



# Grafos IV : Shortest Path



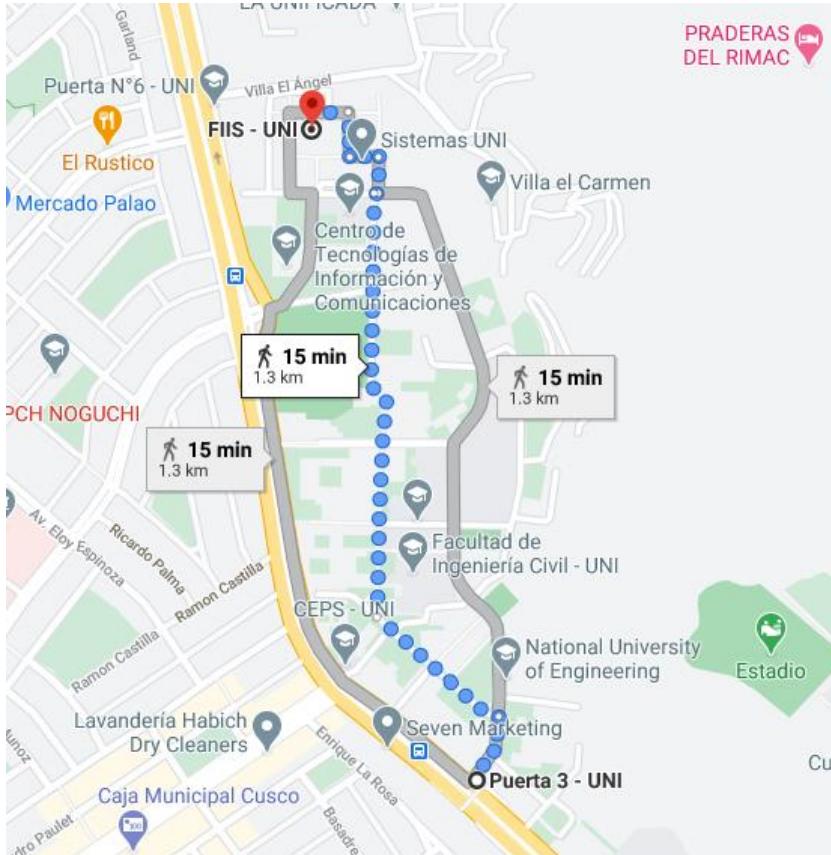
# Contenido

1. Introducción	
2. Propiedades de los Shortest Path y Algoritmo Genérico	
3. Single Source Shortest Path en DAG	
4. Single Source Shortest Path con pesos unitarios	
5. Single Source Shortest Path con pesos no negativos	
6. Single Source Shortest Path con cualquier peso real	
7. All-pairs shortest path	

# Contenido

1. Introducción	→
2. Propiedades de los Shortest Path y Algoritmo Genérico	→
3. Single Source Shortest Path en DAG	→
4. Single Source Shortest Path con pesos unitarios	→
5. Single Source Shortest Path con pesos no negativos	→
6. Single Source Shortest Path con cualquier peso real	→
7. All-pairs shortest path	→

# Motivación

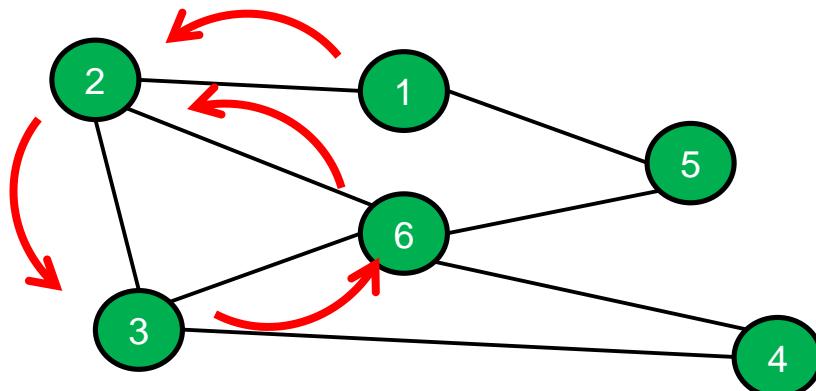


# Conceptos básicos

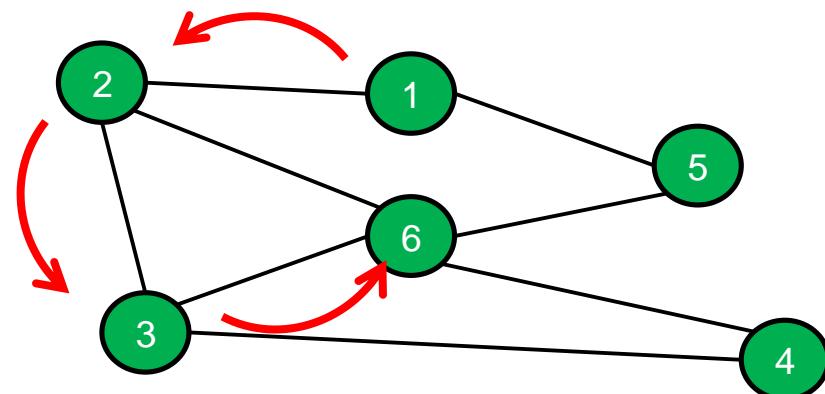
Sea un grafo  $G(V, E)$ . Definimos los siguientes conceptos (tanto para grafos dirigidos como no dirigidos):

**Walk:** Definimos un **walk** como una secuencia de nodos  $v_1, v_2, \dots, v_k$  tal que existe una arista entre cada par de nodos consecutivos de la secuencia. Es decir  $(v_i, v_{i+1}) \in E$ , para  $1 \leq i < k$ . Lo simbolizaremos como  $v_1 \rightarrow v_2 \rightarrow \dots \rightarrow v_k$ . Donde  $u \rightarrow v$  significa que hay una arista entre  $u$  y  $v$

**Path:** Un **path** es un walk que no tienen ningún nodo repetido



Walk : $1 \rightarrow 2 \rightarrow 3 \rightarrow 6 \rightarrow 2$



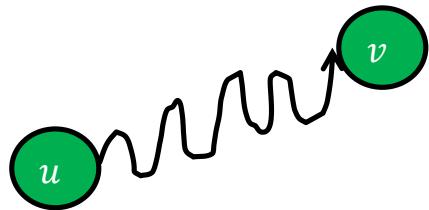
Path : $1 \rightarrow 2 \rightarrow 3 \rightarrow 6$

# Conceptos básicos

## Simbología de Paths:

Cuando queremos representar un path entre dos nodos  $u$ ,  $v$  y no nos importa mucho los nodos intermedios, podemos simbolizar eso como  $u \rightsquigarrow v$

Gráficamente lo podemos ver de esta forma:

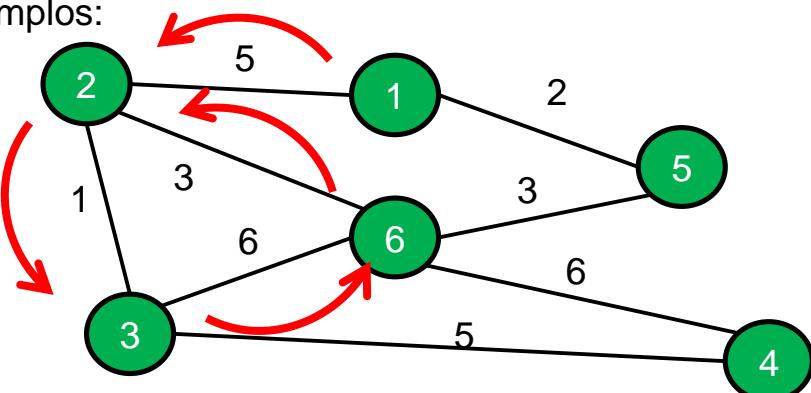


# Conceptos básicos

Sea un grafo con pesos  $G(V, E, w)$ . Cuya función de pesos es  $w: E \rightarrow \mathbb{R}$ .

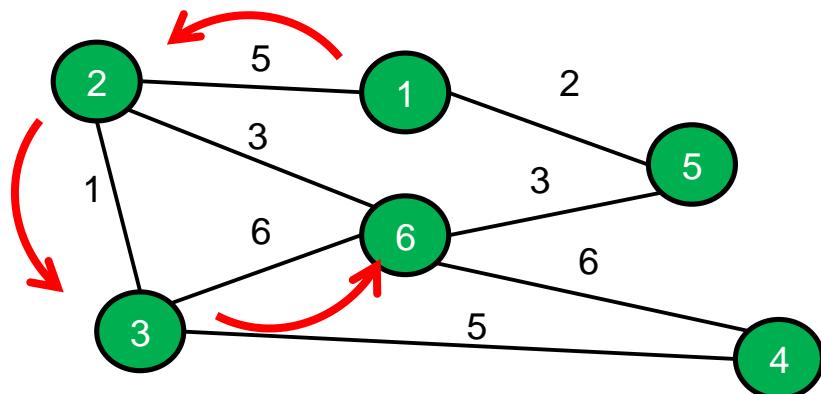
**Peso/Longitud de un path/walk:** Es la sumatoria de pesos de cada una de las aristas del path o del walk. Es decir, si se tiene el path/walk  $p$ , la longitud o peso del path/walk sería  $w(p) = \sum_{e \in p} w(e)$ . En caso el path/walk no tenga aristas se define su peso como 0.

Ejemplos:



Walk :  $1 \rightarrow 2 \rightarrow 3 \rightarrow 6 \rightarrow 2$

Peso : 15



Path :  $1 \rightarrow 2 \rightarrow 3 \rightarrow 6$

Peso : 12

# Conceptos básicos

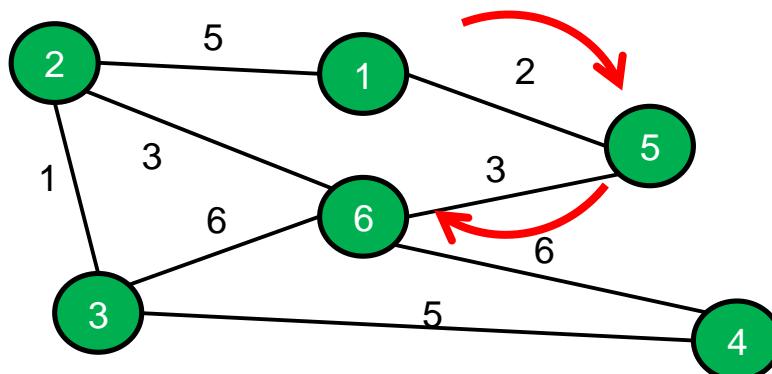
Sea un grafo con pesos  $G(V, E, w)$ . Cuya función de pesos es  $w: E \rightarrow \mathbb{R}$ .

**Shortest path/walk:** Sean los nodos  $s, t$  definimos el *shortest path* (o walk) entre ambos nodos como aquel path (o walk)  $v_1 \rightarrow v_2 \rightarrow \dots \rightarrow v_k$  con  $v_1 = s$ ,  $v_k = t$  que tenga peso mínimo.

Ejemplo:  $s = 1, t = 6$

**Nota:**

Pueden haber múltiples  
Shortest paths/walks



Shortest Walk:  $1 \rightarrow 5 \rightarrow 6$ , peso = 5

Shortest Path :  $1 \rightarrow 5 \rightarrow 6$ , peso = 5

# Preguntas

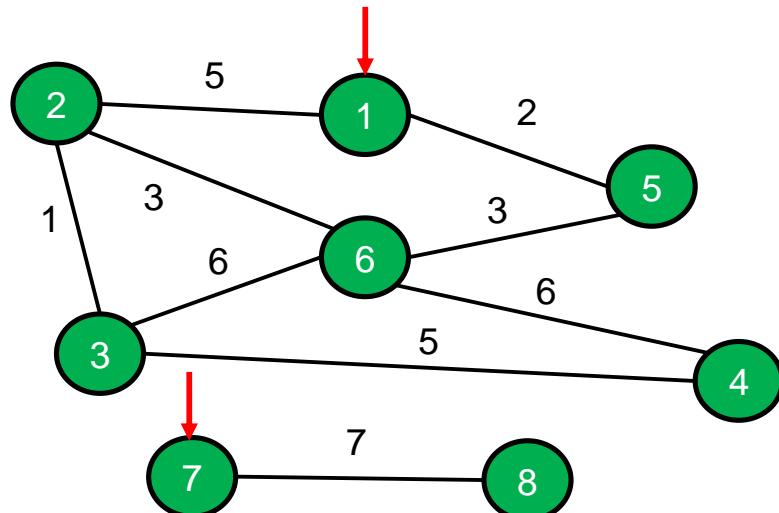
1. ¿Siempre existe shortest path o shortest walk?
2. ¿Un shortest walk siempre es un shortest path?

# ¿Siempre existe shortest path / shortest walk?

Caso 1: No existe ningún path/walk entre  $s, t$

a) Para grafos no dirigidos, considere un grafo no conexo

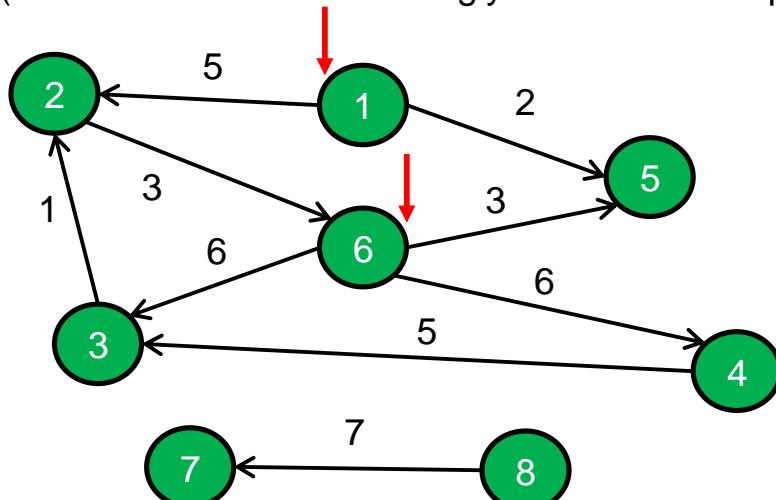
Ejemplo : No existe ningún path/walk entre 1 y 7



(En este caso podemos definir al peso como  $+\infty$ )

b) Para grafos dirigidos, considere un grafo que no sea fuertemente conexo

Ejemplo : No hay ningún path/walk desde 6 hasta 1  
(no están en la misma strongly connected component)



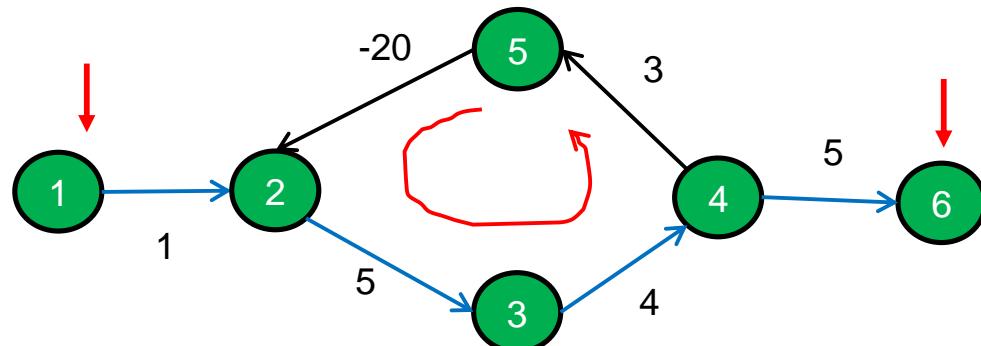
# ¿Siempre existe shortest path / shortest walk?

## Caso 2: Existe algún ciclo negativo alcanzable desde $s$

Considere un grafo **dirigido** con pesos negativos en el que existen ciclos negativos. Entonces el **shortest walk** no existe ya que siempre puedo seguir pasando por el ciclo negativo y volviendo al peso total más negativo por lo cual tiende a  $-\infty$ . Sin embargo, el **shortest path** sí existe.

En caso haya un grafo **no dirigido**, basta que haya alguna arista negativa para formar el ciclo negativo.

Ejemplo : Considere el shortest walk entre 1 y 6. Cada vez que paso por el ciclo pintado de rojo, siempre haré que el peso total disminuya en 8. Note que el shortest path sí existe y tiene valor de peso 15



Shortest Walk: No definido, peso  $\rightarrow -\infty$   
Shortest Path :  $1 \rightarrow 2 \rightarrow 4 \rightarrow 6$ , peso = 15

Hallar un shortest path (**no hay repetición de vértices**) en un grafo con ciclos negativos es un problema complicado (NP-Hard)

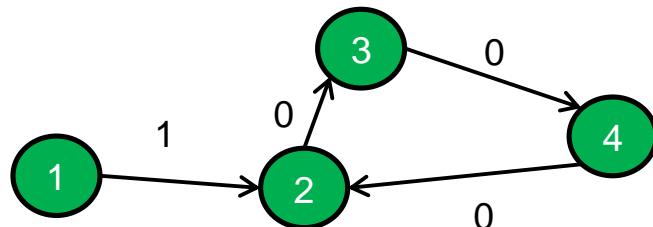


# ¿Un shortest walk siempre es un shortest path?

La diapositiva anterior mostraba que era posible que el shortest walk no exista mientras que el shortest path cuando hay ciclos negativos.

Sin embargo, **si no existen ciclos negativos en el grafo**, el peso de cualquier shortest walk entre dos nodos  $s, t$  es igual al peso de cualquier shortest path entre dos nodos  $s, t$ . Esto debido a que si no hay ciclos negativos, tomar un ciclo solo empeora la respuesta o la deja igual (en caso sea un ciclo de suma cero), por lo que no hay una ventaja de un walk respecto de un path.

Respecto a la pregunta: si no hay ciclos negativos, casi siempre se cumplirá que un shortest walk sea también un shortest path (la excepción es cuando existen ciclos de peso 0). Es por eso que a partir de ahora solo usaremos el término **shortest path**



Ejemplo:  
1 → 2 → 3 → 4 → 2 es un shortest walk pero no es shortest path

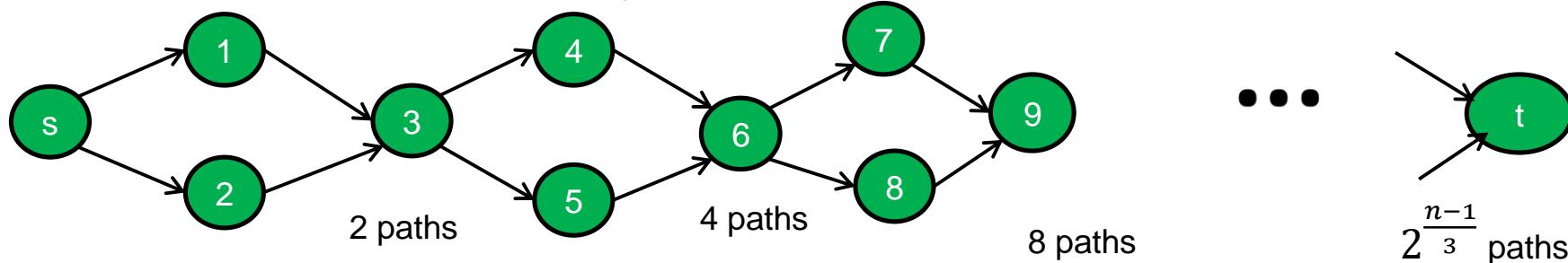
# Número de paths

Antes de ver algunos algoritmos que resuelven el problema, convendría saber una cota del número de paths que puede haber en un grafo cualquiera, para ver por qué la **fuerzabruta** no es viable.

En general podemos decir que el número de paths puede crecer de forma exponencial, e incluso en el orden de los factoriales.

En un grafo completo, el número de paths es aproximadamente  $(n - 2)! \times e$ .

También podemos producir otra clase de grafos con una cantidad exponencial de caminos:



También existen fórmulas para hallar la cantidad exacta de *walks* con cierto número de aristas:

[https://cp-algorithms.com/graph/fixed\\_length\\_paths.html](https://cp-algorithms.com/graph/fixed_length_paths.html)

# Variantes

Existen algunas variantes en la formulación del problema del shortest path :

- **Single source shortest path (SSSP):** Encontrar el shortest path desde un nodo fuente  $s$  hasta cualquier nodo  $u$  del grafo.
- **Single destination shortest path:** Encontrar el shortest path desde cualquier nodo  $u$  hasta un nodo terminal  $t$ . (Se puede resolver en la misma complejidad que el SSSP, revirtiendo las aristas y haciendo SSSP desde  $t$ ).
- **All-pairs shortest path (APSP):** Encontrar el shortest path entre todos los pares de nodos (se puede hacer aplicando SSSP  $n$  veces, donde  $n$  es el número de nodos, pero hay formas más eficientes).
- **Multi-source shortest path:** Para todo nodo  $u$  encontrar el shortest path que vaya desde un nodo fuente  $s \in S$  (conjunto de nodos fuentes) hasta  $u$  (se puede resolver agregando un nodo ficticio  $s'$  conectado a todos los nodos  $s \in S$  con arista de peso 0 y luego aplicando SSSP desde  $s'$ ).

# Contenido

1. Introducción	
<b>2. Propiedades de los Shortest Path y Algoritmo Genérico</b>	
3. Single Source Shortest Path en DAG	
4. Single Source Shortest Path con pesos unitarios	
5. Single Source Shortest Path con pesos no negativos	
6. Single Source Shortest Path con cualquier peso real	
7. All-pairs shortest path	

# Definiciones

- Llamaremos **distancia** entre  $u, v$ , denotado como  $\delta(u, v)$ , al peso del shortest path entre  $u$  y  $v$ . En caso no exista camino entre  $u, v$  se le asignará el valor de  $+\infty$ . Si hay un ciclo negativo entre ambos nodos, se le asignará el valor de  $-\infty$ . (En realidad el shortest path sí existe en este caso, pero abusaremos de la notación para referirnos a shortest walk). En la variante de SSSP, podremos utilizar  $\delta(u) = \delta(s, u)$
- Denotaremos como  $d(u, v)$  al **estimado de la distancia** calculado por nuestros algoritmos en cada iteración. En la variante de SSSP, podremos utilizar  $d(u) = d(s, u)$
- Definamos al operador  $+$  para *paths* o *walks* como la concatenación de *paths* o *walks*. Es decir, sea 2 *paths* (o 2 *walks*)  $a = a_0, a_1, \dots, a_{k1}$ ,  $b = b_0, b_1, b_2, \dots, b_{k2}$  (con  $a_{k1} = b_0$ ), entonces  
$$a + b = a_0, a_1, \dots, a_{k1}, b_0, b_1, b_2, \dots, b_{k2}$$
- **Nota:** Los algoritmos y propiedades que mencionaremos en las siguientes diapositivas serán formulados para grafos dirigidos. Pero también pueden ser utilizados para grafos no dirigidos transformando una arista no dirigida  $u - v$  en dos aristas  $u \rightarrow v$ ,  $v \rightarrow u$

# Propiedades

- $\delta(s, s) = 0$  si no hay ciclos negativos (trivial)
- **Optimal Substructure:** Los subpaths de un shortest path son también shortest path. Es decir, dado un shortest path  $p = v_0 \rightarrow v_1 \rightarrow \dots \rightarrow v_k$  que va de  $v_0$  a  $v_k$ . Entonces para cualquier  $i, j$  tal que  $0 \leq i \leq j \leq k$ , cumple que el subpath  $p_{ij} = v_i \rightarrow v_{i+1} \rightarrow \dots \rightarrow v_j$  también es un shortest path, pero que va del nodo  $v_i$  al nodo  $v_j$ .

## Demostración:

Por reducción al absurdo, supongamos que  $p_{ij}$  no fuera shortest path.

Entonces existe otro path  $p'_{ij}$  tal que  $w(p'_{ij}) < w(p_{ij})$

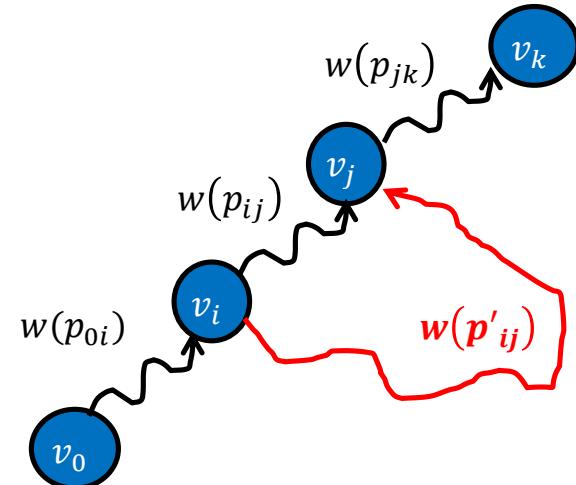
Además nosotros podemos descomponer a  $p$  en 3 caminos  $p_{0i}, p_{ij}, p_{jk}$

Por lo que  $w(p) = w(p_{0i}) + w(p_{ij}) + w(p_{jk})$

Sin embargo nosotros podríamos construir un nuevo path  $p'$

reemplazando  $p_{ij}$  por  $p'_{ij}$  en  $p$  tal que

$w(p') = w(p_{0i}) + w(p'_{ij}) + w(p_{jk}) < w(p)$ . Por lo que  $p$  no sería shortest path, llegando a un absurdo ■



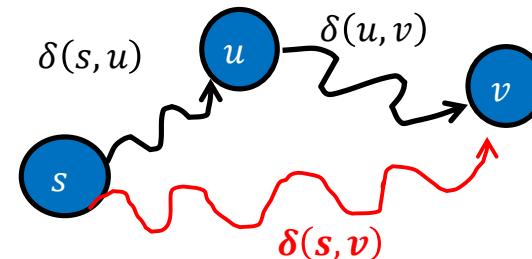
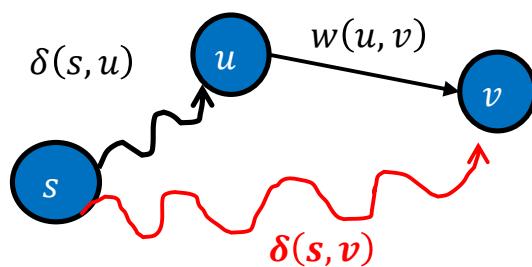
- **Corolario 1:** Si el path  $p = s \rightsquigarrow u \rightarrow v$  es un shortest path, entonces  $\delta(s, v) = \delta(s, u) + w(u, v)$
- **Corolario 2:** Si el path  $p = s \rightsquigarrow u \rightsquigarrow v$  es un shortest path, entonces  $\delta(s, v) = \delta(s, u) + \delta(u, v)$

# Propiedades

- **Triangle Inequality:** Para cualquier arista  $(u, v) \in E$  se cumple que  $\delta(s, v) \leq \delta(s, u) + w(u, v)$

## Demostración:

Por reducción al absurdo, supongamos que  $\delta(s, v) > \delta(s, u) + w(u, v)$ . Entonces si nosotros juntamos el shortest path desde  $s$  hasta  $u$  con la arista  $u \rightarrow v$  y formamos  $p = s \rightsquigarrow u \rightarrow v$ . Entonces estaríamos formando un camino  $p$  que va desde  $s$  a  $v$  y tiene un peso  $w(p) = \delta(s, u) + w(u, v)$ , menor que el del shortest path que va de  $s$  a  $v$ , lo cual es una contradicción. ■



- **Generalización:**  $\delta(s, v) \leq \delta(s, u) + \delta(u, v)$

# Relaxation

En los algoritmos que mostremos para resolver el **SSSP**, la idea es mantener el **estimado de la distancia**  $d(u)$  como una cota superior de la distancia verdadera  $\delta(u)$ . Es decir mantener  $d(u) \geq \delta(u)$  a lo largo de todo el algoritmo, con la esperanza que al finalizar el algoritmo tengamos  $d(u) = \delta(u)$  para todos los nodos.

Por ello empezaremos nuestro estimado de la distancia  $d(u)$  en  $+\infty$  para todos los nodos. A excepción de  $d(s)$  que será 0.

Llamaremos a una arista **tensa**, si cumple que  $d(u) + w(u, v) < d(v)$  . Esto significa que nuestra estimación no es correcta (why? Triangle Inequality) y habrá que corregirla.

Para corregirla utilizaremos un procedimiento llamado **relajación** que será una forma de mejorar el estimado. También podemos guardar un arreglo de **predecesores**  $parent(v)$  que representa el último nodo que relajó a  $v$ . En la imagen de la derecha se presenta la fórmula de relajación.

---

**Function** Relax ( $u, v$ ) :

$d(v) = d(u) + w(u, v)$
$parent(v) = u$

**end**

Note que una arista con  $d(u) = +\infty$  no puede estar tensa.

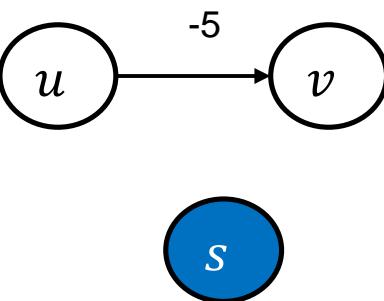
Note que una relajación de una arista tensa  $(u, v)$  decrece el valor de  $d(v)$

# Relaxation

**Notación adicional:** En las siguientes diapositivas, con fines de facilitar la explicación, decir que una arista  $(u, v)$  fue relajada, será equivalente a decir que  $u$  relajó a  $v$  o que  $v$  fue relajado por  $u$ . Es decir, en cierto modo extenderemos la noción de relajación de una arista a **relajación de un nodo**.

**Pequeña advertencia:** Tenga cuidado con los  $\infty$

Suponga que tienes el siguiente grafo:



Inicialmente  $d(s) = 0, d(u) = +\infty, d(v) = +\infty$

Sabemos que  $\delta(v) = +\infty$ , pero ¿La arista  $(u, v)$  está tensa? ( $d(u) + w(u, v) < d(v)$ )?  
No lo está ya que  $\infty - 5 = \infty$ . Pero si pasas el pseudocódigo a código y defines a tu infinito como un número muy grande  $INF$ , tendrás que  $INF - 5 < INF$  y pensarás que sí está tensa cuando no lo está.

Por eso, si es que hay aristas negativas, probablemente necesites agregar una condición más :

*if  $d(u) \neq INF \&& d(u) + w(u, v) < d(v)$*

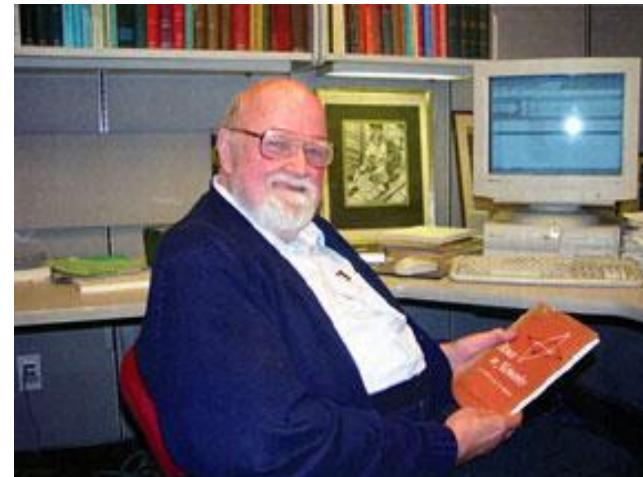
# Generic Shortest Path Algorithm

Lester Ford Jr. publicó en 1956 un algoritmo genérico para resolver el SSSP. Algoritmos similares fueron propuestos de forma independiente por George Dantzig en 1957 y por George Minty en 1958.

El algoritmo relaja aristas tensas mientras todavía existan. El objetivo es que al final del algoritmo  $d(u) = \delta(u)$  y que a través de la secuencia de  $\text{parent}(u)$  se forme un posible shortest path.

En palabras del propio Ford:

A computing procedure is the following. Assign initially  $x_0 = 0$  and  $x_i = \infty$  for  $i \neq 0$ . Scan the network for a pair  $P_i$  and  $P_j$  with the property that  $x_i - x_j > l_{ji}$ . For this pair replace  $x_i$  by  $x_j + l_{ji}$ . Continue this process. Eventually no such pairs can be found, and  $x_N$  is now minimal and represents the minimal distance from  $P_0$  to  $P_N$ . Clearly, if no such pairs can be found, the system (3) is satisfied. We shall now prove optimality.



Lester Ford

# Generic Shortest Path Algorithm

---

**Function** Relax( $u, v$ ) :

$d(v) = d(u) + w(u, v)$

$parent(v) = u$

**end**

---

---

**Function** Initialize( $s$ ) :

**foreach** node  $u$  in  $V_G$  **do**

$d(u) = \infty$

$parent(s) = NULL$

**end**

$d(s) = 0$

**end**

---

---

## Algorithm 1: FordSSSP

---

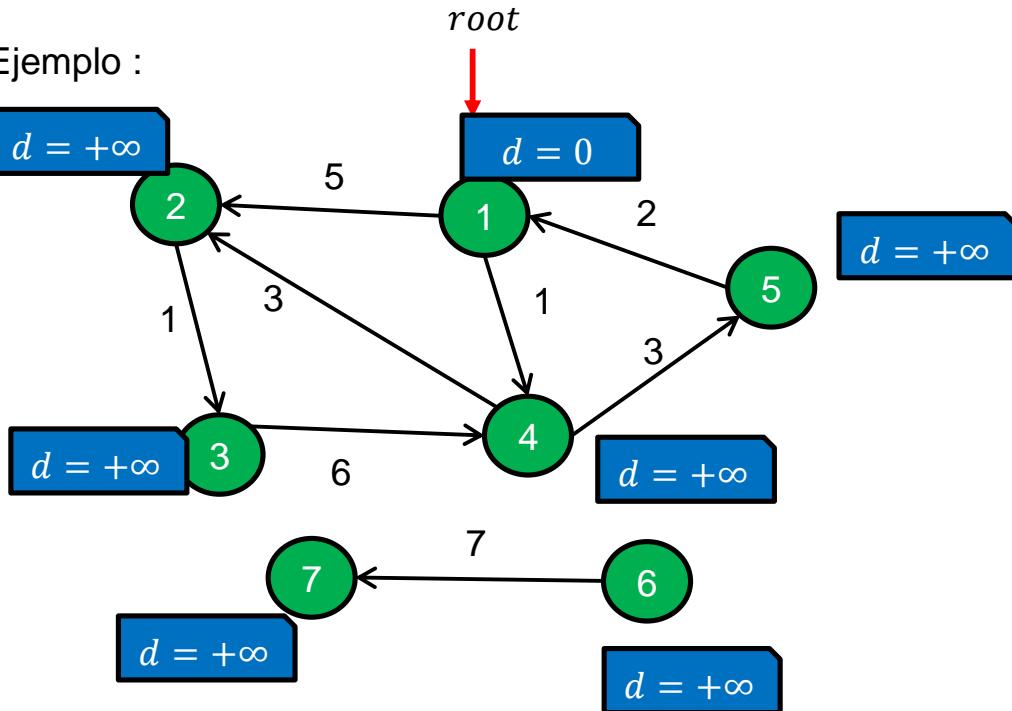
**input** :  $G(V_G, E_G, w)$  ,  $source : s$

**output:**  $d[ ]$ ,  $parent[ ]$

- 1 Initialize ( $s$ )
  - 2 **while** *there exist tense edges* **do**
  - 3     | Pick a tense edge  $(u, v)$  //  $d(u) + w(u, v) < d(v)$
  - 4     | Relax ( $u, v$ )
  - 5 **end**
  - 6 **return**  $d, parent$
-

# Generic Shortest Path Algorithm

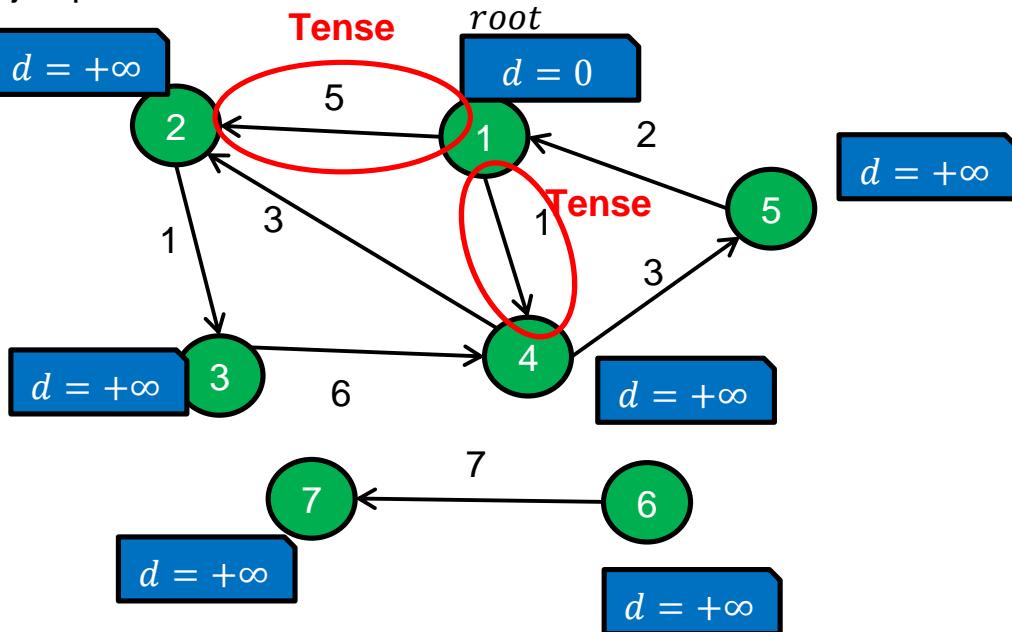
Ejemplo :



$u$	$d(u)$	$parent(u)$
1	0	NULL
2	$+\infty$	NULL
3	$+\infty$	NULL
4	$+\infty$	NULL
5	$+\infty$	NULL
6	$+\infty$	NULL
7	$+\infty$	NULL

# Generic Shortest Path Algorithm

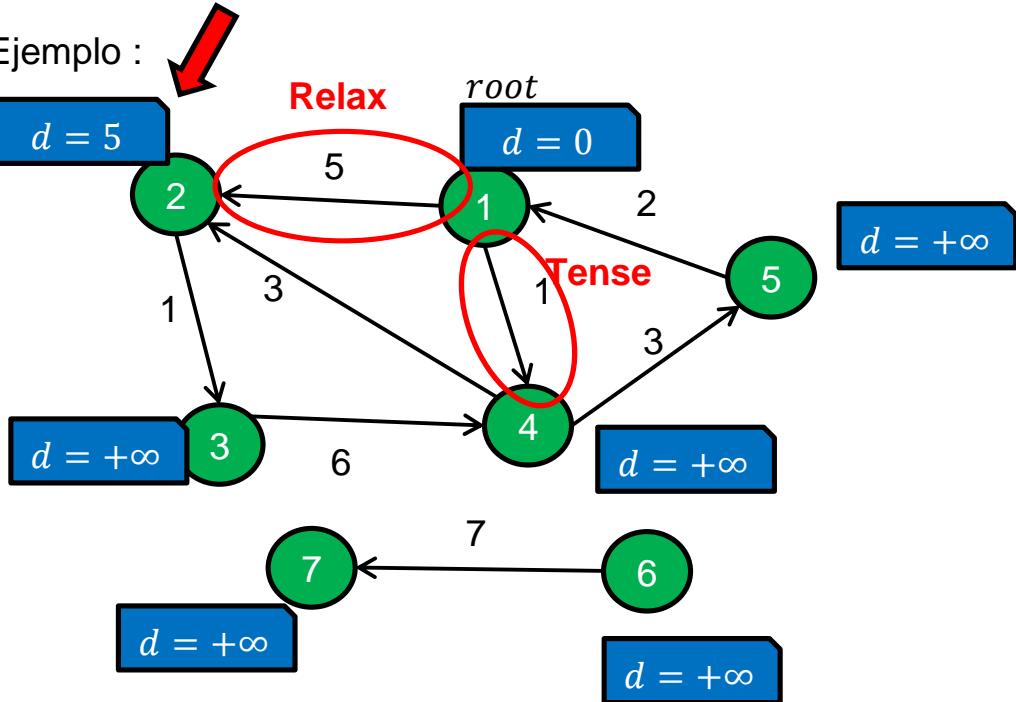
Ejemplo :



$u$	$d(u)$	$\text{parent}(u)$
1	0	NULL
2	$+\infty$	NULL
3	$+\infty$	NULL
4	$+\infty$	NULL
5	$+\infty$	NULL
6	$+\infty$	NULL
7	$+\infty$	NULL

# Generic Shortest Path Algorithm

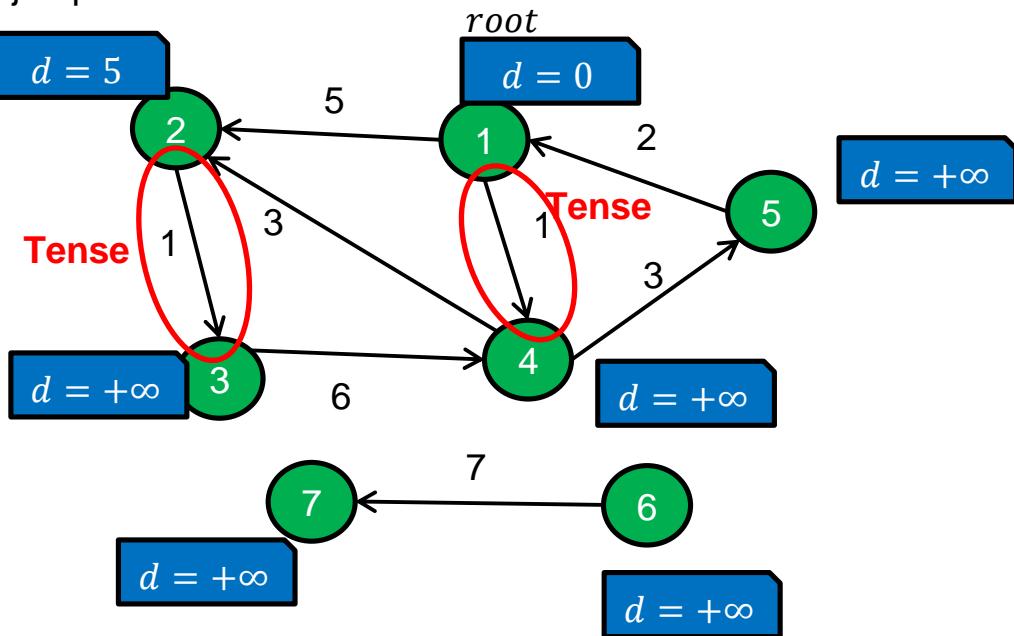
Ejemplo :



$u$	$d(u)$	$\text{parent}(u)$
1	0	NULL
2	5	1
3	$+\infty$	NULL
4	$+\infty$	NULL
5	$+\infty$	NULL
6	$+\infty$	NULL
7	$+\infty$	NULL

