VIETNAM NATIONAL UNIVERSITY – HO CHI MINH CITY

INTERNATIONAL UNIVERSITY

SCHOOL OF COMPUTER SCIENCE AND ENGINEERING



**DATA STRUCTURES AND ALGORITHMS**

(IT013IU)

## **MINESWEEPER**

FINAL PROJECT REPORT

Instructor: Thai Trung Tin

Semester 2, 2025-2026 Submitting date: 06/06/2025

| Full name | Student ID |
| --- | --- |
| Nguyễn Hoàng Bảo Trân | ITITSB22027 |
| Lê Nguyễn Thanh Trúc | ITITWE22168 |
| Lê Võ Hồng Na | ITITSB23007 |

**TABLE OF CONTENTS**

[MINESWEEPER 1](#_heading=h.tl3cmbf4tq1l)

[1. Introduction 3](#_heading=h.cap7a4cizmfb)

[1.1. Objectives 3](#_heading=h.7ak44ot8nmw9)

[1.2. Game Overview 3](#_heading=h.xmcq7yfo3w4e)

[1.3. Developer Team 3](#_heading=h.e1anititqvik)

[2. System Design 4](#_heading=h.5xemfbsby1us)

[2.1. Core Functionalities 4](#_heading=h.8gq4g09lo1r3)

[2.2. Use Case Diagram 5](#_heading=h.9x8k8oczu6ad)

[2.3. Use Case Diagram 6](#_heading=h.gpuz2aqqyodq)

[2.4. Work tools and platforms 7](#_heading=h.a4ivx6nnxkg7)

[3. Application of DSA 7](#_heading=h.ih8q5tw3rfib)

[3.1. Data Structures Used 7](#_heading=h.yhfairvr0tao)

[3.2. Algorithms Implemented 8](#_heading=h.dpo2mupyqmsn)

[1. Flood Fill Algorithm 8](#_heading=h.2hnygylt37)

[2. AI Algorithms: 9](#_heading=h.2dc960lpzjyu)

[2.1. Basic Logical Deduction 9](#_heading=h.4mq8p9pe07fx)

[2.2. Chained Subset Reasoning 10](#_heading=h.c88e91xc98ln)

[2.3. Heuristic-Based Probability Move 10](#_heading=h.n3e5rp4i4jd8)

[2.4. Undo/Redo Stack 11](#_heading=h.ru9p9kmgwla)

[2.5. Ranking and Sorting 12](#_heading=h.s4obn9epqqjd)

[3.3. Performance Evaluation 12](#_heading=h.kfbflszenxly)

[4. UI & UX 13](#_heading=h.1ykh1gfsv954)

[5. Final Application 15](#_heading=h.23zqno66fpwg)

[5.1. How to play 15](#_heading=h.mmit8smqwoi8)

[5.2. Demo video game 16](#_heading=h.4z89onu6nylm)

[6. References 16](#_heading=h.rlyzzzgcgdk1)

# 

# 

# 

# 

# 

# 

# 

# 1. Introduction

## 1.1. Objectives

The objective of this project is to create a complete and playable version of the classic game Minesweeper, while applying the knowledge we have learned in the Data Structures and Algorithms (DSA) course. Through this game, we want to improve our understanding and usage of important concepts like 2D arrays, recursion, and queue structures. One of the main challenges we aim to solve is how to reveal empty cells using a flood-fill algorithm, such as BFS or DFS. In addition, we focus on building a simple and user-friendly interface that lets players interact with the game easily, such as left-clicking to reveal a cell and right-clicking to place a flag. We also handle win and loss conditions clearly, and make sure the game runs smoothly without bugs. Our goal is to write clean, organized code that is easy to understand and update. Overall, this project helps us see how classic games like Minesweeper can be a great way to apply and practice what we’ve learned in class.

## 1.2. Game Overview

Minesweeper is a classic single-player puzzle game in which the player's goal is to uncover all the empty cells on a grid without clicking on any hidden mines. The game board consists of a grid of covered cells, and some of these cells contain hidden mines.

