Kyle Tranfaglia

COSC 411: Artificial Intelligence

Homework01: 15-Puzzle

29 Sept. 2024

Sliding Puzzle Report

Overview:

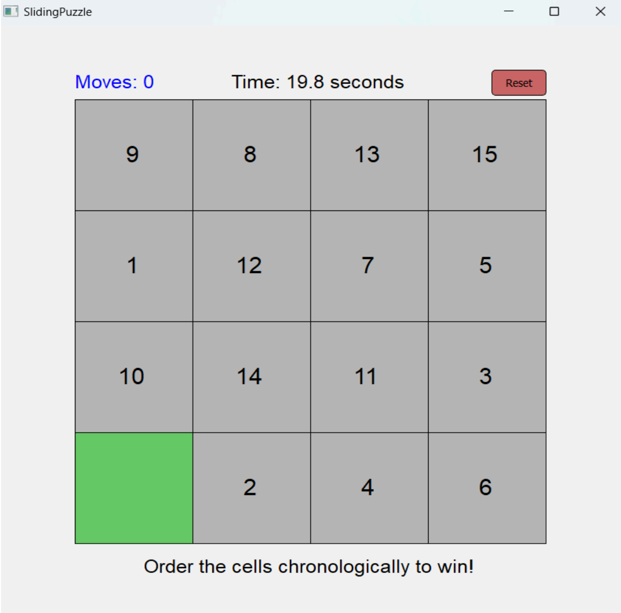
This program implements a sliding puzzle game called 15-puzzle using PyQt5 for the GUI. The puzzle consists of a 4x4 grid with numbered cells from 1 to 15 and one empty cell. The goal is to arrange the numbers sequentially from 1 to 15 (the last cell is empty) by sliding cells into the empty space.

Walkthrough:

The game board is a 4x4 2D list, with each position containing a number, where one position is 0, the empty space cell. During initialization, the program creates a random solvable arrangement by shuffling numbers and checking solvability using mathematical rules based on inversion counts. The game grid is drawn using a QPainter object in the paintEvent method. The numbers 1 to 15 are drawn in cells, and the empty cell is filled with a green color. It also displays the current number of moves the user has made and a timer that tracks the time elapsed in seconds and milliseconds, both located above the 4x4 grid. An additional label below the grid instructs the player to "Order the cells chronologically to win." Finally, a red “Reset” button is located above the grid that, upon click, reinitializes the board: new solvable position, reset moves, reset the timer.

The mousePressEvent method listens for user clicks within the game window. When the user clicks a cell, the program calculates the cell’s row and column, identifies the position of the empty cell, and if the clicked cell is in the same row or column as the empty cell, it slides the cells between the empty cell and the clicked cell into the empty cell's position. The program updates the move counter for every cell that moves upon a click, meaning up to three moves can happen with one click event. Clicks outside the board are ignored, and clicks on cells not in the same row or column as the empty cell are also ignored. After each move, the program checks if the board is in a winning configuration, and if so, it displays a grey overlay with a congratulations message, the user’s total moves, and the time took to complete the game. Clicking anywhere on the overlay will trigger an initialization event to restart the game for easy replayability.

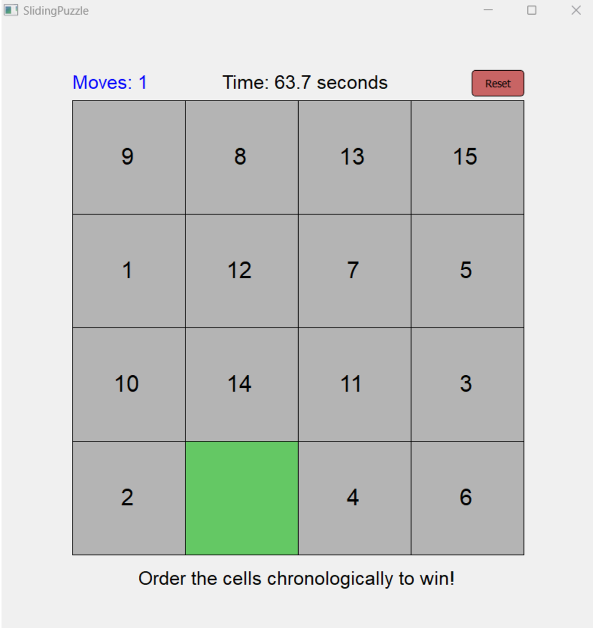
Gameplay:



