Algoritmo de Bellman-Ford

Clase 26

IIC 2133 - Sección 2

Prof. Mario Droguett

Sumario

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 ∈ V − {s} :
        u.color ← blanco;  d[u] ← ∞;  π[u] ← ∅
    s.color ← gris;  d[s] ← 0;  π[s] ← ∅
    Q ← cola de prioridades vacía (Min Heap)
    Insert(Q, s)
    while Q no está vacía :
        u ← Extract(Q)
        for v ∈ α[u] :
            if v.color = blanco ∨ v.color =gris :
                if d[v] > d[u] + cost(u, v) :
                    d[v] ← d[u] + cost(u, v);  π[v] ← u
                    DecreaseKey(Q, v, d[v])
                if v.color =blanco :
                    v.color ← gris;  Insert(Q, v)
        u.color ← negro
```

Correctitud de Dijkstra

Demostración

Finitud

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 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

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.

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

$$P(n) := \begin{array}{l} \text{al inicio de la } n\text{-ésima iteración del } \textbf{while} \\ \text{el nodo } u \text{ extraído de } Q \text{ cumple } d[u] = \delta(s, u) \end{array}$$

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 $d[s] = 0$.

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 $k < n$.

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

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

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,

Correctitud de Dijkstra

Demostración

Para argumentar que existe un camino de s hasta u,

- Si no existe tal camino, el costo ideal es $\delta(s, u) = \infty$

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 $d[u] = \infty$

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

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

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

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

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

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

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

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

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

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

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

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,

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

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.

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

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

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

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

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

Complejidad de Dijkstra

En el peor caso, el algoritmo realiza

- $\mathcal{O}(V)$ operaciones `Extract`

Complejidad de Dijkstra

En el peor caso, el algoritmo realiza

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

Complejidad de Dijkstra

En el peor caso, el algoritmo realiza

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

Si la cola se implementa como heap binario,

Complejidad de Dijkstra

En el peor caso, el algoritmo realiza

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

Si la cola se implementa como heap binario,

- La operación `Extract` es $\mathcal{O}(\log(V))$

Complejidad de Dijkstra

En el peor caso, el algoritmo realiza

- $\mathcal{O}(V)$ operaciones `Extract`
- $\mathcal{O}(|E|)$ operaciones $d[v] \leftarrow d[u] + cost(u, v)$ (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 $\mathcal{O}(\log(V))$

Complejidad de Dijkstra

En el peor caso, el algoritmo realiza

- $\mathcal{O}(V)$ operaciones `Extract`
- $\mathcal{O}(|E|)$ operaciones $d[v] \leftarrow d[u] + cost(u, v)$ (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 $\mathcal{O}((V + E)\log(V))$

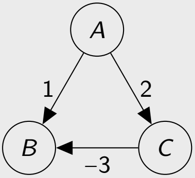
¿Por qué Dijkstra no sirve con costos negativos?

```
Dijkstra(s):
    for u ∈ V − {s} :
        u.color ← blanco;  d[u] ← ∞;  π[u] ← ∅
    s.color ← gris;  d[s] ← 0;  π[s] ← ∅
    Q ← cola de prioridades vacía (Min Heap)
    Insert(Q, s)
    while Q no está vacía :
        u ← Extract(Q)
        for v ∈ α[u] :
            if v.color = blanco ∨ v.color =gris :
                if d[v] > d[u] + cost(u, v) :
                    d[v] ← d[u] + cost(u, v);  π[v] ← u
                    DecreaseKey(Q, v, d[v])
            if v.color =blanco :
                v.color ← gris;  Insert(Q, v)
        u.color ← 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

$$\begin{aligned} d[X] &:= \text{ menor costo acumulado hasta } X \\ \pi[X] &:= \text{ ancestro de } X \text{ en ese camino} \end{aligned}$$

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

$$\begin{aligned} d[X] &:= \text{ menor costo acumulado hasta } X \\ \pi[X] &:= \text{ ancestro de } X \text{ en ese camino} \end{aligned}$$

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

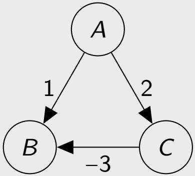
Dijkstra con costos negativos RIP

Ejemplo

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

$$\begin{aligned} d[X] &:= \quad \text{menor costo acumulado hasta } X \\ \pi[X] &:= \quad \text{ancestro de } X \text{ en ese camino} \end{aligned}$$

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



	$(d[X], \pi[X])$			
t	A	B	C	Q
1	$(0, \varnothing)$	$(\infty, \varnothing)$	$(\infty, \varnothing)$	$\boxed{A}_0$ $\boxed{}_1$

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 ∞

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 ∞

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

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 ∞

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

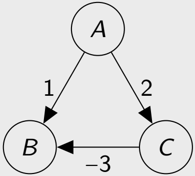
Dijkstra con costos negativos RIP

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 ∞

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

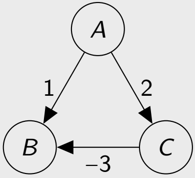


	$(d[X], \pi[X])$			
t	A	B	C	Q
1	$(0, \varnothing)$	$(\infty, \varnothing)$	$(\infty, \varnothing)$	$\boxed{A}_0$ $\boxed{}_1$
2	$(0, \varnothing)$	$(1, A)$	$(2, A)$	$\boxed{B}_0$ $\boxed{C}_1$

