Breve introducción al desarrollo de motores de ajedrez

Francisco Javier Blázquez Martínez

Resumen

Trabajo realizado en el marco de la asignatura *Métodos algorítmicos en resolución de problemas*. Se introducen diversas técnicas para el desarrollo de motores de ajedrez centradas principalmente en los algoritmos de búsqueda y exploración en el árbol de posibles movimientos. He optado por no profundizar en los detalles técnicos de representación del tablero, movimientos y reglas del juego por mayor simplicidad.

1. Introducción

El ajedrez es sin lugar a dudas uno de los juegos más extendidos y con mayor impacto en nuestra sociedad. Es de sobra conocido por todos su alta complejidad y la dificultad de alcanzar un buen nivel de juego en este. Esto, junto con toda la historia que acarrea este deporte así como con la concepción social del juego, muchas veces tan relacionado con la inteligencia, concentración y el desarrollo cognitivo lo ha hecho objeto de estudio tantas veces hasta nuestros tiempos.

Es por tanto lógico que, con el auge de la computación y la aparición de nuevas ramas del conocimiento en este ámbito (como la *inteligencia artificial*) se tratara de aplicar los nuevos conocimientos y nuevas técnicas a la resolución de este juego. De hecho, al ajedrez se le ha llegado a llamar la *Drosophila Melanogaster* (mosca de la fruta) de la inteligencia artificial por la cantidad de estudios y experimentos sobre este juego surgidos en este ámbito. El término "resolución" ha sido empleado a drede pues, en contra de lo que mucha gente pueda pensar, no se conoce (tampoco por los motores de juego más potentes) una estrategia de juego perfecto, esto es, que garantice el mejor resultado posible partiendo de una posición dada cualquiera (resolución fuerte en el sentido de [2]) ni tampoco una estrategia que garantice la victoria para algún bando o las tablas a partir de la posición inicial (resolución débil en el sentido de [2]). Es más, también se desconoce si la aparente ventaja de las blancas en la posición incial permitiría que, con juego perfecto (independientemente de conocer la estrategia) se pudiera forzar siempre una victoria o un empate.

El hecho del que surge esta confusión, pensar que el ajedrez posee una estrategia de juego óptima que puede ser calculada por los ordenadores modernos, es la abismal superioridad con respecto al juego humano de estos en la actualidad. Es más, el desarrollo de motores de juego para el ajedrez en computadoras, en un principio antagonista al juego humano y rechazado por gran parte de la élite del ajedrez mundial, rápidamente con la mejora de los ordenadores personales se confirmó como la más útil herramienta para el aprendizaje y el estudio del ajedrez. Actualmente, los grandes jugadores cuentan como herramienta imprescindible de estudio y mejora de su juego con unas grandes bases de datos de aperturas (primeros movimientos de cada partida) y un potente motor de ajedrez que les permite analizar posiciones concretas y analizar sus errores en partidas ya disputadas. Esto se opone a las legiones de analistas que antiguamente rodeaban a los jugadores de primera fila mundial. Gracias a esto, el ajedrez humano de alto nivel se ha "democratizado", siendo posible mejorar enormemente en breves periodos de tiempo a cualquier jugador con buena capacidad de juego y motivación con la única ayuda de un sencillo motor de análisis. Esto se puede apreciar en el gran aumento del numero de jugadores con un ELO [8] mayor de 2000 puntos.

Es el propósito de este breve artículo introducir algunas técnicas (principalmente algorítmicas) que, junto con la gran capacidad de cómputo de los procesadores actuales hacen que los mejores jugadores de ajedrez del mundo actualmente no sean de carne y hueso. La mayoría de estas técnicas aparecen en [1] aunque de forma poco profunda (especialmente en cuanto a los esquemas algorítmicos se refiere). Esta ha sido la fuente principal de información de este artículo a partir de la cual otras muchasreferencias han completado los puntos que no quedaban totalmente claros o eran en exceso ambiguos. Para una introducción más profunda al terreno de los motores de ajedrez y su historia, desde sus inicios hasta la actualidad, donde el motor de Google AlphaZero desarrollado con técnicas distintas a los motores de ajedrez clásicos (basadas en inteligencia artificial y redes neuronales) venció al considerado mejor motor de ajedrez, Stockfish 10, pasando por el conocido caso de Deep Blue y la primera victoria a un vigente campeón del mundo recomiendo leer [4–7].

2. Fundamentos del desarrollo de motores de juego

Los motores de ajedrez requieren irremediablemente de una serie de componentes para funcionar. Tratamos aquí de introducirlos dando así al mismo tiempo una visión más global de la estructura de este documento.

El primero de ellos, como no podía ser de otra forma, es una representación del tablero y de las piezas, más aún que esto, una representación del estado del tablero. Hace falta una representación del tablero completa más alla del tablero y la colocación de sus piezas pues, una misma situación del tablero puede darse con el turno de movimiento para las blancas o para las negras. Otros factores que considerar son los derechos de enroque de ambos reyes (pueden ser distintos en posiciones con idéntica distribución) y el número de movimientos reversibles consecutivos por parte de ambos oponentes (pues llegar a 50 de estos movimientos

consecutivos implica las tablas). Esta es la parte más técnica del desarrollo del motor de juego, requiere tomar una serie de decisiones de desarrollo (principalmente las estructuras de datos que subyacen a esta representación) que son cruciales para posteriormente la ejecución de los movimientos y la exploración del árbol de juego.

El segundo de estos componentes es un mecanismo para detectar y poder ejecutar (de la forma más rápida posible, este punto será crítico para el rendimiento) los movimientos legales a partir de una posición dada, esto es, de su representación interna. Vemos así claramente que hay una dependencia absoluta con la representación del tablero elegida.

