ÁRBOL DE JUEGOS: DEL ALGORITMO MINIMAX CON PODA α-β AL ALGORITMO MONTECARLO TREE SEARCH. FUERZA BRUTA VS ALEATORIEDAD

Jacinto Rafael Quevedo Sarmiento[1]
Eduardo Gregorio Quevedo Gutiérrez[2]

[1]Consejería de Educación, Universidades, Cultura y Deportes
Gobierno de Canarias

[2]Universidad de Las Palmas de Gran Canaria

Realizar pruebas al azar puede ser un buen camino para llegar a un resultado difícil de conseguir mediante fórmulas o algoritmos (Stanislaw Ulam)

Resumen

En este artículo, para juegos como el 3-en-raya (Tic-Tac-Toe), Conecta4, L, Damas, Othello, Ajedrez y Go entre otros, se discutirán dos famosos algoritmos de búsqueda en árboles llamados Minimax y Montecarlo Tree Search (abreviado como MCTS). Se tratará de descubrir la intuición detrás de los algoritmos de búsqueda en árboles. Después de eso, veremos cómo Minimax (con poda α-β) y MCTS se pueden usar en implementaciones de juegos clásicos y modernos para construir IA de juegos sofisticados. En dichos algoritmos se manifiestan dos formas de pensamiento computacional: en el Minimax la fuerza bruta con adecuadas funciones de evaluación, en el MCTS la aleatoriedad. El Ajedrez, según Herbert Simon, la Drosophila de la IA, ha sido superado por la máquina, caso de DeepBlue con sus entrañas de Minimax α-β, caso de AlphaZero con MCTS en sus venas.

Abstract

In this article, for games like 3-in-a-row (Tic-Tac-Toe), Conecta4, L, Checkers, Othello, Chess and Go among others, two famous tree search algorithms called Minimax and Montecarlo Tree Search (abbreviated MCTS) will be discussed. The intuition behind the tree search algorithms will be commented. After that, we will

FORMACIÓN DEL PROFESORADO E INVESTIGACIÓN EN EDUCACIÓN MATEMÁTICA XIII
Quevedo, J.R.; Quevedo, E.G. (2021). *Árbol de juegos: Del algoritmo Minimax con poda α-β al algoritmo Montecarlo Tree Search. Fuerza bruta vs. Aleatoriedad. pp. 169-203*

see how Minimax (with α-β pruning) and MCTS can be used in both classic and modern game implementations to build sophisticated game AI. In these algorithms, two forms of computational thinking are manifested: in the Minimax, brute force with adequate evaluation functions, in the MCTS, randomness. Chess, according to Herbert Simon, the Drosophila of AI, has been overtaken by the machine, the case of DeepBlue with its guts of Minimax α-β, the case of AlphaZero with MCTS in its veins.

Introducción

Muchos recuerdan que en el Museo Elder de la Ciencia y la Tecnología había un robot, llamado Eldi (ver Figura 1), que jugaba con el público a los juegos: "3-en-raya (Tic-Tac-Toe)", "Búsqueda del Tesoro" (Treasure Search) y al "Juego de la L" y realizaba más de 25 partidas diarias de cada juego…. y ¡las ganaba o empataba casi todas! Cuando descansaba se enchufaba el programa Fritz 6.0 y era invencible en Ajedrez, tanto en partidas rápidas como lentas. ¿Era Eldi inteligente? ¿Tenía Eldi intuición? ¿Se habían hecho realidad las previsiones de Alan Turing sobre el "pensamiento" de las máquinas?

Es posible que, mucho de ese público, se preguntara si existía, para cada uno de esos juegos, una determinada estrategia que pudiera aprovechar y que le permitiera ganar todas las veces (o al menos forzar un empate). ¿Existe tal algoritmo que te muestre cómo puedes derrotar a tu oponente en un momento dado? Resulta que lo hay. Para ser precisos, hay un par de algoritmos que se pueden utilizar para predecir los mejores movimientos posibles en juegos como el Conecta4, Damas, Othello, Ajedrez y Go, y además en los juegos que Eldi practicaba. Una de estas familias de algoritmos aprovecha la búsqueda en árboles y opera en árboles de estado del juego.

En este artículo discutiremos dos famosos algoritmos de búsqueda en árboles llamados Minimax y Montecarlo Tree Search (abreviado como MCTS). Comenzaremos nuestro viaje hacia los algoritmos de búsqueda en árboles descubriendo la intuición detrás de su funcionamiento interno. Después de eso, veremos cómo Minimax y MCTS se pueden usar en implementaciones de juegos modernos para construir IA de juegos sofisticados. También arrojaremos algo de luz sobre los desafíos computacionales que enfrentaremos y cómo manejarlos a través de técnicas de optimización del rendimiento.

Figura 1. Robot Eldi en acción en el Museo Elder de la Ciencia y la Tecnología

La intuición detrás de la búsqueda de árboles

Imaginemos que estás jugando a 3-en-raya con el robot Eldi. Mientras juegas, te preguntas sobre cuál podría ser la estrategia óptima. ¿Cuál es el mejor movimiento que debes elegir en una situación determinada? ¿Cómo lo hace Eldi? En términos generales, hay dos modos en los que puede operar al determinar el próximo movimiento que se desea jugar:

- Agresivo:
  - Realice un movimiento que provoque una victoria inmediata (si es posible)
  - Realice un movimiento que establezca una situación ganadora en el futuro.
- Defensivo:
  - Realice un movimiento que evite que su oponente gane en la siguiente ronda (si es posible)
  - Realice un movimiento que evite que su oponente establezca una situación ganadora futura en la próxima ronda.

Estos modos y sus respectivas acciones son básicamente las únicas estrategias que debes seguir para ganar el juego de 3-en-raya (Tic-Tac-Toe). Lo "único" que tienes que hacer es mirar el estado actual del juego en el que te encuentras y jugar simulaciones a través de todos los próximos movimientos potenciales que podrían jugarse. Lo haces fingiendo que has jugado un movimiento determinado y luego continúas jugando hasta el final, alternando entre el jugador **X** y **O.** Mientras haces eso, estás construyendo un árbol de juego de todos los movimientos posibles que tú y tu oponente jugarían. La Figura 2 muestra una versión simplificada de dicho árbol de juegos (sólo se usarán ejemplos de árboles de juegos simplificados para ahorrar espacio).

Por supuesto, el conjunto de reglas estratégicas que hemos discutido está diseñado específicamente para el juego de 3-en-raya. Sin embargo, podemos generalizar este enfoque para que funcione con otros juegos de mesa como Ajedrez o Go. Echemos un vistazo a Minimax, un algoritmo de búsqueda de árbol que

abstrae nuestra estrategia 3-en-raya (Tic-Tac-Toe) para que podamos aplicarla a otros juegos de mesa de dos jugadores.

