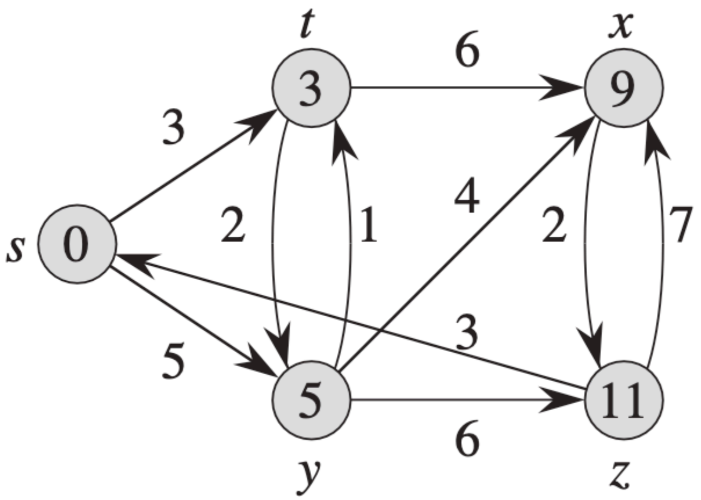
Grafos: Cómputo de caminos más cortos

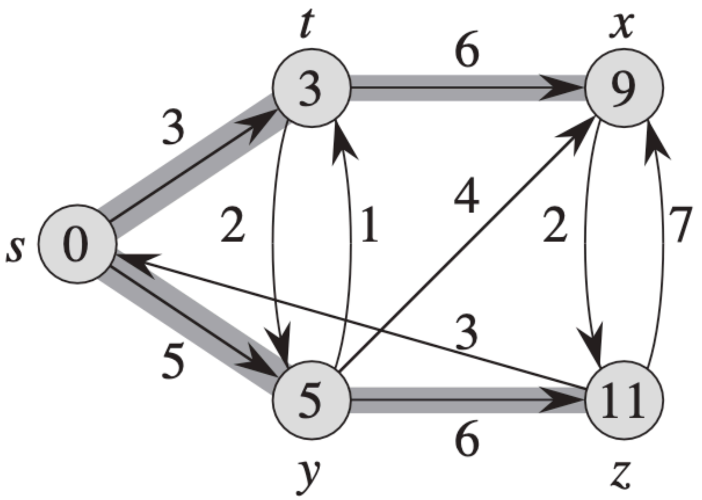
Estructuras de Datos y Algoritmos /
Algoritmos y Estructuras de Datos II
Año 2025
Dr. Pablo Ponzio
Universidad Nacional de Río Cuarto
CONICET

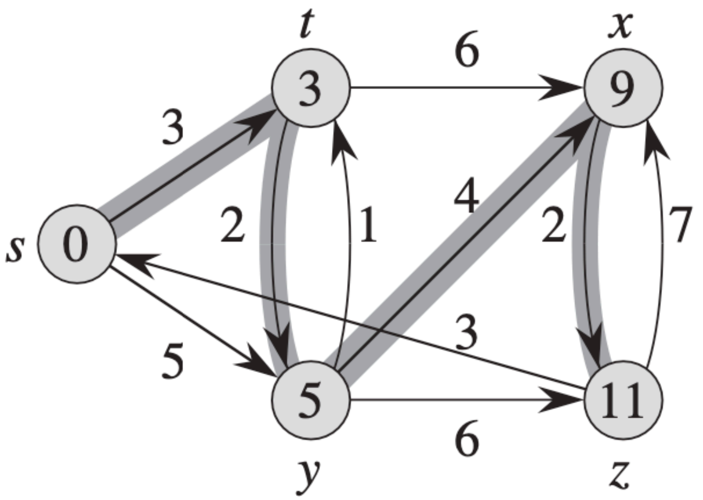
CREER... CREAR... CRECER

Problema de los caminos más cortos en grafos

- Encontrar el camino de menor costo para llegar de un vértice a otro en un grafo
  - Las aristas del grafo tienen pesos, que representan el costo de tomar la arista

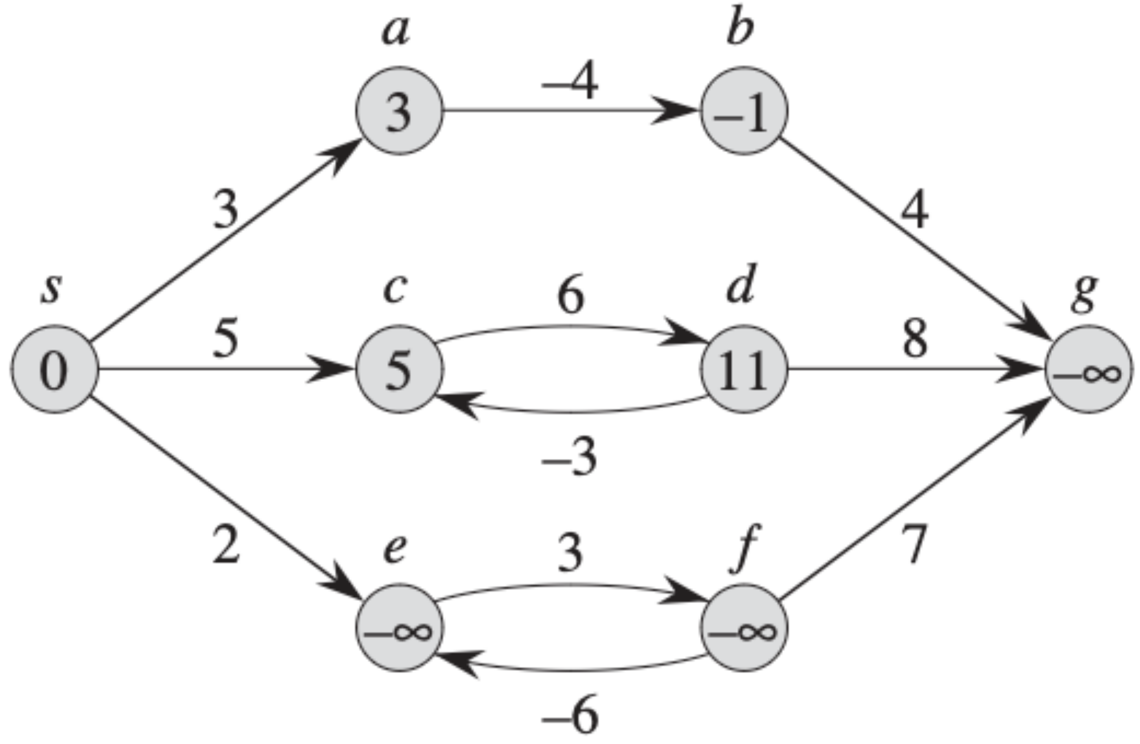






- Notar que podría haber más de una forma de llegar a un nodo con el mínimo costo, en tal caso elegiremos cualquiera de los dos caminos
- Algunas aplicaciones: Buscar caminos más cortos en mapas, buscar el camino más rápido para enviar un paquete en redes de computadoras, etc.

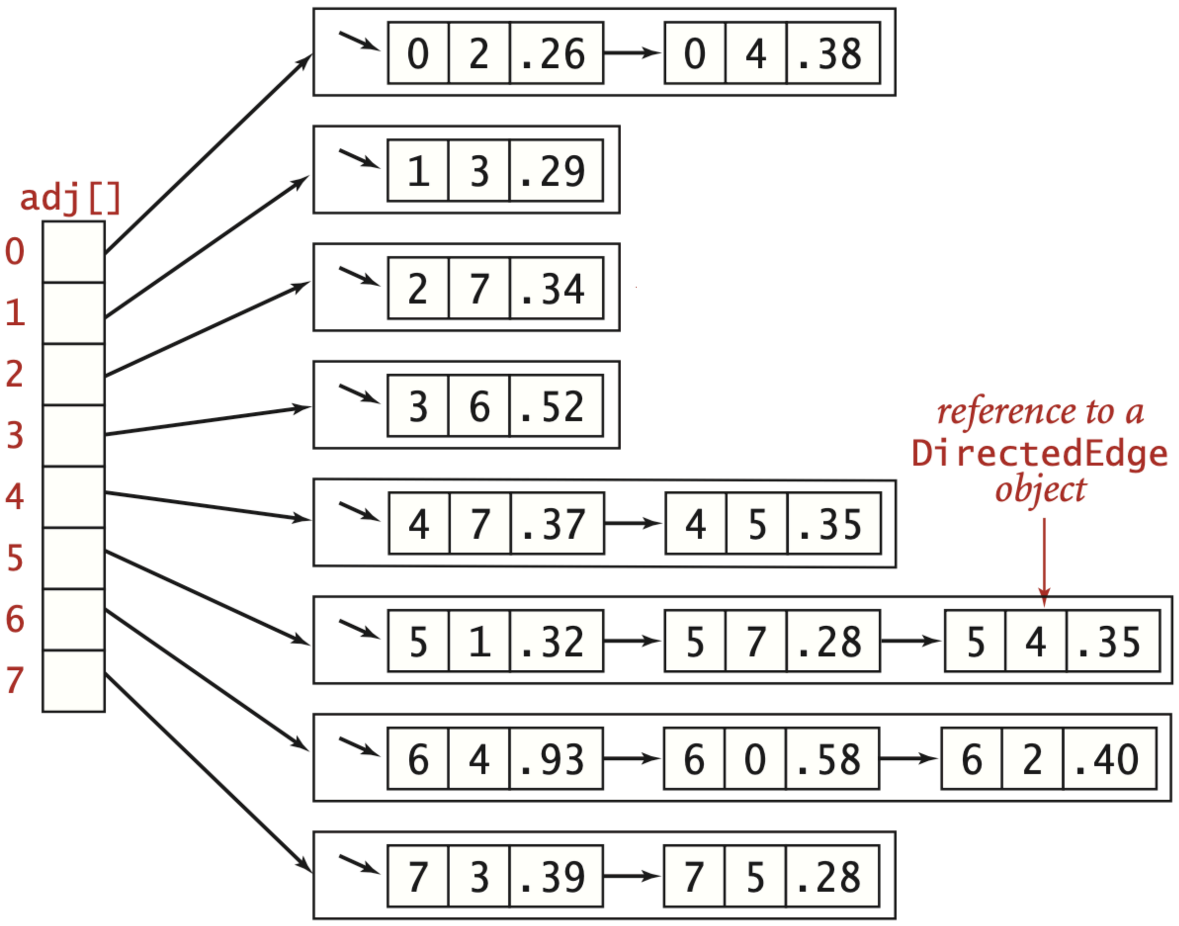
Caminos más cortos, ciclos y propiedades



- La figura muestra dentro de cada nodo la distancia del camino más corto para alcanzar el nodo a partir de s
- Las aristas y los ciclos de costo negativo pueden traer problemas
- Algunos algoritmos requieren que todas las aristas tengan costos positivos (ej. Dijkstra)
- Otros soportan aristas con costos negativos (ej. Bellman-Ford, Kruskal)
- Los ciclos de costo negativo no son soportados por ningún algoritmo y los descartaremos
  - Hay algoritmos para detectar y reportar ciclos de costo negativo (Bellman-Ford, Kruskal)
- Propiedad: Los caminos más cortos no pueden tener ciclos
  - Por lo tanto, un camino más corto tiene a lo sumo |V| vértices y |V|-1 aristas, y en los algoritmos es suficiente considerar caminos de a lo sumo |V|-1 aristas
- Propiedad: Los subcaminos de los caminos más cortos son caminos más cortos

Grafos dirigidos con pesos

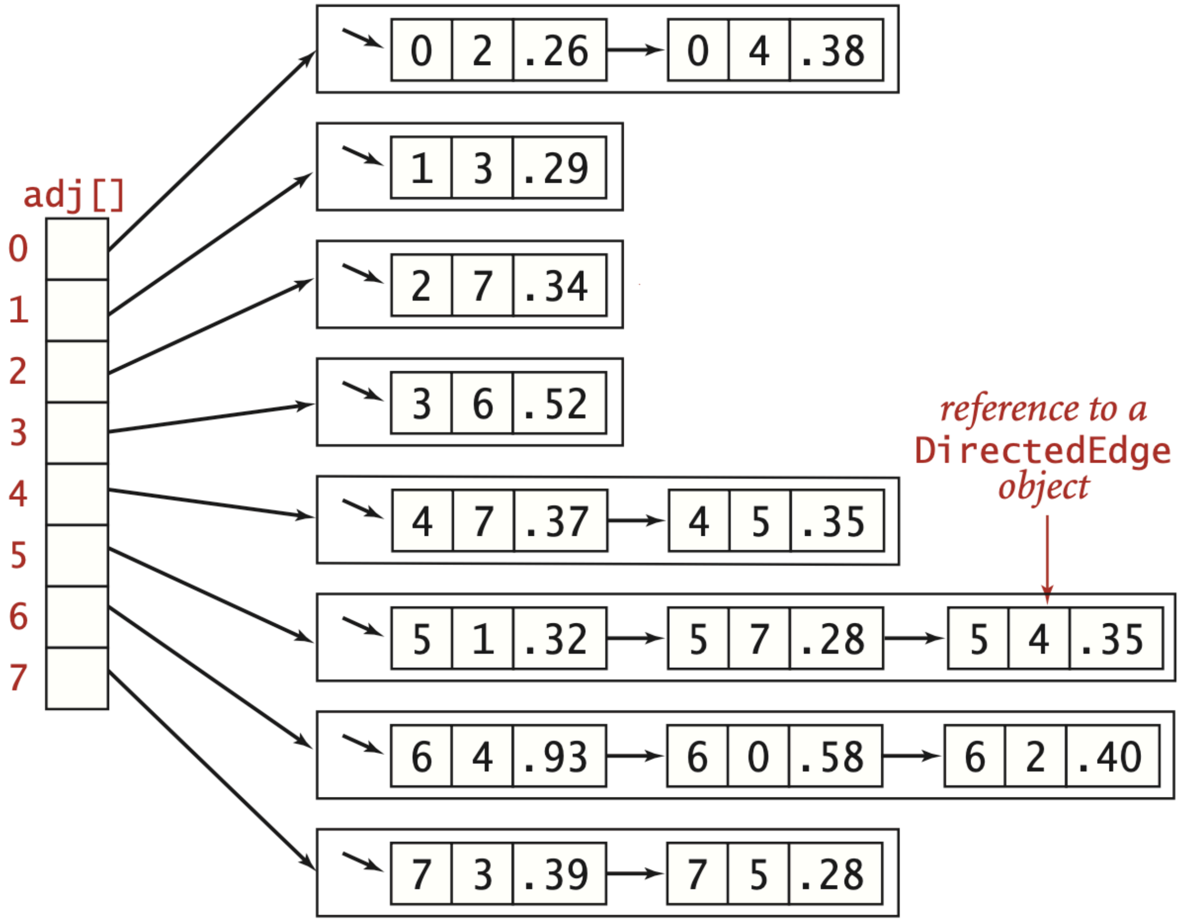
- Vamos a almacenar en las listas de adyacencia objetos de tipo DirectedEdge, que representan las aristas del grafo



```java
/**
 *  DirectedEdge class represents a
 *  weighted edge in an EdgeWeightedDigraph.
 */
public class DirectedEdge {
    final int from;
    final int to;
    final double weight;
    /**
     * @post Initializes a directed edge from vertex from
     * to vertex to with the given weight.
     */
    public DirectedEdge(int from, int to, double weight) {
        this.from = from;
        this.to = to;
        this.weight = weight;
    }
}
```

Grafos dirigidos con pesos

- Vamos a almacenar en las listas de adyacencia objetos de tipo DirectedEdge, que representan las aristas del grafo



```
/**
 *  DirectedEdge class represents a
 *  weighted edge in an EdgeWeightedDigraph.
 */
public class DirectedEdge {
    final int from;
    final int to;
    final double weight;
    /**
      * @post Initializes a directed edge from vertex from
      * to vertex to with the given weight.
      */
    public
        th
        th
        th
    }
}
```

Ejercicio: Para la representación con matrices de adyacencia podemos guardar los pesos en la matriz. Implementar grafos con pesos usando matrices de adyacencia

Grafos dirigidos con pesos

```java
/**
 *  EdgeWeightedDigraphs represents an edge-weighted
 *  digraph of vertices named 0 through V - 1, where each
 *  directed edge is of type DirectedEdge and has a real-valued weight.
 */
public class EdgeWeightedIntDigraph {
    private final int V;
    private int E;
    private List<DirectedEdge>[] adj;
    /**
     * @pre V >= 0
     * @post Initializes an edge-weighted digraph with V vertices and 0 edges.
     */
    public EdgeWeightedIntDigraph(int V) {
        if (V < 0)
            throw new IllegalArgumentException("Number of vertices in a Digraph must be non-negative");
        this.V = V;
        this.E = 0;
        adj = new LinkedList[V];
        for (int v = 0; v < V; v++)
            adj[v] = new LinkedList<DirectedEdge>();
    }
```

Grafos dirigidos con pesos

```
/**
 * @pre 0 <= e.from < V && 0 <= e.to < V
 * @post Adds the directed edge e (e.from->e.weight->e.to)
 *   to this edge-weighted digraph. */
public void addEdge(DirectedEdge e) {
    if (e.from < 0 || e.from >= V)
        throw new IllegalArgumentException("vertex " + e.from +
                "  is not between 0 and " + (V-1));
    if (e.to < 0 || e.to >= V)
        throw new IllegalArgumentException("vertex " + e.to +
                "  is not between 0 and " + (V-1));
    adj[e.from].add(e);
    E++;
}

/**
 * @pre 0 <= v < V
 * @post Returns the list of edges going out from vertex v. */
public List<DirectedEdge> adj(int v) {
    if (v < 0 || v >= V)
        throw new IllegalArgumentException("vertex " + v +
                "  is not between 0 and " + (V-1));
    return adj[v];
}
```

Grafos dirigidos con pesos

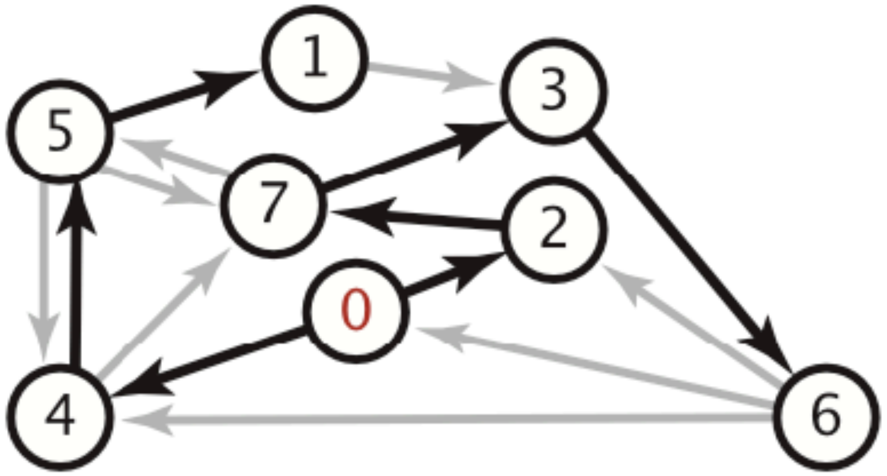
```
/**
 * @pre 0 <= e.from < V && 0 <= e.to < V
 * @post Adds the directed edge e (e.from->e.weight->e.to)
 *   to this edge-weighted digraph. */
public void addEdge(DirectedEdge e) {
    if (e.from < 0 || e.from >= V)
        throw new IllegalArgumentException("vertex " + e.from +
                " is not between 0 and " + (V-1));
    if (e.to < 0 || e.to >= V)
        throw new IllegalArgumentException("vertex " + e.to +
                " is not between 0 and " + (V-1));
    adj[e.from].add(e);
    E++;
}

/**
 * @pre 0 <= v < V
 * @post Returns the list of edges going out from vertex v. */
public List<DirectedEdge> adj(int v) {
```

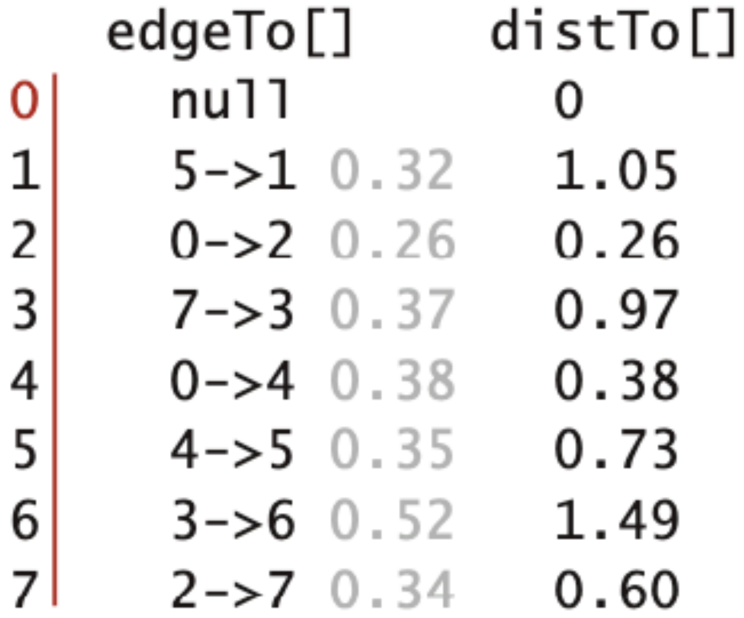
Ejercicio: Para agregar información a los nodos podemos construir dos índices aparte, como lo hicimos con los grafos no dirigidos genéricos. Implementar grafos dirigidos genéricos con pesos.

```
}
```

Estructuras de datos para almacenar caminos más cortos

- Las estructuras para guardar las distancias y los caminos más cortos son similares a las que usamos para DFS y BFS
- Vamos a usar un arreglo adicional `distTo` para almacenar la mínima distancia del vértice origen a los otros vértices del grafo
- Y un arreglo `edgeTo` para almacenar la arista con el predecesor del vértice en el camino más corto desde el origen
    - Con `edgeTo` podemos reconstruir el camino al igual que lo hacíamos en DFS y BFS

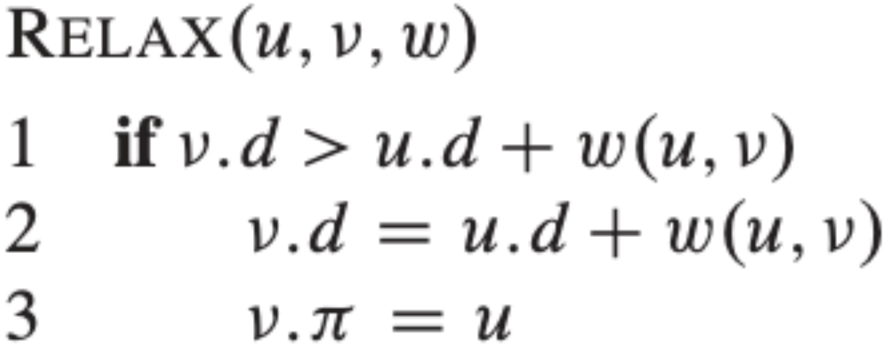




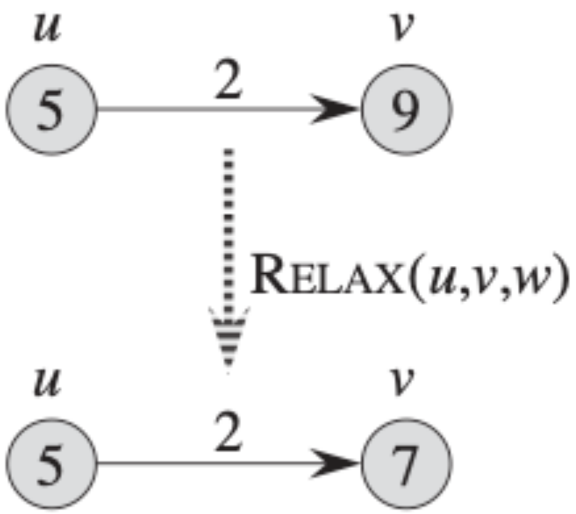
	edgeTo[]		distTo[]
0	null		0
1	5->1	0.32	1.05
2	0->2	0.26	0.26
3	7->3	0.37	0.97
4	0->4	0.38	0.38
5	4->5	0.35	0.73
6	3->6	0.52	1.49
7	2->7	0.34	0.60

