Programación dinámica

Rutas más cortas en grafos direccionales

IIC2133

Yadran Eterovic

2021-1

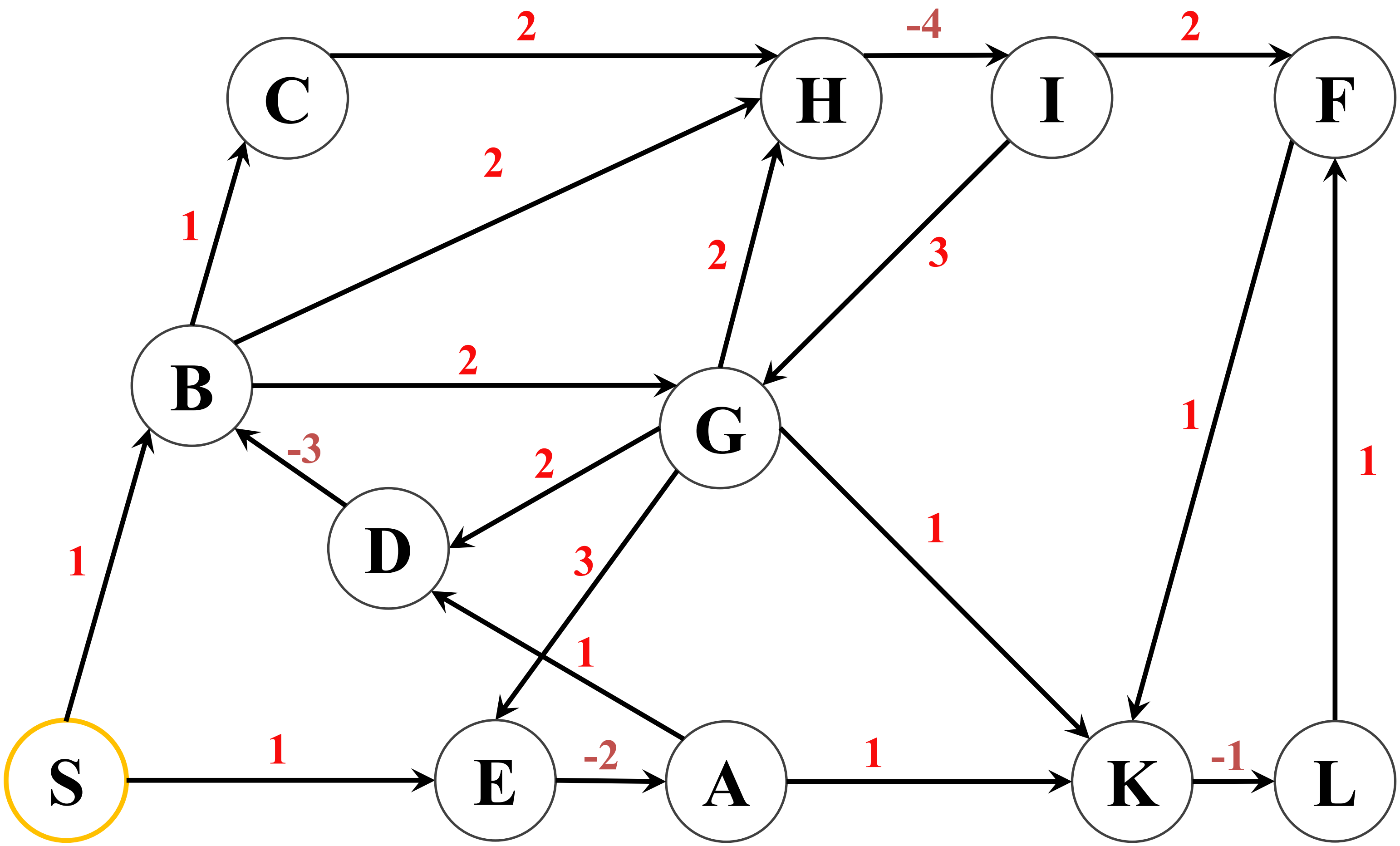
Rutas más cortas desde un nodo (a todos los otros nodos)

Si los costos de las aristas son todos ≥ 0, entonces podemos emplear el algoritmo (codicioso) de Dijkstra, que puede implementarse de modo que corra en tiempo $O((V+E)\log V)$

En cambio, si el grafo admite aristas con costos negativos

… tenemos que "empezar de nuevo" (a buscar un algoritmo):

- p.ej., en el grafo de la próxima diap., si aplicamos Dijkstra desde el nodo S, vamos a encontrar la ruta de S a B con costo 1 (la propia arista (S,B))
- … pero la ruta más corta de S a B es $\langle S, E, A, D, B\rangle$, con costo -3



C
H
I
F
B
G
D
S
E
A
K
L
2
-4
2
2
1
2
3
2
-3
2
1
1
1
3
1
1
1
-2
1
-1

Lo que sí es cierto es que la ruta ⟨*S*, *B*⟩, con costo 1, es la ruta más corta de *S* a *B si sólo permitimos rutas que tengan a lo más una arista*

Nuestro nuevo enfoque va a consistir en buscar rutas más cortas **con un cierto número máximo de aristas**:

- p.ej., en el grafo anterior la ruta más corta de *S* a *B* con (a lo más) una arista es ⟨*S*, *B*⟩, de costo 1
- … la ruta ⟨*S*, *B*⟩, de costo 1, es también la ruta más corta de *S* a *B* con (a lo más) dos y también con (a lo más) tres aristas
- … pero la ruta más corta de *S* a *B* con (a lo más) cuatro aristas es ⟨*S*, *E*, *A*, *D*, *B*⟩, de costo –3

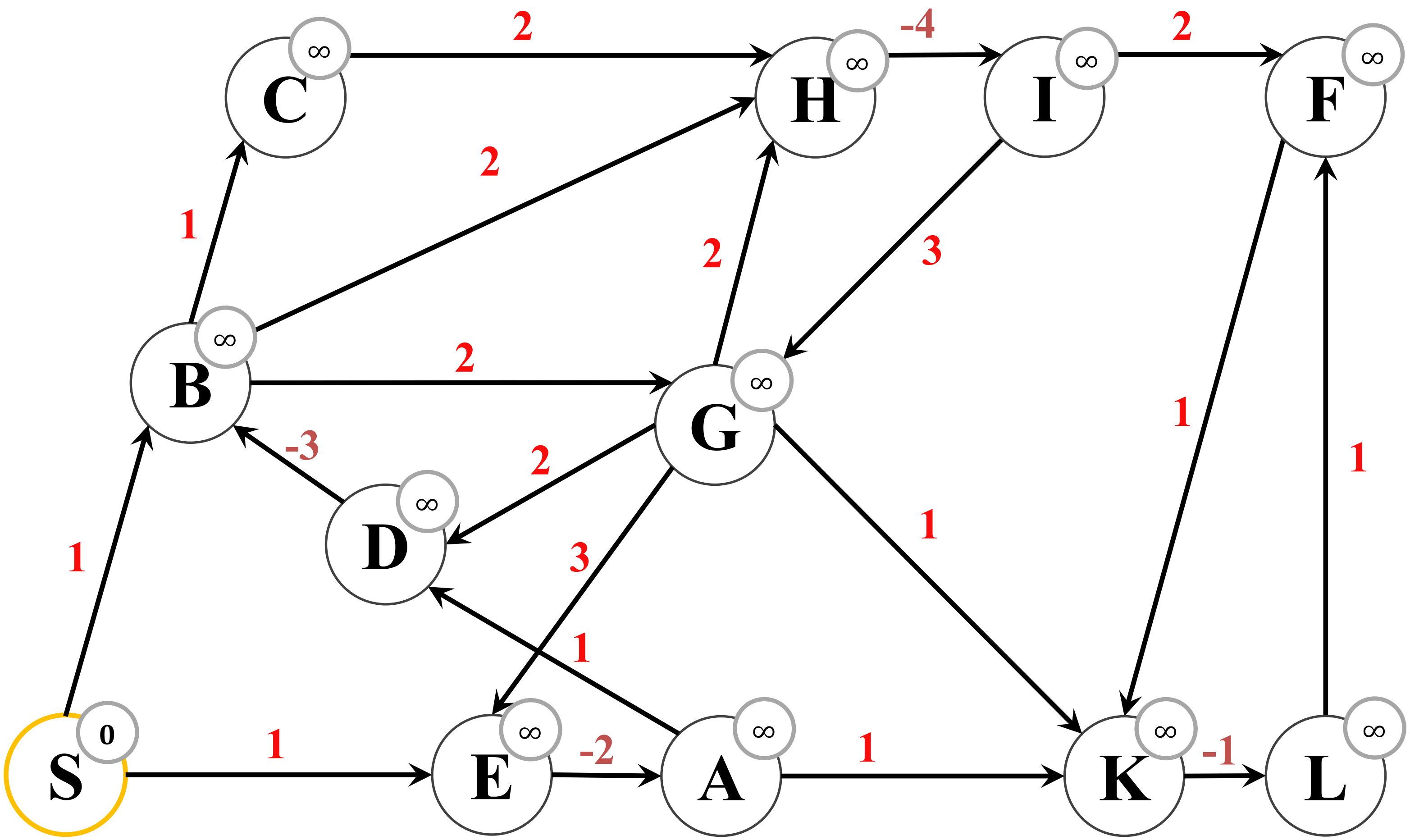
La idea es buscar primero las rutas más cortas desde *S* con a lo más una arista

