PROYECTO AJEDREZ DE ALICIA: MINIMAX Y PODA ALPHA BETA

Tina María Torres - 2259729

Marlon Astudillo Muñoz - 2259462

Juan José Gallego Calderón - 2259433

Juan José Hernández Arenas - 2259500

Jersson Gutierrez Gonzalez - 2060071

Universidad Del Valle

Ingeniería De Sistemas

Inteligencia Artificial

Tuluá, Valle Del Cauca

2024

HUMANOS VS I.A

CREANDO EL TABLERO

```python
def create_board(self, empty=False):
    if empty:
        return [[None for _ in range(8)] for _ in range(8)] #Tablero 8x8

    # Configuración inicial del ajedrez
    pieces = ["R", "N", "B", "Q", "K", "B", "N", "R"]
    board = []

    # Fila de piezas negras
    board.append([("black", piece) for piece in pieces])
    # Peones negros
    board.append([("black", "P") for _ in range(8)])
    # Filas vacías
    for _ in range(4):
        board.append([None for _ in range(8)])
    # Peones blancos
    board.append([("white", "P") for _ in range(8)])
    # Fila de piezas blancas
    board.append([("white", piece) for piece in pieces])

    return board
```

1. Introducción a la creación del tablero de ajedrez

La función `create_board` se utiliza para generar un tablero de ajedrez de **8x8**, ya sea vacío o con la configuración inicial de las piezas, este tablero está conformado por una matriz.

2. Parametros

- **`empty`** *(booleano)*:
    - Si es `True`, se crea un tablero vacío (solo contiene `None`).

```
[
    [None, None, None, None, None, None, None, None],
    [None, None, None, None, None, None, None, None],
    [None, None, None, None, None, None, None, None],
    [None, None, None, None, None, None, None, None],
    [None, None, None, None, None, None, None, None],
    [None, None, None, None, None, None, None, None],
    [None, None, None, None, None, None, None, None],
    [None, None, None, None, None, None, None, None],
]
```

    - Si es `False`, se inicializa el tablero con la **configuración estándar del ajedrez**.

```
[
    [("black", "R"), ("black", "N"), ("black", "B"), ("black", "Q"), ("black", "K"), ("black", "B"), ("black", "N"), ("black", "R")],
    [("black", "P"), ("black", "P"), ("black", "P"), ("black", "P"), ("black", "P"), ("black", "P"), ("black", "P"), ("black", "P")],
    [None, None, None, None, None, None, None, None],
    [None, None, None, None, None, None, None, None],
    [None, None, None, None, None, None, None, None],
    [None, None, None, None, None, None, None, None],
    [("white", "P"), ("white", "P"), ("white", "P"), ("white", "P"), ("white", "P"), ("white", "P"), ("white", "P"), ("white", "P")],
    [("white", "R"), ("white", "N"), ("white", "B"), ("white", "Q"), ("white", "K"), ("white", "B"), ("white", "N"), ("white", "R")],
]
```

3. Puntos Importantes

1. **Estructura de Datos:**
    - Cada celda contiene una tupla `(color, pieza)` cuando hay una pieza.
    - `None` representa una celda vacía.
2. **Uso del Parámetro `empty`:**
    - Permite reutilizar la función para crear un tablero limpio o inicializado.
3. **Configuración de Piezas:**
    - La fila de piezas sigue el orden estándar: Torre, Caballo, Alfil, Reina, Rey, Alfil, Caballo, Torre.
    - Los peones ocupan la segunda y penúltima fila.

4. Introducción a `display_board`:

Método auxiliar que realiza la tarea de imprimir un tablero individual en la consola, la función `display_board` se encarga de mostrar en consola una representación textual de un tablero individual. Cada celda del tablero se representa ya sea con el primer carácter del nombre de una pieza o con un punto (`.`) si está vacía.

```
def display_board(self, board):
    for row in board:
        print(" ".join([f"{cell[1][0]}" if cell else "." for cell in row])) # mostrar tablero en texto
```

- **Entrada (`board`)**:

- ➢ `board`: Una lista de listas que representa un tablero de ajedrez de 8x8.
  - Cada celda del tablero puede ser:
  - Una tupla como `("black", "Queen")`, que representa una pieza negra Reina.
  - `None`, si la celda está vacía.

