Aplicación de distintos algoritmos de IA y RL para aprender a jugar a Connect4

Ivan Sanchez Resina

Resumen– El objetivo de este proyecto es implementar un juego de Connect 4 y añadirle varios algoritmos de IA que sirvan como oponentes a un jugador real y entre ellos mismos. De estos algoritmos implementados realizar varias pruebas sobre los parámetros que regulan su comportamiento con el objetivo de estudiar cómo interactua cada uno con los otros y cual es el más eficiente.

Los algoritmos implementados son Minimax, Minimax con poda alfabeta, Monte Carlo Tree Search y una implementación parcial del algoritmo de Reinforcement Learning conocido como AlphaZero, el cual aplica el MCTS pero utilizando una red neuronal entrenada para predecir los mejores movimientos
Palabras clave– Algortimos IA, Monte Carlo Tree Search, Teoría de juegos, Minimax, Minimax AlphaBeta, Connect4, AlphaZero, Red Neuronal

Abstract– The idea of this project is to implement a fully-functional game of Connect 4 and add different IA algorithms that can act as opponents to a human Player and between themselves. Of these implemented algorithms, realise several tests on the parameters that regulate their behaviour in order to study how each interacts with the others and which one is the most effective.

The implemented algorithms are Minimax, Minimax with alpha-beta pruning, Monte Carlo Tree Search and a partial implementation of the Reinforcement Learning algorithm known as AlphaZero, that applies the MCTS but using a trained neural network to predict the bests movements.
Keywords– IA-driven algorithm, Monte Carlo Tree Search, Game Theory, Minimax, Minimax AlphaBeta, Connect4, AlphaZero, Neural Network

1 Introducción

Los juegos constituyen una parte muy importante en la vida de las personas. Especialmente cuando somos pequeños, ya que nos entretienen y nos divierten mientras nos permiten **desarrollar varias de nuestras capacidades** sin darnos cuenta. De las varias formas que pueden tomar la que más me ha llamado la atención siempre, y no solo como jugador si no también a un nivel puramente conceptual, son los juegos de mesa.

Hay muchos tipos de juegos de mesa, pero creo que los más interesantes son los que, con un mínimo de materiales y un puñado escaso de normas permiten **desarrollar tus habilidades cognitivas mientras te diviertes** junto a otras personas. Desde luego hay mucho mérito en crear un juego complejo, con varias páginas de normativa, mil casuísticas y excepciones, una caja enorme con materiales para montar el escenario y que requiera un gran número de jugadores. Pero siempre he pensado que los juegos que más mérito tienen son precisamente los que huyen de esto. Un tablero y un puñado de fichas, varias decenas de cartas permiten tener horas de diversión con un tiempo de aprendizaje mínimo y sin necesidad de materiales, ya que juegos como la oca, el parchís o el 3 en ralla se pueden realizar incluso sobre una hoja de papel, con dos garabatos y varias piedrecillas como fichas.

• E-mail de contacto: isanres@gmail.com
• Mención realizada: Computación
• Trabajo tutorizado por: Maria Vanrell (departament)
• Curs 2019/20

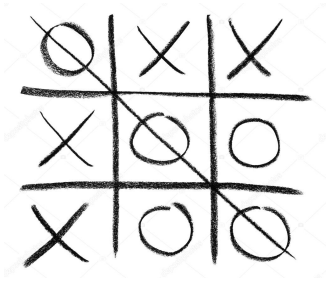

Fig. 1: No se requiere más que una superfície donde dibujar

Pero no solo en la sencillez radica su atractivo, o al menos no para mí. Lo que más me intriga es cómo estos juegos, en principio tan simples, permiten este desarrollo de distintas habilidades. **Rapidez mental, pensamiento lateral, capacidad de cómputo o memoria** son solo algunos ejemplos [1] de las habilidades que puedes mejorar jugando de forma asidua a estos juegos.

Siendo una parte tan importante de nuestra sociedad, era evidente que llegaría un momento en el que se introduciría la **automatización de los oponentes**, para permitir más opciones y dar flexibilidad a estos juegos, mediante los algoritmos de IA oponentes. Sin embargo... ¿es realmente eso todo lo que pueden aportar los algoritmos y la automatización a los juegos de mesa? ¿Es la **sustitución del oponente humano** todo a lo que han de aspirar estos agentes artificiales? Es evidente que si se limitan a eso los juegos pierden parte de su gracia, ya que eliminas la parte comunitaria, el aliciente de competir contra otras personas y, más importante, de mejorar a un juego con otras personas. Y es aquí donde entran las técnicas de Reinforcement Learning, o RL, que permiten a los agentes artificiales no solo ser buenos a un juego si no **aprender a jugarlo** por sí mismos.

Estas técnicas permiten principalmente dos cosas, siendo la más llamativa sin duda el hecho de que permite a un agente artificial mejorar por su propia cuenta para saber adaptarse a distintos tipos de jugadores y superar así a los mejores jugadores humanos de cada juego. Sin embargo hay una segunda aplicación de estas técnicas que a menudo pasa desapercibida: la utilización de un agente artificial que puede aprender a jugar a un juego nos permite **desgranar cada pequeño detalle de ese juego**. No sólo creo que sea importante el resultado final de estos algoritmos (es decir, un jugador muy competente del juego) si no que creo que es casi más interesante el hecho de que podamos **ver y entender cómo ha aprendido a jugar** al juego, qué procesos hay detrás del juego, clasificar tipos de jugadores e identificar cómo potenciar esas habilidades que comentaba al principio que estos juegos nos permiten adquirir. Porque de esta forma se **abre un nuevo mundo de posibilidades**. Agentes que permiten realizar un seguimiento intensivo de pacientes con Alzheimer, que permiten identificar los puntos flacos de un estudiante que está en pleno proceso de aprendizaje y actúa en consecuencia para potenciarlos o que incluso pueden **actuar no ya como diagnóstico si no como remedio parcial** también. Hoy en día ya es posible diseñar juegos que se pueden utilizar como refuerzo para la recuperación de pacientes con discapacidades cognitivas [2], pero la utilización de los datos de estos agentes artificiales, la utilización de algoritmos de RL como AlphaZero o QLearning nos puede permitir dar un paso más allá.