Dijkstra con costos negativos RIP

Ejemplo

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

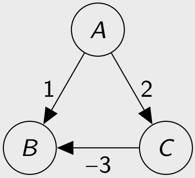


	$(d[X], \pi[X])$			
t	A	B	C	Q
2	$(0, \varnothing)$	$(1, A)$	$(2, A)$	$\boxed{B}_0$ $\boxed{C}_1$
3	$(0, \varnothing)$	$(1, A)$	$(2, A)$	$\boxed{C}_0$ $\boxed{}_1$

Dijkstra con costos negativos RIP

Ejemplo

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

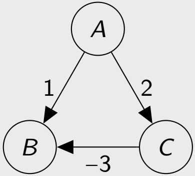


	$(d[X], \pi[X])$			
t	A	B	C	Q
3	$(0, \varnothing)$	$(1, A)$	$(2, A)$	$\boxed{C}_0$ $\boxed{}_1$
4	$(0, \varnothing)$	$(1, A)$	$(2, A)$	$\boxed{}_0$ $\boxed{}_1$

Dijkstra con costos negativos RIP

Ejemplo

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

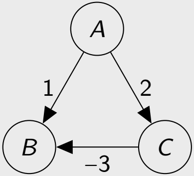


	$(d[X], \pi[X])$			
t	A	B	C	Q
4	$(0, \varnothing)$	$(1, A)$	$(2, A)$	$\boxed{}_0 \ \boxed{}_1$

La ruta de A hasta B entregada cumple

Ejemplo

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



	$(d[X], \pi[X])$			
t	A	B	C	Q
4	$(0, \varnothing)$	$(1, A)$	$(2, A)$	$\underset{0}{\square}\ \underset{1}{\square}$

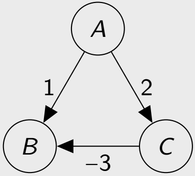
La ruta de A hasta B entregada cumple

$$cost((A, B)) = 1 > -1 = cost((A, C, B))$$

Dijkstra con costos negativos RIP

Ejemplo

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



	$(d[X], \pi[X])$			
t	A	B	C	Q
4	$(0, \varnothing)$	$(1, A)$	$(2, A)$	0 1

La ruta de A hasta B entregada cumple

$$cost((A, B)) = 1 > -1 = cost((A, C, B))$$

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

El problema de los costos negativos

El problema de los costos negativos

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

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 $d[u]$...

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 $d[u]\ldots$
- $\ldots$ no podemos llegar con un costo menor en $k+1$ pasos

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 $d[u]\ldots$
- $\ldots$ 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

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 $d[u]$

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 $d[u]\ldots$
- $\ldots$ 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

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 $d[u]\ldots$
- $\ldots$ 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...

Ahora bien, no todo está perdido...

Ejemplo

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

Ahora bien, no todo está perdido...

Ejemplo

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

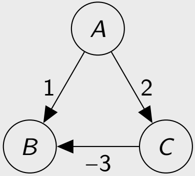
Para ser más precisos, nos interesa el costo mínimo al usar **a lo más** k aristas

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



$(d[X], \pi[X])$

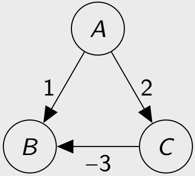
k	A	B	C
0	$(0, \varnothing)$	no aplica	no aplica

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



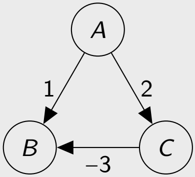
	$(d[X], \pi[X])$		
k	A	B	C
0	$(0, \varnothing)$	no aplica	no aplica
1	$(0, \varnothing)$	$(1, A)$	$(2, A)$

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



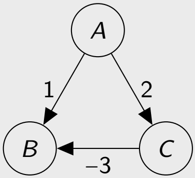
	$(d[X], \pi[X])$		
k	A	B	C
0	$(0, \varnothing)$	no aplica	no aplica
1	$(0, \varnothing)$	$(1, A)$	$(2, A)$
2	$(0, \varnothing)$	$(-1, C)$	$(2, A)$

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



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

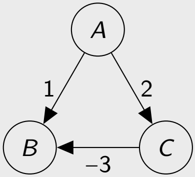
Notemos que para C, la ruta (A, C) es óptima con a lo más 1 y a lo más 2 aristas

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	$(0, \varnothing)$	no aplica	no aplica
1	$(0, \varnothing)$	$(1, A)$	$(2, A)$
2	$(0, \varnothing)$	$(-1, C)$	$(2, A)$

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

Objetivos de la clase

- ☐ Comprender el problema de los costos negativos en Dijkstra

Objetivos de la clase

- ☐ Comprender el problema de los costos negativos en Dijkstra
- ☐ Comprender el algoritmo de Bellman-Ford

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

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

Una idea de algoritmo

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

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

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

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

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
- ...