❖ **Salida en Consola**:

- ➢ Cada fila del tablero se imprime en una línea de texto.
- ➢ Las piezas se representan con la primera letra de su nombre (por ejemplo, `R` para Torre, `Q` para Reina).
- ➢ Las celdas vacías se representan con un punto (`.`).

```
def display_boards(self):
    print("\nBoard A:")
    self.display_board(self.boards["A"]) # imprime el tablero A
    print("\nBoard B:")
    self.display_board(self.boards["B"]) # imprime el tablero B
```

5. Introducción a `display_boards`

- Método diseñado para mostrar en la consola el estado actual de los dos tableros del ajedrez de Alicia: **tablero A** y **tablero B**. Esto es especialmente útil para depuración o cuando no se cuenta con una interfaz gráfica y se tienen que hacer pruebas.

MOVIMIENTOS DE LAS FICHAS

```python
def get_legal_moves(self, position, start_board, end_board):
    x, y = position
    piece = self.boards[start_board][x][y]

    if not piece:
        return []

    color, piece_type = piece

    if color != self.current_turn:
        return []

    moves = []

    if piece_type == "P": # Peón
        moves = self.get_pawn_moves(x, y, start_board, end_board)
    elif piece_type == "N": # Caballo
        moves = self.get_knight_moves(x, y, start_board, end_board)
    elif piece_type == "B": # Alfil
        moves = self.get_bishop_moves(x, y, start_board, end_board)
    elif piece_type == "R": # Torre
        moves = self.get_rook_moves(x, y, start_board, end_board)
    elif piece_type == "Q": # Reina
        moves = self.get_queen_moves(x, y, start_board, end_board)
    elif piece_type == "K": # Rey
        moves = self.get_king_moves(x, y, start_board, end_board)

    return moves

def get_pawn_moves(self, x, y, start_board, end_board):
    # Movimientos posibles del peón
```

1. Introducción a la creación del tablero de ajedrez

Implementa la lógica para calcular los movimientos legales de todas las piezas en una variante de ajedrez con dos tableros

- `start_board` (Tablero A)
- `End_board` (Tablero B)

Cada pieza tiene reglas específicas de movimiento, y el programa determina las posiciones válidas que puede ocupar una pieza según las reglas estándar del ajedrez.

2. Componentes Principales

1. **Función `get_legal_moves`:**
    - Determina los movimientos legales de una pieza en una posición dada.
    - Clasifica las piezas según su tipo (`P`, `N`, `B`, `R`, `Q`, `K`) y llama a la función específica que calcula sus movimientos.

```python
def get_legal_moves(self, position, start_board, end_board):
    x, y = position
    piece = self.boards[start_board][x][y]

    if not piece:
        return []

    color, piece_type = piece

    if color != self.current_turn:
        return []

    moves = []

    if piece_type == "P": # Peón
        moves = self.get_pawn_moves(x, y, start_board, end_board)
    elif piece_type == "N": # Caballo
        moves = self.get_knight_moves(x, y, start_board, end_board)
    elif piece_type == "B": # Alfil
        moves = self.get_bishop_moves(x, y, start_board, end_board)
    elif piece_type == "R": # Torre
        moves = self.get_rook_moves(x, y, start_board, end_board)
    elif piece_type == "Q": # Reina
        moves = self.get_queen_moves(x, y, start_board, end_board)
    elif piece_type == "K": # Rey
        moves = self.get_king_moves(x, y, start_board, end_board)

    return moves
```

