AED3 > Clase 7 >
Camino mínimo. Uno a todos.

Repaso

Dado un grafo $G=(V, E, w)$ con $w: E \longrightarrow R$

- Costo: $w(T) = \sum_T w(e)$... Como un abuso de notación se usa w tanto para el costo de una arista como de todo el árbol.
- Para los grafos no pesados todo AG es AGM porque $w=1 \Rightarrow \sum_T w=m=n-1$
- También puede haber varios AGM.

Topología de problemas de camino mínimo

1. uno a uno
2. uno a muchos
3. muchos a muchos

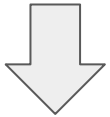
Problemas de camino mínimo con un único origen

Definición 1: Camino mínimo elemental

Dados $u,v \in G$, quiero un camino entre u y v (P_{uv}),

tal que $w(P_{uv}) = d(u,v)$

(sea mínimo).



No se conocen algoritmos polinomiales para este problema,

pero …

Definición 2: Camino mínimo no elemental

Dado $u \in G$, un origen, quiero los caminos a todos lo $v \in G$ alcanzables,

tenemos que suponer que G es conexo (o que estamos en una componente conexa)

Problemas de camino mínimo con un único origen

Ejemplos de problemas de camino mínimo…

Problemas de camino mínimo con un único origen

Teorema 1: Subestructura óptima

Sea $G = (V, E, w)$ un digrafo, $v \in G$ un origen. Sea $P = v_1 \dots v_k$ un camino mínimo de G. Entonces,

$\forall i,j\ 1 \leq i \leq j \leq k$, P_{ij} es un camino mínimo de v_i a v_j.

Demo (por Absurdo):

$P = v_1 \dots v_i \dots v_j \dots v_k = P_{1i} + P_{ij} + P_{jk}$

$\Rightarrow w(P) = w(P_{1i}) + w(P_{ij}) + w(P_{jk})$

Si existe Q de v_i a v_j tq $w(Q) \leq w(P_{ij})$

$\Rightarrow w(P') = w(P_{1i}) + w(Q) + w(P_{jk}) \leq w(P)$

Problemas de camino mínimo con un único origen

Teorema 1: Subestructura óptima

Sea $G = (V, E, w)$ un digrafo, $v \in G$ un origen. Sea $P = v_1 \dots v_k$ un camino mínimo de G. Entonces,

$\forall i,j \; 1 \leq i \leq j \leq k$, P_{ij} es un camino mínimo de v_i a v_j.

OJO!

Aristas negativas.

Ciclos

Ciclos de peso negativo w(C)<0	⇒	Problema mal definido
Ciclos de peso positivo w(C)>0	⇒	No es parte del camino mínimo
Ciclos de peso nulo w(C)=0	⇒	Seguro existe un camino alternativo que no use este ciclo

Topología de problemas de camino mínimo

1. uno a uno
2. uno a muchos
3. muchos a muchos

¿Tienen aristas negativas?

¿Tienen ciclos?

¿Cómo es el peso de los ciclos?

1. Negativo
2. Positivo
3. Nulo

Algoritmos de camino mínimo con un único origen

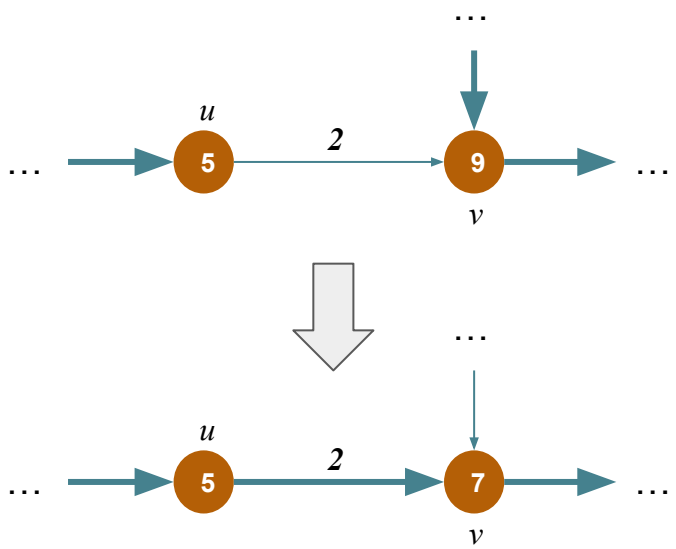
Input: $G = (V, E, w)$ y $s \in G$ un origen

Output: Distancias de cada nodo u a s ($u.d$) y la estructura de los caminos mínimos o árbol de caminos mínimos ($u.pred$)

Algoritmos de camino mínimo con un único origen

Propiedad de RELAJACIÓN

```
RELAX ( u , v, w ) :
|     if v.d > u.d + w(u,v):
|     |     v.d = u.d + w(u,v)
|     |     v.pred = u
```



Teorema 1: Subestructura óptima

Sea $G = (V, E, w)$ un digrafo, $v \in G$ un origen. Sea $P = v_1 \dots v_k$ un camino mínimo de G. Entonces,

$\forall i,j\ 1 \le i \le j \le k$, P_{ij} es un camino mínimo de v_i a v_j.

Algoritmos de camino mínimo con un único origen

$d(u, v)$ distancia estimada entre u y v, $\delta(u, v)$ distancia mínima (real) entre u y v

Desigualdad triangular Sea $(u , v) \in E$, $\delta(s, v) \leq \delta(s, u) + w(u, v)$

Límite superior Sea $v \in V$, $d(s, v) \geq \delta(s, v)$, y una vez que $d(s, v) = \delta(s, v)$ ya no cambia.

Nodos no alcanzables Si no hay camino de s a v, entonces $d(s, v) = \delta(s, v) = \infty$.

Convergencia Sea P = s→…→ u → v un camino mínimo de s a v y $d(s, u) = \delta(s, u)$ en un paso dado. Entonces, luego de relajar (u, v), ocurre que $d(s, v) = \delta(s, v)$.

Relajación Si $P = v_0, v_1, \ldots , v_k$ es un camino mínimo de $s = v_0$ a v_k, y se relajan los ejes en orden $(v_0, v_1), (v_1, v_2), \ldots , (v_{k-1}, v_k)$, entonces $d(s, v_k) = \delta(s, v_k)$. Esta propiedad se mantiene aunque se relajen otras aristas en el medio.

Subgrafo de predecesores Si se cumple que $d(s, v) = \delta(s, v) \ \forall \ v \in V$, entonces el subgrafo que forman los predecesores es un s-ACM.

Relajación

$d(u, v)$ distancia estimada entre u y v, $\delta(u, v)$ distancia mínima (real) entre u y v

Relajación Si $P = v_0, v_1, \dots, v_k$ es un camino mínimo de $s = v_0$ a v_k, y se relajan los ejes en orden $(v_0, v_1), (v_1, v_2), \dots, (v_{k-1}, v_k)$, entonces $d(s, v_k) = \delta(s, v_k)$. Esta propiedad se mantiene aunque se relajen otras aristas en el medio.

Demo: (inducción)

Caso base: $P = v_0 \Rightarrow d(s = v_0, v_0) = \delta(v_0, v_0) = 0$. Por la prop. del **Límite superior** $d(v_0, v_0)$ ya no cambia.

Hipótesis inductiva: $P = v_0, v_1, \dots, v_{i-1}$ con $d(s, v_{i-1}) = \delta(s, v_{i-1})$

Paso inductivo: voy a relajar (v_{i-1}, v_i). Por la prop. de **Convergencia**, si se cumple la Hipótesis inductiva $\Rightarrow d(v_0, v_i) = \delta(v_0, v_i)$. Por la prop. del **Límite superior** $d(v_0, v_i)$ ya no cambia.

Repaso: DAGs

```
RELAX ( u , v, w ) :
|     if v.d > u.d + w(u,v):
|     |     v.d = u.d + w(u,v)
|     |     v.pred = u
```

```
INIT ( G, s ) :
|     for v in V:
|     |     v.d = Inf
|     |     v.pred = None
s.d = 0
```

```
DAG ( G, s ) :
|     TOPOLOGICAL-SORT(G ):
|     INIT(G, s)
|     for u in V (en el orden de TOPOLOGICAL-SORT:
|     |     for v in Adj[u]:
|     |     |     RELAX(u, v)
```

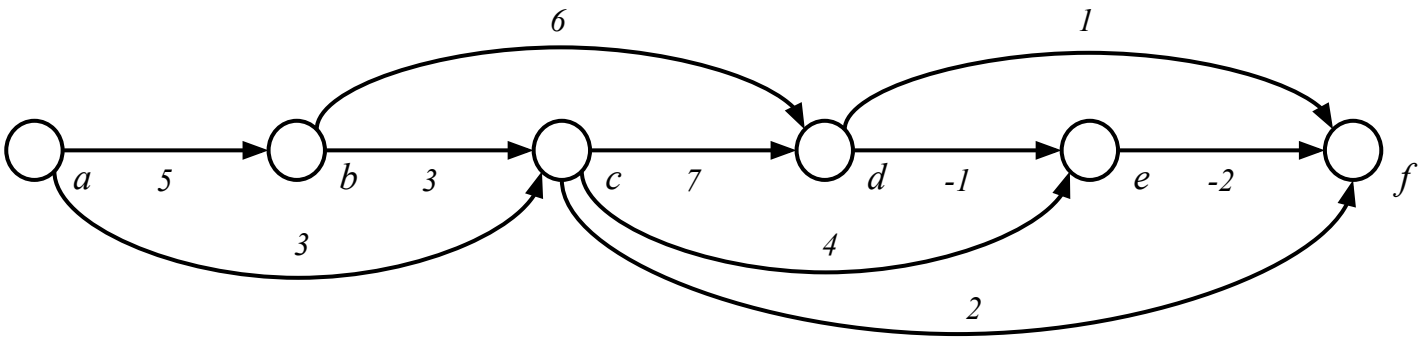
Repaso: DAGs

¿Cuánto tiempo me va a llevar la tarea?

¿Cuál es el cuello de botella?

```
DAG ( G, s ) :
|     TOPOLOGICAL-SORT(G ):
|     INIT(G, s)
|     for u in V (en el orden de TOPOLOGICAL-SORT:
|     |     for v in Adj[u]:
|     |     |     RELAX(u, v)
```

```
RELAX ( u , v, w ) :
|     if v.d > u.d + w(u,v):
|     |     v.d = u.d + w(u,v)
|     |     v.pred = u
```

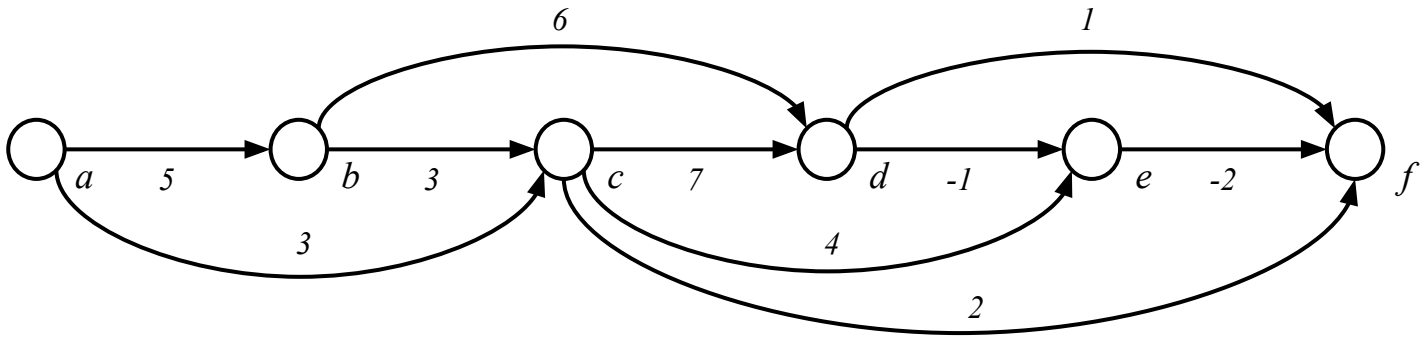


Repaso: DAGs

TOPOLOGICAL-SORT: a→b→c→d→e→f

```
DAG ( G, s ) :
|     TOPOLOGICAL-SORT(G ):
|     INIT(G, s)
|     for u in V (en el orden de TOPOLOGICAL-SORT:
|     |     for v in Adj[u]:
|     |     |     RELAX(u, v)
```

```
RELAX ( u , v, w ) :
|     if v.d > u.d + w(u,v):
|     |     v.d = u.d + w(u,v)
|     |     v.pred = u
```



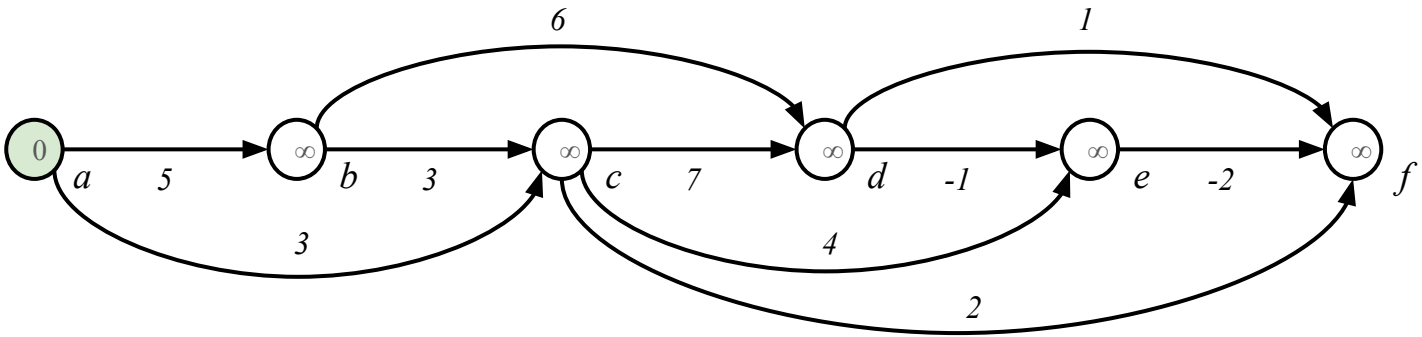
Repaso: DAGs

TOPOLOGICAL-SORT: a→b→c→d→e→f

```
DAG ( G, s ) :
|     TOPOLOGICAL-SORT(G ):
|     INIT(G, s)
|     for u in V (en el orden de TOPOLOGICAL-SORT:
|     |     for v in Adj[u]:
|     |     |     RELAX(u, v)
```

```
INIT ( G, s ) :
|     for v in V:
|     |     v.d = Inf
|     |     v.pred = None
s.d = 0
```

```
RELAX ( u , v, w ) :
|     if v.d > u.d + w(u,v):
|     |     v.d = u.d + w(u,v)
|     |     v.pred = u
```

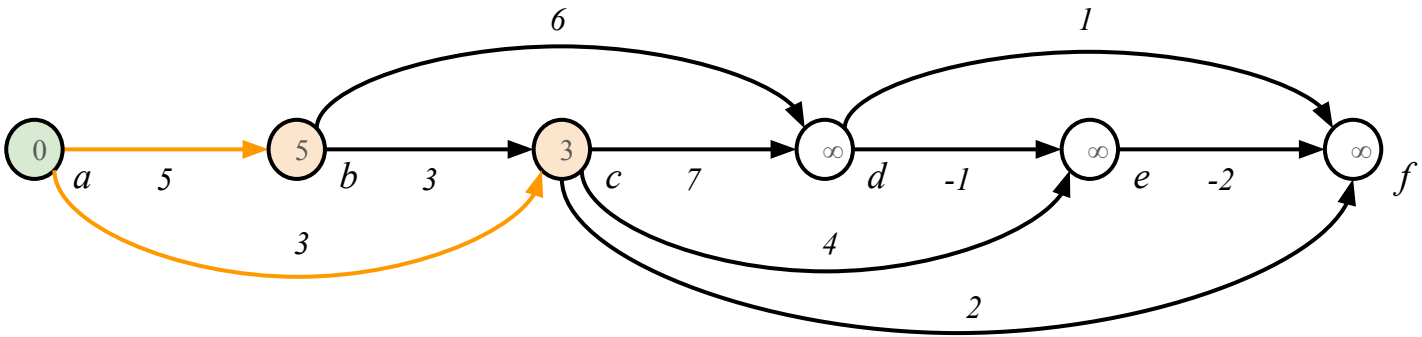


Repaso: DAGs

TOPOLOGICAL-SORT: a→b→c→d→e→f

```
DAG ( G, s ) :
|     TOPOLOGICAL-SORT(G ):
|     INIT(G, s)
|     for u in V (en el orden de TOPOLOGICAL-SORT:
|     |     for v in Adj[u]:
|     |     |     RELAX(u, v)
```

```
RELAX ( u , v, w ) :
|     if v.d > u.d + w(u,v):
|     |     v.d = u.d + w(u,v)
|     |     v.pred = u
```

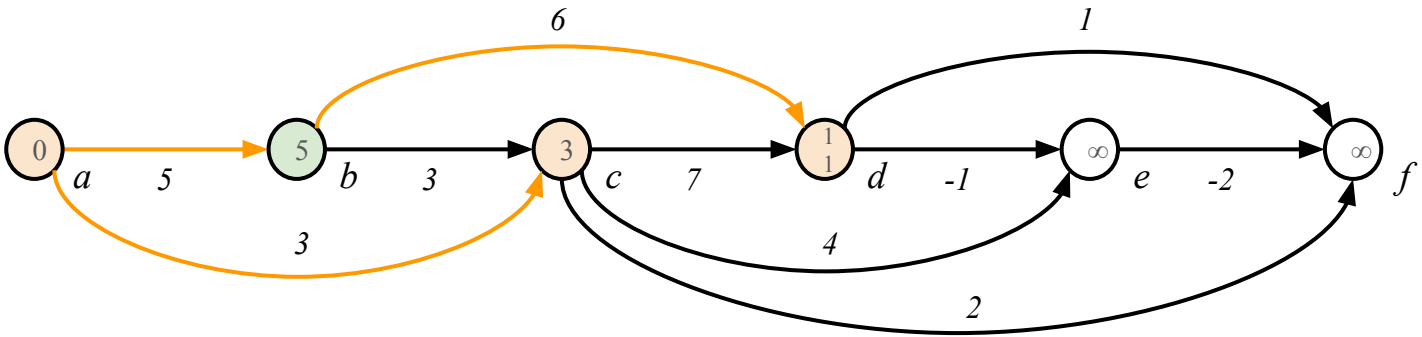


Repaso: DAGs

TOPOLOGICAL-SORT: a→b→c→d→e→f

```
DAG ( G, s ) :
|     TOPOLOGICAL-SORT(G ):
|     INIT(G, s)
|     for u in V (en el orden de TOPOLOGICAL-SORT:
|     |     for v in Adj[u]:
|     |     |     RELAX(u, v)
```

```
RELAX ( u , v, w ) :
|     if v.d > u.d + w(u,v):
|     |     v.d = u.d + w(u,v)
|     |     v.pred = u
```

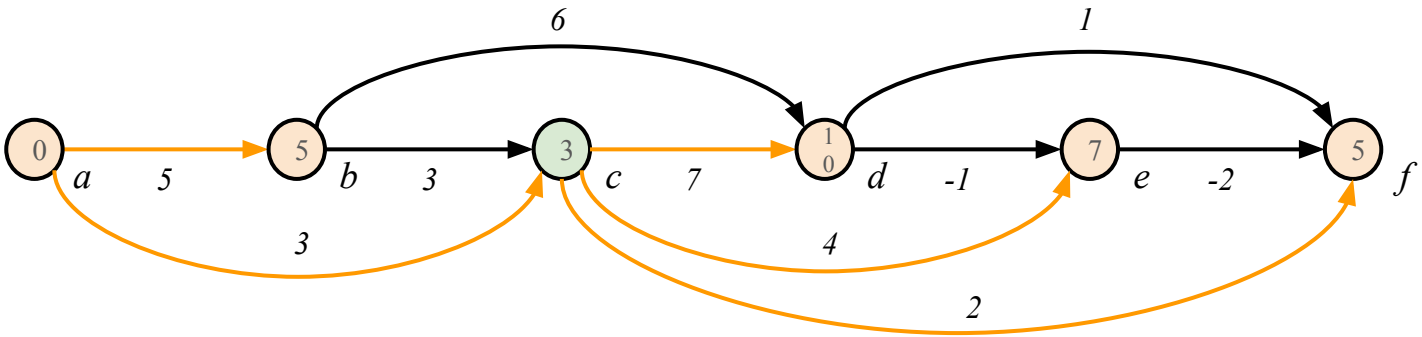


Repaso: DAGs

TOPOLOGICAL-SORT: a→b→c→d→e→f

```
DAG ( G, s ) :
|     TOPOLOGICAL-SORT(G ):
|     INIT(G, s)
|     for u in V (en el orden de TOPOLOGICAL-SORT:
|     |     for v in Adj[u]:
|     |     |     RELAX(u, v)
```

```
RELAX ( u , v, w ) :
|     if v.d > u.d + w(u,v):
|     |     v.d = u.d + w(u,v)
|     |     v.pred = u
```

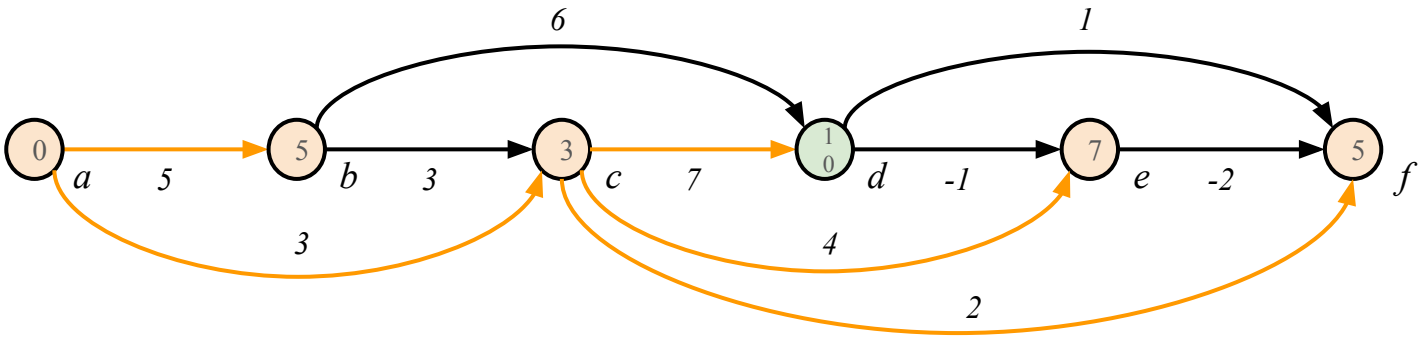


Repaso: DAGs

TOPOLOGICAL-SORT: a→b→c→d→e→f

```
DAG ( G, s ) :
|     TOPOLOGICAL-SORT(G ):
|     INIT(G, s)
|     for u in V (en el orden de TOPOLOGICAL-SORT:
|     |     for v in Adj[u]:
|     |     |     RELAX(u, v)
```

```
RELAX ( u , v, w ) :
|     if v.d > u.d + w(u,v):
|     |     v.d = u.d + w(u,v)
|     |     v.pred = u
```