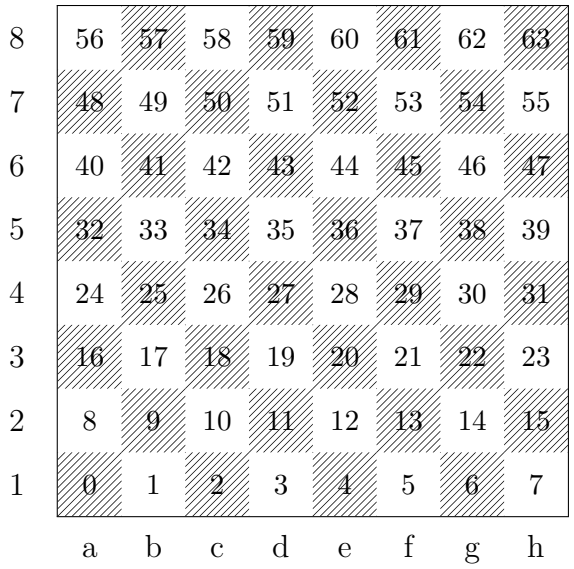
Partiendo de esta base, que constituye el cuerpo de todo motor de ajedrez (una representación interna que permite la ejecución sobre ella de cualquier partida de ajedrez con las reglas de juego actuales) tenemos que construir un sistema "inteligente", que sepa por sí mismo decidir que movimiento ejecutar ante cada posición del tablero. Surge así la necesidad de incorporar dos nuevos elementos a nuestro motor de ajedrez. Un sistema de evaluación que nos permita intuir qué posiciones son favorables para nosotros y qué posiciones son favorables para nuestro adversario y, teniendo esto, un mecanismo para explorar las posibles posiciones causadas por un movimiento. Esto último, un mecanismo de búsqueda en el árbol de posibles movimientos.

Entraremos ahora más en detalle en la representación del tablero, movimientos, evaluación de las posiciones y especialmente en técnicas de búsqueda. Todas las fuentes consultadas pueden verse en la bibliografía, pero he optado por destacar en este punto las empleadas en esta breve introducción para desarrolladores [10, 11]. Una gran fuente de ejemplos concretos de motores de ajedrez desarrollados y de código abierto puede consultarse en [28].

3. Representación del tablero

Como ya hemos dicho, las decisiones de implementación que tomemos en este punto son cruciales para el rendimiento final de nuestro motor de juego. De estas depende directamente el coste de analizar los movimientos posibles y ejecutarlos, la base de nuestro sistema de búsqueda del mejor movimiento. Otro factor a tener en cuenta es la cantidad de memoria de la que puede disponer nuestro motor durante la ejecución. Algunas estructuras presentadas fueron ampliamente usadas en la segunda mitad del siglo XX y posteriormente cayeron paulatinamente en desuso con el aumento de la capacidad de las memorias. Analizamos ahora una serie de estructuras de datos ideadas para la representación de un tablero de ajedrez. Para una información más detallada recomiendo consultar [12–14].

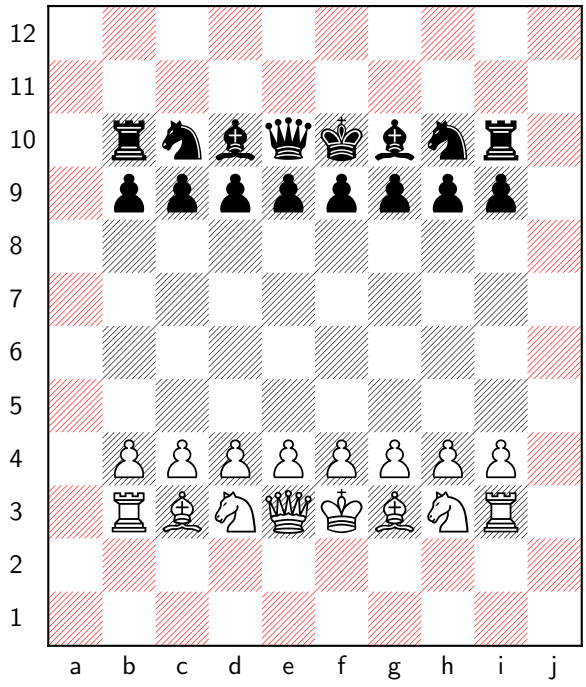
- **Tablero 8x8:**
  Primera aproximación surgida, consistente en la representación de la totalidad del tablero, las 64 casillas, indicando en cada una la presencia o ausencia de una pieza, el tipo de esta y su color. Generalmente se implementa con un array unidimensional de longitud 64 indexado como se ve en la figura.



Así, se puede conseguir una representación completa del tablero con 64B (un byte por casilla es suficiente para todas las posibilidades), necesitando además de esto únicamente almacenar derechos de enroque, derechos de comer al paso y turno de juego. Es fácil ver, sin embargo, que buscar todos los movimientos posibles para un jugador requiere recorrer todo el tablero y, para cada pieza de este jugador, explorar todas las posibles casillas de destino (viendo si están ocupadas por una pieza propia). Además, surgieron problemas con esta representación en la detección de movimientos que podían "salir" del tablero, movimientos a índices fuera de los contemplados.

- **Tablero 10x12:**
  Es fácil ver que con la indexación dada a las casillas del tablero en la estructura de datos del apartado anterior, moverse a la casilla superior es sumar ocho al índice, a la inferior restar ocho, a la casilla de la derecha es sumar uno (¡no siempre!) y a la de la izquierda es restar uno (¡no siempre!). Sin embargo, como se ve fácilmente, en los bordes del tablero surgen situaciones anómalas como sobrepasar el rango permitido del índice o hacer "saltos de tablero" de la columna derecha a la izquierda. Esto hacía muy costosas la comprobación de la legalidad de los movimientos generados y, por tanto, aumentaba el coste de la generación de movimientos, punto crítico para el rendimiento del motor. Se ideó entonces esta estructura que engloba el tablero con un margen de casillas marcadas como ilegales. Son necesarias las dos filas completas superiores e inferiores debido al movimiento de los caballos, que saltan siempre a casillas separadas por una fila o columna completa a su posición original.

