Universidad de Cantabria

Facultad
de
Ciencias

Jugando al Othello con Inteligencia Artificial
(Playing Othello with Artificial Intelligence)

Trabajo de Fin de Grado
para acceder al

GRADO EN INGENIERÍA INFORMÁTICA

Autor: Sandra Alegría Ocampo

Director: Inés González Rodríguez

Julio - 2024

Resumen

En este trabajo se desarrollará una aplicación que permitirá jugar al juego de mesa conocido como Othello o Reversi. La aplicación permite jugar a dos jugadores humanos pero, también, que un jugador humano juegue contra un jugador artificial. Dicho jugador toma decisiones utilizando técnicas de búsqueda con adversarios, propias del ámbito de la Inteligencia Artificial. En concreto, se utiliza el algoritmo minimax con poda alfa-beta y profundidad acotada. Para ello, se analizan experimentalmente distintas funciones heurísticas de evaluación no terminales para estados del tablero. Además se emplean tablas de transposición para mejorar la eficiencia del proceso de búsqueda para la toma de decisiones.

Palabras clave: Othello, Inteligencia Artificial, algoritmo minimax, poda alfa-beta, tabla de transposición, función heurística.

Abstract

In this work, an application will be developed that allows playing the board game known as Othello or Reversi. The application allows two human players to play, but also enables a human player to play against an artificial player. This artificial player makes decisions using adversarial search techniques, typical of the field of Artificial Intelligence. Specifically, the minimax algorithm with alpha-beta pruning and bounded depth is used. For this purpose, various non-terminal evaluation heuristic functions for board states are experimentally analyzed. Additionally, transposition tables are employed to improve the efficiency of the search process for decision-making.

Key words: Othello, Artificial Intelligence, minimax algorithm, alpha-beta pruning, transposition table, heuristic function.

Índice general

Índice de figuras

Índice de Algoritmos

Índice de tablas

Capítulo 1

Introducción

1.1. Inteligencia artificial en juegos de mesa

La historia de la inteligencia artificial y los juegos de mesa se remonta a los inicios de la informática, ya que en aquellos tiempos se desarrollaban programas para jugar a algunos juegos como el ajedrez o las damas, de esta manera se probaba si las máquinas eran capaces de resolver desafíos que requerían una cierta inteligencia. Se escogían los juegos de mesa para estas funciones ya que tienen reglas claras, finitas y bien definidas, que pueden ser fácilmente implementables en una inteligencia artificial, ya que el entorno en el que el agente inteligente toma las decisiones es simple. Además son juegos con muchas posibilidades que requiere de gran estrategia, donde se puede probar la toma de decisiones de la inteligencia artificial.

Dando un vistazo a la historia en torno a esto, en los inicios Alan Turing, el reconocido como creador de la informática, reinventó el algoritmo Minimax y lo empleó para jugar al ajedrez[3]. Más adelante en 1952 A. S. Douglas, como parte de su tesis doctoral, logró desarrollar el primer software que dominó un juego, concretamente el *Tic-Tac-Toe*[3] (Tres en línea). Pasado un tiempo se empezó a desarrollar programas que aprenden a medida que juegan, Arthur Samuel fue el primero en inventar el aprendizaje por refuerzo, logrando un programa que aprendió a jugar a las damas jugando contra sí mismo[3].

Sin embargo, estos programas seguían sin ser lo suficientemente buenos como para derrotar a los maestros de aquellos juegos de mesa. Hasta que tras tres décadas de investigación, en 1994 el programa de ordenador Chinook le arrebató el titulo al entonces campeón del mundo en las damas, Marion Tinsley[3], ya que este tuvo que retirarse de la competición después de seis empates por problemas médicos. Años más tarde ocurrió un acontecimiento altamente sonado en aquella época. En 1997 un programa de ordenador,

Deep Blue, desarrollado por IBM venció al gran maestro Garry Kasparov. Este acontecimiento dejó ver a las personas los grandes avances informáticos, sobre todo los relativos a la inteligencia artificial.

Actualmente se sigue investigando acerca de estos campos, además ahora se cuenta con hardware más potente que permite realizar pruebas más rápidas con programas que antes eran impensables.

1.2. Othello

El Othello es un juego de mesa clásico para dos jugadores, también conocido popularmente por el nombre Reversi. Su origen está un poco diluido en el tiempo ya que ha sufrido varias transformaciones a lo largo de los años hasta llegar a lo que es en la actualidad. Su historia comenzó en el año 1880, en el cual Lewis Waterman y John W. Mollett comercializaron dos juegos con reglas muy similares. Casi un siglo más tarde Goro Hasegawa cambió un par de reglas de aquellos juegos y en el año 1971 registró lo que desde entonces conocemos como Othello. Parte del atractivo del Othello reside en que sus reglas son sencillas, con lo que incluso un jugador principiante puede disfrutar de una partida, pero para jugar bien hace falta tener en cuenta muchos factores [4].

Conceptualmente, el Othello pertenece a la misma familia de juegos de mesa que el Go, donde el objetivo es capturar territorio rodeando las fichas del oponente. Para jugar se necesita un tablero de 8x8 casillas y 64 fichas con una cara negra y una cara blanca. Además, debe contar con la dispoisición inicial que se muestra en la Figura 1.1, con cuatro fichas (dos por jugador) ocupando las casillas centrales, con las blancas en las casillas de la diagonal principal (D4 y E5) y las negras en las correspondientes a la diagonal secundaria (D5 y E4).

El juego se desarrolla por turnos comenzando por el jugador que utiliza las fichas negras. En su turno el jugador debe realizar un movimiento que consiste en colocar una pieza con su color hacia arriba en una posicion legal, en caso de que esto sea posible, en caso contrario se pasaría el turno al otro jugador. Para que una posición sea legal ha de estar vacía y ser adyacente (en horizontal, vertical o diagonal) a una casilla ocupada por una ficha del oponente. Además, debe cumplirse que, de colocarse una ficha del jugador en dicha posición, esta ficha, junto a otra del mismo jugador, encerrarían (en línea recta) al menos una ficha del contrincante. Esto es esencial para entender el funcionamiento del Othello, por ello en la Figura 1.2 podemos observar un ejemplo con las colocaciones legales en un turno para el jugador que controla las fichas negras indicadas con fichas semitransparentes.

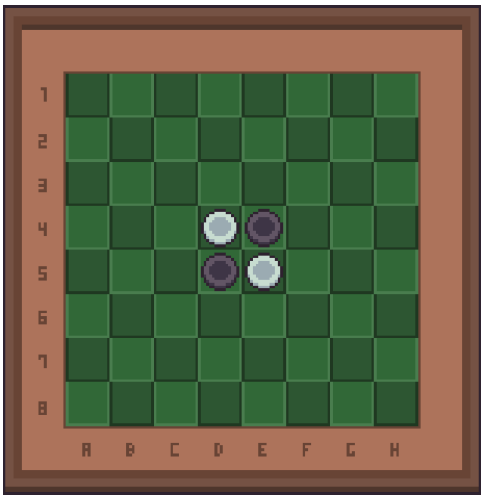

Figura 1.1: Disposición inicial en tablero del juego Othello

Una vez realizada una colocación legal, se voltean todas las fichas del contrincante que se han encerrado, de forma que pasan a tomar el color del jugador que ha realizado el movimiento. La partida finaliza en el momento en el que se llenan todas las casillas del tablero o en el momento en que ninguno de los dos jugadores pueda realizar al menos una colocación legal.

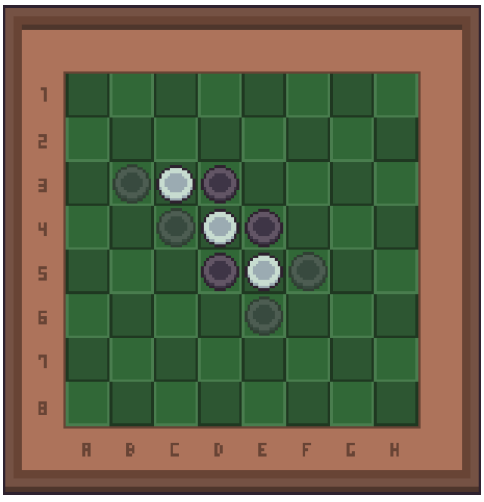

Figura 1.2: Colocaciones legales en el turno para el jugador que controla las fichas negras: B3, C4, E6 y F5.

El juego de mesa Othello es interesante desde el punto de vista de la inteligencia artificial ya que tiene un espacio de búsqueda más pequeño que el ajedrez, pero definir una función de evaluación para este es difícil, ya que

la ventaja material (es decir, la diferencia de fichas en el tablero) no es tan importante como lo es la movilidad (cantidad de movimientos posibles en el siguiente turno) [1]. Además, el Othello, tiene un conjunto de reglas simples en comparación con otros juegos de mesa y a su vez es lo suficientemente complejo para permitir realizar una exploración significativa mediante inteligencia artificial. El factor de ramificación que se obtiene al explorar el Othello es mayor que es de las damas pero menor que el del ajedrez. De esta forma se consigue realizar búsquedas importantes sin necesidad de grandes recursos para llevar a cabo la computación de las mismas.

La investigación en inteligencia sobre Othello se remonta al menos a la década de 1970, en la cual diferentes investigadores desarrollaban diversos algoritmos de inteligencia artificial para el juego. En la investigación actual de inteligencia artificial, Othello se utiliza, entre otros, en el desarrollo de jugadores generados automáticamente mediante técnicas de computación evolutiva, para evaluar técnicas como el aprendizaje por refuerzo o como banco de pruebas para nuevos enfoques. En el contexto de la educación en informática, el juego Othello se ha utilizado como herramienta de enseñanza en varias áreas, incluidas, pero no limitadas a la inteligencia artificial [5].

1.3. Objetivo

El objetivo de este trabajo es programar un jugador basado en Inteligencia Artificial (más concretamente, técnicas de búsqueda con adversarios) para el juego de mesa conocido como Othello. Este objetivo puede desglosarse en los siguientes subobjetivos:

- Partiendo de las técnicas de búsqueda con adversarios estudiadas en la asignatura de Introducción a los Sistemas Inteligentes, documentarse sobre maneras de mejorar la eficiencia de las mismas, como puede ser el uso de tablas de transposición, así como de los heurísticos específicos del Othello para la evaluación de estados no terminales.
- Diseñar e implementar un entorno de juego de Othello.
- Diseñar, implementar y evaluar un jugador inteligente para el Othello.

