Artificial intelligence : Optional project (Water Sort)

About me

- Full name → Pooria Rahimi
- Student number → 99521289

For this project, we are dealing with three files: `game.py`, `main.py`, and `ai_solution.py`. As mentioned, there is no need to modify the existing code in `main.py` and `game.py`, meaning we are not allowed to change these codes. We only need to define and implement the functions in the `ai_solution.py` file.

▼ Part 1 :

Implementation of the functions in the `ai_solution.py` file :

The `GameSolution` class is designed to solve the Water Sort game using different methods. Below, I'll explain what each method does and the overall functioning of the class.

Class and Methods

`__init__(self, game)`

- **Description**: This constructor initializes the `GameSolution` instance. It takes a game instance and sets various attributes.
- **Parameters**:
  - `game`: An instance of the Water Sort game.

`solve(self, current_state)`

- **Description**: This method attempts to find a solution to the Water Sort game from the current state.
- **Parameters**:
  - `current_state`: A list of lists representing the colors in each tube.
- **Overall Function**: This method iteratively explores different configurations using Depth-First Search (DFS) to find a solution.

`DFS(self, tubes, depth, limit, stack)`

- **Description**: This method performs a Depth-First Search (DFS) to find the solution by traversing the search tree.
- **Parameters**:
  - `tubes`: The current state of the tubes.
  - `depth`: The current depth in the search.
  - `limit`: The maximum depth to search.
  - `stack`: A list of current moves.
- **Overall Function**: This method starts from the current state and explores all possible states to find a solution.

`optimal_solve(self, current_state)`

- **Description**: This method attempts to find an optimal solution (minimum number of moves) to the Water Sort game from the current state.
- **Parameters**:
  - `current_state`: A list of lists representing the colors in each tube.
- **Overall Function**: This method uses a priority search algorithm with a priority queue (heap) to find the optimal solution.

`Hash(state)`

- **Description**: This method generates a hash of the current state of the tubes to facilitate comparison and avoid revisiting the same states.
- **Parameters**:

    - `state`: A list of lists representing the current state of the tubes.
- **Output**: A hash value.

`IsSolved(state)`

- **Description**: This method checks whether the game is solved.
- **Parameters**:
    - `state`: A list of lists representing the current state of the tubes.
- **Output**: A boolean value (True or False).

`Neighbors(state)`

- **Description**: This method finds all possible neighbors (valid moves) from the current state.
- **Parameters**:
    - `state`: A list of lists representing the current state of the tubes.
- **Output**: A list of pairs (new state, performed move).

`DFSNeighbors(state)`

- **Description**: This method finds all possible moves from the current state. Unlike `Neighbors`, this method does not create new states but only returns the possible moves.
- **Parameters**:
    - `state`: A list of lists representing the current state of the tubes.
- **Output**: A list of possible moves.

`Next(self, state, neigh)`

- **Description**: This method applies a move to the current state.
- **Parameters**:
    - `state`: A list of lists representing the current state of the tubes.
    - `neigh`: A pair (source tube, destination tube) representing the move performed.

- **Output**: The number of moves performed.

`Prev(self, state, neigh, tmp)`

- **Description**: This method reverses a move to revert to the previous state.
- **Parameters**:
  - `state`: A list of lists representing the current state of the tubes.
  - `neigh`: A pair (source tube, destination tube) representing the move performed.
  - `tmp`: The number of moves to be reversed.

`res(self, state)`

- **Description**: This method generates an evaluation value for the current state to help determine priority in the search.
- **Parameters**:
  - `state`: A list of lists representing the current state of the tubes.
- **Output**: A numerical evaluation value for the current state.

