Caminos más cortos en un DAG y el algoritmo de Dijkstra

Y Super Mario: un grafo de estados con costos

Carlos A Delgado S

Pontificia Universidad Javeriana Cali

Octubre de 2026

Contenido

1 Antes de empezar

2 Caminos más cortos en un DAG

3 El algoritmo de Dijkstra

4 UVa 10269: Adventure of Super Mario

5 Problemas de juez

6 Errores comunes

7 Ejercicios

8 Lo que sigue

9 Referencias

Contenido

1 Antes de empezar

2 Caminos más cortos en un DAG

3 El algoritmo de Dijkstra

4 UVa 10269: Adventure of Super Mario

5 Problemas de juez

6 Errores comunes

7 Ejercicios

8 Lo que sigue

9 Referencias

Objetivos de la sesión

Al terminar la clase, usted podrá

- Calcular los caminos más cortos desde una fuente en un grafo dirigido acíclico con una sola pasada en orden topológico.
- Seguir en una tabla el algoritmo de Dijkstra con cola de prioridad y decir qué hace con una pareja vieja.
- Explicar por qué Dijkstra exige pesos no negativos y dar el costo de cada algoritmo.

Y también

- Modelar UVa 10269 como un grafo de estados con costos: estado, transiciones, estado inicial y estados finales.
- Resolverlo con Dijkstra en lugar de la búsqueda en amplitud.

Lo que ya se tiene: Bellman-Ford-Moore

Relajar todas las aristas, muchas veces

Bellman-Ford hace $n - 1$ rondas y en cada una relaja todas las aristas: $O(nm)$. Lo hace así porque no sabe en qué orden conviene relajarlas, y una ronda más dice si hay un ciclo negativo alcanzable.

Relajar la arista (u, v) de peso w

Si `dist[u] + w < dist[v]`, entonces `dist[v] = dist[u] + w` y `pred[v] = u`.

La pregunta de hoy

¿Y si se supiera en qué orden relajar? ¿Bastaría con relajar cada arista una sola vez?

Contenido

1 Antes de empezar

2 Caminos más cortos en un DAG

3 El algoritmo de Dijkstra

4 UVa 10269: Adventure of Super Mario

5 Problemas de juez

6 Errores comunes

7 Ejercicios

8 Lo que sigue

9 Referencias

La idea, en palabras

Una sola pasada (CLRS, Sección 24.2)

Si el grafo no tiene ciclos, en lugar de relajar todas las aristas n veces se relajan una sola vez, procesando las aristas de cada vértice en orden topológico.

Por qué basta

- Se empieza con las aristas del primer vértice del orden y el efecto se propaga hacia adelante.
- Las aristas que faltan por relajar salen de vértices que están más adelante en el orden; ninguna llega a un vértice ya procesado.
- Por eso, cuando se revisan las aristas de u, el costo del camino más corto hasta u ya no puede mejorar.

La idea, en palabras

Teorema (CLRS, Teorema 24.5)

Si G es un grafo dirigido acíclico con pesos y fuente s, al terminar el algoritmo $dist[v] = \delta(s, v)$ para todo $v \in V$, y pred forma un árbol de caminos más cortos.

Los pesos negativos no estorban

Sin ciclos no hay ciclos negativos, así que los pesos pueden ser negativos.

El pseudocódigo

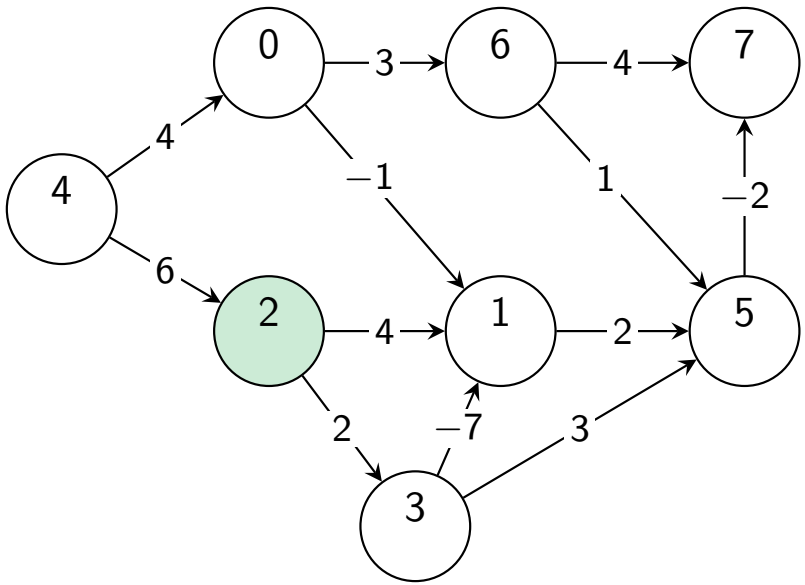
Caminos más cortos en un DAG

para cada v: $dist[v] \leftarrow \infty$; $pred[v] \leftarrow -1$
$dist[s] \leftarrow 0$
$topo \leftarrow$ orden topológico de G
para cada u en $topo$
    para cada arista (u, v) de peso w
        si $dist[u] + w < dist[v]$: $dist[v] \leftarrow dist[u] + w$; $pred[v] \leftarrow u$

Frente a Bellman-Ford

Es Bellman-Ford con una sola ronda, en la que las aristas se recorren en orden topológico de su origen.

El grafo del ejemplo



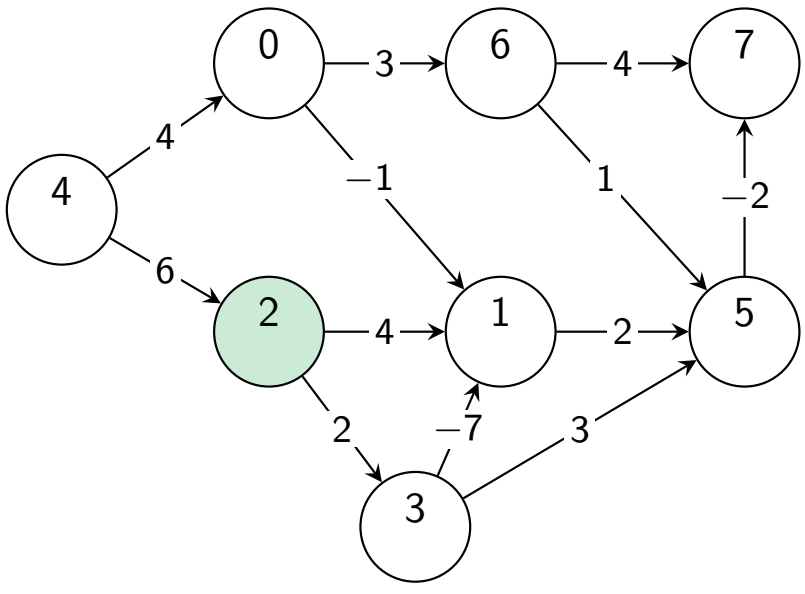
Datos

Ocho vértices, doce aristas, tres de ellas negativas. La fuente es $s = 2$.

Pregunta

¿Desde 2 se llega a 4, a 0 o a 6? ¿Qué distancia les quedará?

El grafo del ejemplo



Respuesta

No: a 4 no llega ninguna arista, y a 0 y a 6 solo se llega desde 4. Su distancia queda en ∞.

Las listas

`G[u]` es la lista de pares `(v, peso)`, ordenada por `v`. Por ejemplo, `G[2] = [(1, 4), (3, 2)]`.

El código completo

```python
from collections import deque

INF = float("inf")

# Kahn: G[u] tiene pares (v, peso)
def topoSort(G):
  q, topo, inc = deque(), [], [0 for _ in range(len(G))]
  # inc[v]: cuántas aristas llegan a v
  for u in range(len(G)):
    for (v, _) in G[u]:
      inc[v] += 1

  # sin aristas de entrada: van primero
  for u in range(len(G)):
    if inc[u] == 0:
      q.append(u)

  while len(q) != 0:
    u = q.popleft()
    topo.append(u)

    # quitar las aristas de u
    for (v, _) in G[u]:
      inc[v] -= 1
      if inc[v] == 0:
        q.append(v)
  return topo
```

```python
# Complejidad: n = |V|, m = |E| ---> O(n + m)
def ssspDAG(G, s):
  dist, pred = [INF for _ in range(len(G))], [-1 for _ in range(len(G))]
  dist[s] = 0

  topo = topoSort(G)
  # una pasada, en orden topológico
  for u in topo:
    for (v, duv) in G[u]:
      # relajar la arista (u, v)
      if dist[u] + duv < dist[v]:
        dist[v] = dist[u] + duv
        pred[v] = u
  return dist, pred
```

topoSort: cuántas aristas llegan a cada vértice

```
13 def topoSort(G):
14   q, topo, inc = deque(), [], [0 for _ in range(len(G))]
15   # inc[v]: cuántas aristas llegan a v
16   for u in range(len(G)):
17     for (v, _) in G[u]:
18       inc[v] += 1
19
20   # sin aristas de entrada: van primero
21   for u in range(len(G)):
22     if inc[u] == 0:
23       q.append(u)
```

Las líneas resaltadas

- 14: la cola q, la lista `topo` y el contador `inc`, en cero.
- 16 a 18: cada arista (u, v) suma uno a `inc[v]`.
- 21 a 23: los vértices sin aristas de entrada pueden ir primero y entran a la cola.

