View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Repositorio Institucional Universidad de Málaga

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA
Ingeniería del Software

Desarrollo del videojuego Reversi con Inteligencia Artificial

Developing Reversi videogame with Artificial Intelligence

Realizado por
Jesús de Sosa Cruz
Tutorizado por
Enrique Domínguez Merino
Departamento
Lenguajes y Ciencias de la Computación

UNIVERSIDAD DE MÁLAGA
MÁLAGA, Junio de 2017

Fecha defensa:
El Secretario del Tribunal

Resumen: El proyecto se basa en la implementación del videojuego Reversi con inteligencia artificial añadida. El entorno de desarrollo utilizado ha sido Unity 3D para la implementación del apartado visual y Microsoft Visual Studio para el desarrollo de la parte de programación en lenguaje C#.

En él se podrán realizar partidas de un jugador contra la máquina, la cual consta de cuatro modos de dificultad. El primero es el Novato, el cual jugará de manera aleatoria, el segundo es el Intermedio, el cual está desarrollado utilizando el algoritmo, el tercero es el modo Avanzado que cuenta con la implementación del algoritmo Minimax con poda Alfa Beta, para agilizar la búsqueda y poder buscar estados a mayor profundidad. Cabe destacar que tanto Minimax como la poda Alta-Beta están desarrollados con una variante de Minimax nombrada Negamax. Por último el modo Experto que cuenta con el algoritmo de búsqueda Negascout.

El juego también cuenta con la posibilidad de realizar partidas con otro jugador o simplemente ver como ambas inteligencias juegan entre ellas.

Palabras claves: Reversi, Othello, C#, Unity 3D, Inteligencia Artificial, Minimax, Poda Alfa-Beta, Negamax, Videojuego, Microsoft Visual Studio.

Abstract: The project is based on the implementation of the game Reversi with artificial intelligence. The development environment used has been Unity 3D for the visual implementation and Microsoft Visual Studio for the development of the programming part in C#.

In this project the player can play games against the machine, which has four different levels of difficulty. The first is Rookie, which it's going to play randomly, the second is the Intermediate, which is going to play using the algorithm Minimax, the third is Advanced mode that has the same algorithm Minimax but with Alpha-Beta pruning, to speed up the search and be able to look for states in greater depth. It should be noted that both Minimax and Alta-Beta are developed with a variant of Minimax named Negamax. The last one is Expert mode, wich has Negascout search algorithm.

The game also has the possibility to play with another player or simply see see how both intelligences play between them.

Keywords: Reversi, Othello, C#, Unity 3D, Artificial Intelligence, Minimax, Alfa-Beta Pruning, Negamax, Videogame, Microsoft Visual Studio.

Índice

1. Introducción....9

1.1 Motivaciones....9

1.2 Objetivos....9

1.3 Estructura de la memoria....10

1.4 Estado del arte....10

2. Estudio y análisis....13

2.1 Reglas del juego....13

2.2 Modelado del juego....14

3. Diseño del apartado visual....15

3.1 Menú principal....15

3.2 Menú de juego....17

3.3 Jugador humano....18

4. Implementación de la Inteligencia Artificial....21

4.1 Jugador Aleatorio....22

4.2 Jugador Intermedio....22

4.3 Jugador Avanzado....24

4.4 Jugador Experto....25

4.5 Heurístico....26

5. Integración de módulos....31

6. Pruebas y resultados....33

6.1 Pruebas....33

6.2 Resultados....34

7. Trabajos futuros....37

8. Conclusiones....39

Referencias bibliográficas....41

1. Introducción

Este proyecto se basa en la implementación del juego de mesa Reversi con Inteligencia Artificial con el entorno Unity3D[1] y el lenguaje C#, en los siguientes puntos se irá concretando y explicando los pasos seguidos y las soluciones encontradas, así como problemas durante el desarrollo.

1.1 Motivaciones

La motivación para la realización de este proyecto era el aprendizaje del entorno de desarrollo de videojuegos Unity3D, el cual mezcla vista y programación para la implementación de un videojuego. Así como el uso del nuevo control de versiones integrado el entorno, denominado Collaborate.

También el aprendizaje más a fondo del lenguaje de programación C# con el entorno de programación de Microsoft Visual Studio y la implementación de distintos algoritmos de decisión en juegos con adversario.

1.2 Objetivos

El objetivo principal es la implementación del juego de tablero Reversi, utilizando la plataforma Unity3D y el lenguaje C# en el entorno de Microsoft Visual Studio[2].

También uno de los objetivos principales es el contar con Inteligencia Artificial de distintos grados de dificultad así como el de realizar un pequeño estudio para comprobar la diferencia entre ellas y el porcentaje de partidas ganadas contra un jugador aleatorio y contra un jugador humano, así como comprobar si es mejor comenzar el primero o el segundo, o la diferencia que puede haber con el tiempo máximo asignado por turno de juego.

1.3 Estructura de la memoria

La memoria estará estructurada siguiendo el orden de desarrollo, habiendo sido desarrollado con una metodología incremental[3][4] y, por ello, siguiendo un orden preestablecido, comenzando por el análisis del juego, seguido del diseño del apartado visual y todos sus problemas encontrados, así como su resolución final.
Tras ello se detallará el desarrollo de las reglas del juego y el funcionamiento final para el turno de un jugador humano. Seguido de este desarrollo, entrará la implementación de la Inteligencia Artificial adaptándose al entorno de juego.
Para acabar la memoria, se realizarán comentarios sobre problemas encontrados tras la implementación de todos los módulos y la integración de los mismos.

1.4 Estado del arte

El estado del arte del juego de tablero Reversi es amplio pese a no ser un juego bastante común, como podría ser el ajedrez o las damas, pero eso no quita que sea un juego atractivo, diferente y que con él se puedan realizar estudios interesantes en diferentes ramas. Concretando exactamente a lo que este trabajo busca, bien es cierto que hay mucha documentación al respecto y, con ellos, muchos estudios realizados de diferentes fuentes, cada cual más englobado en un campo específico que en otro. Se destacarán los más impresionantes y, con ellos, lo que sirvió para entender aún más como abordar el proyecto, mejorarlo y poder añadir algo que hiciera este estudio un poco diferente.

Uno de los más interesantes que se encontró fue creado por Steve McGuire[5], el cual estudia bastante a fondo diversas estrategias del juego. Pese a no haber sido utilizado en este proyecto, era bastante interesante conocer de primera mano las mejores estrategias para comenzar una partida dependiendo del movimiento rival, para, en pocos turnos, conseguir un estado de tablero que nos genera una alta probabilidad de victoria. Aunque lo realmente destacable y útil personalmente en este proyecto fue el estudio de estrategias generales del juego. Con ellas se busca una forma de jugar específica que, siendo ejecutada correctamente, también nos lleva a una victoria.

Destaca en su estudio que las posiciones del tablero no son iguales, sino que cada una tiene una mayor o menor importancia para nosotros. Esto quiere decir que, por ejemplo, las esquinas son bastante clave para la victoria, pero no esenciales para la misma. Otra de ellas es la más primitiva y la que toda persona iniciada en juegos de tablero busca, y es la de tener el máximo número de fichas en el tablero en todo momento. Aunque esta última no siempre nos conducirá a la victoria como se muestra en las Figuras 1 y 2, donde, aparentemente van a ganar las blancas pero que finalmente acaba mal para las mismas.

