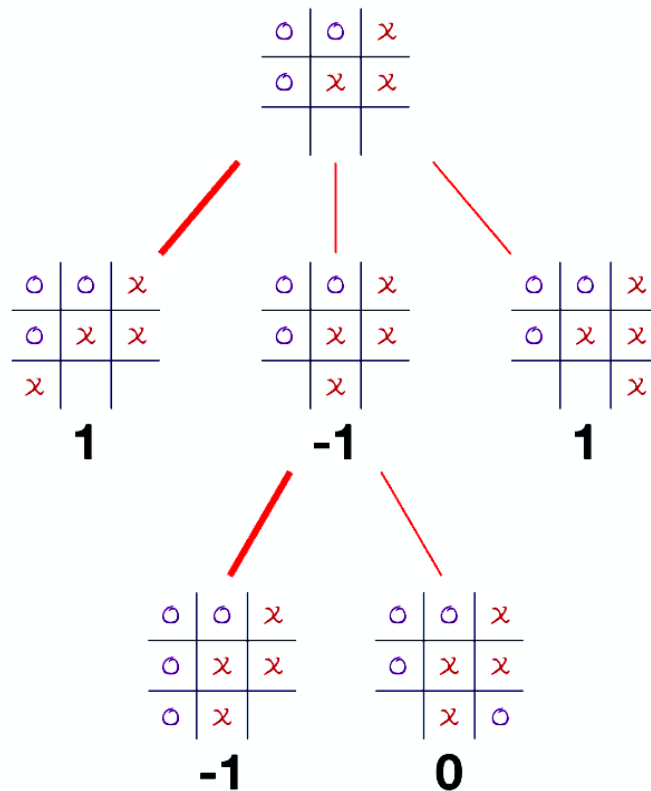


Modelos Avanzados de Computación

Curso 2024/2025

Trabajo final:

Tic Tac Toe & MiniMax



Autores:

Miguel Quiroga Campos

Alberto Aponte Araujo

Índice

Introducción.....	3
Módulos importados.....	4
Declaraciones type.....	5
Funciones.....	6
Resultados.....	16
Conclusión.....	18

Introducción

En esta memoria se presenta el desarrollo e implementación de un proyecto cuyo objetivo es crear un programa que simule el clásico juego Tic Tac Toe (también conocido como tres en raya) para jugar contra un agente inteligente. Este agente está diseñado utilizando el algoritmo Minimax, para conseguir tomar las decisiones óptimas.

El proyecto ha sido desarrollado íntegramente en el lenguaje de programación Haskell, cuyas cualidades no solo facilitaron la implementación del algoritmo, sino que también permitieron estructurar el código de forma elegante y modular, enfatizando la claridad y la expresividad.

El programa desarrollado se caracteriza principalmente por:


- Poder elegir jugar como 'X' o como 'O', permitiendo que el agente se adapte para jugar con la ficha contraria.
- Competir contra un agente que utiliza el algoritmo Minimax para analizar todos los posibles estados del juego y determinar el movimiento óptimo.

Su desarrollo se ha llevado a cabo en varias etapas, que incluyen el modelado del tablero de juego, la implementación del algoritmo Minimax, el diseño de la interfaz de usuario, y la gestión de turnos y resultados.

Este proyecto no solo busca ofrecer una experiencia de juego desafiante y entretenida, sino también ilustrar cómo los principios de la inteligencia artificial pueden integrarse con lenguajes funcionales para resolver problemas prácticos. Además, nos ha servido como una oportunidad educativa para profundizar en el análisis de algoritmos como Minimax y explorar el potencial de Haskell como herramienta para implementar soluciones complejas.

En las secciones posteriores de esta memoria, se describe en detalle la implementación de cada sección del código, y los resultados obtenidos.

