Pareja: 02
Alba Ramos Pedroviejo
Nicolás Serrano Salas

Práctica 3
Grupo: 2361
INTART

Para el desarrollo de esta práctica hemos estudiado los diferentes mecanismos más importantes para ganar una partida de Othello. Para ello, según hemos visto, una de las mejores soluciones es asignar valores a cada una de las casillas y, en especial, a las esquinas. No obstante también hemos querido tener en cuenta, tras evaluar el tablero con los distintos pesos, la movilidad de cada jugador. Hemos desarrollado hasta un total de 30 funciones las cuales hemos ido comparando entre ellas mediante un mecanismo de ensayo y error para saber cual es mejor entre ellas. Muchas de estas funciones son mezclas de varias funciones anteriores por las cuales nos dimos cuenta que estas son mejores que otras.

A continuación están las tres funciones con sus funciones que subimos al último torneo.

Funciones finales:

Primera función:

```
(defun mobility-modified-matrix-solution-2-algoriths (player board)
 (let ((foo (modified-matrix-solution-2-algoriths player board))
       (bar (mobility player board)))
   (cond ((and (< foo 0) (not (equal bar 0))) (/ foo bar))
         ((and (< foo 0) (equal bar 0)) foo)
         (t (* foo bar)))))
```

Esta función devuelve el valor actual del tablero para el jugador, teniendo en cuenta las modificaciones del tablero e intentando maximizar también la movilidad que tendrá ese jugador tras el movimiento. La primera condición si el valor del tablero es negativo, queremos obtener el tablero que más movimientos nos deje y al dividir el valor por el número de movimientos nos da un número negativo mayor (más cercano a 0). La segunda condición indica que si el numero movimientos es cero y el valor del tablero negativo, como no se puede dividir directamente devolvemos el valor del tablero. Finalmente si el valor de la matriz es positivo queremos obtener el que mayor valor tenga y mayor número de movimientos nos dé.

Segunda función:

```
(defun mobility-modified-matrix-solution-2-algoriths-2 (player board)
 (let ((foo (modified-matrix-solution-2-algoriths player board))
       (bar (mobility player board)))
   (cond ((and (< foo 0) (not (equal bar 0))) (/ foo bar))
         ((and (< foo 0) (equal bar 0)) foo)
         ((equal foo 0) bar)
         (t (* foo bar)))))
```

Esta segunda función sigue una lógica similar a la primera pero tiene un acercamiento diferente en las condiciones. La primera condición si el valor del tablero es negativo, queremos obtener el tablero que más movimientos nos deje y al dividir el valor por el número de movimientos nos da un número negativo mayor (más cercano a 0). La segunda condición indica que si el numero movimientos es cero y el valor del tablero negativo, como no se puede dividir directamente devolvemos el valor del tablero. La tercera es que si el valor del tablero es 0, se devuelve el valor de la movilidad. Finalmente si el valor de la matriz es positivo queremos obtener el que mayor valor tenga y mayor número de movimientos nos dé.

Tercera función:

```
(defun both-mobility-modified-matrix-solution-2-algoriths-2 (player
board)
   (+  (mobility-modified-matrix-solution-2-algoriths-2 player board)
       (other-mobility player board)))
```

Esta función utiliza la función auxiliar mobility-modified-matrix-solution-2-algoriths-2 (segunda función) pero implementa también la lógica de la segunda función de tener en cuenta la movilidad del otro jugador.

Parámetros auxiliares:

```
(defparameter matrix-solution-2
 '((120 -20 20 5  5  20 -20 120)
   (-20 -40 -5 -5 -5 -5 -40 -20)
   (20  -5  15 3  3  15 -5  20)
   (5   -5  3  3  3  3  -5  5)
   (5   -5  3  3  3  3  -5  5)
   (20  -5  15 3  3  15 -5  20)
   (-20 -40 -5 -5 -5 -5 -40 -20)
```

```
   (120 -20 20 5  5  20 -20 120)))

(defparameter *weights*
 '#(0   0   0  0  0  0  0   0   0 0
    0 120 -20 20  5  5 20 -20 120 0
    0 -20 -40 -5 -5 -5 -5 -40 -20 0
    0  20  -5 15  3  3 15  -5  20 0
    0   5  -5  3  3  3  3  -5   5 0
    0   5  -5  3  3  3  3  -5   5 0
    0  20  -5 15  3  3 15  -5  20 0
    0 -20 -40 -5 -5 -5 -5 -40 -20 0
    0 120 -20 20  5  5 20 -20 120 0
    0   0   0  0  0  0  0   0   0 0))
```

Estos parámetros definen los pesos para las diferentes casillas en la matriz. Ambos son los mismos pesos, pues son los mejores que hemos encontrado, pero para distintas funciones, como hemos observado mediante pruebas y ambas de nuestras fuentes.

