Minesweeper Game

**Project Report**

**-Abstract:**

This project implements a text-based version of the popular Minesweeper game, developed using Java. The game allows players to select difficulty levels, revealing cells, flagging mines, and utilizing hints to uncover safe areas while avoiding mines. The program integrates key concepts from discrete mathematics, including logic, set theory, combinatorics, and graph theory. The grid-based board is represented using 2D arrays, and a recursive flood fill algorithm is employed to reveal adjacent cells with no nearby mines. The game also features a mine placement algorithm based on randomization and checks for victory conditions when all safe cells are revealed. This project serves to demonstrate the application of mathematical reasoning and algorithms in the development of a classic puzzle game, providing both an engaging user experience and a practical example of discrete mathematical principles.

**-Introduction:**

Minesweeper is a well-known single-player puzzle game where the objective is to clear a grid of squares without detonating any hidden mines. The player is provided with a grid of cells, some of which contain mines, while others indicate the number of adjacent mines. The game involves strategic thinking, where the player must make decisions based on the numbers revealed and avoid triggering mines.

This project implements a text-based version of Minesweeper using Java, offering users a choice of three difficulty levels: Easy, Medium, and Hard. The program features a grid-based board represented using 2D arrays, where mines are randomly placed, and adjacent cells are evaluated for the number of mines surrounding them. Players interact with the game by revealing cells, flagging suspected mines, and requesting hints to aid their progress.

The implementation leverages fundamental discrete mathematical concepts such as combinatorics, logic, set theory, and graph theory. Recursive algorithms are used to reveal cells with no adjacent mines, and victory conditions are checked by verifying if all non-mined cells are revealed. This project demonstrates the practical application of these mathematical concepts in solving complex problems, enhancing both problem-solving skills and understanding of algorithms.

**-Objectives:**

The main objectives of this Minesweeper project are as follows:

**-**Game Implementation: To implement a fully functional text-based Minesweeper game using Java, allowing users to play interactively with different difficulty levels.

**-**Mathematical Logic Integration: To apply discrete mathematical concepts such as combinations & permutations, set theory, and recursive algorithms in the design and development of the game. This includes the logic for placing mines, calculating adjacent mines, and revealing safe cells.

-User Interactivity: To design an intuitive and user-friendly interface that allows players to input their moves, flag cells, and request hints. This involves validating user input and providing clear feedback for each action.

-Efficiency and Optimization: To ensure the program performs efficiently with proper handling of the game board, especially with larger grids, through the effective use of arrays and recursion.

-Game Flow Management: To implement a smooth game flow that checks for winning conditions, handles the game-over scenarios, and provides the user with appropriate notifications upon game completion.

-Algorithm Design: To design algorithms that handle key operations such as mine placement, number calculation for adjacent cells, cell revealing, and flagging. Recursive techniques are used for automatic revealing of safe cells.

By achieving these objectives, this project aims to combine game development with discrete mathematics to create an engaging and educational experience while demonstrating the practical applications of these concepts in computer science.

**-System Design:**

The system design for the Minesweeper game focuses on creating a structured, modular, and efficient program that can handle various aspects of the game, including board setup, game logic, user interactions, and game flow management. Below is a detailed explanation of the system design components:

**1. Overview of the System**

The Minesweeper game is designed as a console-based application written in Java. The user interacts with the game through a simple text interface, where they can choose difficulty levels, make moves (reveal or flag cells), and receive hints. The game progresses with recursive algorithms to reveal safe cells and checks for winning or losing conditions.

**2. Key Components**

The system is divided into several key components, each responsible for specific tasks:

**-Main Menu and User Interface**:

* 1. The main menu presents users with options to start a new game or quit the game.
  2. After selecting the difficulty level, the game board is initialized, and players can input their actions such as revealing or flagging cells.
  3. The program handles user input through the console, validating commands and providing feedback to the user.

**-Game Initialization and Setup**:

* 1. The game setup involves initializing the game board based on the selected difficulty (Easy, Medium, or Hard).
  2. The game board is represented as a 2D array of characters for visual representation and a 2D array of integers for managing the minefield.
  3. A recursive algorithm is used to calculate the number of adjacent mines for each cell.

**-Minefield Generation**:

* 1. The program places mines randomly on the game board by selecting random coordinates until the desired number of mines is placed.
  2. After mines are placed, the adjacent cells are checked, and the number of surrounding mines is calculated for each non-mine cell.

