UA

Conecta 6

Alexander Andonov Aracil
DNI: 74526740L

Daniel Asensi Roch
DNI: 48776120C

18 de noviembre de 2023

Índice

1. Introducción

La presente memoria expone el desarrollo y las decisiones tomadas durante la implementación de un motor de juego para el juego de estrategia Conecta 6. Este juego, aunque comparte similitudes con el clásico Go, introduce la variante desafiante de ganar alineando seis piedras de manera consecutiva, lo cual plantea interesantes desafíos desde la perspectiva de la inteligencia artificial y la programación de algoritmos de búsqueda adversaria.

1.1. Planteamiento del problema

El problema consiste en diseñar y programar un motor de juego capaz de competir en el juego Conecta 6 de manera eficiente y efectiva. Se busca que este motor no solo juegue siguiendo las reglas establecidas sino que también sea capaz de tomar decisiones inteligentes y estratégicas durante la partida. Esto implica la creación de un algoritmo de búsqueda adversaria que pueda manejar una amplia gama de posibles escenarios en el juego y responder a ellos en tiempo real, considerando las limitaciones de tiempo de procesamiento y recursos computacionales como memoria y CPU.

El juego Conecta 6 ofrece un espacio de búsqueda extenso y profundo, lo que hace impracticable la búsqueda exhaustiva sin algún tipo de heurística o poda. Además se requiere cuidadosa atención al diseño de una función de evaluación efectiva, la cual puede ser mejorada mediante técnicas de aprendizaje automático.

1.2. Objetivos

Los objetivos de esta práctica son múltiples y se detallan a continuación:

- **Desarrollo de un Algoritmo de Búsqueda Adversaria:** Implementar un algoritmo efectivo que permita al motor de juego decidir las mejores jugadas a realizar. Dicho algoritmo debe ser capaz de manejar la complejidad del juego y optimizar las decisiones bajo restricciones de tiempo.
- **Optimización del Motor de Juego:** Integrar optimizaciones al algoritmo de búsqueda para que el motor de juego opere dentro de los límites de uso de CPU y memoria establecidos, garantizando su viabilidad para el uso en tiempo real.
- **Diseño de una Función de Evaluación:** Crear una función de evaluación que permita al motor de juego valorar las posiciones en el tablero de manera que pueda prever el potencial de victoria, identificar amenazas y oportunidades y, por consiguiente, decidir las jugadas estratégicamente.
- **Implementación de Aprendizaje Automático:** Aplicar técnicas de aprendizaje automático para afinar la función de evaluación y mejorar la capacidad predictiva y estratégica del motor de juego.

- **Desarrollo de una Interfaz de Comunicación:** Desarrollar una interfaz que permita la comunicación entre el motor de juego y otros programas o usuarios, siguiendo los protocolos establecidos para la práctica.
- **Experimentación y Validación:** Realizar una serie de pruebas experimentales para evaluar y validar el rendimiento del motor de juego en términos de fuerza de juego y velocidad de procesamiento, ajustando las heurísticas y los parámetros de los algoritmos según los resultados obtenidos.

2. Estado del Arte

El desarrollo de soluciones basadas en inteligencia artificial para juegos ha experimentado avances significativos en las últimas décadas. Estos avances se han visto impulsados por el rápido desarrollo de la potencia computacional y los avances en algoritmos de aprendizaje automático. La evolución de estas tecnologías refleja no solo mejoras cuantitativas en términos de velocidad y eficiencia de procesamiento, sino también mejoras cualitativas en la capacidad de las máquinas para aprender, adaptarse y tomar decisiones. Este campo ha pasado de programas de juego simples basados en reglas fijas a sistemas de inteligencia artificial complejos capaces de aprender y mejorar de forma autónoma, abriendo nuevas fronteras en la investigación de la IA y sus aplicaciones prácticas.

2.1. Primeras Soluciones

La incursión inicial en la aplicación de inteligencia artificial en juegos se caracterizó por innovaciones pioneras, marcando el comienzo de una era de avances tecnológicos. Entre estos desarrollos iniciales, DeepBlue de IBM se destaca como un hito, demostrando por primera vez la capacidad de una máquina para superar a un campeón mundial en un juego de estrategia complejo.

2.1.1. DeepBlue

DeepBlue [1], una creación pionera de IBM, marcó un hito histórico en la inteligencia artificial aplicada a juegos. Este sistema fue diseñado específicamente para jugar al ajedrez a un nivel competitivo. En 1997, logró una victoria histórica contra el entonces campeón mundial de ajedrez, Garry Kasparov. Lo notable de DeepBlue era su capacidad para utilizar una combinación de fuerza bruta para analizar millones de posiciones por segundo, una evaluación heurística avanzada desarrollada con la ayuda de maestros de ajedrez, y una extensa base de datos de aperturas y finales de partidas. Su éxito no solo fue un hito en el ajedrez sino que también marcó un punto de inflexión en el estudio y las aplicaciones de la IA en juegos de estrategia, demostrando el potencial de la inteligencia artificial para competir y superar a los humanos en tareas complejas de toma de decisiones .

2.2. DeepMind

La evolución de la inteligencia artificial en juegos tomó un giro significativo con las contribuciones de DeepMind. Sus creaciones, AlphaGo y MuZero, no solo superaron las barreras existentes en juegos de alta complejidad sino que también redefinieron el potencial de aprendizaje y adaptabilidad de las máquinas, abriendo nuevas posibilidades en el campo de la IA.

2.2.1. AlphaGo

AlphaGo [2], desarrollado por DeepMind, representó un avance significativo en el campo de la IA aplicada a juegos de estrategia complejos. Su desempeño contra Lee Sedol, uno de los jugadores más destacados en el juego de Go, en 2016 fue un acontecimiento mundial. A diferencia de los juegos como el ajedrez, el Go tiene un número astronómicamente alto de posibilidades, lo que lo hace menos susceptible a enfoques basados en la fuerza bruta. AlphaGo combinó redes neuronales profundas con algoritmos de búsqueda Monte Carlo Tree Search (MCTS) para evaluar y seleccionar jugadas. Estas redes se entrenaron mediante el aprendizaje supervisado de partidas de Go profesionales y el aprendizaje por refuerzo a partir de juegos jugados contra sí mismo. Esta capacidad de aprendizaje y adaptación mostró una forma innovadora y efectiva de abordar juegos de alta complejidad, destacando la creciente sofisticación y potencial de la IA.