Figura 1. Van ganando blancas

Figura 2. Ganan negras

Es por ello que no siempre tener más fichas equivale a una victoria asegurada. Por ello introduce un concepto bastante interesante y que si asegura conseguir una victoria cuantas más fichas tengamos, siempre y cuando cumplan esa regla.
Trata en buscar fichas estables, estas son fichas que será imposible que, en turnos futuros pasen a ser del rival, siendo como su propio nombre indican, estables.
Con esta estrategia se consigue que, aunque se vaya perdiendo, si se mantiene un número alto de fichas estables, se podrá obtener más fácilmente una victoria.

También se tiene en cuenta la estrategia de posicionamiento, que implica no todas las posiciones del tablero son iguales, sino que hay unas que son mejores que otras, como por ejemplo las esquinas, que son las que más estabilidad dan a lo largo de la partida, al ser unas posiciones que una vez obtenidas jamás serán perdidas.

En este estudio destaca otras estrategias como la de movilidad, evaporación… pero al ser muchas distintas decidí centrarme en la unión de las que más me parecieron útiles, ente ellas están las fichas estables, el posicionamiento, controlar las esquinas y el número de fichas.

Otro estudio observado es el de Michael J. Korman[6], el cual estudia cómo funciona el juego con diferentes métodos para añadir jugadores autónomos, como son el algoritmo Minimax, búsqueda iterativa y poda Alfa-Beta. Es por este estudio lo que me incitó a ir más allá y probar el añadido de Negascout.
El último estudio relacionado con este proyecto lo realizó Jack Chen, en el cual documenta aspectos teóricos sobre distintos algoritmos y sus tiempos de respuesta del juego Reversi así como diferentes formas de almacenamiento de la información.

Demuestra en él como el uso de la poda Alfa-Beta mejora considerablemente la velocidad frente al algoritmo Minimax sin poda. Así como que, dependiendo de la función de evaluación utilizada como la forma de búsqueda están muy presentes en la eficiencia de los tiempos de ejecución finales.

Bien es cierto que este juego y este campo está bastante estudiado y documentado de diferentes formas, tanto para buscar las mejores estrategias como para encontrar mayor eficiencia frente a otras técnicas de programación. En este caso era un reto utilizar un entorno diferente y visual como es Unity3D, un lenguaje de programación como C# e intentar juntar formas de pensar, así como aportar un nuevo punto de vista para conseguir integrar todas esas ideas en un proyecto diseñado en un entorno diferente.

2. Estudio y análisis

Para comenzar el desarrollo de este proyecto, hacía falta un estudio previo y un buen análisis de los requisitos a tener en cuenta una vez avanzado el proyecto. Es por ello que en primer lugar había que conocer el juego y sus reglas.

2.1 Reglas del juego

Reversi[7] es un juego de tablero en el cual dos jugadores en un tablero, comúnmente de 8 filas y 8 columnas, van colocando sus fichas, las cuales pueden ser blancas o negras, para conseguir tener un mayor número de piezas que el rival. Partiendo de esto, se tiene claro la posibilidad de añadir un jugador autónomo con el cual poder realizar partidas, aunque aún era necesario más para conocer el funcionamiento del mismo.

Las reglas son sencillas, los jugadores tienen un tablero inicial y podrán colocar fichas en posiciones donde consigan flanquear a fichas rivales. Una vez realizado el movimiento, estas fichas flanqueadas pasarán a ser del color del jugador rival. Teniendo esto en cuenta, denotamos un turno de juego a la necesidad de tener al menos una casilla del tablero donde haya mínimo una ficha enemiga flanqueada. En caso de no cumplir esto, pasaría al turno rival. En caso de que ambos sufran un cambio de turno, se habrá llegado a un estado final del juego. Otro estado final resultaría la colocación de las 64 fichas completando el tablero por completo, estando la posibilidad siempre de empate al estar formado por un número de piezas par.

Una vez analizadas las reglas del juego y su funcionamiento, llegamos a la necesidad de estudiar cómo desarrollarlo de la mejor manera y la más clara posible.

2.2 Modelado del juego

El principal problema era adaptar de forma simple varios jugadores autónomos sin necesidad de controlar en la vista qué tipo de jugador está jugando o a que algoritmo llamar. Por ello el uso de una interfaz para estos jugadores era más que necesaria, ya que si llamamos a un método común e internamente realizan diferentes funciones ya obteníamos lo que necesitábamos.

Es lo que muestra la Figura 3, un Diagrama de Clases en el cual tenemos una Interfaz IJugador, de las cuales heredarán nuestros distintos jugadores autónomos, los cuales son JugadorAleatorio, JugadorMinimax, JugadorAlfaBeta y JugadorNegascout.

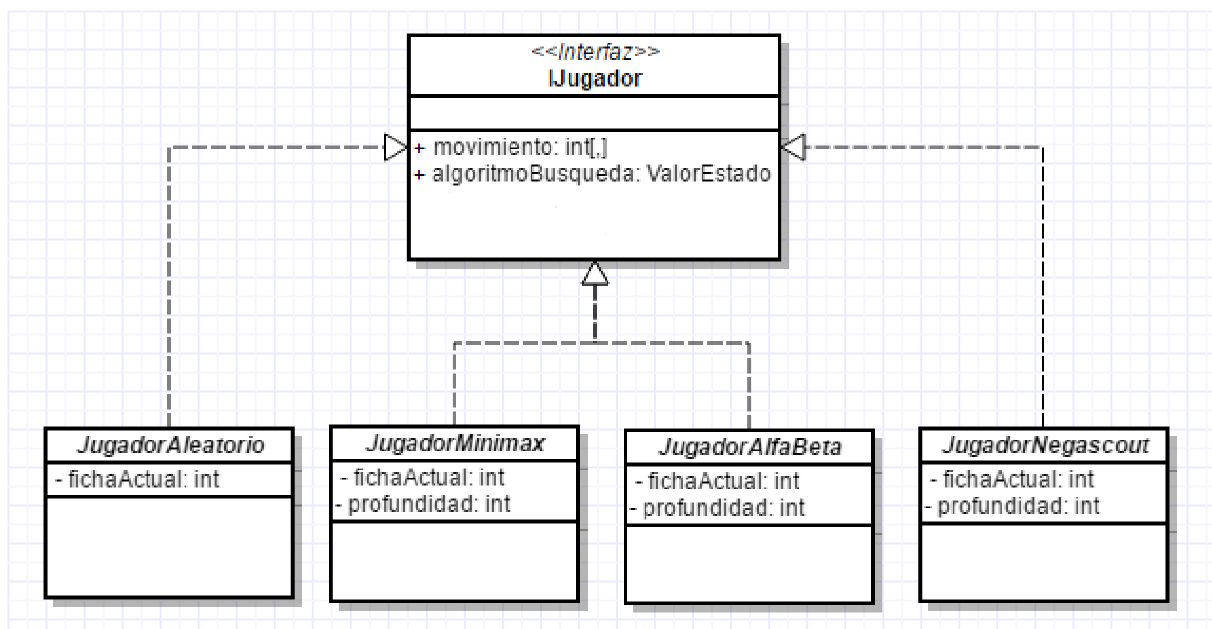

Figura 3. Diagrama de Clases de los jugadores autónomos

La interfaz IJugador contaría con dos métodos necesarios, uno de movimiento, el cual se llamaría desde la vista y que serviría para realizar el turno de un jugador autónomo y el método algoritmoBusqueda, el cual llevará implementado el algoritmo en cuestión, como pueda ser Minimax, Negascout, entre otros. Ambos métodos los veremos más a fondo más adelante.

