Memoria del proyecto

Reversi

Miguel Alexander Maldonado Lenis

Índice

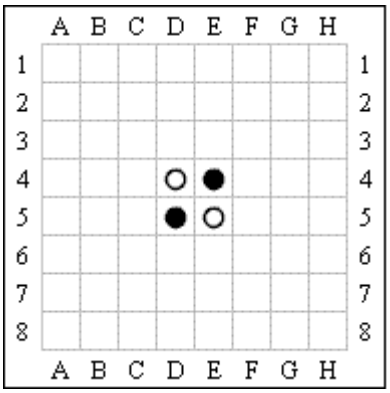
Resumen

En este proyecto se ha implementado el juego de mesa Reversi usando el lenguaje de programación Prolog con la ayuda del compilador SWI-Prolog, el cual también ofrece la posibilidad de ejecutar y debuguear los programas implementados en este lenguaje.

El objeto de este proyecto es el juego de mesa Reversi, en el cual dos jugadores comparten 64 fichas iguales de caras distintas que se van colocando por turnos en un tablero de 64 casillas. Cada jugador tiene asignado un color ganando quien tenga más fichas de su color al finalizar la partida. El fin de la partida es provocado al no quedar ninguna casilla libre, es decir, al no haber más movimientos

Reglas del juego

- El juego comienza situando cuatro fichas (dos de cada color) en las cuatro casillas centrales del tablero, de forma que cada pareja de fichas iguales forme una diagonal entre sí.

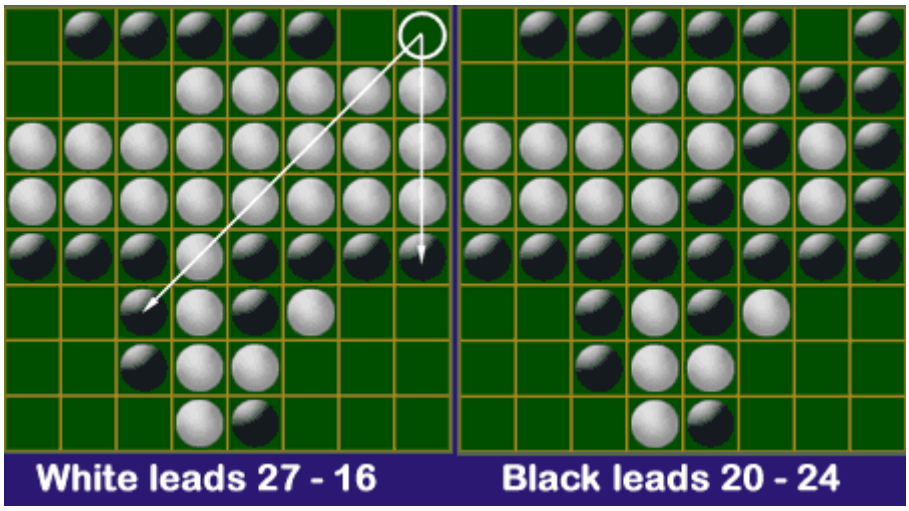


- Realiza el primer movimiento el jugador que juegue con negras.

- Los movimientos consisten en incorporar fichas al tablero a razón de una por turno, nunca en desplazar fichas de las que ya estuvieran sobre el tablero.

- Movimientos:
  - Sólo podrá incorporarse una ficha flanquendo a una o varias fichas contrarias.
  - Por flanquear se entiende el hecho de colocar la nueva ficha en un extremo de una hilera de fichas del color del contrario (una o más fichas) en cuyo extremo opuesto hay una ficha del color de la que se incorpora, sin que existan casillas libres entre ninguna de ellas. Esta hilera puede ser indistintamente vertical, horizontal o diagonal. De este modo, las fichas del contrincante quedan encerradas entre una que ya estaba en el tablero y la nueva ficha.
  - Cada vez que un jugador incorpora una ficha, y por lo tanto encierra a otras del contrario, debe dar la vuelta a las fichas encerradas convirtiéndolas así en propias.
  - Si en una sola incorporación se provocase esta situación de *flanqueo* en más de una línea, se voltearán todas las fichas contrarias que estuvieran implicadas en cada uno de los *flanqueos*.

- Si no fuera posible para un jugador encerrar a ninguna ficha, deberá pasar en su turno, volviendo el mismo a su oponente.



