Aprendiendo a jugar a Otelo

1st Manuel Jesús Sánchez García
Ingeniería Informática, US
Ingeniería del Software
Sevilla, España
mansangar13@alum.us.es

2nd Antonio Montero López
Ingeniería Informática, US
Ingeniería del Software
Sevilla, España
antmonlop4@alum.us.es

***Abstract*—Este trabajo es el desarrollo de un agente inteligente para el juego "Otelo" con inteligencia artificial, con el objetivo de construir jugadores capaces de tomar decisiones estratégicas mediante la combinación de algoritmos de búsqueda y redes neuronales. Se utiliza el algoritmo de búsqueda de Monte Carlo Tree Search (MCTS) con el enfoque de Upper Confidence Bound Tree (UCT) y una red neuronal para estimar el valor de las posiciones, guiando así la expansión y selección de nodos dentro del árbol de búsqueda.**

I. Introducción

Otelo es un juego de estrategia de dos jugadores. Se juega en un tablero 8x8, es decir, 64 casillas. Esto da una gran cantidad de posiciones posibles, por lo que resulta un gran desafío estudiar cada una de ellas, lo que es perfecto para una inteligencia artificial.

En este proyecto, abordamos este problema creando un agente inteligente combinando dos técnicas principalmente: Monte Carlo Tree Search y una red neuronal. MTCS resulta muy efectivo para identificar los posibles movimientos a partir de una posición específica y la red neuronal permite aprender los patrones del juego de partidas anteriores e identificar cuál será el mejor movimiento a partir de esa posición determinada.

Hemos utilizado Monte Carlo Tree Search para generación de datos de entrenamiento, y posteriormente hemos utilizado una red neuronal para evaluar estas posiciones del tablero.

II. Preliminares

A. Reglas del Juego Otelo

Como hemos comentado anteriormente, Otelo se juega en un tablero 8x8 y el objetivo es tener la mayoría de fichas de tu color en el tablero al finalizar la partida. Al comienzo del juego, el tablero se dispone con cuatro fichas en el centro: dos fichas blancas en posiciones opuestas y dos fichas negras en las otras dos posiciones centrales. El jugador con fichas negras mueve primero.

En su turno, un jugador puede realizar diferentes movimientos con estas reglas:

- Colocar una ficha de su color sobre una casilla vacía de manera que, en línea recta (horizontal, vertical o diagonal), una o más fichas del oponente queden atrapadas entre la ficha que se coloca y otra ficha del propio jugador.
- Todas las fichas del oponente que queden atrapadas de esta manera se voltean y pasan a ser del color del jugador que hizo la jugada.

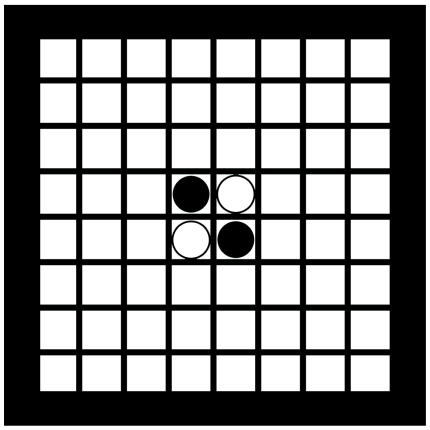

Fig. 1. Estado inicial de una partida.

- Si un jugador no puede realizar un movimiento válido, debe pasar su turno.
- Si ambos jugadores no pueden realizar movimientos válidos, la partida finaliza.

El juego puede terminar de dos formas: o bien todas las casillas del tablero están ocupadas o ninguno de los dos jugadores puede realizar un movimiento válido. En cualquier caso, el ganador es el jugador que tenga más fichas de su color en el tablero al final de la partida.

B. Algoritmo Monte Carlo Tree Search (MCTS)

Monte Carlo Tree Search sirve para estimar la utilidad de un estado aplicando una política. Es un algoritmo de búsqueda adversaria que realiza una exploración aleatoria, desde un estado inicial, de las distintas posiciones posibles, junto con una explotación de esos movimientos más beneficiosos para el jugador que realice la acción. Tiene cuatro fases principales:

- **Selección:** Desde un nodo raíz, se seleccionan los nodos hijos, según una política de selección, hasta llegar a un nodo que no esté completamente extendido.
- **Expansión:** Se añade un nodo hijo al nodo seleccionado.