Así pues, mi intención al desarrollar este proyecto era utilizarlo como **una aproximación al mundo de los agentes artificiales aplicados a los juegos de mesa**, con la ambición de entender cómo funcionan y ser capaces de progresar hacia una aplicación más óptima de las técnicas que se han usado, que permitan la creación de agentes que puedan utilizarse como soporte para personas con problemas de salud mentales o infantes en desarrollo.

2 Estado del arte

El uso de Reinforcement Learning (RL) para juegos de mesa está en auge ahora mismo. Constantemente aparecen **modificaciones a algoritmos existentes y algoritmos nuevos que superan a los anteriores por mucho**. De hecho, cuando se inició este proyecto, en septiembre de 2019, uno de los algoritmos de RL aplicado a juegos de mesa más importante, por no decir el más importante y eficaz era AlphaZero, mientras que hace apenas un mes, en diciembre, apareció el algoritmo MuZero, hermano menor de AlphaZero, cuya principal característica es que puede aprender sin un simulador perfecto. Un simulador perfecto es como se conoce a los juegos como Go, Ajedrez o Connect4, los cuales **te permiten saber exactamente qué estado tendrás después de realizar cada movimiento** o incluso varios movimientos. MuZero, sin embargo, permite aprender en entornos donde no sabe exactamente qué efectos tendrán sus acciones, lo cual abre un nuevo mundo de entornos a los que se pueden aplicar estos algoritmos.

Volviendo a AlphaZero, este algoritmo es a su vez una evolución de AlphaGo, el cual, entre sus varias versiones, llegó a destrozar a varios de los mejores jugadores de Go y incluso a ganar al mejor jugador del mundo [3]. La evolución es tal que AZ fue capaz de vencer a la versión de AG que venció a uno de los mejores jugadores de Go del mundo en 2016... tras solo 3 días de entrenamiento y sin tener **ningún tipo de conocimiento previo** sobre el Go, simplemente extrayendo la información de las partidas. 21 días después, fue capaz de vencer al mejor jugador del mundo y a otros 60 jugadores profesionales sin problemas. Poco después fue capaz de obliterar por completo a Stockfish, la herramienta de entrenamiento de ajedrez que utilizada por la gran mayoría de profesionales del ajedrez... tras solo 4 horas de entrenamiento y sin ningún conocimiento previo sobre ajedrez.[4][5]

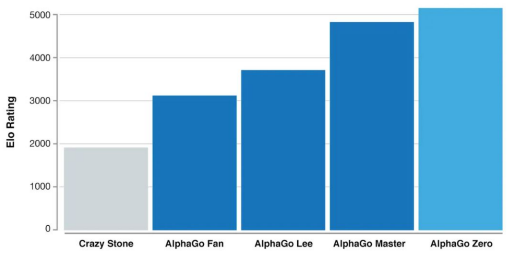

Fig. 2: Las victorias de AZ contra las versiones de AG. La última empezó con 0 conocimiento sobre el juego

Por si fuera poco, todas estas mejoras, actualizaciones y victorias han sucedido en **los últimos 3 años**, lo que demuestra que estamos ante un campo en pleno auge y que promete bastante. Aún así, lo más impresionante en mi opinión sobre la evolución en este campo de los últimos años es que las mejoras no solo han sido respecto a el número de victorias. Los avances se han realizado en tiempo de computación, en entornos a los que aplicar los algoritmos y también para la eficacia del algoritmo. No es precisamente un campo con pocas oportunidades para innovar, si no uno que **permite a todo el mundo aplicar su aproximación al mismo y contribuir** al desarrollo de esta tecnología.

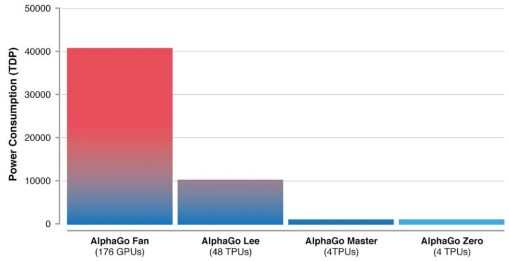

Fig. 3: Cada una de las nuevas versiones mejoraba la eficiencia (velocidad) además de la eficacia (victorias)

3 Objetivos

A nivel general y sin entrar en requisitos para considerar completo el proyecto este trabajo tiene principalmente un objetivo: servir como herramienta que me permita introducirme en el mundo de los algoritmos de RL aplicados a juegos de mesa y **permitirme entender cómo funcionan para seguir explorando este campo en un futuro**.

Esta idea se delimitó más al inicio del proyecto con el objetivo de tener una carga de trabajo asequible y que proporcionara un **equilibrio correcto entre aplicación de conocimientos que ya poseía e investigación y aplicación de algoritmos desconocidos**. Así pues, se decidió lo siguiente:

- El juego que se desarrollaría sería el Connect4 o Conecta 4 con gravedad, un juego sencillo con normas básicas que simplificaba la parte del proyecto centrada en la representación del juego y las funciones básicas como mover ficha o las posibilidades en cada ronda.
- Antes de iniciar la aplicación de un algoritmo de RL se implementaría un algoritmo conocido que no pudiese lastrar mucho el desarrollo
- Como algoritmo de RL se utilizaría un esqueleto de código que adaptaba el AlphaZero a Connect4, así que no se tendría que implementar de 0, solo entenderlo y variar los parámetros para entender su funcionamiento
- En caso de que el desarrollo avanzara de forma correcta se podrían tomar dos caminos de mejora: modificar el juego base e implementar el ajedrez o aplicar un segundo algoritmo de RL

Con estas directivas en mente se establecieron los siguientes objetivos:

- Implementar un juego funcional de Connect4
- Implementar el algoritmo Minimax como algoritmo conocido
- Entender el funcionamiento de AZ y adaptar el código para que pudiese jugar contra el Minimax
- Realizar variaciones en ambos algoritmos para estudiar la interacción entre ellos
- Investigar un segundo algoritmo de RL para poder implementarlo en caso de que la implementación avanzara de forma correcta

4 Metodología y herramientas

A lo largo del proyecto se ha usado un número reducido de herramientas, debido principalmente a que las tres fases que se han ido alternando a lo largo del desarrollo (investigación, implementación y gestión) se han podido llevar a cabo utilizando una o dos herramientas distintas únicamente.
Así pues, haré en esta sección una breve mención a las herramientas utilizadas y después explicaré las fases que se han ido realizando durante el desarrollo.

