Problema 1: Generador de Laberintos con DFS

El prototipo de la función principal es el siguiente:

```
def generate_maze(
    width_in_pixels: int,
    height_in_pixels: int,
    wall_thickness_in_pixels: int,
    number_of_walls: int,
    wall_color=(0, 0, 0),
    path_color=(255, 255, 255),
)
```

Donde:

- width_in_pixels: ancho de la imagen resultante
- height_in_pixels: alto de la imagen resultante
- wall_thickness_in_pixels: grosor de las paredes
- number_of_walls: Factor de bifurcación de las paredes
- wall_color: color de las paredes
- path_color: color de los caminos

1. Configuración y ajuste inicial

1.1 Ajuste de Dimensiones

Primero, se garantiza que las dimensiones sean compatibles con el grosor de las paredes y evitar bordes blancos, de la siguiente forma:

- Verificar que width_in_pixels y height_in_pixels sean múltiplos de wall_thickness_in_pixels.
- Si no lo son, ajustar sumando los píxeles necesarios.
- Calcular el número de celdas: (dimension - wall_thickness) // (2 * wall_thickness).
- Si el número de celdas es par, agregar una celda más (sumar 2 * wall_thickness).

1.2 Cálculo de Tamaños Reales

Se calculan las dimensiones reales de la siguiente forma:

- real_width = n_cells_x * cell_size + wall_thickness_in_pixels
- real_height = n_cells_y * cell_size + wall_thickness_in_pixels

2. Creación de matriz del laberinto

2.1 Creación de la Matriz Base

Se crea una matriz de tamaño (real_height, real_width) llena de paredes (valor 1), y se definen los valores:

- 0 = camino (path_color)
- 1 = pared (wall_color)

2.2 Apertura de Celdas Iniciales

Para cada celda en la cuadrícula, abrir un espacio cuadrado del tamaño de una pared en las coordenadas:

- x0 = x * cell_size + wall_thickness_in_pixels
- y0 = y * cell_size + wall_thickness_in_pixels

3. DFS con control de bifurcaciones

3.1 Inicialización del DFS

Se crea una estructura con las siguientes características:

- visited: Matriz booleana para marcar celdas visitadas.
- stack: Pila para el recorrido DFS.
- directions: Lista de direcciones posibles [(-1,0), (1,0), (0,-1), (0,1)].
- Punto de inicio: Celda (0, 0) marcada como visitada.

3.2 Cálculo de Probabilidad de Seguir Recto

Se utiliza el parámetro number_of_walls como un factor para ramificar las paredes.

- Parámetros:
  - total_cells = n_cells_x * n_cells_y
  - min_walls = max(1, total_cells // 8) (caminos muy rectos)
  - max_walls = total_cells * 2 (caminos muy ramificados)
- Normalización: nw = max(min_walls, min(number_of_walls, max_walls))
- Probabilidad: p_straight = 1.0 - (nw - min_walls) / (max_walls - min_walls + 1e-9) * 0.85
  - number_of_walls bajo → alta probabilidad de seguir recto → caminos largos
  - number_of_walls alto → baja probabilidad de seguir recto → más bifurcaciones

3.3 Proceso de Generación del Laberinto

Este es el proceso principal del DFS. Mientras la pila no esté vacía:

1. Obtener celda actual: (cy, cx), prev_dir = stack[-1]
2. Buscar vecinos no visitados: Explorar las 4 direcciones posibles.
3. Si hay vecinos disponibles, decidir dirección:

- Si hay dirección previa y existe un vecino en esa dirección, usar probabilidad p_straight para decidir si seguir recto.
- Si no hay dirección previa o se decide cambiar, elegir aleatoriamente.
    - Abrir camino: Eliminar la pared entre la celda actual y la vecina elegida.
    - Marcar como visitada: La nueva celda se marca como visitada.
    - Agregar a la pila: stack.append(((ny, nx), (dy, dx)))

4. Si no hay vecinos: Hacer backtracking (stack.pop())

3.4 Apertura de Caminos

- Coordenadas de celdas:
    - Celda actual: (cy, cx)
    - Celda vecina: (ny, nx)
- Cálculo de coordenadas en píxeles:
    - y0 = cy * cell_size + wall_thickness_in_pixels
    - x0 = cx * cell_size + wall_thickness_in_pixels
    - y1 = ny * cell_size + wall_thickness_in_pixels
    - x1 = nx * cell_size + wall_thickness_in_pixels
- Apertura según dirección:
    - Abajo (dy == 1): maze[y0 + wall_thickness_in_pixels : y1, x0 : x0 + wall_thickness_in_pixels] = 0
    - Arriba (dy == -1): maze[y1 + wall_thickness_in_pixels : y0, x0 : x0 + wall_thickness_in_pixels] = 0
    - Derecha (dx == 1): maze[y0 : y0 + wall_thickness_in_pixels, x0 + wall_thickness_in_pixels : x1] = 0
    - Izquierda (dx == -1): maze[y0 : y0 + wall_thickness_in_pixels, x1 + wall_thickness_in_pixels : x0] = 0

4. CREACIÓN DE ENTRADA Y SALIDA

4.1 Selección de Lados

Lados disponibles: [("top", 0), ("bottom", real_height-1), ("left", 0), ("right", real_width-1)]
Selección aleatoria: Elegir dos lados diferentes para entrada y salida.

