BFS y algoritmo de Dijkstra

Clase 20

IIC 2133 - Sección 2

Prof. Mario Droguett

Sumario

Rutas en viajes

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

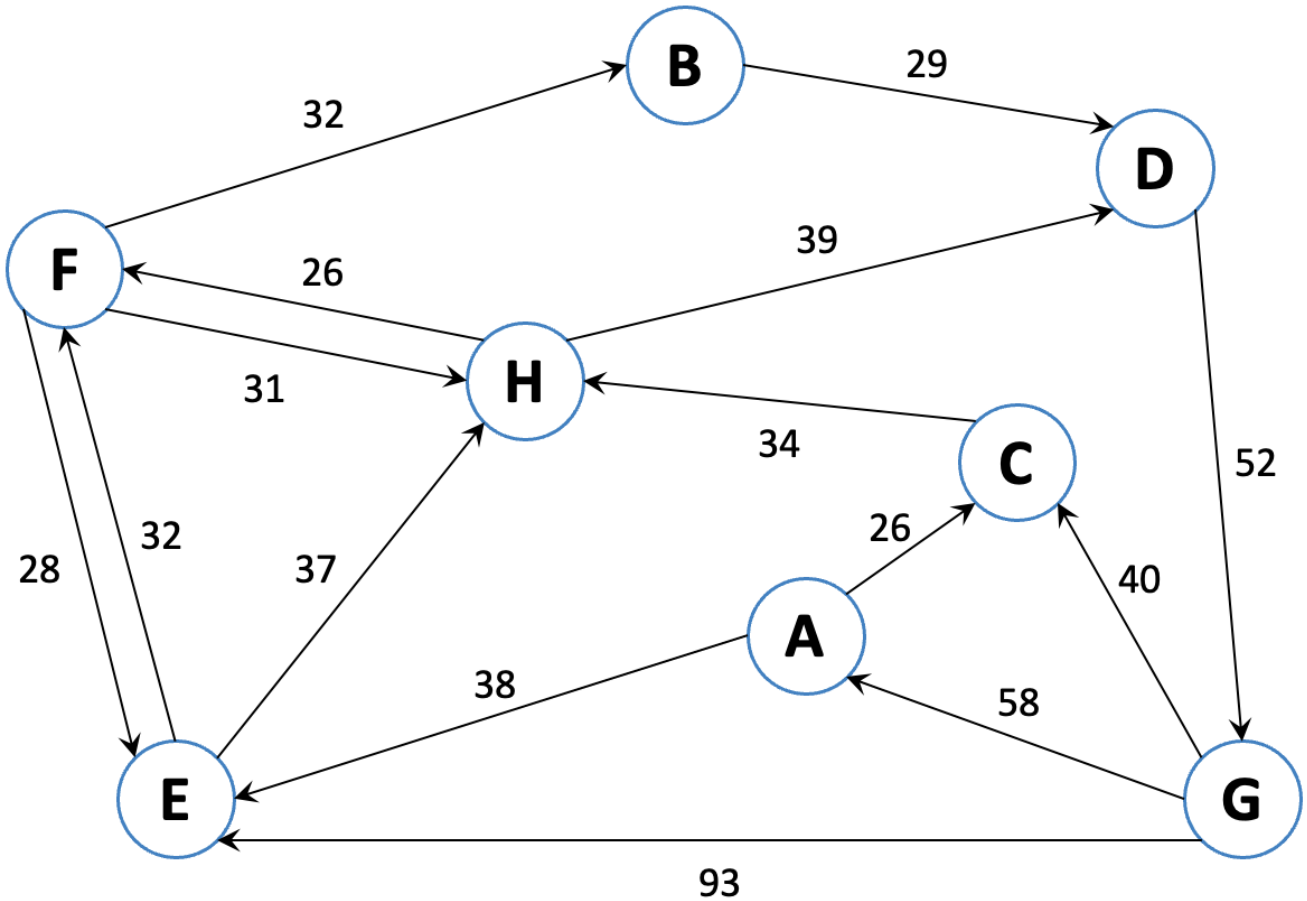
- Podemos pensar en un grafo **dirigido**: caminos y puntos de intersección
- El auto tiene un consumo $R(v)$ cuando la rapidez es v
- El costo del combustible es C
- Cada arista tiene un largo x_i y una velocidad obligatorio v_i
- Además, cada arista tiene un peaje con costo $0 \leq p_i$

El costo de usar el camino i es

$$\sigma_i = x_i \cdot C \cdot R(v_i) + p_i$$

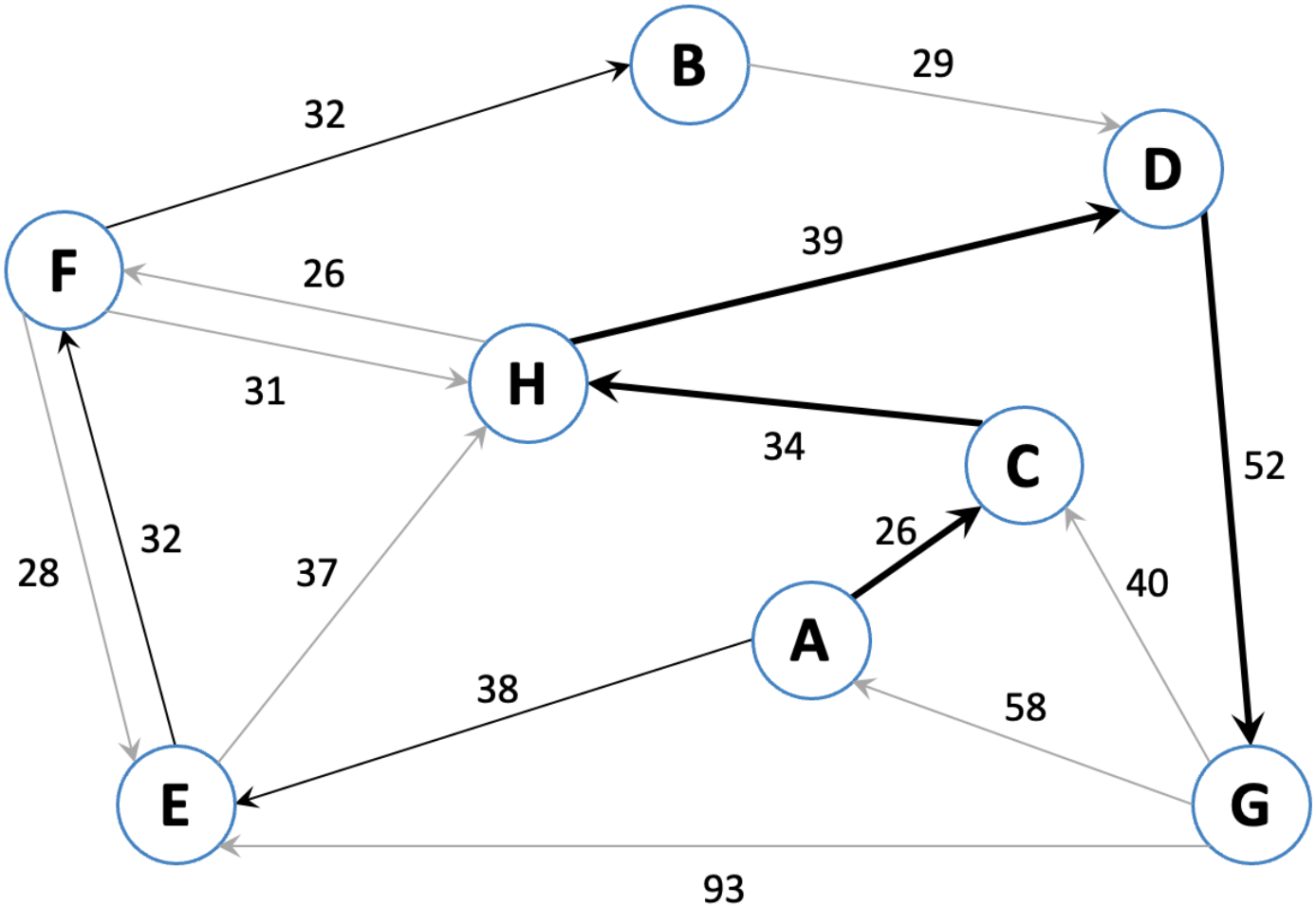
Podemos representar los σ_i en un **grafo dirigido con costos**

