Técnicas para Explorar Gráficas

Emmanuel Peto Gutiérrez

María De Luz Gasca Soto

Mayo, 2020

Departamento de Matemáticas,
Facultad de Ciencias, UNAM.

Índice general

1 Búsqueda en Amplitud, BFS **7**
1.1 Análisis de complejidad 10
1.2 El árbol-BFS 10

2 Búsqueda en Profundidad, DFS **11**
2.1 Análisis de complejidad 15
2.2 Bosque DFS 15
2.3 Aplicaciones de DFS 16
2.3.1 Vértices de corte 16
2.3.2 Puentes 18
2.3.3 Bloques 19

3 *Topological Sorting*, TS **21**
3.1 Análisis de complejidad 25

Bibliografía **26**

Luz Gasca y Emmanuel Peto. F.C, UNAM

Técnicas Básicas: BFS, DFS, TS

El primer problema que se encuentra al intentar diseñar un algoritmo que involucre gráficas es cómo revisar la entrada. Una estructura de una dimensión (como un arreglo o una lista) se puede revisar fácilmente de forma lineal. Explorar una gráfica no es tan sencillo. Presentamos dos algoritmos para explorar una gráfica: **búsqueda en amplitud**, *Breadth-first search*, BFS y **búsqueda en profundidad**, *Depht-first search*, DFS.

De ambos algoritmos se puede obtener información de la gráfica como: encontrar los vértices alcanzables desde un vértice fuente s, y obtener las componentes conexas. Del algoritmo BFS se puede obtener la distancia de un vértice fuente s a todos los vértices alcanzables desde s. Con el algoritmo DFS se pueden encontrar los vértices de corte y los bloques de la gráfica.

Otro algoritmo importante, que revisaremos, es *Topological Sorting* también llamado **ordenamiento topológico**. Aquí, cada tarea se representa con un vértice y existe una arista dirigida (o arco) de la tarea x a la tarea y si la tarea y no puede empezar hasta que x termine; la gráfica debe ser acíclica. Con el algoritmo TS se pueden organizar tareas.

Luz Gasca y Emmanuel Peto. F.C, UNAM

Capítulo 1

Búsqueda en Amplitud, BFS

La **Búsqueda por amplitud**[1] es uno de los algoritmos más simples para explorar una gráfica y arquetipo para otros algoritmos importantes. Por ejemplo, un algoritmo para determinar árboles generadores de peso mínimo y un algoritmo para rutas más cortas usan ideas similares al BFS.

Dada una gráfica $G = (V, E)$ y un vértice distinguido s, denominado fuente, el algoritmo BFS explora sistemáticamente las aristas de G para descubrir todos los vértices alcanzables desde s[2]. Calcula la distancia (mínimo número de aristas) de s a cada vértice alcanzable. También produce un árbol BFS, con raíz s, que contiene todos los vértices alcanzables. Para cualquier vértice v alcanzable desde s, la trayectoria en el árbol-BFS de s a v corresponde a la ruta más corta desde s a v en G, en cuanto a número de aristas.

La Técnica de Búsqueda en amplitud se llama así porque expande la frontera entre los vértices descubiertos y los no descubiertos uniformemente a través de la amplitud de la frontera. Es decir, el algoritmo visita todos los vértices a distancia k desde s antes de descubrir cualquier vértice a distancia $k + 1$.

Para hacer un seguimiento del progreso, BFS distingue los vértices visitados de los no visitados. Esto se puede hacer colocando una variable lógica como atributo del vértice, digamos, `visitado`, que empieza en falso y cambia a verdadero cuando el algoritmo descubre al vértice.

El algoritmo BFS construye un árbol, inicialmente conteniendo solo su raíz, la cual es el vértice fuente s. Cada vez que se descubre un vértice no visitado v mientras se revisa la lista de adyacencias de un vértice visitado u, el vértice v y la arista (u, v) se agregan al árbol. Se dice que u es el **predecesor** o **padre** de v en el árbol-BFS. Ya que cada vértice es descubierto a lo más una vez, éste tiene a lo más un padre. Las relaciones de **ancestro** o **descendiente** en un árbol se definen con respecto a la raíz s: si u está en la trayectoria en el árbol desde s a v, entonces u es un ancestro de v y v es un descendiente de u.

