AI - MSE1 PROJECT - PUZZLE SOLVER

(AI101B)

MCA 2nd SEM (C)

SUBMITTED BY : -

1. Riya Gaur(202410116100171)
2. Priyanshi(202410116100153)
3. Riya(202410116100170)
4. Pallavi(202410116100139)

UNDER THE SUPERVISION OF:

Ms. Komal Salgotra

ABSTRACT

The "Puzzle Solver" is an interactive Python-based sliding puzzle game designed to enhance problem-solving skills and logical thinking. Developed using the Pygame library, the game presents a shuffled 4x4 grid of tiles, where players must rearrange the tiles into their correct order by strategically sliding them into the empty space.

This project highlights Python's efficiency in game development by integrating graphical rendering, event-driven programming, and logic-based tile movement. The game tracks player moves, enforces valid shifts, and introduces a time-based challenge to enhance engagement. Additionally, the intuitive user interface ensures a seamless and enjoyable gaming experience.

By combining structured logic, graphical elements, and real-time user interaction, the "Puzzle Solver" serves as both an entertaining and educational tool, showcasing the practical application of programming concepts in game design while improving cognitive skills

INTRODUCTION

The "Puzzle Solver" is a Python-based game application that challenges players with a classic sliding puzzle. Built using the Pygame library, this game divides an image into a 4x4 grid of tiles and shuffles them, requiring the player to rearrange the pieces to restore the original image. The game integrates interactive mechanics, such as move counting and a time-based challenge, making it engaging for users.

Sliding puzzles have been a popular form of entertainment and brain training for decades, helping individuals develop spatial reasoning and problem-solving skills. This project brings that experience to a digital platform using Python, making it easily accessible and interactive. The game offers a well-designed graphical interface where users can interact with the tiles and attempt to solve the puzzle efficiently. It challenges players to think ahead, plan their moves strategically, and complete the puzzle before the time runs out.

By leveraging Python’s powerful libraries, the project highlights the practical application of programming in game development. The game employs an intuitive tile-based design, allowing users to move individual puzzle pieces into an empty space to rearrange the image. The logic behind the tile movement ensures that only valid moves are executed, maintaining the structure of the puzzle. Additionally, the game provides visual cues to enhance user experience, such as highlighting movable tiles and displaying real-time updates of move counts and elapsed time.

This project is an excellent demonstration of how Python can be used to create interactive and engaging games. It showcases the ability to handle graphics rendering, event-driven programming, and logic-based movement within a structured and user-friendly environment. By implementing a sliding puzzle game, this project aims to provide entertainment while also enhancing cognitive skills, making it both a fun and intellectually stimulating experience.

SCOPE OF THE PROJECT

The Puzzle Solver project is designed as an interactive sliding puzzle game that challenges players to arrange shuffled tiles back into their correct positions. The game not only provides entertainment but also serves an educational purpose by enhancing logical reasoning and problem-solving skills.

This project is particularly useful for:

1. Students & Learners: Helps in improving critical thinking and spatial awareness.

2. Developers & Enthusiasts: Demonstrates Python's capabilities in game development using Pygame.

3. Casual Gamers: Offers an engaging and challenging experience through time-based gameplay and strategic moves.

4. The game can further be expanded to include different difficulty levels, larger grids, or even AI-based puzzle-solving techniques.

FEATURES OF THE PUZZLE SOLVER

1. The Puzzle Solver game includes several interactive and functional features:

2. 4x4 Sliding Puzzle Grid: The game board consists of a 4x4 grid where one tile space is left empty to allow movement.

3. Randomized Shuffling: The tiles are automatically shuffled at the beginning to provide a unique challenge every time.

4. Tile Movement Mechanics: Players can move tiles into the empty space by using valid shifts, ensuring logical gameplay.

