# Diseño de algoritmos

*Jesús Bermúdez de Andrés. UPV-EHU*
*Guías para la solución de ejercicios: Recorridos de grafos*

*Curso 2008-09*

1. Considere la siguiente representación con listas de adyacencias del grafo no dirigido $G = (V, A)$:

| $V = [V(1), V(2), V(3), V(4), V(5), V(6), V(7), V(8)]$ | |
|---|---|
| $V(1) = [4, 2, 5]$ | $V(5) = [1, 2, 6]$ |
| $V(2) = [1, 4, 5, 6]$ | $V(6) = [2, 7, 5]$ |
| $V(3) = [8, 4, 7]$ | $V(7) = [3, 6, 8]$ |
| $V(4) = [2, 3, 8, 1]$ | $V(8) = [7, 3, 4]$ |

   *a*) Escriba la lista de vértices y la lista de aristas —en el orden que son visitados— correspondiente al recorrido en profundidad de $G$ comenzando en el vértice 1. Observe que al ser un grafo no dirigido, cada arista se visitará dos veces; ponga en la lista esa segunda visita cuando corresponda.

   *b*) Escriba la lista de vértices y la lista de aristas —en el orden que son visitados— correspondiente al recorrido en anchura de $G$ comenzando en el vértice 1. Observe que al ser un grafo no dirigido, cada arista se visitará dos veces; ponga en la lista esa segunda visita cuando corresponda.

**Solución:**

*a*) Recorrido en profundidad de $G = (V, A)$ partiendo del vértice 1.
Lista de vértices, en el orden en que son visitados: $1, 4, 2, 5, 6, 7, 3, 8$.
Lista de aristas, en el orden en que son visitadas:
$\{1,4\}, \{4,2\}, \{2,1\}, \{2,4\}, \{2,5\}, \{5,1\}, \{5,2\},$
$\{5,6\}, \{6,2\}, \{6,7\}, \{7,3\}, \{3,8\}, \{8,7\}, \{8,3\},$
$\{8,4\}, \{3,4\}, \{3,7\}, \{7,6\}, \{7,8\}, \{6,5\}, \{2,6\},$
$\{4,3\}, \{4,8\}, \{4,1\}, \{1,2\}, \{1,5\}$.

*b*) Recorrido en anchura de $G = (V, A)$ partiendo del vértice 1.
Lista de vértices, en el orden en que son visitados: $1, 4, 2, 5, 3, 8, 6, 7$.
Lista de aristas, en el orden en que son visitadas:
$\{1,4\}, \{1,2\}, \{1,5\}, \{4,2\}, \{4,3\}, \{4,8\}, \{4,1\},$
$\{2,1\}, \{2,4\}, \{2,5\}, \{2,6\}, \{5,1\}, \{5,2\}, \{5,6\},$
$\{3,8\}, \{3,4\}, \{3,7\}, \{8,7\}, \{8,3\}, \{8,4\}, \{6,2\},$
$\{6,7\}, \{6,5\}, \{7,3\}, \{7,6\}, \{7,8\}$.

2. Considere la siguiente representación con listas de adyacencias del grafo dirigido $G = (V, A)$:

| $V = [V(1), V(2), V(3), V(4), V(5), V(6), V(7), V(8)]$ | |
|---|---|
| $V(1) = [4]$ | $V(5) = [1, 2, 6]$ |
| $V(2) = [1]$ | $V(6) = [2, 7]$ |
| $V(3) = [8]$ | $V(7) = [3]$ |
| $V(4) = [2, 3, 8]$ | $V(8) = [7]$ |

   *a*) Escriba la lista de vértices y la lista de aristas —en orden que son visitados— correspondiente al recorrido en profundidad de $G$ comenzando en el vértice 1.

   *b*) Escriba la lista de vértices y la lista de aristas —en el orden que son visitados— correspondiente al recorrido en anchura de $G$ comenzando en el vértice 1.

   **Solución:**

   *a*) Recorrido en profundidad de $G = (V, A)$ partiendo del vértice 1.
   Lista de vértices, en el orden en que son visitados: $1, 4, 2, 3, 8, 7, 5, 6$.
   Lista de aristas, en el orden en que son visitadas:
   $(1, 4), (4, 2), (2, 1), (4, 3), (3, 8), (8, 7), (7, 3)$,
   $(4, 8), (5, 1), (5, 2), (5, 6), (6, 2), (6, 7)$.