Moves: 0 Time: 19.8 seconds

9	8	13	15
1	12	7	5
10	14	11	3
🟩	2	4	6

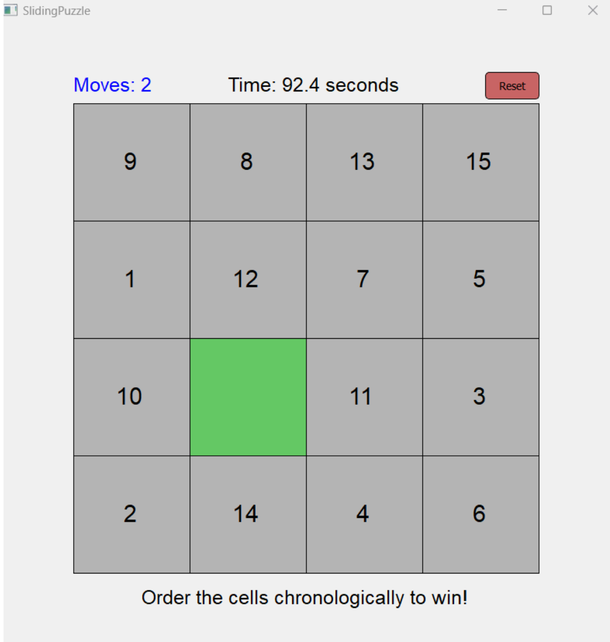
Order the cells chronologically to win!



Moves: 1 Time: 63.7 seconds

9	8	13	15
1	12	7	5
10	14	11	3
2	🟩	4	6

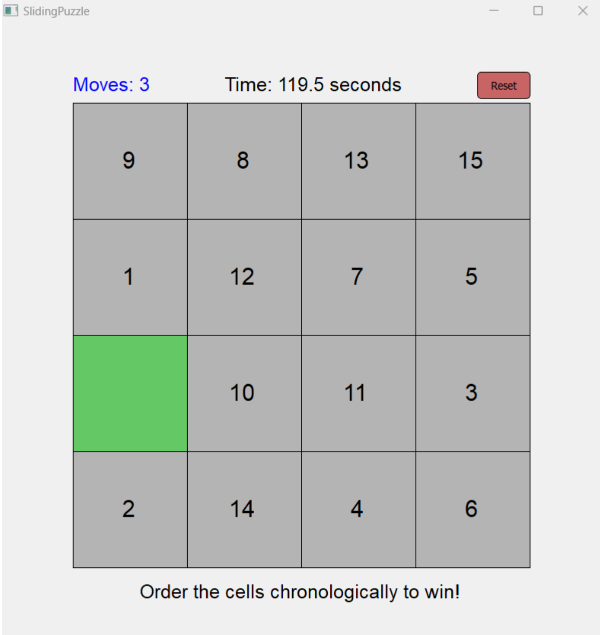
Order the cells chronologically to win!



Moves: 2 Time: 92.4 seconds

9	8	13	15
1	12	7	5
10	🟩	11	3
2	14	4	6

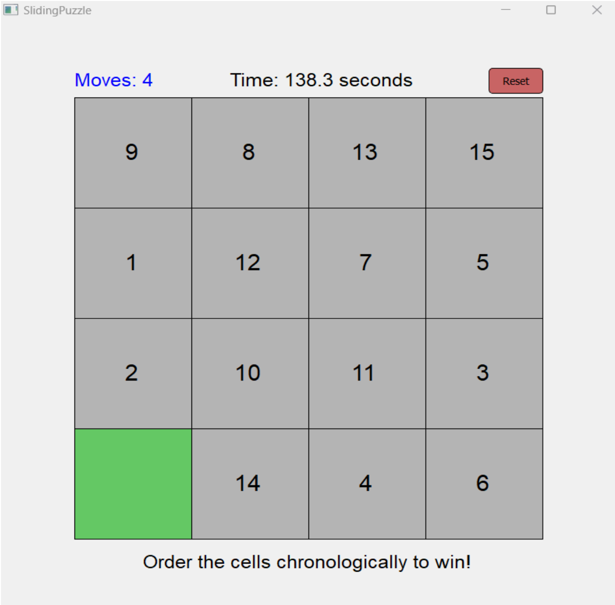
Order the cells chronologically to win!