3. Diseño del apartado visual

Para el diseño del apartado visual del juego se ha utilizado Unity3D, un motor de desarrollo que es utilizado principalmente para el desarrollo de videojuegos. Este motor es el más utilizado por aquellas personas que deseen comenzar en el mundo de los videojuegos, ya que su aprendizaje no es más sencillo que otros entornos.

Para el desarrollo del Reversi se necesitó la implementación de dos menús diferentes. El primero la pantalla principal, la cual contaría con distintos modos de juego y la segunda la ventana de juego, donde aparecerá el tablero.

3.1 Menú principal

En el menú principal del juego aparecerán las diversas opciones de juego habilitadas para el mismo. Existen tres posibilidades diferentes, poder observar distintas formas de Inteligencia Artificial jugar entre ellas, poder batirnos contra una nosotros mismos o simplemente jugar con algún compañero en el modo dos jugadores.

Figura 4. Pantalla de selección de modo de juego

En el modo IA VS IA, tendremos la posibilidad de elegir el nivel de dificultad de ambos jugadores, teniendo en cuenta que el primer jugador será el del color blanco y el segundo el de color negro.

Figura 5. Selección de nivel de dificultad de los jugadores autónomos

En el menú de un jugador, en el lado izquierdo, aparecerá solamente a elegir el color de la ficha y, a su derecha, la opción de elegir al igual que el modo anterior, un nivel de dificultad para su rival. El último modo solo nos dará a intercambiar los colores de las fichas de cada jugador, pulsando sobre el color de la ficha en cuestión.

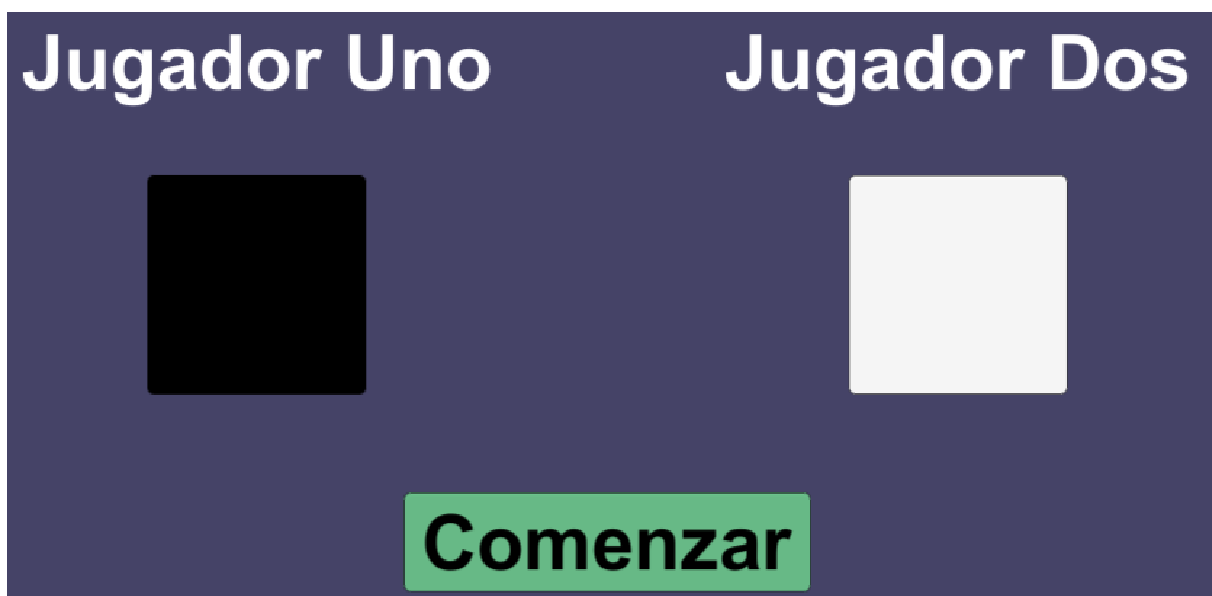

Figura 6. Selección del color de ficha de los jugadores humanos

3.2 Menú de juego

En el menú de juego se observará el tablero con las fichas posicionadas en el estado inicial básico de Reversi, cuatro fichas, dos blancas y dos negras en el centro formando una cruz. La interfaz constará de un indicativo de la ficha que toca mover en el turno actual, así como la opción de poder abandonar la partida.

En caso de perder turno, aparecerá temporalmente un aviso indicando que no había movimientos posibles y que se efectuó un cambio de turno. En caso de llegar a un estado final, indicará quién es el ganador o si ha resultado en empate y el número de fichas de cada color.

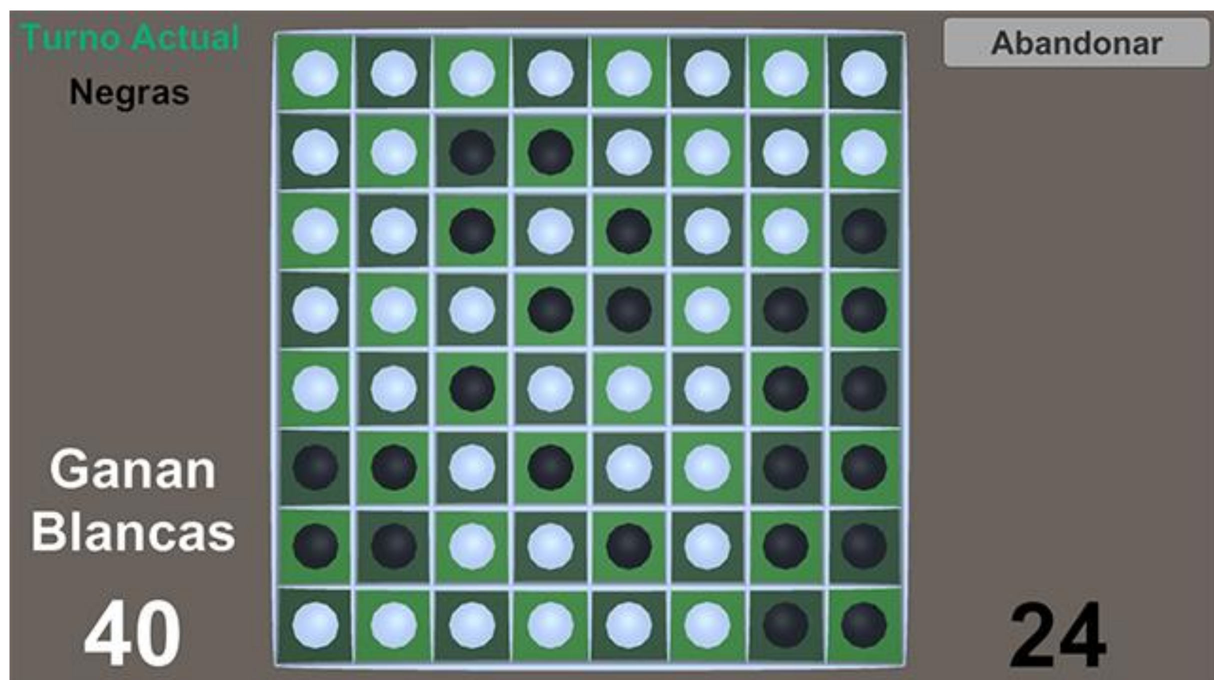

Figura 7. Interfaz menú de juego

Cada casilla del tablero es un objeto de Unity3D llamado GameObject, el cual nos servirá para detectar la colisión con el ratón en el turno del jugador humano. Las fichas situadas en él también son objetos, y en ellas se asigna un Script donde se almacena su color actual, su estado y un color temporal. Este color será utilizado para indicar al jugador qué fichas rivales cambiarán respecto a la posición actual del ratón.

