**LAB – 3**

Implement 8 puzzle problem using A\* search algorithm.

from heapq import heappush, heappop

# Goal state

goal\_state = [

[1, 2, 3],

[8, 0, 4],

[7, 6, 5]

]

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

def is\_goal(state):

return state == goal\_state

def misplaced\_tiles(state):

"""Count number of misplaced tiles compared to goal."""

count = 0

for i in range(3):

for j in range(3):

if state[i][j] != 0 and state[i][j] != goal\_state[i][j]:

count += 1

return count

def manhattan\_distance(state):

"""Calculate total Manhattan distance of all tiles from goal."""

distance = 0

for i in range(3):

for j in range(3):

tile = state[i][j]

if tile != 0:

goal\_x, goal\_y = divmod(tile - 1, 3)

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

return distance

def get\_neighbors(state):

neighbors = []

# Find blank position

for i in range(3):

for j in range(3):

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

x, y = i, j

break

for dx, dy in directions:

nx, ny = x + dx, y + dy

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

new\_state = [list(row) for row in state]

new\_state[x][y], new\_state[nx][ny] = new\_state[nx][ny], new\_state[x][y]

neighbors.append(new\_state)

return neighbors

def state\_to\_tuple(state):

return tuple(tuple(row) for row in state)

def reconstruct\_path(came\_from, current):

path = []

while current in came\_from:

path.append(current)

current = came\_from[current]

path.reverse()

return path

def a\_star\_search(initial\_state, heuristic\_func):

open\_list = []

closed\_set = set()

g\_score = {state\_to\_tuple(initial\_state): 0}

f\_score = {state\_to\_tuple(initial\_state): heuristic\_func(initial\_state)}

came\_from = {

heappush(open\_list, (f\_score[state\_to\_tuple(initial\_state)], initial\_state))

while open\_list:

current\_f, current\_state = heappop(open\_list

if is\_goal(current\_state):

return reconstruct\_path(came\_from, state\_to\_tuple(current\_state))

closed\_set.add(state\_to\_tuple(current\_state))

for neighbor in get\_neighbors(current\_state):

neighbor\_t = state\_to\_tuple(neighbor)

if neighbor\_t in closed\_set:

continue

tentative\_g\_score = g\_score[state\_to\_tuple(current\_state)] + 1

if neighbor\_t not in g\_score or tentative\_g\_score < g\_score[neighbor\_t]:

came\_from[neighbor\_t] = state\_to\_tuple(current\_state)

g\_score[neighbor\_t] = tentative\_g\_score

f\_score[neighbor\_t] = tentative\_g\_score + heuristic\_func(neighbor)

heappush(open\_list, (f\_score[neighbor\_t], neighbor))

return None # No solution found

# Example initial state

initial\_state = [

[2, 8, 3],

[1, 6, 4],

[7, 0, 5]

]

print("Using Misplaced Tiles heuristic:")

solution = a\_star\_search(initial\_state, misplaced\_tiles)

if solution:

for step in solution:

for row in step:

print(row)

print("------")

else:

print("No solution found.")

print("\nUsing Manhattan Distance heuristic:")

solution = a\_star\_search(initial\_state, manhattan\_distance)

if solution:

for step in solution:

for row in step:

print(row)

print("------")

else:

print("No solution found.")

**Output:**