   *b*) Recorrido en anchura de $G = (V, A)$ partiendo del vértice 1.
   Lista de vértices, en el orden en que son visitados: $1, 4, 2, 3, 8, 7, 5, 6$.
   Lista de aristas, en el orden en que son visitadas:
   $(1, 4), (4, 2), (4, 3), (4, 8), (2, 1), (3, 8), (8, 7)$,
   $(7, 3), (5, 1), (5, 2), (5, 6), (6, 2), (6, 7)$.

3. Escriba y analice un algoritmo que calcule los vectores *indegree*$(1 \ldots n)$ y *outdegree*$(1 \ldots n)$ de un grafo dirigido utilizando un recorrido en profundidad del grafo.

   Para cada vértice $v$ del grafo: *indegree*$(v)$= número de aristas que llegan a $v$, y *outdegree*$(v)$= número de aristas que salen de $v$.

   **Solución:**

```
proc DEGREE_EN_PROFUNDIDAD (G = (N, A))
  for cada nodo v ∈ N loop
    marca(v) ← false
    indegree(v)← 0
    outdegree(v) ← 0
  end for
  for cada nodo v ∈ N loop
    if not marca(v) then DEGREE_MARCA_PROF(v)
```

```
proc DEGREE_MARCA_PROF (v)
  marca(v) ← true
  for cada w adyacente a v loop
    outdegree(v) ← outdegree(v) + 1
    indegree(w) ← indegree(w) + 1
    if not marca(w) then DEGREE_MARCA_PROF (w)
```

Es evidente que es una adaptación simple del algoritmo de recorrido en profundidad, al que se han añadido dos operaciones de orden constante en el bucle de inicialización y otras dos operaciones de orden constante en el bucle de adyacentes al nodo $v$. Por tanto el algoritmo es del mismo orden que el recorrido en profundidad $\Theta(n + a)$.

4. En una sociedad recreativa, con $n$ personas asociadas, reina la camaradería. Entre otras cosas, se fían dinero unas a otras. Escriba y analice un algoritmo que —conocido el estado actual de los débitos— determine si entre las personas asociadas hay alguna que tenga el perfíl de *suma deudora*; es decir, una persona que deba dinero a todas las de la sociedad y ninguna le deba dinero a ella.

   **Solución:**
   Podemos representar el enunciado como un problema de tratamiento de grafos dirigidos. Cada persona se representa mediante un nodo, y decimos que hay una arista del nodo $i$ al nodo $j$ si $i$ debe dinero a $j$. Obsérvese que podemos tener las dos aristas $(i, j)$ y $(j, i)$.

   Es fácil deducir que si hay alguna persona *suma deudora*, sólo hay una.

   Planteado así el problema, lo que tenemos que calcular es si existe en el grafo algún nodo al que no llegan aristas y del que salen $n - 1$ aristas (suponiendo que $n$ es el número de nodos del grafo).

   Utilizando una matriz de adyacencia $G$ para representar el grafo ($G[i, j] = 0$ representa que no hay arista $(i, j)$, y $G[i, j] = 1$ representa que sí la hay), el algoritmo debe comprobar si hay algún nodo $k$ tal que $G[i, k] = 0 \forall i \neq k$ y $G[k, j] = 1 \forall j \neq k$. Una implementación simple para esta comprobación lleva a un algoritmo de $O(n^2)$.

   Utilizando una representación con listas de adyacencias, el algoritmo debe encontrar un nodo con una lista de $n - 1$ adyacentes y comprobar que ese nodo no aparece como adyacente de ningún otro nodo, para determinar la existencia de una persona *suma deudora*; si no se satisface esa condición, no existe persona *suma deudora*. El algoritmo será de $O(n+a)$, siendo $n$ el número de nodos y $a$ el número de aristas.

5. Considere una red internacional de $n$ ordenadores. Cada ordenador $X$ puede comunicarse con cada uno de sus vecinos $Y$ al precio de 1 doblón por comunicación. A través de las conexiones de $Y$ y subsiguientes, $X$ puede comunicarse con el resto de ordenadores de la red pagando a cada ordenador que utilice para la conexión el doblón correspondiente. Escriba un algoritmo que calcule

la tabla *precio*(1..$n$) tal que *precio*($i$) sea el número mínimo de doblones que le cuesta al ordenador, numerado con el 1, establecer conexión con el ordenador numerado con el $i$.

