Minimax y alfa-beta en módulos de ajedrez

UNIVERSIDAD
COMPLUTENSE
MADRID

Doble grado de Ingeniería Informática y Matemáticas

Pablo Martín Huertas

26-05-2018

Índice

1. Historia del desarrollo de módulos de ajedrez

Una vez finalizada la segunda guerra mundial y con el éxito del desarrollo de los primeros ordenadores aparecerá por primera vez la formulación de como sería el funcionamiento de un primitivo módulo de ajedrez. En 1948 Alan Turing presenta su artículo "Faster than Thought" en colaboración David Champernowne. En este escrito se muestra una serie de pautas que serán necesarias para la programación de un módulo de ajedrez como por ejemplo la evaluación de piezas o la comprobación de jugadas legales. A pesar de que se muestra una idea general de cómo debería funcionar un módulo de ajedrez, no se indica ningún código concreto y no se hace referencia a la técnica que va a ser el núcleo de todo módulo de ajedrez: el algoritmo del minimax. Sin embargo, tan solo un año más tarde, el 9 de marzo de 1949 Claude Shannon un matemático e ingeniero eléctrico americano presentó un artículo al que nombró "Programming a Computer for playing Chess". En este escrito, mucho antes de que dicha programación se llevara a la práctica, ya se propuso la implementación de árbol de juego a partir del cual, mediante el uso de un algoritmo de minimax se conseguiría la toma de decisiones del programa.

En ese mismo artículo Claude ya advirtió que dicha aproximación mediante simple fuerza bruta no iba a poder ser fructífera y que sería necesario desarrollar una forma de elección más inteligente para ser capaz de abordar el problema del tamaño del árbol. Claramente el matemático estaba en lo cierto; ya que, ni siquiera a día de hoy con la avanzada potencia que tiene el hardware actual es posible que mediante fuerza bruta se pueda implementar un módulo razonable.

Ocho años más tarde en 1957 se propone por primera vez el empleo de técnicas de alfa-beta al recorrido del árbol de juego. Dicha implementación supuso un aumento de rendimiento tan inmenso que se presagió que en diez años sería posible vencer al campeón del mundo con un ordenador. Dicha predicción fue un fracaso total, de hecho, tomará mucho más tiempo conseguir ganar al campeón indiscutible. No fue hasta 1997, cuarenta años después que por primera vez un módulo sería capaz de tomarse la corona de campeón del mundo.
Dicho acontecimiento fue el famoso partido entre el campeón de la época el ruso Kasparov contra el superordenador Deep Blue desarrollado por IBM.

Este desarrollo de los hechos nos sugiere la pregunta: ¿Qué hizo falta duran-

te esos 40 años para poder llegar al nivel de juego que mostró Deep Blue? A lo largo de este trabajo vamos a estudiar una serie de mejoras al algoritmo de toma de decisiones que han sido vitales para conseguir el rendimiento de los módulos actuales. Esto nos da una perspectiva de lo increíblemente ineficiente que es la fuerza bruta y de la necesidad de generar algoritmos inteligentes que sean capaces de razonar con intuición tal y como lo haría un humano en su lugar.

Es por esta razón que la mayoría de mejoras que se van a proponer al algoritmo de minimax van a ser un intento de que el módulo intente comportarse tal y como lo hacen los jugadores de élite; es decir, centrándose en los movimientos y variantes más importantes ahorrándose así una inmensa búsqueda en variantes poco prósperas.

Para acabar esta introducción, cabe indicar que en la actualidad los módulos se encuentran muy por encima de los humanos. Simplemente un teléfono móvil es capaz de vencer al mejor jugador sin el más minimo problema. Esto es un logro realmente impresionante sobretodo en el apartado software. De hecho, se han realizado pruebas de enfrentar en idénticas condiciones de hardware al famoso Deep Blue con el módulo actual Stockfish en 100 partidas, y el resultado fue una victorial total por parte de Stockfish con todo victorias. Lo que claramente muestra el refinamiento de las técnicas empleadas hoy en día.

Figura 1: Partido entre Kasparov y Deep Blue

2. Base de un módulo de ajedrez

Esta sección pretende ser un breve repaso a cómo está realizada la implementación interna de los datos que va a usar el módulo. En nuestro caso nos centraremos más en los algoritmos, sin embargo para tener una mejor idea de cómo funcionan conviene conocer cómo son los datos para entender el tratamiento que se va a hacer de los mismos.
Una vez tengamos la estructura, acabaremos la sección explicando el funcionamiento del minimax que es la pieza fundamental que hará funcionar al módulo. En los apartados siguientes se concluirá con el refinamiento de dicho algoritmo de minimax mediante el empleo de diversas técnicas.