1.4. Organización de la memoria

El resto de la memoria está organizada como sigue. En el capítulo 2 modelamos el juego Othello como un problema de búsqueda con adversarios,

dentro del contexto de la Inteligencia Artificial, y revisamos las técnicas y algoritmos de la misma que pueden aplicarse. Posteriormente, los capítulos 3 y 4 se dedican a describir los aspectos relativos al diseño y desarrollo del software. Seguidamente, en el capítulo 5 presentamos las pruebas realizadas, principalmente para evaluar el jugador no humano desarrollado. Finalmente, en el capítulo 6 terminamos con unas conclusiones.

Capítulo 2

Inteligencia Artificial para Othello

Othello es un juego para dos jugadores que se desarrolla por turnos. Además es lo que en teoría de juegos se denomina juego determinista, de información perfecta y suma cero. Es determinista porque no hay incertidumbre respecto al resultado de un movimiento de un jugador. Es de información perfecta, ya que todo lo referente al desarrollo de la partida es totalmente visible para ambos jugadores en todo momento incluidos los posibles movimientos del contrincante. Por último, es un juego de suma cero, por que la ganancia de un jugador es exactamente la perdida del otro. Por lo tanto, o bien hay un ganador y un perdedor o bien se produce un empate.

Modelar el Othello como un juego de suma cero permite adoptar decisiones sobre qué movimiento debe realizar un jugador en un momento dado utilizando técnicas de Inteligencia Artificial, en concreto, de búsqueda en espacios de estados con adversarios. En la Sección 2.2 introduciremos el algoritmo minimax, que recibe su nombre porque etiqueta a ambos jugadores como jugador MAX y jugador MIN y que, en condiciones ideales, encuentra la decisión óptima para MAX, es decir, aquella que le permite obtener la máxima ganancia posible, suponiendo que su contrincante MIN también juega de manera óptima. Posteriormente, describiré técnicas que permiten mejorar la eficiencia media del algoritmo, así como heurísticos propios del Othello que permiten tomar decisiones subóptimas cuando hay restricciones de tiempo de cómputo o de uso de memoria.

2.1. Juegos de suma cero

Un juego de suma cero con información perfecta puede definirse con los siguientes componentes [1]:

- s_0: Corresponde al estado inicial, en este caso, como se ha explicado en la sección 1.2, un tablero de 8x8 casillas vacío, salvo las cuatro casillas centrales, que contienen 2 fichas blancas en la diagonal principal y 2 fichas negras en la diagonal secundaria.
- MUEVE(s): El jugador al que le toca colocar ficha en el estado s. Para el estado inicial (s_0), MUEVE retorna el jugador que juega con negras.
- ACCIONES(s): El conjunto de acciones aplicables en el estado s, en nuestro caso colocar una ficha del color del jugador que mueve en alguna de las casillas legales.
- RESULTADO(s,a): El modelo de transición que define el estado resultante de tomar la acción a en el estado s. En nuestro caso, será el tablero resultante de colocar la ficha del jugador que mueve en s en la casilla indicada por a, realizar los volteos de las fichas del contrincante que han quedado encerradas y pasar el turno a dicho contrincante.
- ES-TERMINAL(s): Una prueba terminal que es verdadera cuando se cumplen las condiciones de finalización de la partida y falsa en caso contrario.
- UTILIDAD(s,p): Una función de utilidad que define el valor numérico final para el jugador p en el estado terminal s. Si p gana el valor será 400, si pierde -400 y si empata 0.

2.1.1. Árbol de juego

El estado inicial, junto con las funciones ACCIONES y RESULTADO definen el espacio de estados, un grafo en el que los vértices corresponden a estados del juego, los arcos corresponden a los movimientos o acciones aplicables en el estado en el origen y cuyo resultado es el estado en el destino. En dicho grafo, un mismo estado puede alcanzarse por más de un camino (es decir, aplicando distintas secuencias de movimientos). Al recorrer parte o la totalidad del grafo de estados para determinar qué movimiento debe realizarse en el estado del que partimos se genera un árbol de búsqueda.

Denominamos árbol de juego al árbol de búsqueda que se genera recorriendo todos los posibles caminos desde el estado inicial a un estado terminal cualquiera.

Nótese que, en el árbol de búsqueda, cada vez que se cambia de nivel (se realiza una acción) cambia el turno del jugador al que le toca mover. En concreto, si desarrollamos el árbol de juego del Othello a partir del estado inicial, entonces el nivel 1 (correspondiente a la raíz) y todos los niveles impares corresponderían a los turnos del jugador que juega con negras, mientras que los niveles pares corresponderían al jugador con blancas. Además, en el árbol de juego, los nodos hoja corresponden a estados terminales, es decir, aquellos estados en los que no hay ninguna acción aplicable (es decir, el tablero está lleno o no hay ninguna posición legal en la que colocar una ficha del jugador en posesión del turno).

Si definimos la complejidad del espacio de estados como el número de estados del mismo (es decir, el número de estados o disposiciones del tablero alcanzables desde el estado inicial mediante una secuencia de movimientos legales), entonces la complejidad del espacio de estados del Othello es 10^{28}. Por otro lado, si definimos la complejidad del árbol de juego como el número de hojas (correspondientes a estados terminales) en el árbol de juego, en el Othello es 10^{58} [6].

2.2. Algoritmo minimax [1]

Para un estado del juego (puede ser el estado inicial o un estado posterior en la partida), denominamos MAX al jugador al que le toca mover en un estado y MIN a su contrincante y consideramos el árbol de juego que se forma a partir de ese estado. MAX debe elegir qué movimiento realizar y determinar una estrategia ganadora teniendo en cuenta que no controla los movimientos del jugador MIN (y que los intereses de MIN son contrarios a los suyos).

MAX puede encontrar una estrategia óptima calculando el valor minimax de cada estado en el árbol de juego. Dicho valor corresponde a la utilidad o ganancia para MAX de llegar a dicho estado, suponiendo que ambos jugadores juegan de manera óptima desde ese momento hasta el final de la partida. Si el estado es terminal, claramente, el valor vendrá dado por su utilidad para MAX (según la función UTILIDAD que forma parte de la definición del juego). En un estado no terminal, MAX debería elegir mover al estado que le proporcione mayor ganancia, mientras que si el turno es de MIN, éste debería elegir mover al estado que proporcione la menor ganancia para MAX.

Por lo tanto, para un estado cualquiera s, su valor minimax viene dado

por:

$$\text{MINIMAX(s)} = \begin{cases} \text{UTILIDAD}(s, \text{MAX}) & \text{si ES-TERMINAL}(s) \\ \text{máx}\{\text{MINIMAX}(\text{RESULTADO}(s,a)) : a \in \text{ACCIONES}(s)\} & \text{si MUEVE}(s) = \text{MAX} \\ \text{mín}\{\text{MINIMAX}(\text{RESULTADO}(s,a)) : a \in \text{ACCIONES}(s)\} & \text{si MUEVE}(s) = \text{MIN} \end{cases}$$

La decisión minimax que MAX debería adoptar en la raíz del árbol de juego es realizar la acción que determina su valor minimax, es decir, la que le lleva a un estado con máximo valor.

El algoritmo minimax consiste en hacer un recorrido primero en profundidad con *backtracking* del árbol de juego para calcular el valor minimax de cada nodo del árbol.

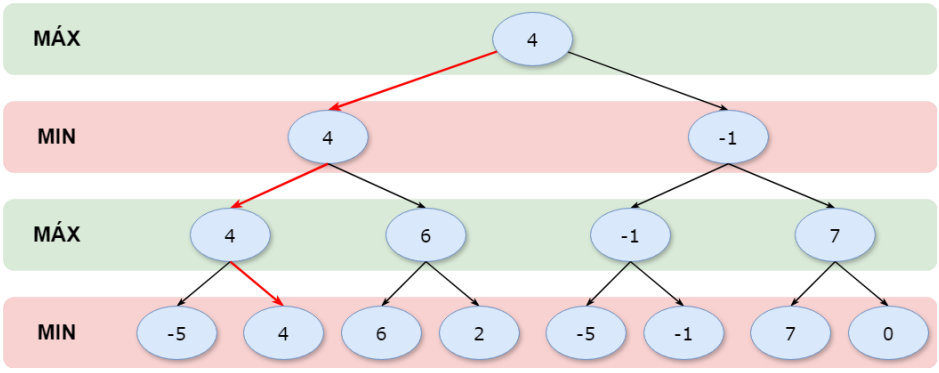

Figura 2.1: Árbol de juego con los nodos etiquetados como resultado del algoritmo minimax.

En la Figura 2.1 se muestra un árbol de juego en el que cada nodo está etiquetado con su valor minimax (valor que se calcula con el algoritmo minimax) y se destaca en rojo la rama que corresponde a los movimientos que realizarían ambos jugadores si jugasen de manera óptima (por tanto, el arco en rojo que sale de la raíz sería el correspondiente a la acción seleccionada por el algoritmo).

Dado el tamaño del árbol de juego del Othello, es imposible en la práctica recorrerlo en su totalidad con el algoritmo minimax (recursivo o iterativo) para etiquetar todos los nodos. En consecuencia, para juegos complejos como el Othello se opta por aplicar minimax con profundidad acotada. En su versión con profundidad acotada, cuando se llega a la profundidad máxima, si el estado alcanzado no es terminal (lo más probable), se evaluará dicho estado con una función de evaluación heurística. Esto hace que la decisión adoptada ya no tenga por qué ser óptima (lo que sí ocurre con minimax). Podemos observar un pseudocódigo de dicha implementación en el Algoritmo 1.

El peso total de la decisión en el algoritmo recae en la función evaluadora. Por tanto, cuanto mejor sea la función que evalúa cada nodo mejores decisiones tomará el algoritmo.