5. Move Counter: The game tracks the number of moves a player makes, encouraging strategic decision-making.

6. Time-Based Challenge: A countdown timer enhances the difficulty level, making the game more engaging.

7. Graphical Interface: Designed using Pygame, the UI is simple, interactive, and easy to navigate.

8. Real-time Event Handling: Uses event-driven programming to capture user input and update the game state dynamically.

METHODOLOGY

The development of the Puzzle Solver game follows a structured methodology that includes problem analysis, algorithm design, implementation, and testing. The game is built using Python and the Pygame library, integrating graphical elements, event-driven programming, and logical problem-solving mechanisms. The methodology is broken down into the following phases:

1. Problem Analysis & Conceptualization

Before starting the development, an analysis was conducted to determine the key requirements of the game:

a. A 4x4 sliding puzzle grid where tiles must be arranged in ascending order.

b. A graphical interface for an engaging and interactive experience.

c. A shuffling algorithm to ensure a randomized game state.

d. Tile movement mechanics with valid move verification.

e. Win condition detection to confirm puzzle completion.

f. Move counter & timer for tracking player performance.

2. Algorithm Design & Logical Framework

The core logic of the Puzzle Solver game is structured as follows:

A. Board Representation & Initialization

a. A 4x4 grid is represented as a 2D list, where each tile is assigned a unique number (1 to 15), and one space is left empty.

b. A shuffle function randomly rearranges the tiles to create different challenges.

B. Tile Movement & Validity Checking

a. The game listens for user input (keyboard or mouse clicks).

b. A function ensures that a move is valid (only adjacent tiles can move into the empty space).

c. If the move is valid, the selected tile swaps its position with the empty space.

C. Shuffle Algorithm

a. A random shuffle function is implemented to rearrange the tiles.

b. To avoid unsolvable puzzles, an inversion count method can be applied to check solvability.

D. Win Condition Detection

a. After every move, the program checks whether all tiles are arranged in ascending order (from 1 to 15).

b. If the arrangement is correct, a victory message is displayed and the game ends.

3. Implementation Phase

The game was developed using Python and Pygame with the following key components:

A. User Interface (UI) Development

a. The graphical layout is created using Pygame's rendering functions.

b. Tiles are drawn dynamically based on their current position in the 2D grid.

c. Colors, fonts, and animations are used to improve visual appeal.

B. Event Handling & Game Loop

a. The game runs using an event-driven programming model.

b. The event loop continuously listens for user actions and updates the game state accordingly.

c. Keyboard and mouse events are processed to detect tile movements.

C. Move Counter & Timer

a. A counter keeps track of the number of moves a player makes.

b. A timer starts when the game begins and stops when the puzzle is solved.

c. The move counter and timer are displayed on the game screen.

4. Testing & Debugging

After implementation, the game was tested under various conditions:

A. Functional Testing

a. Ensuring all tiles move correctly according to the rules.

b. Verifying shuffle randomness for diverse gameplay experiences.

c. Checking winning condition logic for correct detection.

B. Performance Optimization

a. The event loop was optimized to reduce lag and ensure smooth transitions.

b. Efficient rendering techniques were used to improve performance.

5. Future Enhancements

a. While the current version of the game functions effectively, future improvements can include:

b. Different difficulty levels (3x3, 5x5 grids).

c. AI-based solver that auto-solves the puzzle for educational purposes.

d. Multiplayer mode where users compete for the fastest solution.

e. Leaderboard system to track high scores and best times.

