An intelligent N-Puzzle Solver

Farah Hossam El-Din **20225195**

Ziad Yasser Adel **20225165**

Introduction and overview :

The N-Puzzle is a classic sliding puzzle that challenges the player to arrange a grid of numbered tiles in the correct order by sliding them into an empty space. This project focuses on developing an intelligent N-Puzzle solver that can efficiently solve puzzles of sizes 8, 15, and 24 using a Best-First Search algorithm. The solver incorporates at least four heuristic functions to guide the search process and improve the efficiency of finding the solution.

Application :

Similar applications exist in various forms, including desktop, web, and mobile versions of sliding puzzles. These applications generally feature a simple user interface that allows the user to shuffle the tiles and attempt to solve the puzzle manually. Some advanced applications provide hints or automated solving options, which employ heuristic-based algorithms to demonstrate or achieve optimal solutions.

Proposed Solution :

Main Functionalities/Features

The proposed N-Puzzle solver includes the following features:

- **Puzzle Initialization: Users can select the puzzle size (8, 15, or 24).**
- **Shuffle: The solver can shuffle the puzzle tiles into a random, solvable configuration.**
- **Solve: Using the Best-First Search algorithm, the solver attempts to solve the puzzle, displaying the steps taken.**
- **Heuristics Options: Users can choose from multiple heuristic functions (e.g., Misplaced Tiles, Manhattan Distance).**
- **Visualization: A visual representation of the puzzle solving process, showing the movement of tiles and the progression towards the solution.**

Applied Algorithm :

Best-First Search Algorithm

The Best-First Search algorithm is a search strategy where the most promising node (as determined by a given heuristic) is expanded first. In the context of the N-Puzzle, the algorithm evaluates the possible moves and selects the one that appears to lead closest to the goal state.

Function Workflow

Initialization:

priority_Queue: A priority queue is initialized to keep track of the states that need to be explored. This queue ensures that states are expanded based on their priority, determined by the heuristic function.

Visited: A set is initialized to keep track of states that have already been visited, preventing the algorithm from revisiting the same state and entering an infinite loop.

Starting the Search:

The initial state of the puzzle (self.tiles_grid) is pushed onto the priority queue. The priority is determined by the heuristic function (func), which evaluates how close the current state is to the goal state.

Main Loop:

The algorithm enters a loop that continues until the priority queue is empty. In each iteration:

The state with the highest priority (lowest heuristic value) is popped from the queue.

The current state is checked against the goal state (self.tiles_grid_completed). If they match, the puzzle is solved, and the function returns the solution.

The current state is marked as visited to avoid revisiting it.

The algorithm then explores all possible neighboring states (states that can be reached by a single tile move). If a neighbor has not been visited, it is pushed onto the priority queue with its corresponding heuristic value.

Completion:

If the algorithm finds a solution, it prints "Completed" and returns the final state.

If the priority queue is exhausted without finding a solution, it prints "No Solution" and returns None.

Heuristic Function

The func parameter is a heuristic function that estimates the cost of reaching the goal state from the current state. This function plays a crucial role in guiding the Best-First Search algorithm by assigning priorities to different states. The heuristic function's design significantly impacts the efficiency and effectiveness of the search process.

Heuristic Functions :

Misplaced Tiles: Counts the number of tiles not in their goal position.

Manhattan Distance: Calculates the total distance each tile is from its goal position.

Linear Conflict: An enhancement of Manhattan Distance that adds penalties for tiles that are in the same row or column as their goal position but are blocked by other tiles.

Misplaced Tiles With penalty of row and column : an enhance of misplaced tiles This is the basic count of how many tiles are not in their correct positions.

1st Function - Misplaced Tiles Heuristic:

Description:

The Misplaced Tiles heuristic counts the number of tiles that are not in their goal position. This is a simple and effective way to estimate how far the current state is from the goal.

Formula:

$h(n)$**=Number of misplaced tiles**

Explanation:

Correct Board Setup:

The function first creates the correct goal configuration for the board. For a 3x3 puzzle, the goal is to have tiles numbered 1 through 8 arranged in order with the last position being the empty space (0).

Counting Misplaced Tiles:

It iterates through the current state of the board and compares each tile's position with its correct position in the goal configuration.

If a tile is not in its correct position, the count number_of_places is incremented.

Final Heuristic Value:

The total number of misplaced tiles is returned as the heuristic value. This value represents the estimated number of moves needed to solve the puzzle, assuming each move places one tile in its correct position.

2nd Function - Heuristic Calculation (Misplaced Tiles With penalty)

Number of Misplaced Tiles: This is the basic count of how many tiles are not in their correct positions.

Penalty for Row/Column Misplaced:

Row Penalty: **A penalty is added if a tile is in the correct row but in the wrong column.**

Column Penalty: **A penalty is added if a tile is in the correct column but in the wrong row.**

Formula : **h(n)= Number of misplaced tiles + Penalty for row/column misplaced**

Explanation :

Purpose: **This function not only counts the number of misplaced tiles but also adds additional penalties for tiles that are in the correct row or column but not in their exact position (an enhanced heuristic).**

Operation:

It still initializes the `correct board` (goal state).

It counts the number of misplaced tiles just like the original function.

For each misplaced tile, it checks:

If the tile is in the correct row but the wrong column (and adds a penalty).

If the tile is in the correct column but the wrong row (and adds a penalty).

It then returns the sum of misplaced tiles plus the penalties.

Output: The sum of the number of misplaced tiles and the penalties for row/column misplaced tiles.