2.3. Poda Alfa-Beta [1] [2]

En muchas ocasiones, es posible calcular la decisión minimax sin necesidad de visitar todos los nodos del árbol de juego, es decir, se podría realizar podas en el recorrido minimax sin afectar a la decisión de MAX. Una de estas técnicas de poda es la llamada poda alfa-beta.

Se incorporan a la búsqueda dos variables alfa y beta que permitirán gestionar el recorrido del árbol de juego, ignorando los subárboles de este que probablemente no contengan el mejor movimiento . Podemos ver dicha modificación en el Algoritmo 2. En la figura 2.2 se muestra el ejemplo de la Figura 2.1 con la mejora aplicada al algoritmo, la poda alfa-beta. Con esta modificación el algoritmo se ahorra la exploración de tres nodos.

En la práctica, de media, se producen podas y ahorra tiempo. Sin embargo, en el peor de los casos no se produce poda alguna y la complejidad temporal es la misma que en minimax.

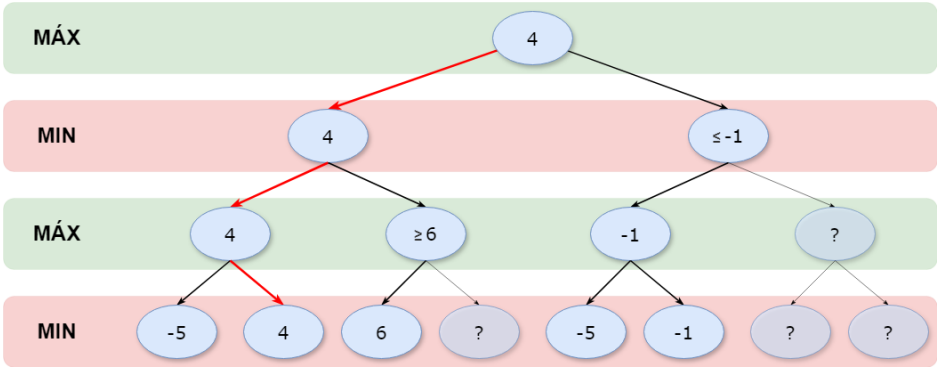

Figura 2.2: Árbol de juego de la Figura 2.1 etiquetado después de aplicar el algoritmo mínimax con poda alfa-beta; en gris etiquetados con un interrogante, los nodos no visitados.

Algoritmo 1 Algoritmo minimax con profundidad acotada

Entrada: juego, max, profundidad
Salida: mejorPuntuacion, mejorMovimiento

```
# Comprobar fin de la recursión
si juego.finJuego() o profundidad ==MAXIMA_PROFUNDIDAD entonces devolver juego.evalua(max), null
fin si
# Inicializar valores
si juego.jugadorActivo() == max entonces
    mejorPuntuacion ← −∞
si no
    mejorPuntuacion ← ∞
fin si
mejorMovimiento ← null
# Recorrer colocaciones posibles
para movimiento en juego.obtenerMovimientos() hacer
    # Realizar colocacion
    nuevoJuego ← juego.colocaFicha(movimiento)
    # Aplicar recursión
    puntuacion, _ ← minimax(nuevoJuego, max, profundidad − 1)
    # Actualizar valores
    si juego.jugadorActivo() == max entonces
        si puntuacion > mejorPuntuacion entonces
            mejorPuntuacion ← puntuacion
            mejorMovimiento ← movimiento
        fin si
    si no
        si puntuacion < mejorPuntuacion entonces
            mejorPuntuacion ← puntuacion
            mejorMovimiento ← movimiento
        fin si
    fin si
fin para
    devolver mejorPuntuacion, mejorMovimiento
```

Algoritmo 2 Algoritmo minimax con profundidad acotada y poda alfa-beta

Entrada: juego, max, profundidad, alfa, beta
Salida: mejorPuntuacion, mejorMovimiento

```
# Comprobar fin de la recursión
si juego.finJuego() o profundidad == MAX_PROFUNDIDAD entonces
    devolver juego.evalua(max), null
fin si
si juego.jugadorActivo() == max entonces
    mejorPuntuacion ← −∞
si no
    mejorPuntuacion ← ∞
fin si
mejorMovimiento ← null
para movimiento en juego.obtenerMovimientos() hacer
    nuevoJuego ← juego.colocaFicha(movimiento)
    puntuacion, _ ← minimax(nuevoJuego, max, profundidad − 1, alfa, beta)
    si juego.jugadorActivo() == max entonces
        si puntuacion > mejorPuntuacion entonces
            mejorPuntuacion ← puntuacion
            mejorMovimiento ← movimiento
            beta ← min(beta, puntuacion)
            si beta <= alfa entonces
                break
            fin si
        fin si
    si no
        si puntuacion < mejorPuntuacion entonces
            mejorPuntuacion ← puntuacion
            mejorMovimiento ← movimiento
            alfa ← max(alfa, puntuacion)
            si alfa >= beta entonces
                break
            fin si
        fin si
    fin si
fin para
    devolver mejorPuntuacion, mejorMovimiento
```

2.4. Función evaluadora

En la versión del algoritmo con profundidad acotada, la función evaluadora es la responsable de asignar un valor a cada situación del tablero (es decir, cada nodo del árbol de juego), aunque no corresponda a un estado terminal.

El acierto de la decisión imperfecta que tome MAX usando profundidad acotada depende fuertemente de la calidad de la función evaluadora, puesto esta es la que guía la búsqueda minimax. En ese sentido, en la literatura se han estudiado diversas funciones heurísticas que normalmente tienen en cuenta los siguientes aspectos o características del estado a evaluar[7]:

- **Fichas:** El algoritmo 3 evalúa la diferencia de fichas que hay entre los jugadores. Viene motivado porque en el Othello, cuando finaliza la partida, gana el jugador con más fichas sobre el tablero, por tanto cuentas más fichas tenga un jugador, mejor.

- **Movilidad:** El algoritmo 4 evalúa la diferencia de movimientos posibles entre los jugadores. La idea es que, cuantos más posibles movimientos tenga un jugador en el siguiente turno, más posibilidades tendrá de tomar posiciones favorables que le ayuden a inclinar la partida a su favor.

- **Esquinas:** El algoritmo 5 evalúa la diferencia de fichas colocadas en las esquinas entre los jugadores. Las esquinas en el Othello son posiciones esenciales ya que las fichas colocadas en dichos espacios no pueden ser encerradas de ninguna manera, siendo imposibles de voltear, pero sí pueden encerrar fichas rivales, convirtiéndose así en un activo seguro para el jugador en cuestión.

- **Adyacencia a las esquinas:** El algoritmo 6 evalúa la diferencia de fichas colocadas en casillas adyacentes a esquinas vacías del tablero entre los jugadores. Sigue la lógica inversa que la evaluación de las esquinas: si un jugador coloca sus fichas en las casillas adyacentes a esquinas vacías, está generando una situación desfavorable para sí mismo. Esto se debe a que está propiciando la posibilidad de que el jugador contrario pueda emplear dicha esquina en su siguiente colocación.

La evaluación final de un estado no terminal combina estos cuatro aspectos para así dar un valor global que tenga en cuenta todo este conocimiento heurístico, tal y como se muestra en el Algoritmo 7.

Algoritmo 3 Evaluador de la diferencia de fichas entre los jugadores en el tablero

Entrada: $juego, jugador$
Salida: $puntuacion$
$jugadorContrario \leftarrow juego.obtenerJugadorContrario(jugador)$
$fichasJugador \leftarrow juego.obtenerFichasJugador(jugador)$
$fichasContrario \leftarrow juego.obtenerFichasJugador(jugadorContrario)$
$totalFichas \leftarrow fichasJugador + fichasContrario$
si $fichasJugador > fichasContrario$ **entonces**
  $puntuacion \leftarrow 100 * (fichasJugador/totalFichas)$
si no si $fichasJugador < fichasContrario$ **entonces**
  $puntuacion \leftarrow -100 * (fichasContrario/totalFichas)$
si no
  $puntuacion \leftarrow 0$
fin si
  devolver $puntuacion$

Algoritmo 4 Evaluador de la diferencia de movilidad entre los jugadores en el tablero

Entrada: $juego, jugador$
Salida: $puntuacion$
$jugadorContrario \leftarrow juego.obtenerJugadorContrario(jugador)$
$movimientosJugador \leftarrow len(juego.posiblesMovimientosJugador(jugador))$
$movimientosContrario \leftarrow len(juego.posiblesMovimientosJugador(jugadorContrario))$
$totalMovilidad \leftarrow movimientosJugador + movimientosContrario$
si $movimientosJugador > movimientosContrario$ **entonces**
  $puntuacion \leftarrow 100 * (movimientosJugador/totalFichas)$
si no si $movimientosJugador < movimientosContrario$ **entonces**
  $puntuacion \leftarrow -100 * (movimientosContrario/totalFichas)$
si no
  $puntuacion \leftarrow 0$
fin si
  devolver $puntuacion$

Algoritmo 5 Evaluador de la diferencia de esquinas entre los jugadores en el tablero

Entrada: $juego, jugador$
Salida: $puntuacion$
$jugadorContrario \leftarrow juego.obtenerJugadorContrario(jugador)$
\# Obtener esquinas de cada jugador
$movimientosJugador \leftarrow juego.obtenerEsquinasJugador(jugador)$
$movimientosContrario \leftarrow juego.obtenerEsquinasJugador(jugadorContrario)$
\# Calcular puntuacion tras la evaluación
$puntuacion \leftarrow 25 * esquinasJugador - 25 * esquinasContrario$
devolver $puntuacion$

Algoritmo 6 Evaluador de la diferencia de fichas en casillas adyacentes a las esquinas entre los jugadores en el tablero

Entrada: $juego, jugador$
Salida: $puntuacion$
$jugadorContrario \leftarrow juego.obtenerJugadorContrario(jugador)$
\# Obtener adyacentes de cada jugador
$adyacentesJugador \leftarrow juego.obtenerAdyacentesEsquinasJugador(jugador)$
$adyacentesContario \leftarrow juego.obtenerAdyacentesEsquinasJugador(jugadorContrario)$
\# Calcular puntuacion tras la evaluación
$puntuacion \leftarrow -12{,}5 * esquinasJugador + 12{,}5 * esquinasContrario$
devolver $puntuacion$