4.2 Apertura de Aberturas

- Función open_on_side: Abre una abertura en el lado especificado.
- Proceso para cada lado:
    - Top/Bottom: Elegir posición x aleatoria, abrir desde el borde hasta la celda.
    - Left/Right: Elegir posición y aleatoria, abrir desde el borde hasta la celda.
- Coordenadas de apertura:
    - x0 = x * cell_size + wall_thickness_in_pixels
    - y0 = y * cell_size + wall_thickness_in_pixels

5. CONVERSIÓN A IMAGEN

- Matriz RGB: Crear matriz de tamaño (real_height, real_width, 3).
- Asignación de colores:
    - Paredes (maze == 1): img_rgb[maze == 1] = wall_color
    - Caminos (maze == 0): img_rgb[maze == 0] = path_color
- Conversión: Image.fromarray(img_rgb)
- Resultado: Imagen PIL con el laberinto generado.

Problema 2: Camino más corto con BFS

El prototipo de la función principal es el siguiente:

```
def solve_maze(
        path_input_image: str,
        path_output_image: str
)
```

Recibe dos strings, la ruta a la imagen del laberinto y la ruta donde guardar la imagen del laberinto con el camino más corto

FUNCIONAMIENTO DEL PROGRAMA

FUNCIÓN PRINCIPAL: solve_maze()

Parámetros:

- path_input_image: Ruta de la imagen del laberinto a resolver
- path_output_image: Ruta donde se guardará la imagen con la solución

Proceso paso a paso:

1. Carga y validación de imagen
    - Lee la imagen usando cv2.imread()
    - Verifica que la imagen se haya cargado correctamente
    - Convierte la imagen a escala de grises para simplificar el procesamiento
2. Binarización de la imagen
    - Aplica un umbral (wall_threshold = 80) para distinguir paredes de caminos
    - Los píxeles con valor > 80 se consideran caminos libres (valor 1)
    - Los píxeles con valor ≤ 80 se consideran paredes (valor 0)
3. Detección de puntos de entrada/salida
    - Examina todos los bordes de la imagen (superior, inferior, izquierdo, derecho)
    - Identifica píxeles blancos en los bordes que representan posibles entradas/salidas

    - Elimina duplicados para obtener lista única de puntos de borde
4. Búsqueda del camino óptimo
    - Prueba todas las combinaciones posibles de puntos de entrada y salida
    - Para cada par, ejecuta el algoritmo BFS para encontrar el camino más corto
    - Se detiene al encontrar el primer camino válido
5. Generación de imagen resultado
    - Convierte la imagen en escala de grises a color BGR
    - Marca el camino encontrado en color rojo (BGR: [0, 0, 255])
    - Guarda la imagen resultado en la ruta especificada

FUNCIÓN AUXILIAR: bfs_shortest_path()

Algoritmo BFS (Breadth-First Search - Búsqueda en Anchura)

Parámetros:

- binary: Matriz binaria del laberinto (1=camino libre, 0=pared)
- start: Coordenadas del punto de inicio (y, x)
- end: Coordenadas del punto de destino (y, x)

Funcionamiento:

1. Inicialización
    - Define direcciones de movimiento: arriba, abajo, izquierda, derecha
    - Crea una cola (deque) con el punto inicial y su camino
    - Mantiene un conjunto de nodos visitados para evitar ciclos
2. Proceso de búsqueda
    - Extrae el primer elemento de la cola (FIFO)
    - Verifica si es el punto destino; si es así, retorna el camino
    - Explor|s vecinos válidos (dentro de límites y caminos libres)
    - Añade vecinos no visitados a la cola con su camino actualizado
3. Garantía de camino mínimo
    - BFS garantiza encontrar el camino con menor número de pasos
    - Explora primero todos los nodos a distancia n antes de pasar a distancia n+1

FUNCIÓN DE PRUEBA: test_maze_solver()

Propósito: Función de ejemplo que demuestra el uso del solucionador

Características:

- Define archivos de entrada y salida predeterminados
- Mide y reporta el tiempo de ejecución
- Maneja excepciones y proporciona retroalimentación al usuario

- Indica éxito o falla de la operación

ESPECIFICACIONES TÉCNICAS

FORMATO DE ENTRADA

- Tipo de archivo: Imagen PNG, JPG u otros formatos soportados por OpenCV
- Características del laberinto:
    - Paredes: Píxeles oscuros (valor ≤ 80 en escala de grises)
    - Caminos: Píxeles claros (valor > 80 en escala de grises)
    - Entrada/Salida: Aberturas en los bordes de la imagen

FORMATO DE SALIDA

- Imagen resultado: Misma resolución que la entrada
- Visualización del camino: Línea roja que marca la solución
- Información en consola: Tiempo de ejecución y coordenadas de entrada/salida

COMPLEJIDAD ALGORÍTMICA

- Tiempo: O(V + E) donde V = número de celdas, E = número de conexiones
- Espacio: O(V) para almacenar nodos visitados y cola de búsqueda
- Garantía: Encuentra el camino más corto en número de pasos