… luego las rutas más cortas desde *S* con a lo más dos aristas

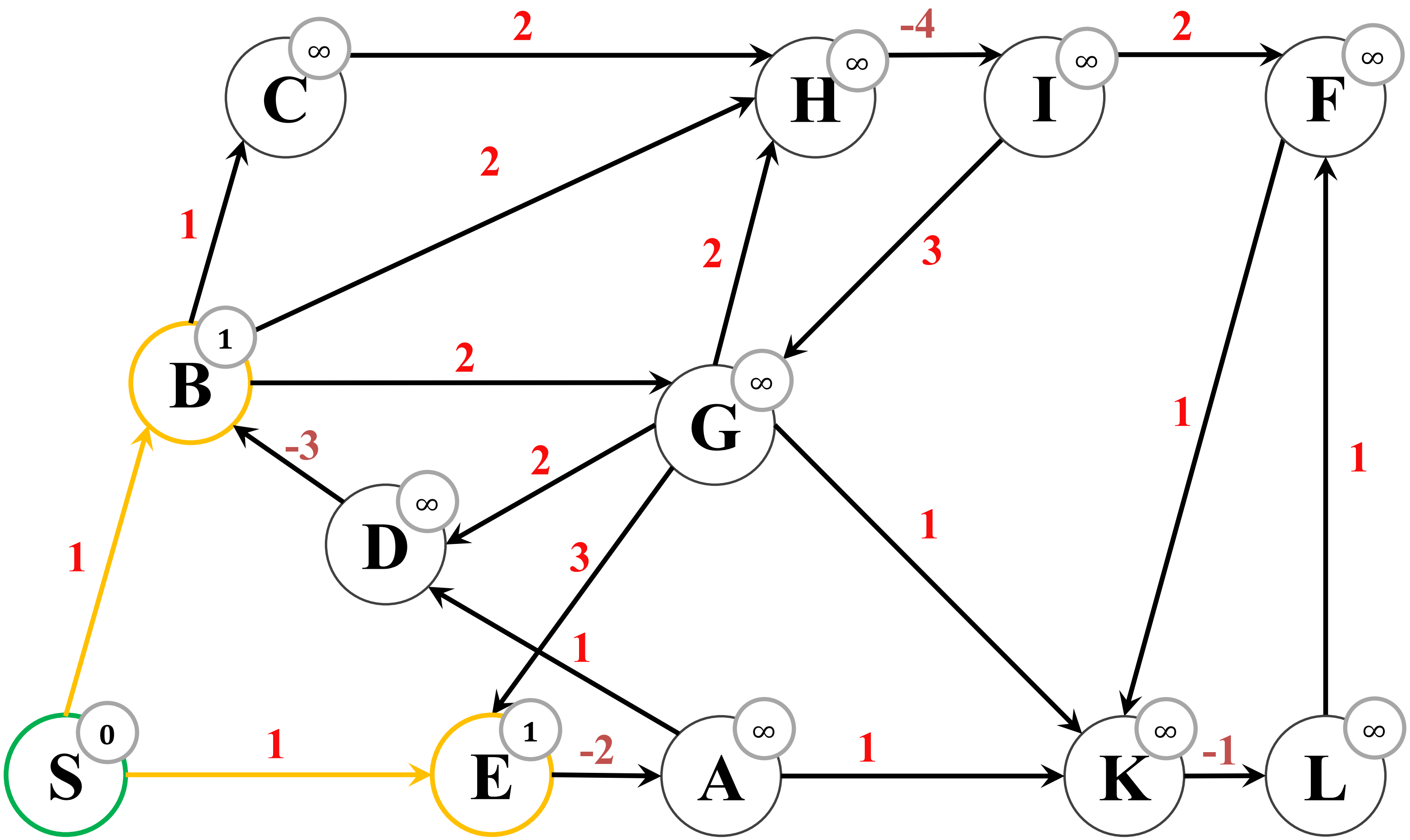
… luego las rutas más cortas desde *S* con a lo más tres aristas, etc.

Si al buscar las rutas más cortas desde *S* con a lo más *k* aristas, lo hacemos a partir de las rutas más cortas con a lo más *k*–1 aristas

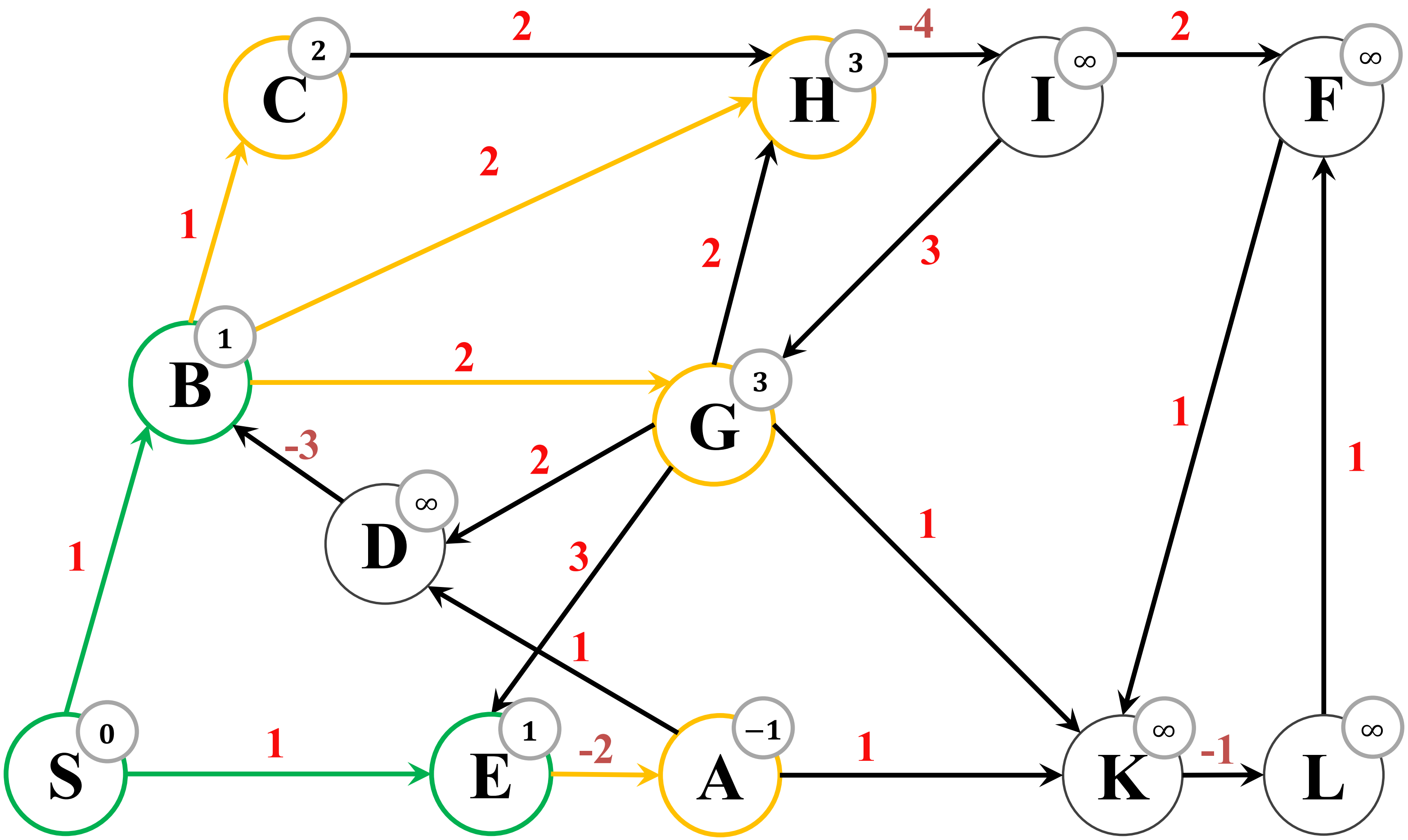
… entonces estamos empleando el enfoque de programación dinámica



C
∞
H
∞
I
∞
F
∞
B
∞
G
∞
D
∞
S
0
E
∞
A
∞
K
∞
L
∞
2
-4
2
2
1
2
3
2
-3
2
1
1
1
3
1
1
1
1
-2
1
-1



S
0
B
1
E
1
C
∞
H
∞
I
∞
F
∞
G
∞
D
∞
A
∞
K
∞
L
∞
1
1
1
2
2
2
-4
2
2
3
-3
2
3
1
1
1
1
1
-2
1
-1



C
2
H
3
I
∞
F
∞
B
1
G
3
D
∞
S
0
E
1
A
−1
K
∞
L
∞
2
-4
2
1
2
2
3
2
-3
2
1
1
3
1
1
1
1
-2
1
-1

Primero las rutas más cortas desde *S* con a lo más una arista:

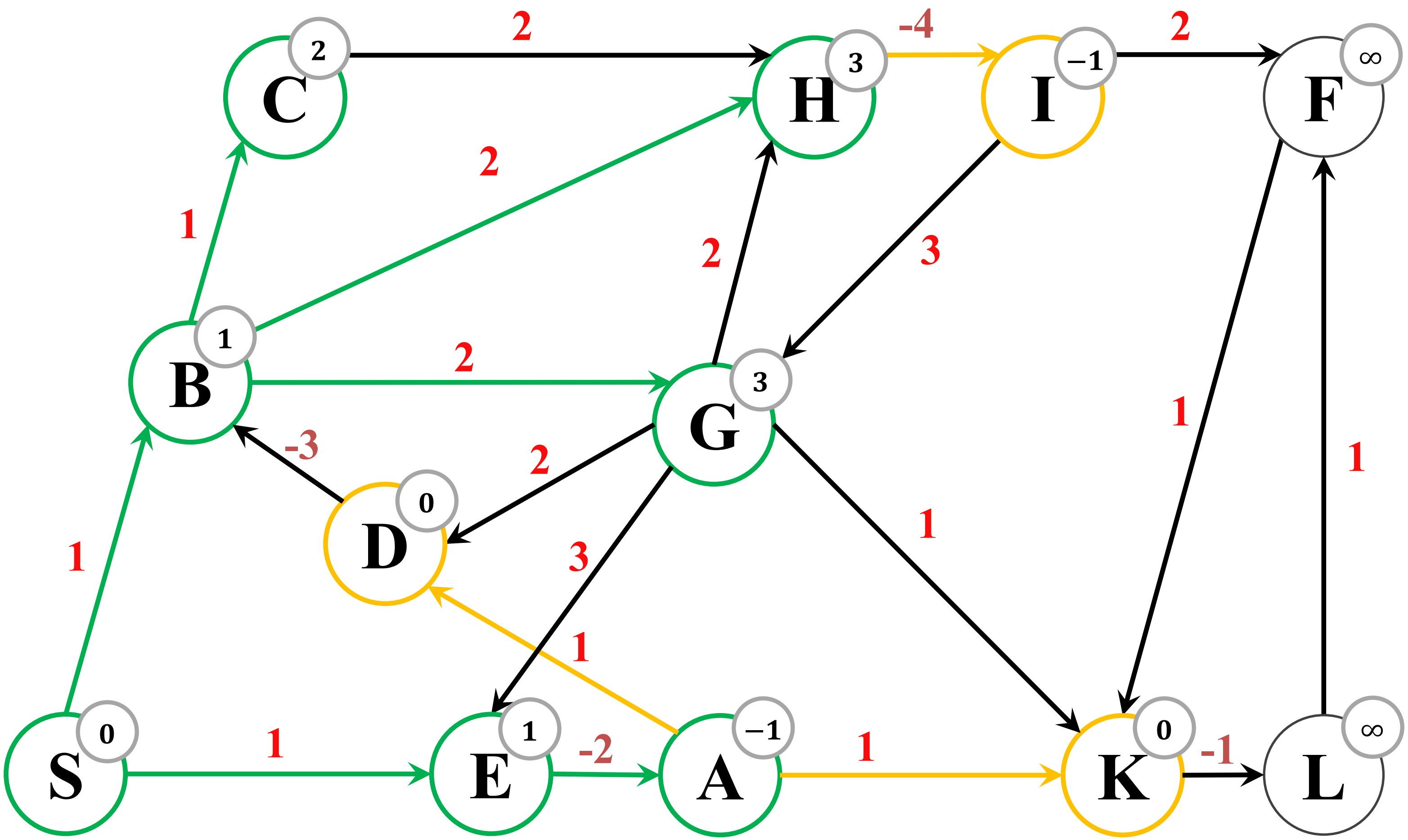
- $\langle S, B \rangle$ y $\langle S, E \rangle$, básicamente las aristas que salen de *S*
- (no hay rutas más cortas con a lo más una arista de *S* a *H*, de *S* a *G*, etc.)

