PROJECT NAME: 15-PUZZLE SOLVER GAME

SHOAIB ALI (BIT-24S-003)

ADDUL SALAM (BIT-24S-012)

GOVINDA (BIT-24S-031)

AHSAN ALI (BIT-24S-032)

What is the 15-Puzzle?

- The 15-puzzle is a 4x4 grid-based sliding puzzle with 15 numbered tiles and one empty space.

- The tiles are randomly arranged and must be moved to achieve a specific order:
- 1 2 3 4
- 5 6 7 8
- 9 10 11 12
- 13 14 15 [ ]

- Only tiles adjacent to the empty space can be moved.

- It is used widely in artificial intelligence for testing search algorithms and heuristics.

Project Objectives

- Create an interactive puzzle game using Python.
- Implement an AI solver using the A* search algorithm.
- Use Manhattan Distance as a heuristic to guide the search.
- Provide a Graphical User Interface (GUI) using Tkinter.
- Optional: Add features like AI hints, step-by-step solving, and image tiles.

A Algorithm Explanation

- A* is an informed search algorithm used in pathfinding and graph traversal.
- It uses the evaluation function:
- $f(n) = g(n) + h(n)$
- where:
- $g(n)$ is the actual cost from the start to node n.
- $h(n)$ is the estimated cost from node n to the goal.
- A* uses a priority queue to explore the lowest-cost paths first.
- It guarantees optimal and complete solutions when the heuristic is admissible (never overestimates).

Manhattan Distance

A heuristic function that estimates the distance of each tile from its goal position.

For each tile, the distance is:

|current_x - goal_x| + |current_y - goal_y|

Total Manhattan Distance = sum of all tile distances.

It is admissible and consistent, making it ideal for A* in the 15-puzzle.

More accurate than the "number of misplaced tiles" heuristic.

System Architecture

1. GUI Module (Tkinter):

Renders the puzzle grid and buttons.

Handles user input (tile clicks).

2. Puzzle Logic:

Manages tile states, valid moves, and board updates.

Detects solved state.

- 3. Solver Module (A Algorithm):*
- Builds node states and uses a priority queue to explore paths.
- Uses Manhattan Distance to estimate cost to goal.
- 4. Threading (Optional):
- Solving runs in a separate thread to keep the GUI responsive.

: Features and Enhancements

- Randomly shuffled board at launch.
- Tile movement through mouse clicks.
- Auto-solve button using A* algorithm.
- Optional:
- AI hint system to suggest next best move.
- Image-based tiles for a visual version.
- Step-by-step animation of the solution path.
- Undo/redo functionality.

Conclusion

- This project showcases AI in action using search algorithms.
- It blends game development, GUI design, and algorithmic logic.
- A great learning tool to understand heuristics, priority queues, and state exploration.
- Can be extended further for advanced user interaction and mobile deployment.

Coding this project

- import tkinter as tk
- import heapq
- import random
- from threading import Thread

- # Heuristic: Manhattan Distance
- def manhattan(puzzle):
-     distance = 0
-     for i, val in enumerate(puzzle):
-         if val == 0:
-             continue

```
 goal_row, goal_col = divmod(val - 1, 4)
         curr_row, curr_col = divmod(i, 4)
         distance += abs(goal_row - curr_row) + abs(goal_col - curr_col)
    return distance

# Generate valid moves
def get_neighbors(state):
    neighbors = []
    idx = state.index(0)
    row, col = divmod(idx, 4)
    moves = [(-1, 0), (1, 0), (0, -1), (0, 1)]
```

```
for dr, dc in moves:
        new_r, new_c = row + dr, col + dc
        if 0 <= new_r < 4 and 0 <= new_c < 4:
            new_idx = new_r * 4 + new_c
            new_state = list(state)
            new_state[idx], new_state[new_idx] = new_state[new_idx], new_state[idx]
            neighbors.append(tuple(new_state))
    return neighbors
```

A* Algorithm

```
def a_star(start):
    goal = tuple(range(1, 16)) + (0,)
    open_set = []
    heapq.heappush(open_set, (manhattan(start), 0, start, []))
    visited = set()

    while open_set:
        est_total, cost, curr, path = heapq.heappop(open_set)
        if curr in visited:
            continue
        visited.add(curr)
        if curr == goal:
            return path
        for neighbor in get_neighbors(curr):
            if neighbor not in visited:
                heapq.heappush(open_set, (cost + 1 + manhattan(neighbor), cost + 1, neighbor, path + [neighbor]))
    return None
```

- GUI Class
- class PuzzleGUI:
-     def _init_(self, master):
-         self.master = master
-         self.master.title("15-Puzzle Solver with A*")
-         self.board = list(range(1, 16)) + [0]
-         while True:
-             random.shuffle(self.board)
-             if self.is_solvable(self.board):
-                 break
-         self.buttons = []
-         self.draw_board()

-         solve_btn = tk.Button(master, text="Solve Puzzle", command=self.solve)

```python
solve_btn.grid(row=4, column=0, columnspan=4, sticky="nsew")

def draw_board(self):
    for i in range(16):
        row, col = divmod(i, 4)
        num = self.board[i]
        if len(self.buttons) < 16:
            btn = tk.Button(self.master, text=str(num) if num != 0 else "", width=6, height=3,
                            command=lambda i=i: self.move_tile(i))
            btn.grid(row=row, column=col)
            self.buttons.append(btn)
        else:
            self.buttons[i].config(text=str(num) if num != 0 else "")
```

```python
def move_tile(self, i):
    zero = self.board.index(0)
    if abs(zero - i) in (1, 4) and (zero // 4 == i // 4 or zero % 4 == i % 4):
        self.board[zero], self.board[i] = self.board[i], self.board[zero]
        self.draw_board()

  def solve(self):
    def auto():
        path = a_star(tuple(self.board))
        if path:
            for step in path:
                self.board = list(step)
                self.draw_board()
                self.master.update()
                self.master.after(200)
```

```
    def is_solvable(self, board):
        inv = 0
        for i in range(15):
            for j in range(i + 1, 16):
                if board[i] and board[j] and board[i] > board[j]:
                    inv += 1
        row = board.index(0) // 4
        return (inv + row) % 2 == 0

# Run GUI
if _name_ == "_main_":
    root = tk.Tk()
    app = PuzzleGUI(root)
    root.mainloop()
```