Orden topológico con la profundidad y el valor *low*

Tarjan para componentes fuertemente conexos, puntos de articulación y puentes

Carlos A Delgado S

Pontificia Universidad Javeriana Cali

Octubre de 2026

Contenido

1 Antes de empezar

2 Recuento

3 Orden topológico con la profundidad

4 El valor *low*

5 El algoritmo de Tarjan

6 Puntos de articulación y puentes

7 Problemas de juez

8 Errores comunes

9 Ejercicios

10 Referencias

Contenido

1 Antes de empezar

2 Recuento

3 Orden topológico con la profundidad

4 El valor *low*

5 El algoritmo de Tarjan

6 Puntos de articulación y puentes

7 Problemas de juez

8 Errores comunes

9 Ejercicios

10 Referencias

Objetivos de la sesión

Al terminar la clase, usted podrá

- Obtener un orden topológico con la búsqueda en profundidad y demostrar por qué el orden por f decreciente lo es.
- Detectar un ciclo con los colores de la profundidad.
- Calcular $v.low$ y explicar qué dice de un vértice.

Y también

- Ejecutar a mano e implementar el algoritmo de Tarjan para componentes fuertemente conexos, en $\Theta(V + E)$.
- Encontrar los puntos de articulación y los puentes de un grafo no dirigido con una sola profundidad, y justificar cada criterio.

La pregunta de hoy

Lo que quedó de la conectividad

Kosaraju encuentra los componentes fuertemente conexos con dos profundidades y una copia del grafo al revés. Los puntos de articulación y los puentes quedaron solo como definición: para hallarlos, lo único que se tiene es quitar cada pieza y volver a contar.

La pregunta

¿Cuánto de eso se puede sacar de *una sola* búsqueda en profundidad, mirando los tiempos que deja en cada vértice? La respuesta pasa por un número nuevo, $v.low$, y empieza por algo que ya se insinuó: el orden topológico sale de los tiempos de finalización.

Contenido

1 Antes de empezar

2 Recuento

3 Orden topológico con la profundidad

4 El valor *low*

5 El algoritmo de Tarjan

6 Puntos de articulación y puentes

7 Problemas de juez

8 Errores comunes

9 Ejercicios

10 Referencias

Dos tiempos y un paréntesis

Las marcas (CLRS, Sección 22.3)

$v.d$ es el instante en que la profundidad descubre v y lo pinta de gris; $v.f$, el instante en que termina con él y lo pinta de negro. Un vértice está gris exactamente en $[v.d, v.f]$.

Teorema 22.7 — del paréntesis

Para dos vértices u y v, los intervalos $[u.d, u.f]$ y $[v.d, v.f]$ son disjuntos, o uno está contenido en el otro. Está contenido exactamente cuando ese vértice es descendiente del otro en el bosque de la profundidad.

Teorema 22.9 — del camino blanco

v es descendiente de u si y solo si, en el instante $u.d$, hay un camino de u a v hecho solo de vértices blancos.

Los cuatro tipos de arista

Según el color de v cuando se explora (u, v)

- **Blanco**: arista de árbol; v se descubre desde u.
- **Gris**: arista de retroceso; v es ancestro de u y sigue abierto.
- **Negro** con $u.d < v.d$: arista hacia adelante, a un descendiente ya terminado.
- **Negro** con $u.d > v.d$: arista cruzada, a otra rama o a otro árbol.

En un grafo no dirigido (Teorema 22.10)

Solo hay aristas de árbol y de retroceso. Las otras dos clases no aparecen: cada arista se ve desde los dos extremos, y desde el primero que se explora ya sale como de árbol o de retroceso.

El algoritmo de Kosaraju

Kosaraju(G)

1. Profundidad sobre G; los vértices por f decreciente en *ord*.
2. Construir G^T.
3. Recorrer *ord* y, desde cada vértice sin asignar, una profundidad sobre G^T: lo que alcanza es un componente.

Lo que cuesta

Dos profundidades y una copia del grafo: $\Theta(V + E)$ en tiempo y $\Theta(V + E)$ de memoria extra por G^T. Sobre el grafo de ocho vértices salió $\{a, b, e\}$, $\{c, d\}$, $\{f, g\}$, $\{h\}$, en ese orden: un orden topológico de G^{SCC}.

Contenido

1 Antes de empezar

2 Recuento

3 Orden topológico con la profundidad

4 El valor *low*

5 El algoritmo de Tarjan

6 Puntos de articulación y puentes

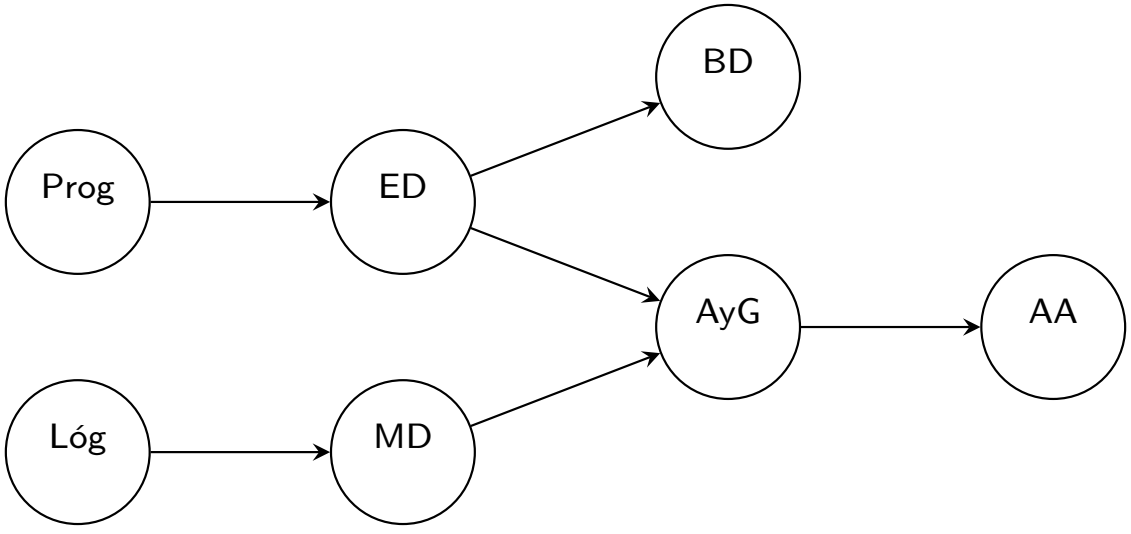
7 Problemas de juez

8 Errores comunes

9 Ejercicios

10 Referencias

Por tanteo



Antes de ver el algoritmo

Kahn empieza por las fuentes: lo que puede ir *primero*. La profundidad no cuenta grados de entrada, pero sí sabe cuándo termina con un vértice. Si se arranca en AyG, ¿cuál termina primero? ¿Puede ese ir al principio del plan, o le toca ir al final?

