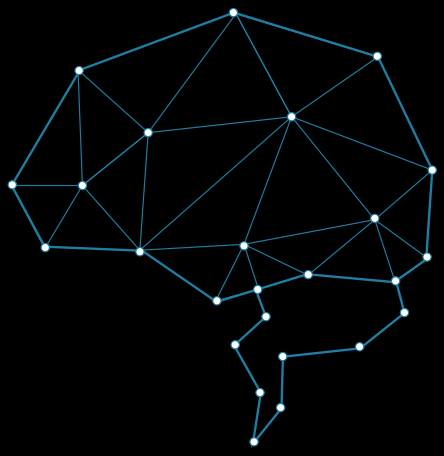
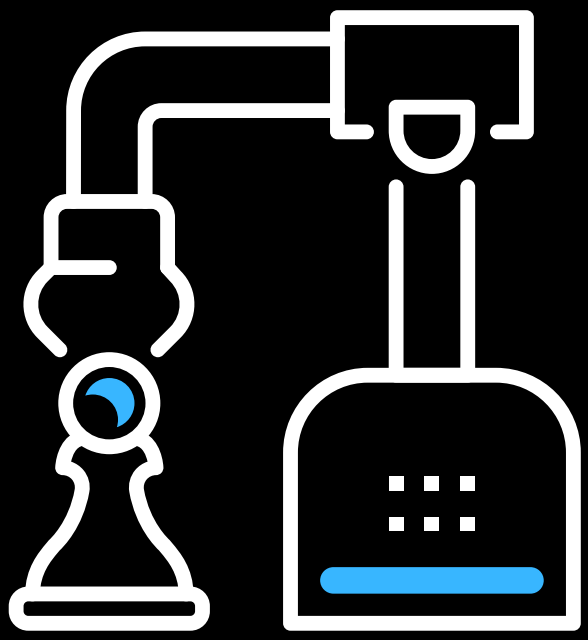


AJEDREZ Y COMPUTACIÓN

Alejandro Fernández Camello



Ajedrez y Computación

Alejandro Fernández Camello

10 de noviembre de 2023

Índice general

Índice de tablas

Índice de figuras

Introducción

¿Por qué deberíamos entrelazar el estudio del ajedrez y la computación? El ajedrez es indudablemente uno de los juegos más enigmáticos y sofisticados que se disfrutan en cada rincón de nuestro vasto planeta. En su primera jugada, el participante con las piezas blancas tiene la opción de seleccionar entre 20 movimientos distintos. De la misma manera, el competidor con las piezas negras puede responder con 20 movimientos alternativos. Este aparente matiz sencillo implica que el número de juegos posibles con tan solo dos movimientos se eleva a un total de 400. Como es evidente, el crecimiento es exponencial y, por consiguiente, se transforma rápidamente en una tarea abrumadora para los seres humanos, quienes deben depender de su instinto y de un cálculo preciso y perspicaz.

Afortunadamente, las máquinas, con su capacidad para llevar a cabo millones de operaciones por segundo y ejecutarlas sin desfallecer, nos proporcionan una solución viable. Esta potencia computacional ha permitido a los humanos desvelar nuevos secretos en este juego milenario. No obstante, a pesar del inmenso apoyo que brindan, estas máquinas también tienen sus limitaciones. Un ejemplo palpable es que aún no se ha despejado la incógnita de si un resultado es inevitable desde el comienzo de una partida. En otras palabras, aún no sabemos si, dados los movimientos óptimos por ambos bandos, una victoria o un empate

es un desenlace ineluctable.

La relación entre el ajedrez y la computación ha sido estrecha desde los albores de esta última disciplina. Alan Turing, uno de los pioneros de la informática y ferviente jugador de ajedrez, es reconocido como el creador del primer programa de ajedrez, denominado Turochamp. Este programa se basaba en el algoritmo Minimax y en el uso de una heurística (conceptos que se analizarán más detalladamente en secciones posteriores de este libro). Aunque Turing no pudo probar su programa en una computadora antes de su muerte, logró simular el algoritmo actuando él mismo como una máquina capaz de ejecutar una serie de instrucciones.

Este libro se estructura en cinco capítulos, que abordan el ajedrez desde diferentes ángulos. La primera parte se enfoca en problemas y minijuegos derivados del ajedrez, como el reto de colocar n damas en un tablero de $n \times n$ sin que se ataquen entre sí y el tour del caballo, donde, utilizando un caballo, debemos recorrer todas las casillas del tablero sin repetir ninguna. Estos juegos se vinculan con conceptos informáticos como grafos, complejidad algorítmica, caminos hamiltonianos, coloreado de grafos, entre otros.

El segundo capítulo se concentra en la inteligencia artificial aplicada al ajedrez de forma general, incluyendo los métodos que los ordenadores empleaban para jugar al ajedrez en décadas pasadas.

El tercer capítulo explora los avances en inteligencia artificial usando aprendizaje por refuerzo y qué técnicas han permitido a AlphaZero revolucionar el ajedrez tal y cómo lo conocemos.

El cuarto capítulo discute los últimos avances en inteligencia artificial. La respuesta de Stockfish ante la derrota contra

AlphaZero y cómo un nuevo modelo llamado Maia busca imitar la toma de decisiones humanas en ajedrez.

En el quinto capítulo se analizarán los estándares más importantes de ajedrez. Entre ellos se incluye FEN, PGN y UCI. Estos se utilizan en muchos programas de ajedrez para facilitar la comunicación dentro de ellos y con otros programas. En especial, FEN y PGN son muy utilizados y es muy recomendable entender cómo funcionan.

Este libro tiene como propósito introducir al lector en el fascinante mundo de la computación a través de su relación con el ajedrez. Aunque está especialmente pensado para aquellos familiarizados con el ajedrez, se proporcionan dos anexos con información sobre las reglas del ajedrez y la notación utilizada para aquellos que no lo estén. En lo que respecta a la computación, no se requiere ningún conocimiento previo. Todos los conceptos informáticos utilizados a lo largo del libro se explican según aparecen, siempre procurando que sean accesibles para alguien no familiarizado con la informática. Por esta razón, se recomienda encarecidamente la lectura del libro en orden, ya que los conceptos se van construyendo desde el primer capítulo hasta el último.

Después de leer este libro, adquirirás conocimientos útiles sobre la informática que serán relevantes no solo para el ajedrez, sino también para muchas otras disciplinas en las que la informática juega un papel crucial. Este libro te permitirá, por lo tanto, desarrollar el denominado pensamiento computacional, tan demandado en la actualidad, que te permitirá entender el mundo desde nuevas perspectivas.

Capítulo 1

Problemas ajedrecísticos

1.1. El problema de las n-damas

1.1.1. Un problema milenario

Para estimular su curiosidad y compromiso en este fascinante capítulo, permítanme revelarles un intrigante incentivo: existe una recompensa de un millón de dólares para aquel que logre descubrir un algoritmo polinómico determinista que pueda resolver el problema que estamos a punto de describir para cualquier valor de n. Aunque el término "algoritmo polinómico" puede ser desconocido para algunos, no se preocupen, en las próximas secciones lo abordaremos con detalle para que puedan comprender su significado y relevancia.

Ahora, para mantener la emoción creciente, es hora de presentarles el problema en cuestión:

"Dado un tablero de ajedrez de dimensión $n \times n$ (lo que significa que cada fila y cada columna tiene n casillas), una solución al problema consiste en colocar n damas de tal manera que ninguna de ellas pueda atacar a otra".

Para expresarlo de manera más clara, el objetivo es colocar

las damas de tal manera que no puedan atacarse entre ellas. Esto implica que sólo puede haber una dama en cada fila, columna y diagonal. Para aquellos que no están familiarizados con las reglas del ajedrez, se ha incluido un apéndice al final del libro que explica cómo se mueven las diferentes piezas en este juego.

En caso de que esté pensando en aceptar este desafío, le informamos que resolver este problema es equivalente a demostrar que P=NP, una de las preguntas más importantes y sin resolver en la teoría de la computación. Puede encontrar más información sobre este premio y las reglas del concurso en la página web del Instituto de Matemáticas Clay[1] (última consulta el 8/9/2022).

Para familiarizarnos con la problemática en cuestión, es conveniente comenzar con las dimensiones más pequeñas, denotadas como n, del tablero de ajedrez.

Para $n = 1$, nos encontramos con un tablero que consta de una sola casilla. En este escenario, disponemos de una única dama para colocar. Debido a las dimensiones del tablero, sólo existe una posible ubicación para la dama, siendo esta la única solución viable para un tablero de este tamaño.

Al incrementar a $n = 2$ o $n = 3$, nos encontramos con un obstáculo. En estos casos, es imposible hallar una solución. La razón subyacente a esta afirmación es que en tableros de ajedrez de estas dimensiones, las damas no pueden ser ubicadas de manera que no se amenacen entre sí, según las reglas del juego.

Al aumentar el tamaño del tablero a $n = 4$, la situación se vuelve más interesante, ya que existen múltiples soluciones. En particular, hay dos soluciones posibles para un tablero de este tamaño. Una de las soluciones se puede visualizar directamente en el tablero. La otra solución se puede obtener mediante una

[1] https://www.claymath.org/millennium-problems/p-vs-np-problem

rotación del tablero que ya tiene una solución: simplemente necesitamos desplazar las damas siguiendo las direcciones indicadas por las flechas. De esta manera, se pueden explorar todas las posibles soluciones para un tablero de ajedrez de dimensiones $n = 4$.

La Figura 1.1 presenta ambas soluciones para el problema de las ocho reinas en un tablero de tamaño 4x4. La primera solución consiste en colocar a las damas en las casillas a2, b4, c1 y d3. En la segunda solución, las damas se desplazan a las casillas indicadas por las flechas, situándose en las casillas a3, c4, b1 y d2.

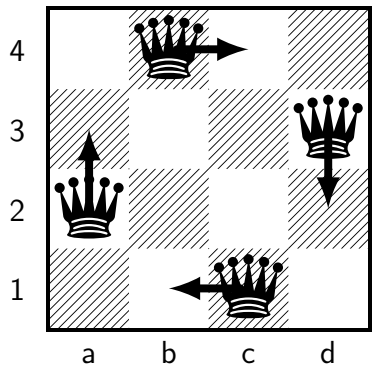
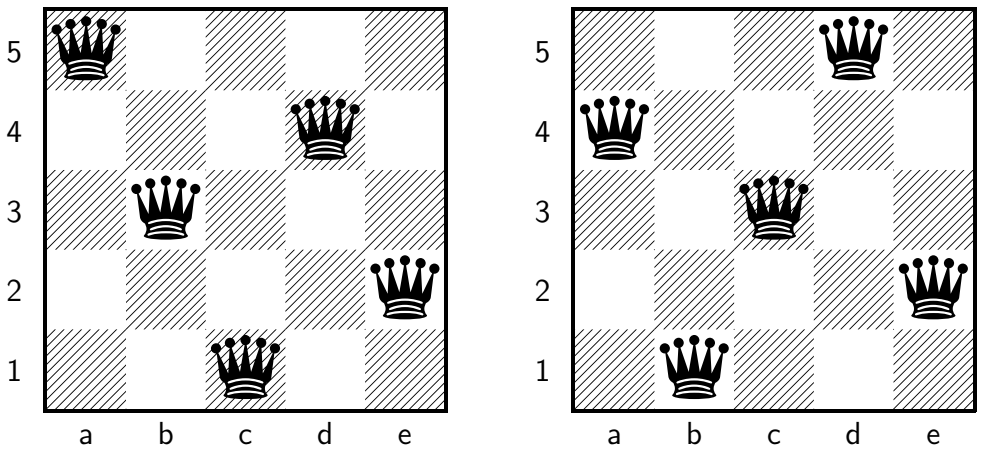


Figura 1.1: Soluciones del tablero 4x4

En el caso de un tablero de ajedrez de tamaño $n \times n$ con $n = 5$, se pueden encontrar numerosas soluciones, siendo hasta 10 soluciones posibles. Sin embargo, hay que tener en cuenta que solo existen realmente 2 soluciones únicas, ya que las demás son derivadas de estas mediante operaciones de reflexión y rotación.

Para ser más específicos, a partir de la primera solución (tablero de la izquierda), se pueden generar un total de 8 soluciones

diferentes, mientras que a partir de la segunda solución (tablero de la derecha), se pueden obtener únicamente 2 soluciones distintas, aplicando las operaciones de rotación y reflexión mencionadas anteriormente.



(a) 8 soluciones (b) 2 soluciones

Figura 1.2: Soluciones del tablero 5x5

A continuación, se explica de manera sencilla el funcionamiento de las rotaciones y reflexiones.

1. Rotación: Esta operación consiste en girar el tablero de ajedrez en ángulos de 90, 180 o 270 grados en sentido horario o antihorario. Al rotar el tablero, se cambia la posición de las piezas, pero la relación entre ellas se mantiene, lo que puede dar lugar a soluciones aparentemente diferentes pero que son equivalentes.

2. Reflexión: La reflexión implica "reflejar en un espejo" el tablero de ajedrez, es decir, invertir las posiciones de las

piezas de forma simétrica con respecto a un eje (horizontal, vertical o diagonal). Al igual que en la rotación, la relación entre las piezas se conserva, por lo que también puede generar soluciones equivalentes a las originales.

Una vez que el lector haya adquirido un buen entendimiento del problema que se está tratando, es posible realizar algunas observaciones generales sobre el mismo. Independientemente de la dimensión n del tablero, existen ciertos patrones geométricos que permanecen constantes y que pueden ayudar a facilitar la búsqueda de una solución de manera más eficaz. Sin embargo, en el enfoque tradicional de este problema, se asume que algunas damas ya están dispuestas en el tablero, lo que, en la mayoría de las circunstancias, obstaculiza la utilización de los patrones geométricos previamente mencionados. Por ende, resulta imperativo desarrollar un algoritmo que pueda resolver el problema bajo estas condiciones específicas.

Uno de los patrones geométricos considerados es el que consiste en ubicar las damas siguiendo el patrón de movimiento del caballo en el ajedrez, es decir, en forma de "L", para evitar que se ataquen entre sí. No obstante, esta solución solo resulta efectiva en casos particulares, dado que, en la mayoría de las situaciones, las damas terminarán atacándose entre sí. Un ejemplo de cómo funciona este patrón se puede observar en la Figura 1.2, donde todas las damas están dispuestas siguiendo el salto del caballo. Para tableros de mayor tamaño, es posible utilizar la misma técnica, pero con ciertas restricciones. La solución mostrada para un tablero de $n \times n$ es un ejemplo de ello.

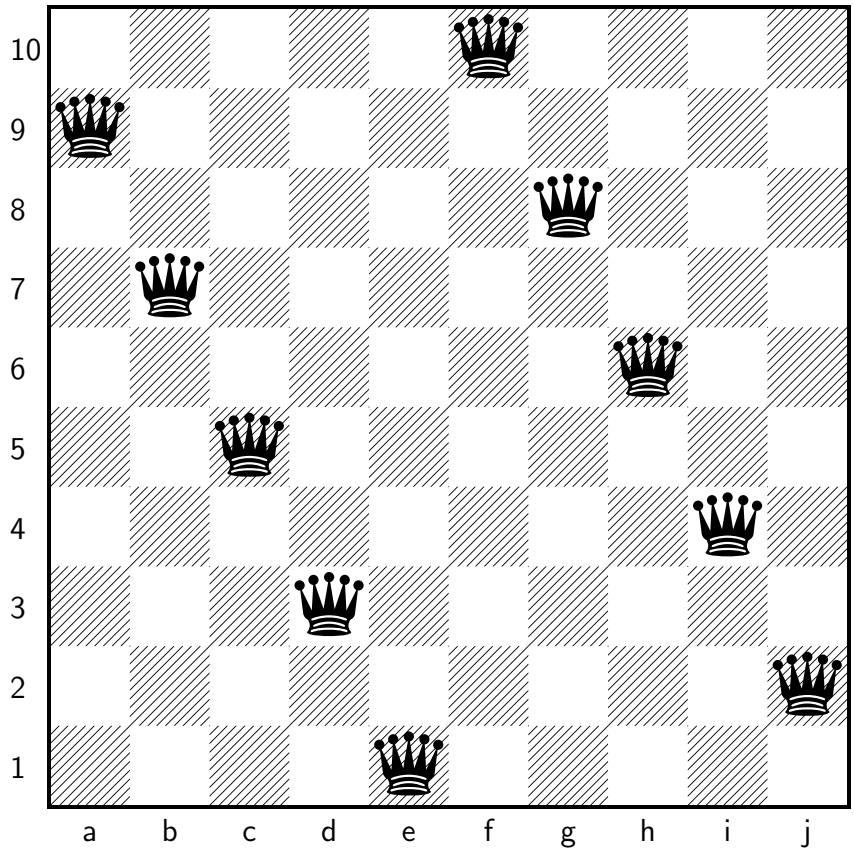


Figura 1.3: Soluciones del tablero 10x10 colocando las damas en L

Es esencial destacar que, si bien la mayoría de las damas se pueden ubicar utilizando el patrón de salto del caballo, en ocasiones es necesario prescindir de este patrón para satisfacer las restricciones del problema.

En síntesis, aunque la estrategia de disposición en forma de "L" puede resultar de gran utilidad, no constituye una solución universal para resolver el problema. En la siguiente sección, se presentará una técnica universal que permitirá resolver el pro-

blema en cualquier situación.

1.1.2. Vuelta atrás versus n-damas

El algoritmo de *Backtracking*, también conocido como "vuelta atrás", es uno de los esquemas algorítmicos más utilizados en la resolución de problemas complejos. De manera general, este algoritmo permite realizar una búsqueda exhaustiva en el espacio de todas las posibles soluciones, seleccionando aquellas que cumplen con los criterios definidos por el problema en cuestión. Además, su capacidad de generar todas las soluciones posibles permite identificar y seleccionar la que sea más óptima o favorable, de acuerdo con los requisitos específicos del problema.

A pesar de estas ventajas, el uso de *Backtracking* presenta una desventaja significativa: el número de soluciones posibles crece exponencialmente en la mayoría de los casos. Si se intenta generar todas las soluciones posibles, el algoritmo tendrá que enfrentarse al coste computacional exponencial inherente a este crecimiento.

Afortunadamente, existen estrategias para mitigar esta complejidad. Dos de las más prominentes son la poda y la búsqueda inteligente. La poda es una técnica que permite abortar la exploración de ciertas ramas del espacio de soluciones tan pronto como se determine que no pueden conducir a una solución válida. Por su parte, la búsqueda inteligente permite definir el orden en el que se exploran las diferentes ramas del espacio de soluciones, dando prioridad a aquellas que, según una heurística predefinida, sean más propensas a contener una solución válida. El concepto de heurística, así como su aplicación en el contexto de la Inteligencia Artificial, será abordado en profundidad en el capítulo dedicado a la Inteligencia Artificial aplicada al ajedrez.

A continuación, nos centraremos en la aplicación del algoritmo de *Backtracking* a nuestro problema en estudio.

1.1.2.1. Representación de los estados

Cuando se realiza un *Backtracking* a través de todas las "soluciones" o "estados" posibles, resulta crucial definir de manera apropiada cómo se van a representar dichos estados. Para el problema específico que estamos abordando, que involucra la ubicación de las damas en un tablero de ajedrez, también debemos considerar que puede haber damas que aún no hayan sido colocadas. Entonces, surge la pregunta: ¿Cómo representamos esto?

Una idea inicial comúnmente considerada es representar la posición de una dama mediante sus coordenadas en el tablero. Aquellas damas que aún no se han colocado simplemente no se incluyen en la lista de coordenadas. La Figura 1.4 muestra el tablero que se usará para mostrar las diferentes representaciones.

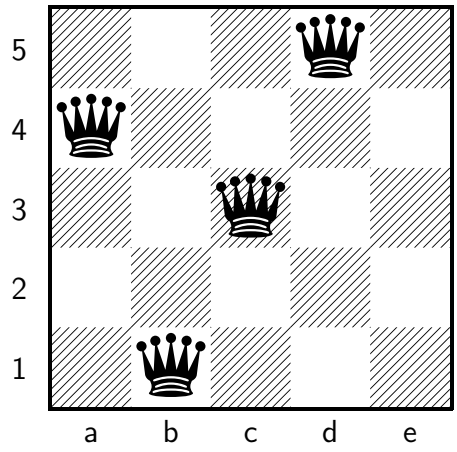


Figura 1.4: Tablero para mostrar la representación

La representación con coordenadas que corresponde al tablero presentado anteriormente es la siguiente:

$$[\text{a4}, \text{b1}, \text{c3}, \text{d5}]$$

Dado que hay cuatro damas, solo habría cuatro coordenadas en esta representación, aunque es posible colocar una quinta dama en el tablero fácilmente, añadiéndola en la coordenada e2.

La representación que hemos utilizado hasta ahora es útil, pero se puede mejorar, especialmente teniendo en cuenta que, de acuerdo con las reglas del problema, solo puede haber una dama en cada columna y fila. Esta restricción nos permite simplificar la representación de manera significativa.

La idea es emplear una lista de números enteros de tamaño n. Para cada columna, vamos a indicar en la lista en qué fila se encuentra la dama. Así, la primera posición de la lista corresponde a la columna "a", la segunda a la columna "b", y así

sucesivamente. Si en la columna "a" aparece el valor 3, esto indica que la dama ubicada en la columna "a" se encuentra en la fila 3. Si no se ha colocado una dama en una columna particular, asignamos un valor de -1 a esa posición en la lista.

Por lo tanto, la nueva representación del tablero en la Figura 1.4 sería:

$$[4, 1, 3, 5, -1]$$

Según la representación, se tiene la dama de la primera columna en la fila 4, la dama de segunda columna en la fila 1, la dama de la columna 3 en la fila 3, la dama de la columna 4 en la fila 5 y la dama de la columna 5 al no estar todavía colocada tiene un -1.

Gracias a esta representación se puede crear un algoritmo de manera sencilla para resolver este problema.

1.1.2.2. Funcionamiento del algoritmo

Para ilustrar cómo opera el algoritmo de *Backtracking*, se presenta primero un ejemplo paso a paso de cómo se resuelve un caso específico. Posteriormente, se proporciona una visión general del funcionamiento del algoritmo, representándolo a través de un diagrama de flujo.

La Figura 1.5 muestra la configuración inicial de las damas que se utilizará para resolver el problema.

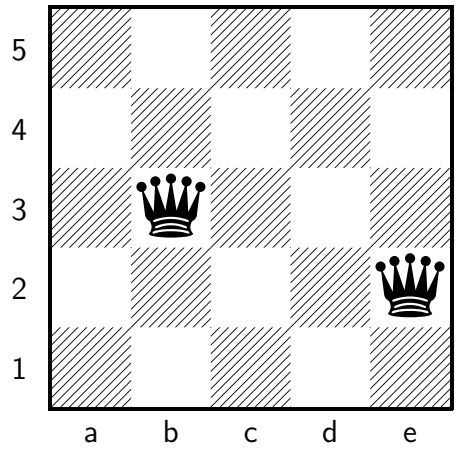


Figura 1.5: Posición inicial de resolución usando vuelta atrás

El primer paso del algoritmo consiste en establecer una representación correspondiente al tablero inicial. Esta representación se obtiene como sigue:

$$[-1,\ 3, -1, -1,\ 2]$$

El algoritmo procederá columna por columna, resolviendo aquellas columnas cuyo valor sea -1, es decir, las columnas que aún no tienen una dama colocada. El proceso comienza por la columna "a".

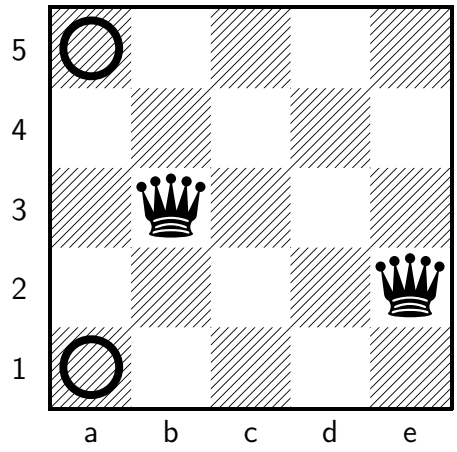


Figura 1.6: Primera fase de resolución usando vuelta atrás

Como se puede observar, hay dos posibles filas para colocar la dama: la 1 y la 5. Se opta por la primera, ya que 1 es menor que 5, aunque la elección podría hacerse en cualquier orden. Una vez colocada la dama en la primera columna, en la fila 1, se avanza al siguiente paso, que consiste en colocar la dama en la tercera columna (la dama de la segunda columna ya está colocada).

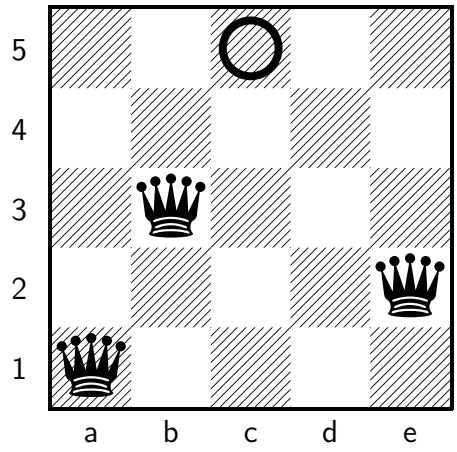


Figura 1.7: Segunda fase de resolución usando vuelta atrás

$$[\ 1,\ 3, -1, -1,\ 2]$$

En este punto, solo hay una opción viable, que consiste en colocar la dama de la columna "c" en la quinta fila, por lo que no se produce ninguna bifurcación. Una bifurcación se produce cuando hay que elegir entre diferentes opciones para colocar la dama en una fila determinada.

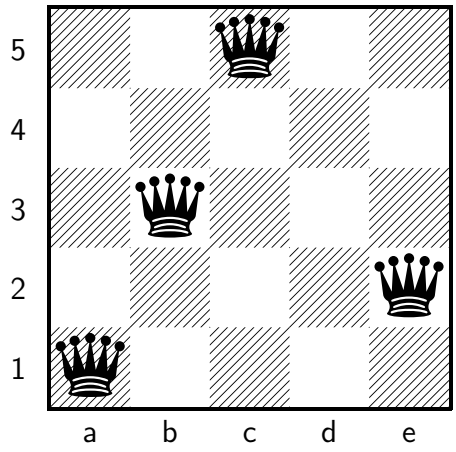


Figura 1.8: Tercera fase de resolución usando vuelta atrás

$$[\ 1,\ 3,\ 5, -1,\ 2]$$

Tras colocar la última dama, nos encontramos con una sorpresa desagradable, ya que no hay ninguna casilla en la columna "d" que no esté siendo atacada por otra dama. Por consiguiente, es imposible situar una dama en esa columna sin que sea atacada. Aquí es cuando el algoritmo de "vuelta atrás" entra en acción: como no se ha podido encontrar una solución en este camino, es necesario retroceder a la bifurcación anterior, donde se debía elegir entre diferentes opciones. En este caso concreto, se retrocede a la primera columna, donde se debía elegir entre las filas 1 y 5. Si hubiera habido varias opciones en la columna 3, se habría retrocedido a esa columna. Finalmente, se coloca la dama de la columna "a" en la quinta fila.

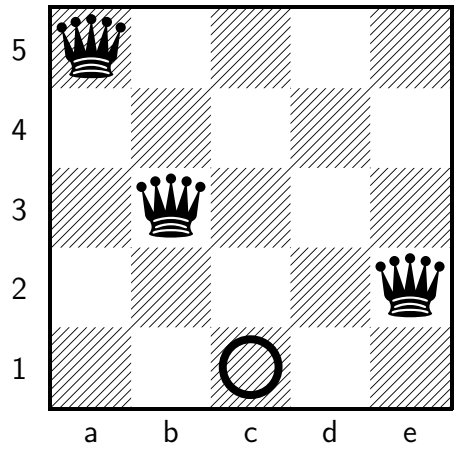


Figura 1.9: Cuarta etapa en la resolución del problema con la técnica de vuelta atrás

$$[5,\ 3, -1, -1,\ 2]$$

La solución continúa con la colocación de la siguiente reina en la única posición que queda disponible. Esta posición corresponde a la primera fila de la tercera columna.

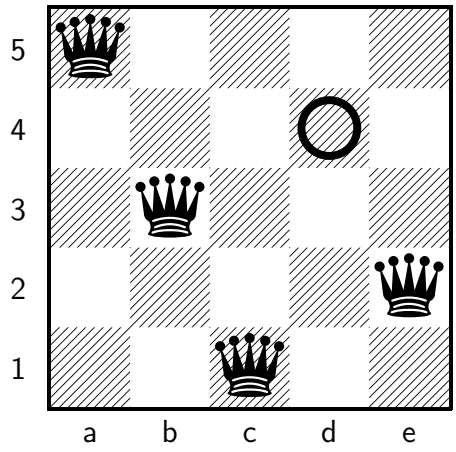


Figura 1.10: Quinta etapa en la resolución del problema con la técnica de vuelta atrás

$$[5,\ 3,\ 1, -1,\ 2]$$

Afortunadamente, en este caso, es posible continuar colocando reinas en la siguiente columna.

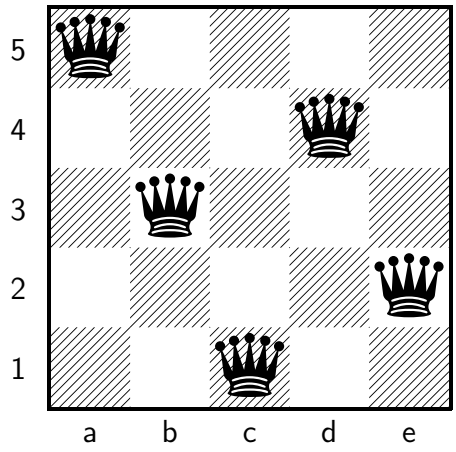


Figura 1.11: Fase final en la resolución del problema con la técnica de vuelta atrás

Una vez que se ha colocado una reina en cada columna y se ha confirmado que ninguna comparte la misma fila o diagonal con otra, podemos deducir que hemos hallado una solución. Según las necesidades específicas del problema, podríamos detenernos en este punto o continuar nuestra búsqueda hasta descubrir todas las posibles soluciones. En el caso presente, dado que no existen más bifurcaciones, es evidente que hemos llegado a una única solución.

Tras el análisis del algoritmo, surge una cuestión pertinente: ¿Cómo determina el ordenador que no existen dos reinas en la misma columna, fila o diagonal? En relación a la columna, no tendríamos que verificarlo, ya que la representación de los estados ya lo asegura. Respecto a las filas, sería sencillo comprobarlo examinando la lista del estado actual y verificando si el número de fila ya se encuentra en dicha lista. Sin embargo, existe un método aún más eficiente que implicaría el uso de una lista de

0s y 1s de tamaño igual al número de filas. En esta lista, un 0 en la posición i indicaría que la fila i está desocupada, y un 1 indicaría lo contrario. Este tipo de valor, que solo puede ser 0 o 1, se conoce en la informática como booleano.

Por último, en el caso de las diagonales, la situación sería algo más compleja, pero afortunadamente existe una fórmula que simplifica considerablemente este proceso si convertimos las columnas en números. Para llevar a cabo esta conversión, simplemente se reemplaza cada letra por su correspondiente posición en el alfabeto: la "a" se convierte en 1, la "b" en 2, y así sucesivamente. Con esta equivalencia en mano, podemos aplicar la fórmula.

Dos damas, representadas por las posiciones i y j en un vector V, están en la misma diagonal si y solo si $|i - j| = |V[i] - V[j]|$. En otras palabras, la diferencia absoluta entre las columnas i y j debe ser igual a la diferencia absoluta entre las filas $V[i]$ y $V[j]$. El operador de valor absoluto, $|\cdot|$, se utiliza para eliminar cualquier signo negativo. Por ejemplo, $|2| = 2$ y $|-3| = 3$.

Una desventaja de este método es su complejidad computacional, ya que requiere comparar cada dama con todas las demás. Para ilustrar, con $n = 8$ damas, se necesitarían $7+6+5+4+3+2+1 = 28$ comparaciones. Sin embargo, existe una forma más eficiente de abordar este problema: marcando las diagonales ya ocupadas para evitar repetir comparaciones innecesarias. [2]

Una vez analizado un ejemplo y comprendido su funcionamiento, podemos definir el algoritmo mediante un diagrama de flujo que ilustre el proceso general de resolución. La Figura 1.12 muestra dicho diagrama de flujo. En un programa real, sería ne-

[2]Otro método más rápido para representar estos estados sería utilizando conjuntos de bits o *bitsets* [1].

cesario detallar más específicamente las acciones realizadas en cada paso del diagrama de flujo, pero dado que este libro tiene un carácter divulgativo, no es necesario entrar en esos detalles técnicos.

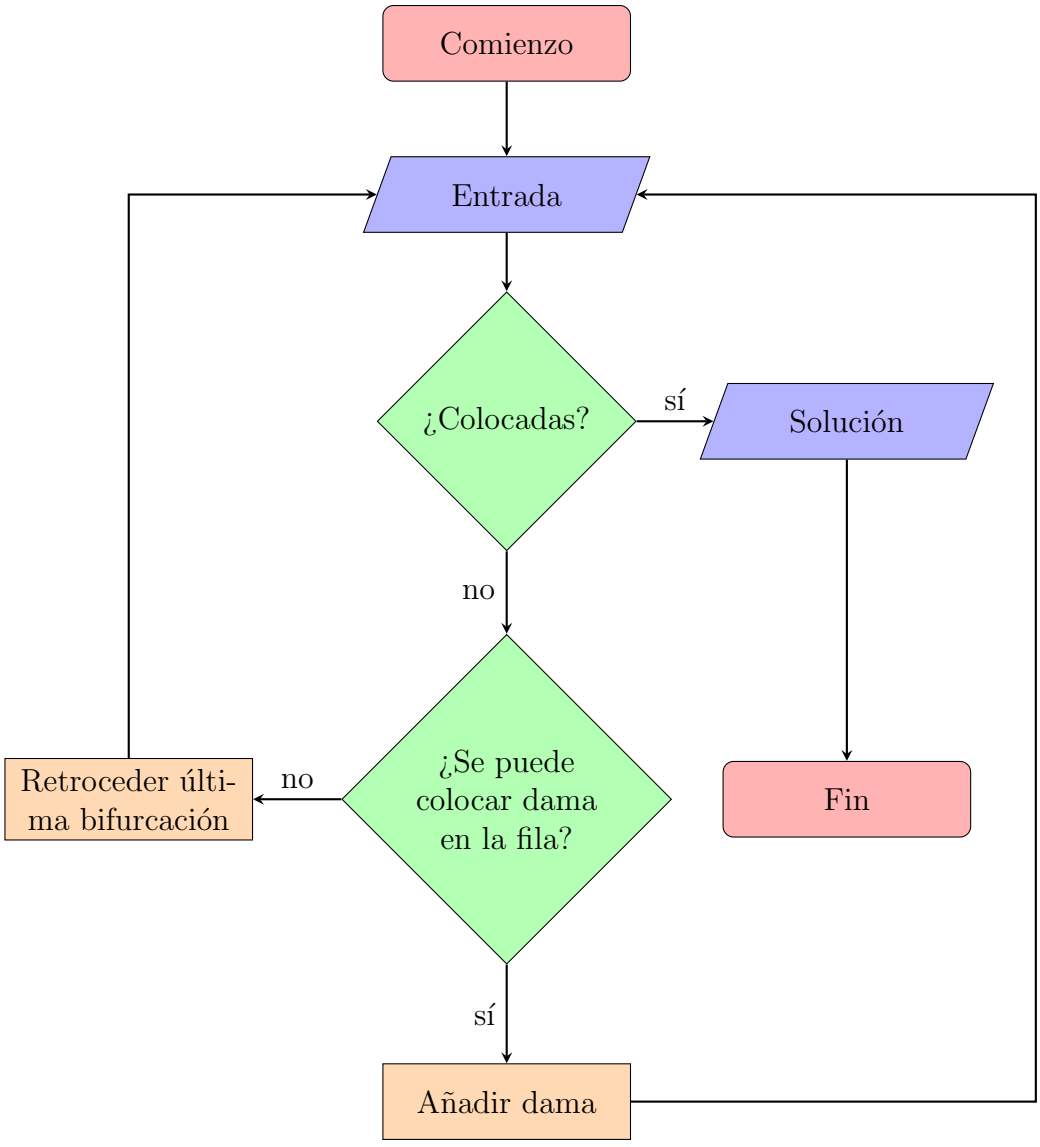


Figura 1.12: Diagrama de flujo de *Backtracking* aplicado a *n*-damas

A continuación, se describen detalladamente cada uno de los pasos del algoritmo representados en el diagrama de flujo.

1. Comienzo: Se inicia el algoritmo.
2. Entrada: Se introduce el problema con las damas ya colocadas en el tablero.
3. ¿Colocadas?: Se verifica si todas las damas han sido ubicadas en el tablero. Si la respuesta es afirmativa, el algoritmo ha encontrado una solución y procede a finalizar. De lo contrario, el algoritmo continúa con el siguiente paso, que consiste en determinar si es posible colocar una dama en la siguiente fila sin interferir con las damas ya ubicadas.
4. ¿Se puede colocar dama en la fila?: Se evalúa si es posible ubicar una dama en la fila actual sin que sea atacada por otras damas presentes en el tablero. Si la respuesta es afirmativa, el algoritmo sitúa una dama en la fila actual y avanza al siguiente paso. Si la respuesta es negativa, el algoritmo retrocede a la última bifurcación, es decir, al punto en el que tuvo que elegir entre diferentes opciones para ubicar las damas.
5. Retroceder última bifurcación: El algoritmo vuelve a la última bifurcación, eliminando la dama ubicada en esa instancia de la entrada.
6. Añadir dama: Se coloca una dama en una posición válida de la fila actual, y luego se retorna al paso 3 para verificar si todas las damas están ubicadas en el tablero.
7. Solución: Se produce una salida que contiene la disposición de las damas en el tablero, constituyendo una solución al

problema.

8. Fin: El algoritmo concluye su ejecución.

En resumen, el algoritmo de *Backtracking* aplicado al problema de las n-damas se basa en la exploración de las posibles soluciones mediante la colocación de damas en el tablero de forma iterativa. Si en algún momento no es posible colocar una dama en la fila actual, el algoritmo retrocede hasta la última bifurcación y prueba con una disposición diferente. Este proceso se repite hasta encontrar una solución en la que todas las damas estén colocadas en el tablero sin atacarse entre sí.

1.1.3. Complejidad algorítmica

El objetivo de este capítulo es utilizar el algoritmo que hemos discutido previamente como un pretexto para introducir el concepto de complejidad algorítmica. La complejidad algorítmica surge como una herramienta para medir y comparar la eficiencia de diferentes algoritmos de una manera que no dependa de un hardware de computadora específico. Para establecer esta medida independiente del hardware, recurrimos a una máquina teórica: la Máquina de Turing.

Antes de seguir adelante, proporcionaremos una definición de la Máquina de Turing. Tenga en cuenta que, dado el objetivo divulgativo de este libro, no se busca la rigurosidad de una definición matemática.

La Máquina de Turing, concebida por Alan Turing, consta de una cinta infinita en ambas direcciones. Esta cinta es analizada por una "cabeza" de lectura y escritura que puede moverse a la izquierda o a la derecha. Esta cabeza es capaz de leer y escribir diversos símbolos en la cinta. La máquina tiene un conjunto de

estados que, dependiendo del símbolo leído, le indican qué acción debe realizar.

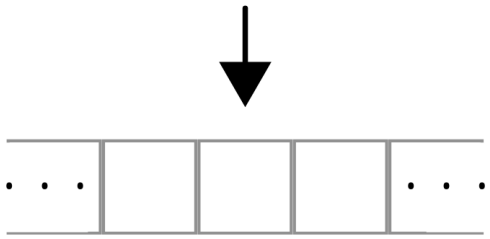


Figura 1.13: Máquina de Turing

Podemos ofrecer una definición más formal de la Máquina de Turing a través de la siguiente tupla:

$$MT = \{\sum, Q, q_0, F, \delta\}$$

Esta tupla consta de cinco elementos: el alfabeto, el conjunto de estados, el estado inicial, los estados finales y la función de transición, respectivamente.

$\sum$ o alfabeto: Representa el conjunto de símbolos que pueden ser utilizados en una Máquina de Turing específica. Esto incluye el símbolo de espacio en blanco, comúnmente representado por "." .

Q o conjunto de estados: Son los estados que la máquina puede adoptar. Estos estados, en conjunto con los símbolos, determinan las acciones que la cabeza debe realizar.

q_0 o estado inicial: Es el estado en el que comienza la Máquina de Turing. Es único.

F o estados finales: Son los estados que, cuando se alcanzan, permiten a la máquina finalizar su proceso. Puede haber varios.

δ o función de transición: Es una función que, dada una entrada que consta del estado actual y el símbolo leído por la cabeza, devuelve el próximo estado al que debe pasar la máquina, el símbolo que debe escribir la cabeza en su posición actual y la dirección hacia la cual debe moverse la cabeza.

Aunque la Máquina de Turing es una herramienta valiosa para entender la teoría de la computación, su naturaleza abstracta significa que rara vez se usa en la práctica para determinar la complejidad de un algoritmo.

Dicho esto, la Máquina de Turing posee un poder computacional impresionante y es capaz de resolver una vasta cantidad de problemas. Sin embargo, existen ciertos problemas que están más allá de sus capacidades, a menudo asociados con el llamado problema de la parada. Estos problemas son intrínsecamente irresolubles, no solo para la Máquina de Turing, sino para cualquier ordenador moderno, dado que todas las computadoras se basan en la estructura de la Máquina de Turing.[3]

En la práctica, para calcular la complejidad de un algoritmo, utilizamos una serie de principios y métodos bien establecidos. Comenzamos definiendo una función que represente la complejidad aproximada del algoritmo. Esta función toma como valor una variable que representa algún aspecto clave de la entrada del algoritmo.

Por ejemplo, en el problema de las n-reinas, esta variable representativa es n, que indica el tamaño del tablero. En una versión del problema en la que ya se han colocado algunas reinas,

[3]Para más información sobre este aspecto de la computación: [2]

sería más apropiado usar n menos el número de reinas colocadas; por lo tanto, la variable más adecuada en este caso sería el número de reinas que quedan por colocar. Si el algoritmo debe operar sobre una lista o un vector, esta variable representativa suele ser el tamaño de estos.

Para determinar el valor de salida de la función de complejidad, debemos analizar cuántas veces se ejecuta una operación representativa dada en función de la variable elegida. En el caso del problema de las n-reinas, la operación representativa sería el número de veces que se coloca una reina en el tablero, que en un nivel más bajo se traduce en una asignación en la lista de columnas.

En la mayoría de los casos, resulta imposible determinar con precisión el número exacto de veces que se ejecutará una operación en concreto. En tales casos, es útil considerar el peor escenario posible. De este modo, podemos obtener una cota superior para la complejidad, lo que significa que, para una entrada determinada, el algoritmo no tomará más tiempo que el tiempo correspondiente al peor de los casos.

Aun así, a menudo será imposible encontrar una medida exacta de la complejidad. Sin embargo, lo que nos interesa no es tanto la exactitud de la medida, sino la forma en que el número de operaciones crece en función de la variable representativa de la entrada. Este crecimiento se agrupa en diferentes categorías o "familias" de funciones, de acuerdo con su tasa de crecimiento. A este concepto se le conoce como notación *big-O*.

Las familias más comunes, ordenadas de la mejor a la peor en términos de crecimiento, son las siguientes (usaremos n para representar la variable representativa):

$O(1)$: Independientemente del valor de n, el tiempo de eje-

cución del algoritmo permanece constante. Este es el tipo de complejidad que se asocia con las operaciones básicas (sumas, restas, multiplicaciones, etc.) que puede realizar una computadora, como sumar dos números o realizar asignaciones.

$O(\log n)$: Esta es una tasa de crecimiento muy favorable, ya que es inversamente proporcional al crecimiento exponencial. Ejemplos de algoritmos que presentan este tipo de crecimiento son la búsqueda binaria y la búsqueda en un árbol binario de búsqueda.

$O(n)$: En este caso, el tiempo de ejecución del algoritmo crece linealmente con n. Aunque es más lento que $O(\log n)$, sigue siendo aceptable para la mayoría de las aplicaciones. Ejemplos de este tipo de crecimiento son la búsqueda de un elemento en una lista o vector no ordenado y la elevación de un número a una potencia.

$O(n \log n)$: Este tipo de crecimiento es ligeramente más rápido que $O(n)$, y es común en los algoritmos relacionados con la ordenación de elementos. Ejemplos de algoritmos con este tipo de crecimiento son Heapsort y Quicksort (en el caso promedio).

$O(n^a)$ donde $a > 1$: Este tipo de crecimiento incluye todos los algoritmos cuyo tiempo de ejecución crece como un polinomio de n, con excepción de los casos lineales ya mencionados. Aunque el valor de a puede ser alto, este tipo de crecimiento sigue siendo preferible a los tipos de crecimiento que mencionaremos a continuación. Ejemplos de este tipo de crecimiento son la multiplicación de matrices $O(n^3)$ y los algoritmos que utilizan dos bucles anidados $O(n^2)$.

$O(a^n)$ donde $a > 1$: Este es el inicio de los algoritmos con tiempo de ejecución no polinómico, y son la pesadilla de cualquier programador. Aunque n sea pequeño, la computación re-

querida para resolver el problema puede ser inmensa. Ejemplos de problemas con este tipo de crecimiento son la satisfacibilidad booleana y la partición de un conjunto en dos subconjuntos de igual suma.

$O(n!)$, $O(n^n)$: Estos son los casos de crecimiento más rápido y se encuentran entre los más difíciles de manejar. Cuando n es un número grande, tanto $n!$ como n^n crecen más rápido que cualquier exponencial, sin importar el valor de a en la exponencial. Ejemplos de problemas con este tipo de crecimiento son el problema de las n-reinas y el problema del viajero.

Una vez definidas las familias, podemos usar unas reglas prácticas y simples para calcular la complejidad de un problema específico. En primer lugar, definimos las operaciones básicas, que son las diferentes operaciones aritméticas, lógicas, asignaciones y similares. Todas estas operaciones tienen una complejidad $O(1)$, es decir, son constantes, y constituyen los bloques de construcción más básicos de cualquier algoritmo.

La complejidad lineal aparece cuando se ejecutan una o varias de las operaciones básicas definidas anteriormente en un bucle que itera un número de veces que crece linealmente con n. Por ejemplo, si se asigna el valor 1 a todos los elementos de una lista de tamaño n, la complejidad será $O(n)$. Para alcanzar una complejidad de $O(n^2)$, necesitas tener una estructura de bucle anidado en la que ambos bucles estén iterando proporcionalmente a n. Un ejemplo común de esto es cuando se recorren todos los pares de elementos en una lista. En general, si tienes a bucles anidados que iteran proporcionalmente a n, obtendrás una complejidad de $O(n^a)$.

Por otro lado, la complejidad $O(\log n)$ a menudo se logra mediante un enfoque de "divide y vencerás", en el que el problema

se divide en mitades sucesivas hasta llegar a un caso base. La búsqueda binaria es un ejemplo común de esto.

Para las complejidades exponenciales $O(a^n)$, a menudo surgen en algoritmos que exploran todas las posibles combinaciones o permutaciones de un conjunto. El problema del viajante de comercio es un ejemplo notorio.

Finalmente, el cálculo de la complejidad de un algoritmo puede implicar sumar las complejidades de las diferentes partes del algoritmo. Sin embargo, cuando se suman complejidades, solo la de crecimiento más rápido es relevante. Por ejemplo, si una parte de tu algoritmo tiene una complejidad de $O(n^2)$ y otra parte tiene una complejidad de $O(n)$, la complejidad total será $O(n^2)$, ya que n^2 crece más rápido que n. Este principio se aplica a todas las formas de combinar complejidades.

Es importante recordar que estas son reglas generales y que puede haber casos en los que estas reglas no se apliquen exactamente. Sin embargo, son una buena guía para entender cómo escala un algoritmo a medida que el tamaño de la entrada aumenta. Una vez conocidos estos conceptos podemos comenzar a analizar la complejidad de las n-damas.

En la exploración del problema de las n-damas, hemos considerado diversas estrategias para su solución, basándonos en diferentes representaciones del estado. De igual forma, analizaremos la complejidad de estas soluciones siguiendo el mismo criterio.

Consideremos un tablero de 4x4 en el que necesitamos ubicar 4 damas. Si generamos todas las combinaciones posibles sin importar el orden, el cálculo sería el siguiente:

$$\frac{16!}{4!(16-4)!} = \frac{16 \times 15 \times 14 \times 13}{4!} = 1820$$

Para llegar a este resultado, razonamos de la siguiente manera: hay 16 casillas disponibles para la primera dama, 15 para la segunda, 14 para la tercera y 13 para la última. Como el orden no importa, debemos dividir entre 4!, según los principios de combinatoria. Este resultado se puede generalizar para un tablero de n x n con la siguiente fórmula:

$$\alpha(n) = \frac{(n^2)!}{n!(n^2-n)!} = \frac{n^2 * (n^2-1) * ... * (n^2-n+1)}{n!}$$

Así, $\alpha(n)$ indica el número de formas de situar n damas en un tablero de n x n.

En secciones anteriores, se discutieron representaciones más eficientes de una solución en lugar de simplemente usar coordenadas. Por ejemplo, podríamos asignar un número entre 1 y n (ambos inclusive) a cada columna, lo cual representa la fila en la que se sitúa la dama de dicha columna. Esta representación reduciría significativamente el número de opciones posibles.

Para un tablero de 4x4 en el que debemos colocar 4 damas, cada columna ofrece cuatro opciones, lo cual se repite cuatro veces. Este número es considerablemente alto para un tablero de 4x4. La fórmula general en este caso sería simplemente:

$$\alpha(n) = n^n$$

Estos dos métodos presentan algunas de las peores complejidades que hemos analizado hasta ahora.

Si recordamos, se puede optimizar la búsqueda descartando algunas posibles soluciones a medida que se generan. Por ejemplo, es evidente que cualquier configuración que coloque dos damas en la misma fila no es una solución válida, por lo que

podemos descartarla inmediatamente y evitar gastar recursos innecesarios. Sin embargo, calcular la complejidad en este caso es un desafío, ya que necesitamos una aproximación; sería imposible encontrar una fórmula que proporcione la complejidad exacta para cualquier n.

Podemos ir descartando las filas en las que ya hemos colocado una dama, reduciendo en cada paso el número de filas a comprobar en la columna siguiente. Para las diagonales, la situación es más complicada. En el peor de los casos, podemos descartar una casilla en la columna siguiente, pero es posible que no podamos descartar ninguna en las columnas posteriores. En el peor de los casos, por tanto, la fórmula para este método sería simplemente el factorial de n. En la primera columna, hay n opciones; en la segunda, $n-1$; y así sucesivamente. La fórmula por tanto sería el famoso factorial.

$$\beta(n) = n!$$

Este es el rendimiento estándar del algoritmo de *backtracking* para las n-damas. Se pueden mejorar aún más los resultados utilizando heurísticas o representaciones más eficientes de las diagonales, pero esto está fuera del alcance de este libro.

En conclusión, tras haber analizado la complejidad del problema de las n-damas, podemos afirmar que aún no se ha encontrado ningún algoritmo capaz de resolverlo en un tiempo razonable para valores grandes de n. Es decir, no se ha descubierto ningún algoritmo polinómico determinista que lo resuelva. Por tanto, se clasifica como un problema NP-difícil.

