Ejes Puentes - Puntos de Articulación - Componentes Biconexas

Problemas, Algoritmos y Programación

Septiembre de 2016

Contenidos

DFS may refer to...

- Discrete Fourier Series
- Distributed File System
- Diego Fernández Slezak
- Depth First Search
- Docenas Finitas de otros Significados (?)

DFS may refer to...

- ~~Discrete Fourier Series~~
- ~~Distributed File System~~
- ~~Diego Fernández Slezak~~
- Depth First Search
- ~~Docenas Finitas de otros Significados (?)~~

DFS

- *"Depth-first search yields valuable information about the structure of a graph."*

  Introduction to Algorithms, Cormen et al.

- A diferencia del BFS, que suele utilizarse con la misión específica de resolver el problema de caminos mínimos desde un origen v, el algoritmo de DFS suele usarse para obtener información útil sobre el grafo en sí, que puede ser luego utilizada por algoritmos posteriores.

Idea general

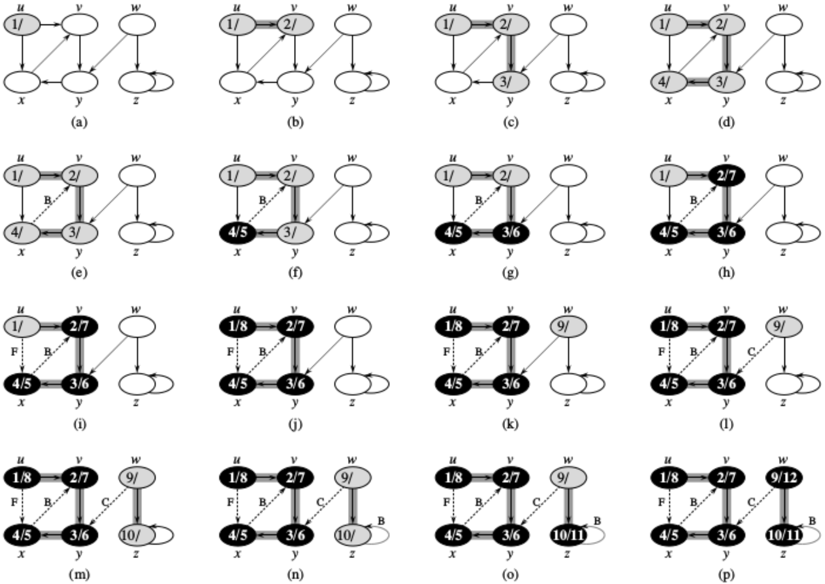
- La idea del DFS consiste en recorrer el grafo caminando por las aristas, utilizando siempre las aristas disponibles (aún no utilizadas) del último nodo descubierto.
- Para visualizar más fácil la ejecución del algoritmo, es conveniente pensar en que los nodos se pintan de tres colores: Blanco, Gris y Negro.
- Los nodos inician todos pintados de blanco.
- Al descubrir un nodo y comenzar a procesarlo, se lo pinta de gris. Mientras haya nodos blancos, se hace una exploración de DFS desde cualquier nodo blanco.
- Desde el nodo actual en procesamiento, se exploran las aristas salientes, y si alguna llega a un nodo blanco nuevo, se pinta de gris y se sigue explorando desde allí recursivamente.
- Luego de considerar y eventualmente recorrer todas las aristas de un nodo, se completa su procesamiento y se pinta de negro, volviendo la recursión a su padre.

Pseudocódigo

```
dfsRecursivo(G,v):
    pintar v de gris en G
    para cada w blanco vecino de v en G:
        dfsRecursivo(G,w)
    pintar v de negro

dfs(G)
    para cada v en G:
        pintar v de blanco
    para cada v de G:
        si v es blanco:
            dfsRecursivo(G,v)
```

DFS paso por paso



Timestamp

- Además, es muy útil para razonar sobre el algoritmo, y para utilizar en algoritmos posteriores, asociar a cada nodo dos timestamps:
- Un primer timestamp $d[i]$, que para cada nodo i, indica el tiempo en que fue descubierto (se pintó de gris).
- Un segundo timestamp $f[i]$, que para cada nodo i, indica el tiempo en que fue finalizado (se pintó de negro).
- El timestamp es simplemente una variable global, que comienza en 1 y se incrementa hasta $2n$ durante la ejecución del algoritmo.
- Nos permite ordenar los $2n$ eventos de descubrimiento y finalización de un nodo de manera total en el tiempo.

Timestamp (continuado)

- Una propiedad muy importante y útil para razonar, es que los tiempos de finalización y terminación de los distintos nodos presentan estructura de paréntesis.
- Es decir, si armamos un string de longitud $2n$ con paréntesis que abren en las posiciones $d[i]$ y paréntesis que cierran en las $f[i]$, tendremos una cadena de paréntesis balanceada.
- Por ejemplo, una propiedad muy importante es que un descendiente de un nodo en un arbol de dfs, necesariamente tiene sus parentesis contenidos en el de su ancestro.