Rutas en viajes



Notemos que en este problema, los costos son **no negativos**

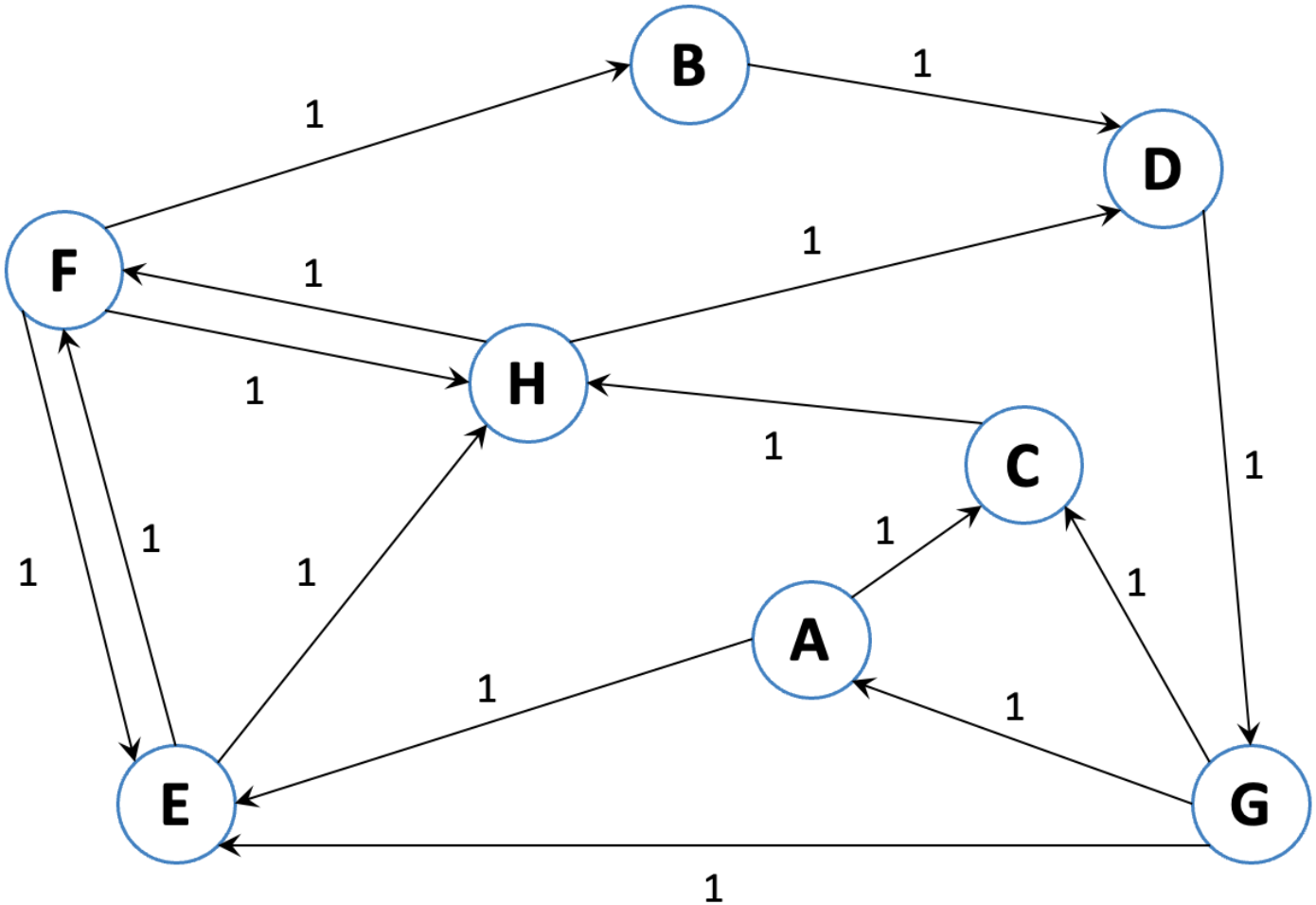
Rutas en viajes



Objetivo: ruta más barata, i.e. suma de los costos debe ser mínima

Rutas en viajes: versión 1.0

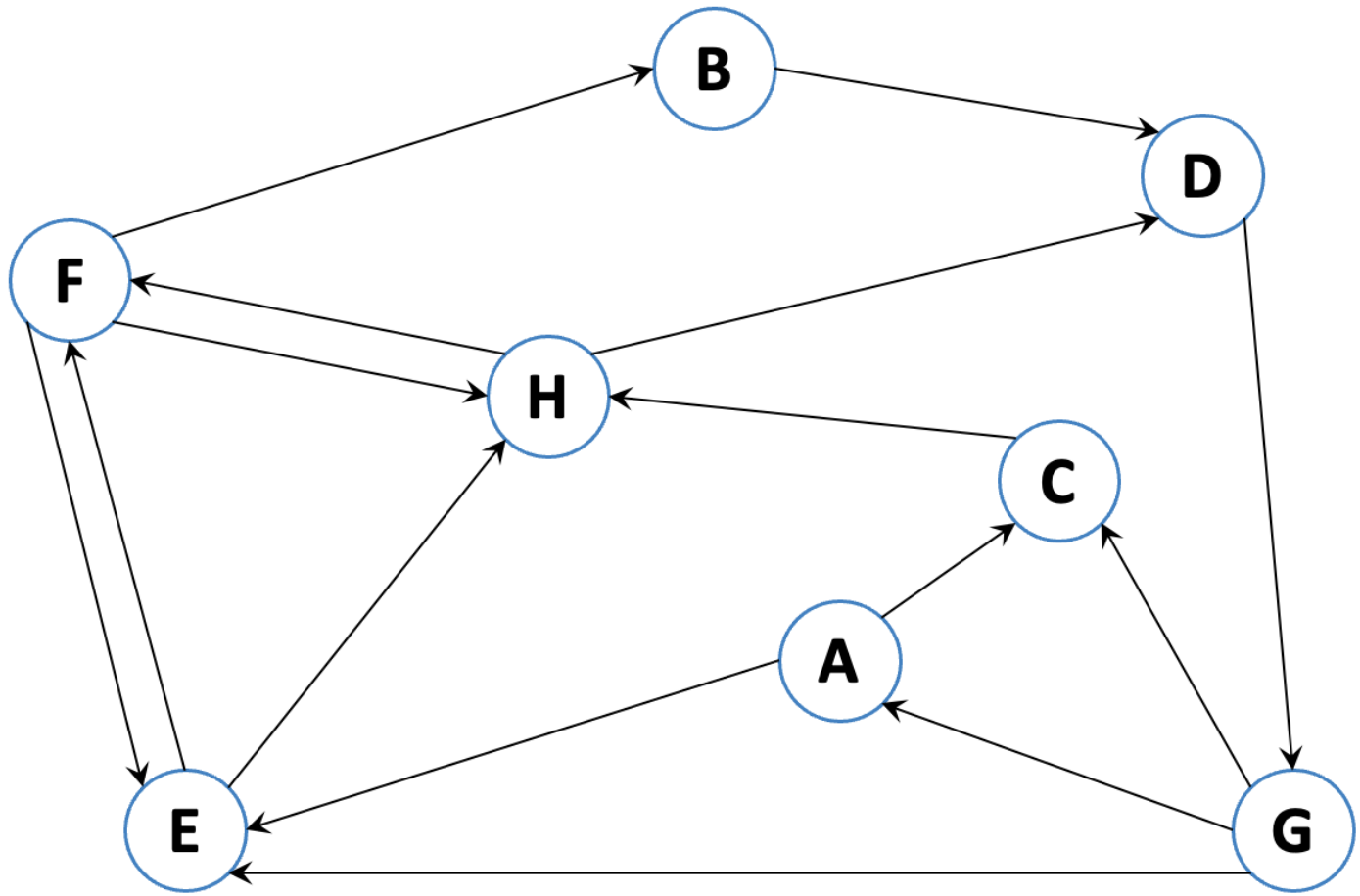
Consideremos el caso en que los costos son iguales: $\sigma_i = K$



¿A qué equivale encontrar la ruta más barata en este caso?

Rutas en viajes: versión 1.0

Consideremos el caso en que los costos son iguales: $\sigma_i = K$



Simplemente buscamos la **ruta más corta** (menos cantidad de aristas)

Sumario

Búsqueda en amplitud

El algoritmo que utilizaremos para resolver este problema simplificado se llama **BFS** o **Búsqueda en amplitud**

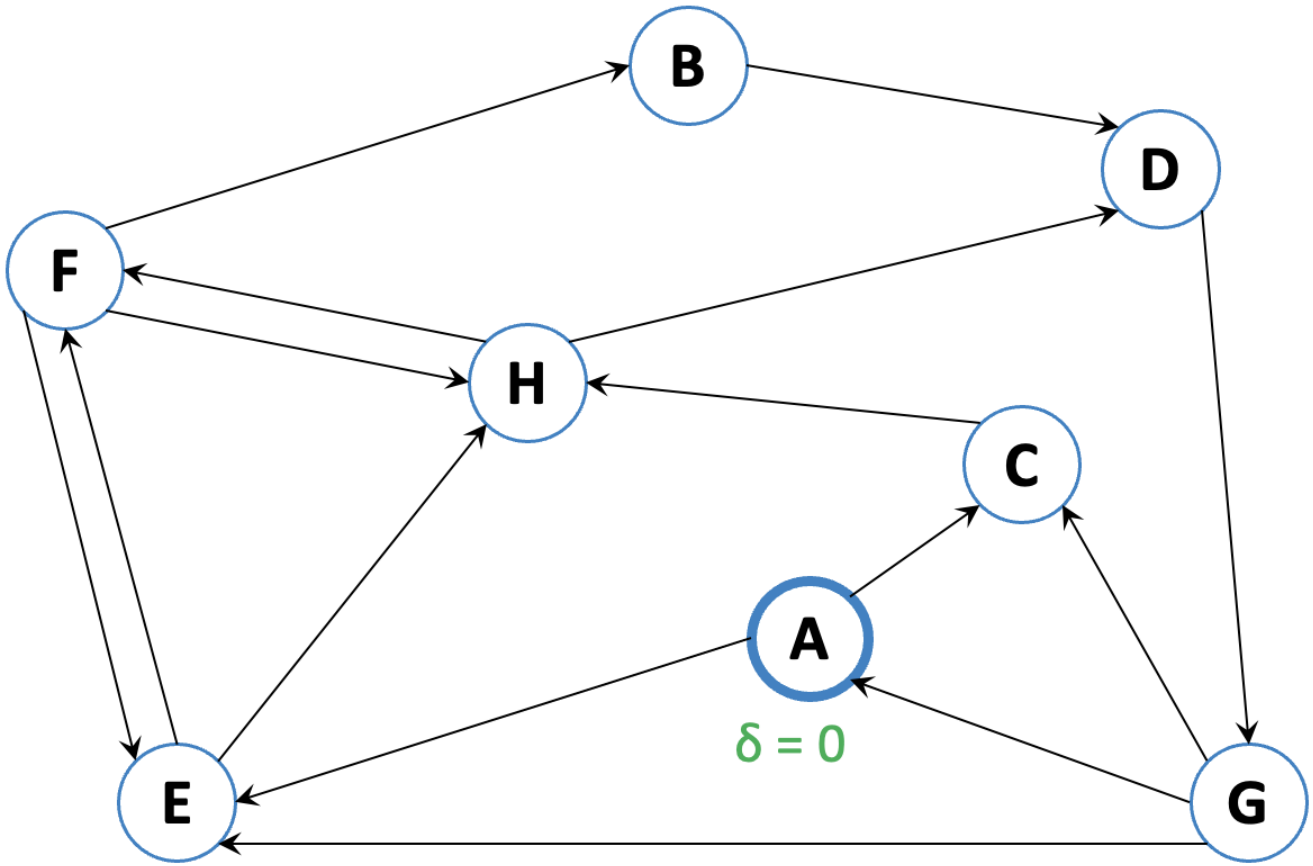
- Tal como DFS es un algoritmo de búsqueda en grafos
- En lugar de agotar rutas antes de visitar vecinos...
- ...BFS revisa los vecinos inmediatos y luego sigue con los vecinos de estos
- Es decir, BFS recorre en base a **distancia desde un nodo inicial**

Para nuestro problema, llevaremos la cuenta de la **distancia** δ desde el punto de inicio hasta cada nodo

