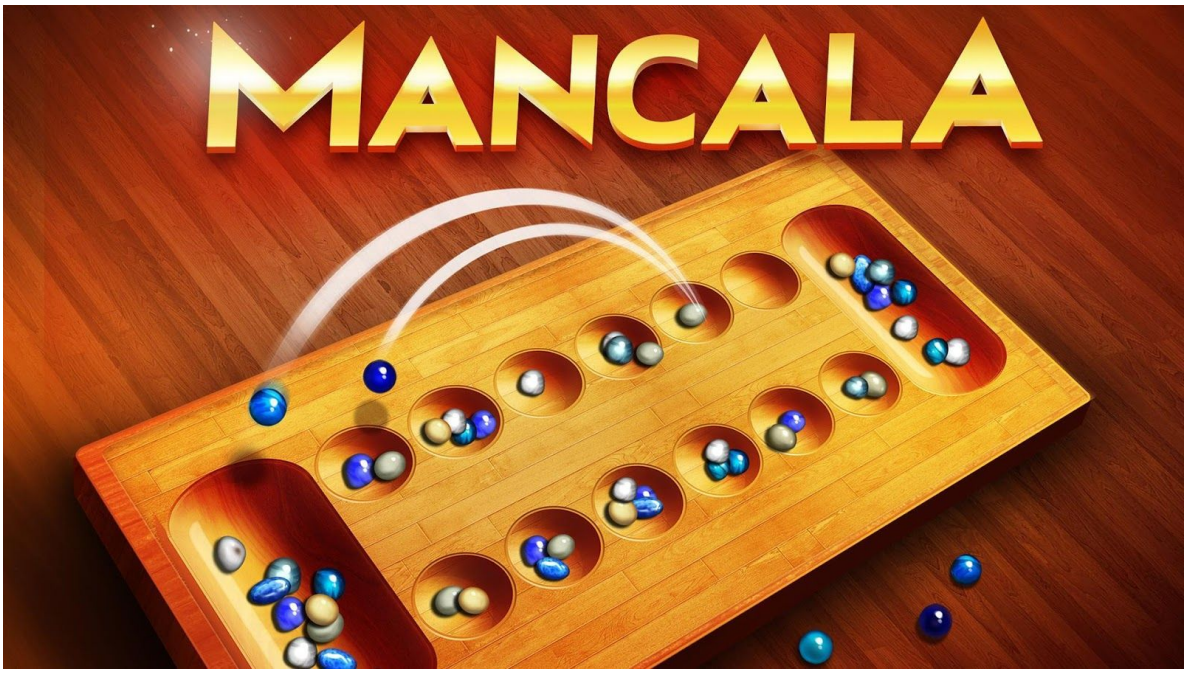
INTELIGENCIA ARTIFICIAL

Práctica 2 – Algoritmo Minimax con poda Alfa-Beta



Patricia Cuesta Ruiz - 03211093V
Laura Ramos Martínez - 03201266B
Isabel Blanco Martínez - 04866897M

Universidad de Alcalá

ÍNDICE

1. Formulación del problema y resolución

2. Resolución Reto 1

3. Resolución Reto 2

4. Resolución Reto 3

5. Resolución Reto 4

6. Resolución Reto 5

1. Formulación del problema y resolución

Esta práctica consiste en la resolución de diferentes retos que afectan a un juego de mesa que debemos versionar, mediante programación funcional. Dicho juego es conocido como Mancala. En nuestro caso, usaremos DrRacket.

A continuación, exponemos las normas del juego:

- **Movimiento de un jugador:** Cada jugador se sitúa a un lado del tablero, controlando la fila de 6 agujeros que esté a su lado. Cada turno, el jugador selecciona un agujero que tenga al menos una ficha de entre los 6 agujeros de su lado, cogiendo las semillas que haya en él (dejándolo vacío) y distribuyendolas una a una en los siguientes agujeros, así como en la casa del jugador, en sentido antihorario.

- **Casa del oponente:** Si al ir colocando las semillas se llegase a la casa del oponente, esta se saltará, ya que un jugador no puede colocar semillas en la casa del oponente.

- **Turno extra:** Si al realizar un movimiento, el último agujero en el que se coloca una semilla es la casa del jugador, este ganará un turno extra seguidamente, pudiendo realizar un movimiento adicional.

- **Robo:** Si al realizar un movimiento, el último agujero en el que se coloca una semilla estaba vacío y pertenece a la fila del propio jugador, este realizará un robo del agujero opuesto en la fila del oponente, llevando todas las semillas que hubiese en dicho agujero directamente a la casa del jugador.

- **Fin de partida:** Si un jugador no puede realizar un movimiento porque todos los agujeros de su fila están vacíos, la partida finalizará. Al finalizar, todas las semillas que queden en la fila del oponente se moverán inmediatamente a la casa del oponente. Ganará el jugador que más semillas tenga en su casa.

Para que el usuario pueda disfrutar de una experiencia completa, se ha desarrollado también una interfaz gráfica que representa el transcurso del juego, y que permite poder visualizar el tablero del juego con las correspondientes semillas y poder colocar las semillas haciendo “click” con el ratón en la posición deseada.

2. Resolución del Reto 1

En este reto hemos de realizar una versión digital del juego orientada a la implementación de una inteligencia artificial. Implementando las reglas del juego, así como el desarrollo de una partida en la que dos IAs (jugador 0 y jugador 1) jueguen una partida completa siguiendo una **estrategia aleatoria**. Se podrá configurar para funcionar en **modo "depuración"**, que imprimirá información por consola, o en **modo "silencioso"**.