2.2.2. MuZero

MuZero [3], también desarrollado por DeepMind, llevó aún más lejos los avances logrados por AlphaGo (ver imagen 1). Mientras que sistemas como AlphaGo requerían un conocimiento previo de las reglas del juego, MuZero fue diseñado para aprender no solo las estrategias sino también las reglas del juego a través de su interacción con el entorno. Esta capacidad de aprender las reglas de forma autónoma representa un avance significativo, ya que permite a la IA abordar una gama aún más amplia de juegos y problemas, incluso aquellos con reglas complejas y dinámicas. Esto significaba que MuZero podía teóricamente aplicarse a cualquier tipo de juego, abriendo posibilidades para explorar nuevos terrenos en el campo de la inteligencia artificial aplicada a juegos y en otros dominios donde la comprensión y la adaptación a reglas dinámicas son cruciales.

2.3. Connect6

El Monte Carlo Tree Search (MCTS), según [4], ha sido una técnica revolucionaria en el juego de Connect6. MCTS es un algoritmo de búsqueda que se utiliza para tomar decisiones óptimas en un espacio de juego. En Connect6, un juego con un vasto y complejo espacio de posibilidades, MCTS ha permitido que las IA exploren y evalúen estrategias de manera eficiente, equilibrando entre la exploración de nuevos movimientos y la explotación de estrategias conocidas. Este enfoque ha demostrado ser particularmente efectivo en juegos donde la anticipación de movimientos del oponente y la planificación a largo plazo son cruciales.

2.3.1. Heurísticas en ConnectX

La investigación realizada por [5], expande el uso de heurísticas en variantes de Connect, incluyendo Connect6. Las heurísticas, en este contexto, se refieren a técnicas que permiten una evaluación rápida y eficiente de las posiciones y movimientos posibles en el tablero. En un juego como Connect6, donde el número de configuraciones de tablero es extremadamente alto, estas heurísticas se vuelven fundamentales. Permiten a la IA descartar opciones me-

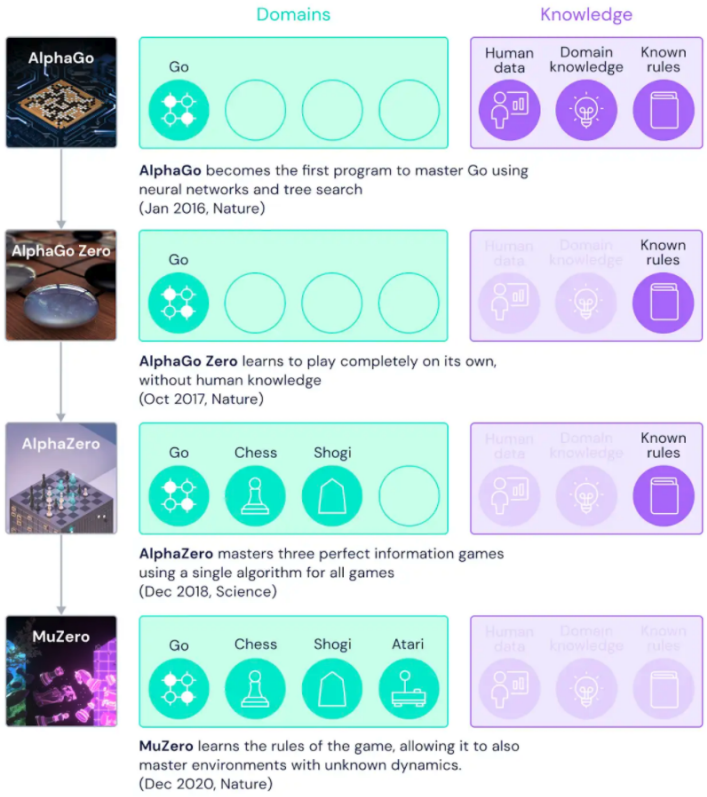

Figura 1: Evolución algorítmica en juegos

nos prometedoras y concentrarse en movimientos que son más probablemente beneficiosos, mejorando así la velocidad y la calidad de la toma de decisiones del juego.

2.4. Juegos de Estrategia y Algoritmos de Búsqueda

El estudio de los juegos de estrategia ha sido un campo fértil para el desarrollo de técnicas avanzadas de inteligencia artificial. Esta área ha ganado prominencia no solo por su aplicación en entretenimiento, sino también por su utilidad en la investigación de toma de decisiones complejas y resolución de problemas. Desde el histórico triunfo de Deep Blue de IBM sobre Garry Kasparov en ajedrez, hasta la victoria de AlphaGo de DeepMind sobre Lee Sedol en Go, se han explorado y desarrollado múltiples enfoques para diseñar algoritmos de IA. Estos juegos representan un desafío único debido a su espacio de búsqueda combinatoria masiva, que exige un equilibrio entre velocidad de cálculo y profundidad estratégica. Juegos como Connect6, con sus innumerables posibilidades y requisitos de planificación a largo plazo, ejemplifican los desafíos y oportunidades en este campo, ofreciendo un terreno fértil para la innovación en algoritmos de búsqueda y estrategias de juego impulsadas por IA.

3. Metodología

Connect6 es un juego de tablero de suma cero donde se enfrentan dos oponentes (las fichas negras y las fichas blancas). En cada turno un jugador puede colocar dos fichas en un tablero de 19x19 posiciones (tablero del estilo de los juegos Go y Gomoku). El jugador con fichas negras comienza la partida colocando una única ficha y el ganador del juego es el jugador que consiga conectar 6 fichas consecutivas en cualquier fila, columna o diagonal.

3.1. Análisis de Estrategias

En esta sección se presentan diversas estrategias efectivas para el juego Connect6 que pueden mejorar tu desempeño y resultados. Las estrategias abarcan desde el control del centro del tablero hasta la planificación anticipada y el aprendizaje continuo. Con un enfoque tanto en el bloqueo como en el ataque, estas estrategias buscan brindar una visión integral para abordar el juego de manera efectiva. A continuación se discuten algunas estrategias generales para Connect6:

- **Controlar el Centro**: El centro del tablero es un área clave en Connect6. Si se controla el centro, se tendrán más oportunidades de construir la línea en cualquier dirección.
- **Priorizar el Bloqueo sobre el Ataque**: Si el oponente tiene una fila de 4, es crucial bloquearla inmediatamente, incluso si se tiene la oportunidad de extender la línea propia. Una fila de 4 puede convertirse en una fila de 6 si se deja sin bloquear.
- **Balanceo entre Ataque y Bloqueo**: Como se pueden colocar dos piezas por turno, es posible usar la primera pieza para bloquear al oponente y la segunda pieza para construir una línea.
- **Planificar con Anticipación**: Trata de pensar algunos movimientos por delante y anticipa lo que tu oponente podría hacer. Esto puede ayudarte a tomar mejores decisiones y mejorar tus posibilidades de ganar.
- **Aprender de los Errores**: La mejor manera de mejorar una estrategia es practicar y aprender de cada juego.

