Reto 4.

Mario Román

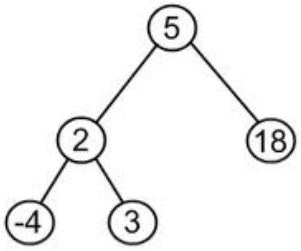
6 de abril de 2015

1. Codificación de un árbol binario.

1.1. Consideraciones previas

Vamos a intentar codificar un árbol binario con la menor cantidad de información posible, esto es, con el menor número de bits que podamos. La condición que debe tener una codificación es que sea unívoca, y nos permita recuperar el árbol de manera única una vez codificado.

Queremos codificar un árbol binario genérico, que puede contener como datos cualesquiera etiquetas. La codificación de estos datos, que podrían tener diversa naturaleza (las etiquetas podrían ser enteros, cadenas, o incluso listas de colas de pilas de árboles de booleanos), se debería considerar aparte. Dado un árbol, tomaremos su preorden y lo almacenaremos en un vector. Este vector contiene la información de las etiquetas del árbol e, implícitamente, el número de nodos que tiene.



Árbol de ejemplo.

En el ejemplo, de 5 nodos, codificaríamos:

$$[5, 2, -4, 3, 18]$$

Es decir, que nuestro problema se centra en codificar la estructura, el esqueleto de un árbol binario. Consideramos que las etiquetas se codifican aparte.

Intentaremos conseguir la mejor codificación posible de la estructura de un árbol de n nodos creando una función biyectiva entre árboles binarios y números naturales; y codificando el número natural que nos dé la biyección.

1.2. Orden total entre árboles.

La herramienta que usaremos para hacer esto posible es la definición de un orden total entre esqueletos de árboles. Abusaremos del lenguaje llamándolos árboles, pero nótese que no tendrán etiquetas.

Definimos un esqueleto de árbol binario con la forma $A = (A_i, A_d)$, es decir, constando sólo de un un subesqueleto izquierdo, y un subesqueleto derecho; o como un esqueleto vacío $A = nulo$. Nótese que la definición es igual a la de los árboles binarios pero sin etiqueta.

Ahora definimos la relación de orden como sigue:

$$A = (A_i, A_d) \leqslant B = (B_i, B_d)$$

Si se cumple:

$$nodostotales(A) < nodostotales(B)$$

O si tienen el mismo número de nodos pero:

$$A_d < B_d$$

O si tienen el mismo subárbol derecho pero:

$$A_i \leqslant B_i$$

Es decir, primero comparamos el número de nodos, luego el subárbol derecho, y como último criterio, comparamos el subárbol izquierdo.

Claramente es una relación de orden. Es antisimétrica porque dos árboles con mismos nodos y subárboles deben ser iguales. Es transitiva por ser composición de varias relaciones de orden. Y es trivialmente reflexiva.

Pero además, es una relación de orden total. Trivialmente lo es para árboles de un nodo. Y, por inducción sobre el número de nodos lo demostramos para árboles de n nodos:

Que sea relación de orden total para árboles de menos n nodos fuerza a que deba darse $A_i \leqslant B_i$ o $A_i \leqslant B_i$, ya que el subárbol izquierdo tiene menos de n nodos, y por tanto, al menos la tercera condición debe cumplirse en alguno de los dos sentidos.

1.3. Convertir árboles en números.

Ahora usaremos que existen $\frac{1}{n+1}\binom{2n}{n}$ árboles binarios de n nodos. Llamamos A_n al conjunto de esqueletos de árboles binarios de n nodos, y sabemos que:

$$\#A_n = \frac{1}{n+1}\binom{2n}{n} = C_n$$

Definimos la siguiente función:

$$f : A_n \longrightarrow \{1, 2, \ldots, C_n\}$$

Como la única biyección estrictamente creciente entre los dos conjuntos. Es decir, la única que cumple:

$$\forall X, Y \in A_n : X \leqslant Y \iff f(X) \leqslant f(Y)$$

Es resultado conocido que entre dos conjuntos con el mismo cardinal y con dos relaciones totales de orden, existe una única aplicación biyectiva creciente.

Una forma práctica de hallar la imagen de un árbol por f es generar la lista ordenada de posibles estructuras de árboles de n y realizar una búsqueda binaria sobre ella (aprovechando que hemos definido una relación de orden) hasta encontrar el índice de un árbol que coincida con el árbol buscado. Esa es su codificación. Lo que estamos haciendo, expuesto de otra forma, es contar todos los posibles árboles, ordenarlos e irles asignando su número según el orden. En la implementación se concretan estos detalles.