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

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

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

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

Una idea de algoritmo

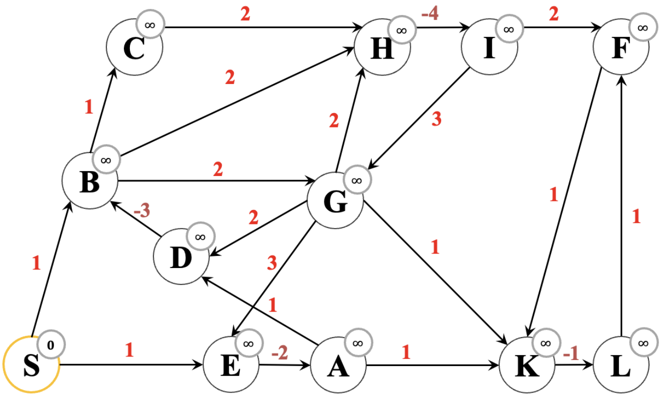
Iniciamos los nodos con $d[u] = \infty$ salvo la fuente S

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

Una idea de algoritmo

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

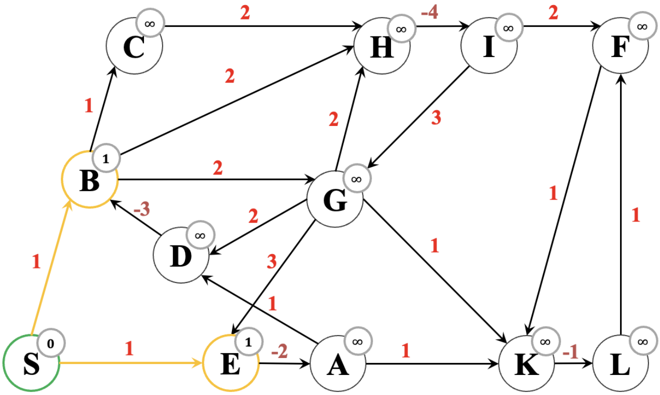
Marcaremos con **verde** los caminos más cortos asegurados **hasta el momento**



Una idea de algoritmo

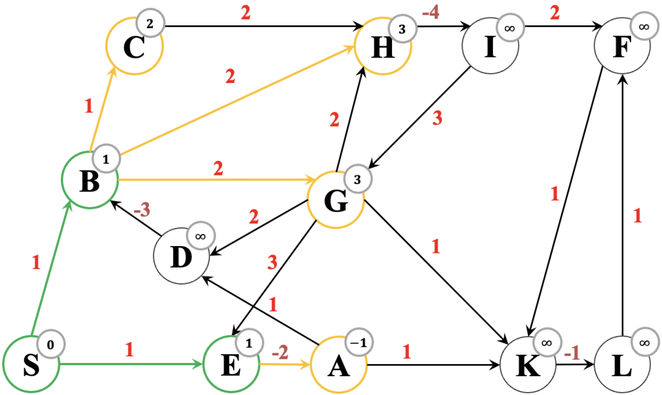
Logramos llegar a B y E con un costo acumulado menor al previo

Notemos que con caminos de largo ≤ 1, llegamos directo desde S



Una idea de algoritmo

Al extender a caminos de largo ≤ 2, estamos usando las rutas ya calculadas

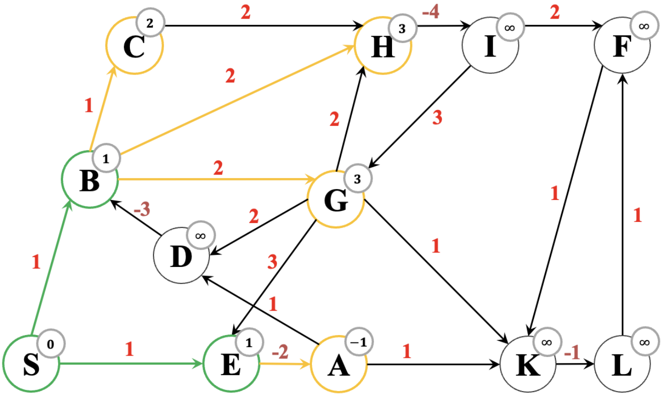


E.g. para G, como el ancestro óptimo es B, usamos la ruta más corta de largo ≤ 1 con la que se llega a B

Una idea de algoritmo

Ojo: ahora extenderemos el rango a caminos de largo ≤ 3

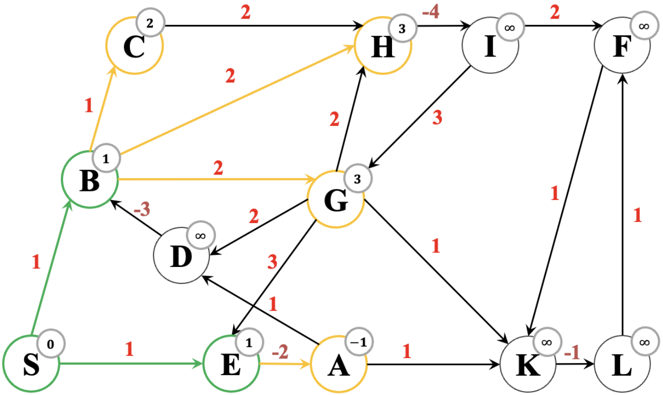
Llegaremos a nodos ya visitados a través de un camino alternativo



