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

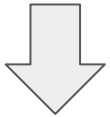
Problemas de camino mínimo con un único origen

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)

si es no pesado ⇒ BFS

si es no dirigido ⇒ Prim / Kruskal

si es pesado y dirigido ⇒ clase de hoy

Problemas de camino mínimo con un único origen

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,

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

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.

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

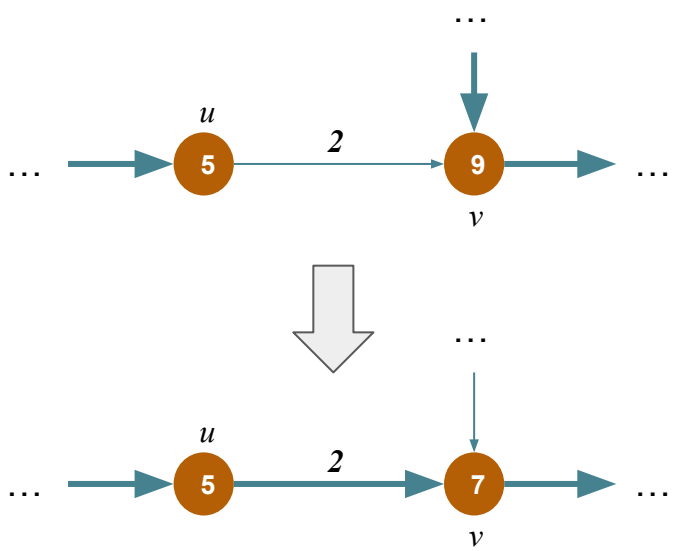
Input: $G = (V, E, w)$ y $s \in G$ un origen

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
```



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 u 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.

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)
```

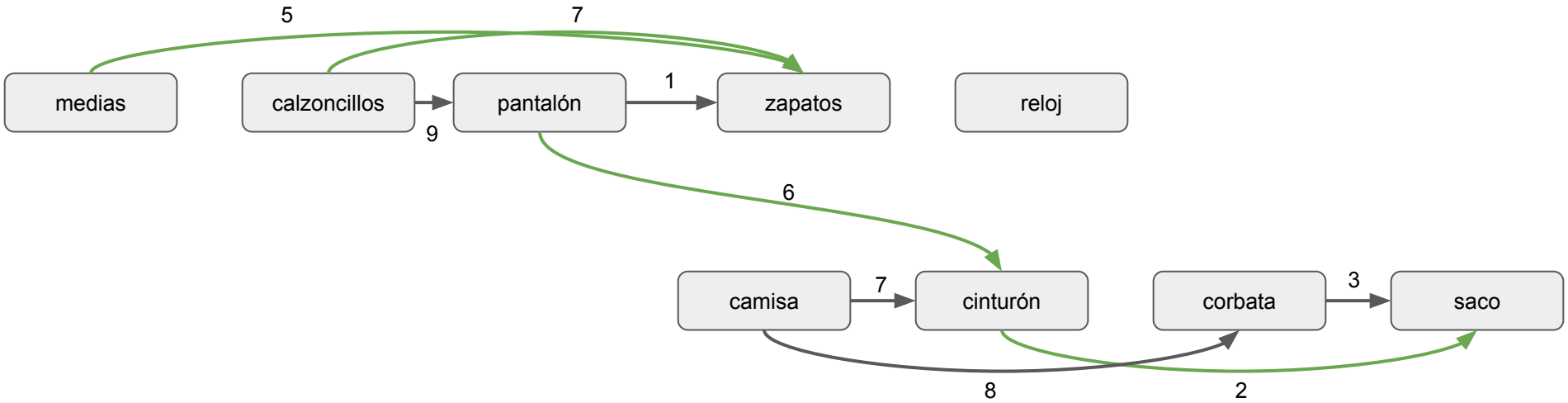
Demo:

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.

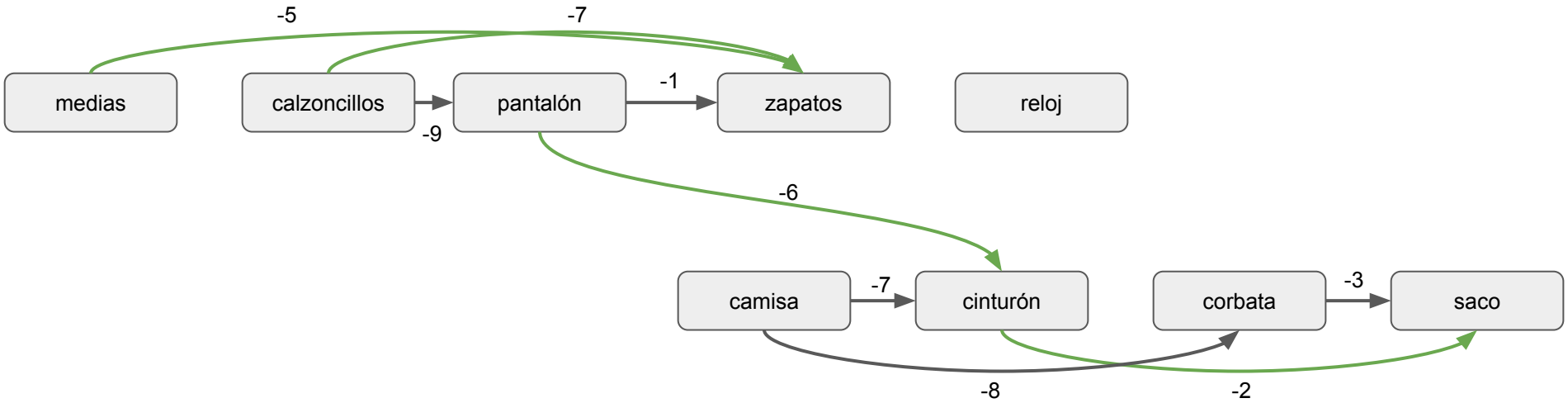
DAGs

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



DAGs

¿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
```

Dijkstra

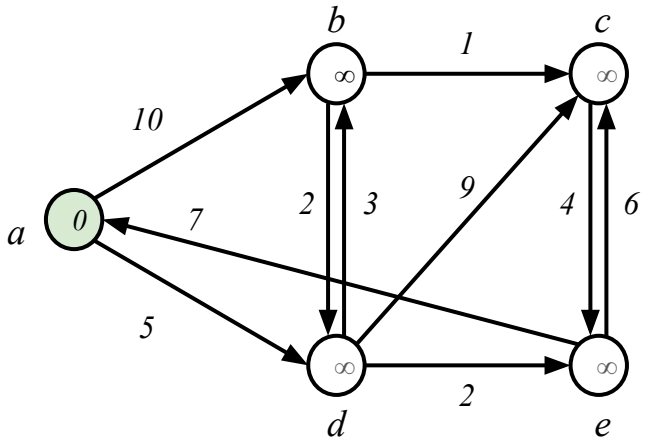
```
DIJKSTRA ( G, s ) :
|     INIT(G, s)
|     S = ∅
|     Q = ∅
|     for u in V:
|     |     INSERT(Q, u)
|     while Q:
|     |     u = EXTRACT-MIN()
|     |     S = S U {u}
|     |     for v in Adj[u]:
|     |     |     RELAX(u, v)
```

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