3rd Function - Manhattan Distance:

Description:

The Manhattan Distance heuristic calculates the sum of the distances of all tiles from their goal positions, where the distance is the number of moves required to move a tile to its goal position in a straight line (horizontally or vertically).

Formula:

$$h(n) = \sum_{i=1}^{N} |x_i - x_i^*| + |y_i - y_i^*|$$

Explanation:

Correct Board Setup:

The goal configuration is set up as in the previous functions.

Manhattan Distance Calculation:

For each tile on the board, the function finds its correct position in the goal configuration.

It then calculates the Manhattan distance for each tile, which is the sum of the horizontal and vertical distances between the current and goal positions.

This distance is added to total_distance.

Final Heuristic Value:

The total Manhattan distance for all tiles is returned as the heuristic value. This value represents the estimated number of moves needed to solve the puzzle, assuming each move can shift a tile directly towards its goal.

4th Function - Linear Conflict :

Description:

A **linear conflict** occurs when two tiles are in their goal row or column but are in the wrong relative order. Specifically:

- In a **row** conflict, if two tiles are in the correct row, but one tile that should be to the left of the other is actually to its right, a conflict exists.
- Similarly, in a **column** conflict, if two tiles are in the correct column, but one tile that should be above the other is actually below it, a conflict exists.

Formula: **h(n) = Manhattan Distance + 2 × (Number of linear conflicts)**

Definition: Two tiles tj and tk are in a linear conflict if tj and tk are in the same line, the goal positions of tj and tk are both in that line, tj is to the right of tk and the goal position of tj is to the left of the goal position of t

Explanation:

1. **Manhattan Distance Calculation**:
    - The function first computes the Manhattan distance for each tile, which is the sum of the absolute differences between the current position of a tile and its goal position.

2. **Linear Conflict Calculation**:

- The function then checks for Linear Conflicts:
  - **Row Conflict**: For each tile in a row, the function checks if there is another tile further along in the same row that should be earlier in the row according to the goal configuration.
  - **Column Conflict**: Similarly, for each tile in a column, the function checks if there is another tile further down in the same column that should be earlier in the column according to the goal configuration.
- If such a conflict is found, it adds 1 to `linear_conflict`.
- The total linear conflict cost is added to the Manhattan distance, with each conflict contributing an additional cost of 2 (since resolving the conflict requires at least two moves).

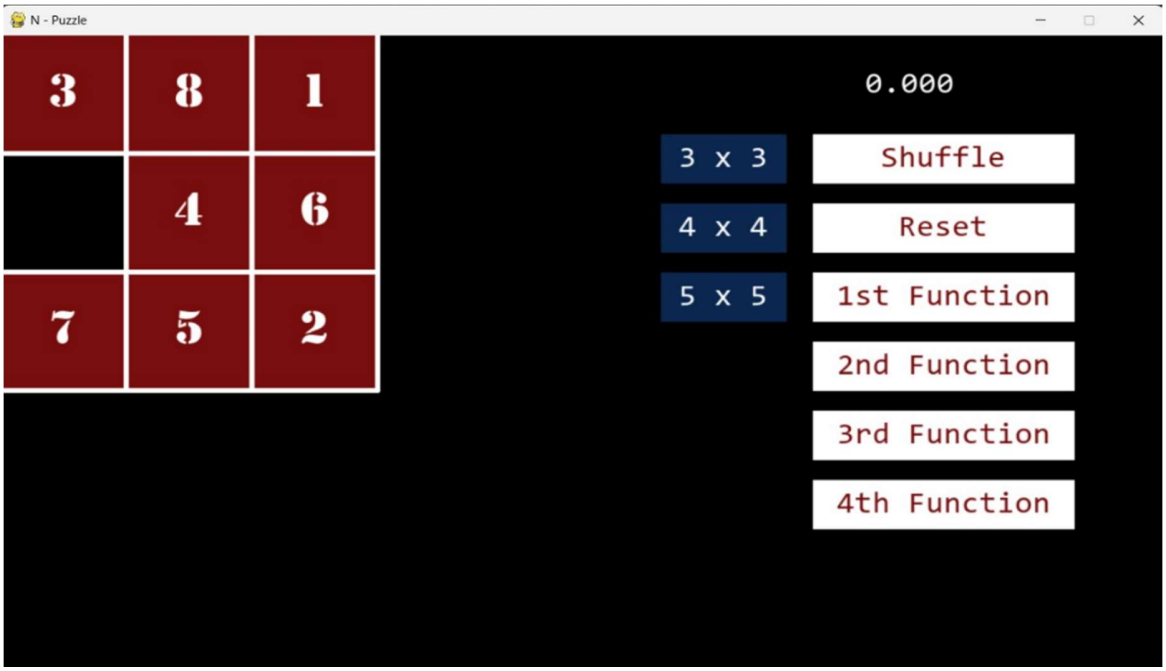
This combined heuristic will generally provide a more accurate estimate of the remaining cost to solve the puzzle, improving the efficiency of the search algorithm