**-Game Play Logic**:

* 1. The core gameplay logic allows players to take actions like revealing or flagging cells.
  2. When a cell is revealed, it is checked for a mine. If no mine is found, the adjacent cells are recursively revealed until safe areas are uncovered.
  3. If a player flags a cell, it is marked with a flag icon.
  4. The game continuously checks for win or loss conditions by ensuring all non-mine cells are revealed or if a mine is triggered.

**-Hint System**:

* 1. A hint system is available that automatically reveals a safe, unrevealed cell. This provides assistance to players who need help progressing through the game.

**-Game Termination**:

* 1. The game terminates when all safe cells are revealed (win condition) or when a mine is triggered (loss condition). In either case, the game board is revealed, and the results (win/loss) are displayed to the player.

#### ****3. Data Structures Used****

**-2D Arrays**:

* + A 2D array board of type char[][] is used to represent the game board visually, where each cell is either hidden ('-'), flagged ('F'), or revealed with the number of adjacent mines.
  + A 2D array minefield of type int[][] holds the minefield data, where each cell is either a number indicating the count of adjacent mines or -1 indicating a mine.
  + A 2D array revealed of type boolean[][] keeps track of whether a cell has been revealed or not.

**-Game State Variables**:

* + SIZE represents the dimensions of the game board (rows and columns).
  + MINES represents the total number of mines on the board.
  + gameRunning is a boolean flag indicating whether the game is ongoing or finished.
  + startTime records the time when the game begins, and the time is used to calculate the total time taken when the game ends.

#### ****4. Game Flow Diagram****

Here’s an overview of how the system works in terms of user interaction:

**-Start the Program**:

* 1. User selects the option to start a new game or quit.
  2. If a new game is selected, the system prompts the user to select a difficulty level.

**-Choose Difficulty**:

* 1. Based on the user's input, the board dimensions and number of mines are set.

**-Game Loop**:

* 1. The user provides input (either "R" to reveal, "F" to flag, or "H" for a hint).
  2. The program processes the input, updates the game board, and checks if the move is valid.
  3. If the player reveals a mine, the game ends with a loss.
  4. If all non-mine cells are revealed, the game ends with a win.

****5. Game Termination**:**

* 1. The final game board is displayed, and the game results (win/loss) are shown.
  2. The time taken to complete the game is also displayed.

Recursive R eveal Logic:

* + When a player reveals a cell, the program checks if it contains a mine or if it is a safe cell with a count of adjacent mines. If the cell is safe, it reveals the number of adjacent mines and recursively reveals neighboring cells if the count is zero.

Mine Placement Algorithm:

* + Random numbers are generated to place mines on the board. The program checks each position to ensure no more than one mine is placed in the same position.

#### ****6. Error Handling and Input Validation****

* The system validates user input to ensure:
  + The coordinates entered for rows and columns are within bounds.
  + The correct action ("R", "F", or "H") is selected.
  + The input format is correct and complete (e.g., both row and column numbers are provided).
  + The system handles invalid input by prompting the user to retry until valid input is received.

**-Discrete Mathematics Logics:**

### 1. ****Matrix Representation and Indexing****

* The program uses a 2D matrix (minefield, board, revealed) to represent the game board, which is a classic discrete structure.
* Matrix traversal, as seen in the initializeBoard(), placeMines(), and calculateNumbers() methods, involves adjacency checks, a concept often used in graph theory and discrete math.

### 2. ****Logical Adjacency Calculation****

* In the calculateNumbers() method, the logic to determine the number of surrounding mines for each cell involves iterating over neighboring cells using nested loops. This can be linked to concepts in graph theory where nodes are connected to adjacent nodes.
* The bounds-checking logic ensures cells are within valid indices, showcasing conditional logic derived from set membership.

### 3. ****Recursive Exploration****

* The revealCell() method recursively reveals connected empty cells. This is akin to a depth-first search (DFS) algorithm used in graph traversal, which is a core concept in discrete mathematics.

### 4. ****Logical Condition Evaluation****

* Conditions like if (minefield[i][j] == -1) to check for mines and other validations demonstrate Boolean logic and conditional reasoning.

### 5. ****Probability (Implicitly)****

* While not explicitly computed, placing mines randomly in placeMines() introduces a probabilistic aspect. If extended, this could involve concepts like combinatorics to analyze configurations or probabilities of certain cell states.

**-Implementation:**

The Minesweeper game implementation is a Java-based console application. It incorporates several discrete mathematical concepts, such as recursion, randomization, and array manipulations, to simulate the game. Below is the step-by-step implementation of the system.

#### ****1. Main Class and Game Flow****

The Main class serves as the entry point to the game and handles user interactions, game initialization, and main gameplay loop.