- Python 3.7 como único lenguaje de programación para todo el código implementado
- Pycharm como entorno de desarrollo principal
- Spyder (distribución de Anacondoa) se ha utilizado para realizar el entrenamiento básico de la NN de AlphaZero y también se ha utilizado para crear el entorno que se ha usado luego desde Pycharm, ya que desde este último entorno no fue posible realizar la instalación de todas las librerías que eran necesarias para ejecutar el código utilizado de AlphaZero
- Google Scholar para investigación sobre los distintos módulos implementados
- Trello como método de organización y visualización de tareas a realizar
- Overleaf para desarrollar el LaTeX del Informe Final
- GitHub se ha utilizado para subir el código al final del desarrollo y también de manera local para la implementación de cada algoritmo que se ha intentado implementar a partir de Minimax

El desarrollo de este proyecto se ha estructurado en 4 períodos, el final de cada uno de los cuales está marcado por la entrega de uno de los informes (inicial, seguimiento 1 y 2 y final). La metodología que se ha utilizado consta, como he comentado antes, **de 3 fases, que se han ejecutado siguiendo la misma estructura** en los 3 últimos períodos. Estas fases son las siguientes:

- Fase de investigación: Durante esta fase se ha recopilado información sobre los distintos algoritmos y se ha intentado entender cuál es su funcionamiento básico y cómo se pueden implementar con la estructura de tablero propia. Dado que el principal objetivo de este proyecto era familiarizarse con distintos algoritmos, ha tenido un papel crucial y se ha llevado a cabo en las 4 fases.

- Fase de desarrollo: Durante esta fase se implementaban los algoritmos estudiados durante la fase de investigación. Esta fase ha tenido su mayor importancia en los períodos 2 y 3, mientras que tuvo un papel menor en el 4 y no estuvo presente en el 1.
- Fase de gestión: Durante esta fase se escoge qué tareas se iban a implementar durante un período y se estimaba el tiempo de investigación y desarrollo que iba a llevar. También se diseñaban planes de contingencia y el mínimo de tareas necesarias para considerar un período satisfactorio. Se realizaba al inicio de cada período y se plasmaban los resultados de esta fase en los informes. Además, se utilizaban 5 tablas en la aplicación Trello para llevar un control de las tareas realizadas y por realizar [6].

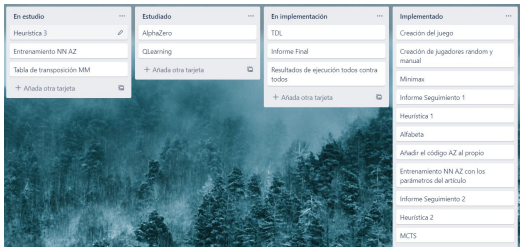

Fig. 4: Imagen de las 5 tablas con las que se ha trabajado en el estado final del proyecto

5 DESARROLLO

El objetivo de esta sección es hacer una **breve presentación de los distintos módulos implementados**, con el objetivo de entender qué estructura tienen y en el caso de los algoritmos una breve explicación con pseudocódigo de su funcionamiento.

5.1. El juego

El primer elemento desarrollado fue el juego en sí, implementando los jugadores Random y Manual para poder realizar pruebas rápidas y sencillas sobre si el funcionamiento del flujo de la partida era el correcto. Además, se podía utilizar el jugador Random para probar la eficacia y eficiencia de los algoritmos y heurísticas implementadas. Este código está formado por 3 archivos de código: **board** (incluye la definición de la representación del tablero), **players** (incluye la definición de los jugadores) y **main** (incluye el flujo principal del juego)

5.1.1. board

El tablero es la parte más importante del código, concretamente la representación que se ha implementado. Después de consultar algunos artículos explicando en profundidad el funcionamiento del juego [7] decidí implementar lo siguiente:

- Como estructura de datos, en vez de una matriz de X*Y, **una lista de listas vacías**, cada una de las cuales representa una columna. Así, para realizar un movimiento simplemente hay que poner un 1 o un 0 en la lista que toque y no hay que jugar con los índices ni pasar varios datos. Además, es más rápido comprobar si en una columna podemos poner ficha o no o el estado actual de la partida.
- Como funciones asociadas hay varias, siendo las más importantes makeMove (coloca una ficha en la columna que toque), isTerminal (devuelve el estado de la partida actual) y children (devuelve una lista con todos los tableros que podemos tener a partir del tablero actual)

Además, tanto el tamaño del tablero como el número de fichas que hay que encadenar para ganar son parametrizables, para poder realizar distintas pruebas sobre el rendimiento de los algoritmos.

5.1.2. players

Para los jugadores se creó una clase principal, que recibe varios parámetros en función del tipo de jugador (se han implementado 4 jugadores: Random, Minimax, Monte-Carlo Tree Search i AlphaZero) y luego una subclase para cada algortimo implementado que tiene una función findMove para decidir el siguiente movimiento a hacer. En los siguientes apartados se profundizará en los tipos de jugadores que hay y en cómo funcionan.

5.1.3. main

El bucle principal no tiene ningún misterio, es el bucle que simula una partida normal y sigue el esquema de la figura 5

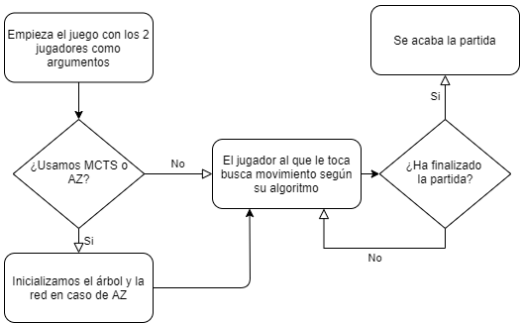

Fig. 5: Estructura principal del juego

5.2. Minimax

El primer jugador implementado fue el Minimax básico. Este algoritmo, está explicado en más profundidad en otros artículos y páginas [8] [9], por lo que simplemente se explicarán dos heurísticas que se han propuesto para que use el algoritmo. En resumen, la heurística es la función matemática que evalua un tablero concreto y determina cuán bueno o malo es para el jugador Minimax.

5.2.1. Heurísticas

Se han desarrollado 2 heurísticas, con 2 mejoras cada una.

H1: En esta heurística se busca el número de piezas juntas que hay de cada jugador y se suma o resta al valor total en función de si son nuestras o de nuestro oponente.