Definiremos una serie de variables globales que usaremos a lo largo de todo el código:

```
#lang racket

(module+ test (require rackunit))

; Declaración de variables globales

(define board '((1 1 1 1 1) (1 1 1 1 1) (1 1 1 1 1) (1 1 1 1 1) (1 1 1 1 1) (1 1 1 1 1) ()
                            (1 1 1 1 1) (1 1 1 1 1) (1 1 1 1 1) (1 1 1 1 1) (1 1 1 1 1) (1 1 1 1 1) ()))
(define shoot-again true)
(define ops '((:primera-casilla 0)
              (:segunda-casilla 1)
              (:tercera-casilla 2)
              (:cuarta-casilla 3)
              (:quinta-casilla 4)
              (:sexta-casilla 5)
              (:septima-casilla 7)
              (:octaba-casilla 8)
              (:novena-casilla 9)
              (:decima-casilla 10)
              (:undecima-casilla 11)
              (:duodecima-casilla 12)
             ))
(define depuracion #f)
```

Función print-board :

La siguiente función imprime el tablero de Mancala. La fila inferior corresponde con las casillas del jugador 1, siendo la casilla de la derecha su casa. La fila superior corresponde con el jugador 0, siendo la casilla de la izquierda del todo su casa.

```
; Función que imprime el tablero del mancala, en cada casilla imprime el valor de la suma de las semillas que se encuentran en esa casilla
(define (print-board)
  (printf   "~% ~%       | ~A |  | ~A |  | ~A |  | ~A |  | ~A |  | ~A | ~%"
            (apply + (list-ref board 12)) (apply + (list-ref board 11)) (apply + (list-ref board 10))
            (apply + (list-ref board 9 )) (apply + (list-ref board 8)) (apply + (list-ref board 7)))
  (printf "~% | ~A |                                                | ~A | ~%"
          (apply + (list-ref board 13)) (apply + (list-ref board 6)))
  (printf   "~%       | ~A |  | ~A |  | ~A |  | ~A |  | ~A |  | ~A | ~%~%"
            (apply + (list-ref board 0)) (apply + (list-ref board 1)) (apply + (list-ref board 2))
            (apply + (list-ref board 3)) (apply + (list-ref board 4)) (apply + (list-ref board 5))))
```

```
       | 5 |  | 5 |  | 5 |  | 5 |  | 5 |  | 5 |

 | 0 |                                          | 0 |

       | 5 |  | 5 |  | 5 |  | 5 |  | 5 |  | 5 |
```

Función reset-game:
La siguiente función resetea el juego y vuelve a poner las fichas en sus posiciones originales.

```
; Funcion que reinicia el tablero a su estado original
(define (reset-game)
  (set! board '((1 1 1 1 1)
                (1 1 1 1 1)
                (1 1 1 1 1)
                (1 1 1 1 1)
                (1 1 1 1 1)
                (1 1 1 1 1)
                ()
                (1 1 1 1 1)
                (1 1 1 1 1)
                (1 1 1 1 1)
                (1 1 1 1 1)
                (1 1 1 1 1)
                (1 1 1 1 1)
                ()
                )
        )
  )
```

Función game-ended?:
La función game-ended? comprueba si alguna de las dos filas está completamente vacía, lo que significaría que el juego ha terminado.

```
;Predicado el cual valida si el juego ya terminó comprobando
; si alguna hilera esta completamente vacía
(define (game-ended?)
  (or (and
       (equal? null (list-ref board 0))
       (equal? null (list-ref board 1))
       (equal? null (list-ref board 2))
       (equal? null (list-ref board 3))
       (equal? null (list-ref board 4))
       (equal? null (list-ref board 5)))

      (and
       (equal? null (list-ref board 7))
       (equal? null (list-ref board 8))
       (equal? null (list-ref board 9))
       (equal? null (list-ref board 10))
       (equal? null (list-ref board 11))
       (equal? null (list-ref board 12))
       )
      )
  )
```

Función get-balls:
La función get-balls devuelve el número de semillas que hay en una casilla dada.

```
; Funcion que obtiene las semillas en una casilla
(define (get-balls casilla)
  (list-ref board casilla))
```

Función valid-operator?:

La función valid-operator? comprueba si la casilla seleccionada por el jugador es suya y si es correcta.

```
; Predicado que valida si es el operador seleccionado es valido
(define (valid-operator? operador estado jugador-actual)
  (let ((operador (car(cdr operador))))
    (if (equal? jugador-actual 0)
        (cond ((= operador 0) (if (equal? null (list-ref estado 0))
                                  #f
                                  #t))
              ((= operador 1) (if (equal? null (list-ref estado 1))
                                  #f
                                  #t))
              ((= operador 2) (if (equal? null (list-ref estado 2))
                                  #f
                                  #t))
              ((= operador 3) (if (equal? null (list-ref estado 3))
                                  #f
                                  #t))
              ((= operador 4) (if (equal? null (list-ref estado 4))
                                  #f
                                  #t))
              ((= operador 5) (if (equal? null (list-ref estado 5))
                                  #f
                                  #t))
              (#t
               #f)
             )
        (cond ((= operador 7) (if (equal? null (list-ref estado 7))
                                  #f
                                  #t))
              ((= operador 8) (if (equal? null (list-ref estado 8))
                                  #f
                                  #t))
              ((= operador 9) (if (equal? null (list-ref estado 9))
                                  #f
                                  #t))
              ((= operador 10) (if (equal? null (list-ref estado 10))
                                   #f
                                   #t))
              ((= operador 11) (if (equal? null (list-ref estado 11))
                                   #f
                                   #t))
              ((= operador 12) (if (equal? null (list-ref estado 12))
                                   #f
                                   #t))
              (#t
               #f)
             )
        )
    )
  )
```

Función expand:
Función que cambia todas las casillas del tablero a un estado determinado y los devuelve en una lista.

```
;Funcion que aplica todos los operadores de *ops* a un estado determinado y los regresa en una lista
(define (expand estado jugador)
  (define-values (sucesores nuevo-estado estado-copia)
    (values null null (full-copy estado)))
  (for ([operador ops])
    (when (valid-operator? operador estado-copia jugador)
      (begin
        (set! nuevo-estado (apply-operator operador estado-copia jugador))
        ;(printf "~a " nuevo-estado)
        (set! sucesores (append sucesores (list nuevo-estado)))
        )
      )
    )
  sucesores
  )
```

