Project Report: 8-Puzzle Problem Solver

1. Cover Page

Field	**Details**
Name	Unnati Khandelwal
Registration no.	25BCY10031
Project Title	8-Puzzle Problem Solver
Course/Module	Fundamental of AI and ML
Date of submission	November 25, 2025
Submitted to	Vityarthi

2. Introduction

This report details the design, implementation, and testing of **The 8-Puzzle Game Engine**. This initial phase focuses on establishing the core game logic in a Python console environment, including board management, manual tile movement, win condition checking, and the implementation of a mathematically crucial **solvability check**. The 8-Puzzle is a classic problem used in AI to demonstrate search algorithms. This engine provides the robust foundation needed to integrate advanced search techniques (like A* and BFS) in future phases.

3. Problem Statement

Effective implementation of informed search algorithms requires a reliable underlying game state engine. The challenge addressed in this phase is to create a core application that handles:

1. **State Initialization:** Generating a random board that is mathematically guaranteed to be solvable.
2. **State Transitions:** Allowing valid manual moves (tile swaps) in the console.
3. **Termination:** Accurately checking for the win state.

This project addresses the problem of **establishing a robust, console-based 8-Puzzle game engine** by using the **inversion count method** to ensure that every game instance is possible to solve, preventing infinite search loops in future AI phases.

4. Functional Requirements

Functional requirements describe the specific actions and tasks the system must be able to perform.

S. No.	Requirement	Description
1.	Board Initialization	The system must be able to randomize the board state upon startup.
2.	Solvability Check	The system **must** mathematically verify that a randomized board configuration is solvable using the inversion count method.
3.	Manual Tile Movement	The user must be able to move tiles manually using directional inputs (U, D, L, R) via the console.
4.	Win Condition	The system must accurately check if the current board state matches the defined winning state ([1, 2, 3], [4, 5, 6], [7, 8, 0]).
5.	Console Display	The current board state and move count must be clearly displayed in the console after every valid move.

5. Non-functional Requirements

Non-functional requirements describe quality attributes, constraints, and how well the system performs.

ID	Requirement	Description
1.	Efficiency	The solvability check (_is_solvable) must execute instantly during board initialization.
2.	Code Integrity	The board manipulation logic (tile swapping) must be entirely encapsulated within the EightPuzzleGame class.
3.	User Feedback	The system must provide clear textual feedback for invalid moves (e.g., "Out of bounds") or invalid inputs.
4.	Maintainability	The code must be modular, well-commented, and use constants (BOARD_SIZE, WIN_STATE, EMPTY_TILE) for core values.

6. System Architecture

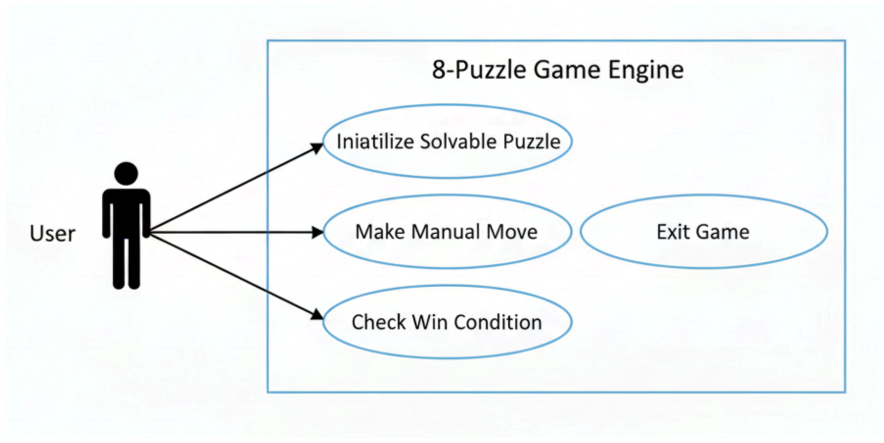
The **8-Puzzle Problem Solver** application follows a **Monolithic Client Logic** architecture implemented as a single Python script.

