AGENTE JUGADOR DE OTHELLO

Búsqueda entre adversarios con MINIMAX

Implementación del algoritmo de búsqueda MINIMAX con poda para dotar a un agente con la capacidad de jugar Othello.

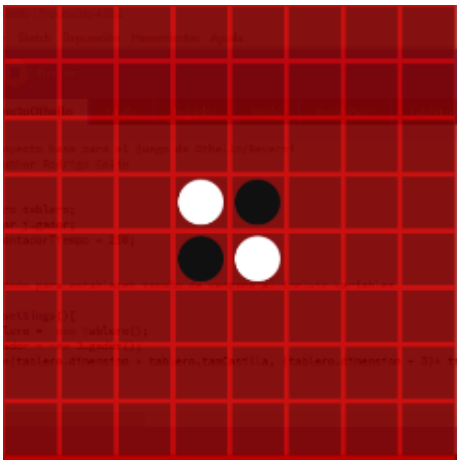
Diego García Vázquez

Índice

Introducción

El trabajo aquí presentado tiene el objetivo de exponer el desarrollo de un agente con la capacidad de jugar Othello. Comenzaremos dando a conocer los elementos que conforman al agente, el problema que se le ha planteado, el entorno donde trabajará y lo más importante, como dará solución al problema.

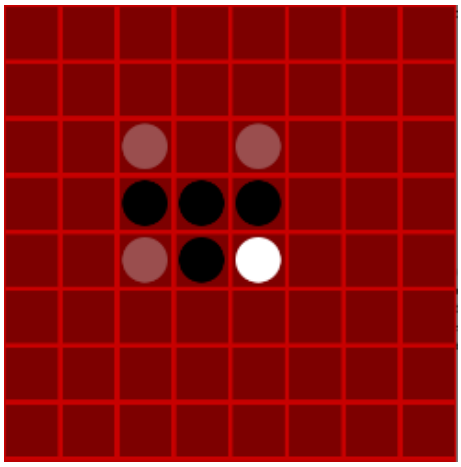
Antes de continuar con la descripción del agente, hablaremos sobre el juego que pretende ganar, el juego Othello es un juego de mesa que de dos jugadores que consta de un tablero de 8x8, 64 fichas bicolor, es decir, las caras tienen colores diferentes (usualmente blanco y negro). A cada jugador se le asigna un color y jugaran por turnos, salvo que un jugador no tenga movimientos disponibles, tendrá que saltar su turno hasta que tenga movimientos disponibles o bien, termine la partida.



Configuración inicial del Tablero

Al comenzar el tablero se encuentra con únicamente 4 fichas, dos para cada jugador, colocadas de la siguiente manera:

El primer turno corresponde al jugador de las fichas negras. Cada movimiento consiste en colocar una ficha en una casilla del tablero que logre encerrar una hilera de cualquier tamaño (horizontal, vertical o diagonal) de fichas del oponente, la hilera o hileras encerradas son capturadas y se voltean, es decir, cambian al color del jugador que realizo la jugada, de esta manera el número de fichas del jugador que ha hecho el movimiento ha aumentado, las fichas del oponente han disminuido y el tablero tiene una casilla menos para realizar un movimiento. La partida termina cuando ningún jugador pueda realizar más movimientos (Claramente eso incluye el caso cuando el tablero se ha llenado) y gana el jugador que ha posicionado mas fichas sobre el tablero, pudiéndose dar un caso de empate si ambos han capturado 32 fichas. Una observación importante es que el numero de fichas que cada jugador tiene a lo largo de la partida resulta irrelevante a la hora de saber que jugador tiene ventaja o bien, tiene más posibilidades de ganar, pues en cuestión de muy pocos turnos pueden ser capturadas gran cantidad de fichas.



Jugador 1 tira en casilla [3,4] y captura [4,4]

El juego Othello, como ya se mencionó, tiene reglas muy sencillas, sin embargo, ganar una partida puede ser una tarea complicada, hace falta un conjunto de estrategias muy eficientes y también mucha intuición, ambas se pueden desarrollar mediante la experiencia. Para construir un agente que sepa jugar, o al menos no se deje perder será necesario dotarlo de ese conocimiento que nosotros como humanos somos capaces de ganar mediante experiencia, es decir, debemos *enseñarle* al agente como jugar, pero debemos hacerlo de una manera que sea capaz de entender y sobre todo aplicar. Para este punto ya es indispensable haber abstraído totalmente tanto el problema a solucionar como la manera en que lo hará, una vez que el agente sepa jugar, debemos asegurarnos que sea capaz de interactuar con el entorno, de este modo, nuestro agente podrá cumplir con su objetivo y dar solución al problema, que en este caso es ganar la partida.

Definición del problema

Para comenzar a construir un agente, es primordial conocer el entorno donde se desarrollará, así como el problema que debe resolver. Dado que buscamos un agente capaz de jugar Othello, no es muy complejo de analizar su entorno, ya que este se limita un tablero de 8x8, sin embargo, el cómo de desarrolla el juego resulta algo menos trivial, pues no basta con conocer las fichas colocadas, más adelante detallaremos una descripción del entorno. Ya que sabemos dónde vivirá nuestro agente, es momento de analizar cuál será su objetivo, es decir, el problema al que dará solución, resulta mucho más sencillo de definir el problema que el entorno, pues el objetivo del agente es ganar, que se traduce en **alcanzar** una configuración del tablero que no tenga más movimientos y se tengan más fichas del agente que del oponente.

El entorno

Abstraer el entorno para el agente es sencillo en un principio, consta de una matriz de 8x8 (Tablero), donde cada entrada corresponderá al estado actual de la casilla correspondiente, sin embargo, es importante que el tablero admita únicamente un movimiento legal, que conozca el turno del siguiente movimiento, y que se actualice correctamente una vez que se ha hecho una jugada. En pocas palabras, el entorno debe ser consistente con el estado actual de la partida.

Ahora que conocemos que elementos conforman el entorno, podemos comenzar a dar una definición del mismo:

Tablero	
mundo	`//Matriz de 8x8 que indica el estado del tablero`
turno	`//Valor booleano que indica que jugador debe jugar`
numeroDeTurno	`//Valor entero que indica el número de turno actual`
setFicha(x,y)	`//Método que realiza movimiento en la casilla` (x,y)
jugadasPosibles()	`//Función que regresa coordenadas de jugadas legales`
esJugable(x,y)	`//Función que indica si la casilla` (x,y) `es un movimiento legal`
sinMovimientos()	`//Función que indica si hay movimientos para el turno actual`
finPartida()	`//Función que indica si la partida ha concluido o no`

En el esquema anterior, vemos las características que identifican el estado del entorno y algunas acciones que debe ser capaz de realizar para interactuar con los jugadores. Como podemos ver el tablero indica cuando una jugada es o no posible, de este modo restringe a los jugadores a mover fichas únicamente donde pueden hacerlo.

Ya que hemos abstraído el entorno de trabajo, ahora debemos aclarar como el agente va a interactuar en el, es decir, la REAS. Cuando una persona juega Othello, lo único que necesita es ver el tablero y poder colocar una ficha, de una forma similar, el agente solo necesitará acceder a las propiedades del tablero mediante las funciones y métodos que se definieron con anterioridad, de ahí la importancia de una correcta abstracción del entorno. A continuación se enuncia la REAS de nuestro agente:

- Medida de Rendimiento: Para medir el rendimiento de nuestro agente debemos evaluar la eficiencia con la que desempeña su trabajo, es decir, la eficiencia con la que gana una partida. Los factores que consideraremos para medir la eficiencia del agente son el tiempo de ejecución, el espacio de memoria que utiliza y que tan bien juega (proporción de partidas ganadas).
- Entorno: El entorno ya se ha estudiado y corresponde al tablero de 8x8 y el estado actual del juego (numero de turno y jugador que realizará el siguiente movimiento).
- Actuadores: Dado que se trata de un agente abstracto y que está construido en su totalidad por software, sus actuadores serán funciones o métodos que le permitan interactuar con su entorno. El único actuador que nuestro agente necesita es un método que le permita realizar un movimiento, es decir, colocar una ficha en el tablero en una posición legal, para esto primero tendrá que saber si la jugada es legal, pero eso es tema de los sensores
- Sensores: Al igual que los actuadores, los sensores de nuestro agente serán funciones que consulten el estado actual del entorno, en otras palabras, funciones que tomen como parámetro el tablero y de él obtengan la información necesaria.

