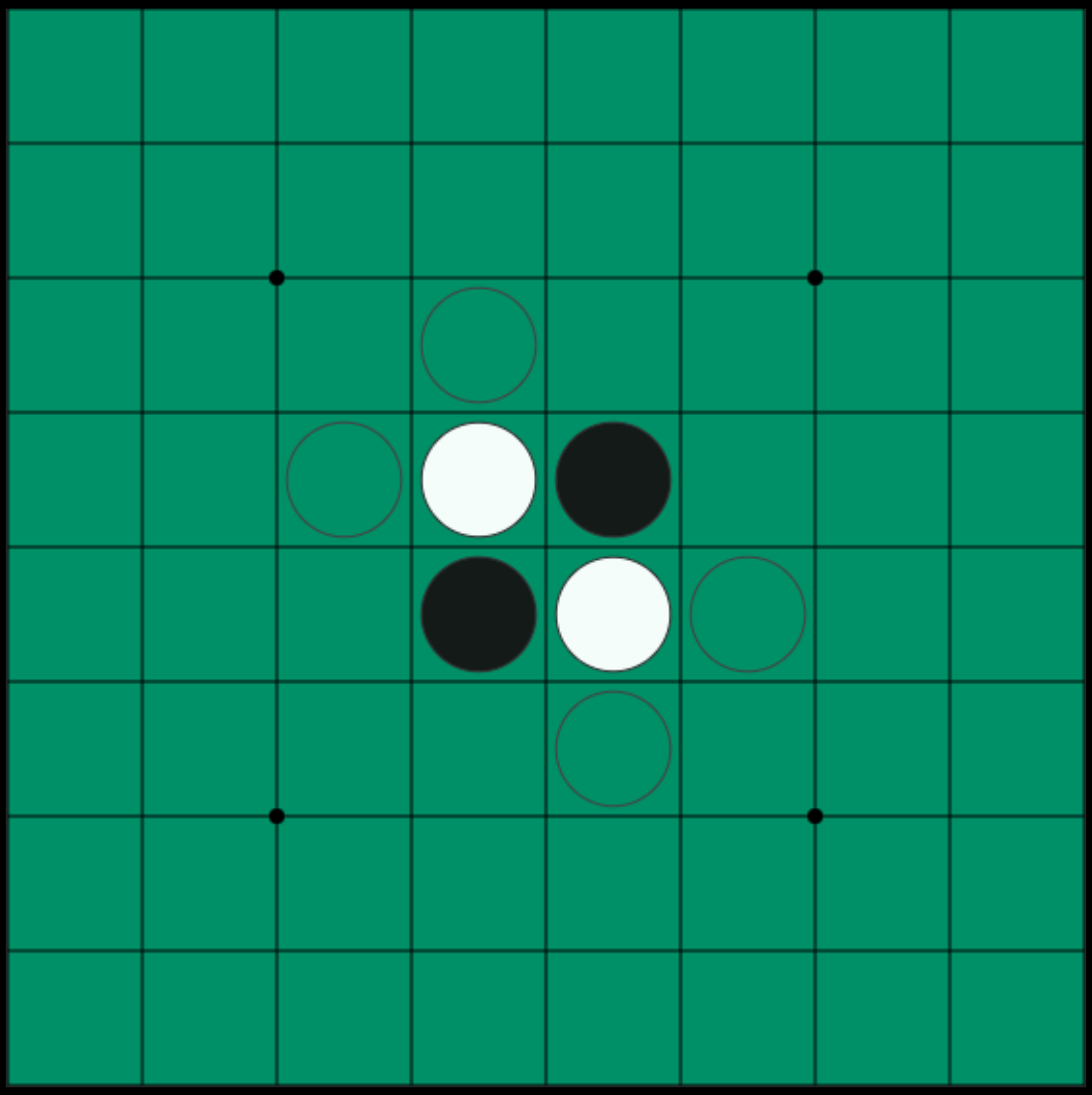


EL JUEGO DE OTHELLO

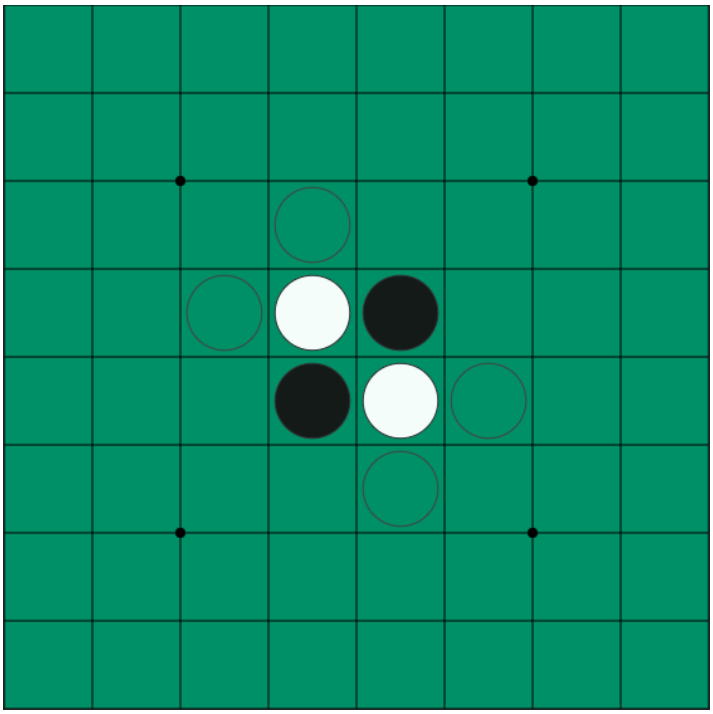
PECL2 INTELIGENCIA ARTIFICIAL

DAVID MARQUEZ MINGUEZ – 47319570Z

INTRODUCCIÓN

En esta practica se pide realizar un programa en Racket que simule el juego de Othello, también conocido como Reversi. El proyecto esta dividido en varios archivos racket, “Othello.rkt”, “funciones-aux.rkt” y “pos-disponibles.rkt”. El primer archivo contiene las funciones principales del juego, funciones que se encargan de iniciar el menú de usuario y las funciones necesarias para ejecutar el juego en sus diferentes modos. Tanto el segundo como el tercer archivo contienen funciones adicionales que son necesaria para la correcta ejecución del programa.

Además se añade un directorio adicional donde se presenta varios tableros adicionales para comprobar el correcto funcionamiento del juego. El la carpeta ‘test’ se incluyen cinco tableros distintos, por otro lado, es posible añadir mas tableros creando un archivo txt respetando la estructura para la correcta lectura del mismo por parte del programa.



En cuanto al juego, se comienza por un estado inicial del tablero que en la mayoría de los casos suele ser el mismo.

En turnos alternos dos jugadores van colocando piezas en un tablero de 8x8 casillas. En la implementación realizada en esta practica existen dos modos de juego, un modo de juego jugador contra jugador, y otro modo de juego jugador contra máquina.

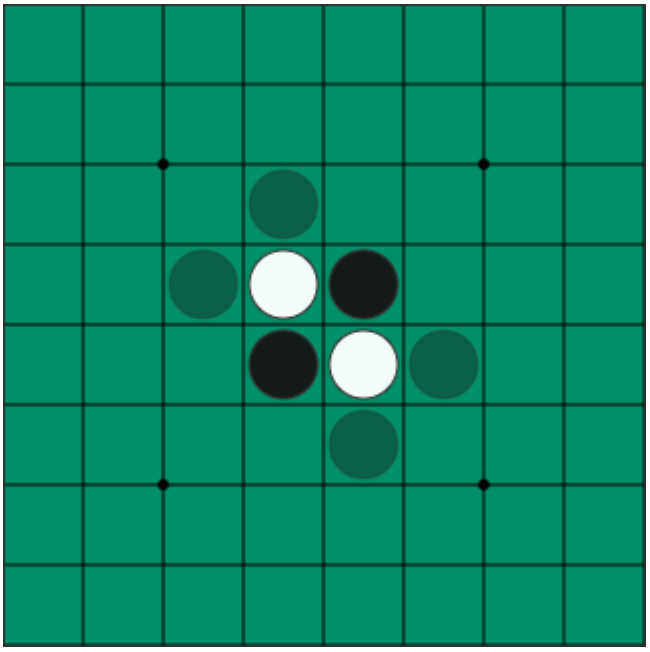
Una vez que el tablero esta lleno, es decir, el juego se ha terminado, se contabilizan las piezas de cada color y se determina el ganador del mismo. En el siguiente apartado se mostrará como se juega y la forma en la que se cambian las piezas de color.