Moves: 3 Time: 119.5 seconds

9	8	13	15
1	12	7	5
🟩	10	11	3
2	14	4	6

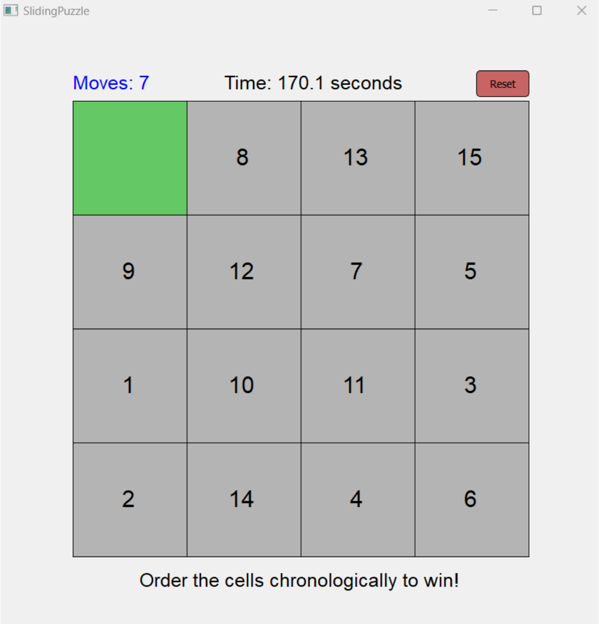
Order the cells chronologically to win!



Moves: 4 Time: 138.3 seconds

9	8	13	15
1	12	7	5
2	10	11	3
🟩	14	4	6

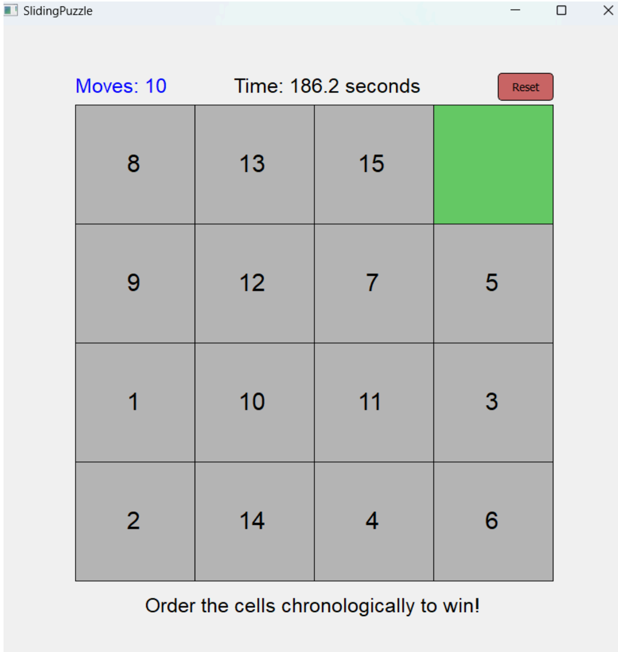
Order the cells chronologically to win!



Moves: 7 Time: 170.1 seconds

🟩	8	13	15
9	12	7	5
1	10	11	3
2	14	4	6

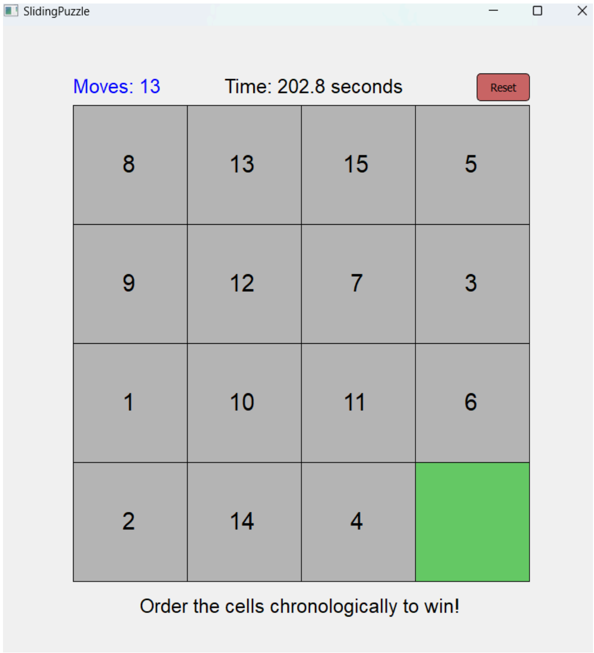
Order the cells chronologically to win!