Función full-copy:
Función auxiliar para la función expand, que duplica una lista.

```
;Funcion auxiliar para expand
(define (full-copy list)
  (if (null? list)
      '()
      (if (list? list)
          (cons (full-copy (car list)) (full-copy (cdr list)))
          list)))
```

Función change-player:
Función que se encarga de cambiar el jugador del turno siguiente.

```
(define (change-player player)
  (if (equal? player 1)
      0
      1
      )
  )
```

Función apply-operator:

La función apply-operator se encarga de devolver todas las posibilidades de movimientos que se pueden realizar.

```
; Funcion que aplica un operador de *ops* a un estado determinado
(define (apply-operator operador estado jugador-actual)
  (define casilla-actual (car(cdr operador)))
  (define-values (ops semillas-casilla estado-resultado)
    (values (car operador)  (get-balls casilla-actual) null))
  ;(printf "~a~%" ops)
  (if (equal? jugador-actual 0)
      (case ops
        [(:primera-casilla) (set! estado-resultado (move-machine-balls estado casilla-actual semillas-casilla))]
        [(:segunda-casilla) (set! estado-resultado (move-machine-balls estado casilla-actual semillas-casilla))]
        [(:tercera-casilla) (set! estado-resultado (move-machine-balls estado casilla-actual semillas-casilla))]
        [(:cuarta-casilla) (set! estado-resultado (move-machine-balls estado casilla-actual semillas-casilla))]
        [(:quinta-casilla) (set! estado-resultado (move-machine-balls estado casilla-actual semillas-casilla))]
        [(:sexta-casilla) (set! estado-resultado (move-machine-balls estado casilla-actual semillas-casilla))]
        [else (printf "error")]
        )
      (case ops
        [(:septima-casilla) (set! estado-resultado (move-machine-balls estado casilla-actual semillas-casilla))]
        [(:octaba-casilla) (set! estado-resultado (move-machine-balls estado casilla-actual semillas-casilla))]
        [(:novena-casilla) (set! estado-resultado (move-machine-balls estado casilla-actual semillas-casilla))]
        [(:decima-casilla) (set! estado-resultado (move-machine-balls estado casilla-actual semillas-casilla))]
        [(:undecima-casilla) (set! estado-resultado (move-machine-balls estado casilla-actual semillas-casilla))]
        [(:duodecima-casilla) (set! estado-resultado (move-machine-balls estado casilla-actual semillas-casilla))]
        [else (printf "error")]
        )
      )
  estado-resultado
  )
```

Función copy-board:

La función copy-board duplica la tabla del tablero.

```
; Funcion que crea una copia del tablero (por motivos de seguridad)
(define (copy-board tablero)
  (let ((estado-copia null))
    (for ([elemento tablero])
      (set! estado-copia (cons elemento estado-copia)))
    (reverse estado-copia)))
```

Función move-machine-balls:

La función move-machine-balls se encarga de mover las semillas de la IA. Además, considera la posibilidad de que pueda volver a tener otro turno ese jugador.

```
; Función que mueve las semillas de la IA
(define (move-machine-balls tablero casilla-actual semillas-casilla)
  (define-values (semilla-a-meter cont estado semillas longitud-semillas estado-copia best-canca shoot-again casilla-target)
    (values null 0 null null 0 null 0 #f (+ 1 casilla-actual)))
  (set! estado-copia (copy-board tablero))
  (for ([can semillas-casilla])
    (set! semillas (cons can semillas))
    )
  (set! semillas (sort semillas >))
  (set! longitud-semillas (length semillas))

  (when (>= (length semillas) (- 13 casilla-actual))
    (begin
      (set! best-canca (car semillas))
      (set! best-canca (list-set estado-copia 13 (append estado-copia (list-ref estado-copia 13))))
      )
    )
  ;Si la longitud de tus semillas es igual a la semilla en la que te encuentras, la IA vuelve a tirar
  (when (= 0 (- (length semillas) (- 13 casilla-actual)))
    (set! shoot-again #t)
    (set! shoot-again #f))

  (for ([semilla semillas])
    (set! semilla-a-meter (car (list-ref estado-copia casilla-actual)))

    (if (and (equal? cont 0 ) (equal? best-canca semilla-a-meter))
        (set! cont (+ 1 cont))
        (begin
          (when (> casilla-target 13)
            (set! casilla-target 0))

          (set! estado-copia (list-set estado-copia casilla-target (append (list semilla-a-meter) (list-ref estado-copia casilla-target))))
          (set! casilla-target (+ 1 casilla-target))
          )
        )
    )
  (set! estado-copia (list-set estado-copia casilla-actual '()))
  (list estado-copia shoot-again casilla-actual)
  )
```

Función juega-random:

La función juega-random simula un turno e imprime si fuera necesario el tablero.

```
; Función juega random
(define (juega-random debug jugador)
  (define posibilidades (expand board jugador))
  (define rng (random (length posibilidades)))
  (set! board (car (list-ref posibilidades rng)))
  (when (equal? debug #t)
    (info-depuracion (list-ref (list-ref posibilidades rng) 2) jugador)
    )
  )
```

Función play-random:

La función play-random es una función recursiva que simula todos los turnos y va cambiando de jugador.

```
; Jugar randomvsrandom
(define (play-random debug jugador1)
  (if (equal? (game-ended?) #f)
      (begin
        (juega-random debug jugador1)
        (play-random debug (change-player jugador1))
        )
      (begin
        (printf "Partida terminada, ganó el jugador: ~a~%" jugador1)
        (printf "Estado final del tablero:~%")
        (print-board)
        )
      )
  )
```

Función info-depuracion:

La función info-depuracion imprime la información referente a cada movimiento de la partida en curso por un jugador.

```
(define (info-depuracion movimiento jugador-actual)
  (printf "Es el turno del jugador ~a.~%" jugador-actual)
  (printf "------------------------------~%")
  (print-board)
  (printf "------------------------------~%")
  (printf "Jugador ~a: elijo el movimiento: ~a~%" jugador-actual movimiento)
  (printf "La heuristica de este movimiento es: ~a~%" (heuristica-simple board))
  (printf "##############################~%")
  )
```

