Name : Anuska Nath
Dept : IT (UG3)
Section : A3
Roll : 002311001003

Assignment 6

Problem:
Look at the classical 8-puzzle problem and its solution from a particular given start state to a specific end state.

Write a Python program to solve the problem using **A*** **Search** in such a way that it can handle the 15-Puzzle problem with the same program.

Choose any suitable heuristic function and execute it in the problem.

Provide necessary inputs to the program. Do not statically mention any of them inside the program.

Input and output states shall be written in the same file and you can read them properly as required.

Intermediate output shall be properly displayed and will be written in a separate output file.

Puzzle Solver Using A* Search

1) Description of Variables and Functions

- **Classes & Core Variables:**
  - **PuzzleNode**
    Represents a state in the search tree.
    - **state:** 2D list representing the puzzle board configuration.
    - **parent:** reference to the previous PuzzleNode.
    - **move:** move that led to this state (e.g., 'Up', 'Down').
    - **depth:** cost from start node to this node (g-cost).
    - **cost:** total cost f = g + h for A* priority.
      Implements < operator for priority queue ordering.
  - **PuzzleSolver**
    The main solver class.
    - **start, goal:** initial and target puzzle states.

        - **n:** puzzle dimension (3 for 8-puzzle, 4 for 15-puzzle).
        - **goal_pos:** dictionary mapping tile values to their goal coordinates for heuristic calculations.
- **Key Functions:**
    - **build_goal_position()**
      Maps each tile value to its goal position.
    - **manhattan_distance(state)**
      Calculates the heuristic: sum of Manhattan distances of all tiles from their goal positions.
    - **get_neighbors(state)**
      Returns all valid next states by sliding the blank tile (0) up/down/left/right.
    - **state_to_tuple(state)**
      Converts a 2D list state into a tuple of tuples to allow hashing and set membership checks.
    - **reconstruct_path(node)**
      Traces back from goal node to start node to recover the solution path.
    - **solve()**
      Implements the A* search algorithm using a priority queue to explore states by lowest cost f = g + h.
- **Utility Functions:**
    - **read_input_file(filename)**
      Reads puzzle size, start state, and goal state from a given input file (ignores blank lines).
    - **write_output(path, filename)**
      Writes the step-by-step solution path and total steps to an output file and prints to console.

---

2) Inputs

- The program reads from **input files** (input1.txt, input2.txt, etc.).
- Format of each input file:
    - First line: integer n specifying puzzle dimension (3 or 4).
    - Next n lines: start state matrix (each line has n integers).
    - Next n lines: goal state matrix (each line has n integers).
- Tiles are represented by integers; blank tile is represented by 0.

Input1.txt:

```
3

1 2 3
4 0 5
6 7 8

1 2 3
4 5 6
7 8 0
```

Input2.txt:

```
4

1 0 2 4
5 7 3 8
9 6 11 12
13 10 14 15

1 2 3 4
5 6 7 8
9 10 11 12
13 14 15 0
```

---

3) Outputs

- Writes the solution steps to an **output file** (output1.txt, output2.txt, etc.) and prints them on the console.
- Each step shows:
  - Step number
  - Move made (or "Initial" for the starting state)
  - Current puzzle state in matrix form
- At the end, prints the total number of moves to reach the goal.
- If no solution exists, outputs "No solution found."

Output1.txt:

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

Step 1: Right
1 2 3
4 5 0
6 7 8
```

Step 2: Down
1 2 3
4 5 8
6 7 0

Step 3: Left
1 2 3
4 5 8
6 0 7

Step 4: Left
1 2 3
4 5 8
0 6 7

Step 5: Up
1 2 3
0 5 8
4 6 7

Step 6: Right
1 2 3
5 0 8
4 6 7

Step 7: Down
1 2 3
5 6 8
4 0 7

Step 8: Right
1 2 3
5 6 8
4 7 0

Step 9: Up
1 2 3
5 6 0
4 7 8

Step 10: Left
1 2 3
5 0 6
4 7 8

Step 11: Left
1 2 3
0 5 6
4 7 8

Step 12: Down
1 2 3
4 5 6
0 7 8

Step 13: Right

1 2 3
4 5 6
7 0 8

Step 14: Right
1 2 3
4 5 6
7 8 0

Total steps: 14

Output2.txt:

Step 0: Initial
1 0 2 4
5 7 3 8
9 6 11 12
13 10 14 15

Step 1: Right
1 2 0 4
5 7 3 8
9 6 11 12
13 10 14 15

Step 2: Down
1 2 3 4
5 7 0 8
9 6 11 12
13 10 14 15

Step 3: Left
1 2 3 4
5 0 7 8
9 6 11 12
13 10 14 15

Step 4: Down
1 2 3 4
5 6 7 8
9 0 11 12
13 10 14 15

Step 5: Down
1 2 3 4
5 6 7 8
9 10 11 12
13 0 14 15

Step 6: Right
1 2 3 4
5 6 7 8
9 10 11 12
13 14 0 15

Step 7: Right
1 2 3 4
5 6 7 8
9 10 11 12
13 14 15 0

Total steps: 7

---

4) Algorithm:

- ***A Search Algorithm**:**
    - Explores puzzle states by expanding the node with the lowest estimated total cost f = g + h, where:
        - g = cost from start to current node (depth).
        - h = heuristic estimate of cost from current node to goal.
    - Uses the **Manhattan Distance** heuristic:
      Sum of the absolute differences of the current tile positions from their goal positions (ignoring the blank tile).
    - Keeps track of visited states to avoid revisiting and loops.
    - Continues expanding nodes until the goal state is found or no more states are available (no solution).