FACULTAD DE MATEMÁTICAS

DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN
E INTELIGENCIA ARTIFICIAL

FUNDAMENTOS DE IA PARA EL AJEDREZ

Autor: Alberto Portillo Raya.
albporray@alum.us.es

Tutores:
Francisco Félix Lara Martín
Fernando Sancho Caparrini

Agradecimientos.

Es complejo resumir en pocas frases la cantidad de emociones que llega a sentir una persona que está a punto de terminar una carrera universitaria. Especialmente si el camino ha sido largo y accidentado. Sólo puedo dar las gracias a aquellas personas que me han acompañado en todo el proceso.

A mis tutores, por darme la oportunidad de hacer un TFG atípico y dar forma a las muchas ideas generadas en el proceso. A mi familia por, pese a no comprender qué estaba pasando siempre o qué necesitaba, apoyarme hasta el final. A mis amigos, por esos abrazos al final de cada derrota y la celebración de cada victoria. A Lucía, por ser mi compañera de vida.

Por último, al profesorado que he encontrado durante mi educación, les dejo las sabias palabras de un gran maestro.

Pass on what you have learned. Strength. Mastery. But weakness, folly, failure also. Yes, failure most of all. The greatest teacher, failure is. [...] We are what they grow beyond. That is the true burden of all masters.

— Yoda a Luke Skywalker.

A mi abuelo.

Índice general

Introducción.

El presente trabajo tiene como objetivo la introducción y desarrollo de los fundamentos teóricos de algunas técnicas clásicas y actuales de la IA aplicadas a la generación de jugadores automáticos de Ajedrez.

Se aborda en el primer capítulo el desarrollo histórico de esta disciplina, abarcando desde los primeros jugadores automáticos de Turing y Shannon [24] hasta los actuales Stockfish y AlphaZero [25], pasando por el creciente desarrollo de mejora del rendimiento de los jugadores automáticos hasta alcanzar la victoria frente al campeón del mundo humano de ajedrez, como fue el caso de IBM Deep Blue. Mostraremos también los problemas asociados al objetivo de poder *evaluar* de manera correcta una posición para obtener la mejor jugada posible, y daremos una primera aproximación de las soluciones parciales encontradas.

En el segundo capítulo, desarrollamos los conceptos básicos sobre juegos finitos que son base para todo el trabajo realizado en los capítulos 3 y 4. Además, describimos las hipótesis sobre los juegos que vamos a considerar y damos un resultado sobre la determinación de los mismos.

Situados en el contexto en que se ha desarrollado toda una línea de trabajo en busca de algoritmos cada vez más eficaces, hemos decidido enfocar este trabajo en el estudio de algunos de los más importantes: el algoritmo Minimax con poda α-β en el capítulo 3, y el Árbol de Búsqueda Monte Carlo en el capítulo 4.

Ambos capítulos tratan de manera general los algoritmos y desarrollan los conceptos que fundamentan el buen rendimiento de los mismos. Se realiza, además, un análisis de sus virtudes y debilidades. En el capítulo 3 se trata con profundidad la técnica de reducción del número de jugadas que analizar, mientras que el capítulo 4 contrasta con el anterior en la filosofía de búsqueda y evaluación de las distintas jugadas posibles.

Tras los resultados aportados en estos dos capítulos, el capítulo 5 particulariza todos los conceptos anteriores al juego del Ajedrez. Damos ejemplos prácticos donde mostramos las virtudes y debilidades de los distintos algoritmos, así como la forma en la que los algoritmos evalúan las posiciones y realizan las búsquedas de jugadas.

La particularización al ajedrez realizada en el capítulo 5 nos da paso de manera natural a la implementación de los algoritmos estudiados en los capítulos 3 y 4. La implementación es realizada en el lenguaje Python si bien es posible desarrollarla en muchos otros lenguajes de programación.

Por último, realizamos un análisis global del trabajo realizado y estudiamos una posible línea de trabajo futuro que continuaría de manera natural hacia las técnicas recientes de Deep Learning y el trabajo actual en jugadores automáticos de ajedrez.

Palabras clave: Minimax, Monte Carlo Tree Search, computer chess.

Summary

The present work has a clear goal: the introduction and development of the theoretical basis and some of the classical and modern techniques in AI applied to the generation of computer chess players.

We start with an historical development of this discipline from Turing and Shannon's first computer chess players [24] to recent computer chess players such as Stockfish or AlphaZero [25] with a mandatory stop on IBM Deep Blue, the first computer chess player that won a match against the World Chess Champion Garry Kasparov. Problems like accurate evaluation are shown and we give a glimpse of some partial solutions.

During the second chapter, we present basic concepts about finite games. This is a basic chapter that is necessary to understand all the work done in chapters 3 and 4. Besides, we describe hypothesis about the games we are dealing with and we prove a result of determination.

After that, we have decided to focus on two algorithms: minimax with α-β pruning (chapter 3) and Monte Carlo Tree Search (chapter 4).

Both chapters describe the algorithms and their performance. We also compare strenghts and weaknesses between them.

After these two central chapters, we apply all the general concepts we already know to the game of chess. We present practical examples where we show strengths and weaknesess and, also, we give some clues to understand how these

algorithms search for the best possible move.

This application takes us to the next natural step: the implementation of the two algorithms studied. The implementation is done using Python software but may be done in plenty of different programming languages.

To conclude, we sum up all the work done with a global analysis and we study a possible line of future work using Machine Learning techniques applied to computer chess players.

Capítulo 1

Contexto histórico.

The enemy knows the system.

— Claude Shannon, matemático e ingeniero eléctrico.

La historia relevante referente a jugadores de ajedrez automáticos comienza con Shannon en 1949-1950. Previamente, se habían realizado algunos intentos engañosos como es el famoso caso del autómata 'El Turco' (1770) que logró derrotar a grandes personajes aficionados al ajedrez de la época como Napoleón Bonaparte [31] o Benjamin Franklin [28]. El secreto consistía en que realmente no había tal autómata sino que un jugador experimentado movía las piezas desde debajo del tablero, oculto al contrincante. También, ya en 1912, Leonardo Torres y Quevedo, ingeniero civil y matemático español, construyó un autómata que era capaz de jugar finales de Rey y Torre contra Rey (RT vs. R) sin intervención humana.

1.1. Primeros pasos.

Shannon no propuso un programa de ajedrez, sino que estudió los problemas básicos involucrados en la creación de uno. Su artículo de 1950 [24] guió al resto de desarrolladores en los años posteriores. Observó que el ajedrez es un juego finito de posiciones cada una de las cuales admite un conjunto de jugadas legales. En el capítulo 2 daremos fundamento a esta observación y en el capítulo 3 veremos cómo se ve traducida al contexto de la teoría de grafos, definiendo el

árbol de juego.

Las reglas del ajedrez aseguran que el juego terminará en victoria para alguno de los jugadores o bien en tablas (empate), ya sea por acuerdo, por repetición triple de una misma posición (evitando los ciclos infinitos), o por otros casos muy específicos, como son el rey ahogado (el rey no está en jaque pero no puede moverse así como el resto de las piezas de ese jugador) o por inexistencia de capturas y movimientos de peones en 50 jugadas. Estas reglas suponen una ligadura para el objetivo marcado para nuestro jugador automático y tendrán que ser tomadas en cuenta cuando tengamos que considerar distintas jugadas posibles.

Shannon estimó la dimensión del árbol (grafo) que representa al juego y concluyó que la cantidad de jugadas era inabordable computacionalmente. No obstante, llegó a comprender que la base para obtener una buena jugada consistía en *evaluar* las mejores opciones disponibles. Introdujo la búsqueda acotada (hasta cierta profundidad del árbol de juego) y usó una primera función de evaluación[1] estudiando pocas características: valor de las piezas y pequeños patrones de peones. Además, su máxima era la que acompaña a este capítulo como epígrafe: el enemigo conoce el sistema, es decir, hemos de buscar la mejor jugada posible suponiendo que el adversario también buscará hacer la mejor jugada y tiene la misma información que nosotros.

Tras él, Turing [16] en 1953 tampoco propuso un jugador automático particular, pero sí introdujo la noción de posición 'muerta' o quiescente (como se definirá en el capítulo 5) para limitar el número de cálculos realizados. Además, estudió varias características asociadas a la función de evaluación que mejoraban la propuesta de Shannon.

Fue en 1956, en Los Alamos [16], donde se creó el programa MANIAC I. Contaba con los estudios previos, especialmente de Shannon, con una primera aproximación de lo que sería el algoritmo minimax (véase capítulo 3) pero con la

[1] Se dará una definición para este concepto en el capítulo 3.

limitación de trabajar sobre un tablero de 6×6 (en contraste con el tablero real de 8×8) casillas para simplificar los cálculos. Eliminaron los alfiles del tablero y algunos movimientos especiales como el enroque.

Tras MANIAC I, Bernstein [16] diseñó en 1958 un programa en IBM para un tablero 8×8 llegando a jugar partidas completas. Consideraba únicamente una fracción de las posibles jugadas legales y la función de evaluación estudiaba varias características como la seguridad del rey o el espacio disponible. Conseguía calcular hasta 4 plies (equivalente a 2 jugadas para cada adversario), como el de Los Álamos, lo cual provocaba fallos por el problema del horizonte (véase capítulo 5) pero era capaz de jugar una partida de ajedrez completa.

En paralelo a Bernstein, Kotok y McCarthy (MIT 1960), [16], incluyeron un análisis en profundidad variable hasta posiciones muertas o hasta una profundidad fijada previamente. Los movimientos estudiados eran aquéllos que incluyeran ciertas características mínimas, evitando el cálculo de jugadas malas. Además, se estudiaban con especial atención las capturas y los jaques. Éste fue el primer jugador automático de ajedrez con un nivel de juego respetable.

1.2. La carrera hacia la derrota del campeón del mundo humano.

Greenblatt en 1966 (Laboratorios de IA del MIT) [16] diseñó un programa que contaba con varias ayudas importantes para eliminar errores y mejorar el rendimiento. Seguía una línea principal de jugadas y recogía estadísticas sobre las posiciones. Realizó un entrenamiento jugando contra sí mismo. La función de evaluación caracterizaba el balance de material, el espacio de las piezas, la estructura de peones, la seguridad del rey y el control de las casillas centrales. Además, se introdujeron una serie de tablas de transposición[2], lo cual ayudaba a no volver a evaluar nodos a los que ya se había llegado por otro camino del árbol.

[2]Trasponer, en este contexto, significa alcanzar una misma posición por dos o más caminos diferentes.

Durante la primera mitad de la década 1970-1980 aumentó la cantidad y la calidad de jugadores automáticos, llegando a celebrarse por primera vez el mundial de ajedrez para computadores. Programas como KAISSA [16, 29], Chess 4.7 o Cray Blitz comenzaron a ganar a jugadores profesionales de ajedrez pero no llegaban al nivel de los mejores. Se llegó a pensar entonces que las máquinas jamás podrían superar al humano si bien el campeón del mundo (1972-1975) Bobby Fisher llegó a plantear alternativas al ajedrez clásico (véase capítulo 5) para retrasar, en su opinión, una probable derrota del humano contra la máquina.

Durante la parte final de los 80 y toda la década de los 90 se realizaron enfrentamientos entre los mejores jugadores automáticos del mundo y campeones del mundo como Anatoly Karpov. Aunque llegaron a perder alguna partida, los humanos aún podían vencer a los algoritmos. Fue en 1997 en el match de revancha cuando IBM Deep Blue consiguió vencer al entonces campeón del mundo de ajedrez, Garry Kasparov, por 4 a 2. El propio Kasparov pidió la revancha pero IBM rechazó y Deep Blue desapareció de la competición dando paso a una nueva etapa tanto para los jugadores automáticos de ajedrez como para los jugadores humanos.

1.3. Ajedrez moderno.

En la década de los 2000 queda patente la superioridad de los algoritmos como Hydra o Fritz frente al cálculo humano y campeones del mundo como Veselin Topalov preparan el match por el título mundial jugando contra el módulo (jugador automático) Rybka. La reducción del coste computacional viene dada tanto por el perfeccionamiento de las funciones de evaluación como por el desarrollo de hardware específico.

En la presente década los módulos de ajedrez suponen un elemento indispensable para la preparación de partidas por parte de los Grandes Maestros [17]. El ajedrez basado en los principios clásicos [30], especialmente en la fase de la apertura, se ve transformado en preparaciones previas a las partidas oficiales de

la mano de ordenadores de gran potencia [11] dando lugar a gran cantidad de posiciones complejas para el cálculo.

Se dan a lo largo de la década grandes competiciones de módulos de ajedrez como es el caso del TCEC (Top Chess Engine Championship) [32]. En 2011 los 3 campeonatos disputados fueron ganados por el software Houdini. En 2013-2017 tienen lugar hasta 6 campeonatos más ganados por distintos módulos.

Desde enero de 2018 se suceden un TCEC tras otro siendo Stockfish [32] el gran campeón. En el momento en el que se realizó la primera versión de este capítulo el TCEC trascurría en su fase final luchando por el título. El entonces campeón, Stockfish, jugaba contra el jugador automático LeelaChessZero, que llevaba una ventaja de 4 victorias. Finalmente, LeelaChessZero venció 53.5 a 46.5.

Este último, basado en técnicas modernas de Deep Learning, surge como consecuencia de la gran revolución surgida tras la aparición de AlphaZero, jugador de Google desarrollado por el mismo creador de Deep Blue, Feng-hsiung Hsu, el cual consiguió batir en un match de 100 partidas a Stockfish sin perder ni una sola y ganando un total de 28 [25].

En los próximos capítulos nos centraremos en detallar la naturaleza de dos algoritmos que han sido fundamentales para el desarrollo de jugadores automáticos.

Capítulo 2

Juegos finitos.

> *The game is not worth playing if your opponent is programmed to lose.*
>
> — Man in Black, Westworld.

Este capítulo trata de manera general un subconjunto de los juegos de tablero existentes. Excluimos juegos (véase siguiente sección) que no cumplen unos requisitos mínimos para nuestros intereses.

A continuación, fijaremos la notación que vamos a usar de manera general y daremos definiciones y resultados básicos en teoría de juegos finitos. No obstante, durante los capítulos 3 y 4 particularizaremos y desarrollaremos dichos conceptos en el contexto que sea necesario.

2.1. Hipótesis sobre los juegos considerados.

Consideramos el estudio de algunos juegos finitos conocidos como son el ajedrez o el Go de manera general. Tomaremos dos hipótesis para definir el conjunto de juegos sobre los que vamos a desarrollar los resultados.

En primer lugar, el juego enfrenta a dos jugadores, Jugador I y Jugador II, en un juego de tablero y piezas. El jugador I comienza el juego realizando un movimiento o acción. Después, el Jugador II responde con otro movimiento y

así sucesivamente, es decir, juegan por *turnos*. En cierto momento fijo el juego acaba y uno de los dos jugadores ha ganado el juego[1]. Por simplicidad consideramos sólo éste tipo de juegos, llamados de suma cero porque ambos jugadores tienen objetivos contrarios (ganar al adversario), para evitar el caso del empate. De este modo, el Jugador I gana el juego si y sólo si el Jugador II pierde y viceversa. La generalización al caso de juegos con posibilidad de empate no es difícil y será realizada en la próxima sección.

En segundo lugar, consideramos que la información es perfecta, es decir, ambos jugadores tienen pleno acceso a toda la información sobre cómo ha transcurrido el juego. Consideramos juegos como el ajedrez, las damas, el Go, el tres en raya, etc. No consideramos juegos que consideran elementos aleatorios (juegos de dados o cartas), ni juegos en los que los jugadores realizan las jugadas simultáneamente (Piedra-Papel-Tijeras), ni juegos en los que un jugador conoce información que el otro no conoce (como Stratego).

La pregunta natural que surge es la siguiente, ¿cómo sabemos si un juego es finito? La respuesta se obtiene estudiando si la secuencia de jugadas es o no finita. La manera en la que denotamos esta secuencia de jugadas puede ser muy variada. La más útil en este momento es asignar a cada posición del tablero diferente, en el caso del ajedrez, un número natural. Dado que el número de casillas es finito (64), que el número de piezas también (32), que no puede haber dos piezas en la misma casilla y que la repetición de una misma posición 3 veces supone las tablas según las reglas del juego, podemos afirmar que el ajedrez posee una secuencia finita de jugadas. Algunos de los números asignados a las distintas posiciones supondrán una posición final (jaque mate) y, más relevante aún, sólo un subconjunto de todas las posiciones serán consideradas legales o válidas para el juego.

Por tanto, nos interesa que cada juego finito tenga asociada una sucesión de números naturales de longitud fija (L) que codifique cada posible partida. Cuando la partida ha acabado, si la sucesión tiene longitud $l < L$ podemos

[1] Aclararemos esta afirmación tras describir las hipótesis sobre los juegos considerados.

añadir, por ejemplo, el número 0 tantas veces como sea necesario hasta que la sucesión tenga longitud L . De esta manera, independientemente de cómo se desarolle la partida, el juego siempre tendría la misma longitud.

Por último, llamaremos `LEGAL` al conjunto de las secuencias finitas que corresponden a jugadas legales (permitidas por las reglas del juego). El conjunto de las secuencias de jugadas ganadoras para el Jugador I será denominado $\texttt{WIN} \subset \texttt{LEGAL}$. Por tanto, $\texttt{LEGAL} - \texttt{WIN}$ corresponde al subconjunto de secuencias ganadoras para el Jugador II.

En las condiciones dadas, nos encontramos preparados para desarrollar las definiciones y resultados básicos en juegos finitos que necesitamos.

2.2. Determinación en juegos finitos.

Partiremos del conjunto de los números enteros no negativos, $\{0, 1, 2, \dots\}$ y lo denotaremos como ω. Como es habitual, para todo número $n \in \omega$, ω^n es el producto cartesiano de ω, es decir, $\omega^n = \underbrace{\omega \times \cdots \times \omega}_{n \text{ veces}}$. Los elementos de ω^n se denotarán como $\langle x_0, x_1, \dots, x_{n-1} \rangle$, con $x_i \in \omega$, $i \in \{0, \dots, n-1\}$ y serán llamados secuencias finitas.

Cuando sea necesario trataremos a las secuencias finitas como funciones $f : \{0, \dots, n-1\} \longrightarrow \omega$ donde si $f \in \omega^n$, $f(m)$ denota el m-ésimo elemento de la secuencia f. La secuencia vacía se denotará como $\langle\rangle$.

Definición 2.2.1. Un juego finito $G_N(A)$ es una tupla definida por un número natural, $N \in \mathbb{N}$, llamado longitud del juego y un subconjunto A de ω^{2N} que llamaremos *condiciones de victoria para el Jugador I*. Una *partida* de nuestro juego se define mediante la siguiente construcción:
Consideramos dos jugadores que, por turnos, realizan una jugada codificada con un número natural. La jugada del Jugador I en el turno i-ésimo es denotada como x_i y la correspondiente jugada del Jugador II como y_i. Tras N turnos la partida es una secuencia $s := \langle x_0, y_0, x_1, y_1, \dots, x_{N-1}, y_{N-1} \rangle$ de longitud $2N$. Diremos que el Jugador I gana la partida si $s \in A$ y el Jugador II gana si $s \notin A$.

Notemos que con esta definición sólo consideramos que las sucesiones tienen longitud exactamente $2N$ para ser consideradas como juego finito y en ningún caso se está restringiendo las posibles jugadas a las reglas legales del juego como hacíamos al definir el conjunto `LEGAL`. No obstante, esto no limita la cantidad de juegos que podemos modelar. Para resolver el primero de los problemas es posible asignar un número (por ejemplo el 0) para representar el final de una partida rellenando a partir de ese turno todos los x_i e y_i con ceros como ya habíamos comentado previamente. El segundo de los problemas puede ser solucionado descalificando de la partida al jugador que realice una jugada ilegal.

Un concepto fundamental al hablar de juegos es la *estrategia* o método para determinar la mejor jugada dada la sucesión hasta el momento. Formalmente,

Definición 2.2.2. Sea $G_N(A)$ un juego finito de longitud N.
Una *estrategia* para el Jugador I es una función

$$\sigma : \{s \in \bigcup_{n<2N} \omega^n : |s| \text{ par}\} \longrightarrow \omega.$$

Análogamente, una *estrategia* para el Jugador II es una función

$$\tau : \{s \in \bigcup_{n<2N} \omega^n : |s| \text{ impar}\} \longrightarrow \omega,$$

es decir, una estrategia es una función que asigna a cada sucesión de números naturales otro número (la jugada a realizar codificada como número). Si la sucesión de números tiene longitud par, le toca mover al Jugador I y si es impar al Jugador II.

Dada una estrategia σ para el Jugador I, podemos mirar la secuencia $t = \langle y_0, \ldots, y_{N-1} \rangle$ de jugadas del Jugador II. Sería posible entonces considerar la partida resultante al aplicar la estrategia σ a la secuencia t, la cual denotaremos como $\sigma * t$. Análogamente, intercambiando el papel de los jugadores, tendríamos el siguiente par de definiciones.