Entradas y salidas para el reto 1

Función “prueba-random”:

La función “prueba-random”,donde se juega una partida comenzando el jugador 1 y con ambos jugadores utilizando una estrategia aleatoria. La función imprime una cabecera informativa y un mensaje final indicando qué jugador (jugador 0 o jugador 1) ha sido el ganador. La función “jugar” y las estrategias en modo “depuración”, mostrando en cada turno qué jugador actúa, el tablero actual y el movimiento elegido por dicho jugador.

```
(define (prueba-random)
  (printf "Comienza el jugador 1, ambos con estratergia random~%")
  (printf "############################~%")
  (play-random #t 1)
  (printf "Partida terminada, ganó el jugador: ~a~%" (ganador? board))
  (printf "Estado final del tablero:~%")
  (print-board)
  (printf "La victoria ha sido por una diferencia de ~a puntos~%" (heuristica-simple board))
  )
```

Salida por pantalla del principio:

```
> (prueba-random)
Comienza el jugador 1, ambos con estratergia random
############################
Es el turno del jugador 1.
----------------------------

      | 6 |  | 6 |  | 6 |  | 0 |  | 5 |  | 5 |

 | 1 |                                    | 0 |

      | 6 |  | 5 |  | 5 |  | 5 |  | 5 |  | 5 |

----------------------------
Jugador 1: elijo el movimiento: 9
La heuristica de este movimiento es: 2
############################
```

Al final de la ejecución:

```
############################
Es el turno del jugador 1.
----------------------------

      | 0 |  | 0 |  | 0 |  | 0 |  | 0 |  | 0 |

 | 24 |                                  | 26 |

      | 2 |  | 1 |  | 1 |  | 1 |  | 2 |  | 2 |

----------------------------
Jugador 1: elijo el movimiento: 12
La heuristica de este movimiento es: 11
############################
Partida terminada, ganó el jugador: 1
Estado final del tablero:

      | 0 |  | 0 |  | 0 |  | 0 |  | 0 |  | 0 |

 | 24 |                                  | 26 |

      | 2 |  | 1 |  | 1 |  | 1 |  | 2 |  | 2 |

La victoria ha sido por una diferencia de 11 puntos
```

Función “prueba-multi-random”:

La función “prueba-multi-random” que juega un cierto número de partidas pasado como parámetro, la mitad de ellas iniciadas por el jugador 1 y la otra mitad por el 2, y muestra el número de victorias, empates y derrotas del jugador 1. En este caso la función jugar y las estrategias deben usarse en modo “silencioso” para que se muestre por consola únicamente un resumen de las partidas.

```
(define (prueba-multi-random cantidad)
  (printf "Se jugarán ~a partidas con agentes aleatorios, la mitad empezando el jugador  1 y la otra mitad empezando el jugadot 2.~%" cantidad)
  (define jugador 1)
  (define-values (ganadas perdidas empates) (values 0 0 0))
  (for ([i cantidad])
    (reset-game)
    (play-random #f jugador)
    (if (equal? (ganador? board) 0)
        (set! perdidas (+ 1 perdidas))
        (if (equal? (ganador? board) 1)
            (set! ganadas (+ 1 ganadas))
            (set! empates (+ 1 empates))
         )
        )
    )
    (printf "Tras ~a, el jugador 1 ha ganado ~a, ha empatado ~a y ha perdido ~a" cantidad ganadas empates perdidas)
)
```

Salida por pantalla, con una cifra que elegimos nosotras. En este caso 1000 partidas:

```
> (prueba-multi-random 1000)
Se jugarán 1000 partidas con agentes aleatorios, la mitad empezando el jugador  1 y la otra mitad empezando el jugadot 2.
Tras 1000, el jugador 1 ha ganado 493, ha empatado 53 y ha perdido 454
```

3. Resolución del Reto 2

En este reto realizamos la implementación de una nueva estrategia de juego basada en el uso de minimax con profundidad limitada y una heurística sencilla. La estrategia minimax se puede configurar con el máximo nivel de profundidad que podrá alcanzar el algoritmo a la hora de evaluar las distintas acciones a realizar. También se podrá elegir la heurística para valorar los tableros una vez alcanzada la profundidad máxima.

En nuestro caso, solo hemos implementado un tipo de heurística simple, por lo que siempre usaremos dicha heurística.

Función heuristica-simple:
La función heurística-simple implementa una heurística simple mediante la operación:

(base1 - base2) + (casillas1 - casillas2)

```
;heuristic-function. Mediante la operación (base1-base2)+(casillas1-casillas2)
(define (heuristica-simple estado)
  (define dif-casas (- (apply + (list-ref estado 6))
                       (apply + (list-ref estado 13))
                       ))
  (define dif-casillas(- (+ (apply + (list-ref estado 0))
                            (apply + (list-ref estado 1))
                            (apply + (list-ref estado 2))
                            (apply + (list-ref estado 3))
                            (apply + (list-ref estado 4))
                            (apply + (list-ref estado 5))
                            )
                         (+ (apply + (list-ref estado 7))
                            (apply + (list-ref estado 8))
                            (apply + (list-ref estado 9))
                            (apply + (list-ref estado 10))
                            (apply + (list-ref estado 11))
                            (apply + (list-ref estado 12))
                            )
                         ))
  (+ dif-casas dif-casillas)
  )
```

Función aplicar min-max:
Función que aplica el algoritmo de min-max para identificar la mejor jugada posible en un jugador pasado por parámetro en el turno, además de poder mostrarse en modo “depuración” o “silencioso”.

```
(define (aplicar-min-max profundidad-max debug jugador-act)
  (define movimiento (min-max board profundidad-max jugador-act))
  (set! board (car movimiento))
  (when (equal? #t debug)
    (info-depuracion (list-ref movimiento 2) jugador-act)
    )

  )
```

Función min-max :

La función min-max aplica el algoritmo del mismo nombre para encontrar cual movimiento es el más favorable al jugador actual.

