

# Algoritmo de Bellman-Ford

Clase 26

IIC 2133 - Sección 2

Prof. Mario Droguett

# Sumario

**Introducción**

Algoritmo de Bellman-Ford

Cierre

# Rutas en viajes 2.0

Consideremos el problema de planificar un viaje en auto desde  $A$  a  $B$

- Modelo de **grafo dirigido con costos**
- Ya sabemos resolver este problema...
- ... siempre que los costos sean **no negativos**

Algoritmo de Dijkstra resuelve el problema que ya conocemos

# Algoritmo de Dijkstra para rutas más cortas

Dijkstra( $s$ ):

for  $u \in V - \{s\}$  :

$u.color \leftarrow \text{blanco}$ ;  $d[u] \leftarrow \infty$ ;  $\pi[u] \leftarrow \emptyset$

$s.color \leftarrow \text{gris}$ ;  $d[s] \leftarrow 0$ ;  $\pi[s] \leftarrow \emptyset$

$Q \leftarrow$  cola de **prioridades** vacía (Min Heap)

Insert( $Q, s$ )

while  $Q$  no está vacía :

$u \leftarrow \text{Extract}(Q)$

    for  $v \in \alpha[u]$  :

        if  $v.color = \text{blanco} \vee v.color = \text{gris}$  :

            if  $d[v] > d[u] + \text{cost}(u, v)$  :

$d[v] \leftarrow d[u] + \text{cost}(u, v)$ ;  $\pi[v] \leftarrow u$

                DecreaseKey( $Q, v, d[v]$ )

        if  $v.color = \text{blanco}$  :

$v.color \leftarrow \text{gris}$ ; Insert( $Q, v$ )

$u.color \leftarrow \text{negro}$

# Correctitud de Dijkstra

## Demostración

### Finitud

Es claro que el algoritmo termina, pues no visita nodos ya descubiertos y cada arista es revisada a lo más una vez. Como el grafo es finito, el algoritmo es finito.

### Correctitud

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

Probaremos la correctitud del algoritmo demostrando la siguiente propiedad

$P(n) :=$  al inicio de la  $n$ -ésima iteración del **while**  
el nodo  $u$  extraído de  $Q$  cumple  $d[u] = \delta(s, u)$

Lo haremos por inducción sobre  $n$ .

# Correctitud de Dijkstra

## Demostración

**C.B.** Para  $i = 1$ , tenemos que se extrae  $s$ . El óptimo es  $\delta(s, s) = 0$  y corresponde con el costo almacenado  $d[s] = 0$ .

**H.I.** Suponemos que al inicio de la  $k$ -ésima iteración, el nodo extraído cumple la propiedad, para  $k < n$ .

**T.I.** Probaremos el resultado para la iteración  $n$ . Supongamos que esta iteración es tal que  $u$  extraído es el primer nodo tal que

$$d[u] \neq \delta(s, u)$$

Llegaremos a una contradicción, que probará que no hay tal  $u$ , i.e. todos los elementos cumplen la propiedad pedida.

# Correctitud de Dijkstra

## Demostración

Para argumentar que existe un camino de  $s$  hasta  $u$ ,

- Si no existe tal camino, el costo ideal es  $\delta(s, u) = \infty$
- Pero este es el valor inicial  $d[u] = \infty$
- Como solo se puede reducir el costo al encontrar caminos desde  $s$ , se contradice el supuesto de que no hay camino

Sea  $p$  un camino de  $s$  a  $u$  de la forma

$$p = s, \dots, x, y, \dots, u$$

tal que  $y$  es el primer nodo gris desde  $s$  en  $p$

- Como  $y$  es gris, está en la cola  $Q$  y aún no ha sido extraído
- Como  $x$  es negro, ya fue extraído de la cola

# Correctitud de Dijkstra

## Demostración

Por **H.I.** y el hecho de que solo se puede alterar el costo de un nodo gris o blanco, sabemos que

$$d[x] = \delta(s, x)$$

Como la arista  $(x, y)$  fue visitada al haber extraído  $x$  y visitado sus vecinos,

$$d[y] = \delta(s, y)$$

Esto es cierto, pues de lo contrario,  $p$  no sería óptimo. Ahora, como  $y$  está antes que  $u$  en  $p$ , y los costos son no negativos

$$d[y] = \delta(s, y) \leq \delta(s, u) \leq d[u]$$

Pero  $u$  fue extraído antes que  $y$  de  $Q$ , por lo que su costo cumple

$$d[u] \leq d[y]$$

De estas dos inecuaciones se deduce que  $d[u] = \delta(s, u)$  (contradicción).  $\square$



# Complejidad de Dijkstra

En el peor caso, el algoritmo realiza

- $\mathcal{O}(V)$  operaciones **Extract**
- $\mathcal{O}(|E|)$  operaciones  $d[v] \leftarrow d[u] + \text{cost}(u, v)$  (que actualizan la cola)

Si la cola se implementa como heap binario,

- La operación **Extract** es  $\mathcal{O}(\log(V))$
- La actualización de costos (prioridad) en el heap es  $\mathcal{O}(\log(V))$