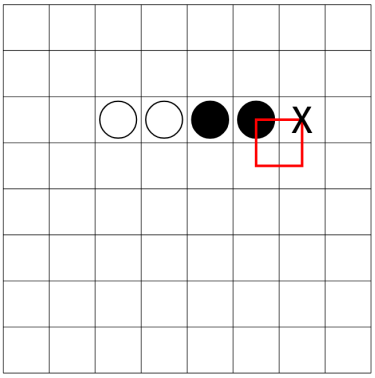

Fig. 2. Captura horizontal: Colocando una ficha en la casilla marcada (X), las fichas negras quedan atrapadas entre fichas blancas y son volteadas.

- **Simulación:** Se simulan las diferentes posiciones desde el nuevo nodo hasta un nodo terminal.
- **Retropropagación:** Se propaga el resultado de la simulación hacia atrás y se actualizan los datos de los nodos visitados.

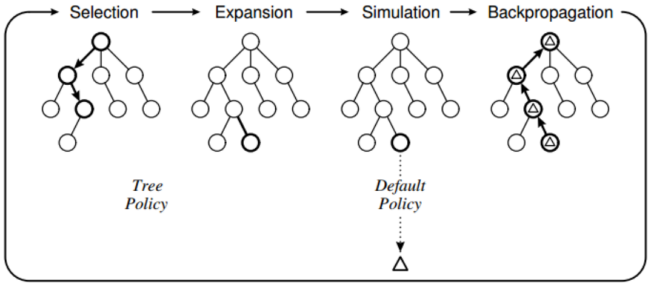

Fig. 3. Selección, expansión, simulación y retropropagación.

La política de selección usada es UCT (Upper Confidence Bound applied to Trees), con ella se realiza la exploración y la explotación con la fórmula:

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

donde $N(s_0)$ es el número de veces que el nodo s_0 (el padre de s) ha sido visitado, $N(s)$ es el número de veces que el nodo hijo, s, ha sido visitado, $Q(s)$ es la recompensa total de todas las jugadas que pasan a través del nodo s, y C_p mayor que 0 es constante.

C. Redes Neuronales para evaluación de posiciones

Una red neuronal con alimentación hacia adelante es un modelo de regresión para aprendizaje supervisado a partir de atributos numéricos. Las redes neuronales se inspiran en la estructura y funcionamiento del cerebro. De esta forma, una neurona puede enviar información (en forma de señales eléctricas) a otras neuronas a las que esté conectada. Estas neuronas, a su vez, propagarán esa información a otras neuronas,

Fig. 4. Explotación y exploración de las posibilidades.

y así sucesivamente. Esto permite aprender patrones muy complejos a partir de datos, en lo que nos respecta, evaluar las posiciones en tiempo real y predecir las probabilidades de victoria desde una posición específica.

En este trabajo, utilizamos una red neuronal con neuronas de entrada por cada posición del tablero, el turno actual y una neurona de salida que nos dará la recompensa de un movimiento en el tablero. Esta red no solo tiene una capa de entrada y de salida, sino que además tiene varias capas ocultas que ayudan en la evaluación de la posición.

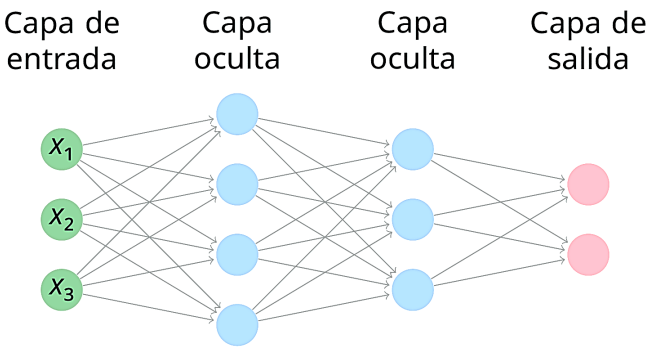

Fig. 5. Ejemplo de capas de entrada, ocultas y salida de una red neuronal.

III. Implementación

A. Estructura del proyecto

El proyecto está organizado en varios módulos:

- **otelo.py:** Incluye las reglas del juego y la lógica del tablero.
- **mcts.py:** Contiene el algoritmo Monte Carlo Tree Search con UCT y la función simula.
- **ia_vs_ia.py:** Genera datos de entrenamiento mediante partidas utilizando el algoritmo de MCTS.
- **modelo.py:** Incluye el código de la red neuronal que se entrena para evaluación de posiciones.
- **main.py:** Contiene la interfaz para jugar contra la IA.