Por definición δ es la **menor distancia** de A a cualquier nodo

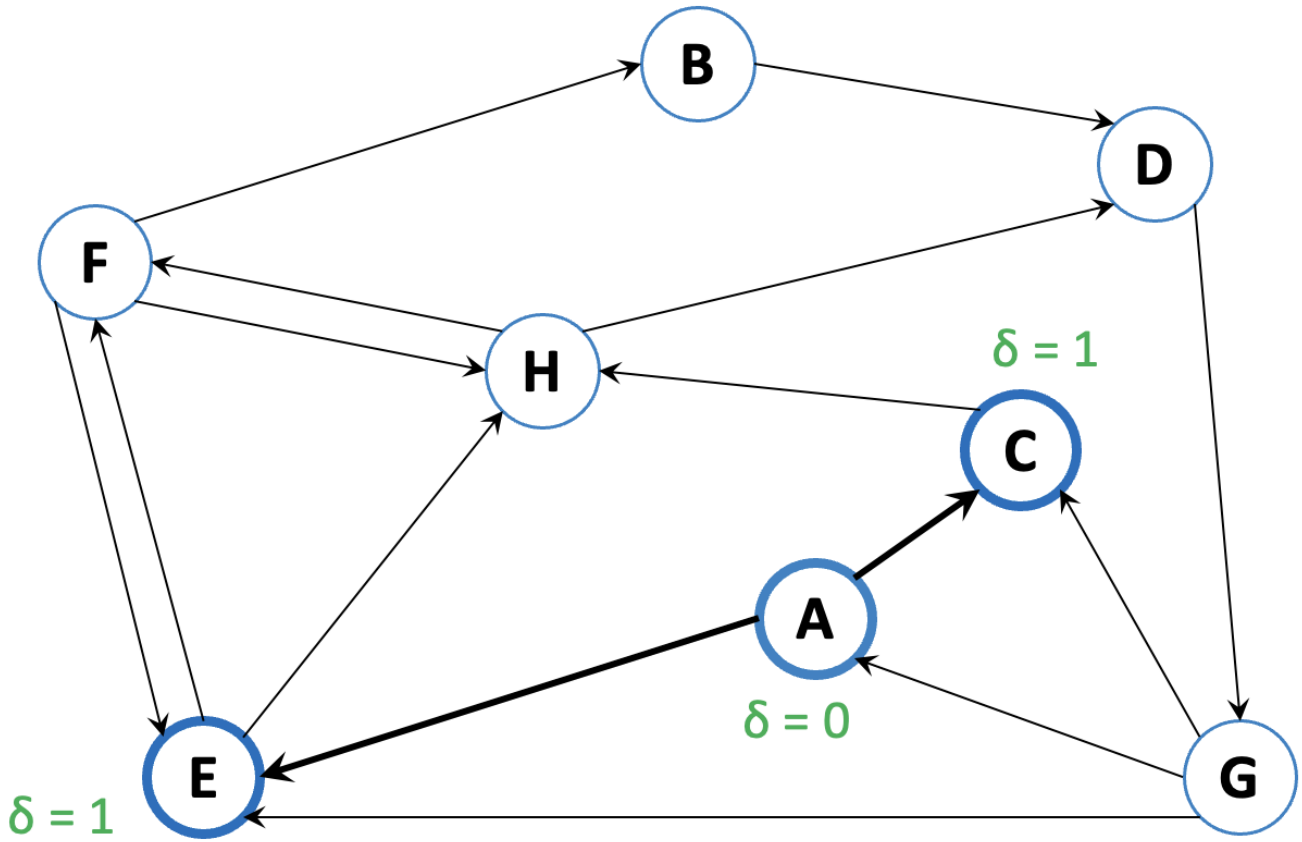
BFS y rutas más cortas

Iniciamos visitando los nodos a distancia $\delta = 0$ desde A



BFS y rutas más cortas

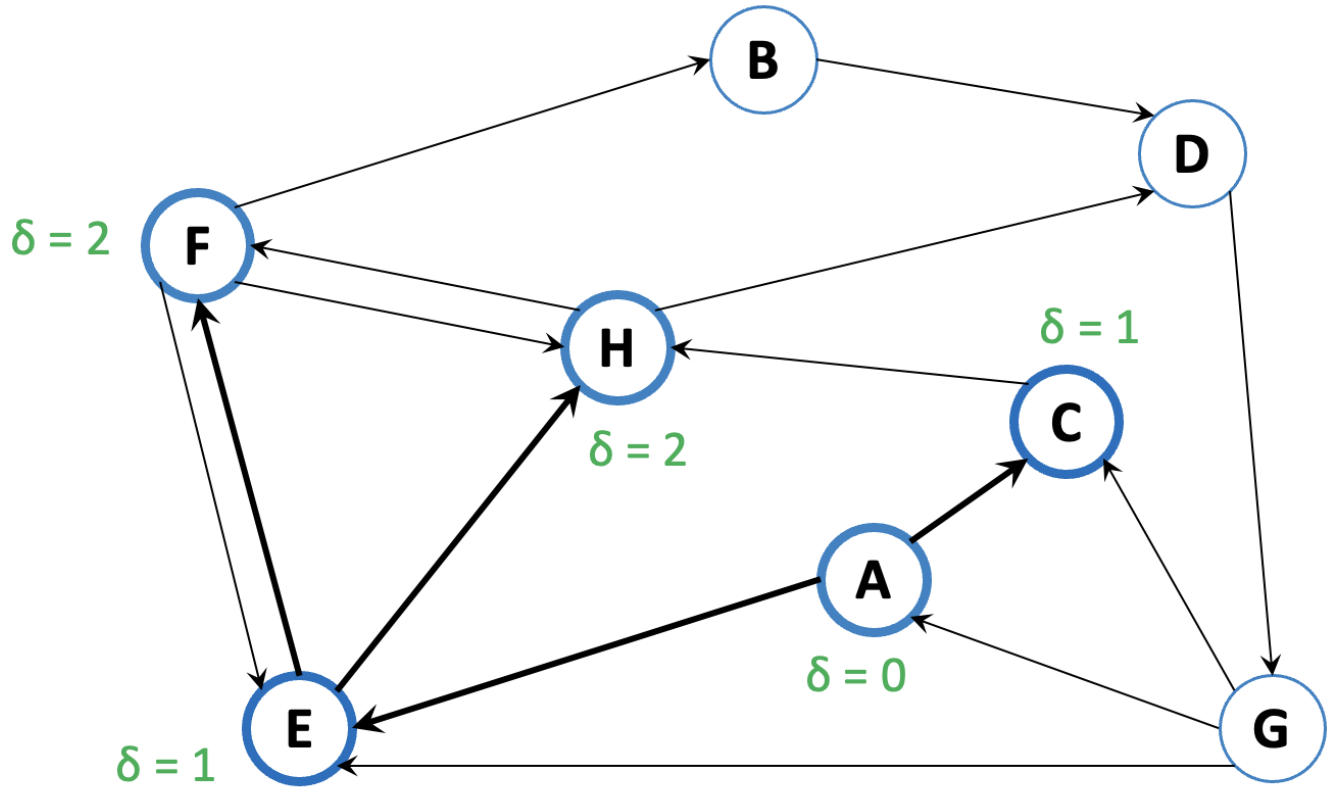
Ahora visitamos los nodos con $\delta = 1$ desde A



Los nodos a distancia $\delta = 1$ son alcanzables por una arista directa desde A

BFS y rutas más cortas

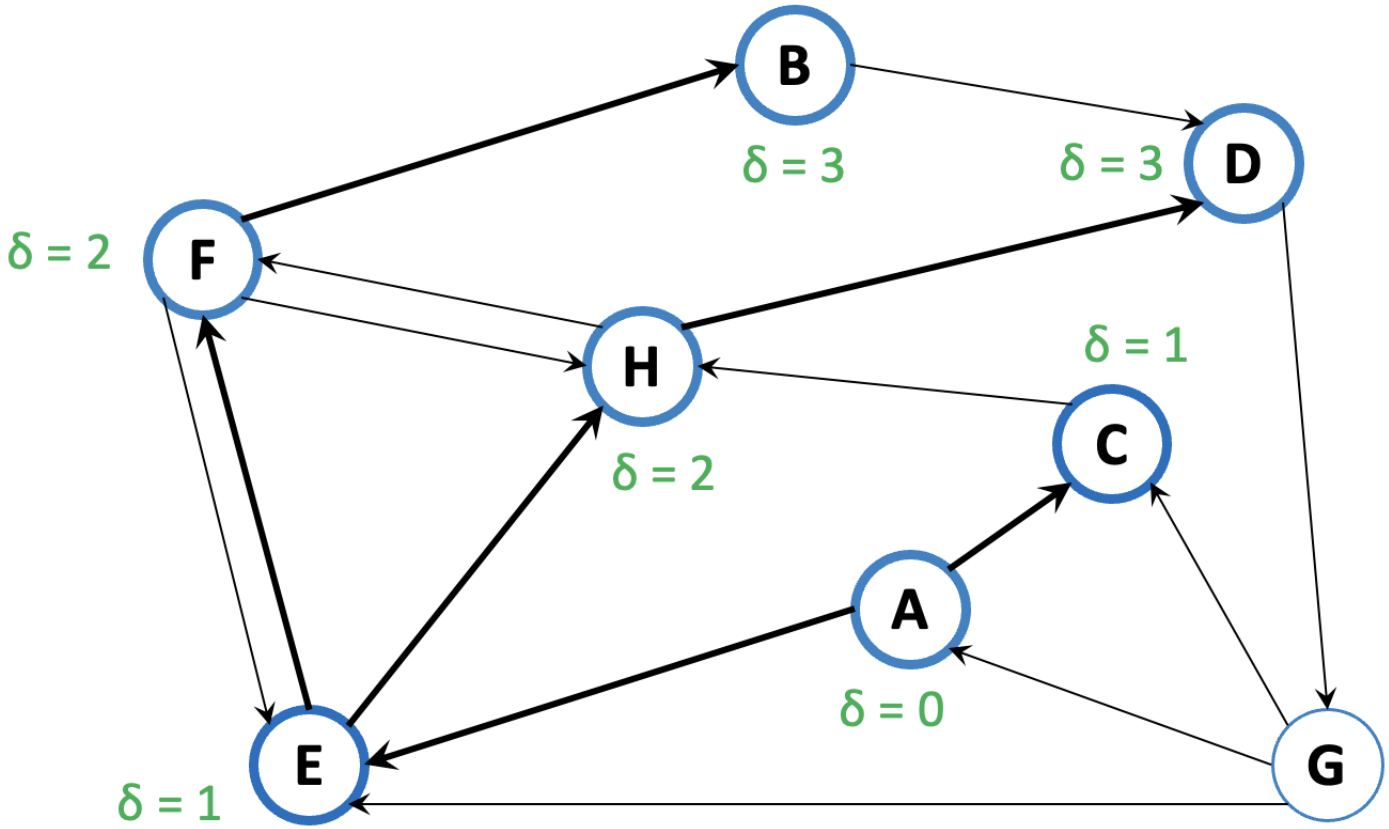
Ahora visitamos los nodos con $\delta = 2$ desde A