… luego las rutas más cortas desde *S* con a lo más dos aristas:

- $\langle S, B, C \rangle$, $\langle S, B, H \rangle$, $\langle S, B, G \rangle$ y $\langle S, E, A \rangle$, básicamente las rutas cuya segunda arista son las aristas que salen de *B* o de *E*

… luego las rutas más cortas desde *S* con a lo más tres aristas:

- naturalmente, hay que mirar las aristas que salen de *C*, *H*, *G* y *A*
- … pero ¡cuidado! : varias de estas aristas van a parar a un mismo (nuevo) nodo o a nodos a los que ya habíamos llegado
- … en estos casos, hay que dejar sólo la mejor ruta



C
2
H
3
I
−1
F
∞
B
1
G
3
D
0
S
0
E
1
A
−1
K
0
L
∞
2
-4
2
2
1
2
3
2
1
1
-3
2
1
3
1
1
1
-2
1
-1

¿Cómo implementamos esta idea?

Recordemos que las rutas más cortas cumplen la **propiedad de subestructura óptima**:

si $u \overset{p}{\leadsto} x \rightarrow v$ es una ruta más corta de u a v

… entonces p es una ruta más corta de u a x

Entonces, podemos hacer una implementación iterativa, que a partir de S —rutas más cortas con a lo más 0 aristas— va alargando las rutas óptimas de a una arista a la vez

En la *k*-ésima iteración —al buscar rutas más cortas con a lo más *k* aristas— hacemos lo siguiente:

- para cada arista $(u,v) \in E$, hay que probar si la ruta $s \rightsquigarrow u \rightarrow v$ es más corta (más barata) que la ruta $s \rightsquigarrow v$ que tenemos hasta ahora
- las rutas $s \rightsquigarrow u$ y $s \rightsquigarrow v$ son las rutas más cortas con a lo más k–1 aristas, calculadas en la iteración anterior

Inicialmente, todos los nodos *u* están a distancia *d*[*u*] = ∞ de *S* (y sólo *S* está a distancia *d*[*S*] = 0 de *S*):

- además, y sólo con el fin de poder reconstruir las rutas al terminar el algoritmo, π`[u]` ← `null`

La prueba de la conveniencia de incluir la arista (*u*,*v*) consiste en la misma actualización que hace Dijkstra:

```
if d[v] > d[u] + costo(u,v):
        d[v] ← d[u] + costo(u,v); π[v] ← u
```

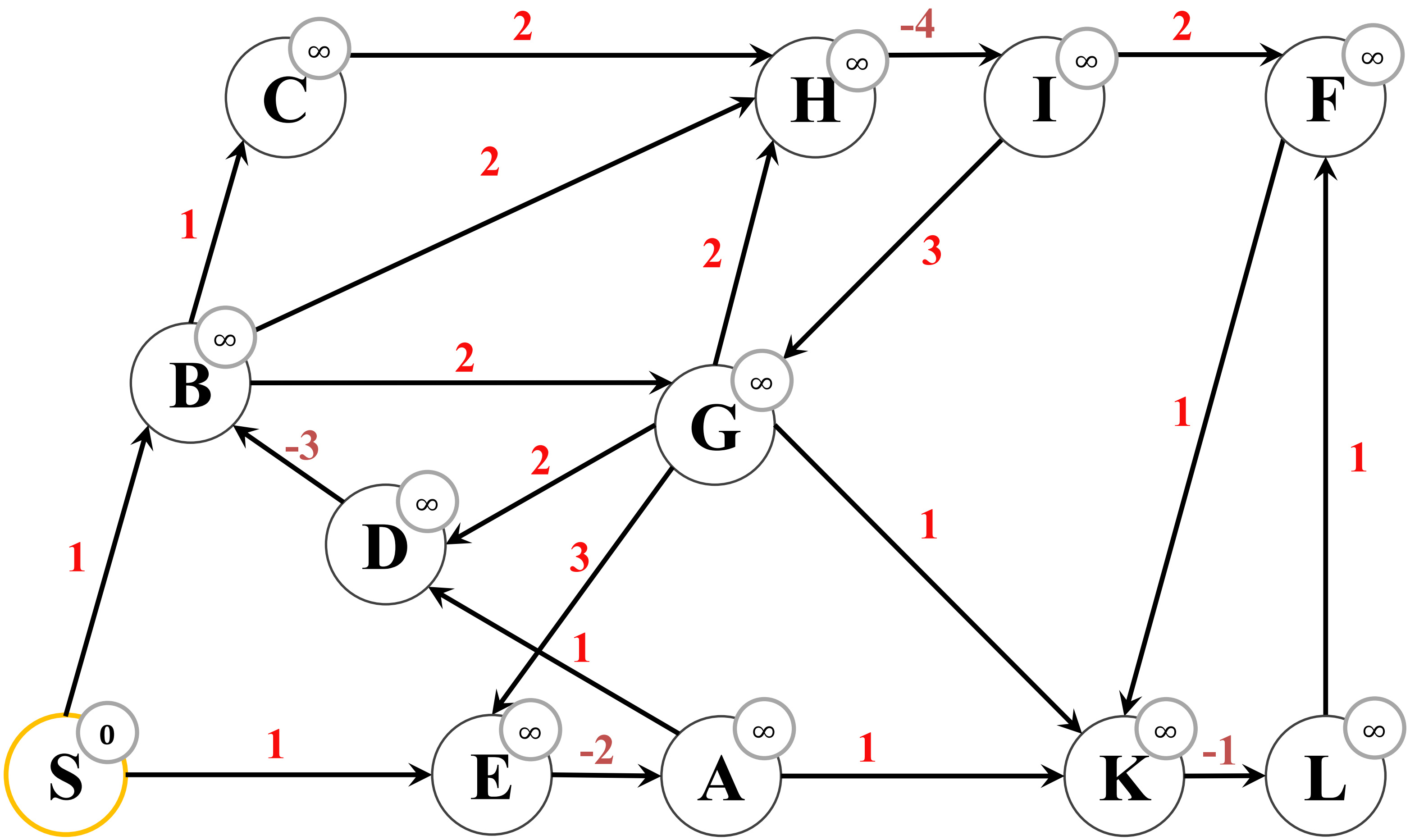
¿Cuántas iteraciones hay que hacer? $k = 1, 2, \ldots,$ **¿?**

- o ¿cuál es el máximo número de aristas que puede tener una ruta más corta?

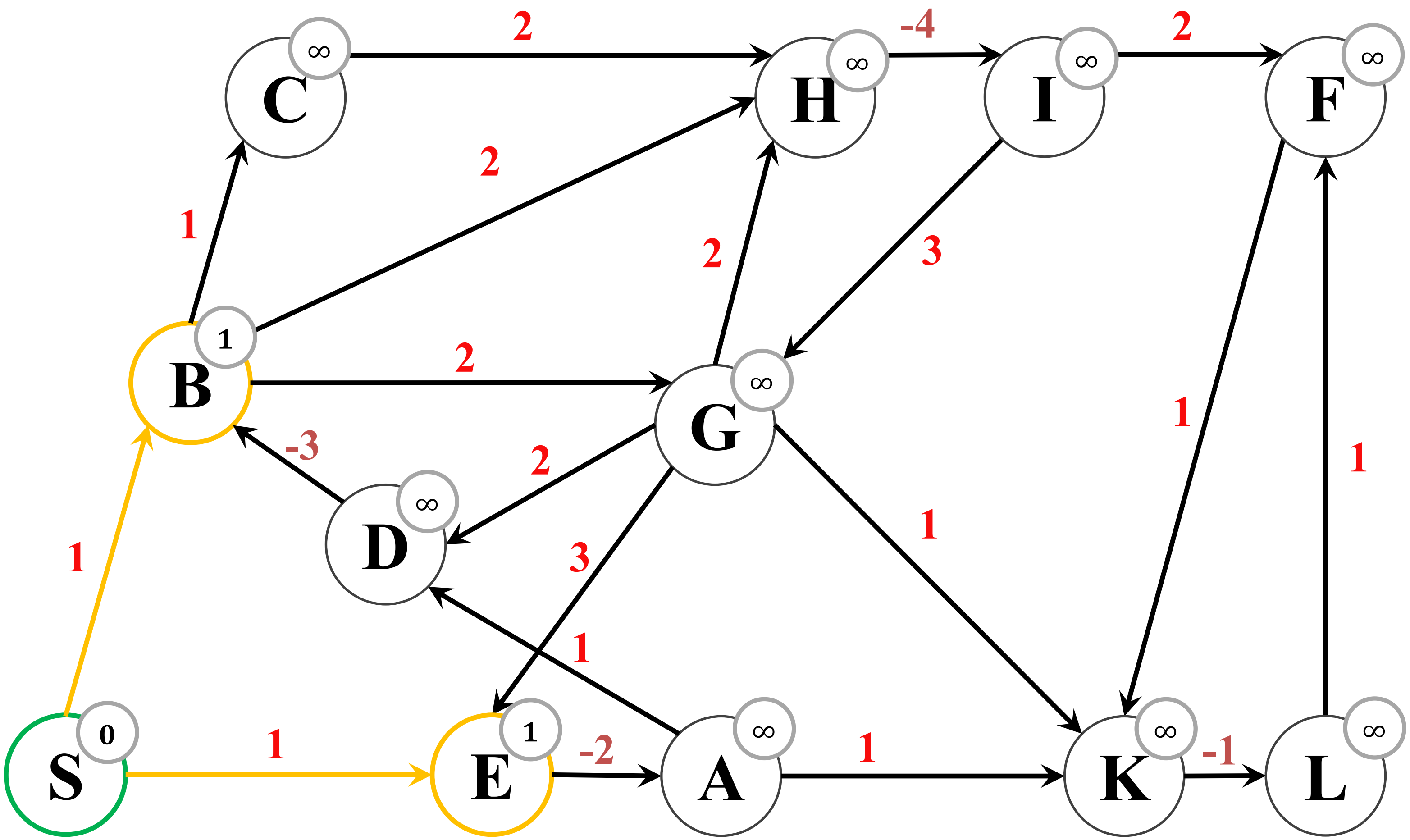
¿Es posible que una ruta más corta contenga un ciclo?

```
Bellman-Ford(s): —s es el vértice de partida
    for each u in V:
        d[u] ← ∞; π[u] ← null
    d[s] ← 0
    for k = 1 … |V|-1:
        for each (u,v) in E:
            if d[v] > d[u] + costo(u,v):
                d[v] ← d[u] + costo(u,v); π[v] ← u
```