When a player clicks on a cell that does not contain a mine, the game either reveals a number (indicating how many mines are adjacent to that cell) or reveals a large area of empty cells if there are no nearby mines. The player can also right-click on a cell to place a flag, marking it as a suspected mine location. The game continues until the player either successfully reveals all non-mine cells (win condition) or accidentally clicks on a mine (loss condition).

In our upgraded version of Minesweeper, we introduce several enhanced features to increase user flexibility and gameplay variety. Players can choose from three difficulty levels—Easy, Medium, and Hard—which automatically adjust the grid size and number of mines. Moreover, we offer a Custom Mode, where users are free to define their own gameplay settings, including the board’s width, height, and total number of mines. This allows for a personalized gaming experience tailored to each player's preference and skill level.

## 1.3. Developer Team

| Full Name - Github Account | Student ID | Responsibility | Contribution |
| --- | --- | --- | --- |
| Lê Võ Hồng Na - [HonggNa](https://github.com/HonggNa) | ITITSB23007 | Slide, Research stack, Report, Fix bugs | 33.3% |
| Nguyễn Hoàng Bảo Trân - [Btran2404](https://github.com/Btran2404) | ITITSB22027 | Slide, Research sorting, Report, Fix bugs | 33.3% |
| Lê Nguyễn Thanh Trúc - [thanhtrucsss](https://github.com/thanhtrucsss) | ITITWE22168 | Double check code, Research AI algorithm, Fix bugs | 33.3% |

# 2. System Design

## 2.1. Core Functionalities

Our custom version of Minesweeper implements several core functionalities that define the overall gameplay experience and system logic:

* Interactive Game Grid

The game displays a grid of clickable tiles using a 2D matrix (MineTile[][] mainBoard).

Each tile responds to mouse actions (left and right click).

Revealing a tile updates the UI based on the number of nearby mines.

* Dynamic Difficulty Levels

Players can select predefined levels: Easy, Medium, or Hard.

Each level automatically adjusts grid size and number of mines using methods like Level.setLevelEasy().

* Custom Game Configuration

Users can define their own game board by setting: Width, Height, Number of Mines.

The system validates custom inputs to ensure playability and avoid crashes.

* Safe First Click

The first left-click always lands on a safe tile.

Mines are initialized after the first click using initializeMines(r, c), which avoids placing mines near the first clicked cell.

* Recursive Reveal (Flood-Fill)

Empty tiles without adjacent mines trigger a recursive reveal using checkMine(), expanding safe zones automatically.

* Flag Placement

Right-clicking a tile toggles a flag to mark a suspected mine.

The system updates the remaining flag count and prevents interaction with flagged tiles.

* Win/Loss Detection

If a player clicks on a mine, the game ends and reveals all mines.

If all non-mine tiles are revealed, the game declares victory, shows a success message, and records the time.

* User Interface Controls

Players can restart the game, switch levels, or view rankings through buttons: Play Again, Level, Custom, Rankings.

* Sound Effects

The game includes sound feedback for actions like clicking, winning, or losing, which can be toggled using a sound control button.

* Hint System

Players can get a logic-based hint suggesting a safe move. This is helpful in complex situations.

* Undo / Redo:

Using stacks, players can undo and redo their last moves, making the gameplay more flexible and forgiving.

* AutoPlay

The game includes smart algorithms that can analyze the current board and make automatic moves based on logic and probability.

## 2.2. Use Case Diagram

## 

## 2.3. Use Case Diagram

| Minesweeper | Start the game | Play the game |
| --- | --- | --- |
| Play again |
| Level |
| Custom |
| Hint |
| Auto-Play |
| Undo, Redo |
| How to play | Show how to play the game |
| Volume | Used to adjust the volume of music |
| Score history | Save game rank |
| Exit | Exit the game |

## 2.4. Work tools and platforms

| Software | Purpose |
| --- | --- |
| JDK 23 | for running java |
| Intellij | IDE to run game code |
| Canva | Design items |
| Github | Upload project code |

# 3. Application of DSA

## 3.1. Data Structures Used

* Time

Instant: Used for storing precise timestamps of two signals (signalA and signalB).

Duration: Represents the time difference between these two timestamps.