Definición 2.2.3. 1. Sea σ una estrategia para el Jugador I en el juego $G_N(A)$. Para todo $t = \langle y_0, \ldots, y_{N-1} \rangle$ definimos

$$\sigma * t := \langle x_0, y_0, x_1, y_1, \ldots, x_{N-1}, y_{N-1} \rangle$$

con la fórmula inductiva

$x_0 := \sigma(\langle\rangle)$

$x_{i+1} := \sigma(\langle x_0, y_0, x_1, y_1, \ldots, x_i, y_i \rangle)$

2. Sea τ una estrategia para el Jugador II en el juego $G_N(A)$. Para todo $s = \langle x_0, \ldots, x_{N-1} \rangle$ definimos

$$s * \tau := \langle x_0, y_0, x_1, y_1, \ldots, x_{N-1}, y_{N-1} \rangle$$

con la fórmula inductiva

$y_0 := \sigma(\langle x_0 \rangle)$

$y_{i+1} := \sigma(\langle x_0, y_0, x_1, y_1, \ldots, x_i, y_i, x_{i+1} \rangle)$

Definición 2.2.4. Sea $G_N(A)$ un juego y σ una estrategia para el Jugador I. Denotamos

$$\texttt{Plays}_N(\sigma) := \{\sigma * t : t \in \omega^N\}$$

al conjunto de todas las posibles partidas para el juego $G_N(A)$ en las que Jugador I usa la estrategia σ. Análogamente,

$$\texttt{Plays}_N(\tau) := \{s * \tau : s \in \omega^N\}$$

es el conjunto de todas las posibles partidas para el juego $G_N(A)$ donde el Jugador II usa la estrategia τ.

Definido el concepto de estrategia y de posibles jugadas siguiendo cierta estrategia, ¿podríamos encontrar una estrategia *ganadora* para alguno de los jugadores? Para responder a esta pregunta necesitamos definir lo que significa que una estrategia sea o no ganadora.

Definición 2.2.5. Sea $G_N(A)$ un juego finito.

1. Una estrategia σ es una estrategia ganadora para el Jugador I si para cualquier $t \in \omega^N$, $\sigma * t \in A$.

2. Una estrategia τ es una estrategia ganadora para el Jugador II si para cualquier $s \in \omega^N$, $s * \tau \notin A$.

Es claro que, al ser dos condiciones disjuntas que completan el espacio de sucesos posibles para el resultado del juego, se tiene el siguiente resultado.

Lema 2.2.1. Para cualquier juego $G_N(A)$, los jugadores I y II no pueden tener ambos estrategias ganadoras.

Podemos dar paso ya a la respuesta a la pregunta anterior. En primer lugar, veamos qué se entiende por determinación en juegos finitos.

Definición 2.2.6. Un juego $G_N(A)$ es determinado si alguno de los jugadores tiene una estrategia ganadora.

Teorema 2.2.2. Todo juego finito $G_N(A)$ está determinado.

Demostración. Notemos que el Jugador I tiene una estrategia ganadora en el juego $G_N(A)$ si y sólo si se cumple que

$$\exists x_0 \, \forall y_0, \exists x_1 \, \forall y_1, \ldots, \exists x_{N-1} \, \forall y_{N-1}, \, (\langle x_0, y_0, x_1, y_1, \ldots, x_{N-1}, y_{N-1}, \rangle \in A)$$

Supongamos entonces que el Jugador I no tiene una estrategia ganadora. Entonces

$$\neg(\exists x_0 \, \forall y_0, \exists x_1 \, \forall y_1, \ldots, \exists x_{N-1} \, \forall y_{N-1}, \, (\langle x_0, y_0, x_1, y_1, \ldots, x_{N-1}, y_{N-1}, \rangle \in A))$$

Por dualidad, se tiene la secuencia

$$\forall x_0 \neg(\forall y_0, \exists x_1 \, \forall y_1, \ldots, \exists x_{N-1} \, \forall y_{N-1}, \, (\langle x_0, y_0, x_1, y_1, \ldots, x_{N-1}, y_{N-1}, \rangle \in A))$$

$$\forall x_0 \exists y_0 \neg(\exists x_1 \, \forall y_1, \ldots, \exists x_{N-1} \, \forall y_{N-1}, \, (\langle x_0, y_0, x_1, y_1, \ldots, x_{N-1}, y_{N-1}, \rangle \in A))$$

$$\forall x_0 \exists y_0 \forall x_1 \neg(\forall y_1, \ldots, \exists x_{N-1} \, \forall y_{N-1}, \, (\langle x_0, y_0, x_1, y_1, \ldots, x_{N-1}, y_{N-1}, \rangle \in A))$$

...

$$\forall x_0 \exists y_0, \forall x_1 \, \exists y_1, \ldots, \forall x_{N-1} \, \exists y_{N-1}, \, (\langle x_0, y_0, x_1, y_1, \ldots, x_{N-1}, y_{N-1}, \rangle \notin A))$$

Así, esta condición se cumple si y sólo si el Jugador II tiene una estrategia ganadora en $G_N(A)$. En el otro sentido, es análogo suponer que el Jugador II no tiene una estrategia ganadora y eso implica que el Jugador I sí la tiene, lo cual termina la prueba. □

Este resultado implica que tenemos que establecer un criterio para determinar cómo se gana, se pierde o se empata una partida para que exista determinación. En el caso del ajedrez, resolvemos el problema definiendo dos juegos diferentes: el ajedrez blanco y el ajedrez negro.

Estos juegos son similares al ajedrez considerado como juego finito en las secciones previas con la salvedad de considerar las tablas como victoria para el Jugador I en el ajedrez blanco y victoria para el Jugador II en el ajedrez negro. Ambos juegos son finitos y determinados según las definiciones y resultados ya desarrollados. Recogemos en la siguiente tabla todos los casos posibles para que podamos trasladar el resultado de los dos tipos de ajedrez al ajedrez que conocemos. (Usaremos la abreviatura w.s. para estrategia ganadora).

Ajedrez blanco	Ajedrez negro	Resultado
I con w.s	I con w.s	I gana
II con w.s.	I con w.s.	Imposible
I con w.s.	II con w.s.	Tablas (Empate)
II con w.s.	II con w.s.	II gana

Tabla 2.1: Tabla de decisión para el resultado del ajedrez.

Notemos que el segundo caso (II con w.s en ajedrez blanco y I con w.s. en ajedrez negro) es imposible por contradicción con el concepto de estrategia ganadora. Ambos jugadores saben que el otro jugador gana y, por tanto, la intersección de estrategias es el vacío. Sin embargo, el tercer caso sí es posible puesto que para ambos jugadores tener una estrategia ganadora incluye la victoria o las tablas. La intersección de ambas estrategias ganadoras da como resultado las tablas.

Es claro, entonces, que podemos obtener el siguiente corolario.

Corolario 2.2.2.1 (Zermelo)**.** En ajedrez, o bien el Jugador I tiene una estrategia ganadora, o bien la tiene el Jugador II o, o bien ambos jugadores tienen estrategias para obtener tablas.

Notemos que los resultados desarrollados en este capítulo nos aseguran la existencia de estrategias y determinación de cierto tipo de juego [13]. En nuestro caso, el ajedrez, sabemos que se dan todas las hipótesis pero no sabemos cuáles son las estrategias ganadoras pese a conocer su existencia. En los dos próximos capítulos damos algoritmos que busquen una estrategia ganadora si bien podemos adelantar que los resultados obtenidos son siempre aproximados.

Capítulo 3

Algoritmo Minimax con poda α-β.

> ***Dr. Strange:*** *I went forward in time, to view alternate futures. To see all the possible outcomes of the coming conflict.*
> ***Peter Quill:*** *How many did you see?*
> ***Dr. Strange:*** *14,000,605.*
> ***Tony Stark:*** *How many did we win?*
> ***Dr. Strange:*** *1.*
>
> — Avengers: Infinity War.

En el capítulo anterior hemos desarrollado el contexto en el que vamos a basar todo este trabajo. Hemos definido los juegos que nos interesan así como lo que significa ganar, perder o empatar dada una estrategia. Hemos visto que, además, si la estrategia es ganadora el juego es determinado y hemos probado que todo juego finito lo es, esto es, posee estrategias ganadoras. No obstante, en ningún momento se ha dado un método para obtener la estrategia ganadora. En este capítulo desarrollamos un método aproximado. Pero, ¿por qué necesitamos que sea aproximado? ¿No es posible encontrar una estrategia ganadora siempre si el juego es finito?

El interés por analizar este tipo de juegos reside en su dificultad de resolución. En el caso del ajedrez hemos intuido que la longitud del juego es grande

pero sin entrar a valorar si computacionalmente podemos analizar todo el juego. En el contexto de la teoría de grafos, que introduciremos a continuación para este capítulo, una partida media consta de unas 40-50 jugadas por cada jugador con un factor de ramificación (número de hojas o variantes por cada rama o posición en el tablero) de aproximadamente 35. Esto da lugar a un grafo de unos $35^{100} \simeq 10^{54}$ nodos. Tanto humanamente como computacionalmente realizar este cálculo es impracticable. Los Grandes Maestros Internacionales [16] han desarrollado distintos métodos o estrategias para estudiar hasta 7 variantes con una profundidad de poco más de 12 jugadas. El objetivo de este capítulo es describir algunas de estas estrategias para la reducción del número de cálculos necesarios (especialmente cuando el tiempo de juego es limitado). La más importante de ellas es la poda α-β, de la cual hablaremos un poco más adelante.

3.1. Definiciones y conceptos básicos.

A partir de este momento resulta conveniente etiquetar al jugador que realiza el primer movimiento (Jugador I) como `MAX` y al segundo (Jugador II) como `MIN` por motivos que surgirán de manera natural en próximas secciones. Consideraremos, asimismo, que nos centramos en la perspectiva del primer jugador a menos que se especifique lo contrario.

Como anunciábamos previamente, una representación especialmente útil para modelizar el ajedrez y derivar resultados interesantes la encontramos en el contexto de la teoría de grafos. Damos a continuación definiciones y conceptos que será necesario tener presentes.

Definición 3.1.1. (Grafo dirigido) Un grafo dirigido G es un par ordenado de conjuntos disjuntos (V, E), siendo V el conjunto de vértices o nodos del grafo (en nuestro caso serán posiciones diferentes del tablero) y E un subconjunto de todos los pares ordenados de V (aristas) que representan las conexiones entre dos nodos cualesquiera. En tal caso, si una arista conecta los nodos n y m se dice que m es *sucesor* o hijo de n, o bien se dice que n es el nodo padre de m. El

número de nodos hijos es denominado *grado* o *factor de ramificación* del nodo. Las aristas se denotan por concatenación de vértices, es decir, si x y z son vértices de G, la arista que los une se denota como xz. La concatenación de k aristas que conectan dos nodos n y n' forman un *camino* de longitud k. Decimos en tal caso que n es un ascendiente de n' y n' es un descendiente de n.
En ciertos casos se asignan unos valores o pesos a las aristas representando el coste o recompensa asociada a recorrer dicha arista al realizar una *búsqueda* en el grafo.

Definición 3.1.2. (Camino) Un *camino* en un grafo G es una secuencia finita de vértices $x_0, \ldots, x_n$ y aristas $a_1, \ldots, a_n$ de G tales que

$$x_0, a_1, x_1, a_2, \ldots a_n, x_n.$$

Un camino se dice *simple* si no hay aristas repetidas. Se representa indicando el vértice inicial y final x_0x_n. Cuando ámbos vértices coinciden se dice que el camino es un *ciclo*.

Nota: Un camino simple en el grafo será una partida según definíamos en el capítulo 2.

Definición 3.1.3. (Árbol) Un *árbol* T es un grafo en el que cada nodo, excepto el nodo raíz, tiene un único nodo padre. Un nodo en un árbol sin sucesores es llamado *hoja* o *nodo terminal*. Volviendo a las definiciones del capítulo anterior, un nodo terminal se corresponderá con un número de la secuencia del juego asociado a una posición final.

Consideramos ahora la aplicación del concepto de árbol a la teoría de juegos definiendo el árbol de juego, nuestro objeto de estudio. Las definiciones a continuación dadas casan con las dadas en el capítulo anterior completándolas y enriqueciendo la teoría.

Definición 3.1.4. Un *árbol de juego* T es un grafo dirigido en el cual cada nodo representa explícitamente una posición o jugada posible para el juego en cuestión. Los sucesores de un nodo proporcionan las posiciones a las que puede acceder el jugador al que le corresponde jugar usando el conjunto de reglas permitidas a partir de la posición (nodo) dada. El conjunto de sucesores a los

que cada adversario puede acceder es denominado un *nivel* del árbol. Cada nivel del árbol supone un *turno* para un jugador.

Definición 3.1.5. (Nodo inicial) Denominamos *nodo inicial* o raíz a la posición inicial del juego representado en el grafo. Los *nodos finales* o terminales son aquellos que representan la victoria, derrota o empate desde el punto de vista del primer jugador. Cada camino entre el nodo inicial y uno final representa una partida diferente y completa como ya hemos comentado.

En el primer turno, `MAX` se encuentra en el nodo inicial y tiene disponible todas las posibles posiciones accesibles desde el nodo inicial siguiendo las reglas del juego. En el segundo turno, `MIN` parte desde el nodo seleccionado por el primer jugador y tiene disponibles todas las posiciones legales posibles, y así sucesivamente.

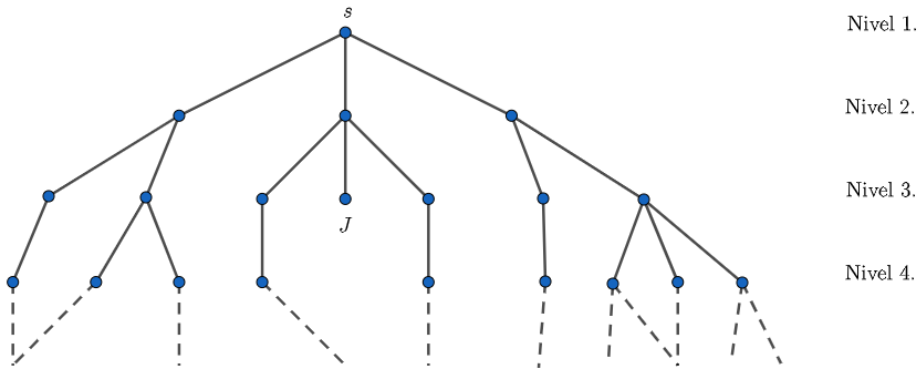


Figura 3.1: Árbol de juego genérico. Podemos observar los distintos niveles (los impares correspondientes al jugador `MAX` y los pares al jugador `MIN`) siendo s el nodo inicial del juego y J un nodo terminal. Nótese que desde distintos nodos podemos converger a una misma posición.

Estamos en condiciones de tratar de asignar a cada nodo del árbol de juego una etiqueta que refleje el estado de una posición, es decir, saber si alguno de los jugadores posee una posición ganadora, perdedora o de tablas[1]. Una primera posibilidad es la siguiente:

[1]Estas etiquetas equivalen a los números asignados en el capítulo anterior a las posiciones terminales.

Sea J un nodo no terminal correspondiente a `MAX`. Entonces podemos definir,

$$\texttt{status}(J) = \begin{cases} \texttt{win}, & \text{alguno de los sucesores de } J \text{ están etiquetados con } \texttt{win} \\ \texttt{loss}, & \text{todos los sucesores de } J \text{ están etiquetados con } \texttt{loss} \\ \texttt{draw}, & \text{algún sucesor de } J \text{ está etiquetado como} \\ & \texttt{draw} \text{ y ninguno como } \texttt{win}, \end{cases}$$

Si J es un nodo correspondiente a `MIN` la equivalencia sería:

$$\texttt{status}(J) = \begin{cases} \texttt{win}, & \text{todos los sucesores de } J \text{ están etiquetados con } \texttt{win} \\ \texttt{loss}, & \text{alguno de los sucesores de } J \text{ están etiquetados con } \texttt{loss} \\ \texttt{draw}, & \text{algún sucesor de } J \text{ está etiquetado como} \\ & \texttt{draw} \text{ y ninguno como } \texttt{loss}. \end{cases}$$

Una vez conocemos el *status* de todo nodo terminal ($\texttt{win}, \texttt{loss}, \texttt{draw}$) podremos determinar el *status* de cada nodo del árbol siguiendo el procedimiento anterior propagando la información hacia los niveles superiores. Para cada nodo J del nivel i-ésimo calculamos el valor de $\texttt{status}(J)$ a partir del etiquetado del de los sucesores de J en el nivel $(i+1)$-ésimo. En último lugar, se etiqueta el nodo raíz a partir de la información obtenida de los nodos del segundo nivel. Por ello, la siguiente definición surge de manera natural.

Definición 3.1.6. *Resolver* el árbol de juego consiste en asignar de manera inequívoca al nodo inicial s la categoría de victoria (`win`), derrota (`loss`) o tablas (`draw`).

Obviamente, esto no es posible hacerlo en la práctica pues implica recorrer y analizar todo el árbol de juego (que, en casos como el ajedrez, ya vimos que era inabarcable desde un punto de vista práctico). Por ello, nuestro objetivo será encontrar la mejor jugada (nodo) posible dentro de las limitaciones computacionales disponibles (tanto de memoria como de tiempo). Necesitamos volver al concepto de estrategia (ganadora) adaptándola al lenguaje de la teoría de grafos.

Definición 3.1.7. Una *estrategia* para `MAX` frente a `MIN` es un subárbol T^* del árbol de juego T llamado *árbol solución*, con raíz s, que contiene un sucesor para cada nodo no terminal de `MAX` (siempre puede responder con una jugada) y todos los sucesores de los nodos no terminales de `MIN` (sus jugadas siempre reciben respuesta) en T^*. Una *estrategia ganadora* es aquélla que garantiza una victoria para uno de los jugadores con independencia de cómo juegue su adversario, es decir, aquella cuyos nodos terminales estén etiquetados como `win`.

Para poder etiquetar los nodos dentro del árbol construiremos una aplicación que asigne un número real a cada sucesor. Esta aplicación es denominada *función de evaluación*, $\mathtt{e}(\cdot)$, y estima el estado en el que se encuentra el nodo J, $\mathtt{status}(J) \simeq \mathtt{e}(J)$[2]. Dedicaremos tiempo para especificar las características que tienen en cuenta las distintas posibles funciones de evaluación en el capítulo 5.

Como ya hemos comentado, conocer con total precisión el etiquetado de los nodos terminales no es tarea sencilla dada la cantidad de nodos a evaluar. Ante tal situación hemos de tomar aproximaciones heurísticas, es decir, técnicas de resolución práctica del problema que primen encontrar un resultado suficiente en un tiempo limitado frente al óptimo real en un tiempo ilimitado. El algoritmo de búsqueda ha de ser capaz de estimar características de la posición que sirvan para obtener una evaluación aproximada. La evaluación total sería, en tal caso, una ponderación de evaluaciones sobre cada una de estas características a estimar.

La heurística que seguiremos es la búsqueda acotada (*bounded look-ahead*), que no trata de usar la función de evaluación directamente, sino que recorre varios niveles hasta cierta cota y evalúa los nodos del nivel final siguiendo la siguiente regla.

[2]Notemos que realmente ambas funciones no son iguales sino que una ayuda a estimar la otra pues $\mathtt{status}(\cdot)$ realiza una descripción cualitativa y estática y mientras que la función de evaluación $\mathtt{e}(\cdot)$ es cuantitativa y dinámica.

Regla minimax.

1. La evaluación del nodo J en el último nivel del árbol acotado se supone igual a la dada por la función de evaluación $\texttt{e}(J)$.

2. Si J es un nodo en el que le toca mover a `MIN`, el valor asignado a ese nodo es igual al mínimo valor de todos sus sucesores.

3. Si J es un nodo en el que le toca mover a `MAX`, el valor asignado a ese nodo es igual al máximo valor de todos sus sucesores.

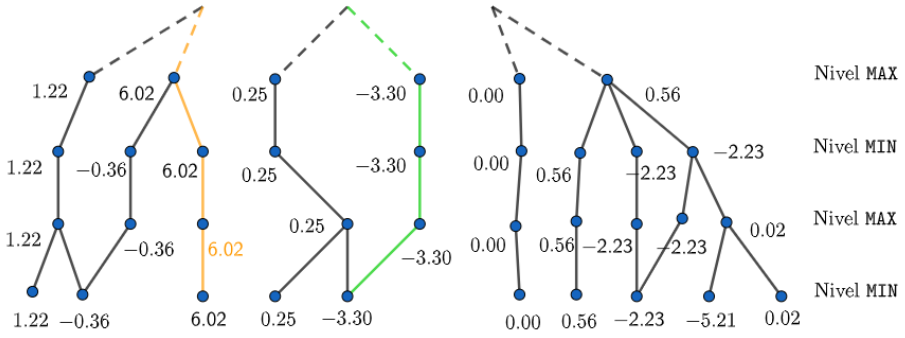


Figura 3.2: Regla minimax. Para cada nivel, el valor de cada nodo se corresponde con el máximo o mínimo de los valores de sus sucesores. La mejor estrategia para el jugador `MAX` se resalta en color naranja y la mejor estrategia para el jugador `MIN` en color verde.

Notemos que estamos tomando como hipótesis que podemos estimar el valor real (estático y cualitativo) de un nodo mediante la función de evaluación (dinámica y variable con la profundidad), es decir, $\texttt{status}(J) = \texttt{e}(J)$.