  De esta forma se reducen los costes de comprobación de la legalidad de un movimiento y se mantienen los mismos costes de búsqueda de piezas, ejecución de movimiento... Sin embargo, juegue el bando que juegue, saber sus posibles movimientos implica buscar por todo el tablero sus piezas (técnica que se puede mejorar con un procesado por cada movimiento y respuesta o bien añadiendo estructuras de datos adicionales). Esto hizo que esta estructura cayera en desuso. La imagen de abajo muestra un esquema de la representación en tableros 10x12.



- **Lista de piezas:**
  Frente a las dos representaciónes anteriores, centradas en la representación del tablero completo, surge la idea de que representar las casillas vacías es innecesario y complica y aumenta el coste de las búsquedas. De esta forma diversos motores optan por la representación del tablero como listas de piezas (blancas y negras) con su posición asociada. Así, saber si determinada casilla está libre u ocupada tiene un coste mayor (pasa de ser constante a ser del orden del número de piezas que quedan en el tablero) pero, a cambio, el coste de la generación de los movimientos legales a partir de una determinada posición es menor. En cualquier caso, esta estructura de datos simplifica la representación del tablero pero complica también la implementación de búsqueda de movimientos, lo que hace que en la práctica la mejora del rendimiento sea nula.

- **Bitboards:**
  Con la aparición de los procesadores de 64 bits, surge una idea muy interesante para aplicar al desarrollo de los motores de ajedrez (así como de damas o cualquier juego sobre un tablero de 64 casillas). La idea es que, en una única palabra de 64 bits, puedo reflejar una determinada condición en todas las casillas del tablero. Esto permite que gran parte de las comprobaciones se reduzcan a operaciones lógicas soportadas por el procesador y, por tanto, ejecutables en un número muy bajo de ciclos.

  Por ejemplo, podemos tener en una sola palabra de memoria la situación de todos los peones blancos (1 en la posición i-ésima implica que hay un peón blanco en esa casilla). Es más, esta idea no vale únicamente para la representación de piezas, sino también para considerar casillas de ataque y procesarlas rápidamente. Si tuvieramos las casillas a las que pueden saltar los caballos negros, para saber qué peones blancos se pueden

comer los caballos negros simplemente tendríamos que hacer la AND lógica de los dos bitboards introducidos.

Existen otras muchas estructuras para la representación del estado del tablero de juego, sin embargo, muchas han caído en desuso. Actualmente las más usadas son los bitboards así como soluciones híbridas, que representan tanto el tablero entero como bitboards para las piezas individuales reduciendo al máximo los tiempos de comprobaciones a cambio de pagar los costes en tiempo de las actualizaciones de ambas representaciones. Otra representación poco usada internamente en los motores de ajedrez pero ampliamente usada para la depuración y las interfaces de usuario (es siempre común que todo motor de ajedrez permita conversiones de la representación interna a esta) es la notación FEN [9]. Para ver también como la codificación Huffman ayudaba en la primera etapa del desarrollo de estos motores a crear representaciones con un menor coste en memoria véase [14].

4. Generación de movimientos

Llegamos ahora a un punto clave de nuestro motor de ajedrez. La generación de movimientos así como la evaluación de la posición serán las tareas que más tiempo involucren (ejecutándose como parte de la búsqueda del mejor movimiento). En otros juegos como las damas o el Go, la generación de movimientos (viene a ser la exploración del árbol de posibilidades) es mucho más sencilla. En el ajedrez sin embargo las reglas del juego complican esto sobremanera. Como ya es sabido, no se permite que el rey del jugador que mueve quede en jaque tras el movimiento, cada pieza tiene unas reglas de movimiento distintas, la movilidad de las piezas depende también de las piezas a su alrededor, las capturas de peones al paso...

Además, recordamos un par de puntos bastante obvios. El primero es la total dependencia con las estructuras de datos elegidas para la representación del tablero. El segundo es la gran insistencia por parte de los desarrolladores de motores de ajedrez en que tanto esta parte, como la representación del tablero engloben a la perfección todas las reglas del ajedrez y estén libres de errores o *bugs*. Pese a la encapsulación de esta parte del código, es prácticamente imposible depurar comportamientos anómalos en la búsqueda del mejor movimiento causados por alguna carencia de las partes anteriores. Todo motor de ajedrez consta también de unos ejemplos de ejecuciones de movimientos que se ejecutan sobre la representación ideada y se contrastan con su resultado precalculado a modo de test.

Volviendo a la generación de movimientos propiamente dicha, cuando un humano juega al ajedrez identifica (generalmente ayudado de la experiencia y el instinto) unos pocos movimientos como factibles descartando desde el principio otros por abusrdos o carentes de sentido con el plan de juego. Esta fue la primera aproximación también para los computadores, sin embargo, ni tan siquiera la aproximación desarrollada en la unión soviética con ayuda del campeón del mundo Mikhail Botvinik desarrolló un juego de alto nivel.

Además, este sistema exigía costosas evaluaciones de la posición y de los movimientos para seleccionar cuales valía la pena explorar. Se llegó entonces a la conclusión (generalizada hoy en día) de que era necesario una búsqueda completa del árbol de posibilidades para que los motores de juego alcanzaran un nivel sobrehumano. Esto es, dada una posición, tendremos que analizar todos los posibles movimientos a partir de esta, sin descartar ninguno. Posteriormente en la búsqueda es posible que a niveles inferiores del árbol de búsqueda se puedan podar ciertas ramas de exploración. Sin embargo, a pesar de que tengamos que generar todos los posibles movimientos para obtener un buen nivel de juego, para minimizar el número de posiciones a explorar (normalmente con una variante del algoritmo $\alpha - \beta$ como se verá después) es común el uso de alguna función de rápida evaluación para ordenar estos movimientos de más a menos prometedores. Otras técnicas comunes son analizar primero las capturas de piezas y estas ordenadas de mayor a menor importancia de la pieza a capturar.

