Documentación del Proyecto de Visualizador de Algoritmos de IA

Descripción General

Este proyecto implementa una aplicación gráfica para visualizar y demostrar el funcionamiento de diferentes algoritmos de Inteligencia Artificial en tres contextos diferentes:

1. **FrozenLake**: Implementación de algoritmos de búsqueda BFS y DFS en un entorno de rejilla.
2. **Grafo de Rumania**: Visualización de algoritmos de búsqueda de caminos (A*, Greedy, Hill Climbing) en un grafo.
3. **Gato Minimax**: Implementación del algoritmo Minimax en un juego de Tic Tac Toe.

Estructura de la Aplicación

La aplicación está construida con Python usando Tkinter para la interfaz gráfica y consta de una clase principal `MainApp` que maneja todas las funcionalidades.

Clase Principal: `MainApp`

Inicialización y Configuración

`__init__(self, root)`

Descripción: Constructor de la clase principal que inicializa la interfaz y todas las variables necesarias. **Parámetros**:

- `root` : Objeto raíz de Tkinter que contiene la ventana principal. **Funcionalidad**:
- Inicializa la ventana principal con tamaño y título.
- Configura variables para cada componente (FrozenLake, Grafo, Gato).
- Crea los frames para cada pantalla.
- Llama a los métodos de inicialización.

`centrar_ventana(self)`

Descripción: Centra la ventana en la pantalla. **Funcionalidad**:

- Calcula las dimensiones de la ventana y la pantalla.
- Posiciona la ventana en el centro de la pantalla.

`mostrar_pantalla(self, pantalla)`

Descripción: Cambia entre las diferentes pantallas de la aplicación. **Parámetros**:

- `pantalla` : Frame que se debe mostrar. **Funcionalidad**:
- Oculta todas las pantallas.
- Muestra la pantalla seleccionada.
- Si es la pantalla de FrozenLake, inicializa el entorno si no existe.

`on_close(self)`

Descripción: Maneja el evento de cierre de la ventana. **Funcionalidad**:

- Muestra un diálogo de confirmación.
- Si se confirma, cierra la aplicación.

`create_layout(self)`

Descripción: Crea la interfaz gráfica completa. **Funcionalidad**:

- Configura el estilo para botones.
- Crea el menú principal con los botones para cada sección.
- Configura los elementos de la pantalla de FrozenLake.
- Configura los elementos de la pantalla del Grafo de Rumania.
- Llama a la función para crear el juego de Gato.

Funciones de FrozenLake

`initialize_frozen_lake(self)`

Descripción: Inicializa el entorno de FrozenLake. **Funcionalidad**:

- Crea un nuevo entorno de FrozenLake usando Gymnasium.
- Reinicia el entorno y obtiene el estado inicial.
- Actualiza la visualización.

`update_frozen_lake_display(self)`

Descripción: Actualiza la visualización del entorno FrozenLake. **Funcionalidad**:

- Obtiene la representación visual del entorno.
- Usa matplotlib para mostrar la imagen en el canvas.

`solve(self)`

Descripción: Ejecuta el algoritmo seleccionado para resolver FrozenLake. **Funcionalidad**:

- Verifica qué algoritmo está seleccionado (BFS o DFS).
- Llama a la función correspondiente para resolver el problema.

`bfs_solve(self)`

Descripción: Inicia la búsqueda de una solución usando BFS. **Funcionalidad**:

- Verifica si ya está resolviendo.
- Actualiza el texto de estado.
- Inicia un hilo separado para ejecutar BFS.

`_bfs_worker(self)`

Descripción: Implementa el algoritmo BFS para FrozenLake. **Funcionalidad**:

- Reinicia el entorno.
- Obtiene la estructura del mapa.
- Ejecuta el algoritmo BFS para encontrar el camino desde el inicio hasta la meta.
- Actualiza la interfaz con el resultado.

`dfs_solve(self)`

Descripción: Inicia la búsqueda de una solución usando DFS. **Funcionalidad**:

- Verifica si ya está resolviendo.
- Actualiza el texto de estado.
- Inicia un hilo separado para ejecutar DFS.

`_dfs_worker(self)`

Descripción: Implementa el algoritmo DFS para FrozenLake. **Funcionalidad**:

- Reinicia el entorno.
- Obtiene la estructura del mapa.
- Ejecuta el algoritmo DFS para encontrar el camino desde el inicio hasta la meta.
- Actualiza la interfaz con el resultado.

`execute_solution(self)`

Descripción: Ejecuta la solución encontrada paso a paso. **Funcionalidad**:

- Reinicia el contador de pasos.
- Inicia la ejecución del primer paso.

`_execute_next_step(self)`

Descripción: Ejecuta el siguiente paso de la solución. **Funcionalidad**:

- Verifica si quedan pasos por ejecutar.
- Ejecuta la acción correspondiente al paso actual.
- Actualiza la visualización.
- Programa la ejecución del siguiente paso después de un retardo.
- Actualiza el estado cuando se completa la solución.

Funciones del Grafo de Rumania

`load_file(self)`

Descripción: Carga un archivo de texto con la estructura del grafo. **Funcionalidad**:

- Abre el archivo "graforumania.txt".
- Interpreta cada línea como una arista del grafo.
- Crea el grafo con NetworkX.
- Calcula las posiciones de los nodos.
- Dibuja el grafo inicial.