Cuantas más juntas más valor tienen. En un principio la escala definida era 10 para 2 fichas juntas, 100 para 3 y 10000 para 4, aunque se realizaron varias pruebas para comprobar cuáles eran los valores más efectivos.

H2: Esta heurística es algo más exhaustiva. Para cada posición del tablero se calcula cuantas posibilidades de ganar tenemos (en un tablero sin piezas hay 4 posibilidades para cada casilla: horizontal, vertical, diagonal derecha y diagonal izquierda). Si hay una pieza que no es nuestra en alguna de las 4 direcciones dentro del alcance esa posibilidad no se contabiliza. Para el número total se suman las posibilidades propias y se restan las del oponente. Debido al gran cómputo que exige esta opción se limitó a buscar en las casillas hasta la máxima fila ocupada en el tablero actual, ya que si no el tiempo para tomar una decisión es demasiado, incluso a profundidades bajas.

Mejoras: Después de implementar ambas heurísticas se introdujo un nuevo elemento a ambas que dió buenos resultados: pesos a las columnas. Para cada heurística se sumaban puntos por las columnas centrales (5 para la 4, 3 para la 3 y la 5, 1 para la 2 y la 6) y se ignoraban las marginales.

5.2.2. Alfabeta

Al igual que con Minimax, no entraré mucho en detalle con la implementación de la mejora AlfaBeta, ya que existen otros artículos que tratan el algoritmo en más profundidad [10] [11], por lo que simplemente explicaré que es una mejora del algoritmo Minimax que no afecta a la eficacia del mismo (gana lo mismo) pero si al tiempo de cómputo, ya que evita explorar las ramas menos prometedoras, mientras que el Minimax base las explora todas. Esto hace que en el Minimax base solo pudiera usar hasta 4-5 de profundidad sin que el tiempo fuera demasiado elevado mientras que con AlfaBeta se ha podido llegar hasta 8 sin problemas. En la tabla 1 se ilustra la mejora en tiempo que introduce la poda alfbeta. Por tanto nuestro jugador Minimax incluirá siempre la mejora de la poda alfabeta.

Tiempo en AB - MM		Profundidad de búsqueda			
		2	3	4	5
Partidas jugadas	5	0.07 - 0.15	0.32 - 0.79	0.86 - 4.83	2.56 - 70.34
	10	0.18 - 0.23	0.38 - 1.23	1.33 - 10.14	4.64 - 130.4
	15	0.21 - 0.41	0.76 - 2.24	2.37 - 13.73	-
	20	0.3 - 0.56	1.17 - 3.94	3.89 - 24.02	-

TAULA 1: COMPARACIÓN EN TIEMPO DE PARTIDAS ENTRE EL MINIMAX BÁSICO I EL MINIMAX CON PODA ALFA-BETA. RESULTADOS PARA DIFERENTES NÚMEROS DE PARTIDAS VERSUS EL JUGADOR RANDOM.

5.3. Monte Carlo Tree Search

Dado que es un algoritmo de búsqueda en árbol, funciona con una representación de nodos, donde un nodo representa un tablero en una situación concreta y las ramas representan un movimiento concreto. además de nodo y rama es importante conocer el término **nodo hoja**, que se refiere a un nodo donde no hemos explorado aún todas las posibles acciones desde él y **nodo terminal**, que se refiere a un nodo que representa una situación final de partida (empate o victoria de uno de los jugadores).
En los siguientes apartados veremos más en profundidad el funcionamiento del algoritmo.

5.3.1. Fases del algoritmo

El funcionamiento es principalmente el siguiente: se ejecuta número predefinido de simulaciones para determinar el mejor movimiento posible. Cada una de estas simulaciones sucede en 4 fases y resumido al máximo consiste en escoger un tablero y simular una partida desde ese tablero en concreto haciendo movimientos aleatorios hasta ver quien gana.

- Fase de selección: Esta fase se da mientras estemos en un nodo explorado. Se aplica una fórmula matemática para escoger cuál de los hijos de este nodo es el mejor para seguir buscando y nos movemos a ese. Esta fase continua hasta que lleguemos a un nodo hoja.

- Fase de expansión: En esta fase lo que hacemos es abrir los hijos del nodo que hemos seleccionado. Es posible abrirlos todos en esta fase o abrir solo uno, más adelante se explicará la diferencia.

- Fase de simulación: Escogiendo uno de los hijos del nodo anterior (si hemos abierto más de uno lo escogemos aleatoriamente) realizamos movimientos aleatorios desde aquí hasta llegar a un nodo terminal. esta fase no abre nodos ni influye de ninguna forma en la estructura del árbol, solo realizamos movimientos sobre un tablero temporal.

- Fase de backpropagation: Una vez hemos simulado hasta un nodo terminal cogemos el resultado de esa partida simulada y actualizamos los valores de los nodos de la ruta que hemos seguido para llegar hasta aquí. Sumamos 1 a las visitas de todos los nodos hasta el nodo raíz y actualizamos el valor de esos nodos (en mi código se hace +1 si hemos ganado, -1 si hemos perdido y +0.5 si hemos empatado)

Cada una de las simulaciones que se realizan siguen la estructura de la Figura 6

esquema MCTS.JPG

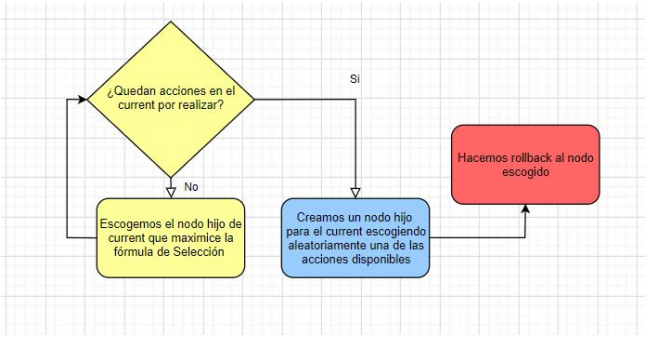

Fig. 6: Esquema de cada una de las simulaciones. Las fases amarillas corresponden a la Selección, la azul a la Expansión, la roja a la Simulación/Rollout y la verde a la Backpropagation

