Implementation of Data Structures and Algorithms in 2048 Game Project

Tran Khanh Tai
Student ID: ITITIU21300
Advisor: Tran Thanh Tung
International University - VNU HCMC

May 27, 2025

Abstract

This report presents a comprehensive analysis of data structures and algorithms implemented in the 2048 game project. It focuses on the design decisions, implementation details, and performance analysis of various algorithms used in the game. The report highlights how object-oriented programming principles are applied to create an efficient and maintainable codebase for the popular sliding block puzzle game.

Contents

1 Introduction

The 2048 game is a popular single-player sliding tile puzzle game developed by Gabriele Cirulli in March 2014. The game involves sliding numbered tiles on a grid to combine them, with the goal of creating a tile with the number 2048. This project implements the game using Java, focusing on applying data structures and algorithms concepts to create an efficient and functional game.

This report details the implementation of the 2048 game with specific emphasis on the data structures used, algorithms implemented, and their complexity analysis. It also explores how object-oriented programming principles have been applied to make the code modular, reusable, and maintainable.

2 Importance of Data Structures and Algorithms (OOP)

Data structures and algorithms form the foundation of efficient programming. In game development:

- **Data Structures** provide organized ways to store and access data, which is critical for representing the game board and state.
- **Algorithms** enable efficient manipulation of these data structures, allowing for game logic implementation.
- **Object-Oriented Programming (OOP)** facilitates modularity, encapsulation, and code reuse through class hierarchies.

The 2048 game benefits from OOP principles through:

- **Encapsulation** - hiding implementation details within classes
- **Inheritance** - extending functionality through class hierarchies
- **Polymorphism** - allowing for flexible behavior implementation
- **Abstraction** - simplifying complex systems into manageable components

3 Purpose of the Project

The main objectives of this project include:

- Implementing a fully functional 2048 game with a graphical user interface
- Demonstrating practical application of data structures and algorithms
- Applying object-oriented design principles to create maintainable code
- Analyzing algorithm efficiency and performance in a real-world application
- Developing problem-solving skills through game logic implementation

4 Application of DSA Principles in the Game

The 2048 game implementation incorporates several key DSA principles:

4.1 Data Structures

- **2D Arrays** - Used to represent the game board grid
- **ArrayLists** - Used for managing dynamic collections of game elements
- **Classes and Objects** - Used to encapsulate game components
- **Queue** - Used in the animation system for sequential processing

4.2 Algorithms

- **Merge Algorithm** - For combining tiles with the same value
- **Random Tile Generation** - For placing new tiles on the board
- **Win/Loss Detection** - For determining game state
- **Tile Movement Algorithms** - For handling directional moves

5 Properties of the 2048 Game

5.1 Goal of the Game

The primary objective of the 2048 game is to slide numbered tiles on a 4×4 grid to combine them and create a tile with the number 2048. The player can continue beyond this goal to achieve higher-numbered tiles and a higher score.

5.2 Rules of the Game

- The game starts with two randomly placed tiles (either 2 or 4) on a 4×4 grid
- The player can slide tiles in four directions: up, down, left, and right
- When two tiles with the same number touch during a move, they merge into one tile with the sum of their values
- After each move, a new tile (either 2 or 4) appears at a random empty position
- The game ends when no valid moves are possible (board is full with no possible merges)
- The player wins when a tile with the value 2048 appears on the board

6 Methodology

6.1 Overview of Classes Used

The game implementation follows an object-oriented approach with several key classes:

- **Game** - The main class that controls game flow
- **Board** - Represents the game board and its state
- **Tile** - Represents individual tiles with their values
- **GamePanel** - Handles the visual representation and user input
- **Direction** - Enum representing the four possible move directions
- **ScoreManager** - Tracks and updates the game score
- **Animation** - Handles smooth visual transitions

6.2 Class Dependencies and Relationships

- The **Game** class contains a **Board** object
- The **Board** class contains a 2D array of **Tile** objects
- The **GamePanel** class renders the **Board** and **Tile** objects
- The **ScoreManager** is updated by the **Game** class when tiles merge
- The **Animation** class works with the **GamePanel** to provide visual feedback