Para esto primero comprobamos si hemos llegado al máximo número de turnos adelante que podemos comprobar o si el estado de juego en el que estamos no tiene ningún descendiente posible para el jugador actual, es decir, si el jugador se ha quedado sin movimientos legales que realizar, en cuyo caso retornamos solamente el valor heurístico del estado actual y no realizamos ningún cálculo más allá.

En caso contrario, y sí podamos continuar la búsqueda primero comprobamos de qué jugador es el turno actual, o en otras palabras, si debemos maximizar la heurística del siguiente movimiento o minimizarla, posteriormente, recorremos todos los hijos del nodo actual, comprobando si alguno de ellos tiene un valor mejor que el mejor valor actual, propagando esto recursivamente encontramos el camino que mejor resultado da al jugador.

```
; Función que encuentra el mejor movimiento mediante minmax
(define (min-max estado-tablero profundidad jugador)
  (define valorAct 0)
  (define cosaAux null)
  (define sucesores (expand estado-tablero jugador))
  (define mejor-mov null)
  ;(printf "~a~%" profundidad)
  (if (or (equal? profundidad 0) (equal? '() sucesores))
      (list estado-tablero 0 0)
      (begin
        (set! mejor-mov (car sucesores))
        (if (equal? jugador 0)
            (begin
              (set! valorAct -1000000000)
              (for ([hijo sucesores])
                ;bucle que recorra todos los sucesores y pille el maximo
                (begin
                  (set! cosaAux (min-max (car hijo) (- profundidad 1) (change-player jugador)))
                  (when (> valorAct (heuristica-simple (car cosaAux)))
                    (begin
                      (set! valorAct (heuristica-simple (car cosaAux)))
                      (set! mejor-mov cosaAux)
                      )
                    )
                  )
                )
              ;(printf "mejor-mov: ~a~%" mejor-mov)
              mejor-mov
              )
            (begin
              (set! valorAct 1000000000)
              (for ([hijo sucesores])
                ;bucle que recorra todos los sucesores y pille el minimo
                (begin
                  (set! cosaAux (min-max (car hijo) (- profundidad 1) (change-player jugador)))
                  (when (< valorAct (heuristica-simple (car cosaAux)))
                    (begin
                      (set! valorAct (heuristica-simple (car cosaAux)))
                      (set! mejor-mov cosaAux)
                      )
                    )
                  )
                )
              ;(printf "mejor-mov: ~a~%" mejor-mov)
              mejor-mov
              )
            )
        )
      )
  )
```

Función play-min-max:
Función que aplica el algoritmo de min-max para un jugador en una partida completa.

```
;miniMax
(define (play-min-max debug profundidad-max-0 profundidad-max-1 jugador1)
  (if (equal? (game-ended?) #f)
      (begin
        (if (equal? jugador1 1)
            (begin
              (aplicar-min-max profundidad-max-0 debug jugador1)
              (play-min-max debug profundidad-max-0 profundidad-max-1(change-player jugador1))
              )
            (begin
              (aplicar-min-max profundidad-max-1 debug jugador1)
              (play-min-max debug profundidad-max-0 profundidad-max-1 (change-player jugador1))
              )
            )
        )
      (begin
        (printf "Partida terminada, ganó el jugador: ~a~%" (ganador? board))
        (printf "Estado final del tablero:~%")
        (print-board)
        )
      )
  )
```

Entradas y salidas para el reto 2

Función "prueba-minimax":
La función "prueba-minimax" recibe por parámetro un booleano para indicar si comienza el jugador 0 (#t) o el jugador 1 (#f), dos valores enteros que serán los niveles de profundidad que se usarán para el minimax y dos funciones que serán las heurísticas a aplicar de los jugadores y un último parámetro que controle si las funciones funcionan en modo "depuración" o en modo "silencioso".

```
(define (prueba-minmax jugador-bool prof-max-0 prof-max-1 heuristica-0 heuristica-1 debug)
  (define jugador 2)
  (if (equal? #t jugador-bool)
      (set! jugador 1)
      (set! jugador 0)
      )
  (play-min-max debug prof-max-0 prof-max-1 jugador)
  )
```

Siendo la salida por pantalla la siguiente con el modo silencioso:

```
> (prueba-minmax #t 2 4 1 1 #f)
Partida terminada, ganó el jugador: 1
Estado final del tablero:

      | 0 |  | 0 |  | 0 |  | 0 |  | 0 |  | 0 |

 | 10 |                                      | 4 |

      | 1 |  | 5 |  | 3 |  | 15 |  | 11 |  | 10 |

>
```

Función “prueba-multi-minimax-random”:

Función que recibe como parámetro un número indicando el número de partidas a jugar, un entero para indicar el nivel de profundidad que se usará en la estrategia minimax y una función que será la heurística que aplicará la estrategia minimax. Esta función lanzará tantas partidas como se haya indicado, donde el jugador 0 utilizará una estrategia minimax con la profundidad y heurística indicadas y el jugador 1 usará una estrategia aleatoria, iniciando la mitad de las partidas el jugador 0 y la otra mitad el jugador 1. La función jugar y las estrategias se lanzan en modo “silencioso”.

```
(define (prueba-multi-random cantidad)
  (printf "Se jugarán ~a partidas con agentes aleatorios, la mitad empezando el jugador  1 y la otra mitad empezando el jugadot 2.~%" cantidad)
  (define jugador 1)
  (define-values (ganadas perdidas empates) (values 0 0 0))
  (for ([i cantidad])
    (reset-game)
    (play-random #f jugador)
    (if (equal? (ganador? board) 0)
        (set! perdidas (+ 1 perdidas))
        (if (equal? (ganador? board) 1)
            (set! ganadas (+ 1 ganadas))
            (set! empates (+ 1 empates))
         )
        )
    )
    (printf "Tras ~a, el jugador 1 ha ganado ~a, ha empatado ~a y ha perdido ~a" cantidad ganadas empates perdidas)
)
```

Resultado de la ejecución tras 500 partidas:

```
> (prueba-multi-minmax-random 500 3 1)
Se jugarán 500 partidas,Comienza el jugador 1, con estratergia minmax y el jugador 0 con estraregia aleatoria
############################
Tras 500, el jugador 1 ha ganado 314, ha empatado 4 y ha perdido 182
>
```

4. Resolución del Reto 3

Investigar al respecto de las heurísticas aplicables en Mancala e implementar una heurística compleja, basada en combinación lineal de dos o más heurísticas sencillas ajustadas mediante pesos, que tenga en cuenta factores adicionales al número de semillas en la casa de cada jugador. Trata de valorar si esta heurística es mejor que la básica y explica los resultados en la documentación.