En mi caso, como he comentado antes, la fase de Expansión es algo diferente, y la definición de nodo hoja para la fase de Selección también. Este es el esquema que siguen esas fases (Fig 7)

esquema MCTS.JPG

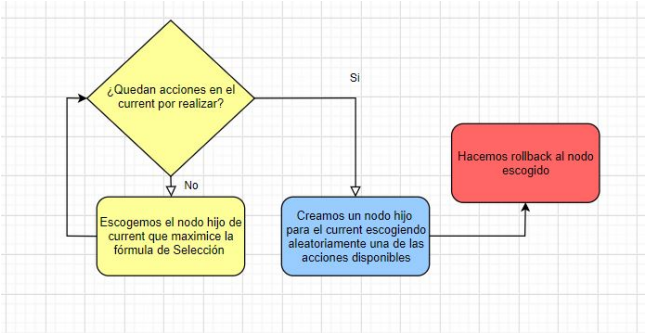

Fig. 7: Solo damos un nodo por explorado si no le quedan movimientos por hacer. Cuando expandimos solo lo hacemos con uno de los movimientos disponibles al azar en vez de todos a la vez.

Esta es una implementación perfectamente válida del MCTS y es la que he escogido realizar por el hecho de que si se expanden todos a la vez al intentar aplicar la fórmula de Selección se intenta calcular la media entre victorias y visitas, y para los hijos no escogidos eso es 0/0, lo cual introduce dificultades en el código. Así nos aseguramos de que todos los nodos a los que les aplicamos la fórmula han sido visitados al menos una vez.

5.3.2. Fórmula de selección

Como comentaba antes, una de las partes más importantes del algoritmo es su fórmula de Selección. Dado que trabajamos con un número de simulaciones limitado y no podemos explorar todas las opciones es evidente que es aquí donde tenemos que realizar el trabajo de cribar y decidir qué ramas son las más prometedoras. En el caso de MCTS, al igual que en los algoritmos de RL, lo más importante es encontrar un equilibrio entre explotación (profundizar en cada una de las ramas) y exploración (investigar varias ramas). Es decir, ni bajar demasiado ni expandirnos demasiado. Por ese motivo la fórmula principal que se usa para este algoritmo (y la que he implementado), UCB1, está compuesta por 2 términos, cada uno de los cuales representa estas dos partes y se complementan. La fórmula es la siguiente:

$$w_i/v_i + 2 * \sqrt{(\log V/v_i)} \quad (1)$$

Donde:

- wi corresponde al valor del nodo actual. Es decir, el número de veces que ha ganado, perdido o empatado en todas sus simulaciones
- vi corresponde al número de veces que hemos visitado el nodo actual
- V corresponde al número de veces que ha sido visitado el nodo padre del actual
- La primera parte de la función representa el valor a explotar, cuán bueno es un estado, mientras que la segunda representa el valor a explorar, donde cuanto más hayamos visitado un nodo menor será el valor.

5.4. AlphaZero

El algoritmo AphaZero utiliza técnicas de Reinforcement Leraning (RL) para entrenar un jugador. El entrenamiento se hace combinadamente con un jugador de MCTS. Primero explicaremos las nociones básicas del RL.

5.4.1. Reinforcement Learning

El aprendizaje por refuerzo (RL) es un área del Machine Learning [12] cuya idea base es asignar a las diferentes acciones sobre el entorno que puede realizar un agente una recompensa (Fig 8) y utilizar las anteriores iteraciones del algoritmo para retroalimentar ese valor, de forma que cada ejecución le permita aprender. Esta es su representación principal:

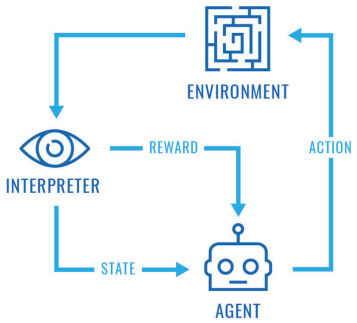

Fig. 8: Esquema general del aprendizaje por refuerzo

Lo más interesante de estos algoritmos, y una de las principales particularidades de, por ejemplo, AZ, es que no requieren explicación sobre las normas del juego (enfocándonos ya en el caso particular que se trata en este proyecto), si no que simplemente con el estímulo de perder/ganar es capaz de aprender a jugar y a identificar buenas situaciones y malas situaciones, lo que permite que venzan a expertos con años de trayectoria simplemente conociendo las normas básicas del juego.

5.4.2. Algoritmo

Pero, ¿qué es exactamente AlphaZero? El código al que llamamos AlphaZero, que se ha utilizado a lo largo de este proyecto no es más que un código que implementa lo necesario para entrenar una red neuronal. Por clarificarlo mejor, AlphaZero consta de 2 elementos principales y 3 fases de ejecución. Todo esto se puede consultar en más profundidad en su Cheat Sheet, publicado por Google DeepMind [13], pero a continuación daré una breve pincelada a sus elementos más importantes.
Como comentaba, AZ consta principalmente de 2 elementos:

- Red Neuronal: La red que va aprendiendo conforme se ejecuta el código y que será el resultado final de la ejecución. recibe como entrada un tablero y devuelve como salida el porcentaje de victoria que tiene ese tablero en concreto.
- Algoritmo MCTS: Este es el algoritmo que se utilizará cada vez que ejecutemos una partida.

Pero, ¿cómo se comunican ambos elementos? Simple, AZ utiliza una modificación de la fórmula de Selección del MCTS que incluye la salida de la red neuronal, lo cual hace que cada vez que se juega una partida la selección de los movimientos más óptimos se vea influida por el estado de entrenamiento de la red actual. Más adelante veremos la fórmula en profundidad.
En cuanto a las 3 fases, se refieren al proceso de entrenamiento que forma el bucle de ejecución principal: creación de un dataset de entrenamiendo simulando partidas, entrenamiento de la red, evaluación de la red actual. La creación del dataset de entrenamiento se realiza una sola vez al inicio de la ejecución, tras realizar todas las partidas que podamos para usarlas como training set pasamos al bucle entrenamiento-evaluación, que repetiremos hasta que estemos satisfechos con el resultado. El bucle principal del algoritmo queda así, pues:

5.4.3. Modificación MCTS

La fórmula de selección que utiliza AZ para aplicar el MCTS es la siguiente:

$$Q + U \tag{2}$$

Donde:

- Q es la media del nodo actual (victorias / visitas, igual que en el MCTS base)
- U corresponde a la siguiente fórmula:

$$c_{puct} * P(s,a) * \frac{\sqrt{\sum_b N(s,b)}}{1 + N(s,a)} \tag{3}$$

En esa fórmula, Q representa el potencial de explotación, nos dice cómo de buena es una opción; mientras que U representa el valor de exploración, teniendo en cuenta el potencial del resto de ramas. En la fórmula para calcular la U los prámetros son los siguientes:

- cpuct: Es un hiperparámetro propio de AZ que indica el porcentaje de exploración que vamos a tener. Es uno de los prámetros que he variado durante las pruebas.
- P(s,a): Es la probabilidad de victoria que calcula la red neuronal a este tablero en concreto
- N(s,X): Es el número de veces que hemos visitado un nodo. En el caso de (s, a) es el nodo actual, en el caso de el sumatorio de (s, b) corresponde a la suma de las visitas realizadas al resto de nodos del árbol.

