Grafos: recorridos, puntos de articulación, componentes conexas

Los gráficos e imágenes de este apunte se tomaron del libro de Drozdek "Estructuras de Datos en C++" (2001)

Recorridos de grafos y puntos de articulación:

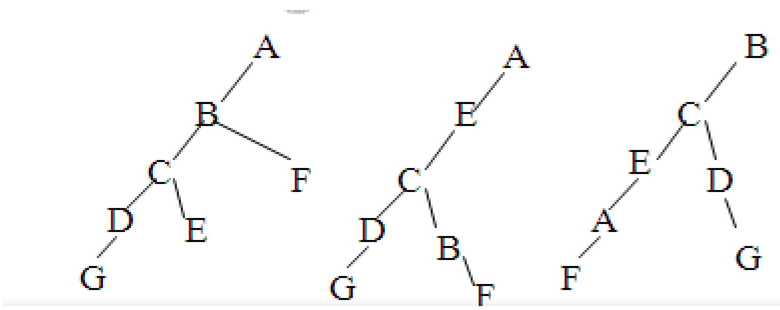
Básicamente los grafos se recorren en profundidad o en anchura.

Recorrido en profundidad (Depth First Search, DFS)

Visita un nodo, a continuación, seleccione uno de sus vecinos y repita el procedimiento. Si no hay vecinos no visitados, dar marcha atrás.

Muchos diferentes árboles de búsqueda primero en profundidad

Arcs nor in the tree are called **back arcs**



Depth First Search

```
void dfs()
{
  for each node  /* the graph may be disconnected, if the
                    graph is connected, don't need the for
                    loop, just start with anyone node */
    if (node is not marked as having been visited)
        visit(node);
}
```

Depth First Search

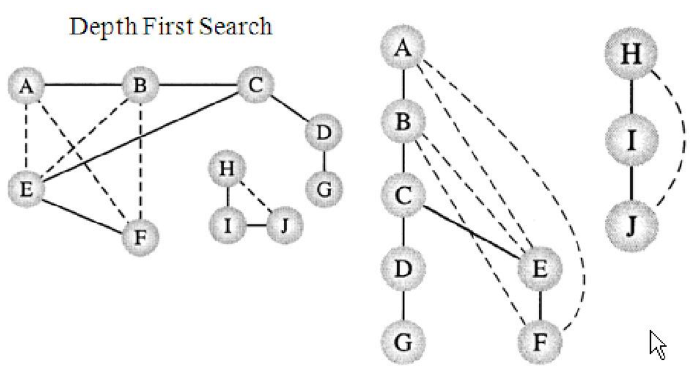
```
void visit(node)
{
  mark node as having been visited;
  process node;
  for all neighbors node1 of node
    if (node1 is not marked as having been visited)
       visit(node1);
}
```

Coste temporal:

El tiempo de ejecución depende de la implementación del grafo. En el algoritmo anteriorla función visit() se llama para cada nodo primero, y luego, para cada nodo adyacente a cada nodo.

Si se usa una representación con matriz de adyacencia, el ciclo chequea cada elemento dela fila correspondiente a cada nodo, y el coste temporal pertenece a O(N^2), siendo N el número de nodos del grafo.

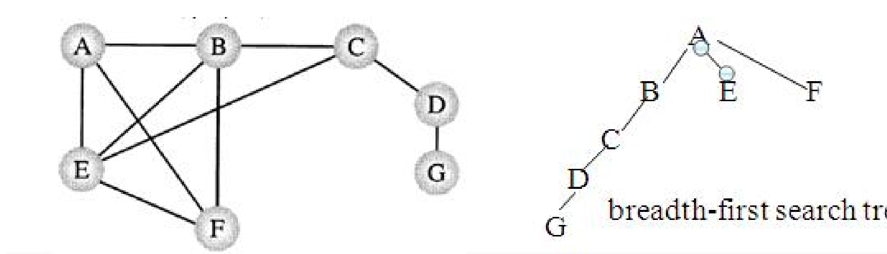
Si se usa una representación con lista de adyacencia, cada arco se visitará dos veces, quedando el oste temporal como O(N + 2 * E), que es O(N + E).



A DFS forest with back-arc marked as dotted line.