Pseudocódigo

```
i entero global
dfsRecursivo(G,v):
    descubrimiento[v].setear(i)
    i++
    para cada w vecino de v en G:
        si descrubrimiento[v] no seteado:
            dfsRecursivo(G,w)
    finalizacion[v].setear(i)
    i++

dfs(G)
    i = 1
    para cada v de G:
        si descrubrimiento[v] no seteado:
            dfsRecursivo(G,v)
```

Clasificacin de las aristas

El algoritmo de DFS nos induce una clasificación de aristas en 4 tipos:

- Tree edge
  - Viaja a un nodo blanco
  - Es con la que se descubre un nodo por primera vez
  - Viaja al hijo del nodo actual en un árbol de DFS
- Back edge
  - Viaja a un nodo gris
  - Viaja a un ancestro del nodo actual
- Forward edge
  - Viaja a un nodo negro con descubrimiento posterior al nodo actual
  - Viaja a un descendiente (no necesariamente hijo) del nodo actual
  - Solo aparece en grafos dirigidos
- Cross edge
  - Viaja a un nodo negro que finaliza antes que el descubrimiento del nodo actual
  - No viaja ni a un ancestro ni a un descendiente
  - Solo aparece en dirigidos

Observaciones útiles

- Los descendientes de un nodo v serán exactamente los nodos blancos alcanzables por un camino de nodos blancos con origen en v, en el instante en que se descubre (white-path-theorem).
- Un grafo es acíclico (dirigido o no) si y solo si un recorrido de DFS no encuentra ninguna back-edge.
- El DFS genera un spanning forest (DFS-forest), donde cada árbol está formado por las tree-edges.
- Si el grafo es no dirigido, el DFS construye un DFS-forest con un spanning tree de cada componente conexa.
- Si el grafo es dirigido, no hay una relación clara entre los árboles del DFS-forest y las "componentes" del grafo original para ninguna definición razonable de componente.

Observaciones útiles (¡Más!)

- Las back-edges siempre están involucradas en un ciclo (concretamente, el que se completa con las tree edges que bajan en sentido inverso)
- Las cross-edges forman un subgrafo acíclico (ya que por definición, siempre van de un nodo a otro con un tiempo de finalización menor)
- Topological sort: Es tomar los vertices en orden inverso de finalizacion, gracias a que (en un DAG):

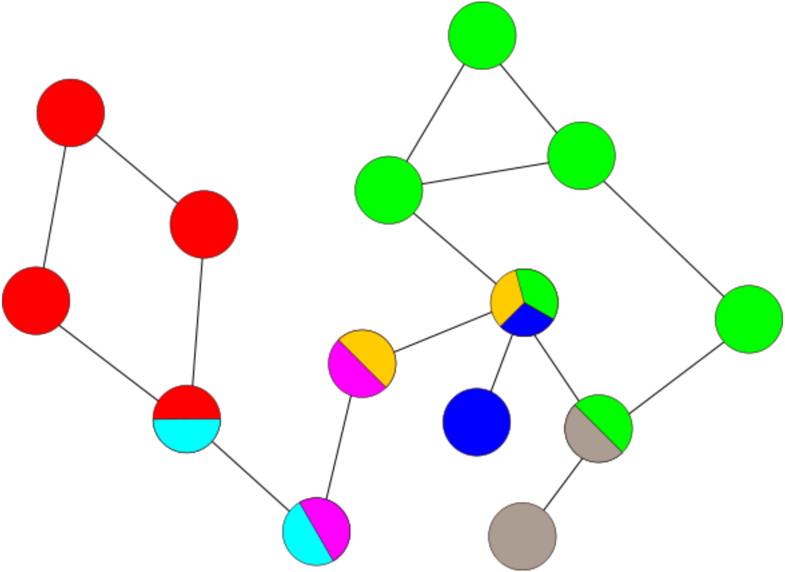
$$a \to b \Rightarrow f[a] > f[b]$$

- Notar que no hace falta correr un sort: si cada vez que finalizamos un nodo lo agregamos a una lista, quedan al final en orden de finalizacin.

Definiciones

- En un grafo no dirigido, un punto de articulación es un nodo tal que al removerlo del grafo, la cantidad de componentes conexas aumenta.
- En un grafo no dirigido, un puente es un eje tal que al removerlo del grafo, la cantidad de componentes conexas aumenta.
- Un grafo no dirigido es biconexo si es conexo y no tiene puntos de articulación.
- En un grafo no dirigido, una componente biconexa es un subgrafo biconexo maximal.

Dibujito



Observaciones

- Las componentes biconexas son conexas, por lo que para simplificar trabajaremos durante la clase de hoy con grafos conexos.
- Los puentes son exactamente las componentes biconexas de 2 nodos (y necesariamente, una arista).
- Las componentes biconexas no particionan los nodos (a diferencia de las componentes conexas o fuertemente conexas (próxima clase)).
- Las componentes biconexas particionan las aristas del grafo (ignorando los nodos aislados), de acuerdo a la relación de equivalencia de cociclidad (pertenencia a un mismo ciclo simple).