El problema

Ya tenemos claro que problema debe solucionar el agente, alcanzar un estado donde este sea quien gane la partida, es decir, llegar a un estado donde no haya movimientos posibles y el agente tenga más fichas capturadas sobre el tablero, suena sencillo, sin embargo llegar a ese estado no es tarea fácil debido a que el movimiento que hará el oponente en el siguiente turno es desconocido, por ello no bastará con una simple búsqueda entre el espacio de estados, será necesario añadir un factor importante que es suponer que el oponente siempre hará el mejor movimiento posible.

Al suponer que el oponente tomará siempre el mejor movimiento disponible, nuestro agente estará preparado para el peor de los escenarios posibles, y en ese supuesto, el agente también tomará el mejor camino que lo pueda guiar a un estado final que le de la victoria.

Hasta este punto se ha mencionado un factor fundamental para nuestro agente que no hemos descrito con claridad, se trata de lo que consideramos el *mejor movimiento posible.* Al hablar del mejor movimiento posible es necesario definir un criterio que, dados dos estados, nos diga cual es *mejor*. Siendo más formales, es necesario definir un orden en el espacio de estados, lo más lógico sería ordenar los estados según sus posibilidades de alcanzar un estado de éxito, de este modo, el agente dará prioridad a aquellos estados que le ofrecen una mayor probabilidad de éxito, una forma de lograrlo es explorar los caminos completos desde el estado actual hasta los estados terminales, dando prioridad a los caminos que conduzcan al éxito y evitando los que conducen a perder la partida.

El orden definido anteriormente puede parecer prometedor, pero como jugadores experimentados de Othello, sabemos que explorar un árbol hasta encontrar un caso de éxito puede ser extremadamente complejo incluso para una computadora, mas adelante estudiaremos la complejidad de expandir el árbol de una partida de Othello, por ahora nos centraremos en cómo evitar expandir hasta el fondo el árbol de una partida. Si expandimos el árbol una cantidad limitada de niveles, no necesariamente encontraremos un estado terminal, de modo que nuestro criterio de orden ya no sirve, eso nos lleva a extender nuestra definición de orden cuando no se alcanzan estados terminales, aquí es donde nos corresponde *enseñarle* al agente como jugar, es momento de aplicar toda la experiencia adquirida como jugadores y dotar al agente de la capacidad de elegir cuál será el mejor camino a seguir, la solución es definir una heurística que le permita al agente seguir un camino que pese a no llevarlo a un estado de éxito, lo acerque tomando en cuenta ciertos elementos del estado, tales elementos serán evaluados por la función heurística, y aquel estado que maximice la función (o minimice en caso de ser un estado donde el oponente debe jugar) será el estado más prometedor para el agente, llevándolo por un camino que le dé más posibilidades de éxito. Es muy importante plasmar de manera correcta dentro de la función heurística la experiencia e intuición que como jugadores nos han hecho ganar partidas, ya que del poder de la función dependerá el poder del agente.

Finalmente, durante la descripción del proceso anteriormente mencionado, se hablo de maximizar (o minimizar) una función de evaluación, por tal motivo, el algoritmo que se empleará para la búsqueda de un camino prometedor será el algoritmo MINIMAX, la idea del algoritmo es la que se ha descrito para el funcionamiento del agente, una búsqueda en profundidad que propaga hacia los nodos superiores los valores obtenidos en los nodos más profundos, tomando máximos o mínimos según la paridad del nivel de búsqueda (maximizar los valores obtenidos en los turnos del agente y minimizar en los turnos del oponente).

Planteamiento de la solución

Durante la definición del problema se expuso a que se enfrenta nuestro agente, como se puede abordar el problema y que elementos se necesitan, sin embargo poco se ha hablado de nuestro agente, en esta sección detallaremos todo lo referente al agente.

En la sección anterior se menciono la importancia de definir una adecuada función heurística para establecer un orden en el espacio de estados, mencionamos también la forma en que se recorrería el árbol de estados y se dio un acercamiento al algoritmo MINIMAX, todos estos componentes serán las entrañas de nuestro agente.

Función de evaluación

Supongamos que tenemos dos posibles movimientos, cada uno conduce a diferentes caminos que al cabo de unos cuantos pasos podríamos tener resultados abismalmente diferentes, la pregunta es, de todos los posibles estados que se pueden alcanzar, ¿Cuál será el más adecuado? y no menos importante, de los dos posibles movimientos que inicialmente teníamos, ¿Cuál de los dos elegir? Actualmente ya sabemos que esa decisión es fácil de tomar si exploramos todas las posibilidades y vemos cual nos conduce a la victoria, pero en lugar de obtener respuestas solo encontramos más preguntas, ¿Qué tan profundo tendremos que ir para encontrar un estado terminal? , Si ambos movimientos llevan a una eventual victoria, ¿Cuál de los dos deberíamos elegir? , Si no tenemos la capacidad de llegar hasta un estado terminal, ¿Cómo sabremos qué camino tomar? Las respuestas a todas estas preguntas se resuelven implementando una función de evaluación. Esta función de evaluación debe indicar que tan optimo es un camino, encuentre o no un estado terminal, es claro que debe tomar en cuenta las estrategias que como humanos llegamos a utilizar, aunque debemos ser cuidadosos, ya que dotar al agente de estrategias que no son del todo buenas, no conseguirá que juegue tan bien como esperamos, por ello es indispensable tener suficiente experiencia en el juego.

La función de valuación propuesta para el agente es la siguiente:

$$evalFunc(n) = \begin{cases} 10000 * (fichasAgente - fichasJugador) & si\ n\ es\ estado\ terminal \\ heuristica(n) & si\ n\ no\ es\ estado\ terminal \end{cases}$$

Analicemos parte por parte la función de evaluación. Primeramente la función de evaluación toma como parámetro un estado (o Nodo), recordemos que el objetivo de la función es definir un orden entre los estados del entorno, por lo tanto *evalFunc* está definida sobre el espacio de estados. Veamos ahora la definición de la función, podemos ver que el valor que retorna cambia dependiendo si se trata de un nodo terminal o no, en caso de ser un nodo terminal la función de evaluación es proporcional a la diferencia de fichas entre el agente y el oponente, si el agente

gana entonces el resultado será positivo, en caso opuesto será negativo, notemos que si se trata de un empate, el resultado es cero. Un último detalle de la función cuando se trata de un nodo terminal es el factor de proporcionalidad, se ha asignado un valor con la intención de que el mínimo valor de obtener una victoria sea superior a cualquier valor de aplicar la función heurística a cualquier estado, dando prioridad a los caminos que conducen a una victoria dada una profundidad fija.

Cuando se aplica la función de evaluación a un nodo no terminal, esta retorna el valor heurístico de dicho nodo, en este caso, si todos los estados alcanzados hasta una profundidad constante no son terminales, se optara por seguir un camino que parezca más prometedor según nuestra función heurística, nuevamente hacemos énfasis en la importancia de plasmar correctamente las estrategias a seguir en nuestra función heurística.

En el siguiente apartado hablaremos sobre las estrategias a seguir y como plasmarlas en una función heurística.

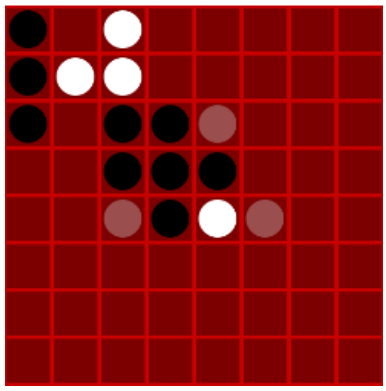
Estrategia de juego Otello

La experiencia adquirida por jugar Othello nos dice que un factor determinante ha sido capturar las equinas, cada partida que el oponente ha capturado las esquinas las posibilidades de ganar han sido casi nulas, por otro lado, una vez que se logran capturar las esquinas, las posibilidades de ganar aumentan, esto mejora si se captura toda una fila o columna de algún extremo. Con base en estas observaciones se planteará la siguiente estrategia para indicar que movimientos debe hacer un agente al jugar contra un oponente.