El objetivo de la practica es aplicar el algoritmo MinMax y poda Alpha-Beta. El algoritmo MinMax elegirá el mejor movimiento del jugador maximizador en cada turno. Se considerarán todos los posibles estados del tablero y se elegirá la mejor jugada. En cuanto al algoritmo de poda, este contiene dos valores, alpha y beta que representan el puntaje máximo que el jugador maximizador tiene asegurado y el puntaje mínimo que el jugador minimizador tiene asegurado respectivamente.

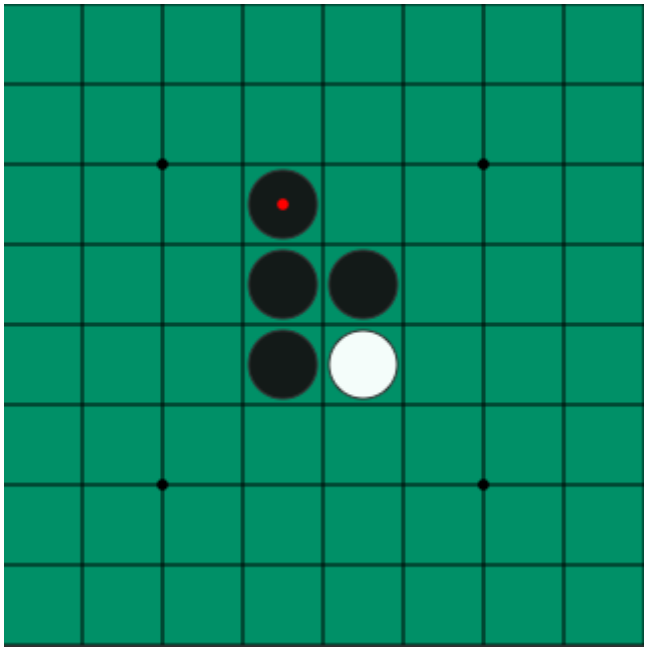
REGLAS DEL JUEGO

Como ya he dicho antes Othello es un juego clásico de dos jugadores, donde cada uno de los jugadores en turnos alternos debe ir colocando fichas de forma que consigan cambiar a su color las máximas fichas posibles del jugador contrario.

En cuanto a la forma de colocar las fichas cada jugador puede colocar su propia ficha de forma que haya al menos una línea horizontal, vertical o diagonal ocupada entre la nueva ficha y otra ficha del mismo color intercalada con una ficha del color contrario. De forma que para el tablero mostrado en la imagen anterior, el jugador con fichas negras tendría las siguiente posibilidades:



En este caso si colocamos una ficha de color negro en la posición (3, 2), tercera coluna, segunda fila, la ficha blanca intercalada entre las fichas negras se cambia de color como se muestra a continuación.



Si de diese el caso en el que un jugador no pudiese realizar un movimiento valido, se pasa el turno al siguiente jugador. Cuando ningún jugador puede realizar un movimiento valido, o cuando el tablero está lleno, la partida se termina y se cuentan las fichas de ambos jugadores. Aquel jugador que acumule mas fichas de su color gana.

ANALISIS DE CODIGO

El análisis del código lo voy a dividir en varias partes, en primer lugar la lectura de los archivos txt, en segundo lugar las funciones auxiliares, en tercer lugar las funciones referentes a las posiciones disponibles y por último el algoritmo MinMax.

En primer lugar en cuanto a la lectura de los ficheros txt, se realizan de la misma manera que en la lectura del sudoku. He implementado dos funciones:

- delete-lists: esta función se encarga de eliminar las líneas del archivo txt que no me interesan, es decir, todas aquellas líneas que no tengan dígitos.
- limpiar-lista: una dispongo de las líneas del txt que me interesan, debo limpiarlas, es decir, eliminar los símbolos que hacen que la lectura no sea limpia. Una vez aplicada esta función convierto el archivo txt en una matriz compuesta únicamente por los dígitos deseados.

Un aves se ha leído el archivo se comprueba si el tablero esta lleno, o si el tablero determinado es válido en cuyo caso se sigue ejecutando el programa.

En cuanto a las funciones auxiliares, son sobre todo funciones get y set de estructuras como columnas, filas, diagonales… Además se incluyen funciones para trasponer e invertir el Othello, así como funciones para cambiar o seleccionar posiciones determinadas del mismo. Algunas de las funciones más importantes son:

- get/set-elem: selecciona/modifica una posición determinada del tablero dadas sus coordenadas en el mismo.
- get/set-fila: selecciona/modifica una fila del tablero, si queremos modificarla debemos pasar como valor una lista del mismo tamaño.
- get/set-columna: selecciona/modifica una columna del tablero, si queremos modificarla debemos pasar como valor una lista del mismo tamaño.
- get/set-diagonal: selecciona/modifica una diagonal del tablero, si queremos modificarla debemos pasar como valor una lista del mismo tamaño.
- get/set-diagonal-secundaria: selecciona/modifica una diagonal secundaria del tablero, si queremos modificarla debemos pasar como valor una lista del mismo tamaño.
- transponer-othello: devuelve el tablero intercambiando filas por columnas.
- invertir-tablero: dado un tablero lo invierte.

A continuación voy a explicar la lógica empleada para encontrar las posiciones disponibles de cada jugador.

Para encontrar las posiciones disponibles de cada jugador donde colocar su ficha he realizado varias funciones para encontrar las fichas disponibles, tanto en las filas como en las columnas, como en las diagonales y las diagonales secundarias.

Para determinar si hay una posición disponible se tiene que dar la siguiente secuencia, si queremos colocar una ficha 2, la secuencia seri ala siguiente: 2 1 X, donde X sería el resto de la lista. En el momento en el que se encuentre una lista con dicha secuencia se determina su puntuación para el posterior cálculo de la mejor jugada.

- pos-disponibles-fila: determina las posiciones disponibles a lo largo de una fila.
- pos-disponibles-columna: determina las posiciones disponibles a lo largo de una columna.
- pos-disponibles-diagonal: determina las posiciones disponibles a lo largo de una diagonal.
- pos-disponibles-diagonal-secundaria: determina las posiciones disponibles a lo largo de una diagonal secundaria.

Una vez determinadas dichas posiciones en el momento en el que el jugador coloca la pieza en una de estas, se ejecuta el algoritmo de expansión de dicha pieza, así como la puntuación de dicha expansión.

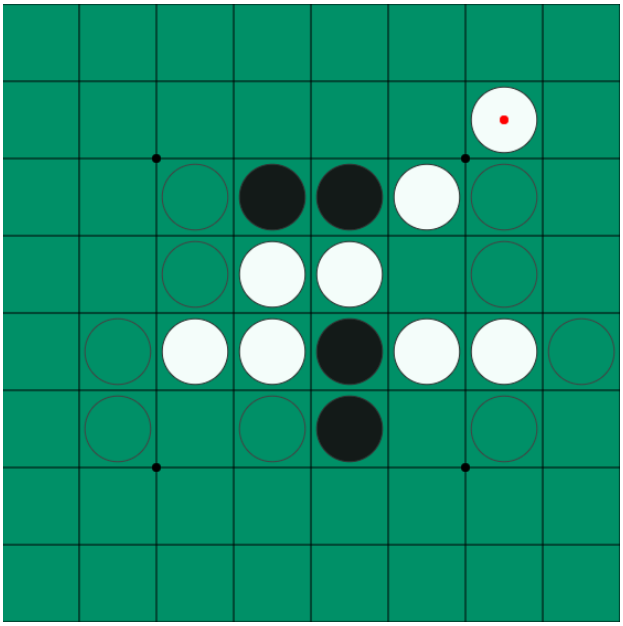
Una vez se ha explicado cómo se punta y se realiza la expansión de cada jugada, voy a explicar como se implementa el algoritmo MinMax. En primer lugar se determinan las posibles posiciones donde el jugador máquina puede colocar su próxima pieza. A continuación se le da una puntuación a cada posible posición en función de las fichas contrarias que va a cambiar de color. Una vez se determina al posición con más puntuación se procede a la expansión de la pieza.