Por tanteo

Lo que se observa

El primero en terminar es AA: no tiene salidas, así que nada tiene que ir después de él, y puede ir *de último*. Luego termina AyG, cuando ya no le falta nada por debajo. La profundidad arma el plan desde el final hacia el principio: el que termina, se pone adelante de los que ya estaban.

El algoritmo

TopologicalSort(G) — CLRS, Sección 22.4

1. Ejecutar *DFS*(G) para calcular $v.f$ de cada vértice
2. Cada vez que un vértice termina, insertarlo al principio de una lista
3. Devolver la lista

En otras palabras

Los vértices por tiempo de finalización decreciente. Es el mismo *ord* que Kosaraju calcula en su primer paso, ahora sobre un grafo sin ciclos.

Lema (CLRS, Lema 22.11)

Un grafo dirigido es acíclico si y solo si la búsqueda en profundidad no produce aristas de retroceso.

El algoritmo

Para qué sirve el lema

Durante la misma profundidad se puede decidir si hay ciclo: basta ver si alguna arista llega a un vértice gris. Si llega, no hay orden topológico y la lista no sirve.

Por qué el orden que sale es topológico

Teorema (CLRS, Teorema 22.12)

Si G es un grafo dirigido acíclico, para toda arista (u, v) de G se cumple $v.f < u.f$. En consecuencia, TopologicalSort(G) produce un orden topológico de G.

Demostración

Se procede de forma directa, por casos sobre el color de v en el instante en que la profundidad explora la arista (u, v). En ese instante u es gris.

Por qué el orden que sale es topológico

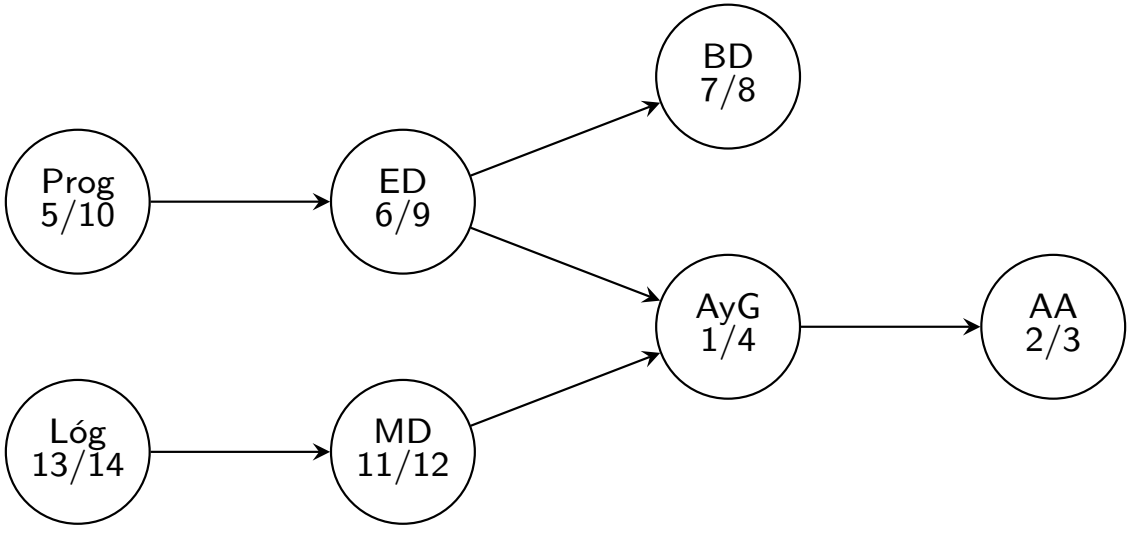
Desarrollo

- v **gris.** Entonces v es ancestro de u y (u, v) es de retroceso. Por el Lema 22.11, G tendría un ciclo, en contra de la hipótesis. Este caso no ocurre.
- v **blanco.** Por el teorema del camino blanco, v queda como descendiente de u, y por el del paréntesis $[v.d, v.f] \subset [u.d, u.f]$: luego $v.f < u.f$.
- v **negro.** v ya terminó y $v.f$ ya tiene valor; u sigue gris, así que $u.f$ se asigna después. Como el reloj solo avanza, $v.f < u.f$.

Conclusión

Por lo tanto, se puede concluir que para toda arista (u, v) de un grafo acíclico $v.f < u.f$, y al listar por f decreciente u queda antes que v: la lista es un orden topológico.

Sobre el plan de estudios



Los tiempos, arrancando en el orden de los números

AyG abre el reloj y baja a AA, que termina en 3. Después arranca Prog, que baja a ED y de ahí a BD; AyG ya está negro. Luego MD y por último Lóg. Cuatro arranques: AyG, Prog, MD y Lóg.

Sobre el plan de estudios

Por f decreciente

Lóg	MD	Prog	ED	BD	AyG	AA
14	12	10	9	8	4	3

Cada una de las seis aristas va de izquierda a derecha.

Y Kahn

Con la cola, Kahn dio Prog, Lóg, ED, MD, BD, AyG, AA. Es otro orden y también es válido: un grafo acíclico puede tener muchos, y cada algoritmo encuentra uno.

