Recorridos de grafos: Teoría y aplicaciones

Jesús Bermúdez de Andrés
Departamento de Lenguajes y Sistemas Informáticos
Universidad del País Vasco/Euskal Herriko Unibertsitatea

3 de mayo de 2011

Índice

1. Exploración de grafos

Muchos problemas interesantes se pueden modelar como problemas de grafos. Asumimos que la estructura de *grafo* (tanto *dirigido* como *no dirigido*) ya es conocida. A continuación recordaremos las dos representaciones de grafos más típicas. La representación con *matriz de adyacencia* y con *lista de adyacencias*. Es importante comprender bien las ventajas e inconvenientes de cada una de ellas con respecto a la cantidad de memoria necesaria y los procedimientos básicos para la manipulación de la información que representan.

- La representación mediante *matriz de adyacencia* supone que disponemos de una numeración $1, \ldots, n$ de los nodos del grafo y consiste en una matriz cuadrada $A = (a_{ij})$, indexada por los nodos, tal que $a_{ij} = 1$ si la arista (i, j) está en el grafo y $a_{ij} = 0$ si no.

- La representación mediante *listas de adyacencias*, consiste en un vector de listas, indexado por los nodos numerados $1, \ldots, n$, tal que para cada nodo i tenemos acceso a una lista con todos sus nodos adyacentes, es decir, el nodo j está en la lista de i si y sólo si la arista (i, j) está en el grafo.

En esta presentación no entraremos en las operaciones de *añadir* o *retirar* nodos y aristas de un grafo, aunque obviamente son operaciones de interés para algunas aplicaciones y habrá que estudiarlas en su caso. Los algoritmos que vamos a estudiar son dos modos básicos de recorrer o explorar un grafo dado. Podemos ver la analogía con estructuras de datos ya conocidas como las *listas* y los *árboles*. La forma básica de explorar una lista es con un recorrido secuencial de sus elementos. Con respecto a los árboles, podemos citar los recorridos de sus nodos (y aristas) en *preorden*, *inorden* y *postorden*.

Los algoritmos que presentamos en este tema son el de *recorrido en profundidad* y el de *recorrido en anchura* de un grafo. El esquema de cada recorrido queda descrito en pseudocódigo, independientemente de que representemos el grafo con una *matriz de adyacencia* o con *listas de adyacencias*; la decisión sobre el uso de una u otra representación vendrá determinada por el problema concreto a solucionar.

Estos dos algoritmos hay que tomarlos como esquemas que únicamente imponen un orden de visita de cada nodo y cada arista del grafo, es decir no hacen nada salvo acceder a los nodos (y las aristas si se desea) en un determinado orden; las operaciones que haya que hacer con cada visita a un nodo dependen, naturalmente, del problema que se quiera resolver y habrá que añadirlas en las líneas de código correspondientes. Veremos algunas aplicaciones típicas de estos recorridos, como son el denominado problema

de *ordenación topológica* y la determinación de *grafo bipartito*. Aprovecharemos, también, para presentar la técnica de diseño de algoritmos de *vuelta atrás*, puesto que puede verse como un caso particular de un recorrido en profundidad.

1.1. El recorrido en profundidad

Sea $G = (N, A)$ un grafo en el que su conjunto de nodos es N y su conjunto de aristas es A. Según vayan siendo visitados los nodos del grafo G, lo anotaremos en un vector de valores booleanos que denominamos *marca* y que está indexado por los identificadores de los nodos, que consideramos que son los números $1, \ldots, n$.

En un grafo dirigido $G = (N, A)$, decimos que un nodo u es *adyacente de* otro v si la arista (v, u) aparece en A (es decir, si existe una arista —que en estos casos se dibuja como una flecha— que sale de v y llega a u).

En un grafo no dirigido $G = (N, A)$, si existe la arista $\{v, u\}$ (que también puede denominarse $\{u, v\}$, porque el orden aquí no importa) decimos que el nodo u es *adyacente de* v y viceversa porque la relación ahora es simétrica (a diferencia del caso de grafo dirigido anterior).

La forma natural de escribir el esquema de un recorrido en profundidad es utilizando la recursión. Cada invocación recursiva se encarga de marcar como visitado el nodo al que se acaba de acceder y después lanzar las invocaciones recursivas pertinentes sobre sus adyacentes que no hayan sido visitados todavía; a este procedimiento lo hemos denominado MarcaProf. Cuando finalice la primera invocación de MarcaProf (v) se habrán marcado todos los nodos visitables desde el nodo v. Si todavía quedasen nodos por marcar, estos se procesarán cuando les llegue el turno en el bucle **for** cada $v \in N$ **loop** del procedimiento principal RecorridoEnProfundidad $(G = (N, A))$. Inicialmente se asigna el valor `falso` a todos los componentes del vector `marca`.

```
proc RecorridoEnProfundidad (G = (N, A))
  for cada v ∈ N loop marca(v) ← falso end loop
  for cada v ∈ N loop
    if ¬ marca(v) then MarcaProf(v)

proc MarcaProf (v)
  marca(v)← verdadero
  for cada w ∈ N adyacente de v loop
    if ¬ marca(w) then MarcaProf(w)
```


El hecho de que las llamadas recursivas generadas por un recorrido en profundidad de un grafo pueden representarse como nodos de un árbol que se recorre desde la raíz a las hojas en el mismo sentido que lo hace un recorrido en preorden, puede ayudar a comprender el calificativo de *profundidad* que se da a esta modalidad de recorrido. Para hacerse una idea más concreta de cómo se va visitando cada elemento de un grafo (nodos y aristas) con un recorrido en profundidad, es un buen ejercicio dibujar una copia del grafo siguiendo el orden de visita que va imponiendo el algoritmo a los nodos y las aristas. Si utilizamos una línea punteada (en vez de una línea continua) para dibujar cada arista que conecte un nodo con otro previamente visitado (es decir, que su valor de marca sea verdadero), la copia del grafo finalmente dibujada podrá mirarse también como una colección de árboles de recorrido en profundidad, si consideramos sólo las líneas continuas.

1.2. Análisis

Sea n el número de nodos y a el número de aristas del grafo. Puesto que cada nodo será visitado exactamente una vez, habrá n llamadas a MarcaProf. Cuando se visita un nodo hay que comprobar si están marcados sus nodos adyacentes, esto es tanto como visitar todas las aristas del grafo:

- si el grafo es no dirigido y lo representamos con *listas de adyacencias* el número de operaciones es proporcional a $2a$ porque cada arista $\{u, v\}$ se visita dos veces, una vez desde el nodo u y otra desde el nodo v. Si el grafo es dirigido ese número será a. En cualquier caso (grafo dirigido o no) el número de operaciones realizadas será de $\Theta(n + a)$.
- si el grafo (dirigido o no) lo representamos con *matriz de adyacencia* entonces se necesita un número de operaciones de $\Theta(n)$ para reconocer los adyacentes de cada nodo, independientemente del número de aristas que tenga el grafo. Entonces al número n de llamadas recursivas hay que sumar $\Theta(n^2)$ operaciones para la visita de las aristas y en consecuencia la función de coste temporal de esta implementación sería de $\Theta(n^2)$.

