

GRADO EN INGENIERÍA EN TECNOLOGÍA INDUSTRIAL

TRABAJO FIN DE GRADO

SISTEMA INTELIGENTE DE JUEGO DE AJEDREZ

Alumno: Ochoa de Eribe, Martínez, Daniel

Director (1): Irigoyen, Gordo, Eloy

Director (2): Larrea, Sukia, Mikel

Curso: 2019-2020

Fecha: Bilbao, 10, 02, 2020

Resumen

Este proyecto tratará de diseñar un sistema inteligente que sea capaz de jugar al ajedrez por sí solo. Para ello, se examinarán algoritmos relacionados con la inteligencia artificial que permitan a una máquina tomar decisiones prácticas en el juego del ajedrez, pasando posteriormente a implementar algunos de estos algoritmos en Python. Se pretende que el motor de ajedrez sea capaz de disputar una partida de ajedrez contra un niño, fomentando así la práctica del juego entre los jóvenes.

Palabras clave: sistema inteligente, ajedrez, Python

Abstract

The aim of this project is to design an intelligent system which will be capable of playing chess by itself. In order to score that, some algorithms related with Artificial Intelligence, which enable a machine taking practical decisions, will be examined. Afterwards, some of these algorithms will be implemented in Python. It is intended that the chess engine can fight for a win against a child, making chess popular among young people.

Key words: intelligent system, chess, Python

Laburpena

Proiektu honen helburua, xakean bakarrik aritzeko gai den sistema adimentsu bat sortzea da. Hau lortzeko, adimen artifizialarekin erlazionatuta dauden algoritmoak aztertuko dira. Ondoren, algoritmo honetako batzuk Pythonen inplementatuko dira. Xakeko motorra ume baten aurka lehian aritzeko gai izatea espero da, xakeak daukan populartasuna gazteen artean hedatzeko.

Gako-hitzak: sistema adimentsua, xakea, Python

Índice

1	Introducción	9
2	Contexto	10
2.1	Entorno tecnológico.....	10
2.1.1	Desarrollo del ajedrez computacional	10
2.1.2	Puesta en práctica de AlphaZero	12
2.2	Entorno de aplicación	18
2.3	Objetivo del TFG y definición del problema a resolver.....	19
3	Alcance del proyecto.....	20
4	Beneficios aportados por el proyecto.....	21
4.1	Beneficios sociales	21
4.1.1	Desarrollo intelectual	21
4.1.2	Formación en carácter.....	21
4.1.3	Formación en valores	22
5	Primer enfoque: motor de ajedrez convencional.....	23
5.1	Módulos de un motor de ajedrez	23
5.1.1	Generador de movimientos	23
5.1.2	Función de evaluación.....	23
5.1.3	Función de búsqueda.....	25
5.2	Aspectos adicionales a tener en cuenta	29
5.2.1	Ordenación	29
5.2.2	Jugadas-asesinas (killer-moves).....	29
5.2.3	Efecto horizonte.....	30
5.2.4	Búsqueda de quiescencia (quiescence search).....	31
5.2.5	Movimiento nulo (null-move pruning)	32
5.2.6	Bases de datos de partidas	33
5.2.7	Tablas de Nalimov	34
6	Segundo enfoque: AlphaZero.....	36
6.1	Árbol de búsqueda Monte Carlo (Monte Carlo tree search)	36
6.1.1	Selección	36
6.1.2	Expansión	37
6.1.3	Simulación	37
6.1.4	Retropropagación	37
6.2	Arquitectura de AlphaZero.....	38
6.3	MCTS y poda alfa-beta	39
7	Descripción de la solución propuesta	41
7.1	Generador de movimientos	41
7.2	Función de evaluación	41
7.3	Función de búsqueda.....	42
8	Descripción de tareas. Planificación. Diagrama de Gantt.....	43

9	Medios y técnicas básicas	46
9.1	Ordenador	46
9.2	GeForce GTX 1050.....	46
9.3	Udemy	46
9.4	Python 3.7	46
9.5	Python-chess	46
9.6	Anaconda Navigator	48
9.7	Github.....	49
9.8	Gitkraken.....	49
10	Presupuesto	50
11	Riesgos del proyecto.....	52
11.1	Clasificación de los riesgos.....	53
11.1.1	Alta tasa de defectos	53
11.1.2	Distracciones	53
11.1.3	Aprendizaje excesivamente lento y costoso	53
11.1.4	Pocos incentivos para seguir prácticas comunes	54
11.1.5	Pérdida de información.....	54
11.2	Medidas de control	55
11.2.1	Alta tasa de defectos	55
11.2.2	Distracciones	55
11.2.3	Aprendizaje excesivamente lento y costoso	55
11.2.4	Pocos incentivos para seguir prácticas comunes	55
11.2.5	Pérdida de información.....	55
12	Conclusiones	56
13	Fuentes de información	58
14	Anexo I: Código	61
15	Anexo II: Resultados experimentales	68
15.1	Enfrentamiento 1	68
15.2	Enfrentamiento 2	68
15.3	Enfrentamiento 3	69
15.4	Enfrentamiento 4	70
15.5	Enfrentamiento 5	71
15.6	Enfrentamiento 6	72
15.7	Enfrentamiento 7	73

Índice de ilustraciones

Ilustración 1. Gary Kasparov vs Deep Blue.....	10
Ilustración 2. AlphaZero vs Stockfish	12
Ilustración 3. Nivel alcanzado por AlphaZero durante el entrenamiento.....	13
Ilustración 4. Resultados del enfrentamiento en el que jugaba la última versión de Stockfish y del enfrentamiento en el que Stockfish jugaba con libro de aperturas.....	14
Ilustración 5. Resultados del enfrentamiento oficial	14
Ilustración 6. Resultados de los enfrentamientos en los que Stockfish tenía ventaja de tiempo.....	15
Ilustración 7. Cantidad de jugadas analizadas por decisión	16
Ilustración 8. Adaptabilidad de AlphaZero con el tiempo disponible para pensar, medido en una escala Elo relativa al nivel de AlphaZero y Elmo con 40 ms por jugada.	16
Ilustración 9. Análisis de las aperturas humanas más populares que fueron jugadas por AlphaZero durante el entrenamiento.....	17
Ilustración 10. Ajedrez en las escuelas	18
Ilustración 11. Esquema del árbol de variantes minimax.....	26
Ilustración 12. Esquema del árbol de variantes minimax. Primer paso.....	26
Ilustración 13. Esquema del árbol de variantes minimax. Segundo paso	27
Ilustración 14. Esquema del árbol de variantes minimax. Tercer paso	27
Ilustración 15. Esquema del árbol de variantes minimax. Cuarto paso	27
Ilustración 16. Esquema del árbol de variantes de la poda alfa-beta.....	28
Ilustración 17. Orden de búsqueda de movimientos	29
Ilustración 18. Posición que ilustra el concepto de jugada asesina	30
Ilustración 19. Posición que ilustra el concepto de efecto horizonte.....	31
Ilustración 20. Pseudocódigo búsqueda de quiescencia	32
Ilustración 21. Posición de zugzwang	33
Ilustración 22. Base de datos de partidas	34
Ilustración 23. Base de datos de finales.....	35
Ilustración 24. Esquema del algoritmo MCTS	38

Ilustración 25. Ejemplo de iteración del MCTS.....	38
Ilustración 26. Diagrama de Gantt.....	43
Ilustración 27. Implementación del mate del pastor en Python-chess	47
Ilustración 28. Posición resultante tras el mate de pastor	48
Ilustración 29. Implementación código en Jupyter Notebooks.....	48
Ilustración 30. Esquema resultante en Gitkraken tras el trabajo.....	49
Ilustración 31. Presupuesto del proyecto	50
Ilustración 32. Planilla del primer enfrentamiento.....	68
Ilustración 33. Posición resultante tras el primer enfrentamiento	68
Ilustración 34. Planilla del segundo enfrentamiento	69
Ilustración 35. Posición resultante tras el segundo enfrentamiento.....	69
Ilustración 36. Planilla del tercer enfrentamiento.....	70
Ilustración 37. Posición resultante tras el tercer enfrentamiento	70
Ilustración 38. Planilla del cuarto enfrentamiento.....	70
Ilustración 39. Posición resultante tras el cuarto enfrentamiento.....	71
Ilustración 40. Planilla del quinto enfrentamiento	71
Ilustración 41. Posición resultante tras el quinto enfrentamiento.....	72
Ilustración 42. Planilla de sexto enfrentamiento	73
Ilustración 43. Posición resultante tras el sexto enfrentamiento	73
Ilustración 44. Planilla del séptimo enfrentamiento	74
Ilustración 45. Posición resultante tras el séptimo enfrentamiento	74

Índice de tablas

Tabla 1. Estadísticas del entrenamiento de AlphaZero	13
Tabla 2. Resultados del primer enfrentamiento.....	13
Tabla 3. Matriz de riesgos.....	52
Tabla 4. Enumeración riesgos del proyecto	54

Índice de ecuaciones

Ecuación 1. Función de utilidad de AlphaZero	36
Ecuación 2. Función de pérdida de AlphaZero	39

1 Introducción

Este documento contiene la explicación del Trabajo de Fin de Grado denominado Sistema Inteligente de Juego de Ajedrez. En una primera parte se presenta el contexto del trabajo, la oportunidad del mismo y los beneficios que se esperan conseguir.

Posteriormente, se analizan los dos enfoques principales que se podrían seguir para la realización del proyecto (uno de ellos tradicional y el otro puntero y novedoso), y se describe la solución adoptada.

A continuación, se planifican todos los trabajos a realizar, viendo las diferentes relaciones entre las tareas a llevar a cabo.

Este trabajo de Fin de Grado cuenta también con un presupuesto a grandes rasgos y con un análisis de riesgos. En este último se enumeran los riesgos principales, dando una explicación de las consecuencias de los mismos y posibles acciones a realizar para evitarlos o conseguir que no supongan una gran pérdida en el proyecto.

Por último, se presentan las conclusiones a las que se han llegado tras la realización del trabajo, así como la bibliografía utilizada durante el mismo.

Existen dos apartados de anexos adicionales:

En uno de ellos se puede visualizar el código desarrollado, con algunas explicaciones para facilitar la comprensión del mismo o algunos comentarios oportunos que explican cómo se solucionaron los principales problemas cuando estos se presentaron.

El otro apartado de anexos contiene resultados experimentales, es decir, partidas de ajedrez en las que se puede apreciar cómo iba mejorando el motor de ajedrez a medida que se le iban añadiendo mejoras.

2 Contexto

2.1 Entorno tecnológico

2.1.1 Desarrollo del ajedrez computacional

Jugar bien al ajedrez es una actividad que requiere de tanta creatividad y de un razonamiento tan sofisticado que una vez se pensó que se trataba de un objetivo inalcanzable para cualquier ordenador. Era frecuentemente reconocido junto con la poesía y la pintura como ejemplo de actividades que solo podrían ser llevadas a cabo por humanos. No obstante, la historia ha demostrado que hemos sido capaces de programar máquinas que dominan con maestría el juego del ajedrez.

En 1997, Deep Blue, un motor de ajedrez creado por IBM, derrotó al entonces campeón del mundo Garry Kasparov. Este acontecimiento hizo historia, al ser la primera vez que una inteligencia artificial conseguía derrotar al mejor jugador del mundo en el ajedrez. En las dos décadas posteriores, tanto los desarrollos de hardware como las investigaciones en el campo de la inteligencia artificial provocaron tal mejoría en el juego de las máquinas, que a día de hoy ningún jugador humano de la élite tiene posibilidades reales de ganar una partida contra un programa de ajedrez moderno.

No obstante, cabe destacar que la forma en la que los ordenadores juegan al ajedrez es muy diferente a la forma en la que lo hacen los humanos. Es cierto, que ambos calculan jugadas y miran más allá para tratar de predecir cómo se va a desarrollar la partida, pero los humanos son mucho más selectivos con las variantes a examinar. Los ordenadores, por otro lado, tienden a usar la fuerza bruta para calcular tantas variantes como sea posible, incluso aquellas que serían descartadas inmediatamente por un entendido del juego. Tomando como ejemplo el match Kasparov vs Deep Blue, Kasparov no era capaz de calcular más de 3-5 posiciones por segundo, mientras que Deep Blue analizaba en torno a 200 millones de posiciones por segundo, y todo ello, para jugar a un nivel muy similar (de las 6 partidas que formaban el match, Deep Blue ganó 2, empató 3 y perdió 1). Esto demuestra que los humanos son mucho más eficientes computacionalmente.



Ilustración 1. Gary Kasparov vs Deep Blue

Ha habido muchos intentos de hacer a los motores de ajedrez más selectivos en su cálculo, sin que estos pasen por alto continuaciones importantes. No obstante, éste ha resultado ser un problema muy complicado, ya que es difícil encontrar reglas fiables que nos indiquen qué ramas se deberían explorar en profundidad y qué ramas deberían ser descartadas instantáneamente. Los grandes maestros de ajedrez hacen esto generalmente en base a su intuición, a su “olfato”. Esta idea es muy abstracta, pero se podría tratar de explicar diciendo que el cerebro humano dispone de unos patrones obtenidos en base a la experiencia que les permite asociar jugadas y planes. La intuición se desarrolla jugando y observando miles de partidas, y nadie hasta el momento ha sido capaz de traducir este conocimiento en órdenes precisas y concretas para los ordenadores que funcionen en todos los casos. Dicho de otra forma, no se ha conseguido programar la intuición, que es, en esencia, la gran ventaja de los humanos sobre las máquinas.

No obstante, cabe mencionar la labor realizada por Marco Costalba, Joona Kiiski, Gary Linscott y Tord Romstad al crear a Stockfish, el motor de ajedrez convencional más fuerte hasta el momento, y que se estima tiene una fuerza de juego de entorno a los 3600 puntos de elo. Para aquellos que no conozcan el sistema de puntuación de la Federación Internacional de Ajedrez (más conocida como FIDE, del acrónimo de su nombre en francés), mencionar que Magnus Carlsen, el mejor jugador del mundo y posiblemente de la historia, posee actualmente 2875 puntos de elo, siendo su máximo histórico de 2882.

Por otro lado, en los últimos años se ha empezado a abordar la programación de los motores de ajedrez desde un punto de vista radicalmente diferente al convencional, mucho más próximo a la forma en la que los humanos razonan en ajedrez y, por lo tanto, mucho más eficiente computacionalmente. En lugar de intentar traducir las complicadas reglas mencionadas previamente en código, este enfoque trata de usar el aprendizaje automático (*machine learning*) para que las máquinas aprendan a jugar al ajedrez por sí solas y, en base a la experiencia obtenida de jugar partidas contra ellas mismas, saquen sus propias “reglas” para tener en cuenta en las partidas.

Este nuevo enfoque ha resultado ser todo un éxito, hasta el punto de que AlphaZero, uno de los desarrollos más recientes de DeepMind (empresa de Google dedicada a la inteligencia artificial), ha revolucionado completamente el mundo del ajedrez computacional al vencer de manera contundente a Stockfish en un match constituido por 100 partidas, de las cuales AlphaZero ganó 25 con las piezas blancas, ganó 3 con las piezas negras, y empató las restantes 72.



Ilustración 2. AlphaZero vs Stockfish

Lo curioso de todo esto es que 24 horas antes del match lo único que sabía AlphaZero sobre el ajedrez era cómo se movían las piezas. Fue gracias a un autoaprendizaje a base de jugar partidas contra sí mismo durante este breve periodo de tiempo que consiguió dominar el juego hasta el punto de ser invencible para un motor tan poderoso como Stockfish.

Y más aún, DeepMind ha conseguido resultados similares en otros juegos de mesa como son el Go o el Soghi. Esto se debe a que el hecho de dominar el aprendizaje desde cero, les ha permitido extrapolar sus desarrollos desde el juego de Go originalmente, hasta cualquier otro dominio.

2.1.2 Puesta en práctica de AlphaZero

El algoritmo de AlphaZero se aplicó en el ajedrez, en el shogi y en Go. Salvo por pequeñas modificaciones relacionadas con las reglas de cada juego, se ajustó dicho algoritmo de la misma manera y se usó la misma red neuronal.

2.1.2.1 Entrenamiento de la red neuronal

Se entrenaron diferentes instancias de AlphaZero para cada juego. El entrenamiento consistió en 700 000 pasos (mini-grupos de tamaño 4096) que empezaron desde parámetros inicializados de forma aleatoria. Se usaron 5000 TPUs de primera generación para generar los juegos contra sí mismo y 64 TPUs de segunda generación para entrenar las redes neuronales.

La siguiente figura muestra el nivel de AlphaZero durante el aprendizaje reforzado, en función del número de pasos de entrenamiento, en una escala de puntos de Elo. En ajedrez, AlphaZero superó a Stockfish después de solo 4 horas (300 000 pasos); en shogi, AlphaZero superó a Elmo tras menos de 2 horas (110 000 pasos); y en Go, AlphaZero superó a AlphaGo Lee después de 8 horas (165 000 pasos).

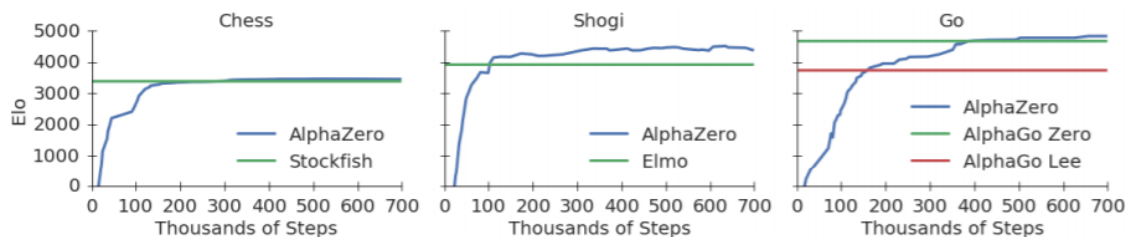


Ilustración 3. Nivel alcanzado por AlphaZero durante el entrenamiento

Y la siguiente tabla ilustra algunos datos estadísticos del entrenamiento de AlphaZero:

	Chess	Shogi	Go
Mini-batches	700k	700k	700k
Training Time	9h	12h	34h
Training Games	44 million	24 million	21 million
Thinking Time	800 sims 40 ms	800 sims 80 ms	800 sims 200 ms

Tabla 1. Estadísticas del entrenamiento de AlphaZero

2.1.2.2 Evaluación de AlphaZero

Tras las sesiones de autoaprendizaje, se evaluaron las instancias entrenadas de AlphaZero contra Stockfish, Elmo y la versión previa de AlphaGo Zero (entrenada por 3 días) en ajedrez, shogi y Go respectivamente, jugando un enfrentamiento a 100 partidas bajo un control de tiempo oficial de un minuto por jugada. AlphaZero y el previo AlphaGo Zero usaron una única máquina con 4 TPUs. Stockfish y Elmo jugaron a su máximo nivel según el informe publicado por DeepMind, usando 64 hilos y un tamaño de hash de 1GB. AlphaZero ganó de forma convincente a todos sus oponentes, perdiendo 0 partidas contra Stockfish y 8 partidas contra Elmo, y venciendo también a la versión previa de AlphaGo Zero.

Game	White	Black	Win	Draw	Loss
Chess	<i>AlphaZero</i>	<i>Stockfish</i>	25	25	0
	<i>Stockfish</i>	<i>AlphaZero</i>	3	47	0
Shogi	<i>AlphaZero</i>	<i>Elmo</i>	43	2	5
	<i>Elmo</i>	<i>AlphaZero</i>	47	0	3
Go	<i>AlphaZero</i>	<i>AG0 3-day</i>	31	–	19
	<i>AG0 3-day</i>	<i>AlphaZero</i>	29	–	21

Tabla 2. Resultados del primer enfrentamiento

Los resultados fueron tan contundentes que fue inevitable que ciertas personas salieran en defensa de los programas convencionales calificando el enfrentamiento de injusto. En el caso del ajedrez se argumentó que Stockfish había partido de una posición desfavorable con las siguientes razones, entre otras:

- En el match no se había utilizado la última versión de Stockfish
- Stockfish había jugado sin libro de apertura, lo que acotaba considerablemente su fuerza de cálculo.
- Stockfish no está diseñado para jugar con el control de tiempo de 1 minuto por jugada que se había utilizado en el match – Se ha invertido mucho esfuerzo en hacer que Stockfish sepa diferenciar una posición crítica y administre su tiempo de tal forma que pueda pensar más en este tipo de posiciones.
- El hardware que había sido utilizado por el equipo de DeepMind estaba expresamente diseñado para AlphaZero, mientras que Stockfish se había ejecutado en un ordenador “normal”.

