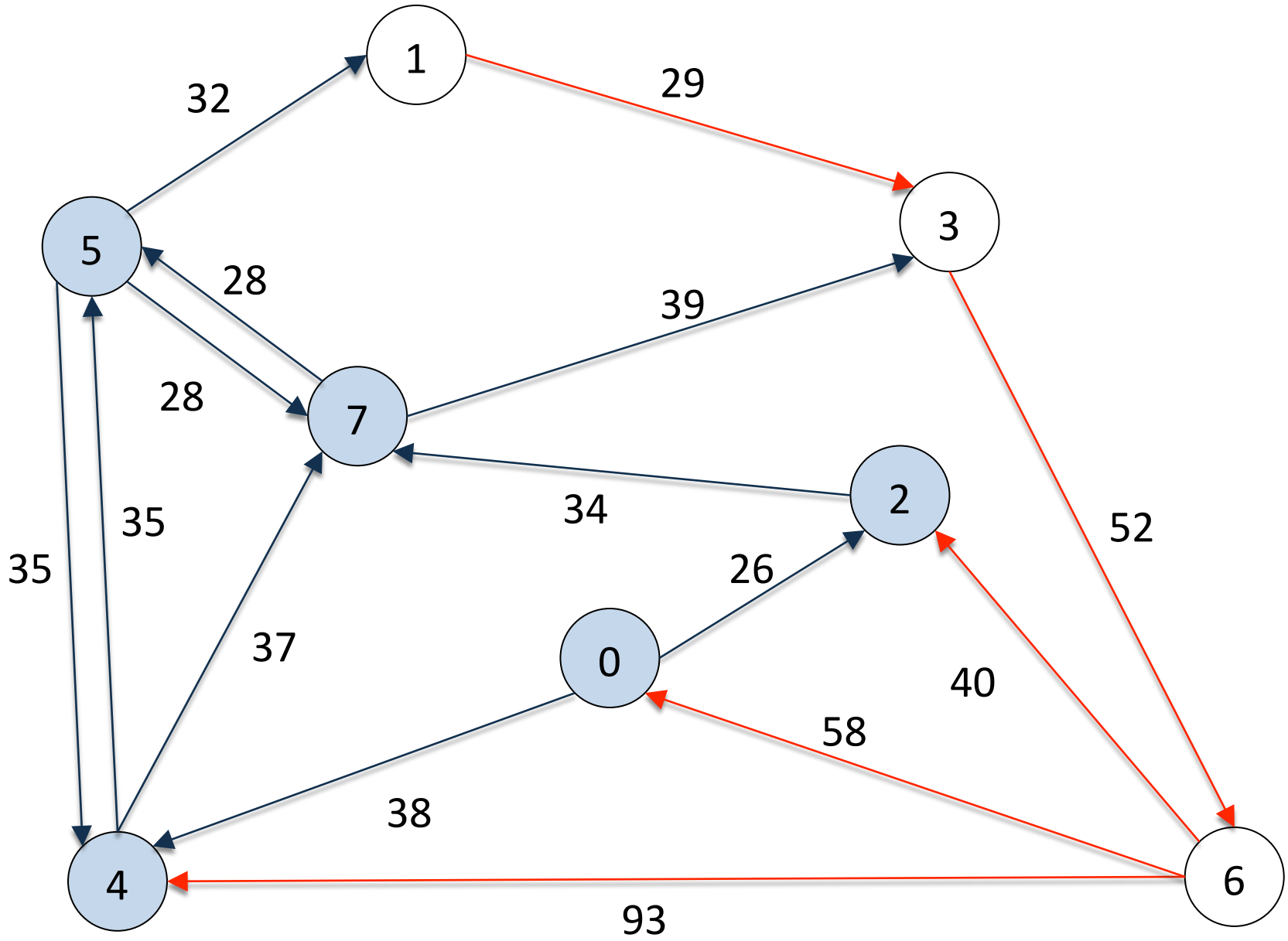
… en que *no hay ciclos* (con costos acumulados) *negativos* alcanzables desde *s*:

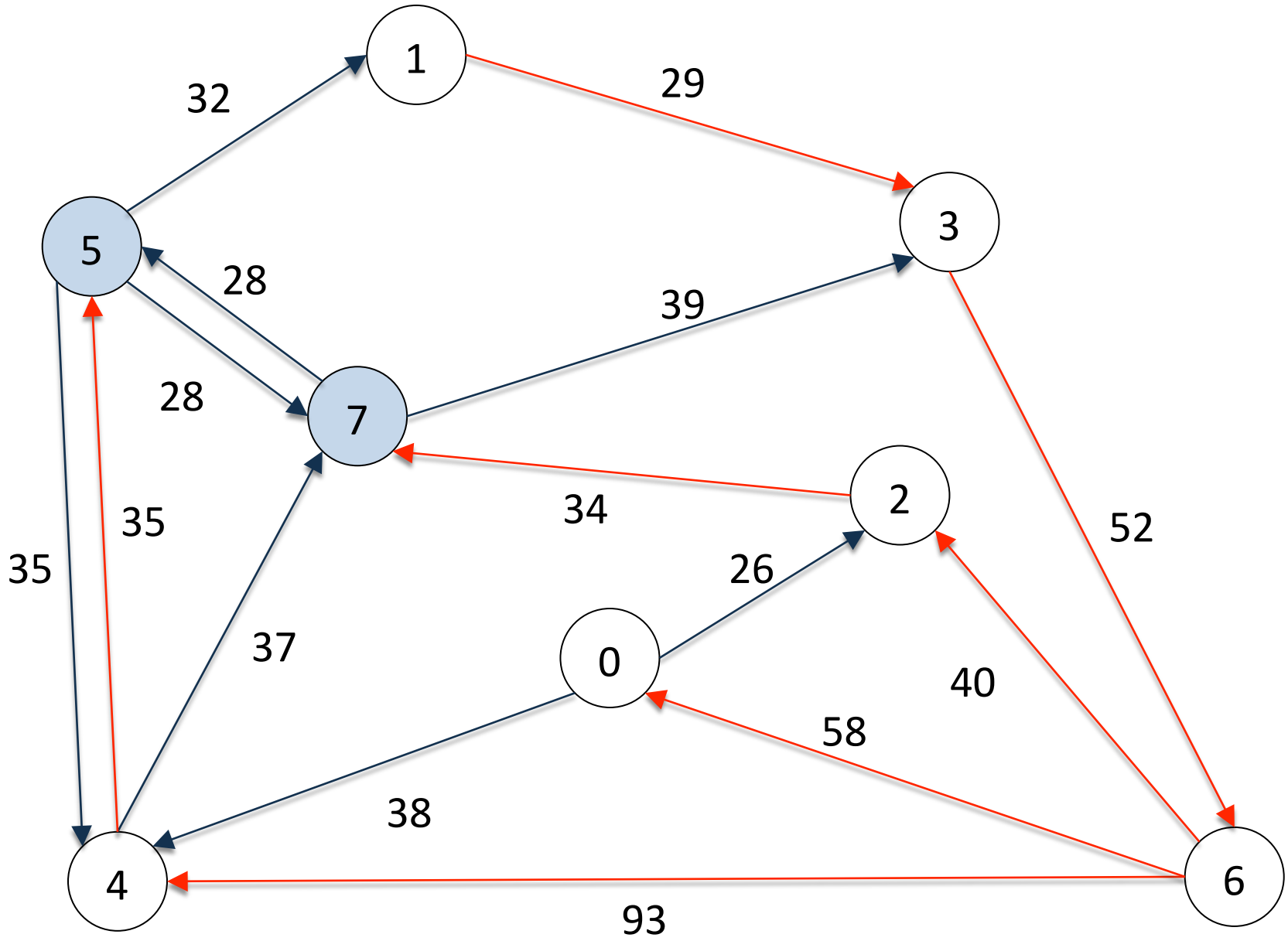
```
init(s)
for k = 1 ... |V|-1:
   for each (u,v) ∈ E:
      reduce(u,v)
```

