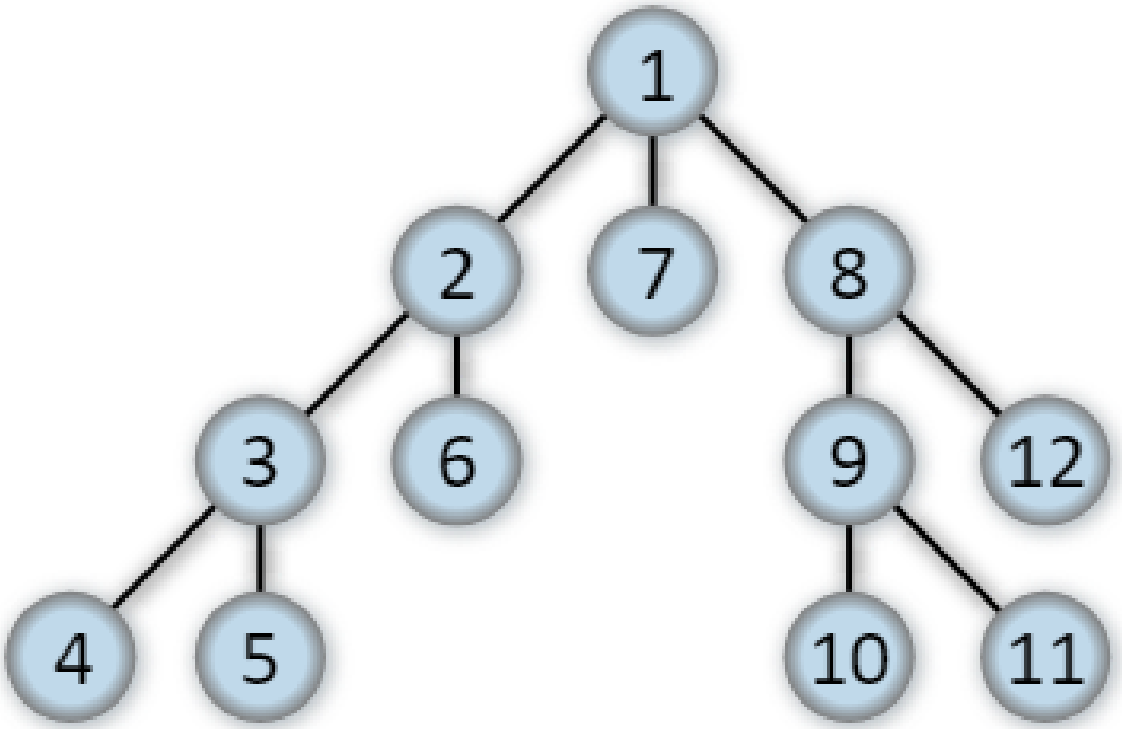
TAREA 2 (24-25) RECORRIDO EN PROFUNDIDAD (RECURSIVIDAD)



Adrián Yared Armas de la Nuez

Contenido

1. Enunciado

1.1 (1 Punto) Índice.
1.2 (3 Puntos) Funcionamiento correcto.
1.3 (2 Puntos) Explicación del funcionamiento y estructuras creadas.
1.4 (2 Puntos) Modifica la matriz de movimientos para permitir saltos en diagonal y añadir más muros (de forma aleatoria) en el interior, olvidar un porcentaje de puntos por dónde se ha pasado.
1.5 (1 Punto) Recorrido en abanico iterativo, indica estructuras creadas.
1.6 (1 Punto) Comparación de los dos algoritmos, dificultades, conclusiones y mejoras (camino más corto).

2. Laberinto en profundidad recursivo

2.1 Estructuras creadas

En este apartado de la actividad, he creado las siguientes estreucturas:

2.1.1.1 Resolver_laberinto

Esta función es la que implementa la lógica para resolver el laberinto. Indica si se puede caminar o no y establece los pasos en el laberinto.