- puntuar-expansión-pos: se encarga de puntuar cada una de las posibles expansiones de una determinada pieza.
- mejor-pos-expandir: a partir de la función anterior se determina la mejor posición donde colocar la pieza para que la expansión sea lo más grande posible. Esta función devuelve una lista con la posición, la dirección y la puntuación de la mejor jugada.
- realizar-expansion-minmax: una vez se ha determinado la mejor puntuación se realiza la expansion de la pieza a lo largo de la dirección determinada por la anterior función.

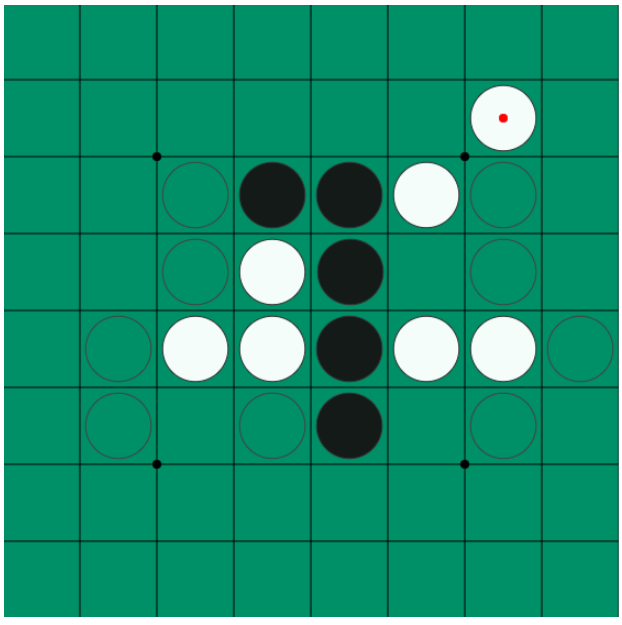
MEJORAS IMPLEMENTADAS

En cuanto a las mejoras implementadas en el código, en el Othello clásico cada jugador puede ir colocando fichas de su color de forma que se forme una fila, una columna o una diagonal entre sus fichas, cambiando así el color de las fichas del color contrario que hay entre ambas. En el Othello que he implementado además de esos tres movimientos posible, he añadido un cuarto movimiento, la alineación diagonal secundaria. Además de comprobar los tres movimientos del Othello clásico, el Othello implementado analiza la posibilidad de que el jugador pueda cambiar las fichas de color situadas en la diagonal secundaria.

Otra mejora es que en el Othello clásico la forma de cambio de color entre fichas es la siguiente:



Existen situaciones en las que hay fichas de distintos colores situadas de forma alterna como sucede en la quinta columna. En el Othello implementado no se contempla esta situación, de forma que si se da el caso en el que una ficha de un color está situada entre medias de fichas de otro color, esta ficha pasa de un color a otro, como se muestra a continuación.



FUNCIONAMIENTO DEL PROGRAMA

Al iniciar el programa se muestra un menú donde el usuario puede elegir tanto el modo de juego como el tipo de Othello con el que empezar.

```
Welcome to DrRacket, version 7.7 [3m].
Language: racket/gui, with debugging; memory limit: 128 MB.
-----BIENVENIDO AL JUEGO DE OTHELLO-----
Que modo de juego deseas jugar humano vs humano o humano vs maquina
humano vs humano
Indique el tablero con el que quiere comenzar a jugar (othello1, othello2, ..., othello5)othello1
JUGADOR 2
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 X 0 0 0 0
0 0 X 1 2 0 0 0
0 0 0 2 1 X 0 0
0 0 0 0 X 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
Determina la columna donde colocar tu
pieza
```

En el momento en el que se introduce un valor de entrada erróneo para el programa como otro modo de juego distinto u otro tablero inicial que no se contempla en el proyecto se muestra un mensaje de error.

```
Welcome to DrRacket, version 7.7 [3m].
Language: racket/gui, with debugging; memory limit: 128 MB.
-----BIENVENIDO AL JUEGO DE OTHELLO-----
Que modo de juego deseas jugar humano vs humano o humano vs maquina
otro modo de jeugo
Indique el tablero con el que quiere comenzar a jugar (othello1, othello2, ..., othello5)othello1
"Debe seleccionar un modo de juego correcto"
>
```

Una vez se ha seleccionado un modo de juego correcto se comienza a jugar. En primer lugar se imprime el tablero y se indica en al parte superior el turno del jugador al que le toca jugar.

En el modo de juego de jugador contra jugador, el primer turno es del jugador que comienza con fichas negras, en este caso representadas con el valor 2. En el modo de jugo jugador contra máquina, el primer movimiento siempre lo realiza el jugador.

```
JUGADOR 2
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 X 0 0 0 0
0 0 X 1 2 0 0 0
0 0 0 2 1 X 0 0
0 0 0 0 X 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
Determina la columna donde colocar tu
pieza
```

Una vez se imprime el tablero, se indican las posiciones donde el jugador puede colocar sus fichas, estas posiciones están marcadas con una cruz “X”. En el momento en el que jugador indica donde quiere colocar la ficha, se comprueba si la posición es válida, y se modifica el tablero en consecuencia.

```
JUGADOR 2
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 X 0 0 0 0
0 0 X 1 2 0 0 0
0 0 0 2 1 X 0 0
0 0 0 0 X 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
Determina la columna donde colocar tu pieza3
Determina la fila donde colocar tu pieza2
----------------------
JUGADOR 1
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 X 2 X 0 0 0
0 0 0 2 2 0 0 0
0 0 X 2 1 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
Determina la columna donde colocar tu
pieza
```

En este caso he seleccionado como posición la (3, 2), como podemos observar se ha cambiado el color de la columna y se ha paso el turno al jugador1. Los turno se van alternando hasta el momento en el que el tablero quede lleno. En la carpeta “test” hay un tablero lleno, voy a ejecutar el programa con ese tablero para comprobar el funcionamiento del algoritmo cuando una partida acaba. El tablero en cuestión es el siguiente:

```
1   ;partida 5
2   ;+-----------------+
3   ;| 1 2 2 2 2 2 2 2 |
4   ;| 1 2 1 2 1 2 2 1 |
5   ;| 1 1 2 2 1 1 2 1 |
6   ;| 1 2 2 2 2 2 1 1 |
7   ;| 1 2 1 1 2 2 2 1 |
8   ;| 1 1 1 1 2 1 2 1 |
9   ;| 1 2 1 2 2 2 1 1 |
10  ;| 2 2 2 2 1 1 2 1 |
11  ;+-----------------+
```

Como podemos observar al ejecutar el programa se cuenta el número de fichas de cada tipo y se determina el ganador, en este caso el jugador con fichas negras.

```
Welcome to DrRacket, version 7.7 [3m].
Language: racket/gui, with debugging; memory limit: 128 MB.
-----BIENVENIDO AL JUEGO DE OTHELLO-----
Que modo de juego deseas jugar humano vs humano o humano vs maquina
humano vs humano
Indique el tablero con el que quiere comenzar a jugar (othello1, othello2, ..., othello5)othello5
Gana la ficha 2
>
```

A continuación voy a ejecutar el programa aplicando el modo de juego jugador contra máquina. El primer turno lo realiza el jugador humano como ya he comentado anteriormente.

```
Welcome to DrRacket, version 7.7 [3m].
Language: racket/gui, with debugging; memory limit: 128 MB.
-----BIENVENIDO AL JUEGO DE OTHELLO-----
Que modo de juego deseas jugar humano vs humano o humano vs maquina
humano vs maquina
Indique el tablero con el que quiere comenzar a jugar (othello1, othello2, ..., othello5)othello1
JUGADOR
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 X 0 0 0 0
0 0 X 1 2 0 0 0
0 0 0 2 1 X 0 0
0 0 0 0 X 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
Determina la columna donde colocar tu
pieza
```

Una vez el jugador humano selecciona la posición donde quiere colocar su ficha, el jugador maquina selecciona al mejor posición onde colocar la suya.

```
MAQUINA
JUGADOR
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 2 2 2 0 0 0
0 0 1 1 1 0 0 0
0 X X X X X 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
Determina la columna donde colocar tu
pieza
```

Una vez que el jugador maquina ha colocado su ficha, podemos ver que en este caso la ha colocado en la posición (2, 4), el turno pasa al jugador humano.