Es por esto, que un año más tarde se decidió repetir el enfrentamiento en “igualdad de condiciones”. En esta ocasión se realizó una serie de matches en los que se procuró corregir cada una de las desventajas. Se jugó un match en el que AlphaZero se enfrentó a la última versión de Stockfish y otro en el que se le facilitó a Stockfish un libro de aperturas. Añadir el libro de aperturas sí pareció ayudar a Stockfish, que finalmente ganó un número considerable de partidas con las blancas, pero no las suficientes como para ganar el match. Estos fueron los resultados:

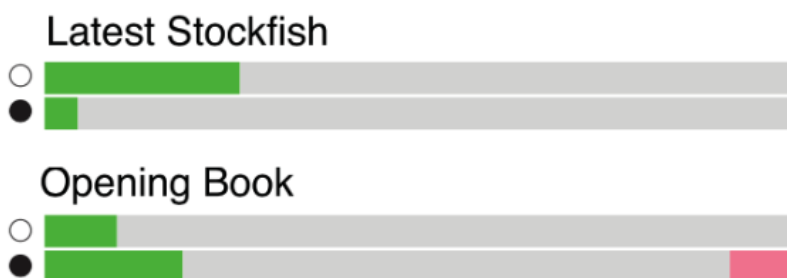


Ilustración 4. Resultados del enfrentamiento en el que jugaba la última versión de Stockfish y del enfrentamiento en el que Stockfish jugaba con libro de aperturas

También se jugó un enfrentamiento “oficial” a 1000 partidas en que tanto AlphaZero como Stockfish tuvieron 3 horas cada uno más un incremento de 15 segundos por jugada. Este control de tiempo dejó obsoleto uno de los mayores argumentos contra el impacto del match original: que el control de tiempo de 1 minuto por jugada jugaba en contra de Stockfish. Con las tres horas más un incremento de 15 segundos, no se puede mantener tal argumento, pues es una cantidad de juego enorme para cualquier módulo. El resultado de este match fue una aplastante victoria de AlphaZero, quien ganó 155 partidas, empató 6 y entabló las restantes 839.

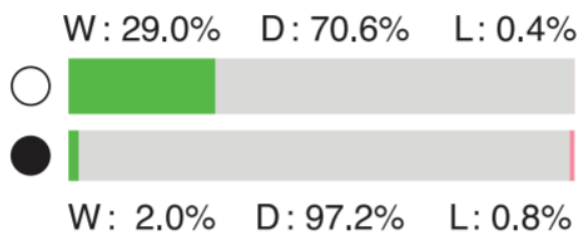


Ilustración 5. Resultados del enfrentamiento oficial

Por último, AlphaZero también venció a Stockfish en una serie de enfrentamientos con ventaja de tiempo para el último, superando al módulo tradicional incluso dándole ventaja temporal de 10 a 1. Stockfish solo empezó a superar a AlphaZero cuando su ventaja llegó a 30 a 1.

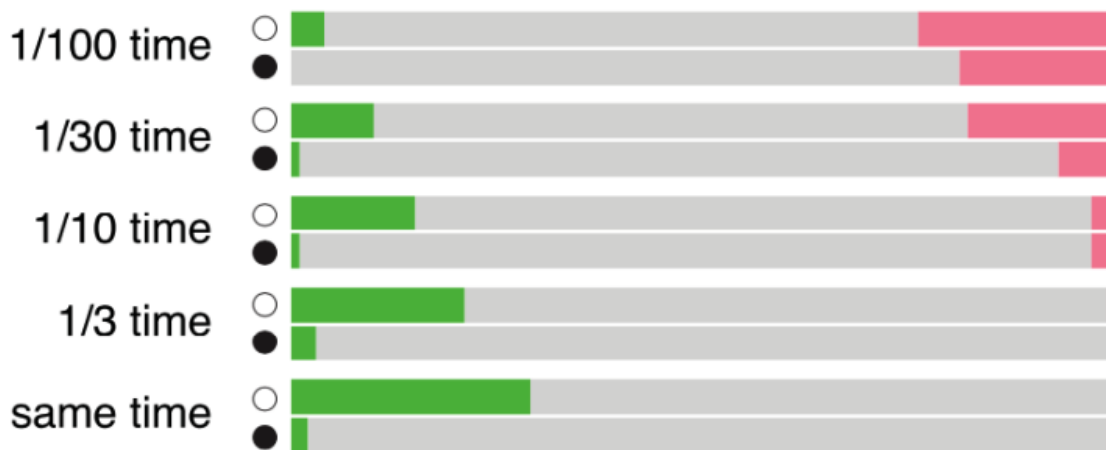


Ilustración 6. Resultados de los enfrentamientos en los que Stockfish tenía ventaja de tiempo

Los resultados de AlphaZero en los enfrentamientos con ventaja de tiempo sugieren una vez más, que no solo es mucho más fuerte que ningún otro módulo tradicional, sino que también usa una búsqueda por movimientos mucho más eficaz.

De todas formas, el objetivo de DeepMind no era derrotar a Stockfish, ni tenía mucho que ver con el ajedrez. Este enfrentamiento tan solo era una forma de demostrar el potencial de la inteligencia artificial para aprender a dominar una disciplina tan complicada como el ajedrez desde cero. Es por eso, que en su primer match no se preocuparon en optimizar el rendimiento de Stockfish. Con hacer un buen papel contra una configuración razonable de Stockfish era suficiente.

2.1.2.3 Estudios realizados sobre AlphaZero

Durante el enfrentamiento, se analizó la relativa eficacia del MCTS de AlphaZero comparada con la de la puntera versión de la poda alfa-beta usada por Stockfish y Elmo. AlphaZero solo calculaba 80 mil posiciones por segundo en ajedrez y 40 mil en shogi, comparadas con las 70 millones que calculaba Stockfish y 35 millones que calculaba Elmo. AlphaZero compensa su menor número de evaluaciones usando su red neuronal para centrarse mucho más selectivamente en las variantes más prometedoras – sin duda una forma de calcular mucho más “humana”.

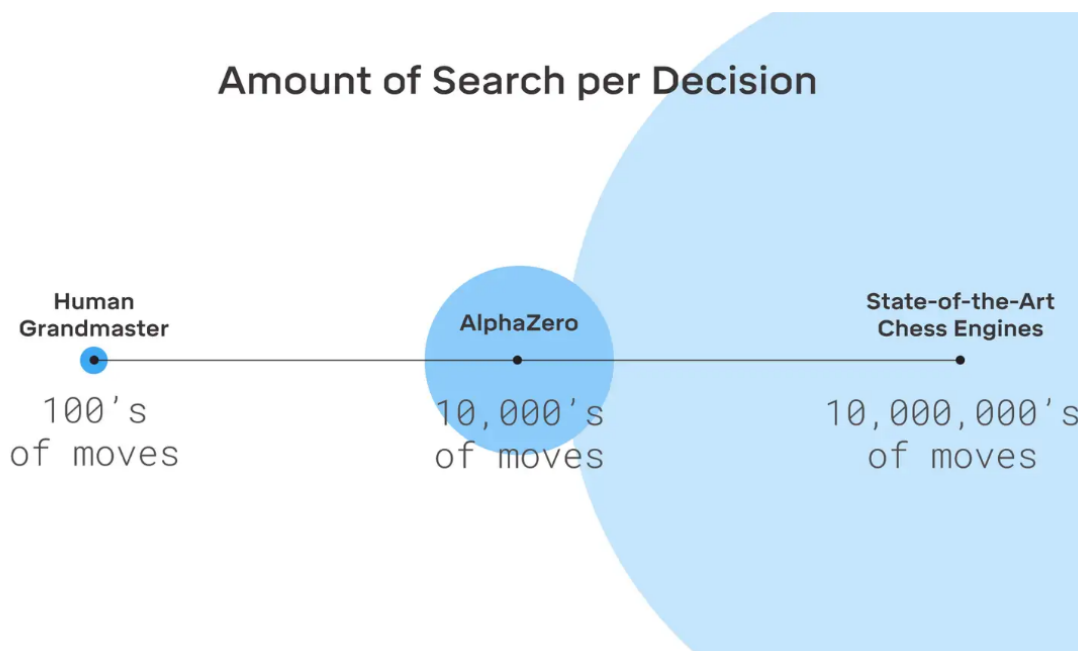


Ilustración 7. Cantidad de jugadas analizadas por decisión

La siguiente figura muestra la adaptabilidad de cada jugador respecto al tiempo por jugada, medido en una escala de Elo relativa a Stockfish o Elmo con un tiempo por jugada de 40 ms. El MCTS de AlphaZero progresaba más efectivamente a medida que se le dejaba más tiempo para pensar que Stockfish o Elmo, lo que pone en duda la tan arraigada creencia de que la poda alfa-beta es intrínsecamente superior en estos campos.

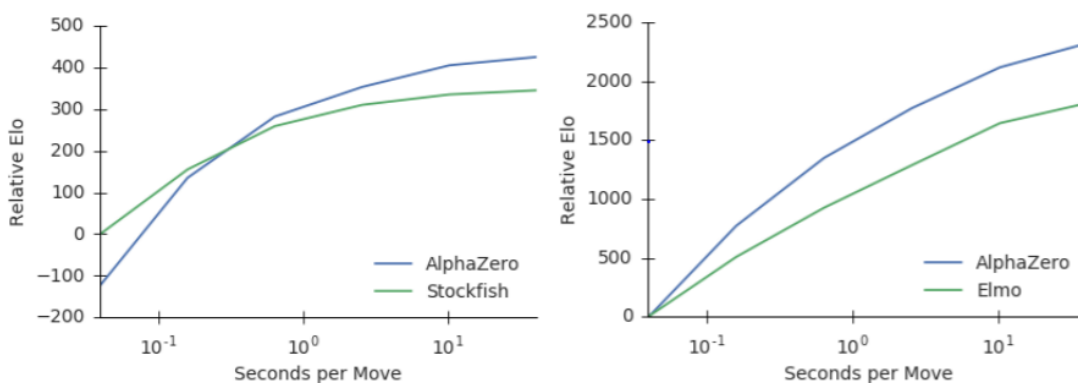


Ilustración 8. Adaptabilidad de AlphaZero con el tiempo disponible para pensar, medido en una escala Elo relativa al nivel de AlphaZero y Elmo con 40 ms por jugada.

Finalmente, se analizó el conocimiento ajedrecístico descubierto por AlphaZero. La siguiente tabla analiza las aperturas humanas más comunes (aquellas jugadas más de 100 000 veces en una base de datos online). Cada una de estas aperturas fue independientemente descubierta y jugada frecuentemente por AlphaZero durante las partidas de entrenamiento contra sí mismo. Empezando desde cada apertura humana, AlphaZero vencía de forma contundente a Stockfish, sugiriendo que ha dominado con maestría un amplio espectro del juego del ajedrez. También resulta alentador confirmar

que todo el esfuerzo que el ser humano ha dedicado al estudio de aperturas ha estado bien encaminado.

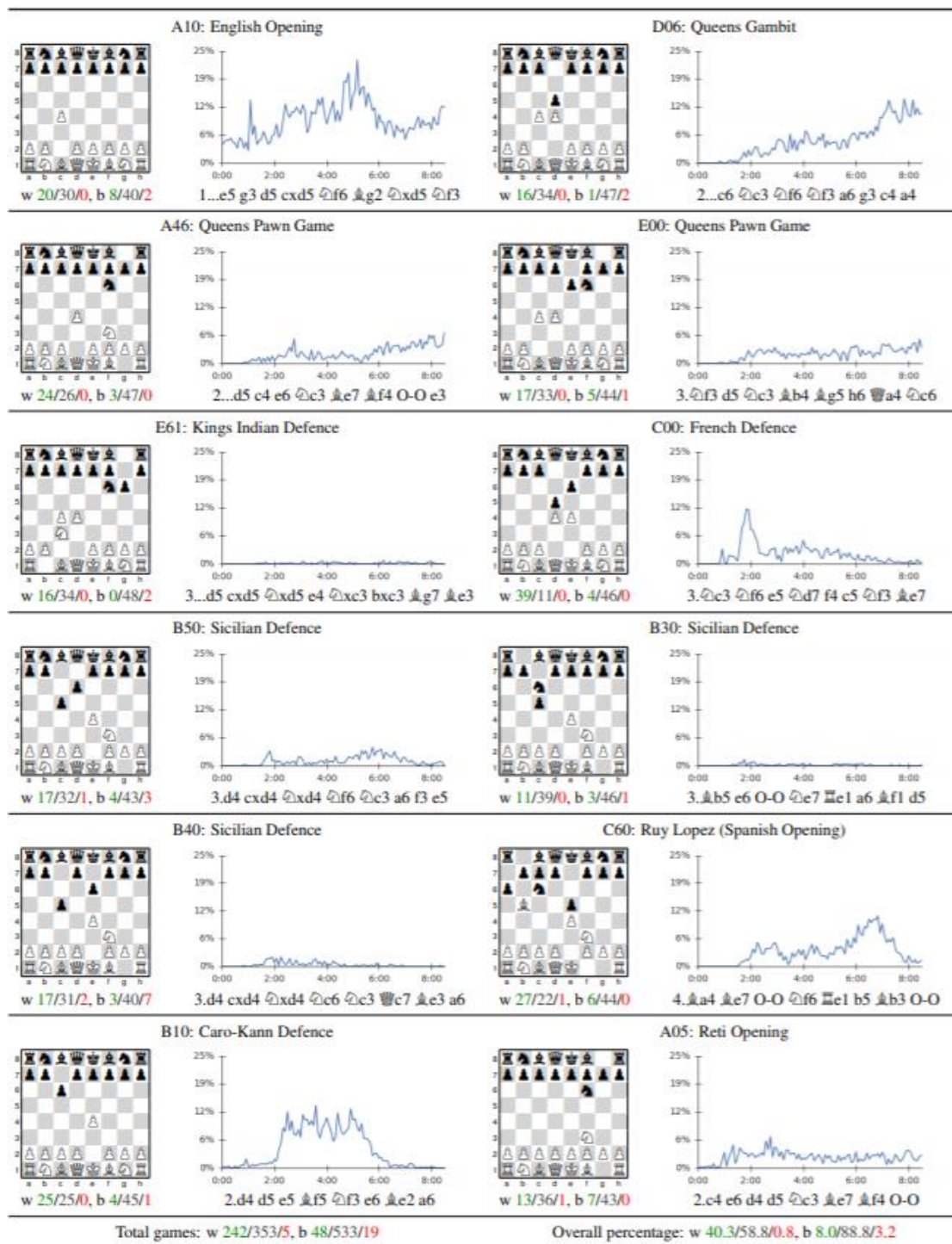


Ilustración 9. Análisis de las aperturas humanas más populares que fueron jugadas por AlphaZero durante el entrenamiento

El juego del ajedrez representa la cúspide de la investigación en inteligencia artificial llevada a cabo durante varias décadas. Los programas de vanguardia están basados en motores poderosos que calculan muchos millones de posiciones y se apoyan en el conocimiento humano para tomar decisiones.

AlphaZero es un algoritmo genérico de aprendizaje reforzado – originalmente concebido para Go – que consiguió mejores resultados al cabo de unas horas, calculando muchísimas menos posiciones y sin ningún tipo de conocimiento al respecto excepto las reglas del ajedrez. Además, el mismo algoritmo fue aplicado sin ninguna modificación al juego del shogi, venciendo nuevamente al estado del arte al cabo de unas pocas horas.

Es evidente que el ajedrez ha sido muy sensible al desarrollo tecnológico e informático, y ofrece un análisis de la inteligencia artificial desde una perspectiva realmente interesante.

2.2 Entorno de aplicación

Ahora bien, el ajedrez ya gozaba de cierta popularidad antes de la incursión de las máquinas en las 64 casillas, y es que todavía a día de hoy, aunque estas nos hayan superado con creces en lo que al dominio del juego se refiere, uno no piensa en ordenadores cuando oye la palabra “ajedrez”.

Para la inmensa mayoría de las personas el ajedrez es considerado como un juego, aburrido para algunos y entretenido para otros, pero un juego, a fin de cuentas. Para aquellos que compiten puede ser considerado como un deporte, con un grado de competitividad muy exigente al más alto nivel. También se puede llegar a considerar como una ciencia, por la extrema complejidad de su estudio, o como un arte, puesto que se puede crear belleza en las partidas (y es que no por casualidad existe un selecto número de partidas, conocidas como “Las inmortales”, que han pasado a la historia por la genialidad de sus jugadas).

Pero sin lugar a dudas, una de las consideraciones más interesantes, e inexplorada hasta hace poco, es la del ajedrez como herramienta pedagógica. De hecho, a día de hoy hay ya en España más de 300 colegios en los que el ajedrez es una asignatura obligatoria, siguiendo la recomendación de la Unesco y el Parlamento Europeo. Esto es debido a que está científicamente demostrado que el ajedrez potencia no sólo el desarrollo intelectual, sino también la formación en carácter y la formación en valores.



Ilustración 10. Ajedrez en las escuelas

2.3 Objetivo del TFG y definición del problema a resolver.

Este proyecto tratará de unificar las dos ideas clave presentadas previamente. El objetivo es programar un motor de ajedrez, para que los jóvenes tengan siempre a su disposición un rival con el que dar los primeros pasos en el juego.

Teniendo en cuenta a quién está dirigido el proyecto, conviene fijar una serie de requisitos que serán imprescindibles para que el programa sea bien recibido:

- El nivel de juego del motor no debe ser excesivamente alto. Si bien puede ser interesante que los niños se vean vencidos en las primeras partidas, estos no deben sentirse sobrepasados en ningún momento. De lo contrario existe el riesgo de que se desmotiven y abandonen la práctica del ajedrez.
- La máquina debe ser capaz de jugar relativamente rápido, pues un niño tiende a aburrirse rápidamente ante la falta de estímulos en cualquier actividad.
- Se pretende llegar al máximo número de niños posibles, por lo que el programa debe ser fácilmente ejecutable desde cualquier ordenador, independientemente de lo moderno que sea o del sistema operativo que utilice.

Se busca así colaborar en el uso del ajedrez como herramienta pedagógica. De esta forma, los jóvenes sacarán partido de los numerosos beneficios de la práctica del ajedrez, detallados algunos de ellos en un apartado posterior.

3 Alcance del proyecto

Como se ha mencionado en el apartado anterior, a día de hoy existen dos enfoques principales a la hora de programar un motor de ajedrez.

El enfoque tradicional consiste en enseñar a la máquina a jugar al ajedrez. El programador ha adquirido previamente una serie de conocimientos estratégicos sobre el juego que va a intentar implantar en la máquina. Este conocimiento estratégico unido a la enorme capacidad de cálculo de los ordenadores modernos puede desembocar en motores tan fuertes como Stockfish.

El otro enfoque y el que más de moda está hoy en día, consiste en enseñar a la máquina únicamente cómo se mueven las piezas. Posteriormente, y por medio de algoritmos relacionados con el “machine learning”, será la máquina la que aprenda a jugar al ajedrez. Cabe destacar que en esta ocasión el programador no tiene por qué haber adquirido previamente conocimientos de ajedrez, aunque sí requerirá de conocimientos mucho más avanzados de programación, al ser necesaria la implementación y entrenamiento de redes neuronales artificiales en el proceso de aprendizaje de la máquina.

Este proyecto tratará de estudiar los dos enfoques. Analizará las ventajas e inconvenientes de cada uno de ellos y a continuación elegirá el más apropiado para la consecución del objetivo buscado.

En lo referente al primer enfoque, se analizará cómo se puede plasmar el conocimiento humano de un juego tan complicado como el ajedrez en líneas de código. Se esquematizará la valoración estratégica de una posición clasificándola en diferentes grupos y se averiguará en qué medida mejora o empeora el rendimiento de la máquina cada uno de estos grupos. Además, se estudiarán los algoritmos más populares para calcular jugadas y se observará cuán grande es la carga computacional que se obtiene al añadir ramas al árbol de variantes.

Para abordar el segundo enfoque se pretende estudiar el funcionamiento de AlphaZero, procurar entender sus algoritmos y la forma en la que lleva a cabo el proceso de autoaprendizaje. Posteriormente, se intentará analizar si existe alguna ventaja adicional de decantarse por este enfoque en lo que al ajedrez como herramienta pedagógica se refiere.