2.1. Estructura de los datos internos

Teniendo en cuenta que el árbol de movimientos necesitará una copia del tablero en cada uno de los nodos, es esencial una implementación eficiente del mismo debido a la enorme cantidad de ramas que se tendrán durante la ejecución. Principalmente existen dos tendencias a la hora de la implementación dependiendo de si se va a centrar como base las piezas en sí o las casillas.

En el método centrado en casillas cada nodo del arból tendrá una copia de un bitboard de 8x8. Cada una de las casillas vendrá dada por un char en el que usarán los bits para distinguir entre el color de la pieza y el tipo. Además hay algunos bits adicionales para comprobar si una pieza ha movido (importante para los peones) y también para saber si se ha hecho enroque en la partida.
Hoy en día debido a que la cantidad de memoría física no supone un problema (es la velocidad de cálculo la que hace cuello de botella), los tableros se representan también con char pero con un tamaño de 12x12. La ventaja que supone es que la necesaria comprobación de si un movimiento está en rango o no, se puede realizar despues de haber asignado la casilla y por tanto se ahorra tener que hacer la comprobación en algunos casos.

Por otro lado tenemos la centralización de las piezas. En este método no hay un tablero de chars, lo que guardamos es un array de palabras de 64 bits. Cada una de esas palabras es una pieza y guarda la información de su color, tipo, posición, si ha movido o no, ...

Objetivamente la centralización en las piezas debería ser el método más eficiente; sin embargo, como hemos indicado antes, el problema no se en-

cuentra en la memoria por lo que se suele optar por la opción de llevar un tablero ya que facilita sustancialmente el código.

Ejemplo de una posible implementación 12x12 (posición inicial)

FFFFFFFFFFFFFFFFFFFF	
FFFFFFFFFFFFFFFFFFFF	Bit 7 – Color (1 negro, 0 blanco)
FF04 02 03 05 06 03 02 04FF	Bit 4 – Flag Enroque
FF01 01 01 01 01 01 01 01FF	Bit 3 – Flag ha movido
FF00 00 00 00 00 00 00 00FF	Bits 2-0 Tipo de pieza
FF00 00 00 00 00 00 00 00FF	1 – Peón
FF00 00 00 00 00 00 00 60FF	2 – Caballo
FF00 00 00 00 00 00 00 00FF	3 – Alfil
FF81 81 81 81 81 81 81 81FF	4 – Torre
FF84 82 83 85 86 83 82 84FF	5 – Reina
FFFFFFFFFFFFFFFFFFFF	6 – Rey
FFFFFFFFFFFFFFFFFFFF	0 – vacío

2.2. Interfaz y estándar de comunicación

Una vez tenemos nuestros datos almacenados el siguiente paso será poder establecer un sistema de entrada y salida. Debido a la gran variedad de módulos que hay en la actualidad y al hecho de que tiene que ser posible que jueguen entre ellos, se ha desarrollado un estándar de comunicación común. Todos los módulos deben adaptarse a dicha interfaz para realizar la transmisión de datos después de que hayan realizado los cálculos oportunos.

El protocolo más usado en la actualidad es con diferencia el CECP: "Chess Engine Communication Protocol". También denominado xboard or WinBoard (dependiendo de su version para Unix o Windows). El módulo debe adaptarse a la forma en la que representa los datos el CECP y cuando la interfaz le pida un movimiento, se ha de realizar una transformación desde los datos del movimiento internamente a los datos que maneja el CECP. La gran ventaja de seguir el estándar es que todos los módulos se compatibilizan entre sí pudiendo competir sin tener que desarrollar programas intermedios adicionales.
Además el propio protocolo proporciona una interfaz de usuario GUI para que se puedan realizar las pruebas que se necesiten durante la depuración del programa.

2.3. Minimax

Tal y como presentó Shannon en su artículo de 1949, el minimax fue y sigue siendo la pieza central de todo módulo de ajedrez. La técnica consiste en los siguientes pasos:

1º - Se despliega el árbol de posibilidades con todos los movimientos legales en una posición concreta.

2º - A cada uno de los nodos del árbol se le atribuye una evaluación. Esto es: se calcula un número donde 0 indica igualdad, positivo indica ventaja blanca y negativo indica ventaja negra. Para entenderlo algo mejor, una primera aproximación por ejemplo sería hacer un recuento del material y dar ventaja al bando con mas piezas. (Más tarde veremos que la función de evaluación es mucho mas elaborada con numerosas heurísticas).