`draw_graph(self)`

Descripción: Dibuja el grafo sin rutas resaltadas. **Funcionalidad**:

- Limpia el canvas.

- Dibuja el grafo con NetworkX.
- Añade etiquetas a las aristas mostrando los pesos.
- Actualiza el canvas.

`run_algorithm(self)`

Descripción: Ejecuta el algoritmo seleccionado en el grafo. **Funcionalidad**:

- Obtiene el algoritmo seleccionado (A*, Greedy, Hill Climbing).
- Ejecuta el algoritmo correspondiente.
- Visualiza el camino encontrado.
- Muestra un mensaje de éxito o error.

`greedy_algorithm(self, start_node, end_node)`

Descripción: Implementa el algoritmo voraz (Greedy). **Parámetros**:

- `start_node` : Nodo de inicio.
- `end_node` : Nodo de destino. **Funcionalidad**:
- Comienza en el nodo inicial.
- En cada paso, elige el vecino no visitado con menor peso.
- Termina cuando llega al nodo destino o cuando no hay más caminos.
- Retorna el camino encontrado.

`hill_climbing_algorithm(self, start_node, end_node)`

Descripción: Implementa el algoritmo de Hill Climbing. **Parámetros**:

- `start_node` : Nodo de inicio.
- `end_node` : Nodo de destino. **Funcionalidad**:
- Calcula heurísticas si no existen.
- Comienza en el nodo inicial.
- En cada paso, elige el vecino no visitado con mejor heurística.
- Termina cuando llega al nodo destino o cuando no hay más caminos.
- Retorna el camino encontrado.

`calculate_heuristics(self, goal_node)`

Descripción: Calcula la heurística para cada nodo hasta el objetivo. **Parámetros**:

- `goal_node` : Nodo destino para calcular las heurísticas. **Funcionalidad**:
- Para cada nodo, calcula la distancia euclidiana directa hasta el nodo destino.
- Almacena estas distancias como heurísticas.

`visualize_path(self, path)`

Descripción: Visualiza el camino encontrado en el grafo. **Parámetros**:

- `path` : Lista de nodos que forman el camino. **Funcionalidad**:
- Limpia el grafo dibujándolo de nuevo.
- Dibuja las aristas del camino en rojo.
- Resalta los nodos del camino en rojo.
- Actualiza el canvas.

Funciones del Juego de Gato (Tic Tac Toe)

`create_tictactoe_game(self)`

Descripción: Crea la interfaz del juego de Gato. **Funcionalidad**:

- Configura el frame principal del juego.
- Inicializa variables del juego.
- Crea el tablero con botones.
- Configura elementos de la interfaz (estado, botones).

`make_move(self, row, col)`

Descripción: Realiza un movimiento en el tablero. **Parámetros**:

- `row` : Fila donde se hizo el movimiento.
- `col` : Columna donde se hizo el movimiento. **Funcionalidad**:
- Verifica si el movimiento es válido.
- Actualiza el tablero y la interfaz.
- Verifica si el juego ha terminado.
- Cambia al turno de la IA.
- Llama a la función para que la IA haga su movimiento.

`ai_move(self)`

Descripción: Realiza el movimiento de la IA usando Minimax. **Funcionalidad**:

- Evalúa todos los movimientos posibles.
- Elige el mejor movimiento según Minimax.
- Actualiza el tablero y la interfaz.
- Verifica si el juego ha terminado.
- Cambia al turno del jugador.

`minimax(self, board, depth, is_maximizing)`

Descripción: Implementa el algoritmo Minimax. **Parámetros**:

- `board` : Estado actual del tablero.
- `depth` : Profundidad actual en la búsqueda.
- `is_maximizing` : Booleano que indica si es el turno de maximizar o minimizar. **Funcionalidad**:
- Verifica si hay un estado terminal (victoria/derrota/empate).
- Si es turno de maximizar, busca el movimiento con el mayor valor.
- Si es turno de minimizar, busca el movimiento con el menor valor.
- Devuelve el valor del mejor movimiento.

`check_winner(self)`

Descripción: Verifica si hay un ganador en el tablero actual. **Funcionalidad**:

- Llama a `check_winner_board` con el tablero actual y el jugador actual.

`check_winner_board(self, board, player)`

Descripción: Verifica si hay un ganador en un tablero dado. **Parámetros**:

- `board` : Tablero a verificar.
- `player` : Jugador para el que se verifica la victoria. **Funcionalidad**:
- Verifica las filas, columnas y diagonales.
- Devuelve True si el jugador ha ganado, False de lo contrario.

`is_board_full(self)`

Descripción: Verifica si el tablero está lleno. **Funcionalidad**:

- Llama a `is_board_full_board` con el tablero actual.

`is_board_full_board(self, board)`

Descripción: Verifica si un tablero dado está lleno. **Parámetros**:

- `board` : Tablero a verificar. **Funcionalidad**:
- Revisa si queda alguna celda vacía.
- Devuelve True si el tablero está lleno, False de lo contrario.

`reset_game(self)`

Descripción: Reinicia el juego de gato. **Funcionalidad**:

- Reinicia las variables del juego.
- Limpia el tablero en la interfaz.
- Actualiza el estado del juego.

Función Principal

`if __name__ == "__main__":`

Descripción: Punto de entrada del programa. **Funcionalidad**:

- Crea la ventana principal de Tkinter.
- Instancia la clase MainApp.
- Inicia el bucle principal de la aplicación.