Otra forma, que no se desarrolla aquí, mucho más eficiente, es la de relacionar el valor de f para un árbol de n nodos, con los valores de $f(A_i)$ y $f(A_d)$, que pueden buscarse inductivamente.

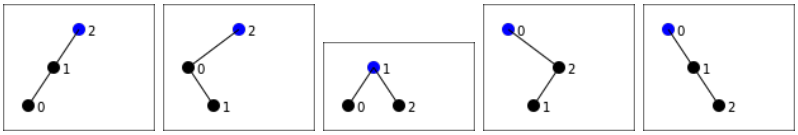
1.4. Palabras binarias y codificación.

La codificación de un árbol binario A se define entonces como un vector con sus elementos en preorden, que nos da el número de nodos del árbol, y el número $f(A)$ escrito como una palabra binaria. Es decir, los números pueden escribirse con los mínimos bits posibles siguiendo este patrón.

$0 \rightarrow$ ""
$1 \rightarrow$ "0"
$2 \rightarrow$ "1"
$3 \rightarrow$ "00"
$4 \rightarrow$ "10"
$5 \rightarrow$ "01"
$6 \rightarrow$ "11"
$7 \rightarrow$ "000"
$8 \rightarrow$ "100"
...

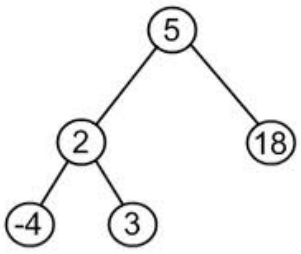
Es decir, guardamos por un lado los nodos, y por otro escribimos el número que determina unívocamente el árbol por la función f.

Por ejemplo, vamos a escribir los árboles de tres nodos ordenados:



Donde las estructuras de los árboles tendrán, respectivamente, las codificaciones siguientes: "","0","1","00","10"

Un ejemplo con el árbol inicial, un árbol con nodos significativos y no sólo estructura, es el siguiente:



Árbol de ejemplo, de nuevo.

Su codificación será: $([5, 2, -4, 3, 18], "1010")$. Ya que es el número 25 entre los posibles árboles de 5 nodos, y la codificación de 25 es "1010". La otra parte de la codificación es el inorden de los nodos.

1.5. Tamaño de la codificación.

Ahora vamos a medir cuánto ocupa la codificación de un árbol de n nodos. Es claro que las etiquetas, si son de tipo $Type$, ocuparán en disco:

$$n * sizeof(Type)$$

Quizá algo más si contamos un caracter terminador o uno que indique cuántos elementos hay.

La parte más interesante para medir será cuánto ocupa la codificación de la estructura. Tenemos que una palabra binaria de, como máximo, m bits puede codificar:

$$\sum_{k=0}^{m} 2^k = 2^{m+1} - 1$$

Donde cada sumando representa las 2^b cadenas posibles con b bits.

Pero tenemos por otro lado que hay $\frac{1}{n+1}\binom{2n}{n}$ posibles árboles, si queremos saber cuánto espacio necesitaremos para codificarlos, resolvemos:

$$2^{m_n+1} - 1 = \frac{1}{n+1}\binom{2n}{n};$$
$$m_n = \left\lceil log_2 \left(\frac{1}{n+1}\binom{2n}{n} + 1 \right) - 1 \right\rceil;$$

Este será el espacio que usemos. Asintóticamente, sabemos que la función de los números de Catalan crece como:

$$C_n \sim \frac{4^n}{n^{3/2}\sqrt{\pi}}$$

Por lo que la función en conjunto se aproximará a:

$$m_n \sim \left\lceil log_2\left(\frac{4^n}{n^{3/2}\sqrt{\pi}}+1\right)-1\right\rceil \sim 2n - log_2\left(n^{3/2}\sqrt{\pi}\right) \sim 2n - \frac{3}{2}log_2(n)$$

Que aunque siga siendo $O(n)$, es menor que la que use invariablemente $2n$, a pesar de que se aproxime asintóticamente a ella.

1.6. Mejor codificación.

Toda codificación de los esqueletos de árboles binarios como palabras binarias debe ser inyectiva, para permitir que pueda recuperarse de ella el árbol binario original de manera unívoca, y sin dar lugar a pérdida de información. Por tanto, será una biyección con su conjunto imagen. El cardinal de la imagen de una biyección debe tener al menos el mismo número de elementos que el cardinal del conjunto de definición, en este caso, para n nodos, C_n elementos.

