KIET Group Of Institutions,Ghaziabad

Department of Computer Science Engineering

Specialization in Artificial Intelligence and Machine Learning

8-PUZZLE SOLVER IN PYTHON

Submitted By:

Shaurya Rastogi

Roll No-:202401100400172

Submitted To:

Mrs. Bhavna Bansal

Course: Introduction To AI

Date: 10/03/2025

Introduction

The 8-puzzle solver is a well-known problem in artificial intelligence and computer science. It's a sliding tile puzzle consisting of a 3x3 grid with eight numbered tiles and one empty space. The goal is to move the tiles around until they're arranged in a specific order — usually from 1 to 8, with the empty space at the end.

The Python code provided for the 8-puzzle solver uses the *A (A-star) search algorithm**, which is one of the most efficient path-finding and search algorithms. A* works by combining:

- **Cost to reach the current state (g)**: The number of moves made so far
- **Estimated cost to reach the goal (h)**: A heuristic — here we use **Manhattan Distance**, which calculates the total distance of tiles from their goal positions

The program works by:

1. **Starting from the initial state** of the puzzle
2. **Generating possible moves** (like sliding tiles up, down, left, or right)
3. **Evaluating the cost and heuristic value** of each possible state
4. **Choosing the state with the lowest total cost** and repeating until the goal state is reached

The output is the sequence of moves needed to solve the puzzle in the shortest possible number of steps.

Methodology

The methodology of the 8-puzzle solver code using the A* search algorithm can be broken down into clear steps:

1. **Problem Representation:**
   - The 8-puzzle is represented as a **3x3 grid**, where the numbers 1–8 are tiles and **0** represents the empty space.
   - Each state of the puzzle is a unique arrangement of these numbers.
   - The **goal state** is the final, sorted arrangement of tiles.
2. **State Space and Moves:**
   - Possible moves: **Up**, **Down**, **Left**, **Right** (sliding the empty space).
   - Each move creates a **new state** by swapping the empty space (0) with an adjacent tile.
3. **Heuristic Function:**
   - We use the **Manhattan Distance** heuristic, which measures how far each tile is from its correct position in the goal state.
4. *A Search Algorithm:**
   A* combines:
   - **g(n): Cost to reach current state** (number of moves made)
   - **h(n): Estimated cost to reach goal** (Manhattan distance)
   - **f(n) = g(n) + h(n): Total estimated cost**

   Steps of the A* algorithm:

   - **Start with the initial state** and push it into a priority queue (min-heap) based on the total cost f(n)f(n)f(n).
   - **Explore the state with the lowest cost**, check if it's the goal state.
   - **Generate neighboring states** by sliding the empty space and calculate their cost.
   - **Add neighbors to the priority queue** if they haven't been explored yet.
   - **Repeat until the goal state is found** or the queue is empty (unsolvable puzzle).
5. **Solution Extraction:**
   Once the goal state is reached, we **trace back the moves** from the goal state to the initial state and return the sequence of moves (Up, Down, Left, Right) in the correct order.

Code Typed

```
# 8-Puzzle Solver Code
import heapq

# Class representing a node in the puzzle
class PuzzleNode:
    def __init__(self, state, parent=None, move=None, cost=0, depth=0):
        self.state = state              # Current state of the puzzle
        self.parent = parent            # Parent node (previous state)
        self.move = move                # Move made to reach this state
        self.cost = cost                # Heuristic cost of this state
        self.depth = depth              # Number of moves made so far

    # Comparison functions for priority queue
    def __lt__(self, other):
        return (self.cost + self.depth) < (other.cost + other.depth)

    def __eq__(self, other):
        return self.state == other.state

    def __hash__(self):
        return hash(str(self.state))

# Heuristic function: Manhattan distance
# Calculates how far tiles are from their goal positions
def heuristic(state, goal):
    distance = 0
    for i in range(3):
        for j in range(3):
            if state[i][j] != 0:
                x, y = divmod(goal.index(state[i][j]), 3)
                distance += abs(x - i) + abs(y - j)
    return distance

# Generate all possible neighboring states by sliding the empty tile (0)
def get_neighbors(state):
    neighbors = []
    # Find position of the empty tile
    x, y = next((i, j) for i, row in enumerate(state) for j, val in
enumerate(row) if val == 0)
    # Possible moves
    moves = [(x-1, y, 'Up'), (x+1, y, 'Down'), (x, y-1, 'Left'), (x, y+1,
'Right')]
```

```
    for nx, ny, move in moves:
        if 0 <= nx < 3 and 0 <= ny < 3:
            # Create a new state by swapping empty tile with adjacent tile
            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, move))

    return neighbors

# A* search algorithm to solve the 8-puzzle
def solve_8_puzzle(start, goal):
    goal_flat = [num for row in goal for num in row]  # Flatten goal state
for easy indexing
    start_node = PuzzleNode(start, cost=heuristic(start, goal_flat))
    frontier = [start_node]  # Priority queue for A* search
    explored = set()         # Set of visited states

    while frontier:
        current = heapq.heappop(frontier)  # Get state with lowest cost

        if current.state == goal:  # Goal state reached
            moves = []
            while current.parent:
                moves.append(current.move)
                current = current.parent
            return moves[::-1]  # Reverse moves to get correct order

        explored.add(current)

        for neighbor, move in get_neighbors(current.state):
            neighbor_node = PuzzleNode(neighbor, current, move,
heuristic(neighbor, goal_flat), current.depth + 1)
            if neighbor_node not in explored and neighbor_node not in
frontier:
                heapq.heappush(frontier, neighbor_node)

    return None  # No solution found

# Main function to execute the solver
if __name__ == "__main__":
    start = [[1, 2, 3], [4, 0, 5], [6, 7, 8]]  # Starting configuration
    goal = [[1, 2, 3], [4, 5, 6], [7, 8, 0]]   # Goal configuration
    solution = solve_8_puzzle(start, goal)
```

```
    if solution:
        print(f"Solution found in {len(solution)} moves: {' ->
'.join(solution)}")
    else:
        print("No solution found")
```

Screenshot of Output