Rolf Genree L. Garces
CMSC 122

December 13, 2024

Final Project Requirement 2

Project Presentation

This document contains the algorithm of the data structures used in my project, 'ColorMatch Hue Puzzle'. Additionally, the report on the time and space complexity of the algorithm is also discussed here.

Game Overview

"ColorMatch Hue Puzzle" features **dynamic grid challenges** with varied grid sizes and randomized tile placements at each level. Each level also includes some **immovable, correctly placed tiles** to guide players. Unlike similar games, ColorMatch Hue Puzzle offers a hint system with optional, limited clues and the option to **view the correct tile arrangement** for additional support. The game also **tracks the user's moves**, encouraging strategic planning to minimize steps.

This project is all about tackling a color-matching puzzle grid. Users can move tiles around to match a specific pattern. An algorithm is utilized that efficiently and systematically solves the puzzle, from the latest configuration made by the user. This report dives into the main features of the algorithm, the data structures used, and the space-time analysis of the algorithms.

Algorithm Description

The algorithm is designed to iteratively solve the puzzle grid, ensuring minimal computational overhead while maintaining accuracy. The solution process involves the following steps:

1. **Identifying Misplaced Tiles:**
    - The algorithm starts by scanning the current grid configuration to identify all movable tiles that are not in their correct positions.
    - A list, misplaced_tiles, stores the coordinates of these tiles. Simultaneously, a set, locked_tiles, tracks tiles that have already been placed correctly to ensure they are not swapped again.

```
def solve_puzzle(self):  1 usage
    #Solves the puzzle step-by-step starting from the current configuration.
    self.clicked_hint()
    correct_grid = self.levels.check_level(self.selected_level - 1)
    locked_tiles = set()  # Keeps track of tiles that are correctly placed

    def find_misplaced_tiles():
        #Identifies all misplaced movable tiles.
        misplaced = []
        for row in range(self.rows):
            for col in range(self.cols):...
        return misplaced
```

2. **Finding the Correct Position:**
    - For each misplaced tile, the algorithm determines the correct position by referencing the predefined solution grid.
    - It uses nested loops to locate a tile of the same color that can be swapped into the correct position.

```
def find_correct_position(tile_position):
    #Finds the correct position for the given tile.
    row, col = tile_position
    correct_color = correct_grid[row][col]
    for r in range(self.rows):
        for c in range(self.cols):
            if (
                    self.level_data[r][c]["movable"] == 1
                    and self.level_data[r][c]["color"] == correct_color
                    and (r, c) not in locked_tiles
            ):
                return r, c
    return None
```

3. **Swapping Tiles:**
    - Once the correct position is identified, the algorithm swaps the colors of the two tiles.
    - After the swap, the algorithm checks if the swapped tiles are now correctly placed. If so, their coordinates are added to the locked_tiles set, and their buttons are disabled in the user interface to prevent further interaction.

```
def perform_swap():
    #Performs a swap for the next misplaced tile.
    #Validates grid to check if it's already solved
    if self.validate_grid(self.selected_level - 1):
        return

    misplaced_tiles = find_misplaced_tiles()
    if not misplaced_tiles:
        # If all tiles are correctly placed; stop solving
        return

    # Picks the first misplaced tile and find its correct position
    tile_to_correct = misplaced_tiles[0]
    correct_position = find_correct_position(tile_to_correct)

    if correct_position:
        # Performs the swap and updates the move count
        self.swap_tiles(tile_to_correct, correct_position)
        self.moves_count+=1
        self.moves.config(text=str(self.moves_count))
```

```
        # Updates locked tiles
        row1, col1 = tile_to_correct
        row2, col2 = correct_position
        if self.level_data[row1][col1]["color"] == correct_grid[row1][col1]:
            locked_tiles.add((row1, col1))
            self.buttons[(row1, col1)].config(state="disabled")
        if self.level_data[row2][col2]["color"] == correct_grid[row2][col2]:
            locked_tiles.add((row2, col2))
            self.buttons[(row2, col2)].config(state="disabled")

    # Continues solving after a short delay
    self.after( ms: 300, perform_swap)

# Start solving
perform_swap()
```

4. **Validation and Iterative Solving:**
    - The grid is validated after every swap by comparing the current grid configuration to the correct solution grid.
    - This process repeats until all tiles are correctly placed.
5. **Integration with User Interaction:**
    - The algorithm seamlessly integrates with the user interface by dynamically updating the puzzle state after each user interaction.
    - It ensures that any user modifications are incorporated before the solving process begins."

Key Data Structure

1. **2D List (`level_data`)**
    - Represents the puzzle grid, where each cell contains a dictionary with the following attributes:

        - `movable`: Boolean indicating if the tile can be moved.
        - `color`: Current color of the tile.
2. **List (`misplaced_tiles`)**
    - Tracks the coordinates of all tiles that are misplaced and require swapping.
3. **Set (`locked_tiles`)**
    - Maintains the coordinates of tiles that are already in their correct positions to prevent redundant operations.
4. **Dictionary (`buttons`)**
    - Maps each tile's coordinates to its respective button in the graphical interface for real-time updates.

Time and Space Complexity

Time Complexity

1. **Finding Misplaced Tiles**:
    - The algorithm traverses the entire grid to identify misplaced tiles, which takes $O(n \times m)$, where n is the number of rows and m is the number of columns.

2. **Finding the Correct Position**:
    - For each misplaced tile, the algorithm iterates through the grid to find its correct position. This operation takes $O(n \times m)$ per tile. In the worst case, if there are k misplaced tiles, the time complexity is $O(k \times n \times m)$.
3. **Swapping Tiles and Validation**:
    - Each swap and validation operation takes $O(1)$ and $O(n \times m)$, respectively.
4. **Overall Time Complexity**:
    - The total complexity is dominated by the tile correction process, resulting in $O(k \times n \times m)$, where k is the number of misplaced tiles.

Space Complexity

1. **Grid Representation**:
    - The grid is stored as a 2D list, consuming $O(n \times m)$ space.
2. **Auxiliary Data Structures**:
    - `misplaced_tiles`: Uses $O(k)$ space, where k is the number of misplaced tiles.
    - `locked_tiles`: Uses $O(k)$ space for tracking correctly placed tiles.
3. **Overall Space Complexity**:
    - Dominated by the grid representation, resulting in $O(n \times m)$.

Conclusion

This algorithm effectively solves the puzzle by systematically identifying and correcting misplaced tiles. The use of efficient data structures ensures that the program is robust, responsive, and seamlessly integrated with the graphical interface. By implementing real-time validation and locking mechanisms, the algorithm adheres to the requirement of solving the puzzle from the user's latest configuration. The program demonstrates an effective solution strategy with reasonable time and space complexity, making it highly suitable for small to medium-sized grids.

Acknowledgement

The development of the algorithm was guided by the help of AIs namely: ChatGPT, Claude AI and were thoroughly analyzed and compared to satisfy the requirements of this project. This allowed the developer to have a rigorous analysis of the problem requirement and ensure a comprehensive and optimized solution to the program.