B. Implemnetación de Otelo

Primero que nada, escogemos para crear el tablero una estructura adecuada, como es un diccionario, el cual tendrá como claves las posiciones de cada casilla del tablero y como

valor 0 si no tiene ficha, 1 si tiene ficha negra y 2 si tiene ficha blanca.

Después, al comprobar si un movimiento es válido, lo que hacemos es recorrer todas las direcciones posibles (arriba, abajo, derecha, izquierda y en diagonal). Para cada dirección, miramos si hay fichas del oponente justo al lado de la posición en la que queremos poner la nuestra. Si encontramos al menos una, seguimos avanzando en esa dirección. Si después de esas fichas del rival hay una ficha nuestra, entonces el movimiento es válido y guardamos las fichas del rival que se deberían voltear.

Si el movimiento es válido, se añade al tablero y se cambian (o voltean) todas las fichas del rival que hayan quedado atrapadas entre la nueva ficha y otra del mismo color.

También hemos creado una función que calcula todos los movimientos posibles que puede hacer un jugador en ese turno. Esto es útil para dar más información mientras se desarrolla el juego y, además, para darle una pequeña ayuda al jugador humano.

Para mostrar el tablero por pantalla usamos letras para representar las fichas: un punto (.) si no hay nada, una N si hay una ficha negra y una B si hay una blanca.

Al final, tenemos una función que cuenta las fichas de cada jugador y dice quién ha ganado. Si hay más fichas blancas, gana el jugador blanco; si hay más fichas negras, gana el jugador negro, y si hay el mismo número, se considera empate.

```python
def crear_tablero():
    res = dict()
    for i in range(0, 8):
        for j in range(0, 8):
            res[(i,j)] = 0

    res[(3, 3)] = 1
    res[(4, 4)] = 1
    res[(3, 4)] = 2
    res[(4, 3)] = 2

    return res
```

Fig. 6. Código función crear_tablero(): Se establece el estado inicial del tablero.

C. *Implementación del algoritmo MCTS con UCT*

La base del agente inteligente que hemos creado es el algoritmo **Monte Carlo Tree Search (MCTS)**, usando como política de selección el método **UCT (Upper Confidence Bound applied to Trees)**. El objetivo es explorar posibles jugadas desde una posición actual y tomar la mejor decisión posible.

Clase Nodo: Cada nodo del árbol representa un estado del tablero. Además del estado y el turno del jugador, también guarda quién es su padre, qué acción lo llevó hasta él, qué movimientos todavía puede hacer, cuántas veces se ha visitado y la recompensa acumulada. Las acciones posibles se barajan al azar para darle variedad a las partidas y evitar que el árbol siempre crezca igual.

```python
class Nodo:
    def __init__(self, tablero, turno, padre=None, accion = None):
        self.estado = tablero
        self.turno = turno
        self.padre = padre
        self.hijos = []
        acciones = list(otelo.posibles_movimientos(tablero, turno).items())
        random.shuffle(acciones)
        self.acciones_posibles = dict(acciones)
        self.acciones_hechas = []
        self.accion = accion
        self.r_acum = 0
        self.n_visitas = 0
```

Fig. 7. Clase `Nodo`: contiene toda la información necesaria de cada posición del juego.

Función `mcts_uct`: Es el punto de entrada al algoritmo. Parte del estado actual del tablero y realiza un número fijo de iteraciones. En cada iteración se selecciona un nodo del árbol, se expande (si tiene movimientos aún no usados), se simula cómo podría continuar la partida desde ese punto y finalmente se retropropaga el resultado hacia atrás, actualizando los nodos por los que ha pasado.

```python
def mcts_uct(tablero, turno, iteraciones=100):
    raiz = Nodo(tablero, turno)
    for i in range(0, iteraciones):
        nuevo_nodo = seleccion(raiz)
        res_simulacion = simula_red(nuevo_nodo.estado, nuevo_nodo.turno)
        retropropaga(nuevo_nodo, res_simulacion)

    return mejor_sucesor_uct(raiz).accion
```

Fig. 8. Función `mcts_uct()`: realiza varias simulaciones y devuelve la mejor acción.

Selección y expansión: La función selección recorre el árbol desde la raíz hasta que encuentra un nodo que aún no ha probado todos sus movimientos. Cuando encuentra uno, la función `expande` se encarga de generar un nuevo nodo a partir de uno de los movimientos que faltaban por explorar.