En general, los problemas pueden dividirse en dos tipos según su dificultad: fáciles, si pueden resolverse en tiempo polinómico determinista; o difíciles, si no se puede encontrar un algoritmo

polinómico determinista que los resuelva. La gran pregunta sin resolver es si ambos conjuntos son idénticos y, por tanto, existe un algoritmo polinómico determinista para resolver cualquier problema computable. Esa es justamente la cuestión

1.1.4. El asombroso crecimiento exponencial

El crecimiento exponencial a menudo se retrata como una fuerza devastadora e incontrolable. ¿Pero es realmente así de malo? Para apreciar la velocidad vertiginosa con la que puede crecer una función exponencial, recurramos a la antigua leyenda del origen del ajedrez.

Según la narración, un monarca de un lejano reino, abrumado por el tedio, convocó un concurso en el que sus súbditos podrían presentar un juego para su entretenimiento. El creador del juego que más deleitara al rey recibiría cualquier recompensa que deseara. Entre las numerosas propuestas, una destacó por encima del resto: el ajedrez. El inventor del juego, un hombre de humilde condición, pidió al rey que colocase un grano de arroz en la primera casilla del tablero de ajedrez, dos granos en la segunda, cuatro en la tercera, y así sucesivamente, hasta llegar a la sexagésimo cuarta casilla.

El rey, inicialmente, se rió de la modesta solicitud, pero cuando sus consejeros calcularon la cantidad de arroz necesaria, se quedaron atónitos. No había suficiente arroz en todo el reino, ni siquiera en varias generaciones, para cumplir con la petición del inventor. Cuando el rey comprendió la magnitud de la solicitud, no pudo más que admirar la astucia del humilde creador del ajedrez.

Después de escuchar esta historia, es natural preguntarse cuántos granos de arroz se requerirían. Para responder a es-

ta pregunta, primero consideramos cuántos granos hay en cada casilla. En la primera casilla hay un grano, es decir, 2^0, en la segunda hay dos granos o 2^1 y en la tercera hay 2^2 o cuatro granos. Continuando con este patrón, la cantidad total de granos se puede calcular matemáticamente utilizando un sumatorio (permite sumar una serie de números que siguen un patrón):

$$\sum_{k=0}^{63} 2^k = 2^{64} - 1 = 18446744073709551615 \text{ granos}$$

La fórmula de la izquierda suma todos los granos de cada casilla, la del centro emplea un "truco"para simplificar la suma con exponenciales, y en la parte derecha se muestra el número total de granos necesarios. La cifra es asombrosa. Para ponerla en perspectiva, la producción mundial actual de arroz es de aproximadamente 743 millones de toneladas, y cada tonelada puede contener unos 50 millones de granos. Por lo tanto, la producción mundial total sería de unos 37150 billones de granos.

Comparando esta cifra con el número de granos requeridos para el tablero de ajedrez, se necesitarían casi 500 años de producción para cubrir la demanda, como se muestra en el siguiente cálculo:

$$\frac{18446744073709551615}{37150 * 10^{12}} = 496,55 \text{ años}$$

Así, como se puede observar, el crecimiento exponencial es verdaderamente rápido y puede llegar a ser abrumador, tal como se ha expuesto en las secciones previas de este capítulo. Por tanto, es crucial no subestimar su impacto.

1.1.5. El enigma de ¿P=NP?

El problema ¿P=NP? busca determinar si el conjunto P, referente a los algoritmos que pueden resolverse en tiempo polinomial determinista, es equivalente al conjunto NP, que comprende los algoritmos resueltos en tiempo polinómico no determinista. Una observación importante a tener en cuenta es que P es un subconjunto de NP, ya que los algoritmos deterministas están incluidos en la categoría no determinista. Es decir, un algoritmo P también será NP, pero un algoritmo NP no necesariamente pertenecerá a P. Dentro de NP existe un subconjunto denominado problemas NP-completos para los que, hasta la fecha, no se ha encontrado un algoritmo que pertenezca a P. La búsqueda de una solución al problema ¿P=NP? se centra en demostrar que existe un algoritmo P para estos problemas.

Desde un punto de vista teórico basado en la Máquina de Turing, se puede definir el problema de la siguiente manera:

"P=NP si existe una Máquina de Turing determinista con una cota superior polinómica temporal que pueda transformar una Máquina de Turing no determinista con cota superior temporal polinómica en una Máquina de Turing determinista con la misma cota superior."

Este enunciado puede resultar complejo al principio. En el caso de las máquinas de Turing no deterministas, un estado particular y la lectura de un símbolo pueden desencadenar una o más acciones en lugar de una sola, como sucede con las máquinas deterministas. Esta característica, aplicada a los ordenadores, implica que deben ejecutarse diferentes acciones, sin saber cuál de ellas llevará a la solución, provocando la temida complejidad exponencial.

En resumen, resolver el problema P=NP consistiría en en-

contrar un algoritmo polinómico determinista que pueda transformar cualquier algoritmo polinómico no determinista en uno polinómico, o demostrar que tal algoritmo no existe. Cualquiera de estas soluciones le concedería al descubridor un premio de un millón de dólares.

Para simplificar, se ha demostrado que existe un problema no polinómico, la satisfacibilidad booleana, al que se puede convertir cualquier algoritmo polinómico. Este problema consiste en determinar si, dada una expresión lógica, existe una entrada que haga que la fórmula sea verdadera. La demostración de este hecho excede el alcance de este libro, pero una intuición de por qué esto podría ser así está relacionada con el hecho de que los ordenadores son, en última instancia, una serie de puertas lógicas. Este fue el primer problema descubierto de los denominados problemas NP-completos, a los que se pueden convertir todos los demás problemas NP mediante un algoritmo polinómico.

Los problemas NP-completos son bastante comunes en la vida cotidiana, y muchos de ellos están relacionados con los grafos, que se discutirán en el próximo apartado. También se encuentran frecuentemente en problemas de optimización, entre otros.

Es importante mencionar que, en la práctica, se suelen utilizar algoritmos aproximados para resolver estos tipos de problemas, con el objetivo de eludir la complejidad exponencial. Estos algoritmos permiten encontrar una solución que, aunque no sea la óptima, puede ser válida según los criterios definidos. En otras palabras, si se utiliza un algoritmo exponencial para entradas pequeñas, podría ser posible encontrar la mejor solución. Sin embargo, en el caso de entradas muy grandes, se recurrirá a un algoritmo aproximado, que en ocasiones no proporcionará la mejor respuesta.

1.2. El tour del caballo

1.2.1. El caballo saltarín

En esta sección, nos adentraremos en la exploración de uno de los problemas más intrigantes originados del fascinante mundo del ajedrez. El protagonista de este enigma es el caballo, una pieza cuyo singular tipo de movimiento lo distingue de todas las demás piezas en el tablero de ajedrez. Esta peculiaridad convierte este problema en un retador pasatiempo intelectual. Sin más preámbulos, la definición del problema:

"Colocamos el caballo en una casilla inicial en un tablero de dimensiones $m \times n$. El objetivo es encontrar un camino que recorra cada casilla del tablero exactamente una vez."

En términos más sencillos, el desafío consiste en recorrer todas las casillas del tablero, sin repetir ninguna, utilizando el característico movimiento en "L" del caballo. Al enfrentarse por primera vez a este desafío, es probable que uno se dé cuenta de que no es nada trivial, y puede requerir varios intentos para lograrlo incluso en el clásico tablero de 8×8. Es importante destacar que el tablero no necesariamente tiene que ser cuadrado; puede ser cualquier rectángulo.

Este problema ha capturado la atención de notables matemáticos a lo largo de la historia. Entre ellos destaca Leonhard Euler, considerado uno de los matemáticos más preeminentes de todos los tiempos. Euler no solo logró resolver este problema para un tablero de 8×8, sino que además construyó un semicuadrado mágico en el que cada fila y columna suman 260, las mitades de las filas y columnas suman 130, y cada número representa el orden del movimiento con el cual el caballo visitó esa casilla en particular.

La elegancia de esta solución se puede apreciar en la Figura 1.14.

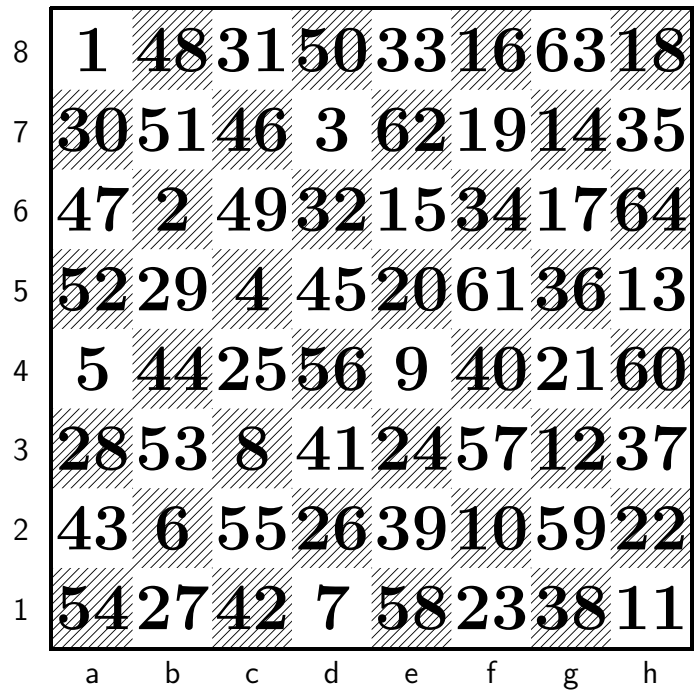
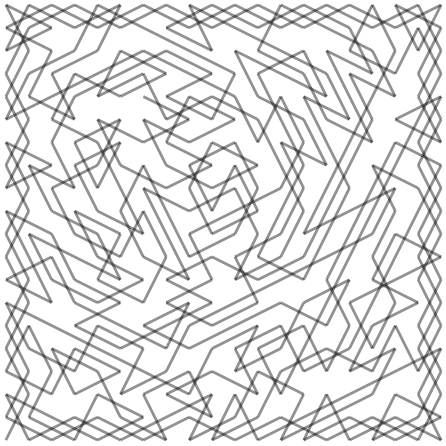
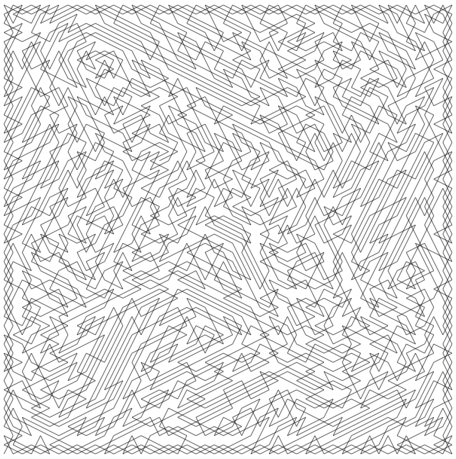


Figura 1.14: Cuadrado mágico de Euler

Más allá de esta intrigante construcción, este problema también puede ser utilizado como un medio para generar arte computacional. A continuación, se presenta una serie de composiciones artísticas inspiradas en los movimientos de un caballo:

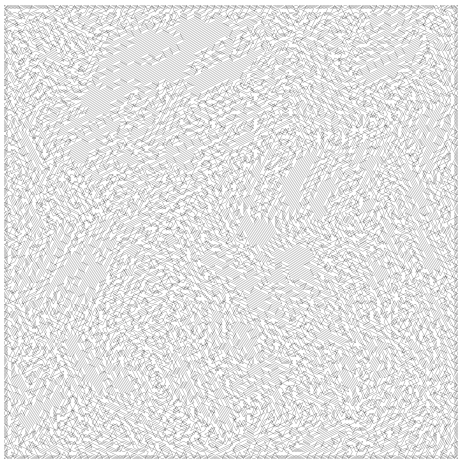


(a) Composición basada en un tablero de 20x20



(b) Composición basada en un tablero de 50x50

(c) Composición basada en un tablero de 64x64



(d) Composición basada en un tablero de 130x130

Figura 1.15: Composiciones artísticas basadas en el Tour del Caballo

Las obras representadas corresponden a tableros de diferentes tamaños: 20x20 (esquina superior izquierda), 50x50 (esquina superior derecha), 64x64 (esquina inferior izquierda) y 130x130

(esquina inferior derecha). Cada una de estas composiciones ilustra la rica variedad de patrones que pueden surgir de la sencilla pero desafiante tarea de recorrer un tablero de ajedrez con un caballo.

Para relacionar más al lector con el problema procederé a mostrar la solución de un tablero 8x8 mediante flechas:

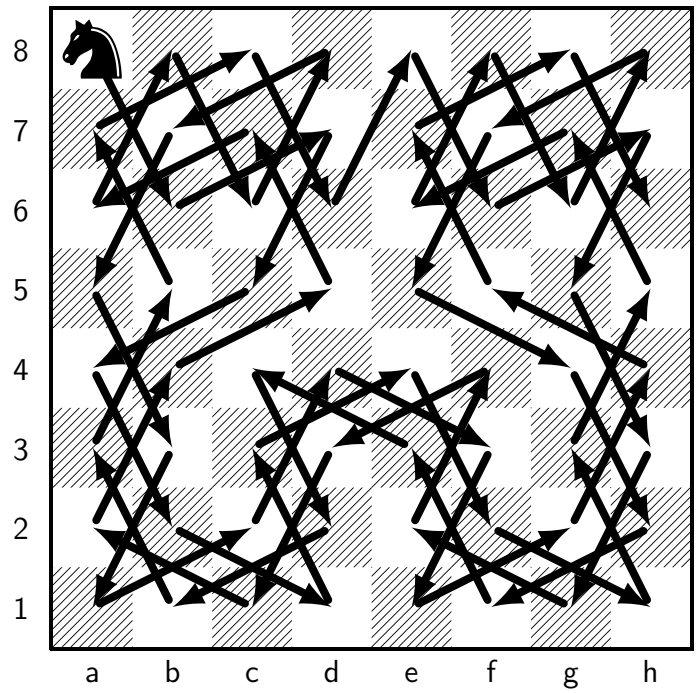


Figura 1.16: Solución al tablero de 8x8

Una observación crucial en la resolución de este problema es que las primeras casillas que visita el caballo tienden a estar ubicadas en las regiones más alejadas del centro. Esta técnica desempeña un papel fundamental en la solución del desafío planteado. Por si no se ha dado cuenta se trata de la solución al

cuadrado mágico de Euler mostrado anteriormente.

El tour del caballo en este contexto está estrechamente vinculado a uno de los problemas más relevantes en el campo de la informática: el conocido problema de encontrar un camino hamiltoniano, el cual se clasifica como un problema NP.

1.2.2. Caminos hamiltonianos y grafos

El problema del camino hamiltoniano se define de la siguiente manera:

"Dado un grafo, se busca encontrar un camino que visite cada uno de los vértices exactamente una vez". (Un camino es la secuencia en la que se recorren los vértices del grafo).

Esta definición presenta una notable similitud con el concepto del tour del caballo. Pero ¿qué son exactamente un grafo y sus vértices?

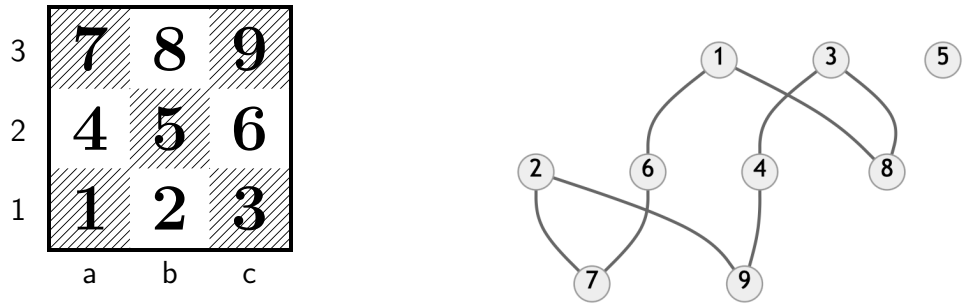
Un grafo es una estructura de datos que proporciona una gran flexibilidad para representar relaciones entre diferentes elementos. Consiste en un conjunto de vértices (también conocidos como nodos) y aristas (o conexiones). Los vértices pueden ser cualquier tipo de objeto, desde números hasta palabras y objetos más complejos. Las aristas, por otro lado, conectan dos vértices si existe alguna relación entre ellos. En el caso de que las aristas tengan un valor numérico que represente algún aspecto de la relación entre los elementos conectados, se denomina grafo valuado.

Volviendo al tema del camino hamiltoniano, al modificar ligeramente las condiciones del problema, surgen dos problemas interesantes adicionales. Si agregamos la restricción de que, al llegar al último vértice, se debe regresar al vértice inicial, se obtiene lo que se conoce como ciclo hamiltoniano. Por otro la-

do, si utilizamos un grafo valuado (cada arista tiene asociado un número que expresa el coste de usar esa arista) y buscamos el camino que minimice la suma de los valores de las aristas recorridas, nos enfrentamos al problema del viajante.

Pero ¿cómo se relacionan los grafos y los diferentes problemas hamiltonianos con el tour del caballo?

Aunque no sea evidente a primera vista, el tablero de ajedrez puede considerarse como un grafo implícito, donde los vértices y las aristas están presentes de manera disimulada. En este caso, los vértices representarían las casillas del tablero, mientras que las aristas modelarían la relación entre dos casillas. Por lo tanto, habría una arista entre dos casillas específicas si es posible moverse de una a otra mediante un único movimiento del caballo. A continuación, se muestra un ejemplo de un tablero de 3×3 y su correspondiente transformación en un grafo que ilustra el anterior concepto.



(a) Tablero de ajedrez (b) Grafo

Figura 1.17: Tablero de ajedrez a grafo

En primer lugar, se numeran las casillas del tablero asignándole a cada uno de ellas un número natural (esto es una técnica

muy usual a la hora de trabajar con grafos) y se dibuja el grafo.

En esta representación, se cumple que existe una arista entre dos casillas si es posible moverse entre ellas en un solo movimiento. Por ejemplo, desde la casilla 9 se puede llegar a las casillas 4 y 2 mediante un movimiento de caballo, mientras que la casilla 5 no tiene ninguna arista, ya que no es posible llegar a ella desde ninguna otra casilla.

Ahora bien, ¿cómo es capaz un ordenador de entender un grafo y cómo se representa para que sea comprendido por una máquina?

1.2.3. Representación de grafos

La representación de un grafo es un elemento crucial en la eficacia y eficiencia de los algoritmos que los utilizan. Existen múltiples representaciones, cada una con sus ventajas y desventajas únicas. Antes de adentrarnos en la explicación de estas representaciones, es esencial reconocer que los grafos pueden ser de dos tipos: dirigidos y no dirigidos.

En los grafos dirigidos, las aristas funcionan como "flechas", es decir, la relación existe sólo desde un vértice hacia otro, y puede darse el caso de que no exista una relación en sentido contrario. En contraste, en los grafos no dirigidos, las aristas indican que existe una relación bidireccional entre los dos vértices, tal como se ilustró con el grafo que discutimos anteriormente.

Entre las diversas representaciones de grafos, la matriz de adyacencia es una opción notable. Como su nombre lo sugiere, es una matriz en la que la fila i y la columna j representan la relación desde el vértice i hacia el vértice j.

Si estamos trabajando con un grafo no ponderado, es común usar un 1 para denotar la existencia de una arista desde i a j,

y un 0 en caso contrario. Sin embargo, si el grafo es ponderado, colocaremos el valor de la arista que conecta los vértices i y j en la respectiva posición de la matriz. Si no existe tal conexión, a menudo se usa un número muy grande para indicar que la arista es inutilizable, aunque también existen otras alternativas.

Tomemos, por ejemplo, la matriz de adyacencia del grafo de un tablero 3x3, que se representaría de la siguiente manera:

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Figura 1.18: Representación usando una matriz de adyacencia

Es relevante notar que la matriz de adyacencia es simétrica. Es decir, si intercambiamos filas por columnas, obtenemos la misma matriz. Esta característica siempre está presente en las matrices de adyacencia de grafos no dirigidos, debido a la naturaleza bidireccional de sus relaciones. Puede parecer que existe una redundancia de información, y en efecto, así es. Por lo tanto, realmente sólo necesitamos la mitad de la matriz, específicamente, la sección por encima o debajo de la diagonal que se extiende desde la esquina superior izquierda hasta la esquina inferior derecha, para tener todos los datos necesarios del grafo. Además, es importante señalar que tanto la columna como la fila del 5 están vacías, ya que es imposible llegar a esa casilla o partir de

ella hacia otra.

La principal ventaja de esta representación es que permite consultar en tiempo constante si se puede pasar de un vértice a otro. Esto es particularmente útil en la implementación de dos algoritmos ampliamente utilizados en grafos: Warshall y Floyd. Warshall es útil para determinar si existe un camino entre un vértice y cualquier otro, y también proporciona dicho camino. Por otro lado, Floyd opera sobre grafos ponderados y proporciona el camino más corto entre dos vértices, si existe, y el valor de dicho camino.

Sin embargo, la matriz de adyacencia también almacena relaciones inexistentes, es decir, los 0s en la matriz. Además, estos 0s superan en número a los 1s, lo que indica que el número de relaciones es pequeño en comparación con el total de posibles relaciones. Esto nos lleva a preguntar: ¿no debería existir una forma de almacenar únicamente las relaciones existentes en el grafo y evitar almacenar los 0s?

Efectivamente, la respuesta es afirmativa. Para este propósito, se puede utilizar la lista de adyacencia. Su principal premisa es almacenar sólo las relaciones entre los diferentes vértices, lo que permite un gran ahorro de memoria e incluso, en ocasiones, de tiempo de cómputo. En una lista de adyacencia, cada vértice tiene asignada una lista en la que figuran los vértices con los que está relacionado. La lista de adyacencia para el grafo anterior se presentaría de la siguiente manera:

$$
\begin{aligned}
1 &\rightarrow \begin{bmatrix} 6 & 8 \end{bmatrix} \\
2 &\rightarrow \begin{bmatrix} 7 & 9 \end{bmatrix} \\
3 &\rightarrow \begin{bmatrix} 4 & 8 \end{bmatrix} \\
4 &\rightarrow \begin{bmatrix} 3 & 9 \end{bmatrix} \\
5 &\rightarrow \\
6 &\rightarrow \begin{bmatrix} 1 & 7 \end{bmatrix} \\
7 &\rightarrow \begin{bmatrix} 2 & 6 \end{bmatrix} \\
8 &\rightarrow \begin{bmatrix} 1 & 3 \end{bmatrix} \\
9 &\rightarrow \begin{bmatrix} 2 & 4 \end{bmatrix}
\end{aligned}
$$

Figura 1.19: Representación usando una lista de adyacencia

Como puede ver, se ha logrado una significativa reducción en la cantidad de información almacenada utilizando la lista de adyacencia. En este caso específico del tablero de ajedrez, somos afortunados ya que una pieza de caballo solo puede moverse a un máximo de dos casillas. Sin embargo, a medida que el tamaño del tablero aumenta, la eficiencia en términos de memoria de la lista de adyacencia sobre la matriz de adyacencia también se incrementa. Esto se debe a que, como mucho, un caballo podrá moverse a 8 casillas diferentes desde una casilla específica, por lo que la lista tendrá a lo sumo ese número de elementos.

La principal desventaja de este método de representación es que, si necesitamos verificar la existencia de una arista entre dos vértices, debemos buscar en la lista del primer vértice para ver

si el segundo vértice aparece o no. Es importante mencionar que esta representación también puede ser usada para grafos ponderados. En ese caso, los elementos de las listas son reemplazados por tuplas de dos valores: el vértice y el peso de la arista. Esta representación puede no ser adecuada para algoritmos como Floyd o Warshall [4], pero es más eficiente para algoritmos que buscan caminos o ciclos hamiltonianos.

Por último, vale la pena mencionar que existe otra forma de representar un grafo, denominada lista de aristas. Como su nombre lo indica, esta lista contiene todas las aristas, cada una expresada como una tupla de dos enteros, donde el primer entero representa el vértice origen y el segundo el vértice destino. Sin embargo, este método de representación se utiliza raramente.

Habiendo examinado las diferentes formas de representación de un grafo, estamos listos para abordar el problema del camino hamiltoniano. Para resolver este desafío, debemos recurrir a una estrategia bien conocida: el *backtracking*.

1.2.4. *Backtracking* vs. el tour del caballo

Al igual que en el problema de las n-damas, nos enfrentamos a una interrogante: ¿cómo podemos prever si una decisión que tomamos nos llevará inexorablemente a un callejón sin salida o a una solución válida?

Necesitaremos la habilidad de retroceder y explorar otros caminos para encontrar una solución válida, si es que existe. Por esta razón, el *backtracking* parece la opción más adecuada de manera intuitiva. Sin embargo, como se mencionó anteriormen-

[4] El algoritmo de Floyd encuentra la ruta más corta entre todos los pares de nodos en un grafo. Por otro lado, el algoritmo de Warshall determina si existe un camino entre todos los pares de nodos.

te, estamos lidiando con un problema NP. Afortunadamente, el "tour del caballo" es una versión específica del problema donde podemos mejorar notablemente la eficiencia del algoritmo utilizando una heurística [5] conocida como la regla de Warnsdorff. Esta regla reduce la complejidad del algoritmo, permitiendo una solución en tiempo lineal.

Una heurística es una guía que nos orienta hacia los estados que son más propensos a conducir a una solución válida. La regla de Warnsdorff nos aconseja dirigirnos hacia el vértice de menor grado. Aquí surge una nueva pregunta: ¿qué es el grado de un vértice?

El grado de un vértice se define como el número de vértices a los que se puede llegar desde dicho vértice con un único movimiento. Es similar al número de aristas de un vértice, pero no se cuentan aquellas aristas que llevan a un vértice ya visitado. Si el grado de un vértice es 0, significa que no podremos realizar ningún movimiento más una vez que lleguemos a ese vértice. Por lo tanto, solo deberíamos dirigirnos a un vértice de grado 0 si este es el último movimiento que pretendemos hacer.

Una vez que hemos aclarado este término, podemos proceder a resolver el problema en un tablero de 5x5. Supongamos que la casilla inicial del caballo corresponde a1, es decir, un caballo situado en la esquina inferior izquierda. Usaremos una matriz de las mismas dimensiones que el tablero para registrar las casillas que ya hemos visitado y avanzar en la resolución del problema. Esta matriz será de enteros, y el valor de una casilla será el número de orden en el que fue visitada. Al comenzar, todos los valores de la matriz serán 0, excepto la casilla inicial, que tendrá

[5] Una heurística es una regla práctica o método aproximado que ayuda a simplificar y resolver problemas, a menudo permitiendo encontrar soluciones suficientemente buenas aunque no siempre óptimas.

el valor 1. El cero indica que esa casilla no ha sido visitada de momento.

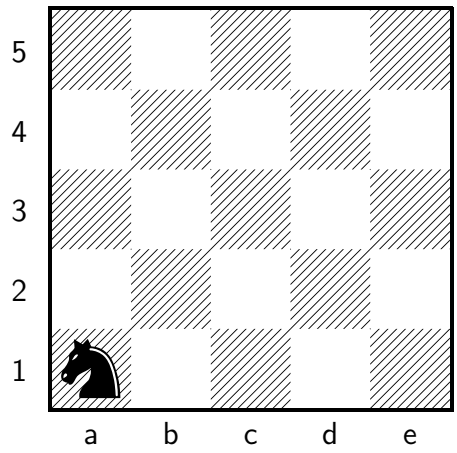
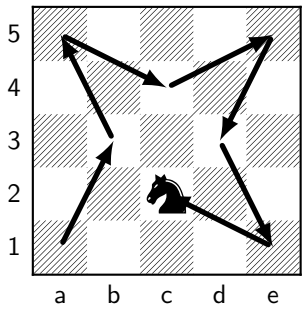


Figura 1.20: Posición inicial del tour del caballo usando vuelta atrás

En la primera vuelta alrededor del tablero, el caballo ha pasado por las cuatro esquinas, ya que estas siempre tienen grado 1, al llevar solo a dos casillas y una de ellas ya estará visitada (por la que ha llegado a la esquina). En el segundo movimiento, al moverse desde la casilla b3 a la a5, se aplica claramente la regla al preferir ir a la esquina a5 con grado 1 en lugar de ir a las casillas c5 o d4 con grado 3. La matriz de visitados quedaría de la siguiente manera, junto al tablero que indica los movimientos del caballo:



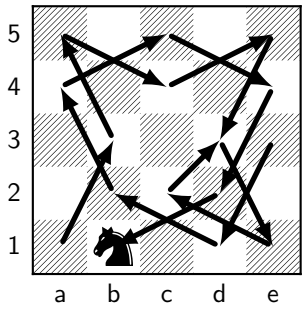
(a) Tablero de ajedrez con los movimientos de la primera fase

$$\begin{bmatrix} 3 & 0 & 0 & 0 & 5 \\ 0 & 0 & 4 & 0 & 0 \\ 0 & 2 & 0 & 6 & 0 \\ 0 & 0 & 8 & 0 & 0 \\ 1 & 0 & 0 & 0 & 7 \end{bmatrix}$$

(b) Matriz con los movimientos de la primera fase

Figura 1.21: Primera fase del tour del caballo usando vuelta atrás

Siguiendo la misma heurística, el caballo realiza una segunda vuelta por el tablero, obteniendo el siguiente tablero y estado:



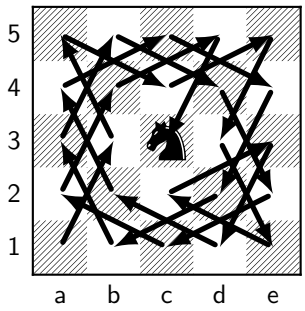
(a) Tablero de ajedrez con los movimientos de la segunda fase

$$\begin{bmatrix} 3 & 0 & 13 & 0 & 5 \\ 12 & 0 & 4 & 0 & 14 \\ 0 & 2 & 0 & 6 & 9 \\ 0 & 11 & 8 & 15 & 0 \\ 1 & 16 & 0 & 10 & 7 \end{bmatrix}$$

(b) Matriz con los movimientos de la segunda fase

Figura 1.22: Segunda fase del tour del caballo usando vuelta atrás

En la última vuelta, el tablero se completaría y se obtendría el siguiente tablero y estado final:



(a) Tablero de ajedrez con los movimientos de la tercera fase

$$\begin{bmatrix} 3 & 18 & 13 & 24 & 5 \\ 12 & 23 & 4 & 19 & 14 \\ 17 & 2 & 25 & 6 & 9 \\ 22 & 11 & 8 & 15 & 20 \\ 1 & 16 & 21 & 10 & 7 \end{bmatrix}$$

(b) Matriz con los movimientos de la tercera fase

Figura 1.23: Tercera fase del tour del caballo usando vuelta atrás

Como resultado interesante, se observa que la última casilla visitada es justamente la central, que en el tablero de 5x5 es la única casilla con grado 8 al inicio del problema. Esto demuestra que el caballo, siguiendo la heurística utilizada, logra alcanzar y visitar todas las casillas del tablero, finalizando en la única casilla con grado 8 al comienzo.

Para terminar, a continuación se presenta el diagrama de flujo que indica el funcionamiento del algoritmo usado para resolver el tour del caballo.

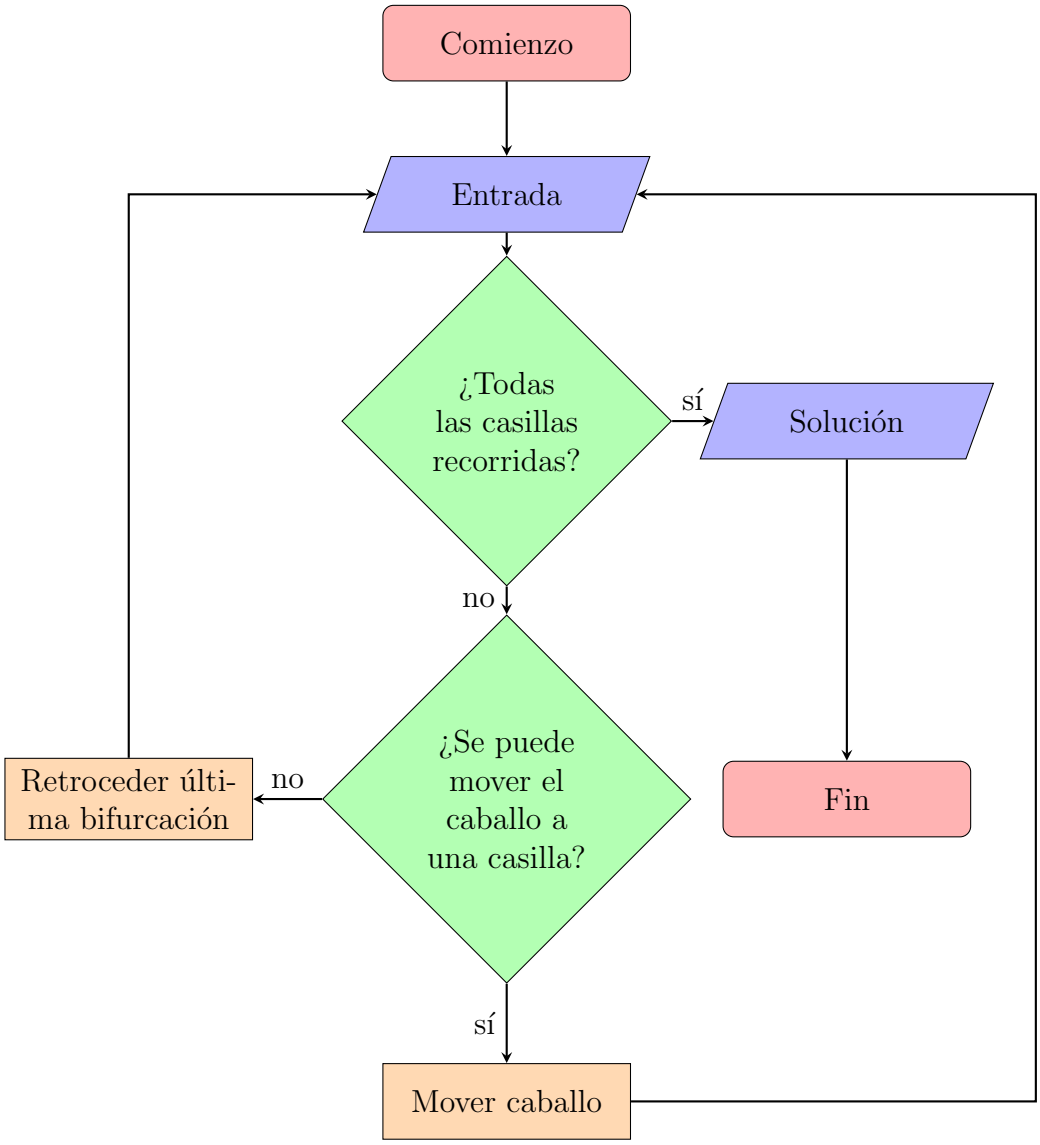


Figura 1.24: Diagrama de flujo de *Backtracking* aplicado al tour del caballo

A continuación, se describen detalladamente cada uno de los pasos del algoritmo representados en el diagrama de flujo.

1. Comienzo: Se inicia el algoritmo.

2. Entrada: Se introduce el problema con la casilla inicial del caballo y el tamaño del tablero. A medida que avance el problema incluirá las casillas ya recorridas y la posición actual del caballo.

3. ¿Todas las casillas recorridas?: Se verifica si se han recorrido todas las casillas con el caballo. Si es así se termina y en caso contrario se continua con el algoritmo en la siguiente fase.

4. ¿Se puede mover el caballo a una casilla?: Se intenta colocar el caballo en una de las casillas a su alcance en un único movimiento. Esas casillas además no deben haberse visitado. En el caso de que haya más de una se ordenan siguiendo la regla de Warnsdorff de menor a mayor.

5. Retroceder última bifurcación: El algoritmo vuelve a la última bifurcación, eliminando el último movimiento del caballo junto a la indicación de que esa casilla ha sido visitada.

6. Mover caballo: Se mueve el caballo a la casilla seleccionada. La casilla en la que estaba anteriormente el caballo se marca como visitada.

7. Solución: Una matriz del mismo tamaño que el tablero usado que indica en qué movimiento ha visitado el caballo esa salida.

8. Fin: El algoritmo concluye su ejecución.

Se puede ver que es muy parecido al algoritmo de las n-damas. Y es natural al usar los dos *backtracking*. Las únicas diferencias reseñables son la representación (por algo es la parte más importante del algoritmo) y el uso de la regla de Warnsdorff. La

regla simplemente cambia el orden en el que se visitan las casillas priorizando aquellas que tengan menos movimientos disponibles.

1.2.5. Complejidad algorítmica del tour del caballo

En el estudio de la complejidad algorítmica, resulta habitual tomar en consideración el caso peor, aquel en que todas las circunstancias posibles que pueden ser adversas efectivamente ocurren. Una estrategia simplificada para calcular este escenario peor podría consistir en suponer que el caballo siempre podrá mover al máximo número de casillas.

Por ejemplo, partiendo de una casilla en un tablero de ajedrez, es posible realizar hasta 8 movimientos distintos, es decir, cada vértice en este contexto tiene un máximo de 8 aristas. Esto implica que para cada casilla transitada, existen 8 posibles elecciones. Dado que el tablero es de dimensiones n x m, esta decisión se tomaría n*m veces, proporcionando la siguiente fórmula para la complejidad en el peor caso:

$$\delta(n, m) = 8^{nm}$$

Donde n denota el número de filas y m el número de columnas. Aunque este cálculo muestra un crecimiento exponencial, afortunadamente este límite superior está bastante alejado de la verdadera complejidad del problema en cuestión.

En la mayoría de los casos, el número de elecciones será menor a 8, dado que puede suceder que la casilla en estudio tenga menos de 8 aristas, o que alguna de estas aristas conduzca a una casilla previamente visitada, la cual no necesitará ser considerada nuevamente. Aplicando la heurística de Warnsdorff, el recorrido del tablero se realiza desde el exterior hacia el interior,

lo que implica que, al llegar a las casillas centrales, ya se habrá visitado las casillas periféricas a estas, evitando así considerar numerosas trayectorias.

Con estas consideraciones, el problema puede resolverse en un tiempo razonable mediante un ordenador, incluso cuando n y m son superiores a 100. En el caso de que el tour del caballo busque encontrar un ciclo hamiltoniano, es decir, un recorrido que desde la última casilla permita regresar a la inicial en un solo movimiento, esta heurística no reduciría la complejidad de manera tan significativa, ya que no orienta la búsqueda hacia la última casilla apropiada.

Comparando la complejidad del problema general del camino/ciclo hamiltoniano y el problema del viajero, se puede observar una notable diferencia respecto al tour del caballo, dado que cada vértice en estos problemas podría tener un número indeterminado de aristas, desde una sola hasta una arista dirigida a cada uno de los otros vértices.

Nuevamente, para el cálculo de la complejidad es necesario considerar el peor caso, en el que todos los vértices están conectados con todos los demás. Si consideramos un grafo con n vértices, donde existe una arista entre todos los vértices, en el primer vértice tendríamos que elegir entre $n-1$ aristas, ya que solo se ha visitado este primer vértice. En el segundo vértice, tendríamos $n-2$ opciones de aristas, ya que el primer vértice y el actual ya han sido visitados. Continuando con este procedimiento, se puede apreciar su similitud con la función factorial, lo que nos lleva a la siguiente fórmula:

$$\gamma(n) = (n-1)! = (n-1) * (n-2) * ... * 2 * 1$$

En la búsqueda del camino/ciclo hamiltoniano, no se alcan-

zaría esta complejidad en un escenario donde todos los vértices están interconectados, ya que habría un camino/ciclo hamiltoniano posible seleccionando las aristas de manera aleatoria. Sin embargo, esta complejidad se presenta con frecuencia en el problema del viajero, debido a la posibilidad común de que todos los vértices estén conectados. En este caso, sería necesario explorar todas las rutas posibles para determinar cuál de ellas tiene un coste menor [6].

Con respecto a la heurística utilizada en el tour del caballo, no resultaría muy útil en el caso general, ya que su aporte a la identificación de las rutas más prometedoras no es suficiente para compensar el coste computacional asociado a su aplicación.

[6] Afortunadamente, es posible reducir esta complejidad utilizando el paradigma de la programación dinámica, que posee una complejidad de $O(2^n * n^2)$. Este tópico será tratado en el próximo capítulo. [3]

1.3. Las marchas del rey

1.3.1. Los caminos del rey

En este capítulo se abordará la cuestión de cuántas rutas puede tomar un rey de ajedrez, partiendo de una casilla específica a otra, utilizando el mínimo número de movimientos. Aunque esta pregunta pueda parecer simple a primera vista, se revelará como una indagación más compleja de lo que inicialmente se podría esperar. Como ilustración de ello, consideremos un tablero estándar de ajedrez de 8x8 y estimemos cuántos movimientos se requerirían para llegar a cada casilla desde una posición inicial del rey en la casilla e5.

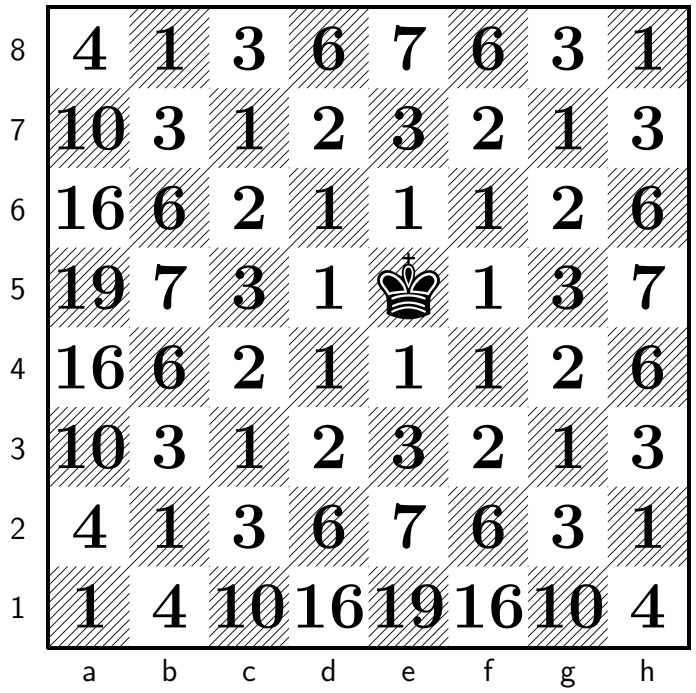


Figura 1.25: Número de rutas hacia cada casilla en un tablero de ajedrez 8x8

Al observar la figura, se pueden hacer varias conclusiones interesantes respecto a cuántas rutas existen para llegar a cada casilla. En primer lugar, vale la pena notar que en las casillas adyacentes al rey - ya sea horizontalmente, verticalmente o en diagonal - solo existe una ruta posible para llegar en el número mínimo de movimientos.

Además de las casillas adyacentes, existen otras marcadas con el número uno, lo que indica que sólo existe una forma de llegar a dicha casilla. Estas son las casillas que se encuentran en las diagonales del rey. Dado que para llegar a estas casillas siempre se deben hacer movimientos en diagonal, es coherente que sólo

haya una ruta. De manera más general, es observable que cuánto más lejos esté una casilla del rey, mayor será el número de rutas disponibles para llegar a ella.

Otra cuestión interesante a considerar en relación con este tema es el número mínimo de movimientos que el rey necesita realizar para llegar a una casilla específica. Este problema es considerablemente más sencillo que el anterior y puede resolverse mediante una fórmula simple. Sin embargo, tener conocimiento de este número mínimo de movimientos puede proporcionar información muy valiosa para resolver la pregunta original.

Para simplificar la comprensión de este problema, podemos comenzar examinando las casillas más cercanas al rey. Primero, la casilla en la que se encuentra el rey tiene una distancia de cero. Las casillas adyacentes al rey - tanto en sentido horizontal, vertical como diagonal - tendrán una distancia de uno. Con respecto a las demás casillas, aquellas adyacentes a las casillas con una distancia de uno tendrán esa distancia más uno, es decir, una distancia de dos. Al calcular la distancia de las casillas adyacentes a aquellas cuya distancia ya se conoce, será posible determinar todas las distancias. La Figura 1.26 ilustra la distancia a cada una de las casillas, considerando al rey situado en la misma casilla en un tablero de igual tamaño.

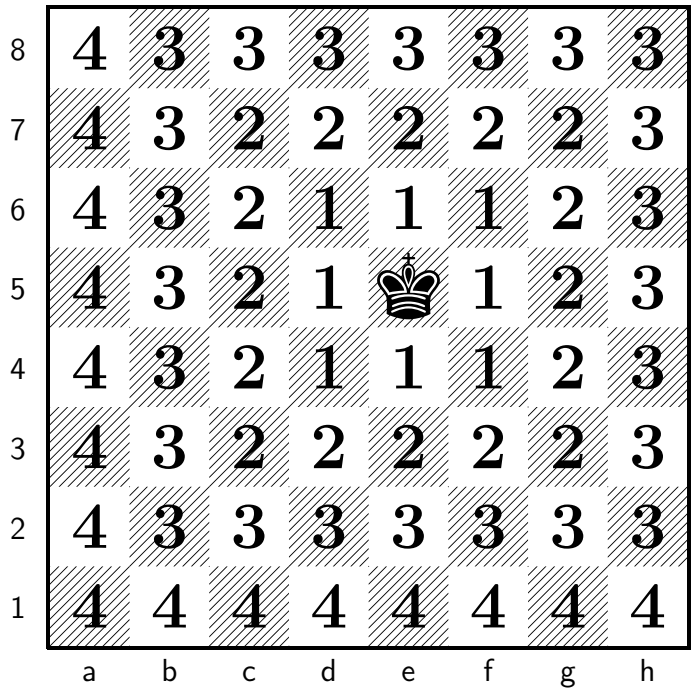


Figura 1.26: Distancias a las casillas en un tablero 8x8

Además de la estrategia mencionada anteriormente, también es posible recurrir a una fórmula directa que, conociendo la posición del rey y las coordenadas de la casilla objetivo, proporciona el número mínimo de movimientos necesarios para que el rey llegue a ella. La fórmula es la siguiente:

$$\text{máx}(|r_0 - a_0|, |r_1 - a_1|)$$

Donde $r = (r_0, r_1)$ son las coordenadas del rey, donde r_0 representa la fila y r_1 la columna, y $a = (a_0, a_1)$ las coordenadas de la casilla cuya distancia se busca calcular, con a_0 siendo la

fila y a_1 la columna de dicha casilla. En el caso de las filas la conversión es directa, pero para la columna se deberá usar la conversión en número de la letra. Es decir, la "a" pasará a ser 1, la "b" 2, etcétera.

Es natural preguntarse de dónde proviene esta fórmula y por qué funciona. En primer lugar, se puede comprobar que para el tablero anterior se obtienen los resultados correctos en todas las casillas, lo que sugiere que la fórmula es confiable. Para garantizar su validez, sería necesario realizar una demostración matemática, pero ya que esto excede el alcance de este libro, tendrás que confiar en mi palabra.

La intuición detrás de esta fórmula se basa en la idea de que el rey tarda el mismo tiempo en moverse a una casilla adyacente, incluso si esta se encuentra en diagonal. Esto no sucede con las distancias más comunes, como la distancia euclidiana (Teorema de Pitágoras) o la distancia Manhattan (donde moverse a una casilla adyacente en diagonal tendría un coste de 2). En nuestra fórmula, lo que sucede es que se calcula cuántas filas y columnas de diferencia existen, y dado que el rey puede moverse en diagonal, la distancia resulta ser el valor máximo entre el número de filas y columnas que separan las dos coordenadas. Esta distancia es conocida como la distancia del tablero de ajedrez o Chebyshov.

Retomando el problema principal tras esta breve digresión, es necesario encontrar una fórmula que nos permita calcular cuántas rutas existen para llegar a una casilla en un número mínimo de movimientos. En primer lugar, debemos enfocarnos en una casilla específica, que en este caso será d1. La Figura 1.27 muestra cuántas rutas existen para llegar a dicha casilla, junto con una observación reveladora.

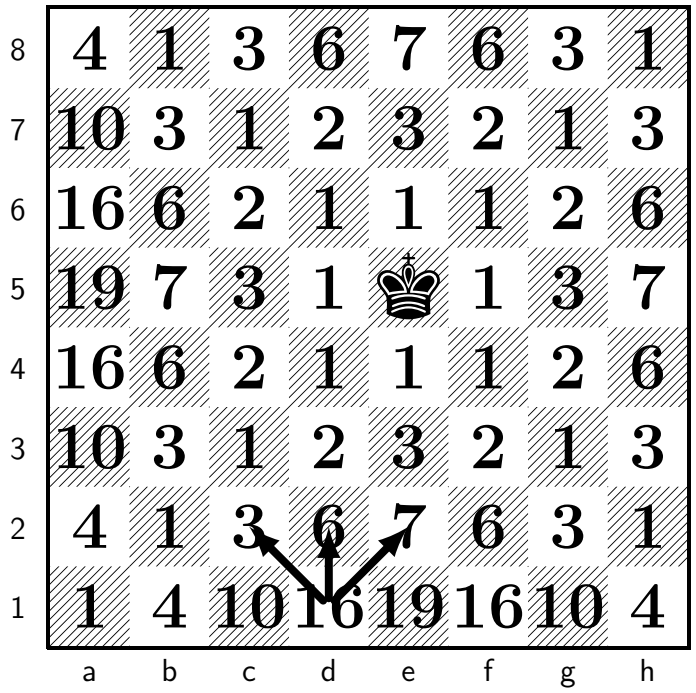


Figura 1.27: Número de rutas hacia la casilla d1

Las flechas en la figura apuntan a tres casillas adyacentes, y curiosamente, la suma de los valores de estas casillas es igual al valor de la casilla d1. Podría surgir la pregunta: ¿por qué elegir estas tres casillas adyacentes en lugar de las demás casillas adyacentes? Para responder a esta pregunta, sería útil para el lector volver a observar la Figura 1.26.

La casilla e1 está a una distancia de 4 casillas del rey, mientras que las casillas adyacentes analizadas están a una distancia de una unidad menos, es decir, 3. A partir de este escenario, podemos generalizar que el número de rutas hacia una casilla dada será la suma del número de rutas hacia las casillas adyacentes

que están a una distancia de una unidad menos. Si el lector es escéptico acerca de esta afirmación, puede proceder a verificarlo en las figuras previamente mostradas.

La Figura 1.28 ilustra el proceso de aplicación continua de esta estrategia hasta alcanzar el valor final.