# Generic Shortest Path Algorithm

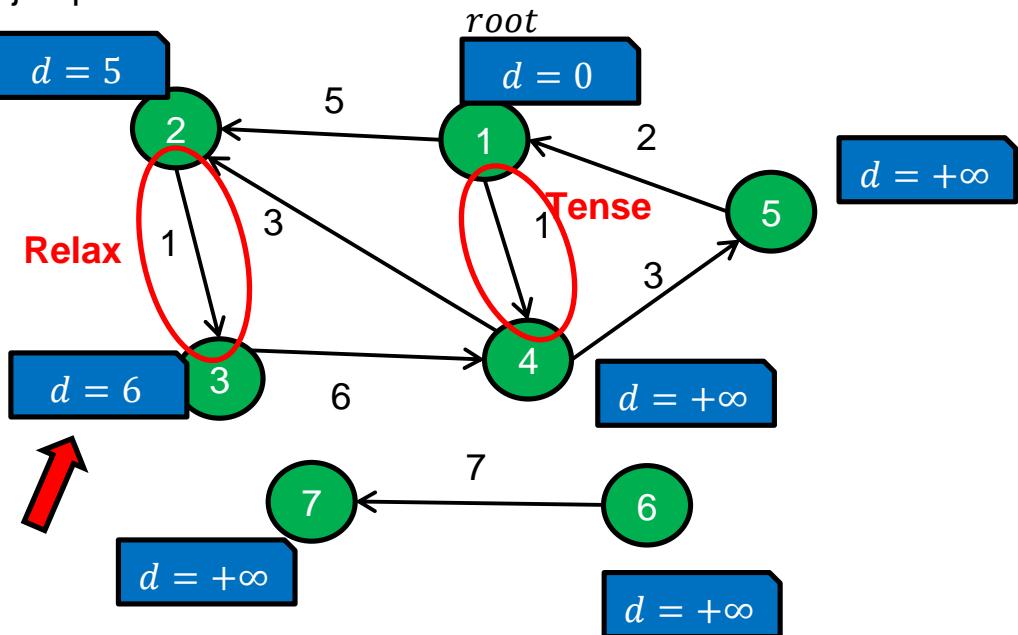
Ejemplo :



$u$	$d(u)$	$\text{parent}(u)$
1	0	NULL
2	5	1
3	$+\infty$	NULL
4	$+\infty$	NULL
5	$+\infty$	NULL
6	$+\infty$	NULL
7	$+\infty$	NULL

# Generic Shortest Path Algorithm

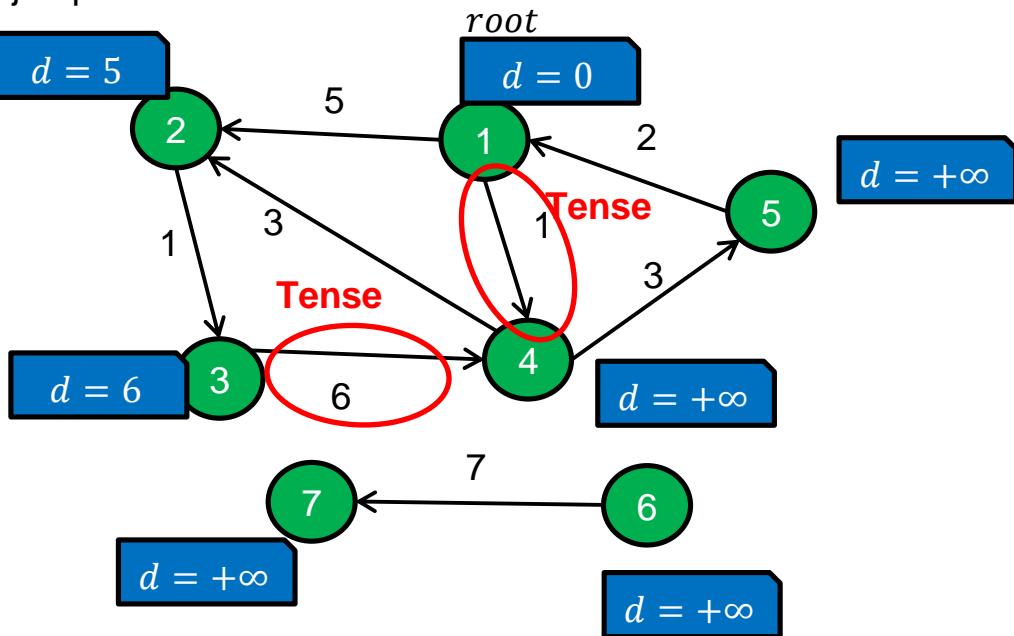
Ejemplo :



$u$	$d(u)$	$parent(u)$
1	0	NULL
2	5	1
3	6	2
4	$+\infty$	NULL
5	$+\infty$	NULL
6	$+\infty$	NULL
7	$+\infty$	NULL

# Generic Shortest Path Algorithm

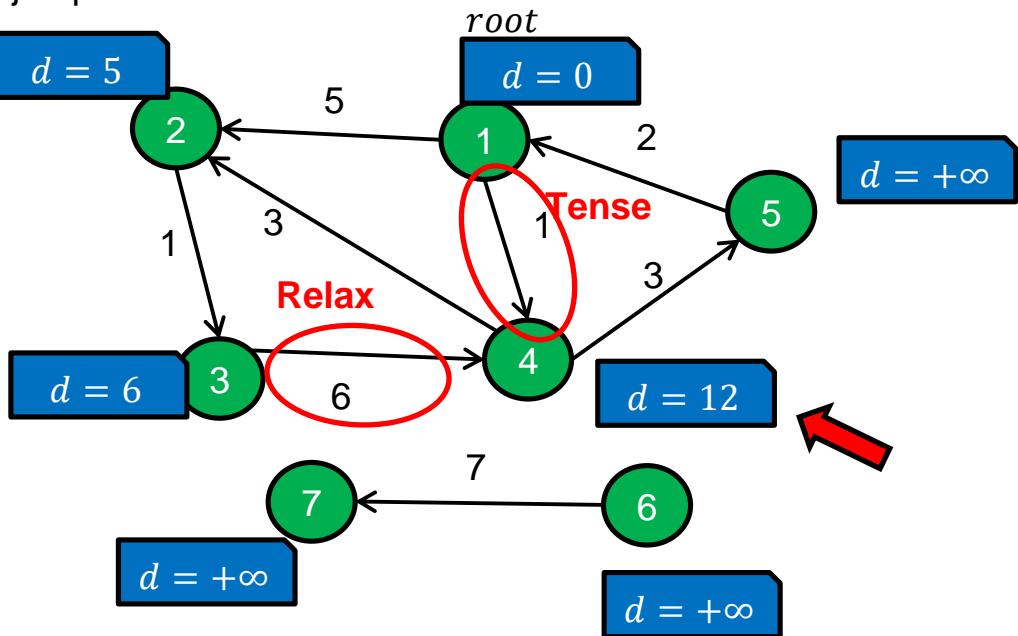
Ejemplo :



$u$	$d(u)$	$\text{parent}(u)$
1	0	NULL
2	5	1
3	6	2
4	$+\infty$	NULL
5	$+\infty$	NULL
6	$+\infty$	NULL
7	$+\infty$	NULL

# Generic Shortest Path Algorithm

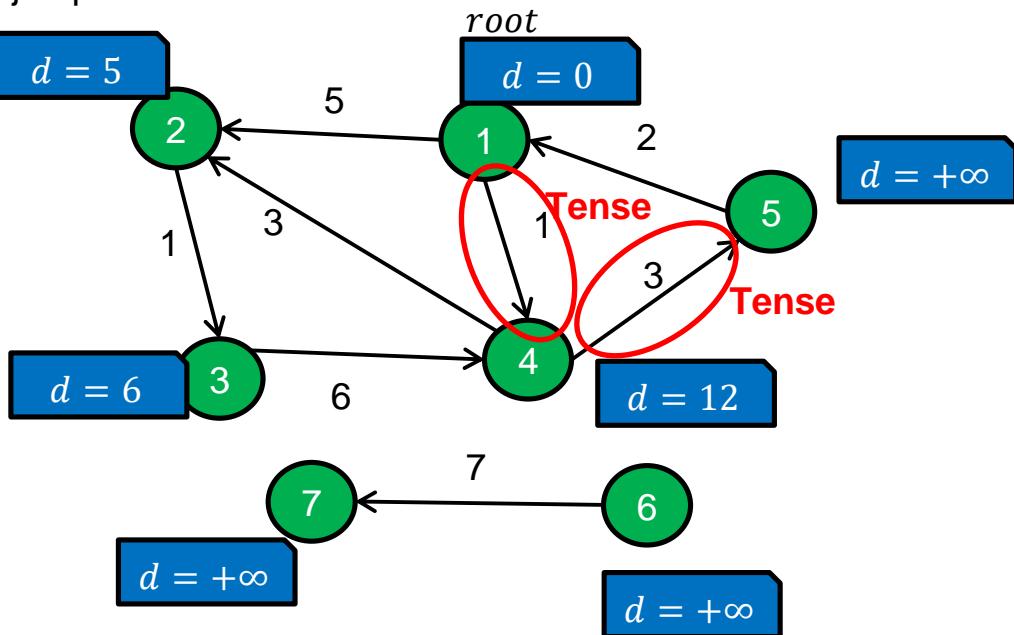
Ejemplo :



$u$	$d(u)$	$\text{parent}(u)$
1	0	NULL
2	5	1
3	6	2
4	12	3
5	$+\infty$	NULL
6	$+\infty$	NULL
7	$+\infty$	NULL

# Generic Shortest Path Algorithm

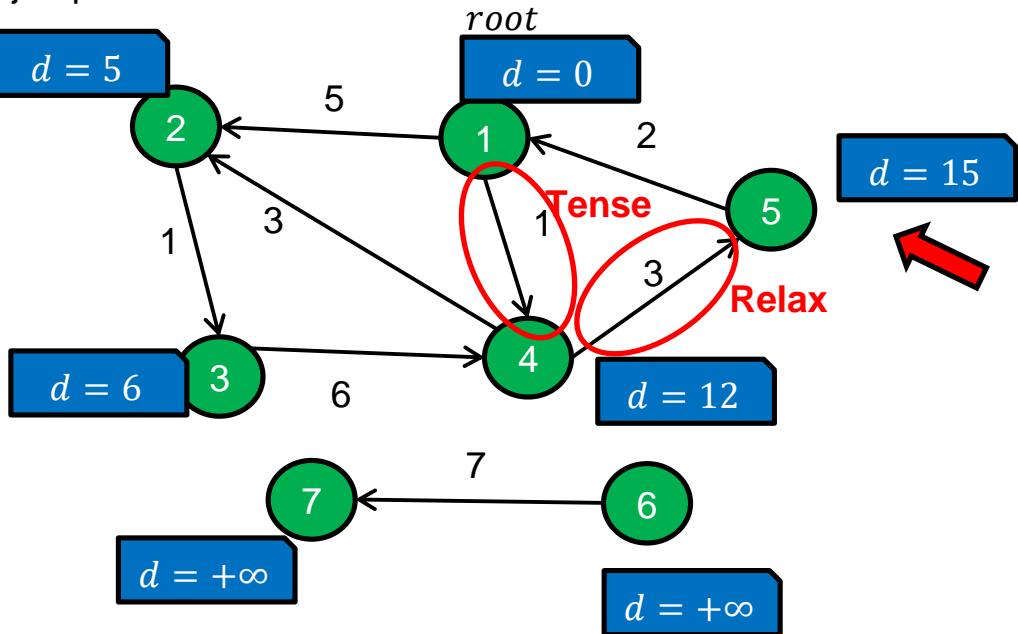
Ejemplo :



$u$	$d(u)$	$parent(u)$
1	0	NULL
2	5	1
3	6	2
4	12	3
5	$+\infty$	NULL
6	$+\infty$	NULL
7	$+\infty$	NULL

# Generic Shortest Path Algorithm

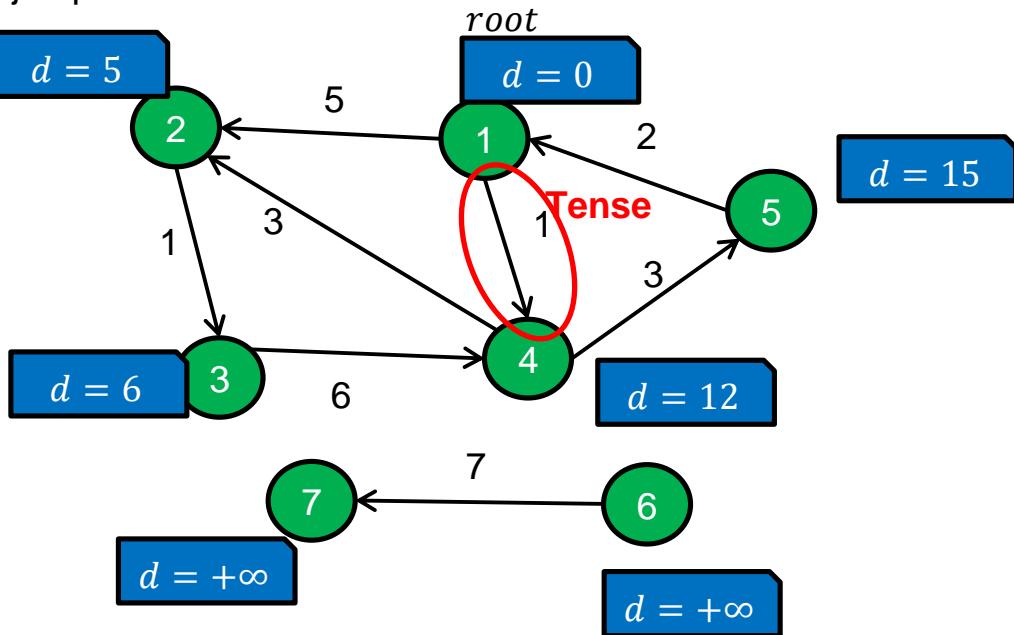
Ejemplo :



$u$	$d(u)$	$\text{parent}(u)$
1	0	NULL
2	5	1
3	6	2
4	12	3
5	15	4
6	$+\infty$	NULL
7	$+\infty$	NULL

# Generic Shortest Path Algorithm

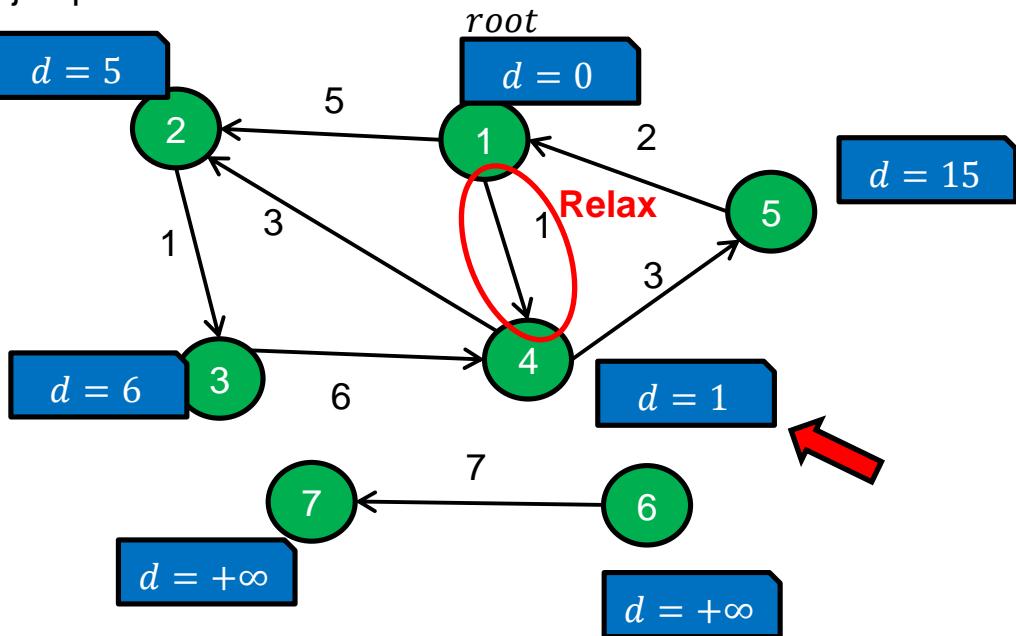
Ejemplo :



$u$	$d(u)$	$parent(u)$
1	0	NULL
2	5	1
3	6	2
4	12	3
5	15	4
6	$+\infty$	NULL
7	$+\infty$	NULL

# Generic Shortest Path Algorithm

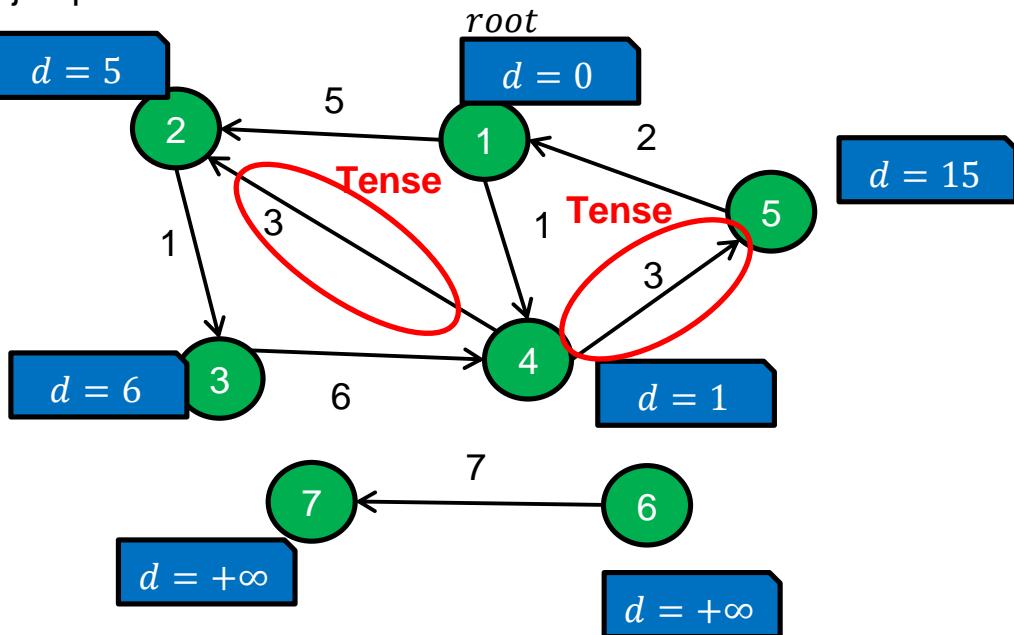
Ejemplo :



$u$	$d(u)$	$\text{parent}(u)$
1	0	NULL
2	5	1
3	6	2
4	1	1
5	15	4
6	$+\infty$	NULL
7	$+\infty$	NULL

# Generic Shortest Path Algorithm

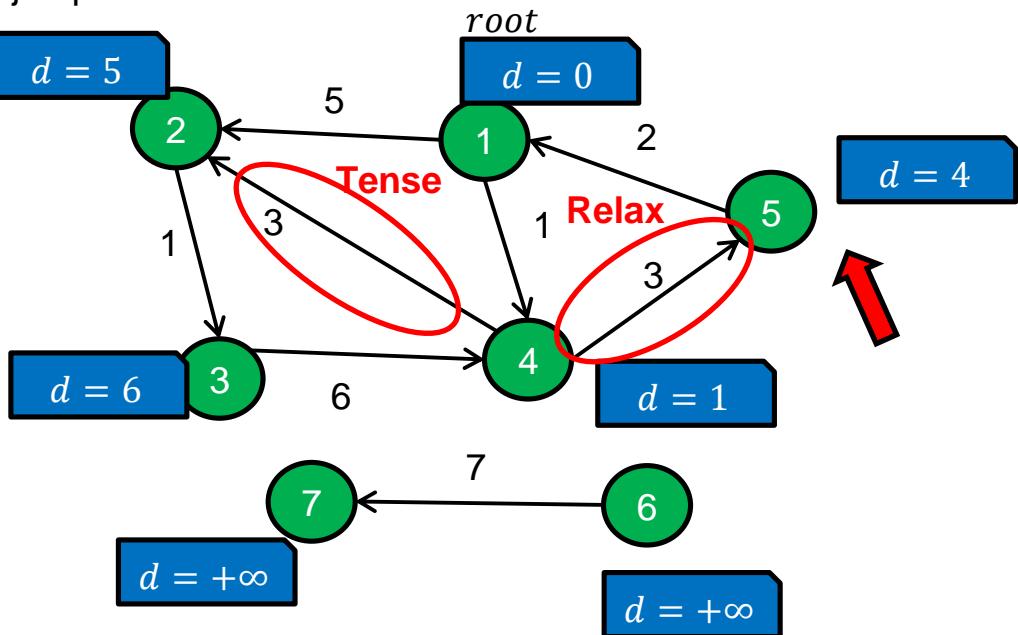
Ejemplo :



$u$	$d(u)$	$parent(u)$
1	0	NULL
2	5	1
3	6	2
4	1	1
5	15	4
6	$+\infty$	NULL
7	$+\infty$	NULL

# Generic Shortest Path Algorithm

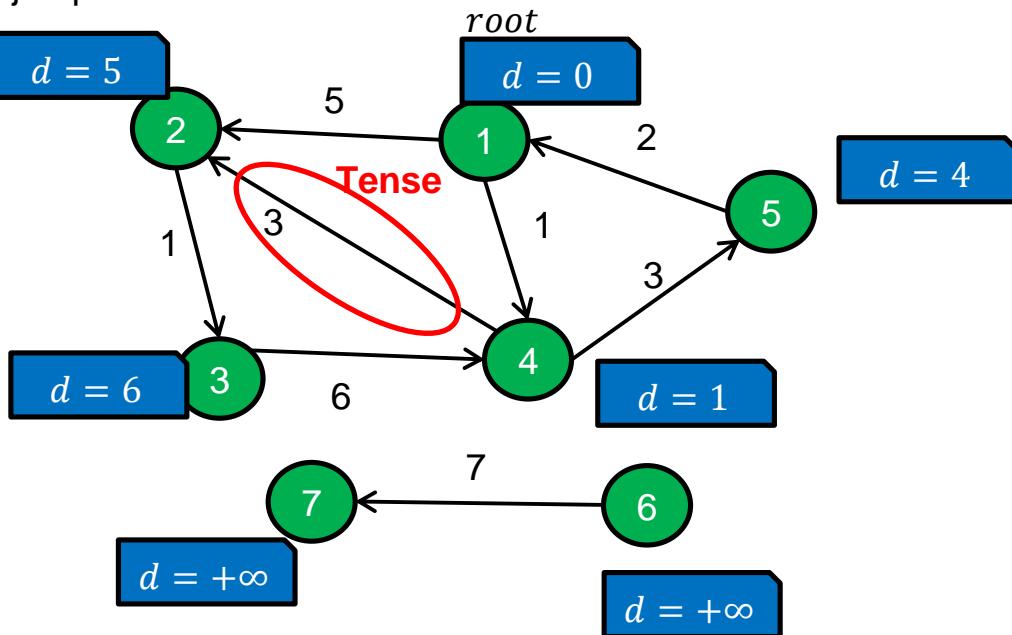
Ejemplo :



$u$	$d(u)$	$\text{parent}(u)$
1	0	NULL
2	5	1
3	6	2
4	1	1
5	4	4
6	$+\infty$	NULL
7	$+\infty$	NULL

# Generic Shortest Path Algorithm

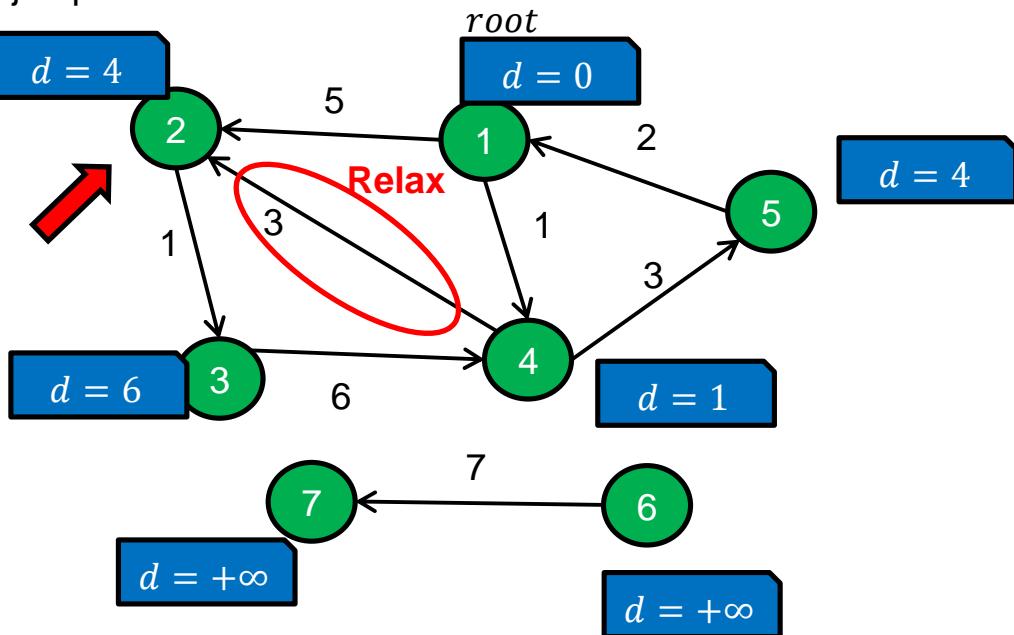
Ejemplo :



<i>u</i>	<i>d(u)</i>	<i>parent(u)</i>
1	0	NULL
2	5	1
3	6	2
4	1	1
5	4	4
6	$+\infty$	NULL
7	$+\infty$	NULL

# Generic Shortest Path Algorithm

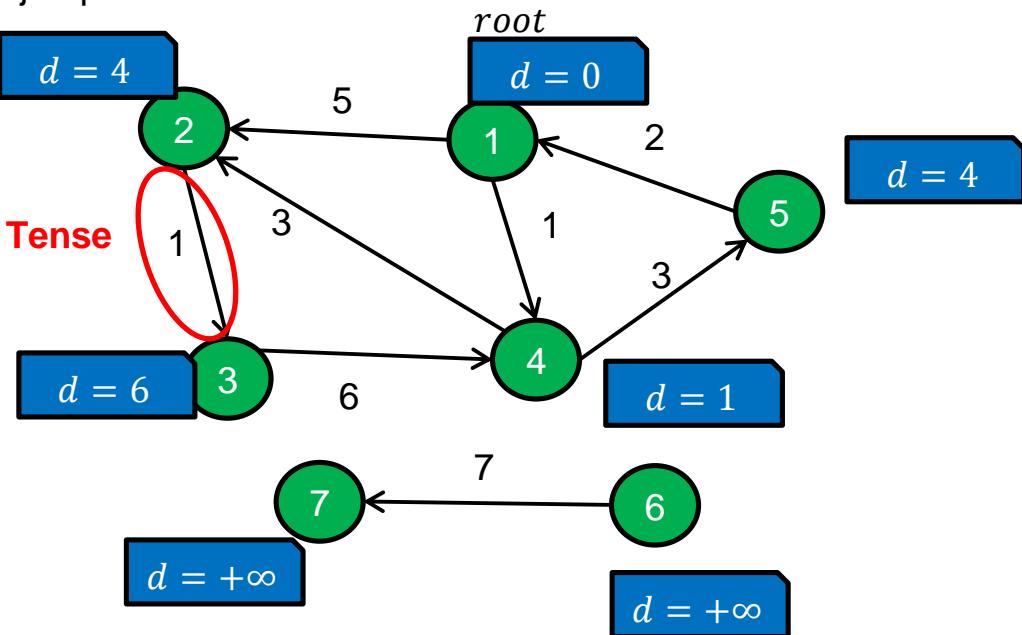
Ejemplo :



$u$	$d(u)$	$parent(u)$
1	0	NULL
2	4	4
3	6	2
4	1	1
5	4	4
6	$+\infty$	NULL
7	$+\infty$	NULL

# Generic Shortest Path Algorithm

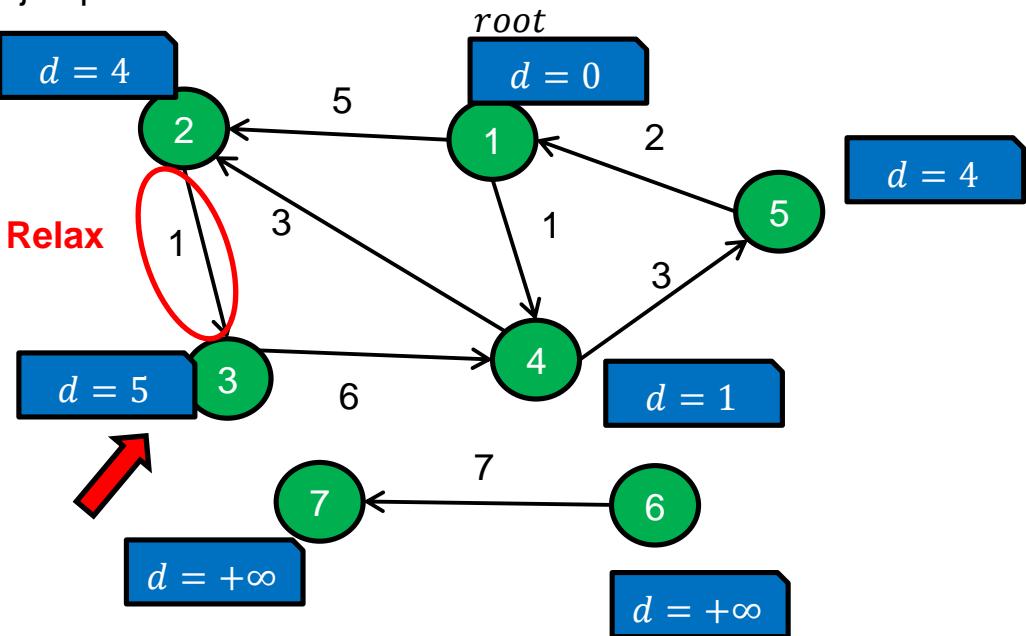
Ejemplo :



<i>u</i>	$d(u)$	$\text{parent}(u)$
1	0	NULL
2	4	4
3	6	2
4	1	1
5	4	4
6	$+\infty$	NULL
7	$+\infty$	NULL

# Generic Shortest Path Algorithm

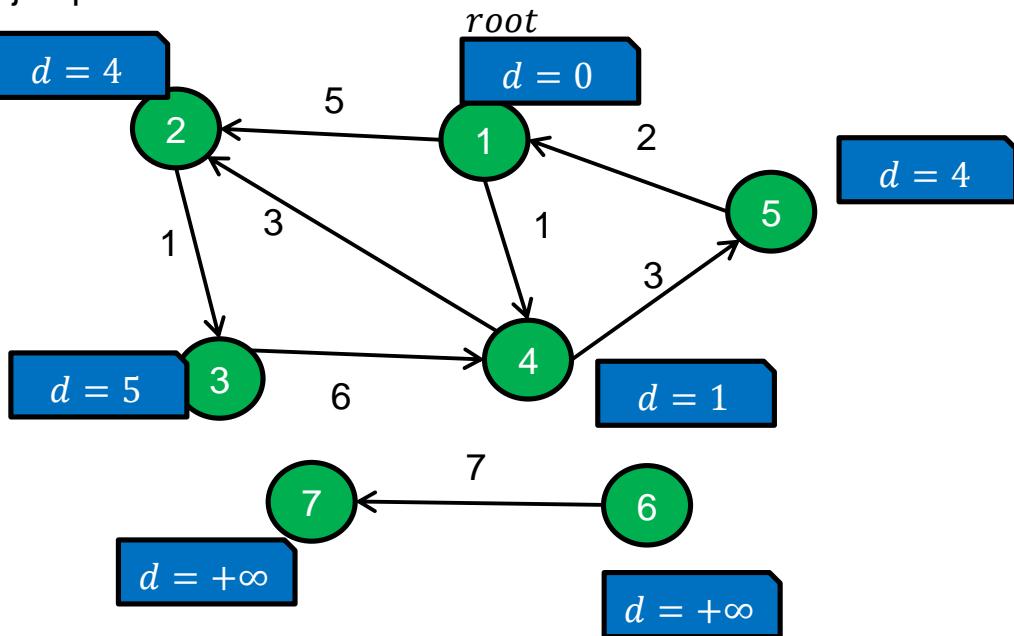
Ejemplo :



<i>u</i>	<i>d(u)</i>	<i>parent(u)</i>
1	0	NULL
2	4	4
3	5	2
4	1	1
5	4	4
6	$+\infty$	NULL
7	$+\infty$	NULL

# Generic Shortest Path Algorithm

Ejemplo :



$u$	$d(u)$	$\text{parent}(u)$
1	0	NULL
2	4	4
3	5	2
4	1	1
5	4	4
6	$+\infty$	NULL
7	$+\infty$	NULL

# Generic Shortest Path Algorithm

En las siguientes diapositivas veremos unas propiedades adicionales que nos ayudarán a demostrar por qué este algoritmo general funciona.

Sin embargo, hay que resaltar que este es un algoritmo general en donde **no** se menciona **cómo** escoger las aristas.

La mayoría de algoritmos de shortest path son un caso particular del algoritmo en donde cambia el criterio de **selección** de las aristas tensas. La complejidad del algoritmo dependerá del **criterio de selección**.

También es posible escoger un criterio de selección “pobre” que pueda llevar a tiempos de ejecución **exponentiales**.



# Propiedades de la relajación

- **Upper bound property:** En cada iteración del algoritmo (luego de la inicialización), siempre se cumple que  $d(u) \geq \delta(u)$  para cualquier nodo  $u \in V$ . Y cuando  $d(u)$  llega a ser  $\delta(u)$  se mantiene en ese valor para siempre.

**Demostración:**

Demostremos que  $d(u) \geq \delta(u)$  por inducción en el número de relajaciones

- **Caso base:** Al inicio  $d(s) = 0 \geq \delta(s)$  ya que  $\delta(s)$  es 0 o  $-\infty$  en caso hayan ciclos negativos que pasen por  $s$ . Los demás vértices tendrán  $d(u) = +\infty$  por lo que evidentemente  $d(u) \geq \delta(u)$ .
- **Paso inductivo:**

Supongamos que para  $i$  relajaciones ejecutadas se cumple la propiedad. Sea  $(u, v)$  la siguiente arista a relajar.

La relajación hará  $d(v) = d(u) + w(u, v) \dots (1)$ , por lo que el único valor que cambiará es  $d(v)$ .

$$\Rightarrow d(u) \geq \delta(u) \quad (\text{por hipótesis inductiva ya que } d(u) \text{ no cambió})$$

$$\Rightarrow d(v) \geq \delta(u) + w(u, v) \quad (\text{sumando } w(u, v) \text{ a ambos lados y usando (1)})$$

$$\Rightarrow d(v) \geq \delta(u) + w(u, v) \geq \delta(v) \quad (\text{por Triangle Inequality})$$

Lo cual demuestra que  $d(v) \geq \delta(v)$ .

Además como las relajaciones solo decrecen el estimado de  $d$ , tenemos que si en algún momento  $d(u) = \delta(u)$ , entonces este nodo mantendrá ese valor hasta el final del algoritmo ■

- **(Corolario) No path property:** Si no existe camino desde  $s$  hasta  $u$  entonces  $d(u) = \delta(u) = +\infty$  durante todo el algoritmo (luego de la inicialización).

# Propiedades de la relajación

- **Convergence property:** Si el path  $p = s \rightsquigarrow u \rightarrow v$  es un shortest path y, en algún momento del algoritmo se tiene que  $d(u) = \delta(u)$ , entonces al relajar la arista  $u \rightarrow v$  se tendrá que  $d(v) = \delta(v)$ .

## Demostración

Por definición de peso de un path tenemos que  $w(p) = w(s \rightsquigarrow u) + w(u, v)$ .

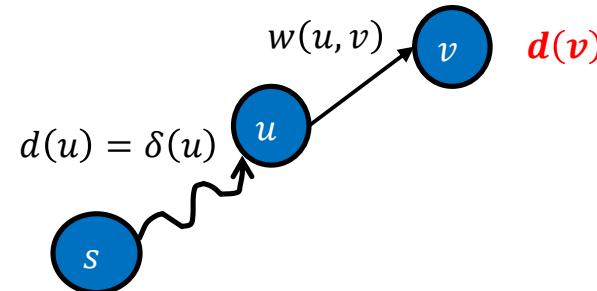
Por ser  $p$  un shortest path, entonces el subpath  $s \rightsquigarrow u$  es shortest path también (**optimal substructure**) por lo que  $\delta(v) = \delta(u) + w(u, v)$  ... (1)

Al relajar la arista tendrás:

$$d(v) = d(u) + w(u, v)$$

$$\Rightarrow d(v) = \delta(u) + w(u, v) \text{ (por hipótesis)}$$

$$\Rightarrow d(v) = \delta(v) \quad (\text{Aplicando (1)}) \blacksquare$$



# Propiedades de la relajación

- **Path relaxation property:** Sea un grafo sin ciclos negativos que y sea  $p = v_0 \rightarrow v_1, \rightarrow \dots \rightarrow v_k$  un shortest path donde  $v_0 = s$ . Si nosotros relajamos las aristas  $(v_0, v_1), (v_1, v_2), \dots, (v_{k-1}, v_k)$  en ese mismo orden (con posibles relajaciones intermedias) entonces tendremos  $d(v_k) = \delta(v_k)$  luego de la última relajación.

## Demostración:

Demostraremos por inducción en el número de vértices del path.

- **Caso base:** Para  $i = 0$  tenemos que  $d(v_0 = s) = 0 = \delta(s)$
- **Paso inductivo:** Por hipótesis inductiva, luego de la  $i - \text{ésima}$  relajación de la lista, tendremos  $d(v_i) = \delta(v_i)$ . En la  $(i + 1) - \text{ésima}$  relajación se relajará la arista  $(v_i, v_{i+1})$ . Por la propiedad que **todo subpath de un shortest path es también shortest path** sabemos que  $v_0, v_1, \dots, v_{i+1}$  también es shortest path. Luego, si aplicamos la **convergence property** tendremos que  $d(v_{i+1}) = \delta(v_{i+1})$ . Por lo que se cumple la propiedad. ■

... (otros relaxes)
$relax(v_0, v_1)$
... (otros relaxes)
$relax(v_1, v_2)$
... (otros relaxes)
... ( <i>relax del path – otros relaxes</i> )
$relax(v_{k-1}, v_k)$

# Propiedades de la relajación

Nota: El nombre de la siguiente propiedad me lo he inventado.

- **Tense-Relaxation property:** Cada vez que una arista  $(u, v) \in E$  relajada. Las nuevas aristas tensas surgidas a partir de la relajación solo pueden ser *outgoing edges* de  $v$ , es decir, del tipo  $(v, out) \in E$

## Demostración:

Suponga que al relajar la arista  $(u, v)$  se crean nuevas aristas tensas. La relajación hará  $d(v) = d(u) + w(u, v)$ . Por lo que el único valor  $d$  que ha cambiado es  $d(v)$ . Por lo que las únicas aristas tensas creadas tienen que ser aquellas que tengan a  $v$  como *endpoint*. Es decir, del tipo  $(in, v) \in E$  o del tipo  $(v, out) \in E$ .

Por contradicción suponga que surgió alguna arista tensa  $in \rightarrow v$ , esto quiere decir que inicialmente no estuvo tensa. Denotemos como  $d'(v)$  el valor del estimado de la distancia justo antes de la relajación y a  $d(v)$  como el valor inmediatamente después. Entonces debió cumplirse que  $d(in) + w(in, v) \geq d'(v)$ . Pero también tenemos que  $d'(v) > d(v)$  ya que las relajaciones solo pueden disminuir el estimado de la distancia

Por lo tanto  $d(in) + w(in, v) > d(v)$  por lo que la arista  $in \rightarrow v$  no está tensa, llegando a una contradicción ■



# Resumen de propiedades vistas

Optimal substructure	Los subpaths de un shortest path son también shortest path. <b>Corolario :</b> Si el path $p = s \rightsquigarrow u \rightarrow v$ es un shortest path, entonces $\delta(s, v) = \delta(s, u) + w(u, v)$
Triangle Inequality	Para cualquier arista $(u, v) \in E$ se cumple que $\delta(s, v) \leq \delta(s, u) + w(u, v)$
Upper bound property	En cada iteración del algoritmo, siempre se cumple que $d(u) \geq \delta(u)$ para cualquier nodo $u$ . Y cuando $d(u)$ llega a ser $\delta(u)$ se mantiene en ese valor para siempre.
No path property	Si no existe camino desde $s$ hasta $u$ entonces $d(u) = \delta(u) = +\infty$ durante todo el algoritmo
Convergence property	Si el path $p = s \rightsquigarrow u \rightarrow v$ es un shortest path y, en algún momento del algoritmo se tiene que $d(u) = \delta(u)$ , entonces al relajar la arista $u \rightarrow v$ se tendrá que $d(v) = \delta(v)$ .
Path relaxation property	Sea un grafo sin ciclos negativos que y sea $p = v_0 \rightarrow v_1 \rightarrow \dots \rightarrow v_k$ un shortest path donde $v_0 = s$ . Si nosotros relajamos las aristas $(v_0, v_1), (v_1, v_2), \dots, (v_{k-1}, v_k)$ en ese mismo orden (con posibles relajaciones intermedias) entonces tendremos $d(v_k) = \delta(v_k)$ luego de la última relajación.
Tense Relaxation property	Cada vez que una arista $(u, v) \in E$ relajada. Las nuevas aristas tensas surgidas a partir de la relajación solo pueden ser <i>outgoing edges</i> de $v$ , es decir, del tipo $(v, out) \in E$

# Generic Shortest Path Algorithm - Correctness

- **Lema 2.1:** Durante la ejecución del Algoritmo Genérico de Shortest Path,  $\forall u \in V / d(u) \neq +\infty$ , se cumple que  $d(u)$  es el peso de algún *walk* que va desde  $s$  a  $u$ .

## Demostración:

Por inducción en el número de relajaciones.

- **Caso base:**

Luego de la inicialización, después de 0 relajaciones, tenemos que  $d(s) = 0$  es el peso del walk que contiene solo al vértice  $s$  y todos los demás nodos  $u \in V$  tienen  $d(u) = +\infty$

- **Paso inductivo:**

Suponga que el lema se cumple luego de  $i$  relajaciones. Sea  $(u, v)$  la arista a relajar en la  $(i + 1)$  – éSIMA relajación. Al hacer la relajación se hará  $d(v) = d(u) + w(u, v)$ . Por lo que el único valor de los estimados de las distancias( $d$ ) es  $d(v)$  por lo que el lema seguirá cumpliendo para los demás nodos. Demostremos que también cumple para  $v$ .