Módulos importados



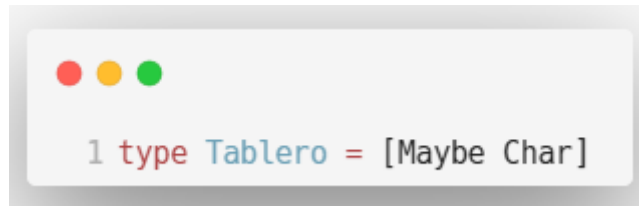
```
1 import Data.List (transpose, maximumBy, minimumBy)
2 import Data.Maybe (isNothing, isJust, fromJust)
3 import System.Process
```

El módulo **Data.List** se ha importado para aprovechar funciones que simplifican operaciones comunes en listas, fundamentales para la lógica del juego y el algoritmo Minimax. En particular, **transpose** permite reorganizar las filas del tablero en columnas, facilitando la comprobación de líneas verticales ganadoras. Por su parte, **maximumBy** y **minimumBy** son esenciales para el algoritmo Minimax, ya que permiten seleccionar el movimiento óptimo al comparar fácilmente sus puntuaciones asociadas.

El módulo **Data.Maybe** lo hemos usado para manejar casillas que pueden estar vacías, crucial para evitar errores en operaciones donde los datos no están garantizados. En concreto, **isNothing** y **isJust** se emplean para verificar si una casilla del tablero está vacía o contiene una ficha. Además, **fromJust** permite extraer el valor contenido en un tipo **Maybe** (en una casilla) cuando se sabe que no es **Nothing**, lo cual nos ha resultado muy útil para procesar resultados como el ganador del juego o realizar movimientos en el tablero.

Y por último, el módulo **System.Process**, imprescindible para poder programar la función **clear**, que nos permite limpiar la pantalla de la línea de comandos tras cada movimiento.

Declaraciones type

A screenshot of a code editor window with a light gray background and a title bar with three colored buttons (red, yellow, green). The code inside the editor is:

```
1 type Tablero = [Maybe Char]
```

```
1 type Tablero = [Maybe Char]
```

En este proyecto, sólo nos ha hecho falta definir un tipo, el tipo **Tablero**, como una lista de valores **Maybe Char**, para representar el estado del juego de forma clara y funcional. Este enfoque nos ha permitido modelar cada casilla del tablero como un valor que puede estar vacío (**Nothing**) o contener una ficha (**Just 'X'** o **Just 'O'**), haciendo el resto de la programación más intuitiva y fácil.

Funciones

```
1 -- Muestra el tablero
2 mostrarTablero :: Tablero -> IO ()
3 mostrarTablero tablero = do
4   putStrLn $ unlines $ map getFila [0, 3, 6]
5   where
6     getFila i = concat [getCasilla (tablero !! j) | j <- [i .. i + 2]]
7     getCasilla Nothing = " ."
8     getCasilla (Just c) = " " ++ [c] ++ " "
```

La función **mostrarTablero** se encarga de mostrar el estado actual del tablero por consola, de manera legible y visual para el usuario. Utilizamos la función **map** para aplicar **getFila** a cada índice de las filas del tablero (0, 3, 6), generando así las tres filas del juego.

Dentro de **getFila**, se recorre cada conjunto de tres posiciones consecutivas del tablero (cada fila) y se utiliza la función **getCasilla** para obtener la ficha de cada casilla. De forma que si una casilla está vacía (**Nothing**), se muestra un punto, y si contiene una ficha (**Just 'X'** o **Just 'O'**), se muestra el símbolo correspondiente separado por espacios.