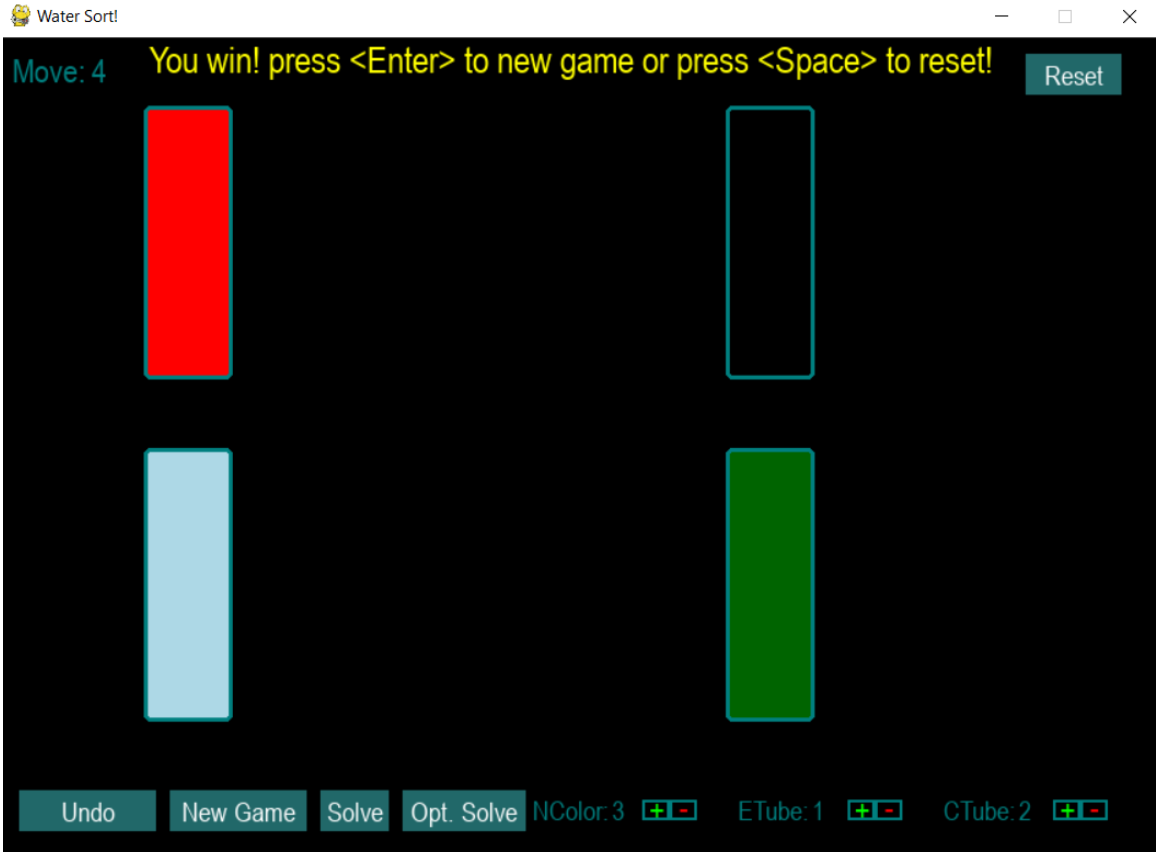
Overall Class Functioning

1. **Initialization**: The game instance is initialized with the constructor.
2. **Searching for Solutions**: Using the `solve` or `optimal_solve` methods, the class employs either DFS or priority search algorithms to find a solution.
3. **Checking Solution**: The `IsSolved` method is used at each step to check if the game is solved.
4. **Generating Possible Moves**: The `Neighbors` or `DFSNeighbors` methods find all possible moves from the current state.
5. **Performing and Reversing Moves**: The `Next` and `Prev` methods are used to perform and reverse moves.
6. **Storing Moves**: Finally, if the game is solved, the sequence of moves is stored in `self.moves`.

