**Informe del Código: Interfaz de Ajedrez de Alicia**

**Juan Pablo Escobar Viveros- 2259519**
**Edgar Andrés Vargas García- 2259690**
**William Alexander Franco Otero- 2259715**
**16/12/2024**

El Minimax es un algoritmo de decisión para juegos de dos jugadores que busca minimizar la máxima pérdida posible. En este caso, se aplica en un contexto de ajedrez con dos tableros (Alice Chess) junto con el ajedrez suicida.

Componentes clave del Minimax en este código:

1. **Estructura Básica**: El método `minimax()` tiene los siguientes parámetros principales:
    - `board1` y `board2`: Los dos tableros de Alice Chess
    - `depth`: Profundidad de búsqueda en el árbol de movimientos
    - `alpha` y `beta`: Para la poda alfa-beta
    - `maximizing_player`: Indica si es el turno del jugador que maximiza (true) o minimiza (false)

**2. Caso Base (Profundidad 0)**:
Cuando se alcanza la profundidad máxima, se evalúa el estado actual del tablero.
3. **Generación de Movimientos**:
Obtiene todos los movimientos posibles para el color actual.
4. **Árbol de Búsqueda - Maximizando**:
5. **Árbol de Búsqueda - Minimizando**: Es similar al maximizando, pero busca minimizar el valor:
**Ejemplo Simplificado del Proceso**: Imaginemos una profundidad de 3:

1. El jugador (maximizando) considera sus movimientos
2. Para cada movimiento, simula el movimiento
3. Llama recursivamente a Minimax para el oponente (minimizando)
4. El oponente minimiza considerando sus propios movimientos
5. Se repite hasta profundidad 0
6. En profundidad 0, se evalúa el estado del tablero

**Particularidades de Este Código**:

- Evalúa los tableros considerando:
    - Valor de las piezas
    - Número de piezas
    - Objetivo de reducir piezas propias

Vamos a desglosar cómo funciona la poda alfa-beta paso a paso:

1. **Inicialización de Alpha y Beta**:

- `alpha` comienza como `-infinity`
- `beta` comienza como `+infinity`

**Ejemplo Ilustrativo**:

Imagina un árbol de búsqueda simple:

Raíz

/ | \

A B C

/ | \ / \ / \

1 5 7 3 6 2 4

Sin poda alfa-beta, se explorarían todos los nodos. Con poda alfa-beta, podríamos cortar ramas que no mejoran el resultado.

**Cómo Funciona Concretamente**:

- Si en la rama del maximizador (A) ya encontramos un valor 7
- Y en la rama del minimizador (B) ya tenemos un valor 3
- Si en las siguientes exploraciones de B no podemos mejorar 3
- Podemos "podar" (cortar) esa rama completa

La condición `if beta <= alpha` detecta cuándo seguir explorando no aportará una mejor solución.

**Beneficios**:

- Reduce dramáticamente el número de nodos evaluados
- Mantiene la misma calidad de decisión
- Mejora la eficiencia computacional

En este código, la poda alfa-beta permite a la IA de ajedrez:

1. Explorar menos movimientos
2. Mantener una búsqueda profunda
3. Tomar decisiones más rápidas

## Informe del segundo proyecto - Inteligencia artificial

## Ajedrez de Alicia y Ajedrez Suicida

1. **Explicación breve de los dos tipos de ajedrez.**
   **Ajedrez Suicida:**

   El ajedrez suicida (también conocido como "ajedrez anti-juego" o "ajedrez de suicidio") es una variante del ajedrez en la que el objetivo no es ganar, sino perder todas tus piezas lo más rápido posible, incluyendo al rey. La idea es que el jugador que se quede sin piezas primero es el que gana la partida.

   **Ajedrez de Alicia:**

   En esta variante, las reglas del ajedrez cambian de manera muy peculiar. Se harán cambios de tablero a la hora en que una de las piezas hagan un movimiento, pasando del tablero original (Tablero 1) al tablero alterno (Tablero 2)

   **¿Cómo funcionan estos dos juntos?**

   Como tal, el/los jugadores pueden jugar en dos tableros al igual que capturar en ambos. Es OBLIGATORIO capturar/comer una vez se puede hacer con una de las fichas; Es decir, si un peón puede capturar o moverse de casilla, será impedido su movimiento para que únicamente pueda capturar.

