Estrategias Heurísticas para el Ajedrez

César Medina Aranibar

Escola Politècnica Superior d'Enginyeria de Vilanova i la Geltrú

Resumen

En este proyecto se hará el estudio, el análisis y el diseño de diferentes algoritmos de búsqueda heurística para el juego del ajedrez, entre los cuales se encuentran dos tipos de estrategia computacional.

Por un lado, la fuerza bruta, donde encontramos el algoritmo de Minimax, para el que se diseñan heurísticas adecuadas al problema (juego), y también la poda Alfa-Beta, una mejora del algoritmo Minimax. Por otro lado, tenemos el algoritmo de Monte Carlo Tree Search que incorpora la aleatoriedad y la estadística para encontrar la mejor jugada aunque puede que no encuentre la solución óptima.

Posteriormente se pasará a implementar los algoritmos en Python, para evaluar y comparar el grado de éxito de cada una de las soluciones planteadas. Se diseñarán un conjunto de pruebas para comparar las decisiones al jugar máquina contra máquina y máquina contra humano.

1. Introducción

Este artículo contiene una versión sintetizada de la memoria del Trabajo Final de Grado titulado "Estrategias heurísticas para el ajedrez". En la memoria se explican las reglas del juego del ajedrez de forma genérica. Asimismo, se repasan los antecedentes históricos del ajedrez en el mundo de los ordenadores hasta los motores de ajedrez actuales (*StockFish* [1], *AlphaZero* [2], *Leela Chess Zero* [3]). Este documento, por su brevedad, no los incluye.

Los algoritmos de búsqueda son uno de los mecanismos más utilizados por la inteligencia artificial para resolver problemas. El juego del ajedrez se puede representar como un árbol en el que cada nodo representa un estado concreto (conjunto de posiciones de las piezas en el tablero) y cada rama representa un movimiento posible. Lo que buscan los algoritmos de búsqueda en este árbol es un camino de la raíz a un nodo hoja que tenga el "mejor" valor posible (que nos haga ganar o acercarnos a ganar). Esa idea del mejor valor posible de un nodo se obtiene de la función de evaluación. Una función de evaluación le da al ordenador una visión estática de la partida de ajedrez. Sin embargo, los algoritmos de búsqueda pueden representar y analizar las posibles jugadas de cada uno de los jugadores a largo plazo. De esta manera se puede dotar a un motor de juego la capacidad de desarrollar estrategias.

El objetivo de este trabajo es estudiar, implementar y analizar diferentes algoritmos de búsqueda heurística que se pueden aplicar al juego del ajedrez siguiendo la regla de "No reinventes la rueda". El objetivo principal de este proyecto no es el de intentar superar a los motores de ajedrez que hay en la actualidad, ya que éstos cuentan con muchos recursos (GPU, TPU, memoria, equipo humano) para nada comparables con los disponibles para este trabajo.

Empezaremos estudiando dos algoritmos de fuerza bruta (Minimax y poda Alfa-Beta), que seleccionan la mejor jugada haciendo una búsqueda exhaustiva de todos los posibles movimientos que se pueden hacer; también compararemos las diferentes heurísticas para cada algoritmo. Luego estudiaremos un algoritmo probabilístico, el Monte Carlo Tree Search, que, comparado con los algoritmos de fuerza bruta, es más rápido ya que se basa en un método no determinista usado para aproximar expresiones matemáticas costosas y complejas. Sin embargo, la solución de dicho algoritmo es sólo probablemente correcta, ya que no asegura en su totalidad que encuentre la jugada óptima.

Después, implementaremos los algoritmos en Python. Para cada algoritmo se harán las pruebas necesarias individualmente y luego en conjunto, del algoritmo más básico al algoritmo más complejo. Cada algoritmo hará de jugador en un motor de juego y se programarán enfrentamientos entre jugadores de un mismo nivel, de niveles distintos y también con un jugador humano. El objetivo de los enfrentamientos es comprobar cómo de bueno es cada algoritmo y analizar, dentro de lo posible, las estrategias de juego que siguen.