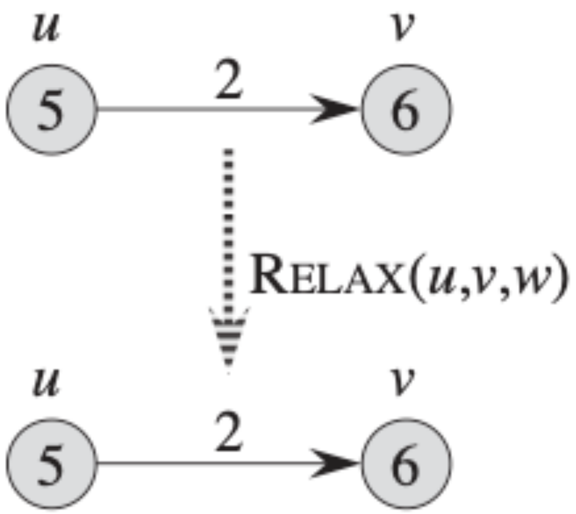
"Relajar" un arco: Pseudocódigo

- Es la operación principal de algunos de los algoritmos
- Consiste en evaluar si un arco u->v es parte del camino más corto entre s y v.
- Si lo es debemos:
  - Actualizar la distancia a v
  - Actualizar el camino para llegar a v
- Sino, no hacemos nada
- Ejemplos:



```
RELAX(u, v, w)
if $v.d > u.d + w(u, v)$
    $v.d = u.d + w(u, v)$
    $v.\pi = u$
```



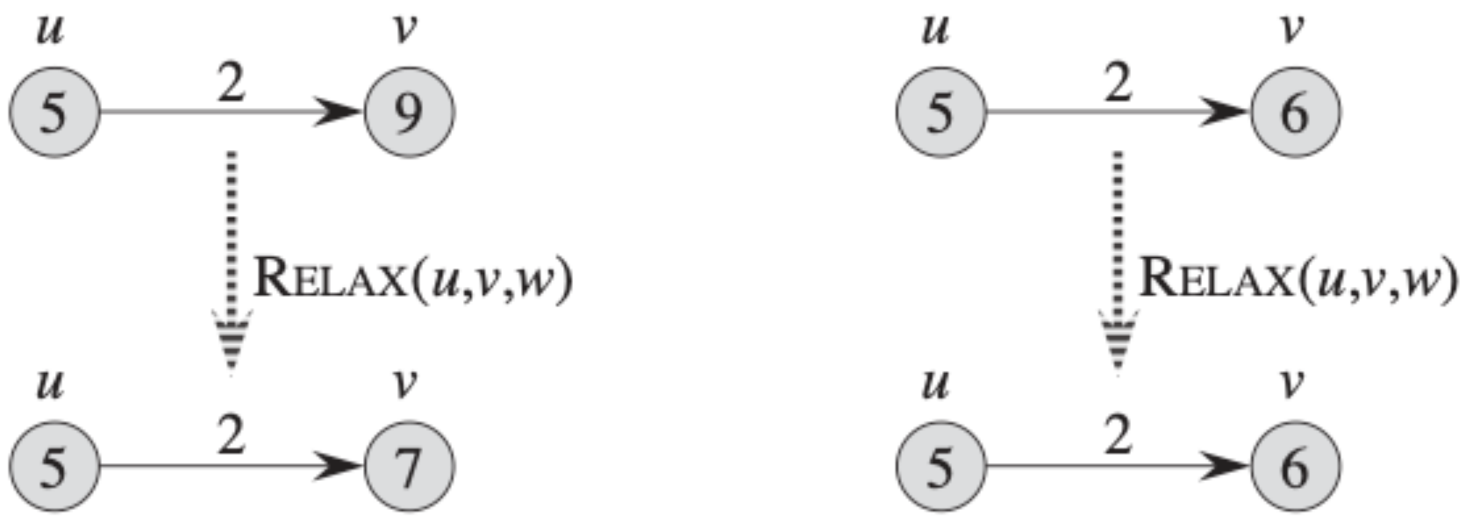


"Relajar" un arco: Pseudocódigo

- Es la operación principal de algunos de los algoritmos
- Consiste en evaluar si un arco u->v es parte del camino más corto entre s y v.
- Si lo es debemos:
  - Actualizar la distancia a v
  - Actualizar el camino para llegar a v
- Sino, no hacemos nada
- Ejemplos:

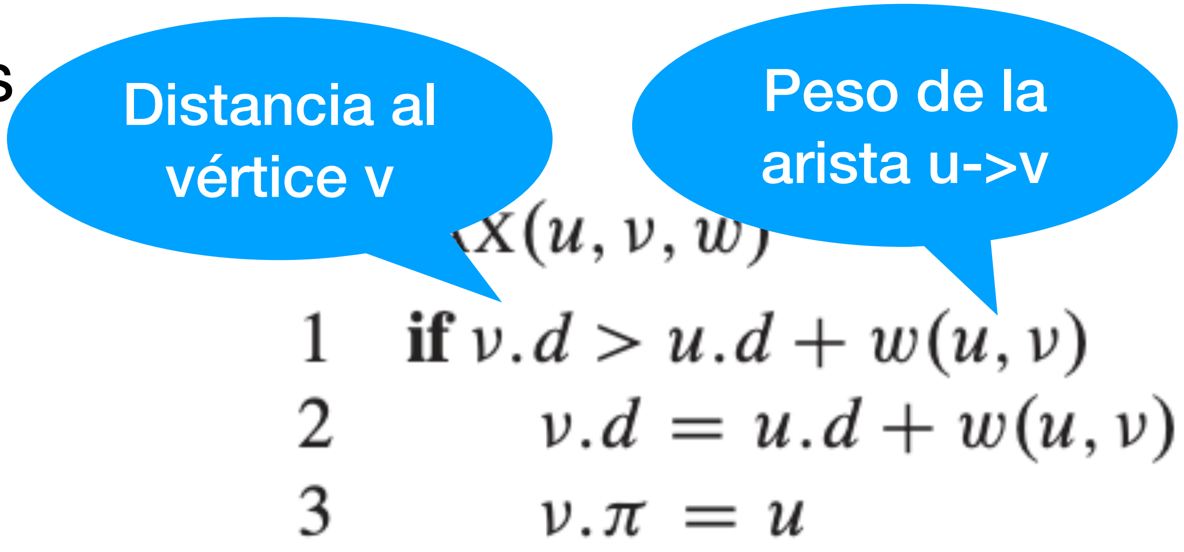
Distancia al vértice v

```
X(u, v, w)
if $v.d > u.d + w(u, v)$
    $v.d = u.d + w(u, v)$
    $v.\pi = u$
```



"Relajar" un arco: Pseudocódigo

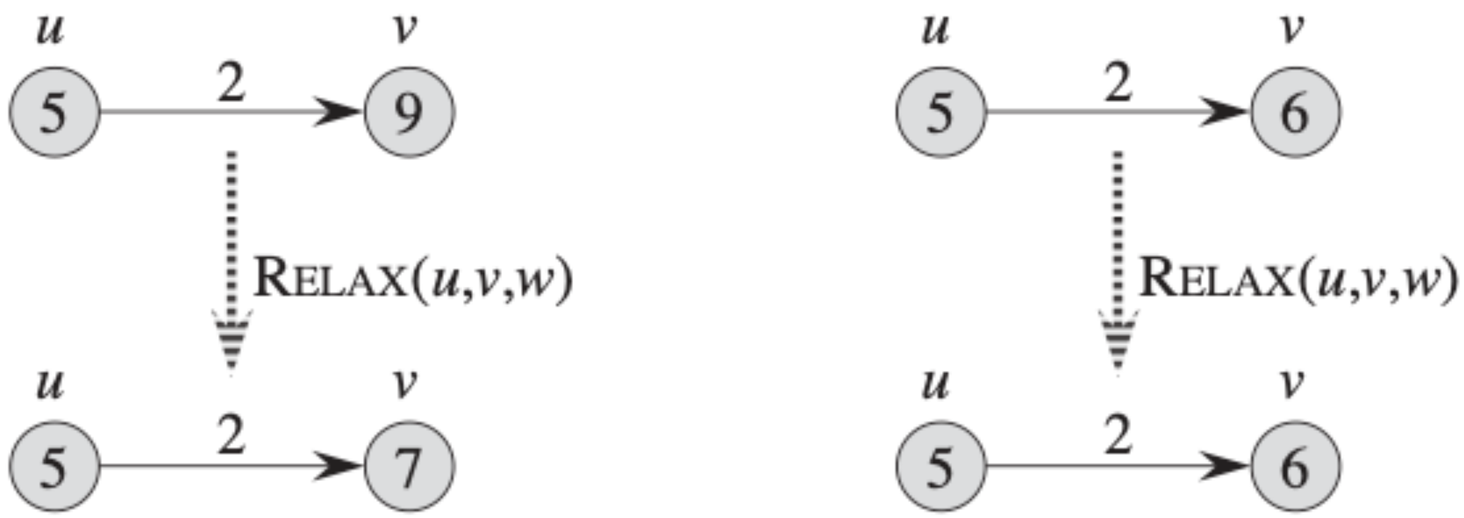
- Es la operación principal de algunos de los algoritmos
- Consiste en evaluar si un arco u->v es parte del camino más corto entre s y v.
- Si lo es debemos:
  - Actualizar la distancia a v
  - Actualizar el camino para llegar a v
- Sino, no hacemos nada
- Ejemplos:



Distancia al vértice v

Peso de la arista u->v

```
X$(u, v, w)$
if $v.d > u.d + w(u, v)$
    $v.d = u.d + w(u, v)$
    $v.\pi = u$
```



"Relajar" un arco: Pseudocódigo

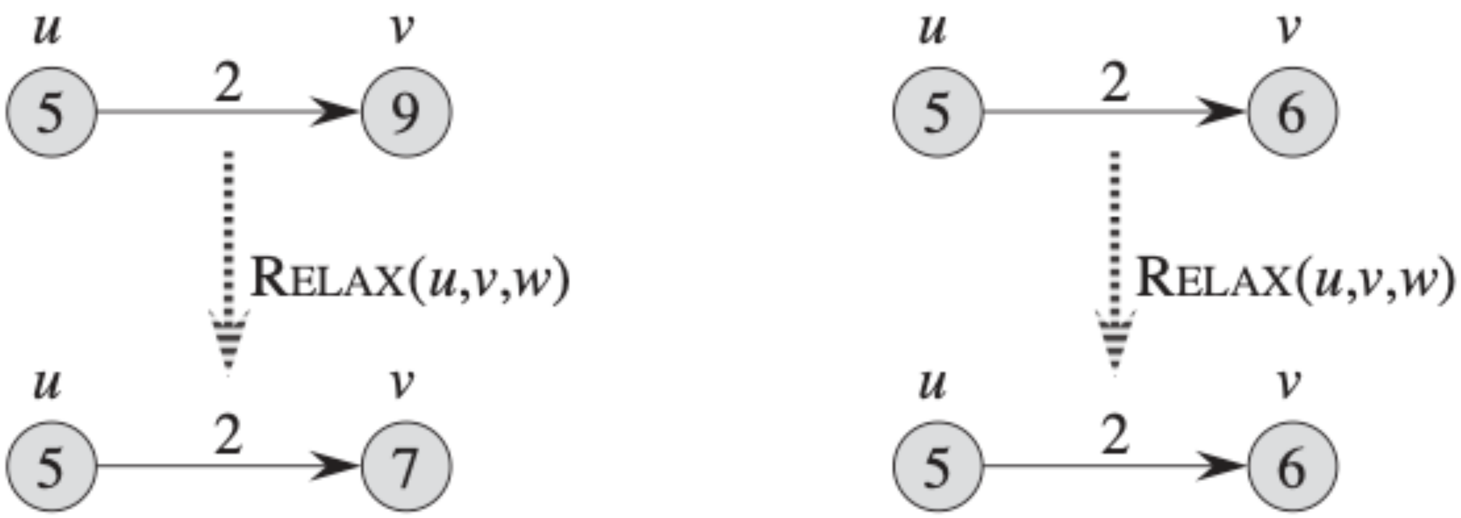
- Es la operación principal de algunos de los algoritmos
- Consiste en evaluar si un arco u->v es parte del camino más corto entre s y v.
- Si lo es debemos:
  - Actualizar la distancia a v
  - Actualizar el camino para llegar a v
- Sino, no hacemos nada
- Ejemplos:

Distancia al vértice v

Peso de la arista u->v

```
X(u, v, w)
if $v.d > u.d + w(u, v)$
    $v.d = u.d + w(u, v)$
    $v.\pi = u$
```

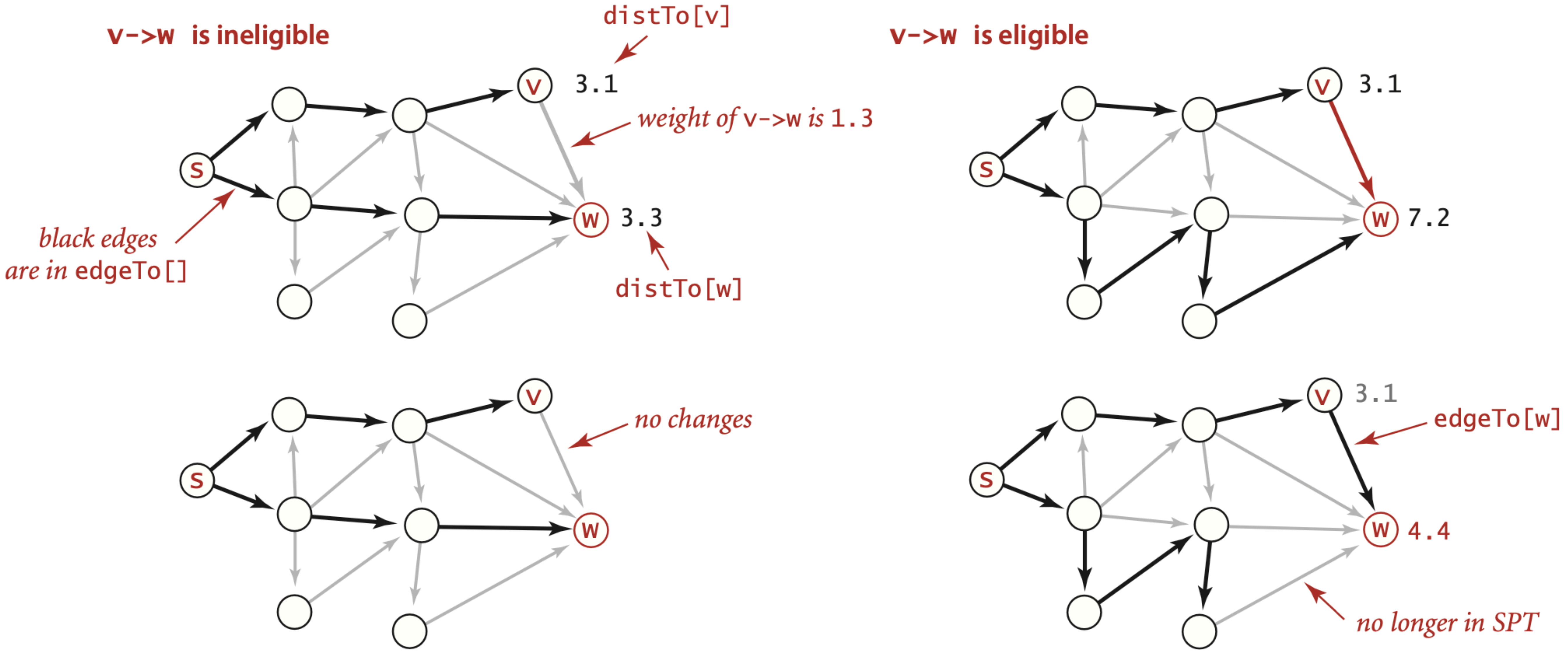
Predecesor del vértice v



"Relajar" un arco: Java

- En Java:

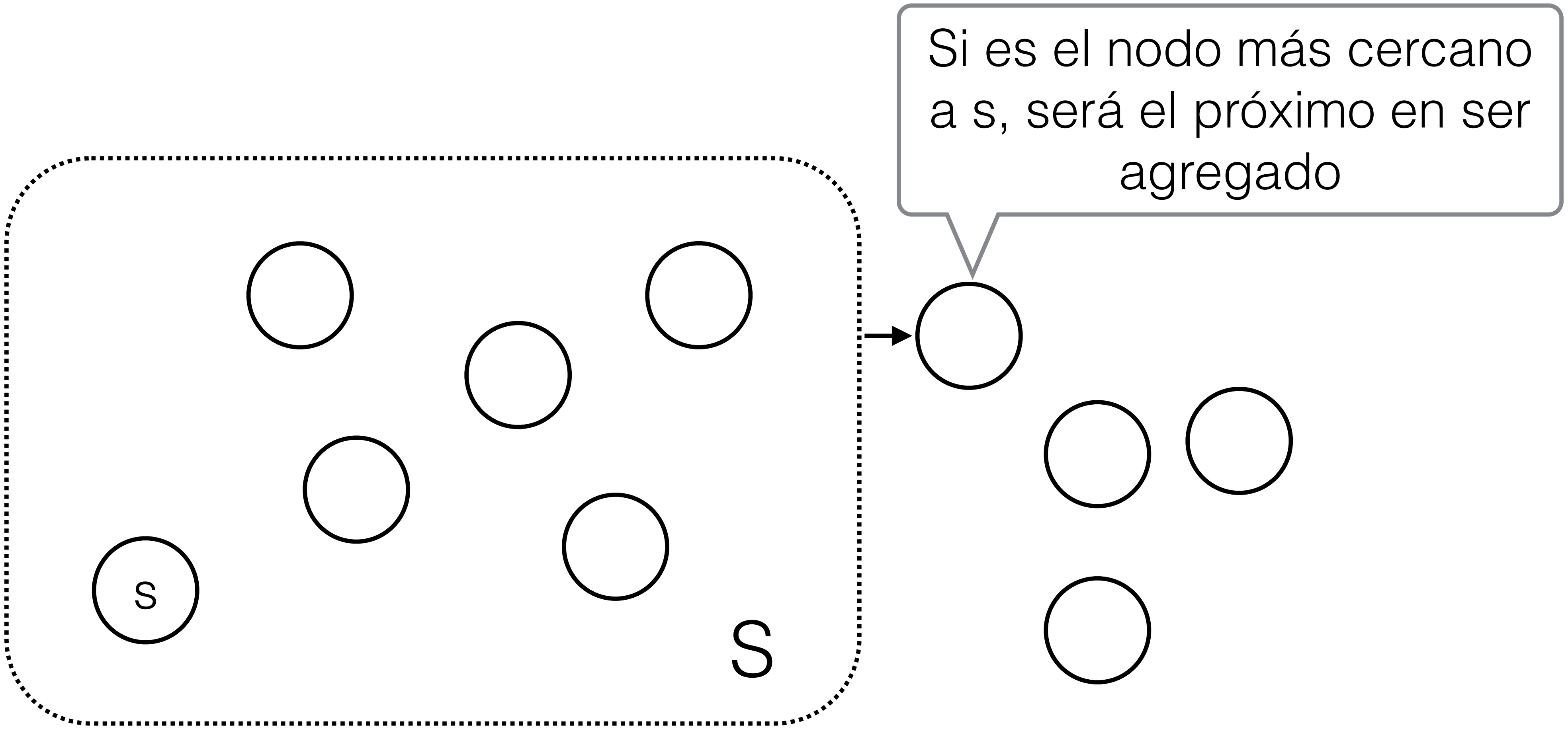
```
private void relax(DirectedEdge e)
{
   int v = e.from(), w = e.to();
   if (distTo[w] > distTo[v] + e.weight())
   {
       distTo[w] = distTo[v] + e.weight();
       edgeTo[w] = e;
   }
}
```



Algoritmo de Dijkstra

El algoritmo de Dijkstra calcula todos los caminos más cortos desde un origen s.