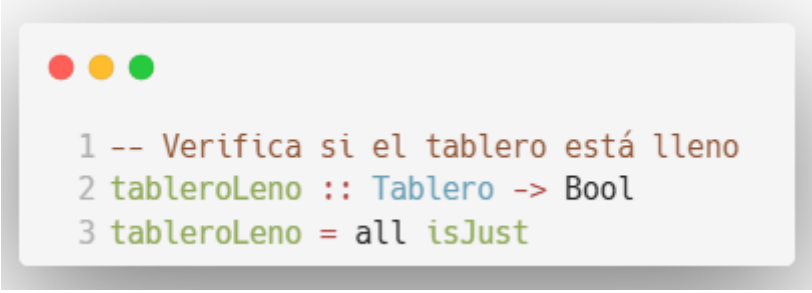
```

1 -- Verifica si hay un ganador
2 hayGanador :: Tablero -> Maybe Char
3 hayGanador tablero = foldl hayLinea Nothing lineasPosibles
4   where
5     lineasPosibles = filas ++ columnas ++ diagonales
6     filas = [[tablero !! i, tablero !! (i + 1), tablero !! (i + 2)] | i <- [0, 3, 6]]
7     columnas = transpose filas
8     diagonales = [[tablero !! 0, tablero !! 4, tablero !! 8], [tablero !! 2, tablero !! 4, tablero !! 6]]
9     hayLinea acc linea
10      | all (== Just 'X') linea = Just 'X'
11      | all (== Just 'O') linea = Just 'O'
12      | otherwise = acc

```

La función **hayGanador** tiene como objetivo verificar si hay un ganador en el juego, es decir, si alguna fila, columna o diagonal contiene tres fichas iguales. Utilizamos **foldl** para recorrer todas las líneas posibles del tablero (filas, columnas y diagonales) y comprobar si alguna de ellas está completamente llena con una sola ficha. Si se encuentra una línea con todas las marcas iguales, se devuelve **Just 'X'** o **Just 'O'** dependiendo del ganador; si no, sigue evaluando las siguientes líneas. Si no se encuentra ninguna línea ganadora devuelve **Nothing**.

Para generar las líneas posibles, se crean explícitamente las filas, columnas y diagonales del tablero. Las filas se generan con una lista intensional, mientras que las columnas se obtienen aplicando la función **transpose** sobre las filas. Las diagonales se definen manualmente, tomando los elementos correspondientes de las dos diagonales principales del tablero.



```
1 -- Verifica si el tablero está lleno
2 tableroLeno :: Tablero -> Bool
3 tableroLeno = all isJust
```

La función **tableroLeno** verifica si todas las casillas del tablero están ocupadas, es decir, si no quedan movimientos posibles.

Esta función utiliza la función de orden superior **all** junto con el predicado **isJust**. De esta forma, se comprueba si todos los elementos de la lista que representa el tablero son del tipo **Just**, lo que significa que cada casilla contiene una ficha (**'X'** o **'O'**) y, por lo tanto, el tablero está lleno.



```
1 -- Obtiene las casillas vacías  
2 casillasVacias :: Tablero -> [Int]  
3 casillasVacias tablero = [i | (i, Nothing) <- zip [0..] tablero]
```

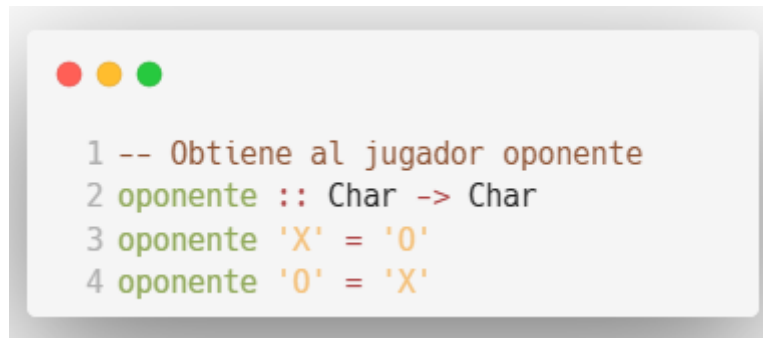
La función **casillasVacias** identifica las posiciones vacías en el tablero, devolviendo una lista con los índices de las casillas disponibles para realizar movimientos.

Para ello, hace uso de una lista intensional, en la cual se agrupan en pares (gracias a **zip**) un índice con cada posición del tablero y selecciona aquellos índices que en su par se encuentre el término **Nothing**.

```
1 -- Realiza un movimiento
2 mueve :: Tablero -> Int -> Char -> Maybe Tablero
3 mueve tablero casilla jugador
4   | casilla >= 0 && casilla < length tablero && isNothing (tablero !! casilla) =
   Just (take casilla tablero ++ [Just jugador] ++ drop (casilla + 1) tablero)
5   | otherwise = Nothing
```

La función **mueve** permite simular un movimiento en el tablero verificando si la posición especificada es válida (es decir, si está dentro de los límites del tablero y la casilla está vacía). Si estas condiciones se cumplen, se crea un nuevo tablero colocando la ficha del jugador ('X' o 'O') en la posición deseada.

