Problemas de cortes

Puentes, puntos de articulación y el árbol de puentes

Carlos A Delgado S

Pontificia Universidad Javeriana Cali

Octubre de 2026

Contenido

1 Antes de empezar

2 Recuento

3 El árbol de puentes

4 Ejercicios resueltos

5 Cómo atacar estos problemas

6 Errores comunes

7 Ejercicios propuestos

8 Referencias

Contenido

1 Antes de empezar

2 Recuento

3 El árbol de puentes

4 Ejercicios resueltos

5 Cómo atacar estos problemas

6 Errores comunes

7 Ejercicios propuestos

8 Referencias

Objetivos de la sesión

Al terminar la clase, usted podrá

- Construir el árbol de puentes de un grafo no dirigido en $\Theta(V + E)$ y demostrar que lo que sale es un árbol.
- Reconocer, leyendo un enunciado de juez, si el corte que pide es por vértice o por arista.
- Orientar las calles de una ciudad para que quede fuertemente conexa, y decir cuáles no se pueden orientar.

Objetivos de la sesión

Y también

- Contar en cuántos pedazos queda una red al quitarle un vértice, con los hijos de la raíz y la comparación $w.low \geq v.d$.
- Contraer un grafo dirigido por sus componentes fuertemente conexos y decidir, mirando los sumideros del DAG que queda, si alguna ciudad se alcanza desde todas las demás.

La pregunta de hoy

Lo que ya se calcula

Una sola profundidad deja $v.d$ y $v.low$ de cada vértice, y con esos dos números salen los puntos de articulación y los puentes en $\Theta(V + E)$.

La pregunta

Casi ninguno se queda en la lista de puentes. Pide orientar calles, contar pedazos, medir distancias. ¿Qué se construye *con* los puentes para que esas preguntas se vuelvan fáciles?

La respuesta de la sesión

Se contraen los pedazos que quedan al quitar los puentes. Lo que resulta es un árbol, y sobre un árbol casi todo es un recorrido. Los componentes biconexos, los que salen de cortar por puntos de articulación, quedan para después.

Contenido

1 Antes de empezar

2 Recuento

3 El árbol de puentes

4 Ejercicios resueltos

5 Cómo atacar estos problemas

6 Errores comunes

7 Ejercicios propuestos

8 Referencias

Los tres criterios

Definición (CLRS, Problema 22-2)

$v.low = \text{mín}\{\, v.d,\ w.d :$
(u, w) *es de retroceso para algún descendiente* u *de* $v\,\}$.

Qué dice

Lo más arriba del árbol a donde se llega desde el subárbol de v con un solo salto de retroceso. En un grafo no dirigido solo hay aristas de árbol y de retroceso (Teorema 22.10), así que no hace falta pila: basta excluir la arista por la que se bajó.

Los tres criterios

Punto de articulación, raíz (Problema 22-2a)

La raíz de un árbol de la profundidad lo es si y solo si tiene al menos dos hijos en ese árbol.

Punto de articulación, los demás (Problema 22-2b)

v lo es si y solo si tiene un hijo w con $w.low \geq v.d$.

Puente (Problema 22-2)

Una arista de árbol (v, w), con w hijo de v, lo es si y solo si $w.low > v.d$. Una arista de retroceso nunca lo es.

Dos detalles que hoy se usan

La desigualdad estricta

Con $w.low = v.d$ hay un retroceso desde el subárbol de w hasta v mismo. Ese salto no sirve contra la articulación, porque v es el que se quita; contra el puente sí, porque v se queda y cierra un ciclo que contiene a (v, w).

La arista de entrada se excluye por su identificador

Preguntar `v != padre` alcanza mientras no haya dos aristas entre el mismo par. Si las hay, la segunda copia es un retroceso legítimo hasta el padre y queda descartada, así que una arista doble sale como puente sin serlo. Desde hoy cada vecino viene con el número de su arista y se compara `i != entrada`.

Dos detalles que hoy se usan

Lo que cuesta

Una profundidad, $\Theta(V + E)$ para las dos respuestas a la vez.

Contenido

1 Antes de empezar

2 Recuento

3 El árbol de puentes

4 Ejercicios resueltos

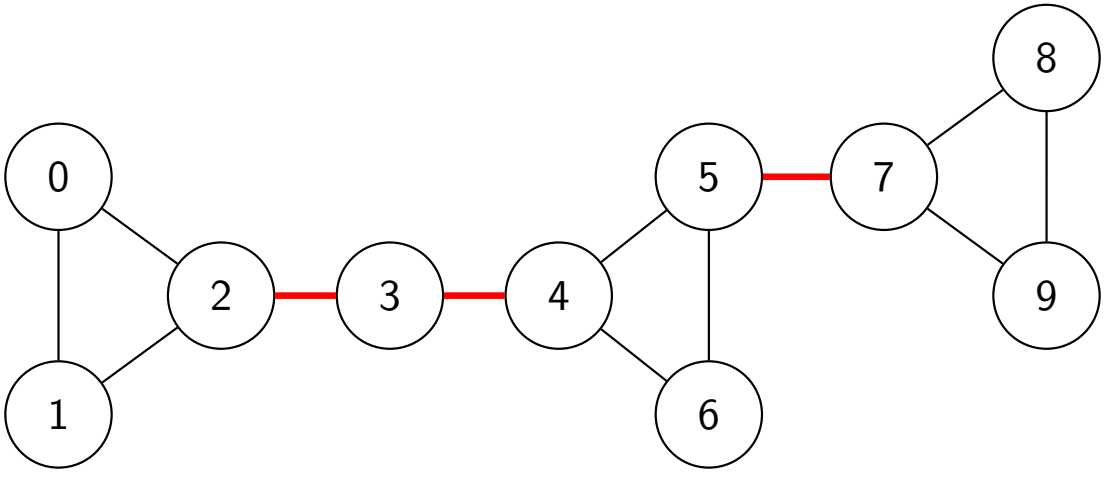
5 Cómo atacar estos problemas

6 Errores comunes

7 Ejercicios propuestos

8 Referencias

Por tanteo



Diez vértices, doce aristas, tres puentes

Un triángulo 0–1–2, el 3 colgando entre dos puentes, el triángulo 4–5–6 y el triángulo 7–8–9. Los puentes son 2–3, 3–4 y 5–7.

Por tanteo

La pregunta

Bórrelos. ¿Cuántos pedazos quedan y quién está en cada uno? Y después: si cada pedazo se encoge a un solo nodo y los tres puentes se dejan como aristas entre esos nodos, ¿qué forma tiene el dibujo que queda?

Lo que se observa

Quedan cuatro pedazos y tres aristas entre ellos, y el dibujo es un camino. Que no se cierre ningún ciclo tiene explicación: una arista dentro de un ciclo nunca es puente, de modo que los puentes son las aristas por las que no hay por dónde volver.

Los componentes 2-arista-conexos

Definición

Los componentes 2-arista-conexos de un grafo no dirigido G son los componentes conexos del grafo que queda al borrar de G todas sus aristas puente.

Adentro no queda ningún puente

Una arista e que sobrevive al borrado no es puente de G, así que está en un ciclo de G. Ninguna arista de ese ciclo es puente, luego el ciclo entero sobrevivió y vive dentro de un solo componente: e tampoco es puente de su componente.

Los componentes 2-arista-conexos

La lectura equivalente

Dos vértices distintos caen en el mismo componente exactamente cuando hay entre ellos dos caminos sin ninguna arista en común. Es el teorema de Menger para aristas, que este curso no demuestra; la definición de arriba es la que se programa.

No son los componentes biconexos

Estos reparten los vértices: cada uno cae en exactamente uno. Los biconexos reparten las aristas, y un punto de articulación queda en varios a la vez. Unos se obtienen quitando puentes y los otros quitando puntos de corte.

El árbol

Teorema

Sea G un grafo no dirigido conexo. Sea T el grafo cuyos nodos son los componentes 2-arista-conexos de G y que tiene una arista $\{ comp(u), comp(v) \}$ por cada puente (u, v) de G. Entonces T es un árbol.

Demostración

Se procede en dos partes: la conexidad de forma directa, y la ausencia de ciclos por contradicción.

El árbol

Desarrollo, primera parte

T es conexo. Un camino de G entre dos vértices cualesquiera recorre vértices de uno u otro componente; cada vez que cambia de componente lo hace por una arista puente, que en T es una arista entre esos dos nodos. Borrando los tramos que se quedan dentro de un componente queda un camino de T.

El árbol

Desarrollo, segunda parte

Supóngase que T tiene un ciclo $C_0, e_1, C_1, e_2, \ldots, e_k, C_k$ con $C_k = C_0$, los nodos $C_0, \ldots, C_{k-1}$ distintos y $k \geq 2$. Cada e_i es un puente de G con un extremo en C_{i-1} y el otro en C_i. Escríbase $e_1 = (u, v)$ con $u \in C_0$ y $v \in C_1$.

Desde v se llega a u sin usar e_1: dentro de C_1 hay camino de v hasta el extremo de e_2, se cruza e_2, y así hasta cruzar e_k y llegar dentro de C_0 hasta u. Los tramos interiores no usan puentes y los cruces usan $e_2, \ldots, e_k$, todos distintos de e_1. Ese camino más e_1 es un ciclo que contiene a e_1, y entonces e_1 no es puente, en contra de lo supuesto.

El árbol

Conclusión

Por lo tanto, se puede concluir que T es conexo y sin ciclos, es decir un árbol. Si G no es conexo el mismo argumento corre dentro de cada componente conexo, y T es un bosque con un árbol por componente.

Dos cuentas que salen gratis

Con G conexo, T tiene tantas aristas como puentes tiene G, y tantos nodos como esas aristas más uno. Un componente que es hoja de T cuelga del resto del grafo por un solo puente.

Cómo se construye

Tres pasadas

1. Los puentes, con Tarjan: una profundidad que calcula d y low y marca las aristas de árbol con $w.low > v.d$. $\Theta(V + E)$.
2. El etiquetado: para cada vértice sin componente, un recorrido que *no cruza puentes*; a todo lo que alcanza se le pone el mismo número. $\Theta(V + E)$.
3. El árbol: una pasada por las aristas puente, agregando a T la arista entre los números de sus dos extremos. $\Theta(E)$.

En total

$\Theta(V + E)$ en tiempo y $\Theta(V + E)$ de memoria. Tres recorridos completos, no uno por vértice ni uno por arista.

Cómo se construye

La segunda pasada no cruza puentes

Es la única condición del recorrido, y es la que separa los pedazos. Si se cruza un puente, dos componentes vecinos quedan con el mismo número, T pierde una arista y el árbol sale mal armado sin que nada avise.

Qué queda guardado

`es_puente`, el conjunto de identificadores de arista; `comp[v]`, el número de componente de cada vértice; y `T`, las listas de adyacencia del árbol.

El grafo y sus aristas

```
ARISTAS = [(0, 1), (1, 2), (2, 0),
           (2, 3),
           (3, 4),
           (4, 5), (5, 6), (6, 4),
           (5, 7),
           (7, 8), (8, 9), (9, 7)]
```

El grafo y sus aristas

```
def construir(vertices, aristas):
    # G[u] guarda parejas (vecino, id de la arista).
    G = {}
    for u in vertices:
        G[u] = []
    for i in range(len(aristas)):
        a, b = aristas[i]
        G[a].append((b, i))
        G[b].append((a, i))
    return G

def descubrir(u, reloj, d, low):
    reloj[0] = reloj[0] + 1
    d[u] = reloj[0]
    low[u] = reloj[0]
```

El grafo y sus aristas

Por qué el identificador

La lista de aristas manda: la arista *i* es `aristas[i]`, y los dos extremos la ven con el mismo número. Eso permite excluir la arista de entrada sin mirar quién es el padre, y marcar un puente guardando un entero.