- **Client Logic (The Engine):** Developed entirely in **Python**, utilizing a single class, EightPuzzleGame, which encapsulates all state (the board matrix) and behavior (shuffling, moving, checking for inversions, checking for win).
- **Interface:** The system uses standard **console input/output (I/O)** for user interaction and game display.
- **Data Store:** None. The game state is maintained purely in memory during runtime.

7. Design Diagrams

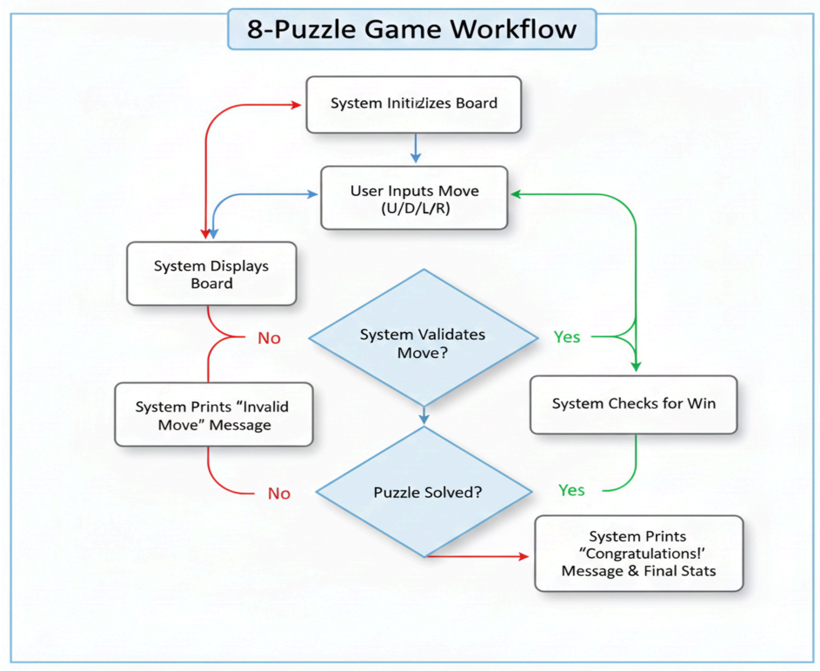
Use Case Diagram

This diagram illustrates the relationship between the primary actor, the **User**, and the fundamental processes of the game engine, including Initialize Solvable Puzzle, Make Manual Move, Check Win Condition, and Exit Game.



Workflow Diagram

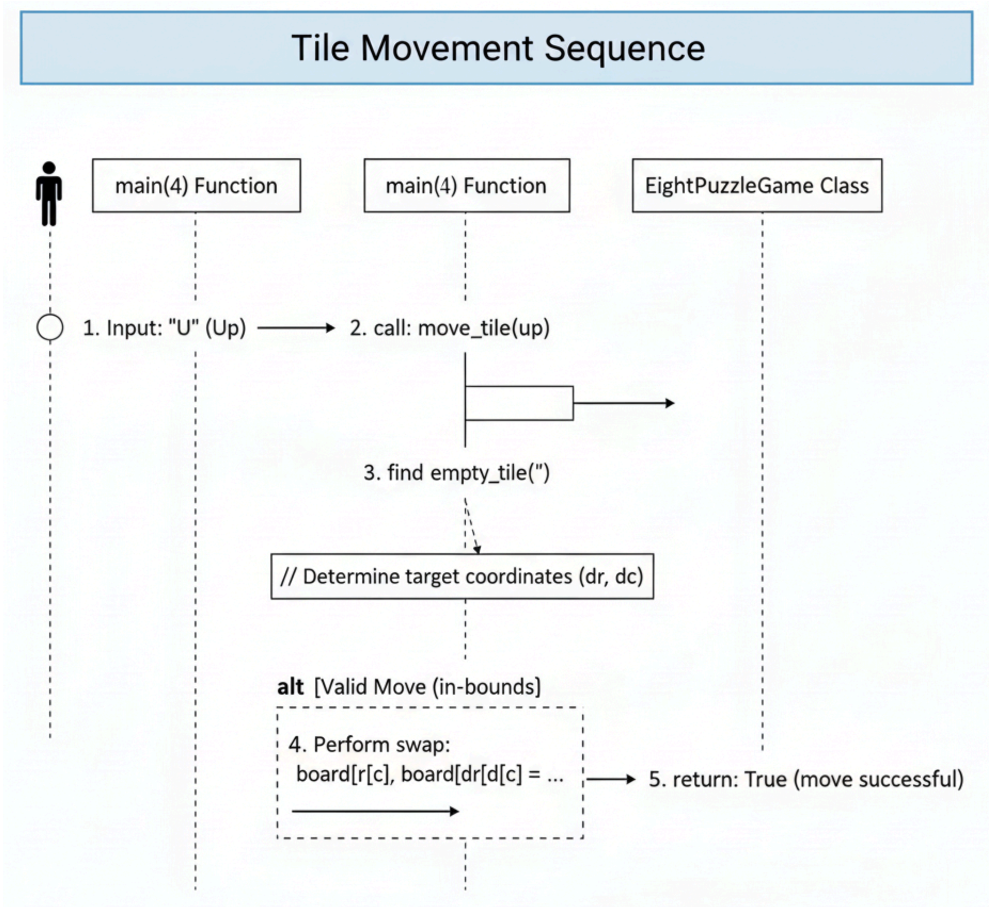
This diagram maps out the manual game loop workflow: *System initializes board ⟶System displays board⟶User inputs move ⟶ System validates move ⟶ If valid: System updates board ⟶ System checks for win ⟶ If not won: loop back.*