El algoritmo de Dijkstra toma tiempo  $\mathcal{O}((V + E) \log(V))$

## ¿Por qué Dijkstra no sirve con costos negativos?

Dijkstra( $s$ ):

**for**  $u \in V - \{s\}$  :

$u.color \leftarrow \text{blanco}$ ;  $d[u] \leftarrow \infty$ ;  $\pi[u] \leftarrow \emptyset$

$s.color \leftarrow \text{gris}$ ;  $d[s] \leftarrow 0$ ;  $\pi[s] \leftarrow \emptyset$

$Q \leftarrow$  cola de **prioridades** vacía (Min Heap)

Insert( $Q, s$ )

**while**  $Q$  no está vacía :

$u \leftarrow \text{Extract}(Q)$

**for**  $v \in \alpha[u]$  :

**if**  $v.color = \text{blanco} \vee v.color = \text{gris}$  :

**if**  $d[v] > d[u] + \text{cost}(u, v)$  :

$d[v] \leftarrow d[u] + \text{cost}(u, v)$ ;  $\pi[v] \leftarrow u$

DecreaseKey( $Q, v, d[v]$ )

**if**  $v.color = \text{blanco}$  :

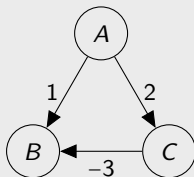
$v.color \leftarrow \text{gris}$ ; Insert( $Q, v$ )

$u.color \leftarrow \text{negro}$

# Dijkstra con costos negativos RIP

## Ejemplo

Tomando el siguiente grafo dirigido con costos negativos, compruebe que Dijkstra no entrega la ruta con el menor costo desde  $A$  hasta  $B$ . Proponga una explicación a este problema.



# Dijkstra con costos negativos RIP

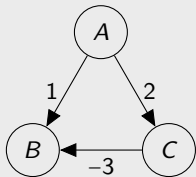
## Ejemplo

Llevaremos en una tabla el estado de  $d[X]$ ,  $\pi[X]$  y  $Q$  al **inicio** de la iteración  $t$  del loop. Recordemos que

$d[X] :=$  menor costo acumulado hasta  $X$

$\pi[X] :=$  ancestro de  $X$  en ese camino

Al inicio de la primera iteración, tenemos los valores iniciales y el nodo de partida como **más prioritario** en  $Q$  (además, es gris)



$t$	$(d[X], \pi[X])$			$Q$	
	$A$	$B$	$C$		
1	$(0, \emptyset)$	$(\infty, \emptyset)$	$(\infty, \emptyset)$	<div><div>A</div><div>0</div></div>	<div><div></div><div>1</div></div>

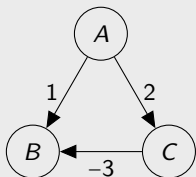
# Dijkstra con costos negativos RIP

## Ejemplo

Al revisar los vecinos de  $A$ , actualizamos sus costos y ancestro, pues tenemos un camino **de largo 1** que es mejor que el costo  $\infty$

Además, al descubrir  $B$  y  $C$ , se pintan grises y se agregan a la cola **respetando prioridad** dada por  $d[X]$

Como revisamos todas las conexiones desde  $A$ , lo terminamos

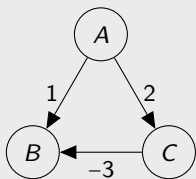


$t$	$(d[X], \pi[X])$			$Q$	
	$A$	$B$	$C$		
1	$(0, \emptyset)$	$(\infty, \emptyset)$	$(\infty, \emptyset)$	<div>A</div> <div>0</div>	<div></div> <div>1</div>
2	$(0, \emptyset)$	$(1, A)$	$(2, A)$	<div>B</div> <div>0</div>	<div>C</div> <div>1</div>

# Dijkstra con costos negativos RIP

## Ejemplo

Por prioridad,  $Q$  entrega a  $B$  como siguiente elemento a revisar. Dado que no tiene vecinos, lo terminamos

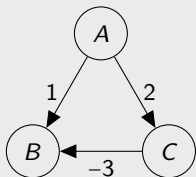


$t$	$(d[X], \pi[X])$			$Q$	
	$A$	$B$	$C$		
2	$(0, \emptyset)$	$(1, A)$	$(2, A)$	<div><math>B</math> 0</div>	<div><math>C</math> 1</div>
3	$(0, \emptyset)$	$(1, A)$	$(2, A)$	<div><math>C</math> 0</div>	<div></div>

# Dijkstra con costos negativos RIP

## Ejemplo

Ahora extraemos de  $Q$  a  $C$ , y como su único vecino ya está terminado, terminamos a  $C$

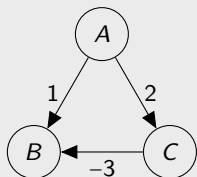


$t$	$(d[X], \pi[X])$			$Q$	
	$A$	$B$	$C$		
3	$(0, \emptyset)$	$(1, A)$	$(2, A)$	<div>C</div> <div>0</div>	<div></div> <div>1</div>
4	$(0, \emptyset)$	$(1, A)$	$(2, A)$	<div></div> <div>0</div>	<div></div> <div>1</div>