Recorrido en anchura o amplitud, o Breadth First Search (BFS)

El DFS se plantea recursivo o usando una cola. Considerando sucesivamente todos los vértices del grafo, plantea visitar todos los adyacentes, luego todos los adyacentes del primer adyacente que no hayan sido visitados, luego todos los adyacentes del segundo adyacente que no hayan sido visitados, y así sucesivamente.



Breadth First Search

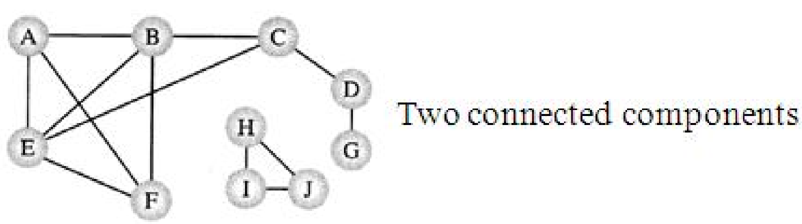
```
void bfs()
{  initialize(queue);
   for each node
     if (node is not marked as having been visited) {
        enq(node, queue);
        while (!empty(queue)) {
           node=deq(queue);
if not visited yet  mark node as having been visited;
           process node;
           for all neighbors node1 of node
              enq(node1,queue);
        }
     }
}
```

Coste temporal: el análisis es análogo al de DFS

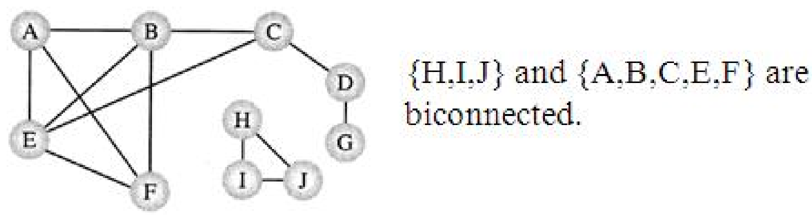
Conectividad y Biconectividad

Un nodo y todos aquello alcanzables desde él forman un componente conectado o conexo (entonces, un gráfo es conexo si sólo tiene un componente conexo).

El recorrido DFS puede modificarse fácilmente para imprimir los componentes conectados de una gráfo.

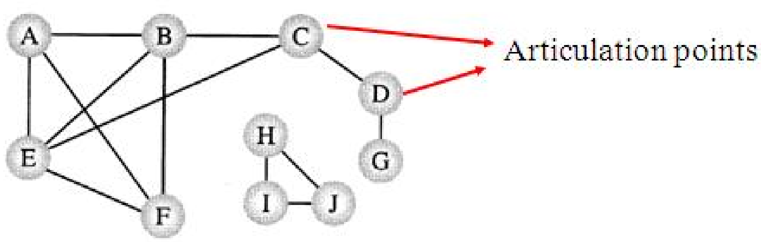


En aplicaciones concretas de grafos, es frecuente tener que establecer no sólo si cada nodo está conectado a todos los demás, sino también si hay al menos dos rutas independientes entre dos nodos. Un conjunto máximo de nodos en los que hay dos caminos diferentes se llama biconexas.

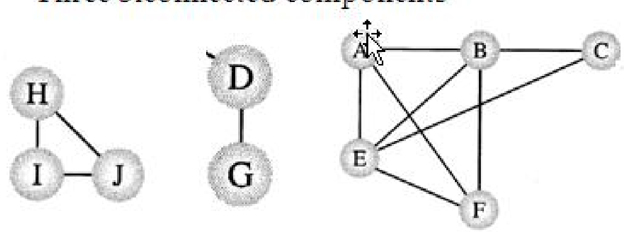


Otra manera de expresar el mismo concepto: no hay puntos *de fallo*, es decir no hay nodos que al ser eliminados junto con las aristas incidentes en ellos, dividen el grafo en dos o más componentes conexas. Los nodos que al ser eliminados junto con las aristas incidentes en ellos 'dividen' al grafo en dos o más componentes conexas se llaman **puntos de articulación**.