A continuación voy a ejecutar otro Othello con la partida mas avanzada para comprobar que la elección de la máquina de realizar la mejor jugada se realiza correctamente.

```
Welcome to DrRacket, version 7.7 [3m].
Language: racket/gui, with debugging; memory limit: 128 MB.
-----BIENVENIDO AL JUEGO DE OTHELLO-----
Que modo de juego deseas jugar humano vs humano o humano vs maquina
humano vs maquina
Indique el tablero con el que quiere comenzar a jugar (othello1, othello2, ..., othello5)othello2
JUGADOR
1 X 0 0 0 0 0 0
0 0 1 0 0 X X 0
0 0 0 2 0 1 0 0
1 0 0 2 2 2 0 0
1 2 1 1 2 X 0 0
0 X 1 X 0 0 2 0
0 X 1 X 0 0 0 0
0 0 1 0 0 0 0 0
Determina la columna donde colocar tu pieza1
Determina la fila donde colocar tu pieza0
-----------------------
MAQUINA
JUGADOR
1 2 0 0 0 0 0 0
0 0 2 0 0 X X 0
0 0 0 2 0 1 0 0
1 0 0 2 2 2 0 0
1 1 1 1 1 1 0 0
0 X 1 X X X 2 0
0 X 1 0 0 0 0 X
0 0 1 0 0 0 0 0
Determina la columna donde colocar tu
pieza
```

En este caso he ejecutado el Othello2, comienza el jugador humano colocando su ficha en la posición (1, 0), a continuación la maquina analiza la mejor jugada.

La mejor jugada se determina de la siguiente manera. Se determinan todas las posibles posiciones donde se puede colocar la pieza y a continuación se da una puntuación a cada jugada determinando el número de fichas que pueden cambiar de color. La puntuación se determina de forma proporcional al número de fichas de color contrario que hay entre la posición donde coloca la ficha y la posición donde encuentra una ficha de su mismo color. En este caso la mejor opción para la máquina es colocar su ficha en la posición (5, 4) puesto que cambia de color a dos fichas de color contrario. Una vez que la maquina realiza su turno, el jugador humano vuelve a jugar.

Como podemos observar, ya he explicado antes que cuando existen fichas alternas de colores distintos y no se ha realizado ningún movimiento, el algoritmo lo detecta y las cambia de color de forma automática. Esto ocurre en la imagen mostrada anteriormente, si estuviésemos jugando al Othello clásico la única ficha que cambiaria su color seria la ficha negra de la posición (4, 4).