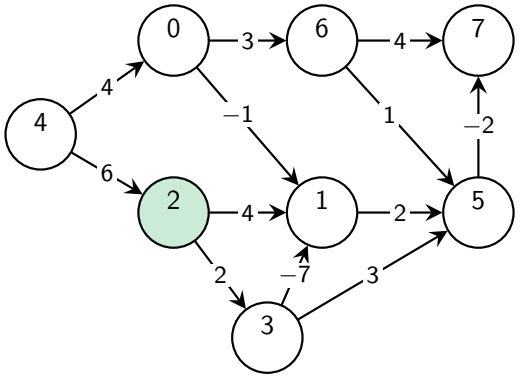
topoSort: sacar y quitar aristas

```
25 while len(q) != 0:
26   u = q.popleft()
27   topo.append(u)
28
29   # quitar las aristas de u
30   for (v, _) in G[u]:
31     inc[v] -= 1
32     if inc[v] == 0:
33       q.append(v)
34 return topo
```

Las líneas resaltadas

- 26 y 27: sale el primero de la cola y pasa al orden.
- 30 a 33: sus aristas se quitan; el vértice que se queda sin aristas de entrada entra a la cola. Cada vértice y cada arista se tocan una vez: $O(n+m)$.

topoSort sobre el grafo del ejemplo



Al empezar, las líneas 16 a 18 dejan

`inc = [1, 3, 1, 1, 0, 3, 1, 2]`.

Sale (26)	Entran (33)	q
	4 (línea 23)	[4]
4	0, 2	[0, 2]
0	6	[2, 6]
2	3	[6, 3]
6		[3]
3	1	[1]
1	5	[5]
5	7	[7]
7		[ ]

`topo = [4, 0, 2, 6, 3, 1, 5, 7]`

ssspDAG: la inicialización

```
37 def ssspDAG(G, s):
38   dist, pred = [INF for _ in range(len(G))], [-1 for _ in
     ↪  range(len(G))]
39   dist[s] = 0
40
41   topo = topoSort(G)
```

Las líneas resaltadas

- 38: todas las distancias en ∞ y todos los predecesores en -1.
- 39: la fuente queda a distancia 0.
- 41: el orden topológico, $O(n + m)$.

ssspDAG: la pasada

```
43 for u in topo:
44   for (v, duv) in G[u]:
45     # relajar la arista (u, v)
46     if dist[u] + duv < dist[v]:
47       dist[v] = dist[u] + duv
48       pred[v] = u
49 return dist, pred
```

Las líneas resaltadas

- 43 y 44: los vértices en orden topológico y, de cada uno, sus aristas. Cada arista se mira una vez.
- 46 a 48: la relajación de Bellman-Ford, sin cambios.

La pasada, vértice por vértice

u (43)	Arista (44)	Candidato (46)	`dist[v]`	Qué pasa
4	$(4,0)$, $(4,2)$	$\infty+4$, $\infty+6$		no mejora: `dist[4]` es ∞
0	$(0,1)$, $(0,6)$	$\infty-1$, $\infty+3$		no mejora

Los dos primeros del orden

4 y 0 van antes que la fuente y sus distancias son ∞. En la línea 46, ∞ más cualquier peso da ∞, y $\infty < \infty$ es falso.

La pasada, vértice por vértice

u (43)	Arista (44)	Candidato (46)	`dist[v]`	Qué pasa
4	$(4,0)$, $(4,2)$	$\infty+4$, $\infty+6$		no mejora: `dist[4]` es ∞
0	$(0,1)$, $(0,6)$	$\infty-1$, $\infty+3$		no mejora
2	$(2,1)$	$0+4=4$	∞	$dist[1] \leftarrow 4$, $pred[1] \leftarrow 2$
2	$(2,3)$	$0+2=2$	∞	$dist[3] \leftarrow 2$, $pred[3] \leftarrow 2$

La fuente

Las aristas de 2 dan los primeros valores finitos: 4 para el vértice 1 y 2 para el vértice 3.

La pasada, vértice por vértice

u (43)	Arista (44)	Candidato (46)	`dist[v]`	Qué pasa
4	$(4,0)$, $(4,2)$	$\infty+4$, $\infty+6$		no mejora: `dist[4]` es ∞
0	$(0,1)$, $(0,6)$	$\infty-1$, $\infty+3$		no mejora
2	$(2,1)$	$0+4=4$	∞	$dist[1] \leftarrow 4$, $pred[1] \leftarrow 2$
2	$(2,3)$	$0+2=2$	∞	$dist[3] \leftarrow 2$, $pred[3] \leftarrow 2$
6	$(6,5)$, $(6,7)$	$\infty+1$, $\infty+4$		no mejora
3	$(3,1)$	$2-7=-5$	4	$dist[1] \leftarrow -5$, $pred[1] \leftarrow 3$
3	$(3,5)$	$2+3=5$	∞	$dist[5] \leftarrow 5$, $pred[5] \leftarrow 3$

Pregunta

$dist[1]$ pasó de 4 a -5. Cuando le llegue el turno a 1, ¿puede volver a cambiar? ¿Qué vértices quedan antes de 1 en el orden?

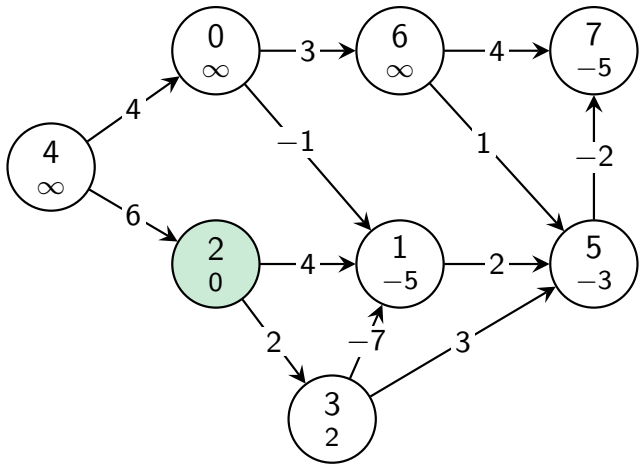
La pasada, vértice por vértice

u (43)	Arista (44)	Candidato (46)	`dist[v]`	Qué pasa
4	$(4,0)$, $(4,2)$	$\infty+4$, $\infty+6$		no mejora: `dist[4]` es ∞
0	$(0,1)$, $(0,6)$	$\infty-1$, $\infty+3$		no mejora
2	$(2,1)$	$0+4=4$	∞	$dist[1] \leftarrow 4$, $pred[1] \leftarrow 2$
2	$(2,3)$	$0+2=2$	∞	$dist[3] \leftarrow 2$, $pred[3] \leftarrow 2$
6	$(6,5)$, $(6,7)$	$\infty+1$, $\infty+4$		no mejora
3	$(3,1)$	$2-7=-5$	4	$dist[1] \leftarrow -5$, $pred[1] \leftarrow 3$
3	$(3,5)$	$2+3=5$	∞	$dist[5] \leftarrow 5$, $pred[5] \leftarrow 3$
1	$(1,5)$	$-5+2=-3$	5	$dist[5] \leftarrow -3$, $pred[5] \leftarrow 1$
5	$(5,7)$	$-3-2=-5$	∞	$dist[7] \leftarrow -5$, $pred[7] \leftarrow 5$
7	no tiene aristas de salida			

Respuesta

No. Las tres aristas que llegan a 1 salen de 0, 2 y 3, que van antes en el orden y ya se procesaron. Cuando sale 1, -5 es su distancia final, y por eso $(1,5)$ se relaja con el valor correcto.

Lo que deja la pasada



	0	1	2	3	4	5	6	7
`dist`	∞	-5	0	2	∞	-3	∞	-5
`pred`	-1	3	-1	2	-1	1	-1	5

Lo que deja la pasada

El camino hasta 7

Se sube por `pred` desde 7: $7 \leftarrow 5 \leftarrow 1 \leftarrow 3 \leftarrow 2$. El camino es $\langle 2, 3, 1, 5, 7 \rangle$, de peso $2 - 7 + 2 - 2 = -5$.

Pregunta

En las anotaciones aparece otro orden topológico del mismo grafo: $4, 0, 6, 2, 3, 1, 5, 7$. ¿Cambia alguna distancia si la pasada usa ese orden?

Respuesta: cualquier orden topológico sirve

Respuesta

No cambia nada. El argumento solo usa que las aristas que llegan a u salen de vértices que van antes en el orden, y eso lo cumple todo orden topológico. Entre 6 y 2 no hay camino, así que da igual cuál vaya primero.

El orden sí tiene que ser topológico

Si 3 fuera después de 1, la arista $(1, 5)$ se relajaría con $dist[1] = 4$ y $dist[5]$ terminaría en 5, no en -3.