# Dijkstra con costos negativos RIP

## Ejemplo

El estado final de los parámetros nos entrega la ruta encontrada por Dijkstra



$t$	$(d[X], \pi[X])$			$Q$	
	$A$	$B$	$C$		
4	$(0, \emptyset)$	$(1, A)$	$(2, A)$	<div style="border: 1px solid black; width: 40px; height: 30px; display: flex; align-items: center; justify-content: center;">0</div>	<div style="border: 1px solid black; width: 40px; height: 30px; display: flex; align-items: center; justify-content: center;">1</div>

La ruta de  $A$  hasta  $B$  entregada cumple

$$\text{cost}((A, B)) = 1 > -1 = \text{cost}((A, C, B))$$

de manera que Dijkstra no entrega la ruta con el costo mínimo en este ejemplo



# El problema de los costos negativos

No olvidemos que Dijkstra se basa en un recorrido BFS del grafo

- Asume que si ya llegamos en  $k$  pasos a un nodo  $u$  con costo  $d[u]$ ...
- ...no podemos llegar con un costo menor en  $k + 1$  pasos
- Esto simplifica el análisis y permite no revisar aristas cuando ya hemos terminado un nodo
- Recordar: si el nodo está terminado/negro, ya no cambia su  $d[u]$

Esto funciona cuando los costos son no negativos, pues una arista  $k + 1$  mantiene o incrementa el costo ya alcanzado en  $k$  pasos

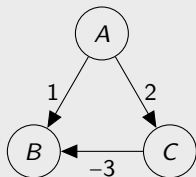
Los costos negativos obligan a revisar caminos de distintos largos antes de dar por terminado un nodo

# Ahora bien, no todo está perdido...

## Ejemplo

Para el grafo dado, si **forzamos** el largo de los caminos, esperamos obtener diferentes ancestros óptimos

Para ser más precisos, nos interesa el costo mínimo al usar **a lo más**  $k$  aristas



$k$	$(d[X], \pi[X])$		
	$A$	$B$	$C$
0	$(0, \emptyset)$	no aplica	no aplica
1	$(0, \emptyset)$	$(1, A)$	$(2, A)$
2	$(0, \emptyset)$	$(-1, C)$	$(2, A)$

Notemos que para  $C$ , la ruta  $(A, C)$  es óptima con a lo más 1 y a lo más 2 aristas

Buscamos un algoritmo que permita resolver esta variante

# Objetivos de la clase

- ☐ Comprender el problema de los costos negativos en Dijkstra
- ☐ Comprender el algoritmo de Bellman-Ford
- ☐ Relacionarlo con los ciclos de costo negativo
- ☐ Comprender la versión del algoritmo para detectar ciclos de costo negativo

# Sumario

Introducción

Algoritmo de Bellman-Ford

Cierre

# Una idea de algoritmo

Para encontrar las rutas **más baratas** desde **una sola fuente** seguiremos la siguiente estrategia

- Buscamos rutas más cortas desde  $s$  con a lo más 1 arista
- Luego, con a lo más 2 aristas
- Luego, con a lo más 3 aristas
- ...

Esta estrategia aprovecha un enfoque de programación dinámica

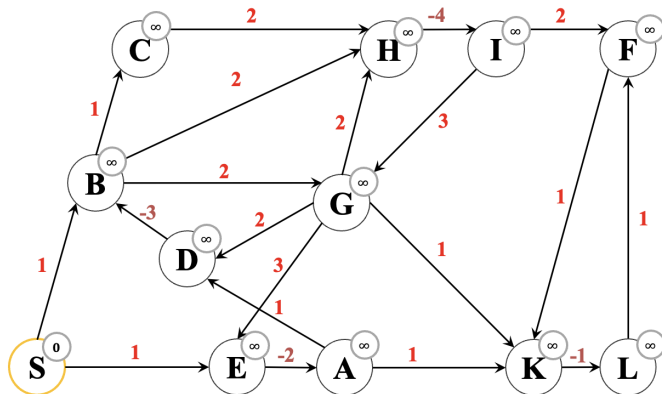
1. Buscamos las rutas más cortas con a lo más  $k$  aristas
2. Podemos utilizar las rutas más cortas con a lo más  $k - 1$  aristas

¿Hasta qué largo de  $k$  es necesario iterar?

# Una idea de algoritmo

Iniciamos los nodos con  $d[u] = \infty$  salvo la fuente  $S$

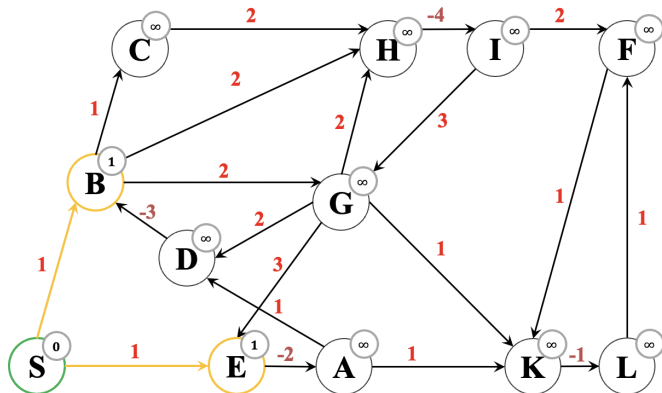
Marcaremos con **verde** los caminos más cortos asegurados **hasta el momento**