2. Algoritmos de búsqueda

Algoritmo Minimax

Es un algoritmo muy utilizado para los juegos de suma cero, dentro de los cuales se encuentra el ajedrez, en los que siempre hay un equilibrio entre los puntos que gana uno de los jugadores con los puntos que pierde el contrincante. De esta manera se examina cada movimiento posible maximizando los puntos del jugador en su turno y minimizando los puntos en el turno del jugador rival. Con esto se puede deducir la estrategia del juego en la cual se supone que el rival siempre va a escoger la jugada que maximice sus puntos, que será la peor jugada posible para el jugador contrario.

Estas observaciones aplicadas al ajedrez se pueden resumir como: dentro de un árbol de búsqueda, se seleccionan los nodos de valor máximo en los niveles que correspondan cuando el turno es tuyo y los nodos de valor mínimo en los niveles que correspondan al turno del jugador rival. De esta manera, cuantos menos puntos sume el oponente, más beneficio habrá para el jugador en cuestión.

En la Figura 1 se muestra cómo está estructurado el árbol de búsqueda.

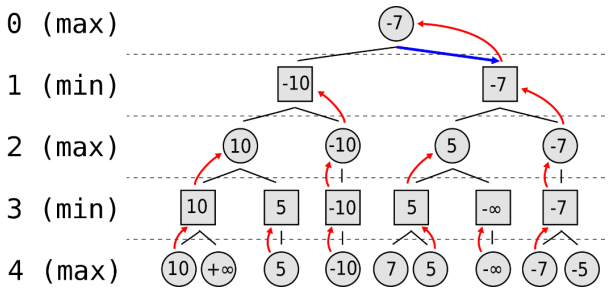

Figura. 1. Esquema general del árbol de búsqueda Minimax [6]

Un grave problema del algoritmo Minimax es que, dependiendo del juego como lo es en el caso del ajedrez, no es factible realizar el árbol completo de jugadas posibles, ya que el número de estados a explorar puede llegar a ser exponencial respecto al número de estas jugadas, llegando a tener un coste temporal exponencial, de tal forma que:

n = ramificación del árbol

m = nivel de profundidad

Coste temporal = $\Theta(n^m)$

Poda Alfa-Beta

El algoritmo poda Alfa-Beta es una técnica que reduce la cantidad de nodos que se tienen que explorar con el algoritmo Minimax, mediante la eliminación de determinadas ramas del árbol que se van construyendo y que se consideran inútiles para la decisión final.

La idea de este algoritmo se basa en comprobar, para cada nodo, su valor y el de su padre, lo que determina el valor de Alfa y Beta. Alfa es el valor del mejor movimiento que se ha encontrado hasta el momento actual en el que se encuentra un nodo Max; por otra parte, Beta es el valor del mejor movimiento actual que se ha encontrado para un nodo Min. Para cada nodo que se explora se comprueba si se pueden mejorar los valores de Alfa o Beta. En caso de ser cierto, se modificará Alfa o Beta dependiendo del nivel en el que se encuentre. Si en una rama determinada existe un valor que impide la mejora de un nodo superior, esa rama se acabará podando y de esta forma no se explorará dicha rama. Esto significa que el orden de exploración influirá en el coste temporal. Si, por ejemplo, se exploran primero los nodos donde sí habrá poda, esto mejorará significativamente el algoritmo; si, por el contrario, estos nodos se exploran los últimos, no habrá casi mejoría.

En la Figura 2 se muestra cómo está estructurado el árbol de búsqueda con poda Alfa-Beta.

Algoritmo Monte Carlo Tree Search

El algoritmo Monte Carlo Tree Search (MCTS), aplicado a un juego como el ajedrez, funciona de la siguiente manera. Desde una determinada posición dada, se generan un número elevado de simulaciones aleatorias (iteraciones), donde se busca el mejor movimiento posible para la siguiente jugada. Se guardan las estadísticas para cada movimiento simulado desde la posición inicial dada y se devuelve el mejor movimiento que será aquel que obtenga el mejor resultado.