Captura de esquinas

Capturar una esquina asegura una posición estratégica, esto se debe a dos razones, primero que una esquina capturada jamás podrá caer en manos del oponente, es decir, una vez que capturas una esquina, el oponente jamás podrá robarla. La siguiente razón para darle prioridad a las esquinas, es que estas ofrecen un buen punto de partida para capturar toda una columna o fila de un extremo, el interés en las hileras de los extremos se detallará más adelante.

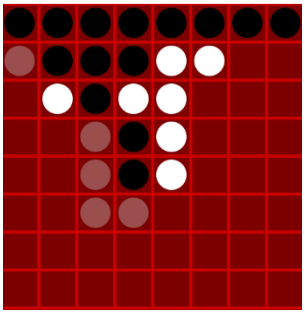
El hecho de tener una esquina ofrece una ventaja también al momento de capturar una diagonal, si se consigue capturar una diagonal completa, tenemos una ficha en medio de cada fila o columna a lo largo y ancho de todo el tablero, es decir, tenemos un punto que parte por el interior a cada fila y columna de tablero, dejando expuestas gran cantidad de hileras del oponente para ser capturadas por al menos un extremo.



Captura de esquina superior derecha

Captura de hileras en los extremos

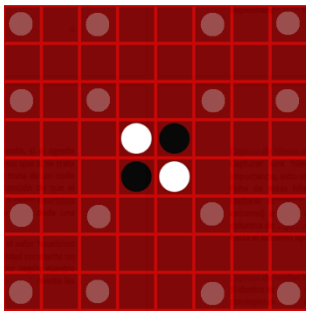
Capturar una hilera de un extremo es una posición de suma importancia, esto se debe a que no solo es imposible recuperar alguna ficha de estas hileras, sino que ofrece un punto de partida para capturar columnas (en caso de haber capturado la fila de algún extremo) o bien, capturar las filas (si es que se capturo toda una columna de algún extremo). Dicho de otra forma, permite expandirnos hacia el extremo opuesto.



Captura de fila superior

Captura de casillas X

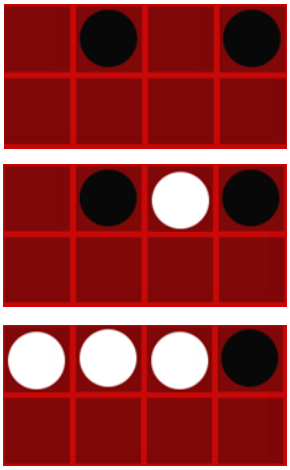
Si dentro del tablero ubicamos cuadros de 3x3 en casa esquina, las esquinas de cada cuadro corresponderán a las casillas que denominaremos X. La importancia de estas casillas se debe a que capturarlas nos facilita llegar a las esquinas. Si se captura una casilla X, deja al oponente en una situación complicada al momento de acercarse a la esquina mas cercana, pues acercarse aun más a la esquina implica capturar una casilla adyacente a la esquina, movimiento muy peligroso si no se cumplen ciertas condiciones, por lo tanto, capturar estas casillas es lo mas cerca que podemos estar de las esquinas sin acercarnos suficiente de modo que el oponente pueda llegar primero a la esquina.



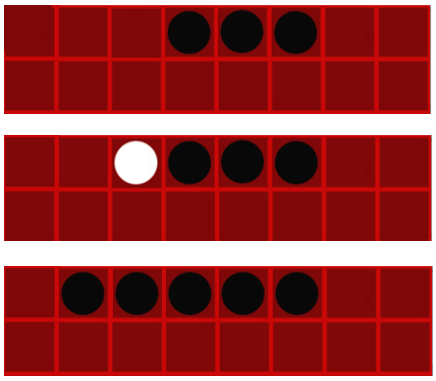
Casillas X

Manejo de casillas adyacentes

Con casillas adyacentes nos referimos a aquellas casillas adyacentes a las esquinas, el manejo adecuado de estas casillas es fundamental, pues son la puerta de acceso a las esquinas, ya sea en favor de nosotros o del oponente, tomar una casilla adyacente de manera anticipada puede representar el comienzo de la derrota o una captura estratégica, pero, ¿De qué depende que la captura de las casillas adyacentes sea benéfica o perjudicial? Como ya se menciono las casillas adyacentes son la puerta de entra a las esquinas, de modo que capturar una casilla adyacente si aun no se tiene control de toda la hilera de un extremo a la que pertenece la casilla, puede representar riesgo, ya que, si el oponente captura alguna casilla de dicha hilera, puede acercarse a la casilla adyacente eventualmente y capturarla colocando ficha en la esquina, ganando esa posición tan valiosa. En la ilustración se puede apreciar una serie de movimientos en la que el oponente captura las esquina dado que capturamos anticipadamente la casilla adyacente sin tener previamente la casilla X cercana.



Oponente captura esquina mediante casilla adyacente



Captura de casilla adyacente y expansión de hilera superior

En contraste con el ejemplo anterior, veremos como capturar esta casilla puede representar una ventaja. Cuando tenemos una hilera continua de fichas capturadas, el oponente puede robar solo si la rodea, eso le llevará al menos dos turnos, por lo tanto, si el oponente intenta rodear dicha hilera, al siguiente turno nosotros capturaremos la ficha recién colocada, esta acción no solo impide que el oponente capture la hilera completa, sino que nos permite extendernos hacia los extremos, sin embargo habremos de ser aún más cuidadosos de evitar que las esquinas adyacentes a la hilera formada no sean capturadas por el oponente.

Cantidad de movimientos y fichas capturadas

Un último factor que tomaremos en cuenta a la hora de jugar es cuantos movimientos disponibles nos deja una jugada, recordemos que limitar los movimientos del oponente puede resultar favorable debido que un turno donde el oponente no tenga movimientos disponibles nos da la oportunidad de mover de nuevo y capturar aun mas fichas. Del mismo modo, maximizar la cantidad de movimientos que pueda realizar el agente le permite encontrar más posibilidades.

Para complementar la estrategia actual, tomemos en cuenta la cantidad de fichas, si bien, por si solas no entrega información suficiente, este criterio será bueno para valorar que estado es mejor que otro siendo que todos los criterios anteriores suman el mismo valor.

Función heurística para Othello

Ya que hemos visto diferentes estrategias para obtener ventajas a la hora de jugar Othello, es momento de plasmarlas en la lógica del agente, encontraremos la mejor manera de combinar las estrategias de modo que nos dé la mejor estimación de éxito y a la vez sea computable en tiempo y espacio moderado.

En el apartado anterior se habló sobre las estrategias a seguir para aventajar una partida de Othello, claramente no todas tienen la misma prioridad, de modo que se van a ponderar para otorgar la jerarquía correspondiente. Si analizamos bien las estrategias, el objetivo central es la captura de las esquinas, por ello este criterio será el más fuerte.

La función heurística se define del a siguiente manera:

$$h(n) = 64E + 16X + 16\sum_{i=1}^{4}(E_i == 0\ ?\ -4 : 1)C_i + 4H_1 + 16H_2 - 4m + p$$

Donde:

E: $Suma\ de\ las\ casillas\ esquina$

X: $Suma\ de\ las\ casillas\ esquina\ de\ los\ cuadros\ 3x3$

C_i: $Suma\ de\ las\ casillas\ adyacentes\ a\ la\ esquina\ i$

H_1: $Hileras\ formadas\ en\ los\ extremos$

H_2: $Hileras\ \text{sin}\ esquinas\ \ formadas\ en\ los\ extremos$

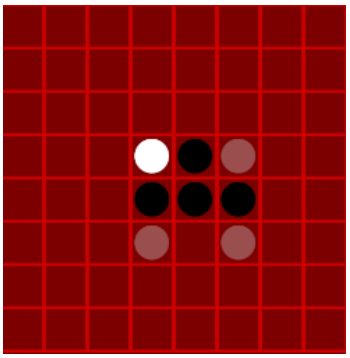
m: $Movimientos\ disponibles\ del\ oponente$

p: $Diferencia\ \ de\ fichas\ del\ jugador\ y\ el\ oponente$

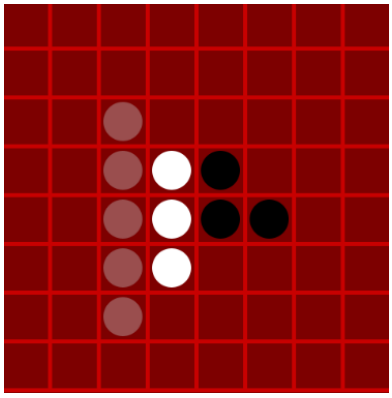
Cabe mencionar que para esta función se contabiliza una casilla vacía como 0, una casilla capturada como 1 y una casilla capturada por el oponente como -1, así los valores más altos corresponderán a aquellos donde la función *h* es óptima. Veamos algunos ejemplos.