2. **Funciones de Movimiento de Piezas:**
    - Peón (`get_pawn_moves`): Movimiento hacia adelante y captura en diagonal.

```python
def get_pawn_moves(self, x, y, start_board, end_board):
    # Movimientos posibles del peón
    moves = []
    direction = -1 if self.current_turn == "white" else 1 # Dirección del peón (hacia adelante)
    target_board = self.boards[end_board] # Tablero objetivo

    # Movimiento simple hacia adelante
    if 0 <= x + direction < 8 and not target_board[x + direction][y]: # Si no hay pieza en la casilla y que este dentro del limite del tablero (x)
        moves.append((x + direction, y)) # Se agrega la casilla a los movimientos posibles

    # Captura diagonal
    for dy in [-1, 1]:
        if 0 <= y + dy < 8 and 0 <= x + direction < 8: # Si no hay pieza en la casilla y que este dentro del limite del tablero (y)
            target_piece = target_board[x + direction][y + dy] # Se toma la pieza en la casilla diagonal objetivo.
            if target_piece and target_piece[0] != self.current_turn: # Si hay una pieza y no es del mismo color
                moves.append((x + direction, y + dy)) # Se agrega la casilla a los movimientos posibles

    return moves
```

    - Caballo (`get_knight_moves`): Movimientos en forma de "L".

```python
def get_knight_moves(self, x, y, start_board, end_board):
    # Movimientos posibles del caballo
    moves = []
    target_board = self.boards[end_board] # Tablero objetivo
    knight_jumps = [
        (2, 1), (2, -1), (-2, 1), (-2, -1), # Movimientos en L
        (1, 2), (1, -2), (-1, 2), (-1, -2) # Movimientos en L
    ]

    for dx, dy in knight_jumps: # Se recorren los saltos del caballo
        nx, ny = x + dx, y + dy # Se calcula la nueva posición
        if 0 <= nx < 8 and 0 <= ny < 8: # Si la nueva posición esta dentro del limite del tablero
            target_piece = target_board[nx][ny] # Se toma la pieza en la nueva posición
            if not target_piece or target_piece[0] != self.current_turn: # Si no hay pieza o la pieza no es del mismo color
                moves.append((nx, ny)) # Se agrega la casilla a los movimientos posibles

    return moves
```

- Alfil (`get_bishop_moves`): Movimientos diagonales deslizantes.

```python
def get_bishop_moves(self, x, y, start_board, end_board): # Alfil
    return self.get_sliding_moves(x, y, start_board, end_board, directions=[(1, 1), (1, -1), (-1, 1), (-1, -1)])
```

- Torre (`get_rook_moves`): Movimientos horizontales y verticales deslizantes.

```python
def get_rook_moves(self, x, y, start_board, end_board): # Torre
    return self.get_sliding_moves(x, y, start_board, end_board, directions=[(1, 0), (0, 1), (-1, 0), (0, -1)])
```

- Reina (`get_queen_moves`): Movimientos combinados de torre y alfil.

```python
def get_queen_moves(self, x, y, start_board, end_board): # Reina
    return self.get_sliding_moves(x, y, start_board, end_board, directions=[
        (1, 1), (1, -1), (-1, 1), (-1, -1), (1, 0), (0, 1), (-1, 0), (0, -1) # Se obtienen los movimientos posibles
    ])
```

- Rey (`get_king_moves`): Movimientos a una casilla en cualquier dirección.

```python
def get_king_moves(self, x, y, start_board, end_board): # Rey
    moves = []
    target_board = self.boards[end_board] # Tablero objetivo
    king_moves = [
        (1, 0), (0, 1), (-1, 0), (0, -1), # Movimientos en cruz
        (1, 1), (1, -1), (-1, 1), (-1, -1) # Movimientos en diagonal
    ]

    for dx, dy in king_moves: # Se recorren los movimientos del rey
        nx, ny = x + dx, y + dy # Se calcula la nueva posición
        if 0 <= nx < 8 and 0 <= ny < 8: # Si la nueva posición esta dentro del limite del tablero
            target_piece = target_board[nx][ny] # Se toma la pieza en la nueva posición
            if not target_piece or target_piece[0] != self.current_turn: # Si no hay pieza o la pieza no es del mismo color
                moves.append((nx, ny)) # Se agrega la casilla a los movimientos posibles

    return moves # Se retornan los movimientos posibles
```

3. **Funciones Auxiliares:**

Descripción General

El método `get_sliding_moves` calcula los movimientos deslizantes posibles de una pieza (como torres, alfiles y la reina) en un tablero de ajedrez de Alicia. Permite evaluar todas las posiciones alcanzables en una dirección específica hasta que la pieza alcanza un límite o encuentra una obstrucción.