---

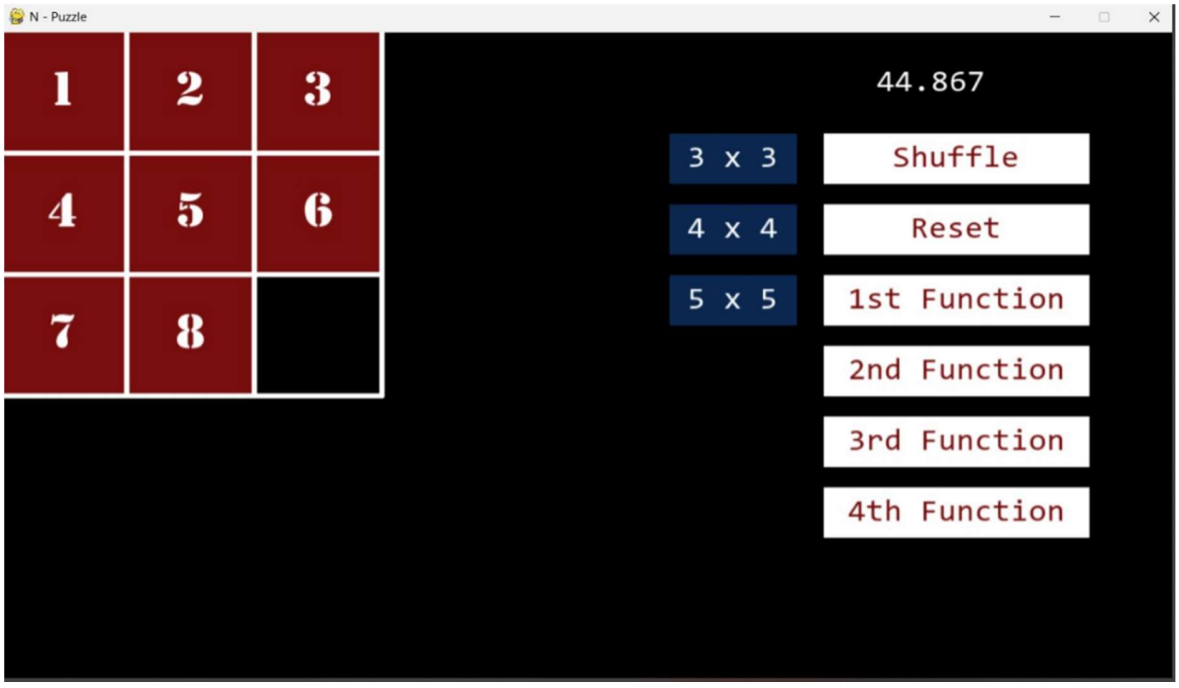
Experiments & Results :

To evaluate the performance of the solver, various experiments will be conducted using different puzzle sizes and initial configurations. The solver's performance will be measured based on the number of moves taken, the time required to find the solution, and the computational resources used.

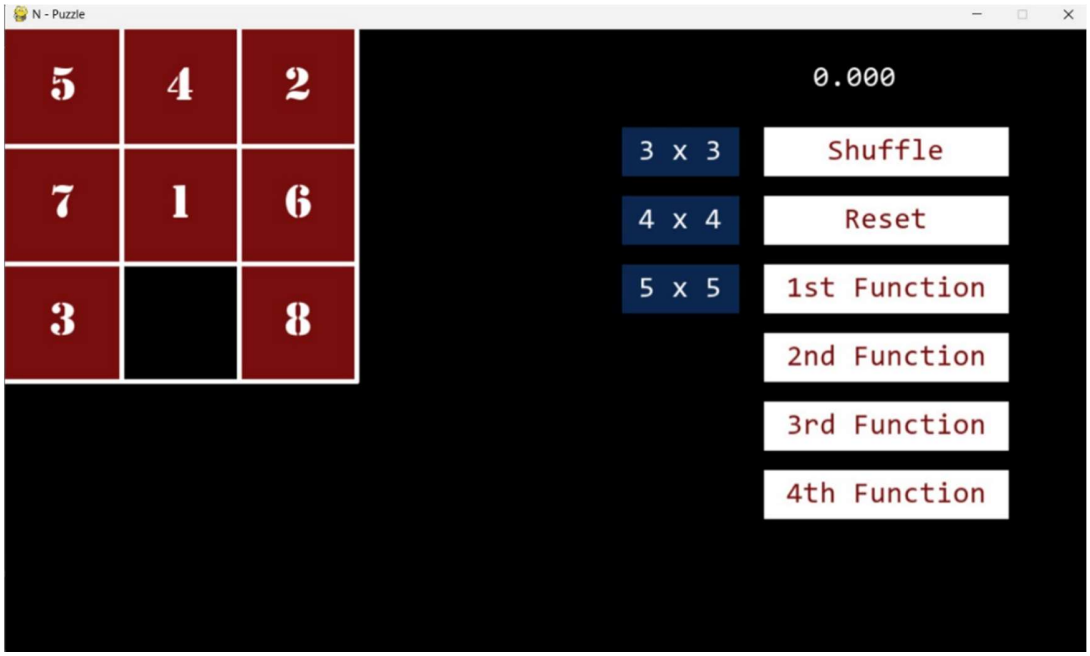
1st Function Tests (misplaced tiles):