Sequence Diagram

The sequence diagram focuses on the process when the user attempts a tile movement:

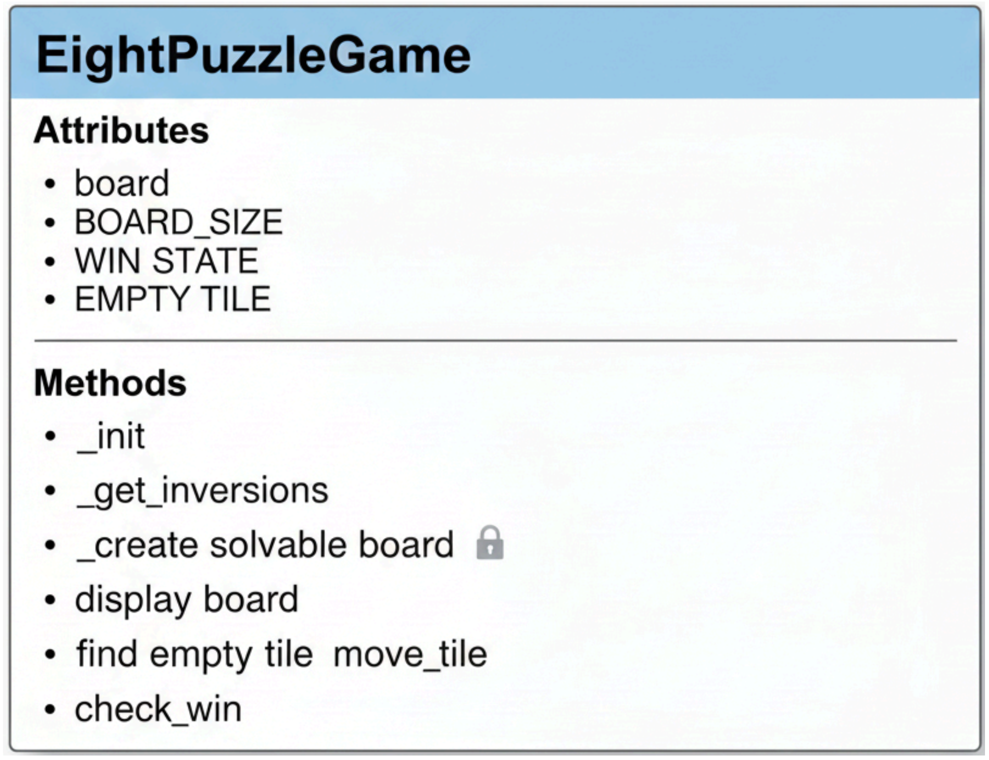
1. **The user** provides directional input (e.g., 'U') to the **main() function**.
2. **main() function** calls game.move_tile(direction).
3. **EightPuzzleGame** calls find_empty_tile() to locate the blank space.
4. **EightPuzzleGame** determines the Target Tile coordinates.
5. **EightPuzzleGame** performs boundary checks.
6. **If valid:** The empty tile (0) and the target tile are swapped.
7. **EightPuzzleGame** returns True to the **main() function**



