8-Puzzle Solver Using A* Algorithm Report

Name: Vineet Singh

Roll No.: 202401100400212

Introduction

The 8-Puzzle problem is a classic problem in artificial intelligence where a 3x3 grid of numbered tiles must be arranged in a goal configuration. The blank tile (represented by 0) can be moved up, down, left, or right. This program implements the A* search algorithm with the Manhattan Distance heuristic to efficiently find the optimal solution.

Methodology

The program uses the A* algorithm to find the shortest path to solve the puzzle. The Manhattan Distance heuristic is used to estimate the cost from the current state to the goal state. A priority queue is used to always expand the node with the lowest cost first. The algorithm tracks visited states to avoid unnecessary computations and loops.

Code

```
import heapq

# Class to store puzzle state and path
class PuzzleNode:
    def __init__(self, state, parent=None, move=None, depth=0, cost=0):
        self.state = state
        self.parent = parent
```

```
        self.move = move
        self.depth = depth
        self.cost = cost

    def __lt__(self, other):
        return self.cost < other.cost

# Function to calculate Manhattan Distance heuristic
def manhattan_distance(state, goal):
    distance = 0
    for i in range(3):
        for j in range(3):
            value = state[i][j]
            if value != 0:
                goal_x, goal_y = divmod(goal.index(value), 3)
                distance += abs(i - goal_x) + abs(j - goal_y)
    return distance

# Function to find all possible moves
def get_possible_moves(state):
    moves = []
    blank_x, blank_y = [(r, c) for r in range(3) for c in range(3) if state[r][c] == 0][0]
    directions = {'Up': (-1, 0), 'Down': (1, 0), 'Left': (0, -1), 'Right': (0, 1)}

    for move, (dx, dy) in directions.items():
        new_x, new_y = blank_x + dx, blank_y + dy
        if 0 <= new_x < 3 and 0 <= new_y < 3:
            new_state = [row[:] for row in state]
            new_state[blank_x][blank_y], new_state[new_x][new_y] = new_state[new_x][new_y],
new_state[blank_x][blank_y]
            moves.append((move, new_state))
    return moves

# A* Algorithm to solve the 8-puzzle
def a_star_solver(start, goal):
    start_flat = sum(start, [])
    goal_flat = sum(goal, [])

    open_list = []
    heapq.heappush(open_list, PuzzleNode(start, None, None, 0, manhattan_distance(start,
goal_flat)))
    visited = set()
```

```
    while open_list:
        current = heapq.heappop(open_list)

        if current.state == goal:
            path = []
            while current.parent:
                path.append(current.move)
                current = current.parent
            return path[::-1]

        visited.add(tuple(sum(current.state, [])))

        for move, new_state in get_possible_moves(current.state):
            if tuple(sum(new_state, [])) not in visited:
                new_cost = current.depth + 1
                total_cost = new_cost + manhattan_distance(new_state, goal_flat)
                heapq.heappush(open_list, PuzzleNode(new_state, current, move, new_cost,
total_cost))

    return None

# Function to take user input
def get_input():
    print("Enter the 8-puzzle (use 0 for the blank space):")
    state = []
    for i in range(3):
        row = list(map(int, input().split()))
        state.append(row)
    return state

# Take input for start and goal states
print("Enter the start state:")
start_state = get_input()

print("Enter the goal state:")
goal_state = get_input()

# Solve the puzzle
solution = a_star_solver(start_state, goal_state)

# Print the solution steps
if solution:
    print("\nSteps to solve:", solution)
```

else:
    print("\nNo solution found. Try again!")

Output/Result

```
Enter the start state:
Enter the 8-puzzle (use 0 for the blank space):
1 2 3
5 6 7
8 4 0
Enter the goal state:
Enter the 8-puzzle (use 0 for the blank space):
1 2 3
4 5 6
7 8 0

Steps to solve: ['Up', 'Left', 'Left', 'Down', 'Right', 'Right', 'Up', 'Left', 'Left', 'Down', 'Right', 'Up', 'Right', 'Down']
```

.

References/Credits

This implementation is based on the A* algorithm and the Manhattan Distance heuristic commonly used in AI search problems. No external sources were used.