6.3 Main Classes and Functionality

6.3.1 Board Class

This class represents the game board as a 2D array of tiles. Sample implementation:

```java
public class Board {
    private Tile[][] grid;
    private final int SIZE = 4;

    public Board() {
        grid = new Tile[SIZE][SIZE];
        // Initialize empty board
        for (int row = 0; row < SIZE; row++) {
            for (int col = 0; col < SIZE; col++) {
                grid[row][col] = null;
            }
        }
        addRandomTile(); // Add initial tiles
        addRandomTile();
    }

    // More methods...
}
```

Listing 1: Board Class Implementation

The **Board** class has the following time complexities:

- Initialization: O(n^2) where n is the board size (4×4)
- Checking for available moves: O(n^2)
- Board state checking: O(n^2)

6.3.2 Tile Class

Represents individual tiles on the board with values and positions:

```java
public class Tile {
    private int value;
    private int row;
    private int col;
    private boolean merged;

    public Tile(int value, int row, int col) {
        this.value = value;
        this.row = row;
        this.col = col;
        this.merged = false;
    }

    // Getters and setters
}
```

Listing 2: Tile Class Implementation

7 Algorithms

7.1 Move and Merge Algorithm

The most complex algorithm in the game is the tile movement and merging logic:

```java
public boolean move(Direction dir) {
    boolean moved = false;
    resetMergeStatus();

    switch (dir) {
        case UP:
            for (int col = 0; col < SIZE; col++) {
                for (int row = 1; row < SIZE; row++) {
                    if (grid[row][col] != null) {
                        moved |= moveUp(row, col);
                    }
                }
            }
            break;
        // Cases for DOWN, LEFT, RIGHT similarly implemented
    }

    if (moved) {
        addRandomTile();
        checkGameOver();
    }

    return moved;
}

private boolean moveUp(int row, int col) {
    boolean moved = false;
    Tile tile = grid[row][col];
    int newRow = row;

    // Move up as far as possible
    while (newRow > 0 && grid[newRow-1][col] == null) {
        newRow--;
    }

    // Check for merge
    if (newRow > 0 && grid[newRow-1][col] != null &&
        grid[newRow-1][col].getValue() == tile.getValue() &&
        !grid[newRow-1][col].isMerged()) {

        // Merge with the tile above
        int mergedValue = tile.getValue() * 2;
        grid[newRow-1][col].setValue(mergedValue);
        grid[newRow-1][col].setMerged(true);
        grid[row][col] = null;
        score += mergedValue;
        moved = true;
    }
    // Just move without merging
    else if (newRow != row) {
        grid[newRow][col] = tile;
        grid[row][col] = null;
        moved = true;
    }

    return moved;
```

```
}
```

Listing 3: Move and Merge Algorithm

Time Complexity Analysis:

- Move operation: O(n^2) where n is the board size
- For each direction, we need to check each cell and potentially move it
- The while loop for moving tiles runs at most O(n) times
- Overall complexity is O(n^3) in the worst case

7.2 Random Tile Generation

After each move, a new tile needs to be added to a random empty cell:

```java
private void addRandomTile() {
    List<int[]> emptyCells = new ArrayList<>();

    // Find all empty cells
    for (int row = 0; row < SIZE; row++) {
        for (int col = 0; col < SIZE; col++) {
            if (grid[row][col] == null) {
                emptyCells.add(new int[]{row, col});
            }
        }
    }

    if (!emptyCells.isEmpty()) {
        int index = (int)(Math.random() * emptyCells.size());
        int[] position = emptyCells.get(index);
        int value = Math.random() < 0.9 ? 2 : 4; // 90% chance for a 2, 10% for a 4

        grid[position[0]][position[1]] = new Tile(value, position[0], position[1]);
    }
}
```

Listing 4: Random Tile Generation

Time Complexity Analysis:

- Finding empty cells: O(n^2)
- Selecting a random cell: O(1)
- Overall complexity is O(n^2)

7.3 Game Over Detection

The game ends when there are no valid moves left:

```java
private boolean isGameOver() {
    // Check for empty cells
    for (int row = 0; row < SIZE; row++) {
        for (int col = 0; col < SIZE; col++) {
            if (grid[row][col] == null) {
                return false; // Game continues if there's an
                   empty cell
            }
        }
    }

    // Check for possible merges horizontally
    for (int row = 0; row < SIZE; row++) {
        for (int col = 0; col < SIZE - 1; col++) {
            if (grid[row][col].getValue() == grid[row][col+1].
               getValue()) {
                return false; // Game continues if a horizontal
                   merge is possible
            }
        }
    }

    // Check for possible merges vertically
    for (int col = 0; col < SIZE; col++) {
        for (int row = 0; row < SIZE - 1; row++) {
            if (grid[row][col].getValue() == grid[row+1][col].
               getValue()) {
                return false; // Game continues if a vertical
                   merge is possible
            }
        }
    }

    return true; // Game over if no empty cells and no possible
       merges
}
```