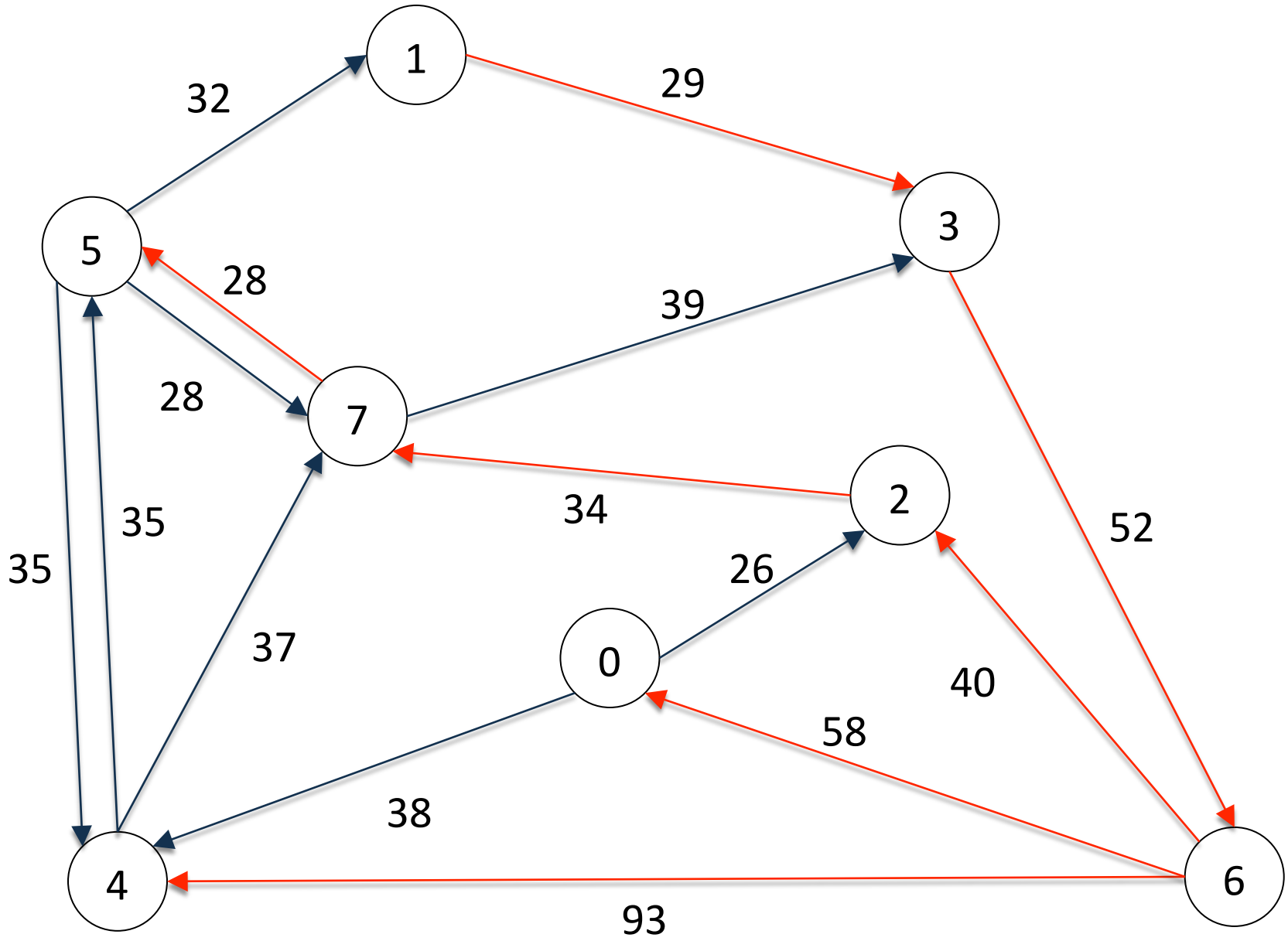
```
init(vertex s):
   for each v ∈ V:
      d[v] = ∞
      π[v] = null
   d[s] = 0

reduce(vertex u, vertex v):
   if d[v] > d[u]+ω(u,v):
      d[v] = d[u]+ω(u,v)
      π[v] = u
```


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


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


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


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


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

El algoritmo toma tiempo $O(|E||V|)$

Cada una de las $|V|$ iteraciones reduce cada una de las $|E|$ aristas

Sin embargo, las únicas aristas que podrían producir un cambio en $d[]$ son aquellas que salen de un vértice cuyo $d[]$ cambió en la iteración anterior

Para seguirle la pista a estos vértices, usamos una cola

La corrección del algoritmo se puede demostrar por inducción

Para cualquier vértice *t* alcanzable desde *s* consideremos una ruta más corta de *s* a *t* :

$v_0 \to v_1 \to \ldots \to v_k$

… en que v_0 es *s* y v_k es *t*

Como no hay ciclos negativos, tal ruta existe y *k* no puede ser mayor que $|V|-1$

Demostramos por inducción sobre *i* que después de la *i*-ésima iteración el algoritmo calcula una ruta más corta de *s* a v_i

El caso base ($i = 0$) es trivial

Para el paso inductivo,

… suponemos que la afirmación es verdadera para i :