3.2. El algoritmo. Versiones básicas.

Volviendo al contexto de teoría de juegos finitos recordemos que, por hipótesis, sabemos que para cada posición p existe un número $N(p)$ tal que ninguna partida que comience en p puede durar más de $N(p)$ movimientos. La cota $N(p)$ tiene una función doble: por un lado, nos asegura que el juego es finito y, por otro, nos limita el número de niveles (y nodos) que tenemos que estudiar en la práctica.

Hasta ahora, hemos tomado el punto de vista del primer jugador para realizar todo el desarrollo de los conceptos básicos. No obstante, existe un procedimiento alternativo conocido como neg-max, que considera cada nodo desde el punto de vista del jugador que tiene que jugar. De este modo, un nodo terminal tiene el estado de `win`, `loss` o `draw` sólo si el jugador que tiene que jugar gana, pierde o hace tablas en ese nodo respectivamente. Definiremos este estado del siguiente modo:

$$\texttt{mistatus}(J) = \begin{cases} \texttt{win}, & \text{algún sucesor de } J \text{ está etiquetado con } \texttt{loss}. \\ \texttt{loss}, & \text{todos los sucesores de } J \text{ están etiquetados con } \texttt{win}. \\ \texttt{draw} & \text{ninguno de los anteriores.} \end{cases}$$

Esta descripción es alternativa a la anterior definición de $\texttt{status}(J)$ pero igualmente equivalente. Ambas descripciones serán usadas según ayuden a facilitar los procedimientos en cada caso.

Si existen d movimientos legales, $p_1, \ldots, p_d$, donde $d > 1$, debemos de elegir el mejor movimiento. El mejor movimiento, según la regla minimax, se decide maximizando el valor de la función de evaluación. Si adoptamos el punto de vista neg-max, sea ese valor $F(p)$ suponiendo que el jugador adversario tiene una estrategia óptima para defenderse. Entonces,

$$F(p) = \begin{cases} \texttt{e}(p) & \text{si } d = 0, \\ \text{máx}\{-F(p_1), \ldots, -F(p_d)\} & \text{si } d > 0. \end{cases} \tag{3.1}$$

Esta fórmula define $F(p)$ para todas las posiciones p por inducción en la longitud del juego a partir de la posición p.

Si adoptamos el punto de vista de `MAX`, en el caso de que el turno sea de `MIN` en una posición terminal para él, su valor se denotará como $-\texttt{e}(p)$. Por ello, `MAX` tratará de maximizar la función de evaluación y `MIN` de minimizarla. Definimos dos funciones de evaluación

$$F(p) = \begin{cases} \mathsf{e}(p) & \text{si } d = 0, \\ \max\{G(p_1), \ldots, G(p_d)\} & \text{si } d > 0. \end{cases}$$

con

$$G(p) = \begin{cases} -\mathsf{e}(p) & \text{si } d = 0, \\ \min\{F(p_1), \ldots, F(p_d)\} & \text{si } d > 0. \end{cases}$$

Es claro que $G(p) = -F(p)$ para toda posición p. Según nos sea conveniente, usaremos la formulación anterior usando las dos funciones de evaluación o la formulación neg-max de la ecuación 3.1.

Veamos un primer algoritmo para calcular $F(p)$.

Algoritmo 1 Versión básica del algoritmo minimax.

Entrada: Función de evaluación $\mathsf{e}(\cdot)$ y posición p.
Salida: F.

```
1: Determina los sucesores de la posición p, p_1, ..., p_d.
2: if d = 0 then
3:     F ← e(p)
4: else
5:     m ← −∞
6:     for i ← 1 ... d do
7:         t ← −F(p_i)
8:         if t > m then
9:             m ← t
10:    F ← m
```

Este algoritmo 'de fuerza bruta' estudia todas las posibles partidas tomando como raíz la posición p. Es posible, como veremos a continuación, mejorar este algoritmo usando una técnica de acotación y corte, ignorando movimientos que no es posible que sean mejores que los ya estudiados.

Siguiendo esta idea, podemos pensar en definir una función similar a 3.1 con cierta cota que evite evaluar todos los nodos sucesores y que, a su vez, no pierda información relevante para la evaluación del nodo.

Así, definimos F_1 como un procedimiento dependiente de dos parámetros $F_1(p, b)$ siendo p la posición dada y b la cota que cumple

$$\begin{cases} F_1(p,b) = F(p) & \text{si } F(p) < b \\ F_1(p,b) \geq b & \text{si } F(p) \geq b \end{cases} \tag{3.2}$$

lo cual implica que $F_1(p,\infty) = F(p)$ evidentemente. Siguiendo esta idea, podemos mejorar el anterior algoritmo de la siguiente manera.

Algoritmo 2 Minimax con cota superior.

Entrada: Función de evaluación $\mathtt{e}(\cdot)$, posición p y cota b.
Salida: F_1.

```
1:  Determina los sucesores de la posición p, p_1, ..., p_d.
2:  if d = 0 then
3:      F_1 ← e(p)
4:  else
5:      m ← −∞
6:      for i ← 1 ... d do
7:          t ← −F_1(p_i, −m)
8:          if t > m then
9:              m ← t
10:         if m ≥ b then
11:             goto [A]          ▷ Condición de corte. No exploramos más.
12:     [A] F_1 ← m
```

Lema 3.2.1. La función F_1 calculada por el algoritmo anterior cumple 3.2.

Demostración. Tenemos que probar que para toda posición p y para toda cota $b \in [-\infty, +\infty]$ se cumple 3.2. Por inducción en la altura de p en el árbol de juego:

Caso base: p es terminal. Entonces, para todo $b \in [-\infty, +\infty]$, $F_1(p,b) = \mathtt{e}(p) = F(p)$ y se verifica 3.2 trivialmente.

Paso de inducción: Supongamos que p no es terminal, que sus sucesores son $p_1, \ldots, p_d$ y que para cada $j \in \{1, \ldots, d\}$, p_j satisface 3.2 para todo valor $b \in [-\infty, +\infty]$.

Por inducción en $i \in \{1, \ldots, d\}$ podemos probar que durante el cálculo de $F_1(p,b)$, al principio de la i-ésima iteración del bucle for (si se produce) se verifica

$$m = \text{máx}\{-F(p_1), \ldots, -F(p_{i-1})\} \text{ y } m < b.$$

Para $i = 1$ esto es trivial (suponemos $m = \text{máx}\,\emptyset = -\infty$ por definición). Si se cumple para un $i < d$ entonces también se tiene para $i + 1 \leq d$ ya que, por hipótesis de inducción

$$F_1(p_i, -m) = \begin{cases} F(p_i), & \text{si } F(p_i) < -m \\ \geq -m, & \text{si } F(p_i) \geq -m. \end{cases}$$

Si $F(p_i) < -m$ entonces $-F_1(p_i, -m) = -F(p_i) > m$ luego el nuevo valor de m (al principio de la $i + 1$-ésima iteración del bucle for) será

$$\text{máx}\{m, -F(p_i)\} = \text{máx}\{-F(p_1), \ldots, -F(p_i)\}$$

si $-F(p_i) < b$. Si por el contrario $-F(p_i) \geq b$ el valor devuelto por el procedimiento será $F_1(p, b) = -F(p_i) \geq b$ y no se producirá la $i + 1$-ésima iteración.

Si $F(p_i) \geq -m$ entonces $-F(p_i) \leq m$ y m conserva su valor al iniciar la $i + 1$-ésima iteración.

Una vez probada esta observación podemos comprobar que el valor calculado $F_1(p, b)$ verifica 3.2 ya que o bien devuelve un valor $\geq b$ o, en caso contrario, agota las d iteraciones del bucle for devolviendo

$$m = \text{máx}\{-F(p_1), \ldots, -F(p_d)\} = F(p) < b.$$

□

Este procedimiento se puede mejorar aún más si introducimos cotas inferiores y superiores, las llamadas cotas α-β.

3.3. Poda α-β.

El algoritmo de poda α-β sirve para agilizar la búsqueda en el árbol sin pérdida de información. Determina el valor minimax recorriendo el árbol en un determinado orden saltándose aquellos nodos que no tienen influencia en el valor minimax del nodo raíz.

Las cotas de corte se actualizan de manera dinámica (cada vez que profundizamos en el árbol de juego) y se transmiten de padres a hijos cumpliendo las siguientes definiciones.

Definición 3.3.1. (Cota α). La cota para el corte de un nodo J para `MIN` es una cota inferior, denominada α, que es igual al mayor valor actualizado de todos los ascendientes de J (que son del jugador `MAX`). La búsqueda termina tan pronto como el valor actualizado sea igual o inferior a α.

Definición 3.3.2. (Cota β). La cota para el corte de un nodo J de `MAX` es una cota superior, denominada β, que es igual al menor valor actualizado de todos los ascendientes de J (que son del jugador `MIN`). La búsqueda termina tan pronto como el valor actualizado sea igual o superior a β.

El algoritmo para la poda y actualización de valores se describe a continuación. Usaremos una nueva función F_2, dependiente de la función de evaluación, $\mathsf{e}(\cdot)$, y de los parámetros α y β, con $\alpha < \beta$. En este caso, se evalúa $\mathsf{e}(J)$ pero sólo si ese valor se encuentra entre α y β. Si no, devuelve α si $\mathsf{e}(J) \leq \alpha$ o β si $\mathsf{e}(J) \geq \beta$.

Necesitamos, por tanto, definir un segundo procedimiento F_2 con tres parámetros p, α, β con $\alpha < \beta$ satisfaciendo las condiciones

$$\begin{cases} F_2(p,\alpha,\beta) \leq \alpha & \text{si } F(p) \leq \alpha \\ F_2(p,\alpha,\beta) = F(p) & \text{si } \alpha < F(p) < \beta \\ F_2(p,\alpha,\beta) \geq \beta & \text{si } F(p) \geq \beta \end{cases} \tag{3.3}$$

que cumple $F_2(p, -\infty, \infty) = F(p)$.

Así, el algoritmo resultante sería:

Algoritmo 3 Minimax con poda α-β.

Entrada: Función de evaluación $\mathsf{e}(\cdot)$, posición p y cotas α, β.
Salida: F_2.

```
1:  Determina los sucesores de la posición p, p_1, ..., p_d.
2:  if d = 0 then
3:      F_2 ← e(p)
4:  else
5:      m ← α
6:      for i ← 1 ... d do
7:          t ← −F_2(p_i, −β, −m)
8:          if t > m then
9:              m ← t
10:         if m ≥ β then
11:             goto [A]
12:     [A] F_2 ← m
```

Lema 3.3.1. La función F_2 calculada por el algoritmo anterior cumple 3.3.

Demostración. Tenemos que probar que para toda posición p y para toda cota real $b \in [-\infty, +\infty]$ se cumple 3.3. Por inducción en la altura de p en el árbol de juego:

Caso base: p es terminal. Entonces, para todos $\alpha, \beta \in [-\infty, +\infty]$ con $\alpha < \beta$, $F_2(p, \alpha, \beta) = \mathsf{e}(p) = F(p)$ y se verifica 3.3 trivialmente.

Paso de inducción: Supongamos que p no es terminal, que sus sucesores son $p_1, \ldots, p_d$ y que para cada $j \in \{1, \ldots, d\}$, p_j satisface 3.3 para todos $\alpha, \beta \in [-\infty, +\infty]$ con $\alpha < \beta$.

Por inducción en $i \in \{1, \ldots, d\}$ podemos probar que durante el cálculo de $F_2(p, \alpha, \beta)$, al principio de la i-ésima iteración del bucle for (si se produce) se verifica

$$m = \text{máx}\{\alpha, -F(p_1), \ldots, -F(p_{i-1})\} \text{ y } m < \beta.$$

Para $i = 1$ esto es trivial pues $m = \text{máx}\{\alpha\} = \alpha$. Si se cumple para un $i < d$

entonces también se tiene para $i+1 \leq d$ ya que, por hipótesis de inducción

$$F_2(p_i, -\beta, -m) = \begin{cases} \leq -\beta, & \text{si } F(p_i) \leq -\beta \\ F(p_i), & \text{si } -\beta < F(p_i) < -m \\ \geq -m, & \text{si } F(p_i) \geq -m. \end{cases}$$

Si $F(p_i) \leq -\beta$ entonces $-F_2(p_i, -\beta, -m) = -F(p_i) \geq \beta$ y el valor devuelto por el procedimiento será

$$F_2(p, \alpha, \beta) = -F_2(p_i, -\beta, -m) \geq \beta$$

sin que se produzcan más iteraciones del bucle for.

Si $-\beta < F(p_i) < -m$ entonces

$$m < -F_2(p_i, -\beta, -m) = -F(p_i) < \beta$$

y el nuevo valor de m al inicio de la $(i+1)$-ésima iteración del bucle for será

$$\text{máx}\{-m, -F(p_i)\} = \text{máx}\{\alpha, -F(p_1), \cdots - F(p_i)\} = -F(p_i) < \beta.$$

Por último, si $F(p_i) \geq -m$, entonces

$$-F_2(p_i, -\beta, -m) \leq m < \beta$$

y m conservará su valor en la $(i+1)$-ésima iteración del bucle for.

□

Este algoritmo está desarrollado bajo la perspectiva neg-max. Para completar de manera coherente la teoría vamos a reescribir el correspondiente algoritmo separado en dos subrutinas (F_2 y G_2), una para cada jugador, que se llaman mutuamente y generan el mismo resultado.

Algoritmo 4 Subrutina 1.

Entrada: Función de evaluación $\mathtt{e}(\cdot)$, posición p y cotas α, β.
Salida: F_2.

1: Determina los sucesores de la posición $p, p_1, \ldots, p_d$.
2: **if** $d = 0$ **then**
3: $\quad F_2 \leftarrow \mathtt{e}(p)$
4: **else**
5: $\quad m \leftarrow \alpha$
6: $\quad$ **for** $i \leftarrow 1 \ldots d$ **do**
7: $\quad\quad t \leftarrow G_2(p_i, m, \beta)$
8: $\quad\quad$ **if** $t > m$ **then**
9: $\quad\quad\quad m \leftarrow t$
10: $\quad\quad$ **if** $m \geq \beta$ **then**
11: $\quad\quad\quad$ **goto** [A]
12: $\quad$ [A] $F_2 \leftarrow m$

Algoritmo 5 Subrutina 2.

Entrada: Función de evaluación $\mathtt{e}(\cdot)$, posición p y cotas α, β.
Salida: F_2.

1: Determina los sucesores de la posición $p, p_1, \ldots, p_d$.
2: **if** $d = 0$ **then**
3: $\quad G_2 \leftarrow -\mathtt{e}(p)$
4: **else**
5: $\quad m \leftarrow \beta$
6: $\quad$ **for** $i \leftarrow 1 \ldots d$ **do**
7: $\quad\quad t \leftarrow F_2(p_i, m, \alpha)$
8: $\quad\quad$ **if** $t < m$ **then**
9: $\quad\quad\quad m \leftarrow t$
10: $\quad\quad$ **if** $m \leq \alpha$ **then**
11: $\quad\quad\quad$ **goto** [A]
12: $\quad$ [A] $F_2 \leftarrow m$

La siguiente pregunta natural es, ¿podemos mejorar aún más éste procedimiento? La respuesta es no, este procedimiento ya es óptimo en cierto sentido que veremos en las secciones 3.4 y 3.5.

No obstante, notemos que la eficiencia de este método depende directamente del orden en el que se encuentran los nodos terminales. En juegos complejos, la diferencia entre el mejor y el peor caso posible puede ser significativa. Estudiemos, a continuación, qué ocurre en el mejor de los casos, es decir, el caso de orden perfecto.

3.4. Análisis del mejor caso. Orden perfecto.

El orden perfecto es aquél en el que la mejor jugada posible en cada turno es siempre el primer sucesor accesible desde la posición en la que se encuentre la partida. Usaremos la siguiente notación para facilitar la prueba de los resultados.

Cada posición en el nivel l tiene asociada una secuencia de números enteros positivos $a_1 \dots a_l$. El nodo raíz tiene asociado la secuencia vacía y los sucesores de la posición $a_1 \dots a_l$ tienen asignadas las correspondientes coordenadas $a_1 \dots a_l 1, a_1 \dots a_l 2, \dots, a_1 \dots a_l d$. Por ejemplo, la posición 314 se obtiene a partir del tercer posible movimiento desde la posición inicial, después el primer movimiento posible y, más tarde, el cuarto.

Definición 3.4.1. Una posición $a_1 \dots a_l$ se dirá que es crítica si $a_i = 1$ para todo i par o para todo i impar. El nodo raíz se considera siempre crítico.

Esta definición nos ayuda a obtener el siguiente teorema debido a Knuth y Moore que completa un resultado previo enunciado por Levin [14].

Teorema 3.4.1. Consideremos un árbol de juego para el cual la evaluación del nodo raíz no es $\pm\infty$, y para el cual el primer sucesor de cada posición es óptimo, es decir,

$$F(a_1 \dots a_l) = \begin{cases} \mathsf{e}(a_1 \dots a_l) & \text{si } a_1 \dots a_l \text{ es terminal,} \\ -F(a_1 \dots a_l 1) & \text{en otro caso.} \end{cases}$$

Entonces, el procedimiento $F_2((\cdot), -\infty, +\infty)$ de α-β examina exactamente las posiciones críticas de este árbol de juego.

Demostración. Diremos que una posición crítica $a_1 \dots a_l$ es de tipo I si todos los a_i son 1; de tipo II si la primera entrada a_j mayor que 1 verifica $l - j$ par y de tipo III en caso contrario ($l - j$ es impar y, por ello, $a_l = 1$). Veamos ahora cómo se comporta el algoritmo en estas posiciones críticas.

Tipo I: Una posición $p = \underbrace{1 \dots 1}_{l}$ de este tipo es examinada con la llamada $F_2(p, -\infty, \infty)$. Si no es terminal, su sucesor $p_1 = \underbrace{1 \dots 11}_{l'}$ es de tipo I y

$F(p) = -F(p_1) \neq \pm\infty$. El resto de sucesores $p_2 = 1\ldots12, \ldots, p_d = 1\ldots1d$ son de tipo II (porque $a_j > 1$ coincide con $a_{l'}$, es decir, $l' = j$ y $l' - j = 0$, que es par) y todos son examinados con la llamada $F_2(p_i, -\infty, F(p_1))$.

Tipo II: En este caso la posición p es examinada con $F_2(p, -\infty, \beta)$ con $-\infty < \beta \leq F(p)$. Si no es terminal, su sucesor p_1 es de tipo III (pues si $l - j$ par, el sucesor tiene longitud $l+1$ y $l+1-j$ es un número impar), $F(p) = -F(p_1)$ y los otros sucesores $p_2, \ldots, p_d$ no son examinados.

Tipo III: En este caso la posición p es examinada con $F_2(p, \alpha, +\infty)$ con $+\infty > \alpha \geq F(p)$. Ahora, si la posición no es terminal, los sucesores son de tipo II (pues, si $l-j$ era impar, $l+1-j$ es par independientemente del valor de a_{l+1}), y son examinados con $F_2(p_i, -\infty, -\alpha)$. Como consecuencia, todas las posiciones críticas son examinadas.

Si una posición $p = a_1 \ldots a_l$ no es crítica, entonces existen $r, s \leq l$ tales que $r < s$ y p es de la forma

$$a_1 \ldots a_r a_{r+1} a_{r+2} \ldots a_s a_{s+1} a_{s+2} \ldots a_l$$

verificando que:

Si $i \in \{1, \ldots, r\}$, $a_i = 1$.
$a_{r+1}, a_{s+1} > 1$.
$s - r = 2h + 1$ para cierto $h \geq 0$.
$a_{r+2i} = 1$ para todo $i \in \{1, \ldots, h\}$.

Es fácil comprobar que: $a_1 \ldots a_r$ es una posición de tipo I y, por tanto, $a_1 \ldots a_r a_{r+1}$ es de tipo II. En consecuencia, dado que $s-(r+1)$ es par, $a_1 \ldots a_s$ es una posición de tipo II luego $a_1 \ldots a_s a_{s+1}$ es un sucesor secundario de $a_1 \ldots a_s$ y, por tanto, no es examinado por el procedimiento ni él ni su descendiente $a_1 \ldots a_l$. Esto prueba que sólo son examinadas las posiciones críticas.

□

Corolario 3.4.1.1. Si toda posición en los niveles $0, 1, \ldots, l-1$ de un árbol de juego en las condiciones del teorema anterior tiene exactamente d sucesores, $d > 0$, entonces la poda α-β examina exactamente

$$d^{\lfloor l/2 \rfloor} + d^{\lceil l/2 \rceil} - 1$$

posiciones en el nivel l.

Demostración. Existen $d^{\lfloor l/2 \rfloor}$ secuencias $a_1 \ldots a_l$ con $1 \leq a_i \leq d$ para todo i tales que $a_i = 1$ para todos los i impares y $d^{\lceil l/2 \rceil}$ con $a_i = 1$ para todos los i pares. Debemos restar 1 por la secuencia $1 \ldots 1$ que ha sido contada dos veces. □

Podríamos pensar que la poda α-β sería más efectiva cuando el orden perfecto se da, es decir, cuando el primer sucesor de cada posición es el mejor movimiento posible. Esto no es siempre cierto como se ejemplifica en [14].

