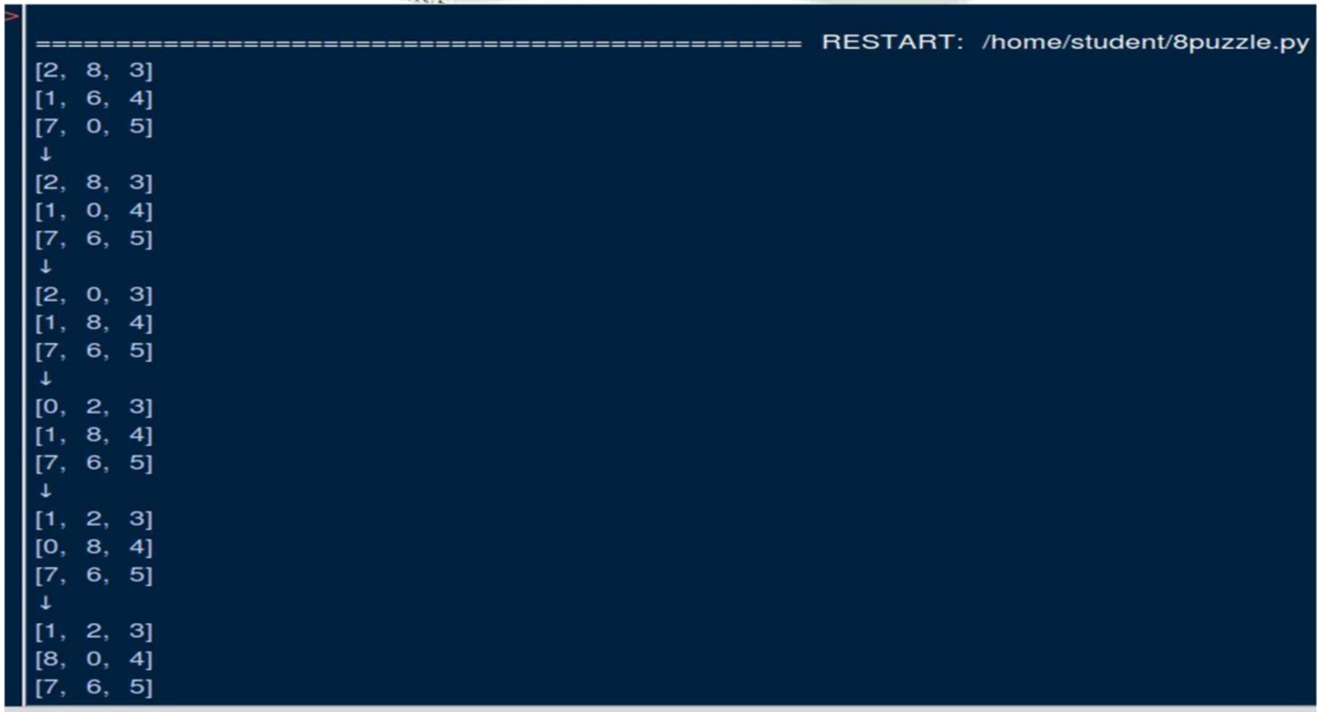
```
    return None, None  # No path found
graph = {  'A': [('B', 6), ('F', 3)],
     'B': [('A', 6), ('C', 3), ('D', 2)],
     'C': [('B', 3), ('D', 1), ('E', 5)],
     'D': [('B', 2), ('C', 1), ('E', 8)],
     'E': [('C', 5), ('D', 8), ('I', 5), ('J', 5)],
     'F': [('A', 3), ('G', 1), ('H', 7)],
     'G': [('F', 1), ('I', 3)],
     'H': [('F', 7), ('I', 2)],
     'I': [('E', 5), ('G', 3), ('H', 2), ('J', 3)],
     'J': [('E', 5), ('I', 3)]}
heuristic = { 'A': 10, 'B': 8, 'C': 5, 'D': 7, 'E': 3, 'F': 6, 'G': 5, 'H': 3, 'I': 1, 'J': 0}
start, goal = 'A', 'J'
path, total_cost = a_star(graph, start, goal, heuristic)
if path:
     print("Path found:", path)
     print("Shortest distance:", total_cost)
else:
     print("No path found")
```