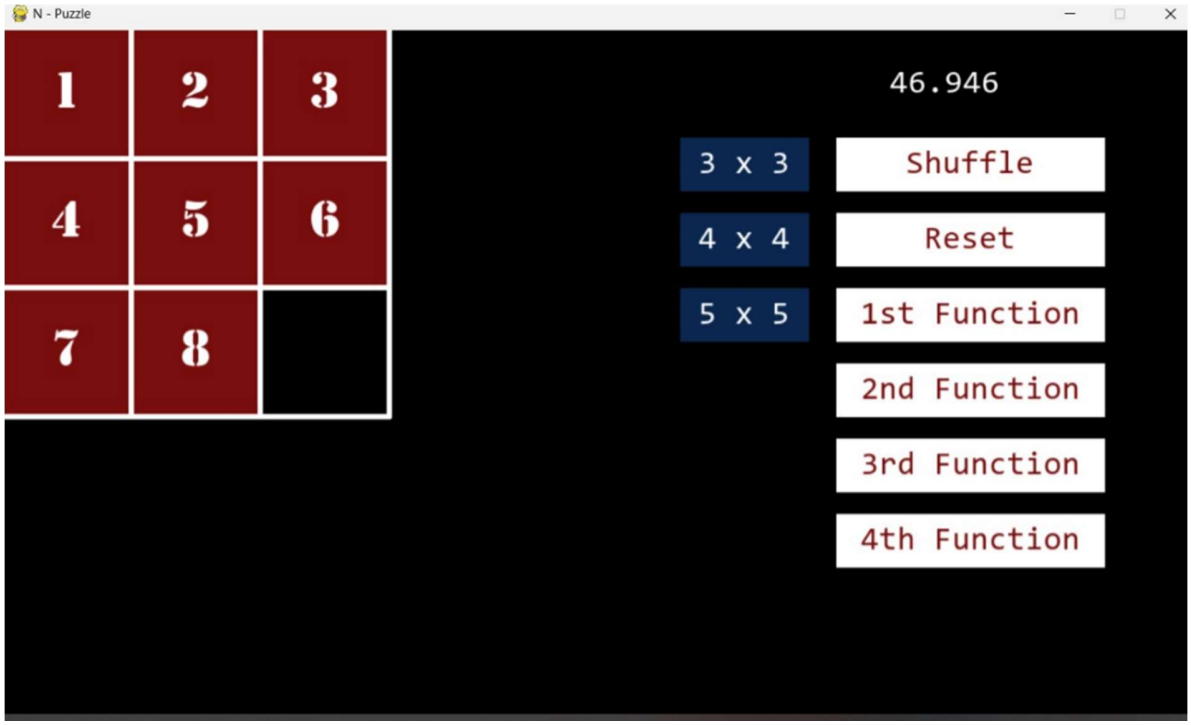
44.867 sec to solve



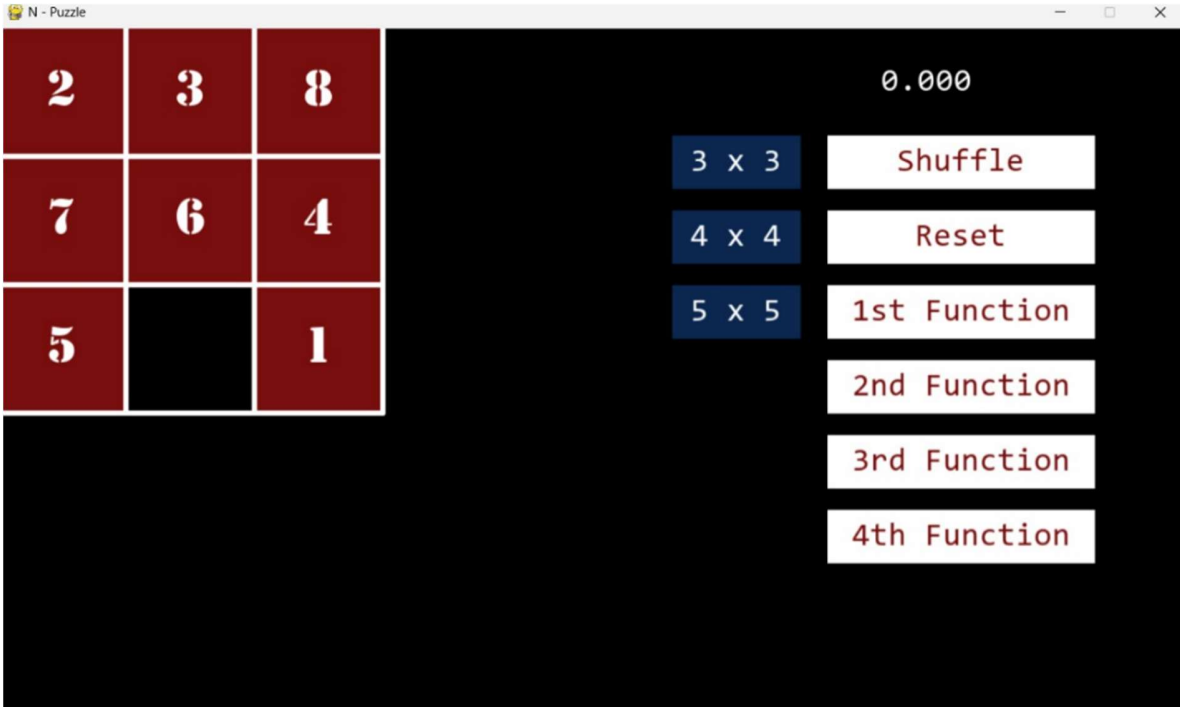
2nd Function Test (Misplaced Tiles with penalty Row and Column) :