Los puentes

```
def puentes_aux(G, u, entrada, reloj, d, low, es_puente):
    # entrada: el id de la arista que bajo a u; -1 en la raiz.
    descubrir(u, reloj, d, low)
    for v, i in G[u]:
        if d[v] == 0:
            puentes_aux(G, v, i, reloj, d, low, es_puente)
            low[u] = min(low[u], low[v])
            if low[v] > d[u]:
                es_puente.add(i)
        elif i != entrada:
            low[u] = min(low[u], d[v])
```

Los puentes

```
def puentes(G):
    # Devuelve el conjunto de ids de arista que son puente.
    d = {}
    low = {}
    for u in G:
        d[u] = 0
        low[u] = 0
    es_puente = set()
    reloj = [0]
    for u in G:
        if d[u] == 0:
            puentes_aux(G, u, -1, reloj, d, low, es_puente)
    return es_puente
```

Dónde está el criterio

`low[v] > d[u]` se evalúa al volver de la recursión, cuando `low[v]` ya es definitivo. Lo que se guarda es `i`, el número de la arista de árbol que acaba de bajar.

El etiquetado

```
def etiquetar(G, u, numero, es_puente, comp):
    # No cruza puentes: lo que alcanza es un componente.
    comp[u] = numero
    for v, i in G[u]:
        if comp[v] == -1 and i not in es_puente:
            etiquetar(G, v, numero, es_puente, comp)
```

El etiquetado

```
def componentes(G, es_puente):
    # comp[v]: el componente de v, y el total de componentes.
    comp = {}
    for u in G:
        comp[u] = -1
    total = 0
    for u in G:
        if comp[u] == -1:
            etiquetar(G, u, total, es_puente, comp)
            total = total + 1
    return comp, total
```

La condición

`i not in es_puente` es todo lo que distingue este recorrido de una profundidad corriente. `comp[v] == -1` dice que v sigue sin componente.

El árbol

```
def arbol_de_puentes(aristas, es_puente, comp, total):
    # Un nodo por componente, una arista por puente.
    T = {}
    for c in range(total):
        T[c] = []
    for i in sorted(es_puente):
        a, b = aristas[i]
        T[comp[a]].append(comp[b])
        T[comp[b]].append(comp[a])
    return T
```

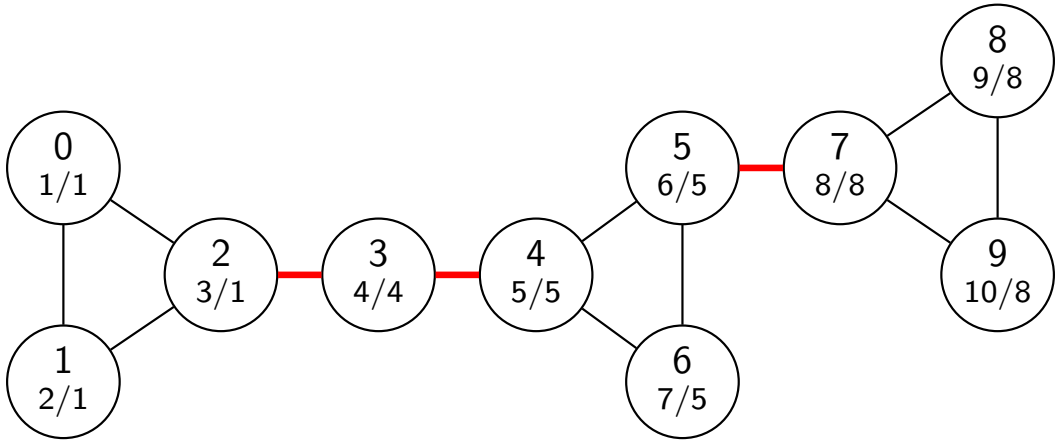
El árbol

```
def contraer(n, aristas):
    # Las tres pasadas seguidas. Theta(V+E) en total.
    G = construir(range(n), aristas)
    es_puente = puentes_con_pila(G)
    comp, total = componentes_con_pila(G, es_puente)
    T = arbol_de_puentes(aristas, es_puente, comp, total)
    return T, comp, es_puente
```

Las tres pasadas, una tras otra

`contraer` devuelve el árbol, la etiqueta de cada vértice y el conjunto de puentes. Con esos tres datos se responde casi cualquier pregunta del enunciado. Las dos primeras pasadas entran por `puentes_con_pila` y `componentes_con_pila`, que hacen lo mismo sin recursión y vienen en los archivos de código de la clase.

Paso a paso, arrancando en 0



d/*low* al terminar

La profundidad baja $0, 1, 2, 3, 4, 5, 6, 7, 8, 9$ en una sola rama. Las aristas de retroceso son 2–0, 6–4 y 9–7.

Paso a paso, arrancando en 0

Arista de árbol (v, w)	$w.low$ vs $v.d$	¿Puente?	Quién es
$(8, 9)$	$8 < 9$	no	
$(7, 8)$	$8 = 8$	no	
$(5, 7)$	$8 > 6$	sí	arista 8
$(5, 6)$	$5 < 6$	no	
$(4, 5)$	$5 = 5$	no	
$(3, 4)$	$5 > 4$	sí	arista 4
$(2, 3)$	$4 > 3$	sí	arista 3
$(1, 2)$	$1 < 2$	no	
$(0, 1)$	$1 = 1$	no	

Dos filas que no cuadran

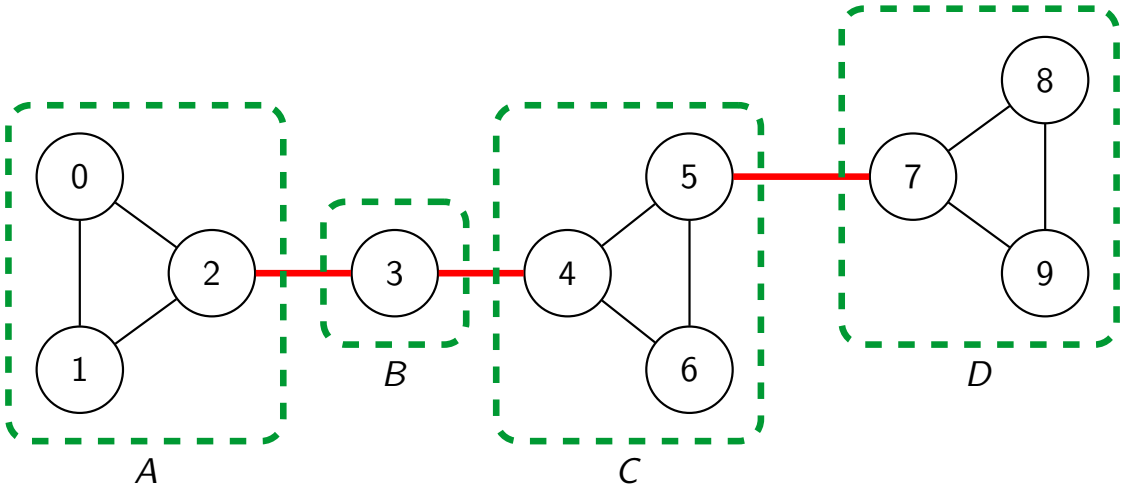
$(5, 7)$ tiene $8 > 6$ y la columna dice puente. $(7, 8)$ tiene $8 = 8$ y dice que no, igual que $(0, 1)$ con $1 = 1$. ¿Qué cambia cuando el *low* del hijo cae justo en el d del padre, y no por debajo?

Paso a paso, arrancando en 0

Las dos igualdades

$(0, 1)$ y $(7, 8)$ dan $w.low = v.d$: el retroceso llega hasta v mismo y cierra un ciclo que contiene la arista. Con el criterio de articulación, que admite la igualdad, el 0 y el 7 serían puntos de corte; el 7 lo es y el 0 no, porque la raíz se decide contando hijos.

Los cuatro componentes



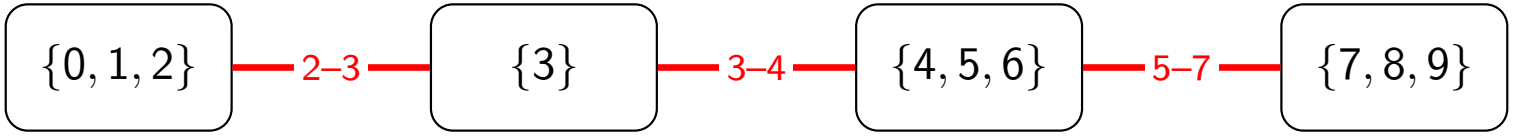
El etiquetado

Arranca en 0 y alcanza 1 y 2 sin cruzar la arista 3: componente $A = 0$.
Sigue con el 3, que tiene sus dos aristas marcadas y queda solo: $B = 1$.
Después 4, que alcanza 5 y 6: $C = 2$. Y 7, que alcanza 8 y 9: $D = 3$.

En el código

`comp` queda $[0, 0, 0, 1, 2, 2, 2, 3, 3, 3]$ para los vértices 0 a 9.

El árbol que queda



Cuatro nodos, tres aristas

`T` queda $\{0{:}[1],\ 1{:}[0,2],\ 2{:}[1,3],\ 3{:}[2]\}$: un camino. Las hojas son A y D, los dos componentes que cuelgan por un solo puente.

El árbol que queda

Qué se responde con esto

- ¿Cuántos puentes hay que cruzar para ir del 1 al 8? La distancia de A a D en el árbol: tres.
- ¿El 4 y el 6 siguen conectados si se cae cualquier arista sola? `comp[4] == comp[6]`, así que sí.
- ¿Qué arista separa más parejas de vértices? Un puente, y se mide con los tamaños de los dos lados en el árbol.

Contenido

1 Antes de empezar

2 Recuento

3 El árbol de puentes

4 Ejercicios resueltos
- UVa 610 — Street Directions
- UVa 10765 — Doves and Bombs
- Capital
- Interplanetary

5 Cómo atacar estos problemas

6 Errores comunes

Contenido

7 Ejercicios propuestos

8 Referencias

UVa 610 — Street Directions

El problema

UVa 610 — Street Directions

`https://vjudge.net/problem/UVA-610`
Envío: `https://vjudge.net/problem/UVA-610`

Lo que pide

Hay n intersecciones ($2 \leq n \leq 1000$) y m calles de doble vía, a lo sumo una por cada par. Hay que convertir en una vía tantas calles como se pueda, de modo que desde cada intersección se siga pudiendo llegar a todas las demás. La calle que no se pueda convertir se imprime en los dos sentidos.

El problema

Entrada y salida

Varios casos; cada uno abre con n y m y sigue con las m parejas `i j`. La entrada termina con $n = m = 0$. Por caso se imprime el número de caso, una línea en blanco, cada calle como la pareja orientada `i j`, y una línea con #. Las calles pueden ir en cualquier orden, y cualquier orientación que cumpla se acepta.