# Una idea de algoritmo

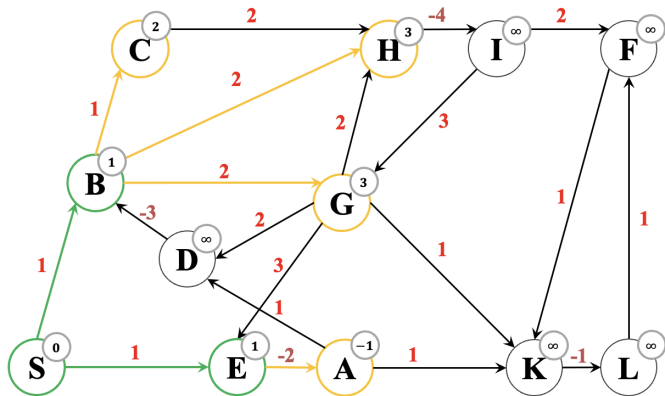
Logramos llegar a *B* y *E* con un costo acumulado menor al previo

Notemos que con caminos de largo  $\leq 1$ , llegamos directo desde *S*



# Una idea de algoritmo

Al extender a caminos de largo  $\leq 2$ , estamos usando las rutas ya calculadas



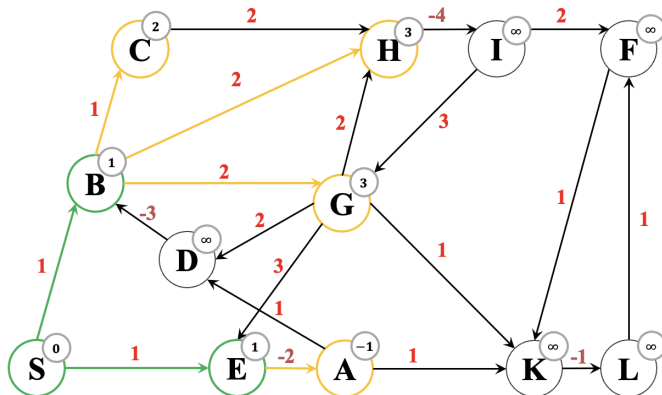
E.g. para  $G$ , como el ancestro óptimo es  $B$ , usamos la ruta más corta de largo  $\leq 1$  con la que se llega a  $B$



# Una idea de algoritmo

Ojo: ahora extenderemos el rango a caminos de largo  $\leq 3$

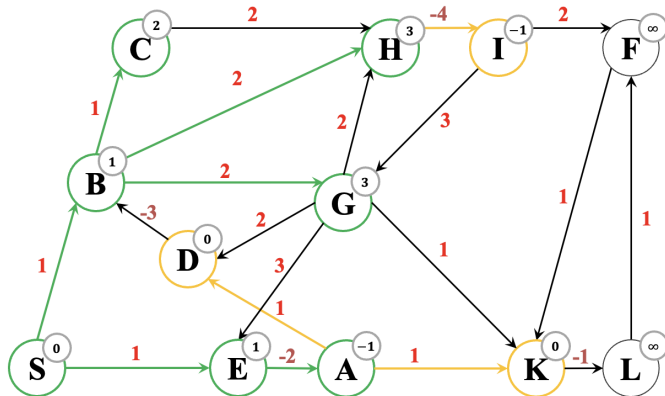
Llegaremos a nodos ya visitados a través de un camino alternativo



¿Qué hacemos en ese caso? ¿Los ignoramos como en Dijkstra?

# Una idea de algoritmo

En este caso, el camino  $(S, B, C, H)$  tiene costo mayor a  $(S, B, H)$ , de manera que no incluimos la arista  $(C, H)$  en dicho camino



# Subestructura óptima en rutas

## Definición

Dado un grafo dirigido  $G$  y nodos  $x, y \in V(G)$ , diremos que  $p : v_0, v_1, \dots, v_n$  es un camino dirigido de  $x$  hasta  $y$  si

- $(v_i, v_{i+1}) \in E(G)$  para cada  $0 \leq i \leq n-1$
- $v_0 = x$  y  $v_n = y$

También denotaremos a  $p$  como  $v_0 \rightarrow v_1 \rightarrow \dots \rightarrow v_n$ . Si existe un camino dirigido  $p$  de  $x$  hasta  $y$  lo denotaremos por  $x \rightsquigarrow_p y$ . Además, el largo del camino dirigido  $p$  es  $|p| = n$

## Proposición

Si el camino dirigido  $u \rightsquigarrow_p x \rightarrow y$  es una ruta más corta de  $u$  a  $y$ , entonces  $p$  es una ruta más corta de  $u$  a  $x$

Un camino óptimo  $|p| \leq k$  no necesariamente contiene uno de largo  $|p'| = k - 1$ . Siempre contiene uno de largo  $|p'| \leq k - 1$

# Subestructura óptima en rutas

En nuestro algoritmo, consideremos la  $k$ -ésima iteración

- Estamos analizando las rutas más cortas con largo  $|p| \leq k$
- Sean  $s \rightsquigarrow u$  y  $s \rightsquigarrow v$  las rutas más cortas con  $|p| \leq k - 1$
- Para cada arista  $(u, v) \in E$  hay que comparar dos rutas
  - $s \rightsquigarrow u \rightarrow v$
  - $s \rightsquigarrow v$  (la misma ruta ya conocida, sin cambios)
- Tal comparación determina si la arista  $(u, v)$  se incluye en la ruta óptima  $s \rightsquigarrow v$  para obtener una de largo  $\leq k$

Notemos que esta comparación es esencialmente la misma que en Dijkstra

¿Cuál es el rango de  $k$  que debemos verificar?