3.1.1. Estrategias de bloqueo

- **Usar Ambos Movimientos para Bloquear**: Como se pueden colocar dos piezas por turno, hay que considerar usar ambas piezas para bloquear diferentes filas de 4 si el oponente tiene más de una.
- **Bloquear en el Punto más Estratégico**: Si hay varios puntos donde se puede bloquear una fila de 4, hay que escoger aquel que también interfiera con la mayoría de las líneas potenciales del oponente. Por ejemplo, si bloquear en el punto A también interrumpe una posible fila de 3 del oponente, elige el punto A en lugar del punto B, que solo interrumpe la fila de 4.

3.1.2. Estrategias de ataque

- **Crear Múltiples Amenazas**: Se trata de construir múltiples líneas al mismo tiempo. De esta manera, el oponente tendrá más dificultades para bloquear todas ellas.

3.2. Negamax

El algoritmo de búsqueda adversaria Negamax (ver figura 1) es una simplificación del algoritmo Minimax (este factor ayuda a simplificar su implementación y facilitar la incorporación de modificaciones). Además a la hora de realizar el diseño del algoritmo se han tomado en cuenta aspectos propios del juego Connect6, como:

- La posibilidad de de un empate.
- La colocación de dos fichas por turno (en el esquema propuesto se considera colocar dos fichas como un único movimiento).
- Límite en la profundidad de exploración.

Algorithm 1 Negamax

Require: $player \in \{1, -1\}$, $board$, $prevMove$, $depth \geq 0$,
  $isTerminal \leftarrow checkWin(board, prevMove)$
  if $isTerminal$ **then**
    return $-\infty$, $\varnothing$ ▷ Nodo Terminal
  else if $depth = 0$ **then**
    return $score(board, player)$, $\varnothing$
  end if
  $moves \leftarrow getValidMoves(board)$
  if $moves = \varnothing$ **then**
    return 0, $\varnothing$ ▷ Tablero lleno
  end if
  $maxValue \leftarrow -\infty$
  $nextPlayer \leftarrow -player$
  for $move \in moves$ **do** ▷ Explorar el siguiente nivel
    $nextBoard \leftarrow makeMove(board, player, move)$
    $value \leftarrow -Negamax(nextPlayer, nextBoard, move, depth - 1)$
    if $value > maxValue$ **then**
      $maxValue \leftarrow value$
    end if
  end for
  return $maxValue$, $moveWithValue(maxValue)$

3.2.1. Poda Alpha-Beta

Una posible mejora del algoritmo Negamax, es la poda Alfa-Beta (ver figura 2). En dicha mejora se establecen dos márgenes (uno superior y otro inferior) que se utilizan para realizar una poda en el árbol de búsqueda. Sin embargo dicha poda se ve afectada en gran medida por la función de evaluación del tablero (*score*) escogida y los valores iniciales de Alfa y Beta.

Algorithm 2 NegamaxAlphaBeta

Require: $player \in \{1, -1\}$, $board$, $prevMove$, $depth \geq 0$, $\alpha \leftarrow -\infty$, $\beta \leftarrow \infty$
  $isTerminal \leftarrow checkWin(board, prevMove)$
  if $isTerminal$ **then**
    return $-\infty, \varnothing$ ▷ Nodo Terminal
  else if $depth = 0$ **then**
    return $score(board, player), \varnothing$
  end if
  $moves \leftarrow getValidMoves(board)$
  if $moves = \varnothing$ **then**
    return $0, \varnothing$ ▷ Tablero lleno
  end if
  $maxValue \leftarrow -\infty$
  $nextPlayer \leftarrow -player$
  for $move \in moves$ **do** ▷ Explorar el siguiente nivel
    $nextBoard \leftarrow makeMove(board, player, move)$
    $value \leftarrow -Negamax(nextPlayer, nextBoard, move, depth - 1, \alpha, \beta)$
    if $value \geq maxValue$ **then**
      $maxValue \leftarrow value$
    end if
    $\alpha \leftarrow max(\alpha, maxValue)$ ▷ Poda Alpha-Beta
    if $\alpha > \beta$ **then**
      $breakLoop()$
    end if
  end for
  return $maxValue$, $moveWithValue(maxValue)$

3.2.2. Move Ordering

El orden de exploración de las ramas del árbol de búsqueda es muy relevante a la hora de utilizar Negamax con poda Alfa-Beta, debido a dos factores:

- La posibilidad de descartar ramas relevantes en la búsqueda.
- Al explorar las ramas más prometedores primero, se podan con antelación las rama no prometedoras.

La ordenación se establece según una función heurística, que representa mediante un valor numérico como de prometedora es una rama (ver figura 3).

Algorithm 3 NegamaxAlphaBeta with Move Ordering

Require: $player \in \{1, -1\}$, $board$, $prevMove$, $depth \geq 0$, $\alpha \leftarrow -\infty$, $\beta \leftarrow \infty$, $heuristic$
  $isTerminal \leftarrow checkWin(board, prevMove)$
  if $isTerminal$ **then**
    return $-\infty, \varnothing$ ▷ Nodo Terminal
  else if $depth = 0$ **then**
    return $score(board, player), \varnothing$
  end if
  $moves \leftarrow getValidMoves(board)$
  $moves \leftarrow orderMoves(board, heuristic)$ ▷ Ordenar la exploración de los movimientos
  if $moves = \varnothing$ **then**
    return $0, \varnothing$ ▷ Tablero lleno
  end if
  $maxValue \leftarrow -\infty$
  $nextPlayer \leftarrow -player$
  for $move \in moves$ **do** ▷ Explorar el siguiente nivel
    $nextBoard \leftarrow makeMove(board, player, move)$
    $value \leftarrow -Negamax(nextPlayer, nextBoard, move, depth - 1, \alpha, \beta)$
    if $value \geq maxValue$ **then**
      $maxValue \leftarrow value$
    end if
    $\alpha \leftarrow max(\alpha, maxValue)$ ▷ Poda Alpha-Beta
    if $\alpha > \beta$ **then**
      $breakLoop()$
    end if
  end for
  return $maxValue$, $moveWithValue(maxValue)$

