Aprendizaje automático en juegos simples

Realizado por:
Javier Gil Blázquez, javgilbla, javicraft14@gmail.com
Rafael Díaz Cárabe, rafdiacar1, rafadc11@gmail.com

Temática elegida: Aprendizaje automático

Hemos elegido resolver la implementación de un sistema de aprendizaje automático aplicado a juegos simples como el ***Tres en raya*** y el ***Juego del Oso***.

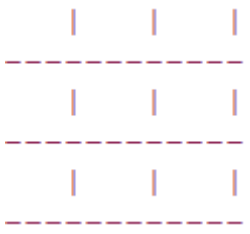
Para ello necesitamos ser capaces de generar un código que nos permite jugar libremente a ambos juego respetando sus reglas y sistema de puntuaciones, y un algoritmo que permita jugar contra la máquina.

Primera parte. Implementación de ambos juegos.

Para ambos escenarios definiremos un tipo **Cuadrícula** el cual será la representación gráfica de nuestro juego.

```
type Cuadricula = Matriz Char
```

Esta cuadrícula será imprimida por consola siguiendo el formato del siguiente código.



```
escribeCuadricula :: [String] -> String
escribeCuadricula [] = []
escribeCuadricula (xs:xss) = ((escribeFila xs)++"\n"++guiones++"\n")++(escribeCuadricula xss)
    where guiones = escribeGuiones (length xs)

escribeFila :: [Char] -> String
escribeFila [] = []
escribeFila (x:xs) = " "++(x:" |")++(escribeFila xs)

escribeGuiones :: Int -> String
escribeGuiones n
    | n == 0 = []
    | otherwise = "----"++(escribeGuiones (n-1))
```

Según los jugadores vayan colocando fichas, estos tableros de juego se rellenaran con sus correspondientes fichas, siguiendo el ejemplo del tres en raya:

```
-Le toca al jugador
-Para escoger casilla recuerda que los n\243meros que puedes escoger oscilan entre 0 y 2.
-Primero indica la fila: 1
-Ahora indica la columna: 1

 X |   | O |
-----------
   | X |   |
-----------
   |   |   |
-----------
```

Para lo cuál se implementa un código de petición por consola que permite al jugador introducir los datos de la posición en la que desea jugar una ficha:

```haskell
juegoMedio :: Cuadricula -> Int -> IO()
juegoMedio c j = do
    putStrLn "Estado del juego:\n"
    representaCuadricula c
    putStrLn $ "-Le toca al jugador " ++ (show j)
    let rangosC = bounds c
    let par1 = fst rangosC
    let par2 = snd rangosC
    let menor = fst par1
    let mayor = snd par2
    putStr "-Para escoger casilla recuerda que los números que puedes escoger oscilan entre "
    putStrLn $ (show menor) ++ " y " ++ (show mayor) ++ "."
    (fil,col) <- revisaIn (menor,mayor)
    let v = devuelveChar j
    cn <- jugada c (fil,col) v
    gestionaTurno cn j
```

```haskell
revisaIn :: (Int,Int) -> IO (Int,Int)
revisaIn (i,j) = do
    f <- leeDigito "-Primero indica la fila: "
    c <- leeDigito "-Ahora indica la columna: "
    if (f>=i && f<=j) && (c>=i && c<=j)
        then return (f, c)
        else do
            putStrLn "¡Fila o columna fuera de la cuadrícula. Vuelva a escoger!"
            revisaIn (i,j)
```

Una vez terminada una partida tendremos dos casos posibles, el caso de que uno de los jugadores gane, y el caso de quedar empate:

```
 X | O | O |
-----------
 X | X |   |
-----------
 X |   | O |
-----------

\255El jugador ha ganado!
```

```
 X | O | O |
-----------
 O | X | X |
-----------
   | X | O |
-----------

-Le toca al jugador
-Para escoger casilla recuerda
-Primero indica la fila: 2
-Ahora indica la columna: 0
Empate...
```