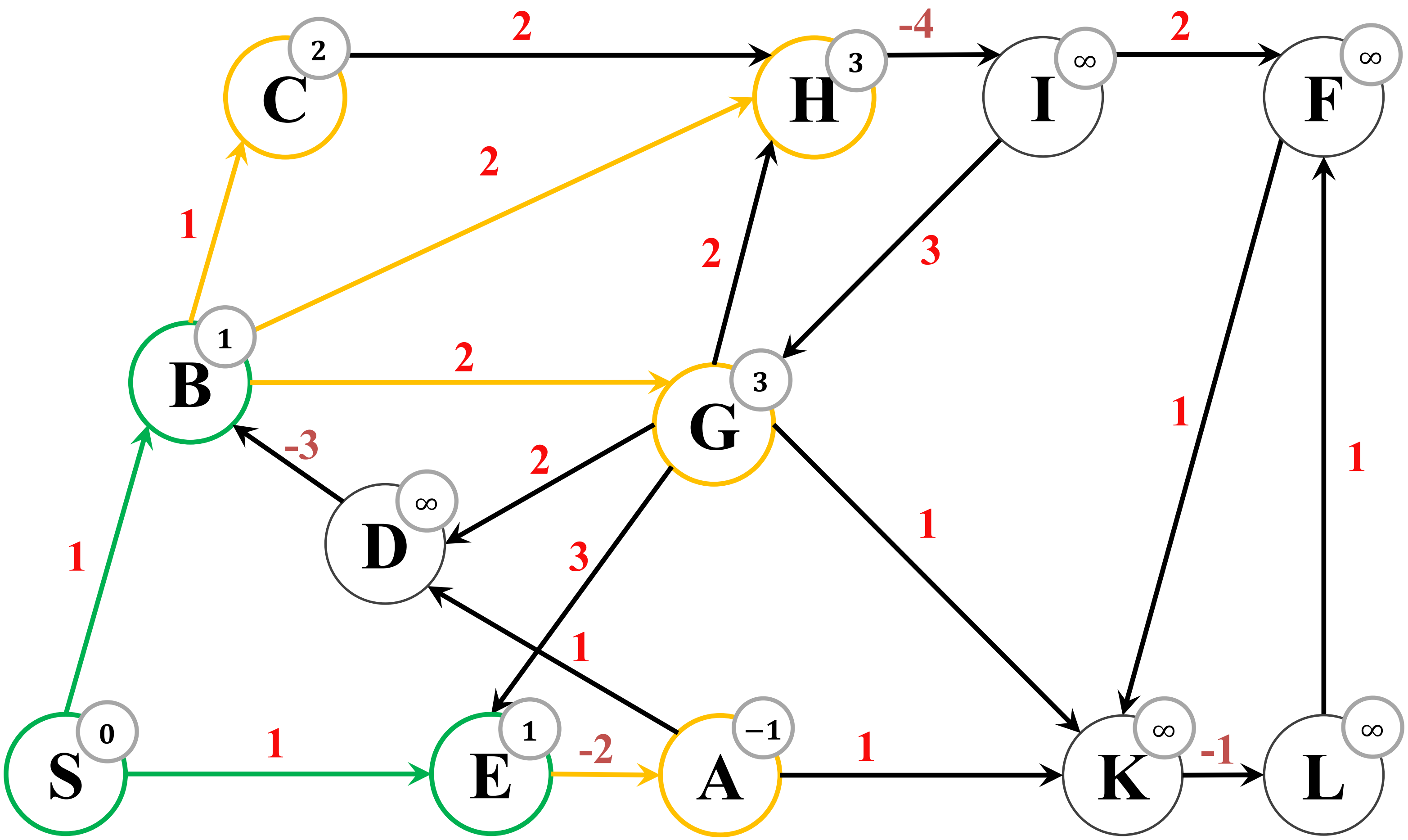
$k = 0$



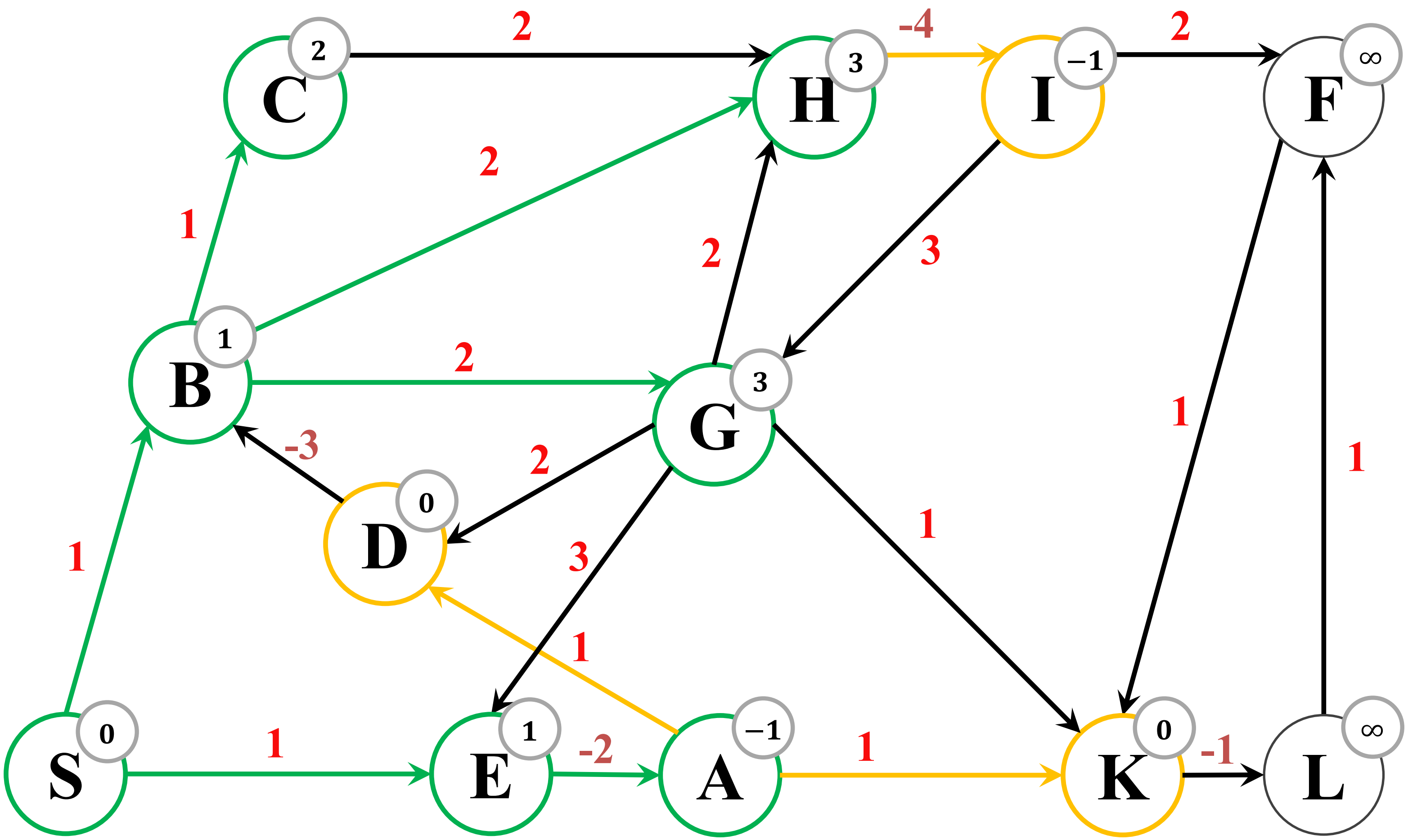
$k = 1$



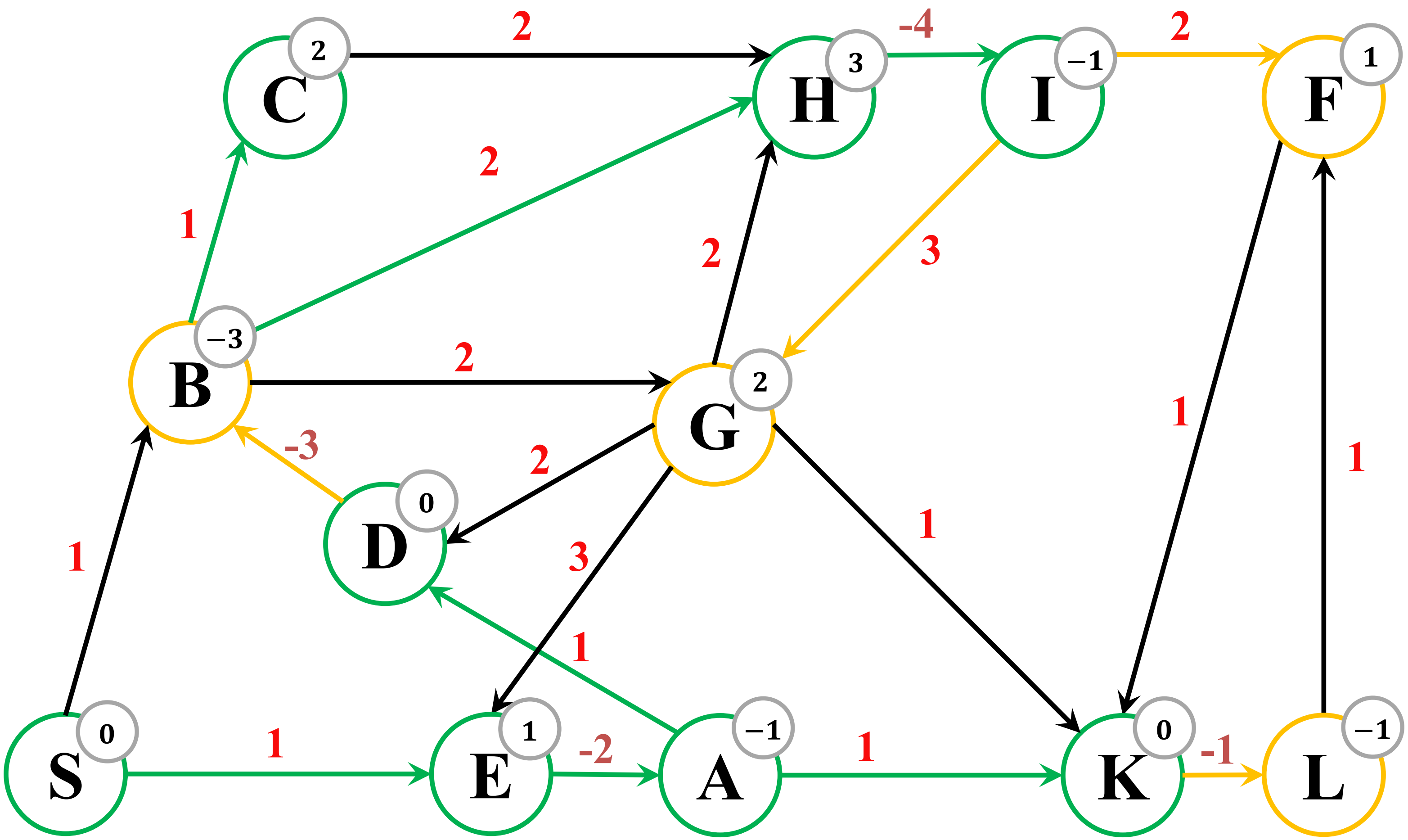
$k = 2$