Algoritmo 7 Evaluador que junta todos evaluadores anteriores

Entrada: $juego, jugador$
Salida: $puntuacion$
$fichas \leftarrow evaluarFichas(juego, jugador)$
$esquinas \leftarrow evaluarEsquinas(juego, jugador)$
$adyacentes \leftarrow evaluarAdyacentes(juego, jugador)$
$movilidad \leftarrow evaluarMovilidad(juego, jugador)$
$puntuacion \leftarrow fichas + esquinas + adyacentes + movilidad$
devolver $puntuacion$

2.5. Tablas de transposición

Con los conceptos explicados anteriormente logramos implementar un método de Inteligencia Artificial funcional para jugar al Othello. Sin em-

bargo, es posible mejorar la eficiencia del algoritmo minimax con poda alfa beta y profundidad acotada observando que, durante la búsqueda, es muy probable que se encuentren estados repetidos (dicho de otro modo, el espacio de búsqueda contiene caminos redundantes, lo que permite alcanzar el mismo estado del juego con distintas secuencias de movimientos). Además, durante una partida el jugador MAX ha de decidir qué movimiento realizar múltiples veces, lo que se traduce en múltiples búsquedas revisitando algunos de los nodos del árbol de juego. Esto es así ya que cada vez que el jugador inteligente tiene que realizar una colocación, recorre (hasta la cota de profundidad dada) todos los estados del tablero de juego que deriven de todas las posibles colocaciones de las que dispone, pudiendo encontrarse en muchas ocasiones con situaciones ídenticas que ya fueron evaluadas en una toma de decisiones previa o incluso en toma de decisiones actual en una colocacion anteriormente evaluada.

Para evitar evaluaciones innecesarias se emplean las llamadas tablas de transposición [2] que mantienen un registro de los estados del tablero visitados y los resultados obtenidos tras la búsqueda de la mejor colocación.

Si combinamos el algoritmo minimax con poda alfa-beta visto anteriormente con las tablas de transposición, conseguimos que cada vez que el algoritmo analice uno de los nodos del árbol de juego primero verifique si se encuentra ya en la memoria, evitandose así realizar los cálculos pertinentes (empleará los valores ya almacenados) reduciendo de esta forma el tiempo de ejecución.

Hay que tener en cuenta que comparar todas las posiciones del tablero una a una para determinar si dos situaciones del tablero son ídenticas o no es un proceso muy costoso. Para acelerar las comparaciones se emplea un valor hash.

2.5.1. Elementos básicos de una tabla de transposición

Llaves Zobrist

La función hash utilizada en programas que juegan juegos de tablero como el Othello es el hashing de Zobrist[2]. Este hashing hace uso de las llaves con su mismo nombre, Zobrist keys.

Una Zobrist key es un conjunto de patrones de bits de longitud fija almacenados para cada estado posible de cada ubicación posible en el tablero. El Othello tiene 64 casillas, y cada casilla puede estar vacía o tener una de 2 piezas diferentes en ella, cada una de uno de los dos colores posibles, los colores que identifican a los jugadores. Para conseguir esto la Zobrist key necesita ser un array de 64 x 2 = 128 entradas de longitud, lo que se traduce

en dos posiciones del array por celda del tablero de juego, cada una para el color de un jugador.

Las llaves Zobrist necesitan ser inicializadas con cadenas de bits aleatorios del tamaño apropiado[2].

Una vez inicializada la llave Zobrist se necesita una función que calcule el hash de cada estado del tablero en los momentos requeridos, el algoritmo 8. Este algoritmo calcula la función hash recorriendo cada celda del tablero y en cuanto encuentra una ficha de alguno de los dos jugadores realiza un cálculo teniendo en cuanta dos valores:

1. **(fila · CELDAS + columna) · 2:** Este cálculo permite indentificar las dos posiciones del array que corresponden a la fila y la columna (en resumen, la celda) en la que hemos encontrado la ficha.

2. **(ficha - 1):** Este cálculo permite identificar cuál de las dos posiciones del array correspondientes a la celda obtenida en el cálculo anterior es la correcta para la ficha leída del tablero de juego, los jugadores viene representados por los valores binarios 0 y 1 (siendo 0 las negras y 1 las blancas).

Algoritmo 8 Cálculo de la función de hashing de Zobrist

```
resultado ← 0
para fila ← 0 hasta CELDAS − 1 hacer
    para columna ← 0 to CELDAS − 1 hacer
        ficha ← obtenerValorCelda(fila, columna)
        si ficha ≠ None entonces
            indice ← (fila ∗ CELDAS + columna) ∗ 2 + ficha
            resultado ← resultado ⊕ zobristKey[indice]
        fin si
    fin para
fin para
devolver resultado
```

Entrada de la tabla de transposición

La tabla de transposición es una tabla hash, en los que en cada par clave-valor, siendo la clave el hash de Zobrist anteriormente explicado, representa un estado del tablero y el valor de la entrada contiene 4 elementos que se explican a continuación:

- **Puntuación:** Valor numérico obtenido de la evaluación para el estado del tablero representado por el valor hash. Cuanto mayor sea la puntuación, más favorable será el movimiento obtenido para MAX.
- **Tipo de puntuación:** Indica si la puntuación obtenida es precisa, un fallo bajo o un fallo alto:
  - **Precisa:** La puntuación es el valor exacta obtenida tras la evaluacion del estado del tablero, ya que se encuentra dentro de los límites de la ventana alfa-beta actual.
  - **Fallo bajo:** La puntuación obtenida se sale de la ventana alfa-beta por ser menor que el valor de alfa.
  - **Fallo alto:** La puntuación obtenida se sale de la ventana alfa-beta por ser mayor que el valor de beta.
- **Mejor movimiento:** Se trata del movimiento elegido por MAX, en el estado actual del tablero, tras su evaluación.
- **Profundidad:** Se almacena la profundidad de la búsqueda en el árbol de juego en la cual se evaluó la puntuación y se determinó el mejor movimiento.

2.5.2. Método de búsqueda final

El algoritmo 9 muestra el método final de búsqueda con adversarios utilizado por el jugador MAX para adoptar una decisión, incorporando profundidad acotada con evaluación heurística, poda alfa beta y tablas de transposición a la búsqueda minimax.

Algoritmo 9 Algoritmo minimax con profundidad acotada, poda alfa-beta y tablas de transposición

Entrada: juego, max, profundidad, alfa, beta
Salida: mejorPuntuacion, mejorMovimiento

```
#Obtener valor hash del estado actual del tablero
valorHash ← juego.hash()
#Obtener entrada de la tabla de transposición
entrada ← tablaTransposicion.obtenerEntrada(valorHash)
#Comprobar que halla entrada en la tabla y su profundidad
si entrada and entrada.getProfundidad() >= profundidad entonces
    #Usar la puntuación almacenada
    si entrada.getTipoPuntuacion() == PRECISA entonces
        devolver entrada.getPuntuacion(), entrada.getMejorMovimiento()
    si no si entrada.getTipoPuntuacion() == FALLO_ALTO and
entrada.getPuntuacion < beta entonces
        #Actualizar beta
        beta ← entrada.getPuntuacion()
    si no si entrada.getTipoPuntuacion() == FALLO_BAJO and
entrada.getPuntuacion > alfa entonces
        #Actualizar alfa
        alfa ← entrada.getPuntuacion()
    fin si
    si alfa >= beta entonces
        #Podar rama del árbol de juego
        devolver entrada.getPuntuacion(), entrada.getMejorMovimiento()
    fin si
fin si
#Comprobar fin de la recursión
si juego.finJuego() o profundidad == MAX_PROFUNDIDAD entonces
     devolver juego.evalua(max), null
fin si
#Inicializar valores
si juego.jugadorActivo() == max entonces
    mejorPuntuacion ← −∞
si no
    mejorPuntuacion ← ∞
fin si
mejorMovimiento ← null
tipoPuntuacion ← PRECISA
```

```
para movimiento en juego.obtenerMovimientos() hacer
    #Colocar ficha
    nuevoJuego ← juego.colocaFicha(movimiento)
    #Aplicar recursión
    puntuacion, _ ← minimax(nuevoJuego, max, profundidad − 1, alfa, beta)
    #Actualizar valores
    si juego.jugadorActivo() == max entonces
        si puntuacion > mejorPuntuacion entonces
            mejorPuntuacion ← puntuacion
            mejorMovimiento ← movimiento
            beta ← min(beta, puntuacion)
            si beta <= alfa entonces
                #Poda alfa-beta
                tipoPuntuacion ← FALLO_ALTO
                break
            fin si
        fin si
    si no
        si puntuacion < mejorPuntuacion entonces
            mejorPuntuacion ← puntuacion
            mejorMovimiento ← movimiento
            alfa ← max(alfa, puntuacion)
            si alfa >= beta entonces
                #Poda alfa-beta
                tipoPuntuacion ← FALLO_BAJO
                break
            fin si
        fin si
    fin si
fin para
#Almacenar nueva puntuación y mejor movimiento en la tabla
nuevaEntradaEntradaTabla(valorHash, mejorPuntuacion, tipoPuntuacion,
mejorMovimiento, profundidad)
tablaTransposicion.almacenarEntrada(nuevaEntrada)
    devolver mejorPuntuacion, mejorMovimiento
```

Capítulo 3

Diseño software

Este capítulo contiene todos los aspectos relativos al diseño de la aplicación para jugar al Othello incorporando un jugador basado en Inteligencia Artificial, previos a la fase de desarrollo de la aplicación. Comenzaremos detallando los requisitos, continuaremos con las herramientas y tecnologías a utilizar y terminaremos con la metodología adoptada.

3.1. Requisitos

En primer lugar, tenemos los requisitos funcionales, que definen cuál ha de ser el comportamiento de la aplicación observado por el usuario final (es decir, su funcionamiento) [8]

3.1.1. Requisitos Funcionales

Comenzamos especificando los requisitos que debería cumplir el sistema software o aplicación a construir, de manera que se alcancen los objetivos planteados en el capítulo 1.