2.1.1.2 Código

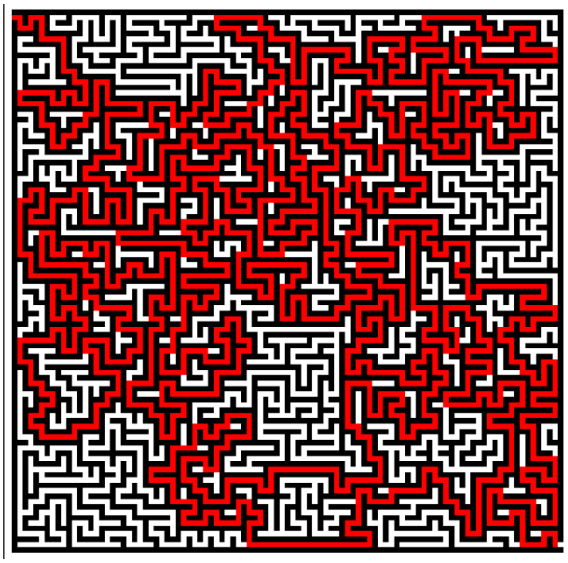
2.1.2.1 mostrar_maze_profundidad(laberinto, camino)

Esta función se encarga de mostrar el laberinto y el camino encontrado.
Crea una copia del laberinto para no modificar el original y dibuja el camino realizado.

2.1.2.2 Código

2.1 Código completo

2.2 Prueba de ejecución



3. Matriz modificada

Modifica la matriz de movimientos para permitir saltos en diagonal y añadir más muros (de forma aleatoria) en el interior, olvidar un porcentaje de puntos por dónde se ha pasado.

3.1 ¿Cómo funciona?

El código proporciona un método para generar un laberinto y encontrar un camino a través de él. Primero, la función create_maze(size) utiliza un algoritmo de retroceso para construir un laberinto de tamaño especificado, llenando un arreglo con muros y creando pasajes. Luego, la función depth_first_search(maze, x, y, end, path, visited) implementa una búsqueda en profundidad para encontrar un camino desde el inicio hasta el final del laberinto, almacenando el camino en una lista. Finalmente, plot_maze(maze, path) visualiza el laberinto y el camino encontrado, utilizando diferentes colores para representar las paredes, los pasajes, el inicio, el final y el camino recorrido. El resultado es una representación gráfica clara del laberinto y la solución.

3.2 Estructuras utilizadas

3.2.1 create_maze(size)

Esta función crea un laberinto utilizando el algoritmo de "backtracking" (retroceso) para generar pasajes.
Bucle de creación:
Mientras haya posiciones en la pila, se toma la última posición añadida (esto imita el comportamiento de una pila).
Se definen las direcciones posibles (movimientos ortogonales) y se mezclan aleatoriamente.
Se busca un nuevo vecino que sea un muro. Si se encuentra uno, se talla un pasaje entre la celda actual y la nueva celda, y se añade la nueva celda a la pila.
Si no se encuentran vecinos, se retrocede eliminando la última celda de la pila.