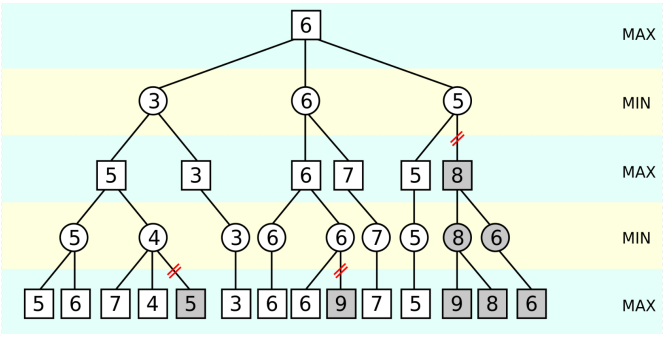

Figura. 2. *Árbol poda Alfa-Beta [7]*

El objetivo de este algoritmo es explorar el árbol de búsqueda parcialmente, de modo que, por cada nodo, calcule un valor heurístico que le permita elegir el que probablemente será el mejor. Ese valor heurístico se obtiene de un análisis estadístico.

Empezamos la exploración a partir de un nodo raíz y, por cada iteración que hagamos, iremos extendiendo el conocimiento que tenemos del espacio de estados. Cada iteración se divide en 4 fases: selección, expansión, simulación y retropropagación.

- **Selección**

La fase de selección se encarga de elegir qué nodo es el mejor candidato para ser expandido y para ello utilizaremos la fórmula UCT (*Upper Confidence for Trees*):

$$UCT(s) = \frac{Q(s)}{N(s)} + 2C_p\sqrt{\frac{\ln N(s_0)}{N(s)}}$$

Donde *N(s)* es el número de veces que el nodo hijo (*s*) ha sido visitado, *Q(s)* es la recompensa o puntuación total acumulada de todas las jugadas que han pasado por el nodo *s*, *Cp* > 0, es una constante que puede ser elegida para ajustar el nivel de exploración requerido, y $N(s_0)$ es el número de veces que el nodo s0, que es padre de s, ha sido visitado.

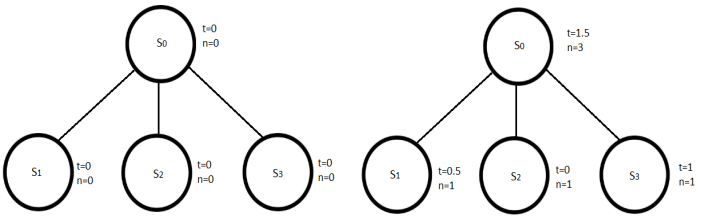

Figura. 3. *Ejemplo de árbol MCTS sin iteraciones (izq.) y con 3 iteraciones (der.)*

En la Figura 3 se muestra el árbol MCTS sin iteraciones (árbol de la izquierda), en el que todos los nodos tienen valor *t*=0 y *n*=0, siendo *t* el valor total del nodo y *n* el número de veces que se ha visitado ese nodo. El árbol de la derecha de la Figura 3, representa el momento en que ya se han hecho tres iteraciones (el nodo raíz tiene *n*=3) y se han almacenado los resultados correspondientes en cada nodo.

- **Expansión**

La fase de expansión se encarga de generar los nodos hijo del nodo actual en el que nos encontramos. En la Figura 4, vemos un ejemplo de cómo el nodo S_3 se ha expandido generando 3 nodos hijo (S_4, S_5 y S_6) con valores iniciales de *t* y *n* iguales a 0.

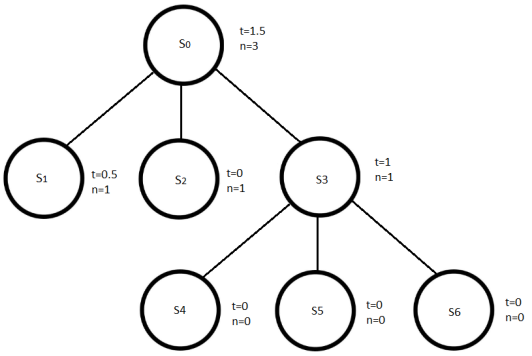

Figura 4. *Ejemplo de expansión árbol MCTS*

- **Simulación**

Tras la expansión, nos encontramos con la fase de simulación. En esta fase comprobamos qué tan bueno es cada uno de los nodos expandidos. Para conocer ese valor, empezamos de un nodo inicial y a partir de dicho nodo comenzaremos a elegir movimientos aleatorios hasta llegar a un nodo terminal, es decir, simular una partida entera de ajedrez utilizando movimientos aleatorios para cada jugador hasta acabar la partida. En algunas situaciones, como en el ajedrez, es más eficiente hacer una exploración en profundidad limitada con una función de evaluación o bien con tiempo limitado.