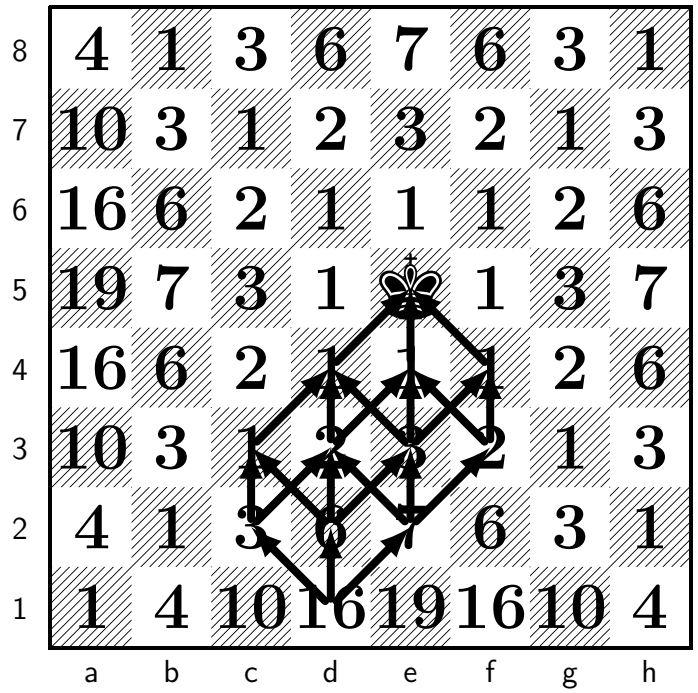


Figura 1.28: Resolución del número de rutas hacia la casilla d1

Por lo tanto, para calcular el número de caminos hacia una casilla específica, debemos conocer cuántas rutas existen hacia las casillas adyacentes más cercanas a la casilla de origen. Esto, a su vez, implica que tendríamos que realizar este mismo cálculo para estas casillas adyacentes utilizando las casillas adyacentes a estas, y así sucesivamente hasta llegar a la casilla inicial. En este

último caso, para facilitar los cálculos, establecemos que hay una única forma de ir a la casilla inicial desde la propia casilla inicial. Con esta información, podemos definir la siguiente fórmula matemática que permite ejecutar todo el proceso descrito:

$$f(x,k) = \begin{cases} 1, & \text{si } x = k \\ \sum_{c \in C_x} f(c,k), & \text{si } x \neq k \end{cases}$$

Donde $f(x,k)$ es la función que indica el número de rutas desde la casilla k hasta la casilla x y C_x son todas las casillas adyacentes más cercanas a la casilla x. Por lo tanto, la fórmula devolverá 1 si estamos en la casilla inicial y, en caso contrario, calculará el valor sumando los valores obtenidos por las funciones en las casillas adyacentes.

Después de analizar la fórmula, podemos notar que contiene recursividad; la parte superior de la función a trozos correspondería al caso base, mientras que la parte inferior sería el caso recursivo. Al ser una función recursiva, sería relevante analizar el número de llamadas recursivas que serían necesarias en el caso de un tratamiento informático de la función. Volviendo a la Figura 1.28, vemos que algunos valores se calculan varias veces, lo que obviamente es una pérdida de tiempo. Respecto a este punto, podemos analizar el número de llamadas recursivas usando la teoría vista en el tema anterior. En el ejemplo analizado anteriormente, pudimos ver que, dependiendo de la casilla, puede haber 1, 2 o 3 llamadas recursivas si no se trata del caso base y, como ya sabemos, hay que escoger el peor caso, donde hay hasta tres llamadas recursivas. Además, debemos tener en cuenta que, en este caso, el tamaño del problema no será el tamaño del tablero, sino la distancia entre la casilla analizada y el rey. A

continuación, se muestra la fórmula que da la complejidad del algoritmo.

$$3^{\max(|x_0-k_0|,|x_1-k_1|)} \in O(3^n)$$

Retomando el problema de las numerosas llamadas recursivas para el número de casillas que procesamos, el lector avispado se dará cuenta de que algunas flechas apuntan a la misma casilla y esto indica que calculamos el valor de la misma casilla varias veces, lo que claramente no es eficiente. ¿No habría alguna forma de evitar esto? La respuesta es sí, cuando se calcula el valor de una casilla, se puede almacenar su valor en un vector o matriz, dependiendo de la situación, para no tener que volver a calcularlo. Este procedimiento se conoce como programación dinámica.

1.3.2. Programación dinámica vs. las marchas del rey

El concepto de "programación dinámica" puede resultar un tanto misterioso a primera vista, y uno podría preguntarse: ¿cuál es el origen de este nombre tan peculiar? La respuesta a esta pregunta es bastante inusual. Richard Bellman, el inventor del término, lo eligió de manera estratégica para evitar que se cancelara su investigación. Consciente de que un nombre atractivo y vago como "programación dinámica"dificultaría cualquier intento de cancelación, optó por esta designación.

Con el origen del nombre aclarado, es momento de explorar la esencia de la programación dinámica y su utilidad en la resolución de problemas. El principal atractivo de la programación dinámica radica en su habilidad para simplificar problemas de alta complejidad, especialmente aquellos con complejidades

exponenciales o superiores, y convertirlos en problemas de complejidad polinómica. Esto es posible gracias a una característica fundamental de la programación dinámica: la capacidad de almacenar y reutilizar soluciones parciales ya calculadas, evitando así el costo de recalculaciones innecesarias. A pesar de ser un concepto simple, su eficacia es extraordinaria y tiene aplicaciones significativas en una amplia gama de problemas.

Ahora que hemos examinado el principio fundamental de la programación dinámica, nos adentraremos en un caso práctico para apreciar su eficacia en la reducción del número de llamadas recursivas. Empezaremos con el tablero que se muestra en la Figura 1.29, enfocándonos en la casilla e1, que se ha utilizado anteriormente.

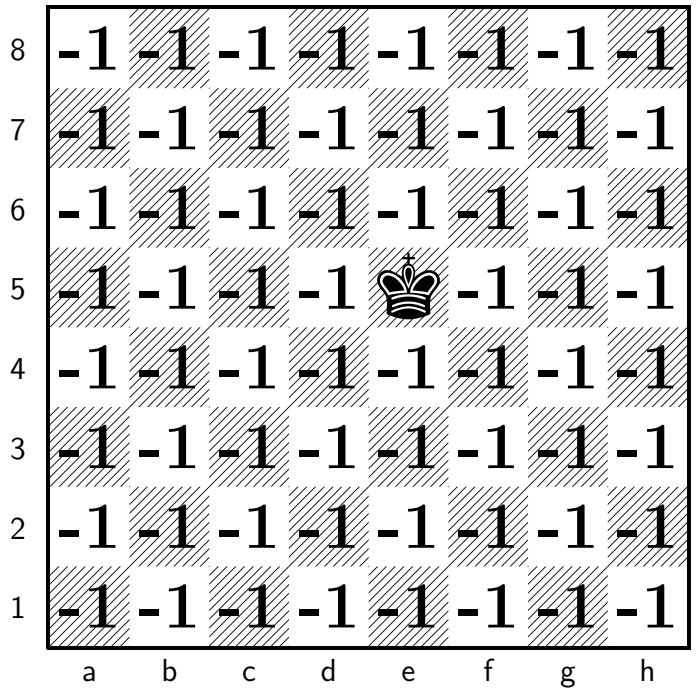


Figura 1.29: Inicio de la resolución de marchas del rey con programación dinámica

Al examinar el tablero, se nota que todas las casillas tienen un valor de -1. Este valor es un marcador que indica que aún no se ha calculado la distancia a esa casilla. En la primera fase de resolución, nos enfocamos en el cálculo de las diferentes formas en que podemos llegar a la casilla c2.

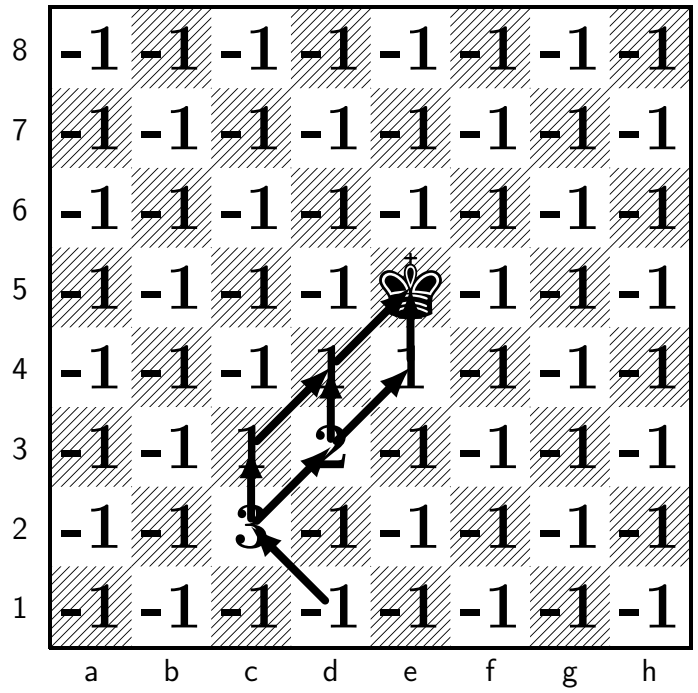


Figura 1.30: Primera fase de la resolución de marchas del rey con programación dinámica

En esta primera fase, se ha calculado el valor de la casilla c2, que resulta ser 3. Además, se han determinado los valores de todas las casillas necesarias para obtener el valor de c2. Para evitar cálculos redundantes, todos estos valores se almacenan en una matriz, preparándolos para ser reutilizados en futuras operaciones. De hecho, la necesidad de esta práctica ya se ha puesto de manifiesto en esta fase, ya que el valor de la casilla d4 ha sido requerido tanto para el cálculo de la casilla c3 como para la casilla d3.

Tras concluir la primera fase, nos embarcamos en la segunda, donde se calcula el valor de la casilla d2. Este cálculo correspon-

de a la segunda llamada recursiva requerida para determinar el valor de la casilla d1. El procedimiento de cálculo de esta fase se ilustra en la Figura 1.31.

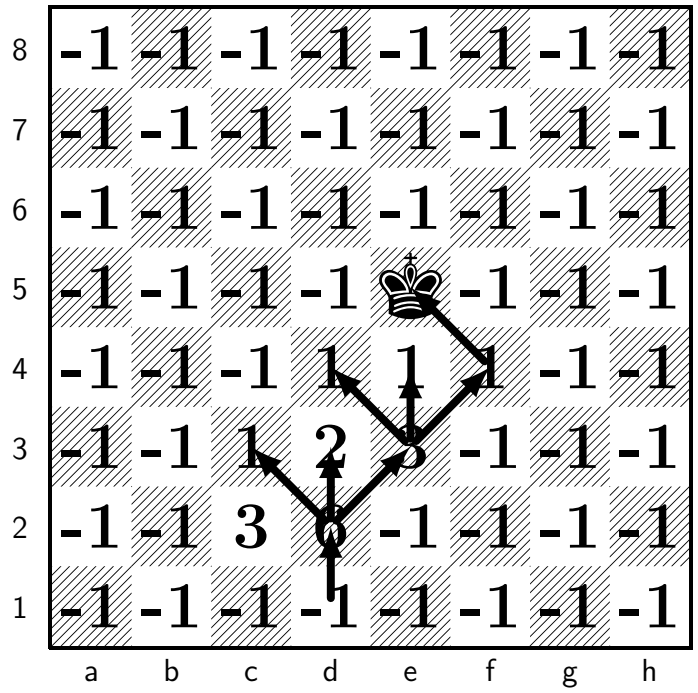


Figura 1.31: Segunda fase de la resolución de marchas del rey con programación dinámica

En esta segunda fase, se hace un uso aún más intensivo de los valores ya calculados, especialmente aquellos que se determinaron en la fase anterior, como las casillas d3 y e4.

Por último, llegamos a la fase final, en la que calculamos finalmente el valor de la casilla d1. Este cálculo se puede apreciar en la Figura 1.32.

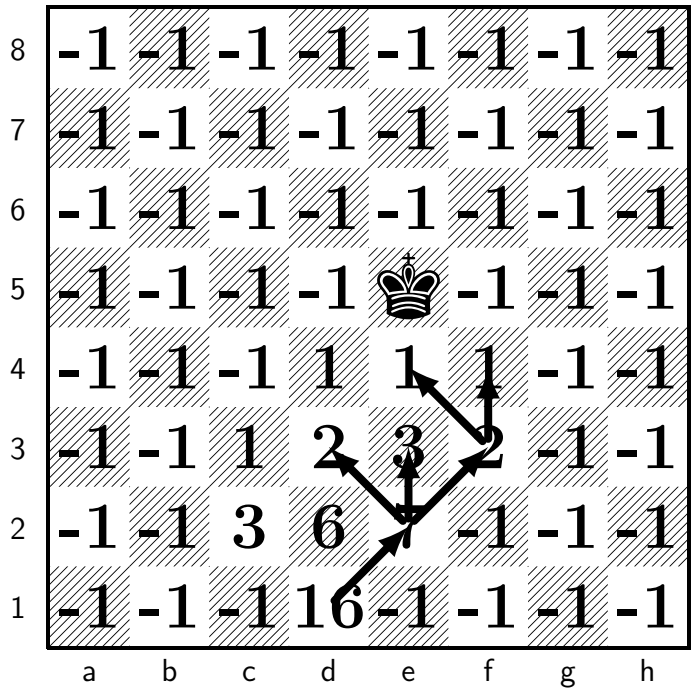


Figura 1.32: Última fase de la resolución de marchas del rey con programación dinámica

El proceso de resolución del problema se intensifica aún más en su uso de los valores de las casillas previamente calculadas. En términos generales, a medida que se avanza en el algoritmo, más cálculos se pueden ahorrar. En este punto, hemos conseguido tal eficiencia que ya no es necesario llegar al caso base.

Para resumir todo lo discutido hasta ahora:

El término "programación dinámica" fue acuñado por Richard Bellman como una estrategia para proteger su investigación de posibles cancelaciones. Este método se caracteriza por su capacidad para simplificar problemas de alta complejidad en problemas de complejidad polinómica. Esto se logra mediante

el almacenamiento y la reutilización de soluciones parciales ya calculadas para evitar recalculaciones.

En un ejemplo práctico, analizamos cómo se aplicaba la programación dinámica a un tablero de ajedrez. En la primera fase, se calculó el número de formas de llegar a la casilla c2, almacenando los valores de las casillas necesarias para este cálculo. La segunda fase implicó el cálculo de la casilla d2, haciendo uso intensivo de los valores ya determinados en la fase anterior.

Finalmente, en la última fase, se calculó el valor de la casilla d1, destacando la eficiencia de la programación dinámica, ya que se ahorraron más cálculos a medida que se avanzaba en el algoritmo. Este procedimiento resultó tan efectivo que ni siquiera fue necesario llegar al caso base. En esencia, la programación dinámica demuestra ser un enfoque excepcionalmente eficaz y económico para la resolución de problemas complejos.

Capítulo 2

Inteligencia artificial

2.1. ¿Qué es la inteligencia artificial?

Para profundizar en el campo de la inteligencia artificial (IA), es imprescindible distinguir claramente varios términos relacionados. En esencia, la inteligencia artificial se refiere al conjunto de algoritmos, técnicas y métodos que permiten a una máquina emular comportamientos inteligentes, que tradicionalmente consideramos humanos. En otras palabras, la IA dota a un sistema computacional de una apariencia de inteligencia similar a la humana. Esta definición, intencionalmente amplia, abarca un espectro de algoritmos heterogéneos, que pueden participar en una partida de ajedrez o, incluso, componer un poema.

Poniendo el foco en el juego del ajedrez, el objetivo es desarrollar un sistema computacional capaz de jugar de una forma similar a un humano, o incluso superarlo. Esto implica que la máquina debe ser capaz de proyectar múltiples jugadas hacia adelante (algo relativamente sencillo para una máquina), y también, adoptar una visión estratégica a largo plazo (algo sumamente complejo para un sistema informático). Se pueden explorar ambos aspectos mediante algoritmos puros o a través del

aprendizaje que los sistemas computacionales pueden adquirir mediante ciertos algoritmos, un campo denominado aprendizaje automático o *machine learning*. Este último enfoque ha demostrado un notable éxito.

2.1.1. Aprendizaje automático

El aprendizaje automático engloba todos los algoritmos que permiten a un sistema computacional aprender a partir de datos. Esta perspectiva contrasta fuertemente con el enfoque tradicional de programación. En el modelo clásico, el primer paso es analizar la información y estudiar el problema manualmente, generalmente realizado por un programador. Posteriormente, se escribe el programa basado en los conocimientos adquiridos y, una vez finalizado, se evalúa su rendimiento y se corrigen los errores detectados. Este proceso se repite tantas veces como sea necesario hasta alcanzar el rendimiento requerido.

Sin embargo, en el aprendizaje automático, aunque se sigue un proceso similar, hay una diferencia esencial: todo el proceso se lleva a cabo por el sistema computacional sin intervención humana después de su inicio. La fase de codificación del programa se reemplaza por el entrenamiento del algoritmo de aprendizaje automático. Los sistemas computacionales son ideales para ejecutar tareas repetitivas, mientras que los humanos eventualmente nos cansamos o nos aburrimos. Por esta razón, estos algoritmos de aprendizaje superan en gran medida a un algoritmo estándar diseñado por un programador.

El dominio del aprendizaje automático es vasto y está creciendo exponencialmente en la actualidad. Casi cualquier programa o aplicación hoy en día utiliza el aprendizaje automático para ejecutar parte de sus funcionalidades. Una forma sencilla de cla-

sificar estos algoritmos es basándose en los tipos de datos que utilizan. Las principales categorías son: aprendizaje supervisado, no supervisado, semisupervisado y reforzado.

Aprendizaje supervisado

En este tipo de aprendizaje, los datos proporcionados vienen con un "etiquetado" o "solución". Por ejemplo, si se intenta determinar si una foto pertenece a un perro, en los datos vendrá indicado que esa foto concreta es de un perro. La máquina se entrena con estos datos, intenta predecir si una foto es de un perro y, en caso de error, puede aprender de sus errores y ajustarse para evitarlos en el futuro. La meta final es que el algoritmo sea capaz de clasificar una imagen nueva que no haya visto antes y de la que probablemente no tenga la etiqueta.

Esta categoría se puede subdividir en dos subcategorías: clasificación y regresión. La clasificación corresponde al caso mencionado anteriormente donde se debe predecir la categoría de un conjunto de datos (una imagen, un vídeo, un texto, etc.) entre un conjunto de categorías predefinidas. En contraposición, la regresión implica predecir un valor continuo (infinitas categorías posibles) para un conjunto de datos dado.

Aprendizaje no supervisado

En el aprendizaje no supervisado, la tarea del sistema computacional es más desafiante ya que no cuenta con etiquetas para los datos. Es decir, se le proporciona una imagen de un perro, pero la máquina no tiene ninguna indicación de que es un perro. Las aplicaciones de este tipo de aprendizaje están relacionadas con el agrupamiento de datos (*clustering*), la reducción de la dimensionalidad de los datos (para resaltar las características más

significativas), y la detección de anomalías en los datos (datos que se desvían de lo común).

Aprendizaje semisupervisado

Como sugiere su nombre, este tipo de aprendizaje combina aspectos del aprendizaje supervisado y no supervisado. Aquí se dispone de un conjunto de datos etiquetados, generalmente pequeño, y otro conjunto sin etiquetas, usualmente más grande. Se utilizan algoritmos que combinan técnicas de aprendizaje supervisado y no supervisado. Un ejemplo destacado en este campo son las redes generativas adversarias (GANs). Estas consisten en dos redes neuronales: una que genera imágenes realistas de cierta categoría y otra que debe determinar qué imágenes son reales y cuáles son generadas. La segunda red neuronal indica a la primera en qué se ha equivocado, permitiendo que la primera ajuste sus parámetros de forma autónoma. El entrenamiento culmina cuando la red neuronal discriminadora ya no puede distinguir entre las imágenes reales y las generadas.

Aprendizaje reforzado

El aprendizaje reforzado se sitúa en una posición intermedia entre el aprendizaje supervisado y no supervisado, aunque sigue una filosofía totalmente diferente. En este modelo, se concibe el aprendizaje como un juego donde el sistema debe tomar una serie de decisiones que resultan en una recompensa determinada. Esta recompensa proporciona la retroalimentación sobre si las acciones tomadas fueron acertadas o no. Debido a que la recompensa se ve afectada por todas las acciones previas, no es posible modelarlo como aprendizaje supervisado. Al mismo

tiempo, tampoco puede considerarse aprendizaje no supervisado dado que existe una retroalimentación.

Los lectores más perspicaces habrán notado que este tipo de aprendizaje es especialmente adecuado para juegos como el ajedrez, y efectivamente así es. La combinación de aprendizaje reforzado y profundo ha demostrado ser muy efectiva en este campo. No profundizaremos más en este tema aquí, ya que se tratará con mayor detalle en el próximo capítulo.

2.1.2. Aprendizaje profundo

El aprendizaje profundo corresponde a los algoritmos de aprendizaje automático que hacen uso de redes neuronales con múltiples capas. Una red neuronal está compuesta por multitud de elementos llamados neuronas o perceptrones. Estos perceptrones tienen una estructura sencilla, basada en la multiplicación y suma de matrices seguida de una función no lineal. Su éxito radica en que, al combinar muchos de estos perceptrones, se pueden aproximar prácticamente cualquier función imaginable.

Aunque las redes neuronales puedan parecer un desarrollo reciente, en realidad su origen se remonta al siglo pasado. Si han existido durante tanto tiempo, ¿por qué no se han empezado a utilizar activamente hasta estos últimos años? La razón principal es que hasta hace poco era prácticamente imposible entrenar y ajustar efectivamente redes neuronales con muchas capas. Sin embargo, gracias a los avances recientes en hardware y en el proceso de retropropagación (*backpropagation*), estas redes han alcanzado un gran éxito en numerosos campos. De nuevo, se tratará este campo más detenidamente en el próximo capítulo.

2.2. Definición formal del ajedrez

El análisis de cómo la inteligencia artificial (IA) y sus subcampos abordan el estudio del ajedrez requiere primeramente del establecimiento de un objetivo definido. El fin perseguido es el desarrollo de una IA que pueda competir eficientemente en una partida de ajedrez completa, ya sea contra un oponente humano o contra otra IA.

A primera vista, este puede parecer un reto colosal. Sin embargo, se simplifica enormemente si se restringe al objetivo de determinar la mejor jugada posible en una posición específica. Este subobjetivo, en realidad, se equipara con el objetivo inicial, dado que si la IA logra discernir la mejor jugada en cualquier situación concreta, realizar óptimamente las jugadas consecutivas resulta una consecuencia directa.

La posibilidad de enfocarnos solamente en la posición actual para encontrar la mejor jugada se deriva del hecho de que, para esta tarea, no es necesaria la información sobre las posiciones previas o eventos ocurridos anteriormente en la partida. Aunque esta observación puede parecer obvia, su relevancia no debe ser subestimada, ya que nos permite categorizar el ajedrez como un proceso de decisión de Markov (PDM).

Pero, ¿qué es un PDM y cómo se relaciona con las Cadenas de Markov? Antes de contestar a estas preguntas, es crucial explicar el concepto de las Cadenas de Markov y su conexión intrínseca con los PDM.

2.2.1. Cadenas de Markov

Las Cadenas de Markov surgen del campo de la probabilidad, lo que puede resultar sorprendente al relacionarlas con el

ajedrez, un juego en el que el azar no tiene cabida. El nexo de unión radica en que, pese a ser el ajedrez un juego puramente determinista (dada una jugada, se puede prever con absoluta certeza la posición resultante), puede ser modelado como un fenómeno probabilístico donde la posición que surge después de una jugada específica tiene un 100 % de probabilidad de ocurrir.

En este contexto, las Cadenas de Markov representan procesos estocásticos en los que la probabilidad de que ocurran uno o más eventos depende únicamente del estado actual del proceso. La Figura 2.1 proporciona un ejemplo visual de una Cadena de Markov.

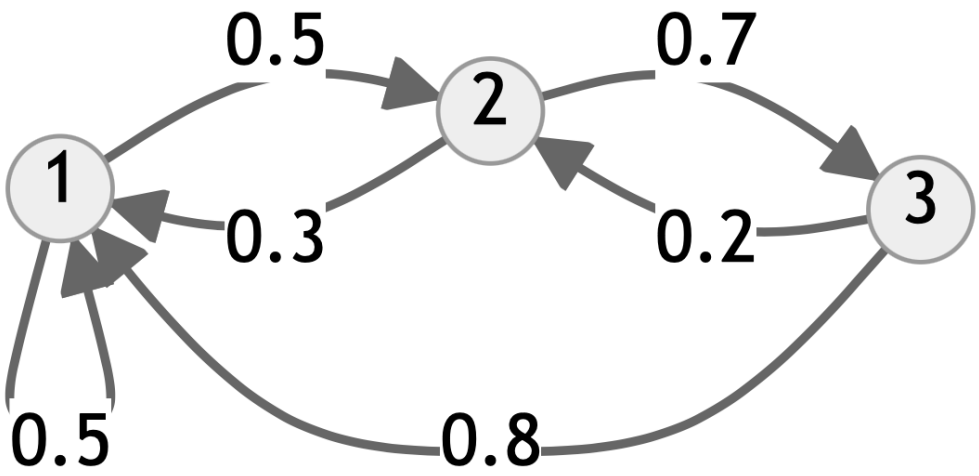


Figura 2.1: Ejemplo de una Cadena de Markov

En el ejemplo citado, las flechas simbolizan las transiciones y los números asociados representan la probabilidad de que se produzca dicha transición. Los círculos, por otro lado, indican los estados. Por ejemplo, desde el estado 2, existe un 70 % de

probabilidad de transitar al estado 3 y un 30 % de permanecer en el estado 1. También es posible que un estado transite a sí mismo, como se puede observar en el estado 1, que tiene un 50 % de probabilidad de mantenerse inalterado.

Para asemejar nuestra definición a un juego de ajedrez, consideremos primero que el estado actual representa la disposición de las piezas en el tablero. Sin embargo, en una cadena de Markov, no encontramos un equivalente a las jugadas o movimientos. Esto se debe a que las Cadenas de Markov no consideran la interacción en su proceso. Para incorporar dicha interacción, necesitamos expandir las capacidades de las Cadenas de Markov al Proceso de Decisión de Markov.

Además, es fundamental tener en cuenta que el ajedrez es un juego de "información perfecta". ¿Qué quiere decir esto? Que tanto el jugador de las piezas blancas como el de las negras están al tanto de la ubicación de todas las piezas, a quién le corresponde el turno y toda la información pertinente al estado actual de la partida.

2.2.2. Proceso de Decisión de Markov

Un Proceso de Decisión de Markov se compone por cuatro elementos:

- Estados (E)
- Acciones (A)
- Función de transición (F)
- Recompensas (R)

$$\text{PDM} = \{E, A, F, R\}$$

Los Estados (E) comprenden todas las posibilidades en las que puede encontrarse el proceso. Algunos estados son iniciales (en ellos puede iniciarse el proceso) y otros son finales (el proceso termina al llegar a estos). Cuando se alcanza un estado final, se dice que ha concluido un "episodio" y se regresa a uno de los estados iniciales para comenzar otro "episodio".

Las Acciones (A) engloban todas las acciones que puede realizar un agente, es decir, la entidad que interactúa con el proceso. Usualmente, dependiendo del estado actual, sólo se pueden ejecutar un subconjunto de acciones del conjunto total.

La Función de transición (F) toma como parámetros el estado actual y una acción, y devuelve un nuevo estado. En otras palabras, determina el estado subsiguiente basándose en el estado actual del proceso y la acción elegida por el agente.

Las Recompensas (R) asignan un valor de recompensa (que puede ser negativo) al agente, en función del estado alcanzado. Esto sirve para guiar al agente hacia los estados más beneficiosos.

2.2.3. Ajedrez como Proceso de Decisión de Markov

Ahora bien, para definir el ajedrez como un Proceso de Decisión de Markov, es necesario describir cada uno de estos elementos en el contexto del juego.

Estados: Similarmente a las Cadenas de Markov, un estado en el ajedrez incluye la disposición de las piezas en el tablero, el turno del jugador, la posibilidad de realizar un enroque

o una captura al paso, entre otros factores relevantes. Los estados iniciales son aquellos que no son finales, es decir, cualquier disposición de piezas que no resulte en una victoria o un empate. Los estados finales incluyen todas las situaciones en las que un jugador ha ganado o se ha cumplido una condición de empate. Siguiendo la analogía con los "episodios" mencionados anteriormente, cada partida de ajedrez sería un episodio.

Acciones: En el ajedrez, con dos jugadores, existen dos agentes que pueden realizar una acción (un movimiento) en su turno. Las posibles acciones de cada agente varían enormemente dependiendo del jugador (las blancas tendrán disponibles movimientos diferentes a las negras).

Función de transición: Esta función toma como entrada la posición actual y el movimiento deseado del jugador en turno (blancas o negras), y devuelve la nueva disposición del tablero como resultado de ese movimiento. Esta función es determinista en el ajedrez, es decir, un movimiento específico siempre resultará en una única disposición del tablero.

Recompensas: La asignación de recompensas queda a criterio del diseñador del sistema, pero debe seguir ciertos principios. Todos los estados no finales deben tener una recompensa neutral de 0, ya que no se puede determinar si esa disposición favorece a alguno de los jugadores. Los estados finales deben clasificarse en función de si resultan en una victoria para uno de los jugadores o un empate. La asignación de recompensas debe seguir el siguiente criterio:

$$R_{ganar} > R_{empatar} > R_{perder}$$

Donde R_{ganar}, $R_{empatar}$ y R_{perder} representan las recompensas por ganar, empatar y perder, respectivamente. Siguiendo esta

convención, la recompensa por ganar debe ser mayor que la de empatar, y la recompensa por empatar debe ser mayor que la de perder. A menudo, se asignan valores de +1 para ganar, 0 para empatar y -1 para perder, aunque estos valores pueden variar según las preferencias del diseñador del sistema.

La posibilidad de modificar la recompensa otorgada por cada movimiento en un juego, que por defecto se asume como cero, brinda la oportunidad de ajustar el transcurso de la partida de acuerdo a nuestras necesidades o preferencias. Al asignar una recompensa negativa a cada movimiento, estamos de hecho incentivando un juego más rápido, buscando finalizar la partida en el menor número de movimientos posible. Contrariamente, una recompensa positiva por movimiento fomentaría una partida de mayor duración. En resumen, el ajuste de las recompensas permite una cierta flexibilidad para calibrar el comportamiento del agente en función de los objetivos específicos que deseemos alcanzar.

Consideremos el juego del ajedrez, en el que implementamos un Proceso de Decisión de Markov (PMD). El procedimiento del juego involucra una secuencia cíclica de pasos que se ejecutan en cada turno, hasta llegar a un estado terminal, es decir, hasta que la partida concluye.

La secuencia de pasos se presenta de la siguiente manera:

1. El agente recibe un estado y una recompensa del entorno. En este contexto, el agente es responsable de realizar los movimientos en el juego. En el ajedrez, por ejemplo, donde los jugadores de las piezas blancas y negras toman turnos para moverse, habría un agente distinto para cada color. El entorno tiene la responsabilidad de proporcionar a los agentes la información sobre el estado actual del juego, in-

cluyendo la posición de las piezas y si la partida ha llegado a su fin. Además, utiliza la función de transición para determinar cuál será el siguiente estado. Si la partida ha terminado, el proceso concluye y se otorga a cada agente su recompensa final.

2. Después de recibir la información sobre la posición actual, el agente decide la acción (movimiento) que tomará en esa posición y la comunica al entorno.

3. Tras recibir el movimiento del agente, el entorno aplica la función de transición para determinar la posición siguiente. Concluido este paso, se retorna al primer paso, y el agente correspondiente al color que toca mover toma el control.

Este proceso se repite hasta que se llegue a un estado terminal, lo cual en el ajedrez significa generalmente un jaque mate, un empate o la rendición de uno de los jugadores. En la Figura 2.2 se muestra un ejemplo de todo este proceso.

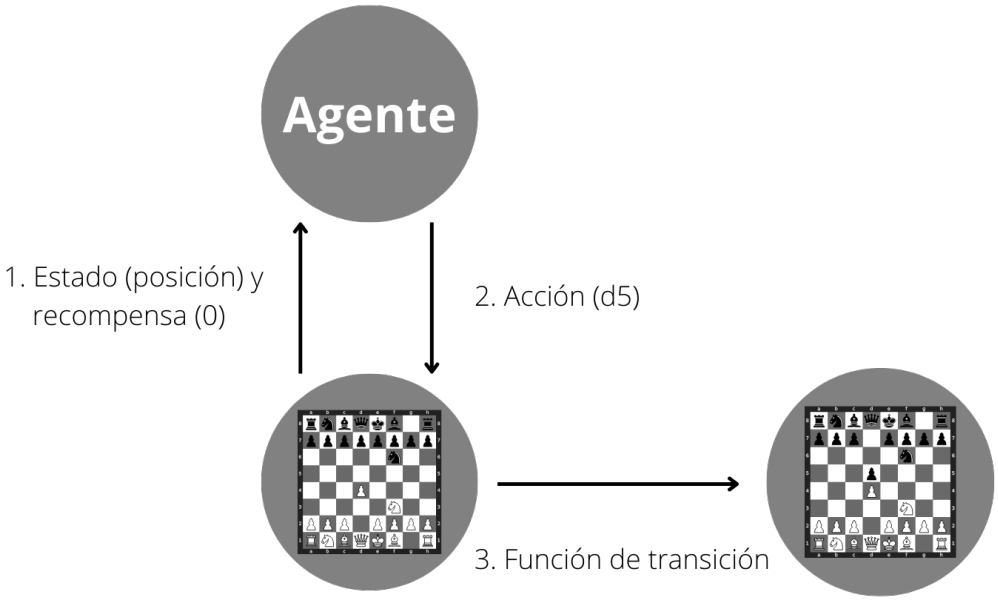


Figura 2.2: Ejemplo del ajedrez como Proceso de Decisión de Markov

Tras esta formalización del ajedrez, cabe preguntarse, ¿qué se necesita programar para jugar ajedrez? Recordando el objetivo inicial de esta sección, queda claro que debemos programar el agente. La programación del proceso es relativamente sencilla en comparación con la creación de un agente capaz de jugar competente al ajedrez. Para permitir que el agente descubra el mejor movimiento en una posición dada, podemos utilizar una técnica de árbol de búsqueda (en la que la IA simula una serie de movimientos futuros tanto para sí misma como para su rival) o intentar estimar la calidad de los distintos estados (decidir quién tiene la ventaja en cada estado, o si hay un equilibrio). En general, una combinación de estos dos enfoques da los mejores resultados.

2.3. Árbol del juego

Los árboles de juego representan una estructura fundamental para la conceptualización de estrategias en juegos como el ajedrez. Esta herramienta estratégica se originó por primera vez en el siglo XIX, atribuyéndose a Charles Babbage las primeras incursiones matemáticas en este campo [4]. No obstante, es a Von Neumann a quien generalmente se le concede el crédito por la creación de este concepto [5]. A pesar de ello, el primer análisis riguroso de los juegos competitivos utilizando este método fue desarrollado por Emilie Borel [6]. En su trabajo, Von Neumann demostró el teorema minimax (que se explicará en detalle en la siguiente sección), sugiriendo que en teoría, uno podría identificar la mejor jugada en una posición de ajedrez utilizando este enfoque.

El concepto principal que subyace en el árbol de juego es el análisis exhaustivo de todas las posibles jugadas desde una posición dada, y a su vez, de todas las respuestas posibles del oponente en cada una de las posibles nuevas posiciones resultantes. Este proceso iterativo se repite hasta alcanzar un estado final para cada una de las ramificaciones. Una vez construido este árbol de juego, es necesario recorrerlo de manera específica para determinar la jugada más óptima en cada una de las posiciones; para ello, se empleará el algoritmo minimax.

A continuación, se presenta un ejemplo ilustrativo de un árbol de juego. Esta representación será notablemente similar a la de una partida de ajedrez real, lo cual no es coincidencia, ya que el árbol de juego es, en esencia, una simulación de una partida que toma en consideración todos los movimientos posibles.

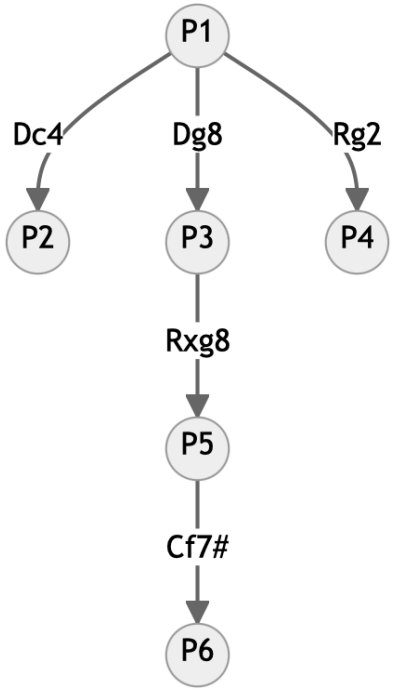
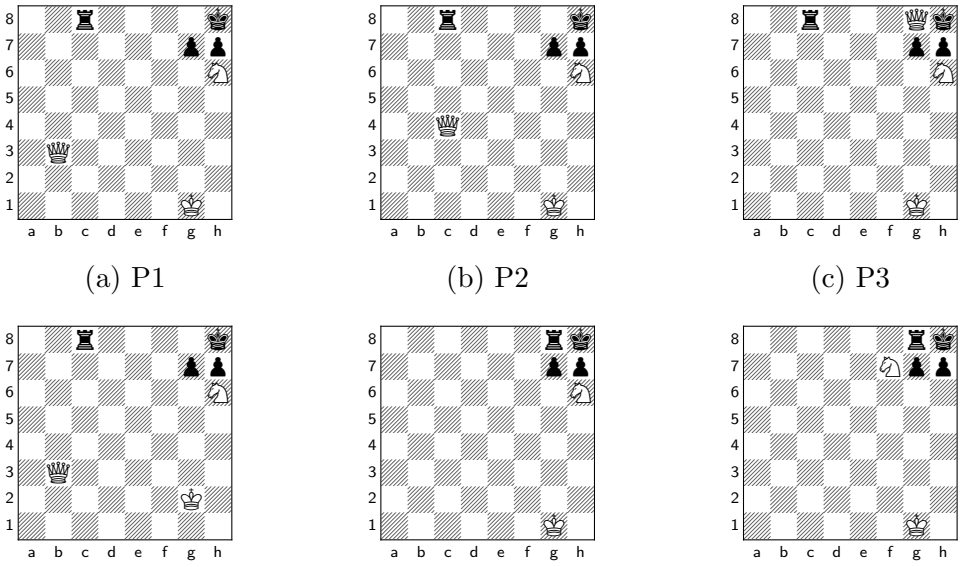


Figura 2.3: Ejemplo de árbol de juego



(a) P1 (b) P2 (c) P3

(d) P4 (e) P5 (f) P6

Figura 2.4: Tableros representados en el ejemplo de árbol de juego

En este escenario, sólo se consideran tres movimientos potenciales de las blancas, aunque en realidad existen más opciones, como Rf2, Da2, Cf7, entre otros. De las tres movidas blancas, sólo se continúa con Dg8, mientras que las otras dos jugadas permiten diversas respuestas por parte de las negras. En contraposición, frente a Dg8, las negras deben responder de manera forzada con Txg8 (capturando a la dama), y las blancas a su vez responderán con Cf7, lo cual resulta en jaque mate al rey negro (las negras no disponen de un movimiento que alivie al rey del ataque), otorgando la victoria a las blancas.

La generación de este árbol sigue un procedimiento semejante al *Backtracking* presentado en el primer capítulo. En cada posición, se enumeran todas las posiciones resultantes posibles y se continua recursivamente con el mismo proceso hasta llegar a un estado final. Debido a la similitud con el *Backtracking*,

este método comparte el problema del crecimiento exponencial de las posiciones a revisar. En el ajedrez, este crecimiento es particularmente acentuado dada la alta cantidad de movimientos posibles desde una determinada posición. Por ejemplo, en la posición inicial, las blancas pueden ejecutar 20 movimientos distintos y esta cifra puede aumentar aún más en el mediojuego o apertura. Sin embargo, existen posiciones donde solo es posible un movimiento (jugada forzada). Debido a estas variaciones, el cálculo de la complejidad del árbol de juego del ajedrez resulta altamente complejo. Afortunadamente, el célebre matemático Shannon llevó a cabo este cálculo por nosotros, proporcionando una estimación del número de posiciones en el árbol de juego (el número de Shannon) y del número de partidas diferentes de ajedrez, ambos íntimamente relacionados.

Según Shannon, el número 10^{120} representa una cota inferior para el número de posiciones distintas en el ajedrez (es decir, el número de posiciones podría ser aún mayor, pero no menor que este límite). Respecto a la cota inferior del número de partidas de ajedrez, se estima en torno a 900^{40}, suponiendo que hay treinta movimientos posibles por cada posición y que cada turno implica un movimiento por bando, es decir $30 * 30 = 900$, asumiendo también que la duración media de una partida es de 40 movimientos, obteniendo el resultado anterior [7].

Al tener en cuenta la complejidad de este problema, queda claro que es inviable generar el árbol de juego completo, por lo que se recurre al uso de estimaciones o heurísticas. Estas heurísticas indicarán qué tan favorable es un estado, permitiendo que solo sea necesario generar una fracción reducida del árbol de juego.

Además, para que el árbol de juego ofrezca información útil,

es necesario recorrerlo de una manera específica. El algoritmo que se utilizará para este propósito es el minimax, junto con su versión mejorada con la poda alfa-beta. Estos conceptos se tratarán en las siguientes secciones.

2.4. Heurísticas

Una heurística es un método estratégico o una regla general que empleamos para simplificar la toma de decisiones o la resolución de problemas complejos. Este concepto puede visualizarse como poseer una brújula mientras se navega por un bosque denso, en vez de un mapa detallado: aunque no asegura que encontraremos el camino más directo a nuestro destino, nos ayuda a evitar perdernos en el laberinto de la incertidumbre.

Las heurísticas nos facilitan la capacidad de tomar decisiones y actuar de manera rápida sin requerir un análisis profundo en cada opción que se nos presenta. En muchas situaciones, estas estrategias mentales resultan ser extremadamente útiles y eficientes. Sin embargo, es importante tener en cuenta que también pueden conducirnos a errores o sesgos cognitivos, dado que se fundamentan en simplificaciones de la realidad, no en su análisis detallado.

Un ejemplo claro de heurística es la regla "si todos los demás están haciendo algo, probablemente sea la mejor opción". Aunque en ciertos casos esta heurística puede resultar eficaz (por ejemplo, elegir un restaurante concurrido en lugar de uno vacío), no siempre asegura la elección óptima (como seguir una moda dañina o perjudicial simplemente porque es popular).

En contextos como el juego del ajedrez, las heurísticas se manifiestan como funciones que toman un estado (posición) como

entrada y devuelven un valor numérico que refleja la "bondad" de ese estado. Es crucial comprender que las heurísticas son aproximaciones, ya que la calidad de un estado solo se puede determinar plenamente mediante el desarrollo del árbol de juego que se origina a partir de ese estado. Con las heurísticas, buscamos evitar precisamente ese despliegue completo. Podemos definir esta noción de manera más formal como sigue:

$$f(\Theta) = \delta_\Theta$$

Θ : Estado
δ_Θ : Valor de la heurística asociado al estado Θ
$f(\Theta)$: Función heurística

2.4.1. Aplicación de las heurísticas al ajedrez

Tras haber establecido una definición abstracta de lo que constituye una heurística, procedamos a explorar cómo se aplica esta en el contexto del ajedrez. Una de las heurísticas más comúnmente utilizadas en ajedrez implica calcular la diferencia en "peones" entre los dos jugadores. Para ello, se asigna a cada pieza (con la excepción del rey) un valor basado en su relevancia estratégica en el juego. Luego, se calcula el valor total de las piezas para cada jugador y se resta el valor total de las piezas del jugador con las piezas negras del valor total de las piezas del jugador con las piezas blancas. En este sentido, si el resultado de esta heurística es positivo, las blancas están en una posición ventajosa; si es 0, la posición está equilibrada; y si es negativo, las negras tienen la superioridad.

A continuación, en la Figura 2.5, se muestra el valor asignado a cada pieza. Por supuesto, el valor de una pieza depende de su

posición específica en el tablero, pero estas estimaciones suelen ser buenas aproximaciones en la mayoría de las situaciones.

Pieza	Valor (en peones)
Dama	9
Torre	5
Alfil	3
Caballo	3
Peón	1

Figura 2.5: Valor en peones asignado a cada pieza en el ajedrez

Siguiendo una escala basada en peones, estos tienen un valor unitario de 1. El alfil y el caballo se les asigna un valor de 3, aunque algunas opiniones sostienen que los alfiles pueden ser ligeramente más valiosos que los caballos, atribuyéndoles un valor de 3,5 peones. La torre se valora en 5 peones, mientras que el valor de la dama depende de la presencia en el tablero de las torres correspondientes a su propio bando. Estas valoraciones de las piezas son el resultado de consensos empíricos de la comunidad ajedrecística a lo largo de los años y, de hecho, es una de las primeras cosas que aprenden los principiantes en este desafiante juego. Cabe destacar que el rey no se incluye en esta valoración debido a su importancia crítica en el juego: su captura termina la partida y siempre está presente en el tablero.

A continuación, se ilustra la aplicación de esta heurística con una posición específica, que se muestra en la Figura 2.6.

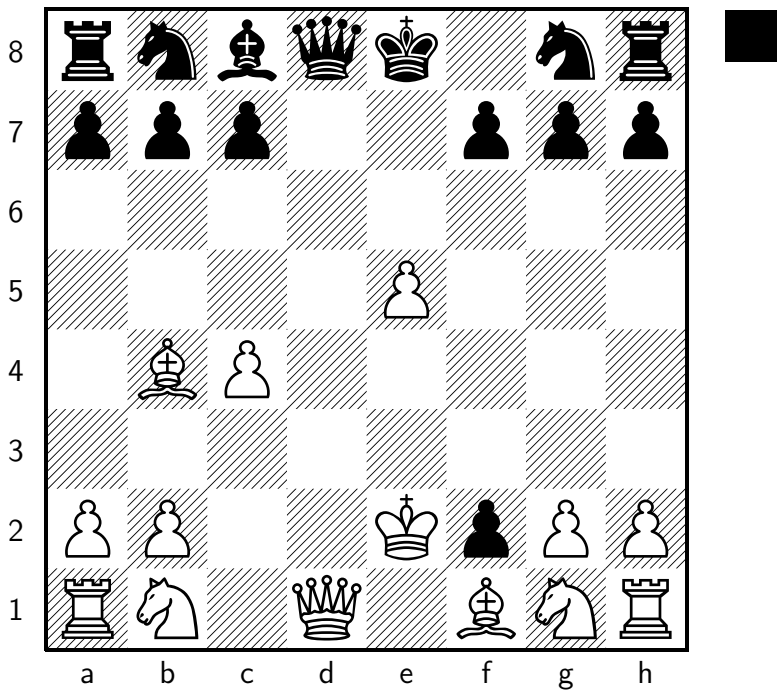


Figura 2.6: Posición de ejemplo para calcular la heurística

Primero, calculamos la suma de los valores de las piezas para cada bando. Las blancas poseen una dama, dos torres, dos alfiles, dos caballos y seis peones, dando un valor total de $9*1+5*2+3*2+3*2+1*6=37$. Por otro lado, las negras tienen una dama, dos torres, un alfil, dos caballos y siete peones, con un valor total de $9*1+5*2+3*1+3*2+1*7=35$. Finalmente, restamos el valor total de las piezas negras del valor total de las piezas blancas, obteniendo $37-35=2$, lo que indica que, según la heurística, las blancas tendrían una ventaja equivalente a dos peones. Pero ¿es esto correcto?

Sorprendentemente, la realidad contradice la predicción de la

heurística. A pesar de que las blancas parecen tener una ventaja según la heurística, en realidad están en una posición perdedora y son las negras las que poseen una ventaja significativa después de la brillante jugada fxg1c.

Desafortunadamente, esta heurística primitiva no suele producir buenos resultados, ya que es demasiado materialista, preocupándose más por la cantidad de piezas que cada bando tiene que por su colocación en el tablero. Esto implica que no es capaz de evaluar adecuadamente la mayoría de las posiciones. Sin embargo, esta heurística proporciona una base sólida para el desarrollo de heurísticas más sofisticadas, que deberían incorporar conceptos más abstractos, como la ubicación de las torres en columnas abiertas, los alfiles en diagonales abiertas, etc. Con la introducción de estos conceptos más complejos, se podría obtener una heurística mucho más precisa.

Antes de la introducción del aprendizaje reforzado, las heurísticas creadas manualmente usadas por los principales módulos de ajedrez (programas de juego de ajedrez) tenían una gran complejidad, lo que les permitía evaluar adecuadamente posiciones estratégicas.

Ahora, podríamos preguntarnos: ¿No sería preferible que los módulos desarrollaran sus propias heurísticas a partir de la experiencia de jugar partidas? La respuesta es sí. Y tal como se mencionó en la sección 4.1, esto entraría en la categoría de aprendizaje automático, donde un algoritmo aprende basándose en su experiencia. Sin embargo, el desafío radica en que el algoritmo no recibirá una retroalimentación inmediata sobre si el movimiento que acaba de realizar es bueno o malo, sino que deberá esperar hasta el final de la partida. Además, dentro de las jugadas que haya realizado pueden haber tanto muy buenas como

muy malas, por lo que para distinguirlas deberá jugar una gran cantidad de partidas e ir variando las jugadas para ver cuáles son las más efectivas.

El tipo de aprendizaje automático que mejor se adapta a esta situación es el aprendizaje reforzado y, combinado con redes neuronales profundas (aprendizaje reforzado profundo), ha permitido que los módulos de ajedrez alcancen niveles de juego anteriormente inimaginables.

2.5. El algoritmo minimax

2.5.1. Introducción

El funcionamiento del algoritmo minimax es intrínsecamente anclado a su denominación. Este algoritmo se utiliza para identificar la jugada óptima teóricamente, en cualquier juego de información perfecta – donde todos los jugadores tienen un conocimiento completo y compartido del estado del juego – y de suma cero – donde las ganancias de un jugador se equilibran con las pérdidas del otro(s). Este algoritmo presupone que tanto el jugador que lo utiliza como el oponente harán movimientos óptimos. Aquí es donde el nombre del algoritmo toma relevancia: el jugador seleccionará el movimiento que maximice su beneficio, mientras que el oponente escogerá el movimiento que minimice el beneficio del jugador.

Estos procesos de maximización y minimización se desarrollan en todos los niveles del árbol de juego, que fue desglosado en una sección previa. Cada estado del árbol se clasifica por su nivel, que se define como el número de transiciones requeridas para llegar a ese estado desde el estado inicial o raíz. Por tanto, la raíz se identifica como nivel 0, los estados directamente

vinculados a ella como nivel 1, y así sucesivamente.

En los niveles pares del árbol de juego, el jugador maximiza su ventaja eligiendo el movimiento más beneficioso, mientras que en los niveles impares, la minimización entra en juego cuando el oponente intenta seleccionar el movimiento que proporcione menos beneficio al jugador.

Un último aspecto a explicar es cómo se determina la valoración de cada estado. Idealmente, se podría explorar cada rama del árbol hasta llegar a un estado final, y de esta forma, conocer la verdadera valoración de cada estado. Sin embargo, en la práctica, esta exploración exhaustiva es inviable, por lo que recurrimos a las heurísticas. En un nivel de profundidad predeterminado, se detiene el desarrollo del árbol de juego y se evalúan los estados finales de cada rama con la heurística. A partir de estas valoraciones, se calculan las valoraciones de los estados restantes en un proceso de abajo hacia arriba, hasta llegar al estado inicial o raíz del árbol de juego. Evidentemente, para este proceso se emplea el algoritmo minimax.

El concepto del algoritmo minimax puede ser un tanto abstracto a primera vista, por lo que se proporciona un ejemplo paso a paso de su aplicación en la sección a continuación.

El primer paso es calcular los valores de los estados finales del árbol de juego generado (en este caso, los estados al final de cada rama), empleando la heurística. Este proceso se representa en la Figura 2.7.