3.2.3. Tablas de transposición Hash

Si se realizan los mismos movimientos pero en diferente orden, se obtiene el mismo tablero. Almacenando los tableros ya explorados y su valor, se eliminan las exploraciones redundantes de ramas con resultado idéntico (ver figura 4). Adicionalmente existe otro factor que consiste en la propiedad de simetría del tablero, es decir el valor asociado a un tablero debería ser conservado ante las rotaciones de $90^{\underline{o}}$, $180^{\underline{o}}$ y $270^{\underline{o}}$.

Algorithm 4 NegamaxAlphaBetaMemory with Move Ordering

Require: $player \in \{1, -1\}$, $board$, $prevMove$, $depth \geq 0$, $\alpha \leftarrow -\infty$, $\beta \leftarrow \infty$, $heuristic$
  if $board \in table$ **then**
    return $tableValue(board)$, $tableMove(board)$ ▷ Tablero explorado anteriormente
  end if
  $isTerminal \leftarrow checkWin(board, prevMove)$
  if $isTerminal$ **then**
    return $-\infty$, $\varnothing$ ▷ Nodo Terminal
  else if $depth = 0$ **then**
    return $score(board, player)$, $\varnothing$
  end if
  $moves \leftarrow getValidMoves(board)$
  $moves \leftarrow orderMoves(board, heuristic)$ ▷ Ordenar la exploración de los movimientos
  if $moves = \varnothing$ **then**
    return 0, $\varnothing$ ▷ Tablero lleno
  end if
  $maxValue \leftarrow -\infty$
  $nextPlayer \leftarrow -player$
  for $move \in moves$ **do** ▷ Explorar el siguiente nivel
    $nextBoard \leftarrow makeMove(board, player, move)$
    $value \leftarrow -Negamax(nextPlayer, nextBoard, move, depth - 1, \alpha, \beta)$
    if $value \geq maxValue$ **then**
      $maxValue \leftarrow value$
    end if
    $\alpha \leftarrow max(\alpha, maxValue)$ ▷ Poda Alpha-Beta
    if $\alpha > \beta$ **then**
      $breakLoop()$
    end if
  end for
  $addBoardToTable(table, board, maxValue)$ ▷ Añadir puntuación del tablero a la tabla
  return $maxValue$, $moveWithValue(maxValue)$

3.2.4. MTD

La implementación del algoritmo MTD en nuestro proyecto surgió como una exploración para mejorar la eficiencia del Negamax con poda Alfa-Beta, especialmente en términos de uso de la memoria y velocidad de ejecución. MTD, una variante del algoritmo Minimax, es conocido por su eficiencia en juegos de estrategia complejos, donde manejar un amplio espacio de búsqueda es fundamental.

El algoritmo MTD (ver figura 5) funciona como una serie de búsquedas binarias ajustadas sobre un valor de guía de puntuación (g). Esta técnica reduce significativamente la cantidad de memoria requerida en comparación con los métodos tradicionales de búsqueda en árboles de juegos. La idea central es que, en lugar de buscar en un amplio rango de valores alfa y beta, MTD realiza iteraciones sucesivas para afinar estos valores, buscando el valor óptimo de forma más directa y eficiente.

Algorithm 5 MTD(f)

Require: $player \in \{1, -1\}$, $board$, $prevMove$, $depth \geq 0$
  $g \leftarrow 0$
  $upperBound \leftarrow \infty$
  $lowerBound \leftarrow -\infty$
  while $lowerBound < upperBound$ **do**
    $\beta \leftarrow max(g, lowerBound + 1)$
    $\alpha \leftarrow \beta - 1$
    $g \leftarrow NegamaxAlphaBetaMemory(player, board, prevMove, depth, \alpha, \beta)$
    if $g < \beta$ **then**
      $upperBound \leftarrow g$
    else
      $lowerBound \leftarrow g$
    end if
  end while
  return g, $moveWithValue(g)$

3.3. Heurísticas

Las heurísticas en el contexto de la inteligencia artificial y, específicamente, en los juegos de estrategia, son métodos utilizados para tomar decisiones rápidas y eficientes. Una heurística es, en esencia, una regla práctica o enfoque simplificado que no garantiza una solución óptima, pero proporciona una solución suficientemente buena en un tiempo razonable. En los juegos, las heurísticas son cruciales para evaluar posiciones y movimientos potenciales, especialmente en situaciones donde el análisis exhaustivo de todas las posibilidades no es viable debido a la limitación de tiempo o recursos computacionales.

3.3.1. Heurística implementada

En nuestro motor de juego, hemos implementado una heurística específica para evaluar posiciones en el tablero. La función ***heuristic*** se centra en calcular un puntaje para un tablero dado y un jugador específico, considerando tanto las oportunidades del jugador como las amenazas del oponente. A continuación se detalla su funcionamiento:

```
1
2  @njit
3  def heuristic(board, player, move=None):
4      opponent = -player
5      score = 0
6
7      # Evaluacion de la maxima secuencia consecutiva de fichas
8      player_max_consecutive = maxConsecutive(board, player)
9      opponent_max_consecutive = maxConsecutive(board, opponent)
10
11     # Si el oponente tiene m s de 5 en linea, la situacion es cr tica
12     if opponent_max_consecutive > 5:
13         return np.inf
14
15     # Puntaje basado en la maxima secuencia del jugador
16     if player_max_consecutive > 2:
17         score += 4 ** (player_max_consecutive)
18
19     # Penalizacion basada en la maxima secuencia del oponente
20     if opponent_max_consecutive > 3:
21         score -= 5 ** (opponent_max_consecutive)
22     elif opponent_max_consecutive > 4:
23         score -= 20 ** (opponent_max_consecutive)
24
25     # Evaluacion adicional de bloqueos y bifurcaciones
26     score_fork, score_block = evaluate_board(board, player, opponent)
27     score += score_block
28     score += score_fork
29
30     return score
```

3.3.2. Evaluación de Bloqueos y Bifurcaciones

La función ***evaluate_board*** analiza el tablero en busca de oportunidades de bloqueo y bifurcación (forks). El bloqueo es una estrategia defensiva donde se impide que el oponente complete una línea de fichas consecutivas. La bifurcación, por otro lado, es una estrategia ofensiva que crea múltiples amenazas simultáneas, forzando al oponente a responder y dejando otras oportunidades abiertas para el jugador. Esta heurística combinada, que evalúa tanto la formación de líneas consecutivas como las estrategias de bloqueo y bifurcación, proporciona una base sólida para la toma de decisiones en nuestro motor de juego. Al equilibrar elementos ofensivos y defensivos, la heurística permite una jugabilidad más estratégica y desafiante.