```
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: ∞}

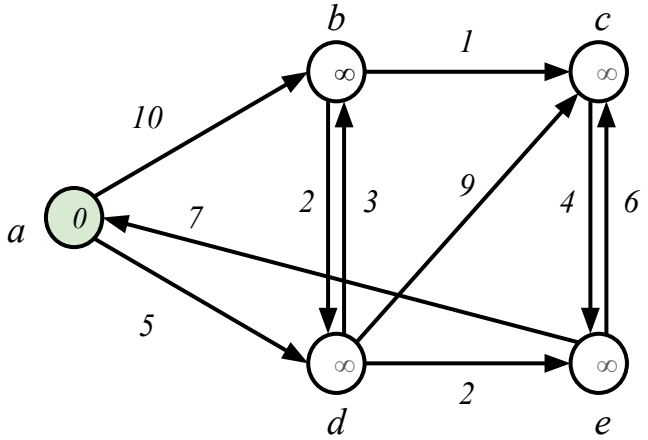
$pred$ = {a: None, b: None, c: None, d: None, e: None}

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: ∞}

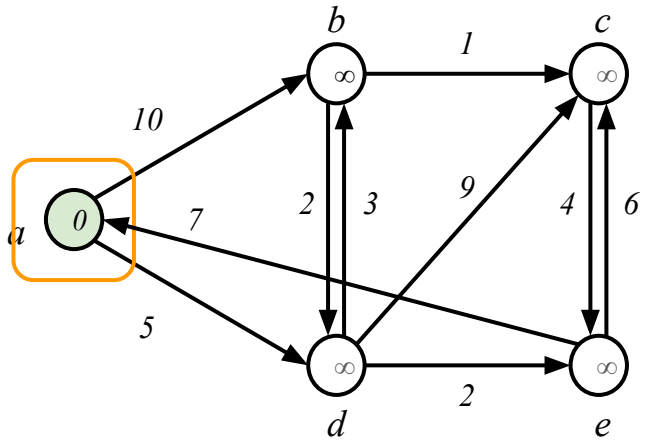
$pred$ = {a: None, b: None, c: None, d: None, e: None}

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}

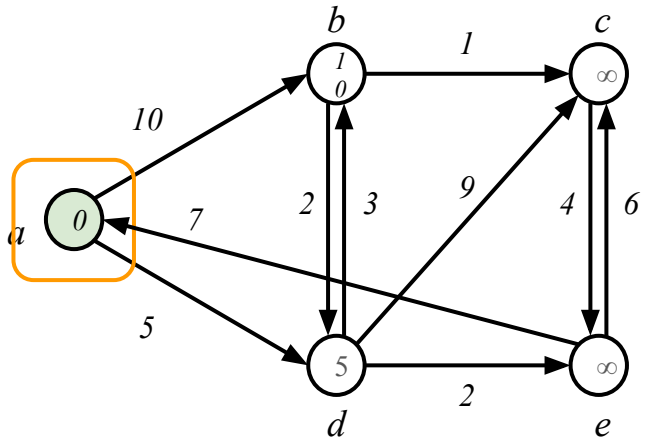
S = [a]

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]

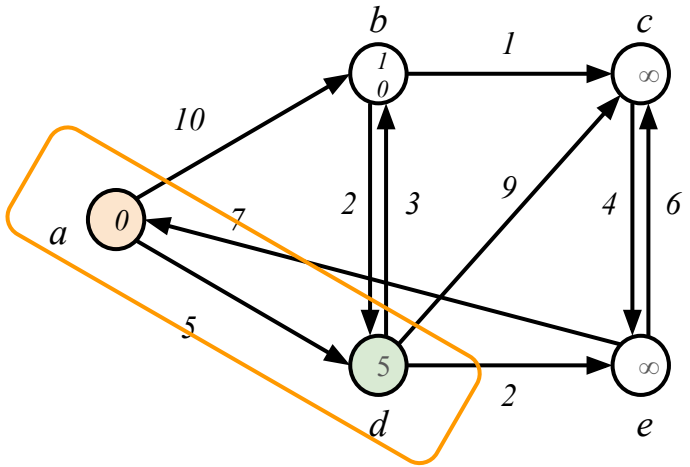
Q = [d, b, c, e]

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}

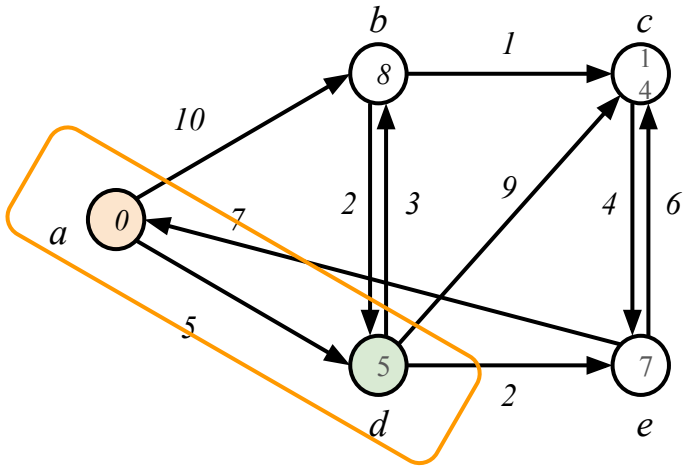
S = [a, d]

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}

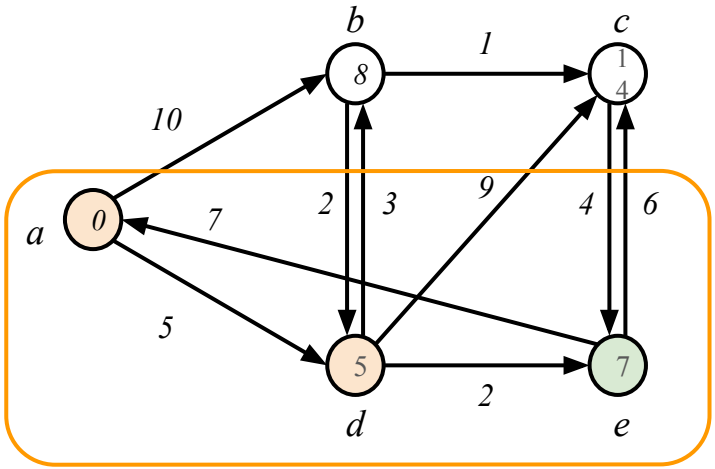
S = [a, d]