Ejemplos de la función heurística.

Para exponer el funcionamiento de nuestra heurística, se hará una exploración de un solo nivel y se calculará *h* en cada uno de los posibles caminos.



En este ejemplo tenemos 3 posibles jugadas, podemos jugar en la casilla D6, F4 y F6, cada una nos dara un estado diferente y tomaremos aquel que maximice *h.*



Si jugamos en la casilla D6 el resultado de h es el siguiente:

$$h = 64 * 0 + 16 * 0 + 16 * 0 + 4 * 0 + 16 * 0 + (3 - 3) - 4 * 5$$

$$h = -20$$

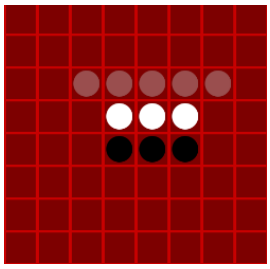
Como podemos ver, el unico criterio que favorece esta jugada es la cantidad de movimientos disponibles , sin embargo sabemos que este por si solo no es un buen criterio para definir si una jugada es buena o no, por lo tanto no parece ser una jugada muy prometedora.

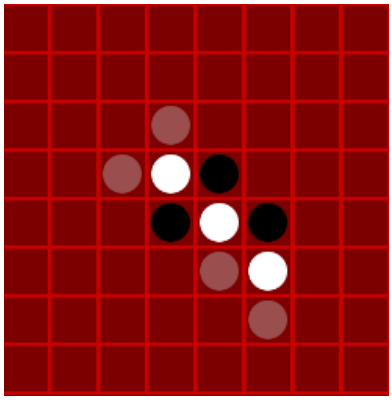
Si por otro lado, jugamos en la casilla F4, el resultado es el siguiente:

$$h = 64 * 0 + 16 * 0 + 16 * 0 + 4 * 0 + 16 * 0 + (3 - 3) - 4 * 5$$

$$h = -20$$

Nuevamente, el unico criterio que aporta información es la cantidad de movimientos disponibles





Finalmente, si jugamos en la casilla F6, el resultado cambia considerablemente, al evaluar h obtenemos lo siguiente:

$$h = 64 * 0 + 16 * 1 + 16 * 0 + 4 * 0 + 16 * 0 + (3 - 3) - 4 * 4$$

$$h = 0$$

La diferencia es notoria con respecto a los resultados de las otras jugadas, al ser esta la jugada que aporta el valor maximo de h, será el camino elgido para continuar con la partida.

En la practica, la jugada F6 resulta ser la mas prometedora ya que esta facilita la captura de las casillas X de la esquina inferior izquierda.

Si continuamos usando esta función, obtendremos buenos resultados en general, sin embargo para explotar su potencial hace falta evaluarla a mayor profunidad para darle mayor precición, es decir, aplicarla en el algoritmo MINIMAX.

Para concluir, la función heurística desarrollada muestra ser un buen criterio para medir que tan viable es una jugada, sin embargo, para ponerla a prueba debe implementarse en un árbol de búsqueda profundo que sea capaz de conocer casos más allá de lo que el humano podría considerar dada la limitación que tenemos frente a las computadoras. En la siguiente sección se hablará del algoritmo MINIMAX y como la función heurística toma parte fundamental en la implementación de este.

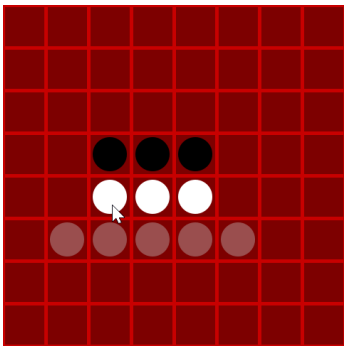
Algoritmo MINIMAX

En el planteamiento de la solución se mencionó un supuesto sumamente importante, suponer que el oponente siempre hará *el mejor movimiento,* ahora que ya sabemos cómo elegir el mejor movimiento (aquel que conduzca al mejor resultado) es momento de abstraer este procedimiento con un algoritmo. El algoritmo correspondiente a la solución que plantea el agente es el algoritmo MINIMAX, recibe ese nombre debido que el agente siempre buscará MAXimizar los valores obtenidos para cada movimiento, mientras que el oponente buscará MINImizarlos (recordemos que valores mayores son benéficos para el agente, mientras que valores menores son benéficos para el oponente).

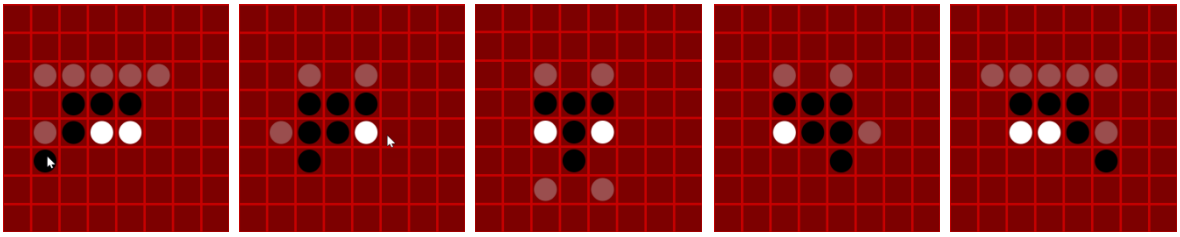
Para explorar caminos de estados del juego nos apoyaremos en una estructura de árbol, donde dado un estado (nodo padre), por cada movimiento posible se expandirá su estado sucesor (nodos hijo), considerando siempre el turno de cada nivel. De esta manera, en algún punto se llegará a un nodo terminal que indicará la ruta más efectiva para el agente, sin embargo, antes de llegar a un nodo terminal, los nodos hoja serán evaluados con la función heurística.

Para poder comprender con claridad el algoritmo MINIMAX haremos uso de un ejemplo donde se exploren únicamente dos niveles (por simplicidad).

Las fichas negras corresponden al agente, supongamos ahora que nos encontramos en el siguiente estado y es turno del agente:

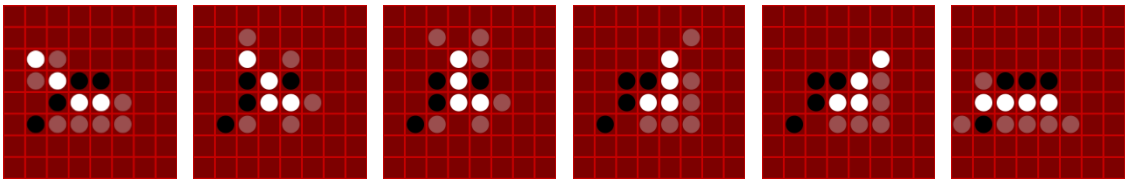


Como podemos ver, el agente tiene 5 posibles movimientos, cada movimiento da como resultado los siguientes estados:

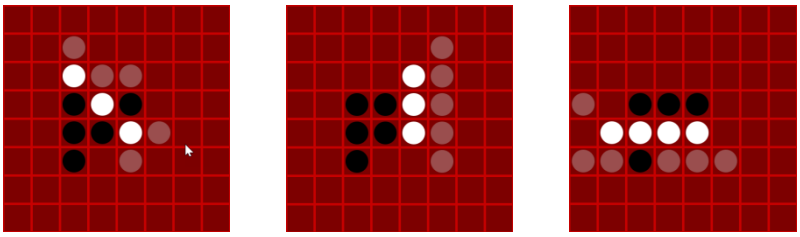


Intuititivamente podemos notar que dos estados parecen ser mas prometedores, el segundo y el cuarto, sin mebargo no nos dejemos llevar por el resultado actual, sino por el mejor resultado al que nos pueden conducir.

