**8-Puzzle Problem Solver**

The Python implementation of the 8-Puzzle Problem Solver using the A\* search algorithm was provided with my answers. One tile is always missing, represented by an underscore (\_). I have also provided details about the code through comments, but this documentation is also useful for my code readers.

The program allows the user to input the initial and goal states of the board and then uses the A\* algorithm with two different heuristics - Manhattan distance and Euclidean distance - to find the optimal solution to the problem. The user can choose between these two heuristics by inputting 0 or 1, respectively.

**Classes**

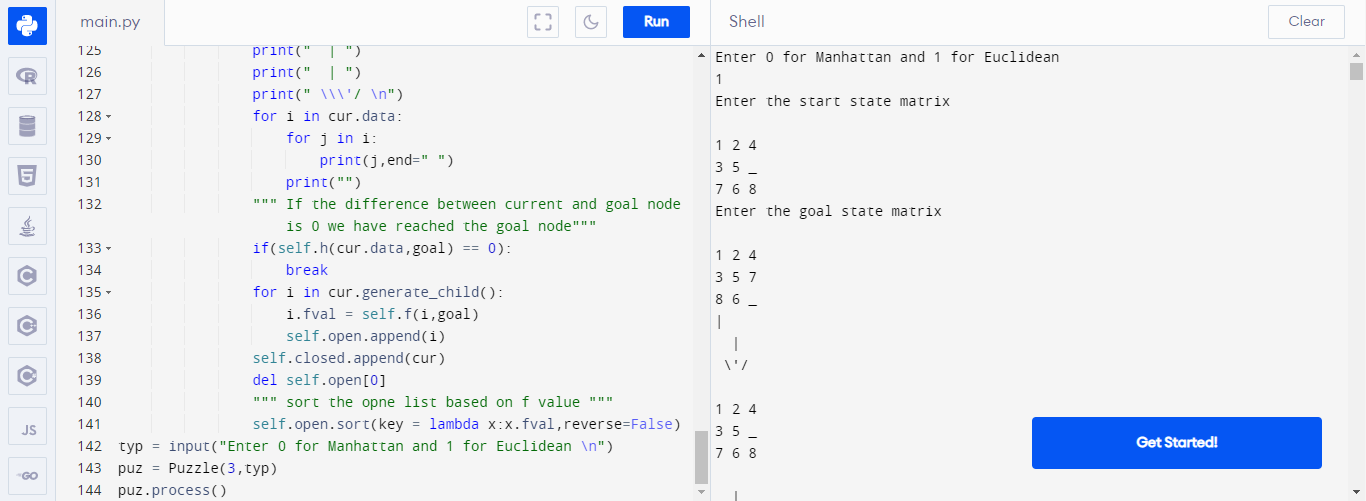
* Node: A class representing each node in the search tree. Each node contains information about the current state of the board, its level in the search tree, and its calculated f-value.
* Puzzle: A class representing the overall puzzle problem. It contains the size of the board, the open and closed lists for the search algorithm, and the heuristics used to calculate f-values.

**Methods**

* generate\_child: Generates child nodes from the given node by moving the empty space either in the four directions {up,down,left,right}.
* shuffle: Moves the empty space in the given direction and returns None if the position values are out of limits.
* copy: A copy function to create a similar matrix of the given node.
* find: Specifically used to find the position of the empty space.
* f: A heuristic function to calculate the f-value of a node, which is equal to the heuristic value plus the level of the node.
* h: A function to calculate the different between the given puzzles.
* euclidean: A function to calculate the Euclidean distance heuristic between the start and goal nodes.
* manhattan: A function to calculate the Manhattan distance heuristic between the start and goal nodes.
* process: Accepts the start and goal states of the board and begins the search algorithm using the chosen heuristic.

**Usage**

To use the program, we have to simply run the code and follow the prompts to input the initial and goal states of the board, as well as the chosen heuristic (0 for Manhattan distance, 1 for Euclidean distance). The program will then print out the optimal solution to the problem, including the number of moves required to reach the goal state and the path taken to get there.



IDA\* is easily created by designing a threshold for the scenario, which I also did. The threshold is a parameter in the IDA\* algorithm that controls the search depth. The threshold value is set to 20 in this implementation. The algorithm will continue to look for a solution by gradually increasing the threshold until it finds one or determines that no solution exists within the given threshold limit.