Lo que cuesta

Tiempo

`topoSort` cuesta $O(n+m)$; la inicialización, $O(n)$; la pasada mira cada arista una vez, $O(n+m)$. Total: $O(n+m)$, contra $O(nm)$ de Bellman-Ford.

Espacio

$O(n)$ para `dist`, `pred`, `inc`, la cola y el orden, además de las listas de adyacencia.

Pregunta

¿Qué devuelve `topoSort` si el grafo tiene un ciclo? ¿Cómo se nota en `topo`?

Respuesta: un ciclo deja vértices por fuera

Respuesta

Los vértices del ciclo nunca llegan a `inc` igual a cero, así que no entran a la cola. `topo` queda con menos de n vértices, y `ssspDAG` no relaja las aristas que salen de los que faltan.

Una revisión barata

Si `len(topo) != len(G)`, el grafo tiene un ciclo y hay que usar otro algoritmo.

Contenido

1 Antes de empezar

2 Caminos más cortos en un DAG

3 El algoritmo de Dijkstra

4 UVa 10269: Adventure of Super Mario

5 Problemas de juez

6 Errores comunes

7 Ejercicios

8 Lo que sigue

9 Referencias

Si hay ciclos, ¿qué vértice va primero?

El problema

Con ciclos no hay orden topológico. Hace falta otra forma de saber cuándo la distancia de un vértice ya no puede mejorar.

La idea de Dijkstra (CLRS, Sección 24.3)

Si ningún peso es negativo, el vértice pendiente con la menor distancia conocida ya tiene su distancia final: cualquier otro camino hasta él pasa por un vértice pendiente que ya está igual de lejos o más, y lo que falta no resta.

Si hay ciclos, ¿qué vértice va primero?

Algoritmo de Dijkstra para caminos más cortos desde una fuente

- Solo se aplica a grafos con pesos que no tengan pesos negativos.
- Usa una cola de prioridad: sale primero el vértice con la menor distancia.
- Complejidad: $O((n + m) \log n)$.

El pseudocódigo: primera versión

Dijkstra con todos los vértices en la cola

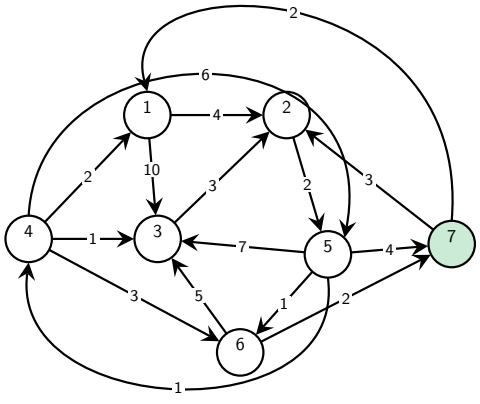
para cada v: $dist[v] \leftarrow \infty$; $pred[v] \leftarrow -1$
$dist[s] \leftarrow 0$
$Q \leftarrow$ todos los vértices, con prioridad $dist$
mientras Q no esté vacía
    $u \leftarrow$ el de menor $dist$ en Q; sacarlo de Q
    para cada arista (u, v) de peso w
        si $dist[u] + w < dist[v]$: $dist[v] \leftarrow dist[u] + w$; $pred[v] \leftarrow u$

Frente al DAG

El orden en que se procesan los vértices ya no se calcula antes: lo decide la cola, por distancia.

La primera versión sobre un grafo con ciclos

Sale	1	2	3	4	5	6	7	Qué cambia
	∞	∞	∞	∞	∞	∞	0	inicio
7 (0)	2	3	∞	∞	∞	∞	0	$pred[1] = pred[2] = 7$

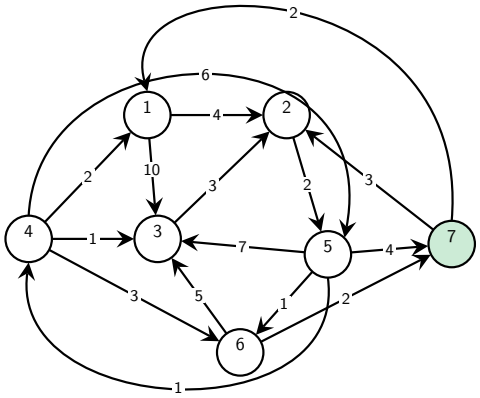


Sale la fuente

7 es el único con distancia finita. Sus aristas dan 2 al vértice 1 y 3 al vértice 2.

La primera versión sobre un grafo con ciclos

Sale	1	2	3	4	5	6	7	Qué cambia
	∞	∞	∞	∞	∞	∞	0	inicio
7 (0)	2	3	∞	∞	∞	∞	0	$pred[1] = pred[2] = 7$
1 (2)	2	3	12	∞	∞	∞	0	$pred[3] = 1$
2 (3)	2	3	12	∞	5	∞	0	$pred[5] = 2$

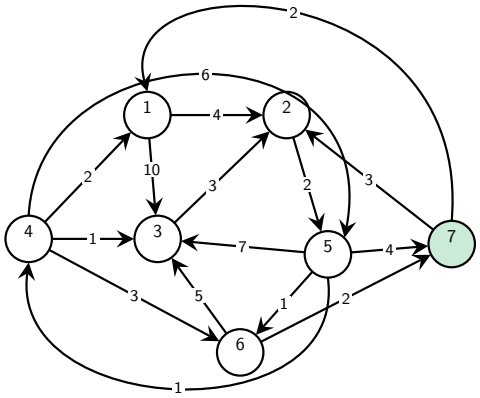


Pregunta

Quedan en la cola 3 con 12 y 5 con 5; los demás, con ∞. ¿Cuál sale, y qué vértices cambian?

La primera versión sobre un grafo con ciclos

Sale	1	2	3	4	5	6	7	Qué cambia
	∞	∞	∞	∞	∞	∞	0	inicio
7 (0)	2	3	∞	∞	∞	∞	0	$pred[1] = pred[2] = 7$
1 (2)	2	3	12	∞	∞	∞	0	$pred[3] = 1$
2 (3)	2	3	12	∞	5	∞	0	$pred[5] = 2$
5 (5)	2	3	12	6	5	6	0	$pred[4] = pred[6] = 5$

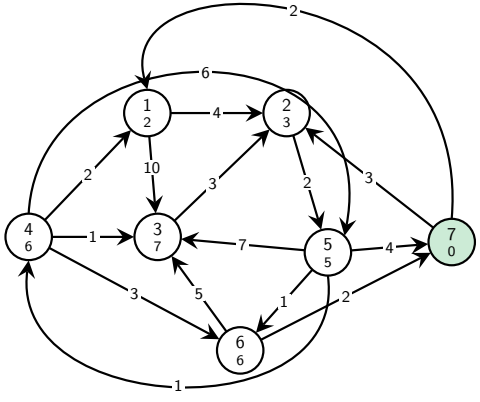


Respuesta

Sale 5. Sus aristas dan 6 a 4 y a 6. La arista $(5, 3)$ da $5 + 7 = 12$, que no es menor que 12; $(5, 7)$ llega a la fuente, que ya salió.

La primera versión sobre un grafo con ciclos

Sale	1	2	3	4	5	6	7	Qué cambia
	∞	∞	∞	∞	∞	∞	0	inicio
7 (0)	2	3	∞	∞	∞	∞	0	$pred[1] = pred[2] = 7$
1 (2)	2	3	12	∞	∞	∞	0	$pred[3] = 1$
2 (3)	2	3	12	∞	5	∞	0	$pred[5] = 2$
5 (5)	2	3	12	6	5	6	0	$pred[4] = pred[6] = 5$
6 (6)	2	3	11	6	5	6	0	$pred[3] = 6$
4 (6)	2	3	7	6	5	6	0	$pred[3] = 4$
3 (7)	2	3	7	6	5	6	0	nada: $7 + 3 > 3$



Un empate

4 y 6 salen con 6. Aquí sale primero 6 y $dist[3]$ pasa por 11 antes de llegar a 7; con 4 primero, pasa de 12 a 7. Las distancias finales son las mismas.

La segunda versión: la que se implementa

Lo que no tiene `heapq`

La primera versión baja la prioridad de un vértice que ya está en la cola. `heapq` no sabe hacerlo: solo mete y saca.

La salida

- En la cola van parejas `(distancia, vértice)`, y al principio solo la de la fuente.
- Cuando una distancia mejora, se mete otra pareja con el valor nuevo. La de antes se queda en la cola.
- Al sacar una pareja se compara con `dist`: si no coincide, es una pareja vieja y se descarta.

