Introduction to AI

Assignment 1 –
Solving the 8 Puzzle

Data Structures Used

1. **BFS**
    a. Deque (Double-ended Queue)
    Used for the "frontier". Implements FIFO (First-In-First-Out) for level-order traversal.
    b. Set (Hash Set)
    "frontier_set" tracks states currently in frontier for O(1) membership checking and "explored" stores visited states to prevent revisiting. Uses tuple representation of board states for hashing.
    c. List
    For tracking expanded nodes and trace data as "expanded_nodes" stores copies of expanded board states and "trace_data" stores detailed execution information.
2. **DFS**
    a. List (Stack)
    Used for the "frontier". Implements LIFO (Last-In-First-Out) for depth-first traversal.
    b. Set (Hash Set)
    "frontier_set" tracks frontier states for O(1) membership checking and "explored" stores visited states. Uses tuple representation for hashing.
    c. List
    For tracking expanded nodes and trace data as "expanded_nodes" stores copies of expanded board states and "trace_data" stores detailed execution information.
3. **Iterative DFS**
    a. List (Stack)
    Used for the "frontier". Implements LIFO (Last-In-First-Out) for depth-first traversal.
    b. Set (Hash Set)

"frontier_set" tracks frontier states per depth iteration, "explored" stores visited states per iteration, and "expanded_set" tracks globally expanded nodes across all iterations (prevents duplicate counting).

c. List
"expanded_nodes" Reset per iteration, stores expanded states and "trace_data" accumulates across all depth iterations.

4. **A* Search**
   a. List (Min-Heap/Priority Queue)
   Used for the "frontier". Stores "AStarNode" objects ordered by f-cost (f = g + h).
   b. Dict (Hash Map)
   "frontier_dict" maps state tuples to "AStarNode" objects. Enables O(1) lookup to check if a state is in frontier. Allows updating node costs when better paths are found.
   "goal_positions" maps tile values to their goal coordinates. Used in both Manhattan and Euclidean distance calculations.
   c. Set (Hash Set)
   "explored" stores visited state tuples.
   d. List
   "expanded_nodes" stores expanded board states and "trace_data" stores detailed trace with cost information.
   e. Custom Class "AStarNode".
5. **PuzzleState Class**
   a. List (2D Array)
   "board" represents the puzzle configuration. Nested lists for row-column structure
   b. Tuple
   Used for hashing and equality comparisons. Enables use of states in Sets and Dict keys.

Algorithm Explanation

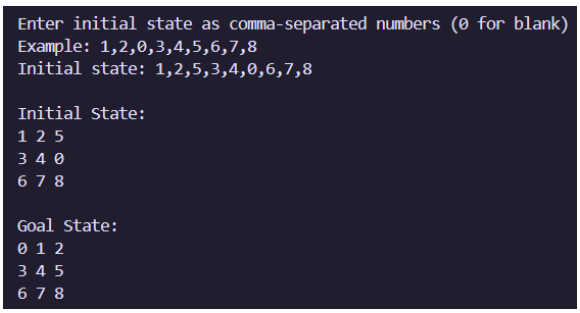
1. **Breadth-First Search (BFS)**
   - Strategy: Explores all neighbors at the current depth before moving deeper.
   - Space Complexity: Very high — stores all frontier nodes and explored states (O(b^d), where b = branching factor, d = solution depth).
   - Time Complexity: High O(b^d)— but guarantees shortest path.

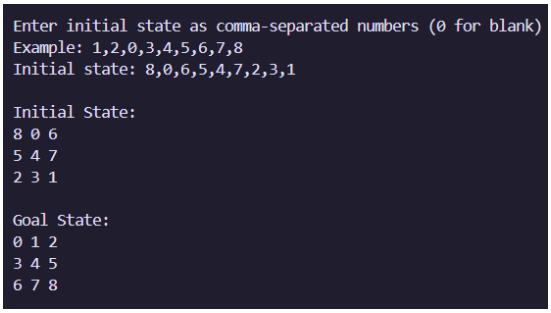
- Optimal? Yes — finds the solution with the fewest moves (for uniform cost).
- Complete? Yes — will find a solution if one exists.
- Use Case: Works well for shallow puzzles but memory-intensive when state space grows.
- Typical Output (8-puzzle): Returns the shortest solution, but might explore thousands of states.