Notemos que hay dos caminos de largo 2 desde A hasta H

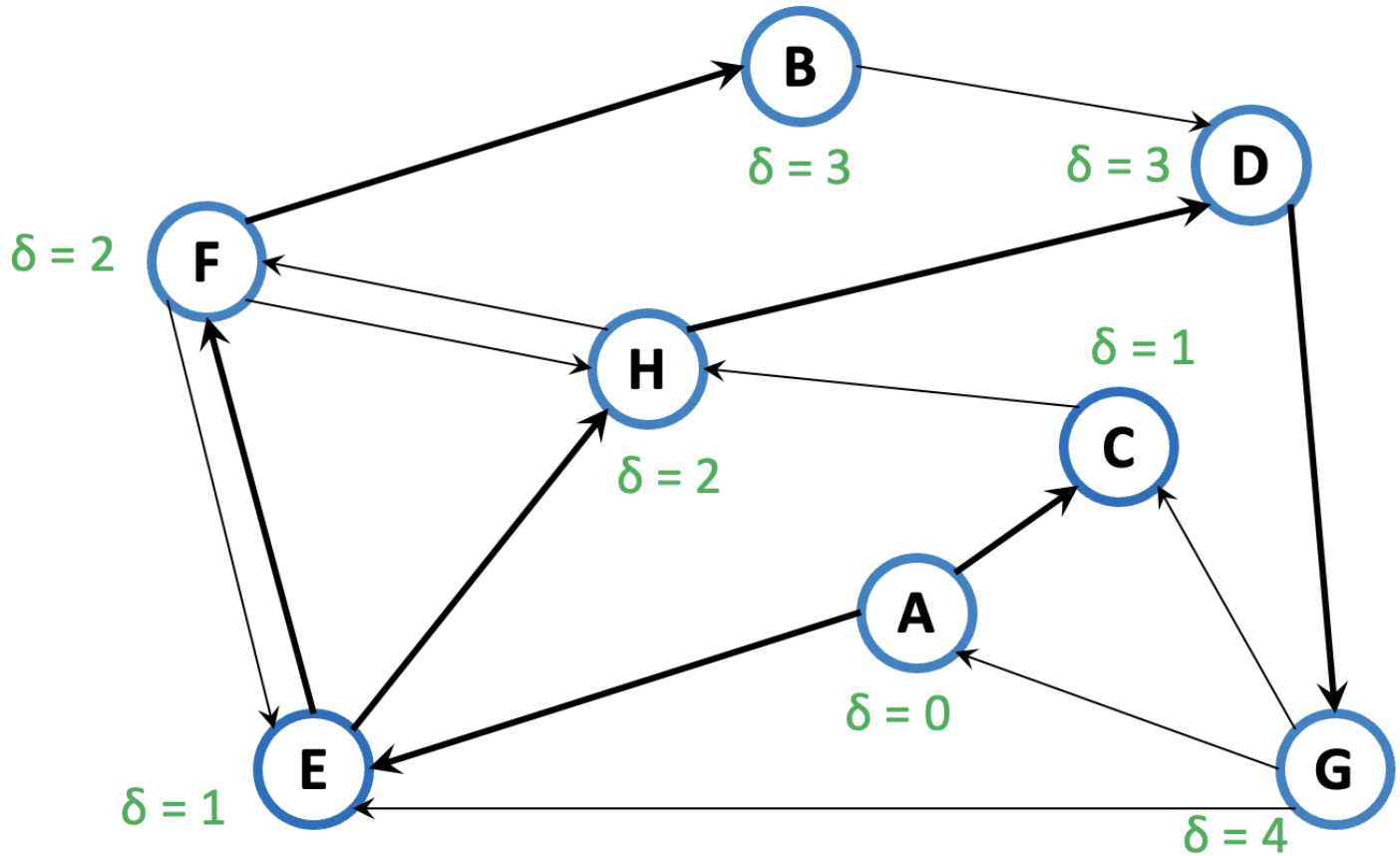
BFS y rutas más cortas

Ahora visitamos los nodos con $\delta = 3$ desde A



BFS y rutas más cortas

Ahora visitamos los nodos con $\delta = 4$ desde A



Notemos que D no es descubierto desde B

Búsqueda en amplitud

BFS descubre los nodos a través del menor número de aristas

- Siempre se visitan primero los nodos a distancia $\delta = k$
- Luego se llega a los que están a distancia $\delta = k + 1$

Debemos asegurar que los nodos son descubiertos en el orden adecuado

¿Cómo distinguir entre descubiertos y no descubiertos?

Implementación de BFS

Podemos distinguir los nodos con *colores*

- blanco: no descubierto
- gris: descubierto con vecinos no descubiertos (pendientes)
- negro: descubierto con vecinos descubiertos (terminado)

Como interesa descubrir nodos **en orden**, hay que almacenar los recién descubiertos

- Usaremos una **cola FIFO**
- Cuando descubrimos un nodo, lo agregamos al final de la cola
- Para revisar nuevos vecinos, sacamos el nodo prioritario

¿DFS se puede pensar con una estrategia análoga?

Implementación de BFS

input : vértice de inicio s

`BFS`(s)**:**

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

$u.color \leftarrow$ blanco; $u.\delta \leftarrow \infty$; $\pi[u] \leftarrow \varnothing$

$s.color \leftarrow$ gris; $s.\delta \leftarrow 0$; $\pi[s] \leftarrow \varnothing$

$Q \leftarrow$ cola vacía

`Insert`(Q, s)

while Q *no está vacía* **:**

$u \leftarrow$ `Extract`(Q)

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

if $v.color = blanco$ **:**

$v.color \leftarrow$ gris; $v.\delta \leftarrow u.\delta + 1$

$\pi[v] \leftarrow u$

$Q.enqueue(v)$

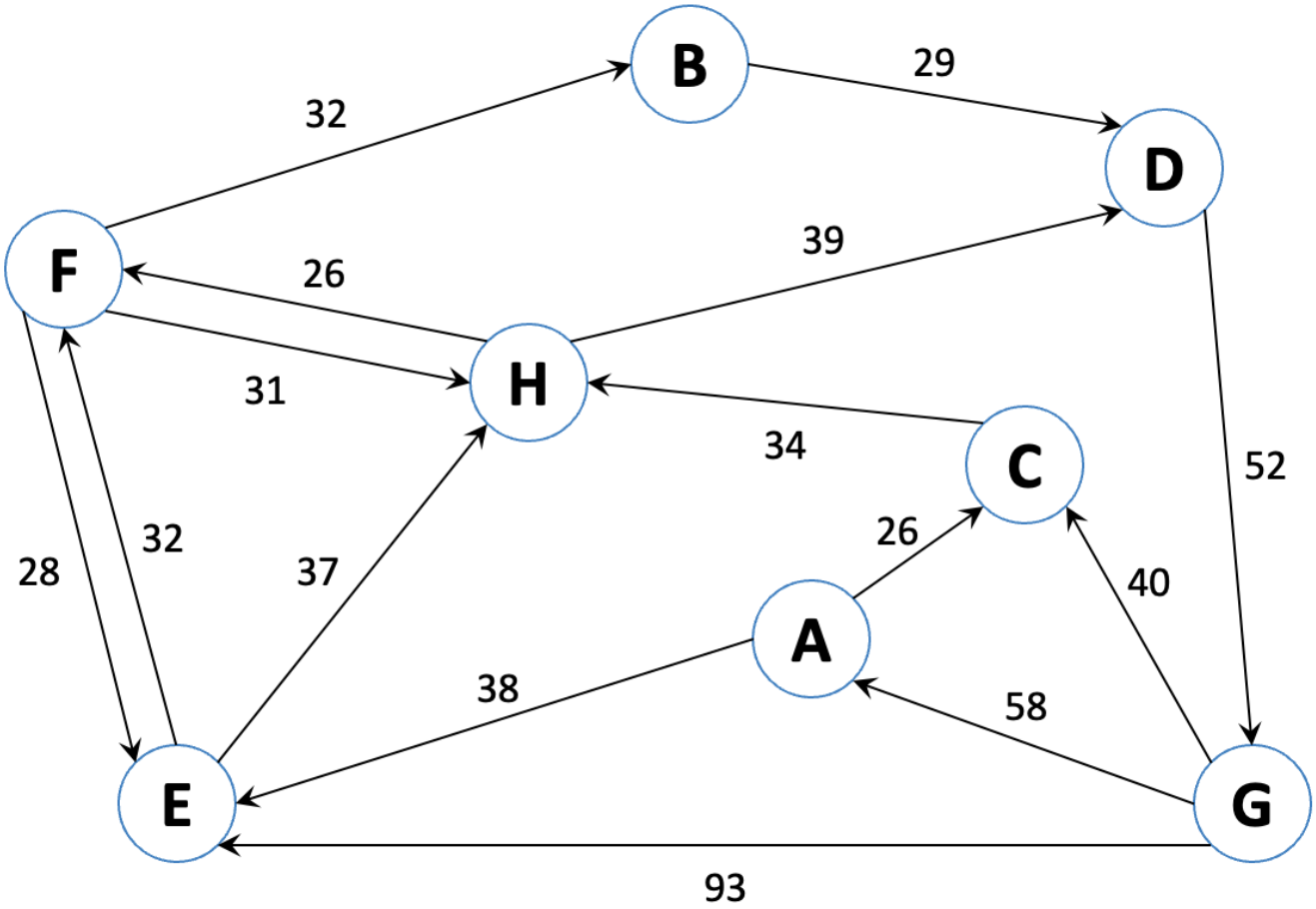
$u.color \leftarrow$ negro

Implementación de BFS

```
BFS(s):
    for u ∈ V − {s} :
        u.color ← blanco;  u.δ ← ∞;  π[u] ← ∅
    s.color ← gris;  s.δ ← 0;  π[s] ← ∅
    Q ← cola vacía
    Insert(Q, s)
    while Q no está vacía :
        u ← Extract(Q)
        for v ∈ α[u] :
            if v.color = blanco :
                v.color ← gris;  v.δ ← u.δ + 1
                π[v] ← u
                Q.enqueue(v)
        u.color ← negro
```

BFS deja en π la representación de un árbol de **rutas más cortas** desde s

Rutas en viajes: versión 2.0



¿BFS funciona cuando queremos rutas más baratas con costos?