La versión recursiva

```
BLANCO, GRIS, NEGRO = 0, 1, 2

def topo_aux(G, u, color, orden, ciclo):
    # u se agrega a orden cuando termina con el: por f creciente.
    color[u] = GRIS
    for v in G[u]:
        if color[v] == BLANCO:
            topo_aux(G, v, color, orden, ciclo)
        elif color[v] == GRIS:
            ciclo[0] = True
    color[u] = NEGRO
    orden.append(u)
```

La versión recursiva

```
def orden_topologico_dfs(G):
    # Devuelve (orden, hay_ciclo). Si hay ciclo, orden no sirve.
    n = len(G)
    color = [BLANCO] * n
    orden = []
    ciclo = [False]
    u = 0
    while u < n:
        if color[u] == BLANCO:
            topo_aux(G, u, color, orden, ciclo)
        u = u + 1
    orden.reverse()
    resultado = (orden, ciclo[0])
    return resultado
```

La versión recursiva

Dos detalles

`orden.append(u)` va *después* del ciclo: es el instante $u.f$. Y `ciclo` es una lista de un elemento para que la recursión pueda cambiarlo; si queda en `True`, la lista no es un orden topológico.

La versión con pila explícita

```
def topo_aux_con_pila(G, u, color, orden, ciclo):
    # Cada vertice entra dos veces: la primera lo pinta de gris y la
    # segunda lo pinta de negro y lo agrega a orden.
    pila = [(u, False)]
    while len(pila) > 0:
        w, finalizando = pila.pop()
        if finalizando:
            color[w] = NEGRO
            orden.append(w)
        elif color[w] == BLANCO:
            color[w] = GRIS
            pila.append((w, True))
            for v in G[w]:
                if color[v] == BLANCO:
                    pila.append((v, False))
                elif color[v] == GRIS:
                    ciclo[0] = True
```

La versión con pila explícita

```
def orden_topologico_dfs_con_pila(G):
    n = len(G)
    color = [BLANCO] * n
    orden = []
    ciclo = [False]
    u = 0
    while u < n:
        if color[u] == BLANCO:
            topo_aux_con_pila(G, u, color, orden, ciclo)
        u = u + 1
    orden.reverse()
    resultado = (orden, ciclo[0])
    return resultado
```

La versión con pila explícita

Lo que cambia

Solo dónde queda lo pendiente. Cada vértice entra dos veces a la pila: con `False` para descubrirlo y con `True` para terminarlo. El par con `True` queda debajo de los vecinos, así que sale cuando ya se terminó con todos ellos, igual que en la recursiva.

Cuando hay un ciclo

El plan con un prerrequisito de más

Se agrega AA $\rightarrow$ ED. La profundidad arranca en AyG, baja a AA y de ahí a ED: los tres están grises. ED mira a AyG y lo encuentra gris.

La arista de retroceso

ED $\rightarrow$ AyG llega a un ancestro que sigue abierto. Junto con el camino de árbol AyG $\rightarrow$ AA $\rightarrow$ ED cierra el ciclo, y `orden_topologico_dfs` devuelve `ciclo = True`.

Gris, no negro

Llegar a un vértice negro no es un ciclo: es una arista hacia adelante o cruzada, como Prog $\rightarrow$ ED $\rightarrow$ AyG cuando AyG ya había terminado. Solo el gris marca el ciclo, porque solo el gris está todavía en el camino de la recursión.

Kahn o profundidad

	Kahn	Profundidad
Qué mira	grados de entrada	tiempos de finalización
Construye el orden	del principio al final	del final al principio
Ciclo	sobran vértices sin emitir	arista a un gris
Estructura	cola	pila (o recursión)
Costo	$\Theta(V + E)$	$\Theta(V + E)$

Lo que cuesta la profundidad

Cada vértice se pinta de gris una vez y de negro una vez, y cada lista de adyacencia se recorre una vez: $\Theta(V + E)$. Invertir la lista al final cuesta $\Theta(V)$.

Contenido

1 Antes de empezar

2 Recuento

3 Orden topológico con la profundidad

4 El valor *low*

5 El algoritmo de Tarjan

6 Puntos de articulación y puentes

7 Problemas de juez

8 Errores comunes

9 Ejercicios

10 Referencias

Dónde empieza un componente

Una pregunta sobre el grafo de ocho vértices

En la profundidad de Kosaraju, $\{c, d\}$ se descubrió completo dentro del intervalo de c, y $\{f, g\}$ dentro del de g. ¿Es casualidad?

No lo es

Sea r el primer vértice de un componente C que la profundidad descubre. En el instante $r.d$, todos los demás de C son blancos y se alcanzan desde r por caminos que no salen de C, todos blancos. Por el teorema del camino blanco, todo C queda como descendiente de r.

Dónde empieza un componente

La raíz del componente

Cada componente fuertemente conexo cuelga de un solo vértice, su *raíz*, dentro de un mismo árbol de la profundidad. Si se sabe reconocer la raíz en el momento en que termina, el componente es lo que se descubrió desde ella y todavía no se ha repartido.

Cómo se reconoce

r es raíz cuando desde su subárbol no hay forma de subir a un vértice descubierto antes que r que siga sin componente. Si la hubiera, r y ese vértice estarían en el mismo componente, y r no sería el primero.

La definición

Definición (Tarjan, 1972)

En la búsqueda en profundidad, $v.low$ es el menor $w.d$ entre v y los vértices w a los que se llega desde el subárbol de v con una arista que no es de árbol, siempre que w siga en la pila de vértices sin componente.

En la misma recursión

- Al descubrir v: $v.low = v.d$.
- Al volver de un hijo w: $v.low = \text{mín}(v.low, w.low)$.
- Ante una arista (v, w) con w en la pila: $v.low = \text{mín}(v.low, w.d)$.

La definición

La condición de raíz

Siempre $v.low \leq v.d$. Hay igualdad cuando nada del subárbol de v sube por encima de v: v es la raíz de su componente.

Un reloj más simple

Para *low* solo importan las comparaciones entre tiempos de descubrimiento, así que el reloj avanza solo al descubrir: $v.d$ va de 1 a $|V|$. Es el orden en que la profundidad encuentra los vértices.

Por qué la pila

Qué guarda

Cada vértice entra a la pila al ser descubierto y sale cuando se cierra su componente. Los descendientes de una raíz r que siguen en la pila quedan encima de r, juntos, porque se descubrieron después de él.

