Tarea 02 Árboles Y Grafos

Josue Peña Atencio - 8935601

19 de agosto de 2018

Índice

1. Problema 22-2

1.1. Índice (a)

Si la raíz del arbol G_π (árbol de búsqueda DFS) de un grafo G tiene más de dos hijos, digamos u y v, la raíz es un punto de articulación.

Ya que si existiera el caso en el que u y v estuvieran conectados (que existiera un borde entre ellos) entonces ambos se encontrarían en el mismo lado del árbol G_π, debido a que estamos usando DFS para generar el árbol.

Es decir que al visitar alguno de los dos nodos conectados a la raíz y entre ellos mismos, la secuencia de recorrido sería $source \to u \to v$, y no $source \to u \to ..., source \to v$.

Luego, si u y v no tienen un borde, al quitar la raíz del árbol, no habría ningún camino desde u para llegar a v sin pasar por *source* y viceversa. La raíz es entonces, un punto de articulación.

1.2. Índice (b)

Si nos detenemos en cualquier nodo v que no sea la raíz en el árbol G_π de algún grafo G, si este tiene un hijo s el cual **sí** tiene un *back edge* a un antecesor p de v entre sus descendientes, esto quiere decir que hay un camino alternativo para llegar desde p hasta s, a través de la arista entre p y el hijo de s, y luego desde ese hijo hasta s.

Todo esto quiere decir que si removemos el vértice v y todas las aristas que lo conectan, el grafo no se separaría, porque todavía existe por lo menos un camino alternativo al que se acaba de eliminar.

Luego si este camino alternativo no existe, es decir, que no haya un *back edge* de algún hijo de s con algún antecesor p de v, no hay forma de llegar desde p hasta s, más que por el camino que pasa por v. Si se elimina v y todas las aristas que lo conectan, el grafo se separaría. Así, v es un punto de articulación.

1.3. Índice (c)

Como el atributo de los vértices *low* requiere de los descendientes de cada nodo, el algoritmo para poder calcular todos los *low* puede ser recursivo sobre el árbol G_π de la búsqueda BFS de algún grafo G.

El caso base para este algoritmo sería cuando nos encontramos con una hoja v en el árbol G_π.
$v.low$ es el mínimo entre su tiempo de descubrimiento ($v.d$) y el tiempo de descubrimiento del vértice con el que v hace *back edge* ($w.d$).

El caso inductivo sería cuando nos encontremos con una rama u en G_π. En este caso, $u.low$ es el mínimo entre tres cosas: $v.d$, $w.d$ y $s.low$ (si w denota el nodo con el que u hace *back edge* y s alguno de los hijos de u).

Entonces, para poder calcular todos los $v.low$ de todos los vértices basta con aplicar el caso base, o aplicar el caso inductivo para todos los hijos del vértice v que se esté visitando en G_π.

1.4. Índice (d)

Podemos usar el algoritmo descrito en el índice (c) para pre-calcular todos los valores *low* y d de todos los vértices del grafo G a computar. Así solamente tenemos que verificar para cada nodo v, $v.low \geq v.d$.

Ya que si la desigualdad se cumple, eso quiere decir que en el sub-árbol de v en G_π no hay un *back edge* a un nodo que fue descubierto antes que v, lo cual nos indica de inmediato que v es un punto de articulación.

1.5. Índice (e)

Un *bridge* no puede existir en un ciclo simple, ya que si éste se remueve y desconecta al grafo (aumenta el número de componentes conectados), no haría parte de un ciclo simple en primer lugar.

Es decir, ya que como un ciclo simple es una secuencia de bordes

$$(u_1, v_1), (v_1, u_2), ..., (v_n, u_1)$$

Si quitamos cualquiera de esos bordes, el grafo no se desconectaría ya que siempre habría un camino alternativo para llegar desde cualquier vértice a cualquier otro dentro del mismo ciclo simple. Eso prueba que ninguno de los bordes en un ciclo simple puede ser un *bridge*.

1.6. Índice (f)

Sabemos que un *bridge* (u, v) tiene la característica que ambos u y v pueden ser puntos de articulación o que solo de uno de ellos lo sea, mientras que el vértice resultante no está conectado a ningún otro nodo (es decir, que su grado es 1 porque solo estaría conectado a otro vértice, con el que forma el borde tipo *bridge*).

Con esa información, con la forma para computar todos los puntos de articulación en un grafo descrita en (d) y con el hecho de que la verificación de si un vértice es de grado 1 se puede hacer mirando si tiene exactamente un vértice conectado (complejidad constante), podemos computar todos los *bridges* de cualquier grafo G.

1.7. Índice (g)

Como se mencionó en (e) un *bridge* nunca hace parte de un ciclo simple, lo cual implica que todo borde que haga parte de un componente biconectado no es un *bridge*.