- El juego termina cuando no quedan casillas libres.

- Gana el jugador que tenga más fichas de su color en el tablero.

Desarrollo del proyecto

Estructuras

Las siguientes estructuras son asertadas en la base de conocimiento.

Tablero: Representado con 8 listas de longitud 8, en cada elemento de las listas puede haber uno de los siguientes valores:

- 0: Vacío
- 1: Jugador
- 2: Máquina

```
Tablero inicial
tablero([0,0,0,0,0,0,0,0]
        ,[0,0,0,0,0,0,0,0]
        ,[0,0,0,0,0,0,0,0]
        ,[0,0,0,1,2,0,0,0]
        ,[0,0,0,2,1,0,0,0]
        ,[0,0,0,0,0,0,0,0]
        ,[0,0,0,0,0,0,0,0]
        ,[0,0,0,0,0,0,0,0]).
```

Jugadores: Para representar a los jugadores se han empleado dos listas (**listaJug1** y **listaJug2**) cuyos elementos corresponden a las fichas que estos mismo han puesto en el tablero. La longitud máxima que pueden alcanzar las listas es de 64, ya que es el número de casillas del tablero.

Flujo

El juego se define mediante un bucle principal en el cual se controla tanto el movimiento del jugador como el de la máquina, el movimiento de la máquina siempre sigue al del jugador.

```
% Bucle principal del juego

bucleJuego(Dificultad):-
    fichas(NumFichas),
    movimientoJugador(NumFichas),
    dibujaTablero,
    fichas(NumFichas2),
    escribeFichasRestantes(NumFichas2),
    sleep(0.4),
    mueveMaquina(Dificultad),
    fichas(FichasRestantes),
    (FichasRestantes > 0 ->
        bucleJuego(Dificultad)
        ;
        haTerminado
    ).
```

Control del movimiento del jugador

¿Jugador bloqueado?

Antes de cada movimiento, primero se comprueba que el jugador no esté bloqueado, es decir, que existe al menos un movimiento posible para dicho jugador.

Este procedimiento se realiza para ambos jugadores (usuario y máquina), pero para el siguiente ejemplo vamos a suponer que el turno es del jugador.

Turno: Usuario

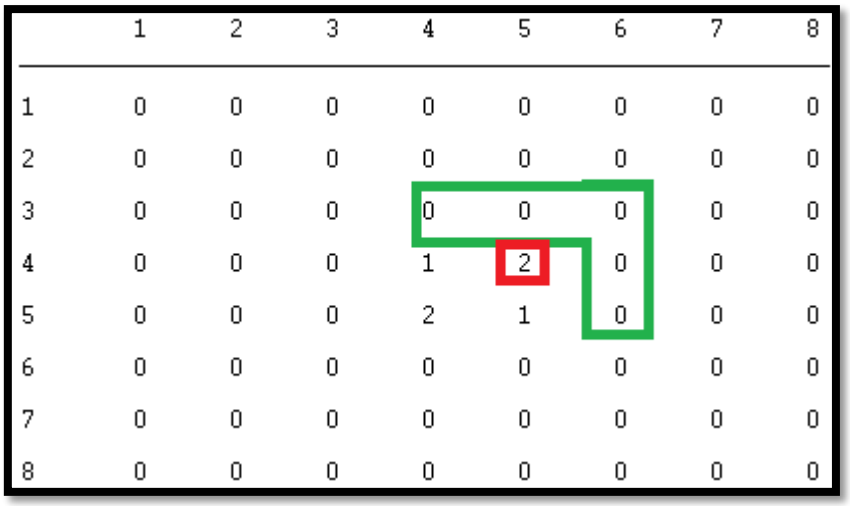
ListasJug1 = {(4,4), (5,5)}

ListaJug2 = {(4,5), (5,4)}

	1	2	3	4	5	6	7	8
1	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0
4	0	0	0	1	2	0	0	0
5	0	0	0	2	1	0	0	0
6	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0
8	0	0	0	0	0	0	0	0

i) Para cada una de las fichas del oponente (máquina) se cogen las posiciones adyacentes que están libres

ListaJug2 = {(4,5), (5,4)}
FichasAdyacentesLibres = {(3,4), (3,5), (3,6), (4,6), (5,6)}