En todos los casos necesitamos un espacio extra de $\Theta(n)$ para almacenar el vector de marcas más el espacio que habrá que reservar para almacenar los registros de la pila de recursión, que en ningún caso superará una altura proporcional a n ya que ese es el número máximo de llamadas recursivas.

A continuación presentamos un esquema general para el recorrido en profundidad de un grafo, indicando los lugares donde convendría realizar una determinada acción, según el tipo de proceso que se necesite para resolver el problema en cuestión.

Esquema general para MarcaProf

```
proc Marca_Prof_Gen(v)
  [Procesar v en primera visita]
  [Como en preorden]
  marca(v) ← cierto
  for cada w ∈ N adyacente de v loop
    [Procesar arista (v, w)]
    if ¬ marca(w) then
      [Procesar arista (v, w) del árbol]
      Marca_Prof_Gen(w)
      [Procesar v al regreso de procesar w]
      [Como en inorden]
    else
      [Procesar arista (v, w). NO es del árbol]
  end for
  [Procesar v al abandonarlo]
  [Como en postorden]
```

1.3. El recorrido en anchura

El recorrido en anchura visita los nodos en orden "creciente de distancia" (medido en número de aristas que le separan) al punto de partida. Desde el nodo de partida se visita a todos sus adyacentes antes de "irse más lejos". Después, sucesivamente, y respetando el mismo orden en el que se fueron visitando, se toma cada uno de ellos como nuevo nodo de partida y se repite la operación. Para gestionar ese tratamiento sucesivo en el orden pertinente utilizamos una *cola* de nodos a la que se van añadiendo nodos justamente en el orden en que son visitados.

Como en el apartado anterior, usamos un vector de marcas para representar los nodos que ya han sido visitados.

```
proc RecorridoEnAnchura (G = (N, A))
  for cada v ∈ N loop marca(v) ← falso end loop
  for cada v ∈ N loop
    if ¬ marca(v) then MarcaAncho(v)

proc MarcaAncho (v)
  C ← new Cola
  marca(v) ← verdadero
  C.insert(v)
  while ¬ C.is_empty() loop
    u ← C.remove_first()
    for cada w ∈ N adyacente de u loop
      if ¬ marca(w) then
        marca(w) ← verdadero
        C.insert(w)
  end while
```


1.4. Análisis

Para cada nodo v del grafo se realizan exactamente una vez estas cuatro operaciones: marca(v) ← falso, marca(v) ← verdadero, C.insert(v) y $u \leftarrow C$.remove_first(), cada una de las cuales puede realizarse en tiempo constante $\Theta(1)$. Además, para cada nodo hay que visitar a todos sus adyacentes; y siguiendo el mismo discurso del apartado anterior concluimos que si el grafo se representa con listas de adyacencias el orden temporal resultante es de $\Theta(n + a)$ y si el grafo se representa con una matriz de adyacencia el orden será de $\Theta(n^2)$.

El espacio extra utilizado es de $\Theta(n)$, lo necesario para el vector de marcas más la máxima longitud de la cola.

Ya hemos dicho antes que estos recorridos, por sí solos, no hacen nada interesante; en general habrá que "rellenarlos" con las instrucciones pertinentes para resolver problemas de interés. Por ejemplo, el que planteamos a continuación.

2. Ordenación topológica

Imaginemos que tenemos una gran colección de tareas que realizar y que sólo podemos llevarlas a cabo una detrás de otra (no podemos realizar varias tareas a la vez). Además tenemos establecida entre algunas de ellas una relación de precedencia. Es decir, simplemente sabemos que una tarea u hay que hacerla antes que otra tarea v, para unas cuantas tareas u y v. Cuando no hay relación de precedencia entre dos tareas, cualquiera de ellas puede hacerse antes que la otra. Naturalmente, la relación de precedencia es transitiva y, para que se pueda respetar el orden de realización, no puede haber "ciclos" en esas relaciones de precedencia.

¿Cómo determinamos un orden entre las tareas, para que se puedan realizar todas, respetando el orden de precedencia?

Este problema puede plantearse como un problema de grafos en el que las tareas son nodos y la relación de que la tarea u precede a la tarea v se

representa con una arista dirigida del nodo u al nodo v. Entonces el problema tiene una solución eficiente y elegante mediante un recorrido en profundidad de ese grafo. Ese problema es conocido como el de la *ordenación topológica* y lo resovemos a continuación.

Una *ordenación topológica* de un grafo dirigido *acíclico* $G = (N, A)$ es una lista de los nodos del grafo tal que si la arista (u, v) aparece en A entonces el nodo u aparece antes que v en la lista. Obsérvese que, en general, de un grafo pueden obtenerse distintas listas que satisfacen la propiedad de ordenación topológica.

Este problema puede resolverse como una aplicación directa del esquema de recorrido en profundidad. Obsérvese que en un grafo dirigido acíclico, cuando finalice la llamada MarcaProf (v) se habrán marcado como visitados todos los nodos a los que se puede ir desde el nodo v. Entonces todos esos nodos deberán aparecer después de v en cualquier ordenación topológica del grafo, pero aprovechando aquí el anidamiento de llamadas recursivas podemos plantearlo de este otro modo: admitiendo que cuando finalice la llamada MarcaProf(v), se habrán puesto en una lista todos los nodos a los que se puede ir desde el nodo v, bastará con que pongamos v al frente de esa lista para garantizar que v satisface los requisitos de la ordenación topológica en relación a todos sus nodos accesibles.

Así pues, la clave está en construir la lista desde el final hacia el principio.

```
func Ordenación_Topológica (G = (N, A)) return Lista_de_nodos
  L ← new Lista
  for cada v ∈ N loop marca(v) ← falso end loop
  for cada v ∈ N loop
    if ¬ marca(v) then OrdenTopo(v, L)
  end for
  return L

proc OrdenTopo (v, L)
  marca(v) ← verdadero
  for cada w ∈ N adyacente de v loop
    if ¬ marca(w) then OrdenTopo(w, L)
  end for
  L.insert_first(v)
```

Las operaciones que se añaden al recorrido en profundidad para diseñar la ordenación topológica son de orden $O(1)$ y sólo se inserta en la lista una vez por cada nodo, por lo tanto la función de coste temporal de esta ordenación topológica es de $\Theta(n + a)$, siendo n el número de nodos y a el número de aristas.

3. Grafo bipartito

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.