Parámetros:

- `x, y`: Coordenadas actuales de la pieza.
- `start_board`: (No utilizado actualmente) tablero inicial.
- `end_board`: Tablero en el que se calcularán los movimientos.
- `directions`: Lista de direcciones de movimiento específicas para la pieza.

Proceso:

- La función recorre cada dirección en `directions` (e.g., diagonales, verticales, horizontales).
- Para cada dirección `(dx, dy)`:
    - Avanza casilla por casilla en esa dirección (`nx, ny`).
    - **Verifica límites**: Comprueba si `(nx, ny)` está dentro del tablero.
    - **Verifica colisiones**:
        - Si hay una pieza del oponente, agrega la casilla como válida y detiene el avance.
        - Si hay una pieza propia, detiene el avance.
        - Si no hay pieza, agrega la casilla como válida y sigue avanzando.

Retorno:

- Devuelve una lista de coordenadas válidas `(nx, ny)`.

```python
def get_sliding_moves(self, x, y, start_board, end_board, directions): # Movimientos deslizantes
    moves = [] # Lista de movimientos
    target_board = self.boards[end_board] # Tablero objetivo

    for dx, dy in directions: # Se recorren las direcciones
        nx, ny = x + dx, y + dy # Se calcula la nueva posición
        while 0 <= nx < 8 and 0 <= ny < 8: # Mientras la nueva posición este dentro del limite del tablero
            target_piece = target_board[nx][ny] # Se toma la pieza en la nueva posición
            if target_piece: # Si hay una pieza
                if target_piece[0] != self.current_turn: # Si la pieza no es del mismo color
                    moves.append((nx, ny)) # Se agrega la casilla a los movimientos posibles
                break # Se rompe el ciclo
            moves.append((nx, ny)) # Se agrega la casilla a los movimientos posibles
            nx += dx # Se actualiza la posición en x
            ny += dy # Se actualiza la posición en y

    return moves
```

Piezas que Utilizan la Función

- **Alfil**: Direcciones diagonales: `[(1, 1), (1, -1), (-1, 1), (-1, -1)]`.
- **Torre**: Direcciones verticales y horizontales: `[(1, 0), (0, 1), (-1, 0), (0, -1)]`.
- **Reina**: Combina direcciones del alfil y la torre.

MINIMAX

```python
def minimax(self, game, depth, maximizing_player): # Algoritmo Minimax
    if depth == 0: # Si la profundidad es 0
        return self.evaluate_board(game), None # Se retorna la evaluación del tablero y ningun movimiento

    best_move = None # Mejor movimiento (ninguno)
    if maximizing_player: # Si es el jugador que maximiza
        max_eval = float('-inf') # Valor de evaluación máximo (-infinito)
        for move in self.get_all_moves(game, "white"): # Se recorren los movimientos posibles de las blancas
            start_x, start_y, end, start_board, end_board = move # Se obtienen los datos del movimiento (posición inicial, posición final, tablero
            cloned_game = copy.deepcopy(game) # Se clona el juego
            cloned_game.move_piece((start_x, start_y), end, start_board, end_board) # Se realiza el movimiento en el juego clonado
            eval, _ = self.minimax(cloned_game, depth - 1, False) # Se obtiene la evaluación del tablero y se le resta 1 a la profundidad
            if eval > max_eval: # Si la evaluación es mayor al valor máximo
                max_eval = eval # Se actualiza el valor máximo
                best_move = move # Se actualiza el mejor movimiento
        return max_eval, best_move # Se retorna el valor máximo y el mejor movimiento
    else: # Si es el jugador que minimiza
        min_eval = float('inf') # Valor de evaluación mínimo (infinito)
        for move in self.get_all_moves(game, "black"): # Se recorren los movimientos posibles de las negras
            start_x, start_y, end, start_board, end_board = move # Se obtienen los datos del movimiento (posición inicial, posición final, tablero
            cloned_game = copy.deepcopy(game) # Se clona el juego
            cloned_game.move_piece((start_x, start_y), end, start_board, end_board) # Se realiza el movimiento en el juego clonado
            eval, _ = self.minimax(cloned_game, depth - 1, True) # Se obtiene la evaluación del tablero  y se le resta 1 a la profundidad
            if eval < min_eval: # Si la evaluación es menor al valor mínimo
                min_eval = eval # Se actualiza el valor mínimo
                best_move = move # Se actualiza el mejor movimiento
        return min_eval, best_move # Se retorna el valor mínimo y el mejor movimiento
```