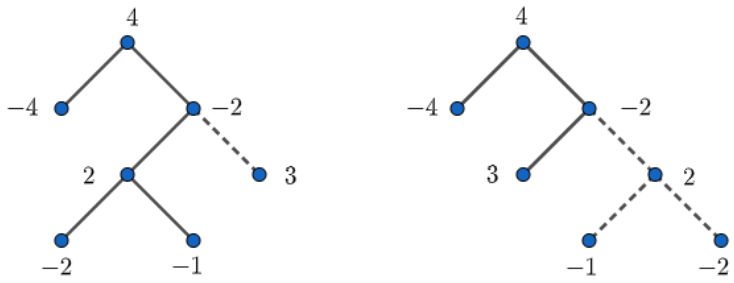


Figura 3.3: En el árbol de la izquierda tenemos una ordenación perfecta de las posiciones. El algoritmo α-β examina las 6 posiciones marcadas (ya que son las posiciones críticas). En cambio, en el árbol de la derecha, el algoritmo α-β sólo examina 3 posiciones a pesar de que su ordenación no es perfecta.

Encontrar el verdadero óptimo no es trivial. No obstante, veamos que es posible siempre encontrar un orden para procesar el árbol tal que el algoritmo α-β examina el mínimo posible de posiciones terminales y ningún algoritmo lo puede hacer mejor.

Teorema 3.4.2. El algoritmo de poda α-β es un procedimiento óptimo en el siguiente sentido: dado cualquier árbol de juego y cualquier algoritmo que calcula el valor de la posición raíz, existe un modo de reordenar los sucesores tal que todas las posiciones terminales examinadas por el algoritmo α-β son examinadas por el algoritmo dado. Además si el valor de la raíz no es $\pm\infty$, el algoritmo α-β examina exactamente las posiciones que son críticas bajo dicha permutación.

Demostración. Definimos las siguientes funciones F_l y F_u, para las mejores cotas posibles para el valor de cualquier posición p, basadas en las posiciones terminales examinadas por el algoritmo dado:

$$F_l(p) = \begin{cases} -\infty & \text{si } p \text{ es terminal y no ha sido examinado} \\ \mathsf{e}(p) & \text{si } p \text{ es terminal y ha sido examinado} \\ \text{máx}\{-F_u(p_1), \ldots, -F_u(p_d)\} & \text{en otro caso} \end{cases} \tag{3.4}$$

$$F_u(p) = \begin{cases} +\infty & \text{si } p \text{ es terminal y no ha sido examinado} \\ \mathsf{e}(p) & \text{si } p \text{ es terminal y ha sido examinado} \\ \text{máx}\{-F_l(p_1), \ldots, -F_l(p_d)\} & \text{en otro caso} \end{cases} \tag{3.5}$$

Notemos que $F_l(p) \leq F_u(p)$ para toda posición p. Además $F(p)$ puede tomar todos los valores entre ambas cantidades pero nunca puede pasar sus límites. En el caso de ser p el nodo raíz, se debe cumplir que $F_l(p) = F_u(p) = F(p)$.

Supongamos que el valor de la raíz no es $\pm\infty$. Veamos cómo reordenar el árbol para que cada posición crítica terminal sea examinada por el algoritmo y que, además, el algoritmo α-β dado por F_2 examina exactamente dichas posiciones terminales. Al igual que hacíamos en el teorema 3.4.1, las posiciones críticas pueden ser clasificadas como tipo I, II o III, siendo la raíz una posición crítica de tipo I. Entonces, por inducción:

Una posición p tipo I cumple $F_l(p) = F_u(p) \neq \pm\infty$ y es examinada por

el algoritmo α-β mediante la función $F_2(p, -\infty, \infty)$. Si p es terminal, debe ser examinada por el algoritmo dado que $F_l(p) \neq -\infty$. Si no es terminal, sean j y k tales que $F_l(p) = -F_u(p_j)$ y $F_u(p) = -F_l(p_k)$. Entonces, por 3.4 y 3.5

$$F_l(p_k) \leq F_l(p_j) \leq F_u(p_j) = -F(p) = F_l(p_k).$$

Por tanto, $F_l(p_j) = F_l(p_k)$ y podemos asumir que $j = k = 1$. La posición p_1 (tras el ordenamiento) es de tipo I. Los otros sucesores $p_2, \ldots, p_d$ son de tipo II y son examinados con $F_2(p_i, -\infty, -F(p_1))$.

Una posición p tipo II cumple $F_l(p) > -\infty$ y es examinada por el algoritmo α-β con $F_2(p, -\infty, \beta)$ y $-\infty < \beta \leq F_l(p)$. Si p es terminal, debe ser examinada por el algoritmo. Si no lo es, sea j tal que $F_l(p) = -F_u(p_j)$ y ordenamos los sucesores si es necesario para que $j = 1$. La posición p_1 tras el reordenamiento es de tipo III y es examinada con $F_2(p_1, -\beta, \infty)$. Como $F_u(p_1) = F_l(p) \leq -\beta$, el procedimiento devuelve un valor menor o igual que $-\beta$. Por tanto, los sucesores $p_2, \ldots, p_d$ (que no son posiciones críticas) no son examinados por el procedimiento α-β ni tampoco sus descendientes.

Una posición p tipo III cumple $F_u(p) < \infty$ y es examinada por el algoritmo α-β mediante $F_2(p, \alpha, \infty)$ siendo $F_u(p) \leq \alpha < \infty$. Si p es terminal, debe ser examinada por el algoritmo. Si no, todos sus sucesores p_i son de tipo II y deben ser examinados por $F_2(p_i, -\infty, -\alpha)$. En este caso, no es necesario reordenar dado que no provoca ninguna diferencia.

El caso en el que el nodo raíz tiene valor ∞ el argumento es similar y debemos tratarlo como una posición tipo II mientras que si tiene valor $-\infty$ como una posición tipo III. □

Notemos que, pese a constatar que el número de posiciones a evaluar por minimax es mínimo si tenemos un cierto ordenamiento del árbol de juego (con respecto a cualquier otro algoritmo de búsqueda), no tenemos un método que nos genere el ordenamiento del árbol.

Es por ello interesante realizar un estudio de la poda α-β desde el punto de vista de la influencia que tienen los nodos podados sobre el nodo raíz. Esta información será útil para construir un criterio que determine si un nodo cualquiera será importante con respecto al rendimiento esperado de la poda α-β.

3.5. Rendimiento del algoritmo.

Supongamos que nos encontramos en cierto punto de la poda. Centrémonos en el camino P_{s-J} de la figura 3.4 Todos los nodos terminales a la izquierda han sido explorados y toca decidir si el nodo J ha de ser cortado, es decir, decidir si tiene o no influencia en el valor del nodo raíz. La información necesaria para tomar la decisión se encuentra en los parámetros $\alpha(K), \beta(L), \alpha(M), \beta(N), \ldots$ donde $\alpha(K)$ es el máximo valor de los subárboles descendientes de K hacia la izquierda de P_{s-J}, $\beta(L)$ es el valor mínimo para los correspondientes subárboles descendientes de L, etc.

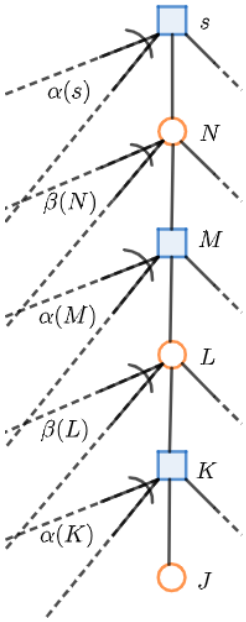


Figura 3.4: Proceso y decisión de la poda para el nodo J.

Supongamos que J es el último nodo en ser evaluado antes de declarar el valor del nodo raíz. En tal caso, ese valor, $\mathtt{e}(s)$ será una función de $\mathtt{e}(J)$. Denotaremos

a esta curva como $\texttt{V}(J) = F_J(x)$. Si $F_J(x)$ es constante, J no afecta al valor de s y puede ser cortado. Llamaremos a F_J función de influencia para el nodo raíz con respecto al nodo J. Notemos que si J realmente influye en la raíz entonces J debe hacerlo a través de sus ascendientes K, L, etc. Así $F_J(x)$ es en realidad una cascada de influencias [20]. Definimos $f_{K,J}$ como la influencia directa del nodo J sobre su nodo padre K con $\texttt{V}(K) = f_{K,J}(\texttt{V}(J))$. Notemos que se tiene la relación por composición

$$F_J(x) = F_K(f_{K,J}(x)) \tag{3.6}$$

Y continuando de manera natural este proceso desde J hasta s se tiene que

$$F_J(x) = f_s(\ldots(f_{L,K}(f_{K,J}))) \tag{3.7}$$

Examinemos detenidamente la función de evaluación padre-hijo $f_{K,J}$. Notemos que esta función está estrechamente relacionada con los parámetros α-β del padre. Si K es un nodo en el que juega `MAX` con parámetro α, entonces

$$f_{K,J}(x) = f^+_\alpha(x) := \text{máx}\{\alpha, x\} \tag{3.8}$$

y equivalentemente para un nodo en el que juega `MIN` con parámetro β, entonces

$$f_{K,J}(x) = f^-_\beta(x) := \text{mín}\{\beta, x\} \tag{3.9}$$

y, por tanto, $F_J(x)$ puede ser escrita como

$$F_J(x) = \ldots f^-_{\beta(N)}(f^+_{\alpha(M)}(f^-_{\beta(L)}(f^+_{\alpha(K)}(x)))) \tag{3.10}$$

La forma de las funciones $f^+_\alpha(x)$ y $f^-_\beta(x)$ y del par

$$L_{\alpha,\beta}(x) := f^-_\beta(f^+_\alpha(x)) = \begin{cases} x, & \alpha \le x \le \beta \\ \alpha, & x \le \alpha \le \beta \\ \beta, & x \ge \alpha \ge \beta \end{cases} \tag{3.11}$$

es mostrada en la figura 3.5. $L_{\alpha,\beta}(x)$ es una función lineal truncada (rampa) por debajo de α y por encima de β.

Teorema 3.5.1. La familia de funciones f_ξ^*, $* = \{+, -\}$, $\xi = \{\alpha, \beta\}$ es cerrada bajo composición.

Demostración. Si componemos $f_{\alpha_1}^+(x)$ y $f_{\alpha_2}^+(x)$ obtenemos otra función de la familia $f_{\alpha_3}^+(x)$ con $\alpha_3 = \text{máx}\{\alpha_1, \alpha_2\}$. Para la familia $f^-(\cdot)$ obtenemos el resultado análogo con $\beta_3 = \text{mín}\{\beta_2, \beta_3\}$ □

Corolario 3.5.1.1. La composición de funciones f_ξ^*, $* = \{+, -\}$, $\xi = \{\alpha, \beta\}$ es conmutativa en el siguiente sentido: al componer f_α^+ con f_β^- obtenemos una función lineal *creciente* truncada para los valores anteriores a α y posteriores a β (véase figura 3.5) mientras que si las componemos en sentido inverso obtendríamos una función lineal *decreciente* y truncada para los mismos valores.

Demostración. Trivial atendiendo a la forma de las funciones descritas y reflejadas en la figura 3.5 y a la noción de conmutatividad dada.

□

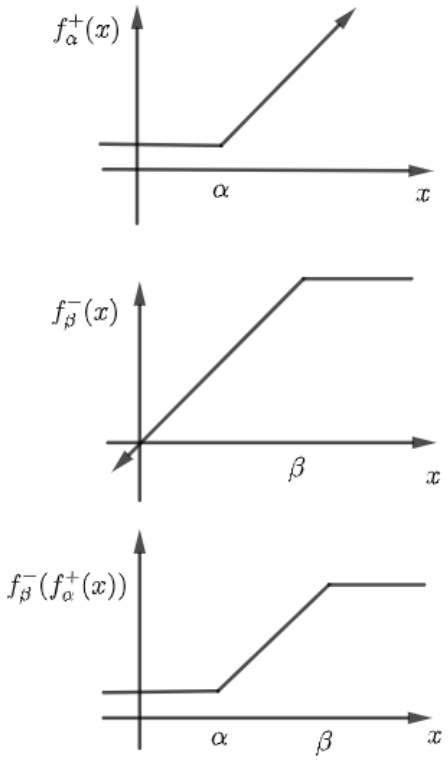


Figura 3.5: Forma cualitativa de las funciones.

Notemos que realmente no nos importa qué tipo de función lineal (creciente o decreciente) resulta de la composición. Lo realmente relevante es dónde se establecen las cotas α y β donde la composición resulta en una constante.

Usando este resultado podemos escribir $F_J(x)$ es una función lineal truncada

$$F_J(x) = f^-_{B(J)}(f^+_{A(J)}(x)) = L_{A(J),B(J)}(x) \tag{3.12}$$

con

$$\begin{aligned} A(J) &= \text{máx}\{\alpha(K), \alpha(M), \dots\} \\ B(J) &= \text{mín}\{\beta(L), \beta(N), \dots\} \end{aligned} \tag{3.13}$$

Por tanto, las funciones de influencia para todos los nodos son funciones lineales truncadas de 45°. Es obvio que $F_J(x)$ será constante si $A(J) \geq B(J)$ en cuyo caso J puede ser cortado. Si no, J debe ser explorado pues su valor aún puede afectar al de la raíz.

Esto da lugar tanto a la condición de corte para α-β como la manera mediante la cual se actualizan las cotas pues α y β son en realidad los parámetros A y B descritos anteriormente. Así, obtenemos una condición necesaria y suficiente para generación (como oposición a la poda) de un nodo J. Como consecuencia hemos probado el siguiente resultado.

Teorema 3.5.2. Sea el nodo J del camino P_{s-J} y sean $A(J)$, $B(J)$ las cantidades definidas en 3.13. Entonces, J será generado por el algoritmo α-β si y sólo si:

$$A(J) < B(J) \tag{3.14}$$

Gracias a esta condición de poda podemos estudiar el rendimiento del algoritmo de poda α-β.

Supondremos que nos encontramos ante el caso de árboles con b sucesores por nodo y realizaremos todo el estudio a una profundidad d. Ya conocemos

gracias Slagle y Dixon [26] que el número de nodos terminales examinados por α-β debe ser al menos $b^{\lceil d/2 \rceil} + b^{\lfloor d/2 \rfloor} - 1$ pero en el peor caso posible puede llegar a ser necesario examinar los b^d nodos. No obstante, también conocemos el resultado de Knuth y Moore (1975) [14] que dice que es posible encontrar una reordenación tal que el algoritmo α-β no necesite buscar más allá del mínimo que acabamos de mostrar. También Fuller, Gaschnig y Gillogly (1973) [9] dedujeron fórmulas para el número medio de posiciones terminales que examinar $I_{\alpha\text{-}\beta}(d, b)$. Mostraremos a continuación el resultado de Pearl que muestra que el factor de ramificación para el algoritmo α-β está acotado inferiormente por la cantidad $\dfrac{\xi_b}{1 - \xi_b}$ con ξ_b raíz positiva de $x^b + x - 1 = 0$ (Baudet (1978). [6]).

Consideremos un nodo arbitrario J en un árbol (d, b, F), esto es, un árbol uniforme de profundidad d, en el que cada nodo no terminal tiene b sucesores y los valores asignados por la función de evaluación a los nodos terminales siguen una distribución de probabilidad F. Ya hemos visto que una condición necesaria y suficiente para que el algoritmo α-β genere J está dada por la inecuación 3.14.

Para obtener el número esperado de nodos terminales examinados por el algoritmo α-β en árboles de juego (d, b, F) necesitamos calcular la probabilidad $P[A(J) < B(J)]$ y sumar para todos los nodos terminales:

$$I_{\alpha\text{-}\beta}(d, b) = \sum_{J \text{ terminal}} P[A(J) < B(J)]. \tag{3.15}$$

Para el caso en el que todos los nodos terminales están ordenados de manera aleatoria y son independientes pero con función de distribución común $F_0(x) = P[V_0 \leq x]$. Si V_k es el valor mínimax del nodo `MIN` al nivel k de profundidad, entonces su distribución F_k se relaciona con el de sus descendientes directos como

$$F_k(x) = 1 - [1 - F_{k-1}(x)]^b \tag{3.16}$$

véase [20] y, de manera recursiva, podemos calcular el resto de niveles de manera que se puede obtener $F_{A(J)}(x)$ y $F_{B(J)}$ para las variables $A(J)$ y $B(J)$ de cualquier nodo terminal J. Además, suponiendo condiciones de regularidad y para

la distribución de valores minimax, podemos calcular la probabilidad anterior mediante la integral de Riemann-Stieltjes [3].

$$P[A(J) < B(J)] = \int_{-\infty}^{+\infty} \sum_{J \text{ terminal}} F_{A(J)}(x) F'_{B(J)}(x)\, dx \tag{3.17}$$

siendo entonces $I_{\alpha\text{-}\beta}(d, b)$ de la forma

$$I_{\alpha\text{-}\beta}(d, b) = \int_{-\infty}^{+\infty} F_{A(J)}(x) F'_{B(J)}(x)\, dx + b^{\lceil d/2 \rceil} + b^{\lfloor d/2 \rfloor} - 1 \tag{3.18}$$

Los términos añadidos representan los nodos críticos (que han de ser examinados). Cada nodo crítico tiene o bien $A(J) = -\infty$ o $B(J) = +\infty$ lo cual no cumple en ningún caso la condición $A(J) \geq B(J)$ razón por la cual han de ser incluidos.

La cantidad $(I_{\alpha\text{-}\beta}(d, b))^{1/d}$ puede ser acotada asintóticamente por $R^*(b) = \dfrac{\xi_b}{1 - \xi_b}$ dando lugar al siguiente teorema probado por Pearl (véase [21]).

Teorema 3.5.3. El número esperado de posiciones terminales examinadas por el algoritmo α-β en la evaluación de árboles de juego (d, b, F) tiene como factor de ramificación

$$R_{\alpha\text{-}\beta} = \lim_{d \to +\infty} (I_{\alpha\text{-}\beta}(d, b))^{1/d} = R^*(b).$$

Este resultado es muy interesante pues supone un óptimo asintótico (véase una vez más el capítulo 8 de [21]) de α-β sobre el conjunto de algoritmos de búsqueda.

Hemos probado que el algoritmo minimax es óptimo para la búsqueda de jugadas dentro del árbol de juego. En el capítulo 4, planteamos un nuevo algoritmo que contrasta con minimax en la manera de obtener la mejor jugada y, aún así, aproxima los resultados dados por minimax con mucha precisión. Detallamos también las ventajas e inconvenientes de ambos algoritmos y planteamos la construcción de métodos híbridos.

Capítulo 4

Árbol de búsqueda Monte Carlo (MCTS).

When you see a good move, look for a better one.

— Dr. Emmanuel Lasker, matemático y 2º campeón del mundo de ajedrez (1894 a 1921).

Hasta ahora hemos estudiado en detalle las características del algoritmo minimax junto a sus mejoras. No obstante, este algoritmo se encuentra con dificultades a la hora de evaluar cierto tipo de posiciones (como veremos en las secciones 4.5, 4.6 y en el capítulo 5). Es por ello interesante explorar otro tipo de algoritmo que tenga filosofía diferente (basada en métodos estocásticos) a la hora de seleccionar la mejor jugada posible y, sin embargo, se aproxime al óptimo dado por minimax. Ese algoritmo es el árbol de búsqueda Monte Carlo (MCTS).

4.1. Introducción.

El árbol de búsqueda Monte Carlo (MCTS) [12, 18, 22] es un método para encontrar decisiones óptimas en un domino dado cuyo resultado es difícil o imposible de resolver usando otras aproximaciones. Su principal característica diferenciadora es que realiza experimentos aleatorios por medio de muestreos para la toma de decisiones, creando un árbol sobre el aplicar una búsqueda con

los resultados obtenidos de los experimentos ejecutados.

Este método ha tenido gran éxito en juegos de tablero que no presentan estrategias ni estructuras conocidas (en los cuales suele ser menos potente la aplicación de algoritmos minimax como los vistos en el capítulo anterior), y ha sido especialmente fructífero en el juego del Go [27], que se conoce especialmente complicado y que ha supuesto el objetivo de la generación automática de jugadores desde que el ajedrez se consideró ya resuelto, así como en otros tipos de juegos que no hemos considerado en el capítulo anterior, como son los juegos con información incompleta o en los que interviene el azar (por ejemplo, juegos de cartas y dados).

Aunque algunos métodos basados en muestreos estadísticos ya eran conocidos en aplicaciones anteriores, una versión primitiva de esta variante que trabaja en la construcción de árboles de juego aparece por primer vez en la tesis doctoral de Bruce Abramsom [1], en la cual se combina una búsqueda mínimax con un modelo estadístico de predicción como sustituto de la función de evaluación, y donde se muestra que el método aproxima de manera precisa, eficiente e independiente del dominio de juego, el valor de un movimiento o jugada.

La idea que sigue este procedimiento la podemos resumir en un algoritmo que viene dado por la repetición (en principio, no acotada) de los siguientes pasos principales[1]:

1. Construimos un árbol de manera incremental, es decir, añadiendo nodos hijos cada cierto tiempo, o cuando se verifica una determinada condición. Además, permitimos que el crecimiento de este árbol sea asimétrico, es decir, no todos los nodos posibles de un nivel son creados a la vez para, más tarde, podar algunas partes como hacía minimax, sino que se crean solo aquellos que parecen más prometedores para la búsqueda que se realiza.