3º - Ahora llega el momento de tomar la decisión sobre qué jugada escoger. Supongamos que en el último nivel del árbol es el turno de las piezas blancas, en este caso de entre los nodos el módulo deberá escoger el que tenga una evaluación con un número positivo mayor. Sin embargo, cuando llegamos a los nodos del nivel anterior es el turno de las negras; así que, se ha de suponer que el jugador de negras escogerá la opción más fuerte para sí mismas (es decir, escogerá la evaluación más negativa). Se prosigue alternando entre el número positivo más grande y el número negativo más pequeño. Una vez llegados a la cima tendremos la variante que es considerada como más fuerte para el módulo según quién tiene el turno del nodo inicial.

Veamos un ejemplo ilustrativo del algoritmo en la práctica:

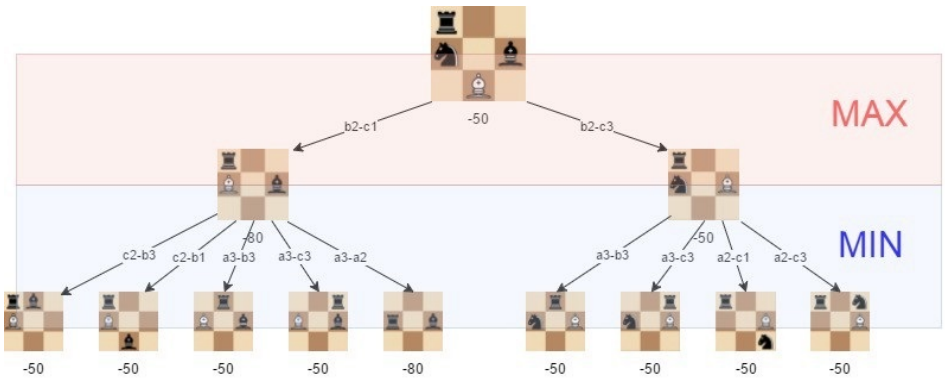

Figura 2: Algoritmo minimax

Primero razonemos tal y como lo haría un humano: Nos fijamos en que en la posición inicial es el turno de las piezas blancas, claramente los únicos movimientos legales son las dos capturas. En el caso de capturar el caballo observamos que perdemos el alfil ante la torre luego nos decantaríamos por capturar el alfil.

Ahora veamos cómo lo haría nuestro módulo: inicialmente desplegaría el árbol de posibles jugadas hasta una profundidad de dos. Luego en cada una de las hojas hace un recuento de material y asigna el valor correspondiente.

Figura 3: Valores de las piezas usados en el diagrama

Una vez calculadas las evaluaciones se identifica que en las hojas es el turno de las piezas negras, consecuentemente se escoge el valor mínimo como jugada candidata (En principio en caso de empate se escoge una aleatoriamente). Una vez tenemos nuestras decisiones pasamos las evaluaciones seleccionadas al nivel superior, en donde se ha de coger el máximo pues esta vez es el turno de las piezas blancas. De esta manera el ordenador jugará por la rama en la que se captura el alfil y sin embargo no puede ser recapturado por la torre.

Ejemplo de pseudocódigo para la implementación de minimax:

```
int maxi( int depth ) {
    if ( depth == 0 ) return evaluate();
    int max = -oo;
    for ( all moves ) {
        score = mini( depth - 1 );
        if( score > max )
            max = score;
    }
    return max;
}
int mini( int depth ) {
    if ( depth == 0 ) return -evaluate();
    int min = +oo;
    for ( all moves ) {
        score = maxi( depth - 1 );
        if( score < min )
            min = score;
    }
    return min;
}
```

2.3.1. Negamax

Tal y como hemos visto en el pseudo código anterior, la implementación del minimax requiere de dos funciones distintas que se vayan alternando entre sí. Para ahorrar código y facilitar la implementación es posible emplear la siguiente propiedad:

Dado un conjunto finito de números U

$$\text{máx}(U) = -\,\text{mín}(-U) \tag{1}$$

Donde $-U$ se define como la negación de cada uno de sus elementos. Aprovechándonos de esto podemos simplificar el pseudocódigo a una única función

```
int Negamax( int depth, enum sign ) {
    if ( depth == 0 ) return evaluate( sign );
    int max = -oo;
    for ( all moves ) {
        score = -Negamax( depth - 1, ++sign );
         if( score > max )
                max = score;
    }
    return max;
}
```

3. Algoritmos de recorrido eficiente del árbol

Una vez visto cómo nuestro módulo elige la variante mediante fuerza bruta usando el algoritmo de minimax, vamos a estudiar distintas formas para aumentar la eficiencia a la hora de recorrer el árbol de juego. Cómo vamos a ver, la clave se encuentra en intentar minimizar el número de nodos que tenemos que recorrer descartando aquellos que o no son lo suficientemente prósperos o ya tenemos otra variante mejor.