Sumario

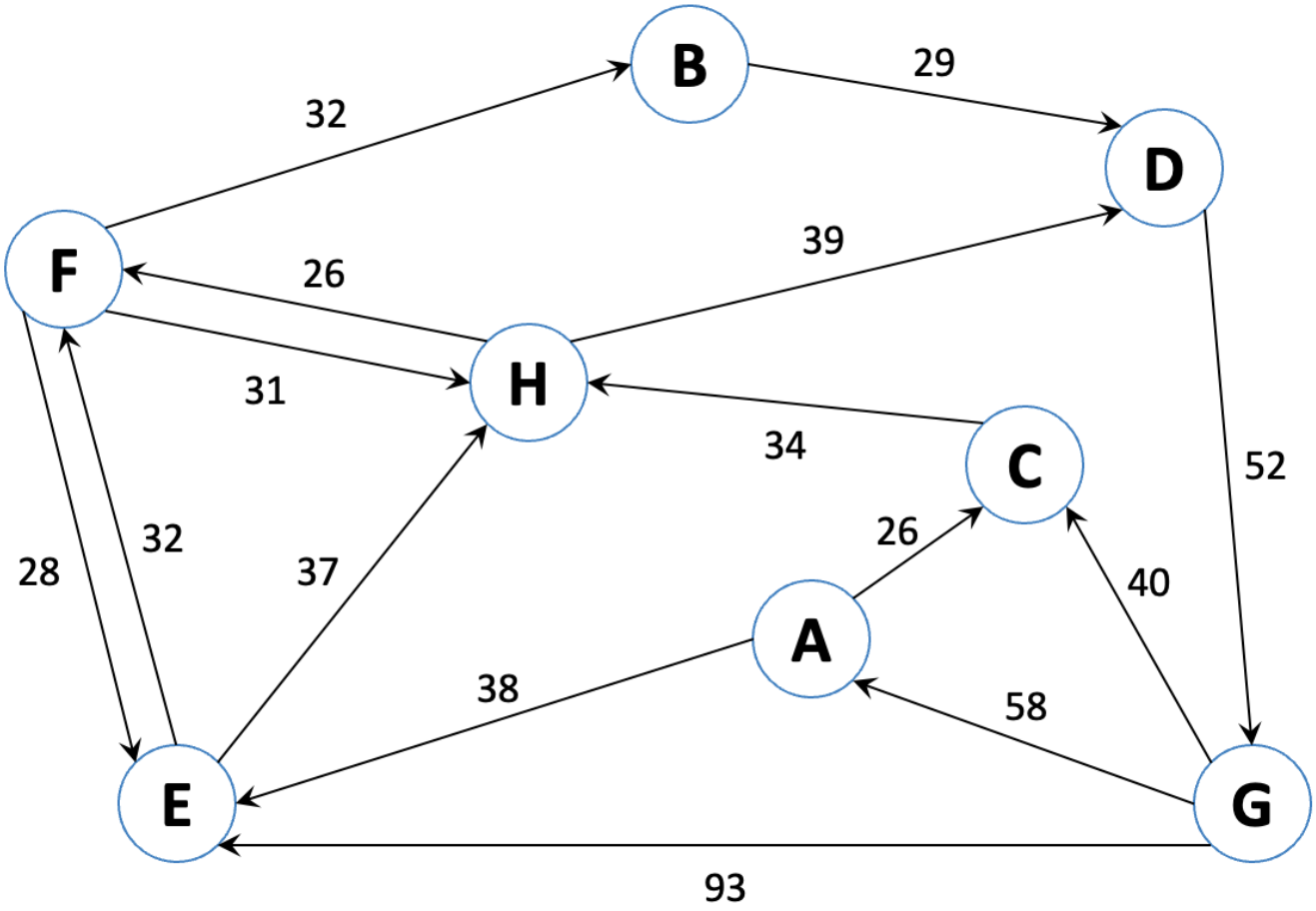
Búsqueda en amplitud *mejorado*

Necesitamos extender BFS para rutas con costos acumulados

- La cola Q debe incorporar los costos
- Al sacar un elemento de Q, lo hemos descubierto por la ruta más barata

¿Qué partes del algoritmo BFS debemos modificar?

Rutas en viajes: versión 2.0

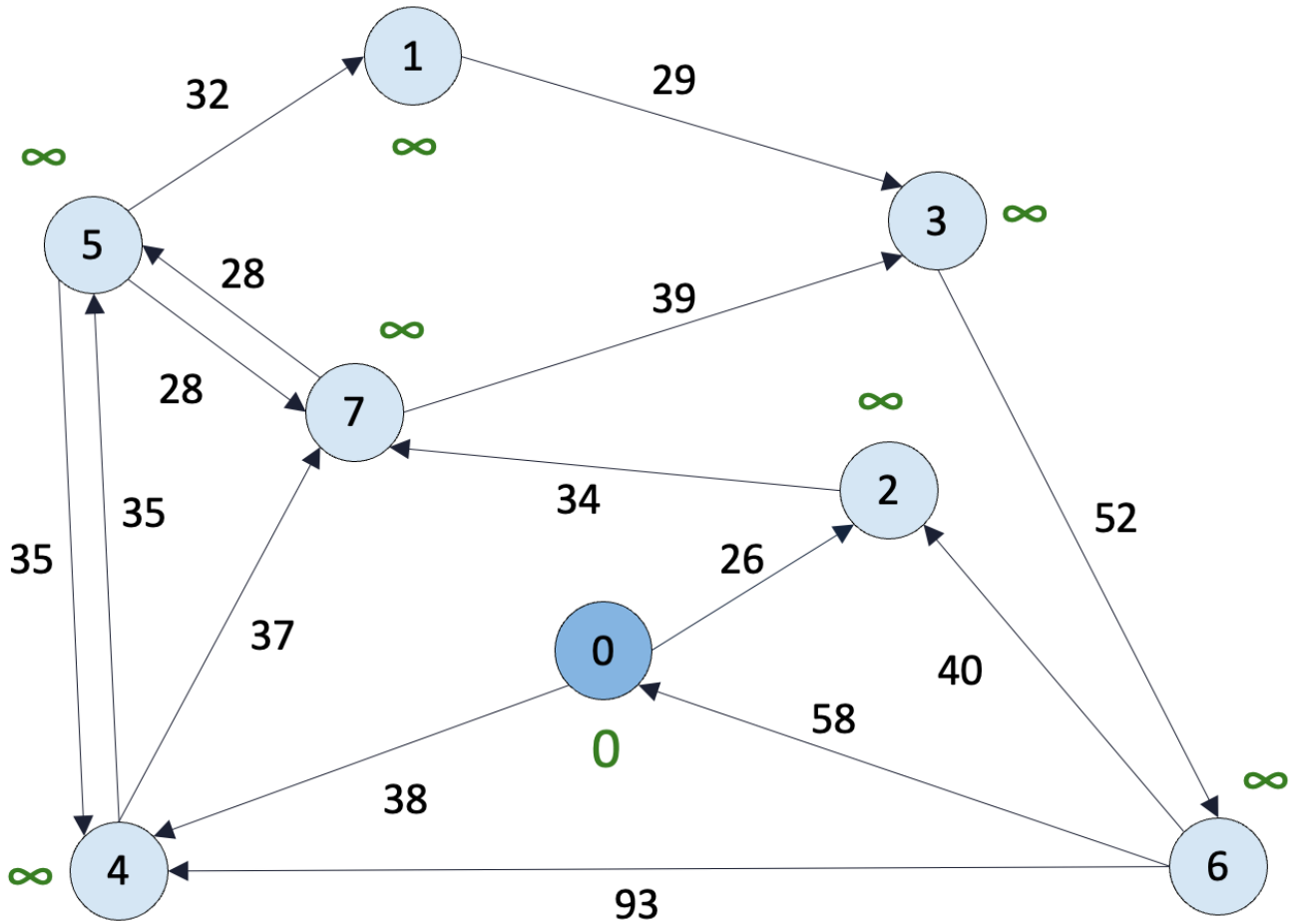


¿BFS funciona cuando queremos rutas más baratas con costos?

