Lab 3:-

8-Puzzel Game

Code:-

Using Depth First Search (DFS)

class SlidingPuzzle:

def \_\_init\_\_(self, board, empty\_pos, path=[]):

self.board = board

self.empty\_pos = empty\_pos

self.path = path

def is\_solved(self):

return self.board == [1, 2, 3, 4, 5, 6, 7, 8, 0]

def get\_moves(self):

x, y = self.empty\_pos

possible\_moves = []

for dx, dy in [(-1, 0), (1, 0), (0, -1), (0, 1)]:

nx, ny = x + dx, y + dy

if 0 <= nx < 3 and 0 <= ny < 3:

new\_board = self.board[:]

new\_board[x \* 3 + y], new\_board[nx \* 3 + ny] = new\_board[nx \* 3 + ny], new\_board[x \* 3 + y]

possible\_moves.append((new\_board, (nx, ny)))

return possible\_moves

def depth\_first\_search(initial\_puzzle):

stack, visited = [initial\_puzzle], set()

while stack:

current\_puzzle = stack.pop()

if current\_puzzle.is\_solved():

return current\_puzzle.path

visited.add(tuple(current\_puzzle.board))

for new\_board, new\_empty\_pos in current\_puzzle.get\_moves():

new\_state = SlidingPuzzle(new\_board, new\_empty\_pos, current\_puzzle.path + [new\_board])

if tuple(new\_board) not in visited:

stack.append(new\_state)

return None

def display\_board(board):

for i in range(0, 9, 3):

print(board[i:i + 3])

print()

def main():

initial\_board = [1, 2, 3, 4, 0, 5, 7, 8, 6]

empty\_pos = initial\_board.index(0)

initial\_puzzle = SlidingPuzzle(initial\_board, (empty\_pos // 3, empty\_pos % 3))

print("Initial state:")

display\_board(initial\_board)

solution = depth\_first\_search(initial\_puzzle)

if solution:

print("Solution found:")

for step in solution:

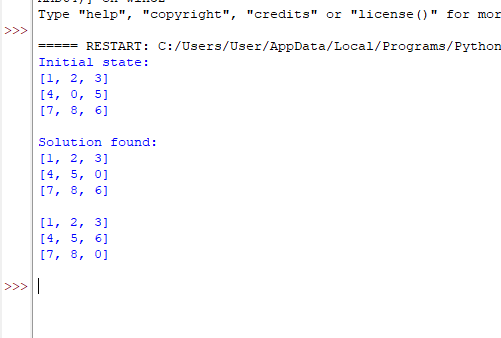
display\_board(step)

else:

print("No solution found.")

if \_\_name\_\_ == "\_\_main\_\_":

main()  
Output:-



Code:-

Using Manhattan Distance

class SlidingPuzzleSolver:

def \_\_init\_\_(self, initial\_state):

self.initial\_state = initial\_state

self.goal\_state = [[1, 2, 3], [4, 5, 6], [7, 8, 0]]

def manhattan\_distance(self, state):

distance = 0

for i in range(3):

for j in range(3):

if state[i][j] != 0:

goal\_i = (state[i][j] - 1) // 3

goal\_j = (state[i][j] - 1) % 3

distance += abs(i - goal\_i) + abs(j - goal\_j)

return distance

def get\_neighbors(self, state):

i, j = next((i, j) for i in range(3) for j in range(3) if state[i][j] == 0)

moves = [(i - 1, j), (i + 1, j), (i, j - 1), (i, j + 1)]

return [self.swap(state, i, j, x, y) for x, y in moves if 0 <= x < 3 and 0 <= y < 3]

def swap(self, state, i1, j1, i2, j2):

new\_state = [row[:] for row in state]

new\_state[i1][j1], new\_state[i2][j2] = new\_state[i2][j2], new\_state[i1][j1]

return new\_state

def dfs\_with\_manhattan(self, state, visited=set()):

if state == self.goal\_state:

return [state]

visited.add(str(state))

neighbors = sorted(self.get\_neighbors(state), key=lambda x: self.manhattan\_distance(x))

for neighbor in neighbors:

if str(neighbor) not in visited:

path = self.dfs\_with\_manhattan(neighbor, visited)

if path:

return [state] + path

return None

def solve(self):

solution = self.dfs\_with\_manhattan(self.initial\_state)

return solution

initial\_state = [[int(x) for x in input(f"Enter row {i + 1}: ").split()] for i in range(3)]

solver = SlidingPuzzleSolver(initial\_state)

solution = solver.solve()

if solution:

print("Solution found:")

for state in solution:

print(\*state, sep='\n', end='\n\n')

else:

print("No solution found.")

Output:-