* GameState:

tileState: stores the state of each tile (0 = unopened, 1 = opened, 2 = flagged)

tileNumber stores the number displayed on each opened tile (-1 if unopened)

* Minesweeper:

MineTile[][] mainBoard: a 2D array that represents the game grid. Each element is a MineTile object, corresponding to a tile in the Minesweeper game.

ArrayList<MineTile> mineList: a dynamic list storing all the tiles that contain mines. This allows flexible access and updates.

Stack<GameState> undoStack and redoStack: these two stacks store previous and future game states, allowing the user to undo and redo moves. The stack structure supports Last-In-First-Out (LIFO) behavior, which is ideal for this feature.

* ScoreFileHandler:

A List<Long>, specifically an ArrayList, is used to store multiple score values read from a file. This allows the scores to be sorted and looped through easily when displaying rankings.

* Sound:

Clip[] (Array): Stores multiple audio clips for sound effects. The fixed-size array holds all loaded sound clips.

## 3.2. Algorithms Implemented

### Flood Fill Algorithm

**Description:**

When a blank tile (with no adjacent mines) is clicked, automatically reveal all neighboring tiles until tiles with numbers are encountered.

Flood Fill using recursive **Depth-First Search (DFS)** in 8 directions.

**Code Snippet:**

| private void checkMine(int r, int c){  if(r<0 || r>=Level.getNumRows() || c < 0 || c>= Level.getNumCols() || !mainBoard[r][c].isEnabled())  return;  MineTile tile = mainBoard[r][c];  if (tile.getIcon() == Display.flagIcon) return;  tile.setEnabled(false);  tilesClicked++;  int minePositions = numOfMinesAround(r,c);  if(minePositions > 0){  tile.setIcon(Display.numberIcons[minePositions]);  tile.setDisabledIcon(Display.numberIcons[minePositions]);  } else {  tile.setIcon(Display.nullIcon);  tile.setDisabledIcon(Display.nullIcon);  checkMine(r - 1, c - 1);  checkMine(r - 1, c);  checkMine(r - 1, c + 1);  checkMine(r, c - 1);  checkMine(r, c + 1);  checkMine(r + 1, c - 1);  checkMine(r + 1, c);  checkMine(r + 1, c + 1);  }  } |
| --- |

### AI Algorithms:

In this implementation of Minesweeper, several artificial intelligence (AI) techniques are applied to assist or automate gameplay.These techniques are mainly implemented in the functions autoSolveBasic(), autoSolveChainedReasoning(),

suggestBestProbabilisticMove(), and suggestBestEdgeMove().

#### 2.1. Basic Logical Deduction

**Description:**

This algorithm applies direct Minesweeper rules: if the number of flagged neighbors equals the number on a revealed tile, remaining neighbors are safe; if the number of hidden tiles equals the remaining mines, all are mines.

**Code Snippet:**

| if (number - flagged == unopened && unopened > 0) {  for (MineTile t : neighbors) {  t.setIcon(Display.flagIcon);  plantingFlag();  }  }  if (flagged == number && unopened > 0) {  for (MineTile t : neighbors) {  checkMine(t.getR(), t.getC());  }  } |
| --- |

**Time complexity:** O(N)

Checks all revealed tiles and their neighbors in linear time.

#### **2.2. Chained Subset Reasoning**

This function enhances logical reasoning by comparing the unknown neighbor sets of two adjacent numbered tiles.

**Description:**

If one tile's unrevealed neighbors are a subset of another's, and the difference in mine counts equals the difference in tile sets, the difference must be all mines.

**Code Snippet:**

| if (u2.containsAll(u1) && u2.size() > u1.size()) {  int diff = num2 - num1;  if (diff >= 0 && (u2.size() - u1.size()) == diff) {  for (MineTile t : u2) {  if (!u1.contains(t)) {  t.setIcon(Display.flagIcon);  plantingFlag();  }  **}}}** |
| --- |

**Time complexity:** O(N²)

Uses chained subset logic to deduce safe tiles or mines by comparing overlapping constraint sets.

#### 2.3. Heuristic-Based Probability Move

This method estimates the probability of each unrevealed tile being a mine and picks the one with the lowest estimated risk.

**Description:**

It examines each unrevealed tile’s neighbors. For neighboring number tiles, it calculates remaining mines vs. unrevealed tiles and chooses the tile with the smallest chance of being a mine.