CODIGO FUENTE

(Othello.rkt)

```
#lang racket/gui


(require "funciones-aux.rkt")

(require "pos-disponibles.rkt")

(require 2htdp/batch-io)


#|

othello -> null

OBJ: imprimir el tablero del juego

PRE:

|#

(define (imprimir-othello othello)

 (cond

   [(empty? othello) (display "")]

   [else (imprimir-lista (car othello)) (display "\n") (imprimir-othello (cdr othello))]))
```

```
#|
lista -> number
OBJ: calcular el tamanno de una lista
PRE:
|#
(define (len lst)
 (cond
   [(empty? lst)  0]
   [(cons? lst)   (+ 1 (length (rest lst)))]))

#|
num, lista -> boolean
OBJ: determinar si un numero esta en una lista
PRE:
|#
(define (member? item seq)
 (sequence-ormap (lambda (x)
                   (equal? item x))
                 seq))

#|
lista -> null
OBJ: imprimir una lista dada
PRE:
|#
(define (imprimir-lista lista)
 (cond
   [(empty? lista) (display "")]
   [else (display (car lista)) (display " ") (imprimir-lista (cdr lista))]))

#|
```

```
lista -> lista
OBJ: eliminar los negativos de una lista
PRE:
|#
(define (eliminar-negativos lista)
 (cond
   [(empty? lista) '()]
   [(pair? (car lista)) (cons (car lista) (eliminar-negativos (cdr lista)))]
   [else (eliminar-negativos (cdr lista))]))

#|
lista -> lista
OBJ: eliminar los ceros de una lista
PRE:
|#
(define (eliminar-ceros lista)
 (cond
   [(empty? lista) '()]
   [(equal? (car lista) 0) (eliminar-ceros (cdr lista))]
   [else (cons (car lista) (eliminar-ceros (cdr lista)))]))

#|
othello -> boolean
OBJ: determinar si el othello esta lleno
PRE:
|#
(define (othello-lleno? othello)
 (cond
   [(empty? othello) true]
   [(member 0 (car othello)) false]
   [else (othello-lleno? (cdr othello))]))
```

```
#|
othello -> null
OBJ: determinar el ganador del juego
PRE:
|#
(define (determinar-ganador othello)
  (cond
    [(> (contar-pieza othello 2) (contar-pieza othello 1)) (display "Gana la ficha 2")]
    [(< (contar-pieza othello 2) (contar-pieza othello 1)) (display "Gana la ficha 1")]
    [else (display "Empate")]))

#|
lista -> lista
OBJ: eliminar las listas innecesarias
PRE: len lista > 9
|#
(define (delete-lists lista)
  (cond
    [(equal? (len lista) 8) lista]
    [(member ";|" (car lista)) (delete-lists (append (cdr lista) (list (car lista))))]
    [else (delete-lists (cdr lista))]))

#|
lista -> lista
OBJ: limpiar una lisra para su posterior lectura
PRE: len lista > 9
|#
(define (limpiar-lista-dos lista)
  (cond
    [(equal? (len lista) 8) lista]
```

```
   [(or (equal? "|" (car lista))(equal?  ";|" (car lista))) (limpiar-lista-dos (cdr lista))]
   [else (limpiar-lista-dos (append (cdr lista) (list (string->number(car lista)))))]))

#|
lista, number -> lista
OBJ: limpiar un othello completo
PRE:
|#
(define (limpiar-othello lista rep)
 (cond
   [(equal? rep 0) lista]
   [else (limpiar-othello (append (cdr lista) (list (limpiar-lista-dos (car lista)))) (- rep 1))]))

#|
othello, numero -> numero
OBJ: contar el numero de apariciones de una pieza en el tablero
PRE:
|#
(define (contar-pieza othello pieza)
 (cond
   [(empty? othello) 0]
   [else (+ (contar-pieza-lista (car othello) pieza) (contar-pieza (cdr othello) pieza))]))

#|
lista, numero -> numero
OBJ: contar el numero de apariciones de una pieza
PRE:
|#
(define (contar-pieza-lista lista pieza)
 (cond
   [(empty? lista) 0]
```

```
   [(equal? (car lista) pieza) (+ 1 (contar-pieza-lista (cdr lista) pieza))]
   [else (contar-pieza-lista (cdr lista) pieza)]))

#|
othello, numero -> lista
OBJ: devolver una lista sin ceros y sin fichas sueltas
PRE: pieza = 1/2
|#
(define (limpiar-lista lista pieza)
  (cond
   [(and (equal? (length lista) 1) (equal? (car lista) pieza)) (list pieza)]
   [(equal? (length lista) 1) '()]
   [(and (equal? (car lista) (get-pieza-contraria pieza)) (equal? (car (cdr lista)) (get-pieza-
contraria pieza))) (limpiar-lista (cdr lista) pieza)]
   [else (cons (car lista) (limpiar-lista (cdr lista) pieza))]
   ))

#|
othello -> boolean
OBJ: determinar si un tablero es valido, un tablero solo sera valido si contiene unicamente
ceros, unos y doses
PRE: len(othello) = 8 x 8 and  -1 < othello[i][j] < 10
|#
(define (othello-valido? othello)
  (cond
   [(empty? othello) true]
   [(or (member 3 (car othello)) (member 4 (car othello)) (member 5 (car othello)) (member 6
(car othello)) (member 7 (car othello)) (member 8 (car othello)) (member 9 (car othello))) false]
   [else (othello-valido? (cdr othello))]))

#|
othello, number -> lista
```

```
OBJ: devolver las apariciones de una pieza en el tablero
PRE: pieza = 1/2
|#
(define (apariciones-pieza othello pieza)
 (for*/list ([i 8]
             [j 8]
             #:when (equal? pieza (get-elem i j othello))) (list i j)))

#|
othello, lista -> lista
OBJ: determinar las posibilidades que tiene una pieza en concreto
PRE:
|#
(define (pos-disponibles-pieza othello pos)
 (delete-voids (list
                (pos-disponibles-fila othello pos)
                (pos-disponibles-columna othello pos)
                (pos-disponibles-diagonal othello pos)
                (pos-disponibles-diagonal-secun othello pos))))

#|
othello, lista -> lista
OBJ: determinar las posibilidades que tiene un jugador de colocar su proxima pieza
PRE:
|#
(define (posibilidades-jugador othello lista-apariciones-pieza)
 (cond
   [(empty? lista-apariciones-pieza) '()]
   [else (append (pos-disponibles-pieza othello (car lista-apariciones-pieza)) (posibilidades-
jugador othello (cdr lista-apariciones-pieza)))]))
```

```
#|

othello, lista -> null

OBJ: imprimir el tablero marcando las posibilidades que tiene un jugador de colocar su
proxima ficha

PRE:

|#

(define (imprimir-tablero-posibles othello lista-posibles)

 (cond

   [(empty? lista-posibles) (imprimir-othello othello)]

   [else (imprimir-tablero-posibles (set-elem (car (car lista-posibles)) (car (cdr (car lista-
posibles))) othello "X") (cdr lista-posibles))]))


#|

lista -> boolean

OBJ: determinar si existe una posible expansion de ficha

PRE:

|#

(define (posible-expansion? lista pieza)

 (cond

   [(< (length lista) 3) false]

   [(and (equal? (list-ref lista 0) pieza) (equal? (list-ref lista 1) (get-pieza-contraria pieza))
(equal? (list-ref lista 2) pieza)) true]

   [else (posible-expansion? (cdr lista) pieza)]))


#|

lista, numero -> numero

OBJ: calcular el numero de fichas que se pueden expandir

PRE:

|#

(define (puntuar-expansion lista numero)

 (cond

   [(equal? (len lista) 1) lista]
```

```
   [(or (equal? (list-ref lista 0) 0) (equal? (list-ref lista 0) (get-pieza-contraria numero)))
(puntuar-expansion (cdr lista) numero)]

   [(and (equal? (list-ref lista 0) numero) (or (equal? (list-ref lista 1) numero) (equal? (list-ref
lista 1) 0))) (puntuar-expansion (cdr lista) numero)]

   [else lista]))


#|

othello, lista -> numero

OBJ: calcular la expansion de una posicion en fila

PRE: pos debe estar dentro de los limites del tablero

|#

(define (puntuar-expansion-fila othello pos)

 (cond

   [(member? 0 (reverse (puntuar-expansion (reverse (puntuar-expansion (get-fila (set-elem
(car pos) (last pos) othello 1) (last pos)) 1)) 1))) 1]

   [else (len (reverse (puntuar-expansion (reverse (puntuar-expansion (get-fila (set-elem (car
pos) (last pos) othello 1) (last pos)) 1)) 1)))]))


#|

othello, lista -> numero

OBJ: calcular la expansion de una posicion en columna

PRE: pos debe estar dentro de los limites del tablero

|#

(define (puntuar-expansion-columna othello pos)

 (cond

   [(member? 0 (reverse (puntuar-expansion (reverse (puntuar-expansion (get-columna (set-
elem (car pos) (last pos) othello 1) (car pos)) 1)) 1))) 1]

   [else (len (reverse (puntuar-expansion (reverse (puntuar-expansion (get-columna (set-elem
(car pos) (last pos) othello 1) (car pos)) 1)) 1)))]))


#|

othello, lista -> numero

OBJ: calcular la expansion de una posicion en diagonal
```

```
PRE: pos debe estar dentro de los limites del tablero

|#

(define (puntuar-expansion-diagonal othello pos)

 (cond

  [(member? 0 (reverse (puntuar-expansion (reverse (puntuar-expansion (get-diagonal (set-elem (car pos) (last pos) othello 1) (car pos) (last pos)) 1)) 1))) 1]

  [else (len (reverse (puntuar-expansion (reverse (puntuar-expansion (get-diagonal (set-elem (car pos) (last pos) othello 1) (car pos) (last pos)) 1)) 1)))]))


#|

othello, lista -> numero

OBJ: calcular la expansion de una posicion en diagonal secundaria

PRE: pos debe estar dentro de los limites del tablero

|#

(define (puntuar-expansion-diagonal-secundaria othello pos)

 (cond

  [(member? 0 (reverse (puntuar-expansion (reverse (puntuar-expansion (get-diagonal-secundaria (set-elem (car pos) (last pos) othello 1) (car pos) (last pos)) 1)) 1))) 1]

  [else (len (reverse (puntuar-expansion (reverse (puntuar-expansion (get-diagonal-secundaria (set-elem (car pos) (last pos) othello 1) (car pos) (last pos)) 1)) 1)))]))


#|

othello, lista -> boolean

OBJ: determina cual es la mejor expansion de una posicion

PRE: pos debe estar dentro de los limites del tablero

|#

(define (puntuar-expansion-pos othello pos)

 (cond

  [(and (> (puntuar-expansion-fila othello pos) (puntuar-expansion-columna othello pos)) (> (puntuar-expansion-fila othello pos) (puntuar-expansion-diagonal othello pos)) (> (puntuar-expansion-fila othello pos) (puntuar-expansion-diagonal-secundaria othello pos))) (list (puntuar-expansion-fila othello pos) "fila" pos)]

  [(and (> (puntuar-expansion-columna othello pos) (puntuar-expansion-fila othello pos)) (> (puntuar-expansion-columna othello pos) (puntuar-expansion-diagonal othello pos)) (>
```

```
(puntuar-expansion-columna othello pos) (puntuar-expansion-diagonal-secundaria othello pos))) (list (puntuar-expansion-columna othello pos) "columna" pos)]

   [(and (> (puntuar-expansion-diagonal othello pos) (puntuar-expansion-columna othello pos)) (> (puntuar-expansion-diagonal othello pos) (puntuar-expansion-fila othello pos)) (> (puntuar-expansion-diagonal othello pos) (puntuar-expansion-diagonal-secundaria othello pos))) (list (puntuar-expansion-diagonal othello pos) "diagonal" pos)]

   [(and (> (puntuar-expansion-diagonal-secundaria othello pos) (puntuar-expansion-fila othello pos)) (> (puntuar-expansion-diagonal-secundaria othello pos) (puntuar-expansion-columna othello pos)) (> (puntuar-expansion-diagonal-secundaria othello pos) (puntuar-expansion-diagonal othello pos))) (list (puntuar-expansion-diagonal-secundaria othello pos) "diagonal-secundaria" pos)]))


#|

lista, lista, numero -> lista

OBJ: expandir una pieza  alo largo de una lista

PRE: lista-aux = '() and pieza = 1/2

|#

(define (mejor-pos-expandir othello lista-max lista-posibles)

 (cond

   [(empty? lista-posibles) lista-max]

   ;[(display (puntuar-expansion-pos othello (car lista-posibles))) (display "\n")(mejor-pos-expandir othello lista-max (cdr lista-posibles))]

   [(> (car (puntuar-expansion-pos othello (car lista-posibles))) (car lista-max)) (mejor-pos-expandir othello (puntuar-expansion-pos othello (car lista-posibles)) (cdr lista-posibles))]

   [else (mejor-pos-expandir othello lista-max (cdr lista-posibles))]))


#|

lista, lista, numero -> lista

OBJ: expandir una pieza  alo largo de una lista

PRE: lista-aux = '() and pieza = 1/2

|#

(define (expandir-pieza lista-aux lista pieza)

 (cond

   [(empty? lista) lista-aux]
```

```
   [(and (equal? (car lista) (get-pieza-contraria pieza)) (equal? (last lista-aux) pieza) (and (> (length lista) 1)) (or (equal? (car (cdr lista)) (get-pieza-contraria pieza)) (equal? (car (cdr lista)) pieza))) (expandir-pieza (append lista-aux (list pieza)) (cdr lista) pieza)]

   [(or (equal? (car lista) 0) (equal? (car lista) pieza) (equal? (car lista) (get-pieza-contraria pieza))) (expandir-pieza (append lista-aux (list (car lista))) (cdr lista) pieza)]

   [else (expandir-pieza lista-aux (cdr lista) pieza)]))


#|

othello, lista, string -> othello

OBJ: realiza la expansion del jugador

PRE:

|#

(define (realizar-expansion othello pos)

 (cond

   [(posible-expansion? (limpiar-lista (get-fila othello (last pos)) (get-elem (car pos) (last pos) othello)) (get-elem (car pos) (last pos) othello)) (set-fila othello (last pos) (cdr (expandir-pieza '(0) (get-fila othello (last pos)) (get-elem (car pos) (last pos) othello))))]

   [(posible-expansion? (limpiar-lista (get-columna othello (car pos)) (get-elem (car pos) (last pos) othello)) (get-elem (car pos) (last pos) othello)) (set-columna othello (car pos) (cdr (expandir-pieza '(0) (get-columna othello (car pos)) (get-elem (car pos) (last pos) othello))))]

   [(posible-expansion? (limpiar-lista (get-diagonal othello (car pos) (last pos)) (get-elem (car pos) (last pos) othello)) (get-elem (car pos) (last pos) othello)) (set-diagonal othello (car (get-primera-pos-diagonal (car pos) (last pos))) (last (get-primera-pos-diagonal (car pos) (last pos))) (cdr (expandir-pieza '(0) (get-diagonal othello (car pos) (last pos)) (get-elem (car pos) (last pos) othello))))]

   [(posible-expansion? (limpiar-lista (get-diagonal-secundaria othello (car pos) (last pos)) (get-elem (car pos) (last pos) othello)) (get-elem (car pos) (last pos) othello)) (set-diagonal-secundaria othello (car (get-primera-pos-diagonal-secun (car pos) (last pos))) (last (get-primera-pos-diagonal-secun (car pos) (last pos))) (cdr (expandir-pieza '(0) (get-diagonal-secundaria othello (car pos) (last pos)) (get-elem (car pos) (last pos) othello))))]

   [else (display "No hay expansion?")]))


#|

othello, lista, string -> othello

OBJ: realiza la expansion de la maquina

PRE:
```

```
|#
(define (realizar-expansion-minmax othello pos direccion)
 (cond
  [(equal? direccion "fila") (set-fila othello (last pos) (cdr (expandir-pieza '(0) (get-fila othello (last pos)) (get-elem (car pos) (last pos) othello))))]
  [(equal? direccion "columna") (set-columna othello (car pos) (cdr (expandir-pieza '(0) (get-columna othello (car pos)) (get-elem (car pos) (last pos) othello))))]
  [(equal? direccion "diagonal") (set-diagonal othello (car (get-primera-pos-diagonal (car pos) (last pos))) (last (get-primera-pos-diagonal (car pos) (last pos))) (cdr (expandir-pieza '(0) (get-diagonal othello (car pos) (last pos)) (get-elem (car pos) (last pos) othello))))]
  [(equal? direccion "diagonal-secundaria") (set-diagonal-secundaria othello (car (get-primera-pos-diagonal-secun (car pos) (last pos))) (last (get-primera-pos-diagonal-secun (car pos) (last pos))) (cdr (expandir-pieza '(0) (get-diagonal-secundaria othello (car pos) (last pos)) (get-elem (car pos) (last pos) othello))))]))

#|
othello, lista, numero -> othello
OBJ: colocar una pieza en una determinada posicion del tablero
PRE: pieza = 1/2
|#
(define (colocar-pieza othello pos pieza)
 (set-elem (car pos) (last pos) othello pieza))

#|
othello, numero -> othello
OBJ: realiza el turno del jugador
PRE: jugador = 1/2
|#
(define (realizar-turno-jugador othello jugador)
 (imprimir-tablero-posibles othello (eliminar-negativos (posibilidades-jugador othello (apariciones-pieza othello jugador))))
 (display "Determina la columna donde colocar tu pieza")
 (define columna (read-line (current-input-port) 'any))
 (display "Determina la fila donde colocar tu pieza")
```

```
  (define fila (read-line (current-input-port) 'any))
  (display "---------------------- \n")
  (realizar-expansion (colocar-pieza othello (list (string->number columna) (string->number fila)) jugador) (list (string->number columna) (string->number fila))))

#|
othello-> othello
OBJ: realiza el turno de la maquina
PRE:
|#
(define (realizar-turno-maquina othello)
  (realizar-expansion-minmax (colocar-pieza othello (last (mejor-pos-expandir othello (list 0 "" '(0 0)) (eliminar-negativos (posibilidades-jugador othello (apariciones-pieza othello 1))))) 1) (last (mejor-pos-expandir othello (list 0 "" '(0 0)) (eliminar-negativos (posibilidades-jugador othello (apariciones-pieza othello 1))))) (second (mejor-pos-expandir othello (list 0 "" '(0 0)) (eliminar-negativos (posibilidades-jugador othello (apariciones-pieza othello 1)))))))

#|
othello, numero -> othello
OBJ: representa una partida humano vs humano
PRE: turno-jugador = 1/2
|#
(define (jugar-othello-humano othello turno-jugador)
  (cond
    [(othello-lleno? othello) (determinar-ganador othello)]
    [(equal? turno-jugador 1) (display "JUGADOR 1 \n")(jugar-othello-humano (realizar-turno-jugador othello turno-jugador) 2)]
    [else (display "JUGADOR 2 \n") (jugar-othello-humano (realizar-turno-jugador othello turno-jugador) 1)]))

#|
othello, numero -> othello
OBJ: representa una partida humano vs maquina
PRE:
```

```
|#
(define (jugar-othello-maquina othello turno)
  (cond
    [(othello-lleno? othello) (determinar-ganador othello)]
    [(equal? turno 1) (display "MAQUINA \n") (jugar-othello-maquina (realizar-turno-maquina
othello) 2)]
    [else (display "JUGADOR \n") (jugar-othello-maquina (realizar-turno-jugador othello 2) 1)]))


#|
nil -> nil
OBJ: interfaz de usuario
PRE:
|#
(define (iniciar-othello)
  (display "-----BIENVENIDO AL JUEGO DE OTHELLO----- \n")
  (display "Que modo de juego deseas jugar humano vs humano o humano vs maquina \n")
  (define modo-juego (read-line (current-input-port) 'any))
  (display "Indique el tablero con el que quiere comenzar a jugar (othello1, othello2, ...,
othello5)")
  (define tablero (read-line (current-input-port) 'any))
  (cond
    [(equal? modo-juego "humano vs humano") (jugar-othello-humano (limpiar-othello (delete-
lists (read-words/line (string-append "./test/" tablero ".txt"))) 8) 2)]
    [(equal? modo-juego "humano vs maquina") (jugar-othello-maquina (limpiar-othello (delete-
lists (read-words/line (string-append "./test/" tablero ".txt"))) 8) 2)]
    [else "Debe seleccionar un modo de juego correcto"]))
```

```
(define test1
  '((2 2 2 2 2 2 0 2)
    (0 0 0 2 2 0 0 0)
    (2 0 2 0 0 2 2 0)
    (0 2 0 2 0 2 2 1)
    (2 0 2 2 2 0 1 2)
    (0 0 1 0 0 0 0 0)
    (1 0 0 0 2 0 0 0)
    (0 0 0 0 0 0 0 0)))

;(eliminar-negativos (posibilidades-jugador test1 (apariciones-pieza test1 1)))

;(imprimir-tablero-posibles test1 (eliminar-negativos (posibilidades-jugador test1 (apariciones-pieza test1 1))))

;(mejor-pos-expandir test1 (list 0 "" '(0 0)) (eliminar-negativos (posibilidades-jugador test1 (apariciones-pieza test1 1))))

;(imprimir-othello (realizar-expansion-minmax (colocar-pieza test1 (last (mejor-pos-expandir test1 (list 0 "" '(0 0)) (eliminar-negativos (posibilidades-jugador test1 (apariciones-pieza test1 1))))) 1) (last (mejor-pos-expandir test1 (list 0 "" '(0 0)) (eliminar-negativos (posibilidades-jugador test1 (apariciones-pieza test1 1))))) (second (mejor-pos-expandir test1 (list 0 "" '(0 0)) (eliminar-negativos (posibilidades-jugador test1 (apariciones-pieza test1 1)))))))

;(display (realizar-turno-maquina test1))

;(jugar-othello-maquina test1 2)
```

```
;(realizar-turno-jugador test1 2)
;(jugar-othello test1 2)


(iniciar-othello)

()
```