Simulación: Una vez expandido, se juega una partida completa desde ese nuevo nodo usando jugadas aleatorias (o con alguna estrategia simple). Cuando la partida acaba, se devuelve +1 si ganó el jugador del nodo inicial, -1 si perdió y 0 si fue empate. En una variante del agente, se puede usar una red neuronal en esta fase para que valore directamente la posición sin necesidad de simular hasta el final.

Retropropagación: La función `retropropaga` actualiza las estadísticas de todos los nodos que formaron parte del camino desde el nodo simulado hasta la raíz. Se suma la recompensa y se incrementa el contador de visitas.

Selección final: Después de todas las simulaciones, se elige como mejor movimiento aquel que tenga la mejor puntuación UCT.

```python
def mejor_sucesor_uct(nodo):
    c_p = 1.4142
    n_s0 = nodo.n_visitas
    uct_hijos = []
    posicion = 0
    for hijo in nodo.hijos:
        q_s = hijo.r_acum
        n_s = hijo.n_visitas
        uct = (q_s/n_s) + 2*c_p*math.sqrt(math.log(n_s0)/n_s)

        uct_hijos.append((posicion, uct))

    if not uct_hijos or len(uct_hijos) == 0:
        return nodo

    mejor_hijo = max(uct_hijos, key=lambda x:x[1])

    return nodo.hijos[mejor_hijo[0]]
```

Fig. 9. Función `mejor_sucesor_uct()`: selecciona el mejor nodo hijo según la fórmula UCT.

D. Generación de datos

Para entrenar la red neuronal, se ha automatizado la generación de un conjunto de datos mediante partidas entre dos instancias del agente MCTS. En cada turno de cada partida, se guarda el estado del tablero y el jugador activo en una lista temporal. Una vez finalizada la partida, se asigna a cada estado el resultado correspondiente desde el punto de vista del jugador que realizó la jugada: +1 si ganó, 0 si hubo empate y -1 si perdió.

Finalmente, todos los datos se almacenan en un fichero CSV, donde cada fila representa un estado del juego junto con su etiqueta de resultado. Esta estructura permite utilizar el conjunto de datos para un entrenamiento supervisado de la red neuronal.

E. Modelo de red neuronal

La red neuronal se ha implementado usando Keras. Se toma como entrada cada una de las posiciones del tablero (64 posiciones) y el turno actual; se evalúa la posición y se establece la salida.

La salida de la red neuronal varía entre -1 y 1, donde -1 es una victoria para las negras, 0 un empate, y 1 una victoria para las blancas. Esta representación es más eficiente que utilizar tres neuronas de salida con softmax, ya que sigue la naturaleza de un resultado con únicamente victorias de negras y blancas.

```python
def crear_fila(tablero, turno, ganador):
    fila = tablero.copy()
    fila["turno"] = turno
    fila["ganador"] = ganador
    return fila

for i in tqdm(range(partidas)):

    turno = 1

    tablero = otelo.crear_tablero()

    partida_actual = []

    while True:

        ...

        movimiento = mcts.mcts_uct(tablero, turno, iteraciones=15)

        otelo.poner_ficha(tablero, movimiento[0], movimiento[1], turno)

        ganador, blancas, negras = otelo.ganador(tablero)

        nueva_fila = crear_fila(tablero, turno, ganador)
        partida_actual.append(nueva_fila)

        turno = 3 - turno
```

Fig. 10. Código función **crear_fila()**: Se crean las filas del .csv necesario para el entrenamiento de la red neuronal.

```python
red_otelo = Sequential()
red_otelo.add(Input(shape=(65,)))
red_otelo.add(Dense(128, activation='relu'))
red_otelo.add(Dense(64, activation='relu'))
red_otelo.add(Dense(32, activation='relu'))
red_otelo.add(Dense(1, activation='tanh'))

optimizador = SGD(learning_rate=0.001)
red_otelo.compile(
    optimizer=optimizador,
    loss='mean_squared_error',
    metrics=['mean_absolute_error']
)
```

Fig. 11. Código red neuronal: Se establecen las capas del agente neuronal y su compilación.

La red neuronal que construimos tiene una estructura sencilla pero eficaz. Se usó un modelo secuencial con:

- Una capa de entrada de 65 neuronas (64 casillas + turno),
- Tres capas ocultas con 128, 64 y 32 neuronas respectivamente, todas con activación ReLU,
- Una capa de salida con una sola neurona y activación `tanh`, para obtener un valor entre -1 y 1.