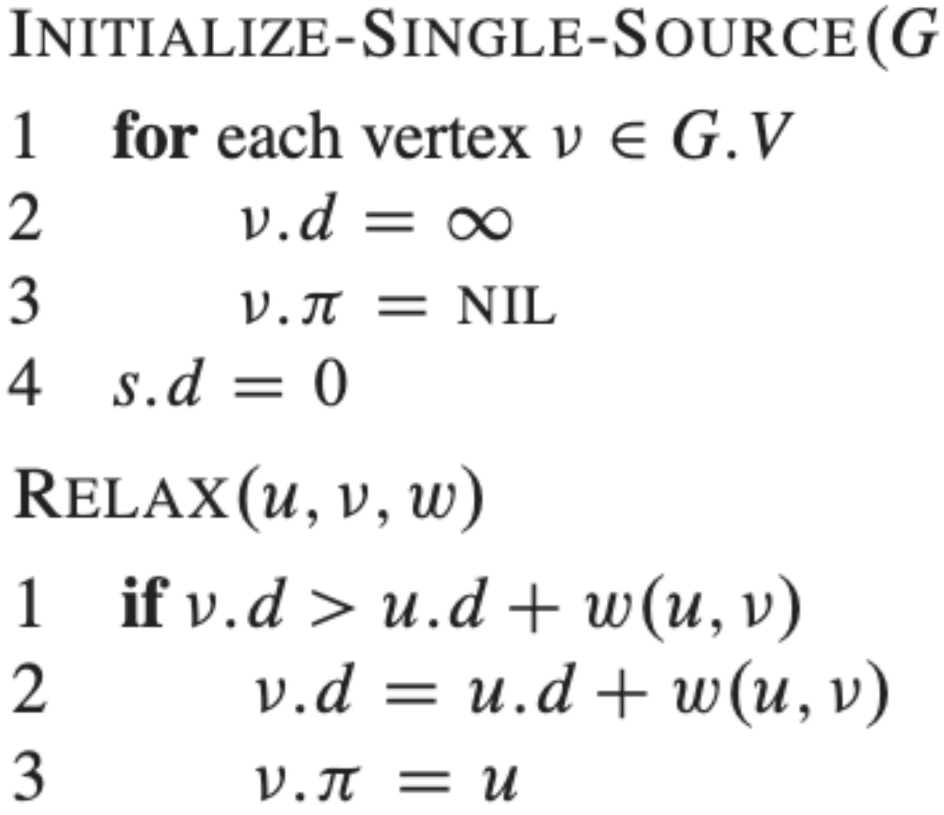
Idea:



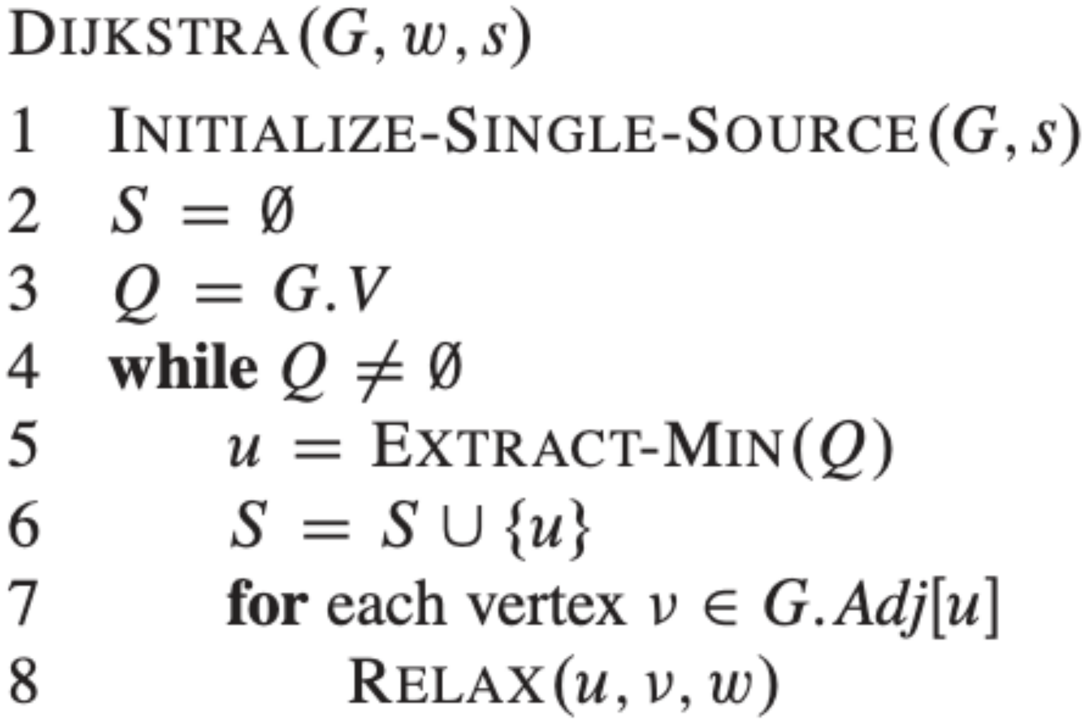
- Invariante: Los nodos en el conjunto S tienen calculado el camino más corto desde s; en cada iteración se agrega el nodo con el camino más corto a s
- Dijkstra es un algoritmo greedy: elige la solución localmente óptima
- Si no hay aristas con pesos negativos, la solución local más óptima es también la mejor solución global

Dijkstra: Pseudocódigo

- Dijkstra usa una cola de prioridad, ordenada por la distancia desde el vértice origen s
- En cada iteración, saca un vértice u de la cola de prioridad:
  - En este punto, u tiene computado el camino más corto a s, y se agrega en S
  - Luego, relaja todos los arcos de los vértices adyacentes a u

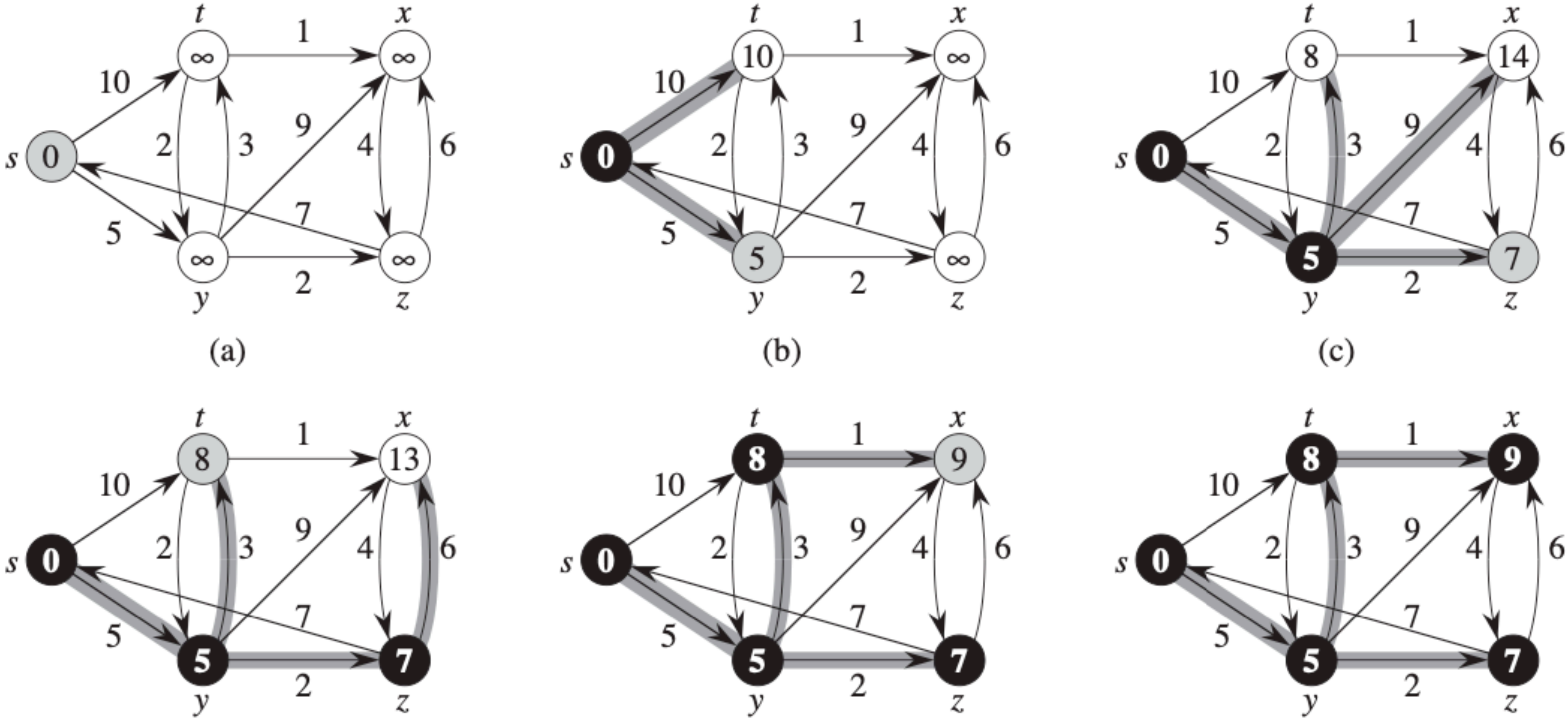


```
INITIALIZE-SINGLE-SOURCE($G, s$)
for each vertex $v \in G.V$
    $v.d = \infty$
    $v.\pi = $ NIL
$s.d = 0$

RELAX($u, v, w$)
if $v.d > u.d + w(u, v)$
    $v.d = u.d + w(u, v)$
    $v.\pi = u$
```



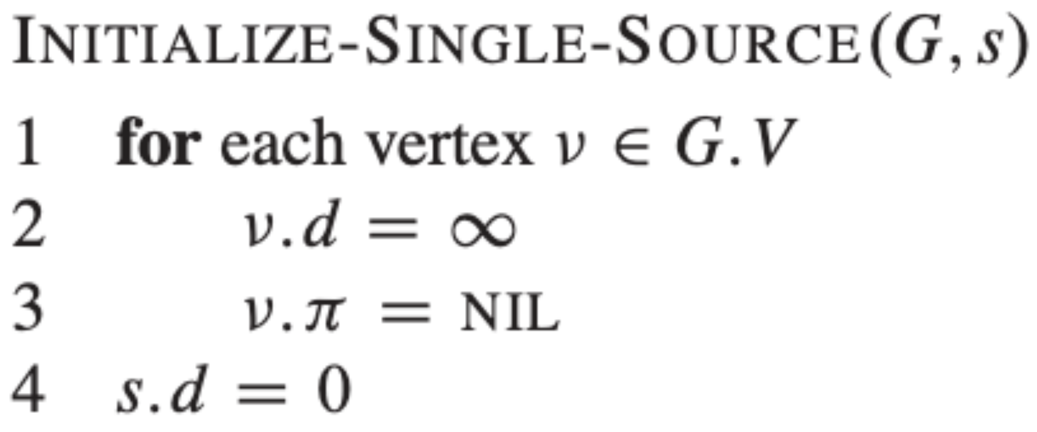
```
DIJKSTRA($G, w, s$)
INITIALIZE-SINGLE-SOURCE($G, s$)
$S = \emptyset$
$Q = G.V$
while $Q \neq \emptyset$
    $u = $ EXTRACT-MIN($Q$)
    $S = S \cup \{u\}$
    for each vertex $v \in G.Adj[u]$
        RELAX($u, v, w$)
```

Dijkstra: Ejemplo

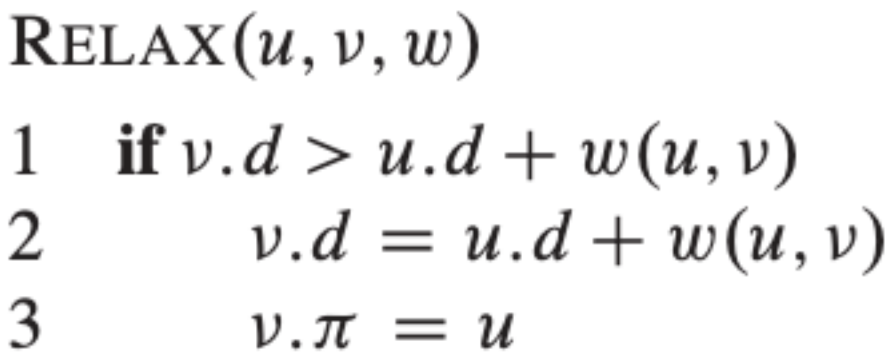


(a) (b) (c)

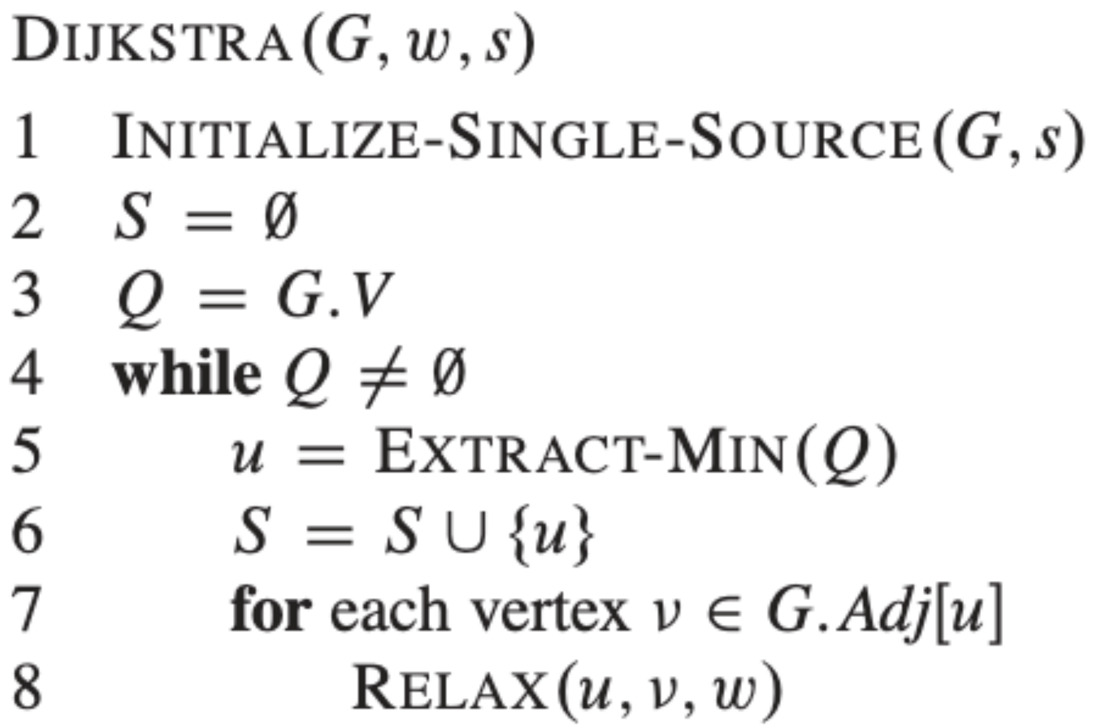
Dijkstra: Tiempo de ejecución



```
INITIALIZE-SINGLE-SOURCE($G, s$)
for each vertex $v \in G.V$
    $v.d = \infty$
    $v.\pi = \text{NIL}$
$s.d = 0$
```



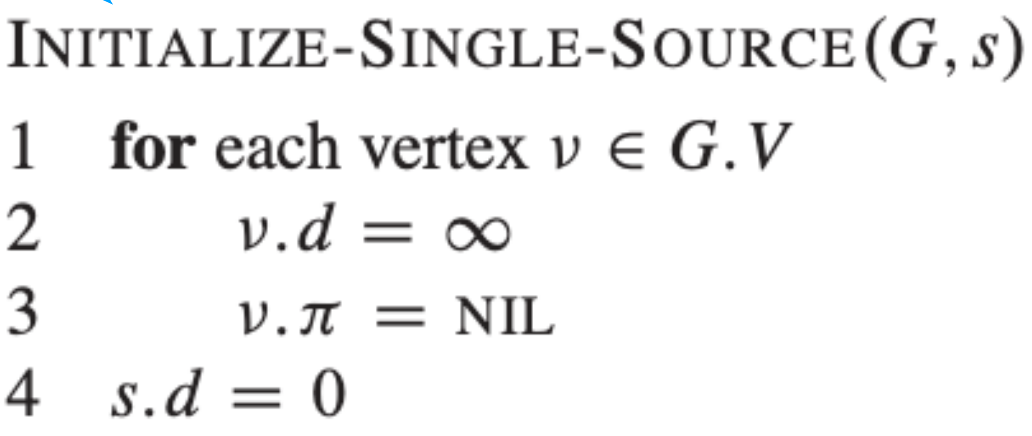
```
RELAX($u, v, w$)
if $v.d > u.d + w(u, v)$
    $v.d = u.d + w(u, v)$
    $v.\pi = u$
```



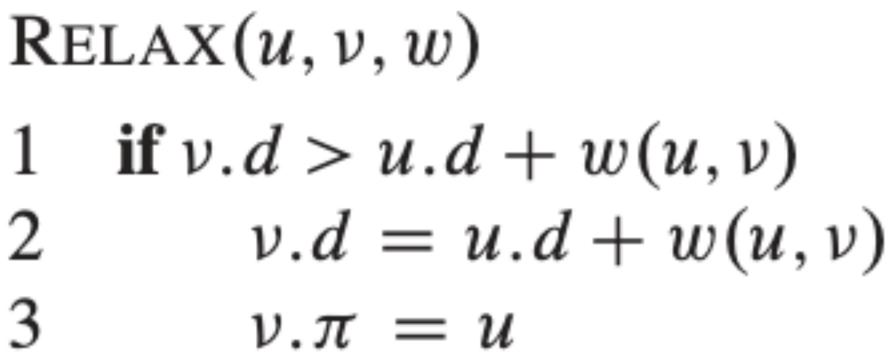
```
DIJKSTRA($G, w, s$)
INITIALIZE-SINGLE-SOURCE($G, s$)
$S = \emptyset$
$Q = G.V$
while $Q \neq \emptyset$
    $u = \text{EXTRACT-MIN}(Q)$
    $S = S \cup \{u\}$
    for each vertex $v \in G.Adj[u]$
        RELAX($u, v, w$)
```

Dijkstra: Tiempo de ejecución

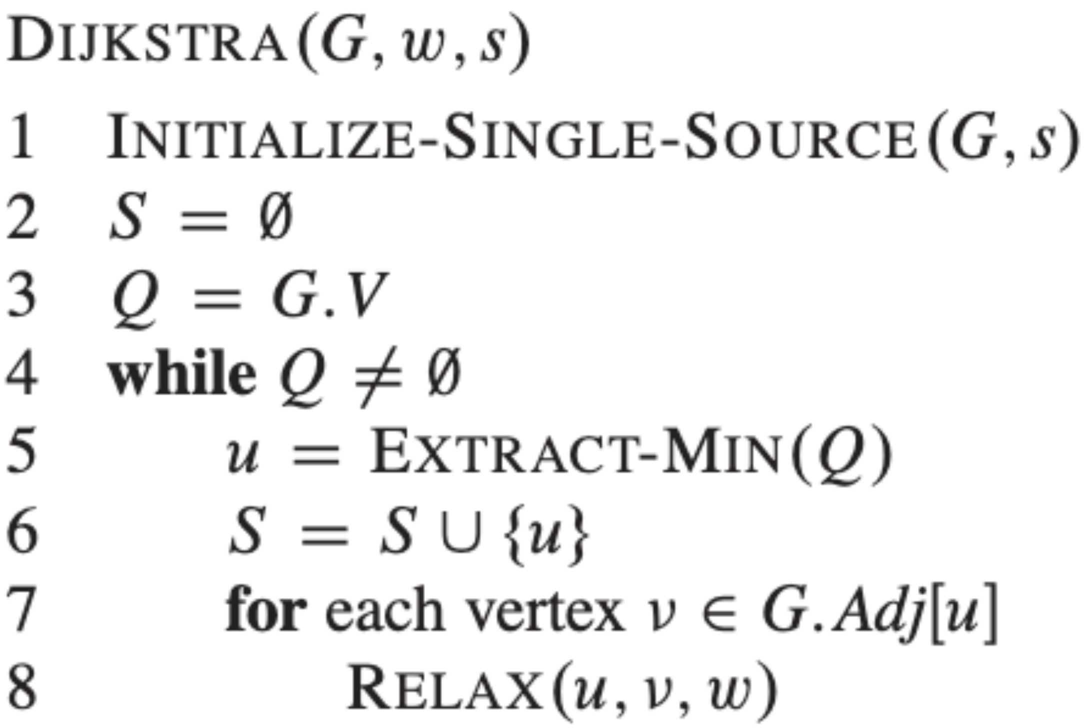
O(V)



```
INITIALIZE-SINGLE-SOURCE($G, s$)
for each vertex $v \in G.V$
    $v.d = \infty$
    $v.\pi = $ NIL
$s.d = 0$
```



```
RELAX($u, v, w$)
if $v.d > u.d + w(u, v)$
    $v.d = u.d + w(u, v)$
    $v.\pi = u$
```



```
DIJKSTRA($G, w, s$)
INITIALIZE-SINGLE-SOURCE($G, s$)
$S = \emptyset$
$Q = G.V$
while $Q \neq \emptyset$
    $u = $ EXTRACT-MIN($Q$)
    $S = S \cup \{u\}$
    for each vertex $v \in G.Adj[u]$
        RELAX($u, v, w$)
```

Dijkstra: Tiempo de ejecución

O(V)

```
INITIALIZE-SINGLE-SOURCE$(G, s)$
for each vertex $v \in G.V$
    $v.d = \infty$
    $v.\pi = \text{NIL}$
$s.d = 0$
```

```
RELAX$(u, v, w)$
if $v.d > u.d + w(u, v)$
    $v.d = u.d + w(u, v)$
    $v.\pi = u$
```

O(V)

```
RA$(G, w, s)$
INITIALIZE-SINGLE-SOURCE$(G, s)$
$S = \emptyset$
$Q = G.V$
while $Q \neq \emptyset$
    $u = $ EXTRACT-MIN$(Q)$
    $S = S \cup \{u\}$
    for each vertex $v \in G.Adj[u]$
        RELAX$(u, v, w)$
```

Dijkstra: Tiempo de ejecución

O(V)

```
INITIALIZE-SINGLE-SOURCE$(G, s)$
for each vertex $v \in G.V$
    $v.d = \infty$
    $v.\pi = \text{NIL}$
$s.d = 0$
```

```
RELAX$(u, v, w)$
if $v.d > u.d + w(u, v)$
    $v.d = u.d + w(u, v)$
    $v.\pi = u$
```

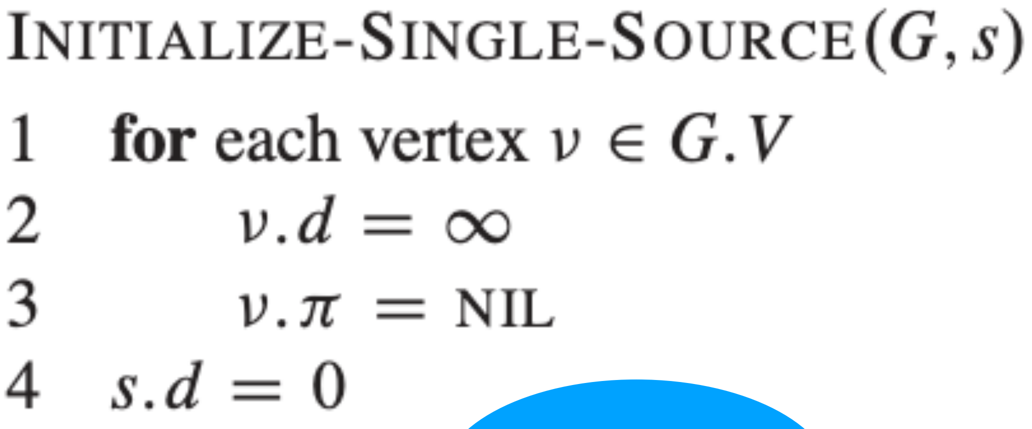
O(V)

O(V)

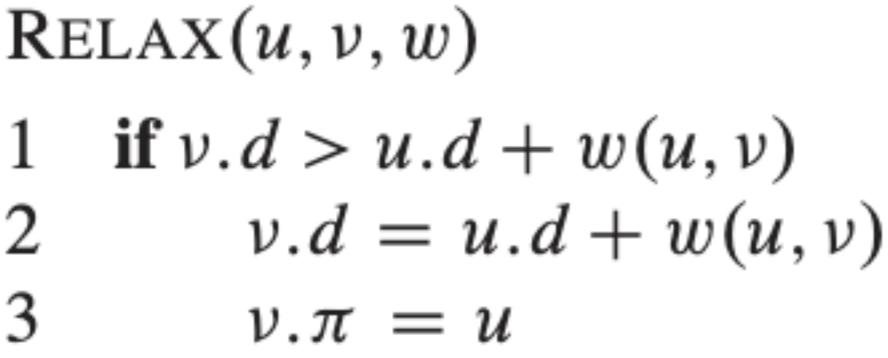
```
RA$(G, w, s)$
NITIALIZE-SINGLE-SOURCE$(G, s)$
$S = \emptyset$
$Q = G.V$
while $Q \neq \emptyset$
    $u = $ EXTRACT-MIN$(Q)$
    $S = S \cup \{u\}$
    for each vertex $v \in G.Adj[u]$
        RELAX$(u, v, w)$
```

Dijkstra: Tiempo de ejecución

O(V)



```
INITIALIZE-SINGLE-SOURCE$(G, s)$
for each vertex $v \in G.V$
    $v.d = \infty$
    $v.\pi = \text{NIL}$
$s.d = 0$
```