Una posible heurística aplicable en Mancala sería intentar que en el movimiento que llevas a cabo, consigas **tener un turno extra**. Sin embargo, esta heurística no siempre puede ser lo más óptimo.

Otra posible heurística puede ser tratar de conseguir tener la **mayor cantidad posible de semillas en la última posición** de nuestra fila. Así lograríamos, en caso de que el otro jugador termine antes, sumar estas semillas en nuestra casa y tendríamos ventaja. Esta heurística sí que podría resultar como efectiva, ya que con esto lograríamos conseguir una ventaja importante al final de la partida.

No hemos sido capaces de implementar esta heurística ya que no dejábamos de tener errores de compilación en su realización, por lo que no la hemos añadido en la práctica resultante.

5. Resolución del Reto 4

Modificación del algoritmo minmax para incluir la optimización de poda alfa-beta. Estudiaremos la diferencia entre el tiempo de ejecución del algoritmo aplicando dicha optimización y sin ella, y compararemos los resultados.

Función alfa-beta:
Esta función implementa la poda alfa-beta al algoritmo min-max, esto le permite eliminar aquellos caminos aún por comprobar que ya sepamos que no van a resultar en un camino óptimo. Esto está logrado gracias a hacer una poda cuando el valor de beta supera al de alfa o el de alfa es menor que el de beta, dejando de comprobar los hijos de ese nodo, reduciendo el tiempo de computación requerido para llegar al camino óptimo.

```
(define (alfa-beta estado-tablero alfa beta profundidad jugador)
  (define valorAct 0)
  (define cosaAux null)
  (define sucesores (expand estado-tablero jugador))
  (define mejor-mov null)
  ;(printf "~a~%" profundidad)
  (if (or (equal? profundidad 0) (equal? '() sucesores))
      (list estado-tablero 0 0)
      (begin
        (set! mejor-mov (car sucesores))
        (if (equal? jugador 0)
            (begin
              (set! valorAct -1000000000)
              (for ([hijo sucesores]
                    #:break (> beta alfa))
                ;bucle que recorra todos los sucesores y pille el maximo
                (begin
                  (set! cosaAux (alfa-beta (car hijo) alfa beta (- profundidad 1) (change-player jugador)))
                  (when (> valorAct (heuristica-simple (car cosaAux)))
                    (begin
                      (set! valorAct (heuristica-simple (car cosaAux)))
                      (set! alfa valorAct)
                      (set! mejor-mov cosaAux)
                      )
                    )
                  )
                )
              ;(printf "mejor-mov: ~a~%" mejor-mov)
              mejor-mov
              )
            (begin
              (set! valorAct 1000000000)
              (for ([hijo sucesores]
                    #:break(> alfa beta))
                ;bucle que recorra todos los sucesores y pille el minimo
                (begin
                  (set! cosaAux (alfa-beta (car hijo) alfa beta (- profundidad 1) (change-player jugador)))
                  (when (< valorAct (heuristica-simple (car cosaAux)))
                    (begin
                      (set! valorAct (heuristica-simple (car cosaAux)))
                      (set! beta valorAct)
                      (set! mejor-mov cosaAux)
                      )
                    )
                  )
                )
              ;(printf "mejor-mov: ~a~%" mejor-mov)
              mejor-mov
              )
            )
        )
      )
  )
```

Función aplicar-alfa-beta:

Función que aplica el algoritmo de min-max con la poda alfa-beta para identificar la mejor jugada posible en un jugador pasado por parámetro en el turno, además de poder mostrarse en modo “depuración” o “silencioso”.

```
(define (aplicar-alfa-beta profundidad-max alfa beta debug jugador-act)
  (define movimiento (alfa-beta board alfa beta profundidad-max jugador-act))
  (set! board (car movimiento))
  (when (equal? #t debug)
    (info-depuracion (list-ref movimiento 2) jugador-act)
    )

  )
```

Función play-alfa-beta:

Función que aplica el algoritmo de min-max con poda alfa-beta para un jugador en una partida completa.

```
; Función juego alfa-beta
(define (play-alfa-beta debug profundidad-max alfa beta jugador1)
  (if (equal? (game-ended?) #f)
      (begin
        (aplicar-alfa-beta profundidad-max alfa beta debug jugador1)
        (play-alfa-beta debug profundidad-max alfa beta (change-player jugador1))
        )
      (begin
        (printf "Partida terminada, ganó el jugador: ~a~%" (ganador? board))
        (printf "Estado final del tablero:~%")
        (print-board)
        )
      )
  )
```

Diferencia entre el tiempo de ejecución del algoritmo con y sin poda alfa-beta

Haciendo uso de current-milliseconds vamos a comparar cual de los dos algoritmos lograría obtener un menor tiempo de ejecución.

```
(define (tiempoEjecucionMinMax )
  (define tiempoMinMax 0)
  (set! tiempoMinMax (current-milliseconds))
  (play-min-max #f 4 1 1)
  (set! tiempoMinMax (- (current-milliseconds) tiempoMinMax))
  (printf "Tiempo de ejecución de min-max:~a~% " tiempoMinMax))

(define (tiempoEjecucionAlfaBeta )
  (define tiempoAlfaBeta 0)
  (set! tiempoAlfaBeta (current-milliseconds))
  (play-alfa-beta #f 4 1 1 1)
  (set! tiempoAlfaBeta (- (current-milliseconds) tiempoAlfaBeta))
  (printf "Tiempo de ejecución de poda alfa-beta:~a~% " tiempoAlfaBeta))
```