OUTPUT:



```
>
================================================ RESTART: /home/student/8puzzle.py
[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]
```

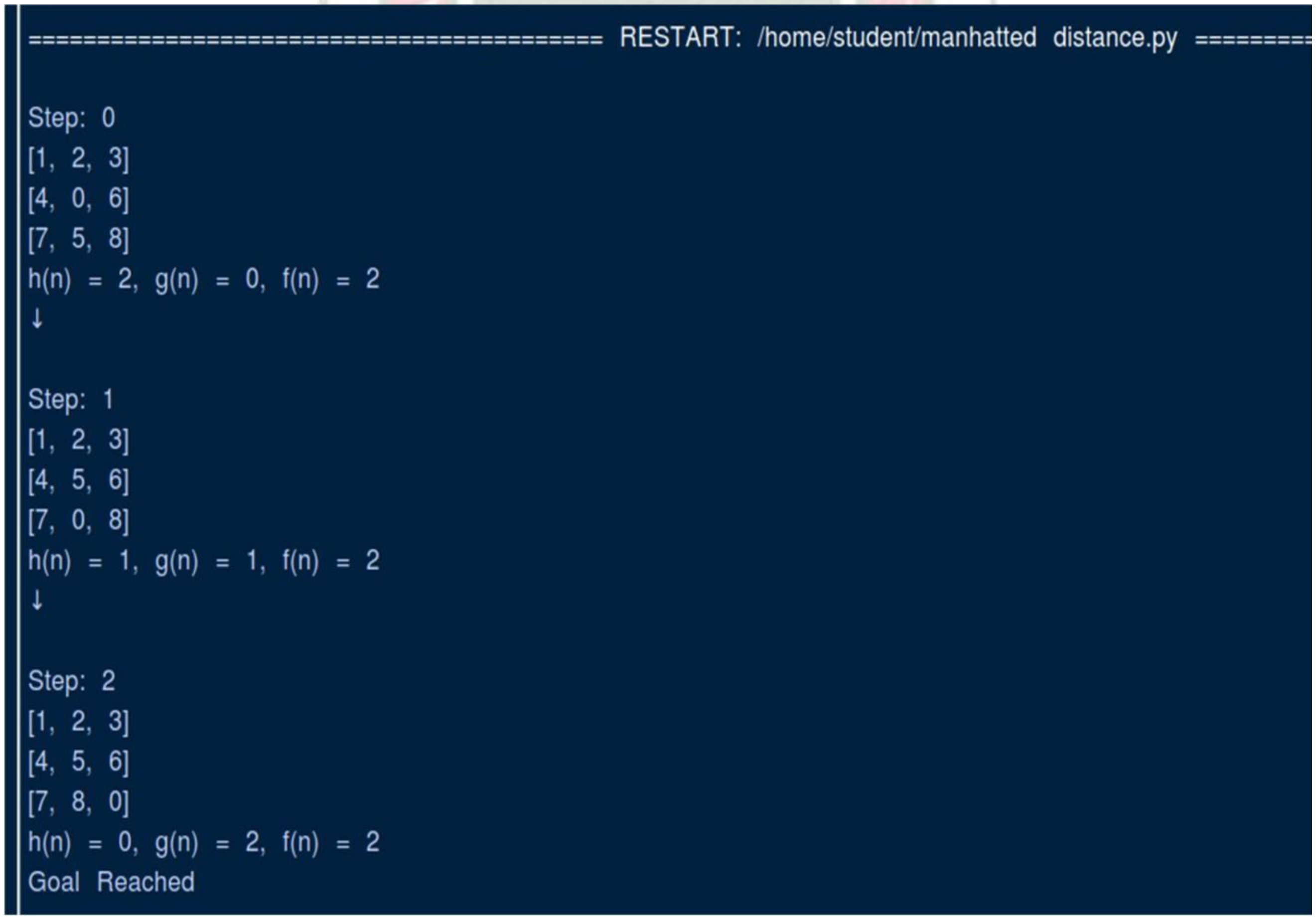
AIM: Implement 8 puzzle problem Using Manhattan Distance

