8 PUZZLE

SOLVER

- Name:- Akshit Jawla
- Roll No:-202401100400026
- Branch:-CSE(AIML)
- Section:-A

PROBLEM STATEMENT

The 8-puzzle is a classic sliding puzzle that consists of a 3x3 grid with 8 numbered tiles (1 through 8) and one empty space (represented as 0). The objective is to rearrange the tiles from a given initial configuration to reach the goal configuration by sliding the tiles into the empty space.

1.Initial State: A scrambled 3x3 grid of tiles.

2.Goal State: The solved configuration of the puzzle:

1 2 3

4 5 6

7 8 0

3. Valid Moves: The empty space can move up, down, left, or right into an adjacent tile, provided the move is within the bounds of the grid.

4. Objective: Find the shortest sequence of moves to transform the initial state into the goal state.

Approach

To solve the 8-puzzle problem, you can use one of the following algorithms:

1.Breadth-First Search (BFS):

Guarantees the shortest path.

Explores all possible states level by level.

Uses a queue to manage states and a set to track visited states.

2.A Algorithm*:

Uses a heuristic function to guide the search.

3.Common heuristics include:

Manhattan Distance: Sum of the distances of each tile from its goal position.

Hamming Distance: Number of tiles in the wrong position.

Prioritizes states with the lowest cost ($f(n) = g(n) + h(n)$), where:

$g(n)$ = cost to reach the current state.

$h(n)$ = heuristic estimate of the cost to reach the goal.

Key Challenges

1.State Space Explosion: The 8-puzzle has 9! (362,880) possible states, so efficient algorithms and data structures are required.

2.Heuristic Design: For A*, the heuristic must be admissible (never overestimates the cost) to guarantee optimality.

3.Memory Management: BFS and A* can consume significant memory for large state spaces.

CODE

```
from heapq import heappush, heappop

# Function to calculate the Manhattan Distance heuristic

def manhattan_distance(state, goal):

    """Calculate the Manhattan Distance heuristic between the current state and goal."""

    distance = 0

    for i in range(9):  # Iterate through all tiles

        if state[i] != 0:  # Skip the empty tile

            # Calculate current position (x1, y1)

            x1, y1 = i % 3, i // 3

            # Calculate goal position (x2, y2)

            x2, y2 = (goal.index(state[i])) % 3, (goal.index(state[i])) // 3

            # Add the Manhattan distance for the current tile

            distance += abs(x1 - x2) + abs(y1 - y2)

    return distance

# Function to generate neighboring states

def get_neighbors(state):

    """Generate valid neighboring states by moving the empty tile (0)."""

    neighbors = []

    zero_idx = state.index(0)  # Find the position of the empty tile

    x, y = zero_idx % 3, zero_idx // 3  # Get its coordinates

    # Define possible movements (Left, Right, Up, Down)

    for dx, dy in [(-1, 0), (1, 0), (0, -1), (0, 1)]:

        nx, ny = x + dx, y + dy  # New coordinates after movement

        if 0 <= nx < 3 and 0 <= ny < 3:  # Check boundaries

            swap_idx = ny * 3 + nx  # Get the 1D index after movement

            neighbor = list(state)  # Copy the current state

            # Swap the empty tile with the target tile

            neighbor[zero_idx], neighbor[swap_idx] = neighbor[swap_idx], neighbor[zero_idx]

            neighbors.append(tuple(neighbor))  # Add the new state to neighbors

    return neighbors

# Function to check if the puzzle is solvable

def is_solvable(state, goal=(1, 2, 3, 4, 5, 6, 7, 8, 0)):
```

```python
    """Check solvability using the inversion count."""
    inversions = 0
    # Remove the empty tile and count inversions in the remaining tiles
    state = [num for num in state if num != 0]
    for i in range(len(state)):
        for j in range(i + 1, len(state)):
            if state[i] > state[j]:
                inversions += 1
    return inversions % 2 == 0  # Puzzle is solvable if inversions are even
# A* Search algorithm to solve the 8-puzzle
def a_star_solver(initial, goal=(1, 2, 3, 4, 5, 6, 7, 8, 0)):
    """Solve the 8-puzzle using A* Search with Manhattan Distance heuristic."""
    if not is_solvable(initial, goal):  # Check if the puzzle is solvable
        return None  # Return None if unsolvable

    heap = []  # Priority queue for A* search
    # Push the initial state with priority based on heuristic
    heappush(heap, (manhattan_distance(initial, goal), 0, initial, []))
    visited = set()  # Keep track of visited states to avoid loops
    while heap:  # While there are states to explore
        _, cost, current, path = heappop(heap)  # Get the state with the lowest priority
        if current == goal:  # Check if the goal state is reached
            return path + [current]  # Return the solution path
        if current in visited:  # Skip already visited states
            continue
        visited.add(current)  # Mark the state as visited
        for neighbor in get_neighbors(current):  # Explore all neighbors
            if neighbor not in visited:  # Only consider unvisited states
                # Calculate priority for the neighbor state
                priority = cost + 1 + manhattan_distance(neighbor, goal)
                # Add the neighbor to the heap
                heappush(heap, (priority, cost + 1, neighbor, path + [current]))
    return None  # Return None if no solution is found
# Prompt user to input the initial state
print("Enter the initial state as 9 space-separated numbers (use 0 for the blank):")
initial_input = input().strip().split()  # Read input from the user
```

```
initial_state = tuple(map(int, initial_input))  # Convert input to a tuple of integers

# Solve the puzzle

solution = a_star_solver(initial_state)  # Call the A* solver

if solution:

    print(f"Solution found in {len(solution) - 1} moves:")

    for step, state in enumerate(solution):  # Print each step of the solution

        print(f"Step {step}:")

        # Format the state as a 3x3 grid

        print("\n".join(" ".join(map(str, state[i*3:(i+1)*3])) for i in range(3)))

        print()

else:

    print("No solution exists.")  # Notify the user if the puzzle is unsolvable
```

OUTPUT

Enter the initial state as 9 space-separated numbers (use 0 for the blank):

1 2 3 4 5 6 0 7 8

Solution found in 2 moves:

Step 0:

1 2 3

4 5 6

0 7 8

Step 1:

1 2 3

4 5 6

7 0 8

Step 2:

1 2 3

4 5 6

7 8 0