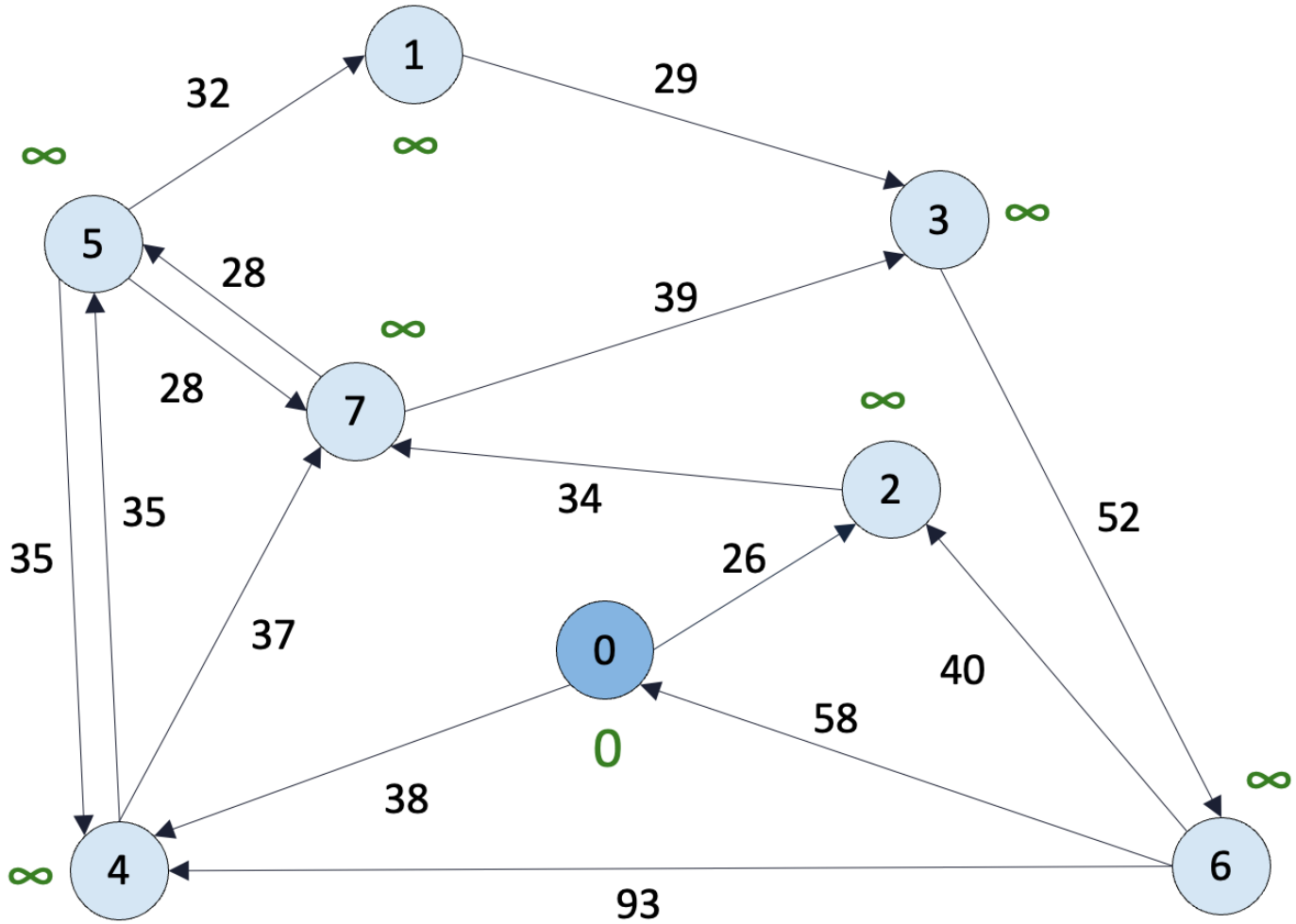
Rutas en viajes: versión 2.0

Ejercicio

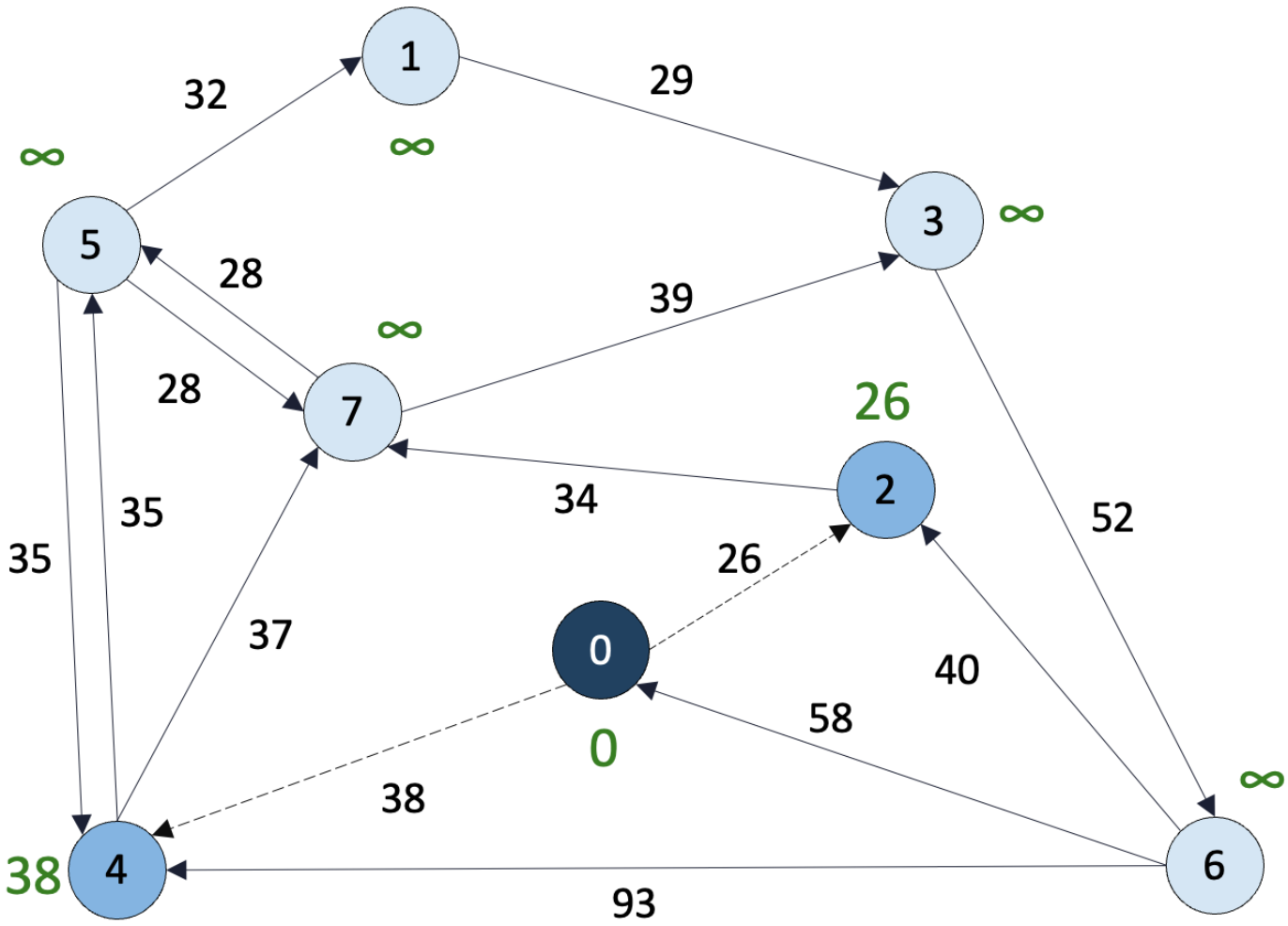
Determine la ruta más barata desde el nodo 0 hasta todos los demás nodos de la siguiente red con costos no negativos