DISCRIPTION: The A* algorithm using the Manhattan Distance heuristic is commonly applied to grid-based pathfinding problems. The Manhattan Distance heuristic h(n) calculates the sum of the absolute differences between the current node's coordinates and the goal node's coordinates (h(n) = |x1 - x2| + |y1 - y2|). The algorithm prioritizes nodes based on the cost function f(n) = g(n) + h(n), where g(n) is the actual cost from the start node. This heuristic is suitable for movement in a 4-directional grid (up, down, left, right) without diagonal movement. A* ensures an optimal path when the heuristic does not overestimate the actual cost. It is widely used in AI, robotics, and game development for pathfinding.

CODE:

```
 import heapq
class Puzzle:
    def _init_(self, board, g, parent=None):
        self.board = board
        self.g = g  # Depth (cost so far)
        self.h = self.manhattan_distance()
        self.f = self.g + self.h  # A* function: f(n) = g(n) + h(n)
        self.parent = parent  # To track the path
  def __lt__(self, other):
        return self.f < other.f  # Priority queue comparison
def manhattan_distance(self):
        goal_pos = {1: (0, 0), 2: (0, 1), 3: (0, 2),
                    4: (1, 0), 5: (1, 1), 6: (1, 2),
                    7: (2, 0), 8: (2, 1), 0: (2, 2)}
distance = 0
        for i in range(3):
            for j in range(3):
                value = self.board[i][j]
                if value != 0:  # Ignore blank tile
                    goal_x, goal_y = goal_pos[value]
                    distance += abs(i - goal_x) + abs(j - goal_y)
        return distance
def get_blank_pos(self):
        """Find blank space (0)"""
        for i in range(3):
            for j in range(3):
                if self.board[i][j] == 0:
                    return i, j
def generate_moves(self):
        """Generate possible moves"""
        x, y = self.get_blank_pos()
        moves = []
        directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]  # Up, Down, Left, Right
for dx, dy in directions:
            nx, ny = x + dx, y + dy
            if 0 <= nx < 3 and 0 <= ny < 3:
                new_board = [row[:] for row in self.board]
                new_board[x][y], new_board[nx][ny] = new_board[nx][ny], new_board[x][y]
                moves.append(Puzzle(new_board, self.g + 1, self))
return moves
def solve_puzzle(initial_state):
    """A* Search to solve the 8-puzzle"""
    start = Puzzle(initial_state, 0)
    goal = [[1, 2, 3], [4, 5, 6], [7, 8, 0]]
open_list = []
    closed_set = set()
    heapq.heappush(open_list, start)
while open_list:
        current = heapq.heappop(open_list)
if current.board == goal:
            path = []
            while current:
                path.append(current)
                current = current.parent
            return path[::-1]  # Reverse to get the correct order
closed_set.add(tuple(map(tuple, current.board)))
for move in current.generate_moves():
            if tuple(map(tuple, move.board)) not in closed_set:
                heapq.heappush(open_list, move)
return None  # No solution found
initial_state = [[1, 2, 3], [4, 0, 6], [7, 5, 8]]
solution = solve_puzzle(initial_state)
if solution:
    for step in solution:
        print("\nStep:", step.g)
        for row in step.board:
            print(row)
        print(f"h(n) = {step.h}, g(n) = {step.g}, f(n) = {step.f}")
        print("↓" if step.board != [[1, 2, 3], [4, 5, 6], [7, 8, 0]] else "Goal Reached 驩
else:
    print("No solution found.")
```

OUTPUT:



```
========================================= RESTART: /home/student/manhatted distance.py =========

Step: 0
[1, 2, 3]
[4, 0, 6]
[7, 5, 8]
h(n) = 2, g(n) = 0, f(n) = 2
↓

Step: 1
[1, 2, 3]
[4, 5, 6]
[7, 0, 8]
h(n) = 1, g(n) = 1, f(n) = 2
↓

Step: 2
[1, 2, 3]
[4, 5, 6]
[7, 8, 0]
h(n) = 0, g(n) = 2, f(n) = 2
Goal Reached
```