1. Introducción al Algoritmo Minimax

El algoritmo **Minimax** es una técnica de búsqueda utilizada en juegos de dos jugadores como el ajedrez. Permite tomar decisiones óptimas al evaluar posibles movimientos, asumiendo que el jugador rival

también toma las mejores decisiones. El objetivo es **maximizar** el puntaje del jugador actual (jugador MAX) y **minimizar** el puntaje del oponente (jugador MIN).

2. Propósito de la Función

La función `minimax` tiene como propósito evaluar el estado actual del juego hasta una **profundidad dada** y retornar la mejor jugada posible. Esto se logra al explorar todos los movimientos disponibles recursivamente, asignando valores a cada estado del tablero y seleccionando el movimiento con la mejor evaluación.

3. Parámetros de la Función

- `game`: Representación actual del estado del juego.
- `depth`: Profundidad máxima a la que se evaluarán los movimientos (nivel de recursión).
- `maximizing_player`: Un booleano que indica si es el turno del jugador que **maximiza** (True) o del que **minimiza** (False).

4. Funcionamiento Paso a Paso

1. **Caso Base (Profundidad 0)**:
   - Si la profundidad llega a **0**, la función retorna la evaluación del tablero actual usando `self.evaluate_board(game)`. En este punto, no se generan más movimientos.

```
if depth == 0: # Si la profundidad es 0
    return self.evaluate_board(game), None # Se retorna la evaluación del tablero y ningun movimiento
```

2. **Inicialización**:
   - `best_move` se inicializa como `None` (todavía no hay mejor movimiento).

```
best_move = None # Mejor movimiento (ninguno)
```

   - Dependiendo de si es el turno del **maximizador** o **minimizador**, se inicializa una variable de evaluación (`max_eval` o `min_eval`).

```
if maximizing_player: # Si es el jugador que maximiza
    max_eval = float('-inf') # Valor de evaluación máximo (-infinito)
else: # Si es el jugador que minimiza
    min_eval = float('inf') # Valor de evaluación mínimo (infinito)
```

5. Lógica del Jugador Maximizador (Blancas-Humano)

- Se establece `max_eval` como `-infinito` porque buscamos el mayor valor posible.
- Se recorren todos los movimientos legales para las piezas **blancas** usando `self.get_all_moves(game, "white")`.
- Para cada movimiento:
  1. Se **clona** el estado actual del juego (`copy.deepcopy`).
  2. Se realiza el movimiento en el juego clonado.

3. Se llama recursivamente a `minimax` con el nuevo estado del juego, disminuyendo la profundidad en 1 y cambiando a **jugador minimizador**.
4. Se compara la evaluación obtenida con `max_eval` y, si es mayor, se actualiza el valor de `max_eval` y `best_move`.

```python
if maximizing_player: # Si es el jugador que maximiza
    max_eval = float('-inf') # Valor de evaluación máximo (-infinito)
    for move in self.get_all_moves(game, "white"): # Se recorren los movimientos posibles de las blancas
        start_x, start_y, end, start_board, end_board = move # Se obtienen los datos del movimiento (posición inicial, posición final, tablero A
        cloned_game = copy.deepcopy(game) # Se clona el juego
        cloned_game.move_piece((start_x, start_y), end, start_board, end_board) # Se realiza el movimiento en el juego clonado
        eval, _ = self.minimax(cloned_game, depth - 1, False) # Se obtiene la evaluación del tablero y se le resta 1 a la profundidad
        if eval > max_eval: # Si la evaluación es mayor al valor máximo
            max_eval = eval # Se actualiza el valor máximo
            best_move = move # Se actualiza el mejor movimiento
    return max_eval, best_move # Se retorna el valor máximo y el mejor movimiento
```