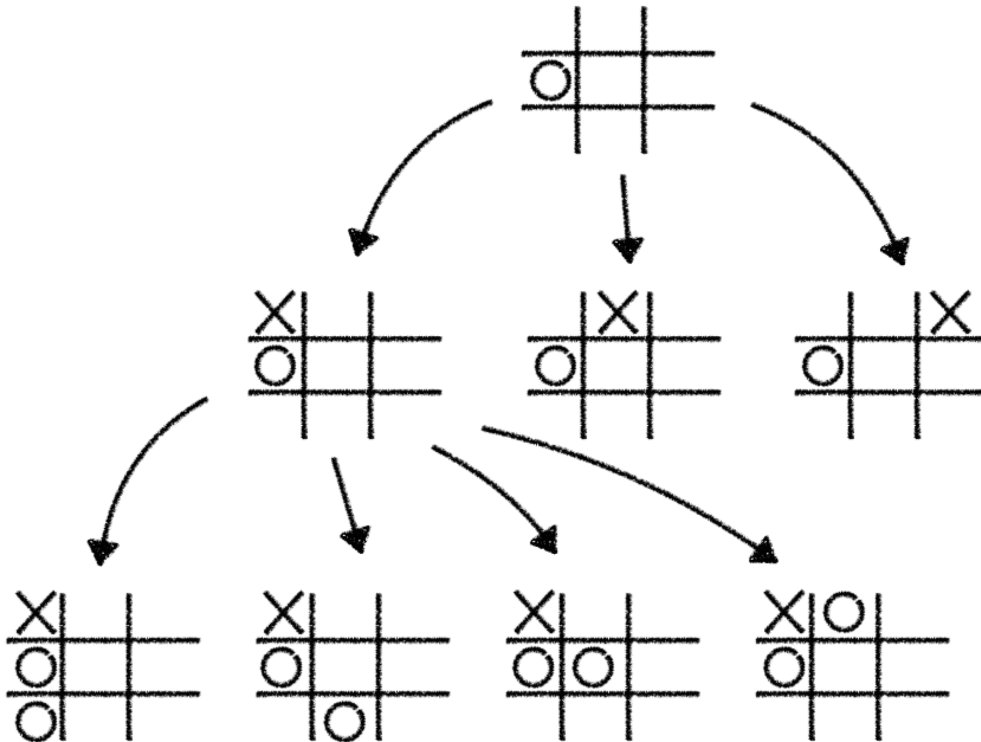

Figura 2.- Versión simplificada del árbol de juegos

El algoritmo Minimax

Dado que hemos desarrollado una intuición para los algoritmos de búsqueda de árboles, cambiemos nuestro enfoque de juegos simples como 3-en-raya a juegos más complejos como el Ajedrez.

Antes de sumergirnos, recapitulemos brevemente las propiedades de un juego de Ajedrez. El ajedrez es un juego determinista de dos jugadores de información perfecta. ¿Suena confuso? Vamos a aclararlo:

En Ajedrez, dos jugadores (Blanco y Negro) juegan entre sí. Se asegura que cada movimiento que se realiza se "cumplirá" sin aleatoriedad (el juego no utiliza ningún elemento aleatorio, como un dado, ruleta, …). Durante el juego, todos los jugadores pueden observar todo el estado del juego. No hay información oculta, por lo que todos tienen información perfecta sobre todo el juego en un momento dado.

Gracias a esas propiedades, siempre podemos calcular qué jugador está actualmente por delante y cuál está detrás. Hay varias formas diferentes de hacer esto para el juego de Ajedrez. Un método para evaluar el estado actual del juego es sumar todas las piezas blancas restantes en el tablero y restar todas las negras restantes. Hacer esto producirá un valor único donde un valor grande favorece al blanco y un valor pequeño favorece al negro. Este tipo de función se denomina **función de evaluación**.

Basándonos en esta función de evaluación, ahora podemos definir el objetivo general durante el juego para cada jugador individualmente. Las blancas intentan maximizar este objetivo mientras que las negras intentan minimizarlo.

Supongamos que estamos inmersos en un juego de ajedrez en curso. Somos jugadores blancos y ya hemos jugado un par de movimientos inteligentes, lo que resultó en un gran número calculado por nuestra función de evaluación. Ahora es nuestro turno, pero estamos estancados. ¿Cuál de los posibles movimientos es el mejor que podemos realizar?

Resolveremos este problema con el mismo enfoque que ya encontramos en nuestro ejemplo de juego 3-en-raya (Tic-Tac-Toe). Creamos un árbol de movimientos potenciales que podrían realizarse en función del estado del juego en el que nos encontramos. Para simplificar las cosas, pretendemos que solo hay dos movimientos posibles que podemos jugar (en el ajedrez hay en promedio ~ 30 opciones diferentes para cada estado de juego dado). Comenzamos con un nodo raíz (blanco) que representa el estado actual. A partir de ahí, estamos ramificando dos nodos secundarios (negros) que representan el estado del juego en el que nos encontramos después de realizar uno de los dos movimientos posibles. A partir de estos dos nodos secundarios, volvemos a ramificar dos nodos secundarios (blancos) separados. Cada uno de ellos representa el estado del juego en el que nos encontramos después de realizar uno de los dos movimientos posibles, que podríamos jugar desde el nodo negro. El árbol resultante se corresponde al diagrama presentado en la Figura 3. Así mismo, el cálculo del resultado del juego para cada estado final con nuestra función de evaluación se presenta en la Figura 4.

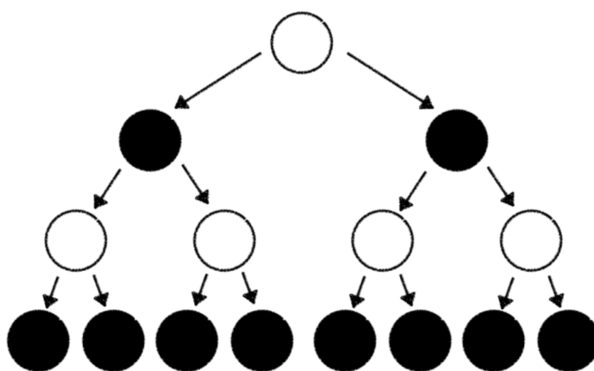

Figura 3.- Árbol considerado en el ejemplo expuesto

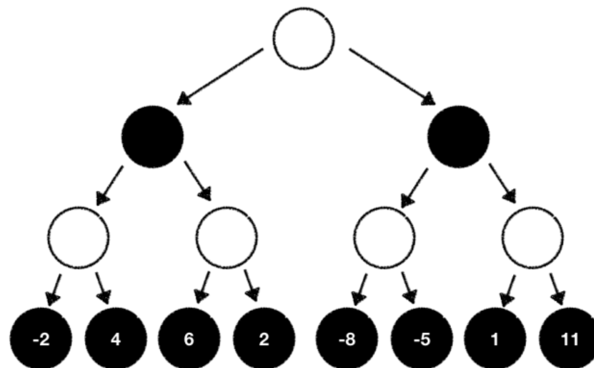

Figura 4.- Cálculo de resultado del juego para cada estado final con la función de evaluación

Dado que estamos al final del árbol, ahora con esta información, sabemos el resultado del juego que podemos esperar cuando tomamos todos los movimientos descritos comenzando desde el nodo raíz y terminando en el último nodo donde calculamos la evaluación del juego. Puesto que somos jugadores blancos, parece que la mejor jugada a elegir es la que nos preparará para terminar eventualmente en el estado de juego con el resultado más alto calculado por nuestra función de evaluación.

Si bien esto es cierto, hay un problema. Todavía está involucrado el jugador negro y no podemos manipular directamente qué movimiento elegirá. Si no podemos manipular esto, ¿por qué no estimamos lo que probablemente hará el jugador negro en función de nuestra función de evaluación? Como jugador blanco siempre intentamos maximizar nuestro resultado. El jugador negro siempre intenta minimizar el resultado. Con este conocimiento, ahora podemos retroceder a través de nuestro árbol de juego y calcular los valores para todos nuestros nodos de árbol individuales paso a paso. Las blancas intentan maximizar, como se presenta en la

Figura 5, mientras que las negras quieren minimizarlo, como se muestra en la Figura 6. Una vez hecho esto, podemos elegir el siguiente movimiento en función de los valores de evaluación que acabamos de calcular. En nuestro caso, se elige el movimiento presentado en la Figura 7, a fin de maximizar el resultado.

Lo que acabamos de aprender es el procedimiento general del llamado algoritmo Minimax. El algoritmo Minimax obtuvo su nombre del hecho de que un jugador quiere **Mini**-mizar el resultado, mientras que el otro trata de **Max**-imizarlo. El código se presenta en la Figura 8.