3.1. Alfa-beta

Los cortes alfa-beta son con diferencia la técnica por excelencia para obtener una mejora de rendimiento sustancial tal y como se dio a entender en la introducción. Para ilustrar su funcionamiento consideremos el siguiente ejemplo: a el módulo se le presenta una posición y le pedimos que dé una respuesta a profundidad 2. Inicialmente es el turno de las piezas blancas y hay 10 posibles movimientos. Supongamos que bajamos por el primero de los diez nodos y el algoritmo de minmax nos devuelve una evaluación de igualdad con un 0. Acto seguido vamos a analizar el segundo nodo, en este caso vemos que una de las posibles respuestas del negro al nodo dos es de capturar una de nuestras torres; es decir, suponiendo que la torre tiene un valor de 50 tendríamos asegurado al menos una evaluación -50. Consecuentemente podemos rechazar todas las otras posibles respuestas del negro porque aunque puede que exista una aún mas fuerte, el blanco ya va a elegir la variante del primer nodo sin considerar el segundo.

Resumiendo: una vez hemos encontrado un nodo con cierta puntación, esto nos condiciona la forma en la que vamos a analizar los siguientes. Ya que,

en los nodos siguientes siempre que encontremos una variante que es peor que la mejor hasta el momento, ésta se deshechará pues no merece la pena considerarla.

En comprobaciones de carácter práctico se aproxima que si un algoritmo debe recorrer una cantidad de nodos x. Entonces si se aplica la técnica de alfa-beta se obtiene un promedio de $\sqrt{x}$ nodos recorridos. La relación fue demostrada por Michael Levin en 1961. Dando lugar al que hoy en día se conoce como el **teorema de Levin:** [1]

Dado un árbol con una profundidad par n y supuesta b la cantidad de ramas que aparecen en cada nodo. Entonces el número de nodos terminales es:

$$T = b^n \tag{2}$$

Sin embargo, empleando la heurística de los cortes alfa-beta entonces el número de nodos a examinar es:

$$T = 2b^{n/2} - 1 \tag{3}$$

En nuestro pseudocódigo quedaría algo así:

```
int alphaBetaMax( int alpha, int beta, int depthleft ) {
   if ( depthleft == 0 ) return evaluate();
   for ( all moves) {
      score = alphaBetaMin( alpha, beta, depthleft - 1 );
      if( score >= beta )
         return beta;
      if( score > alpha )
         alpha = score;
   }
   return alpha;
}
```

[1] Para más información sobre el teorema véase la webgrafía y la bibliografía. Además en la webgrafía se ha indicado un enlace en el que se pueden ver datos empíricos de pruebas usando $n = 40$, $b = 40$

```
int alphaBetaMin( int alpha, int beta, int depthleft ) {
   if ( depthleft == 0 ) return -evaluate();
   for ( all moves) {
      score = alphaBetaMax( alpha, beta, depthleft - 1 );
      if( score <= alpha )
         return alpha;
      if( score < beta )
         beta = score;
   }
   return beta;
}
```

A continuación podemos ver en la siguiente figura un ejemplo de nodos cuyo recorrido nos ahorramos con la técnica descrita en el pseucódigo:

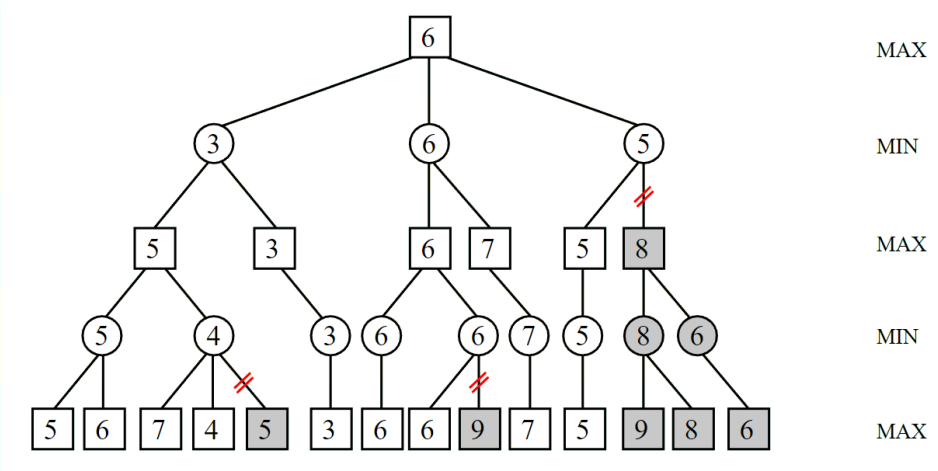

Figura 4: Cortes alfa-beta

3.2. Mejoras del algoritmo alfa-beta

En esta sección vamos a mostrar distintos refinamientos de los cortes alfa-beta. El factor más importante va a ser la observación de que el orden en el que se recorre el árbol importa. Es decir si encontramos una variante muy favorable al principio de la búsqueda, entonces los cortes alfa-beta van a ser abundantes pues dicha evaluación se usará para descartar otros nodos que forzosamente serán menos prósperos. Usaremos este hecho para intentar minimizar la cantidad de cálculos a realizar.

3.2.1. Tablas de transposición

En esta primera mejora nuestro objetivo va a ser evitar tener que expandir nodos que ya tienen solución. Por ejemplo dada la posición inicial del tablero mediante las secuencias de movimientos:

a) 1.d4 e6 2.c4 d5

b) 1.d4 d5 2.c4 e6

Se obtiene una posición idéntica, por lo que si calculamos una de las variantes la otra claramente tendrá la misma evaluación. Luego siempre que nos encontremos una posición que ya hemos cálculado mediante una secuencia de movimientos alternativa vamos a atribuirle la evaluación ya calculada.

Aquí es donde entra en juego las tablas de transposición. Se trata de guardar en memoria un hash (número único para cada una de las posiciones ya tratadas) junto con su evaluación. Así cuando lleguemos a otro nodo se calculará el nuevo hash y se comprobará si está ya en la tabla o no. De esta manera siempre que salte un acierto, podremos asignar la evaluación ya almacenada. Cabe destacar que una vez que se realiza una jugada en principio se ha de crear una tabla nueva pues los nodos de la tabla anterior tendrán una profundidad distinta.

3.2.2. Búsqueda de la variante principal

Ya se indicó que el orden en el que recorremos el árbol tiene un impacto en la cantidad de cortes que realizará la técnica de alfa-beta. Es justo esta característica la que vamos a aprovechar en esta mejora.

En la primera iteración del algoritmo del minimax no habrá ninguna alteración; sin embargo, se guardará la rama que nos ha proporcionado la evaluación definitiva. Una vez se realice la jugada y pasemos a la ejecución en el nuevo nodo se dará prioridad a la rama que guardamos en la iteración anterior. Puesto que dicha variante fue la más prolífica, es intuitivo pensar que cuando la profundidad baje 1 nodo más a partir de la nueva jugada, dicha rama seguirá siendo una de las candidatas a ser la óptima. Es por esta razón que una vez que recorramos esta variante en la nueva iteración tendremos una cota muy "fuerte" consiguiendo así que los cortes alfa-beta se apliquen en abundancia.

3.3. Búsqueda paralela

En la actualidad debido a las limitaciones hardware de velocidad que tienen los procesadores se ha optado por compensar a base de añadir múltiples núcleos. De esta manera para asegurar el máximo rendimiento de nuestro módulo será necesaria una estrategia de búsqueda en varios hilos de ejecución de manera simultánea.

Ahora bien tal y como hemos comentado en la sección anterior, hay mejoras como las de la búsqueda de la variante principal que nos gustaría conservar. Para ello, será necesario alguna forma de comunicación entre los distintos hilos.