Moves: 10 Time: 186.2 seconds

8	13	15	🟩
9	12	7	5
1	10	11	3
2	14	4	6

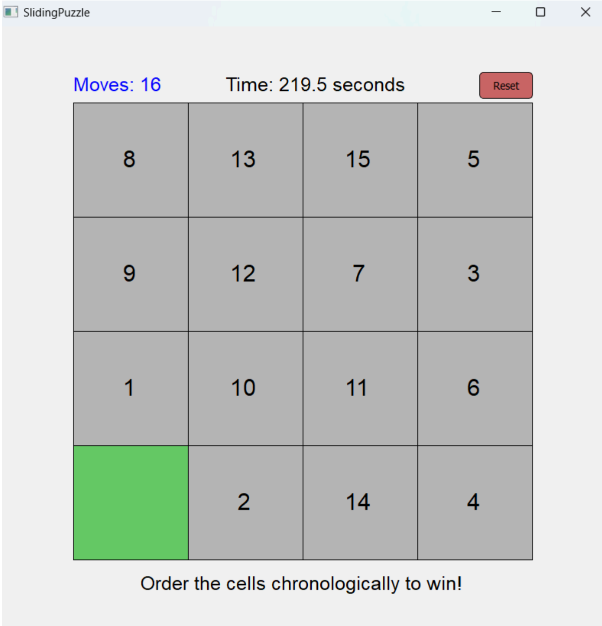
Order the cells chronologically to win!



Moves: 13 Time: 202.8 seconds

8	13	15	5
9	12	7	3
1	10	11	6
2	14	4	🟩

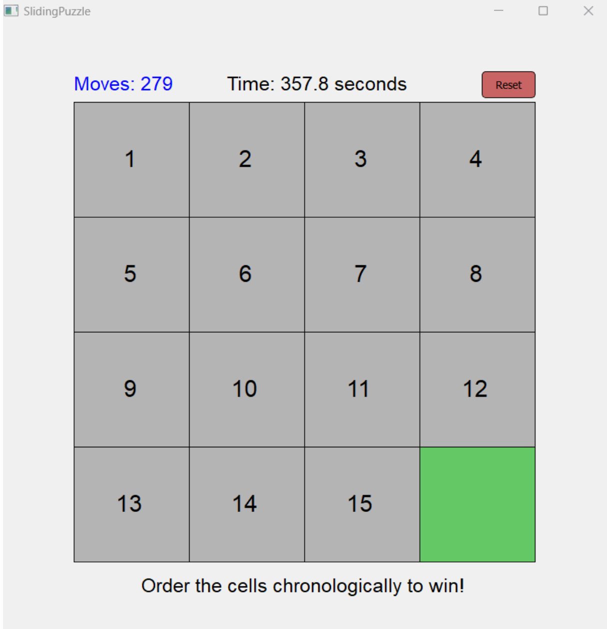
Order the cells chronologically to win!



Moves: 16 Time: 219.5 seconds

8	13	15	5
9	12	7	3
1	10	11	6
🟩	2	14	4

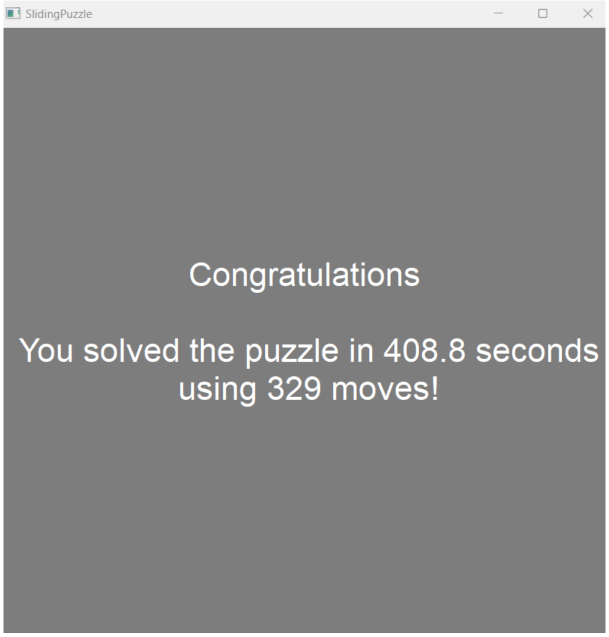
Order the cells chronologically to win!