**Solución:**
Consideramos que el grafo que representa la red de ordenadores es conexo. Basta un recorrido en anchura del grafo, partiendo del nodo 1. Es decir, invocar PRECIO_DESDE (1).

```
proc PRECIO_DESDE (v)
  C ← Cola_vacía()
  marca(v) ← true
  precio(v) ← 0
  C.añadir(v)
  while not C.vacía() loop
    u ← C.retirar_primero()
    for cada nodo w adyacente a u loop
      if not marca(w) then
        precio(w) ← precio(u) + 1
        marca(w) ← true
        C.añadir(w)
```

Es fácil comprobar que el tiempo de este algoritmo es el mismo que el de un recorrido en anchura; es decir, $\Theta(n + a)$, siendo $n$ el número de ordenadores y $a$ el número de aristas (comunicaciones) del grafo.

6. Diseñe y analice un algoritmo que, dado un árbol $A$, determine la profundidad $P$ con el máximo número de nodos de $A$. Si hubiera varias profundidades con ese mayor número de nodos, determínese la mayor profundidad.

**Solución:**
Diseñamos un algoritmo que realiza un recorrido en anchura del árbol, partiendo de la raíz. Los elementos que ponemos en la cola son parejas formadas por el identificador de un nodo y el valor de la profundidad en la que se encuentra ese nodo.

```
func PROF_MAX_NODOS(A: árbol) return natural
  actual ← 0    {número de nodos contabilizados}
  última_prof ← 0   {en profundidad última_prof}

  C ← Cola_vacía()
  C.añadir((A.raíz(), 0))

  max ← 1         {número de nodos de A en}
  prof_max ← 0   {la profundidad prof_max}

  while not C.vacía() loop
    (x, p) ← C.retirar_primero()
```

```
    if p = última_prof then
      actual ← actual + 1 {x es un nodo más de la profundidad en curso}
    else {hemos terminado los nodos de una profundidad}
      if actual ≥ max then
        max ← actual
        prof_max ← última_prof
      end if
      actual ← 1          {x es el primer nodo de la}
      última_prof ← p    {profundidad última_prof}
    end if
    for cada hijo w de x loop
      C.añadir((w, p + 1))
    end for
  end loop
  return prof_max
```

Este es un recorrido en anchura en el que se hacen un número constante de operaciones elementales por cada nodo. Por lo tanto, el algoritmo es $\Theta(n)$ ya que, al tratarse de un árbol, el número de aristas es $n - 1$.

7. Escriba y analice un algoritmo que determine si un grafo dirigido $G = (N, A)$, representado por su lista de adyacencias, es o no un árbol. La existencia de una arista con origen en el nodo $a$ y destino en un nodo $b$ debe interpretarse en el sentido de *a es padre de b* en el hipotético árbol.

   **Solución:**
   Obsérvese que no basta con calcular el *indegree* y *outdegree* de cada nodo. Compruébelo con un grafo que tenga la siguiente colección de aristas:

   $A = \{(u, v), (v, u), (w, x), (w, y)\}$.

   Puede intentarse una solución que busque, primero, la existencia de un solo nodo con *indegree* 0 (si esto fracasa, el grafo no es un árbol) y, después, haciendo un recorrido desde ese nodo, comprobar que se visitan todos los nodos y que ninguno es visitado más de una vez. Pero parece que esa solución haría varios recorridos del grafo y no sería tan eficiente como la que proponemos a continuación, que hace un solo recorrido del grafo y no necesita localizar la eventual raíz del hipotético árbol.

   Para resolver el problema podemos efectuar un recorrido en profundidad del grafo, marcando a cada nodo con un número distinguido, de manera que si dos nodos tienen el mismo número de marca es porque pertenecen al mismo subárbol, visitado por el recorrido en profundidad desde un mismo nodo inicial que llevará, además, una marca de raíz. Esta marca de raíz se perderá si en un eventual recorrido posterior, tal nodo resulta ser hijo de algún otro nodo. Obsérvese que esto llevará a que nodos que están en un mismo subárbol puedan tener números de marca distintos, pero esto no causará ningún inconveniente (en particular, no contradice lo que hemos dicho al principio de este párrafo).