Así pues, mediante este proyecto se pretende entregar el software mediante el cual un niño tenga la posibilidad de disputar una partida de ajedrez contra la computadora, y un análisis detallado del mismo.

4 Beneficios aportados por el proyecto

4.1 Beneficios sociales

Los beneficios aportados por el proyecto son íntegramente sociales. Como ya se ha comentado, la idea es facilitar a los niños el aprendizaje del ajedrez, otorgándoles un rival que esté siempre disponible y juegue a un nivel asequible para ellos. La práctica del ajedrez presenta beneficios para todas las edades en general, pero en el caso de los niños, les ayuda de una manera especial en una triple vertiente: desarrollar sus capacidades intelectuales, formarse en carácter y formarse en valores. Se detallan algunos de estos beneficios a continuación:

4.1.1 Desarrollo intelectual

4.1.1.1 Poder de concentración

Todo aquel que haya interactuado con niños se habrá podido dar cuenta de que si hay algo que les pierde, es la falta de concentración. Precisamente, en ajedrez éste es el peor error. De nada sirve haber jugado una partida extraordinaria durante 40 jugadas, si en la jugada 41 falla la concentración. Si se comete un solo error grave, se pierde la partida. Una lección que un joven ajedrecista no tardará mucho en aprender.

4.1.1.2 Creatividad e imaginación

En este juego no solo es necesario estudiar jugadas y seguir un patrón de movimientos determinado, sino que en ocasiones es interesante usar la imaginación y prever diferentes posibilidades de lo que puede ocurrir durante la partida. Asimismo, es necesario crear jugadas sorprendentes e inesperadas.

4.1.2 Formación en carácter

4.1.2.1 La toma de decisión

El ajedrez obliga a sus participantes a tomar decisiones con responsabilidad. Durante la partida, el niño se enfrenta a diferentes problemas y debe aplicar una estrategia a la vez que tiene en cuenta la del contrario. El tiempo es un factor importante en la partida, por lo que, además, el niño aprende a tomar decisiones bajo presión. Incluso muchas veces no existe ninguna jugada buena, por lo que el joven también aprende a optar por el menor de los males.

4.1.2.2 Pensamiento autocrítico

La victoria y la derrota se desarrollan en ajedrez de una manera muy especial. A diferencia de otras disciplinas, la suerte prácticamente no influye y no se puede excusar la derrota en el tiempo o el equipo. Cuando se juega al ajedrez se está solo frente al tablero y el resultado de la partida depende únicamente de uno mismo. Por lo tanto, en ajedrez, el que pierde es el que más aprende. Tras una derrota, se tiende a dedicarle un tiempo a la reflexión de forma prácticamente subconsciente. ¿Por qué he perdido hoy? ¿Qué puedo hacer la próxima vez para no cometer un error similar? Es decir, se desarrolla el pensamiento autocrítico de una manera muy intensa.

4.1.2.3 Organización y planificación

Los juegos de estrategia se han revelado como una forma extraordinaria de desarrollar la parte de nuestro cerebro que se dedica a la planificación, la gestión del tiempo y la organización.

Este es un punto muy interesante, porque según afirman los grandes fisiólogos, el cerebro justamente fue creado para direccionar (saber orientarse y tener idea de donde buscar alimentos) y planificar (anticiparse a acontecimientos como la llegada de la noche y actuar en consecuencia).

4.1.3 Formación en valores

4.1.3.1 Aceptación de reglas

Como en cualquier otro juego, el ajedrez cuenta con sus propias reglas cuyo incumplimiento no es aceptable en ningún caso.

4.1.3.2 Inteligencia emocional

Favorece el equilibrio entre lo racional y lo emocional, haciendo que los pequeños acepten y aprendan a encajar tanto los triunfos como los fracasos.

5 Primer enfoque: motor de ajedrez convencional

En este apartado se va a analizar la programación de un motor de ajedrez por el método tradicional. Esta forma de programar ha dominado el mundo del ajedrez computacional desde la aparición de la computadora en la década de los 50 hasta prácticamente el día de hoy, alcanzando su máxima popularidad en el año 1997, cuando tuvo lugar el carismático enfrentamiento entre Kasparov y Deep Blue.

5.1 Módulos de un motor de ajedrez

Todo motor de ajedrez consta principalmente de tres módulos: el generador de movimientos, la función de evaluación y la función de búsqueda.

5.1.1 Generador de movimientos

Este primer módulo es el encargado de definir las jugadas posibles dada una posición, teniendo en cuenta que estas jugadas sean legales en base a las reglas del ajedrez.

5.1.2 Función de evaluación

La función de evaluación es una parte muy importante de un motor de ajedrez, y prácticamente todas las mejoras de los motores de ajedrez más fuertes de la actualidad se dan en la función de evaluación.

Este módulo realiza una evaluación estática de una posición. Es decir, la entrada a dicha función es la posición que queremos analizar y la salida es una valoración, representada con un número entero.

La forma de interpretar este número que obtenemos a la salida es la siguiente. Partimos de que 0 representa la igualdad absoluta entre ambos bandos. A partir de aquí, cuanto más positivo sea el número, más ventaja tendrán las blancas, y cuanto más negativo sea el número, más ventaja tendrán las negras.

Los razonamientos en los que se basan los módulos para definir este número pueden ser tan variados como los razonamientos en los que se basa un gran maestro de ajedrez para evaluar una posición. Se presentan los utilizados por Stockfish a continuación:

5.1.2.1 Material

Se trata de otorgar un valor a cada pieza y realizar un recuento de las piezas que hay en la posición, para que matemáticamente se pueda calcular la pérdida o ganancia de material durante el juego.

5.1.2.2 Tablas de pieza-casilla

Cada pieza recibe un bonus dependiendo de en qué casilla se encuentre, independientemente de dónde se encuentren las demás piezas. De esta forma, se consigue que los peones tiendan a avanzar o que los alfiles y caballos tiendan a desarrollarse, por ejemplo.

5.1.2.3 Estructura de peones

Durante una partida de ajedrez puede darse el caso de que dos peones acaben en la misma columna, o que un peón no tenga obstaculizado el camino para llegar a la coronación por otro peón del bando enemigo. Estas situaciones se conocen como peón doblado y pasado respectivamente y generalmente resultan en una desventaja en el caso del peón doblado y una ventaja en el del pasado. Otros patrones a tener en cuenta a la hora de valorar la estructura de peones son conocidos en el argot del ajedrez como peones aislados, retrasados, islas de peones... todos ellos con su correspondiente bonus.

5.1.2.4 Evaluación específica de las piezas

Cada pieza recibe un bonus dependiendo en qué casilla se encuentre, pero esta vez se tienen en cuenta el resto de las piezas. Así pues, una torre en una columna abierta o un caballo bloqueando un peón serán mejor valorados.

5.1.2.5 Movilidad

En ajedrez la actividad de las piezas es uno de los factores más importantes, y esto está estrechamente unido con la movilidad de las piezas, o lo que es lo mismo, con el número de casillas que tienen disponibles para realizar sus movimientos.

5.1.2.6 Seguridad del Rey

El objetivo final del juego es acorralar al rey, por lo que a la hora de realizar una buena valoración no se puede pasar por alto este factor. Una buena estructura de peones alrededor de este será valorada positivamente, mientras que piezas atacantes del rival a pocas casillas del mismo, acarrearán una valoración negativa.

5.1.2.7 Amenaza

Las piezas indefensas son penalizadas, mientras que las piezas bien defendidas reciben un bonus.

5.1.2.8 Espacio

Se recibe un bonus por tener casillas vacías y “seguras” en nuestro lado del tablero.

5.1.2.9 Tablas muertas

Con ciertas combinaciones de piezas resulta imposible decantar el resultado hacia uno de los bandos. Por ejemplo, un caballo y un rey son incapaces de dar jaque mate al otro rey, por lo que la valoración pasa automáticamente a 0.0.

5.1.2.10 Control del centro

Las cuatro casillas centrales del tablero son un excelente punto de apoyo estratégico, por lo que se recibirá un bonus en caso de controlarlas.

Todos estos conceptos que se acaban de analizar no dejan de ser parte de las conclusiones a las que el ser humano ha llegado, principalmente en base a la experiencia, tras siglos de desempeño en el juego. No obstante, nunca hay que dejar de tener en cuenta que, en ajedrez, cada posición es un mundo, y existen excepciones por todos los lados. Es aquí donde fallan los motores de ajedrez convencionales, ya que

no es posible realizar una función de evaluación que sea objetiva en el 100% de los casos, y cuando se trata de continuar progresando al más alto nivel, son estos pequeños detalles los que marcan la diferencia.

5.1.3 Función de búsqueda

Este segundo módulo es el encargado de calcular diferentes variantes dada una posición. Se pretende seguir un razonamiento bastante similar al humano. Muchas veces, una mera evaluación estratégica de una posición no es suficiente para saber cual es la jugada correcta. Uno puede realizar una jugada que le otorgue control del centro, actividad de sus piezas, ventaja de espacio e incluso esperanzas de un ataque prometedor, pero de nada sirve si dicha jugada lleva a un mate forzado a favor del rival tres jugadas más allá.

Es por esta razón que el cálculo de variantes es de vital importancia en el juego del ajedrez, tanto para las máquinas como para los humanos. El planteamiento correcto en una partida sería calcular tantas jugadas como nos sea posible, o suficiente si supiéramos que hemos llegado a una posición tranquila, y solo entonces proceder a una evaluación estratégica de la posición.

Existen varios algoritmos para hacer que la máquina calcule, pero los dos más populares son minimax y la poda alpha-beta.

5.1.3.1 Algoritmo minimax

La concepción del principio minimax se le atribuye a John von Neumann, quien en su artículo de 1928 "Sobre la teoría de los juegos de sociedad" asentó las bases de la moderna teoría de juegos y probó el teorema fundamental del minimax, por el que se demuestra que para juegos de suma cero con información perfecta entre dos competidores existe una única solución óptima.

Para entender cómo funciona este algoritmo es importante recordar lo que significa un juego de suma cero. Esto quiere decir que lo que supone una ventaja de cierta magnitud para un jugador, supone una desventaja de la misma magnitud para el otro jugador. Si las blancas capturan un caballo, entonces las negras pierden un caballo, de tal forma que 1 caballo que tienen las blancas de ventaja - 1 caballo que tienen las negras de desventaja = 0. Este es un ejemplo muy simple, pero se podría llegar a la misma conclusión después de analizar una posición muy complicada.

La siguiente idea que hay que tener presente para entender este algoritmo es que cada jugador va a optar por las jugadas que más le convengan a él. Y según hemos deducido en el párrafo anterior, la jugada que más le convenga a un jugador será la que menos le convenga al otro jugador.

Así pues, el algoritmo minimax, como su propio nombre indica (minimizing the maximum loss), tratará de encontrar la jugada que nos lleve a una posición más ventajosa teniendo en cuenta que nuestro rival va a tratar de responder con aquellas jugadas que menos nos favorezcan. La mejor forma de entender este concepto es mediante un árbol de variantes:

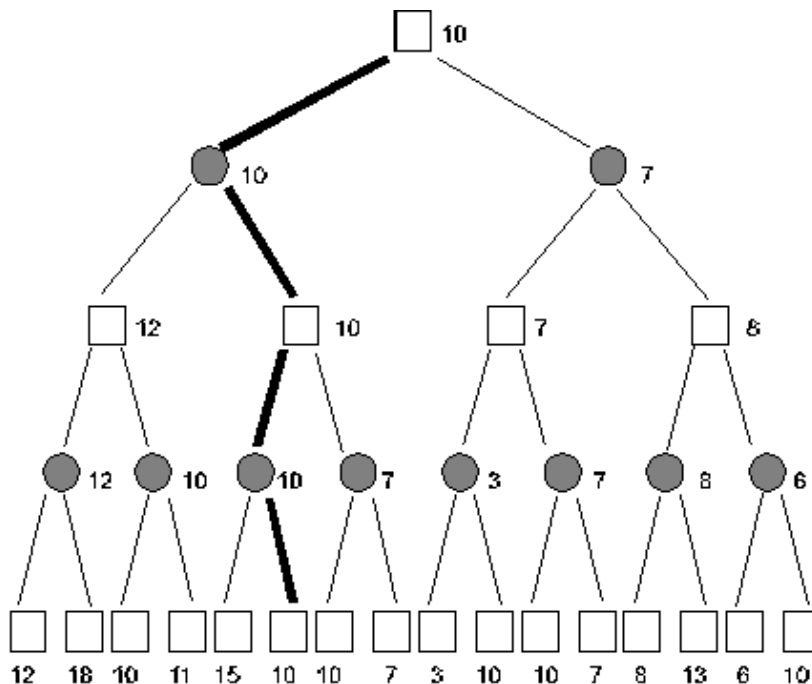


Ilustración 11. Esquema del árbol de variantes minimax

Aquí se puede observar una posición que está siendo analizada a profundidad 4. Esto quiere decir que la máquina calculará todas nuestras jugadas posibles, luego todas las jugadas posibles de nuestro rival a cada una de nuestras jugadas posibles, y así sucesivamente hasta llegar a 4 movimientos más allá. A continuación, la máquina pasará a analizar todas las posiciones resultantes y les asignará un valor numérico. Partiendo que consideramos 0 como igualdad, cuanto más positivo sea ese valor más ventaja tendrán las blancas, y cuanto más negativo sea más ventaja tendrán las negras.

Estas valoraciones se corresponden con la última fila del árbol de la figura. A partir de aquí empezamos un proceso ascendente de elección de jugadas. Como podemos ver eran las negras las que realizaban la jugada inmediatamente anterior, y éstas tratarán de escoger la jugada más conveniente para ellas, o lo que es lo mismo, la jugada más negativa. Así pues, entre 12 y 18 eligen 12, entre 10 y 11 eligen 10, entre 15 y 10 eligen 10, entre 10 y 7 eligen 7, entre 3 y 10 eligen 3, entre 10 y 7 eligen 7, entre 8 y 13 eligen 8, y entre 6 y 10 eligen 6.

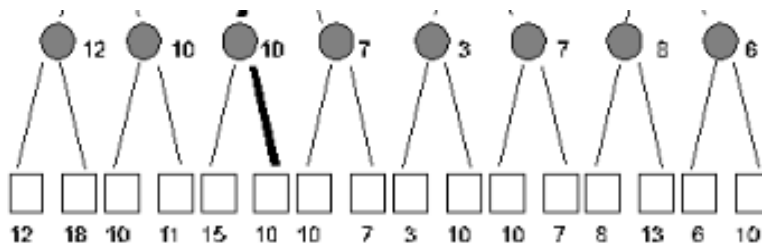


Ilustración 12. Esquema del árbol de variantes minimax. Primer paso

A continuación, pasamos al siguiente nivel, y ahora son las blancas las que tienen que elegir. Nuevamente, las blancas escogerán la jugada más conveniente para ellas, o lo que es lo mismo, la jugada más positiva. Entre 12 y 10 eligen 12, entre 10 y 7 eligen 10, entre 3 y 7 eligen 7, y entre 8 y 6 eligen 8.

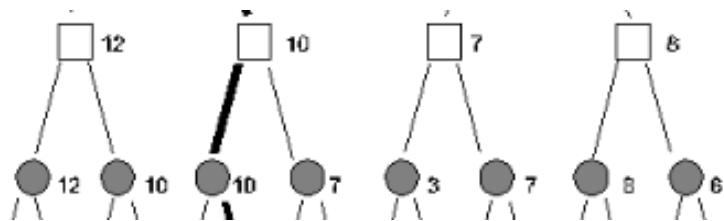


Ilustración 13. Esquema del árbol de variantes minimax. Segundo paso

En el siguiente nivel, les toca nuevamente a las negras y estas optan, una vez más, por la jugada más negativa posible. Entre 12 y 10 eligen 10, y entre 7 y 8 eligen 7.

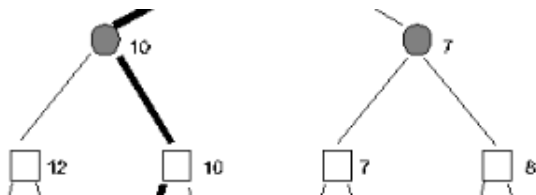


Ilustración 14. Esquema del árbol de variantes minimax. Tercer paso

Por último, son las blancas las que juegan ahora, y eligen la jugada más positiva posible. Entre 10 y 7 se quedan con 10.

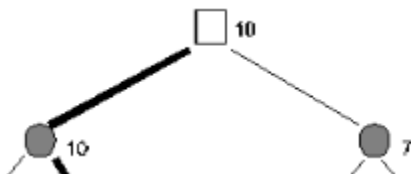


Ilustración 15. Esquema del árbol de variantes minimax. Cuarto paso

Con todo, ya habríamos trazado el camino a seguir durante las próximas 4 jugadas, que llevaría a una valoración de 10. Cabe resaltar que el proceso de selección se realiza en orden ascendente, pero las jugadas propiamente dichas se ejecutan en el tablero en orden descendente, siguiendo el camino marcado en negrita a lo largo del árbol de jugadas.

Otro aspecto que merece la pena resaltar es que el número de variantes reflejadas en este ejemplo no tiene nada que ver con la realidad. Aquí, tan solo se han tenido en cuenta dos posibles respuestas en cada posición, por lo que el número de posiciones resultantes a profundidad 4 ha resultado ser $2^4 = 16$. En la realidad se podría estimar que el número de jugadas posibles dada una posición es de alrededor de 30. Esto quiere decir que el número de posiciones resultantes a profundidad 4 sería $30^4 = 810000$. Con tan solo 4 jugadas de profundidad ya nos podemos hacer a la idea de que carga de trabajo va a tener la máquina, pues va a tener que analizar nada menos que 810000 posiciones y luego empezar el proceso de selección descrito anteriormente.

Pues bien, este número de posiciones resultantes aumenta exponencialmente si quisiéramos calcular a profundidades mayores, por lo que aun con la tecnología existente hoy en día, resulta inabordable a partir de ciertas profundidades. De hecho, teniendo en cuenta que una partida de ajedrez tiene alrededor de 40 jugadas (80 movimientos en total), si una máquina fuera capaz de analizar 30^{80} posiciones, que es

igual a 10^{123} posiciones aproximadamente, el juego del ajedrez estaría resuelto. Algo poco factible teniendo en cuenta que se estima que en el universo existen entre 4×10^{78} y 6×10^{79} átomos.

5.1.3.2 Poda alfa-beta (alpha-beta pruning)

La poda alfa-beta es una mejora del algoritmo minimax. Si se aplica correctamente, devuelve exactamente la misma solución que el algoritmo minimax, solo que reduciendo considerablemente el número de nodos evaluados.

Entre los pioneros en el uso de esta técnica encontramos a Arthur Samuel, D.J. Edwards y T.P. Hart, Alan Kotok, Alexander Brudno, Donald Knuth y Ronald W. Moore. A día de hoy es el algoritmo más usado para calcular variantes y aunque con algunas mejoras adicionales, es también el usado por Stockfish.

Para explicar la idea subyacente bajo este algoritmo, tomaremos como referencia el siguiente árbol de variantes:

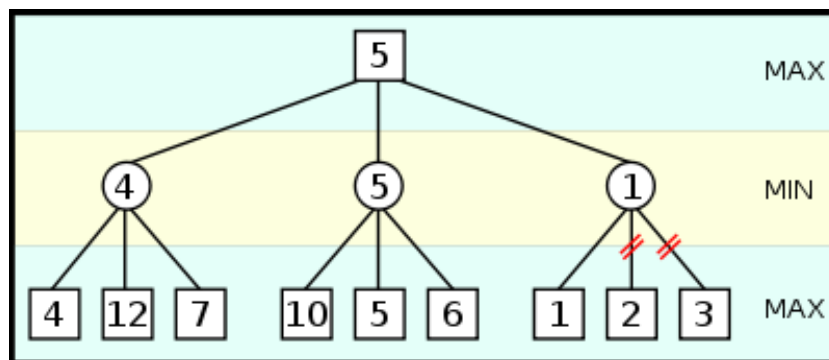


Ilustración 16. Esquema del árbol de variantes de la poda alfa-beta

Podemos observar, que la máquina juega con blancas (pues en la posición original tenemos a max) y ha calculado las posiciones resultantes para profundidad 2, evaluando todas las posiciones resultantes y asignándoles sus respectivos valores.