3.2.2 Código completo

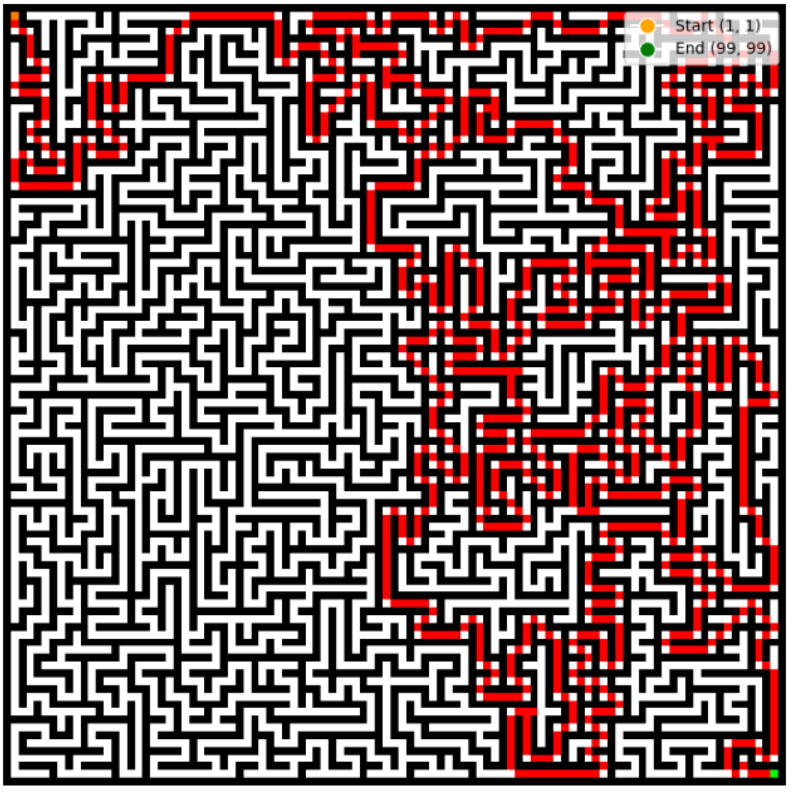
```
import random
import matplotlib.pyplot as plt
import numpy as np

def create_maze(size):
    # Initialize the maze with walls
    maze = [['#' for _ in range(size)] for _ in range(size)]

    # List of positions to visit (stack)
    stack = []

    # Start at a random cell
    start_x, start_y = 1, 1
    maze[start_y][start_x] = ' '  # Mark the starting cell as a passage
    stack.append((start_x, start_y))

    while stack:
        x, y = stack[-1]
        # Movement directions: limiting to orthogonal movements
        directions = [(-2, 0), (2, 0), (0, -2), (0, 2)]  # Only
orthogonal movements
        random.shuffle(directions)  # Shuffle directions
```

```
        # Find a new neighbor
        found = False
        for dx, dy in directions:
            nx, ny = x + dx, y + dy
            if 1 <= nx < size - 1 and 1 <= ny < size - 1 and
maze[ny][nx] == '#':
                maze[y + dy // 2][x + dx // 2] = ' '  # Carve a passage
                maze[ny][nx] = ' '  # Carve the new cell
                stack.append((nx, ny))  # Add the new cell to the stack
                found = True
                break

        if not found:
            stack.pop()  # Backtrack if no more neighbors

    return maze

def plot_maze(maze, path=[]):
    maze_array = np.array(maze)
    maze_numeric = np.where(maze_array == '#', 1, 0)

    color_map = np.zeros((*maze_numeric.shape, 3))

    color_map[maze_numeric == 1] = [0, 0, 0]  # Walls in black
    color_map[maze_numeric == 0] = [1, 1, 1]  # Passages in white
    color_map[1, 1] = [1, 0.5, 0]  # Start (orange)
    color_map[99, 99] = [0, 1, 0]  # End (green)

    # Change the path color from yellow to red
    for (x, y) in path:
        if (x, y) != (1, 1) and (x, y) != (99, 99):
            color_map[y, x] = [1, 0, 0]  # Path in red

    plt.figure(figsize=(9, 9))
    plt.imshow(color_map)
    plt.axis('off')

    # Add legend on the right
```

```
    start_marker = plt.Line2D([0], [0], marker='o', color='w',
markerfacecolor='orange', markersize=10, label='Start (1, 1)')
    end_marker = plt.Line2D([0], [0], marker='o', color='w',
markerfacecolor='green', markersize=10, label='End (99, 99)')
    plt.legend(handles=[start_marker, end_marker], loc='upper right')

    plt.show()

def depth_first_search(maze, x, y, end, path, visited):
    if (x, y) == end:
        path.append((x, y))
        return True

    if maze[y][x] == ' ' and (x, y) not in visited:
        path.append((x, y))
        visited.add((x, y))  # Mark as visited

        # Fixed directions for movement (diagonals first)
        directions = [(1, 1), (1, -1), (-1, 1), (-1, -1), (0, 1), (1,
0), (0, -1), (-1, 0)]

        for dx, dy in directions:
            nx, ny = x + dx, y + dy
            if 0 <= nx < len(maze[0]) and 0 <= ny < len(maze) and
maze[ny][nx] in (' ', end):
                if depth_first_search(maze, nx, ny, end, path,
visited):
                    return True

        path.pop()
        maze[y][x] = ' '  # Unmark to allow other paths

    return False

# Create a maze of 101x101
size = 101
maze = create_maze(size)

# Define the start and end points
```

```
start = (1, 1)
end = (99, 99)

# Initialize the path list and visited set
path = []
visited = set()  # Track visited cells

# Perform depth-first search
depth_first_search(maze, start[0], start[1], end, path, visited)

# Plot the maze with the found path
plot_maze(maze, path)
```

3.3 Prueba de ejecución



4. Recorrido en abanico iterativo.

4.1 Explicación

Este código genera un laberinto y encuentra un camino a través de él utilizando un algoritmo de búsqueda por niveles. Primero, la función create_maze(size) crea un laberinto de tamaño size al inicializar todas las celdas como muros y luego utiliza un enfoque de retroceso para tallar pasajes aleatorios. Luego, la función iterative_fan_search(maze, start, end) implementa un algoritmo de búsqueda que utiliza una cola para explorar las celdas del laberinto, asegurándose de no visitar celdas ya recorridas, y buscando un camino desde el punto de inicio hasta el final. Finalmente, la función plot_maze(maze, path) visualiza el laberinto, mostrando las paredes, los pasajes, el inicio y el final, así como el camino encontrado, utilizando un mapa de colores. El resultado es una representación gráfica del laberinto con el camino resaltado.