La función logra obtener el tablero deseado dividiéndolo en dos partes: las casillas antes y después de la posición especificada, usando las funciones **take** y **drop**, y luego reconstruyendo el tablero con la ficha del jugador en la posición seleccionada. Si el movimiento es válido, la función devuelve el nuevo tablero envuelto en **Just**; en caso contrario, devuelve **Nothing**, indicando que el movimiento no es posible.



```
1 -- Obtiene al jugador oponente
2 oponente :: Char -> Char
3 oponente 'X' = 'O'
4 oponente 'O' = 'X'
```

La función **oponente** determina el jugador contrario al dado como entrada. Si el jugador es **'X'**, la función devuelve **'O'**, y viceversa.

Esto se logra mediante un patrón de correspondencia directa: se especifica que para el carácter **'X'** el resultado será **'O'** y para **'O'** será **'X'**. Este comportamiento garantiza que, dado un jugador actual, siempre se pueda identificar de manera clara y directa al oponente.

```

1 -- Minimax
2 minimax :: Tablero -> Char -> Char -> Int
3 minimax tablero jugador agente
4   | isJust ganador = case ganador of
5       Just 'X' -> if agente == 'X' then 1 else -1
6       Just 'O' -> if agente == 'O' then 1 else -1
7       _         -> 0
8   | tableroLeno tablero = 0
9   | jugador == agente = maximum $ map (\casilla -> minimax (fromJust (mueve tablero casilla jugador))
10  (oponente jugador) agente) (casillasVacias tablero)
11 | otherwise
12   = minimum $ map (\casilla -> minimax (fromJust (mueve tablero casilla jugador))
13  (oponente jugador) agente) (casillasVacias tablero)
14 where
15   ganador = hayGanador tablero

```

La función **minimax** es la implementación del algoritmo Minimax, utilizado para evaluar el estado del tablero y tomar las decisiones óptimas durante el juego. Este algoritmo considera todos los posibles movimientos futuros y evalúa sus resultados para maximizar las oportunidades de victoria del agente y minimizar las del oponente.

Como cualquier función recursiva, consta de:

- **Casos base:**
 - Si el juego tiene un ganador (**isJust ganador**), se devuelve una puntuación basada en quién ganó:
 - Si el ganador es el agente, se devuelve **1**.
 - Si el ganador es el oponente, se devuelve **-1**.
 - En caso de empate, se devuelve **0**.
 - Si el tablero está lleno (**tableroLeno**), se devuelve **0**, indicando un empate.
- **Casos recursivos:**
 - Si es el turno del **agente** (**jugador == agente**), la función calcula la puntuación máxima entre todos los movimientos posibles (**casillasVacias tablero**). Para cada casilla vacía, se simula un movimiento con **mueve**, generando un nuevo estado del tablero, que se le pasa recursivamente a **minimax**, alternando el turno entre el jugador actual y su oponente.
 - Si es el turno del **oponente**, se calcula la puntuación mínima, ya que el oponente (el humano) buscará minimizar las oportunidades del agente.

```
1 -- Obtiene el mejor movimiento para el agente
2 mejorMovimiento :: Tablero -> Char -> Int
3 mejorMovimiento tablero jugador = fst $ maximumBy comparaMovimientos movimientos
4   where
5     movimientos = [(casilla, minimax (fromJust (mueva tablero casilla jugador)) (oponente jugador) jugador) |
6       casilla <- casillasVacias tablero]
7     comparaMovimientos (_, puntuacion1) (_, puntuacion2) = compare puntuacion1 puntuacion2
```

La función **mejorMovimiento** se implementa para determinar la casilla óptima en el tablero donde el agente debería poner su próxima ficha, maximizando sus posibilidades de ganar, o minimizando las del oponente. Se construye utilizando una lista de pares **(casilla, puntuación)**, donde cada casilla vacía del tablero es evaluada mediante la función **minimax**, que calcula el valor asociado a realizar un movimiento en esa posición. Se utiliza **fromJust** para obtener el **Tablero** resultante de aplicar un movimiento al tablero, y se invoca a **minimax** con el turno del oponente.