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}

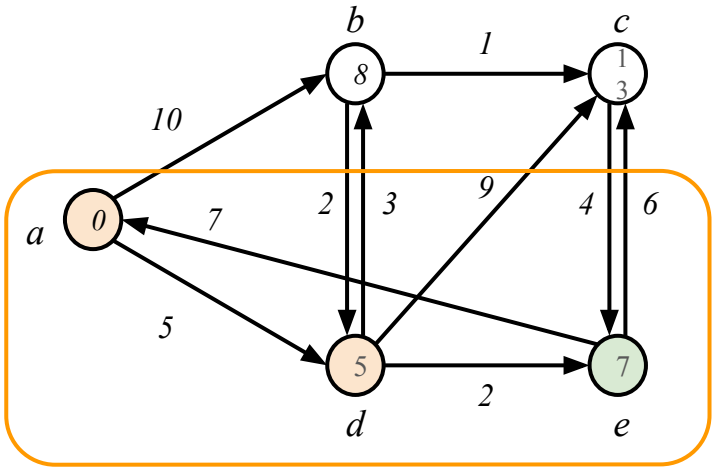
S = [a, d, e]

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]

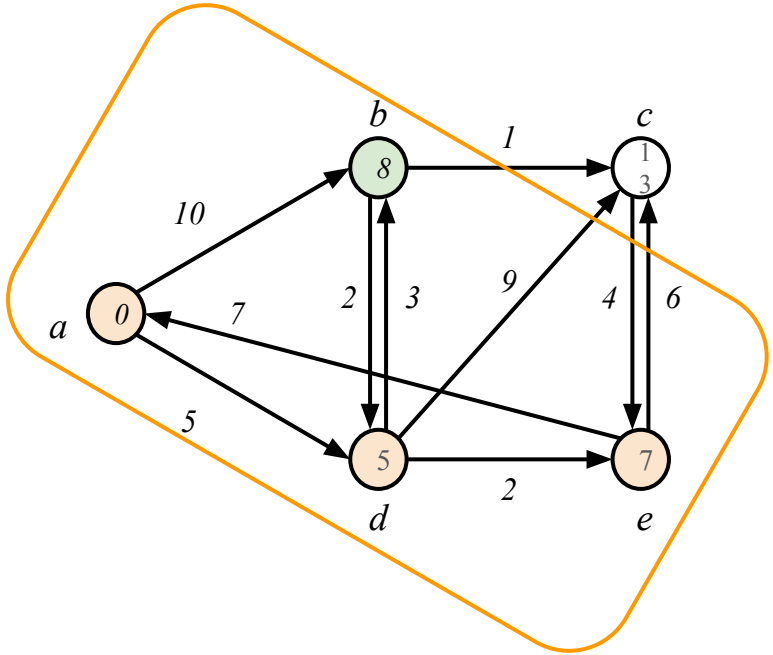
Q = [b, c]

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]

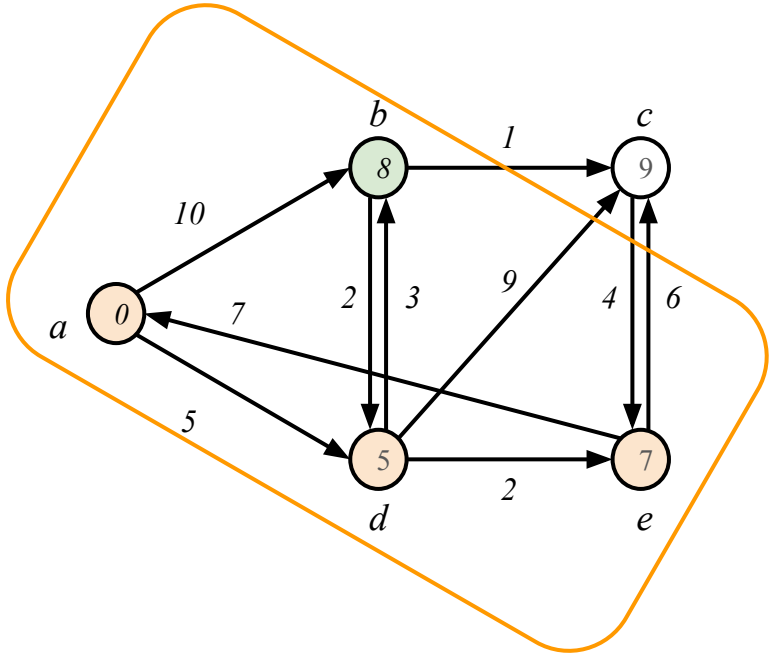
Q = [c]

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}

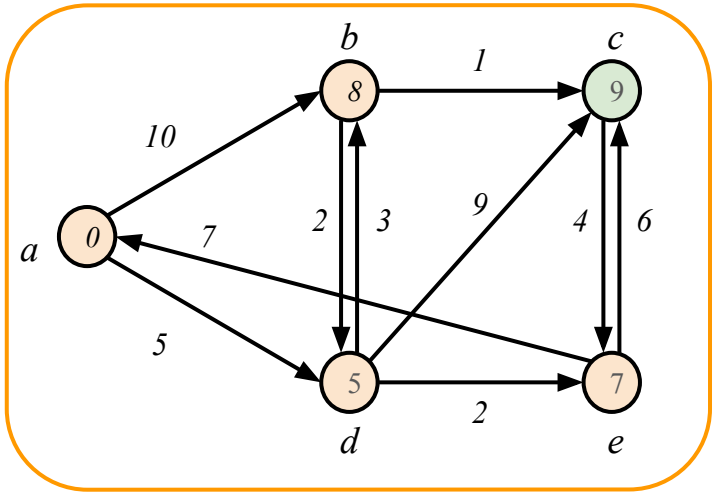
S = [a, d, e, b]