En el Cuadro 1.1 se muestra un pseudocódigo para el algoritmo BFS donde la gráfica está representada con listas de adyacencias. La lista de adyacencias es un atributo de un vértice y se llama `ady`. El padre (o predecesor) de un vértice está representado con el

[1] Normalmente se refiere a este algoritmo como BFS por su nombre en inglés: *Breadth-first search.*

[2] Es decir, todos los vértices v para los cuales exista un camino de s a v.

atributo `p`, y éste es nulo si el vértice no tiene predecesor. El atributo `d` guarda la distancia desde el origen s hasta el vértice u calculada por el algoritmo. El algoritmo también usa una cola Q, para administrar los vértices descubiertos o visitados.

```
BFS(G, s)
 1 for each u ∈ G.V
 2    u.visitado = false
 3    u.d = ∞
 4    u.p = null
 5 s.visitado = true
 6 s.d = 0
 7 Q = ∅
 8 Q.enqueue(s)
 9 while Q ≠ ∅
10    u = Q.dequeue()
11    for each v ∈ u.ady
12       if !v.visitado
13          v.visitado =true
14          v.d = u.d + 1
15          v.p = u
16          Q.enqueue(v)
```

Cuadro 1.1: Pseudocódigo para BFS.

Las líneas 1-4 establecen las condiciones iniciales de los vértices: marca a todos como no visitados, la distancia `d` la establece en infinito para cada vértice u y el padre de cada vértice se define como `null`. La línea 5 marca al vértice s como visitado. La línea 6 inicializa s.`d` en 0, ya que la distancia de un vértice a sí mismo es 0. Las líneas 8-9 inicializan la cola Q conteniendo solo el vértice s.

El ciclo **while** itera mientras aun haya vértices en la cola, los cuales son vértices descubiertos para los cuales su lista de adyacencias no ha sido explorada por completo.

La Figura 1.1 muestra el progreso de`BFS` en una gráfica de ejemplo, donde el vértice inicial es $s = v_1$. Los vértices visitados se marcan de gris y las aristas que pertenecen al `árbol-BFS` se muestran sombreadas. A la derecha está la cola Q y abajo de cada vértice se muestra el atributo distancia, `d`. La distancia inicial de cada vértice se muestra dentro de los nodos, en la gráfica. Por ejemplo, expliquemos un par de iteraciones. Empezamos en v_1, su distancia será 0. El vértice v_1 mete a la cola a los vértices v_5 y v_0, con distancia 1. En la siguiente iteración saca a v_5 de la cola y mete a sus vecinos: v_2 y v_6, con distancia 2.