2. **Implementación:**

   **A. IA (ia.py):** Se inicializa la inteligencia artificial y el grueso del código en la primera función de la clase.
   **board_instance**: Instancia del tablero, que contiene dos tableros **(board1 y board2).**
   **Color:** Color de las piezas de la IA.
   **piece_values:** Diccionario que asigna valores a las piezas para evaluar el tablero.

   Con la funcion **get_all_possible_moves** genera **todos los movimientos posibles** para un color específico, donde:

   **Prioriza capturas:** Si un movimiento implica capturar una pieza en el tablero actual, se agrega a una lista separada.
   **Doble tablero:** Considera las reglas del Alice Chess, donde las piezas pueden moverse entre dos tableros:
   Movimiento normal (dentro del mismo tablero).
   Movimiento "cruzado" (a otro tablero si la casilla está libre).

**Cuenta con los parámetros:**
**board1, board2:** Los dos tableros.
**color:** Color de las piezas que se deben mover.

**Y retorna:**
Lista de movimientos posibles, priorizando los que implican capturas.

Por medio de **evaluate_board** se evalúa el estado actual de ambos tableros utilizando el ajedrez suicida:
**Valor de las piezas:** Resta puntos si la IA tiene piezas en el tablero.
**Contador de piezas:** Prefiere tener **menos piezas** que el oponente (estrategia suicida).
El puntaje resultante favorece configuraciones donde la IA tiene menos piezas que el rival.
Recibe **board1, board2** para al final devolver el puntaje (score) que representa la evaluación del estado del tablero.

En **minimax** se implementa el algoritmo **Minimax con poda alfa-beta** para determinar el mejor movimiento:
Se explora recursivamente movimientos posibles hasta cierta profundidad.
**Maximizing player:** La IA intenta minimizar su puntaje (siguiendo la lógica suicida).
**Alpha-beta pruning:** Optimiza la búsqueda descartando ramas que no pueden mejorar el resultado. Recibe:
**board1, board2.**
**depth:** Profundidad de búsqueda.
**alpha, beta:** Valores utilizados para la poda.
**maximizing_player:** Define si es el turno del jugador maximizador o minimizador.
En este, devuelve el puntaje óptimo para el estado actual.

Y por ultimo, en **choose_best_move se** determina el **mejor movimiento** disponible usando el algoritmo Minimax:

Primero, se llama a **get_all_possible_moves** para obtener todos los movimientos posibles. Despues, para cada movimiento:
Realiza una copia de los tableros.
Simula el movimiento.
Evalúa el tablero usando **minimax.**
Selecciona el movimiento con el puntaje más favorable (priorizando menos piezas).
Toma depth (Profundidad de búsqueda (por defecto 3)) para retornar el mejor movimiento (o None si no hay movimientos disponibles).

La heurística de esta I.A está basada en 3 conceptos importantes:
1. El valor de cada una de las fichas asignadas inmediatamente en su código, estos valores se **sumarán** o **restarán** dependiendo del color de las piezas, lo que le permite

diferenciar entre piezas propias y del oponente.
2. Al ser un ajedrez suicida, se prioriza la muerte de las fichas propias. Se cuenta la cantidad de piezas restantes para **cada jugador**, donde se da un **bono positivo** a la evaluación si el oponente tiene más piezas que el AI.
Esto incentiva al algoritmo a realizar movimientos que resulten en la **eliminación de sus propias piezas**.
3. En el método **get_all_possible_moves**, se priorizan los movimientos que resultan en **capturas**: Si en la posición destino hay una pieza del oponente (indicado por dest_board1), se considera un **movimiento de captura** y se prioriza.
Si no es una captura, se añade a la lista de movimientos regulares.

**B. Board (board.py):**

La clase Board implementa la lógica del tablero doble para Alice Chess, junto con métodos para mover piezas, validar movimientos, deshacer jugadas y verificar el estado del juego.

