Implementation of 8 Puzzle Problem using non heuristic approach

Using BFS

```
from collections import deque


# Class representing the state of the board

class BoardNode:

def __init__(self, board, parent=None, move=None):

self.board = board  # The current state of the board

self.parent = parent  # The parent node (previous state)

self.move = move  # The move taken to reach this state


def is_goal(self, goal_state):

return self.board == goal_state  # Check if the current state matches the goal state


def __hash__(self):

return hash(str(self.board))  # Unique identifier for the state


def __eq__(self, other):

return self.board == other.board  # Equality check for two BoardNodes


def __repr__(self):

return '\n'.join([' '.join(map(str, row)) for row in self.board])  # Display the board state



# BFS class implementing the search algorithm

class BFS:

def __init__(self, initial_board, goal_state):

self.initial_node = BoardNode(initial_board)  # Initialize with the starting state
```

```
self.goal_state = goal_state  # Goal state for the puzzle

def search(self):
visited = set()  # Set to keep track of visited states
queue = deque([self.initial_node])  # Initialize the queue for BFS
solutions = []  # List to store all solutions

while queue:
node = queue.popleft()  # Remove the first node from the queue

if node.is_goal(self.goal_state):  # Check if the current node is a goal state
path = self.get_solution_path(node)  # Get the solution path
solutions.append(path)  # Store the solution path
continue  # Continue to search for other solutions

visited.add(node)  # Mark the current node as visited

# Generate successors and add them to the queue
for neighbor in self.get_successors(node):
if neighbor not in visited and neighbor not in queue:
queue.append(neighbor)  # Add the new state to the queue

# Display all solutions found
for idx, solution in enumerate(solutions):
print(f"Solution {idx + 1}: Moves: {solution}")

print(f"Total number of optimal solutions: {len(solutions)}")  # Print total solutions
```

```
def get_successors(self, node):

board = node.board

x, y = self.find_zero(board)  # Find the position of the empty tile (0)

successors = []


# Define possible moves (direction, new x, new y)

moves = [('UP', -1, 0), ('DOWN', 1, 0), ('LEFT', 0, -1), ('RIGHT', 0, 1)]


# Generate new board states for each possible move

for move_name, dx, dy in moves:

new_x, new_y = x + dx, y + dy

if 0 <= new_x < 3 and 0 <= new_y < 3:  # Check if the new position is within bounds

new_board = [row[:] for row in board]  # Copy the current board state

new_board[x][y], new_board[new_x][new_y] = new_board[new_x][new_y],
new_board[x][y]  # Swap

successors.append(BoardNode(new_board, node, move_name))  # Create a new BoardNode


return successors


def find_zero(self, board):

# Locate the empty space (0) in the board

for i in range(len(board)):

for j in range(len(board[i])):

if board[i][j] == 0:

return i, j

return None
```

```
def get_solution_path(self, node):

# Reconstruct the path to the solution by following parent nodes

path = []

while node.parent is not None:

path.append(node.move)  # Add the move taken to the path

node = node.parent  # Move to the parent node

return path[::-1]  # Reverse the path to get the correct order



# Function to get user input for the board configuration

def get_user_input(board_type):

print(f"Enter the {board_type} 8-puzzle board configuration (3x3) row by row (use 0 for the
empty tile):")

board = []

for _ in range(3):

row = list(map(int, input().split()))  # Input a row

board.append(row)  # Add row to the board

return board


# Main function to run the program

if __name__ == "__main__":

initial_board = get_user_input("initial")  # Get the initial board configuration from the user

goal_state = get_user_input("goal")  # Get the goal board configuration from the user


# Create a BFS instance and start the search

bfs_solver = BFS(initial_board, goal_state)

bfs_solver.search()  # Perform the search for solutions
```

Output:

Enter the initial 8-puzzle board configuration (3x3) row by row (use 0 for the empty tile):

1 2 3

0 4 6

7 5 8

Enter the goal 8-puzzle board configuration (3x3) row by row (use 0 for the empty tile):