Vamos a diseñar 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.

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.

4. Número de caminos

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.

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

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
```

Si nuestra ciudad tiene asignado el número k, la solución la encontraremos en caminos(k) después de ejecutar Caminos ($G = (N, A)$, caminos(1..n)).

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 $\Theta(n + a)$.

5. Diseño de algoritmos de vuelta atrás

Hay problemas para los que no se conoce mejor solución que la que consiste, básicamente, en una búsqueda sistemática en un espacio de propuestas posibles de solución. Aquí presentamos la *vuelta atrás* como una técnica que consiste en construir la solución de un problema mediante extensiones de *ensayos* parciales de solución. En general, se parte del ensayo neutro [ ] y en un estado intermedio del proceso, tras haber construido el ensayo parcial $[x_1, \ldots, x_i]$, se tantea sistemáticamente con cada posibilidad x_{i+1} de extender el ensayo:

- si $[x_1, \ldots, x_i, x_{i+1}]$ es un ensayo *aceptable*, se extiende $[x_1, \ldots, x_i]$ a $[x_1, \ldots, x_i, x_{i+1}]$ y se continúa con este último ensayo parcial.
- si ninguna de las posibilidades para x_{i+1} hacen que $[x_1, \ldots, x_i, x_{i+1}]$ sea *aceptable* entonces significa que el ensayo $[x_1, \ldots, x_i]$ no es extensible a solución, por consiguiente retiramos la última opción x_i añadida a ese ensayo y continuamos con el ensayo parcial $[x_1, \ldots, x_{i-1}]$ tanteando otra posible opción.

El criterio que decide si un ensayo es aceptable es propio de cada algoritmo de vuelta atrás y es esencial en el rendimiento del mismo.

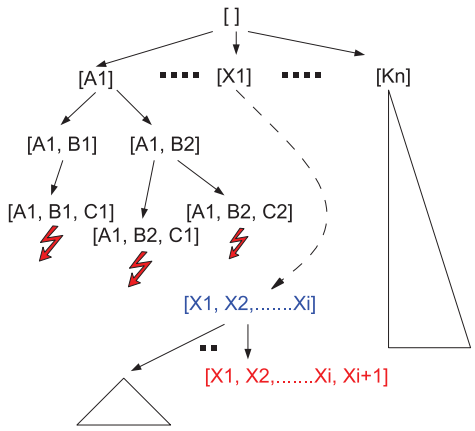

Figura 1: Árbol de ensayos

Esta estrategia equivale al recorrido en profundidad de un árbol implícito (ya que ese árbol no está representado por ninguna estructura de datos) comenzando desde la raíz, tal que cada nodo se corresponde con un ensayo parcial y cada arista que parte de un nodo se corresponde con una opción de tanteo. Cuando un ensayo no es aceptable equivale a podar la posible rama del árbol que pudiera crecer desde ese nodo. Cuanto más "inteligente" sea la poda más eficiente será el algoritmo. En consecuencia, las hojas de ese árbol implícito se corresponden con ensayos no extensibles a solución o bien con ensayos que son solución.

Con frecuencia se utiliza la técnica de vuelta atrás para abordar problemas que tienen varias soluciones, y con esta técnica se pueden calcular sistemáticamente todas ellas. A continuación presentamos un esquema general de un algoritmo de vuelta atrás para calcular todas las soluciones de un problema, posteriormente introducimos una pequeña modificación para que el algoritmo termine en cuanto encuentre la primera solución.

Esquema de vuelta atrás que calcula todas las soluciones

```
proc Ensayar (E)
  if Solución(E) then
    "gestionar el E solución"
  {else }
    for cada posibilidad p de extender E loop
      if Es_aceptable(E + p) then
        Ensayar (E + p)
```


Debe retirarse la cláusula {**else** } en caso de que haya soluciones que sean prefijos de otras soluciones.

Esquema de vuelta atrás que calcula la primera solución

```
proc Ensayar (E, éxito)
  if Solución(E) then
    "gestionar el E solución"
    éxito ← true
  else
    for cada posibilidad p de extender E loop
      if Es_aceptable(E + p) then
        Ensayar (E + p, éxito)
        if éxito then return
```

Análisis de un algoritmo de vuelta atrás

Un modo de aproximarse al análisis de la eficiencia de los algoritmos de vuelta atrás es analizar el árbol implícito de ensayos explorado por el algoritmo:

1. contar el número de ensayos explorados.

2. analizar el coste de calcular las posibilidades de extender cada ensayo y el coste de generar cada nuevo ensayo.

3. analizar el coste de comprobación de la aceptación de un ensayo.

Hay que observar que el resultado es una cota superior.

El rendimiento real del algoritmo puede ser mucho mejor, ya que la función de aceptación de ensayos puede podar una cantidad considerable de ramas del árbol implícito. De hecho, la utilidad de los algoritmos de vuelta atrás depende principalmente de la capacidad de aplicación de funciones de aceptación que poden convenientemente ramas del árbol de ensayos. Puede realizarse una evaluación usando un Método de Montecarlo.

6. El coloreado de mapas

El problema de colorear mapas políticos de manera que dos países que tengan frontera común estén pintados de distinto color provocó durante muchos años estudios matemáticos que intentaban justificar que cualquier mapa político "razonable" podía pintarse usando muy pocos colores cumpliendo esas restricciones de coloreado de países fronterizos. Finalmente se llegó a demostrar que son suficientes cuatro colores.

El problema que aquí nos planteamos es escribir un algoritmo que calcule todas las formas posibles de colorear los países de un determinado mapa usando sólo cuatro colores de manera que ningún par de países fronterizos esté pintado del mismo color.

En primer lugar conviene definir una representación conveniente del mapa. Cuando hay que representar alguna relación entre elementos, en este caso concreto la relación de frontera entre dos países, es frecuente que la noción de grafo nos ayude. En este caso podemos representar el mapa mediante un grafo no dirigido en el que los nodos representan a los países y cada arista representa que sus extremos son países fronterizos. En particular, representaremos un mapa con n países mediante una matriz de adyacencia $F(1..n, 1..n)$ con valores booleanos.

Vamos a presentar un algoritmo de vuelta atrás en el que un ensayo de longitud k consiste en la lista de colores (tomados de los cuatro posibles) que se han asignado a los países numerados con $[1, \ldots, k]$. Los ensayos los representaremos con un vector *color(1..n)* con valores en el conjunto $\{1, 2, 3, 4\}$ que representa a los cuatro colores. En cada momento del proceso sólo son representativos los k primeros componentes del vector *color*; cuando k llegue a ser n y se satisfaga la condición de aceptación tendremos una solución del problema. Una extensión de un ensayo de longitud k consiste en dar color al país número $k + 1$. Supongamos que tenemos un ensayo de longitud k que satisface las restricciones del problema, entonces una extensión de ese ensayo es aceptable si el color asignado al país número $k + 1$ es distinto del color asignado a sus adyacentes que estén numerados del 1 al k. La llamada inicial será CUATRO_COLORES(0, F, *color*).