Estas comprobaciones de victoria o empate se comprueban mediante las siguientes funciones:

```haskell
hay3EnRaya :: Cuadricula -> Bool
hay3EnRaya c = or [if x==3 then True else False | x<-lss]
    where fs = listaFilas c
          cs = listaColumnas c
          ds = diagonalesMatriz c
          fsx = [x | x<-fs,x=="XXX"]
          csx = [x | x<-cs,x=="XXX"]
          dsx = [x | x<-ds,x=="XXX"]
          fso = [x | x<-fs,x=="OOO"]
          cso = [x | x<-cs,x=="OOO"]
          dso = [x | x<-ds,x=="OOO"]
          ess = fsx++csx++dsx++fso++cso++dso
          lss = [length x | x<-ess]
```

Para no implementar todo el código auxiliar en módulo principal hemos generado un módulo auxiliar que recoge todas las funciones de generación de cuadrículas vacías y funciones de comprobación de datos las cuales nos permitirán conocer el estado de la cuadrícula y del juego:

```haskell
module Utiles
    (Matriz,
     matrizUnitaria,
     matrizNueva,
     listaFilas,
     listaColumnas,
     diagonalesMatriz,
     actualizaValor,
     valido,
     escribeCuadricula,
     traduceCadena
    ) where
```

Al igual que hicimos con el tipo Cuadrícula, generamos un tipo sinónimo ***Matriz*** para almacenar los estados de la partida sin necesitar una representación gráfica futura:

```haskell
type Matriz a = Array (Int,Int) a
```

Para la ejecución del código simplemente tendremos que cargar nuestro programa en un entorno que pueda ejecutar las extensiones de código de .hs de Haskell:

```
λ> :l 3enRaya.hs
[1 of 2] Compiling Utiles           ( Utiles.hs, interpreted )
[2 of 2] Compiling Main             ( 3enRaya.hs, interpreted )
Ok, two modules loaded.
Collecting type info for 2 module(s) ...
λ> main
Escoge. Modo 1 jugador o 2 jugadores. Para escoger simplemente pon el n\243mero
```

A continuación se nos irán mostrando una serie de instrucciones para continuar la ejecución:

Para empezar elegiremos cuantos jugadores queremos ser, en el caso de querer aplicar los algoritmos de aprendizaje automático, tendremos que jugar contra la máquina, esto se hará introduciendo un 1 por consola.

A continuación podemos empezar una partida nueva o cargar una ya guardada incluyendo la dirección de un fichero .txt con la guía de una cuadrícula previamente guardada.

Finalmente elegiremos la dificultad de la máquina y si queremos empezar primero o tras una jugada de la máquina.

```
Escoge. Modo 1 jugador o 2 jugadores. Para escoger simplemente pon el n\243mero1
\250Quieres empezar un juego nuevo o cargar una partida?
Escribe 'nuevo' o 'cargar' por favor
nuevo
Primero escoja una dificultad. Puede escoger entre simple(1) o complejo(2).1
Ahora escoja si quiere empezar usted o la m\240quina por favor.1
Estado del juego:


   |   |   |
------------
   |   |   |
------------
   |   |   |
------------

-Le toca al jugador
-Para escoger casilla recuerda que los n\243meros que puedes escoger oscilan entre 0 y 2.
-Primero indica la fila:
```

Una vez comenzado el tablero, seguiremos jugando colocando la posición de la cuadrícula en la que queremos colocar una ficha y tras cada jugada nuestra o de la máquina se nos preguntará si queremos guardar o no la partida, en caso de no desear hacerlo, solo tendremos que pulsar enter.

```
-Le toca al jugador
-Para escoger casilla recuerda que los n\243meros que puedes escoger oscilan entre 0 y 2.
-Primero indica la fila: 1
-Ahora indica la columna: 1

   |   |   |
-----------
   | X |   |
-----------
   |   |   |
-----------

\250Desea guardar partida?
Si desea guardar partida escriba <<SI>> por favor

Estado del juego:


   |   |   |
-----------
   | X |   |
-----------
   |   |   |
-----------

-Le toca a la m\240quina

   |   | O |
-----------
   | X |   |
-----------
   |   |   |
-----------

\250Desea guardar partida?
Si desea guardar partida escriba <<SI>> por favor
```

Si seguimos jugando llegaremos a un estado de victoria de uno de los dos jugadores o de empate, como se ha mostrado al principio del documento.

Segunda parte. Implementación Aprendizaje Automático.

Para la implementación del aprendizaje automático usaremos el método minimax con poda básica por profundidad.