Class/Component Diagram

This diagram visualizes the single, central class: **EightPuzzleGame**.

- **Attributes:** board, BOARD_SIZE, WIN_STATE, EMPTY_TILE.
- **Methods:** __init__, _get_inversions, _is_solvable, _create_solvable_board, display_board, find_empty_tile, move_tile, check_win.



8. Design Decisions & Rationale

Decision	**Rationale**
Technology Stack (Python/Console)	Python was chosen for its clean syntax, allowing quick development and easy implementation of complex algorithms. A console interface was sufficient for initial development, focusing on core logic over complex GUI implementation.
Solvability Check (_is_solvable)	The inversion counting method was implemented based on mathematical proofs. This ensures that the generated board is always solvable, which is critical. Failing to check solvability would result in an impossible game instance, frustrating the user and breaking any future AI solver.
Board Representation	A 2D Python list (list of lists) was chosen to represent the board, as it naturally maps to the 3*3 grid structure, simplifying tile coordinate lookups and movement logic.
Separation of Concerns	All game state and modification logic is strictly contained within the EightPuzzleGame class, separating the engine from the main() console loop, which only handles I/O.

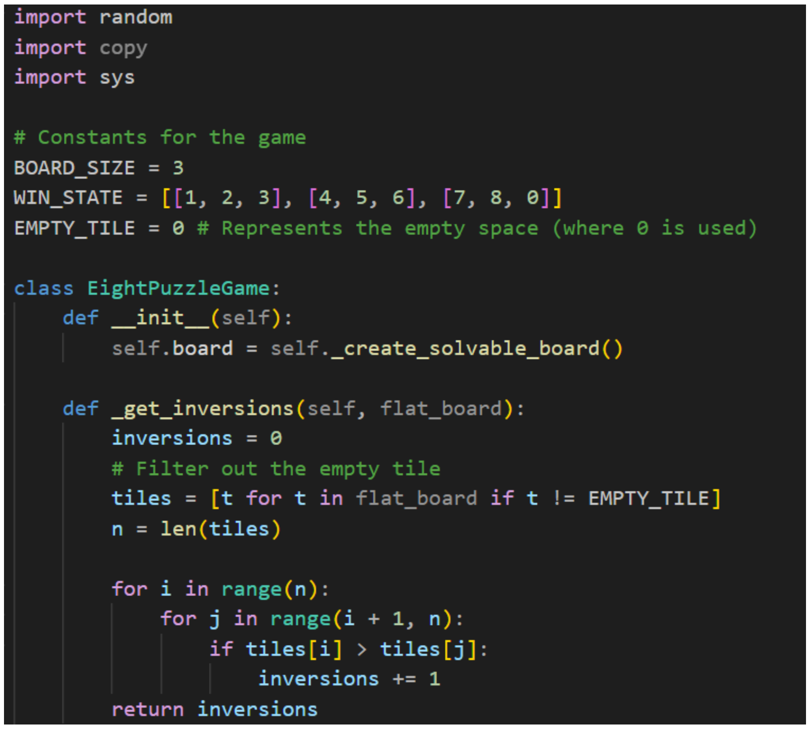
9. Implementation Details

The core implementation is based on the **EightPuzzleGame** class written in Python.

