Grafos

Pedro O. Pérez M., MTI

Análisis y diseño de algoritmos
Tecnológico de Monterrey

pperezm@tec.mx

04-2019

Contenido

Definición

Un grafo no direccionado $G = (V, E)$ consiste de una colección de vértices, V y una colección de arcos, E. Representamos cada arco, $e \in E$, como un subcojunto de dos elementos $e = \{u, v\}$ siendo $u, v \in V$, llamando a u y v los puntos terminales de e.

Un grafo direccionado $G = (V, E)$ consiste de una colección de vértices, V y una colección de arcos, E. Representamos cada arco, $e \in E$, como un par ordenado de dos elementos $e = (u, v)$ siendo $u, v \in V$. Llamamos a u el punto inicial y a v el punto final de e.

- Un grafo no direccionado está conectado si, para cada par de nodos u y v, existe un camino de u a v.
- Un grafo direccionado está fuertemente conectado si, para cada par de vértices u y v, existe un camino de u a v y de v a u.
- En un grafo no direccionado $G = (V, E)$ una secuencia de nodos P = [v_1, v_2, ..., v_{k-1}, v_k] con la propiedad de que cada par consecutivo v_i, v_{i+1} está conectado por un arco en G, es llamado un camino (path) de v_1 a v_k.
- Un ciclo es un camino P = [v_1, v_2, ..., v_{k-1}, v_k] en el cual, para cualquier $k > 2$, los primeros k˘1 vértices son distintos y $v_1 = v_k$.

Procedure 1 DFS

Input: $u : Vertex, G : Graph, Reached : Set$

```
Mark u as Explored and add to Reached
for each (u, v) in G incident to u do
  if v is not marked Explored then
    DFS(v, G, Reached)
  end if
end for
```

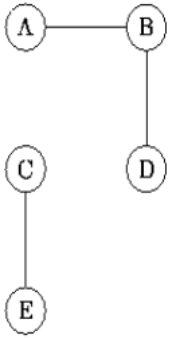
Procedure 2 BFS

Input: u : *Vertex*, G : *Graph*, *Reached* : *Set*

```
Queue Q
Q.enqueue(u)
while Q is not empty do
    e ← Q.dequeue()
    if e is not marked as Explored then
        Mark e as Explored and add to Reached
        for each (e, v) incident to e do
            Q.enqueue(v)
        end for
    end if
end while
```

Subgrafos máximos

Considere un grafo G formado a partir de un gran número de vértices conectados por arcos. G se dice que está conectado si existe un camino entre cualquier par de vértices en G. Por ejemplo, el siguiente grafo no está conectado, porque no hay trayectoria de A a C.



Este grafo contiene, sin embargo, un número de subgrafos que están conectados, uno para cada uno de los siguientes conjuntos de vértices: (A), (B), (C), (D), (E), (A, B), (B,D), (C, E), (A, B, D). Un subgrafo conectado es máximo si no hay vértices y arcos en el grafo original que podrían añadirse al subgrafo y todavía dejarlo conectado. En la imagen anterior, hay dos subgrafos máximos, uno asociada con los vértices (A, B, D) y el otro con los vértices (C, E). Desarrollar un algoritmo para determinar el número de subgrafos máximos conectados de un gráfico dado.

`http://bit.do/eNTvC`

Procedure 3 COUNTING_GRAPHS

Input: G : *Graph*
  Reached : *Set*
  $acum \leftarrow 0$
  Mark all the vertexes in G as *No_Explored*
  for *vertex* in G **do**
    if *vertex* is not marked *Explored* **then**
      $DFS(vertex, G, Reached)$
      $acum \leftarrow acum + 1$
    end if
  end for
  return *acum*

Topological Sort

Un "Topological Sort"de un Grafo Direccionado Acíclico (Directed Acyclic Graph, DAG) es un ordenamiento lineal de los vértices que aparecen en un DAG tal que si el vértice u aparece antes de v es porque existe un arco $(u \rightarrow v)$ en el DAG. Cada DAG tiene al menos, y posiblemente más, "topological sort".

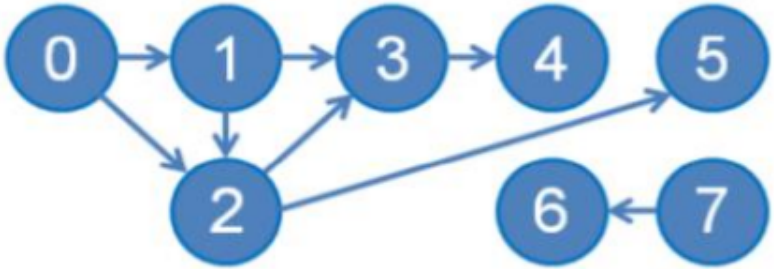
Procedure 4 DFS2

Input: u : *Vertex*, G : *Graph*, *Reached* : *Set*, TS : *Stack*

```
Mark u as Explored and add to Reached
for each (u, v) incident to u do
  if v is not marked Explored then
    DFS2(v, G, Reached, TS)
  end if
end for
TS.push(u)
```

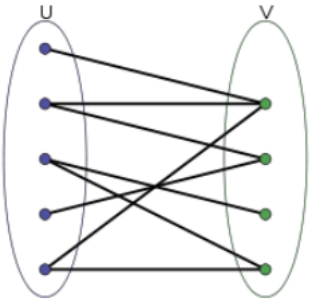
Procedure 5 TOPOLOGIGAL_SORT

Input: G : *Graph*
  Reached : *Set*
  TS : *Stack*
  for each *vertex* in G **do**
    if *vertex* is not marked *Explored* **then**
      $DFS2(G, v, Reached, TS)$
    end if
  end for
  while TS is not empty **do**
    print $TS.top()$
    $TS.pop()$
  end while


0
1
3
4
5
2
6
7

Grafo bipartita

Un grafo bipartita (o bigrafo) $G = (V, E)$ es un grafo cuyos vértices pueden ser divididos en dos conjuntos disjuntos R y S tal que cada arco conecta a un vértice en R con un vértice en S.