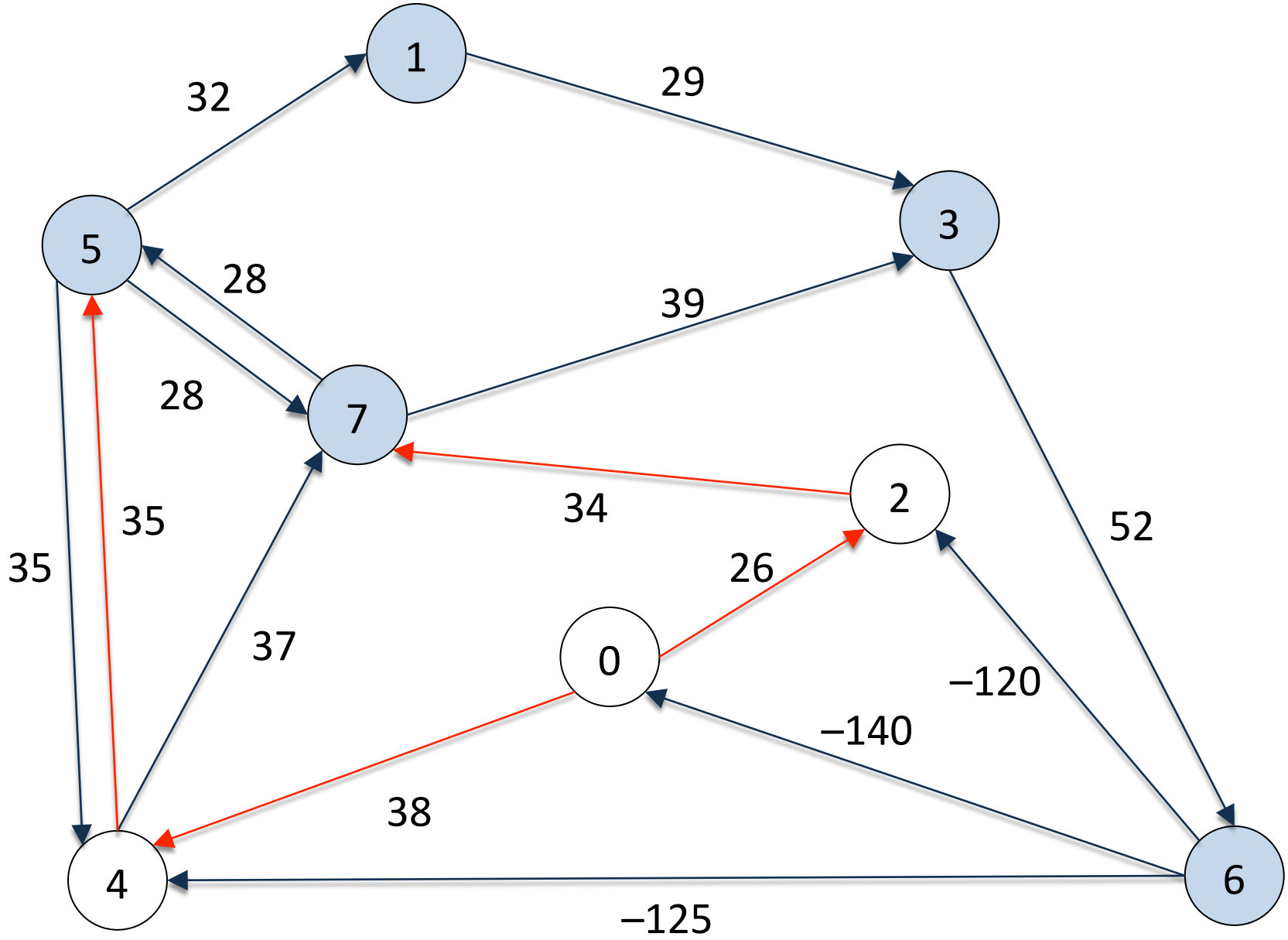
$v_0 \rightarrow \ldots \rightarrow v_i$ es una ruta más corta de s a v_i y su costo es $\delta[s,v_i]$

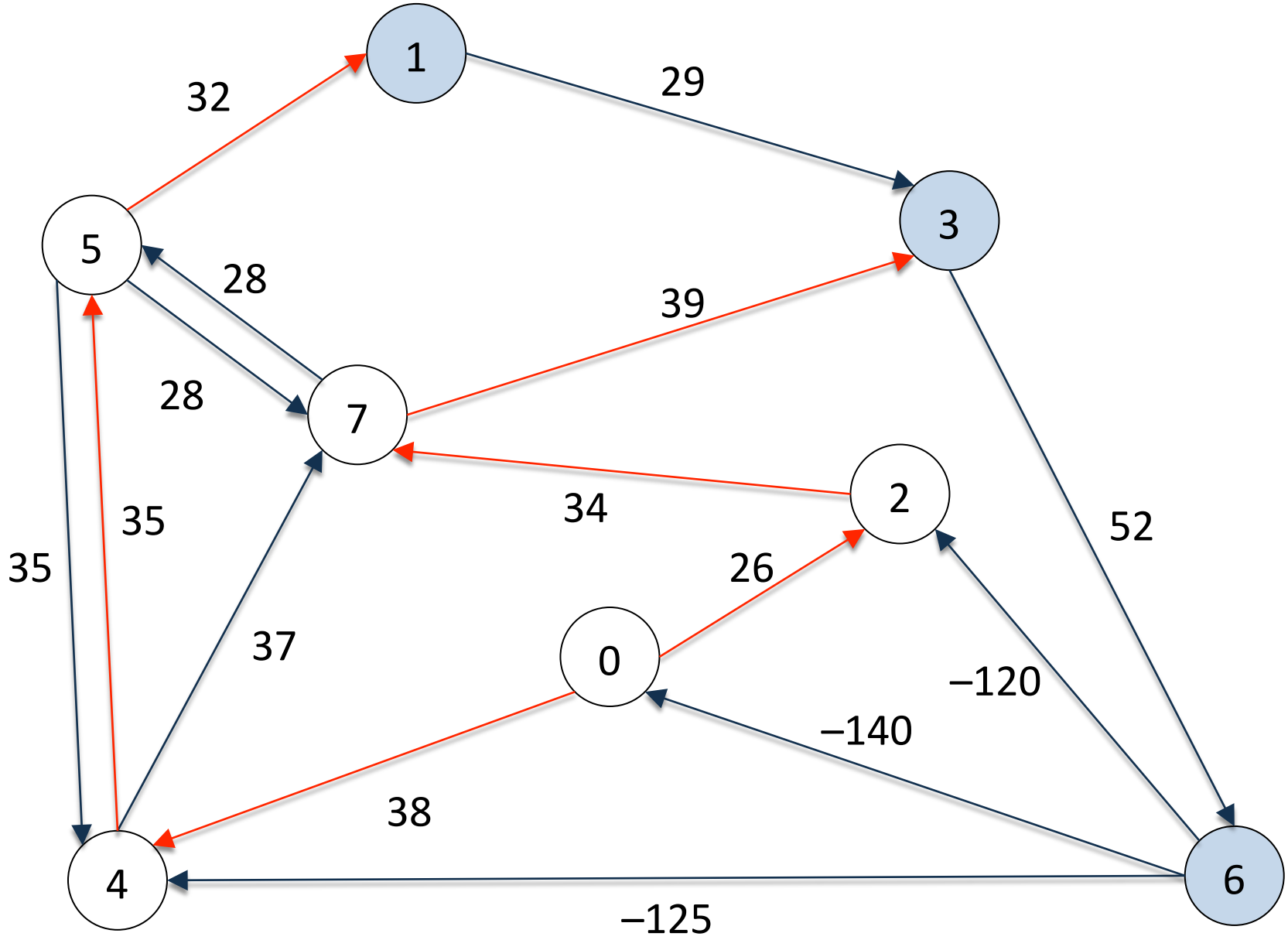
… luego, reducimos todos los vértices en la i-ésima iteración:

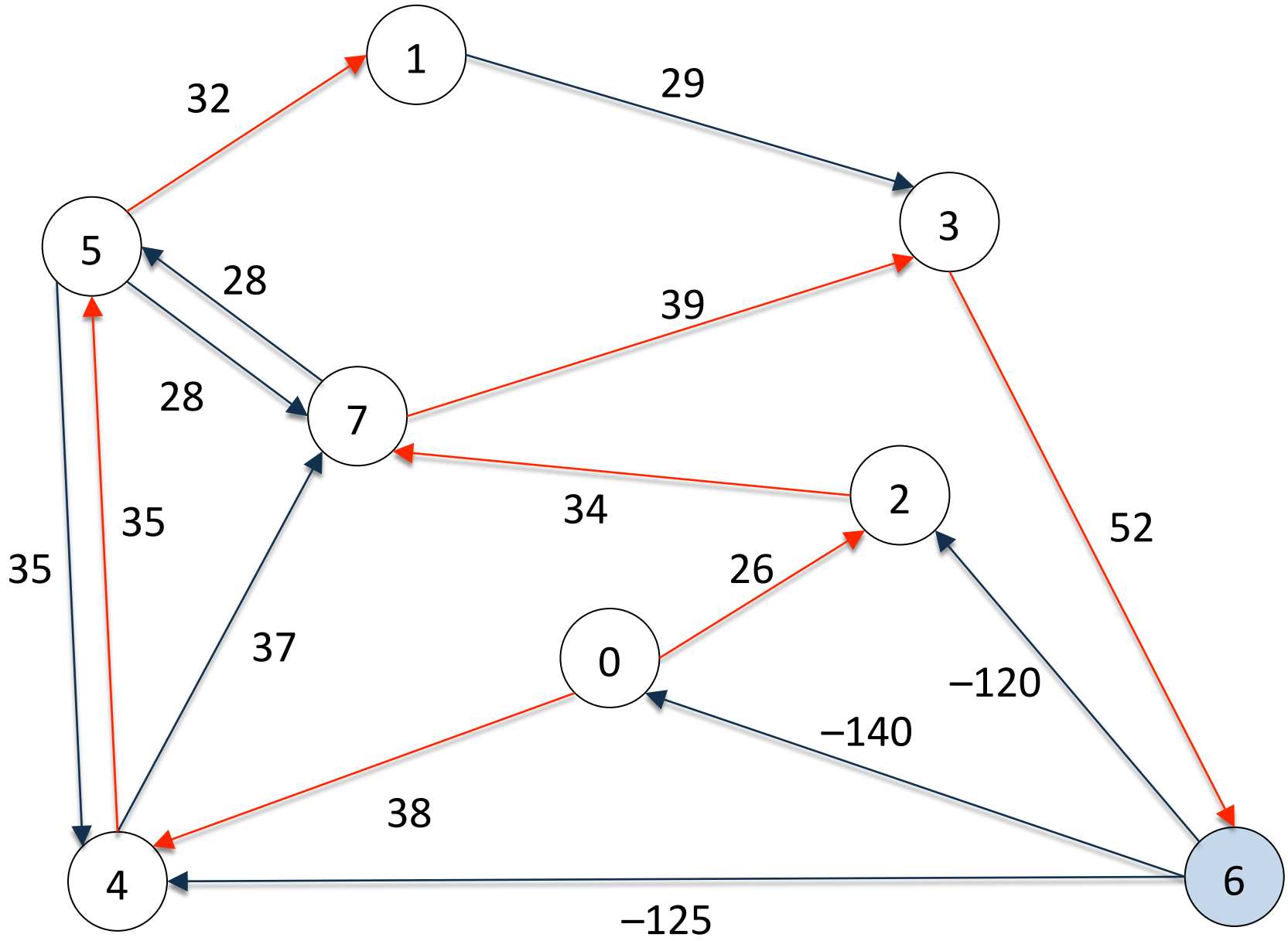
$d[v_{i+1}] \leq \delta[s,v_i] + \omega(v_i , v_{i+1})$