Por hipótesis inductiva  $d(u)$  es el peso de algún *walk*. Sea  $p$  un *walk* cuya longitud es  $d(u)$ . Entonces podemos formar el *walk*  $p' = p + (u, v)$  tal que  $w(p') = w(p) + w(u, v) = d(u) + w(u, v) = d(v)$ . Por lo que  $d(v)$  es el peso de  $p'$  y luego de la  $(i + 1)$  – éSIMA relajación, el lema se sigue cumpliendo. ■

# Generic Shortest Path Algorithm - Correctness

- **Lema 2.2:** Si no hay ciclos negativos, durante la ejecución del Algoritmo Genérico de Shortest Path,  $\forall u \in V / d(u) \neq +\infty$ , se cumple que  $d(u)$  es el peso de algún *path* que va desde  $s$  a  $u$ . Es más, existe algún *path*  $p = a_0, a_1, \dots, a_k$  con  $a_0 = s$ ,  $a_k = u$  tal que  $w(p) = d(u)$  y  $w(p_i) \geq d(a_i)$  para todo  $0 \leq i \leq k$ , donde  $w(p_i)$  es el peso del subpath de  $p$  desde  $s$  hasta  $a_i$

## Demostración:

Por inducción en el número de relajaciones.

- **Caso base:**

Luego de la inicialización, después de 0 relajaciones, tenemos que  $d(s) = 0$  es el peso del path  $p$  que contiene solo al vértice  $s$  y  $w(p_0) = d(s) = 0$ . Además todos los demás nodos  $u \in V$  tienen  $d(u) = +\infty$ .

- **Paso inductivo:**

Suponga que el lema se cumple luego de  $i$  relajaciones. Sea  $(u, v)$  la arista a relajar en la  $(i+1)$ -ésima relajación. Al hacer la relajación se hará  $d(v) = d(u) + w(u, v)$ . Por lo que el único valor de los estimados de las distancias ( $d$ ) que cambia es  $d(v)$ . Los demás valores  $d(x)$ ,  $x \neq v$  seguirán cumpliendo que son el peso de algún path. Además suponga que existía un path  $p$  que cumplía la 2da parte del lema y que ese path incluía a  $v$  en la posición  $x$ . A pesar de que  $d(v)$  ha cambiado, solo pudo haber disminuido por lo que  $w(p_x) \geq d(a_x = v)$  se seguirá cumpliendo luego de la relajación.

Lo único que falta demostrar es que  $d(v)$  al ser relajado también es el peso de algún path (y no de un walk con ciclo).

# Generic Shortest Path Algorithm - Correctness

- **Lema 2.2:** Si no hay ciclos negativos, durante la ejecución del Algoritmo Genérico de Shortest Path,  $\forall u \in V / d(u) \neq +\infty$ , se cumple que  $d(u)$  es el peso de algún *path* que va desde  $s$  a  $u$ . Es más, existe algún *path*  $p = a_0, a_1, \dots, a_k$  con  $a_0 = s$ ,  $a_k = u$  tal que  $w(p) = d(u)$  y  $w(p_i) \geq d(a_i)$  para todo  $0 \leq i \leq k$ , donde  $w(p_i)$  es el peso del subpath de  $p$  desde  $s$  hasta  $a_i$

**Demostración (continuación):**

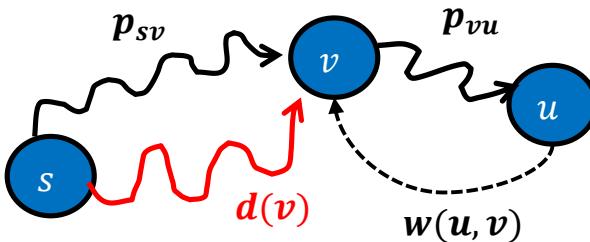
- **Paso inductivo:**

Lo único que falta demostrar es que  $d(v)$  al ser relajado también es el peso de algún path. Por hipótesis inductiva sabemos que, antes de la relajación,  $d(v)$  es el peso de algún path  $p = a_0, a_1, \dots, a_k$  con  $a_0 = s$ ,  $a_k = u$  tal que  $w(p) = d(u)$  y  $w(p_i) \geq d(a_i)$  para todo  $0 \leq i \leq k$ .

Por contradicción suponga que  $p + (u, v)$  formara un ciclo. Dividamos  $p$  en  $p_{sv}$  y  $p_{vu}$  denotando los subpaths entre  $s, v$  y entre  $v, u$  respectivamente.

$$\text{Entonces } d(u) = w(p_{sv}) + w(p_{vu}) \dots (1)$$

Por hipótesis inductiva  $d(v)$  antes de relajación también es el peso de algún path (aunque no necesariamente es  $p_{sv}$ ) y además  $w(p_{sv}) \geq d(v) \dots (2)$



Antes de la relajación de  $(u, v)$  esta arista tuvo que estar tensa, por lo que  $d(u) + w(u, v) < d(v)$

$$\Rightarrow \cancel{w(p_{sv})} + w(p_{vu}) + w(u, v) < w(p_{sv}) \text{ (aplicando (1) y (2))}.$$

$\Rightarrow w(p_{uv}) + w(u, v) < 0$ . Lo cual significa que hay un ciclo negativo ( $p_{vu} + (u, v)$ ), que es una contradicción.

Esto prueba que  $p + (u, v)$  es un path válido con peso  $d(v)$  y se cumple el lema ■

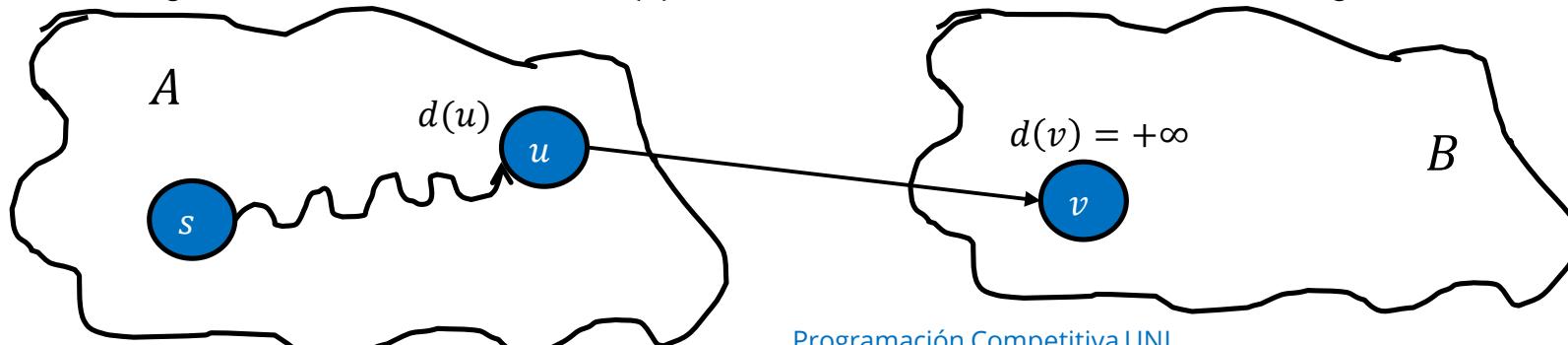
# Generic Shortest Path Algorithm - Correctness

- **Lema 2.3:** Si el Algoritmo genérico del Shortest Path termina, entonces todos los nodos  $u \in V$  que son alcanzables desde  $s$  tienen  $d(u) \neq +\infty$  al finalizar el algoritmo.

## Demostración:

Si el algoritmo termina quiere decir que ya no hay más aristas tensas. Por reducción al absurdo suponga que existe algún  $v$  alcanzable por  $s$  que tenga  $d(v) = +\infty$ . Podemos definir el grupo  $A$  donde  $A$  contiene a todos los nodos alcanzables por  $s$  con  $d(u) < +\infty$  y definimos el grupo  $B$  con los nodos  $u$  alcanzables por  $s$  con  $d(u) = +\infty$ . Entonces debe existir alguna arista que une algún vértice  $u \in A$  con otro vértice  $v \in B$  (La arista debe existir debido a que  $A \neq \emptyset$  porque contiene al menos a  $s$ ; y porque si no existiera esa arista, entonces  $v$  no sería alcanzable por  $s$ )

Sin embargo  $d(u) + w(u, v) < +\infty = d(v)$ . Entonces la arista  $u \rightarrow v$  está tensa, llegando a una contradicción. ■



# Generic Shortest Path Algorithm - Correctness

- **Lema 2.4:** El Algoritmo Genérico del Shortest Path termina si y solo si no existen ciclos negativos alcanzables desde  $s$ .

**Demostración:**

- a) **Si no existen ciclos negativos alcanzables desde  $s \Rightarrow$  el algoritmo termina**

Como no existen ciclos negativos alcanzables, entonces  $\delta(v) \in \mathbb{R}$  o  $\delta(v) = +\infty$ .

En caso  $\delta(v) = +\infty$ , por **no path property** el valor de  $d(v)$  nunca cambia.

En caso  $\delta(v) \in \mathbb{R}$  (existe shortest path) considere que en cada iteración el algoritmo elige una arista tensa y la relaja, decreciendo el valor de  $d(v)$ . Por el lema 2.2 sabemos que  $d(v)$  siempre será siempre el peso de un path. Además por **upper bound property** tenemos que  $d(v) \geq \delta(v)$ .

Es decir en cada relajación a  $v$ , el valor  $d(v)$  se vuelve el peso de un path con menor peso que antes (es decir, de un path distinto). Esto puede seguir hasta que llega a ser el peso de un shortest path. Como hay un número finito de paths (aproximadamente  $(V - 2)! \times e$  en un grafo completo) entonces solo puedo relajar un número finito de veces cada nodo.

# Generic Shortest Path Algorithm - Correctness

- **Lema 2.4:** El Algoritmo Genérico del Shortest Path termina si y solo si no existen ciclos negativos alcanzables desde  $s$ .

**Demostración:**

- b) **Si existen ciclos negativos alcanzables desde  $s \Rightarrow$  el algoritmo no termina**

Por reducción al absurdo, supongamos que el algoritmo termine. Por el lema 2.3, todos los nodos  $u$  alcanzables por  $s$  cumplen que  $d(u) \neq +\infty$ . Además  $d(u) \neq -\infty$  porque solo pudo haber decrecido un número finito de veces.

Suponga que  $c = v_1 \rightarrow v_2 \rightarrow \dots \rightarrow v_k \rightarrow v_1$  es un ciclo negativo alcanzable.

Entonces  $w(c) = w(v_1, v_2) + w(v_2, v_3) + \dots + w(v_k, v_1) < 0$

Como ya no hay aristas tensas, se debe cumplir que :

$$d(v_1) + w(v_1, v_2) \geq d(v_2) \Rightarrow d(v_1) + w(v_1, v_2) - d(v_2) \geq 0$$

$$d(v_2) + w(v_2, v_3) \geq d(v_3) \Rightarrow d(v_2) + w(v_2, v_3) - d(v_3) \geq 0$$

...

$$d(v_k) + w(v_k, v_1) \geq d(v_1) \Rightarrow d(v_k) + w(v_k, v_1) - d(v_1) \geq 0$$

Si sumamos todo, tendremos una suma telescópica donde todos los  $d(v_i)$  se eliminan.

Nos queda  $w(c) = w(v_1, v_2) + w(v_2, v_3) + \dots + w(v_k, v_1) \geq 0$  lo cual es una contradicción. ■

# Generic Shortest Path Algorithm - Correctness

□ **Lema 2.5:** Si el Algoritmo Genérico de Shortest Path termina, entonces  $d(u) = \delta(u)$  para todo nodo  $u \in V$

**Demostración:**

Si  $\delta(u) = +\infty$ , por **no path property**  $d(u)$  mantendrá ese valor hasta el final. Como el algoritmo termina entonces no hay ciclos negativos por el lema 2.4. El último caso que queda es cuando  $\delta(u) \in \mathbb{R}$  (el shortest path existe)

Sea  $p = v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow \dots \rightarrow v_k$  un shortest path que va desde  $s$  hasta  $u$  ( $v_0 = s, v_k = u$ )

Como el algoritmo terminó, no existen aristas tensas por lo que debe cumplirse que

$d(v_i) + w(v_i, v_{i+1}) \geq d(v_{i+1})$  para todo  $0 \leq i < k$  es decir:

$$d(v_0) + w(v_0, v_1) \geq d(v_1) \Rightarrow w(v_0, v_1) \geq d(v_1) - d(v_0)$$

$$d(v_1) + w(v_1, v_2) \geq d(v_2) \Rightarrow w(v_1, v_2) \geq d(v_2) - d(v_1)$$

...

$$d(v_{k-1}) + w(v_{k-1}, v_k) \geq d(v_k) \Rightarrow w(v_{k-1}, v_k) \geq d(v_k) - d(v_{k-1})$$

Sumando todo tenemos que  $w(p) = \sum_{i=0}^{k-1} w(v_i, v_{i+1}) \geq d(v_k) - d(v_0) \Rightarrow w(p) \geq d(u) \Rightarrow \delta(u) \geq d(u) \dots (1)$

Por **upper bound property**,  $d(u) \geq \delta(u) \dots (2)$

Las desigualdades de (1) y (2) implican que  $d(u) = \delta(u)$  ■

# Generic Shortest Path Algorithm - Correctness

- **Lema 2.6:** Durante la ejecución del Algoritmo Genérico de Shortest Path, si  $\text{parent}(u) \neq \text{NULL}$ , entonces  $d(\text{parent}(u)) + w(\text{parent}(u), u) \leq d(u)$

## Demostración:

Por definición,  $\text{parent}(u) \rightarrow u$  es la última arista que cambió el valor de  $d(u)$ . Justo después de hacer la relajación, se tendrá que  $d(u) = d(\text{parent}(u)) + w(\text{parent}(u), u)$ . A partir de ese momento  $d(u)$  no volvió a cambiar, pero es posible que  $d(\text{parent}(u))$  haya disminuido luego por otras relajaciones (o pudo haberse quedado igual). Por otro lado  $w(\text{parent}(u), u)$  es siempre constante porque es el peso de una arista.

Por lo tanto, en las siguientes iteraciones se cumplirá que  $d(\text{parent}(u)) + w(\text{parent}(u), u) \leq d(u)$  ■

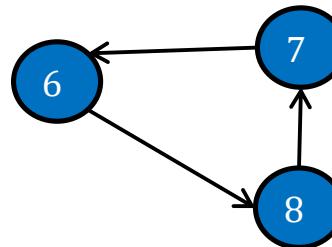
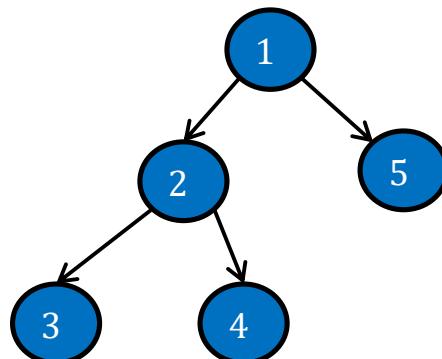
# Generic Shortest Path Algorithm - Correctness

**Definición (Grafo de predecesores)** : Definiremos a  $T(V_T, E_T)$  como el grafo de predecesores de  $G(V, E)$  al subgrafo **no dirigido** de  $G$  inducido por los valores de  $\text{parent}(u)$ . Donde  $V_T$  son todos los nodos en donde  $\text{parent}(u) \neq \text{NULL}$  o  $u = s$ , y  $E_T$  son todas las aristas entre  $u$  y  $\text{parent}(u)$ . Formalmente :

$$V_T = \{u \in V / \text{parent}(u) \neq \text{NULL} \vee u = s\}$$

$$E_T = \{(\text{parent}(u), u) / u \in V_T - \{s\}\}$$

Hemos definido al grafo no dirigido solo por conveniencia. Pero si utilizamos la dirección  $\text{parent}(u) \rightarrow u$  Como cada nodo solo tiene a lo más un predecesor, entonces el indegree de cada nodo es a lo más 1, hipotéticamente el grafo podría tener una forma como esta (las direcciones de las aristas son solo para un mejor entendimiento):



Sin embargo, el siguiente lema define exactamente cuál es la estructura del grafo de predecesores del algoritmo genérico.

# Generic Shortest Path Algorithm - Correctness

- **Lema 2.7:** Si no hay ciclos negativos alcanzables desde  $s$ , el grafo de predecesores  $T(V_T, E_T)$  es un árbol con raíz en  $s$  durante toda la ejecución del algoritmo (luego de la inicialización).

## Demostración:

El grafo es no dirigido por definición. Además tiene  $|V_T| - 1$  aristas. Solo hace falta demostrar que no tiene ciclos.

- **Caso base:** Al inicio  $T$  solo consiste de un nodo  $s$  y no tiene ciclos.
- **Paso inductivo:** Supongamos que se cumple el lema luego de la  $i$ -ésima relajación. Con fines de llegar a una contradicción, supongamos que la siguiente arista a relajar produjera un ciclo. Suponga que el ciclo formado es  $c = v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow \dots \rightarrow v_k$  con  $v_0 = v_k$  y en donde  $\text{parent}(v_i) = v_{i-1}$  para  $1 \leq i \leq k$ . Sin pérdida de generalidad suponga que la última arista en relajarse (y la que creó el ciclo) ha sido la de  $v_{k-1} \rightarrow v_k$ .

Por el lema 2.6 tenemos que  $d(v_{i-1}) + w(v_{i-1}, v_i) \leq d(v_i)$ , para  $1 \leq i < k$

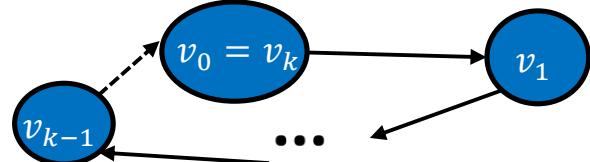
Considere el momento justo **antes** de la relajación de  $v_{k-1} \rightarrow v_0$  (antes de formar el ciclo). En ese momento debió cumplirse que la arista  $v_{k-1} \rightarrow v_0$  estaba tensa, por lo que  $d(v_{k-1}) + w(v_{k-1}, v_0) < d(v_0)$ . Tenemos:

$$w(v_{k-1}, v_0) < d(v_0) - d(v_{k-1})$$

$$w(v_0, v_1) \leq d(v_1) - d(v_0)$$

$$w(v_{k-2}, v_{k-1}) \leq d(v_{k-1}) - d(v_{k-2})$$

Sumando todo tenemos  $w(c) = w(v_{k-1}, v_0) + w(v_0, v_1) + \dots + w(v_{k-2}, v_{k-1}) < 0$ . Lo cual es una contradicción porque no habían ciclos negativos. Por lo tanto, la siguiente arista a relajar no puede producir un ciclo y se mantiene la estructura de árbol. ■



# Generic Shortest Path Algorithm - Correctness

- **Teorema (Correctness of Generic Shortest Path Algorithm):** Si el algoritmo Genérico de Shortest Path termina, entonces para todos los nodos  $u$  alcanzables desde  $s$ ,  $d(u)$  representa la longitud del *path*  $p = s \rightarrow \dots \rightarrow \text{parent}(\text{parent}(u)) \rightarrow \text{parent}(u) \rightarrow u$  que a la vez es un shortest path.

## Demostración:

Por el lema 2.4 sabemos que como el algoritmo termina, entonces no hay ciclos negativos. Por el lema 2.3 sabemos que todos los nodos  $u$  que son alcanzables desde  $s$  tienen  $d(u) \neq +\infty$ . Por el lema 2.7 sabemos que el subgrafo de predecesores es un árbol por lo que  $p = s \rightarrow \dots \rightarrow \text{parent}(\text{parent}(u)) \rightarrow \text{parent}(u) \rightarrow u$  es un *path* válido (sin ciclos). Por el lema 2.6 sabemos que  $d(\text{parent}(u)) + w(\text{parent}(u), u) \leq d(u)$ .

Como el algoritmo terminó, no debe haber aristas tensas, por lo que  $d(\text{parent}(u)) + w(\text{parent}(u), u) \geq d(u)$ .

Las dos desigualdades implican que  $d(\text{parent}(u)) + w(\text{parent}(u), u) = d(u) \dots (1)$

Para una mejor visualización renombremos algunas variables  $p = s \rightarrow v_1 \rightarrow v_2 \rightarrow \dots \rightarrow v_k \rightarrow u$

Aplicando la ecuación (1) en cada arista del *path*  $p$ :

$$w(s, v_1) = d(v_1) - d(s)$$

~~$$w(v_1, v_2) = d(v_2) - d(v_1)$$~~

~~$$w(v_2, v_3) = d(v_3) - d(v_2)$$~~

...

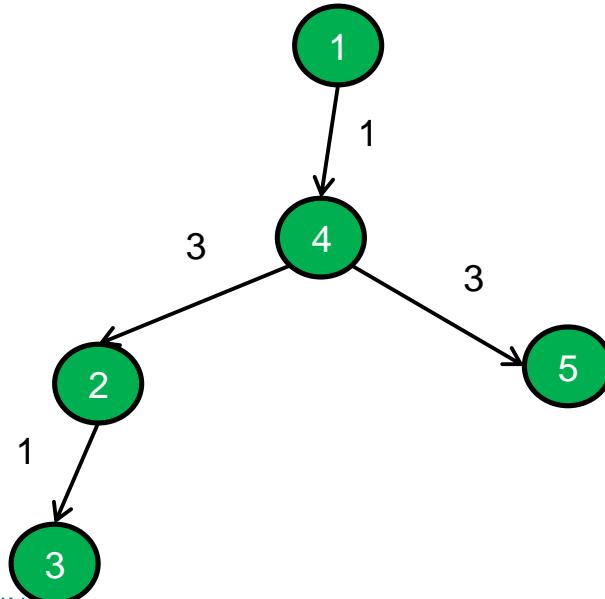
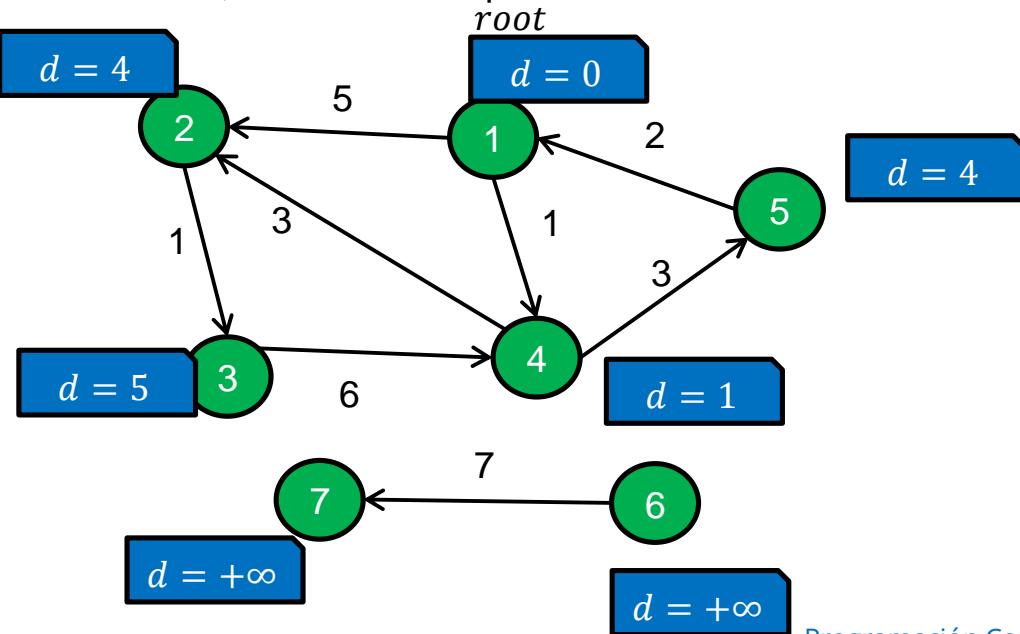
~~$$w(v_k, u) = d(u) - d(v_k)$$~~

Sumando todo tenemos que  $w(p) = w(s, v_1) + w(v_1, v_2) + w(v_2, v_3) + \dots + w(v_k, u) = d(u) - d(s) = d(u)$  que es igual a  $\delta(u)$  por el lema 2.5. ■

# Shortest Path Tree

**Definición:** Un shortest path tree de  $G(V, E, w)$  es un subgrafo de  $G$  no dirigido, conexo y acíclico (árbol) en donde cada camino que va desde  $s$  a  $u$  en el árbol, es un shortest path del grafo original  $G$

**Corolario (Predecessor subgraph property):** El grafo de predecesores, al finalizar el Algoritmo Genérico de Shortest Path, es un shortest path tree.

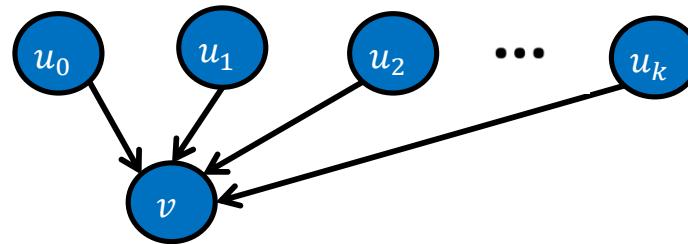


# Generic Shortest Path Algorithm

□ **Lema 2.8:** Al terminar el Algoritmo Genérico de Shortest Path, todos los nodos  $v \neq s$  alcanzables desde  $s$  cumplen que  $d(v) = \min_{(u,v) \in E} \{d(u) + w(u,v)\}$ .

**Demostración:**

Sabemos que al terminar el algoritmo, todos los nodos cumplen que  $d(v) = \delta(v)$ . Considere todos los nodos  $u_i$  para los que existe alguna arista  $(u_i, v)$ .



Por **triangle inequality** se debe cumplir que  $d(v) \leq d(u_i) + w(u_i, v)$ . Es decir  $d(v) \leq \min_{(u,v) \in E} \{d(u) + w(u,v)\}$

Supongamos (con fines de contradicción) que  $d(v) < \min_{(u,v) \in E} \{d(u) + w(u,v)\} \dots (1)$ . Sea  $p = s \rightsquigarrow x \rightarrow v$  un shortest path hasta  $v$ . Entonces  $d(v) = d(x) + w(x, v)$  (por **optimal substructure**).

Pero (1) implica que  $d(v) < d(x) + w(x, v)$  llegando a una contradicción.

$$\Rightarrow d(v) = \min_{(u,v) \in E} \{d(u) + w(u,v)\}$$

# Cota del número de relajaciones

Como se mencionó anteriormente, la complejidad del Generic Shortest Path Algorithm dependerá de cómo se seleccione las aristas tensas. Sin embargo, también podemos obtener una posible cota utilizando el hecho de que  $d(u)$  en todo momento es  $+\infty$  o la longitud de algún path de  $s$  a  $u$  (Lema 2.2), y en cada relajación este  $d(u)$  decrece por lo que sería la longitud de **un nuevo path** en cada relajación. Como hay  $O((V - 1)!)$  paths desde  $s$  en un grafo completo, entonces esa sería una primera cota para el número de relajaciones.

Si las aristas tienen pesos enteros podemos hallar otra fórmula para la cota. Suponga que el máximo valor absoluto de los pesos de las aristas es  $C$  (asumiendo  $C > 0$ ). Es decir, el máximo peso de un path está limitado por  $VC$  (porque un path tiene a lo más  $(V - 1)$  aristas) y el mínimo peso de un path por  $-VC$ . El peor caso sería que  $d(u)$  sea  $VC$  en la primera relajación y que  $\delta(u) = -VC$  y que en cada relajación  $d(u)$  decrezca en 1. Por lo que el número de relajaciones de cada nodo  $u$  está acotado por  $2VC = O(VC)$ . Multiplicando por el número de nodos tendríamos una complejidad de  $O(V^2C)$ .

Tipo de grafo	Complejidad
Pesos reales	$O((V - 1)!)$
Pesos enteros	$O(\min((V - 1)!, V^2C))$

**Nota:** Igual la complejidad completa del algoritmo también dependerá qué tan eficiente **encuentres** una arista tensa. Por ejemplo, si para encontrarla demoras  $O(E)$  tu complejidad podría llegar a  $O(E \times (V - 1)!)$

# Linear Programming

El Generic Shortest Path Algorithm nos marco general para resolver el problema del Shortest Path. Pero también nos otros resultados interesantes. Note que al finalizar el algoritmo no hay aristas tensas, por lo que todas las aristas  $(u, v)$  del grafo deben cumplir **triangle inequality** :  $d(u) + w(u, v) \geq d(v)$ .

Sin embargo, cumplir esas ecuaciones no es suficiente, debido a que si por ejemplo establezco  $d(u) = 0$  para todos los nodos, se cumplirán todas las ecuaciones sin necesariamente formar shortest distances válidas.

Sin embargo, por el lema 2.8, se debe cumplir que  $\delta(v) = d(v) = \min_{(u,v) \in E} \{d(u) + w(u, v)\}$  . Es decir,  $d(v)$

tendrá el mayor valor posible tal que se cumpla la propiedad de **triangle inequality**.

Podemos formular un **programa lineal** para poder encontrar la longitud del shortest path de  $s$  a  $t$  (o a todos los nodos).

$$\begin{aligned} & \max d(t) \\ \text{s. t.: } \\ & d(s) = 0 \\ & d(v) - d(u) \leq w(u, v), \text{para toda arista } (u, v) \in E \end{aligned}$$

$$\begin{aligned} & \max \sum d(u) \\ \text{s. t.: } \\ & d(s) = 0 \\ & d(v) - d(u) \leq w(u, v), \text{para toda arista } (u, v) \in E \end{aligned}$$

Maximizar  $d(t)$  hará que se también se maximicen todos los estimados de los nodos que estén en el shortest path entre  $s$  y  $t$ . Aunque habrán ciertos nodos que no tengan calculado correctamente su distancia. Si queremos obtener todas las distancias (para todos los nodos), podemos plantear el programa de la derecha.

# Modified Generic Shortest Path Algorithm

Cuando uno quiera implementar el algoritmo Genérico, uno puede aprovecharse de la **Tense-Relaxtion Property** y notar que cada vez que relajamos un vértice  $u$  solo deberíamos agregar **outgoing edges** de  $u$ . Podemos utilizar un **contenedor general**, llamémoslo “lista” en donde estarán presentes aquellos nodos que tengan alguna **potencial** arista tensa.

A la derecha se muestra un pseudocódigo de este algoritmo. No te que las líneas 8, 9, 10 no son realmente necesarias para que el algoritmo funcione. Pero pueden ayudar a mejorar la eficiencia ya que no tiene sentido insertar en la lista un nodo que ya está en ella.

---

## Algorithm 2: Modified-FordSSSP

```
input :  $G(V_G, E_G, w)$  , source :  $s$ 
output:  $d[ ]$ , parent[ ]
1 Initialize ( $s$ )
2  $list = \{s\}$ 
3 while  $list$  is not empty do
4     Remove a node  $u$  from  $list$ 
5     foreach edge  $u \rightarrow v$  in  $E_G$  do
6         if  $u \rightarrow v$  is tense then
7             Relax ( $u, v$ )
8             if  $v$  is not in  $list$  then
9                 Add node  $v$  to  $list$ 
10            end
11        end
12    end
13 end
14 return  $d, parent$ 
```

---



# Modified Generic Shortest Path Algorithm

- **Lema 2.9:** Al inicio de cada iteración del bucle while del Modified Generic Shortest Path Algorithm (línea 4), si una arista  $(u, v) \in E$  está tensa entonces  $u$  está en la lista del algoritmo.

## Demostración:

Demostremos por inducción en el número de iteraciones del bucle while

- **Caso base:**

En la primera iteración las únicas aristas tensas son  $e = s \rightarrow u$  ( $e \in E$ ) ya que  $d(u) = +\infty$  para todo nodo  $u \in V$  con  $u \neq s$  y  $d(s) = 0$ . Y el lema cumple ya que  $s$  está en la cola (y es el único).

- **Paso inductivo:**

Suponga que en al inicio  $i - \text{ésima}$  iteración se cumple el lema. En ese momento se hará sacar el nodo  $u$  y luego se agregarán todos los nodos  $v$  tal que  $u \rightarrow v$  esté tensa. Sea  $S = v_0, v_1, \dots, v_k$  la secuencia de nodos agregados en la  $i - \text{ésima}$  iteración . Por **Tense-Relaxation Property**, al relajar la arista  $u \rightarrow v_i$ , las únicas aristas tensas que se pueden crear son  $(v_i \rightarrow z) \in E$  y como  $v_i$  ha sido agregado en la cola, entonces estas aristas mantendrán la invariante. Además al relajar la arista  $u \rightarrow v_i$  esta dejará de estar tensa por lo que ya no es necesario que  $u$  esté en la cola.

Por ello, al comenzar la  $(i + 1) - \text{ésima}$  iteración, se cumplirá el lema, lo cual finaliza la inducción y demuestra el lema. ■

- **Corolario:** Al finalizar el algoritmo, no hay aristas tensas y, por lo tanto, las distancias calculadas son correctas y el grafo inducido por  $\text{parent}(u)$  es un shortest path tree.

# Complejidad del Modified Generic Shortest Path Algorithm

Es fácil notar que el peor caso del **número de relajaciones** esta versión modificada no puede ser peor que el peor caso de la versión más general (la planteada por Ford). Por lo tanto una primera cota para el número de relajaciones sería  $O((V - 1)!)$ , donde  $V$  es el número de vértices.

Sin embargo, podemos ir un poco más allá de esto. Aunque no lo probaremos formalmente, **el número de relajaciones de este algoritmo es  $O(2^V)$**  (la cota exacta es  $2^{V-1}$ ). La justificación va por el lado de que si en algún momento el estimado  $d(u)$  fue obtenido a partir del path  $s, v_1, v_2, \dots, v_k = u$ , es imposible que cualquier otro path obtenido a partir de  $s + \text{algún reordenamiento de } \{v_1, v_2, \dots, v_k\}$  pueda llegar a cambiar algún otro estimado a través de relajaciones (se demuestra por inducción). Es decir ya no serán “inspeccionados” todos los posibles paths, sino un menor número que puede ser obtenido a partir de suma de combinatorias (ya que no importa el orden de cada conjunto de nodos ya que solo 1 de los posibles órdenes existirá)  $\#relax = \sum_{k=0}^{V-1} C_k^{V-1} = 2^{V-1}$ .

Además, cada relajación tiene por consiguiente una entrada de un nuevo nodo a la cola (en caso este todavía no esté ahí) y cada vez que entra un nodo, se inspeccionan todas sus outgoing edges que están acotadas por  $E$  (o por  $V$  en caso no hayan múltiples aristas) por lo que **la complejidad total del algoritmo sería  $O(E \times 2^V)$  o de  $O(V \times 2^V)$  dependiendo del caso.**

# Modified Generic Shortest Path Algorithm

Como podemos ver, este algoritmo modificado posee un **contenedor general** que lo hemos definido como “lista”.

**¿Qué opciones de contenedor podríamos utilizar específicamente para implementar el algoritmo?**



# Modified Generic Shortest Path Algorithm

Como podemos ver, este algoritmo modificado posee un **contenedor general** que lo hemos definido como “lista”.

**¿Qué opciones de contenedor podríamos utilizar específicamente para implementar el algoritmo?**

Stack

Queue

Priority  
queue

Deque

Exploraremos algunas de estas opciones en los siguientes algoritmos. Pero desde ya sepan que si usan un **stack** la complejidad es exponencial (y sí se pueden construir contraejemplos que tarde una cantidad exponencial de relajaciones). Por lo que no exploraremos la opción del stack.

# Contenido

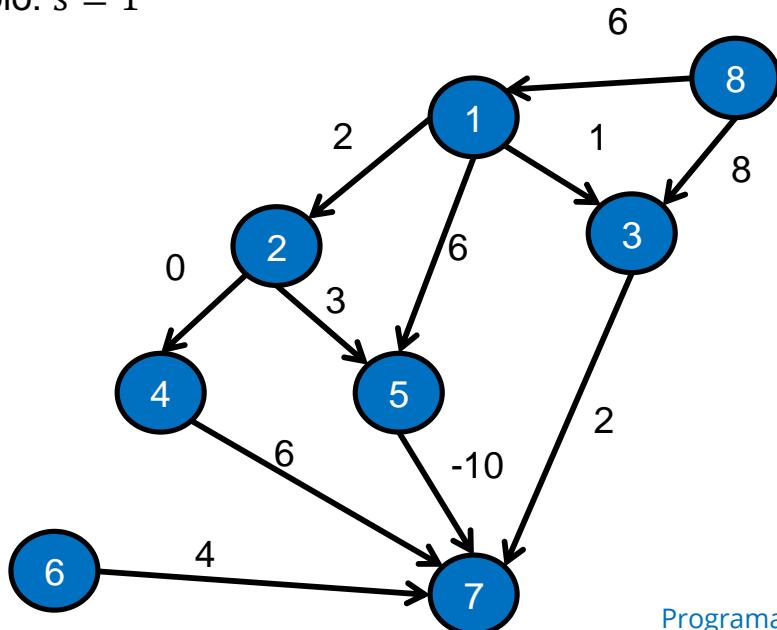
1. Introducción	
2. Propiedades de los Shortest Path y Algoritmo Genérico	
<b>3. Single Source Shortest Path en DAG</b>	
4. Single Source Shortest Path con pesos unitarios	
5. Single Source Shortest Path con pesos no negativos	
6. Single Source Shortest Path con cualquier peso real	
7. All-pairs shortest path	

# Single Source Shortest Path – DAG

**Problema:** Sea un grafo dirigido acíclico (DAG) con pesos  $G(V, E, w)$ .

Cuya función de pesos es  $w: E \rightarrow \mathbb{R}$ . Se pide hallar el peso del shortest path desde un nodo fuente  $s$  hasta cualquier otro nodo  $u$  (distancia del nodo  $u$ ), denotado como  $\delta(u)$

Ejemplo:  $s = 1$



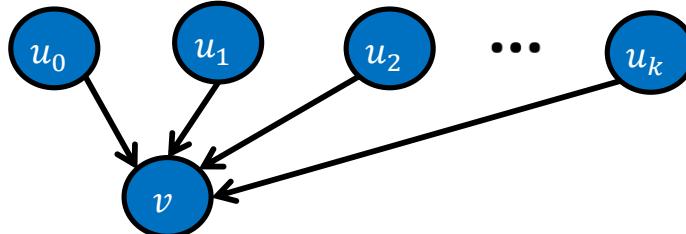
$u$	$\delta(u)$
1	0
2	2
3	1
4	2
5	5
6	$+\infty$
7	-5
8	$+\infty$

# SSSP - Dynamic Programming

Como el grafo es un DAG, no existen ciclos (ni ciclos negativos) así que para todos los nodos alcanzables desde  $s$ , siempre existe un shortest path.

Supongamos que para un nodo  $v$  existiera algún shortest path desde  $s$ . Sea  $p = s \rightsquigarrow u \rightarrow v$  un shortest path hasta  $v$ . Por la propiedad de **optimal substructure** sabemos que  $\delta(v) = \delta(u) + w(u, v)$ . Sin embargo, un nodo  $v$  puede tener muchos nodos candidatos  $u$  con los que tenga alguna arista. Pero por el lema 2.8 tenemos que  $\delta(v) = \min_{(u,v) \in E} \{\delta(u) + w(u, v)\}$ .

Justamente la propiedad de optimal substructure y el hecho de que no haya ciclos, nos garantiza que podemos aplicar **dynamic programming**.



**Caso base:**  $\delta(s) = 0$

**Recursividad:** Para hallar  $\delta(v)$ , recursivamente hallamos  $\delta(u)$  para los  $u$  que tengan una arista  $(u, v)$  y luego aplicamos  $\delta(v) = \min_{(u,v) \in E} \{\delta(u) + w(u, v)\}$

# Dynamic Programming

Para la implementación, tener en cuenta que vamos a recorrer la lista de adyacencia en orden **inverso**. Es decir para un determinado  $v$  queremos todos los nodos  $u$  tal que exista la arista  $u \rightarrow v$

**Complejidad:**  $O(V + E)$

```
const Long INF = 1e18;
const Long MX = 1e5;
struct Graph{
    vector<pair<Long, Long>> rev[MX];
    Long d[MX];
    bool used[MX];

    void addEdge(Long u, Long v, Long w) {
        rev[v].push_back({u, w});
    }

    Long distance(Long u, Long s) { //O(V + E)
        if (u == s) {
            return 0;
        }
        if (used[u]) {
            return d[u];
        }
        used[u] = true;
        d[u] = INF;
        for (auto e : rev[u]) {
            Long v = e.first;
            Long w = e.second;
            d[u] = min(d[u] , distance(v, s) + w);
        }
        return d[u];
    }
} G;
```

# Longest Path en DAG

Supongamos que ahora nos pidieran obtener el path con peso **máximo** en un DAG. ¿Podemos adaptar nuestra solución de shortest path para que también resuelva este problema?

## ¡Multipliquemos las aristas por -1!

¿Por qué funciona? Sea  $p'$  un shortest path que va desde  $s$  a  $u$  en el grafo con las aristas multiplicadas por -1. Sea  $p$  el path original (sin multiplicar por -1). Entonces  $w(p') = -w(p)$ .

Como  $p'$  es shortest path, es el path con mínimo  $w(p')$ , lo cual minimiza  $-w(p)$  que es equivalente a maximizar  $w(p)$ . Por lo que  $p$  sería shortest path.

Podemos aplicar el mismo algoritmo de dp. Pero **no olvide también multiplicar por -1 la distancia final** obtenida como resultado.

- Una solución alternativa es hacer dp pero con la ecuación  $d(v) = \max_{(u,v) \in E} \{d(u) + w(u,v)\}$ . Lo cual puede ser demostrado por inducción.

**Nota:** Si el grafo NO es un DAG. El problema se vuelve NP-Hard.

# SPPP – Topological Sorting

Aunque pareciera que la solución con dp no usara el Generic Shortest Path Algorithm, en realidad en el código sí se realizan relajaciones implícitas dentro del bucle for.

Sin embargo, existe una forma un poco más natural de ver estas relajaciones si pasamos el dp a una forma iterativa utilizando **topological sorting**. 

El algoritmo anterior igual necesita ir a la lista de adyacencia reversa de cada nodo (ya que hace lo mismo que el dp, pero de forma iterativa). Pero aquí tenemos la ventaja de que  podemos cambiarlo a la lista de adyacencia directa.

La justificación de por qué funciona es una aplicación directa de [path relaxation property](#).