# Largos posibles para rutas óptimas

Nos interesa conocer el largo máximo de una ruta más corta  $p$  desde  $s$  a cualquier nodo de  $G$

Para esto, definimos una herramienta útil

## Definición

Sea  $G$  un grafo no dirigido con costos. Un ciclo  $p : v_0, v_1, \dots, v_n$  de  $G$  es un **ciclo con costo negativo** si

$$\text{cost}(p) = \sum_{i=0}^{n-1} \text{cost}(v_i, v_{i+1}) < 0$$

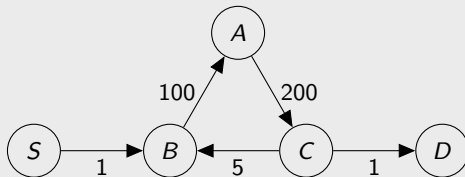
¿Sabemos algo sobre ciclos en rutas más cortas?

En especial, ¿sabemos algo sobre ciclos con costos negativos?

# Largos posibles para rutas óptimas

## Ejemplo

En el siguiente grafo, el ciclo  $B, A, C, B$  es de costo no negativo



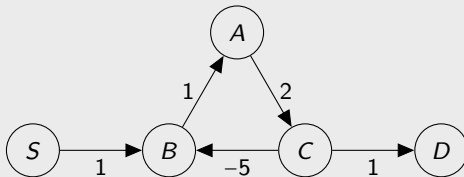
Es claro que este ciclo no hace parte de ningún camino más corto, por muy barata que sea la arista  $(C, B)$

Se puede demostrar que toda ruta más corta  
**no tiene** ciclos de costo no negativo

# Largos posibles para rutas óptimas

## Ejemplo

En el siguiente grafo, el ciclo  $B, A, C, B$  es de costo negativo



¿Cuál es el camino dirigido más barato de  $S$  hasta  $D$ ?

- $cost(S, B, A, C, D) = 5$
- $cost(S, B, A, C, B, A, C, D) = 3$
- $cost(S, B, A, C, B, A, C, B, A, C, D) = 1$

Si existe un ciclo de costo negativo **alcanzable desde  $S$** ,  
no hay ruta más corta desde  $S$  hasta todo nodo

# Largos posibles para rutas óptimas

Dado un camino más corto  $p$  de  $s$  a  $u$

- Sabemos que  $G$  no tiene ciclos con costo negativo alcanzables desde  $s$  (impiden la existencia de rutas más cortas)
- $p$  tampoco tiene ciclos de costo no negativo

Luego, la ruta más corta más larga  $p$  no tiene ciclos

- Sabemos que no repite vértices ( $p$  es simple)
- La ruta más larga posible tendría  $|p| = |V|$

Los posibles valores de  $k$  están en el rango  $1 \dots |V| - 1$



# Algoritmo de Bellman-Ford

BellmanFord( $s$ ):

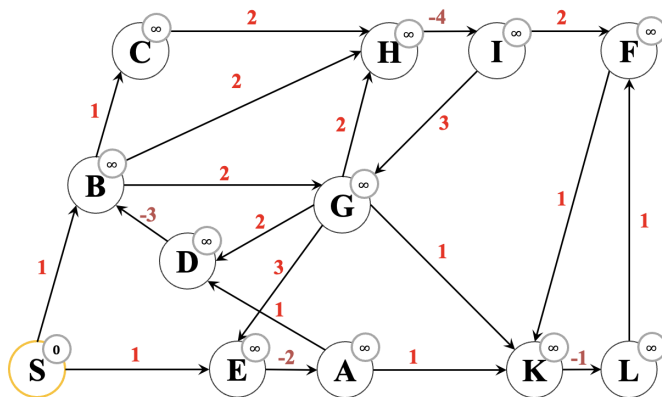
```
1  for  $u \in V$  :  
2       $d[u] \leftarrow \infty$ ;  $\pi[u] \leftarrow \emptyset$   
3   $d[s] \leftarrow 0$   
4  for  $k = 1 \dots |V| - 1$  :  
5      for  $(u, v) \in E$  :  
6          if  $d[v] > d[u] + \text{cost}(u, v)$  :  
7               $d[v] \leftarrow d[u] + \text{cost}(u, v)$   
8               $\pi[v] \leftarrow u$ 
```

Al revisar  $|V| - 1$  veces **todas** las aristas, nos aseguramos de actualizar rutas en caso de encontrar atajos