Para acelerar la generación de movimientos suele ser usual un preprocesado en el que se almacenan en bitboards las posiciones de ataque de cada pieza (o únicamente de las piezas de largo alcance) sin contar con las piezas que se puedan interponer en su camino, únicamente contando con los límites del tablero. Así, para el movimiento de las torres, alfiles y la reina los posibles movimientos se hallan con un recorrido lineal por las casillas preprocesadas hasta llegar al borde del tablero o a una pieza amiga o enemiga. La ventaja de estos mapas de ataque es que permiten la generación de movimientos de forma muy sencilla y su actualización es también rápida (en el caso de torres y alfiles, una de las dos direcciones de movimiento siempre permanecerá inalterada).

5. Evaluación de la posición

En el ajedrez, y esto se puede comprobar experimentalmente habiendo implementado el apartado anterior, en una posición normal el jugador con el turno dispone de unos treinta movimientos posibles. Además, estos habilitarán e imposibilitarán nuevos movimientos. En otras palabras, nuestro árbol de exploración va a ser exponencial (y con una base elevada). Además, ya aventurábamos en el apartado 2 que el ajedrez no es un juego que esté resuelto, no se conoce una estrategia que aplicar dada una posición para no meter la pata y darle ventaja al oponente. Nuestra única posibilidad es por tanto explorar las repercusiones de cada movimiento y quedarnos con el que nos permita obtener una cierta ventaja o, ceder lo mínimo posible ante nuestro oponente.

Esto sería muy fácil de desarrollar con algoritmos vistos en clase si el ajedrez fuera mucho menos profundo. Hemos visto por ejemplo que para hallar la mejor jugada en el juego de las n cerillas bastaba con generar el árbol de juego y aplicar el algoritmo min-max asignando un valor 1 a las hojas en las que resultábamos ganadores y un -1 a las hojas en las que vencía el oponente. Pues bien, en el ajedrez esto no es posible. No nos es posible debido a la naturaleza exponencial del árbol de búsqueda. Llegar a todas las hojas (finales de partida) a partir de una posición no final queda definitivamente lejos de la capacidad de cómputo actual.

Es por esto que se opta por truncar el árbol a partir de cierta profundidad (dependiendo de la plataforma y sus capacidades). La filosofía será la misma que en el algoritmo min-max pero elegiremos el movimiento que nos lleve a una mejor posición tras este número predeterminado de movimientos. Por contra, tenemos que definir entonces qué entendemos por una mejor o peor posición. El problema de evaluación consiste entonces de definir una función que para cada posible representación del tablero nos indique (de forma cuantitativa) lo favorable o desfavorable que es esta para cada jugador. Esto va a ser clave pues, nuestro motor de ajedrez va a tomar mejorar esta función como único objetivo. De hecho, el mismo motor (misma representación y mismo esquema de búsqueda) con una distinta función de evaluación podría desarrollar un juego totalmente distinto.

Ni que decir tiene que, para un juego tan antiguo y que tantas pasiones ha desatado como es el ajedrez, se han escrito ríos de tinta sobre evaluación de posiciones. Para los jugadores no principiantes es claro que el control del centro del tablero, la movilidad, coordinación, nivel de desarrollo de las piezas, cantidad de piezas de cada bando (cómo no), debilidades en las estructuras de peones, debilidades en la defensa del rey y otros muchos factores son de importante consideración en la evaluación de una posición. No es el objetivo de este trabajo discutir qué ponderación para todos estos factores sería la correcta. Si que entra dentro de nuestro objetivo sin embargo aclarar que la función de evaluación es la base para la toma de decisiones en la elección del movimiento de forma automática por parte de nuestro motor de ajedrez y, que debe aproximar lo más fielmente la realidad (si, por ejemplo, en nuestra función de evaluación los peones tuvieran un mayor peso que el rey, nuestro motor optaría por defender todos los peones en el tablero aún a costa de recibir jaque mate, lo que es absurdo). Otro error posible de las funciones de evaluación es olvidar que el sistema de juego de un motor de ajedrez (o al menos de los motores de ajedrez de alto nivel) es radicalmente distinto al juego humano. Hay posiciones de alta complejidad para un humano (con muchas piezas activas y muchos cambios posibles) que no deben ser penalizadas por nuestra función de evaluación pues, como hemos aclarado, el motor de juego analizará todas las posibles jugadas sin excepción y la penalización en tiempo, lógicamente, sera varios órdenes de magnitud menor que en el juego humano y con una probabilidad de error mucho menor. Complicaciones que suelen ser evitadas en determinadas circunstancias por jugadores humanos (también de alto nivel) aquí dejan de ser en absoluto indeseables.