Qué se saca

Cuando $r.low = r.d$, se desapila hasta r incluido. Lo que sale es el componente: los descendientes de r que no se habían ido con una raíz más profunda.

Por qué la pila

Una arista hacia un componente cerrado

Una arista (v, w) hacia un w que ya salió de la pila lleva a un componente cerrado. De allí no se vuelve a v: si se pudiera, v y w estarían en el mismo componente y w no se habría cerrado sin v. Esa arista no puede bajar $v.low$, y por eso se pregunta por `en_pila[w]` y no por si w ya fue descubierto.

Contenido

1 Antes de empezar

2 Recuento

3 Orden topológico con la profundidad

4 El valor *low*

5 El algoritmo de Tarjan

6 Puntos de articulación y puentes

7 Problemas de juez

8 Errores comunes

9 Ejercicios

10 Referencias

El algoritmo

Tarjan(G)

1. Para cada vértice u sin descubrir, ejecutar *Visit*(u)

Visit(u)

1. Asignar $u.d$ y $u.low$; apilar u
2. Para cada v adyacente a u:
3. si v no ha sido descubierto: *Visit*(v) y $u.low = \text{mín}(u.low, v.low)$
4. si no, y v está en la pila: $u.low = \text{mín}(u.low, v.d)$
5. Si $u.low = u.d$: desapilar hasta u; eso es un componente

El algoritmo

Atribución

Robert Tarjan, 1972, en el artículo que mostró cuántos problemas de grafos se resuelven en tiempo lineal con la búsqueda en profundidad. Una sola profundidad, sin G^T.

Tarjan en Python

```
def en_blanco(grafo):
    # d[v] = 0 dice que v no ha sido descubierto.
    d = {}
    low = {}
    en_pila = {}
    for v in grafo:
        d[v] = 0
        low[v] = 0
        en_pila[v] = False
    resultado = (d, low, en_pila)
    return resultado
```

Tarjan en Python

```
def descubrir(u, reloj, d, low, pila, en_pila):
    # u recibe su numero de descubrimiento y entra a la pila.
    reloj[0] = reloj[0] + 1
    d[u] = reloj[0]
    low[u] = reloj[0]
    pila.append(u)
    en_pila[u] = True

def sacar_componente(u, pila, en_pila, componentes):
    # Desapila hasta u, incluido: eso es un componente.
    comp = []
    w = None
    while w != u:
        w = pila.pop()
        en_pila[w] = False
        comp.append(w)
    componentes.append(comp)
```

Tarjan en Python

```
def tarjan(grafo):
    # grafo: dict que asocia cada vertice a la lista de sucesores.
    # Devuelve los componentes en el orden en que se cierran.
    d, low, en_pila = en_blanco(grafo)
    reloj = [0]
    pila = []
    componentes = []

    def visit(u):
        descubrir(u, reloj, d, low, pila, en_pila)
        for v in grafo[u]:
            if d[v] == 0:
                visit(v)
                low[u] = min(low[u], low[v])
            elif en_pila[v]:
                low[u] = min(low[u], d[v])
        if low[u] == d[u]:
            sacar_componente(u, pila, en_pila, componentes)

    for u in grafo:
        if d[u] == 0:
            visit(u)
    return componentes
```

Tarjan con pila explícita

```
def tarjan_desde(grafo, s, reloj, d, low, pila, en_pila, componentes):
    # llamadas guarda pares [u, i]: i es el sucesor de u que toca
    # revisar cuando la busqueda vuelva a u.
    descubrir(s, reloj, d, low, pila, en_pila)
    llamadas = [[s, 0]]
    while len(llamadas) > 0:
        u, i = llamadas[-1]
        if i < len(grafo[u]):
            llamadas[-1][1] = i + 1
            v = grafo[u][i]
            if d[v] == 0:
                descubrir(v, reloj, d, low, pila, en_pila)
                llamadas.append([v, 0])
            elif en_pila[v]:
                low[u] = min(low[u], d[v])
        else:
            llamadas.pop()
            if low[u] == d[u]:
                sacar_componente(u, pila, en_pila, componentes)
            if len(llamadas) > 0:
                p = llamadas[-1][0]
                low[p] = min(low[p], low[u])
```

Tarjan con pila explícita

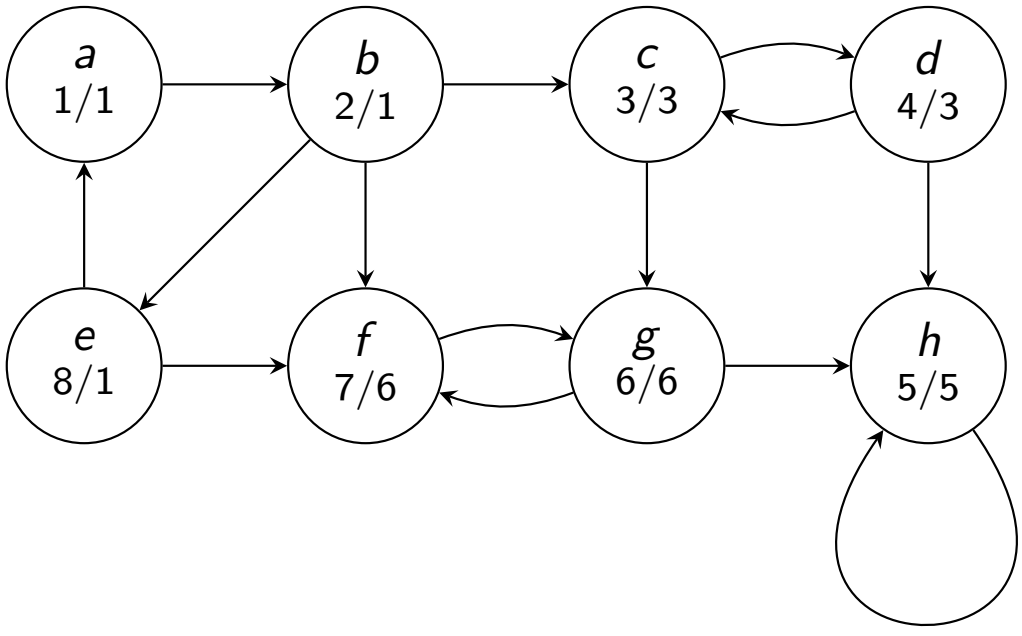
```
def tarjan_con_pila(grafo):
    # Lo mismo que tarjan, con la profundidad en una lista.
    d, low, en_pila = en_blanco(grafo)
    reloj = [0]
    pila = []
    componentes = []
    for s in grafo:
        if d[s] == 0:
            tarjan_desde(grafo, s, reloj, d, low, pila, en_pila,
                         componentes)
    return componentes
```