RF.1: El juego será jugado por dos jugadores, el jugador 1 y el jugador 2.

RF.2: El juego se jugará en un tablero con 64 casillas, 8 filas (numeradas del A a la H) x 8 columnas (numeradas del 1 al 8).

RF.3: La aplicación permitirá a dos jugadores jugar al Othello

RF.4: El juego deberá notificar el ganador, en caso de haberlo, o que ha habido empate en caso contrario.

RF.5: El juego deberá contar con una interfaz gráfica.

RF.6: El usuario podrá navegar por el juego empleando el ratón.

RF.7: La interfaz gráfica indicará cuál es el jugador activo en el turno.

RF.8: El juego contará con un registro de movimientos, por cada colocación que se haga se mostrará en la interfaz el jugador que lo ha realizado y la casilla donde ha colocado su ficha (letra,número).

RF.9: Cuando la partida acabe la interfaz gráfica deberá contar con un botón para guardar el estado final de la partida.

RF.10: El juego podrá ser jugador por un jugador humano y un jugador inteligente (una inteligencia artificial integrada en el juego).

RF.11: La aplicación deberá tener un modo desarrollador (sin interfaz gráfica) para poder lanzar partidas enfrentando a 2 jugadores basados en inteligencia artificial.

RF.12: El juego contará con un menú principal que mostrará el nombre del juego (OTHELLO).

RF.13: El usuario podrá navegar por el menú empleando el ratón.

RF.14: El menú principal se podrá elegir el modo de juego entre dos: jugador humano contra jugador humano o jugador humano contra jugador inteligente.

3.1.2. Requisitos No Funcionales

Un requisito no funcional puede describirse como un atributo relativo a la calidad, el rendimiento o la seguridad del sistema o, en general, una restricción genérica sobre el sistema software a construir [Pressman2020]. En nuestro caso, los requisitos no funcionales son los siguientes:

RNF.1: El estilo visual debe ser *pixelart* [1].

RNF.2: En partidas humano contra jugador inteligente, la toma de decisiones por parte de la inteligencia artificial ha de ser rápida para que el ritmo de la partida sea ágil.

[1] Técnica de arte digital que consiste en editar imágenes a nivel de píxel.

3.2. Diagrama de casos de uso

En la figura 3.1 se observa el caso de uso del menú principal. Además se ha en la tabla 3.1 se ha incorporado la plantilla para el caso "Jugar Othello jugador contra inteligencia artificial".

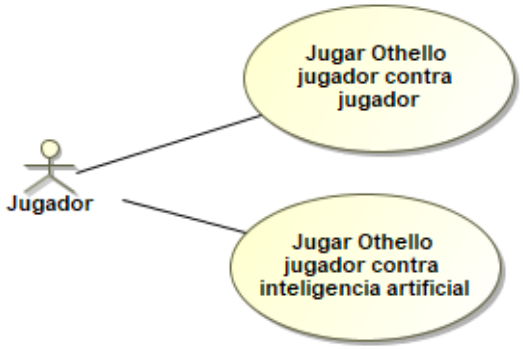

Figura 3.1: Diagrama de casos de uso del menú principal.

Tabla 3.1: Plantilla para el caso de uso "Jugar Othello jugador contra inteligencia artificial"

Nombre	Jugar Othello jugador contra inteligencia artificial.
Descripción	La aplicación genera la interfaz necesaria para jugar al Othello jugador contra inteligencia artificial.
Actores primarios	Jugador
Actores secundarios	-
Eventos de activación	El caso de uso comienza cuando el Jugador selecciona la opción de jugar contra una inteligencia artificial (1 vs IA) en el menú principal.
Precondiciones	-
Flujo principal	1. La aplicación genera un jugador inteligente. 2. La aplicación genera la interfaz gráfica necesaria. 3. La aplicación elimina la interfaz gráfica del menú principal.
Postcondiciones	-
Flujos alternativos	-

3.3. Tecnologías y herramientas

El desarrollo se ha llevado a cabo en el Visual Studio Code. Se ha empleado GitHub como sistema de control de versiones.

El lenguaje de programación elegido para realizar el desarrollo completo de la aplicación es Python. Uno de los motivos es que es un lenguaje simple, pero que cuenta con gran cantidad de librerías que expanden enormemente las posibilidades. En añadido, admite programación orientada a objetos, una característica muy útil para estructurar el software de manera eficiente y escalable. Además, como se puede observar en la Figura 3.2, es uno de los lenguajes más usados actualmente. Además, al ser un lenguaje sencillo y flexible, permite desarrollar prototipos y probar distintos algoritmo de manera muy ágil. Por último, Python es un lenguaje interpretado/multiplataforma por lo que la aplicación podrá funcionar en cualquier sistema operativo [9].

Figura 3.2: Lenguajes de programación más usados en 2023.
https://spectrum.ieee.org/the-top-programming-languages-2023

La implementación del menú principal y la interfaz del juego se ha desarrollado mediante la librería de Python Pygame que proporciona un conjunto de módulos para la creación de videojuegos en 2 dimensiones orientada al manejo de *sprites*.

Para la creación y modificación de los *sprites*[2] se ha empleado el software Aseprite, un editor gráfico orientado a *pixelart* en dos dimensiones, ya que es un software simple de usar para gente con poca experiencia en la creación de pixelarts, pero a su vez es muy completo en cuanto a herramientas.

[2]En el ámbito de los gráficos por computador y videojuegos se trata de un mapa de bits dibujados en la pantalla de ordenador que representan personajes u objetos.

3.4. Metodología

El desarrollo de la aplicación se ha realizado siguiendo una metodología iterativa e incremental. Esta metodología, como indica su nombre, sigue dos principios fundamentales:

- **Iterativa:** El desarrollo se divide en distintos bloques temporales llamados iteraciones [10]. Como podemos observar en la Figura 3.3, en cada iteración se realiza el mismo ciclo de trabajo (requisitos, diseño, desarrollo, pruebas...) hasta conseguir desarrollar una versión funcional y entregable.
- **Incremental:** En cada iteración el proyecto va creciendo y/o evolucionando, empleando el conocimiento optenido de iteraciones anteriores.

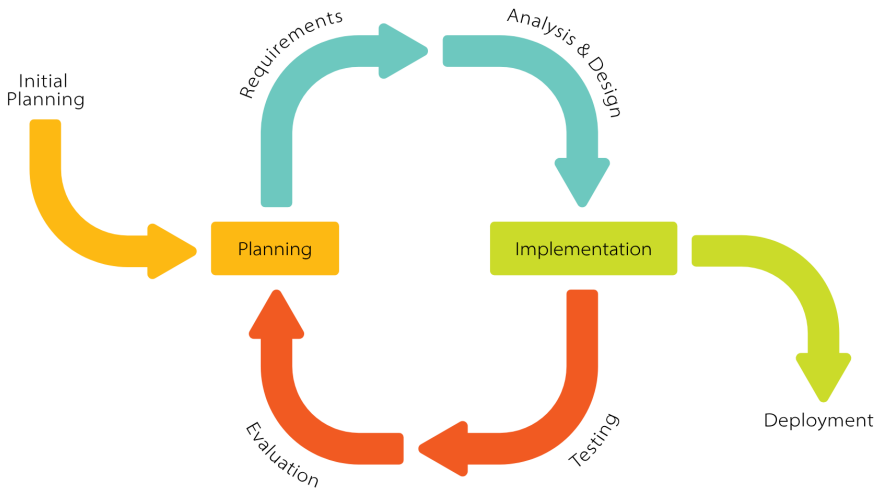

Figura 3.3: Diagrama del ciclo de trabajo de una iteración de la metodología iterativa e incremental.
Krupadeluxe - Own work, CC BY-SA 4.0,
https://commons.wikimedia.org/w/index.php?curid=104924876

3.4.1. Iteraciones

El desarrollo del proyecto se ha dividido en cuatro iteraciones, que cubren todos lo requisitos previamente expuestos:

Iter.1 Desarrollo del motor de juego con las funcionalidades básicas necesarias para una partida de Othello. Abarca los requisitos del **RF.1** al **RF.4**.

Iter.2 Desarrollo de la interfaz gráfica del juego para partidas entre dos jugadores humanos. Abarca los requisitos del **RF.5** al **RF.9**.

Iter.3 Desarrollo de un jugador inteligente y un modo desarrollador para realizar las pruebas referentes a la inteligencia artificial. Abarca los requisitos **RF.10** y **RF.11**.

Iter.4 Desarrollo de un menú principal en la interfaz gráfica y adaptación de la interfaz para introducir la posibilidad del enfrentamiento contra la inteligencia artificial. Abarca los requisitos del **RF.12** al **RF.14**.

En el repositorio de Git[3] se encuentran todos los archivos necesarios para la ejecución de la aplicación tras la última iteración.

[3]https://github.com/sandraalegriaa/TFG

Capítulo 4

Desarrollo Software

4.1. Arquitectura Software

El proyecto se ha estructurado mediante una arquitectura en capas. Se ha elegido esta arquitectura ya que por su simplicidad, es recomendable para proyectos con restricciones significativas de tiempo[11], como es el caso de este proyecto. Además, en la aplicación hay una clara separación de las tareas entre clases. Las reglas de juego del Othello podrían varíar y no afectaría de ninguna forma a la interfaz, igual que pasaría en la situación inversa. Esta situación es muy interesante ya que en una arquitectura en capas, cada capa tiene un papel diferente a las demás del que responsabilizarse dentro de la aplicación. Esto hace que una arquitectura en capas sea óptima para este trabajo.

El número de capas empleadas puede variar según el proyecto, la mayoría de las arquitecturas en capas constan de cuatro capas estándar: presentación, negocio, persistencia y base de datos[11]. Sin embargo, debido a la naturaleza de esta aplicación, solamente resultan necesarias dos capas:

- **Capa de presentación:** Esta capa se corresponde con la interfaz gráfica de la aplicación, con la que el usuario interactúa. En el juego desarrollado, correspondería al menú principal al comienzo del juego y a la interfaz gráfica con el tablero y las fichas para jugar al Othello.
- **Capa de lógica del negocio:** En esta capa se reciben las peticiones del usuario y se envían las respuestas tras el proceso ya que es la capa en la que se establecen las reglas que deben cumplirse. En la aplicación desarrollada, en esta capa se integrarían las clases encargadas de implementar el motor de juego, encargado de gestionar los turnos y asegurarse del cumplimiento de las normas del Othello, así como las

clases correspondientes a ambos jugadores (incluyendo jugador humano y jugador no humano basado en Inteligencia Artificial).

