Práctica 4: árboles de probabilidad (práctica final)

Nuevas tecnologías de la programación

Contenido:

1 Objetivos **1**

2 Representación **2**

3 Variable **3**

4 Dominio **3**

5 Asignación **5**

6 Valores **7**
- 6.1 ValoresArray 10
- 6.2 ValoresArbol 12
  - 6.2.1 Nodo, NodoHoja y NodoVariable 12
  - 6.2.2 Construcción de objetos de tipo ValoresArbol 17

7 Clases Potencial, PotencialArray y PotencialArboles **19**

8 Observaciones **20**

9 Material a entregar **20**

1 Objetivos

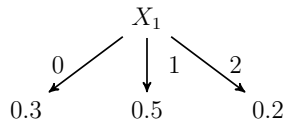
El objetivo de esta práctica es trabajar de forma autónoma en el diseño e implementación de un sistema completo usando las características de la orientación a objetos ofrecida por Scala. El sistema a implementar debe permitir la representación y manejo, de forma flexible, de distribuciones de probabilidad (en forma de tablas (arrays) y árboles).

2 Representación

El elemento básico a representar en el sistema es la distribución de probabilidad (llamada **potencial** de forma general). Los objetos de este tipo, sea cual sea su representación interna, asignan un valor a cada combinación de valores de un conjunto de variables. Supongamos que se define un potencial $P(X_1)$, donde X_1 es una variable con tres posibles valores (los valores se denominarán con enteros, comenzando en 0). En este caso, el potencial podría quedar definido de la siguiente forma:

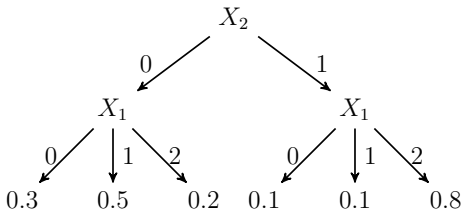
X_1	$P(X_1)$
$X_1 = 0$	0.3
$X_1 = 1$	0.5
$X_1 = 2$	0.2

La representación indicada es la más inmediata y se denomina **tabla de probabilidad**. Es posible otra representación alternativa más ventajosa mediante una estructura tipo árbol:



El ejemplo anterior muestra una distribución (potencial) definida sobre una única variable, pero podría tratarse de la misma forma el caso de potenciales definidos sobre varias variables. Por ejemplo, imaginemos que el dominio está compuesto por X_2 (variable con dos posibles estados) y X_1 (variable con tres estados):

X_2, X_1	$P(X_2, X_1)$
$X_2 = 0, X_1 = 0$	0.3
$X_2 = 0, X_1 = 1$	0.5
$X_2 = 0, X_1 = 2$	0.2
$X_2 = 1, X_1 = 0$	0.1
$X_2 = 1, X_1 = 1$	0.1
$X_2 = 1, X_1 = 2$	0.8



A continuación se describen los elementos necesarios para poder representar distribuciones de probabilidad en nuestro sistema

3 Variable

Es el elemento básico de este sistema. Cada variable queda caracterizada por un nombre (cadena de caracteres) y un número de estados. Se asume que una variable con 3 estados tiene como valores posibles $\{0, 1, 2\}$.

Debería permitirse la creación de los objetos de la clase tal y como se indica en el siguiente ejemplo de código:

```
1    val X1 = Variable("X1",3)
2    val X2 = Variable("X2",4)
3    val X3 = Variable("X3",2)
4    val X4 = Variable("X4",2)
5    val X5 = Variable("X5",2)
6    val X6 = Variable("X6",2)
```

4 Dominio

Los potenciales se definen realmente sobre conjuntos de variables (el almacenamiento podría hacerse mediante una lista, asumiendo que no hay variables repetidas). Por conveniencia interesa almacenar el índice de cada variable (**indiceVariables**) y sus pesos (**pesosVariables**). En ambos casos podríamos usar mapas para guardar esta información.

El peso de una variable representa el valor asociado a la misma. Por ejemplo, imaginemos el dominio $\{X_1, X_2, X_3\}$, con 5, 4 y 3 valores posibles. Los pesos correspondientes serían: $\{12, 3, 1\}$ (el peso de X_3, la variable que representan las *unidades* es 1; el peso de X_2 es igual al número de valores posibles de X_1; el peso de X_3 se obtiene multiplicando el número de estados de las variables previas). En definitiva, el peso de cada variable se obtiene multiplicando el número de estados de las variables de menor valor (a la derecha, en la lista). Para poder recorrer los posibles valores de un dominio de variables se consideran índices: un índice de valor 0 representa la combinación de valores de variables en que todas ellas toman su menor valor (0). Dada una cierta combinación de valores de variables ($X_1 = 1, X_2 = 3, X_3 = 2$) el índice correspondiente se obtendría multiplicando los valores de las variables por su peso: $1 * 12 + 3 * 3 + 2 * 1 = 23$. Algunas funciones de interés para los objetos de esta clase podrían ser:

- **vacio**: determina si se trata de un dominio vacio
- **longitud**: indica el número de variables que forman el dominio
- **pesos**: devuelve la lista de pesos de un dominio
- **maximoIndice**: devuelve el máximo índice del dominio (en realidad una unidad más). En el caso del ejemplo indicado más arriba el máximo valor del índice sería 60 y 59 se correspondería entonces con la situación en que todas las variables toman sus valores máximos: $X_1 = 4, X_2 = 3, X_3 = 2$. Esto se debe a que el primer índice válido es 0.
- **apply**: de forma que podamos obtener la variable que ocupa una determinada posición en el dominio

- **toString**: para facilitar los mensajes por pantalla que pudieran necesitarse.
- +: permitiendo agregar una nueva variable al dominio. Se sigue siempre la idea básica de inmutabilidad: el resultado de esta operación será un nuevo dominio donde se agrega la variable pasada como argumento, siempre que no aparezca ya en él.
- -: permite eliminar una variable del dominio. El resultado será un nuevo objeto de la clase.