```
proc CUATRO_COLORES(k, F(1..n, 1..n), color(1..n))
{k: número de países coloreados}
  if k = n then
    IMPRIMIR(color(1..n))
  else
    for C in 1..,4 loop
    color(k + 1) ← C
    if ES_ACEPTABLE(F, color, k + 1) then
      CUATRO_COLORES(k + 1, F, color)

func ES_ACEPTABLE(F, color, último) return boolean
{último: índice del último país coloreado}
  acept ← true
  país ← 1
  while país < último ∧ acept loop
    if F(país, último) ∧ color(país)=color(último) then
      acept ← false
    end if
    país ← país + 1
  end loop
  return acept
```


Análisis

Cada ensayo tiene cuatro posibles extensiones y la longitud de los ensayos solución es n, por lo tanto el número de nodos del árbol de ensayos está acotado superiormente por

$$1 + 4 + 4^2 + \cdots + 4^n = (4^{n+1} - 1)/3 \in O(4^n)$$

Obsérvese que, además, la comprobación de aceptación de una extensión no se realiza en tiempo constante ya que, en el peor caso, hay que visitar los k componentes de *color* que componen el ensayo en curso. O sea que la función ES_ACEPTABLE tiene un coste temporal en O(n). En consecuencia, en el caso peor, la función de coste temporal de CUATRO_COLORES(0, F, *color*) cuando el mapa tiene n países es de $O(n4^n)$.

Las secciones siguientes contienen sólo una mínima explicación para poder leer el pseudo-código de los algoritmos. Cada una de ellas debe completarse con anotaciones y explicaciones pertinentes que se darán en clase.

7. Número mínimo de monedas

Supongamos que tenemos un determinado número n de monedas, de valores $v_1, \ldots, v_n$ respectivamente. Queremos determinar cual es el número mínimo de monedas que necesitamos para sumar una determinada cantidad C.

En este caso, decidimos que un ensayo sea una secuencia de los valores de esas monedas, que denominaremos *SEC*. Para completar la representación de un ensayo tendremos: una variable *suma* que representa la suma de los

valores de *SEC*, una variable *long* que representa la longitud de *SEC* y una variable k que representa el subíndice de la moneda que hay que considerar a continuación.

La parte del esquema de vuelta atrás dedicada a *"gestionar el ensayo solución"* sirve para memorizar la mejor solución encontrada hasta el momento. La solución óptima será la que quede registrada en la secuencia OPT, de longitud $longOPT$.

Para comprender bien la condición de poda del árbol de ensayos hay que tener en cuenta que se supone que tenemos ordenadas las monedas de manera que $v_1 \leq \ldots \leq v_n$. Para comprender el árbol de ensayos hay que considerar lo siguiente:

- Ensayo: Un secuencia de valores de monedas
- Las posibilidades de extender un ensayo $[v_i, \ldots, v_{k-1}]$ son $v_k \ldots v_n$
- El ensayo $[v_i, \ldots, v_{k-1}, v_k]$ es aceptable si su longitud es menor que $longOPT$ y su suma menor o igual que C
- Un ensayo es solución si su suma es igual a C

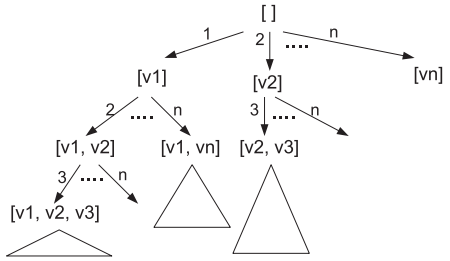

Figura 2: Árbol de ensayos para calcular el mínimo número de monedas

Entonces la solución puede ser la siguiente:

```
proc MinMonedas(SEC, k, long, suma, OPT, longOPT)
  if suma = C then
    if long < longOPT then
      OPT ← SEC
      longOPT ← long
  else
    while (k ≤ n) ∧ (1 + long < longOPT) ∧ (suma + v_k ≤ C) loop
      SEC[1 + long] ← v_k
      MinMonedas(SEC, k + 1, 1 + long, suma + v_k, OPT, longOPT)
      k ← k + 1
```


La llamada inicial debe ser MinMonedas(SEC, 1, 0, 0, OPT, $n+1$).

8. Suma de subconjunto

Disponemos de valores $V = \{V_1, \ldots, V_n\}$ enteros positivos y queremos saber todas las formas posibles de sumar D con subconjuntos de V.

- Un ensayo constará de un vector *conjunto* de ceros y unos, de manera que $conjunto[i] = 1$ representa que V_i pertenece al subconjunto seleccionado y $conjunto[i] = 0$ representa que no.
- Tendremos dos variables: *suma* que representa la suma de los valores del subconjunto seleccionado y *resto* que representa la suma de los valores que quedan todavía por considerar.
- La variable j representa el subíndice del valor de V que hay que considerar a continuación.
- Para podar más ramas es importante considerar los valores de V en orden creciente: Suponemos que $V_1 \leq \ldots \leq V_n$.
- Un ensayo es aceptable si $suma + resto \geq D$ (para que sea posible llegar a sumar D) y $suma \leq D$ (para que la suma del ensayo no exceda de la cantidad a sumar).