4.2. Diagrama de clases

En la Figura 4.1 se encuentra el diagrama de clases de la aplicación. Por claridad no se indican los atributos y métodos de cada clase ya que resultaría en un diagrama demasiado extenso y de difícil visualización.

Como se ha explicado anteriormente, se ha empleado Pyhton para la implementación de la aplicación, por ello se han seguido los estándares marcados en dicho lenguaje de programación. Se destacará sobre todos el siguiente, en Python no existen los atributos y métodos privados al uso, es decir, aquellos a los que solo se puede acceder desde el propio objeto de la clase en cuestión. En consecuencia, debido a un convenido altamente extendido por los programadores que usan este lenguaje, todos aquellos métodos y atributos que deban ser tratados como tal se declaran con un nombre predecedido de un guión bajo ("_") [12].

Comentamos a continuación las clases más relevantes. La clase **InterfazOthello** se encarga de gestionar los eventos de clic generados por el usuario mediante la interacción con la aplicación. Por la forma que tiene Pygame de gestionar los eventos, es la clase **InterfazOthello** la encargada de gestionar la partida. Esto lo realiza a través de un bucle que se mantiene activo durante el uso de la aplicación. Cuando captura un evento de clic comprueba si se encuentra dentro de un elemento interactuable y, en caso afirmativo, llama a los métodos pertinentes para su correcta resolución. Para los eventos generados dentro del área correspondiente al tablero de juego, a pesar de ser la clase que gestiona el orden de actuación, no realiza ningún cálculo referente a la lógica del Othello, todo lo ello queda delegado en la clase **MotorDeJuego**.

La clase **InterfazOthello** también es la encargada de proporcionar al usuario información sobre el estado de la partida, que recoge de **MotorDeJuego**, como el tablero con el estado actual, el jugador que debe realizar una colocación y el registro de movimientos. Cuando la partida se dé por finalizada de alguna de las dos formas posibles también mostrará el resultado de la misma.

Para plasmar esa información en forma de interfaz emplea las clases **Ficha** y **FichaSemitransparente** que se encargan de cargar los *sprites* de las fichas y generar el *asset*[1] necesario.

MotorDeJuego es la clase en la que se establecen las reglas que debe seguir la partida, siendo fiel a las reglas clásicas del Othello. Además, ges-

[1]Representación de cualquier elemento que puede ser utilizado en un juego.

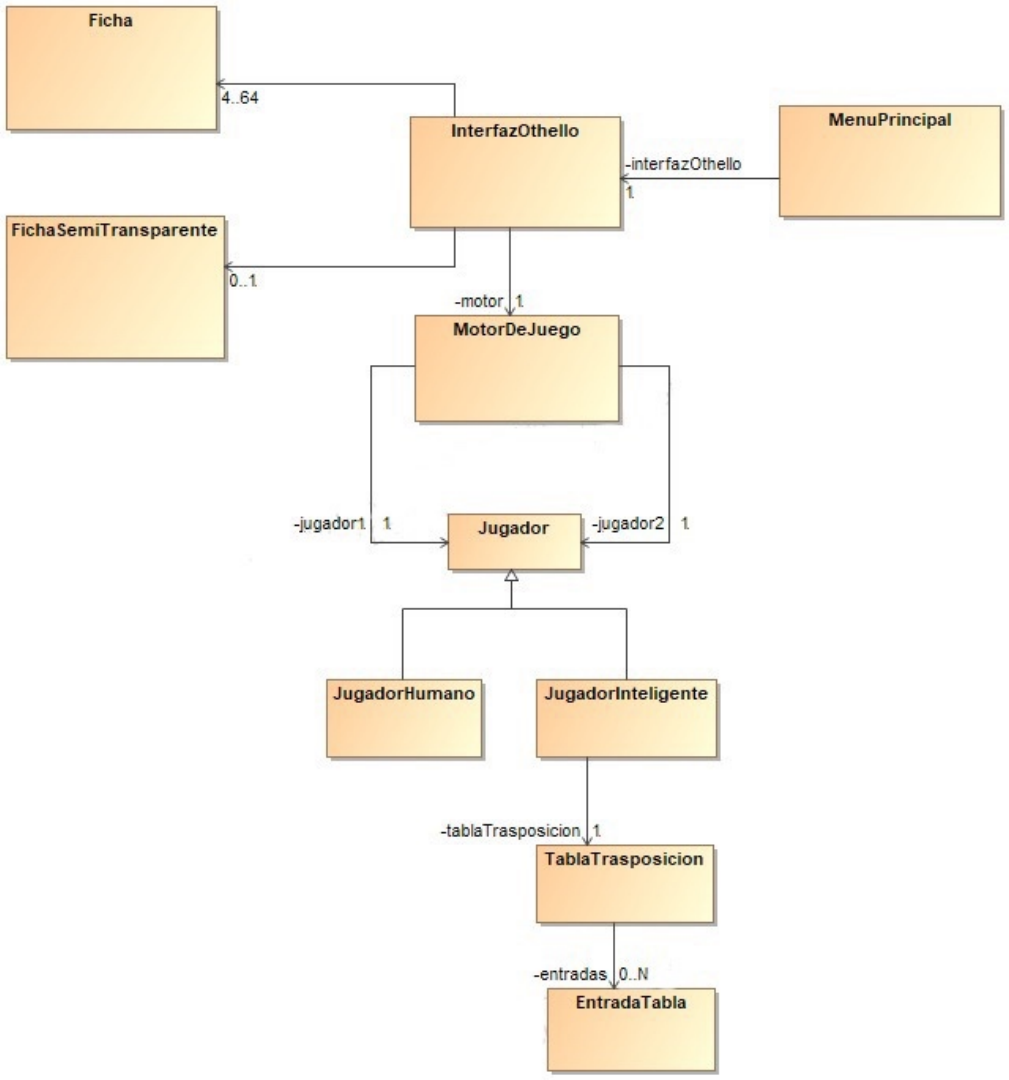

Figura 4.1: Diagrama de clases de la aplicación.

tiona los cambios de turno entre jugadores. En ella se encuentra el tablero que refleja el estado de la partida, así como los métodos necesarios para la colocación de fichas, volteo de las fichas encerradas, llevar la cuenta de la cantidad de fichas que tiene cada jugador en todo momento para declarar un empate o la victoria a uno o a otro.

La clase **Jugador** contiene los dos atributos obligatorios indistintamente para los jugadores, sean humanos o inteligentes (una inteligencia artificial), a saber, color y turno. Color hace referencia al color que representa las fichas del jugador en cuestión y turno indica el lugar que ocupa el jugador en

cuanto al orden de colocación. Incluye también los observadores asociados a estos dos atributos; no se implementan modificadores ya que estos valores son inamovibles una vez creados los objetos.

El usuario gestiona sus colocaciones mediante el mecanismo explicado anteriormente en la interfaz por lo que la clase **JugadorHumano** solo hereda los atributos y métodos de su superclase **Jugador**.

La clase **JugadorInteligente** implementa la inteligencia artificial anteriormente explicada (el algoritmo minimax con poda alfa-beta modificado con tablas de transposición). La lógica referente a las tablas de transposición se encuentra en las clases **TablaTransposición** y **EntradaTabla**. Cada entrada en la tabla almacena un valor hash, correspondiente a un estado del tablero de juego y el mejor movimiento encontrado para el estado, además de la puntuación obtenida en la evaluación. La tabla de transposición es en esencia una lista de entradas de tabla.

Por último, la clase **MenuPrincipal** se encarga de generar la ventana que actúa como menú, en la que el usuario puede elegir la modalidad de juego que quiere ejecutar.

Una vez indicadas las clases de la aplicación podemos señalar que clase corresponde a cada capa en la arquitectura.

- **Capa de presentación:** En el juego desarrollado en este proyecto corresponde a las clases **MenuPrincipal** e **InterfazOthello**.
- **Capa de lógica del negocio:** En el juego desarrollado en este proyecto corresponde a las clases **MotorDeJuego**, **Jugador** (además de sus subclases **JugadorHumano** y **JugadorInteligente**), **TablaTransposicion** y **EntradaTabla**.

4.3. Interfaz gráfica

La parte gráfica de la aplicación se puede dividir en dos interfaces diferentes: el menú principal (figura 4.2) y la interfaz de Othello (figura 4.3). Ambas tienen las mismas dimensiones, 1100x700 píxeles y emplean el *pixelart* como estílo artístico, según el requisito RNF1..

Como se ha mencionado anteriormente en la sección 3.3 las interfaces se han desarrollado empleando la librería Pygame.

Los *sprites* utilizados han sido obtenido de Itch.io[2] y Kenny[3] y algunos de ellos como el tablero de juego ha sido modificado empleando Aseprite

[2]https://dimexel.itch.io/classic-board-game
[3]https://kenney.nl/assets

Figura 4.2: Menú principal de la aplicación.