3.3 Jugador humano

El jugador humano, a diferencia de los jugadores autónomos, es un jugador con una implementación diferente, el cual se nutre de los elementos de la vista y requiere del uso de la misma. Es por ello que, para realizar un movimiento en un turno donde el jugador humano esté presente, requerirá de una serie de requisitos.

Para la realización del movimiento, era necesaria la detección del ratón en una casilla, por ello se hizo uso de una funcionalidad que da Unity para estos casos, los Ray. Un Ray no es más que un rayo en línea recta. En este caso, el rayo es creado desde la posición central de la cámara hasta la posición del ratón, dejando así un rayo hacia el fondo del campo de juego, lo que conseguirá que se detecte la posición del ratón en los objetos que se encuentran debajo del mismo. Esto se consigue gracias a la función Physics.Raycast, la cual devuelve true en caso de detectar colisión con algún objeto del juego y que nos devolverá por su segundo parámetro mediante el uso de out el objeto colisionado.

```
RaycastHit hit;
Ray ray = Camera.main.ScreenPointToRay(Input.mousePosition);

if (Physics.Raycast(ray, out hit))
```

Figura 8. Inicialización del rayo

En C# hay varias formas de conseguir, mediante los parámetros de una función, rescatar distintos valores que serán cambiados dentro de ellas. Es el caso de 'out' y también de 'ref'. Out, el usado en este caso, indica que el objeto deberá ser inicializado dentro de la clase. En caso contrario, ref indica que el objeto debe ser inicializado antes de uasrlo como parámetro. Lo positivo del uso de herramientas como estas nos agilizan el desarrollo y nos evitan el uso de variables globales que solo se utilicen en pequeñas partes de código o solo en un método de toda la clase.

En caso de detectar colisión con una posición válida de tablero, es decir, que no haya fichas colocadas, se realizará a tiempo real la búsqueda de las fichas que cambiarían en caso de que la ficha fuera colocada ahí. Para mostrarlo visualmente, estas fichas serán coloreadas temporalmente en rojo como se puede observar en la Figura 9.

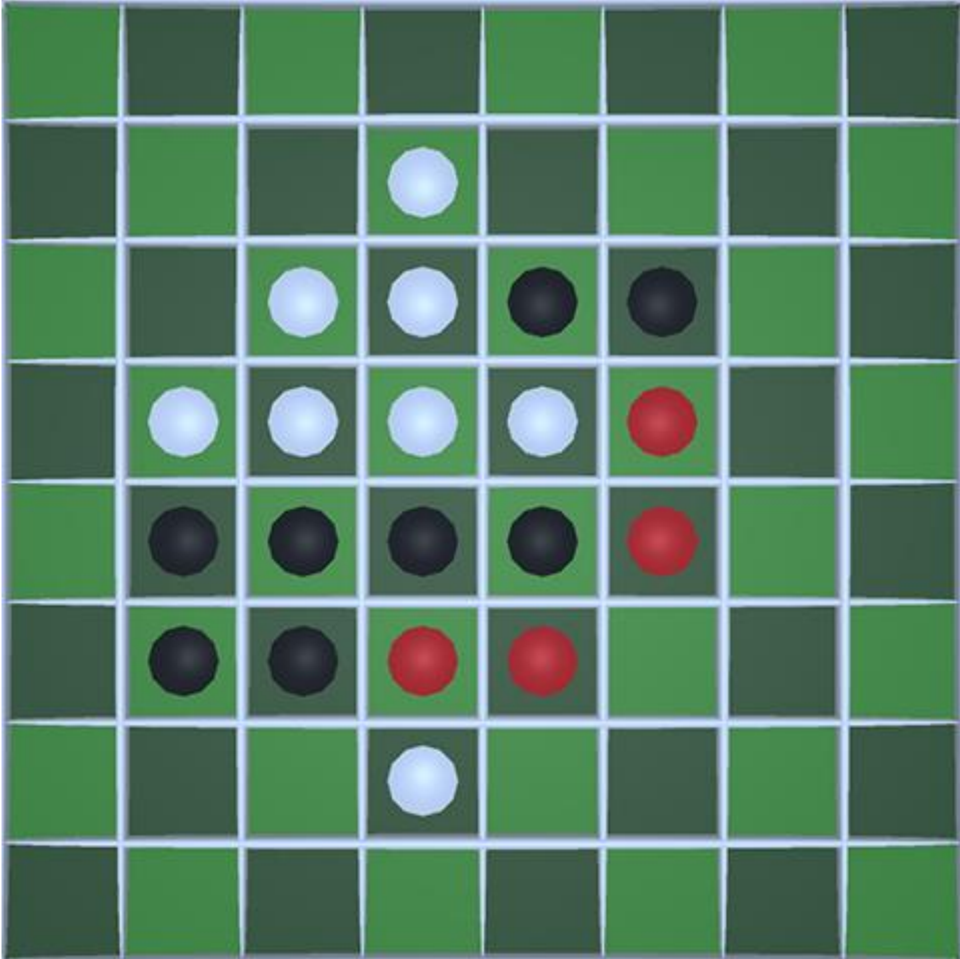

Figura 9. Cambio de fichas temporal

Si se detecta una pulsación con el ratón, automáticamente esas fichas cambiarán de color, sin necesitar realizar una búsqueda, ya que se tienen almacenadas las fichas rojas en todo momento.

4. Implementación de la Inteligencia Artificial

Como se ha venido comentando en anteriores capítulos, en este desarrollo se ha procedido a la implementación de cuatro jugadores autónomos, cada cual con su algoritmo de búsqueda[8] definido.

Cada jugador consta de dos métodos diferentes, el de movimiento y el del algoritmo de búsqueda. La decisión de subdividirlo en dos viene por la necesidad de tener una función común que pudiera ser utilizada en los turnos del juego, independientemente del tipo de jugador autónomo que esté en cada turno. El método movimiento recibirá siempre el valor Reversi, una clase que nos servirá para, recibiendo un color de ficha y un estado del tablero, encontrar los distintos hijos posibles del mismo.

A su vez, el método algoritmoBusqueda recibirá diferentes tipos de parámetros, ya que, en caso de no recibir alguno, se les asigna un valor por defecto. En este caso recibirá el valor Reversi, la ficha actual, el límite de profundidad, el signo al que se debe evaluar el heurístico y valores alfa y beta.

```
int[,] movimiento(Reversi estado);
ValorEstado algoritmoBusqueda(Reversi estado, int turno = 0, int profundidad = 0,
                              int signo = 0, int alfa = 0, int beta = 0);
```

Figura 10. Métodos jugadores autónomos

La clase ValorEstado, la cual devuelve el algoritmo de búsqueda, es un objeto que contiene el valor heurístico obtenido y el estado del tablero asignado al mismo. Esto implica que una vez terminado el algoritmo de búsqueda, devolverá el mejor estado de tablero tras realizar un movimiento y su valor heurístico asignado. Ahora, una vez conocido cómo funcionará cada jugador por dentro, se procederá a entrar en más detalle en cada uno de ellos.