(funciones-aux.rkt)

```
#lang racket

(provide (all-defined-out))

#|
numero, numero, othello -> numero
OBJ: devolver el valor de una posicion del othello
PRE: 0 < x-coor and y-coor < 9
|#
(define (get-elem x-coor y-coor othello)
  (list-ref (list-ref othello y-coor) x-coor))

#|
numero, numero, othello, numero  -> numero
OBJ: modificar un valor determinado del tablero
PRE: 0 < x-coor and y-coor < 9
|#
(define (set-elem x-coor y-coor othello value)
  (cond
    [(equal? y-coor 0) (cons (list-set (list-ref othello y-coor) x-coor value) (cdr othello))]
    [else (cons (car othello)(set-elem x-coor (- y-coor 1) (cdr othello) value))]))
```

```
#|
othello, numero -> lista
OBJ: devolver una determinada fila del tablero
PRE: 0 < fila < 8
|#
(define (get-fila othello fila)
  (list-ref othello fila))

#|
othello, numero, lista -> othello
OBJ: modificar una fila del othello
PRE: 0 < fila < 8 and lentgth(lista) = 8
|#
(define (set-fila othello fila lista)
  (cond
    [(empty? lista) othello]
    [else (set-fila (set-elem (- 8 (length lista)) fila othello (car lista)) fila (cdr lista))]))

#|
numero, othello -> lista
OBJ: obtener una columna determinada del tablero
PRE: columna < len(lista)
|#
(define (get-columna othello columna)
  (cond
    [(empty? othello) '()]
    [else (cons (get-num-columna columna (car othello)) (get-columna (cdr othello) columna))]))

#|
othello, numero, lista -> othello
OBJ: modificar una columna del othello
```

```
PRE: 0 < columna < 8 and lentgth(lista) = 8

|#

(define (set-columna othello columna lista)

 (cond

   [(empty? lista) othello]

   [else (set-columna (set-elem columna (- 8 (length lista)) othello (car lista)) columna (cdr
lista))]))


#|

numero, numero, lista -> numero

OBJ: obtiene un numero de una lista dada

PRE: indice < len(lista)

|#

(define (get-num-columna indice lista)

 (cond

   [(equal? indice 0) (car lista)]

   [else (get-num-columna (- indice 1) (cdr lista))]))


#|

numero -> numero

OBJ: devolover la pieza contraria a la dada

PRE: pieza = 1/2

|#

(define (get-pieza-contraria pieza)

 (cond

   [(equal? pieza 1) 2]

   [(equal? pieza 2) 1]))


#|

othello, numero, numero -> lista

OBJ: obtener la diagonal principal del othello
```

```
PRE: 0 < fila and columna < 9

|#

(define (get-diagonal othello columna fila)

 (cond

   [(> 0 (- fila columna)) (get-diagonal-aux othello columna fila)]

   [else (get-diagonal-aux (trasponer-othello othello) fila columna)]))


(define (get-diagonal-aux othello columna fila)

 (cond

   [(equal? 8 (- columna fila)) '()]

   [else (cons (list-ref (car othello) (- columna fila)) (get-diagonal-aux  (cdr othello) (+ columna
1) fila ))]))


#|

othello, numero, numero, lista -> othello

OBJ: modificar una diagonal del tablero

PRE: 0 < fila and columna < 9 and len(lista) = len(diagonal)

|#

(define (set-diagonal othello columna fila lista)

 (cond

   [(empty? lista) othello]

   [else (set-diagonal (set-elem columna fila othello (car lista)) (+ 1 columna) (+ fila 1) (cdr
lista))]))


#|

numero, numero -> lista

OBJ: obtener la posicion inicial de una diagonal

PRE:

|#

(define (get-primera-pos-diagonal columna fila)

 (cond

   [(> columna fila) (list (- columna fila) 0)]
```

```
   [else (list 0 (- fila columna))]))

#|
othello, numero, numero -> lista
OBJ: obtener la diagonal secundaria del othello
PRE: 0 < fila and columna < 9
|#
(define (get-diagonal-secundaria othello columna fila)
  (get-diagonal (invertir-othello othello) (- 7 columna) fila))

#|
numero, numero -> lista
OBJ: obtener la posicion inicial de una diagonal secundaria
PRE:
|#
(define (get-primera-pos-diagonal-secun columna fila)
  (cond
    [(or (equal? columna 7) (equal? fila 0)) (list columna fila)]
    [else (get-primera-pos-diagonal-secun (+ columna 1) (- fila 1))]))

#|
othello, numero, numero, lista -> othello
OBJ: modificar una diagonal secundaria del tablero
PRE: 0 < fila and columna < 9 and len(lista) = len(diagonal)
|#
(define (set-diagonal-secundaria othello columna fila lista)
  (cond
    [(empty? lista) othello]
    [else (set-diagonal-secundaria (set-elem columna fila othello (car lista)) (- columna 1) (+ fila
1) (cdr lista))]))
```

```
#|
othello -> othello
OBJ: transponer el tablero
PRE:
|#
(define trasponer-othello
 (lambda (xss)
   (cond
    [(empty? xss)        empty]
    [(empty? (first xss)) empty]
    [else              (define first-column   (map first xss))
                       (define other-columns  (map rest  xss))
                       (cons first-column

                            (trasponer-othello other-columns))])))

#|
othello -> othello
OBJ: invertir el tablero
PRE:
|#
(define (invertir-othello othello)
 (cond
   [(empty? othello) '()]
   [else (cons (reverse (car othello)) (invertir-othello (cdr othello)))]))

#|
elem -> lista
OBJ: determinar si la entradad dada es una lista, en caso contrario devolver lista vacia
PRE:
|#
```

```
(define (delete-voids l)
 (cond
   [(empty? l) '()]
   [(pair? (car l)) (append (car l)(delete-voids (cdr l)))]
   [else (delete-voids (cdr l))]))
```