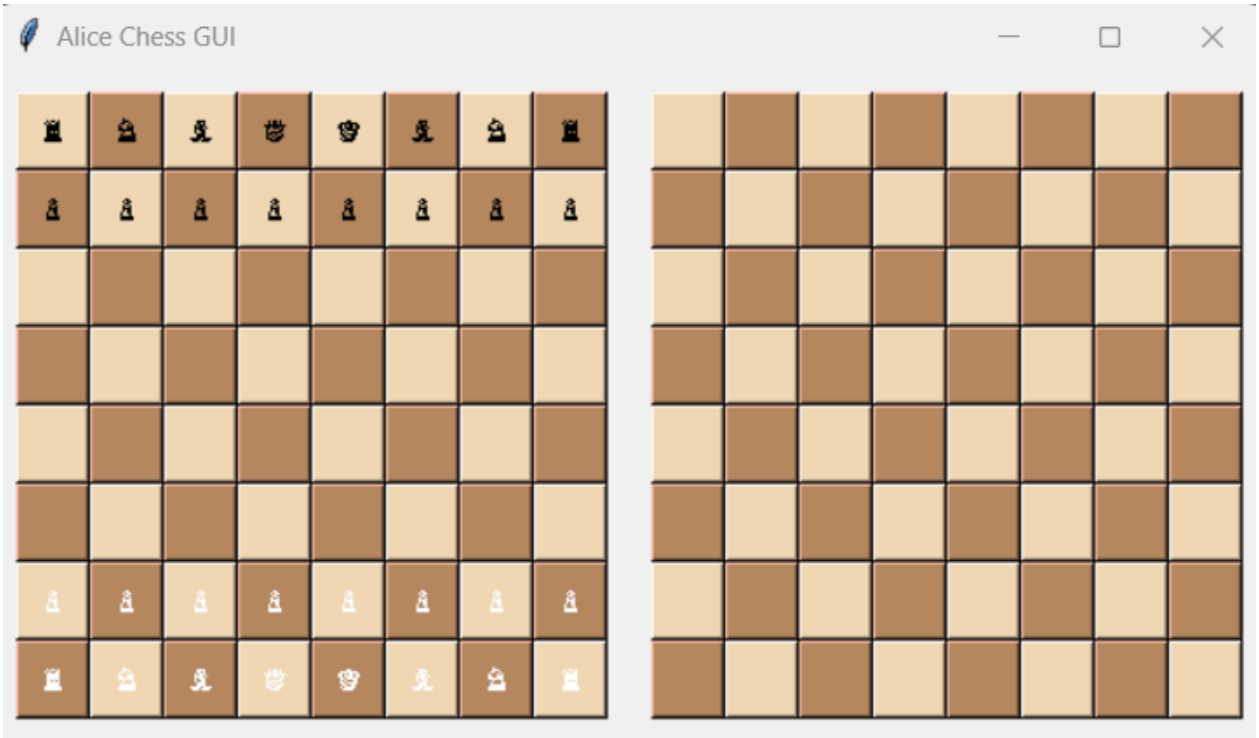
6. Lógica del Jugador Minimizador (Negras-I.A)

- Se establece `min_eval` como **`+infinito`** porque buscamos el menor valor posible.
- Se recorren todos los movimientos legales para las piezas **negras**.
- El proceso es similar al jugador maximizador, pero se busca el **menor** valor de evaluación:
    1. Se clona el juego y se realiza el movimiento.
    2. Se llama recursivamente a `minimax` cambiando a **jugador maximizador**.
    3. Se actualiza `min_eval` y `best_move` si se encuentra una evaluación menor.

Al finalizar la evaluación recursiva, la función retorna:

- El **valor de evaluación** del mejor movimiento encontrado.
- El **mejor movimiento** en sí.

Interfaz



1. Descripción General

La clase `AliceChessGUI` representa la interfaz gráfica para el juego de ajedrez de Alicia, utilizando la biblioteca **Tkinter**. El juego se caracteriza por su mecánica distintiva, con dos tableros (A y B), entre los cuales las piezas pueden moverse de manera especial. La clase incorpora la lógica necesaria para visualizar el tablero, realizar movimientos y ejecutar jugadas de la inteligencia artificial.