El uso de **maximumBy** permite seleccionar el par (movimiento) con la mayor puntuación, comparando sus puntuaciones mediante la función **comparaMovimientos**. Esta implementación asegura que el agente siempre tome la decisión más favorable al analizar todos los movimientos posibles.

```

1 -- Bucle principal del juego
2 partida :: Tablero -> Char -> Char -> IO ()
3 partida tablero jugador agente = do
4     clear
5     mostrarTablero tablero
6     if isJust (hayGanador tablero) then do
7         let ganador = fromJust (hayGanador tablero)
8         if ganador == agente then
9             putStrLn "¡Ooops, el agente te ha ganado!"
10        else
11            putStrLn "¡Enhorabuena, has ganado!"
12    else if tableroLleno tablero then
13        putStrLn "¡Es un empate!"
14    else do
15        if jugador == agente then do
16            putStrLn "Turno del agente..."
17            let casilla = mejorMovimiento tablero jugador
18            putStrLn $ "El agente elige la casilla: " ++ show casilla
19            partida (fromJust (mueve tablero casilla jugador)) (oponente jugador) agente
20        else do
21            putStrLn "Tu turno (elige una casilla del 0 al 8):"
22            casilla <- readLn
23            case mueve tablero casilla jugador of
24                Just nuevoTablero -> partida nuevoTablero (oponente jugador) agente
25                Nothing -> do
26                    putStrLn "ERROR: Movimiento inválido."
27                    partida tablero jugador agente

```

La función **partida** implementa el bucle principal del juego, gestionando el flujo de la partida entre el jugador humano y el agente.

En cada turno, tras borrar (**clear**) el tablero previo, se muestra el actual con **mostrarTablero**, y luego se verifica si el juego ha terminado, ya sea porque hay un ganador (**hayGanador tablero**) o porque el tablero está lleno (**tableroLleno tablero**). Si existe un ganador, se muestra un mensaje indicando quién ganó, dependiendo de si fue el agente o el jugador. En caso de empate, se muestra un mensaje correspondiente.

Si el juego no ha terminado, se alterna entre el turno del agente y el turno del jugador humano. Cuando es el turno del agente, se calcula el mejor movimiento posible con **mejorMovimiento**, se actualiza el tablero simulando el movimiento, y se llama recursivamente a **partida** con el tablero actualizado y el turno del oponente. Cuando es el turno del jugador humano, se solicita una casilla por consola, y si el movimiento es válido (**mueve** devuelve **Just nuevoTablero**), se actualiza el tablero y se continúa la partida; de lo contrario, se muestra un mensaje de error y se repite el turno.

```
1 -- Inicia el juego, permitiendo al usuario elegir su jugador
2 main :: IO ()
3 main = do
4     putStrLn "¡Bienvenido al Tic Tac Toe!"
5     putStrLn "¿Quieres jugar como 'X' o 'O'?"
6     eleccion <- getLine
7     let jugador = if eleccion == "X" then 'X' else 'O'
8     let agente = if jugador == 'X' then 'O' else 'X'
9     let tablero = replicate 9 Nothing
10    partida tablero 'X' agente
11    putStrLn "Pulsa Enter para salir..."
12    _ <- getLine
13    putStrLn "Adios!"
```

La función **main** actúa como punto de inicio y fin del programa, permitiendo al usuario configurar y comenzar una partida, así como terminarla.

Primero, muestra un mensaje de bienvenida al jugador y solicita que elija con qué ficha quiere jugar ('X' o 'O'). La entrada del usuario se captura con **getLine**, y mediante una estructura condicional, se asignan las fichas: si el jugador elige 'X', el agente jugará como 'O', y viceversa.