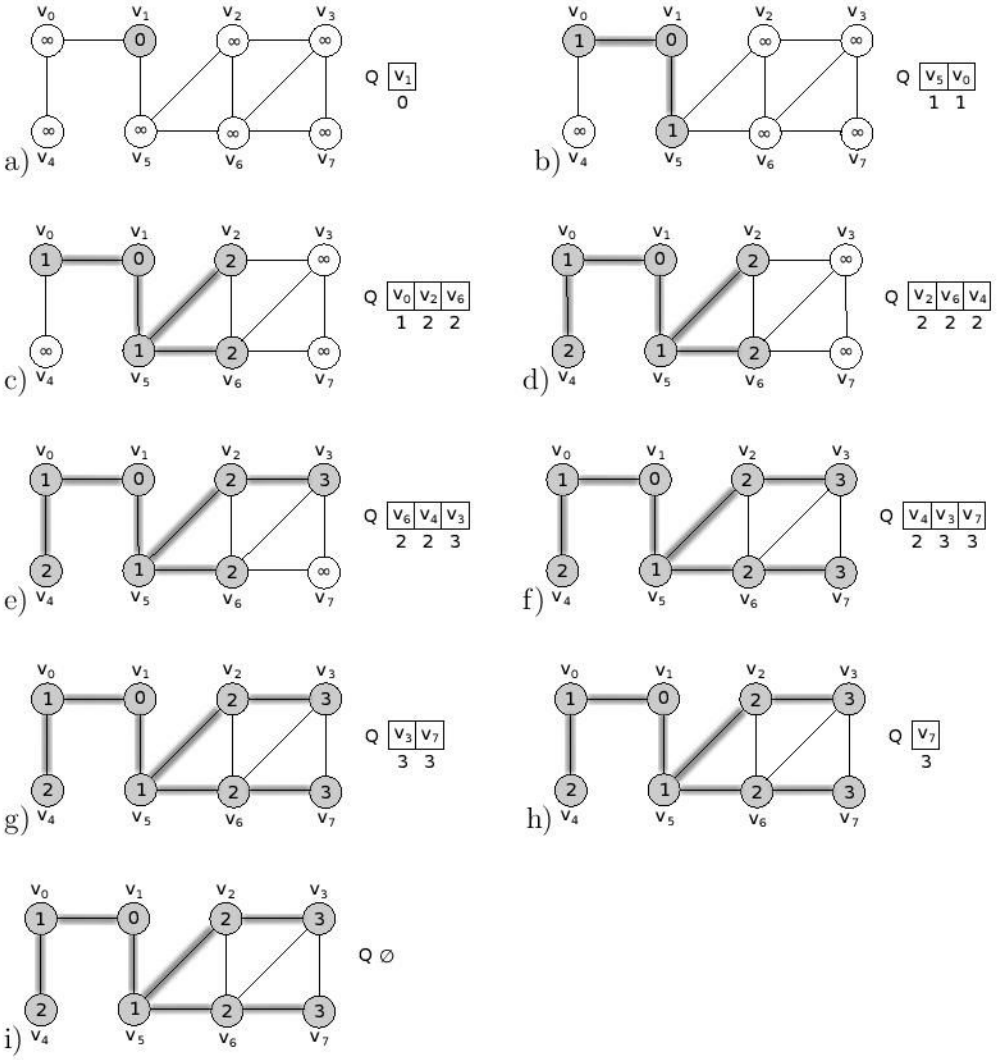

Figura 1.1: Ejecución del algoritmo BFS.

1.1. Análisis de complejidad

Después de la inicialización, el algoritmo BFS nunca cambia un vértice a `no visitado`, y así la prueba en la línea 12 asegura que cada vértice sea encolado, agregado a la cola Q, a lo más una vez, y por lo tanto es sacado de Q a lo más una vez. Las operaciones de encolar, `enqueue`, y desencolar, `dequeue`, toman tiempo constante. Por lo tanto el tiempo total dedicado a operaciones de cola es $O(|V|)$. Como el algoritmo revisa la lista de adyacencias de cada vértice solo cuando el vértice es eliminado de Q, revisa cada lista a lo más una vez. Ahora bien, ya que la suma de las longitudes de las listas de adyacencias es $\Theta(|E|)$, el tiempo total invertido en revisar las listas de adyacencias es $O(|E|)$. La parte de inicialización toma tiempo $O(|V|)$, así que el tiempo de ejecución total del algoritmo BFS, en el peor de los casos, es $O(|V| + |E|)$.

1.2. El árbol-BFS

El algoritmo BFS construye un árbol mientras explora la gráfica, como se ilustra con las aristas sombreadas en la Figura 1.1. El árbol corresponde al atributo p. Más formalmente, para una gráfica $G = (V, E)$ con origen s, se define la **subgráfica predecesora** de G como $G_p = (V_p, E_p)$, donde

$$V_p = \{v \in V : v.p \neq null\} \cup \{s\} \text{ y}$$

$$E_p = \{(v.p, p) : v \in V_p - \{s\}\}$$

La subgráfica predecesora G_p es un árbol BFS si V_p consiste de vértices alcanzables desde s y, para todo $v \in V_p$, la subgráfica G_p contiene una única trayectoria de s a v que es también un camino más corto desde s a v en G. Las aristas en E_p se llaman **aristas de árbol**.

El siguiente procedimiento imprime la ruta más corta desde el origen s a un vértice v.

```
imprimeRuta(v)
1 String salida = v
2 temp = v.p
3 while temp ≠ null
4     salida = temp + salida
5     temp = temp.p
6 print(salida)
```

Cuadro 1.2: Imprimir la ruta más corta.

Luz Gasca y Emmanuel Peto. F.C, UNAM

Capítulo 2

Búsqueda en Profundidad, DFS

El objetivo en este capítulo es introducir la técnica para explorar gráficas denominada **búsqueda en profundidad**[1]. Este algoritmo resulta ser una poderosa técnica para resolver diversos problemas de la teoría de gráficas.

El algoritmo DFS deriva del problema de idear un método para visitar sistemáticamente todas las cámaras en un laberinto. Intuitivamente, DFS comienza en una cámara y procede a lo largo de pasillos de cámara en cámara dejando un marcador en cada cámara visitada. La búsqueda siempre procede a lo largo de pasillos no marcados siempre que sea posible; es decir, va a continuar tan profundamente como se pueda en el laberinto sin revisitar una cámara. Cuando una cámara C es alcanzada y todas sus cámaras vecinas ya han sido visitadas, la búsqueda revisita la cámara que fue visitada inmediatamente antes de la primera visita a C; es decir, si B fue descubierta justo antes de C, entonces se regresa a B. Dicho laberinto se puede representar con una gráfica cuyos vértices corresponden a las cámaras y las aristas a los pasillos. A continuación describimos de manera más formal la aplicación del algoritmo DFS en una gráfica.

