8-PUZZLE

SOLVER

USING

ALGORITHMS

TITLE PAGE

Project: 8-Puzzle Solver using Algorithm

Student Name: Devendra Kumar Yadav
Roll Number: 2024011004000079(06)
Branch: CSE AIML
Institution: KIET Group of Institution

Date: 10/03/2025

1. Introduction

The 8-Puzzle Problem is a classic challenge in artificial intelligence that involves organizing numbered tiles within a 3×3 grid to match a specific goal arrangement. The puzzle has eight numbered tiles and one empty space, which allows the tiles to be moved around. The goal is to shift the tiles using valid moves (Up, Down, Left, or Right) until they are correctly arranged.

To solve this efficiently, we use the *A Search Algorithm**, which helps find the best path by making smart decisions at each step. It relies on a heuristic function—in this case, the Manhattan Distance heuristic—to estimate how many moves are needed to reach the solution in the shortest possible way.

2. Methodology

Problem Representation:

- The puzzle is defined as a 3×3 matrix with each tile labeled by a number.
- The empty tile (represented by 0) allows movement of surrounding tiles.

Algorithm Utilized: A* Search Algorithm

Efficiently searches for the solution path with the least cost using the evaluation function:

f(n) = g(n) + h(n)

- g(n): Moves executed from the start state.
- h(n): The heuristic function (Manhattan Distance in this case).

Steps Involved:

1. Start by adding the initial puzzle state to a priority queue.
2. Always pick the state with the lowest cost to explore next.
3. Generate neighboring states by shifting the blank tile in legal directions.
4. Compute Manhattan Distance for each new state.
5. Continue expanding the best state until the goal state is achieved.
6. Trace back the steps taken to find the sequence of moves that led to the solution.

3. Code Implementation

The solution is implemented in Python using the `heapq` priority queue for efficient state expansion. Below is the core logic:

```
import heapq

class PuzzleState:
    def __init__(self, board, parent=None, move="", depth=0, cost=0):
```

```
        self.board = board
        self.parent = parent
        self.move = move
        self.depth = depth
        self.cost = cost

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

def get_manhattan_distance(board, goal):
    """Calculates Manhattan Distance heuristic."""
    distance = 0
    flat_goal = sum(goal, [])  # Flatten the goal state (2D → 1D)

    for i in range(3):
        for j in range(3):
            if board[i][j] == 0:  # Ignore empty tile
                continue
            x, y = divmod(flat_goal.index(board[i][j]), 3)  # Correct position lookup
            distance += abs(i - x) + abs(j - y)

    return distance

def solve_8_puzzle(initial_state, goal_state):
    start = PuzzleState(initial_state, cost=get_manhattan_distance(initial_state,
goal_state))
    open_list = []
    heapq.heappush(open_list, start)
    closed_set = set()

    while open_list:
        current_state = heapq.heappop(open_list)

        if current_state.board == goal_state:
            path = []
            while current_state.parent:
                path.append(current_state.move)
                current_state = current_state.parent
            return path[::-1]  # Return moves from start to goal

        closed_set.add(tuple(map(tuple, current_state.board)))
```

```
        for neighbor in get_neighbors(current_state, goal_state):
            if tuple(map(tuple, neighbor.board)) in closed_set:
                continue
            heapq.heappush(open_list, neighbor)

    return None  # No solution found
```

4. Output/Result

For the initial state:
1 2 3
5 6 0
7 8 4

And for the goal state:
1 2 3
4 5 6
7 8 0

```
Initial State:
1 2 3
5 6 0
7 8 4


Solution Found!
Moves: Left -> Left -> Down -> Right -> Right -> Up -> Left -> Down -> Left -> Up -> Right -> Right -> Down
```

The sequence of moves for the solution is:
Solution found: Down -> Right -> Up -> Left -> Down -> Right

5. References & Credits

- Artificial Intelligence: A Modern Approach (Stuart Russell, Peter Norvig)
- Online AI resources for Algorithm
- Python Documentation

6. Conclusion

This project successfully applied the A* Search Algorithm to solve the 8-Puzzle Problem. By using a heuristic function, we were able to significantly narrow down the search space, making the solution process more efficient. The algorithm effectively finds the optimal path to the goal state. Looking ahead, we could explore different heuristics, such as the Misplaced Tiles heuristic, to see how they impact performance.