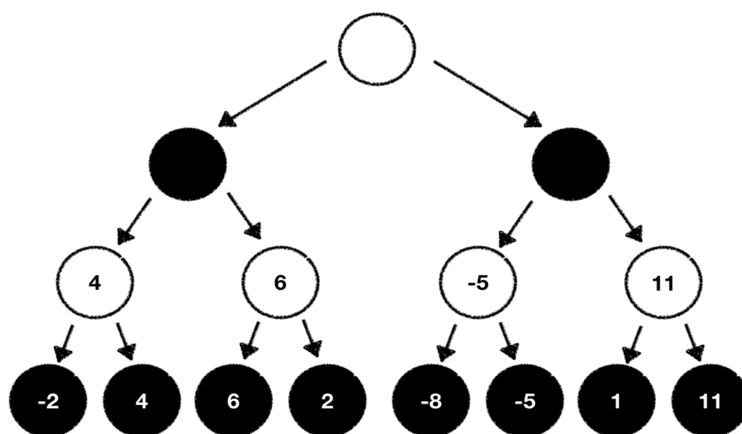

Figura 5.- Maximización del resultado

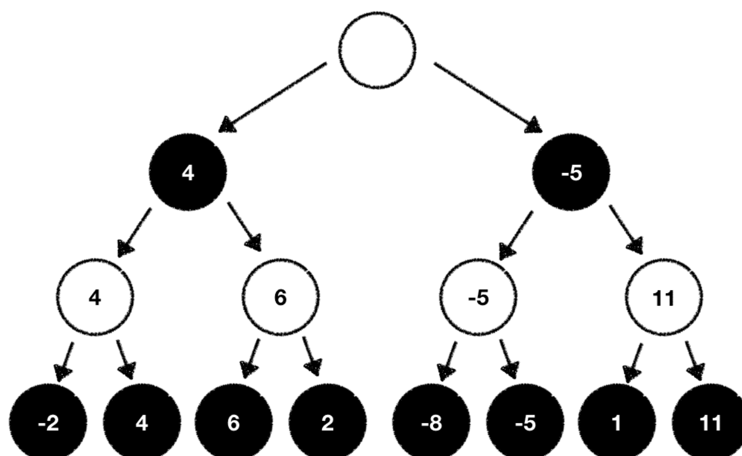

Figura 6.- Minimización del resultado

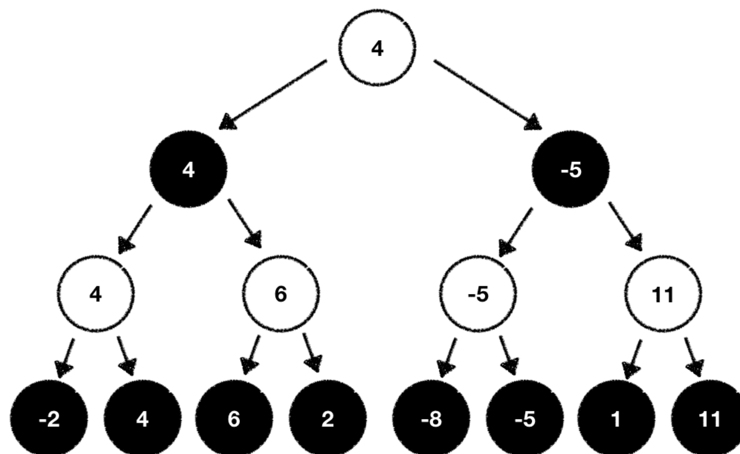

Figura 7.- Elección del movimiento en función de los valores de la evaluación

```
def minimax(state, max_depth, is_player_minimizer):
    if max_depth == 0 or state.is_end_state():
        # Estamos al final. Es hora de evaluar el estado
en el que nos encontramos
        return evaluation_function(state)
    # ¿Es el jugador actual el minimizador?
    if is_player_minimizer:
        value = -math.inf
        for move in state.possible_moves():
            evaluation = minimax(move, max_depth - 1,
False)
            min = min(value, evaluation)
        return value
```

```
    # ¿o el maximizador?
    value = math.inf
    for move in state.possible_moves():
        evaluation = minimax(move, max_depth - 1, True)
        max = max(value, evaluation)
    return value
```

Figura 8.- Código del algoritmo Minimax

Búsqueda de reducción de espacio con poda α-β

Minimax es un algoritmo de búsqueda de árboles simple y elegante. Con suficientes recursos informáticos, siempre encontrará el próximo movimiento óptimo para jugar.

Pero hay un problema. Si bien este algoritmo funciona perfectamente con juegos simplistas como el 3-en-raya (Tic-Tac-Toe), computacionalmente no es factible implementarlo para juegos estratégicamente más complicados como el ajedrez. La razón de esto es el llamado factor de ramificación de árboles. Ya hemos tocado brevemente ese concepto antes, pero echémosle un segundo vistazo.

En nuestro ejemplo anterior, hemos restringido artificialmente los movimientos potenciales que uno puede jugar a dos para mantener la representación del árbol simple y fácil de razonar. Sin embargo, la realidad es que normalmente hay más

de dos posibles movimientos siguientes. En promedio, hay aproximadamente 30 movimientos que un jugador de Ajedrez puede jugar en cualquier estado de juego dado. Esto significa que cada nodo del árbol tendrá aproximadamente 30 hijos diferentes. Esto se llama ancho del árbol. Denotamos el ancho de los árboles como w. Pero hay más. Se necesitan aproximadamente 85 turnos consecutivos para terminar una partida de Ajedrez. Traducir esto a nuestro árbol significa que tendrá una profundidad promedio de 85. Denotamos la profundidad de los árboles como d. Dado w y d podemos definir la fórmula w^d, que nos mostrará cuántas posiciones diferentes tenemos que evaluar en promedio.

Conectando los números de Ajedrez obtenemos 30^{85}. Tomando el juego de mesa Go como ejemplo que tiene un ancho w de aproximadamente 250 y una profundidad media d de 150 obtenemos 250^{150}. Les animo a que escriban esos números en su calculadora y presione enter. No hace falta decir que las computadoras de la generación actual e incluso los sistemas distribuidos a gran escala tardarán "una eternidad" en procesar todos esos cálculos.

¿Significa esto que Minimax solo se puede usar para juegos como 3-en-raya (Tic-Tac_Toe)? Absolutamente no. Podemos aplicar algunos trucos ingeniosos para optimizar la estructura de nuestro árbol de búsqueda.

En términos generales, podemos reducir el ancho y la profundidad de los árboles de búsqueda podando nodos individuales y ramas de ellos. Veamos cómo funciona esto en la práctica.

Poda alfa-beta

Recuerde que Minimax se basa en la premisa de que un jugador intenta maximizar el resultado del juego basándose en la función de evaluación, mientras que el otro

intenta minimizarlo. Este comportamiento de juego se traduce directamente en nuestro árbol de búsqueda. Durante el recorrido desde la parte inferior hasta el nodo raíz, siempre elegimos el "mejor" movimiento respectivo para cualquier jugador. En nuestro caso, el jugador blanco siempre escogió el valor máximo mientras que el jugador negro escogió el valor mínimo. Observando el árbol de la Figura 9 se puede explotar este comportamiento para optimizarlo. Así es como:

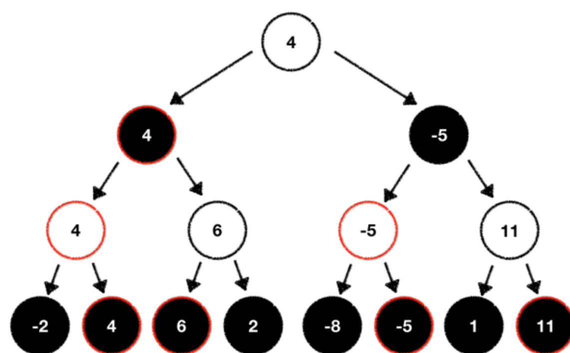

Figura 9.- Optimización de árbol

Mientras caminamos a través de los movimientos potenciales que podemos jugar dado el estado actual del juego en el que nos encontramos, debemos construir nuestro árbol en profundidad. Esto significa que debemos comenzar en un nodo y expandirlo jugando el juego hasta el final antes de retroceder y elegir el siguiente nodo que queremos explorar, tal y como se presenta en la Figura 10.