2. En cada iteración del algoritmo usamos una regla para encontrar el nodo más prometedor (aquel que se estime con más probabilidad de dirigirnos

[1] Veremos con más precisión estos pasos más adelante, en la sección que detalla las fases del algoritmo.

a una victoria o a una decisión óptima). Como veremos, esta regla debe ser capaz de mantener un equilibrio adecuado entre la *exploración* (crecimiento del árbol por zonas que no hayan sido analizadas) y la *explotación* (crecimiento de ramas que ya conocemos y que con gran probabilidad llevan al jugador a una victoria), que representa el problema fundamental en todos los procesos de búsqueda y optimización.

3. A partir del nodo recién seleccionado se realiza una simulación, es decir, ampliamos aleatoria y temporalmente (esto quiere decir que los nodos que se creen a partir de este punto no serán definitivos y serán olvidados inmediatamente) el árbol de juego a partir de este nodo hasta encontrar una posición terminal. De esta forma, obtenemos información estocástica (aunque incompleta) acerca de cómo de bueno es el nuevo nodo añadido.

4. Cuando se hayan realizado suficientes simulaciones (este número puede venir determinado por una cota absoluta o una cota temporal) realizamos un movimiento/decisión, es decir, con la información recogida en las simulaciones vemos qué movimiento es el que da lugar a más posiciones terminales favorables.

4.2. Fundamentos.

Para establecer los fundamentos teóricos que hay detrás de este método hemos de hacer una breve visita a la Teoría de Decisión (en particular, veremos los Procesos de Decisión de Markov), la cual combina Teoría de Probabilidad con Teoría Económica de Utilidad[2] para dar un marco general para la toma de decisiones bajo cierta incertidumbre. En el caso concreto que nos ocupa, las decisiones que tengamos que tomar vendrán dadas en forma de secuencias de acciones, como definíamos en el capítulo de juegos finitos. Pasemos pues a definir qué entendemos por proceso de decisión.

[2]En este contexto, la utilidad es el beneficio que se obtiene de adquirir información.

Definición 4.2.1. Un Proceso de Decisión de Markov (MDP, por sus siglas en inglés) modela problemas en los que hay que realizar elecciones secuencialmente y que pueden representarse por medio de cuatro componentes:

$$\begin{cases} S & \text{Conjunto de estados posibles, con } s_0 \in S \text{ siendo el estado inicial.} \\ A & \text{Conjunto de acciones disponibles.} \\ T(s,a,s') & \text{Modelo de transición que determina la probabilidad de alcanzar el} \\ & \text{estado } s' \text{ si la acción } a \text{ se aplica al estado } s. \\ R(s) & \text{Función de recompensa asociada al estado alcanzado } s. \end{cases}$$

Una decisión es un par (s, a) en el cual la probabilidad de llegar al estado s' viene dada por una función de distribución que depende del par.

Obsérvese que, en el caso que nos ocupa, el conjunto de estados posibles será el conjunto de posiciones legales del juego, y el conjunto de acciones posibles será el conjunto de caminos entre nodos padre-hijo, es decir, los movimientos legales que permite realizar el juego. Es evidente el paralelismo que existe entre la estructura de estados y acciones que se puede generar en el contexto de las decisiones y la estructura de árbol de juego que hemos visto en las secciones anteriores de esta memoria.

Definido el problema, ¿cómo sabemos qué acción tomar? Desde el punto de vista de la automatización de la toma de decisiones, lo que se busca es la generación de un sistema capaz de devolver la acción adecuada cuando recibe como entrada el estado actual del problema. En este contexto, esta respuesta viene dada por lo que se denomina una política:

Definición 4.2.2. Una *política*, π, es una aplicación entre estados y acciones que especifica qué acción será elegida del espacio de acciones, A. El objetivo es encontrar la política con mayor recompensa esperada.

Llamaremos Q-valor de una acción a la recompensa esperada de dicha acción:

$$Q(s,a) = \frac{1}{N(s,a)} \sum_{i=1}^{N(s)} \mathbb{I}_i(s,a) z_i$$

donde $N(s, a)$ es el número de veces que la acción a ha sido seleccionada para

el nodo s (a lo largo de todas los experimentos realizados previamente), $N(s)$ es el número de veces que el juego ha contenido al nodo s en un experimento, z_i es el resultado (recompensa en los MDP) del i-ésimo experimento para el nodo s (normalmente, $+1$, -1, o 0 si ha encontrado una posición de victoria, derrota o tablas, respectivamente), y $\mathbb{I}_i(s, a)$ es 1 si la acción a fue seleccionada para el estado s en la i-ésima partida que parte del nodo s, y 0 en otro caso.

Es decir, la política (o estrategia vista desde el punto de vista de juegos finitos) es la aplicación que nos indica cómo debemos expandir el árbol usando la información que ya conocemos e intentando maximizar, con dicha información, el valor esperado de la recompensa. La maximización de dicha recompensa se consigue buscando posiciones terminales ganadoras para el jugador en cuestión.

Esta idea de estudiar distintas partidas dentro del árbol y de recompensar la búsqueda asimétrica dentro del mismo para encontrar posiciones terminales ganadoras será la semilla que germinará convirtiéndose en nuestro algoritmo MCTS.

Un caso especialmente relevante para entender el algoritmo MCTS es el conocido Problema de las Tragaperras Múltiples [5, 15]. En este tipo de problemas un jugador ha de tomar una decisión sobre un conjunto de máquinas tragaperras (o una máquina con varias palancas y/o botones). En concreto, debe decidir en qué máquinas juega, y en qué orden. Cada tragaperras devuelve una recompensa aleatoria y dependiente de la máquina. El objetivo del jugador es maximizar la suma de las recompensas obtenidas tras una secuencia de elecciones sobre el conjunto de las máquinas.

Como se puede observar, este problema puede ser interpretado como un caso particular de MDP, donde las secuencias de acciones consisten en tomar una decisión entre las K posibles acciones (selección de la máquina) para maximizar la recompensa obtenida. Las elecciones tomadas no son triviales pues no conocemos en principio la distribución de la función de recompensa. Para poder resolver este problema desde un punto de vista real, un jugador debería realizar estimaciones con la información que obtenga de experimentos (jugadas o

intentos) ya realizados. Esto lleva a un gran problema, que resolveremos en la sección 4.4, como es el ya comentado sobre el balanceo entre la explotación de la acción que creemos óptima para obtener mayor recompensa con la exploración de otras posibles acciones que parecen no ser óptimas pero pueden llegar a ser mejores en el futuro al estudiar más partes del árbol (e informar, posiblemente, de cómo se comportan otras zonas de juego).

El problema de las K máquinas tragaperras se puede definir formalmente como sigue:

Definición 4.2.3. Supongamos que tenemos una sucesión de variables aleatorias:

$$\{X_{i,n} : 1 \leq i \leq K,\ n \geq 1\}$$

donde $1 \leq i \leq K$ representa cada una de las máquinas, y n es la jugada n-ésima[3].

Fijado el valor de i, la sucesión de variables $X_{i,1},\ X_{i,2}, \ldots$ representa las jugadas sucesivas que se pueden hacer sobre la máquina i-ésima. Vamos a suponer que estas variables son independientes e idénticamente distribuidas siguiendo cada una de ellas una ley desconocida de media μ_i.

Además de la independencia de jugadas dentro de una misma máquina, supondremos que se da la independencia entre máquinas, es decir, que las variables $X_{i,s}$ y $X_{j,t}$ son independientes (aunque probablemente no idénticamente distribuidas) para cualesquiera $1 \leq i < j \leq K$ y cada $s, t \geq 1$.

Nuestro objetivo es encontrar una política (para elegir qué acción/máquina elegir) que maximice la recompensa/ganancia obtenida o, lo que es equivalente, que minimice la *pérdida* posible de no haber elegido la mejor máquina existente en cada una de las jugadas.

Definición 4.2.4. Si $T_i(n)$ es el número de veces que la política π ha seleccionado la máquina i durante las primeras n jugadas, entonces la pérdida se define como:

$$R_N = \mu^* n - \sum_{i=1}^{K} \mu_i E[T_i(n)], \qquad \mu^* = \max_{i=1,\ldots,K} \mu_i$$

[3]Tenemos K sucesiones de decisiones, una para cada máquina, y para cada una de ellas podemos estudiar la jugada o decisión n-ésima.

donde μ^* es la mejor posible recompensa esperada. Es decir, la pérdida es la tasa de derrotas esperadas si elegimos de manera incorrecta una acción.

Notemos que es central no asignar probabilidades nulas a cualquier posible acción en ningún momento para no obviar la solución óptima. Es por ello necesario establecer unas cotas de confianza para la función de recompensa que aseguren que no se dé tal situación.

Para este problema la cota más conocida para obtener una acción óptima es la llamada cota superior de confianza o *upper confident bound* (UCB). Hay varias variantes similares de la misma, pero para nuestro objetivo es suficiente analizar cualitativamente la expresión dada como:

$$UCB1 = \bar{X}_j + \sqrt{\frac{2\log n}{T_j(n)}}$$

donde $\bar{X}_j = \frac{1}{T_j(n)} \sum_{i=1}^{T_j(n)} X_{j,i}$ es la recompensa media de la máquina j-ésima, y n es el número total de partidas simuladas.

Auer [5] probó que, sin conocimiento previo de la distribución de la función de recompensa, UCB1 tenía un crecimiento esperado de la función de pérdida logarítmico (en n). Ese crecimiento coincide con la mejor estrategia teórica para este problema, lo que quiere decir que la diferencia entre la jugada dada por UCB1 con respecto a la mejor jugada posible crece muy poco con el número de jugadas, y tiene el beneficio adicional de ser muy fácil de calcular, pues solo hay que llevar un contador para cada máquina.

El primer sumando de la expresión anterior apoya la explotación de las máquinas con mayor probabilidad de ganar, pues asegura que la recompensa obtenida esté dentro del intervalo. El término de la raíz prima la exploración de los términos menos visitados, pues si $T_j(n)$ es pequeño la cota se dispara, mientras que si es grande la cota es cada vez menor y, por tanto, más precisa.

Las similitudes entre este conocido problema de las tragaperras múltiples y nuestro problema de decisión en el árbol de juego son evidentes, ya que decidir qué acción tomar para transformar adecuadamente nuestro estado de juego es si-

milar a decidir qué máquina de tragaperras elegir para nuestra siguiente partida.

Así pues, es momento ya de dar el siguiente paso y presentar con más detalle el algoritmo MCTS, que veremos que lleva incluido un procedimiento de evaluación en la toma de decisiones que se parece mucho al mecanismo UCB1 visto para este problema.

4.3. Fases y estrategias del algoritmo.

En primer lugar es necesario tener en cuenta que una de las dificultades a las que se enfrenta el algoritmo MCTS, como el de la mayoría de algoritmos de búsqueda, reside en caer en trampas (combinaciones de pocos movimientos que resulten en una posición ganadora para uno de los jugadores) cuando el número de simulaciones no es suficientemente alto por no haber explorado suficientes nodos o hasta suficiente profundidad.

Para solucionar este problema vamos a dar una alternativa al UCB1: el llamado UCT (Upper Confident bound for Trees). Esta cota minimiza los riesgos de caer en este tipo de trampas y se basa en la idea de que si tenemos estadísticas disponibles para todos los estados sucesores del estado actual, entonces podemos extraer una estadística de bondad para el mismo. Esto quiere decir que en lugar de hacer muchas simulaciones aleatorias, esta variante hace iteraciones de un proceso que consta de varias fases y que tiene como objetivo mejorar nuestra información del sistema (exploración) a la vez que potencia aquellas opciones más prometedoras (explotación). Estas fases son las que describen con precisión el algoritmo a la vez que le dan la riqueza y potencia de cálculo necesaria para destacar entre los algoritmos de búsqueda.

Las cuatro fases del algoritmo MCTS son:

1. Selección: El objetivo de esta fase es encontrar el nodo hijo más prometedor desde el estado en el que nos encontremos. Para ello, se recorre el árbol desde el nodo raíz hasta una posición posterior yendo de nodo prometedor en

nodo prometedor por el camino que los une. Va a caballo entre exploración (lo cual evita convergencia demasiado rápida a una trampa) y explotación de los nodos más prometedores.

Para valorar qué nodo seleccionar, aplicamos UCB1 la primera vez que un nodo sea visitado. A partir de ese momento el objetivo es maximizar la siguiente fórmula UCT:

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

donde $N(s_0)$ es el número de veces que el nodo s_0 (padre de s) ha sido visitado, $N(s)$ es el número de veces que se ha visitado el nodo hijo, s, y $Q(s)$ es la recompensa total de las partidas que pasan a través del nodo s (C_p es una constante que debe ser ajustada, y que puede depender de las características del juego).

Aquí la primera fracción representa la recompensa media de todas las jugadas. Cuando $N(s) = 0$, es decir, cuando el sucesor nunca se ha visitado, el valor UCT es ∞ y, por tanto, sería elegido por la estrategia. Para evitar este problema, esta fórmula solo se utiliza en caso de que todos los hijos hayan sido visitados al menos una vez. Además, como con UCT se busca un equilibrio entre la explotación y exploración, el factor $\frac{Q(s)}{N(s)}$ decrece a medida que cada estado se visita. Por otra parte, el término de exploración $\sqrt{\frac{\log N(s_0)}{N(s)}}$ crece cuando se visita otro sucesor de su nodo padre (hermano suyo). Así, aseguramos que incluso sucesores con recompensas bajas sean seleccionados si se les da el tiempo suficiente (y evitamos caer en trampas). La constante C_p se escoge para ajustar el nivel de exploración. En [15] demostraron que $C_P = 1/\sqrt{2}$ es un valor óptimo cuando las recompensas se encuentran en el intervalo $[0, 1]$. En caso contrario, se deben determinar experimentalmente. Además, Kocsis y Szepesvari demostraron en [15] que la probabilidad de encontrar una acción no óptima para la raíz del árbol tiende a 0 a velocidad polinomial si el número de iteraciones tiende a infinito, es decir, UCT$\rightarrow$ Minimax.

Todas estas consideraciones serán detalladas en la siguiente sección. De mo-

mento nos basta una idea superficial para entender las fases del algoritmo.

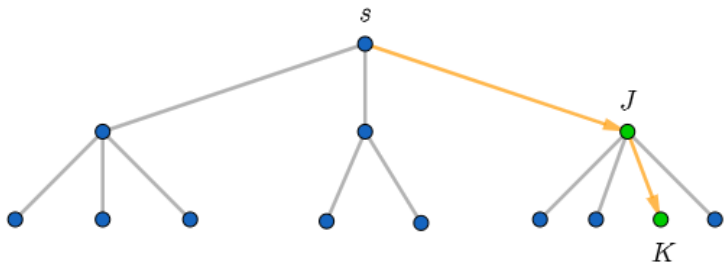


Figura 4.1: Fase de selección. A partir del nodo raíz s se ha elegido el nodo J como más prometedor. A continuación, se realizan el resto de fases del algoritmo y se vuelve a elegir el siguiente nodo prometedor K. El proceso se repite tantas veces como sea posible en el tiempo disponible.

2. Expansión: Esta fase es más sencilla conceptualmente. Simplemente añadimos un nodo o posición nueva al árbol para la posterior simulación.

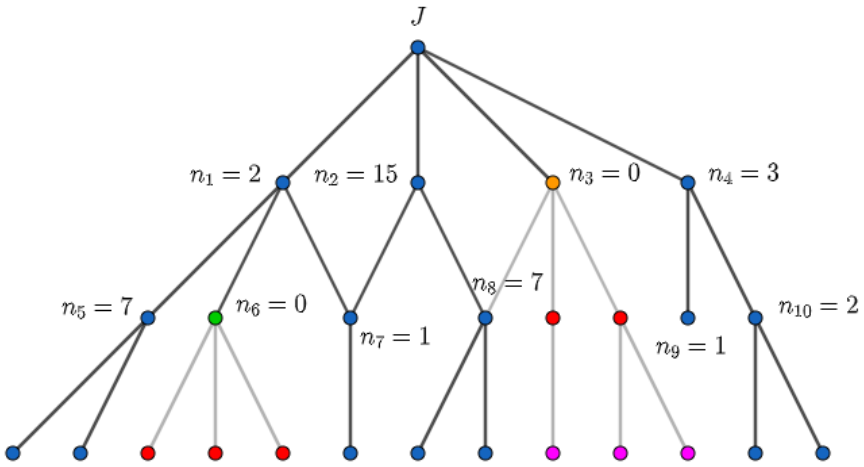


Figura 4.2: Fase de expansión. Si nuestra estrategia de expansión es añadir nodos sin visitas previas, el nodo con $n_3 = 0$ es el nodo al que expandirse desde J y el nodo con $n_6 = 0$ es el que expandirse desde el que tiene contador de visitas $n_1 = 2$. Los sucesores de estos nodos están marcados en rojo y no han sido visitados y serán susceptibles de ser añadidos al árbol cuando vuelva a realizarse la expansión.

Algunas estrategias de expansión estudiadas por la bibliografía discuten sobre la posibilidad de expandirse a un nodo sea cual sea el número de veces que

haya sido visitado antes. Otras referencias proponen expandirse a un nodo sólo a partir de cierto número mínimo de visitas e incluso en algunos casos se discute la posibilidad de expandirse a más de un nodo en cada iteración.

3. Simulación: Comenzamos en el nodo que queremos simular. A partir de él, tomamos distintas opciones aleatorias disponibles en el conjunto de jugadas legales. Con ellas formamos caminos buscando posiciones terminales. Esta fase es costosa pues pueden existir muchas jugadas legales posibles. Por ello, es posible intentar buscar hasta una cierta profundidad o tiempo acotado (pero entonces ha de disponerse de una función de evaluación que indique la bondad del estado no terminal que se alcance). Este punto es una debilidad añadida al algoritmo pues la fuerza del mismo consiste en encontrar posiciones terminales, no siendo necesaria ninguna información adicional sobre el juego.

Hemos de tener también en cuenta que debemos realizar todas las simulaciones posibles para el nodo, con el fin de minimizar la probabilidad de caer en trampas o errores y proporcionar un mejor valor estimado del nodo simulado, como ya hemos comentado en varias ocasiones.

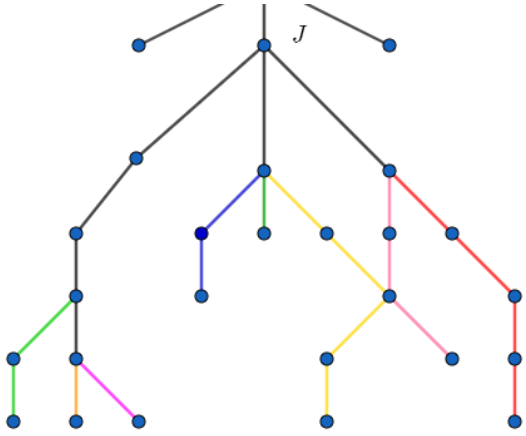


Figura 4.3: Fase de simulación para el nodo J. Cada color refleja un camino simulado diferente. Las zonas de intersección se reflejan en negro por superposición. Recordemos que no todos los nodos que aparecen en el nivel del nodo J tienen por qué estar incluidos en el árbol de juego.

4. Retropropagación: En esta fase se actualiza el valor de los nodos previos con la información obtenida en la simulación. Para ello, se procede de la siguiente forma: se actualiza primero el nodo hoja, luego el padre de éste y así sucesivamente hasta alcanzar la raíz. En cada actualización se suma 1 al contador $N(s)$ y se actualiza el valor del nodo usando el resultado de la simulación.

Las estrategias más comunes en la literatura estudian el caso de retropropagación: usando el valor máximo de todos los resultados (similar a minimax), usando la media, o por combinación de varias técnicas.

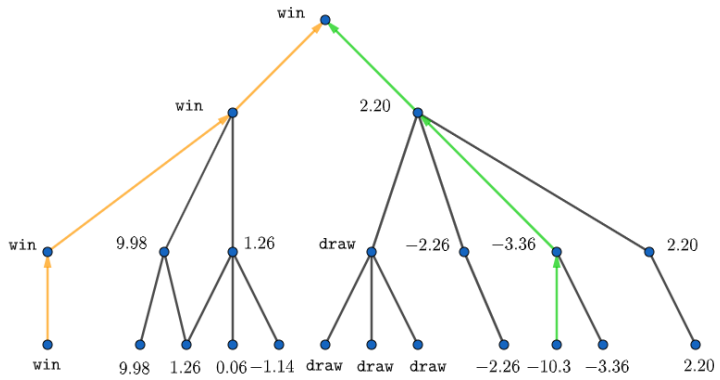


Figura 4.4: Fase de retropropagación. En este caso la actualización del nodo del nivel 4 marcado como `win` actualiza el valor de todos sus antecesores, es decir, el algoritmo ha encontrado una estrategia ganadora. Otro ejemplo es el camino marcado de verde. Se usa la estrategia de actualización dada por el máximo. Vemos que en este caso el jugador tiene clara oportunidad de ganar el juego o, al menos, de tener una situación favorable. No obstante, existen caminos cercanos que dejan al jugador en una mala posición, p.e., los sucesores del nodo marcado con 2,20 del segundo llevan o bien al empate o bien a una posición desfavorable en la mayoría de casos. Este tipo de *trampas* son las que pueden hacer que el algoritmo falle en ciertas ocasiones si no encuentra un camino favorable.