6 RESULTADOS

Tras el tiempo que ha durado el proyecto finalmente se ha podido implementar completamente el algoritmo Minimax, con la optimización Alfabeta y dos heurísticas; el algoritmo MCTS y la incorporación de una red neuronal entrenada con el algoritmo AlphaZero para utilizar la modificación dle algoritmo MCTS.

Tal y como se propuso en los objetivos del proyecto, se han hecho enfrentamientos entre estos 3 algoritmos (el Minimax se descartó a favor de Minimax con Alfabeta debido a problemas de tiempo demostrados en el apartado 5.2.3 1) con las siguientes configuraciones:

- MM AB Heurística 1: Modificando la profundidad
- MM AB Heurística 2: Modificando la profundidad
- MCTS con la fórmula UCB1: Modificando el número de simulaciones
- MCTS con la red neuronal: Modificando el número de simulaciones y el parámetro cpuct

Como se comentó en el apartado Minimax, se han aplicado 2 mejoras a las heurísticas, una de ellas a ambas y la otra solo a la H1. Se ha considerado que ambas heurísticas eran lo bastante diferentes entre si como para hacerlas competir contra MCTS como si fueran dos algoritmos distintos, sin embargo las optimizaciones eran tan superiores que sustituyeron a las anteriores heurísticas. En este apartado se mostrarán los resultados extraídos tras las distintas pruebas en forma de tabla, además de las conclusiones que se extraen de ellos.

6.1. Comparación de las Heurísticas

En este apartado se medirá la eficacia de las 2 heurísticas y se estudiará cuál es la configuración de parámetros óptima a fin de enfrentarlas a los otros algoritmos.

6.1.1. Heurística 1

Como se explicó en el apartado de heurísticas, la primera heurística estaba basada en otorgar un valor (v1) a las parejas presentes en el tablero y otro (v2) a los tríos. Si ese conjunto de fichas era de un jugador, se sumaba el valor a la heurística. En caso contrario, se restaba. Inicialmente los valores establecidos fueron 10 puntos para 2 fichas juntas, 100 para 3 y 10000 para 4. Esto quiere decir que 9 parejas son menos valiosas que 1 trío, por lo que se realizaron algunas pruebas variando esta proporción para ver cuál era la mejor configuración:

% de victoria según param (v1/v2)		Profundidad			
		2	4	8	10
Valores probados	5/11	100	100	100	100
	5/21	100	100	100	100
	5/41	100	100	0	0
	5/71	100	100	0	0
	5/91	100	100	0	0

TAULA 2: SELECCIÓN DE PARÁMETROS DE LA HEURÍSTICA 1. EJECUCIÓN CONTRA V1 = 10 Y V2 = 100

Como se puede comprobar 2 a partir de la equivalencia 8 parejas son menos 1 trío (esto quiere decir v1 = 5 y v2 = 41) empezamos a ver como para profundidades altas es más efectivo dar más valor a los tríos, mientras que para profundidades bajas lo más efectivo es disminuir la diferencia pareja-trío. Como con valores distintos no se han detectado diferentes rendimientos **estos son los valores de v1 y v2 que se utilizarán para H1** a partir de ahora.

Una vez hecha esta prueba se añadió la mejora de dar más valor a las columnas centrales, aunque esa desde el principio dió mejores resultados en cualquier configuración por lo que no se han incluido las pruebas y se ha introducido directamente a la versión actual.

6.1.2. Heurística 2

En este caso la única mejora disponible era la de dar más valor a las columnas centrales, y al igual que en el caso anterior, dió mejores resultados desde el principio y en cualquier configuración, por lo que pasó a ser la versión definitiva.

6.2. Pruebas Minimax

Una vez acabada la afinación de las heurísticas quise comprobar la efectividad de ambas, así que las enfrenté contra el jugador random y contra ellas mismas.

6.2.1. H1 vs Random

Para esta partida, y teniendo en cuenta los datos extraídos de la tabla 2, se utilizaron como valores v1 y v2 5 y 41 respectivamente

% vict. de H1	Profundidad				
	2	4	6	8	10
Media	39.9	76.43	71.53	80.38	78.45

TAULA 3: MEDIA DE VICTORIAS DE H1 DE 151 PARTIDAS JUGADAS CONTRA EL JUGADOR RANDOM.

6.2.2. H2 vs Random

Para esta prueba se eliminó la opción profundidad 10, ya que con esta heurística tarda demasiado en ejecutarse una simple partida (+2 minutos), lo que hace que tenga que rebajar demasiado el número de partidas por prueba para sacar resultados concluyentes.

% vict. de H2	Profundidad			
	2	4	6	8
Media	84.3	85.3	90.2	85.27

TAULA 4: MEDIA DE VICTORIAS DE H2 DE 151 PARTIDAS JUGADAS CONTRA EL JUGADOR RANDOM.

6.2.3. H1 vs H2

En las pruebas anteriores (3 4) se realizaron 151 partidas para cada enfrentamiento, y en todas se colocó al algoritmo MMAB como segundo jugador, ya que se ha comprobado que es la posición más perjudicada y la menos propensa a ganar. Para la comparativa entre ambas heurísticas se realizaron sólo 2 ejecuciones, de 51 partidas, donde se alternaba constantemente qué heurística jugaba como jugador 2, para que hubiese igualdad de condiciones.
Finalmente esta prueba resultó ser menos esclarecedora de los esperado, ya que ambos algoritmos se comportaron exactamente igual: el Jugador1 ganaba siempre, independientemente de la profundidad o la versión de la heurística que utilizásemos.