CODE OF THE PROJECT

File nmae : - puzzle_solver.py

```
import pygame
import random
import os
import time

# Initialize pygame
pygame.init()
pygame.mixer.init()

# Game settings
GRID_SIZE = 4
TILE_SIZE = 600 // GRID_SIZE
IMAGE_PATH = "puzzle_image.jpg"
SOUND_PATH = "move.wav"
MUSIC_PATH = "background.mp3"

# Load image
original_image = pygame.image.load(IMAGE_PATH)
image = pygame.transform.scale(original_image, (600, 600))
reference_image = pygame.transform.scale(original_image, (300, 300))

# Load sounds
move_sound = pygame.mixer.Sound(SOUND_PATH) if
os.path.exists(SOUND_PATH) else None

# Load background music
if os.path.exists(MUSIC_PATH):
    pygame.mixer.music.load(MUSIC_PATH)
    pygame.mixer.music.play(-1)

# Timer & Moves
TOTAL_TIME = 2 * 60  # 2 minutes in seconds
start_time = time.time()
moves = 0

# Create tiles
tiles = []
tile_map = {}
```

```
tile_positions = [(x, y) for y in range(GRID_SIZE) for x in
range(GRID_SIZE)]

for i, (x, y) in enumerate(tile_positions[:-1]):
    rect = pygame.Rect(x * TILE_SIZE, y * TILE_SIZE, TILE_SIZE,
TILE_SIZE)
    tile_image = image.subsurface(rect).copy()
    tiles.append((tile_image, (x, y)))
    tile_map[(x, y)] = i

empty_tile = tile_positions[-1]
tiles.append((None, empty_tile))
tile_map[empty_tile] = len(tiles) - 1

# Shuffle Function (Ensures solvability)
def shuffle_tiles():
    global tiles, empty_tile
    positions = [tile[1] for tile in tiles]
    random.shuffle(positions)
    while not is_solvable(positions):
        random.shuffle(positions)
    for i, pos in enumerate(positions):
        tiles[i] = (tiles[i][0], pos)
    empty_tile = positions[-1]

def is_solvable(positions):
    inversion_count = 0
    flat_list = [pos for pos in positions if pos != empty_tile]
    for i in range(len(flat_list)):
        for j in range(i + 1, len(flat_list)):
            if flat_list[i] > flat_list[j]:
                inversion_count += 1
    return inversion_count % 2 == 0 if GRID_SIZE % 2 == 1 else
(inversion_count + GRID_SIZE - empty_tile[1]) % 2 == 0

shuffle_tiles()

# Initialize screen
screen = pygame.display.set_mode((1000, 600))
pygame.display.set_caption("Super Smart Puzzle")

# Draw Function
def draw_tiles():
```

```
    screen.fill((135, 206, 250))  # Sky Blue Background

    # Reference image box
    pygame.draw.rect(screen, (150, 150, 150), (650, 100, 320, 320),
border_radius=20)
    screen.blit(reference_image, (660, 110))
    pygame.draw.rect(screen, (255, 255, 255), (660, 110, 300, 300), 3)

    # "Can You Solve It?" text
    title_font = pygame.font.Font(None, 60)
    title_text = title_font.render("Can You Solve It?", True, (0, 0, 0))
    text_rect = title_text.get_rect(center=(810, 50))
    screen.blit(title_text, text_rect)

    # Draw Tiles
    for tile, pos in tiles:
        if tile:
            screen.blit(tile, (pos[0] * TILE_SIZE, pos[1] * TILE_SIZE))

    # Countdown Timer & Moves Counter
    font = pygame.font.Font(None, 36)
    elapsed_time = int(time.time() - start_time)
    remaining_time = max(0, TOTAL_TIME - elapsed_time)  # Count
down time
    minutes = remaining_time // 60
    seconds = remaining_time % 60
    timer_text = f"Time: {minutes}:{seconds:02d}  Moves: {moves}"

    if remaining_time == 0:
        show_time_up_screen()
    else:
        timer_surface = font.render(timer_text, True, (0, 0, 0))
        screen.blit(timer_surface, (660, 450))

    # Winning Message
    if is_solved():
        win_font = pygame.font.Font(None, 50)
        win_text = win_font.render("    Puzzle Solved!    ", True, (0, 255, 0))
        screen.blit(win_text, (660, 500))

    pygame.display.flip()

# Full-Screen "Time's Up!" Message
```

```
def show_time_up_screen():
    screen.fill((0, 0, 0))  # Black Background
    font = pygame.font.Font(None, 100)
    text = font.render("⏳ TIME'S UP! ⏳", True, (255, 0, 0))  # Red Color
    text_rect = text.get_rect(center=(500, 300))
    screen.blit(text, text_rect)
    pygame.display.flip()
    time.sleep(3)  # Show for 3 seconds
    pygame.quit()
    exit()

# Check Winning Condition
def is_solved():
    for i, (tile, pos) in enumerate(tiles):
        if pos != tile_positions[i]:
            return False
    return True

# Get adjacent tiles
def get_adjacent_tiles():
    x, y = empty_tile
    moves = [(x-1, y), (x+1, y), (x, y-1), (x, y+1)]
    return [pos for pos in moves if 0 <= pos[0] < GRID_SIZE and 0 <=
pos[1] < GRID_SIZE]

# Move tile function
def move_tile(clicked_tile):
    global empty_tile, moves
    for i, (_, pos) in enumerate(tiles):
        if pos == clicked_tile:
            tiles[i] = (tiles[i][0], empty_tile)
            empty_tile = clicked_tile
            moves += 1
            if move_sound:
                move_sound.play()
            break

# Game loop
running = True
while running:
    draw_tiles()
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
```

```
            running = False
        elif event.type == pygame.MOUSEBUTTONDOWN:
            mouse_x, mouse_y = pygame.mouse.get_pos()
            if mouse_x < 600:
                clicked_tile = (mouse_x // TILE_SIZE, mouse_y // TILE_SIZE)
                if clicked_tile in get_adjacent_tiles():
                    move_tile(clicked_tile)

                            pygame.quit()
```