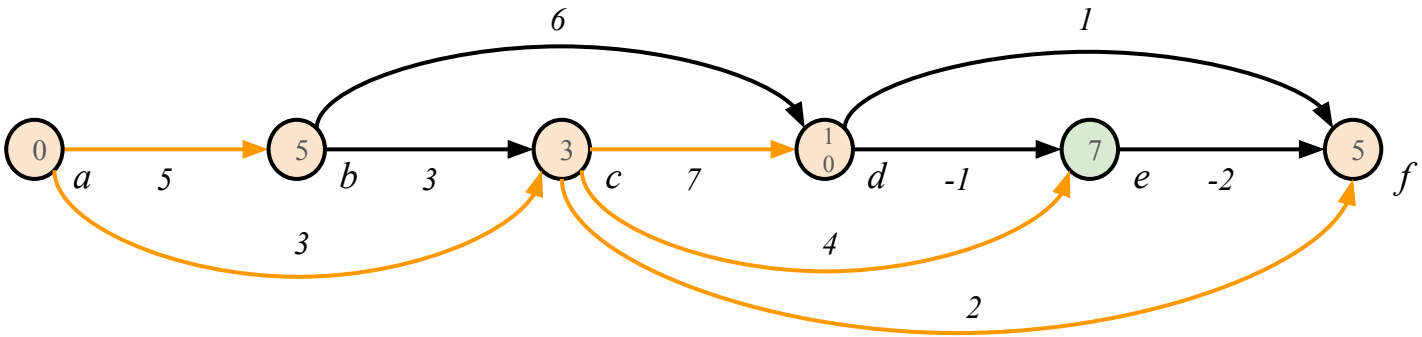


Repaso: DAGs

TOPOLOGICAL-SORT: a→b→c→d→e→f

```
DAG ( G, s ) :
|     TOPOLOGICAL-SORT(G ):
|     INIT(G, s)
|     for u in V (en el orden de TOPOLOGICAL-SORT:
|     |     for v in Adj[u]:
|     |     |     RELAX(u, v)
```

```
RELAX ( u , v, w ) :
|     if v.d > u.d + w(u,v):
|     |     v.d = u.d + w(u,v)
|     |     v.pred = u
```

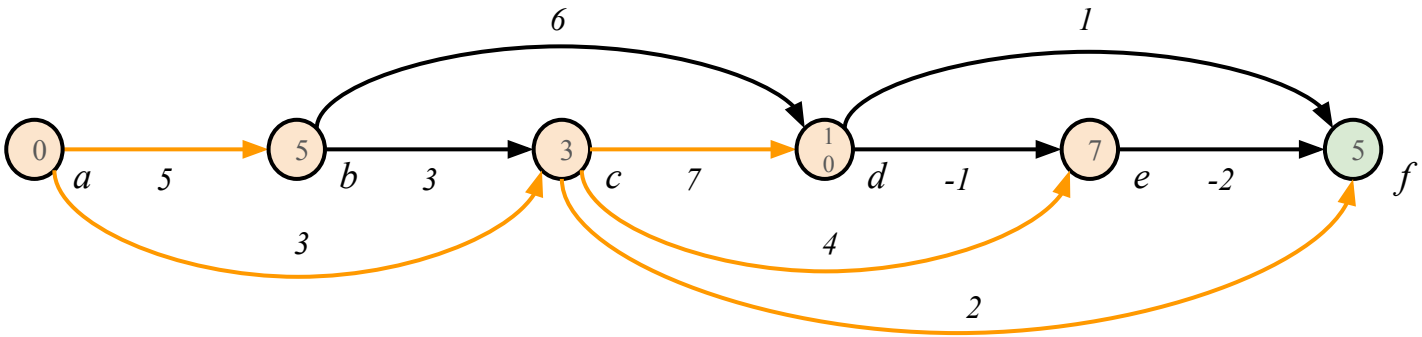


Repaso: DAGs

TOPOLOGICAL-SORT: a→b→c→d→e→f

```
DAG ( G, s ) :
|     TOPOLOGICAL-SORT(G ):
|     INIT(G, s)
|     for u in V (en el orden de TOPOLOGICAL-SORT:
|     |     for v in Adj[u]:
|     |     |     RELAX(u, v)
```

```
RELAX ( u , v, w ) :
|     if v.d > u.d + w(u,v):
|     |     v.d = u.d + w(u,v)
|     |     v.pred = u
```



DAGs

```
RELAX ( u , v, w ) :
|     if v.d > u.d + w(u,v):
|     |     v.d = u.d + w(u,v)
|     |     v.pred = u
```

```
INIT ( G, s ) :
|     for v in V:
|     |     v.d = Inf
|     |     v.pred = None
s.d = 0
```

```
DAG ( G, s ) :
|     TOPOLOGICAL-SORT(G ):
|     INIT(G, s)
|     for u in V (en el orden de TOPOLOGICAL-SORT):
|     |     for v in Adj[u]
|     |     |     RELAX(u, v)
```

DAGs (Complejidad)

```
RELAX ( u , v, w ) :
|     if v.d > u.d + w(u,v):
|     |     v.d = u.d + w(u,v)
|     |     v.pred = u
```

$O(1)$

```
INIT ( G, s ) :
|     for v in V:
|     |     v.d = Inf
|     |     v.pred = None
s.d = 0
```

$O(V)$

```
DAG ( G, s ) :
|     TOPOLOGICAL-SORT(G ):
|     INIT(G, s)
|     for u in V (en el orden de TOPOLOGICAL-SORT):
|     |     for v in Adj[u]
|     |     |     RELAX(u, v)
```

$O(V+E)$

$O(V)$

$O(V+E)$

$O(E)$

DAGs (Correctitud)

```
RELAX ( u , v, w ) :
|     if v.d > u.d + w(u,v):
|     |     v.d = u.d + w(u,v)
|     |     v.pred = u
```

```
DAG ( G, s ) :
|     TOPOLOGICAL-SORT(G, s):
|     INIT(G, s)
|     for u in V (en el orden de TOPOLOGICAL-SORT):
|     |     for v in Adj[u]:
|     |     |     RELAX(u, v)
```

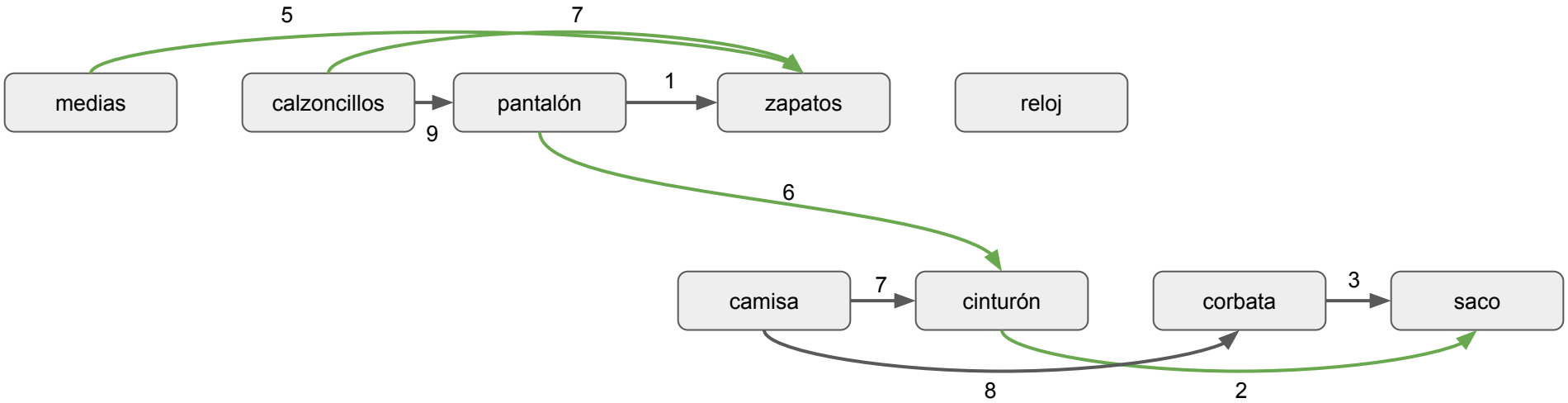
Demo:

TOPOLOGICAL-SORT hace que todos los caminos sean recorridos de izquierda a derecha.

Luego, si se recorre en orden vale la propiedad de **Relajación** de caminos. Finalmente, a partir del **subgrafo de predecesores** se arma el s-ACM.

DAGs

¿Cuánto tiempo me va a llevar la tarea? ¿Cuál es el cuello de botella?



DAGs

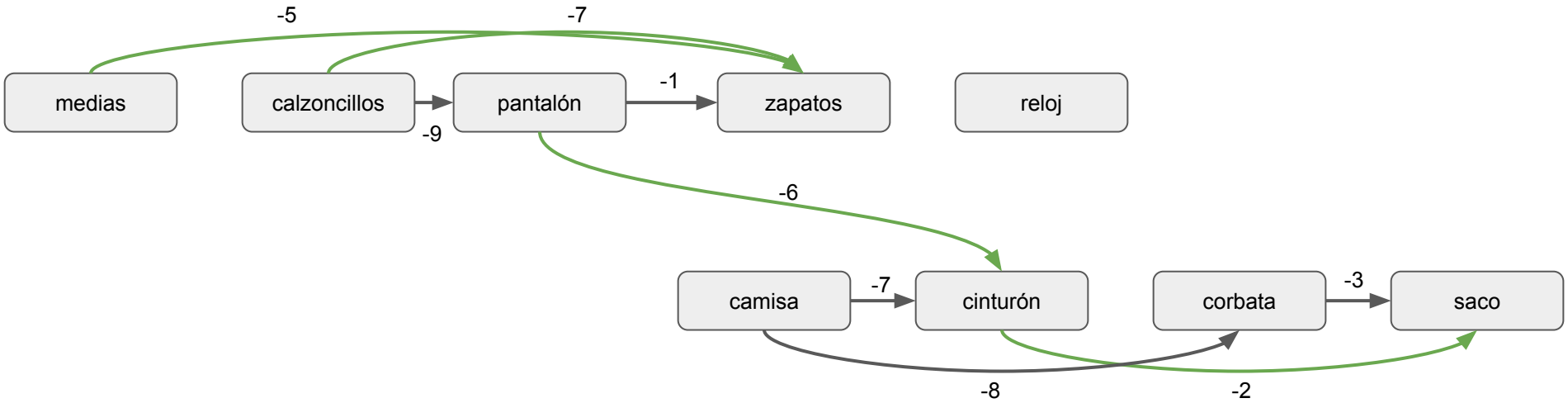
Camino mínimo vs Camino máximo

Dados P_1, P_2 dos caminos en $G=(V, E, w)$, y $w' = -w$ una función de pesos (costos) tq $G'=(V, E, w')$. Entonces,

$$w(P_1) \leq w(P_2) \Leftrightarrow w'(P_1) \geq w'(P_2)$$

DAGs

¿Cuánto tiempo me va a llevar la tarea? ¿Cuál es el cuello de botella?



Dijkstra (Aristas no negativas: $w(u, v) \geq 0$)

Breadth First Search (BFS) iterativo (versión CLRS)

```
BFS ( G , s ) :
|     for cada nodo u ∈ G.V - { s }
|     |     u.color    = n        # w: nuevo, g: frontera descubierta, k: usado
|     |     u.d        = ∞        # distancia
|     |     u.π        = NIL      # parent / predecesor
|     s.color    = g
|     s.d        = 0
|     s.π        = NIL
|     Q          = Ø        # Q: cola: Guardo los que tengo que explorar a continuación: frontera
|     ENQUEUE(Q,s)          # agrega s a la cola Q
|     while Q ≠ 0 :
|     |     u = DEQUEUE( Q )
|     |     for cada v ∈ G.Adj[ u ] :
|     |     |     if v.color == w :     # si no fue visita aún
|     |     |     |     v.color = g     # lo marco
|     |     |     |     v.d = u.d + 1   # actualizo la distancia
|     |     |     |     v.π = u         # u es el predecesor de v
|     |     |     |     ENQUEUE(Q,s)    # guardo v para explorar después
|     |     u.color = k # termino de explorar y lo marco
```

Prim (1957; versión CLRS cap. 21)

```
PRIM ( r , G ) :
|     for u in V:
|     |     u.key      = Inf
|     |     u.parent   = None
|     r.key = 0
|     Q = 0
|     for u in G.V
|     |     INSERT(Q,u) # Cola de prioridad (key)
|     while Q :
|     |     u = EXTRACT-MIN(Q)
|     |     for v in Adj[u] :
|     |     |     if (v in Q) AND (w(u,v) < v.key):
|     |     |     |     v.parent = u
|     |     |     |     v.key = w(u,v)
|     |     |     |     DECREASE-KEY(Q, v, w(u,v))
```

Dijkstra (Aristas no negativas: $w(u, v) \geq 0$)

```
RELAX ( u , v, w ) :
|     if v.d > u.d + w(u,v):
|     |     v.d = u.d + w(u,v)
|     |     v.pred = u
```

Dijkstra (Aristas no negativas: $w(u, v) \geq 0$)

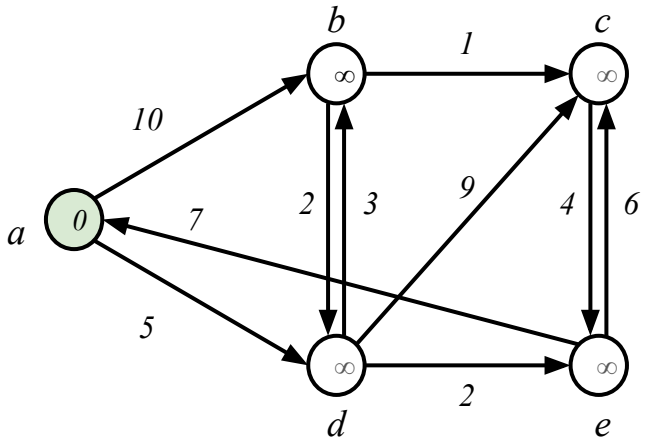
```
DIJKSTRA ( G, s ) :
|     INIT(G, s)
|     S = ∅
|     Q = ∅
|     for u in V:
|     |     INSERT(Q, u)
|     while Q:
|     |     u = EXTRACT-MIN(Q)
|     |     S = S U {u}
|     |     for v in Adj[u]:
|     |     |     RELAX(u, v)
```

```
INIT ( G, s ) :
|     for v in V:
|     |     v.d = Inf
|     |     v.pred = None
s.d = 0
```

```
RELAX ( u , v, w ) :
|     if v.d > u.d + w(u,v):
|     |     v.d = u.d + w(u,v)
|     |     v.pred = u
|     |     DECREASE-KEY(Q, v, v.d)
```

Dijkstra

```
DIJKSTRA ( G, s ) :
|     INIT(G, s)
|     S = ∅
|     Q = ∅
|     for u in V:
|     |     INSERT(Q, u)
|     while Q:
|     |     u = EXTRACT-MIN()
|     |     S = S U {u}
|     |     for v in Adj[u]:
|     |     |     RELAX(u, v)
```




```
INIT ( G, s ) :
|     for v in V:
|     |     v.d = Inf
|     |     v.pred = None
s.d = 0
```

d = {a: 0, b: ∞, c: ∞, d: ∞, e: ∞}

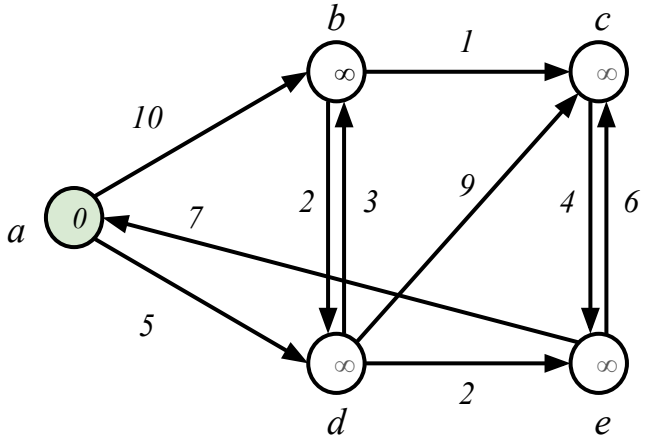
$pred$ = {a: None, b: None, c: None, d: None, e: None}

S = [ ]

Q = [ ]

Dijkstra

```
DIJKSTRA ( G, s ) :
|     INIT(G, s)
|     S = ∅
|     Q = ∅
|     for u in V:
|     |     INSERT(Q, u)
|     while Q:
|     |     u = EXTRACT-MIN()
|     |     S = S U {u}
|     |     for v in Adj[u]:
|     |     |     RELAX(u, v)
```



d = {a: 0, b: ∞, c: ∞, d: ∞, e: ∞}

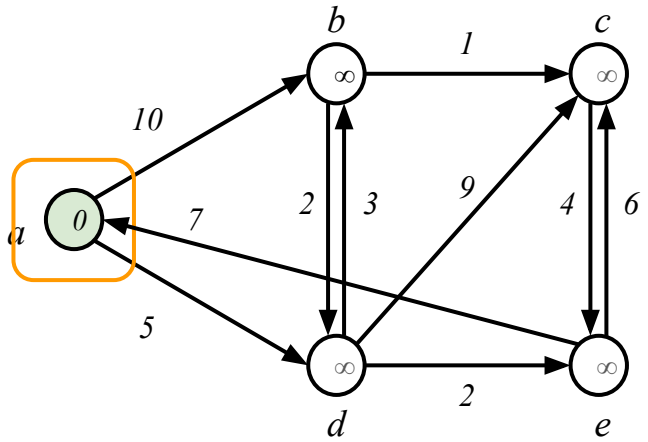
$pred$ = {a: None, b: None, c: None, d: None, e: None}

S = [ ]

Q = [a, b, c, d, e]

Dijkstra

```
DIJKSTRA ( G, s ) :
|     INIT(G, s)
|     S = ∅
|     Q = ∅
|     for u in V:
|     |     INSERT(Q, u)
|     while Q:
|     |     u = EXTRACT-MIN()
|     |     S = S U {u}
|     |     for v in Adj[u]:
|     |     |     RELAX(u, v)
```




```
RELAX ( u , v, w ) :
|     if v.d > u.d + w(u,v):
|     |     v.d = u.d + w(u,v)
|     |     v.pred = u
|     |     DECREASE-KEY(Q, v, v.d)
```

d = {a: 0, b: ∞, c: ∞, d: ∞, e: ∞}

$pred$ = {a: None, b: None, c: None, d: None, e: None}

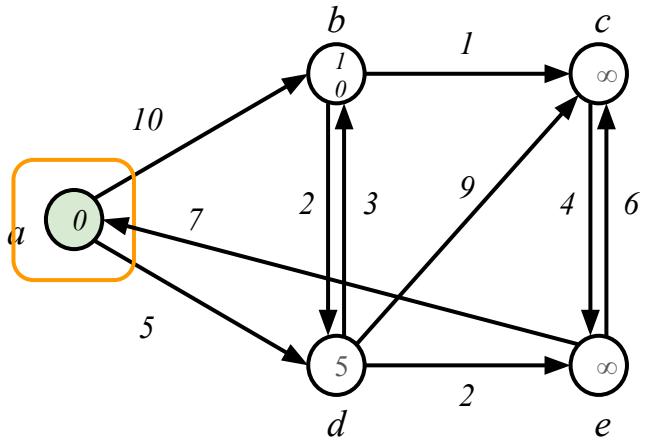
S = [a]

Q = [b, c, d, e]

$Adj[a]$ = [b, d]

Dijkstra

```
DIJKSTRA ( G, s ) :
|     INIT(G, s)
|     S = ∅
|     Q = ∅
|     for u in V:
|     |     INSERT(Q, u)
|     while Q:
|     |     u = EXTRACT-MIN()
|     |     S = S U {u}
|     |     for v in Adj[u]:
|     |     |     RELAX(u, v)
```




```
RELAX ( u , v, w ) :
|     if v.d > u.d + w(u,v):
|     |     v.d = u.d + w(u,v)
|     |     v.pred = u
|     |     DECREASE-KEY(Q, v, v.d)
```

d = {a: 0, b: 10, c: ∞, d: 5, e: ∞}

pred = {a: None, b: a, c: None, d: a, e: None}

S = [a]

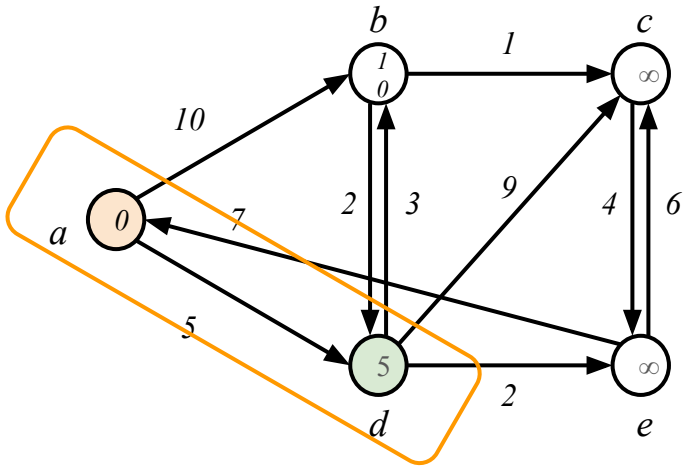
Q = [d, b, c, e]

Adj[a] = [b, d]