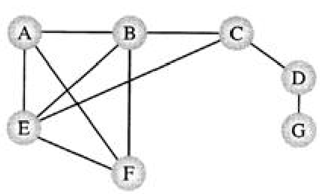
Si un gráfico no contiene puntos de articulación, entonces es biconexo. Si un gráfico contiene puntos de articulación, entonces se puede dividir el gráfico en partes; cada parte es un subgrafo máximo biconexo.



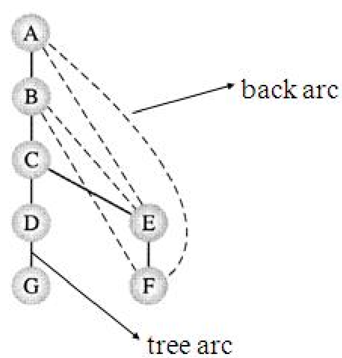
Three biconnected components

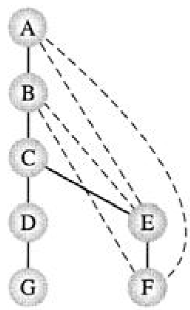


DFS can be used to find the articulation points of a graph, and hence the biconnected components.



In DFS tree, if a node has no back arcs, then it is an articulation point.





node	A	B	C	D	E	F	G
DFS number	1	2	3	4	6	7	5
Child lowest		1	1	4	1	1	5

If (child-lowest>=DFS-number-of- node) then node is an articulation point.

Therefore, C and D are articulation points.

For root, it is an articulation point if it **has more than one child** in the DFS tree.

```
int visitNonRoot(node)
{
  Set node's dfs number to be ++num;
  lowest = num;
  for all neighbors node1 of node
    if (node1's dfs number == 0) { /* unvisited node */
      childlowest = visitNonRoot(node1);
      lowest =  minimum of lowest and childlowest;
      if (childlowest >= the dfs number of node)
        node is an articulation point;
    }
    else /* back-arc */
      lowest =  minimum of lowest and the dfs number of node1;
  return lowest;
  }

void visitRoot(node)
{
  Set node's dfs number to be ++num;
  for all neighbors node1 of node
    if (node1's dfs number == 0) /* unvisited node */
      visitNonRoot(node1);
  if (node has more than one child) /* check root */
    node is an articulation point;
  }
void articulation
{
  Set the dfs number of each node to 0;
  num = 0;
  for each node
    if (node's dfs number == 0) { /* unvisited node */
      visitRoot(node);
  }
```

Functions para detectar los puntos de articulación de un gafo

Otro modo de hacer los recorridos: algoritmos de Tarjan

Los algoritmos de Tarjan son conocidos porque en base a ellos se pueden desarrollar resoluciones a otros problemas importantes, como por ejemplo, la determinación de componentes conexas de un grafo.

Recorrido de Tarjan en profundidad (pseudocódigo):

Este algoritmo considera que los vértices están numerados de forma ascendente. Usa un vector num con tantas posiciones como vértices haya para almacenar el número de orden de cada vértice en el recorrido que se realiza. Además el conjunto *edges* almacena las aristas utilizadas en el recorrido.

```
//Recorrido en profundidad de Tarjan (recursivo)

DepthFirstSearch
{
  para cada vértice v
   {
 num(v)=0;
}
edges= {};
i=1;
mientras exista vértice v / num (v) ==0
 hacer
     {DFS(v);}
}
```

```
DFS (v)
{
num(v)=i+1;
para cada vértice u adyacente a v hacer
    {
     Si ( num(u)==0)
          {
            Añadir arista (uv) a edges;
            DFS (u);
          }
    }
}
```

Recorrido de Tarjan en anchura o amplitud:

```
//Recorrido en anchura o amplitud de Tarjan

BreadthFirstSearch()
{
para  cada vértice u  hacer
   { num(u)=0;}
edges={};
i=1;
mientras exista vértice v / num(v)==0 hacer
 {
  num(v)=i++;
acolar (v);
mientras cola no vacía hacer
{
V=desacolar ();
para cada vértice u adyacente a v hacer
{
si  (num (u) ==0)
{
num(u)=i++;
acolar (u);
añadir arista (vu) a edges;
}
}
}
}
```

Detección de ciclos usando el recorrido en profundidad:

Se puede hacer un recorrido en profundidad de Tarjan usando el siguiente algoritmo en lugar del DFS

```
//DFS de Tarjan para la detección de ciclos

digraphCycleDetectionDFS(v)
{
num (v)=i++;
para cada vértice u adyacente a v hacer
  {
 Si (num(u)==0)
   {
    pred(u)=v;
   digraphCyleDetectionDFS(u);
   }
else if (num(u) != ∞)
{
```

```
  pred(u)= v;
  Ciclo detectado!!
}
}
num(v)=∞;

}
```

Recorridos topológicos:

Estos recorridos solo se aplican a grafos acíclicos, y permiten linealizar un grafo, es decir, listar la secuencia de los vértices del grafo de tal modo que se respete la precedencia de los mismos.

Un grafo dirigido acíclico se puede recorrer en profundidad o en anchura.

Recorrido topológico en profundidad

El pseudocódigo del algoritmo de recorrido topológico en profundidad se puede enunciar así:

(El algoritmo devuelve una lista en la cual quedan insertados los nodos según fueron visitados en el recorrido)

```
// Recorrido topológico en profundidad

TopolDFS
{
Para cada vértice del grafo, marcarlo como no visitado
Para cada vértice v del grafo
   Si v no fue visitado entonces ejecutar la función rec para v
}
Funcion rec aplicada a un vertice v (genera una lista)

{
Marcar v como visitado
Para cada vértice w adyacente a v hacer
     Si w no visitado entonces ejecutar función rec para w
Insertar v al frente en una lista //esta lista va incorporando los nodos visitados
}
```

Recorrido topológico en anchura

El algoritmo del recorrido topológico en anchura, puede expresarse informalmente así:

Se construye una tabla en la cual se anota el grado de entrada de cada vértice del grafo. (En esta tabla siempre hay al menos un vértice de grado 0, ya que , de otro modo, el grafo no seria acíclico).

Luego se ejecuta esto:

```
//Recorrido topológico en anchura

TopolBFS:
{
para  cada vértice hacer
 {
  si (grado_entrada (v) ==0)
      {acolar (v);}
}
mientras (cola no vacía) hacer
  {
    U= desacolar();
Colocar u al final en una lista;
Para cada vértice w adyacente a u hacer
   {
      Decrementar en 1 el grado de w;
      Si (grado_entrada(w)==0)
                  {acolar(w);}
    }
   }
}
```

<u>Algoritmo de Tarjan para la determinación de componentes conexas de un grafo:</u>

```
strongDFS(v)
  pred(v) = num(v) = i++;
  push(v);
  for all vertices u adjacent to v
    if num(u) is 0
      strongDFS(u);
      pred(v) = min(pred(v),pred(u));           // take a predecessor higher up in
    else if num(u) < num(v) and u is on stack   // tree; update if back edge found
      pred(v) = min(pred(v),num(u));            // to vertex u is in the same SCC;
  if pred(v) == num(v)                          // if the root of a SCC is found,
    w = pop();                                  // output this SCC, i.e.,
    while w ≠ v                                 // pop all vertices off the stack
      output w;                                 // until v is popped off;
      w = pop();
    output w;                                   // w == v;

stronglyConnectedComponentSearch()
  for all vertices v
    num(v) = 0;
  i = 1;
  while there is a vertex v such that num(v) == 0
    strongDFS(v);
```

Figure 8.17 contains an example of execution of Tarjan's algorithm. The digraph in Figure 8.17a is processed by a series of calls to `strongDFS()`, which assigns to vertices a through k the numbers shown in parentheses in Figure 8.17b. During this process, five SCCs are detected: $\{a,c,f\},\{b,d,e,g,h\},\{i\},\{j\}$, and $\{k\}$. Figure 8.17c contains the depth-first search trees created by this process. Note that two trees are created so that the number of trees does not have to correspond to the number of SCCs, as the number of trees did not correspond to the number of blocks in the case for undirected graphs. Figure 8.17d indicates, in italics, numbers assigned to $num(v)$ and all changes of parameter $pred(v)$ for all vertices v in the graph. It also shows the SCC's output during the processing of the graph.

Algoritmo de Tarjan para determinar componentes fuertemente conexas