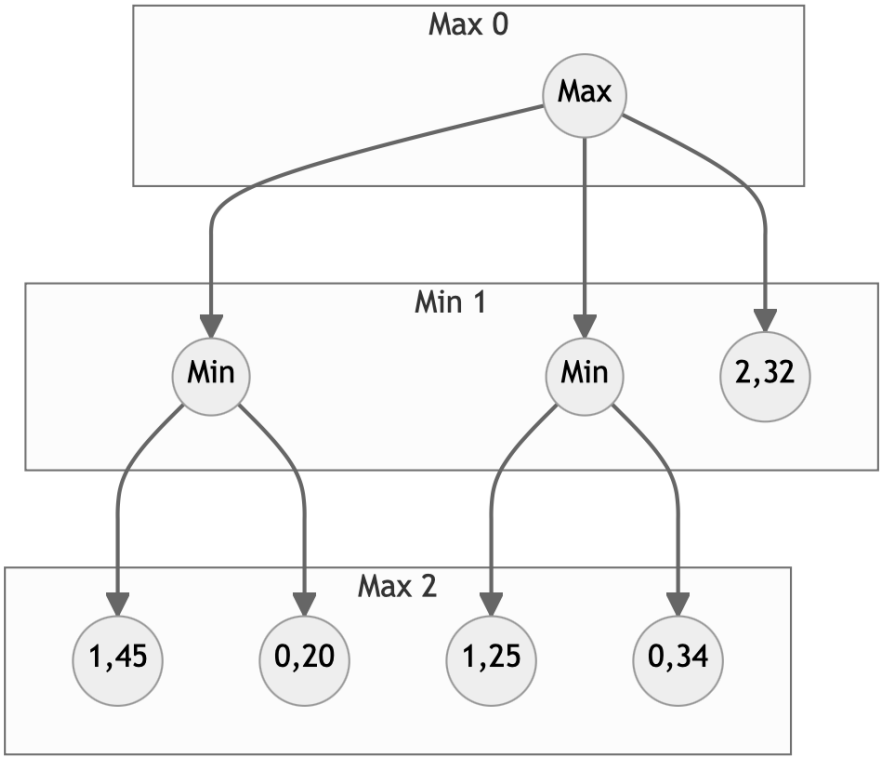


Figura 2.7: Primer paso del algoritmo Minimax genérico

En el árbol de decisión, aquellos valores que ya han sido determinados se ilustran utilizando su valor numérico correspondiente. Por otro lado, los nodos cuyos valores aún no han sido calculados se representarán como Max o Min, esto dependerá de si su valor será el máximo o mínimo de sus nodos hijos respectivamente. Es importante destacar que los nodos cuyos valores ya han sido calculados no poseen nodos hijos, es decir, no están vinculados a ningún nodo en un nivel inferior. En la terminología del árbol de decisiones, a estos nodos se les llama nodos hoja, haciendo una analogía con las hojas de un árbol en el mundo natural.

Una vez que se han calculado las heurísticas de estos estados

finales, podemos proceder a determinar la heurística del nivel inmediatamente adyacente. Para ello, debemos tomar el valor mínimo de los estados en el nivel inferior, lo cual corresponde a la fase de minimización. Por ejemplo, en el nodo ubicado a la izquierda del nivel 1, se tomará mín$(1,45;0,2)$, aquí, el uso del punto y coma es para evitar confusiones con la coma que se utiliza para separar cifras decimales y enteras, y se obtendrá como resultado 0,2, ya que este valor es menor que 1,45. Siguiendo este proceso, en el estado central del nivel 1, seleccionamos mín$(0,25;0,34)$ y obtenemos 0,25, ya que es menor que 0,34. Finalmente, el nodo ubicado a la derecha en el nivel 1, al ser un estado final, ya tiene un valor calculado, por lo que se ignora temporalmente. En la Figura 2.8, se ilustra el resultado de este proceso.

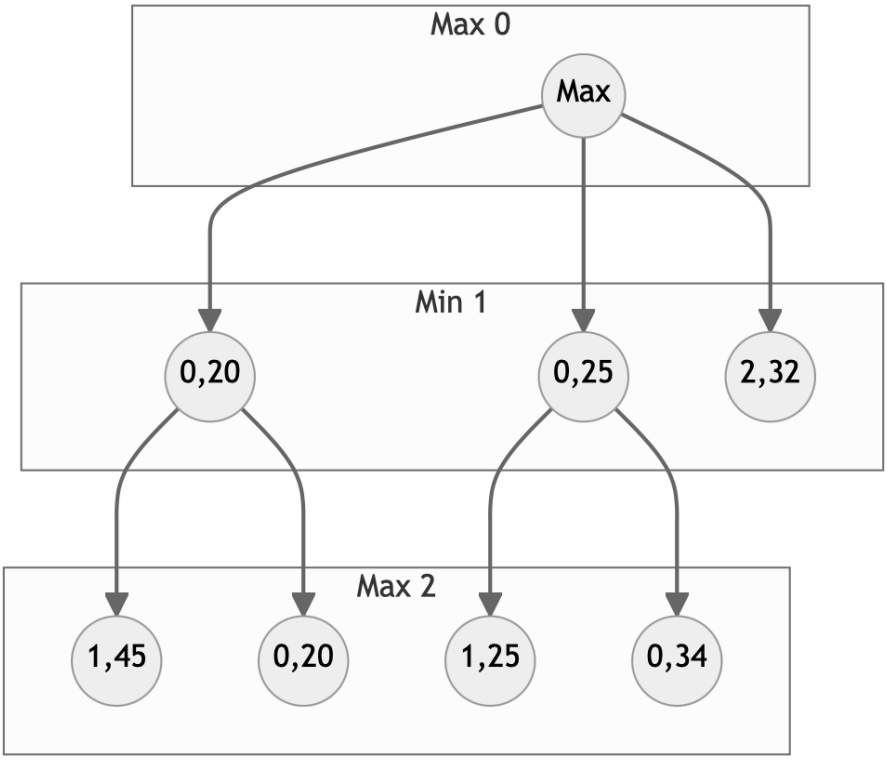


Figura 2.8: Segundo paso del algoritmo Minimax genérico

Ahora, debemos llevar a cabo una fase de maximización. Es decir, necesitamos obtener el valor máximo de los estados adyacentes inferiores al estado situado en el nivel cero o estado inicial. Así, aplicamos máx$(0,2; 0,25; 2,32)$, obteniendo como resultado 2,32, ya que este valor es mayor que 0,2 y 0,25. Dado que este es el estado inicial, se ha completado el algoritmo Minimax. Por tanto, podemos determinar que la mejor transición es la que lleva al estado con el valor 2,32. En la Figura 2.9, se muestra el árbol de juego finalizado.

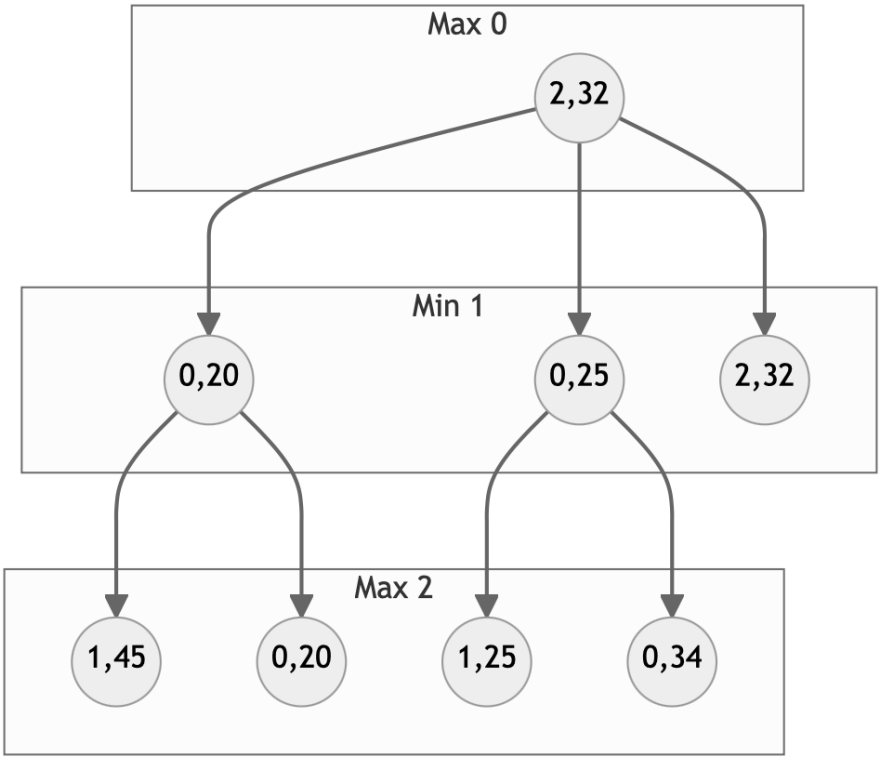


Figura 2.9: Tercer paso del algoritmo Minimax genérico

Es relevante mencionar que no todos los estados finales se encuentran en el mismo nivel de profundidad. En el ejemplo anterior, el estado final con valor 2,32 está en el nivel 1, en lugar de estar en el nivel 2, como los demás. Esto puede suceder si el estado con valor 2,32 representa un estado final en el árbol de juego completo (es decir, no se puede expandir más desde allí) o bien, se ha decidido no expandir más ese nodo. Esta última técnica es comúnmente utilizada, ya que mediante una heurística adicional (la cual determina cuán beneficioso es expandir un estado) se puede decidir si es valioso continuar expandiendo ese estado o finalizar la ramificación en ese punto.

Después de revisar este ejemplo general y sencillo, podemos

avanzar a un ejemplo más complejo y realista en el que se aplique el algoritmo minimax a un juego de ajedrez.

2.5.2. Algoritmo minimax aplicado al ajedrez

2.5.2.1. Consideraciones especiales

El ajedrez, como caso específico de la aplicación del algoritmo minimax, posee algunas particularidades. Lo más relevante es que el tamaño del árbol de juego del ajedrez es inmensamente grande, un hecho que se ha enfatizado repetidamente. Esto nos motiva a buscar formas de reducir el número de estados que se añaden al árbol de juego simulado que utiliza el algoritmo minimax. Una estrategia que ya se ha mencionado consiste en usar una heurística que permita evaluar cuán "interesante" es un estado para decidir explorarlo. Este concepto de "interesante" ayuda a establecer una prioridad en la expansión de los estados, ya que es probable que no tengamos tiempo de explorar todos hasta su conclusión. Al definir esta heurística, deberíamos considerar factores como el nivel del estado en relación al estado inicial, el valor de la heurística que indica la calidad de una posición, entre otros.

Finalmente, cabe destacar que la expansión de este árbol de juego estará limitada principalmente por el tiempo disponible. Por ejemplo, en una partida con tiempo limitado, la computadora deberá devolver la mejor jugada que haya podido encontrar en el breve tiempo que tiene. Por el contrario, si la partida permite un tiempo de juego más extenso, la computadora tendrá más tiempo para encontrar la mejor jugada, lo que probablemente resultará en una mejor solución en la misma posición que en el caso anterior, dado que ha podido expandir más el árbol

de juego.

2.5.2.2. Ejemplo aplicado al ajedrez

Se presenta el árbol de juego ilustrado en la Figura 2.10, que representa la secuencia de movimientos en el ejemplo de aplicación del algoritmo minimax al juego de ajedrez. La posición inicial corresponde al comienzo de una partida de ajedrez estándar, donde las blancas realizan la jugada d4, seguida por la respuesta de las negras con d5. A partir de esta posición, las blancas deben tomar la decisión de determinar cuál movimiento es el mejor.

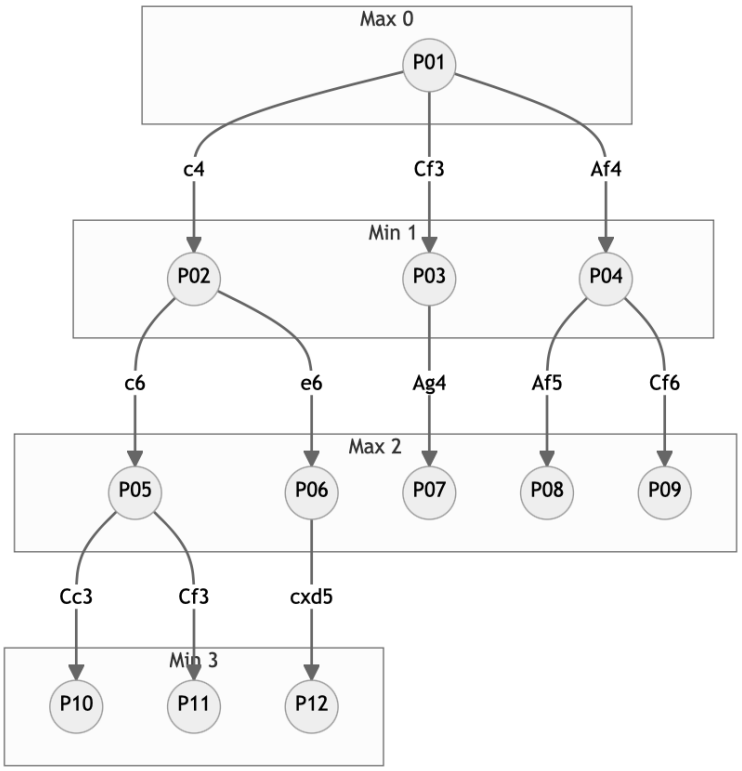
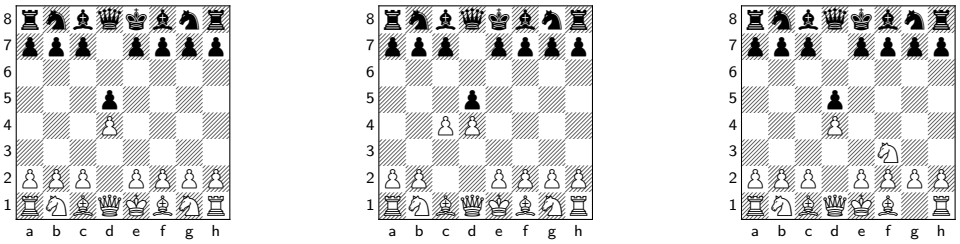
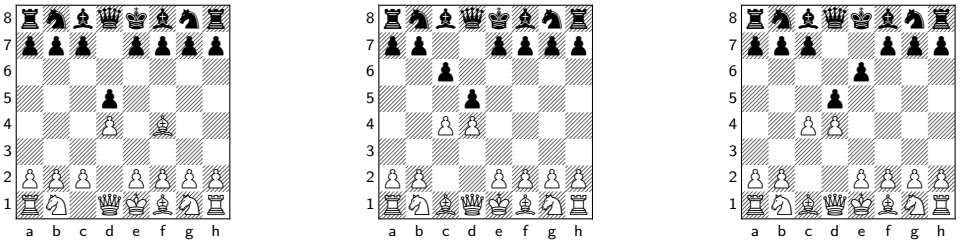


Figura 2.10: Árbol de juego del ejemplo de minimax aplicado al ajedrez

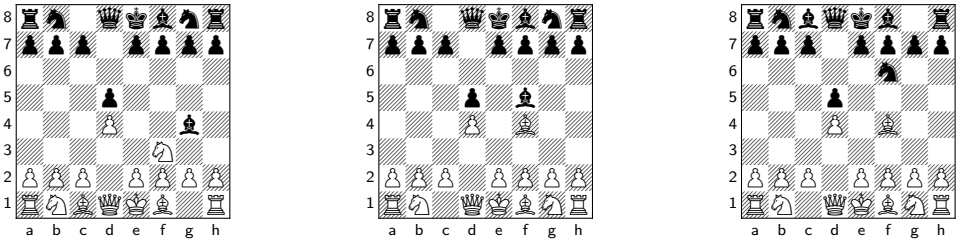
Cada nodo del árbol representa una posición del tablero y las diferentes ramas indican los posibles movimientos que se pueden realizar desde esa posición. La Figura 2.11 muestra los tableros correspondientes a cada posición representada en el ejemplo de aplicación del algoritmo minimax al ajedrez.



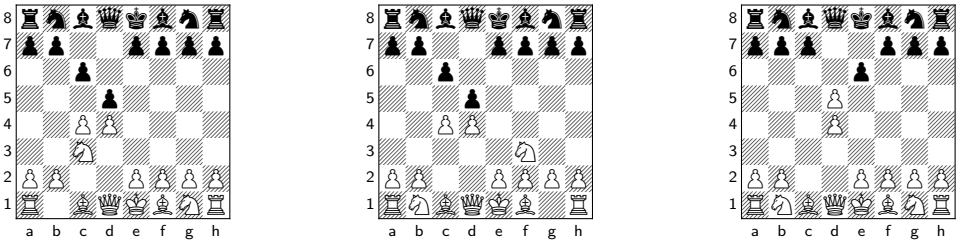
(a) P01 (b) P02 (c) P03



(d) P04 (e) P05 (f) P06



(g) P07 (h) P08 (i) P09



(j) P10 (k) P11 (l) P12

Figura 2.11: Tableros representados en el ejemplo de minimax aplicado al ajedrez

Una vez que se ha obtenido el árbol de juego y las posicio-

nes correspondientes, se puede proceder a la siguiente fase del algoritmo Minimax, que consiste en calcular el valor de las posiciones terminales. Estas posiciones se refieren a aquellas que no tienen ramas saliendo de ellas, es decir, no se pueden realizar más movimientos a partir de ellas.

Al calcular la heurística de estas posiciones terminales, se obtendrá una aproximación realista del valor de cada posición. El resultado de este cálculo dará lugar a una nueva representación del árbol de juego, como se muestra en la Figura 2.12.

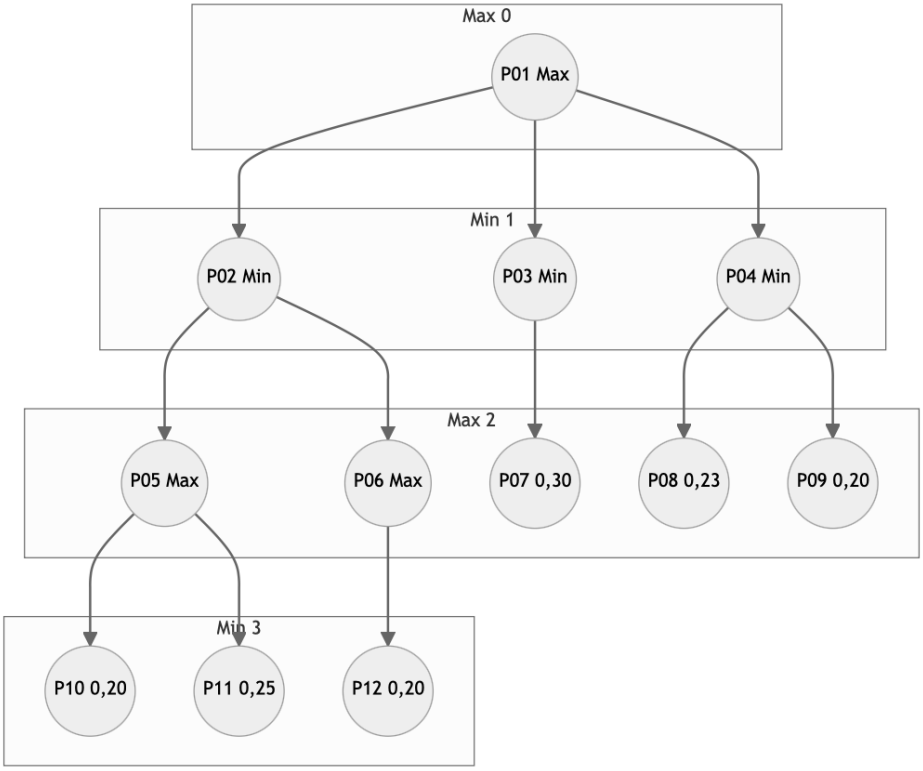


Figura 2.12: Primera fase del ejemplo de minimax aplicado al ajedrez

Al examinar el árbol de juego, se puede notar que no todos los estados se encuentran en el mismo nivel. Por ejemplo, el estado

P10 está en el mismo nivel que P11 y P12, es decir, en el nivel 3. Sin embargo, P07, P08 y P09 se sitúan en el nivel 2. Esta disparidad en los niveles implica que será necesario encontrar tanto el máximo como el mínimo de esos niveles respectivamente.

En términos prácticos, esto significa que, en el proceso de evaluación y selección de movimientos, se deberá buscar el máximo valor posible en el nivel 3 (donde se encuentran P10, P11 y P12), mientras que en el nivel 2 se buscará el mínimo valor. Estas operaciones de búsqueda y selección permitirán determinar los movimientos más favorables para cada jugador, considerando tanto las posibilidades de éxito como las contramedidas que puedan adoptar los oponentes en cada nivel del árbol de juego.

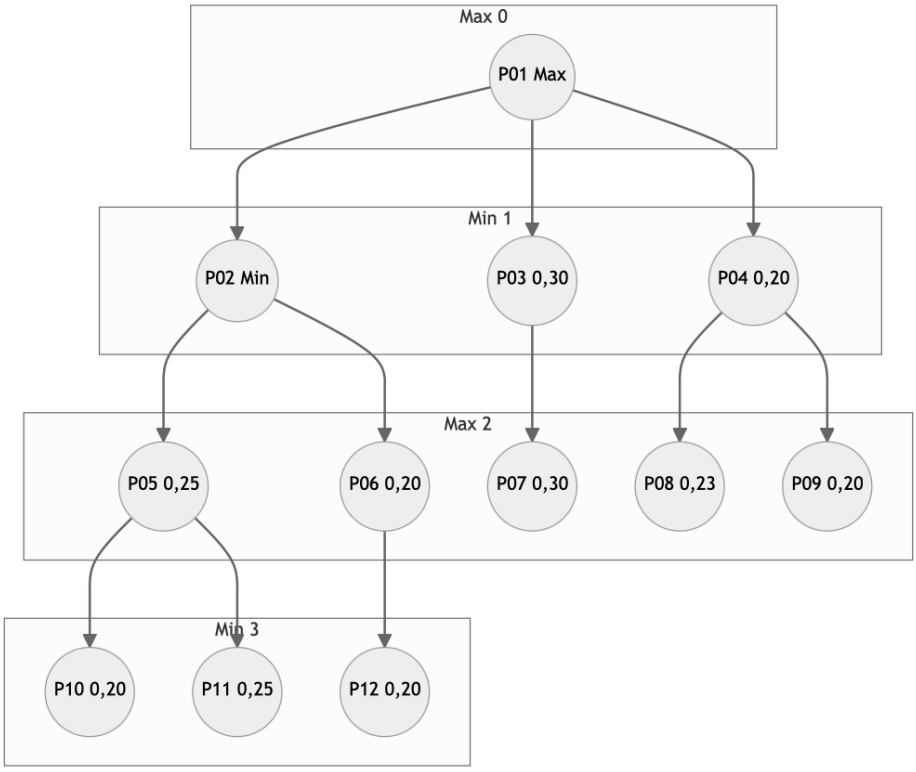


Figura 2.13: Segunda fase del ejemplo de minimax aplicado al ajedrez

En la segunda fase del algoritmo, se realiza la maximización y minimización para calcular los valores de algunos estados en el nivel 2 y nivel 1, respectivamente.

En el nivel 2, el estado P05 se calcula como el máximo entre los valores de P10 y P11, lo cual resulta en un valor de 0,25. Por otro lado, el estado P06 toma el valor de P12, ya que P12 es su único descendiente en el árbol de juego.

En cuanto al proceso de minimización en el nivel 1, el estado P03 toma el valor de P07, dado que P07 es su único descendiente. Por otro lado, el estado P04 toma el mínimo valor entre P08 y P09, que es 0,20.

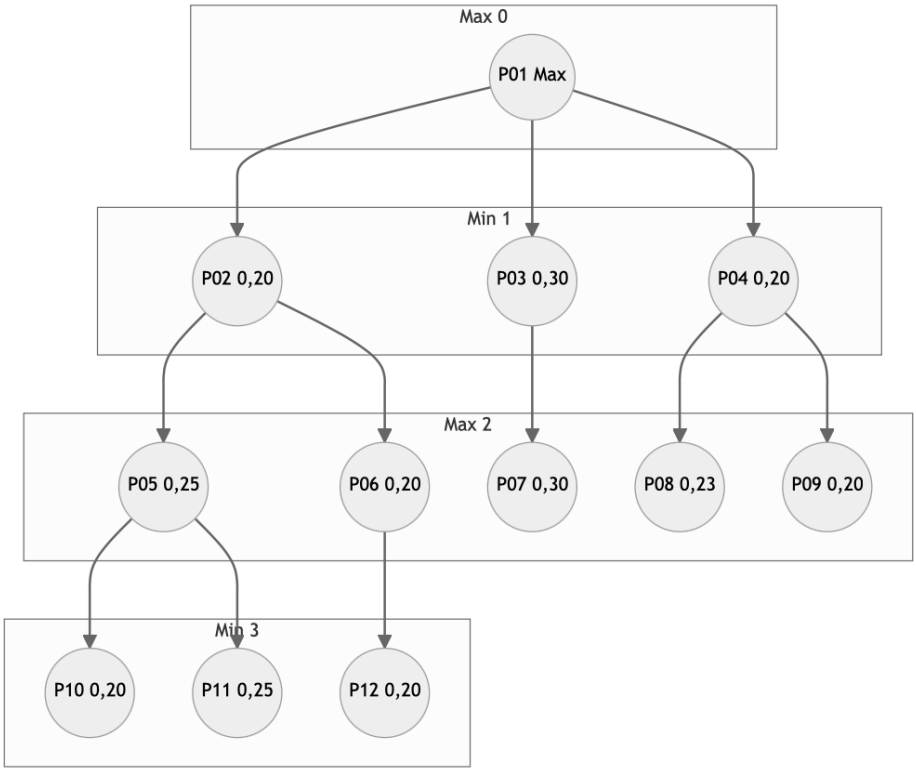


Figura 2.14: Tercera fase del ejemplo de Minimax aplicado al ajedrez

En la tercera fase del algoritmo, se realiza el cálculo del valor del estado restante en el nivel 1, que es P02. En este caso, el valor de P02 se determina como el mínimo entre los valores de P05 y P06, ya que se encuentra en un nivel impar, lo que implica una fase de minimización.

Con este cálculo adicional, se han obtenido todos los valores necesarios para calcular el valor del estado original. Este estado original está representado en la Figura 2.15. A partir de los valores calculados en las fases anteriores, se puede determinar el valor óptimo de este estado según el algoritmo minimax, considerando las estrategias de maximización y minimización en los niveles correspondientes del árbol de juego.

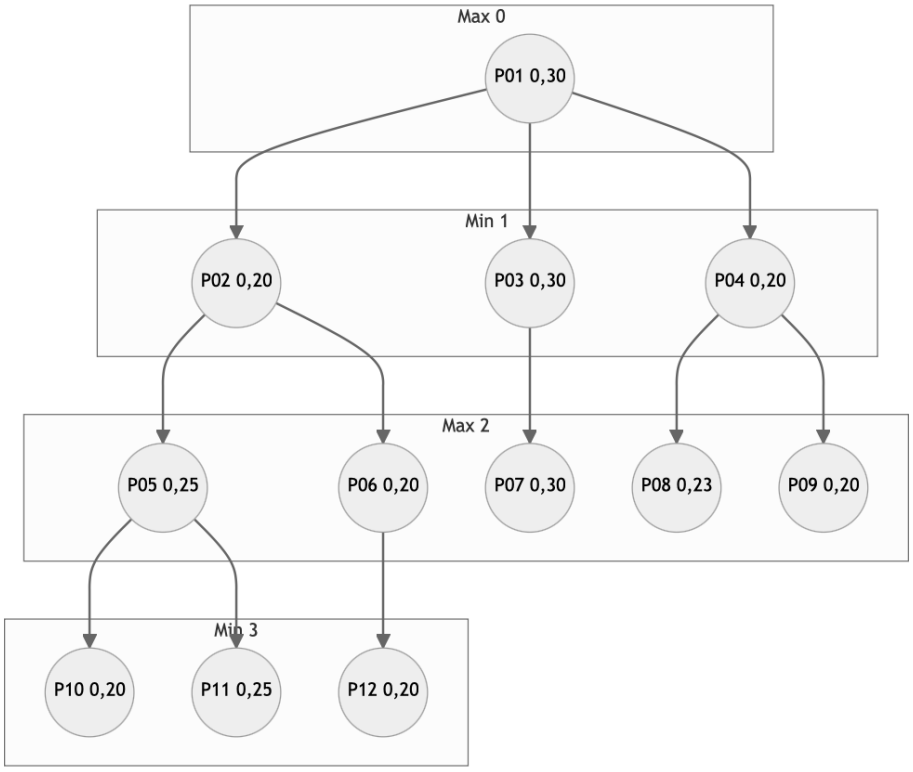


Figura 2.15: Cuarta fase del ejemplo de minimax aplicado al ajedrez

Finalmente, se calcula el valor del estado inicial, denominado P01. Este cálculo se realiza tomando el valor máximo entre los estados P02, P03 y P04, resultando en un valor de 0,30. Este valor indica que el estado descendiente que lo posee corresponde a la mejor jugada potencial. De esta manera, según el árbol de juego desarrollado, la transición al estado P03, o lo que es equivalente, la jugada Cf3, se identifica como la mejor opción.

Es importante recalcar que los árboles de juego generados por ordenadores pueden alcanzar un tamaño formidable, llegando incluso a contener más de un millón de estados, un número que eclipsa de manera significativa al número de estados en el ejemplo proporcionado anteriormente. Por lo tanto, se vuelve esencial explorar métodos para reducir el tamaño de este árbol. En una sección previa, se ha presentado cómo es posible dar prioridad a ciertos estados en función de su valor. Sin embargo, existe una técnica que puede disminuir de forma significativa el tamaño del árbol de juego, a veces incluso reduciéndolo a la mitad. Esta técnica se conoce como poda alfa-beta, y se puede considerar como una mejora del algoritmo Minimax. Este tema será discutido con mayor profundidad en la siguiente sección de este capítulo.

2.6. Poda alfa-beta

La poda alfa-beta es una técnica eficiente que reduce drásticamente la cantidad de estados que se visitan en el árbol de juego del algoritmo minimax. Su objetivo principal es evitar explorar ramas del árbol de juego que no conducirán a resultados mejores que los ya encontrados. Esta estrategia se divide en dos variantes: la poda alfa, aplicada en los niveles de maximización,

y la poda beta, utilizada en los niveles de minimización.

Para implementar la poda alfa-beta, se emplean dos valores: alfa (α) y beta (β). El valor de alfa representa el mejor resultado obtenido hasta el momento durante la fase de maximización, mientras que el valor de beta cumplirá la misma función en la fase de minimización.

Los valores alfa y beta se transmiten de los padres a los hijos en el árbol de juego, pero no en la dirección opuesta. Además, los valores alfa y beta también pueden ser heredados entre hermanos.

En el contexto de la poda alfa-beta, se considera que un estado es el padre de otro estado si el primero se encuentra en el nivel inmediatamente superior al segundo y existe un enlace que los conecta. De manera inversa, se dice que un estado es el hijo de otro estado si el hijo está en el nivel inmediatamente inferior al padre y hay un enlace entre ellos. El concepto de hermano está estrechamente relacionado con estas dos relaciones. Se dice que un nodo (estado) es el hermano de otro estado si comparten el mismo padre. En los árboles por definición un nodo solo puede tener un padre.

Al calcular el valor de alfa se utiliza la siguiente fórmula:

$$\alpha := \text{máx}(\alpha, \beta_{\text{hijo}})$$

Esta fórmula se ejecuta una vez por cada hijo de un estado de maximización. Cada vez que se ejecuta, el valor de alfa se actualiza. Inicialmente, se establece $\alpha = -\infty$, lo que significa que cualquier número será mayor que este valor. Sin embargo, es importante destacar que el valor inicial de alfa puede ser obtenido a partir de los hermanos y el padre, lo cual se explicará

detalladamente en un ejemplo posterior. Por lo tanto, no siempre será igual a $-\infty$.

Por otro lado, la fórmula para calcular el valor de beta es similar, pero en este caso se toma el mínimo y se utilizan los valores de alfa de los hijos:

$$\beta := \text{mín}(\beta, \alpha_{\text{hijo}})$$

Al igual que con alfa, esta fórmula se ejecuta la misma cantidad de veces que hijos tenga un estado de minimización. Si un estado de minimización no ha heredado el valor de su padre o de alguno de sus hermanos, se establece $\beta = +\infty$. Esto asegura que cualquier valor de alfa que se encuentre en los hijos será menor que el valor de beta.

En resumen, estas fórmulas permiten actualizar y mantener los valores de alfa y beta durante el proceso de búsqueda en el árbol de juego. La fórmula de alfa utiliza el máximo entre el valor actual de alfa y los valores de beta de los hijos, mientras que la fórmula de beta utiliza el mínimo entre el valor actual de beta y los valores de alfa de los hijos. Estas actualizaciones son esenciales para llevar a cabo las podas alfa-beta y optimizar la búsqueda del algoritmo minimax en el árbol de juego.

El proceso de poda ocurre de esta manera: si en algún punto mientras se explora el árbol, el valor de alfa es mayor o igual que beta, entonces sabemos que las próximas ramas o nodos que se iban a explorar en ese camino no van a cambiar el resultado, ya que alfa representa la mejor jugada que el jugador maximizador ya tiene garantizada, y beta representa la mejor jugada que el jugador minimizador ya tiene garantizada. Como resultado, no tiene sentido seguir explorando ese camino y es "podado".

La comprensión de la poda alfa-beta puede resultar desafiante sin ejemplos prácticos. Por lo tanto, a continuación se presentan tanto un ejemplo genérico como un ejemplo específico relacionado con el ajedrez, con el objetivo de facilitar la comprensión del algoritmo.

2.6.1. Ejemplo genérico

El siguiente ejemplo aporta una extensión al presentado en el capítulo previo, con una sutil pero importante variante. En esta ocasión, cada nodo albergará no solo su valor propio, sino también un valor denominado α o β, de acuerdo con la alternancia correspondiente entre niveles: los niveles pares llevarán el valor β y los impares el valor α. La numeración de los niveles comienza desde cero.

Este esquema conlleva una particularidad: no será factible explorar el árbol nivel por nivel como se hizo en los ejemplos anteriores, pues necesitaremos hacer referencia a los valores α y β de los nodos hermanos.

Sin más preámbulos, se presenta el ejemplo concreto en la siguiente figura, a la que haremos referencia como Figura 2.16:

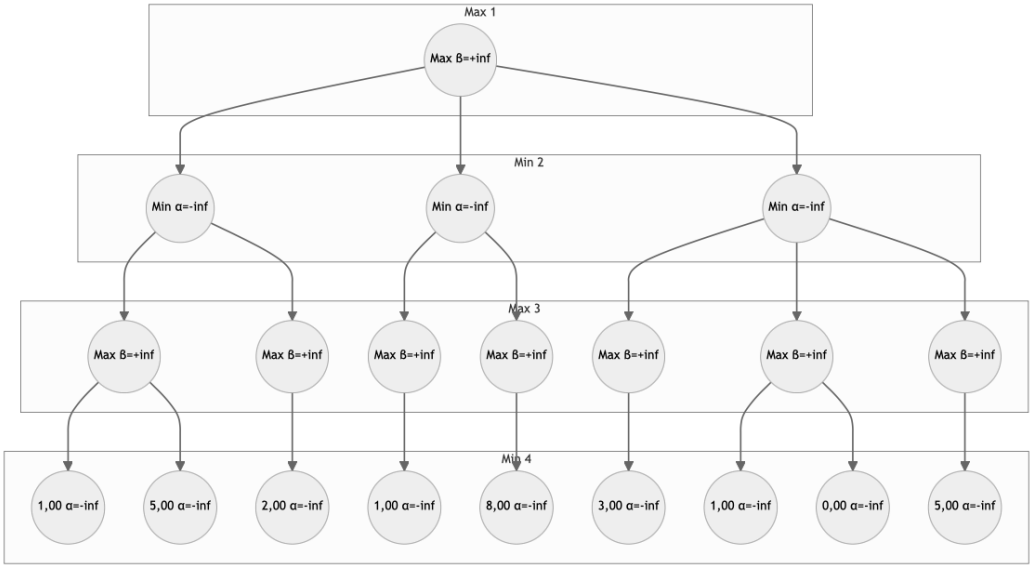


Figura 2.16: Árbol de juego genérico con poda alfa-beta

Iniciaremos nuestra exploración desde el extremo izquierdo del árbol. De haber optado por comenzar desde la derecha, hubiéramos obtenido un árbol ligeramente diferente, pero la evaluación en el nodo final hubiera permanecido invariable. Por tanto, ambas estrategias son válidas. Asimismo, si alteramos el orden de visita entre los nodos hermanos, el resultado final se mantiene, aunque el número de nodos a procesar puede variar. Lamentablemente, la determinación del orden óptimo para ordenar los nodos —de modo que minimice la cantidad de procesamiento— es imposible sin desarrollar el árbol en su totalidad. Tras la primera fase de resolución, obtendremos el árbol que se muestra en la Figura 2.17.

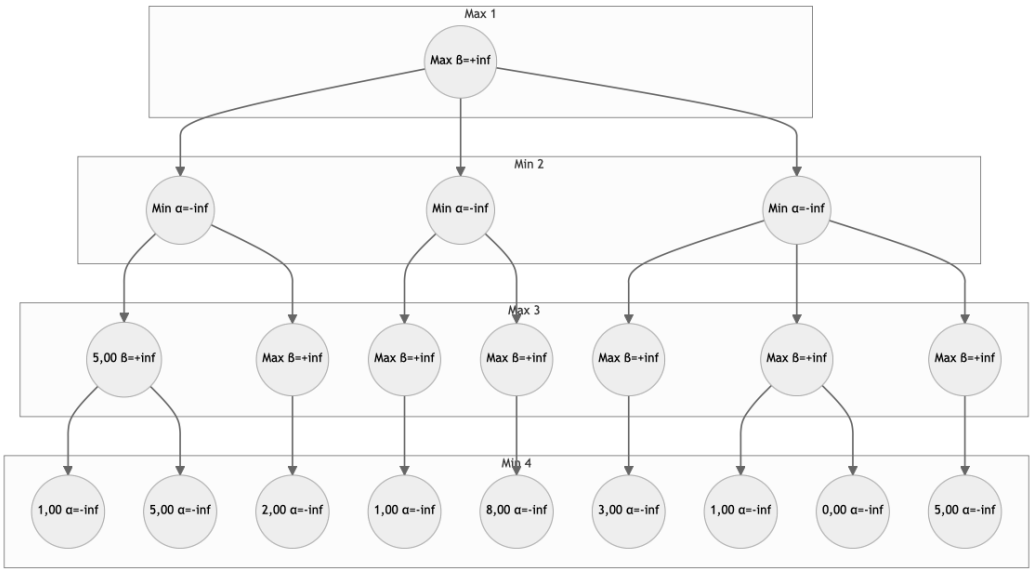


Figura 2.17: Primera fase de resolución de árbol de juego genérico con poda alfa-beta

El nodo procesado, que es el primer elemento del nivel 2, tiene asignado un valor heurístico de 5. Este valor se obtiene al ser el máximo entre 1 y 5. Además, como el nodo es de maximización y es el primer nodo procesado en ese nivel, se le asigna un valor de beta de $+\infty$ (infinito positivo). A continuación, al explorar el nodo hermano de este, se obtiene el árbol representado en la Figura 2.18, correspondiente a la segunda fase de resolución del árbol de juego genérico con poda alfa-beta.

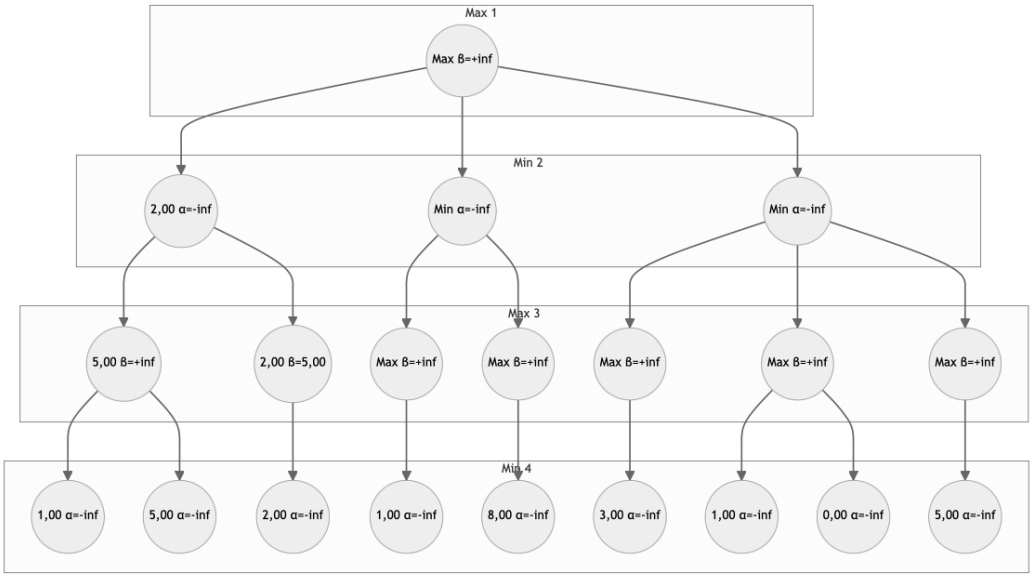


Figura 2.18: Segunda fase de resolución de árbol de juego genérico con poda alfa-beta

En este caso, lamentablemente, no se ha obtenido una ventaja significativa, ya que se ha tenido que procesar el mismo número de nodos que si se hubiera utilizado el algoritmo minimax. Esto suele ser común al principio del árbol.

La beta del hermano del nodo procesado en el primer paso tiene un valor de 5, ya que se ha actualizado según la fórmula y ha tomado el valor de 5 de su hermano. A continuación, se asciende un nivel y se llega al nodo padre de estos, el cual tiene un valor de 2. El valor de α de este nodo es menos infinito, dado que es el primer nodo procesado de entre sus hermanos.

En la siguiente fase, se procede a procesar el segundo hijo del

nodo inicial, como se muestra en la Figura 2.19.

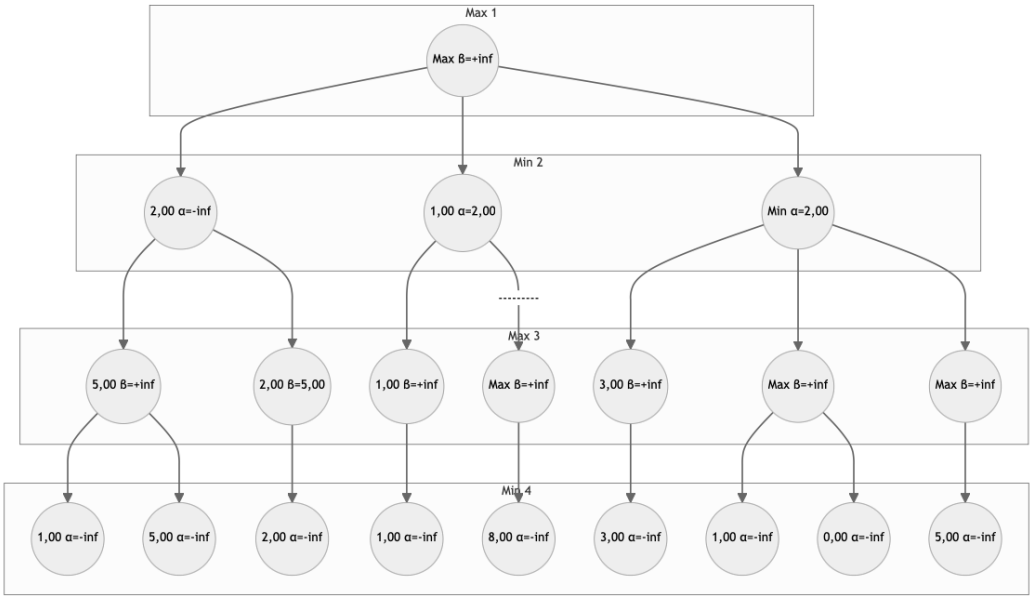


Figura 2.19: Tercera fase de resolución de árbol de juego genérico con poda alfa-beta

En la etapa inicial, se procede a procesar el hijo izquierdo del segundo hijo del nodo inicial. Este nodo obtiene un valor de 1. Dado que la α actual es 2, se activa la poda alfa, lo que implica que no será necesario procesar ningún otro hijo de este nodo. Por esta razón, no es procesado y muestra con un corte esa parte del árbol.

La lógica detrás de esta poda es que no se puede obtener un valor mejor que 1 en este nodo, ya que al ser un nodo de minimización, siempre se elegirá el valor 1 o uno menor en otro

hijo. Además, dado que el valor de α es 2 y este es mayor que 1, no se incrementa.

En la última etapa, se procede a procesar el hijo más a la derecha del nodo inicial, lo que resulta en los resultados mostrados en la Figura 2.20.

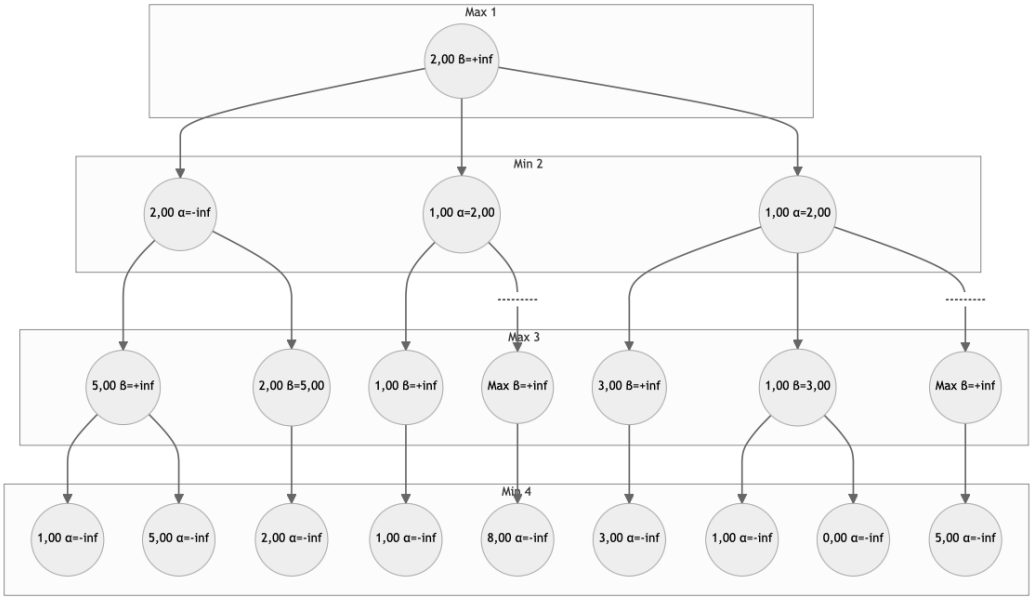


Figura 2.20: Última fase de resolución de árbol de juego genérico para mostrar la poda alfa-beta

En el primer hijo del nodo procesado, se asigna un valor de beta de 3 a su hermano. Al explorar los hijos de ese hermano y obtener los valores de 0 y 1, se continúa evaluando los otros hijos solo si ninguno de ellos es mayor que 3. En este caso, como el valor máximo obtenido es 1 en el nodo central del nodo

procesado, se actualiza el valor de beta a 1.

Al subir al siguiente nivel, se recibe el valor de 1 desde ese nodo, y al ser el valor de alfa 2 proveniente de su hermano, se realiza una poda alfa y no es necesario continuar evaluando los otros hijos.

Finalmente, se obtiene el valor del nodo inicial, que es el máximo entre los valores de sus hijos, lo que resulta en un valor de 2. El valor de beta es $+\infty$ ya que no tiene ningún hermano ni padre que establezca un valor límite.

2.6.2. Ejemplo aplicado al ajedrez

En el caso del ajedrez, el funcionamiento del algoritmo alfa-beta es exactamente igual al ejemplo genérico anteriormente presentado. Por lo tanto, en esta sección nos centraremos en las implicaciones y la importancia del orden en el que se visitan los nodos.

La Figura 2.21 muestra el ejemplo específico que se va a tratar, el cual es el mismo que se presentó en la Figura 2.15 en la sección anterior, relacionada con el algoritmo minimax.

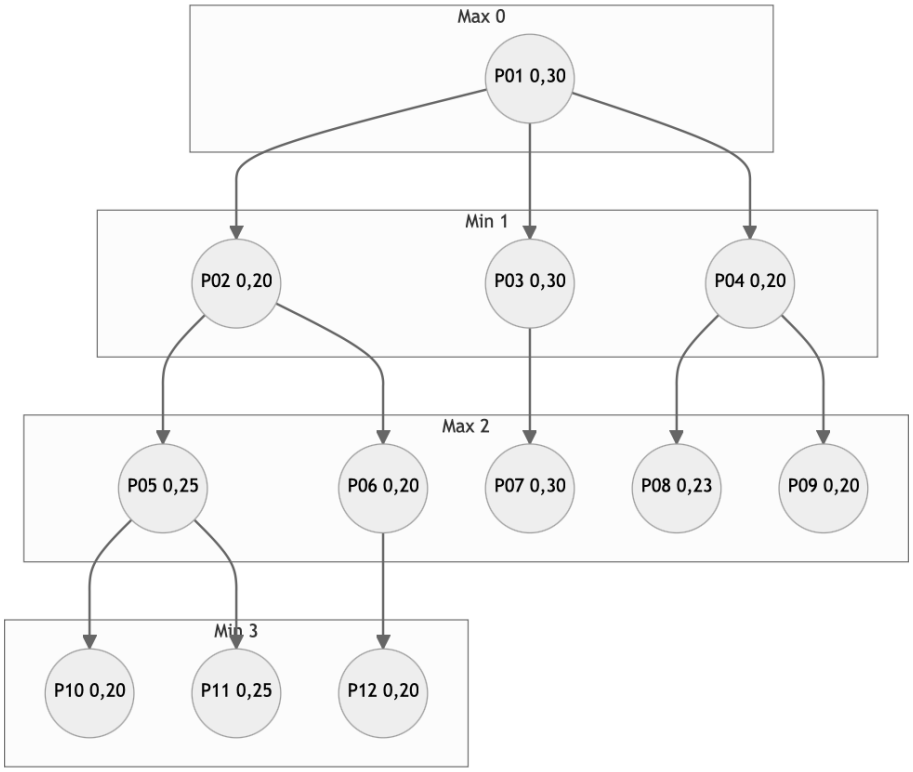


Figura 2.21: Ejemplo de poda alfa-beta aplicado al ajedrez

Al crear los órdenes para recorrer los árboles, es importante recordar dos reglas básicas:

- Un nodo solo puede ser procesado si se han procesado previamente sus hijos.
- Si se sigue un recorrido de izquierda a derecha, se procesan primero los nodos situados más a la izquierda que aún no han sido procesados, y viceversa.

A continuación, se utilizará el recorrido habitual comenzando desde la izquierda y avanzando hacia la derecha. Por lo tanto, el orden de procesamiento será el siguiente:

$$P10 \rightarrow P11 \rightarrow P05 \rightarrow P12 \rightarrow P06 \rightarrow P02$$

$$P07 \rightarrow P03 \rightarrow P08 \rightarrow P09 \rightarrow P04 \rightarrow P01$$

La Figura 2.22 muestra el árbol usando el recorrido de izquierda-derecha descrito anteriormente.