El dato del enunciado que decide todo

Con todas las calles de doble vía se puede ir de cualquier intersección a cualquier otra. El grafo es conexo.

¿Entendimos el problema?

En una frase

Darle un solo sentido al mayor número de calles, sin que ninguna intersección quede incomunicada de otra.

Entradas

Dos enteros, n y m, y en seguida m parejas. El primer caso de ejemplo trae $n = 7$, $m = 10$ y las calles 1–2, 1–3, 2–4, 3–4, 4–5, 4–6, 5–7, 6–7, 2–5, 3–6.

Salidas

El número de caso, una línea en blanco, una línea por cada sentido habilitado y una línea con #. Para el primer caso el ejemplo imprime diez: `1 2`, `2 4`, `3 1`, `3 6`, `4 3`, `5 2`, `5 4`, `6 4`, `6 7`, `7 5`.

¿Entendimos el problema?

El caso a mano

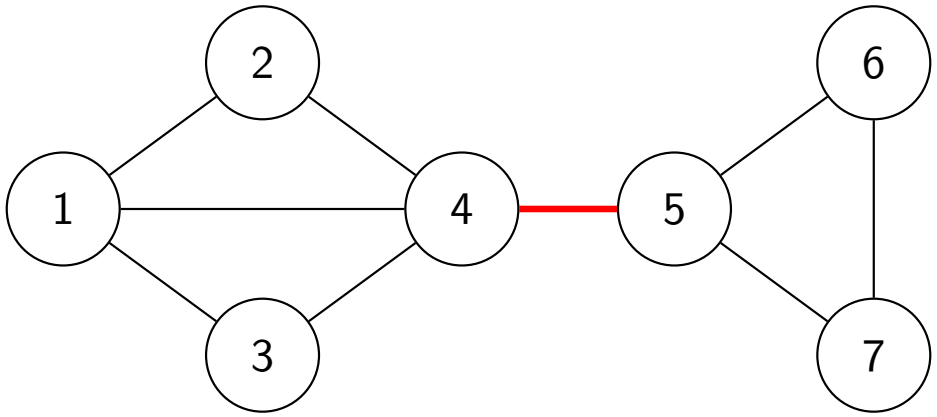
Cada línea de la salida se busca en la lista de calles. `1 2` es la calle 1–2. `3 1` es la 1–3, leída de 3 a 1. `5 2` es la 2–5, y `4 3` es la 3–4. Siguiendo así con las diez líneas se cubren las diez calles sin repetir ninguna: en este caso todas quedaron de una vía.

El segundo caso trae nueve calles y su salida ocupa diez líneas. Ahí aparecen `4 5` y `5 4`: la calle 4–5 en los dos sentidos.

Diez líneas para nueve calles

¿Por qué el segundo caso gasta diez líneas en nueve calles, y el primero diez en diez? La calle repetida es la 4–5. ¿Qué tiene de distinta?

Qué es el grafo



El segundo caso de ejemplo

Siete intersecciones y nueve calles: 1–2, 1–3, 1–4, 2–4, 3–4, 4–5, 5–6, 5–7, 7–6.

Qué es el grafo

La traducción

Vértices, las intersecciones. Aristas no dirigidas, las calles. Convertir una calle en una vía es orientar una arista; que desde cada intersección se llegue a todas las demás es que el grafo dirigido resultante sea fuertemente conexo.

La pregunta antes del algoritmo

En este dibujo, ¿qué calle no se puede orientar? Y ¿por qué las otras ocho sí?

Un puente no se puede orientar

Teorema

Si (u, v) es puente de G, ninguna orientación que le dé un solo sentido a (u, v) deja el grafo dirigido fuertemente conexo.

Demostración

Se procede de forma directa, partiendo el grafo por la arista.

Desarrollo

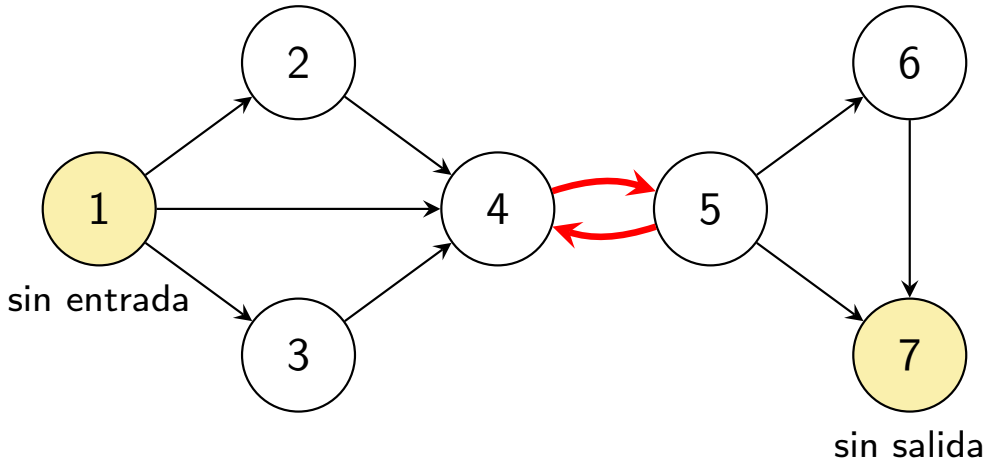
Al quitar (u, v) el grafo queda en dos pedazos, U con u y V con v, sin ninguna arista entre ellos. Todo camino de v a u en G usa (u, v), porque es la única arista que cruza. Si se orienta $u \to v$, desde v no hay forma de volver a u; si se orienta $v \to u$, desde u no hay forma de llegar a v.

Un puente no se puede orientar

Conclusión

Por lo tanto, se puede concluir que un puente tiene que quedar en doble vía, y las calles que el enunciado llama no convertibles son exactamente los puentes.

Orientar por el número de la intersección



El intento

El puente 4–5 queda en doble vía, que es lo único que ya se sabe. Las otras ocho calles reciben un sentido cualquiera, y el más fácil de escribir es de la intersección menor a la mayor.

Orientar por el número de la intersección

Lo que se rompe

Al 1 no entra nada: sus tres calles, 1–2, 1–3 y 1–4, salieron todas. Del 7 no sale nada: sus dos calles, 5–7 y 6–7, entraron las dos. Nadie llega al 1, y quien llegue al 7 se queda ahí.

Tapar un agujero abre otro

Voltear la 1–2 le da entrada al 1, pero entonces las dos calles del 2 salen de él y el 2 se queda sin entrada. Voltear la 6–7 le da salida al 7 y deja al 6 sin salida. El agujero se mueve.

Orientar por el número de la intersección

Sin ciclos no hay regreso

Orientando siempre del número menor al mayor, todo camino dirigido aumenta el número: no hay ciclos. Y en un grafo dirigido sin ciclos nadie vuelve al punto de partida. El sentido de una calle no se puede escoger mirándola sola, y el orden de los números no sabe nada de la ciudad. ¿Qué recorrido sí sabe por dónde se llegó a cada intersección?

Las demás sí, y la profundidad las orienta

Teorema

Sea G conexo y sea r un vértice. Oriéntense las aristas de árbol de la profundidad desde r del padre al hijo, las de retroceso del descendiente al ancestro, y cada puente en los dos sentidos. El grafo dirigido que resulta es fuertemente conexo.

Demostración

Se procede de forma directa para la ida desde r, y por contradicción sobre el vértice de menor d que no logra volver.

Las demás sí, y la profundidad las orienta

Desarrollo, la ida

Toda arista de árbol quedó orientada del padre al hijo, así que el camino de árbol de r a cualquier x es un camino dirigido. Desde r se llega a todos.

Las demás sí, y la profundidad las orienta

Desarrollo, la vuelta

Sea S el conjunto de vértices desde los que se llega a r. Si $S \neq V$, tómese $v \notin S$ con $v.d$ mínimo, y sea p su padre. Hay dos casos.
Si (p, v) es puente, quedó en doble vía y existe $v \to p$, con $p.d < v.d$.
Si no lo es, entonces $v.low \leq p.d$, y por la definición de *low* hay un descendiente u de v con un retroceso (u, w) tal que $w.d = v.low \leq p.d < v.d$. Bajando por aristas de árbol de v a u, todas orientadas hacia abajo, y cruzando $u \to w$, se llega a w.
En los dos casos v alcanza un vértice de d menor que $v.d$. Ese vértice está en S, por la minimalidad de $v.d$ o porque es r. Entonces $v \in S$, en contra de lo supuesto.

Las demás sí, y la profundidad las orienta

Conclusión

Por lo tanto, se puede concluir que desde r se llega a todos y desde todos se llega a r: el grafo dirigido es fuertemente conexo. Como solo los puentes quedaron en doble vía y ninguno podía dejar de estarlo, la orientación convierte el máximo número de calles.

Atribución

El resultado es de Robbins, 1939: un grafo conexo admite una orientación fuertemente conexa si y solo si no tiene puentes.

El algoritmo

Orientar(G)

1. Para cada vértice s sin descubrir, ejecutar *OrientarDesde*(s)

OrientarDesde(u)

1. Asignar $u.d$ y $u.low$
2. Para cada vecino v de u por la arista i:
3. si v no ha sido descubierto: emitir (u, v), *OrientarDesde*(v), $u.low = \text{mín}(u.low, v.low)$ y, si $v.low > u.d$, emitir también (v, u)
4. si no, y i no es la arista de entrada y $v.d < u.d$: emitir (u, v) y $u.low = \text{mín}(u.low, v.d)$

El algoritmo

Por qué $v.d < u.d$ en el paso 4

Una arista que no es de árbol se ve desde los dos extremos. Desde el descendiente el otro tiene d menor, y desde el ancestro, mayor. La comparación deja pasar una sola de las dos miradas, y la salida trae cada calle una sola vez.

Lo que cuesta

Una profundidad por caso, $\Theta(n + m)$ en tiempo y $\Theta(n + m)$ de memoria. Con $n \leq 1000$ cada caso es inmediato; lo que suma es la cantidad de casos, así que la salida se arma en una lista y se imprime de un golpe.

El código

```
def orientar_aux(G, u, entrada, reloj, d, low, salida):
    descubrir(u, reloj, d, low)
    for v, i in G[u]:
        if d[v] == 0:
            salida.append((u, v))
            orientar_aux(G, v, i, reloj, d, low, salida)
            low[u] = min(low[u], low[v])
            if low[v] > d[u]:
                salida.append((v, u))
        elif i != entrada and d[v] < d[u]:
            salida.append((u, v))
            low[u] = min(low[u], d[v])
```

El código

```
def orientar(G):
    d = {}
    low = {}
    for u in G:
        d[u] = 0
        low[u] = 0
    reloj = [0]
    salida = []
    for s in G:
        if d[s] == 0:
            orientar_aux(G, s, -1, reloj, d, low, salida)
    return salida
```

Las tres emisiones

La de árbol, al bajar. La de retroceso, en el `elif`. Y la vuelta del puente, al subir, cuando `low[v] > d[u]`.