Una idea de algoritmo

Ojo: ahora extenderemos el rango a caminos de largo ≤ 3

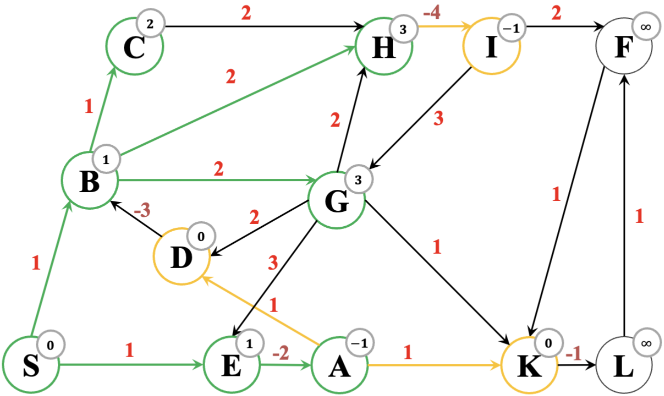
Llegaremos a nodos ya visitados a través de un camino alternativo



¿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 incluímos la arista (C, H) en dicho camino



Subestructura óptima en rutas

Subestructura óptima en rutas

Definición

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

Subestructura óptima en rutas

Definición

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

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

Subestructura óptima en rutas

Definición

Dado un grafo dirigido G y nodos $x, y \in V(G)$, diremos que $p : v_0, v_1, \ldots, v_n$ 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 $v_n = y$

Subestructura óptima en rutas

Definición

Dado un grafo dirigido G y nodos $x, y \in V(G)$, diremos que $p : v_0, v_1, \ldots, v_n$ 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 $v_0 \to v_1 \to \cdots \to v_n$. Si existe un camino dirigido p de x hasta y lo denotaremos por $x \leadsto_p y$. Además, el largo del camino dirigido p es $|p| = n$

Subestructura óptima en rutas

Definición

Dado un grafo dirigido G y nodos $x, y \in V(G)$, diremos que $p : v_0, v_1, \ldots, v_n$ 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 $v_0 \to v_1 \to \cdots \to v_n$. 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 $u \rightsquigarrow_p x \to y$ es una ruta más corta de u a y, entonces p es una ruta más corta de u a x

Subestructura óptima en rutas

Definición

Dado un grafo dirigido G y nodos $x, y \in V(G)$, diremos que $p : v_0, v_1, \ldots, v_n$ 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 $v_0 \to v_1 \to \cdots \to v_n$. 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 $u \rightsquigarrow_p x \to y$ 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

Subestructura óptima en rutas

En nuestro algoritmo, consideremos la k-ésima iteración

- Estamos analizando las rutas más cortas con largo $|p| \leq k$

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 $|p| \leq k - 1$

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

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
  - $s \rightsquigarrow u \rightarrow v$

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)

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 $\leq k$

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

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

Largos posibles para rutas óptimas

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

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

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

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

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 $p : v_0, v_1, \ldots, v_n$ de G es un **ciclo con costo negativo** si

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

¿Sabemos algo sobre ciclos en rutas más cortas?
En especial, ¿sabemos algo sobre ciclos con costos negativos?

Largos posibles para rutas óptimas

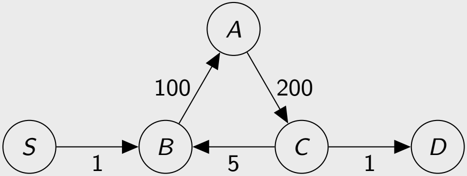
Ejemplo

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

Largos posibles para rutas óptimas

Ejemplo

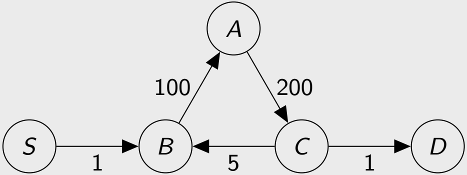
En el siguiente grafo, el ciclo B, A, C, B es de costo no negativo



Largos posibles para rutas óptimas

Ejemplo

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

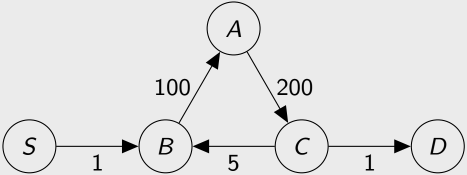


Es claro que este ciclo no hace parte de ningún camino más corto, por muy barata que sea la arista (C, B)

Largos posibles para rutas óptimas

Ejemplo

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



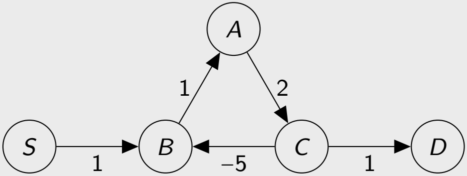
Es claro que este ciclo no hace parte de ningún camino más corto, por muy barata que sea la arista (C, B)

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

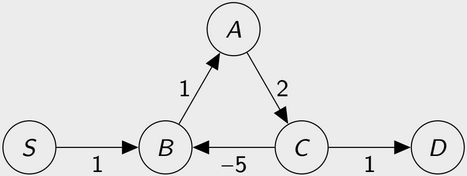


¿Cuál es el camino dirigido más barato de S hasta D?