- **Tiempo de ejecución de Minimax en una partida:**

El tiempo de ejecución en este ejemplo es de 397 milisegundos.

```
> (tiempoEjecucionMinMax )
Partida terminada, ganó el jugador: 1
Estado final del tablero:

      | 0 |  | 0 |  | 0 |  | 0 |  | 0 |  | 0 |

 | 10 |                                        | 4 |

      | 1 |  | 5 |  | 3 |  | 15 |  | 11 |  | 10 |

Tiempo de ejecución de min-max:397
```

- **Tiempo de ejecución con Poda alfa-beta en una partida:**

El tiempo de ejecución en este ejemplo es de 189 milisegundos. Esto se debe a que este algoritmo siempre debe finalizar antes que el de minimax porque se realiza una poda y se eliminan los nodos.

```
> (tiempoEjecucionAlfaBeta )
Partida terminada, ganó el jugador: 1
Estado final del tablero:

      | 0 |  | 0 |  | 0 |  | 0 |  | 0 |  | 0 |

 | 10 |                                        | 4 |

      | 1 |  | 5 |  | 3 |  | 15 |  | 11 |  | 10 |

Tiempo de ejecución de poda alfa-beta:189
```

6. Resolución del Reto 5

Realizamos los tests unitarios de las funciones utilizadas en el programa exceptuando aquellas que únicamente imprimen por consola y aquellas que no devuelven nada, para comprobar su correcto funcionamiento.
Por lo tanto, mostramos el test de 13 de 20 funciones, sin contar las 4 funciones de entrada y salida.

Test unitario para reset-game:

```
(module+ test (begin (reset-game)
                     (check-equal?
                      '((1 1 1 1 1)
                        (1 1 1 1 1)
                        (1 1 1 1 1)
                        (1 1 1 1 1)
                        (1 1 1 1 1)
                        (1 1 1 1 1)
                        ()
                        (1 1 1 1 1)
                        (1 1 1 1 1)
                        (1 1 1 1 1)
                        (1 1 1 1 1)
                        (1 1 1 1 1)
                        (1 1 1 1 1)
                        ())
                      board))
  )
```

Test unitario para game-ended?:

```
(module+ test (begin (reset-game)
                     (check-equal? #f (game-ended?))
                     )
  )
```

Test unitario para la función get-balls:

```
(module+ test (begin (reset-game)
                     (check-equal? 5 (apply + (get-balls 3)))
                     )
  )

(module+ test (begin (reset-game)
                     (check-equal? 0 (apply + (get-balls 13)))
                     )
  )
```

Test unitario para la función valid-operator:

```
(module+ test (begin (reset-game)
                     (check-equal? #t (valid-operator? (list-ref ops 1) board 0))
                     )
  )

(module+ test (begin (reset-game)
                     (check-equal? #t (valid-operator? (list-ref ops 7) board 1))
                     )
  )
```

Test unitario para la función apply-operator:

```
(module+ test (begin (reset-game)
                    (check-equal?
                     '((() (1 1 1 1 1 1) (1 1 1 1 1 1) (1 1 1 1 1 1) (1 1 1 1 1 1) (1 1 1 1 1 1) ()
                           (1 1 1 1 1) (1 1 1 1 1) (1 1 1 1 1) (1 1 1 1 1) (1 1 1 1 1) (1 1 1 1 1) ()) #f 0)
                     (apply-operator (list-ref ops 0) board 0)
                     )
                    )
  )

(module+ test (begin (reset-game)
                    (check-equal?
                     '(((1 1 1 1 1) (1 1 1 1 1) (1 1 1 1 1) (1 1 1 1 1) (1 1 1 1 1) (1 1 1 1 1) ()
                                    () (1 1 1 1 1 1) (1 1 1 1 1 1) (1 1 1 1 1 1) (1 1 1 1 1 1) (1 1 1 1 1 1) (1 1 1 1 1 1) ()) #f 7)
                     (apply-operator (list-ref ops 6) board 1)
                     )
                    )
  )
```

Test unitario para la función copy-board:

```
(module+ test (begin (reset-game)
                     (check-equal? board (copy-board board))
                     )
  )
```

Test unitario para move-machine-balls:

```
(module+ test (begin (reset-game)
                    (check-equal? '(((1 1 1 1 1) (1 1 1 1 1) () (1 1 1 1 1 1) (1 1 1 1 1 1) (1 1 1 1 1 1)
                                                 (1) (1 1 1 1 1 1) (1 1 1 1 1) (1 1 1 1 1) (1 1 1 1 1) (1 1 1 1 1) (1 1 1 1 1) ()) #f 2)
                                  (move-machine-balls board 2 (list-ref board 0))
                                  )
            )
  )
```

Test unitario para change-player:

```
(module+ test (check-equal? 0 (change-player 1)))
(module+ test (check-equal? 1 (change-player 0)))
```