**Code Snippet:**

| double prob = (double) left / unknown;  if (prob < minProb) {  minProb = prob;  bestTile = tile;  } |
| --- |

**Time complexity:** O(N)

Heuristic-based method to find the tile with the lowest probability of being a mine and suggest it as the best move.

* suggestBestEdgeMove(): Edge-Favoring Heuristic

This fallback heuristic is used when no clear safe move is found. It prioritizes edge tiles assuming fewer adjacent unknowns.

**Description:**

If probabilistic logic is unavailable, the function selects an unopened tile from the border of the board, ideally one that has some local number constraints.

**Code Snippet:**

| if (r == 0 || r == rows - 1 || c == 0 || c == cols - 1) {  edgeTiles.add(mainBoard[r][c]);  } |
| --- |

**Time complexity:** O(E) (E ≤ N), worst case O(N)

Makes guesses around the board edges, usually fewer tiles than total.

#### 2.4. Undo/Redo Stack

**Description:**

Allow players to undo and redo actions using a LIFO (Last In, First Out) stack

**Code Snippet:**

| Stack<GameState> undoStack = new Stack<>();  Stack<GameState> redoStack = new Stack<>();  public void saveStateForUndo() {  undoStack.push(new GameState(this));  redoStack.clear();  }  public void undo() {  if (!undoStack.isEmpty()) {  redoStack.push(new GameState(this));  GameState prev = undoStack.pop();  prev.restore(this);  display.boardPanel.repaint();  } else {  JOptionPane.showMessageDialog(display.frame, "No more undo available!");  }  }  public void redo() {  if (!redoStack.isEmpty()) {  undoStack.push(new GameState(this));  GameState next = redoStack.pop();  next.restore(this);  display.boardPanel.repaint();  } else {  JOptionPane.showMessageDialog(display.frame, "No more redo available!");  }  } |
| --- |

#### 

#### 2.5. Ranking and Sorting

**Description:**

Sort and retrieve the Top 10 players based on score or time.

**Code Snippet:**

| public static String toStringScore(String fileName) {  List<long[]> list = readScoreFromFile(fileName);  list.sort((a, b) -> {  if (a[1] != b[1]) return Long.compare(b[1], a[1]);  return Long.compare(a[0], b[0]);  });  ...  } |
| --- |

## 3.3. Performance Evaluation

Our Minesweeper game integrates traditional game logic with advanced features such as AI-assisted moves, undo/redo functionality, and customizable difficulty levels. Below is a comprehensive evaluation of the system's performance across core areas:

**Time Complexity Analysis**

* Basic Gameplay (Reveal, Flag):

Reveal logic (checkMine()): O(n), where *n* is the number of recursively revealed tiles using flood-fill (DFS).

Flagging and UI update: O(1)

* AI Algorithms:

autoSolveBasic(): O(N)

Applies basic logic across revealed tiles and their neighbors.

autoSolveChainedReasoning(): O(N²)

Compares subsets of neighbor tiles for advanced logical deductions.

suggestBestProbabilisticMove(): O(N)

Scans unrevealed tiles and estimates mine probability heuristically.

suggestBestEdgeMove(): O(E) ⊆ O(N)

Selects edge tiles for fallback guesses.

* Undo/Redo Stack Operations:

Push and pop operations are constant time: O(1)

Efficient state management using Stack<GameState> ensures minimal overhead.

* Mine Placement:

Randomized algorithm takes approximately O(k), where *k* is the number of mines.

**Space Complexity and Memory Usage**

* Game Board (MineTile[][]):

Space complexity is O(rows × cols), with each cell storing UI state and logic data.

* Mine List (ArrayList<MineTile>):

O(m), where *m* is the total number of mines.

* Undo/Redo System (Stack<GameState>):

Space usage depends on the number of moves made; the worst case is O(M), where *M* is total move count.

* AI Processing Sets and Lists:

Temporary structures in AI reasoning are lightweight and only persist during processing, ensuring no memory leaks.

# 4. UI & UX

**UI and design**

This is a classic Minesweeper game implemented as a standalone desktop application using Java Swing. The interface is minimalistic and intuitive, designed to provide a smooth and familiar gameplay experience for users.