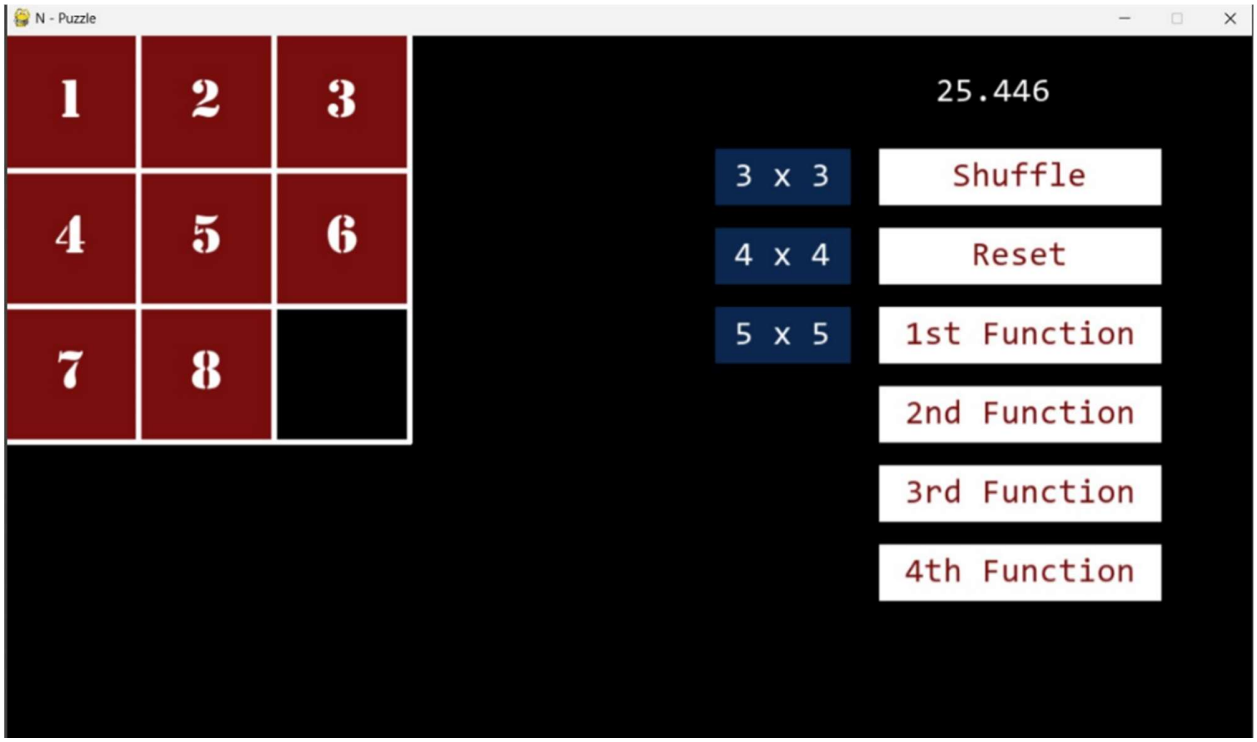
46.946 Sec To Solve



3rd Function Test (Manhattan Distance) :