En la Figura 5, observamos que se ha seleccionado el nodo S_4 para hacer una simulación de una partida completa, en la que podemos ver que se ha perdido la partida y el valor que devuelve es 0.

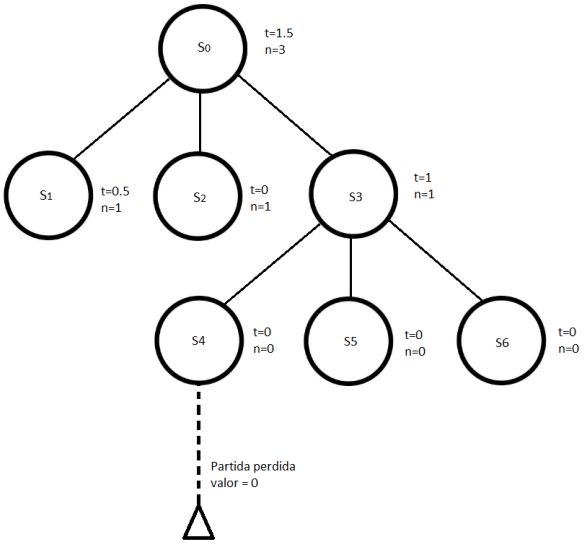

Figura 5. *Ejemplo de simulación de un árbol MCTS*

- **Retropropagación**

Se ejecuta cuando el estado final del juego ha sido alcanzado. Ahora se trata de llevar la información desde el nodo hoja actual, recorriendo todos los nodos antecesores, hasta llegar al nodo raíz sumando y actualizando los valores obtenidos en cada nodo que encontremos en el camino. Cuando se cumple esta tarea podemos concluir que se ha producido una iteración. En la Figura 6, donde los datos en color azul corresponden a los valores actualizados, vemos un ejemplo de ejecución de la fase de retropropagación.

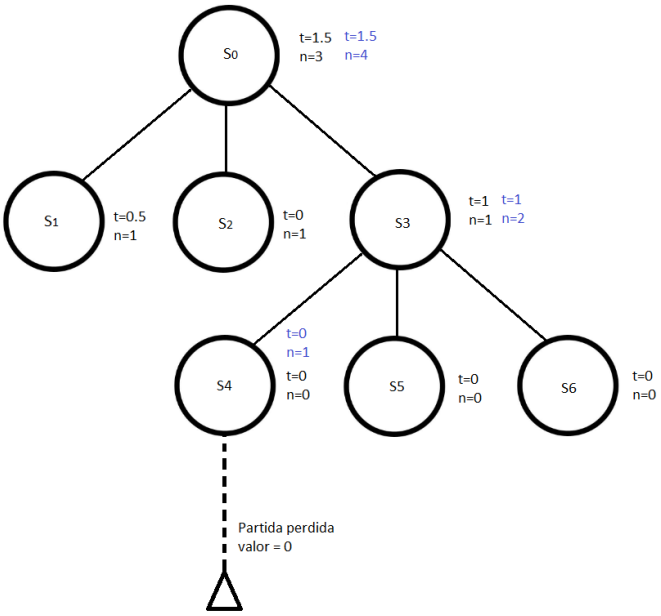

Figura 6. *Ejemplo de retropropagación en un árbol MCTS*

3. Desarrollo de los algoritmos

En la implementación de los algoritmos se ha utilizado una biblioteca de Python llamada `python-chess` [5]. Esta biblioteca nos brinda las herramientas para mostrar un tablero, con generador de movimientos y métodos para validarlos. La Figura 7 muestra una vista del tablero visualizado en Google Colab.