4. Resultados

En esta sección, presentamos los resultados obtenidos con nuestro algoritmo Negamax, al que hemos denominado NegamaxUltimate. A través de un proceso exhaustivo de pruebas y mejoras, hemos logrado que NegamaxUltimate juegue de manera eficiente y estratégica, aunque no alcanza la perfección. Este equilibrio entre eficacia y practicidad ha sido fundamental en su diseño y funcionamiento.

4.1. Inicio de Partida

NegamaxUltimate muestra un comportamiento inicial distinto dependiendo del color de las fichas que maneja.

- **Con Fichas Negras:** Cuando juega con las fichas negras, nuestra IA tiende a colocar su primera ficha en el centro del tablero. Este enfoque centralizado le permite tener una mayor flexibilidad y control en las etapas iniciales del juego, abriendo un abanico de posibilidades estratégicas.
- **Con Fichas Blancas:** Por otro lado, cuando juega con las fichas blancas, NegamaxUltimate opta por un enfoque más defensivo, colocando su primera ficha arriba a la izquierda y abajo a la izquierda del rival. Este posicionamiento inicial busca contrarrestar las ventajas que el rival podría tener por jugar primero y establecer un juego más controlado (ver imagen 2).

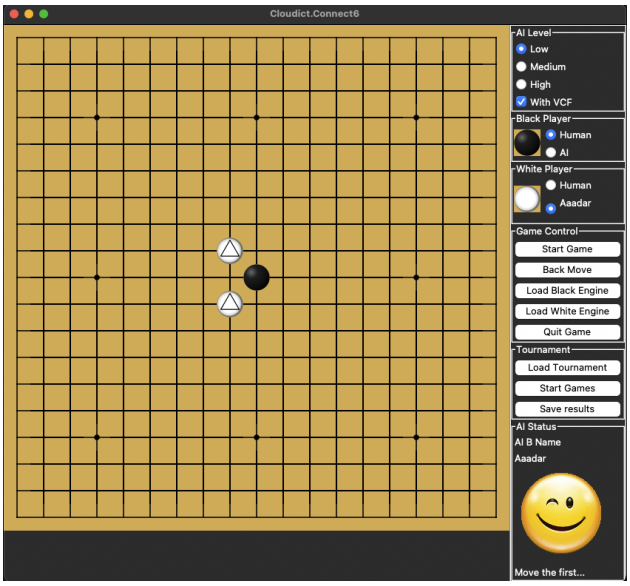

Figura 2: Inicio de partida

4.2. Obtención de bloqueos dobles

Una de las fortalezas clave de NegamaxUltimate es su capacidad para realizar bloqueos dobles eficaces. Esta habilidad se extiende a distintas configuraciones en el tablero:

- **Diagonales:** Puede bloquear efectivamente intentos del oponente de formar líneas diagonales, una estrategia común en juegos de alineación.
- **Verticales y Horizontales:** Además, es competente en la realización de bloqueos en líneas verticales y horizontales, impidiendo al oponente aprovechar estas direcciones para sus secuencias de fichas.
- **Partes Separadas del Mapa:** Notablemente, también logra realizar bloqueos en diferentes áreas del tablero simultáneamente. Esto demuestra una comprensión integral del juego y la capacidad de reaccionar ante múltiples amenazas.

Esto se puede analizar en la siguiente imagen: 3

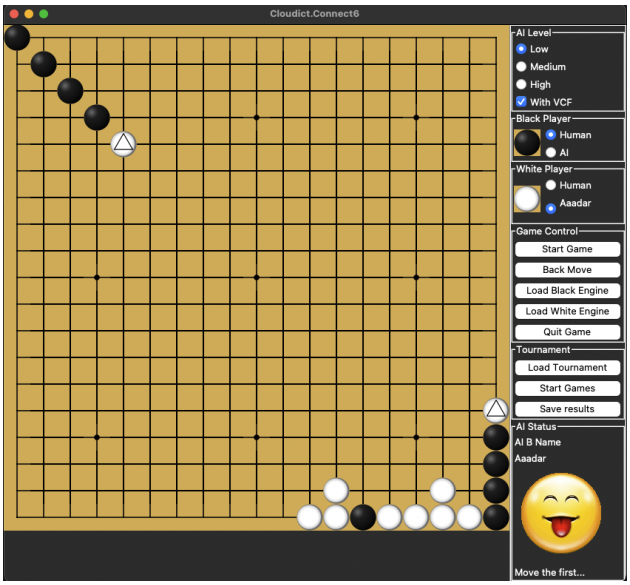

Figura 3: Bloqueos Dobles

4.3. Partida VS Cloudict

En los enfrentamientos contra Cloudict, un conocido jugador de habilidad avanzada, NegamaxUltimate no logra mantener una racha ganadora pero ofrece una resistencia notable. Aunque termina perdiendo estas partidas, muestra una capacidad considerable para dar "pelea", desafiando al oponente y demostrando la solidez de sus estrategias y su capacidad de adaptación. Este nivel de desempeño subraya el éxito de NegamaxUltimate en alcanzar un nivel competitivo, aunque aún hay margen para mejorar y aspirar a la perfección. Estas mencionadas partidas se pueden analizar en las imágenes: 4,5