IV. Pruebas y experimentación

Las primeras pruebas realizadas fueron para validar que la lógica del juego Otelo funcionaba correctamente. Se implementó un modo donde dos personas podían jugar entre sí por consola, y se comprobó que todas las jugadas posibles eran válidas, incluyendo las capturas en diferentes direcciones, los turnos, el paso de turno si no había movimientos disponibles, y la detección del final de partida.

Una vez validada la lógica básica, comenzamos con la implementación del algoritmo MCTS. Primero lo probamos enfrentando al agente contra un jugador humano, y se observó que era capaz de devolver jugadas razonables. Sin embargo, al hacer que dos agentes MCTS jugaran entre sí (a través del fichero `IA_vs_IA.py`), notamos que los resultados no variaban aunque aumentáramos el número de iteraciones: todas las partidas terminaban con el mismo resultado, concretamente 40-24 a favor del jugador con fichas negras.

Este comportamiento nos indicaba que algo no estaba funcionando bien en la aleatoriedad o en la evaluación de los movimientos. Investigando el problema, descubrimos que la lista de movimientos posibles se procesaba en el mismo orden cada vez, lo que provocaba que el árbol creciera siempre igual. Para solucionarlo, introdujimos una aleatorización en el orden de las acciones disponibles al crear los nodos. Además, revisamos la función de simulación y la forma en la que se propagaban las recompensas.

Tras estos cambios, el comportamiento del agente mejoró notablemente. Las partidas entre agentes MCTS empezaron a producir resultados más variados y realistas. El ratio de victorias entre negras y blancas comenzó a acercarse al 50%, con pequeñas variaciones razonables. También hicimos pruebas enfrentando al agente contra humanos, y comprobamos que ya era bastante difícil ganarle, especialmente con un número alto de iteraciones (5000).

Una vez validado que el algoritmo funcionaba correctamente, preparamos el sistema de generación de datos para entrenar la red neuronal. Para ello, el agente MCTS jugó un total de 150 partidas contra sí mismo con 5000 iteraciones por turno. En cada jugada se guardó el estado del tablero y el jugador que movía. Al finalizar la partida, se asignó a cada estado el resultado correspondiente (+1, 0 o -1) desde el punto de vista del jugador que realizó la jugada. Este proceso generó un archivo CSV con unas 9000 líneas, que utilizamos como dataset para el entrenamiento.

A. Generación de datos

Como se ha explicado, se utilizó el archivo `IA_vs_IA.py` para hacer que el agente jugara partidas automáticas contra sí mismo. Cada vez que se realizaba una jugada, se guardaba el tablero (como una lista de 64 posiciones más el turno) en una lista temporal. Al acabar la partida, cada jugada se etiquetaba con el resultado final desde el punto de vista del jugador que la hizo.

Finalmente, todos los datos se introdujeron en un archivo CSV. Este archivo contiene una fila por cada jugada, con 65 columnas (64 casillas + turno) y una columna adicional con la etiqueta del resultado.

B. Entrenamiento del modelo

Para entrenar la red neuronal, primero cargamos el archivo `partidas_ia_vs_ia.csv`, que contiene todas las jugadas realizadas por los agentes MCTS al enfrentarse entre sí. El fichero tiene una columna por cada casilla del tablero (de `(0, 0)` a `(7, 7)`), una columna adicional con el turno y una más con el ganador final de la partida.

Una vez cargados los datos, separamos las columnas de entrada (`atributos`) de la columna de salida (`ganador`). Para convertir el objetivo en algo numérico y continuo, codificamos el resultado así:

- Si ganaron las fichas negras (jugador 1), el valor es -1.0.
- Si ganaron las blancas (jugador 2), el valor es 1.0.
- Si hubo empate, el valor queda como 0.0.

Después dividimos los datos en entrenamiento (80%) y validación (20%) con `train_test_split`, asegurando que el entrenamiento y las pruebas fueran independientes.

Para entrenar usamos el optimizador `SGD` (descenso del gradiente estocástico) con una tasa de aprendizaje de 0.001. Como función de pérdida empleamos el error cuadrático medio (`mean_squared_error`), y como métrica adicional la media del error absoluto.

El entrenamiento se realizó durante 2000 épocas, con un tamaño de lote (`batch_size`) de 32. También fijamos una semilla aleatoria con `set_random_seed` para que los resultados fueran reproducibles. Al finalizar, se evaluó el modelo con el conjunto de validación y se guardó como `red_otelo.h5`, que luego se usa en el agente MCTS para evaluar los estados.