El código completo

```python
from heapq import heappush,heappop

INF = float('inf')

# Implementación algoritmo de Dijkstra para la versión uno a uno
def dijkstraST(G, s, t):
  dist = [INF for _ in range(len(G))] ; dist[s] = 0
  pqueue, found = list(), False
  #for u in range(len(G)): heappush(pqueue, (dist[u], u))
  heappush(pqueue, (dist[s], s))

  while len(pqueue)!=0 and not found:
    du,u = heappop(pqueue)
    # al salir t, su distancia es final
    if u == t:
      found = True
    else:
      # pareja vieja: se descarta
      if dist[u] == du:
        for v,duv in G[u]:
          if du+duv<dist[v]:
            dist[v] = du+duv
            heappush(pqueue, (dist[v], v))
  return dist[t]
```

```python
# Implementación algoritmo de Dijkstra para la versión uno a todos
# Complejidad: T(n, m) E O((n + m) * log n)
def dijkstra(G, s):
  dist = [INF for _ in range(len(G))] ; dist[s] = 0
  pred = [-1 for _ in range(len(G))]
  pqueue = list()
  #for u in range(len(G)): heappush(pqueue, (dist[u], u))
  heappush(pqueue, (dist[s], s))

  while len(pqueue)!=0:
    # la pareja de menor distancia
    du,u = heappop(pqueue)
    # si no coincide, la pareja es vieja
    if dist[u] == du:
      for v,duv in G[u]:
        # si mejora, v entra otra vez
        if du+duv<dist[v]:
          dist[v] = du+duv
          pred[v] = u
          heappush(pqueue, (dist[v], v))
  return dist
```

dijkstra: la inicialización

```
34 def dijkstra(G, s):
35   dist = [INF for _ in range(len(G))] ; dist[s] = 0
36   pred = [-1 for _ in range(len(G))]
37   pqueue = list()
38   #for u in range(len(G)): heappush(pqueue, (dist[u], u))
39   heappush(pqueue, (dist[s], s))
```

Las líneas resaltadas

- 35 y 36: distancias en ∞, la de la fuente en 0, y predecesores en -1.
- 37 y 39: la cola es una lista que `heappush` y `heappop` manejan como montículo. Empieza con la pareja `(0, s)`.
- La línea 38, comentada, es la primera versión: todos los vértices en la cola desde el principio.

dijkstra: sacar la pareja menor

```
41 while len(pqueue)!=0:
42   # la pareja de menor distancia
43   du,u = heappop(pqueue)
44   # si no coincide, la pareja es vieja
45   if dist[u] == du:
```

Las líneas resaltadas

- 41: el ciclo sigue mientras haya parejas.
- 43: `heappop` saca la pareja de menor distancia; las tuplas se comparan primero por la distancia.
- 45: si `dist[u]` ya no es `du`, la distancia de u mejoró después de meter esta pareja. La pareja es vieja y no se procesa.

dijkstra: la relajación

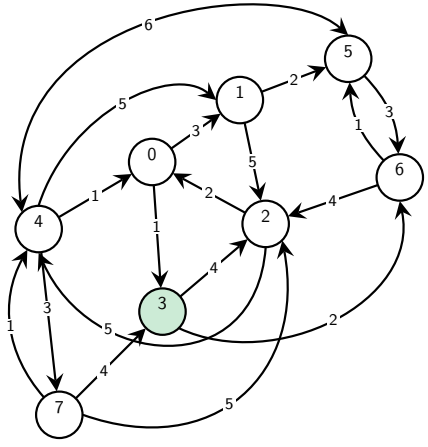
```
46      for v,duv in G[u]:
47          # si mejora, v entra otra vez
48          if du+duv<dist[v]:
49              dist[v] = du+duv
50              pred[v] = u
51              heappush(pqueue, (dist[v], v))
52  return dist
```

Las líneas resaltadas

- 46: las aristas de u, como pares `(v, duv)`.
- 48 a 50: la relajación, con `du` en lugar de `dist[u]`; en este punto valen lo mismo.
- 51: v entra a la cola con su distancia nueva, aunque ya tenga ahí una pareja con la distancia anterior.

La segunda versión sobre el grafo de las anotaciones

Sale (43)	Cambia (48–51)	pqueue después
$(0, 3)$	$dist[2] \leftarrow 4$, $dist[6] \leftarrow 2$, $pred = 3$	$(2, 6)$, $(4, 2)$

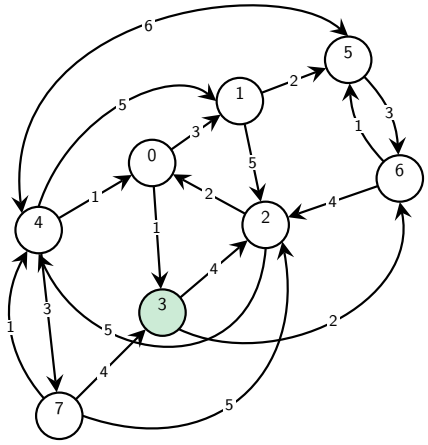


Sale la fuente

`dijkstra(G, 3)`. La línea 39 mete $(0, 3)$; al sacarla, las aristas $(3, 2)$ y $(3, 6)$ meten dos parejas.

La segunda versión sobre el grafo de las anotaciones

Sale (43)	Cambia (48–51)	pqueue después
$(0, 3)$	$dist[2] \leftarrow 4$, $dist[6] \leftarrow 2$, $pred = 3$	$(2, 6)$, $(4, 2)$
$(2, 6)$	$dist[5] \leftarrow 3$, $pred[5] = 6$; $(6, 2)$ da $6 > 4$	$(3, 5)$, $(4, 2)$
$(3, 5)$	$dist[4] \leftarrow 9$, $pred[4] = 5$	$(4, 2)$, $(9, 4)$

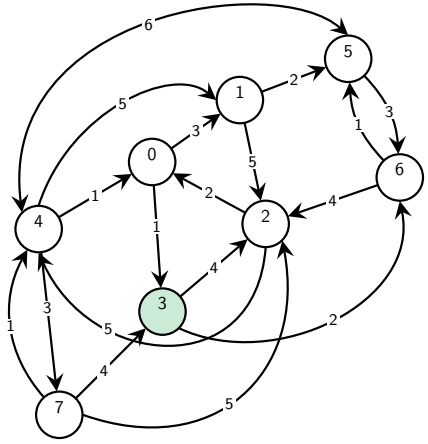


Pregunta

Ahora sale $(4, 2)$. Las aristas de 2 van a 0, con peso 2, y a 4, con peso 5. ¿Qué parejas entran a la cola?

La segunda versión sobre el grafo de las anotaciones

Sale (43)	Cambia (48–51)	pqueue después
$(0, 3)$	$dist[2] \leftarrow 4$, $dist[6] \leftarrow 2$, $pred = 3$	$(2, 6)$, $(4, 2)$
$(2, 6)$	$dist[5] \leftarrow 3$, $pred[5] = 6$; $(6, 2)$ da $6 > 4$	$(3, 5)$, $(4, 2)$
$(3, 5)$	$dist[4] \leftarrow 9$, $pred[4] = 5$	$(4, 2)$, $(9, 4)$
$(4, 2)$	$dist[0] \leftarrow 6$, $pred[0] = 2$; $(2, 4)$ da 9, no mejora	$(6, 0)$, $(9, 4)$
$(6, 0)$	$dist[1] \leftarrow 9$, $pred[1] = 0$	$(9, 1)$, $(9, 4)$

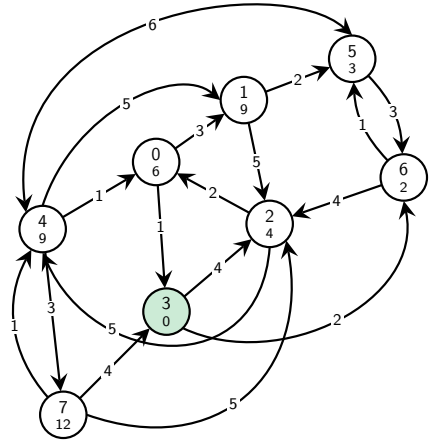


Respuesta

Solo $(6, 0)$. Por $(2, 4)$ el candidato es $4 + 5 = 9$, igual a `dist[4]`: la línea 48 pide menor estricto, así que $pred[4]$ sigue en 5.

La segunda versión sobre el grafo de las anotaciones

Sale (43)	Cambia (48–51)	pqueue después
$(0, 3)$	$dist[2] \leftarrow 4$, $dist[6] \leftarrow 2$, $pred = 3$	$(2, 6)$, $(4, 2)$
$(2, 6)$	$dist[5] \leftarrow 3$, $pred[5] = 6$; $(6, 2)$ da $6 > 4$	$(3, 5)$, $(4, 2)$
$(3, 5)$	$dist[4] \leftarrow 9$, $pred[4] = 5$	$(4, 2)$, $(9, 4)$
$(4, 2)$	$dist[0] \leftarrow 6$, $pred[0] = 2$; $(2, 4)$ da 9, no mejora	$(6, 0)$, $(9, 4)$
$(6, 0)$	$dist[1] \leftarrow 9$, $pred[1] = 0$	$(9, 1)$, $(9, 4)$
$(9, 1)$	nada: $14 > 4$ y $11 > 3$	$(9, 4)$
$(9, 4)$	$dist[7] \leftarrow 12$, $pred[7] = 4$	$(12, 7)$
$(12, 7)$	nada	vacía



Lo que queda

$dist = (6, 9, 4, 0, 9, 3, 2, 12)$ y
$pred = (2, 0, 3, -1, 5, 6, 3, 4)$. El camino a 7 es $\langle 3, 6, 5, 4, 7 \rangle$: $2 + 1 + 6 + 3 = 12$.

