Lab 4

1. Write a Program to Implement 8-Puzzle problem using Python.

import heapq

# Goal state to compare

GOAL\_STATE = [[1, 2, 3],

              [4, 5, 6],

              [7, 8, 0]]

# Movements: (dx, dy)

MOVES = {

    'UP': (-1, 0),

    'DOWN': (1, 0),

    'LEFT': (0, -1),

    'RIGHT': (0, 1)

}

class PuzzleState:

    def \_\_init\_\_(self, board, path=[], cost=0):

        self.board = board

        self.path = path

        self.cost = cost

        self.heuristic = self.calculate\_heuristic()

        self.total\_cost = self.cost + self.heuristic

    def \_\_lt\_\_(self, other):

        return self.total\_cost < other.total\_cost

    def find\_zero(self):

        for i in range(3):

            for j in range(3):

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

                    return i, j

    def calculate\_heuristic(self):

        # Manhattan Distance

        distance = 0

        for i in range(3):

            for j in range(3):

                value = self.board[i][j]

                if value != 0:

                    goal\_x = (value - 1) // 3

                    goal\_y = (value - 1) % 3

                    distance += abs(i - goal\_x) + abs(j - goal\_y)

        return distance

    def generate\_successors(self):

        successors = []

        x, y = self.find\_zero()

        for move, (dx, dy) in MOVES.items():

            new\_x, new\_y = x + dx, y + dy

            if 0 <= new\_x < 3 and 0 <= new\_y < 3:

                new\_board = [row[:] for row in self.board]

                new\_board[x][y], new\_board[new\_x][new\_y] = new\_board[new\_x][new\_y], new\_board[x][y]

                successors.append(PuzzleState(new\_board, self.path + [move], self.cost + 1))

        return successors

    def is\_goal(self):

        return self.board == GOAL\_STATE

    def board\_tuple(self):

        return tuple(tuple(row) for row in self.board)

def solve\_puzzle(start\_board):

    start\_state = PuzzleState(start\_board)

    frontier = []

    heapq.heappush(frontier, start\_state)

    visited = set()

    while frontier:

        current\_state = heapq.heappop(frontier)

        if current\_state.is\_goal():

            return current\_state.path

        visited.add(current\_state.board\_tuple())

        for successor in current\_state.generate\_successors():

            if successor.board\_tuple() not in visited:

                heapq.heappush(frontier, successor)

    return None

# Example use

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

    start\_board = [[0, 2, 3],

                   [4, 8, 1],

                   [6, 5, 7]]

    solution = solve\_puzzle(start\_board)

    if solution:

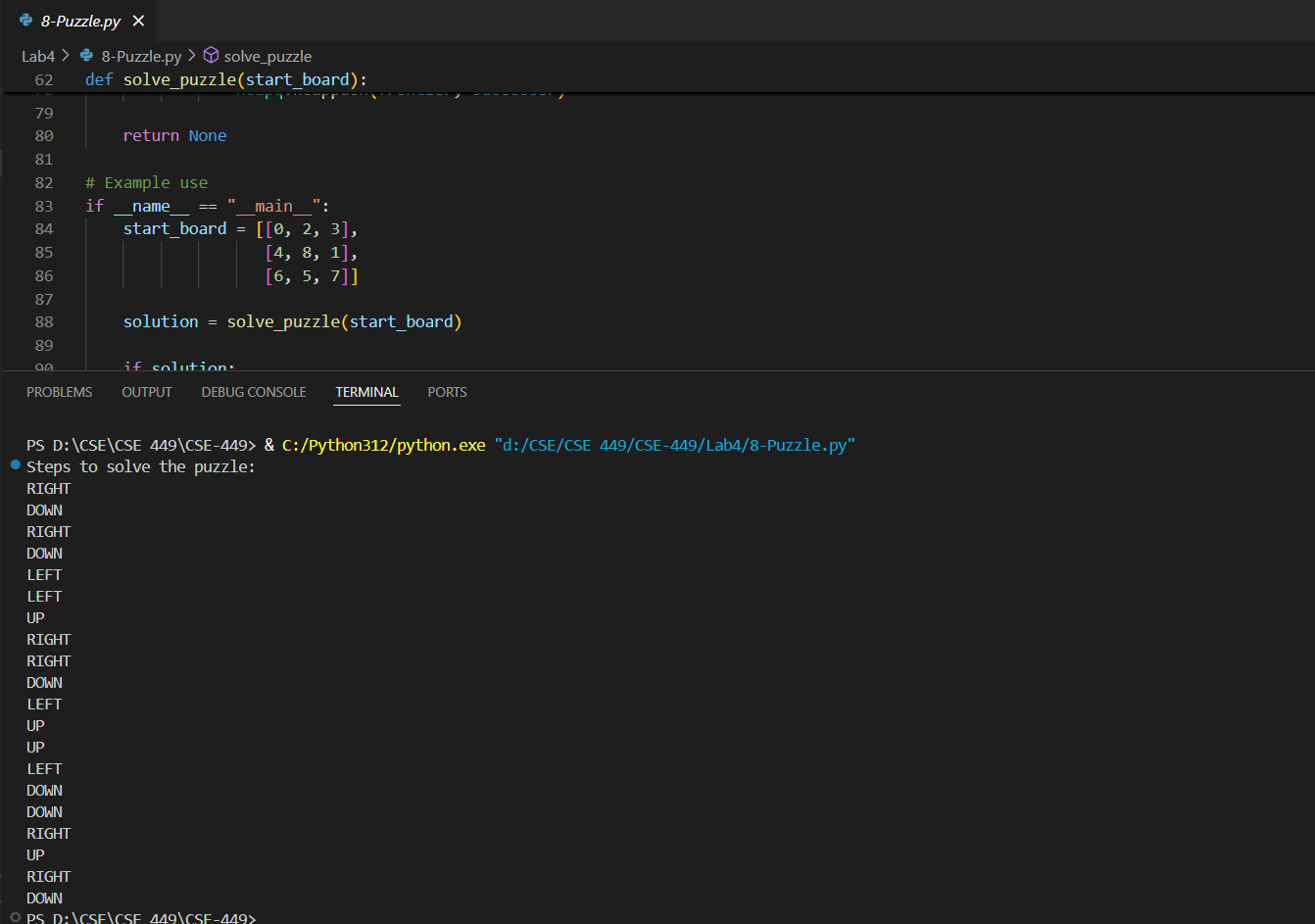
        print("Steps to solve the puzzle:")

        for step in solution:

            print(step)

    else:

        print("No solution found.")



1. Write a Program to Implement Monkey Banana Problem using Python.

# Global Variable i

i = 0

def Monkey\_go\_box(x, y):

    global i

    i = i + 1

    print('step:', i, 'monkey slave', x, 'Go to ' + y)

def Monkey\_move\_box(x, y):

    global i

    i = i + 1

    print('step:', i, 'monkey take the box from', x, 'deliver to ' + y)

def Monkey\_on\_box():

    global i

    i = i + 1

    print('step:', i, 'Monkey climbs up the box')

def Monkey\_get\_banana():

    global i

    i = i + 1

    print('step:', i, 'Monkey picked a banana')

# Read the input operating parameters

codeIn = input(">> ")

codeInList = codeIn.split()

# The operating parameters indicate the locations of monkey, banana, and box respectively

monkey = codeInList[0]

banana = codeInList[1]

box = codeInList[2]

print('The steps are as follows:')

# Please use the least steps to complete the monkey picking banana task

Monkey\_go\_box(monkey, box)

Monkey\_move\_box(box, banana)

Monkey\_on\_box()

Monkey\_get\_banana()