Supongamos que la gráfica G con $V(G) = \{v_1, v_2, ...v_n\}$ está representada con listas de adyacencias. En una búsqueda en profundidad de G, el vértice que está siendo visitado actualmente se designa como el **vértice activo**. Iniciamos la búsqueda en profundidad para G seleccionando un primer vértice a visitar (digamos, v_1). El vértice v_1 será entonces el primer vértice activo, y le asignamos la etiqueta 1. Después, seleccionamos al primer vértice adyacente a v_1 (el primero en la lista de adyacencias), lo etiquetamos con 2, y éste se convierte en el nuevo vértice activo. También designamos al vértice v_1 como el predecesor del vértice etiquetado con 2.

En general, sea u el vértice activo en la búsqueda y supongamos que no todos los vértices en la componente de G que contiene a u han sido visitados. Entonces procedemos de la siguiente manera:

- Si existen vértices no visitados adyacentes a u, seleccionamos al primer vértice no visitado de la lista de adyacencias de u y lo etiquetamos con con la siguiente etiqueta disponible. El vértice que acaba de ser etiquetado se convierte en el nuevo vértice

[1] Abreviado como DFS por su nombre en inglés: *Depht-First Search.*

activo y designamos al vértice u como su predecesor.

- Si, por otra parte, todos los vértices adyacentes a u ya han sido visitados, entonces **retrocedemos**[2] o regresamos (es decir, se revisita) al vértice que estaba activo antes de que u fuera visitado por primera vez y lo designamos como el vértice activo actual. Este paso se repite hasta que todos los vértices de la componente han sido visitados.

- Si no han sido visitados todos los vértices de G, entonces se elige a un vértice no visitado de otra componente y se marca como el siguiente vértice activo.

En el cuadro 2.1 se muestra un pseudocódigo para la técnica DFS, que aplica el algoritmo de búsqueda en profundidad a cada vértice que no haya sido visitado. Esto es para garantizar que se aplique a todas las componentes de la gráfica.

```
DFS(G)
1 for each u ∈ G.V
2    u.visitado = false
3    u.p  = null
4 t = 0            // marca de tiempo, variable global
5 for each u ∈ G.V
6    if !u.visitado
7       DFS_Visit(G, u)
```

Cuadro 2.1: Pseudocódigo para DFS.

El proceso que realiza la búsqueda en una componente es DFS_Visit. El Cuadro 2.2 presenta la versión recursiva de éste. La búsqueda en profundidad genera una marca de tiempo para cada vértice y se realiza con t, la cual es una variable global. La etiqueta (o marca de tiempo) se guarda en el atributo `dfi` [3] del vértice, llamado **índice de profundidad**, y representa el momento en el que el vértice fue descubierto. Si un vértice adyacente a u no ha sido visitado entonces se aplica la búsqueda en profundidad sobre éste; es decir, se aplica DFS_Visit de forma recursiva. El atributo `p` guarda al padre de cada vértice.

En el Cuadro 2.3 se muestra la versión iterativa de DFS_Visit. Para éste se considera que cada vértice tiene un iterador de la lista de adyacencias (`it`)[4]. El iterador mantiene un seguimiento de los vértices que ya han sido revisados en la lista de adyacencias del vértice activo v. El algoritmo también utiliza una pila S, el vértice en el tope de la pila es el activo.

[2] En inglés: *backtrack*.
[3] *depth-first search index*
[4] En Java se puede obtener con `listIterator()`, que es un método de `List`.

```
DFS_visit(u)
1 t = t + 1
2 u.dfi = t
3 u.visitado = true
4 for each v ∈ u.ady
5    if !v.visitado
6       v.p = u
7       DFS_Visit(v)
```

Cuadro 2.2: Pseudocódigo para DFS_Visit recursivo.

```
DFS_visit(u)
 1 S = ∅
 2 t = t + 1
 3 u.dfi = t
 4 S.push(u)
 5 while S ≠ ∅
 6    v = S.peek()
 7    if v.it.hasNext()
 8       w = v.it.next()
 9       if !w.visitado
10          w.visitado = true
11          w.p  = v
12          w.dfi = t
13          t = t + 1
14          S.push(w)
15    else
16       S.pop()
```

Cuadro 2.3: Pseudocódigo para DFS_Visit iterativo.