Más información sobre esto puede verse en [18,19]. Una función de evaluación bastante eficaz y sencilla puede ser la siguiente (valores positivos significan ventaja blanca):

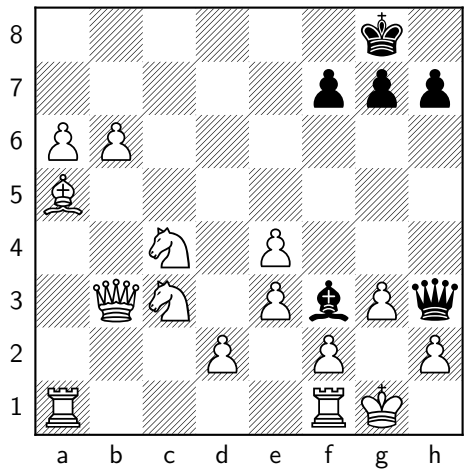
$$E() = 200(K - K') + 9(Q - Q') + 5(R - R') + 3(B - B' + N - N') + \\ 1(P - P') - 0{,}5(d - d' + b - b' + s - s') + 0{,}1(M - M')$$

K,Q,R,B,N,P = Número de reyes, reinas, torres, alfiles, caballos y peones respectivamente.
d,b,s = Número de peones doblados, bloqueados o solitarios.
M = Movilidad, número de movimientos legales.

6. Búsqueda

En este apartado vamos a ver el cerebro de nuestro motor. La base de su potencia de juego y, el punto donde definitivamente se decide si nuestro programa jugará a un nivel medio-alto o a un nivel muy superior al de cualquier humano.

Ya disponemos en este punto de una función de evaluación que, dada una posición, estima (de forma cuantitativa) qué jugador tiene ventaja. Pues bien, es claro que esta estimación la hace de forma estática, contando únicamente con la disposición actual del tablero. Se podrían entonces dar situaciones cómo la siguiente:



En esta posición nuestra función de evaluación (y cualquiera que se precie) da una ventaja a las piezas blancas. Es lógico, la diferencia de material es abismal. La función de evaluación anteriormente introducida da una ventaja de alrededor de 20 puntos. Sin embargo, el jaque mate de las piezas negras al rey blanco es, llegados a este punto, absolutamente ineludible. Este es un claro ejemplo de que únicamente con una evaluación estática del tablero nuestro motor nunca desarrollará un juego de alto nivel, caería en trampas de este tipo. Buscaría posiciones en las que maximizara la función de evaluación únicamente en ese instante, sin tener en cuenta las consecuencias de la posición en los siguientes movimientos. Además, este ejemplo ha sido claro porque las negras darán mate en la siguiente jugada, sin embargo, hay posiciones que ya están perdidas, esto es, que permiten una secuencia de movimientos forzados que lleva al mate en una mayor profundidad. Hay posiciones perdidas (o ganadas, que también queremos que vea nuestro motor) que se resuelven en 4,5,10 o un número mayor de movimientos. Es por esto que la búsqueda es necesaria para saber qué posiciones son deseables, mucho más allá de su función de evaluación. Para la búsqueda, como ya anticipábamos, se usa el algoritmo $\alpha - \beta$ (como en casi todo juego de información perfecta de dos oponentes) pero, con algunas técnicas para optimizar la búsqueda y que nuestro motor desarrolle un juego de alto nivel. Exponemos estas interesantes técnicas a continuación.

6.1. Mejoras del algoritmo $\alpha - \beta$

- **Tablas de transposiciones:**

  Sucede en el ajedrez que se puede llegar a una misma posición con distintos movimientos, además, incluso con distinto número de movimientos (transposiciones de una posición). Esto hace que una posición ya evaluada a lo largo de nuestra búsqueda puede que aparezca de nuevo en otras etapas o niveles de nuestra búsqueda. Es claro entonces que no es deseable tener que volver a emplear un valioso tiempo de CPU para analizar su árbol de exploración hasta una profundidad determinada para calcular algo que ya tuvimos en nuestro poder.

  La tabla de transposiciones surge precisamente para solucionar este problema. Es una estructura de datos basada en la idea de los mapas hash que almacenan cada posición analizada la mejor jugada calculada. Esto nos permite, a parte de evitar el cálculo en numerosas posiciones (en los finales con pocas piezas se puede podar hasta el 90 % del árbol de exploración) poder ganar profundidad en la búsqueda e incorporar técnicas de preprocesado, analizando movimientos y situaciones con anterioridad almacenando los resultados en la tabla de transposiciones. Esto último es principalmente útil para el análisis de aperturas.

  Para más información sobre esta estructura de datos y téncnicas para generar las claves hash a partir de una posición puede verse [13, 23].

- **Ordenación de los posibles movimientos:**

  Como ya sabemos, las podas que realiza el algoritmo $\alpha - \beta$ durante la búsqueda en el árbol de exploración dependen del orden de búsqueda seguido. Además, sabemos que analizar primero los mejores movimientos nos permite que el algoritmo termine analizando un número mucho menor de posiciones. Surge así la idea de combinar el algoritmo $\alpha - \beta$ con un preprocesado (de bajo coste algorítmico) que ordene los posibles movimientos mediante algún criterio para aumentar (en promedio) la eficacia del algoritmo. El problema que surge aquí es: ¿cómo realizar la ordenación?.

  Ante esto, la intuición nos plantea diversos enfoques. Uno de estos enfoques podría ser analizar primero las capturas pues, son las que tienen una mayor probabilidad de obtener una ventaja considerable (gracias a la diferencia de material). Otra posibilidad es aplicar la función de evaluación a las posiciones que resultan tras un movimiento en la posición inicial, ordenarlos, y buscar en este orden; sin embargo, esta técnica en ajedrez se ha mostrado poco efectiva. Una técnica que sí que se ha mostrado efectiva y es ampliamente utilizada es la conocida como heurística "*killer*". Esta consiste en probar movimientos que se han resultado útiles en posiciones similares (las ya analizadas en niveles superiores o las que se han probado efectivas en posiciones hermanas, del mismo nivel). Esto, claro está, requiere también de una organización en memoria para guardar estos movimientos y técnicas de búsqueda y referencia a esta estructura de datos. Más información sobre esta técnica puede verse en [24].

  Sin embargo, el enfoque óptimo es en absoluto intuitivo. Se presenta a continuación.