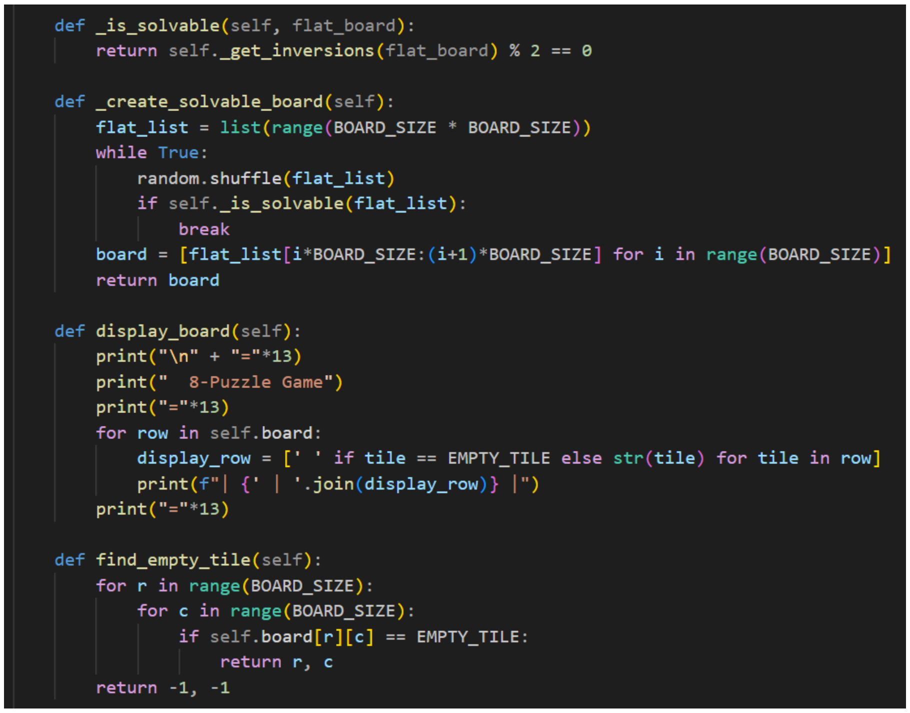
- **Initialization and Solvability:** The __init__ method calls the _create_solvable_board loop, which relies on the private methods _get_inversions and _is_solvable. The solvability logic adheres to the rule that the 8-Puzzle is solvable if the number of inversions is even.
- **State Transition (Movement):** The move_tile(direction) method performs the central state transition. It first locates the empty tile using find_empty_tile(), calculates the target tile's coordinates based on the directional input, and then uses a Python tuple swap to exchange the empty tile (0) with the target tile.
- **Win Check:** The check_win method simply compares the current 2D board list with the predefined WIN_STATE list.
- **Console I/O:** The display_board method formats the 2D list into a readable console grid, replacing the 0 with a blank space for visual clarity.

10. Screenshots / Results

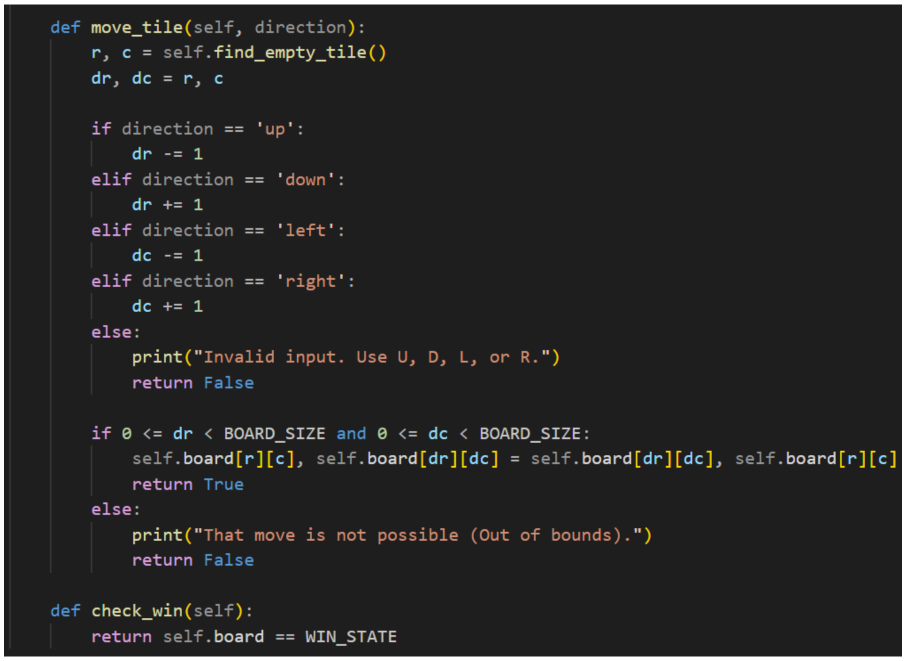
Description: This result shows the interactive nature of the console application. The user is prompted for input after the board state and current move count are displayed. The movement logic handles the swap of the empty space (0) with an adjacent tile based on directional input.