En el siguiente fragmento de código se muestran algunas operaciones realizadas sobre el dominio (deben usarse simplemente a modo de ejemplo; la misma funcionalidad puede obtenerse considerando el diseño que os parezca más oportuno):

```
1   // Se crea dominio vacio
2   val dominioVacio=Dominio(List())
3
4   // Se comprueba que funciona el metodo asociado a comprobar la condicion
5   // de dominio vacio
6   println("Comprobacion de vacio sobre dominio vacio: "+ dominioVacio.vacio)
7
8   // Se crean 4 variables
9   val X1 = Variable("X1",3)
10  val X2 = Variable("X2",4)
11  val X3 = Variable("X3",2)
12  val X4 = Variable("X4",2)
13
14  // Se crea un dominio con las variables creadas antes
15  val dominioNoVacio=Dominio(List(X1,X2,X3,X4))
16
17  // Este dominio ya no esta vacio
18  println("Comprobacion de vacio sobre dominio no vacio: "+dominioNoVacio.vacio)
19
20  // Se obtiene la longitud del dominio
21  val longitud=dominioNoVacio.longitud
22  println("Longitud del dominio no vacio (debe ser 4): "+longitud)
23
24  // Se muestra el objeto usando toString
25  println(dominioNoVacio)
26
27  // Se suma una variable al dominioNoVacio
28  val X5 = new Variable("X5", 5)
29  val dominioSuma=dominioNoVacio+X5
30  println("Dominio suma (+X5): "+dominioSuma)
31
32  // Se crea un dominio sobre X4, X5 y X6
33  val X6 = new Variable("X6", 3)
34  val dominioNoVacio2=new Dominio(List(X1, X2, X5, X6))
35
36  // Se genera ahora la suma de los dos dominios no vacios
37  val dominioSuma2=dominioNoVacio + dominioNoVacio2
38  println("Suma de dominios: "+dominioSuma2)
39
40  // Prueba de calculo del maximo indice
```

```
41    val maximoIndice=dominioSuma2.maximoIndice
42    println("Maximo indice de dominio de suma: "+ maximoIndice)
```

La salida obtenida es (se han partido algunas líneas para mejorar la presentación):

```
Comprobacion de vacio sobre dominio vacio: true
Comprobacion de vacio sobre dominio no vacio: false
Longitud del dominio no vacio (debe ser 4): 4
X1(s: 3 w: 16) X2(s: 4 w: 4) X3(s: 2 w: 2) X4(s: 2 w: 1)
Dominio suma (+X5): X1(s: 3 w: 80) X2(s: 4 w: 20) X3(s: 2 w: 10) X4(s: 2 w: 5)
X5(s: 5 w: 1)
Operacion de suma de dominios (X1, X2, X3, X4) + (X1, X2, X5, X6)
Suma de dominios: X1(s: 3 w: 240) X2(s: 4 w: 60) X3(s: 2 w: 30) X4(s: 2 w: 15)
X5(s: 5 w: 3) X6(s: 3 w: 1)
Maximo indice de dominio de suma (X1, X2, X3, X4, X5, X6): 720
```

5 Asignación

Esta clase representa la combinación de un dominio con valores para sus variables. Por esta razón queda caracterizado por un dominio y una lista de valores. La creación del objeto requerirá que se comprueben los valores de las variables (que tienen que estar en el rango comprendido entre 0 y $s-1$, donde s representa el número de estados de la variable correspondiente).

Conviene almacenar los valores de cada variable en un mapa con pares **variable valor** (en un dato miembro que podría llamarse **datos**). Algunas funciones de interés se indican a continuación:

- **vacia**: indica si la asignación está definida sobre un dominio vacío o no.
- **obtenerNumeroVariables**: devuelve el número de variables que define la asignación (su dominio).
- **obtenerValorVariable**: recibe como argumento el valor de una variable y devuelve el valor asignado a la misma.
- $+$: permite agregar un par valor-variable a la asignación.
- **calcularIndice**: devuelve el índice asociado a la asignación (mediante la forma de cálculo comentada previamente: la suma de la multiplicación del valor de cada variable por su peso).
- **toString**: muestra por pantalla la información relevante de la asignación.
- **proyectar**: recibe como argumento un dominio y devuelve como resultado una asignación en que se encuentran únicamente las variables presentes en el dominio y sus valores.
- se podrán crear objetos de diferentes formas:

- a partir de un dominio y de una lista de valores.
- a partir de un dominio únicamente; se asignará el valor 0 a todas las variables.
- a partir de un dominio y un valor de índice. En este caso hay que calcular el valor de cada variable dado el índice. Supongamos el dominio mencionado anteriormente formado por las variables $\{X_1, X_2, X_3\}$ con 5, 4 y 3 estados respectivamente. Para calcular el valor de cada variable asociado a un índice hay que dividir el índice por el peso la variable (división entera) y calcular posteriormente el resto de la división por el número de estados de la variable. Por ejemplo, imaginemos el índice 59. Para calcular qué valor tendría la variable X_1 en la asignación correspondiente a dicho índice se calcula $59/12 = 4$. Ahora se calcularía $4\%5 = 4$, obteniéndose el valor de dicha variable. Recordemos que el índice 59 representa la asignación dada por $X_1 = 4, X_2 = 3, X_3 = 2$.

El código incluido a continuación muestra algunos ejemplos de uso:

```
1   // Se crea asignacion vacia y se comprueba su chequeo
2   val asignacionVacia=Asignacion(Dominio(List()), List())
3   println("Comprobacion vacio asignacion vacia: "+asignacionVacia.vacia)
4
5   // Se crean 4 variables con diferentes estados
6   val X1 = Variable("X1",3)
7   val X2 = Variable("X2",4)
8   val X3 = Variable("X3",2)
9   val X4 = Variable("X4",2)
10
11  // Se crea asignacion , dando valores 2, 3, 1 y 0 a las variables
12  val asignacion1=Asignacion(Dominio(List(X1, X2, X3, X4)), List(2,3,1,0))
13  println("Comprobacion vacio sobre asignacion no vacia: " + asignacion1.vacia)
14  println("Se muestra la asignacion: ")
15  println(asignacion1)
16
17  // Calculo del indice asociado a la asignacion (debe ser 46)
18  val indice1=asignacion1.calcularIndice
19  println("indice1 (debe ser 46): " + indice1)
20
21  // A partir del indice obtenemos la asignacion
22  val asignacionDeIndice = Asignacion(asignacion1.dominio, indice1)
23
24  // Se muestra la asignacion obtenida: debe ser X1=2, X2=3, X3=1, X4=0
25  println("Asignacion resultante: " + asignacionDeIndice)
```

```
Comprobacion vacio asignacion vacia: true
Comprobacion vacio sobre asignacion no vacia: false
Se muestra la asignacion:
[X1 - 2]  [X2 - 3]  [X3 - 1]  [X4 - 0]
indice1 (debe ser 46): 46
Asignacion resultante: [X1 - 2]  [X2 - 3]  [X3 - 1]  [X4 - 0]
```