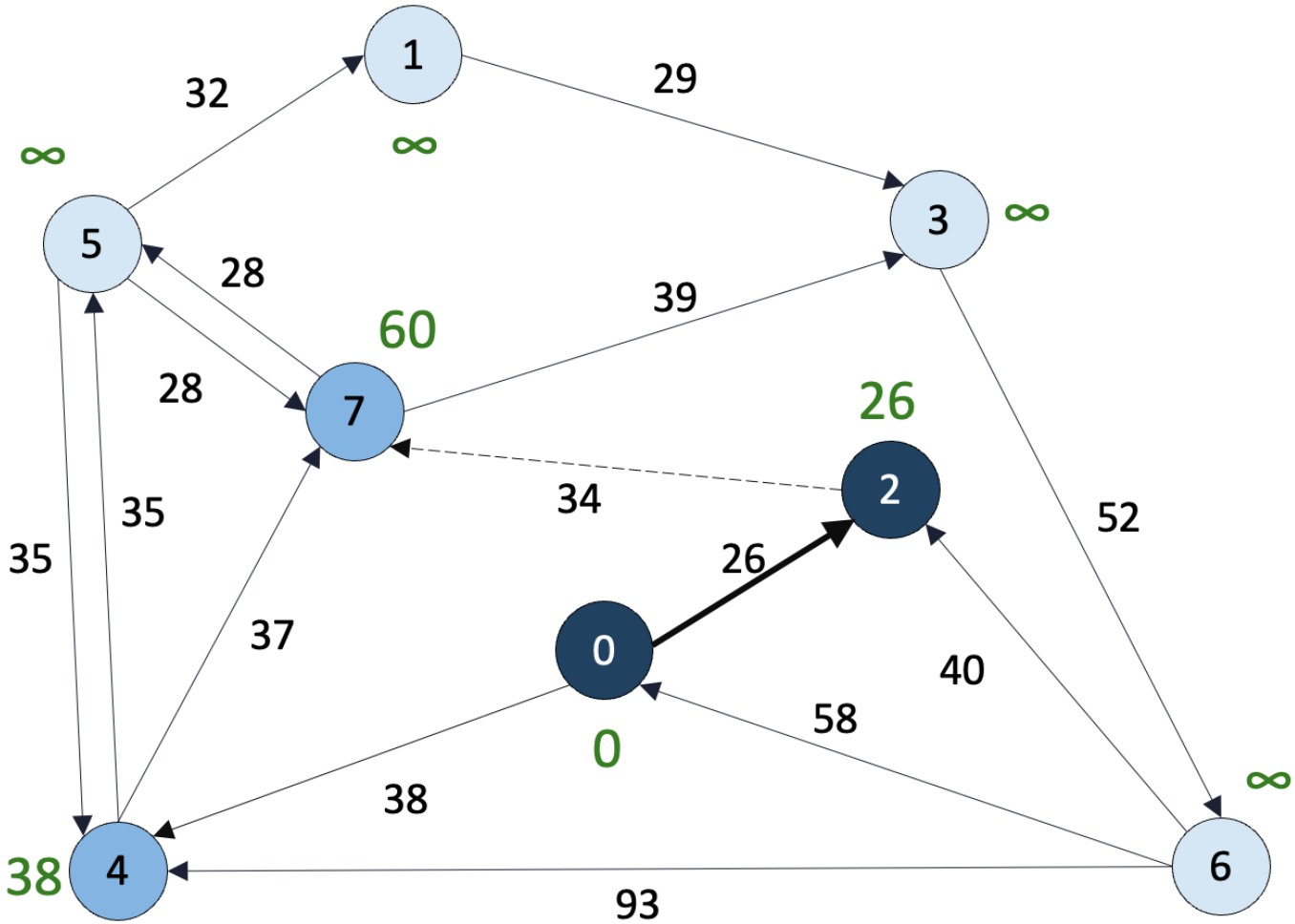
Rutas en viajes: versión 2.0



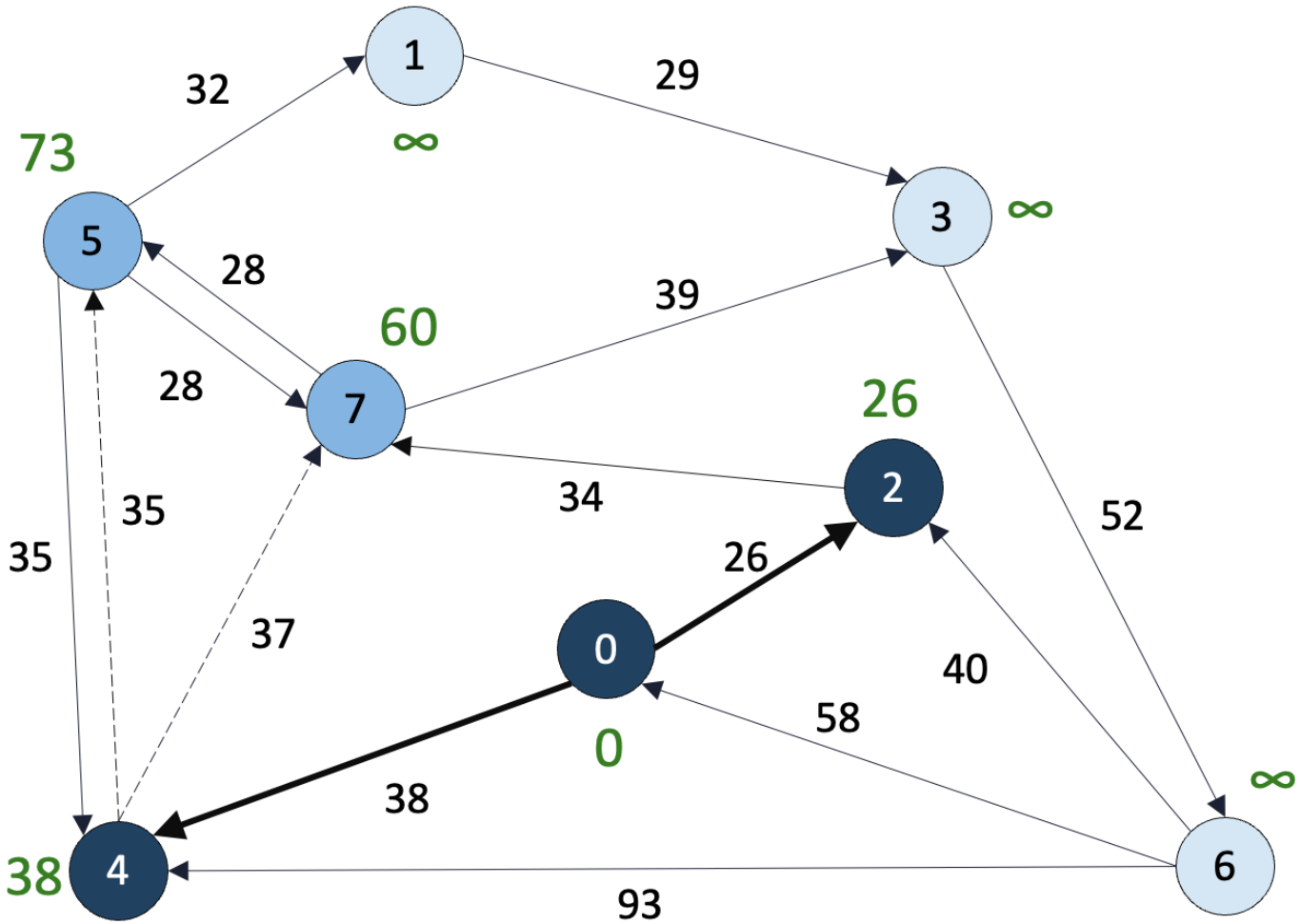
Rutas en viajes: versión 2.0



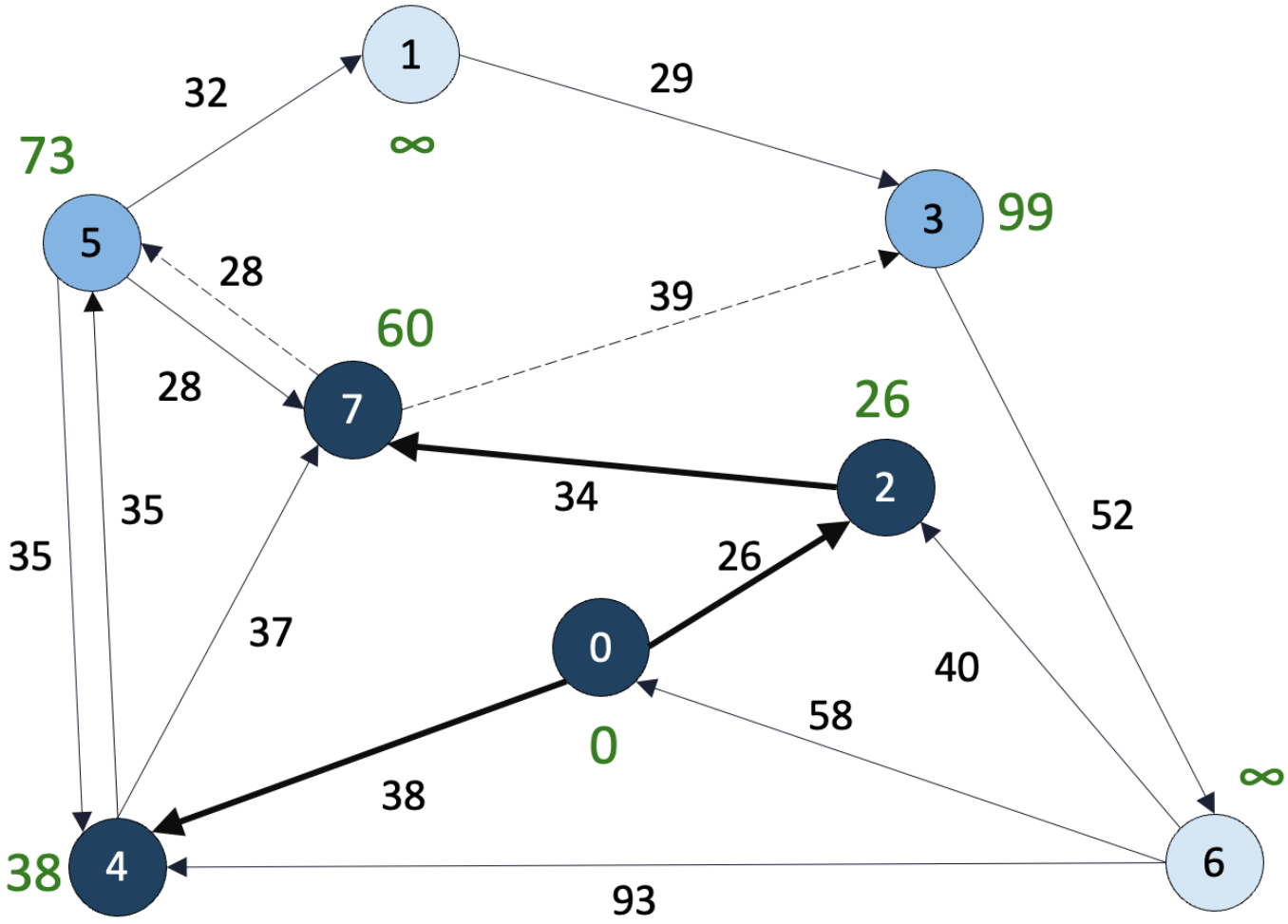
Rutas en viajes: versión 2.0



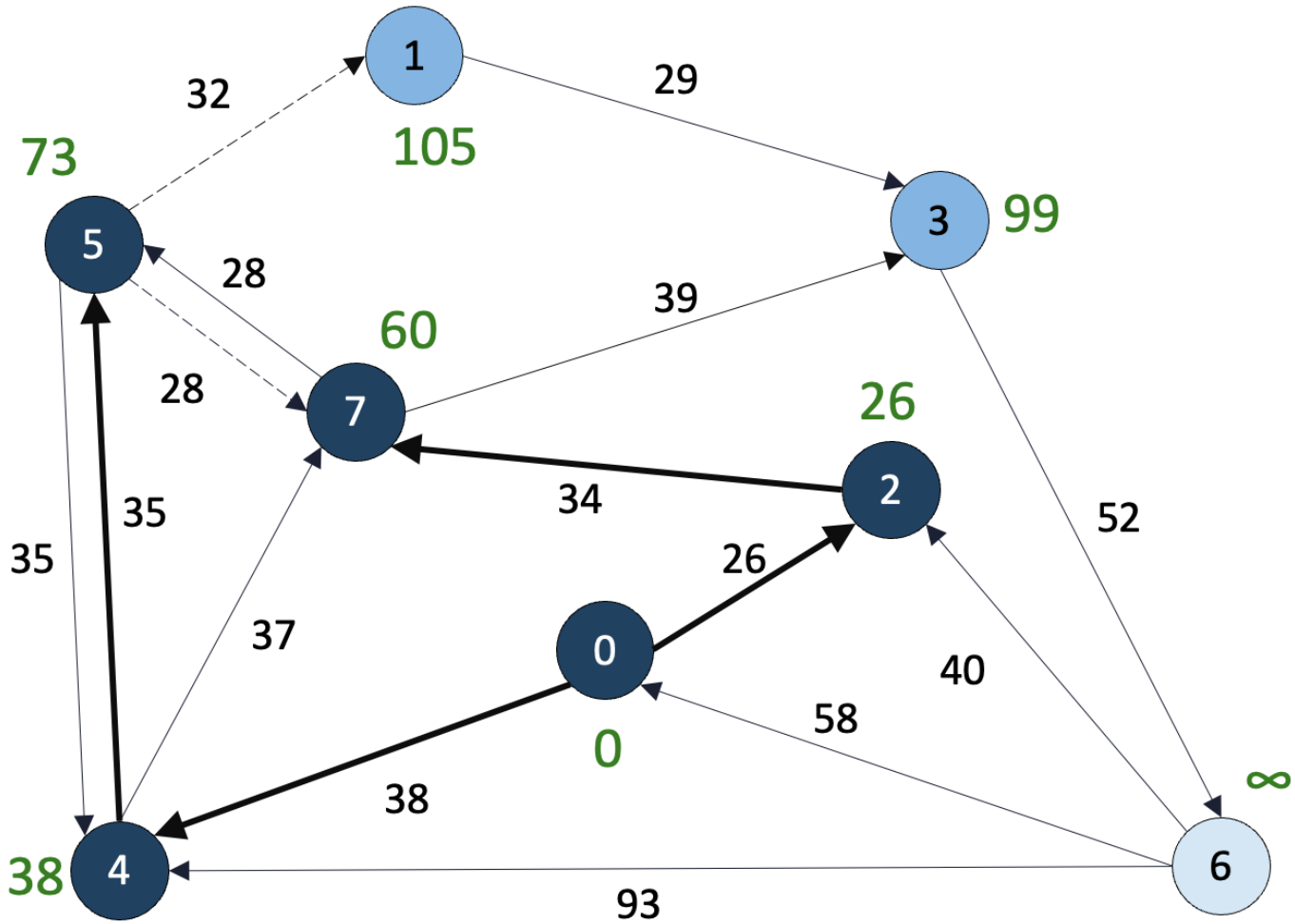
Rutas en viajes: versión 2.0



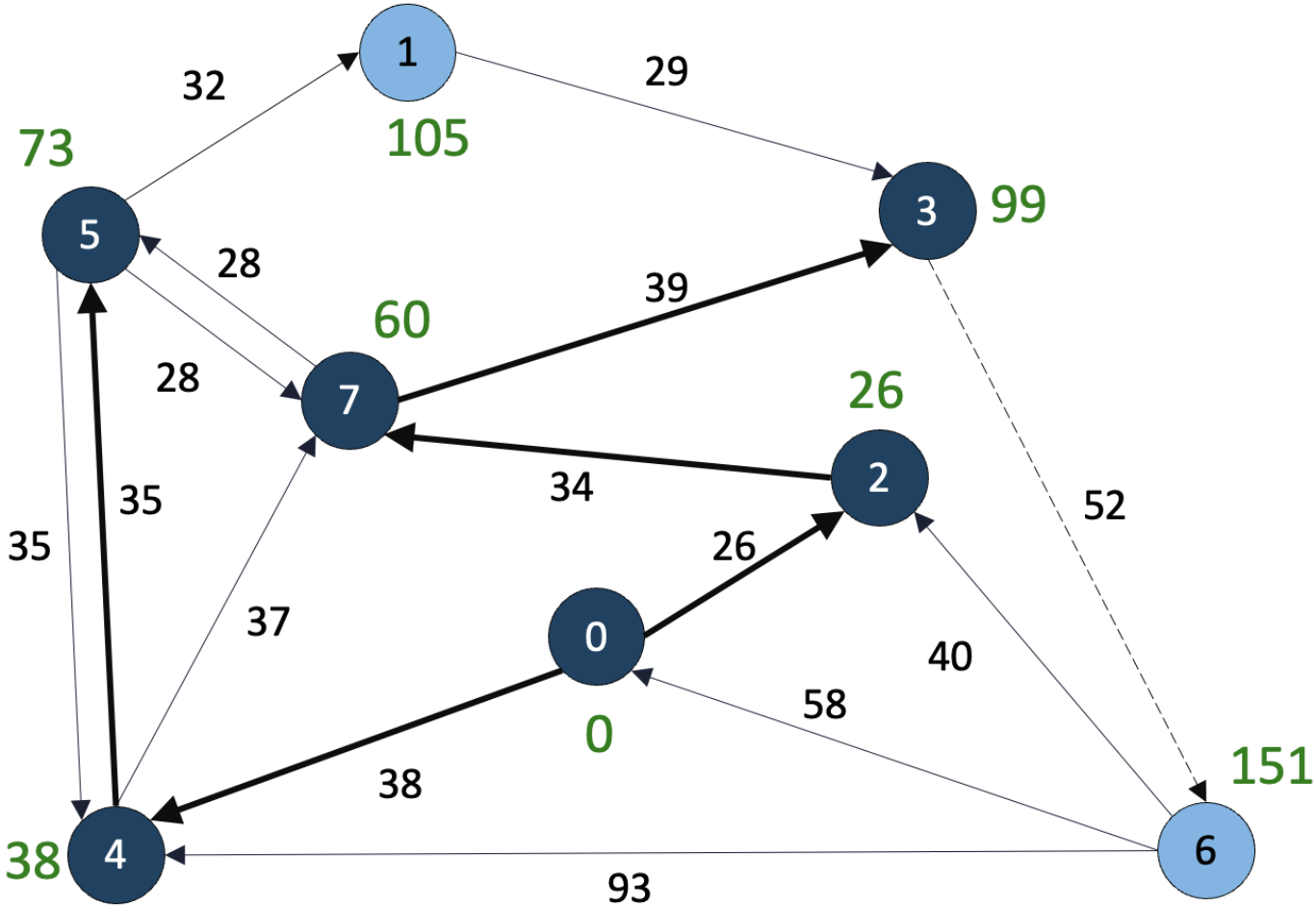
Rutas en viajes: versión 2.0



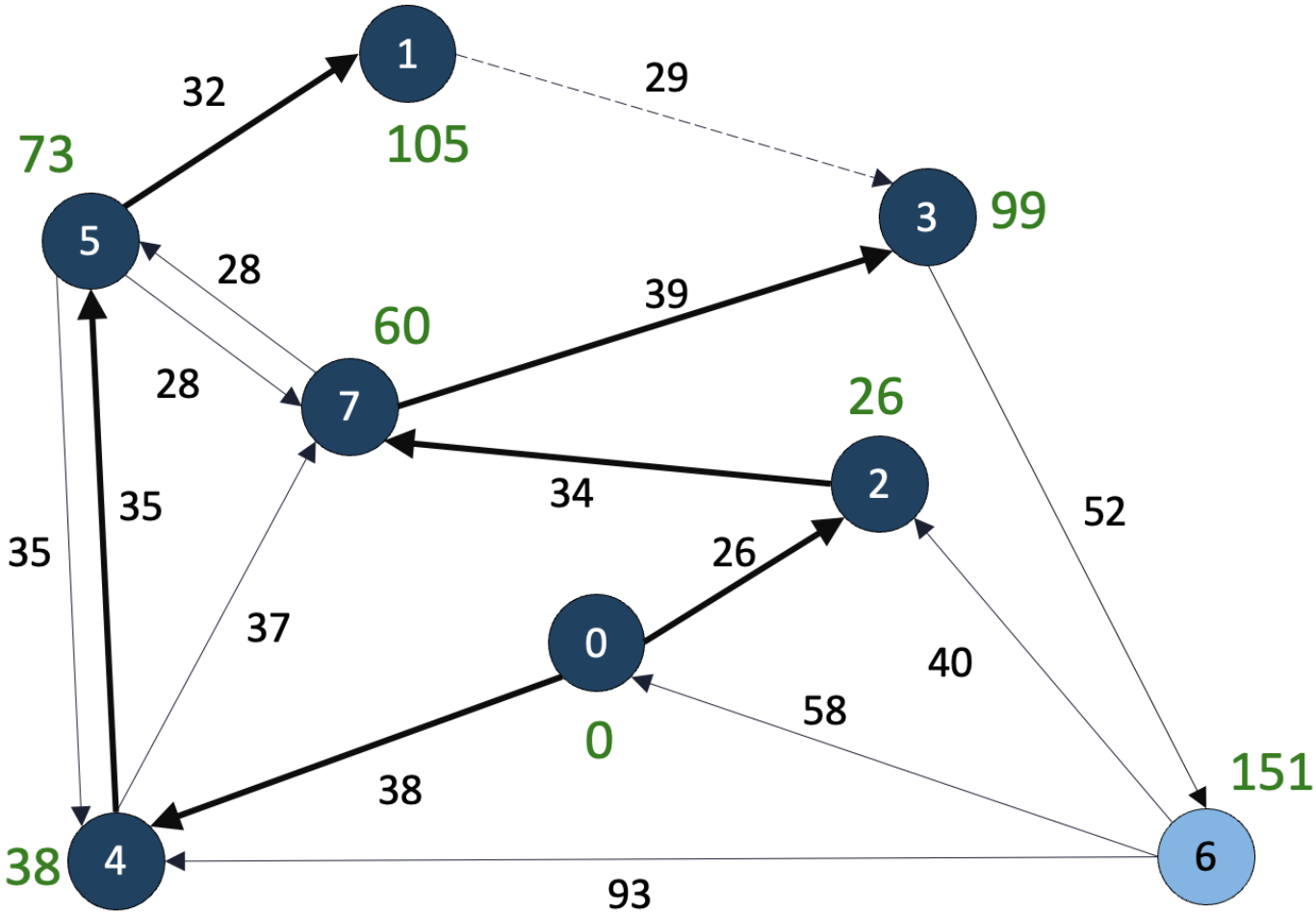
Rutas en viajes: versión 2.0



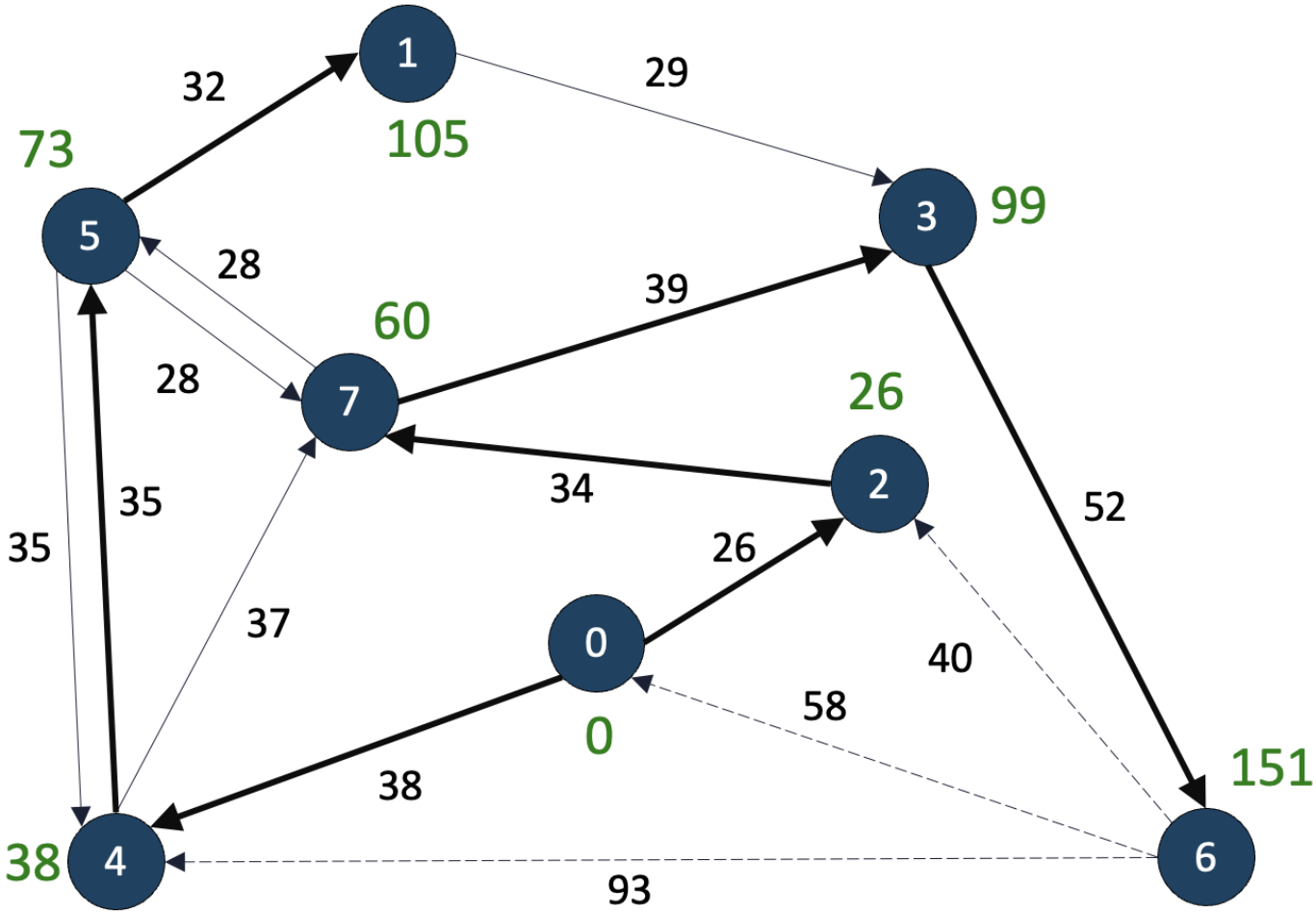
Rutas en viajes: versión 2.0



Rutas en viajes: versión 2.0



Rutas en viajes: versión 2.0



Algoritmo de Dijkstra

```
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
    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
                if v.color =blanco :
                    v.color ← gris;  Q.enqueue(v)
        u.color ← negro
```

Dijkstra solo cambia el costo y ruta si
se encuentra una arista que baja el costo

Algoritmo de Dijkstra

El algoritmo de Dijkstra encuentra las rutas más baratas desde s

- Solo a aquellos vértices alcanzables desde s
- Ojo: puede haber más de una ruta con el mismo costo
- Dijkstra encuentra **una**

¿Qué estrategia algorítmica es usada por este algoritmo?

Algoritmo de Dijkstra

El algoritmo de Dijkstra es **codicioso**

Además el problema de rutas más cortas tiene **subestructura óptima**

Dada una ruta entre v_0, v_k

- Si $p = v_0, v_1, \ldots, v_k$ es una ruta de costo mínimo
- Entonces la ruta

$$p_{ij} = v_i, \ldots, v_j, \qquad 0 \leq i \leq j \leq k$$

es una ruta más corta de v_i a v_j

Si es codicioso, ¿cómo sabemos que funciona en todas las instancias?