```
proc SumaSubconjunto(suma, resto, j, conjunto)
  if suma = D then
    Imprimir(conjunto[1..j − 1])
  else
    if suma + V_j ≤ D then
      conjunto[j] ← 1
      SumaSubconjunto(suma + V_j, resto − V_j, j + 1, conjunto)
      if suma + resto − V_j ≥ D then
        conjunto[j] ← 0
        SumaSubconjunto(suma, resto − V_j, j + 1, conjunto)
```

La llamada inicial debe ser SumaSubconjunto(0, $\sum_{i=1}^{n} V_i$, 1, $conjunto$).

9. El problema de la mochila (*0-1 Knapsack problem*)

Tenemos una mochila con capacidad para transportar un máximo de M unidades de peso y ponen a nuestra disposición n objetos con su peso y valor correspondientes. Queremos cargar la mochila con una colección de esos objetos de manera que no se supere la capacidad de la mochila y que contenga el máximo valor posible. Dicho de un modo más formal, dado M y dados n objetos con valores unitarios $(v_1, \ldots, v_n)$ y pesos $(p_1, \ldots, p_n)$ respectivamente queremos calcular un vector $(x_1, \ldots, x_n)$ de valores $x_i = 0, 1$ tal que

$$\text{se maximice} \quad \sum_{i=1}^{n} x_i v_i$$

$$\text{restringido a que} \quad \sum_{i=1}^{n} x_i p_i \leq M$$

Este es un problema de optimización; de todas las formas posibles de cargar la mochila, sin exceder su capacidad, deseamos elegir una que maximice su valor.

Hay varias opciones para la definición de los ensayos. Decidimos que los ensayos sean vectores de longitud $i \leq n$ con valores $0, 1$ que representan las opciones tomadas con respecto a los objetos numerados con $\{1, \ldots, i\}$, un valor $x_i = 0$ representa la opción de no poner el objeto i en la mochila y un valor $x_i = 1$ representa la opción de ponerlo. Una solución de este problema es un vector de longitud n con valores 0 o 1.

Podemos utilizar el esquema de vuelta atrás que nos proporciona todas las formas de cargar la mochila sin sobrepasar su capacidad y en la parte dedicada a *"gestionar el ensayo solucion"* poner las instrucciones correspondientes para memorizar la mejor solución encontrada hasta el momento.

En el algoritmo que presentamos a continuación, los ensayos se representan mediante un vector X(1..n) y una variable j; el valor de la variable j representa el identificador del objeto que debe estudiarse a continuación para extender el ensayo. Por lo tanto el valor inicial de j será 1 y eso representa el ensayo inicial; cuando j tenga, por ejemplo, el valor 3 significa que tenemos un ensayo de longitud 2 con valores [X(1), X(2)] y cuando $j = n+1$ significa que tenemos un ensayo de longitud n y habrá que gestionarlo como solución posible. Utilizaremos un vector XOPT(1..n) y una variable VOPT para memorizar la mejor solución encontrada hasta el momento y su valor, repectivamente.

Además disponemos de dos variables SV y SP que representan el valor y el peso respectivamente del ensayo en curso X. El valor de M es la capacidad de la mochila y los valores y pesos respectivos de los objetos están representados por los vectores V(1..n) y P(1..n).


```
proc MOCHILA_0_1(SP, SV, X, j, VOPT, XOPT)
   if j = n + 1 then
      if SV > VOPT then
      XOPT(1..n) ← X(1..n)
      VOPT ← SV
      end if
   else
      for k in 0 ... 1 loop
         if SP + k×P(j) ≤ M then
            X(j) ← k
            MOCHILA_0_1(SP + X(j)×P(j), SV + X(j)×V(j), X, j + 1,
                        VOPT, XOPT)
```

La llamada inicial deberá ser MOCHILA_0_1(0, 0, X, 1, 0, XOPT), representando los valores nulos iniciales de SP, SV y VOPT, y el valor de $j = 1$. Obtendremos el resultado en el valor final de las variables VOPT y XOPT.

Análisis

El modo más conveniente de aproximarse al análisis de la eficiencia de los algoritmos de vuelta atrás es analizar el árbol implícito de ensayos explorado por el algoritmo:

1. contar el número de ensayos explorados.
2. analizar el coste de calcular las posibilidades de extender cada ensayo y el coste de generar cada nuevo ensayo.
3. analizar el coste de comprobación de la aceptación de un ensayo.

En el caso del algoritmo que acabamos de presentar para resolver el problema de la mochila es fácil comprobar que las operaciones de los apartados 2 y 3 son de orden constante y por tanto el análisis del apartado 1 determinará el orden del algoritmo.

Una cota superior del número de ensayos explorados por el algoritmo viene dada por la suma siguiente:

$$1 + 2 + 2^2 + \cdots + 2^n = 2^{n+1} - 1$$

donde el 1 corresponde al ensayo neutro inicial y cada sumando posterior es igual al doble del sumando anterior ya que cada ensayo puede extenderse

con dos nuevas opciones. La longitud máxima de un ensayo es n y por eso el último sumando es 2^n.

En consecuencia la función de coste temporal de este algoritmo es $O(2^n)$. Es importante observar que hemos establecido una cota superior y por eso utilizamos la notación asintótica o mayúscula. El rendimiento real del algoritmo puede ser mucho mejor ya que la función de aceptación de ensayos puede podar una cantidad considerable de ramas del árbol implícito. De hecho la utilidad de los algoritmos de vuelta atrás depende principalmente de la capacidad de aplicación de funciones de aceptación que poden convenientemente las ramas del árbol de ensayos.

Analizar con precisión el número de ensayos que serán realmente explorados por un algoritmo de vuelta atrás puede ser un empeño demasiado exigente. Obsérvese, por ejemplo, que en el algoritmo del problema de la mochila, el número de ensayos realmente explorado no sólo depende del tamaño n de la entrada sino también de los valores concretos de los pesos de los objetos y la capacidad de la mochila. Para un mismo tamaño de la entrada hay instancias que pueden dar lugar a un árbol con pocos ensayos y otras que pueden necesitar de la exploración de un árbol con una cantidad exponencial de ensayos.

Admitiendo que el estudio analítico de todas las posibilidades es prácticamente inabordable y adoptando el enfoque de análisis en el caso peor, deberíamos concluir que el algoritmo que hemos presentado es de orden exponencial en el tamaño de la entrada y por consiguiente sólo sería aplicable a entradas de tamaño reducido.

Puede decirse que, en general, llegaríamos a una conclusión similar con todos los algoritmos de vuelta atrás ya que normalmente, en el caso peor, necesitarán explorar árboles con una cantidad de ensayos exponencial en el tamaño de la entrada o superior. Sin embargo ese juicio sería injusto con la técnica de vuelta atrás. Típicamente, esta técnica se usa para resolver problemas para los que no se conoce otra técnica mejor. Además, frecuentemente se consigue encontrar criterios de aceptación de ensayos que podan suficientes ramas como para hacer útil el algoritmo de vuelta atrás correspondiente sobre entradas de tamaño realista.

9.1. Algoritmo alternativo a Mochila_0_1

En esta sección presentamos un algoritmo alternativo al anterior, basándonos en otro árbol de ensayos distinto.

1. En primer lugar, en vez de buscar soluciones de longitud n, consideramos solución cualquier ensayo cuyo peso no exceda la capacidad de la mochila.

2. En este caso, tenemos soluciones que son prefijos de otras soluciones.

3. Para podar ramas que ya no son prometedoras, utilizaremos un criterio que aprovecha lo que conocemos de este problema para el caso de las fracciones de objetos (véase algoritmos voraces).

4. Diremos que la extensión de un ensayo es aceptable si cabe alguna esperanza de que mejore el beneficio de la mejor carga de mochila encontrada hasta el momento.

Asumimos que $\frac{v_1}{p_1} \geq \frac{v_2}{p_2} \geq \ldots \geq \frac{v_n}{p_n}$. Consideremos que tenemos el ensayo $(x_1, \ldots, x_i)$ con $peso = \sum_{j=1}^{i} x_j p_j$ y $valor = \sum_{j=1}^{i} x_j v_j$.

Imaginemos que podemos añadir al ensayo $(x_1, \ldots, x_i)$ los objetos completos siguientes numerados hasta el $k-1$, y que excedemos la capacidad M de la mochila si añadimos el objeto número k completo.

Entonces, el peso_cota que podemos llevar en la mochila, partiendo del ensayo $(x_1, \ldots, x_i)$ es

$$peso\_cota = peso + \sum_{j=i+1}^{k-1} p_j$$

y una cota superior del beneficio sería

$$valor\_cota = (valor + \sum_{j=i+1}^{k-1} v_j) + (M - peso\_cota)\frac{v_k}{p_k}$$

Si valor_OPT es el máximo valor conseguido hasta el momento, diremos que $(x_1, \ldots, x_i)$ es aceptable si:

1. $peso \leq M$ (su peso no excede la capacidad de la mochila) y

2. $valor\_cota >$ valor_OPT (y promete más que lo conocido)

Entonces el algoritmo puede ser el siguiente:

{Precondición: $\frac{v_1}{p_1} \geq \frac{v_2}{p_2} \geq \ldots \geq \frac{v_n}{p_n}$}
proc MOCHILA_0_1_V(X, SP, SV, i, VOPT, XOPT)
  if SV > VOPT **then**
    XOPT(1..i) ← X(1..i)
    VOPT ← SV
  end if