Si en algún momento del recorrido nos topamos con un nodo que está marcado y no es raíz, o bien siéndolo tiene la misma marca que estemos usando en ese momento: el grafo no es un árbol. Si al final de todo esto, sólo queda un nodo marcado como raíz: el grafo es un árbol.

Representaremos que un nodo $v$ no ha sido visitado mediante marcaNum$(v) = 0$, y los números distinguidos a los que nos hemos referido anteriormente, se representarán con marcaNum$(v) \neq 0$. Las tablas marcaNum$(1..n)$ y es_raíz$(1..n)$, así como la variable num_raíces, son globales para el procedimiento MARCA_PROF.

```
func RECONOCE_ÁRBOL (G = (N, A)) return boolean
  fracaso ← false
  for cada nodo v ∈ N loop
    marcaNum(v) ← 0
    es_raíz(v) ← false
  end for
  etiqueta ← 0
  num_raíces ← 0
  for cada nodo v ∈ N loop
    if marcaNum(v) = 0 then
      etiqueta ← etiqueta + 1
      es_raíz(v) ← true
      num_raíces ← num_raíces + 1
      MARCA_PROF(v, etiqueta, fracaso)
      if fracaso then return false
  end for
  return (¬fracaso ∧ num_raíces = 1)

proc MARCA_PROF (u, num, f)
  marcaNum(u) ← num
  for cada w adyacente a u loop
    if marcaNum(w) = 0 then
      MARCA_PROF (w, num, f)
      if f then return
    else
      if es_raíz(w) ∧ marcaNum(w) ≠ num then
        es_raíz(w) ← false
        num_raíces ← num_raíces − 1
      else
        f ← true
        return
```

Este algoritmo es $O(n)$, siendo $n$ el número de nodos del grafo. Obsérvese que vamos recorriendo el eventual árbol — una arista por cada nodo — hasta determinar que es un árbol, o bien terminar antes porque detectamos fracaso.

8. Entre los asistentes a una *Convención* son bien conocidas las relaciones de antipatía mutua entre muchas personas. Los organizadores de la cena final, en su deseo de mantener el mejor ambiente posible, han contratado un restaurante con dos comedores en los que repartir a los asistentes.

   Escriba y analice un algoritmo que determine si es posible separar a los asistentes de modo que en cada comedor no haya ninguna persona que sienta antipatía por otra del mismo comedor.

   **Solución:**
   El problema puede trasladarse a un problema de grafos. Las personas son los nodos y la relación de antipatía entre dos personas se representa mediante una arista que conecta los nodos que representan a esas dos personas. El grafo es no dirigido puesto que la relación de antipatía es mutua.

   El problema puede resolverse con un recorrido en anchura del grafo. Añadiremos una marca más por cada nodo, para indicar el comedor al que ha sido asignada la persona representada por ese nodo. Para cada nodo $v$, $comedor(v) = true$ significa que $v$ está asignado al comedor número 1 y el valor $false$ significa que está asignado al comedor número 2.

   Obsérvese que el grafo no es necesariamente conexo. El procedimiento MARCA_COMEDOR($v$, opción, *resultado*) comienza con el valor de *resultado* igual a *true*, y terminará sin modificar ese valor si es posible ubicar a las personas de la componente conexa de $v$ en dos comedores, respetando las restricciones del problema; en caso contrario, terminará con el valor de *resultado* igual a $false$.

```
func CONVENCIÓN (G = (N, A)) return boolean
  es_posible ← true {valor inicial}
  for cada v ∈ N loop marca(v) ← false end for
  for cada v ∈ N loop
    if not marca(v) then MARCA_COMEDOR(v, true, es_posible) end if
    if not es_posible then return false end if
  end for
  return true


proc MARCA_COMEDOR(v, opción, resultado)
  C ← Cola_vacía()
  marca(v) ← true
  comedor(v) ← opción
  C.añadir(v)
  while not C.vacía() loop
    u ← C.retirar_primero()
    for cada w adyacente de u loop
      if not marca(w) then
        marca(w) ← true
        comedor(w) ← not comedor(u)   {llevar a w a otro comedor}
```

```
      C.añadir(w)
    else  {w está marcado, y por tanto asignado a un comedor.}
          {Si es el mismo que el de u entonces no es posible repartirlos}
      if comedor(u) = comedor(w) then
        resultado ← false
        return
```

Este algoritmo es de $O(n + a)$, siendo $n$ el número de nodos y $a$ el número de aristas del grafo, puesto que se trata de un recorrido en anchura del grafo que sólo añade una cantidad constante de operaciones elementales al procesamiento de cada nodo.