2. Componentes Principales de la Clase

1. **Atributos del Constructor (`__init__`)**

```python
class AliceChessGUI:  # Se define la clase para la interfaz gráfica del ajedrez de Alicia
    def __init__(self, root):  # Constructor de la clase, se pasa el objeto root de Tkinter
        self.root = root  # Asigna la ventana principal de la interfaz
        self.game = AliceChess()  # Crea una instancia del juego de ajedrez
        self.ai = AliceAI(depth=2)  # Crea una instancia de la IA con una profundidad de búsqueda de 2
        self.selected_piece = None  # Inicializa la pieza seleccionada
        self.selected_board = None  # Inicializa el tablero seleccionado
```

   - **`self.root`**: Ventana principal de la aplicación.
   - **`self.game`**: Instancia de la clase que maneja la lógica del juego (AliceChess).
   - **`self.ai`**: Instancia de la inteligencia artificial (`AliceAI`), configurada con una profundidad de búsqueda de `2`.

- **`self.selected_piece`** y **`self.selected_board`**: Variables para almacenar la pieza y el tablero seleccionados por el usuario.

2. **Configuración de los Tableros**

```
# Diccionario que contiene los frames de los tableros A y B
self.board_frames = {
    "A": tk.Frame(root),
    "B": tk.Frame(root)
}
```

   - Se crean dos tableros (`A` y `B`), cada uno en un **Frame** de Tkinter.
   - Las celdas de los tableros (8x8) se construyen mediante un bucle doble anidado, usando **Labels** con colores alternados según la posición.

3. **Diccionarios de Almacenamiento**

```
# Configurar tableros
for board_key in self.board_frames:  # Recorre los tableros A y B
    self.board_frames[board_key].grid(row=0, column=0 if board_key == "A" else 1, padx=10, pady=10)
    # Asigna los tableros en la misma fila pero en columnas diferentes (A en 0, B en 1)

# Diccionario para almacenar las celdas de cada tablero
self.cells = {
    "A": {},
    "B": {}
}

# Crear las celdas de cada tablero (8x8)
for board_key in self.board_frames:
    for x in range(8):  # Recorre las filas
        for y in range(8):  # Recorre las columnas
            cell = tk.Label(self.board_frames[board_key], width=4, height=2, bg=self.get_cell_color(x, y), relief="raised")
            # Crea una celda con color dependiendo de la posición
            cell.grid(row=x, column=y)  # Coloca la celda en la cuadrícula
            cell.bind("<Button-1>", lambda e, bx=x, by=y, board=board_key: self.cell_clicked(bx, by, board))
            # Asocia un evento para hacer clic en la celda
            self.cells[board_key][(x, y)] = cell  # Almacena la celda en el diccionario

self.update_board()  # Actualiza la visualización del tablero
```

   - **`self.board_frames`**: Contiene los dos tableros en forma de `Frame`.
   - **`self.cells`**: Guarda las celdas de cada tablero, permitiendo un acceso fácil a sus configuraciones.

3. Métodos Importantes

1. **`get_cell_color(x, y)`**
   - Devuelve el color de una celda según su posición: blanco (#F0D9B5) o marrón (#B58863), emulando un tablero de ajedrez clásico.

```
def get_cell_color(self, x, y):  # Determina el color de la celda
    return "#F0D9B5" if (x + y) % 2 == 0 else "#B58863"  # Si la suma de las coordenadas es par, es blanco, sino es marrón
```

2. **`update_board()`**
    - Actualiza la visualización de ambos tableros.
    - Utiliza un diccionario de símbolos para representar las piezas (por ejemplo, ♙ para peón).
    - Configura el color del texto según el color de la pieza (blanco o negro).

```
def update_board(self):  # Actualiza el tablero visualmente
    piece_symbols = {
        "P": "♙", "N": "♘", "B": "♗", "R": "♖", "Q": "♕", "K": "♔"  # Diccionario con los símbolos de las piezas
    }

    for board_key, board in self.game.boards.items():  # Recorre los tableros del juego
        for x, row in enumerate(board):  # Recorre las filas del tablero
            for y, cell in enumerate(row):  # Recorre las columnas
                if cell:  # Si hay una pieza en la celda
                    color, piece = cell  # Obtiene el color y tipo de pieza
                    symbol = piece_symbols[piece]  # Obtiene el símbolo de la pieza
                    self.cells[board_key][(x, y)].config(text=symbol, fg="white" if color == "white" else "black")  # Actualiza la celda
                else:  # Si no hay una pieza
                    self.cells[board_key][(x, y)].config(text="")  # Limpia la celda
```