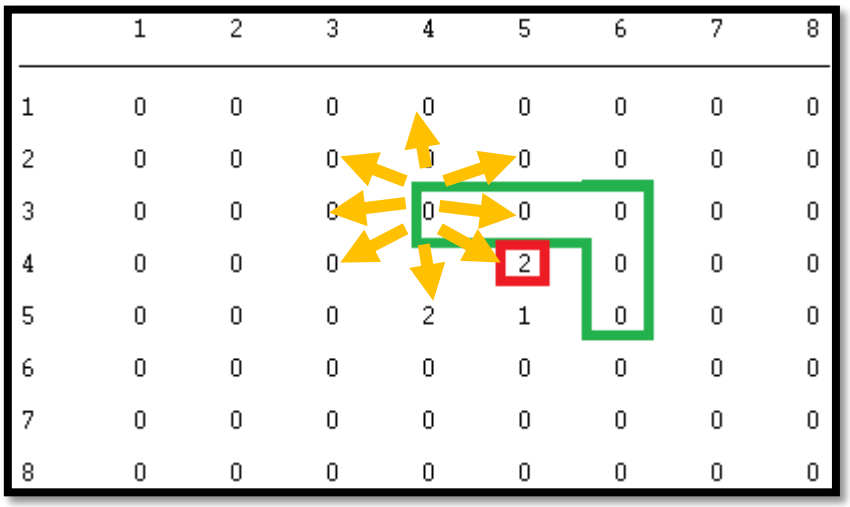


	1	2	3	4	5	6	7	8
1	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0
4	0	0	0	1	2	0	0	0
5	0	0	0	2	1	0	0	0
6	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0
8	0	0	0	0	0	0	0	0

ii) Para cada una de las posiciones adyacentes libres se verifica si existe un posible flanqueo. Este cálculo continúa hasta que no existen más fichas en la lista FichasAdyacentesLibres.

Para ello se deben encontrar las direcciones en las cuales validar dicho flanqueo.

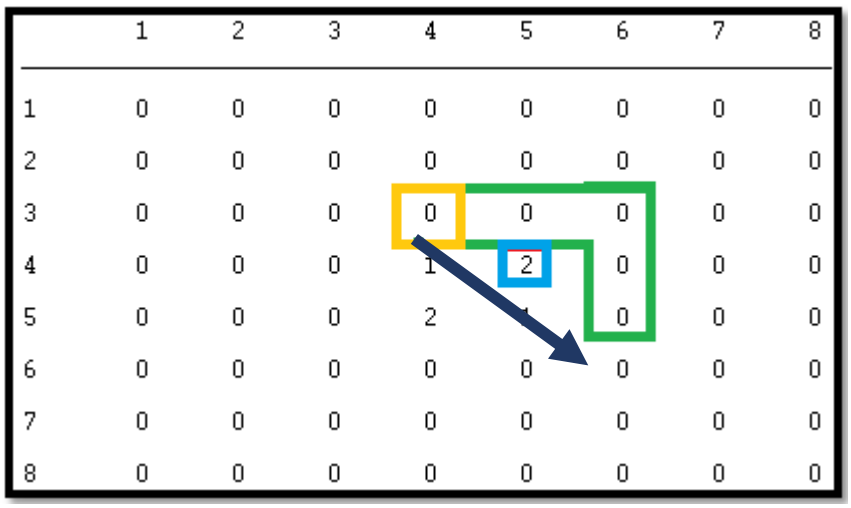
PosiciónLibreAEvaluar = (3,4)



	1	2	3	4	5	6	7	8
1	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0
4	0	0	0	1	2	0	0	0
5	0	0	0	2	1	0	0	0
6	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0
8	0	0	0	0	0	0	0	0

Esto se consigue comprobando las posiciones adyacentes del oponente (máquina) que existen para la posición libre (posible posición de usuario) que se esté evaluando en el momento.

PosiciónLibreAEvaluar = (3,4)
FichasAdyacentesJugador = {(4,5)}



	1	2	3	4	5	6	7	8
1	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0
4	0	0	0	1	2	0	0	0
5	0	0	0	2	1	0	0	0
6	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0
8	0	0	0	0	0	0	0	0

Se comprueba que existe el *flanqueo*. Nos movemos en la dirección encontrada mientras haya fichas del oponente parando al encontrar la ficha del usuario que cierra el *flanqueo* o fallando si encontramos vacío.