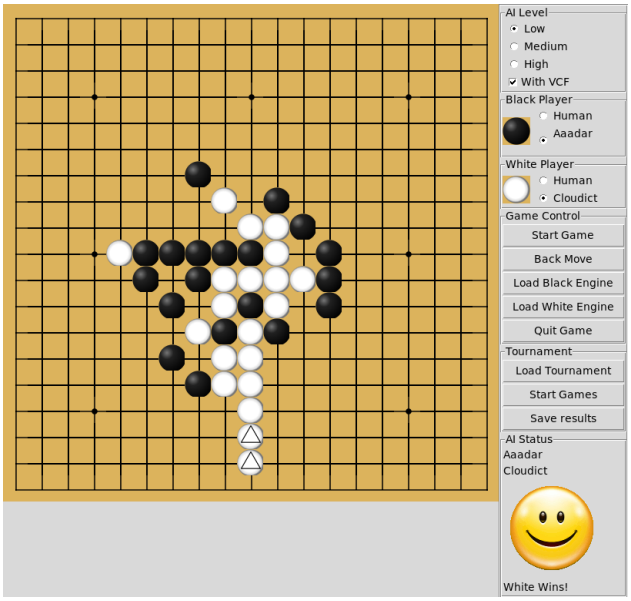

Figura 4: Partida siendo negras

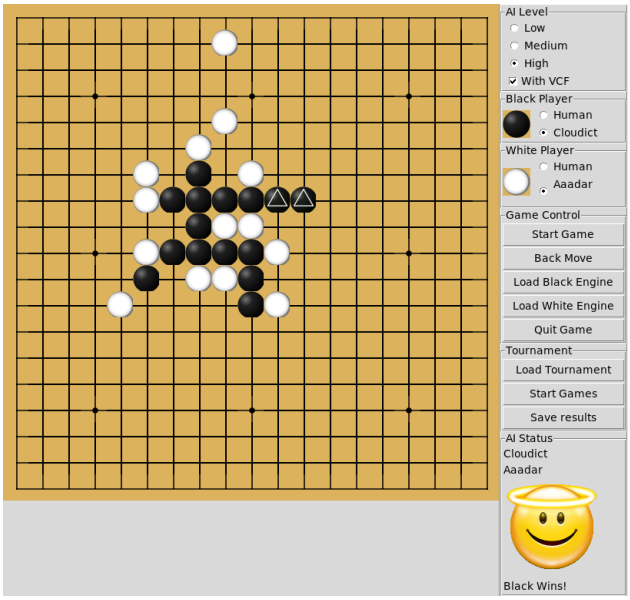

Figura 5: Partida siendo blancas

4.3.1. Análisis de Tiempos y Podas

El análisis detallado de los tiempos de ejecución y la eficacia de las podas en diferentes variantes de Negamax revela aspectos cruciales sobre el rendimiento y la eficiencia de estos algoritmos. A través de las gráficas generadas previamente, podemos observar visualmente la relación entre la profundidad de búsqueda y el tiempo de ejecución, proporcionando una comparativa clara y comprensible entre las distintas implementaciones de Negamax.

- **NegaMax Básico (Depth 1):** Con un tiempo de ejecución de 0.540 segundos y 63904 nodos visitados, esta versión básica muestra una exploración completa del árbol de juego sin implementar poda. La falta de poda se traduce en un menor tiempo de ejecución para una profundidad limitada, pero con una eficacia estratégica reducida, algo que se aprecia claramente en las gráficas 6.

- **NegaMaxAlphaBeta (Depth 1):** Al implementar la poda Alpha-Beta, el tiempo de ejecución asciende a 2.235 segundos para la misma cantidad de nodos visitados. Este aumento, que es evidente en las gráficas, se debe a la gestión adicional de las variables de poda, una sobrecarga necesaria para mejorar la eficiencia en profundidades mayores.

- **NegaMaxAlphaBetaMemory (Depth 1 y 3):** Registrando tiempos de 1.264 y 1.562 segundos respectivamente, esta versión optimiza el uso de la memoria. Las gráficas muestran cómo esta optimización afecta positivamente al rendimiento en mayores profundidades, evidenciando una reducción significativa en el número de nodos visitados.

- **NegaMaxUltimate (Depth 3, 6, 10):** Con tiempos de 5.877, 8.299 y 8.334 segundos en las profundidades respectivas, esta variante demuestra un incremento considerable en el tiempo de ejecución, como se muestra en las gráficas 6. A pesar de esto, los refinamientos adicionales permiten un juego más estratégico y una evaluación más precisa de las posiciones.

- **MTD (Depth 3, 6, 10):** El algoritmo MTD, con tiempos comparables a NegaMaxUltimate en profundidades mayores, destaca por su habilidad para manejar un alto número de ramificaciones y golpes de tabla. Las gráficas reflejan cómo esta capacidad se traduce en un juego más refinado y estratégico, aunque con un costo de tiempo similar a NegaMaxUltimate.

