8-Puzzle Solver with A* Search - Step-by-Step Process

1. **Import Required Libraries**

   - Import essential libraries such as NumPy, matplotlib, IPython widgets, and others.

2. **Define Puzzle State Class**

   - Represent each puzzle configuration with a class storing the state, parent, cost, action, and empty tile position.

3. **Heuristic Function**

   - Define the Manhattan Distance function to evaluate how far each tile is from its goal position.

4. **Generate Possible Actions**

   - Determine valid moves for the empty tile (up, down, left, right) based on its current position.

5. **State Transition Function**

   - Generate a new puzzle state by applying a valid move on the current state.

6. **Solvability Check**

   - Implement an inversion count method to determine whether a puzzle is solvable.

7. **A* Search Algorithm**

   - Use a priority queue to explore states based on cost + heuristic (f = g + h). Store explored states to avoid revisiting.

8. **Solution Path Reconstruction**

- Once the goal state is reached, trace back using parent references to reconstruct the sequence of moves.

9. **Create Random Solvable State**
   - Generate random puzzles and validate their solvability using the inversion count logic.

10. **Visualization Functions**
    - Create functions to display puzzle states using matplotlib.
    - Animate the solution path for better visual understanding.
    - Show puzzle steps in a grid layout.

11. **User Interface Setup with ipywidgets**
    - Add buttons for solving the puzzle, generating new puzzles, and custom user input.
    - Use output widgets to dynamically update visuals and messages.

12. **Button Click Handlers**
    - Define handlers for each button to trigger puzzle regeneration, custom input parsing, and solving.

13. **Display the Full UI Layout**
    - Arrange puzzle states, control buttons, and visual output areas using VBox and HBox from ipywidgets.

14. **Run the Application**
    - Display the final interactive UI with the puzzle solving functionality.