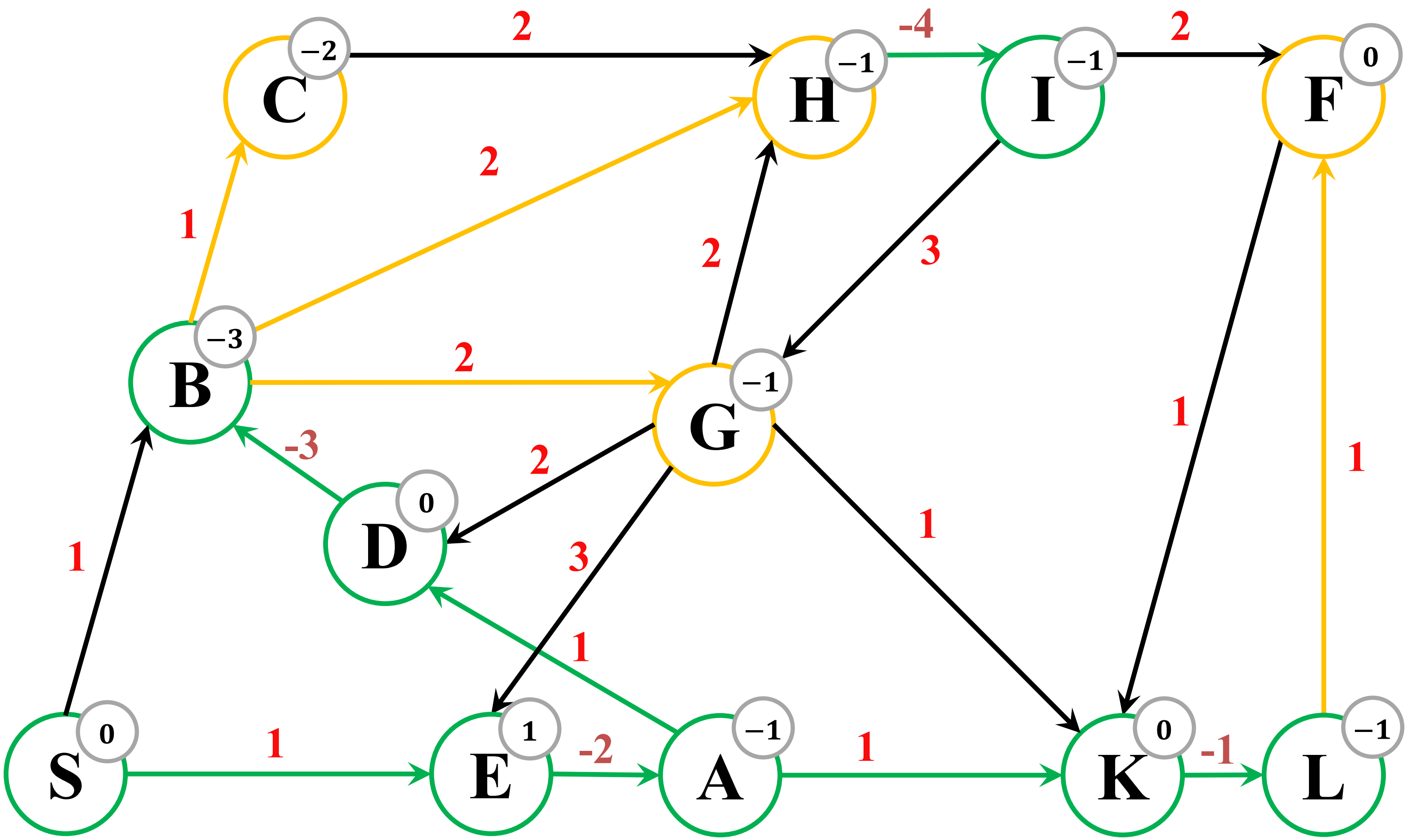
$k = 3$



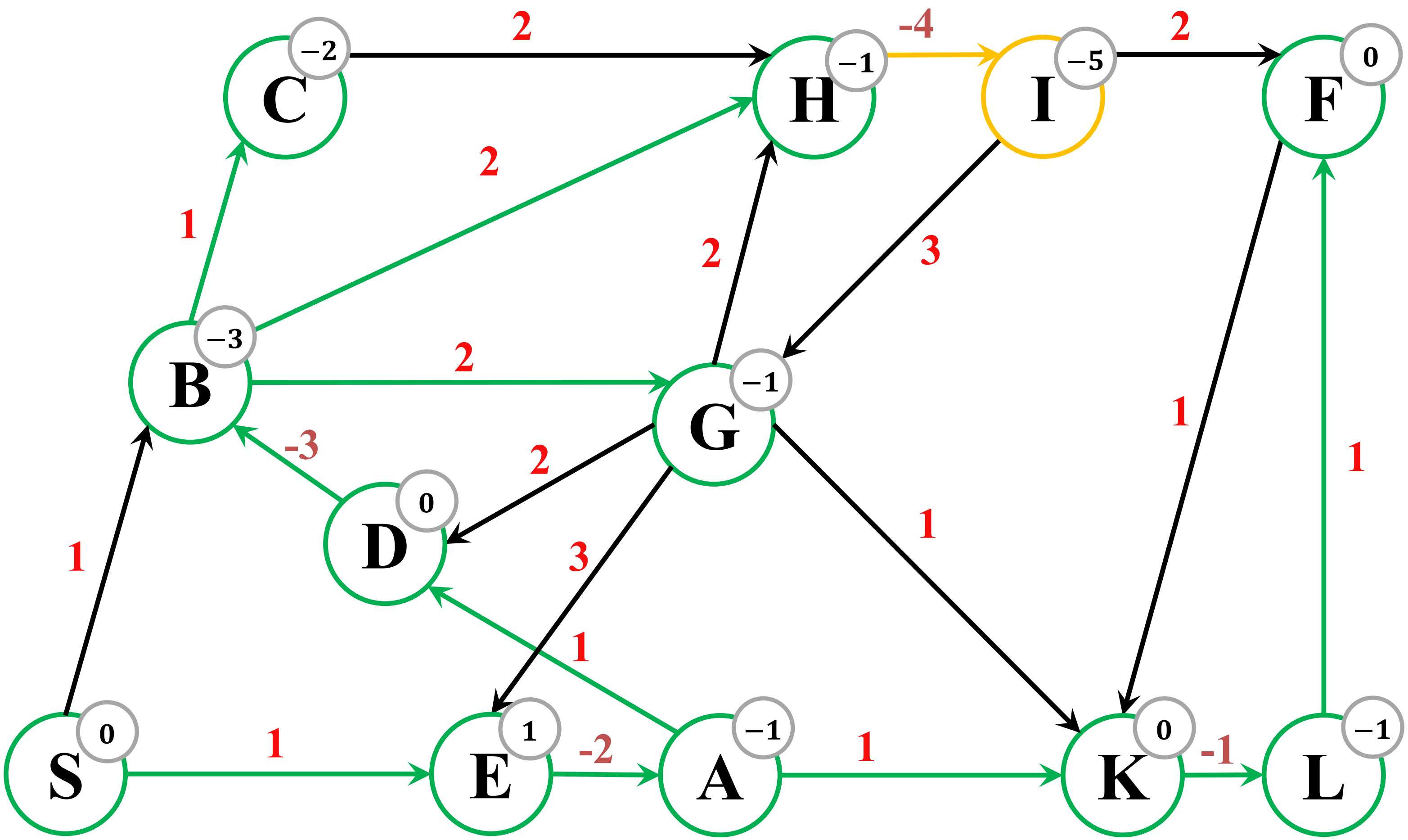
$k = 4$



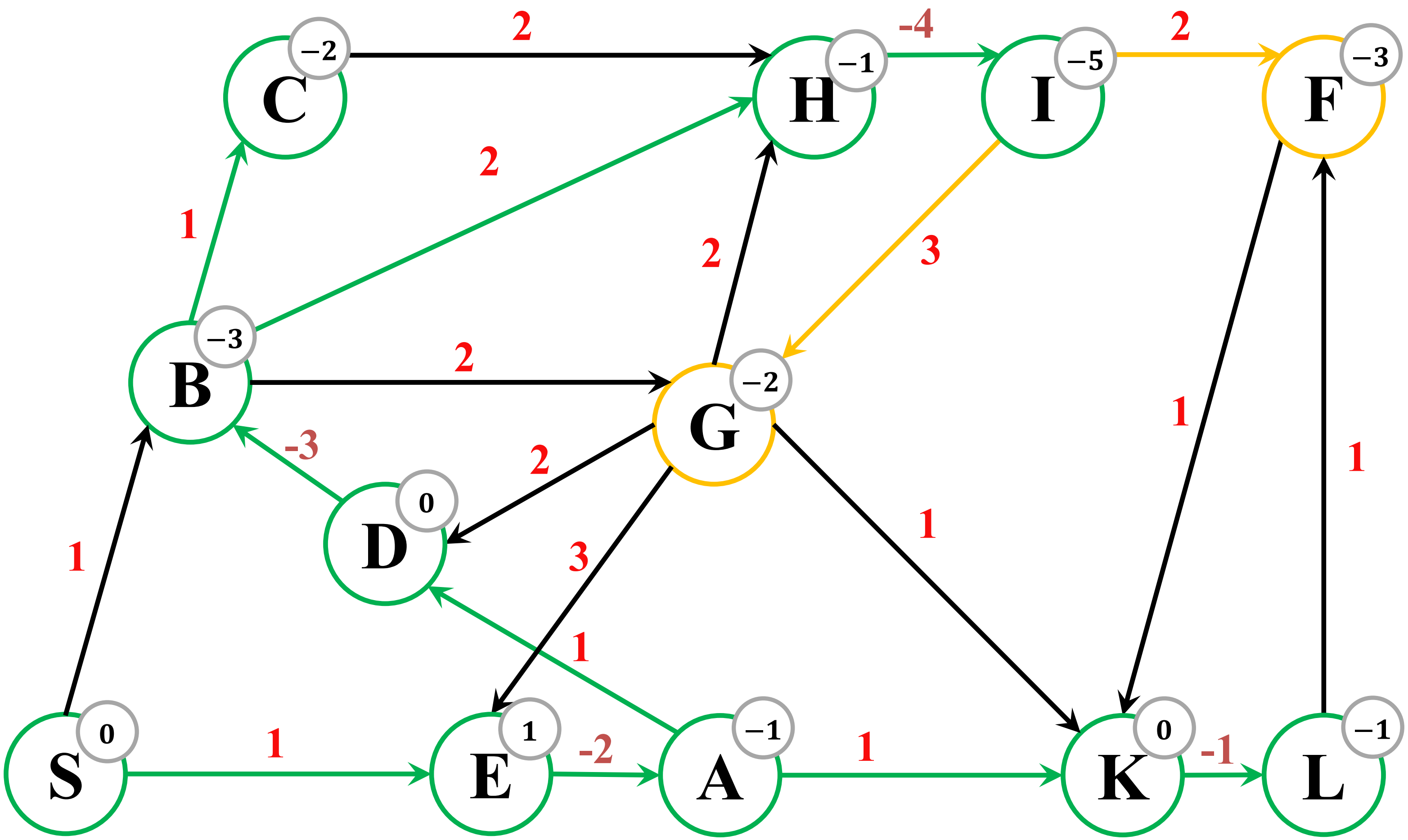
$k = 5$



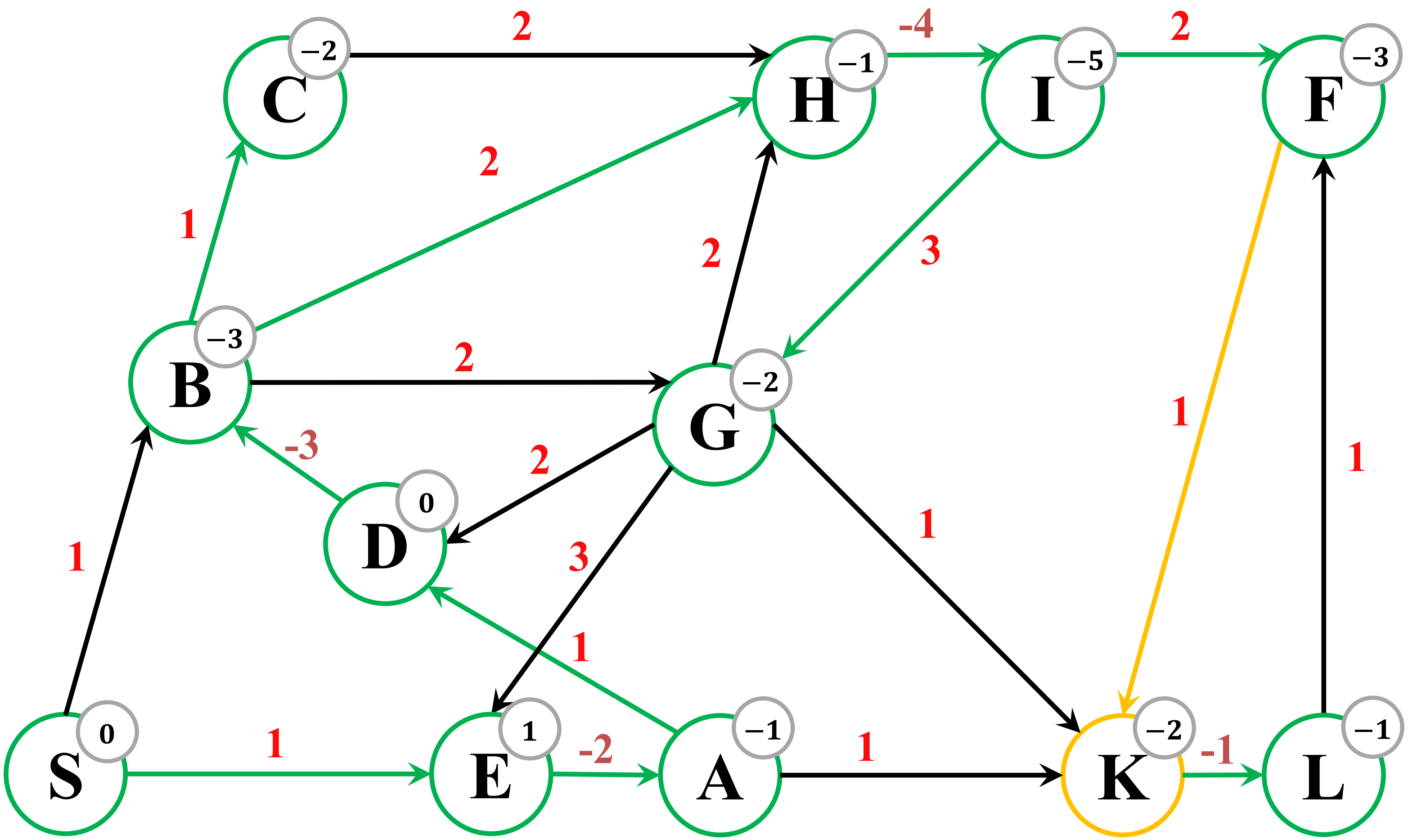