6 Valores

Los objetos de esta clase se usan para almacenar y gestionar los valores que definen los potenciales. Para permitir una representación flexible de los valores el sistema debe poder adaptarse a la gestión mediante arrays y mediante árboles. Por esta razón habrá clases específicas para estos tipos de almacenamiento: **ValoresArray** y **ValoresArbol**. Todos los objetos tipo **Valores** se caracterizan por un dominio (el mismo del potencial al que se asociarán). La funcionalidad que deberán aportar los objetos de esta clase es la siguiente:

- **toString**: permite mostrar por pantalla la información relativa a un potencial. La forma de visualizar los objetos de esta clase dependerán de su tipo concreto específico. A continuación pueden verse la cadena de salida producida por este método sobre objetos tipo **ValoresArray** y **ValoresArbol** respectivamente:

```
 Valores:
[X3 - 0]  [X4 - 0] = 0.2
[X3 - 0]  [X4 - 1] = 0.8
[X3 - 1]  [X4 - 0] = 0.6
[X3 - 1]  [X4 - 1] = 0.4
```

```
 Valores:
X11 : 0
   X21 : 0
      X31 : 0
          = 0.020000000000000004
      X31 : 1
          = 0.18000000000000002
   X21 : 1
      X31 : 0
          = 0.010000000000000002
      X31 : 1
          = 0.09000000000000001
   X21 : 2
      X31 : 0
          = 0.06999999999999999
      X31 : 1
          = 0.63
X11 : 1
   X21 : 0
      X31 : 0
          = 0.48
      X31 : 1
          = 0.32000000000000006
   X21 : 1
```

```
    X31 : 0
        = 0.12
    X31 : 1
        = 0.08000000000000002
X21 : 2
    X31 : 0
        = 0.0
    X31 : 1
        = 0.0
```

- **obtenerValor**: devuelve el valor correspondiente a la asignación pasada como argumento.
- **obtenerValores**: devuelve una lista con todos los valores almacenados en el potencial.
- **obtenerVariables**: devuelve la lista de variables que definen el dominio del conjunto de valores.
- **combinar**: recibe como argumento otro objeto de la clase **Potencial** y devuelve el conjunto de valores resultante de la combinación. El esquema de funcionamiento de este método podría ser el siguiente:
  - si el objeto sobre el que se hace la llamada es de tipo **ValoresArray**, entonces:
    - si el objeto pasado como argumento es tipo **ValoresArbol** se convierte a tipo **ValoresArbol**.
    - el objeto pasado como argumento se usa directamente.
    - se usa el método auxiliar de la clase **ValoresArray** para combinar dos objetos de tipo **ValoresArray** (el objeto sobre el que se hizo la llamada y el objeto pasado como argumento, tal o cual o tras su conversión) y genera un objeto del mismo tipo.
  - si el objeto sobre el que se hace la llamada es de tipo **ValoresArbol** entonces
    - si el objeto pasado como argumento es de tipo **ValoresArray** se convierte a tipo **ValoresArbol**.
    - si es de tipo **ValoresArbol** se usa directamente.
    - se realiza la combinación mediante un método auxiliar de la clase **ValoresArbol** que combina dos objetos de tipo **ValoresArbol** y genera un resultado de este mismo tipo.

    En definitiva, el método c**combinar** implementado en la clase **Valores** termina usando métodos de las clases específicas **ValoresArray** y **ValoresArbol**.
- **restringir**: recibe como argumento una variable y un valor de la misma y devuelve el conjunto de valores consistente con el valor de la variable (solo contiene aquellos valores asociados a asignaciones consistentes con el valor de la variable pasado como argumento).
- **convertir**: convierte la colección de valores de un tipo a otro. Si se aplica sobre un objeto de la clase **ValoresArray** devuelve un objeto de tipo **ValoresArbol**. En caso de aplicarse sobre un objeto **ValoresArbol** el resultado será un objeto **ValoresArray**.

A modo de ejemplo, consideremos dos potenciales $P(X_1, X_2)$ y $P(X_2, X_3)$, siendo todas las variables binarias. Los valores asociados al primero son: $\{0.3, 0.7, 0.6, 0.4\}$ y al segundo $\{0.9, 0.1, 1, 0\}$. Cada potencial se compone de un dominio y un dato de tipo **Valores** y asumimos que los objetos están referenciados por variables llamadas *potencial*1 y *potencial*2 respectivamente. Al hacer la siguiente llamada:

```
1 potencial1.combinar(potencial2)
```

se desencadena el siguiente proceso:

- la llamada se traslada a los objetos de la clase **Valores** de cada objeto, mediante la sentencia:

  ```
  1 valores.combinar(otro.valores)
  ```

  donde **valores** sería el dato miembro de **potencial1** y **otro.valores** el correspondiente a **potencial2**. En el método **combinar** de la clase **Valores** se examinan los tipos de estos objetos y se podrían dar los siguientes casos:

  - **this** (el objeto sobre el que se hace la llamada) y el objeto pasado como argumento (suponemos se llama **otro**) son ambos de tipo **ValoresArray**. En este caso se ejecutaría la siguiente sentencia:

    ```
    1 this.combinarArrayArray(otro)
    ```

  - **this** es de tipo **ValoresArray** y **otro** es de tipo **ValoresArbol**. En este caso es necesario hacer la conversión del argumento y posteriormente la combinación:

    ```
    1 val otroFinal=otro.convertir
    2 this.combinarArrayArray(otroFinal)
    ```

  - **this** es de tipo **ValoresArbol** y **otro** de tipo **ValoresArray**. En este caso el argumento ha de convertirse y después se produce la combinación:

    ```
    1 val otroFinal=otro.convertir
    2 this.combinarArbolArbol(otroFinal)
    ```

  - si ambos son de tipo **ValoresArbol** se realiza la combinación directamente:

    ```
    1 this.combinarArbolArbol(otro)
    ```

La detección del tipo concreto de cada objeto puede hacerse usando las facilidades que presenta **Scala** a tal efecto.

6.1 ValoresArray

Es un tipo específico de **Valores**. Todos los objetos de este tipo constan de un dominio y de un conjunto de datos (cuyo tipo puede ser una lista de valores). Los métodos de interés se han definido al hablar de la clase **Valores**. Tened en cuenta que algunos podrían implementarse en la clase base y otros en las clases derivadas.