FILE NAME : - temp.CodeRunnerFile.py

```
import pygame
import random
import os
import time

# Initialize pygame
pygame.init()
pygame.mixer.init()

# Game settings
GRID_SIZE = 4  # 4x4 grid
TILE_SIZE = 600 // GRID_SIZE
IMAGE_PATH = "puzzle_image.jpg"
SOUND_PATH = "move.wav"
MUSIC_PATH = "background.mp3"

# Load image
original_image = pygame.image.load(IMAGE_PATH)
image = pygame.transform.scale(original_image, (600, 600))
```

```
reference_image = pygame.transform.scale(original_image, (300, 300))

# Load move sound
move_sound = pygame.mixer.Sound(SOUND_PATH) if
os.path.exists(SOUND_PATH) else None

# Load background music
if os.path.exists(MUSIC_PATH):
    pygame.mixer.music.load(MUSIC_PATH)
    pygame.mixer.music.play(-1)  # Loop music infinitely

# Timer & Moves
start_time = time.time()
moves = 0

# Create tiles
tiles = []
tile_map = {}
tile_positions = [(x, y) for y in range(GRID_SIZE) for x in
range(GRID_SIZE)]

for i, (x, y) in enumerate(tile_positions[:-1]):
    rect = pygame.Rect(x * TILE_SIZE, y * TILE_SIZE, TILE_SIZE,
TILE_SIZE)
    tile_image = image.subsurface(rect).copy()
    tiles.append((tile_image, (x, y)))
    tile_map[(x, y)] = i

empty_tile = tile_positions[-1]
```