2. **Depth-First Search (DFS)**
   - Strategy: Explores as far as possible along each branch before backtracking.
   - Space Complexity: Low — proportional to the maximum depth of the tree (O(bm)).
   - Time Complexity: High O(b^m) — can explore deep but irrelevant paths.
   - Optimal? No — DFS may find a long solution before a short one.
   - Complete? No — may get stuck in infinite loops without cycle checks.
   - Use Case: When we want to quickly reach a deep state or space is limited.
   - Typical Output (8-puzzle): Might find a (very) non-optimal path or none at all.
3. **Iterative Deepening DFS (IDDFS)**
   - Strategy: Combines the space efficiency of DFS with the optimality of BFS by repeatedly doing DFS with increasing depth limits.
   - Space Complexity: Low — only needs stack space (O(b*m)).
   - Time Complexity: Higher than DFS alone but much better than BFS in terms of memory.
   - Optimal? Yes — when uniform step cost (like in 8-puzzle).
   - Complete? Yes — same guarantees as BFS.
   - Use Case: When memory is tight and depth of solution is unknown.
   - Typical Output (8-puzzle): Finds the optimal solution depth (e.g., 31 moves) but expands many repeated nodes.
4. **A* Search (A-star)**
   - Strategy: Uses both actual cost so far (g(n)) and an admissible heuristic (h(n)) like Manhattan or Euclidean distance to guide search toward the goal.
   - Space Complexity: High — needs to store all nodes in memory similar to BFS.
   - Time Complexity: (O(b^d) much less with an admissible heurestic) Much more efficient than BFS if heuristic is good.
   - Optimal? Yes — if heuristic is admissible (never overestimates).
   - Complete? Yes — will find the optimal solution if the heuristic is admissible.

- Use Case: Best algorithm for 8-puzzle when optimality and efficiency are both important.
- Typical Heuristics for 8-puzzle:
    - Manhattan distance: Sum of distances of tiles from goal position → admissible
    - Euclidean distance: Geometric distance of tiles → admissible but less efficient
- Typical Output (8-puzzle): Quickly solves with far fewer node expansions compared to BFS or IDDFS.

Sample Runs





1. **BFS**

```
Running BFS...

States explored: 10

Solution found with 3 moves:
================================================

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

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

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

Move 3: Left
0 1 2
3 4 5
6 7 8
Execution Time : 0.015212059020996094
Max Depth : 3
```

```
Running BFS...

States explored: 181439

Solution found with 31 moves:
================================================
```

```
Move 31: Up
0 1 2
3 4 5
6 7 8
Execution Time : 16.73160481452942
Max Depth : 31
```

2. **DFS**

```
Move 59123: Up
0 1 2
3 4 5
6 7 8
Execution Time : 12.343486070632935
Max Depth : 66123
```

```
Move 18055: Up
0 1 2
3 4 5
6 7 8
Execution Time : 14.52535605430603
Max Depth : 66125
```

3. **IDFS**

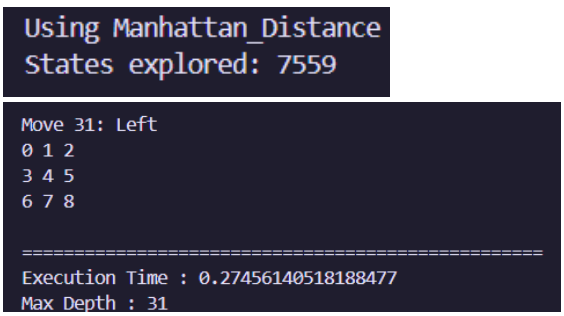
```
Running IDDFS...

States explored: 11

Solution found with 3 moves:
==============================================

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

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

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

Move 3: Left
0 1 2
3 4 5
6 7 8
Execution Time: 0.00995 seconds
Max Depth Reached: 3
```

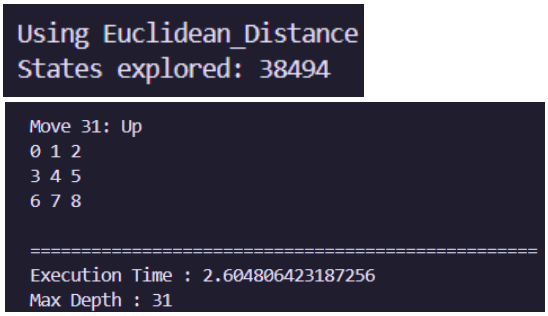
```
Move 31: Up
0 1 2
3 4 5
6 7 8
Execution Time: 65.28657 seconds
Max Depth Reached: 31
```