**__init__** Inicializa:
**board1**: El primer tablero con las piezas en su posición inicial (usando la función create_initial_board de pieces).
**board2**: El segundo tablero vacío (matriz de 8x8 con None).
**current_player**: Jugador actual, comenzando con 'white'.
**move_history**: Historial de movimientos, útil para deshacer jugadas.
**game_over** y winner: Variables para verificar el estado del juego.

**get_piece** obtiene una pieza en una posición específica de un tablero seleccionado (board1 o board2). Recibe los datos:
**position**: Posición (x, y) de la pieza.
**board_number**: Número del tablero (1 o 2).

**3. is_valid_move**

Verifica si un movimiento es válido según estos parámetros:

1. Revisa que haya una pieza en la posición inicial.
2. Confirma que la pieza pertenece al jugador actual.
3. Comprueba si el movimiento destino está dentro de los posibles movimientos de la pieza.
4. Asegura que la casilla en el tablero contrario esté vacía (regla del Alice Chess).

Parámetros:
**start**: Posición inicial (x, y).
**end**: Posición final (x, y).
**board_number**: Tablero actual (1 o 2).

Y este retorna True si el movimiento es válido, False en caso contrario.

**4. move_piece**

Mueve una pieza de una posición a otra si el movimiento es válido:

1. Llama a **is_valid_move** para verificar la validez.
2. Después, actualiza el tablero moviendo la pieza a la posición destino.
3. Transfiere automáticamente la pieza al otro tablero con **_transfer_to_other_board**.
4. Guarda el movimiento en **move_history**.
5. Alterna el turno entre jugadores (**white** ↔ **black**).
6. Llama a **_check_game_status** para verificar si el juego ha terminado.

Parámetros:
**start**: Posición inicial (x, y).
**end**: Posición final (x, y).
**board_number**: Tablero actual.

**5. _transfer_to_other_board**

Transfiere una pieza a la misma posición en el tablero opuesto si la casilla está vacía.

Parámetros:
**position**: Posición (x, y) de la pieza.
**target_board_number**: Tablero opuesto (1 ↔ 2).

**6. print_board**

Imprime el estado del tablero seleccionado en un formato legible:
Representa piezas y muestra las casillas vacías.

Parámetros:

**board_number**: Tablero a imprimir (1 o 2).

**C. Piezas (pieces.py):**

Este módulo define las clases que representan las piezas de ajedrez y sus movimientos válidos en un tablero. Cada pieza extiende la clase base Piece y sobrescribe el método get_possible_moves para definir su comportamiento específico.

**1. Clase Base: Piece**

**Atributos:**

**color:** 'white' o 'black'.

**name:** Nombre de la pieza (ej., 'Pawn', 'Rook').

**has_moved:** True si la pieza ha sido movida (útil para peones y enroques).

**Métodos:**

**__repr__:** Representación de la pieza en texto (ej., white Pawn).

**2. Clase Pawn (Peón)**

**Movimiento estándar:** un cuadro hacia adelante.

**Movimiento especial:** Si no se ha movido aún, puede avanzar dos cuadros.

Captura diagonalmente.

**Métodos auxiliares:**

**_is_valid_move:** Verifica si la casilla destino está vacía.

**_is_valid_capture:** Valida si hay una pieza contraria en la casilla diagonal.

**3. Clase Rook (Torre)**

**Movimiento:** horizontal y vertical en cualquier cantidad de casillas.

Continúa moviéndose en una dirección hasta encontrar un obstáculo.

**4. Clase Knight (Caballo)**

**Movimiento:** en forma de "L" (dos cuadros en una dirección y uno perpendicularmente).

Puede saltar sobre otras piezas.

**5. Clase Bishop (Alfil)**

**Movimiento:** diagonalmente en cualquier cantidad de casillas.
Se mueve hasta encontrar una pieza o el borde del tablero.

**6. Clase Queen (Reina)**

**Movimiento:** combinación de torre y alfil.

Puede moverse horizontal, vertical y diagonalmente en cualquier cantidad de casillas.

**7. Clase King (Rey)**

**Movimiento:** un cuadro en cualquier dirección (horizontal, vertical o diagonal).

**8. Función create_initial_board**

Inicializa un tablero de ajedrez con la configuración estándar:

Devuelve una matriz 8x8 representando el tablero inicial.