La lectura y la salida

```
def leer_pareja(datos, pos):
    # Los dos enteros desde pos, y la posicion siguiente.
    return int(datos[pos]), int(datos[pos + 1]), pos + 2
```

La lectura y la salida

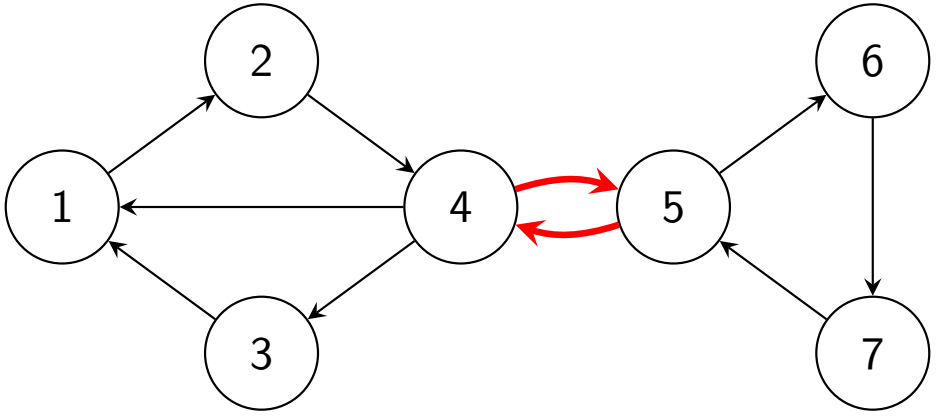
```
def main():
    datos = stdin.read().split()
    lineas = []
    caso = 1
    n, m, pos = leer_pareja(datos, 0)
    while n != 0 or m != 0:
        aristas = []
        for _ in range(m):
            a, b, pos = leer_pareja(datos, pos)
            aristas.append((a, b))
        G = construir(range(1, n + 1), aristas)
        lineas.append(str(caso))
        lineas.append("")
        for u, v in orientar_con_pila(G):
            lineas.append(str(u) + " " + str(v))
        lineas.append("#")
        caso = caso + 1
        n, m, pos = leer_pareja(datos, pos)
    print("\n".join(lineas))
```

La lectura y la salida

Tres detalles del formato

- Después del número de caso va una línea en blanco.
- Cada caso cierra con una línea que trae solo #.
- Las intersecciones van de 1 a n, así que el diccionario se arma con `range(1, n + 1)`.

El segundo caso, orientado



Diez líneas para nueve calles

El único puente es 4–5, y sale en los dos sentidos. Las otras ocho calles salen una vez cada una.

El segundo caso, orientado

Paso	Qué emite	Por qué
$1 \to 2 \to 4$	`1 2`, `2 4`	aristas de árbol
4 mira el 1	`4 1`	retroceso, $1.d < 4.d$
$4 \to 3$	`4 3`	arista de árbol
3 mira el 1	`3 1`	retroceso
$4 \to 5 \to 6 \to 7$	`4 5`, `5 6`, `6 7`	aristas de árbol
7 mira el 5	`7 5`	retroceso
5 termina	`5 4`	$5.low = 5 > 4.d = 3$: puente

La comprobación

Desde el 5 se llega al 4 por el puente y de ahí al 1; desde el 1 se vuelve al 5 por $1 \to 2 \to 4 \to 5$. El triángulo 5–6–7 gira en un sentido y el 1–2–4 en el otro.

UVa 10765 — Doves and Bombs

El problema

UVa 10765 — Doves and Bombs

`https://vjudge.net/problem/UVA-10765`
Envío: `https://vjudge.net/problem/UVA-10765`

Lo que pide

Una red conexa de n estaciones ($3 \leq n \leq 10000$) unidas por vías. El *valor paloma* de una estación es en cuántos pedazos queda la red al bombardearla, que es cuántas palomas hacen falta para que un mensaje llegue a todas las estaciones que siguen en pie. Hay que listar las m estaciones de mayor valor.

El problema

Entrada y salida

Cada caso abre con n y m, sigue con las parejas `x y` de las vías y cierra con una línea `-1 -1`. La entrada termina con $n = m = 0$. Por caso se imprimen exactamente m líneas con la estación y su valor, ordenadas por valor decreciente y, a igual valor, por número creciente; después, una línea en blanco.

Por qué el valor es el número de pedazos

Una paloma entra a un pedazo y de ahí el mensaje sigue en tren. Cada pedazo necesita la suya, y una sola no alcanza para dos pedazos porque la paloma no vuelve a volar.

¿Entendimos el problema?

En una frase

Para cada estación, contar en cuántos pedazos queda la red si esa estación cae, y listar las m que dejan más pedazos.

Entradas

n y m, y en seguida las vías, una por línea, hasta un `-1 -1`. El caso de ejemplo trae $n = 8$, $m = 4$ y las ocho vías 0–4, 1–2, 2–3, 2–4, 3–5, 3–6, 3–7, 6–7.

Salidas

Exactamente m líneas con la estación y su valor. En el ejemplo son cuatro: `2 3`, `3 3`, `4 2` y `0 1`.

¿Entendimos el problema?

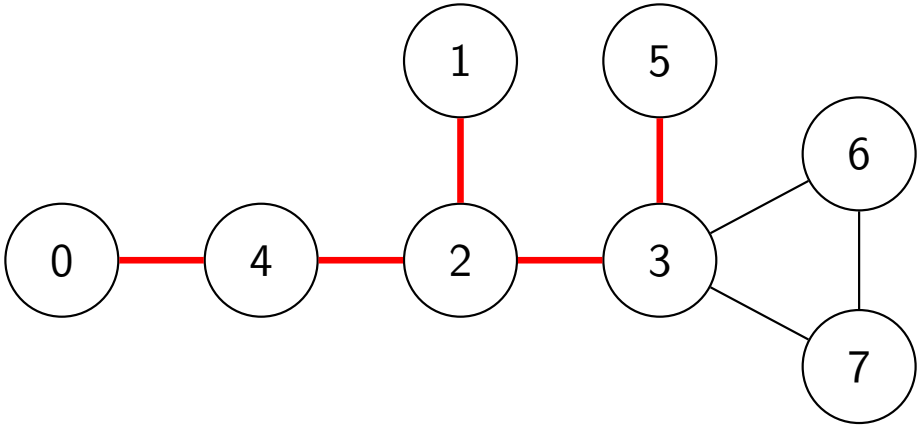
El caso a mano

Sin el 2 la red queda en $\{0, 4\}$, $\{1\}$ y $\{3, 5, 6, 7\}$: tres pedazos. Sin el 3 queda en $\{0, 1, 2, 4\}$, $\{5\}$ y $\{6, 7\}$: tres otra vez. Sin el 4 queda en $\{0\}$ y el resto: dos. Las otras cinco estaciones se pueden quitar sin partir nada, así que valen 1.

Cuatro líneas para ocho estaciones

La vía 0–4 es la única que toca al 0, y aun así el 0 vale 1 mientras el 4 vale 2. ¿Por qué los dos extremos de una misma vía no valen lo mismo? Y de las cinco estaciones que valen 1 se imprime una sola: ¿por qué la cuarta línea es `0 1` y no `1 1`?

Qué es el grafo



El caso de ejemplo

Ocho estaciones y ocho vías: 0–4, 1–2, 2–3, 2–4, 3–5, 3–6, 3–7, 6–7. Los puentes van en rojo.

Qué es el grafo

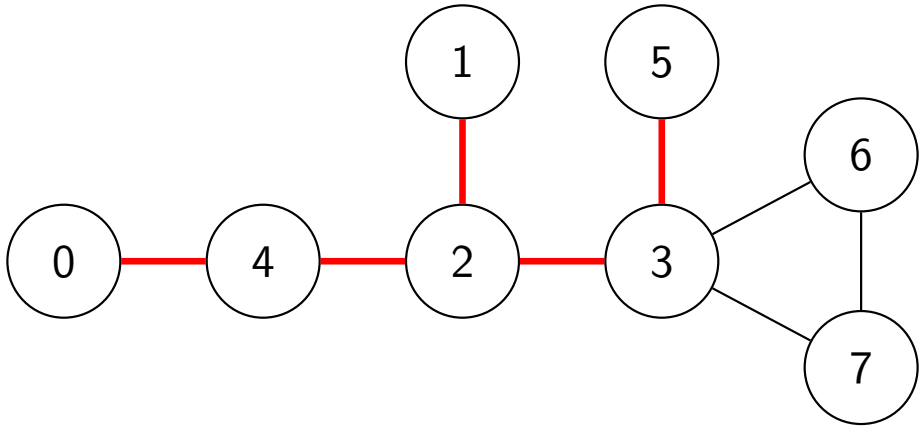
La traducción

Vértices, las estaciones, numeradas de 0 a $n-1$. Aristas no dirigidas, las vías. Bombardear una estación es quitar un vértice, así que el corte es por vértice: el valor paloma de v es el número de componentes de $G-v$.

La pregunta antes del algoritmo

En el dibujo, ¿cuánto vale el 3? ¿Y el 2? ¿Y el 6? Una de las tres respuestas es 1, y conviene ver por qué.

¿En cuántos pedazos queda la red sin el 3?



Lo que desaparece

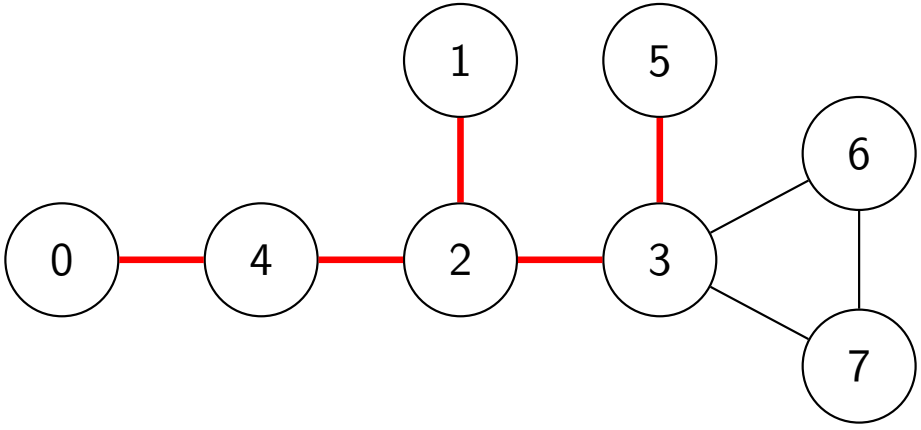
Se borra el 3 y con él las cuatro vías que lo tocan: 2–3, 3–5, 3–6 y 3–7. Quedan siete estaciones y cuatro vías.

¿En cuántos pedazos queda la red sin el 3?

Dos números a mano

El grado del 3 es 4. Y bajando desde el 0, la profundidad recorre 0, 4, 2, y del 2 salen el 1 y el 3; del 3 bajan el 5 y el 6, y del 6 el 7, de modo que el 3 tiene dos hijos. ¿Cuál de los dos números, el 4 o el 2, tiene algo que ver con la respuesta?

Tres pedazos



Cuáles son

$\{5\}$, que colgaba solo del 3. $\{6, 7\}$, que conserva su vía 6–7 y perdió las dos que lo sostenían. Y $\{0, 1, 2, 4\}$, el lado por donde la profundidad llegó al 3.

Tres pedazos

De dónde sale el tres

Los hijos del 3 son dos, el 5 y el 6, y los dos subárboles se desprendieron: $1 + 2 = 3$, donde ese 1 es el lado que contiene al 0. El grado 4 no sirve para contar: la vía 2–3 sube al pedazo de arriba, y la 3–6 y la 3–7 llegan las dos al mismo pedazo de abajo.

