Encontrando puentes en grafos

Eric Brandwein

7 de mayo de 2024

Antes que nada: recomiendo ir haciéndose dibujitos de todo lo que vayan leyendo para convencerse. Yo no los hice por vago nomás. Ahora sí, empecemos.

Definición (Arista puente)

Una arista e de un grafo G es una arista puente si $G \setminus e$ tiene más componentes conexas que G.

Vamos a tratar de resolver el siguiente problema.

Problema (Guía 4 Ejercicio 2)

Dado un grafo G, enumerar las aristas puente de G en tiempo $O(n + m)$.

Empecemos con la siguiente propiedad de las aristas puente.

Lema 1

Una arista (u, v) de un grafo G es puente si y sólo si no hay camino entre u y v en $G \setminus (u, v)$.

Demostración

$\Rightarrow$) Supongamos que (u, v) es un puente. Llamemos G' a $G \setminus (u, v)$. Queremos ver que no hay camino entre u y v en G'. Para encontrar una contradicción, supongamos que sí existe camino $\langle u \to v \rangle$ entre u y v en G'.
Sabemos que el grafo G' tiene más componentes conexas que G por la definición de puente. Vamos a ver dos casos: uno en el que G es conexo, y otro en el que no.

<u>G conexo:</u> Como G' tiene más componentes conexas que G, tiene que tener por lo menos dos, o sea, ser disconexo. Esto quiere decir que existen dos vértices u' y v' tales que no hay camino entre u' y v' en G'. Pero G sí es conexo, o sea que hay un camino $\langle u' \to u \rangle$ entre u' y u en G. Podría ser que $\langle u' \to u \rangle$ contuviese la arista

(u, v). Forzosamente esta arista debería ser la última de $\langle u' \to u \rangle$, y entonces habría un camino $\langle u' \to v \rangle$ entre u' y v que no contendría la arista (u, v). No tengo ganas de definir un nuevo vértice w que sea u o v dependiendo de esto, así que digamos sin pérdida de generalidad que $\langle u' \to u \rangle$ no contiene a (u, v).
Por otro lado, también existe un camino $\langle v' \to v \rangle$ en G. Por la misma razón de antes, o $\langle v' \to v \rangle$ no contiene a (u, v), o existe otro camino $\langle v' \to u \rangle$ que no contiene a (u, v). Supongamos que de estos dos caminos, $\langle v' \to u \rangle$ no contiene a (u, v). Entonces todas las aristas de estos dos caminos pertenecen también a G', y luego podemos concatenar los caminos $\langle v' \to u \rangle$ y $\langle u' \to u \rangle$ para conseguir un camino entre v' y u' en G'. Por lo tanto sí había camino entre esos dos vértices en G', lo cual contradice lo que habíamos dicho antes.
En cambio, supongamos que $\langle v' \to v \rangle$ no contiene a (u, v). Entonces si concatenamos $\langle v' \to v \rangle$ con $\langle u \to v \rangle$ y después con $\langle u' \to u \rangle$, vemos que hay un camino entre v' y u' en G', lo cual también contradice lo que habíamos dicho antes.
En ambos casos llegamos a una contradicción, y luego si G es conexo no puede haber camino entre u y v en G'.

<u>G disconexo:</u> En este caso, G tiene más de una componente conexa. Supongamos que (u, v) pertenece a la componente conexa X de G.
Sea Y otra componente conexa de G. Todas las aristas de Y están contenidas también en G'. Por lo tanto sigue existiendo camino entre todo par de vértices de Y en G'. Esto nos dice que todas las componentes conexas de G diferentes de X siguen existiendo en G'. Los únicos vértices que pueden pertenecer a componentes conexas diferentes a las de G son entonces los vértices de X. Como en G' hay más componentes conexas que en G, y cada componente conexa debe tener al menos un vértice, no pueden pertenecer todos los vértices de X a la misma componente conexa en G'. Con esto vemos que (u, v) es puente en G sí y sólo si es puente en el subgrafo de G que solo tiene los vértices de X y las aristas de G entre vértices de X. Esto se llama el *subgrafo inducido de G por X*, y lo escribimos $G[X]$.
El conjunto X es una componente conexa de G, y entonces $G[X]$ es conexo. Volvemos luego al caso anterior en el que nuestro grafo era conexo, y en el que habíamos deducido que no había camino entre u y v. Como no hay camino entre u y v en $G[X]$, tampoco lo hay en G, ya que X es su componente conexa. Así llegamos a lo que queríamos demostrar.