Largos posibles para rutas óptimas

Ejemplo

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



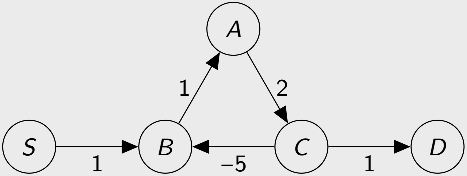
¿Cuál es el camino dirigido más barato de S hasta D?

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

Largos posibles para rutas óptimas

Ejemplo

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



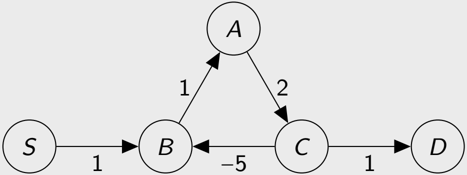
¿Cuál es el camino dirigido más barato de S hasta D?

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

Largos posibles para rutas óptimas

Ejemplo

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



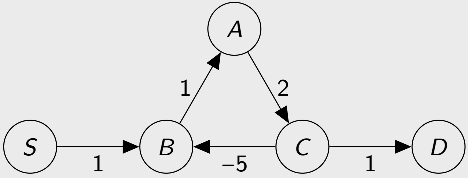
¿Cuál es el camino dirigido más barato de S hasta D?

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

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

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)

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

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

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)

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 $|p| = |V|$

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 $1 \ldots |V| - 1$

Algoritmo de Bellman-Ford

```
BellmanFord(s):
    for u ∈ V :
        d[u] ← ∞;   π[u] ← ∅
    d[s] ← 0
    for k = 1 ... |V| − 1 :
        for (u, v) ∈ E :
            if d[v] > d[u] + cost(u, v) :
                d[v] ← d[u] + cost(u, v)
                π[v] ← u
```

Algoritmo de Bellman-Ford

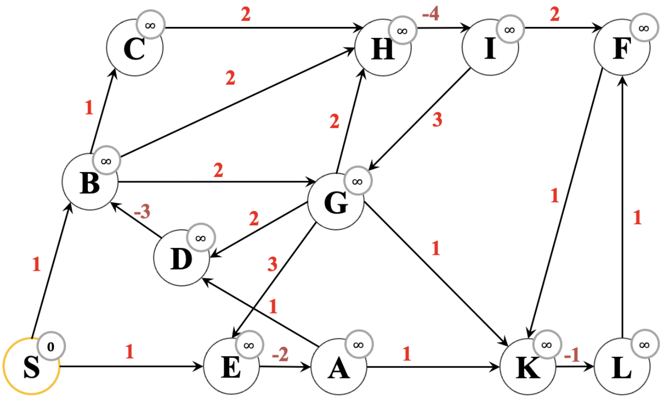
```
BellmanFord(s):
    for u ∈ V :
        d[u] ← ∞;   π[u] ← ∅
    d[s] ← 0
    for k = 1 ... |V| − 1 :
        for (u, v) ∈ E :
            if d[v] > d[u] + cost(u, v) :
                d[v] ← d[u] + cost(u, v)
                π[v] ← u
```