Figura 7. *Representación del tablero usando el módulo* `python-chess`

Se han implementado 6 jugadores distintos que aplican diferentes estrategias de juego:

- **Jugador Humano**

Función básica donde se le pide al usuario que introduzca un movimiento, y si ese movimiento es considerado válido lo devuelve, en caso de no serlo vuelve a pedir al usuario que introduzca otro movimiento.

- **Jugador Aleatorio**

Como primera aproximación a un jugador que no tiene ni idea de jugar, se ha programado un jugador capaz de hacer un movimiento aleatorio de entre todos los posibles movimientos disponibles que hay.

- **Jugador Básico**

Jugador "semi inteligente" capaz de usar una función de evaluación básica para elegir la mejor jugada [4]. Esta función, da un valor numérico a cada pieza que representa la importancia o el peso que tiene dicha pieza en el juego. La evaluación de una determinada posición se realiza de la siguiente manera:

1. Se hace la suma de los valores de todas las piezas propias a las que llamaremos peso_jugador1.
2. Se hace la suma de los valores de todas las piezas enemigas a las que llamaremos peso_jugador2.
3. Se resta peso_jugador2 a peso_jugador1 y, cuanto mayor sea el resultado del valor obtenido de esta resta, mayor será la probabilidad de que éste sea el valor de la mejor jugada.

- **Jugador Minimax**

El jugador Minimax representará un jugador "inteligente", en comparación con nuestro jugador básico. Implementa el algoritmo Minimax.

- **Jugador Poda Alfa-Beta**

El jugador Alfa_Beta es la mejora del jugador Minimax aplicando poda Alfa-Beta.

- **Jugador Monte Carlo**

El jugador Monte Carlo es el encargado de decidir el número de iteraciones que se hará en el árbol de nodos dado un tablero. El número de iteraciones es totalmente arbitrario dependiendo del hardware de la máquina en la que se use. Cuanto mayor sea este número, mejor será la jugada elegida, pero tardará más en pensar su jugada.

Para desarrollar el jugador montecarlo, necesitaremos hacer uso de una clase y una estructura de datos llamada `Nodo`, en la cual almacenaremos la información necesaria para que funcione de manera correcta y la dotaremos de un conjunto de métodos. Además se han implementado las funciones necesarias como el cálculo del valor de UCT de un nodo y las funciones correspondientes a las 4 fases del algoritmo: selección, expansión, simulación y retropropagación.

4. Pruebas y resultados

Jugador	Rival	V	D	Tablas	Partidas totales
Aleatorio	Humano	10	0	0	10
Aleatorio	Aleatorio	9	7	84	100
Básico	Humano	10	0	0	10
Básico	Aleatorio	3	0	27	30
Básico	Básico	0	2	48	50
Minimax	Humano	4	1	1	6
Minimax	Aleatorio	4	0	26	30
Minimax	Básico	5	0	45	50
Minimax	Minimax	0	1	11	12
Alfa-Beta	Humano	1	1	3	5
Alfa-Beta	Aleatorio	5	0	15	20
Alfa-Beta	Básico	7	1	22	30
Alfa-Beta	Minimax	2	0	28	30
Monte Carlo	Humano	1	0	2	3
Monte Carlo	Aleatorio	9	0	1	10
Monte Carlo	Básico	4	1	5	10
Monte Carlo	Minimax	0	1	9	10
Monte Carlo	Alfa-Beta	2	1	7	10

Tabla 1. *Tabla con todos los resultados de las partidas disputadas entre los jugadores*

En la Tabla 1 se detallan las pruebas realizadas, que consisten en jugar partidas entre los jugadores ya desarrollados y mostrar los resultados. Se pueden observar todos los resultados en los que el Jugador disputó contra el rival, dándonos estadísticas de las V=victorias, las D=derrotas, las tablas y el número total de partidas.

En la memoria se analizan los resultados y se describe con mayor detalle el progreso de cada estrategia en una partida concreta. Se razona el por qué, por ejemplo, la estrategia de Monte Carlo no siempre vence a las estrategias de sus contrincantes. También se comparan los movimientos de una u otra estrategia al principio de una partida, a la mitad del juego y al final de la partida.