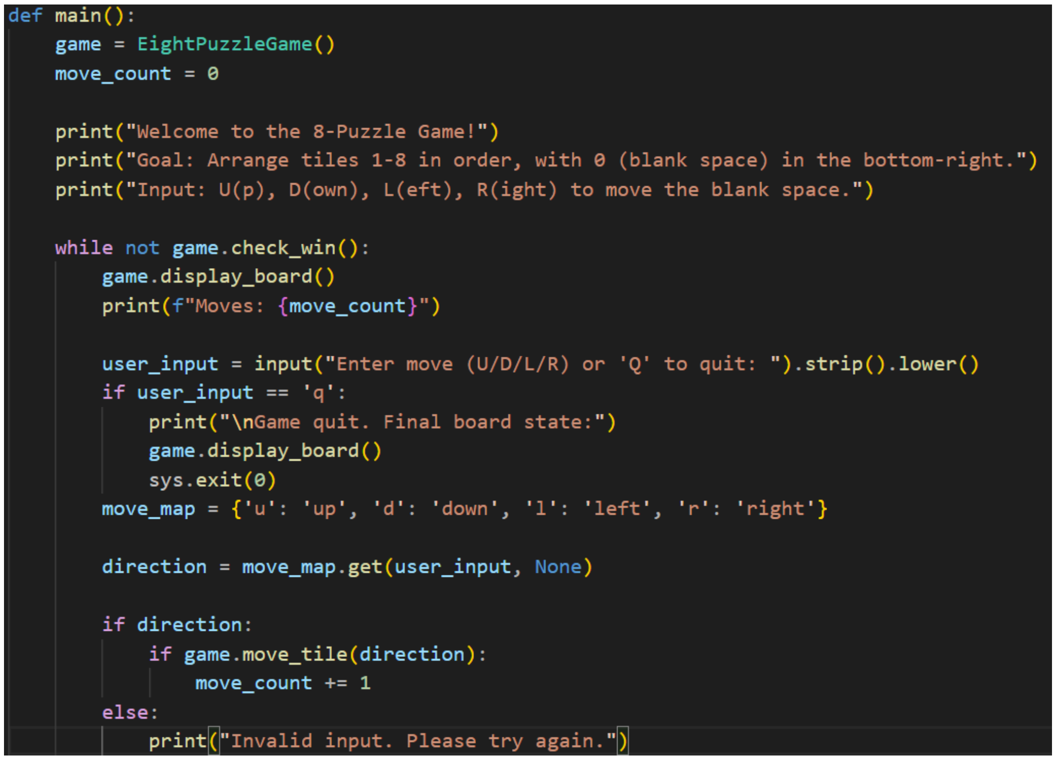
```python
import random
import copy
import sys

# Constants for the game
BOARD_SIZE = 3
WIN_STATE = [[1, 2, 3], [4, 5, 6], [7, 8, 0]]
EMPTY_TILE = 0 # Represents the empty space (where 0 is used)

class EightPuzzleGame:
    def __init__(self):
        self.board = self._create_solvable_board()

    def _get_inversions(self, flat_board):
        inversions = 0
        # Filter out the empty tile
        tiles = [t for t in flat_board if t != EMPTY_TILE]
        n = len(tiles)

        for i in range(n):
            for j in range(i + 1, n):
                if tiles[i] > tiles[j]:
                    inversions += 1
        return inversions
```