Dijkstra

```
DIJKSTRA ( G, s ) :
|     INIT(G, s)
|     S = ∅
|     Q = ∅
|     for u in V:
|     |     INSERT(Q, u)
|     while Q:
|     |     u = EXTRACT-MIN()
|     |     S = S U {u}
|     |     for v in Adj[u]:
|     |     |     RELAX(u, v)
```

```
RELAX ( u , v, w ) :
|     if v.d > u.d + w(u,v):
|     |     v.d = u.d + w(u,v)
|     |     v.pred = u
|     |     DECREASE-KEY(Q, v, v.d)
```



d = {a: 0, b: 10, c: ∞, d: 5, e: ∞}

$pred$ = {a: None, b: a, c: None, d: a, e: None}

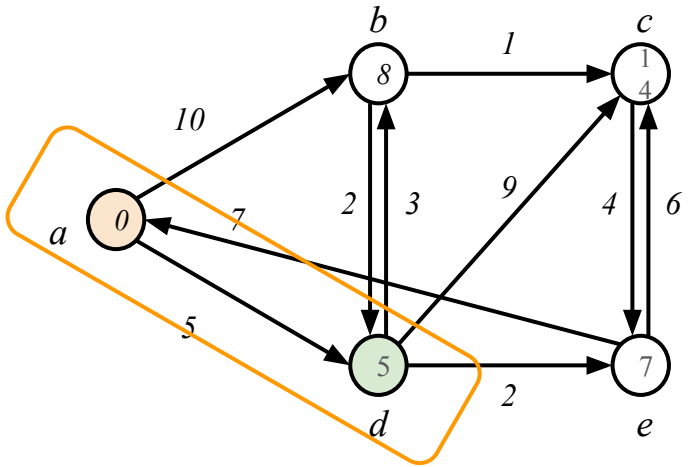
S = [a, d]

Q = [b, c, e]

$Adj[d]$ = [b, c, e]

Dijkstra

```
DIJKSTRA ( G, s ) :
|     INIT(G, s)
|     S = ∅
|     Q = ∅
|     for u in V:
|     |     INSERT(Q, u)
|     while Q:
|     |     u = EXTRACT-MIN()
|     |     S = S U {u}
|     |     for v in Adj[u]:
|     |     |     RELAX(u, v)
```




```
RELAX ( u , v, w ) :
|     if v.d > u.d + w(u,v):
|     |     v.d = u.d + w(u,v)
|     |     v.pred = u
|     |     DECREASE-KEY(Q, v, v.d)
```

d = {a: 0, b: 8, c: 14, d: 5, e: 7}

pred = {a: None, b: d, c: d, d: a, e: d}

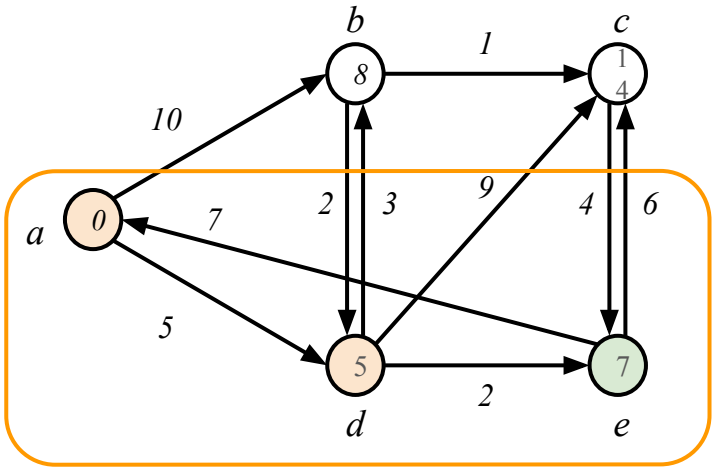
S = [a, d]

Q = [e, b, c]

Adj[d] = [b, c, e]

Dijkstra

```
DIJKSTRA ( G, s ) :
|     INIT(G, s)
|     S = ∅
|     Q = ∅
|     for u in V:
|     |     INSERT(Q, u)
|     while Q:
|     |     u = EXTRACT-MIN()
|     |     S = S U {u}
|     |     for v in Adj[u]:
|     |     |     RELAX(u, v)
```




```
RELAX ( u , v, w ) :
|     if v.d > u.d + w(u,v):
|     |     v.d = u.d + w(u,v)
|     |     v.pred = u
|     |     DECREASE-KEY(Q, v, v.d)
```

d = {a: 0, b: 8, c: 14, d: 5, e: 7}

pred = {a: None, b: d, c: d, d: a, e: d}

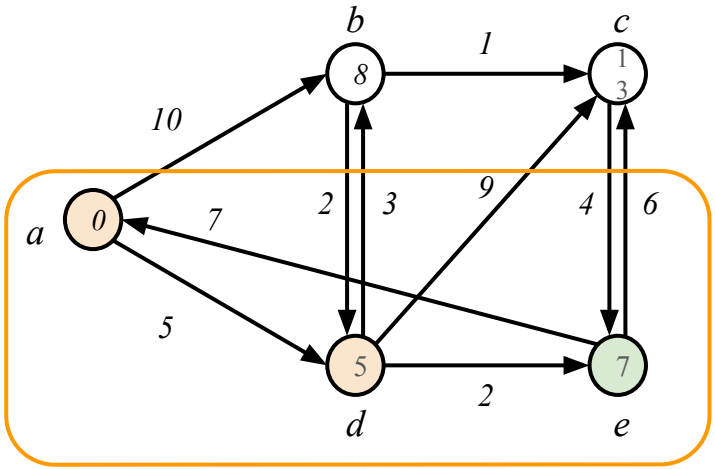
S = [a, d, e]

Q = [b, c]

Adj[e] = [a, c]

Dijkstra

```
DIJKSTRA ( G, s ) :
|     INIT(G, s)
|     S = ∅
|     Q = ∅
|     for u in V:
|     |     INSERT(Q, u)
|     while Q:
|     |     u = EXTRACT-MIN()
|     |     S = S U {u}
|     |     for v in Adj[u]:
|     |     |     RELAX(u, v)
```




```
RELAX ( u , v, w ) :
|     if v.d > u.d + w(u,v):
|     |     v.d = u.d + w(u,v)
|     |     v.pred = u
|     |     DECREASE-KEY(Q, v, v.d)
```

d = {a: 0, b: 8, c: 13, d: 5, e: 7}

pred = {a: None, b: d, c: e, d: a, e: d}

S = [a, d, e]

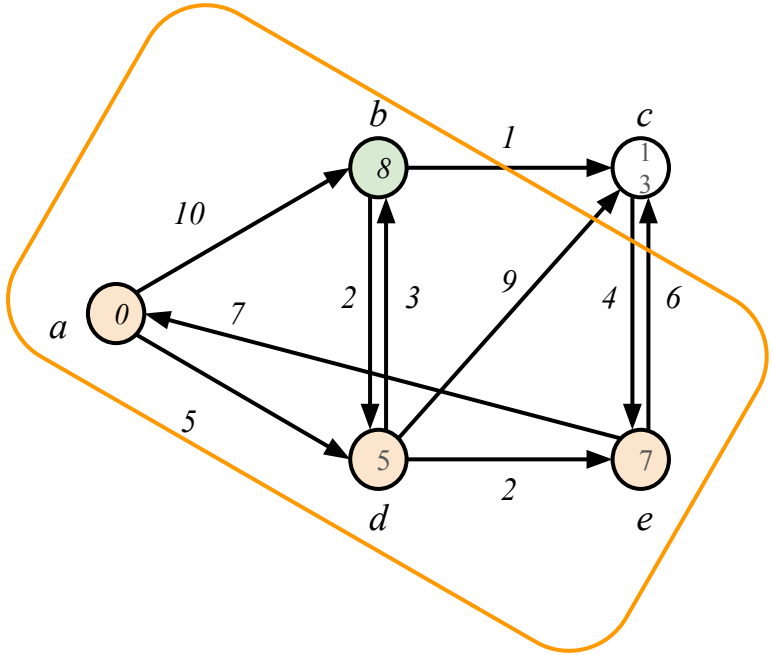
Q = [b, c]

Adj[e] = [a, c]

Dijkstra

```
DIJKSTRA ( G, s ) :
|     INIT(G, s)
|     S = ∅
|     Q = ∅
|     for u in V:
|     |     INSERT(Q, u)
|     while Q:
|     |     u = EXTRACT-MIN()
|     |     S = S U {u}
|     |     for v in Adj[u]:
|     |     |     RELAX(u, v)
```

```
RELAX ( u , v, w ) :
|     if v.d > u.d + w(u,v):
|     |     v.d = u.d + w(u,v)
|     |     v.pred = u
|     |     DECREASE-KEY(Q, v, v.d)
```



d = {a: 0, b: 8, c: 13, d: 5, e: 7}

pred = {a: None, b: d, c: e, d: a, e: d}

S = [a, d, e, b]

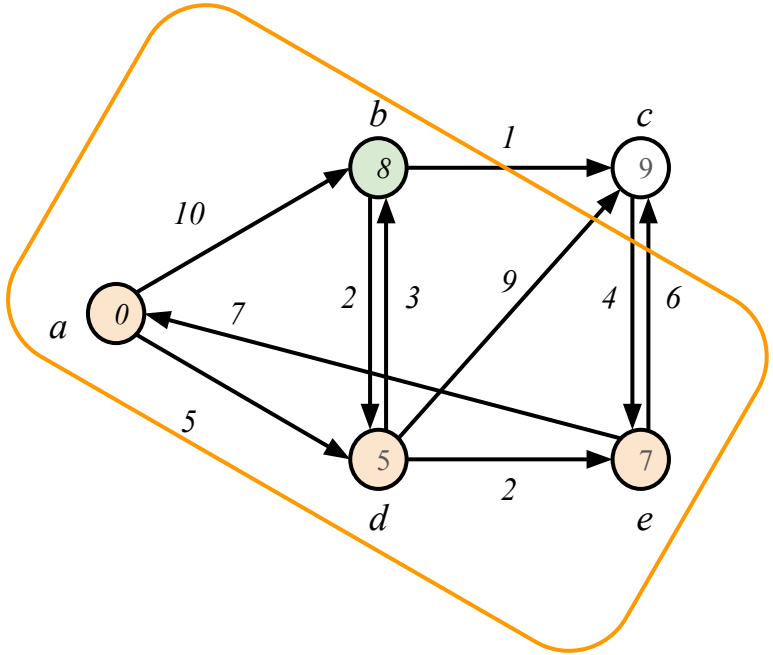
Q = [c]

Adj[b] = [c, d]

Dijkstra

```
DIJKSTRA ( G, s ) :
|     INIT(G, s)
|     S = ∅
|     Q = ∅
|     for u in V:
|     |     INSERT(Q, u)
|     while Q:
|     |     u = EXTRACT-MIN()
|     |     S = S U {u}
|     |     for v in Adj[u]:
|     |     |     RELAX(u, v)
```

```
RELAX ( u , v, w ) :
|     if v.d > u.d + w(u,v):
|     |     v.d = u.d + w(u,v)
|     |     v.pred = u
|     |     DECREASE-KEY(Q, v, v.d)
```



d = {a: 0, b: 8, c: 9, d: 5, e: 7}

pred = {a: None, b: d, c: b, d: a, e: d}

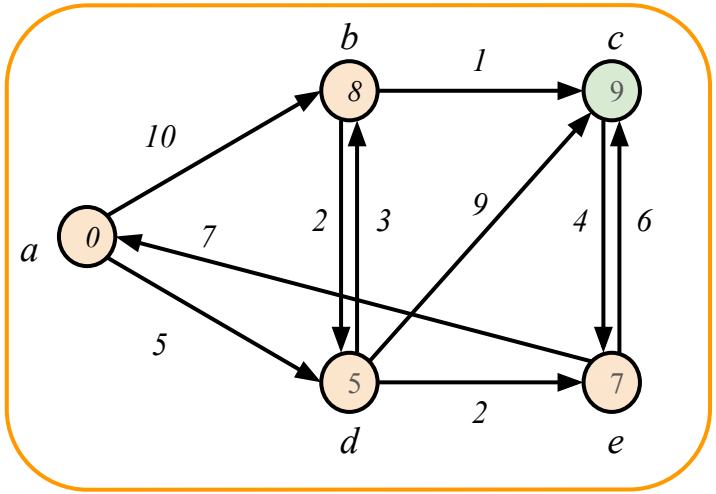
S = [a, d, e, b]

Q = [c]

Adj[b] = [c, d]

Dijkstra

```
DIJKSTRA ( G, s ) :
|     INIT(G, s)
|     S = ∅
|     Q = ∅
|     for u in V:
|     |     INSERT(Q, u)
|     while Q:
|     |     u = EXTRACT-MIN()
|     |     S = S U {u}
|     |     for v in Adj[u]:
|     |     |     RELAX(u, v)
```




```
RELAX ( u , v, w ) :
|     if v.d > u.d + w(u,v):
|     |     v.d = u.d + w(u,v)
|     |     v.pred = u
|     |     DECREASE-KEY(Q, v, v.d)
```

d = *{a: 0, b: 8, c: 9, d: 5, e: 7}*

pred = *{a: None, b: d, c: b, d: a, e: d}*

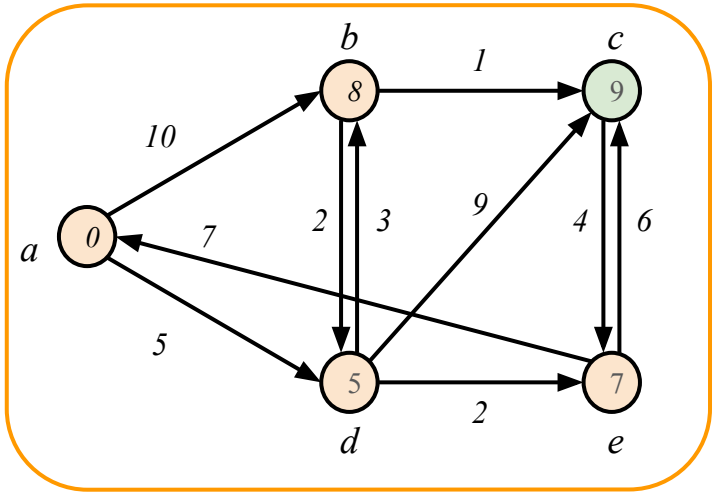
S = *[a, d, e, b, c]*

Q = *[]*

Adj[c] = *[e]*

Dijkstra

```
DIJKSTRA ( G, s ) :
|     INIT(G, s)
|     S = ∅
|     Q = ∅
|     for u in V:
|     |     INSERT(Q, u)
|     while Q:
|     |     u = EXTRACT-MIN()
|     |     S = S U {u}
|     |     for v in Adj[u]:
|     |     |     RELAX(u, v)
```




```
RELAX ( u , v, w ) :
|     if v.d > u.d + w(u,v):
|     |     v.d = u.d + w(u,v)
|     |     v.pred = u
|     |     DECREASE-KEY(Q, v, v.d)
```

d = *{a: 0, b: 8, c: 9, d: 5, e: 7}*

pred = *{a: None, b: d, c: b, d: a, e: d}*

S = *[a, d, e, b, c]*

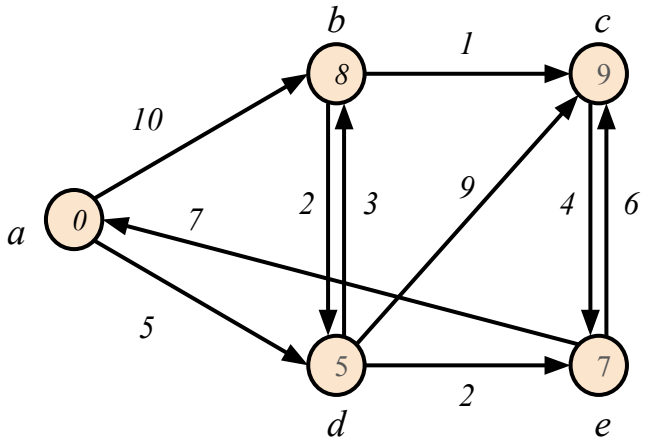
Q = *[]*

Adj[c] = *[e]*

Dijkstra

```
DIJKSTRA ( G, s ) :
|     INIT(G, s)
|     S = ∅
|     Q = ∅
|     for u in V:
|     |     INSERT(Q, u)
|     while Q:
|     |     u = EXTRACT-MIN()
|     |     S = S U {u}
|     |     for v in Adj[u]:
|     |     |     RELAX(u, v)
```

```
RELAX ( u , v, w ) :
|     if v.d > u.d + w(u,v):
|     |     v.d = u.d + w(u,v)
|     |     v.pred = u
|     |     DECREASE-KEY(Q, v, v.d)
```



d = {a: 0, b: 8, c: 9, d: 5, e: 7}

$pred$ = {a: None, b: d, c: b, d: a, e: d}

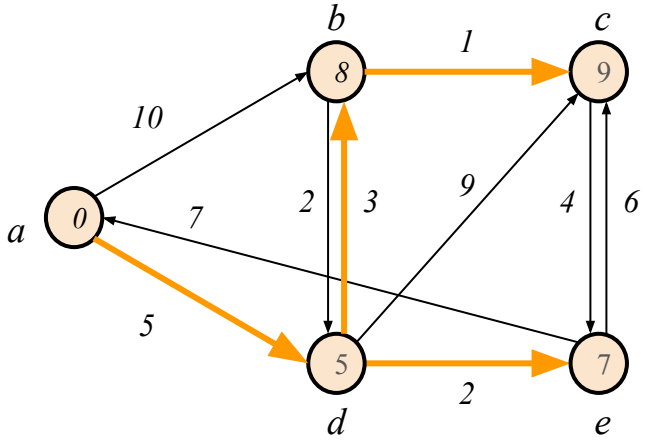
S = [a, d, e, b, c]

Q = []

Dijkstra

```
DIJKSTRA ( G, s ) :
|     INIT(G, s)
|     S = ∅
|     Q = ∅
|     for u in V:
|     |     INSERT(Q, u)
|     while Q:
|     |     u = EXTRACT-MIN()
|     |     S = S U {u}
|     |     for v in Adj[u]:
|     |     |     RELAX(u, v)
```

```
RELAX ( u , v, w ) :
|     if v.d > u.d + w(u,v):
|     |     v.d = u.d + w(u,v)
|     |     v.pred = u
|     |     DECREASE-KEY(Q, v, v.d)
```



d = {a: 0, b: 8, c: 9, d: 5, e: 7}

$pred$ = {a: None, b: d, c: b, d: a, e: d}

S = [a, d, e, b, c]

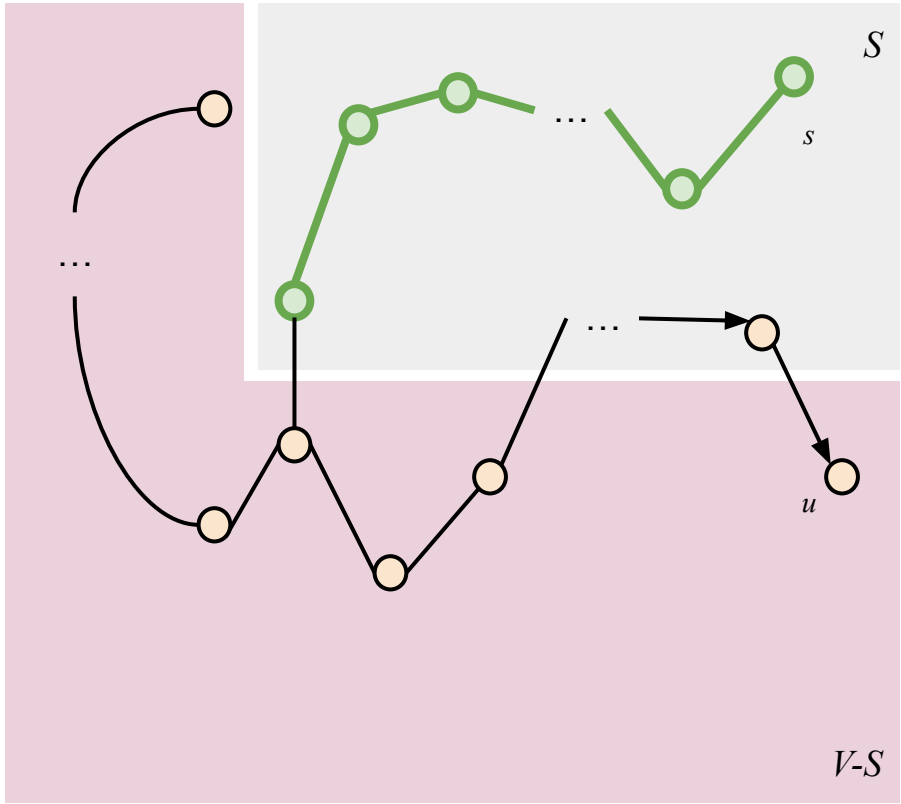
Q = []

Dijkstra (Correctitud)

Demo (inducción en vértices):

Caso base: $S = [s]$, $d(s,s) = \delta(s,s) = 0$

Hipótesis inductiva: $\forall v \in S,\ d(s,v) = \delta(s,v)$



Dijkstra (Correctitud)

Demo (inducción en vértices):

Caso base: $S = [s]$, $d(s,s) = \delta(s,s) = 0$

Hipótesis inductiva: $\forall v \in S,\ d(s,v) = \delta(s,v)$

Sea $u \in V\text{-}S$, supongo que existe $y \in V\text{-}S$ el siguiente fuera de S, $x \in S$ su predecesor.

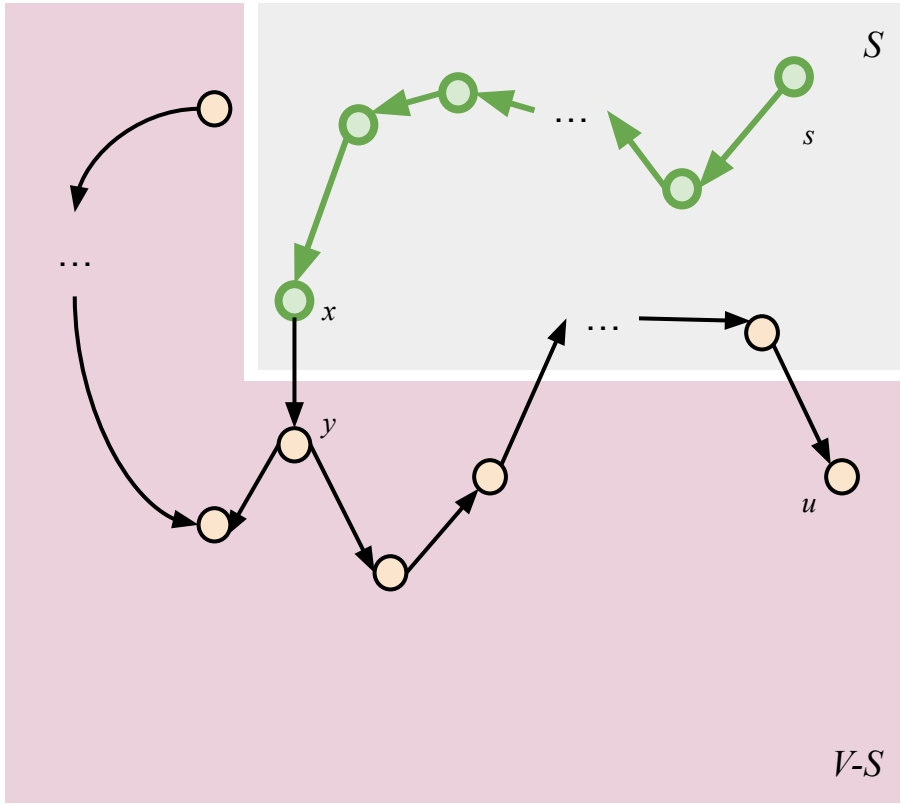
Como $w \geq 0 \Rightarrow \delta(s, y) \leq \delta(s, u)$

Supongamos que EXTRACT-MIN devuelve u, significa que $u.d \leq y.d$ en ese paso.