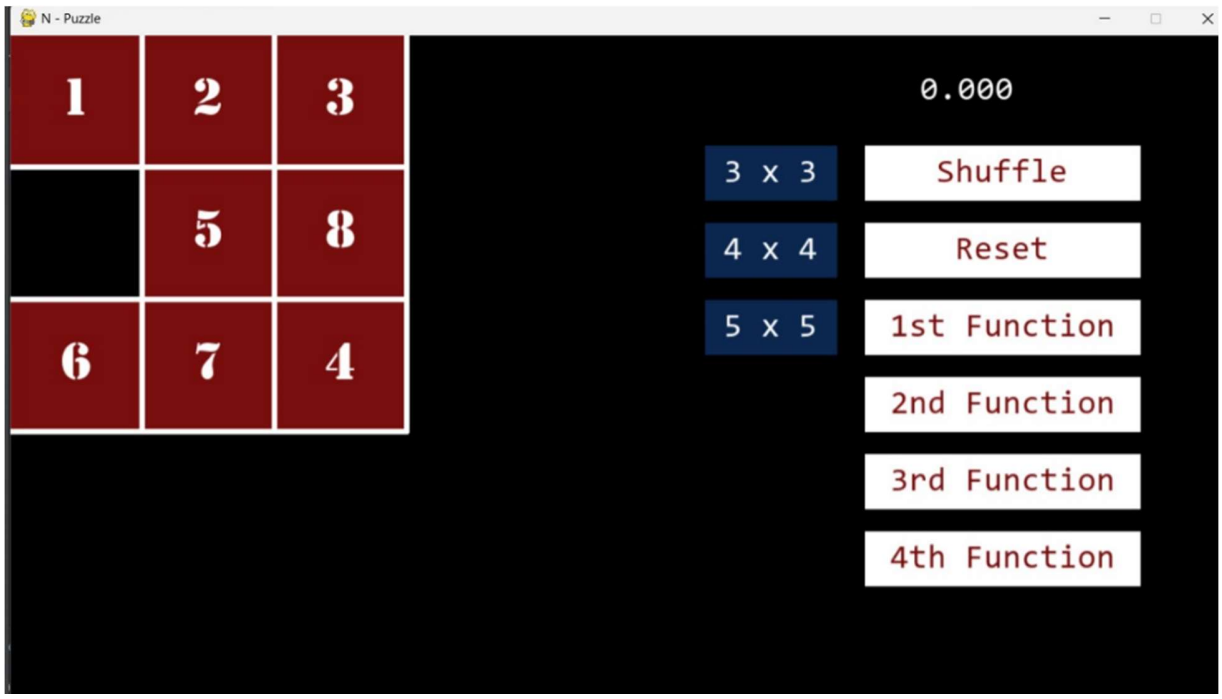


25.446 Sec To Solve

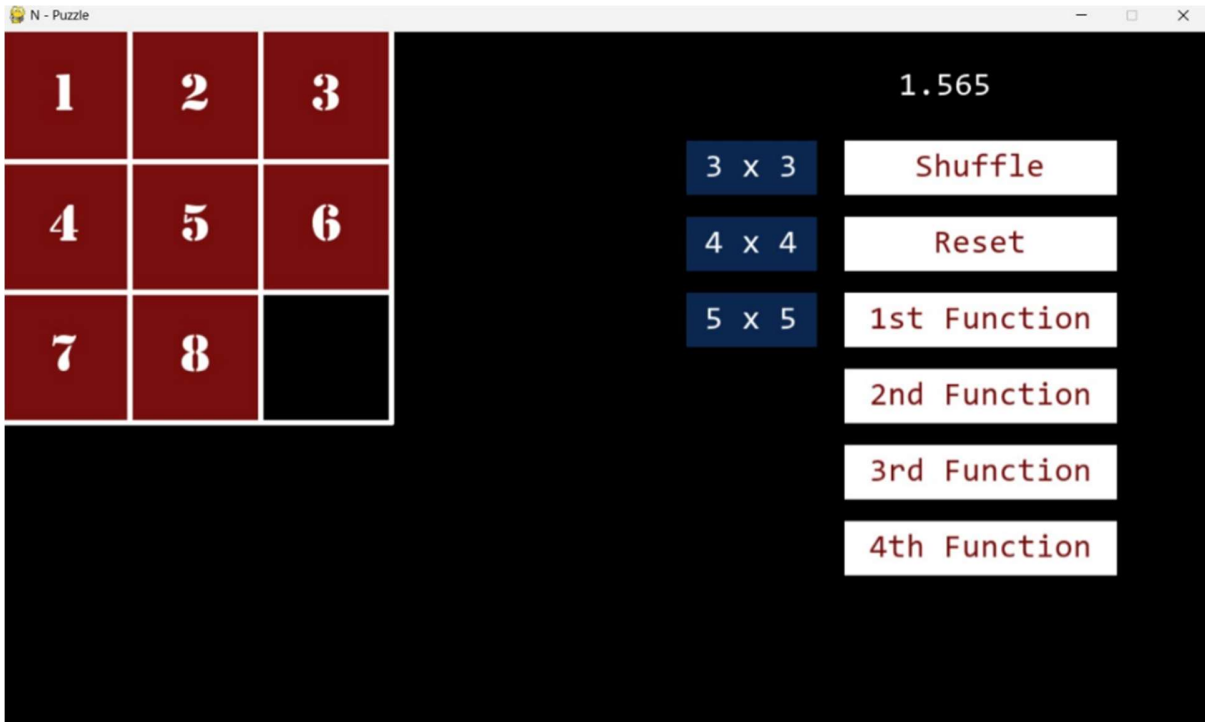


4th Function Test (Linear Conflict):

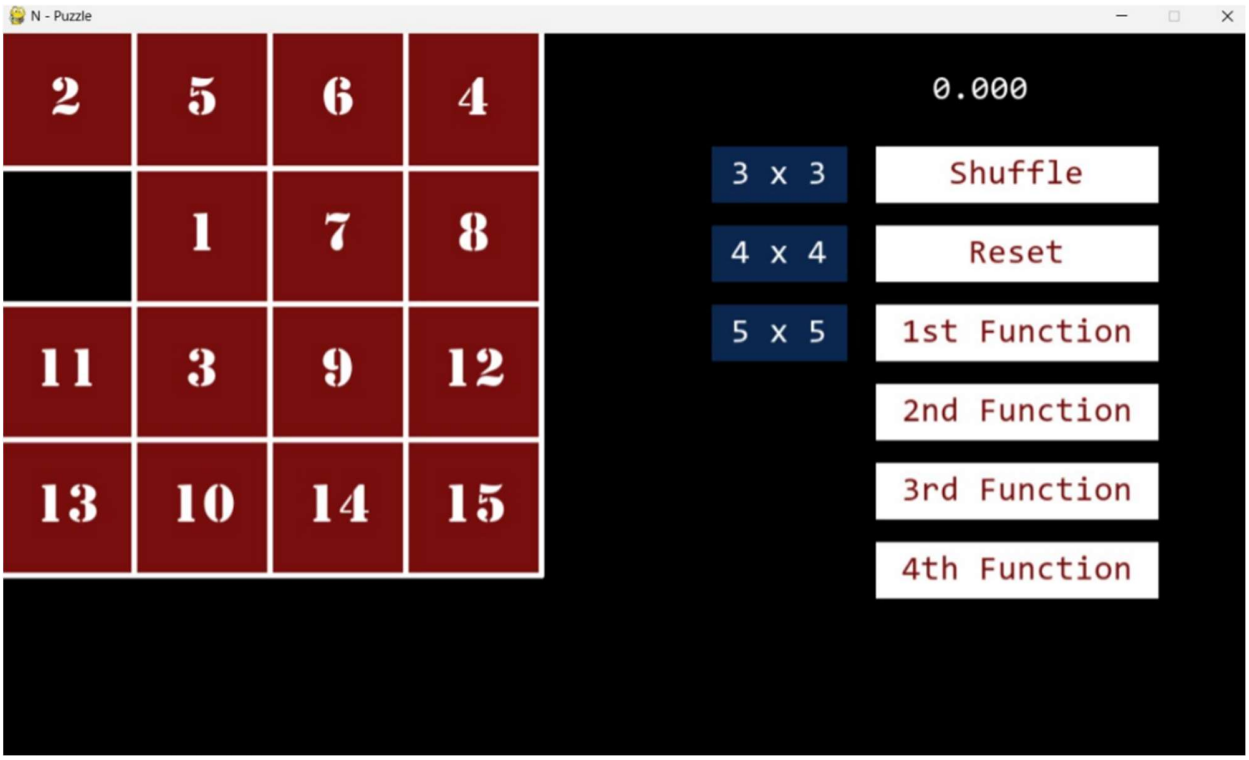
First Test (3x3) :