Empezamos pues el proceso ascendente, entre 4, 12 y 7, min elige 4 (el valor más pequeño posible), entre 10, 5 y 6, min elige 5, y entre 1, 2 y 3, min elige 1. Pasamos al siguiente nivel y como era de esperar, entre 4, 5 y 1, max elige 5 (el valor más alto posible).

Ahora bien, podemos comprobar que no era necesario que la máquina evaluara las dos últimas posiciones (las de valores 2 y 3), ya que una vez que sabe el resultado de la primera evaluación de esa rama (la de valor 1), min elegirá en el mejor de sus casos un valor que sea igual o inferior a uno. Pero al pasar al siguiente nivel le toca elegir a max, y teniendo en cuenta que max puede elegir entre 4, 5 y un valor igual o inferior a 1, max elegirá la opción de 5 independientemente de cual sea la valoración de las dos últimas posiciones.

Como resultado de este razonamiento, llegamos a la conclusión de que las dos últimas ramas del árbol pueden ser podadas como se muestra en la imagen, evitando así evaluar dos posiciones y perder el tiempo que ello nos requeriría.

Para traducir esta idea al lenguaje de programación se definen dos límites, α y β , que corresponden a la evaluación más conveniente encontrada hasta el momento para las blancas y las negras respectivamente. Es por eso, que estos límites se definen inicialmente como $-\infty$ y $+\infty$, e irán siendo actualizados a medida que se vayan evaluando diferentes variantes. En el caso de max, si la evaluación de una nueva posición nos da como resultado un valor superior al anterior, se le asignará a alfa este valor. En el caso de min, si la evaluación de una nueva posición nos da como resultado un valor menor que el anterior, se le asignará a beta este valor. En el caso de max, si alpha fuera mayor que beta será que la posición es tan buena para las blancas que las negras no la permitirán jamás, por lo que se podría romper el bucle y dejar de evaluar ahí. En el caso de min, si beta fuera menor que alfa será que la posición es tan buena para las negras que las blancas no la permitirán jamás, por lo que se podría romper el bucle y dejar de evaluar ahí.

5.2 Aspectos adicionales a tener en cuenta

Se ha comentado en el apartado anterior que Stockfish utiliza una versión mejorada de la poda alfa-beta para calcular variantes. Algunas de las principales ideas de mejora que podemos encontrar en su código, son las siguientes:

5.2.1 Ordenación

La eficiencia en la poda alfa-beta depende del orden de los movimientos en que se realiza esta operación. Desafortunadamente, ordenar los movimientos de la mejor forma implica encontrar los mejores y buscar primero sobre estos, lo cual es una tarea bastante difícil de lograr.

Por ejemplo, el orden podría iniciarse con capturas, coronaciones de peón (las cuales cambian dramáticamente el balance de material) o jaques (los cuales a menudo permiten pocas respuestas legales), siguiendo con movimientos que causaron recientes cortes en otras variantes a la misma profundidad (denominados jugadas-asesinas, killer-moves) y por último observar el resto de los movimientos.



Ilustración 17. Orden de búsqueda de movimientos

5.2.2 Jugadas-asesinas (killer-moves)

Varias estrategias existen para guardar los movimientos asesinos. Lo más simple es mantener una lista bastante corta de a lo más dos movimientos de profundidad. Para un mejor entendimiento de este concepto, se muestra un ejemplo a continuación:



Ilustración 18. Posición que ilustra el concepto de jugada asesina

La figura ilustra el funcionamiento de esta heurística. Si es el turno de mover de las blancas, éstas intentarían el movimiento 1.Cxh6 debido a que captura la torre. Luego de examinar las réplicas del negro encontrará que este movimiento es refutado por la respuesta 1...Ta1 mate. Entonces, cuando el programa examine nuevos movimientos para el blanco, el primer movimiento negro que tomará como respuesta será 1...Ta1 debido a que es un movimiento legal que genere un corte en una variante anterior.

5.2.3 Efecto horizonte

El efecto horizonte es un problema de los motores de ajedrez convencionales que puede ocurrir cuando dado un nodo, todos los movimientos se calculan a una determinada profundidad. Puede darse el caso de que una vez alcanzada dicha posición, el módulo realice una mala jugada pensando que va a evitar así una pérdida mayor, cuando en realidad lo único que está haciendo es posponer esa pérdida.

Para entender mejor este concepto se toma la siguiente posición a modo de ejemplo:



Ilustración 19. Posición que ilustra el concepto de efecto horizonte

Se puede comprobar que la torre de las blancas está atrapada en la casilla a5, por lo que cualquier esfuerzo por salvarla sería en vano. Ahora bien, el motor de ajedrez está analizando la posición a profundidad 1 (una jugada de las blancas y una de las negras). Tras una breve reflexión decide hacer la jugada 1.Axf7+. Parece una jugada extraña, pero desde el punto de vista de la máquina está llena de lógica, ya que una jugada más allá (que es hasta donde llega la profundidad del motor), las blancas habrán perdido el alfil, pero aún conservarán la torre. La sorpresa llega cuando, tras la jugada Txf7 de las negras, el motor se da cuenta de que la torre sigue estando perdida, por lo que el sacrificio del alfil ha sido en vano.

La solución es lo que se conoce como la búsqueda de quiescencia.

5.2.4 Búsqueda de quiescencia (quiescence search)

La búsqueda de quiescencia consiste en no limitar la profundidad de análisis con un simple número, sino que el módulo analiza las variantes hasta que la posición es suficientemente estable, como para poder ser analizada de forma estática.

Como ya se ha comentado previamente, los jugadores humanos tienen suficiente intuición como para saber cuándo abandonar el cálculo de una jugada fea, o continuar examinando una jugada prometedora. La búsqueda de quiescencia trata de estimular este comportamiento aconsejando al ordenador calcular más profundamente posiciones “volátiles” que posiciones “tranquilas”.

Cualquier criterio razonable puede ser tenido en cuenta a la hora de diferenciar entre posiciones “volátiles” y posiciones “tranquilas”. Un criterio bastante común es considerar que una posición es “tranquila” cuando no existen capturas ni amenazas.

Un pseudocódigo que ilustra este concepto sería el siguiente:

```

function quiescence_search(node, depth) is
  if node appears quiet or node is a terminal node or depth = 0 then
    return estimated value of node
  else
    (recursively search node children with quiescence_search)
    return estimated value of children

function normal_search(node, depth) is
  if node is a terminal node then
    return estimated value of node
  else if depth = 0 then
    if node appears quiet then
      return estimated value of node
    else
      return estimated value from quiescence_search(node, reasonable_depth_value)
  else
    (recursively search node children with normal_search)
    return estimated value of children

```

Ilustración 20. Pseudocódigo búsqueda de quiescencia

5.2.5 Movimiento nulo (null-move pruning)

Esta técnica permite reducir de forma drástica el factor de ramificación con un cierto riesgo de perder información importante. La idea es dar al oponente una jugada de ventaja, y si la posición sigue siendo buena, (alfa mayor que beta), se asume que el alfa real seguirá siendo mayor que beta y, por tanto, se poda esa rama y se siguen examinando otros nodos. Se ahorra tiempo porque se reduce la profundidad.

Por supuesto esta técnica falla en las posiciones de zugzwang, esto es, las posiciones en las que cualquier movimiento permitido empeoraría la posición, y se desearía que fuera el turno del rival, como es el caso de la siguiente posición:

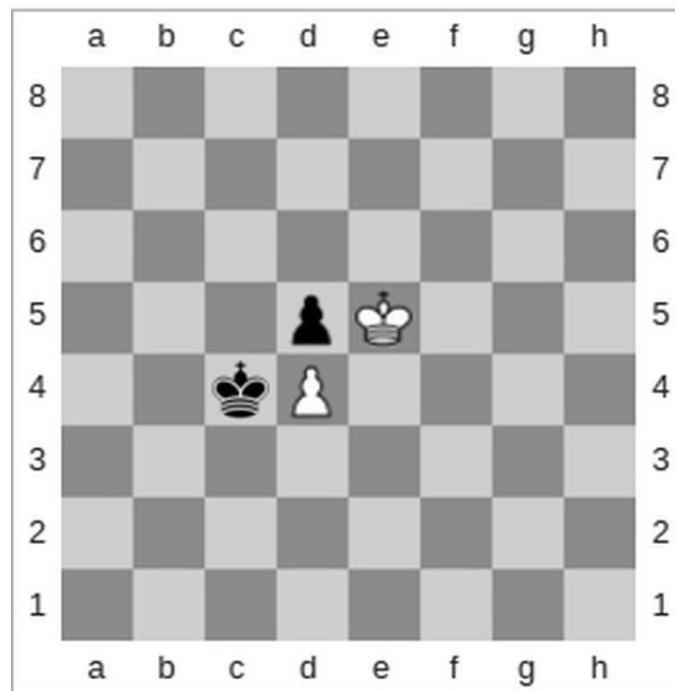


Ilustración 21. Posición de zugzwang

Aquí, ambos reyes están atados a la defensa de su peón, por lo que el bando al que le toque mover perdería dicho peón y, por consiguiente, la partida.

5.2.6 Bases de datos de partidas

Existen numerosas bases de datos en las que hay almacenadas millones de partidas que se han jugado con anterioridad. Durante las primeras jugadas de la partida, estas bases de datos son de gran utilidad, pues se puede tener una idea de cuan efectiva es una jugada analizando por ejemplo el número de victorias que se ha obtenido con ella.

Cabe mencionar también, que en la apertura el nivel de un gran maestro está incluso por encima que el de un motor de ajedrez contemporáneo que no cuenta con base de datos. Esto se debe, a que existe un gran estudio de la primera fase del juego que es fruto de muchos años de análisis y puesta en práctica. Últimamente, estos análisis se hacen incluso soportados por un motor de ajedrez, por lo que las jugadas novedosas en la apertura cuentan con la creatividad de los humanos y la capacidad de cálculo de las máquinas.

No es de extrañar, por tanto, que la mayoría de los motores suela traer integrada alguna base de datos con partidas de los mejores jugadores humanos del mundo.



Ilustración 22. Base de datos de partidas

5.2.7 Tablas de Nalimov

Mediante las Tablas de Nalimov, Stockfish consigue un juego perfecto durante el final. Las tablas de Nalimov son una base de datos que almacena todas las posibles posiciones con pequeños grupos de material. Estas posiciones están de manera concluyente determinadas como ganadoras, perdedoras, o tablas para el jugador que mueve.

También está determinado el número de jugadas para el final con el mejor juego de cada lado, así como la mejor jugada en cada posición incluida (identificando la jugada que gana más rápido contra una defensa perfecta, o la jugada que pierde más lentamente contra un ataque óptimo). Hay bases de datos disponibles para finales de 3 a 6 piezas (contando los reyes) y para algunos de 7 piezas.

Su construcción es inversa, es decir, partiendo de los diferentes finales posibles (mate o tablas) se analizan todas las posibles jugadas hacia atrás.

Negras ganan en 49	
Cd4	Negras ganan en 48
Ch4	Negras ganan en 49
Re6	Negras ganan en 50
Rc8	Negras ganan en 50
Re8	Negras ganan en 50
Rd8	Negras ganan en 50
Re7	Negras ganan en 50
Ce7	Negras ganan en 50
Cg7	Negras ganan en 50
d4	Tablas
Ch6	Blancas ganan en 12
Ce3	Blancas ganan en 10
Cg3	Blancas ganan en 10
Cd6	Blancas ganan en 10

Juegan blancas
 Juegan negras

Al paso

a b c d e f g h

1 2 3 4 5 6 7 8

Ilustración 23. Base de datos de finales

6 Segundo enfoque: AlphaZero

AlphaZero ha revolucionado el mundo del ajedrez computacional, no solo porque se ha proclamado el mejor jugador de ajedrez de la historia, sino también porque sus creadores lo han conseguido de una forma nueva, radicalmente diferente a la forma en la que se venían programando los motores de ajedrez.

Usando redes neuronales artificiales entrenadas, AlphaZero usa el aprendizaje automático (machine learning) para solucionar los siguientes problemas que plantea el ajedrez:

- Evaluar estáticamente una posición - estimar cuan buena es una posición sin calcular.
- Decidir qué variantes deben ser analizadas en profundidad y qué variantes deben ser descartadas.
- Ordenar los movimientos - determinar que movimientos analizar primero, lo que afecta significativamente a la eficiencia computacional.

De esta forma se busca que las redes neuronales artificiales pasen a ser un sustituto de la intuición, y consigan que los motores de ajedrez jueguen más eficientemente.

A continuación, se pretende analizar a AlphaZero con cierto nivel de detalle, apoyándose en el documento que DeepMind publicó tras el carismático enfrentamiento que tuvo lugar entre AlphaZero y Stockfish. No obstante, antes de entrar de lleno en este análisis, conviene recordar brevemente en que consiste el Árbol de Búsqueda Monte Carlo (MCTS), ya que éste es el método empleado por AlphaZero para calcular variantes.

6.1 Árbol de búsqueda Monte Carlo (Monte Carlo tree search)

El árbol de búsqueda Monte Carlo (MCTS, de sus siglas en inglés) es otra forma de abordar el problema del cálculo en la programación de un módulo de ajedrez. Aunque la elección de jugadas se hace de una forma similar al algoritmo minimax (se eligen las jugadas más prometedoras en función de una evaluación previamente determinada), el proceso llegar a dicha evaluación es radicalmente diferente.

Cada iteración de este proceso está formada por cuatro etapas principales:

6.1.1 Selección

Comenzamos por el nodo raíz, que representa la posición actual del tablero. A partir de ahí seleccionamos el nodo más “urgente” de acuerdo a una función de utilidad. En el caso de AlphaZero la función de utilidad es la siguiente:

$$U(s, a) = Q(s, a) + c_{puct} \cdot P(s, a) \cdot \frac{\sqrt{\sum_b N(s, b)}}{1 + N(s, a)}$$

Ecuación 1. Función de utilidad de AlphaZero

Básicamente, esta función prioriza la elección entre aquellos nodos que hasta el momento se consideren más prometedores por haber desembocado en mejores

resultados, o bien los nodos de los que no tengamos mucha información por haber sido visitados pocas veces. El proceso de selección continúa recursivamente hasta llegar a un nodo que represente un estado terminal o a un nodo que no haya sido extendido.

6.1.2 Expansión

Tras finalizar el proceso de selección nos encontramos en un nodo que ha sido visitado previamente pero no ha sido extendido. Es entonces cuando tiene lugar la etapa de expansión, que consiste en calcular los nodos “hijo” del nodo en el que nos encontrábamos.

6.1.3 Simulación

Una vez calculados los nodos hijo se selecciona uno de ellos al azar (pues la función de utilidad va a dar el mismo resultado para todos ellos) y se procede a una simulación “aleatoria” de una partida. La simulación puede ser aleatoria completamente, disponer de pequeñas heurísticas de ponderación o disponer de heurísticas computacionalmente costosas a cambio de estrategias más elaboradas.

6.1.4 Retropropagación

Finalmente, en la etapa de retropropagación, se actualizan las estadísticas de todos los nodos anteriores teniendo en cuenta el resultado de la última simulación.

De forma esquemática, se representa el algoritmo de cada iteración en la siguiente figura:

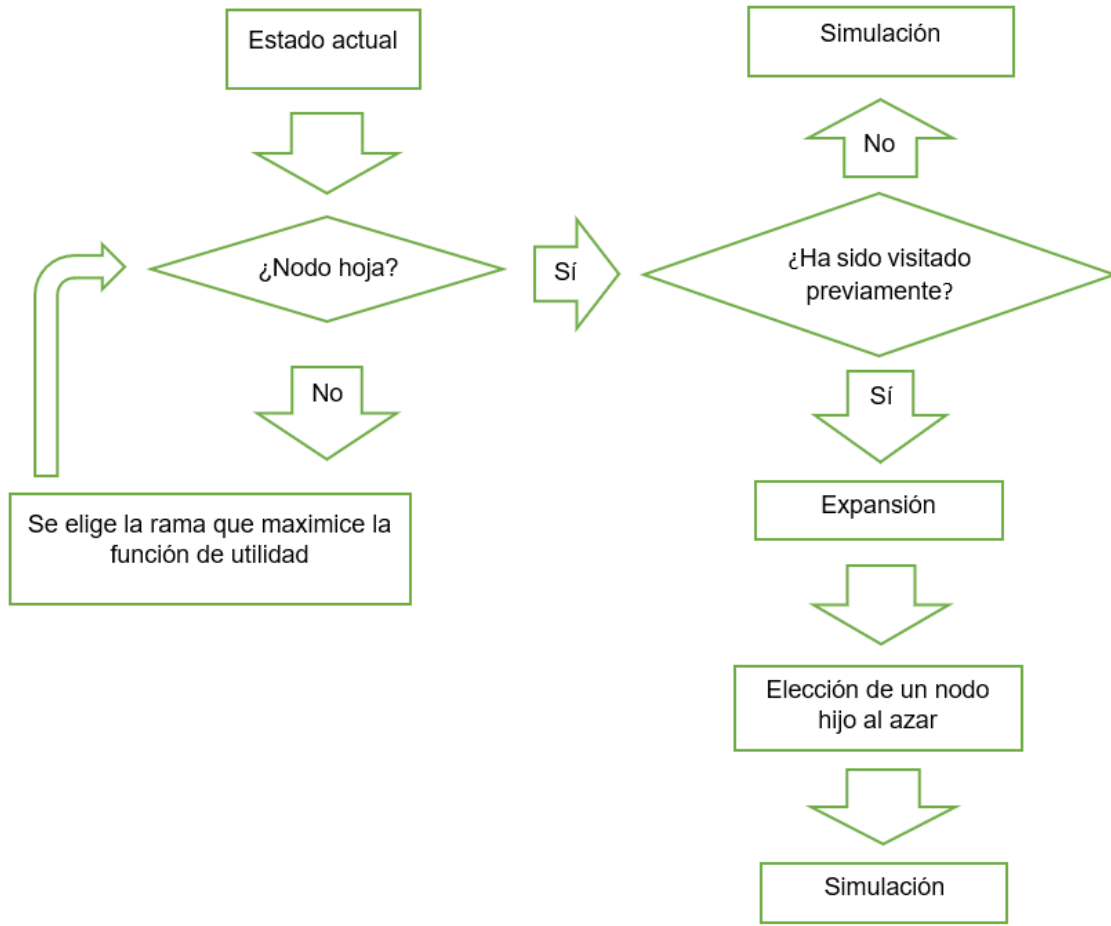


Ilustración 24. Esquema del algoritmo MCTS

Y un ejemplo de una iteración se representa en la siguiente figura:

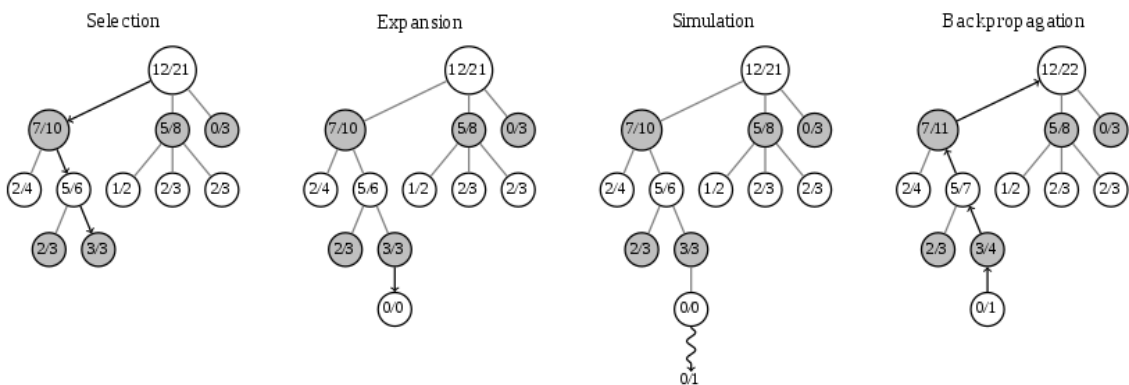


Ilustración 25. Ejemplo de iteración del MCTS

6.2 Arquitectura de AlphaZero

El algoritmo de AlphaZero es una versión genérica del algoritmo de AlphaGo Zero que fue inicialmente introducido en el contexto del Go. Este algoritmo sustituye las funciones de evaluación de los motores tradicionales en las que se intentaba implantar el conocimiento humano, por redes neuronales artificiales y un algoritmo de aprendizaje reforzado que comienza con una *tabula rasa*.