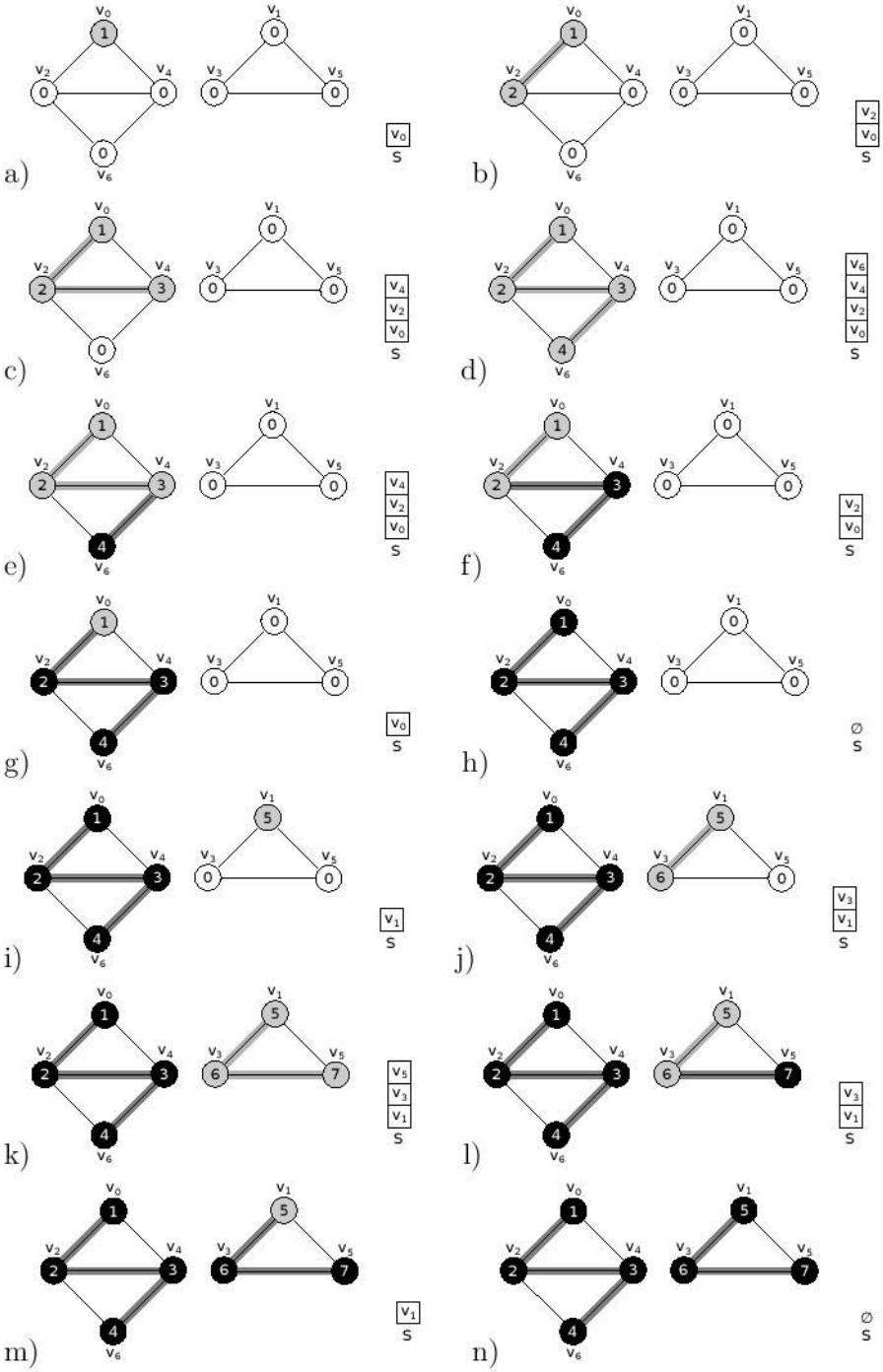

Figura 2.1: Ejemplo de DFS en una gráfica.

En la Figura 2.1 presentamos un ejemplo, una gráfica donde se ejecuta el algoritmo DFS y se utiliza una pila S. Los vértices blancos son los que no han sido visitados, los grises son los que están actualmente en la pila y los negros son los que ya salieron de la pila. Las aristas sombreadas son las que pertenecen al árbol generado por DFS.