4.1 Jugador Aleatorio

El primer jugador autónomo, el Novato, es el más simple de todos. Como cualquier persona que apenas conoce un juego ni las estrategias, acaba jugando aleatoriamente, y es así como este jugador lo hace, por ello es denominado JugadorAleatorio. Entre todos los posibles futuros estados del tablero, elegirá uno al azar, independientemente de si es o no el mejor movimiento posible en su turno actual.

El resto de jugadores autónomos si siguen un algoritmo que se centra en encontrar el mejor movimiento actual buscando en futuros estados el mejor desenlace para ellos.

4.2 Jugador Intermedio

El jugador intermedio cuenta con el algoritmo Minimax, un algoritmo de decisión que busca minimizar la máxima pérdida. Eso implica que se buscará un estado de juego donde el jugador rival no consiga su mayor estado posible pero que siempre se encuentre en uno peor o igual al de jugador autónomo.

Este algoritmo implica que, para el turno del jugador autónomo podremos nombrarlo como MAX, ya que busca el mejor estado, y al rival como MIN, ya que tendremos que asegurarnos de encontrar el peor estado posible para el rival, eso implica minimizar su mejor movimiento. Para una implementación más limpia, decidí elegir una variante de Minimax denominada Negamax, que consiste, en vez de comprobar en que jugador estoy y realizar una búsqueda para maximizar u otra para minimizar, se realiza una sola búsqueda, donde se cumple la condición $\max(a, b) = -\min(-a, -b)$.

Al jugador Minimax se le asocia un color de una ficha y una profundidad máxima hasta la que realizará el algoritmo de búsqueda. Teniendo estos valores, podemos comenzar a explicar el funcionamiento del mismo en el juego Reversi. En este caso decidí usar la profundidad tres, equivalente a ver el estado de los siguientes dos turnos del jugador, ya que utilizar una profundidad mayor aumentaba el tiempo de espera.

Si superamos tiempo de espera por turno, si la profundidad actual es cero o el estado de juego es final, tomaremos el valor del heurístico del estado en cuestión. En el caso de que no podamos mover, se realizará una búsqueda para la ficha del rival, el cual si tendrá movimientos al no haberse detectado como un estado final del juego.

```
if (Evaluador.temporizador.ElapsedMilliseconds > 2000 ||
    profundidad == 0 || Evaluador.esFinal(estado.getTablero()))
{
    mejorValor = signo * Evaluador.heuristico(estado.getTablero(), fichaActual);
    mejorEstado = estado.getTablero();
}
else if(!Evaluador.hayMovimientos(estado.getTablero(), fichaActual))
{
    mejorValor = -algoritmoBusqueda(estado, -fichaActual, profundidad, -signo).getValor();
    mejorEstado = estado.getTablero();
}
```

Figura 11. Finalización de búsqueda y cambio de turno en jugadores autónomos

Si nada de esto ocurre realizaremos la búsqueda en los diferentes hijos que se obtienen, es decir, los distintos posibles estados dependiendo de dónde se coloque la ficha en el tablero. Por ello, y en el caso de que si haya posibles movimientos, el algoritmo Minimax buscará siempre el mejor caso de todos, observando todas y cada una de las posibilidades del árbol obtenido.

```
foreach (Reversi reversi in hijos)
{

    valorAux = -algoritmoBusqueda(reversi, -fichaActual, profundidad - 1, -signo).getValor();

    if((valorAux > mejorValor) || (mejorEstado == null))
    {

        mejorValor = valorAux;
        mejorEstado = reversi.getTablero();

    }

}
```

Figura 12. Algoritmo Minimax con variante Negamax

Si el número de hijos crece muy rápidamente será poco eficiente la búsqueda del mejor estado usando mucha profundidad, y como en Reversi es muy frecuente, se requieren de algoritmos que mejoren la velocidad de Minimax.

4.3 Jugador Avanzado

Le sigue el jugador avanzado, el cual cuenta con poda Alfa-Beta, una mejora añadida al algoritmo Minimax. Esto da como resultado una mayor eficiencia, ya que éste decrementa el número de nodos a visitar podando los que, cumpliendo una condición, no es necesario que visite. Cabe destacar que el hecho de que esta poda elimine nodos y no los visite, no desencadenará en un resultado diferente al de Minimax.

La mejora de nivel destaca en que, al ser un algoritmo muchísimo más rápido que Minimax, es posible realizar una búsqueda a mayor profundidad, siendo para este caso una profundidad de cinco. Aparte de esta diferencia, requerirá de dos variables más, alfa y beta. Alfa será utilizado para maximizar para el jugador autónomo el estado resultante. Para Beta ocurre lo contrario, será el valor a minimizar en cualquier caso.

El funcionamiento en casos finales de juego y cambios de turno es idéntico al algoritmo Minimax, pero donde si hay un cambio significativo es cuando tenemos que buscar el mejor estado de entre todos los posibles. Como se está utilizando la variante Negamax, siempre se realizará una búsqueda que maximice el estado, ya que se realizará un control con los valores alfa y beta para que el algoritmo acabe maximizando todos los valores encontrados.

Para la elección del mejor estado tendremos que obtener el valor de uno de los hijos. Si este valor es superior al valor de beta realizaremos una poda, devolviendo éste como mejor estado y obviando el resto. En caso contrario, comprobaremos si es superior a alfa, en caso afirmativo, alfa se convertirá en el valor obtenido.

```
if(valorAux >= beta)
{
    return new ValorEstado(mejorValor, mejorEstado);
}

if(valorAux > alfa)
{
    alfa = valorAux;
}
```

Figura 13. Poda Alfa-Beta

4.4 Jugador Experto

Por último, el jugador Experto, cuenta con la implementación del algoritmo Negascout[9], un algoritmo que siguiendo el estilo de implementación de la poda Alfa-Beta en la variante Negamax, consigue ser aún más eficiente con el uso de la ventana nula. Su funcionamiento es el siguiente, este algoritmo realiza una búsqueda completa del primer hijo con una ventana abierta, eso implica que no hay ningún tipo de restricción, pero, tras esta búsqueda, el resto se realizará bajo el uso de una ventana nula.

La ventana nula es utilizada para realizar la poda masiva en el algoritmo Negascout. En ella se asignan los valores [alfa, alfa+1], de manera que, independientemente del valor heurístico obtenido, siempre fallará inferior o superiormente. En caso de ser inferior se continuará de forma normal con la búsqueda, pero si es superiormente y acabamos obteniendo un valor alfa mayor o igual que el valor de beta, se realizará una poda y obtendremos como nuevo mejor resultado el nuevo valor de alfa.