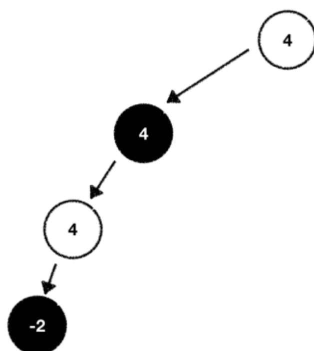

Figura 10.- Exploración de nodos en árbol

Seguir este procedimiento nos permite identificar movimientos que nunca se jugarán antes. Después de todo, un jugador maximiza el resultado mientras que el otro lo minimiza. La parte del árbol de búsqueda en la que un jugador terminaría en una situación peor según la función de evaluación se puede eliminar por completo de la lista de nodos que queremos expandir y explorar. Podamos esos nodos de nuestro árbol de búsqueda y, por lo tanto, reducimos su ancho, tal y como se presenta en la Figura 11. Cuanto mayor sea el factor de ramificación del árbol, mayor será la cantidad de cálculos que potencialmente podemos ahorrar.

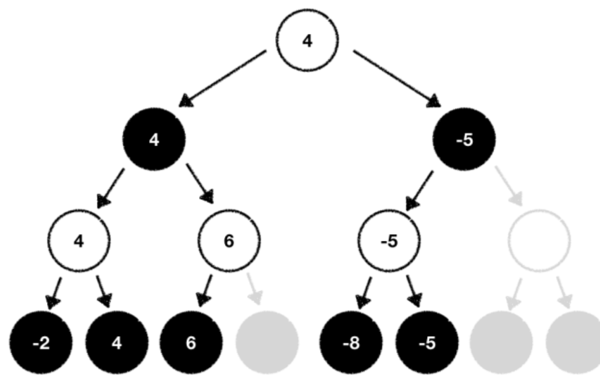

Figura 11.- Mecanismo de poda en árbol

Suponiendo que podamos reducir el ancho en un promedio de 10, terminaríamos con $w^d = (30 - 10)^{85} = 20^{85}$ cálculos que tenemos que realizar. Eso ya es una gran victoria.

Esta técnica de podar partes del árbol de búsqueda que nunca se considerarán durante el juego se llama poda alfa-beta. La poda alfa-beta debe su nombre a los parámetros α y β que se utilizan para realizar un seguimiento de la mejor

puntuación que cualquiera de los jugadores puede lograr mientras camina por el árbol. El código del algoritmo se presenta en la Figura 12.

```
def minimax(state, max_depth, is_player_minimizer, alpha,
beta):
    if max_depth == 0 or state.is_end_state():
        return evaluation_function(state)


    if is_player_minimizer:
        value = -math.inf
        for move in state.possible_moves():
            evaluation = minimax(move, max_depth - 1,
False, alpha , beta)
            min = min(value, evaluation)
            # Seguimiento de nuestra mejor puntuación
actual
            beta = min(beta, evaluation)
            if beta <= alpha:
                break
        return value
    value = math.inf
    for move in state.possible_moves():
```

```
            evaluation = minimax(move, max_depth - 1, True,
alpha, beta)
            max = max(value, evaluation)
            # Seguimiento de nuestra mejor puntuación actual
            alpha = max(alpha, evaluation)
            if beta <= alpha:
                break
      return value
```

Figura 12.- Código del algoritmo de poda alfa-beta

Juego "Búsqueda del Tesoro" (Treasure Search)

En un artículo de Peter Frey "Machine Problem Solving" de 1980 se discutió el algoritmo de poda alfa-beta aplicado a la búsqueda en un árbol de anticipación (look-ahead tree) para la mejor estrategia en un juego de dos jugadores. Incluyó un juego de demostración en lenguaje BASIC (y posteriormente en Ensamblador) del procedimiento alfa-beta llamado "Búsqueda del Tesoro" (Treasure Search). En este juego, el usuario juega contra la computadora en una cuadrícula de 8 por 8 que tiene un dígito entre 1 y 9 asignado al azar a cada una de las 64 ubicaciones de la cuadrícula. Los jugadores se turnan, con movimientos y restricciones a modo de un rey de ajedrez, para moverse de un lugar a otro, agregando el valor de cada lugar en el que aterrizan, a sus puntajes. Una vez que se ha ocupado una ubicación, su valor se reduce a 0. Cada jugador intenta seguir un camino que conduce a los

números más altos; el primer jugador en llegar a 100 puntos gana. Este fue uno de los juegos que Eldi practicaba en el Museo Elder.

En este juego competitivo entre el robot Eldi y algún visitante del Museo Elder, se incluyó la **heurística asesina (Killer heuristic),** que es un método de ordenación de movimientos, basado en la observación de que un movimiento fuerte o un pequeño conjunto de tales movimientos en una posición particular, puede ser igualmente fuerte en posiciones similares en el mismo movimiento (capa) en el árbol de juego. Retener tales movimientos evita el esfuerzo de redescubrirlos en los nodos hermanos.

Esta técnica mejora la eficiencia de la poda alfa-beta, que a su vez mejora la eficiencia del algoritmo Minimax. La poda alfa-beta funciona mejor cuando se consideran primero los mejores movimientos. Esto se debe a que los mejores movimientos son los que tienen más probabilidades de producir un *corte*, una condición en la que el programa de juego sabe que la posición que está considerando, no podría haber resultado de la mejor jugada de ambos lados y, por lo tanto, no es necesario considerarla más. Es decir, el programa de juego siempre hará su mejor movimiento disponible para cada posición. Solo necesita considerar las posibles respuestas del otro jugador a ese mejor movimiento y puede omitir la evaluación de las respuestas a movimientos (peores) que no hará.

Juego de la L

El otro juego que practicaba Eldi con los visitantes del Museo Elder era el famoso “Juego de la L”. El **juego de la L** es un simple juego de mesa de estrategia abstracto inventado por Edward de Bono. Fue introducido en su libro *The Five-Day Course in Thinking* (1967).

El juego de la L es un juego de dos jugadores que se juega en un tablero de 4 × 4 cuadrados. Cada jugador tiene un tetromino en forma de L de 3 × 2, y hay dos piezas neutrales de 1 × 1, tal y como se presenta en la Figura 13.

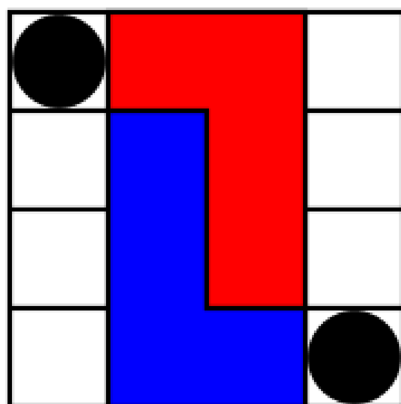

Figura 13.- Disposición del juego de la L

Reglas:

En cada turno, un jugador debe mover primero su pieza L y luego, opcionalmente, puede mover cualquiera de las piezas neutrales. El juego se gana dejando al oponente incapaz de mover su pieza L a una nueva posición.

Las piezas no pueden superponerse ni cubrir otras piezas. Al mover la pieza L, se recoge y luego se coloca en casillas vacías en cualquier parte del tablero. Puede girarse o incluso voltearse al hacerlo; la única regla es que debe terminar en una posición diferente a la posición en la que comenzó, cubriendo así al menos un cuadrado que no cubría anteriormente. Para mover una pieza neutral, un jugador simplemente la levanta y luego la coloca en un cuadrado vacío en cualquier parte del tablero.

Estrategia:

Una estrategia básica es usar una pieza neutral y la propia pieza para bloquear un cuadrado de 3 × 3 en una esquina, y usar una pieza neutral para evitar que la pieza L del oponente cambie a una posición de imagen especular. Otra estrategia básica es mover una pieza L para bloquear la mitad del tablero, y usar las piezas neutrales para evitar posibles posiciones alternas del oponente. Estas posiciones a menudo se pueden lograr una vez que se deja una pieza neutral en uno de los ocho espacios asesinos en el perímetro del tablero. Los espacios asesinos son los espacios en el perímetro, pero no en una esquina. En el siguiente movimiento, uno hace que el asesino colocado previamente sea parte de la casilla de uno, o lo usa para bloquear una posición del perímetro, y hace un bloque cuadrado o de media tabla con la propia L y una pieza neutral movida.