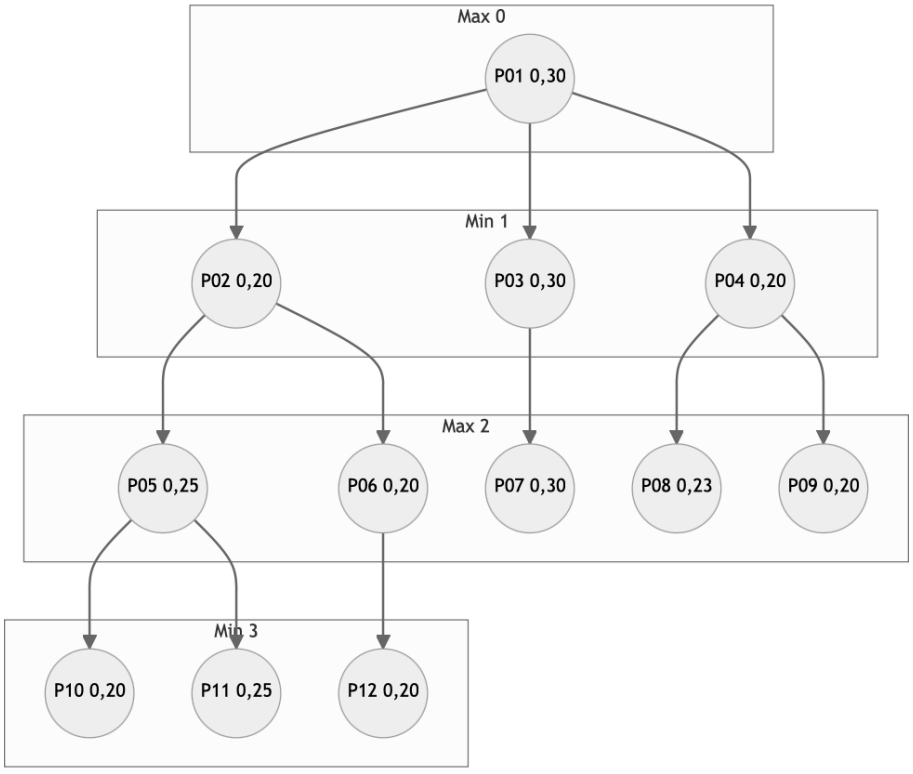


Figura 2.22: Ejemplo de poda alfa beta aplicado al ajedrez usando orden izquierda-derecha

Como se puede observar, en este orden de recorrido no se descarta ningún nodo. Esto indica que no se obtendría ninguna

mejora al utilizar el complejo algoritmo de poda alfa-beta en lugar del sencillo algoritmo minimax.

En el siguiente caso, se cambiará el orden y se seguirá un recorrido de derecha a izquierda sobre el mismo árbol. Por lo tanto, el nuevo orden de procesamiento será el siguiente:

$$P09 \rightarrow P08 \rightarrow P04 \rightarrow P07 \rightarrow P03 \rightarrow P12$$

$$P06 \rightarrow P11 \rightarrow P10 \rightarrow P05 \rightarrow P02 \rightarrow P01$$

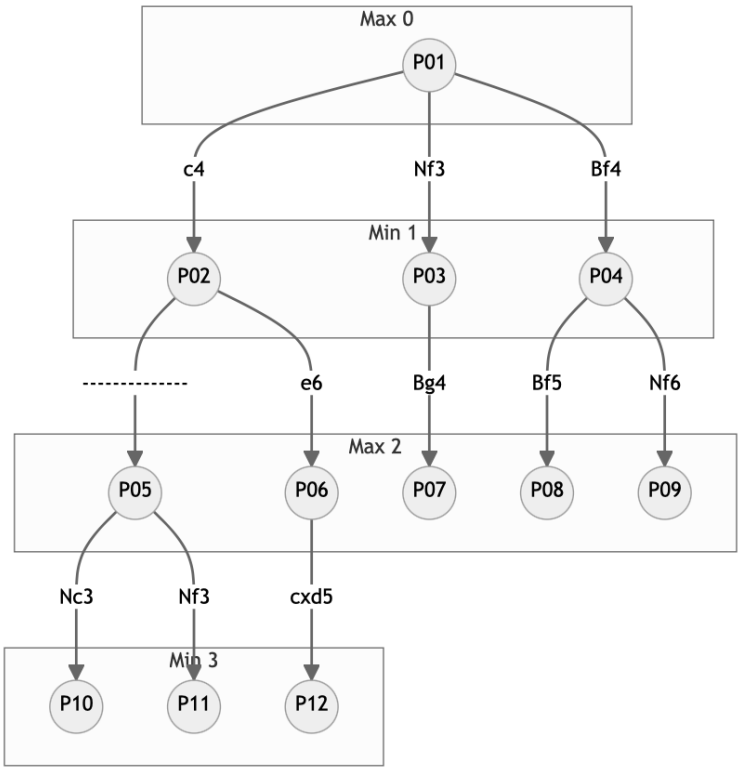


Figura 2.23: Ejemplo de poda alfa beta aplicado al ajedrez usando orden derecha-izquierda

Con el cambio de orden, se ha logrado una mejora sustancial. Ahora, no es necesario visitar tres de los nodos del árbol original (P05, P10 y P11). Esto se debe a que cuando el nodo P02 recibe el valor del nodo P06, que es 0.20, tiene un valor de alfa más alto, lo que activa la poda alfa y no es necesario generar más nodos hijos para el nodo P02.

Como conclusión de ambos ejemplos, se puede afirmar que la poda alfa-beta puede reducir significativamente el número de nodos que se deben procesar. En el último ejemplo, se eliminan 3 nodos de un total de 12, lo que representa un 25 % de los nodos eliminados. Cuanto más grande sea el árbol, mayor será el porcentaje de nodos que se pueden eliminar utilizando este procedimiento, llegando a niveles superiores al 50 % de eliminación en algunos casos.

Otra conclusión importante es la relevancia del orden en el que se procesan los nodos. Para seleccionar este orden, también se pueden utilizar heurísticas, procesando primero aquellos nodos cuya heurística tenga un valor mayor. Esto puede ayudar a mejorar aún más la eficiencia de la poda alfa-beta al reducir la cantidad de nodos a considerar en cada nivel del árbol.

2.7. Representación del tablero

Tras explorar los mecanismos de búsqueda en juegos de inteligencia artificial, es crucial examinar otro componente esencial: la representación del tablero de juego. Este elemento cumple con dos funciones primordiales en el contexto de la búsqueda. La primera de ellas es la generación de todas las jugadas posibles a partir de la posición actual en el juego. En segundo lugar, se encarga de las transiciones de una jugada a la siguiente, es

decir, provee la nueva posición en el tablero basada en la jugada y el estado previo del mismo. Ambas funciones son invocadas repetidamente en el árbol de juego. Dada la relevancia de estas operaciones y el hecho de que la calidad de una jugada depende tanto de la profundidad de expansión del árbol como del tiempo que esto conlleva, es esencial que ambas funciones sean implementadas de la manera más eficiente posible.

Para una implementación eficaz de estas funciones, es imperativo considerar cómo se representa el tablero de juego en la memoria de la computadora. Existen diversos métodos para ello, cada uno con sus propios beneficios y limitaciones. Podemos agrupar estos métodos en tres categorías generales: los métodos centrados en las piezas, los métodos centrados en el tablero y los métodos híbridos [8]. La nomenclatura de estas categorías es bastante descriptiva.

Los métodos centrados en las piezas se enfocan en representar las piezas de manera directa, manteniendo listas u otras estructuras de datos con la información asociada a cada pieza y su posición en el tablero. Por otra parte, los métodos centrados en el tablero se centran en examinar cada casilla del tablero de manera individual para determinar si contiene una pieza y, en caso afirmativo, qué tipo de pieza es. Finalmente, los métodos híbridos son aquellos que combinan aspectos de ambos enfoques.

Aparte de la información del propio tablero o las piezas, es necesario almacenar información adicional, como la posibilidad de realizar enroques, el número de veces que la posición actual se ha repetido, la cantidad de movimientos sin capturas ni avances de peón, entre otros. Este tipo de información es relativamente sencilla de representar, por lo que no se hará mucho hincapié en ella, centrándonos más en la información del tablero y las piezas.

Antes de discutir las técnicas principales, es necesario establecer la diferencia entre las jugadas pseudolegales y las jugadas legales. Las jugadas legales son todas aquellas que se pueden realizar en una posición de acuerdo con las reglas del ajedrez. Las jugadas pseudolegales, por otro lado, incluyen las jugadas legales y aquellas que no se pueden realizar porque dejarían al rey bajo ataque y, por tanto, serían ilegales según las reglas del ajedrez. Es importante considerar estos grupos por separado debido a que las jugadas pseudolegales son más fáciles de calcular en comparación con las jugadas legales. Comprobar si el rey está en jaque después de un movimiento implica un coste computacional adicional significativo. Para solucionar este problema, se pueden generar los movimientos uno por uno, verificando si el rey queda en jaque antes de realizar cada movimiento. Otra opción es permitir el movimiento pseudolegal y, si el rival tiene la oportunidad de capturar al rey en su turno, se invalida ese movimiento.

A continuación, comenzaremos con un resumen breve de las principales técnicas de representación del tablero y después profundizaremos en las más relevantes:

- **Matriz 2D:** Este método centrado en el tablero emplea una matriz de 8×8 donde un valor 0 en una posición de la matriz indica que la casilla correspondiente está vacía. A cada tipo de pieza se le asigna un número; por ejemplo, a los peones se les asigna un 1, a los caballos un 2, etc. Si la pieza es blanca, se le asigna un valor positivo; si es negra, un valor negativo. Por ejemplo, si al alfil se le asigna el número 3, un alfil blanco sería 3 y un alfil negro -3.

- **Lista de piezas:** Esta técnica centrada en las piezas mantiene una lista de todas las piezas en juego. Cada entrada

de la lista contiene el tipo de pieza, el color y la casilla en la que se encuentra. Por ejemplo, para un alfil blanco en la casilla a4, la lista almacenará una entrada con el alfil, el color blanco y la casilla a4.

- ***Bitboard*:** Esta es la representación más comúnmente utilizada y está centrada en el tablero. La idea principal es usar *bitsets* [1] para representar matrices de 8×8. Dado que solo se pueden usar 0s y 1s, es necesario usar una matriz por cada tipo de pieza y color, lo que da un total de 12 matrices. Si la matriz de una pieza tiene un 1 en una determinada posición, esa casilla contendrá esa pieza; si, por el contrario, tiene un 0, no habrá una pieza de ese tipo en esa casilla. A pesar de que pueda parecer menos eficiente que la matriz de 2D, al tener que usar 12 matrices en lugar de una, este método tiene una serie de ventajas que se discutirán más adelante.

- **Vectores de ataque:** Este método, centrado en el tablero, representa los movimientos de las piezas como vectores. Cada pieza tiene asignados una serie de vectores según su movimiento. Por ejemplo, la dama combina los vectores de las torres y los alfiles. Gracias a los vectores, se pueden generar los movimientos. Cuando se generan movimientos, la pieza se desplaza usando el vector hasta encontrar un obstáculo, y se consideran como movimientos todas las casillas que haya pasado. Para el caballo, peón y rey, será necesario considerar sus peculiaridades en los movimientos.

[1] Un *bitset* es una estructura de datos simple que se utiliza para representar un conjunto de elementos, como números o valores booleanos. Funciona asignando un solo bit (0 o 1) a cada elemento en el conjunto, lo que permite verificar la pertenencia de un elemento en tiempo constante.

De todas las técnicas descritas, solo abordaremos en profundidad la técnica *bitboard*. Esto se debe a que una de sus variantes, el *magic bitboard*, es una de las más utilizadas en la actualidad. Además de las mencionadas, existen muchas más representaciones que funcionan mejor o peor dependiendo de la estructura del propio módulo e incluso del lenguaje de programación utilizado para crear este.

El concepto *Bitboard* se apoya en el uso de los llamados *bitsets*. Un *bitset* es un conjunto de bits, cada bit puede tomar únicamente dos valores: 0 o 1. De esta forma, nos permite representar si una casilla está ocupada por un tipo de pieza (incluyendo el color) o no.

Uno de los atributos más relevantes de los *bitset* es su capacidad para ser representados mediante un número entero. Esto resulta particularmente útil en situaciones en las que necesitamos manipular o representar información en forma de bits.

Consideremos un *bitset* de cuatro bits como ejemplo. En este caso, el número 3 se representa como "0011". Por tanto, un *bitset* de cuatro posiciones puede representar cualquier número desde 0 hasta 15. Podemos convertir un *bitset* a su equivalente numérico mediante la fórmula:

$$n = \sum_{i=0}^{k-1} b_i * 2^i$$

En la fórmula, n es el número en formato decimal, k es el número de bits que tiene el *bitset*, y b_i es el bit en la posición i contando desde la derecha. Así, el primer bit sería el último, el segundo bit sería el penúltimo, y así sucesivamente. En consecuencia, un *bitset* de k posiciones puede representar números

desde 0 hasta $2^k - 1$. Esto implica que un *bitset* puede representar 2^k números en total, aunque al empezar en 0 no llega a incluir el número 2^k.

Haciendo uso de este principio, consideremos la aplicación en el juego del ajedrez. Sabemos que un tablero de ajedrez tiene 64 casillas, por lo que requeriríamos un *bitset* de tamaño 64 para su representación. Afortunadamente, los ordenadores modernos están optimizados para trabajar de forma eficiente con enteros de 64 bits, ya que los soportan de manera nativa. Por lo tanto, con tan solo 12 enteros, ¡podríamos representar todos los posibles tableros de ajedrez! En términos de eficiencia en el uso de memoria (medida en términos de *bits* utilizados), esta no es necesariamente la mejor opción, pero el verdadero valor de este método reside en su capacidad para generar rápidamente los movimientos posibles desde una posición determinada mediante el uso de operaciones lógicas.

Los ordenadores realizan dos tipos de operaciones básicas: aritméticas y lógicas. Las operaciones aritméticas son operaciones estándar como suma, resta, multiplicación y división, mientras que las operaciones lógicas operan sobre el álgebra booleana. Este tipo de álgebra sólo contiene dos valores, 0 y 1, y constituye la base fundamental de la computación. Las operaciones aritméticas normalmente requieren un gran número de operaciones lógicas para su ejecución, por lo que las operaciones lógicas suelen ser más rápidas [2].

Las operaciones lógicas suelen realizarse sobre una o dos variables, aunque son fácilmente extensibles a un mayor número de variables. Para representar estas operaciones, se utilizan las llamadas puertas lógicas. Estas toman una o más variables como

[2]Esto se refiere a la electrónica digital

entrada y generan una única salida. Tanto las variables como la salida pueden tomar los valores 0 o 1.

Las principales puertas lógicas para una variable son las que se muestran en la Tabla 2.1.

Valores	NOT	Identidad
0	1	0
1	0	1

Tabla 2.1: Puertas lógicas de una variable

La puerta lógica NOT invierte el valor de la entrada. Si la entrada es 0, la salida será 1 y viceversa. Por otro lado, la puerta de identidad simplemente deja el valor de entrada sin cambios.

A continuación, consideremos las puertas lógicas de dos variables, que son algo más complejas pero pueden ser extendidas a más de dos variables. La Tabla 2.2 muestra las principales puertas lógicas de dos variables.

Valores	OR	AND	XOR
00	0	0	0
01	1	0	1
10	1	0	1
11	1	1	0

Tabla 2.2: Puertas lógicas de dos variables

La puerta lógica OR produce una salida de 1 siempre que al menos una de las entradas sea 1. Este comportamiento se mantiene si se extiende a más de dos variables: la salida será 1 si al menos una de todas las variables es 1.

La puerta lógica AND produce una salida de 1 únicamente cuando todas las entradas son 1. Esta lógica también se aplica cuando hay más de dos entradas.

Para concluir nuestra exploración de las puertas lógicas, la puerta XOR representa un caso de uso peculiar. Se puede concebir como una versión más "estricta" de la puerta OR. Similarmente, la puerta XOR opera al igual que una puerta OR, con una excepción significativa: en el caso de que todas las entradas sean 1, la salida será 0.

Cuando expandimos esta puerta a más de dos entradas, se pueden seguir dos métodos: devolverá 1 si el número de entradas que son 1 es impar, y resultará verdadera si y solo si hay exactamente una entrada que es 1. Sin embargo, es importante destacar que ninguno de estos enfoques se considera universalmente aplicable. Desde una perspectiva electrónica, una puerta XOR con más de dos variables no se implementa generalmente.

Existen varias otras puertas lógicas además de las ya mencionadas. Estas pueden derivarse mediante la combinación de las puertas previamente descritas. Las puertas NAND y NOR merecen una mención especial. Son el resultado de la combinación de las puertas OR y AND con una puerta NOT, respectivamente. En otras palabras, toman el valor de las puertas OR y AND y luego lo invierten. La importancia de estas puertas radica en su consideración como "universales". Esto significa que, utilizando exclusivamente puertas NAND o NOR, podemos recrear todas las demás puertas lógicas, incluso aquellas con una sola entrada. Estas operaciones desempeñan un papel fundamental para lograr que los *bitboards* sean eficientes.

Para ilustrar estos conceptos, mostraremos un ejemplo de cómo se convierte un tablero de ajedrez en su representación de *bitboard*.

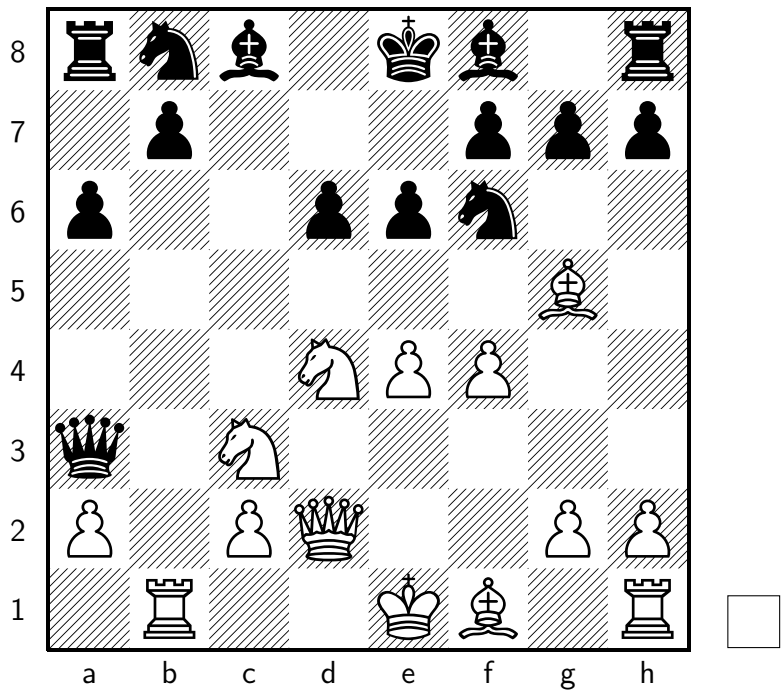


Figura 2.24: Tablero de ejemplo para bitboard

En esta representación, los bits se originan en la esquina inferior derecha, lo que implica que el bit 0 (ubicado al final) corresponde a la casilla h1. Los bits se asignan por columnas, y una vez que se han asignado todos los bits de una columna, se procede a la próxima fila comenzando por la columna inicial. Por lo tanto, el bit 1 corresponde a g1, el bit 2 a f1 y el bit 8 a h2, y así sucesivamente.

Para empezar, representaremos los peones blancos. Cada 8 bits se separarán en líneas diferentes para simplificar la lectura. La representación resultante sería la siguiente:

```
00000000
```

```
00000000
00000000
00000000
00001100
00000000
10100011
00000000
```

Las filas 4 y 2, donde se encuentran los peones blancos, son las únicas que contienen unos. A continuación, mostramos otro ejemplo con las torres negras.

```
10000001
00000000
00000000
00000000
00000000
00000000
00000000
00000000
```

Las torres negras se mantienen en sus posiciones iniciales. Este mismo procedimiento se utilizaría para representar el resto de las piezas.

Para generar los movimientos, se emplea una técnica altamente eficiente conocida como *magic bitboards*. Esta técnica se utiliza para calcular los movimientos de los alfiles y las torres, y por ende, los de la dama. Los caballos, el rey y los peones son más sencillos de calcular, por lo que no es necesario usar esta técnica con ellos. Esto se debe a que los movimientos de estas piezas no están "obstruidos" por otras piezas.

La técnica *magic bitboards* consiste en multiplicar el tablero actual, representado por su *bitset*, por ciertos "números mágicos". Para cada posición de un alfil o una torre, y una disposición de piezas que pueden bloquear su trayectoria, existe un número mágico que, multiplicado por el *bitset* que contiene todas las piezas del tablero (este se calcula haciendo una OR entre todos los *bitsets*), nos proporciona todos los movimientos posibles del alfil o la torre en un nuevo *bitset*. Calcular estos números mágicos implica un alto coste computacional, pero solo es necesario hacerlo una vez y sirve para cualquier posición. Este es el motivo por el que esta técnica ofrece un rendimiento tan notable.

Para resumir, existen una multitud de técnicas que permiten representar tableros de ajedrez, focalizándose principalmente en la generación de movimientos y en las transiciones entre distintos estados del juego. Una de las técnicas más destacadas y ampliamente empleadas es la de los *magic bitboards*. Este método es conocido por su excepcional eficiencia en términos de rendimiento y velocidad.

Además de los *magic bitboards*, existen otras representaciones del tablero que son más adecuadas para ciertas aplicaciones específicas. Por ejemplo, el método de *one-hot encoding*, que comparte similitudes con la representación *bitboard*, y la representación HalfKP, son ambos muy utilizados en contextos de redes neuronales debido a sus propiedades particulares.

Por otro lado, la notación FEN (Forsyth-Edwards Notation) es muy utilizada para almacenar y recuperar posiciones de tablero de ajedrez, dada su capacidad de representar de manera concisa el estado completo de un juego.

Es importante señalar que todas estas técnicas serán expuestas con mayor detalle en capítulos posteriores de este libro, brin-

dando al lector una comprensión más profunda de sus características y aplicaciones en diferentes contextos.

2.8. Técnicas avanzadas en el árbol del juego

En esta sección, nos enfocaremos en las diferentes técnicas empleadas para optimizar la búsqueda en la poda alfa-beta, una estrategia esencial para la eficiencia en juegos de IA. Estas técnicas están diseñadas para reducir el tamaño del árbol de juego explorado, permitiendo una exploración más precisa y efectiva de los nodos relevantes para el problema en cuestión. Aunque existe una multitud de técnicas disponibles, este libro se centrará solo en las más significativas y eficaces. Sin embargo, es importante tener en cuenta que estas técnicas requieren ciertos recursos computacionales, lo que implica que es crucial ponderar el beneficio obtenido frente al coste computacional asociado.

El primer aspecto crucial que consideraremos es el orden en que se exploran las jugadas. Uno de los órdenes de exploración más comunes se detalla a continuación:

- Capturas o promociones
- Movimientos hacia delante
- Movimientos laterales
- Movimientos hacia atrás

La lógica general detrás de este orden de exploración es dar prioridad a aquellos movimientos que pueden resultar en cambios significativos en la posición en el tablero. En consecuencia, las capturas y promociones, que alteran el número de piezas

en el tablero, se consideran primero. Una vez que estos movimientos se han examinado, se analizan los "movimientos hacia adelante", que son aquellos que avanzan las piezas hacia el territorio enemigo. Estos movimientos suelen ser preferibles a los que mantienen a las piezas en la misma fila o que retroceden hacia su propio campo. Esta lógica se aplica igualmente a los movimientos laterales y hacia atrás. Además, si ya hemos explorado parcialmente una posición (hasta una cierta profundidad, pero planeamos explorar más allá de esta posteriormente), se pueden aplicar técnicas más avanzadas para seleccionar cuál de estas opciones continuará en la búsqueda.

El siguiente problema es determinar hasta qué profundidad se debe realizar la búsqueda. Una opción podría ser establecer una profundidad fija para terminar la búsqueda. Sin embargo, este enfoque tiene una desventaja importante: ¿qué sucede si justo después de esa profundidad se realiza una jugada que cambia la posición, típicamente una captura o promoción? Para abordar este problema, se debe utilizar una técnica llamada *quiescence search*, que se centra en detener la exploración de una posición cuando esta es "tranquila". Una posición se considera tranquila generalmente cuando no es posible realizar ninguna captura o promoción. Salvo en casos excepcionales, es prudente terminar la búsqueda en una de estas posiciones tranquilas y continuar en las que sean más inestables.

Otra técnica sencilla pero muy efectiva para reducir el número de jugadas a explorar es la poda del movimiento nulo (*null move pruning*). El concepto es simple: se realiza una jugada "nula", en la que se cede el turno sin realizar ninguna acción. Aunque esta estrategia no se puede implementar en una partida de ajedrez real, se ha demostrado que es muy efectiva en la optimización

de la exploración. Si la jugada anterior a la jugada nula no ha alterado la valoración de la posición, se puede inferir que esa jugada no es relevante y, por lo tanto, se puede descartar.

Otra técnica importante es la utilización de la tabla de transposiciones. Esta se orienta a evitar calcular múltiples veces la misma posición. En ajedrez, se puede llegar a la misma posición con diferentes órdenes de jugadas, lo que podría implicar calcular varias veces la misma posición. La tabla de transposición se encargará de almacenar las posiciones visitadas y devolverá el valor de ese estado si se vuelve a pasar por esa posición. Para facilitar la comparación y búsqueda de posiciones, se utiliza un proceso conocido como *hashing*, que convierte cada posición en un número con una probabilidad muy baja de repetición.

Además, los programas de ajedrez se benefician de ayuda adicional para las primeras jugadas de las partidas y para los finales de estas. Para el inicio de las partidas (aperturas), cuentan con los libros de aperturas que incluyen las jugadas para las posiciones que se dan en las primeras jugadas habitualmente. Estas jugadas, obtenidas de partidas de ajedrez jugadas por humanos, son especialmente útiles para los ordenadores en las posiciones iniciales complejas donde el desarrollo del árbol de juego puede suponer un coste elevado. En lo que respecta a los finales de las partidas, se utilizan las tablas Syzygy. Estas tablas contienen la solución del ajedrez (saber desde cualquier posición cuál es el resultado inevitable) para 7 piezas en el tablero o menos. Esto significa que, con estas tablas, si la posición tiene menos de 7 piezas, el ordenador puede consultarlas para saber cuál es el mejor movimiento sin necesidad de realizar una búsqueda.

Un reto significativo de la técnica de búsqueda es el denominado "efecto horizonte". Este se produce cuando es inevitable

un resultado en una posición, pero debido a la limitación de la profundidad es imposible llegar a esa conclusión desarrollando un árbol de juego no completo. En los casos más sencillos, donde las jugadas que llevan a la conclusión son capturas, se puede evitar este problema utilizando la técnica de *quiescence search*, pero en casos más complejos, como fortalezas (a pesar de encontrarse con ventaja material no es posible progresar) o series de jugadas más complejas, este efecto puede ser un problema.

Además, los programas de ajedrez pueden tener dificultades para evaluar estratégicamente una posición, ya que muchas jugadas en avance hacen que el tamaño del árbol de juego sea inabarcable. Las técnicas como el movimiento nulo pueden aliviar en cierta medida este problema, pero no lo eliminan por completo.

En el siguiente capítulo, analizaremos el uso del aprendizaje reforzado, una técnica que puede mejorar el rendimiento de la IA frente a estos problemas anteriores.

Capítulo 3

Aprendizaje profundo reforzado

En esta sección, se explorará el fascinante campo del aprendizaje reforzado profundo, que ha ganado notoriedad gracias a su aplicación en una variedad de tareas complejas, desde la conducción de automóviles autónomos hasta el dominio de juegos de estrategia, como el ajedrez. Para una comprensión completa, es crucial introducir una serie de conceptos esenciales que se relacionan entre sí de manera integral.

Comenzaremos con la noción de "recompensa", un concepto fundamental en el aprendizaje reforzado. En este contexto, la recompensa representa una señal de *feedback* que el agente recibe tras realizar una acción en un entorno determinado. El objetivo de un agente es maximizar la suma total de las recompensas a lo largo del tiempo, lo que se denomina la "recompensa acumulada".

A partir de la idea de recompensa, podemos definir la "política". En aprendizaje reforzado, la política se refiere a la estrategia que sigue el agente para seleccionar las acciones basándose en el estado del entorno. Por tanto, se puede entender como el comportamiento del agente en un momento dado.

A continuación, el "modelo del entorno" se describe como

la representación que el agente tiene del entorno. Este modelo se utiliza para prever cómo el entorno cambiará con base en las acciones del agente. Los algoritmos de aprendizaje reforzado pueden ser *model-free*, donde el agente aprende directamente la política óptima sin un modelo del entorno, o *model-based*, donde el agente aprende un modelo del entorno y lo usa para planificar.

A partir de estos conceptos, surgieron diversos algoritmos de aprendizaje reforzado, incluyendo REINFORCE, DQN (Deep Q-Learning), Actor-Crítico y MCTS (Monte Carlo Tree Search) por nombrar algunos. Cada uno de estos algoritmos presenta sus propias ventajas y desventajas en función de la tarea específica y las características del entorno, pero todos ellos juegan un papel en el desarrollo de AlphaZero.

A continuación, examinaremos el funcionamiento de las "redes neuronales", que son los bloques de construcción de los modelos de aprendizaje profundo. Las redes neuronales son una serie de algoritmos que intentan reconocer patrones subyacentes a través de la simulación del proceso de reconocimiento en el cerebro humano.

Finalmente, con una comprensión sólida de estos conceptos, podemos profundizar en AlphaZero, un algoritmo desarrollado por DeepMind. AlphaZero representa el apogeo del aprendizaje reforzado profundo, ya que combina estos conceptos de manera única para superar a los sistemas de inteligencia artificial más avanzados en juegos como el ajedrez, el shogi y el Go. La clave del éxito de AlphaZero radica en su capacidad para enseñarse a sí mismo a jugar estos juegos a un nivel experto, simplemente jugando contra sí mismo y utilizando el aprendizaje reforzado para mejorar continuamente su política de juego. Esto se logra mediante la aplicación de las redes neuronales para aprender la

política y el valor de cada posición del tablero, lo que permite a AlphaZero planificar estrategias sofisticadas y tomar decisiones óptimas en cada movimiento.

3.1. Recompensa

La recompensa es un elemento clave en el aprendizaje reforzado, ya que proporciona a la inteligencia artificial (IA) retroalimentación respecto a su rendimiento, señalándole si su actuación es correcta o si, por el contrario, deja mucho que desear. Cabe recordar que las recompensas se obtienen solo en los estados terminales, como sería el final de una partida de ajedrez. No obstante, la cantidad de estados no terminales suele ser mucho mayor, lo que obliga a evaluar correctamente el valor del estado actual.

Un método para llevar a cabo esta evaluación implica valorar los estados terminales a los que podría llegar la IA mediante la poda alfa-beta, presentada en el capítulo anterior. Sin embargo, debido a la inmensidad del árbol de búsqueda, esta estrategia no es factible. En el capítulo anterior también se discutió una solución a este problema: el uso de una heurística. En este contexto, se aplicará un procedimiento análogo, en el que la heurística se calculará a partir de la recompensa. A este valor se le denominará "recompensa esperada".

El cálculo directo de la recompensa esperada es inviable por el motivo previamente mencionado. No obstante, es posible obtener aproximaciones. Estas se calcularán a través de la simulación de numerosos episodios, un término genérico utilizado en el aprendizaje reforzado para referirse a todas las partidas del juego, como serían las partidas en el ajedrez. La estimación ini-

cial se mejora a medida que se realizan más episodios, afinando la estimación de la recompensa esperada.

Para determinar la recompensa esperada, es necesario sumar todas las recompensas obtenidas en los estados posteriores hasta la conclusión del episodio. En el caso del ajedrez, este proceso se simplifica ya que la recompensa es cero en todos los estados no terminales. Además, se puede aplicar un "descuento" a la recompensa, lo que significa que cuanto más tiempo se tarde en obtener una recompensa, menor será su valor. Así, una recompensa de 1 obtenida en el estado actual tendrá más valor en la recompensa esperada que una recompensa de 1 obtenida tres estados después.

A continuación se presenta un ejemplo para solidificar estos conceptos. La Figura 3.1 muestra el juego en cuestión.

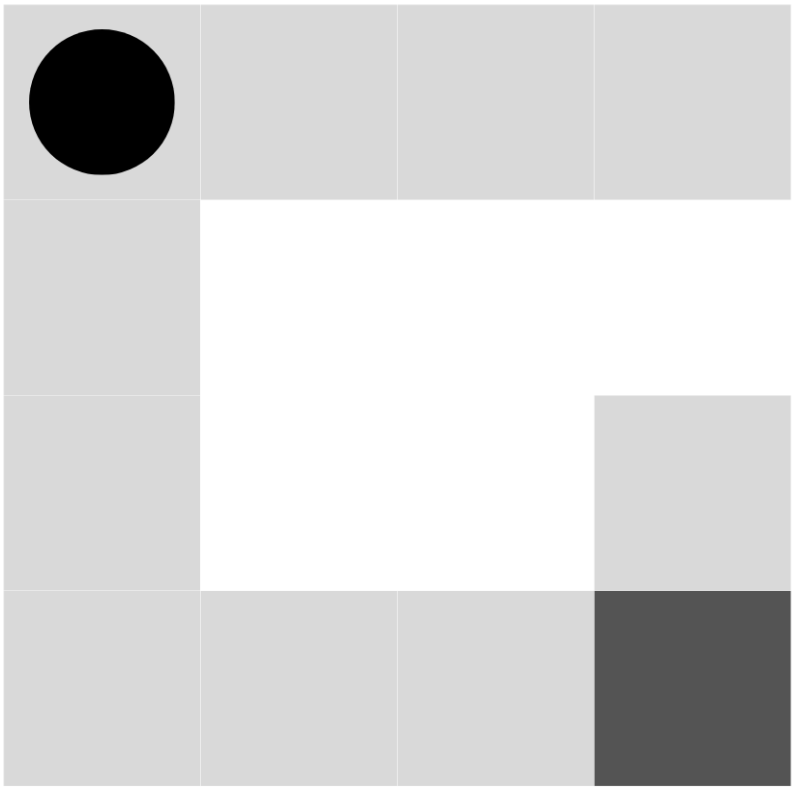


Figura 3.1: Juego de *Grid* de 2 dimensiones

El objetivo del juego es guiar la ficha negra hasta la casilla gris oscuro. Al llegar a la casilla gris oscuro, se obtiene una recompensa de 1. La ficha puede moverse arriba, abajo, a la derecha o a la izquierda. Si estos movimientos conducen a una posición fuera del tablero, el episodio finaliza con una recompensa de -0,5. Por tanto, la ficha debe alcanzar la casilla gris

oscuro a través de un camino específico. Como la ficha no conoce este camino, el agente tendrá que descubrirlo a partir de las recompensas obtenidas. Una vez que un episodio termina, la ficha negra retorna a la posición inicial.

Inicialmente, se realiza un recorrido por el juego. El recorrido seguido se muestra en la Figura 3.2.

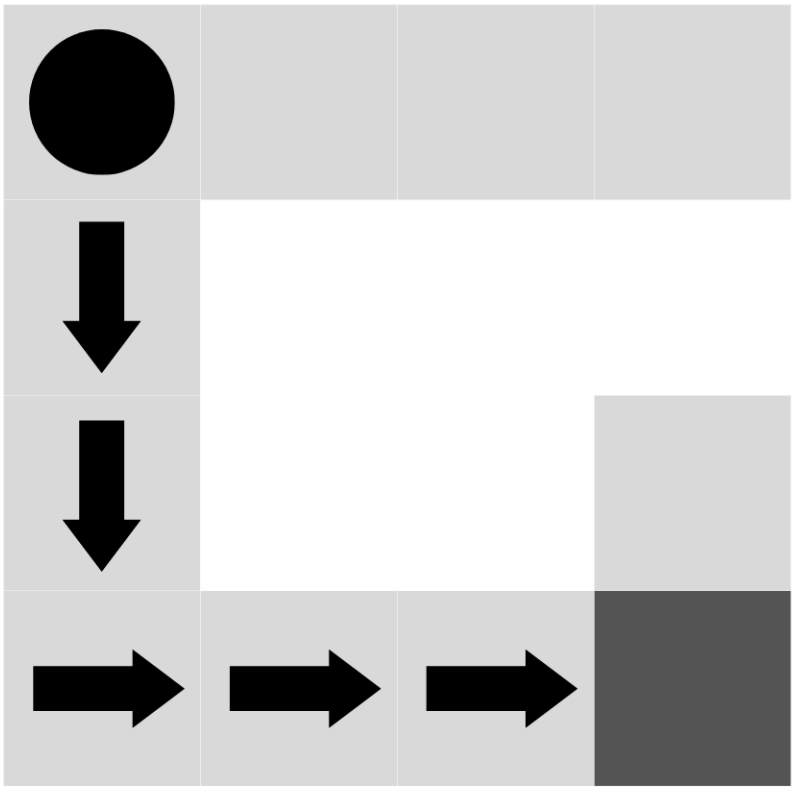


Figura 3.2: Primer recorrido en el *Grid* de 2 dimensiones

Con este recorrido, se logra finalizar el juego de manera exitosa. Por tanto, todos los estados recibirán una recompensa positiva. En este caso, se usará un factor de descuento γ de 0,99, lo que significa que la recompensa esperada del estado siguiente se reducirá en un 0,99.

Con este factor de descuento, se puede calcular de manera sencilla la recompensa esperada para todos los estados por los que ha pasado el agente. La Figura 3.3 muestra el resultado de estos cálculos.

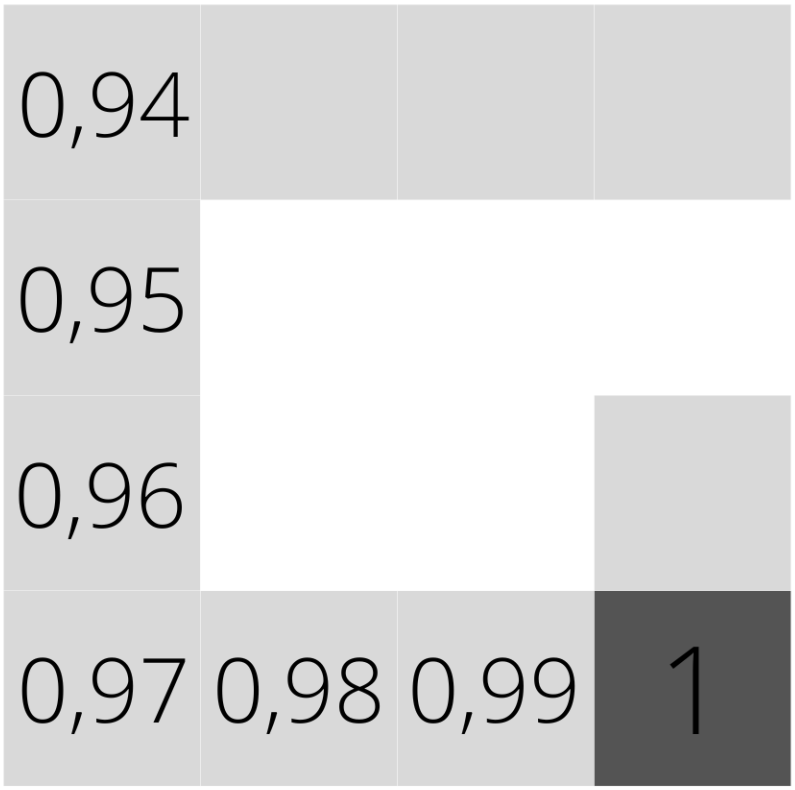


Figura 3.3: Recompensas esperadas después del primer recorrido en el *Grid* de 2 dimensiones

Se empieza con el estado terminal, que tiene una recompensa de 1. Se procede al penúltimo estado, multiplicando la recompensa del estado final por 0,99 y añadiendo la recompensa de este nuevo estado (en este caso 0). De esta forma, se obtiene la recompensa esperada de este estado. Este proceso se repite hasta llegar al estado inicial, que tiene una recompensa de 0,94

(todos los valores están redondeados a dos decimales).

Se realiza otro recorrido por el *Grid*, pero en esta ocasión no será tan exitoso y la ficha se saldrá del camino. El camino seguido se muestra en la Figura 3.4.

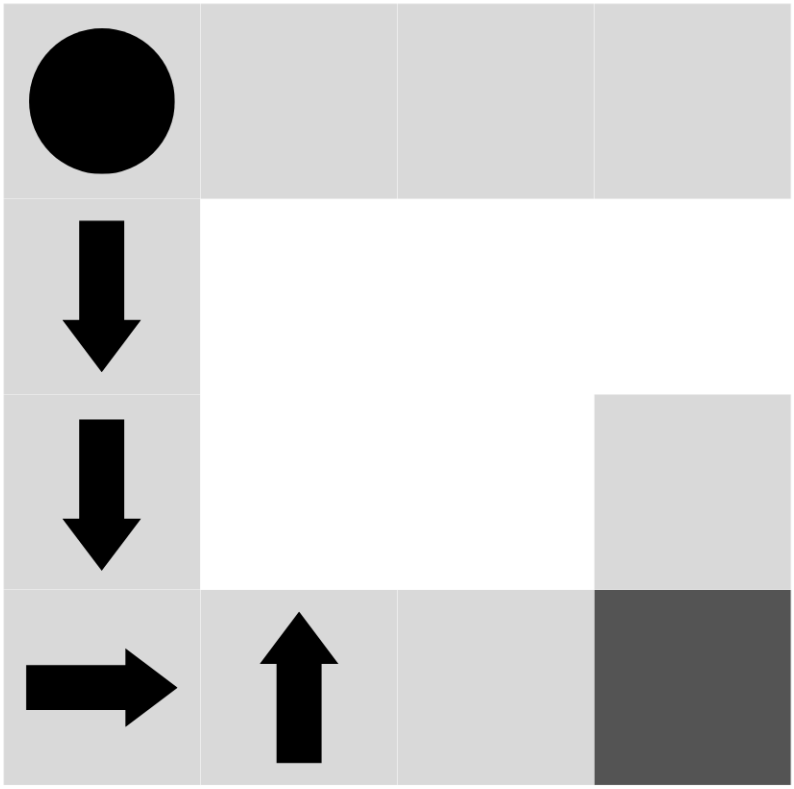


Figura 3.4: Segundo recorrido en el *Grid* de 2 dimensiones

Ahora es el momento de actualizar las casillas por las que ha pasado la ficha. Su nuevo valor se calculará como la media de

las recompensas esperadas del primer y segundo recorrido para las casillas que se incluyen en ambos caminos. La Figura 3.5 muestra el resultado final de estos cálculos.

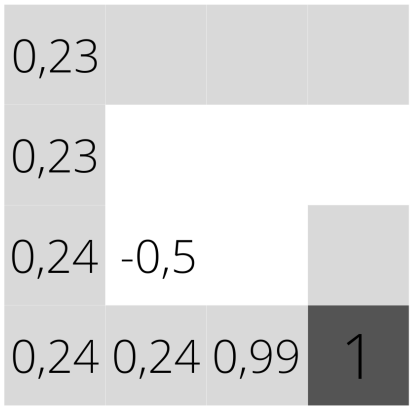


Figura 3.5: Recompensas esperadas después del segundo recorrido en el *Grid* de 2 dimensiones

Se comienza de nuevo por el estado terminal (la casilla con valor -0,5 que es donde la ficha se ha desviado del camino) y se procesa cada estado, calculando su nuevo valor tal y como se ha explicado anteriormente. Este proceso puede repetirse tantas veces como sea necesario. Aunque cuanto más se repite, más precisos serán los valores de cada estado, hay que tener en cuenta que esto conlleva un costo computacional.

El juego presentado anteriormente es totalmente determinista (se sabe a qué estado se pasará con cada acción), pero en algunos casos, una acción determinada podría conducir a diferentes estados según una distribución de probabilidad. El problema con este tipo de juegos es que el proceso que se ha seguido anteriormente ya no sería válido. Sin embargo, se puede solucionar de

manera sencilla asignando los valores a la acción tomada en un estado específico, es decir, el valor se conoce a partir del estado y la acción tomada en ese estado.

Otro aspecto a considerar es determinar qué recorrido realizar. En el caso anterior, el recorrido ha sido aleatorio. Esta sería una política aleatoria. La política define qué acción tomar dada un estado en particular. Este tema se tratará en la próxima sección.

3.2. Política

La política, en términos de aprendizaje automático, representa las decisiones tomadas por un agente en función de un estado dado. Estas políticas pueden clasificarse de dos maneras principales: deterministas y no deterministas. La principal distinción entre ambas radica en su predictibilidad: una política determinista siempre resultará en la misma acción dada la misma situación o estado, mientras que una política no determinista puede conducir a diferentes acciones incluso para un mismo estado.

Además, las políticas también pueden clasificarse en función del número de acciones posibles. Cuando este número es finito, nos encontramos ante un espacio de acciones discreto. En este caso, se asigna un identificador único a cada acción, usualmente a través de un proceso de enumeración. Sin embargo, también puede ocurrir que el número de acciones posibles sea infinito, y en este caso, estaríamos hablando de un espacio de acciones continuo.

A lo largo de este texto, vamos a asumir que la política opera sobre un espacio de acciones discreto. El análisis y estudio de espacios de acciones continuos es un tema de gran profundidad

y complejidad, y queda fuera del alcance de este libro. Además, no tiene ninguna relación con el ajedrez.

Las políticas deterministas se pueden representar mediante la siguiente función:

$$\begin{aligned} f(\alpha) &= \beta \\ \alpha &\in E \\ \beta &\in A \end{aligned}$$

donde:

- α representa el estado actual,
- β es la acción a tomar,
- E es el conjunto de todos los estados posibles,
- A es el conjunto de todas las acciones posibles ($A = \{0, 1, ..., n-1\}$) y
- n es el número total de acciones.

Dado que las acciones están enumeradas desde 0 hasta $n-1$, cualquier número en ese rango podría ser retornado por la función. Si tenemos asociados los valores a las acciones para un estado en particular, podemos crear una política determinista sencilla: se seleccionará la acción con el valor más alto. A esta política se la llama "avariciosa" porque siempre elige la acción con el mayor valor. Esta política es óptima si los valores asociados a cada acción son los reales y no una aproximación. Sin embargo, como hemos visto, tener estos valores reales en la mayoría de los casos es inalcanzable.

La estructuración de las políticas no deterministas es un poco más compleja. En lugar de devolver un solo número, estas políticas devuelven un vector con n elementos, donde n es el número de acciones. Los elementos de este vector son probabilidades, de modo que el valor en la posición i del vector representa la probabilidad de escoger la acción i. Al tratarse de probabilidades, la suma de todos los elementos del vector debe ser 1. La función que representa estas políticas es la siguiente:

$$\begin{aligned} f(\alpha) &= \Delta \\ \alpha &\in E \\ \Delta &= \{\delta_0, ..., \delta_{n-1}\} \end{aligned}$$

donde:

- α es el estado actual,
- Δ es la distribución de probabilidad de las acciones,
- δ_i es la probabilidad de que la acción i sea seleccionada,
- E es el conjunto de todos los estados posibles y
- n es el número total de acciones.

Después de recibir la salida de esta función, se elige una acción de acuerdo con las probabilidades obtenidas. Esto permite al agente realizar una variedad de acciones en lugar de repetir siempre la misma, lo cual puede ser muy deseable cuando se enfrenta a adversarios humanos. Al igual que en el caso de las políticas deterministas, si disponemos de los valores asociados a las acciones, podemos transformarlos en una política no

determinista utilizando la función Softmax (para más detalles, ver la sección de redes neuronales). Esta función, en términos generales, crea una distribución de probabilidad de tal manera que cuanto mayor sea el valor de una acción, mayor será su probabilidad de ser seleccionada.

En el caso del tablero de juego 2D descrito en la sección anterior, hay cuatro acciones posibles (arriba, derecha, abajo e izquierda). Enumeramos estas acciones de la siguiente manera para poder implementar las políticas:

0 : Arriba

1 : Derecha

2 : Abajo

3 : Izquierda

Dadas las recompensas esperadas (valores) mostradas en la Figura 3.6, podemos mostrar cómo seleccionaría las acciones una política determinista y una no determinista.

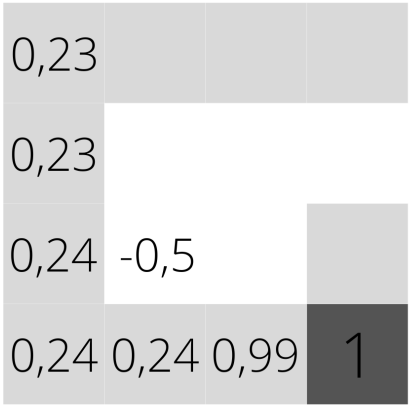


Figura 3.6: Recompensas esperadas para calcular la política en el *Grid* de 2 dimensiones

La pieza de juego está ubicada en la casilla inicial (esquina superior izquierda) y puede realizar las cuatro acciones. Dado que las casillas sin ningún valor tendrían un valor de 0, la política determinista devolvería el valor 2, correspondiente a moverse hacia abajo. Este movimiento llevaría al estado con mayor valor. Siguiendo este procedimiento, la política determinista continuaría seleccionando el valor 2 hasta llegar a la esquina inferior izquierda. A partir de ese punto, sólo devolvería el valor 1, correspondiente al movimiento a la derecha, hasta llegar a la casilla final.

En el caso de la política no determinista, la selección de acciones sería algo más complicada ya que requiere el uso de la función Softmax. Al aplicar esta función a la casilla inicial, se obtendrían los siguientes resultados:

$$\{0{,}2347\ 0,2347\ 0,2959\ 0,2347\}$$

La acción de moverse a la derecha tiene la mayor probabilidad de ser elegida, pero las otras opciones no están muy lejos. Esto es razonable dado que la diferencia de 0.24 a 0 es minúscula. A las políticas no deterministas se les suele añadir un parámetro llamado "temperatura", que permite controlar si se dan más probabilidades a las acciones con mayores valores o si todas las acciones tienen una probabilidad similar.

En la política existe un dilema muy importante conocido como el "dilema de exploración-explotación". Este dilema surge cuando el agente debe decidir entre explorar nuevas acciones (exploración) o intentar mejorar las acciones conocidas con variaciones mínimas (explotación). Si el agente se dedica a explorar todo el tiempo, las estimaciones pueden no ser realistas. Por otro lado, si se centra únicamente en la explotación, puede no encontrar la mejor solución, ya que puede que esté en caminos no explorados.

Para equilibrar este dilema, se suele utilizar un parámetro conocido como ϵ (épsilon). Este parámetro indica la probabilidad de elegir una acción de manera aleatoria. Por ejemplo, si ϵ es 0.05, entonces en el 5 % de las situaciones se elegirá la acción al azar. Al principio, el valor de ϵ será muy cercano a 1, pero disminuirá progresivamente a medida que se realicen más episodios hasta que su valor sea insignificante o directamente 0.

De esta manera, se fomenta la exploración al principio del entrenamiento del agente y, a medida que avanza, se centra más en mejorar las estimaciones para las acciones conocidas. En el caso de las políticas no deterministas, se puede prescindir de ϵ y configurar la política de tal manera que se interese por las acciones no tomadas al principio. Sin embargo, en las políticas deterministas es prácticamente obligatorio usar este parámetro

para lograr un buen rendimiento en el aprendizaje de la política.

Finalmente, en la política existe una división entre si usar la política aprendida para el entrenamiento (*on-policy*) o bien se usa una política diferente que la usada en el entrenamiento (*off-policy*).La principal diferencia es que en las políticas *on-policy*, la política que se va mejorando progresivamente se utiliza en el propio entrenamiento.

Además de la política, también puede ser útil conocer el comportamiento del entorno. Para ello, se busca obtener un "modelo" del entorno. Este modelo permite saber cómo reaccionará el entorno a las diferentes acciones.

3.3. Modelo

El objetivo principal de nuestro modelo es comprender en profundidad las dinámicas subyacentes de un juego. Esto implica conocer las probabilidades de transición a diferentes estados luego de la ejecución de una acción específica. Tal entendimiento es particularmente útil en el contexto de los juegos no deterministas, donde las probabilidades posteriores a una acción no pueden ser previamente conocidas con certeza. No obstante, ¿cómo es esto aplicable en un juego como el ajedrez que es inherentemente determinista?