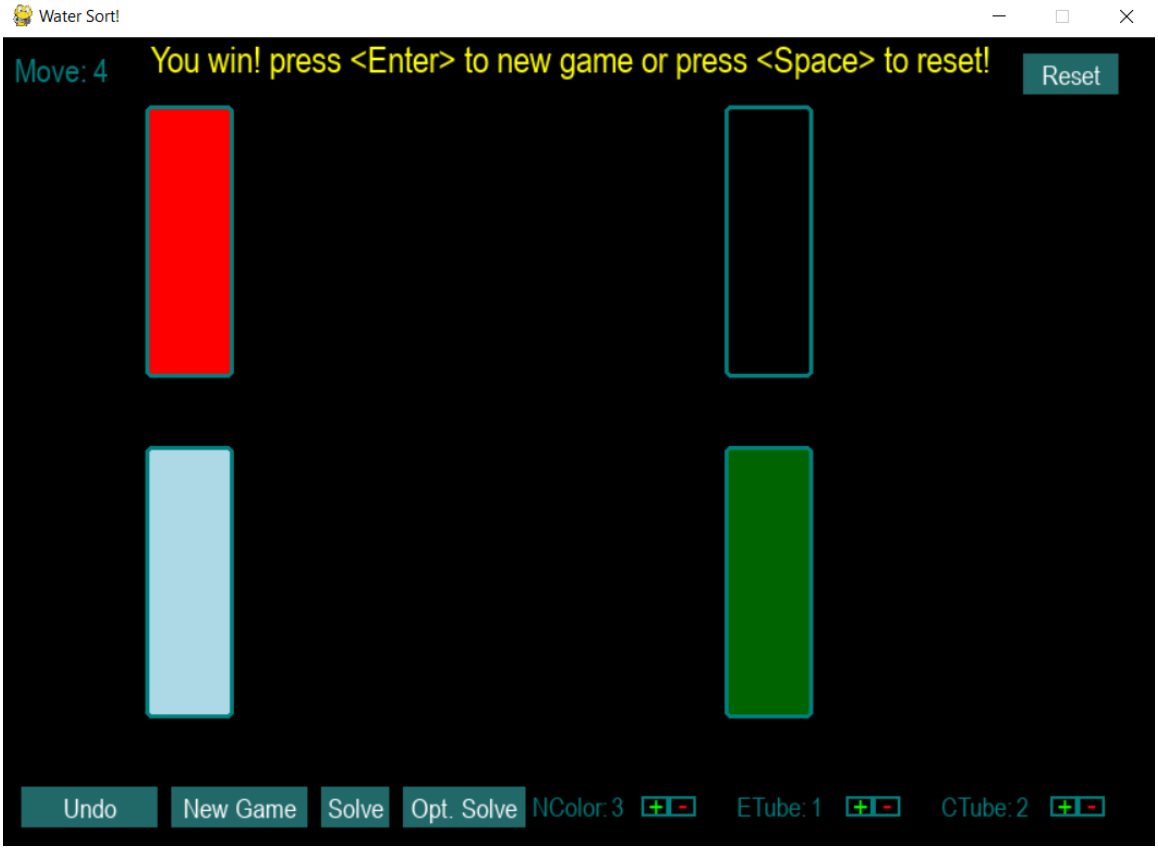
And finally, we will display screenshots of the game output here :

Start game :

Solve function :