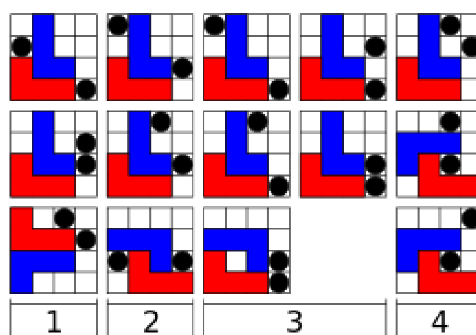

Figura 14.- Análisis del juego de la L – Situación 1

Análisis:

En la Figura 14 se pueden observar todas las posiciones, para mover el rojo, donde el rojo perderá contra un azul perfecto, y el número máximo de movimientos restantes para el rojo. Al mirar hacia adelante un movimiento y asegurarse de que uno nunca termine en ninguna de las posiciones anteriores, se puede evitar perder.

En la Figura 15 se presentan todas las posiciones finales posibles, el jugador de la L azul ha ganado.

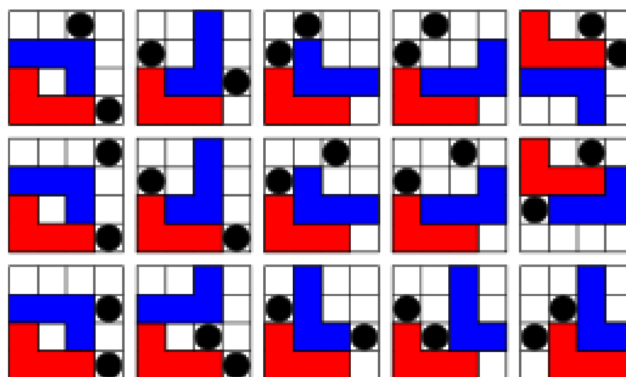

Figura 15.- Análisis del juego de la L – Situación 2

En un juego con dos jugadores perfectos, ninguno de los dos ganará ni perderá. El juego L es lo suficientemente pequeño como para tener una solución completa. Hay 2296 posibles formas válidas diferentes de disponer las piezas, sin contar una rotación o espejo de una disposición como una nueva disposición, y considerando que las dos piezas neutrales son idénticas.

El uso de la poda alfa-beta para reducir el ancho de los árboles nos ayuda a utilizar el algoritmo Minimax en juegos con grandes factores de ramificación que anteriormente se consideraban computacionalmente demasiado costosos. De hecho, Deep Blue , la computadora de Ajedrez desarrollada por IBM que derrotó al campeón mundial de Ajedrez Garry Kasparov en 1997, utilizó en gran medida algoritmos de búsqueda basados en alfa-beta paralelizados.

Existe una implementación del algoritmo del 3 en raya (Tic-Tac-Toe Minimax & poda alfa-beta) Scratch, que se ha adaptado al lenguaje castellano, tal y como se presenta en la Figura 16. La Figura 17 muestra el código en Scratch para el algoritmo Tic-Tac-Toe AI Minimax, mientras que la Figura 18 muestra el

algoritmo Tic-Tac-Toe AI Alpha-Beta. Tal y como se observa son implementaciones sencillas, que pueden revisarse de forma sencilla al tratarse de una programación intuitiva definida por bloques.

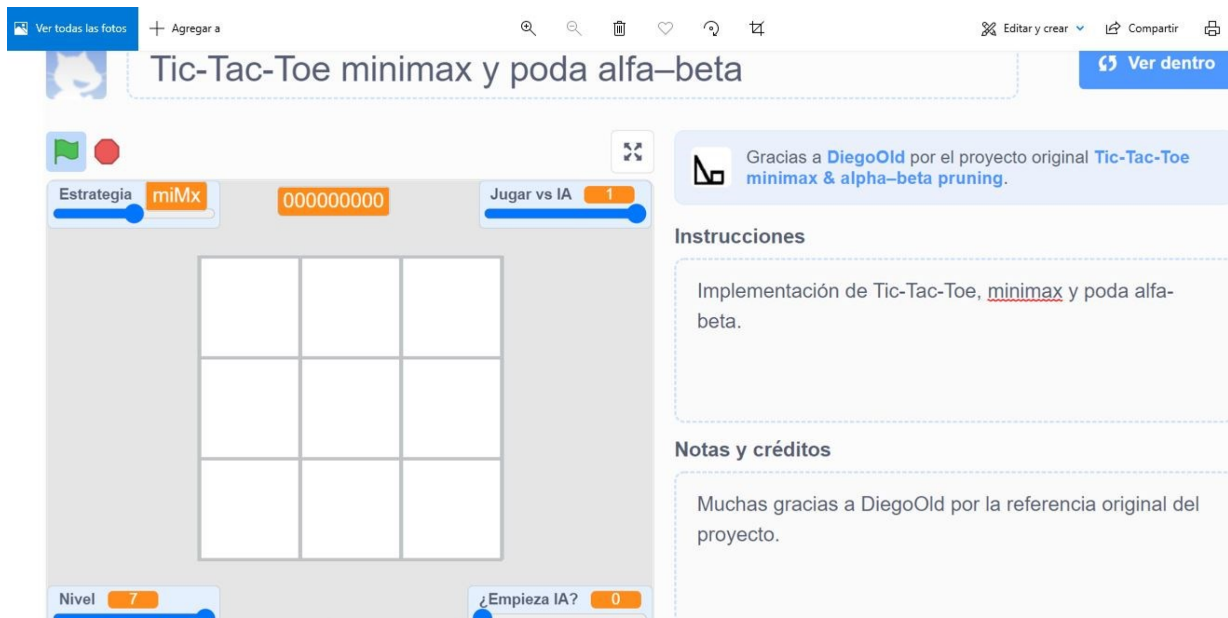

Figura 16.- Análisis del juego del Tic-Tac-Toe.
Fte: https://scratch.mit.edu/projects/613412579/