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}*

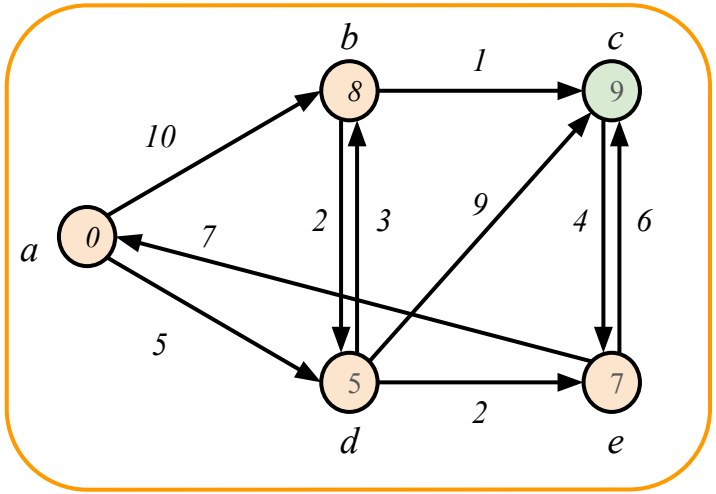
S = *[a, d, e, b, c]*

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]*

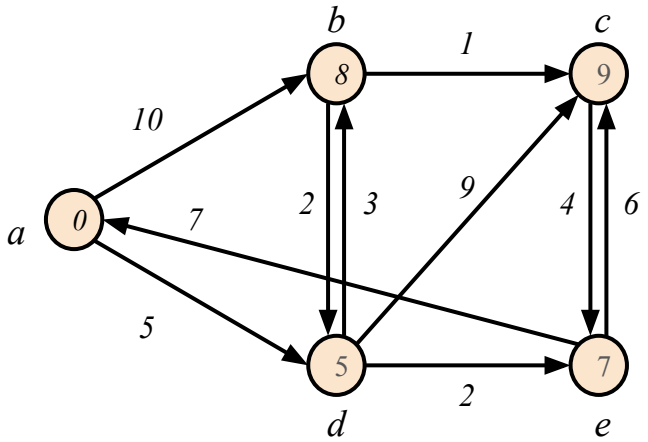
Q = *[]*

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}

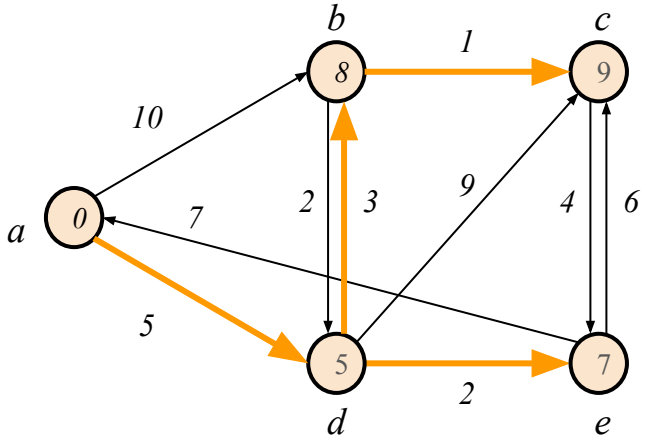
S = [a, d, e, b, c]

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]

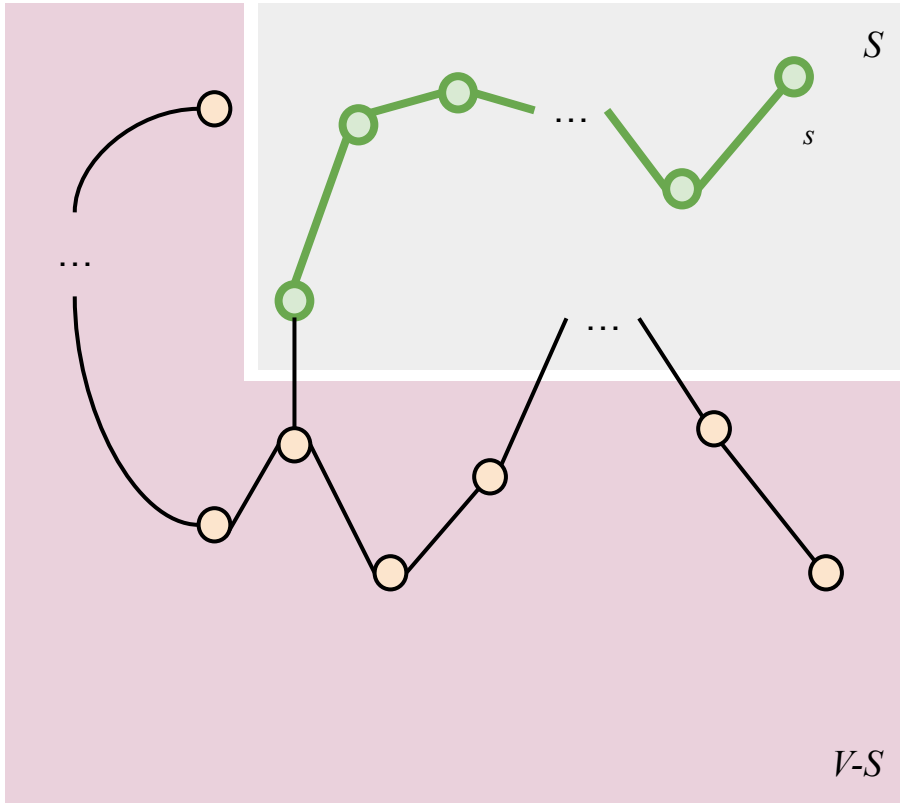
Q = []

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.

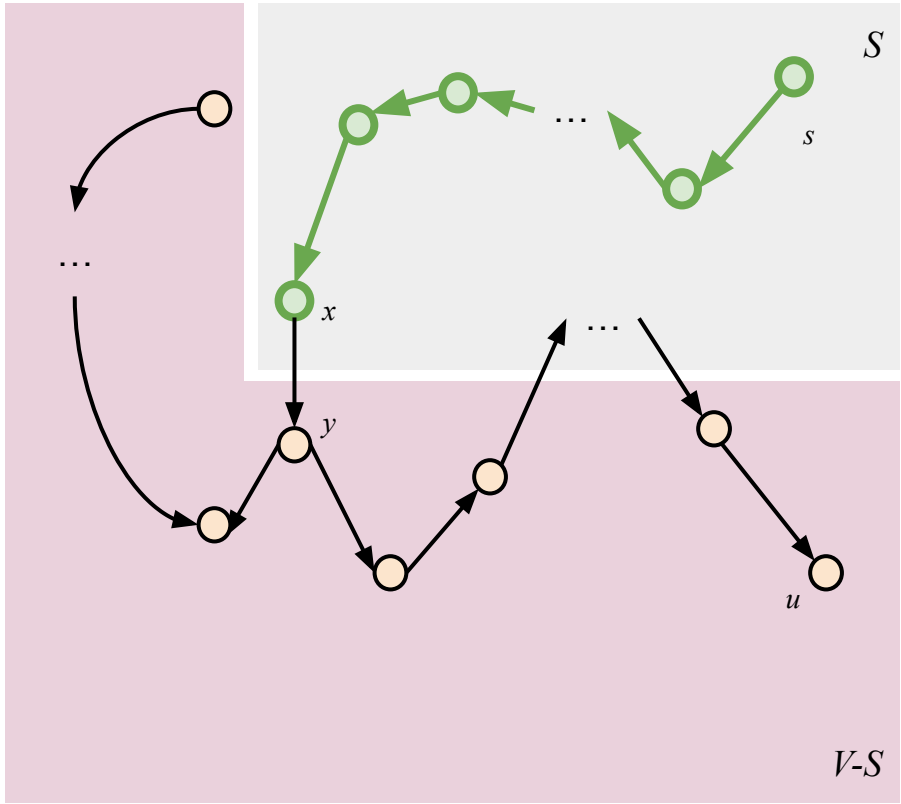
Como $w \geq 0 \Rightarrow \delta(s, y) \leq \delta(s, u)$

Si 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.