Procedure 6 BIGRAPH

Input: G : *Graph*
  Q : *Queue*
  Color : *Array*
  isBipartite : *boolean*
  $INIT(Color, -1)$
  $isBipartite \leftarrow true$
  $vertex \leftarrow$ some vertex in G
  $Q.enqueue(vertex)$
  while Q is not empty **do**
    NEXT SLIDE
  end while
  return *isBipartite*

```
u ← Q.dequeue()
for each (u, v) incident in u do
  if Color[v] = −1 then
    Color[v] ← 1 − Color[u]
    Q.enqueue(v)
  else
    if Color[v] = Color[u] then
      isBipartite ← false
    end if
  end if
end for
```

Punto Articulado

Un punto articulado (o puente) se define como un vértice in un grafo $G = (V, E)$ cuya remoción (todos los arcos que inciden sobre él también son removidos desconecta el grafo G.Un grafo que no tiene ningún punto de articulación se le conoce como biconectado.

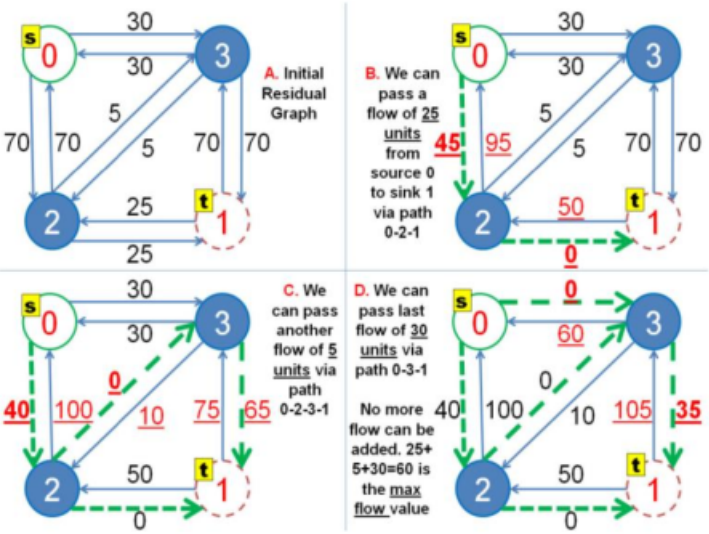
Procedure 7 FIND_POINT

Input: G : *Graph*

```
originalCC ← COUNTING_GRAPHS(G)
for each v in G do
    Remove v and its incident edges
    newCC ← COUNTING_GRAPHS(G)
    if newCC > originalCC then
        return true
    end if
    Restore v and its incident edges
end for
return false
```

Flujo máximo

Una red de flujo es un grafo dirigido, donde cada arco tiene capacidad y cada arco recibe un flujo. La cantidad de flujo de un arco no puede exceder la capacidad del mismo. Un flujo debe satisfacer la restricción de que la cantidad de flujo en un nodo es igual a la cantidad de flujo que sale de ella, a menos que sea una *fuente*, que sólo tiene flujo saliente, o un *sumidero*, que solo tiene flujo saliente.


s
0
3
2
t
1
30
30
5
70
70
5
70
70
25
25
A. Initial Residual Graph
B. We can pass a flow of 25 units from source 0 to sink 1 via path 0-2-1
45
95
50
0
C. We can pass another flow of 5 units via path 0-2-3-1
40
100
0
10
75
65
50
0
D. We can pass last flow of 30 units via path 0-3-1
No more flow can be added. 25+5+30=60 is the max flow value
60
105
35

Procedure 8 MAX_FLOW

Input: G : *Graph*

Setup directed residual graph with edge capacity equal to original graph
$max\_flow \leftarrow 0$
while there exists an augmenting path p from s to t **do**
Find f, the minimum edge weight along the path
Decrease capacity of the ougoing edges
Increase capacity of the incomming egdes
$max\_flow \leftarrow max\_flow + f$
end while
return max_flow

All-Pairs Shortest Path

El algoritmo Floyd – Warshall permite encontrar las rutas más cortas en un grfo ponderado con costos positivos o negativos (pero sin ciclos negativos). Una sola ejecución del algoritmo encontrará los costos de las rutas más cortas entre **todos los pares de vértices**. Aunque no devuelve detalles de las rutas en sí, es posible reconstruir las rutas con simples modificaciones al algoritmo. Las versiones del algoritmo también se pueden usar para encontrar el cierre transitivo de una relación R.

Procedure 9 VERSION1

Input: M : *Adjacent_Matrix*

```
for k ← 1 to M.length do
  for i ← 1 to M.length do
    for j ← 1 to M.length do
      M[i][j] ← M[i][k] and M[k][j]
    end for
  end for
end for
```

Procedure 10 VERSION2

Input: $M : Adjacent\_Matrix$

```
for k ← 1 to M.length do
  for i ← 1 to M.length do
    for j ← 1 to M.length do
      M[i][j] ← MIN(M[i][j], M[i][k] + M[k][j])
    end for
  end for
end for
```