```
if i < n then
   for s in 0...1 loop
      if SP + s×P(i+1) ≤ M ∧
            ES_ACEPTABLE(X, i+1, s, SP, SV, VOPT) then
         X(i+1) ← s
         MOCHILA_0_1_V(X, SP + s×P(i+1), SV + s×V(i+1), i+1,
                       VOPT, XOPT)
```

La llamada inicial debe ser: MOCHILA_0_1_V(X, 0, 0, 0, 0, XOPT)

```
func ES_ACEPTABLE(X, e, X_e, SP, SV, VOPT) return boolean
   k ← e
   peso_cota ← SP + X_e×P(e)
   SV ← SV + X_e×V(e)
   while k < n ∧ peso_cota ≤ M loop
      k ← k + 1
      peso_cota ← peso_cota + P(k)
      SV ← SV + V(k)
   end loop
   if peso_cota > M then
   {k es el primer objeto que no cabe}
      peso_cota ← peso_cota − P(k)
      SV ← SV − V(k)
      valor_cota ← SV + (M − peso_cota)×V(k)/P(k)
   else   {peso_cota≤ M y k = n}
      valor_cota ← SV
   end if
   return valor_cota > VOPT
```

10. Ejercicios

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.

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.

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.

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.

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.

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.

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.

8. 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.

9. Escriba un algoritmo de *vuelta atrás* para determinar el mínimo número de monedas necesarias para sumar una cantidad exacta L, cuando se dispone de n clases diferentes de monedas, con valores $v_1 \dots v_n$ respectivamente, y es posible tomar tantas monedas de cada clase como se desee.

10. Escriba un algoritmo de *vuelta atrás* que, dados dos números enteros positivos X y r, calcule todas las formas posibles de descomponer X como suma de r cuadrados. Es decir, calcule todos los multiconjuntos de r números enteros positivos $\{x_1, \dots, x_r\}$ tales que $x_1^2 + \dots + x_r^2 = X$.

11. Una agencia matrimonial dispone de las tablas de información $S(1 \dots n, 1 \dots n)$ y $C(1 \dots n, 1 \dots n)$ sobre las preferencias de n señoras y n caballeros, respectivamente. Para cada señora i, $S(i, j)$ es un número que representa el orden de preferencia de la señora i por el caballero j. Análogamente, para cada caballero x, $C(x, y)$ es un número que representa el orden de preferencia del caballero x por la señora y.

Escriba un algoritmo de *vuelta atrás* que indique a la agencia matrimonial una forma de emparejar biyectivamente a las señoras con los caballeros de manera que se minimice la suma del producto de las respectivas preferencias de cada pareja; es decir,

$$\text{minimizar} \sum_{(a,b)\in \text{Parejas}} S(a,b) \times C(b,a)$$

12. Tenemos n programas para grabar en un disco, pero el espacio de memoria que necesitan excede la capacidad del disco. Cada programa P_i requiere m_i kilobytes de memoria, la capacidad del disco es de C kilobytes y $C < \sum_{i=1}^{n} m_i$.

    Diseñe un algoritmo, utilizando la técnica de *vuelta atrás*, que calcule una colección de esos programas para grabar en el disco de manera que se minimice el espacio que queda libre en el disco después de la grabación.

13. Diseñe un algoritmo de *vuelta atrás* que, dado un número entero positivo C, calcule la lista de números enteros positivos dispuestos en orden creciente (y por lo tanto, distintos) que sumen C con la mayor cantidad de impares posible. Por ejemplo, si $C = 6$ entonces la salida puede ser: $[1, 2, 3]$ o bien $[1, 5]$ ya que ambas tienen 2 impares.

14. Considere un tablero escaqueado (como el del ajedrez) de tamaño $n \times n$, con un caballo (pieza del juego de ajedrez) situado en un escaque arbitrario de coordenadas (x, y). El problema consiste en determinar $n^2 - 1$ movimientos del caballo tal que cada escaque del tablero se visite exactamente una vez, si tal secuencia de movimientos existe.

    Para abreviar notación y cálculo de coordenadas puede disponer de la función que comentamos a continuación: El número entero m representa una de las ocho alternativas posibles de movimiento de un caballo. Comenzando por el norte, y en el sentido de las agujas del reloj, las numeramos $1 \dots 8$. Partiendo de las coordenadas (i, j) y realizando la alternativa m, las coordenadas de la posición siguiente vienen dadas por la expresión $f(i, j, m)$.

11. Bibliografía recomendada

- Brassard, G. y Bratley; P.: "Fundamentals of algorithmics". Prentice Hall (1997).

- Cormen T.H., Leiserson C.E., Rivest R.L. y Stein C.: "Introduction to Algorithms"(second edition). The MIT Press (2001).
- Horowitz E., Sahni S. y Rajasekaran S.: "Computer Algorithms". Computer Science Press (1998).
- Levitin A.: "Introduction to the design and analysis of algorithms"(second edition). Addison Wesley (2007).

12. Guías de solución de los ejercicios

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 mediante un número marcaNum(v)

distinto de 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. 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 $\Theta(n + a)$.

9. Escriba un algoritmo de *vuelta atrás* para determinar el mínimo número de monedas necesarias para sumar una cantidad exacta L, cuando

se dispone de n clases diferentes de monedas, con valores $v_1 \dots v_n$ respectivamente, y es posible tomar tantas monedas de cada clase como se desee.

Solución:
Supongamos que $v_1 \leq \ldots \leq v_n$.

Decidimos que un ensayo quede representado mediante una secuencia SEC=$[s_1, \dots s_{long}]$, de valores de monedas tal que $s_i \leq s_{i+1}$ y $\sum_{i=1}^{long} s_i \leq L$. Sea *long* y *suma* la longitud y suma de valores de SEC, respectivamente. SEC puede estar implementada con un array.

Por tanto, si $s_{long} = v_k$, k es la clase de moneda con la que comenzaríamos a extender el ensayo SEC.

Denominamos OPT y longOPT al mejor ensayo calculado hasta el momento y su longitud, respectivamente.