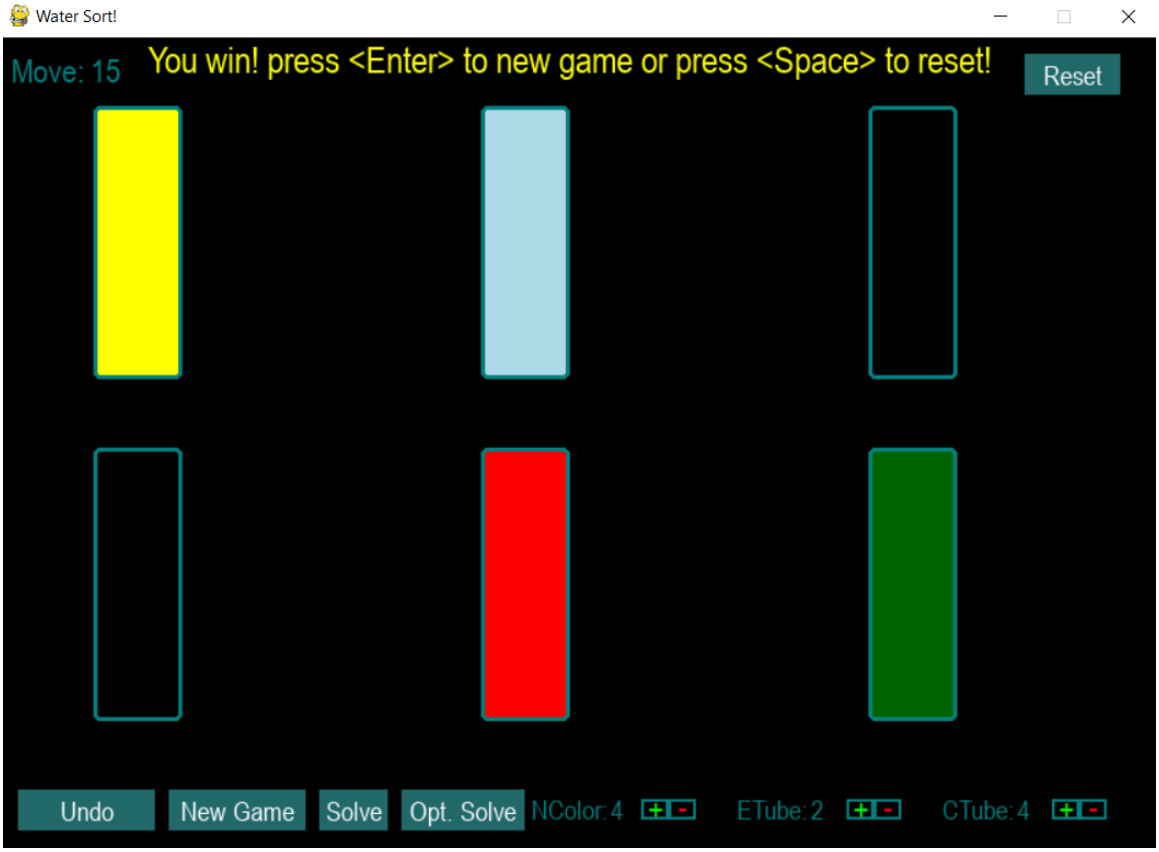


Optimal Solve function :

