Juego del 8 DFS

Trabajo Corte 1 - Computación 2

Nombre: Mariana Rodríguez Pérez

Fecha: Marzo 2025

El juego del ocho es un rompecabezas deslizante que consiste en una cuadrícula de 3x3 donde se encuentran 8 fichas numeradas y una casilla vacía. El objetivo es mover las fichas, deslizando aquellas que se encuentran en la misma fila o columna que la casilla vacía, hasta ordenar los números de forma secuencial (de izquierda a derecha y de arriba hacia abajo), dejando la casilla vacía en la última posición.

El código se realizó en Python utilizando conceptos básicos de programación orientada a objetos. Se creó una clase llamada **Game** que:

- Inicializa el tablero con una matriz 3x3.
- Define métodos para visualizar el tablero usando **matplotlib**.
- Implementa funciones para encontrar la casilla vacía y validar si los movimientos (arriba, abajo, izquierda o derecha) son permitidos.
- Permite realizar los movimientos actualizando el estado del tablero y registra el historial de movimientos.

Esta estructura modular facilita la extensión y mejora del código, permitiendo una gestión clara de la lógica del juego y su visualización gráfica.

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

class Position:
    def __init__(self, x, y):
        self.x = x
        self.y = y
    def __str__(self):
        return f"[x:{self.x}, y:{self.y}]"
    def __repr__(self):
        return f"[x:{self.x}, y:{self.y}]"

class Node:
    def __init__(self, stage, move):
        self.stage = stage
        self.move = move
        self.deep = None
```

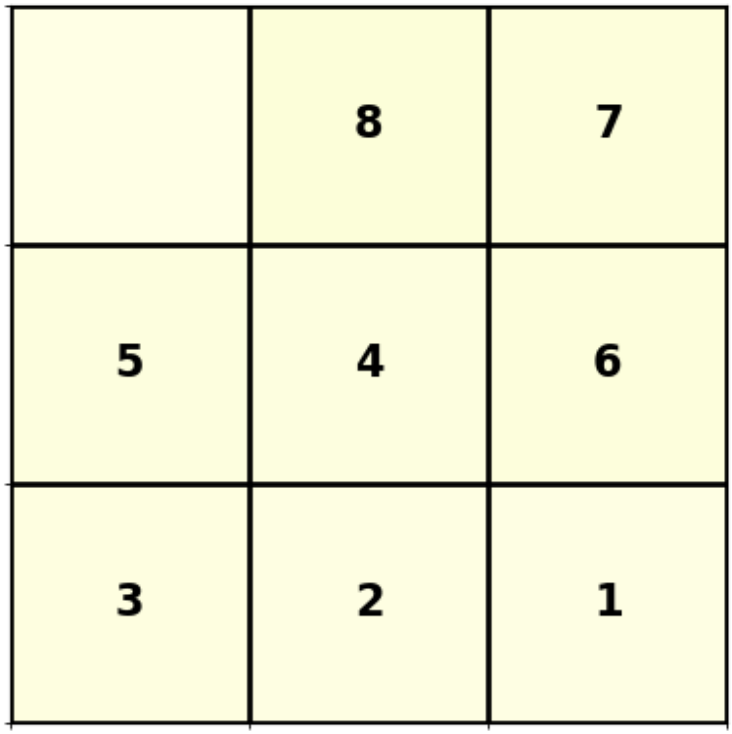
```
    def __str__(self):
        return f"[move: {self.move}, stage: {self.stage}, deep:
{self.deep}]"

    def __repr__(self):
        return f"[move: {self.move}, stage: {self.stage}, deep:
{self.deep}]"

class Game:

    GOAL = [[1, 2, 3],
            [4, 5, 6],
            [7, 8, None]]

    def __init__(self, initial_board):
        self.board = [row[:] for row in initial_board]
        self.position_empty_space = self.__find_empty_space_position()

    def __find_empty_space_position(self):
        for y in range(len(self.board)):
            for x in range(len(self.board[y])):
                if self.board[y][x] is None:
                    return Position(x, y)
        raise Exception("No se encontró un espacio vacío en el juego")

    def __copy(self):
        copy_board = [row[:] for row in self.board]
        return Game(copy_board)

    def show(self):
        _, ax = plt.subplots()
        img_data = np.array([[0 if x is None else x for x in row] for
row in self.board])
        plt.imshow(img_data, cmap="YlGn", interpolation="nearest",
vmin=0, vmax=255)
        ax.set_xticks(np.arange(-0.5, len(self.board[0]), 1),
minor=True)
        ax.set_yticks(np.arange(-0.5, len(self.board), 1), minor=True)
        ax.grid(which="minor", color="black", linestyle="-",
linewidth=2)
        ax.set_xticks([])
        ax.set_yticks([])
        for i in range(len(self.board)):
            for j in range(len(self.board[0])):
                value = self.board[i][j]
                text = str(value) if value is not None else " "
                ax.text(j, i, text, ha='center', va='center',
fontsize=16, fontweight='bold')
        plt.show()
```