```
proc ENSAYAR (L, SEC, k, long, suma, OPT, longOPT)
  if suma = L then
    if long < longOPT then
      OPT ← SEC
      longOPT ← long
    end if
  else
    while long + 1 < longOPT ∧ suma + v_k ≤ L ∧ k ≤ n loop
      SEC(1 + long) ← v_k
      ENSAYAR (L, SEC, k, 1 + long, suma + v_k, OPT, longOPT)
      k ← k + 1
    end loop
```

La llamada inicial pertinente sería ENSAYAR (L, [ ], 1, 0, 0, OPT, longOPT)

10. Escriba un algoritmo de *vuelta atrás* que, dados dos números enteros positivos X y r, calcule todas las formas posibles de descomponer X como suma de r cuadrados. Es decir, calcule todos los multiconjuntos de r números enteros positivos $\{x_1, \dots, x_r\}$ tales que $x_1^2 + \dots + x_r^2 = X$.

    Solución:

    Un ensayo será una secuencia de valores $[x_1, \dots, x_k]$ de manera que $x_1 \leq \ldots \leq x_k$. Obsérvese que, debido a la conmutatividad de la suma, si estamos estudiando $1^2 + 1^2 + 2^2$ y previamente hemos estudiado $1^2 + 1^2 + 1^2 + 2^2$, no interesa repetir el trabajo con $1^2 + 1^2 + 2^2 + 1^2$.

    En el algoritmo siguiente, SEC representa el ensayo actual, de longitud $K <= r$, S es la suma de los cuadrados de los números de SEC, y j es el primer número entero con el que extenderíamos SEC. SEC está implementada con un array.

```
proc ENSAYAR (X, r, SEC, S, K, j)
  if S = X ∧ K = r then
    'Añadir SEC a la respuesta'
  else  {S < X ∨ K < r }
    while j^2 + S ≤ X(*) ∧ K < r loop
      SEC(K + 1) ← j
      ENSAYAR (X, r, SEC, S+j^2, K + 1, j)
      j ← j + 1
    end loop
```

    (*) Otra expresión booleana, para restringir más el abanico de extensiones, podría ser la siguiente. Obsérvese que el máximo j sería un i tal que $(r-K)i^2 = X-$ S (al principio, cuando $K = 0$ y S= 0, sería $i = \sqrt{\frac{X}{r}}$). Entonces una condición de terminación del blucle, más restrictiva, sería $j^2(r-K) \leq X-$ S.

    La llamada inicial sería ENSAYAR (X, r, [ ], 0, 0, 1).

11. Una agencia matrimonial dispone de las tablas de información $S(1 \dots n, 1 \dots n)$ y $C(1 \dots n, 1 \dots n)$ sobre las preferencias de n señoras y n caballeros, respectivamente. Para cada señora i, $S(i, j)$ es un número que representa el orden de preferencia de la señora i por el caballero j. Análogamente, para cada caballero x, $C(x, y)$ es un número que representa el orden de preferencia del caballero x por la señora y.

    Escriba un algoritmo de *vuelta atrás* que indique a la agencia matrimonial una forma de emparejar biyectivamente a las señoras con los caballeros de manera que se minimice la suma del producto de las respectivas preferencias de cada pareja; es decir,

$$\text{minimizar} \sum_{(a,b) \in \text{Parejas}} S(a,b) \times C(b,a)$$

    Solución:

Una posibilidad es considerar como *ensayo* una secuencia de números de $1 \ldots n$, de manera que SEC=$[3, 1, 5]$ representa que la señora número 1 está emparejada con el caballero número 3 y las señoras 2 y 3 emparejadas, respectivamente, con 1 y 5. Un ensayo puede extenderse con cualquier número de $1 \ldots n$ que no esté en la secuencia SEC.

Las posibilidades de extender un ensayo de longitud $i-1$ consisten en emparejar a la señora número i con cada uno de los caballeros que todavía no han sido emparejados.

La condición de poda más simple consiste en no continuar con un ensayo SEC de longitud p, cuya última extensión se ha realizado con el valor k, si resulta que *Beneficio*(SEC') + $S(p,k)C(k,p)$ "no tiene esperanza de mejorar el óptimo temporal calculado hasta ahora" (SEC' representa el ensayo SEC antes de añadirle el elemento k en la posición número p).

Además de la secuencia SEC, podemos representar la presencia de un número en SEC con un array, denominado hombres, de n valores booleanos. De este modo, comprobar si un número está en SEC se realiza en tiempo constante.

Denominamos OPT y sumaOPT al mejor ensayo calculado hasta el momento y su suma, respectivamente.