El triángulo no salva al 6 ni al 7

El 3, el 6 y el 7 forman triángulo, y el par 6–7 se desprende igual: las dos vías que lo unían al resto iban al 3. Un ciclo aguanta la caída de una vía, no la del vértice que lo cierra.

De contar componentes a contar hijos

Teorema

Sea G conexo y sea T un árbol de la profundidad con raíz r. El número de componentes de $G - v$ es

$$c(v) = \begin{cases} |\{\textit{hijos de } r \textit{ en } T\}| & \textit{si } v = r, \\ 1 + |\{w \textit{ hijo de } v : w.low \geq v.d\}| & \textit{si } v \neq r. \end{cases}$$

Demostración

Se procede de forma directa, contando qué queda pegado al quitar v: los subárboles de sus hijos, por un lado, y lo que está fuera del subárbol de v, por el otro.

De contar componentes a contar hijos

Desarrollo, los subárboles de los hijos

Todo vértice de $G - v$ está en el subárbol de algún hijo de v o fuera del subárbol de v. El subárbol de un hijo w se mantiene conexo por sus propias aristas de árbol. Entre los subárboles de dos hijos distintos no hay ninguna arista, porque sería cruzada y en un grafo no dirigido no las hay (Teorema 22.10). Lo único que puede pegarlos a algo es un retroceso que salte por encima de v, y eso pasa exactamente cuando $w.low < v.d$.

De contar componentes a contar hijos

Desarrollo, lo de afuera

Con $v \neq r$, lo que está fuera del subárbol de v no es vacío: contiene a r y al padre de v. Y es un solo pedazo, porque ninguno de sus vértices es descendiente de v y su camino de árbol hasta r no pasa por v. Los subárboles con $w.low < v.d$ se le pegan y los de $w.low \geq v.d$ quedan sueltos, uno por uno. De ahí el 1 más la cuenta.

Con $v = r$ no hay nada afuera. Un retroceso desde el subárbol de un hijo solo puede ir a un ancestro, y el único ancestro es r, que ya no está. Cada subárbol queda suelto y los pedazos son tantos como hijos.

De contar componentes a contar hijos

Conclusión

Por lo tanto, se puede concluir que al quitar v quedan tantos pedazos como hijos tiene la raíz, si v es la raíz, y uno más el número de hijos con $w.low \geq v.d$ en los demás casos. De paso, $c(v) \geq 2$ exactamente cuando v es punto de articulación: los dos criterios del recuento son ese caso de esta cuenta.

En el dibujo

El 3 vale 3: se desprenden el 5 y el par 6–7, y queda el resto con el 2. El 2 vale 3: se desprenden el 1 y todo lo que cuelga del 3, y queda el lado del 4. El 6 vale 1: está en el triángulo con el 3 y el 7, y al quitarlo nada se parte.

El código

```
def paloma_aux(G, u, entrada, raiz, reloj, d, low, valor):
    descubrir(u, reloj, d, low)
    if u == raiz:
        valor[u] = 0
    for v, i in G[u]:
        if d[v] == 0:
            paloma_aux(G, v, i, raiz, reloj, d, low, valor)
            low[u] = min(low[u], low[v])
            if low[v] >= d[u] or u == raiz:
                valor[u] = valor[u] + 1
        elif i != entrada:
            low[u] = min(low[u], d[v])
```

El código

```
def valores_paloma(G):
    d = {}
    low = {}
    valor = {}
    for u in G:
        d[u] = 0
        low[u] = 0
        valor[u] = 1
    reloj = [0]
    for s in G:
        if d[s] == 0:
            paloma_aux(G, s, -1, s, reloj, d, low, valor)
    return valor
```

El código

De dónde sale el 1

`valor[u]` arranca en 1: es el pedazo que se queda con el padre. Encima se suma un hijo por cada `low[v] >= d[u]`. La raíz no tiene padre, así que se pone en 0 al descubrirla y suma todos sus hijos, que es lo que dice la otra rama del teorema.

El orden de la salida

```
def mejores(valor, m):
    # Valor decreciente y, a igual valor, numero creciente.
    pares = []
    for v in valor:
        pares.append((-valor[v], v))
    pares.sort()
    lineas = []
    for k in range(m):
        peso, v = pares[k]
        lineas.append(str(v) + " " + str(-peso))
    return lineas
```

El orden de la salida

```
def leer_pareja(datos, pos):
    # Los dos enteros desde pos, y la posicion siguiente.
    return int(datos[pos]), int(datos[pos + 1]), pos + 2
```

El orden de la salida

```
def main():
    datos = stdin.read().split()
    bloques = []
    n, m, pos = leer_pareja(datos, 0)
    while n != 0 or m != 0:
        aristas = []
        a, b, pos = leer_pareja(datos, pos)
        while a != -1 or b != -1:
            aristas.append((a, b))
            a, b, pos = leer_pareja(datos, pos)
        G = construir(range(n), aristas)
        bloques.append("\n".join(mejores(valores_paloma_con_pila(G), m)))
        n, m, pos = leer_pareja(datos, pos)
    for bloque in bloques:
        print(bloque)
        print()
```

El orden de la salida

El signo menos

El orden pedido es decreciente por valor y creciente por número. Las parejas se arman como $(-valor, v)$ y se ordenan de forma corriente: el signo invierte el primer campo y deja el segundo como está.

Paso a paso, arrancando en 0

Vértice	0	4	2	1	3	5	6	7
d	1	2	3	4	5	6	7	8
low	1	2	3	4	5	6	5	5

La rama

$0 \to 4 \to 2$, y de 2 salen el 1 y el 3; de 3 salen el 5 y el 6, y de 6 el 7. El único retroceso es 7–3, con $3.d = 5$: baja $7.low$ y $6.low$ a 5.

Paso a paso, arrancando en 0

Hijo w de v	$w.low$ vs $v.d$	¿Suma?	$valor[v]$
1 de 2	$4 \geq 3$	sí	2
5 de 3	$6 \geq 5$	sí	2
7 de 6	$5 \geq 7$	no	1
6 de 3	$5 \geq 5$	sí	3
3 de 2	$5 \geq 3$	sí	3
2 de 4	$3 \geq 2$	sí	2
4 de 0	raíz con un hijo	no	1

La salida con $m = 4$

Los valores son 2:3, 3:3, 4:2 y 1 para las demás. Las cuatro primeras líneas quedan `2 3`, `3 3`, `4 2`, `0 1`.

Paso a paso, arrancando en 0

Una fila que no cuadra

El 6 tiene un hijo, el 7, y se queda en 1. El 3 tiene dos hijos y llega a 3. Los tres están en el mismo triángulo. ¿Qué tiene el 7 que no tengan el 5 ni el 6?

Paso a paso, arrancando en 0

El 7 está en la tabla y no suma

$7.\mathit{low} = 5$ es menor que $6.d = 7$: el retroceso 7–3 salta por encima del 6 y lo deja fuera del corte. Es el mismo número que sí hace sumar al 3, leído desde otro padre.

Lo que cuesta

Una profundidad por caso, $\Theta(n + m)$, más $O(n \log n)$ por el ordenamiento de la salida. Las estaciones tienen grado a lo sumo 10, de modo que $m = O(n)$.

Capital

El problema

Problema A, *capital*

N ciudades y M carreteras de una sola vía, $N \leq 10^5$ y $M \leq 2 \cdot 10^5$. La capital tiene que ser alcanzable *desde todas* las demás ciudades. Se piden todas las candidatas, en orden creciente.

La muestra

Con 4 ciudades y las carreteras $1 \to 2$, $3 \to 2$, $4 \to 3$ y $2 \to 1$, la salida es 2 y la lista 1 2.

¿Entendimos el problema?

Entradas y salidas

Entra un grafo dirigido. Sale cuántas ciudades cumplen y cuáles, ordenadas.

La muestra a mano

Desde el 4 se llega al 3, al 2 y al 1. Desde el 3, al 2 y al 1. Entre el 1 y el 2 se va y se vuelve. Al 3 no llega nadie salvo el 4, y al 4 no llega nadie: ninguno de los dos puede ser capital. Quedan el 1 y el 2.

¿Por qué la respuesta trae dos y no una?

Porque 1 y 2 se alcanzan entre sí: cualquiera de los dos sirve, y el enunciado pide todas las candidatas.

Qué es el grafo

El de componentes, otra vez

Los vértices son las ciudades y las aristas las carreteras, dirigidas. Se contraen los componentes fuertemente conexos y queda G^{SCC}, que es un DAG.

Quién puede ser capital

Una ciudad es candidata exactamente cuando su componente es el *sumidero* de G^{SCC}: el que no tiene aristas de salida. Si del componente sale una arista, de allá no se vuelve, y las ciudades del otro lado se quedan sin camino a la capital.

Si hay más de un sumidero, no hay capital

Dos sumideros no se alcanzan entre sí, así que ninguna ciudad sirve. En un DAG con un solo sumidero, ese sumidero se alcanza desde todos.

El algoritmo

El algoritmo, en tres pasos

1. Calcular los componentes fuertemente conexos.
2. Marcar los componentes con alguna arista que salga hacia otro.
3. Si queda exactamente un componente sin marcar, imprimir sus vértices ordenados; si no, no hay candidatas.

Lo que cuesta

Los componentes en $\Theta(V + E)$, el barrido de aristas en $\Theta(E)$ y el orden final en $O(V \log V)$.

El algoritmo

Quién calcula los componentes

Tarjan, con el mismo *low* de los puentes. Lo que cambia es que el grafo va dirigido: el recorrido lleva una pila con los vértices abiertos, una arista baja el *low* solo si apunta a un vértice que sigue en esa pila, y el vértice que termina con $low[u] = d[u]$ cierra su componente.

El grafo y la pila

```
def construir(vertices, aristas):
    # G[u] guarda los vecinos de u; cada carretera entra en un solo
    ↪  sentido.
    G = {}
    for u in vertices:
        G[u] = []
    for a, b in aristas:
        G[a].append(b)
    return G

def descubrir(u, reloj, d, low):
    reloj[0] = reloj[0] + 1
    d[u] = reloj[0]
    low[u] = reloj[0]
```

El grafo y la pila

Una sola vía

La carretera $a \to b$ entra una vez, en la lista de a. Por eso el vecino va solo, sin el identificador de arista que pedían los puentes: aquí nadie tiene que excluir la arista por la que bajó.

El grafo y la pila

```
def cerrar(u, pila, en_pila, comp, total):
    # u quedo como raiz de su componente: lo que esta encima de u en
    ↪  la pila,
    # y u mismo, son un componente.
    ultimo = None
    while ultimo != u:
        ultimo = pila.pop()
        en_pila[ultimo] = False
        comp[ultimo] = total[0]
    total[0] = total[0] + 1
```

Qué sale de la pila

Lo que entró después de u y todavía no tiene componente, más u mismo. Esos vértices se alcanzan entre sí, así que reciben el mismo número.