4. **A***

```
States explored: 4

Solution found with 3 moves:
==================================================

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

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

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

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

==================================================

Detailed Trace (Admissibility Check):
-----------------------------------------------------------------------------------------------
Step   Action       g(n)   h(n)     f(n)     True d*   Admissible   Frontier   Explored
-----------------------------------------------------------------------------------------------
1      dequeue      0      3.00     3.00     3         Yes          0          1
2      dequeue      1      2.00     3.00     2         Yes          2          2
3      dequeue      2      1.00     3.00     1         Yes          2          3
4      dequeue      3      0.00     3.00     0         Yes          3          4
Execution Time : 0.0016086101531982422
Max Depth : 3
```

```
Using Euclidean_Distance
States explored: 4

Solution found with 3 moves:
==================================================

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

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

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

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

==================================================

Detailed Trace (Admissibility Check):
-----------------------------------------------------------------------------------------------
Step   Action       g(n)   h(n)     f(n)     True d*   Admissible   Frontier   Explored
-----------------------------------------------------------------------------------------------
1      dequeue      0      3.00     3.00     3         Yes          0          1
2      dequeue      1      2.00     3.00     2         Yes          2          2
3      dequeue      2      1.00     3.00     1         Yes          2          3
4      dequeue      3      0.00     3.00     0         Yes          3          4
Execution Time : 0.0
Max Depth : 3
```





Notes on admissibility

Both are equally admissible. Both heuristics satisfy $h(n) \leq h^*(n)$ for all states. However, we can compare their informativeness:

- Manhattan Distance is more informed. It provides tighter bounds (closer to true cost) and leads to fewer node expansions in general.
- Euclidean Distance is less informative. It is more optimistic (lower estimates) and may explore more nodes before finding an optimal solution, but it still guarantees an optimal solution due to admissibility.

As test case gets more complex:

Aspect	**Manhattan**	**Euclidean**
Nodes Expanded	Fewer	More
Memory Usage	Lower	Higher
Optimality	Guaranteed	Guaranteed
Speed	Faster	Slower

Algorithm Operation

1. **Breadth-First Search (BFS)**
   - Start with the initial 8-puzzle state and add it to a queue (FIFO structure).
   - Repeatedly dequeue the front state and check:
   - If it matches the goal state → construct and return the solution path.
   - Keep track of explored states to avoid revisiting the same board configurations.
   - For each dequeued state, generate all of its valid neighboring states (possible configurations after one move).
   - Add each unseen neighbor to the end of the queue to be explored later.
   - Track statistics along the way such as the number of states expanded, depth reached, and total explored count.

- If the queue becomes empty and the goal hasn't been found, no solution exists.

2. **Depth-First Search (DFS)**
   - Start at the initial 8-puzzle state and push it onto a stack (LIFO structure).
   - Repeatedly pop the top state from the stack and check:
   - If it matches the goal state → return the solution path.
   - Keep a set of explored states to avoid revisiting previous configurations.
   - For each popped state, generate its neighboring states (possible moves) and push unexplored ones onto the stack.
   - The search continues, always diving deeper into one branch before exploring alternatives.
   - If the stack becomes empty and the goal hasn't been found, no solution exists.
3. **Iterative Deepening Seach (IDDFS)**
   - Combines BFS's optimality and DFS's space efficiency.
   - Repeats DFS with increasing depth limits until the goal is found.
   - High-Level Workflow:
     - Start with depth limit = 0.
     - Run Depth-Limited DFS from the initial state.
     - If goal is not found, increase depth limit by 1 and repeat.
     - Stop when goal is found and return results.
4. **A* Search**
   - Start at the initial 8-puzzle state and push it onto a min-heap (priority queue) with f(n) = g(n) + h(n).
   - Repeatedly pop the state with the LOWEST f-cost from the heap and check:
   - If it matches the goal state → return the solution path.
   - Keep a set of explored states to avoid revisiting previous configurations.
   - For each popped state, generate its neighboring states (possible moves):
     - Calculate g(n) = current path cost + 1 (one move).
     - Calculate h(n) = heuristic estimate to goal (Manhattan or Euclidean distance).
     - Calculate f(n) = g(n) + h(n) (total estimated cost).
     - If the neighbor is not in the frontier → add it to the heap.
     - If the neighbor is already in the frontier with a higher g-cost → update it with the better path.