El cálculo continúa hasta que no existen más fichas en la lista FichasAdyacentesJugador.

iii) Si se encuentra un *flanqueo* se assertan las fichas libres que corresponderían a la posición en la cual el usuario debería poner su ficha para realizar un *flanqueo*.

¿La posición es válida?

Esta comprobación sólo se realiza para el jugador ya que es el único que puede introducir a su elección la posición. La máquina dispone para cada turno de todos los posibles movimientos que puede realizar con el procedimiento arriba explicado.

Una vez el usuario ha introducido la posición en la cual quiere poner su ficha, se realizan las siguientes validaciones:

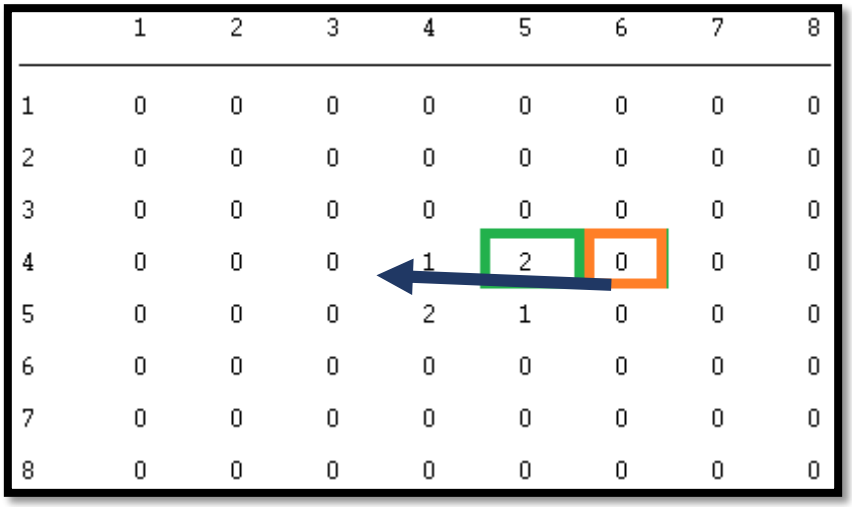
i) Se comprueba que la posición está libre.

ii) Después, se comprueba que existen fichas del oponente adyacentes.

PosiciónJugador = (4,6)
FichaOponenteAdyacente = (4,5)

	1	2	3	4	5	6	7	8
1	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0
4	0	0	0	1	2	0	0	0
5	0	0	0	2	1	0	0	0
6	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0
8	0	0	0	0	0	0	0	0

iii) Por último, se comprueba que existe al menos un *flanqueo* y si existe se efectúa el cambio de propietario de las fichas *flanqueadas* y se pone la nueva ficha.



	1	2	3	4	5	6	7	8
1	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0
4	0	0	0	1	1	1	0	0
5	0	0	0	2	1	0	0	0
6	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0
8	0	0	0	0	0	0	0	0

Lógica

La vista lógica del proyecto es la siguiente:

- **algoritmo.pl** – Definición del algoritmo minimax con poda alfa-beta
- **heurística.pl** – Definición de la heurística empleada en el algoritmo (se explica más abajo)
- **reversi.pl** – Definición del juego

La "inteligencia" de la máquina se ha realizado adaptando el algoritmo minimax con poda alfa-beta (algorithms.pl) implementado en el proyecto tictactoe-prolog de Guilherme Balena Versiani.