The user interface consists of the following main sections:

* Game Board

The primary interactive area of the Minesweeper game is the game board, which is structured as a grid of clickable cells arranged in rows and columns. Each cell is implemented using a **JButton** and can exist in one of several states: hidden (default), revealed (displaying a number or a mine), or flagged (marked by the player as a potential mine). Players interact with the board entirely through mouse actions. A left-click reveals the selected cell, while a right-click toggles a flag on or off. Once revealed, a cell will either show a number indicating the count of adjacent mines, remain blank if there are none, or display a mine icon if the player has accidentally clicked on a bomb.

* Control Panel

At the bottom of the interface lies the control panel, which provides players with essential game interaction features. This panel includes a “Play Again” button to quickly restart the game, and a set of difficulty options—Easy, Medium, and Hard—that allow players to choose predefined board sizes and mine counts. A “Ranking” button is also available, enabling players to view their scores or compare results. For greater customization, a “Custom” mode lets users manually configure the board’s dimensions and the number of mines to suit their preferences. Additionally, a “Sound” toggle button allows players to enable or disable game audio, enhancing user comfort without disrupting gameplay. These controls are clearly laid out, providing a seamless and efficient user experience.

* Pop-up Dialogs and Game Feedback

The game incorporates various pop-up dialogs and feedback mechanisms to enhance user interaction and provide clear communication during gameplay. When the player clicks on a mine, a “Game Over” dialog appears, notifying them of their loss and offering the option to play again. Upon successfully uncovering all safe cells, a “Victory” dialog is displayed to congratulate the player. If the player enters invalid values in custom mode—such as specifying more mines than the number of available cells—an alert dialog informs them of the error and prompts correction



Figure 1: User Interface

**User Experience (UX)**

First-click safety: The game ensures that the first cell clicked by the player is never a mine, providing a fair starting point.

Visual consistency: Icons and numbers follow traditional Minesweeper conventions, making the game instantly recognizable.

Responsiveness: The game responds instantly to user actions without noticeable delay.

Keyboard-free: All gameplay is controlled via mouse, simplifying interaction.

# 5. Final Application

## 5.1. How to play

To run and play our custom Minesweeper game, follow these simple steps:

1. Download and run the game

* Clone the project from [**Link Github**](https://github.com/thanhtrucsss/MineSweeper) or download the full source code folder.
* Open the project in any Java IDE of your choice
* Locate the Application.java file, which contains the main() method, and run it to launch the game

1. Choose the game mode

* Once the game window appears, you can choose how you want to play:
* Click the "Level" button to select a predefined difficulty: Easy, Medium, or Hard.
* Or click the "Custom" button to enter your own board configuration by choosing: Number of lengths, number of widths, number of mines.

1. Play the game

* Left-click on a tile to reveal it.
* Right-click to place or remove a flag.
* The game ensures that your first click is always safe (no mine).

1. Win or lose

* -If you reveal all non-mine tiles, a "You Won" message appears, and your completion time is saved.
* If you click on a mine, the game shows "Game Over, you lose", and all mines are revealed.

1. Restart the game.

* After winning or losing, click the "Play Again" button to start a new game with the same settings.

## 5.2. Demo video game

[Game Demo Video](https://www.canva.com/design/DAGplYTg2lA/TDu8if3_XvHX3tLaKNUpkA/edit?utm_content=DAGplYTg2lA&utm_campaign=designshare&utm_medium=link2&utm_source=sharebutton)

# 6. References

GeeksforGeeks. (2025, Mar 27). Stack Data Structure. Retrieved from: <https://www.geeksforgeeks.org/stack-data-structure/>

GeeksforGeeks. (2025). Artificial Intelligence (AI) Algorithms. Retrieved from:

<https://www.geeksforgeeks.org/ai-algorithms/>

GeeksforGeeks. (2023, Aug 24). ArrayList vs LinkedList. Retrieved from

<https://www.geeksforgeeks.org/arraylist-vs-linkedlist-java/>

Reference Video:

<https://www.youtube.com/watch?v=5VrMVSDjeso>

THE END.