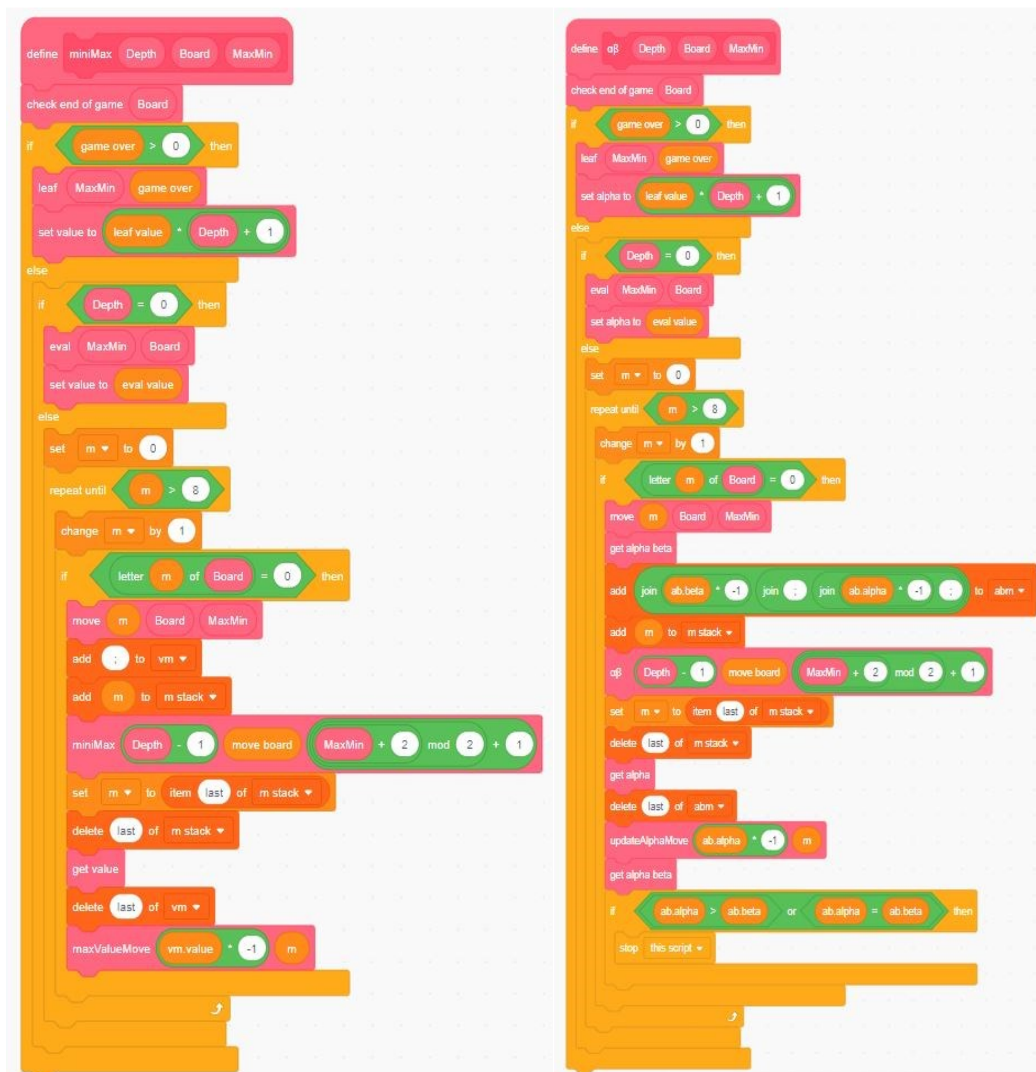

Figura 17 (izquierda).- Algoritmo Tic-Tac-Toe AI Minimax (en Lenguaje Scratch). Fuente: https://scratch.mit.edu/projects/613412579/

Figura 18 (derecha).- Algoritmo Tic-Tac-Toe Alpha-Beta (en Lenguaje Scratch) Fte: https://scratch.mit.edu/projects/613412579/

Búsqueda en árboles de Montecarlo

Parece que Minimax combinado con la poda Alpha-Beta es suficiente para construir IA de juegos sofisticados. Pero hay un problema importante que puede

hacer que estas técnicas sean inútiles. Es el problema de definir una función de evaluación sólida y razonable. Recuerde que en Ajedrez nuestra función de evaluación sumó todas las piezas blancas en el tablero y restó todas las negras. Esto resultó en valores altos cuando el blanco tenía un borde y en valores bajos cuando la situación era favorable para el negro. Si bien esta función es una buena línea de base y definitivamente vale la pena experimentar con ella, generalmente hay más complejidades y sutilezas que uno necesita incorporar para llegar a una función de evaluación sólida.

Las métricas de evaluación simples son fáciles de engañar y explotar una vez que se descubren los aspectos internos subyacentes. Esto es especialmente cierto para juegos más complejos como Go. Diseñar una función de evaluación que sea lo suficientemente compleja como para capturar la mayor parte de la información necesaria del juego requiere mucho pensamiento y experiencia en el dominio interdisciplinar de Matemáticas, Ingeniería de Software, Psicología y el juego en cuestión.

¿No existe una función de evaluación de aplicación universal que podamos aprovechar para todos los juegos, sin importar cuán simples o complejos sean?

¡Sí hay! Y se llama aleatoriedad. Con la aleatoriedad, dejamos que el azar sea nuestra guía para descubrir qué próximo movimiento podría ser el mejor para elegir.

En el popular juego de estrategia infantil “Juego de los barquitos” o “Hundir la flota” los jugadores colocan sus barcos, de distintos tamaños, sobre una rejilla de coordenadas; entonces por turnos deben disparar al enemigo esperando escuchar los famosos “agua”, “tocado” o “hundido” según lo que suceda. En la figura 19

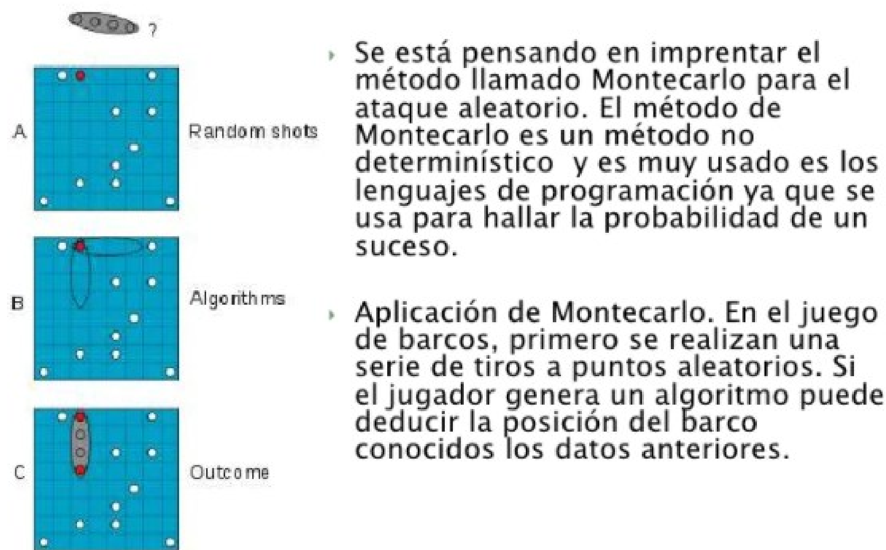

Figura 19.- Análisis matemático del juego "Batalla Naval"

Fte: https://www.microsiervos.com/archivo/juegos-y-diversion/batalla-naval-analizado-matematicamente.html

se presenta un ejemplo de análisis matemático del juego "Batalla Naval".

A continuación, exploraremos el algoritmo llamado Montecarlo Tree Search (MCTS) que se basa en gran medida en la aleatoriedad (el nombre "Montecarlo" proviene del nombre de la ciudad de los juegos de azar en Montecarlo) como un componente central para las aproximaciones de valor. Como su nombre lo indica, MCTS también crea un árbol de juego y realiza cálculos sobre él para encontrar la ruta del resultado potencial más alto. Pero hay una ligera diferencia en cómo se construye este árbol.

Pretendamos una vez más que estamos jugando al Ajedrez como jugador blanco. Ya hemos jugado durante un par de rondas y de nuevo depende de nosotros elegir el próximo movimiento que nos gustaría jugar. Además, supongamos que no conocemos ninguna función de evaluación que podamos

aprovechar para calcular el valor de cada posible movimiento. ¿Hay alguna forma en que podamos descubrir qué movimiento nos pondría en una posición en la que podríamos al final ganar? Resulta que hay un enfoque realmente simple que podemos adoptar para resolver esto. ¿Por qué no dejamos que ambos jugadores jueguen docenas de juegos aleatorios comenzando desde el estado en el que nos encontramos actualmente? Si bien esto puede parecer contrario a la intuición, tiene sentido si lo piensa. Si ambos jugadores comienzan en el estado de juego dado, juegan miles de juegos aleatorios y el jugador blanco gana el 80% de las veces, entonces debe haber algo en el estado que le dé a las blancas una ventaja. Lo que estamos haciendo aquí es básicamente explotar la Ley de los Grandes Números (LGN) para encontrar el resultado "verdadero" del juego para cada movimiento potencial que podamos realizar.

La siguiente descripción mostrará en detalle cómo funciona el algoritmo MCTS. En aras de la simplicidad, nuevamente nos enfocamos únicamente en dos movimientos jugables en cualquier estado dado (como ya hemos descubierto, hay un promedio de aproximadamente 30 movimientos diferentes que podemos jugar en Ajedrez).

Antes de continuar, necesitamos hacer algunas definiciones menores del camino. En MCTS hacemos un seguimiento de dos parámetros diferentes para cada nodo de nuestro árbol. A esos parámetros los llamamos t y n. t significa "total" y representa el valor total de ese nodo. n es el "número de visitas" que refleja el número de veces que hemos visitado este nodo mientras caminamos por el árbol. Al crear un nuevo nodo siempre inicializamos ambos parámetros con el valor 0. Además de los dos nuevos parámetros que almacenamos para cada nodo, existe la fórmula denominada "Límite de Confianza Superior" (LCS), que es la siguiente (1):

$$x_i + C\sqrt{\frac{\ln(N)}{n_i}} \qquad (1)$$

Básicamente, esta fórmula nos ayuda a decidir desde qué nodo próximo y, por lo tanto, posible movimiento del juego, debemos elegir para comenzar nuestra serie de juegos aleatorios (llamada "lanzamiento"). En la formula x_i representa el valor promedio del estado del juego con el que estamos trabajando, C es una constante llamada "temperatura" que debemos definir manualmente (solo la configuramos en 1.5 en nuestro ejemplo aquí), N representa las visitas del nodo padre y n_i representa las visitas actuales de los nodos. Cuando usamos esta fórmula en nodos candidatos para decidir cuál explorar más, siempre estamos interesados en el resultado mayor.