En este algoritmo se requiere que los nodos estén ordenados, ya que toma el primer hijo como si fuera el mejor estado. Si éste cumple los requisitos no explora nodos que el algoritmo Alfa-Beta podaría. Lo positivo, pese que no hay forma de ordenar correctamente los hijos en este juego, es que en el peor de los casos este algoritmo acabará actuando y realizando el mismo trabajo que el que haría el algoritmo Alfa-Beta. Este último caso se dará cuando la búsqueda en la ventana nula no se cumpla.

```
valorAux = -algoritmoBusqueda(estado, -fichaActual, profundidad, -signo, -beta, -alfa).getValor();

if ((valorAux > alfa) && (valorAux < beta))
{
    valorAux = -algoritmoBusqueda(estado, -fichaActual, profundidad, -signo, -beta, -valorAux).getValor();
}
```

Figura 14. Doble búsqueda en Negascout en caso de no cumplir la ventana

La mejora de este algoritmo en tiempo de ejecución es notable, ya que ha sido posible el aumento de profundidad a nueve sin empeorar demasiado el rendimiento de juego. Pese a esta mejora, siempre hay estados que ramifiquen en muchos nodos, requiriendo un límite de tiempo para evitar largas esperas en uno de los turnos del rival.

4.5 Heurístico

Para la implementación del heurístico se tuvieron en cuenta varias estrategias diferentes, las cuales fueron mencionadas en anteriores capítulos. Para recordarlas, se utilizaron el número de fichas de cada jugador, las estables, el posicionamiento de cada una de ellas y el control de esquinas.

Hay unos casos especiales que se pueden llegar a dar y que, si se cumplen, estas estrategias no se tienen que tener en cuenta. Y estos casos son los denominados estados finales, los cuales llevan a un estado en el que el juego ha acabado. Es el caso de la pérdida total de fichas de un jugador o un estado final de la partida donde no haya posibilidad para ninguno de los dos jugadores para colocar fichas o ambos hayan completado el tablero.

```
int h = 0;

if(fichasEnemigo == 0)
{
    h = int.MaxValue;
}
else if(fichasJugador == 0)
{
    h = int.MinValue;
}
else if (esFinal(tablero))
{

    if(fichasJugador != fichasEnemigo)
    {
        h = fichasJugador > fichasEnemigo ? int.MaxValue : int.MinValue;
    }

}
```

Figura 15. Heurístico de estados finales de juego

Se puede observar que, en caso de empate, no se tomará el estado como negativo ni positivo, sino que mantendrá el valor cero. En caso de que ninguno de estos casos ocurra, tendremos que realizar un cálculo heurístico para determinar el valor del estado para el jugador.

Para el cálculo del heurístico, se llevará una cuenta del número de fichas que tiene cada jugador, así como el número de fichas estables, el valor de la posición de cada ficha en el juego y el número de esquinas controladas por cada jugador. Teniendo en cuenta que cada uno de estos casos no tienen el mismo peso, es decir, que tener más fichas que el otro hemos comprobado antes que no es del todo seguro, por ello le asigné un peso a cada uno de ellos, para así tener más en cuenta factores más importantes a la hora de elegir un estado u otro. En primer lugar y, dependiendo del número de fichas de cada jugador, asigno un valor positivo o negativo con la proporción de fichas del jugador que en ese momento tenga más fichas.

```
if (fichasJugador > fichasEnemigo)
{
    h += 20 * (fichasJugador / (fichasJugador + fichasEnemigo));
}
else if (fichasJugador < fichasEnemigo)
{
    h += -20 * (fichasEnemigo / (fichasJugador + fichasEnemigo));
}
```

Figura 16. Heurístico del número de fichas por jugador

El número de esquinas que cada jugador tiene es muy importante, ya que no solo son posiciones que nos aseguran fichas estables, sino que también nos aseguran la creación de más de este tipo de fichas ya que las esquinas son las posiciones perfectas para flanquear las fichas enemigas. Es por ello que, y teniendo en cuenta que como máximo podremos tener cuatro esquinas, se le asigna una proporción bastante alta.

```
h += 500 * (numEsquinasJugador - numEsquinasEnemigo);
```

Figura 17. Heurístico de esquinas controladas por jugador

En este caso si es importante tener en cuenta que el jugador enemigo no consiga las esquinas, por ello también devolvería un valor bastante bajo en ese caso.

El siguiente a tener en cuenta ha sido el posicionamiento de las fichas. En el tablero de Reversi cada posición tiene una importancia mayor que otras, tanto para bien como para mal, y es por eso que, teniendo en cuenta este factor, se puede tener un control de las mejores y peores posiciones en las que se puede tener una ficha. Aquí muestro el factor de posicionamiento utilizado para este heurístico.

25	-5	15	10	10	15	-5	25
-5	-10	-4	2	2	-4	-10	-5
15	-4	3	4	4	3	-4	15
10	2	4	0	0	4	2	10
10	2	4	0	0	4	2	10
15	-4	3	4	4	3	-4	15
-5	-10	-4	2	2	-4	-10	-5
25	-5	15	10	10	15	-5	25

Figura 18. Matriz de valores de posiciones en el tablero

Se observa que las esquinas, también en este cálculo, son tomadas como el mejor valor posible. Las adyacentes a las esquinas son las peores posiciones, ya que el hecho de colocar una ficha ahí da acceso al rival a hacerse con las tan preciadas esquinas. También tienen un gran valor los laterales, ya que darán un más fácil acceso a las esquinas.

Aquí se muestra que el peso de este valor es bastante importante, ya que tener un buen posicionamiento en el tablero nos puede llevar a futuro a un estado más satisfactorio a tener un estado inicial muy malo.

```
if(valorPosicionesSuma != 0)
{

    if (valorPosicionesJugador > valorPosicionesEnemigo)
    {
        h += 75 * (valorPosicionesJugador / valorPosicionesSuma);
    }
    else if (valorPosicionesJugador < valorPosicionesEnemigo)
    {
        h += -75 * (valorPosicionesEnemigo / valorPosicionesSuma);
    }

}
```

Figura 19. Heurístico del valor de las posiciones por jugador

Por último se tiene en cuenta el número de las fichas estables. Estas fichas eran aquellas que no se podrán perder una vez se consiguen estabilizar en lo que queda de partida, ya que serán infranqueables. Pese a ser algo positivo, no se le asignó un peso muy grande, aunque si más que el hecho de tener o no más fichas no estables.

```
if (valorNumEstablesSuma != 0)
{

    if (numEstablesJugador > numEstablesEnemigo)
    {
        h += 50 * (numEstablesJugador) / (valorNumEstablesSuma);
    }
    else if (numEstablesJugador < numEstablesEnemigo)
    {
        h += -50 * (numEstablesEnemigo) / (valorNumEstablesSuma);
    }

}
```

Figura 20. Heurístico del número de fichas estables por jugador

5. Integración de módulos

Una vez implementado el módulo visual junto al jugador humano y haber completado el desarrollo de los jugadores autónomos, llegaba el momento de integrar ambas partes para completar el desarrollo del juego Reversi.