5. Conclusiones y trabajo futuro

Mis objetivos al hacer este proyecto eran conocer un poco mejor los algoritmos de búsqueda, sobre todo en las estrategias que se utilizan computacionalmente para resolver un problema. Más concretamente, investigar las diferentes estrategias de algoritmos y aplicarlos al ajedrez, haciendo que se enfrentaran entre ellos y sacar conclusiones de cuál es mejor y en qué circunstancias. Tras finalizar el trabajo, considero que mis objetivos se han cumplido de forma satisfactoria.

En las pruebas realizadas entre el jugador Monte Carlo y el jugador poda Alfa-Beta, el segundo se ha demostrado superior, ya que el algoritmo Alfa-Beta necesita menos recursos a la hora de buscar las jugadas más prometedoras. Además, su función de evaluación le permite intentar

dominar el campo, eliminando a las piezas con más peso del enemigo, dejando solo al rey para después capturarlo. El problema más grave que tiene nuestro jugador Monte Carlo es la falta de recursos para poder aprovechar mejor su método estocástico no determinista pues, para que por estadística dé un mejor resultado, necesita de muchas más simulaciones de partidas, que en nuestro caso no hemos podido darle.

Como trabajo futuro de este proyecto se proponen dos líneas de mejora. Una está orientada al uso de redes neuronales, que hacen que en lugar de que el motor tenga que buscar las buenas jugadas siempre desde el principio, éste pueda ir aprendiendo y guardando las mejores jugadas que hace en sus partidas. Los mejores motores de ajedrez actuales (*StockFish NNUE*, *AlphaZero, Llela Chess Zero*), entrenan redes neuronales para calcular la calidad de un movimiento dado en un estado concreto del juego. Poder hacer uso de la función de evaluación de *StockFish*, por ejemplo, podría llevarnos a un nuevo nivel de evaluación de los posibles mejores movimientos. Otra línea de mejora se centra en aumentar la capacidad de nuestros recursos de hardware (trabajar con GPUs y TPUs) y alcanzar un mejor rendimiento de los algoritmos paralelizando su código o partes de este.

6. Agradecimientos

Quiero agradecer a mi tutora de proyecto Neus, por ayudarme y motivarme durante estos meses de proyecto.

A mi profesor de PROP Bernat, por darme la idea de estudiar los algoritmos de búsqueda para juegos.

A mi hermano por ayudarme y darme algunos consejos para seguir con el proyecto.

A mis amigos de clase por ayudarme año a año y compartir los buenos momentos conmigo.

Finalmente a mis padres por apoyarme y estar siempre ahí.

Referencias

[1] *StockFish chess* [En línea]. [Consulta: Abril del 2022]. Disponible en: https://stockfishchess.org/about/

[2] *AlphaZero*, la IA capaz de aprender ella misma a jugar al ajedrez y ganar a todas a las IAs adiestradas por humanos. [En línea]. [Consulta: Mayo del 2022]. Disponible en: https://www.xataka.com/robotica-e-ia/alphazero-ia-capaz-aprender-ella-a-jugar-al-ajedrez-ganar-a-todas-a-ias-adiestradas-humanos

[3] *Leela Chess Zero*. [En línea]. [Consulta: Mayo del 2022]. Disponible en: https://lczero.org/

[4] Bobby Fischer, Libro: *Bobby Fischer Teaches Chess*, Basic Systems Inc, Bantam Books, 1966.

[5] Python-chess, Python-Chess, [En línea, acceso: Junio de 2022].
Disponible en:
https://python-chess.readthedocs.io/en/latest/index.html

[6] Wikipedia, Algoritmo minimax, [En línea, acceso: Junio de 2022].
Disponible en: https://es.wikipedia.org/wiki/Minimax

[7] Wikipedia, Algoritmo poda alfa-beta. [En línea, acceso: Junio de 2022].
Disponible en:
https://es.wikipedia.org/wiki/Poda_alfa-beta