Y $\delta(s, u) \leq u.d$ por propiedad del **límite superior**.

Por otro lado, como $x \in S \Rightarrow x.d = \delta(s, x)$ y al agregar x a S, (x, y) se relaja $\Rightarrow y.d = \delta(s, y)$

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



Dijkstra (Correctitud)

Demo (inducción en vértices):

Caso base: $S = [s]$, $d(s,s) = \delta(s,s) = 0$

Hipótesis inductiva: $\forall v \in S,\ d(s,v) = \delta(s,v)$

Sea $u \in V\text{-}S$, supongo que existe $y \in V\text{-}S$ el siguiente fuera de S, $x \in S$ su predecesor.

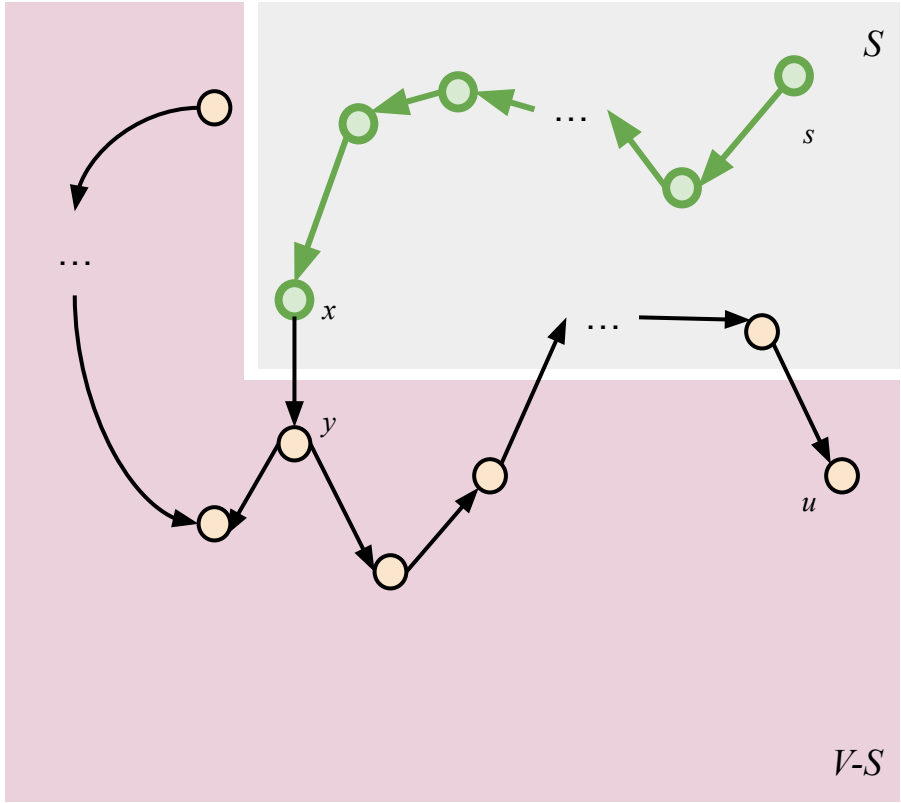
Como $w \geq 0 \Rightarrow \delta(s, y) \leq \delta(s, u)$

Supongamos que EXTRACT-MIN devuelve u, significa que $u.d \leq y.d$ en ese paso.

Y $\delta(s, u) \leq u.d$ por propiedad del **límite superior**.

Por otro lado, como $x \in S \Rightarrow x.d = \delta(s, x)$ y al agregar x a S, (x, y) se relaja $\Rightarrow y.d = \delta(s, y)$

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



Dijkstra (Complejidad)

```
DIJKSTRA ( G, s ) :
|     INIT(G, s)
|     S = ∅
|     Q = ∅
|     for u in V:
|     |     INSERT(Q, u)
|     while Q:
|     |     u = EXTRACT-MIN()
|     |     S = S U {u}
|     |     for v in Adj[u]:
|     |     |     RELAX(u, v)
```

$O(1)$ $O(V)$

$O(V)$

$O(V + V + V^2 + E) = O(V^2 + E) = O(V^2)$

```
INIT ( G, s ) :
|     for v in V:
|     |     v.d = Inf
|     |     v.pred = None
s.d = 0
```

$O(V)$

```
RELAX ( u , v, w ) :
|     if v.d > u.d + w(u,v):
|     |     v.d = u.d + w(u,v)
|     |     v.pred = u
|     |     DECREASE-KEY(Q, v, v.d)
```

$O(1)$

Dijkstra (Complejidad)

```
DIJKSTRA ( G, s ) :
|     INIT(G, s)
|     S = ∅
|     Q = ∅
|     for u in V:
|     |     INSERT(Q, u)
|     while Q:
|     |     u = EXTRACT-MIN()
|     |     S = S U {u}
|     |     for v in Adj[u]:
|     |     |     RELAX(u, v)
```

O(1) *O(V)* *O(log(V))*

```
INIT ( G, s ) :
|     for v in V:
|     |     v.d = Inf
|     |     v.pred = None
s.d = 0
```

O(V)

```
RELAX ( u , v, w ) :
|     if v.d > u.d + w(u,v):
|     |     v.d = u.d + w(u,v)
|     |     v.pred = u
|     |     DECREASE-KEY(Q, v, v.d)
```

O(log(V))

$O(V + V + V^2 + E) = O(V^2 + E) = O(V^2)$

Con min-heap... $O(V + V + V*\log(V) + E*\log(V)) = O(E*\log(V)) \sim O(V*\log(V))$ si es ralo

Dijkstra (Complejidad)

```
DIJKSTRA ( G, s ) :
|     INIT(G, s)
|     S = ∅
|     Q = ∅
|     for u in V:
|     |     INSERT(Q, u)        O(1)      O(V)
|     while Q:
|     |     u = EXTRACT-MIN()   O(log(V))
|     |     S = S U {u}
|     |     for v in Adj[u]:
|     |     |     RELAX(u, v)
```

```
INIT ( G, s ) :                                   O(V)
|     for v in V:
|     |     v.d = Inf
|     |     v.pred = None
s.d = 0
```

```
RELAX ( u , v, w ) :
|     if v.d > u.d + w(u,v):
|     |     v.d = u.d + w(u,v)
|     |     v.pred = u
|     |     DECREASE-KEY(Q, v, v.d)        O(1)
```

$O(V + V + V^2 + E) = O(V^2 + E) = O(V^2)$

Con min-heap… $O(V + V + V*\log(V) + E*\log(V)) = O(E*\log(V)) \sim O(V*\log(V))$ *si es ralo*

Con fibonacci-heap… $O(V + V + V*\log(V) + E) = O(V*\log(V) + E) \sim O(V*\log(V))$ *si es ralo*

((INICIO PARÉNTESIS)) Algunas mejoras para uno a uno

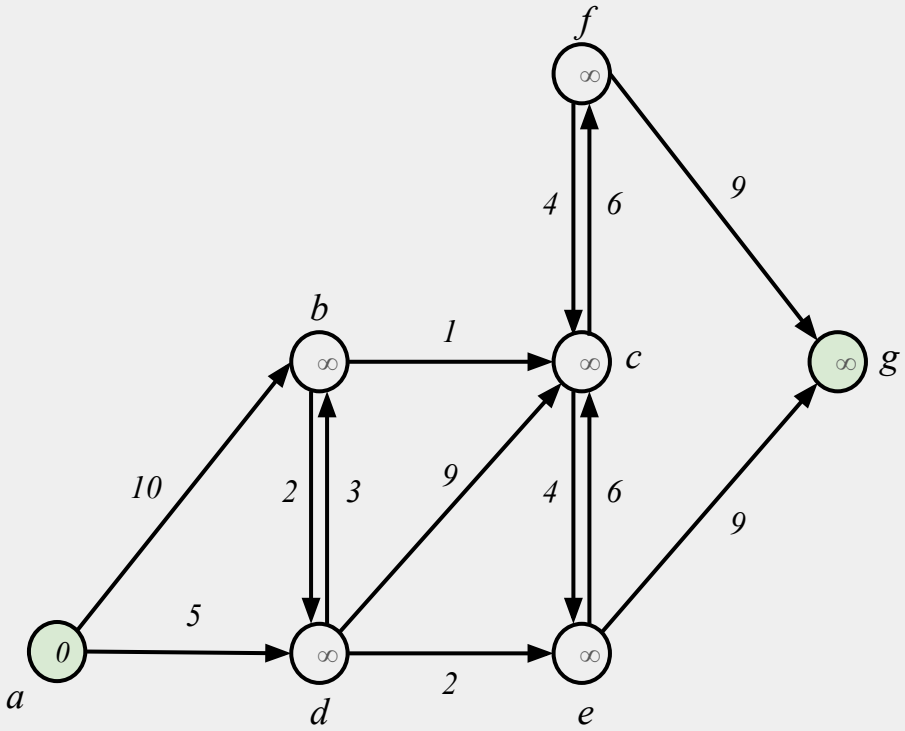
(No cambian el peor caso pero mejoran mucho el promedio)

IDEA GRAL. No explorar todo el grafo.

- Cortar cuando encuentro el target.
- Bi-directional Dijkstra
- A*

A*

~ Uno a uno



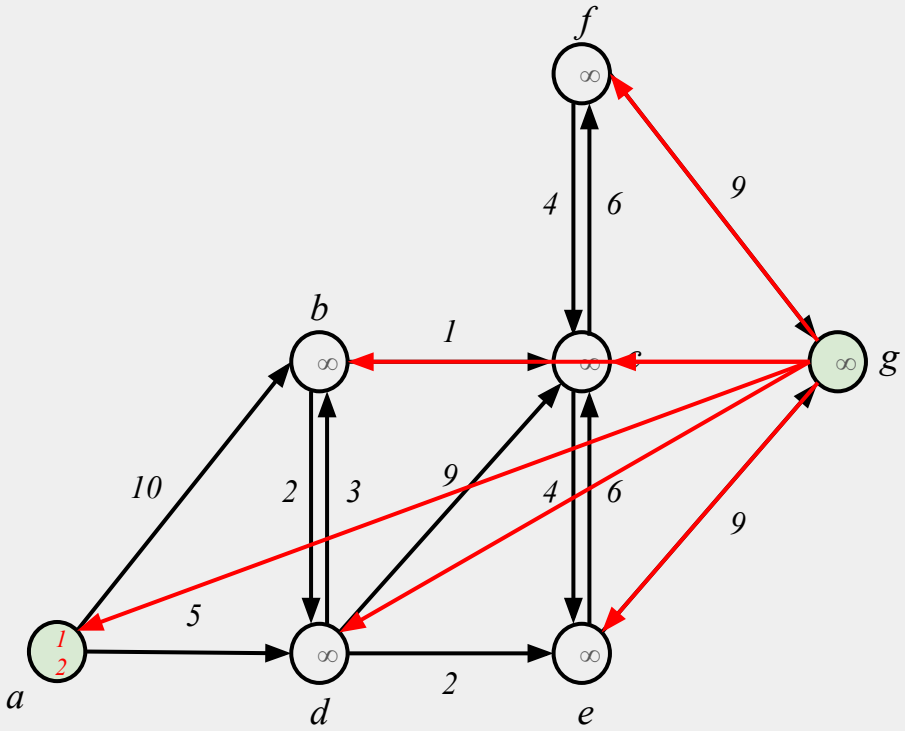
Hart, P. E., Nilsson, N. J., & Raphael, B. (1968). A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics*, *4*(2), 100-107.

A*

Inicializo los valores con una distancia estimada al objetivo (heurística). Es importante que subestime (admisible), y que sea consistente (que respete las distancias).

$h(g,x) \leq d(g,x)$ (admisible)

$|h(g,x) - h(g,y)| \leq d(x,y)$ (consistente)

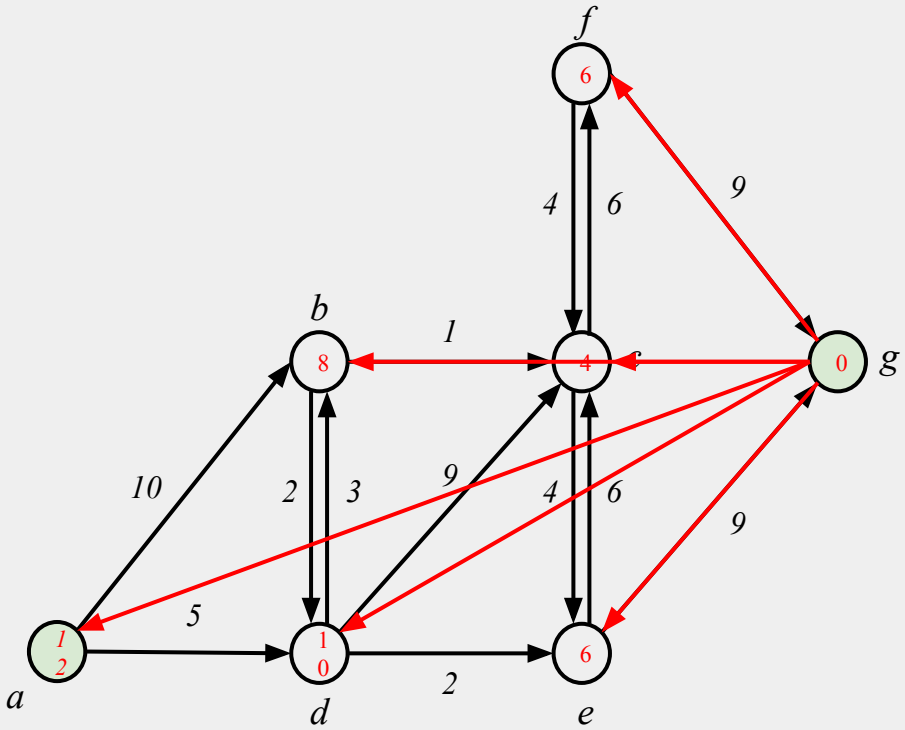


A*

Inicializo los valores con una distancia estimada al objetivo (heurística). Es importante que subestime (admisible), y que sea consistente (que respete las distancias).

$h(g,x) \leq d(g,x)$ (admisible)

$|h(g,x) - h(g,y)| \leq d(x,y)$ (consistente)

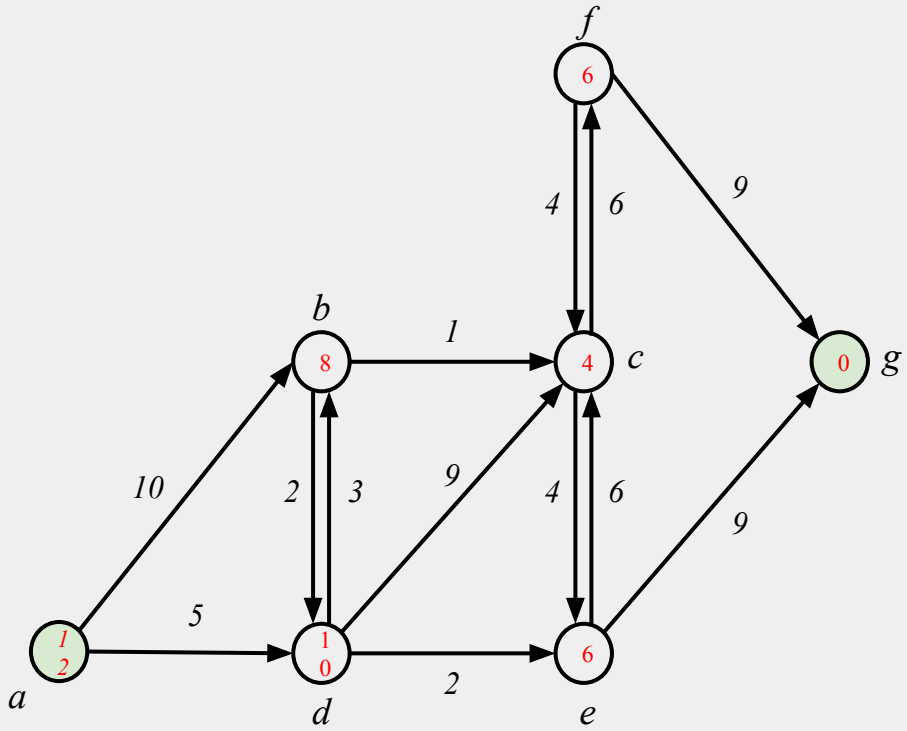


A*

Inicializo los valores con una distancia estimada al objetivo (heurística). Es importante que subestime (admisible), y que sea consistente (que respete las distancias).

$h(g,x) \leq d(g,x)$ (admisible)

$|h(g,x) - h(g,y)| \leq d(x,y)$ (consistente)

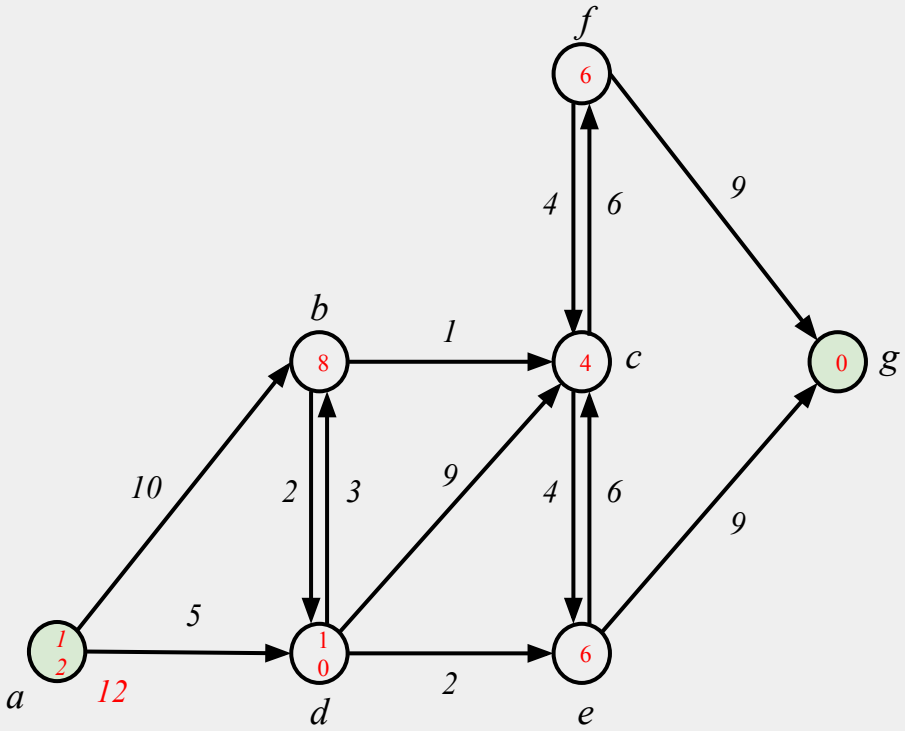


A*

Inicializo los valores con una distancia estimada al objetivo (heurística). Es importante que subestime (ver más adelante).

A* score = heurística (h) + costo

a = 5

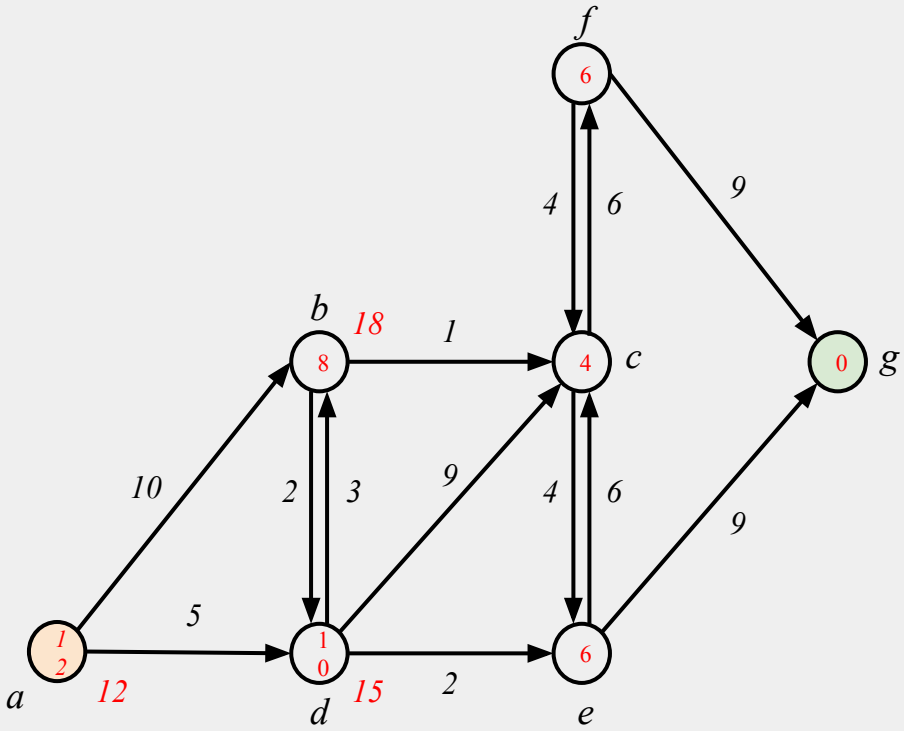


A*

Inicializo los valores con una distancia estimada al objetivo (heurística). Es importante que subestime (ver más adelante).