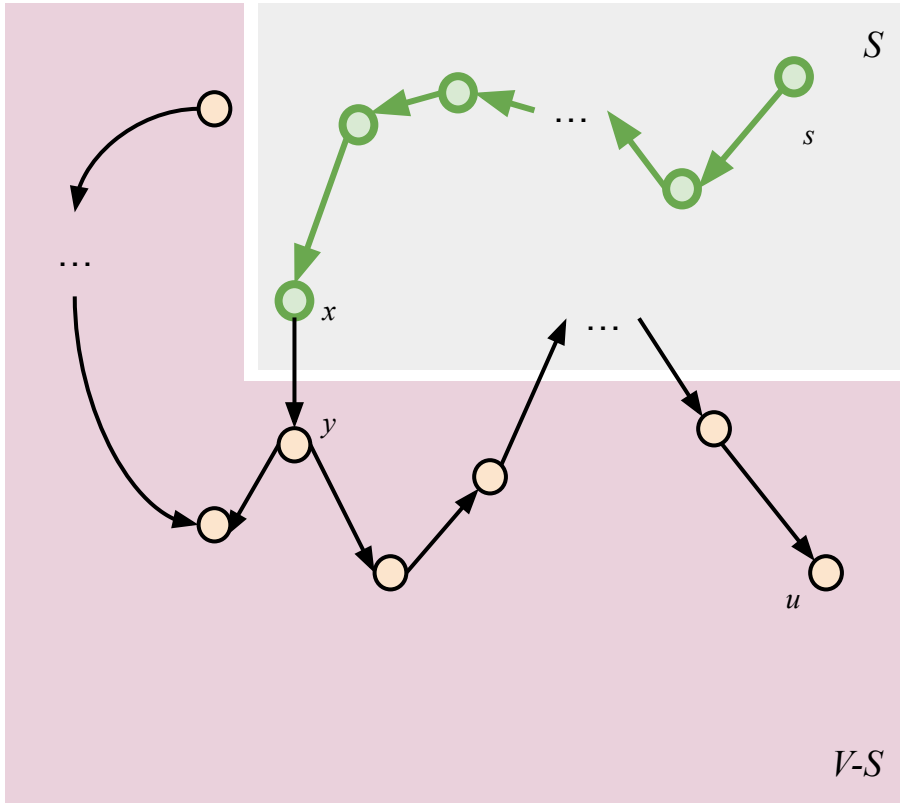
Como $w \geq 0 \Rightarrow \delta(s, y) \leq \delta(s, u)$