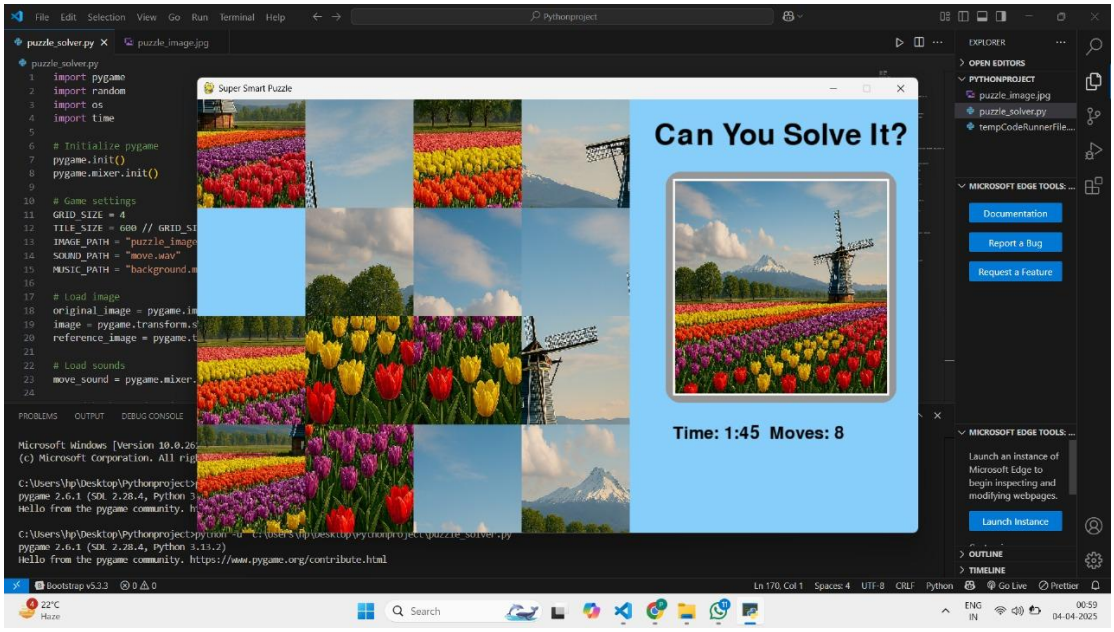
```
tiles.append((None, empty_tile))
tile_map[empty_tile] = len(tiles) - 1

# Shuffle Function (Ensures solvability)
def shuffle_tiles():
    global tiles, empty_tile
    positions = [tile[1] for tile in tiles]
    random.shuffle(positions)
    while not is_solvable(positions):
        random.shuffle(positions)
    for i, pos in enumerate(positions):
        tiles[i] = (tiles[i][0], pos)
    empty_tile = positions[-1]

def is_solvable(positions):
    inversion_count = 0
    flat_list = [pos for pos in positions if pos != empty_tile]
    for i in range(len(flat_list)):
        for j in range(i + 1, len(flat_list)):
            if flat_list[i] > flat_list[j]:
                inversion_count += 1
    return inversion_count % 2 == 0 if GRID_SIZE % 2 == 1 else
(inversion_count + GRID_SIZE - empty_tile[1]) % 2 == 0

shuffle_tiles()

# Initialize screen
screen = pygame.display.set_mode((1000, 600))
pygame.display.set_caption("Super Smart Puzzle")
```

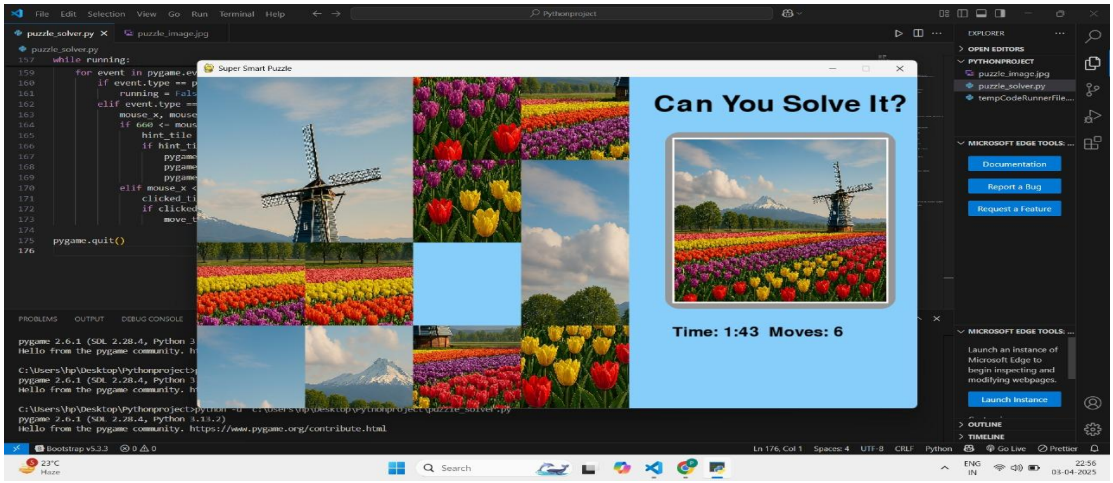
```
# Draw Function
def draw_tiles():
    screen.fill((30, 30, 30))
    pygame.draw.rect(screen, (150, 150, 150), (650, 100, 320, 320),
border_radius=20)
    screen.blit(reference_image, (660, 110))
    pygame.draw.rect(screen, (255, 255, 255), (660, 110, 300, 300), 3)

    for tile, pos in tiles:
        if tile:
            screen.blit(tile, (pos[0] * TILE_SIZE, pos[1] * TILE_SIZE))

    # Timer and Moves Counter
    font = pygame.font.Font(None, 36)
    elapsed_time = round(time.time() - start_time)
    text = font.render(f"Time: {elapsed_time}s  Moves: {moves}", True,
(255, 255, 255))
    screen.blit(text, (660, 450))

    # Winning Message
    if is_solved():
        font = pygame.font.Font(None, 50)
        win_text = font.render("    Puzzle Solved!    ", True, (0, 255, 0))
        screen.blit(win_text, (660, 500))

    pygame.display.flip()

# Check Winning Condition
```