Moves: 279 Time: 357.8 seconds

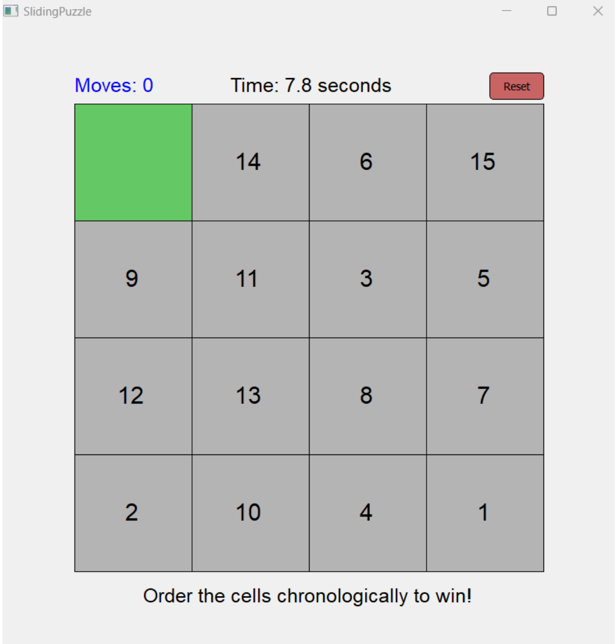
1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	🟩

Order the cells chronologically to win!



Congratulations

You solved the puzzle in 408.8 seconds using 329 moves!



Moves: 0 Time: 7.8 seconds

🟩	14	6	15
9	11	3	5
12	13	8	7
2	10	4	1

Order the cells chronologically to win!

*Note: I beat the game twice in this demonstration. The picture that shows the win state is not the normal behavior for my program. I designed my program so the grey overlay appears upon a win, and clicking the overlay generates a new game board, as shown above. For the first game completion, in which the winning state is on the board, I turned off the overlay functionality to demonstrate game solvability and game progression.

Code:

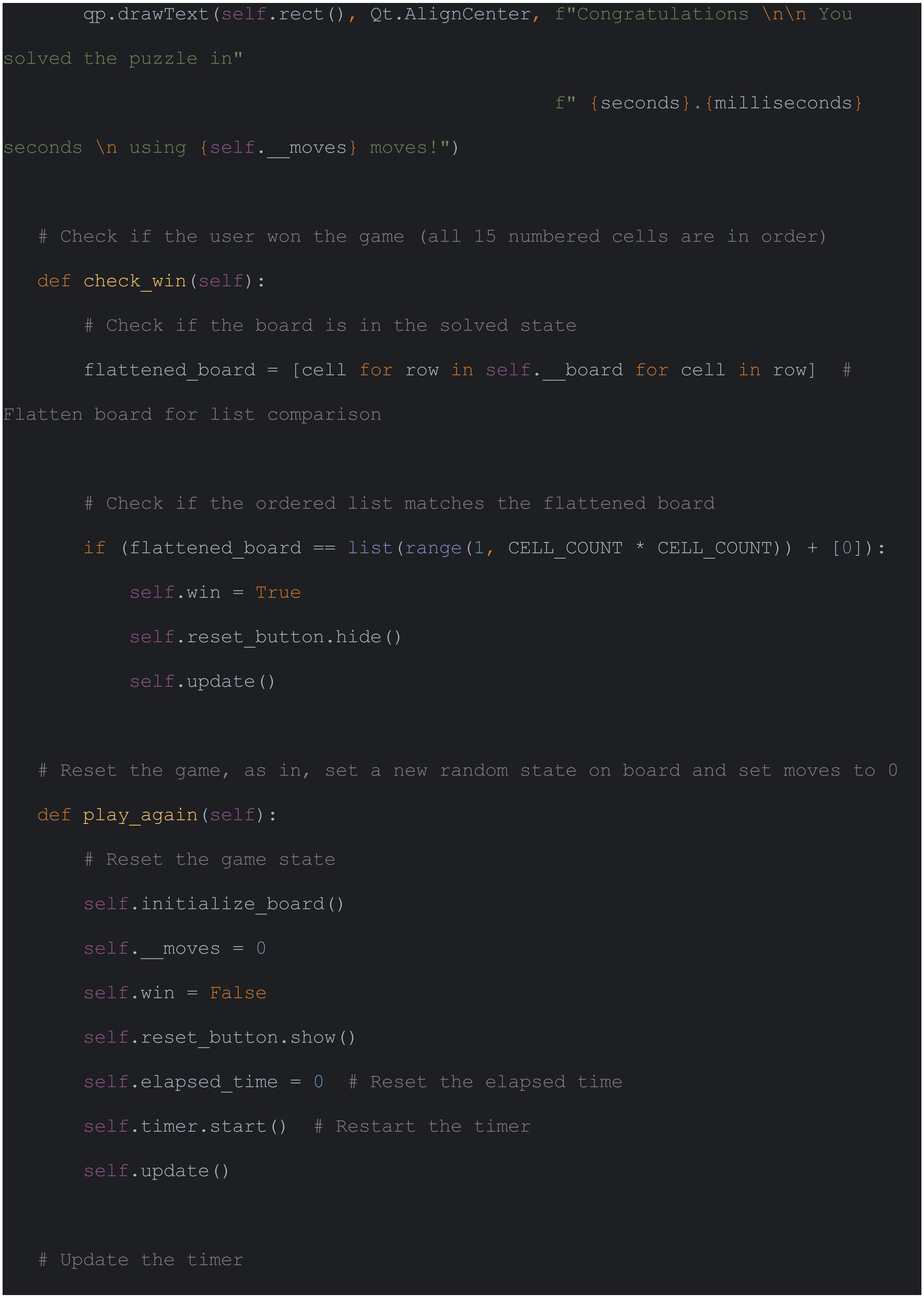
```python
#  Kyle Tranfaglia
#  COSC411 - Homework01
#  Last updated 09/23/24
#  This program uses PyQt5 packages to construct a game called 15-puzzle
import sys
import random
from PyQt5.QtGui import QPainter, QColor, QFont, QPen
from PyQt5.QtWidgets import QWidget, QApplication, QPushButton
from PyQt5.QtCore import Qt, QTimer

# Set game specifications: window size, cell/grid size, cell count, and grid starting location
CELL_COUNT = 4
CELL_SIZE = 150
GRID_ORIGINX = 100
GRID_ORIGINY = 100
W_WIDTH = 800
W_HEIGHT = 800


# Sliding puzzle object to handle all game setup and functionality
class SlidingPuzzle(QWidget):

   def __init__(self):
       super().__init__()
       self.setWindowTitle('SlidingPuzzle')
       self.setGeometry(300, 300, W_WIDTH, W_HEIGHT)
```

```
        self.__moves = 0
        self.win = False
        self.__board = [[-1 for _ in range(CELL_COUNT)] for _ in
range(CELL_COUNT)]
        self.__order = None
        self.initialize_board()

        # Setup timer to display seconds and milliseconds since the puzzle was
started
        self.timer = QTimer(self)
        self.timer.timeout.connect(self.update_time)
        self.elapsed_time = 0
        self.timer.start(100)  # Update every millisecond second

        # Create a reset button outside the grid (top-right corner)
        self.reset_button = QPushButton('Reset', self)
        self.reset_button.setStyleSheet("""QPushButton {background-color:
#cc6666; border: 1px solid black;
        border-radius: 5px; font-size: 14px;}""")
        self.reset_button.setGeometry(W_WIDTH - 169, GRID_ORIGINY - 40, 70, 35)
        self.reset_button.clicked.connect(self.play_again)
        self.show()

    # Initializes the board with random cell numbers if solvable
    def initialize_board(self):
        # Initialize and shuffle the cell order array
        self.__order = list(range(CELL_COUNT * CELL_COUNT))
        random.shuffle(self.__order)  # Initial shuffle
        # Shuffle the board order until the permutation is solvable
        while (not self.is_solvable()):
            random.shuffle(self.__order)

        # Populate the board with the shuffled numbers by row
        index = 0
        for r in range(CELL_COUNT):
            for c in range(CELL_COUNT):
                self.__board[r][c] = self.__order[index]
                index += 1

    # Check if the current board shuffle is solvable using a mathematical formula that counts inversions
    def is_solvable(self):
        inversions = self.count_inversions()  # Get number of inversions
        empty_row = CELL_COUNT - (self.__order.index(0) // CELL_COUNT)  # Find row from bottom with empty cell

        # Check grid width (compatible with future cell amount changes) and assess inversion rule
        if (CELL_COUNT % 2 == 1):  # Odd Width
            return inversions % 2 == 0
        else:  # Even Width
            if (empty_row % 2 == 1):  # Empty space on an odd row (counting from the bottom)
                return inversions % 2 == 0
            else:  # Empty space on an even row
                return inversions % 2 == 1
```