6.2.4. Conclusiones

De las pruebas que se han realizado se han extraido dos conclusiones principales. La primera, más simple, es que parece ser que **H2 es la más efectiva de las dos**, ya que pese a empatar con H1 en la prueba de enfrentamiento tiene más porcentaje medio de victorias y, además, más estabilidad. Sin embargo la segunda conclusión es la más enriquecedora, y es que en las tablas de resultados se puede comprobar cómo **hay una profundidad a partir de la cuál el rendimiento del algoritmo decae**. Esto es algo chocante, ya que asumía que cuanto más profunda fuera la búsqueda mejor rendimiento tendría y que no se hacía de mucha profundidad por recursos. Sin embargo, esto parece ser un fenómeno común conocido como **Game-tree search pathology** [14] [15], que indica justo eso, que en la mayoría de juegos hay un punto de profundidad que una vez superado comienza a proporcionar peores resultados.

6.3. MCTS

En este apartado se medirá la eficacia del algoritmo MCTS contra las heurísticas en su mejor profundidad (6). En un principio se iba a medir antes, al igual que el MMAB, contra el random, pero este algoritmo vencía en todas las partidas al random a no ser que se descendiera el número de simulaciones a un valor irrisorio (por debajo de 100).
El resultado de las ejecuciones del MCTS variando su único parámetro (el número de simulaciones) contra el algoritmo MMAB H2 es el siguiente:

%Victorias MCTS	Parámetro Núm. Simulaciones			
	1000	2000	3000	4000
Random	100	100	100	100
H1	100	97	97	100
H2	100	81.4	90.3	100
Media	100	92.8	95.7	100

TAULA 5: PORCENTAGE DE VICTORIAS DEL JUGADOR MCTS VERSUS DIVERSOS JUGADORES PARA DIFERENTES VALORES EN EL NÚMERO DE SIMULACIONES (PROMEDIO SOBRE 31 PARTIDAS JUGADAS)

Tras obtener estos resultados, se realizó una breve prueba con el parámetro simulaciones = 5000 para tratar de averiguar si a partir de 2000 empeoraba y lo de los 4000 era un caso aislado o si el algoritmo dibujaba esa extraña curva de manera natural, y lo que se comprobó fue que, efectivamente, con 5000 y con 6000 también el porcentaje de victoria es del 100, por lo que queda por averiguar a qué se debe ese descenso es dos de los valores probados.
Con eso en menté se reprodujo el mismo experimento con los valores de simulación menores por si habían sido casos aislados, y los resultados obtenidos fueron los siguientes:

% Victorias MCTS	Parámetro Num. Simulaciones		
	1000	2000	3000
Media	98.4	93.3	100

TAULA 6: EJECUCIONES COMPLEMENTARIAS A LA TABLA 5 (PROMEDIO SOBRE 51 PARTIDAS JUGADAS)

Lo cual demuestra que el algoritmo ejecutado con 1000 simulaciones es más propenso a funcionar mejor que con 2000, aunque a partir de ahí sigue la progresión de forma natural. No se ha encontrado una explicación parecida a la encontrada con el caso de la game-tree search pathology, por lo que queda de nuevo como punto a investigar.

6.4. AlphaZero

Finalmente se realizaron las pruebas sobre el algoritmo AZ, con el que se modifica el parámetro cpuct. Al igual que con el resto de jugadores, las primeras pruebas fueron contra el jugador Random, pero independientemente del parámetro escogido vencía en todas las ejecuciones, por lo que no se adjunta esa gráfica. La evaluación del comportamiento de AZ se ha hecho solamente contra el mejor jugador de los analizados, que el el MCTS de 4000 simulaciones. Se han evaluado diferentes versions de AZ variando el parámetro cpuct. Asimismo se ha analizado el tiempo de respuesta promedio para cada caso. Todos los resultados se muestran en la tabla 7.

Ejecución 31 partidas		$\%victorias$ AZ	t_{medio}	t_{max}	t_{min}
cpuct	1	39.1	300	400	100
	0.8	28.57	260	360	108
	0.5	23.8	210	360	100
	0.2	10.5	106	300	90

TAULA 7: PORCENTAJE DE VICTORIAS DE AZ VERSUS MCTS(4000 SIM.) VARIANDO PARÁMETRO CPUCT DE AZ. LAS COLUMNAS A LA DERECHA INDICAN TIEMPOS PROMEDIOS SOBRE 31 PARTIDAS.

Como se puede observar, los resultados en rendimiento son bastante decepcionantes, ya que ninguna de las ejecuciones ha logrado empatar siquiera al MCTS puro, lo cual demuestra que esta red neecsitaba más entrenamiento o al menos más afinamiento. Sin embargo, se pueden observar algunas cosas interesantes en esta gráfica:

- Para empezar se ha incluido en la gráfica el tiempo medio dado que quería comprobar el efecto del parámetro cpuct sobre el tiempo de ejecución. Como se explicó anteriormente, cpuct es el parámetro que indica qué peso tendrá la exploración frente a la explotación. Por lo tanto, cuanto más pequeño sea este valor más nos vamos a centrar en realizar simulaciones sobre nodos conocidos y menos en abrir nuevos nodos. Con estas pruebas se ha demostrado que disminuirlo no solo tiene un efecto claro sobre el rendimiento (cuanta menos exploración menos victorias) si no también sobre el tiempo, ya que ha disminuido en consonancia.
- Por otra parte, se han incluido también los valores de tiempo mínimos y máximos porque examinando los resultados apareció algo muy curioso: todas las victorias del algoritmo AZ han tenido un tiempo de ejecución inferior a la media, y el valor mínimo de todas las pruebas ha pertenecido siempre a una victoria de AZ, lo cual parece indicar que este algoritmo está jugando de manera más ”limpiaz menos a ensayo y error que el MCTS puro, es decir, de alguna forma ha aprendido a hacer ciertas jugadas o a adelantarse a ciertos movimientos, lo que hace que sus partidas acaben antes. Esto será interesante explorarlo con una versión más entrenada de la red, para ver si es coincidencia o es realmente una consecuencia.

Además, también se ha enfrentado este algoritmo contra las dos heurísticas, por si era más efectivo contra una que contra la otra, y alternando el jugador 1 por si marcaba una diferencia demasiado grande, y los resultados han sido los que se muestran en la Tabla 8, donde se ha ejecutado el algoritmo AZ con el parámetro cpuct = 1 contra la mejor versión de cada heurística.