AlphaZero utiliza una red neuronal $(\mathbf{p}, v) = f_{\theta}(s)$ con parámetros θ . Esta red neuronal toma la posición del tablero s como entrada, y proporciona un vector de probabilidades de movimiento \mathbf{p} con componentes $p_a = Pr(a|s)$ para cada acción a , y un valor escalar v que estima el resultado esperado z desde la posición s , $v \approx E[z|s]$ en la salida. AlphaZero aprende estas probabilidades de movimiento y estimaciones de resultado exclusivamente jugando contra sí mismo, y luego las utiliza para guiar su búsqueda.

En vez de una poda alfa-beta con sus correspondientes mejoras, AlphaZero usa un árbol de búsqueda Monte-Carlo (MCTS) de propósito general. Cada búsqueda consiste en una serie de partidas simuladas contra sí mismo que atraviesan el árbol desde la raíz hasta algunas hojas. Cada simulación elige en cada estado s un movimiento a con pocas visitas, mucha probabilidad de movimiento y un valor alto (obtenido de la media de los estados hoja de las simulaciones que eligieron a en la posición s) de acuerdo con la red neuronal actual f_{θ} . La búsqueda proporciona un vector $\boldsymbol{\pi}$ que representa la distribución de probabilidad de los movimientos.

Los parámetros θ de la red neuronal son inicializados de forma aleatoria, y posteriormente son entrenados mediante aprendizaje reforzado mientras AlphaZero juega contra sí mismo. Las partidas se juegan eligiendo los movimientos de los dos bandos mediante el MCTS, $a_t \sim \boldsymbol{\pi}_t$. Al final de cada partida, mediante la posición terminal se computa el resultado de la partida de acuerdo a las reglas del ajedrez: $z = -1$ si es una derrota, 0 si tablas, y 1 si es una victoria. A continuación, los parámetros de la red neuronal θ son actualizados de tal forma que minimicen el error entre el resultado estimado v_t y el resultado de la partida z , y maximice la similitud entre el vector de probabilidades \mathbf{p}_t proporcionado por la red neuronal y el vector de probabilidades $\boldsymbol{\pi}_t$ proporcionado por la búsqueda. Específicamente, los parámetros θ son ajustados mediante descenso de gradiente en una función de pérdida l que suma el error cuadrático medio del resultado estimado v y real z , y la entropía cruzada entre las distribuciones de probabilidad de la red neuronal \mathbf{p} y de la búsqueda $\boldsymbol{\pi}$.

$$l = (z - v)^2 - \boldsymbol{\pi}^T \log \mathbf{p} + c \|\theta\|^2$$

Ecuación 2. Función de pérdida de AlphaZero

donde c es un parámetro que controla el nivel de la regularización de los pesos de las capas de la red neuronal. Los parámetros actualizados son usados en las subsiguientes partidas contra sí mismo.

6.3 MCTS y poda alfa-beta

Durante las últimas cuatro décadas los motores de ajedrez más fuertes han usado la poda alfa-beta como algoritmo de cálculo. AlphaZero usa un enfoque notablemente diferente, que se guía por estadísticas obtenidas de la evaluación de diferentes posiciones dentro cada rama del árbol de variantes, en vez de calcular la evaluación minimax de cada una de esas ramas. No obstante, los programas de ajedrez tradicionales que usaban el MCTS eran mucho más débiles que los que usaban la poda alfa-beta, mientras que los programas que basaban la poda alfa-beta en redes neuronales han sido incapaces de competir con funciones de evaluación rápidas y relativamente precisas.

AlphaZero evalúa posiciones usando una aproximación no lineal basada en una red neuronal, en vez de la aproximación lineal usada en los programas de ajedrez convencionales. Esto proporciona una estimación mucho más precisa de la evaluación real, si bien puede introducir errores de aproximación. Como el MCTS se basa en estadísticas, estos errores de aproximación tienden a ser cancelados al evaluar una rama larga del árbol. Por el contrario, la poda alfa-beta calcula un minimax explícito, que tiende a propagar los errores de aproximación más grandes hasta la raíz del árbol de variantes. Usar el MCTS posibilita que AlphaZero combine de forma efectiva su red neuronal con un algoritmo de cálculo potente.

7 Descripción de la solución propuesta

Como se ha visto en los apartados anteriores, existen una infinidad de ideas relacionadas con la programación de un motor de ajedrez, todas ellas muy interesantes y capaces de lograr mejoras significativas en el juego de la máquina.

No obstante, merece la pena recordar que el presente proyecto no está enfocado en conseguir un motor de ajedrez que compita por el título mundial, sino en desarrollar un programa que facilite el uso del ajedrez como herramienta pedagógica. Es por ello que hay que tener muy presentes los requisitos planteados en el apartado 2.3:

- El nivel de juego del motor no debe ser excesivamente alto. Si bien puede ser interesante que los niños se vean vencidos en las primeras partidas, estos no deben sentirse sobrepasados en ningún momento. De lo contrario existe el riesgo de que se desmotiven y abandonen la práctica del ajedrez.
- La máquina debe ser capaz de jugar relativamente rápido, pues un niño tiende a aburrirse rápidamente ante la falta de estímulos en cualquier actividad.
- Se pretende llegar al máximo número de niños posible, por lo que el programa debe ser fácilmente ejecutable desde cualquier ordenador, independientemente de lo moderno que sea o del sistema operativo que utilice.

El tercer requisito obliga al proyecto a decantarse por el primer enfoque, ya que ejecutar un software tan puntero como el de AlphaZero, es necesario tener un hardware igualmente puntero. A día de hoy muchos de los componentes difícilmente se podrían encontrar en el mercado, y aunque se encontraran, el precio de su compra sería inaccesible para muchas familias, por lo que no se conseguiría llegar al máximo número de niños posible.

Por otro lado, el motor se programará en Python, pues debido a la ingente cantidad de información disponible para este lenguaje de programación, la tarea se simplificará notoriamente.

Así pues, se tiene claro que se va a programar un motor de ajedrez convencional, y como se ha explicado en el apartado correspondiente, deberá estar formado por tres módulos: el generador de movimientos, la función de evaluación y la función de búsqueda.

7.1 Generador de movimientos

Se hará uso de la librería python-chess, descrita en un apartado posterior. Esta librería también servirá de soporte para programar una interfaz adecuada en la que se desarrolle el juego y una planilla en la que queden anotadas las jugadas para poder conservar las partidas.

7.2 Función de evaluación

Para diseñar la función de evaluación, habrá que atender al primer requisito. Se pretende que el motor de ajedrez juegue al nivel necesario para que los niños que acaban de aprender a mover las piezas den su primer salto de calidad, que consistirá en interiorizar una de las lecciones más importantes del ajedrez: el valor de las piezas

es relativo. Esta idea nos servirá para dar forma a la función de evaluación, como se detalla más adelante.

La idea es que el programa propuesto analice todas las posiciones, basándose únicamente en el valor “estándar” de las piezas. De esta forma, para conseguir derrotar a este programa, el retador tiene que haber asimilado la lección anterior y ser capaz de aplicarla correctamente. A la hora de determinar un valor para cada pieza se seguirá el criterio que Roberto Grau utiliza en su libro “Tratado General de Ajedrez”:

- Peón = 1
- Caballo = 3
- Alfil = 3
- Torre = 5
- Dama = 9

Al rey no hace falta asignarle ningún valor, ya que siempre que una partida esté en juego los dos bandos van a contar con él.

Por supuesto, la función de evaluación también debe ser capaz de detectar el jaque mate cuando este se presente.

7.3 Función de búsqueda

Para diseñar la función de búsqueda, habrá que atender al segundo requisito. El programa debe responder con su jugada prácticamente de forma instantánea. Así, se conseguirá que los niños se encuentren constantemente involucrados en la partida y no desconecten del juego.

Esta función de búsqueda se programará en base a la poda Alpha-beta, pues, aunque el algoritmo minimax sea más intuitivo y fácil de programar, la eficiencia computacional de la poda Alpha-beta compensa con creces el pequeño esfuerzo adicional necesario. La profundidad a la que se analizarán las posiciones se establece experimentalmente en 3 movimientos. A esa profundidad se consigue que el motor evite cometer disparates como sacrificar la dama para ganar un peón y que además juegue de forma fluida.

8 Descripción de tareas. Planificación. Diagrama de Gantt

A continuación, se muestra la planificación que se ha seguido para desarrollar este proyecto mediante un diagrama de Gantt. El proyecto comienza el 16/10/2018 con la primera reunión con el tutor y finaliza el 22/07/2019, fecha en la que estaba prevista la entrega de la memoria del TFG

Para la planificación del proyecto se tiene en cuenta que solo se va a llevar a cabo los días laborables y también se consideran cinco periodos de tiempo en los que no se va a avanzar con el proyecto, que son:

- Navidad: del 25/12/2018 al 06/01/2019
- Exámenes del primer cuatrimestre: del 07/01/2019 al 25/01/2019
- Semana Santa: del 14/04/2019 al 20/04/2019
- Exámenes del segundo cuatrimestre: del 20/05/2019 al 07/06/2019
- Exámenes de recuperación: del 24/06/2019 al 05/07/2019

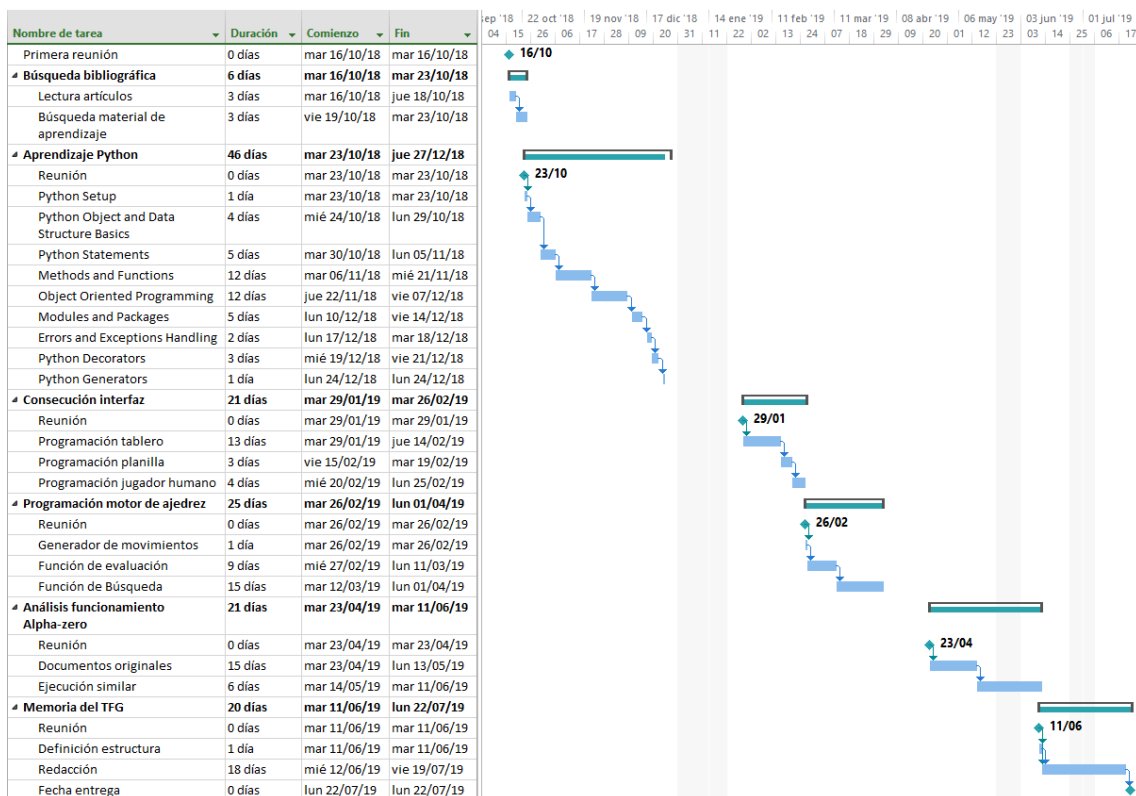


Ilustración 26. Diagrama de Gantt

Primeramente, se realiza una reunión, el 16/10/2018, con el director en la que se le da forma a la idea del alumno y se piensa cómo se podría llevar a cabo. Al ser un proyecto novedoso y poco común en la escuela de ingenieros de Bilbao se decide dejar unos días de margen para realizar una búsqueda bibliográfica y verificar que el proyecto puede ser llevado a cabo. En el caso de que así fuera, se especula que el proyecto se desarrollaría en Python, principalmente por ser este uno de los lenguajes de programación que más popularidad han cobrado en los últimos años y, por ende, disponer de mucha información y facilidades al respecto. En esta reunión también se define cual sería el equipo de trabajo, formado por Eloy Irigoyen Gordo y Mikel Larrea Sukia como director y codirector respectivamente y Daniel Ochoa de Eribe Martínez como alumno.

Así pues, la primera tarea consiste en realizar dicha búsqueda bibliográfica, y tiene lugar entre el 16/10/2018 y el 23/10/2018. Este periodo de tiempo sirve para leer artículos relacionados con el tema y buscar libros y cursos audiovisuales que permitan al alumno aprender a programar con Python.

Tras esta primera tarea, tiene lugar una segunda reunión, el 23/10/2018, en la que se juntan los tres miembros que forman el equipo y tiene lugar un *brainstorming*, en el que se exponen las ideas principales y conclusiones a las que los tres miembros del equipo han llegado por separado. Se decide que el proyecto puede ser llevado a cabo y que, efectivamente, será llevado a cabo en Python.

Esto da paso a la segunda tarea, que consiste en aprender a programar en Python. Esta tarea tiene lugar entre el 23/10/2018 y el 24/12/2018. Se le asignan dos meses a esta tarea porque el alumno parte desde cero en Python, si bien es cierto que éste poseía conocimientos previos de programación, en Pascal y Matlab, adquiridos durante cursos anteriores de la carrera. Para llevar a cabo esta tarea se recurre a un curso audiovisual creado por José Portilla, denominado *Go from Zero to Hero in Python 3*, y disponible en la plataforma de aprendizaje en línea Udemy. Las lecciones principales que componen este curso son:

- Python Setup
- Python Object and Data Structure Basics
- Python Statements
- Methods and Functions
- Object Oriented Programming
- Modules and Packages
- Errors and Exceptions Handling
- Python Decorators
- Python Generators

Tras finalizar la segunda tarea, y a la vuelta de las vacaciones de navidad y el periodo de inactividad debido a los exámenes del primer cuatrimestre, se realiza nuevamente una reunión, el 29/01/2019, en la que el alumno expone los conocimientos adquiridos durante los dos últimos meses y plantea una posible solución al diseño del motor de ajedrez, tomando Python-chess como punto de partida. El resto de miembros del equipo le dan su visto bueno, y procede a planificar lo que resta del proyecto.

La tercera tarea consiste en conseguir una interfaz adecuada para que se desarrolle el juego, y tiene lugar entre el 29/01/2019 y el 26/02/2019. Esta tarea, abarca no solo la programación del tablero de ajedrez y la ejecución de la partida como tal, sino también la programación de una planilla en la que queden escritas las jugadas para poder analizar las partidas a posteriori y la programación de un jugador humano.

La cuarta tarea consiste por fin en programar el motor de ajedrez como tal, y tiene lugar entre el 26/02/2019 y el 23/04/2019, conteniendo el periodo de inactividad debido a Semana Santa. Dado que el generador de movimientos es inmediato con python-chess, el primer paso de esta tarea consiste en conseguir la función de evaluación deseada. A continuación, se centrarán los esfuerzos en la función de búsqueda: por simplicidad, primero se tratará de implementar el algoritmo minimax y a continuación, tomándolo como referencia, la poda Alpha-beta.

La quinta tarea consiste en analizar el funcionamiento de AlphaZero, y tiene lugar entre el 23/04/2019 y el 11/06/2019, conteniendo el periodo de inactividad debido a los exámenes del segundo cuatrimestre. Básicamente la tarea consistirá en intentar

entender las ideas de los documentos originales de AlphaZero, que están disponibles en Github, y en intentar ejecutar algo similar.

La sexta y última tarea consiste en redactar la memoria y tiene lugar entre el 11/06/2019 y el 22/07/2019, conteniendo el periodo de inactividad debido a los exámenes de recuperación. Primero se define la estructura de la memoria, y después se redacta.

9 Medios y técnicas básicas

9.1 Ordenador

Se ha contado con dos ordenadores, uno personal y otro proporcionado por el departamento de Ingeniería de Control y Automática.

9.2 GeForce GTX 1050

Tarjeta gráfica que cuenta con una unidad de procesamiento gráfico (GPU) desarrollada por la empresa estadounidense NVIDIA.

9.3 Udemy

Plataforma de aprendizaje en línea, de donde se ha obtenido el curso audiovisual “Go from Zero to Hero in Python 3”, creado por José Portilla.

9.4 Python 3.7

Lenguaje de programación de alto nivel cuya filosofía hace hincapié en una sintaxis que favorezca el código legible. A día de hoy es uno de los lenguajes de programación más populares y existe una enorme cantidad de librerías e información al respecto circulando por internet. Otras de sus ventajas son que puede ser desarrollado en diversas plataformas como Unix, GNU/Linux, macOS, Windows... y que es gratuito, incluso para propósitos empresariales.

9.5 Python-chess

Librería de Python orientada al ajedrez, con generación y validación de movimientos y soporte para formatos comunes.

Esta librería ha servido como base del proyecto, y ha permitido centrar los esfuerzos en los módulos más interesantes del motor de ajedrez: la función de búsqueda y la función de evaluación. A continuación, se presenta un ejemplo de implementación del mate del pastor en python-chess:

```
>>> import chess

>>> board = chess.Board()

>>> board.legal_moves
<LegalMoveGenerator at ... (Nh3, Nf3, Nc3, Na3, h3, g3, f3, e3, d3, c3, ...)>
>>> chess.Move.from_uci("a8a1") in board.legal_moves
False

>>> board.push_san("e4")
Move.from_uci('e2e4')
>>> board.push_san("e5")
Move.from_uci('e7e5')
>>> board.push_san("Qh5")
Move.from_uci('d1h5')
>>> board.push_san("Nc6")
Move.from_uci('b8c6')
>>> board.push_san("Bc4")
Move.from_uci('f1c4')
>>> board.push_san("Nf6")
Move.from_uci('g8f6')
>>> board.push_san("Qxf7")
Move.from_uci('h5f7')

>>> board.is_checkmate()
True

>>> board
Board('r1bqkb1r/pppp1Qpp/2n2n2/4p3/2B1P3/8/PPPP1PPP/RNB1K1NR b KQkq - 0 4')
```

Ilustración 27. Implementación del mate del pastor en Python-chess



Ilustración 28. Posición resultante tras el mate de pastor

9.6 Anaconda Navigator

Distribución libre y abierta de los lenguajes de programación Python y R para ciencia computacional.

Dentro de Anaconda Navigator, los Jupyter Notebooks han resultado ser de gran utilidad. En ellos se puede ejecutar código en Python de una manera fácil e intuitiva, debido a la forma en la que están distribuidos por medio de celdas en la que se puede ver tanto la entrada como la salida del código.

```
In [17]: mylist = [(1,2),(3,4),(5,6),(7,8)]
```

```
In [18]: len(mylist)
```

```
Out[18]: 4
```

```
In [19]: for item in mylist:
          print(item)
```

```
(1, 2)
(3, 4)
(5, 6)
(7, 8)
```

Ilustración 29. Implementación código en Jupyter Notebooks

9.7 Github

Plataforma de desarrollo colaborativo para alojar proyectos utilizando el sistema de control de versiones Git. Se utiliza principalmente para la creación de código fuente de programas de computadora.

Por medio de Github se ha podido trabajar conjuntamente con el ordenador personal y el del laboratorio. También ha servido para que los directores del proyecto pudieran hacer un seguimiento del mismo en todo momento. Además, numerosos repositorios de la plataforma han servido de inspiración y punto de partida para la programación del motor de ajedrez.

9.8 Gitkraken

Cliente de GIT GUI para Windows, Mac y Linux. Ayuda a los desarrolladores a trabajar de una forma más productiva y eficiente con Git. Es libre para usos no comerciales.

Durante el desarrollo del proyecto ha servido para organizar el código e ir añadiendo mejoras al programa de manera ordenada.



Ilustración 30. Esquema resultante en Gitkraken tras el trabajo

10 Presupuesto

A continuación, se muestra el presupuesto necesario para la ejecución del proyecto. Para calcularlo se han tenido en cuenta los costes asociados a la mano de obra, imputación proporcional de costes amortizables, gastos adicionales y costes indirectos.

