Solução para o problema EightPuzzle

Função principal onde é definido o estado inical

```
import random
import copy

class PuzzleState:  # Define a classe PuzzleState para representar o estado do quebra-cabeça
    def __init__(self, puzzle):
        self.puzzle = puzzle
        self.side_length = int(len(puzzle) ** 0.5)  # Calcula o lado do tabuleiro baseado no comprimento da lista
        self.goal_state = list(range(1, len(puzzle))) + [0]  # Estado final do quebra-cabeça

    def __str__(self):
        puzzle_str = ""
        for i in range(self.side_length):
            # Gera uma representação visual do quebra-cabeça, substituindo 0 por espaço em branco
            puzzle_str += " ".join(str(x) if x != 0 else " " for x in self.puzzle[i * self.side_length:(i + 1) * self.side_length]) + "\n
        return puzzle_str  # Retorna a representação visual do quebra-cabeça como uma string

    def move_blank(self, direction):  # Método para mover o espaço em branco na direção especificada
        blank_index = self.puzzle.index(0)  # Obtém o índice do espaço em branco
        blank_row = blank_index // self.side_length  # Obtém a linha do espaço em branco
        blank_col = blank_index % self.side_length  # Obtém a coluna do espaço em branco

        # Move o espaço em branco na direção especificada, se possível
        if direction == "up" and blank_row > 0:
            new_row = blank_row - 1
            new_index = new_row * self.side_length + blank_col
            self.puzzle[blank_index], self.puzzle[new_index] = self.puzzle[new_index], self.puzzle[blank_index]
        elif direction == "down" and blank_row < self.side_length - 1:
            new_row = blank_row + 1
            new_index = new_row * self.side_length + blank_col
            self.puzzle[blank_index], self.puzzle[new_index] = self.puzzle[new_index], self.puzzle[blank_index]
        elif direction == "left" and blank_col > 0:
            new_col = blank_col - 1
            new_index = blank_row * self.side_length + new_col
            self.puzzle[blank_index], self.puzzle[new_index] = self.puzzle[new_index], self.puzzle[blank_index]
        elif direction == "right" and blank_col < self.side_length - 1:
            new_col = blank_col + 1
            new_index = blank_row * self.side_length + new_col
            self.puzzle[blank_index], self.puzzle[new_index] = self.puzzle[new_index], self.puzzle[blank_index]

    # Método para verificar se o estado atual é o estado final do quebra-cabeça
    def is_goal_state(self):
        return self.puzzle == self.goal_state

    # Método para obter os movimentos legais disponíveis para o espaço em branco
    def legal_moves(self):
        blank_index = self.puzzle.index(0)  # Obtém o índice do espaço em branco
        blank_row = blank_index // self.side_length  # Obtém a linha do espaço em branco
        blank_col = blank_index % self.side_length  # Obtém a coluna do espaço em branco

        moves = []  # Inicializa uma lista para armazenar os movimentos legais disponíveis
        # Verifica quais movimentos são legais para o espaço em branco e os adiciona à lista de movimentos
        if blank_row > 0:
            moves.append("up")
        if blank_row < self.side_length - 1:
            moves.append("down")
        if blank_col > 0:
            moves.append("left")
        if blank_col < self.side_length - 1:
            moves.append("right")

        return moves  # Retorna a lista de movimentos legais disponíveis

def solve_puzzle(initial_state):  # Função para resolver o quebra-cabeça usando busca em largura
    frontier = [(initial_state, [])]  # Inicializa a fronteira com o estado inicial e uma lista vazia de movimentos
    explored = set()  # Inicializa o conjunto de estados explorados como vazio

    while frontier:  # Loop principal da busca em largura
        state, path = frontier.pop(0)  # Remove o primeiro estado da fronteira
        explored.add(tuple(state.puzzle))  # Adiciona o estado atual ao conjunto de estados explorados

        if state.is_goal_state():  # Verifica se o estado atual é o estado final do quebra-cabeça
            return path

        for move in state.legal_moves():
            new_state = copy.deepcopy(state)
            new_state.move_blank(move)
```

```
                new_state.move_blank(move)
                if tuple(new_state.puzzle) not in explored:  # Verifica se o novo estado já foi explorado
                    frontier.append((new_state, path + [move]))

    return None  # Retorna None se não for encontrada uma solução

if __name__ == "__main__":  # Verifica se o script está sendo executado como programa principal
    puzzle = list(range(0, 9))  # Cria uma lista com os números de 1 a 8
    random.shuffle(puzzle)      # Embaralha os números
    estado_inicial = PuzzleState(puzzle)  # Cria o estado inicial do quebra-cabeça
    print("Estado Inicial:")
    print(estado_inicial)
    solucao = solve_puzzle(estado_inicial)  # Resolve o quebra-cabeça

    if solucao:  # Verifica se foi encontrada uma solução
        print("Solução encontrada em", len(solucao), "passos:")  # Imprime o número de passos necessários
        for passo, movimento in enumerate(solucao, 1):  # Loop sobre os movimentos na solução
            print("Passo", passo, ":", movimento)  # Imprime o número do passo e o movimento realizado
            estado_inicial.move_blank(movimento)  # Realiza o movimento no estado inicial
            print(estado_inicial)  # Imprime o estado atual do quebra-cabeça após o movimento
    else:
        print("Nenhuma solução encontrada.")  # Imprime uma mensagem caso não seja encontrada uma solução
```

```
2 1 5
4 7 6

Passo 13 : down
8 1 3
2   5
4 7 6

Passo 14 : left
8 1 3
  2 5
4 7 6

Passo 15 : up
  1 3
8 2 5
4 7 6

Passo 16 : right
1   3
8 2 5
4 7 6

Passo 17 : down
1 2 3
8   5
4 7 6

Passo 18 : left
1 2 3
  8 5
4 7 6

Passo 19 : down
1 2 3
4 8 5
  7 6

Passo 20 : right
1 2 3
4 8 5
7   6

Passo 21 : up
1 2 3
4   5
7 8 6

Passo 22 : right
1 2 3
4 5
7 8 6

Passo 23 : down
1 2 3
4 5 6
7 8
```

Clique duas vezes (ou pressione "Enter") para editar