Sea  $p = v_0 \rightarrow v_1 \rightarrow \dots \rightarrow v_k$  un shortest path donde  $v_0 = s$ . El ordenamiento topológico garantizará que la arista  $v_i \rightarrow v_{i+1}$  sea relajada antes que la arista  $v_{i+1} \rightarrow v_{i+2}$

---

### Algorithm 3: TopoSort SSSP-DAG

---

```
input :  $G(V_G, E_G, w)$  , source :  $s$ 
output:  $d[ ]$ , parent[ ]
1 Initialize( $s$ )
2 foreach node  $v$  in  $V_G$  in topological sort do
3   foreach ingoing edge  $u \rightarrow v$  in  $E_G$  do
4     if  $u \rightarrow v$  is tense then
5       | Relax( $u, v$ )
6     end
7   end
8 end
9 return  $d, parent$ 
```

---

### Algorithm 4: TopoSort2 SSSP-DAG

---

```
input :  $G(V_G, E_G, w)$  , source :  $s$ 
output:  $d[ ]$ , parent[ ]
1 Initialize( $s$ )
2 foreach node  $u$  in  $V_G$  in topological sort do
3   foreach outgoing edge  $u \rightarrow v$  in  $E_G$  do
4     if  $u \rightarrow v$  is tense then
5       | Relax( $u, v$ )
6     end
7   end
8 end
9 return  $d, parent$ 
```

---

# SPPP – Topological Sorting

```
const Long MX = 1e5;
const Long INF = 1e18;
struct Graph{
    vector<pair<Long, Long>> adj [MX];
    Long d[MX];
    Long indegree[MX];

    void addEdge(Long u, Long v, Long w) {
        adj[u].push_back({v, w});
        indegree[v]++;
    }
}
```

```
void shortestPath(Long s, Long n) { //O(V + E)
    deque<Long> q;
    for (Long i = 0; i < n; i++) {
        d[i] = INF;
        if (indegree[i] == 0) {
            q.push_back(i);
        }
    }
    d[s] = 0;

    while(!q.empty()) { //topological sort
        Long u = q.front();
        q.pop_front();
        for (auto e : adj[u]) {
            Long v = e.first;
            Long w = e.second;
            if (d[u] != INF && d[u] + w < d[v]) { //tense
                d[v] = d[u] + w; //relax
            }
            indegree[v]--;
            if (indegree[v] == 0) {
                q.push_back(v);
            }
        }
    }
}
```

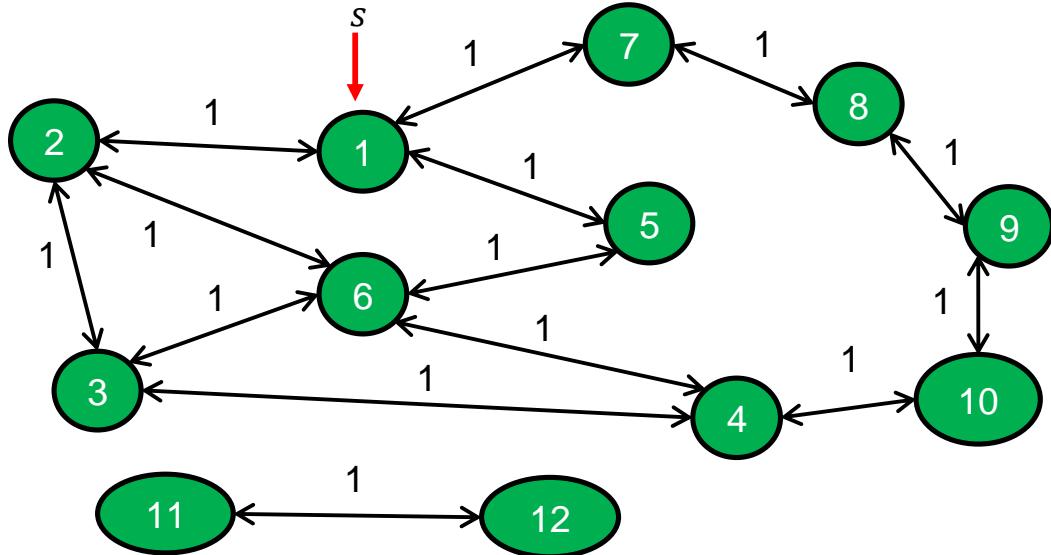
# Contenido

1. Introducción	
2. Propiedades de los Shortest Path y Algoritmo Genérico	
3. Single Source Shortest Path en DAG	
<b>4. Single Source Shortest Path con pesos unitarios</b>	
5. Single Source Shortest Path con pesos no negativos	
6. Single Source Shortest Path con cualquier peso real	
7. All-pairs shortest path	

# Single Source Shortest Path – Unweighted graph

**Problema:** Sea un grafo dirigido con pesos  $G(V, E, w)$ . Cuya función de pesos es  $w: E(G) \rightarrow \{1\}$ . Es decir todas las aristas tienen peso 1.

Se pide hallar la longitud del shortest path desde un nodo fuente  $s$  hasta cualquier otro nodo  $u$  (distancia del nodo  $u$ ), denotado como  $\delta(u)$



$u$	$\delta(u)$
1	0
2	1
3	2
4	3
5	1
6	2
7	1
8	2
9	3
10	4
11	$+\infty$
12	$+\infty$

# Breadth First Search (BFS)

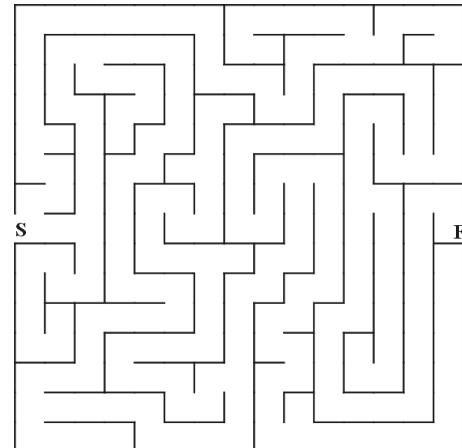
En español **Búsqueda en Amplitud**. Fue inventado por Konrad Zuse en 1945 para su tesis de doctorado sobre un lenguaje de programación creado por él llamado Plankalkül. En su tesis aparece un ejemplo de cómo utilizar en lenguaje, en el que se describe el algoritmo de bfs para contar componentes conexas de un grafo. Sin embargo, debido al fin del imperio Nazi, no pudo publicar su trabajo hasta 1972.

Luego fue redescubierto por Edward Moore en 1959 que en “Shortest Path through a maze” con el nombre de “Algorithm A” que servía para encontrar el shortest path (según el número de aristas) en un laberinto.



*Edward Moore*

*Konrad Zuse*



# Breadth First Search (BFS)

El algoritmo irá recorriendo el grafo y en cada iteración irá expandiendo la frontera entre vértices visitados vs no visitados a lo largo del “ancho” de esta. El algoritmo tiene la característica de que irá recorriendo los vértices en orden no decreciente de su distancia (en términos de cantidad de aristas) con respecto a la fuente  $s$ .

Para lograr esto, BFS utiliza una cola FIFO. Para ello distinguiremos 3 posibles estados de un nodo:

Estado	Color
No visitado	Blanco
En cola	Gris
Visitado	Verde

Cada vez que desencolemos un nodo  $u$ , lo estaremos visitando (cambiando su estado a “visitado”). Luego inspeccionaremos a todos sus vecinos  $v$  que estén en el estado “no visitado”, les actualizaremos su distancia estimada  $d(v) = d(u) + 1$  y luego pondremos a  $v$  en la cola, cambiando su estado a “en cola”

---

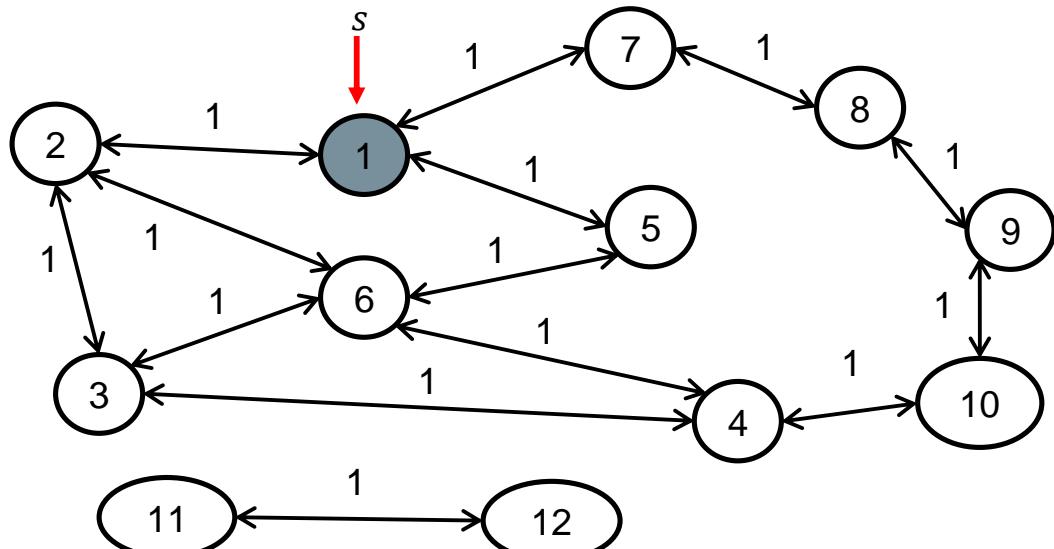
## Algorithm 5: BFS

```
input :  $G(V_G, E_G)$  , source :  $s$ 
output:  $d[ ]$ , parent[ ]
1 Initialize( $s$ )
2  $q = \emptyset$ 
3  $q.push(s)$ 
4 while  $q$  is not empty do
5    $u = q.pull()$ 
6   Mark  $u$  as visited
7   foreach edge  $u \rightarrow v$  in  $E_G$  do
8     if  $v$  is not visited then
9        $d(v) = d(u) + 1$  // relax
10      parent( $v$ ) =  $u$ 
11       $q.push(v)$ 
12      Mark  $v$  as in queue
13    end
14  end
15 end
16 return  $d$ , parent
```

---

# Breadth First Search (BFS)

Cola	Nodo	1									
	d	0									

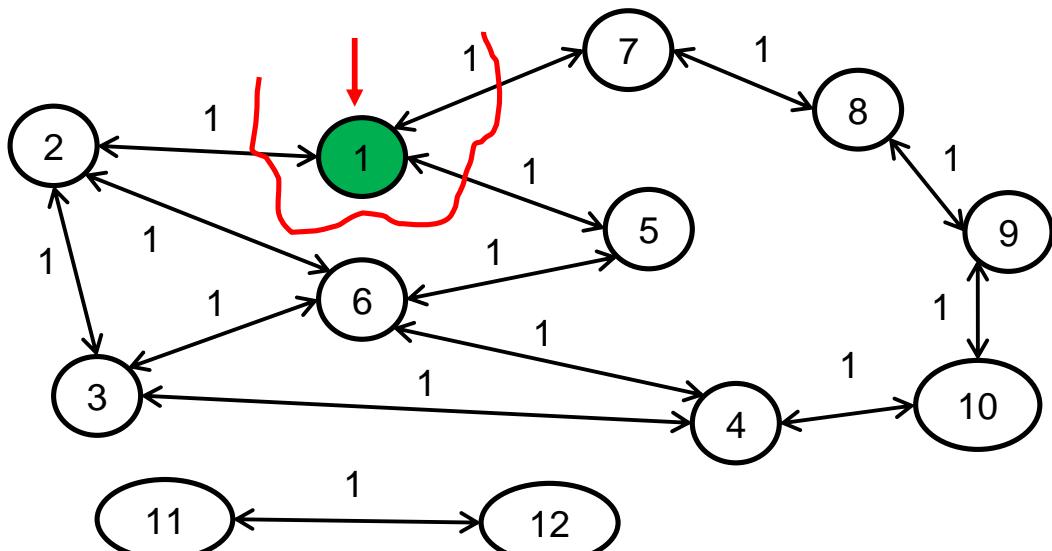


Red arrow pointing from the graph to the distance table.

$u$	$d(u)$
1	0
2	$+\infty$
3	$+\infty$
4	$+\infty$
5	$+\infty$
6	$+\infty$
7	$+\infty$
8	$+\infty$
9	$+\infty$
10	$+\infty$
11	$+\infty$
12	$+\infty$

# Breadth First Search (BFS)

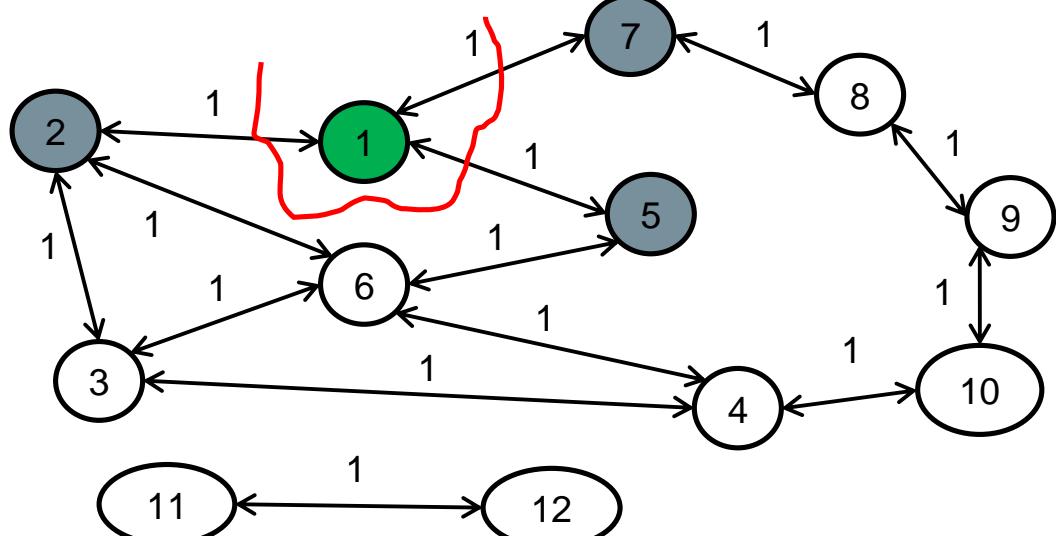
Cola	Nodo										
	d										



$u$	$d(u)$
1	0
2	$+\infty$
3	$+\infty$
4	$+\infty$
5	$+\infty$
6	$+\infty$
7	$+\infty$
8	$+\infty$
9	$+\infty$
10	$+\infty$
11	$+\infty$
12	$+\infty$

# Breadth First Search (BFS)

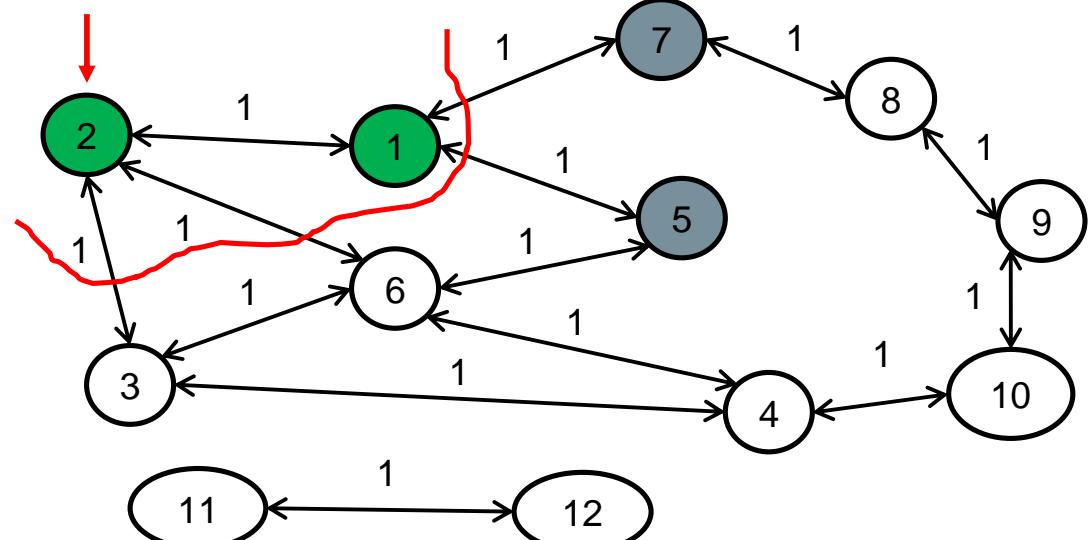
Cola	Nodo	2	5	7					
	d	1	1	1					



$u$	$d(u)$
1	0
2	1
3	$+\infty$
4	$+\infty$
5	1
6	$+\infty$
7	1
8	$+\infty$
9	$+\infty$
10	$+\infty$
11	$+\infty$
12	$+\infty$

# Breadth First Search (BFS)

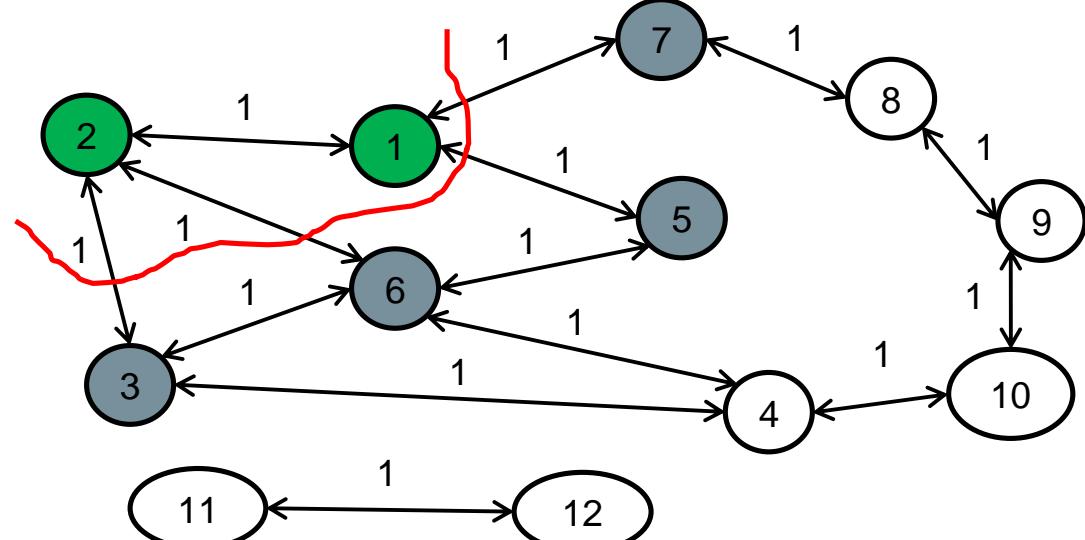
Cola	Nodo	5	7								
	d	1	1								



$u$	$d(u)$
1	0
2	1
3	$+\infty$
4	$+\infty$
5	1
6	$+\infty$
7	1
8	$+\infty$
9	$+\infty$
10	$+\infty$
11	$+\infty$
12	$+\infty$

# Breadth First Search (BFS)

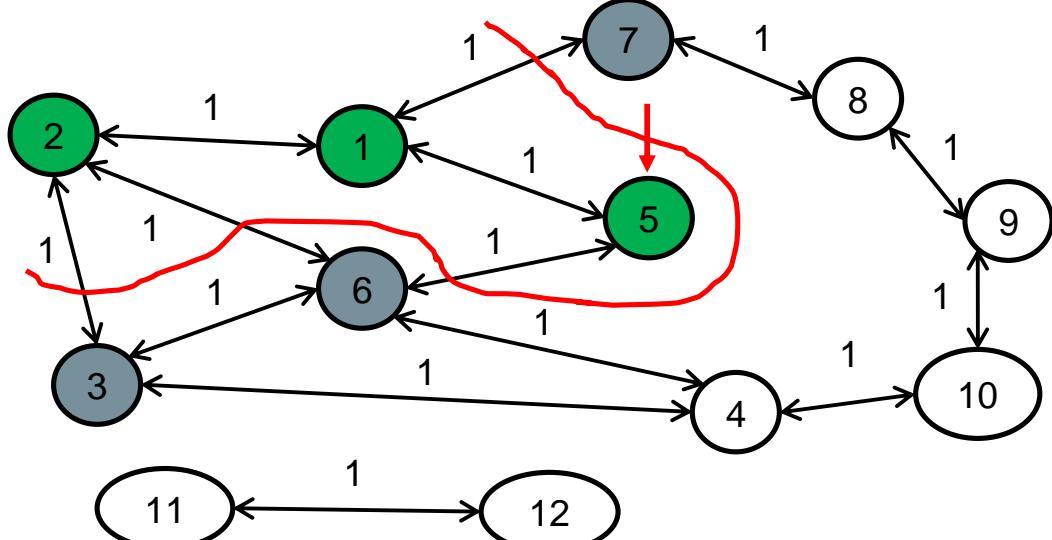
Cola	Nodo	5	7	3	6				
	d	1	1	2	2				



$u$	$d(u)$
1	0
2	1
3	2
4	$+\infty$
5	1
6	2
7	1
8	$+\infty$
9	$+\infty$
10	$+\infty$
11	$+\infty$
12	$+\infty$

# Breadth First Search (BFS)

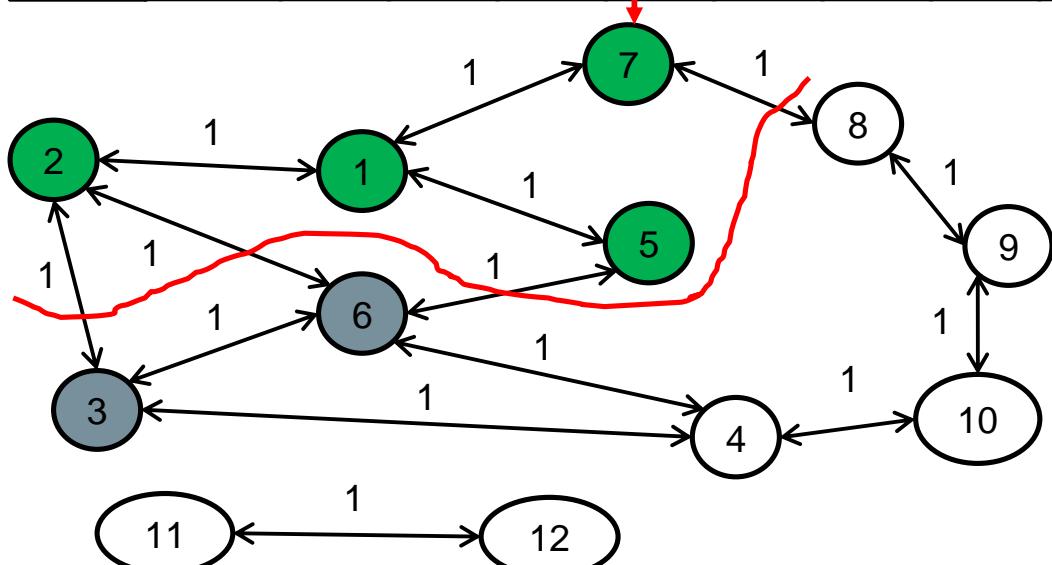
Cola	Nodo	7	3	6					
	d	1	2	2					



$u$	$d(u)$
1	0
2	1
3	2
4	$+\infty$
5	1
6	2
7	1
8	$+\infty$
9	$+\infty$
10	$+\infty$
11	$+\infty$
12	$+\infty$

# Breadth First Search (BFS)

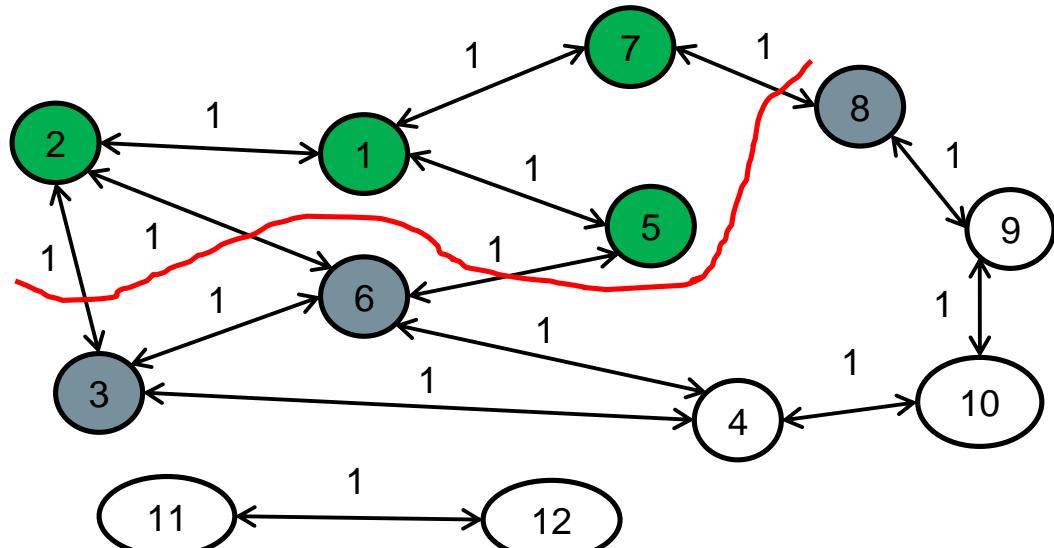
Cola	Nodo	3	6								
	d	2	2								



$u$	$d(u)$
1	0
2	1
3	2
4	$+\infty$
5	1
6	2
7	1
8	$+\infty$
9	$+\infty$
10	$+\infty$
11	$+\infty$
12	$+\infty$

# Breadth First Search (BFS)

Cola	Nodo	3	6	8					
	d	2	2	2					

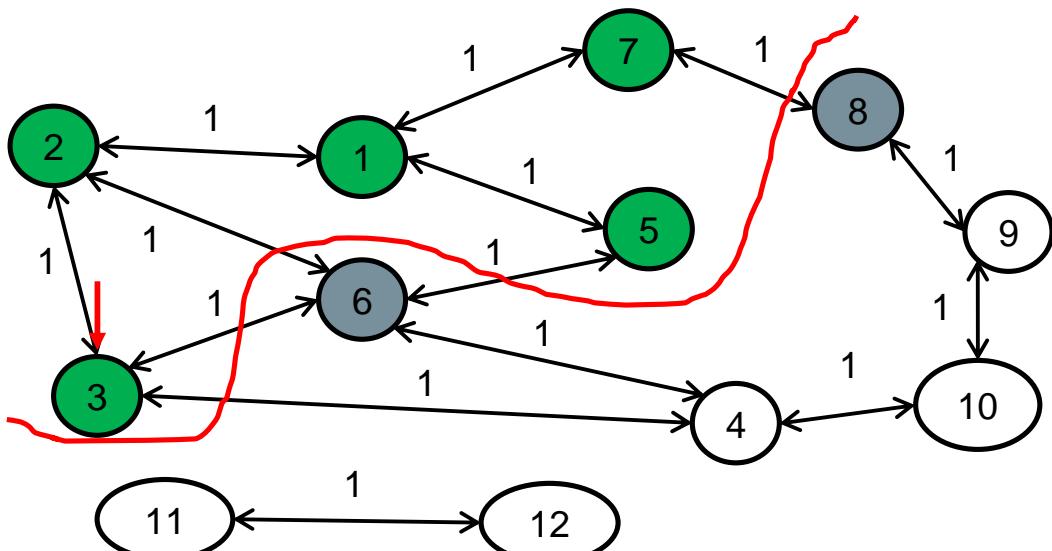


$u$	$d(u)$
1	0
2	1
3	2
4	$+\infty$
5	1
6	2
7	1
8	2
9	$+\infty$
10	$+\infty$
11	$+\infty$
12	$+\infty$



# Breadth First Search (BFS)

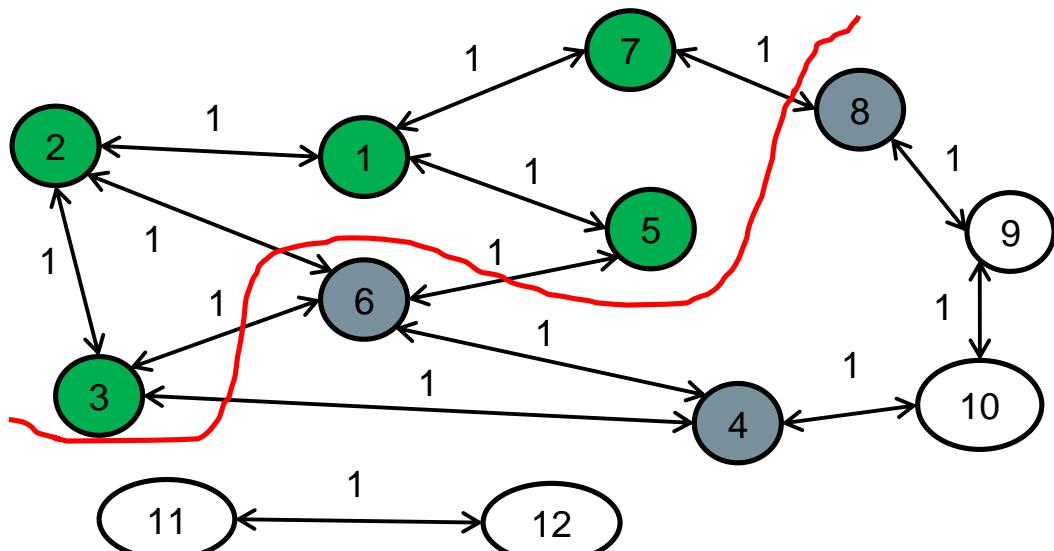
Cola	Nodo	6	8								
	d	2	2								



$u$	$d(u)$
1	0
2	1
3	2
4	$+\infty$
5	1
6	2
7	1
8	2
9	$+\infty$
10	$+\infty$
11	$+\infty$
12	$+\infty$

# Breadth First Search (BFS)

Cola	Nodo	6	8	4					
	d	2	2	3					

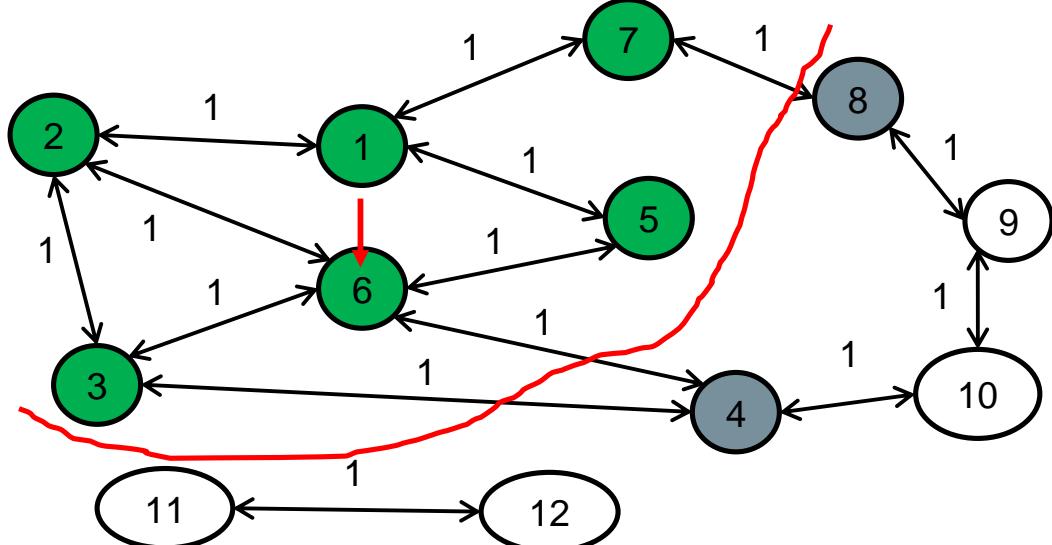


→

$u$	$d(u)$
1	0
2	1
3	2
4	3
5	1
6	2
7	1
8	2
9	$+\infty$
10	$+\infty$
11	$+\infty$
12	$+\infty$

# Breadth First Search (BFS)

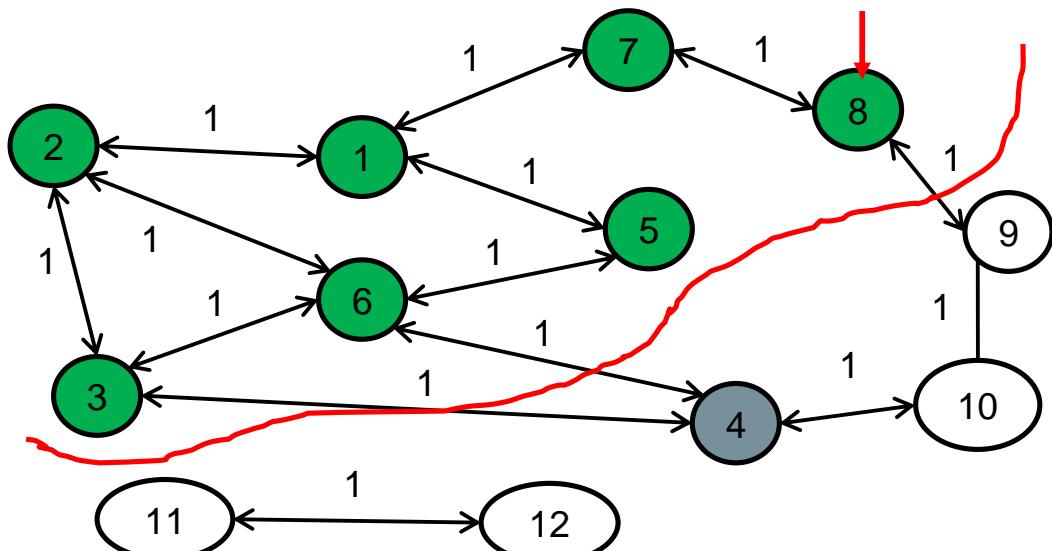
Cola	Nodo	8	4							
	d	2	3							



$u$	$d(u)$
1	0
2	1
3	2
4	3
5	1
6	2
7	1
8	2
9	$+\infty$
10	$+\infty$
11	$+\infty$
12	$+\infty$

# Breadth First Search (BFS)

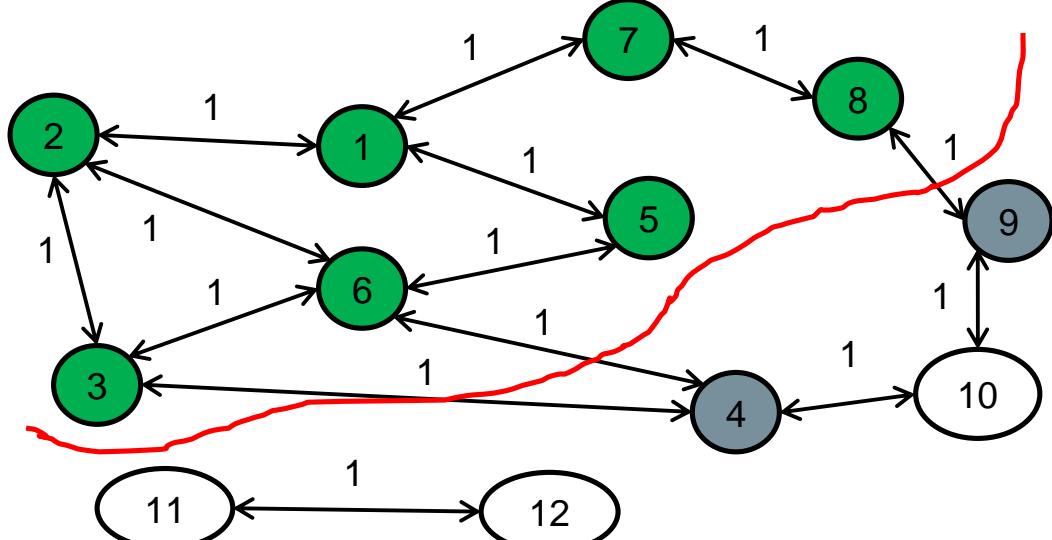
Cola	Nodo	4									
	d	3									



$u$	$d(u)$
1	0
2	1
3	2
4	3
5	1
6	2
7	1
8	2
9	$+\infty$
10	$+\infty$
11	$+\infty$
12	$+\infty$

# Breadth First Search (BFS)

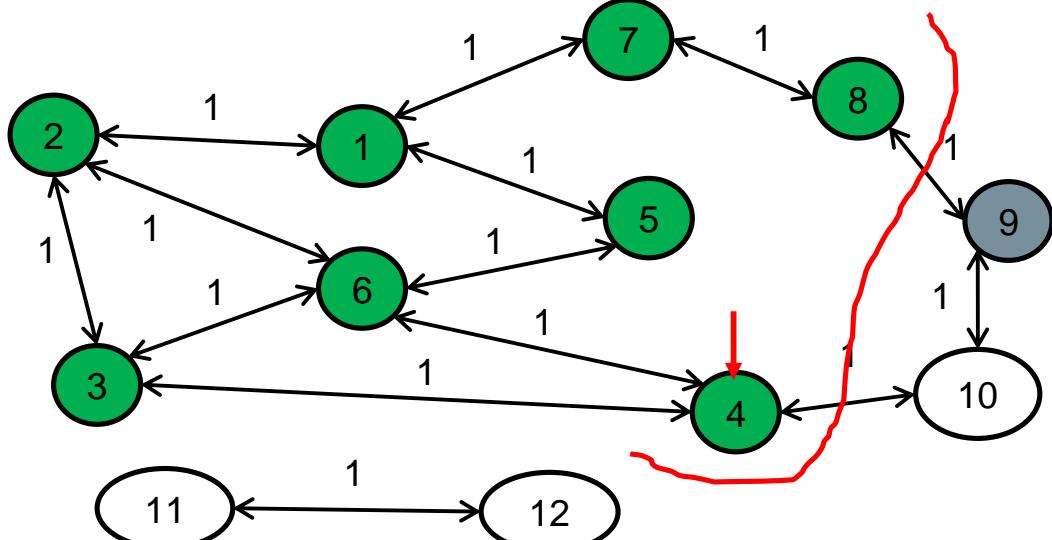
Cola	Nodo	4	9							
	d	3	3							



$u$	$d(u)$
1	0
2	1
3	2
4	3
5	1
6	2
7	1
8	2
9	3
10	$+\infty$
11	$+\infty$
12	$+\infty$

# Breadth First Search (BFS)

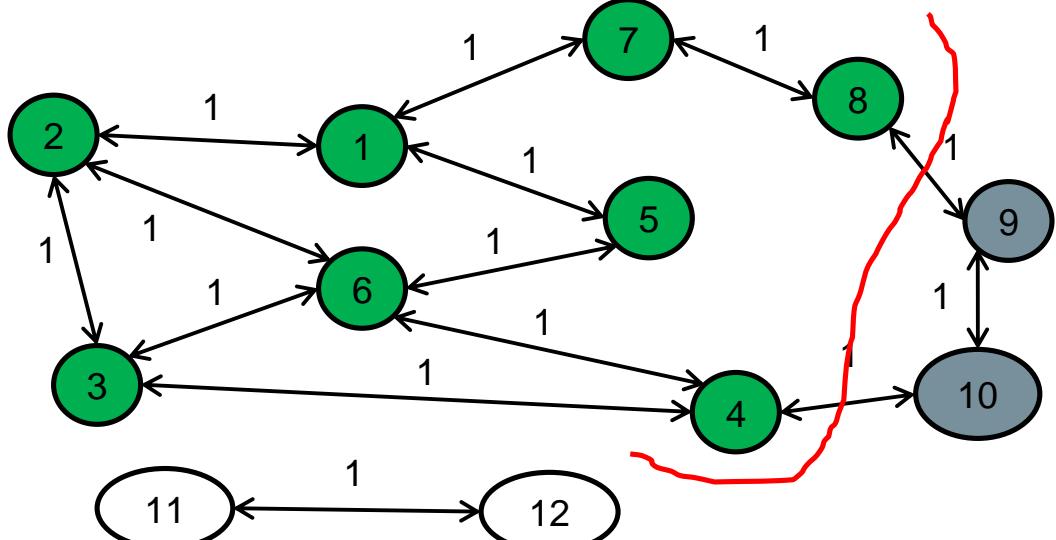
Cola	Nodo	9								
	d	3								



$u$	$d(u)$
1	0
2	1
3	2
4	3
5	1
6	2
7	1
8	2
9	3
10	$+\infty$
11	$+\infty$
12	$+\infty$

# Breadth First Search (BFS)

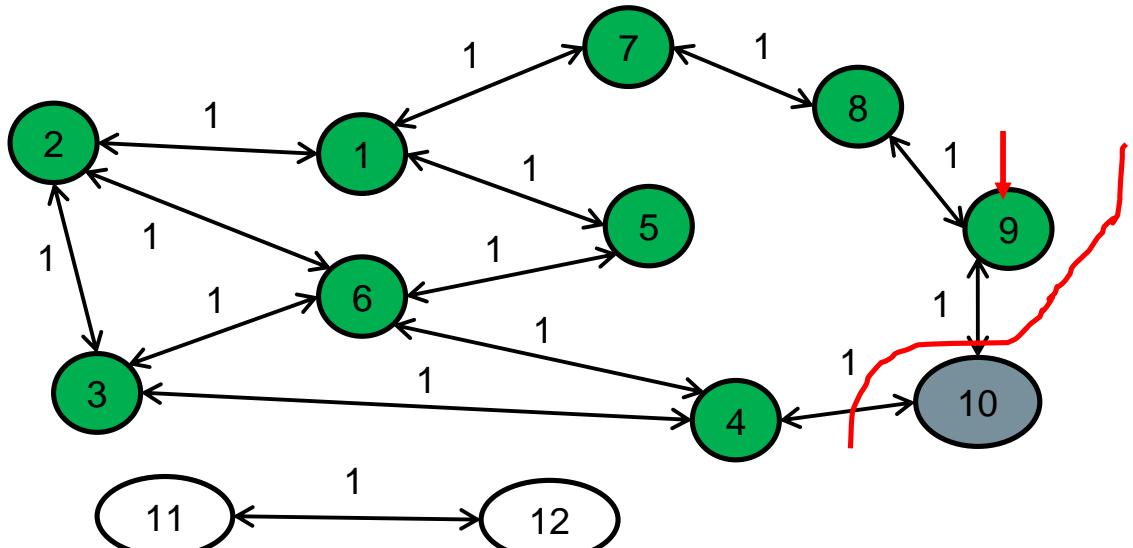
<b>Cola</b>	<b>Nodo</b>	9	10						
	<b>d</b>	3	4						



$u$	$d(u)$
1	0
2	1
3	2
4	3
5	1
6	2
7	1
8	2
9	3
10	4
11	$+\infty$
12	$+\infty$

# Breadth First Search (BFS)

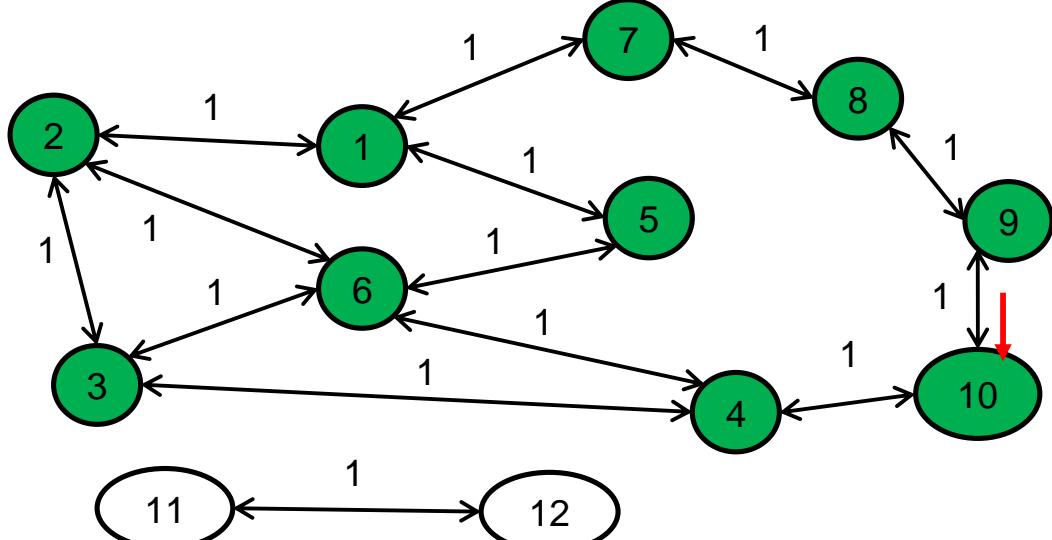
Cola	Nodo	10								
	d	4								



$u$	$d(u)$
1	0
2	1
3	2
4	3
5	1
6	2
7	1
8	2
9	3
10	4
11	$+\infty$
12	$+\infty$

# Breadth First Search (BFS)

Cola	Nodo										
	d										



$u$	$d(u)$
1	0
2	1
3	2
4	3
5	1
6	2
7	1
8	2
9	3
10	4
11	$+\infty$
12	$+\infty$

# Breadth First Search (BFS)

Los estados que describimos antes (visitado, en cola, no visitado) fueron puestos para una mejor visualización del algoritmo en el ejemplo.