Una pareja vieja

El grafo de la primera versión, con el código

`dijkstra(G, 7)`. La pareja $(12, 3)$ entra cuando sale 1 y se queda en la cola hasta el final, sin que nadie la use: el camino bueno hasta 3 aparece después, por 4.

Sale (43)	Línea 45	pqueue después
$(0, 7)$	$0 = 0$: se procesa	$(2, 1)$, $(3, 2)$
$(2, 1)$	$2 = 2$: se procesa; mete $(12, 3)$	$(3, 2)$, $(12, 3)$
$(3, 2)$	$3 = 3$: se procesa; mete $(5, 5)$	$(5, 5)$, $(12, 3)$
$(5, 5)$	$5 = 5$: se procesa; mete $(6, 4)$ y $(6, 6)$	$(6, 4)$, $(6, 6)$, $(12, 3)$
$(6, 4)$	$6 = 6$: se procesa; mete $(7, 3)$	$(6, 6)$, $(7, 3)$, $(12, 3)$
$(6, 6)$	$6 = 6$: se procesa	$(7, 3)$, $(12, 3)$
$(7, 3)$	$7 = 7$: se procesa	$(12, 3)$
$(12, 3)$	$dist[3] = 7 \neq 12$: **se descarta**	vacía

Una pareja vieja

Pregunta

En el empate entre $(6, 4)$ y $(6, 6)$, `heappop` sacó primero $(6, 4)$. ¿Por qué, si en la primera versión salió primero 6?

Respuesta: el empate lo decide la tupla

Respuesta

Con distancias iguales, Python compara el segundo elemento de la tupla, y $4 < 6$. En la primera versión el empate se rompió a mano. Con 4 primero, *dist*[3] pasa de 12 a 7 sin pasar por 11.

dijkstraST: uno a uno

```
18 while len(pqueue)!=0 and not found:
19   du,u = heappop(pqueue)
20   # al salir t, su distancia es final
21   if u == t:
22     found = True
23   else:
24     # pareja vieja: se descarta
25     if dist[u] == du:
26       for v,duv in G[u]:
27         if du+duv<dist[v]:
28           dist[v] = du+duv
29           heappush(pqueue, (dist[v], v))
30 return dist[t]
```

Las líneas resaltadas

- 18: el ciclo para cuando `found` es verdadero.
- 21 y 22: la primera vez que sale t, su distancia es la final.
- 30: devuelve `dist[t]`, que es ∞ si t no se alcanza.

Por qué los pesos no pueden ser negativos

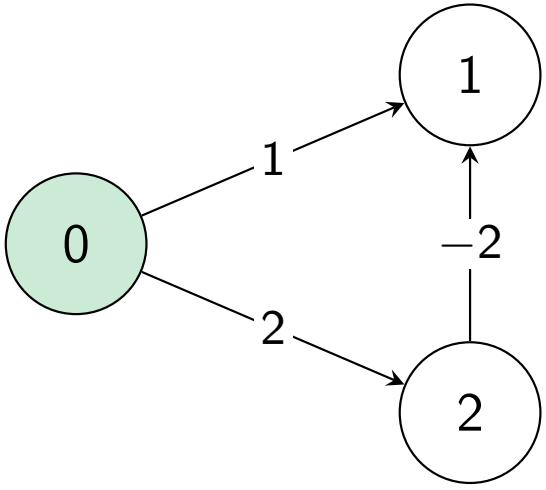
Teorema (CLRS, Teorema 24.6)

Si G es dirigido con pesos $w(u, v) \geq 0$ y fuente s, al terminar Dijkstra $dist[u] = \delta(s, u)$ para todo $u \in V$. La demostración muestra que u ya tiene ese valor cuando sale de la cola.

Dónde entra la hipótesis

Sea p un camino más corto de s a u y sea y el primer vértice de p que sigue pendiente. Entonces $dist[y] = \delta(s, y) \leq \delta(s, u) \leq dist[u]$; la desigualdad del medio se cumple porque el tramo de y a u pesa 0 o más. Como u salió primero, $dist[u] \leq dist[y]$, y todo es igual.

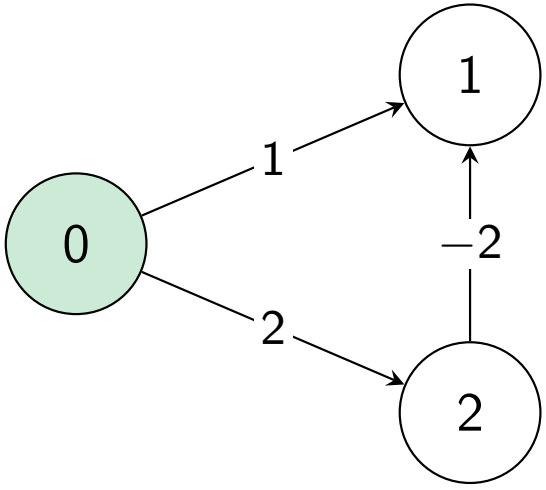
Un contraejemplo



Pregunta

`dijkstraST(G, 0, 1)`. ¿Qué devuelve?
¿Cuál es la distancia de 0 a 1?

Un contraejemplo



Respuesta

Sale $(0, 0)$ y entran $(1, 1)$ y $(2, 2)$. Sale $(1, 1)$: es t, y la función devuelve 1. Pero $\langle 0, 2, 1 \rangle$ pesa $2 - 2 = 0$.

Un contraejemplo

Y la versión uno a todos

`dijkstra(G, 0)` saca $(2, 2)$, mejora $dist[1]$ a 0 y mete $(0, 1)$: aquí acierta, pero procesa el vértice 1 dos veces. Con pesos negativos un vértice puede salir muchas veces y el costo $O((n + m) \log n)$ deja de valer; con un ciclo negativo, el `while` no termina.

Con pesos negativos

Si el grafo es un DAG, la pasada en orden topológico; si no, Bellman-Ford-Moore.

Lo que cuesta Dijkstra

Cuántas parejas entran a la cola

Con pesos no negativos cada vértice se procesa una vez, cuando sale su pareja buena, y en ese momento cada una de sus aristas mete a lo sumo una pareja. Entran a lo sumo $m + 1$ parejas.

Tiempo

Cada `heappush` y cada `heappop` cuesta $O(\log m)$, y $\log m \leq \log n^2 = 2 \log n$. Con las $O(n + m)$ vueltas: $O((n + m) \log n)$.

Lo que cuesta Dijkstra

Algoritmo	Pesos	Tiempo
Búsqueda en amplitud	todos iguales	$O(n+m)$
Pasada en orden topológico	cualquiera, sin ciclos	$O(n+m)$
Dijkstra	no negativos	$O((n+m)\log n)$
Bellman-Ford-Moore	cualquiera	$O(nm)$

Pregunta

Si todos los pesos valen 1, ¿qué hace Dijkstra que no haga la búsqueda en amplitud?

Respuesta: la cola de prioridad sobra

Respuesta

Con pesos iguales, la búsqueda en amplitud ya saca los vértices en orden de distancia, y su cola común hace el trabajo del montículo sin el $\log n$. Dijkstra da las mismas distancias y paga ese logaritmo.

Para grafos de estado

Si las transiciones cuestan distinto, la búsqueda en amplitud deja de servir y entra Dijkstra. Es lo que pasa con Super Mario.

Contenido

1 Antes de empezar

2 Caminos más cortos en un DAG

3 El algoritmo de Dijkstra

4 UVa 10269: Adventure of Super Mario

5 Problemas de juez

6 Errores comunes

7 Ejercicios

8 Lo que sigue

9 Referencias

UVa 10269, Adventure of Super Mario

El problema

Hay A aldeas, numeradas de 1 a A, y B castillos, de $A+1$ a $A+B$, unidos por M vías de doble sentido. Caminar por una vía de longitud ℓ toma ℓ unidades de tiempo. Mario sale del castillo $A+B$ y va a su casa, en la aldea 1.

Enunciado y envío: `https://vjudge.net/problem/UVA-10269`

UVa 10269, Adventure of Super Mario

La bota mágica

- Con la bota puesta, Mario corre de un lugar a otro en tiempo 0.
- Cada carrera empieza y termina en una aldea o en un castillo, cubre a lo sumo L kilómetros y nunca atraviesa un castillo: si hay uno en el camino, ahí para.
- La bota se usa a lo sumo K veces.