La función aquí presentada alcanza el mínimo especificado en esa cota, así que cualquier otra codificación debe usar los mismos o más elementos que ella. Es decir, cualquier otra codificación llevará árboles en palabras binarias que como poco, alcanzarán las C_n palabras distintas, y necesitará usar, como poco, tantos bits como en este caso.

Podríamos obtener mejoras en la codificación de las etiquetas, o si tenemos una restricción sobre ellas, o si nos preocupamos en codificar mejor los árboles más comunes para mejorar el caso medio; pero en las condiciones que hemos fijado, tenemos la mínima codificación posible.

1.7. Implementación.

Se presenta una implementación de la codificación en lenguaje de programación Haskell. Usamos por comodidad este lenguaje porque nos permite hacer definiciones más concisas de los árboles y las funciones sobre ellos que C++, a pesar de que perdemos mucha eficiencia. Nótese que no se ha hecho referencia a la eficiencia del algoritmo en ningún momento. Sólo nos interesa obtener la minima codificación, y los algoritmos usados para comprimir y descomprimir serán muy ineficientes.

Al final de la implementación se da un ejemplo de uso.

```
import Data.List
import Data.Maybe
import Data.Char
```

1.7.1. Esqueletos de árbol

Definimos lo que será el esqueleto de un árbol, es decir, un árbol sin etiquetas en los nodos, un arbol binario puro, considerando solo su estructura.

Un esqueleto se define como un esqueleto vacío o como un punto (nodo) del que salen dos esqueletos de árbol.

```
data EsqArbol = EsqVacio
  | Punto (EsqArbol) (EsqArbol)
    deriving (Show, Eq)
```

Definimos una función recursiva trivial para contar nodos de un esqueleto. Creamos una función que crea todos los posibles esqueletos de n nodos. Para ello, en el caso inductivo, crea, para todo i en el rango [0,...,n-1] todos los posibles árboles con i nodos a la izquierda y todos los posibles con n-1-i nodos a la derecha.

```
nodos :: EsqArbol → Int
nodos EsqVacio = 0
nodos (Punto a b) = 1 + (nodos a) + (nodos b)
arbolesNodos :: Int → [EsqArbol]
arbolesNodos 0 = [EsqVacio]
arbolesNodos 1 = [(Punto EsqVacio EsqVacio)]
arbolesNodos n = concat [[Punto x y | y ← arbolesNodos (n − 1 − i), x ← arbolesNodos (i)] | i ← [0 . . n
```

Como curiosidad, podemos notar que hay C_n arboles de n nodos, donde C_n es el enésimo numero de Catalan. La longitud de la lista creada por `arbolesNodos n` es C_n.

Ahora vamos a codificar un esqueleto binario. Para ello, asignamos a cada árbol su posicion en la lista de arboles generados, y escribimos el número como palabra binaria.

Para pasar a binario, intentamos aprovechar al maximo el espacio, para ello usamos todas las palabras binarias posibles. Es decir, transformamos:

$$0- > \_$$

$$1- > 0$$

$$2- > 1$$

$$3- > 00$$

$$4- > 01$$

Adjuntamos además otra funcion para pasarlo a una cadena legible de texto.

```
aBinario :: Int → [Int]
aBinario 0 = []
aBinario n = r : aBinario q
  where (q, r) = divMod (n − 1) 2
binarioLegible :: Int → String
binarioLegible x = map intToDigit $ reverse $ aBinario x
codifica :: EsqArbol → Int
codifica x = fromJust $ elemIndex x (arbolesNodos len)
  where len = nodos x
codificaLegible :: EsqArbol → String
codificaLegible x = binarioLegible $ codifica x
```

Por último, implementamos la decodificación. Nótese que aquí debe conocerse el numero de nodos.

```
aNatural :: [Int] → Int
aNatural [] = 0
aNatural (x : xs) = 2 * (aNatural xs) + x + 1

decodifica :: Int → Int → EsqArbol
decodifica x nodos = (arbolesNodos nodos) !! x

decodificaBinario :: [Int] → Int → EsqArbol
decodificaBinario xs nodos = decodifica (aNatural xs) nodos

decodificaLegible :: String → Int → EsqArbol
decodificaLegible s nodos = decodificaBinario (reverse (map digitToInt $ s)) nodos
```