- **$\alpha - \beta$ con profundización iterativa:**

  Supongamos que queremos realizar una búsqueda dada una posición hasta una determinada profundidad k. Si tuviéramos los resultados de la búsqueda en todos los nodos a esta profundidad, podríamos ordenarlos y aplicar entonces el algoritmo $\alpha - \beta$. Sin embargo, es claro que antes de realizar la búsqueda va a ser imposible que dispongamos de los resultados de esta y, mucho menos, en todas las hojas del árbol de exploración (pues precisamente queremos podar lo máximo posible).

  La idea de esta técnica de profundización iterativa es que, para realizar una búsqueda hasta profundidad k, vamos a ejecutar una búsqueda hasta profundidad k-1 y a ordenar los resultados en base a esta para realizar la búsqueda hasta profundidad k. Decíamos que es absolutamente contraintuitivo pues para realizar la búsqueda hasta profundidad k, vamos a ejecutar este algoritmo para profundidades 1,2,...,k. ¿Cómo es posible entonces que afirmemos que aumentar el número de posiciones a explorar nos beneficie?.

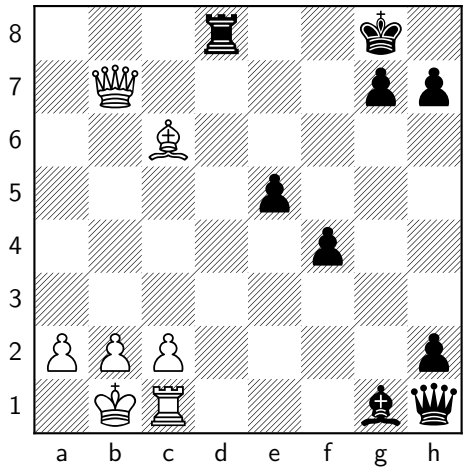
  Resulta que, debido a la naturaleza exponencial del árbol de exploración, el aumento del coste algorítmico en cuanto a realizar la búsqueda a profundidad k o, a profundidades k y k-1 es realmente bajo (la búsqueda a profundidad k-1 tiene un coste muchísimo menor que la búsqueda a profundidad k). Sin embargo, gracias disponer de la información de la búsqueda hasta una profundidad una unidad inferior, el algoritmo $\alpha - \beta$ para la profundidad dada k parte de una ordenación inicial que aumenta considerablemente el número de ramas de exploración podadas.

6.2. Mejoras del algoritmo de búsqueda

Con la aplicación del algoritmo $\alpha - \beta$ sobre la función de evaluación a una profundidad fija obtenemos una aproximación inicial y un motor de juego operativo aunque de un nivel medio (dependiendo también de las características de la plataforma). La tabla de trasposiciones y la técnica de la profundización iterativa son absolutamente necesarias para poder aumentar la profundidad y visión de nuestro motor y conseguir así un nivel de juego avanzado. Sin embargo, estas no son suficientes y, a pesar de todas las técnicas presentadas algunos problemas persisten. Presentamos estos y sus enfoques y técnicas de resolución.

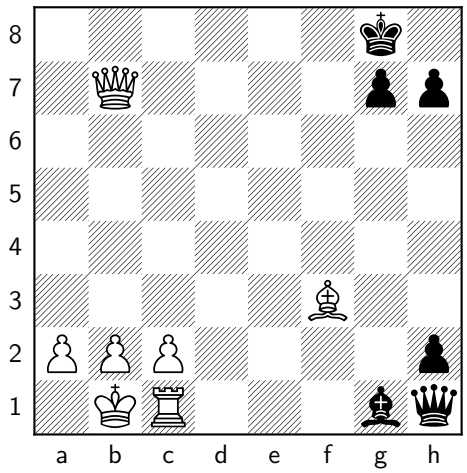
- **Efecto horizonte:**

  Se aplica este término para referirse a que, para nuestro motor de juego, nada existe y nada importa más allá de la profundidad de búsqueda que nosotros establezcamos. Ese es nuestro horizonte (ver [25]). Esto conlleva una serie de anomalías en la decisión del mejor movimiento. Para una mejor comprensión de este efecto vamos a analizar la posición incluida a continuación desde el enfoque de un motor de juego con profundidad de búsqueda de hasta 6 movimientos. Es el turno del jugador con las piezas negras.



En esta posición la reina negra está absolutamente atrapada y además está siendo amenazada por el alfil blanco. En conclusión, hagamos lo que hagamos, tarde o temprano, nuestra reina va a caer en manos del oponente. De hecho, en todas las variantes cae en 6 movimientos o menos (dentro de nuestro horizonte), en todas salvo una única jugada.

Podemos obstaculizar el alfil colocando nuestra torre en medio (regalamos la torre con Td5) y luego colocando a su vez en medio los peones de las columnas e y f en ese orden. En esta variante (..,Td5 Axd5,e4 Axe4,f3 Axf3) tras seis movimientos la posición sería la siguiente:



El motor de ajedrez en estas circunstancias puede pensar que es un genio, ha encontrado la única jugada que salva la reina. Para él, sin ninguna duda va a ser preferible entregar la torre y dos peones a cambio de salvar la dama (en base a las ponderaciones de la función de evaluación). Sin embargo no ha salvado la dama, ha pospuesto el inevitable final y ha regalado material a su rival, algo de lo que se dará cuenta tras sacrificar su torre cuando en el siguiente movimiento todas, absolutamente todas las líneas pierdan la dama dentro de su horizonte.