Lo primero era la unión de los datos de la vista principal y la vista del juego. La manera más fácil para pasar valores de una vista a otra es mediante el uso de clases estáticas, en este caso se utilizó la clase GameOptions, la cual almacena qué tipo de jugador es cada uno y el color de sus fichas. Estos valores son rescatados en la vista del juego principal y ahí es donde se preparan los jugadores que, gracias al uso de la interfaz IJugador era más sencillo de realizar.

```
fichaJugadorUno = GameOptions.getFichaJugadorUno();

switch (GameOptions.getJugadorUno())
{
    case 0:
        esHumanoTurnoUno = true;
        break;
    case 1:
        jugadorUnoIA = new JugadorAleatorio();
        break;
    case 2:
        jugadorUnoIA = new JugadorMinimax(fichaJugadorUno, 3);
        break;
    case 3:
        jugadorUnoIA = new JugadorAlfaBeta(fichaJugadorUno, 5);
        break;
    case 4:
        jugadorUnoIA = new JugadorNegascout(fichaJugadorUno, 9);
        break;
}
```

Figura 21. Inicialización de los jugadores

Se observa que tanto la ficha del jugador como el tipo son asignados mediante valores recogidos de la clase GameOptions. También es en la inicialización del tablero donde son asignadas las profundidades de los jugadores autónomos.

También era necesaria la integración de distintas formas de ver el juego. Mientras que en el juego se almacena un array bidimensional de objetos ficha, los jugadores autónomos hacen uso del tablero en forma de array bidimensional de enteros. Esta forma de almacenamiento es usada para agilizar el funcionamiento tanto en la vista, en la cual es más ágil cambiar los colores temporales, ya que directamente son los objetos y no hay que buscarlos, y en los algoritmos de búsqueda de los jugadores autónomos, ya que solo tienen en cuenta valores enteros y hacen un uso menor de memoria que lo que haría un array de objetos.

De esta forma, cuando un jugador autónomo realiza un movimiento, este pasa a la vista y se le deben asignar los nuevos valores enteros a las fichas, cambiando el color al valor entero del mismo, eso significa que, si en la posición (7, 7) la ficha era neutra pero en el array de enteros hay un 1, la ficha pasará a ser blanca.

Curiosamente en la integración de ambos módulos no hubo problemas, ya que, desde el inicio del desarrollo estaban las ideas claras del desarrollo de la Inteligencia Artificial y, cuando se comenzó el desarrollo de la vista, el cambio de turnos ya seguía una base utilizable, ya que se podían realizar partidas de 2 jugadores humanos y el añadido de jugadores autónomos no fue más que adaptar si el jugador era o no era humano.

6. Pruebas y resultados

Una vez integradas todas las partes, se obtuvo un desarrollo completo del proyecto deseado, pero en todo trabajo software siempre pueden haber pequeños detalles a pulir, como pueden ser errores no controlados o detalles no tenidos en cuenta. Por ello, se realizaron sendas pruebas y se detectaron pequeños errores solventados, pero que en el desarrollo inicial no se tuvieron en cuenta. También se dedicará una sección para observar estadísticamente los resultados conseguidos tras sendas partidas ente los distintos niveles de dificultad de los jugadores autónomos y se detallará la diferencia que tendría éste desarrollo en equipos con características diferentes.

6.1 Pruebas

Las pruebas que dieron problemas fueron con el jugador humano, ya que hace uso de la vista y sus acciones son a tiempo real. Es por ello que se realizaron varias partidas entre dos jugadores y se intentaban realizar estados completamente imposibles.

Un gran error se escondía tras un pequeño detalle no controlado. Anteriormente se comentó que, si el ratón era situado en una casilla y ésta era un movimiento posible, todas las fichas a cambiar eran coloreadas de color rojo temporalmente. En el cambio de casillas estaba controlado que, si colocábamos el ratón en una ficha donde el movimiento no era posible, el color rojo desaparecía. Pero internamente no se había realizado la limpieza de éstas fichas, es decir, que pese a no ser rojas, aún se creía internamente que esas fichas podían cambiarse. Es por ello que si se pulsaba en una casilla aleatoria, independientemente de si esa era o no una casilla correcta, se colocaba la ficha y las fichas anteriormente rojas pasaban a ser del color del jugador, ya que la lista de fichas temporales aún tenía datos.

Tras realizar las pruebas se detectó un pequeño error que no se tuvo en cuenta en todo el desarrollo y que, pese a no manchar la funcionalidad del juego es bastante necesario. ¿Cómo salir del juego si no había ningún botón para ello? Efectivamente, en caso de estar en pantalla completa había que salir utilizando métodos bastante comunes para cerrar aplicaciones de una forma poco normal. Tras detectar ese error, se desarrolló la opción sin problemas con el resto.

6.2 Resultados

En este apartado se realizará un pequeño estudio sobre el índice de victorias de cada jugador autónomo. En la tabla 1 observaremos el número de partidas ganadas y la puntuación de media en las partidas, así como comprobar la diferencia de comenzar en el primer turno o hacerlo en el segundo.

Primer Turno VS Segundo Turno	**Partidas ganadas por el primer jugador**	**Partidas ganadas por el segundo jugador**	**Media de resultados**
Negascout vs Minimax	9	1	45 - 19
Negascout vs Alfa-Beta	8	2	40 - 24
Alfa-Beta vs Minimax	8	2	39 – 25
Alfa-Beta vs Negascout	3	7	28 - 36
Minimax vs Alfa-Beta	1	9	23 – 41
Minimax vs Negascout	0	10	16 – 48
Minimax vs Minimax	4	6	31 – 33
Alfa-Beta vs Alfa-Beta	8	2	34 – 30
Negascout vs Negascout	8	2	33 - 31

Tabla 1. Media de puntos

Se puede analizar de los datos obtenidos que los algoritmos que usan una mayor profundidad resultan ganadores frente al resto, eso implica que el heurístico utilizado busca mejorar el estado frente al resto y es por ello por lo que probablemente jugadores de similar profundidad y algoritmo de búsqueda tengan un resultado final muy parecido, pese a que al final la victoria se decante por uno de los dos.

También, si se observan las partidas entre algoritmos similares podemos llegar a afirmar que es indiferente en qué turno empezar para ser o no ganador ya que, pese a que en las partidas de Alfa-Beta y Negascout la diferencia de victorias sea grande, la puntuación media obtenida es más cercana al empate.

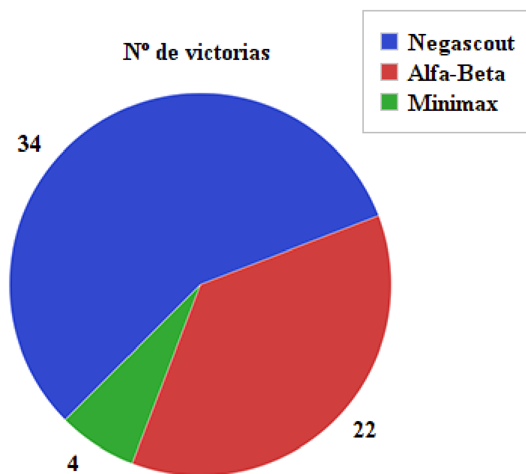

Figura 22. Número de victorias

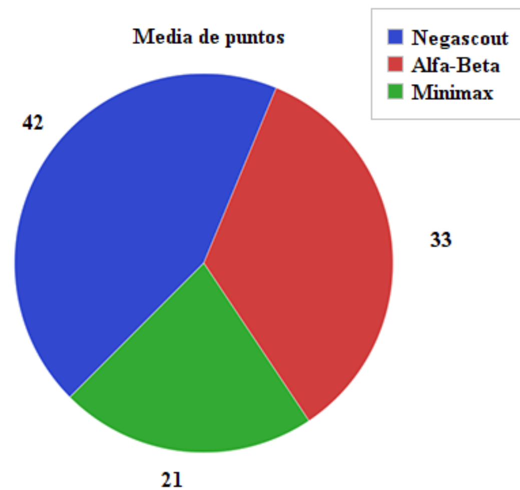

Figura 23. Media de puntos

7. Trabajos futuros

En este apartado me gustaría nombrar aquellas funcionalidades que me habrían gustado llevar a cabo en el proyecto pero que, por requerir mucho más tiempo de desarrollos y pruebas del que contaba, no se han podido llevar a cabo.