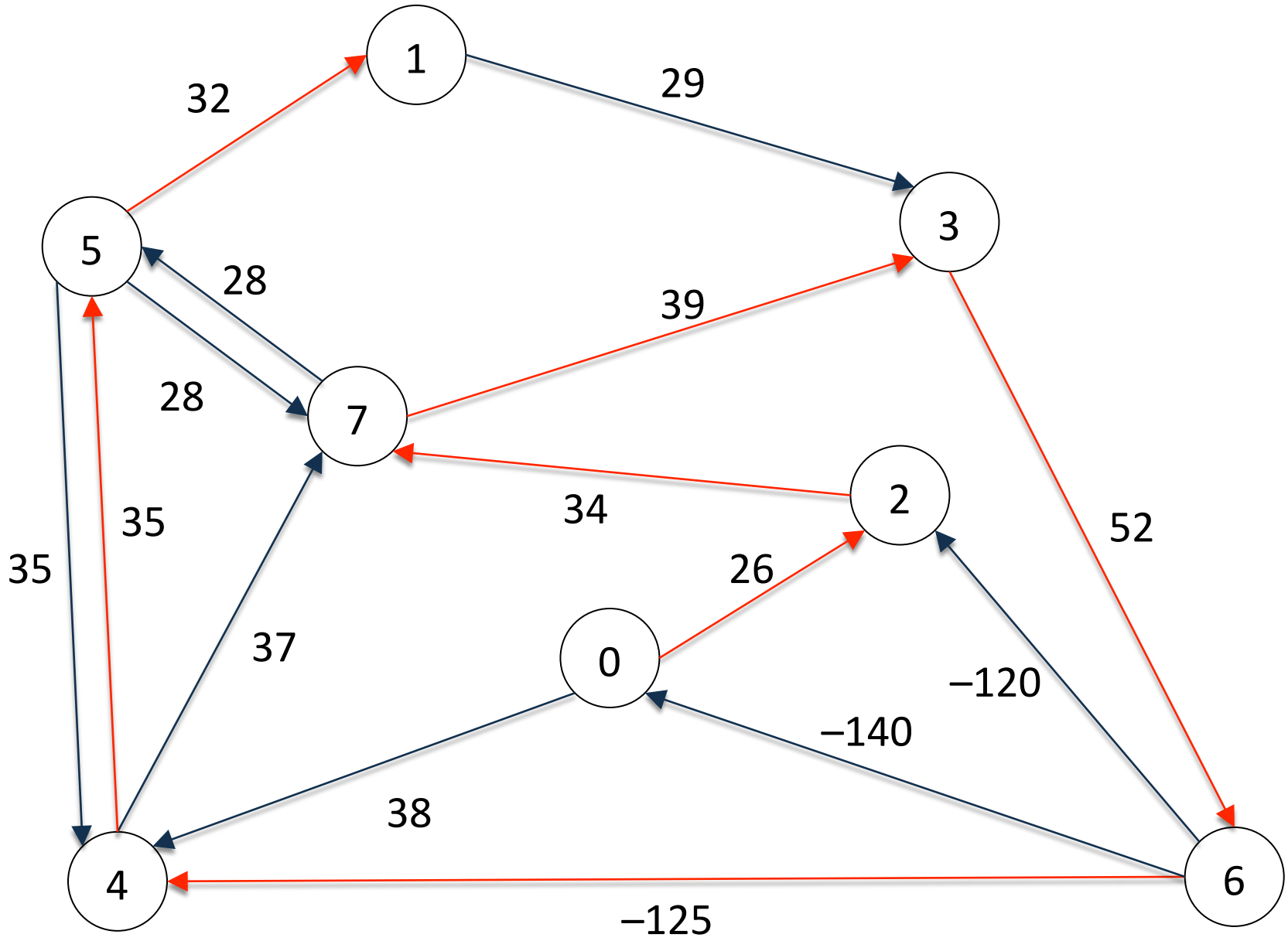
Por la propiedad de convergencia (diap. #13), después de la i-ésima iteración, $d[v_{i+1}]$ debe ser igual a $\delta[s,v_i] + \omega(v_i , v_{i+1}) = \delta[s,v_{i+1}]$