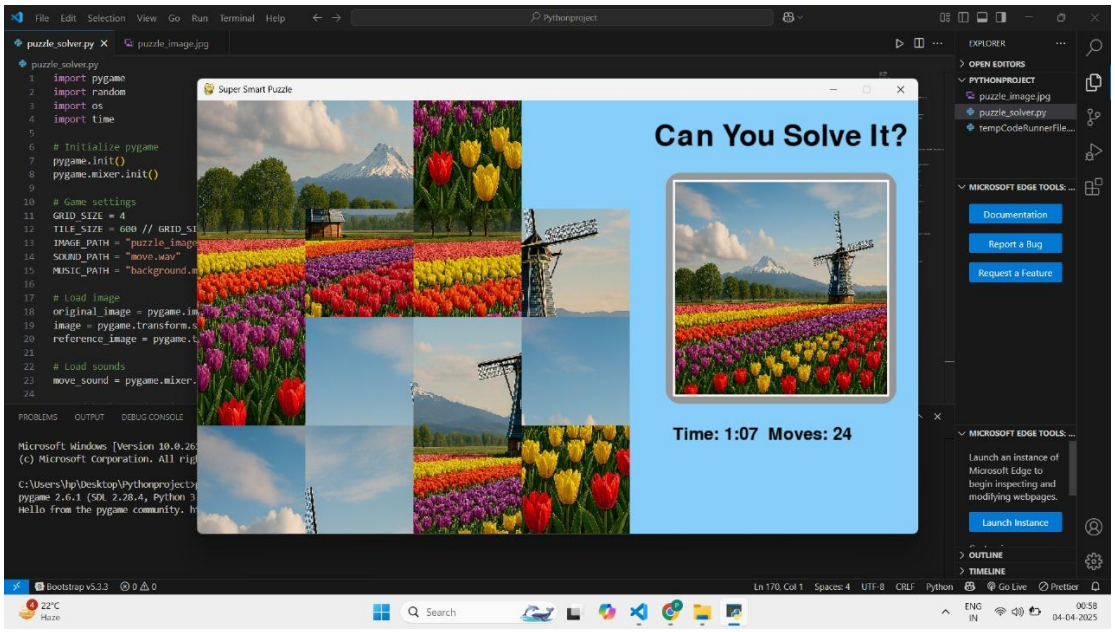
```
def is_solved():
    for i, (tile, pos) in enumerate(tiles):
        if pos != tile_positions[i]:
            return False
    return True

# Get adjacent tiles
def get_adjacent_tiles():
    x, y = empty_tile
    moves = [(x-1, y), (x+1, y), (x, y-1), (x, y+1)]
    return [pos for pos in moves if 0 <= pos[0] < GRID_SIZE and 0 <=
pos[1] < GRID_SIZE]

# Move tile function
def move_tile(clicked_tile):
    global empty_tile, moves
    for i, (_, pos) in enumerate(tiles):
        if pos == clicked_tile:
            tiles[i] = (tiles[i][0], empty_tile)
            empty_tile = clicked_tile
            moves += 1
            if move_sound:
                move_sound.play()
            break

# Hint System
def get_hint():
    for i, (tile, pos) in enumerate(tiles):
        if pos != tile_positions[i] and tile is not None:
```