Cuando el tiempo de simulación haya concluido, el movimiento elegido es el más prometedor, dada la información obtenida en las anteriores fases. De nuevo, existen varias estrategias para la elección del mismo, siendo lo más común la de balancear la valoración de los nodos con el número de veces que han sido visitados para la siguiente selección.

Una vez seleccionado el nodo el proceso se repite hasta alcanzar un tiempo fijo de computación, y se realiza la mejor jugada hasta el momento.

Por último, el algoritmo completo con sus fases puede ser representado con pseudocódigo de la siguiente manera.

Algoritmo 6 Pseudocódigo del algoritmo MCTS.

Entrada: Posición inicial s_0.
Salida: Nodo seleccionado.

```
1:  function MCTS-UCT(s_0)
2:  Crear nodo raíz n_0 con estado s_0
3:      while Existan recursos computacionales do
4:          n ← Selección(n_0)
5:          G ← Simula(s(n))
6:          Retropropaga(n, G)
7:  Devuelve a(MejorHijo(n_0)).
8:
9:  function Selección(n)
10:     while s(n) no sea terminal do
11:         if nno está completamente expandido then
12:             Devuelve Expande(n)
13:         else
14:             n ← MejorSucesorUCT(n, c)
15:     Devuelve n.
16:
17: function Expande(n)
18:     a ← elige acción no probada de Acciones(s(n))
19:     Añade un nuevo sucesor m de n.
20:     s(m) ← aplica(a, s(n))
21:     a(m) ← a
22:     Devuelve m.
23:
24: function MejorSucesorUCT(n)
25:     Devuelve argmax(UCT(n))
26:
27: function Simula(s)
28:     while a no sea terminal do
29:         a ← un elemento de A(s)
30:         s ← f(s, a).
31:     Devuelve recompensa del estado s.
32:
33: procedure Retropropaga(n, G)
34:     while n ≠ 0 do
35:         N(n) ← N(n) + 1
36:         Q(n) ← Q(n) + G
37:         n ← Padre de n.
```

Definidas las fases vamos a desarrollar los detalles concernientes a cómo las cotas realizan la labor de la compensación entre la exploración y explotación de nodos del árbol de juego en el algoritmo MCTS.

4.4. Exploración vs explotación.

Como hemos comentado, los métodos Monte Carlo se han usado en juegos como los que ya hemos comentado. Simulando una gran cantidad de posibles partidas, la jugada realizada en cada turno es aquella con la mayor probabilidad de victoria. En la mayor parte de los casos, las acciones empleadas para realizar las simulaciones se estudian usando experimentos aleatorios uniformes en todas las posibles ramas del árbol de juego pero sin garantías de encontrar el óptimo.

No obstante, podemos mejorar las simulaciones mediante las cotas de confianza y las funciones de recompensa y pérdida. Nos centraremos en dar fundamento teórico a esas cotas ahora, concretamente en referencia al buen balanceo entre exploración y explotación de los nodos. La diferencia principal (y muy importante, ya que complica considerablemente los cálculos asociados) entre el caso de Monte Carlo para las máquinas tragaperras vistas anteriormente (y para el que habíamos dado una cota relativamente eficiente por medio de UCB1) y el caso de la generación dinámica de los árboles de juego, es que en estos últimos las diversas variables aleatorias a una misma rama de juego no son independientes ni idénticamente distribuidas, por lo que no es tan fácil probar la convergencia del algoritmo a un óptimo. A pesar de ello, veremos que se probó esta convergencia por medio de una ligera variante del algoritmo anterior.

La clave del éxito para el MCTS vino de manos de Kocsis y Szepesvári [15] gracias al mecanismo de selección que propusieron. Tenía una pequeña probabilidad de error si el algoritmo tenía que detenerse antes de realizar un número suficiente de simulaciones y convergía al óptimo con el suficiente tiempo.

Si recordamos la ecuación

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

decíamos que el primer sumando debía ser entendido como la esperanza de la recompensa dentro del intervalo $[0, 1]$. Con esa consideración, la constante C_p del término de exploración se puede calcular para que satisfaga la desigualdad de Hoeffding [15], necesaria para probar los teoremas que aseguran la convergencia del método.

Los teoremas presentados a continuación demuestran la potencia real del algortimo MCTS con la cota UCT y la convergencia a minimax. Más allá de la forma de las cotas que se van a dar a continuación (para lo cual habría que entrar en bastante más detalle), la importancia de dichos resultados reside en la interpretación de los mismos.

Como hemos visto en la presentación anterior del algoritmo, los valores que etiquetan los nodos internos se calculan promediando las recompensas acumuladas de las jugadas que comienzan con ese nodo. A partir de estos valores se calcula el peso de la acción que genera ese estado/nodo, por tanto, la convergencia de esos promedios determina la convergencia de los valores asignados a las acciones (y que son, finalmente, los que deciden la acción a tomar). El siguiente teorema da una cota de esta convergencia que es fundamental para asegurar el buen comportamiento de este paso del algoritmo:

Teorema 4.4.1. Sea $\overline{X}_n = \sum_{i=1}^{K} \frac{T_i(n)}{n} \overline{X}_{i,T_i(n)}$. Entonces

$$|E[\overline{X}_n - \mu^*]| \leq |\delta_n^*| + \mathcal{O}\left(\frac{K(C_p^2 \log(n) + N_0)}{n}\right)$$

Este teorema nos dice que las recompensas medias de los nodos internos se concentran rápidamente en torno a sus medias teóricas.

El siguiente teorema muestra que el número de veces que una máquina es usada (número de veces que se elige una acción en un nodo dado) puede ser

acotado inferiormente por el logaritmo del número de veces que es visitada (número de veces que el nodo es visitado):

Teorema 4.4.2. Existe una constante positiva ρ tal que, para todas las máquinas i y para todo n, $T_i(n) \geq \lceil \rho \log(n) \rceil$

Este resultado junto al anterior nos ayuda a generalizar el hecho de obtener recompensas próximas a las medias teóricas rápidamente para los nodos. Formalmente:

Teorema 4.4.3. Sean $\delta > 0$ fijo y $\Delta_n = 9\sqrt{2n\log(2/\delta)}$. Las siguientes cotas se dan si n es suficientemente grande.

$$P(n\overline{X}_n \geq nE[\overline{X}_n] + \Delta_n) \leq \delta,$$

$$P(n\overline{X}_n \leq nE[\overline{X}_n] - \Delta_n) \leq \delta.$$

Damos también una cota para la probabilidad de fallo del algoritmo para el nodo raíz:

Teorema 4.4.4. Sea $\hat{I}_t = \arg\max_i \overline{X}_{i,T_i(t)}$. Entonces, existe una costante C tal que:

$$P(\hat{I}_t \neq i^*) \leq C\left(\frac{1}{t}\right)^{\frac{\rho}{2}\left(\frac{\min_{i \neq i^*} \Delta_i}{36}\right)^2}$$

En particular, se tiene que:

$$\lim_{t\to\infty} P(\hat{I}_t \neq i^*) = 0,$$

Es decir, la probabilidad de fallo del algoritmo UCB1 para la selección de un nodo prometedor tiende a 0 cuando el número de visitas tiende a infinito (aquí i^* representa la elección óptima).

Estamos en condiciones de presentar el correspondiente teorema de convergencia para UCT, el cual justifica la importancia de este algoritmo de selección de nodos:

Teorema 4.4.5. Consideremos el algoritmo UCT ejecutándose en un árbol de juego de profundidad D, con un máximo de K acciones posibles para un

estado (ramificación máxima), y con recompensas estocásticas en las hojas en el intervalo $[0, 1]$. Entonces, el sesgo de la recompensa estimada, $E[\overline{X}_n] - \mu^*$, es $\mathcal{O}((KD\log(n) + K^D)/n)$. Además, la probabilidad de fallo en el nodo raíz converge a cero a velocidad polinomial cuando $n \to \infty$ (el número de muestras crece).

4.5. Fortalezas y Debilidades.

MCTS es potente en ciertos dominios, pero no en otros [4, 19, 22] pues existen distintos grados de aplicabilidad.

Puede fallar cuando exista una única variante ganadora, porque se haya visitado menos veces que los nodos vecinos y la cantidad de simulaciones no haya permitido encontrar la variante completa, si bien es cierto que puede parar en cualquier momento y devuelve el mejor valor que haya calculado hasta ese instante.

Además, es más fácil que caiga en trampas o jugadas erróneas si no ha realizado suficientes simulaciones por problemas de tiempo [4].

Sin embargo, es especialmente útil en ciertas ocasiones donde algoritmos como minimax se encuentran con problemas. En juegos con un gran factor de ramificación existe el problema de asignar con seguridad un valor estático (al aumentar la profundidad poco a poco) a un nodo no terminal. Nos gustaría que, si bien el valor real del nodo no pudiéramos estimarlo con precisión, al menos sí poder saber que no va a variar mucho. El problema con las funciones de evaluación usadas es que, al tener que operar por niveles y de manera simétrica, no puede alcanzar posiciones terminales tan rápido como MCTS. Mostraremos algunos ejemplos de este tipo para el caso del ajedrez en el siguiente capítulo.

Además, en juegos como Go, es muy difícil determinar de antemano si una pieza será o no capturada, o si un grupo estará o no seguro en cierta zona del tablero. El crecimiento asimétrico y la función de recompensa son capaces de

solventar de manera más rápida ese tipo de situaciones al no tener que considerar características propias del juego. El algoritmo sólo necesita saber las reglas del juego (jugadas legales y posiciones terminales). Las estrategias ganadoras serán obtenidas de manera natural por el algoritmo, como veremos a continuación. Esto supone una gran ventaja como motor de juego general (al no necesitar conocimiento específico más allá de las jugadas legales) y, en especial, para juegos estocásticos.

4.6. Mejoras.

Una ventaja indiscutible del algoritmo MCTS es que permite la hibridación de distintas técnicas, como pueden ser el uso del algoritmo minimax para la evaluación de estados intermedios y la reutilización de conocimientos generales sobre el dominio de juego.

La hibridación de métodos resulta útil para resolver problemas que no son fácilmente tratables con distintas técnicas por separado. Este hecho facilita que tenga un mayor rendimiento en juegos como el Go, en los que es improbable obtener una posición claramente perdedora en pocas jugadas desde un estado dado, frente a otros juegos, como el ajedrez, en los que pequeñas combinaciones de 3 o 4 jugadas pueden resultar en victoria asegurada para uno de los jugadores.

El conocimiento previo, dado por el humano o como consecuencia del aprendizaje automático (del cual hablaremos un poco las conclusiones), puede ayudar a reducir el número de cálculos y posibles fallos en este tipo de problemas. En el caso del ajedrez, el conocimiento teórico humano sobre distintas aperturas y estructuras sirve para avanzar en aproximadamente las primeras 15 jugadas sin tener que realizar grandes cálculos y evitando pequeñas trampas conocidas que puedan suponer una diferencia de tiempo importante. De igual manera, la inclusión de las tablas de Nalimov (o finales) suponen un algoritmo directo y teórico para jugar los finales con menos de 7 piezas. De esta manera, el cálculo se puede centrar en los nodos internos (el medio juego) hasta alcanzar una posición teórica conocida que no necesitemos seguir explorando.

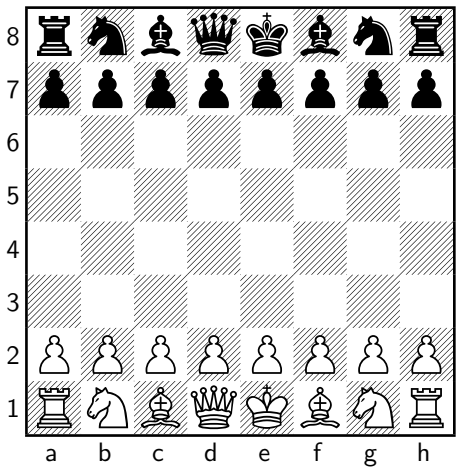
Capítulo 5

Aplicación al ajedrez.

It is important to draw wisdom from different places. If you take it from only one place it becomes rigid and stale.

— Iroh, 'El dragón del oeste'.

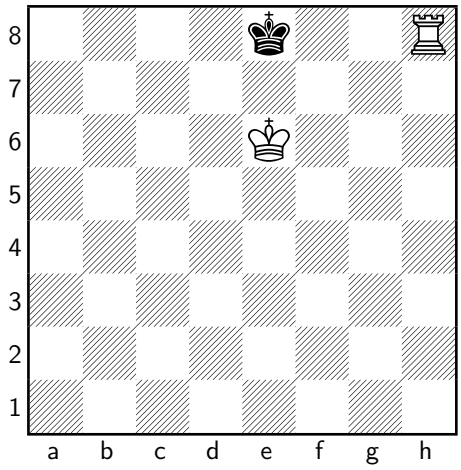
El objetivo de este capítulo es poner en contexto todos los conceptos ya desarrollados en capítulos anteriores para el juego del ajedrez. Para comenzar, nuestro árbol de juego parte del nodo raíz s que en este caso es la conocida posición inicial de ajedrez.



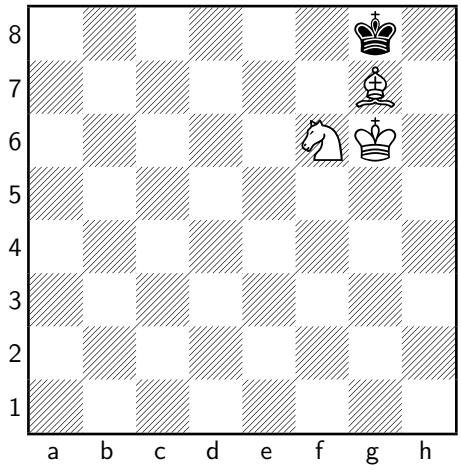
Tablero 1. Nodo raíz para el ajedrez.

Usaremos la notación algebraica para referirnos a las jugadas del tablero. El lector puede encontrar una breve explicación de esta notación en el Anexo 1.

El objetivo de ambos jugadores en el ajedrez es dar jaque mate al otro jugador, es decir, amenazar el rey en la casilla en la que se encuentre e impedir su movimiento (legal) a cualquier casilla a su alcance. Ejemplos básicos y conocidos son los siguientes mates:



Tablero 2. Mate de torre.



Tablero 3. Mate de alfil y caballo.

Para alcanzar posiciones de este tipo hemos de recorrer todo el árbol de juego y obtener una estrategia ganadora como ya se ha descrito. Profundizaremos a continuación en cómo evalúan las posiciones los algoritmos para obtener la mejor jugada posible. Nos detendremos en primer caso en la piedra angular del algoritmo α-β, la función de evaluación.

Antes de ello, como divertimento matemático y ajedrecístico, vamos a introducir el concepto de ajedrez aleatorio, ajedrez de Fisher o ajedrez 960. Esta variante del ajedrez no tiene siempre el mismo nodo raíz dado en el Tablero 1 sino que, si bien la posición de los peones permanece inalterada, la del resto de piezas en la primera y octava fila se elige de modo aleatorio. Fue propuesta por el mismo Bobby Fisher en busca de evitar la preparación teórica de posiciones en la apertura ($\sim$ 15 primeras jugadas) algo que ocurre tanto a nivel humano [17] como a nivel máquina con la inclusión de grandes bases de datos para las aperturas en busca de impedir el avance de las máquinas frente a los humanos en este juego.

El nombre de ajedrez 960 proviene de la cantidad de posiciones diferentes como nodo inicial posibles. Veamos cuál es el algoritmo de posicionamiento de las piezas y realicemos el cálculo combinatorio.

En primer lugar, se debe cumplir que los alfiles estén en casillas de colores diferentes y el rey debe situarse entre las dos torres para poder realizarse el enroque. Además, la posición ha de ser simétrica, es decir, el jugador de piezas blancas y el jugador de piezas negras tendrán colocadas sus piezas en la misma posición.

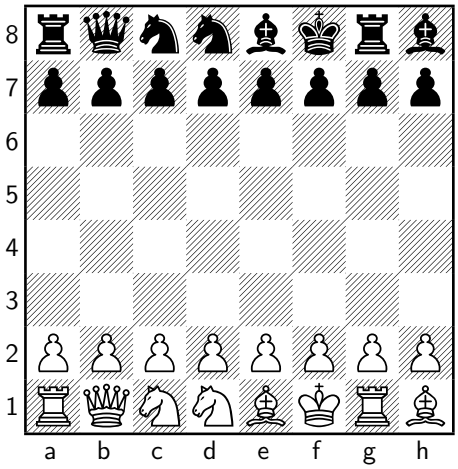
Un posible algoritmo para realizar el montaje del juego es el siguiente. Colocaremos para empezar el alfil de casillas negras (A_n) en una de las cuatro disponibles. A continuación colocaremos el alfil de casillas blancas (A_b) en alguna de las 4 casillas blancas disponibles. Después realizaremos el mismo proceso con la dama D (6 posibilidades) y los caballos C_b y C_n (5 y 4 posibilidades respectivamente). En este momento sólo quedan 3 casillas disponibles. Dispondremos las piezas restantes de manera que el rey esté entre las dos torres.

Realizando el cálculo según la regla del producto, el número de combinacio-

nes posibles es

$$C = C_{A_n} \cdot C_{A_b} \cdot C_D \cdot C_{C_b} \cdot C_{C_n} = 4 \cdot 4 \cdot 6 \cdot 5 \cdot 4 = 1920 \text{ posiciones.}$$

Tengamos en cuenta que en realidad como la posición de los caballos es totalmente intercambiable así que tenemos que dividir este número por 2 para obtener el número total de posiciones diferentes, obteniendo las 960 que dan nombre a esta variante.



Tablero 4. Posición aleatoria para la variante Ajedrez 960.

5.1. El núcleo del algoritmo α-β: la función de evaluación.

Como primera aproximación podemos definir la función de evaluación como una combinación lineal de m características.

$$\mathsf{e}(x) = \omega_1 \cdot f_1(x) + \omega_2 \cdot f_2(x) + \cdots + \omega_m \cdot f_m(x)$$

donde ω_i es el peso asociado a la característica $f_i(x)$. Una versión muy básica de función de evaluación consistiría en considerar las diferencias de material, es decir, la diferencia en número de piezas de cada uno de los jugadores.

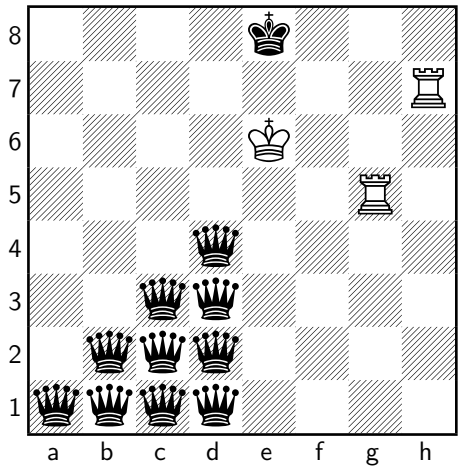
Tengamos en cuenta que suele ser habitual normalizar el valor de las piezas

Pieza	Valor $\omega_i(x)$	Característica (diferencia de material) $f_i(x)$
Dama (D)	9	#{damas blancas- damas negras}
Torre (T)	5	#{torres blancas- torres negras}
Alfil (A)	3	#{alfiles blancos- alfiles negros}
Caballo (C)	3	#{caballos blancos- caballos negros}
Peón ()	1	#{peones blancos- peones negros}

Tabla 5.1: Valor de las piezas en ajedrez con normalización respecto al peón. Se añaden además los símbolos que representan a todas las piezas del tablero a excepción del rey cuyo símbolo es **R** y de los peones, que no tienen símbolo asignado sino que su posición se declara mediante las coordenadas en el tablero.

(y de la posición en general) con respecto al valor del peón. Un peón vale teóricamente 100 centipeones, siendo esta unidad la mínima usada para evaluar una posición.

Notemos que esta función de evaluación no es fiable pues, como vemos en el siguiente ejemplo, es posible tener gran ventaja de material y aún así encontrarse ante una posición perdida.



Tablero 5. En este caso extremo e irreal, el jugador negro tiene gran ventaja de material. La función de evaluación le daría una ventaja de $9\cdot(0-10)+5\cdot(2-0) = -80{,}00$. No obstante, el jugador blanco puede dar mate con el movimiento **Tg8#**.

Es evidente que debemos tener en cuenta más características para que la función de evaluación realmente estime bien el estado de la posición. Desarro-

llamos brevemente alguna de las mejoras posibles a la función de evaluación con especial interés del caso de la poda α-β, que es la que realmente necesita evaluar todas las características de los nodos para estimar el estado de la posición.

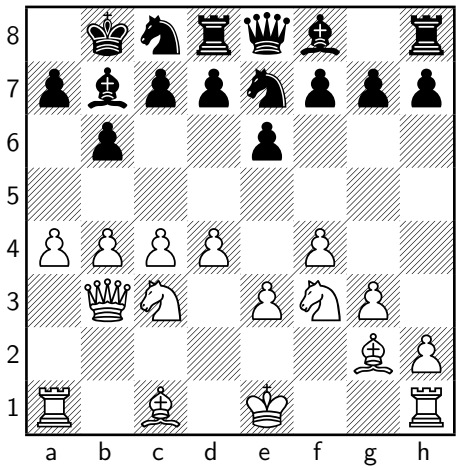
1. Evaluación gradual: En este tipo de evaluación, los pesos ω_i son funciones del tiempo $\omega_i(t)$ (esto quiere decir que dos posiciones iguales pueden tener distintas valoraciones si se producen en distintos momentos del desarrollo del juego). Normalmente, no se usa un grano tan fino como es el número de la jugada, sino que se distinguen tres fases principales de la partida: apertura, medio juego y final. También podrían ser variables en función del número de piezas restantes en el tablero, etc.