El código de los componentes

```
def scc_aux(G, u, reloj, d, low, pila, en_pila, comp, total):
    # La pila guarda los vertices abiertos cuyo componente aun no se
    ↪  cierra.
    descubrir(u, reloj, d, low)
    pila.append(u)
    en_pila[u] = True
    for v in G[u]:
        if d[v] == 0:
            scc_aux(G, v, reloj, d, low, pila, en_pila, comp, total)
            low[u] = min(low[u], low[v])
        elif en_pila[v]:
            low[u] = min(low[u], d[v])
    if low[u] == d[u]:
        cerrar(u, pila, en_pila, comp, total)
```

El código de los componentes

El `elif`

Una arista hacia un vértice ya visitado baja el *low* solo cuando ese vértice sigue en la pila. Si su componente ya se cerró, de allá no se vuelve y la arista no dice nada de u.

El código de los componentes

```
def componentes(G):
    # comp[v]: el componente de v, y cuantos componentes hay.
    d = {}
    low = {}
    en_pila = {}
    comp = {}
    for u in G:
        d[u] = 0
        low[u] = 0
        en_pila[u] = False
        comp[u] = -1
    reloj = [0]
    total = [0]
    pila = []
    for s in G:
        if d[s] == 0:
            scc_aux(G, s, reloj, d, low, pila, en_pila, comp, total)
    return comp, total[0]
```

El código de los componentes

Lo que queda al terminar

`comp`, con el componente de cada ciudad, y cuántos componentes hay. `componentes_con_pila` hace lo mismo sin recursión y viene en los archivos de código de la clase; con $N = 10^5$ es la que se envía.

El código del sumidero

```
def salidas(G, comp, total):
    # Cuantas carreteras salen de cada componente hacia otro
    ↪  componente.
    salen = {}
    for c in range(total):
        salen[c] = 0
    for u in G:
        for v in G[u]:
            if comp[u] != comp[v]:
                salen[comp[u]] = salen[comp[u]] + 1
    return salen
```

Sobre las etiquetas, no sobre el grafo contraído

El DAG no se construye. Basta contar, para cada componente, las carreteras que terminan en otro componente; el que queda en cero es un sumidero.

El código del sumidero

```
def capitales(G):
    # Los vertices del unico componente sumidero, en orden creciente.
    ↪  La lista
    # queda vacia cuando hay mas de un sumidero.
    comp, total = componentes_con_pila(G)
    salen = salidas(G, comp, total)
    sumideros = []
    for c in range(total):
        if salen[c] == 0:
            sumideros.append(c)
    elegidas = []
    if len(sumideros) == 1:
        for u in G:
            if comp[u] == sumideros[0]:
                elegidas.append(u)
    elegidas.sort()
    return elegidas
```

El código del sumidero

Las dos respuestas posibles

Con un solo sumidero salen sus ciudades en orden creciente. Con dos o más la lista queda vacía, y de ahí sale el `0` del enunciado.

La lectura y la salida

```
def leer_pareja(datos, pos):
    # Los dos enteros desde pos, y la posicion siguiente.
    return int(datos[pos]), int(datos[pos + 1]), pos + 2
```

La lectura y la salida

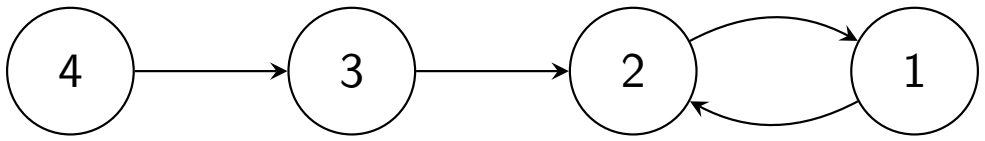
```
def main():
    datos = stdin.read().split()
    lineas = []
    pos = 0
    while pos + 1 < len(datos):
        n, m, pos = leer_pareja(datos, pos)
        aristas = []
        for _ in range(m):
            a, b, pos = leer_pareja(datos, pos)
            aristas.append((a, b))
        elegidas = capitales(construir(range(1, n + 1), aristas))
        lineas.append(str(len(elegidas)))
        if len(elegidas) > 0:
            numeros = []
            for u in elegidas:
                numeros.append(str(u))
            lineas.append(" ".join(numeros))
    print("\n".join(lineas))
```

La lectura y la salida

Un caso tras otro

La entrada trae casos hasta que se agotan los números. Las líneas se acumulan y se imprimen de una sola vez, porque un `print` por ciudad con $N = 10^5$ se paga en tiempo.

Paso a paso, la muestra



Las cuatro carreteras

Del 4 al 3, del 3 al 2, y entre el 1 y el 2 se va y se vuelve. Nadie entra al 4.

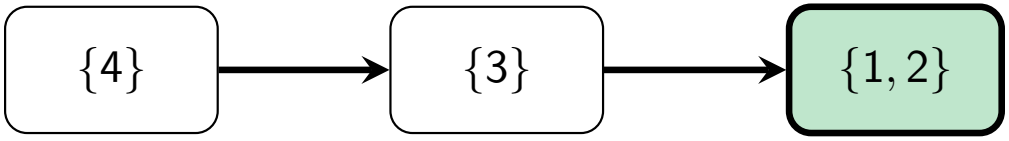
Paso a paso, la muestra

Componente	Vértices	Carreteras que salen de él	¿Sumidero?
$\{4\}$	4	$4 \to 3$	no
$\{3\}$	3	$3 \to 2$	no
$\{1, 2\}$	1, 2	ninguna	sí

Por qué salen tres componentes y no cuatro

El 1 y el 2 se alcanzan entre sí, así que van juntos. Al 3 se llega desde el 4, pero del 3 no hay vuelta: cada uno queda solo. Las dos carreteras que cruzan de un componente a otro son $4 \to 3$ y $3 \to 2$.

Paso a paso, la muestra



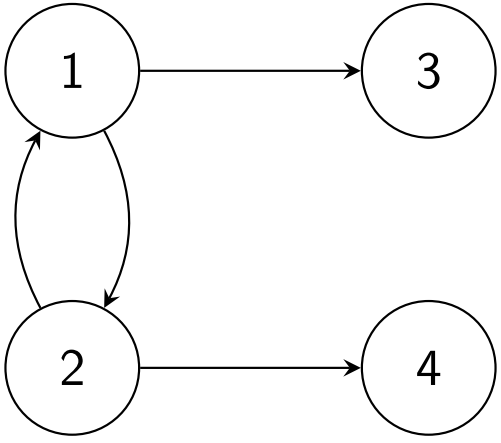
La respuesta

Un único componente sin carreteras de salida, el $\{1, 2\}$. Se imprime cuántas ciudades tiene y cuáles, ordenadas: 2 y la línea `1 2`.

Desde el 4 hasta la capital

$4 \to 3 \to 2$, y del 2 al 1. Las cuatro ciudades llegan al 1 y al 2, cada una por su camino, que es lo que el enunciado pide de una capital.

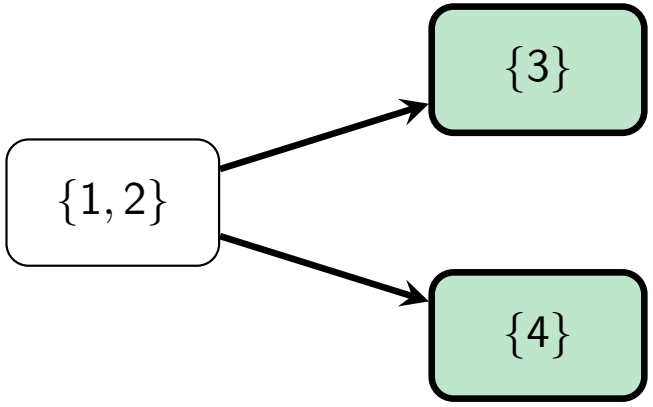
Un caso con dos sumideros



La entrada

Cuatro ciudades y las carreteras $1 \to 2$, $2 \to 1$, $1 \to 3$ y $2 \to 4$.

Un caso con dos sumideros



Componente	Vértices	Carreteras que salen de él	¿Sumidero?
$\{1, 2\}$	1, 2	$1 \to 3$, $2 \to 4$	no
$\{3\}$	3	ninguna	sí
$\{4\}$	4	ninguna	sí

Un caso con dos sumideros

Ninguna ciudad sirve

Del 3 no sale ninguna carretera, así que el 3 no alcanza al 4. Del 4 tampoco sale ninguna, así que el 4 no alcanza al 3. El 1 y el 2 quedan descartados porque ni el 3 ni el 4 llegan a ellos. La salida es `0`.

El segundo sumidero cierra el caso

En cuanto aparecen dos componentes sin salida no hay que mirar nada más: esos dos no se alcanzan entre sí, y lo que una capital tiene que hacer es alcanzarse desde los dos.

La otra forma de resolverlo

Gabow: dos pilas en vez de *low*

El titular calcula los componentes con *Gabow*. Donde Tarjan guarda un número por vértice, Gabow lleva dos pilas: `pilaS` con los vértices abiertos, que es la misma de Tarjan, y `pilaP` con los candidatos a representante del componente.

La otra forma de resolverlo

```
for(int i = 0; i < adj[v].size(); i++){
  w = adj[v][i];
  if(visitado[w] == -1)
    gabowAux(w);
  else if(sccInd[w] == -1){
    while(visitado[pilaP.top()] > visitado[w])
      pilaP.pop();
  }
}
```

Al bajar, dentro de gabowAux

La arista hacia un vértice abierto saca de `pilaP` todo lo que se descubrió después de él. Ese es el efecto de bajar el *low*: lo que queda en `pilaS` por encima de la cima de `pilaP` va a parar al mismo componente que esa cima.

La otra forma de resolverlo

```
if(v == pilaP.top()){
  numSCC++;
  while(pilaS.top() != v){
    sccInd[pilaS.top()] = numSCC - 1;
    pilaS.pop();
  }
  sccInd[pilaS.top()] = numSCC - 1;
  pilaS.pop();
  pilaP.pop();
}
```

Al cerrar

El vértice que sobrevive en la cima de `pilaP` es el que cierra el componente, y la pregunta `v == pilaP.top()` es la condición `low[u] == d[u]`. Lo que se vacía de `pilaS` hasta *v* es lo que `cerrar` saca de la pila en la versión con *low*.

La otra forma de resolverlo

Los mismos componentes

Lo que sale es la misma partición, y el conteo de carreteras que cruzan de un componente a otro no cambia. El archivo completo, en C++, está en los códigos de la clase.

Interplanetary

El problema

La pregunta

Se parte de un vértice y se camina por las aristas. ¿Hasta dónde se llega, si cada puente se puede pasar en un solo sentido?

Lo que entra

Un grafo no dirigido, un entero en cada vértice y el vértice de arranque. El entero de un vértice es su *influencia*.

Cuál es el sentido permitido

Los extremos de un puente caen en dos componentes 2-arista-conexos distintos. Cada componente lleva la suma de las influencias de sus vértices, y el puente se pasa únicamente hacia el componente de suma estrictamente mayor. Las aristas que no son puente se cruzan en los dos sentidos.

¿Entendimos el problema?

En una frase

Listar los vértices a los que se llega desde el de arranque, con los puentes transitables en un solo sentido.