Veamos ahora el caso de un grafo con algunas aristas con costos negativos


1
32
29
3
5
28
39
28
7
35
34
2
52
35
26
37
0
−120
−140
38
4
6
−125


1
32
29
3
5
28
39
28
7
35
34
2
52
35
26
37
0
−120
−140
38
4
6
−125


1
32
29
3
5
28
39
28
7
35
34
2
52
35
26
37
0
−120
−140
38
4
6
−125


1
32
29
3
5
28
39
28
7
35
35
34
2
52
26
37
0
−120
−140
38
4
6
−125

El problema de *rutas más cortas entre todos los pares de vértices*

Podemos ejecutar $|V|$ veces un algoritmo para rutas más cortas desde un vértice, una vez para cada vértice en el rol de s :

- si los costos de las aristas son no negativos, podemos usar el algoritmo de Dijkstra

  … el tiempo de ejecución sería $O(VE \log V)$

- si las aristas pueden tener costos negativos, debemos usar el algoritmo de Bellman-Ford

  … el tiempo de ejecución sería $O(V^2E)$, que para grafos densos es $O(V^4)$

Podemos mejorar este último desempeño

Representaremos el grafo G por su *matriz de adyacencias* (en vez de las listas de adyacencias, que hemos usado mayoritariamente)

Si los n vértices están numerados 1, 2, …, $|V|$ (o sea, $|V| = n$),

… el input es una matriz W que representa los costos de las aristas

$W = (\omega_{ij})$, en que

$\omega_{ij} = 0$ si $i = j$

$=$ costo de la arista direccional (i, j) si $i \neq j$ y $(i, j) \in E$

$= \infty$ si $i \neq j$ y $(i, j) \notin E$

Suponemos que G no contiene ciclos de costo negativo

El algoritmo de Floyd-Warshall

El algoritmo considera los vértices intermedios de una ruta más corta

Si los vértices de G son $V = \{1, 2, \ldots, n\}$, consideremos el subconjunto $\{1, 2, \ldots, k\}$, para algún k

Para cualquier par de vértices $i, j \in V$, consideremos todas las rutas de i a j cuyos vértices intermedios están en $\{1, 2, \ldots, k\}$

… y sea p una ruta más corta entre ellas

k puede ser o no un vértice intermedio de p

Si k **no es** un vértice intermedio de p,

… entonces todos los vértices intermedios de p están en el conjunto $\{1, 2, \ldots, k-1\}$

⇒ una ruta más corta de i a j con todos los vértices intermedios en $\{1, 2, \ldots, k-1\}$ es también una ruta más corta de i a j con todos los vértices intermedios en $\{1, 2, \ldots, k\}$

Si k **es** un vértice intermedio de p, entonces descomponemos p en dos tramos:

el tramo p_1 de i a k y el tramo p_2 de k a j

$\Rightarrow$ por el principio de optimalidad, p_1 es una ruta más corta de i a k con todos los vértices intermedios en $\{1, 2, \ldots, k-1\}$

… y p_2 es una ruta más corta de k a j con todos los vértices intermedios en $\{1, 2, \ldots, k-1\}$

Sea $d_{ij}^{(k)}$ el costo de una ruta más corta de i a j, tal que todos los vértices intermedios están en $\{1, 2, \ldots, k\}$

Cuando $k = 0$, una ruta de i a j sin vértices intermedios con número mayor que 0 simplemente no tiene vértices intermedios, y tiene a lo más una arista $\Rightarrow d_{ij}^{(0)} = \omega_{ij}$

Definimos $d_{ij}^{(k)}$ recursivamente por

$$d_{ij}^{(k)} = \omega_{ij} \qquad \text{si } k = 0$$

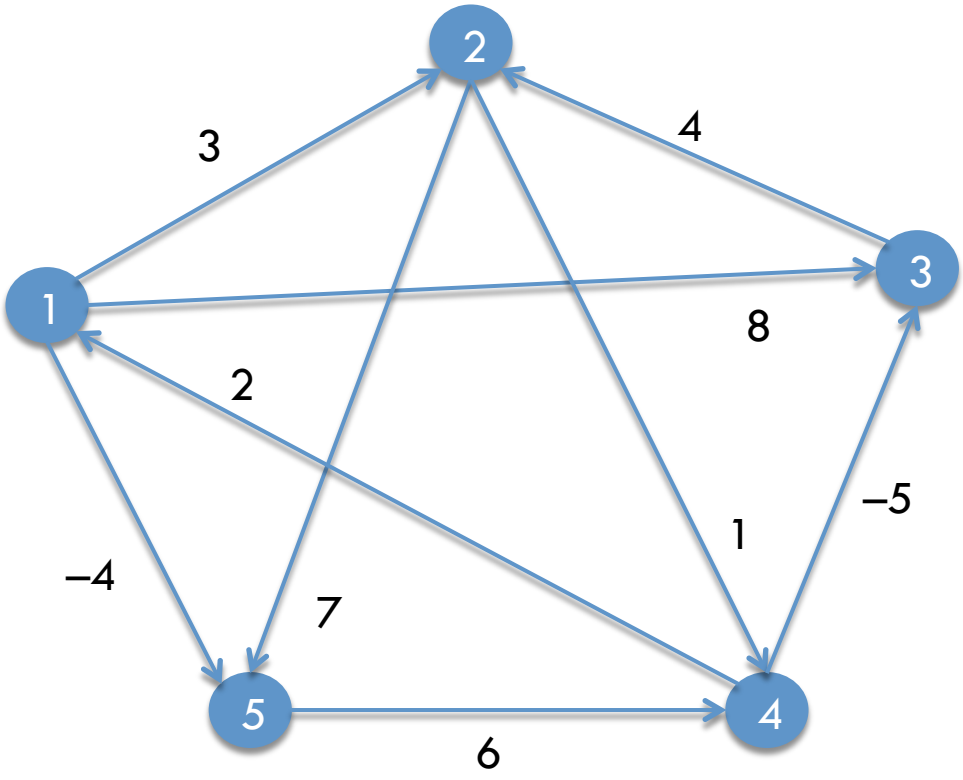
$$= \min(d_{ij}^{(k-1)} , d_{ik}^{(k-1)} + d_{kj}^{(k-1)}) \qquad \text{si } k \geq 1$$