(pos-disponibles.rkt)

```
#lang racket

(require "funciones-aux.rkt")

(provide (all-defined-out))

#|
othello, lista -> lista
OBJ: determinar las posiciones disponibles de una pieza a lo largo de su fila por la izquierda
PRE:
|#
(define (pos-disponibles-fila-izq othello pos)
 (cond
   [(equal? -1 (car pos)) -1]
   [(equal? 0 (get-elem (car pos) (last pos) othello)) pos]
   [else (pos-disponibles-fila-izq othello (list (- (car pos) 1) (last pos)))]))

#|
othello, lista -> lista
OBJ: determinar las posiciones disponibles de una pieza a lo largo de su fila por la derecha
PRE:
|#
(define (pos-disponibles-fila-der othello pos)
 (cond
```

```
   [(equal? 8 (car pos)) -1]
   [(equal? 0 (get-elem (car pos) (last pos) othello)) pos]
   [else (pos-disponibles-fila-der othello (list (+ (car pos) 1) (last pos)))]))

#|
othello, lista -> lista
OBJ: determinar las posiciones disponibles de una pieza a lo largo de su fila
PRE:
|#
(define (pos-disponibles-fila othello pos)
 (cond
   [(equal? (car pos) 0)
    (cond
     [(equal? (get-pieza-contraria (get-elem (car pos) (last pos) othello)) (get-elem (+ (car pos) 1) (last pos) othello)) (list (pos-disponibles-fila-der othello pos))])]
   [(equal? (car pos) 7)
    (cond
     [(equal? (get-pieza-contraria (get-elem (car pos) (last pos) othello)) (get-elem (- (car pos) 1) (last pos) othello)) (list (pos-disponibles-fila-izq othello pos))])]
   [else
    (cond
     [(and (equal? (get-pieza-contraria (get-elem (car pos) (last pos) othello)) (get-elem (- (car pos) 1) (last pos) othello)) (equal? (get-pieza-contraria (get-elem (car pos) (last pos) othello)) (get-elem (+ (car pos) 1) (last pos) othello))) (list (pos-disponibles-fila-izq othello pos) (pos-disponibles-fila-der othello pos))]
     [(equal? (get-pieza-contraria (get-elem (car pos) (last pos) othello)) (get-elem (- (car pos) 1) (last pos) othello)) (list (pos-disponibles-fila-izq othello pos))]
     [(equal? (get-pieza-contraria (get-elem (car pos) (last pos) othello)) (get-elem (+ (car pos) 1) (last pos) othello)) (list (pos-disponibles-fila-der othello pos))])]))

#|
othello, lista -> lista
OBJ: determinar las posiciones disponibles de una pieza a lo largo de su columna por arriba
PRE:
```

```
|#
(define (pos-disponibles-columna-arriba othello pos)
 (cond
   [(equal? -1 (last pos)) -1]
   [(equal? 0 (get-elem (car pos) (last pos) othello)) pos]
   [else (pos-disponibles-columna-arriba othello (list (car pos) (- (last pos) 1)))]))

#|
othello, lista -> lista
OBJ: determinar las posiciones disponibles de una pieza a lo largo de su columna por abajo
PRE:
|#
(define (pos-disponibles-columna-abajo othello pos)
 (cond
   [(equal? 8 (last pos)) -1]
   [(equal? 0 (get-elem (car pos) (last pos) othello)) pos]
   [else (pos-disponibles-columna-abajo othello (list (car pos) (+ (last pos) 1)))]))

#|
othello, lista -> lista
OBJ: determinar las posiciones disponibles de una pieza a lo largo de su columna
PRE:
|#
(define (pos-disponibles-columna othello pos)
 (cond
   [(equal? 0 (last pos))
    (cond
     [(equal? (get-pieza-contraria (get-elem (car pos) (last pos) othello)) (get-elem (car pos) (+
(last pos) 1) othello)) (list (pos-disponibles-columna-abajo othello pos))])]
   [(equal? 7 (last pos))
    (cond
```

```
      [(equal? (get-pieza-contraria (get-elem (car pos) (last pos) othello)) (get-elem (car pos) (-
(last pos) 1) othello)) (list (pos-disponibles-columna-arriba othello pos))])]

   [else

    (cond

      [(and (equal? (get-pieza-contraria (get-elem (car pos) (last pos) othello)) (get-elem (car
pos) (- (last pos) 1) othello)) (equal? (get-pieza-contraria (get-elem (car pos) (last pos) othello))
(get-elem (car pos) (+ (last pos) 1) othello))) (list (pos-disponibles-columna-arriba othello pos)
(pos-disponibles-columna-abajo othello pos))]

      [(equal? (get-pieza-contraria (get-elem (car pos) (last pos) othello)) (get-elem (car pos) (+
(last pos) 1) othello)) (list (pos-disponibles-columna-abajo othello pos))]

      [(equal? (get-pieza-contraria (get-elem (car pos) (last pos) othello)) (get-elem (car pos) (-
(last pos) 1) othello)) (list (pos-disponibles-columna-arriba othello pos))])]))


#|

othello, lista -> lista

OBJ: determinar las posiciones disponibles de una pieza a lo largo de su diagonal principal por
abajo/derecha

PRE:

|#

(define (pos-disponibles-diag-abajo-der othello pos)

 (cond

   [(or (equal? 8 (car pos)) (equal? 8 (last pos))) -1]

   [(equal? 0 (get-elem (car pos) (last pos) othello)) pos]

   [else (pos-disponibles-diag-abajo-der othello (list (+ (car pos) 1) (+ (last pos) 1)))]))


#|

othello, lista -> lista

OBJ: determinar las posiciones disponibles de una pieza a lo largo de su diagonal principal por
arriba/izquierda

PRE:

|#

(define (pos-disponibles-diag-arriba-izq othello pos)

 (cond

   [(or (equal? -1 (car pos)) (equal? -1 (last pos))) -1]
```

```
   [(equal? 0 (get-elem (car pos) (last pos) othello)) pos]
   [else (pos-disponibles-diag-arriba-izq othello (list (- (car pos) 1) (- (last pos) 1)))]))

#|
othello, lista -> lista
OBJ: determinar las posiciones disponibles de una pieza a lo largo de su diagonal principal
PRE:
|#
(define (pos-disponibles-diagonal othello pos)
 (cond
   [(or (and (equal? 7 (car pos)) (equal? 0 (last pos))) (and (equal? 0 (car pos)) (equal? 7 (last pos)))) -1]
   [(or (equal? 0 (car pos)) (equal? 0 (last pos)))
    (cond
     [(equal? (get-pieza-contraria (get-elem (car pos) (last pos) othello)) (get-elem (+ (car pos) 1) (+ (last pos) 1) othello)) (list (pos-disponibles-diag-abajo-der othello pos))])]
   [(or (equal? 7 (car pos)) (equal? 7 (last pos)))
    (cond
     [(equal? (get-pieza-contraria (get-elem (car pos) (last pos) othello)) (get-elem (- (car pos) 1) (- (last pos) 1) othello)) (list (pos-disponibles-diag-arriba-izq othello pos))])]
   [else
    (cond
     [(and (equal? (get-pieza-contraria (get-elem (car pos) (last pos) othello)) (get-elem (- (car pos) 1) (- (last pos) 1) othello)) (equal? (get-pieza-contraria (get-elem (car pos) (last pos) othello)) (get-elem (+ (car pos) 1) (+ (last pos) 1) othello))) (list (pos-disponibles-diag-arriba-izq othello pos) (pos-disponibles-diag-abajo-der othello pos))]
     [(equal? (get-pieza-contraria (get-elem (car pos) (last pos) othello)) (get-elem (+ (car pos) 1) (+ (last pos) 1) othello)) (list (pos-disponibles-diag-abajo-der othello pos))]
     [(equal? (get-pieza-contraria (get-elem (car pos) (last pos) othello)) (get-elem (- (car pos) 1) (- (last pos) 1) othello)) (list (pos-disponibles-diag-arriba-izq othello pos))])]))

#|
othello, lista -> lista
```