Observaciones (más)

- Podríamos obtener algo parecido a la condensación de un grafo dirigido para este caso.
- Si miramos el dibujo y nos imaginamos que cada componente biconexa se transforma en un nodo, lo que queda tiene "Pinta de árbol".
- Sin embargo, simplemente contraer cada componente a un único nodo y unir componentes que se tocan no funciona, pues el resultado puede no ser árbol.

Observaciones (más)

- Podríamos obtener algo parecido a la condensación de un grafo dirigido para este caso.
- Si miramos el dibujo y nos imaginamos que cada componente biconexa se transforma en un nodo, lo que queda tiene "Pinta de árbol".
- Sin embargo, simplemente contraer cada componente a un único nodo y unir componentes que se tocan no funciona, pues el resultado puede no ser árbol.
- Solución: utilizar un nodo por cada componente biconexa, y también un nodo por cada punto de articulación.

Block-cut tree

- Conectamos un punto de articulación a las componentes biconexas que lo contienen (siempre serán al menos 2). El resultado es un árbol.
- Notar que si bien todo árbol es bipartito, aquí la bipartición tiene un significado claro en relación al problema: por un lado puntos de articulación, por otro componentes.
- Además, todas las hojas están en el mismo conjunto de la bipartición, lo cual no ocurre en cualquier árbol, ya que en nuestro caso los puntos de articulación nunca son hoja.
- A este árbol se lo conoce como el block-cut tree del grafo.
- Esto es un árbol siempre que asumamos que el grafo original es conexo, sino es un bosque.

Cálculo mediante DFS

- La idea central es computar mediante recursión durante el recorrido del DFS, un valor $low[v]$ que para cada nodo, indique la menor distancia de la raíz del árbol de DFS actual a la que es posible saltar, desde alguna parte del sub-árbol de DFS con raíz en v, llamandole $depth[v]$ a esta distancia.
- Este valor puede computarse simplemente como el mínimo entre el *low* de los hijos del nodo actual, y la profundidad mínima a la que llega una back-edge que sale del nodo actual.

Cálculo mediante DFS (puentes)

- Observación: Un tree-edge del nodo v a uno de sus hijos w es puente si y solo si, $low[w] \geq depth[w]$, es decir, no existe ningún back-edge que permita salir del subárbol con raíz en w.
- Observación: Un back-edge nunca puede ser puente.

Cálculo mediante DFS (puntos de articulación)

- Observación: La raíz de un árbol de DFS es un punto de articulación si y solo si tiene más de un hijo.
- Observación: Un nodo v distinto de la raíz es punto de articulación, si y solo si, para alguno de sus hijos w, se cumple $low[w] \geq depth[v]$.

Cálculo mediante DFS (componentes biconexas)

- Para calcular las componentes biconexas, basta agregar una pila al DFS anterior:
- Cada vez que recorremos una arista, agregarla a la pila.
- Cada vez que volvemos de una llamada recursiva por un eje (v, w), con v padre de w, chequeamos si ahí termina una componente biconexa, es decir, $low[w] \geq depth[v]$
- Si ese es el caso, desapilamos aristas hasta desapilar la arista (v, w), que fue la primera que pusimos al bajar y marca el comienzo de la componente. Todas esas forman la componente biconexa.

Pseudocódigo

```
dfs(G,v,d,padre,pila):
    depth[v] = d
    low[v] = d
    if v != padre:
        apilar en pila (padre,v)
    para todo w vecino de v distinto del padre:
        si depth[w] = -1:
            low[v] = min(low[v],dfs(G,w,d+1))
            if low[w] >= depth[v]:
                marcar v como punto de articulacion
                reportarComponente(pila,(v,w))
            if low[w] >= depth[w]:
                marcar (v,w) como puente
        else:
            low[v] = min(low[v],depth[w])
    return low[v]
biconexas(G):
    para todo v en G:
        depth[v] = -1
    dfs(G,raiz,0,raiz,pilaVacia)
```

Pseudocódigo

```
reportarComponente(pila,(v,w)):
    componente = vacia
    agregar tope de la pila a componente
    mientras el tope de la pila no sea (v,w):
        desapilar tope de la pila
        agregar tope de la pila a componente
    desapilar tope de la pila
    reportar componente
```

Bibliografía

- *Introduction to Algorithms, 2nd Edition*. MIT Press.
  Thomas H. Cormen
  Charles E. Leiserson
  Ronald L. Rivest
  Clifford Stein
  Sección 22 (DFS y temas relacionados)
- Tarjan, R. Depth first search and linear graph algorithms. SIAM J Comput. 1972;1:146160.(Link)

Presentación TP2

A continuación presentaremos el TP2...