Se hacen aquí algunos comentarios sobre la forma de implementar el método **combinarArrayArray** mencionado anteriormente. Se denota por **this** el objeto sobre el que se hará la llamada (**valores1** en el ejemplo) y por **otro** el objeto pasado como argumento (**valores2** en el ejemplo):

```
1 // Sentencia de ejemplo de uso de la operacion
2 valores1.combinar(valores2)
```

Imaginemos que el dominio de **valores1** es $\{X_1, X_2\}$ y el de **valores2** $\{X_2, X_3\}$. Por simplicidad supongamos que todas las variables son binarias (2 valores). Las listas de valores podrían ser: $\{0.3, 0.7, 0.6, 0.4\}$ y $\{0.9, 0.1, 1, 0\}$ respectivamente. El funcionamiento del método puede describirse de la siguiente forma:

- se genera el dominio del conjunto de valores resultante (puede aplicarse el operador + definido en la clase **Dominio**); lo designamos como **dominioFinal**.

  ```
  1 val dominioFinal = dominio + otro.dominio
  ```

- se recorren todos los índices válidos de **dominioFinal**. El dominio contiene las variables $\{X_1, X_2, X_3\}$ y los índices irán desde 0 hasta **maximoIndice** (8), excluyendo este último valor y:

  - se obtiene la asignación de valores en el dominio resultante que se corresponde con el valor de índice considerado (**asignacionFinal**). En la primera iteración contendrá el valor 0 para todas las variables; en la última el valor 1).

    ```
    1 val asignacionFinal = Asignacion(dominioFinal, indice)
    ```

  - se usa la proyección de **asignacionFinal** en el dominio de **this** (será una asignación definida únicamente sobre $\{X_1, X_2\}$).

    ```
    1 val asignacionThis = Asignacion.proyectar(asignacionFinal, dominio)
    ```

  - se proyecta **asignacionFinal** en el dominio **otro** (el resultado será una asignación definida en $\{X_2, X_3\}$).

    ```
    1 val asignacionOtro = Asignacion.proyectar(asignacionFinal, otro.dominio)
    ```

  - se obtienen los valores correspondientes a estas asignaciones en ambos objetos (en la primera iteración serán 0.1 y 0.9, por ejemplo).

```
val producto = obtenerValor(asignacionThis) * otro.obtenerValor(asignacionOtro)
```

- la lista de valores generada de esta forma se usa para crear un objeto nuevo (tipo **ValoresArray**) a partir del dominio final y de la lista de valores. El resultado tiene $\{X_1, X_2, X_3\}$ como dominio y 0.27, 0.03, 0.7, 0, 0.54, 0.06, 0.4 y 0 como valores.

Observad que la iteración sobre el conjunto de índices posibles debe hacerse usando las características propias de la programación funcional.

Con respecto a la restricción (operación **restringir**), supongamos el conjunto de valores llamado **valores1** sobre el que se desea hacer la restricción sobre $X_1 = 0$. La llamada al método se haría de la siguiente forma:

```
valores1.restringir(X1,0)
```

asumiendo que $X1$ es un objeto de tipo **Variable**. El funcionamiento del método podría esquematizarse de la siguiente forma:

- se genera un dominio a partir de **this**, eliminando la variable pasada como argumento (en el ejemplo visto se obtendría un dominio definido sobre $\{X_2, X_3\}$; se denominará **dominioFinal**.

```
val dominioFinal = dominio - variable
```

- se recorren todos los posibles índice de este dominio. Para cada índice:
  - se genera una asignación para este índice (**asignacionFinal**).

    ```
    val asignacionFinal = Asignacion.apply(dominioFinal, indice)
    ```

  - se genera una asignación sobre el dominio de **this** a partir de **asignacionFinal**, agregando la variable y el valor pasados como argumento (se obtendría **asignacionCompleta**).

    ```
    val asignacionCompleta=asignacionFinal + (variable, estado)
    ```

  - se crea una nueva asignacion con las variables del dominio para mantener el orden de las variables y se proyecta sobre ella la asignacion completa del paso anterior

    ```
    val asignacionOrdenada=Asignacion.proyectar(asignacionCompleta, dominio)
    ```

  - se obtiene el valor correspondiente a **asignacionOrdenada**.

```
val valor = obtenerValor(asignacionOrdenada)
```

- los valores obtenidos de esta iteración (realizada de la forma usual en programación funcional) se usan para crear un objeto de tipo **ValoresArray** a partir de **dominioFinal** y de los valores obtenidos.

En el ejemplo del potencial visto antes los valores obtenidos serían: $\{0.3, 0.7\}$.

6.2 ValoresArbol

Es un tipo específico de **Valores**. Todos los objetos de este tipo constan de un dominio y de un nodo llamado **raiz** (su tipo será **Nodo**, tipo genérico para los nodos del árbol). Esto hace que las operaciones en este tipo se deleguen sobre la raíz. Por ejemplo:

- **toString**: se llama al método **toString** de la clase base (**Valores**, responsable de mostrar el dominio) y se llama a continuación al método **toString** sobre el dato miembro **raiz**.
- en este caso, el método **combinar** genera el dominio final (agregando los dominios de **this** y del objeto pasado como argumento) y genera un objeto de tipo **ValoresArbol**:

```
// Se genera el dominio resultante
val dominioFinal = dominio + otro.dominio

// Se llama al metodo de combinacion sobre ambas raices
ValoresArbol(dominioFinal, raiz.combinar(otro.raiz))
```

- esta misma idea se usará para implementar el método **restringir**:

```
val dominioFinal = dominio - variable

// Se llama al metodo de restriccion sobre la raiz
ValoresArbol(dominioFinal, raiz.restringir(variable, estado))
```

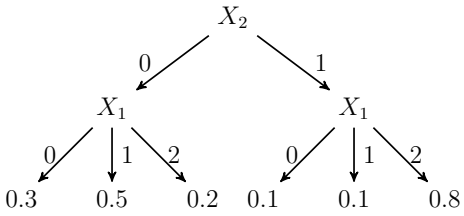
Se describen a continuación las clases necesarias para definir los árboles de valores: quedan definidos por un dominio y un nodo que hace de raíz del árbol asociado. Se indica a continuación la forma de implementar los tipos de nodos.

6.2.1 Nodo, NodoHoja y NodoVariable

La clase **Nodo** es la clase genérica para la construcción de árboles. Cada nodo tiene un dato miembro, tipo variable, que define el **nivel** en que se encuentre (su uso se explicará más adelante). La clase **Nodo** se usará como tipo básico para dos tipos de nodos: **NodoVariable** y **NodoHoja**. Los primeros sirven para representar nodos asociados a variables, que contendrán tantos hijos (ramas) como estados tenga la variable. Por su parte, los nodos hoja son terminales y almacenan únicamente un valor.