Sorprendentemente, se puede abordar el ajedrez desde un punto de vista no determinista. Primero, es fundamental recordar que el ajedrez es un juego bipartito, es decir, siempre existe un adversario que nos enfrenta. Este oponente será sometido a diversas posiciones de tablero en las cuales las probabilidades de realizar determinadas jugadas no serán necesariamente iguales. Por ende, nuestro modelo se enfocará en anticipar y predecir las

posibles jugadas del contrincante.

En el contexto del ajedrez, se suele asumir que el rival optará por la mejor jugada posible, siguiendo una lógica similar al algoritmo Minimax. Esto conduce a una coincidencia entre el objetivo de la política y del modelo, ambos persiguen identificar el mejor movimiento posible para cada posición. Esto implica que, de hecho, solo es necesario entrenar uno de ellos, ya que los resultados podrán ser aplicados al otro de manera directa.

Esta es la razón por la cual el uso de un modelo puede ser extremadamente fructífero, sin la necesidad de realizar cálculos complejos, ya que la información necesaria se puede obtener directamente de la política.

Para ilustrar aún más este concepto, consideremos el ejemplo de un *Grid* 2D. Conocer el modelo de este juego en su forma original, es decir, en un entorno determinista, no añadiría valor, ya que ya sabemos a qué estado lleva cada acción. Sin embargo, si modificamos ligeramente la definición del juego, incorporando una probabilidad del 10 % de que cada acción pueda resultar en un estado no previsto, las definiciones de las acciones cambiarían a:

$$
\begin{aligned}
&0 : \text{Arriba } 90\,\%,\ \text{Derecha } 10\,\% \\
&1 : \text{Derecha } 90\,\%,\ \text{Abajo } 10\,\% \\
&2 : \text{Abajo } 90\,\%,\ \text{Izquierda } 10\,\% \\
&3 : \text{Izquierda } 90\,\%,\ \text{Arriba } 10\,\%
\end{aligned}
$$

Estas acciones son similares a las presentadas en la sección anterior, con el matiz adicional previamente mencionado. Además, dichas acciones modelan de forma más precisa situaciones de la vida real, por ejemplo, la tarea de guiar a un robot a través de

un camino, donde las partes móviles del robot pueden introducir imprecisiones o errores.

Nuestro modelo, en este caso, buscará descifrar estas distribuciones de probabilidad asociadas al juego, ya que éstas no son conocidas por el agente que realiza las acciones.

Finalmente, habiendo expuesto los tres pilares del aprendizaje por refuerzo (recompensa esperada, política y modelo), podemos clasificar los diferentes algoritmos de aprendizaje reforzado en función de qué objetivo(s) se enfocan en lograr.

3.4. Algoritmos de aprendizaje reforzado

El campo del aprendizaje reforzado ha experimentado una notable efervescencia en los últimos años, que ha conducido a la creación de una diversidad considerable de algoritmos. Es importante mencionar que discutir todos estos algoritmos excedería el propósito de este libro. Por lo tanto, la discusión se limitará a los algoritmos más significativos y a aquellos que poseen una relevancia especial en el contexto del ajedrez.

Los algoritmos de aprendizaje reforzado se pueden clasificar en tres categorías principales: basados en la política (que se enfocan en el cálculo de la política), basados en los valores (cuyo objetivo es calcular la recompensa esperada) y basados en el modelo (que buscan calcular el modelo). Cabe señalar que algunos algoritmos pueden clasificarse en varias categorías, ya que su diseño implica el cálculo de varios de estos componentes.

Pasaremos a detallar los algoritmos más relevantes.

El algoritmo REINFORCE busca generar una distribución de probabilidades para un estado específico. Es decir, intenta formular una política no determinista, donde las acciones con los

mejores resultados tienen una mayor probabilidad de ocurrencia. Este algoritmo requiere una red neuronal para generar esta distribución de probabilidad basada en el estado actual. Por lo tanto, se clasifica como un algoritmo basado en la política.[9]

El algoritmo DQN (Deep Q-Networks) busca calcular la recompensa esperada en función de un estado dado y la acción a realizar en ese estado. Utiliza una red neuronal profunda para cumplir este propósito. Una característica distintiva de este algoritmo es su naturaleza *off-policy*, lo que significa que su entrenamiento no depende de la política seguida. En teoría, se podría utilizar cualquier política, aunque en la práctica, ciertas políticas son más efectivas que otras. Además, si se dispone de datos previos sobre las acciones realizadas y las recompensas obtenidas, estos pueden ser aprovechados por el algoritmo. Una vez concluido el entrenamiento, las acciones son seleccionadas siguiendo los procedimientos discutidos en la sección de Política. Este algoritmo pertenece a la categoría de los basados en el valor.[10]. Sin embargo, cabe señalar que es adecuado únicamente para los casos en los que el espacio de acciones es discreto.

El algoritmo Actor-Crítico se sitúa en las categorías de basados tanto en el valor como en la política. Consta de dos componentes principales: el actor, que aprende la política, y el crítico, que aprende la función que relaciona las acciones realizadas en un estado dado con su recompensa esperada. De esta forma, el crítico proporciona al actor la información necesaria para optimizar la política. Su variante más utilizada es A2C (Advantage Actor-Critic), donde se aprende una función de ventaja que informa sobre la calidad de una acción en comparación con otras acciones posibles. [11]

El último algoritmo a considerar en esta sección tiene una

relevancia particular para el ajedrez y pertenece a la categoría de los basados en el modelo. Su diseño conceptual es similar al algoritmo Minimax, pero realiza una búsqueda más selectiva. Este algoritmo simula un número determinado de partidas y utiliza el modelo para seleccionar las jugadas más probables en estas simulaciones. Esto permite enfocar el análisis en las jugadas más prometedoras dentro de una posición e ignorar las demás. Gracias a esta característica, este algoritmo es comparable a Minimax, pero es significativamente más rápido. Al igual que Minimax, los nodos ubicados en los primeros niveles del árbol almacenan las estadísticas de las partidas simuladas, y aquellas con mejores estadísticas son utilizadas para seleccionar el movimiento más adecuado [12]. En la sección AlphaZero, se proporcionarán detalles adicionales sobre cómo este algoritmo se utiliza en el ajedrez.

3.5. Redes neuronales

En los últimos años, las redes neuronales y el aprendizaje reforzado han sido los principales impulsores del increíble avance de la inteligencia artificial. En teoría, las redes neuronales son capaces de aprender cualquier función dada una entrada y la salida esperada. Aunque en la práctica no es tan sencillo, siguen logrando resultados sorprendentes.

En general, las redes neuronales se dividen en capas. Cada capa aplica una función a la salida de la capa anterior. Al finalizar este proceso, se compara la salida de la última capa con la salida esperada. Esta comparación permite que la red neuronal determine qué aspectos de sus parámetros internos deben modificarse para mejorar en el futuro.

Existen capas simples, como los perceptrones, y capas más complejas, como las capas convolucionales o recursivas, entre otras. Además, al final de cada capa se aplica una función de activación, que agrega no linealidad a la salida. Esta función de activación suele ser una función no lineal sin parámetros ajustables. Se utiliza para modelar características no lineales del modelo, es decir, aquellas en las que un cambio en una característica no implica un cambio proporcional en la salida.

3.5.1. Perceptrón

El elemento fundamental de las redes neuronales es el perceptrón. Se trata de un componente muy simple que recibe varias entradas y produce una salida. Sin embargo, cuando múltiples perceptrones actúan en conjunto, logran un rendimiento sorprendente. La Figura 3.7 muestra un perceptrón simple con tres entradas.

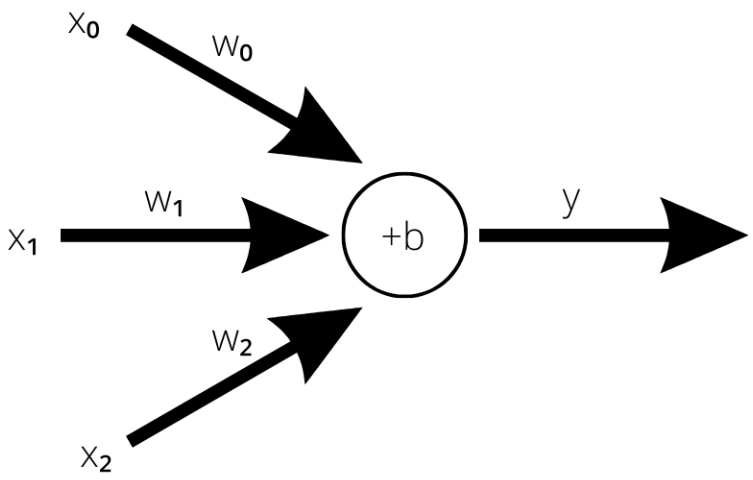


Figura 3.7: Perceptrón con 3 entradas

A partir de la imagen anterior, podemos establecer una definición matemática utilizando la siguiente función:

$$f(x) = b + \sum_{i=0}^{2} x_i w_i$$

Generalizando esta definición para n entradas, obtendríamos lo siguiente:

$$f(x) = b + \sum_{i=0}^{n-1} x_i w_i$$

Aquí, x_i representa las entradas que se multiplican por sus respectivos pesos w_i, y luego se les suma el valor b. Por lo tanto, los parámetros a aprender son b y todos los pesos w_i. Normalmente, se utiliza una notación vectorial para expresar el perceptrón, donde tanto las entradas como los parámetros se representan como vectores. La notación utilizada para el perceptrón de la Figura 3.7 sería la siguiente:

$$\begin{bmatrix} 1 & x_0 & x_1 & x_2 \end{bmatrix} \qquad \begin{bmatrix} b \\ w_0 \\ w_1 \\ w_2 \end{bmatrix}$$

(a) Representación de la entrada del perceptrón de 3 entradas

(b) Representación de los parámetros del perceptrón de 3 entradas

Figura 3.8: Representación vectorial del perceptrón de 3 entradas

Esta representación presenta una peculiaridad. La primera entrada, con un valor de 1, es fija y no se considera una entrada propiamente dicha. Se representa de esta manera para poder sumar directamente el producto de los dos vectores finales al valor de b. La generalización de esta definición para n entradas es sencilla a partir de este ejemplo. Simplemente se deben agregar las entradas y parámetros a sus respectivos vectores, sin olvidar incluir 1 y b al inicio de los vectores.

Además, esta representación resulta muy útil cuando se tienen varios perceptrones en la misma capa, ya que al representar los parámetros de cada uno en un vector, se pueden combinar todos los vectores en una matriz. Por ejemplo, consideremos una capa con tres perceptrones y tres entradas. Podríamos representarlos de la siguiente manera utilizando matrices:

$$\begin{bmatrix} 1 & x_0 & x_1 & x_2 \end{bmatrix}$$

(a) Representación de la entrada de la capa de perceptrones de 3 entradas

$$\begin{bmatrix} b_0 & b_1 & b_2 \\ w_{00} & w_{01} & w_{02} \\ w_{10} & w_{11} & w_{12} \\ w_{20} & w_{21} & w_{22} \end{bmatrix}$$

(b) Representación de los parámetros de la capa de perceptrones de 3 entradas

Figura 3.9: Representación matricial de la capa de perceptrones de 3 entradas

En la Figura 3.9, el peso w_{ij} se refiere al peso i asociado al perceptrón j. Es importante destacar que la entrada se mantiene constante sin importar el número de perceptrones en la capa. Para obtener las salidas, solo es necesario realizar una multiplicación matricial, que consiste en multiplicar uno a uno cada elemento de los vectores de parámetros con el vector de entrada. La Figura 3.10 ilustra cómo sería la salida utilizando la misma representación.

$$\begin{bmatrix} 1 & x_0 & x_1 & x_2 \end{bmatrix} \times \begin{bmatrix} b_0 & b_1 & b_2 \\ w_{00} & w_{01} & w_{02} \\ w_{10} & w_{11} & w_{12} \\ w_{20} & w_{21} & w_{22} \end{bmatrix} = \begin{bmatrix} y_0 & y_1 & y_2 \end{bmatrix}$$

Figura 3.10: Representación matricial de la salida de la capa de perceptrones de 3 entradas

La multiplicación de matrices se realiza en la práctica aplicando las fórmulas mostradas en la Figura 3.11. Estas fórmulas representan el proceso directo para calcular la salida de una capa de perceptrones de 3 entradas.

$$y_0 = b_0 + \sum_{i=0}^{2} x_{i0} w_{i0}$$

$$y_1 = b_1 + \sum_{i=0}^{2} x_{i1} w_{i1}$$

$$y_2 = b_2 + \sum_{i=0}^{2} x_{i2} w_{i2}$$

Figura 3.11: Fórmula directa para calcular la salida de capa de perceptrones de 3 entradas

El proceso de aprendizaje de los perceptrones es bastante simple. Comparan la salida obtenida por el perceptrón con la salida esperada y utilizan la diferencia entre ambas para ajustar

los pesos. Sin embargo, el principal problema de los perceptrones radica en su naturaleza totalmente lineal, lo cual se hace evidente al representarlos mediante vectores y matrices. Para superar esta limitación, se aplica una función de activación a la salida del perceptrón, lo que introduce no linealidad en el proceso.

Al tratar con imágenes, surgen nuevos desafíos. Las imágenes se representan mediante matrices, donde cada posición corresponde a un píxel y cada matriz indica la intensidad de un color u otros atributos relacionados con ese píxel. Sin embargo, los perceptrones no pueden procesar directamente estas matrices, ya que se requiere convertirlas en vectores. Sin embargo, esta conversión implica la pérdida de información sobre la disposición espacial de los píxeles y su relación entre sí. Elementos que estaban contiguos en la matriz pueden estar muy separados en el vector resultante. Esta información espacial es crucial para el procesamiento de imágenes (y el ajedrez), por lo que se necesita un método alternativo.

Afortunadamente, existe un procedimiento altamente efectivo conocido como convolución, el cual aborda esta problemática. La convolución permite preservar la estructura espacial de las imágenes al procesarlas, lo que resulta fundamental para tareas como el reconocimiento de objetos y la extracción de características.

3.5.2. Convolución

La convolución se asemeja al funcionamiento de un perceptrón, pero opera en regiones cuadradas (en ocasiones excepcionales se pueden usar secciones rectangulares) de una imagen. Para comprender mejor el funcionamiento de una capa de convolución, es útil utilizar un ejemplo. En la Figura 3.12, se muestra la configu-

ración inicial de la entrada y los parámetros de la convolución.

$$\begin{bmatrix} 0,24 & 0,12 & 0,09 \\ 0,13 & 0,13 & 0,21 \\ 0,05 & 0,12 & 0,24 \end{bmatrix}$$

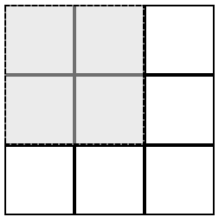
(a) Entrada de la convolución de ejemplo

$$\begin{bmatrix} 0,56 & -0,54 \\ 0,07 & 0,32 \end{bmatrix}$$

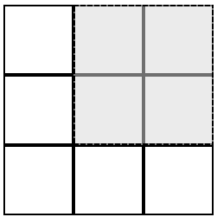
(b) Parámetros de la convolución de ejemplo

Figura 3.12: Ejemplo de convolución

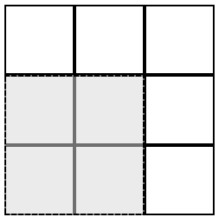
En este ejemplo, utilizaremos un *stride* (desplazamiento) con un valor de 1 y un valor de $b = 1$ (que se suma de manera similar a los perceptrones). Con la configuración actual, se realizarán un total de 4 convoluciones. En la Figura 3.13, se muestra cómo se llevarían a cabo las convoluciones.



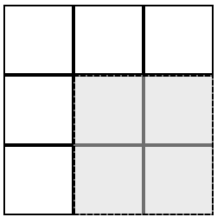
(a) Primera convolución



(b) Segunda convolución



(c) Tercera convolución



(d) Cuarta convolución

Figura 3.13: Convoluciones realizadas en el ejemplo

Dado que se realizarán cuatro convoluciones, la salida resultante será una matriz de tamaño 2×2. Para obtener el resultado de cada convolución, se multiplica la entrada (la región sombreada en cada convolución) por los parámetros correspondientes de la convolución. Luego se suman todos estos resultados y se le agrega b, lo cual será el valor de salida para esa convolución es-

pecífica. La Figura 3.14 muestra la salida obtenida al aplicar la convolución.

$$\begin{bmatrix} 1,12 & 1,1 \\ 1,05 & 1,05 \end{bmatrix}$$

Figura 3.14: Salida de la convolución de ejemplo

En el análisis detallado de la operación de convolución, observamos que el componente situado en el vértice superior izquierdo del resultado se determina a través de un cálculo específico: se realiza la suma de las multiplicaciones de cada elemento del kernel de convolución con el píxel correspondiente de la imagen de entrada. Para ejemplificar, consideremos los siguientes valores: 0,24, 0,12, 0,13, 0,13, los cuales son multiplicados respectivamente por 0,56, $-0{,}54$, 0,07, 0,32 para luego ser sumados entre sí y finalmente se añade una constante 1, el término de sesgo denotado por $b = 1$. De este modo, el valor calculado para el componente en la esquina superior izquierda es 1,12.

Un parámetro importante en el proceso de convolución es el denominado *padding*. Este procedimiento implica el incremento del tamaño de las imágenes a través de la adición de ceros en los bordes de la misma. La utilidad de esta técnica radica en su capacidad de permitir realizar convoluciones de manera más eficaz y precisa en las regiones periféricas de las imágenes, en especial cerca de las esquinas.

En conjunción con las operaciones de convolución, se utiliza comúnmente una capa de *pooling* en las redes neuronales convolucionales. Esta capa desempeña un papel crucial en la disminución del tamaño de la salida conservando, al mismo tiempo,

la información más significativa. A pesar de su semejanza con las convoluciones, al seleccionar bloques de la entrada, la capa de *pooling* se distingue por no poseer parámetros propios para aprender. Existen principalmente dos tipos de operaciones de *pooling*: el *max pooling*, que selecciona el valor máximo del bloque de entrada, y el *average pooling*, que calcula la media de todos los valores en dicho bloque. Ambos métodos de *pooling* consisten en convertir, en la práctica, un conjunto de píxeles (como los agrupados en cuadrados de 4, 9, 16, entre otros) en un solo píxel, logrando así una reducción en la complejidad.

3.5.3. Funciones de activación

Las funciones de activación desempeñan un papel fundamental al ampliar las capacidades de las redes neuronales en diversos ámbitos, al proporcionarles características no lineales, las cuales son necesarias en la mayoría de los problemas. Estas funciones generalmente no requieren parámetros y tres de las más relevantes son ReLU, Sigmoid y Softmax.

ReLU (Rectified Linear Unit) es ampliamente utilizada en las capas intermedias de las redes neuronales, es decir, en todas las capas excepto la última. Su funcionamiento es bastante simple. Si la entrada es positiva, se mantiene sin cambios, pero si es negativa, se convierte en cero. Matemáticamente, se define de la siguiente manera:

$$f(x) = \begin{cases} x, \text{ si } x \geq 0 \\ 0, \text{ si } x < 0 \end{cases}$$

Es comúnmente reconocido que el uso de ReLU, posterior a las operaciones de convolución, proporciona buenos resulta-

dos. De manera predeterminada, esta capa se implementa en las etapas intermedias de la red neuronal (todas las capas excepto la última). Para la última capa, es preferible utilizar distintos tipos de funciones de activación que limiten el rango de los valores posibles a un subconjunto más reducido. Algunas de las más destacadas son las funciones Sigmoid, Softmax y la tangente hiperbólica. Esta última se analizará en detalle en el capítulo dedicado a AlphaZero. Estas funciones restringen los valores a estar entre 0 y 1 para Sigmoid y Softmax, y entre -1 y 1 para la tangente hiperbólica.

Las funciones Sigmoid y Softmax comparten un propósito común: transformar la salida en una distribución de probabilidad (Sigmoid también se puede utilizar para regresiones en el intervalo $(0, 1)$). La diferencia radica en que la función Sigmoid se aplica sobre un solo perceptrón, mientras que la función Softmax puede aplicarse sobre dos o más perceptrones. Por ende, Sigmoid es útil para decisiones binarias (por ejemplo, determinar si un objeto pertenece o no a una categoría), mientras que Softmax puede generalizarse a tantas categorías como se desee. Por lo tanto, la elección entre ambas dependerá del número de categorías que se necesite clasificar.

La función Sigmoid se define matemáticamente de la siguiente forma:

$$f(x) = \frac{1}{1 + e^{-x}}$$

Indistintamente de qué tan grande o pequeño sea el valor de x, este nunca excederá los límites de 1 y 0 respectivamente. Si x es igual a 0, el resultado será $1/2$, por lo que todos los valores positivos estarán en el intervalo $(\frac{1}{2}, 1)$ y los negativos en $(0, \frac{1}{2})$.

Por su parte, la función Softmax permite transformar un conjunto de valores en una distribución de probabilidad, de manera tal que los valores más altos posean mayor probabilidad de ocurrir que aquellos con valores más bajos. Esta transformación se realiza mediante el uso de la función exponencial. La definición matemática de Softmax es:

$$f(x_i) = \frac{e^{x_i}}{\sum_{i=1}^{n} e^{x_i}}$$

El denominador en esta fracción corresponde a la suma de todos los valores de entrada x_i, cada uno elevado a la potencia de e, mientras que el numerador corresponde al valor x_i individual, también elevado a e.

Para ilustrar el funcionamiento de la función Softmax, consideremos el siguiente vector:

$$\begin{bmatrix} 1 & 4 & 3 & 1 \end{bmatrix}$$

Primero, se calcula el denominador común a los cuatro elementos, que resulta ser 80.12. Luego, se procesa cada elemento del vector individualmente, obteniendo los siguientes resultados:

$$\begin{bmatrix} 0,034 & 0,681 & 0,251 & 0,034 \end{bmatrix}$$

Como se puede observar, el elemento con valor 4 en el vector original es el que predomina con una probabilidad de 0,681 de ocurrir. También es importante notar que la suma de todos los elementos es 1, corroborando que la función Softmax ha convertido el vector original en una distribución de probabilidad.

Softmax es frecuentemente utilizada en problemas de clasificación multiclase (más de dos clases) y en situaciones donde se necesita seleccionar una acción de manera no determinista.

3.5.4. *Backpropagation*

Backpropagation es el algoritmo que facilita el cálculo de las actualizaciones requeridas para todos los parámetros de una red neuronal. Tal como se mencionó anteriormente, cuando se cuenta con una sola capa, el cálculo de las actualizaciones de los pesos es bastante sencillo. Sin embargo, cuando el número de capas incrementa, este proceso se vuelve más complejo. Afortunadamente, el algoritmo de *backpropagation* simplifica enormemente esta tarea mediante el uso de derivadas.

El funcionamiento de este algoritmo se basa en primer lugar en el cálculo secuencial de las capas de la red neuronal, almacenando información relevante como la salida de cada una de estas capas. Al llegar al final, se compara la salida obtenida de la red neuronal con la salida esperada, utilizando para ello una función de pérdida. Esta función generará un número que indicará cuán distantes están ambas salidas; a mayor número, peor será el rendimiento de la red neuronal.

El siguiente paso es la "propagación hacia atrás", que es la esencia del algoritmo. Partiendo de la pérdida, se recorre cada capa comenzando por la última, modificando los parámetros de la red neuronal. Dentro de este proceso existe un subalgoritmo llamado optimizador, cuyo objetivo es minimizar la función de pérdida.

La función de pérdida tiene dos parámetros: la salida esperada y y la salida generada por la red neuronal $\hat{y}$. La definición matemática de la función de pérdida sería:

$$f(\hat{y}, y) = \alpha$$

Donde α representa el valor correspondiente a la pérdida. Sin embargo, las entradas a la función de pérdida pueden ser heterogéneas, y pueden incluso ser de diferente tipo. Un escenario común es cuando la salida de la red neuronal es una distribución de probabilidad, mientras que la salida esperada es simplemente un número que indica la acción; en tal caso, se elegirá la probabilidad de ejecutar esa acción según la distribución de probabilidad.

La elección de la función de pérdida está estrechamente ligada al tipo de problema que se busca resolver. Así, si el problema es de clasificación, la función de pérdida será muy distinta a la que se usaría en un problema de regresión.

El optimizador busca minimizar la función de pérdida, actuando como una especie de "guía" para la red neuronal. El método habitual para conseguir esto es el descenso de gradiente, que tiene como objetivo encontrar los valores mínimos de una función, en este caso, la función de pérdida. El optimizador utiliza el descenso de gradiente junto con la modificación de otros parámetros, principalmente el coeficiente de aprendizaje (que indica cuánto se modifican los parámetros de la red neuronal) para optimizar el rendimiento de la red.

Entre los optimizadores disponibles para las redes neuronales, uno de los más utilizados es el SGD (Stochastic Gradient Descent). Este algoritmo implementa una versión del descenso de gradiente que introduce variaciones aleatorias con el propósito de evitar quedar atrapado en óptimos locales. Los óptimos locales son puntos en los que la función alcanza un valor mínimo en una región específica del espacio de búsqueda, pero que no

necesariamente representan el valor mínimo global de la función. Estos óptimos locales pueden limitar la capacidad del optimizador para encontrar la solución más óptima, de ahí la importancia de tener estrategias para evitarlos.

El SGD resuelve este problema al introducir un elemento de aleatoriedad en el proceso de descenso del gradiente. En lugar de utilizar todo el conjunto de datos para calcular el gradiente en cada paso (como se hace en el descenso de gradiente estándar), el SGD selecciona un subconjunto aleatorio (o incluso un solo ejemplo) para hacer el cálculo. Esto introduce una variabilidad que puede ayudar a salir de los óptimos locales.

Otro optimizador destacable es Adam (Adaptive Moment Estimation), que es una extensión del SGD que introduce varias mejoras. La primera de estas mejoras es el uso de tasas de aprendizaje adaptativas, que significa que Adam ajusta la tasa de aprendizaje (el tamaño de los pasos que se dan en el descenso del gradiente) para cada parámetro individualmente en función de estimaciones del primer y segundo momento (es decir, la media y la varianza) de los gradientes.

Además, Adam también incluye un mecanismo conocido como "momento", que hace que el optimizador no solo tenga en cuenta el gradiente actual, sino también los gradientes de las iteraciones anteriores. Esto puede ayudar a acelerar la convergencia del algoritmo y también a superar los óptimos locales y las zonas planas de la función de pérdida.

Finalmente, Adam también introduce una corrección de sesgo para sus estimaciones del primer y segundo momento, lo que ayuda a obtener estimaciones más precisas al principio del entrenamiento.

Estas características hacen que Adam sea un optimizador

muy eficaz y ampliamente utilizado en la formación de redes neuronales. Sin embargo, como cualquier método, no es una solución universal y puede no ser el mejor optimizador para todas las situaciones o para todos los tipos de redes neuronales. Por lo tanto, es recomendable experimentar con diferentes optimizadores y configuraciones para encontrar la opción más adecuada para cada problema específico.

Una vez analizados todos los componentes que constituyen el aprendizaje reforzado profundo, es posible adentrarse en el estudio de AlphaZero y su aplicación en el ámbito del ajedrez. AlphaZero representa la convergencia de diversos bloques previamente examinados, permitiendo comprender cómo se aplican estos conceptos en este juego milenario.

3.6. AlphaZero

En esta sección profundizaremos en el análisis de un módulo revolucionario que ha transformado el panorama del ajedrez tal como lo conocemos. Este módulo, denominado AlphaZero, es famoso por su impacto en el ámbito del ajedrez computacional. Aunque el código fuente de AlphaZero no está a disposición del público, la metodología y su funcionamiento han sido minuciosamente descritos en el artículo de investigación publicado por Deep Mind [13].

Por otro lado, existe una versión de código abierto, u *open source* [1], denominada Leela Chess Zero (LCZero), cuya arquitectura y mecanismo de funcionamiento se asemejan notablemente a los de AlphaZero.

Tanto LCZero como AlphaZero operan sobre el principio de

[1] Este término indica que el código fuente está disponible para consulta y uso gratuito

utilizar una inteligencia artificial (IA) sin conocimientos previos del juego de ajedrez, de ahí el término "Zero" en sus respectivos nombres. Esta IA se va educando y mejorando a medida que juega contra sí misma, lo cual permite prescindir de cualquier prejuicio o sesgo humano inherente. Esta metodología representa un marcado contraste con los enfoques tradicionales, en los cuales la búsqueda de jugadas se basa en conocimientos aportados por expertos en ajedrez.

En términos generales, ambos sistemas se componen de dos elementos fundamentales: una red neuronal y un árbol de búsqueda Monte Carlo. La red neuronal toma como entrada una posición específica del ajedrez y devuelve, como salida, la distribución de probabilidad de las posibles jugadas y una valoración de dicha posición, la cual se situará entre -1 y 1. El árbol de búsqueda Monte Carlo, utilizando la información proporcionada por la red neuronal, selecciona el mejor movimiento a realizar en la posición dada.

El principal desafío que enfrenta LCZero en comparación con AlphaZero radica en su limitada capacidad computacional. Para mitigar este problema, los desarrolladores de LCZero han implementado una solución distribuida, permitiendo así que cualquier usuario contribuya al entrenamiento de LCZero, aportando su "granito de arena" [2].

3.6.1. Arquitectura de la Red Neuronal

La red neuronal en cuestión cuenta con un gran número de capas, particularmente en la sección de extracción de características. La entrada está formada por una serie de canales (matrices de 8x8), los cuales adoptan únicamente valores 0 o 1. Estos ca-

[2] https://lczero.org/contribute/

nales contienen la información de la posición en una forma que resulta fácilmente interpretable para la red neuronal. Posteriormente, se realiza la extracción de características; durante este proceso, la red neuronal identifica y extrae patrones a partir de los canales de entrada. Estos patrones son luego procesados por las capas finales de la red para determinar el valor del estado actual y la política de selección de acciones.

La Figura 3.15 muestra la red neuronal usada por AlphaZero con las partes antes mencionadas.

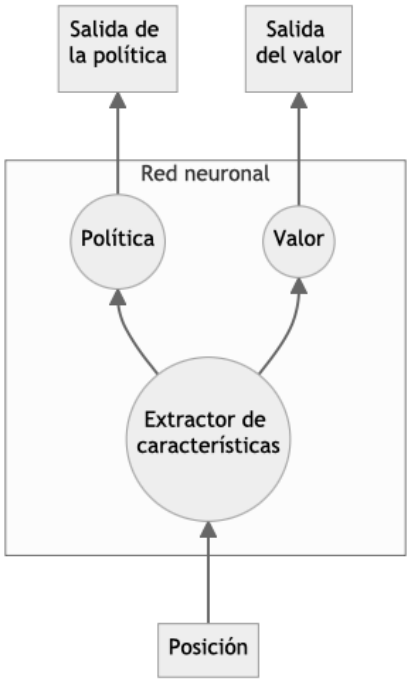


Figura 3.15: Red neuronal de AlphaZero

3.6.1.1. Entrada

Según se especifica en el artículo de investigación, la entrada comprende un total de 119 canales. Estos canales se pueden

clasificar en dos grupos principales.

El primer grupo se compone de la información relativa a la disposición de las piezas en el tablero de ajedrez y dos canales adicionales para indicar si una posición específica se ha repetido una o dos veces.

Para representar la disposición de las piezas se utiliza el formato conocido como *one-hot encoding*. En este formato, todos los elementos de la matriz 8×8 se establecen inicialmente en 0 y se cambian a 1 si una pieza de un tipo particular se encuentra en esa casilla. De este modo, cada tipo de pieza está asociado con un canal específico.

A modo de ilustración, consideremos el tablero mostrado en la Figura 3.16. A continuación, se muestran los canales correspondientes a varias de las piezas presentes en este tablero.

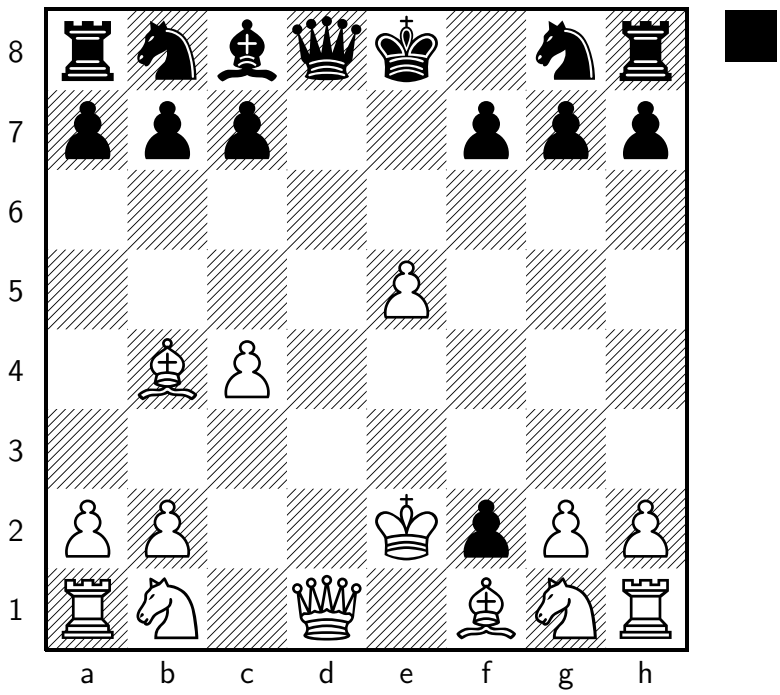


Figura 3.16: Posición de ejemplo para mostrar el formato *one-hot encoding*

En primer lugar, se inicia con los peones blancos. La Figura 3.17 muestra la matriz resultante. En esta matriz, todas las posiciones ocupadas por peones blancos se representan con el valor 1, mientras que las posiciones vacías se representan con el valor 0. Cada tipo de pieza (peón, caballo, alfil, torre, dama y rey de ambos colores) se asigna a un canal específico, lo que da un total de 12 canales distintos.

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Figura 3.17: Representación en formato *one-hot encoding* de los peones blancos

En el caso siguiente, se procede con los caballos negros. La Figura 3.18 ilustra el canal correspondiente a esta pieza.

$$\begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Figura 3.18: Representación en formato *one-hot encoding* de los caballos negros

De forma análoga, se procedería con el resto de tipos de piezas. Continuando con los canales, es pertinente hablar sobre los dos canales utilizados para informar sobre la repetición de la posición. El primer canal tendrá todos sus valores a 1 solo si la posición se ha repetido una vez, mientras que el segundo ca-

nal se activará si la posición actual se ha repetido exactamente dos veces. Esto proporciona a la red neuronal la información de que si repite la posición una vez más (habiendo sido repetida un total de tres veces), el juego se declarará en tablas. En total, este primer grupo de canales cuenta con $12 + 2 = 14$ canales. Este grupo se repite 8 veces, ya que se almacenan las 8 últimas posiciones en lugar de únicamente la última [3]. De esta forma, habrá $14 * 8 = 112$ canales en total.

Además, se debe tener en cuenta la posibilidad de *en passant* (comer al paso). Para representar esta eventualidad, el peón que puede ser capturado *en passant* se traslada de la quinta fila a la última fila en su canal correspondiente. Dado que un peón nunca puede estar en la última fila, esta representación no genera ninguna confusión.

Por otra parte, el segundo grupo de canales almacena información relativa a la posición actual que no está directamente relacionada con la disposición de las piezas. Este grupo consta de 7 canales, sumando un total de 119 canales cuando se combina con el primer grupo. A continuación, se detallan los canales de este segundo grupo:

- Color: Indica el turno del jugador. Si es el turno de las blancas, todos los valores del canal serán 0; de lo contrario, serán 1.

- Número de movimientos: Representa el número total de movimientos realizados en la partida. Todos los valores de la matriz corresponderán a este número. Por ejemplo, si se

[3] En mi opinión, no considero necesario almacenar las 8 últimas posiciones, siendo solo necesaria la última, ya que la información de los dos primeros grupos de canales provee toda la necesaria para la posición actual

han realizado 49 movimientos, todos los valores de la matriz serán 49.

- Enroque corto de las blancas: Indica si las blancas pueden realizar un enroque corto. Si es posible, todos los valores del canal serán 1; en caso contrario, serán 0.
- Enroque largo de las blancas: Similar al anterior, pero para el enroque largo.
- Enroque corto de las negras: Similar al anterior, pero para las negras y el enroque corto.
- Enroque largo de las negras: Similar al anterior, pero para el enroque largo.
- Número de movimientos sin progreso: Similar al canal del número de movimientos, pero cuenta solo aquellos en los que no se ha producido un progreso. Se considera que un movimiento produce progreso si un peón avanza o se captura una pieza. Si se realizan 50 movimientos sin progreso, la partida se declarará en tablas.

Así es, la entrada a la red neuronal constará de 119 matrices de 8×8. Además, es relevante señalar que el tablero siempre se presenta desde la perspectiva del jugador que tiene el turno, es decir, no todos los jugadores verán exactamente el mismo tablero, sino que uno de ellos lo percibirá girado.

Esta entrada se somete a una capa de convolución, inaugurando la fase de extracción de características. Cada capa de convolución está conformada por filtros convolucionales, que se complementan con una normalización de *batch* y una función de activación ReLU. Este esquema es consistente a lo largo de todas las capas convolucionales de esta red neuronal.

La normalización de *batch* es una técnica que normaliza la entrada. Su uso es muy común, pues tiende a mejorar la estabilidad y el rendimiento de las redes neuronales [14]. Para aplicarla, se calculan la media y la desviación estándar de la entrada. Luego, se aplica la siguiente fórmula a cada valor de entrada (x_i):

$$y_i = \frac{x_i - \mu}{\sigma}$$

Donde μ es la media y σ es la desviación estándar.

En una variante más completa de la normalización de *batch*, se incluyen dos parámetros adicionales que permiten escalar y desplazar la salida normalizada (y_i). Así, el resultado final (z_i) se calcula como:

$$z_i = \gamma * y_i + \beta$$

En este caso, los parámetros aprendibles son γ (escala) y β (desplazamiento).

3.6.1.2. Extracción de características

La porción de la red neuronal que se dedica a la extracción de características es bastante voluminosa. Se compone de un total de 40 bloques residuales. Cada uno de estos bloques residuales incluye dos capas convolucionales, pero la segunda capa tiene una particularidad notable. Lo especial de esta segunda capa es que posee una conexión a la entrada original del bloque residual, que se activa después de la normalización de *batch* en esta capa. Este enlace residual permite sumar la salida, tras su normalización, con la entrada original del bloque. Esta operación

de suma se realiza elemento a elemento. Finalmente, a este resultado se le aplica la función de activación ReLU de la segunda capa convolucional.

Gracias a esta conexión residual, se puede realizar la retropropagación (*backpropagation*) de manera más directa, sin necesidad de pasar por todas las capas intermedias.

La Figura 3.19 ilustra la estructura de un bloque residual como el que se ha descrito anteriormente.

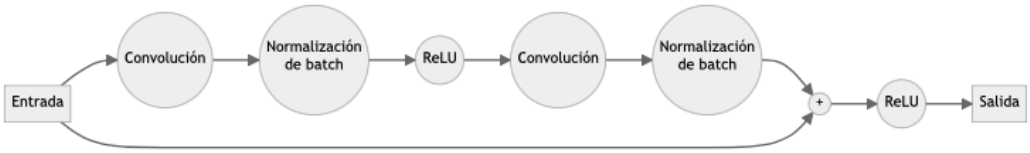


Figura 3.19: Estructura de un bloque residual en AlphaZero

3.6.1.3. Política

El proceso comienza con la extracción de características, las cuales son posteriormente canalizadas a través de una serie de filtros de convolución. A estos datos se les aplica una normaliza-

ción de *batch*, seguida por la implementación de una función de activación ReLU. Al finalizar este proceso, obtenemos un tensor (varias matrices ordenadas) de dimensiones $8 \times 8 \times 73$. Este tensor refleja la distribución de probabilidad de los diferentes movimientos posibles en un tablero de ajedrez, obtenida tras aplicar la función Softmax.

Aunque a primera vista, $8 * 8 * 73 = 4672$ puede parecer una cantidad exorbitante de movimientos posibles, esta cifra se debe a que incluye todas las combinaciones de casillas y movimientos factibles en el juego, e incluso algunos más. Cada componente del tensor 8×8 corresponde a una casilla específica del tablero de ajedrez en la cual se ubica una pieza que está por moverse. De manera más específica, existen realmente 73 movimientos posibles.

Dentro de estos 73 movimientos posibles, los primeros 56 corresponden a los movimientos de la dama, incluyendo todos los movimientos posibles para la dama, rey, alfil y torre. Los ocho movimientos subsecuentes, es decir, aquellos del 57 al 64, engloban los ocho posibles saltos del caballo. Finalmente, los 9 movimientos restantes están reservados para las situaciones en que un peón es promovido a otra pieza que no es una dama.

La lógica detrás de los movimientos de la dama se rige por la orientación de un compás; en consecuencia, la dama puede moverse en cualquiera de las 8 direcciones indicadas por este. En cada dirección, la dama tiene la capacidad de moverse entre 1 y 7 casillas. El producto de estos valores nos proporciona el total de movimientos posibles para la dama. Durante la codificación, se asignará a cada dirección un valor numérico entre 1 y 7, tal y como se muestra en la Figura 3.20. A este valor se le sumará el número de casillas que la dama planea moverse.

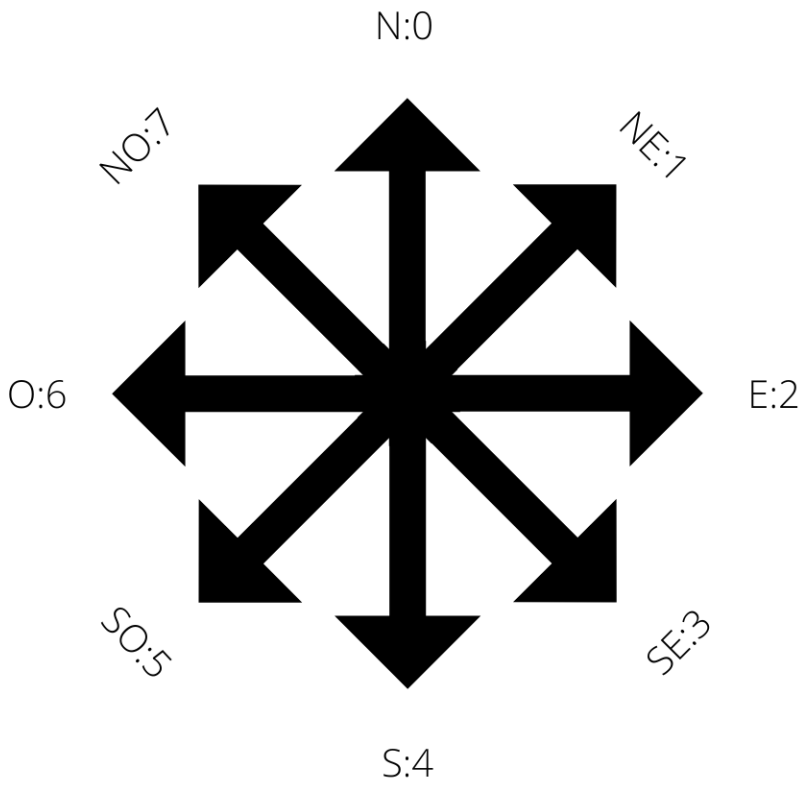


Figura 3.20: Asignación de valores a las direcciones de los movimientos de dama

La fórmula por tanto quedará así:

$$f(\alpha_d, c) = 7 * \alpha_d + c$$

α_d es el coeficiente de las direcciones (consultar Figura 3.20) y c el número de casillas a mover.

A continuación un ejemplo para clarificar el funcionamiento.

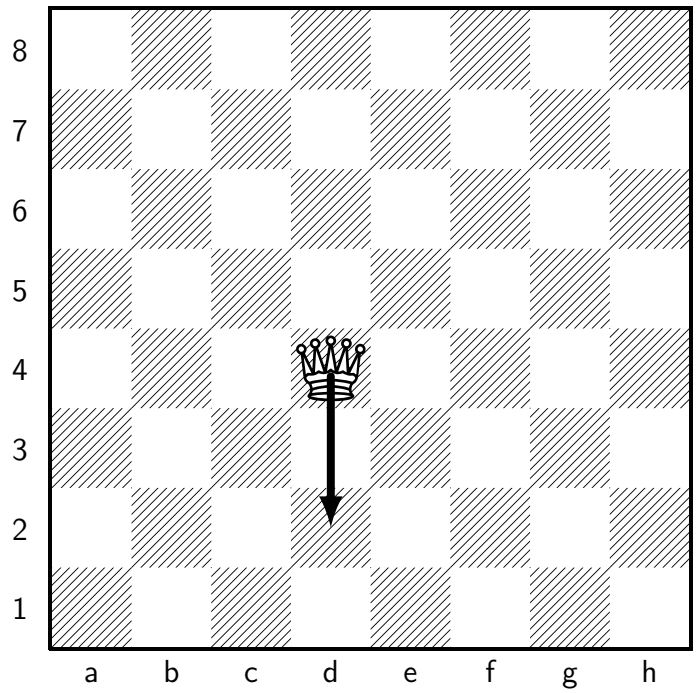


Figura 3.21: Ejemplo de codificación de los movimientos de la dama

La dama tiene como objetivo desplazarse a la casilla señalada por la flecha, específicamente, de d4 a d2. Este movimiento, que se efectúa en dirección sur, tiene asignado un coeficiente de 4. Así, la acción resultante será $4 * 7 + 2 = 30$ en el caso de que el movimiento abarque dos casillas. En relación a las coordenadas de la subsección de 8x8, estaríamos hablando de $(4, 4)$, pues se sitúa en la fila 4 y en la columna "d", la cual tiene un valor numérico equivalente a 4.

En el caso de los movimientos del caballo, el proceso se asemeja. Se aplicará nuevamente el criterio de dirección de las agujas

del reloj, por lo que las casillas ubicadas en la esquina superior derecha recibirán valores de 1 y 2. La fórmula correspondiente sería:

$$f(\beta_s) = 56 + \beta_s$$

En esta fórmula, β_s denota el coeficiente del salto del caballo conforme al sentido de las agujas del reloj. A este coeficiente se le suma 56 para indicar un movimiento de caballo.

La Figura 3.22 proporciona un ejemplo ilustrativo de este esquema de codificación.

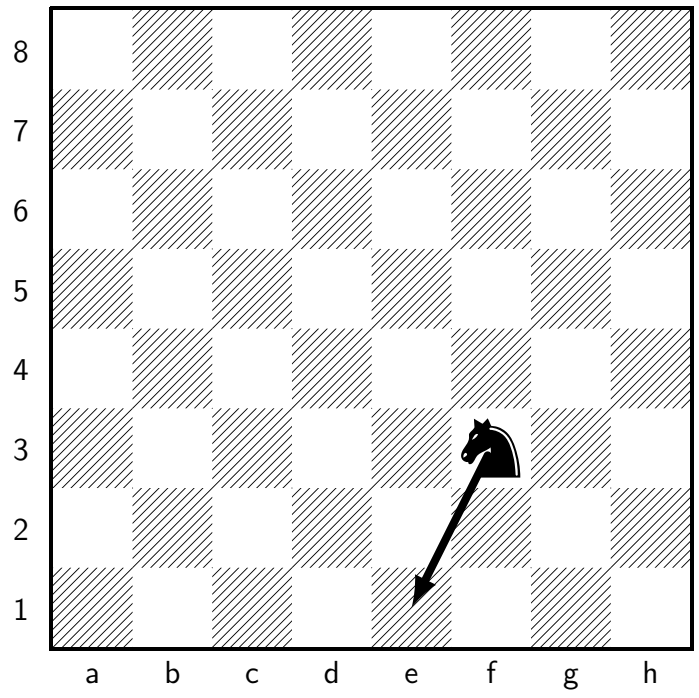


Figura 3.22: Ejemplo de codificación de los movimientos del caballo

En el ejemplo, el caballo localizado en la casilla f3 pretende moverse a la casilla e1. Este movimiento se identifica por un coeficiente de 5, siendo el quinto salto en el sentido de las agujas del reloj, por lo que la acción se computará como $56 + 5 = 61$. Las coordenadas correspondientes serían $(3, 6)$ debido a que el caballo está situado en la casilla f3 (fila 3, columna 6).

Finalmente, se debe considerar la promoción del peón. Un peón puede convertirse en dama, caballo, alfil o torre una vez que alcanza la última fila. Si se transforma en dama, su movimiento se codifica normalmente como tal. Sin embargo, si se promociona a alguna de las otras tres piezas, necesita su propio código de movimiento. Además, cuando un peón se promociona, puede hacerlo avanzando un paso en la diagonal izquierda superior, un paso hacia adelante, o un paso en la diagonal derecha superior. Es decir, habrá $3 * 3 = 9$ movimientos posibles en total. Se aplicará un procedimiento análogo al de los movimientos de la dama, donde se asignará un coeficiente a cada pieza promovida y luego se sumará el movimiento. Los coeficientes a utilizar son:

- Torre: 0
- Alfil: 1
- Caballo: 2

Respecto a los movimientos, se asignarán los siguientes valores:

- Diagonal izquierda superior: 1
- Paso hacia adelante: 2
- Diagonal derecha superior: 3

La fórmula a aplicar será:

$$f(\delta_p, m) = 64 + 3 * \delta_p + m$$

Donde δ_p es el coeficiente de la pieza promovida y m el valor del movimiento correspondiente. Al valor original se le resta uno.

A continuación, se muestra un ejemplo de la promoción de un peón. La Figura 3.23 representa la posición inicial.

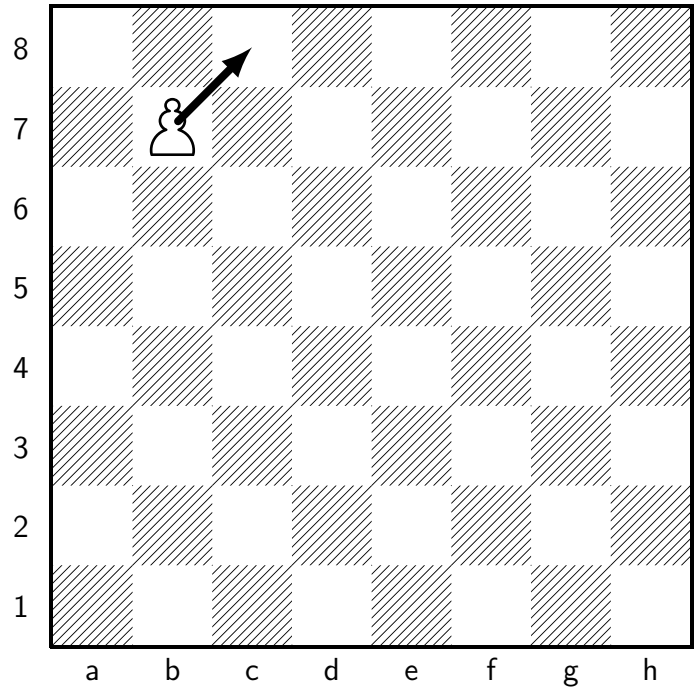


Figura 3.23: Ejemplo de codificación de la coronación de un peón

En este caso, el peón tiene como objetivo promocionarse a alfil, por lo que el coeficiente $\delta_p = 1$. Aplicando la fórmula,

obtenemos que la acción es $64 + 3 * 1 + 3 = 70$ (considerando un movimiento hacia la diagonal derecha superior).

Un posible interrogante podría ser: ¿cómo se representan las promociones en el lado de las piezas negras? Sin embargo, esto no sería necesario ya que el tablero se visualiza siempre desde la perspectiva del jugador, por lo que los peones siempre se promocionarían en la última fila.