Si 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)
```

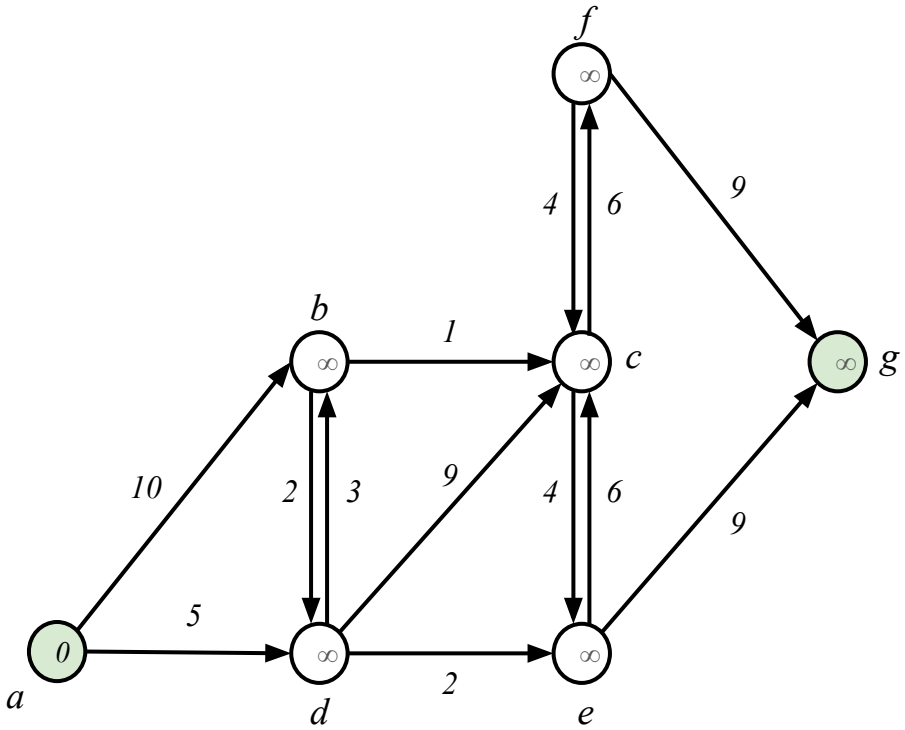
$O(V + V + V^2 + E) = O(V^2 + E) = O(V^2)$

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*

A*

~ Uno a uno

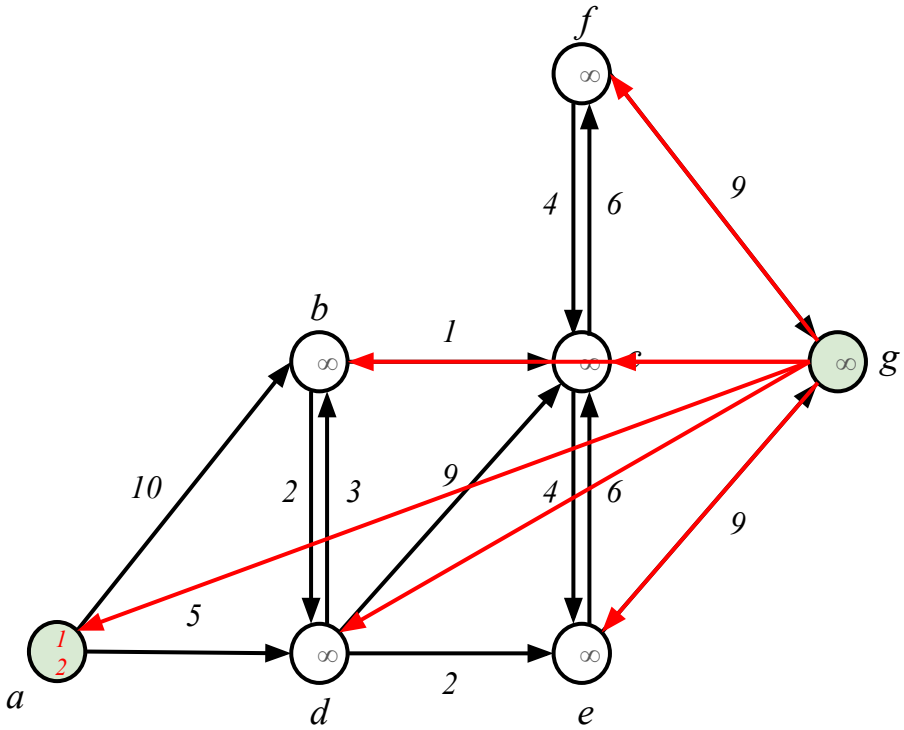


A*

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)

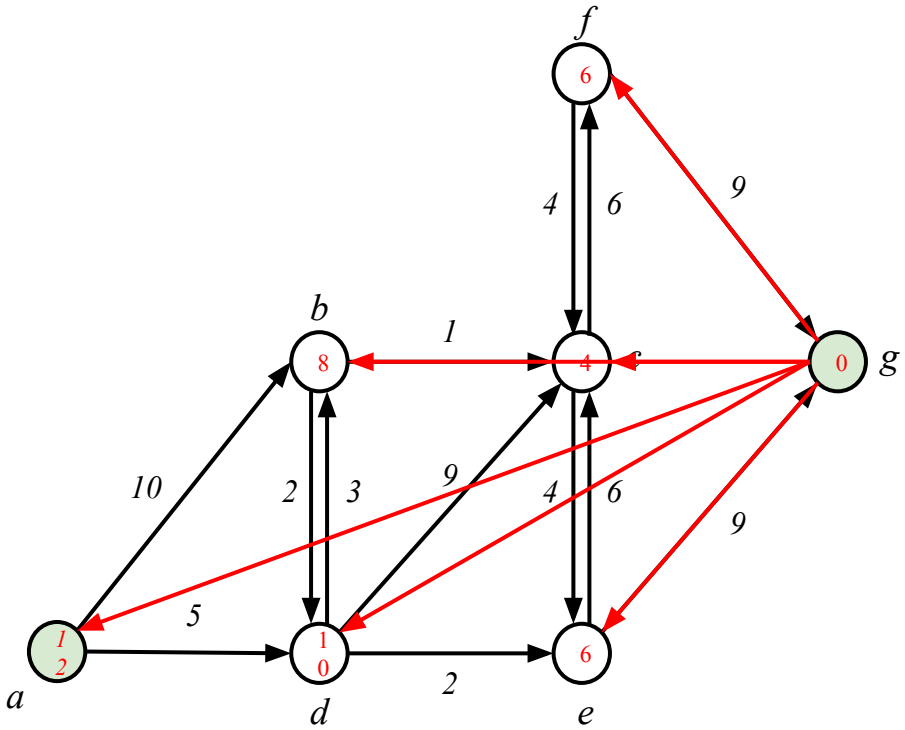


A*

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)