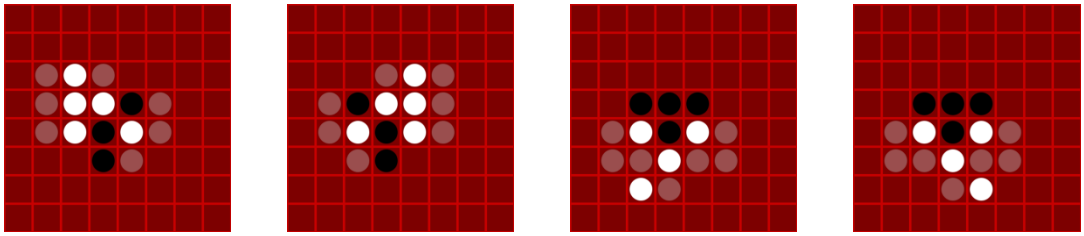
Exploremos el primer estado hijo, tomemos en cuenta que es turno del oponente el cual tiene 6 posibles movimientos, los estados resultantes de aplicar dichos movimientos son los siguientes:



Para el segundo nodo hijo del segundo nivel, tenemos los siguientes posibles estados:

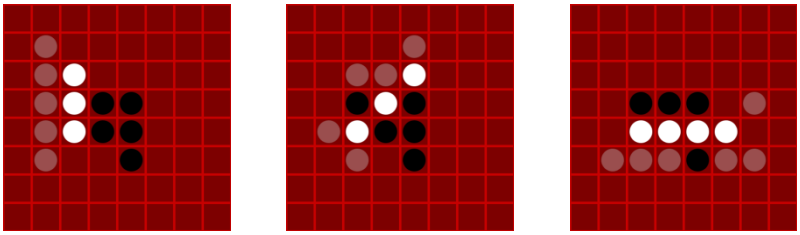


Para el tercer nodo hijo del segundo nivel, tenemos los siguientes posibles estados:

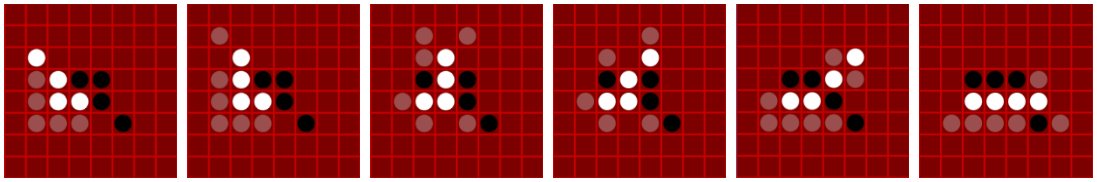


Para el cuarto y quinto hijo del segundo nivel, los resultados son muy parecidos al primero y segundo, con diferencia de estar reflejados y desplazados, a simple vista podría parecer que son situaciones igual de beneficas o perjudiciales, pero veremos que son en realidad muy distintos.

Posibles estados hijo del cuarto nodo del segundo nivel:

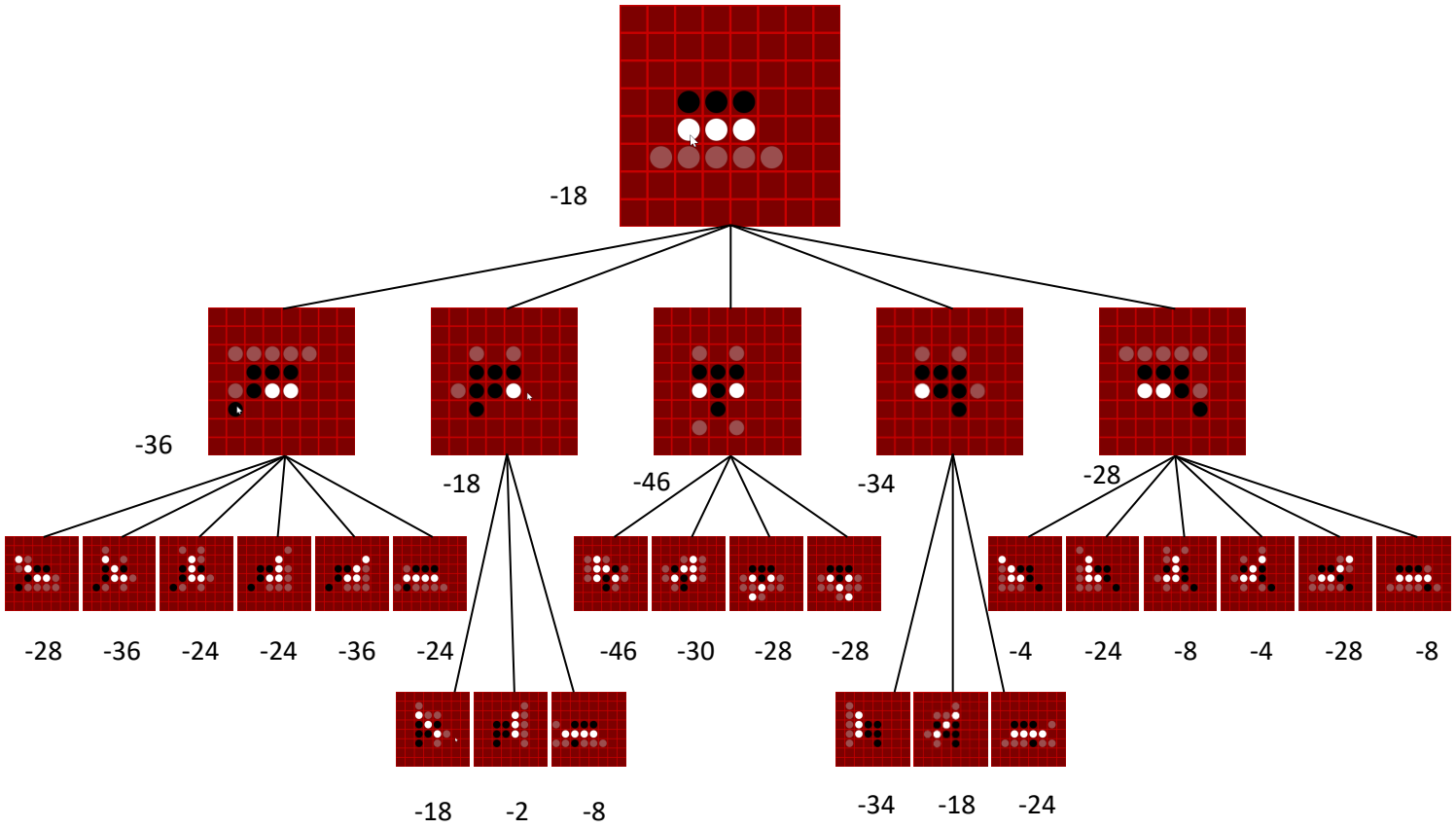


Posibles estados hijo del cuarto nodo del segundo nivel:



Hasta ahora solo hemos explorado dos niveles, vemos que ya contamos con una cantidad considerable de nodos explorados, comenzamos con 5 posibles movimientos, después cada hijo tiene 6,3,4,3,6 posibles movimientos respectivamente, en promedio es un factor de bifurcación de 4.4, sin embargo confirme aumentan los movimientos, este factor varia considerablemente, de hecho el factor de bifurcación aumenta hasta llegar a un punto donde hay demasiadas piezas sobre el tablero y el factor comienza a disminuir nuevamente hasta llegar a cero (fin de la partida). La complejidad temporal del árbol podría aproximarse al orden $O(b^n)$ donde b es el factor de bifurcación y n el número de niveles que descendemos. Para aproximar el factor de bifurcación veremos su comportamiento a lo largo de los turnos del juego, recordemos que la heurística busca minimizar el número posible de movimientos del oponente. Por ahora nos enfocaremos en ejemplificar MINIMAX, mas tarde se hablará sobre la complejidad temporal y espacial.

Claramente ninguno de los nodos hoja alcanzados son nodos terminales, de modo que al aplicar la función de evaluación en realidad se estará aplicando la función heurística previamente definida, veamos como se ve el árbol.