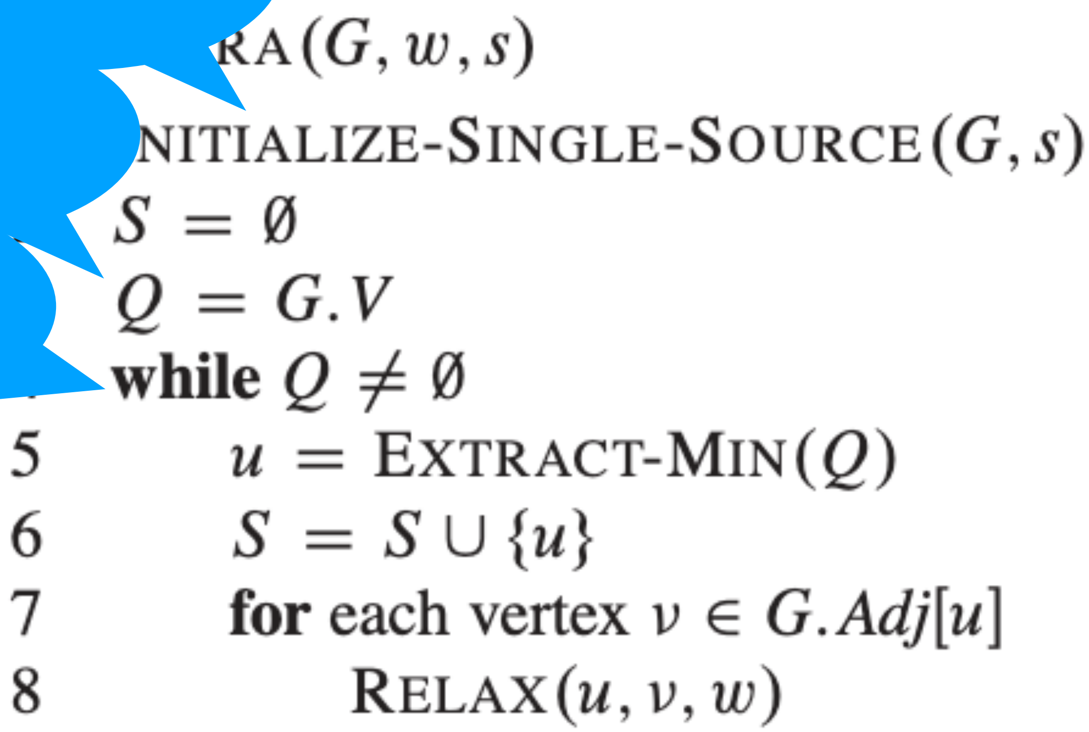


```
RELAX$(u, v, w)$
if $v.d > u.d + w(u, v)$
    $v.d = u.d + w(u, v)$
    $v.\pi = u$
```

O(V)

O(V)

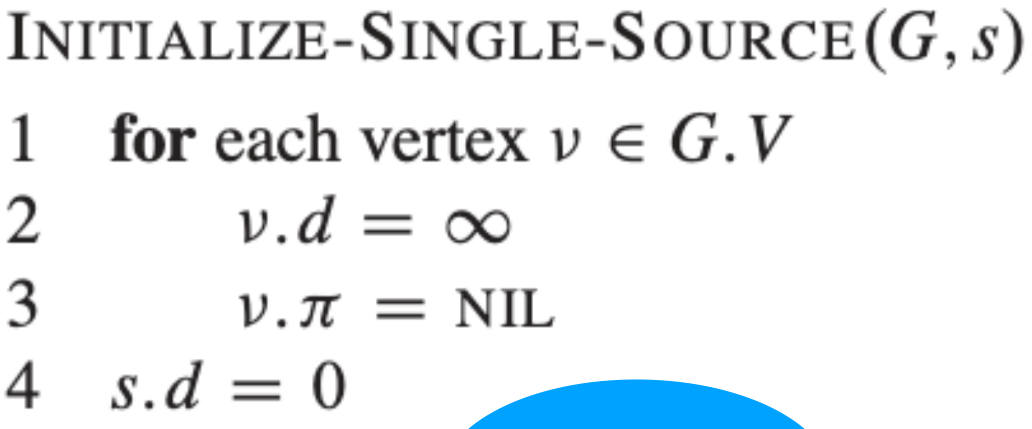
Se ejecuta V veces



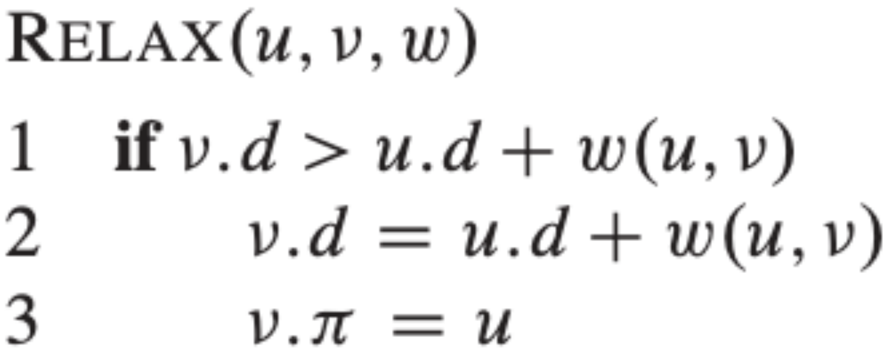
```
RA$(G, w, s)$
NITIALIZE-SINGLE-SOURCE$(G, s)$
$S = \emptyset$
$Q = G.V$
while $Q \neq \emptyset$
    $u = $ EXTRACT-MIN$(Q)$
    $S = S \cup \{u\}$
    for each vertex $v \in G.Adj[u]$
        RELAX$(u, v, w)$
```

Dijkstra: Tiempo de ejecución

O(V)



```
INITIALIZE-SINGLE-SOURCE$(G, s)$
for each vertex $v \in G.V$
    $v.d = \infty$
    $v.\pi = \text{NIL}$
$s.d = 0$
```



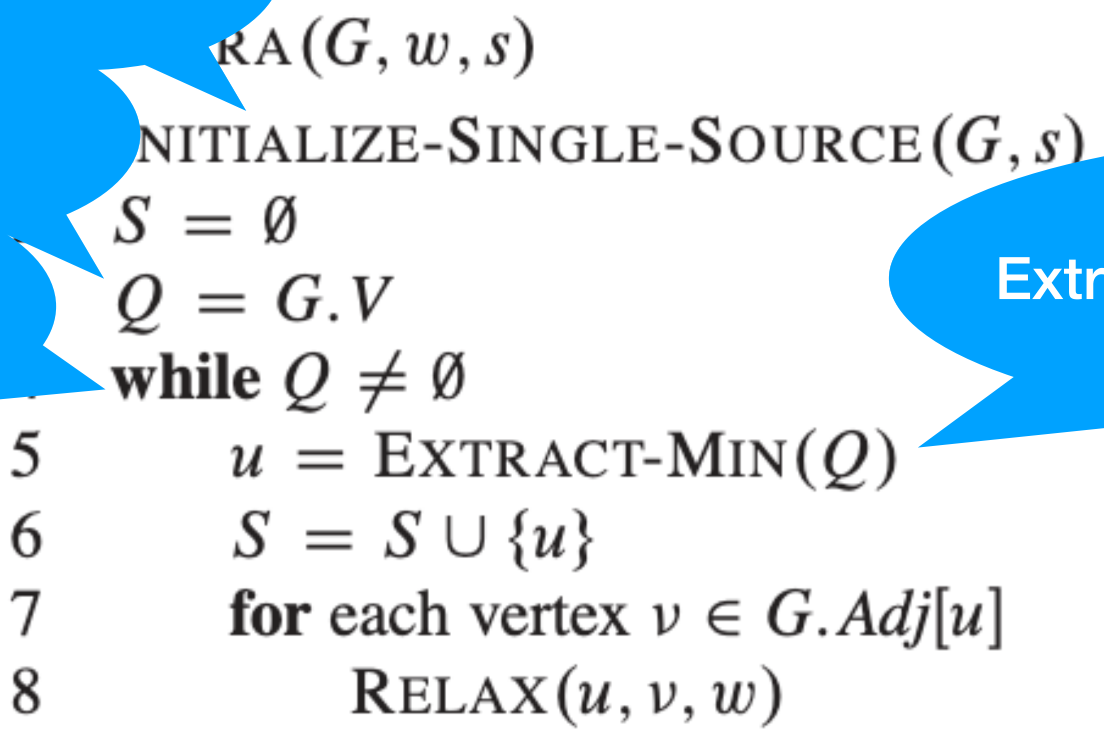
```
RELAX$(u, v, w)$
if $v.d > u.d + w(u, v)$
    $v.d = u.d + w(u, v)$
    $v.\pi = u$
```

O(V)

O(V)

Se ejecuta V veces

O(V log V), ya que ExtractMin es O(log V) con heaps



```
RA$(G, w, s)$
NITIALIZE-SINGLE-SOURCE$(G, s)$
$S = \emptyset$
$Q = G.V$
while $Q \neq \emptyset$
    $u = \text{EXTRACT-MIN}(Q)$
    $S = S \cup \{u\}$
    for each vertex $v \in G.Adj[u]$
        RELAX$(u, v, w)$
```

Dijkstra: Tiempo de ejecución

O(V)

```
INITIALIZE-SINGLE-SOURCE$(G, s)$
for each vertex $v \in G.V$
    $v.d = \infty$
    $v.\pi = \text{NIL}$
$s.d = 0$
```

```
RELAX$(u, v, w)$
if $v.d > u.d + w(u, v)$
    $v.d = u.d + w(u, v)$
    $v.\pi = u$
```

O(V)

O(V)

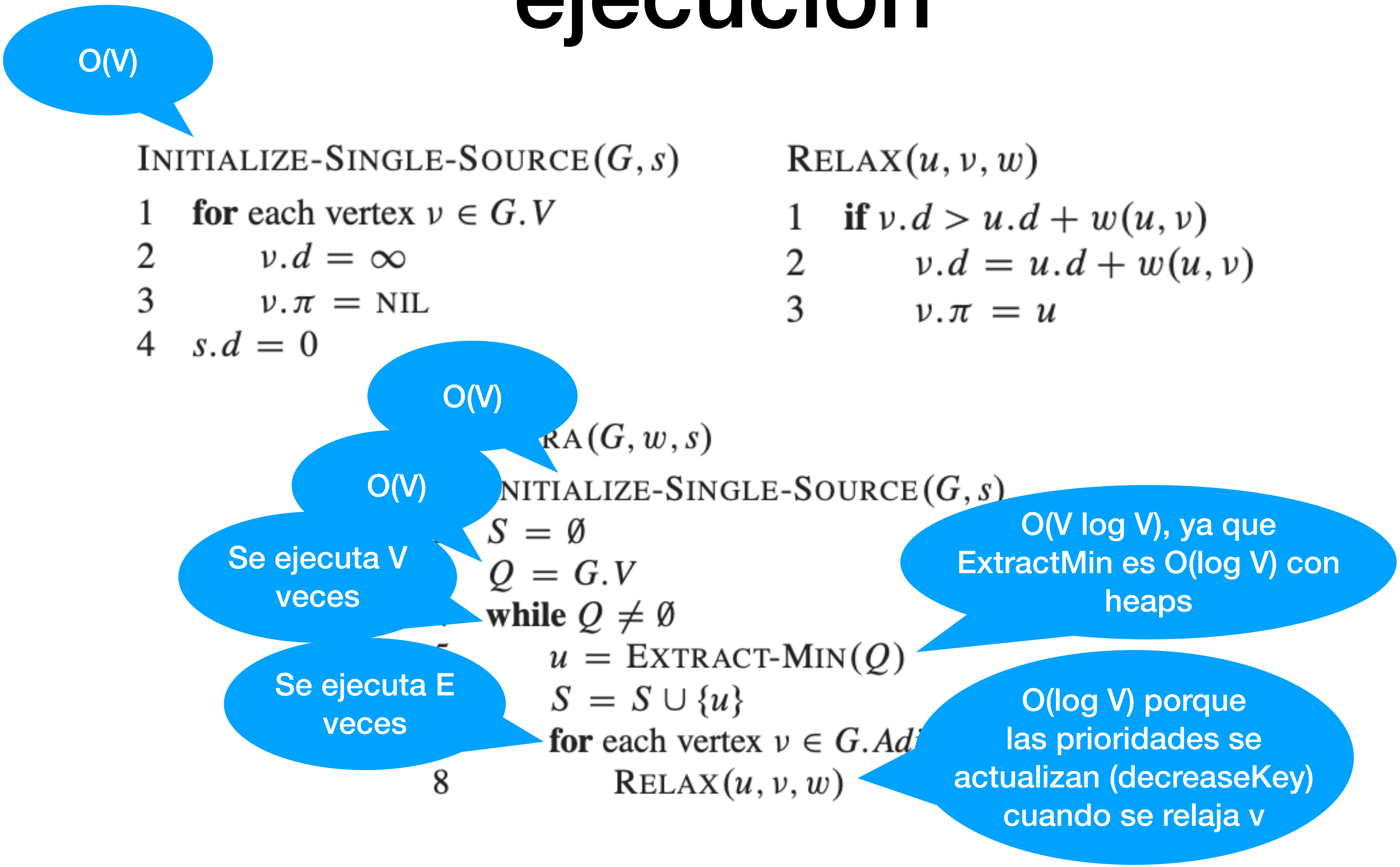
Se ejecuta V veces

Se ejecuta E veces

O(V log V), ya que ExtractMin es O(log V) con heaps

```
RA$(G, w, s)$
NITIALIZE-SINGLE-SOURCE$(G, s)$
$S = \emptyset$
$Q = G.V$
while $Q \neq \emptyset$
    $u = $ EXTRACT-MIN$(Q)$
    $S = S \cup \{u\}$
    for each vertex $v \in G.Adj[u]$
        RELAX$(u, v, w)$
```

Dijkstra: Tiempo de ejecución



O(V)

```
INITIALIZE-SINGLE-SOURCE$(G, s)$
for each vertex $v \in G.V$
    $v.d = \infty$
    $v.\pi = \text{NIL}$
$s.d = 0$
```

```
RELAX$(u, v, w)$
if $v.d > u.d + w(u, v)$
    $v.d = u.d + w(u, v)$
    $v.\pi = u$
```

O(V)

O(V)

Se ejecuta V veces

Se ejecuta E veces

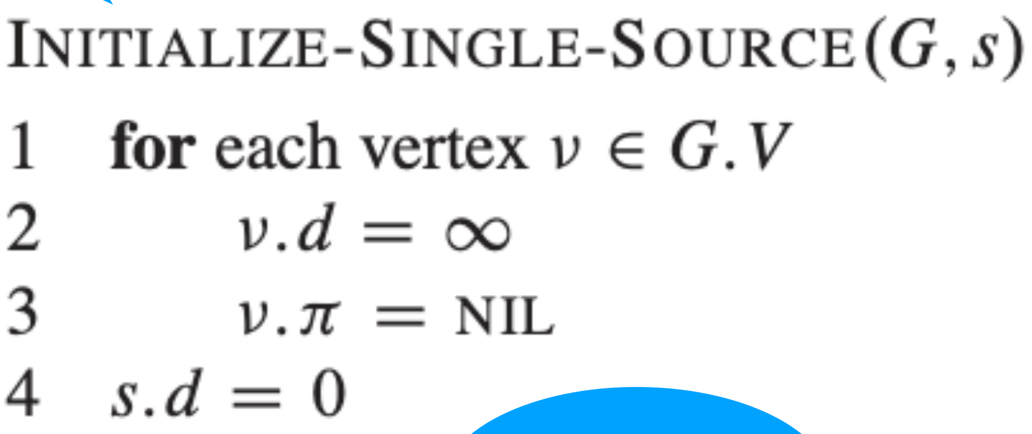
```
RA$(G, w, s)$
NITIALIZE-SINGLE-SOURCE$(G, s)$
$S = \emptyset$
$Q = G.V$
while $Q \neq \emptyset$
    $u = $ EXTRACT-MIN$(Q)$
    $S = S \cup \{u\}$
    for each vertex $v \in G.Ad$
        RELAX$(u, v, w)$
```

O(V log V), ya que ExtractMin es O(log V) con heaps

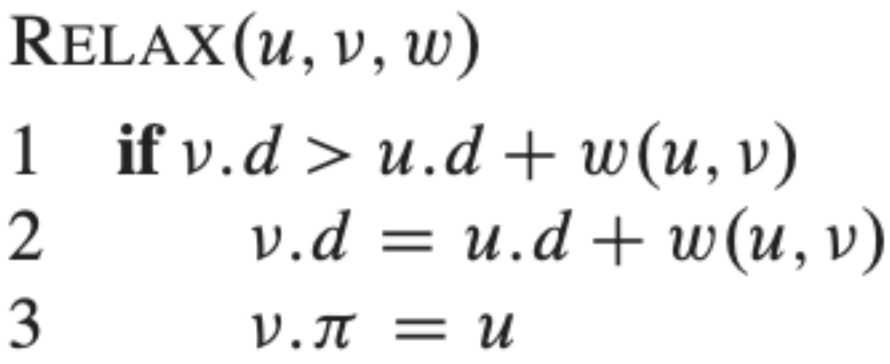
O(log V) porque las prioridades se actualizan (decreaseKey) cuando se relaja v

Dijkstra: Tiempo de ejecución

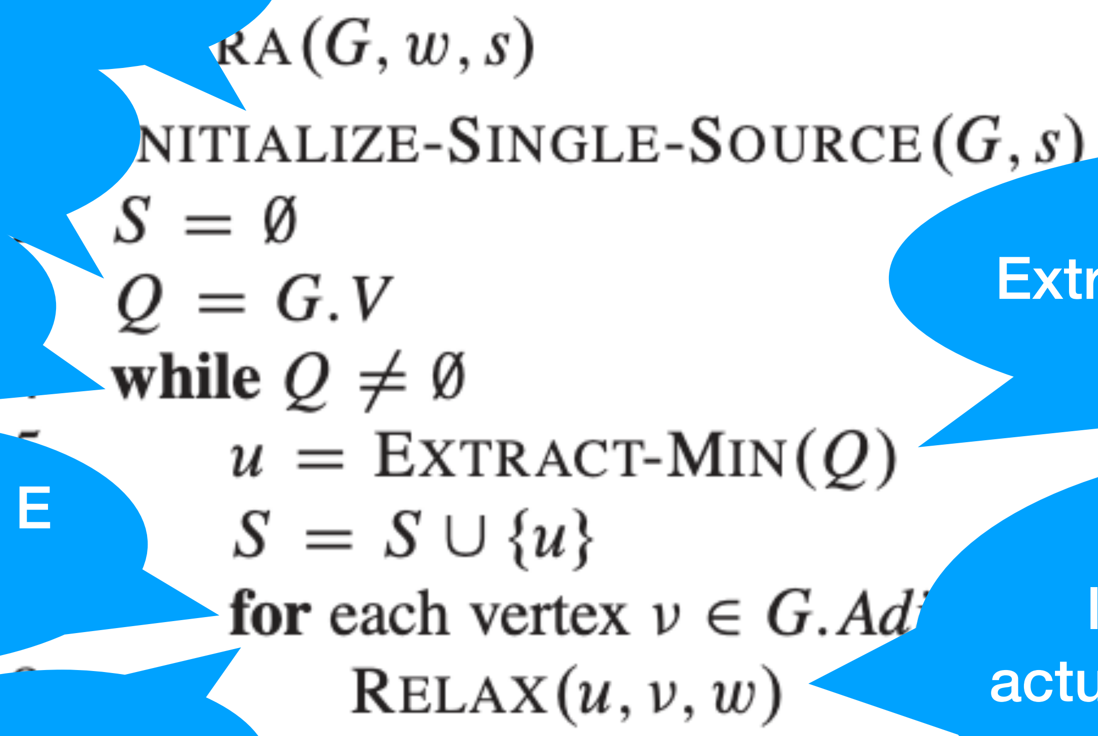
O(V)



```
INITIALIZE-SINGLE-SOURCE$(G, s)$
for each vertex $v \in G.V$
    $v.d = \infty$
    $v.\pi = \text{NIL}$
$s.d = 0$
```



```
RELAX$(u, v, w)$
if $v.d > u.d + w(u, v)$
    $v.d = u.d + w(u, v)$
    $v.\pi = u$
```



```
RA$(G, w, s)$
NITIALIZE-SINGLE-SOURCE$(G, s)$
$S = \emptyset$
$Q = G.V$
while $Q \neq \emptyset$
    $u = $ EXTRACT-MIN$(Q)$
    $S = S \cup \{u\}$
    for each vertex $v \in G.Adj$
        RELAX$(u, v, w)$
```

O(V)

O(V)

Se ejecuta V veces

O(V log V), ya que ExtractMin es O(log V) con heaps

Se ejecuta E veces

O(log V) porque las prioridades se actualizan (decreaseKey) cuando se relaja v

O(E log V)

Dijkstra: Tiempo de ejecución

O(V)

```
INITIALIZE-SINGLE-SOURCE$(G, s)$
for each vertex $v \in G.V$
    $v.d = \infty$
    $v.\pi = $ NIL
$s.d = 0$
```

```
RELAX$(u, v, w)$
if $v.d > u.d + w(u, v)$
    $v.d = u.d + w(u, v)$
```

Dijkstra es O(V log V + E log V)

O(V)

```
RA$(G, w, s)$
    NITIALIZE-SINGLE-SOURCE$(G, s)$
    $S = \emptyset$
    $Q = G.V$
    while $Q \neq \emptyset$
        $u = $ EXTRACT-MIN$(Q)$
        $S = S \cup \{u\}$
        for each vertex $v \in G.Ad$
            RELAX$(u, v, w)$
```

O(V)

Se ejecuta V veces

O(V log V), ya que ExtractMin es O(log V) con heaps

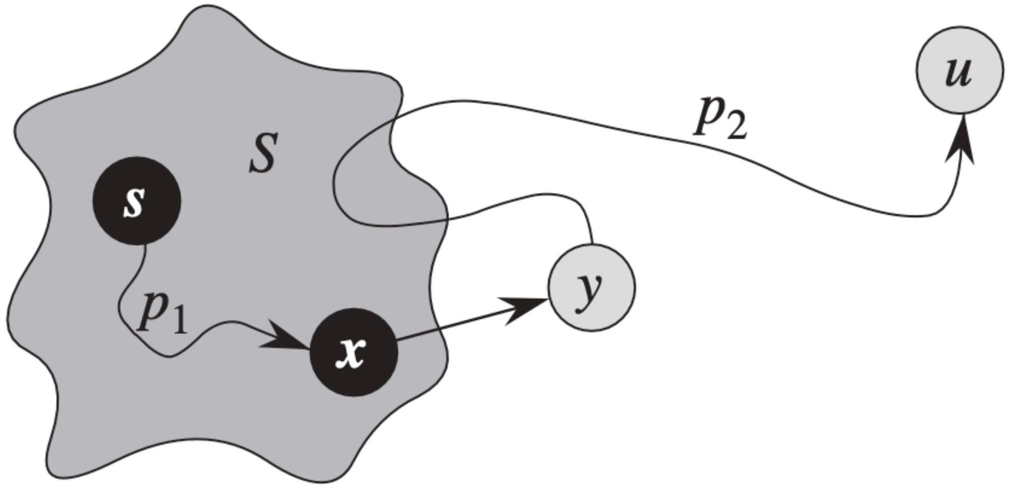
Se ejecuta E veces

O(log V) porque las prioridades se actualizan (decreaseKey) cuando se relaja v

O(E log V)