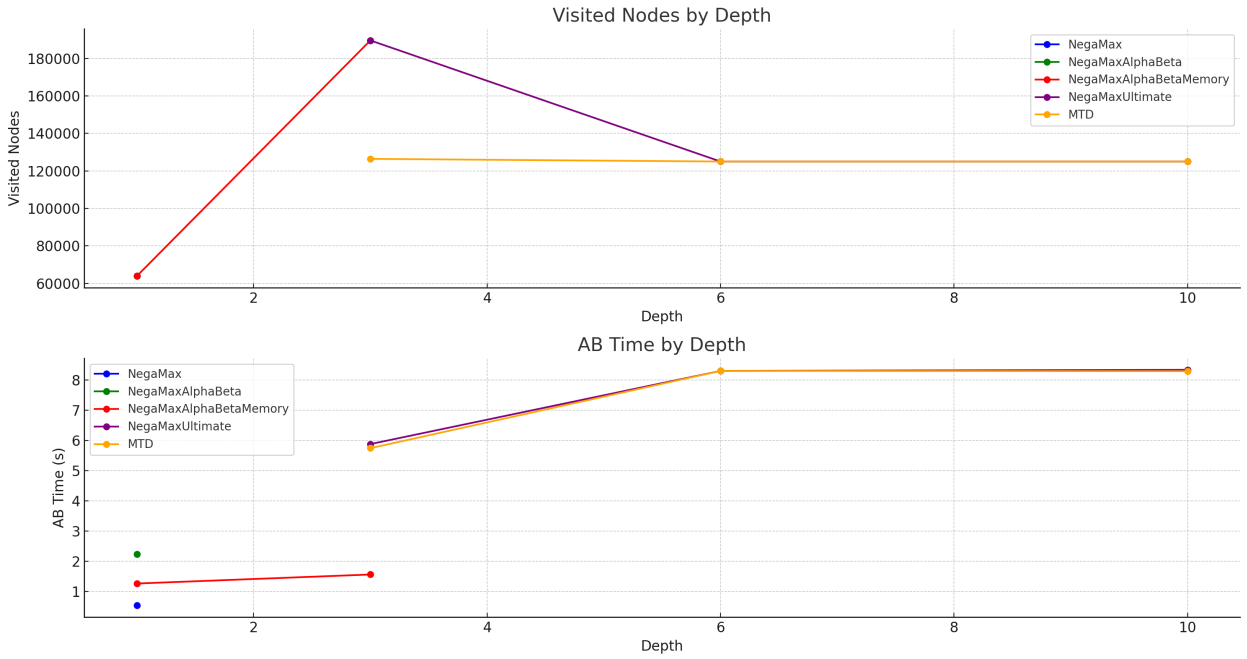

Figura 6: Gráficas de tiempo y nodos explorados

En conclusión, el análisis de las gráficas muestra claramente el compromiso entre eficiencia temporal y profundidad estratégica en las distintas versiones de Negamax. Las implementaciones más avanzadas, aunque más lentas, ofrecen una evaluación más precisa de las posiciones y una mejor estrategia de juego, aspectos fundamentales para competir a un alto nivel.

4.4. Consumo de memoria

En términos de consumo de memoria, el análisis indica que NegamaxUltimate es más eficiente. Inicia con un uso menor y mantiene un pico de memoria inferior en comparación con MTD(f). Este aspecto es crucial en aplicaciones donde los recursos de memoria son una consideración importante. La eficiencia de NegamaxUltimate en el uso de la memoria podría atribuirse a su implementación más simplificada y a la eficacia de sus heurísticas, lo que reduce la necesidad de almacenar extensas estructuras de datos o realizar cálculos complejos. Por otro lado, MTD(f), a pesar de ser eficiente en términos de tiempo de ejecución, demanda más recursos de memoria, probablemente debido a su proceso iterativo y la gestión de un rango más amplio de posibles movimientos y escenarios.

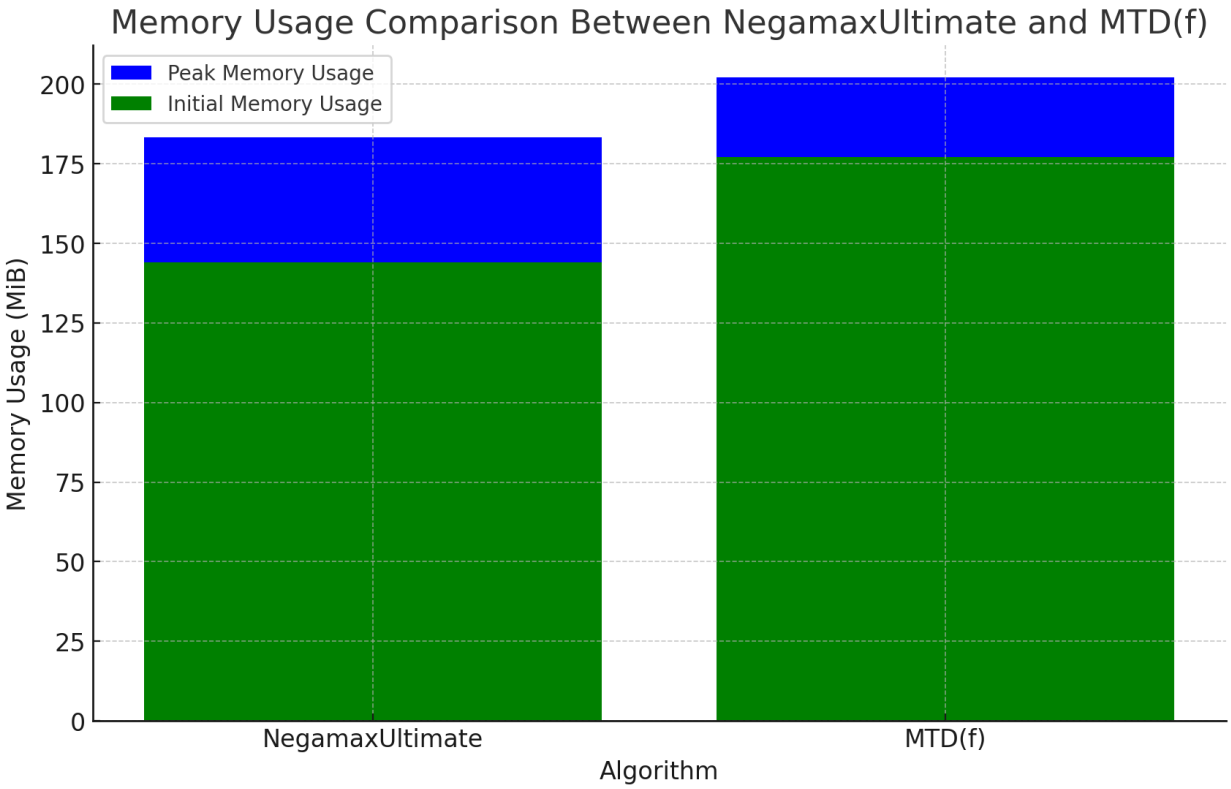

Figura 7: Gráfica que compara el consumo de memoria entre NegamaxUltimate y MTD(f).

Esta comparativa visual 7 resalta la diferencia en el uso de memoria entre ambos algoritmos, mostrando que MTD(f) consume más memoria tanto inicialmente como en su pico máximo. Esto podría deberse a la naturaleza del algoritmo MTD(f), que, aunque eficiente en términos de búsqueda, puede requerir más recursos para gestionar sus iteraciones y la evaluación de posiciones.

- NegamaxUltimate comienza con un uso de memoria de 144.1 MiB y alcanza un pico de 183.2 MiB.
- MTD(f) inicia con 177.1 MiB y llega a un máximo de 202.1 MiB.

5. Experimentación

En esta sección, abordamos las diversas soluciones que probamos y eventualmente descartamos durante el desarrollo del motor del juego. Cada técnica y enfoque fue evaluado en función de su eficacia, velocidad y capacidad para generar movimientos estratégicos. El proceso de experimentación es crucial en el desarrollo de sistemas de inteligencia artificial, ya que permite identificar las metodologías más adecuadas y eficientes para el problema en cuestión.