Ejecución		$\%vict.$ AZ	t_{medio}	t_{max}	t_{min}
H1	J1	100	120	140	84
	J2	92.5	176	387	116
H2	J1	76.2	130	173	78
	J2	62.5	115	123	98

TAULA 8: PORCENTAJE DE VICTORIAS DE AZ VERSUS LAS DOS HEURÍSTICAS IMPLEMENTADAS. LAS COLUMNAS A LA DERECHA INDICAN TIMPOS PROMEDIOS SOBRE 51 PARTIDAS

De nuevo son unos resultados algo más decepcionantes de lo esperado, aunque habiendo demostrado anteriormente que el MCTS base es superior a este AZ era de esperar. Sin embargo, de nuevo han habido algunos detalles a destacar:

- El hecho de que parece ser que AZ actua mejor jugando como J2 cuando las pruebas con el resto de algoritmos (inlcuido el MCTS puro) parecían decir que el jugador 1 tiene, por lo general, ventaja.

- En este caso los tiempos de ejecucion son menores y mucho más estables.
- En este caso no se ha podido percibir ningún patrón en los tiempos de ejecución, ambos algoritmos han tenido aproximadamente los mismos tiempos y ambos han sido más rápidos y más lentos que el otro en algunas ejecuciones.
- Se demuestra de nuevo que la heurística 1 es peor que la 2, ya que la diferencia en victorias es mucho mayor.
- Ese incremento de tiempo medio en la ejecución contra H1 como J2 se debe a las 2 partidas que ganó H1, que subieron la media al ser ejecuciones de más de 300 segundos.

7 Conclusiones

Desde el principio he distinguido dos facetas diferentes en este proyecto, la investigación y la implementación. Estoy muy satisfecho en cuanto al trabajo de investigación que he realizado en este proyecto, ya que he sido capaz de entender varios algoritmos distintos relacionados con el campo del Reinforcement Learning y gracias a eso he adquirido una base importante en ese campo que me permitirá seguir trabajando en él. Sin embargo, en cuanto a la implementación la verdad es que me gustaría haber podido hacer más, ya que además de AZ y MCTS he estudiado 2 algoritmos, QLearning y Temporal Difference Learning, que no he sido capaz de adjuntar en mi proyecto. También me habría gustado trabajar más en el entrenamiento de la red neuronal, aunque eso es algo que trabajaré en el futuro ya que para obtener resultados concluyentes en ese aspecto tendría que haber dedicado gran parte de este proyecto únicamente a eso cuando no era uno de los objetivos principales. En resumen, estoy orgulloso del trabajo que he realizado y impaciente por utilizarla como base para desarrollar e implementar algoritmos similares (como el MuZero que salió hace unos meses que amplía las posibilidades de este campo a niveles inimaginables).
Además, también me gustaría mencionar que me he llevado una grata sorpresa al realizar la investigación ya que este es un campo realmente vivo y no me ha sido difícil encontrar explicaciones de los distintos algoritmos o ejemplos de ejecución. Incluso he recibido bastantes respuestas en algunas preguntas que he realizado en artículos o foros relacionados. Creo que la conclusión más importante que tengo tras realizar este proyecto es que estoy deseando seguir profundizando e investigando en este campo y bastante ilusionado con las cosas que se pueden llegar a conseguir.

Agradecimientos

Por mi parte me gustaría agradecer principalmente a mi familia, especialmente mi madre, que han entendido la importancia que este proyecto tenía para mi y me han apoyado y animado a seguir en todo momento.
También a mis compañeros y amigos, tanto de la universidad como de fuera de ella, que han sabido ver cuando necesitaba un respiro y me han apoyado y cuidado de mi durante el desarrollo. Gracias por estar ahí cuando lo neecsitaba y cuando creía que no lo necesitaba
Y agradecer, por supuesto, a la mayoría de los profesores de la carrera de Ingeniería Informática, especialmente a los de la mención de Computación y especialmente a los de las asignaturas Conocimiento Raciocinio e Incertidumbre, Inteligencia Artificial y Aprendizaje Computacional por presentarme los conceptos y las herramientas que me han llevado a escoger este tema y encontrar mi verdadera vocación.
Dentro de esta categoría me gustaría destacar a mi tutora y profesora de IA, Maria Vanrell, quien me ha animado y motivado en todas las reuniones que hemos tenido haciéndome ver las enormes posibilidades de este campo.
Finalmente, pero no por ello menos importante, a las personas detrás de los artículos adjuntados en la bibliografía, especialmente a los autores de *Towards Data Science* por esas expliaciones sobre los distintos algoritmos que me han ayudado a comprender más los entresijos de las herramientas que espero utilizar en el futuro.
Gracias a todos y a todas por todo, espero que este sea solo el inicio de una larga carrera en este campo y espero hacer honor a todos los conocimientos que he adquirido durante el desarrollo de este proyecto para hacer del mundo un lugar mejor.

Referencias

[1] Second Nature Academy. 8 ways board games teach like skills, 2016.

[2] Miriam Llorens Monzó. Sistematizacion de un juego de mesa para la rehabilitaciÓn cognitiva de pacientes con daÑo cerebral adquirido. 2016.

[3] DeepMind Blog. Alphago china, 2017.

[4] DeepMind Blog. Alphago zero: Starting from scratch, 2017.

[5] Mike Klein. Google's alphazero destroys stockfish in 100-game match, 2017.

[6] Ivan Sanchez Resina. Tableros utilizados durante el desarrollo del proyecto, 2019.

[7] Victor Allis. A knowledge-based approach of connect-four. 1988.

[8] Wikipedia. Minimax, 2019.

[9] Marissa Eppes. Game theory — the minimax algorithm explained, 2019.

[10] Wikipedia. Alpha-beta pruning, 2019.

[11] JavaTpoint. Alpha-beta pruning, 2018.

[12] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[13] David Foster. Alphazero cheat sheet, 2017.

[14] Brandon Wilson Dada Nau, Austin Parker. Error minimizing minimax: Avoiding search pathology in game trees. 2009.

[15] David Mutchler. The multi-player version of minimax displays game-tree pathology. *Artificial Intelligence*, 64(2):323–336, 1993.