2.1. Análisis de complejidad

¿Cuál es el tiempo de ejecución de DFS? Las líneas $1-3$ y $5-7$ toman $\Theta(|V|)$, except por la llamada a DFS_Visit. Considerando la versión recursiva del proceso DFS_Visit, el cual se llama exactamente una vez por cada vértice $v \in V$. Para el vértice u el tiempo de ejecución de las líneas $4-7$ es $|u.ady|$. Ya que $\sum_{v \in V} |v.ady|$ es $\Theta(|E|)$, el costo de ejecutar el proceso DFS_Visit en todos los vértices es $\Theta(|E|)$. Por lo tanto, el costo total de ejecutar DFS, en el peor de los casos, es $\Theta(|V| + |E|)$.

2.2. Bosque DFS

La subgráfica predecesora de DFS se define como: $G_p(V, E_p)$, donde

$$E_p = \{(v.\mathtt{p}, \mathtt{v}) : \mathtt{v} \in \mathtt{V} \quad \text{y} \quad v.\mathtt{p} \neq \text{null}\}.$$

G_p es una subgráfica generadora de G y se le llama **bosque-DFS**. Si la gráfica G es conexa, entonces G_p es un árbol generador, llamado **árbol-DFS**.

Sea F el bosque-DFS (G_p), cada arista de G que no pertenece a F se denomina **arco de regreso**, *back-edge*. Necesariamente, cada *back-edge* une dos vértices en la misma componente en G y, por lo tanto, en una componente de F. Es costumbre dibujar cada *back-edge* en un árbol enraizado mediante una línea punteada dirigida hacia atrás. En la Figura 2.2 se muestra la gráfica de la Figura 2.1 con los *back-edges* como flechas punteadas y las aristas del bosque como flechas no punteadas.

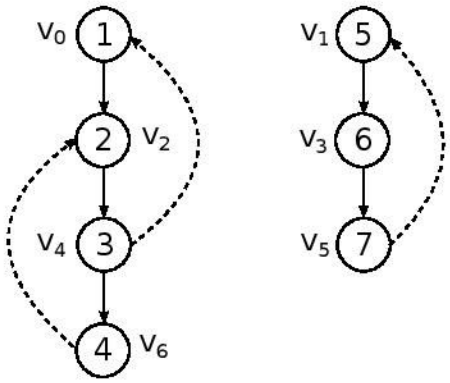

Figura 2.2: Bosque resultante de aplicar DFS a una gráfica.

Teorema 2.1 Cada *back-edge* (u, v) de una gráfica G une un ancestro y un descendiente. En particular, si el vértice u es visitado antes que v en una búsqueda en profundidad de G, entonces el vértice v es descendiente de u,[2].

2.3. Aplicaciones de DFS

Con ayuda de la búsqueda en profundidad, podemos determinar los vértices de corte, puentes y bloques de una gráfica. Primero, daremos la terminología adicional.

Sea v un vértice de la gráfica G. Una **rama** de G en v es una subgráfica conexa maximal H de G que contiene a v tal que v no es un vértice de corte de H. El número de ramas que contienen a v es igual al número de bloques que contienen a v. La Figura 2.3 muestra un vértice v y sus ramas.

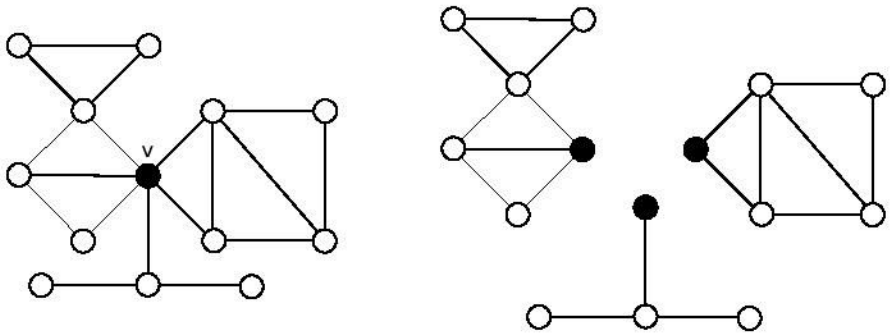

Figura 2.3: Las ramas de un vértice

2.3.1. Vértices de corte

Ahora vamos a presentar el primero de dos resultados los cuales, juntos, caracterizan a los vértices de corte. Ya que un vértice es de corte en una gráfica si y sólo si es un vértice de corte de alguna componente de la gráfica, se puede restringir a gráficas conexas.

Teorema 2.2 Sea T un árbol-DFS de una gráfica conexa G y sea r la raíz del árbol T. Entonces r es un vértice de corte si y sólo si tiene al menos dos hijos en T, [2].

Después de haber ejecutado DFS desde la raíz r se puede comprobar fácilmente si ésta es un vértice de corte con el algoritmo en el Cuadro 2.4.

