Estructura General

El sistema está organizado en módulos siguiendo buenas prácticas de ingeniería de software, el cual se compone de **Agente.py,** la cual nos ayudan implementar el agente inteligente y su respectiva lógica. **Laberinto.py** nos ayuda a gestionar la estructura del laberinto y como este se crea, todo esto en la carpeta Core.

Por otro lado tenemos **Busqueda.py** la cual contiene la logica de los respectivos algoritmos (BFS, DFS, A*, IDS) y **visualizacion.py** el cual nos permite visualizar el árbol de búsqueda que hace el agente, todo esto en la carpeta Algoritmos.

Y por ultimo **gui.py** el cual como su nombre indica es la interfaz gráfica donde podemos interactuar con la aplicación.

Componentes Principales

Clase Nodo

Representa un estado dentro del espacio de búsqueda. Se compone de:

- **`estado`:** posición (fila, columna)
- **`padre`:** nodo anterior (para reconstrucción del camino)
- **`acción`:** movimiento que generó el nodo
- **`costo`:** acumulado hasta ese punto

¿Por que usar nodos en vez de tuplas como fila y columna?
Esto tiene una explicación bastante simple, usar las tuplas es como tener una foto de donde estas sin saber como llegaste a ese punto. En cambio si usamos nodos tenemos un registro completo de como llegamos a ese punto (Pues tenemos los anteriores componentes) y se hace mas fácil el gráfico del árbol de búsqueda.

Se sobrecarga el método **`__lt__`**para que los nodos se puedan comparar en estructuras como **`heapq`** (usado en A*).

Funciones Clave

- **`reconstruir_camino`:** Recorre nodos hacia atrás para reconstruir el camino desde la meta al origen.
- **`acciones_validas`:** Genera posibles movimientos según el mapa del laberinto.
- **`elegir_algoritmo`:** Ejecuta el algoritmo seleccionado dinámicamente.
- **`agente_atrapado`:** Evalúa si el agente está sin movimientos válidos.
- **`sugerir_algoritmo`:** Sugiere el algoritmo más eficiente según el contexto actual.

- **dibujar_laberinto:** realiza el laberinto y el agente (gui.py)
- **dibujar_panel**: dibuja el panel de control y maneja las interacciones (gui.py)
- **dibujar_arbol_busqueda**: mostrar el arbol de busqueda (gui.py)
- **actuar**: Logica principal del agente (verifica si llego a la meta, si está atrapado, cambia de algoritmo, etc.) (agente.py).
- **cambiar_algoritmo**: (agente.py).
- **reiniciar**: restablece el estado del agente (agente.py).
- **generar_laberinto**: crea un laberinto con una densidad configurable (laberinto.py)
- **asegurar_camino**: verifica la conectividad de los caminos (laberinto.py)
- **calcular_situacion**: evalúa el entorno del agente (laberinto.py)

Estructuras de Datos Utilizadas

- **`deque`**: Para las colas FIFO en BFS y pilas LIFO en DFS.
- **`heapq`**: Cola de prioridad eficiente usada por A* para seleccionar el nodo con menor costo heurístico.
- **`set`**: Para registrar estados ya visitados y evitar ciclos.
- **`dict`**: En la visualización, se usa para mapear conexiones padre-hijo.
- **`nx.DiGraph`**: Grafo dirigido que modela el árbol de búsqueda para la visualización.

Librerías Usadas

- **Pygame** → Librería para crear interfaces gráficas y simulaciones interactivas en 2D, ideal para visualizar algoritmos de forma animada.
- **Networkx** → Librería para crear, manipular y analizar grafos, utilizada para implementar y visualizar algoritmos como BFS, DFS o A*.
- **matplotlib** →Librería de visualización 2D que permite graficar datos y representar grafos junto con NetworkX de forma estática y clara.
- **Numpy** → Librería para cálculos numéricos eficientes con vectores y matrices, útil en algoritmos que requieren operaciones matemáticas rápidas.
- **Pydot** → Librería que permite crear y exportar grafos en formato DOT para visualizarlos con Graphviz de forma estructurada y profesional.
- **Pygraphviz** → Interfaz de Python para Graphviz que permite crear y renderizar grafos con control preciso sobre su disposición visual.

Pruebas Realizadas

Escenarios de Evaluación

Se probaron múltiples mapas diseñados para simular distintos grados de complejidad:

Escenario	Características	Objetivo
1. Básico	Camino directo sin obstáculos	Evaluar rendimiento base
2. Bifurcaciones	Múltiples caminos posibles	Verificar capacidad de exploración
3. Trampa	Solo una salida válida	Medir resiliencia y lógica de escape
4. Obstáculos Dinámicos	Cambios en tiempo real	Evaluar adaptabilidad y planificación
5. Laberinto Grande	Gran cantidad de nodos	Evaluar rendimiento y memoria

Variables Probadas

- Tamaño del laberinto
- Posición inicial y objetivo
- Posicionamiento dinámico de obstáculos
- Activación del cambio automático de algoritmos

Comportamiento del Agente en Pruebas

Algoritmo	Tiempo Promedio	Nodos Explorados	Camino Óptimo	Comportamiento
A*	Bajo	Bajo/medio	Si	Inteligente y guiado
BFS	Medio	Alto	Si	Explotación completa
DFS	Bajo	Bajo	Medio	Explora rápido, pero arriesga
IDS	Alto	Medio/Alto	SI	Eficiente en memoria, más lento

Modo Adaptativo

- El agente cambió automáticamente entre algoritmos según la situación del laberinto.
- Se evaluó con éxito en escenarios donde se bloqueaban caminos intermedios.
- En todos los casos, el agente fue capaz de recuperarse y hallar soluciones sin intervención manual.

Visualización del Árbol de Búsqueda

El uso de **`VisualizadorArbol`** permitió representar gráficamente el proceso de cada algoritmo:

- Cada nodo es un estado del agente.
- Se visualiza la expansión de caminos, bifurcaciones y retrocesos.
- Útil para debug, análisis educativo o investigación.

Conclusión

La implementación del agente inteligente mediante un sistema adaptable basado en múltiples algoritmos de búsqueda proporciona:

- Robustez en escenarios dinámicos
- Flexibilidad para distintas condiciones
- Visualización clara del razonamiento del agente

Este diseño no solo resuelve el problema actual del laberinto, sino que también es fácilmente escalable para otros entornos mas complejos.