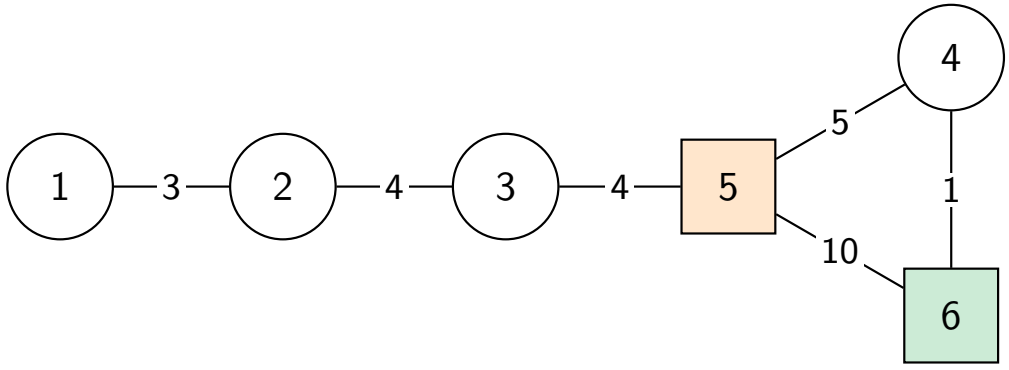
Entrada y salida

$1 \leq A, B \leq 50$, $1 \leq L \leq 500$, $0 \leq K \leq 10$ y longitudes de 1 a 100. Se imprime el menor tiempo para llegar a casa.

El ejemplo del enunciado

```
1
4 2 6 9 1
4 6 1
5 6 10
4 5 5
3 5 4
2 3 4
1 2 3
```

Salida: 9



$A = 4$, $B = 2$, $M = 6$, $L = 9$, $K = 1$. Los cuadros son los castillos 5 y 6.

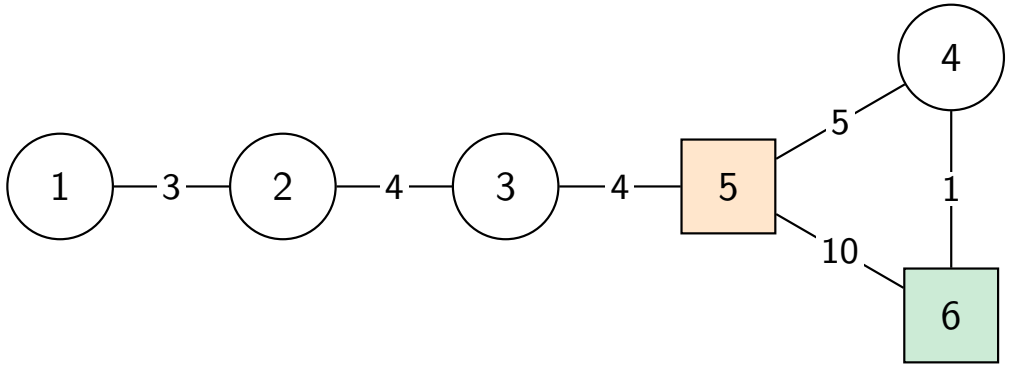
Pregunta

Sin la bota, ¿cuánto tarda Mario de 1 a 6? ¿Dónde conviene ponérsela?

El ejemplo del enunciado

```
1
4 2 6 9 1
4 6 1
5 6 10
4 5 5
3 5 4
2 3 4
1 2 3
```

Salida: 9



$A = 4$, $B = 2$, $M = 6$, $L = 9$, $K = 1$. Los cuadros son los castillos 5 y 6.

Respuesta

A pie, $\langle 1, 2, 3, 5, 4, 6 \rangle$ toma $3 + 4 + 4 + 5 + 1 = 17$. Con la bota puesta en 2, la carrera $\langle 2, 3, 5 \rangle$ cubre $8 \leq 9$ y para en el castillo 5; el resto a pie: $3 + 0 + 5 + 1 = 9$.

Por qué no basta el grafo de lugares

El lugar no dice todo

Desde la aldea 3 se puede seguir corriendo o no, según si Mario trae la bota puesta, cuánto le queda de la carrera y cuántos usos le quedan. Dos llegadas a 3 con el mismo tiempo pueden terminar en tiempos distintos.

Los costos no son iguales

Caminar cuesta la longitud de la vía y correr cuesta 0. La búsqueda en amplitud, que se usó en New Villa, cuenta pasos y no sirve: sobre el grafo de estados se usa Dijkstra.

Los estados

Representación de los estados

- Ubicación de Mario: $a \in [1, \ldots, A + B]$.
- Usos disponibles de la bota: $b \in [0, \ldots, K]$.
- ¿Tiene puesta la bota?: $c \in \{0, 1\}$.
- Distancia pendiente con la bota: $d \in [0, \ldots, L]$.

Los estados son tuplas (a, b, c, d).

Inicial y finales

Estado inicial: $(1, K, 0, 0)$. Estados finales: los de la forma $(A + B, b, c, d)$. Las vías son de doble sentido y una carrera al revés también es válida, así que ir de 1 a $A + B$ cuesta lo mismo que volver.

Las transiciones

Moverse sin la bota: $(a, b, 0, d) \to (a', b, 0, d)$

Cambia a. Condición: (a, a') es una arista de peso v. Costo: v.

Moverse con la bota: $(a, b, 1, d) \to (a', b, 1, d - v)$

Cambian a y d. Condiciones: (a, a') es una arista de peso v, $d \geq v$ y no se atraviesa un castillo: $a \leq A$, o la carrera empieza en a ($d = L$). Costo: 0.

Las transiciones

Quitarse la bota: $(a, b, 1, d) \to (a, b, 0, 0)$

Cambian c y d. Condición: $c = 1$. Costo: 0.

Ponerse la bota: $(a, b, 0, d) \to (a, b - 1, 1, L)$

Cambian b, c y d. Condiciones: $c = 0$ y $b > 0$. Costo: 0.

Pregunta

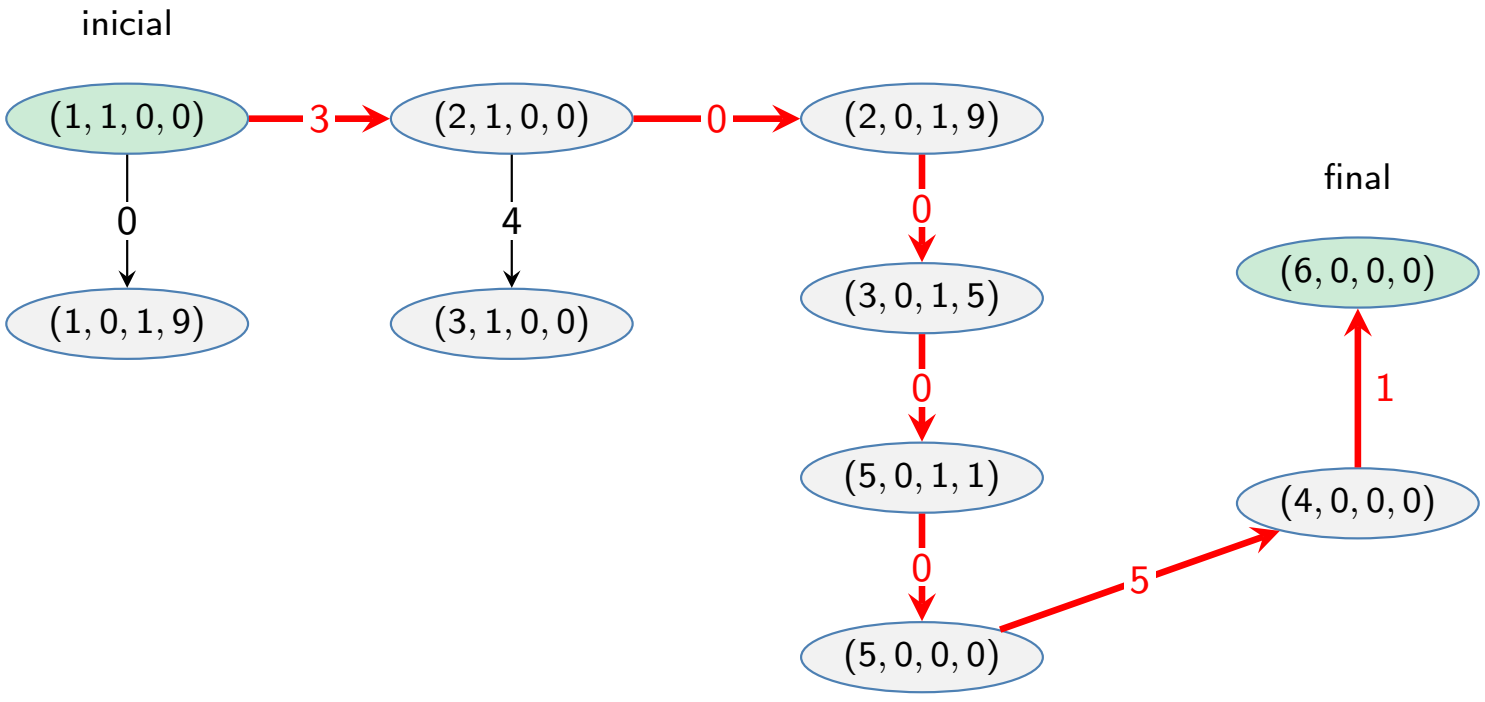
Mario llega corriendo al castillo 5 con $d = 1$. ¿Puede seguir corriendo hacia 4?

Respuesta: en un castillo la carrera termina

Respuesta

No, por dos razones: 5 es un castillo y la carrera no empezó ahí ($d = 1 \neq L$), y además la vía mide $5 > 1$. Puede quitarse la bota y caminar, o ponérsela otra vez si le quedan usos; en el ejemplo $K = 1$ y no le quedan.

