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

Assignment 4

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 Depth-first search in such a way that it can handle the 15-Puzzle problem with the same program.

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.

Sliding-Tile Puzzle Solver (DFS with Pruning)

Problem:

- Solve n-puzzle (3×3 or 4×4) from given start to goal state.
- Allowed move: slide tile adjacent to blank (0).
- Input: input.txt (start + goal states).
- Output: output.txt (start → intermediates → goal, total steps).

Algorithm (DFS + Branch-and-Bound)

1. Represent the board as a tuple of tuples (hashable, immutable).
2. Use Depth-First Search:

    - Explore states recursively.
    - Maintain a visited set (for current recursion path).
3. Pruning:
    - If path length ≥ best known (best_path), stop exploring.
    - If depth > max_depth, backtrack.
4. Goal check: If state = goal → update best_path.
5. After search: print/write states in order, total moves = len(best_path)-1.

Guarantees shortest path if max_depth ≥ optimal depth.

Data Structures:

- State: tuple[tuple[int,...],...] (board config).
- Path: list of states.
- Visited: set of states in current recursion (avoids cycles).
- best_path: global list storing shortest solution.

Functions:

- read_state(filename): Parse input.txt, extract start and goal.
- print_state(state, step, file): Print/write a board state with step number.
- find_blank(state): Locate blank (0).
- swap(state, i1, j1, i2, j2): Return new state after swapping positions.
- get_neighbors(state): Generate valid moves (Up, Down, Left, Right).
- dfs_shortest(state, goal, path, visited, max_depth): Core DFS with pruning; updates best_path.

Variables:

- Global:
  - best_path: current shortest solution (None if none found).
- Main:
  - input_file, output_file: filenames.
  - start, goal: puzzle states.
  - visited: {start} initially.
  - path: [start] initially.
  - max_depth: recursion cutoff (default 30).
- Inside helpers:
  - n: board dimension.
  - (x,y): blank location.
  - (dx,dy): direction deltas.
  - step: step counter for output.

Execution Flow:

1. Read states from input.
2. Initialize visited and path.
3. Run dfs_shortest.
4. If solution:
   - Print/write Start, Step 0 … Goal.
   - Report “Goal reached in k steps!”.

5. Else: report “No solution within depth limit”.

Complexity:

- Time: O(bd)O(b^d)O(bd), where b≤4b ≤ 4b≤4, d=d =d= depth of solution.
- Space: O(d)O(d)O(d) recursion + best_path.
- Pruning improves performance once a solution is found.

Input:

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

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

Output (console):

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

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

Step 2:
1 2 3 4
5 6 7 8

9 10 11 12
13 14 15 _

Goal reached in 2 steps!

Output.txt:

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

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

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

Goal reached in 2 steps!