1 2 3

4 5 6

7 8 0

Solution 1:

Moves: ['RIGHT', 'DOWN', 'RIGHT']

Final Board State:

1 2 3

4 5 6

7 8 0

Number of moves: 3

Solution 2:

Moves: ['RIGHT', 'RIGHT', 'DOWN', 'LEFT', 'UP', 'RIGHT', 'DOWN', 'LEFT', 'UP', 'RIGHT', 'DOWN']

Final Board State:

1 2 3

4 5 6

7 8 0

Number of moves: 11

Using DFS

```
class BoardNode:
def __init__(self, board, parent=None, move=None):
self.board = board
self.parent = parent
self.move = move

def is_goal(self, goal_state):
return self.board == goal_state

def __hash__(self):
return hash(str(self.board))

def __eq__(self, other):
return self.board == other.board

def __repr__(self):
return '\n'.join([' '.join(map(str, row)) for row in self.board])

class DFS:
def __init__(self, initial_board, goal_state):
self.initial_node = BoardNode(initial_board)
self.goal_state = goal_state
self.found_solutions = []
self.unique_states = set()  # Set to track unique states
self.permutation_count = 0   # Counter for permutations
```

```
def search(self):
visited = set()
self._dfs(self.initial_node, visited)

# Print solutions and states
if self.found_solutions:
for solution in self.found_solutions:
self.print_solution(solution)
else:
print("No solution found.")

# Print unique states encountered
print(f"Total unique states encountered: {len(self.unique_states)}")
print(f"Total permutations encountered: {self.permutation_count}")

def _dfs(self, node, visited):
if node.is_goal(self.goal_state):
self.found_solutions.append(self.get_solution_path(node))
return True

visited.add(node)  # Add the current node to visited set
# Add current state as a hashable tuple
self.unique_states.add(tuple(map(tuple, node.board)))  # Add current state to unique states
self.permutation_count += 1  # Increment permutation count

for neighbor in self.get_successors(node):
if neighbor not in visited:
if self._dfs(neighbor, visited):
```

```
return True

return False

def get_successors(self, node):
board = node.board
x, y = self.find_zero(board)
successors = []
moves = [('UP', -1, 0), ('DOWN', 1, 0), ('LEFT', 0, -1), ('RIGHT', 0, 1)]

for move_name, dx, dy in moves:
new_x, new_y = x + dx, y + dy
if 0 <= new_x < 3 and 0 <= new_y < 3:  # Check bounds
new_board = [row[:] for row in board]  # Make a copy
new_board[x][y], new_board[new_x][new_y] = new_board[new_x][new_y], new_board[x][y]
successors.append(BoardNode(new_board, node, move_name))

return successors

def find_zero(self, board):
for i in range(3):
for j in range(3):
if board[i][j] == 0:
return i, j

def get_solution_path(self, node):
path = []
```

```
while node:
if node.move:
path.append(node.move)
node = node.parent
return path[::-1]  # Reverse path to show from start to goal


def print_solution(self, path):
print("Solution moves:", path)


def get_user_input():
print("Enter the initial state (3 rows of 3 numbers, use 0 for empty space):")
board = []
for _ in range(3):
row = list(map(int, input().strip().split()))
board.append(row)
return board
def main():
initial_board = get_user_input()
print("Enter the goal state (3 rows of 3 numbers, use 0 for empty space):")
goal_state = []
for _ in range(3):
row = list(map(int, input().strip().split()))
goal_state.append(row)
dfs_solver = DFS(initial_board, goal_state)
dfs_solver.search(
if __name__ == "__main__":
main()
```

Output:

Enter the initial state (3 rows of 3 numbers, use 0 for empty space):

1 2 3

0 4 6

7 5 8

Enter the goal state (3 rows of 3 numbers, use 0 for empty space):

1 2 3

4 5 6

7 8 0

Total unique states encountered: 852

Total permutations encountered: 852