4.2 Estructuras utilizadas

4.2.1.1 create_maze(size)

Esta función genera un laberinto de tamaño size. Comienza creando una matriz cuadrada llena de muros ('#'). Se utiliza un enfoque de retroceso con una pila para tallar pasajes, eligiendo direcciones aleatorias y asegurando que cada celda se visite una vez.

4.2.1.2 Código

```
def create_maze(size):
    # Initialize the maze with walls
    maze = [['#' for _ in range(size)] for _ in range(size)]
    stack = []
    start_x, start_y = 1, 1
    maze[start_y][start_x] = ' '  # Set the starting point as a passage
    stack.append((start_x, start_y))

    while stack:
        x, y = stack[-1]
        # Define possible movement directions (up, down, left, right)
        directions = [(-2, 0), (2, 0), (0, -2), (0, 2)]
        random.shuffle(directions)  # Shuffle directions to ensure
randomness

        found = False
        for dx, dy in directions:
```

```
            nx, ny = x + dx, y + dy
            # Check if the next cell is within bounds and is a wall
            if 1 <= nx < size - 1 and 1 <= ny < size - 1 and
maze[ny][nx] == '#':
                maze[y + dy // 2][x + dx // 2] = ' '  # Carve passage
between cells
                maze[ny][nx] = ' '  # Carve the new cell
                stack.append((nx, ny))  # Add new cell to stack
                found = True
                break

        if not found:
            stack.pop()  # Backtrack if no unvisited neighbors are
found

    return maze
```

4.2.2.1 plot_maze(maze, path=[])

Esta función se encarga de visualizar el laberinto. Convierte la matriz del laberinto en un arreglo de NumPy, asignando colores a muros y pasajes, y destacando el punto de inicio y fin. Utiliza matplotlib para mostrar la imagen del laberinto.

4.2.2.2 Código

```
def plot_maze(maze, path=[]):
    # Convert maze to a numpy array for easier manipulation
    maze_array = np.array(maze)
    maze_numeric = np.where(maze_array == '#', 1, 0)  # Convert walls
and passages to numeric values

    # Create a color map for plotting
    color_map = np.zeros((*maze_numeric.shape, 3))
    color_map[maze_numeric == 1] = [0, 0, 0]  # Walls in black
    color_map[maze_numeric == 0] = [1, 1, 1]  # Passages in white
    color_map[1, 1] = [1, 0.5, 0]  # Start point in orange
    color_map[99, 99] = [0, 1, 0]  # End point in green

    # Color the path found in the maze
```

```
    for (x, y) in path:
        if (x, y) != (1, 1) and (x, y) != (99, 99):
            color_map[y, x] = [1, 0, 0]  # Path in red

    plt.figure(figsize=(10, 10))  # Set the figure size
    plt.imshow(color_map)  # Display the maze
    plt.axis('off')  # Turn off the axis

    # Add legend
    plt.scatter(1, 1, color='orange', label='Start (1, 1)', s=100)  #
Start point
    plt.scatter(99, 99, color='green', label='End (99, 99)', s=100)  #
End point
    plt.legend(loc='upper right')  # Move the legend to the upper right

    plt.show()
```

4.2.3.1 iterative_fan_search(maze, start, end)

Implementa un algoritmo de búsqueda que utiliza una cola (deque) para encontrar un camino desde el punto de inicio al punto de fin en el laberinto. Se marcan las celdas visitadas para evitar ciclos, y se exploran las celdas vecinas en cuatro direcciones (arriba, abajo, izquierda, derecha).