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 $d[s] = 0$.

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

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

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

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

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). □

Uso de la cola de prioridad

Ya vimos que la cola de prioridad implementada como heap permite

- Insertar valores con prioridad
- Extraer el más prioritario

Para Dijsktra además necesitamos **actualizar** prioridad cuando cambie el costo óptimo

1. Cambiamos prioridad del nodo del heap
2. Hacemos intercambios hacia arriba si es necesario

La propiedad de heap permite que esta operación
solo afecte nodos en la ruta del nodo a la raíz

Uso de la cola de prioridad

input : heap representado como arreglo $H[0 \ldots n-1]$,
índice i,
nueva prioridad $k > H[i]$

`IncreaseKey`(H, i, k)**:**
  $H[i] \leftarrow k$
  while $i > 0 \wedge H[\texttt{Parent}(i)] < H[i]$ **:**
    $H[i] \leftrightharpoons H[\texttt{Parent}(i)]$
    $i \leftarrow \texttt{Parent}(i)$

En tiempo $\mathcal{O}(\log(V))$ actualizamos la prioridad y mantenemos el heap

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 con `IncreaseKey` es $\mathcal{O}(\log(V))$

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

Algunas variantes

Podemos adaptar el algoritmo para resolver otros problemas comunes

- Rutas más cortas en grafos acíclicos
- Rutas más cortas de un vértice a otro (específico)
- Rutas más cortas entre **todos** los pares de vértices
- Rutas más cortas en grafos Euclideanos