Esta representación, no obstante, presenta una problemática. Muchas de las acciones no pueden ocurrir en todas las posiciones. Por ejemplo, si se tiene una dama situada en la esquina inferior izquierda (casilla a1) nunca podrá moverse al sureste, sur, suroeste, oeste y noroeste. En el caso de un caballo en esa casilla, solo podría moverse a 2 de las 8 posibles casillas. Además, un peón solo podrá promocionar si se encuentra en la penúltima fila. Teniendo en cuenta estos factores, Leela Chess Zero pudo reducir el número total de movimientos a 1858 [15]. Para lograrlo se sigue el mismo sistema anterior, pero se eliminan todas las jugadas imposibles.

3.6.1.4. Valor

Afortunadamente, el mecanismo de la salida del valor es considerablemente más simple que el de la política. El proceso comienza con la aplicación de un filtro convolucional, seguido de una normalización de *batch*, y después se implementa una función de activación ReLU. A partir de este punto, se transita de las matrices bidimensionales a un vector, iniciando la aplicación de una capa de perceptrones. Luego se implementa otra función de activación ReLU, seguida de otra capa de perceptrones que consta únicamente de un perceptrón. A la salida de este último, se aplica una función de activación conocida como Tanh.

La función de activación Tanh tiene la característica especial de transformar todos los valores al rango $(-1, 1)$. La definición matemática de esta función es la siguiente:

$$tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

Esta función resulta ideal en relación a la recompensa esperada que también se encuentra dentro de este rango. De esta manera, valores cercanos a -1 indican una posición al borde de la derrota, si es cercano a 0 se interpreta como una posición igualada, y finalmente, si su valor se acerca a 1, la victoria es prácticamente segura. En caso de que esta función reciba un valor muy alto positivo, devolverá un valor cercano a 1, mientras que si este valor alto es negativo, el resultado estará próximo a -1. Estas recompensas esperadas siempre serán evaluadas desde la perspectiva del jugador, de manera que si se lleva la ventaja, las recompensas serán superiores a 0 tanto para las piezas blancas como para las negras.

3.6.2. Árbol de búsqueda Monte Carlo

El Árbol de Búsqueda Monte Carlo (MCTS, por sus siglas en inglés) es uno de los aspectos más sobresalientes de AlphaZero. Este mecanismo de búsqueda arbórea sigue una lógica similar al algoritmo Minimax y su versión mejorada con la poda alfa-beta, pero introduce el componente de simulaciones.

Cuando la inteligencia artificial se encuentra en un estado particular, genera un árbol de manera similar a los algoritmos previamente mencionados, aunque este será mucho más reducido. Este árbol comprenderá los estados que resultan de tomar

una acción en el estado original, y a su vez los estados que surgen de tomar acciones en los estados previamente generados. Este proceso concluye cuando se llega a un nodo hoja del árbol. Cada nodo del árbol conserva la siguiente información:

- N: La cantidad de veces que esa acción ha sido seleccionada en las simulaciones.
- W: El valor acumulativo de este estado según las simulaciones.
- Q: El valor promedio de este estado en función de las simulaciones. Se obtiene dividiendo el valor total entre la cantidad de veces que esa acción ha sido seleccionada. En otras palabras, $Q = W/N$
- P: La probabilidad inicial de seleccionar esa acción (proporcionada por la política de la red neuronal).

El proceso de simulación se realiza un número determinado de veces, que puede estar definido por el tiempo disponible (por ejemplo, todas las simulaciones posibles en un minuto) o por un número fijo de simulaciones. Al seleccionar la acción a tomar, se consideran tres parámetros: Q, N y P. A mayores valores de Q y P, mayores posibilidades tendrá la simulación de ser seleccionada; sin embargo, en contraposición, se busca que N tenga el valor más bajo posible para maximizar la función. N opera de esta manera para facilitar la exploración de aquellos nodos que aún no han sido suficientemente explorados. La fórmula para obtener el valor de la acción sería la siguiente:

$$A = Q + \frac{P}{1 + N}$$

Como se puede observar, al aumentar Q y P se incrementa el valor de A, mientras que al aumentar N se disminuye el valor de A. Se suma 1 a N para evitar divisiones por cero (N puede tener un valor de 0).

En un nodo determinado, se seleccionará la acción que conduzca al estado con el mayor valor V. Este proceso culmina al llegar a un nodo hoja, dando inicio a la fase de actualización. Se obtiene el valor del nodo hoja v utilizando la red neuronal y se llevan a cabo las siguientes actualizaciones en todos los nodos que se han visitado:

$$\begin{aligned} N &:= N + 1 \\ W &:= W + v \\ Q &:= \frac{W}{N} \end{aligned}$$

Por un lado, se incrementa el número de simulaciones realizadas en ese nodo. El valor total W se incrementa al sumarle el valor obtenido en esta simulación, y se recalcula el valor de Q con los valores actualizados de W y N.

Una vez finalizadas todas las simulaciones, se selecciona el mejor movimiento de acuerdo con ellas. Para ello, se elige la acción que ha tenido un mayor número de simulaciones, es decir, que tiene un mayor valor de N. Esta es la estrategia competitiva (que busca jugar de la mejor manera posible), pero puede ser que el sistema esté en entrenamiento y tenga mayor interés en explorar (resurgiendo así el dilema de exploración versus explotación). En este último caso, el sistema generará una distribución de probabilidad basada en el N de los diferentes nodos. Para generar la probabilidad, simplemente suma todas las simulaciones y divide las simulaciones de cada nodo entre este total.

Después de elegir la acción, se descarta todo el árbol excepto el subárbol correspondiente a la acción seleccionada, esto permite reutilizar los cálculos realizados en el anterior paso para calcular las siguientes jugadas. [16] [17].

A continuación, se presenta un ejemplo ilustrativo que tiene por objetivo esclarecer el funcionamiento del árbol de búsqueda Monte Carlo.

Consideremos la posición que se muestra en la Figura 3.24, donde las blancas deben seleccionar entre tres movimientos posibles: Rf3, b3 y Db3.

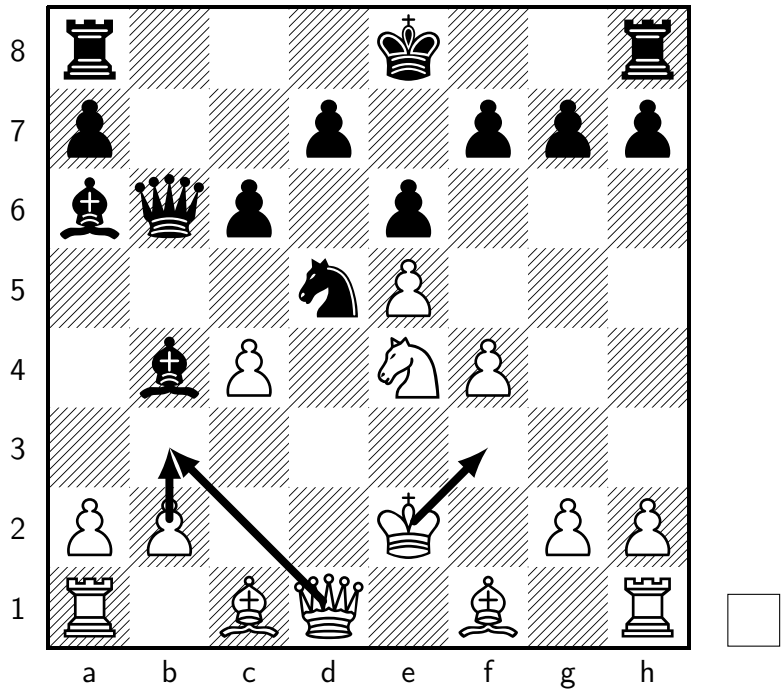


Figura 3.24: Posición de ejemplo para el análisis con MCTS

El agente que juega con las piezas blancas ha generado un

árbol de búsqueda Monte Carlo, y está a punto de llevar a cabo la última simulación del árbol. En el nodo sólo se muestran los valores N, W y P, ya que Q puede calcularse fácilmente a partir de W y N.

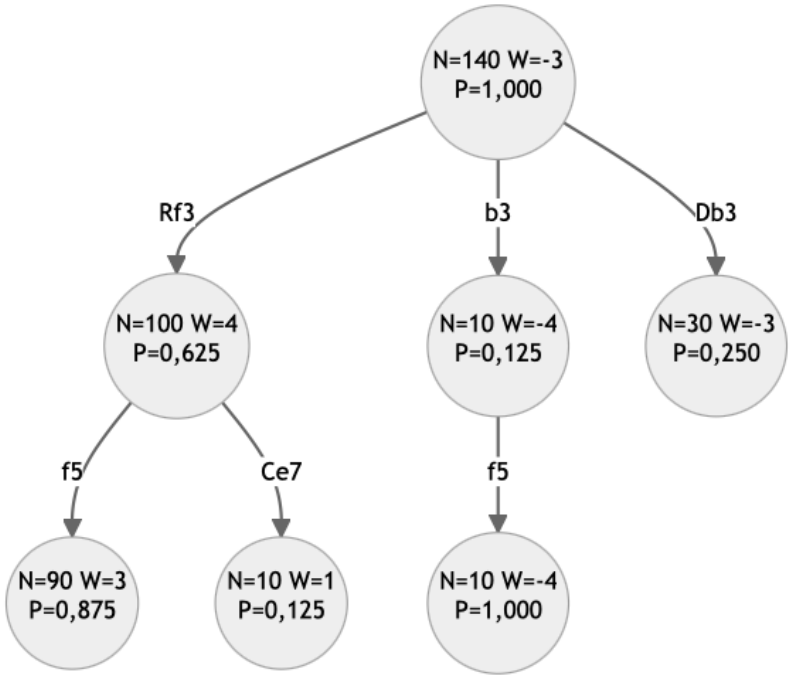


Figura 3.25: Estado del árbol MCTS antes de realizar la última simulación

Se efectúa la última simulación, seleccionando los movimientos Rf3 y f5, ya que son los que presentan el valor más alto según la fórmula empleada para las simulaciones. Al llegar al nodo después de f5, se ejecuta la red neuronal, obteniendo un valor de $v = 0,6$. Con este valor, se procede a actualizar todos los nodos que fueron atravesados durante la simulación, lo que resulta en el siguiente árbol:

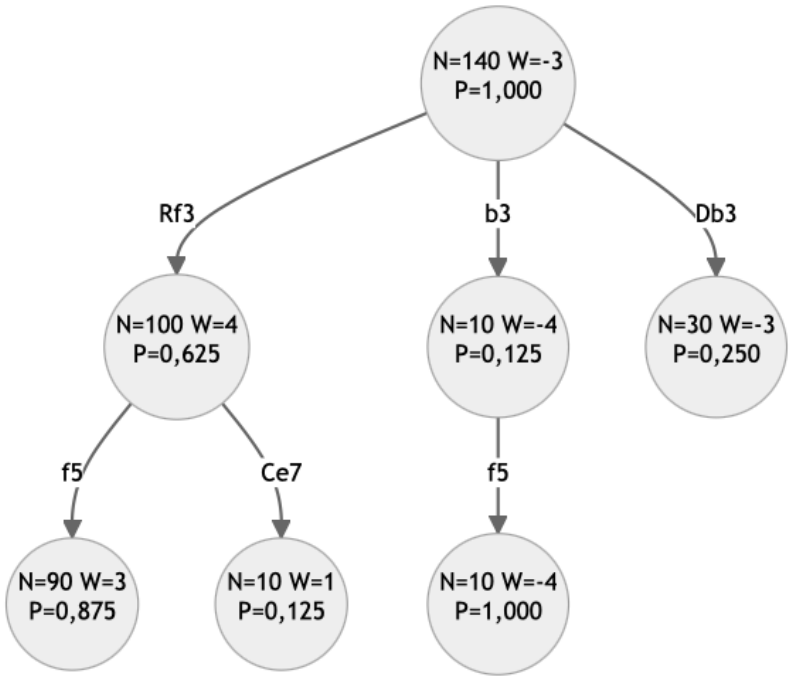


Figura 3.26: Estado del árbol MCTS después de finalizar la última simulación

Una vez que la última simulación se ha completado, es el momento de seleccionar la acción a llevar a cabo. El criterio para la elección se basa en el número de simulaciones N, por lo que la acción seleccionada es Rf3. Después de esta decisión, se descarta todo el árbol excepto la parte que se origina desde el nodo al que ha llevado el movimiento Rf3.

Tras haber estudiado el funcionamiento del árbol de búsqueda Monte Carlo, el próximo paso es examinar cómo fue entrenada y evaluada la inteligencia artificial.

3.6.3. Entrenamiento

El proceso de entrenamiento de la red neuronal se inicia desde cero, sin aportarle ningún conocimiento preexistente sobre

el ajedrez, convirtiéndola en una *tabula rasa*. Su única opción para adquirir conocimientos es jugar partidas contra sí misma, aprender de sus propios errores y, de esta manera, mejorar su rendimiento de manera gradual.

Existen diferencias significativas en el aprendizaje de AlphaZero con respecto a su precursor, AlphaGo Zero. En el caso de AlphaZero, los parámetros de la red neuronal son actualizados constantemente, sin importar los resultados. En contraste, AlphaGo Zero solo actualiza sus parámetros si la red neuronal más reciente logra superar a su versión anterior.

Durante el proceso de entrenamiento, AlphaZero genera una gran cantidad de partidas contra sí mismo. Estas partidas se convierten en el corpus de entrenamiento de la red neuronal. Todas las posiciones de estas partidas son aleatorizadas y, para cada una, se almacenan las probabilidades de búsqueda correspondientes al árbol de búsqueda Monte Carlo, así como el resultado de la partida. Posteriormente, la red neuronal recibe cada posición y devuelve tanto la política (probabilidades de búsqueda) como el valor del estado (resultado de la partida).

Para evaluar el desempeño de la red, se utilizan dos métricas: la función de pérdida de entropía cruzada (Cross-Entropy Loss) para comparar la política devuelta por la red con la almacenada, y la función de pérdida de error cuadrático medio (Mean-Squared Loss) para comparar el valor del estado. Con estas dos funciones se obtiene la pérdida total.

Finalmente, con el fin de prevenir el *overfitting* [4], se introduce un componente de regularización en el cálculo de la pérdida

[4]El *overfitting* o sobreajuste se produce cuando un modelo de redes neuronales se entrena en exceso con un conjunto de datos específico, lo que lleva a aprender patrones que son específicos de esos datos pero que no se generalizan bien a nuevos datos. Esto puede resultar en un rendimiento deficiente cuando se aplica el modelo a nuevas muestras.

total.

Pérdida de Entropía Cruzada (Cross-Entropy Loss)

La función de pérdida de Entropía Cruzada se emplea habitualmente en problemas de clasificación o en la determinación de la política de acciones. Esta función asume que la salida de la red neuronal es una distribución de probabilidad, mientras que la salida real o esperada es un vector en formato *one-hot encoding*. Por ejemplo, si tenemos un total de 5 acciones posibles y la acción escogida es la tercera, el vector en este formato sería el siguiente:

$$\begin{bmatrix} 0 & 0 & 1 & 0 & 0 \end{bmatrix}$$

En este vector, todos los valores son cero excepto en la posición 3, que corresponde a la acción seleccionada y tiene valor 1. Este vector y la distribución de probabilidad se introducen en la fórmula de la función de pérdida de Entropía Cruzada:

$$\mathcal{L}_1(\hat{y}, y) = -\sum_{i=1}^{n} y_i * \log \hat{y}_i$$

Aquí, y representa al vector en formato *one-hot encoding* e $\hat{y}$ al vector de probabilidades. [18]

Una observación importante es que, a pesar de que el vector tiene n elementos, solo el valor en la posición que tiene un 1 afectará al resultado final, pues cuando $y_i = 0$, el término correspondiente en la suma se cancela [5]. Por lo tanto, esta función

[5] En el caso de que se obtenga $0 * \log 0$, se toma por convenio que es igual a 0 para facilitar el entrenamiento

de pérdida evalúa la probabilidad de que se produzca la acción o la clase esperada. Es importante destacar que la función siempre arrojará valores positivos, dado que el término logarítmico proporcionará valores negativos (al recibir valores entre 0 y 1) que se anulan con el signo negativo que antecede al sumatorio.

Pérdida de Error Cuadrático Medio (Mean-Squared Loss)

La función de pérdida de Error Cuadrático Medio se utiliza con frecuencia en problemas de regresión, al igual que la función de pérdida de Entropía Cruzada. En este caso, esta función se usa para aproximar la recompensa esperada. A diferencia de la función de Entropía Cruzada, su funcionamiento es más simple, ya que compara la diferencia entre dos valores numéricos mediante la siguiente fórmula:

$$\mathcal{L}_2(\hat{y}, y) = (\hat{y} - y)^2$$

Para calcular esta pérdida, se resta la predicción de la red neuronal al valor real y se eleva al cuadrado el resultado, asegurando así que la pérdida sea un valor positivo. Si esta pérdida perteneciera a un *batch* (es decir, en una sola pasada de la red neuronal se calculan varias entradas), se tendría que dividir este valor entre n, siendo n el número de entradas. La fórmula quedaría de la siguiente forma:

$$\mathcal{L}_2(\hat{y}, y) = \frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2$$

Donde $\hat{y}_i$ y y_i corresponden respectivamente a los valores de la posición i en la predicción de la red neuronal y el valor real

[19]. Se aplicaría de la misma manera a la Entropía Cruzada si fuera un *batch* calculando la media entre todas.

Regularización

AlphaZero emplea un tipo de regularización conocido como L2, una de las técnicas más utilizadas en el campo de la inteligencia artificial. El propósito de la regularización es disminuir la complejidad del modelo y evitar el sobreajuste, o *overfitting*. Para lograr esto, se añade un término adicional a la función de pérdida general, que incrementa su valor a medida que la complejidad del modelo aumenta. Pero ¿cómo se mide dicha complejidad? En realidad, es bastante sencillo: se hace uso de los propios pesos de la red neuronal. A mayor distancia de estos pesos con respecto a cero, mayor será la complejidad del modelo.

La regularización L1 calcula el valor absoluto de los pesos, mientras que la L2 calcula el cuadrado de los pesos. La fórmula de la regularización L2 es la siguiente:

$$L2(w) = \alpha * \sum_{i=1}^{n} (w_i)^2$$

En esta fórmula, w representa todos los pesos de la red neuronal y α es un parámetro de regularización con un valor muy pequeño, típicamente de 0,01 o 0,001, o incluso menor. La pérdida total del modelo, que se emplea para la retropropagación o *backpropagation* para calcular la actualización de los pesos de la red neuronal, se obtiene sumando las dos funciones de pérdida descritas anteriormente y el término de regularización L2 [20].

El *overfitting* ocurre cuando el modelo se ajusta muy bien a los datos de entrenamiento, pero tiene un pobre rendimiento en

la generalización para entradas nuevas o no vistas anteriormente. Dado que el *overfitting* indica que el modelo es demasiado complejo, la regularización puede ser un método efectivo para mitigar este problema.

Por lo tanto, la pérdida total del modelo se calcula de la siguiente manera:

$$\mathcal{L} = \mathcal{L}_1 + \mathcal{L}_2 + L2$$

3.6.4. Evaluación

Tras la finalización del proceso de entrenamiento de la red neuronal, se vuelve imprescindible evaluar su rendimiento en comparación con el de los humanos y otros programas computacionales. No obstante, considerando que en el juego del ajedrez, los programas informáticos han superado con creces a los jugadores humanos durante muchos años, el benchmarking de nuestra red se realizará, por tanto, solo contra programas informáticos.

En esta línea, el equipo de AlphaZero optó por contrastar su desempeño con el programa que fue proclamado campeón del Top Chess Engine Championship (TCEC) en el año 2016. Dicho programa es conocido con el nombre de Stockfish, y se caracterizaba en aquel momento por poseer una arquitectura sustentada en la técnica de poda alfa-beta y una serie de heurísticas de gran complejidad.

En el enfrentamiento consistente en un total de 100 partidas, AlphaZero logró un impresionante registro de 28 victorias y 72 empates contra este formidable rival. Estos excelentes resultados provocaron una revolución sin precedentes en el campo de los

programas de ajedrez, poniendo en evidencia el potencial de la inteligencia artificial y las redes neuronales en el dominio de este milenario juego de estrategia.

3.6.5. Implicaciones

El cambio más notorio que se ha producido en el ámbito de la inteligencia artificial fue la transición desde el uso de heurísticas diseñadas y codificadas por los humanos hasta la adopción de heurísticas generadas a través de redes neuronales. Esta transformación marcó un punto de inflexión en cómo las máquinas aprenden, interpretan y toman decisiones basadas en los datos que procesan.

Este cambio, sin embargo, tuvo como consecuencia que los módulos que todavía utilizaban las heurísticas antiguas se encontraban en desventaja frente a aquellos que habían incorporado la nueva tecnología. Estos módulos más antiguos, que una vez dominaron sus respectivos campos, carecían de la capacidad para competir efectivamente con los módulos basados en redes neuronales, resultando en una disparidad considerable en términos de eficiencia y precisión.

No obstante, es importante destacar que muchos de estos sistemas basados en heurísticas tradicionales han podido adaptarse a esta nueva era de aprendizaje automático. Un ejemplo notable es el del programa de ajedrez Stockfish. Este motor de ajedrez, que una vez se basó en heurísticas diseñadas manualmente, ha logrado integrar con éxito una red neuronal en su arquitectura, lo que ha potenciado considerablemente su rendimiento. Esta adaptación no solo ha permitido a Stockfish mantenerse competitivo frente a sus contemporáneos basados en redes neuronales, sino que también ha demostrado la versatilidad y la capacidad

de adaptación inherentes a estos sistemas. Veremos en más detalle como logró Stockfish integrar las redes neuronales en el próximo capítulo.

Capítulo 4

Estado del arte

El término "estado del arte", una frase de origen anglosajón, es utilizado para describir los avances más recientes y significativos en un área de estudio o campo específico. Consecuentemente, el presente capítulo se dedica a explorar y analizar estos avances más novedosos y revolucionarios que han surgido en el ámbito del ajedrez, en particular durante el período comprendido entre los años 2020 y 2023. Esta etapa representa la vanguardia de la investigación en este juego milenario, siendo un testigo de la constante evolución y progreso que caracteriza al estudio del ajedrez.

Por una parte, examinaremos el fenómeno de cómo Stockfish, uno de los programas de ajedrez más robustos y precisos, logró sobreponerse tras su derrota frente a AlphaZero, y cómo retomó su lugar en la élite de los módulos de ajedrez. Detallaremos los cambios estratégicos y las optimizaciones técnicas que permitieron a este potente motor de ajedrez reclamar su antigua gloria y prestigio.

Por otra parte, nos desviaremos completamente del objetivo tradicionalmente analizado en los capítulos anteriores, que es jugar al ajedrez de la mejor forma posible desde un punto de vista

táctico y estratégico. En lugar de eso, nos enfocaremos en un enfoque emergente y fascinante en el campo de la inteligencia artificial aplicada al ajedrez: la imitación de la toma de decisiones humanas durante una partida de ajedrez. Analizaremos las técnicas y métodos utilizados en esta línea de investigación, y evaluaremos cuán exitosos han sido estos intentos por replicar y emular la complejidad y sutileza del pensamiento humano en el ajedrez.

4.1. Stockfish contraataca

El equipo de desarrollo de Stockfish, lejos de quedarse de brazos cruzados tras su derrota frente a AlphaZero, decidió adoptar una estrategia que recuerda al viejo dicho: "Si no puedes vencer a tu enemigo, únete a él". De este modo, optaron por incorporar redes neuronales en la estructura de Stockfish. Su objetivo principal era mantener la arquitectura original del programa, pero reemplazar la heurística desarrollada por programadores con una basada en una red neuronal. Esta modificación permitiría a Stockfish evaluar las posiciones del juego con una precisión mucho mayor, especialmente las posiciones "tranquilas".

Sin embargo, esta nueva estrategia presentaba un desafío importante. Al implementar redes neuronales en la heurística, se incrementaría considerablemente el coste computacional de su cálculo, lo que reduciría la velocidad de la búsqueda. Las heurísticas basadas en redes neuronales, con su gran número de capas, tienen un coste computacional muy superior al de las heurísticas empleadas por módulos de ajedrez tradicionales. Por esta razón, AlphaZero optó por utilizar un árbol de búsqueda Monte Carlo en lugar de la poda alfa-beta.

Para la fortuna de los desarrolladores de Stockfish, la solución a este problema surgió de un lugar inesperado: el mundo del Shogi. El Shogi, un juego de estrategia muy popular en Japón y con muchas similitudes al ajedrez, permitiendo la transferencia de muchos conceptos entre ambos juegos. En este contexto, surgió una nueva arquitectura de red neuronal llamada NNUE (de sus siglas en inglés, "Efficiently Updateable Neural Networks") [21]. Esta arquitectura buscaba crear una red neuronal que pudiera calcularse con mucha rapidez, lo cual ofreció un impulso significativo para el desarrollo de Stockfish.

La red neuronal tipo NNUE se diseñó con el propósito primordial de maximizar su velocidad de cálculo. La premisa es simple: a mayor rapidez de cálculo, mayor será el número de nodos que se pueden explorar y, por tanto, mejor será la jugada que se encuentre. Las NNUE se caracterizan por tener un número de capas muy reducido (es poco común encontrar más de cinco) y por utilizar capas de perceptrones.

Para adecuar la representación de una posición de ajedrez a una capa de perceptrones, es necesario adoptar un enfoque completamente novedoso. Aquí es donde entra en juego el formato conocido como HalfKP, una representación innovadora y eficiente que permite transformar las posiciones del ajedrez en un formato adecuado para el uso con redes neuronales basadas en perceptrones.

El término HalfKP se deriva de Half-King-Piece, un nombre que refleja claramente su mecanismo de funcionamiento. El concepto fundamental detrás de HalfKP es establecer una relación entre la posición de los reyes de ambos bandos y la disposición de las demás piezas en el tablero de ajedrez.

En concreto, HalfKP emplea una representación binaria (solo

toma valor 0 o 1) para indicar la presencia de un cierto tipo de pieza en una posición específica, condicionada por la posición en la que se encuentre tu rey. Por ejemplo, si tu rey está situado en la casilla e1 y un caballo de tu bando se encuentra en la casilla b1, entonces la entrada correspondiente a estas coordenadas en la representación binaria tendrá un valor de 1. Este procedimiento se aplica de manera análoga para el bando contrario.

La representación binaria consta de un total de $64 \times 64 \times 8 \times 2 = 81920$ entradas. El primer factor de 64 corresponde a las 64 casillas posibles para el rey, el segundo 64 hace referencia a las diferentes ubicaciones en las que podría encontrarse cualquier otra pieza, el 8 se refiere a los distintos tipos de piezas posibles excluyendo a los reyes (es decir, peón, torre, caballo, alfil, dama y sus contrapartes negras), y finalmente, el 2 indica que este proceso se lleva a cabo para los reyes de ambos bandos.

En contraste, la salida de este sistema es mucho más simple y consta de un solo valor numérico que representa la evaluación de la posición en términos de centipeones, donde 1 peón equivale a 100 centipeones. Esta es una de las unidades de medida más comúnmente utilizadas para evaluar la posición en una partida de ajedrez.

A continuación se presenta un ejemplo del funcionamiento de la entrada dado su complejidad. Dado el tablero mostrado en la Figura 4.1.

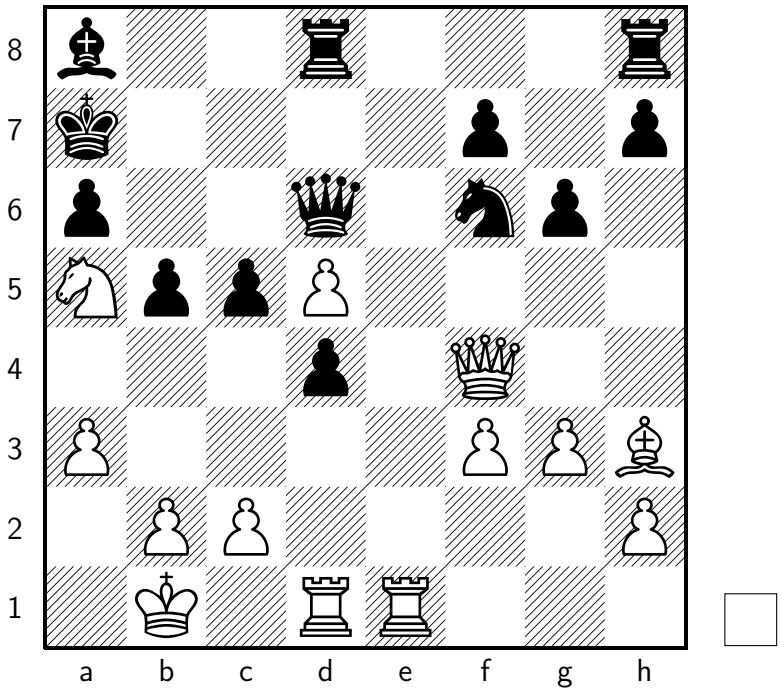


Figura 4.1: Tablero de ejemplo para HalfKP

Vamos a tener marcado con 1 las siguientes entradas:

- Rey propio b1 y peón propio a3
- Rey propio b1 y peón propio b2
- Rey propio b1 y peón propio c2
- Rey propio b1 y peón propio d5
- Rey propio b1 y peón propio f3
- Rey propio b1 y peón propio g3
- Rey propio b1 y peón propio h2

- Rey propio b1 y peón rival a6
- Rey propio b1 y peón rival b5
- Rey propio b1 y peón rival c5
- Rey propio b1 y peón rival d4
- Rey propio b1 y peón rival f7
- Rey propio b1 y peón rival g6
- Rey propio b1 y peón rival h7
- Rey propio b1 y caballo propio a5
- Rey propio b1 y caballo rival f6
- Rey propio b1 y alfil propio h3
- Rey propio b1 y alfil rival a8
- Rey propio b1 y torre propia d1
- Rey propio b1 y torre propia e1
- Rey propio b1 y torre rival d8
- Rey propio b1 y torre rival h8
- Rey propio b1 y dama propia f4
- Rey propio b1 y dama rival d6

Para el monarca adversario, emplearíamos una designación similar. Solo deberíamos reemplazar la frase "Rey propio b1" por "Rey rival a7", invirtiendo los términos 'propio' y 'rival' en las piezas de la lista. Aquellas casillas que no aparezcan en

ninguna de las listas permanecerán a cero. Con respecto a los valores de entrada, es importante destacar que ciertas entradas nunca podrán tener un valor de 1. Por ejemplo, en una situación donde simultáneamente se presente 'rey propio e1' y 'pieza e1'. A pesar de ello, para facilitar el procesamiento, se suelen mantener estas posiciones.

Es notable que esta representación contiene cierta redundancia, es decir, incluye información adicional. Esta característica se evidencia en la repetición de la colocación de las piezas para cada rey. Sin embargo, tal redundancia ofrece ciertas ventajas, especialmente en el contexto de las redes neuronales, las cuales son capaces de optimizar sus resultados en esta circunstancia. Otro beneficio de esta representación radica en su facilidad para realizar actualizaciones. Si se desplaza una pieza, solo es necesario asignar dos de las entradas a 1 y otras dos a 0. En contraste, si se mueve uno de los reyes, se requerirá realizar una mayor cantidad de modificaciones. Afortunadamente, en el ajedrez, los movimientos del rey son infrecuentes, excepto en las etapas finales del juego.

Además, esta representación brinda una ventaja por su simplicidad al "reflejar" la posición en el tablero. Si deseamos intercambiar las piezas blancas por las negras en una posición dada, solo debemos intercambiar la sección correspondiente a nuestro rey por la del rey adversario. Al mantener esta representación siempre orientada hacia el jugador que tiene el turno, permite una rápida adaptación a las diferentes configuraciones del juego.

Para entrenar este tipo de red neuronal se utiliza una combinación de aprendizaje supervisado y reforzado. En el aprendizaje supervisado, se emplean partidas de alta calidad que involucran a los grandes maestros más fuertes y partidas entre módulos, es-

pecialmente de la versión *open source*de AlphaZero, Leela Chess Zero. Después de completar esta etapa de entrenamiento supervisado, se procede a combinarla con el aprendizaje reforzado, en el cual se juegan partidas contra sí mismo, siguiendo el enfoque utilizado por AlphaZero.

En cuanto a la red neuronal, cuenta con una primera capa de 256 perceptrones, cada uno de los cuales puede recibir un total de 40960 entradas. Los 256 valores generados por esta capa se transfieren a una segunda capa que consta de 32 perceptrones. Los resultados de esta segunda capa se procesan en una tercera capa también de 32 perceptrones. Finalmente, estos últimos consolidan sus resultados en una salida única, la cual se ha comentado previamente [22].

En la primera capa, se manejan 40960 entradas, pero, ¿no se mencionó que se tienen en total 81920 entradas? Efectivamente, eso es cierto, pero la entrada se divide entre la sección del rey propio y la del rey rival. Cada una de estas secciones se procesa de forma independiente por los mismos perceptrones y se transfieren a la segunda capa, resultando en un total de 512 entradas en lugar de las 256 inicialmente esperadas debido a los 256 perceptrones de la primera capa.

Una manera de conceptualizar este proceso es pensar que la red neuronal está evaluando la disposición de las piezas en relación a cada rey, intentando determinar cuál de los dos bandos presenta una situación más favorable.

El empleo de esta red neuronal conlleva una serie de ventajas para su implementación en el ámbito de la computación. Estos beneficios incluyen la capacidad de reutilizar cálculos ya realizados a partir de otras posiciones, lo que se logra mediante mínimas variaciones en las entradas. Además, la estructura de

la red permite ejecutar diversos cálculos de manera paralela, es decir, realizar varias operaciones simultáneamente. Este aspecto facilita un notable incremento en la velocidad de procesamiento, contribuyendo a un desempeño más eficiente y rápido de la red neuronal en el análisis de las posiciones en el juego de ajedrez.

Tras la implementación de esta mejora, Stockfish ha conseguido reafirmar su posición entre los mejores módulos de ajedrez a nivel mundial. Sin embargo, la competencia para determinar el mejor programa de ajedrez es un evento anual de alto nivel y la contienda es feroz. Participan múltiples programas distintos, pero aquellos que generalmente ocupan las posiciones de liderazgo emplean enfoques similares al de Stockfish o al de AlphaZero/Leela Chess Zero. La continua innovación y mejora en las estrategias de programación de ajedrez aseguran una competencia emocionante e incierta, mostrando el gran avance en la intersección de la inteligencia artificial y el juego de ajedrez.

4.2. Imitando la toma de decisiones humana

A lo largo de este texto hemos profundizado en una multitud de estrategias orientadas a alcanzar un objetivo singular: desarrollar habilidades superlativas en el juego de ajedrez. Aunque la meta de alcanzar la perfección en este juego de estrategia todavía está lejos de ser alcanzada, y no se espera que se logre en un futuro próximo, existen otros objetivos fascinantes que merecen atención. Uno de estos propósitos destacados es el de "imitar" el comportamiento humano. Ahora bien, ¿en qué consiste precisamente esta imitación del comportamiento humano?

La imitación del comportamiento humano en este contexto se traduce en la capacidad de jugar al ajedrez de manera semejante

a cómo lo haría una persona. El aspecto crucial a entender aquí es que los programas de ajedrez contemporáneos no reflejan un estilo de juego que se asemeje significativamente al de un ser humano. Esto ocurre debido a que el método de determinar las jugadas es absolutamente distinto.

A lo largo de este texto hemos detallado cómo los ordenadores utilizan una combinación de exploración en el árbol de juego y el empleo de heurísticas para tomar decisiones. En contraste, los seres humanos utilizamos los mismos componentes, pero de manera completamente diferente. Un humano no tiene la capacidad de explorar más de 100 posiciones en su mente, incluso los individuos más dotados, mientras que un ordenador puede explorar millones de posiciones en cuestión de segundos. No obstante, la calidad de la heurística humana, es decir, la habilidad de evaluar una posición y discernir cuáles son las jugadas con mayor probabilidad de suceder, es notablemente superior. AlphaZero, en cambio, representa un punto intermedio donde consigue tener una heurística más avanzada, pero su habilidad para buscar entre nodos es menor en comparación con los programas de ajedrez tradicionales.

La Tabla 4.1 proporciona una comparación detallada de las diversas maneras en las que se puede abordar el ajedrez, desde la perspectiva humana y de la máquina. Las aproximaciones mostradas en la tabla pueden variar en función del tiempo dedicado a analizar una posición concreta.

Modelo	Nodos explorados	Calidad de la heurística
Humanos	<100	Muy buena
AlphaZero	≈10000	Buena
Programas tradicionales	>1000000	Mala

Tabla 4.1: Comparación de uso de búsqueda y heurísticas entre humanos y programas de ordenador

La notable discrepancia entre el estilo de juego de los programas de ajedrez y los jugadores humanos hace que sea un desafío considerable para los primeros imitar con precisión el comportamiento de los últimos. Sin embargo, AlphaZero, con su enfoque innovador, ha logrado exhibir un estilo de juego más "humano" en comparación con los programas tradicionales.

Un interrogante fundamental en este punto es ¿qué es lo que distingue el juego humano del de los ordenadores? Entre las diferencias más destacables se encuentra la habilidad para evaluar posiciones estratégicas. Estas situaciones se caracterizan por no ser un enfrentamiento directo entre los competidores, sino más bien una maniobra sutil en las que los jugadores reubican sus piezas detrás de sus líneas de defensa, buscando una ventaja estratégica que pueda llevarlos a la victoria al momento del choque decisivo. En estas circunstancias, los jugadores humanos solían tener un desempeño superior a sus contrapartes de silicio, al menos hasta la aparición de AlphaZero.

Los ordenadores, por otro lado, destacan en posiciones tácticas, donde un solo movimiento puede ser crucial para el desenlace de la partida. Gracias a su capacidad superior para explorar extensivamente el árbol de juego, pueden identificar con mayor facilidad el momento clave o crítico.

La distinción entre el juego humano y el de los ordenadores es especialmente evidente en las posiciones puramente estratégicas

y tácticas. En las estratégicas, los ordenadores a veces pueden parecer erráticos o sin una dirección clara. En cambio, en las posiciones tácticas, pueden realizar movimientos que inicialmente pueden parecer incomprensibles para un humano, pero que tras varias jugadas, se revelan como la opción más óptima.

Para abordar los desafíos que se presentaban, se desarrolló un nuevo modelo basado en el aprendizaje profundo, conocido como Maia. Este modelo consta de dos versiones: La primera, Maia [23], tiene como propósito imitar las estrategias de jugadores que se encuentran dentro de un rango de ELO especificado. En otras palabras, reproduce el estilo de juego promedio de los jugadores dentro de ese rango. La segunda versión, Transfer Maia [24], lleva el concepto un paso más allá, replicando los movimientos de jugadores específicos. Esto implica que, gracias a Transfer Maia, puedes tener la oportunidad de jugar contra los estilos de juego de los mejores jugadores de ajedrez de hace cien años.

Ambas versiones de Maia utilizan una red neuronal que es muy similar a Leela Chess Zero, la cual es una versión de código abierto de Alphazero, aunque existen diferencias notables. La primera es que se ha disminuido el número de bloques en el extractor de características. Los autores del estudio encontraron que añadir más bloques solo mejora marginalmente el rendimiento de la red [23]. Sin embargo, el cambio más sustancial en la red neuronal reside en la modificación de los objetivos y los datos utilizados para el entrenamiento. Como se ha expuesto en este libro, las redes neuronales son capaces de desarrollar cualquier algoritmo siempre y cuando se les proporcionen los datos adecuados. Por lo tanto, una misma red neuronal puede adaptarse a objetivos muy distintos simplemente modificando los datos de entrada.

Anteriormente, tanto AlphaZero como Leela Chess Zero requerían generar sus propios datos de entrenamiento a través de jugar partidas contra sí mismos. Ahora, con Maia, el proceso es considerablemente más sencillo: simplemente necesitamos buscar partidas del rango de ELO o del jugador de interés.

En términos de entrenamiento, es importante tener en cuenta que tenemos dos salidas: valor y política, y que cada posición en el juego será nuestra entrada [1]. Para el valor, Maia utiliza una ligera variación: la salida corresponde a las probabilidades de ganar/empatar/perder desde la posición actual para el jugador medio o el jugador específico. Esta probabilidad se compara con el resultado real desde la perspectiva del jugador, independientemente de si juega con las piezas blancas o negras. En cuanto a la política, simplemente se compara la salida de la red neuronal con el movimiento que el jugador realizó en la partida. La salida de la red neuronal brinda la probabilidad de cada movimiento legal posible. Para lograr esto, se aplica una máscara de movimientos legales, que permite solo considerar estos y descartar los movimientos no legales. Todas las partidas las han extraido de la plataforma de ajedrez *online* Lichess.

El modelo Transfer Maia implementa una estrategia de entrenamiento que se asemeja a la convencional, pero con la particularidad de que inicia con los pesos predefinidos del modelo correspondiente al jugador medio que presenta un rango de ELO más cercano al del jugador que se pretende emular. Este proceso se conoce como *transfer learning*, una técnica muy recurrida que permite reducir el tiempo de entrenamiento de una red neuronal al aprovechar los pesos ya calculados de otro modelo.

[1] Igual que AlphaZero, Maia también toma en cuenta las posiciones previas de la partida en la entrada

Los resultados obtenidos a partir de estos modelos son sólidos, aunque no extraordinarios. Para evaluar su desempeño se utiliza una métrica conocida como precisión, que compara la jugada con mayor probabilidad de ser seleccionada por la red neuronal con la jugada que realmente fue efectuada en esa posición. De acuerdo con esta métrica, Maia sobresale, obteniendo mejores resultados en comparación con Stockfish y Leela Chess Zero. En términos concretos, la precisión de Maia orientado al jugador promedio oscila entre el 48 % y el 52 %, dependiendo del rango de ELO. Esto implica que, de 100 posiciones, acierta en aproximadamente 50. Aunque este resultado supera a los modelos previos (con una precisión inferior al 45 %), la probabilidad de acierto es comparable a la de obtener cara al lanzar una moneda. Como norma general, la precisión tiende a aumentar con el rango de ELO. Mi hipótesis acerca de esta tendencia se basa en que los estilos de juego en niveles más bajos suelen ser más variables, mientras que a niveles más altos los estilos de juego tienden a ser más uniformes. El modelo orientado a jugadores específicos muestra un rendimiento superior, con una precisión entre el 55 % y el 62.5 %, lo que indica claramente mejores probabilidades que el simple lanzamiento de una moneda. Además, según los autores del estudio, son capaces de identificar a los jugadores con una precisión del 95 % utilizando su respectivo modelo [24], lo que podría tener numerosas aplicaciones en la detección de trampas en el ajedrez *online*.

La explicación de que los resultados de estos modelos no sean superiores probablemente reside en que la posición en el tablero no es el único factor que determina las jugadas de un jugador. Existen otros aspectos que pueden influir. Un factor determinante es el tiempo. Los jugadores de ajedrez cuentan con un

tiempo limitado para realizar sus jugadas, y si se agota, pierden la partida automáticamente. Por tanto, la forma de jugar variará significativamente si se cuenta con pocos segundos en contraposición a tener una hora en el reloj. Este factor es fácilmente cuantificable en el ajedrez *online* y podría representar una valiosa mejora para Maia. Otras variables que podrían tener influencia incluyen el estado de ánimo del jugador (la toma de decisiones puede variar dependiendo de si el jugador se encuentra estresado, enojado o en un estado normal) e incluso el entorno de juego (jugando en persona o *online*) [25]. Estos factores son muy complicados de medir para ser usados por una red neuronal.

En resumen, a pesar de que los resultados de ambos modelos de Maia son prometedores, la complejidad inherente al problema dificulta la obtención de una solución superior, dado que muchos de los factores que influyen en la toma de decisiones humanas son difíciles de medir. No obstante, este campo de investigación es extremadamente fascinante y sus hallazgos podrían tener aplicaciones en otros juegos, como Go o Shogi.

Capítulo 5

Estándares

En este pequeño capítulo abordaremos algunos de los estándares fundamentales implementados en los programas informáticos de ajedrez. Pero antes, debemos preguntarnos: ¿qué es un estándar?

Un estándar es una especificación técnica, o un conjunto de directrices y reglas diseñadas para garantizar la interoperabilidad y la compatibilidad entre diferentes sistemas, dispositivos o aplicaciones. En el contexto específico de la informática, los estándares aseguran que los programas desarrollados por diversas organizaciones y personas puedan ser compatibles en numerosos aspectos, potenciando así sus funcionalidades y permitiendo una interacción fluida entre ellos.

Los estándares que rigen el ajedrez informático son múltiples y variados. En este libro, sin embargo, nos centraremos en tres de los más trascendentales.

El primero de ellos es FEN (Forsyth-Edwards Notation), un estándar utilizado para representar posiciones de ajedrez. El nombre de este estándar proviene de sus creadores, David Forsyth, quien fue el primero en desarrollar este sistema, y Steven Edwards, quien lideró la especificación tanto del estándar FEN

como del PGN (Portable Game Notation), que es el segundo estándar que analizaremos.

PGN es empleado para describir de forma precisa una partida de ajedrez completa, desde su inicio hasta su conclusión. Dado que en algunas ocasiones la posición inicial de una partida no comienza con la posición inicial habitual, es necesario especificar esta posición utilizando FEN, de modo que estos dos estándares se complementan y trabajan conjuntamente.

Finalmente, exploraremos el último estándar, conocido como UCI (Universal Chess Interface). Este cumple una doble función: por un lado, especifica cómo deben representarse los movimientos en ajedrez para que puedan ser interpretados sin dificultades por los módulos de ajedrez; por otro lado, regula la comunicación entre las interfaces gráficas de los programas de ajedrez y el módulo respectivo, que es el programa encargado de determinar las jugadas. La conjunción de estas especificaciones contribuye a una experiencia de ajedrez digital más fluida y efectiva. Esto se materializa en que podamos usar cualquier módulo en cualquier interfaz.

5.1. FEN

La Notación de Forsyth-Edwards (FEN, por sus siglas en inglés) establece un estándar universal que puede ser interpretado por humanos y ordenadores de manera sencilla y eficiente. Este estándar se utiliza para representar posiciones de ajedrez, siendo su principal finalidad contener toda la información relevante de una posición para que se pueda reproducir con exacti-

tud y sin obstáculos. Se emplea el formato de texto plano[1] en la notación FEN, lo que facilita su interpretación por humanos y su procesamiento por ordenadores.

La notación FEN se divide en seis campos, cada uno separado por espacios. A continuación, se detalla el contenido y la interpretación de cada uno de estos campos:

1. **Colocación de las piezas:** Define la disposición de las piezas en el tablero, fila por fila, comenzando desde la esquina superior izquierda. Cada pieza se representa por su letra identificativa en inglés (consultar la Tabla C.1 para las correspondencias exactas de las piezas). Las piezas blancas se denotan con mayúsculas y las negras con minúsculas. Si existen espacios vacíos entre las piezas, se añade un número que representa la cantidad de espacios. Las filas se separan con '/'.

2. **Bando:** Indica el turno de juego. Se utiliza 'w' para indicar que es el turno de las blancas y 'b' para las negras.

3. **Enroques:** Describe las posibilidades de enroque. Se coloca un '-' si no hay opciones de enroque para ninguno de los bandos. En caso contrario, se añade una 'K' para un enroque corto y una 'Q' para un enroque largo. Si la letra está en mayúscula, corresponde a las blancas; si está en minúscula, a las negras.

4. ***En passant*:** Señala la posibilidad de realizar una jugada de comer al paso. Se utiliza un '-' si no es posible; de lo

[1]En el campo de la informática, los archivos se pueden categorizar en dos tipos: texto plano y binario. Mientras que el texto plano es legible tanto para humanos como para ordenadores, el binario es interpretable solo por máquinas. A pesar de que los archivos binarios consumen menos memoria que su equivalente en texto plano, este último se prefiere por su simplicidad y universalidad.

contrario, se indica la casilla por donde ha pasado el peón después de mover dos espacios, es decir, la casilla en la que terminaría el peón oponente si realizara la captura al paso.

5. **Movimientos desde la última captura o avance de peón:** Proporciona el número de jugadas transcurridas sin que haya habido capturas ni movimientos de peón, lo que se relaciona con la regla de los 50 movimientos. Un "movimiento"se considera completo cuando ambos jugadores han realizado una jugada. Por lo tanto, esta regla se activaría cuando este contador alcance 100.

6. **Contador de movimientos:** Representa el número total de movimientos realizados desde el inicio de la partida.

Para aclarar estos conceptos, se propone un ejemplo a continuación. La Figura 5.1 muestra una posición de ajedrez que representaremos usando la notación FEN.

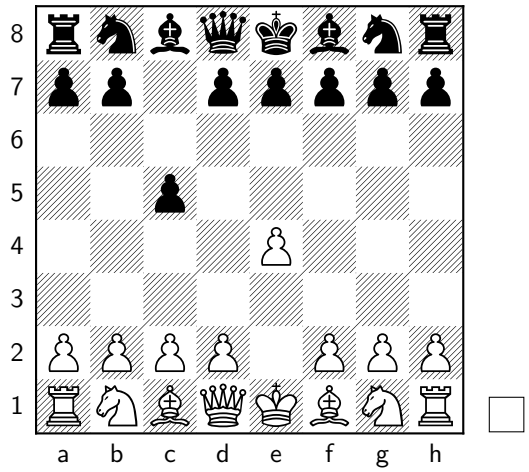


Figura 5.1: Tablero de ejemplo para FEN

La disposición de las piezas se expresaría así (las filas se han dispuesto de manera separada para facilitar la comprensión, pero en realidad se separan con '/' tal y como se ha mencionado anteriormente):

```
rnbqkbnr
pp1ppppp
8
2p5
4P3
8
PPPP1PPP
RNBQKBNR
```

El turno del bando correspondería a las blancas, por lo que se escribe 'w'. Ambos bandos tienen todas las opciones de enroque disponibles, lo que se representa como 'KQkq'. No hay posibilidad de captura al paso, por lo que se coloca '-'. La última jugada fue c5, lo que indica que han pasado 0 movimientos desde la última jugada de "avance", y al encontrarse en el segundo movimiento de la partida, se indica con un 2 en el último campo.

La representación FEN final de la posición es la siguiente (todo se escribe en la misma línea):

```
rnbqkbnr/pp1ppppp/8/2p5/4P3/8/PPPP1PPP/RNBQKBNR
w KQkq - 0 2
```

La notación FEN, aunque útil, tiene ciertas limitaciones. No especifica si ha habido repeticiones consecutivas de la misma posición (según las reglas del ajedrez, tres repeticiones de la misma

posición resultan en un empate). Para subsanar este inconveniente, se ha desarrollado la Descripción de Posición Extendida (EDP, por sus siglas en inglés), una extensión de FEN que incluye dicha información. Asimismo, para el ajedrez 960 (una variante popular del ajedrez en la que la disposición inicial de las piezas en la primera y última fila es aleatoria), FEN presenta algunas limitaciones en cuanto a la representación del enroque. Para este caso, existen dos soluciones: Shredder-FEN y X-FEN.

5.2. PGN

PGN (Portable Game Notation), según lo establecido por Edwards [26], es la modalidad estándar más extendida para la representación de partidas de ajedrez. Siguiendo los principios del FEN, se busca una representación fácil de comprender para las personas y, al mismo tiempo, eficiente para el procesamiento por parte de los sistemas computacionales. Este estándar se centra en consolidar toda la información relevante de la partida, lo que incluye no solo los movimientos del juego, sino también datos adicionales, como la identidad de los jugadores, el lugar de la partida, entre otros detalles.