Una parte del grafo de estados



El camino resaltado cuesta $3 + 0 + 0 + 0 + 0 + 5 + 1 = 9$: caminar a 2, ponerse la bota, correr a 3 y a 5, quitársela y caminar a 4 y a 6.

Cuántos estados y cuántas transiciones

En el peor caso

$$n = (A + B) \cdot 2 \cdot (K + 1) \cdot (L + 1), \qquad m = (A + B) \cdot n.$$

Con $A + B = 100$, $K = 10$ y $L = 500$: $n = 100 \cdot 2 \cdot 11 \cdot 501 = 1\,102\,200$ estados. Dijkstra cuesta $O((n + m) \log n)$.

Solo los alcanzables

El grafo de estados no se construye: los sucesores de cada estado se calculan cuando sale de la cola, y solo se guardan los estados a los que se llega.

Del estado al código

El estado del código: `(u, l, k)`

- `u` es a, el lugar, y `k` es b, los usos que quedan.
- `l` es d, la distancia pendiente. No hace falta c: quitarse la bota y caminar es una sola transición, que deja `l` en 0.

En la cola

Tuplas `(c, u, l, k)`, donde `c` es el tiempo. El diccionario `vis` guarda el menor tiempo conocido de cada estado: hace el papel de `dist`.

El código completo

```
from sys import stdin
from heapq import heappop, heappush

def main():
  cases = int(stdin.readline())
  while cases:
    a, b, m, L, K = map(int,
    ↪  stdin.readline().split())
    G = [[] for _ in range(a + b + 1)]
    for _ in range(m):
      u, v, p = map(int,
      ↪  stdin.readline().split())
      G[u].append((v, p))
      G[v].append((u, p))

    # estado (u, l, k): lugar, carga que le
    # queda a la bota (0 sin bota) y usos
    # vis[estado]: menor tiempo conocido
    queue, vis, flag = [], dict(), False
    heappush(queue, (0, 1, 0, K))
    vis[(1, 0, K)] = 0
    while len(queue) > 0 and not flag:
      (c, u, l, k) = heappop(queue)

      # al salir a + b, c es el mínimo
      if u == a + b: flag = True
      else:
        # tupla vieja: se descarta
        if vis[(u, l, k)] == c:
          # se coloca los zapatos especiales
          if l != L and k > 0:
            if (u, L, k - 1) not in vis or
            ↪  vis[(u, L, k - 1)] > c:
              heappush(queue, (c, u, L, k - 1))
              vis[(u, L, k - 1)] = c
          for (v, p) in G[u]:
            if (v, 0, k) not in vis or vis[(v,
            ↪  0, k)] > c + p:
              # se mueve sin usar los zapatos
              heappush(queue, (c + p, v, 0, k))
              vis[(v, 0, k)] = c + p
            # se mueve usando los zapatos
            # alcanza la carga y u es aldea
            # o la carrera empieza en u
            if l >= p and (u <= a or l == L):
              if (v, l - p, k) not in vis or
              ↪  vis[(v, l - p, k)] > c:
                heappush(queue, (c, v, l - p,
                ↪  k))
                vis[(v, l - p, k)] = c
    print(c)
    cases -= 1

main()
```

La lectura y el estado inicial

```
15 a, b, m, L, K = map(int, stdin.readline().split())
16 G = [[] for _ in range(a + b + 1)]
17 for _ in range(m):
18   u, v, p = map(int, stdin.readline().split())
19   G[u].append((v, p))
20   G[v].append((u, p))
21
22 # estado (u, l, k): lugar, carga que le
23 # queda a la bota (0 sin bota) y usos
24 # vis[estado]: menor tiempo conocido
25 queue, vis, flag = [], dict(), False
26 heappush(queue, (0, 1, 0, K))
27 vis[(1, 0, K)] = 0
```

Las líneas resaltadas

- 16: lugares de 1 a $a + b$; la posición 0 no se usa.
- 19 y 20: cada vía entra en las dos listas.
- 26 y 27: el estado inicial $(1, 0, K)$, con tiempo 0: en la aldea 1, sin la bota y con los K usos.

Sacar un estado

```
28 while len(queue) > 0 and not flag:
29   (c, u, l, k) = heappop(queue)
30
31   # al salir a + b, c es el mínimo
32   if u == a + b: flag = True
33   else:
34     # tupla vieja: se descarta
35     if vis[(u, l, k)] == c:
```

Las líneas resaltadas

- 29: sale la tupla de menor tiempo.
- 32: si es el castillo $a + b$, ese tiempo es la respuesta y el ciclo para, como en `dijkstraST`.
- 35: la tupla vieja se descarta, como en la línea 45 de `dijkstra`.

Ponerse la bota

```
36 # se coloca los zapatos especiales
37 if l != L and k > 0:
38   if (u, L, k - 1) not in vis or vis[(u, L, k - 1)] > c:
39     heappush(queue, (c, u, L, k - 1))
40     vis[(u, L, k - 1)] = c
```

Las líneas resaltadas

- 37: la carrera no puede estar recién empezada (`l != L`) y tiene que quedar algún uso.
- 38 a 40: el estado $(u, L, k-1)$ entra con el mismo tiempo `c`, si mejora lo que hay en `vis`.

Caminar

```
41 for (v, p) in G[u]:
42   if (v, 0, k) not in vis or vis[(v, 0, k)] > c + p:
43     # se mueve sin usar los zapatos
44     heappush(queue, (c + p, v, 0, k))
45     vis[(v, 0, k)] = c + p
```

Las líneas resaltadas

- 41: las vías que salen de u, con su longitud `p`.
- 42, 44 y 45: caminar a v cuesta `p` y deja `1` en 0. Si Mario traía la bota puesta, se la quitó antes de caminar.

Correr

```
46 # se mueve usando los zapatos
47 # alcanza la carga y u es aldea
48 # o la carrera empieza en u
49 if l >= p and (u <= a or l == L):
50   if (v, l - p, k) not in vis or vis[(v, l - p, k)] > c:
51     heappush(queue, (c, v, l - p, k))
52     vis[(v, l - p, k)] = c
```

Las líneas resaltadas

- 49: la carga alcanza para la vía, y u es una aldea o la carrera empieza en u. Si u es un castillo y `l < L`, Mario llegó corriendo a él y ahí para.
- 50 a 52: el estado $(v, l - p, k)$ entra con el mismo tiempo `c`.

El código sobre el ejemplo

Turno	Sale (29)	De dónde viene
1	$(0, 1, 0, 1)$	estado inicial, líneas 26 y 27
2	$(0, 1, 9, 0)$	se pone la bota en 1, líneas 37 a 40
3	$(0, 2, 6, 0)$	corre de 1 a 2, líneas 49 a 52
6	$(0, 3, 2, 0)$	corre de 2 a 3

Las tuplas son `(c, u, l, k)`. Salen 25; la tabla muestra las del camino de la respuesta y las primeras, con su turno.

Pregunta

Las primeras en salir cuestan 0: Mario se pone la bota en 1 y corre a 2 y a 3. ¿Por qué ese camino no da la respuesta?

El código sobre el ejemplo

Turno	Sale (29)	De dónde viene
1	$(0, 1, 0, 1)$	estado inicial, líneas 26 y 27
2	$(0, 1, 9, 0)$	se pone la bota en 1, líneas 37 a 40
3	$(0, 2, 6, 0)$	corre de 1 a 2, líneas 49 a 52
6	$(0, 3, 2, 0)$	corre de 2 a 3
9	$(3, 2, 0, 1)$	camina de 1 a 2, líneas 42 a 45
10	$(3, 2, 9, 0)$	se pone la bota en 2
13	$(3, 3, 5, 0)$	corre de 2 a 3
15	$(3, 5, 1, 0)$	corre de 3 al castillo 5

Las tuplas son `(c, u, l, k)`. Salen 25; la tabla muestra las del camino de la respuesta y las primeras, con su turno.

Respuesta

Llega a 3 con 2 de carga y la vía a 5 mide 4. Sin usos, todo lo demás es a pie: $4 + 5 + 1 = 10$. Caminar primero a 2 cuesta 3, pero la carrera desde 2 llega hasta el castillo 5.

El código sobre el ejemplo

Turno	Sale (29)	De dónde viene
1	$(0, 1, 0, 1)$	estado inicial, líneas 26 y 27
2	$(0, 1, 9, 0)$	se pone la bota en 1, líneas 37 a 40
3	$(0, 2, 6, 0)$	corre de 1 a 2, líneas 49 a 52
6	$(0, 3, 2, 0)$	corre de 2 a 3
9	$(3, 2, 0, 1)$	camina de 1 a 2, líneas 42 a 45
10	$(3, 2, 9, 0)$	se pone la bota en 2
13	$(3, 3, 5, 0)$	corre de 2 a 3
15	$(3, 5, 1, 0)$	corre de 3 al castillo 5
24	$(8, 4, 0, 0)$	camina de 5 a 4
25	$(9, 6, 0, 0)$	camina de 4 a 6: línea 32, imprime 9

Las tuplas son `(c, u, l, k)`. Salen 25; la tabla muestra las del camino de la respuesta y las primeras, con su turno.