Una de las posibles propuestas sería la siguiente: en el momento de realizar la búsqueda del siguiente movimiento inicialmente paramos todos los hilos. Acto seguido realizamos la búsqueda por la rama de la variante principal que hemos guardado. Una vez se ha obtenido esa evaluación (que se usará para los cortes alfa-Beta de cada uno de los restantes hilos) se lanza la búsqueda a las ramas abiertas pero con la información de dicha variante principal. Gráficamente sería algo así:

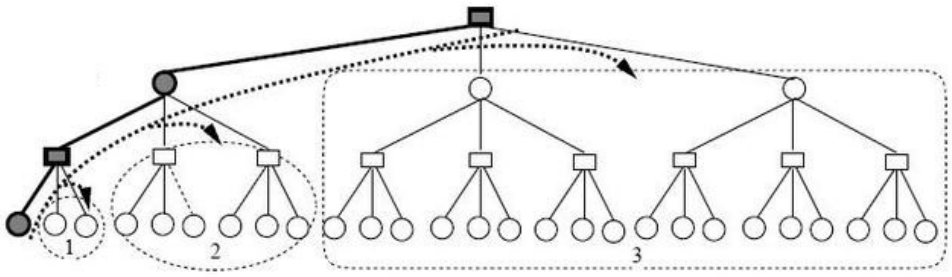

Figura 5: Los nodos negros son la variante principal. Una vez analizada se abren las otras ramas indicadas con flechas para hilos distintos.

4. Algoritmos de conocimiento

Hasta ahora hemos estado estudiando distintas formas de aumentar la eficiencia del recorrido del árbol. Pero existen una serie de técnicas que tiene un enfoque distinto. La clave ahora va estar en influenciar la toma

de decisiones mediante unos paramétros externos. Nuestro objetivo va a ser intentar simular la forma de razonamiento de los humanos. Por ejemplo: en el caso de la variante principal los humanos siempre vemos de manera intuitiva cuales son los movimientos razonables primero. Lo ideal sería que el ordenador pudiese hacer lo mismo; es decir, si nos ponemos en la técnica de la variante principal ¿Cómo sabe el módulo cual es la mejor rama candidata a dicha variante? Para ello vamos a intentar atribuir nociones estratégicas a nuestro módulo.

4.1. Generación de movimientos

Mientras se está desplegando el árbol de juego es necesario ir pieza por pieza evaluando sus movimientos legales. Sin embargo, esto no siempre es así y en ciertas circustancias se puede obtener un ahorro en los cálculos. Por ejemplo cuando un rey se encuentra en jaque, dicho jugador solo tiene tres posibilidades: capturar la pieza atacante, bloquearla con otra pieza o mover al rey a una casilla sin amenaza. Esto quiere decir que en la situación de jaque podemos implementar un algoritmo de generación de movimiento más específico sin tener que tener en cuenta todas las piezas.

Además algunos módulos actuales hacen una generación de movimiento en "paquetes". Es decir: primero generan los movimientos sin capturas, luego las capturas y finalmente los jaques. Sin embargo, no hay evidencia empírica que soporte que dicha diferenciación proporcione un aumento de rendimiento; más bien, se usa para facilitar la programación de otras técnicas.

4.2. Heurísticas de evaluación

En nuestra primera aproximación de la evaluación que dimos en el algoritmo del minimax hicimos un recuento de la cantidad de piezas de cada bando. Observando un poco más detenidamente es fácil darse cuenta de que esto es una evaluación realmente superficial. Existen muchos factores adicionales que los humanos usamos para dar un veredicto ante una posición concreta. Es por esta razón que el apartado de Heurísticas de evaluación es probablemente el más rico de todos en variedad de conceptos y usualmente en los TCEC (campeonatos mundiales de módulos) es el módulo con una mejor heurística el que sale vencedor. Veamos un par de ejemplos:

1º) Movilidad: Es razonable la idea de que una pieza centralizada debe ser más poderosa, esto se debe a que desde las casillas centrales su rango de

movimientos emana influencia hacia todas las direcciones del tablero. En ese caso diremos que la pieza es flexible ya que se puede relocalizar a cualquier otra parte del tablero sin problemas. Para reflejar este hecho en nuestros

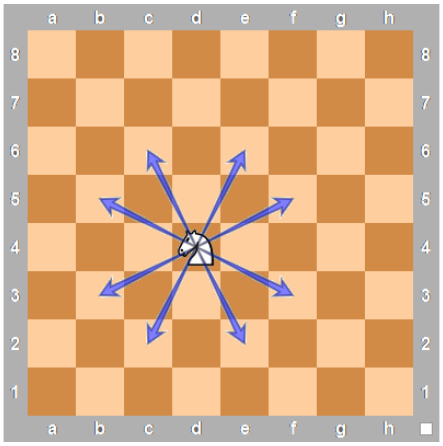

Figura 6: Flexibilidad de un caballo centralizado.