4. GUI (gui.py):

**5. Main (main.py):**

**AliceSuicideChess (Clase Principal):**

Es la clase que orquesta el funcionamiento del juego. Esta clase implementa el juego **Alice Chess con reglas de ajedrez suicida** y su interacción con la interfaz gráfica (ChessGUI), el tablero (Board) y la inteligencia artificial (ChessAI).

**1. __init__**

Inicializa el juego con:

**self.board**: Instancia del tablero (Board) con dos tableros.
**suicide_mode**: Activa las reglas de ajedrez suicida (debes capturar si es posible).
**alice_mode**: Activa las reglas del ajedrez en dos tableros.
**current_player**: Jugador actual ('white' al inicio).

**ai_player**: Configura la IA para jugar como 'black'.
**game_over y winner**: Variables que rastrean el estado del juego.

**2. check_game_over**

Verifica si el juego ha terminado bajo las reglas de ajedrez suicida según:
Si el **jugador actual no tiene piezas** o **no puede realizar movimientos válidos**, pierde.

Recorre el tablero en busca de piezas del jugador actual y verifica si alguna puede moverse.

Si no hay movimientos posibles, **el otro jugador gana** y game_over se establece como True.

**3. is_forced_capture**

Determina si hay **movimientos de captura obligatorios**:

- En ajedrez suicida, si es posible capturar una pieza, el jugador **debe capturar**.
- Recorre el tablero en busca de piezas del color actual y verifica si tienen movimientos de captura.

**4. validate_move**

Valida los movimientos según las reglas:

Si hay capturas obligatorias (**is_forced_capture**), el jugador debe elegir un movimiento de captura.

Si el movimiento no cumple con esta regla, se lanza un error.

**Parámetros:**

**start**: Posición inicial de la pieza (x, y).

**end**: Posición destino de la pieza (x, y).

**board_num**: Número del tablero actual.

**5. start_game**

Inicia el juego y configura la interfaz gráfica:

Crea una instancia de **ChessGUI** y la asocia al tablero (Board).

Sobrescribe el método **on_square_click** de la GUI para incluir validación de movimientos según las reglas del juego.

**Después de cada movimiento:**

Valida el movimiento usando **validate_move** y verifica si el juego ha terminado con **check_game_over**. Si es así, muestra un mensaje con el ganador.

**Interfaz con ChessGUI:**

**custom_move_method** reemplaza el comportamiento de clics en la GUI:

Valida los movimientos, actualiza el tablero y cambia de turno y muestra un mensaje cuando el juego termina.

**6. main:**

Inicia el juego creando una instancia de **AliceSuicideChess** y ejecutando **start_game**.

**Componentes Principales de la GUI**

**3.1 Creación de la Interfaz Gráfica**

- Implementada usando la clase tkinter.
- Se crean dos tableros visuales mediante **tk.Canvas**.
- Los tableros tienen un tamaño de **8x8** y cada celda alterna colores (blanco y gris).

**3.2 Control de Eventos**

El código permite la interacción del usuario mediante clics:

1. **Primer clic**: Selecciona una pieza.
2. **Segundo clic**: Intenta mover la pieza a una nueva posición.

3. Si el movimiento falla, se muestra un **mensaje de error**.

### 3.3 Botones de Control

Se incluye un botón llamado **“Mover IA”** que activa un movimiento de la inteligencia artificial, calculado por la clase ChessAI.

### Funciones Clave

A continuación, se describen las funciones principales:

- **create_board_frames()**: Crea y organiza los dos tableros en la interfaz.
- **draw_board()**: Dibuja las celdas y piezas en el tablero correspondiente.
- **on_square_click()**: Maneja la lógica de selección y movimiento de piezas.
- **update_boards()**: Actualiza visualmente los tableros después de un movimiento.
- **make_ai_move()**: Llama a la IA para realizar un movimiento automático.

### Conclusión GUI

El código proporciona una interfaz gráfica funcional y amigable para un juego de ajedrez con características especiales, permitiendo interacción **manual** y **automática**. La integración con clases externas (Board y ChessAI) asegura una separación clara entre la lógica del juego y la interfaz de usuario.