Ahora que hemos evaluado los nodos hoja, es momento de decidir que camino deberá tomar el agente, haremos de las reglas mencionadas al inicio de esta sección, maximizar los valores en el turno del agente u minimzarlos en el turno del oponente. En el estado inicial, es turno del agente de mover, por lo tanto, los nodos hijos corresponden a un turno del oponente, habra entonces que tomar el valor minimo de los nodos hoja para cada nodo hijo. Para el primer nodo hijo el valor mas pequeño es -36, notemos que hay dos nodos que comparten el valor minimo, en general, la decisión dependerá de la implementación del algoritmo, puede tomar el primero o el ultimo, o incluso podría aplicarse un nuevo criterio como evaluar al función heuristica, la mejor manera de decidir que camino tomar es ir mas profundo. Para los siguientes nodos hijo tomaremos nuevamente el minimo de sus nodos hoja correspondientes, dejandonos los valores -18,-46, -34 y -28. Ya que hemos propagado un nivel hacia arriba los valores optimos para el jugador del turno correspondiente (en este caso el agente) repetiremos el procedimiento pero ahora con el otro jugador (en este caso el agente), en lugar de tomar los minimos tomaremos el máximo, facilmente podemos identificar que el máximo es -18, esto significa que, si el oponente siempre juega el mejor movimiento disponible, al cabo de dos movimientos en el mejor de los casos nuestro agente llegará a un estado que al evaluarlo dará un valor de -18, no parece algo prometedor, sin embargo esto puede deberse a que al descender únicamente dos niveles hemos caido en el turno del oponente, haciendo que en ese momento el tenga una ventaja sobre el tablero, ventaja que se contrarrestará al siguiente turno cuando el agente haga un movimiento nuevamente.

Como podemos ver, la escencia del algoritmo MINIMAX es acotar las posiblilidades de victoria del oponente, suponiendo siempre que realiazará el mejor movimiento posible, entonces nosotros iremos tomando caminos donde el mejor movimiento del oponente sea menos perjudicial en comparación con haber tomado otro camino, es decir, realizar otro movimiento. Finalmente, en nuestro ejemplo, el movimiento del agente será jugar en la casilla [3,6], notemos que este movimiento coincide con lo que intuitivamente veiamos como un movimiento más prometedor.

Para implementar el algoritmo MINIMAX se hará una exploración en profundidad, como la profundidad esta limitada, el algoritmo será eficiente en cuanto al uso de memoria, ya que la cantidad de nodos en pila será proporcional al factor de bifurcación. Notemos que a mayor profundidad, el agente tendrá una visión mucho mas completa de la evolución de la partida, es necesario poner a prueba al agente para encontrar un equilibrio entre la profundidad y el tiempo de ejecución.

Para cerrar esta sección, como ha sido costumbre, formalizaremos el concepto del algoritmo MINIMAX mostrando el pseudocódigo con la intención de escribir de manera formal y analitica el funcionamiento del algoritmo.

```
int MINIMAX(Nodo, turno, profundidad)
{
      If(profundidad >= limite || esNodoTerminal(Nodo))
            return evalFunc(Nodo, turno)
      For each hijo h de Nodo:
            M[i++] = MINIMAX(h, !turno, profundidad + 1)
      return (turno ? max(M) : min(M))
}
```

El algoritmo MINIMAX tiene una implementación sencilla, de hecho, la manera de plantear la solución tambien lo es, todo se basa en el supuesto de que el oponente siempre tomará el mejor moviento que tenga disponible y en la manera que hemos desarrollado para medir que tan bueno es un estado, con todos los componentes listos ya estamos listos para implementar al agente que jugará Othello. En la siguiente sección se describirá todo lo referente a la implementación del agente, el entorno y se mostrará el funcionamiento del mismo, comparandolo contra otros agentes ya existentes en el mercado.

Implementación del agente y su entorno

Diseño de clases

Como siguiente paso, vamos a llevar a la práctica todo aquello que se desarrolló en las secciones anteriores. Dado que buscamos simular un juego que existe en la vida real haremos uso de la Programación Orientada a Objetos ya que modelar cada elemento que compone una partida de Othello con clases será de gran utilidad y practicidad a la hora de codificar.

Como primer paso identificaremos cada elemento que conforma nuestro modelo de la partida de Othello. El elemento principal ya lo tenemos identificado, y de hecho ya hemos dado una definición de él durante el planteamiento del problema, se trata del tablero. El siguiente componente es nuestro agente, después de todo, será este quien lleve la lógica del juego. Para apoyarnos en la búsqueda de estados y la implementación de MINIMAX haremos uso de una estructura que servirá como Nodo para nuestro árbol de búsqueda. Finalmente, las clases que conformarán el proyecto son las siguientes:

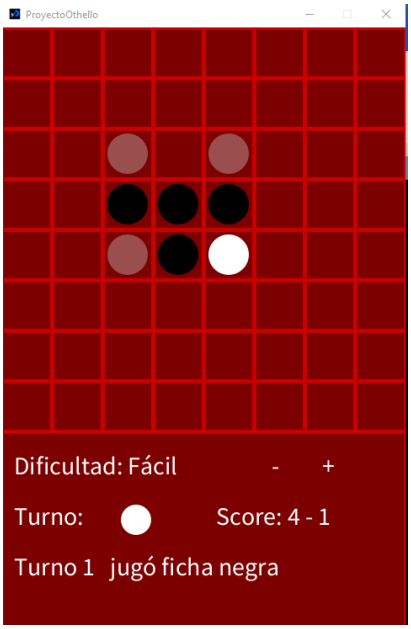
Jugador	
profundidad	`//Valor entero que indica la máxima profundidad de exploración`
setDificultad(n)	`//Método que ajusta la profundidad de exploración`
Jugar(Tablero)	`//Función que explora árbol e indica la casilla por Jugar`
generaHijos(Nodo)	`//Función que agrega nodos hijo al nodo actual`
heuristica(Nodo)	`//Función que obtiene el valor heurístico de un nodo`
evalFunc()	`//Función de evaluación de un Nodo`
minimax(args[])	`//Función de exploración con algoritmo`

Nodo	
nombre	//Cadena de texto que identifica a un nodo
estado	//Matriz que indica el estado que representa
vecinos	//Lista que contiene los nodos hijo
agregaVecino(n)	//Método que añade un nodo hijo
insertaVecino (n)	//Método que agrega nodo hijo al inicio de la lista
imprimeEstado ()	//Función retorna una representación visual del Nodo (estado)
muestraVecinos ()	//Función que muestra la lista de nodos hijo

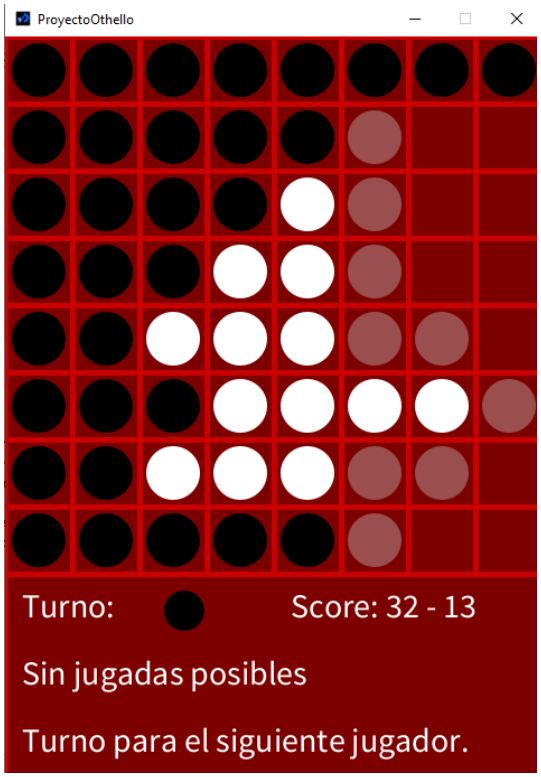
Como podemos ver, cada clase cuenta con funciones y métodos que les permiten interactuar entre sí, adicional a estas clases, se agregará una clase que se encargará de manejar el flujo del juego, dentro de esta clase principal se hará una instancia de una clase Tablero y una clase Jugador, las cuales emularán el entorno y el agente respectivamente.