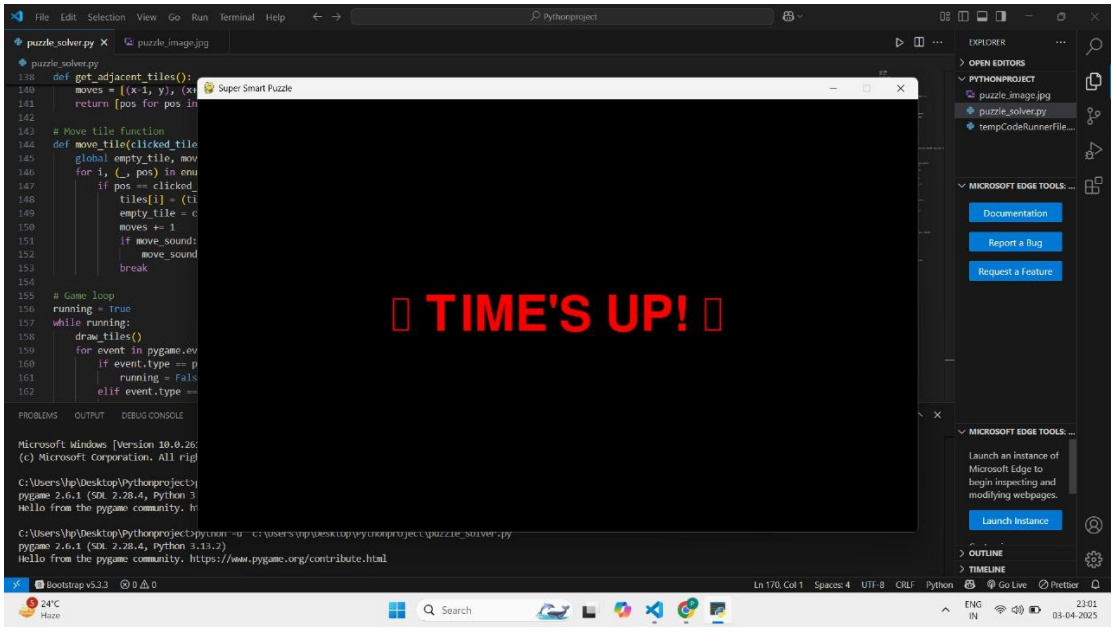
```
            return pos  # Returns a wrong tile position for hint
    return None

# Game loop
running = True
while running:
    draw_tiles()
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            running = False
        elif event.type == pygame.MOUSEBUTTONDOWN:
            mouse_x, mouse_y = pygame.mouse.get_pos()
            if 660 <= mouse_x <= 960 and 500 <= mouse_y <= 540:  # Hint
button
                hint_tile = get_hint()
                if hint_tile:
                    pygame.draw.rect(screen, (255, 0, 0), (hint_tile[0] *
TILE_SIZE, hint_tile[1] * TILE_SIZE, TILE_SIZE, TILE_SIZE), 5)
                    pygame.display.flip()
                    pygame.time.delay(500)
            elif mouse_x < 600:
                clicked_tile = (mouse_x // TILE_SIZE, mouse_y // TILE_SIZE)
                if clicked_tile in get_adjacent_tiles():
                    move_tile(clicked_tile)
pygame.quit()
```

SCREENSHOTS






Super Smart Puzzle
Can You Solve It?
Time: 1:07 Moves: 24


Super Smart Puzzle
TIME'S UP!