Ahora, ¿cómo saber si el resto de los vértices son de corte o no? Para completar la caracterización de los vértices de corte, necesitamos un término adicional. Sea F un bosque-DFS de una gráfica G y supongamos que v es un vértice que pertenece a alguna componente T de F. El **punto bajo**, *low-point*, (v.`low`) de v es el valor más bajo u.`dfi` de un vértice u de T que puede ser alcanzado desde v por una $(u, v)-$trayectoria que

```
corte_raiz(r)
1 n = 0
2 for each w ∈ v.ady
3     if w.p == v
4         n++
5 if n > 1
6     print(r+"es un vértice de corte")
```

Cuadro 2.4: Algoritmo para verificar si la raíz es Vértice de corte.

consiste de aristas de T seguido por a lo mas un *back-edge*. Ya que la ruta de v a u puede ser trivial, se sigue que $v.\texttt{low} \leq v.\texttt{dfi}$ para todo vértice $v \in G.V$.

En realidad, $v.\texttt{low} < v.\texttt{dfi}$ sólo si hay un *back-edge* de v o un descendiente de v a un ancestro de v. La Figura 2.4 muestra una gráfica con su árbol-DFS correspondiente, donde el número a la izquierda es el índice `dfi` y el de la derecha es el punto bajo, `low`. El punto bajo de v_5, por ejemplo, es 4 ya que v_5, v_6, v_4 es una $(v_5, v_4)-$trayectoria en F que usa exactamente un *back-edge*: (v_6, v_4).

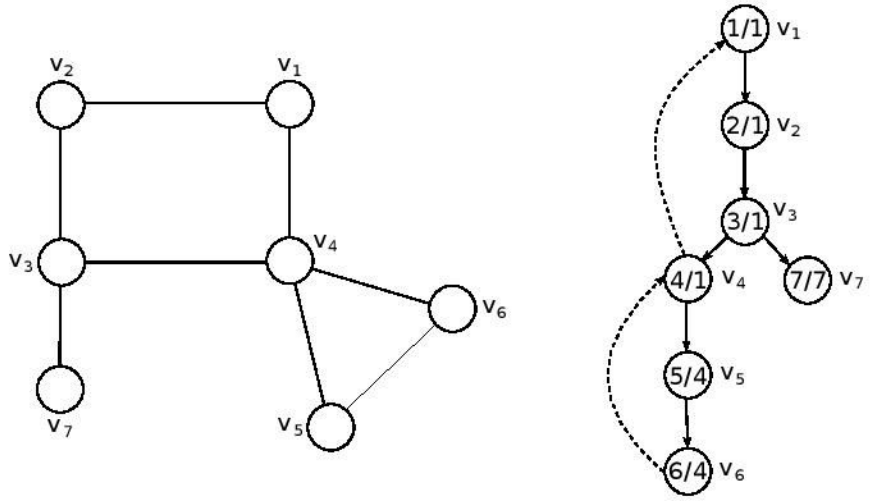

Figura 2.4: Los puntos bajos de los vértices de una gráfica.

Teorema 2.3 Sea T un árbol-DFS de una gráfica conexa G, y sea u un vértice que no es la raíz de T. Entonces u es un vértice de corte de G si y sólo si tiene un hijo v en T tal que $v.\texttt{low} \geq u.\texttt{dfi}$, [2].

Después de haber calculado el índice de DFS, (`dfi`) en los vértices de una gráfica, podemos calcular los puntos bajos con el algoritmo mostrado el cuadro 2.5. En el mismo algoritmo se pueden identificar los vértices de corte.

```
LowPoint(v)
 1 v.low = v.dfi
 2 for each w ∈ v.ady
 3    if w.dfi ≥ v.dfi
 4       LowPoint(w)
 5       if w.low ≥ v.dfi and v.p ≠ null
 6          print(v + "es un vértice de corte")
 7       v.low = min(v.low, w.low)
 8    else
 9       if v.p ≠ w
10          v.low = min(v.low, w.dfi)
```

Cuadro 2.5: Cálculo de puntos bajos (*lowpoint*) de los vértices en una gráfica conexa.

Por definición, un punto bajo $v.\texttt{low}$ es el mínimo entre los tres:

1. $v.\texttt{dfi}$
2. El mínimo $w.\texttt{dfi}$ entre todos los *back-edges* de la forma (v, w).
3. El mímimo $w.\texttt{low}$ entre todas las aristas de árbol de la forma (v, w).