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

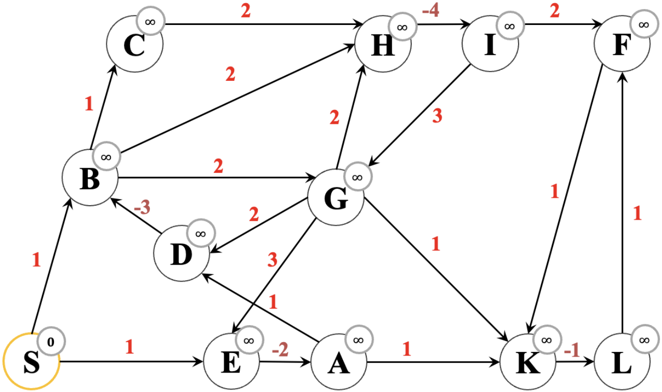
Algoritmo de Bellman-Ford

Ejercicio

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

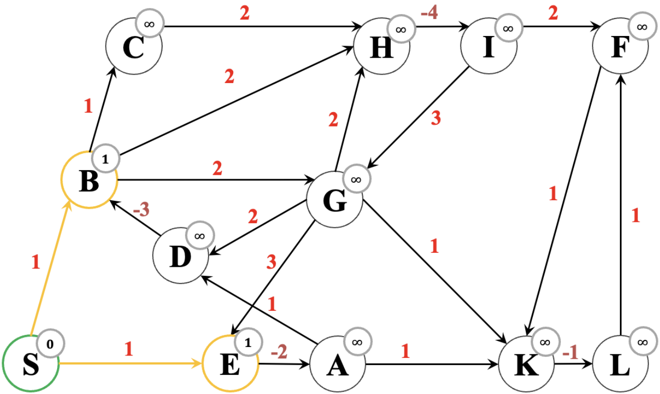


Algoritmo de Bellman-Ford



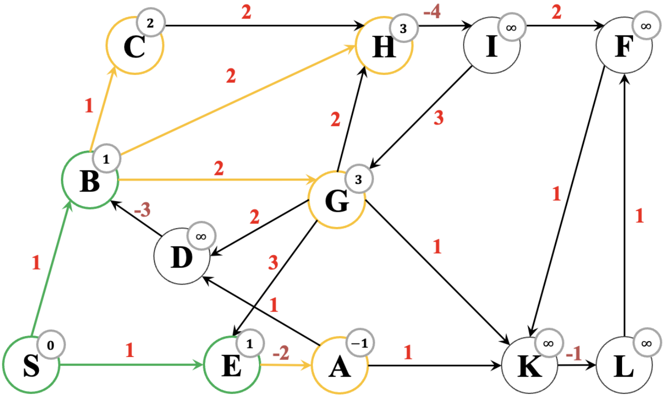
$k = 0$

Algoritmo de Bellman-Ford



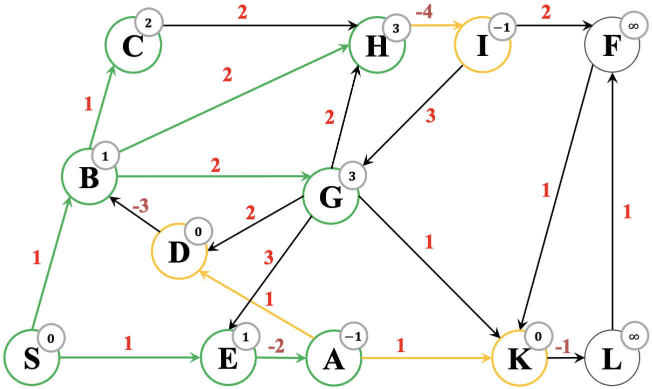
$k = 1$

Algoritmo de Bellman-Ford



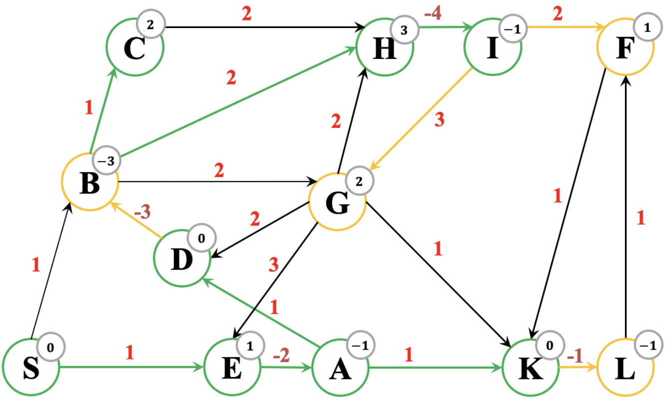
$k = 2$

Algoritmo de Bellman-Ford



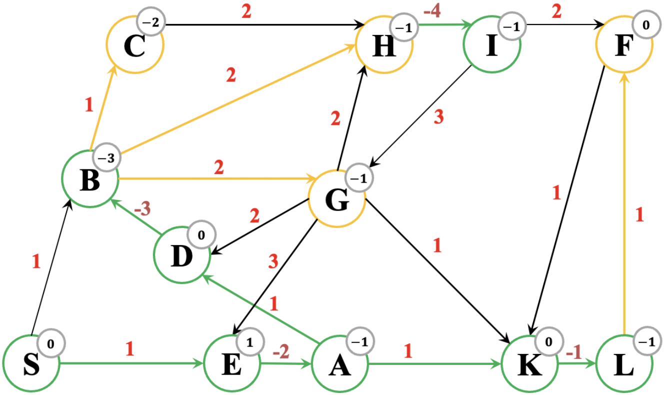
$k = 3$

Algoritmo de Bellman-Ford



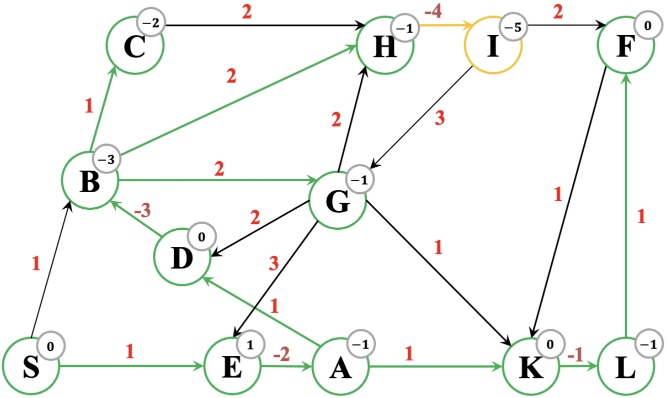
$k = 4$

Algoritmo de Bellman-Ford



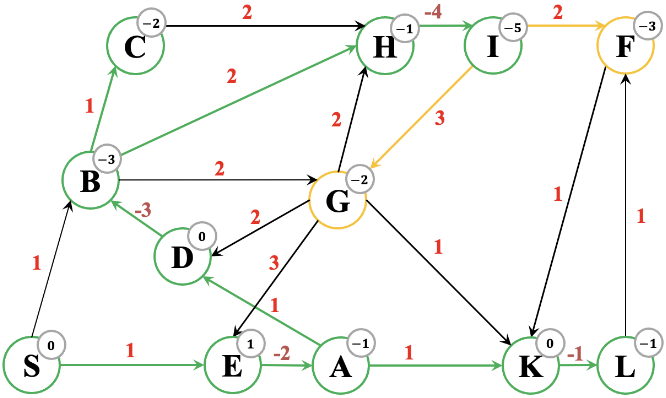
$k = 5$

Algoritmo de Bellman-Ford