Dijkstra: Prueba de corrección

- Invariante del ciclo de Dijkstra: para todos los vértices v en el conjunto S, $v.d$ es igual a la distancia del camino más corto de s a `v`, denotada como $\delta(s, v)$
- Debemos probar que cada vez que Dijkstra agrega un vértice `u` a S se cumple que $u.d = \delta(s, u)$
- Razonemos por el absurdo: Sea `u` el primer vértice que el algoritmo agrega a S tal que `u` no está en el camino más corto de `s` a `u`, es decir, $u.d \neq \delta(s, u)$
- Como $u.d \neq \delta(s, u)$, debe existir un camino más corto de `s` a `u`, que contenga un vértice `y` fuera de S que aparece antes que `u` en el camino
- Es decir, podemos descomponer el camino más corto de `s` a `u` en `s--p1->x->y--p2->u`, donde `s`, `x`, y todos los vértices en `p1` están en S, e `y` es el primer vértice fuera de S (ver Figura)
    - Si no existiera tal `y`, `u` es el primer vértice fuera de `S` en el camino más corto de `s` a `u`
    - Como $x.d = \delta(s, x)$ (`x` está en S), y `u` es el nodo que le sigue en el camino más corto de `s` a `u`, al relajar el arco `x->u` tenemos que $u.d = \delta(s, u)$
    - Esto contradice la hipótesis $u.d \neq \delta(s, u)$, por lo que `y` debe existir



Dijkstra: Prueba de corrección

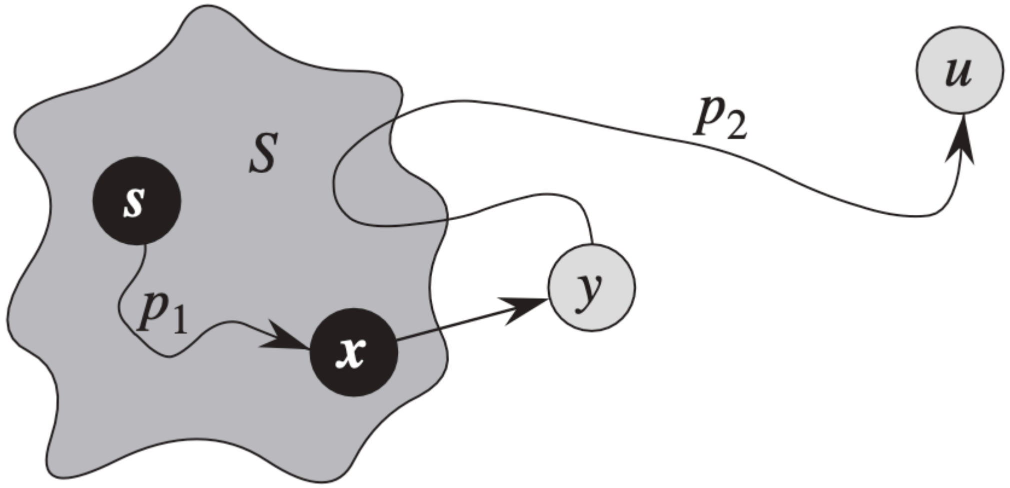
- Notar que, `s--p1->x->y` es un subcamino del camino más corto de `s` a `u`, por lo que `s--p1->x->y` es un camino más corto de `s` a `y`
  - Subcaminos de un camino más corto son caminos más cortos
- Como $x.d = \delta(s, x)$ (`x` está en S) y `s--p1->x->y` es un camino más corto de `s` a `y`, al relajar el arco `x->y` tenemos que $y.d = \delta(s, y)$
- Entonces:

$$\begin{aligned} y.d &= \delta(s, y) \\ &\leq \delta(s, u) && \text{\{no hay arcos negativos en p2\}} \\ &\leq u.d && \text{\{el algoritmo no puede asignar a } u.d \text{ un valor menor a } \delta(s, u)\text{\}} \end{aligned}$$

- Pero `u` se agregó a S, por lo que debe valer que $u.d \leq y.d$ (sino `y` debería haberse agregado a S)
- De $u.d \geq y.d$ y $u.d \leq y.d$ se deduce $u.d = y.d$, y por lo tanto:

$$y.d = \delta(s, y) = \delta(s, u) = u.d$$

- Esto contradice la hipótesis $u.d \neq \delta(s, u)$, y se concluye que $u.d = \delta(s, u)$ siempre que se agrega un vértice `u` a S
- Es decir, siempre que Dijkstra agrega un vértice `u` a S, `u` está en un camino más corto de `s` a `u`



Dijkstra: Implementación

```
/**
 * @pre 0=<s<V && G has no edges with negative weight
 * @post Computes the shortest paths from the source vertex s to every other
 *   vertex in the edge-weighted digraph G.
 */
public Dijkstra(EdgeWeightedIntDigraph G, int s) {
    if (s < 0 || s >= G.V())
        throw new IllegalArgumentException("vertex " + s +
                " is not between 0 and " + (G.V()-1));
    // Initialization
    distTo = new double[G.V()];
    edgeTo = new DirectedEdge[G.V()];
    pq = new IndexMinPQ<Double>(G.V());
    for (int v = 0; v < G.V(); v++) {
        if (v != s) {
            distTo[v] = Double.POSITIVE_INFINITY;
            pq.insert(v, distTo[v]);
        }
    }
    distTo[s] = 0.0;
    pq.insert(s, distTo[s]);
    // Compute shortest paths
    while (!pq.isEmpty()) {
        int v = pq.delMin();
        for (DirectedEdge e : G.adj(v))
            relax(e);
    }
}
```

Dijkstra: Implementación

Cola de prioridad (heap) que guarda prioridades asociadas a índices

```
/**
 * @pre 0=<s<V && G has no edges with negative weight
 * @post Computes the shortest paths from the source vertex s to every other
 *   vertex in the edge-weighted digraph G.
 */
public Dijkstra(EdgeWei                    t s) {
    if (s < 0 || s >= 
        throw new Illeg                     ertex " + s +
                " is not                   (G.V()-1));
    // Initialization
    distTo = new double[G.V  ];
    edgeTo = new DirectedEdge[G.V()];
    pq = new IndexMinPQ<Double>(G.V());
    for (int v = 0; v < G.V(); v++) {
        if (v != s) {
            distTo[v] = Double.POSITIVE_INFINITY;
            pq.insert(v, distTo[v]);
        }
    }
    distTo[s] = 0.0;
    pq.insert(s, distTo[s]);
    // Compute shortest paths
    while (!pq.isEmpty()) {
        int v = pq.delMin();
        for (DirectedEdge e : G.adj(v))
            relax(e);
    }
}
```

Dijkstra: Implementación

```
/**
 * @pre 0=<s<V && G has no edges with negative weight
 * @post Computes the shortest paths from the source vertex s to every other
 *   vertex in the edge-weighted digraph G.
 */
public Dijkstra(EdgeWei                    t s) {
    if (s < 0 || s >= (
        throw new Illeg                    ertex " + s +
                " is not                  (G.V()-1));
    // Initialization
    distTo = new double[G.V()];
    edgeTo = new DirectedEdge[G.V()];
    pq = new IndexMinPQ<Double>(G.V());
    for (int v = 0; v < G.V(); v++) {
        if (v != s) {
            distTo[v] = Double.POSITIVE_I
            pq.insert(v, distTo[v]);
        }
    }
    distTo[s] = 0.0;
    pq.insert(s, distTo[s]);
    // Compute shortest paths
    while (!pq.isEmpty()) {
        int v = pq.delMin();
        for (DirectedEdge e : G.adj(v))
            relax(e);
    }
}
```

Cola de prioridad (heap) que guarda prioridades asociadas a índices

```
/**
 * @post Relax edge e and update pq
 *   and edgeTo if changed.
 */
private void relax(DirectedEdge e) {
   int v = e.from, w = e.to;
   if (distTo[w] > distTo[v] + e.weight) {
       distTo[w] = distTo[v] + e.weight;
       edgeTo[w] = e;
       pq.decreaseKey(w, distTo[w]);
   }
}
```

Dijkstra: Implementación

```
/**
 * @pre 0=<s<V && G has no edges with negative weight
 * @post Computes the shortest paths from the source vertex s to every other
 *   vertex in the edge-weighted digraph G.
 */
public Dijkstra(EdgeWei                t s) {
    if (s < 0 || s >= G
        throw new Illeg                ertex " + s +
                " is not               (G.V()-1));
    // Initialization
    distTo = new double[G.V  ];
    edgeTo = new DirectedEdge[G.V()];
    pq = new IndexMinPQ<Double>(G.V());
    for (int v = 0; v < G.V(); v++) {
        if (v != s) {
            distTo[v] = Double.POSITIVE_I
            pq.insert(v, distTo[v]);
        }
    }
    distTo[s] = 0.0;
    pq.insert(s, distTo[s]);
    // Compute shortest paths
    while (!pq.isEmpty()) {
        int v = pq.delMin();
        for (DirectedEdge e : G.adj(v))
            relax(e);
    }
}
```

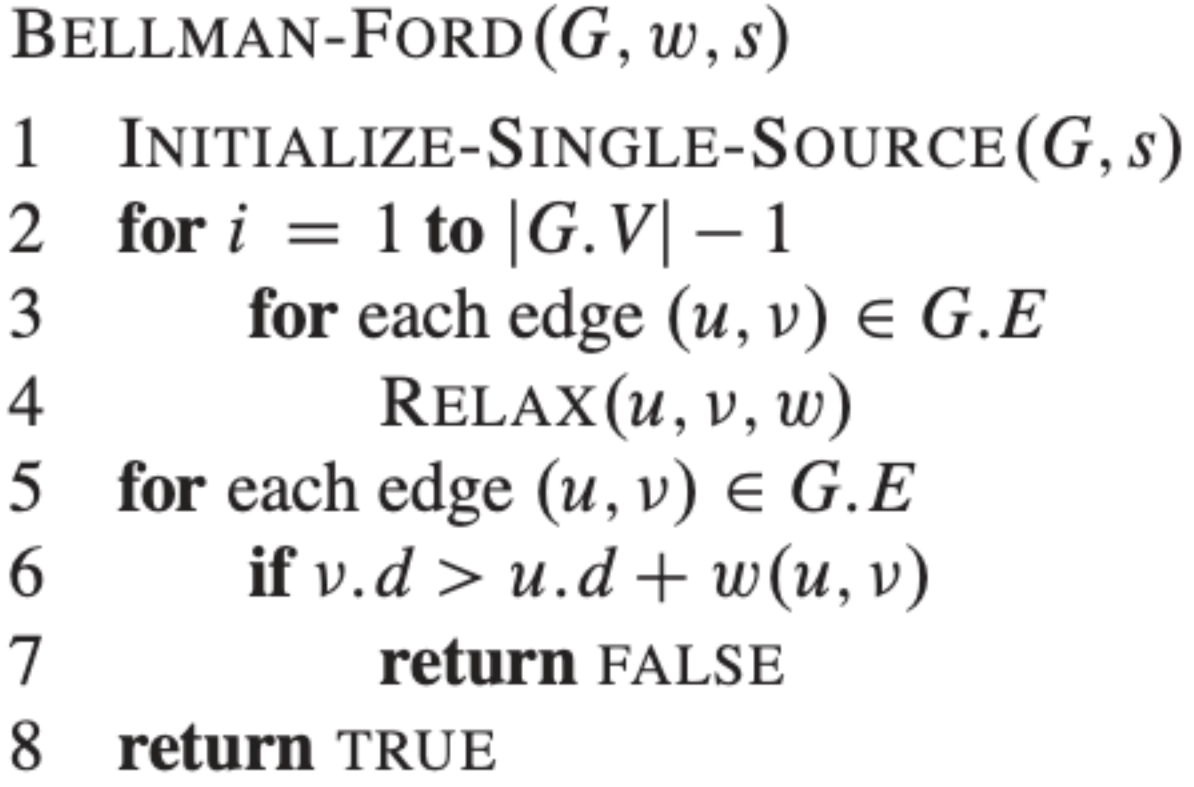
Cola de prioridad (heap) que guarda prioridades asociadas a índices

```
/**
 * @post Relax edge e and update pq
 *   and edgeT
 */
private vo                               {
    int v =
    if (distT                      weight) {
        distTo[w]            ] + e.weight;
        edgeTo[w] = e;
        pq.decreaseKey(w, distTo[w]);
    }
}
```

Cambiar la prioridad de un vértice es O(log V) con heaps

Bellman-Ford

- El algoritmo de Bellman-Ford soporta aristas con pesos negativos
- Y detecta ciclos negativos en el grafo (retorna falso)
- Idea: Si relajamos V-1 veces cada arista obtenemos todos los caminos más cortos (tienen V-1 aristas como máximo)

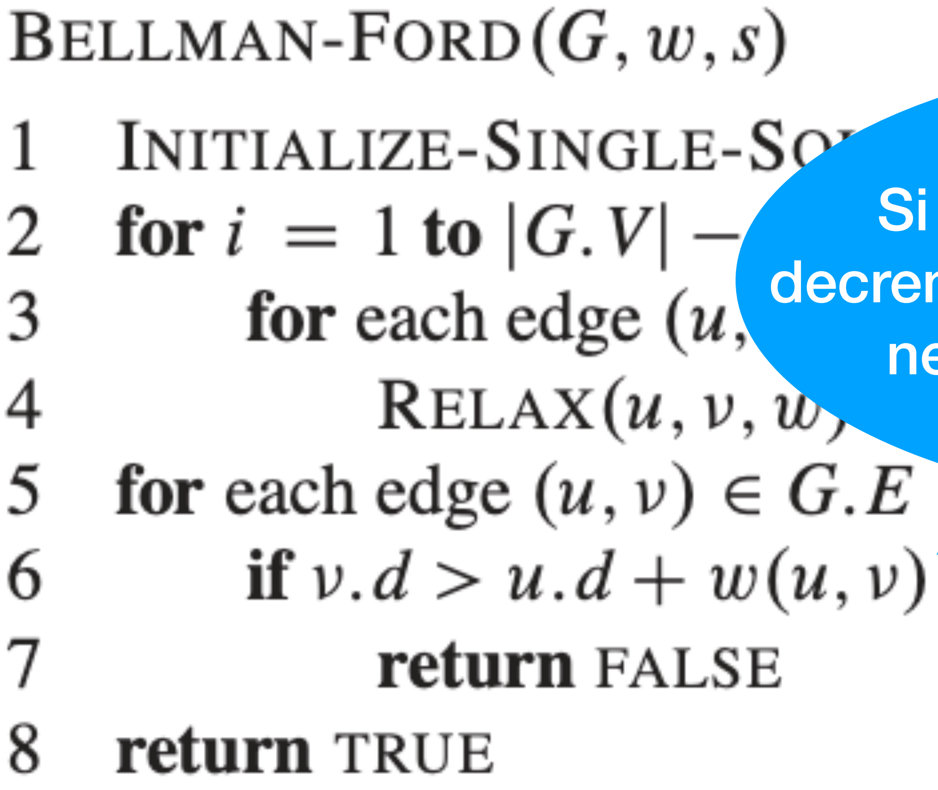


```
BELLMAN-FORD($G, w, s$)
INITIALIZE-SINGLE-SOURCE($G, s$)
for $i = 1$ to $|G.V| - 1$
    for each edge $(u, v) \in G.E$
        RELAX($u, v, w$)
for each edge $(u, v) \in G.E$
    if $v.d > u.d + w(u, v)$
        return FALSE
return TRUE
```

- Sin embargo, es mucho más ineficiente que Dijkstra (O(V log V+E log V)), ya que tarda O(V+VE+E) = O(VE) en el peor caso

Bellman-Ford

- El algoritmo de Bellman-Ford soporta aristas con pesos negativos
- Y detecta ciclos negativos en el grafo (retorna falso)
- Idea: Si relajamos V-1 veces cada arista obtenemos todos los caminos más cortos (tienen V-1 aristas como máximo)



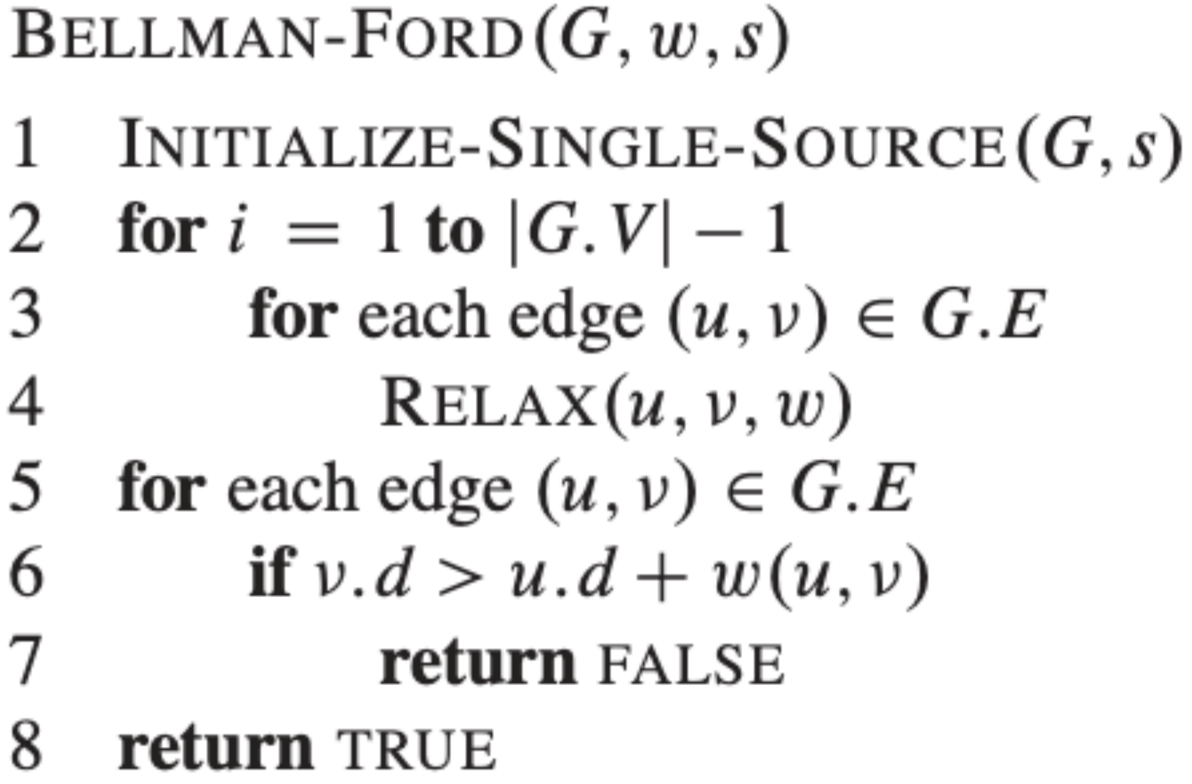
```
BELLMAN-FORD$(G, w, s)$
INITIALIZE-SINGLE-SO
for $i = 1$ to $|G.V| -$
    for each edge $(u,$
        RELAX$(u, v, w$
for each edge $(u, v) \in G.E$
    if $v.d > u.d + w(u, v)$
        return FALSE
return TRUE
```

Si el costo de un nodo sigue decrementándose, tenemos un ciclo negativo y retornamos false

- Sin embargo, es mucho más ineficiente que Dijkstra (O(V log V+E log V)), ya que tarda O(V+VE+E) = O(VE) en el peor caso

Bellman-Ford

- El algoritmo de Bellman-Ford soporta aristas con pesos negativos
- Y detecta ciclos negativos en el grafo (retorna falso)
- Idea: Si relajamos V-1 veces cada arista obtenemos todos los caminos más cortos (tienen V-1 aristas como máximo)

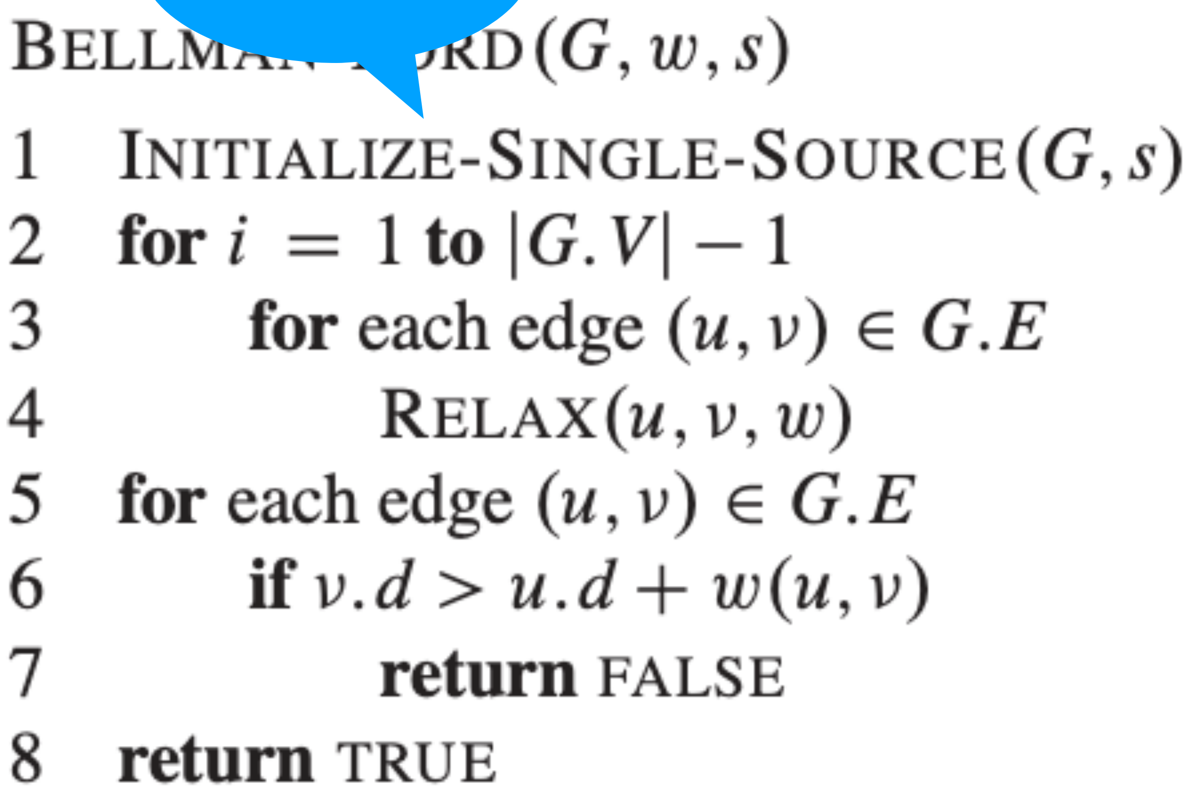


```
BELLMAN-FORD($G, w, s$)
INITIALIZE-SINGLE-SOURCE($G, s$)
for $i = 1$ to $|G.V| - 1$
    for each edge $(u, v) \in G.E$
        RELAX($u, v, w$)
for each edge $(u, v) \in G.E$
    if $v.d > u.d + w(u, v)$
        return FALSE
return TRUE
```

- Sin embargo, es mucho más ineficiente que Dijkstra (O(V log V+E log V)), ya que tarda O(V+VE+E) = O(VE) en el peor caso

Bellman-Ford

- El algoritmo de Bellman-Ford soporta aristas con pesos negativos
- Y detecta ciclos negativos en el grafo (retorna falso)
- Idea: Si relajamos V-1 veces cada arista obtenemos todos los caminos más cortos (tienen V-1 arista [illegible] mo)