Tarjan con pila explícita

Dos pilas distintas

`pila` es la de Tarjan: los vértices sin componente. `llamadas` reemplaza la pila de la recursión: cada par $[u, i]$ dice por cuál sucesor de u va el ciclo. Lo que la recursiva hace al volver de $\mathit{Visit}(v)$ —actualizar el *low* del padre— aquí pasa al sacar el par de u de `llamadas`, con el vértice `p` que queda arriba.

Paso a paso: la primera rama



Cada vértice con d/low al terminar

Las listas de adyacencia son las de la clase de conectividad, y la profundidad arranca en a.

Paso a paso: la primera rama

Paso	Qué pasa	Pila
a, b, c, d	se descubren con $d = 1, 2, 3, 4$	$a\, b\, c\, d$
$d \to c$	c en la pila: $d.low = 3$	
$d \to h$	se descubre h, $d = 5$	$a\, b\, c\, d\, h$
$h \to h$	el lazo no baja nada	
h termina	$h.low = h.d = 5$: sale $\{h\}$	$a\, b\, c\, d$
d termina	$d.low = 3 \neq 4$: se queda	

Pregunta

$d.low = 3$ dice que desde d se vuelve a c. ¿Qué habría pasado si $d \to c$ no existiera?

Paso a paso: el resto

Paso	Qué pasa	Pila
$c \to g$, $g \to f$	g: $d = 6$; f: $d = 7$	$a\, b\, c\, d\, g\, f$
$f \to g$	g en la pila: $f.low = 6$	
$g \to h$	h ya salió: no cuenta	
g termina	$g.low = g.d = 6$: sale $\{f, g\}$	$a\, b\, c\, d$
c termina	$c.low = c.d = 3$: sale $\{c, d\}$	$a\, b$
$b \to e$, $e \to a$	e: $d = 8$; a en la pila: $e.low = 1$	$a\, b\, e$
$e \to f$, $b \to f$	f ya salió: no cuentan	
a termina	$a.low = a.d = 1$: sale $\{a, b, e\}$	vacía

Paso a paso: el resto

El resultado

Los componentes salen en el orden $\{h\}$, $\{f, g\}$, $\{c, d\}$, $\{a, b, e\}$: los mismos cuatro de Kosaraju. Leídos al revés son un orden topológico de G^{SCC}; Tarjan los entrega de los sumideros hacia las fuentes, porque un componente solo se cierra cuando ya se cerró todo lo que se alcanza desde él.

La arista $g \to h$

Si se contara, $g.low$ bajaría a 5, g no sería raíz y $\{f, g\}$ se iría con $\{c, d\}$ en un solo componente equivocado. Es el caso que separa "en la pila" de "ya descubierto".

Tarjan o Kosaraju

	Kosaraju	Tarjan
Profundidades	dos	una
Grafo transpuesto	sí	no
Datos por vértice	f y componente	d, *low*, en pila
Orden de salida	fuentes primero	sumideros primero
Tiempo	$\Theta(V+E)$	$\Theta(V+E)$

Lo que cuesta Tarjan

Cada vértice se descubre una vez, entra a la pila una vez y sale una vez; cada arista se mira una vez desde su origen. $\Theta(V+E)$ en tiempo y $\Theta(V)$ de memoria extra.

Contenido

1 Antes de empezar

2 Recuento

3 Orden topológico con la profundidad

4 El valor *low*

5 El algoritmo de Tarjan

6 Puntos de articulación y puentes

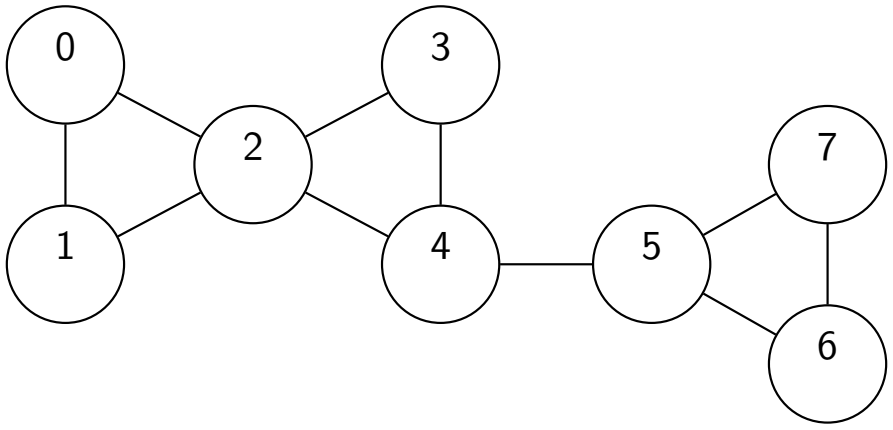
7 Problemas de juez

8 Errores comunes

9 Ejercicios

10 Referencias

El grafo de la sección



Ocho vértices y diez aristas

Dos triángulos que comparten el 2, la arista 4–5 y un tercer triángulo. Listas: 0:[1, 2], 1:[0, 2], 2:[0, 1, 3, 4], 3:[2, 4], 4:[2, 3, 5], 5:[4, 6, 7], 6:[5, 7], 7:[5, 6].

Pregunta

¿Cuáles son los puntos de articulación? ¿Y los puentes? Hay un punto de articulación sin ningún puente al lado.

