(1) A\* Algorithm – 8-puzzle game

**Input-**

import heapq

class PuzzleState:

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

self.board = board

self.zero\_pos = zero\_pos

self.moves = moves

self.parent = parent

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

def heuristic(self):

"""Calculate the Manhattan distance as the heuristic."""

return self.manhattan\_distance()

def manhattan\_distance(self):

"""Compute the Manhattan distance for the current board state."""

distance = 0

for i in range(3):

for j in range(3):

tile = self.board[i][j]

if tile != 0: # Exclude the empty space

target\_x = (tile - 1) // 3

target\_y = (tile - 1) % 3

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

return distance

def generate\_successors(self):

"""Generate successor states by moving the empty tile (0)."""

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))

return successors

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

return self.cost < other.cost

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

"""Implement A\* search algorithm."""

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)

open\_set = []

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

closed\_set = set()

while open\_set:

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

if current\_state.board == 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):

"""Print the solution path."""

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(prompt):

"""Get board configuration from user input."""

board = []

print(prompt)

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

# Main execution

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

initial\_board = get\_board\_input("Enter the initial board configuration (use 0 for the empty space):")

goal\_board = get\_board\_input("Enter the goal board configuration (use 0 for the empty space):")

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

if solution:

print("Solution found:")

print\_solution(solution)

else:

print("No solution exists.")

Output:

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

Row 1 (space-separated numbers): 2 8 3

Row 2 (space-separated numbers): 1 6 4

Row 3 (space-separated numbers): 0 7 5

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

Row 1 (space-separated numbers): 1 2 3

Row 2 (space-separated numbers): 8 0 4

Row 3 (space-separated numbers): 7 6 5

Solution found:

[2, 8, 3]

[1, 6, 4]

[0, 7, 5]

[2, 8, 3]

[1, 6, 4]

[7, 0, 5]

[2, 8, 3]

[1, 0, 4]

[7, 6, 5]

[2, 0, 3]

[1, 8, 4]

[7, 6, 5]

[0, 2, 3]

[1, 8, 4]

[7, 6, 5]

[1, 2, 3]

[0, 8, 4]

[7, 6, 5]

[1, 2, 3]

[8, 0, 4]

[7, 6, 5]

=== Code Execution Successful ===