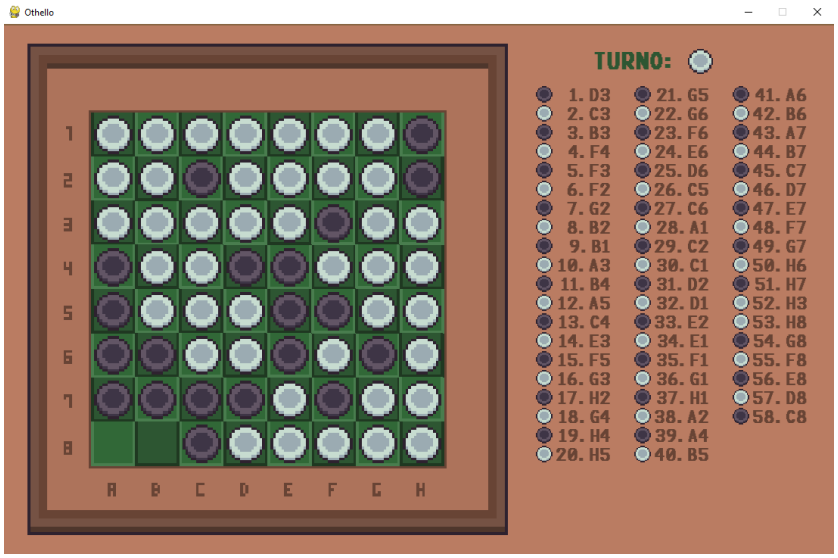

Figura 4.3: Interfaz de Othello de la aplicación.

para ser más fieles a la estética buscada en la aplicación y al Othello. El *sprite* que ha sufrido más modificaciones ha sido el tablero de juego, como se puede observar en las figuras 4.4 y 4.5, esta decisión se ha tomado en primer lugar para que los colores se correspondan con los originales del Othello, y por último para que las etiquetas referentes a las celdas sean más simples para el usuario. Sin embargo algunos otros como las fichas de los jugadores

se mantienen idénticas y otros como el *sprite* de guardado han sido realizado de cero observando imagenes de referencia.

Tanto en el menú principal como en la interfaz de Othello encontramos textos con información para el usuario. Dichos textos emplean la fuente Karma Suture, obtenida en 1001 Free Fonts para preservar el estilo *pixelart*.

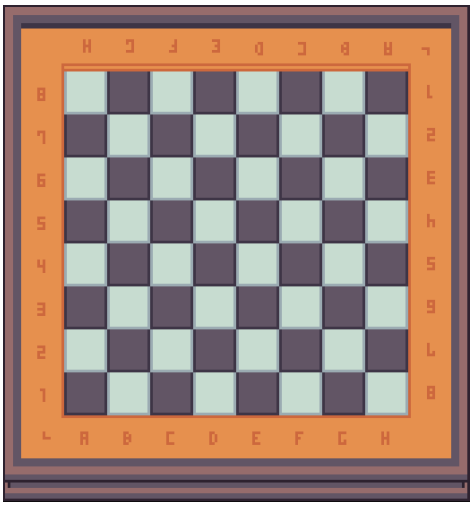

Figura 4.4: Tablero de juego obtenido en Itch.io.

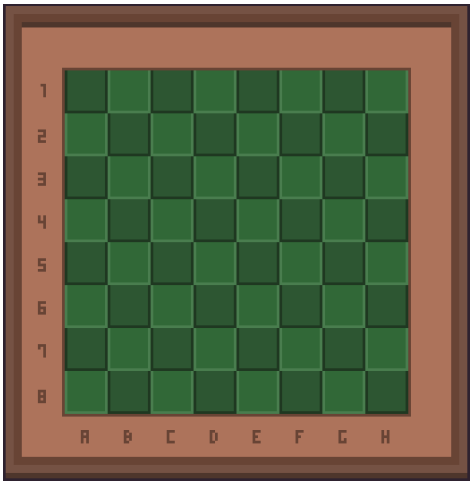

Figura 4.5: Tablero de juego modificado con Aseprite.

Capítulo 5

Pruebas

5.1. Pruebas del evaluador

Para poder analizar el rendimiento del evaluador empleado en la aplicación (explicado en la sección 2.4) se ha desarrollado un evaluador aleatorio empleando la librería random de Python. Este evaluador simula un usuario principiante que podría emplear la aplicación, con los conocimientos suficientes del juego para poder desarrollar una partida con normalidad pero sin mucho, o casi ningún conocimiento de la estrategia del mismo.

Para realizar las pruebas se han tomado resultados de profundidades razonables para el algoritmo 2. Es decir, se empieza a observar a partir de 2 de profundidad, que a pesar de ser demasiado reducida, al menos explora un turno del jugador inteligente y uno del rival. La última profundidad explorada es 9 ya que más allá de este valor el coste temporal sería excesivo.

Por cada valor de profundidad se han lanzado 30 partidas entre dos jugadores no humanos, uno de ellos usando la función de evaluación del algoritmo 7 (denominado Evaluador) y el otro (denominado Aleatorio), evaluando los estados con un número aleatorio entre -100 y 100, registrando el número de partidas ganadas por cada jugador, así como los empates producidos.

Por último, se ha añadido una columna en la tabla (**Empates**) para los posibles empates que pueda haber entre ambos jugadores ya que según las reglas del Othello son una posibilidad.

En la tabla 5.1 podemos ver los resultados obtenidos para un total de 30 partidas por profundidad. También encontramos el resultado de calcular los porcentajes de victorias de cada jugador y los porcentaje de empates para todas las profundidades analizadas.

Tabla 5.1: Número de victorias (junto al porcentaje) del evaluador en diferentes profundidades

Profundidad	**Evaluador**	**Aleatorio**	**Empates**
2	23 (76,67 %)	6 (20 %)	1 (3,33 %)
3	26 (86,67 %)	4 (13,33 %)	0 (0 %)
4	23 (76,67 %)	7 (23,33 %)	0 (0 %)
5	24 (86,67 %)	6 (20 %)	0 (0 %)
6	23 (76,67 %)	5 (16,67 %)	2 (6,67 %)
7	18 (60 %)	11 (36,67 %)	1 (3,33 %)
8	19 (63,33 %)	9 (30 %)	2 (6,67 %)
9	28 (93,33 %)	2 (6,67 %)	0 (0 %)
Media	**23**	**6,25**	**0,75**

El porcentaje de victoria obtenido para el evaluador implementado es de 77,50 % frente al 20,83 % del evaluador aleatorio. De esta forma obtenemos que nuestro evaluador sale victorioso un 56,67 % más que el evaluador aleatorio. Estos datos indican que nuestro evaluador es correcto por dos cuestiones, la primera es que gana con el suficiente margen para considerarse mejor que una selección sin pautas definidas, como podría hacer un jugador del Othello inexperto. En segundo lugar, es un evaluador que deja margen al jugador para poder ganar alguna de las partidas, de esta forma queda algo equilibrado y se evitan así posibles frustraciones del usuario al enfrentarse contra un jugador imbatible.

Cabe destacar que las pruebas aquí realizadas son independientes de los siguientes apartados, en los que trataremos aspectos de las tablas de transposición. Esto se debe a que las tablas de transposición no influyen de ninguna manera en la evaluación, por ello estos análisis no quedarán invalidados con ninguna de las pruebas posteriores.

5.2. Pruebas tamaño de la tabla de transposición

Una vez se ha demostrado que la implementación de la tabla de transposición es beneficiosa, se debe encontrar el mejor tamaño para la tabla de transposición, es decir, de cuentas entradas dispone.

Para ello se ha tenido en cuenta que se va a utilizar el evaluador combinado por tanto solo es necesario realizar las pruebas sobre este evaluador.

Tabla 5.2: Tiempos de ejecución por movimiento del evaluador en función del tamaño de tabla.

Profundidad	$16 \cdot 10^6$	$2 \cdot 10^6$	$2 \cdot 10^5$
2	0,02	0,02	0,02
3	0,11	0,12	0,10
4	0,20	0,19	0,20
5	1,07	0,95	1,05
6	1,64	1,56	2,06
7	7,60	8,41	12,98
8	13,68	13,44	15,56
9	85,84	81,61	89,89

Se ha decidido representar los valores en una gráfica para mejorar la comprensión de la elección realizada. Adicionalmente,

En la Figura 5.1 se incluye una gráfica con algunos de los datos de la tabla 5.3, se ha decidido no incluir en la gráfica las profundidades: 2, 3, 4 por ser demasiado reducidas, obteniendo de esta manera peores decisiones y la 9 por necesitar demasiado tiempo para cada movimiento, empeorando la experiencia del usuario.

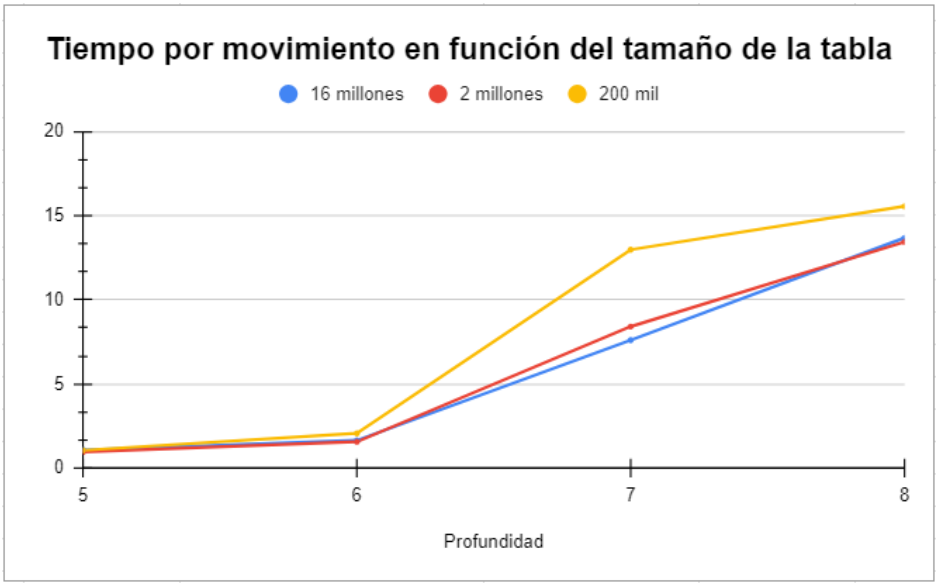

Figura 5.1: Gráfica con tiempos por movimiento para el evaluador combinado en función de la profundidad y el tamaño de la tabla(con transposición).