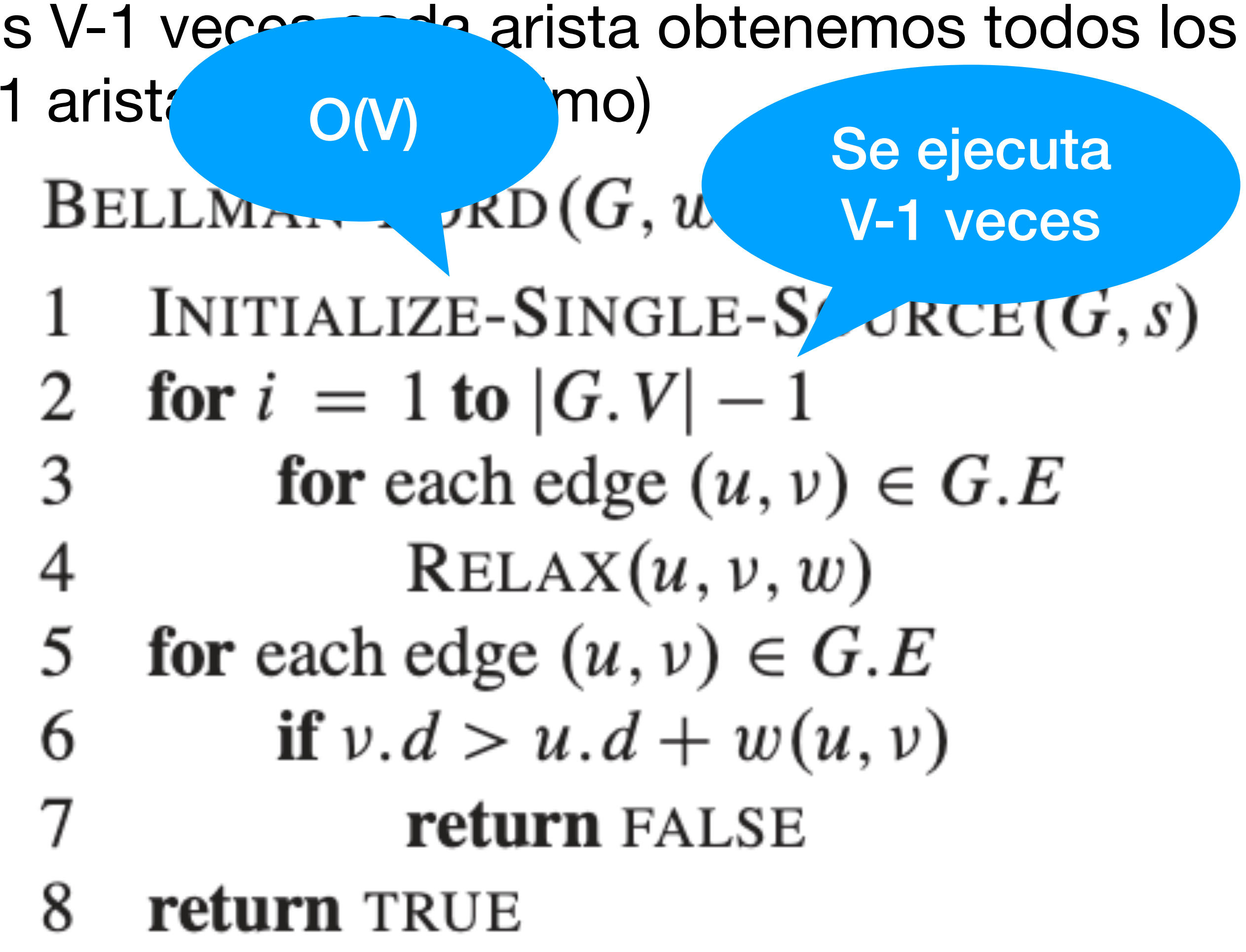
O(V)

```
BELLMAN-FORD$(G, w, s)$
INITIALIZE-SINGLE-SOURCE$(G, s)$
for $i = 1$ to $|G.V| - 1$
    for each edge $(u, v) \in G.E$
        RELAX$(u, v, w)$
for each edge $(u, v) \in G.E$
    if $v.d > u.d + w(u, v)$
        return FALSE
return TRUE
```

- Sin embargo, es mucho más ineficiente que Dijkstra (O(V log V+E log V)), ya que tarda O(V+VE+E) = O(VE) en el peor caso

Bellman-Ford

- El algoritmo de Bellman-Ford soporta aristas con pesos negativos
- Y detecta ciclos negativos en el grafo (retorna falso)
- Idea: Si relajamos V-1 veces cada arista obtenemos todos los caminos más cortos (tienen V-1 arista mo)

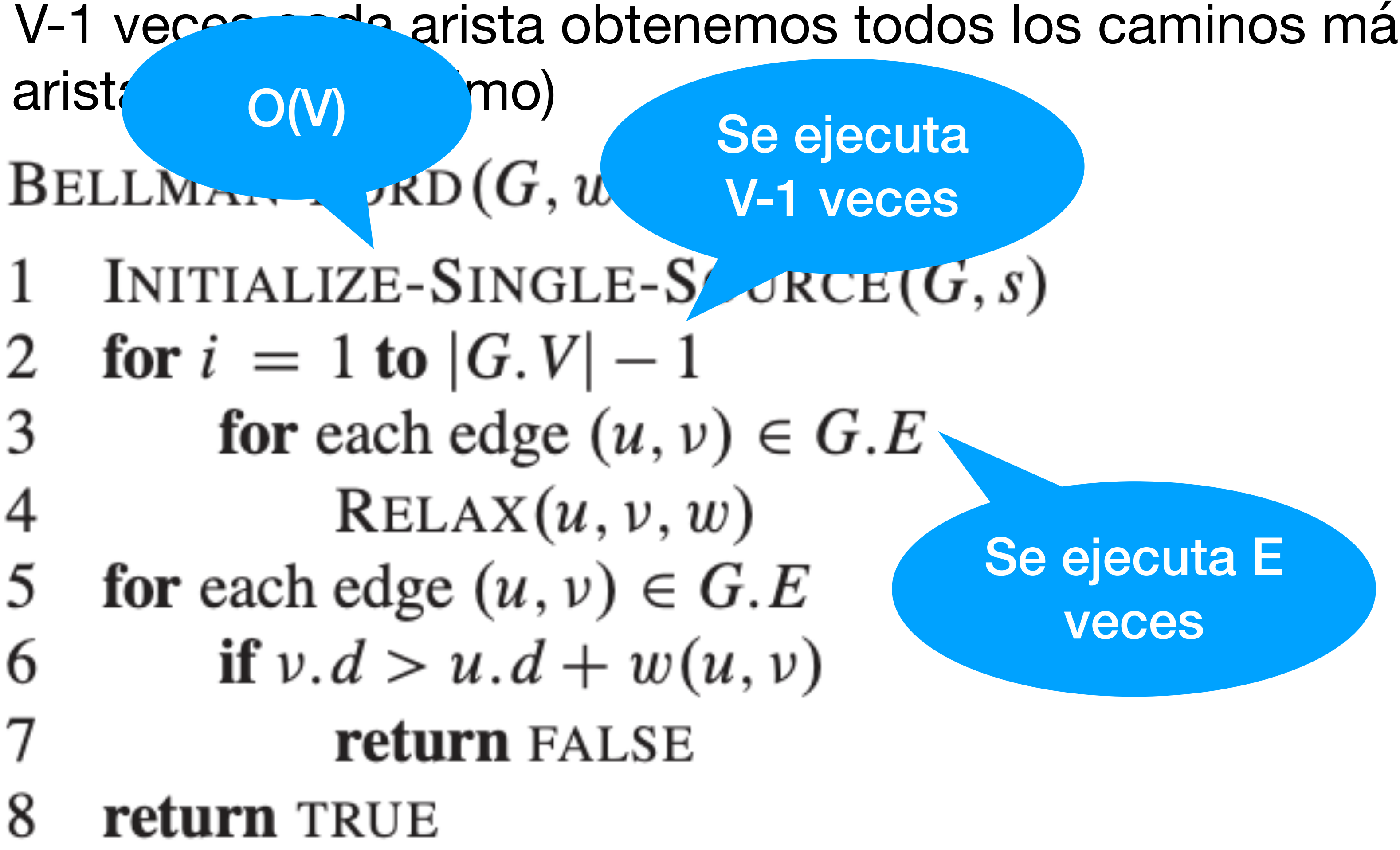


```
BELLM RD(G, w
INITIALIZE-SINGLE-S URCE(G, s)
for $i = 1$ to $|G.V| - 1$
    for each edge $(u, v) \in G.E$
        RELAX$(u, v, w)$
for each edge $(u, v) \in G.E$
    if $v.d > u.d + w(u, v)$
        return FALSE
return TRUE
```

- Sin embargo, es mucho más ineficiente que Dijkstra (O(V log V+E log V)), ya que tarda O(V+VE+E) = O(VE) en el peor caso

Bellman-Ford

- El algoritmo de Bellman-Ford soporta aristas con pesos negativos
- Y detecta ciclos negativos en el grafo (retorna falso)
- Idea: Si relajamos V-1 veces cada arista obtenemos todos los caminos más cortos (tienen V-1 arist mo)

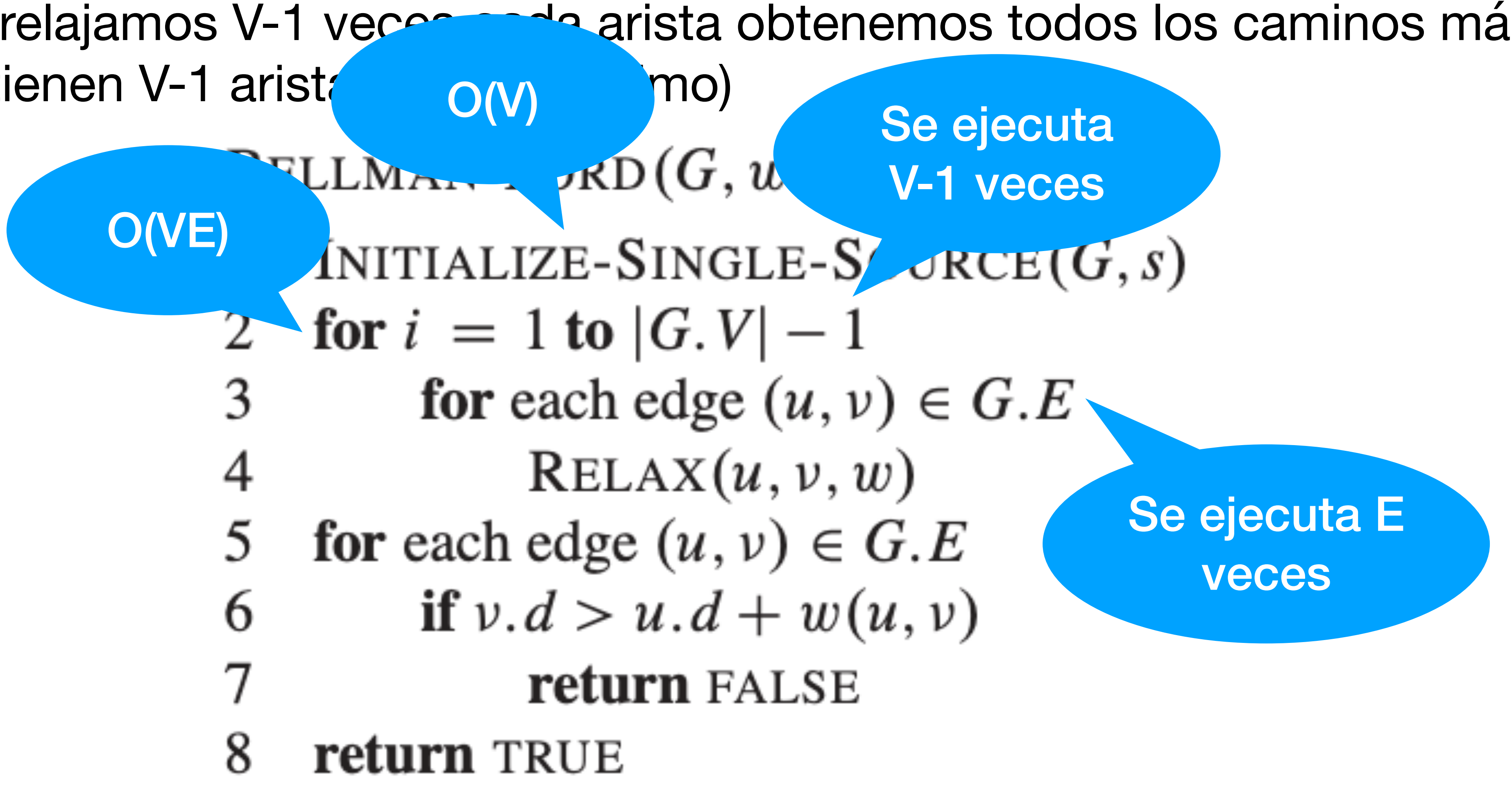


```
BELLMAN-FORD(G, w
INITIALIZE-SINGLE-SOURCE(G, s)
for $i = 1$ to $|G.V| - 1$
    for each edge $(u, v) \in G.E$
        RELAX$(u, v, w)$
for each edge $(u, v) \in G.E$
    if $v.d > u.d + w(u, v)$
        return FALSE
return TRUE
```

- Sin embargo, es mucho más ineficiente que Dijkstra (O(V log V+E log V)), ya que tarda O(V+VE+E) = O(VE) en el peor caso

Bellman-Ford

- El algoritmo de Bellman-Ford soporta aristas con pesos negativos
- Y detecta ciclos negativos en el grafo (retorna falso)
- Idea: Si relajamos V-1 vec arista obtenemos todos los caminos más cortos (tienen V-1 arist mo)



O(V)

Se ejecuta V-1 veces

O(VE)

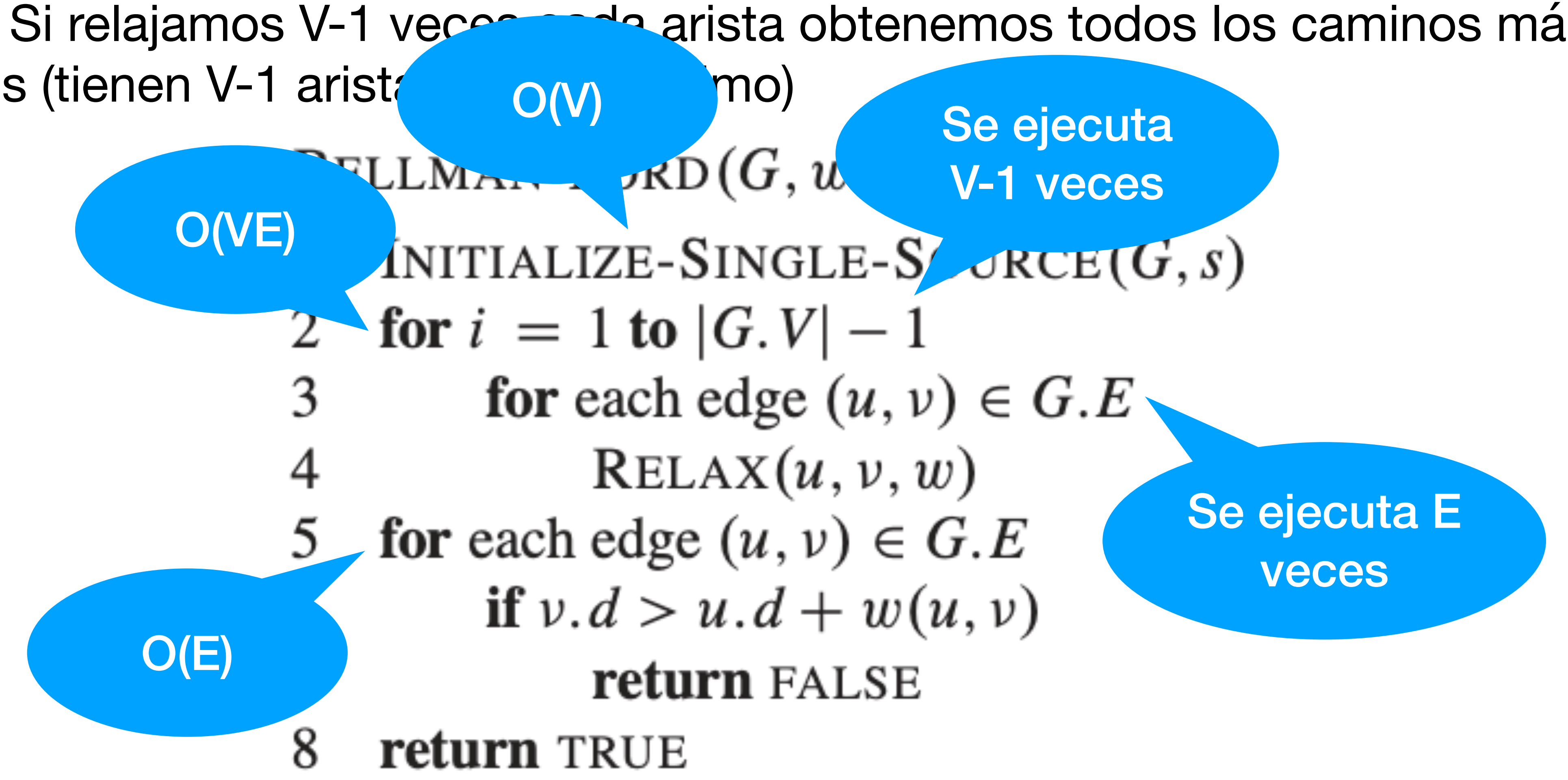
Se ejecuta E veces

```
BELLMA ORD(G, w
    INITIALIZE-SINGLE-S URCE(G, s)
    for $i = 1$ to $|G.V| - 1$
        for each edge $(u, v) \in G.E$
            RELAX$(u, v, w)$
    for each edge $(u, v) \in G.E$
        if $v.d > u.d + w(u, v)$
            return FALSE
    return TRUE
```

- Sin embargo, es mucho más ineficiente que Dijkstra (O(V log V+E log V)), ya que tarda O(V+VE+E) = O(VE) en el peor caso

Bellman-Ford

- El algoritmo de Bellman-Ford soporta aristas con pesos negativos
- Y detecta ciclos negativos en el grafo (retorna falso)
- Idea: Si relajamos V-1 veces cada arista obtenemos todos los caminos más cortos (tienen V-1 arista mo)

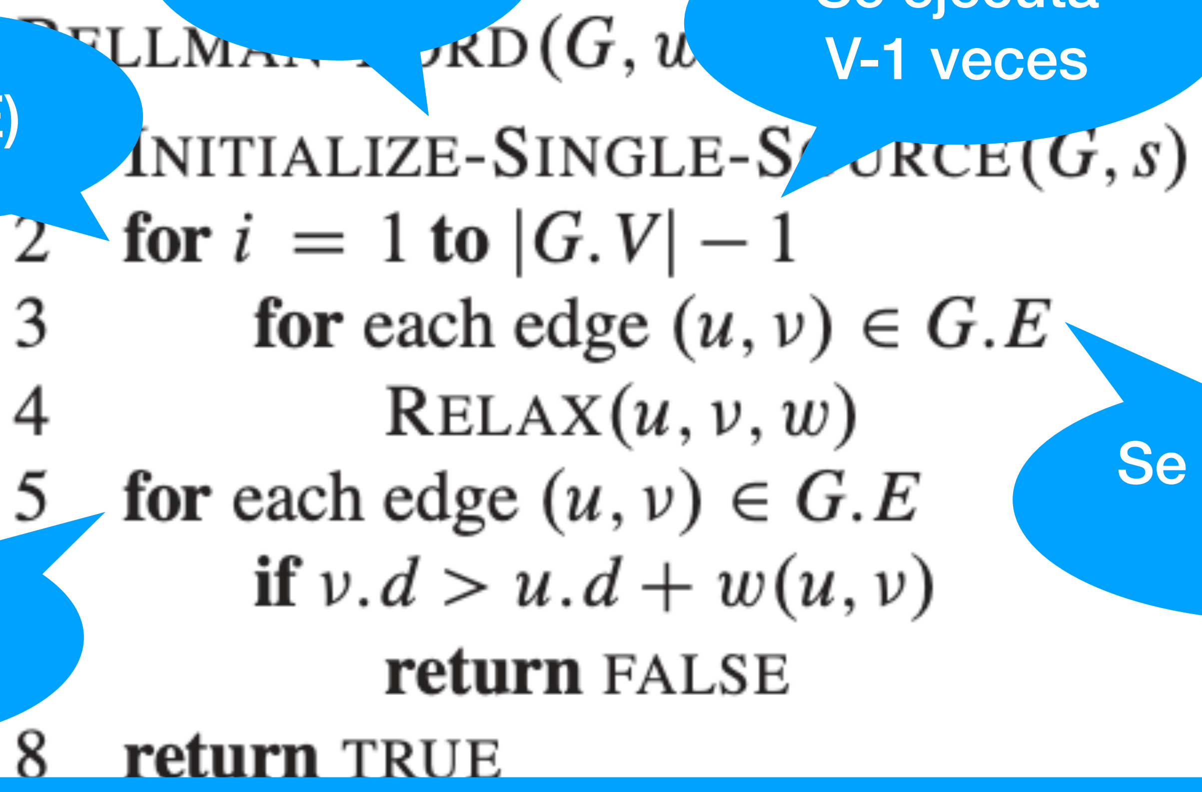


```
BELLMAN-FORD(G, w
    INITIALIZE-SINGLE-SOURCE(G, s)
    for i = 1 to |G.V| − 1
        for each edge (u, v) ∈ G.E
            RELAX(u, v, w)
    for each edge (u, v) ∈ G.E
        if v.d > u.d + w(u, v)
            return FALSE
    return TRUE
```

- Sin embargo, es mucho más ineficiente que Dijkstra (O(V log V+E log V)), ya que tarda O(V+VE+E) = O(VE) en el peor caso

Bellman-Ford

- El algoritmo de Bellman-Ford soporta aristas con pesos negativos
- Y detecta ciclos negativos en el grafo (retorna falso)
- Idea: Si relajamos V-1 vec arista obtenemos todos los caminos más cortos (tienen V-1 arist mo)



```
ELLMA  ORD(G, w
INITIALIZE-SINGLE-S URCE(G, s)
for $i$ = 1 to $|G.V| - 1$
    for each edge $(u, v) \in G.E$
        RELAX$(u, v, w)$
for each edge $(u, v) \in G.E$
    if $v.d > u.d + w(u, v)$
        return FALSE
return TRUE
```

O(V)

O(VE)