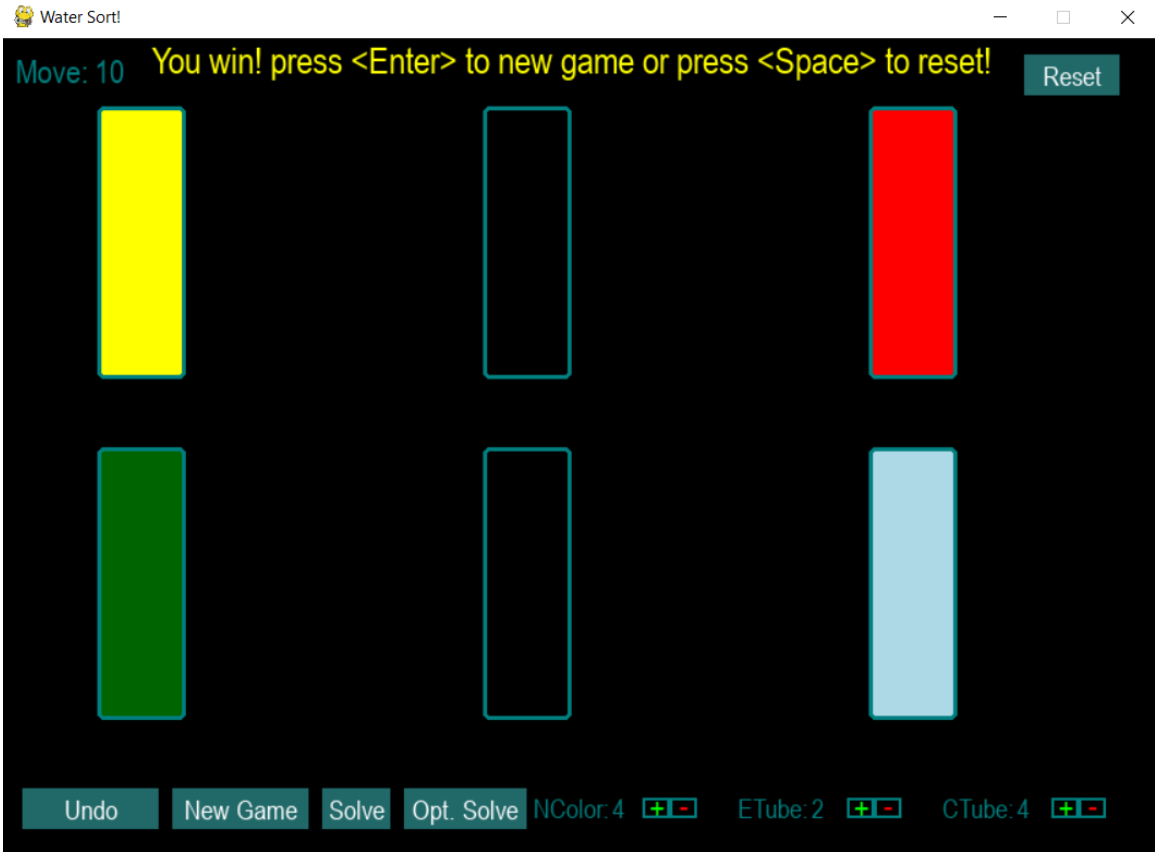


New game by new parameter :

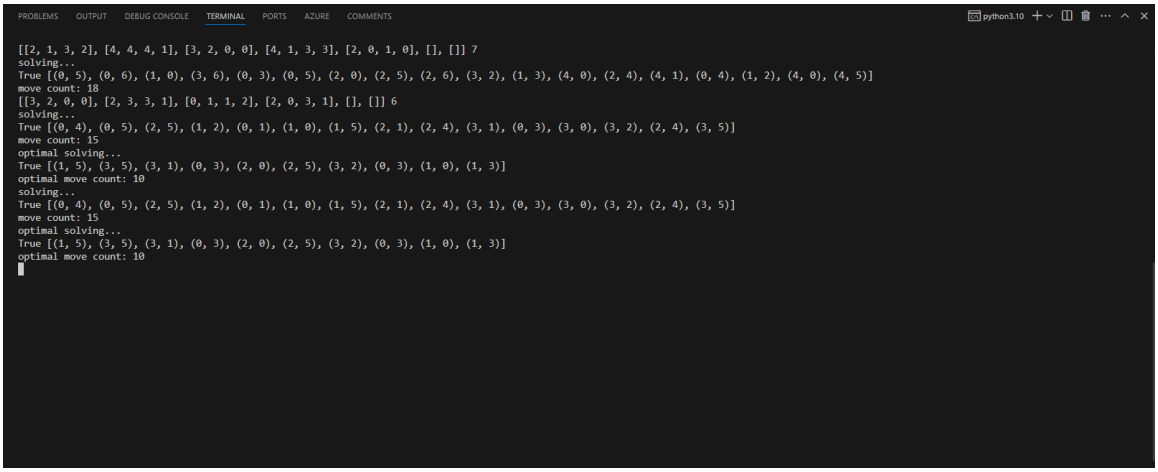
Solve function :



Optimal Solve function :



Cmd Output :



The End😔