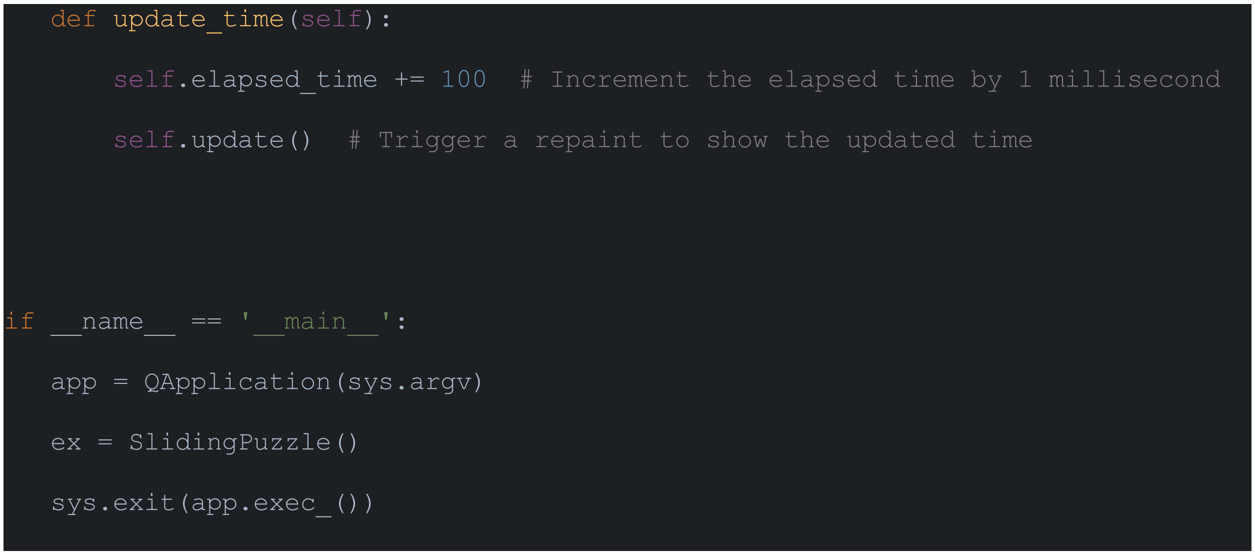
```
   # Count the number of inversions on the board (when a cell precedes another
cell with a smaller number on it)
   def count_inversions(self):
       inversions = 0
       # Loop through 1D ordered board and counting inversions (ignore empty
cell)
       for i in range(len(self.__order)):
           for j in range(i + 1, len(self.__order)):
               if self.__order[i] != 0 and self.__order[j] != 0 and
self.__order[i] > self.__order[j]:
                   inversions += 1
       return inversions

   # Creates and updates game grid
   def paintEvent(self, event):
       # Setup QPainter
       qp = QPainter()
       qp.begin(self)

       # Fill the entire grid region with light grey color
       grid_width = grid_height = CELL_COUNT * CELL_SIZE
       qp.fillRect(GRID_ORIGINX, GRID_ORIGINY, grid_width, grid_height,
QColor(180, 180, 180))

       # Set text font and color, then draw the move counter above the top-left
corner of the grid
       qp.setFont(QFont('Arial', 15))
       qp.setPen(QPen(Qt.blue))
       qp.drawText(GRID_ORIGINX, GRID_ORIGINY - 15, f"Moves: {self.__moves}")
```

```
        qp.setPen(QPen(Qt.black))

        # Convert milliseconds to seconds and milliseconds
        seconds = self.elapsed_time // 1000
        milliseconds = (self.elapsed_time % 1000) // 100

        # Draw the timer above the grid
        qp.drawText(GRID_ORIGINX + 200, GRID_ORIGINY - 15, f"Time: {seconds}.{milliseconds} seconds")
        qp.setFont(QFont('Arial', 18))

        # Loop through 2D board array and draw the board with numerically labeled cells
        for r in range(len(self.__board)):
            for c in range(len(self.__board[r])):
                number = self.__board[r][c]  # Get the number at the current position

                # Draw the number if it's not 0 (0 represents the empty tile)
                if (number != 0):
                    # Calculate cell center (x,y)
                    text_x = GRID_ORIGINX + c * CELL_SIZE + CELL_SIZE // 2 - 10  # Center horizontally
                    text_y = GRID_ORIGINY + r * CELL_SIZE + CELL_SIZE // 2 + 10  # Center vertically

                    # Adjust x coordinate to center for double-digit numbers
                    if (number / 10 >= 1):
                        text_x -= 10  # Shift text to the left for double digits
                    qp.drawText(text_x, text_y, str(number))  # Draw the number centered in the cell
                else:
                    # Fill the empty block with a green color
                    qp.fillRect(GRID_ORIGINX + c * CELL_SIZE, GRID_ORIGINY + r * CELL_SIZE, CELL_SIZE, CELL_SIZE,
                                QColor(102, 204, 102))

                # Draw the cell border
                qp.drawRect(GRID_ORIGINX + c * CELL_SIZE, GRID_ORIGINY + r * CELL_SIZE, CELL_SIZE, CELL_SIZE)

        # Draw the instructional text below the board
        qp.setFont(QFont('Arial', 15))
        qp.drawText(GRID_ORIGINX + 88, GRID_ORIGINY + grid_height + 40, "Order the cells chronologically to win!")

        # If the user wins, show the overlay
        if (self.win):
            self.timer.stop()
            self.draw_overlay(qp, seconds, milliseconds)
        qp.end()

    # Handle mouse click event
    def mousePressEvent(self, event):
        # If user won, reset the game on click
        if (self.win):
            self.play_again()
            return

        # Calculate row and column of mouse click
        row = (event.y() - GRID_ORIGINY) // CELL_SIZE
        col = (event.x() - GRID_ORIGINX) // CELL_SIZE

        # Check that the row and column are within the CELL_COUNT * CELL_COUNT grid
        if (0 <= row < CELL_COUNT and 0 <= col < CELL_COUNT):