Se ejecuta V-1 veces

Se ejecuta E veces

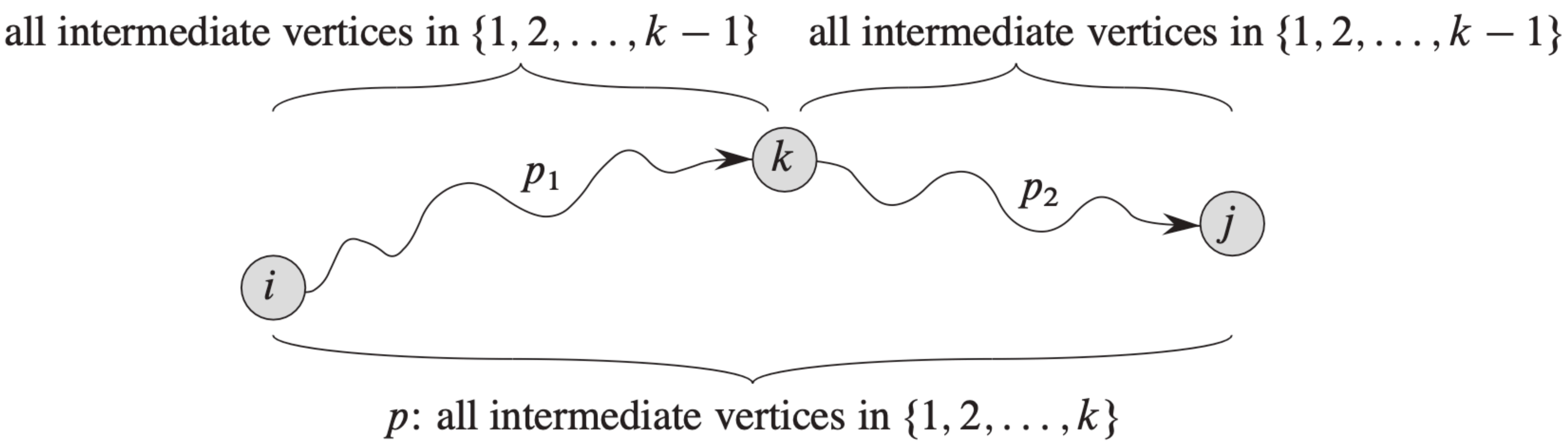
O(E)

- Sin e ya que tarda

Ejercicio: Implementar el algoritmo de Bellman-Ford y ejecutarlo a mano para algún grafo

Algoritmo de Floyd-Warshall

- El camino más corto p de i a j, pasando por los k primeros nodos, puede:
    - O bien pasar por k, es decir p = i --p1--> k --p2--> j, y todos los vértices intermedios de p1 y p2 están entre 1 y k-1

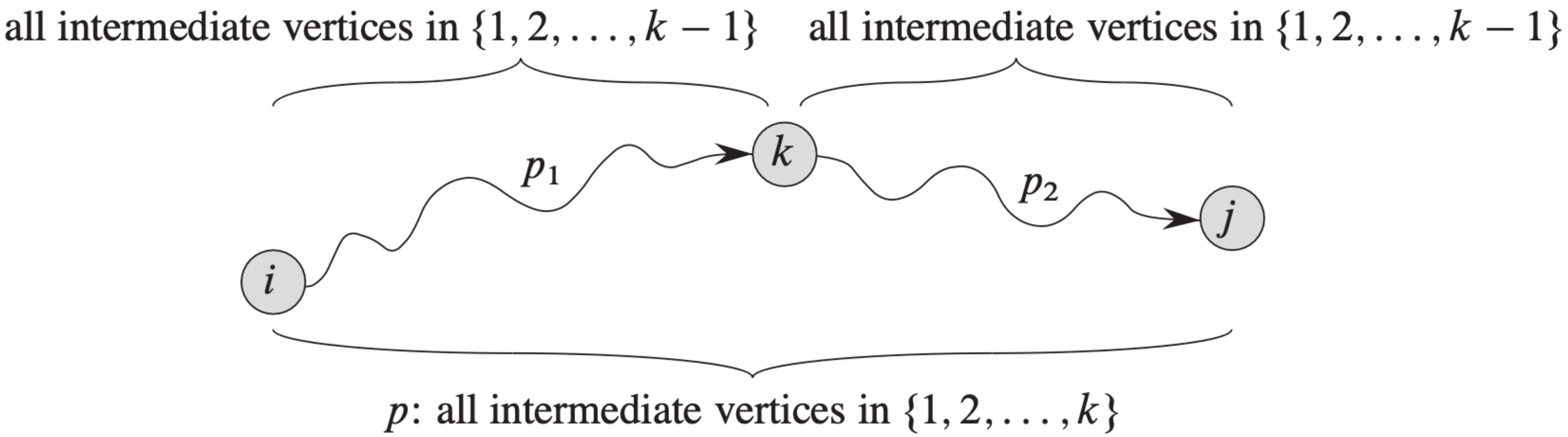


    - O no pasar por k, es decir p = i --p1--> j, y los vértices intermedios de p1 están entre 1 y k-1
- La solución recursiva a este problema viene dada por matrices $D^{(0)}, \dots, D^{(n)}$, tales que cada elemento de cada matriz se define como:

$$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{if } k = 0 \, , \\ \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right) & \text{if } k \geq 1 \, . \end{cases}$$

Algoritmo de Floyd-Warshall

- El camino más corto p de i a j, pasando por los k primeros nodos, puede:
  - O bien pasar por k, es decir p = i --p1--> k --p2--> j, y todos los vértices intermedios de p1 y p2 están entre 1 y k-1



  - O no pasar por k, es decir p = i --p1--> j, y los vértices intermedios de p1 ntre 1 y k-1

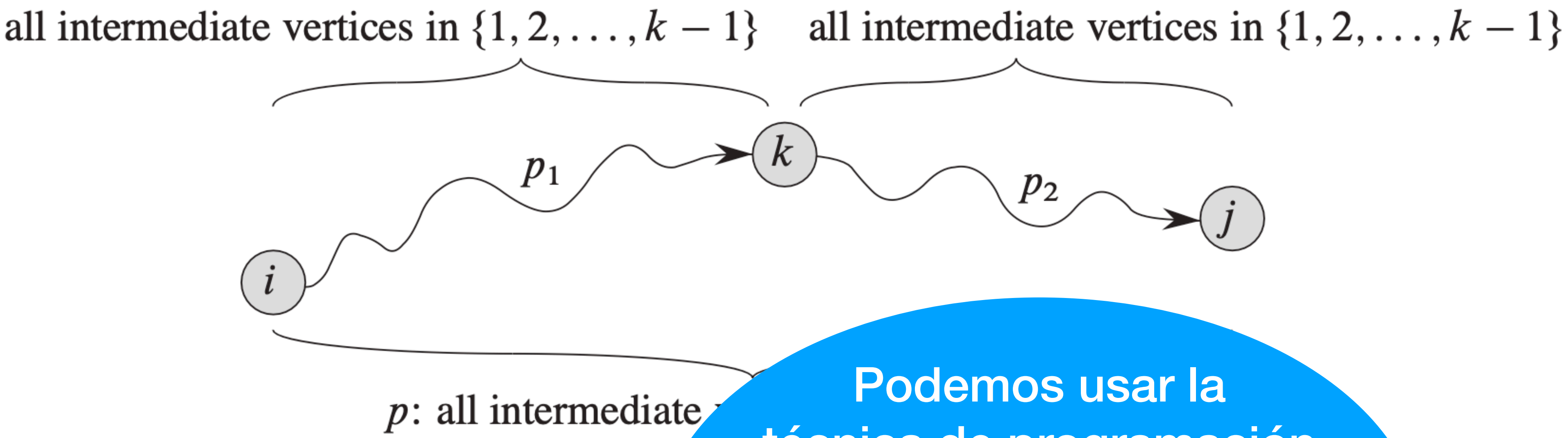
El algoritmo recursivo es $O(3^V)$

ursiva a este problema viene dada por matrices $D^{(0)}, \ldots, D^{(n)}$, a elemento de cada matriz se define como:

$$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{if } k = 0\,, \\ \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right) & \text{if } k \geq 1\,. \end{cases}$$

Algoritmo de Floyd-Warshall

- El camino más corto p de i a j, pasando por los k primeros nodos, puede:
  - O bien pasar por k, es decir p = i --p1--> k --p2--> j, y todos los vértices intermedios de p1 y p2 están entre 1 y k-1



  - O no pasar por k, es decir p = i … dios de p1 … ntre 1 y k-1

ursiva a este problem … $^{(0)}, \ldots, D^{(n)}$,
a elemento de cada matriz

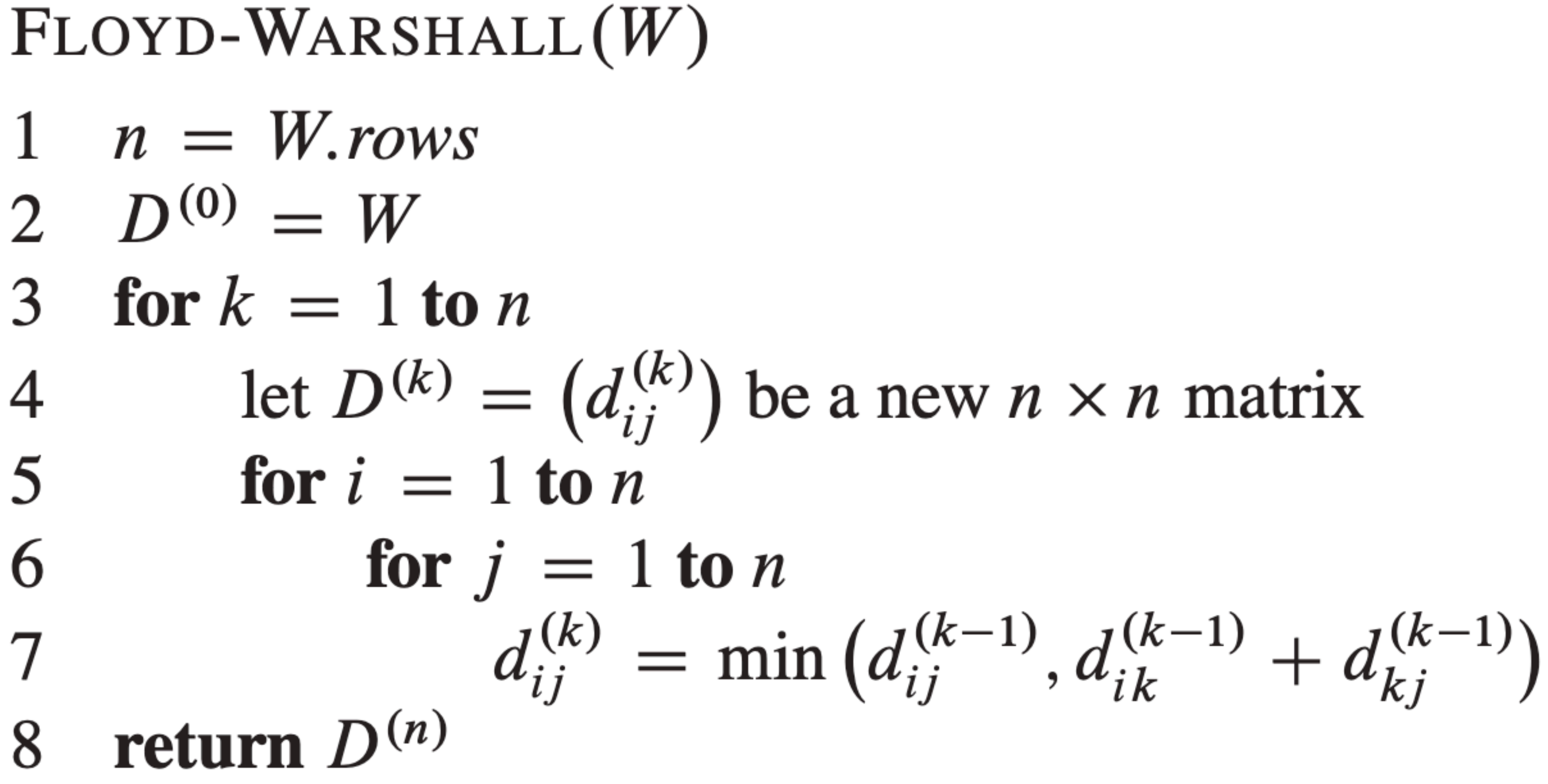
$$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{if } k = 0\,, \\ \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right) & \text{if } k \geq 1\,. \end{cases}$$

El algoritmo recursivo es $O(3^V)$

Podemos usar la técnica de programación dinámica y usar matrices para guardar los resultados intermedios, esto da un algoritmo $O(V^3)$

Algoritmo de Floyd-Warshall

- El algoritmo de Floyd-Warshall computa los caminos más cortos entre todos los pares de nodos del grafo (en lugar de a partir de un origen)
- Para esto computa las matrices $D^{(0)}, \ldots, D^{(n)}$ iterativamente:



```
FLOYD-WARSHALL($W$)
$n = W.rows$
$D^{(0)} = W$
for $k = 1$ to $n$
    let $D^{(k)} = \left(d_{ij}^{(k)}\right)$ be a new $n \times n$ matrix
    for $i = 1$ to $n$
        for $j = 1$ to $n$
            $d_{ij}^{(k)} = \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right)$
return $D^{(n)}$
```

- El algoritmo usa la técnica de programación dinámica: guarda en matrices los resultados intermedios de lo que de otra manera serían llamadas recursivas (el algoritmo recursivo es exponencial)

Algoritmo de Floyd-Warshall

- El algoritmo de Floyd-Warshall computa los caminos más cortos entre todos los pares de nodos del grafo (en lugar de a partir de un origen)

$O(V^3)$: Es rápido solo para grafos de tamaño moderado

puta las matrices $D^{(0)}, \dots, D^{(n)}$ iterativamente:

```
-WARSHALL$(W)$
    $= W.rows$
    $D^{(0)} = W$
for $k = 1$ to $n$
    let $D^{(k)} = \left(d_{ij}^{(k)}\right)$ be a new $n \times n$ matrix
    for $i = 1$ to $n$
        for $j = 1$ to $n$
            $d_{ij}^{(k)} = \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right)$
return $D^{(n)}$
```

- El algoritmo usa la técnica de programación dinámica: guarda en matrices los resultados intermedios de lo que de otra manera serían llamadas recursivas (el algoritmo recursivo es exponencial)

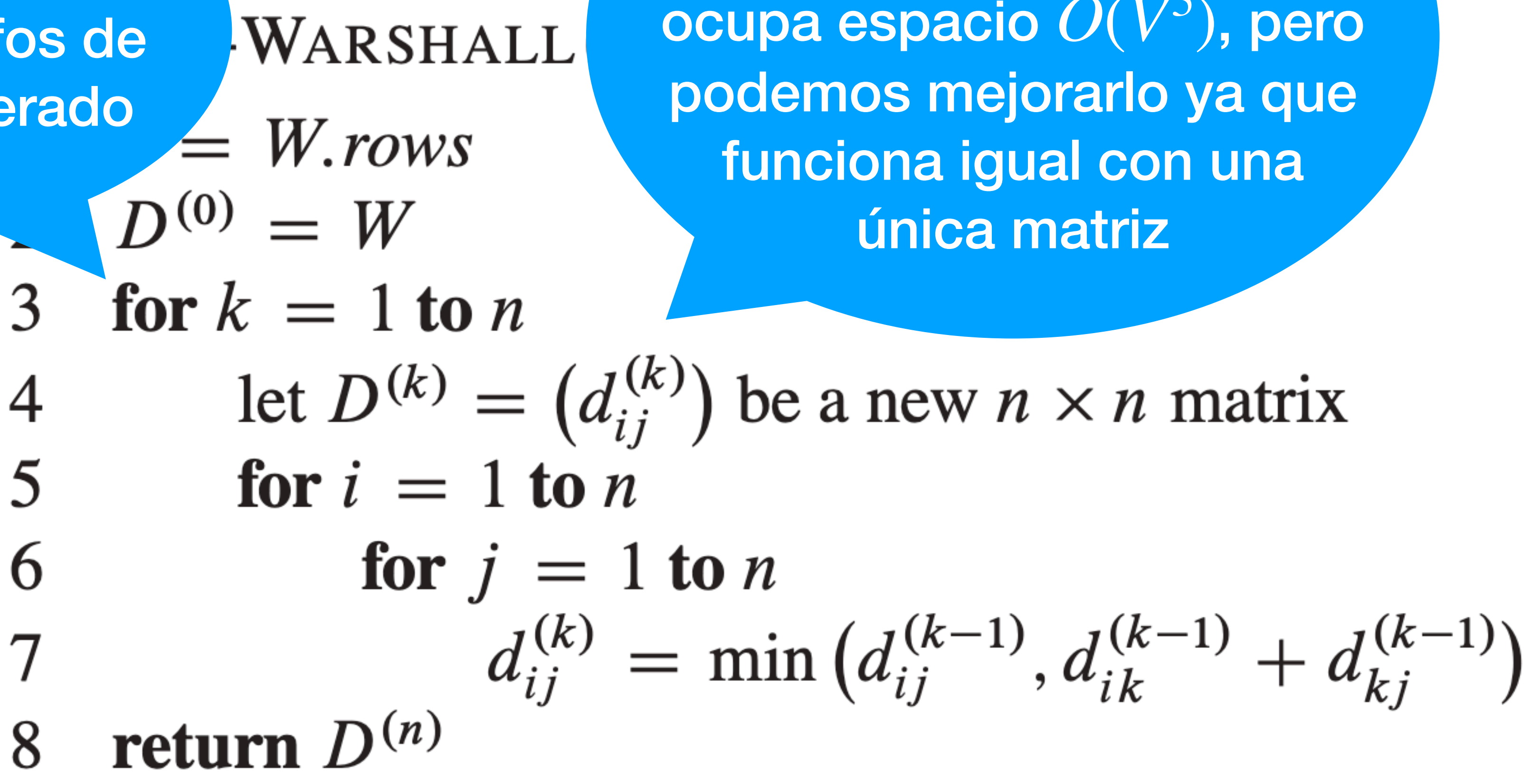
Algoritmo de Floyd-Warshall

- El algoritmo de Floyd-Warshall computa los caminos más cortos entre todos los pares de nodos del grafo (rtir de un origen)

puta las matrice nte:

$O(V^3)$: Es rápido solo para grafos de tamaño moderado

Requiere V matrices de tamaño V^2, es decir ocupa espacio $O(V^3)$, pero podemos mejorarlo ya que funciona igual con una única matriz



```
-WARSHALL
  = $W.rows$
  $D^{(0)} = W$
for $k = 1$ to $n$
    let $D^{(k)} = (d_{ij}^{(k)})$ be a new $n \times n$ matrix
    for $i = 1$ to $n$
        for $j = 1$ to $n$
            $d_{ij}^{(k)} = \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right)$
return $D^{(n)}$
```

- El algoritmo usa la técnica de programación dinámica: guarda en matrices los resultados intermedios de lo que de otra manera serían llamadas recursivas (el algoritmo recursivo es exponencial)

Algoritmo de Floyd-Warshall

- El algoritmo de Floyd-Warshall computa los caminos más cortos entre todos los pares de nodos del grafo (rtir de un origen)

puta las matrice te:

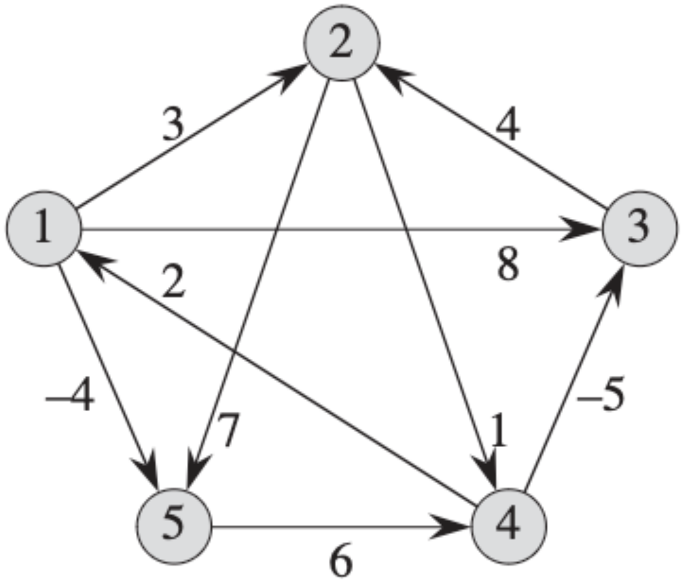
```
-WARSHALL
= $W.rows$
$D^{(0)} = W$
for $k = 1$ to $n$
    let $D^{(k)} = (d_{ij}^{(k)})$ be a new $n \times n$ matrix
    for $i = 1$ to $n$
        for $j = 1$ to
            $d_{ij}^{(k)}$ =   $_{j}^{(k-1)})$
return $D^{(n)}$
```

$O(V^3)$: Es rápido solo para grafos de tamaño moderado

Requiere V matrices de tamaño V^2, es decir ocupa espacio $O(V^3)$, pero podemos mejorarlo ya que funciona igual con una única matriz

Podemos mejorarlo observando que funciona igual si trabajamos sobre una única matriz

- El algoritmo usa la técnica de prog guarda en matrices los resultados intermedios de lo que de otra manera serían llamadas recursivas (el algoritmo recursivo es exponencial)



Para computar caminos entre nodos consideramos dos casos:

- Si pasamos por k en el camino de i a j (i-->k-->j), el predecesor de j es el predecesor en el camino de k a j
- Sino el predecesor no cambia

$$\pi_{ij}^{(k)} = \begin{cases} \pi_{ij}^{(k-1)} & \text{if } d_{ij}^{(k-1)} \le d_{ik}^{(k-1)} + d_{kj}^{(k-1)} , \\ \pi_{kj}^{(k-1)} & \text{if } d_{ij}^{(k-1)} > d_{ik}^{(k-1)} + d_{kj}^{(k-1)} . \end{cases}$$

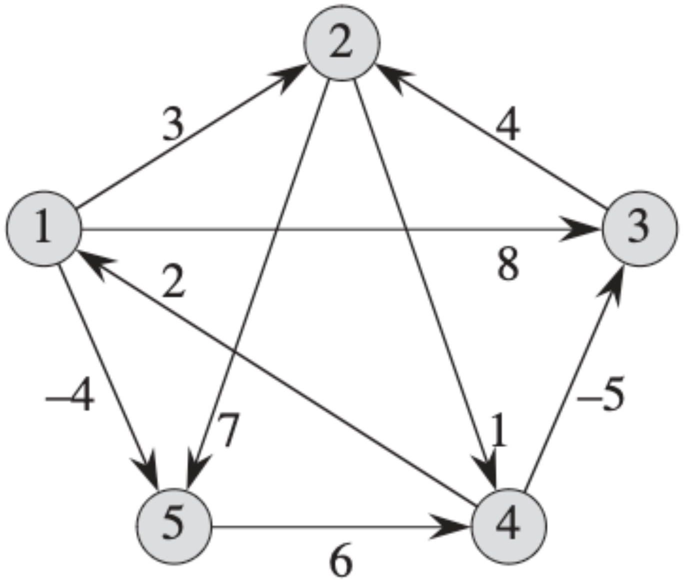
Distancias

Caminos

$$D^{(0)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \qquad \Pi^{(0)} = \begin{pmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & \text{NIL} & 4 & \text{NIL} & \text{NIL} \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(1)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \qquad \Pi^{(1)} = \begin{pmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & 1 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

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



Para computar caminos entre nodos consideramos dos casos:

- Si pasamos por k en el camino de i a j (i-->k-->j), el predecesor de j es el predecesor en el camino de k a j
- Sino el predecesor no cambia

$$\pi_{ij}^{(k)} = \begin{cases} \pi_{ij}^{(k-1)} & \text{if } d_{ij}^{(k-1)} \le d_{ik}^{(k-1)} + d_{kj}^{(k-1)} , \\ \pi_{kj}^{(k-1)} & \text{if } d_{ij}^{(k-1)} > d_{ik}^{(k-1)} + d_{kj}^{(k-1)} . \end{cases}$$