A* score = heurística + costo

a = 12, d = 15

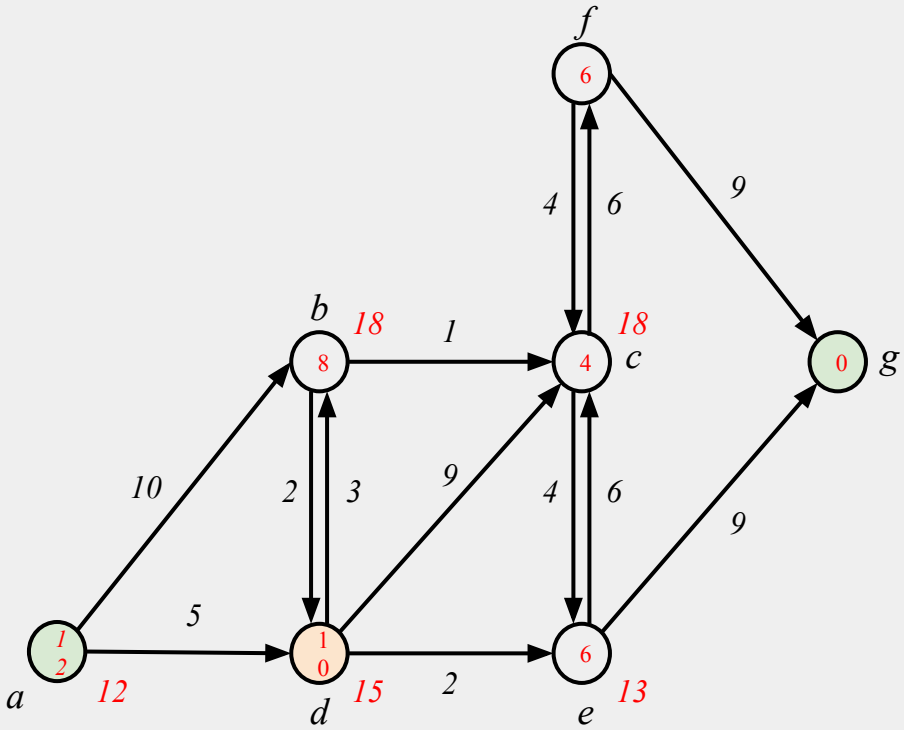


A*

Inicializo los valores con una distancia estimada al objetivo (heurística). Es importante que subestime (ver más adelante).

A* score = heurística + costo

a = 12, d = 15, e = 13

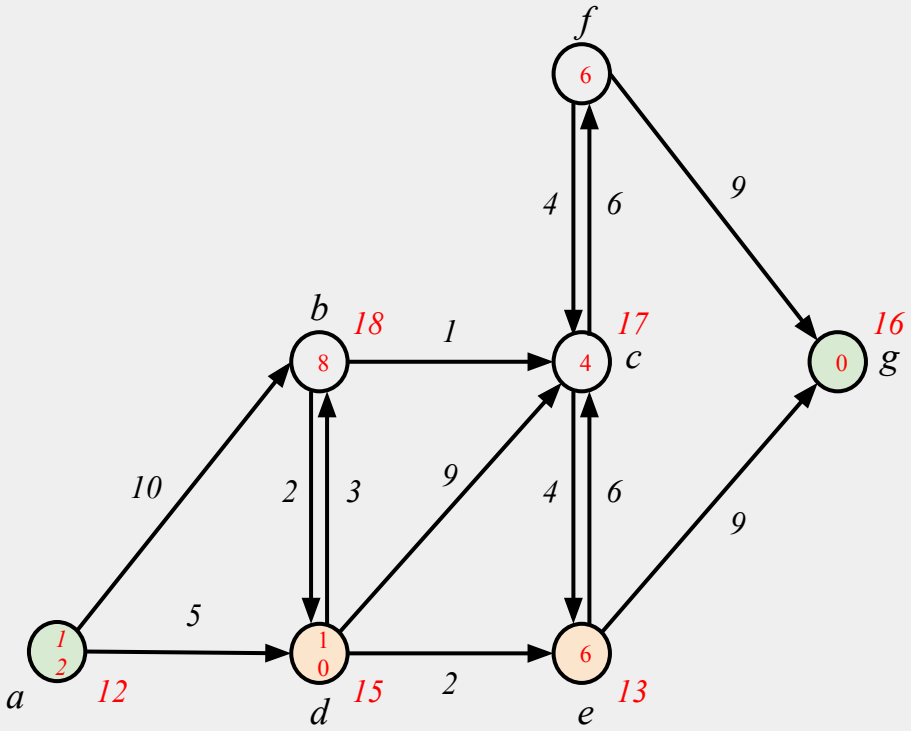


A*

Inicializo los valores con una distancia estimada al objetivo (heurística). Es importante que subestime (ver más adelante).

A* score = heurística + costo

a = 12, d = 15, e = 13, g = 16

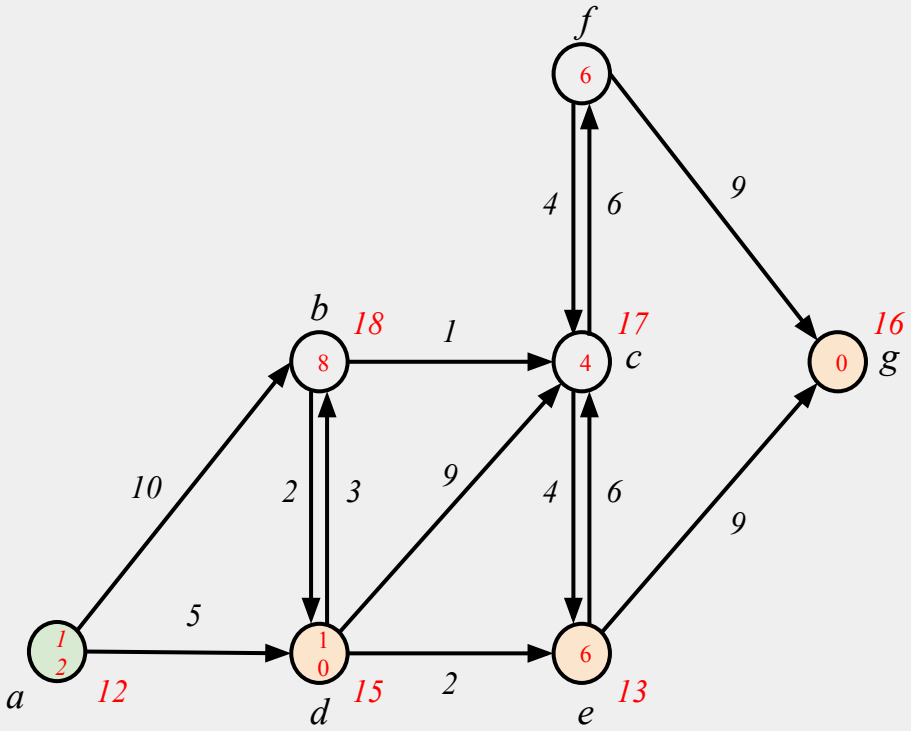


A*

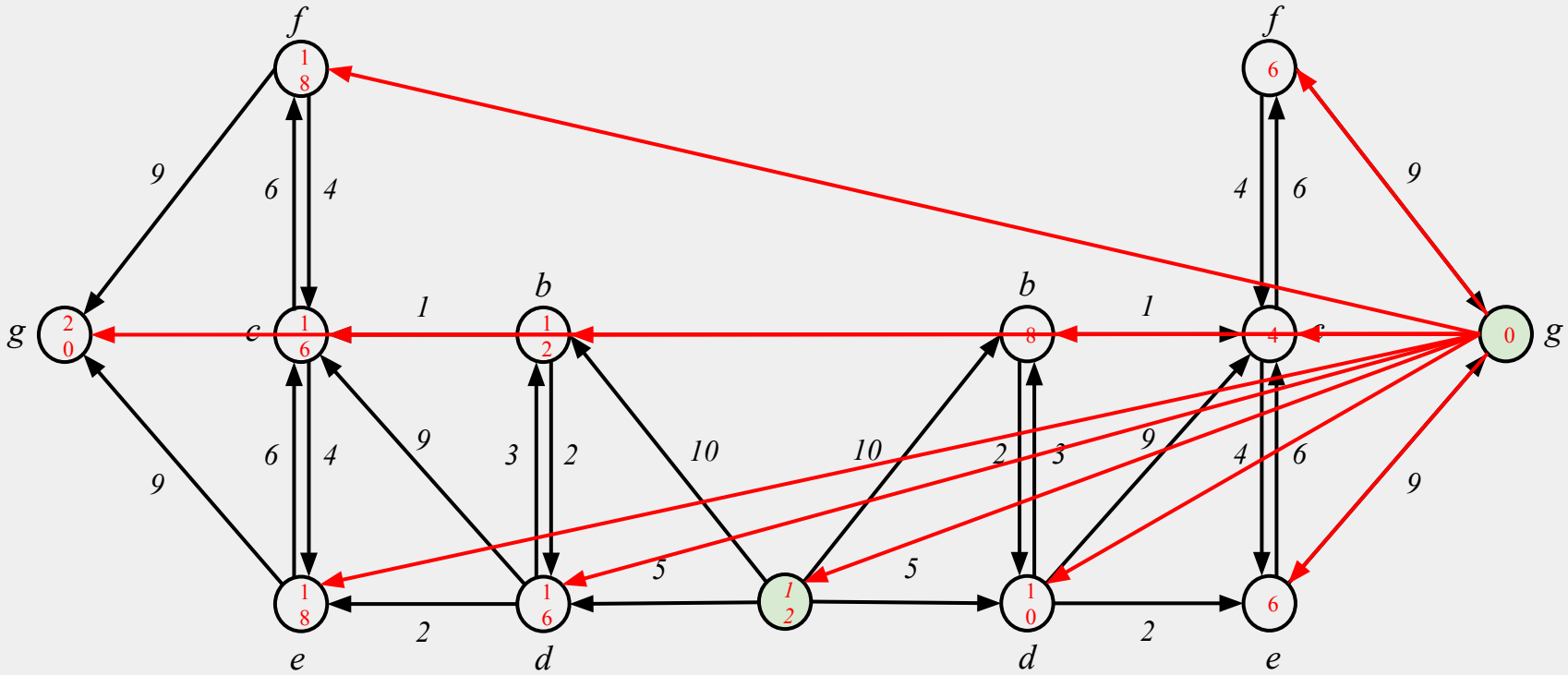
Inicializo los valores con una distancia estimada al objetivo (heurística). Es importante que subestime (ver más adelante).

A* score = heurística + costo

a = 12, d = 15, e = 13, g = 16



A*


f
18
9
6
4
g
20
c
16
1
b
12
1
b
8
1
f
6
4
6
9
4
c
6
0
g
9
6
4
9
3
2
10
10
2
3
9
4
6
9
5
5
18
16
12
10
6
2
2
e
d
d
e

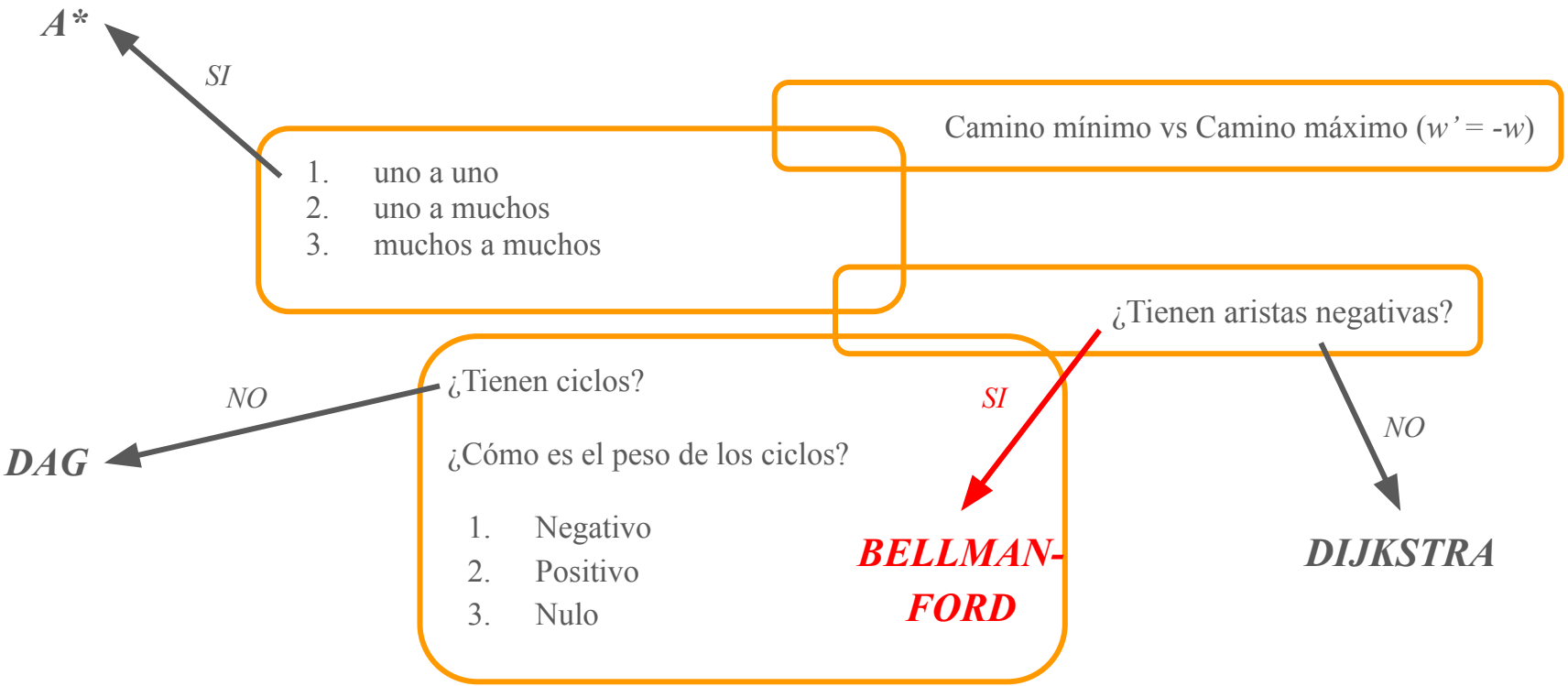
((FIN PARÉNTESIS))

A*

```
A-estrella ( G, s ) :
|     INIT(G, s)
|     for v in V:
|     |     v.h = heuristica(g,h)
|     |     v.a = v.d + v.h
|     S = ∅
|     Q = [s]
|
|     while Q:
|     |     u = EXTRACT-MIN(Q)
|     |     S = S U {u}
|     |     if u == g:
|     |     |     return S # O le pido todo el grafo
|     |     |
|     |     for v in Adj[u]:
|     |     |     if v not in S:
|     |     |     |     if v not in Q:
|     |     |     |     |     Q = Q U {v}
|     |     |     |     |     v.d = u.d + w(u,v)
|     |     |     |     |     v.pred = u
|     |     |     |     |     v.a = v.d + v.h
|     |     |     |     |     DECREASE-KEY(Q, v, v.a)
```

```
INIT ( G, s ) :
|     for v in V:
|     |     v.d = Inf
|     |     v.pred = None
s.d = 0
```

Repaso: Topología de problemas de camino mínimo



Bellman-Ford

```
BELLMAN-FORD ( G, s ) :
|     INIT(G, s)
|     for i = 1 to n-1:
|     |     for (u,v) in E:
|     |     |     RELAX(u, v)
|     for (u,v) in E:
|     |     if v.d < u.d + w(u,v):
|     |     |     return False
|     return True, v
```

```
INIT ( G, s ) :
|     for v in V:
|     |     v.d = Inf
|     |     v.pred = None
s.d = 0
```

```
RELAX ( u , v, w ) :
|     if v.d > u.d + w(u,v):
|     |     v.d = u.d + w(u,v)
|     |     v.pred = u
```

Bellman-Ford

```
BELLMAN-FORD ( G, s ) :
|     INIT(G, s)
|
|     for i = 1 to n-1:
|     |     for (u,v) in E:
|     |     |     RELAX(u, v)
|
|     for (u,v) in E:
|     |     if v.d < u.d + w(u,v):
|     |     |     return False
|
|     return True, v
```

¡Relajo TODAS las aristas a la vez!

¡n - 1 veces!

```
RELAX ( u , v, w ) :
|     if v.d > u.d + w(u,v):
|     |     v.d = u.d + w(u,v)
|     |     v.pred = u
```

Bellman-Ford

```
BELLMAN-FORD ( G, s ) :
|     INIT(G, s)
|
|     for i = 1 to n-1:
|     |     for (u,v) in E:
|     |     |     RELAX(u, v)
|
|     for (u,v) in E:
|     |     if v.d < u.d + w(u,v):
|     |     |     return False
|
|     return True, v
```

```
RELAX ( u , v, w ) :
|     if v.d > u.d + w(u,v):
|     |     v.d = u.d + w(u,v)
|     |     v.pred = u
```

Chequeo que no haya ciclos negativos:

Repito una vez más,

si todavía es posible seguir achicando distancias ⇒

¡Hay ciclos negativos!

Nota: No puedo decir a quienes afectan, sólo que son alcanzables desde “s”)

Bellman-Ford

```
BELLMAN-FORD ( G, s ) :
|     INIT(G, s)
|
|     for i = 1 to n-1:
|     |     for (u,v) in E:
|     |     |     RELAX(u, v)
|
|     for (u,v) in E:
|     |     if v.d < u.d + w(u,v):
|     |     |     return False
|
|     return True, v
```

```
RELAX ( u , v, w ) :
|     if v.d > u.d + w(u,v):
|     |     v.d = u.d + w(u,v)
|     |     v.pred = u
```

Si no corta ⇒ ¡NO hay ciclos negativos!

Devuelve lo mismo que tenía.

Bellman-Ford

```
BELLMAN-FORD ( G, s ) :
|     INIT(G, s)
|
|     for i = 1 to n-1:
|     |     for (u,v) in E:
|     |     |     RELAX(u, v)
|
|     for (u,v) in E:
|     |     if v.d < u.d + w(u,v):
|     |     |     return False
|
|     return True, v
```

```
RELAX ( u , v, w ) :
|     if v.d > u.d + w(u,v):
|     |     v.d = u.d + w(u,v)
|     |     v.pred = u
```

Lema (casi correctitud):

$G = (V, E, w)$ sin ciclos negativos, luego de $n-1$ iteraciones $\Rightarrow$

$v.d = \delta(s,v) \quad \forall\ v$ alcanzable desde s

Demo:

Bellman-Ford

```
for i = 1 to n-1:
|     for (u,v) in E:
|     |     RELAX(u, v)
```

```
RELAX ( u , v, w ) :
|     if v.d > u.d + w(u,v):
|     |     v.d = u.d + w(u,v)
|     |     v.pred = u
```

Lema (casi correctitud):

$G = (V, E, w)$ sin ciclos negativos, luego de $n-1$ iteraciones $\Rightarrow v.d = \delta(s,v) \ \forall\ v$ alcanzable desde s

Demo:

Sea $P = v_0, v_1, \ldots, v_{k-1}, v_k$ es un camino mínimo de $s = v_0$ a v_k,

como es camino simple $\Rightarrow k \leq n-1$

en cada iteración se relajan todas las aristas, en particular las de P

es decir que en algún momento de la primer iteración relajo $(v_0, v_1) \Rightarrow d(s, v_1) = \delta(s, v_1)$ y no cambia más! (x ***Límite superior***)

luego, en algún momento de la segunda iteración, relajo $(v_1, v_2) \Rightarrow d(s, v_2) = \delta(s, v_2)$ y no cambia más! (x ***Límite superior***, *y* ***propiedad de Relajación***, *lo hice en orden*)

Bellman-Ford

```
for i = 1 to n-1:
|     for (u,v) in E:
|     |     RELAX(u, v)
```

```
RELAX ( u , v, w ) :
|     if v.d > u.d + w(u,v):
|     |     v.d = u.d + w(u,v)
|     |     v.pred = u
```

Lema (casi correctitud):

$G = (V, E, w)$ sin ciclos negativos, luego de $n-1$ iteraciones $\Rightarrow v.d = \delta(s,v) \ \forall\ v$ alcanzable desde s

Demo:

Sea $P = v_0, \ldots, v_k$ es un camino mínimo de $s = v_0$ a v_k, como es camino simple $\Rightarrow k \leq n-1$

es decir que en algún momento de la primer iteración relajo $(v_0, v_1) \Rightarrow d(s, v_1) = \delta(s, v_1)$ y no cambia más! (x ***Límite superior***)

luego, en algún momento de la segunda iteración, relajo $(v_1, v_2) \Rightarrow d(s, v_2) = \delta(s, v_2)$ y no cambia más! (x ***Límite superior***, *y* ***propiedad de Relajación***, *lo hice en orden*)

luego, en algún momento de la tercera iteración, relajo $(v_2, v_3) \Rightarrow d(s, v_3) = \delta(s, v_3)$ y no cambia más! (x ***Límite superior***, *y* ***propiedad de Relajación***, *lo hice en orden*)

…, en algún momento de la k-ésima iteración, relajo $(v_{k-1}, v_k) \Rightarrow d(s, v_k) = \delta(s, v_k)$ y no cambia más! (x ***Límite superior***, *y* ***propiedad de Relajación***, *lo hice en orden*)

Bellman-Ford

```
for i = 1 to n-1:
|     for (u,v) in E:
|     |     RELAX(u, v)
```

```
RELAX ( u , v, w ) :
|     if v.d > u.d + w(u,v):
|     |     v.d = u.d + w(u,v)
|     |     v.pred = u
```

Lema (casi correctitud):

$G = (V, E, w)$ sin ciclos negativos, luego de $n-1$ iteraciones $\Rightarrow v.d = \delta(s,v) \ \forall\ v$ alcanzable desde s

Demo:

Sea $P = v_0, v_1, \ldots, v_{k-1}, v_k$ es un camino mínimo de $s = v_0$ a v_k,

como es camino simple $\Rightarrow k \leq n-1$

en cada iteración se relajan todas las aristas, en particular las de P

…, en algún momento de la k-ésima iteración, relajo $(v_{k-1}, v_k) \Rightarrow d(s, v_k) = \delta(s, v_k)$ y no cambia más! (x ***Límite superior***, *y* ***propiedad de Relajación***, *lo hice en orden*)

y k es a lo sumo $n-1$

$\Rightarrow v.d = d(s, v) = \delta(s, v) \ \forall\ v$ alcanzable desde s

Bellman-Ford

```
for i = 1 to n-1:
|     for (u,v) in E:
|     |     RELAX(u, v)
```

```
RELAX ( u , v, w ) :
|     if v.d > u.d + w(u,v):
|     |     v.d = u.d + w(u,v)
|     |     v.pred = u
```

Lema (casi correctitud):

G = (V, E, w) sin ciclos negativos, luego de *n-1* iteraciones $\Rightarrow$ $v.d = \delta(s,v) \ \forall \ v$ alcanzable desde *s*

Corolario:

Luego de ejecutar Bellman-Ford, si no tengo ciclos negativos, obtengo un camino simple entre *s* y *v*, y $v.d = \delta(s,v) \ \forall \ v$ alcanzable desde *s*

Bellman-Ford

```
BELLMAN-FORD ( G, s ) :
|     INIT(G, s)
|
|     for i = 1 to n-1:
|     |     for (u,v) in E:
|     |     |     RELAX(u, v)
|
|     for (u,v) in E:
|     |     if v.d < u.d + w(u,v):
|     |     |     return False
|
|     return True, v
```

```
RELAX ( u , v, w ) :
|     if v.d > u.d + w(u,v):
|     |     v.d = u.d + w(u,v)
|     |     v.pred = u
```