Sin embargo, formulemos el algoritmo de una forma un poco distinta tal que se parezca a nuestro algoritmo genérico.

Cada vez que desencolemos un nodo  $u$  de la cola, en vez de meter a la cola los vecinos  $v$  que no estén visitados, revisemos si la arista  $u \rightarrow v$  está tensa. En caso lo esté, relajémosla y metamos a  $v$  a la cola. El pseudocódigo se puede ver a la derecha.

Esta versión se asemeja un poco más a nuestro **Algoritmo Genérico de Shortest Path**. Demostraremos que efectivamente se trata de un caso particular de este. También demostraremos que esta versión es equivalente al bfs que se usó en el ejemplo anterior (en donde cada nodo solo se visita una vez).

---

## Algorithm 6: BFS2

---

```
input :  $G(V_G, E_G)$  , source :  $s$ 
output:  $d[ ]$ ,  $parent[ ]$ 
1 Initialize ( $s$ )
2  $q = \emptyset$ 
3  $q.push(s)$ 
4 while  $q$  is not empty do
5    $u = q.pull()$ 
6   foreach edge  $u \rightarrow v$  in  $E_G$  do
7     if  $d(u) + 1 < d(v)$  then
8       // is tense
9        $d(v) = d(u) + 1$  // relax
10       $parent(v) = u$ 
11       $q.push(v)$ 
12    end
13  end
14 return  $d$ ,  $parent$ 
```

---

# BFS - Correctness

- **Teorema 4.1 (Correctness):** Si el algoritmo BFS termina, se debe cumplir que  $d(u) = \delta(u)$  para todo  $u \in V$  y el grafo de predecesores inducido por  $\text{parent}(u)$  es un shortest path tree.

## Demostración:

El algoritmo es un caso particular del **Modified Generic Shortest Path Algorithm** ya que solo se implementó la "lista" general como una cola. Y por el corolario del [lema 2.9](#) sabemos que el algoritmo es correcto. ■

- ❖ **Observación:** Si el algoritmo es usado en un grafo con pesos 1, no habrá ciclos negativos y el algoritmo siempre terminaría.

**Nota:** Para la demostración del teorema no hemos usado el hecho de que los pesos sean de tamaño 1 (simplemente usamos el lema 2.9 que es general para cualquier peso).

Por lo tanto, la correctitud de este algoritmo también es válido para **CUALQUIER** función de pesos  $w: E \rightarrow \mathbb{R}$  siempre y cuando no hayan ciclos negativos (obviamente reemplazando la condición de arista tensa por  $d(u) + w(u, v) < d(v)$  y la relajación por  $d(v) = d(u) + w(u, v)$ ).

El único problema es que una función de pesos general puede causar que el bfs corra en una complejidad mayor. Sin embargo, para el caso particular de pesos unitarios, los 2 siguientes lemas nos ayudarán a mostrar que la complejidad del BFS es lineal cuando todos los pesos son 1.



# BFS - Complexity

- **Lema 4.2:** Suponga que la cola del bfs en determinado momento tiene la secuencia de vértices  $v_0, v_1, \dots, v_k$ , donde  $v_0$  está al inicio de la cola y  $v_k$  al final. Entonces se cumple que  $d(v_i) \leq d(v_{i+1})$  para  $0 \leq i < k$  y que  $d(v_k) \leq d(v_0) + 1$ , es decir,  $d(v_k) = d(v_0)$  o  $d(v_k) = d(v_0) + 1$ .

## Demostración:

Queremos demostrar en todo momento los valores de  $d$  de los nodos presentes en la cola tiene alguna de estas estructuras :

$x$	$x$	$\dots$	$x$	$x$	$x$	$x$	$\dots$	$x$	$x + 1$	$\dots$	$x + 1$
-----	-----	---------	-----	-----	-----	-----	---------	-----	---------	---------	---------

Por inducción demostraremos que al **principio** de cada iteración del bucle while se cumple el lema:

- **Caso base:** En la primera iteración, solo el nodo  $s$  está en la cola, y se cumple el lema trivialmente
- **Paso inductivo:** Suponga que el lema cumple al inicio de la  $i$ -ésima iteración y que la cola en esa iteración es  $v_0, v_1, \dots, v_k$ . La primera operación será el pull del nodo  $v_0$  que será seguido de un número de operaciones push (podrían ser 0 push). Entonces pueden suceder 2 casos al finalizar la iteración:

**1) La cola queda vacía:** En este caso el algoritmo terminaría y no habría siguiente iteración.

**2) La cola no queda vacía:** Suponga que hubo  $p$  operaciones push. La cola sería  $v_1, \dots, v_k, v_{k+1}, \dots, v_{k+p}$  .

- ❖ Por hipótesis inductiva  $d(v_i) \leq d(v_{i+1})$ ,  $0 < i \leq k$  ... (1)
- ❖ Por las relajaciones del push  $d(v_i) = d(v_0) + 1$ ,  $k < i \leq k + p \Rightarrow d(v_i) \leq d(v_{i+1})$ ,  $k < i \leq k + p$  ... (2)
- ❖ Por hipótesis inductiva  $d(v_k) \leq d(v_0) + 1 \Rightarrow d(v_k) \leq d(v_{k+1})$  ... (3)
- ❖ Por hipótesis inductiva  $d(v_0) + 1 \leq d(v_1) + 1 \Rightarrow d(v_i) \leq d(v_1)$ ,  $k < i \leq k + p$  ... (4)  $\Rightarrow d(v_{k+p}) \leq d(v_0)$  ... (5)

(1), (2), (3) y (5) demuestran que el lema también cumplirá al inicio de la iteración ( $i + 1$ ). En particular la ecuación (4) demuestra que el lema cumplía también **durante** la iteración  $i$  . ■

# BFS - Complexity

- **Lema 4.3 :** Sea  $D$  el arreglo en donde  $D_i$  es el valor de  $d(u)$  que le fue asignado al nodo  $u$  que ingresó a la cola del bfs en el  $i$  – ésmo push. Entonces el arreglo  $D$  es no decreciente.

## Demostración:

Por inducción en el número de operaciones push.

- **Caso base:** Luego del primer push, entra el nodo  $s$  con  $d(s) = 0$  y el arreglo  $D = \{d(s) = 0\}$  es no decreciente trivialmente.
- **Paso inductivo:** Suponga que en el  $i$  – ésmo push se cumple que  $D$  es no decreciente. Suponga que el nodo pusheado en ese push es  $u$ . Queremos demostrar que el siguiente nodo pusheado (si es que existe) tendrá un estimado mayor o igual que  $d(u)$ . Suponga que la cola luego de pushear  $u$  es  $v_1, v_2, \dots, v_k = u$ . A continuación se procederá a hacer pullear el nodo  $v_1$  pero quizás este nodo no pushee a ningún nodo. Entonces definamos a  $v_i$  como el primer nodo de esa lista de esa lista en pushear el siguiente nodo  $x$  (en caso no exista, no habrá siguiente push y el algoritmo termina). Justo antes de sacar a  $v_i$  la cola será  $v_i, v_{i+1}, \dots, v_k = u$ . Por el **lema 4.2** sabemos que  $d(v_k = u) \leq d(v_i) + 1$ . El siguiente nodo pusheado  $x$  tendrá  $d(x) = d(v_i) + 1$ . Por lo tanto  $d(u) \leq d(x)$  y se seguirá cumpliendo que  $D$  es no decreciente. ■

# BFS - Complexity

- **Lema 4.4 :** Cada nodo entra a lo más una vez a la cola del bfs.

## Demostración:

Cada push que se hace a un nodo  $v$  en la cola, está acompañado de una relajación de una arista tensa  $(u, v)$ . Cada relajación de una arista  $(u, v)$  decrece el valor de  $d(v)$ .

Por contradicción, suponga que un nodo  $v$  ha ingresado más de 1 vez en la cola. La 2da vez que entre, su  $d(v)$  será menor que la primera vez que ingresó. Pero al cumplirse eso, entonces el arreglo  $D$  definido en el **lema 4.3** ya no sería no decreciente, llegando a una contradicción. ■

- **Corolario:** La complejidad del bfs es  $O(V + E)$  (habrán  $O(V)$  pulls y  $O(E)$  inspecciones para un push)

**Nota:** El lema implica que en el algoritmo ya no es necesario verificar que  $d(u) + 1 < d(v)$  para insertarlo en la cola. Basta con verificar que  $d(u) = +\infty$  (que significa que todavía no ha entrado en la cola).

**Nota2:** Generalmente se cumple que  $V < E$  por lo que también se suele poner la complejidad como  $O(E)$ .

# BFS - Code

La implementación es la misma para grafos dirigidos como para no dirigidos. Si el grafo es no dirigido simplemente se deberá transformar una arista  $u - v$  en dos aristas  $u \rightarrow v$ ,  $v \rightarrow u$ .

Podemos usar la estructura *queue* de C++, pero también podemos usar *deque* con operaciones *push\_back()* y *pop\_front()*

```
const Long INF = 1e18;
const Long MX = 1e5;

struct Graph{
    Long d[MX];
    Long parent[MX];
    vector<Long> adj[MX];

    void clear(Long n) {
        for (int i = 0; i < n; i++) {
            adj[i].clear();
        }
    }

    void addEdge (Long u, Long v) {
        adj[u].push_back(v);
        //adj[v].push_back(u); //Undirected graph
    }
}
```

```
void bfs(Long s, Long n) { //O(V + E)
    for (Long i = 0; i < n; i++) {
        d[i] = INF;
        parent[i] = -1;
    }
    d[s] = 0;
    deque<Long> q;
    q.push_back(s);
    while (!q.empty()) {
        Long u = q.front();
        q.pop_front();
        for (Long v : adj[u]) {
            if (d[v] == INF) { //d[u] + 1 < d[v]
                d[v] = d[u] + 1; //relax
                parent[v] = u;
                q.push_back(v);
            }
        }
    }
};
```

# BFS – General weights

También podemos utilizar la versión del bfs que funciona para cualquier función de pesos. Sin embargo recuerde que será complejidad exponencial para una función de pesos general (**¿o quizás menos?**, ¡lo veremos luego!). Note que si los pesos siguen siendo 1, la complejidad sí es lineal porque hace lo mismo que el código anterior.

```
struct Graph{
    Long d[MX];
    Long parent[MX];
    vector<pair<Long, Long>> adj [MX];

    void addEdge(Long u, Long v, Long w) {
        adj [u].push_back({v, w});
    }
}
```

```
void bfs(Long s, Long n) {
    for (Long i = 0; i < n; i++) {
        d[i] = INF;
        parent[i] = -1;
    }
    d[s] = 0;
    deque<Long> q;
    q.push_back(s);
    while (!q.empty()) {
        Long u = q.front();
        q.pop_front();
        for (auto e : adj[u]) {
            Long v = e.first;
            Long w = e.second;
            if (d[u] + w < d[v]) {
                d[v] = d[u] + w;
                parent[v] = u;
                q.push_back(v);
            }
        }
    }
}
```



(for general weights)

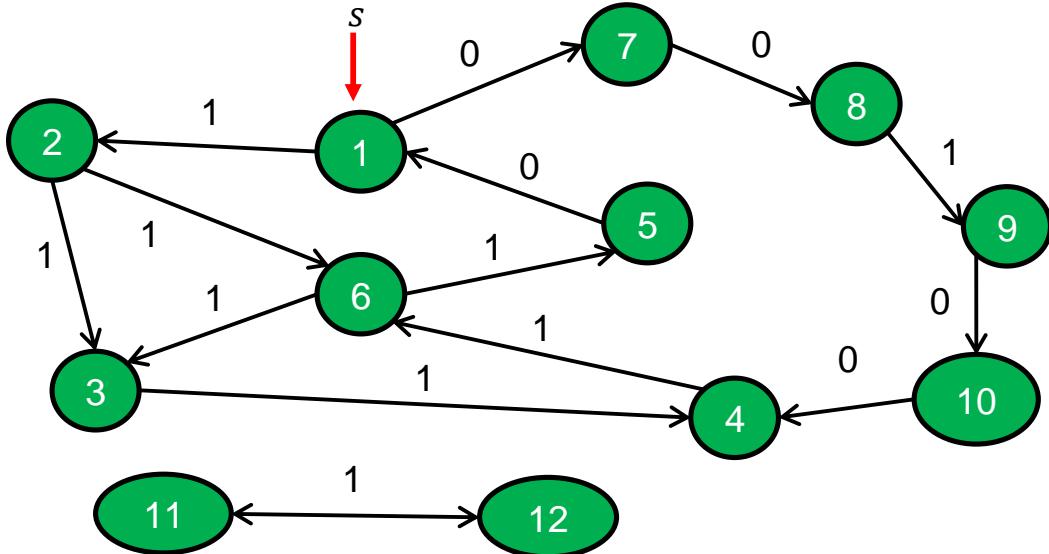


Es una herramienta sorpresa que nos servirá más tarde

# Single Source Shortest Path – 0-1 Weights

**Problema:** Sea un grafo dirigido con pesos  $G(V, E, w)$ . Cuya función de pesos es  $w: E(G) \rightarrow \{0, 1\}$ . Es decir todas las aristas tienen peso 0 o 1.

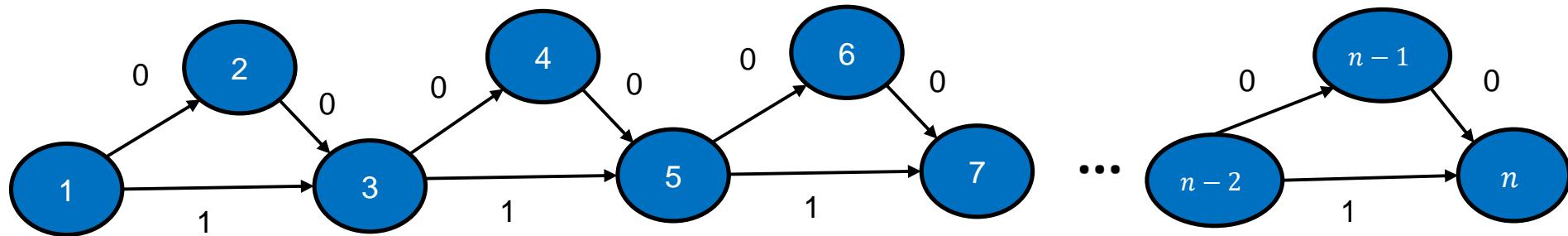
Se pide hallar la longitud del shortest path desde un nodo fuente  $s$  hasta cualquier otro nodo  $u$  (distancia del nodo  $u$ ), denotado como  $\delta(u)$



$u$	$\delta(u)$
1	0
2	1
3	2
4	1
5	3
6	2
7	0
8	0
9	1
10	1
11	$+\infty$
12	$+\infty$

# Single Source Shortest Path – 0-1 Weights

Como el cambio en los pesos ha sido pequeño, puede ser tentador utilizar la generalización del bfs que hicimos para cualquier función de pesos reales y rezar por que la complejidad no sea exponencial. Aunque no es el caso que sea exponencial, podemos construir este caso:



- El nodo 1 entrará 1 vez en la cola
- El nodo 3 entrará 2 veces en la cola
- El nodo 5 entrará 3 veces en la cola
- El nodo 7 entrará 4 veces en la cola
- El nodo  $n$  entrará  $\frac{n+1}{2}$  veces en la cola

Este caso fuerza al bfs a hacer  $\Omega(n^2)$  relajaciones. Además se puede demostrar que el peor caso para **el número de relajaciones** de este algoritmo debe ser  $O(n^2)$  ya que se vio anteriormente que si los pesos son enteros el número de relajaciones es  $O(n^2C)$ . En este caso  $C = 1$ . Ojo que la complejidad total podría ser incluso un poco mayor. Probablemente  $O(n^2m)$

# Single Source Shortest Path – 0-1 Weights

¿Podemos conseguir algo mejor que  $O(n^2)$ ? ¿Cómo es que el bfs era  $O(n + m)$  para aristas con peso unitario? ( $n$  nodos y  $m$  aristas)

Evidentemente es por lema 4.2. En particular el hecho de que en la cola del bfs los valores de  $d$  sean una secuencia no decreciente formando algunas de estas estructuras:

$x$	$x$	$\dots$	$x$	$x$	$x$	$x$	$\dots$	$x + 1$	$x + 1$
-----	-----	---------	-----	-----	-----	-----	---------	---------	---------

Cuando llegan las aristas cero esto falla, imagina que estamos en el caso 2

$x$	$x$	$\dots$	$x + 1$	$x + 1$
-----	-----	---------	---------	---------

Sacamos el primer nodo con  $d(u) = x$  y al relajar la arista de peso 0 meteremos un nodo con  $d(v) = x + 0 = x$

Y quedará

$x$	$x$	$\dots$	$x + 1$	$x + 1$	$x$
-----	-----	---------	---------	---------	-----

Lo cual obviamente rompe la invariante de la estructura.

# BFS-01

Pero, ¿y si metemos a la arista 0 al principio de la cola?

Podemos utilizar un **deque** para simular eso.

Este algoritmo también es llamado bfs-01

---

## Algorithm 7: BFS-01

---

```
input :  $G(V_G, E_G, w)$  , source :  $s$ 
output:  $d[ ]$ ,  $parent[ ]$ 
1 Initialize ( $s$ )
2  $q = \emptyset$ 
3  $q.push(s)$ 
4 while  $q$  is not empty do
5    $u = q.pop\_front()$ 
6   foreach edge  $u \rightarrow v$  in  $E_G$  do
7     if  $d(u) + w(u, v) < d(v)$  then
        // is tense
         $d(v) = d(u) + w(u, v)$  // relax
         $parent(v) = u$ 
        if  $w(u, v) == 0$  then
          |  $q.push\_front(v)$ 
        else
          |  $q.push\_back(v)$ 
        end
      end
    end
  end
17 end
18 return  $d, parent$ 
```

---

# BFS-01

## Correctness y complejidad del BFS-01:

Aunque no haremos una demostración de la correctitud ni la complejidad, es fácil ver que el **lema 4.1** se sigue cumpliendo aquí, porque para ese lema no hay una diferenciación entre hacer un `push_back` o un `push_front`, igual el nodo siempre entra en la cola y eso es lo que importa (ya que está basado en el lema 2.9 que se hizo para un contenedor general). Eso demostraría correctness.

Además el **lema 4.2** también se pueden demostrar de una manera similar a como se demostró para el bfs normal (el análisis de las operaciones `push_front` es muy similar, para las otras operaciones el análisis es el mismo). Lamentablemente el **lema 4.3 y 4.4** ya no se cumplirían exactamente. Sin embargo, de todas formas se puede demostrar que cada nodo puede entrar a lo más 2 veces en la cola del bfs01. Y si entra 2 veces, en la 2da vez ya no relaja a nadie (aunque igual desperdiciarías un 2do for inspeccionando a sus vecinos, a menos que lo cortes con un arreglo de visitados).

La complejidad sería  $O(V + 2E) = O(V + E)$

# BFS-01 Code

```
const Long MX = 1e5;
const Long INF = 1e18;

struct Graph{
    Long d[MX];
    vector<pair<Long, Long>> adj [MX];

    void clear(Long n) {
        for (Long i = 0; i < n; i++) {
            adj[i].clear();
        }
    }

    void addEdge(Long u , Long v, Long w) {
        adj[u].push_back({v, w});
        //adj[v].push_back({u, w}); //Undirected graph
    }
}
```

```
void bfs01(Long s, Long n){ //O(V + E)
    for(Long i = 0; i < n; i++){
        d[i] = INF;
    }
    d[s] = 0;
    deque<Long> q;
    q.push_back(s);
    while(!q.empty()) {
        Long u = q.front();
        q.pop_front();
        for (auto e : adj[u] ) {
            Long v = e.first;
            Long w = e.second;
            if(d[u] + w < d[v]) {
                d[v] = d[u] + w;
                if(w == 0) q.push_front(v);
                else q.push_back(v);
            }
        }
    }
}
```

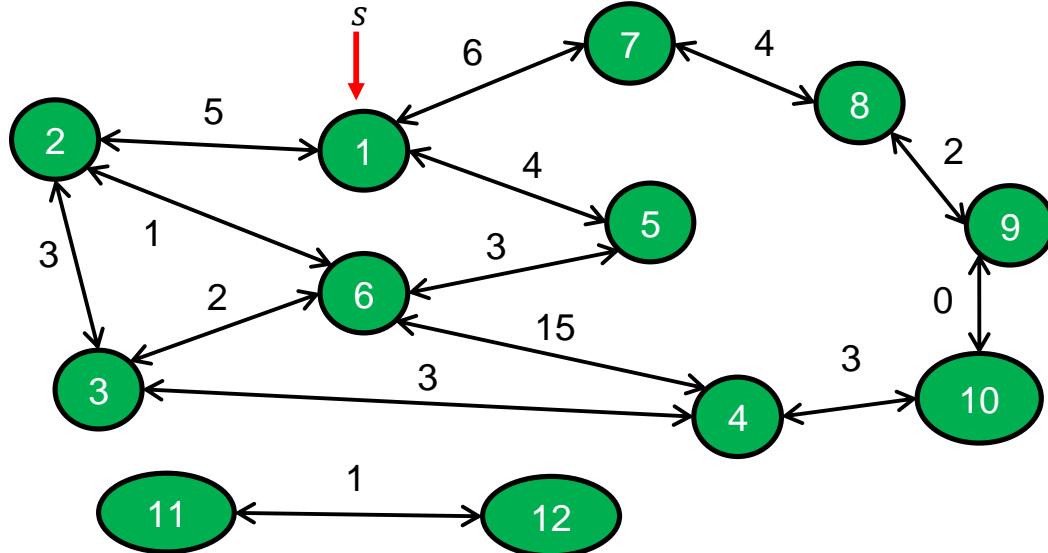
# Contenido

1. Introducción	
2. Propiedades de los Shortest Path y Algoritmo Genérico	
3. Single Source Shortest Path en DAG	
4. Single Source Shortest Path con pesos unitarios	
<b>5. Single Source Shortest Path con pesos no negativos</b>	
6. Single Source Shortest Path con cualquier peso real	
7. All-pairs shortest path	

# Single Source Shortest Path – Non Negative Weights

**Problema:** Sea un grafo dirigido con pesos  $G(V, E, w)$ . Cuya función de pesos es  $w: E(G) \rightarrow \mathbb{R}^+ \cup \{0\}$ . Es decir todas las aristas tienen peso no negativos (positivos o cero).

Se pide hallar la longitud del shortest path desde un nodo fuente  $s$  hasta cualquier otro nodo  $u$  (distancia del nodo  $u$ ), denotado como  $\delta(u)$



$u$	$\delta(u)$
1	0
2	5
3	8
4	11
5	4
6	6
7	6
8	10
9	12
10	12
11	$+\infty$
12	$+\infty$

# Single Source Shortest Path – Non Negative Weights

Nuevamente puede ser tentador utilizar nuestro código de bfs general que hemos hecho ¡y funcionará! pero la complejidad no es tan buena (¡pronto llegaremos a eso! **Spoiler:** no es exponencial pero tampoco es  $O(V + E)$ ). Sin embargo, afortunadamente hay un algoritmo más potente para este caso de aristas con pesos positivos (o cero). Y somos afortunados porque este es el caso más “común” tanto en la vida real como en los problemas de programación competitiva (una arista de peso negativo es un tanto artificial, pero sí existen aplicaciones).

En 1957 Leyzorek, Gray, Johnson, Ladew, Meaker, Petry y Seitz describieron un algoritmo que solucionaba el problema de shortest path para grafos con pesos no negativos en un reporte anual de un proyecto de “Case Institute of Technology”.

El mismo algoritmo fue concebido por Edsger Dijkstra en 1956 pero recién publicado en 1959 y también por otros autores como George Minty cerca de 1960 y también por Peter Whiting y John Hillier en 1960.

El algoritmo es conocido como **Dijkstra's Algorithm**.

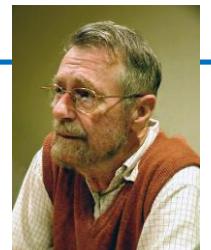


# Edsger Dijkstra

Edsger Dijkstra fue un computer scientist holandés. Presentó su algoritmo en una demostración de la funcionalidad de una nueva computadora llamada ARMAC en 1956, mostrando cómo se podría usar para hallar el shortest path entre 2 ciudades de Holanda en un pequeño mapa creado por él. Específicamente, dijo lo siguiente en una entrevista:



“What’s the shortest way to travel from Rotterdam to Groningen? It is the algorithm for the shortest path, which I designed in about 20 minutes. One morning I was shopping in Amsterdam with my young fiancée, and tired, we sat down on the café terrace to drink a cup of coffee and I was just thinking about whether I could do this, and I then designed the algorithm for the shortest path. As I said, it was a 20-minute invention. In fact, it was published in 1959, three years later. The publication is still quite nice. One of the reasons that it is so nice was that I designed it without pencil and paper. Without pencil and paper you are almost forced to avoid all avoidable complexities. Eventually that algorithm became, to my great amazement, one of the cornerstones of my fame.”



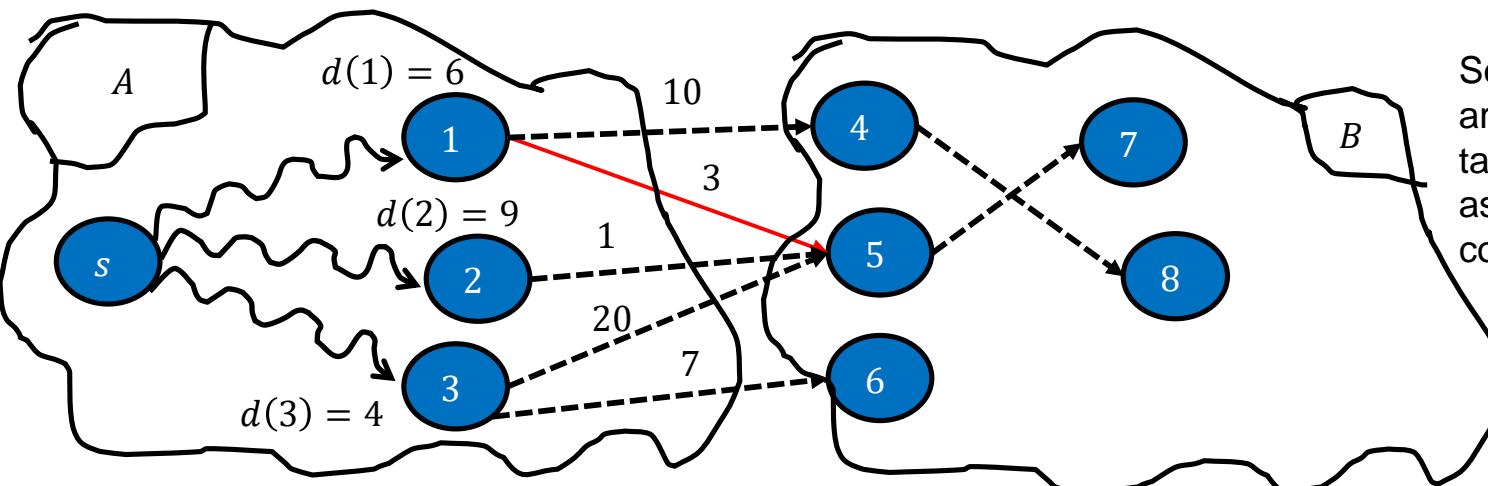
Edsger Dijkstra

En 1959 publica “**A Note on Two Problems in Connexion with Graphs**” en donde resuelve el problema del MST (describiendo el algoritmo de Jarnik-Prim que conoces) y también el problema del shortest path con el algoritmo que se describirá en la siguiente diapo.

# Dijkstra's Algorithm

El algoritmo de Dijkstra es un algoritmo **greedy** que trata de tomar el camino de menor peso en cada iteración. Dijkstra en su publicación original lo planteó de una forma similar a la siguiente:

Supongamos que hemos calculado hasta el momento los shortest path (y sus distancias) de ciertos nodos. Llamemos a este conjunto  $A$  y al conjunto de los nodos que todavía quedan por calcular, como conjunto  $B$ . En el grupo  $B$  deberán existir algunos nodos  $v$  tal que existe la arista  $u \rightarrow v$  tal que  $u \in A$  (en caso no existan, significa que todos los nodos restantes en  $B$  son inalcanzables y el algoritmo terminaría). Entre todas las aristas de ese tipo, escoge la arista  $u \rightarrow v$  que genere el camino con **menor peso total** y actualiza la distancia de  $v$  con ese peso. Por ejemplo, suponga que en algún momento del algoritmo se tiene este caso:



Se elegiría como siguiente arista a la  $1 \rightarrow 5$  con tamaño total 9 y se asignaría la distancia de 5 como 9.

# Dijkstra's Algorithm

¿Cómo podríamos implementar eso? Bueno Dijkstra en su publicación original lo hizo con un algoritmo  $O(V^2)$  pero lo plantearemos de una forma similar. Podemos simular esa selección del mejor path, poniendo todos los paths potenciales en una lista ordenada.

La lista ordenada mantendrá los paths ordenados según su peso.

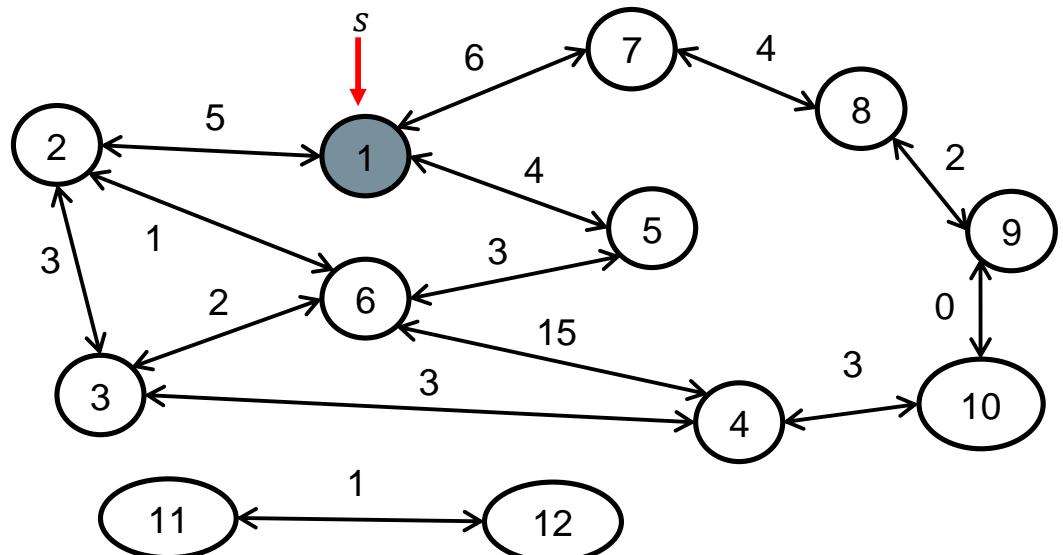
En cada iteración sacaremos el path de peso mínimo de la lista e inspeccionaremos el último nodo del path. Si este nodo está siendo visitado por primera vez lo marcaremos como visitado, lo que significa que su **estimado ya es correcto y es igual a la longitud del path extraído** por lo que analizaremos todos sus vecinos que formen una arista tensa y la relajaremos, añadiendo un nuevo path a la cola. En caso este nodo extraído ya haya sido visitado antes, no tiene sentido procesarlo, porque significa que ya hemos encontrado un mejor camino de  $s$  a  $u$  antes.



# Dijkstra's Algorithm

Cola

1

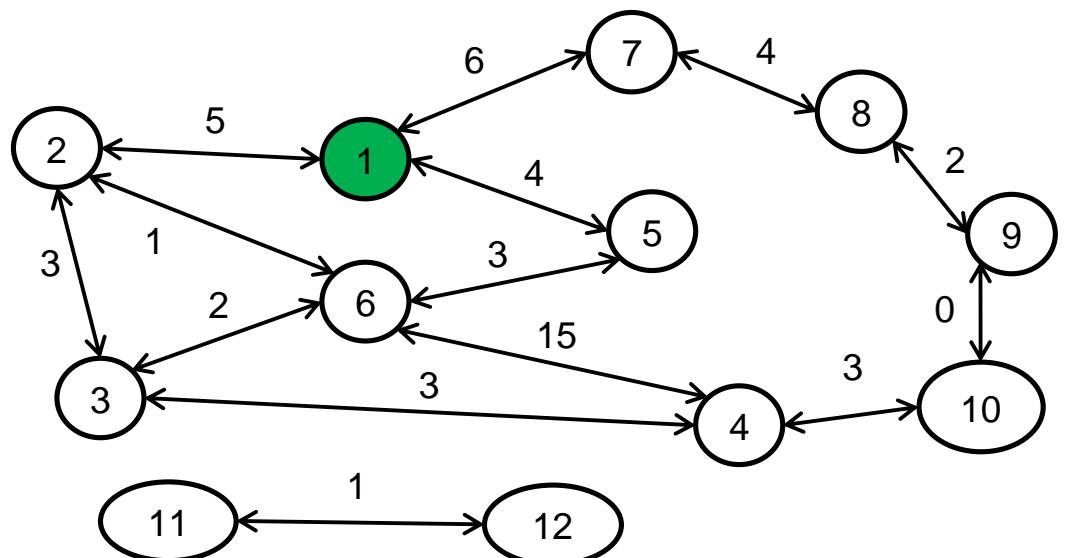


Path actual

$u$	$d(u)$	$parent(u)$
1	0	NULL
2	$+\infty$	NULL
3	$+\infty$	NULL
4	$+\infty$	NULL
5	$+\infty$	NULL
6	$+\infty$	NULL
7	$+\infty$	NULL
8	$+\infty$	NULL
9	$+\infty$	NULL
10	$+\infty$	NULL
11	$+\infty$	NULL
12	$+\infty$	NULL

# Dijkstra's Algorithm

Cola

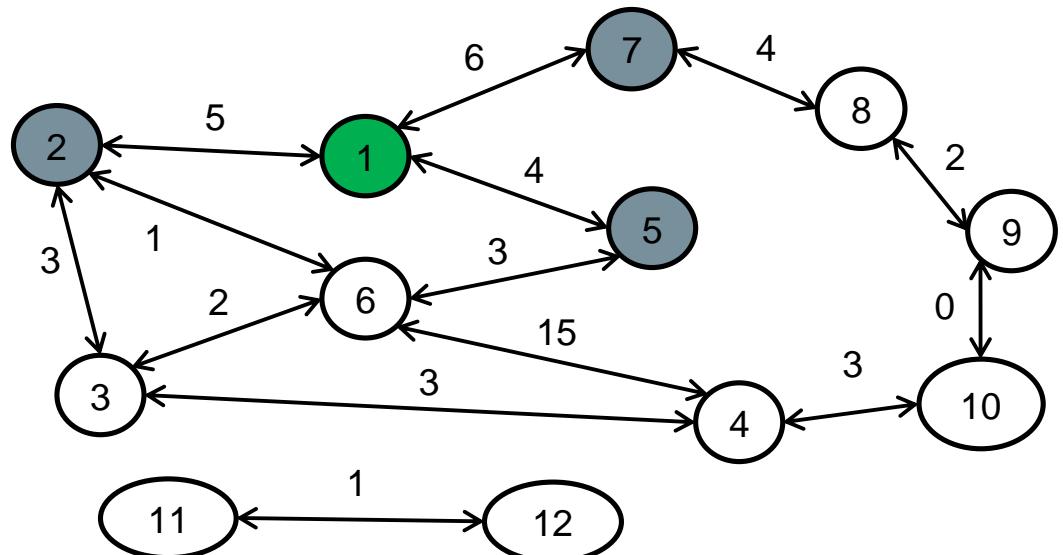
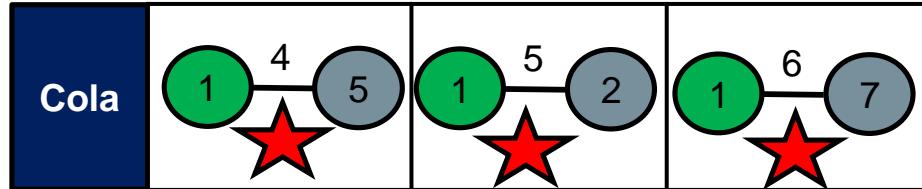


Path actual



$u$	$d(u)$	$parent(u)$
1	0	NULL
2	$+\infty$	NULL
3	$+\infty$	NULL
4	$+\infty$	NULL
5	$+\infty$	NULL
6	$+\infty$	NULL
7	$+\infty$	NULL
8	$+\infty$	NULL
9	$+\infty$	NULL
10	$+\infty$	NULL
11	$+\infty$	NULL
12	$+\infty$	NULL

# Dijkstra's Algorithm

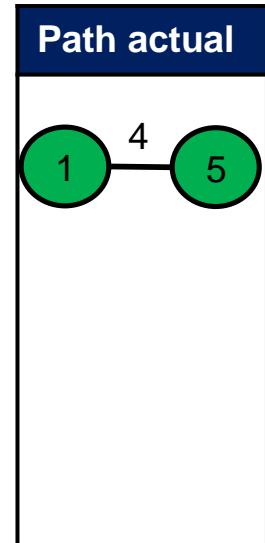
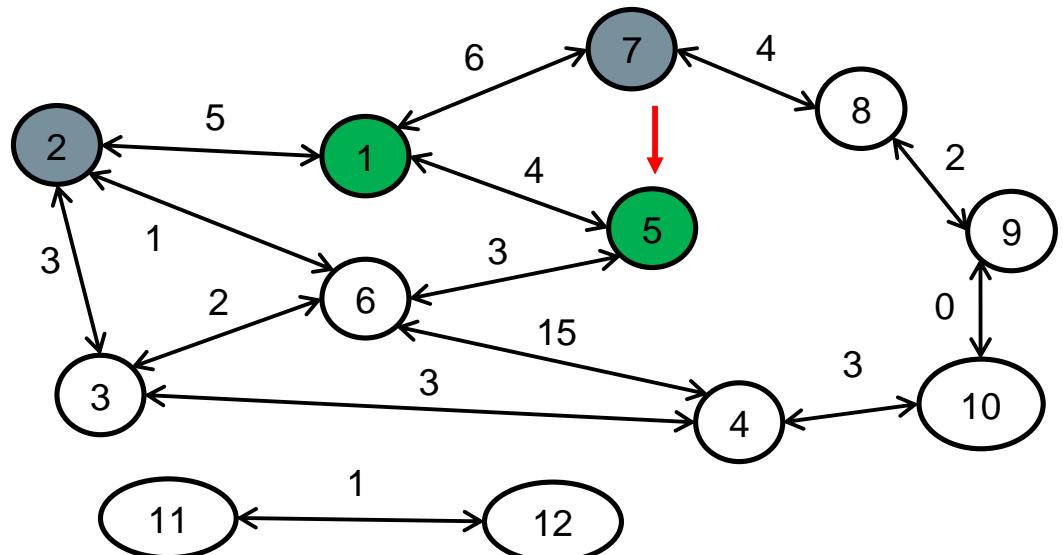
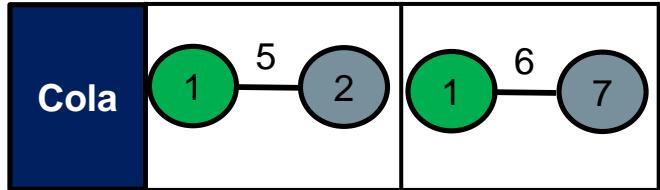


Path actual

Path actual

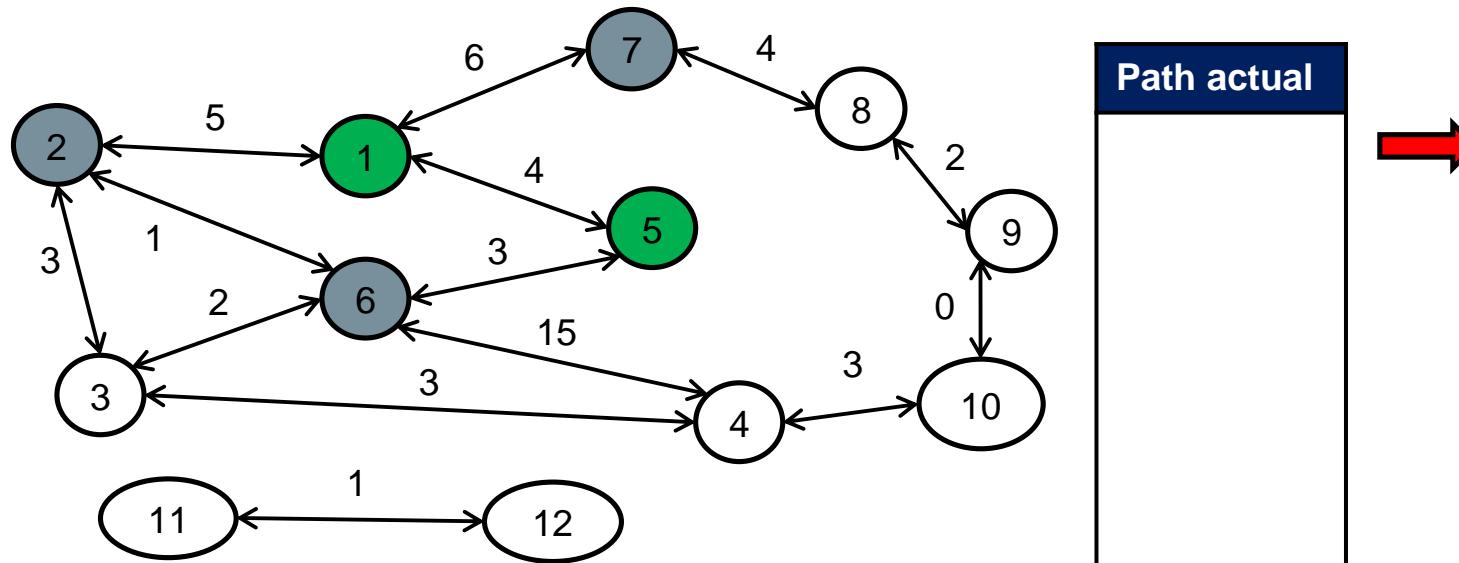
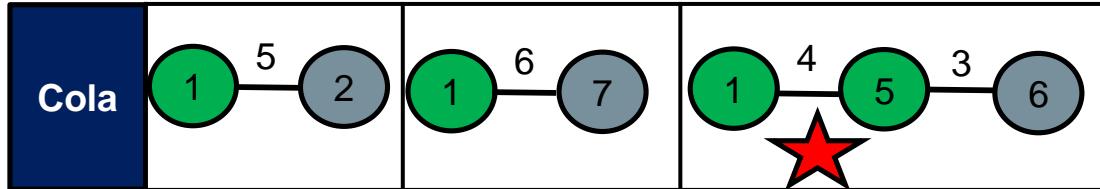
<i>u</i>	<i>d(u)</i>	<i>parent(u)</i>
1	0	NULL
2	5	1
3	$+\infty$	NULL
4	$+\infty$	NULL
5	4	1
6	$+\infty$	NULL
7	6	1
8	$+\infty$	NULL
9	$+\infty$	NULL
10	$+\infty$	NULL
11	$+\infty$	NULL
12	$+\infty$	NULL