La clase **Nodo** debe definir, de forma directa o dejando como abstractas, las siguientes operaciones:

- **obtenerValores**: devuelve una lista con todos los valores almacenados en el objeto: un valor único en el caso de nodos hojas y una lista en el caso de nodos variable (el de todos los nodos hoja por debajo del mismo). Por ejemplo, la llamada a este método sobre el nodo raíz del árbol de la figura:

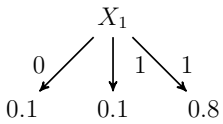


  producirá una lista con los valores 0.3, 0.5, 0.2, 0.1, 0.1 y 0.8.

- **obtenerHijo**: devuelve el nodo hijo con el índice indicado (el mismo nodo en el caso de nodo hoja). Imaginemos que **raiz** es la referencia al nodo raíz del árbol de la figura anterior. La llamada

```
raiz.obtenerHijo(1)
```

  devolverá el árbol correspondiente a $X_2 = 1$ (rama de la derecha):



- **obtenerValor**: obtiene el valor correspondiente a la asignación pasada como argumento.

- **combinar** y **restringir** se explicarán de forma específica para cada tipo de nodo.

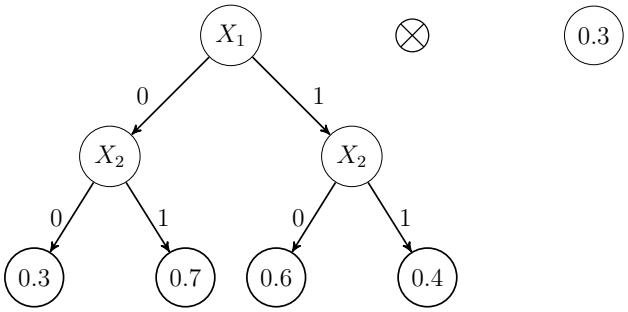
La combinación se debe hacer tal y como se indica a continuación (como antes, se asume que **this** alude al objeto sobre el que se hace la llamada y **otro** es el nodo pasado como argumento). Se consideran los casos posibles:

- **this** y **otro** referencian a nodos hoja. El resultado será un nuevo nodo hoja con valor el obtenido al multiplicar los valores almacenados en **this** y **otro**.

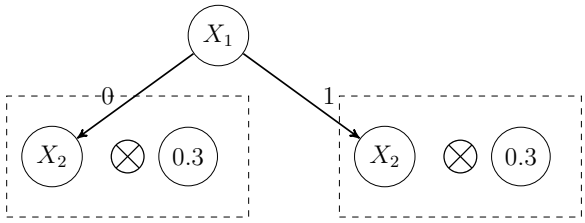
- **this** es nodo hoja y **otro** es nodo variable. En este caso se produce una llamada a combinar de la forma:

```
otro.combinar(this)
```

  Esta situación puede visualizarse en el siguiente gráfico, donde **otro** estaría referenciando al nodo raíz del árbol de la izquierda (en esta representación se han usado círculos para representar que se trata de nodos del árbol) y **this** referencia el nodo hoja situado a la derecha y con valor 0.3:



En esta situación la operación se propagaría por los nodos hijos del árbol con X_1 como raíz:

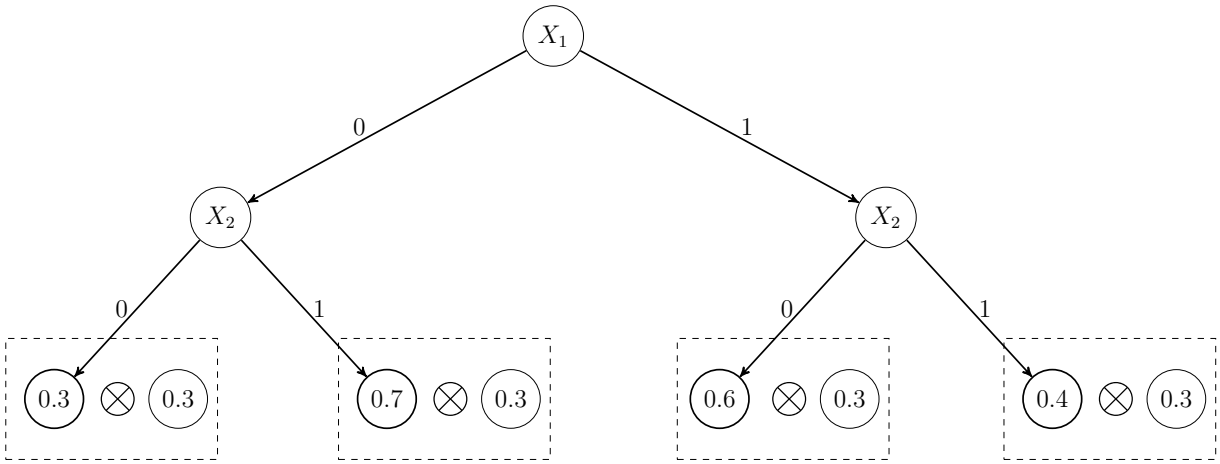


La imagen representa que el resultado será un nuevo nodo variable definido sobre X_1; sus hijos serán el resultado de realizar la combinación de los árboles hijos con el nodo valor para 0.3.

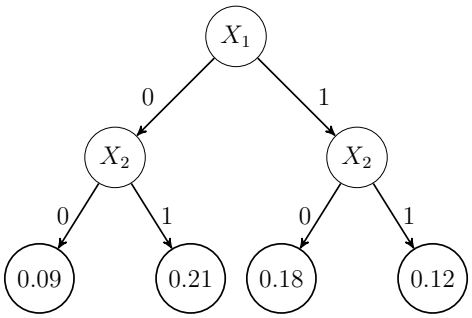
- en esta situación se trataría ahora de combinaciones en que la llamada se produce sobre un nodo variable y el argumento es un nodo hoja. La operación a realizar ahora sería:

```
hijo(indice).combinar(otro)
```

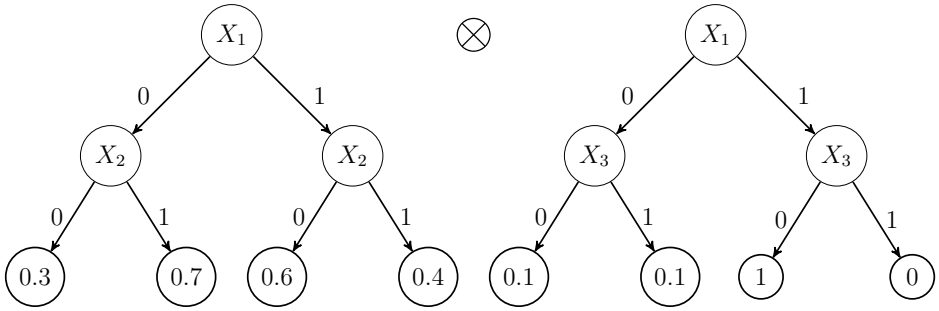
y la operación se propaga de nuevo hacia los hijos:



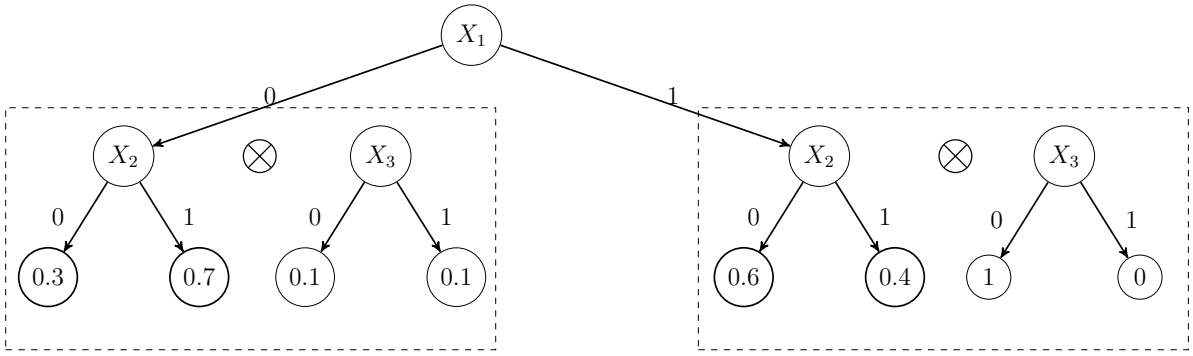
- ahora todas las operaciones pendientes operan únicamente sobre nodos hoja y se resolverían como se ha indicado antes, produciendo como resultado:



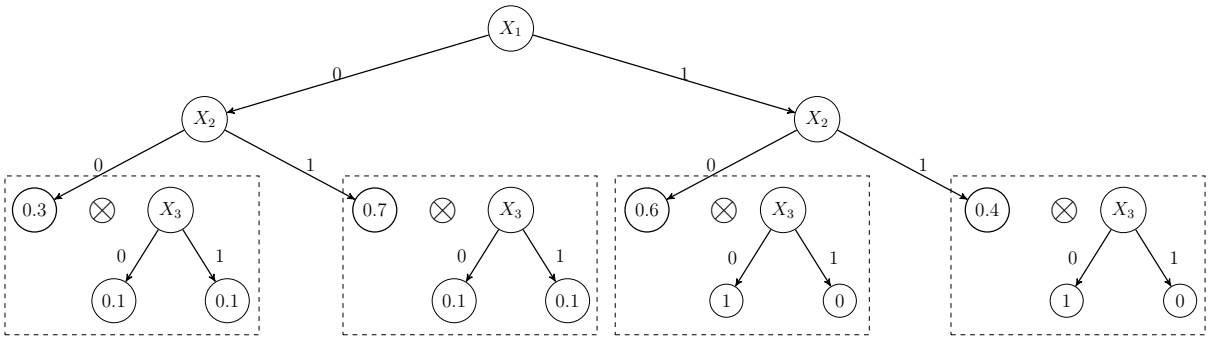
- queda por analizar el caso en que ambos nodos sean tipo variable. Sería la situación mostrada en la siguiente figura:



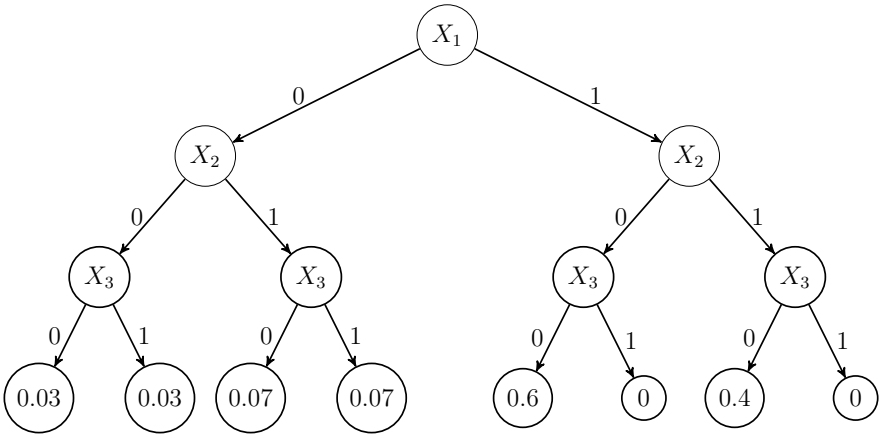
el árbol de la izquierda es aquel sobre el que se hace la llamada. La operación se propaga a los nodos hoja, pero es necesario modificar el árbol de la derecha (pasado como argumento). Al propagar la operación sobre el nodo variable asociado a $X_1 = 0$ se ha fijado el valor de esta variable. La operación con el árbol pasado como argumento requiere que este se restrinja (la operación de restricción se explica después) para trabajar únicamente con la parte del árbol consistente con $X_1 = 0$:



- Se aprecia en la figura que cada nodo hijo trabajará con la parte del árbol pasado como argumento consistente con el valor de X_1. Volvemos a estar en el caso de operaciones sobre nodos tipo variable y la operación seguirá propagándose hacia los hijos.



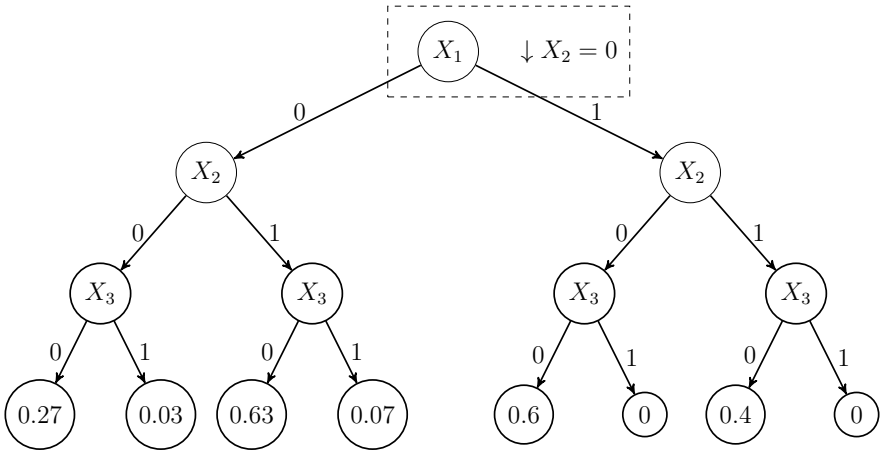
- En este momento ya se trata de operaciones entre nodos hoja y nodos variable, ya explicadas con anterioridad. El resultado final de la combinación será:



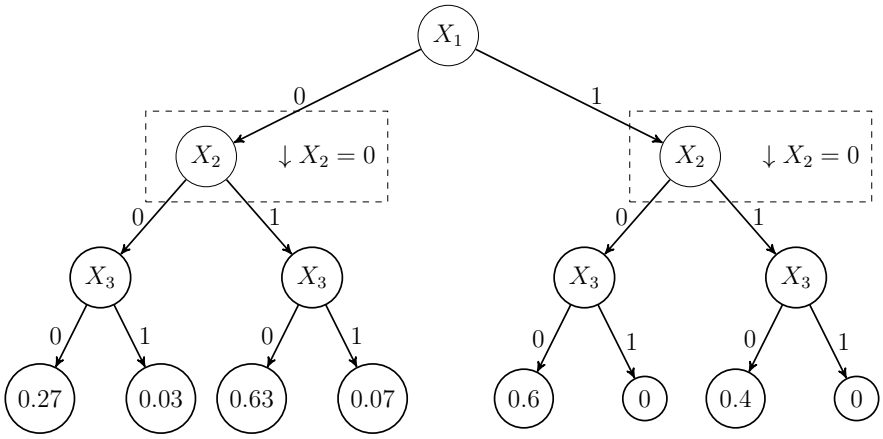
Para la implementación de la operación de restricción se seguirán las siguientes indicaciones:

- si **this** es un nodo hoja, se devuelve un nuevo nodo hoja almacenando en valor almacenado en **this**.
- si **this** es un nodo variable y su variable coincide con la variable sobre la que se restringe, basta con devolver como resultado el hijo asociado al valor pasado como argumento.
- en caso contrario (**this** no se corresponde con la variable por la que se restringe) y es necesario propagar la operación de restricción sobre los nodos hijos. El resultado de operar con cada hijo se convertirá en hijo del nodo resultante (definido sobre la misma variable que **this**).

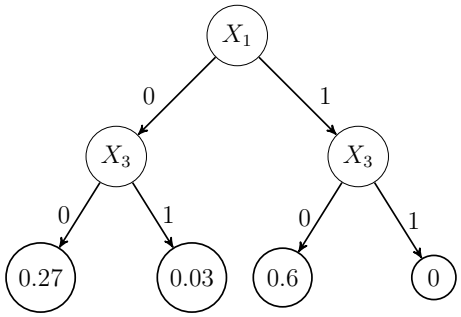
Un ejemplo de aplicación de operación se muestra en el gráfico siguiente (se trata de restringir el árbol mostrado sobre para quedarse con la parte consistente con $X_2 = 0$. El recuadro en línea discontinua muestra el nodo sobre el que se realiza la operación en cada momento:



El nodo sobre el que se aplica la operación está asociado a X_1 y la restricción se basa en X_2. Por esta razón se crearía un nodo resultado de tipo variable asociado a X_1. Sus hijo serán el resultado de las nuevas llamadas, tal y como se indica en el gráfico:



Ahora si hay coincidencia entre la variable de la restricción X_2 y la variable de los nodos con los que se trabaja. Ya no hay más llamadas al método y basta con devolver los hijos de la izquierda, asociados al valor $X_2 = 0$. El resultado final será un nuevo objeto:



6.2.2 Construcción de objetos de tipo ValoresArbol

Mención especial necesita el método necesario para construir objetos de esta clase. Lo ideal sería poder crear objetos sin usar el operador **new**, a partir de un objeto de la clase **Dominio** y una lista de valores (es decir, podría ser el método **apply** de un objeto compañero a la clase).

El método de construcción se basa en la existencia de un método auxiliar recursivo que va realizando la construcción. Necesita como argumentos:

- **indice**: indice la variable del dominio que se está considerando en el proceso de construcción.
- **asignacion**: indica la asignación de valores de las variables usadas para llegar hasta el nodo en construcción

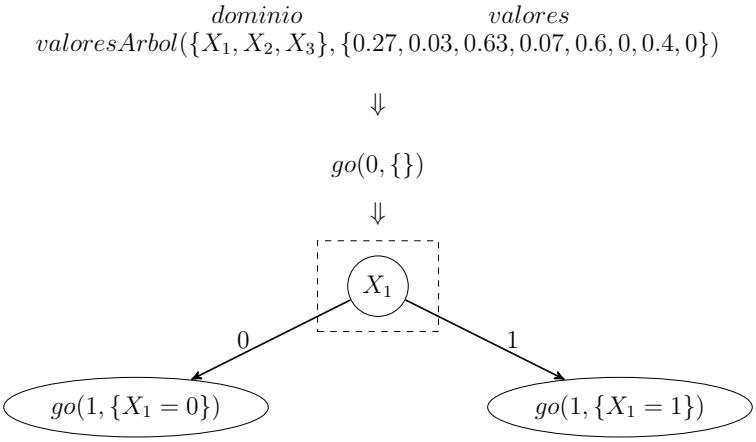
El resultado del método auxiliar es un objeto de la clase **Nodo**, que se explicará a continuación. El proceso entero de construcción se desencadena con una llamada a este método auxiliar (supongamos se denomina **go**) de la forma siguiente (también se indica al final la forma en que se construiría el objeto de tipo **ValoresArbol** con el resultado de la llamada al método auxiliar):

```
1 // Se construye el nodo raiz del objeto a construir
2 // El proceso se desencadena con indice=0, para considerar
3 // la primera variable y una asignacion vacia
4 val nodoRaiz = go(0, Asignacion(Dominio(List())))
5
6 // Ahora se puede construir el objeto de tipo ValoresArbol
7 new ValoresArbol(dominio, nodoRaiz)
```

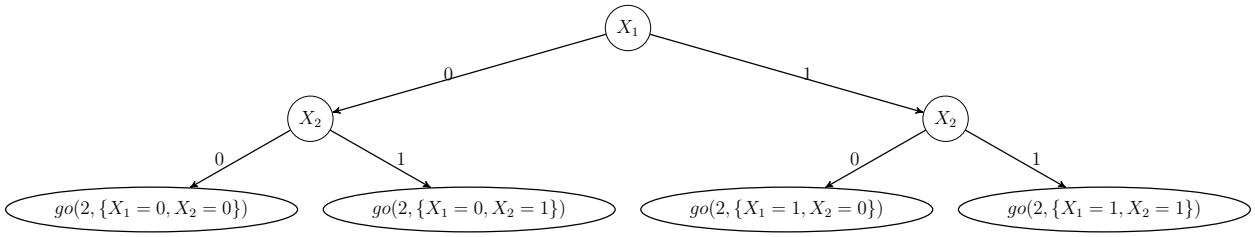
El trabajo del método auxiliar **go** puede describirse de la siguiente forma:

- se obtiene la variable correspondiente al índice pasado como argumento, accediendo al dominio (es un argumento del método de construcción (**apply**) en que se usa este método auxiliar).
- se crea un nodo tipo **NodoVariable** definido sobre la variable seleccionado en el paso anterior.
- si no es la última variable del dominio es necesario iterar (al modo de programación funcional) sobre los estados de la variable y generar nuevas llamadas al método auxiliar, incrementando el valor del índice y agregando a la asignación la variable considerada y el estado correspondiente. El resultado de cada nueva llamada se convierte en un hijo del nodo creado en el paso anterior.
- si es la última variable hay que iterar también sobre los valores de la variable, pero no se generan nuevas llamadas recursivas a **go**. En este caso para cada valor de la variable se crea una asignación nueva agregando la variable considerada y el valor que corresponde a la iteración actual (**asignacionHoja**) y se crea un nuevo objeto de tipo **NodoHoja**. El valor usado en la construcción del nodo hoja se selecciona de la lista de valores pasada a **apply** con el índice almacenado en la asignación (usando $asignacionHoja.calcularIndice$).
- el método auxiliar finaliza devolviendo el nodo construido.

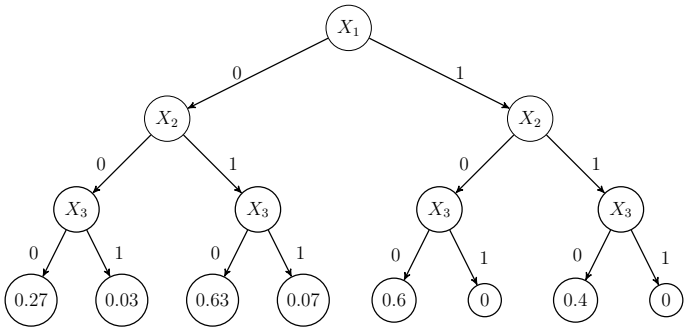
El proceso de construcción puede ilustrarse con las siguientes figuras:



Se aprecia que la llamada al método de construcción se produce con un objeto de la clase **Dominio** (primer agumento) y una lista de valores como segundo argumento. Esta llamada se traduce en lanzar el método auxiliar **go** con índice 0 y una asignación vacía. Este método, por tanto, trabaja con la primera variable X_1 (variable en posición 0 en el dominio). Al no tratarse de la última variable del dominio genera un nodo variable para X_1 (en la cada con línea discontinua de la figura). Sus hijos serán lo que devuelvan las llamadas a **go** que se representan sobre los hijos. Estas llamadas darán lugar a la siguiente situación:



Estas nuevas llamadas se producen al tratar con X_2, una variable que no ocupa la última posición en el dominio. Estas llamadas tratarán ahora con X_3, la última variable del dominio. En este caso ya no se producen más llamadas recursivas y las asignaciones se usarán para acceder a la posición adecuada de la lista de valores:



7 Clases Potencial, PotencialArray y PotencialArboles

Son las clases de más alto nivel. La primera de ellas ofrece el tipo genérico que se usará para la gestión de potenciales. Todos los potenciales se caracterizan por un **dominio** y un almacén

de valores (objeto de la clase **Valores**). La funcionalidad que que deben ofrecer los potenciales se describe a continuación:

- **toString**: muestra por pantalla el contenido de un objeto. La forma de visualización dependerá del tipo de potencial deque se trate. Se muestran a continuación un ejemplo de visualización de un potencial tipo array y otro potencial tipo árbol. En el primer caso es un potencial definido sobre las variables binarias X_3 y X_4, ambas binarias. En el caso del árbol el potencial se define sobre las variables X_{11}, X_{21} y X_{31}, con 2, 3 y 2 estados respectivamente.

  La visualización tipo árbol requiere que los nodos del árbol dispongan de un dato miembro (llamado **nivel**) que permite asociar un número de espacios en blanco asociado a cada nivel para que la visualización sea parecida a la indicada. Se recomienda implementar en los nodos un método (**asignarNivel**) que asigne los niveles y que se llamará antes de proceder a recorrer el árbol para mostrarlo por pantalla. Se tratará de un método recursivo que recibe como argumento el valor de índice a asignar y se propaga por los nodos de la red asignando el valor de nivel que corresponda. El caso base será, obviamente, el producido al alcanzarse nodos hoja, situación en la que no se generarán más llamadas.

- **combinar**: este método se basa en delegar la combinación en el dato miembro **valores**. Posteriormente, se crea un objeto concreto de tipo **PotencialArray** o **PotencialArbol** según sea el tipo del objeto sobre el que se hizo la llamada.

- **restringir**: el esquema de funcionamiento es similar al de **combinar**. La operación se delega sobre el dato miembro **valores** y al final se genera un nuevo potencial con el tipo concreto del objeto sobre el que se hizo la llamada.

- **obtenerValores**: devuelve la lista de valores del potencial. Como en los casos anteriores, esta operación se delega en el dato miembro **valores** (y permite que el método se implemente de forma completa en la clase **Potencial**).

- **convertir**: produce la conversión entre los dos tipos de potencial. Si el objeto sobre el que se hace la llamada es de tipo **PotencialArray** el resultado será de tipo **PotencialArbol** y viceversa.

8 Observaciones

Al igual que en prácticas anteriores el código implementado debe superar un determinado conjunto de pruebas que garanticen su correcto funcionamiento, aunque no hace falta en este caso que se use ninguna librería de casos de prueba (basta con aportar un objeto que extienda de **App** y donde se incorporen las sentencias necesarias para probar la funcionalidad requerida).

9 Material a entregar

Al final de la realización de la práctica se entregará un archivo comprimido con el contenido completo de la práctica, tal y como se integra en el proyecto con el entorno de desarrollo que hayáis usado. Se incluirá también un pequeño documento indicando el entorno de desarrollo y

una breve valoración de la práctica (si los conceptos vistos son novedosos, si os ha parecido de interés, problemas encontrados, etc) en tres o cuatro líneas.

La fecha límite de entrega será el 20 de Junio de 2018. La entrega se hará en PRADO pero recordar que esta práctica se defiende en persona. Las citas para la defensa se concertarán avisando por correo a mgomez@decsai.ugr.es.