**Experiment 1:**

**Aim:** To write a python program to implement a 8 puzzle program.

**Algorithm:**

1.Define the problem with an initial and goal state.

2.Use Manhattan distance as the heuristic.

3.Implement the A\* algorithm to evaluate states.

4.Generate valid successors and avoid revisiting states.

5.Stop when the goal state is reached or declare unsolvable.

Program:

from queue import PriorityQueue

# Define the goal state

def find\_position(state, value):

"""Find the position of a value in the puzzle state."""

for i, row in enumerate(state):

for j, val in enumerate(row):

if val == value:

return i, j

return None

def calculate\_manhattan\_distance(state, goal\_state):

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

distance = 0

for i in range(len(state)):

for j in range(len(state[i])):

if state[i][j] != 0: # Ignore the blank tile

goal\_pos = find\_position(goal\_state, state[i][j])

distance += abs(i - goal\_pos[0]) + abs(j - goal\_pos[1])

return distance

def get\_neighbors(state):

"""Get all possible states from the current state."""

neighbors = []

size = len(state)

blank\_pos = find\_position(state, 0) # Find the blank tile position

x, y = blank\_pos

# Directions: up, down, left, right

directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]

for dx, dy in directions:

nx, ny = x + dx, y + dy

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

# Create a new state with the blank tile swapped

new\_state = [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 a\_star\_8\_puzzle(start\_state, goal\_state):

"""Solve the 8-puzzle using A\* search."""

open\_list = PriorityQueue()

open\_list.put((0, start\_state))

came\_from = {}

g\_cost = {tuple(map(tuple, start\_state)): 0}

f\_cost = {tuple(map(tuple, start\_state)): calculate\_manhattan\_distance(start\_state, goal\_state)}

while not open\_list.empty():

\_, current = open\_list.get()

# Check if the goal is reached

if current == goal\_state:

path = []

while tuple(map(tuple, current)) in came\_from:

path.append(current)

current = came\_from[tuple(map(tuple, current))]

path.append(start\_state)

path.reverse()

return path

current\_tuple = tuple(map(tuple, current))

for neighbor in get\_neighbors(current):

neighbor\_tuple = tuple(map(tuple, neighbor))

tentative\_g\_cost = g\_cost[current\_tuple] + 1 # All moves have a cost of 1

if neighbor\_tuple not in g\_cost or tentative\_g\_cost < g\_cost[neighbor\_tuple]:

g\_cost[neighbor\_tuple] = tentative\_g\_cost

f\_cost[neighbor\_tuple] = tentative\_g\_cost + calculate\_manhattan\_distance(neighbor, goal\_state)

open\_list.put((f\_cost[neighbor\_tuple], neighbor))

came\_from[neighbor\_tuple] = current

return None

def input\_puzzle(prompt):

"""Take a puzzle input from the user."""

print(prompt)

puzzle = []

print("Enter the puzzle row by row, with 0 for the blank tile. Separate numbers with spaces.")

for i in range(3):

while True:

try:

row = list(map(int, input(f"Row {i + 1}: ").strip().split()))

if len(row) != 3:

raise ValueError("Each row must have exactly 3 numbers.")

if not all(0 <= x <= 8 for x in row):

raise ValueError("Numbers must be between 0 and 8.")

puzzle.append(row)

break

except ValueError as e:

print("Invalid input:", e)

return puzzle

# Main function

def main():

print("8-Puzzle Problem Solver using A\* Algorithm")

start\_state = input\_puzzle("\nEnter the start state:")

goal\_state = input\_puzzle("\nEnter the goal state:")

print("\nSolving the 8-puzzle problem...\n")

path = a\_star\_8\_puzzle(start\_state, goal\_state)

if path:

print("Solution found!")

for step, state in enumerate(path):

print(f"\nStep {step}:")

for row in state:

print(" ".join(map(str, row)))

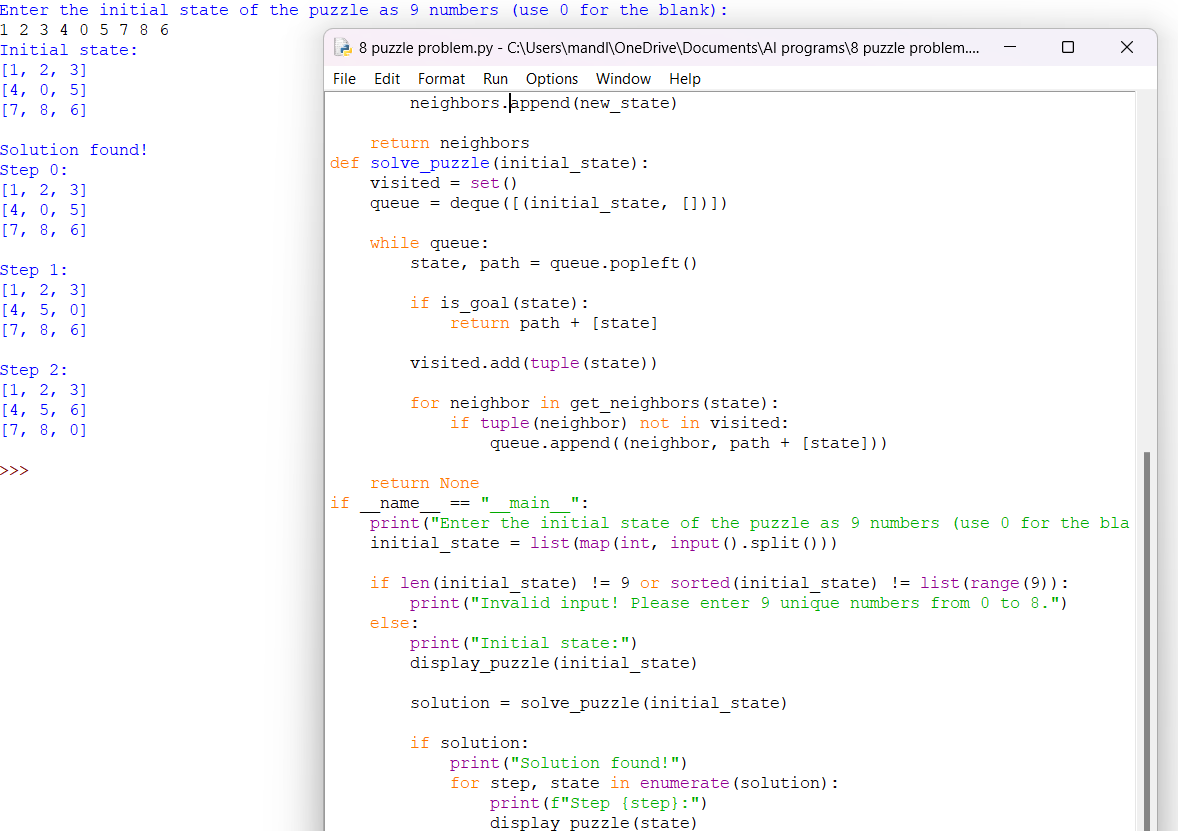
else:

print("No solution exists for the given start and goal states.")

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

main()

**Output:**

****

**Result:** Thus, the program was successfully completed using python programming.