$\Leftarrow$) Supongamos que no hay camino entre u y v en G'. En particular, G' es disconexo, es decir, tiene por lo menos dos componentes conexas. Si G es conexo, ya sabemos entonces por definición que (u, v) es un puente. Supongamos entonces que G es disconexo.
Sea X la componente conexa de G que contiene a (u, v). Vimos antes que (u, v) es puente en G si y sólo si es puente en $G[X]$. Volvemos a lo mismo: si no hay camino entre u y v en el grafo conexo $G[X]$, entonces (u, v) es puente en $G[X]$ por lo visto recién, y por lo tanto (u, v) es puente también en G. $\square$

Ahora sí, vamos a resolver este problema usando las propiedades del árbol enraizado que nos da DFS.

El siguiente teorema sale de la teórica.

Teorema 1

El árbol de recorrido de DFS sobre un grafo conexo G solamente tiene tree edges y backward edges.

Sabiendo esto, vamos a primero analizar qué aristas de un árbol DFS pueden ser puentes.

Lema 2

Una backward edge no puede ser un puente.

Demostración

Como G es conexo, entre todo par de vértices existe un camino de tree edges en cualquier árbol DFS T de G. En particular, si (u, v) es una backward edge de T, existe otro camino entre u y v compuesto sólo por tree edges, y por Lema 1 la arista (u, v) no es un puente. □

Las únicas aristas que pueden ser un puente entonces son las tree edges de algún árbol DFS de G. Vamos a ver cómo podemos identificarlas.

Definición (Cobertura)

Decimos que una backward edge b *cubre* a una tree edge (u, v) de un árbol DFS T con u padre de v si b conecta un descendiente de u con un ancestro de v en T.

Lema 3

Una tree edge de un árbol DFS T de un grafo conexo G es un puente si y sólo si no hay ninguna backward edge que la cubra.

Demostración

$\Rightarrow$) Supongamos que (u, v) es una tree edge de T que es un puente. Queremos ver que no hay ninguna backward edge que la cubra. Supongamos entonces que existe una backward edge b que cubre a (u, v), es decir, que va de un ancestro de v llamado `ancestro` a un descendiente de u llamado `descendiente`. Existe entonces un camino (posiblemente vacío) $\langle v \to$ `descendiente`$\rangle$ de tree edges entre v y `descendiente`, y un camino (posiblemente vacío) $\langle$`ancestro` $\to u\rangle$ de tree edges entre `ancestro` y u. Concatenando estos dos caminos con la arista b, obtenemos

un camino entre v y u que no contiene a (u, v), y por Lema 1 la arista (u, v) no es un puente. ABSURDO! que vino de suponer que (u, v) era un puente.

$\Leftarrow$) Supongamos que (u, v) es una tree edge de T que no es un puente. Queremos ver que hay una backward edge que la cubre.
Como (u, v) no es un puente, existe un camino $\langle u \to v \rangle$ entre u y v en $G \setminus (u, v)$ por Lema 1. Este camino no puede consistir solamente de aristas de T, porque sino habría dos caminos distintos en T entre dos vértices, lo cual contradice la definición de árbol. Por lo tanto, $\langle u \to v \rangle$ tiene que contener al menos una arista que no es de T. Por Teorema 1, todas las aristas que no pertenezcan a T son backward edges. Por lo tanto, $\langle u \to v \rangle$ tiene al menos una backward edge.
Además, $\langle u \to v \rangle$ tiene sólo tree edges y backward edges, las cuales por definición conectan solamente ancestros con descendientes. Como $\langle u \to v \rangle$ comienza en un ancestro de v y termina en un descendiente de u, por un argumento inductivo se puede ver que $\langle u \to v \rangle$ contiene sólo ancestros de v y descendientes de u. En particular debe existir una arista b que conecte un ancestro v llamado `ancestro` con un descendiente de u llamado `descendiente`. Si b es una tree edge, entonces existen dos caminos entre `ancestro` y `descendiente` en T: el que pasa por b, y el que pasa por (u, v). Esto contradice la definición de árbol. Por lo tanto, b es una backward edge. Como b conecta un ancestro de v con un descendiente de u, b cubre a (u, v). Esto es lo que queríamos demostrar. $\square$