2. Pareja de alfiles: Se añade un bonus ($\sim$ 0,5 peones) por tener ambos alfiles frente a los dos caballos, o un alfil y un caballo, especialmente en la parte final de la partida y en posiciones abiertas (con pocos peones).

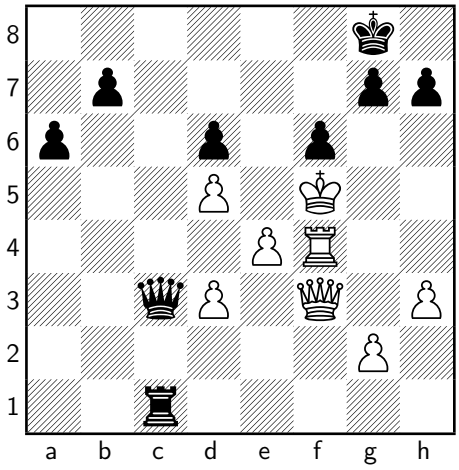
3. Tabla de piezas y casillas: Se puede asignar un valor específico que considere qué pieza está en qué parte del tablero. Por ejemplo, los caballos son muy potentes en las cuatro casillas centrales, mientras que pierden fuerza en el borde. De igual modo, piezas sin movilidad (por impedimento de otras piezas de menor valor y/o propias) serán penalizadas.

4. Seguridad del rey: Teniendo en cuenta el objetivo final del juego, se puede valorar hasta qué punto el rey está asegurado por la posición que ocupa él y las piezas que le rodean. Por ejemplo, contando el número de peones cercanos o piezas mayores.

5. Estructura de peones: En este caso se introduce una penalización para el caso en el que encontremos dos o más peones en la misma columna, así como una bonificación por peones pasados (sin oposición por otro peón en su columna ni en las adyacentes).



Tablero 6. Las blancas cuentan con gran cantidad de espacio tras 12 jugadas. El material está igualado y aún así Stockfish a profundidad 23 da una ventaja de 2.1 a las blancas.

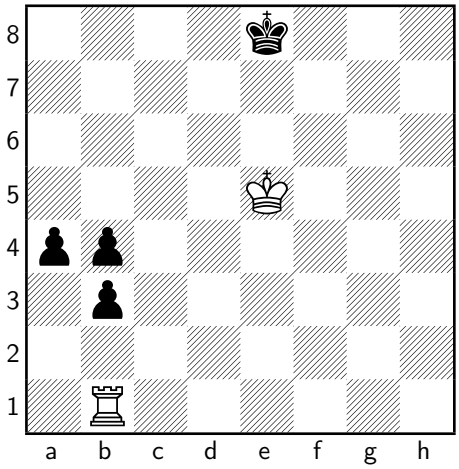


Tablero 7. Las negras cuentan con un único peón de más. Sin embargo, Stockfish a profundidad 23 da una ventaja de -5.6 a las negras como consecuencia de la mala estructura de peones de las blancas y la inseguridad de su rey.

Además de estas, vamos a estudiar algunos efectos y estrategias cualitativamente diferentes:

El efecto horizonte.

Uno de los grandes problemas computacionales a la hora de evaluar una posición es decidir hasta qué profundidad hemos de mirar el árbol de juego que se deriva de la posición actual [7, 23]. En el siguiente diagrama las blancas acaban de jugar **1. Tb1** y podría parecer para nuestro módulo de ajedrez una posición muy favorable al blanco pues todas las características anteriores así lo confirman: tenemos más material y buena posición general, pues hay peones pasados negros pero la torre blanca los controla. Si el blanco hubiera calculado hasta profundidad 2, antes de realizar el movimiento de torre, podría pensar que es inevitable que gane, pero en realidad la posición está completamente perdida tras 1. Tb1 a3. 2. Txb3 a2, y ahora el peón de columna a es imparable puesto el peón de b4 controla la casilla a3 y el peón de a2 controla la casilla b1. Es inevitable la coronación del peón llegando a un final ganado teóricamente por las negras.



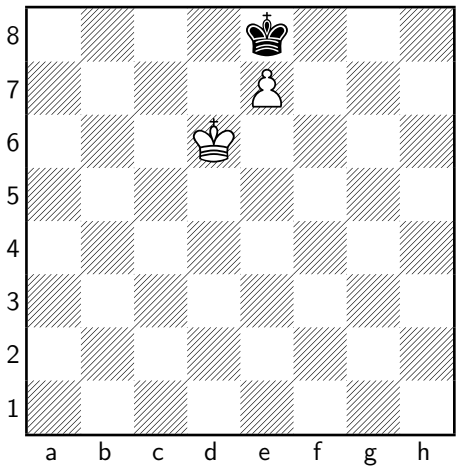
Tablero 8. Efecto horizonte.

El software de Stockfish[1] a profundidad 23 en la web Lichess[2] consigue calcular que la mejor jugada no era colocar la torre frente a los peones sino acercar el rey para capturarlos apoyado por la torre.

[1] Campeón del TCEC en varias ocasiones.
[2] Web de código abierto para jugar y entrenar ajedrez.

Movimiento nulo, movimiento asesino y quiescencia.

El movimiento nulo es una estrategia propuesta por Goetsch y Campbell [10] para aumentar el número de podas realizadas con α-β. Consiste en lo siguiente: Supongamos que vamos a evaluar un nodo tipo `MAX`. Evaluamos el nodo suponiendo que dejamos mover al otro jugador sin realizar nosotros ninguna jugada. Si la posición obtenida no es peor que la anterior tenemos una cota inferior de confianza para el valor del nodo. Si el valor es igual o mayor que el valor β del nodo padre podemos podar y no revisar ninguno de los nodos hermanos (del mismo nivel). Esta técnica es interesante y muy útil, pero tiene un punto débil, especialmente para el ajedrez, y son las llamadas posiciones *zugzwang*, aquellas en las que cualquier movimiento es peor que no hacer nada (véase el siguiente tablero).

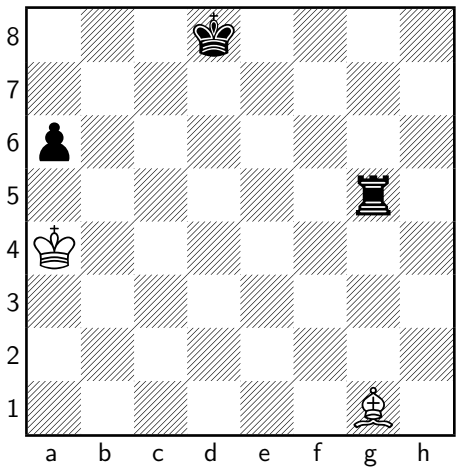


Tablero 9. En este caso ambos jugadores están en zuzgwang. Si mueven las negras, las blancas coronan el peón en dos jugadas y ganan. Si mueven las blancas, o bien el rey negro se queda ahogado (sin movimientos legales) o bien se captura el peón blanco, se darían las tablas y las blancas no ganarían.

En contraposición, el movimiento asesino trata de estudiar primero las jugadas que supongan capturas de piezas con alto valor así como posibles jaques mate.

Existe, además, la estrategia que busca priorizar la evaluación de caminos de jugadas donde existan intercambios de piezas o amenazas de mate hasta que lleguen a una posición en la que no sean posibles. Estas posiciones son llamadas quiescentes.

Por último, existe la técnica conocida como **'de perdidos al río'**. Se basa en intentar aprovechar posibles errores del adversario. Supongamos que estamos en una posición perdedora (valores α-β bajos). En este caso, la mejor estrategia es no realizar la mejor jugada sino la que más posibles errores pueda provocar al adversario.

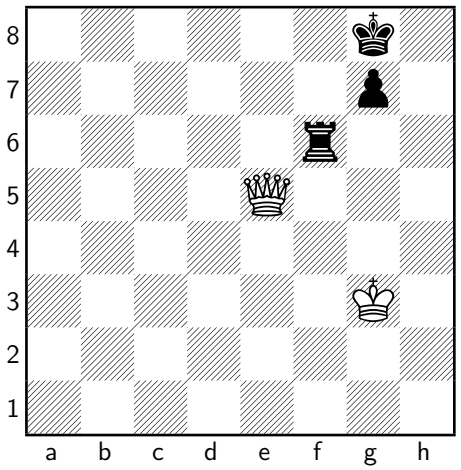


Tablero 10. En este caso la mejor jugada según Stockfish a profundidad 23 es **Ae3** con evaluación -4.6, es decir, las blancas están perdiendo. Sin embargo, si jugáramos **Af2** con evaluación -4.8 las blancas podrían jugar erróneamente **a5** permitiendo **Ah4** capturando la torre (que no puede mover pues cubre al rey) y entablando la partida.

5.2. La potencia de MCTS. Fortalezas y tablas en ajedrez.

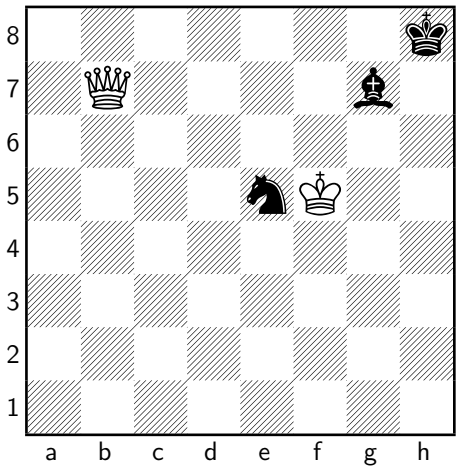
El uso más interesante del MCTS en ajedrez resulta en dos tipos de posiciones opuestas por su naturaleza.

El primero de estos tipos son las llamadas fortalezas, estas son, posiciones en las que uno de los jugadores posee menos piezas o de valor menor que el adversario y, pese a ello, es capaz de defenderse sin llegar a perder la partida.



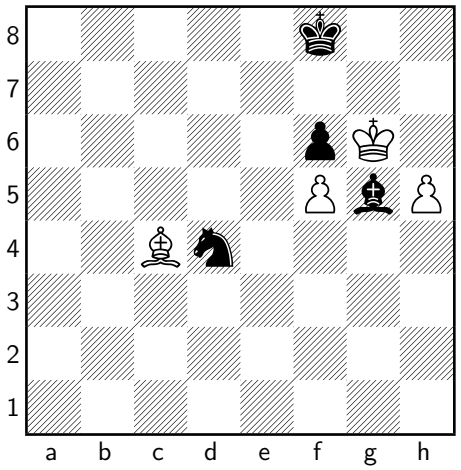
Tablero 11. Ejemplo de fortaleza. Las negras sólo deben mover su torre de f6 a h6 o f8 y las blancas no podrán hacer nada para progresar.

En este tipo de posiciones los métodos de evaluación y búsqueda como minimax tienen más problemas debido al problema del horizonte mencionado anteriormente. Existen posibilidades para hacer tablas, pero suele ser necesario evaluar muchos más niveles del árbol de las que el algoritmo puede manejar. De este modo, la evaluación siempre será superior para el bando con más piezas o de mayor valor, pero no será capaz de ganar la partida. No obstante, MCTS al no encontrar ningún nodo próximo con una posición final por ningún camino evaluará la posición como tablas.



Tablero 12. Ejemplo de fortaleza más compleja. Las negras tienen todas sus piezas protegidas, impiden la llegada del rey a las casillas g6, h6, f7 y f8. La dama blanca puede dar jaques pero el rey negro siempre va a pivotar entre las casillas g8 y h7 y nunca se va a poder dar mate.

Como caso opuesto, en posiciones con algunos caminos ganadores para uno de los jugadores, pero también fuera del horizonte, minimax evaluará una posición como tablas o como ligeramente desigual, que rápidamente MCTS encontraría explotando el camino de nodos prometedores y dando una evaluación más precisa de la posición (al construir un árbol asimétrico que es capaz de explotar posiciones más prometedoras).



Tablero 13. Ejemplo del campeonato por el título mundial de 2018. Blancas:

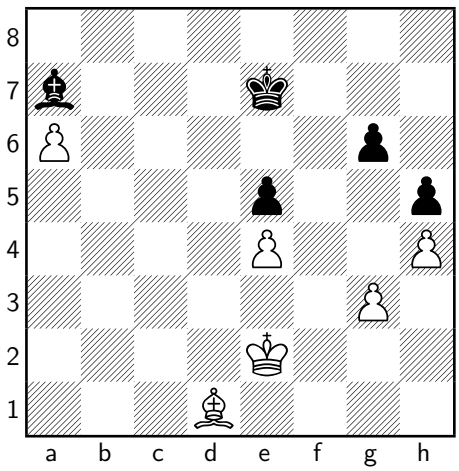
Caruana, F. Negras: Carlsen, M. Mueven negras. El superodenador Sesse llega a calcular un mate en 36 jugadas mientras que Stockfish hasta nivel 53 de profundidad evalúa la ventaja como $-2{,}18$. Analizar todos los niveles hace que no encuentre posiciones terminales en el tiempo disponible.

Dado que estas situaciones no son comunes en ajedrez, la mayoría de los mejores módulos de ajedrez están basados en minimax con α-β, libro de aperturas y libro de finales teóricos.

Métodos hibrídos y entrenamiento con aprendizaje automático reforzado han ayudado a que MCTS tenga relevancia en el campo de la programación de jugadores automáticos siendo MCTS Komodo y AlphaZero los mejores jugadores automáticos que incluyen este algoritmo.

5.3. Un ejemplo en el que ambos fallan.

El siguiente ejemplo es debido al análisis del Gran Maestro Daniel Alsina para la web Chess24 [2]. La partida está encmarcada dentro de la final del TCEC que comentábamos en el capítulo 1. Juega con las piezas blancas el jugador automático basado en aprendizaje automático LeelaChessZero (que no es exactamente MCTS pero posee similitudes) frente al jugador automático basado en minimax con poda α-β Stockfish.

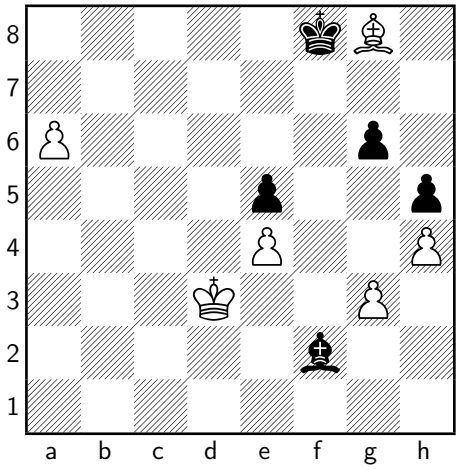


Tablero 14. Complejo final de peones y alfiles de distinto color. Hasta este momento ninguno de los jugadores automáticos ha mostrado ninguna evaluación extraña para un jugador humano experimentado.

LeelaChessZero evalúa la posición con una pequeña ventaja para las piezas blancas pero insuficiente para la victoria (pues evalúa que lo más probable es llegar a un final con un peón de más para las blancas que acabará en tablas), mientras que Stockfish da una ventaja decisiva para las blancas sin llegar a calcular una variante de jugadas concreta que lleve a esa victoria.

Es cierto que, en posiciones en las que existen fortalezas, los módulos como Stockfish suelen dar ventaja de en torno a 3 peones pese a no encontrar la variante ganadora, pero no es este el caso. En este ejemplo, Stockfish da una ventaja de más de 13 peones, ventaja que implica la victoria directa en la mayoría de los casos. En la posición actual del tablero, una evaluación tan alta implicaría que Stockfish es capaz de ver cómo el peón de la columna a llega a coronar (llegar a última fila y convertirse en una dama).

El GM Alsina, tras forzar a ambos módulos a calcular variantes seleccionadas por él, se da cuenta de que la única manera de intentar ganar ventaja es capturar los peones de las columnas g y h.



Tablero 15. Momento crítico para la victoria de las piezas blancas.

En esta posición, las negras acaban de jugar **Rf8** amenazando capturar el alfil blanco. Incluso en esta posición, ambos módulos siguen con evaluaciones similares a las del tablero anterior pero siguen sin llegar a calcular qué jugada es la ganadora. Alsina detecta que puede ser **Ah7** y tras **Rg7** el alfil se sacrifica por el peón **Axg6**, el rey negro lo captura y el rey blanco se dirige a apoyar la coronación del peón de la columna a. Obviamente, el alfil negro deberá sacrificarse por el peón para evitar dicha coronación, llegando a un final de peones y reyes.

Una vez se jugó **Ah7**, ambos módulos fueron capaces de ver toda la línea hasta llegar a un final ganado para las blancas. No obstante, resulta interesante observar cómo los módulos siguen fallando en cierto tipo de posiciones en las que, por un lado, el efecto horizonte sigue siendo un gran problema para los módulos basados en minimax y, por otro lado, los algoritmos basados en MCTS evalúan como tablas posiciones que se pueden ganar en los casos en los que sólo una variante es ganadora.

Capítulo 6

Construcción de un jugador automático.

> *Play the opening like a book, the middle game like a magician, and the endgame like a machine.*
>
> — Rudolf Spielmann.

Finalmente, este capítulo trata la implementación de los algoritmos de búsqueda de jugadas en un lenguaje de programación concreto, en este caso, Python.

Desarrollaremos un estudio bibliográfico sobre el trabajo existente, nos centraremos en entender la implementación de los algoritmos estudiados, pasando por encima de detallar el funcionamiento de algunas funciones auxiliares menos relevantes. Veamos, en primer lugar, una posible implementación del algoritmo minimax en su versión más básica.

6.1. Implementación de Minimax.

El paquete **chess** de Python nos ayuda a realizar la implementación en la geometría del tablero de ajedrez, donde éste se representa como una lista de longitud 64. En cada posición de esta lista tendremos un símbolo que especifique si hay una pieza blanca: peón (P), caballo (N), alfil (B), torre (R), dama (Q) y rey (K), si hay una pieza negra (usando los mismos caracteres en minúscula), o

nada, lo cual denotaremos con un punto (.).

Damos en primer lugar el valor de las piezas para definir una versión básica de la función de evaluación.

Algoritmo 7 Definición del valor de las piezas renormalizando a 10 el valor del peón.

Entrada: Pieza.
Salida: Valor de la pieza.

```
1: def getPieceValue(piece):
2: if (piece == None) then
3:     return 0
4: value = 0
5: if piece == 'P' or piece == 'p' then
6:     value = 10
7: if piece == 'N' or piece == 'n' then
8:     value = 30
9: if piece == 'B' or piece == 'b' then
10:    value = 30
11: if piece == 'R' or piece == 'r' then
12:    value = 50
13: if piece == 'Q' or piece == 'q' then
14:    value = 90
15: if piece == 'K' or piece == 'k' then
16:    value = 900
17: return value
```

Notemos que asignamos el valor 0 a las casillas en blanco y un valor muy alto al rey para que la futura función de evaluación no considere sacrificarlo por otra pieza. Vemos cómo, definida la geometría del tablero y las piezas, podemos obtener una función de evaluación considerando la diferencia de material en el tablero tal y como se describió en el capítulo 5.

Algoritmo 8 Función de evaluación básica. La inclusión de más características a considerar resultaría en un jugador automático de mayor nivel.

Entrada: Posición.
Salida: Evaluación de la posición.

```
1: def evaluation(board):
2: i = 0
3: evaluation = 0
4: x = True
5: try:
6:     x = bool(board.piece_at(i).color)
7: except AttributeError as e:
8:     x = x
9: while i < 63 do
10:    i+ = 1
11:    if x then
12:        evaluation = evaluation + (getPieceValue(str(board.piece_at(i)))
13:    else
14:        evaluation = evaluation -getPieceValue(str(board.piece_at(i))))
15: return evaluation
```

Recordemos que en Python las listas comienzan con el índice 0. Este algoritmo lee la lista **board** y detecta si hay o no una pieza en cada posición de la lista (líneas 5-8). Cuando detecta una pieza, añade su valor a la función de evaluación, con el signo más para las blancas, y con el signo menos para las negras (líneas 11-5). Finalmente, devolvemos la evaluación final, que en este caso no es más que la diferencia de material en el tablero.

Veamos ahora cómo generar los movimientos que forman el árbol de juego. La generación de los sucesores construirá el árbol de juego nivel a nivel. Posteriormente, la función de evaluación los estudiará para aplicar el algoritmo minimax.

Algoritmo 9 Algoritmo minimax. Versión básica.