Finalmente se ha decidido emplear 200 mil entradas en la tabla de transposición. Esta decisión se ha tomado por tres razones:

- Se ha decidido emplear la profundidad 6 para la ejecución de la aplicación contra el usuario. Esto se debe a que se ha considerado que la experiencia del usuario empeoraría notablemente a partir de 4 segundos por movimiento. Al ser 6 la profundidad más elevada por debajo de esa condición, es la que se va a implementar.
- En la gráfica 5.1 vemos que una tabla con $16 \cdot 10^6$ o $2 \cdot 10^6$ de entradas emplean tiempos bastante similares por movimiento en todas las profundidades analizadas. Dando ambas mejores resultados que una tabla de transposición con $2 \cdot 10^5$ entradas. Sin embargo, para la profundidad 6 la diferencia es mínima por lo que se puede asumir.
- Por último, a pesar de no dar los mejores resultados, una tabla con $2 \cdot 10^5$ entradas necesita bastante menos memoria que una tabla con $2 \cdot 10^6$ o $16 \cdot 10^6$ de entradas.

5.3. Pruebas tabla de transposición

Una vez implementadas las tablas de transposición se realizan estos análisis para cerciorarnos de que están aportando beneficios a la aplicación. En la tabla 5.3 se han recogido los tiempos por movimiento para los dos evaluadores involucrados en la prueba anterior, tanto con transposición como sin ella para poder analizar de esta manera los resultados obtenidos. Cabe destacar que para mayor precisión cada valor mostrado es la media obtenida en 30 partidas llevadas a cabo en las mismas condiciones y características.

Al observar la tabla se puede afirmar que la tabla de transposición está mejorando el tiempo en la ejecución de cada movimiento, para el evaluador combinado, en las profundidades de valor 6 en adelante. Esta afirmación puede ser percibida en la gráfica 5.2. Sin embargo para el evaluador aleatorio está ocurriendo todo lo contrario, es decir, en vez de ver una mejora temporal se ve que hay un incremento en los tiempos medios necesarios para cada movimiento. Esto se debe a que en el jugador con evaluador aleatorio el coste de la evaluación de estados terminales es insignificante, por lo que el ahorrar algunas evaluaciones no compensa el coste adicional de mantener la tabla de transposición.

Tabla 5.3: Tiempos de ejecución por movimiento de los evaluadores (combinado - aleatorio) en diferentes profundidades.

Profundidad	Con transposición(s)	Sin transposición(s)
2	0,02 - 0,01	0,02 - 0,00
3	0,12 - 0,02	0,10 - 0,01
4	0,19 - 0,06	0,17 - 0,04
5	0,95 - 0,20	0,92 - 0,11
6	1,56 - 0,47	1,92 - 0,40
7	8,41 - 1,87	9,02 - 1,10
8	13,44 - 4,04	17,61 - 2,95
9	81,61 - 10,21	99,05 - 9,17

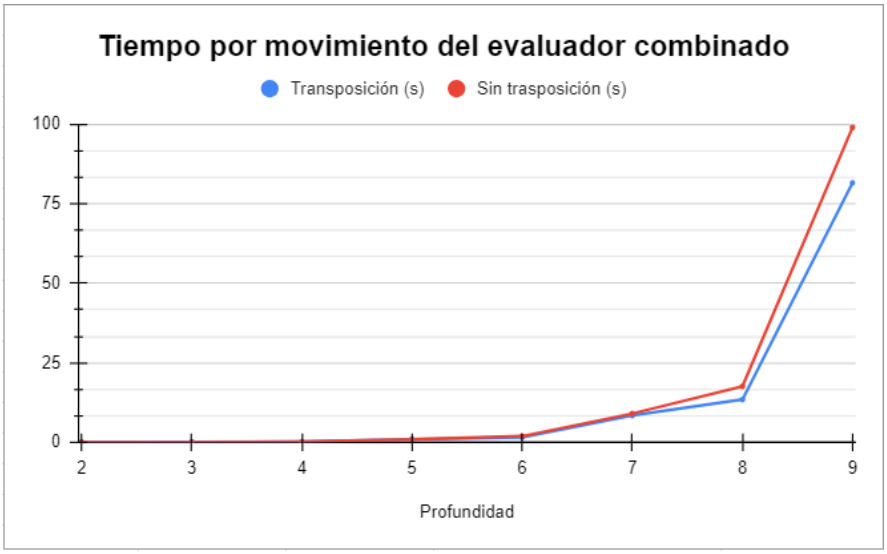

Figura 5.2: Gráfica con tiempos por movimiento para el evaluador combinado en función de la profundidad (con y sin transposición).

Por la forma que tiene el algoritmo minimax de encontrar el mejor movimiento cuanta más profundidad se emplee, el algoritmo encontrará el mejor movimiento de manera más precisa. Sin embargo, a medida que incrementa la profundidad, se incrementa el tiempo de ejecución del algoritmo, esto no es importante para partidas con dos jugadores inteligentes pero si para partidas que involucran un jugador humano, que puede sentir rechazo hacia la aplicación si esta tarda demasiado tiempo en realizar los turnos del contrario. En dicho caso podemos decir que, para valores bajos de la cota de profundidad (menores que 6, correspondientes a a lo sumo 3 rondas de

2 turnos/jugadores) el coste extra de mantener la tabla de transposición no compensa el ahorro computacional que pueda producirse en la búsqueda al evitar evaluar de manera repetida el mismo estado. En cambio, al aumentar el nivel de profundidad, el número de estados repetidos que se puede encontrar en el árbol de búsqueda aumenta lo suficiente como para que el coste computacional adicional incurrido por la tabla de transposición se vea compensado por la reducción del coste en las evaluaciones repetidas. Por lo tanto, interesa incorporar el uso de tabla de transposición a la búsqueda mínimax con profundidad acotada cuando la cota de profundidad es 6 o mayor

Por tanto, a pesar de que en la gráfica 5.2 se observa que no es beneficiosa la implementación de las tablas de transposición para valores bajos de profundidad, podemos ignorarlos ya que no aportan demasiado.

Por último, el análisis que se obtiene de esta toma de tiempos es que para profundidades muy bajas, el coste temporal de recorrer la tabla de transposición pesa más que el propio análisis que hace el evaluador a cada situación del tablero que se encuentra, ya que son bastante reducidas. Esto mismo pasa para todas las profundidades en el evaluador aleatorio, debido a que elegir un número aleatorio para cada situación es más eficiente que recorrer la tabla de transposición en su totalidad.

Como se ha podido observar a lo largo de todas estas pruebas, la aplicación emplea una profundidad de 6. Debido al hardware empleado para realizar las pruebas (un ordenador de sobremesa común), queda limitada bastante la potencia del programa al tener que mantener unos costes temporales asequibles para el usuario. Además, en este trabajo el objetivo era desarrollar una inteligencia artificial que jugara al Othello de manera correcta, no perfecta, para darle opciones al usuario contra el que se enfrentará.

Sin embargo, se puede observar en la tabla 5.1 que a medida que avanza la profundidad, el jugador inteligente va siendo cada vez más y más preciso. De esta manera, si tuviésemos un ordenador capaz de ejecutar esta aplicación, reduciendo en gran medida los costes temporales de dichas profundidades, se podría obtener un jugador inteligente bastante superior en capacidades. Esto permitiría también que saliesen a relucir las ventajas temporales que aporta la implementación de las tablas de transposición, que como hemos visto en la sección **??**, con la profundidad empleada no son demasiado notables.

Capítulo 6

Conclusiones

En este trabajo se ha desarrollado una aplicación para poder jugar al juego de mesa conocido como Othello. Dicha aplicación ofrece la posibilidad de que el humano juegue contra la máquina. Para ello, se ha desarrollado un jugador no humano basado en técnicas de Inteligencia Artificial. En concreto, se ha explotado el algoritmo minimax para búsqueda en juegos de suma cero. Dado el tamaño del árbol de juego, el algoritmo minimax incorpora poda alfa-beta, profundidad acotada con función de evaluación heurística y tablas de transposición. Se ha evaluado experimentalmente y añadido distintas opciones para obtener un jugador no humano que juegue al Othello de manera correcta pero no perfecta, de forma que la experiencia de juego para el humano sea buena.

Por último, en caso de tener más tiempo este proyecto podría extenderse de tres formas posibles:

- Extender las posibilidades de la aplicación: Se podrían añadir menús y opciones a la interfaz gráfica que mejorase la experiencia del usuario.
- Diversificar al jugador inteligente: Se podrían crear tres dificultades (fácil, medio y difícil) modificando la función evaluadora que emplea el algoritmo minimax.
- Seguir explorando las tablas de transposición: Se podrían realizar pruebas en un ordenador más potente para poder realizar pruebas más precisas de los beneficios de las tablas de transposición para este caso concreto, obteniendo de esta forma más conclusiones al respecto.

Bibliografía

[1] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach, Global Edition*. Pearson, 4th edition, 2021.

[2] I. Millington. *AI for Games*. CRC Press, third edition, 2019.

[3] Georgios N Yannakakis and Julian Togelius. *Artificial intelligence and games*. Springer, 2018.

[4] Paul S Rosenbloom. A world-championship-level othello program. *Artificial Intelligence*, 19(3):279–320, 1982.

[5] Carlos N Silla, Marcelo Paglioney, and Iuri GP Mardegany. jothellot: A java-based open source othello framework for artificial intelligence undergraduate classes. In *2016 IEEE Frontiers in Education Conference (FIE)*, pages 1–7. IEEE, 2016.

[6] H Jaap Van Den Herik, Jos WHM Uiterwijk, and Jack Van Rijswijck. Games solved: Now and in the future. *Artificial Intelligence*, 134(1-2): 277–311, 2002.

[7] Arvind Vijayakumar. Developing an artificial intelligence bot for othello. In *2015 IEEE Integrated STEM Education Conference*, pages 216–220. IEEE, 2015.

[8] Roger S Pressman. *Software engineering: a practitioner's approach*. Palgrave macmillan, 2005.

[9] AL Sayeth Saabith, T Vinothraj, and M Fareez. Popular python libraries and their application domains. *International Journal of Advance Engineering and Research Development*, 7(11), 2020.

[10] Craig Larman and Victor R Basili. Iterative and incremental developments. a brief history. *Computer*, 36(6):47–56, 2003.

[11] Mark Richards. *Software architecture patterns*. O'Reilly Media, 2015.

[12] Python Software Foundation. The Python Tutorial. 9.6 classes, 2001-2024. URL https://docs.python.org/3/tutorial/classes.html.