Perfecto, entonces la idea del algoritmo sería esta:

1. Hacer un recorrido DFS sobre el grafo G desde un vértice cualquiera.
2. Identificar las tree edges y backward edges de G.
3. Para cada tree edge (u, v), ver si hay alguna backward edge que la cubra. Si no hay ninguna, entonces (u, v) es un puente. Sino, no lo es.

Ya sabemos cómo hacer el punto 1 en $\Theta(n + m)$.

El 2 podemos hacerlo fijándonos si el vértice al que llega una arista ya estaba visitado o no cuando lo recorremos. Si no estaba visitado, entonces la arista es una tree edge. Si ya estaba visitado, entonces la arista es una backward edge. Esto no me cambia la complejidad.

El 3 es un poco más complicado. Supongamos que hacemos lo más obvio: para cada tree edge recorremos todas las backward edges y vemos si alguna la cubre. Esto nos da una complejidad de por lo menos m por cada tree edge, y eso sin ni siquiera saber cuánto cuesta ver si una backward edge cubre a una tree edge. Como hay $n - 1$ tree edges, nos da una complejidad total de $\Omega(nm)$, que es más que lo que queremos. Necesitamos hacer algo más inteligente.

Lo que nos podemos dar cuenta es de lo siguiente:

Lema 4

Si una backward edge b cubre a una tree edge (u, v), entonces pasa una de dos cosas:

1. O b cubre también a una arista que incide en un hijo de v; o
2. b incide en v.

Demostración

Otra forma de decir esto es que si b no incide en v, entonces cubre a una arista que incide en un hijo de v. Sea `descendiente` el descendiente de u que pertence a b, y `ancestro` el ancestro de v que pertenece a b. Como `descendiente` $\neq v$, sabemos que `descendiente` es un descendiente también de v. Como la relación de ancestro es transitiva, `ancestro` también es un ancestro de `descendiente`. Por lo tanto b cubre a la arista que conecta a `descendiente` con su padre, que es una arista que incide en un descendiente de v. Dado esto, por inducción se puede ver que b cubre a una arista que incide en un hijo de v. □

Esto nos da una idea de cómo podemos hacer el 3 de forma más eficiente. Para cada tree edge (u, v), nos guardamos el mínimo nivel alcanzado por una backward edge que cubre a (u, v), o sea, el nivel del ancestro de más arriba. Llamemos a esto $\mathsf{minNivelCubierto}(v)$, donde v es hijo de u. Para calcular esto para una tree edge (u, v), nos vamos a fijar en el $\mathsf{minNivelCubierto}(\texttt{hijo})$ para cada `hijo` de v en el árbol DFS, y lo vamos a comparar con el mínimo nivel alcanzado por una backward edge que incide en v. Llamemos $\mathsf{nivel}(v)$ al nivel de v, y $\mathsf{minBackwardEdge}(v)$ al mínimo nivel alcanzado por una backward edge que incide en v. Entonces,

$$\mathsf{minNivelCubierto}(v) = \text{mín} \left\{ \begin{array}{l} \mathsf{nivel}(v), \\ \mathsf{minBackwardEdge}(v), \\ \text{mín}_{\texttt{hijo} \in \mathsf{hijos}(v)} \{\mathsf{minNivelCubierto}(\texttt{hijo})\} \end{array} \right\}.$$

Con esto calculado, podemos saber fácilmente si una tree edge (u, v) está cubierta o no: si $\mathsf{minNivelCubierto}(v) < \mathsf{nivel}(v)$, la arista (u, v) está cubierta, sino, no. Y una vez que sabemos si está cubierta, podemos decidir si es un puente o no usando el Lema 3.

Podríamos demostrar que la función $\mathsf{minNivelCubierto}(v)$ efectivamente es el mínimo nivel alcanzado por una backward edge que cubre a (u, v). Esto se los dejo como ejercicio.

Retomando entonces: para calcular lo que nos pide el punto 3, después de correr el DFS calculamos para cada vértice v su $\mathsf{minNivelCubierto}(v)$ de forma recursiva. No hace falta calcularlo dos veces para un mismo vértice, así que guardamos los valores en un arreglo, como hacemos siempre en programación dinámica.

Analicemos cuánto nos cuesta calcular $\mathsf{minNivelCubierto}(v)$ para cada v. Primero, $\mathsf{nivel}(v)$ podemos calcularlo como $\mathsf{nivel}(u) + 1$, donde u es el padre de v, así que es $O(1)$. Podemos calcular $\mathsf{minBackwardEdge}(v)$ recorriendo todas las aristas que inciden

en v, viendo cuáles son backward edges, y tomando el mínimo nivel alcanzado por una de ellas. Esto nos cuesta $O(\text{grado}(v))$. Y el mínimo de los $\text{minNivelCubierto}(\texttt{hijo})$ entre los hijos de v lo podemos calcular en $O(\text{grado}(v))$ también, porque máximo hay esa cantidad de hijos, y para cada uno de ellos ya calculamos $\text{minNivelCubierto}(\texttt{hijo})$. O sea que calcular cada $\text{minNivelCubierto}(v)$ nos cuesta $O(\text{grado}(v) + 1)$. La sumatoria de los grados de todos los vértices da $2m$, así que sumando esto para los n vértices, nos da una complejidad de $O(2m + n) = O(m + n)$.

Perfeeecto, conseguimos que los tres pasos nos cuesten $O(m+n)$. **Pero ojo**, estuvimos asumiendo hasta ahora que G era conexo. ¿Cómo lo adaptamos para G disconexo? Fácil, corremos un DFS por cada componente conexa, que ya vimos que no cambia la complejidad.

Para completar, acá está el código en limpio en Python.

```python
def dfs(lista_de_adyacencias,
        vertice,
        visitados,
        padres,
        niveles,
        min_nivel_cubierto):
    visitados[vertice] = True
    for vecino in lista_de_adyacencias[vertice]:
        if not visitados[vecino]:
            niveles[vecino] = niveles[vertice] + 1
            padres[vecino] = vertice
            dfs(lista_de_adyacencias,
                vecino,
                visitados,
                padres,
                niveles,
                min_nivel_cubierto
            )
            min_nivel_cubierto[vertice] = min(
                min_nivel_cubierto[vertice], min_nivel_cubierto[vecino]
            )
        elif vecino != padres[vertice]: # es un back edge
            min_nivel_cubierto[vertice] = min(
                min_nivel_cubierto[vertice], niveles[vecino]
            )


def no_cubiertos(n, padres, niveles, min_nivel_cubierto):
    return [
        vertice for vertice in range(n)
        if (
            min_nivel_cubierto[vertice] >= niveles[vertice] and
            padres[vertice] != -1
        )
```

```
    ]


def puentes(lista_de_adyacencias):
    n = len(lista_de_adyacencias)
    visitados = [False] * n
    padres = [-1] * n
    niveles = [0] * n
    min_nivel_cubierto = list(range(n))
    for vertice in range(n):
        if not visitados[vertice]:
            niveles[vertice] = 0
            dfs(lista_de_adyacencias,
                vertice,
                visitados,
                padres,
                niveles,
                min_nivel_cubierto
            )

    return [
        (padres[vertice], vertice)
        for vertice in no_cubiertos(n, padres, niveles, min_nivel_cubierto)
    ]
```

Y para rellenar la última hoja, una foto de un gatito.