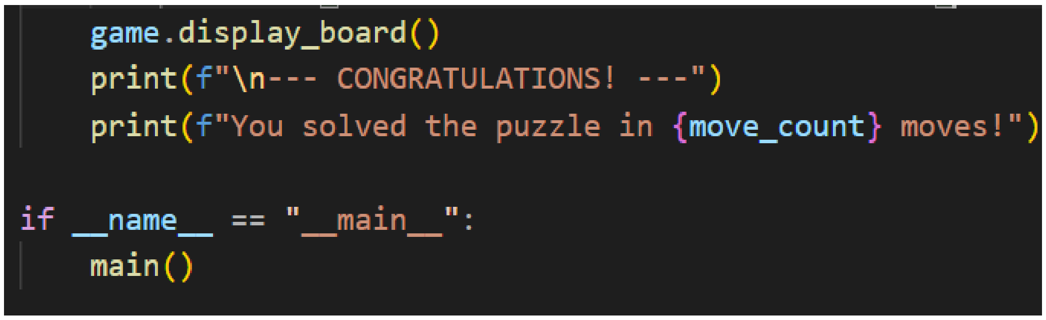
```python
    def _is_solvable(self, flat_board):
        return self._get_inversions(flat_board) % 2 == 0

    def _create_solvable_board(self):
        flat_list = list(range(BOARD_SIZE * BOARD_SIZE))
        while True:
            random.shuffle(flat_list)
            if self._is_solvable(flat_list):
                break
        board = [flat_list[i*BOARD_SIZE:(i+1)*BOARD_SIZE] for i in range(BOARD_SIZE)]
        return board

    def display_board(self):
        print("\n" + "="*13)
        print("  8-Puzzle Game")
        print("="*13)
        for row in self.board:
            display_row = [' ' if tile == EMPTY_TILE else str(tile) for tile in row]
            print(f"| {' | '.join(display_row)} |")
        print("="*13)

    def find_empty_tile(self):
        for r in range(BOARD_SIZE):
            for c in range(BOARD_SIZE):
                if self.board[r][c] == EMPTY_TILE:
                    return r, c
        return -1, -1
```



```python
    def move_tile(self, direction):
        r, c = self.find_empty_tile()
        dr, dc = r, c

        if direction == 'up':
            dr -= 1
        elif direction == 'down':
            dr += 1
        elif direction == 'left':
            dc -= 1
        elif direction == 'right':
            dc += 1
        else:
            print("Invalid input. Use U, D, L, or R.")
            return False

        if 0 <= dr < BOARD_SIZE and 0 <= dc < BOARD_SIZE:
            self.board[r][c], self.board[dr][dc] = self.board[dr][dc], self.board[r][c]
            return True
        else:
            print("That move is not possible (Out of bounds).")
            return False

    def check_win(self):
        return self.board == WIN_STATE
```