$k = 6$



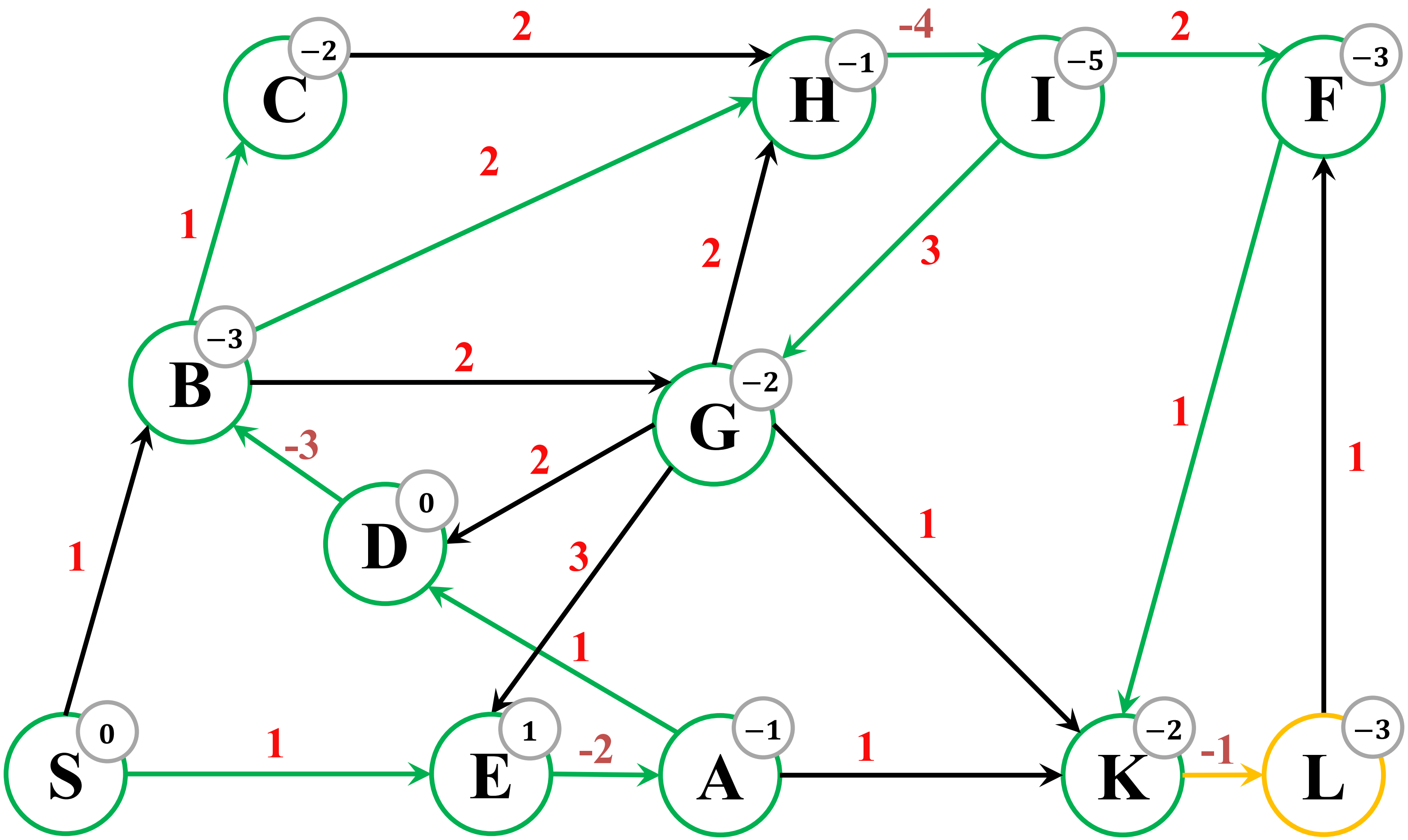
$k = 7$



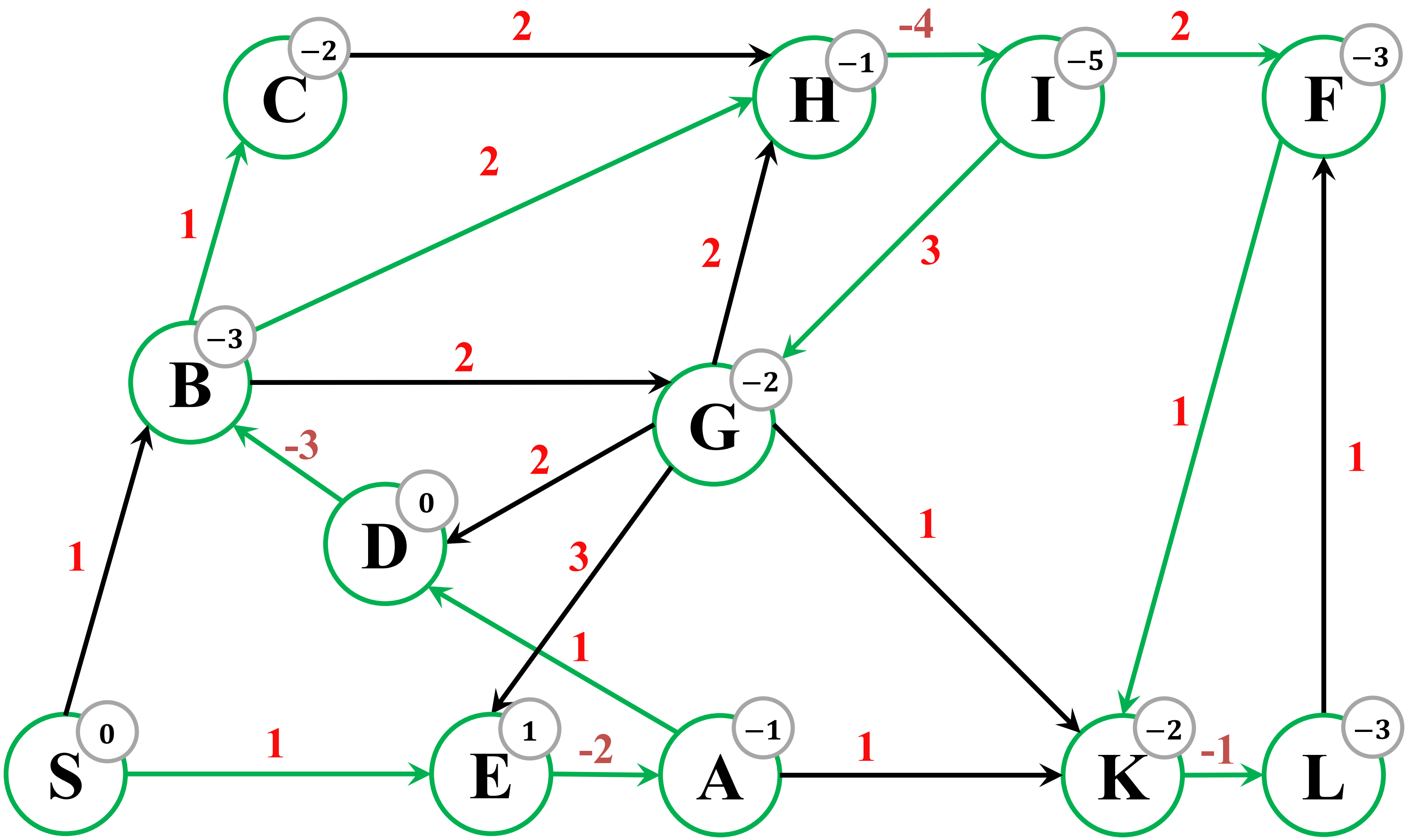
$k = 8$



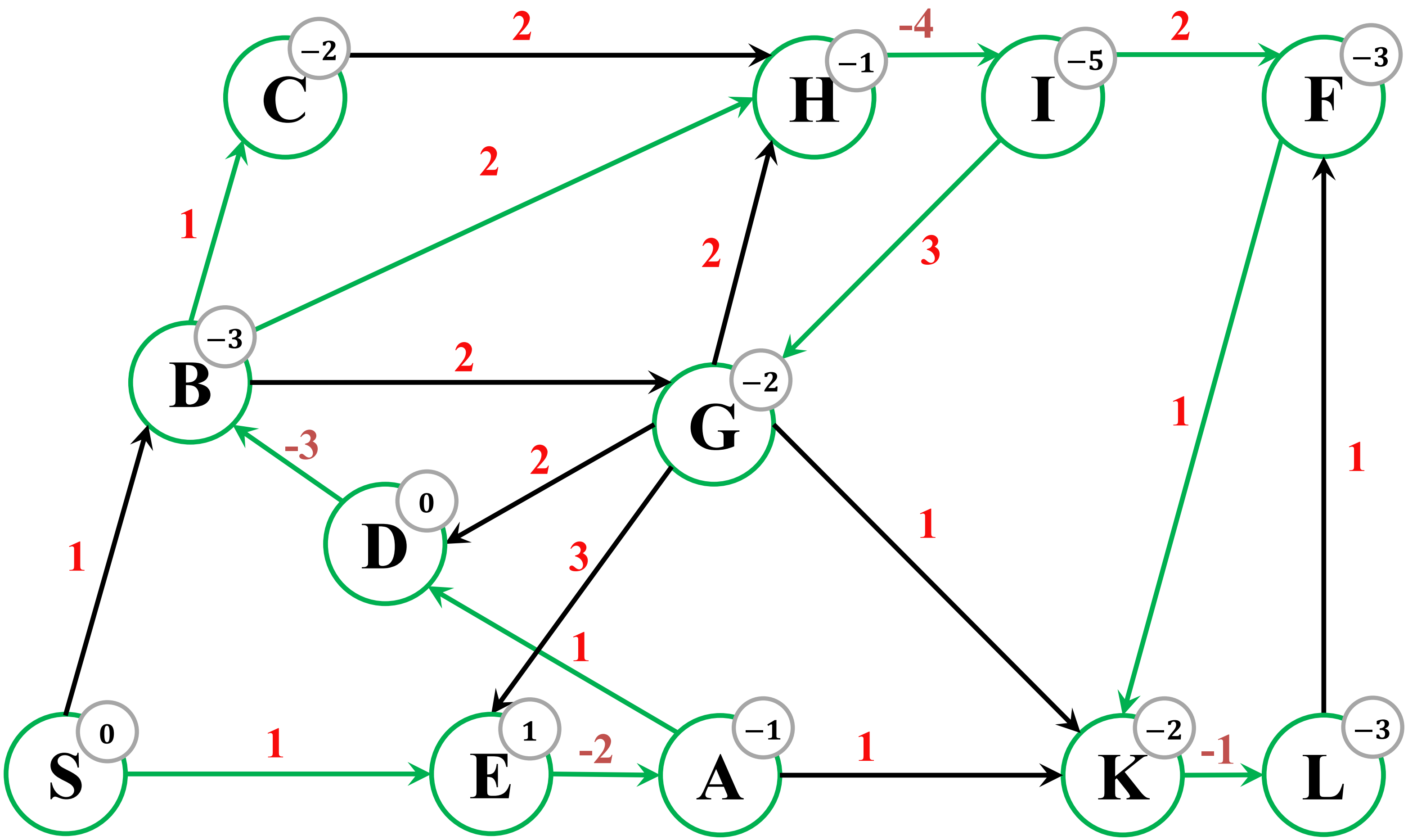