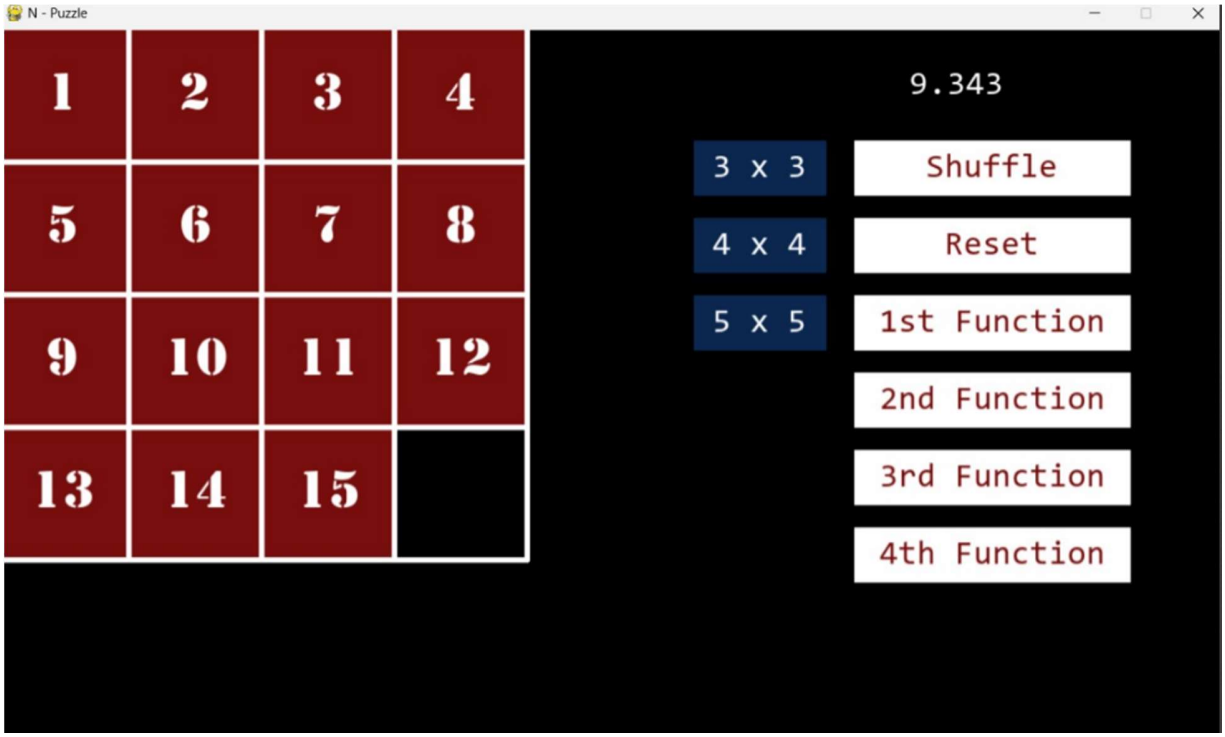
Solved in 1.56 s



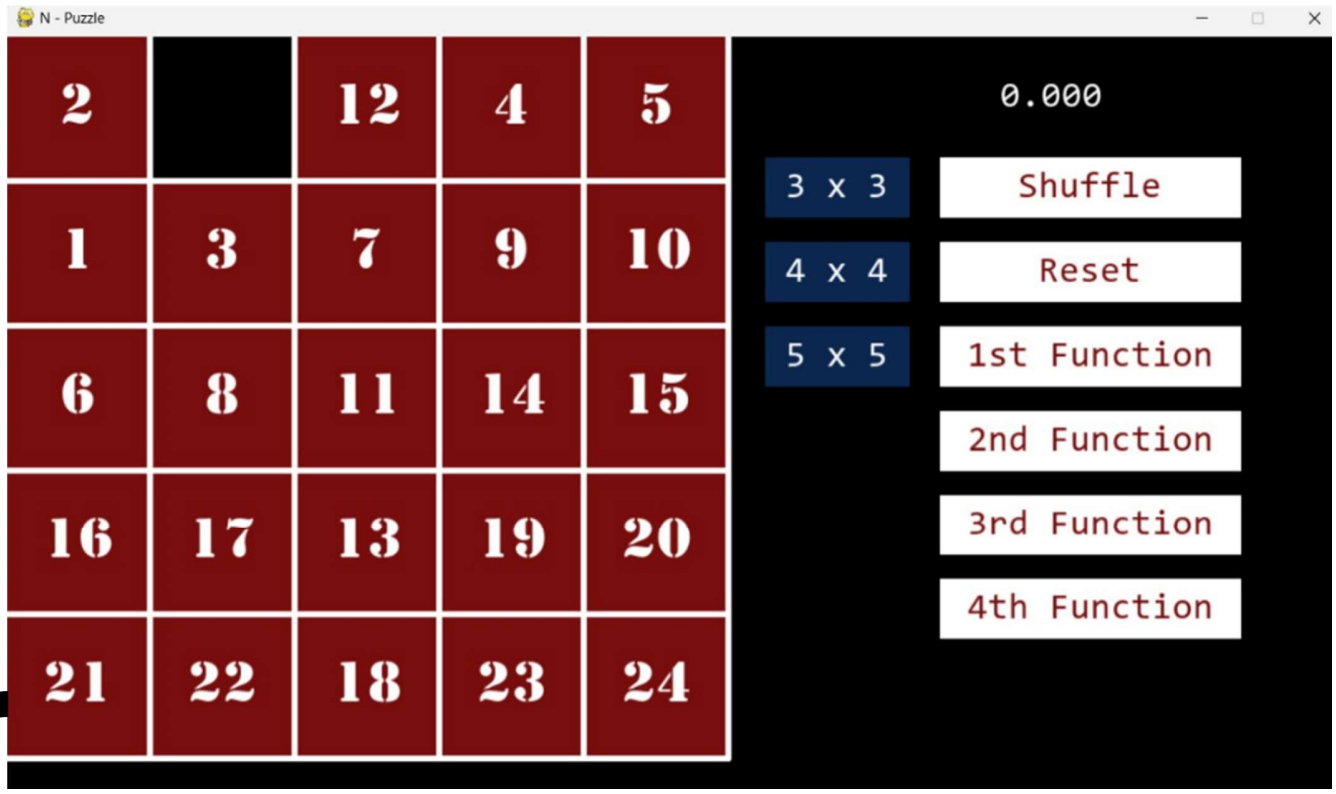
Second Test (4x4) :