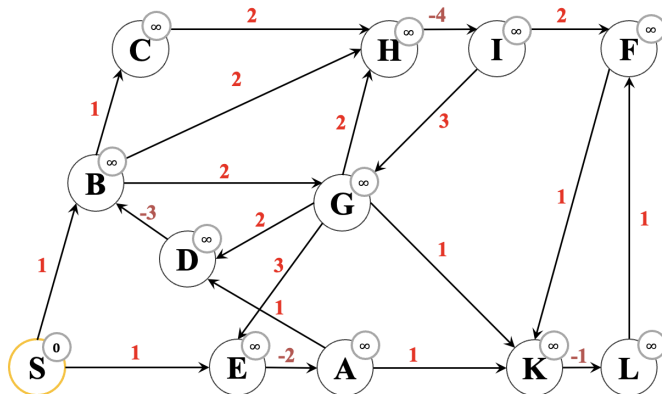
# Algoritmo de Bellman-Ford

## Ejercicio

Determine los costos mínimos de las rutas más baratas para ir de S a los demás nodos del siguiente grafo

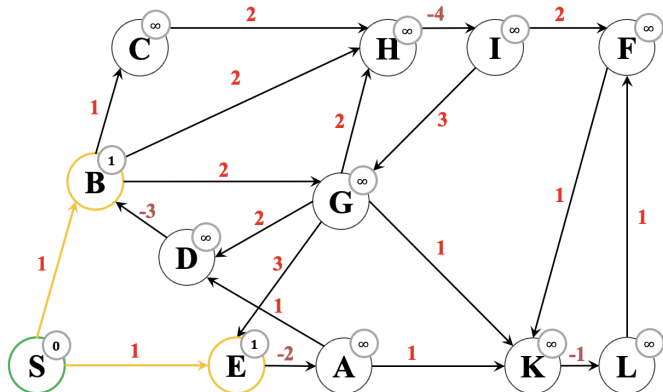


# Algoritmo de Bellman-Ford



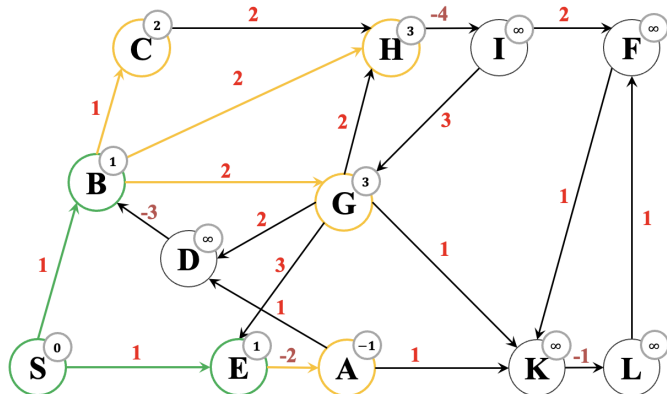
$k = 0$

# Algoritmo de Bellman-Ford



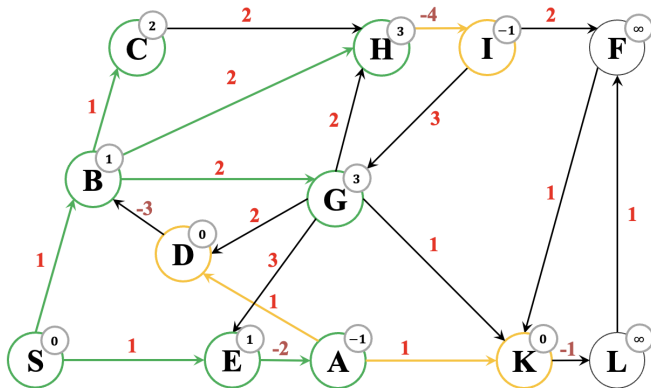
$k = 1$

# Algoritmo de Bellman-Ford



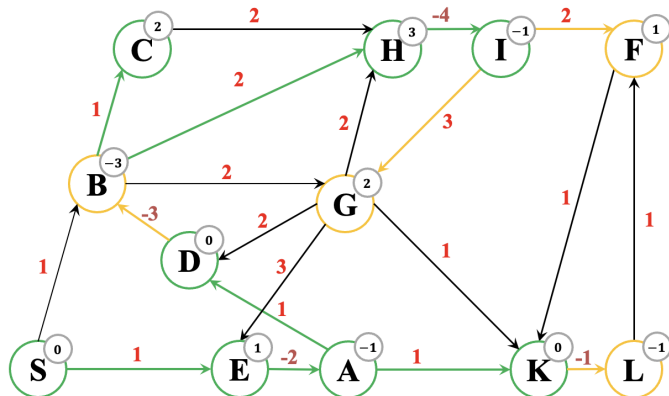
$k = 2$

# Algoritmo de Bellman-Ford



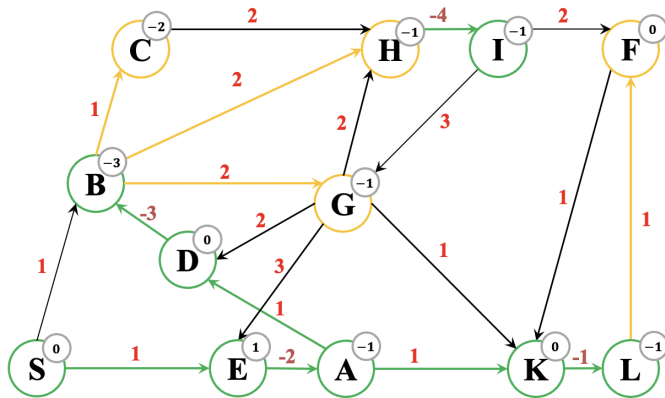
$k = 3$

# Algoritmo de Bellman-Ford



$k = 4$

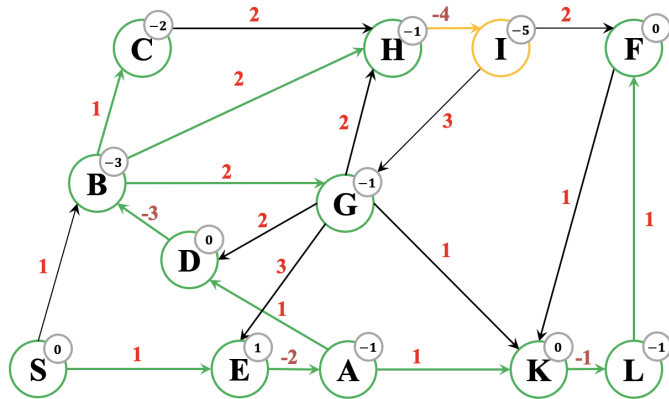
# Algoritmo de Bellman-Ford



$k = 5$

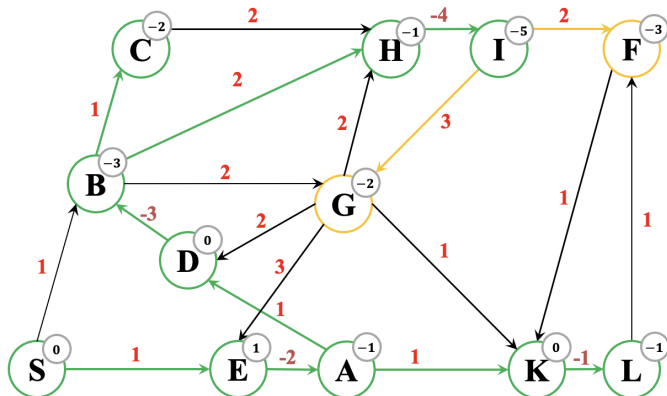


# Algoritmo de Bellman-Ford



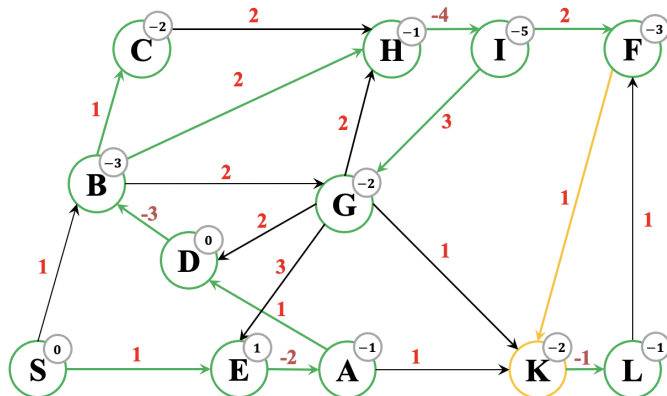
$k = 6$

# Algoritmo de Bellman-Ford



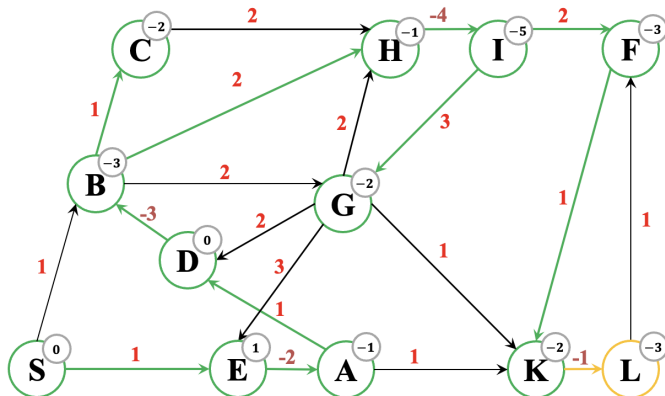
$k = 7$

# Algoritmo de Bellman-Ford



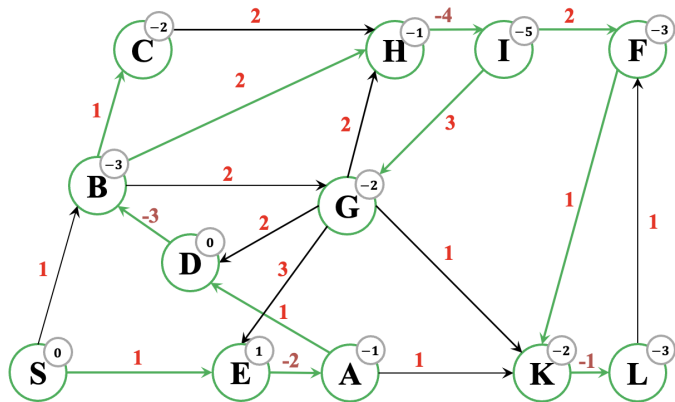
$k = 8$

# Algoritmo de Bellman-Ford



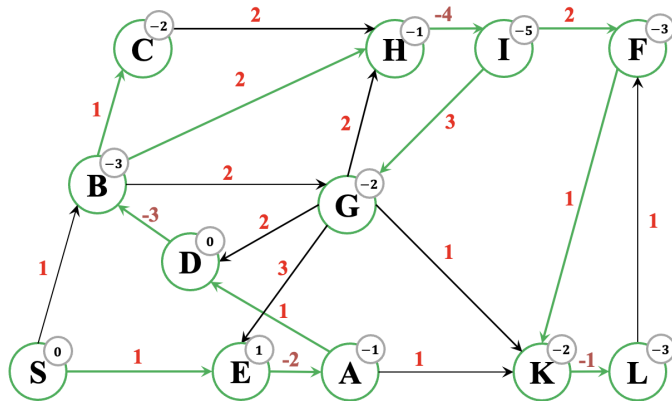
$k = 9$

# Algoritmo de Bellman-Ford



$k = 10$

# Algoritmo de Bellman-Ford



$k = 11$