En cuanto a la representación de la partida en sí, se utiliza una lista de jugadas desde la posición inicial hasta la posición final de la partida. La representación de los movimientos se hace mediante la notación algebraica, la cual se describe en detalle en el anexo 3.

En la mayoría de los casos, la notación algebraica emplea las designaciones en inglés para las piezas. También se muestra el número de movimiento antes de cada par de jugadas. Al finalizar la partida, se indica el resultado: "1-0" para la victoria de

las blancas, "0-1" para la victoria de las negras, "1/2-1/2" para los empates y "*" si la partida aún no ha concluido. Es posible añadir comentarios después de cada jugada, encerrándolos entre "". Estos comentarios pueden contener información adicional sobre cada movimiento, como el tiempo restante de cada jugador, un recurso que se utiliza en plataformas como Lichess [27].

Previo a la lista de movimientos, se incorpora información adicional a la partida. Esta información puede ser heterogénea, aunque los programas informáticos suelen requerir la inclusión de siete elementos específicos. Estos son:

- Evento (*Event*): Designa el evento en el que se desarrolló la partida.
- Sitio (*Site*): Se refiere al lugar donde se disputó la partida, incluyendo la ciudad, región y país. En caso de que la partida haya sido disputada en una plataforma *online*, se indica correspondientemente.
- Fecha (*Date*): Fecha en que se llevó a cabo la partida.
- Resultado (*Result*): Resultado final de la partida.
- Blancas (*White*): Identidad del jugador con las piezas blancas.
- Negras (*Black*): Identidad del jugador con las piezas negras.

Además de estos campos obligatorios, se pueden incluir otros datos adicionales, como el ELO [2], el tiempo asignado para cada movimiento, entre otros. En caso de que la posición inicial no sea la estándar, se debe agregar un campo FEN indicando esta posición inicial.

[2]Sistema oficial de ranking en ajedrez

A continuación, se presenta un ejemplo de una partida utilizando este formato. Se trata de una partida histórica que marcó un hito en la relación entre los humanos y las máquinas en el ajedrez. Esta partida corresponde al sexto encuentro entre Deep Blue y el entonces campeón mundial Garry Kasparov, que concluyó con una puntuación de 3.5 a 2.5 a favor de Deep Blue.

```
[Event "IBM Man-Machine"]
[Site "New York, NY USA"]
[Date "1997.05.11"]
[Round "6"]
[Result "1-0"]
[White "Deep Blue"]
[Black "Garry Kasparov"]

1.e4 c6 2.d4 d5 3.Nc3 dxe4 4.Nxe4 Nd7 5.Ng5 Ngf6 6.Bd3
e6 7.N1f3 h6 8.Nxe6 {El movimiento sorprendió a los
espectadores, ya que no se creía posible que una
máquina pudiera realizar un sacrificio a largo plazo.}
Qe7 9.O-O fxe6 10.Bg6+ Kd8 11.Bf4 b5 12.a4 Bb7 13.Re1
Nd5 14.Bg3 Kc8 15.axb5 cxb5 16.Qd3 Bc6 17.Bf5 exf5
18.Rxe7 Bxe7 19.c4 1-0
```

Como se puede observar, la partida incluye los siete campos obligatorios y un comentario a la jugada más inesperada e importante de la partida.

5.3. UCI

La Universal Chess Interface (UCI) es un estándar que facilita dos funciones primordiales en el ámbito del ajedrez digital.

En primer lugar, permite representar las jugadas de una forma accesible tanto para los ordenadores como para las personas. En segundo lugar, UCI establece el protocolo de comunicación entre la interfaz gráfica del software de ajedrez y el módulo de procesamiento de jugadas. La adopción de este estándar posibilita que cualquier interfaz gráfica que lo implemente sea compatible con todos los módulos que también hagan uso de la UCI, reflejando la eficacia y la universalidad del sistema.

La representación de las jugadas en UCI es intuitiva y directa, constituida por dos coordenadas consecutivas que señalan el punto de origen y el destino de la pieza en movimiento. Si la jugada conlleva una coronación, se añade a la secuencia de coordenadas la letra que identifica a la nueva pieza, en minúscula.

A modo de ejemplo, si deseamos realizar la jugada Cf3 desde la posición inicial, en formato UCI se representaría como g1f3, indicando el movimiento del caballo desde la posición g1 a f3. La claridad de esta representación radica en que la jugada puede ser interpretada sin necesidad de un conocimiento previo del tablero. Si se proporciona simplemente Cf3, no podríamos determinar el punto de origen de la pieza sin analizar el tablero. La jugada nos indica que el caballo se sitúa en f3, pero podría haber partido de d2, d4, g1, etc. El formato UCI elimina esta ambigüedad y simplifica el procesamiento de la jugada al especificar directamente el punto de origen y destino.

El protocolo de comunicación de UCI, aunque su profundización está orientada principalmente a los desarrolladores, facilita una "separación de responsabilidades" entre la interfaz gráfica y el módulo de procesamiento. La interfaz gráfica es responsable de visualizar el tablero de ajedrez, la partida (generalmente en formato PGN), el libro de aperturas (que contiene información

sobre los movimientos típicos en las primeras jugadas de un partido), el tiempo restante de los jugadores, entre otros detalles. El módulo, por otro lado, se dedica a generar las jugadas en respuesta a la información recibida de la interfaz gráfica sobre la posición actual, incluyendo la ubicación de las piezas, la posibilidad de enroques, el tiempo restante, etc. Muchos de estos módulos permiten configurar el nivel de ELO que "intentarán" jugar.

El valor agregado de este protocolo reside en que, independientemente de la estructura interna del software de la interfaz gráfica y del módulo, ambos pueden comunicarse de manera eficaz siempre que se adhieran a la misma interfaz UCI. El módulo puede hacer uso de tecnologías como las redes neuronales o enfoques más tradicionales y seguirán siendo compatibles mientras se adhiera al protocolo.

En resumen, el uso de estándares como UCI en el mundo del ajedrez digital brinda una serie de ventajas, como la simplicidad y claridad en la representación de jugadas, la eficiencia en el procesamiento de las mismas y la interoperabilidad entre diferentes software de ajedrez. Además, estos estándares promueven una comunicación fluida entre la interfaz gráfica y el módulo de procesamiento, facilitando la adaptación y compatibilidad con diversas tecnologías y metodologías. En última instancia, la adopción de estos estándares contribuye a una mayor accesibilidad, comprensibilidad y disfrute del ajedrez.

Apéndice A

Diagramas de flujo

En este libro se ha optado por la utilización de diagramas de flujo para ilustrar la lógica intrínseca de los algoritmos discutidos. La elección de esta herramienta de representación visual radica en su claridad y accesibilidad, lo cual facilita enormemente la comprensión del lector, incluso en ausencia de conocimientos previos de programación.

Para iniciar nuestro recorrido, es crucial conocer los elementos básicos o bloques que componen los diagramas de flujo. Estos se clasifican principalmente en cuatro tipos:

- Comienzo/Fin
- Entrada/Salida
- Bifurcación
- Proceso

El bloque "Comienzo/Fin" marca el inicio y el cierre del flujo del programa. Un diagrama de flujo puede tener múltiples bloques de "Fin", pero solo un bloque de "Comienzo". Ambos son representados por un rectángulo de bordes redondeados. La Figura A.1 ilustra estos bloques.

Figura A.1: Representación de comienzo y fin en un diagrama de flujo

Los bloques de "Entrada/Salida" definen las entradas y salidas del algoritmo. A diferencia de los bloques de "Comienzo" y "Fin", estos pueden llevar un nombre descriptivo en lugar de un término genérico, proporcionando una visión más detallada de la funcionalidad del algoritmo. Estos bloques se representan a través de romboides, como se muestra en la Figura A.2.

Figura A.2: Representación de entradas y salidas en un diagrama de flujo

El bloque "Bifurcación" brinda la posibilidad de ejecutar distintas acciones dependiendo del cumplimiento de una condición determinada. Esto amplía la expresividad de los algoritmos y

permite incorporar lógica condicional. Se representan con un cuadrado rotado 45 grados. La Figura A.3 ilustra este bloque.

Figura A.3: Representación de bifurcaciones en un diagrama de flujo

Por último, los bloques de "Procesoïndican las acciones que se realizan en relación al algoritmo. Aunque su representación suele ser abstracta en la mayoría de los casos, estos bloques suelen incluir una descripción más detallada de la acción que se está llevando a cabo. Se representan mediante un rectángulo. La Figura A.4 muestra cómo se representa un bloque de proceso en un diagrama de flujo.

Figura A.4: Representación de procesos en un diagrama de flujo

Además de los bloques mencionados, se utilizan flechas direccionales para establecer el orden de los bloques y las conexiones entre ellos. Estas flechas generalmente no contienen información adicional, a excepción de las bifurcaciones, donde cada flecha especifica la condición que conduce al bloque de destino. Estas flechas indican la secuencia y la dirección del flujo dentro del algoritmo.

A lo largo de este libro, se presentarán ejemplos de diagramas de flujo exhaustivamente explicados y detallados, lo que te permitirá comprender de manera efectiva la lógica de los algoritmos.

Apéndice B

Ajedrez

La Real Academia Española (RAE) define al ajedrez de la siguiente manera:

> "Juego de mesa entre dos personas que se practica sobre un damero en el que se disponen las 16 piezas de cada jugador, desiguales en importancia y valor, que se desplazan y comen las del contrario según ciertas reglas."

Sin embargo, esta definición simplista no logra abarcar la complejidad que encierra el ajedrez. El ajedrez puede ser considerado como un deporte, una ciencia y un arte, todo al mismo tiempo. Se considera un deporte debido al considerable esfuerzo físico y mental que requiere. De hecho, jugar una partida de ajedrez durante cuatro horas (o incluso más) puede resultar más agotador que practicar muchos otros deportes. Además, el ajedrez se enmarca en la ciencia gracias a la aplicación del método científico en su estudio, lo cual ha permitido un amplio desarrollo y una fructífera relación con la computación. Por último, el ajedrez también puede ser considerado un arte, ya que brinda a

los ajedrecistas un gozo estético a través de patrones, posiciones y jugadas excepcionales.

Históricamente, el ajedrez ha gozado de cierta popularidad en círculos intelectuales, lo que ha contribuido a su reputación de erudición. Sin embargo, en la actualidad, ha logrado llegar a un público mucho más amplio gracias a la promoción de modalidades rápidas como *eSport* y a la gran popularidad de la serie *Gambito de Dama*. Estos factores han ampliado enormemente su base de seguidores y entusiastas.

Dicho esto, es importante establecer las reglas de este fascinante y aparentemente inagotable juego.

Reglas básicas

El ajedrez, al igual que cualquier otro juego, tiene un conjunto de reglas y un objetivo específico que los jugadores deben alcanzar. El objetivo es común para ambos contrincantes y consiste en lograr dar jaque mate al rey rival. El jaque mate se produce cuando el rey de un jugador se encuentra amenazado por una de las piezas del oponente de tal manera que no puede escapar de esta amenaza ni moverse a una casilla no amenazada. A partir de este objetivo, se pueden definir tres posibles resultados en una partida: las negras ganan si logran dar jaque mate al rey blanco, las blancas ganan si sus piezas logran dar jaque mate al rey contrario, y finalmente, se produce un empate o tablas si no es posible dar jaque mate o se cumplen algunas de las siguientes condiciones:

- Ambas partes acuerdan un empate.
- Se produce un ahogado donde uno de los bandos no puede

realizar ningún movimiento.

- Se han realizado 50 jugadas sin capturar ninguna pieza ni mover ningún peón.
- Se repite la misma posición en el tablero tres veces.

El juego se desarrolla en turnos alternando entre los jugadores, comenzando con las blancas. En cada turno, un jugador puede mover una de sus piezas, lo que puede resultar en la captura de una pieza del oponente. Dependiendo de la pieza, se disponen de diferentes movimientos posibles. La verdadera complejidad del ajedrez radica en lograr una coordinación adecuada entre las piezas, teniendo en cuenta los movimientos específicos de cada una de ellas.

Piezas

Rey

La pieza más importante en el ajedrez es el rey, ya que si este recibe jaque mate (es decir, está amenazado por una pieza enemiga y no puede realizar ninguna jugada legal para escapar de la amenaza), se pierde automáticamente la partida. Por lo general, al principio y a mitad de la partida, se debe mantener al rey en una posición segura. Sin embargo, el rey suele desempeñar un papel muy relevante en el final del juego.

En el tablero, al comienzo de la partida, el rey se encuentra ubicado en el centro de la primera fila desde la perspectiva de cada jugador.

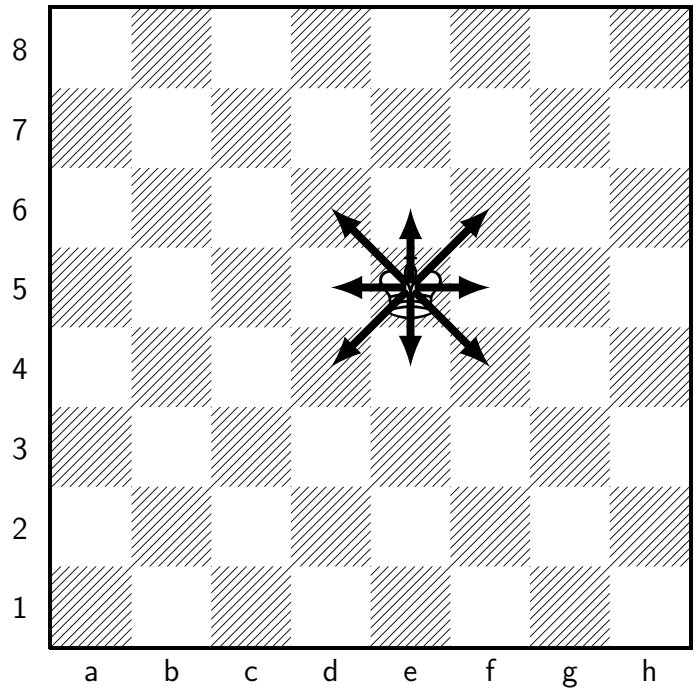


Figura B.1: Posibles movimientos del rey

Como se puede observar en el tablero, el rey puede moverse a todas las casillas adyacentes, ya sea en dirección horizontal, vertical o diagonal, pero solo puede moverse un paso a la vez. Además, el rey cuenta con un movimiento especial llamado enroque, que permite mover al rey y a una torre simultáneamente en la misma jugada, proporcionando al rey una ubicación más segura. En el Anexo 3, se muestra un ejemplo de cómo se efectúa este movimiento especial en la Figura C.6.

Dama

Después del rey, la dama es considerada la pieza más importante en el ajedrez, ya que sus movimientos combinan los de una torre y un alfil. Esta versatilidad de movimientos le otorga un gran poder en el tablero. Al inicio de la partida, la dama se encuentra ubicada en la casilla inmediata a la derecha del rey desde la perspectiva de cada jugador.

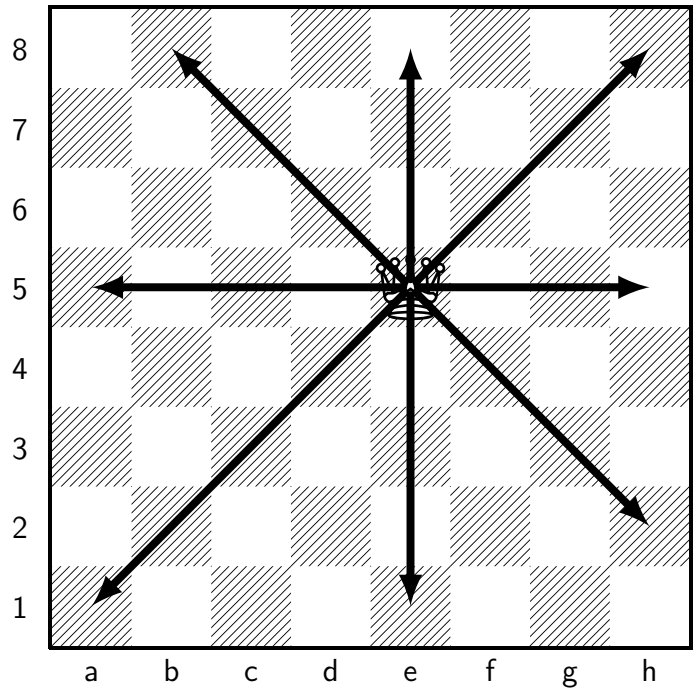


Figura B.2: Posibles movimientos de la dama

La dama puede moverse a cualquier casilla a lo largo de la fila, columna o diagonal en la que se encuentre. Como se puede apreciar en la Figura B.2, la dama puede moverse en todas

las direcciones a cualquier distancia, siempre y cuando no haya ninguna pieza bloqueando su camino. La pérdida prematura de la dama puede llevar a una derrota casi inevitable, debido a su capacidad de controlar grandes áreas del tablero y participar en numerosas combinaciones y ataques.

Torre

Cada jugador dispone de dos torres, situadas en las esquinas de su correspondiente lado del tablero. Las torres tienen la capacidad de desplazarse a cualquier casilla a lo largo de la fila y columna en la que estén ubicadas.

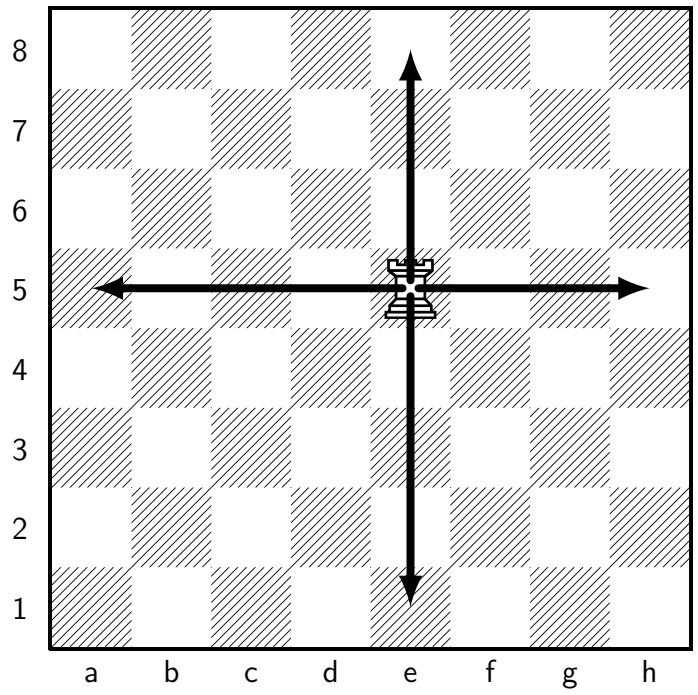


Figura B.3: Posibles movimientos de la torre

Como se puede apreciar en la Figura B.3, las torres pueden moverse a lo largo de las filas y columnas en todas las direcciones. Pueden desplazarse desde un extremo del tablero hasta el otro, lo que les confiere una gran capacidad estratégica y táctica, especialmente en la mitad y el final de la partida. Las torres son piezas valiosas en el ajedrez, ya que pueden controlar columnas y filas enteras, participar en ataques combinados y proteger al rey en posiciones defensivas. Su movilidad y versatilidad las convierten en piezas clave para la planificación de estrategias y la consecución de objetivos en el juego.

Alfil

Cada jugador dispone de dos alfiles, uno situado a la izquierda de la dama y otro a la derecha del rey.

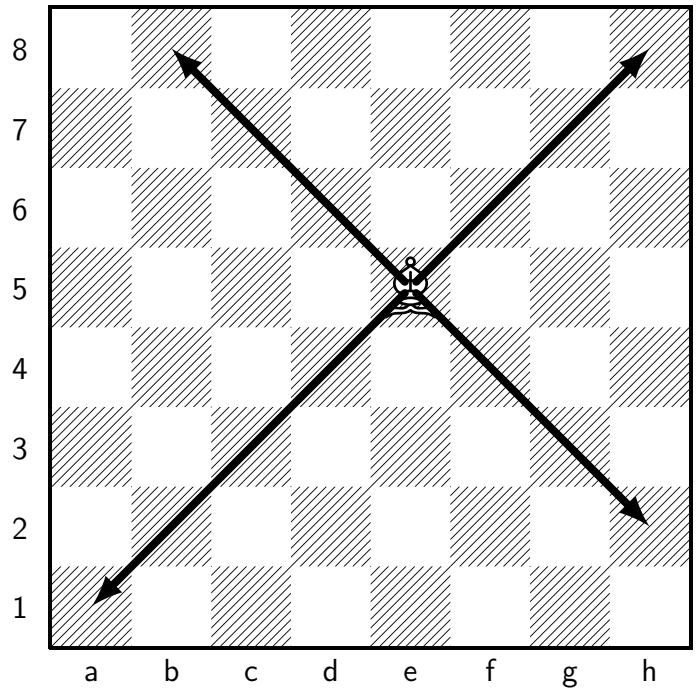


Figura B.4: Posibles movimientos del alfil

Al igual que las torres, los alfiles son piezas valiosas en el ajedrez debido a su capacidad de movimiento único a lo largo de las diagonales. Como se muestra en la Figura B.4, los alfiles pueden desplazarse en diagonal por todo el tablero. Pueden moverse desde un extremo hasta el otro en un solo movimiento, aprovechando su movimiento característico. Esta capacidad de movimiento diagonal les confiere una gran utilidad estratégica y táctica, ya que pueden controlar casillas de diferentes colores y participar en ataques combinados. Los alfiles son piezas clave en la apertura y el medio juego, donde su movilidad puede influir en el desarrollo de la partida y la ocupación de posiciones

estratégicas.

Caballo

El caballo es una de las piezas más singulares en el ajedrez, ya que tiene un movimiento en forma de L y difiere del movimiento de las demás piezas. Además, a diferencia de las otras piezas, el caballo tiene la capacidad de saltar sobre otras piezas en su trayectoria.

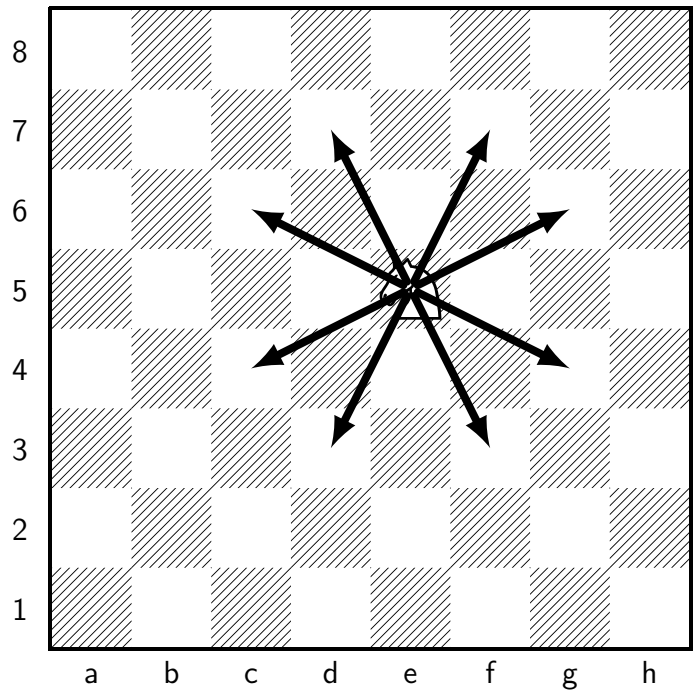


Figura B.5: Posibles movimientos del caballo

Como se muestra en la Figura B.5, el caballo puede moverse en un movimiento en L, avanzando dos casillas en una direc-

ción (horizontal o vertical) y luego girando en ángulo recto para avanzar una casilla adicional en una dirección perpendicular. Esta peculiaridad de movimiento permite al caballo saltar sobre otras piezas en su trayectoria, lo que lo convierte en una pieza impredecible y estratégicamente interesante en el juego. Cada jugador cuenta con un par de caballos, que se sitúan entre las torres y los alfiles en la configuración inicial del tablero. Los caballos son conocidos por su capacidad de maniobrar rápidamente por el tablero y pueden desempeñar un papel importante en la creación de amenazas, la defensa y la realización de combinaciones tácticas.

Peón

A pesar de ser la pieza de menor valor, el peón tiene un movimiento bastante peculiar en el ajedrez. En su primer movimiento, tiene la opción de avanzar dos casillas en lugar de una, lo cual le confiere cierta flexibilidad táctica.

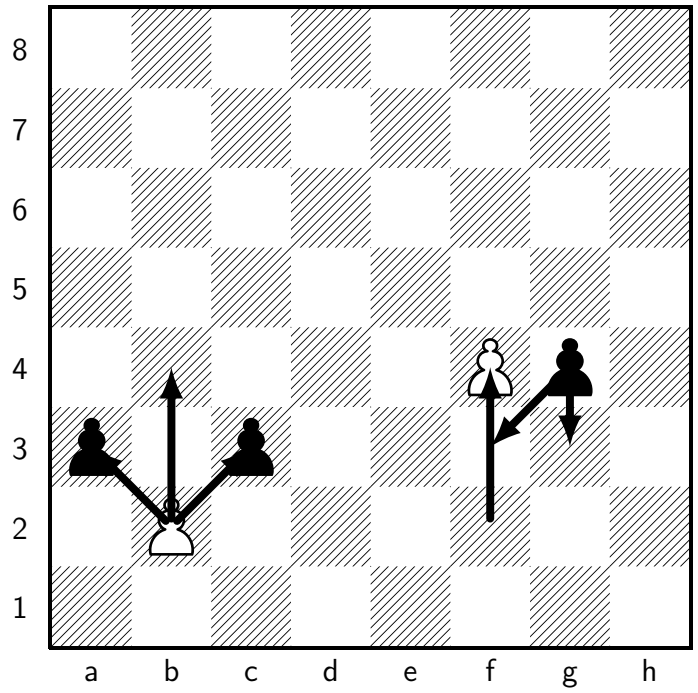


Figura B.6: Posibles movimientos del peón

El peón tiene una particularidad en su movimiento y captura. En cuanto a su movimiento regular, el peón puede avanzar una casilla hacia adelante en su columna, como se muestra en la Figura B.6. Además, cuando se encuentra en una posición de captura, el peón solo puede capturar una pieza en las dos casillas diagonales hacia el otro lado del tablero desde donde se encuentra.

El peón también tiene una jugada especial llamada "captura al paso". Esta jugada se aplica cuando un peón enemigo avanza dos casillas desde su posición inicial y se encuentra en una posición adyacente al peón rival. En ese caso, el peón puede capturar

al peón enemigo como si este solo hubiera avanzado una casilla.

Cabe destacar que el peón no puede retroceder en su movimiento. Sin embargo, cuando un peón alcanza el extremo opuesto del tablero, se produce la coronación. En este momento, el peón puede transformarse en cualquier otra pieza, a excepción del rey, lo que brinda al jugador la oportunidad de mejorar su posición estratégica en el juego.

Apéndice C

Notación en ajedrez

En el presente apéndice, profundizaremos en la notación algebraica de ajedrez, la cual es la más utilizada en la actualidad. Optaremos por su versión abreviada, donde sólo se especifica la casilla de destino de la pieza, a diferencia de la versión extendida que incluye también la casilla de origen. La lógica de esta notación es muy directa: cada movimiento se registra indicando el tipo de pieza, seguido de la designación de la columna (mediante una letra minúscula) y un número para especificar la fila donde se va a colocar esta nueva pieza. En ocasiones, no se requiere la letra que indica el tipo de pieza, ya que, en el caso de los peones, esta se omite, lo que implica que si no se especifica el tipo de pieza, se asume que es un peón.

No obstante, esta notación puede presentar ciertas complicaciones. Por ejemplo, puede ocurrir que dos piezas del mismo tipo puedan moverse a la misma casilla, y con la información proporcionada originalmente por esta notación, no sería posible discernir cuál de estas piezas es la que se mueve. Para resolver este problema, se incluye, entre la letra que indica el tipo de pieza y la letra de la columna, la fila o columna original (en casos muy excepcionales puede ser necesario especificar ambas,

la columna y la fila, para evitar cualquier ambigüedad), a fin de indicar cuál pieza específica se está moviendo.

Las columnas se designan con letras de la 'a' a la 'h' de izquierda a derecha, mientras que las filas se numeran del 1 al 8 de abajo hacia arriba.

La representación de cada pieza a través de una letra varía dependiendo del idioma. En este texto, se utilizará las designaciones en español. Al ser las designaciones inglesas las más utilizadas se muestran también sus equivalentes. A continuación, se presenta una tabla con las designaciones para cada tipo de pieza en ambos idiomas.

Pieza	Letras en español	Letras en inglés
Rey	R	K
Dama	D	Q
Torre	T	R
Alfil	A	B
Caballo	C	N

Tabla C.1: Designaciones en español e inglés para cada tipo de pieza de ajedrez

El patrón que sigue en español es realmente simple, solo se toma la primera letra del nombre de la pieza. En inglés, se sigue un patrón similar, utilizando los correspondientes nombres en inglés (*King*, *Queen*, *Rook*, *Bishop* y *Knight*, en el orden que aparecen en la tabla). Sin embargo, se observa que *King* y *Knight* comienzan con la misma letra, por lo que para el caballo se usa la segunda letra.

Finalmente, cabe destacar que cuando se realiza una captura de pieza, se añade la "x" para indicarlo, así como se añade "#" para indicar el jaque mate. Sin embargo, estos símbolos, aunque útiles para identificar rápidamente estas situaciones en

la partida, no aportan información adicional al movimiento en sí, ya que estas circunstancias pueden deducirse a partir de la posición original y del movimiento realizado. Por esta razón, a veces se opta por omitir estos indicadores.

Ejemplos

Habiendo introducido los principios teóricos, es esencial ilustrar con ejemplos para lograr una comprensión más clara de la notación algebraica.

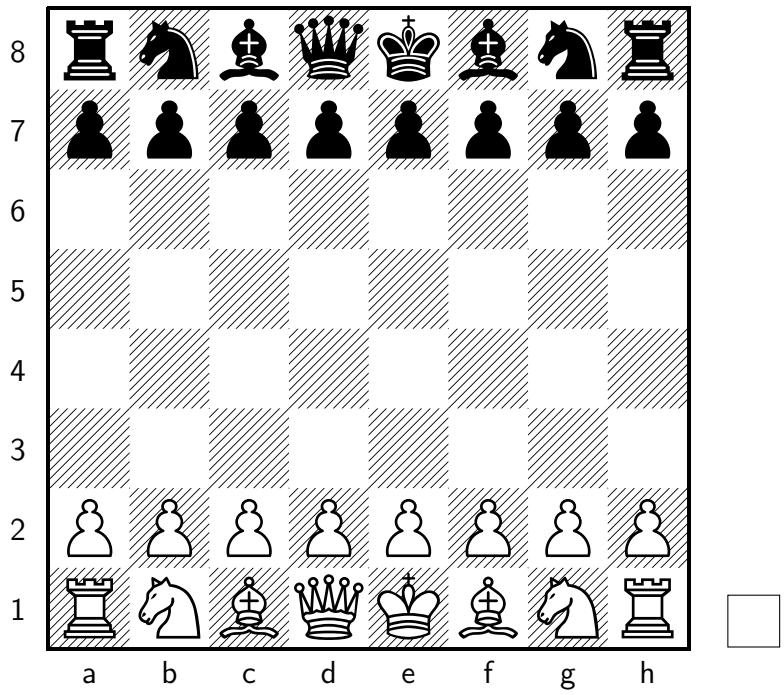


Figura C.1: Posición inicial de un juego de ajedrez

Observando la Figura C.1, se puede apreciar la disposición

de las filas y las columnas en el tablero, tal como se describió en la introducción. Las piezas blancas realizan un movimiento, resultando en la posición mostrada en la Figura C.2.

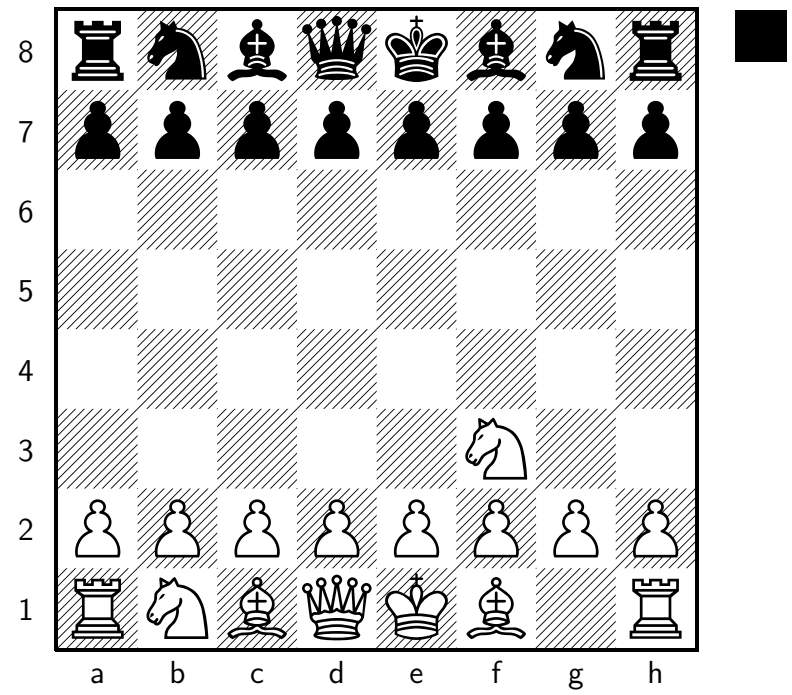


Figura C.2: Posición después de Cf3

Este movimiento se representa como Cf3 en notación algebraica, dado que se mueve el caballo desde la casilla g1 a la casilla f3. Dado que no hay otro caballo que pueda moverse a la casilla f3, no es necesario proporcionar información adicional.

A partir de esta posición, las piezas negras realizan la jugada Cf6, llevándonos a la posición mostrada en la Figura C.3.

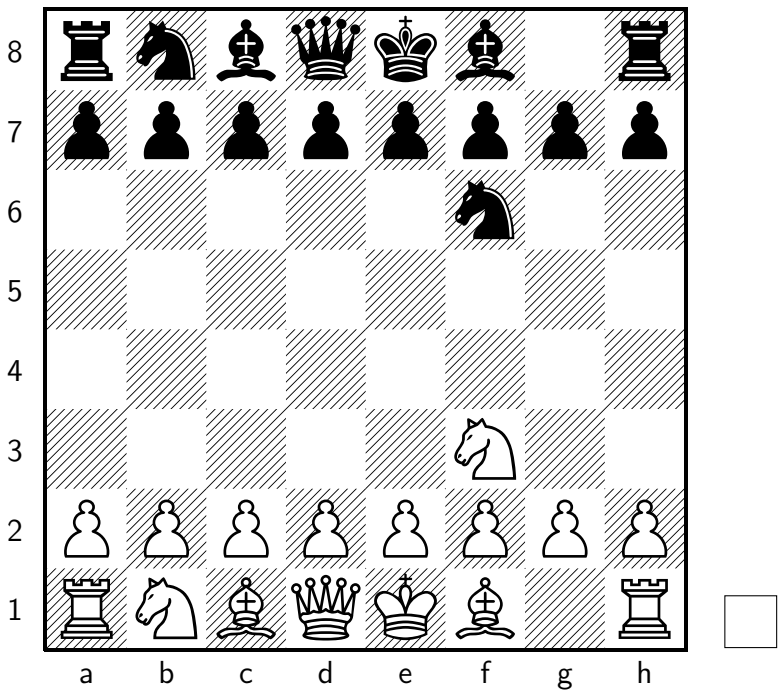


Figura C.3: Posición después de Cf6

Las piezas blancas continúan con d4 y las piezas negras responden con d5, resultando en la siguiente posición:

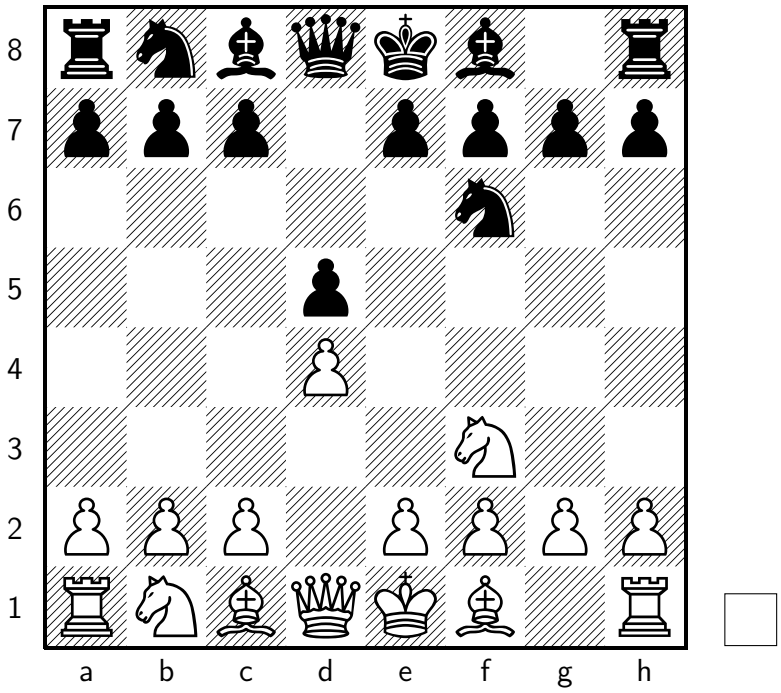


Figura C.4: Posición después de d4 y d5

Aquí surge un pequeño problema: en la posición anterior, las piezas blancas desean mover el caballo de b1 a d2, pero el caballo blanco de f3 también puede moverse a esa casilla. Por lo tanto, es necesario añadir la columna o la fila, por lo que la jugada podría representarse como Cbd2 o C1d2. Ambas opciones son válidas, aunque Cbd2 es la elección más común.

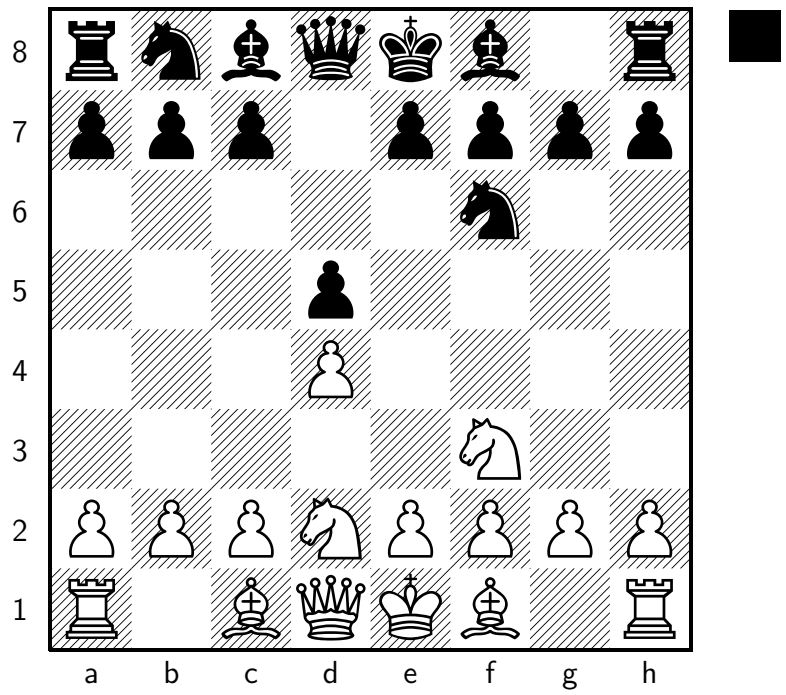
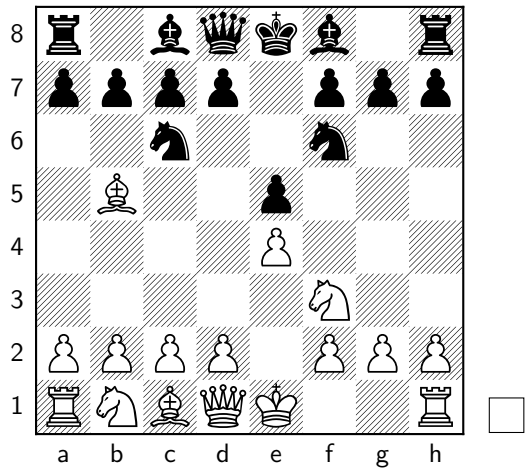


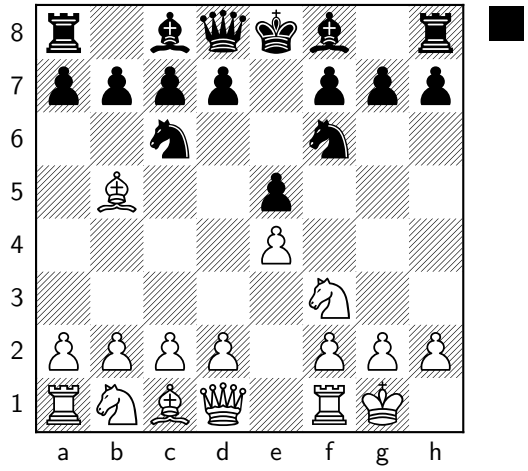
Figura C.5: Posición después de Cbd2

Además de las jugadas comúnmente vistas, existen algunas jugadas especiales que merecen ser mencionadas brevemente. La primera de ellas es la denominada promoción, que ocurre cuando un peón alcanza la última fila opuesta y se le ofrece la oportunidad de transformarse en otra pieza. Las opciones de transformación incluyen alfil, caballo, torre o dama. Dado que la dama es la pieza más poderosa, es habitual que el peón se transforme en esta pieza, pero dependiendo de las circunstancias, podría transformarse en otro tipo de pieza. Para señalar esta jugada, se añade una letra minúscula al final de la jugada original, indicando en qué pieza se ha transformado.

Finalmente, queda la jugada especial conocida como enroque, que permite mover simultáneamente al rey y a una torre. Existen dos tipos de enroque: el enroque corto, donde el rey se traslada a la columna g y la torre a la columna f (utilizando la torre más cercana al rey), y se marca en la notación con "0-0". El enroque largo, por otro lado, implica usar la torre más alejada del rey, y el rey se traslada a la columna c y la torre a la columna d. Se marca con "0-0-0". Para efectuar el enroque, ni el rey ni la torre con la que se realiza deben haberse movido previamente. A continuación se muestra un ejemplo de cómo se realiza el enroque en la Figura C.6. Como dato curioso, esta posición pertenece a la apertura española, también conocida como la apertura Ruy López. Ruy López es considerado el primer campeón mundial de ajedrez y es oriundo de Extremadura, España.



(a) Posición antes del enroque



(b) Posición después del enroque

Figura C.6: Ejemplo de cómo efectuar el enroque

En el ejemplo anterior, las piezas blancas han realizado el enroque corto, lo que se denotaría con la jugada "0-0". El enroque es una jugada muy común, ya que permite poner al rey a salvo y al mismo tiempo permite que la torre con la que se realiza el

enroque entre en juego rápidamente.

Además, en partidas grabadas o transmitidas, a menudo se registra cada jugada junto con el número de jugada, por lo que una secuencia de movimientos podría verse así:

1. e4 e5
2. Nf3 Nc6
3. Bb5 a6
4. Ba4 Nf6
5. O-O Be7

Esto representa cinco movimientos (o diez jugadas) de la apertura española o apertura Ruy López. Como puedes ver, los movimientos de las piezas blancas y negras se registran juntos, con el movimiento de las piezas blancas primero.

Apéndice D

Recomendaciones

Durante la elaboración de este libro, fue imprescindible acudir a una diversidad de fuentes, de distintas naturalezas y niveles de tecnicidad. Este proceso resultó en una valiosa recopilación de referencias que pueden utilizarse para profundizar en los temas tratados en cada capítulo del libro. Por lo tanto, este anexo funciona como un recurso de apoyo para aquellos lectores que deseen ampliar su conocimiento en alguna área específica. Las referencias son variadas, abarcando desde libros de divulgación científica, hasta artículos académicos. Para cada sugerencia, se proporcionará una breve descripción que permita al lector decidir su elección. Lamentablemente, la mayoría de las recomendaciones están disponibles únicamente en inglés.

Las referencias están organizadas de acuerdo a su relevancia para los diferentes capítulos del libro.

Capítulo 1: Algoritmos

- *Algorithms Illuminated*: Esta serie de cuatro libros permite al lector pasar de un nivel básico a una comprensión avanzada de los algoritmos.

- *Introduction to Algorithms*: Este clásico en el campo de los algoritmos es frecuentemente utilizado como texto de referencia en numerosos cursos de algoritmia.
- *The Outer Limits of Reason*: Este libro ofrece una reflexión sobre los límites de la ciencia, las matemáticas y la computación.
- *Algorithms to Live By*: Esta obra presenta una interesante propuesta sobre cómo aplicar los algoritmos en nuestras actividades cotidianas.
- *Competitive Programming 3*: Este libro avanzado se enfoca en los algoritmos utilizados en la programación de competencia.

Capítulo 2: Juegos de estrategia y algoritmos

- *Minimax Algorithm in Game Theory — Set 4 (Alpha-Beta Pruning)*: Este artículo proporciona una explicación detallada del algoritmo de poda alfa-beta.
- *Programming a Computer for Playing Chess*: Este es un artículo académico antiguo que propone cómo se podría programar a una computadora para jugar al ajedrez.
- Chess Programming Wiki: Esta wiki ofrece un amplio contenido sobre los conceptos de programación relacionados con el ajedrez. Aunque es un recurso avanzado, es ideal para profundizar en los temas abordados en este libro.

Capítulo 3: Inteligencia artificial y juegos de estrategia

- *Neural Network for Chess*: Este libro ofrece una visión práctica de las redes neuronales aplicadas al ajedrez, utilizando Python como lenguaje de programación. Esta obra es gratuita y puede encontrarse fácilmente en Internet.
- *Foundations of Deep Reinforcement Learning*: Esta obra aborda en profundidad el aprendizaje por refuerzo profundo y cómo implementarlo con Python.
- *How Smart Machines Think*: Este libro presenta los métodos empleados por las inteligencias artificiales contemporáneas. El capítulo 15, dedicado a AlphaGo Zero, está íntimamente relacionado con este libro.
- *Reinforcement Learning*: Este libro, considerado una biblia en el campo del aprendizaje por refuerzo, aborda la mayoría de los algoritmos básicos que constituyen la base de los más complejos.
- *AI and Machine Learning for Coders*: Este es un libro práctico orientado a programadores interesados en inteligencia artificial y aprendizaje automático.
- *AlphaGo - The Movie*: Este documental sobre AlphaGo está estrechamente relacionado con AlphaZero y destaca cómo revolucionó el mundo del juego de Go.
- *Deep Learning with Python*: Este libro ofrece una excelente introducción al aprendizaje profundo, utilizando Python como lenguaje de programación.

- *Mastering the game of go without human knowledge*: Este es el artículo académico que presentó al mundo a AlphaGo Zero, cuya arquitectura es casi idéntica a la de AlphaZero.
- *Mastering chess and shogi by self-play with a general reinforcement learning algorithm*: Este artículo académico constituye la base para la sección de AlphaZero en este libro. Es altamente recomendable para quienes deseen conocer más sobre AlphaZero.
- Leela Chess Zero: Esta página web ofrece amplia información sobre el proyecto Leela Chess Zero.

Capítulo 4: Motores de ajedrez

- Página oficial de Stockfish: Esta web contiene toda la información relacionada con Stockfish, incluyendo cómo descargarlo, documentación relevante, el código fuente, etc.
- Página oficial de Maia: Este sitio web proporciona toda la información pertinente sobre Maia, desde cómo jugar contra ella en Lichess, hasta detalles sobre sus resultados y logros.

Capítulo 5: Notación y estándares en ajedrez

- *La notación FEN en ajedrez, ¿la conoces?*: Este artículo trata sobre la notación FEN. (Consultado el 8 de septiembre de 2022)
- *Standard: Portable Game Notation Specification and Implementation Guide*: Este documento especifica el estándar

y la implementación de la notación PGN. (Consultado el 8 de septiembre de 2022)

Bibliografía

[1] Steven Halim and Felix Halim. *Competitive Programming 3*, chapter 8, pages 299–302. Lulu, third edition, 2013.

[2] Noson S. Yanofsky. *The Outer Limits of Reason*, chapter 6, pages 135–161. MIT Press, 2016.

[3] Steven Halim and Felix Halim. *Competitive Programming 3*, chapter 3, pages 110–111. Lulu, third edition, 2013.

[4] Devin Monnens. I commenced an examination of a game called tit-tat-to: Charles babbage and the first computer game. In *DiGRA Conference*, 2013.

[5] J v. Neumann. Zur theorie der gesellschaftsspiele. *Mathematische annalen*, 100(1):295–320, 1928.

[6] Emile Borel. La théorie du jeu et les équations intégrale-sa noyau symétrique. *Comptes rendus de l'Académie des Sciences*, 173(1304-1308):58, 1921.

[7] Claude E. Shannon. Programming a computer for playing chess. *Philosophical Magazine*, 41(314), March 1950.

[8] Board Representation - Chessprogramming wiki — chessprogramming.org. `https://www.chessprogramming.org/Board_Representation`.

[9] Laura Graesser and Wah Loon Keng. *Foundations of deep reinforcement learning: theory and practice in Python*, chapter 2. Addison-Wesley Professional, 2019.

[10] Laura Graesser and Wah Loon Keng. *Foundations of deep reinforcement learning: theory and practice in Python*, chapter 4. Addison-Wesley Professional, 2019.

[11] Laura Graesser and Wah Loon Keng. *Foundations of deep reinforcement learning: theory and practice in Python*, chapter 6. Addison-Wesley Professional, 2019.

[12] Sean Gerrish. *How smart machines think*, chapter 15, pages 241–243. MIT Press, 2018.

[13] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *arXiv preprint arXiv:1712.01815*, 2017.

[14] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International conference on machine learning*, pages 448–456. pmlr, 2015.

[15] Desarrolladores de Leela Chess Zero. Leela chess zero network topology. `https://lczero.org/dev/backend/nn/`.

[16] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert,

Lucas Baker, Matthew Lai, Adrian Bolton, et al. Mastering the game of go without human knowledge. *nature*, 550(7676):354–359, 2017.

[17] David Foster. Alphago zero explained in one diagram. `https://medium.com/applied-data-science/alphago-zero-explained-in-one-diagram-365f5abf67e0`.

[18] Kiprono Elijah Koech. Cross-entropy loss function. `https://towardsdatascience.com/cross-entropy-loss-function-f38c4ec8643e`.

[19] George Seif. Understanding the 3 most common loss functions for machine learning regression. `https://towardsdatascience.com/understanding-the-3-most-common-loss-functions-for-machine-learning-regression-23e0ef3e14d3`.

[20] Anuja Nagpal. L1 and l2 regularization methods. `https://towardsdatascience.com/l1-and-l2-regularization-methods-ce25e7fc831c`.

[21] Yu Nasu. Efficiently updatable neural-network-based evaluation functions for computer shogi. *The 28th World Computer Shogi Championship Appeal Document*, 185, 2018.

[22] Dominik Klein. Neural networks for chess. *arXiv preprint arXiv:2209.01506*, 2022.

[23] Reid McIlroy-Young, Siddhartha Sen, Jon Kleinberg, and Ashton Anderson. Aligning superhuman ai with human behavior: Chess as a model system. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 1677–1687, 2020.

[24] Reid McIlroy-Young, Russell Wang, Siddhartha Sen, Jon Kleinberg, and Ashton Anderson. Learning models of individual behavior in chess. In *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pages 1253–1263, 2022.

[25] Steffen Künn, Christian Seel, and Dainis Zegners. Cognitive performance in the home office - evidence from professional chess. *IZA Discussion Paper*, 13491, 2020.

[26] Steven J Edwards. Portable game notation specification and implementation guide. *Retrieved April*, 4:2011, 1994.

[27] lichess.org. Lichess. `https://lichess.org`.