            empty_row, empty_col = self.find_empty_cell()   # Find the position of the empty cell (denoted by 0)

            # Check if cell is in the same row as the empty cell
            if (row == empty_row):
                move_count = abs(col - empty_col)   # Number of cells to move
                if (col < empty_col):   # Slide right
                    for i in range(empty_col, col, -1):
                        self.__board[row][i] = self.__board[row][i - 1]
                elif (col > empty_col):   # Slide left
                    for i in range(empty_col, col):
                        self.__board[row][i] = self.__board[row][i + 1]

            # Check if cell is in the same column as the empty cell
            elif (col == empty_col):
                move_count = abs(row - empty_row)   # Number of cells to move
                if (row < empty_row):   # Slide down
                    for i in range(empty_row, row, -1):
                        self.__board[i][col] = self.__board[i - 1][col]
                elif (row > empty_row):   # Slide up
```

```
                    for i in range(empty_row, row):
                        self.__board[i][col] = self.__board[i + 1][col]

            # Clicked cell is not swappable
            else:
                return

            self.__board[row][col] = 0  # place empty cell at the clicked
position
            self.__moves += move_count  # Update move count by the number of
cells that moved
            self.check_win()  # Check if user won
            self.update()

    # Return location (row, column) of the empty cell
    def find_empty_cell(self):
        # Find the row and column of the empty cell (denoted by 0)
        for r in range(CELL_COUNT):
            for c in range(CELL_COUNT):
                if self.__board[r][c] == 0:
                    return r, c

    # Draw the victory screen to let the user know they won
    def draw_overlay(self, qp, seconds, milliseconds):
        qp.fillRect(self.rect(), QColor(128, 128, 128))  # Draw the grey overlay

        # Display win message
        qp.setPen(QPen(Qt.white))
        qp.setFont(QFont('Arial', 26))
```



```
        qp.drawText(self.rect(), Qt.AlignCenter, f"Congratulations \n\n You
solved the puzzle in"
                                                f" {seconds}.{milliseconds}
seconds \n using {self.__moves} moves!")

    # Check if the user won the game (all 15 numbered cells are in order)
    def check_win(self):
        # Check if the board is in the solved state
        flattened_board = [cell for row in self.__board for cell in row]  #
Flatten board for list comparison

        # Check if the ordered list matches the flattened board
        if (flattened_board == list(range(1, CELL_COUNT * CELL_COUNT)) + [0]):
            self.win = True
            self.reset_button.hide()
            self.update()

    # Reset the game, as in, set a new random state on board and set moves to 0
    def play_again(self):
        # Reset the game state
        self.initialize_board()
        self.__moves = 0
        self.win = False
        self.reset_button.show()
        self.elapsed_time = 0  # Reset the elapsed time
        self.timer.start()  # Restart the timer
        self.update()

    # Update the timer
```



```
    def update_time(self):
        self.elapsed_time += 100  # Increment the elapsed time by 1 millisecond
        self.update()  # Trigger a repaint to show the updated time



if __name__ == '__main__':
    app = QApplication(sys.argv)
    ex = SlidingPuzzle()
    sys.exit(app.exec_())
```

References:

https://doc.qt.io

Classmate Discussions:

Cole Barbes: brief discussion to verify the logic of finding the row containing the empty cell from the bottom up

Timmy Mckirgan: Provided the idea to add a timer