9. Sea $G = (N, A)$ un grafo no dirigido con pesos asociados a los nodos (ojo, no a las aristas). Escriba un algoritmo que calcule la longitud del camino más largo que puede recorrerse en ese grafo, pasando siempre de un nodo a otro con mayor peso asociado, y cual es el nodo origen de ese camino.

   **Solución:**
   Lo podemos resolver con un recorrido en profundidad del grafo. Representaremos mediante la tabla $P(1..n)$ los pesos asociados a cada nodo del grafo $G = (N, A)$.

   Utilizaremos una tabla camino$(1..n)$ tal que camino$(v)$ sea el número natural que indica la longitud del camino más largo que comienza en $v$ y recorre nodos con pesos estrictamente crecientes.

```
func Camino_Largo (G = (N, A)) return nodo × natural
  for cada nodo v ∈ N loop marca(v) ← false end for
  for cada nodo v ∈ N loop
    if not marca(v) then Marca_Prof(v, camino(1..n))
  end for
  origen ← 1   {un nodo inicial}
  for v ← 2...n loop
    if camino(origen) < camino(v) then origen ← v
  end for
  return (origen, camino(origen))


proc Marca_Prof (u, camino(1..n))
  marca(u) ← true
  camino(u) ← 0
  for cada w adyacente de u loop
    if P(w) > P(u) then
      if not marca(w) then   {camino(w) no está calculado todavía}
        Marca_Prof (w, camino(1..n))   {ahora, camino(w) está calculado}
      end if
      camino(u) ← max{camino(u), 1 + camino(w)}
```

Este algoritmo sigue directamente el esquema de un recorrido en profundidad de un grafo de $n$ nodos y $a$ aristas, con una cantidad constante de operaciones elementales añadidas por cada nodo, así que es de $\Theta(n + a)$.

10. Es conocido que todos los caminos llevan a Roma. Usando una buena guía y un mapa de carreteras, hemos creado un grafo dirigido y sin ciclos (*dag*) que describe multitud de buenos itinerarios para viajar desde nuestra ciudad a Roma.

    Escriba y analice un algoritmo que calcule cuántos itinerarios posibles podemos seguir desde nuestra ciudad a Roma, según el grafo creado.

    **Solución:**

    Basta un recorrido en profundidad del grafo, que calcule el número de caminos desde cada nodo al nodo que representa a Roma, anotándolo justamente cuando termina el recorrido en profundidad desde ese nodo. Para cada nodo $v$, vamos a disponer de caminos($v$) para representar ese valor.

    Puede hacerse así: Supongamos que numeramos los nodos de manera que asignamos el número 1 al nodo que representa a Roma.

```
proc Caminos ( G = (N, A), caminos(1..n) )
  for cada i ∈ N loop
    marca(i) ← false
    caminos(i) ← 0
  end for
  marca(1) ← true   {Roma se marca como visitada}
  caminos(1) ← 1   {Desde Roma a Roma, el camino sin aristas}
  for i ← 2 . . . n loop
    if not marca(i) then Marca_Prof(i, caminos)


proc Marca_Prof(i, caminos(1..n))
  marca(i) ← true
  for cada w adyacente a i loop
    if marca(w) then   {conocemos caminos(w)}
      caminos(i) ← caminos(i) + caminos(w)
    else
      Marca_Prof(w, caminos(1..n)) {conocemos caminos(w)}
      caminos(i) ← caminos(i) + caminos(w)
  end for
```

    Este algoritmo sigue directamente el esquema de un recorrido en profundidad de un grafo de $n$ nodos y $a$ aristas, con una cantidad constante de operaciones elementales añadidas por cada nodo, así que es de $\Theta(n + a)$.