Assignment - 4

Name : Soham Das

Roll : 002311001004

Department : Information Technology (A3)

Solving the N-Puzzle Problem using Depth-First Search

In this assignment, I have developed a scalable program to solve the classic N-Puzzle problem, which includes variants like the 8-Puzzle and 15-Puzzle. The program approaches this challenge as a state-space search, where the objective is to find a valid sequence of moves to transition from a given initial configuration to a specified goal configuration.

My Approach

I chose to implement the solution using the **Depth-First Search (DFS)** algorithm. DFS is a graph traversal algorithm that explores as far as possible along each branch before backtracking. By treating each possible configuration of the puzzle as a "state" or "node" in a graph, DFS systematically explores potential solution paths.
The program is designed to be flexible, allowing the user to define the puzzle size (N), making it capable of handling an 8-Puzzle (3x3 grid), a 15-Puzzle (4x4 grid), or other sizes dynamically.

Implementation Details

My implementation incorporates several key features to ensure correctness, efficiency, and robustness.

- **State Representation**: Each state of the puzzle is represented as a 2D list (a matrix). To reconstruct the final solution path, I used a Node class that stores not only the state itself but also a reference to its parent node and the move (Up, Down, Left, or Right) taken to reach it.
- **Solvability Check**: A crucial feature of this program is its ability to determine if a puzzle is solvable *before* beginning the search. Approximately half of all possible starting configurations of an N-Puzzle are impossible to solve. By calculating the number of **inversions** (pairs of tiles that are out of order) and considering the position of the blank tile, the program can immediately inform the user if a solution exists, preventing a long and fruitless search.

- **Cycle Prevention**: To avoid getting stuck in infinite loops (e.g., moving a tile back and forth), the program maintains a visited set. Before exploring a new state, it checks if that state has already been visited. This dramatically improves efficiency by ensuring that each unique puzzle configuration is processed only once.
- **Depth-Limited Search**: A standard DFS can sometimes get lost exploring an extremely deep, non-optimal path. To mitigate this, my implementation uses a **maximum recursion depth**. If a solution is not found within this set depth, the search along that path is terminated, and the algorithm backtracks to explore other possibilities.

Input and Output Handling

The program is fully interactive and does not contain any hard-coded puzzle states.

- **User Input**: The user is prompted to enter the puzzle size (e.g., 3 for an 8-puzzle), the initial state, and the goal state row by row. The program includes validation to ensure the entered states are valid for the given puzzle size.
- **Intermediate Output File (n_puzzle_intermediate_steps.txt)**: To provide insight into the algorithm's process, all intermediate steps are logged to this file. It records every state the DFS algorithm visits, showing the depth of the search and the exact path of exploration and backtracking. This creates a transparent audit trail of the search process.
- **Final Output File (n_puzzle_results.txt)**: This file serves as the final report. It contains a clean record of the initial state, the goal state, and, if a solution is found, the complete, step-by-step path from start to finish. If no solution is found within the depth limit, this is also clearly stated.

Example: Solving an 8-Puzzle

Let's consider a simple 8-Puzzle (3x3) scenario.
Input:

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

- **Goal State:**
  1 2 3
  4 5 6
  7 8 0

Output:

Part 1: Intermediate Exploration (n_puzzle_intermediate_steps.txt)

This file would contain a detailed log of the DFS traversal. An excerpt might look like this, showing the algorithm exploring different moves from the initial state:

Visiting at depth 0:
1 2 3
0 4 6
7 5 8

Visiting at depth 1:
1 2 3
4 0 6
7 5 8

Visiting at depth 2:
1 0 3
4 2 6
7 5 8

...

This log demonstrates the "deep dive" nature of DFS, as it follows one path until it hits a dead end or the depth limit, then backtracks to try another.

Part 2: Solution Path (n_puzzle_results.txt)

Once the goal state is found, this file presents the final, reconstructed path.

--- Initial State ---
1 2 3
0 4 6
7 5 8

--- Goal State ---
1 2 3
4 5 6
7 8 0

--- Solution Path ---
Step 0: Initial State
1 2 3
0 4 6
7 5 8

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

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

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

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

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

... and so on until the goal is reached.

This final output provides a clear sequence of moves that successfully solves the puzzle.