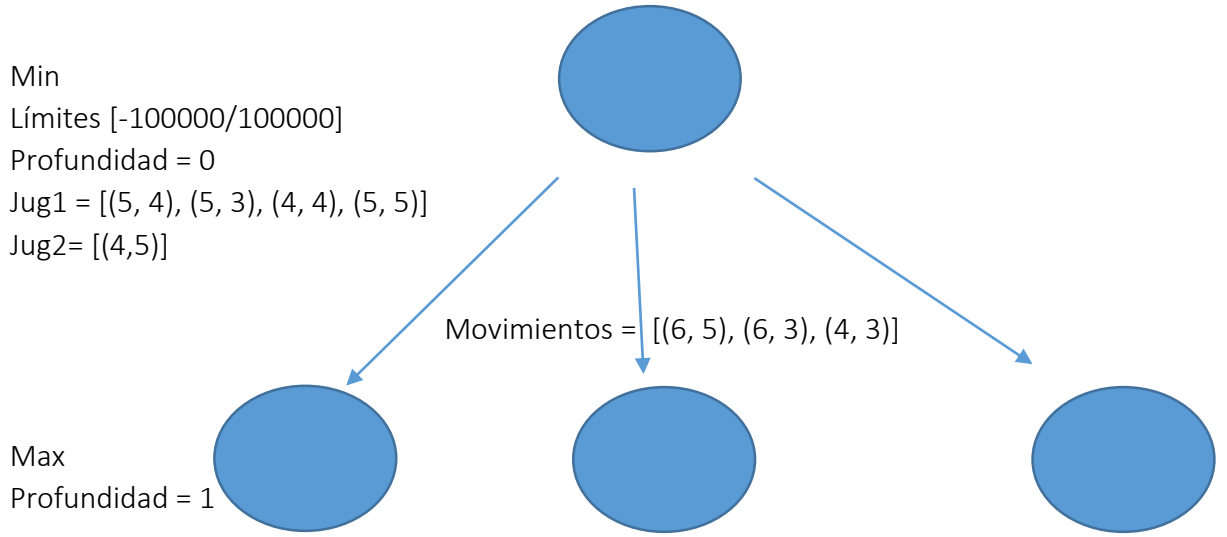
En concreto se han modificado los predicados ***value***, ***moves*** y ***put***, además de añadir una cláusula más al predicado ***goodenough*** para evitar el fallo en ejecución del programa debido al no tener definida una condición de parada en el supuesto de no existir más movimientos posibles para evaluar, caso representado con una lista vacía

```
goodenough(_, [], _, Move, Val, GoodMove, Val, _, _, _, _) :- !.
```

A continuación se presenta un diagrama del funcionamiento del predicado minimax (algoritmo.pl) con un ejemplo concreto:

Estado del tablero

	1	2	3	4	5	6	7	8
1	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0
4	0	0	0	1	2	0	0	0
5	0	0	1	1	1	0	0	0
6	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0
8	0	0	0	0	0	0	0	0



Movimieto = (6,5)
Límites [-100000/0]
Jug1= [(5,4), (4,3), (4,4)]
Jug2= [(4,5), (6,5), (5,5)]
ValorHeurística = 0
Tablero:

```
[[0,0,0,0,0,0,0,0],
 [0,0,0,0,0,0,0,0],
 [0,0,0,0,0,0,0,0],
 [0,0,0,1,2,0,0,0],
 [0,0,1,1,2,0,0,0],
 [0,0,0,0,2,0,0,0],
 [0,0,0,0,0,0,0,0],
[0,0,0,0,0,0,0,0]]
```

Movimieto = (6,3)
Límites [-100000/0]
Jug1= [(5,5), (5,3), (4,4)]
Jug2= [(4,5), (6,3), (5,4)]
ValorHeurística = 0
Tablero:

```
[[0,0,0,0,0,0,0,0],
 [0,0,0,0,0,0,0,0],
 [0,0,0,0,0,0,0,0],
 [0,0,0,1,2,0,0,0],
 [0,0,1,2,1,0,0,0],
 [0,0,2,0,0,0,0,0],
 [0,0,0,0,0,0,0,0],
 [0,0,0,0,0,0,0,0]]
```

Movimieto = (4,3)
Límites [-100000/0]
Jug1= [(5,5), (5,3), (5,4)]
Jug2= [(4,5), (4,3), (4,4)]
ValorHeurística = 0
Tablero:

```
[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],
 [0,0,0,0,0,0,0,0],
 [0,0,0,0,0,0,0,0],
 [0,0,0,0,0,0,0,0]]
```

Puesto que el valor de la heurística es igual en los tres casos, se queda con el primer movimiento.

La heurística del juego (heurística.pl) se basa en las siguientes estrategias explicadas en el documento Juego de Inteligencia Artificial: Othello, las cuales son:

3. Esquinas: Con esta estrategia se comprueba el número de fichas colocadas en las esquinas del tablero.

4. Laterales: Se ha adaptado para comprobar el número de fichas agrupadas en las esquinas del tablero.

5. Casillas X: Son casillas contiguas a una esquina en diagonal. Se ha adaptado para comprobar si existen fichas en esta disposición.