C. Resultados

Después de entrenar el modelo, integramos la red neuronal en el agente MCTS, reemplazando la función `simula()` por `simula_red()`, que simplemente evalúa un estado llamando al modelo.

Realizamos nuevas partidas con el agente que usaba red y comparamos los resultados con la versión anterior. En general, se notó una mejora, puesto que el agente con red neuronal ganaba bastantes más partidas, incluso con menos iteraciones que el algoritmo MCTS con una simulación aleatoria. En este caso, hablamos de más o menos en torno al 70-75% de las victorias que eran de las fichas blancas.

El principal problema es que el simula con red neuronal tardaba bastante más tiempo, lo que puede deberse a que la red neuronal no se ejecutaba mediante CPU en lugar de usar una GPU dedicada.

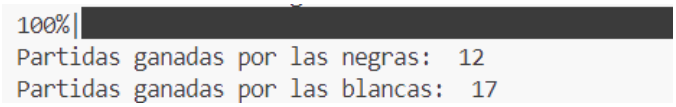

Fig. 12. Resultados de partidas entre agentes usando MCTS sin red neuronal con 1000 iteraciones.

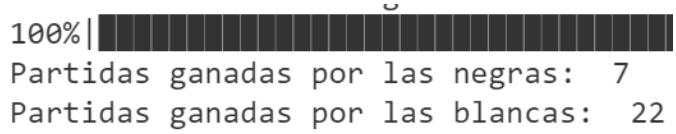

Fig. 13. Resultados de partidas entre un agente usando MCTS contra agente usando la red neuronal (MTCS 1000 iteraciones y red neuronal 500).

CONCLUSIONES

Este proyecto nos ha parecido muy interesante y, sobre todo, muy útil para aprender cosas nuevas. Al principio pensábamos que iba a ser bastante complicado, sobre todo por tener que usar algoritmos como Monte Carlo Tree Search (MCTS), que no conocíamos y que al principio nos costó bastante entender y poner en práctica.

Una de las partes más difíciles fue hacer que el agente funcionara bien. Al principio, las partidas entre IAs siempre terminaban igual, sin importar las iteraciones que pusieras. Esto nos obligó a revisar el código, meter aleatoriedad y ajustar cómo se daban las recompensas. Una vez solucionado, el comportamiento del agente mejoró mucho y las partidas empezaron a tener más variedad.

También fue muy interesante ver cómo se podía usar una red neuronal dentro del propio agente para tomar mejores decisiones. Aprendimos a generar datos, entrenar un modelo y usarlo directamente en el juego.

En general, ha sido un proyecto muy completo con el que hemos aprendido bastante, tanto de programación como de inteligencia artificial, y que además nos ha resultado muy entretenido de hacer.

USO DE LA IA

En este caso se utilizó ChatGPT después de terminar la implementación del juego para que comprobara los movimientos realizados en una partida humano vs humano, para asegurarnos de que la lógica del juego era correcta y poder continuar con la implementación del agente. Link de la conversación: https://chatgpt.com/share/684715cb-3a3c-8005-b3a4-ed825635a7dc

BIBLIOGRAFÍA

REFERENCES

[1] Russell, Stuart and Norvig, Peter. Artificial Intelligence: A Modern Approach (2010). Chapter 22.

[2] S. Russell and P. Norvig, Artificial Intelligence: A Modern Approach, 4th ed., Pearson, 2020, ch. 5, “Adversarial Search”.

[3] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton, “A Survey of Monte Carlo Tree Search Methods,” IEEE Transactions on Computational Intelligence and AI in Games, vol. 4, no. 1, pp. 1–43, 2012, doi: 10.1109/TCIAIG.2012.2186810.

[4] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis, “Mastering the game of Go with deep neural networks and tree search,” Nature, vol. 529, no. 7587, pp. 484–489, 2016, doi: 10.1038/nature16961.

[5] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, Y. Chen, T. Lillicrap, F. Hui, L. Sifre, G. van den Driessche, T. Graepel, and D. Hassabis, “Mastering the game of Go, Chess, and Shogi by self-play with a general reinforcement learning algorithm,” arXiv preprint, arXiv:1712.01815, 2017. arXiv:1712.01815.

[6] Fernando Sancho Caparrini, ”Algoritmo de Monte Carlo aplicado a Búsquedas en Espacios de Estados”, https://www.cs.us.es/ fsancho/Blog/posts/MCTS.md