Luego, para probar que dos componentes biconectados no son el mismo (que están particionados o separados), basta con observar si estos no comparten algún borde. Supongamos dos componentes biconectados B_1, B_2 y un borde *bridge* v tal que v particiona a los componentes.

Si v *sí* particiona a B_1 y B_2, eso quiere decir que no existen bordes en común entre estos, lo cual es verdad ya que si existieran, estos tendrían que pasar por v y v tendría que ser parte de un ciclo simple entre B_1 y B_2, lo cual ya probamos que es falso. De esa forma, los bordes tipo *bridge* particionan los componentes biconectados en un grafo.

1.8. Índice (h)

Para poder marcar a cada vértice con las condiciones requeridas, podemos usar el algoritmo para encontrar todos los *bridges* del grafo G, y al encontrarlos los eliminamos. De esta forma, solamente quedarían bordes entre todos los componentes biconectados, lo cual haría sencillo marcar cada nodo en estos componentes con un mismo número *e.bcc* para cada componente particular.

2. Ejercicio 4: *Butterflies*

Si queremos verificar que los m juicios realizados en los pares de mariposas es consistente con los dos grupos en los que estas pueden ser clasificadas (A y B), tenemos que recorrer todas las conexiones que existen entre todas las mariposas que fueron sometidas a los juicios.

Para esto podemos emplear BFS, con unas mínimas modificaciones.

n será el numero de nodos (mariposas) y m el de los arcos (juicios). Las etiquetas *Same*, *Different* y *Ambiguous* serán la clasificación de cada arista en el conjunto de aristas E en el grafo del problema, el cual es no dirigido. A continuación muestro el algoritmo para poder resolver este problema:

```
def check_judgements(A, B, E, source):
  visited[source] = True
  Q = queue()
  Q.push(source)
  is_consistent = True
  while Q is not empty and is_consistent == True:
    v = Q.pop()
    for all edges (v, u) in E:
      if visited[u] == False:
        if (v, u).label == "Same":
          is_consistent = (v and u belong to the same set)
        else if (v, u).label == "Different":
          is_consistent = (v and u belong to different sets)
        visited[u] = True
        Q.push(u)
  return is_consistent
```

Si lo que es asignado a $is_consistent$ llega a ser falso (es decir, en los casos que los requisitos para que el juicio sea consistente no se cumplan) sabremos inmediatamente que todo el grafo lo es, por lo que retornaría $False$.

De lo contrario, éste se recorrería todo y no se encontraría ninguna inconsistencia, por lo que retornaría $True$.

3. Ejercicio 11: *Virus*

Para poder responder la pregunta de si un computador C_a que fue infectado en un tiempo x pudo haber infectado a algún otro computador C_b en un tiempo y debemos mirar cronológicamente los computadores con los que el C_a o *source* interaccionó, y así mismo, con los que estos se comunicaron posteriormente a su infección.

Podemos visualizar a todos los computadores en la red como los nodos en un grafo dirigido, y el tiempo en el que interaccionaron como el peso de las aristas que los conectan. E denota el conjunto de los *triplets*. A continuación muestro cómo se realizaría el recorrido en este grafo usando DFS:

```
def inspect_network(E, Ca, Cb, time_x, time_y):
  assert time_x <= time_y
  visited[Ca] = True
  S = stack()
  S.push(Ca)
  while S is not empty and visited[Cb] == False:
    v = S.pop()
    visited[v] = True
    for all triplets (v, u, t) in E:
      if visited[u] == False and time_x <= t <= time_y:
        S.push(u)
        visited[u] = True
  return visited[Cb]
```

De esta forma, si Cb sí fue infectado entre los tiempos $time_x$ y $time_y$, este será visitado por el algoritmo anterior, por lo cual la respuesta sería $True$.

Si no se llegó a infectar, eso quiere decir que a partir del computador Ca y del tiempo $time_x$ no hubo ningún tipo de contacto entre los que éste se comunicó y el computador Cb en el lapso de tiempo estipulado, por lo cual retornaría $False$.

4. Material consultado

1. Articulation Points and Bridges
   `https://www.hackerearth.com/practice/algorithms/graphs/articulation-points-and-bridges/tutorial/`
2. Biconnected Components
   `https://www.hackerearth.com/practice/algorithms/graphs/biconnected-components/tutorial/`
3. Algorithms: Graph Search, DFS and BFS.
   `https://www.youtube.com/watch?v=zaBhtODEL0w`

4. 5.2 Articulation Point and Biconnected Components.
   https://www.youtube.com/watch?v=jFZsDDB0-vo

5. Graph Data Structure And Algorithms
   https://www.geeksforgeeks.org/graph-data-structure-and-algorithms/

6. Back edges in a graph
   https://stackoverflow.com/questions/44494426/back-edges-in-a-graph/44494705