6. Fronteras: Se ha adaptado para determinar el número de casillas libres adyacentes a las fichas del jugador.

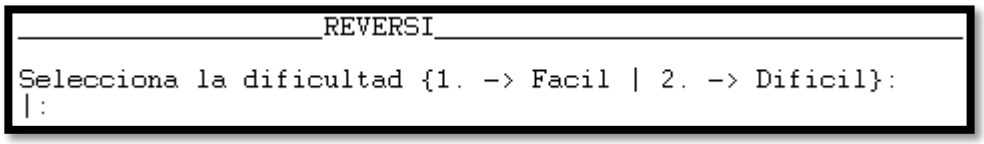
Por otro lado se han implementado dos estrategias nuevas la cuales son las siguientes:

1. Win: Este predicado devuelve un valor que aumenta cuanto mayor es la diferencia entre el número de fichas del jugador del turno actual y su oponente sólo si el jugador del turno actual tiene más fichas que su oponente.

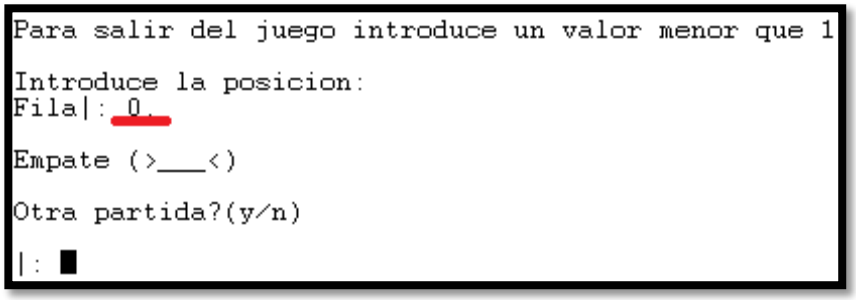
2. Fichas agrupadas: Se comprueba tanto en vertical como en horizontal (filas y columnas), el número de fichas agrupadas partiendo la primera de ellas desde un límite del tablero. Cuanto mayor sea el número, mayor es la posibilidad de ganar puesto que si un grupo de fichas parten desde uno de los extremos del tablero, por su disposición, es menos probable que puedan ser *flanqueadas.*

Instrucciones

El usuario puede elegir dos modalidades de dificultad, una vez elegida se enfrentará a la máquina cuya lógica está definida para ganar al usuario.



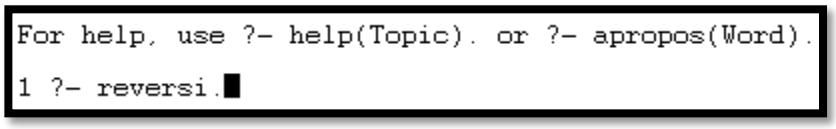
En cualquier momento de una partida en curso el usuario podrá terminar la misma introduciendo un valor menor que 1. Tanto al finalizar una partida como al terminar una en curso, se le preguntará al usuario si desea seguir jugando o salir definitivamente de la aplicación.



La aplicación se puede ejecutar con el entorno SWI Prolog el cual se puede obtener en la siguiente dirección:

http://www.swi-prolog.org/Download.html

Para iniciar el juego se debe introducir el predicado **reversi.**



Conclusiones

Durante el desarrollo del proyecto la mayor dificultad encontrada fue la de adaptar un algoritmo de poda alfa-beta de un código de otra aplicación puesto que debía comprenderse de forma muy clara el funcionamiento del mismo para poder unirlo, aun así se presentaron problemas, como la existencia de bucles en la exploración del árbol debido a la necesidad de la correcta disposición de ciertos cortes en el procedimiento, excesivo número de aserciones en la base de conocimiento debido a la dificultad de la recursividad, etc, que retrasaron el desarrollo.

Prolog permite que el desarrollo de este tipo de juegos se pueda realizar de una forma más fluida puesto que conlleva una gran ventaja que el propio lenguaje incluya la recursión necesaria para la implementación de la lógica de un contrincante virtual.

Bibliografía

Código algoritmo minimax con poda alfa-beta:
https://code.google.com/p/tictactoe-prolog/

SWI-Prolog:
http://www.swi-prolog.org/

Reversi:
https://es.wikipedia.org/wiki/Reversi

Heurística:
http://www.it.uc3m.es/jvillena/irc/practicas/07-08/Othello.pdf

Estructura basada en el juego del molinero:
http://wikis.fdi.ucm.es/PDA/Archivo:Molinero.zip