Teorema (correctitud):

Si $G = (V, E, w)$ no contiene ciclos negativos alcanzable desde s, luego de ejecutar Bellman-Ford $\Rightarrow$ $v.d = \delta(s,v) \ \forall \ v$ alcanzable desde s y los predecesores ($v.pred$) forman un *s-ACM*

Si $G = (V, E, w)$ contiene ciclos negativos alcanzable desde s, luego de ejecutar Bellman-Ford $\Rightarrow$ Devuelve *False*

Demo:

Bellman-Ford

```
BELLMAN-FORD ( G, s ) :
|     INIT(G, s)
|
|     for i = 1 to n-1:
|     |     for (u,v) in E:
|     |     |     RELAX(u, v)
|
|     for (u,v) in E:
|     |     if v.d < u.d + w(u,v):
|     |     |     return False
|
|     return True, v
```

```
RELAX ( u , v, w ) :
|     if v.d > u.d + w(u,v):
|     |     v.d = u.d + w(u,v)
|     |     v.pred = u
```

Teorema (correctitud):

Si $G = (V, E, w)$ no contiene ciclos negativos alcanzable desde s, luego de ejecutar Bellman-Ford $\Rightarrow$ $v.d = \delta(s,v)$ $\forall$ v alcanzable desde s y los predecesores ($v.pred$) forman un *s-ACM*

Si $G = (V, E, w)$ contiene ciclos negativos alcanzable desde s, luego de ejecutar Bellman-Ford $\Rightarrow$ Devuelve *False*

Demo:

Si $G = (V, E, w)$ no contiene ciclos negativos alcanzable desde s, luego de ejecutar Bellman-Ford $\Rightarrow$

(por ***Lema anterior*** + ***Límite superior***, *es decir que en el segundo for no se actuliza más*)

$v.d = \delta(s,v)$ $\forall$ v alcanzable desde s y los predecesores ($v.pred$) forman un *s-ACM*

Bellman-Ford

```
BELLMAN-FORD ( G, s ) :
|     INIT(G, s)
|
|     for i = 1 to n-1:
|     |     for (u,v) in E:
|     |     |     RELAX(u, v)
|
|     for (u,v) in E:
|     |     if v.d < u.d + w(u,v):
|     |     |     return False
|
|     return True, v
```

```
RELAX ( u , v, w ) :
|     if v.d > u.d + w(u,v):
|     |     v.d = u.d + w(u,v)
|     |     v.pred = u
```

Teorema (correctitud):

…

Si $G = (V, E, w)$ contiene ciclos negativos alcanzable desde s, luego de ejecutar Bellman-Ford $\Rightarrow$ Devuelve *False*

Demo:

Si $G = (V, E, w)$ contiene ciclos negativos alcanzable desde s, luego de ejecutar Bellman-Ford $\Rightarrow$ Existe $C = v_0, \ldots v_k$, con $v_0 = v_k$ tq $\sum_i w(v_{i-1}, v_i) < 0$

(por el absurdo) Supongo que devuelve True $\Rightarrow$ No entra a ningún "if" $\Rightarrow$

Bellman-Ford

```
BELLMAN-FORD ( G, s ) :
|     INIT(G, s)
|
|     for i = 1 to n-1:
|     |     for (u,v) in E:
|     |     |     RELAX(u, v)
|
|     for (u,v) in E:
|     |     if v.d < u.d + w(u,v):
|     |     |     return False
|
|     return True, v
```

```
RELAX ( u , v, w ) :
|     if v.d > u.d + w(u,v):
|     |     v.d = u.d + w(u,v)
|     |     v.pred = u
```

Demo: Si $G = (V, E, w)$ contiene ciclos negativos alcanzable desde s, luego de ejecutar Bellman-Ford $\Rightarrow$ Existe $C = v_0, \ldots v_k$, con $v_0 = v_k$ tq $\sum_i w(v_{i-1}, v_i) < 0$

(por el absurdo) Supongo que devuelve True $\Rightarrow$ No entra a ningún "if" $\Rightarrow$

$d(s,v_i) \leq d(s,v_{i-1}) + w(v_{i-1},v_i) \Rightarrow$

$d(s,v_i) - d(s,v_{i-1}) \leq w(v_{i-1},v_i) \Rightarrow$

$\sum_i (d(s,v_i) - d(s,v_{i-1})) \leq \sum_i w(v_{i-1},v_i) < 0$

$v_1 - v_0 + v_2 - v_1 + \ldots + v_{k-1} - v_{k-2} + v_k - v_{k-1} =$

$v_k - v_0 =$

$v_0 - v_0 =$

0

Bellman-Ford

```
BELLMAN-FORD ( G, s ) :
|     INIT(G, s)
|
|     for i = 1 to n-1:
|     |     for (u,v) in E:
|     |     |     RELAX(u, v)
|
|     for (u,v) in E:
|     |     if v.d < u.d + w(u,v):
|     |     |     return False
|
|     return True, v
```

```
RELAX ( u , v, w ) :
|     if v.d > u.d + w(u,v):
|     |     v.d = u.d + w(u,v)
|     |     v.pred = u
```

Demo: Si $G = (V, E, w)$ contiene ciclos negativos alcanzable desde s, luego de ejecutar Bellman-Ford $\Rightarrow$ Existe $C = v_0, \ldots v_k$, con $v_0 = v_k$ tq $\sum_i w(v_{i-1},v_i) < 0$

(por el absurdo) Supongo que devuelve True $\Rightarrow$ No entra a ningún "if" $\Rightarrow$

$d(s,v_i) \leq d(s,v_{i-1}) + w(v_{i-1},v_i) \Rightarrow$

$d(s,v_i) - d(s,v_{i-1}) \leq w(v_{i-1},v_i) \Rightarrow$

$\sum_i (d(s,v_i) - d(s,v_{i-1})) \leq \sum_i w(v_{i-1},v_i) < 0$

$v_1 - v_0 + v_2 - v_1 + \ldots + v_{k-1} - v_{k-2} + v_k - v_{k-1} = v_k - v_0 = v_0 - v_0 = 0$

$0 \leq \sum_i w(v_{i-1},v_i) < 0$

¡Absurdo!

Bellman-Ford

```
BELLMAN-FORD ( G, s ) :
|     INIT(G, s)
|
|     for i = 1 to n-1:
|     |     for (u,v) in E:
|     |     |     RELAX(u, v)
|     |
|     for (u,v) in E:
|     |     if v.d < u.d + w(u,v):
|     |     |     return False
|     |
|     return True, v
```

O(V)

O(1) } *O(E)* } *O(V*E)*

O(1) } *O(E)*

*O(V + V*E + E)*

denso (E~V²): *O(V³)*

ralo (E~V): *O(V²)*

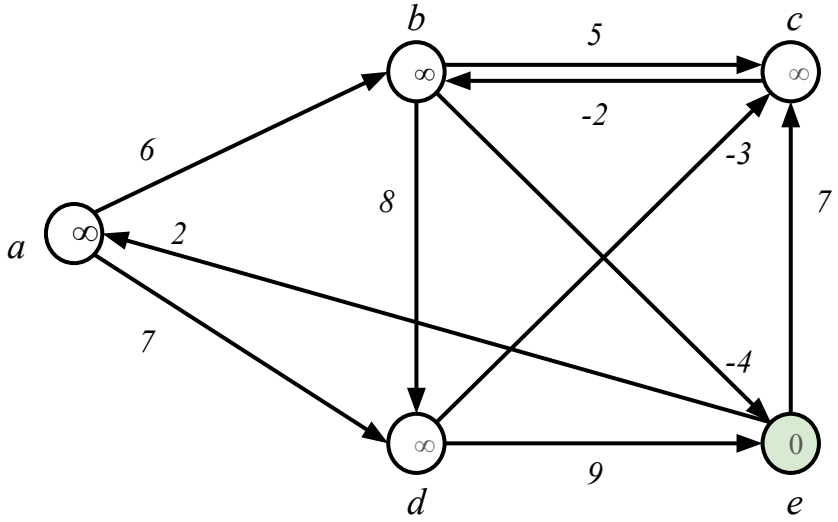
```
RELAX ( u , v, w ) :
|     if v.d > u.d + w(u,v):
|     |     v.d = u.d + w(u,v)
|     |     v.pred = u
```

O(1)

Bellman-Ford

```
BELLMAN-FORD ( G, s ) :
|     INIT(G, s)
|
|     for i = 1 to n-1:
|     |     for (u,v) in E:
|     |     |     RELAX(u, v)
|
|     for (u,v) in E:
|     |     if v.d < u.d + w(u,v):
|     |     |     return False
|
|     return True, v
```

```
RELAX ( u , v, w ) :
|     if v.d > u.d + w(u,v):
|     |     v.d = u.d + w(u,v)
|     |     v.pred = u
```



(e,c), (c,b), (b,c), (a,b), (a,d), (d,e), (b,e), (d,c), (b,d), (b,c), (e,a)

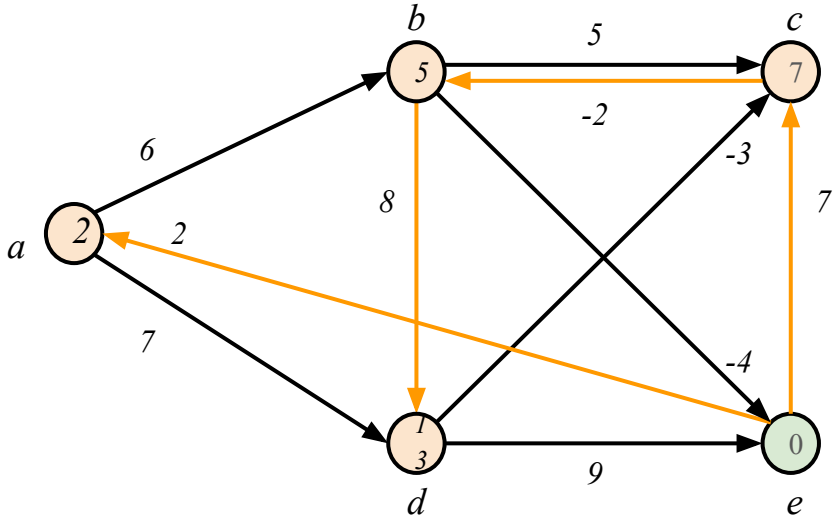
d = {a: ∞, b: ∞, c: ∞, d: ∞, e: 0}

pred = {a: None, b: None, c: None, d: None, e: None}

Bellman-Ford

```
BELLMAN-FORD ( G, s ) :
|     INIT(G, s)
|
|     for i = 1 to n-1:
|     |     for (u,v) in E:
|     |     |     RELAX(u, v)
|
|     for (u,v) in E:
|     |     if v.d < u.d + w(u,v):
|     |     |     return False
|
|     return True, v
```

```
RELAX ( u , v, w ) :
|     if v.d > u.d + w(u,v):
|     |     v.d = u.d + w(u,v)
|     |     v.pred = u
```



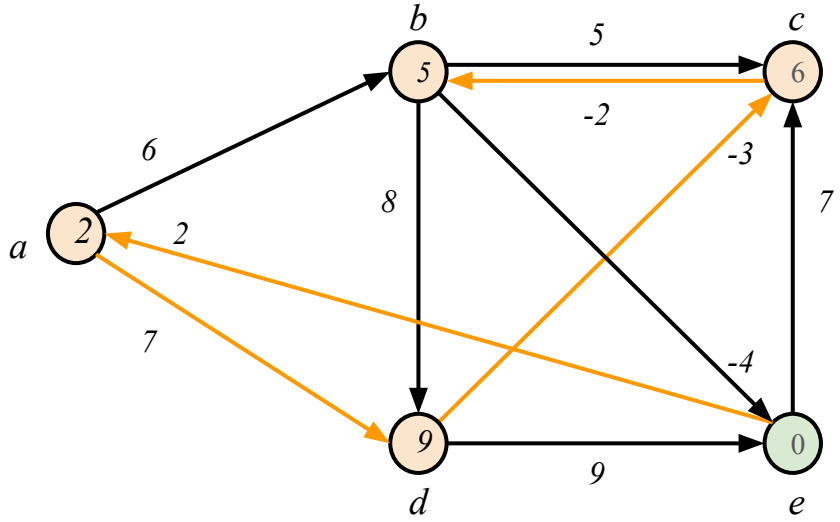
(e,c), (c,b), (b,c), (a,b), (a,d), (d,e), (b,e), (d,c), (b,d), (b,c), (e,a)

i = *1*
d = *{a: 2, b: 5, c: 7, d: 13, e: 0}*
pred = *{a: e, b: c, c: e, d: b, e: None}*

Bellman-Ford

```
BELLMAN-FORD ( G, s ) :
|     INIT(G, s)
|
|     for i = 1 to n-1:
|     |     for (u,v) in E:
|     |     |     RELAX(u, v)
|
|     for (u,v) in E:
|     |     if v.d < u.d + w(u,v):
|     |     |     return False
|
|     return True, v
```

```
RELAX ( u , v, w ) :
|     if v.d > u.d + w(u,v):
|     |     v.d = u.d + w(u,v)
|     |     v.pred = u
```



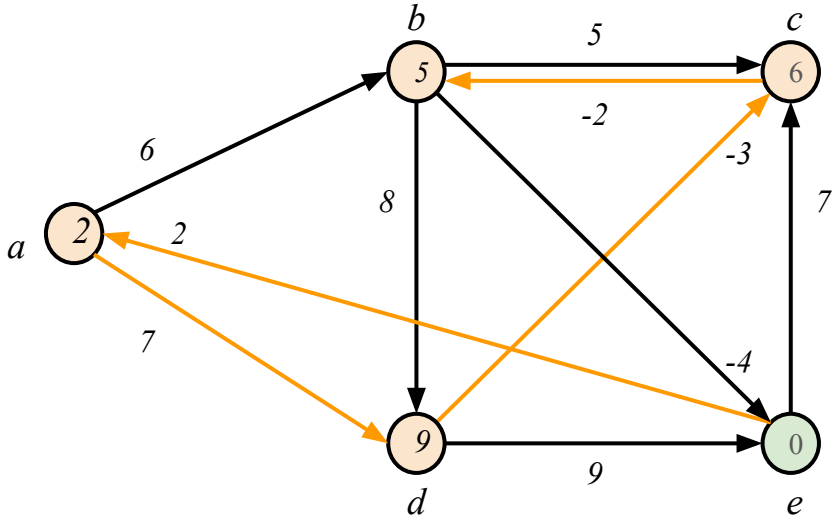
(e,c), (c,b), (b,c), (a,b), (a,d), (d,e), (b,e), (d,c), (b,d), (b,c), (e,a)

i = 2
d = {a: 2, b: 5, c: 6, d: 9, e: 0}
pred = {a: e, b: c, c: d, d: a, e: None}

Bellman-Ford

```
BELLMAN-FORD ( G, s ) :
|     INIT(G, s)
|
|     for i = 1 to n-1:
|     |     for (u,v) in E:
|     |     |     RELAX(u, v)
|
|     for (u,v) in E:
|     |     if v.d < u.d + w(u,v):
|     |     |     return False
|
|     return True, v
```

```
RELAX ( u , v, w ) :
|     if v.d > u.d + w(u,v):
|     |     v.d = u.d + w(u,v)
|     |     v.pred = u
```



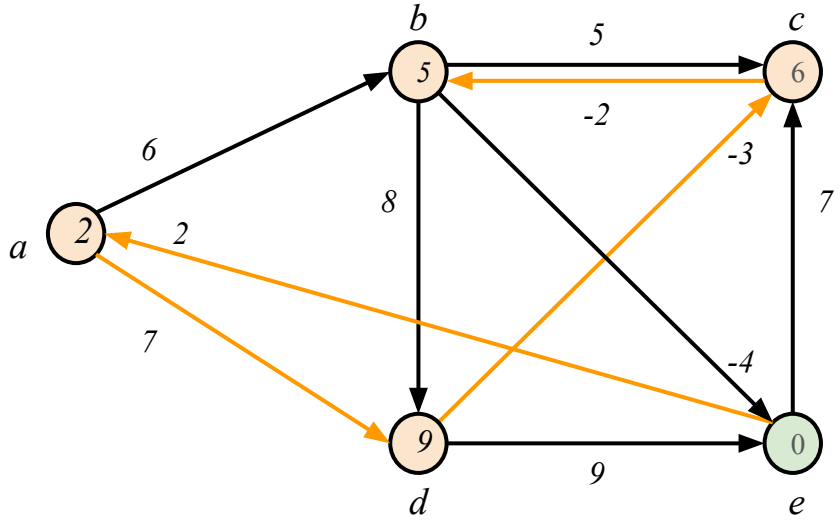
(e,c), (c,b), (b,c), (a,b), (a,d), (d,e), (b,e), (d,c), (b,d), (b,c), (e,a)

i = *3…*
d = *{a: 2, b: 5, c: 6, d: 9, e: 0}*
pred = *{a: e, b: c, c: d, d: a, e: None}*

Bellman-Ford

```
BELLMAN-FORD ( G, s ) :
|     INIT(G, s)
|
|     for i = 1 to n-1:
|     |     for (u,v) in E:
|     |     |     RELAX(u, v)
|
|     for (u,v) in E:
|     |     if v.d < u.d + w(u,v):
|     |     |     return False
|
|     return True, v
```

```
RELAX ( u , v, w ) :
|     if v.d > u.d + w(u,v):
|     |     v.d = u.d + w(u,v)
|     |     v.pred = u
```



(e,c), (c,b), (b,c), (a,b), (a,d), (d,e), (b,e), (d,c), (b,d), (b,c), (e,a)

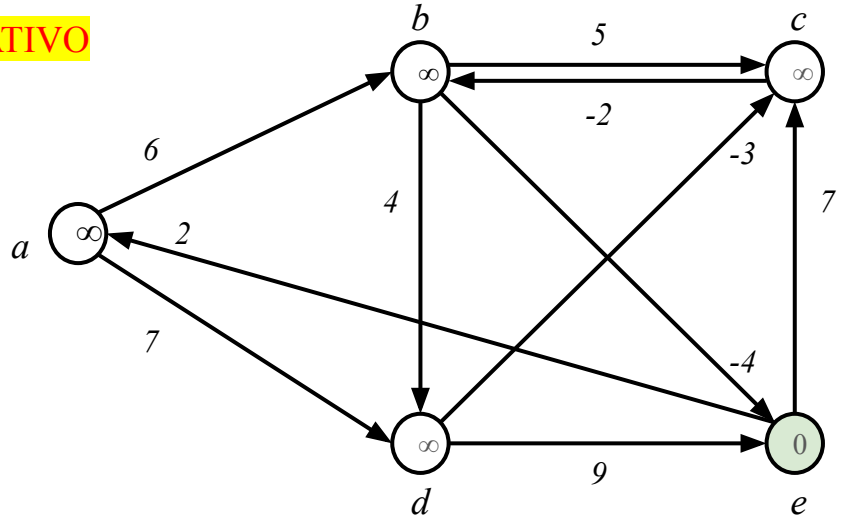
i = 3...
d = {a: 2, b: 5, c: 6, d: 9, e: 0}
pred = {a: e, b: c, c: d, d: a, e: None}

Bellman-Ford

```
BELLMAN-FORD ( G, s ) :
|     INIT(G, s)
|
|     for i = 1 to n-1:
|     |     for (u,v) in E:
|     |     |     RELAX(u, v)
|
|     for (u,v) in E:
|     |     if v.d < u.d + w(u,v):
|     |     |     return False
|
|     return True, v
```

```
RELAX ( u , v, w ) :
|     if v.d > u.d + w(u,v):
|     |     v.d = u.d + w(u,v)
|     |     v.pred = u
```

CON CICLO NEGATIVO



(e,c), (c,b), (b,c), (a,b), (a,d), (d,e), (b,e), (d,c), (b,d), (b,c), (e,a)

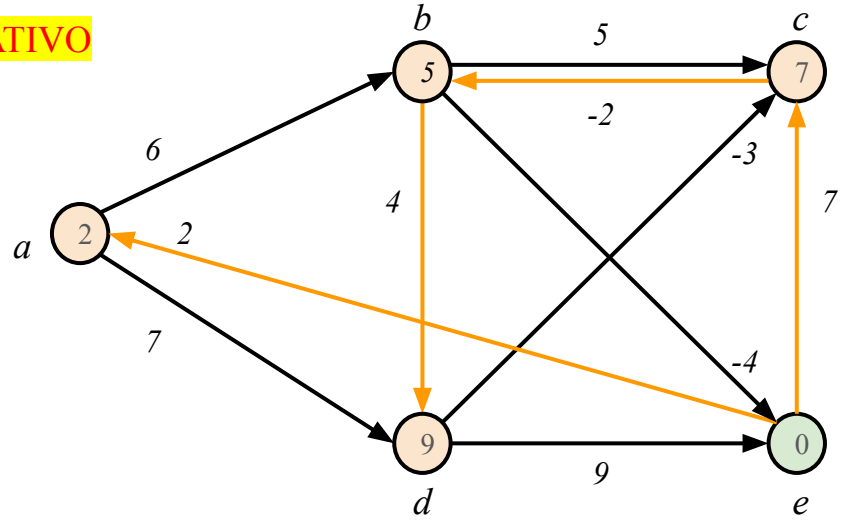
d = {a: ∞, b: ∞, c: ∞, d: ∞, e: 0}
pred = {a: None, b: None, c: None, d: None, e: None}

Bellman-Ford

```
BELLMAN-FORD ( G, s ) :
|     INIT(G, s)
|
|     for i = 1 to n-1:
|     |     for (u,v) in E:
|     |     |     RELAX(u, v)
|
|     for (u,v) in E:
|     |     if v.d < u.d + w(u,v):
|     |     |     return False
|
|     return True, v
```

```
RELAX ( u , v, w ) :
|     if v.d > u.d + w(u,v):
|     |     v.d = u.d + w(u,v)
|     |     v.pred = u
```

CON CICLO NEGATIVO



(e,c), (c,b), (b,c), (a,b), (a,d), (d,e), (b,e), (d,c), (b,d), (b,c), (e,a)