El final

$(9, 6, 0, 0)$ sale antes que cualquier otra tupla del castillo 6, así que 9 es el menor tiempo. Es el camino resaltado en el grafo de estados.

Pregunta: el tamaño en el código

Pregunta

El estado del código no guarda c. ¿Cuántos estados `(u, l, k)` hay en el peor caso?

Respuesta: la mitad de los estados

Respuesta

$(A + B)(L + 1)(K + 1) = 100 \cdot 501 \cdot 11 = 551\,100$, la mitad de la cuenta con c. Por cada estado se miran a lo sumo las vías de su lugar.

Contenido

1 Antes de empezar

2 Caminos más cortos en un DAG

3 El algoritmo de Dijkstra

4 UVa 10269: Adventure of Super Mario

5 Problemas de juez

6 Errores comunes

7 Ejercicios

8 Lo que sigue

9 Referencias

UVa 10986, Sending email

El problema

$n \leq 20\,000$ servidores y $m \leq 50\,000$ cables de doble sentido, con latencias de 0 a 10 000. El menor tiempo de S a T, o `unreachable`.

Enunciado y envío: `https://vjudge.net/problem/UVA-10986`

Cómo atacarlo

Pesos no negativos y un solo destino: `dijkstraST(G, S, T)`, con cada cable en las dos listas. Si devuelve ∞, se imprime `unreachable`. La trampa: m puede ser 0, y la salida lleva `Case #x:` delante.

UVa 1112, Mice and Maze

El problema

$N \leq 100$ celdas y pasajes de un solo sentido con un tiempo. En cada celda hay un ratón que sigue el camino más rápido a la salida E. ¿Cuántos llegan en a lo sumo T unidades?

Enunciado y envío: `https://vjudge.net/problem/UVA-1112`

Cómo atacarlo

Son N caminos que terminan en E. Si se invierten las aristas, son caminos que salen de E: un solo `dijkstra(G, E)` sobre el grafo invertido, y se cuentan las distancias $\leq T$. La trampa: hay líneas en blanco entre casos, a la entrada y a la salida.

UVa 10000, Longest Paths

El problema

Un grafo dirigido sin ciclos de $n \leq 100$ vértices y un punto de partida s. La longitud del camino más largo que sale de s y el vértice donde termina; si hay varios, el de menor número.

Enunciado y envío: `https://vjudge.net/problem/UVA-10000`

Cómo atacarlo

El camino más largo con aristas de peso 1 es el más corto con aristas de peso -1. Sin ciclos, `ssspDAG` los calcula en una pasada; Dijkstra no sirve con esos pesos. La trampa: el formato de la salida, con una línea en blanco después de cada caso.

Contenido

1 Antes de empezar

2 Caminos más cortos en un DAG

3 El algoritmo de Dijkstra

4 UVa 10269: Adventure of Super Mario

5 Problemas de juez

6 Errores comunes

7 Ejercicios

8 Lo que sigue

9 Referencias

Errores comunes

Relajar en un orden que no es topológico

Una sola pasada solo sirve en orden topológico. Con otro orden, un vértice puede procesarse antes de tener su distancia final.

Dijkstra con pesos negativos

`dijkstraST` puede devolver un valor equivocado, y `dijkstra` pierde su costo o no termina.

Procesar parejas viejas

Sin la línea 45, un vértice se procesa una vez por cada pareja suya en la cola, con distancias que ya no son las suyas.

Errores comunes

Meter en la cola `(v, distancia)`

Las tuplas se comparan por su primer elemento. Con el vértice adelante, sale primero el de menor número, no el más cercano.

Olvidar en el estado lo que decide el futuro

En Super Mario, un estado sin la carga que le queda a la bota junta llegadas que no son equivalentes.

Contenido

1 Antes de empezar

2 Caminos más cortos en un DAG

3 El algoritmo de Dijkstra

4 UVa 10269: Adventure of Super Mario

5 Problemas de juez

6 Errores comunes

7 Ejercicios

8 Lo que sigue

9 Referencias

Ejercicios

1. Un DAG a mano

Sea D el grafo con $V = \{0, \ldots, 5\}$ y estas aristas, en este orden de entrada, como `u v peso`: `1 3 4`, `1 2 -1`, `0 1 2`, `2 4 5`, `2 3 1`, `3 5 3`, `3 4 -2`, `4 5 1`, `0 3 6`. Cada arista se agrega al final de `G[u]`. Dé el orden que devuelve `topoSort`, la tabla de `ssspDAG(G, 1)` y el camino de 1 a 5.

2. Parejas viejas

Sea H el grafo con $V = \{0, \ldots, 5\}$ y aristas `0 1 7`, `0 2 2`, `2 1 3`, `2 3 8`, `1 3 1`, `1 4 6`, `3 4 2`, `4 5 1`, `3 5 5`, `2 4 9`, en ese orden. Haga la tabla de `dijkstra(G, 0)` con lo que sale y la cola después. ¿Cuántas parejas salen y cuántas descarta la línea 45?

Ejercicios

3. Un DAG con un peso negativo

Sea N el grafo con aristas `0 1 2`, `0 2 5`, `2 1 -4`, `1 3 1`. Calcule `dijkstraST(G, 0, 3)`, `dijkstra(G, 0)` y `ssspDAG(G, 0)`. ¿Cuáles aciertan? ¿Cuántas veces se procesa el vértice 3 en `dijkstra`?

4. El camino más largo

Modifique `ssspDAG` para que calcule los caminos más largos desde s en un DAG, sin cambiar los pesos de `G`. Aplíquelo al grafo de la clase desde el vértice 4: ¿cuál es el vértice más lejano y por qué camino?

Ejercicios

5. Otra aventura de Mario

$A = 3$, $B = 2$, $L = 7$, y las vías 1 2 4, 2 3 3, 3 5 6, 2 4 2, 4 5 3, 1 3 9.

- Liste los sucesores del estado inicial $(1, K, 0, 0)$ con su costo.
- Dé el menor tiempo para $K = 0$, $K = 1$ y $K = 2$, con el camino de cada uno.
- ¿Puede Mario correr de 1 a 5 en una sola carrera?

6. Una demostración

Demuestre que en un DAG, cuando `ssspDAG` procesa u, $dist[u] = \delta(s, u)$. Escriba la demostración en cuatro partes: teorema, estrategia, desarrollo y conclusión.

Pistas

Ejercicios 1 a 3

1. El orden es $0, 1, 2, 3, 4, 5$, y 0 no se alcanza desde 1. $dist[5] = -1$ por $\langle 1, 2, 3, 4, 5 \rangle$.
2. Salen 10 parejas y 4 son viejas. $dist = (0, 5, 2, 6, 8, 9)$.
3. `dijkstraST` devuelve 3 y la distancia es 2. Las otras dos aciertan.

Pistas

Ejercicios 4 a 6

4. Cambie ∞ por $-\infty$ y $<$ por $>$. El más lejano es 5, a 12, por $\langle 4, 2, 1, 5 \rangle$.
5. Los tiempos son 9, 3 y 0. La carrera de 1 a 4 mide 6 y para en el castillo.
6. Se procede por inducción sobre la posición de u en el orden: las aristas que llegan a u salen de vértices anteriores.

Contenido

1 Antes de empezar

2 Caminos más cortos en un DAG

3 El algoritmo de Dijkstra

4 UVa 10269: Adventure of Super Mario

5 Problemas de juez

6 Errores comunes

7 Ejercicios

8 Lo que sigue

9 Referencias

Lo que sigue

Caminos más cortos entre todas las parejas

Hasta ahora, una fuente. Si se quieren las distancias entre todas las parejas, se puede correr Dijkstra desde cada vértice, o usar Floyd-Warshall, que trabaja sobre una matriz.

Para investigar

- CLRS, Sección 25.2: ¿qué significa $d_{ij}^{(k)}$ en Floyd-Warshall, y por qué el ciclo de k va por fuera?
- En Super Mario, ¿cómo se calcularía de antemano qué lugares alcanza una carrera desde cada lugar, pasando solo por aldeas? ¿Qué estado queda si ese dato se conoce?

Contenido

1 Antes de empezar

2 Caminos más cortos en un DAG

3 El algoritmo de Dijkstra

4 UVa 10269: Adventure of Super Mario

5 Problemas de juez

6 Errores comunes

7 Ejercicios

8 Lo que sigue

9 Referencias

Referencias



Cormen, T. H., Leiserson, C. E., Rivest, R. L., Stein, C. *Introduction to Algorithms*, 3rd ed. MIT Press, 2009. Sección 24.2, Teorema 24.5 (caminos más cortos en un DAG); Sección 24.3, Teorema 24.6 (Dijkstra); Sección 25.2 (Floyd-Warshall).



Dijkstra, E. W. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.



Halim, S., Halim, F., Effendy, S. *Competitive Programming 4*. Lulu, 2020. Capítulo 4: caminos más cortos desde una fuente y grafos de estados.



UVa Online Judge, problemas 1112, 10000, 10269 y 10986. `https://vjudge.net`