· main method: Displays a menu for the user to choose either to start a new game or quit. If the user chooses a new game, it proceeds to the difficulty selection.

· chooseDifficulty method: Allows the player to select the difficulty level (Easy, Medium, or Hard). Based on the selected difficulty, it calls the initializeGame method with appropriate size and mine count.

#### ****2. Game Initialization****

The game initializes the board, minefield, and revealed arrays based on the selected difficulty level.

* initializeGame method: Sets the board dimensions (SIZE), number of mines (MINES), and initializes the board and minefield arrays. It then places the mines and calculates the adjacent mines for each cell.

#### ****3. Board Setup and Mine Placement****

The board is initialized with hidden cells, and mines are placed randomly.

· initializeBoard method: Initializes the board with '-' (hidden cells) and sets up the revealed cells as false.

· placeMines method: Randomly places mines on the board.

· calculateNumbers method: For each non-mine cell, calculates the number of adjacent mines by checking the surrounding cells.

#### ****4. Game Mechanics and User Interaction****

The main game loop processes user inputs and updates the game state accordingly.

· playGame method: Handles the game loop where the user inputs their move. The user can reveal cells, flag cells, or ask for a hint. The game continues until the user either wins or hits a mine.

· isNumeric method: Ensures the row and column input is a valid number.

· revealCell method: Reveals a cell and recursively reveals surrounding cells if the selected cell is empty (i.e., it has no adjacent mines).

· flagCell method: Flags or unflags a cell.

· hintSystem method: Reveals a safe, unrevealed cell as a hint.

#### ****5. Ending the Game****

The game checks for a win or loss condition, displays the final board, and calculates the time taken.

* gameOver method: Handles the end of the game, either by losing (hitting a mine) or winning (revealing all safe cells). It reveals the entire board and displays the game results along with the time taken.

**-Testing:**

To ensure the functionality of the Minesweeper game, various tests can be implemented to verify different aspects of the program. These tests can be performed manually or through automated unit testing in Java. Here's a basic outline of the areas to test and example test cases:

### ****1. Board Initialization Tests****

#### ****-Objective****:

Ensure the board is initialized correctly, with proper dimensions and mines placed randomly.

* Test Case 1: Verify that the board dimensions match the selected difficulty (6x6, 12x12, or 18x18).
* Test Case 2: Check that the number of mines is placed correctly based on the difficulty level.
* Test Case 3: Ensure that no mine is placed in the same location twice.

### ****2. Game Mechanics Tests****

#### ****-Objective****:

Ensure that actions like revealing and flagging cells work as expected.

* Test Case 1: After revealing a non-mine cell, ensure the cell is updated correctly with the number of adjacent mines or is blank if there are no adjacent mines.
* Test Case 2: After revealing an empty cell (0 adjacent mines), ensure that surrounding cells are automatically revealed.
* Test Case 3: Flagging a cell should change its state to marked, and it should not be revealed until the player unflags it.
* Test Case 4: Ensure that if a mine is revealed, the game ends and all mines are shown on the board

.

### ****3. Game Flow Tests****

#### ****-Objective****:

Ensure that the game progresses through the intended flow without errors.

* Test Case 1: Verify that the game prompts for user input and responds accordingly (e.g., "R 3 4" to reveal, "F 2 2" to flag).
* Test Case 2: Check that the game loops correctly, asking for new inputs until the game is over.
* Test Case 3: Ensure that the game correctly handles invalid input (out-of-bounds coordinates, invalid actions).

### ****4. Win Condition Tests****

#### ****-Objective****:

Ensure that the game ends when the player correctly reveals all safe cells.

* Test Case 1: Verify that the game correctly identifies when all non-mine cells have been revealed, triggering a win.
* Test Case 2: Check that the game ends after revealing a mine, triggering a game-over message.

### ****5. Timer Tests****

#### ****-Objective****:

Ensure that the timer starts when the game begins and stops when the game ends.

* Test Case 1: Verify that the timer starts when the game begins and that the time is calculated correctly.
* Test Case 2: Check that the timer stops when the game ends, and the elapsed time is displayed correctly.

**-Conclusion:**

In conclusion, the Minesweeper game not only meets the requirements of a fun and challenging puzzle game but also serves as a robust learning tool for exploring discrete mathematics and programming concepts. It can be expanded further by adding more advanced features such as a more sophisticated hint system, difficulty scaling, and the ability to save and load game progress.

**-References:**

AI (ChatGPT 4.0) & BlackBox AI.

Online Resources for Java Language Programming.

Discrete Mathematics with Applications By SUSANNA S. EPP.

Youtube.