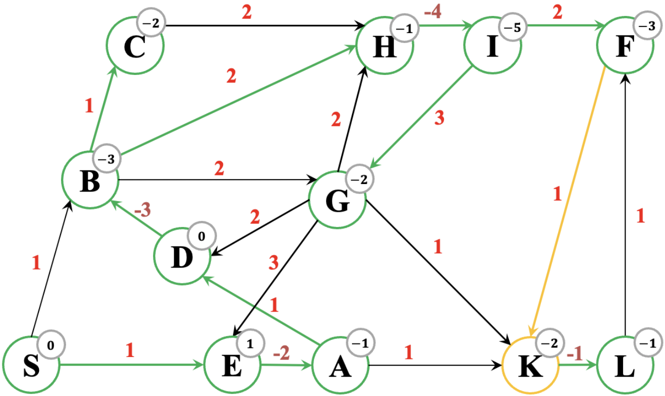
$k = 6$

Algoritmo de Bellman-Ford



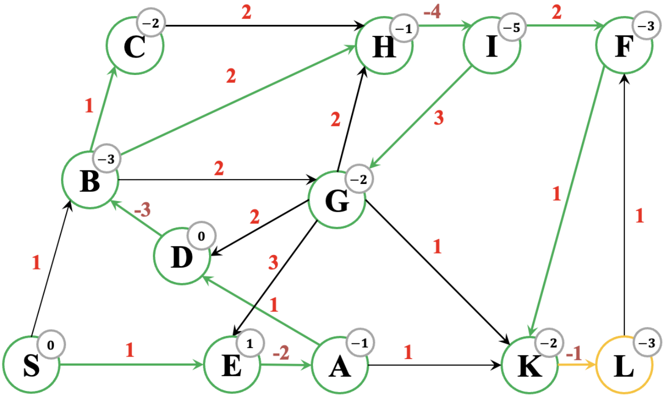
$k = 7$

Algoritmo de Bellman-Ford



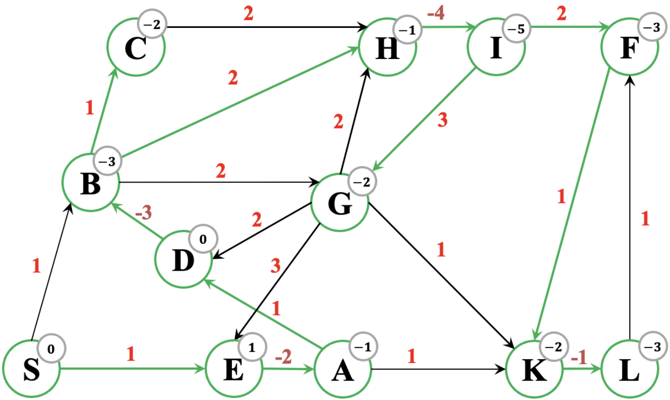
$k = 8$

Algoritmo de Bellman-Ford



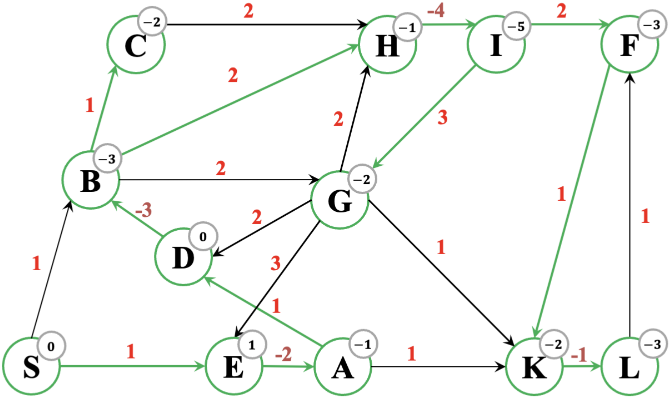
$k = 9$

Algoritmo de Bellman-Ford



$k = 10$

Algoritmo de Bellman-Ford



$k = 11$

Ciclos con costo negativos

Ciclos con costo negativos

Detalle importante!

Ciclos con costo negativos

Detalle importante!

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

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**

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)

Ciclos con costo negativos

Detalle importante!