Tenga en cuenta que esta fórmula existe y será útil para nosotros mientras trabajamos sin nuestro árbol. Entraremos en más detalles sobre su uso mientras caminamos por nuestro árbol. Con esto fuera del camino, es hora de aplicar MCTS para encontrar el mejor movimiento que podamos jugar.

Comenzamos con el mismo nodo raíz del árbol con el que ya estamos familiarizados. Este nodo raíz es nuestro punto de inicio y refleja el estado actual del juego. Basándonos en este nodo, ramificamos nuestros dos nodos secundarios, como se presenta en la Figura 20.

Lo primero que debemos hacer es usar la fórmula (1) y calcular los resultados para ambos nodos secundarios. Resulta que necesitamos insertar 0 para casi todas las variables en nuestra fórmula (1), ya que aún no hemos hecho nada con nuestro árbol y sus nodos. Esto resultará en ∞ para ambos cálculos, de acuerdo con (2),

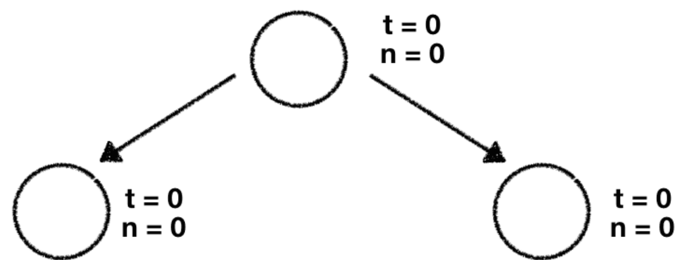

Figura 20.- Ramificación de dos nodos secundarios aplicando MCTS

reemplazando el 0 en el denominador con un número muy pequeño, a fin de evitar la división por cero.

$$S_1 = 0 + 1.5\sqrt{\frac{\ln(0)}{0.0001}} = \infty \quad (2)$$

En consecuencia, se puede elegir qué nodo queremos explorar más a fondo. Seguimos adelante con el nodo más a la izquierda y realizamos nuestra fase de implementación, lo que significa que jugamos docenas de juegos aleatorios comenzando con este estado de juego.

Una vez hecho esto, obtenemos un resultado para este lanzamiento específico (en nuestro caso, el porcentaje de victorias para el jugador blanco). Lo siguiente que debemos hacer es propagar este resultado por el árbol hasta llegar al nodo raíz. Entonces, actualizamos ambos *t* y *n* con los valores respectivos para cada nodo que encontremos. El árbol resultante se presenta en la Figura 21.

A continuación, comenzamos en nuestro nodo raíz nuevamente. Una vez más se usamos la fórmula LCS (1), conectamos nuestros números y calculamos su puntaje para ambos nodos (3).

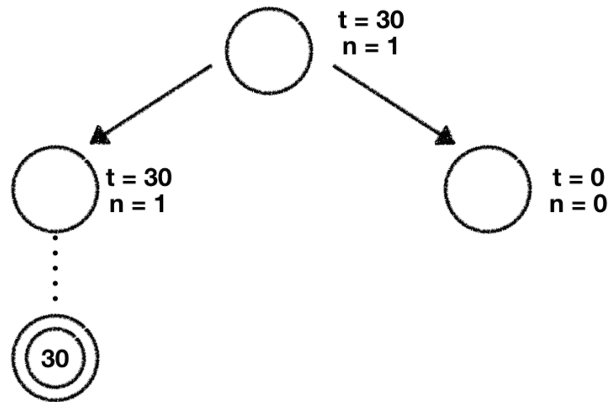

Figura 21.- Árbol resultante tras propagar resultados – Paso 1

$$
\begin{aligned}
S_1 &= 30 + 1.5\sqrt{\frac{\ln(1)}{1}} = 30 \\
S_2 &= 0 + 1.5\sqrt{\frac{\ln(0)}{0.0001}} = \infty
\end{aligned}
\tag{3}
$$

Dado que siempre elegimos el nodo con el valor más alto, ahora exploraremos el de la derecha. Una vez más, realizamos nuestro lanzamiento en función del movimiento que propone este nodo y recopilamos el resultado final después de que hayamos terminado todos nuestros juegos aleatorios.

Lo último que debemos hacer es propagar este resultado hasta llegar a la raíz del árbol. Mientras hacemos esto, actualizamos los parámetros de cada nodo que encontramos, tal y como se presenta en la Figura 22.

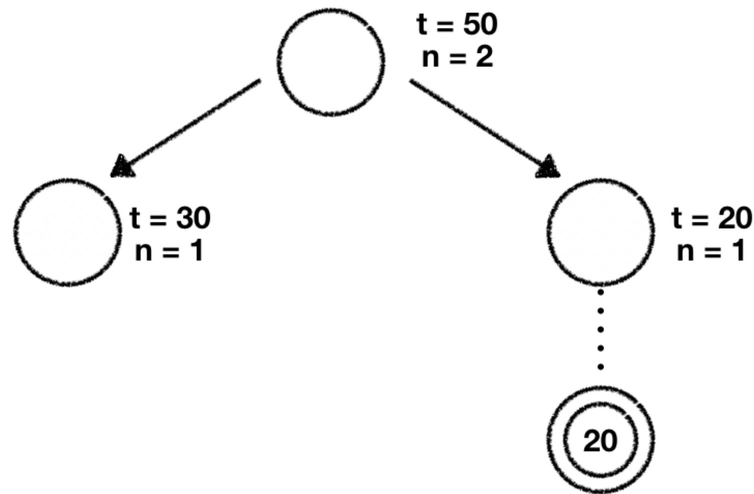

Figura 22.- Árbol resultante tras propagar resultados – Paso 2

Ahora hemos explorado con éxito 2 nodos secundarios en nuestro árbol. Puede que ya lo hayas adivinado. Comenzaremos de nuevo en nuestro nodo raíz y calcularemos la puntuación LCS de cada nodo secundario para determinar el nodo que debemos explorar más a fondo. Al hacer esto, obtenemos los siguientes valores (4):

$$S_1 = 30 + 1.5\sqrt{\frac{\ln(2)}{1}} \approx 31.25$$
$$S_2 = 20 + 1.5\sqrt{\frac{\ln(2)}{1}} \approx 21.25 \quad (4)$$

El valor más grande es el que hemos calculado para el nodo más a la izquierda, por lo que decidimos explorar ese nodo más.

Ya que este nodo no tiene nodos secundarios, agregamos dos nuevos nodos que representan los movimientos potenciales que podemos ejecutar en el árbol. Inicializamos ambos parámetros (t y n) con 0.

Ahora tenemos que decidir cuál de esos dos nodos deberíamos explorar más a fondo. Usamos la fórmula LCS para calcular sus valores. Dado que ambos t y n tienen valores de cero son ambos ∞ así que decidimos elegir el nodo más a la izquierda. Una vez más hacemos un lanzamiento, recuperamos el valor de esos juegos y propagamos este valor hasta el árbol hasta llegar al nodo raíz de los árboles, actualizando todos los parámetros del nodo a lo largo del camino, como se presenta en la Figura 23.