1.7.2. Árboles de verdad

Para ver la aplicacion real del algoritmo, vamos a escribir arboles de verdad y a codificarlos. Definimos un arbol de manera parecida al esqueleto. Como diferencia, ahora permitimos que la etiqueta contenga un dato de tipo a.

```
data Arbol a = ArbolVacio
  | Nodo a (Arbol a) (Arbol a)
  deriving (Show, Eq)
```

Ahora codifica un árbol binario. Para ello, toma sus elementos y los guarda, por otro lado, lo transforma en esqueleto. La codificacion son sus elementos, implicita su longitud, y la codificacion binaria del esqueleto.

```
transformaEsqueleto :: Arbol a → EsqArbol
transformaEsqueleto ArbolVacio = EsqVacio
transformaEsqueleto (Nodo _ a b) = (Punto (transformaEsqueleto a) (transformaEsqueleto b))

preorden :: Arbol a → [a]
preorden ArbolVacio = []
preorden (Nodo x a b) = x : ((preorden a) ++ (preorden b))

codificaArbol :: Arbol a → ([a], String)
codificaArbol x = (preorden x, codificaLegible ∘ transformaEsqueleto $ x)
```

Y ahora decodifica un arbol binario. Para ello, decodifica el esqueleto del arbol y rellena el esqueleto con los elementos del preorden.

```
rellenaEsqueleto :: EsqArbol → [a] → Arbol a
rellenaEsqueleto EsqVacio _ = ArbolVacio
rellenaEsqueleto _ [] = ArbolVacio
rellenaEsqueleto (Punto u v) (x : xs) = (Nodo x subIzquierdo subDerecho)
  where (parteIzquierda, parteDerecha) = splitAt (nodos u) xs
    subIzquierdo = rellenaEsqueleto u parteIzquierda
    subDerecho = rellenaEsqueleto v parteDerecha

decodificaArbol :: ([a], String) → Arbol a
decodificaArbol (datos, codificado) = rellenaEsqueleto (decodificaLegible codificado nodos) datos
  where nodos = length datos
```

1.7.3. Funciones auxiliares

Para poder aprovechar la codificacion creada, se presentan funciones auxiliares que faciliten ejemplos de uso.

```
insercionEnArbol :: (Ord a) ⇒ a → Arbol a → Arbol a
insercionEnArbol x ArbolVacio = (Nodo x ArbolVacio ArbolVacio)
insercionEnArbol x (Nodo a izq der)
    | x ⩽ a = Nodo a (insercionEnArbol x izq) der
    | x > a = Nodo a izq (insercionEnArbol x der)
crearArbol :: (Ord a) ⇒ [a] → Arbol a
crearArbol = foldr insercionEnArbol ArbolVacio ∘ reverse
```

Definimos el orden total entre esqueletos de árbol.

```
esqCompara :: EsqArbol → EsqArbol → Ordering
esqCompara EsqVacio EsqVacio = EQ
esqCompara EsqVacio _ = LT
esqCompara _ EsqVacio = GT
esqCompara a b
    | (nodos a) < (nodos b) = LT
    | (nodos a) > (nodos b) = GT
    | (esqCompara ad bd) ≡ LT = LT
    | (esqCompara ad bd) ≡ GT = GT
    | otherwise = (esqCompara ai bi)
      where (Punto ai ad) = a
            (Punto bi bd) = b
```

1.7.4. Ejemplo de uso

Para comprobar el correcto funcionamiento de la codificacion expuesta, probamos a codificar un arbol sencillo. El resultado de la ejecucion de una sesion interactiva de ghci, compilando este codigo, se muestra a continuacion.

– Creamos un arbol binario desde la lista [2,5,3,4,6,5]

*¿let p = crearArbol [2,5,3,4,6,5] *¿p Nodo 2 ArbolVacio (Nodo 5 (Nodo 3 ArbolVacio (Nodo 4 ArbolVacio (Nodo 5 ArbolVacio ArbolVacio))) (Nodo 6 ArbolVacio ArbolVacio))

– Codificamos el arbol

*¿let cod = codificaArbol p *¿cod ([2,5,3,4,5,6],"1000")

– Decodificamos y comprobamos que equivale al original.

*¿decodificaArbol cod Nodo 2 ArbolVacio (Nodo 5 (Nodo 3 ArbolVacio (Nodo 4 ArbolVacio (Nodo 5 ArbolVacio ArbolVacio))) (Nodo 6 ArbolVacio ArbolVacio))