$k = 9$



$k = 10$

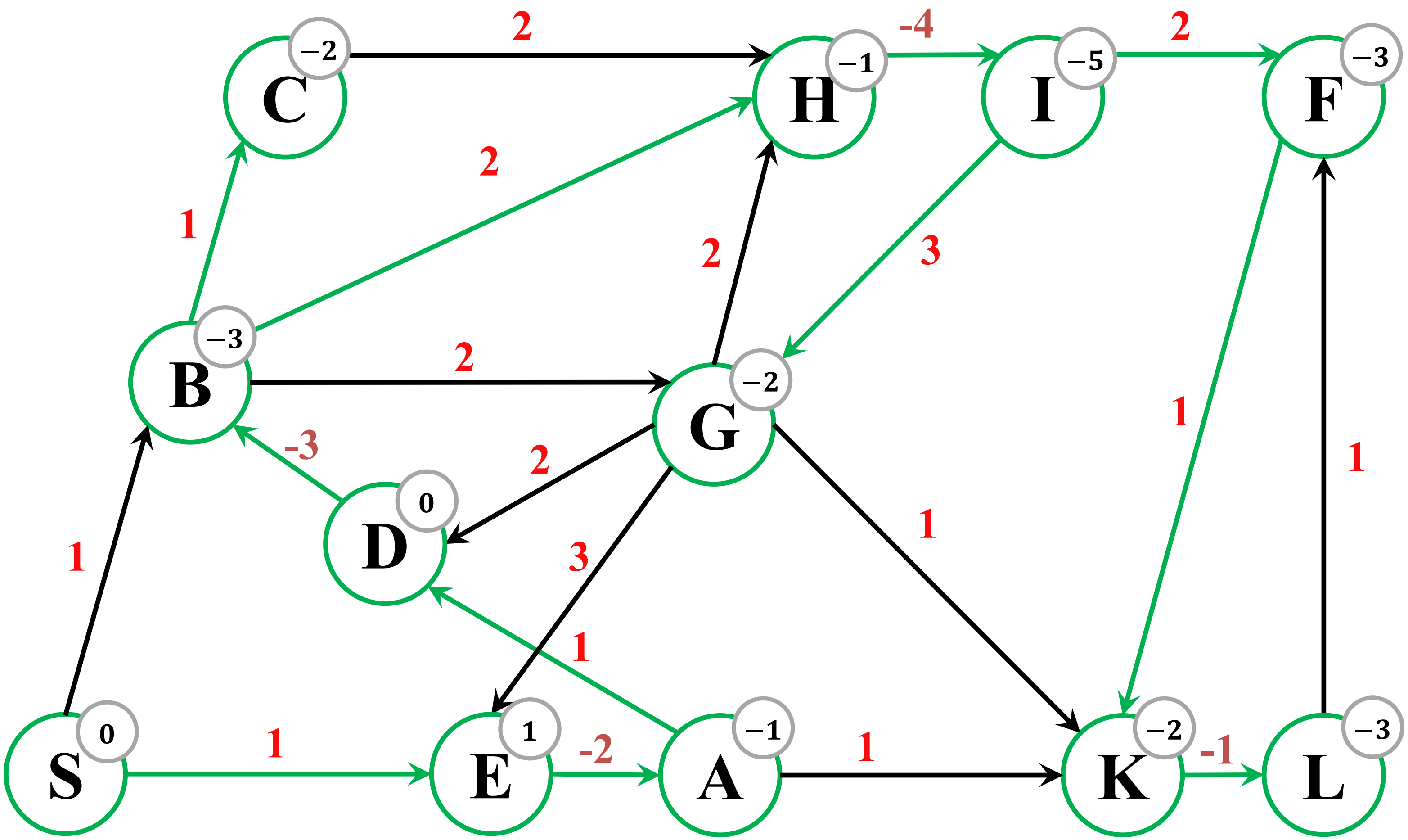


$k = 11$

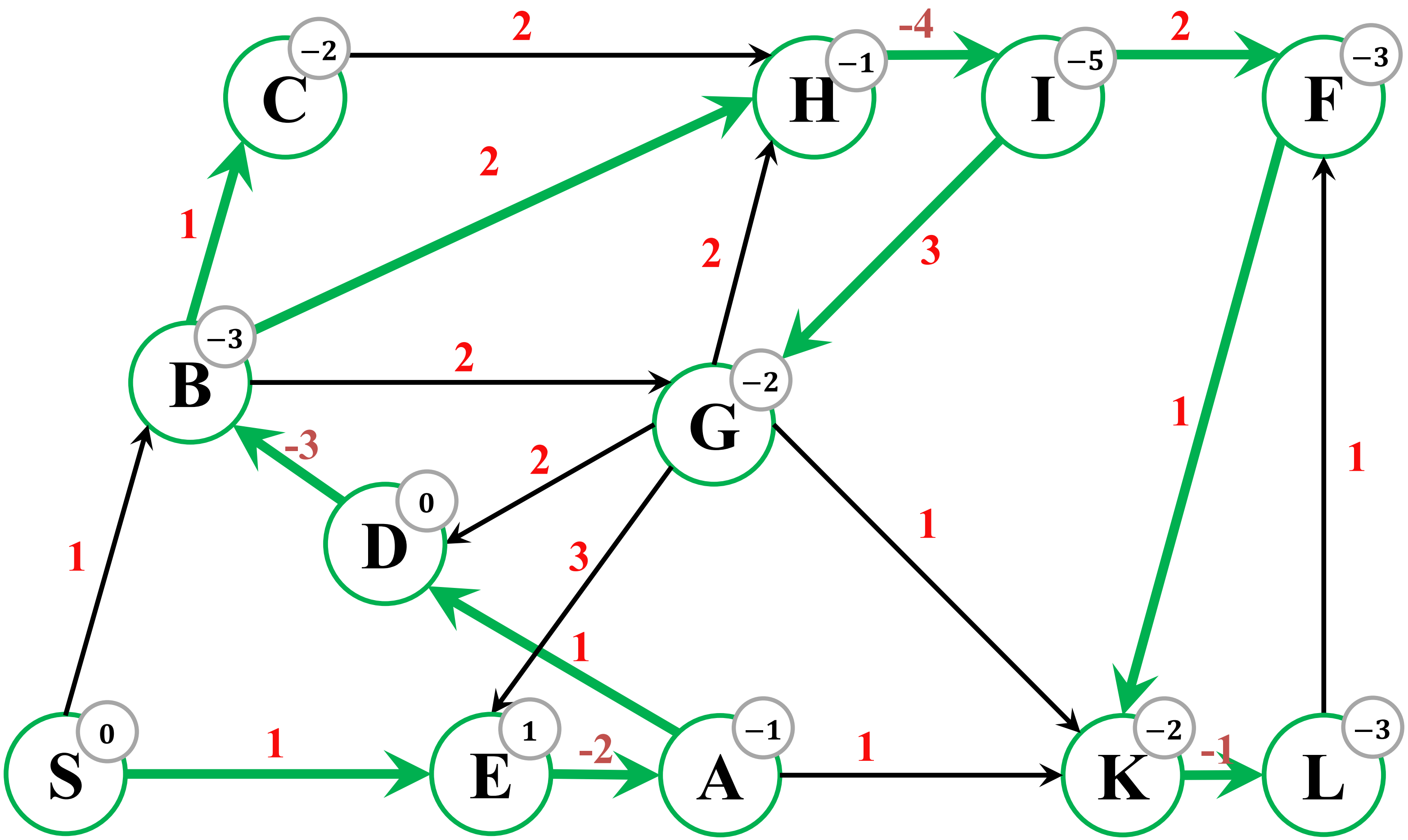


¿Cueal es la complejidad de Bellman-Ford?