# Dijkstra's Algorithm



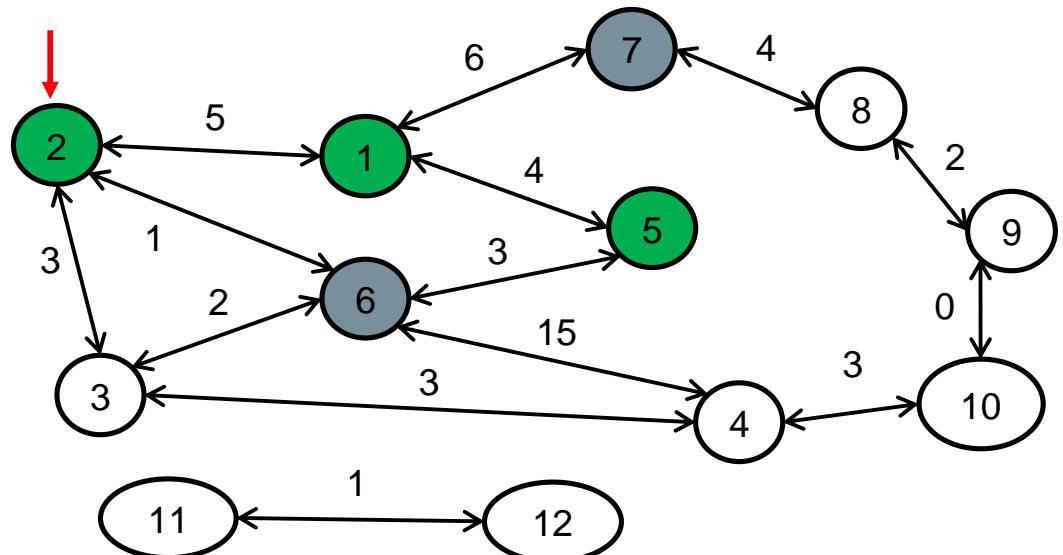
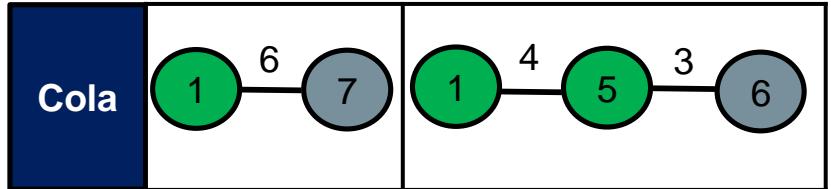
$u$	$d(u)$	$parent(u)$
1	0	NULL
2	5	1
3	$+\infty$	NULL
4	$+\infty$	NULL
5	4	1
6	$+\infty$	NULL
7	6	1
8	$+\infty$	NULL
9	$+\infty$	NULL
10	$+\infty$	NULL
11	$+\infty$	NULL
12	$+\infty$	NULL

# Dijkstra's Algorithm



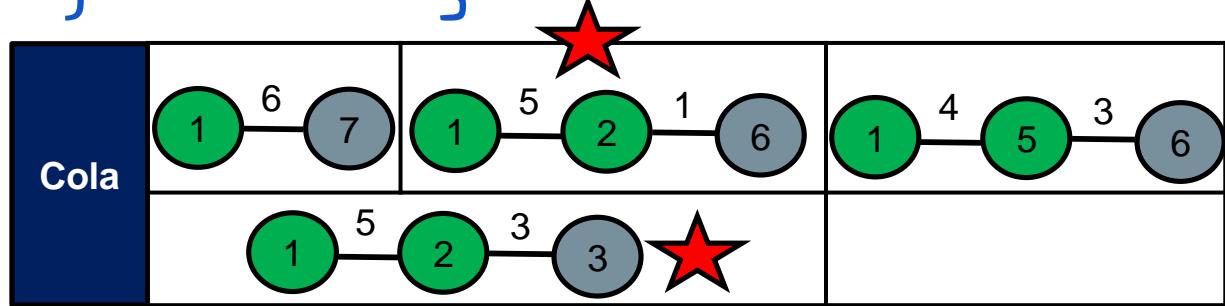
$u$	$d(u)$	$parent(u)$
1	0	NULL
2	5	1
3	$+\infty$	NULL
4	$+\infty$	NULL
5	4	1
6	7	5
7	6	1
8	$+\infty$	NULL
9	$+\infty$	NULL
10	$+\infty$	NULL
11	$+\infty$	NULL
12	$+\infty$	NULL

# Dijkstra's Algorithm



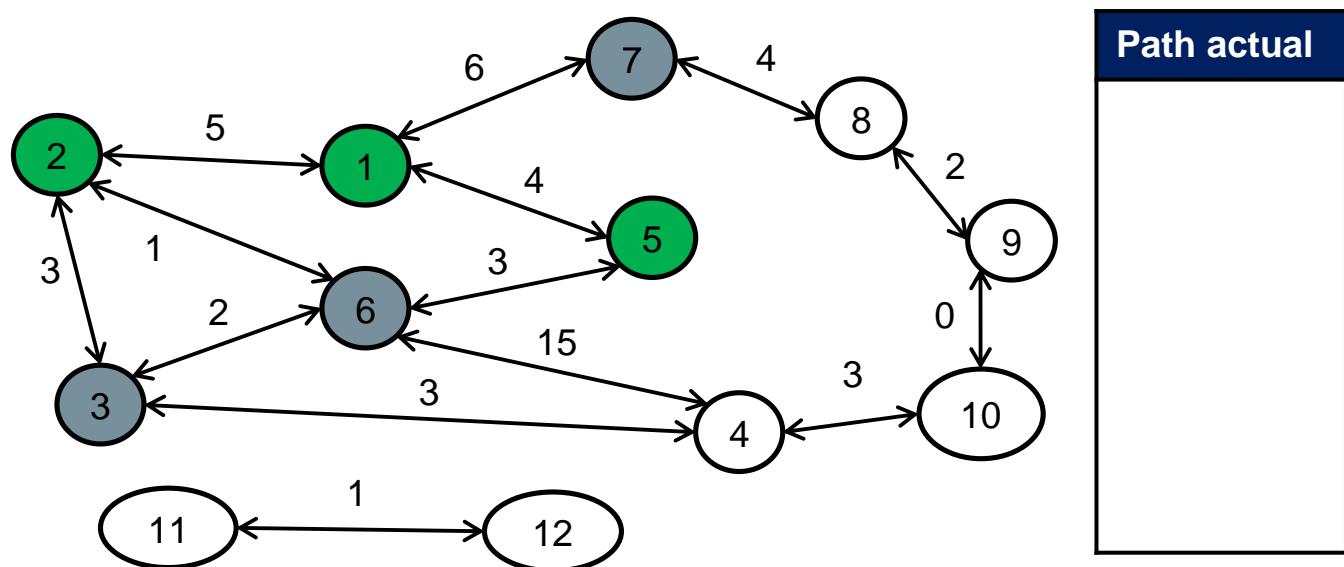
Path actual		
$u$	$d(u)$	$parent(u)$
1	0	NULL
2	5	1
3	$+\infty$	NULL
4	$+\infty$	NULL
5	4	1
6	7	5
7	6	1
8	$+\infty$	NULL
9	$+\infty$	NULL
10	$+\infty$	NULL
11	$+\infty$	NULL
12	$+\infty$	NULL

# Dijkstra's Algorithm

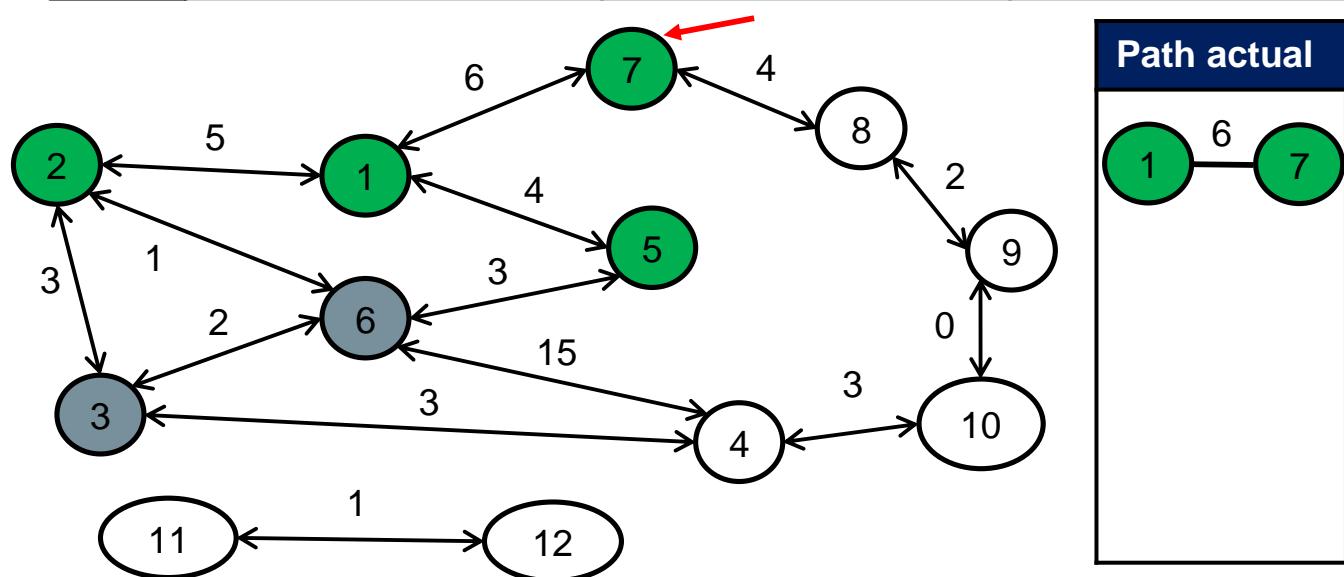
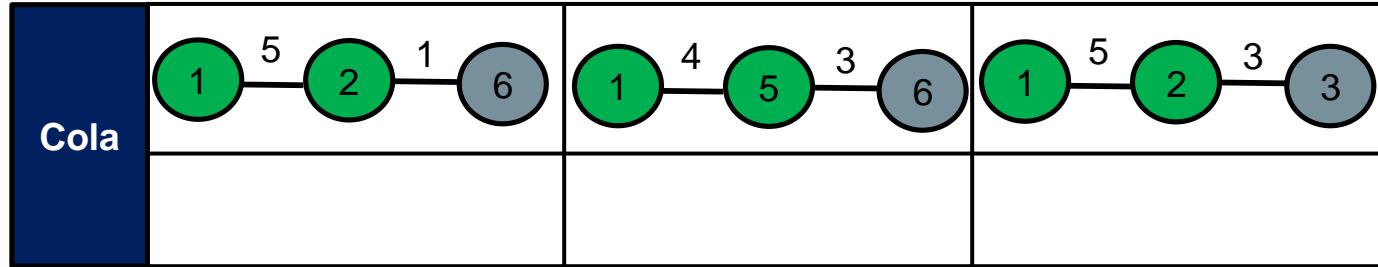


Red arrows point from the bottom row of the grid to the first two rows of the table.

<i>u</i>	<i>d(u)</i>	<i>parent(u)</i>
1	0	NULL
2	5	1
3	8	2
4	$+\infty$	NULL
5	4	1
6	6	2
7	6	1
8	$+\infty$	NULL
9	$+\infty$	NULL
10	$+\infty$	NULL
11	$+\infty$	NULL
12	$+\infty$	NULL

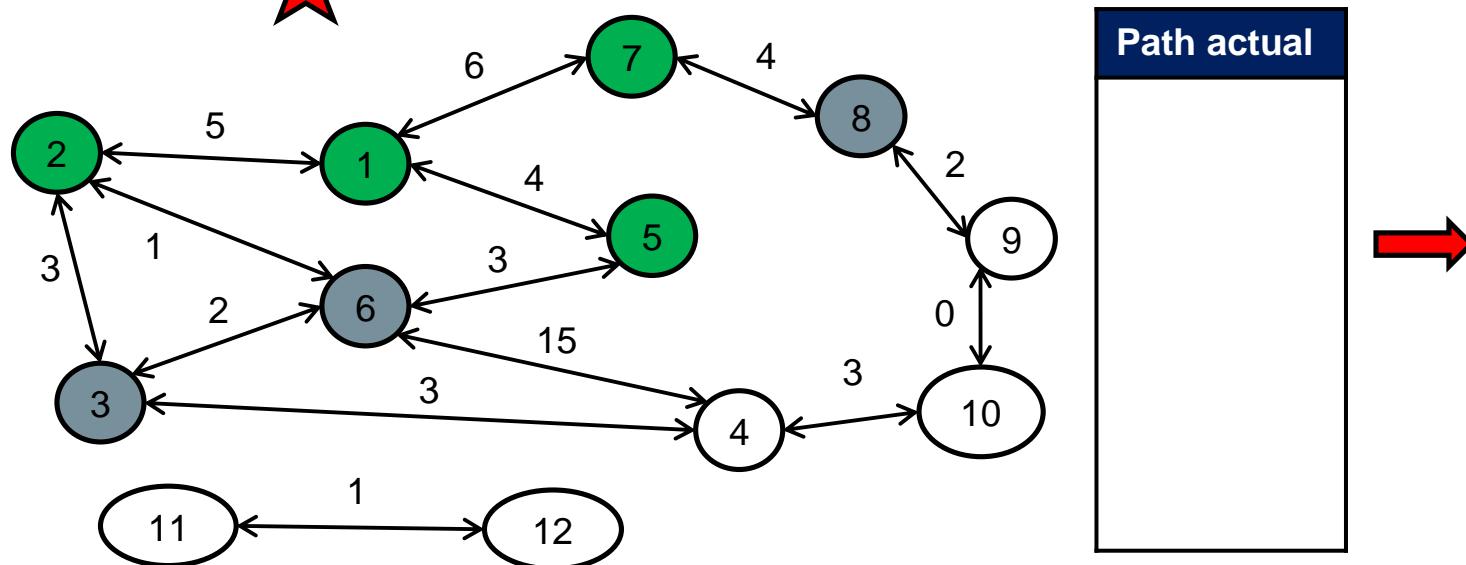
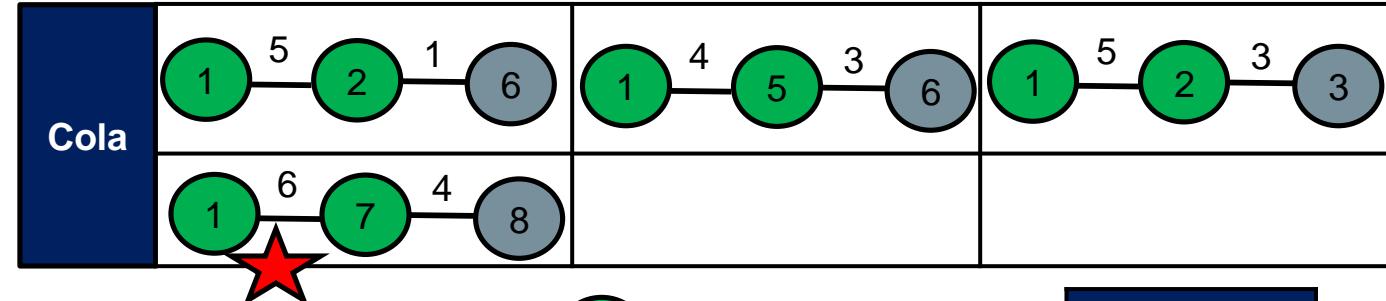


# Dijkstra's Algorithm



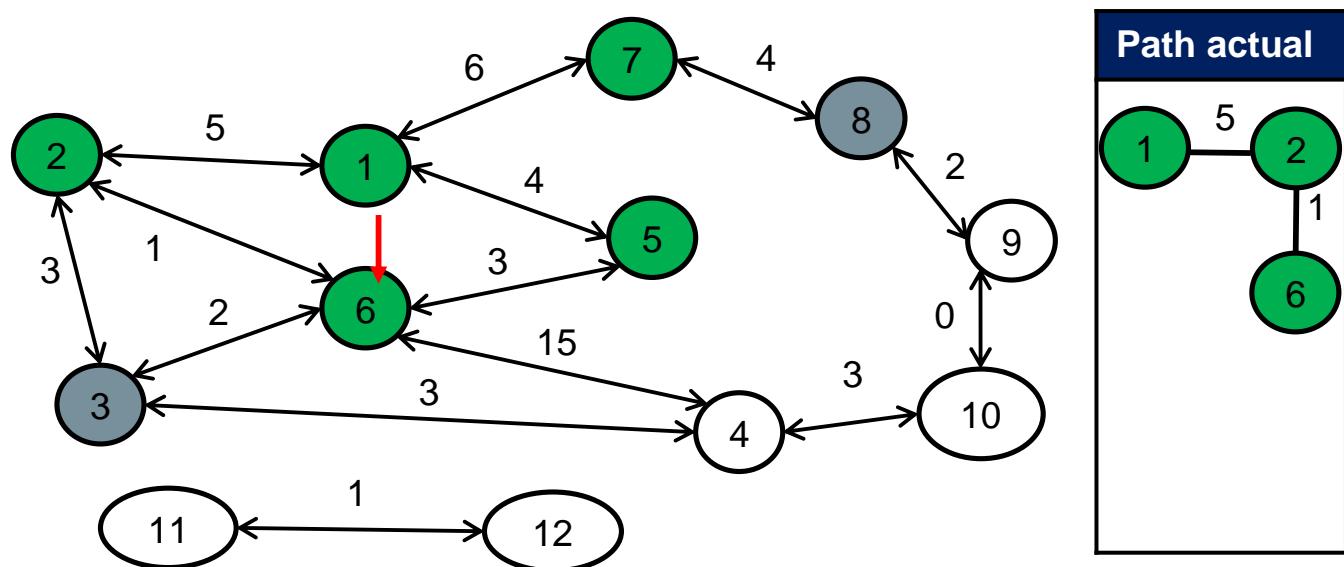
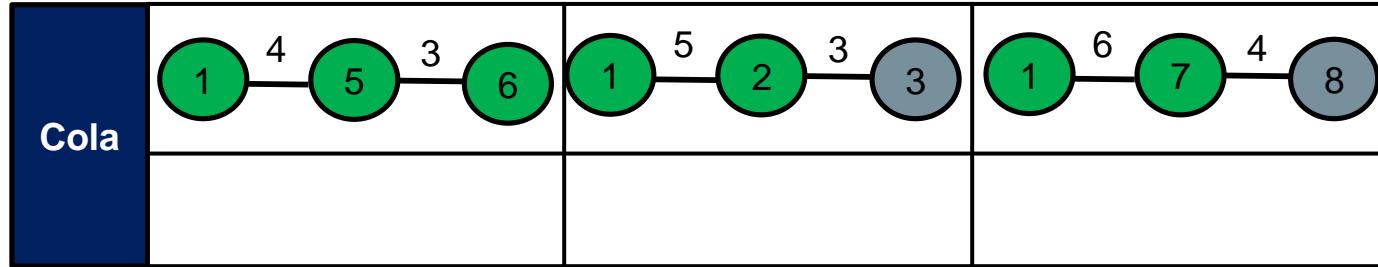
<i>u</i>	<i>d(u)</i>	<i>parent(u)</i>
1	0	NULL
2	5	1
3	8	2
4	$+\infty$	NULL
5	4	1
6	6	2
7	6	1
8	$+\infty$	NULL
9	$+\infty$	NULL
10	$+\infty$	NULL
11	$+\infty$	NULL
12	$+\infty$	NULL

# Dijkstra's Algorithm



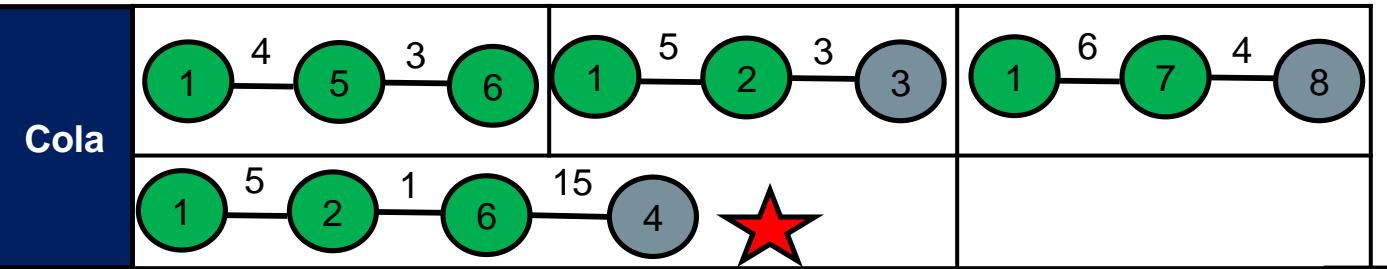
<i>u</i>	<i>d(u)</i>	<i>parent(u)</i>
1	0	NULL
2	5	1
3	8	2
4	$+\infty$	NULL
5	4	1
6	6	2
7	6	1
8	10	7
9	$+\infty$	NULL
10	$+\infty$	NULL
11	$+\infty$	NULL
12	$+\infty$	NULL

# Dijkstra's Algorithm

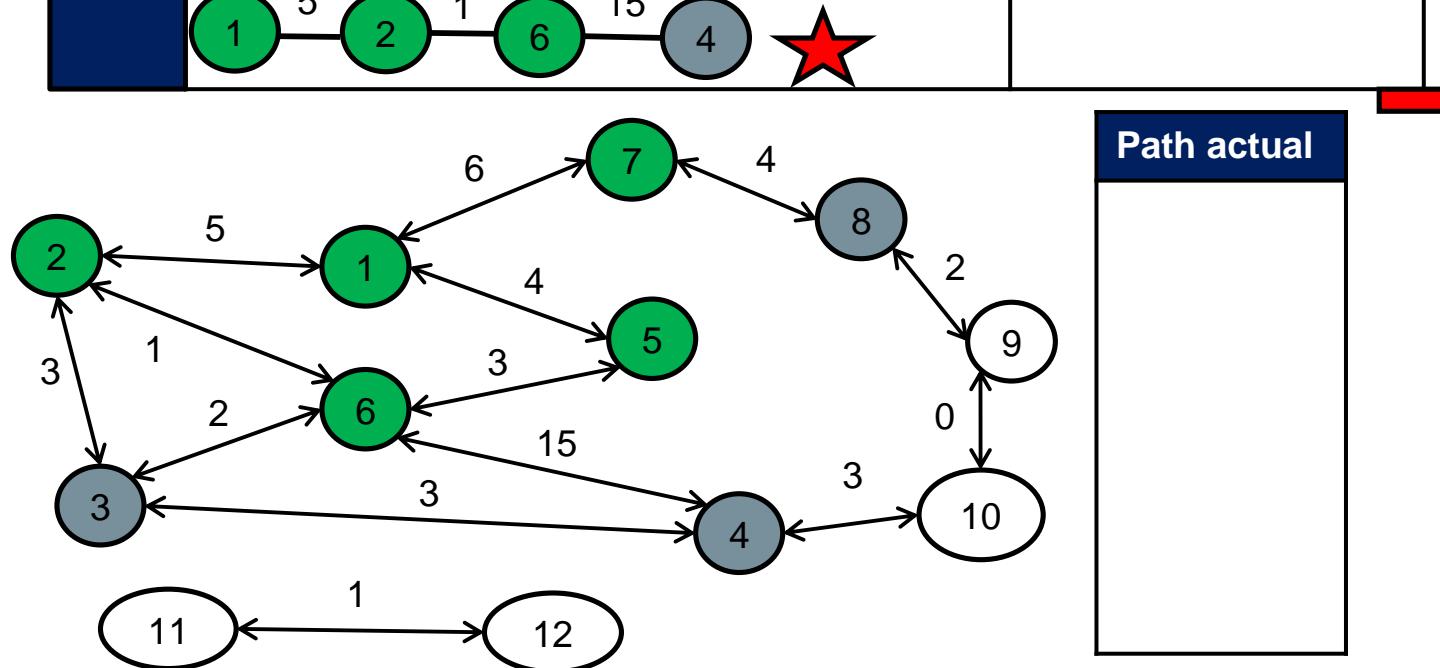


<i>u</i>	<i>d(u)</i>	<i>parent(u)</i>
1	0	NULL
2	5	1
3	8	2
4	$+\infty$	NULL
5	4	1
6	6	2
7	6	1
8	10	7
9	$+\infty$	NULL
10	$+\infty$	NULL
11	$+\infty$	NULL
12	$+\infty$	NULL

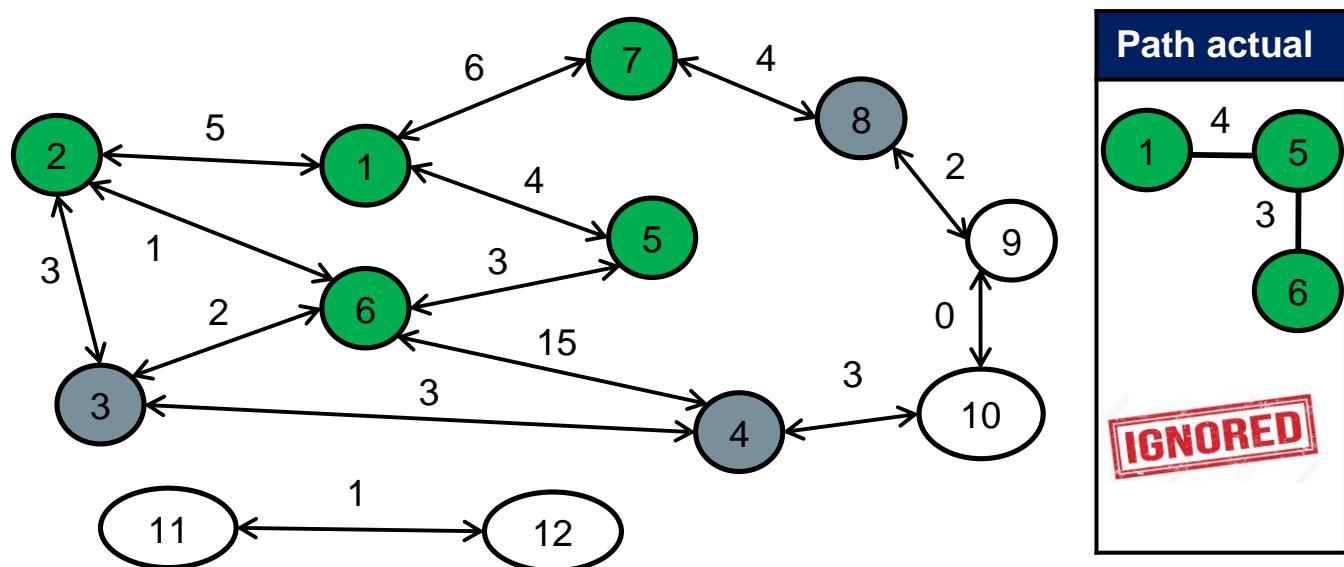
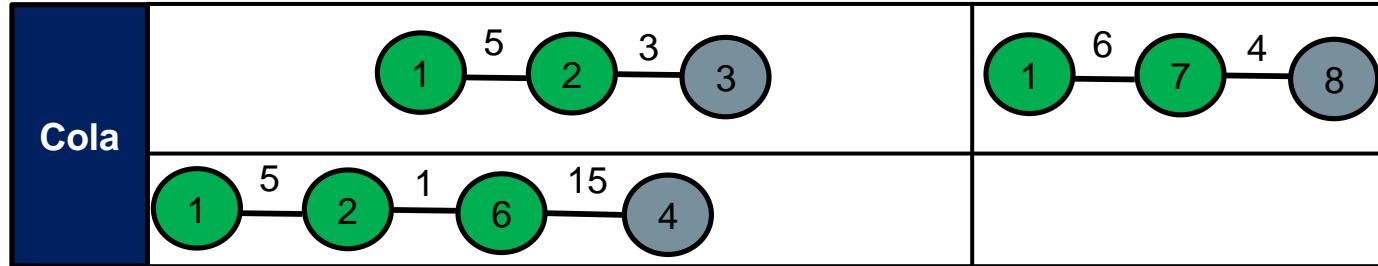
# Dijkstra's Algorithm



<i>u</i>	<i>d(u)</i>	<i>parent(u)</i>
1	0	NULL
2	5	1
3	8	2
4	21	6
5	4	1
6	6	2
7	6	1
8	10	7
9	$+\infty$	NULL
10	$+\infty$	NULL
11	$+\infty$	NULL
12	$+\infty$	NULL

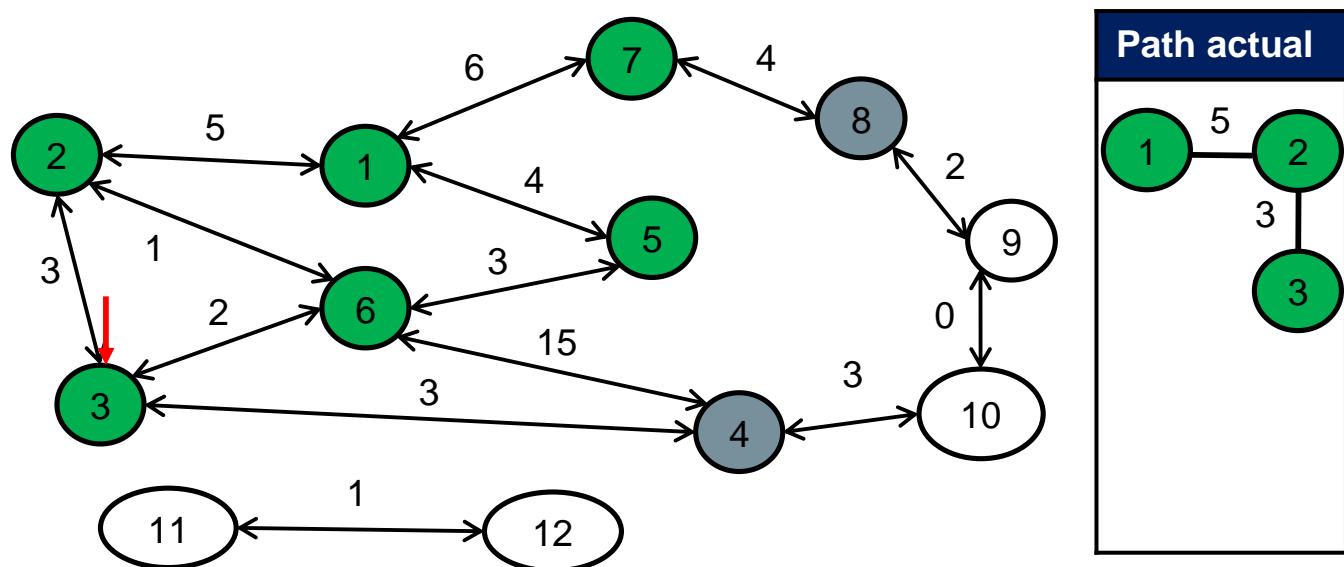
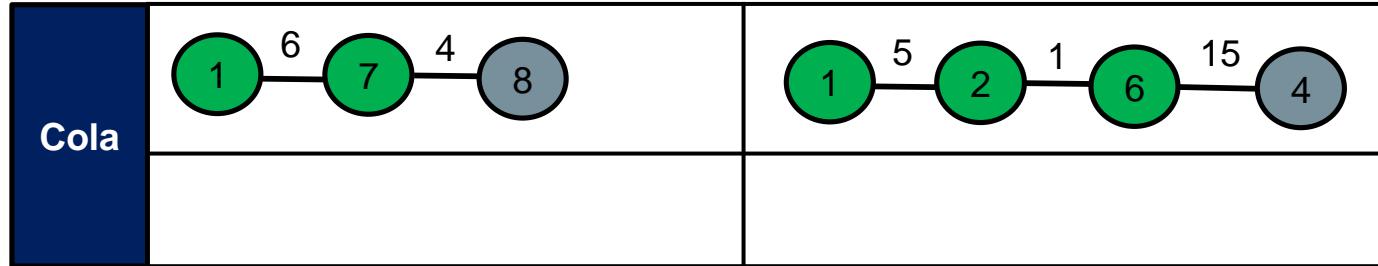


# Dijkstra's Algorithm



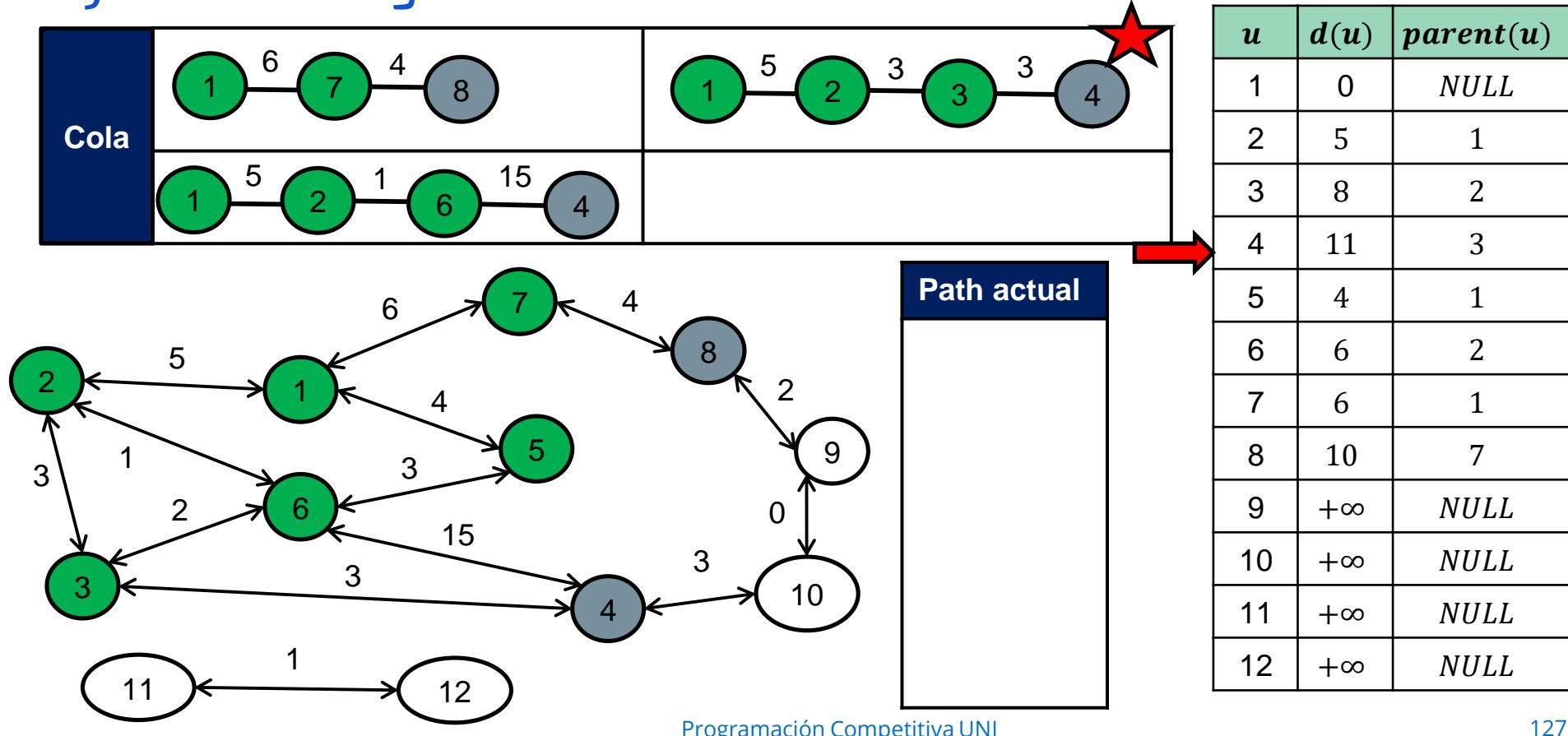
<i>u</i>	<i>d(u)</i>	<i>parent(u)</i>
1	0	NULL
2	5	1
3	8	2
4	21	6
5	4	1
6	6	2
7	6	1
8	10	7
9	$+\infty$	NULL
10	$+\infty$	NULL
11	$+\infty$	NULL
12	$+\infty$	NULL

# Dijkstra's Algorithm

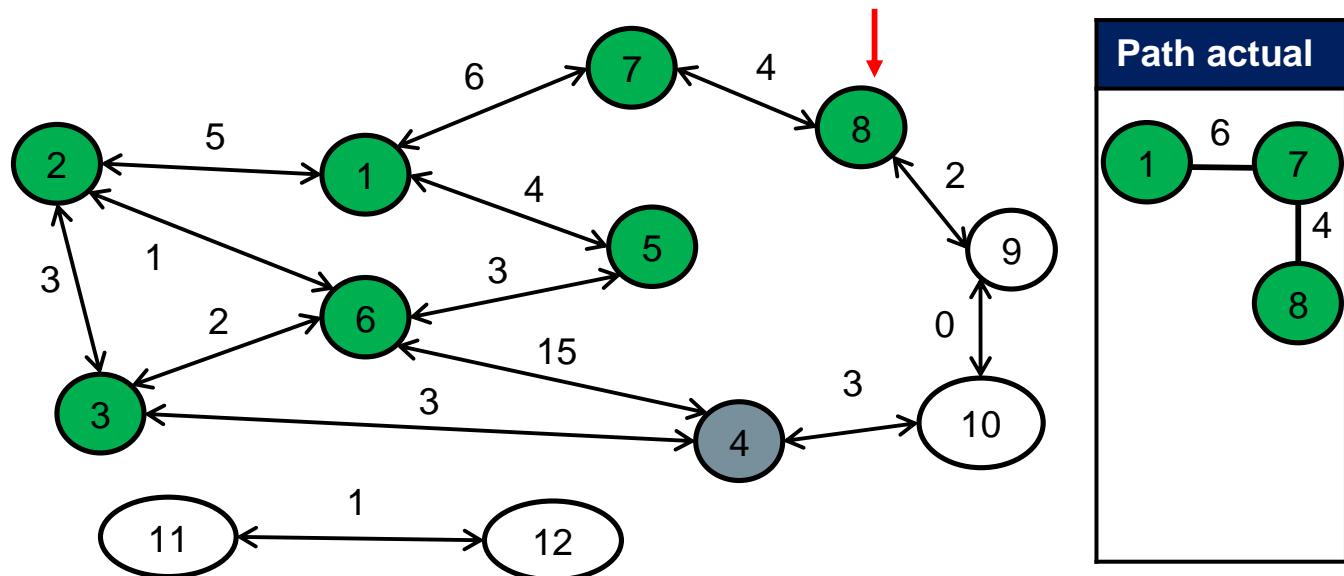
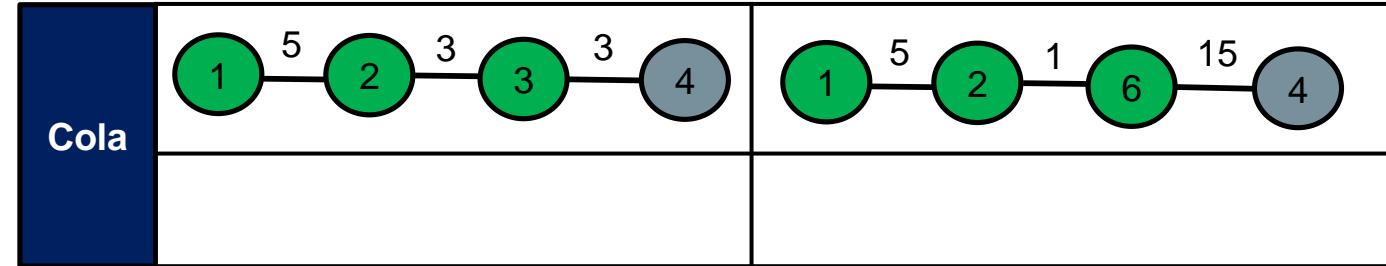


<i>u</i>	<i>d(u)</i>	<i>parent(u)</i>
1	0	NULL
2	5	1
3	8	2
4	21	6
5	4	1
6	6	2
7	6	1
8	10	7
9	$+\infty$	NULL
10	$+\infty$	NULL
11	$+\infty$	NULL
12	$+\infty$	NULL

# Dijkstra's Algorithm

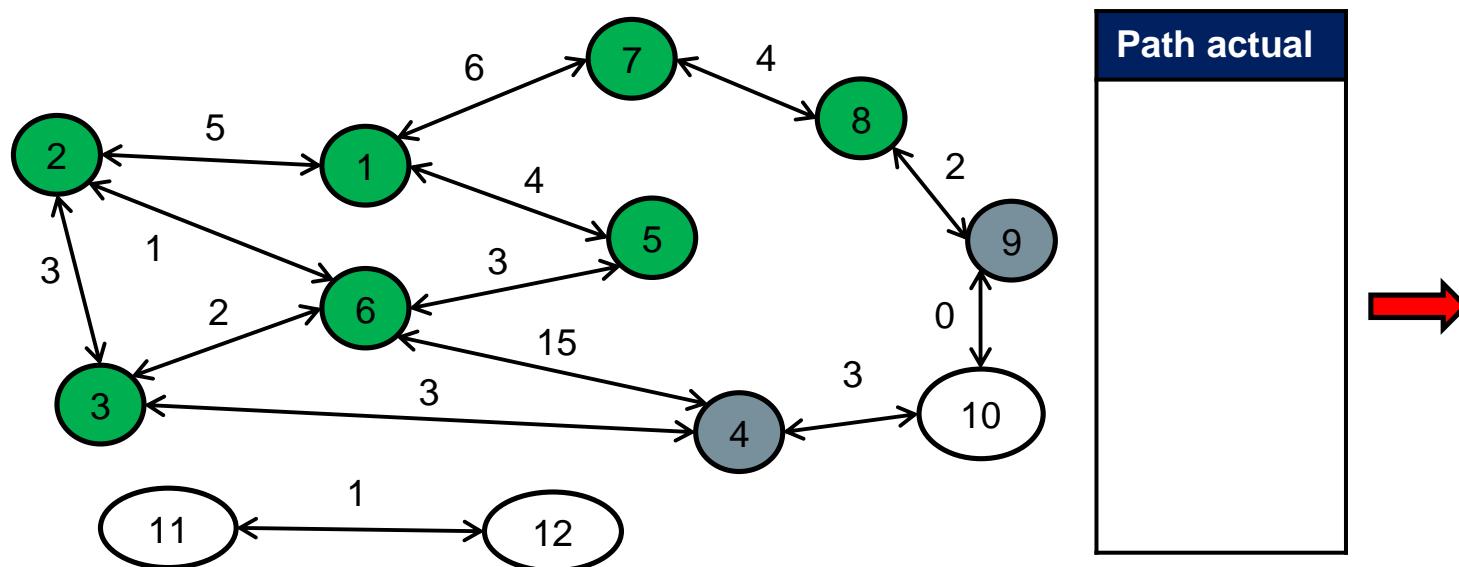
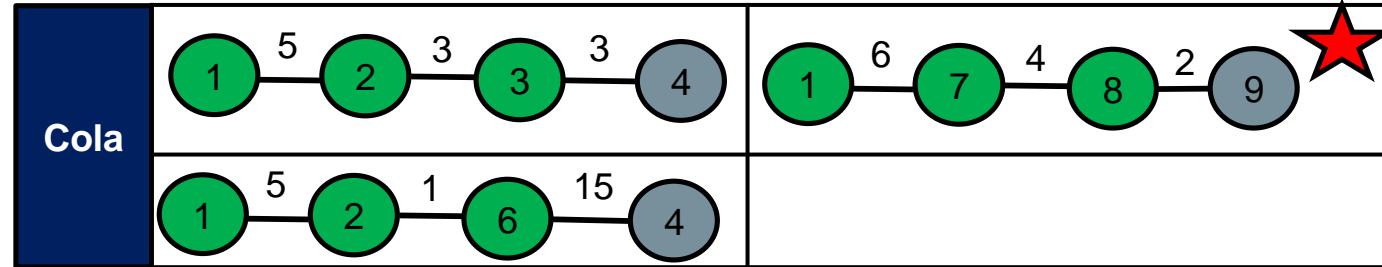


