Informe: Laberinto Saltarines // Tarea 1
Alumno: Diego Ignacio Perez Torres
Profesor: Julio Erasmo Godoy Del Campo

(en el readme esta aun mas resumido el informe)

Descripción de funciones:

Archivo: utils.py

read_labyrinth_file(file_path)
Lee un archivo con especificaciones de laberintos y devuelve una lista de objetos Laberinto.
Abre el archivo y lee las dimensiones, posiciones inicial y destino, y la matriz de números.
Crea objetos Laberinto usando cargar_laberinto_desde_matriz.
Finaliza la lectura al encontrar una línea con "0".

Archivo: interfaz.py

__init__(self, laberinto, solucion)
Inicializa la interfaz gráfica configurando la ventana, colores y almacenando el laberinto y su solución.

dibujar_laberinto(self, pantalla)
Dibuja el laberinto en la ventana de Pygame, coloreando las celdas según su tipo (inicio, destino o normal) y mostrando los números.

animar_solucion(self)
Anima el recorrido de la solución resaltando las celdas paso a paso, pausando entre cada movimiento y cerrando la ventana al finalizar.

Archivo: laberinto.py

–Laberinto (Clase)
__init__(self, filas, columnas, inicio, destino, matriz)
Inicializa un objeto Laberinto almacenando dimensiones, posiciones y la matriz de números.

movimientos_validos(self, posicion)
Calcula y devuelve una lista de movimientos válidos desde una posición, usando el número saltarín de la celda y respetando los límites del laberinto.

cargar_laberinto_desde_matriz(cls, matriz)
Crea un objeto Laberinto a partir de una matriz, extrayendo y validando las dimensiones, posiciones inicial y destino.

Archivo: busquedas.py

–Nodo (Clase)
Representa un nodo en el árbol de búsqueda, almacenando su posición, valor, lista de hijos y nodos visitados.

Método agregar_hijo(nodo_hijo): Agrega un nodo hijo a la lista de hijos.

arbol_busqueda(Laberinto)
Esta función construye un árbol de búsqueda a partir de un laberinto. El laberinto se representa mediante un objeto Laberinto

Funcionamiento

Nodo raíz:
Se crea el nodo raíz a partir de la posición inicial del laberinto.
Se inicializa la lista de nodos visitados con la posición inicial.

Construcción del árbol:
Durante la construcción del árbol de búsqueda, se utiliza una lista llamada nodos_nivel_actual para procesar los nodos de cada nivel. El procedimiento sigue estos pasos:

Para cada nodo en el nivel actual, se calculan todos los movimientos válidos desde su posición.

-Por cada movimiento válido:
Si la posición aún no ha sido visitada en la misma rama, se crea un nuevo nodo hijo.
Se actualiza la lista de nodos visitados del hijo para reflejar el camino recorrido.
El nuevo hijo se agrega tanto al nodo actual como a la lista de nodos del siguiente nivel.

Una vez procesados todos los nodos del nivel actual, se continúa con los nodos del siguiente nivel, avanzando progresivamente en la construcción del árbol.

Para evitar la repetición de nodos en una misma rama, cada nodo cuenta con un atributo nodos_visitados. Esta lista almacena todas las posiciones que han sido recorridas en el camino hasta ese nodo, asegurando que no se formen ciclos ni se dupliquen pasos en una misma trayectoria.

Devuelve el nodo raíz del árbol de búsqueda.

dfs(arbol_busqueda, destino)
Esta función realiza una búsqueda en profundidad (DFS) en el árbol de búsqueda para encontrar un camino desde la raíz hasta un nodo que coincida con la posición destino.

El procedimiento es el siguiente:

Inicialización:
Se utiliza una pila (stack) para gestionar los nodos a explorar. Cada elemento de la pila es una tupla que contiene el nodo_actual y el camino recorrido desde la raíz hasta ese nodo.

Búsqueda:
Mientras la pila tenga elementos:

- Se extrae el último nodo agregado (siguiendo el orden LIFO).
- Si el nodo actual corresponde al destino, se devuelve el camino completo.
- Si no, se agregan todos los hijos del nodo actual a la pila, cada uno con su respectivo camino actualizado.

Retorno:
Si al finalizar la búsqueda no se encuentra el destino, se devuelve una lista vacía.

bfs(arbol_busqueda, destino)
Esta función implementa una búsqueda en anchura (BFS) sobre el árbol de búsqueda para encontrar un camino desde la raíz hasta un nodo que coincida con la posición destino.

El procedimiento es el siguiente:

Inicialización:
Se emplea una cola (queue) para gestionar los nodos a explorar. Cada elemento en la cola es una tupla formada por el nodo_actual y el camino recorrido hasta ese punto.

Búsqueda:
Mientras la cola tenga elementos:

Se extrae el primer nodo agregado (siguiendo el orden FIFO).
Si el nodo actual corresponde al destino, se devuelve el camino completo.
En caso contrario, se añaden los hijos del nodo actual a la cola, actualizando el camino para cada uno.

Retorno:
Si no se encuentra el destino tras explorar todos los nodos, se devuelve una lista vacía.

costo_uniforme(arbol_busqueda, destino)
Simplemente llama a bfs, ya que como todos los movimientos tienen el mismo costo, se puede usar BFS para encontrar el camino de costo uniforme.
Devuelve el camino al destino.

Ejemplo de input/output:

Ejemplo de Entrada

4 4 0 0 3 3
2 1 1 1
3 3 1 2
3 2 2 1
2 1 3 1
0

Ejemplo de Salida

Se han cargado 1 laberintos desde el archivo.

Procesando laberinto 1...
DFS - Número de movimientos: 4
Costo Uniforme - Número de movimientos: 3

Conclusiones:

La implementación inicial del árbol de búsqueda facilitó considerablemente la implementación de los algoritmos de búsqueda, ya que simplemente consistía en aplicar las estructuras FIFO o LIFO sobre la expansión del camino a través de sus nodos hijos. En cuanto a la tupla (nodo, camino), esta resultó ser innecesaria, dado que se podría haber consultado directamente nodo.nodos_visitados; sin embargo, la mantuve en el código debido a que me fue útil para la depuración.

En resumen, aunque la creación del árbol al principio implica un gasto considerable de memoria, resultó ser una herramienta valiosa para visualizar y entender mejor la implementación de cada algoritmo, lo que permitió identificar que las diferencias entre ellos son mínimas, limitándose a una simple instrucción.