Tests unitarios para función expand:

```
(module+ test (begin (reset-game)
                     (check-equal? '((((() (1 1 1 1 1 1) (1 1 1 1 1 1) (1 1 1 1 1 1) (1 1 1 1 1 1) (1 1 1 1 1 1) () (1 1 1 1 1 1) (1 1 1 1 1 1) (1 1 1 1 1 1) (1 1 1 1 1 1) (1 1 1 1 1 1) (1 1 1 1 1 1) (1 1 1 1 1 1) (()) #f 0)
                                     (((1 1 1 1 1 1) () (1 1 1 1 1 1) (1 1 1 1 1 1) (1 1 1 1 1 1) (1 1 1 1 1 1) (1) (1 1 1 1 1 1) (1 1 1 1 1 1) (1 1 1 1 1 1) (1 1 1 1 1 1) (1 1 1 1 1 1) (1 1 1 1 1 1) (1 1 1 1 1 1) (()) #f 1)
                                     (((1 1 1 1 1 1) (1 1 1 1 1 1) () (1 1 1 1 1 1) (1 1 1 1 1 1) (1 1 1 1 1 1) (1) (1 1 1 1 1 1) (1 1 1 1 1 1) (1 1 1 1 1 1) (1 1 1 1 1 1) (1 1 1 1 1 1) (1 1 1 1 1 1) (1 1 1 1 1 1) (()) #f 2)
                                     (((1 1 1 1 1 1) (1 1 1 1 1 1) (1 1 1 1 1 1) () (1 1 1 1 1 1) (1 1 1 1 1 1) (1) (1 1 1 1 1 1) (1 1 1 1 1 1) (1 1 1 1 1 1) (1 1 1 1 1 1) (1 1 1 1 1 1) (1 1 1 1 1 1) (1 1 1 1 1 1) (()) #f 3)
                                     (((1 1 1 1 1 1) (1 1 1 1 1 1) (1 1 1 1 1 1) (1 1 1 1 1 1) () (1 1 1 1 1 1) (1) (1 1 1 1 1 1) (1 1 1 1 1 1) (1 1 1 1 1 1) (1 1 1 1 1 1) (1 1 1 1 1 1) (1 1 1 1 1 1) (1 1 1 1 1 1) (()) #f 4)
                                     (((1 1 1 1 1 1) (1 1 1 1 1 1) (1 1 1 1 1 1) (1 1 1 1 1 1) () (1) (1 1 1 1 1 1) (1 1 1 1 1 1) (1 1 1 1 1 1) (1 1 1 1 1 1) (1 1 1 1 1 1) (1 1 1 1 1 1) (1 1 1 1 1 1) (()) #f 5))
                                    (expand board 0)
                                    )
                     )
  )

(module+ test (begin (reset-game)
                     (check-equal? '((((1 1 1 1 1) (1 1 1 1 1) (1 1 1 1 1) (1 1 1 1 1) (1 1 1 1 1) (1 1 1 1 1) () () (1 1 1 1 1 1) (1 1 1 1 1 1) (1 1 1 1 1 1) (1 1 1 1 1 1) (1 1 1 1 1 1) (()) #f 7)
                                     (((1 1 1 1 1) (1 1 1 1 1) (1 1 1 1 1) (1 1 1 1 1) (1 1 1 1 1) (1 1 1 1 1) (1 1 1 1 1) () (1 1 1 1 1) () (1 1 1 1 1 1) (1 1 1 1 1 1) (1 1 1 1 1 1) (1 1 1 1 1 1) (1 1 1 1 1 1) (1)) #f 8)
                                     (((1 1 1 1 1 1) (1 1 1 1 1) (1 1 1 1 1) (1 1 1 1 1) (1 1 1 1 1) (1 1 1 1 1) (1 1 1 1 1) () (1 1 1 1 1) (1 1 1 1 1) () (1 1 1 1 1 1) (1 1 1 1 1 1) (1 1 1 1 1 1) (1 1 1 1 1 1) (1)) #f 9)
                                     (((1 1 1 1 1 1) (1 1 1 1 1 1) (1 1 1 1 1) (1 1 1 1 1) (1 1 1 1 1) (1 1 1 1 1) (1 1 1 1 1) () (1 1 1 1 1) (1 1 1 1 1) (1 1 1 1 1) () (1 1 1 1 1 1) (1 1 1 1 1 1) (1 1 1 1 1 1) (1)) #f 10)
                                     (((1 1 1 1 1 1) (1 1 1 1 1 1) (1 1 1 1 1) (1 1 1 1 1) (1 1 1 1 1) (1 1 1 1 1) (1 1 1 1 1) () (1 1 1 1 1) (1 1 1 1 1) (1 1 1 1 1) (1 1 1 1 1) () (1 1 1 1 1 1) (1 1 1 1 1 1) (1)) #f 11)
                                     (((1 1 1 1 1 1) (1 1 1 1 1 1) (1 1 1 1 1 1) (1 1 1 1 1) (1 1 1 1 1) (1 1 1 1 1) (1 1 1 1 1) () (1 1 1 1 1) (1 1 1 1 1) (1 1 1 1 1) (1 1 1 1 1) (1 1 1 1 1) () (1)) #f 12))
                                    (expand board 1)
                                    )
                     )
  )
```

Test unitario para full-copy:

```
(module+ test (check-equal? board (full-copy board)))
```

Tests unitarios para la función heuristica-simple:

```
(module+ test (begin (reset-game)
                     (check-equal? 0 (heuristica-simple board))
                     )
  )

(module+ test (check-equal? 4 (heuristica-simple
                               '(() (1) (1) (1) (1) () (1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1)
                                    () () () () () () (1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1))
                               )
                            )
  )
```

Test unitario para la función min-max:

```
(module+ test (begin (reset-game)
                     (check-equal? (min-max board 2 0)
                                   '((() (1 1 1 1 1 1) (1 1 1 1 1 1) (1 1 1 1 1 1) (1 1 1 1 1 1) (1 1 1 1 1 1) ()
                                         (1 1 1 1 1 1) (1 1 1 1 1 1) (1 1 1 1 1 1) (1 1 1 1 1 1) (1 1 1 1 1 1) (()) #f 0)
                                   )
                     )
  )
```

Test unitario para la función alfa-beta:

```
(module+ test (begin (reset-game)
                     (check-equal? '((() (1 1 1 1 1 1) (1 1 1 1 1 1) (1 1 1 1 1 1) (1 1 1 1 1 1) ()
                                         (1 1 1 1 1 1) (1 1 1 1 1 1) (1 1 1 1 1 1) (1 1 1 1 1 1) (1 1 1 1 1 1) (()) #f 0)
                                   (alfa-beta board -1000 1000 2 0)
                                   )
                     )
  )
```