A continuación, se inicializa el tablero como una lista de nueve elementos **Nothing**, representando un tablero vacío. Entonces, se llama a la función **partida**, que implementa el bucle principal del juego, comenzando siempre con 'X' como el primer jugador, de acuerdo con las reglas.

Al finalizar la partida, se espera hasta pulsar alguna tecla para cerrar el programa, pudiendo así visualizar tranquilamente el resultado final del juego.



```
1 -- Limpia la pantalla
2 clear :: IO ()
3 clear = do
4     system "cls"
5     return ()
```

La función **clear** se utiliza para limpiar la pantalla de la consola durante la ejecución del programa, justo antes de mostrar una actualización del tablero, proporcionando una experiencia de juego más ordenada y agradable. Esta función llama al comando de Windows **"cls"**, que es el comando para limpiar la consola. Acto seguido, devolvemos un valor vacío (**return ()**), para cumplir con **IO ()**.

Resultados

Partida 1

```
¡Bienvenido al Tic Tac Toe!  
¿Quieres jugar como 'X' o 'O'?  
X  
. . .  
. . .  
. . .  
  
Tu turno (elige una casilla del 0 al 8):  
4  
. . .  
. X .  
. . .  
  
Turno del agente...  
El agente elige la casilla: 8  
. . .  
. X .  
. . O  
  
Tu turno (elige una casilla del 0 al 8):  
5  
. . .  
. X X  
. . O  
  
Turno del agente...  
El agente elige la casilla: 3  
. . .  
O X X  
. . O  
  
Tu turno (elige una casilla del 0 al 8):  
2  
. . X  
O X X  
. . O  
  
Turno del agente...  
El agente elige la casilla: 6  
. . X  
O X X  
O . O  
  
Tu turno (elige una casilla del 0 al 8):  
7  
. . X  
O X X  
O X O  
  
Turno del agente...  
El agente elige la casilla: 0  
O . X  
O X X  
O X O  
  
¡Ooops, el agente te ha ganado!
```

Partida 2

```
¡Bienvenido al Tic Tac Toe!  
¿Quieres jugar como 'X' o 'O'?  
O  


|   |   |   |
|---|---|---|
| . | . | . |
| . | . | . |
| . | . | . |

  
Turno del agente...  
El agente elige la casilla: 8  


|   |   |   |
|---|---|---|
| . | . | . |
| . | . | . |
| . | . | X |

  
Tu turno (elige una casilla del 0 al 8):  
4  


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

  
Turno del agente...  
El agente elige la casilla: 7  


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

  
Tu turno (elige una casilla del 0 al 8):  
6  


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

  
Turno del agente...  
El agente elige la casilla: 2  


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

  
Tu turno (elige una casilla del 0 al 8):  
5  


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

  
Turno del agente...  
El agente elige la casilla: 3  


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

  
Tu turno (elige una casilla del 0 al 8):  
0  


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

  
Turno del agente...  
El agente elige la casilla: 1  


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

  
¡Es un empate!
```

Conclusión

En conclusión, este proyecto ha logrado implementar en Haskell de manera efectiva el juego de Tic Tac Toe utilizando el algoritmo Minimax como adversario, lo que ha permitido desarrollar un agente invencible. Gracias a este algoritmo, el agente evalúa todas las posibles jugadas y selecciona siempre la mejor opción, lo que lo convierte en un oponente extremadamente difícil de derrotar. De hecho, debido a la naturaleza del Minimax, el mejor resultado que el jugador humano puede esperar es un empate, ya que el agente nunca cometerá un error al tomar sus decisiones. Este proyecto nos ha demostrado cómo la inteligencia artificial, incluso en su forma más simple, puede optimizar la experiencia de juego al ofrecer un desafío constante e interesante para el jugador humano.

Cabe mencionar también que la naturaleza funcional de Haskell ha sido clave para implementar algoritmos recursivos como Minimax y manejar estructuras de datos como el tablero de juego de forma relativamente sencilla y elegante. Sin duda es un lenguaje que quedará en nuestras memorias.