4.2.3.2 Código

```
def iterative_fan_search(maze, start, end):
    queue = deque([start])  # Use a queue for the fan search
    path = []
    visited = set()  # To avoid cycles

    while queue:
        x, y = queue.popleft()  # Remove the first element from the
queue

        if (x, y) == end:
            path.append((x, y))  # Append the end point to the path
            return path

        if (x, y) not in visited:
```

```
            visited.add((x, y))  # Mark the cell as visited
            path.append((x, y))  # Add to the path

            # Directions for moving in the fan search (right, down,
left, up)
            directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]
            for dx, dy in directions:
                nx, ny = x + dx, y + dy
                if 0 <= nx < len(maze[0]) and 0 <= ny < len(maze) and
maze[ny][nx] in (' ', end):
                    queue.append((nx, ny))  # Add to the queue

    return []  # Return an empty path if no path exists
```

4.3 Código completo

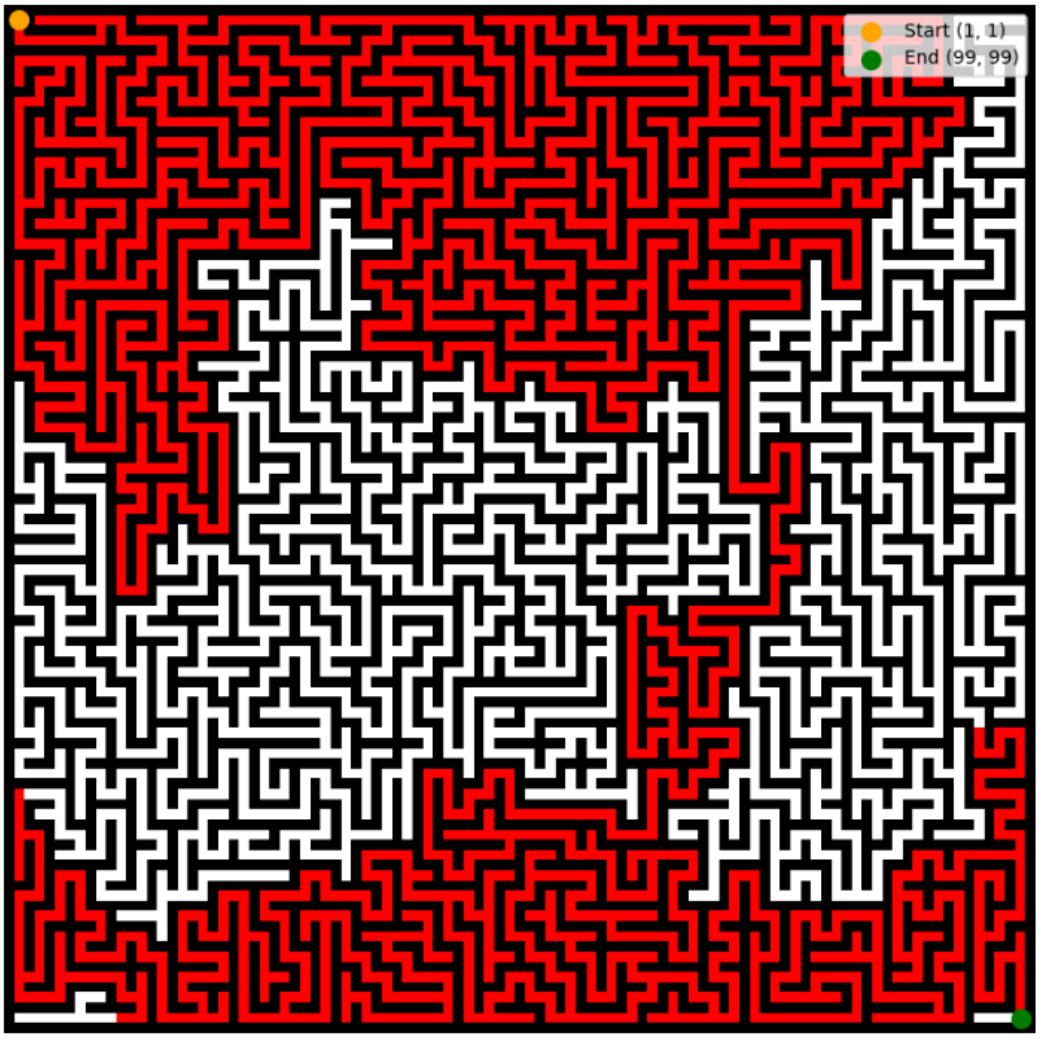
```
import random
import matplotlib.pyplot as plt
import numpy as np
from collections import deque

def create_maze(size):
    # Initialize the maze with walls
    maze = [['#' for _ in range(size)] for _ in range(size)]
    stack = []
    start_x, start_y = 1, 1
    maze[start_y][start_x] = ' '  # Set the starting point as a passage
    stack.append((start_x, start_y))

    while stack:
        x, y = stack[-1]
        # Define possible movement directions (up, down, left, right)
        directions = [(-2, 0), (2, 0), (0, -2), (0, 2)]
        random.shuffle(directions)  # Shuffle directions to ensure
randomness

        found = False
        for dx, dy in directions:
```

```
            nx, ny = x + dx, y + dy
            # Check if the next cell is within bounds and is a wall
            if 1 <= nx < size - 1 and 1 <= ny < size - 1 and
maze[ny][nx] == '#':
                maze[y + dy // 2][x + dx // 2] = ' '  # Carve passage
between cells
                maze[ny][nx] = ' '  # Carve the new cell
                stack.append((nx, ny))  # Add new cell to stack
                found = True
                break

        if not found:
            stack.pop()  # Backtrack if no unvisited neighbors are
found

    return maze

def plot_maze(maze, path=[]):
    # Convert maze to a numpy array for easier manipulation
    maze_array = np.array(maze)
    maze_numeric = np.where(maze_array == '#', 1, 0)  # Convert walls
and passages to numeric values

    # Create a color map for plotting
    color_map = np.zeros((*maze_numeric.shape, 3))
    color_map[maze_numeric == 1] = [0, 0, 0]  # Walls in black
    color_map[maze_numeric == 0] = [1, 1, 1]  # Passages in white
    color_map[1, 1] = [1, 0.5, 0]  # Start point in orange