Entrada: Profundidad máxima a evaluar, posición del tablero, si juegan blancas (`TRUE`) o negras (`FALSE`).
Salida: Mejor movimiento.

```
1: def minimax(depth, board, is_maximizing):
2: if (depth == 0) then
3:     return -evaluation(board)
4: possibleMoves = board.legal_moves
5: if (is_maximizing) then
6:     bestMove = -9999
7:     for x in possibleMoves do
8:         move = chess.Move.from_uci(str(x))
9:         board.push(move)                          ▷ Realizamos el movimiento.
10:        bestMove = max(bestMove,minimax(depth−1, board, not is_maximizing))
11:        board.pop()                               ▷ Deshacemos el movimiento
12:    return bestMove
13: else
14:    bestMove = 9999
15:    for  x in possibleMoves do
16:        move = chess.Move.from_uci(str(x))
17:        board.push(move)
18:        bestMove = min(bestMove, minimax(depth−1, board, not is_maximizing))
19:        board.pop()
20:    return bestMove
```

Veamos cómo sería una ejecución del algoritmo minimax siguiendo esta implementación:

1.Recibimos como entrada una profundidad, una posición y qué jugador tiene que mover. Si queremos aplicar el algoritmo a profundidad cero simplemente llamamos a la función de evaluación y la aplicamos a la posición.

2. Si elegimos una profundidad positiva, generamos todos los movimientos legales (función que proporciona el paquete **chess**). Si le toca jugar a las blancas estudiamos todos los posibles sucesores y, secuencialmente, estudiamos el máximo entre el mejor movimiento calculado hasta el momento y la ejecución del algoritmo minimax en el siguiente nivel.[1]

[1]Si estábamos aplicando minimax a una profundidad de 10 niveles, al aplicar minimax tenemos que hallar un máximo o un mínimo entre el mejor movimiento calculado hasta el

3. Si le toca jugar a las negras el razonamiento es análogo.

Como vemos, la implementación es sencilla (máxime si poseemos un paquete con funciones predefinidas que nos resuelven detalles relativos a la generación de movimientos legales). El siguiente paso natural es, como sabemos, la mejora con la poda α-β.

Algoritmo 10 Algoritmo minimax con poda α-β.

Entrada: Profundidad máxima a evaluar, posición del tablero, cotas α y β y si juegan blancas (`TRUE`) o negras (`FALSE`).
Salida: Mejor movimiento.

```
1: def minimax(depth, board, alpha, beta, is_maximizing):
2: if (depth == 0): then
3:     return -evaluation(board)
4:     possibleMoves = board.legal_moves
5: if (is_maximizing): then
6:     bestMove = −9999
7:     for x in possibleMoves: do
8:         move = chess.Move.from_uci(str(x))
9:         board.push(move)
10:        bestMove =
11:        max(bestMove,minimax(depth−1,  board,alpha,beta, not is_maximizing))
12:        board.pop()
13:        alpha = max(alpha,bestMove)
14:        if beta <= alpha: then
15:            return bestMove
16:     return bestMove
17: else
18:     bestMove = 9999
19:     for x in possibleMoves: do
20:         move = chess.Move.from_uci(str(x))
21:         board.push(move)
22:         bestMove =
23:         min(bestMove, minimax(depth−1, board,alpha,beta, not is_maximizing))
24:         board.pop()
25:         beta = min(beta,bestMove)
26:         if (beta <= alpha): then
27:             return bestMove
28:     return bestMove
```

momento y la aplicación del algoritmo al siguiente nivel, es decir, aplicamos minimax con una profundidad de 9 niveles para las negras.

Esta implementación es muy similar a la del algoritmo minimax anterior, con la salvedad de la condición de poda que se da en las líneas 13-15 y 25-27.

Esta implementación, no obstante, genera un jugador automático de poco nivel, pues la función de evaluación estima muy pocas características. En el archivo adjunto **sunfish.py** encontramos el caso de un jugador en el que se tienen en cuenta varias características descritas en el capítulo 5, como por ejemplo la variabilidad en el valor de las piezas en función de la casilla en la que se encuentran. Realizando un seguimiento de este jugador automático en la web Lichess, encontramos que su nivel de juego (contra otros jugadores automáticos) es aproximadamente la de un jugador aficionado de alto nivel (1900 de elo[2]), es decir, no tiene un nivel profesional ($\geq$ 2300 de elo), pero es capaz de ganar a jugadores experimentados.

[2]Sistema de puntuación oficial para ajedrez. Un jugador principiante tiene un elo de 1200, un aficionado aproximadamente 1700 y jugadores aficionados experimentados 2000. Los jugadores profesionales consiguen títulos de maestría a partir de los 2300, 2400 y 2500 de elo junto a otros méritos. El campeón del mundo actual, Magnus Carlsen, tiene un elo de 2870. El campeón del último TCEC, LeelaChessZero tiene un elo aproximado de 3500.

6.2. Implementación de MCTS.

Por otro lado, veamos una posible implementación del algoritmo MCTS:

Algoritmo 11 Implementación de MCTS genérico. Se definen las fases básicas.

```
1: def __init__(self, exploration_weight=1):
2: self.Q = defaultdict(int)                          ▷ Recompensa de cada nodo.
3: self.N = defaultdict(int)                          ▷ Visitas a cada nodo.
4: self.children = dict()                             ▷ Sucesores cada nodo.
5: self.exploration_weight = exploration_weight

6: def choose(self, node):
7: if node.is_terminal(): then
8:     raise RuntimeError(f 'El nodo elegido {node} es terminal')
9: if  node not in self.children: then
10:    return node.find_random_child()

11: def score(n):
12: if  self.N[n] == 0: then
13:    return float('-inf')
14:    return self.Q[n] / self.N[n]
15: return max(self.children[node], key=score)

16: def do_rollout(self, node):
17: path = self._select(node)
18: leaf = path[-1]
19: self._expand(leaf)
20: reward = self._simulate(leaf) self._backpropagate(path, reward)

21: def _select(self, node):
22: path = []
23: while True: do
24:    path.append(node)
25:    if  node not in self.children or not self.children[node]: then
26:        return path
27:    unexplored = self.children[node] - self.children.keys()
28:    if unexplored: then
29:        n = unexplored.pop() path.append(n)
30:         return path
31:    node = self._uct_select(node)

32: def _expand(self, node):
33: if  node in self.children: then
34:    return {'Ya expandido'}
35: self.children[node] = node.find_children()
```

Algoritmo 12

```
36: def _simulate(self, node):
37: invert_reward = True
38: while True: do
39:     if  node.is_terminal(): then
40:         reward = node.reward()
41:         return 1 - reward if invert_reward else reward
42:     node = node.find_random_child()
43:     invert_reward = not invert_reward

44: def _backpropagate(self, path, reward):
45: for  node in reversed(path): do
46:     self.N[node] += 1
47:     self.Q[node] += reward
48:     reward = 1 - reward        ▷ 1 para el jugador que mueve, 0 para el otro.

49: def _uct_select(self, node):
50: assert all(n in self.children for n in self.children[node])
51: log_N_vertex = math.log(self.N[node])

52: def uct(n):
53:  return self.Q[n] / self.N[n] +
54:          self.exploration_weight * math.sqrt( log_N_vertex / self.N[n] )
55: return max(self.children[node], key=uct)
```

En primer lugar, creamos variables para guardar las estadísticas de los nodos: recompensas, visitas y sucesores.

A continuación, creamos una función capaz de detectar si el nodo seleccionado es o no terminal (**choose**). A su vez, **score** y **do_rollout** generan las primeras estadísticas para construir ramas del árbol y simular los nodos cercanos.

Las fases de selección (usando el criterio UCT) y expansión se definen a continuación junto a la simulación de nuevos nodos y la retropropagación de la información obtenida.

Como aplicación sencilla, véase (y compílese para jugar) el tic-tac-toe (o 3 en raya) en el archivo adjunto **tictactoe.py**. En dicho archivo se define el entorno y reglas de juego donde aplicar la implementación anterior.

Como trabajo futuro sería posible mejorar las implementaciones presentadas, especialmente en la función de evaluación del algoritmo minimax y en la implementación de MCTS en un entorno de ajedrez. Sería interesante realizar una comparativa de rendimientos de varios tipos de jugadores automáticos realizando pruebas con distintas funciones de evaluación, distintos valores de la profundidad máxima considerada, etc.

Capítulo 7

Conclusiones y trabajo futuro.

El trabajo realizado ha tratado los temas fijados en los objetivos del mismo. Se ha realizado una revisión de los fundamentos de inteligencia artificial, comenzando por el establecimiento de un marco general y presentando posteriormente algunos resultados interesantes para los algoritmos Minimax y MCTS.

El desarrollo teórico ha sido extenso y ha tratado de ser un compendio de distintas referencias clásicas, a la vez que se han formalizado conceptos que suelen figurar en la literatura de manera vaga, especialmente en lo referente a Monte Carlo, para el que, debido a su reciente aparición en comparación con Minimax, es mucho más difícil encontrar referencias adecuadas.

Este aspecto del trabajo ha sido central en el desarrollo del mismo pues, como trabajo en matemáticas que es, debe ser preciso en las definiciones y resultados obtenidos. Si bien algunas pruebas técnicas no se han incluido, en parte debido a su dificultad técnica, que superaba los objetivos de este trabajo, y en parte para no resultar en una lectura demasiado pesada, el lector interesado puede encontrarlas en las referencias aportadas en la memoria.

La aplicación práctica hacia el juego del Ajedrez proporciona asimismo un

doble objetivo de este trabajo. Por un lado, la modelización matemática de un problema, su planteamiento abstracto, hipótesis, desarrollo de resultados, y su posterior aplicación de vuelta al problema real original, es una de las características más destacadas de las matemáticas con respecto a otras disciplinas. Por otro lado, como aficionado al ajedrez y matemático, encuentro una gran motivación en el estudio de dos pasiones unidas en el campo de la computación y de los jugadores automáticos de ajedrez.

La implementación de los algoritmos estudiados en busca de la construcción completa de un jugador automático de ajedrez ha escapado a las limitaciones de tiempo y profundidad típicas de un Trabajo Fin de Grado, y finalmente sólo ha sido posible abordar una implementación básica de los mismos. No obstante, tengo la intención personal de continuar el trabajo en el futuro con una fijación más importante en este aspecto, así como en el siguiente paso natural al que ya hemos hecho alusión varias veces a lo largo de la memoria, este es, el uso de técnicas de aprendizaje automático para aumentar el nivel de nuestro jugador.

Concluiremos este trabajo dando una breve descripción de estas técnicas y cómo podrían ser aplicadas.

7.1. El papel del Aprendizaje Automático.

Uno de los objetivos del aprendizaje automático es construir un método que permita reconocer patrones para un conjunto de datos y dar, a partir de él, predicciones para datos nuevos. Normalmente, este objetivo lo aborda por medio de la resolución de dos problemas principales: el problema de regresión, para la predicción de valores numéricos, y el problema de clasificación, para hacer lo propio clasificando por categorías o características.

Hay muchas aproximaciones posibles para realizar estas tareas. A cada una de esas posibles aproximaciones (que combinan una familia de algoritmos con una familia de funciones matemáticas representadas de forma computacional) se le suele asociar el concepto de Modelo de Aprendizaje.

Uno de los modelos de aprendizaje que más éxitos está cosechando en los últimos años y es responsable del nuevo vigor que está tomando la Inteligencia Artificial en todos los campos, aunque su aparición es muy anterior, es el de redes neuronales. En este modelo, el elemento básico usado es el de neurona artificial (una modelización simplificada de lo que sería una neurona real). Una de las cualidades más interesantes de estas neuronas artificiales es la capacidad para combinar de forma sencilla gran cantidad de neuronas para que compartan información, y así obtener modelos de alta complejidad que pueden realizar tareas dependientes de muchos factores y que reconocen patrones complejos en los datos.

Al igual que en la mayoría de los modelos basados en Aprendizaje Automático, para que las neuronas categoricen o calculen de manera correcta datos tenemos que realizar un *entrenamiento* o estimación de parámetros del modelo [8]. De este modo, la inferencia realizada con datos nuevos es más fiable cuanto más datos en la muestra de entrenamiento poseamos.

En 2015 el grupo de Inteligencia Artificial de Google, DeepMind, consiguió un hito histórico al construir un jugador automático de Go, AlphaGo, que, basado en el uso conjunto de MCTS, aprendizaje con refuerzo y redes neuronales profundas (lo que se denomina Deep Learning) se convirtió en la primera máquina de Go en ganar a un jugador profesional de Go, Lee Sedol, campeón del mundo.

Poco después, en 2017, los mismos investigadores dan un paso más a este proyecto y desarrollan un jugador automático de ajedrez [25], Alpha Zero (realmente, es un jugador simultáneo de Go, Ajedrez y Shogi). La particularidad de este jugador era su entrenamiento, que fue realizado sin datos aportados por humanos, es decir, sin más información que las reglas del juego (por eso el apellido Zero). El entrenamiento de la red neuronal fue realizado por el propio algoritmo al entrenarse jugando contra sí mismo y derrotando con un impresionante resultado de 100-0 a su versión anterior AlphaGo.

La idea principal que hay detrás de la aproximación de DeepMind (y que ya se había intentado con anterioridad, pero sin los recursos técnicos que aporta una empresa como Google) es la de sustituir las grandes tablas de asociación que podemos extraer de MCTS por medio de una red neuronal profunda que extrapola los conocimientos adquiridos por la ejecución de MCTS para obtener una política más robusta y general.

La mejora con respecto a la fase de selección en MCTS es consecuencia de considerar más rápidamente las mejores opciones futuras de expansión en lugar de estudiar todas de manera equiprobable.

Una vez realizado MCTS para un estado, éste se convierte en un dato para entrenar la red neuronal y se vuelve a empezar. Al actualizar toda la red podemos enfrentar la nueva y la antigua red a una serie de partidas fijadas. Si surge como vencedora en un porcentaje superior a la anterior ($\sim 55\,\%$), la red antigua se desecha y volvemos a repetir el proceso completo.

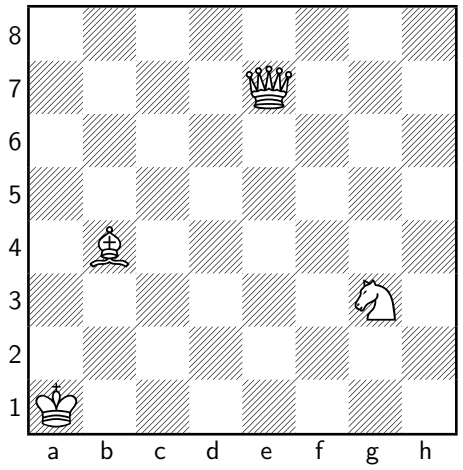
Por ejemplo, DeepMind estimó que siguiendo este proceso, tras 4 horas de entrenamiento desde cero (en una máquina simple con 4 TPUs, un procesador específico para entrenar redes neuronales) jugaba con un elo superior a Stockfish 8. Sin embargo, y como comparativa entre las diversas aproximaciones, AlphaZero solo busca unas 80,000 posiciones por segundo en ajedrez, frente a los 70 millones de posiciones que analiza Stockfish, lo que indica que las posiciones evaluadas por AlphaZero son mucho más prometedoras y es capaz de discernir partes del árbol de juego que le van a proporcionar más rendimiento.

La fundamentación teórica de este proceso es interesante y continúa de manera natural el trabajo realizado. Se propone como trabajo futuro el desarrollo teórico necesario para fundamentar la teoría de aprendizaje automático aplicado al juego de ajedrez así como su implementación en un lenguaje de programación adecuado, analizando implementaciones libres que ya se pueden encontrar disponibles.

Anexo A. Notación algebraica en ajedrez.

La notación algebraica es la notación más usada para representar movimientos y secuencias de movimientos en ajedrez así como para la anotación de partidas de manera oficial en los torneos de ajedrez.

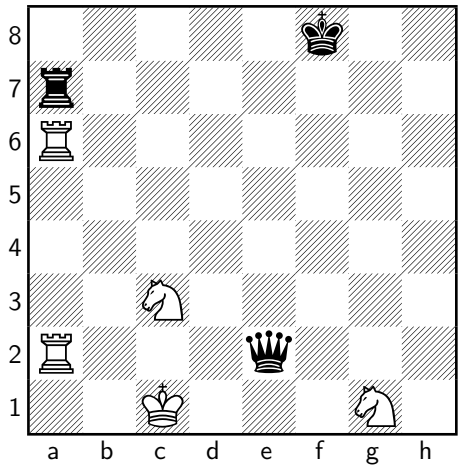
Combina las ya conocidas letras para representar las piezas: peón (), caballo (C), alfil (A), torre (T), dama (D) y rey (R) con las coordenadas del tablero en columnas (a-h) y filas (1-8).



Tablero 16. Ejemplo de notación. Para describir la posición de las piezas escribiríamos: **Ra1**, **Ab4**, **Cg3**, **De7**.

Pero esto no es suficiente para describir todos los casos. Cuando una pieza mueve y captura a otra además de indicar qué pieza se mueve y a dónde añadi-

mos un 'x' antes de las coordenadas para indicar la captura. También, si dos piezas iguales pueden acceder a la misma casilla en un movimiento añadimos la letra de la columna en la que se encuentre la pieza que mueve (si ambas están en distintas columnas) o el número de la fila de la pieza que mueve (si ambas están en la misma columna). Véase el siguiente ejemplo.



Tablero 17. Si el caballo de **g1** quisiera capturar la dama de e2 el movimiento sería denotado como **Cgxe2**. También si la torre de **a2** quisiera acceder a la misma casilla el movimiento sería **Txe2** pues la otra torre no puede acceder a e2. Así la torre de **a6** capturara la torre de a7 sería **Txa7** pero si quisiera mover a a4 sería **T6a4**.

Bibliografía

[1] B Abramson. The expected-outcome model of two-player games, 1987.

[2] D. Alsina. Entendiendo a leela zero. *chess24.es*.

[3] T. M. Apostol. *Análisis matemático.* Reverté, 2006.

[4] O. Arenz. Monte carlo chess. Master's thesis, Universidad de Darmstadt, 2012.

[5] P. Auer et al. Finite-time analysis of the multiarmed bandit problem. *Machine Learning*, 2002.

[6] G. M. Baudet. On the branching factor of the alpha-beta pruning algorithm. *Artificial Intelligence 10(2):173-99*, 1978.

[7] D. Borrajo et al. *Inteligencia artificial: métodos y técnicas.* Editorial Centro de Estudios Ramón Areces, S.A, 1997.

[8] W. Feller. *Introducción a la teória de probabilidades y sus aplicaciones.* Limusa-Wiley, 1973.

[9] S. H. Fuller et al. An analysis of the alpha-neta pruning algorithm. *Dept. of Computer Science Report. Carnegie-Mellon University*, 1973.

[10] M. S. Goetsch, G. Campbell. Experiments with the null-move heuristic. *Computers, Chess, and Cognition, Springer-Verlag, pp. 159–168*, 1990.

[11] O. Hagen A. Hjellen, B. Rolfsrud. Her er carlsens ukjente vm-våpen.

[12] J. I. Illana. Métodos monte carlo. *Universidad de Granada.*, 2013.

[13] Y. Khomskii. Infinite games. *Universidad de Sofía*, 2010.

[14] R. W. Knuth, D. E. Moore. An analysis of alpha-beta pruning. *Artificial Intelligence 6(4):293-326*, 1975.

[15] L. Kocsis and C. Szepesvári. Bandit based monte-carlo planning. In *Proceedings of the Seventeenth European Conference on Machine Learning (ECML 2006)*, Lecture Notes in Computer Science, pages 282–293, Berlin/Heidelberg, Germany, 2006. Springer.

[16] D. N. L. Levy. *Computer Games I.* Springer-Verlag, 1987.

[17] D. Martínez. Entrenamiento con grandes maestros. *chess24.es.*

[18] G. D. Modia-Pozuelo. Jugadores automáticos basados en árboles de búsqueda monte carlo. *Universidad Compluetense de Madrid*, 2018.

[19] B. Nasarre-Embid. Método de monte-carlo tree search (mcts) para resolver problemas de alta complejidad: Jugador virtual para el juego del go. Master's thesis, Universidad de Zaragoza, 2012.

[20] M. E. J. Newman. *Networks: An Introduction.* Oxford University Press, 2010.

[21] J. Pearl. *Heuristics. Intelligent Search Strategies for Computer Problem Solving.* Addison Wesley, 1984.

[22] E Powley et al. A survey of monte carlo tree search methods. *IEEE Transactions on computational intelligence and AI in games, Vol 4, No. 1*, Marzo 2012.

[23] P. Russel, S. Norvig. *Artificial Intelligence: A Modern Approach.* Prentice Hall, 2003.

[24] C. E. Shannon. Programming a computer for playing chess. *Philosophical Magazine, Ser. 4. Vol 41, No. 314*, 1950.

[25] T. Schrittwieser J. Antonoglou I. Lai M. Guez A. Lanctot M. Sifre L. Kumaran D. Graepel T. Lillicrap T. Simonyan K. Hassabis D. Silver, D. Hubert. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *arXiv:1712.01815v1 [cs.AI]*, 2017.

[26] J. K. Slagle, J. R. Dixon. Experiments with some programs that search game trees. *JACM 16(2):189-207.*, 1969.

[27] Y. Wang et al. Modification of uct with patterns in monte-carlo go. *inria-00117266v1*, 2006.

[28] Chessprogramming.org Wiki. *Benjamin Franklin.*

[29] Chessprogramming.org Wiki. *Caissa.*

[30] Chessprogramming.org Wiki. *José Raúl Capablanca.*

[31] Chessprogramming.org Wiki. *Napoleon Bonaparte vs The Turk (Automaton).*

[32] Chessprogramming.org Wiki. *Top Chess Engine Championship.*