La versión que uno escribe primero

```
def articulaciones_ingenuo(G):
    # Quita cada vertice y cuenta los componentes que quedan.
    base = contar_componentes(G, -1, None)
    articulaciones = []
    u = 0
    while u < len(G):
        if len(G[u]) > 0 and contar_componentes(G, u, None) > base:
            articulaciones.append(u)
        u = u + 1
    return articulaciones
```

Es la definición ejecutada

`contar_componentes(G, u, None)` recorre el grafo como si u no existiera. Sobre el grafo de la sección devuelve $[2, 4, 5]$.

La versión que uno escribe primero

Lo que cuesta

Un recorrido completo por cada vértice: $\Theta(V \cdot (V + E))$. Para los puentes, lo mismo con cada arista: $\Theta(E \cdot (V + E))$. Con 10^5 vértices son del orden de 10^{10} pasos. La meta es sacar las dos respuestas de una sola profundidad.

low en un grafo no dirigido

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 cambia frente a Tarjan para componentes

No hay aristas cruzadas ni hacia adelante (Teorema 22.10), así que no hace falta pila: todo vértice ya descubierto que se ve desde u es ancestro o descendiente. Lo único que hay que excluir es la arista por la que se llegó, la que va al padre: vista al revés parece de retroceso y no lo es.

low en un grafo no dirigido

Qué dice

$v.low$ es lo más arriba en el árbol que se puede llegar desde el subárbol de v con un solo salto de retroceso. Si un hijo w de v tiene $w.low \geq v.d$, todo el subárbol de w depende de v para llegar al resto del grafo.

La raíz

Teorema (CLRS, Problema 22-2a)

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

Demostración

Se procede de forma directa, en las dos direcciones, con el Teorema 22.10.

Desarrollo

Si la raíz r tiene dos hijos w_1 y w_2, no hay ninguna arista entre el subárbol de w_1 y el de w_2: sería cruzada, y en un grafo no dirigido no las hay. Sin r, quedan separados. Si r tiene un solo hijo, todos los demás vértices están en su subárbol, que sigue conexo por las aristas de árbol sin pasar por r.

La raíz

Conclusión

Por lo tanto, se puede concluir que la raíz es punto de articulación exactamente cuando tiene dos hijos o más.

La raíz

En el grafo de la sección

Arrancando en 0, la raíz baja a 1, y desde 1 se llega a 2 y a todo lo demás: un solo hijo, y 0 no es articulación. Arrancando en 2, la raíz tiene dos hijos, 0 y 3, y sí lo es: sin el 2, el 0 y el 1 quedan aislados del resto.

Por qué la raíz va aparte

Para la raíz siempre vale $w.low \geq r.d$, porque $r.d$ es el mínimo del árbol. La condición de los demás vértices la marcaría siempre.

Los demás vértices

Teorema (CLRS, Problema 22-2b)

Un vértice v que no es raíz es punto de articulación si y solo si tiene un hijo w con $w.low \geq v.d$.

Demostración

Se procede de forma directa, leyendo qué dice $w.low$.

Desarrollo

$w.low \geq v.d$ dice que ninguna arista de retroceso del subárbol de w llega a un ancestro propio de v. Como no hay aristas cruzadas, el subárbol de w solo toca al resto del grafo a través de v: sin v, queda separado del padre de v, que existe porque v no es raíz. Si todos los hijos tienen $w.low < v.d$, cada subárbol tiene una arista que salta por encima de v y lo mantiene unido al resto.

Los demás vértices

Conclusión

Por lo tanto, se puede concluir que v es punto de articulación exactamente cuando algún hijo cumple $w.low \geq v.d$.

Los puentes

Teorema (CLRS, Problema 22-2)

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

Por qué la desigualdad es estricta

Con $w.low = v.d$ hay una arista de retroceso desde el subárbol de w hasta v mismo: con ella y el camino de árbol se cierra un ciclo que contiene a (v, w), y una arista en un ciclo no es puente. Para el punto de articulación esa arista no sirve, porque llega a v, que es el que se quita; para el puente sí, porque v se queda.

Y las de retroceso

Una arista de retroceso cierra un ciclo con el camino de árbol entre sus extremos. No es puente.

Los dos criterios en Python

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

def revisar_hijo(u, v, es_raiz, d, low, es_art, puentes):
    # v es hijo de u y ya termino: su low es definitivo.
    low[u] = min(low[u], low[v])
    if not es_raiz and low[v] >= d[u]:
        es_art[u] = True
    if low[v] > d[u]:
        puentes.append((u, v))
```

Los dos criterios en Python

```
def cortes_aux(G, u, padre, reloj, d, low, es_art, puentes):
    # padre = -1 dice que u es la raiz de su arbol.
    descubrir(u, reloj, d, low)
    hijos = 0
    for v in G[u]:
        if d[v] == 0:
            hijos = hijos + 1
            cortes_aux(G, v, u, reloj, d, low, es_art, puentes)
            revisar_hijo(u, v, padre == -1, d, low, es_art, puentes)
        elif v != padre:
            low[u] = min(low[u], d[v])
    if padre == -1 and hijos >= 2:
        es_art[u] = True
```

Los dos criterios en Python

```
def cortes(G):
    # Devuelve (articulaciones, puentes). d[v] = 0: sin descubrir.
    n = len(G)
    d = [0] * n
    low = [0] * n
    es_art = [False] * n
    puentes = []
    reloj = [0]
    for u in range(n):
        if d[u] == 0:
            cortes_aux(G, u, -1, reloj, d, low, es_art, puentes)
    resultado = (marcados(es_art), puentes)
    return resultado
```

Los dos criterios en Python

```
def marcados(es_art):
    # Los vertices con es_art en True, en orden.
    articulaciones = []
    u = 0
    while u < len(es_art):
        if es_art[u]:
            articulaciones.append(u)
        u = u + 1
    return articulaciones
```

Los dos criterios en Python

Dónde está cada criterio

`v != padre` excluye la arista por la que se llegó. Al volver de `cortes_aux`, `revisar_hijo` hace las dos comparaciones: $\geq$ para el punto de articulación, que no se aplica a la raíz, y $>$ para el puente. La raíz se decide al final, contando sus hijos.

Con pila explícita