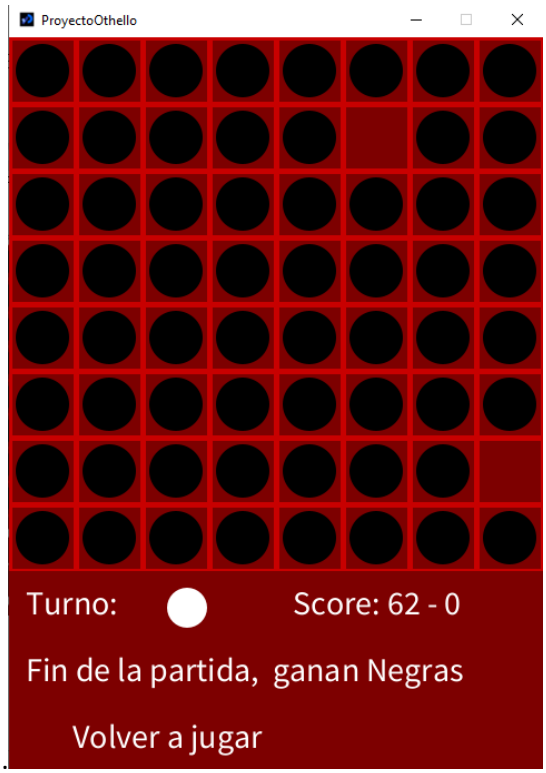
Interfaz gráfica del juego

Como siguiente paso, describiremos la interfaz gráfica del juego, dado que el oponente será un humano, este requiere de un entorno visual que le permita ver el estado actual del juego. La interfaz visual será la siguiente:



La interfaz gráfica del juego cuenta con el tablero de 8x8, representando las fichas con círculos negros para las fichas del agente y círculos blancos para las fichas del oponente, en la parte inferior contamos con un indicador del turno actual, un ajustador de dificultad del juego, el puntaje actual y el número de turno, así como que jugador realizó el movimiento anterior. Finalmente contamos con un espacio que muestra mensajes en caso de no haber movimientos disponibles para un jugador o bien cuando acaba una partida, dando la opción de iniciar una partida nueva





Poda alfa-beta

Experimentalmente se ha encontrado que el factor de bifurcación es variable, y aumenta considerablemente conforme aumenta el número de fichas en el tablero, al cabo de ciertos turnos el factor disminuye nuevamente debido a la disminución de casillas disponibles, sin embargo durante los turnos donde el factor de ramificación es más alto, la cantidad de nodos por explorar es sumamente alto, eso se refleja en mayor tiempo de procesamiento que imposibilita llevar a cabo una partida de manera fluida, es decir, se hace injugable, la solución es eficientar la exploración reduciendo la cantidad de ramas por explorar, una alternativa es utilizar la poda alfa-beta, en esencia, la poda alfa beta descarta una rama dado que su exploración no puede ofrecer un mejor resultado que las ramas ya exploradas. La manera de saber cuándo una rama no puede mejorar los resultados obtenidos en las ramas ya exploradas es siguiendo la lógica del algoritmo MINIMAX, supongamos que nos encontramos en un nivel MAX, buscamos el movimiento que nos

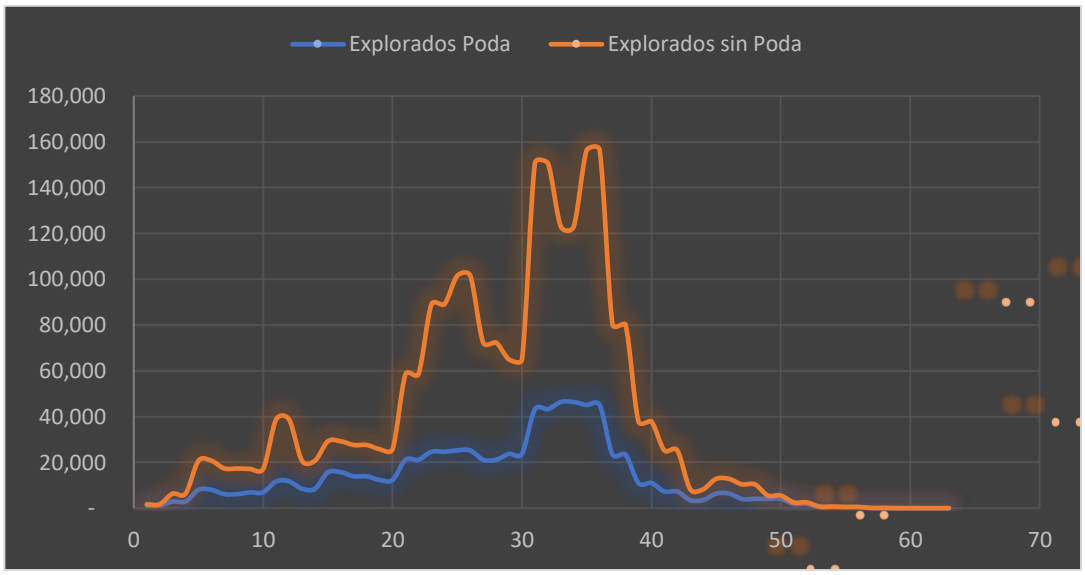
dé el valor más alto, supongamos ahora que hemos explorado algunas ramas y hasta el momento el valor más alto es A, al explorar el siguiente nodo, uno de sus hijos da un valor $A' < A$, dado que el nodo que exploramos se encuentra en un nivel MIN, el valor que tome el nodo explorado será menor o igual a A', por lo tanto el valor que obtenga el nodo explorado será menor que A, en tal caso no será una rama elegible por el agente debido a que dentro de las ramas ya exploradas existe una mejor solución. La ganancia está en que no fue necesario expandir los demás nodos hijos del nodo que nos encontrábamos explorando, ahorrando considerable tiempo y espacio al no expandir más ramas. Para implementar la mejora en la lógica del agente se deben hacer unas modificaciones ligeras al algoritmo MINIMAX, se deben añadir dos parámetros que servirán de cotas para los valores que pueden mejorar o el resultado, cuando un resultado no se encuentre dentro de las cotas se dirá que esa rama será podada y se omitirá la exploración de los siguientes nodos hijos, retornando así el valor más optimo obtenido para ese nodo (tal valor no aportará un camino más prometedor). A continuación, mostraremos el pseudocódigo de la función MINIMAX con la poda alfa-beta implementada:

```
int MINIMAX(Nodo, turno, profundidad, alfa, beta)
{
	alfaM = alfa
	betaM = beta
	If(profundidad >= limite || esNodoTerminal(Nodo))
		return evalFunc(Nodo, turno)
	For each hijo h de Nodo:
	{
		M[i++] = MINIMAX(h, !turno, profundidad + 1, alfaM, betaM)
		If(turno)
			alfaM = max(M)
		Else
			betaM = min(M)

		If(alfaM >= betaM)
			break;
	}
	return (turno ? max(M) : min(M))
}
```

Como podemos ver, la implementación de la poda no altera mucho la implementación original de MINIMAX, únicamente se añade un par de parámetros que delimitan el rango de valores que pueden mejorar las ramas ya exploradas, así como un bloque de instrucciones que actualizan estos valores conforme se profundiza la exploración y rompe la iteración de generación de hijos para realizar la poda.

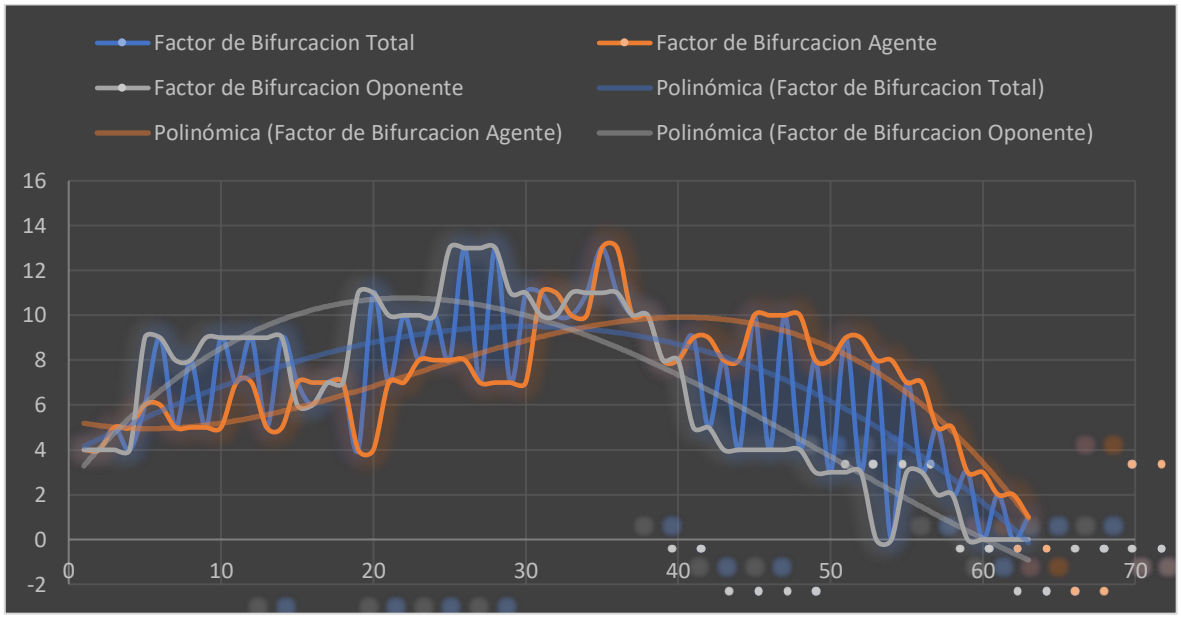
Una vez implementado, la mejora fue notoria, los tiempos de ejecución bajaron considerablemente, esto nos permitió aumentar la profundidad de exploración, mejorando así el rendimiento del agente. A continuación, veremos un comparativo del total de nodos explorados por el agente hasta una profundidad de 4 niveles cuando se utiliza y no se utiliza poda alfa-beta.