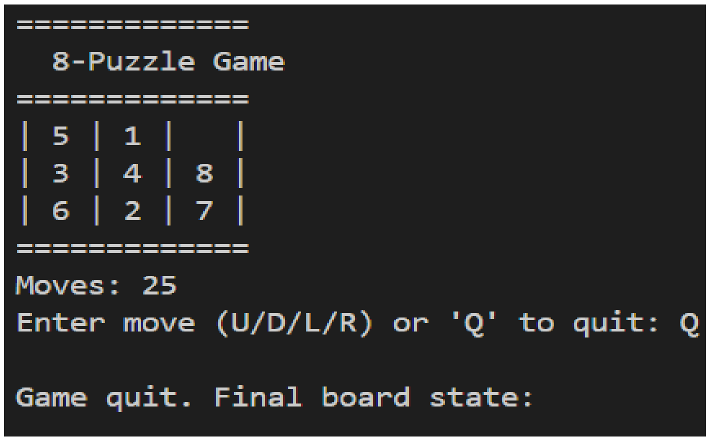


```python
def main():
    game = EightPuzzleGame()
    move_count = 0

    print("Welcome to the 8-Puzzle Game!")
    print("Goal: Arrange tiles 1-8 in order, with 0 (blank space) in the bottom-right.")
    print("Input: U(p), D(own), L(eft), R(ight) to move the blank space.")

    while not game.check_win():
        game.display_board()
        print(f"Moves: {move_count}")

        user_input = input("Enter move (U/D/L/R) or 'Q' to quit: ").strip().lower()
        if user_input == 'q':
            print("\nGame quit. Final board state:")
            game.display_board()
            sys.exit(0)
        move_map = {'u': 'up', 'd': 'down', 'l': 'left', 'r': 'right'}

        direction = move_map.get(user_input, None)

        if direction:
            if game.move_tile(direction):
                move_count += 1
        else:
            print("Invalid input. Please try again.")
```



```python
    game.display_board()
    print(f"\n--- CONGRATULATIONS! ---")
    print(f"You solved the puzzle in {move_count} moves!")

if __name__ == "__main__":
    main()
```

OUTPUT :



```
=============
  8-Puzzle Game
=============
| 5 | 1 |   |
| 3 | 4 | 8 |
| 6 | 2 | 7 |
=============
Moves: 25
Enter move (U/D/L/R) or 'Q' to quit: Q

Game quit. Final board state:
```

11. Testing Approach

Testing focused on verifying the correctness and integrity of the core game engine logic, particularly the initialization and state transition mechanisms.

- **Unit Testing (Solvability Check):** Tested known unsolvable and solvable flat board configurations against the _is_solvable function to ensure correct inversion counting and solvability determination.
- **Integration Testing (Initialization):** Repeatedly ran the _create_solvable_board function multiple times to ensure the generated board was always random but passed the _is_solvable check, preventing the creation of unwinnable games.
- **Functional Testing (Movement):** Manually tested the move_tile function by inputting all four directions from various board positions (corners, edges, center) to verify: 1) the correct tile was swapped, and 2) moves outside the board boundaries were correctly rejected with feedback.

12. Challenges Faced

1. **Implementing the Inversion Logic:** The initial challenge was correctly calculating the number of inversions on the flattened board while ensuring the empty tile (0) was properly ignored, a necessity for the mathematical rule governing solvability.
2. **Robust Randomization:** Ensuring the board randomization loop efficiently produced a solvable state without excessive recursive calls or long delays, although the probability of an unsolvable state is 50/50, requiring the loop structure.
3. **Clean Console Display:** Formatting the 2D list output to look like a clean 3*3 grid, including handling the empty space character, required careful use of Python's string formatting features.

13. Learnings & Key Takeaways

1. **Mathematical Constraints in Games:** The most significant learning was the critical role of the mathematical constraint (inversion parity) in guaranteeing a solvable game state, a fundamental requirement for any search problem.
2. **Object-Oriented Design:** Reinforcement of using a dedicated class (EightPuzzleGame) to manage state, ensuring that the board state is protected and only modified by defined class methods.
3. **Foundation for AI:** This phase solidified the understanding that a reliable game engine is a prerequisite for implementing and testing complex AI search algorithms effectively.

14. Future Enhancements

1. **AI Solver Integration:** Implement the core search algorithms (Breadth-First Search, A* with Manhattan Distance heuristic) by integrating new classes that leverage the move_tile and check_win methods of the current engine.
2. **Graphical User Interface (GUI):** Migrate the application from the console to a web (e.g., React/HTML) or desktop GUI using a framework like Tkinter or Pygame, enabling visual interaction and algorithm visualization.
3. **Performance Metrics:** Introduce a mechanism to track and display performance metrics during the AI solving process (e.g., time taken, number of nodes expanded, path length).

15. References

1. Python Documentation (https://docs.python.org/)
2. Russell, Stuart J., and Peter Norvig. *Artificial Intelligence: A Modern Approach*. (General reference for the 8-Puzzle and Solvability check theory).