5.1. Monte Carlo Tree Search (MCTS)

Iniciamos experimentando con el Monte Carlo Tree Search (MCTS), un algoritmo popular y eficaz en muchos juegos de estrategia. Sin embargo, para nuestro caso particular, el MCTS no cumplió con las expectativas. Encontramos que, aunque teóricamente sólido, en la práctica necesitaba explorar una cantidad excesiva de movimientos para generar estrategias viables. Esta extensa exploración resultaba en una velocidad de ejecución demasiado lenta, lo que era impracticable para nuestro propósito. Además, los movimientos sugeridos por el MCTS a menudo eran subóptimos, lo que llevó a la decisión de descartar este enfoque en favor de métodos más prometedores.

5.2. Intento de Implementación de AlphaZero

AlphaZero, conocido por su rendimiento sobresaliente en juegos como el ajedrez y Go, fue otro enfoque que consideramos. Intentamos implementar un sistema basado en AlphaZero, pero nos enfrentamos a desafíos significativos. El principal obstáculo fue la necesidad de una cantidad extremadamente grande de entrenamientos para alcanzar un nivel de juego competente. Descubrimos que se requerían más de 100,000 partidas para que el sistema comenzara a realizar movimientos estratégicamente sólidos. Dada la intensidad computacional y el tiempo necesario para tal entrenamiento, esta opción resultó ser inviable para nuestras necesidades.

5.2.1. Finalmente Descartado

Después de considerar los desafíos y limitaciones, decidimos descartar la implementación de AlphaZero. A pesar de su potencial teórico, las demandas prácticas de entrenamiento y optimización superaron los recursos y el tiempo disponibles en nuestro proyecto.

5.3. Heurísticas Descartadas

Durante el desarrollo, probamos varias heurísticas, algunas de las cuales resultaron ser ineficaces y fueron descartadas.

5.3.1. Heurística Dummy

Una de las primeras heurísticas que probamos fue la heurística *Dummy*. Esta heurística era demasiado simplista; se centraba principalmente en crear líneas de fichas consecutivas, pero no consideraba aspectos estratégicos como bloqueos o jugadas avanzadas. A continuación, se muestra el código de esta heurística:

```
1
2 def heuristic(board, player, move=None):
3     ones = maxConsecutive(board, player)
4     score = 0
5
6     if ones > 2:
7         score += 10**(ones-1)
8
9     return score
```

5.3.2. Heurísticas de *Forks y Blocks*

Experimentamos también con dos heurísticas separadas: una centrada en crear bifurcaciones (Forks) y otra en realizar bloqueos (Blocks). Aunque estas heurísticas ofrecían un enfoque más estratégico que la heurística Dummy, todavía no eran suficientemente eficaces. Finalmente, decidimos unificar estas dos heurísticas en una sola, incorporando además movimientos *Killermoves* para aumentar la eficiencia y la velocidad de la toma de decisiones. Esta nueva heurística combinada mostró un rendimiento superior y fue uno de los enfoques que mantuvimos en el desarrollo final del motor del juego.

6. Conclusiones

En la realización de este proyecto, hemos alcanzado varios hitos significativos y obtenido aprendizajes valiosos en el campo de la inteligencia artificial aplicada a juegos de estrategia, específicamente en el desarrollo de un motor de juego para Conecta 6. A continuación, se resumen los aspectos más destacados y las conclusiones obtenidas:

1. **Desarrollo de un Algoritmo Eficiente:** Hemos implementado con éxito el algoritmo NegamaxUltimate, que ha demostrado ser eficaz en el juego Conecta 6. A pesar de no ser perfecto, su rendimiento es notable, especialmente en la realización de bloqueos dobles y en la adaptación a diferentes situaciones de juego.

2. **Optimizaciones Implementadas:** Las mejoras realizadas en la poda Alpha-Beta y la incorporación de técnicas como MTD(f) y heurísticas avanzadas han demostrado ser cruciales para mejorar la eficiencia del motor de juego, tanto en términos de tiempo de ejecución como en el uso de la memoria.

3. **Desafíos y Aprendizajes:** El proyecto presentó desafíos significativos, como la complejidad de implementar AlphaZero y las limitaciones del MCTS. Estos retos nos llevaron a explorar y finalmente adoptar enfoques más adecuados para nuestro contexto, subrayando la importancia de la adaptabilidad y la innovación en la IA.

4. **Experimentación y Resultados:** Las pruebas realizadas contra jugadores avanzados como Cloudict y los análisis de rendimiento han sido fundamentales para evaluar la efectividad de nuestro motor de juego. Aunque no siempre victorioso, NegamaxUltimate mostró una capacidad competitiva impresionante.

5. **Futuras Direcciones:** Aunque hemos logrado avances significativos, hay espacio para la mejora continua. La integración de técnicas de aprendizaje automático más avanzadas y la optimización adicional de algoritmos podrían ser áreas de desarrollo futuro.

6. **Contribución al Campo de la IA:** Este proyecto contribuye al entendimiento y desarrollo de algoritmos de búsqueda y estrategias en juegos de estrategia. Ha demostrado cómo la combinación de técnicas tradicionales de IA y enfoques innovadores puede resultar en un sistema de juego robusto y estratégicamente hábil.

Referencias

[1] M. Campbell, A. J. Hoane, and F. H. Hsu, "Deep blue," *Artificial Intelligence*, vol. 134, pp. 57–83, 1 2002.

[2] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. V. D. 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 2016 529:7587*, vol. 529, pp. 484–489, 1 2016. [Online]. Available: https://www.nature.com/articles/nature16961

[3] J. Schrittwieser, I. Antonoglou, T. Hubert, K. Simonyan, L. Sifre, S. Schmitt, A. Guez, E. Lockhart, D. Hassabis, T. Graepel, T. Lillicrap, and D. Silver, "Mastering atari, go, chess and shogi by planning with a learned model," *Nature 2020 588:7839*, vol. 588, pp. 604–609, 12 2020. [Online]. Available: https://www.nature.com/articles/s41586-020-03051-4

[4] S. J. Yen and J. K. Yang, "Two-stage monte carlo tree search for connect6," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 3, pp. 100–118, 6 2011.

[5] S. Shah and S. Gupta, "Reinforcement learning for connectx," 10 2022. [Online]. Available: https://arxiv.org/abs/2210.08263v1