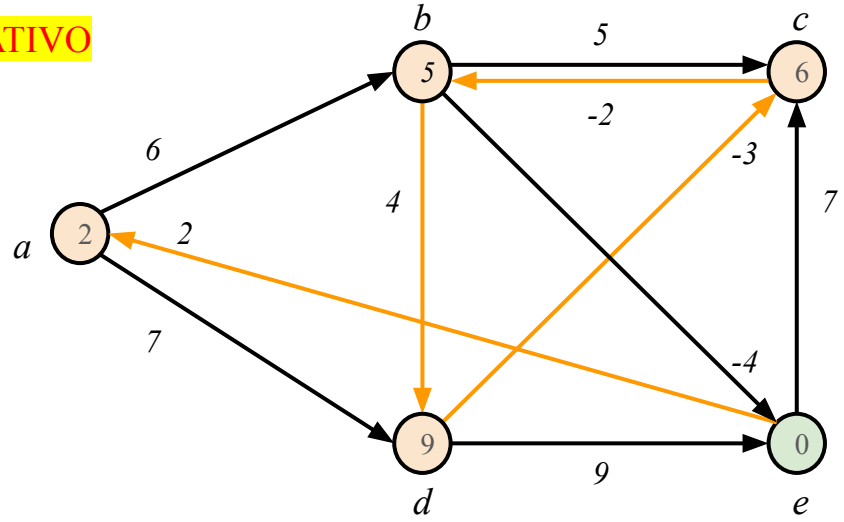
i = 1
d = {a: 2, b: 5, c: 7, d: 9, e: 0}
pred = {a: e, b: c, c: e, d: b, e: None}

Bellman-Ford

```
BELLMAN-FORD ( G, s ) :
|     INIT(G, s)
|
|     for i = 1 to n-1:
|     |     for (u,v) in E:
|     |     |     RELAX(u, v)
|
|     for (u,v) in E:
|     |     if v.d < u.d + w(u,v):
|     |     |     return False
|
|     return True, v
```

```
RELAX ( u , v, w ) :
|     if v.d > u.d + w(u,v):
|     |     v.d = u.d + w(u,v)
|     |     v.pred = u
```

CON CICLO NEGATIVO



(e,c), (c,b), (b,c), (a,b), (a,d), (d,e), (b,e), (d,c), (b,d), (b,c), (e,a)

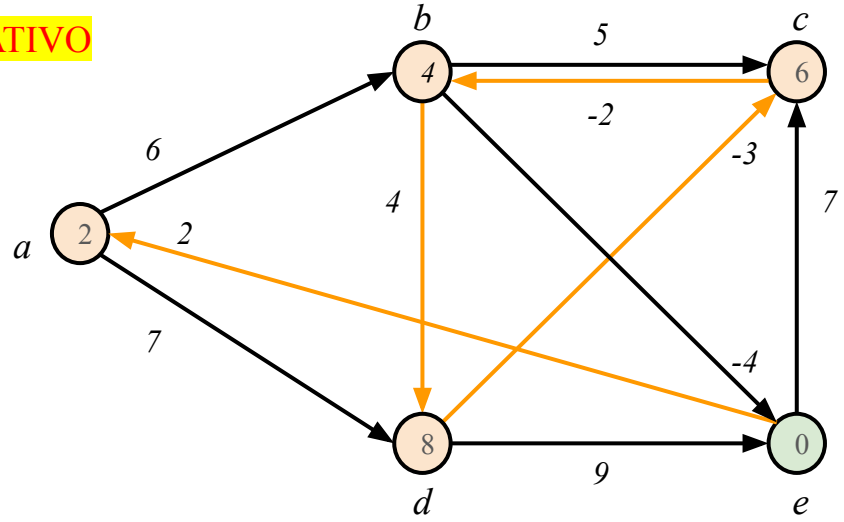
i = 2
d = {a: 2, b: 5, c: 6, d: 9, e: 0}
pred = {a: e, b: c, c: d, d: b, e: None}

Bellman-Ford

```
BELLMAN-FORD ( G, s ) :
|     INIT(G, s)
|
|     for i = 1 to n-1:
|     |     for (u,v) in E:
|     |     |     RELAX(u, v)
|
|     for (u,v) in E:
|     |     if v.d < u.d + w(u,v):
|     |     |     return False
|
|     return True, v
```

```
RELAX ( u , v, w ) :
|     if v.d > u.d + w(u,v):
|     |     v.d = u.d + w(u,v)
|     |     v.pred = u
```

CON CICLO NEGATIVO



(e,c), (c,b), (b,c), (a,b), (a,d), (d,e), (b,e), (d,c), (b,d), (b,c), (e,a)

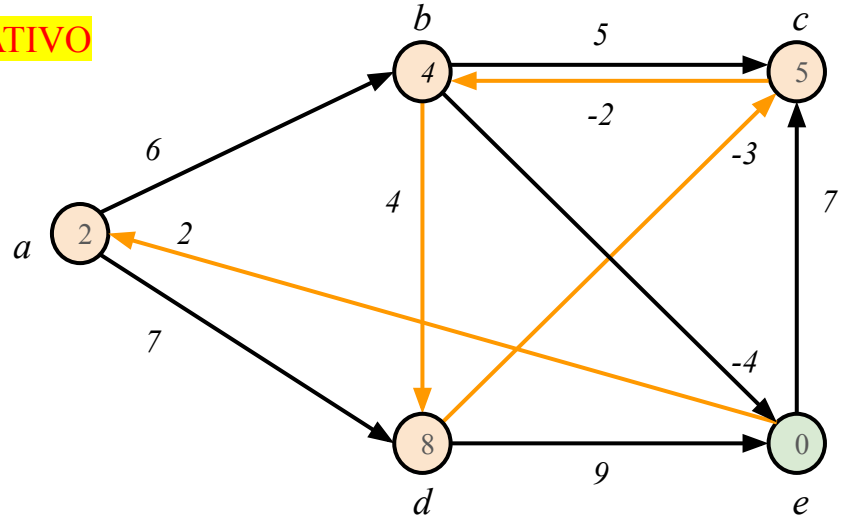
i = 3
d = {a: 2, b: 4, c: 6, d: 8, e: 0}
pred = {a: e, b: c, c: d, d: b, e: None}

Bellman-Ford

```
BELLMAN-FORD ( G, s ) :
|     INIT(G, s)
|
|     for i = 1 to n-1:
|     |     for (u,v) in E:
|     |     |     RELAX(u, v)
|
|     for (u,v) in E:
|     |     if v.d < u.d + w(u,v):
|     |     |     return False
|
|     return True, v
```

```
RELAX ( u , v, w ) :
|     if v.d > u.d + w(u,v):
|     |     v.d = u.d + w(u,v)
|     |     v.pred = u
```

CON CICLO NEGATIVO



(e,c), (c,b), (b,c), (a,b), (a,d), (d,e), (b,e), (d,c), (b,d), (b,c), (e,a)

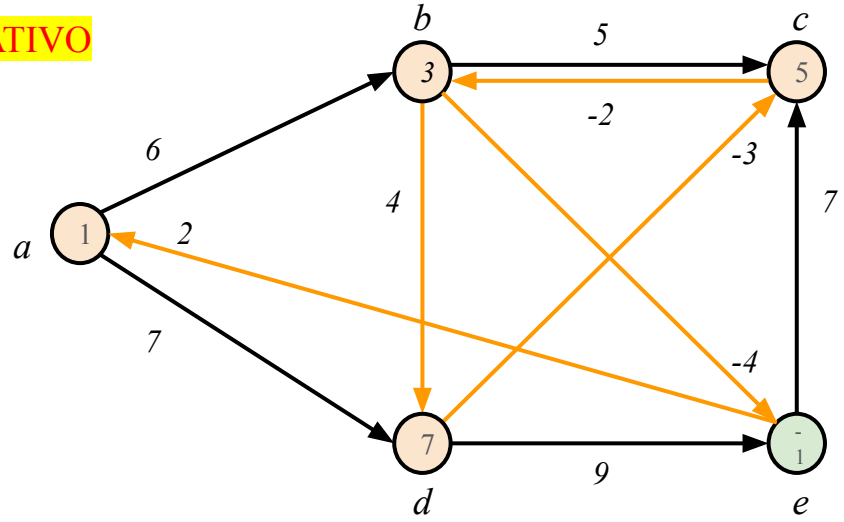
i = 4
d = {a: 2, b: 4, c: 5, d: 8, e: 0}
pred = {a: e, b: c, c: d, d: b, e: None}

Bellman-Ford

```
BELLMAN-FORD ( G, s ) :
|     INIT(G, s)
|
|     for i = 1 to n-1:
|     |     for (u,v) in E:
|     |     |     RELAX(u, v)
|
|     for (u,v) in E:
|     |     if v.d < u.d + w(u,v):
|     |     |     return False
|
|     return True, v
```

```
RELAX ( u , v, w ) :
|     if v.d > u.d + w(u,v):
|     |     v.d = u.d + w(u,v)
|     |     v.pred = u
```

CON CICLO NEGATIVO



(e,c), (c,b), (b,c), (a,b), (a,d), (d,e), (b,e), (d,c), (b,d), (b,c), (e,a)

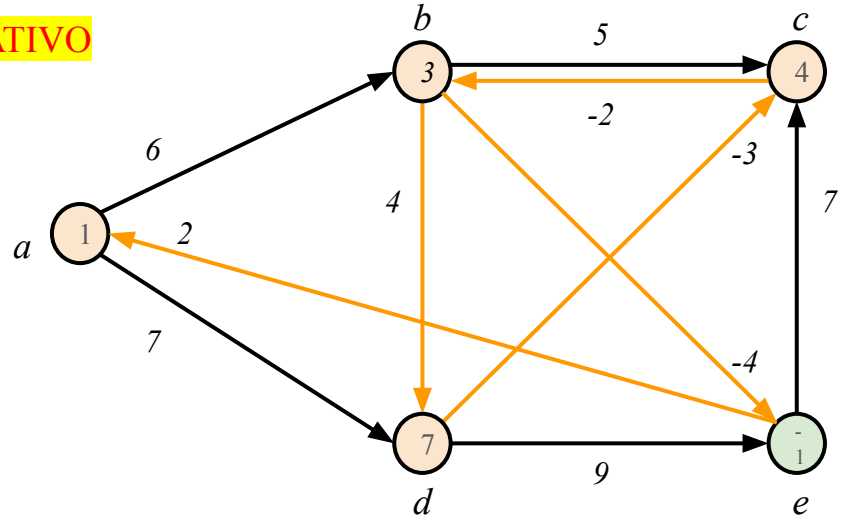
i = 5
d = {a: 1, b: 3, c: 5, d: 7, e: -1}
pred = {a: e, b: c, c: d, d: b, e: b}

Bellman-Ford

```
BELLMAN-FORD ( G, s ) :
|     INIT(G, s)
|
|     for i = 1 to n-1:
|     |     for (u,v) in E:
|     |     |     RELAX(u, v)
|
|     for (u,v) in E:
|     |     if v.d < u.d + w(u,v):
|     |     |     return False
|
|     return True, v
```

```
RELAX ( u , v, w ) :
|     if v.d > u.d + w(u,v):
|     |     v.d = u.d + w(u,v)
|     |     v.pred = u
```

CON CICLO NEGATIVO



(e,c), (c,b), (b,c), (a,b), (a,d), (d,e), (b,e), (d,c), (b,d), (b,c), (e,a)

d = {a: 1, b: 3, **c: 4**, d: 7, e: -1}
pred = {a: e, b: c, c: d, d: b, e: b}

Programación lineal

En general, dados: *A: m x m*, *b: n x m*, *c: n x 1*. Se quiere encontrar *x: n x 1* tq

MAXIMICE la FUNCIÓN OBJETIVO: $\Sigma_i c_i x_i$

con las RESTRICCIONES: $Ax \leq b$

Se conocen algoritmos polinomiales como SIMPLEX para resolverlo si se sabe que la solución existe

Programación lineal

En general, dados: *A: m x m*, *b: n x m*, *c: n x 1*. Se quiere encontrar *x: n x 1* tq

MAXIMICE la FUNCIÓN OBJETIVO: $\Sigma_i c_i x_i$

con las RESTRICCIONES: $Ax \leq b$

Se conocen algoritmos polinomiales como SIMPLEX para resolverlo si se sabe que la solución existe o para algunos casos particulares.

Programación lineal

En general, dados: *A: m x m*, *b: n x m*, *c: n x 1*. Se quiere encontrar *x: n x 1* tq

MAXIMICE la FUNCIÓN OBJETIVO: $\Sigma_i c_i x_i$

con las RESTRICCIONES: $Ax \leq b$

Se conocen algoritmos polinomiales como SIMPLEX para resolverlo si se sabe que la solución existe o para algunos casos particulares.

PERO si queremos saber si la solución existe

Programación lineal

En general, dados: *A: m x m*, *b: n x m*, *c: n x 1*. Se quiere encontrar *x: n x 1* tq

MAXIMICE la FUNCIÓN OBJETIVO: $\Sigma_i c_i x_i$

con las RESTRICCIONES: $Ax \leq b$

Se conocen algoritmos polinomiales como SIMPLEX para resolverlo si se sabe que la solución existe o para algunos casos particulares.

PERO si queremos saber si la solución existe, y no nos interesa la función objetivo

⇒ **BELLMAN - FORD**

Sistemas de diferencias

$$A \quad x \quad \leq \quad b$$

$$\begin{bmatrix} 1 & -1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & -1 \\ 0 & 1 & 0 & 0 & -1 \\ -1 & 0 & 1 & 0 & 0 \\ -1 & 0 & 0 & 1 & 0 \\ 0 & 0 & -1 & 1 & 0 \\ 0 & 0 & -1 & 0 & 1 \\ 0 & 0 & 0 & -1 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{bmatrix} \leq \begin{bmatrix} 0 \\ -1 \\ 1 \\ 5 \\ 4 \\ -1 \\ -3 \\ -3 \end{bmatrix} \Rightarrow \begin{bmatrix} x_1 - x_2 \leq 0 \\ x_1 - x_5 \leq -1 \\ x_2 - x_5 \leq 1 \\ x_3 - x_1 \leq 5 \\ x_4 - x_1 \leq 4 \\ x_4 - x_3 \leq -1 \\ x_5 - x_3 \leq -3 \\ x_5 - x_4 \leq -3 \end{bmatrix}$$

Sistemas de diferencias

$$A \quad x \quad \leq \quad b$$

$$\begin{bmatrix} 1 & -1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & -1 \\ 0 & 1 & 0 & 0 & -1 \\ -1 & 0 & 1 & 0 & 0 \\ -1 & 0 & 0 & 1 & 0 \\ 0 & 0 & -1 & 1 & 0 \\ 0 & 0 & -1 & 0 & 1 \\ 0 & 0 & 0 & -1 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{bmatrix} \leq \begin{bmatrix} 0 \\ -1 \\ 1 \\ 5 \\ 4 \\ -1 \\ -3 \\ -3 \end{bmatrix}$$

sol.: x = (-5, -3, 0, -1, -4)

sol.: x' = (0, 2, 5, 4, 1)

Lema: Si x es solución de $Ax \leq b$

$\Rightarrow x+d$, $d = cte$ también es solución de $Ax \leq b$

Demo: $x_2 - x_1 \leq b$

$(x_2 + d) - (x_1 + d) \leq b$

$x_2 - x_1 \leq b$

Sistemas de diferencias

$$A \quad x \quad \leq \quad b$$

$$\begin{bmatrix} 1 & -1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & -1 \\ 0 & 1 & 0 & 0 & -1 \\ -1 & 0 & 1 & 0 & 0 \\ -1 & 0 & 0 & 1 & 0 \\ 0 & 0 & -1 & 1 & 0 \\ 0 & 0 & -1 & 0 & 1 \\ 0 & 0 & 0 & -1 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{bmatrix} \leq \begin{bmatrix} 0 \\ -1 \\ 1 \\ 5 \\ 4 \\ -1 \\ -3 \\ -3 \end{bmatrix}$$

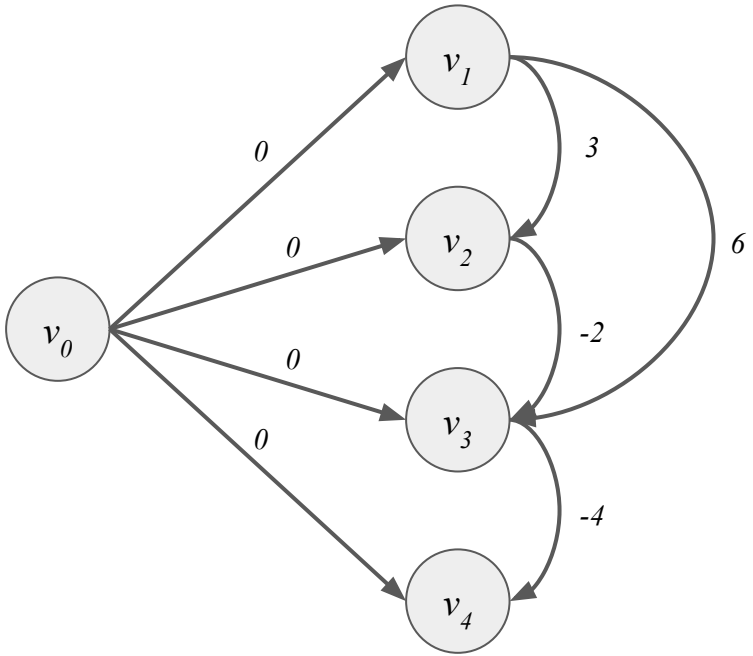
Apps:

Siguiendo con la organización de tareas… esto puede servir para agregarle restricciones de tiempo.

Sistemas de diferencias

$$A \quad x \quad \leq \quad b$$

$$\begin{bmatrix} x_2 - x_1 \leq 3 \\ x_3 - x_2 \leq -2 \\ x_3 - x_1 \leq 6 \\ x_4 - x_3 \leq -4 \end{bmatrix} \Rightarrow$$



$V = \{v_0, v_1, v_2, v_3, v_4\}$

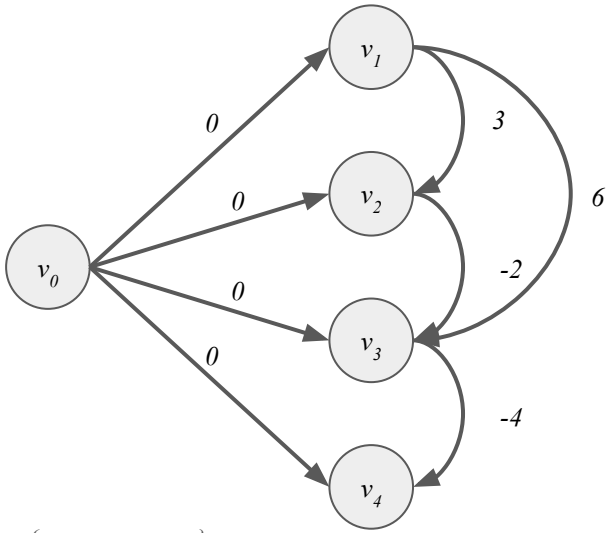
$E = \{(v_0, v_1), (v_0, v_2), (v_0, v_3), (v_0, v_4), (v_1, v_2), (v_2, v_3), (v_3, v_4), (v_1, v_3)\}$

$w(v_i, v_j) = b_k$

$w(v_0, v_i) = 0$

Sistemas de diferencias

$$A \quad x \quad \leq \quad b$$



$V = \{v_0, v_1, v_2, v_3, v_4\}$,

$E = \{(v_i, v_j): x_j - x_i \leq b_k\} \cup \{(v_0, v_1), \ldots, (v_0, v_k)\}$

$w(v_i, v_j) = b_k$, $w(v_0, v_i) = 0$

Teorema:

1) Si G no contiene ciclo negativos $\Rightarrow$

$x = (\delta(v_0,v_1), \delta(v_0,v_2), \ldots, \delta(v_0,v_k))$ es una sol. posible

2) Si G contiene ciclo negativos $\Rightarrow$

no existe solución

Demo:

1) (Desig. triang.) $\Rightarrow$

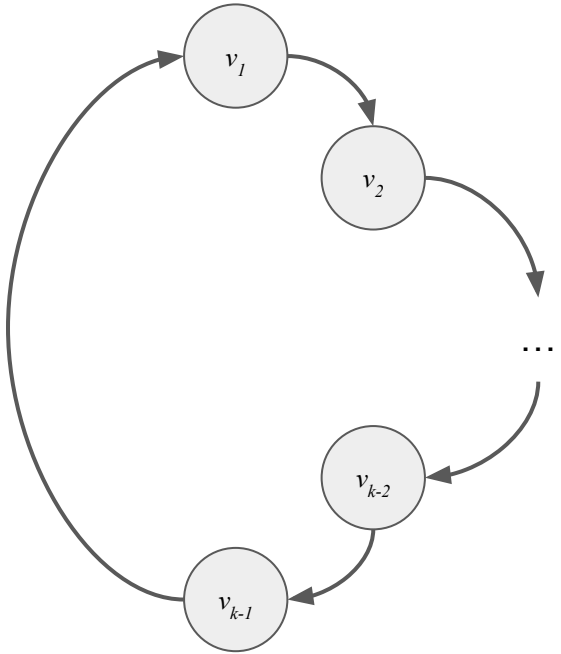
$$\delta(v_0,v_j) \leq \delta(v_0,v_i) + w(v_i,v_j)$$

$$\delta(v_0,v_j) - \delta(v_0,v_i) \leq w(v_i,v_j)$$

$$x_j - x_i \leq b_k$$

Sistemas de diferencias

$$A \quad x \quad \leq \quad b$$



Teorema: 2) Si G contiene ciclo negativos $\Rightarrow$ no existe solución

Demo:

2) (x Absurdo)

Supongo que existe un ciclo neg. $c = v_1, \ldots, v_{k-1}, v_k$ $(v_1 = v_k)$

$$\Rightarrow \quad x_2 - x_1 \leq w(v_2, v_1)$$

$$x_3 - x_2 \leq w(v_3, v_2)$$

$$\ldots$$

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

$$x_k - x_{k-1} \leq w(v_k, v_{k-1})$$

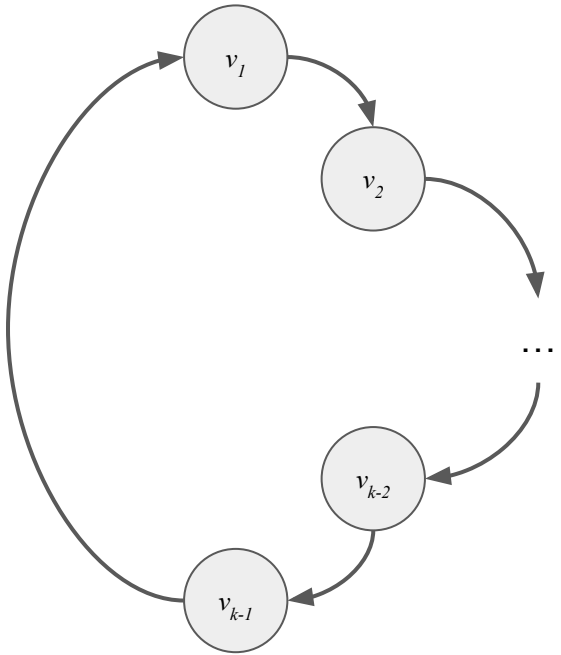
$$x_k - x_{k-1} + x_{k-1} - x_{k-2} + \ldots + x_3 - x_2 + x_2 - x_1 \leq \sum^{k-1} w(v_i, v_{i+1})$$

$$x_k - x_1 \leq \sum^{k-1} w(v_i, v_{i+1})$$

$$(x_k = x_1) \quad 0 \leq \sum^{k-1} w(v_i, v_{i+1}) = w(c)$$

Sistemas de diferencias

$$A \quad x \quad \leq \quad b$$



Teorema: 2) Si *G* contiene ciclo negativos ⇒ no existe solución

Demo:

2) (x Absurdo)

Supongo que existe un ciclo neg. $c = v_1, \ldots, v_{k-1}, v_k$ $(v_1 = v_k)$

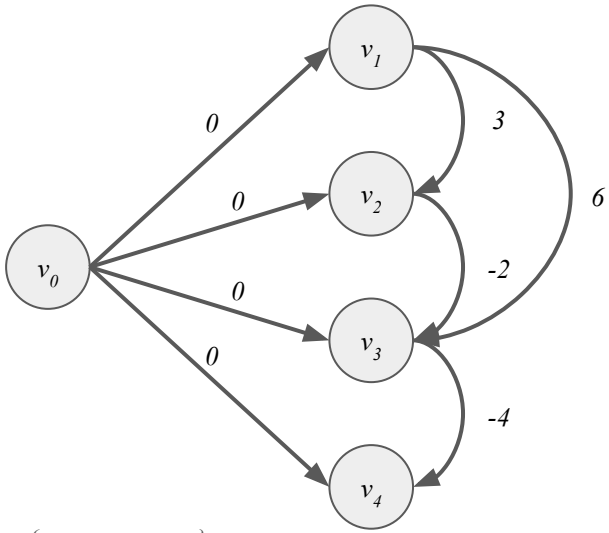
$\Rightarrow \quad 0 \leq \sum^{k-1} w(v_i, v_{i+1}) = w(c)$

$0 \leq w(c) < 0$ *(ciclo negativo)*

¡Absurdo!