Distancias

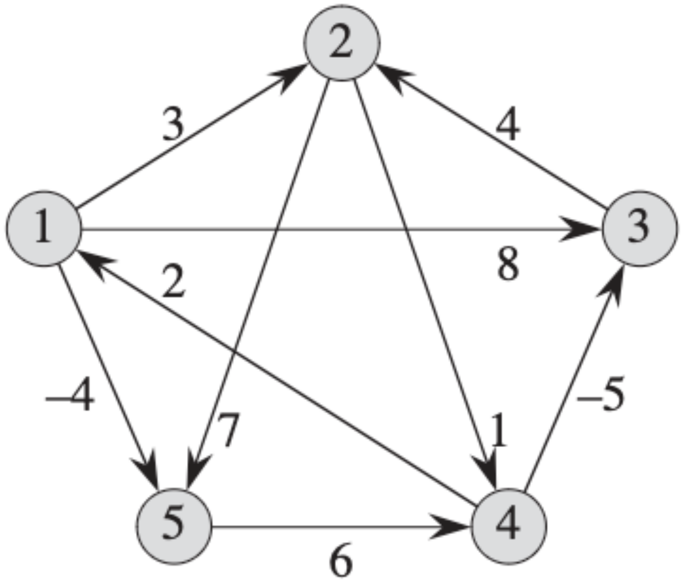
Caminos

$$D^{(0)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \qquad \Pi^{(0)} = \begin{pmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & \text{NIL} & 4 & \text{NIL} & \text{NIL} \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

Comenzamos con w_{ij} para cada arista i->j, 0 para la distancia de i a i, e infinito para el resto

$$\begin{pmatrix} & 3 & 8 & \infty & -4 \\ & 0 & \infty & 1 & 7 \\ & 4 & 0 & \infty & \infty \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \qquad \Pi^{(1)} = \begin{pmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & 1 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

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



Para computar caminos entre nodos consideramos dos casos:

- Si pasamos por k en el camino de i a j (i-->k-->j), el predecesor de j es el predecesor en el camino de k a j
- Sino el predecesor no cambia

$$\pi_{ij}^{(k)} = \begin{cases} \pi_{ij}^{(k-1)} & \text{if} \quad \ldots \quad d_{kj}^{(k-1)} , \\ \pi_{kj}^{(k-} \ldots & \ldots -1) . \end{cases}$$

Hay 0 nodos intermedios, por lo que $\Pi_{ij} = i$ para cada arista i->j, o null en caso contrario

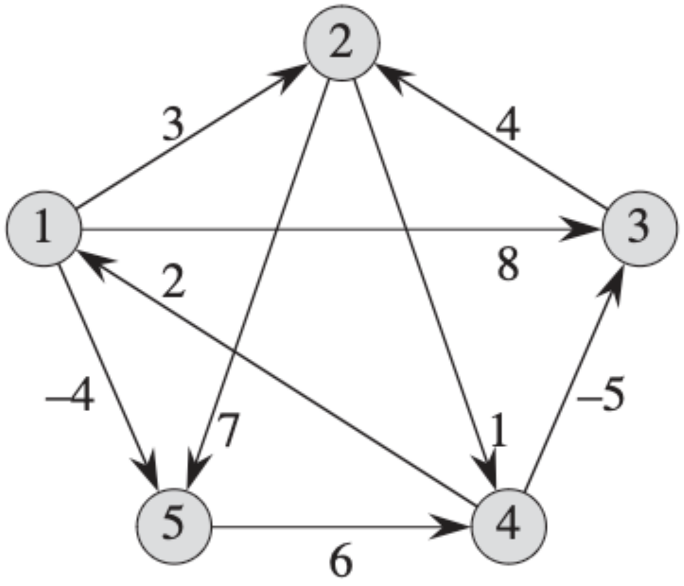
Distancias ...os

$$D^{(0)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \qquad \Pi^{(0)} = \begin{pmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & \text{NIL} & 4 & \text{NIL} & \text{NIL} \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

Comenzamos con w_{ij} para cada arista i->j, 0 para la distancia de i a i, e infinito para el resto

$$\begin{pmatrix} & 3 & 8 & \infty & -4 \\ & 0 & \infty & 1 & 7 \\ & 4 & 0 & \infty & \infty \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \qquad \Pi^{(1)} = \begin{pmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & 1 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

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



Para computar caminos entre nodos consideramos dos casos:

- Si pasamos por k en el camino de i a j (i-->k-->j), el predecesor de j es el predecesor en el camino de k a j
- Sino el predecesor no cambia

$$\pi_{ij}^{(k)} = \begin{cases} \pi_{ij}^{(k-1)} & d_{kj}^{(k-1)} , \\ \pi_{kj} & -1) . \end{cases}$$

Hay 0 nodos intermedios, por lo que $\Pi_{ij} = i$ para cada arista i->j, o null en caso contrario

Distancias

os

$$D^{(0)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & & & \\ \infty & & & & \\ 2 & & & & \\ \infty & & & & \end{pmatrix}$$

$$\Pi^{(0)} = \begin{pmatrix} \mathrm{NIL} & 1 & 1 & \mathrm{NIL} & 1 \\ \mathrm{NIL} & \mathrm{NIL} & \mathrm{NIL} & 2 & 2 \\ \mathrm{NIL} & 3 & \mathrm{NIL} & \mathrm{NIL} & \mathrm{NIL} \\ 4 & \mathrm{NIL} & 4 & \mathrm{NIL} & \mathrm{NIL} \\ \mathrm{NIL} & \mathrm{NIL} & \mathrm{NIL} & 5 & \mathrm{NIL} \end{pmatrix}$$

Cada vez que pasamos por k para ir de i a j, hacemos $\Pi_{ij} = \Pi_{kj}$

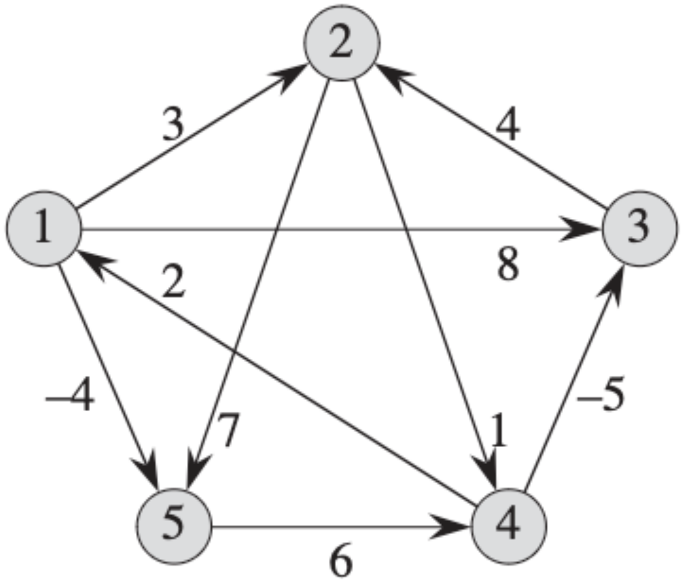
Comenzamos con w_{ij} para cada arista i->j, 0 para la distancia de i a i, e infinito para el resto

$$\begin{pmatrix} & 3 & & & \\ & 0 & \infty & 1 & 7 \\ & 4 & 0 & \infty & \infty \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix}$$

$$\Pi^{(1)} = \begin{pmatrix} \mathrm{NIL} & 1 & 1 & \mathrm{NIL} & 1 \\ \mathrm{NIL} & \mathrm{NIL} & \mathrm{NIL} & 2 & 2 \\ \mathrm{NIL} & 3 & \mathrm{NIL} & \mathrm{NIL} & \mathrm{NIL} \\ 4 & 1 & 4 & \mathrm{NIL} & 1 \\ \mathrm{NIL} & \mathrm{NIL} & \mathrm{NIL} & 5 & \mathrm{NIL} \end{pmatrix}$$

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

$$\Pi^{(2)} = \begin{pmatrix} \mathrm{NIL} & 1 & 1 & 2 & 1 \\ \mathrm{NIL} & \mathrm{NIL} & \mathrm{NIL} & 2 & 2 \\ \mathrm{NIL} & 3 & \mathrm{NIL} & 2 & 2 \\ 4 & 1 & 4 & \mathrm{NIL} & 1 \\ \mathrm{NIL} & \mathrm{NIL} & \mathrm{NIL} & 5 & \mathrm{NIL} \end{pmatrix}$$



Distancias

Caminos

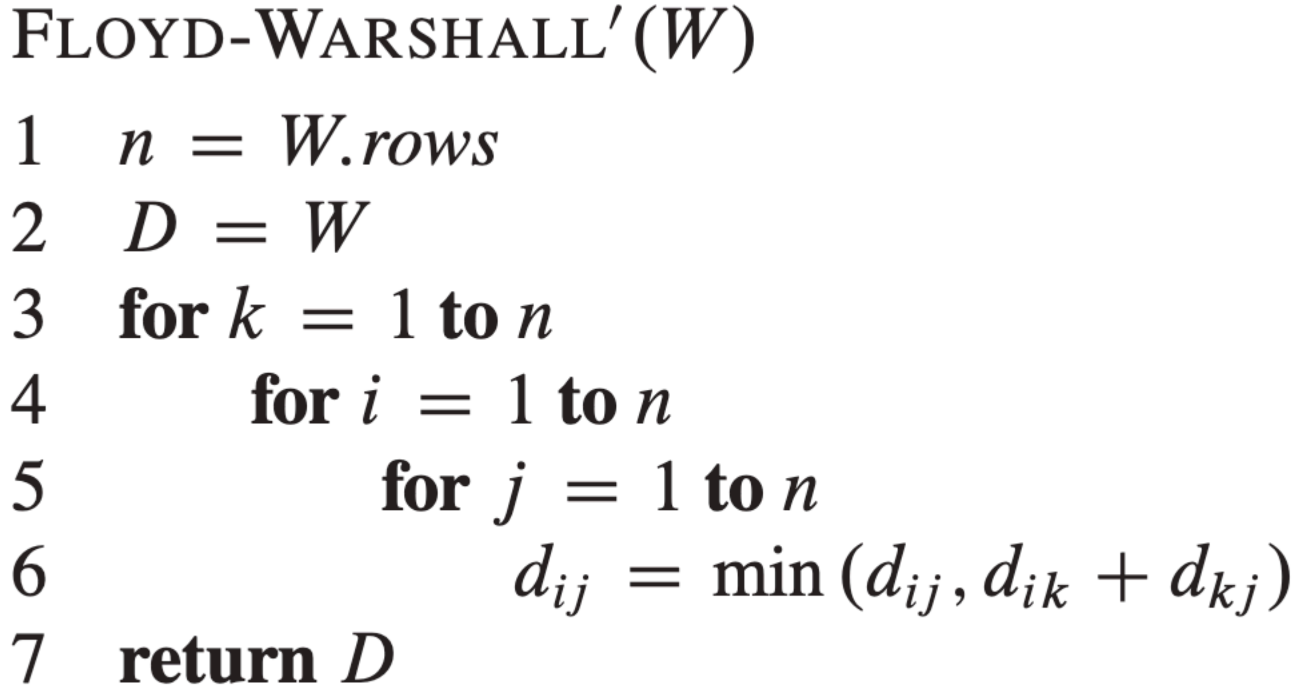
$$D^{(2)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \qquad \Pi^{(2)} = \begin{pmatrix} \text{NIL} & 1 & 1 & 2 & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL}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

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

$$D^{(4)} = \begin{pmatrix} 0 & 3 & -1 & 4 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \qquad \Pi^{(4)} = \begin{pmatrix} \text{NIL} & 1 & 4 & 2 & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(5)} = \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \qquad \Pi^{(5)} = \begin{pmatrix} \text{NIL} & 3 & 4 & 5 & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{pmatrix}$$

Algoritmo de Floyd-Warshall

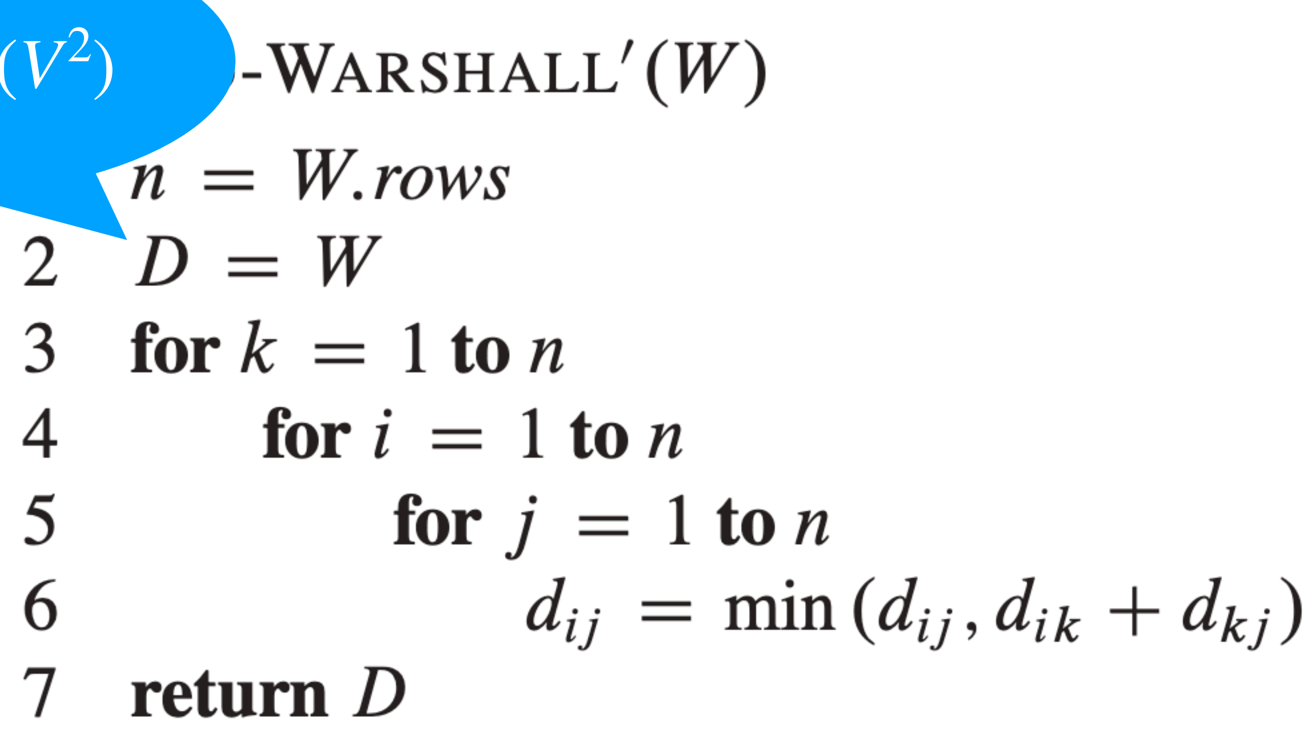


```
FLOYD-WARSHALL′($W$)
$n = W.rows$
$D = W$
for $k = 1$ to $n$
    for $i = 1$ to $n$
        for $j = 1$ to $n$
            $d_{ij} = \min(d_{ij}, d_{ik} + d_{kj})$
return $D$
```

- Ventajas: El algoritmo de Floyd-Warshall soporta arcos con costo negativo
  - Y puede modificarse para detectar ciclos infinitos
- Para grafos densos (donde E es $O(V^2)$), el tiempo de ejecución de Floyd-Warshall ($O(V^3)$) es mejor que Dijkstra aplicado una vez a cada vértice ($O(V^3 logV)$)
  - Ejecutar Dijkstra V veces en un grafo denso es $O(V * V^2 logV)$, es decir, $O(V^3 logV)$

Algoritmo de Floyd-Warshall

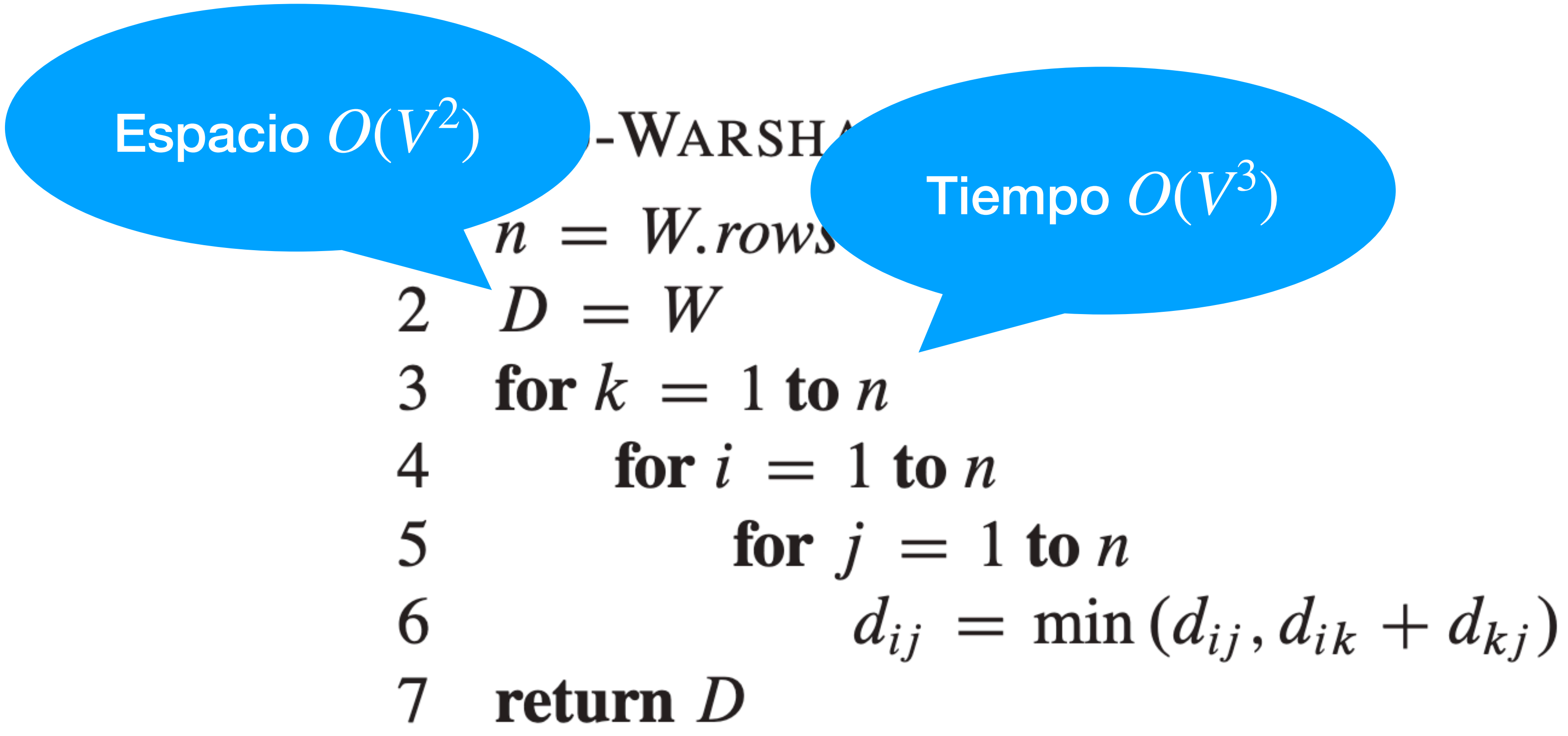
Espacio $O(V^2)$



```
-WARSHALL′(W)
$n = W.rows$
$D = W$
for $k = 1$ to $n$
    for $i = 1$ to $n$
        for $j = 1$ to $n$
            $d_{ij} = \min(d_{ij}, d_{ik} + d_{kj})$
return $D$
```

- Ventajas: El algoritmo de Floyd-Warshall soporta arcos con costo negativo
  - Y puede modificarse para detectar ciclos infinitos
- Para grafos densos (donde E es $O(V^2)$), el tiempo de ejecución de Floyd-Warshall ($O(V^3)$) es mejor que Dijkstra aplicado una vez a cada vértice ($O(V^3 log V)$)
  - Ejecutar Dijkstra V veces en un grafo denso es $O(V * V^2 log V)$, es decir, $O(V^3 log V)$

Algoritmo de Floyd-Warshall



Espacio $O(V^2)$

Tiempo $O(V^3)$

```
-WARSHA
n = W.rows
D = W
for k = 1 to n
    for i = 1 to n
        for j = 1 to n
            $d_{ij} = \min(d_{ij}, d_{ik} + d_{kj})$
return D
```

- Ventajas: El algoritmo de Floyd-Warshall soporta arcos con costo negativo
  - Y puede modificarse para detectar ciclos infinitos
- Para grafos densos (donde E es $O(V^2)$), el tiempo de ejecución de Floyd-Warshall ($O(V^3)$) es mejor que Dijkstra aplicado una vez a cada vértice ($O(V^3 logV)$)
  - Ejecutar Dijkstra V veces en un grafo denso es $O(V * V^2 logV)$, es decir, $O(V^3 logV)$

Algoritmo de Floyd-Warshall: Java

```java
/**
 * @post Computes shortest paths from each vertex to every other
 * vertex in the edge-weighted digraph G.
 */
public void FloydWarshall() {
    int V = G.V();
    distTo = new double[V][V];
    edgeTo = new DirectedEdge[V][V];
    // initialize distances to infinity
    for (int v = 0; v < V; v++) {
        distTo[v][v] = 0.0;
        for (int w = 0; w < V; w++) {
            if (v != w)
                distTo[v][w] = Double.POSITIVE_INFINITY;
        }
    }
    // initialize distances using edge-weighted digraph's
    for (int v = 0; v < G.V(); v++) {
        for (DirectedEdge e : G.adj(v)) {
            distTo[e.from][e.to] = e.weight;
            edgeTo[e.from][e.to] = e;
        }
    }
    // continues...
```

Algoritmo de Floyd-Warshall: Java

```
/**
 * @post Computes shortest paths from each vertex to every other
 * vertex in the edge-weighted digraph G.
 */
public void FloydWarshall() {
    int V = G.V();
    distTo = new double[V][V];
    edgeTo = new DirectedEdge[V][V];
    // initialize distances to infinity
    for (int v = 0;
        distTo[v][
        for (int w
            if (v !
                dis
        }
    }
    // initialize
    for (int v = 0;
        for (Direct
            distTo[
            edgeTo[
        }
    }
    // continues...
```

```
    // continues...
    // Floyd-Warshall updates
    for (int k = 0; k < V; k++) {
        // compute shortest paths using only 0, 1, ..., k as
        // intermediate vertices
        for (int i = 0; i < V; i++) {
            for (int j = 0; j < V; j++) {
                if (distTo[i][j] > distTo[i][k] + distTo[k][j]) {
                    distTo[i][j] = distTo[i][k] + distTo[k][j];
                    edgeTo[i][j] = edgeTo[k][j];
                }
            }
        }
    }
}
```

Actividades

- Leer los capítulos 24 y 25 del libro "Introduction to Algorithms, 3rd Edition". T. Cormen, C. Leiserson, R. Rivest, C. Stein. MIT Press. 2009

Bibliografía

- "Algorithms (4th edition)". R. Sedgewick, K. Wayne. Addison-Wesley. 2016
- "Introduction to Algorithms, 3rd Edition". T. Cormen, C. Leiserson, R. Rivest, C. Stein. MIT Press. 2009
- "Data Structures and Algorithms". A. Aho, J. Hopcroft, J. Ullman. Addison-Wesley. 1983