```
    def is_game_win(self):
        return self.board == self.GOAL

    def is_allowed_move_up(self):
        return self.position_empty_space.y > 0

    def is_allowed_move_down(self):
        return self.position_empty_space.y < len(self.board) - 1

    def is_allowed_move_left(self):
        return self.position_empty_space.x > 0

    def is_allowed_move_right(self):
        return self.position_empty_space.x < len(self.board[0]) - 1

    def move_up(self):
        self.board[self.position_empty_space.y]
[self.position_empty_space.x] = \
            self.board[self.position_empty_space.y - 1]
[self.position_empty_space.x]
        self.board[self.position_empty_space.y - 1]
[self.position_empty_space.x] = None
        self.position_empty_space.y -= 1

    def move_down(self):
        self.board[self.position_empty_space.y]
[self.position_empty_space.x] = \
            self.board[self.position_empty_space.y + 1]
[self.position_empty_space.x]
        self.board[self.position_empty_space.y + 1]
[self.position_empty_space.x] = None
        self.position_empty_space.y += 1

    def move_left(self):
        self.board[self.position_empty_space.y]
[self.position_empty_space.x] = \
            self.board[self.position_empty_space.y]
[self.position_empty_space.x - 1]
        self.board[self.position_empty_space.y]
[self.position_empty_space.x - 1] = None
        self.position_empty_space.x -= 1

    def move_right(self):
        self.board[self.position_empty_space.y]
[self.position_empty_space.x] = \
            self.board[self.position_empty_space.y]
[self.position_empty_space.x + 1]
        self.board[self.position_empty_space.y]
[self.position_empty_space.x + 1] = None
        self.position_empty_space.x += 1
```

```
    def next_allowed_moves(self):
        next_nodes = []
        if self.is_allowed_move_up():
            copy_game = self.__copy()
            copy_game.move_up()
            next_nodes.append(Node(copy_game.board, "UP"))
        if self.is_allowed_move_down():
            copy_game = self.__copy()
            copy_game.move_down()
            next_nodes.append(Node(copy_game.board, "DOWN"))
        if self.is_allowed_move_left():
            copy_game = self.__copy()
            copy_game.move_left()
            next_nodes.append(Node(copy_game.board, "LEFT"))
        if self.is_allowed_move_right():
            copy_game = self.__copy()
            copy_game.move_right()
            next_nodes.append(Node(copy_game.board, "RIGHT"))
        return next_nodes

def dfs_solve(game, depth, visited):
    if game.is_game_win():
        return []
    if depth == 0:
        return None
    board_tuple = tuple(tuple(row) for row in game.board)

    if board_tuple in visited:
        return None
    visited.add(board_tuple)

    for node in game.next_allowed_moves():
        new_game = Game([row[:] for row in node.stage])
        result = dfs_solve(new_game, depth - 1, visited)
        if result is not None:
            return [node.move] + result
    return None

def dfs(game, max_depth=50):
    for d in range(1, max_depth + 1):
        visited = set()
        result = dfs_solve(game, d, visited)
        if result is not None:
            return result
    return None

initial_board = [
    [None, 8, 7],
    [5, 4, 6],
```

```
    [3, 2, 1]
]

game = Game(initial_board)
print("Tablero inicial:")
game.show()

first_time = datetime.datetime.now()

solution_moves = dfs(game, max_depth=50)
later_time = datetime.datetime.now()

if solution_moves is not None:
    print("\nSecuencia de movimientos para alcanzar el GOAL:")
    for move in solution_moves:
        print(move)
    print("La cantidad de movimientos sugeridos usando DFS es:",
len(solution_moves))
else:
    print("No se encontró solución.")

print(f"El tiempo de ejecución usando DFS: {later_time - first_time}")
```

```
Tablero inicial:
```



```
Secuencia de movimientos para alcanzar el GOAL:
DOWN
DOWN
RIGHT
UP
UP
LEFT
DOWN
DOWN
RIGHT
UP
UP
RIGHT
DOWN
DOWN
LEFT
UP
UP
LEFT
DOWN
RIGHT
RIGHT
DOWN
LEFT
LEFT
UP
RIGHT
DOWN
RIGHT
UP
UP
LEFT
LEFT
DOWN
RIGHT
DOWN
RIGHT
La cantidad de movimientos sugeridos usando DFS es: 36
El tiempo de ejecución usando DFS: 0:00:36.557009
```