# Dijkstra's Algorithm



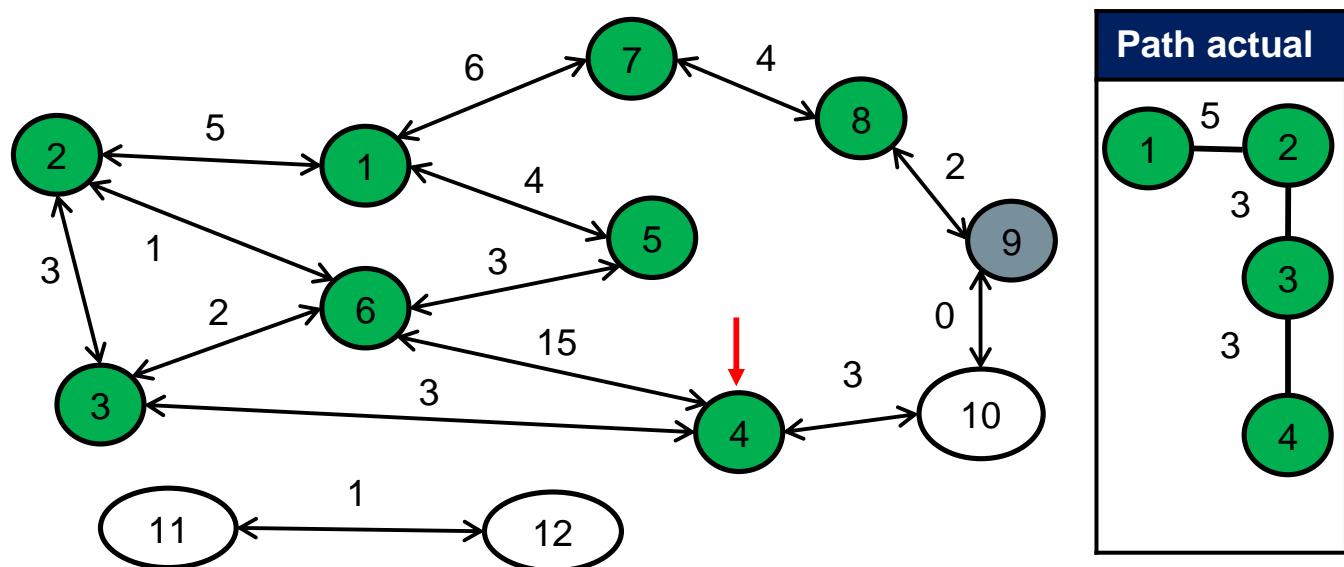
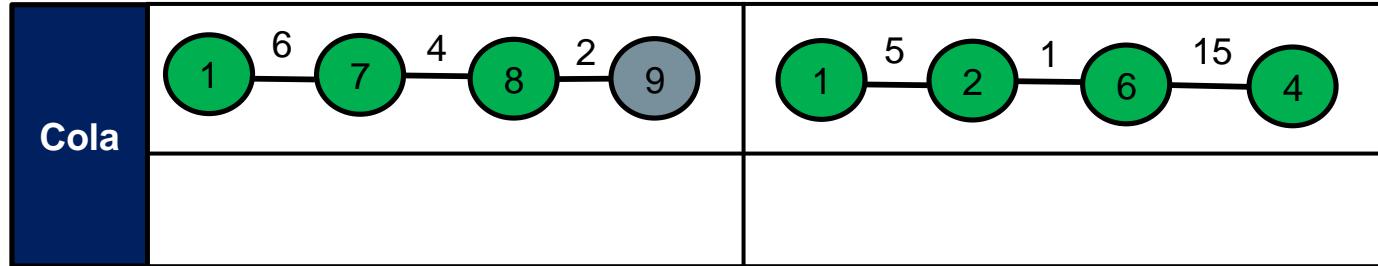
<i>u</i>	<i>d(u)</i>	<i>parent(u)</i>
1	0	NULL
2	5	1
3	8	2
4	11	3
5	4	1
6	6	2
7	6	1
8	10	7
9	$+\infty$	NULL
10	$+\infty$	NULL
11	$+\infty$	NULL
12	$+\infty$	NULL

# Dijkstra's Algorithm

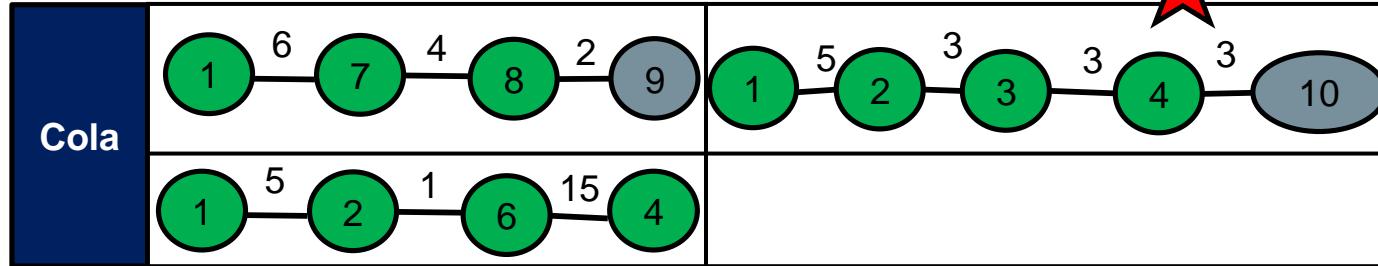


<i>u</i>	<i>d(u)</i>	<i>parent(u)</i>
1	0	NULL
2	5	1
3	8	2
4	11	3
5	4	1
6	6	2
7	6	1
8	10	7
9	12	8
10	$+\infty$	NULL
11	$+\infty$	NULL
12	$+\infty$	NULL

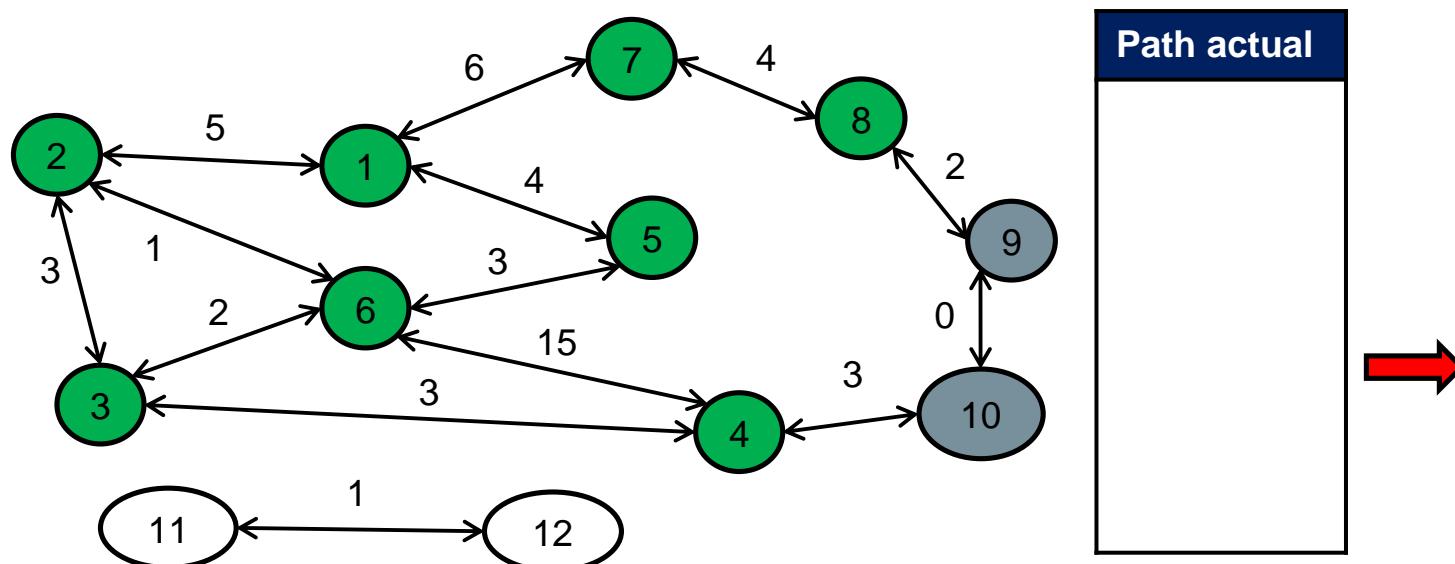
# Dijkstra's Algorithm



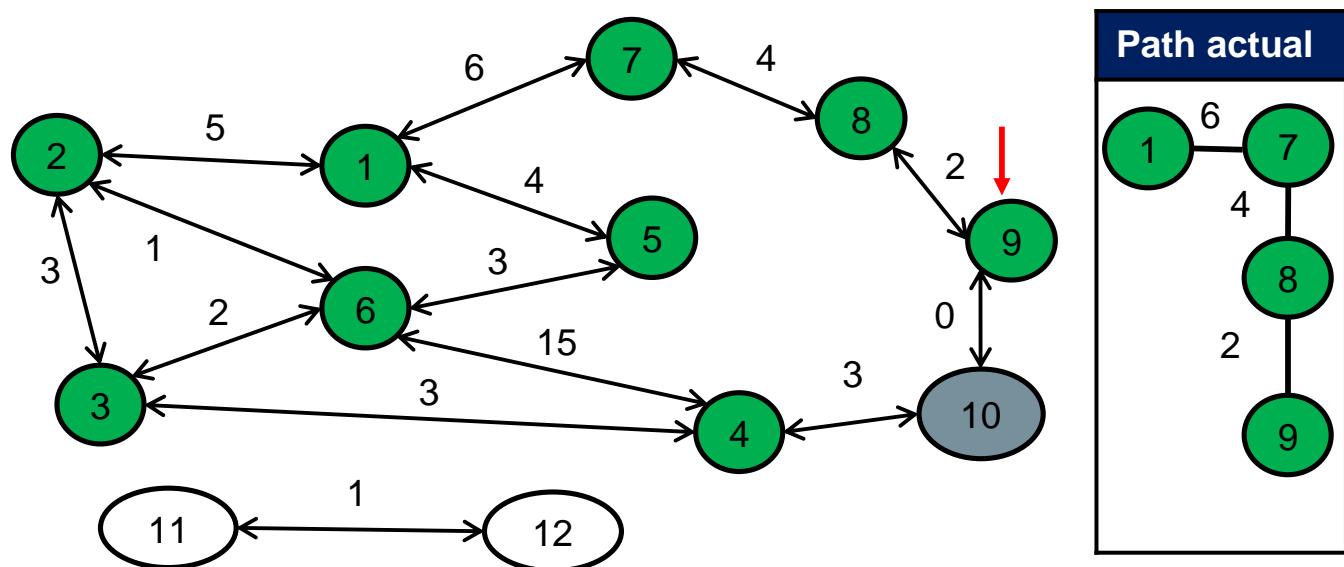
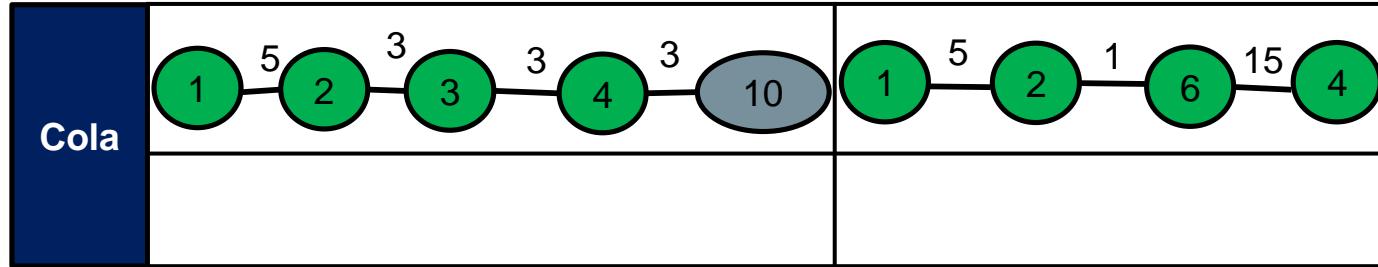
# Dijkstra's Algorithm



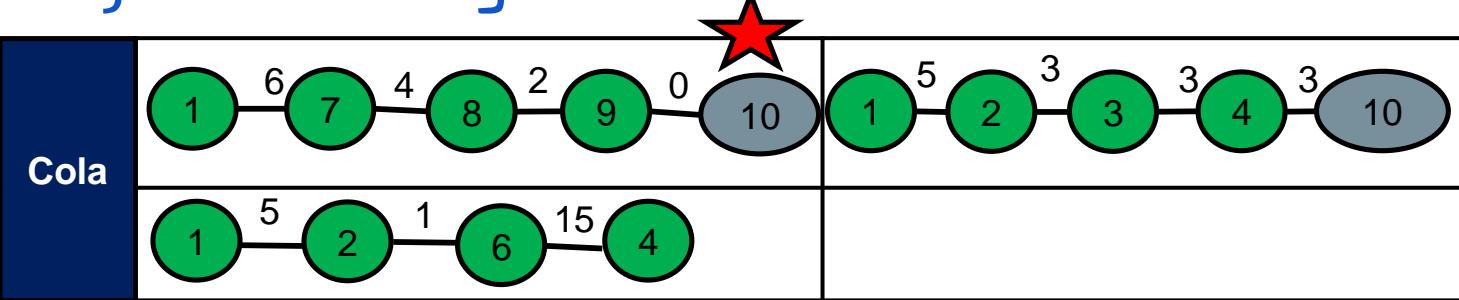
<i>u</i>	<i>d(u)</i>	<i>parent(u)</i>
1	0	NULL
2	5	1
3	8	2
4	11	3
5	4	1
6	6	2
7	6	1
8	10	7
9	12	8
10	14	3
11	$+\infty$	NULL
12	$+\infty$	NULL



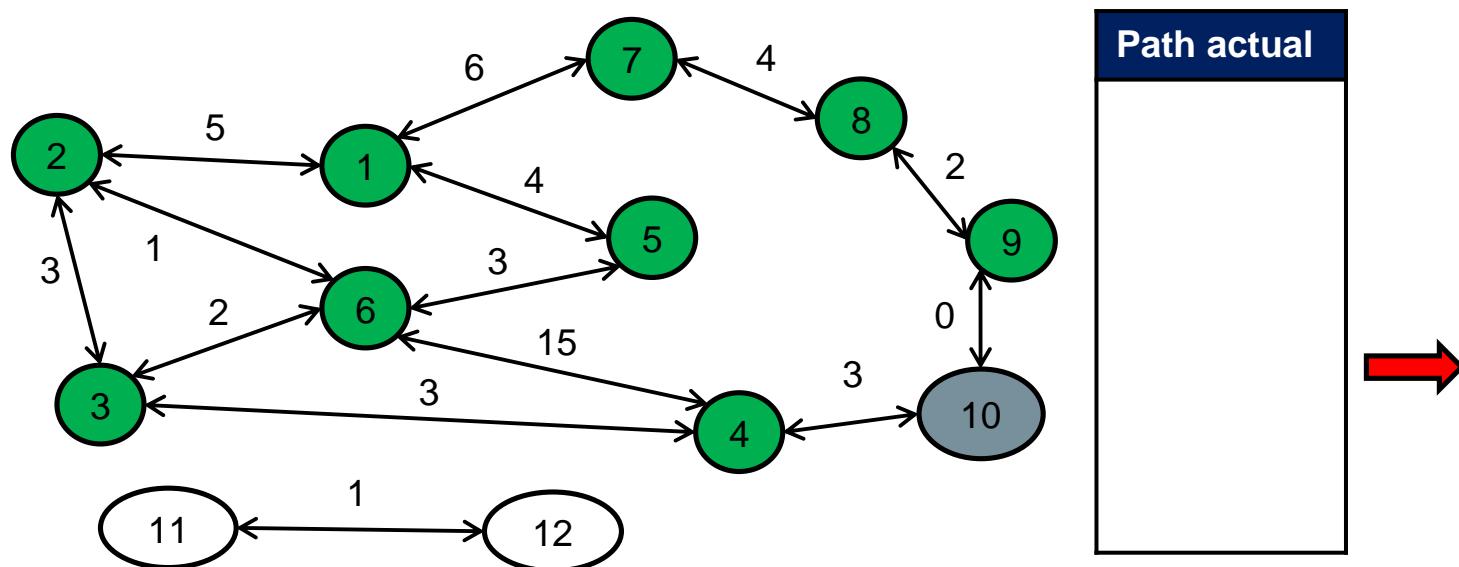
# Dijkstra's Algorithm



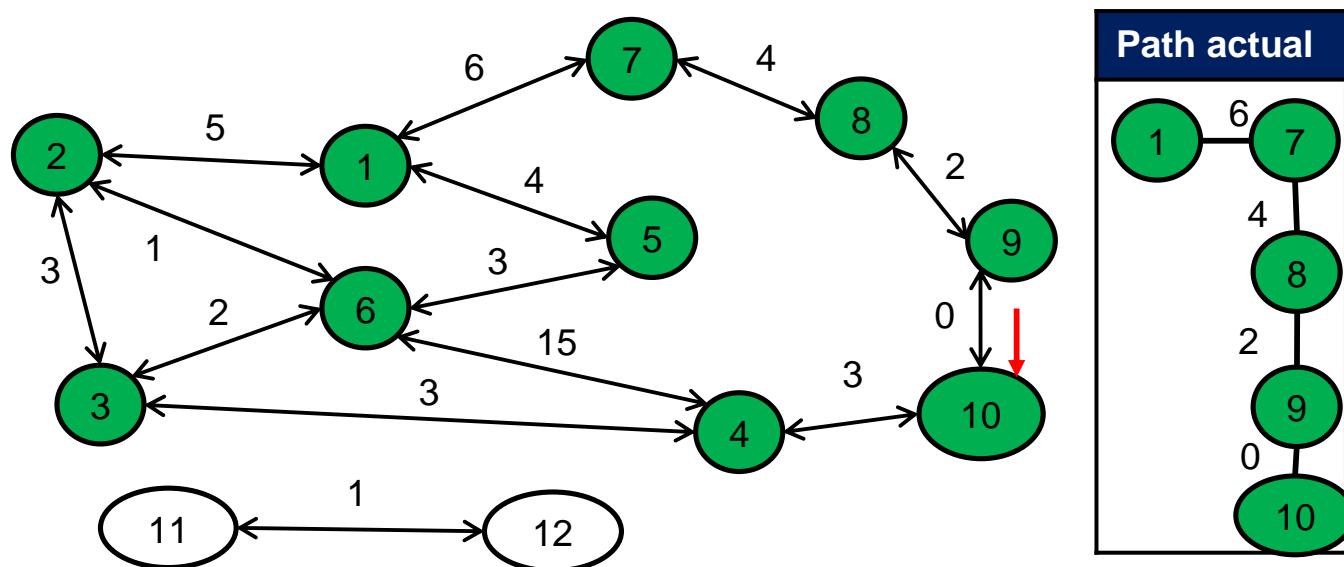
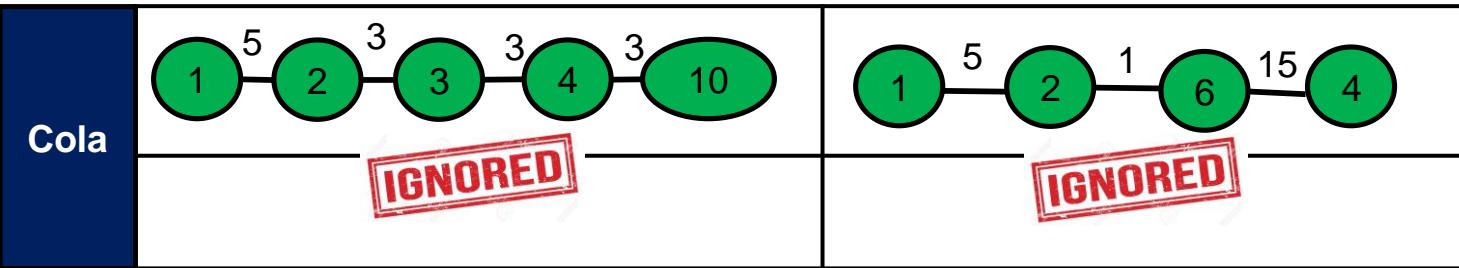
# Dijkstra's Algorithm



<i>u</i>	<i>d(u)</i>	<i>parent(u)</i>
1	0	NULL
2	5	1
3	8	2
4	11	3
5	4	1
6	6	2
7	6	1
8	10	7
9	12	8
10	12	9
11	$+\infty$	NULL
12	$+\infty$	NULL



# Dijkstra's Algorithm



# Generic Dijkstra's Algorithm

Analizando el ejemplo anterior, podemos ver que, en realidad, no nos importa guardar todo el path en la cola de prioridad. Solo nos importa el último vértice y el peso del path, por lo que podríamos guardarlo como una pareja (*node, weight*).

Incluso podemos simplificarlo un poco más, de forma que se parezca un poco más a nuestro **Modified Generic Shortest Path Algorithm**. Del ejemplo que hemos analizado, podemos notar que lo que realmente nos interesa es encontrar el nodo  $u$  de la cola que tenga menor  $d(u)$ .

Para una mayor facilidad, utilizaremos en la demostración este algoritmo dijkstra “genérico” que se ve en la imagen de la derecha.

Note que aquí “list” es un contenedor general “mágico” tal que se pueda hacer eficiente la operación de obtener el nodo con menor  $d(u)$ . Más adelante veremos como implementarlo con una estructura de datos.

---

## Algorithm 8: Generic Dijkstra

---

```
input :  $G(V_G, E_G, w)$ , source :  $s$ 
output:  $d[ ], parent[ ]$ 

1 Initialize( $s$ )
2  $list = \{s\}$ 
3 while  $list$  is not empty do
4     Remove the node  $u$  from  $list$  with minimum  $d(u)$ 
5     foreach edge  $u \rightarrow v$  in  $E_G$  do
6         if  $u \rightarrow v$  is tense then
7             Relax( $u, v$ )
8             if  $v$  is not in  $list$  then
9                 Add node  $v$  to  $list$ 
10            end
11        end
12    end
13 end
14 return  $d, parent$ 
```

---

# Generic Dijkstra's Algorithm - Correctness

- **Teorema (Correctness):** Si el Generic Dijkstra termina, se debe cumplir que  $d(u) = \delta(u)$  para todo  $u \in V$  y el grafo de predecesores inducido por  $\text{parent}(u)$  es un shortest path tree.

## Demostración:

Se deduce directamente del [lema 2.9](#) ya que del **Modified Generic Shortest Path Algorithm** solo hemos especificado la forma de extraer el nodo  $u$ . ■

- ❖ **Observación:** Si utilizamos el algoritmo en un grafo de pesos no negativos, no habrán ciclos negativos y por lo tanto el algoritmo siempre terminaría.

# Generic Dijkstra's Algorithm - Complexity

- **Lema 5.1:** Sea  $D$  el arreglo en donde  $D_i$  es el valor de  $d(u)$  del nodo  $u$  que fue sacado en el  $i$ -ésima ejecución de la línea 4 (de sacar un nodo de la lista). Entonces, si todos los pesos de las aristas son no negativos, el arreglo  $D$  es no decreciente.

## Demostración:

Por inducción en el número de ejecuciones de la línea 4.

- **Caso base:**

El primer nodo en salir de la cola es  $s$  con  $d(s) = 0$  y el arreglo  $D = \{d(s) = 0\}$  es no decreciente trivialmente.

- **Paso inductivo:**

Suponga que en la  $i$ -ésima ejecución se cumple que  $D$  es no decreciente. Suponga que el nodo sacado de la lista en esa ejecución es  $u$ . Suponga que la lista al sacar el nodo  $u$  es  $q = v_1, v_2, \dots, v_k$ . El siguiente nodo en salir en ser sacado de la lista puede ser un nodo de  $q$  o puede ser un nodo que este por agregar  $u$ . Como  $u$  tenía el mínimo estimado entre todos los nodos de  $q$ , se debe cumplir que  $d(u) \leq d(v_i) \dots (1)$ . Cuando  $u$  relaje una arista tensa  $(u, x)$  se actualizará  $d(x) = d(u) + w(u, v)$  y como  $w(u, v) \geq 0$  se cumplirá que  $d(u) \leq d(x) \dots (2)$  (indiferente de que sea un nuevo nodo añadido a la lista o que sea un nodo ya presente en la lista).

Si denotamos a  $q'$  como la lista luego de que  $u$  sea procesado y realiza todas las relajaciones pertinentes. Por (1) y (2) sabemos que cualquier nodo  $v \in q'$  cumplirá que  $d(u) \leq d(v)$ . Por lo tanto luego de la  $(i+1)$ -ésima ejecución, el arreglo  $D$  seguirá siendo no decreciente y se cumplirá el lema. ■

# Generic Dijkstra's Algorithm - Complexity

- **Lema 5.2:** Si todos los pesos son no negativos, cada nodo es sacado de la lista a lo más una vez (y por lo tanto también entrará a lo más una vez).

## Demostración:

Cada vez que un nodo  $v$  entra a la lista, está acompañado de una relajación de una arista tensa  $(u, v)$ . Cada relajación de una arista  $(u, v)$  decrece el valor de  $d(v)$ .

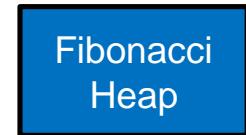
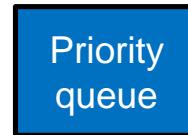
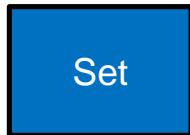
Por contradicción, suponga que un nodo  $v$  es sacado más de 1 vez de la lista. La 2da vez que sea sacado, su  $d(v)$  será menor que la primera vez que ingresó (porque lo han tenido que relajar para que entre). Pero al cumplirse eso, entonces el arreglo  $D$  definido en **el lema 5.1** ya no sería no decreciente, llegando a una contradicción. ■

- **Corolario:** Suponga que la complejidad de sacar un nodo de la lista es  $O(|pull|)$  y la complejidad de meter un nodo en la lista es  $O(|push|)$ . Entonces, si todos los pesos son no negativos, la complejidad del Generic Dijkstra es  $O((|pull| + |push|) \times V + E)$

# Dijkstra's Algorithm Implementation

El último lema y corolario nos dan una buena idea de que la complejidad de Dijkstra podría llegar a ser muy buena, provista de un estructura de datos eficiente.

En C++ podríamos usar un **priority\_queue** (que usa la estructura “heap”) o un **set**(que usa la estrutura “Red black Tree”). También existe una estructura con la mejor complejidad para este tipo de tareas que es una llamada **Fibonacci heap** (que es como un “fancy priority queue”) sin embargo son difíciles de implementar y la constante alta que tienen hace que no sea tan rápido en la práctica.

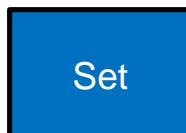


# Dijkstra's Algorithm Implementation - Set

Una primera idea podría ser por ejemplo usar un **set** de nodos y crearles un operador de comparación personalizado que utilice los estimados  $d()$  para comparar los nodos. ¡Sin embargo hay un problema con esa idea! Cuando relajas una arista  $(u, v)$  es posible que  $v$  ya esté en el set, y al hacer la relajación vas a cambiar el valor de  $d(v)$  por lo tanto el ordenamiento del set ya no sería válido (no se ordenará automáticamente). Es más, la estructura se malograría al hacer esto y ya no funcionaría bien.

Una posible solución sería que antes de la relajación se elimine a  $v$  del set y luego de la relajación se lo vuelva a insertar.

Como cada operación del set es logarítmica, y el tamaño de la cola a lo más llega a ser  $V$  la complejidad sería  $O((V + E) \log V)$ . Asumiendo que  $V \leq E$  tendríamos complejidad  $O(E \log V)$



# Dijkstra's Algorithm Implementation - Set

Un error muy común es solo comparar  $d(u) < d(v)$  dentro de la función comparadora del set. Si haces eso, el set tomará a dos elementos con  $d(u) == d(v)$  como iguales y solo se quedará con uno de ellos.

```
const Long MX = 1e5;
const Long INF = 1e18;

Long d[MX];
struct Compare{
    bool operator() (Long u, Long v) const {
        if (d[u] == d[v]) return u < v; //IMPORTANT
        return d[u] < d[v];
    }
};

struct Graph{
    vector<pair<Long, Long>> adj [MX];
    Long parent[MX];

    void addEdge(Long u, Long v, Long w) {
        adj[u].push_back({v , w});
        //adj[v].push_back({u , w}); //Undirected
    }
}
```

```
void dijkstra(Long s, Long n){ //O(E log V)
    for(Long i = 0; i < n; i++) {
        parent[i] = -1;
        d[i] = INF;
    }
    set<Long, Compare> q;
    d[s] = 0;
    q.insert(s);
    while(!q.empty()){
        Long u = *q.begin();
        q.erase(q.begin());
        for(auto e : adj[u]){
            Long v = e.first;
            Long w = e.second;
            if(d[u] + w < d[v]){
                q.erase(v);
                d[v] = d[u] + w;
                parent[v] = u;
                q.insert(v);
            }
        }
    }
} G;
```

# Dijkstra's Algorithm Implementation – Set 2

En vez de crear un comparador, podemos enviar al set una pareja de valores ( $d(node), node$ )

La complejidad sigue siendo  $O(E \log V)$  pero trabajar con pairs puede aumentar un poco la constante

```
const Long MX = 1e5;
const Long INF = 1e18;

struct Graph{
    vector<pair<Long, Long>> adj[MX];
    Long parent[MX];
    Long d[MX];

    void addEdge(Long u, Long v, Long w) {
        adj[u].push_back({v, w});
        //adj[v].push_back({u, w}); //Undirected
    }
}
```

```
void dijkstra(Long s, Long n){ //O(E log V)
    for(Long i = 0; i < n; i++) {
        parent[i] = -1;
        d[i] = INF;
    }
    set<pair<Long, Long>> q;
    d[s] = 0;
    q.insert({d[s], s});
    while(!q.empty()) {
        Long u = q.begin()->second;
        q.erase(q.begin());
        for(auto e : adj[u]) {
            Long v = e.first;
            Long w = e.second;
            if(d[u] + w < d[v]) {
                q.erase({d[v], v});
                d[v] = d[u] + w;
                parent[v] = u;
                q.insert({d[v], v});
            }
        }
    }
} G;
```

# Dijkstra's Algorithm Implementation – Priority Queue

Podemos utilizar un **priority queue** de parejas ( $weight, node$ ) como lo hicimos en el código anterior. Sin embargo esta estructura trae otro problema ya que aquí no podemos eliminar como lo hacíamos en el set. Entonces tendremos que permitir que en la cola de prioridad hayan varias parejas que tengan el mismo nodo (segunda componente) pero diferente peso del path (primera componente). Sin embargo esto es fácil de tratar. Primero note que todas las parejas  $p = (w, u)$  deben cumplir  $w \geq d(u)$  (No puede ser menor porque nadie ha podido relajar  $u$  hasta un valor menor todavía). Luego aquellas parejas que cumplan  $w > d(u)$  no nos sirven porque no van a poder relajar a nadie y podemos ignorarlas (**si no las ignoramos la complejidad aumenta a  $O(E^2 \log V)$**  pero el algoritmo sigue siendo correcto).

Como cada operación del **priority queue** es logarítmica. Cada nodo  $u$  puede entrar en la cola hasta  $degree(u)$  veces por lo que el tamaño de la cola a lo más llega a ser  $\sum indgree(u) = E$  la complejidad sería  $O(E \log E)$ . Asumiendo que no hay aristas múltiples tendríamos que  $E \leq V^2$  por lo que la complejidad final sería  $O(E \log V)$

Priority  
queue

# Dijkstra's Algorithm Implementation – Priority Queue

Aunque tiene la misma complejidad, un heap tiene menos constante que un red black tree.

Para facilitar la lectura del código se ha utilizado la palabra “Path” para reemplazar a *pair <Long, Long >*

```
typedef pair<Long, Long> Path;

const Long MX = 1e5;
const Long INF = 1e18;

struct Graph{
    vector<pair<Long, Long>> adj[MX];
    Long parent[MX];
    Long d[MX];

    void addEdge(Long u, Long v, Long w) {
        adj[u].push_back({v, w});
        //adj[v].push_back({u, w}); //Undirected
    }
}
```

```
void dijkstra(Long s, Long n){ //O(E log V)
    for(Long i = 0; i < n; i++) {
        parent[i] = -1;
        d[i] = INF;
    }
    priority_queue<Path, vector<Path>, greater<Path>> q;
    d[s] = 0;
    q.push({d[s], s});
    while(!q.empty()) {
        Path p = q.top();
        q.pop();
        Long u = p.second;
        Long weight = p.first;
        if (weight != d[u]) {
            continue;
        }
        for(auto e : adj[u]){
            Long v = e.first;
            Long w = e.second;
            if(d[u] + w < d[v]){
                d[v] = d[u] + w;
                parent[v] = u;
                q.push({d[v], v});
            }
        }
    }
} G;
```

# Dijkstra's Algorithm Implementation – Dense Graph

Si el grafo es denso ( $E \approx V^2$ ) nos puede convenir más un algoritmo  $O(V^2)$  como el siguiente.

```
const Long MX = 1e5;
const Long INF = 1e18;

struct Graph{
    vector<pair<Long, Long>> adj[MX];
    bool vis[MX];
    Long d[MX];
    Long parent[MX];

    void addEdge(Long u, Long v, Long w) {
        adj[u].push_back({v, w});
        adj[v].push_back({u, w});
    }
}
```

```
void dijkstra(Long s, Long n){ //O(V^2)
    for(Long i = 0; i < n; i++) {
        vis[i] = false;
        parent[i] = -1;
        d[i] = INF;
    }
    d[s] = 0;
    for(Long i = 0; i < n; i++) {
        Long u = -1;
        for(Long j = 0; j < n; j++) {
            if(!vis[j] && (u == -1 || d[j] < d[u])) {
                u = j;
            }
        }
        if(u == -1 || d[u] == INF) {
            break;
        }
        vis[u] = true;
        for(auto e : adj[u]) {
            Long v = e.first;
            Long w = e.second;
            if(d[u] + w < d[v]) {
                d[v] = d[u] + w;
                parent[v] = u;
            }
        }
    }
} G;
```

# Función para obtener el path

A continuación presentamos una función general que nos permitirá obtener la secuencia de vértices que forman el shortest path hasta algún nodo  $u$ . Esta función es válida para cualquier algoritmo que sea una implementación del algoritmo genérico de Ford (o el modificado). Es decir, funciona para cualquier implementación de Dijkstra, BFS, etc. (siempre y cuando hayas calculado los *parent*)

```
vector<Long> getPath(Long v) {
    if (d[v] == INF) {
        return {};
    }
    vector<Long> path;
    while (v != -1) {
        path.push_back(v);
        v = parent[v];
    }
    reverse(path.begin(), path.end());
    return path;
}
```

# Dijkstra's Algorithm – Negative Edges

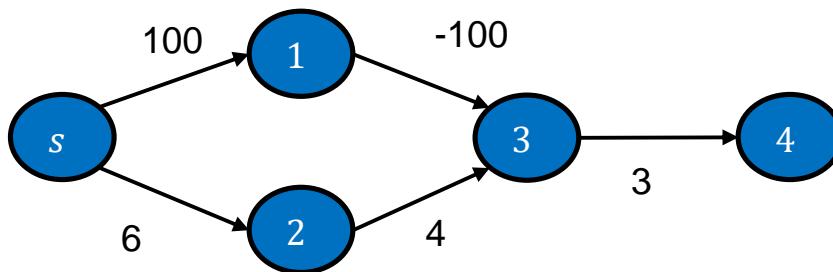
¿Qué pasa si existen aristas de **pesos negativo**?

En el Generic Dijkstra, el lema 5.1 ya no se cumpliría siempre (y tampoco el lema 5.2) por lo tanto es posible que un vértice sea extraído de la cola más de una vez. Pero el algoritmo **Generic Dijkstra seguirá funcionando** ya que el teorema de su correctness no depende de los lemas 5.1 ni 5.2.

¿Cuáles son los efectos en las implementaciones del Generic Dijkstra?

Depende.

1) Si en tu implementación forzaste a que cada nodo sea “visitado” una sola vez (utilizando un arreglo booleano por ejemplo), entonces el algoritmo ya no funcionará y dará respuestas incorrectas (por ejemplo el algoritmo que presentamos para Dense Graphs ya no funcionaría). Ejemplo:



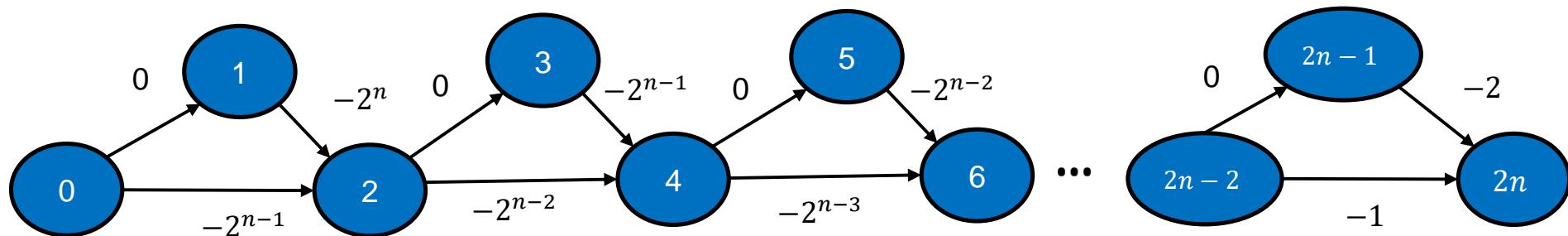
El algoritmo le dará prioridad al camino  $s \rightarrow 2 \rightarrow 3 \rightarrow 4$  cuando el shortest path es obviamente  $s \rightarrow 1 \rightarrow 3 \rightarrow 4$ . Cuando intente ir por el 2do camino, se encontrará que 3 ya está visitado y lo ignorará, obteniendo una distancia incorrecta para el nodo 4

# Dijkstra's Algorithm – Negative Edges

¿Cuáles son los efectos en las implementaciones del Generic Dijkstra?

Depende.

2) Si en tu implementación permites que un nodo sea “visitado” varias veces (como la implementación del set1, set2, priority\_queue de las diapos anteriores), entonces tu algoritmo funcionará, pero será muy lento. Una primera cota sería la misma del **Modified Generic Shortest Path Algorithm**  $O(2^V)$  relajaciones lo cual daría una complejidad total de  $O(E \times 2^V)$ . ¿Pero realmente será posible construir un caso exponencial? Sí. Douglas Shier y Christoph Witzgall plantearon la siguiente familia de grafos:



Esta familia de grafos producirá en Dijkstra  $O(2^n) = O(2^{V/2})$  relajaciones.

Donald Jhonson incluso llegó a crear el peor caso posible utilizando una familia de grafos completos que causaban  $O(2^V)$  relajaciones

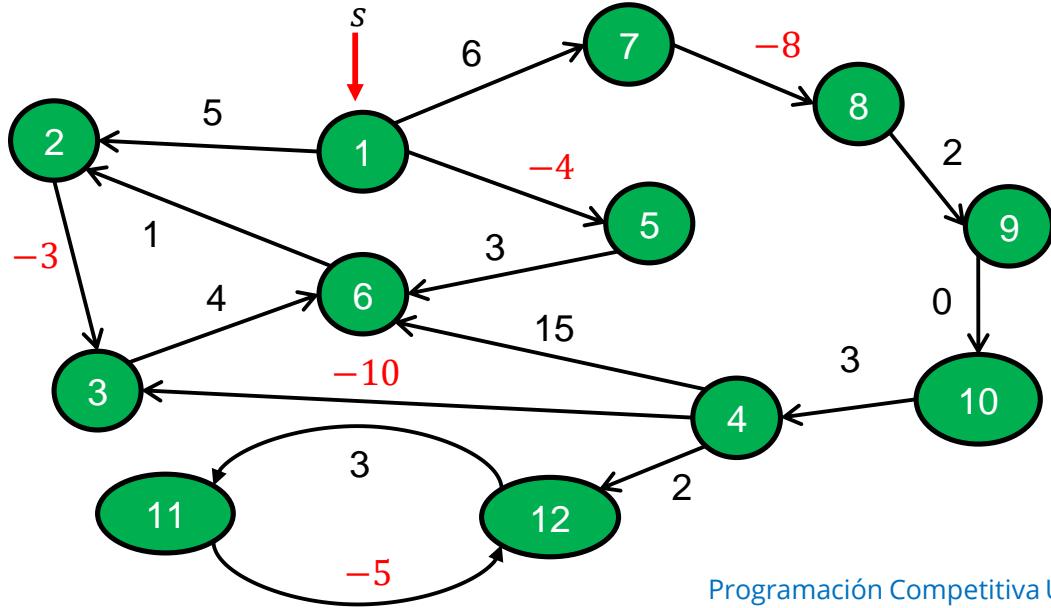
# Contenido

1. Introducción	→
2. Propiedades de los Shortest Path y Algoritmo Genérico	→
3. Single Source Shortest Path en DAG	→
4. Single Source Shortest Path con pesos unitarios	→
5. Single Source Shortest Path con pesos no negativos	→
<b>6. Single Source Shortest Path con cualquier peso real</b>	→
7. All-pairs shortest path	→

# Single Source Shortest Path – Any weight function

**Problema:** Sea un grafo dirigido con pesos  $G(V, E, w)$ . Cuya función de pesos es  $w: E(G) \rightarrow \mathbb{R}$ .

Se pide hallar la longitud del shortest path desde un nodo fuente  $s$  hasta cualquier otro nodo  $u$  (distancia del nodo  $u$ ), denotado como  $\delta(u)$ .