En el grafico podemos ver que en promedio, la cantidad de nodos explorados utilizando la poda alfa beta es de un 48% del total de nodos explorados en el caso en que no se utiliza la poda, sin embargo, a pesar de que solo se explora la mitad de nodos en total, puede apreciarse que en aquellos turnos donde el factor de ramificación es mayor (del turno 20 al 40) el promedio de nodos usando poda es tan solo del 32% del total de nodos explorados en caso de no usar poda, lo cual es de gran utilidad, pues justo en el momento en que la complejidad es mas alta, el rendimiento de la poda es el más optimo, reduciendo el tiempo de ejecución hasta en un 68%, es decir, el tiempo se reduce en casi una tercera parte.

Los datos anteriormente expuestos nos llevan a analizar la complejidad del árbol de búsqueda, tanto espacial como temporal, como podemos ver la curva de nodos explorados no es monótona, sin embargo, podemos verla como una curva que crece cada vez más rápido hasta llegar a un máximo que se encuentra entre los turnos 30 y 40, a partir de aquí comienza a decrecer rápidamente hasta llegar a cero (el momento donde termina la partida). Esto nos indica que el orden de la complejidad no es muy sencillo de aproximar, pues el factor de bifurcación varia

considerablemente en cada movimiento, experimentalmente se puede ver que sigue un orden cuadrático, sin embargo, si analizamos el factor de bifurcación de cada jugador por separado parece tener un comportamiento cúbico. En cuanto a la complejidad espacial, dar una aproximación es un poco más sencillo pues al tratarse de una búsqueda en profundidad esta será lineal de la forma $O(bn)$.



Es importante conocer que tan complejo es para el agente explorar el árbol dada una profundidad, a mayor profundidad el agente será capaz de evaluar con mayor certeza cada rama explorada, pues vera movimientos más lejanos, sin embargo, la complejidad del problema hará crecer rápidamente la cantidad de nodos por explorar, aun cuando se utilice la poda, por tal motivo es necesario encontrar un equilibro entre la profundidad y los recursos necesarios, maximizando el rendimiento del agente pero manteniendo un tiempo de ejecución moderado.

La profundidad máxima que soportó el agente durante las pruebas fue de 6 niveles, al explorar el árbol con un máximo de 6 niveles se llegaron a explorar hasta 1.8 Millones de nodos dentro de un tiempo moderado, los recursos computacionales fueron los siguientes:

- Procesador: Ryzen 5 5600G (6 núcleos, 12 hilos) 4.2 GHz
- Memoria RAM: 32 GB DDR4 3200Mhz

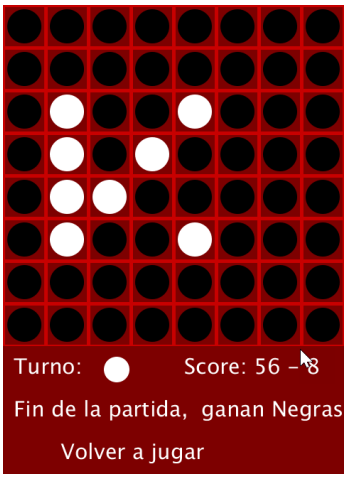
Si aumentamos un nivel mas de profundidad el tiempo máximo de exploración crece considerablemente y se pierde la fluidez del juego, sin embargo, 6 niveles de profundidad han sido suficientes para obtener muy buenos resultados por parte del agente. En la siguiente sección se mostrará el comportamiento del agente en partidas reales de Othello.

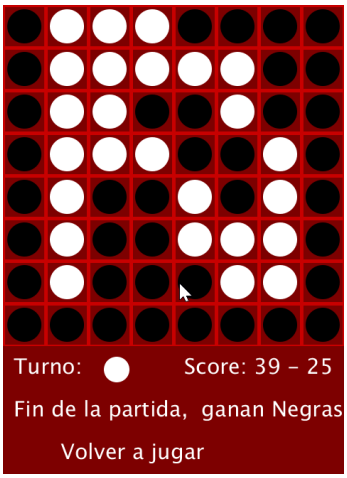
Finalmente, se han implementado 4 niveles de dificultad, Fácil, Medio, Difícil, Experto y SUPREMO, correspondientes a 2,3,4,5 y 6 niveles de profundidad de exploración.

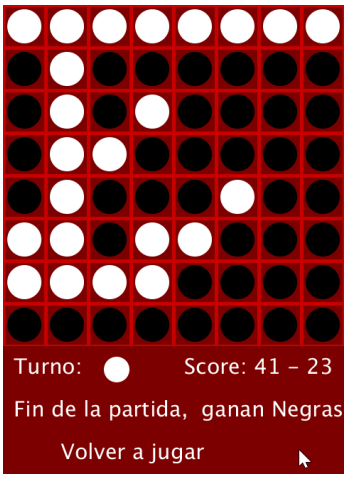
Resultados del Agente

Después de jugar varias partidas en la dificultad más alta (6 niveles de profundidad) el agente ha sido capaz de ganar en todas las ocasiones, incluso se ha puesto a prueba contra otros programas de Othello (Lite Games) y ha resultado ganador en diferentes ocasiones.

A continuación se muestran resultados de nuestro agente vs el agente de Lite Games en diferentes dificultades (normal, alta, muy alta) del agente oponente, mientras que nuestro agente explora 4 niveles de profundidad. Como podemos ver, en cada caso nuestro agente gano de manera contundente.







En general nuestro agente no solo no pierde con facilidad, sino que es sumamente difícil ganarle, se ha visto un rendimiento muy alto explorando a 4 niveles de profundidad, ofrece muy poco tiempo de procesamiento y una alta tasa de victorias, como vimos en la sección anterior, con 4 niveles de profundidad expande como máximo un valor cercano a los 160K nodos, un valor moderado y totalmente manejable por la computadora. Al expandir 6 niveles de profundidad el tiempo de ejecución aumenta considerablemente, y la tasa de victorias no aumento considerablemente en las pruebas realizadas, es necesario aumentar el volumen de oponentes, con estrategias diferentes, de ese modo podremos ver que ajustes pueden hacerse a la función de evaluación y si aumentar dos niveles de profundidad mejora considerablemente la tasa de victorias del agente.

Conclusiones

Con base en los análisis hechos a las estrategias conocidas para jugar Othello, se logro construir un agente capaz de potenciarlas y jugar de manera muy eficiente, después de múltiples pruebas, el agente gano todas las partidas y no le tomo mucho tiempo de procesamiento en realizar las exploraciones, dicho de otro modo, es muy bueno y rápido al momento de jugar Othello.

Para cambiar la dificultad del agente se ha reducido la profundidad de exploración, de manera muy notoria el agente disminuye su tasa de victorias, cuando el agente explora con una profundidad menor a 4, es muy posible ganarle al agente, sin embargo sigue ofreciendo un gran desafío para ganar, pudiendo ganar con facilidad a un jugador novato.

Para concluir, se construyo un agente capaz de jugar en varios niveles de dificultad que cumple óptimamente con sus objetivos, el tiempo de ejecución y los recursos consumidos se encuentran en un margen mínimo cuando se explora a 4 niveles de profundidad, llegando al límite de lo jugable cuando la profundidad aumenta a 6 niveles.

Bibliografía

García Serrano, A. (2016). *Inteligencia Artificial. Fundamentos, práctica y aplicaciones.* Ciudad de México: Alfaomega.