En la línea 1 se aplica la primera regla para un punto bajo, en la línea 10 se aplica la segunda regla y en la 7 la tercera regla. En la línea 3 se comprueba si la arista (v, w) es del árbol; es decir, si w es hijo de v. En la línea 9 se verifica si la arista (v, w) es un *back-edge*; es decir, que $w.\texttt{dfi} < v.\texttt{dfi}$ y w no sea padre de v. En la línea 5 se revisa si v cumple la condición para ser vértice de corte; esto es, que $w.\texttt{low} \geq v.\texttt{dfi}$. Además se tiene que comprobar que v no sea la raíz del árbol; es decir, que su padre sea diferente a null.

Ya que el algoritmo LowPoint explora la gráfica en profundidad, igual que DFS, la complejidad del proceso es $O(|V| + |E|)$.

2.3.2. Puentes

Hay problemas donde es esencial conocer si una gráfica tiene puentes. Con ayuda de la búsqueda en profundidad, se puede presentar una caracterización de los puentes. Observemos que si una arista e es un puente de una gráfica G, entonces e pertenece a todos los bosques-DFS de G. Como en el caso de los vértices de corte, es suficiente con caracterizar a los puentes de gráficas conexas.

Teorema 2.4 Sea T un árbol-DFS de una gráfica conexa G. Una arista (u, v) de G, con $u.\texttt{dfi} < v.\texttt{dfi}$, es un puente de G si y sólo si (u, v) es arista de T y $v.\texttt{low} > u.\texttt{dfi}$. [2].

Ya que los puentes solo pueden ser aristas del árbol-DFS, basta con comprobar que una arista de la forma $(v.\mathtt{p}, \mathtt{v})$ cumpla $v.\mathtt{low} > v.\mathtt{p.dfi}$ para saber si es puente[5].

2.3.3. Bloques

Hay situaciones en las que es valioso conocer los bloques de una gráfica. En esta sección discutimos cómo emplear la técnica DFS para desarrollar un algoritmo que encuentre los bloques de una gráfica.

Cuando se lleva a cabo una búsqueda en profundidad, por supuesto, la búsqueda procede de componente a componente en G. Supongamos que G es conexa. Si G tiene al menos dos bloques, entonces G tiene al menos dos bloques terminales (bloques que contienen solo un vértice de corte).

Supongamos que la búsqueda inicia en un vértice r que no es un vértice de corte de G. Independientemente de si r pertenece a un bloque terminal o no, la búsqueda procederá eventualmente a un bloque terminal B por primera vez con la arista (u, v), donde u es el vértice de corte de B en G. Por el algoritmo LowPoint 2.5 se puede determinar si u es un vértice de corte.

Desde el momento en el que se entra a B hasta que ha sido completamente explorado, la búsqueda se queda en B (no pasa a otro bloque hasta que se termina de explorar B). Así, v es el único hijo de u en B;, es decir, cualquier vértice en B es descendiente de v. Por esta razón, se utiliza una pila[6]. Los vértices de B se colocan en una pila en el orden en el que fueron visitados por primera vez. Si, después del retroceso $v \to u$, se descubre que u es un vértice de corte, entonces se imprimen (y se remueven) los vértices desde el tope de la pila hasta v, incluyendolo. Estos vértices, junto con u inducen a B. Al eliminar estos vértices de la pila, esencialmente se remueve el bloque terminal B de G. El proceso continúa ahora en u, y se tiene un bloque menos que descubrir. De hecho, el algoritmo detecta de forma exitosa bloques terminales hasta que todos los bloques de G han sido determinados.

Si la búsqueda comienza en un vértice de corte r, entonces el algoritmo detecta todos los bloques que no contienen a r y los imprime como se menciona. Los bloques que contienen a r se pueden encontrar usando el Teorema 2.4. Si se retrocede a r, a través de una arista (x, r), y se descubre que hay vértices no visitados adyacentes a r, entonces se sabe que un bloque ha sido determinado. El bloque es el inducido por los vértices en la pila, desde el tope hasta x (incluyéndolo) junto con r.

[5]Es evidente que $v.\mathtt{p.dfi}$ es mayor a $v.\mathtt{dfi}$, ya que $v.\mathtt{p}$ fue visitado antes.

[6]Es una pila diferente a la usada en el DFS iterativo.

Bibliografía

[1] Udi Manber, *Introduction to Algorithms, A Creative Approach.* Addison-Wesley. 1989.

[2] Gary Chartran and O. Oellerman *Applied and Algorithmic Graph Theory.* McGraw-Hill, Inc. 1993.

[3] Thomas H. Cormen and C. E. Leiserson and R. L. Rivestn *Introduction to Algorithms.* The MIT Press, 2002.