```
def cortes_desde(G, s, reloj, d, low, es_art, puentes):
    # llamadas guarda ternas [u, padre, i]. Devuelve los hijos de s.
    descubrir(s, reloj, d, low)
    hijos = 0
    llamadas = [[s, -1, 0]]
    while len(llamadas) > 0:
        u, padre, i = llamadas[-1]
        if i < len(G[u]):
            llamadas[-1][2] = i + 1
            v = G[u][i]
            if d[v] == 0:
                if u == s:
                    hijos = hijos + 1
                descubrir(v, reloj, d, low)
                llamadas.append([v, u, 0])
            elif v != padre:
                low[u] = min(low[u], d[v])
        else:
            llamadas.pop()
            if padre != -1:
                revisar_hijo(padre, u, padre == s, d, low, es_art, puentes)
    return hijos
```

Con pila explícita

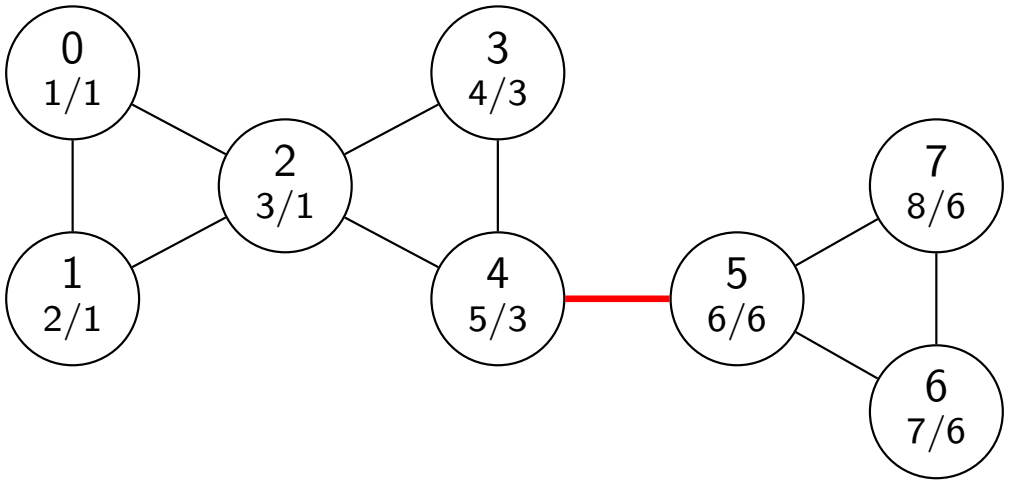
```
def cortes_con_pila(G):
    # Lo mismo que cortes, con la profundidad en una lista.
    n = len(G)
    d = [0] * n
    low = [0] * n
    es_art = [False] * n
    puentes = []
    reloj = [0]
    for s in range(n):
        if d[s] == 0 and cortes_desde(G, s, reloj, d, low, es_art,
        ↪  puentes) >= 2:
            es_art[s] = True
    resultado = (marcados(es_art), puentes)
    return resultado
```

Con pila explícita

Lo que cambia

`revisar_hijo`, que la recursiva llama al volver de `cortes_aux`, aquí se llama al sacar la terna de u de `llamadas`, con su padre: ahí u ya terminó y su *low* es definitivo.

Paso a paso, arrancando en 0



d/low al terminar

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

Paso a paso, arrancando en 0

Arista de árbol (v, w)	$w.low$ vs $v.d$	Articulación	Puente
$(6, 7)$	$6 < 7$	no	no
$(5, 6)$	$6 = 6$	5 sí	no
$(4, 5)$	$6 > 5$	4 sí	sí
$(3, 4)$	$3 < 4$	no	no
$(2, 3)$	$3 = 3$	2 sí	no
$(1, 2)$	$1 < 2$	no	no
$(0, 1)$	raíz con un hijo	0 no	no

El resultado

Puntos de articulación 2, 4 y 5; un solo puente, $(4, 5)$. El 2 es articulación sin ningún puente al lado: separa dos triángulos que solo lo comparten a él.

Lo que cuesta

Una profundidad

Cada vértice se descubre una vez y cada arista se mira dos veces, una desde cada extremo. Las comparaciones son $O(1)$ por arista de árbol: $\Theta(V + E)$ para las dos respuestas a la vez.

Frente a la versión ingenua

De $\Theta(V \cdot (V + E))$ a $\Theta(V + E)$. Con 10^5 vértices y $2 \cdot 10^5$ aristas, de unos $3 \cdot 10^{10}$ pasos a unos $3 \cdot 10^5$.

Contenido

1 Antes de empezar

2 Recuento

3 Orden topológico con la profundidad

4 El valor *low*

5 El algoritmo de Tarjan

6 Puntos de articulación y puentes

7 Problemas de juez

8 Errores comunes

9 Ejercicios

10 Referencias

Cuatro para practicar

UVa 796 — Critical Links

`https://vjudge.net/problem/UVA-796`
Envío: `https://vjudge.net/problem/UVA-796`
Los puentes de una red. La lista de salida va ordenada y cada puente con el extremo menor primero. El grafo puede venir desconectado y con vértices sin aristas: el ciclo externo arranca en todos.

UVa 315 — Network

`https://vjudge.net/problem/UVA-315`
Envío: `https://vjudge.net/problem/UVA-315`
Cuántos puntos de articulación hay. La trampa está en la lectura: cada línea trae un vértice seguido de sus vecinos, en cantidad variable, y una línea con un 0 solo cierra el caso.

Cuatro para practicar

UVa 247 — Calling Circles

`https://vjudge.net/problem/UVA-247`
Envío: `https://vjudge.net/problem/UVA-247`
Los círculos de llamadas son los componentes fuertemente conexos. Los nombres se traducen a números con un diccionario antes de correr Tarjan.

UVa 11838 — Come and Go

`https://vjudge.net/problem/UVA-11838`
Envío: `https://vjudge.net/problem/UVA-11838`
¿La ciudad es fuertemente conexa? Hay calles de una vía y de doble vía ($P = 2$ agrega las dos direcciones). Basta ver si Tarjan saca un solo componente.

Contenido

1 Antes de empezar

2 Recuento

3 Orden topológico con la profundidad

4 El valor *low*

5 El algoritmo de Tarjan

6 Puntos de articulación y puentes

7 Problemas de juez

8 Errores comunes

9 Ejercicios

10 Referencias

Errores comunes

1. Ordenar por d en lugar de por f

Sobre el plan de estudios, por d decreciente sale Lóg, MD, BD, ED, Prog, AA, AyG: AA queda antes que AyG, y la arista AyG $\to$ AA va al revés. El que garantiza el orden es f.

2. Llamar ciclo a cualquier vértice ya visitado

En la profundidad, llegar a un negro no es ciclo. Solo el gris lo es. Con `visitado` de dos estados, el plan de estudios parecería tener ciclo en Prog $\to$ ED $\to$ AyG.

Errores comunes

3. En Tarjan, preguntar si w fue descubierto

Con `elif d[v] != 0` en vez de `elif en_pila[v]`, la arista $g \to h$ baja $g.low$ a 5 y $\{c, d, f, g\}$ sale como un solo componente.