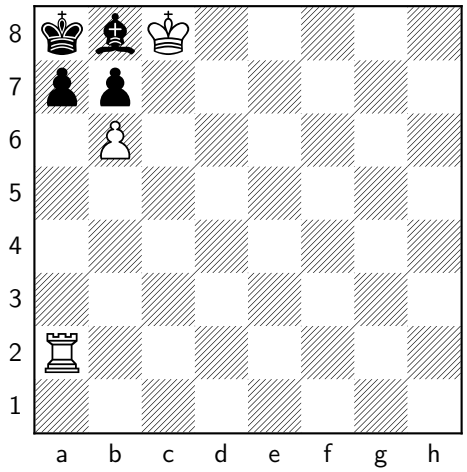
- **Búsqueda hasta inactividad:**

  Esta suele ser la forma de remediar el efecto horizonte anteriormente planteado. La técnica consiste en, al llegar a la profundidad límite de exploración de la búsqueda, analizar aquellos movimientos que se consideran drásticos (capturas, jaques y promociones de peones), movimientos que hacen variar de forma drástica la función de evaluación.

  Esto tiene un serio inconveniente. Tenemos que ampliar nuestro árbol de exploración unos niveles más allá de su profundidad fijada pues, para cada posición final (cuyo número es exponencial con la profundidad del árbol) vamos a ejecutar esta búsqueda reducida a sólo capturas y movimientos críticos. A pesar de esto, el número de capturas posibles es en general mucho menor al número de posibles movimientos y se va reduciendo conforme se van capturando las piezas.

- **Movimiento nulo:**

  Hay posiciones sumamente extrañas en ajedrez en las cuales, tener no la opción, sino la obligación de mover provoca consecuencias fatales. Una de estas posiciones, llamadas de *zugzwang* es la siguiente. Mueven blancas.



  En esta posición, las blancas pueden jugar Ta6!, situando su torre delante de los peones del oponente. Es fácil ver que, si pudieran ceder el turno, su fortaleza sería inexpugnable pero que al no ser esto posible, sus dos únicos movimientos pierden inmediatamente. Pues bien, como hemos dicho, ya visto el ejemplo, remarcamos que estas posiciones son sumamente extrañas. Por norma general, tener el turno de movimiento es una ventaja que, se supone, debe permitirnos aunque sea de forma muy sutil mejorar nuestra posición.

  Con la premisa anterior; ¿qué sucedería si cediendo el turno al oponente nuestro motor evalúa nuestra posición como favorable? ¡Conseguiríamos una poda-β sin realizar ningún movimiento!. Además, esto no implica añadir un gran coste a la búsqueda pues para ver cómo responder a un movimiento nulo hay que explorar un nivel menos en el árbol de búsqueda (al ser exponencial, este análisis extra tiene un coste bajo).

- **MTD(f):**

  Este algoritmo se puede ver como una evolución del algoritmo $\alpha - \beta$. Se ha confirmado como el más rápido y eficaz mecanismo para realizar la búsqueda en el juego del ajedrez (y en otros muchos). Veamos en que consiste.

  Supongamos que contamos antes de la ejecución del algoritmo $\alpha - \beta$ con una estimación del resultado que va a arrojar (como es el caso en el $\alpha - \beta$ con profundización iterativa). Podemos entonces aplicar este algoritmo considerando únicamente aquellas ramas del árbol de exploración que arrojen un resultado que no diste más de una cierta cantidad M de esta estimación. De esta forma, realizamos podas preventivas. Si al final de la ejecución el algoritmo arroja un valor estrictamente dentro de los márgenes, confirmamos nuestra estimación preliminar. Las podas preventivas realizadas nos han permitido llegar a esta estimación de forma más rápida. En caso contrario, cuando el algoritmo arroja un valor de los extremos permitidos, debemos repetir el proceso con un margen M mayor para garantizar un resultado.

  El algoritmo MTD(f) va incluso un paso más allá. ¿Qué sucedería si M tiende a cero? ¿Qué sucedería si M fuera cero? Tendríamos entonces un margen tan estrecho que las podas preventivas serían muy severas y el número de fallos (resultados extremos obtenidos) sería muy elevado. Estos fallos sin embargo, sí que podrían darnos una idea de si el valor real del resultado del algoritmo $\alpha - \beta$ es mayor o menor que la estimación en la que nos hemos centrado. Podemos conseguir así una búsuqeda binaria en los posibles valores de nuestra función de evaluación.

  Para aclararnos... ¿Cómo podemos aplicar esto? Nosotros tenemos una posición totalmente representada. A partir de esta consideramos los posibles movimientos de los que disponemos. El algoritmo $\alpha - \beta$ ejecutado a partir de estas posiciones nos daría el valor de cada posición contando con sus consecuencias hasta la profundidad prefijada. El algoritmo MTD(f) nos permitirá por contra estimar el valor de cada posición ejecutándolo repetidas veces con márgenes 0 centrado en distintas estimaciones. Así podemos realizar esto para cada posible movimiento y, con las estimaciones obtenidas (pueden ser arbitrariamente precisas) podemos elegir el mejor movimiento.

Todas estas técnicas son suficientes para que un motor de ajedrez desarrolle un juego muy superior al de cualquier humano en un ordenador personal. Destacamos aquí de nuevo la importancia de las tablas de transposiciones y la grandísima mejora que se experimenta al evitar la reevaluación de posiciones. Esto hace que la grandísima mayoría de motores de ajedrez actuales opten por una combinación MTD(f) + Tablas de transposiciones.

Otros factores que no hemos mencionado por su escaso interés algorítmico pese a su importancia en las competiciones es el tiempo. En las competiciones (también las de computadores) se dispone de un límite de tiempo a distribuir a lo largo de la partida. La profundidad de búsqueda suele ser así variable en función de este. No hemos creído conveniente profundizar en esto.

7. Conclusión

Como ya hemos mencionado, el desarrollo de un motor de ajedrez puede ser (y suele ser) planteado en base a una serie de etapas. La primera, representación del tablero y reglas de juego, sienta los pilares de todo nuestro motor. Es aquí imprescindible que el código sea libre de errores y elegir unas estructuras de datos adecuadas (generalmente bitboards). Tras esto ya deberíamos poder generar movimientos de forma sistemática dada una posición, lo que nos habilita poder aplicar los algoritmos de búsqueda. No obstante, esta búsqueda sería ciega sin una función de evaluación, que debe ser realista y, para mayor diversión y poder apreciar el cambio de juego de nuestro motor al modificarla, debe estar bien encapsulada.