Este método consiste en generar un árbol de decisión que usará la máquina para ejecutar sus movimientos tras realizar una valoración a cada nodo del árbol. Esta valoración será calculada mediante la función de maximizar y minimizar, que nos permitirá ajustar cuál es el mejor movimiento posible.

Primero vamos a definir los tipos ***Árbol*** y ***Puntuación*** que usaremos como bases para tratar el algoritmo.

```
data Arbol a = N a [Arbol a]
```

```
type Puntuacion = Int
```

También propondremos una ***Profundidad Máxima de Búsqueda***, esta variable global servirá para determinar la profundidad máxima de creación del árbol de decisiones.
A mayor valor de la variable, mejor jugadas realizará la máquina, pero esto conlleva un coste temporal superior:

-Para el Tres en Raya.

```
profBus :: Int
profBus = 6
```

-Para el juego del OSO.

```
profBus :: Int
profBus = 14
```

Este algoritmo necesita una variedad de funciones para generar los árboles, puntuarlos y seleccionar finalmente la jugada mejor valorada:

-Para el Tres en Raya.

```
mejorMov :: Cuadricula -> Cuadricula
mejorMov = seleccion . maximiza . poda profBus . creaArbol
```

-Para el juego del OSO.

```
mejorMov :: (Cuadricula, (Int, Int)) -> (Cuadricula,(Int,Int),Char)
mejorMov = seleccion . maximiza . poda profBus . creaArbol
```

Nota: A partir de ahora nos centraremos en la implementación del algoritmo en el juego del OSO ya que es el más completo de los dos.

De derecha a izquierda tenemos:

- Una función para crear los árboles que vamos a usar para encontrar el mejor movimiento.
- La poda que haremos en el árbol principal y que dependerá de la profundidad prefijada.
- Las funciones de maximización y minimización típicas del algoritmo ***minimax***.
- La función de selección para encontrar el mejor árbol.

Función *creaArbol*.

Esta función genera el árbol de decisiones base usado en el resto de funciones necesarias para el algoritmo minimax. Recibe un par de Cuadrícula de juego y la puntuación actual que llevan ambos jugadores en el juego del OSO, y devuelve un Árbol de Cuadrícula de juego, con las puntuaciones actuales, la posición a jugar y la letra a jugar ‘O’ o ‘S’.

```
creaArbol :: (Cuadricula,(Int,Int)) -> Arbol (Cuadricula,(Int,Int),(Int,Int),Char)
```

Ejemplo en el juego del OSO

Funciones movimientos y auxiliar.

Estas funciones son necesarias para la creación del árbol:

```
movimientos :: Cuadricula -> Int -> [(Cuadricula,(Int,Int))]
```

La función movimientos, determina que jugadas son las que se usarán para crear el árbol según las casillas libres que queden en la cuadrícula de juego.

Debido a que este juego tiene dos opciones para realizar una jugada, ya sea colocar una 'O' o colocar una 'S', necesitamos un par de funciones auxiliares que determinen qué piensa la máquina que debería jugar tanto si misma como el otro jugador, estas dos funciones son idénticas, con la única diferencia de a quien se le suma la puntuación de realizar esa jugada:

```
auxiliarM :: Cuadricula -> (Int,Int) -> (Cuadricula,(Int,Int))
```

```
auxiliarH :: Cuadricula -> (Int,Int) -> (Cuadricula,(Int,Int))
```

Función poda.

Con esta función pretendemos eliminar todos los árboles generados que superen la profundidad máxima de busqueda.

```
poda :: Int -> Arbol a -> Arbol a
```

Ejemplo en el juego del OSO

Función maximiza y minimiza.

Estas dos funciones de estructura similar puntúan los movimientos disponibles de la máquina, mientras que maximiza valora las jugadas con coste -1, minimiza las valora con +1.

```
maximiza :: Arbol (Cuadricula,(Int,Int),(Int,Int),Char) -> Arbol (Puntuacion, Cuadricula,(Int,Int),Char)
```

Ejemplo en el juego del OSO

Función selección.

Esta función selecciona la jugada que mayor puntuación le de a la máquina.

```
seleccion :: Arbol (Puntuacion, Cuadricula,(Int,Int),Char) -> (Cuadricula,(Int,Int),Char)
```

Ejemplo en el juego del OSO