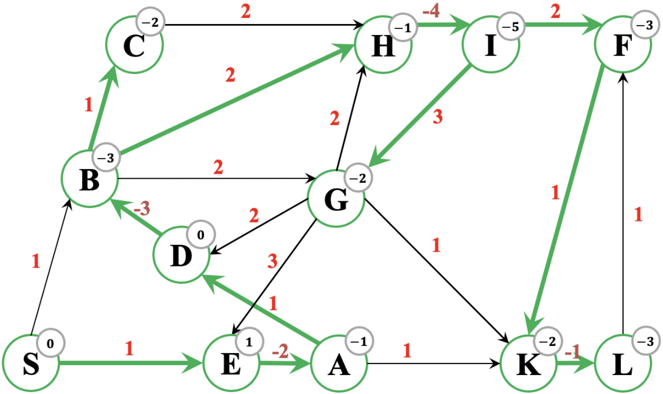
- Dijimos que $|V| - 1$ iteraciones eran suficientes porque no habían ciclos

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):
    for u ∈ V :
        d[u] ← ∞;  π[u] ← ∅
    d[s] ← 0
    for k = 1 ... |V| − 1 :
        for (u, v) ∈ E :
            if d[v] > d[u] + cost(u, v) :
                d[v] ← d[u] + cost(u, v)
                π[v] ← u
    for (u, v) ∈ E :
        if d[v] > d[u] + cost(u, v) :
            return false
    return true
```

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

Complejidad

```
ValidBellmanFord(s):
    for u ∈ V :
        d[u] ← ∞;   π[u] ← ∅
    d[s] ← 0
    for k = 1 ... |V| − 1 :
        for (u, v) ∈ E :
            if d[v] > d[u] + c(u, v) :
                d[v] ← d[u] + c(u, v)
                π[v] ← u
    for (u, v) ∈ E :
        if d[v] > d[u] + c(u, v) :
            return false
    return true
```

Actualización de costos

Complejidad

```
ValidBellmanFord(s):
    for u ∈ V :
        d[u] ← ∞;  π[u] ← ∅
    d[s] ← 0
    for k = 1 ... |V| − 1 :
        for (u, v) ∈ E :
            if d[v] > d[u] + c(u, v) :
                d[v] ← d[u] + c(u, v)
                π[v] ← u
    for (u, v) ∈ E :
        if d[v] > d[u] + c(u, v) :
            return false
    return true
```

Actualización de costos

- $\mathcal{O}(V)$ iteraciones

Complejidad

```
ValidBellmanFord(s):
    for u ∈ V :
        d[u] ← ∞;  π[u] ← ∅
    d[s] ← 0
    for k = 1 ... |V| − 1 :
        for (u, v) ∈ E :
            if d[v] > d[u] + c(u, v) :
                d[v] ← d[u] + c(u, v)
                π[v] ← u
    for (u, v) ∈ E :
        if d[v] > d[u] + c(u, v) :
            return false
    return true
```

Actualización de costos

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

Complejidad

```
ValidBellmanFord(s):
    for u ∈ V :
        d[u] ← ∞;  π[u] ← ∅
    d[s] ← 0
    for k = 1 ... |V| − 1 :
        for (u, v) ∈ E :
            if d[v] > d[u] + c(u, v) :
                d[v] ← d[u] + c(u, v)
                π[v] ← u
    for (u, v) ∈ E :
        if d[v] > d[u] + c(u, v) :
            return false
    return true
```

Actualización de costos

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

Complejidad

```
ValidBellmanFord(s):
    for u ∈ V :
        d[u] ← ∞;  π[u] ← ∅
    d[s] ← 0
    for k = 1 ... |V| − 1 :
        for (u, v) ∈ E :
            if d[v] > d[u] + c(u, v) :
                d[v] ← d[u] + c(u, v)
                π[v] ← u
    for (u, v) ∈ E :
        if d[v] > d[u] + c(u, v) :
            return false
    return true
```

Actualización de costos

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

Chequeo de ciclos

Complejidad

```
ValidBellmanFord(s):
    for u ∈ V :
        d[u] ← ∞;  π[u] ← ∅
    d[s] ← 0
    for k = 1 ... |V| − 1 :
        for (u, v) ∈ E :
            if d[v] > d[u] + c(u, v) :
                d[v] ← d[u] + c(u, v)
                π[v] ← u
    for (u, v) ∈ E :
        if d[v] > d[u] + c(u, v) :
            return false
    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

Complejidad

```
ValidBellmanFord(s):
    for u ∈ V :
        d[u] ← ∞;   π[u] ← ∅
    d[s] ← 0
    for k = 1 ... |V| − 1 :
        for (u, v) ∈ E :
            if d[v] > d[u] + c(u, v) :
                d[v] ← d[u] + c(u, v)
                π[v] ← u
    for (u, v) ∈ E :
        if d[v] > d[u] + c(u, v) :
            return false
    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

Objetivos de la clase

Objetivos de la clase

- ☐ Comprender el problema de los costos negativos en Dijkstra

Objetivos de la clase

- ☐ Comprender el problema de los costos negativos en Dijkstra
- ☐ Comprender el algoritmo de Bellman-Ford

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

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