# Ciclos con costo negativos

Detalle importante!

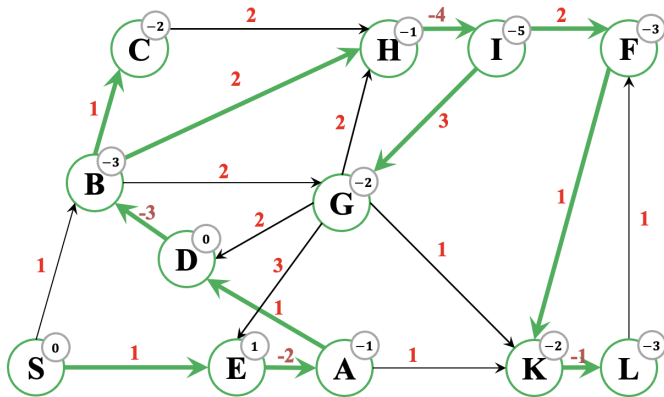
- Dijimos que  $|V| - 1$  iteraciones eran suficientes porque no habían ciclos

Lo que interesa no es que el **grafo sea acíclico**

Importa que **desde la fuente** no se llegue a ningún **ciclo con costo negativo** (pues es el único tipo de ciclo que puede aparecer)

¿Cómo chequeamos que no hayan ciclos de costo negativo?

## Ciclos con costo negativos



¿Qué pasa si al agregar una arista más, logramos mejorar un costo?



# Algoritmo de Bellman-Ford

```
ValidBellmanFord(s):  
1  for  $u \in V$  :  
2       $d[u] \leftarrow \infty$ ;  $\pi[u] \leftarrow \emptyset$   
3   $d[s] \leftarrow 0$   
4  for  $k = 1 \dots |V| - 1$  :  
5      for  $(u, v) \in E$  :  
6          if  $d[v] > d[u] + \text{cost}(u, v)$  :  
7               $d[v] \leftarrow d[u] + \text{cost}(u, v)$   
8               $\pi[v] \leftarrow u$   
9  for  $(u, v) \in E$  :  
10     if  $d[v] > d[u] + \text{cost}(u, v)$  :  
11         return false  
12 return true
```