módulos vamos a alterar la función de evaluacion de la siguiente manera: A la hora de hacer el recuento vamos a tener en cuenta en qué casilla se sitúa cada pieza. En función de su localización se le añadirá en valor absoluto una cantidad prefijada. En la siguiente figura se puede ver un ejemplo de como la posición de ciertas piezas afectaría a su valor que ahora es relativo.

```
[ -3.0, -4.0, -4.0, -5.0, -5.0, -4.0, -4.0, -3.0],
[ -3.0, -4.0, -4.0, -5.0, -5.0, -4.0, -4.0, -3.0],
[ -3.0, -4.0, -4.0, -5.0, -5.0, -4.0, -4.0, -3.0],
[ -3.0, -4.0, -4.0, -5.0, -5.0, -4.0, -4.0, -3.0],
[ -2.0, -3.0, -3.0, -4.0, -4.0, -3.0, -3.0, -2.0],
[ -1.0, -2.0, -2.0, -2.0, -2.0, -2.0, -2.0, -1.0],
[  2.0,  2.0,  0.0,  0.0,  0.0,  0.0,  2.0,  2.0 ],
[  2.0,  3.0,  1.0,  0.0,  0.0,  1.0,  3.0,  2.0 ]
```

```
[-5.0, -4.0, -3.0, -3.0, -3.0, -3.0, -4.0, -5.0],
[-4.0, -2.0,  0.0,  0.0,  0.0,  0.0, -2.0, -4.0],
[-3.0,  0.0,  1.0,  1.5,  1.5,  1.0,  0.0, -3.0],
[-3.0,  0.5,  1.5,  2.0,  2.0,  1.5,  0.5, -3.0],
[-3.0,  0.0,  1.5,  2.0,  2.0,  1.5,  0.0, -3.0],
[-3.0,  0.5,  1.0,  1.5,  1.5,  1.0,  0.5, -3.0],
[-4.0, -2.0,  0.0,  0.5,  0.5,  0.0, -2.0, -4.0],
[-5.0, -4.0, -3.0, -3.0, -3.0, -3.0, -4.0, -5.0]
```

Figura 7: Variación del valor de distintas piezas según la casilla.

2º) Seguridad del rey: puesto que el fin de las partidas decisivas viene dado por un jaque mate, el rey es sin duda la pieza más valiosa de cada bando. Para reflejar esta idea en nuestro algoritmo se propone lo siguiente: primero

se localiza dónde está el rey, luego en la proximidad del mismo (por ejemplo casillas que estén a 4 pasos menos) pondremos constantes multiplicativas que darán más valor a las piezas del rival si se encuentran dentro de ese radio de influencia. De esta manera el algoritmo del minimax procurará mantener a su rey de las piezas del adversario. Un cambio adicional que se puede hacer a esta propuesta es dar más valor de base a un rey que se ha enrocado, de esta manera aumentará la frecuencia en la que el módulo de el enroque como jugada óptima.

4.3. Libro de aperturas

Finalmente vamos a ver la técnica de conocimiento estático por excelencia de los módulos. A lo largo de la historia del ajedrez se han ido probando distintas aperturas con más y menos éxito. Después de tantos años se ha conseguido identificar aquellas aperturas cuyo porcentaje de victorias es más alto en la élite mundial. Dichas aperturas usualmente son consideradas las más sólidas y han aguantado el paso del tiempo consiguiendo respuestas a todos los problemas que se pueden plantear. Este es un conocimiento que un jugador de ajedrez adquiere mientras va madurando en su nivel de juego.

Una vez más en nuestro afán de que los módulos de ajedrez intenten imitar al razonamiento humano vamos a introducir el concepto de base de datos de aperturas. Se trata de o bien con ayuda de un jugador profesional o bien haciendo pruebas empíricas en las que el módulo juega contra sí mismo, vamos a decidir una serie de movimientos iniciales. Dichos movimientos se realizarán sin necesidad de ningún tipo de cálculo. Hay que tener en cuenta que al comienzo de la partida al haber gran cantidad de piezas el número de posibilidades es sobrecogedor incluso para los ordenadores de hoy en día (es claro que a menor número de piezas la precisión con la que un ordenador toma decisiones es inigualable); es por esta razón que automatizar los primeros movimientos en analogía a como se juega las partidas de los jugadores de élite supone un gran aumento en el nivel del juego del módulo.