Con todo esto, la ingente información de la que se dispone sobre este tema (algunos enlaces sumamente útiles en la bibliografía de este trabajo) y algunas referencias concretas de motores de código abierto que pueden verse en [28] creemos que es suficiente para introducirse al desarrollo de un motor propio de ajedrez. Esperamos que sea de ayuda.

Referencias

[1] *Chess Programming Wiki*. Versión del 15 de abril de 2019. Disponible en:
https://www.chessprogramming.org/Main_Page

[2] Schaeffer, Burch, Björnsson, Kishimoto Müller, Lake, Lu, Sutphen. (2007). "Checkers is solved". *Revista Science* 317, (5844), 1518-1522. Artículo disponible en:
https://science.sciencemag.org/content/317/5844/1518/tab-pdf

[3] Silver, Hubert, Schrittwieser, Antonoglou , Lai, Guez, Lanctot , Sifre, Kumaran, Graepel, Lillicrap, Simonyan, Hassabis. (2018). "A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play". *Revista Science* 362 (6419), 1140-1144. Artículo disponible en:
https://science.sciencemag.org/content/362/6419/1140/tab-pdf

[4] *Computer chess*. Versión del 17 de abril de 2019. Disponible en:
https://en.wikipedia.org/wiki/Computer_chess

[5] *History of computer chess*. Versión del 17 de abril de 2019. Disponible en:
https://www.chessprogramming.org/History

[6] *Deep Blue*. Versión del 17 de abril de 2019. Disponible en:
https://en.wikipedia.org/wiki/Deep_Blue_(chess_computer)
https://en.wikipedia.org/wiki/Deep_Blue_versus_Garry_Kasparov

[7] *AlphaZero*. Versión del 17 de abril de 2019. Disponible en:
https://en.wikipedia.org/wiki/AlphaZero
https://en.wikipedia.org/wiki/Leela_Chess_Zero

[8] *Sistema de puntuación ELO*. Versión del 17 de abril de 2019. Disponible en:
https://es.wikipedia.org/wiki/Sistema_de_puntuacion_Elo

[9] *Forsyth–Edwards Notation*. Versión del 17 de abril de 2019. Disponible en:
https://en.wikipedia.org/wiki/Forsyth-Edwards_Notation

[10] *Getting Started in chess programming*. Versión del 18 de abril de 2019. Disponible en:
https://www.chessprogramming.org/Getting_Started

[11] *Chess Programming Part I: Getting Started*. Versión del 18 de abril de 2019. Disponible:
http://archive.gamedev.net/archive/reference/articles/article1014.html

[12] *Board representation*. Versión del 17 de abril de 2019. Disponible en:
https://www.chessprogramming.org/Board_Representation

[13] *Chess Programming Part II: Data Structures*. Versión 18 de abril de 2019. Disponible:
https://www.gamedev.net/articles/programming/artificial-intelligence/chess-programming-part-ii-data-structures-r1046

[14] *Board representation (chess)*. Versión 18 de abril de 2019. Disponible en: https://en.wikipedia.org/wiki/Board_representation_(chess)#Huffman

[15] *Data structures for chess programs*. Jean Goulet. McHill University. Disponible en: http://digitool.library.mcgill.ca/R/?func=dbin-jump-full&object_id=65427

[16] *Move generation*. Versión del 17 de abril de 2019. Disponible en: https://www.chessprogramming.org/Move_Generation

[17] *Chess Programming Part III: Move Generation*. Versión del 17 de abril de 2019. https://www.gamedev.net/articles/programming/artificial-intelligence/chess-programming-part-iii-move-generation-r1126

[18] *Evaluation*. Versión del 17 de abril de 2019. Disponible en: https://www.chessprogramming.org/Evaluation#Basic_Evaluation_Features

[19] *Chess Programming Part VI: Evaluation Functions*. Versión del 17 de abril de 2019. https://www.gamedev.net/articles/programming/artificial-intelligence/chess-programming-part-vi-evaluation-functions-r1208

[20] *Search*. Versión del 17 de abril de 2019. Disponible en: https://www.chessprogramming.org/Search

[21] *Chess Programming Part IV: Basic Search*. Versión del 17 de abril de 2019. Disponible: https://www.gamedev.net/articles/programming/artificial-intelligence/chess-programming-part-iv-basic-search-r1171

[22] *Chess Programming Part V: Advanced Search*. Versión del 17 de abril de 2019. https://www.gamedev.net/articles/programming/artificial-intelligence/chess-programming-part-v-advanced-search-r1197

[23] *Transposition table*. Versión del 17 de abril de 2019. https://www.chessprogramming.org/Transposition_Table

[24] *Killer Heuristic*. Versión del 17 de abril de 2019. https://www.chessprogramming.org/Killer_Heuristic

[25] *Horizon effect*. Versión del 17 de abril de 2019. https://en.wikipedia.org/wiki/Horizon_effect

[26] *MTD(f), A Minimax Algorithm faster than NegaScout*. Versión del 17 de abril de 2019. http://people.csail.mit.edu/plaat/mtdf.html

[27] *MTD-f*. Versión del 17 de abril de 2019. https://en.wikipedia.org/wiki/MTD-f

[28] *Open source chess engines*. Versión del 17 de abril de 2019. Disponible en: https://www.chessprogramming.org/Category:Open_Source

[29] *Stockfish*. Versión del 18 de abril de 2019. Disponible en: https://www.chessprogramming.org/Stockfish