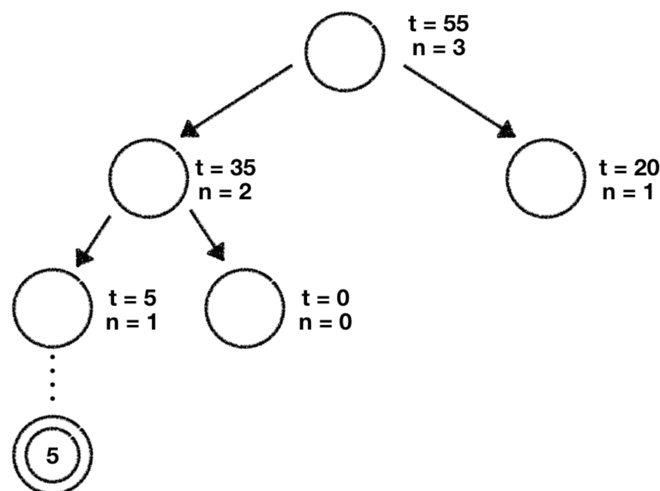

Figura 23.- Árbol resultante tras propagar resultados – Paso 3

La próxima iteración comenzará una vez más en el nodo raíz donde usamos la fórmula LCS para decidir qué nodo hijo queremos explorar más a fondo. Como podemos ver un patrón aquí y no quiero aburrirlos, no voy a describir los próximos pasos con gran detalle. Lo que haremos es seguir exactamente el mismo

procedimiento que hemos utilizado anteriormente, que se puede resumir de la siguiente manera:

- Comience en el nodo raíz y use la fórmula LCS para calcular la puntuación de cada nodo secundario
- Elija el nodo secundario para el que ha calculado la puntuación LCS más alta
- Compruebe si el hijo ya ha sido visitado
- Si no es así, haz un lanzamiento
- En caso afirmativo, determine los próximos estados potenciales a partir de ahí.
- Utilice la fórmula de LCS para decidir qué nodo hijo elegir
- Hacer un lanzamiento
- Propague el resultado a través del árbol hasta que llegue al nodo raíz.
- Repetimos este algoritmo hasta que nos quedemos sin tiempo o alcanzamos un valor umbral predefinido de visitas, profundidad o iteraciones. Una vez que esto sucede, evaluamos el estado actual de nuestro árbol y elegimos los nodos secundarios que maximizan el valor t. Gracias a las decenas de partidas que hemos jugado y a la Ley de los Grandes Números, podemos estar muy seguros de que esta jugada es la mejor que podemos realizar.

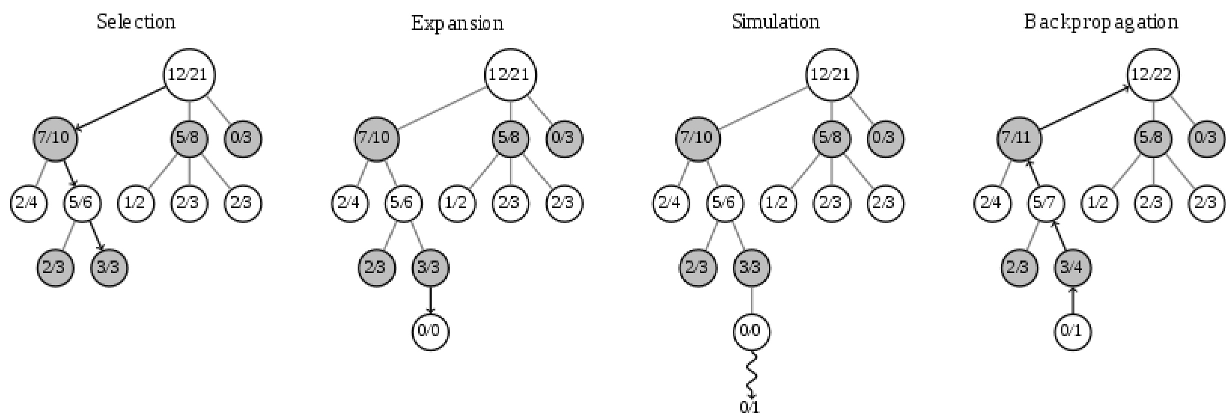

Figura 24.- Resumen de pasos del algoritmo MCTS

La Figura 24 presenta un resumen de los pasos del algoritmo MCTS:

- *Selección*: empezar desde la raíz R y seleccionar nodos hijos sucesivos hasta alcanzar un nodo hoja L. La sección de abajo describe más de una manera de elegir nodos hijos, que permiten que el árbol de juego se expanda hacia movimientos más prometedores, que es la esencia del árbol de búsqueda Montecarlo.
- *Expansión*: a menos que L termine el juego con una victoria/pérdida para cualquiera de los jugadores, crear uno o más nodos hijos y elegir entre ellos un nodo C. Los nodos hijos son cualquier movimiento válido desde la posición del juego definida por L.
- *Simulación*: jugar una reproducción aleatoria desde el nodo C.
- *Retropropagación*: utilizar el resultado de la reproducción para actualizar la información en los nodos en el camino de C a R.

Puede estar de acuerdo en que parece que MCTS es muy intensivo en computación, ya que tiene que ejecutar miles de juegos aleatorios. Esto es definitivamente cierto y debemos ser muy inteligentes en cuanto a dónde debemos

invertir nuestros recursos para encontrar el camino más prometedor en nuestro árbol. Podemos controlar este comportamiento con el parámetro "temperatura" mencionado anteriormente C en nuestra fórmula LCS. Con este parámetro equilibramos la compensación entre "exploración y explotación". Un gran C el valor nos pone en modo de "exploración". Pasaremos más tiempo visitando los nodos menos explorados. Un pequeño valor para C nos pone en modo de "explotación" donde volveremos a visitar los nodos ya explorados para recopilar más información sobre ellos.

Dada la simplicidad y aplicabilidad debido a la explotación de la aleatoriedad, MCTS es un algoritmo de búsqueda de árbol de juegos ampliamente utilizado. DeepMind extendió MCTS con Deep Neural Networks para optimizar su rendimiento en la búsqueda de los mejores movimientos de Go para jugar. La IA del Juego resultante fue tan fuerte que alcanzó un rendimiento de nivel sobrehumano y derrotó al Campeón del Mundo de Go Lee Sedol por 4-1 en las 5 partidas disputadas. Posteriormente DeepMind desarrolló con los mismos ingredientes computacionales el Programa AlphaZero, que ganó con solvencia un torneo de Ajedrez enfrentándose al líder de los motores de ajedrez, el famoso StockFist.

Conclusión

En este artículo hemos analizado dos algoritmos de búsqueda de árboles diferentes que se pueden usar para construir IA de juegos sofisticados.

Si bien Minimax combinado con la poda alpha-beta es una solución sólida para abordar juegos donde se puede definir fácilmente una función de evaluación para estimar el resultado del juego, Montecarlo Tree Search (MCTS) es una solución

de aplicación universal dado que no es necesaria una función de evaluación debido a su dependencia de la aleatoriedad.

Minimax y MCTS son solo el comienzo y se pueden ampliar y modificar fácilmente para trabajar en entornos más complejos.

DeepMind combinó inteligentemente MCTS con Deep Neural Networks para predecir los movimientos del juego Go, mientras que IBM extendió la búsqueda del árbol alpha-beta para calcular los mejores movimientos de Ajedrez posibles para jugar.

Referencias bibliográficas

Alpha-Beta Pruning. https://www.geeksforgeeks.org/minimax-algorithm-in-game-theory-set-4-alpha-beta-pruning/

Frey, P. (1980). Machine Problem Solving, *BYTE*.

Gale, S. (1981). Alpha-Beta Tree Search Converted to Assembler, *BYTE*.

Knuth, D. (1975). An Analysis of Alpha-Beta Pruning, *Artifficial Inteligence.*

Minimax. https://www.geeksforgeeks.org/minimax-algorithm-in-game-theory-set-1-introduction/?ref=lbp

Minimax (Tic-Tac-Toe). https://www.geeksforgeeks.org/minimax-algorithm-in-game-theory-set-3-tic-tac-toe-ai-finding-optimal-move/?ref=lbp

Montecarlo tree search. https://www.geeksforgeeks.org/ml-monte-carlo-tree-search-mcts/

Othello (Reversi) AI. https://scratch.mit.edu/projects/402238860

Tic-Tac-Toe Minimax & poda alfa-beta. https://scratch.mit.edu/projects/613412579/