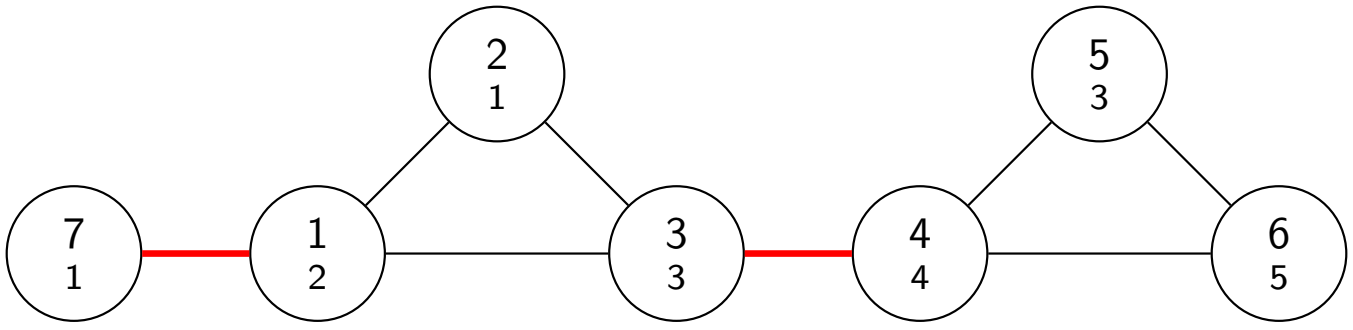
Entradas

n y m; las n influencias, una por vértice y en orden; las m aristas, una pareja por línea; y el vértice de arranque.

Salidas

Una línea con los vértices alcanzados, creciente por la influencia del componente, después por la influencia propia y al final por número.

¿Entendimos el problema?



El caso de trabajo

Siete vértices y ocho aristas: 1–2, 2–3, 1–3, 3–4, 4–5, 5–6, 4–6 y 1–7. El número pequeño debajo de cada vértice es su influencia. Se arranca en el 1.

¿Entendimos el problema?

El caso a mano

Del 1 se va al 2 y al 3 sin tocar ningún puente: los tres son un triángulo y suman $2 + 1 + 3 = 6$. El 4, el 5 y el 6 son otro triángulo y suman $4 + 3 + 5 = 12$. El 7 está solo y vale 1. Del triángulo de arranque al segundo se pasa por el puente 3–4, que va de 6 a 12. El otro puente, 1–7, iría de 6 a 1. La salida es `2 1 3 5 4 6`.

Seis vértices en la salida, y el grafo tiene siete

El 7 está a una sola arista del arranque y se queda afuera. ¿Qué tiene el puente 1–7 que no tenga el 3–4?

Por qué el puente se pasa hacia arriba

La diferencia entre los dos puentes

El 3–4 sube: $6 < 12$. El 1–7 bajaría, y $6 < 1$ es falso. Por eso el 7 no entra en la lista.

Si se pasara en cualquier sentido

Es lo primero que uno escribe: una profundidad corriente, que cruza toda arista que encuentre. Desde el 1 se baja al 7; la única arista del 7 es la que acaba de usarse, así que el recorrido se devuelve por el mismo puente. Con ese criterio salen los siete vértices, las influencias no deciden nada y la respuesta es el grafo entero.

Por qué el puente se pasa hacia arriba

Con la comparación puesta

Cada puente se pasa hacia una suma mayor, de modo que las sumas por donde va el recorrido crecen y ninguno se vuelve a pasar en el otro sentido: $6 < 12$ deja entrar al segundo triángulo y $12 < 6$ no deja volver. El recorrido termina y el conjunto que alcanza queda bien definido.

El algoritmo

Tres pasadas

1. Tarjan marca los puentes.
2. El etiquetado que no los cruza da los componentes 2-arista-conexos, y de paso suma la influencia de cada uno.
3. El recorrido desde el inicio cruza una arista cuando no es puente, o cuando lo es y lleva a un componente de influencia mayor.

Lo que cuesta

Las tres pasadas son profundidades sobre el mismo grafo, $\Theta(V + E)$ cada una. El orden final de la salida, $O(V \log V)$.

Los componentes y su influencia

```
def sumar_aux(G, u, numero, es_puente, influencia, comp, suma):
    # No cruza puentes: lo que alcanza es un componente, y de paso
    ↪  acumula la
    # influencia de cada vertice que etiqueta.
    comp[u] = numero
    suma[numero] = suma[numero] + influencia[u]
    for v, i in G[u]:
        if comp[v] == -1 and i not in es_puente:
            sumar_aux(G, v, numero, es_puente, influencia, comp, suma)
```

El etiquetado de la primera parte, con una suma encima

Es el recorrido que no cruza puentes. La línea de la suma es lo único que se agrega: cada vértice que se etiqueta aporta su influencia al componente.

Los componentes y su influencia

```
def componentes(G, es_puente, influencia):
    # comp[v]: el componente de v; suma[c]: la influencia de todo c.
    comp = {}
    for u in G:
        comp[u] = -1
    suma = []
    total = 0
    for u in G:
        if comp[u] == -1:
            suma.append(0)
            sumar_aux(G, u, total, es_puente, influencia, comp, suma)
            total = total + 1
    return comp, suma
```

Los componentes y su influencia

Las dos cosas que salen

El componente de cada vértice, en `comp`, y la influencia de cada componente, en `suma`. Los puentes llegan ya calculados por `puentes`, el Tarjan de la primera parte de la sesión.

El recorrido que respeta los puentes

```
def se_puede(u, v, i, es_puente, comp, suma):
    # Una arista que no es puente se cruza en los dos sentidos; un
    ↪  puente,
    # solo hacia el componente de influencia estrictamente mayor.
    return i not in es_puente or suma[comp[u]] < suma[comp[v]]

def alcance_aux(G, u, es_puente, comp, suma, visto, llegan):
    visto[u] = True
    llegan.append(u)
    for v, i in G[u]:
        if not visto[v] and se_puede(u, v, i, es_puente, comp, suma):
            alcance_aux(G, v, es_puente, comp, suma, visto, llegan)
```

La condición, aparte

Dentro de un componente se anda libre: ninguna de esas aristas es puente. El puente se cruza solo hacia más influencia, y la desigualdad es estricta, así que un empate lo deja cerrado en los dos sentidos.

El recorrido que respeta los puentes

```
def alcance(G, inicio, es_puente, comp, suma):
    # Los vertices a los que se llega desde inicio.
    visto = {}
    for u in G:
        visto[u] = False
    llegan = []
    alcance_aux(G, inicio, es_puente, comp, suma, visto, llegan)
    return llegan
```

Qué devuelve

Los vértices alcanzados, en el orden en que el recorrido los encontró. Ese no es el orden de la salida: falta ordenarlos.

El orden de la salida

```python
def ordenar(llegan, influencia, comp, suma):
    # Creciente por la influencia del componente, luego por la propia y
    ↪  al
    # final por el numero del vertice.
    claves = []
    for u in llegan:
        claves.append((suma[comp[u]], influencia[u], u))
    claves.sort()
    salida = []
    for total, propia, u in claves:
        salida.append(u)
    return salida
```

Las tres claves

La influencia del componente, la del vértice y el número. El número desempata siempre, así que la línea que sale es única.

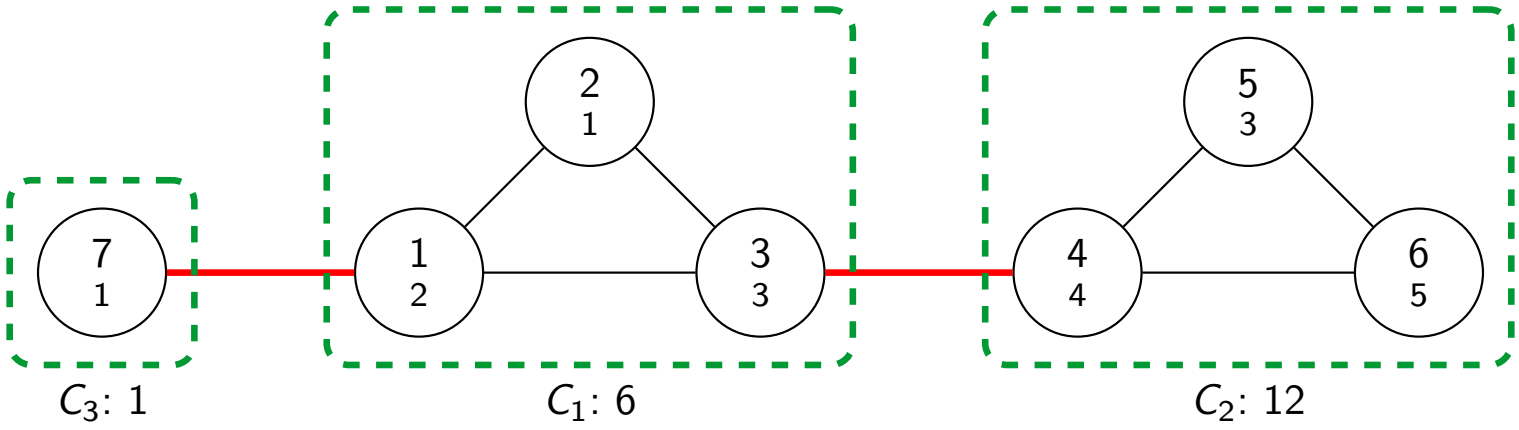
El orden de la salida

```python
def resolver(vertices, aristas, influencia, inicio):
    # Las tres pasadas seguidas. Theta(V+E) en total, mas el orden
    ↪  final.
    G = construir(vertices, aristas)
    es_puente = puentes_con_pila(G)
    comp, suma = componentes_con_pila(G, es_puente, influencia)
    llegan = alcance_con_pila(G, inicio, es_puente, comp, suma)
    return ordenar(llegan, influencia, comp, suma)
```

Las tres pasadas en una sola llamada

Entran por las versiones de pila, que son las que se envían cuando la profundidad puede pasarse del límite de recursión de Python. Las recursivas (`puentes`, `componentes` y `alcance`) están en el mismo archivo y dan lo mismo.

Paso a paso, desde el 1



Paso 1: los puentes

Tarjan marca 1–7 y 3–4. Las otras seis aristas están en un triángulo, y una arista de un ciclo no desconecta nada.

Paso a paso, desde el 1

Componente	Vértices	Influencias	Suma
C_1	1, 2, 3	2, 1, 3	6
C_2	4, 5, 6	4, 3, 5	12
C_3	7	1	1

Paso 2: los componentes

El etiquetado que no cruza puentes deja `comp` en $0, 0, 0, 1, 1, 1, 2$ para los vértices 1 a 7, y `suma` en $[6, 12, 1]$.

Paso a paso, desde el 1

Puente	Desde	Hacia	¿Se pasa?
1–7	C_1: 6	C_3: 1	no, $6 < 1$ es falso
3–4	C_1: 6	C_2: 12	sí, $6 < 12$

Paso 3: el recorrido

Arranca en el 1, recorre C_1 completo, pasa el 3–4 y recorre C_2 completo. El 7 no se visita.

Paso a paso, desde el 1

Vértice	2	1	3	5	4	6
Influencia de su componente	6	6	6	12	12	12
Influencia propia	1	2	3	3	4	5

La salida

Las dos filas de la tabla son las dos primeras claves del orden y el número del vértice desempata. Quedan los seis alcanzados en una línea:
`2 1 3 5 4 6`.

La otra forma de resolverlo

La del titular, en Python

Las tres pasadas son las mismas. Lo que cambia es dónde vive el estado: el grafo, los puentes, las etiquetas y las influencias son arreglos globales de tamaño fijo, y la influencia del componente sube como valor de retorno de la recursión en vez de acumularse en una lista.