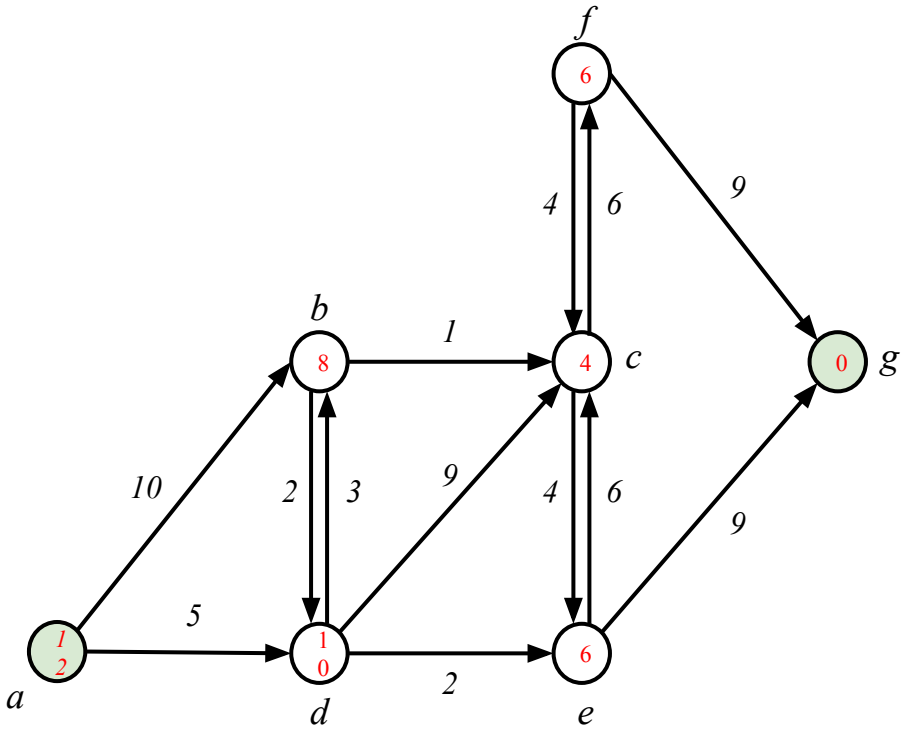


A*

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)

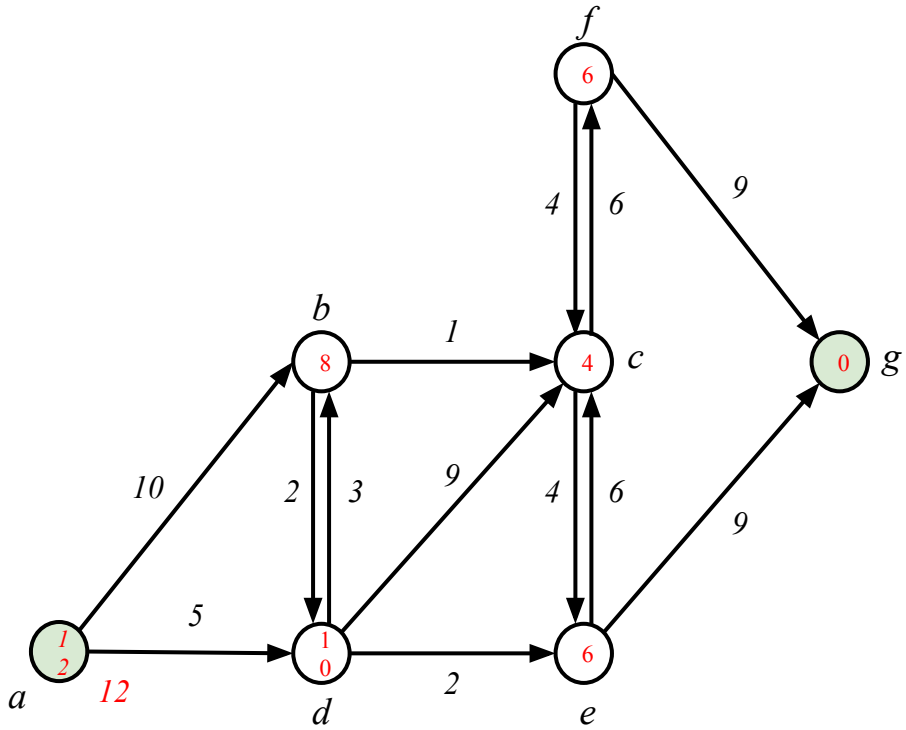


A*

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

A* score = heurística (h) + costo

a = 5

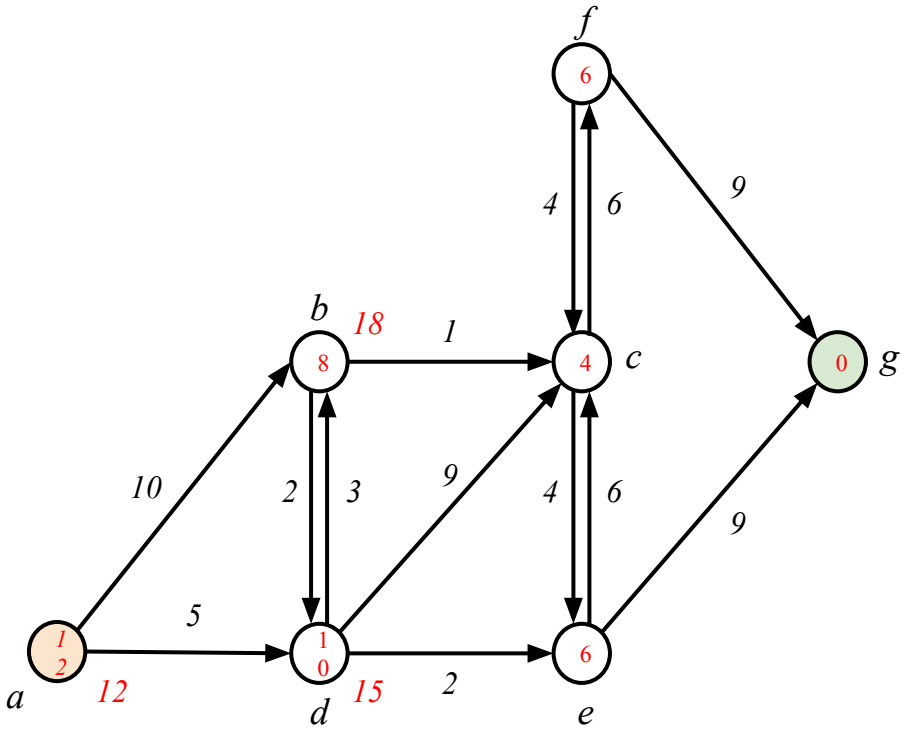


A*

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

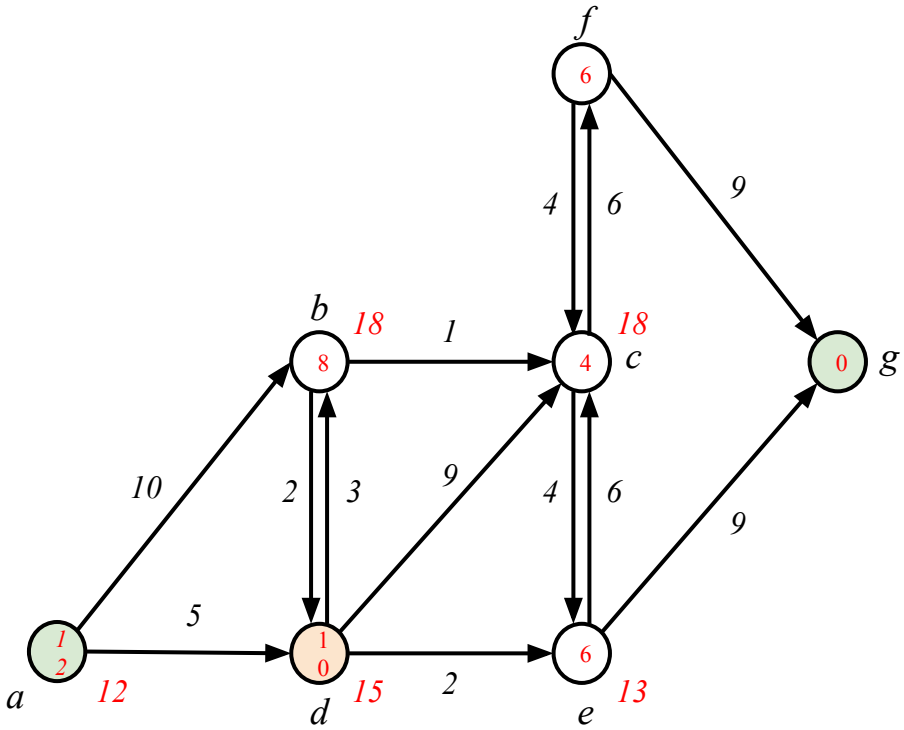


A*