4. No excluir la arista al padre

Sin `v != padre`, todo hijo tiene $w.low \leq v.d$ y $w.low > v.d$ no se cumple nunca: no aparece ningún puente. Los puntos de articulación siguen saliendo bien, porque su condición admite la igualdad; por eso el error pasa inadvertido si solo se prueban ellos.

Errores comunes

5. Usar $\geq$ para el puente

En el triángulo 0–1–2, la arista $(0, 1)$ tiene $1.\mathit{low} = 1 = 0.d$ y saldría como puente, aunque está en un ciclo.

6. Aplicarle a la raíz la condición de los demás

Para la raíz $w.\mathit{low} \geq r.d$ vale siempre. Arrancando en 0, el 0 saldría como articulación, y no lo es.

Contenido

1 Antes de empezar

2 Recuento

3 Orden topológico con la profundidad

4 El valor *low*

5 El algoritmo de Tarjan

6 Puntos de articulación y puentes

7 Problemas de juez

8 Errores comunes

9 Ejercicios

10 Referencias

Ejercicios

1. Orden topológico a mano

Sea D el grafo dirigido con $V = \{a, b, c, d, e, f, g\}$ y aristas $a \to b$, $a \to d$, $b \to c$, $d \to c$, $d \to e$, $f \to e$, $g \to f$, con las listas en orden alfabético. Calcule $v.d$ y $v.f$ arrancando por orden alfabético, escriba el orden por f decreciente y compruebe arista por arista que es topológico. Después corra Kahn con cola y compare.

2. El ciclo en la misma profundidad

A D se le agrega $e \to a$. Diga qué arista encuentra un vértice gris, cuál es el ciclo y qué devuelve `orden_topologico_dfs`.

Ejercicios

3. Por qué no sirve d

Dé un grafo acíclico de tres vértices donde el orden por d creciente no sea topológico, y uno donde el orden por f creciente tampoco lo sea. Explique con el Teorema 22.12 por qué el de f decreciente siempre lo es.

4. Tarjan a mano

Sea K el grafo dirigido con $V = \{1, \ldots, 7\}$ y aristas $1 \to 2$, $2 \to 3$, $3 \to 1$, $2 \to 4$, $4 \to 5$, $5 \to 6$, $6 \to 4$, $6 \to 7$. Ejecute Tarjan arrancando en 1: escriba d y *low* de cada vértice, el contenido de la pila cada vez que se cierra un componente y los componentes en el orden en que salen. Verifique que, leídos al revés, son un orden topológico de K^{SCC}.

Ejercicios

5. La pregunta por la pila

Encuentre un grafo dirigido de tres vértices en el que Tarjan con `elif d[v] != 0` en lugar de `elif en_pila[v]` dé componentes equivocados. Señale la arista que baja un *low* que no debía.

6. Los cortes de H, con *low*

Sea H el grafo no dirigido con $V = \{0, \ldots, 6\}$ y aristas 0–1, 1–2, 2–0, 2–3, 3–4, 4–5, 5–6, 6–4. Corra la profundidad desde 0, escriba d y *low* de cada vértice y llene la tabla de aristas de árbol con las dos comparaciones. Compare con lo que encontró quitando pieza por pieza.

Ejercicios

7. La raíz

Demuestre que la raíz de un árbol de la profundidad de un grafo no dirigido es punto de articulación si y solo si tiene al menos dos hijos. Escriba la demostración en cuatro partes: teorema, estrategia, desarrollo y conclusión, y diga en qué paso usa que no hay aristas cruzadas.

8. La igualdad

Dé un grafo no dirigido con una arista de árbol (v, w) en la que $w.low = v.d$. Muestre que v es punto de articulación y que (v, w) no es puente, y escriba el ciclo que contiene a (v, w).

Lo que sigue

Componentes biconexos

Los puntos de articulación parten un grafo no dirigido en pedazos que ya no tienen ninguno: los componentes biconexos. Cada arista pertenece a exactamente uno, y un punto de articulación está en varios a la vez. En el grafo de la sección son los tres triángulos y la arista 4–5.

Para investigar

CLRS, Problema 22-2, partes g y h: por qué los componentes biconexos reparten las aristas y no los vértices, y cómo la misma profundidad con *low* los saca guardando aristas en una pila en lugar de vértices.

Contenido

1 Antes de empezar

2 Recuento

3 Orden topológico con la profundidad

4 El valor *low*

5 El algoritmo de Tarjan

6 Puntos de articulación y puentes

7 Problemas de juez

8 Errores comunes

9 Ejercicios

10 Referencias

Referencias

- Cormen, T. H., Leiserson, C. E., Rivest, R. L., Stein, C. *Introduction to Algorithms*, 3rd ed. MIT Press, 2009. Sección 22.3, Teoremas 22.7, 22.9 y 22.10 (tiempos, paréntesis, camino blanco y aristas en grafos no dirigidos); Sección 22.4, Lema 22.11 y Teorema 22.12 (orden topológico); Sección 22.5 (componentes fuertemente conexos); Problema 22-2 (puntos de articulación, puentes y componentes biconexos).
- 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: orden topológico, puntos de articulación y puentes, y componentes fuertemente conexos.
- Erickson, J. *Algorithms*. 2019. Capítulo 6, búsqueda en profundidad: orden topológico y componentes fuertemente conexos.

Referencias