3. **`show_valid_moves(valid_moves, board)`**
    - Resalta las celdas correspondientes a los movimientos válidos en color verde claro (#90EE90).

```
def show_valid_moves(self, valid_moves, board):  # Muestra los movimientos válidos en el tablero
    for x, y in valid_moves:  # Recorre las posiciones de los movimientos válidos
        self.cells[board][(x, y)].config(bg="#90EE90")  # Cambia el color de la celda a verde claro
```

4. **`reset_highlights()`**
    - Restaura los colores originales de todas las celdas de ambos tableros.

```
def reset_highlights(self):  # Restablece los resaltados de las celdas
    for board_key in self.cells:  # Recorre ambos tableros
        for x, y in self.cells[board_key]:  # Recorre todas las celdas
            self.cells[board_key][(x, y)].config(bg=self.get_cell_color(x, y))  # Restaura el color original de la celda
```

5. **`cell_clicked(x, y, board)`**
    - Lógica principal para manejar los clics en las celdas.
    - **Funcionalidades**:
        - Resalta movimientos válidos cuando el usuario selecciona una pieza.
        - Verifica y realiza movimientos válidos.
        - Restablece la selección después de un movimiento exitoso.
        - Comprueba si es el turno de la inteligencia artificial.

```python
def cell_clicked(self, x, y, board):  # Función que se llama al hacer clic en una celda
    self.reset_highlights()  # Restaura el color original de las celdas
    cell = self.game.boards[board][x][y]  # Obtiene la pieza en la celda seleccionada

    if self.selected_piece and (self.selected_board == board):  # Si ya hay una pieza seleccionada en el tablero actual
        # Intenta mover la pieza seleccionada
        target_board = "A" if board == "B" else "B"  # Determina el tablero objetivo (opuesto al actual)
        if (x, y) in self.game.get_legal_moves(self.selected_piece, board, target_board):
            # Si el movimiento es válido
            self.game.move_piece(self.selected_piece, (x, y), self.selected_board, target_board)  # Mueve la pieza
            self.selected_piece = None  # Resetea la pieza seleccionada
            self.selected_board = None  # Resetea el tablero seleccionado
            self.update_board()  # Actualiza el tablero
            self.check_ai_turn()  # Verifica si es el turno de la IA
            return

    # Si se selecciona una pieza propia
    if cell and cell[0] == self.game.current_turn:
        self.selected_piece = (x, y)  # Establece la pieza seleccionada
        self.selected_board = board  # Establece el tablero seleccionado
        valid_moves = self.game.get_legal_moves((x, y), board, "A" if board == "B" else "B")  # Obtiene los movimientos válidos
        self.show_valid_moves(valid_moves, "A" if board == "B" else "B")  # Muestra los movimientos válidos
```

6. **`check_ai_turn()`**
    - Si el turno actual corresponde a la IA (piezas negras), llama al método `ai_move()` después de una breve pausa (500 ms).

```python
def check_ai_turn(self):  # Verifica si es el turno de la IA
    if self.game.current_turn == "black":  # Si es el turno de la IA (negras)
        self.root.after(500, self.ai_move)  # Espera 500ms y luego realiza el movimiento de la IA
```

7. **`ai_move()`**
    - Utiliza el algoritmo **Minimax** de la clase `AliceAI` para calcular el mejor movimiento.
    - Ejecuta el movimiento seleccionado y actualiza el tablero.

```python
def ai_move(self):  # Realiza el movimiento de la IA
    _, best_move = self.ai.minimax(self.game, self.ai.depth, maximizing_player=False)  # Obtiene el mejor movimiento de la IA
    if best_move:  # Si hay un movimiento válido
        start_x, start_y, end, start_board, end_board = best_move  # Descompone el movimiento
        self.game.move_piece((start_x, start_y), end, start_board, end_board)  # Mueve la pieza de la IA
        self.update_board()  # Actualiza el tablero
```