    color_map[99, 99] = [0, 1, 0]  # End point in green

    # Color the path found in the maze
    for (x, y) in path:
        if (x, y) != (1, 1) and (x, y) != (99, 99):
            color_map[y, x] = [1, 0, 0]  # Path in red

    plt.figure(figsize=(10, 10))  # Set the figure size
    plt.imshow(color_map)  # Display the maze
    plt.axis('off')  # Turn off the axis
```

```
    # Add legend
    plt.scatter(1, 1, color='orange', label='Start (1, 1)', s=100)  #
Start point
    plt.scatter(99, 99, color='green', label='End (99, 99)', s=100)  #
End point
    plt.legend(loc='upper right')  # Move the legend to the upper right

    plt.show()

def iterative_fan_search(maze, start, end):
    queue = deque([start])  # Use a queue for the fan search
    path = []
    visited = set()  # To avoid cycles

    while queue:
        x, y = queue.popleft()  # Remove the first element from the
queue

        if (x, y) == end:
            path.append((x, y))  # Append the end point to the path
            return path

        if (x, y) not in visited:
            visited.add((x, y))  # Mark the cell as visited
            path.append((x, y))  # Add to the path

            # Directions for moving in the fan search (right, down,
left, up)
            directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]
            for dx, dy in directions:
                nx, ny = x + dx, y + dy
                if 0 <= nx < len(maze[0]) and 0 <= ny < len(maze) and
maze[ny][nx] in (' ', end):
                    queue.append((nx, ny))  # Add to the queue

    return []  # Return an empty path if no path exists

# Create a maze of size 101x101
size = 101
```

```
maze = create_maze(size)

# Define the start and end points
start = (1, 1)
end = (99, 99)

path = iterative_fan_search(maze, start, end)# Perform the fan search
plot_maze(maze, path)# Plot the maze with the found path
```

4.4 Prueba de ejecución



5. Comparación

5.1 Búsqueda en Profundidad Recursiva (DFS)

Cómo Funciona: Explora profundamente cada camino antes de retroceder.
Ventajas:

- Fácil de implementar.
- Usa menos memoria en laberintos grandes.

Desventajas:

- No garantiza el camino más corto.
- Puede entrar en ciclos sin control adecuado.

5.2 Búsqueda en Abanico Iterativa

Cómo Funciona: Explora todas las celdas a nivel antes de bajar al siguiente nivel.
Ventajas:

- Siempre encuentra el camino más corto en laberintos no ponderados.
- Evita ciclos al marcar las celdas visitadas.

Desventajas:

- Consume más memoria.
- Implementación más compleja.

5.3 Conclusiones

5.4 Mejoras

En profundidad: Para posibles mejoras podría evitar revisar celdas y usar una pila específica.
Búsqueda en Abanico: Optimizar el uso de memoria y considerar heurísticas (Busqueda mediante un método más caótico) para mejorar la búsqueda.

6. Colab

Enlace a colab