La matriz $D^{(n)} = (d_{ij}^{(n)})$ da la respuesta final:

$d_{ij}^{(n)} = \delta(i, j)$ para todo $i, j \in V$

El algoritmo de Floyd-Warshall, *bottom-up*, toma tiempo $O(V^3)$


```
D(0) = W
for k = 1 … n:
    D(k) = (dij(k))
    for i = 1 … n:
        for j = 1 … n:
            dij(k) = min(dij(k-1), dik(k-1)+dkj(k-1))
return D(n)
```



$D^{(0)} =$

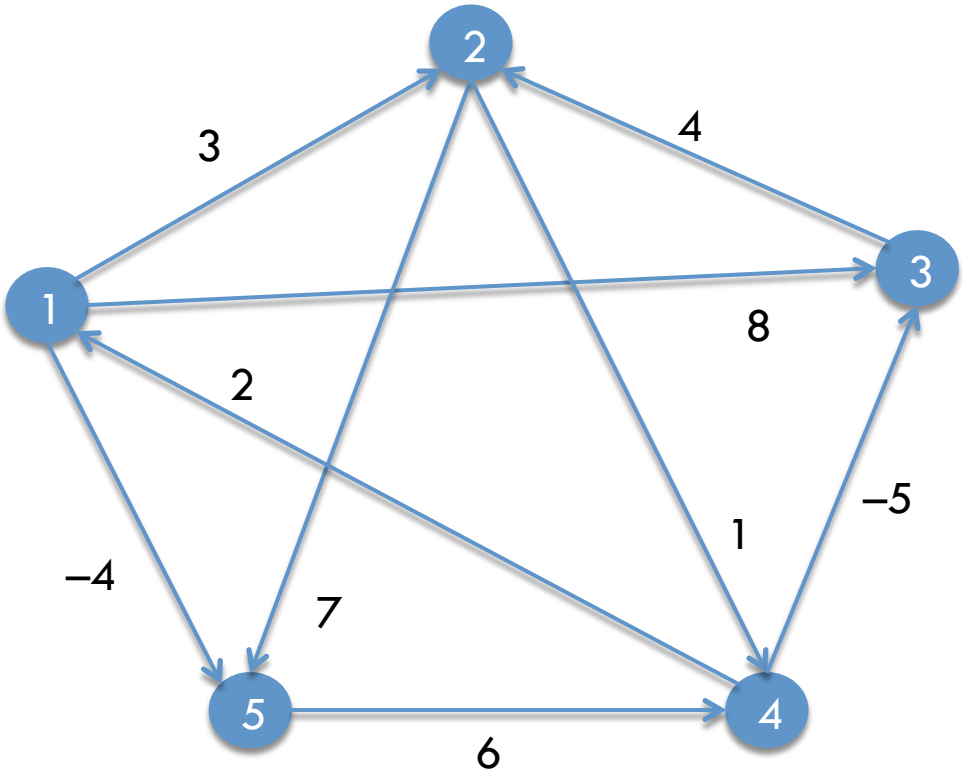
0	3	8	∞	-4
∞	0	∞	1	7
∞	4	0	∞	∞
2	∞	-5	0	∞
∞	∞	∞	6	0

$D^{(1)} =$

0	3	8	∞	-4
∞	0	∞	1	7
∞	4	0	∞	∞
2	5	-5	0	-2
∞	∞	∞	6	0

$D^{(2)} =$

0	3	8	4	-4
∞	0	∞	1	7
∞	4	0	5	11
2	5	-5	0	-2
∞	∞	∞	6	0



$D^{(3)} =$

0	3	8	4	-4
∞	0	∞	1	7
∞	4	0	5	11
2	-1	-5	0	-2
∞	∞	∞	6	0

$D^{(4)} =$

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

$D^{(5)} =$

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