¿Qué significa si hacemos una iteración adicional y efectivamente mejoramos algunas rutas?


S
0
B
−3
C
−2
D
0
E
1
A
−1
G
−2
H
−1
I
−5
F
−3
K
−2
L
−3
1
1
1
2
2
2
-3
2
3
1
-2
1
2
1
-4
2
3
1
1
-1
1

Árbol de rutas más cortas



Rutas más cortas entre todos los pares de nodos

Podemos ejecutar $|V|$ veces un algoritmo para rutas más cortas desde un nodo, una vez para cada nodo 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

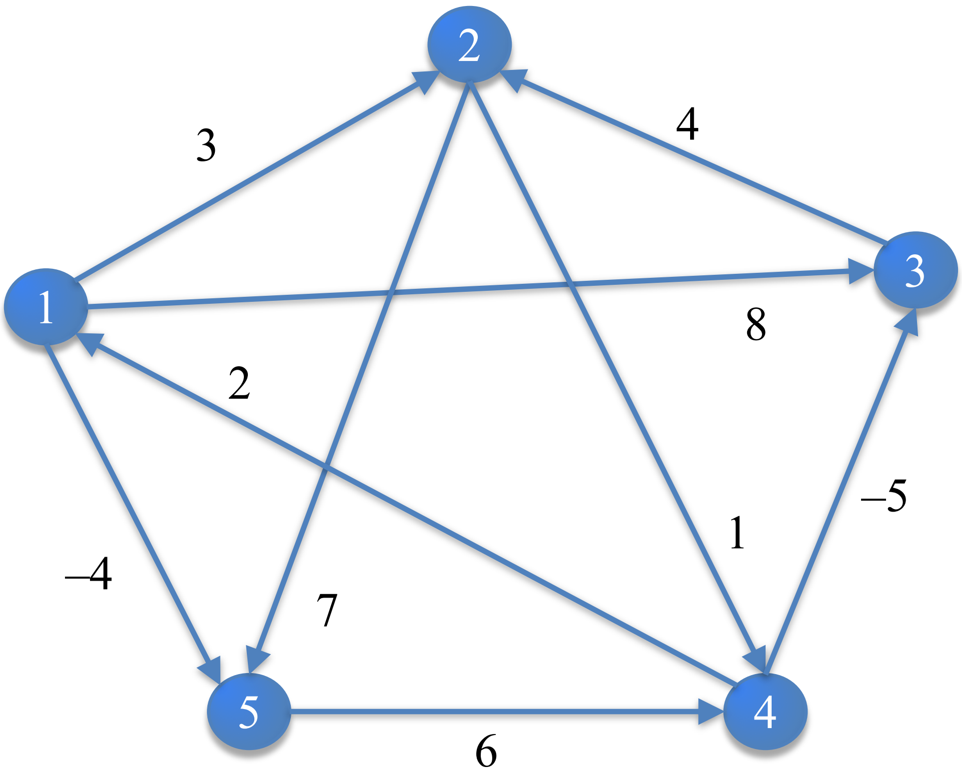
  … el tiempo de ejecución sería O(V^2E), que para grafos densos es O(V^4)

Podemos mejorar este último desempeño

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

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

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



$$W = \begin{matrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{matrix}$$

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

$$
\begin{aligned}
\omega_{ij} &= 0 && \text{si } i = j \\
&= \text{costo de la arista direccional } (i, j) && \text{si } i \neq j \text{ y } (i, j) \in E \\
&= \infty && \text{si } i \neq j \text{ y } (i, j) \notin E
\end{aligned}
$$

Suponemos que G **no contiene ciclos de costo negativo**

El algoritmo de Floyd-Warshall

El algoritmo considera los nodos intermedios de una ruta más corta

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

Para cualquier par de nodos i, $j \in V$,

… consideremos todas las rutas de i a j cuyos nodos intermedios están todos tomados del conjunto $\{1, 2, \ldots, k\}$

… y sea $\boldsymbol{p}$ una ruta más corta entre ellas

El nodo k puede ser o no un nodo (intermedio) de $\boldsymbol{p}$

Si k **no es** un nodo de ***p***,

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

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

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

Si k **es** un nodo de ***p***, entonces podemos dividir ***p*** en dos tramos:

el tramo p_1 de i a k

… y el tramo p_2 de k a j

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

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

Cuando $k = 0$,

… una ruta de *i* a *j* sin nodos intermedios con número mayor que 0

… simplemente no tiene nodos intermedios,

… y por lo tanto tiene a lo más una arista $\Rightarrow d_{ij}^{(0)} = \omega_{ij}$

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

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

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

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:
    sea D^(k) = (d_ij^(k)) una nueva matriz
    for i = 1 … n:
        for j = 1 … n:
            d_ij^(k) = min(d_ij^(k-1), d_ik^(k-1)+d_kj^(k-1))
return D^(n)
```

$D^{(0)} = W$

for $k = 1 \ldots n$:

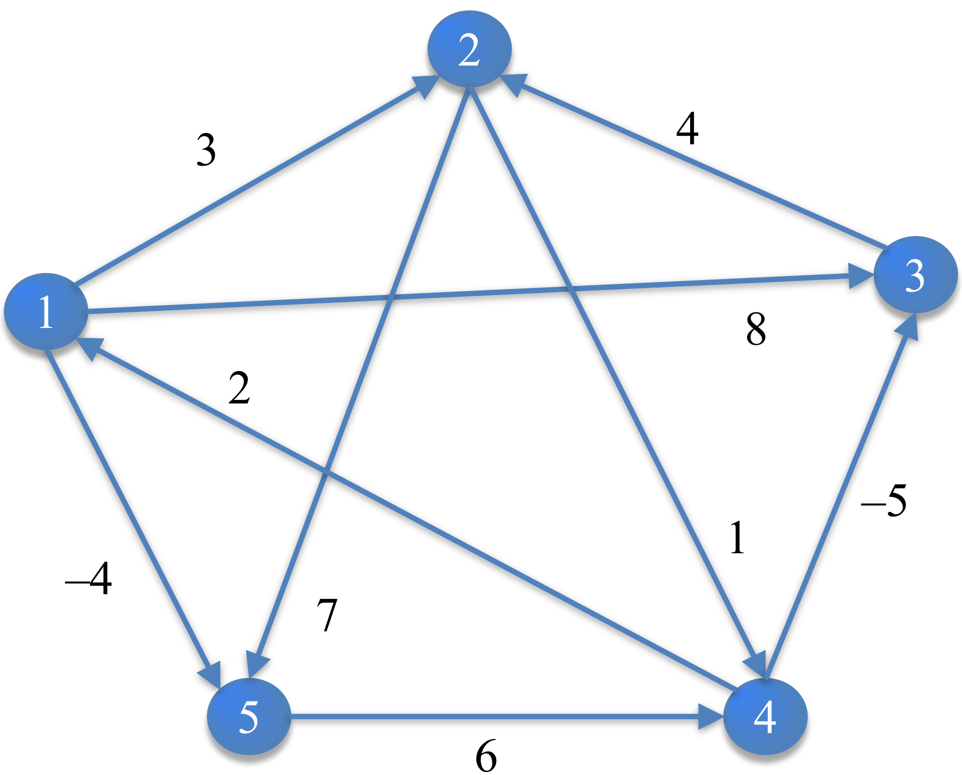
  sea $D^{(k)} = (d_{ij}^{(k)})$ *una nueva matriz*

  for $i = 1 \ldots n$:

    for $j = 1 \ldots n$:

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

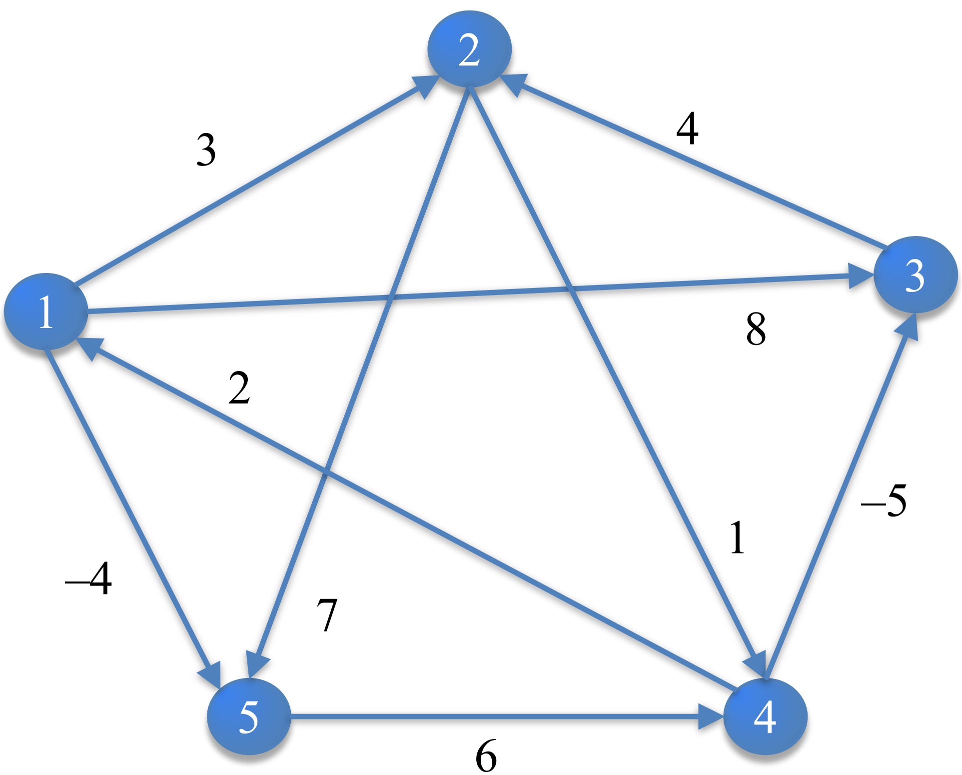
return $D^{(n)}$



$$D^{(0)} = \begin{matrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{matrix}$$

$$D^{(1)} = \begin{matrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{matrix}$$

$$D^{(2)} = \begin{matrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{matrix}$$



$$D^{(3)} = \begin{matrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{matrix}$$

$$D^{(4)} = \begin{matrix} 0 & 3 & -1 & 4 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{matrix}$$

$$D^{(5)} = \begin{matrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{matrix}$$