Comenzaré con la funcionalidad que fue desarrollada, se trataba de una funcionalidad que evitaría la búsqueda por ramas cuyos estados se habían calculado con antelación. Esta funcionalidad podría haber disminuido bastante los tiempos de ejecución, pero, pese sabiendo que en el primer turno de juego de Reversi nos encontramos en un estado que, coloquemos donde coloquemos todos los siguientes estados son idénticos girando o transponiendo el tablero, no dio ningún tipo de mejora, ya que cuando se realizaba el cálculo en turnos avanzados, el hecho de detectar estados repetidos era poco común o casi imposible.

La forma para detectar si un estado era similar a otro era mediante la utilización de una función que convertía el tablero en 4 números de 32 bits (o dos de 64 bits). Esto se conseguía de la siguiente manera. Tomemos como ejemplo los números de 32 bits. Si cogemos dos filas, tendremos 16 valores, los cuales, bit a bit, podremos rellenar con 1's si hay una ficha blanca o 0's si no la hay. Tras completar los primeros 16, rellenamos los 16 restantes siguiendo la misma mentalidad pero con las fichas negras. Con este método, se conseguiría que los estados del tablero tuvieran un valor asignado único y que no se podría repetir.

Se decidió almacenar de dos maneras diferentes, con la utilización de un diccionario hash enlazado, pero decrementaba la eficiencia considerablemente. También se ideó la implementación con un árbol AVL, un árbol que se equilibra automáticamente cuando una de sus ramas es más larga de la cuenta. Pero pese a que mejoraba considerablemente respecto al diccionario, seguía empeorando bastante el desarrollo normal del juego. Es por ello que este método se decidió abandonar y eliminar del producto final.

Ahora se detallarán funcionalidades que habrían resultado interesantes de llegar a implementar pero que, por falta de tiempo en el proyecto, se dejaron apartadas.

Una de las funcionalidades era el añadido de las Claves Zobrist[10] (Zobrist Keys), el cual es utilizado para el cálculo de los valores hash de cada estado del tablero. Para ello se deberá, como primer paso, realizar un número de claves concretas. En el caso del Reversi, donde hay 64 casillas y 2 colores diferentes, serían necesarias 2 x 64 = 128 claves diferentes. Para ello se puede generar un array de tamaño 128, donde cada entrada es un valor aleatorio.

Una vez tenemos estas claves, y asignando las 64 primeras casillas a las blancas y las 64 siguientes a las negras, podemos crear un valor hash del estado actual del tablero. Para ello se realizará la operación XOR de los valores asignados dependiendo de la posición del tablero y el color de la ficha en cuestión. Por ejemplo, si en la casilla (0,0) hay una ficha blanca y en los valores la posición 0 tenemos un valor aleatorio, con él se realizará la función XOR con el valor de otra casilla, por ejemplo la (0,1), que suponiendo que hubiera una ficha negra, podríamos coger el valor de la posición 65, la cual sería equivalente a la 1 pero para las negras.

Bien es cierto que con este método se realizaría un valor hash de una manera rápido y obtendríamos un valor directamente y no sería necesario enlazar varios como anteriormente se comentó. Aunque este método tiene varios problemas, como puede ser, y dependiendo del lenguaje, la obtención de esas claves, ya que las funciones aleatorias a veces no funcionan tan bien como nos gustaría. Otro de los problemas que acompañan este método, aunque es demasiado poco común, es la posibilidad de que, al realizar el hash de un estado obtengamos el mismo valor que se podría obtener realizando el has de otro. Este caso dependerá de los valores aleatorios generados anteriormente y, claramente, se tendrá difícil control sobre ellos.

8. Conclusiones

Como se observan en los resultados de las pruebas realizadas, con el heurístico utilizado, los jugadores ganan más conforme su nivel asignado y profundidad, pero eso significa que se haya dado con el mejor heurístico posible, ya que un jugador bastante experimentado y que conozca el tipo de tácticas utilizadas y algunas más, podría vencer sin problemas a estos jugadores.

También tener en cuenta que en un juego cuya ramificación aumenta considerablemente turno tras turno y que la profundidad utilizada no es demasiado grande para evitar tiempos eternos de esperas que, pese a contar el programa con un tiempo máximo por turno, abusar del mismo aumentando la profundidad mucho sería una mala práctica, ya que el algoritmo nunca comprobaría todas las ramas y, con ello, obtener resultados erróneos y que probablemente no lleven al mejor resultado final.

Lo más positivo que me llevo de este desarrollo es el aprendizaje de un nuevo entorno de programación como es Unity3D y Microsoft Visual Studio, junto con un lenguaje bastante completo como C#. Ha sido bastante fácil la adaptación a todos ellos, aunque el aspecto visual de Unity3D si tuve algo más de trabajo al no haberlo utilizado prácticamente nada con anterioridad. Pero eso no le quita al proyecto que, las horas que pasé realizándolas, hayan sido satisfactorias tanto en aprendizaje como en una meta personal que cumplir.

Lo negativo probablemente sea la falta de tiempo y de experiencia en este tipo de proyectos, lo que me llevó a no poder realizar todo lo que me habría gustado, como la implementación de la función hash de claves Zobrist o incluso llegar a añadir la posibilidad de realizar partidas en línea, función que Unity tiene integrada pero que, por desconocimiento, me habría resultado un gasto enorme de tiempo.

Referencias bibliográficas

[1] Manual Unity3D; <https://docs.unity3d.com/Manual/index.html>

[2] Microsoft Visual Studio; <https://www.visualstudio.com/es-es/docs/vs/overview> [Consulta: 23 de Mayo de 2017]

[3] Demián Gutierrez; *“Métodos de Desarrollo de Software”*; Julio 2011. <http://www.codecompiling.net/files/slides/IS_clase_13_metodos_y_procesos.pdf>

[4] “Desarrollo iterativo”; <https://proyectosagiles.org/desarrollo-iterativo-incremental>

[5] Steve MacGuire; <http://www.samsoft.org.uk/reversi/strategy.htm>

[6] Michael J. Korman; <http://mkorman.org/othello.pdf>

[7] Reversi/Othello y sus reglas; <https://es.wikipedia.org/wiki/Reversi>

[8] J.D. Funge; *“Artificial Intelligence for Computer Games”*; 2004

[9] “Algoritmo Negascout” <https://chessprogramming.wikispaces.com/NegaScout>

[10] Ian Millington, John Funge; *“Artificial Intelligence for Games”*; 2009.

[11] Stuart J. Russel; *“Artificial intelligence: a modern approach”*; 2010.

[12] David M. Bourg and Glenn Seemann.; *“AI for game developers”*; 2004.

[13] Alberto García Serrano; *“Inteligencia artificial: fundamentos, práctica y aplicaciones”*; 2016.