$u$	$\delta(u)$
1	0
2	-2
3	-7
4	3
5	-4
6	-3
7	6
8	-2
9	0
10	0
11	$-\infty$
12	$-\infty$

# Single Source Shortest Path – Any weight function

En 1955 Shimbel publicó un algoritmo que permitía hallar  $\delta(u, v)$  para cualquier función de pesos reales (siempre que no haya ciclos negativos) utilizando operaciones con matrices. Incluso el algoritmo se puede generalizar y dar el peso de los shortest path que usen exactamente  $k$  aristas (Para más información: [https://cp-algorithms.com/graph/fixed\\_length\\_paths.html](https://cp-algorithms.com/graph/fixed_length_paths.html))

En 1958 Richard Bellman utilizó una formulación similar a la propuesta por Ford para hallar un algoritmo que permitía resolver el problema de shortest path path cualquier función de pesos reales (mientras no haya ciclos negativos), obteniendo un algoritmo similar al de Shimbel.

Max Woodbury y George Dantzig ya habían descubierto (pero no publicado) el mismo algoritmo de Bellman en 1957 y fueron posteriormente reconocidos por el mismo Bellman en su publicación.

George Minty en 1958 también llegó a un algoritmo similar.

Sin embargo, el algoritmo es conocido como **The Bellman-Ford algorithm**



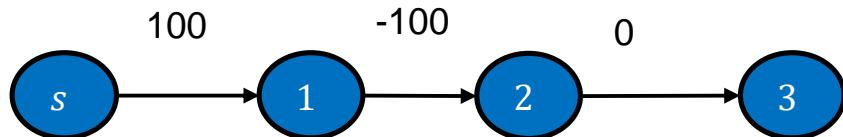
Richard Bellman

# Bellman-Ford Algorithm

Podemos resumir el algoritmo como “Mientras hayan aristas tensas, relaja todas la lista de aristas del grafo”. Es obvio que este es un caso particular del algoritmo genérico de Ford, por lo que la demostración de su correctitud es directa.

La complejidad del algoritmo es  $O(E \times \#iterations)$ . Faltaría encontrar una cota para el número de iteraciones.

**Observación:** El número exacto de iteraciones depende del orden en que sean visitadas las aristas del grafo. Por ejemplo en el siguiente grafo:



Si las aristas están en el orden  $s \rightarrow 1, 1 \rightarrow 2, 2 \rightarrow 3$  entonces el algoritmo terminará luego de la 1ra iteración. Si están en el orden inverso, demorará 3 iteraciones.

---

## Algorithm 9: Bellman-Ford

---

```
input :  $G(V_G, E_G, w)$  , source :  $s$ 
output:  $d[ ]$ , parent[ ]
1 Initialize ( $s$ )
2 while there exist tense edges do
3   foreach edge  $e = u \rightarrow v$  in  $E_G$  do
4     if  $u \rightarrow v$  is tense then
5       | Relax ( $u, v$ )
6     end
7   end
8 end
9 return  $d, parent$ 
```

---

# Bellman-Ford Algorithm - Correctness

- **Teorema (Correctness):** Si el algoritmo de Bellman-Ford termina, se debe cumplir que  $d(u) = \delta(u)$  para todo  $u \in V$  y el grafo de predecesores inducido por  $\text{parent}(u)$  es un shortest path tree.

**Demostración:**

Se deduce directamente del teorema de la correctitud del **Generic Shortest Path Algorithm** ■

# Bellman-Ford Algorithm - Complexity

- **Lema 6.1:** Suponga que no hay ciclos negativos alcanzables desde  $s$ . Sea  $u$  un nodo alcanzable desde  $s$ , y sea  $p = v_0 \rightarrow v_1 \rightarrow \dots \rightarrow v_k$  un shortest path de  $k$  aristas donde  $v_0 = s$  y  $v_k = u$ . Denotemos como  $\#iteraciones$  al número de iteraciones necesarias en Bellman-Ford para que  $d(u) = \delta(u)$ . Entonces  $\#iteraciones \leq k$ .

## Demostración:

Demostremos por inducción que luego de  $i$  iteraciones ( $i \leq k$ ) se cumple que  $d(v_i) = \delta(v_i)$ .

- **Caso base:** Luego de 0 iteraciones  $d(v_0 = s) = 0 = \delta(v_0)$
- **Paso inductivo:**

Suponga que luego de  $i$  iteraciones ( $i < k$ ) se cumple que  $d(v_i) = \delta(v_i)$ . Por optimal substructure sabemos que  $\delta(v_{i+1}) = \delta(v_i) + w(v_i, v_{i+1}) \dots (1)$ . En la siguiente iteración, Bellman-Ford tratará de relajar todas las  $E$  aristas (en caso estén tensas). En particular, intentará relajar  $(v_i, v_{i+1})$ .

1) **Si la relaja,** por convergence property  $d(v_{i+1}) = \delta(v_{i+1})$ .

2) **Si no la relaja,** significa que no estuvo tensa y  $d(v_i) + w(v_i, v_{i+1}) \geq d(v_{i+1}) \Rightarrow \delta(v_{i+1}) \geq d(v_{i+1}) \dots (1)$ . Si juntamos (1) con upper bound property tenemos que  $d(v_{i+1}) = \delta(v_{i+1})$ .

En ambos casos se cumple la hipótesis. ■

# Bellman-Ford Algorithm - Complexity

- **Lema 6.2:** Si no hay ciclos negativos alcanzables desde  $s$ , el algoritmo de Bellman-Ford termina luego de  $\# \text{iteraciones} \leq |V| - 1$

## Demostración:

Todo path tiene a lo más  $|V| - 1$  aristas. Entonces por el lema 6.1, para cualquier nodo  $u$  el número de iteraciones que se necesita para que  $d(u) = \delta(u)$  es menor o igual a  $|V| - 1$ . ■

- **Corolario 1 (Complexity):** La complejidad del algoritmo de Bellman Ford es  $O(|V||E|)$
- **Corolario 2 (Negative Cycle Detection):** Si luego de  $|V| - 1$  iteraciones todavía existen aristas tensas, entonces existen un ciclo negativo alcanzable desde  $s$ .

# Bellman-Ford Algorithm - Code

Inicialmente podemos dar el siguiente código que calculará correctamente  $\delta(u)$  siempre y cuando  $\delta(u)$  no sea  $-\infty$ . También es capaz de detectar ciclos negativos. Podemos usar un vector de aristas ya que no es necesaria la lista de adyacencia

```
const Long MX = 1e4;
const Long INF = 1e18;

struct Edge{
    Long u, v, w;
    Edge() {}
    Edge(Long u, Long v, Long w) :
        u(u) , v(v) , w(w) {}
};

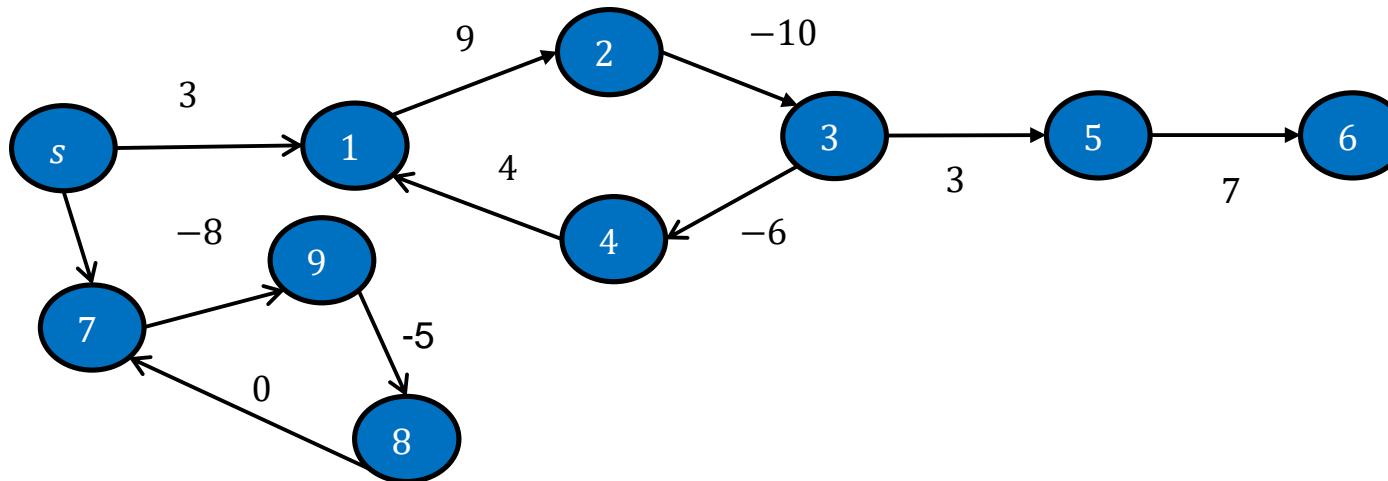
struct Graph{
    vector<Edge> edges;
    Long d[MX];
    Long parent[MX];

    void addEdge(Long u, Long v, Long w) {
        edges.push_back(Edge(u, v , w));
    }
};
```

```
bool bellmanFord(Long s , Long n){ //O(VE)
    for(Long i = 0; i < n; i++){
        d[i] = INF;
        parent[i] = -1;
    }
    d[s] = 0;
    bool tense;
    for (Long i = 0; i < n; i++) {
        tense = false;
        for (Edge e : edges ) {
            if (d[e.u] != INF && d[e.u] + e.w < d[e.v]) {
                d[e.v] = d[e.u] + e.w; //relax
                parent[e.v] = e.u;
                tense = true;
            }
        }
        if(!tense) {
            break;
        }
    }
    if(tense) {
        return true; //negative cycle found
    } else{
        return false; //no negative cycle
    }
}
```

# Bellman-Ford Algorithm – Negative Cycles

¿Qué pasa si queremos también que  $d(u) = -\infty$  en aquellos vértices que pasen por algún ciclo negativo? Los primeros candidatos a tener  $d(u) = -\infty$  son aquellos nodos que hayan sido relajados en la iteración  $\#V$  aunque no necesariamente son todos los nodos que necesitamos. Por ejemplo si tenemos el siguiente grafo:



Podría ser que en la iteración  $\#V$  solo las aristas  $2 \rightarrow 3$ ,  $3 \rightarrow 5$ ,  $7 \rightarrow 9$  hayan sido relajadas, sin embargo  $3, 5$  y  $9$  no son los únicos nodos con  $\delta(u) = -\infty$ . Sin embargo, si nosotros hacemos un  $dfs$  desde ese conjunto de nodos para ver a quiénes alcanzan esos nodos, obtendremos el conjunto de nodos con  $\delta(u) = -\infty$ .

# Bellman-Ford Algorithm – Negative Cycles

- **Lema 6.3:** Suponga que existe ciclos negativos alcanzables desde  $s$  y que en la iteración  $|V| - \text{ésima}$  de Bellman-Ford se relajaron el conjunto de vértices  $X = x_0, x_1, \dots, x_t$ . Entonces cualquier nodo  $u$  que tenga  $\delta(u) = -\infty$  es alcanzable por algún nodo de  $X$ .

## Demostración:

Como hay ciclos negativos alcanzables desde  $s$ , se deduce que el conjunto  $X$  no es vacío debido al **Corolario de Negative Cycle Detection**.

Supongamos por contradicción que existiera algún nodo  $u$  con  $\delta(u) = -\infty$  que no sea alcanzable por ningún nodo en  $X$ . Como  $\delta(u) = -\infty$  tiene que haber algún ciclo negativo en algún *walk* que vaya desde  $s$  hasta  $u$ .

Suponga que  $c = v_1 \rightarrow v_2 \rightarrow \dots \rightarrow v_k \rightarrow v_1$  es algún ciclo negativo en algún *walk* de ese tipo.

$$\text{Entonces } w(c) = w(v_1, v_2) + w(v_2, v_3) + \dots + w(v_k, v_1) < 0$$

Como  $u$  no es alcanzable por el conjunto  $X$ , ninguna arista del ciclo estuvo tensa en la última iteración, entonces:

$$d(v_1) + w(v_1, v_2) \geq d(v_2) \Rightarrow \cancel{d(v_1)} + w(v_1, v_2) - \cancel{d(v_2)} \geq 0$$

$$d(v_2) + w(v_2, v_3) \geq d(v_3) \Rightarrow \cancel{d(v_2)} + w(v_2, v_3) - \cancel{d(v_3)} \geq 0$$

...

$$d(v_k) + w(v_k, v_1) \geq d(v_1) \Rightarrow \cancel{d(v_k)} + w(v_k, v_1) - \cancel{d(v_1)} \geq 0$$

Si sumamos todo, tendremos una suma telescópica donde todos los  $d(v_i)$  se eliminan.

Nos queda  $w(c) = w(v_1, v_2) + w(v_2, v_3) + \dots + w(v_k, v_1) \geq 0$  lo cual es una contradicción. ■

# Bellman-Ford Algorithm – Code Negative Cycles

```
const Long MX = 1e4;
const Long INF = 1e18;

struct Graph{
    vector<pair<Long, Long>> adj[MX];
    Long d[MX];
    Long parent[MX];

    void clear(Long n) {
        for (int i = 0; i < n; i++) {
            adj[i].clear();
        }
    }

    void addEdge(Long u, Long v, Long w) {
        adj[u].push_back({v, w});
    }

    void dfs(Long u) {
        d[u] = -INF;
        for (auto e : adj[u]) {
            Long v = e.first;
            if (d[v] != -INF) {
                dfs(v);
            }
        }
    }
}
```

```
bool bellmanFord(Long s, Long n){ //O(VE)
    for(Long i = 0; i < n; i++) {
        d[i] = INF;
        parent[i] = -1;
    }
    d[s] = 0;
    vector<bool> cycle(n, false);
    bool tense;
    for (Long i = 0; i < n; i++) {
        tense = false;
        for (Long u = 0; u < n; u++) {
            for (auto e : adj[u]) {
                Long v = e.first;
                Long w = e.second;
                if (d[u] != INF && d[u] + w < d[v]) { //tense
                    d[v] = d[u] + w; //relax
                    parent[v] = u;
                    tense = true;
                    if (i == n - 1) {
                        cycle[v] = true;
                    }
                }
            }
            if (!tense) {
                break;
            }
        }
        for (int u = 0; u < n; u++) {
            if (cycle[u] && d[u] != -INF) {
                dfs(u);
            }
        }
    }
    return tense;
}
```

# GetNegativeCycle Function

```
vector<Long> getNegativeCycle(Long u, Long n) {
    //go back n times to find a cycle
    assert(d[u] == -INF);
    for (int i = 0; i < n; i++) {
        u = parent[u];
    }

    vector<Long> cycle = {u};
    u = parent[u];
    while (u != cycle[0]) {
        cycle.push_back(u);
        u = parent[u];
    }
    return cycle;
}
```

# Moore's Improvement

Si nos fijamos en el algoritmo de Bellman-Ford, podemos notar que en cada iteración que inspecciona a todas las aristas, **no todas ellas están tensas**. ¿Y si ahorramos tiempo y solo chequeamos las tensas (o las candidatas a tensas)?

Eso es lo que hace el algoritmo que presentó Edward Moore (sí, el “creador” del BFS) en 1959 con el nombre de “Algorithm D”. Moore guarda todas las aristas en una cola y en cada relajación agrega el nodo tenso a la cola (si es que no está en ella todavía). Es decir es un caso particular del **Modified generic shortest path algorithm**.



## Algorithm 10: Moore's Algorithm

```
input :  $G(V_G, E_G, w)$  , source :  $s$ 
output:  $d[ ]$ , parent[ ]
```

- 1 Initialize ( $s$ )
- 2  $q = \emptyset$
- 3  $q.push(s)$
- 4 while  $q$  is not empty do
- 5  $u = q.pop()$
- 6 foreach edge  $u \rightarrow v$  in  $E_G$  do
- 7 if  $d(u) + w(u, v) < d(v)$  then
- 8 // is tense
- 9 Relax ( $u, v$ )
- 10 if  $v$  is not in  $q$  then
- 11 |  $q.push(v)$
- 12 end
- 13 end
- 14 end
- 15 return  $d, parent$

# Moore's Improvement - Complejidad

Al ser un refinamiento de Bellman-Ford, es sencillo demostrar que el algoritmo funciona (herencia del algoritmo genérico modificado) y que su complejidad es  $O(VE)$ . Sin embargo es siempre más eficiente que Bellman-Ford.

Es posible crear casos de grafos completos que causen lleguen a ese peor caso (ver <https://codeforces.com/blog/entry/16221?#comment-211370>), sin embargo en grafos random suele funcionar muy rápido. Tanto así que en 1994, el investigador chino Duan Fanding “redescubrió” el algoritmo y “demostró” (su demostración estaba mal) que el algoritmo corría en complejidad  $O(kE)$  donde  $k$  era una constante pequeña por lo que concluía que la complejidad era  $O(E)$  (lo cual está mal). Debido a esto él lo llamó **Shortest Path Faster Algorithm (SPFA)** y se popularizó con ese nombre.

**Nota:** Anteriormente (en la sección de BFS) presentamos este mismo algoritmo pero sin las líneas 9,10,11. Si bien el algoritmo sigue siendo correcto al quitarle estas líneas, la complejidad puede aumentar hasta  $O(VE^2)$

# SPFA - Code

```
const Long MX = 1e4;
const Long INF = 1e18;

struct Graph{
    vector<pair<Long, Long>> adj [MX];
    Long d[MX];
    Long parent[MX];

    void clear(Long n) {
        for (Long i = 0; i < n; i++) {
            adj[i].clear();
        }
    }

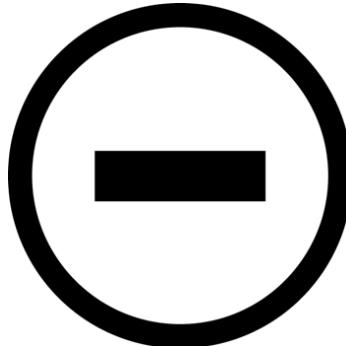
    void addEdge(Long u, Long v, Long w) {
        adj[u].push_back({v, w});
    }
}
```

```
void spfa(Long s, Long n){ //O(VE)
    for(Long i = 0; i < n; i++){
        d[i] = INF;
        parent[i] = -1;
    }
    queue<Long> q;
    vector<bool> inQueue(n , false);
    d[s] = 0;
    inQueue[s] = true;
    q.push(s);
    while(!q.empty()){
        Long u = q.front();
        q.pop();
        inQueue[u] = false;
        for(auto e : adj[u]){
            Long v = e.first;
            Long w = e.second;
            if(d[u] + w < d[v]){
                d[v] = d[u] + w;
                parent[v] = u;
                if(!inQueue[v]){
                    q.push(v);
                    inQueue[v] = true;
                }
            }
        }
    }
} G;
```

# SPFA – Negative Cycle

El código anterior no es capaz detectar ciclos negativos (entraría en bucle infinito). Para poder detectarlos y romper el bucle while, podemos mantener un contador de cuántas veces un nodo sale de la cola. Si un nodo entra en la cola  $V$  veces, significa que hubo un ciclo negativo en alcanzable desde  $s$ .

También podemos hallar los nodos que tienen  $\delta(u) = -\infty$  de la misma forma en que lo hallamos en el algoritmo de Bellman-Ford, utilizando un *dfs* desde todos aquellos nodos que estén tensos luego de haber cortado el bucle.



# SPFA – Code Negative Cycle

```
bool spfa(Long s, Long n){ //O(VE)
    for(Long i = 0; i < n; i++){
        d[i] = INF;
        parent[i] = -1;
    }
    queue<Long> q;
    vector<bool> inQueue(n , false);
    vector<Long> cnt(n, 0);
    d[s] = 0;
    inQueue[s] = true;
    q.push(s);
    bool negativeCycle = false;
    while(!q.empty()){
        Long u = q.front();
        q.pop();
        inQueue[u] = false;
        cnt[u]++;
        if(cnt[u] >= n){
            negativeCycle = true;
            break;
        }
        for(auto e : adj[u]){
            Long v = e.first;
            Long w = e.second;
            if(d[u] + w < d[v]){
                d[v] = d[u] + w;
                parent[v] = u;
                if(!inQueue[v]){
                    q.push(v);
                    inQueue[v] = true;
                }
            }
        }
    }
}
```

```
if (!negativeCycle) {
    return false;
}
vector<bool> cycle(n , false);
for (int u = 0; u < n; u++) {
    for (auto e : adj[u]) {
        Long v = e.first;
        Long w = e.second;
        if (d[u] != INF && d[u] + w < d[v]) {
            cycle[v] = true;
        }
    }
}
for (int u = 0; u < n; u++) {
    if (cycle[u] && d[u] != -INF) {
        dfs(u);
    }
}
return true;
}
```

# Aplicaciones – Difference constraints

**Problema:** Dar alguna solución para el siguiente sistema de inecuaciones o decir que no tiene solución:

$$x_1 - x_2 \leq 5$$

$$x_1 - x_3 \leq -6$$

$$x_3 - x_2 \leq 4$$

$$x_4 - x_5 \leq 7$$

**Problema general:** Se tienen  $n$  incógnitas y  $m$  constraints. Hallar alguna solución del sistema de inecuaciones o decir que no hay solución. Cada constraint es de la forma :

$$x_l - x_r \leq b_k, \quad 1 \leq l, r \leq n \quad l \neq r \quad 1 \leq k \leq m$$

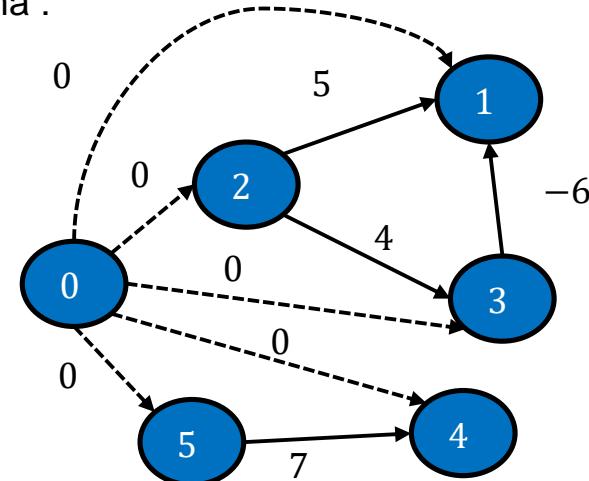
**Transformándolo en grafos:** Definamos el grafo  $G(V, E, w)$  como el “grafo de constraints” del sistema anterior, como un grafo de  $n + 1$  nodos y  $m$  aristas y construyámoslo de la siguiente forma:

$$V = \{s, 1, 2, \dots, n\}$$

$$E = \{(r, l) \mid x_l - x_r \leq b_k \text{ es una constraint}\} \cup \{(s, i), \forall i \in [1, n]\}$$

$$w(r, l) = b_k, \text{ para toda constraint } x_l - x_r \leq b_k$$

$$w(s, i) = 0, \forall i \in [1, n]$$



# Aplicaciones – Difference constraints

- **Lema 6.4:** Si el grafo de constraints tiene algún ciclo negativo, entonces el sistema de difference constraints no tiene solución.

## Demostración:

Por contradicción, suponga que exista solución y que existiera algún ciclo negativo. Suponga que el ciclo negativo es  $c = v_1, v_2, v_3, \dots, v_k$  con  $v_1 = v_k$ ,  $w(c) < 0$ . Es claro que  $s$  no puede pertenecer al ciclo ya que  $s$  no tiene aristas entrantes. Cada arista del ciclo define una constraint:

$$v_2 - v_1 \leq w(v_1, v_2)$$

$$v_3 - v_2 \leq w(v_2, v_3)$$

$$v_4 - v_3 \leq w(v_4, v_3)$$

...

$$v_{k-1} - v_{k-2} \leq w(v_{k-2}, v_{k-1})$$

$$v_1 - v_{k-1} \leq w(v_{k-1}, v_1)$$

Al sumar telescopíicamente las inecuaciones, todos los términos de la izquierda se eliminan por lo que queda:

$$0 \leq w(v_1, v_2) + w(v_2, v_3) + \dots + w(v_{k-1}, v_1) = w(c)$$

Lo cual es una contradicción ya que el ciclo era de peso negativo. ■

# Aplicaciones – Difference constraints

- **Lema 6.5:** Si el grafo de constraints no tiene ciclos negativos, entonces existe solución y una de ellas es  $x_u = \delta(s, u) = \delta(u)$ ,  $\forall u \in [1, n]$

## Demostración:

Como  $s$  está unido con todos los nodos, entonces ningún nodo es inalcanzable. Aparte, como no hay ciclos negativos, entonces  $\delta(u) \in \mathbb{R}$ .

Supongamos que en el grafo existe una arista  $r \rightarrow l$ . Por [triangle inequality](#) tenemos que  
$$\delta(l) \leq \delta(r) + w(r, l)$$
$$\Rightarrow \delta(l) - \delta(r) \leq w(r, l)$$

Recuerde que  $w(r, l) = b_k$  para la ecuación  $x_l - x_r \leq b_k$ . Entonces  $x_l = \delta(l)$ ,  $x_r = \delta(r)$  cumplen la inecuación.

■

# Aplicaciones – Difference constraints

**Solución al problema:** Construir el grafo de constraints y correr Bellman-Ford o SPFA para hallar la respuesta en  $O(VE)$

**Observación:** Note que la solución dada por el algoritmo cumple que  $x_i \leq 0$  debido al nodo artificial fuente que agregamos con aristas de peso 0 hacia todos los demás vértices.

¿Y si quiero soluciones positivas?

A cada  $x_i$  podemos agregarle alguna constante  $d$  (la misma para todos) y seguirá siendo un solución válida ya que si  $x_l - x_r \leq b_k \Rightarrow (x_l - d) - (x_r - d) \leq b_k$

# Aplicaciones – Minimum Mean Cycle

**Definición:** Sea un ciclo  $C = v_0, v_1, v_2, \dots, v_{k-1}, v_k$ , con  $v_0 = v_k$  se define su peso promedio como

$$\mu(C) = \frac{w(C)}{k} = \frac{\sum_{i=0}^{k-1} w(v_i, v_{i+1})}{k}$$

**Problema (Minimum Mean Cycle):** Sea un grafo dirigido con pesos  $G(V, E, w)$  cuya función de pesos es  $w: E(G) \rightarrow \mathbb{R}$ . Se pide hallar el mínimo peso promedio de un ciclo ( $\min_C \mu(C)$ ) o decir que no existen ciclos.

**Solución (Lawler 1976):** ¿Qué pasa si le restamos un valor  $x$  a todas las aristas del grafo  $G$  obteniendo un nuevo grafo  $G'$ ?

Sea  $C'$  un ciclo cualquiera en  $G'$  y sea  $C$  el mismo ciclo pero en el grafo original  $G$

$$\Rightarrow \mu(C') = \frac{w(C')}{k} = \frac{w(C) - kx}{k}$$

$$\Rightarrow \mu(C') = \mu(C) - x \dots (1)$$

- **Caso 1: Existe algún ciclo negativo**

Sea  $C'$  un ciclo negativo.  $\mu(C') < 0$

Utilizando (1):  $\mu(C) < x$

- **Caso 2: No existe ciclos negativos**

Para todo ciclo  $C'$ .  $\mu(C') \geq 0$

Utilizando (1):  $\mu(C) \geq x \quad \forall \text{ciclo } C \text{ de } G$

El caso 1 significa que  $x$  es estrictamente mayor que la respuesta, mientras que en el caso 2  $x$  es menor o igual que la respuesta. ¡Podemos usar binary search!

# Aplicaciones – Minimum Mean Cycle

**Solución (Lawler 1976):** De la diapositiva anterior, podemos concluir que el Minimum Mean Cycle es el menor  $x$  tal que al restarle  $x$  a todas las aristas del grafo, no existan aristas negativas.

Es simplemente hacer un binary search (en un rango **continuo**) y en cada función *check* podemos usar Bellman-Ford o SPFA para detectar ciclos negativos.

Lo último que faltaría es hallar los límites del binary search y la complejidad total del algoritmo.

**¿Cuál es el máximo  $x$  aceptable para probar?** Sería el valor extremo que nos permite llegar al caso 1 (existencia de algún ciclo negativo). El peor caso sería volver todas las aristas negativas para “forzar” a que exista un ciclo negativo transformando todas las aristas en negativas, lo cual se puede lograr restándole a todas las aristas el **máximo peso + 1** ya que  $w_i - \max w_i \leq 0$ . Si aún con aristas negativas, el grafo no tiene un ciclo negativo, significa que no tiene ningún ciclo (es un DAG).

**¿Cuál es el mínimo  $x$  aceptable para probar?** Sería el valor extremo que nos permita llegar al caso 2 (sin ciclos negativos). Con un razonamiento similar, podemos forzar a que todas las aristas sean positivas restándole a todas el **mínimo peso**.

**Complejidad:** Suponiendo que  $W$  es el máximo valor absoluto de los pesos y que se quiere una precisión de  $10^{-d}$ , la complejidad es  $O(VE \times \log(W \times 10^d))$

# Contenido

1. Introducción	➡
2. Propiedades de los Shortest Path y Algoritmo Genérico	➡
3. Single Source Shortest Path en DAG	➡
4. Single Source Shortest Path con pesos unitarios	➡
5. Single Source Shortest Path con pesos no negativos	➡
6. Single Source Shortest Path con cualquier peso real	➡
<b>7. All-pairs shortest path</b>	➡

# All Pairs Shortest Path (APSP)

**Problema:** Sea un grafo dirigido con pesos  $G(V, E, w)$ . Cuya función de pesos es  $w: E(G) \rightarrow \mathbb{R}$ . Se pide hallar la longitud de los shortest path entre cada par de vértices  $u, v$  denotado como  $\delta(u, v)$ .

**Solución inicial:** Es fácil notar que podemos resolver APSP llamando a algún algoritmo de SSSP tomando como fuente a cada vértice, pero ¿cuánto sería la complejidad de eso? (asumamos  $V < E$ )

Función de pesos	Algoritmo	Complejidad APSP	
		General	Grafo denso
Pesos unitarios	BFS	$O(VE)$	$O(V^3)$
Pesos 0 o 1	BFS 01	$O(VE)$	$O(V^3)$
Pesos no negativos	Dijkstra	$O(VE \log V)$	$O(V^3 \log V)$
	Dijkstra Dense	$O(V^3)$	$O(V^3)$
Cualquier peso	Bellman Ford	$O(V^2 E)$	$O(V^4)$
	SPFA	$O(V^2 E)$	$O(V^4)$

Muy lento

# Floyd-Warshall Algorithm

En 1962 Stephen Warshall publicó un algoritmo para resolver un problema de matrices que, al generalizarse, podía resolver el problema de APSP. Esta generalización, fue dada en el algoritmo propuesto por Robert Floyd en ese mismo año.

El algoritmo de Warshall fue publicado antes por Bernard Roy en 1959, e incluso tiene cierta similitud con el algoritmo de Stephen Kleene publicado en 1956. Sin embargo el algoritmo que presentaremos para resolver el APSP es generalmente conocido como **Floyd-Warshall Algorithm**.



*Robert Floyd*



*Stephen Warshall*

# Floyd-Warshall Algorithm

El algoritmo utiliza **programación dinámica**. Inicialmente supongamos que no existen ciclos negativos. Y digamos que los nodos están numerados del 1 al  $|V|$ . Para facilitar la explicación también supongamos que no existen aristas múltiples ni self-loops (en el código veremos cómo trabajar con esto).

Sea  $d(u, v, k)$  el peso del shortet path que va desde  $u$  a  $v$  y utiliza solo nodos intermedios cuyo número sea  $\leq k$  (los nodos  $u, v$  no tienen esta restricción). Lo que nosotros queremos es  $d(u, v, |V|)$

- **Caso base:**

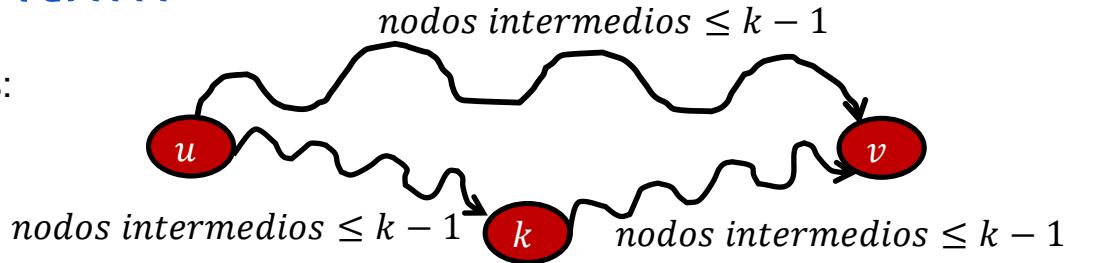
$k = 0$ , es decir los paths no tienen ningún nodo intermedio, por lo tanto

$$d(u, v, 0) \left\{ \begin{array}{ll} 0, & \text{si } u = v \\ w(u, v), & \text{si existe la arista } (u, v) \\ +\infty, & \text{en cualquier otro caso} \end{array} \right.$$

# Floyd-Warshall Algorithm

- **Recurrencia:**

Si queremos  $d(u, v, k)$  tenemos dos casos:



1) El shortest path que va de  $u$  a  $v$  con nodos intermedios  $\leq k$  **no** utiliza el nodo  $k$ .

En este caso  $d(u, v, k) = d(u, v, k - 1)$

2) El shortest path que va de  $u$  a  $v$  con nodos intermedios  $\leq k$  **sí** utiliza al nodo  $k$ .

En este caso podemos dividir el path  $u \rightsquigarrow v$  en  $(u \rightsquigarrow k) + (k \rightsquigarrow v)$ . Note que como no hay ciclos negativos, el nodo  $k$  no puede estar como nodo intermedio en  $u \rightsquigarrow k$  ni en  $k \rightsquigarrow v$  (porque sino habría un ciclo, lo cual empeora o deja igual a la respuesta). Además por **optimal substructure**  $u \rightsquigarrow k$  y  $k \rightsquigarrow v$  también son shortest path.

Entonces  $d(u, v, k) = d(u, k, k - 1) + d(k, v, k - 1)$

Juntando ambos casos:

$$\Rightarrow d(u, v, k) = \min(d(u, v, k - 1), d(u, k, k - 1) + d(k, v, k - 1))$$

# Floyd-Warshall Algorithm

Lo anterior nos da un algoritmo simple de complejidad en tiempo  $O(V^3)$  y complejidad en memoria  $O(V^3)$ , lo cual es mejor que hacer Bellman-Ford o spfa  $V$  veces (no es mejor que hacer dijkstra o bfs  $V$  veces, pero en caso de los grafos densos, llega al mismo resultado).

La parte principal del algoritmo (luego de la inicialización) sería algo así:

```
for (int k = 1; k <= n; k++) {
    for (int u = 1; u <= n; u++) {
        for (int v = 1; v <= n; v++) {
            if (d[u][k][k - 1] == INF || d[k][v][k - 1] == INF) {
                d[u][v][k] = d[u][v][k - 1];
            } else {
                d[u][v][k] = min(d[u][v][k - 1], d[u][k][k - 1] + d[k][v][k - 1]);
            }
        }
    }
}
```

# Floyd-Warshall Algorithm - Optimization

Podemos hacer una última optimización sencilla en la **memoria**. Un truco muy común en dp's es que, en ciertos casos especiales, puedes eliminar una dimensión.

En este caso podemos **eliminar la 3ra dimensión** (el  $k$ ) y dejar la memoria en  $O(V^2)$ . ¿Por qué? Porque en cada iteración solo nos importa la fase  $k - 1$  y la fase  $k$  (la actual y la previa). Supongamos que transformamos  $d(u, v, k) = \min(d(u, v, k - 1), d(u, k, k - 1) + d(k, v, k - 1))$   
en  
 $d(u, v) = \min(d(u, v), d(u, k) + d(k, v))$

Esta transformación es válida debido a que, si no hay ciclos negativos, se cumple que  
 $d(u, k, k - 1) = d(u, k, k) \dots (1)$  y que  $d(k, v, k - 1) = d(k, v, k) \dots (2)$

Un dato curioso es que  $d(u, v) = \min(d(u, v), d(u, k) + d(k, v))$  es equivalente a  
*if*( $d(u, k) + d(k, v) < d(u, v)$ ){  
     $d(u, v) = d(u, k) + d(k, v)$   
}

¿Te suena?

# Floyd-Warshall Algorithm - Optimization

Podemos hacer una última optimización sencilla en la **memoria**. Un truco muy común en dp's es que, en ciertos casos especiales, puedes eliminar una dimensión.

En este caso podemos **eliminar la 3ra dimensión** (el  $k$ ) y dejar la memoria en  $O(V^2)$ . ¿Por qué? Porque en cada iteración solo nos importa la fase  $k - 1$  y la fase  $k$  (la actual y la previa). Supongamos que transformamos  $d(u, v, k) = \min(d(u, v, k - 1), d(u, k, k - 1) + d(k, v, k - 1))$   
en  
 $d(u, v) = \min(d(u, v), d(u, k) + d(k, v))$

Esta transformación es válida debido a que, si no hay ciclos negativos, se cumple que  
 $d(u, k, k - 1) = d(u, k, k) \dots (1)$  y que  $d(k, v, k - 1) = d(k, v, k) \dots (2)$

Un dato curioso es que  $d(u, v) = \min(d(u, v), d(u, k) + d(k, v))$  es equivalente a  
*if*( $d(u, k) + d(k, v) < d(u, v)$ ){  
     $d(u, v) = d(u, k) + d(k, v);$   
}

¿Te suena? **Relajación**



# Floyd-Warshall Algorithm – Implementation Details

Ciclos negativos:



Finalmente, ¿qué sucede si hay ciclos negativos?. Para aquellos pares de nodos  $(u, v)$  en los cuales no exista ningún ciclo negativo en ningún camino que vaya de  $u$  a  $v$ , entonces  $d(u, v)$  tendrá el valor correcto al terminar Floyd-Warshall ( $d(u, v) = \delta(u, v)$ ). Los demás pares de nodos tendrán  $\delta(u, v) = -\infty$  y su  $d(u, v)$  tendrá cualquier cosa, ¿entonces cómo los identificamos?

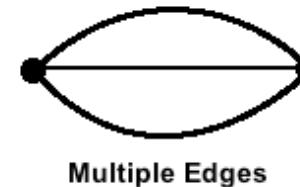
Empecemos por algo más sencillo, identificar a los nodos  $x$  que pueden “alcanzar” a algún ciclo negativo. Estos nodos, al finalizar Floyd-Warshall, tendrán  $d(x, x) < 0$ .

Finalmente para detectar si  $\delta(u, v) = -\infty$  tengo que buscar si existe algún nodo  $x$  tal que  $d(x, x)$  y que exista camino desde  $u$  a  $x$  y exista camino desde  $x$  a  $v$

**Múltiples aristas y self-loops:**

¿Qué hago si hay múltiples aristas entre  $(u, v)$ ? Toma la de menor peso.

¿Qué hago si hay self-loops  $(u, u)$ ? Si el peso es positivo, no afecta. Si pesos negativos para  $(u, u)$ , coge el menor peso positivo (nota que esto occasionará ciclos negativos).



Multiple Edges



Loop

# Floyd-Warshall Algorithm – Implementation Details

Reconstrucción del shortest path:



Aparte de la distancia, ¿podemos reconstruir el shortest path?

Necesitaremos un arreglo  $parent(u, v)$  que sería el nodo anterior a  $v$  en el camino  $u \rightsquigarrow v$ . Inicialmente podemos setear a todos los parent en -1. Luego cada vez que se agregue una arista  $(u, v)$  al grafo podemos hacer  $parent(u, v) = u$ . Finalmente en cada relajación que hacemos, además de hacer  $d(u, v) = d(u, k) + d(k, v)$ , también podemos hacer  $parent(u, v) = parent(k, v)$ .

Podemos usar los parent para elaborar nuestras funciones *getPath* y *getNegativeCycle* de la misma forma que lo hicimos en Bellman-Ford.

## Overflow:

Si existen ciclos negativos, en este algoritmo las distancias pueden crecer de forma negativa exponencialmente (a diferencia de Bellman-Ford que a lo mucho llegan a ser  $-E * V * W$ , donde  $W$  es el máximo peso en valor absoluto). Por lo tanto es conveniente limitar a  $d(u, v)$  como  $-\infty$



# Floyd-Warshall Algorithm – Code

Esta implementación funciona aún si hay aristas múltiples o self-loops.

**Nota:** Llamar a initialize antes de agregar las aristas.

**Nota 2:** Indexamos desde 0 pero igual funciona

```
const int MX = 500;
const Long INF = 1e18;
struct Graph{
    Long d[MX][MX];
    Long parent[MX][MX];

    void initialize(Long n) {
        for (int u = 0; u < n; u++) {
            for (int v = 0; v < n; v++) {
                d[u][v] = INF;
                parent[u][v] = -1;
            }
            d[u][u] = 0;
        }
    }

    void addEdge(Long u, Long v, Long w) {
        d[u][v] = min(d[u][v], w);
    }
}
```

```
void floydWarshall(Long n) { //O(V^3)
    for (int k = 0; k < n; k++) {
        for (int u = 0; u < n; u++) {
            for (int v = 0; v < n; v++) {
                if (d[u][k] == INF) continue;
                if (d[k][v] == INF) continue;
                if (d[u][k] + d[k][v] < d[u][v]) {
                    d[u][v] = d[u][k] + d[k][v];
                    parent[u][v] = parent[k][v];
                }
            }
        }
    }
    //negative cycles
    for (int u = 0; u < n; u++) {
        for (int v = 0; v < n; v++) {
            for (int k = 0; k < n; k++) {
                if (d[k][k] < 0 && d[u][k] != INF && d[k][v] != INF) {
                    d[u][v] = -INF;
                }
            }
        }
    }
}G;
```

# Floyd-Warshall Algorithm – getPath

```
vector<Long> getPath (Long u, Long v) {
    if (d[u][v] == INF) {
        return { };
    }
    vector<Long> path;
    while (v != -1) {
        path.push_back(v);
        v = parent[u][v];
    }
    reverse(path.begin(), path.end());
    return path;
}
```

# Floyd-Warshall Algorithm – getNegativeCycle

```
vector<Long> getNegativeCycle (Long u, Long v, Long n) {
    //go back n times to find a cycle
    assert(d[u][v] == -INF);
    for (int i = 0; i < n; i++) {
        v = parent[u][v];
    }

    vector<Long> cycle = {v};
    v = parent[u][v];
    while (v != cycle[0]) {
        cycle.push_back(v);
        v = parent[u][v];
    }
    return cycle;
}
```

# Referencias

- ❑ Algorithms. Jeff Erickson: <https://courses.engr.illinois.edu/cs374/fa2019/A/notes/08-sssp.pdf>
- ❑ Introduction to Algorithms. CLRS. Chapters 22, 24, 25, 29
- ❑ On the history of the shortest path problem. A. Schrijver.  
<http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.451.4085&rep=rep1&type=pdf>
- ❑ L.R. Ford. Network flow theory. RAND Corporation.
- ❑ Misa, Thomas. (2010). An Interview with Edsger W. Dijkstra. Commun. ACM. 53. 41-47. 10.1145/1787234.1787249.
- ❑ Eugène L. Lawler (1976): Combinatorial optimization: Networks and Matroids. Dover Publications