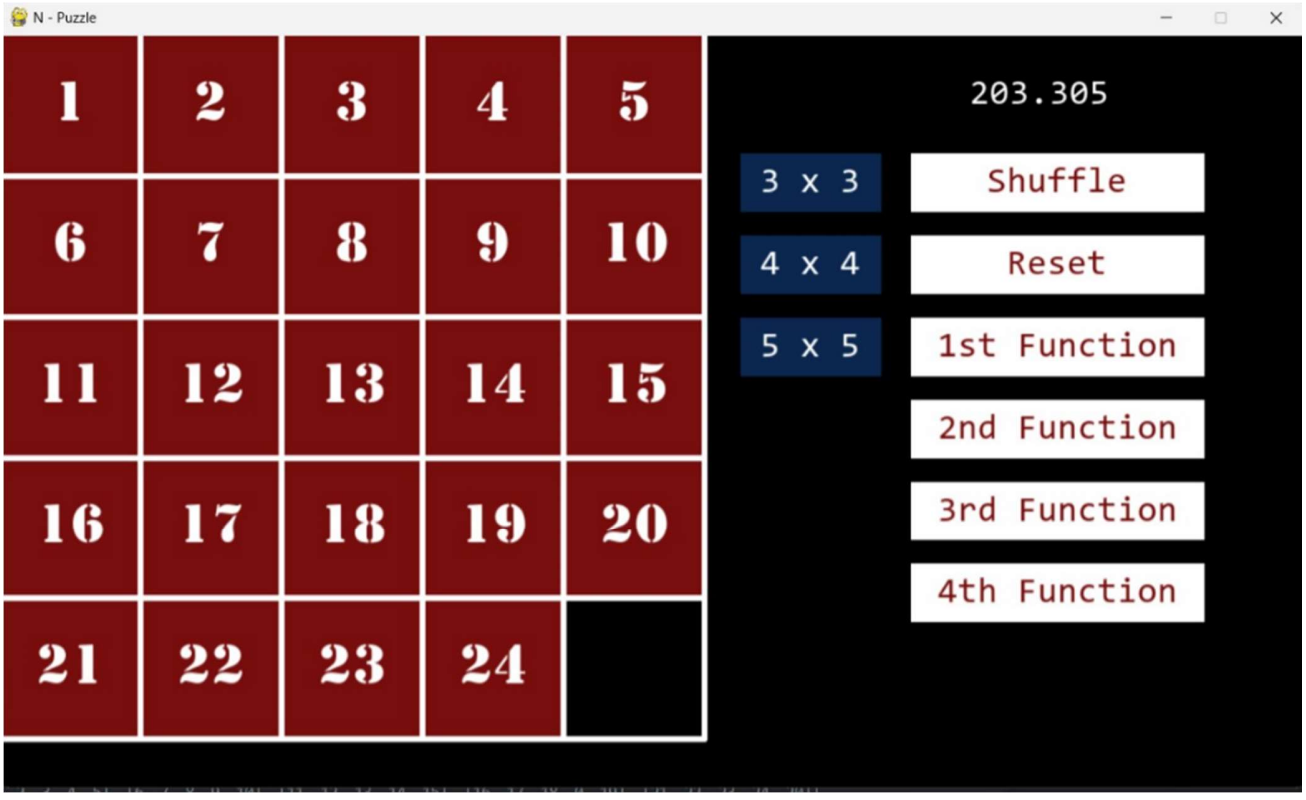
Solved in 9.3 s



Third Test (5x5) :



Solved in 203.3 sec



Future Work

Future modifications could include:

Optimization Techniques: Implementing additional optimizations to reduce the search space further.

Alternative Algorithms: Exploring other search algorithms like A* or IDA* to compare performance.

Dynamic Heuristic Adjustment: Investigating the potential of dynamically adjusting the heuristic function based on the puzzle's current state

---

THANKS