Por último, indicar que la implementación de una base de datos de aperturas se puede hacer con gran facilidad debido a la ECO (Encyclopedia of Chess Openings). Se trata de un banco de información online que clasifica a las aperturas en 5 secciones desde la A hasta la E en función de la similitud entre los mismos. Dentro de cada sección se atribuyen números a cada una de las subvariantes, por ejemplo la apertura española o apertura Ruy López, una de las más populares en la historia y todavía empleada abundantemente

al más alto nivel se le asigna el código C60.

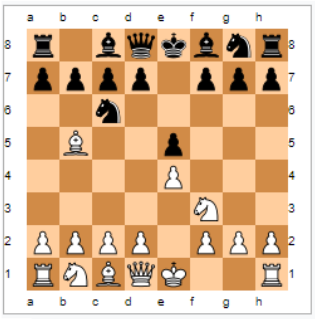

Figura 8: Apertura española: 1.e4 e5 2.Nf3 Nc6 3.Bb5

5. Conclusiones finales

A lo largo de este trabajo se ha realizado un repaso un tanto superficial del funcionamiento de los módulos de ajedrez debido a la elevada complejidad que poseen en la actualidad. A pesar de que hoy en día se usen técnicas bastante más sofisticadas que la presentadas aquí, se espera que se haya transmitido una idea general del funcionamiento básico de un módulo. También para más información se han indicado una serie de referencia en las últimas secciones.

No obstante, conviene destacar la idea clave que ha querido transmitir este trabajo y es que la fuerza bruta computacional es incapaz de abordar el problema que presenta un árbol de juego tan grande como el del ajedrez. El raciocinio humano tiene una herramienta muy potente: la intuición. Ésta nos permite descartar inmediatemente las jugadas que no merecen la pena calcular para poder así centrarnos en lo esencial. Una vez se presentó el algoritmo del minimax que conforma el núcleo de todo módulo, poco a poco se fueron estudiando una serie de técnicas cuyo objetivo es que el módulo sea capaz de mimetizar la forma de pensar de un humano. Desde los cortes alfa-beta con su descarte de variantes innecesarias por tener cotas mejores, hasta heurísticas como el grado de peligro que corre el rey en una configuración concreta del tablero. Todas las mejoras a nuestro algoritmo de minimax comparten el mismo objetivo común: otorgar al módulo la capacidad de decidir de manera *inteligente* la forma en la que emplea su poder de cálculo.

6. Webgrafía

https://docs.google.com/file/d/0B0xb4crOvCgTNmEtRXFBQUIxQWs/edit [2]

https://en.wikipedia.org/wiki/Claude_Shannon#Shannon's_computer_chess_program

https://en.wikipedia.org/wiki/Computer_chess

http://www-03.ibm.com/ibm/history/ibm100/us/en/icons/deepblue/

https://chessprogramming.wikispaces.com/Board%20Representation

https://chessprogramming.wikispaces.com/Chess%20Engine%20Communication%20Protocol

https://chessprogramming.wikispaces.com/Minimax

https://en.wikipedia.org/wiki/Minimax

https://medium.freecodecamp.org/simple-chess-ai-step-by-step-1d55a9266977

https://en.wikipedia.org/wiki/Alpha%E2%80%93beta_pruning

https://chessprogramming.wikispaces.com/Alpha-Beta[3]

https://chessprogramming.wikispaces.com/Transposition%20Table

https://chessprogramming.wikispaces.com/Principal%20Variation%20Search

https://www.ics.uci.edu/~eppstein/180a/990202b.html

https://chessprogramming.wikispaces.com/Parallel%20Search#ParallelAlphaBeta

https://chessprogramming.wikispaces.com/Move%20Generation

https://chessprogramming.wikispaces.com/Iterative%20Deepening

https://chessprogramming.wikispaces.com/King%20Safety

https://en.wikipedia.org/wiki/Encyclopaedia_of_Chess_Openings

https://chessprogramming.wikispaces.com/ECO

[2] Páginas escaneadas del artículo original de "Faster than Thought" escrito por Alan Turing en colaboración con David Champernowne.

[3] En este enlace se pueden encontrar los resultados de pruebas de los cortes alfa-beta en módulos de ajedrez. Estos se realizan con los parámetros $n = 40$, $b = 40$ y están en consonancia con el teorema de Levin.

7. Bibliografía

XXII. Programming a Computer for Playing Chess - Claude E. Shannon

Heuristic Evaluation Functions for General Game Playing - James Clune Department of Computer Science The University of California, Los Angeles

Michael Levin (1963). Primitive Recursion, AIM-055[4]

[4]Escrito sobre complejidad de algoritmos y el teorema de Levin