Funciones auxiliares:

Cálculo de las matrices:

A continuación encontramos las diferentes funciones auxiliares para calcular el valor de cada tablero

```
(defun get-elemento (player elem-tablero elem-solucion)
 (cond ((equal player elem-tablero) elem-solucion)
       ((equal (opponent player) elem-tablero) (- 0 elem-solucion))
       (T 0)))

(defun suma-fila (player fila-tablero fila-solucion)
 (reduce #'+ (mapcar #'(lambda (x y) (get-elemento player x y))
fila-tablero fila-solucion)))

(defun suma-matriz (player board solucion)
 (reduce #'+ (mapcar #'(lambda (x y) (suma-fila player x y)) board
solucion)))

(defun matrix-solution-2-algorith (player board)
 (suma-matriz player (get-board board) matrix-solution-2))
```

Estas funciones son las encargadas de calcular el valor del tablero con el parámetro matrix-solution-2.
Va comparando posición por posición del tablero y suma o resta los valores si la posición es ocupada por el usuario o el oponente.

Funciones de movilidad

```
(defun mobility (player board)
 "The number of moves a player has.
 (length (legal-moves player board)))

(defun other-mobility (player board)
 (- 0 (length (legal-moves (opponent player) board))))
```

Estas funciones se encargan de calcular la lista de posibles movimientos del usuario. La primera calcula la del propio usuario y la segunda la del oponente.

Modificación de los pesos

```
(defun modified-matrix-solution-2-algoriths (player board)
 (let ((w (matrix-solution-2-algorith player board)))
   (dolist (corner '(11 18 81 88))
     (when (not (eql (bref board corner) empty))
       (dolist (c (neighbors corner))
         (when (not (eql (bref board c) empty))
           (incf w (* (- 5 (aref *weights* c))
                      (if (eql (bref board c) player)
                          +1 -1)))))))
   w))
```

Esta función es la encargada de modificar los pesos de las casillas adyacentes a las esquinas cuando estas ya están ocupadas, pues ya no tienen un valor tan negativo.
Recogida de la primera fuente. Esta función estaba basada en la función modified-weighted-squares de la primera fuente.

Otras funciones

```
(defun mobility-matrix-solution-2-algorith-2 (player board)
 (let ((foo (matrix-solution-2-algorith player board))
       (bar (mobility player board)))
   (cond ((and (< foo 0) (not (equal bar 0))) (/ foo bar))
         ((and (< foo 0) (equal bar 0)) foo)
         (t (* foo bar)))))
```

Esta función fue una de las primeras y mejores funciones que hicimos al principio. Devuelve el valor actual del tablero para el jugador, intentando maximizar también la movilidad que tendrá ese jugador tras el movimiento. La primera condición si el valor del tablero es negativo, queremos obtener el tablero que más movimientos nos deje y al dividir el valor por el número de movimientos nos da un número negativo mayor (más cercano a 0). La segunda condición indica que si el numero movimientos es cero y el valor del tablero negativo, como no se puede dividir directamente devolvemos el valor del tablero. Finalmente si el valor de la matriz es positivo queremos obtener el que mayor número de movimientos nos dé.

Fuentes:

[1] Norvig, P. (1991). *Paradigms of artificial intelligence programming: case studies in Common Lisp*. San Francisco, CA: Morgan Kaufman Publishers. Recuperado de https://github.com/norvig/paip-lisp/blob/master/docs/chapter18.md

[2] Zhang, Z and Chen, Y. *Searching Algorithms in Playing Othello*
Recuperado de
http://web.eecs.utk.edu/~zzhang61/docs/reports/2014.04%20-%20Searching%20Algorithms%20in%20Playing%20Othello.pdf