OBJ: determinar las posiciones disponibles de una pieza a lo largo de su diagonal secundaria por abajo/izquierda

PRE:

|#

```
(define (pos-disponibles-diag-secun-abajo-izq othello pos)
 (cond
  [(or (equal? -1 (car pos)) (equal? 8 (last pos))) -1]
  [(equal? 0 (get-elem (car pos) (last pos) othello)) pos]
  [else (pos-disponibles-diag-secun-abajo-izq othello (list (- (car pos) 1) (+ (last pos) 1)))]))
```

#|

othello, lista -> lista

OBJ: determinar las posiciones disponibles de una pieza a lo largo de su diagonal secundaria por arriba/derecha

PRE:

|#

```
(define (pos-disponibles-diag-secun-arriba-der othello pos)
 (cond
  [(or (equal? 8 (car pos)) (equal? -1 (last pos))) -1]
  [(equal? 0 (get-elem (car pos) (last pos) othello)) pos]
  [else (pos-disponibles-diag-secun-arriba-der othello (list (+ (car pos) 1) (- (last pos) 1)))]))
```

#|

othello, lista -> lista

OBJ: determinar las posiciones disponibles de una pieza a lo largo de su diagonal secundaria

PRE:

|#

```
(define (pos-disponibles-diagonal-secun othello pos)
 (cond
  [(or (and (equal? 7 (car pos)) (equal? 7 (last pos))) (and (equal? 0 (car pos)) (equal? 0 (last pos)))) -1]
  [(or (equal? 7 (car pos)) (equal? 0 (last pos)))
```

```
    (cond

      [(equal? (get-pieza-contraria (get-elem (car pos) (last pos) othello)) (get-elem (- (car pos) 1)
(+ (last pos) 1) othello)) (list (pos-disponibles-diag-secun-abajo-izq othello pos))]))]

   [(or (equal? 0 (car pos)) (equal? 7 (last pos)))

    (cond

      [(equal? (get-pieza-contraria (get-elem (car pos) (last pos) othello)) (get-elem (+ (car pos)
1) (- (last pos) 1) othello)) (list (pos-disponibles-diag-secun-arriba-der othello pos))])]

   [else

    (cond

      [(and (equal? (get-pieza-contraria (get-elem (car pos) (last pos) othello)) (get-elem (- (car
pos) 1) (+ (last pos) 1) othello)) (equal? (get-pieza-contraria (get-elem (car pos) (last pos)
othello)) (get-elem (+ (car pos) 1) (- (last pos) 1) othello))) (list (pos-disponibles-diag-secun-
arriba-der othello pos) (pos-disponibles-diag-secun-abajo-izq othello pos))]

      [(equal? (get-pieza-contraria (get-elem (car pos) (last pos) othello)) (get-elem (- (car pos) 1)
(+ (last pos) 1) othello)) (list (pos-disponibles-diag-secun-abajo-izq othello pos))]

      [(equal? (get-pieza-contraria (get-elem (car pos) (last pos) othello)) (get-elem (+ (car pos)
1) (- (last pos) 1) othello)) (list (pos-disponibles-diag-secun-arriba-der othello pos))])]))
```