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

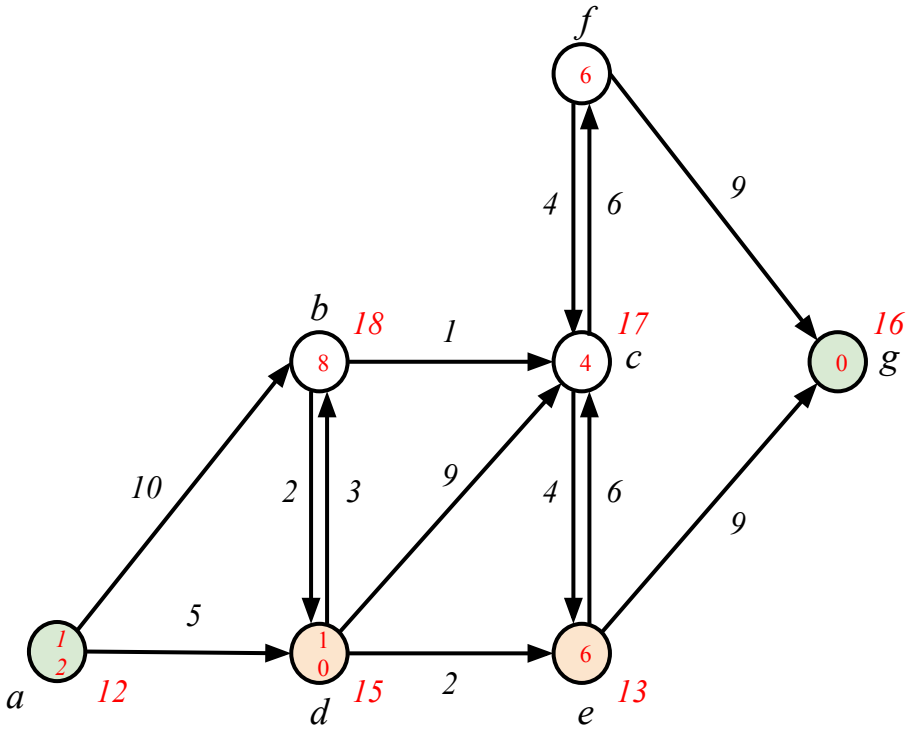


A*

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

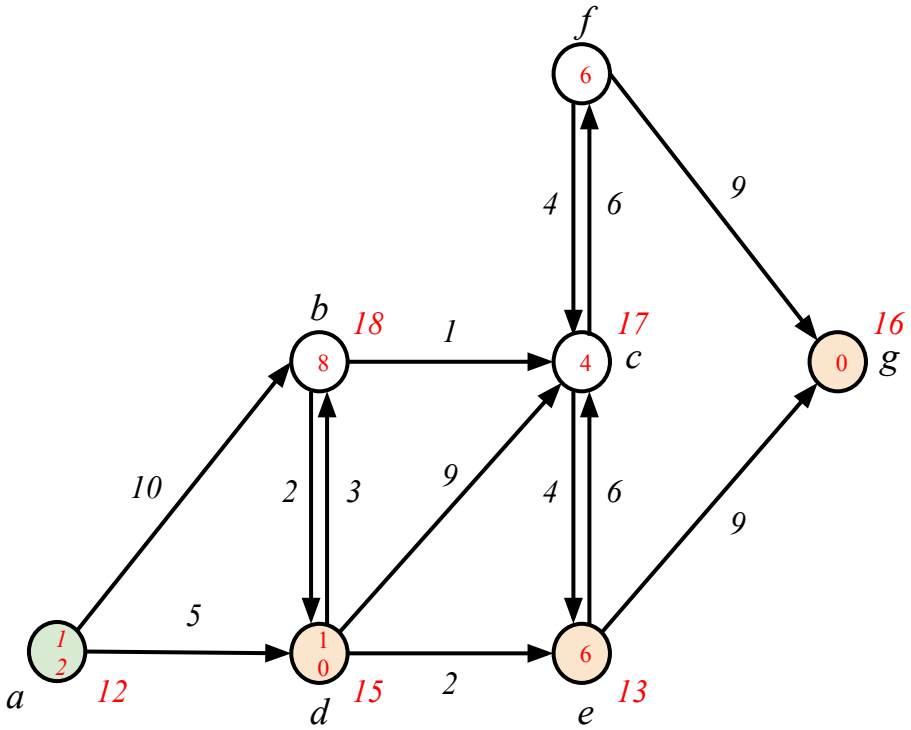


A*

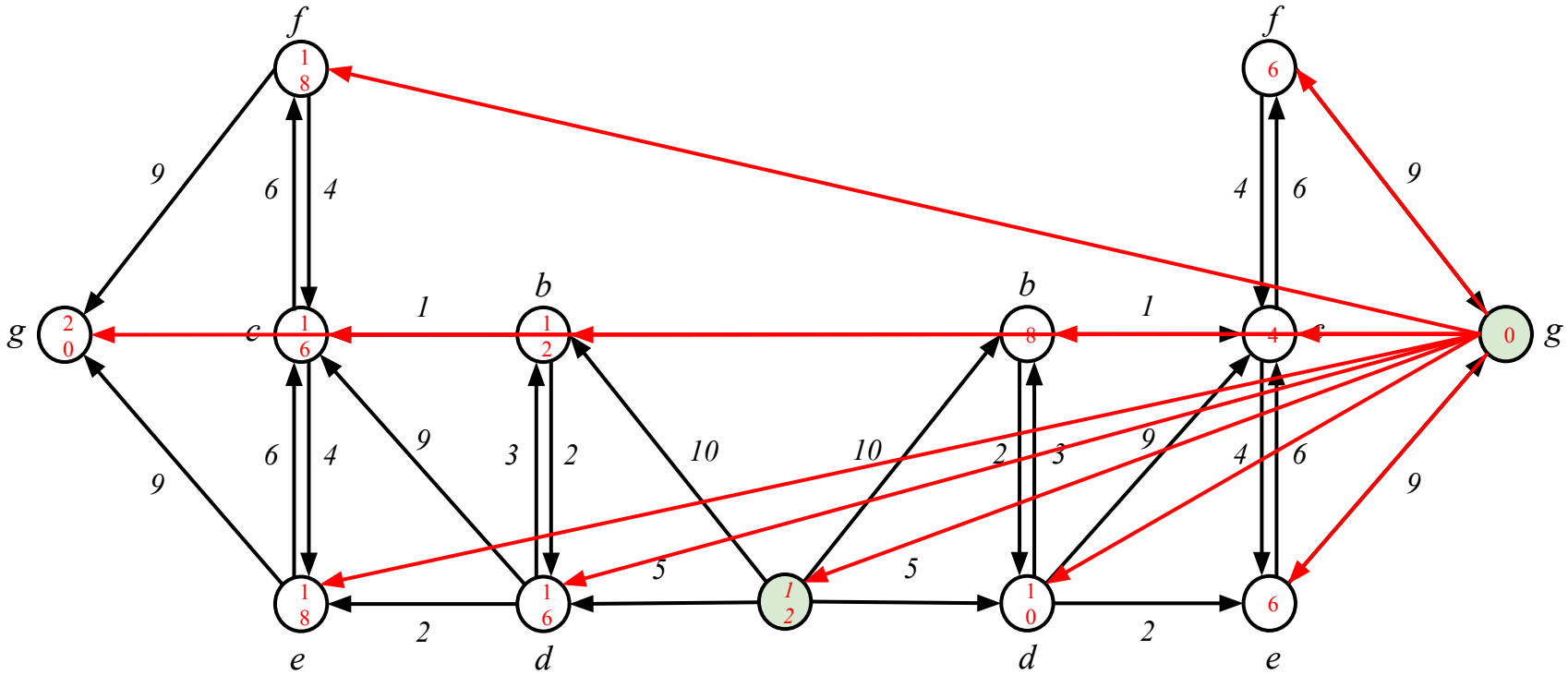
Inicializo los valores con una distancia estimada al objetivo (heurística). Es importante que subestime (ver más adelante).

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