Sistemas de diferencias

$$A \quad x \quad \leq \quad b$$



$V = \{v_0, v_1, v_2, v_3, v_4\}$,

$E = \{(v_i, v_j): x_j - x_i \leq b_k\} \cup \{(v_0, v_1), \ldots, (v_0, v_k)\}$

$w(v_i, v_j) = b_k$, $w(v_0, v_i) = 0$

En general, dadas *A: m x m*, *b: n x m*.

Se quiere encontrar *x: n x 1* tq cumpla las RESTRICCIONES Ax≤b

$O(V*E) =$

$O((n + 1)*(m + n))$

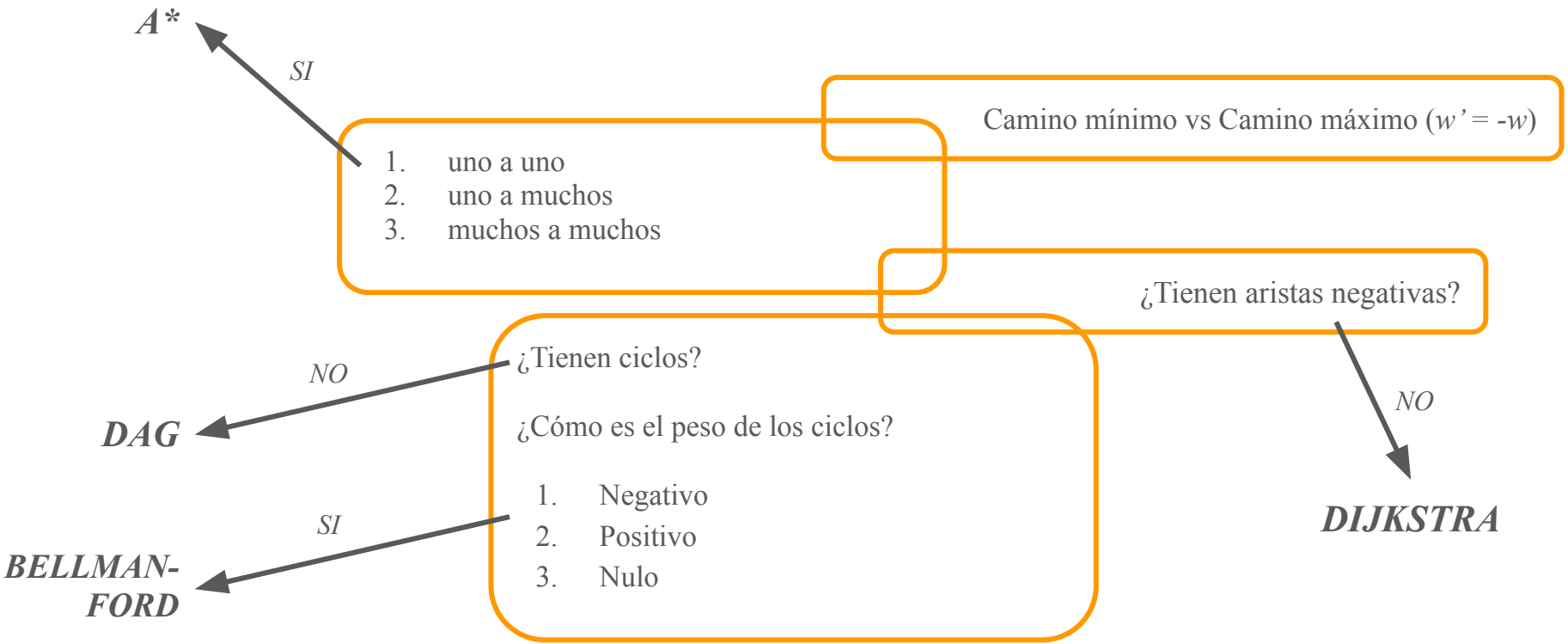
$O(n*m + n^2)$

denso ($m \sim n^2$): $O(n^3)$

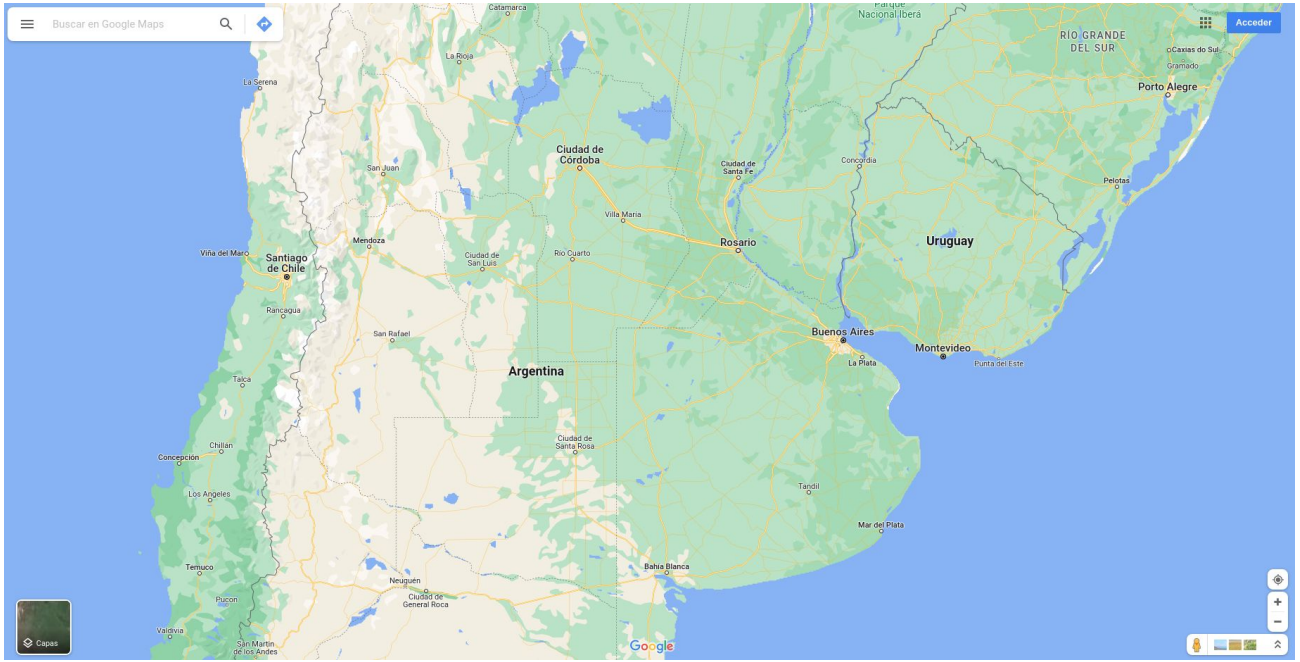
ralo ($m \sim n$): $O(n^2)$

AED3 > Clase 8 > Camino mínimo. Todos a todos.

Repaso: Topología de problemas de camino mínimo



Problema:



Tengo un mapa (tipo Google Maps) en el que voy a consultar recorridos, y quiero que la consulta sea O(1) ⇒ Tengo que pre-computar todas las distancias (pagando con espacio). $d = [\ \delta_{i,j}\ ]$, d: $n \times m$

Repaso: Algoritmos. Uno a todos.

	Algoritmo	*Complejidad*	*Ralo*	*Denso*
no pesado				
no negativos				
general				
DAGs				

Repaso: Algoritmos. Uno a todos.

	Algoritmo	*Complejidad*	*Ralo*	*Denso*
no pesado	BFS	O(V + E)	O(V)	O(V^2)
no negativos	Dijkstra	O(V*log(V) + E)	O(V*log(V))	O(V^2)
general	Bellman-Ford	O(V*E)	O(V^2)	O(V^3)
DAGs	Topological-Sort + “B-F”	O(V + E)	O(V)	O(V^2)

x V (!)

Repaso: Algoritmos. Uno a todos.

	Algoritmo	*Complejidad*	*Ralo*	*Denso*
no pesado	BFS	O((V + E) * V)	O(V^2)	O(V^3)
no negativos	Dijkstra	O((V*log(V) + E) * V)	O(V^2*log(V))	O(V^3)
general	Bellman-Ford	O((V*E) * V)	O(V^3)	O(V^4)

Idea 1: Programación dinámica

1) **Caracterizar la estructura de la solución óptima (subestructura óptima).**

2) **Definir el valor de la solución óptima de forma recursiva.**

3) **Computar el valor de la solución óptima (bottom-up)**

4) **Construir la solución óptima a partir del output**

Programación dinámica: Caracterizar la estructura de la solución óptima (subestructura óptima).

C.M. …

p: $i \rightsquigarrow j$ (c.m.) , $|p| \leq m$ (si no hay ciclos negativos, $m < \infty$)

si $i = j \Rightarrow w(p) = 0$

si $i \neq j \Rightarrow w(p) = 0 \quad \Rightarrow \quad p'$: $i \rightsquigarrow k$ (c.m.), tq p: $i \rightsquigarrow k \longrightarrow j$, $|p'| \leq m$ (c.m.)

$\Rightarrow \quad w(p) = w(p') + w(k,j)$

$\Rightarrow \quad \delta(i,j) = \delta(i,k) + w(k,j)$

Programación dinámica: Definir el valor de la solución óptima de forma recursiva.

$l_{ij}^{(m)}$ es el peso mínimo de cualquier camino de i a j con como máximo m aristas.

si m = 0 $\Rightarrow$ $l_{ij}^{(o)} = 0$ *si* $i=j$

$l_{ij}^{(o)} = \infty$ *si* $i \neq j$

si m > 0 $\Rightarrow$ $\min_{1 \leq k \leq n} (l_{ik}^{(m-1)} + w_{kj})$ (incluyendo k=j)

Programación dinámica: Definir el valor de la solución óptima de forma recursiva.

$l_{ij}^{(m)}$ es el peso mínimo de cualquier camino de i a j con como máximo m aristas.

si m = 0 $\Rightarrow$ $l_{ij}^{(o)} = 0$ *si* $i=j$

$l_{ij}^{(o)} = \infty$ *si* $i \neq j$

si m > 0 $\Rightarrow$ $\min_{1 \leq k \leq n} (l_{ik}^{(m-1)} + w_{kj})$ o $\min (l_{ij}^{(m-1)}, \min_{1 \leq k \leq n} (l_{ik}^{(m-1)} + w_{kj}))$

Programación dinámica: Definir el valor de la solución óptima de forma recursiva.

$l_{ij}^{(m)}$ es el peso mínimo de cualquier camino de i a j con como máximo m aristas.

si m = 0 $\Rightarrow$ $l_{ij}^{(o)} = 0$ *si* $i=j$

$l_{ij}^{(o)} = \infty$ *si* $i \neq j$

si m > 0 $\Rightarrow$ $\min (l_{ij}^{(m-1)}, \min_{1 \leq k \leq n} (l_{ik}^{(m-1)} + w_{kj}))$

$$l_{ij}^{(m)} = \min (l_{ij}^{(m-1)}, \min_{1 \leq k \leq n} (l_{ik}^{(m-1)} + w_{kj}))$$

Programación dinámica: Computar el valor de la solución óptima (bottom-up).

$W = (w_{ij})$, $L^{(m)} = (l_{ij}^{(m)})$

$L^{(1)} \longrightarrow L^{(2)} \longrightarrow L^{(3)} \longrightarrow \ldots \longrightarrow L^{(n-1)}$

Programación dinámica: Computar el valor de la solución óptima (bottom-up).

$W = (w_{ij})$, $L^{(m)} = (l_{ij}^{(m)})$

$L^{(1)} = (w_{ij}) \longrightarrow L^{(2)} \longrightarrow L^{(3)} \longrightarrow \ldots \longrightarrow L^{(n-1)}$

Programación dinámica: Computar el valor de la solución óptima (bottom-up).

$W = (w_{ij})$, $L^{(m)} = (l_{ij}^{(m)})$

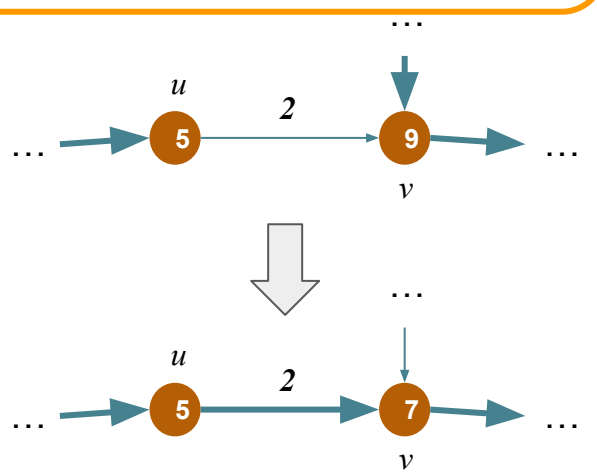
$L^{(1)} = (w_{ij}) \longrightarrow L^{(2)} \longrightarrow L^{(3)} \longrightarrow \ldots \longrightarrow L^{(n-1)} = \delta(i,j)$

```
DP(L,W):
|     n = |L|
|     L' = ( l'ij ), n x n
|     for i = 1…n:
|     |     for j = 1…n:
|     |     |     l'ij = Inf
|     |     |     for k = 1…n
|     |     |     |     l'ij  = min1≤k≤n ( lij , lik + wkj )
```

Programación dinámica: Computar el valor de la solución óptima (bottom-up).

```
RELAX ( u , v, w ) :
|     if v.d > u.d + w(u,v):
|     |     v.d = u.d + w(u,v)
|     |     v.pred = u
```

Propiedad de RELAJACIÓN

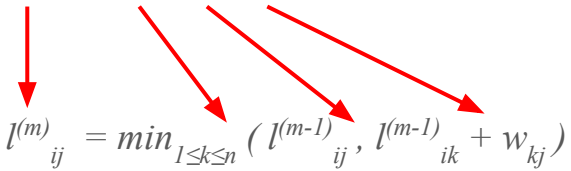



```
DP(L,W):
|     n = |L|
|     L' = ( l'ij ), n x n
|     for i = 1...n:
|     |     for j = 1...n:
|     |     |     l'ij = Inf
|     |     |     for k = 1...n
|     |     |     |     l'ij = min1≤k≤n ( lij , lik + wkj )
```

Multiplicación de matrices

$$C = A * B$$

$$c_{ij} = \Sigma_{k=1\ldots n}\, a_{ik} * b_{kj}$$



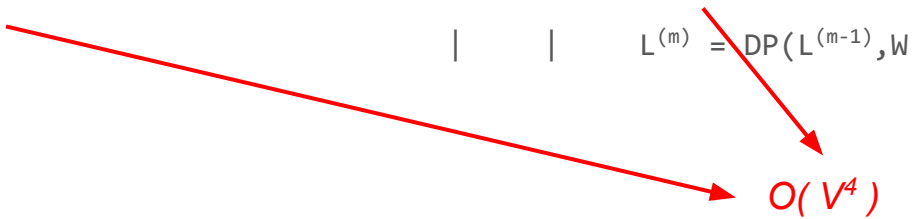
$$l^{(m)}_{ij} = min_{1 \le k \le n} \left(l^{(m-1)}_{ij}, l^{(m-1)}_{ik} + w_{kj} \right)$$

```
MM(A,B):
|     n = |A|
|     ( C : n x n )
|     for i = 1...n:
|     |     for j = 1...n:
|     |     |     c_ij = 0
|     |     |     for k = 1...n
|     |     |     |     c_ij  = c_ij + b_ik * a_kj
```

Programación dinámica: Computar el valor de la solución óptima (bottom-up).

```
DP(L,W):
|    n = |L|
|    L' = ( l'_ij ), n x n
|    for i = 1...n:
|    |    for j = 1...n:
|    |    |    l'_ij = Inf
|    |    |    for k = 1...n
|    |    |    |    l'_ij = min_{1≤k≤n} ( l_ij , l_ik + w_kj )
```

```
APSP(W):
|    n = |W|
|    L^(1) = W
|    for m = 2...n-1
|    |    L^(m) = DP(L^(m-1),W)
```



Programación dinámica: Computar el valor de la solución óptima (bottom-up).

Nos interesa $L^{(n-1)}$ porque $L^{(n-1)} = L^{(n)} = L^{(n+1)} = \dots$

$L^{(1)} = W$

$L^{(2)} = W*W = W^2$

$L^{(4)} = W^2*W^2 = W^4$

…

$L^{(m)} = W^{m/2}*W^{m/2} = W^m$

$m \geq n-1 \Rightarrow \log(n-1)\dots$

```
APSP(W):
|     n = |W|
|     L(1) = W
|     while t ≤ n-1
|     |     L(m) = DP(L(m-1),L(m-1))
              t = 2m
```

$O(V^3 \log(V))$

Programación dinámica: Construir la solución óptima a partir del output.

Algoritmos. Todos a todos.

	Algoritmo	*Complejidad (uno a todos)*	*Complejidad (todos a todos)*	*Ralo*	*Denso*
no pesado	BFS	O(V + E)	O(V*E)	O(V^2)	O(V^3)
no negativos	Dijkstra	O(V*log(V) + E)	O(V^2*log(V) + V*E)	O(V^2*log(V))	O(V^3)
general	Bellman-Ford	O(V*E)	O(V^2*E)	O(V^3)	O(V^4)
general	Program. Dinámica		O(V^3*log(V))	O(V^3*log(V))	O(V^3*log(V))

Floyd-Warshall

Floyd-Warshall

```
FLOYD-WARSHALL(G,W):
|     n = |W|
|     D(0) = W
|     for k = 1...n
|     |     for i = 1...n
|     |     |     for j = 1...n:
|     |     |     |     d(k)ij = min( d(k-1)ij , d(k-1)ik +d(k-1)kj )
```

```
RELAX ( u , v, w ) :
|     if v.d > u.d + w(u,v):
|     |     v.d = u.d + w(u,v)
|     |     v.pred = u
```

Propiedad de RELAJACIÓN

Algoritmos. Todos a todos.

	Algoritmo	*Complejidad (uno a todos)*	*Complejidad (todos a todos)*	*Ralo*	*Denso*
no pesado	BFS	O(V + E)	O(V*E)	O(V^2)	O(V^3)
no negativos	Dijkstra	O(V*log(V) + E)	O(V^2*log(V) + V*E)	O(V^2*log(V))	O(V^3)
general	Bellman-Ford	O(V*E)	O(V^2*E)	O(V^3)	O(V^4)
general	Program. Dinámica		O(V^3*log(V))	O(V^3*log(V))	O(V^3*log(V))
general	Floyd-Warshall		O(V^3)	O(V^3)	O(V^3)

Transitive closure

Transitive closure

```
TC(G):
|     n = |V|
|     for i = 1…n
|     |     for i = 1…n
|     |     |     if (i==j) OR (i,j)∈E
|     |     |     |     t(0)ij=1
|     |     |     else
|     |     |     |     t(0)ij=0
|
|     for k = 1…n
|     |     for i = 1…n
|     |     |     for j = 1…n:
|     |     |     |     t(k)ij = t(k-1)ij OR (t(k-1)ik AND
t(k-1)kj )
```

Johnson

Johnson

```
JOHNSON(G,W):
|     V’  = V ∪ {s}
|     E’  = E ∪ {(s,v): v∈V}
|     w(s,v) = 0 ∀v∈V
|
|     B-F(G’, s)
|     si B-F == FALSO:
|     |     tiene ciclos neg.
|     si no:
|     |     for v∈V:
|     |     |     h(v) = δ(s,v)
|     |     for (u,v)∈E:
|     |     |     w’(u,v) = w(u,v)+h(u)-h(v)
|     |     for u∈V:
|     |     |     DIJKSTRA(G,w’,u)
|     |     |     for u∈V:
|     |     |     |     d(u,v) = δ’(u,v)+h(u)-h(v)
```

Algoritmos. Todos a todos.

	Algoritmo	*Complejidad (uno a todos)*	*Complejidad (todos a todos)*	*Ralo*	*Denso*
no pesado	BFS	O(V + E)	O(V*E)	O(V^2)	O(V^3)
no negativos	Dijkstra	O(V*log(V) + E)	O(V^2*log(V) + V*E)	O(V^2*log(V))	O(V^3)
general	Bellman-Ford	O(V*E)	O(V^2*E)	O(V^3)	O(V^4)
general	Program. Dinámica		O(V^3*log(V))	O(V^3*log(V))	O(V^3*log(V))
general	Floyd-Warshall		O(V^3)	O(V^3)	O(V^3)
general	Johnson’s		O(V^2*log(V) + V*E)	O(V^2*log(V))	O(V^3)