Concepto	Unidades	Nº de unidades	Coste unitario	Coste total
Horas internas				
Ingeniería (estudiante)	h	150	20,00 €	3.000,00 €
Ingeniería (director)	h	25	25,00 €	625,00 €
Ingeniería (codirector)	h	25	25,00 €	625,00 €
Amortizaciones				
Ordenador	h	135	0,06 €	8,10 €
Tarjeta gráfica	h	10	0,02 €	0,20 €
Licencias				
Python	h	70	- €	- €
Gitkraken (no comercial)	h	10	- €	- €
Microsoft Office	h	35	0,02 €	0,70 €
Microsoft Project	h	5	0,05 €	0,25 €
Gastos				
Curso Udemy				11,00 €
Material de oficina				60,00 €
Costes directos				4.330,25 €
Costes indirectos	7%			303,12 €
Total				4.633,37 €

Ilustración 31. Presupuesto del proyecto

En primer lugar, se han considerado las horas de ingeniería correspondientes al estudiante, director y codirector, ya que suponen un gran porcentaje del gasto total (91,73%).

Por otra parte, para las amortizaciones se han tendido en cuenta el coste de adquisición de los dispositivos y el coste de las licencias de los softwares informáticos:

- Ordenador (2 unidades): 674 €
- Tarjeta gráfica: 150 €
- Licencia para Python: gratuita (código abierto)
- Licencia para Gitkraken: gratuita para uso no comercial
- Licencia para Microsoft Office (Word, Excel y Power Point): 149 €
- Licencia para Microsoft Project: 417 €

Para calcular la tasa horaria de los ordenadores y la tarjeta gráfica, se ha supuesto que tienen una vida útil de 6 años. En cuanto a los softwares informáticos, se ha supuesto que las licencias caducan un año después de ser adquiridas.

Como gastos adicionales se ha tenido en cuenta el material de oficina (pen drives, cuadernos...) y el curso audiovisual de José Portilla: "Go from zero to hero in Python 3". Este curso está oficialmente disponible en la plataforma de aprendizaje en línea Udemy por 149 €. No obstante, el curso se adquirió en un periodo de oferta por 11 €.

Para finalizar, se han añadido los costes indirectos, que representan el 7% de los costes directos. Se trata de aquellos costes que no están directamente relacionados con la ejecución del proyecto, pero es necesario incluirlos en el presupuesto (limpieza del aula, luz, aire acondicionado...).

11 Riesgos del proyecto

Históricamente, la programación solía ser una actividad solitaria que requería de una alta concentración e incluso aislamiento total. Los mejores programadores saben como alcanzar un estado mental conocido como *Fluir* o *Zona*, en el cual la mente es capaz de enfocarse en el código y tomar decisiones sumamente creativas y eficientes.

Aunque *Fluir* es algo muy valioso y puede resultar más difícil de lograr en grupo, la realidad muestra que la programación solitaria es demasiado riesgosa y a largo plazo termina costando caro. Puesto que este es en gran medida un proyecto que se llevará a cabo mediante programación solitaria, han de tenerse en cuenta los riesgos más comunes de esta práctica.

A cada uno de los riesgos se le asignará un valor referente a la probabilidad de que ocurra y otro referente al impacto o a la gravedad de las consecuencias que acarrearía. Estos valores irán del 1 (muy bajo) al 5 (muy alto) y mediante ellos se procederá ubicar cada riesgo en la siguiente matriz:





			GRAVEDAD (IMPACTO)				
			MUY BAJO	BAJO	MEDIO	ALTO	MUY ALTO
			1	2	3	4	5
PROBABILIDAD	MUY ALTA	5	5	10	15	20	25
	ALTA	4	4	8	12	16	20
	MEDIA	3	3	6	9	12	15
	BAJA	2	2	4	6	8	12
	MUY BAJA	1	1	2	3	4	5
	Riesgo muy grave. Requiere medidas preventivas urgentes. No se debe iniciar el proyecto sin la aplicación de medidas preventivas urgentes y sin acotar sólidamente el riesgo.						
	Riesgo importante. Medidas preventivas obligatorias. Se deben controlar fuertemente las variables de riesgo durante el proyecto.						
	Riesgo apreciable. Estudiar si es posible introducir medidas preventivas para reducir el nivel de riesgo. Si no fuera posible, mantener las variables controladas.						
	Riesgo marginal. Se vigilará aunque no requiere medidas preventivas de partida.						

Tabla 3. Matriz de riesgos

Posteriormente, y dependiendo del valor obtenido por cada riesgo en la matriz, se establecerán medidas de control o acciones adecuadas para contrarrestarlos.

11.1 Clasificación de los riesgos

11.1.1 Alta tasa de defectos

Éste es el riesgo más obvio. Los seres humanos no somos magos y sin importar que tan preciso se intente ser, es inevitable que ocurran errores de tipeo, se comprenda mal el requerimiento o simplemente ocurra una equivocación.

Por otro lado, las consecuencias de este riesgo podrían llegar a ser desastrosas, ya que se podrían ir arrastrando errores paulatinamente hasta que el código sea tan defectuoso que no merezca la pena seguir con él. Además, muchos pequeños errores podrían pasar por alto fácilmente y su posterior detección requeriría un gasto excesivo de tiempo y esfuerzo.

Se le asigna a este riesgo una probabilidad de 5 (muy alta) y un impacto de 5 (muy alto).

11.1.2 Distracciones

Si ya es difícil de por sí entrar en la *Zona*, existe la dificultad añadida de que una vez dentro, el programador ha de mantenerse ahí. Este riesgo engloba todas aquellas fuentes de distracción con las que el programador podría perder la concentración, ya sean éstas humanas (algún compañero solicitando atención) o artificiales (anuncios, vídeos, pérdida del hilo en la búsqueda de información...).

La probabilidad de que se den distracciones es absoluta, sobre todo aquellas clasificadas como artificiales. Principalmente porque el proyecto requiere de un gran trabajo de búsqueda de información. La mente humana es experta en divagar y es probable que se acabe leyendo algún artículo interesante pero irrelevante en ese momento.

No obstante, las consecuencias de este riesgo no serían demasiado perjudiciales, después de todo se da por hecho que nadie puede escribir código creativo durante ocho horas seguidas. Simplemente se perdería tanto tiempo como durara la distracción y la recuperación de la concentración.

Se le asigna a este riesgo una probabilidad de 5 (muy alto) y un impacto de 1 (muy bajo).

11.1.3 Aprendizaje excesivamente lento y costoso

Como se ha comentado en apartados anteriores, el alumno parte desde cero en cuanto a conocimientos de Python y prácticamente también desde cero en cuanto a experiencia programando. Es por eso que puede darse el caso de que en algún momento se vea abrumado por el aprendizaje y le cueste demasiado esfuerzo seguir adelante con el proyecto, o compatibilizar el mismo con el último año del curso académico.

No obstante, el alumno ha adquirido una sólida base científica y tecnológica durante la carrera y está muy familiarizado con el ajedrez, por lo que se considera que la probabilidad de que vea abrumado por el proyecto es baja.

Por otro lado, las consecuencias de este riesgo serían desastrosas, pues se tendría que dejar de lado el proyecto y darlo por nulo.

Se le asigna a este riesgo una probabilidad de 2 (baja) y un impacto de 5 (muy alto).

11.1.4 Pocos incentivos para seguir prácticas comunes

Cuando se trabaja solo, se tiende a hacer las cosas a la manera propia, de forma que resulte más rápido y sencillo mentalmente. Sin embargo, como en el resto de disciplinas, en el mundo de la programación existe una forma de hacer las cosas y ciertas herramientas que simplifican no solo el trabajo propio, sino también el entendimiento del proyecto a futuras personas interesadas en tu código.

Al ser un programador inexperimentado, la probabilidad de caer en este riesgo es alta, pues no solo existe la tentación de hacer las cosas como uno quiere, sino que también se debe tener en cuenta que muchas de estas prácticas se desconocerán.

En cuanto a las consecuencias de este riesgo, no se consideran demasiado perjudiciales, pues el proyecto podría salir adelante aunque no fuera llevado a cabo de la mejor manera posible.

Se le asigna a este riesgo una probabilidad de 4 (alta) y un impacto de 2 (bajo).

11.1.5 Pérdida de información

Por último, se debe valorar la posibilidad de que por algún problema técnico o fallo humano se pierda parte de la información o alguna mejora no se guarde correctamente.

Dada la tecnología existente hoy en día la probabilidad de que exista algún fallo técnico es prácticamente despreciable, pero no es tan difícil que por algún descuido el trabajo de las últimas horas haya sido en vano o parte del código válido sea sobrescrito.

No obstante, las consecuencias no serían demasiado graves, puesto que al ser una idea que se ha desarrollado durante las últimas horas, lo normal es tenerla todavía en mente y que cueste bastante menos tiempo volverla a desarrollar.

Se le asigna a este riesgo una probabilidad de 2 (baja) y un impacto de 1 (muy bajo).

Así pues, estos serían los resultados obtenidos por cada riesgo en la matriz:

RIESGO	Probabilidad (Ocurrencia)	Gravedad (Impacto)	Valor del Riesgo	Nivel de Riesgo
Alta tasa de defectos	5	5	25	Muy grave
Distracciones	5	1	5	Apreciable
Aprendizaje excesivamente lento y costoso	2	5	10	Importante
Pocos incentivos para seguir prácticas comunes	4	2	8	Apreciable
Pérdida de información	2	1	2	Marginal

Tabla 4. Enumeración riesgos del proyecto

Con todo, existe un riesgo que merece suma atención, otros tres que habrá que vigilar de cerca, y un último riesgo que no merece demasiado atención. Sabiendo esto se puede proceder a establecer las medidas de control para contrarrestarlos.

11.2 Medidas de control

11.2.1 Alta tasa de defectos

Para enfrentar este riesgo se llevará a cabo una planificación cuidadosa del programa que se desea conseguir, estructurando el código adecuadamente y marcando pequeños objetivos para verificar que las últimas modificaciones funcionan correctamente. De aquí surge la idea de programar jugadores, pasando al siguiente jugador una vez se ha comprobado que las modificaciones efectuadas funcionan correctamente. Además, se revisará continuamente el código durante el mismo acto de escribirlo.

11.2.2 Distracciones

Para enfrentar este riesgo se fijará un horario de trabajo que se procurará seguir en la medida de lo posible. Antes de cada sesión de trabajo se decidirá que se pretende conseguir durante la misma, siendo en todo momento consciente de que un exceso de distracciones acarrearía la no consecución de estos objetivos. Además, es muy interesante disponer de un espacio de trabajo aislado en el que sentirse identificado con el proyecto. Es por esto, que los directores del proyecto se encargarán de proporcionar un espacio de trabajo en el Departamento de Ingeniería de Sistemas y Automática de la Escuela de Ingeniería de Bilbao.

11.2.3 Aprendizaje excesivamente lento y costoso

Para enfrentar este riesgo, se procurará ser muy selectivo con los medios de aprendizaje, y se centrarán los esfuerzos en aprender únicamente lo estrictamente necesario para afrontar el proyecto.

11.2.4 Pocos incentivos para seguir prácticas comunes

Para enfrentar este riesgo, se seguirán los consejos de los directores del proyecto, pues estos poseen la experiencia programando de la que carece el alumno.

11.2.5 Pérdida de información

Para enfrentar este riesgo, simplemente se verificará que parte las últimas modificaciones han sido guardadas antes de cerrar todos los programas asociados al trabajo.

12 Conclusiones

Las conclusiones a las que se han llegado tras la realización de este trabajo se pueden clasificar en 3 grupos:

- En lo que al concepto de “inteligencia artificial” se refiere:

Antes que nada, conviene recordar el test de Turing, que básicamente decía lo siguiente: “El día que se pueda hablar de inteligencia artificial será aquel en el que uno esté hablando con un ente al que no puede ver, y no pueda diferenciar si se trata de una máquina o una persona”.

Los motores de ajedrez convencionales como Stockfish todavía se delatan, porque muestran un juego poco intuitivo y muy concreto. Esto se debe a que los motores de ajedrez convencionales no piensan, sino que simulan procesos pensantes apoyándose en lo que mejor sabe hacer un ordenador: calcular. Por ejemplo, cuando Stockfish tiene una partida ganada, siempre elige el camino más rápido para ganar la partida, aunque este camino esté lleno de “riesgos”. A Stockfish no le preocupan los “riesgos” porque lo tiene todo calculado.

Un jugador humano, por el contrario, se caracteriza por un juego intuitivo y mucho más sencillo. En caso de tener una partida ganada, el jugador humano haría la jugada que lleva a una victoria más fácil, aunque tarde unas cuantas jugadas más en dar el jaque mate. Como era de esperar, el estilo de juego de AlphaZero está más próximo al de un humano, por lo que tiene más papeletas para superar el test de Turing.

De todas formas, es cierto que a día de hoy existen otras interpretaciones más extendidas acerca de lo que significa “inteligencia artificial”. Max Tegmark, en su libro “Vida 3.0” define la inteligencia como “capacidad para lograr objetivos complejos”. Desde este punto de vista Stockfish puede ser considerado más inteligente que cualquier humano en lo que a jugar a ajedrez se refiere.

- En cuanto a la utilidad de un motor de ajedrez como herramienta pedagógica:

La ventaja de hacer un programa convencional es que se puede elegir el estilo de juego del motor, como es el caso de este trabajo. De esta forma, se pueden potenciar cierto tipo de pensamientos en la mente del niño. No obstante, el hecho de que el ordenador tenga un estilo de juego determinado puede desembocar en partidas monótonas y aburridas.

La ventaja de hacer un programa en la línea de AlphaZero es que las partidas se aproximarían mucho más a la realidad. El rival recuperaría ese estilo creativo y original característico de los humanos, por lo que las partidas resultarían más divertidas y entretenidas, incluso para un niño.

- En cuanto al conocimiento ajedrecístico aportado por los programas de ajedrez:

Los programas convencionales han jugado un papel muy importante en las últimas décadas, ya que muchos de los análisis de los grandes maestros se han beneficiado de su capacidad de cálculo. No obstante, en cuanto a conocimiento ajedrecístico puro, no

se puede decir que los programas convencionales hayan aportado algo, ya que a fin de cuentas todo lo que saben de ajedrez es lo que les hemos “enseñado” los humanos.

Pero AlphaZero ha llegado a un nivel sobrehumano de forma independiente a nuestro conocimiento, por lo que habrá adquirido sus propias estrategias y su propio entendimiento del juego. Evidentemente, coincidiremos con él en muchas ideas, pero seguro que también tiene otras muchas que enseñarnos.

Sin embargo, lo que está claro es que, con el hardware disponible a día de hoy, tanto niños como ajedrecistas solo pueden soñar con poseer su propia inteligencia artificial. Recordemos que el equipo de DeepMind diseñó un hardware exclusivo para AlphaZero, del cual muchos componentes ni siquiera se encuentran en el mercado, y aunque se comercialicen en un futuro próximo, el precio sería desorbitado.

13 Fuentes de información

- [1] Portilla, J: "Complete Python Bootcamp: Go from zero to hero in Python 3". Udemy. Disponible en <https://www.udemy.com/complete-python-bootcamp/>
- [2] Fiekas, N. (25/06/2019): "python-chess 0.28.1". Disponible en: <https://pypi.org/project/python-chess/>
- [3] "Deep Blue, la máquina que derrotó a Gary Kasparov". Diariocrítico. Disponible en <https://www.diariocritico.com/deep-blue-la-maquina-que-derroto-a-gary-kasparov>
- [4] García, L: "¿Para qué sirve el ajedrez en educación?". BBVA: "Aprendemos Juntos". Disponible en <https://www.youtube.com/watch?v=IPq0RwegByg>
- [5] Punset, E: "El ajedrez". RTVE: "Redes". Disponible en <https://www.youtube.com/watch?v=z-40gzv1xms>
- [6] Brains Nursery Schools: "10 Beneficios del ajedrez en los niños". Disponible en <https://brainsnursery.com/10-razones-ajedrez-bueno-para-ninos/>
- [7] Cuenca, P. (27/03/2017) : "¿Cómo piensan y operan los módulos de análisis?". Chess24. Disponible en <https://chess24.com/es/informate/noticias/como-piengan-y-operan-los-modulos-de-analisis>
- [8] Blank, D. (2016): "Programming a chess player". Department of Computer Science, Bryn Mawr College, Pensilvania (EEUU). Disponible en: <https://jupyter.brynmawr.edu/services/public/dblank/CS371%20Cognitive%20Science/2016-Fall/Programming%20a%20Chess%20Player.ipynb>
- [9] Harris, N. (26/12/2012): "How my chess engine works". Disponible en: <https://www.naftaliharris.com/blog/chess/>
- [10] Sánchez, L. y Edgardo, O. (2010): "Desarrollo de un motor de ajedrez. Algoritmos y Heurísticas para la reducción del espacio de búsqueda". Universidad de Sonora, Hermosillo (México).
- [11] Costalba, M., Kiiski J., Linscott G., Romstad, T.: "Stockfish Chess". Disponible en <https://stockfishchess.org/>
- [12] Costalba, M., Kiiski J., Linscott G., Romstad, T.: "Stockfish Open Source". Disponible en <https://github.com/mcostalba/Stockfish>
- [13] Ray, C. (2014): "How Stockfish Works: An Evaluation of the Databases Behind the Top Open-Source Chess Engine". Good fibrations. Disponible en <http://rin.io/chess-engine/>
- [14] Neumann, J. von (1928). "Zur Theorie der Gesellschaftsspiele". Mathematische Annalen.
- [15] Beal, D. F. (1999): "The Nature of Minimax Search". Universidad de Maastricht. Disponible en https://project.dke.maastrichtuniversity.nl/games/files/phd/Beal_thesis.pdf

- [16] “Mini-Max Algorithm in Artificial Intelligence”. javaTpoint. Disponible en <https://www.javatpoint.com/mini-max-algorithm-in-ai>
- [17] “Minimax” Chess Programming WIKI. Disponible en <https://www.chessprogramming.org/Minimax>
- [18] “Alpha-Beta Pruning”. javaTpoint. Disponible en <https://www.javatpoint.com/ai-alpha-beta-pruning>
- [19] “Alpha-Beta”. Chess Programming WIKI. Disponible en <https://www.chessprogramming.org/Alpha-Beta>
- [20] Frayn, C. (2005): “Computer chess programming theory”. Disponible en <http://www.frayn.net/beowulf/theory.html#negamax>
- [21] Eppstein D. (01/02/1999): “Which nodes to search? Full-width vs. selective search”. ICS 180, Winter 1999: Strategy and board game programming. Dept. Information & Computer Science, UC Irvine. Disponible en <https://www.ics.uci.edu/~eppstein/180a/990204.html>
- [22] “Move Ordering”. Chess Programming WIKI. Disponible en https://www.chessprogramming.org/Move_Ordering
- [23] “Horizon Effect”. Chess Programming WIKI. Disponible en https://www.chessprogramming.org/Horizon_Effect
- [24] “Quiescence Search”. Chess Programming WIKI. Disponible en https://www.chessprogramming.org/Quiescence_Search
- [25] “Killer Move”. Chess Programming WIKI. Disponible en https://www.chessprogramming.org/Killer_Move
- [26] “Null Move Pruning”. Chess Programming WIKI. Disponible en https://www.chessprogramming.org/Null_Move_Pruning
- [27] “Nalinov Tablebases”. Chess Programming WIKI. Disponible en https://www.chessprogramming.org/Nalimov_Tablebases
- [28] Nair, S. (28/05/2019): “Alpha Zero General (any game, any framework!)”. Disponible en <https://github.com/suragnair/alpha-zero-general>
- [29] Nair, S. (29/12/2017): “A Simple Alpha(Go) Zero Tutorial”. Disponible en <http://web.stanford.edu/~surag/posts/alphazero.html>
- [30] Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., Hubert, T., Baker, L., Lai, M., Bolton, A., Chen, Y., Lillicrap, T., Hui, F., Sifre, L., van den Driessche, G., Graepel, T. y Hassabis, D. (19/10/2017): “Mastering the game of Go without human knowledge”. Disponible en: https://www.nature.com/articles/nature24270.epdf?author_access_token=VJXbVjaSHxFoctQQ4p2k4tRgN0jAjWel9jnR3ZoTv0PVW4gB86EEpGqTRDtpIz-2rmo8-KG06gqVobU5NSCFeHILHcVFUeMsbvws-lxjqQGg98faovwixeTUgZAUMnRQ

[31] Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., Hubert, T., Baker, L., Lai, M., Bolton, A., Chen, Y., Lillicrap, T., Hui, F., Sifre, L., van den Driessche, G., Graepel, T. y Hassabis, D. (05/10/2017): "Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm". Disponible en: <https://arxiv.org/abs/1712.01815>

[32] Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T., Lillicrap, T., Simonyan, K., Hassabis, D. (07/12/2018) : "A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play". Science. Disponible en <https://science.sciencemag.org/content/362/6419/1140.full?ijkey=XGd77kl6W4rSc&keytype=ref&siteid=sci>

[33] Silver, D., Hubert, T., Schrittwieser, J., Hassabis, D.: "AlphaZero: Shedding new light on chess, shogi, and Go". DeepMind. Disponible en <https://deepmind.com/blog/article/alphazero-shedding-new-light-grand-games-chess-shogi-and-go>

[34] Narváez, A: "Desvelamos los secretos del juego de AlphaZero: así aprendió a ganar a Stockfish". Thezugzwangblog. Disponible en <https://thezugzwangblog.com/desvelamos-los-secretos-del-juego-alphazero/>

[35] Levine, J: "Monte Carlo Tree Search". University of Strathclyde. Disponible en: <https://www.youtube.com/watch?v=UXW2yZndI7U>

[36] Grau, R: "Tratado General de Ajedrez". «Tomo I, Rudimentos». Editorial Sopena de Argentina.