El resultado de este algoritmo indica si los costos en  $d$  son válidos como costos de rutas más cortas

# Complejidad

ValidBellmanFord( $s$ ):

```
1  for  $u \in V$  :  
2       $d[u] \leftarrow \infty$ ;  $\pi[u] \leftarrow \emptyset$   
3   $d[s] \leftarrow 0$   
4  for  $k = 1 \dots |V| - 1$  :  
5      for  $(u, v) \in E$  :  
6          if  $d[v] > d[u] + c(u, v)$  :  
7               $d[v] \leftarrow d[u] + c(u, v)$   
8               $\pi[v] \leftarrow u$   
9  for  $(u, v) \in E$  :  
10     if  $d[v] > d[u] + c(u, v)$  :  
11         return false  
12  return true
```

Actualización de costos

- $\mathcal{O}(V)$  iteraciones
- Cada una hace  $\mathcal{O}(E)$  comparaciones
- Total:  $\mathcal{O}(VE)$

Chequeo de ciclos

- $\mathcal{O}(E)$  comparaciones

Bellman-Ford es un algoritmo  $\mathcal{O}(VE)$  en el peor caso

# Sumario

Introducción

Algoritmo de Bellman-Ford

Cierre

# Objetivos de la clase

- ☐ Comprender el problema de los costos negativos en Dijkstra
- ☐ Comprender el algoritmo de Bellman-Ford
- ☐ Relacionarlo con los ciclos de costo negativo
- ☐ Comprender la versión del algoritmo para detectar ciclos de costo negativo