La otra forma de resolverlo

```
def getCCAux(u, ind):
  ans = infl[u]
  comps[ind].append(u)
  ccInd[u] = ind
  for v in G[u]:
    if ccInd[v] == -1 and (u, v) not in bridges:
      ans += getCCAux(v, ind)
  return ans
```

La suma sube por el retorno

Cada llamada devuelve la influencia de lo que etiquetó y el padre la agrega a la suya. `getCC` recoge el total de cada componente en `inflComps`.

La otra forma de resolverlo

```
def dfs(u):
  vis[u] = 1
  reach.append(u)
  for w in G[u]:
    if vis[w] == -1 and ((u, w) not in bridges
        or inflComps[ccInd[u]] < inflComps[ccInd[w]]):
      dfs(w)
```

El puente, por pareja de vértices

`bridges` guarda las parejas (u, v) y (v, u) en vez del identificador de la arista, y el padre se excluye por vértice con `w != p[v]`. Con una sola arista entre cada par de vértices las dos formas coinciden. El archivo completo está en los códigos de la clase.

Contenido

1 Antes de empezar

2 Recuento

3 El árbol de puentes

4 Ejercicios resueltos

5 Cómo atacar estos problemas

6 Errores comunes

7 Ejercicios propuestos

8 Referencias

Cinco pasos

Los dos primeros

1. **Leer el grafo.** Quiénes son los vértices y quiénes las aristas, y si viene dirigido o no. Intersecciones y calles; estaciones y vías; ciudades y carreteras de una sola vía.
2. **Decidir qué se corta.** Si lo que el enunciado quita o rompe es un vértice, el criterio es el de articulación ($w.low \geq v.d$ y los hijos de la raíz). Si es una arista, el de puente ($w.low > v.d$). Y si el grafo viene dirigido, lo que se contrae son los componentes fuertemente conexos.

Cinco pasos

Los tres que siguen

3. **Calcular con Tarjan.** Una profundidad, d y low, $\Theta(V + E)$. Con pila explícita si la entrada puede dar una rama de más de mil vértices.
4. **Contraer, si la pregunta no es local.** Si pregunta por caminos, distancias o cuántos puentes se cruzan, se arma el árbol de puentes.
5. **Recorrer lo que quedó.** Sobre el árbol, una profundidad o una amplitud más; sobre el grafo original, una pasada por las aristas.

Cinco pasos

Dónde se para cada uno

UVa 10765 se queda en el paso 3: la respuesta es una cuenta por vértice. UVa 610 llega al paso 3 y emite en la misma pasada. Los problemas que piden distancias llegan al 5.

El paso que más se salta

El 2. Un enunciado que habla de destruir ciudades pide articulaciones; uno que habla de cortar cables o inundar carreteras pide puentes. Resolver el otro da una respuesta razonable que el juez rechaza.

Contenido

1 Antes de empezar

2 Recuento

3 El árbol de puentes

4 Ejercicios resueltos

5 Cómo atacar estos problemas

6 Errores comunes

7 Ejercicios propuestos

8 Referencias

Errores comunes

1. Contar mal los hijos de la raíz

Hijos en el árbol de la profundidad, no vecinos. En el ejemplo de UVa 10765 la raíz 0 tiene un vecino y un hijo, y vale 1; una raíz con tres vecinos de los que dos ya fueron descubiertos también tiene un solo hijo. Contar vecinos infla el valor de la raíz y la lista de salida sale encabezada por la estación equivocada.

2. Usar `low[v] > d[u]` para la articulación

Con la desigualdad estricta se pierden los vértices cuyo hijo tiene un retroceso que llega hasta ellos mismos. En el ejemplo, el 3 dejaría de sumar por su hijo 6, porque $6.\mathit{low} = 5 = 3.d$, y saldría con valor 2 en lugar de 3. El criterio de articulación admite la igualdad; el de puente, no.

Errores comunes

3. Olvidar las aristas repetidas

Dos vías entre las mismas dos estaciones forman un ciclo de longitud dos, así que ninguna de las dos es puente. Con `v != padre` la segunda copia queda descartada por error y la primera sale como puente. Se arregla comparando identificadores de arista: `i != entrada`.

4. Cruzar puentes en el etiquetado

Si el recorrido del paso 2 no revisa `i not in es_puente`, llega al otro lado del puente y dos componentes vecinos se fusionan. El árbol pierde nodos y aristas, el programa no se queja, y las respuestas salen de otro grafo. En el ejemplo de la sesión quedaría un solo nodo en lugar de cuatro.

Errores comunes

5. Reutilizar los arreglos entre casos

Varios casos por entrada. Si d, low, el reloj o el conjunto de puentes se quedan con los valores del caso anterior, el segundo caso sale mal y el primero bien, que es la forma más costosa de buscar el error.

6. Mandar la versión recursiva

Con $n = 10000$ y la red en línea, la profundidad recursiva se cae contra el límite de 1000 marcos de Python. Subirlo con `setrecursionlimit` cambia el `RecursionError` por un desbordamiento de la pila del sistema. Las versiones sin recursión `puentes_con_pila`, `componentes_con_pila`, `orientar_con_pila` y `valores_paloma_con_pila` vienen en los archivos de código de la clase y no tienen ese techo. Son las que se envían al juez.

Errores comunes

7. Emitir dos veces la misma arista de retroceso

En UVa 610, sin la comparación `d[v] < d[u]` cada arista que no es de árbol se emite desde sus dos extremos. La salida trae una calle de más en los dos sentidos y el juez la rechaza, aunque la orientación sea fuertemente conexa.

8. Imprimir dentro del ciclo de casos

Con muchos casos, un `print` por línea es el cuello de botella. Las líneas se acumulan en una lista y se imprimen de un golpe al final.

Contenido

1 Antes de empezar

2 Recuento

3 El árbol de puentes

4 Ejercicios resueltos

5 Cómo atacar estos problemas

6 Errores comunes

7 Ejercicios propuestos

8 Referencias

Ejercicios

1. El árbol de puentes de H, a mano

Sea H el grafo no dirigido con $V = \{0, \ldots, 8\}$ y aristas 0–1, 1–2, 2–0, 2–3, 3–4, 4–2, 4–5, 5–6, 6–7, 7–5, 7–8. Corra la profundidad desde 0, escriba d y *low* de cada vértice, llene la tabla de aristas de árbol con la comparación $w.low > v.d$ y dibuje el árbol de puentes con sus nodos etiquetados por los conjuntos de vértices.

2. La arista doble

A H se le agrega una segunda arista 4–5, paralela a la que ya estaba. Diga qué queda de la lista de puentes, cuántos nodos tiene ahora el árbol, y qué devolvería un programa que excluye la arista de entrada con `v != padre` en lugar de comparar identificadores.

Ejercicios

3. Los dos criterios sobre el mismo grafo

Sobre el H original, liste los puntos de articulación y los puentes. Señale un vértice que sea articulación sin tener ningún puente incidente y explique con d y low por qué pasa.

4. El valor paloma de cada vértice de H

Calcule $c(v)$, el número de componentes de $H - v$, para los nueve vértices, primero quitando cada uno y contando, y después con la fórmula de los hijos. Compruebe que las dos listas coinciden y que $c(v) \geq 2$ exactamente en los puntos de articulación.

Ejercicios

5. Orientar H

Oriente las aristas de H con la regla de UVa 610 arrancando en 0, escriba la lista de parejas que saldría y compruebe que desde el vértice 8 se llega a los otros ocho. Diga cuántas líneas tiene la salida y por qué no son once.

6. La raíz importa, el resultado no

Corra la profundidad de H desde el 4 en vez del 0. Los valores de d y *low* cambian; los puentes y el árbol de puentes, no. Exhiba las dos tablas y explique con el Teorema 22.10 por qué el conjunto de puentes no depende de dónde arranque la profundidad.

Ejercicios

7. Cuántas aristas faltan

Sea T el árbol de puentes de un grafo conexo, con L hojas. Diga cuántas aristas nuevas hacen falta, como mínimo, para que el grafo quede sin puentes, y dé una forma de elegirlas. Pruebe su respuesta con el árbol de H y con un árbol en forma de estrella de cuatro hojas.

8. El puente que más separa

Dado un grafo conexo con n vértices, se quiere el puente que al quitarlo deja los dos pedazos más parejos. Diga cómo se calcula el tamaño de cada lado en $\Theta(V + E)$ usando el árbol de puentes y el número de vértices de cada componente, y cuánto da para el grafo de diez vértices de la sesión.

Para enviar al juez

UVa 610 — Street Directions

`https://vjudge.net/problem/UVA-610`
Envío: `https://vjudge.net/problem/UVA-610`
El parámetro es la calle: hay que decidir cuáles se orientan. La propiedad está en que un puente no puede ser de una vía. La trampa es el formato: línea en blanco tras el número de caso y una línea con # al cerrar.

UVa 10765 — Doves and Bombs

`https://vjudge.net/problem/UVA-10765`
Envío: `https://vjudge.net/problem/UVA-10765`
El parámetro es la estación y la propiedad es el conteo de hijos con $w.low \geq v.d$. La trampa es doble: el `-1 -1` que cierra las vías y los 10000 vértices, que obligan a la pila explícita.

Para enviar al juez

Un tercero, del lado de las aristas

UVa 796 — Critical Links. `https://vjudge.net/problem/UVA-796`
Envío: `https://vjudge.net/problem/UVA-796`
Pide la lista de puentes ordenada, con el extremo menor primero. Sirve para probar la primera de las tres pasadas por separado antes de armar el árbol.

Lo que sigue

Componentes biconexos

El árbol de puentes corta por aristas y agrupa vértices. Cortar por puntos de articulación da otra partición, y esta reparte las *aristas*: cada arista pertenece a un solo componente biconexo y un punto de articulación queda en varios a la vez. La misma profundidad con *low* los encuentra, guardando aristas en una pila en lugar de vértices.

Para investigar

Por qué la partición va sobre las aristas y no sobre los vértices, y qué se apila en cada llamado para sacar un componente al cerrar un punto de articulación. CLRS, Problema 22-2, partes g y h.

Contenido

1 Antes de empezar

2 Recuento

3 El árbol de puentes

4 Ejercicios resueltos

5 Cómo atacar estos problemas

6 Errores comunes

7 Ejercicios propuestos

8 Referencias

Referencias

- Cormen, T. H., Leiserson, C. E., Rivest, R. L., Stein, C. *Introduction to Algorithms*, 3rd ed. MIT Press, 2009. Problema 22-2 (puntos de articulación, puentes y componentes biconexos); Sección 22.3, Teorema 22.10 (en un grafo no dirigido solo hay aristas de árbol y de retroceso).
- UVa 610 — Street Directions.
  `https://vjudge.net/problem/UVA-610`
  Envío: `https://vjudge.net/problem/UVA-610`
- UVa 10765 — Doves and Bombs.
  `https://vjudge.net/problem/UVA-10765`
  Envío: `https://vjudge.net/problem/UVA-10765`
- Robbins, H. E. A theorem on graphs, with an application to a problem of traffic control. *The American Mathematical Monthly*, 46(5):281–283, 1939.

Referencias



Tarjan, R. E. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.



Halim, S., Halim, F., Effendy, S. *Competitive Programming 4*. Lulu, 2020. Capítulo 4: puntos de articulación, puentes y orientación de grafos.