[37] Tegmark, M. (2017): "Life 3.0". Penguin Books.

[38] González de la Torre, S. (23/01/2015): ¿Cómo utilizar un motor de análisis? (1). NowInChess. Disponible en <http://nowinchess.com/2015/01/23/como-utilizar-un-motor-de-analisis-1/>

14 Anexo I: Código

Se presenta a continuación el código desarrollado en Python durante la realización del proyecto. Se explican los aspectos más importantes del código, partiendo de que ya se conoce toda la información presentada en la documentación oficial de python-chess, cuyo link se encuentra en segundo lugar en la bibliografía.

En primer lugar, se importan los módulos y paquetes necesarios de la librería Python-chess, y se inicializa una variable “board” de clase “Board”, que servirá para representar el tablero de ajedrez.

```
import chess
```

```
board = chess.Board()
```

```
import time  
from IPython.display import display, HTML, clear_output
```

A continuación, se programan una serie de funciones, cuyo objetivo es hacer posible la ejecución de una partida de ajedrez entre dos jugadores.

```
def who(player):  
    return "White" if player == chess.WHITE else "Black"
```

```
def display_board(board, use_svg):  
    if use_svg:  
        return board._repr_svg_()  
    else:  
        return "<pre>" + str(board) + "</pre>"
```

```

def play_game(player1, player2, visual="svg", pause=0.1):
    """
    playerN1, player2: functions that takes board, return uci move
    visual: "simple" | "svg" | None
    """
    use_svg = (visual == "svg")
    board = chess.Board()
    try:
        planilla = []
        numero_jugada = 1.0
        while not board.is_game_over(claim_draw=True):
            if board.turn == chess.WHITE:
                uci = player1(board)
            else:
                uci = player2(board)
            name = who(board.turn)
            board.push_uci(uci)
            board_stop = display_board(board, use_svg)
            html = "<b>Move %s %s, Play '%s':</b><br/>%s" % (
                len(board.move_stack), name, uci, board_stop)
            if visual is not None:
                if visual == "svg":
                    clear_output(wait=True)
                    display(HTML(html))
                if visual == "svg":
                    time.sleep(pause)
            planilla.append(str(numero_jugada) + '. ' + uci)
            print(planilla)
            numero_jugada = numero_jugada + 0.5
        except KeyboardInterrupt:
            msg = "Game interrupted!"
            return (None, msg, board)
        result = None
        if board.is_checkmate():
            msg = "checkmate: " + who(not board.turn) + " wins!"
            result = not board.turn
        elif board.is_stalemate():
            msg = "draw: stalemate"
        elif board.is_fifefold_repetition():
            msg = "draw: 5-fold repetition"
        elif board.is_insufficient_material():
            msg = "draw: insufficient material"
        elif board.can_claim_draw():
            msg = "draw: claim"
        if visual is not None:
            print(msg)
        return (result, msg, board)

```

Nótese que la función `play_game` pide dos variables a la entrada, que son precisamente los dos jugadores de ajedrez que se enfrentan en la partida. Estos jugadores tomarán la posición actual del tablero como entrada, y emitirán una jugada a la salida. Por lo tanto, la idea general es recorrer mediante un ciclo todas las jugadas legales de la posición y elegir una de ellas.

El siguiente paso será programar dichos jugadores de ajedrez. Primeramente, y a modo de prueba, se crea una función `random_player`, que elige una de las jugadas legales aleatoriamente.

```
import random
```

```
def random_player(board):
    move = random.choice(list(board.legal_moves))
    return move.uci()
```

Antes de pasar a programar el motor de ajedrez como tal, se crea la función `human_player`, que hará posible que un jugador humano también pueda participar en `play_game`. Esta función también se asegura de que el jugador humano realice jugadas legales, para evitar errores en la función `play_game`.

```
def human_player(board):
    display(board)
    uci = get_move("%s's move [q to quit]> " % who(board.turn))
    legal_uci_moves = [move.uci() for move in board.legal_moves]
    while uci not in legal_uci_moves:
        print("Legal moves: " + (",".join(sorted(legal_uci_moves))))
        uci = get_move("%s's move[q to quit]> " % who(board.turn))
    return uci
```

```
def get_move(prompt):
    uci = input(prompt)
    if uci and uci[0] == "q":
        raise KeyboardInterrupt()
    try:
        chess.Move.from_uci(uci)
    except:
        uci = None
    return uci
```

A continuación, se presenta una primera aproximación al motor de ajedrez buscado, obtenida en la función `player1`. Se vuelven a recorrer todas las jugadas legales mediante un ciclo, pero en esta ocasión, en vez de elegir una de ellas aleatoriamente, se ordenan según la puntuación obtenida al evaluar la posición resultante de cada jugada en la función `staticAnalysis1`, y se elige la jugada que haya obtenido la mayor puntuación.

Para evaluar las posiciones en `staticAnalysis1`, se inicializa la variable "score" y se iguala a 0. Posteriormente, mediante un ciclo se recorren todas las piezas del tablero y se le suma el valor de cada pieza a "score" en el caso de que las piezas sean de `player1`, y se le resta el valor de cada pieza a "score" en el caso de que las piezas sean del oponente. Por último, la función `staticAnalysis1` retorna el valor final de "score" para que la función `player1` pueda proceder a la ordenación de jugadas.

Con esta función se ha conseguido ya la idea principal del motor de ajedrez diseñado, que era que evaluara las posiciones en base al material. No obstante, `player1` todavía desarrolla un juego muy pobre, porque cuando no se da la oportunidad de capturar ninguna pieza, todas las jugadas poseen la misma evaluación y `player1` tiende a ordenarlas por defecto siempre de la misma manera, con lo que se estanca y acaba repitiendo siempre las mismas jugadas.

```
def player1(board):
    moves = list(board.legal_moves)
    for move in moves:
        newboard = board.copy()
        # go through board and return a score
        move.score = staticAnalysis1(newboard, move, board.turn)
    moves.sort(key=lambda move: move.score, reverse=True) # sort on score
    return moves[0].uci()
```

```
def staticAnalysis1(board, move, my_color):
    score = 0
    ## Check some things about this move:
    # To actually make the move:
    board.push(move)
    # Now check some other things:
    for (piece, value) in [(chess.PAWN, 1),
                           (chess.BISHOP, 4),
                           (chess.KING, 0),
                           (chess.QUEEN, 10),
                           (chess.KNIGHT, 5),
                           (chess.ROOK, 3)]:
        score += len(board.pieces(piece, my_color)) * value
        score -= len(board.pieces(piece, not my_color)) * value
    # can also check things about the pieces position here
    return score
```

Player2 es una ligera, pero a la vez notable mejora de player1. En lugar de inicializar la variable “score” a 0, se le asigna un valor aleatorio entre 0 y 1. Este valor no influirá en la evaluación de material final, por ser menor todavía que el valor de un peón. Pero el valor será lo suficientemente grande para que player2 no haga siempre las mismas jugadas en el caso de que no se de la posibilidad de capturar una pieza.

```
def player2(board):
    moves = list(board.legal_moves)
    for move in moves:
        newboard = board.copy()
        # go through board and return a score
        move.score = staticAnalysis2(newboard, move, board.turn)
    moves.sort(key=lambda move: move.score, reverse=True) # sort on score
    return moves[0].uci()
```

```
def staticAnalysis2(board, move, my_color):
    score = random.random()
    ## Check some things about this move:
    # To actually make the move:
    board.push(move)
    # Now check some other things:
    for (piece, value) in [(chess.PAWN, 1),
                           (chess.BISHOP, 4),
                           (chess.KING, 0),
                           (chess.QUEEN, 10),
                           (chess.KNIGHT, 5),
                           (chess.ROOK, 3)]:
        score += len(board.pieces(piece, my_color)) * value
        score -= len(board.pieces(piece, not my_color)) * value
    # can also check things about the pieces position here
    return score
```


El problema principal de player2 es que no es capaz de ganar las partidas, porque pasa por alto el remate final. Por lo tanto, la mejora principal de player3 es que se le añadirá un bonus de 100 puntos al “score” de la posición resultante que haya desembocado en jaque mate. Además, con un razonamiento similar se evita ahogar al rival (ahogar al rival no siempre es malo, pero en la mayoría de los casos es deseable evitarlo), y potenciar todavía más las capturas y el enroque (que una vez más no siempre es bueno enrocarse, pero normalmente es aconsejable hacerlo lo antes posible).

```
def player3(board):
    moves = list(board.legal_moves)
    for move in moves:
        newboard = board.copy()
        # go through board and return a score
        move.score = staticAnalysis3(newboard, move, board.turn)
    moves.sort(key=lambda move: move.score, reverse=True) # sort on score
    return moves[0].uci()
```

```
def staticAnalysis3(board, move, my_color):
    ## Check some things about this move:
    score = random.random()
    score += 1 if board.is_capture(move) else 0
    score += 1 if board.is_castling(move) else 0
    # To actually make the move:
    board.push(move)
    # Now check some other things:
    for (piece, value) in [(chess.PAWN, 1),
                           (chess.BISHOP, 3),
                           (chess.KING, 0),
                           (chess.QUEEN, 9),
                           (chess.KNIGHT, 3),
                           (chess.ROOK, 5)]:
        score += len(board.pieces(piece, my_color)) * value
        score -= len(board.pieces(piece, not my_color)) * value
    # can also check things about the pieces position here
    # Check global things about the board
    score += 100 if board.is_checkmate() else 0
    score -= 100 if board.is_stalemate() else 0
    return score
```

Hasta ahora todas las valoraciones que se han seguido para elegir la próxima jugada se han hecho sobre la posición inminentemente posterior. Esto da lugar a jugadas tan absurdas como sacrificar la dama por un peón, ya que como el motor solo piensa en la jugada inmediatamente anterior, solo se fija en la ganancia del peón. Como no podía ser de otra manera, la siguiente mejora consiste en dotar al motor de ajedrez de la capacidad de cálculo. Para ello, primero se va a optar por el algoritmo minimax, que ya se ha explicado en apartados anteriores.

```
def player4(board):
    score_and_move = minimax(board, 3)
    move = score_and_move[1]
    return move.uci()
```

```

def minimax(board, depth):
    #Returns a tuple (score, bestmove) for the position at the given depth
    if depth == 0 or board.is_checkmate() or board.is_stalemate() or board.is_f
        return [staticAnalysis4(board), None]
    else:
        if board.turn == chess.WHITE:
            bestscore = -float("inf")
            bestmove = None
            for move in list(board.legal_moves):
                newboard = board.copy()
                newboard.push(move)
                score_and_move = minimax(newboard, depth - 1)
                score = score_and_move[0]
                if score > bestscore: # white maximizes her score
                    bestscore = score
                    bestmove = move
            return [bestscore, bestmove]
        else:
            bestscore = float("inf")
            bestmove = None
            for move in list(board.legal_moves):
                newboard = board.copy()
                newboard.push(move)
                score_and_move = minimax(newboard, depth - 1)
                score = score_and_move[0]
                if score < bestscore: # black minimizes his score
                    bestscore = score
                    bestmove = move
            return [bestscore, bestmove]

```

```

def staticAnalysis4(board):
    score = random.random()
    for (piece, value) in [(chess.PAWN, 1),
                          (chess.BISHOP, 3),
                          (chess.KING, 0),
                          (chess.QUEEN, 9),
                          (chess.KNIGHT, 3),
                          (chess.ROOK, 5)]:
        score += len(board.pieces(piece, chess.WHITE)) * value
        score -= len(board.pieces(piece, chess.BLACK)) * value
        # can also check things about the pieces position here
    # Check global things about the board
    if board.turn == chess.BLACK and board.is_checkmate():
        score += 100
    if board.turn == chess.WHITE and board.is_checkmate():
        score -= 100
    return score

```

Finalmente, se le va a aplicar una última mejora al motor de ajedrez. Se va a sustituir el algoritmo minimax por la poda alfa-beta, disminuyendo considerablemente el número de variantes a examinar y, por lo tanto, el tiempo de ejecución.

```

def player5(board):
    score_and_move = alphabeta(board, 3, -float("inf"), float("inf"))
    move = score_and_move[1]
    return move.uci()

```

```

def alphabeta(board, depth, alpha, beta):
    #Returns a tuple (score, bestmove) for the position at the given depth
    if depth == 0 or board.is_checkmate() or board.is_stalemate() or board.is_f
        return [staticAnalysis5(board), None]
    else:
        if board.turn == chess.WHITE:
            bestmove = None
            for move in board.legal_moves:
                newboard = board.copy()
                newboard.push(move)
                score_and_move = alphabeta(newboard, depth - 1, alpha, beta)
                score = score_and_move[0]
                if score > alpha: # white maximizes her score
                    alpha = score
                    bestmove = move
                    if alpha >= beta: # alpha-beta cutoff
                        break
            return [alpha, bestmove]
        else:
            bestmove = None
            for move in board.legal_moves:
                newboard = board.copy()
                newboard.push(move)
                score_and_move = alphabeta(newboard, depth - 1, alpha, beta)
                score = score_and_move[0]
                if score < beta: # black minimizes his score
                    beta = score
                    bestmove = move
                    if alpha >= beta: # alpha-beta cutoff
                        break
            return [beta, bestmove]

```

```

def staticAnalysis5(board):
    score = random.random()
    for (piece, value) in [(chess.PAWN, 1),
                           (chess.BISHOP, 3),
                           (chess.KING, 0),
                           (chess.QUEEN, 9),
                           (chess.KNIGHT, 3),
                           (chess.ROOK, 5)]:
        score += len(board.pieces(piece, chess.WHITE)) * value
        score -= len(board.pieces(piece, chess.BLACK)) * value
    # can also check things about the pieces position here
    # Check global things about the board
    if board.turn == chess.BLACK and board.is_checkmate():
        score += 100
    if board.turn == chess.WHITE and board.is_checkmate():
        score -= 100
    return score

```

```

first_game = play_game(human_player, player5)

```

15 Anexo II: Resultados experimentales

En este apartado se presentan algunas de las partidas que se han ido jugando entre los jugadores programados, para comprobar que el programa respondía con las mejoras esperadas.

15.1 Enfrentamiento 1

- Blancas: player1
- Negras: random_player
- Resultado: 0.5 – 0.5 (tablas por repetición)

```
[ '1.0. g1h3', '1.5. g7g5', '2.0. h3g5', '2.5. f8h6', '3.0. g5h7', '3.5. b7b6', '4.0. h7f8', '4.5. h6f4', '5.0. f8d7', '5.5. c8d7', '6.0. h1g1', '6.5. d7b5', '7.0. g1h1', '7.5. e7e6', '8.0. h1g1', '8.5. f4g3', '9.0. h2g3', '9.5. d8d5', '10.0. g1h1', '10.5. e8d7', '11.0. h1h8', '11.5. c7c6', '12.0. h8g8', '12.5. d5e4', '13.0. g8b8', '13.5. b5a6', '14.0. b8a8', '14.5. f7f5', '15.0. a8a7', '15.5. d7c8', '16.0. a7a6', '16.5. e4c2', '17.0. d1c2', '17.5. c8b7', '18.0. a6b6', '18.5. b7c7', '19.0. b6c6', '19.5. c7d8', '20.0. c6e6', '20.5. d8d7', '21.0. c2f5', '21.5. d7c7', '22.0. e6e8', '22.5. c7b7', '23.0. e8h8', '23.5. b7c7', '24.0. h8g8', '24.5. c7c6', '25.0. g8h8', '25.5. c6c7', '26.0. h8g8', '26.5. c7d6', '27.0. g8h8' ]
draw: claim
```

Ilustración 32. Planilla del primer enfrentamiento

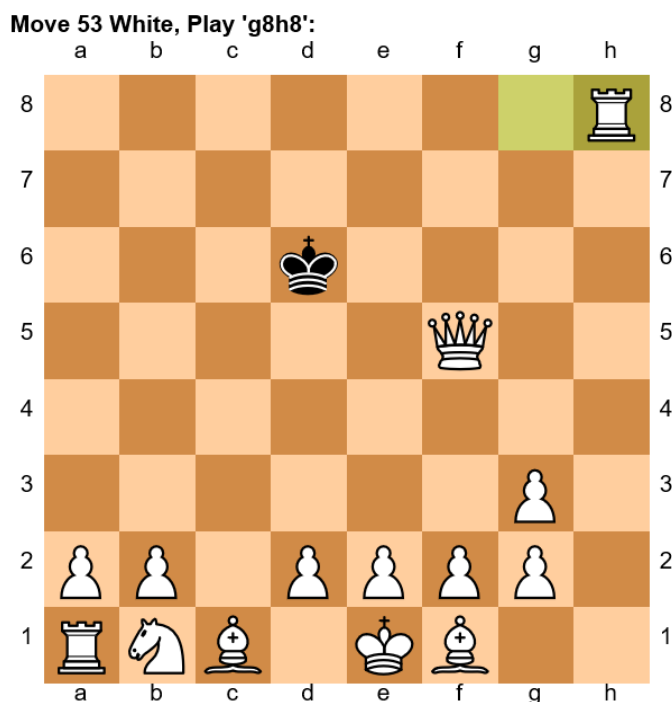


Ilustración 33. Posición resultante tras el primer enfrentamiento

Debido a que random_player hace jugadas aleatorias, player1 se las ha apañado para elegir la jugada que capturaba una pieza de su rival siempre que éste se la ponía en bandeja. No obstante, una vez que random_player se ha quedado sin piezas, player1 ha entrado en el problema de estancación previamente mencionado, y el resultado de la partida ha sido tablas por triple repetición.