```
proc ENSAYAR (S, C, hombres, i, SEC, suma, OPT, sumaOPT)
{i representa la señora en curso}
  if i > n then
    if suma < sumaOPT then
      OPT(1..n) ← SEC(1..n)
      sumaOPT ← suma
    end if
  else
    for j in 1...n loop
      if ¬hombres(j) then
        hombres(j) ← true
        if suma + (S(i,j) × C(j,i)) < sumaOPT then
          SEC(i) ← j
          ENSAYAR (S, C, hombres, i+1, SEC,
                   suma+(S(i,j) × C(j,i)), OPT, sumaOPT)
        end if
        hombres(j) ← false
      end if
    end loop
```


La llamada inicial sería: ENSAYAR (S, C, hombres, 1, [ ], 0, OPT, ∞).

12. Tenemos n programas para grabar en un disco, pero el espacio de memoria que necesitan excede la capacidad del disco. Cada programa P_i requiere m_i kilobytes de memoria, la capacidad del disco es de C kilobytes y $C < \sum_{i=1}^{n} m_i$.

    Diseñe un algoritmo, utilizando la técnica de *vuelta atrás*, que calcule una colección de esos programas para grabar en el disco de manera que se minimice el espacio que queda libre en el disco después de la grabación.

    Solución:
    Una posibilidad es la siguiente: Un ensayo es una secuencia de números tomados de $1 \ldots n$ que representan a los programas correspondientes que grabaríamos en el disco. Por ejemplo $[2, 5, 7]$ representa la posibilidad de grabar los programas número $2, 5$ y 7. Para no estudiar ensayos repetidos decidimos que un ensayo sólo puede extenderse con un número mayor que el último de la secuencia. Así, el ensayo ejemplo puede extenderse con $[2, 5, 7][8]$, $[2, 5, 7][9]$ y aumentando el último número hasta $[2, 5, 7][n]$. Un ensayo es aceptable si la suma de sus kilobytes de memoria no supera la capacidad del disco C. Con este diseño, todo ensayo aceptable es una solución. Además, tendremos soluciones que son prefijo de otras soluciones (habrá que optar por el esquema de *vuelta atrás* que tiene en cuenta esa posibilidad). Según vamos encontrando soluciones nos vamos quedando con la que maximiza la cantidad de memoria ocupada.

    Podemos añadir una condición más para restringir el número de posibilidades de extensión de un ensayo. Si ordenamos previamente los programas según su ocupación de memoria (de menor a mayor, $m_k \leq m_{k+1}$) entonces cuando una extensión $[2, 5, 7][k]$ no es aceptable, tampoco lo serán las siguientes y por tanto podemos terminar el bucle de extensiones de ensayos.

    Otra posibilidad es considerar un ensayo como una secuencia de unos y ceros que representan la inclusión o no, en la solución, del programa que ocupa esa posición en la numeración. Por ejemplo, $[0, 1, 0, 0, 1, 0, 1]$ representa al ejemplo de antes, que incluye los programas número $2, 5$ y 7. Las posibilidades de extensión son dos 1 y 0. Un ensayo es aceptable si la suma de sus correspondientes kilobytes de memoria no supera la capacidad del disco C. Un ensayo es solución cuando tiene longitud n. Según vamos encontrando soluciones nos vamos quedando con la que maximiza la cantidad de memoria ocupada.

También aquí, si ordenamos los programas según necesidad creciente de memoria ($m_k \leq m_{k+1}$), podemos podar algunas ramas siguiendo el mismo criterio que vimos en la solución del problema de la suma de subconjuntos.

13. Diseñe un algoritmo de *vuelta atrás* que, dado un número entero positivo C, calcule la lista de números enteros positivos dispuestos en orden creciente (y por lo tanto, distintos) que sumen C con la mayor cantidad de impares posible. Por ejemplo, si $C = 6$ entonces la salida puede ser: $[1, 2, 3]$ o bien $[1, 5]$ ya que ambas tienen 2 impares.

    Solución:
    A continuación se presenta una solución inmediata, sin aplicación de posibles optimizaciones.

    Un ensayo será una secuencia de números enteros positivos en orden creciente. Las posibilidades de extender un ensayo que termina con el número k consisten en añadir a la secuencia un número. Para garantizar el orden creciente, la primera extensión se realiza con el número $k + 1$ y posteriormente con sus siguientes hasta llegar al máximo que sea aceptable. Es decir, hasta llegar a un número que sumado a los demás del ensayo supere la cantidad C.

    El parámetro S representa el ensayo, el parámetro Impares representa la cantidad de números impares que hay en el ensayo S, y el parámetro Suma representa la suma de los números que hay en S. Análogamente, SOPT e ImparesOPT representan la mejor solución encontrada hasta el momento y el número de impares que contiene. El parámetro k representa el último número de la secuencia S.

    Escribimos el algoritmo suponiendo que representamos el ensayo S con una estructura de lista, sobre la que usamos las operaciones AÑADIR_AL_FINAL(S, x) (que añade el elemento x al final de la lista S) y RETIRAR_ULTIMO(S) (que retira el último de la lista S).

```
proc ENSAYAR (C, S, Impares, Suma, k, SOPT, ImparesOPT)
  if Suma = C then
    if Impares > ImparesOPT then
      SOPT ← S
      ImparesOPT ← Impares
    end if
  else
    e ← k + 1
    while Suma + e ≤ C loop
```



```
      AÑADIR_AL_FINAL(S, e)
      Suma ← Suma + e
      if e mod 2 = 1 then Impares ← Impares + 1 end if
      ENSAYAR(C, S, Impares, Suma, e, SOPT, ImparesOPT)
      RETIRAR_ULTIMO(S)
      Suma ← Suma - e
      if e mod 2 = 1 then Impares ← Impares − 1 end if
      e ← e + 1
    end loop
```

La llamada inicial será ENSAYAR(C, [ ], 0, 0, 0, [ ], 0).

14. Considere un tablero escaqueado (como el del ajedrez) de tamaño $n \times n$, con un caballo (pieza del juego de ajedrez) situado en un escaque arbitrario de coordenadas (x, y). El problema consiste en determinar $n^2 - 1$ movimientos del caballo tal que cada escaque del tablero se visite exactamente una vez, si tal secuencia de movimientos existe.

    Para abreviar notación y cálculo de coordenadas puede disponer de la función que comentamos a continuación: El número entero m representa una de las ocho alternativas posibles de movimiento de un caballo. Comenzando por el norte, y en el sentido de las agujas del reloj, las numeramos $1 \ldots 8$. Partiendo de las coordenadas (i, j) y realizando la alternativa m, las coordenadas de la posición siguiente vienen dadas por la expresión $f(i, j, m)$.

    Solución:
    Un ensayo puede representarse mediante una pila de las posiciones alcanzadas hasta el momento. Inicialmente *ensayo*=$[(x, y)]$, que son las coordenadas de partida. Cuando la longitud de *ensayo* sea n^2 tendremos una solución. No son posiciones aceptables para incluir en *ensayo* aquellas cuyas coordenadas queden fuera del tablero o bien sean coordenadas presentes en *ensayo*. Gestionaremos una matriz de marcas $V(1..n, 1..n)$ que registre las coordenadas visitadas: Inicialmente $V(x, y) = true$ y $\forall (i, j) \neq (x, y)$. $V(i, j) = false$.

```
proc CABALLO (ensayo, éxito, V(1..n, 1..n))
  if ensayo.LONG() = n² then
    éxito ← true
    IMPRIMIR(ensayo)
  else
    for m ∈ 1 . . . 8 loop
```

```
i ← ensayo.CIMA().pri {primera coordenada de la cima}
j ← ensayo.CIMA().seg {segunda coordenada de la cima}
if DENTRO_DE_TABLERO(f(i,j,m))
     ∧ V(f(i,j,m).pri, f(i,j,m).seg) = false then
  V(f(i,j,m).pri, f(i,j,m).seg) ← true
  ensayo.PUSH(f(i,j,m).pri, f(i,j,m).seg)
  CABALLO (ensayo, éxito, V(1..n, 1..n))
  if éxito then
    return
  else
    (i,j) ← ensayo.POP()
    V(i,j) ← false
```

La llamada inicial será CABALLO ($[(x,y)]$, $false$, V), con la inicialización comentada de V.