Listing 5: Game Over Detection

Time Complexity Analysis:

- Checking for empty cells: $O(n^2)$
- Checking for horizontal merges: $O(n^2)$
- Checking for vertical merges: $O(n^2)$
- Overall complexity is $O(n^2)$

7.4 Additional Behaviors and Image Handling

7.4.1 Centralized Image Loading

The game uses a resource manager to efficiently load and cache images:

```java
public class ResourceManager {
    private static Map<String, BufferedImage> images = new
       HashMap<>();

    public static BufferedImage loadImage(String path) {
        if (images.containsKey(path)) {
            return images.get(path);
        }

        try {
            BufferedImage image = ImageIO.read(new File(path));
            images.put(path, image);
            return image;
        } catch (IOException e) {
            e.printStackTrace();
            return null;
        }
    }
}
```

Listing 6: Image Resource Manager

This implementation uses a HashMap for O(1) image lookup, preventing redundant loading of the same resources.

7.5 GamePanel Class

The GamePanel class handles rendering and user interaction:

```java
public class GamePanel extends JPanel {
    private Board board;
    private Map<Integer, Color> tileColors;

    public GamePanel(Board board) {
        this.board = board;
        initializeColors();
        setFocusable(true);
        addKeyListener(new GameKeyAdapter());
    }

    @Override
    protected void paintComponent(Graphics g) {
        super.paintComponent(g);
        Graphics2D g2d = (Graphics2D) g;

        // Draw background
        g2d.setColor(new Color(187, 173, 160));
        g2d.fillRect(0, 0, getWidth(), getHeight());

        // Draw tiles
        for (int row = 0; row < 4; row++) {
            for (int col = 0; col < 4; col++) {
                drawTile(g2d, board.getTile(row, col), row, col);
            }
        }

        // Draw score and game over message if needed
        drawUI(g2d);
    }

    private void drawTile(Graphics2D g2d, Tile tile, int row, int
        col) {
        // Tile drawing implementation
    }
}
```

Listing 7: GamePanel Rendering

7.6 Key Controls

7.6.1 Handling User Input and Key Events

The game responds to arrow key inputs to control tile movement:

```java
private class GameKeyAdapter extends KeyAdapter {
    @Override
    public void keyPressed(KeyEvent e) {
        boolean moved = false;

        switch (e.getKeyCode()) {
            case KeyEvent.VK_UP:
                moved = board.move(Direction.UP);
                break;
            case KeyEvent.VK_DOWN:
                moved = board.move(Direction.DOWN);
                break;
            case KeyEvent.VK_LEFT:
                moved = board.move(Direction.LEFT);
                break;
            case KeyEvent.VK_RIGHT:
                moved = board.move(Direction.RIGHT);
                break;
        }

        if (moved) {
            repaint();
        }
    }
}
```
Listing 8: Key Event Handling

7.6.2 Directional Control and Image Updates

Each key press triggers a move in the corresponding direction, which may update the game state and cause the UI to refresh if tiles were moved or merged.

8 Results, Limitations, and Conclusion

The implementation of the 2048 game demonstrates practical application of data structures and algorithms in a real-world scenario. The key results include:

- A functional game with intuitive controls and visual feedback
- Efficient algorithms for tile movement, merging, and game state management
- Object-oriented design providing modularity and code reuse

8.1 Limitations

- The current implementation has O(n^3) complexity for moves in the worst case
- No undo functionality or game state saving
- Limited animation capabilities

8.2 Conclusion

This project successfully demonstrates the application of data structures and algorithms concepts in game development. The implementation of the 2048 game shows how careful algorithm design and appropriate data structure selection contribute to creating an efficient and enjoyable game experience.

Future improvements could include optimizing the move algorithm for better performance, adding game state saving functionality, and implementing more sophisticated animations.

9 GitHub Repository

The source code for this project is available at: `https://github.com/KhanhTaiTran/DSA-Project.git`