15.2 Enfrentamiento 2

- Blancas: player2

- Negras: random_player
- Resultado: 0.5 – 0.5 (tablas por ahogado)

```
[ '1.0. g1f3', '1.5. b8a6', '2.0. d2d3', '2.5. e7e6', '3.0. d3d4', '3.5. f8d6', '4.0. f3g5', '4.5. d6b4', '5.0. b1d2', '5.5. g8h6', '6.0. g5f7', '6.5. b4c3', '7.0. f7d8', '7.5. e6e5', '8.0. b2c3', '8.5. c7c6', '9.0. d8c6', '9.5. a8b8', '10.0. c6b8', '10.5. b7b6', '11.0. b8a6', '11.5. e8f7', '12.0. d4e5', '12.5. h6f5', '13.0. e2e4', '13.5. c8b7', '14.0. e4f5', '14.5. b7d5', '15.0. a6b4', '15.5. d5c6', '16.0. b4c6', '16.5. d7d5', '17.0. e5d6', '17.5. h7h6', '18.0. c6a7', '18.5. g7g5', '19.0. f5g6', '19.5. f7e6', '20.0. d2b1', '20.5. h8h7', '21.0. g6h7', '21.5. e6d7', '22.0. h7h8q', '22.5. b6b5', '23.0. c1h6', '23.5. d7e6', '24.0. a7b5', '24.5. e6d7', '25.0. b1d2', '25.5. d7e6', '26.0. b5a7', '26.5. e6d6', '27.0. d1g4', '27.5. d6d5', '28.0. h8g7', '28.5. d5c5', '29.0. a2a4', '29.5. c5b6', '30.0. g4h4', '30.5. b6c5', '31.0. h4d8' ]
draw: stalemate
```

Ilustración 34. Planilla del segundo enfrentamiento

Move 61 White, Play 'h4d8':

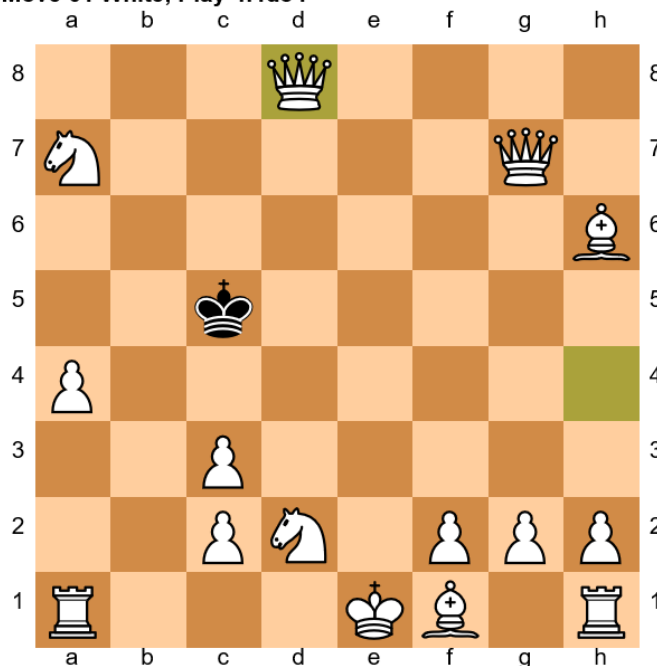


Ilustración 35. Posición resultante tras el segundo enfrentamiento

El desarrollo de la partida ha sido similar al del enfrentamiento anterior, solo que en esta ocasión player2 ha demostrado ser capaz de no estancarse cuando no tiene la oportunidad de capturar ninguna pieza. De todas formas, eso no ha sido suficiente para ganar la partida, ya que, a pesar de tener mucha ventaja de material, player2 no ha sabido dar jaque mate y ha acabado ahogando a su rival.

15.3 Enfrentamiento 3

- Blancas: player3
- Negras: random_player
- Resultado: 1 – 0

['1.0. h2h3', '1.5. c7c5', '2.0. g2g3', '2.5. f7f5', '3.0. g3g4', '3.5. g7g6', '4.0. g4f5', '4.5. d8c7', '5.0. f5g6', '5.5. b7b6', '6.0. g6h7', '6.5. d7d6', '7.0. h7g8q', '7.5. c7d8', '8.0. g8h8', '8.5. c8e6', '9.0. h8f8', '9.5. e8f8', '10.0. c2c3', '10.5. b8a6', '11.0. h1h2', '11.5. d6d5', '12.0. d2d3', '12.5. e6g4', '13.0. h3g4', '13.5. e7e6', '14.0. c1g5', '14.5. f8e8', '15.0. g5d8', '15.5. a8b8', '16.0. d8b6', '16.5. b8b6', '17.0. d1c1', '17.5. e8e7', '18.0. f1g2', '18.5. b6b3', '19.0. a2b3', '19.5. e7e8', '20.0. a1a6', '20.5. e6e5', '21.0. g2d5', '21.5. c5c4', '22.0. d3c4', '22.5. e8e7', '23.0. a6a7', '23.5. e7f6', '24.0. c1h6']
 checkmate: White wins!

Ilustración 36. Planilla del tercer enfrentamiento

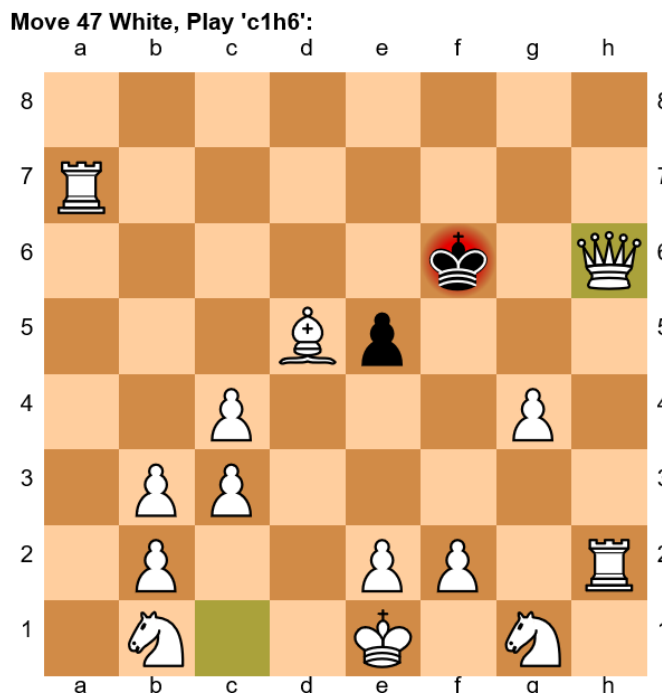


Ilustración 37. Posición resultante tras el tercer enfrentamiento

En esta ocasión, player3 no solo ha demostrado la mejora adoptada por player2, sino que también ha sabido rematar la partida cuando ha tenido la oportunidad.

15.4 Enfrentamiento 4

- Blancas: Player4 (profundidad 3)
- Negras: Player3
- Resultado: 1 – 0

['1.0. e2e3', '1.5. b7b5', '2.0. f1b5', '2.5. f7f5', '3.0. d1h5', '3.5. g7g6', '4.0. h5h4', '4.5. e7e6', '5.0. h4d4', '5.5. f8a3', '6.0. d4h8', '6.5. a3b2', '7.0. h8g8', '7.5. e8e7', '8.0. g8h7', '8.5. e7e8', '9.0. h7g8', '9.5. e8e7', '10.0. g8d8', '10.5. e7d8', '11.0. c1b2', '11.5. e6e5', '12.0. b2e5', '12.5. g6g5', '13.0. e5f6', '13.5. d8e8', '14.0. f6g5', '14.5. b8c6', '15.0. f2f4', '15.5. a7a6', '16.0. b5d3', '16.5. c6a7', '17.0. d3f5', '17.5. a7c6', '18.0. g1e2', '18.5. c6d4', '19.0. e2d4', '19.5. a8b8', '20.0. h1g1', '20.5. b8b1', '21.0. a1b1', '21.5. c7c5', '22.0. d4b3', '22.5. e8f7', '23.0. b3c5', '23.5. a6a5', '24.0. b1b8', '24.5. c8b7', '25.0. b8b7', '25.5. f7f8', '26.0. f5d7', '26.5. a5a4', '27.0. d7a4', '27.5. f8g8', '28.0. d2d3', '28.5. g8h8', '29.0. d3d4', '29.5. h8g8', '30.0. e1d1', '30.5. g8f8', '31.0. a4b3', '31.5. f8e8', '32.0. b3f7', '32.5. e8f8', '33.0. g5h6']
 checkmate: White wins!

Ilustración 38. Planilla del cuarto enfrentamiento

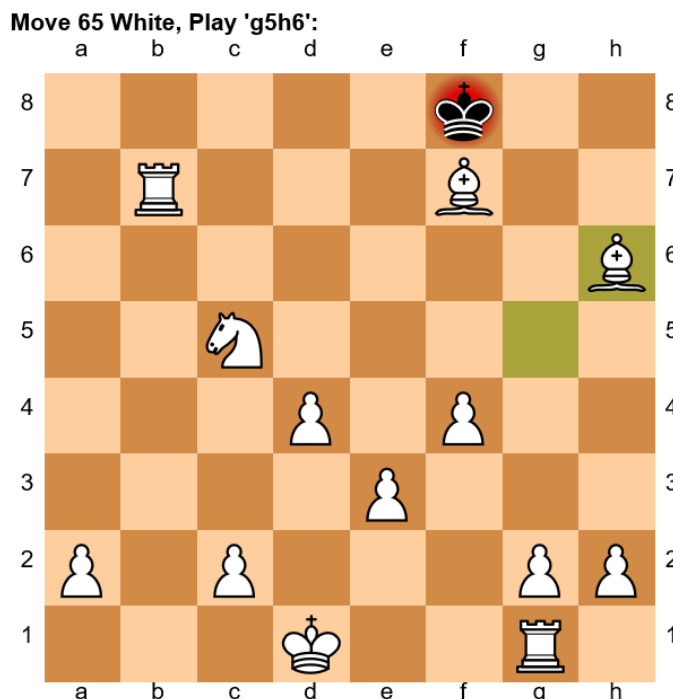


Ilustración 39. Posición resultante tras el cuarto enfrentamiento

Como se puede deducir de la posición resultante, el hecho de ser capaz de anticipar las respuestas de su rival, le ha supuesto a player4 una ventaja más que decisiva.

15.5 Enfrentamiento 5

- Blancas: player5 (profundidad 3)
- Negras: player4 (profundidad 3)
- Resultado: 1 – 0

```
[ '1.0. d2d4', '1.5. b8c6', '2.0. c1g5', '2.5. c6b4', '3.0. d1d2', '3.5. a7a5', '4.0. d4d5', '4.5. f7f6', '5.0. g5h4', '5.5. g7g5', '6.0. h4g3', '6.5. a8a6', '7.0. e2e3', '7.5. a6b6', '8.0. f1e2', '8.5. h7h5', '9.0. c2c3', '9.5. b4a6', '10.0. e2d3', '10.5. h8h6', '11.0. d2e2', '11.5. a6c5', '12.0. e2h5', '12.5. h6h5', '13.0. d3g6' ]
checkmate: White wins!
```

Ilustración 40. Planilla del quinto enfrentamiento

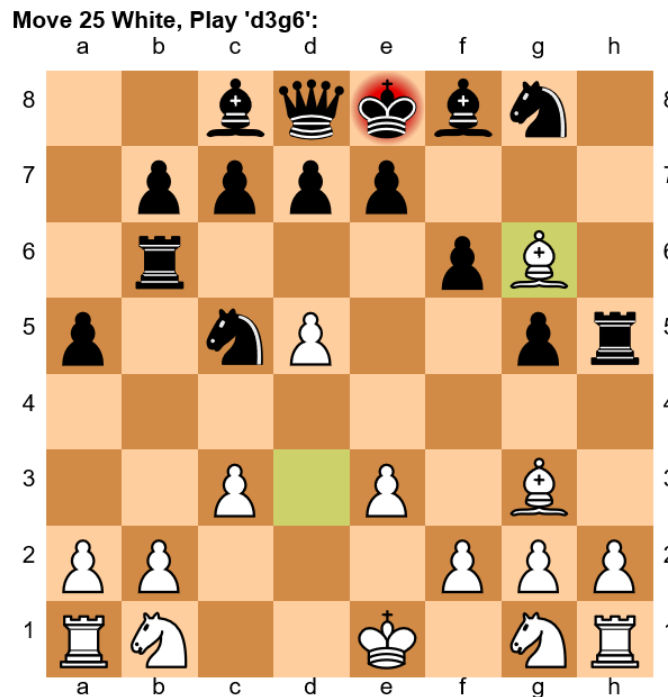


Ilustración 41. Posición resultante tras el quinto enfrentamiento

El resultado de esta partida era completamente impredecible a priori, ya que ambos jugadores tenían exactamente la misma fuerza de juego. No obstante, se han podido comprobar un par de cosas durante el transcurso de la partida

La primera de ellas es que el tiempo que ha necesitado player5 ha sido considerablemente menor, por lo que queda claro que se le facilita mucho el trabajo al motor de ajedrez al utilizar la poda alfa-beta en lugar del algoritmo minimax.

La segunda de ellas es que mantener la iniciativa en ajedrez es tremendamente importante. El bando que ha jugado con blancas ha estado planteando amenazas constantemente y las negras se han limitado a defender. El resultado ha sido una victoria rápida y contundente de las blancas pese a tener los dos bandos exactamente el mismo nivel.

15.6 Enfrentamiento 6

- Blancas: player5 (profundidad 3)
- Negras: player 4 (profundidad 3)
- Resultado: 1 – 0


```
[ '1.0. e2e3', '1.5. e7e6', '2.0. d1h5', '2.5. d8f6', '3.0. f1d3', '3.5. g7g6', '4.0. h5h3', '4.5. f8h6', '5.0. g1e2', '5.5. a7a6', '6.0. a2a3', '6.5. d7d6', '7.0. e2g3', '7.5. f6g5', '8.0. f2f4', '8.5. g5d5', '9.0. b1c3', '9.5. d5c6', '10.0. d3e4', '10.5. d6d5', '11.0. e4d5', '11.5. c6c5', '12.0. d2d4', '12.5. c5a5', '13.0. b2b4', '13.5. a5b6', '14.0. c3a4', '14.5. b6b5', '15.0. a4c3', '15.5. b5b6', '16.0. c3a4', '16.5. b6d6', '17.0. d5e6', '17.5. c8e6', '18.0. f4f5', '18.5. e6d7', '19.0. g3e4', '19.5. d6c6', '20.0. a4c5', '20.5. d7f5', '21.0. h3h4', '21.5. f5e4', '22.0. c5e4', '22.5. c6c2', '23.0. e1g1', '23.5. c2d3', '24.0. e4f2', '24.5. d3c2', '25.0. g2g4', '25.5. c2e2', '26.0. b4b5', '26.5. a6b5', '27.0. f2h1', '27.5. g6g5', '28.0. h4h5', '28.5. e2c4', '29.0. f1f7', '29.5. c4f7', '30.0. h5h3', '30.5. b8c6', '31.0. d4d5', '31.5. f7d5', '32.0. h3h5', '32.5. e8f8', '33.0. c1b2', '33.5. c6e5', '34.0. a1f1', '34.5. e5f3', '35.0. g1g2', '35.5. d5f7', '36.0. f1f3', '36.5. g8f6', '37.0. f3f6', '37.5. f7f6', '38.0. b2f6', '38.5. h6g7', '39.0. h5g5', '39.5. g7f6', '40.0. g5f6', '40.5. f8g8', '41.0. g2g3', '41.5. a8a3', '42.0. f6d8', '42.5. g8g7', '43.0. d8e7', '43.5. g7g8', '44.0. e7a3', '44.5. g8f7', '45.0. a3c5', '45.5. c7c6', '46.0. c5f5', '46.5. f7g7', '47.0. f5e5', '47.5. g7g8', '48.0. e5b8', '48.5. g8g7', '49.0. b8e5', '49.5. g7g8', '50.0. g3f3', '50.5. h7h6', '51.0. e5e8', '51.5. g8g7', '52.0. e8e5', '52.5. g7g8', '53.0. e5b8', '53.5. g8g7', '54.0. b8b7', '54.5. g7g6', '55.0. b7c6', '55.5. g6f7', '56.0. c6b5', '56.5. h8d8', '57.0. b5h5', '57.5. f7g7', '58.0. f3e2', '58.5. d8f8', '59.0. h2h3', '59.5. f8b8', '60.0. h5e5', '60.5. g7h7', '61.0. e5b8', '61.5. h7g6', '62.0. b8f4', '62.5. h6h5', '63.0. f4d6', '63.5. g6g5', '64.0. d6d5', '64.5. g5f6', '65.0. d5h5', '65.5. f6e7', '66.0. h5f5', '66.5. e7d6', '67.0. f5f7', '67.5. d6c5', '68.0. f7b7', '68.5. c5d6', '69.0. h1f2', '69.5. d6e6', '70.0. f2e4', '70.5. e6e5', '71.0. b7f7', '71.5. e5e4', '72.0. f7f5']
checkmate: White wins!
```

Ilustración 42. Planilla de sexto enfrentamiento

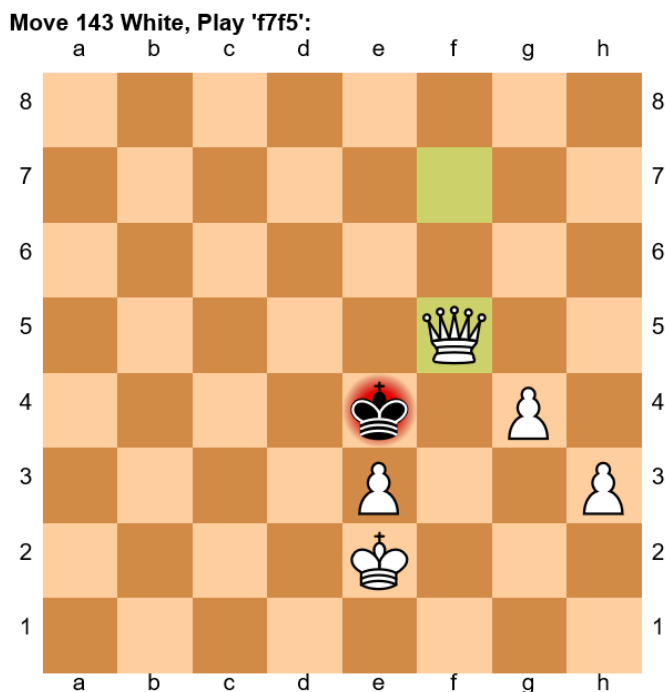


Ilustración 43. Posición resultante tras el sexto enfrentamiento

Esta partida enfrenta de nuevo a los dos jugadores anteriores, pero en esta ocasión el desarrollo de la partida es más normal de acuerdo al similar nivel que poseen los jugadores.

15.7 Enfrentamiento 7

- Blancas: human_player (Daniel Ochoa de Eribe Martínez)
- Negras: jugador5
- Resultado: 1 – 0

['1.0. e2e4', '1.5. e7e5', '2.0. f1c4', '2.5. d8h4', '3.0. b1c3', '3.5. f8c5', '4.0. d1e2', '4.5. c5f2', '5.0. e2f2', '5.5. h4h5', '6.0. g1f3', '6.5. b7b5', '7.0. c4b5', '7.5. c7c6', '8.0. b5a4', '8.5. f7f6', '9.0. e1g1', '9.5. c8a6', '10.0. f1e1', '10.5. h5g4', '11.0. d2d4', '11.5. g8e7', '12.0. d4e5', '12.5. h7h5', '13.0. e5f6', '13.5. g7f6', '14.0. e4e5', '14.5. f6e5', '15.0. f3e5', '15.5. g4e6', '16.0. e5c6', '16.5. e6d6', '17.0. e1e7', '17.5. d6e7', '18.0. c6e7', '18.5. e8e7', '19.0. c1g5', '19.5. e7e8', '20.0. a1e1', '20.5. a6e2', '21.0. e1e2']
 checkmate: White wins!

Ilustración 44. Planilla del séptimo enfrentamiento

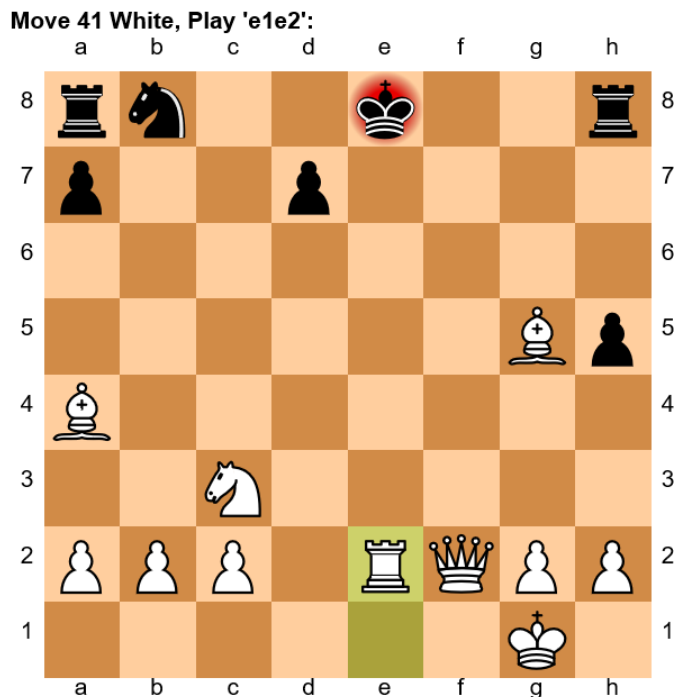


Ilustración 45. Posición resultante tras el séptimo enfrentamiento

Por último, una partida en la que se prueba el motor de ajedrez definitivo contra un jugador humano, para comprobar que la función human_player funciona también correctamente.