import heapq

class PuzzleState:

def \_\_init\_\_(self, board, zero\_pos, moves=0, parent=None, goal\_board=None):

self.board = board

self.zero\_pos = zero\_pos # Position of the empty space (0)

self.moves = moves

self.parent = parent

self.goal\_board = goal\_board

self.cost = self.moves + self.heuristic()

def heuristic(self):

# Manhattan distance heuristic

distance = 0

for i in range(3):

for j in range(3):

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

target\_x = (self.board[i][j] - 1) // 3

target\_y = (self.board[i][j] - 1) % 3

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

return distance

def generate\_successors(self):

successors = []

x, y = self.zero\_pos

directions = [(1, 0), (-1, 0), (0, 1), (0, -1)] # down, up, right, left

for dx, dy in directions:

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

if 0 <= new\_x < 3 and 0 <= new\_y < 3: # within bounds

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

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, (new\_x, new\_y), self.moves + 1, self, self.goal\_board))

return successors

def \_\_lt\_\_(self, other):

return self.cost < other.cost

def a\_star(start\_board, goal\_board):

zero\_pos = next((i, j) for i in range(3) for j in range(3) if start\_board[i][j] == 0)

start\_state = PuzzleState(start\_board, zero\_pos, goal\_board=goal\_board)

open\_set = []

heapq.heappush(open\_set, start\_state)

closed\_set = set()

while open\_set:

current\_state = heapq.heappop(open\_set)

# Check if we reached the goal state

if current\_state.board == current\_state.goal\_board:

return current\_state

closed\_set.add(tuple(map(tuple, current\_state.board)))

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

if tuple(map(tuple, successor.board)) in closed\_set:

continue

heapq.heappush(open\_set, successor)

return None # No solution found

def print\_solution(solution):

path = []

while solution:

path.append(solution.board)

solution = solution.parent

for state in reversed(path):

for row in state:

print(row)

print()

def get\_board\_input():

board = []

print("Enter the board configuration (use 0 for the empty space):")

for i in range(3):

while True:

row = input(f"Row {i + 1} (space-separated numbers): ").strip().split()

if len(row) == 3 and all(num.isdigit() and 0 <= int(num) <= 8 for num in row):

board.append([int(num) for num in row])

break

else:

print("Invalid input. Please enter three numbers (0-8) for each row.")

return board

def get\_goal\_input():

goal = []

print("Enter the goal configuration (use 0 for the empty space):")

for i in range(3):

while True:

row = input(f"Row {i + 1} (space-separated numbers): ").strip().split()

if len(row) == 3 and all(num.isdigit() and 0 <= int(num) <= 8 for num in row):

goal.append([int(num) for num in row])

break

else:

print("Invalid input. Please enter three numbers (0-8) for each row.")

return goal

# Example usage

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

initial\_board = get\_board\_input()

goal\_board = get\_goal\_input()

solution = a\_star(initial\_board, goal\_board)

if solution:

print("Solution found:")

print\_solution(solution)

else:

print("No solution exists.")

OUTPUT:

