VIETNAM INTERNATIONAL UNIVERSITY – HO CHI MINH CITY

INTERNATIONAL UNIVERSITY

ALGORITHMS & DATA STRUCTURES

PROJECT

MINESWEEPER

By

Nguyen Phuc Vinh – ITITIU21350

Advisor: Thai Trung Tin

A report submitted to the School of Computer Science and Engineering in partial fulfillment of the requirements for the Final Project in Algorithms & Data Structures course – 2024

Ho Chi Minh City, Vietnam, 2024

OUTLINE

I. Introduction

In this report, I will introduce my project, which is a Minesweeper game developed as part of the course on Algorithms & Data Structures. Minesweeper is a classic and widely popular game that challenges players to uncover all the mines hidden on a grid without detonating any of them.

For this project, I chose the web environment as the primary platform for developing Minesweeper. This choice leverages the accessibility and ubiquity of web technologies, allowing the game to be played on various devices with internet access. The web platform also provides a rich set of tools and frameworks that facilitate the implementation of algorithms and data structures critical to the game's functionality.

Throughout this report, I will detail the design and implementation process, the algorithms employed, and the challenges encountered during development. By doing so, I aim to demonstrate the practical application of theoretical concepts learned in the course and showcase the final product's capabilities and features.

II. UML

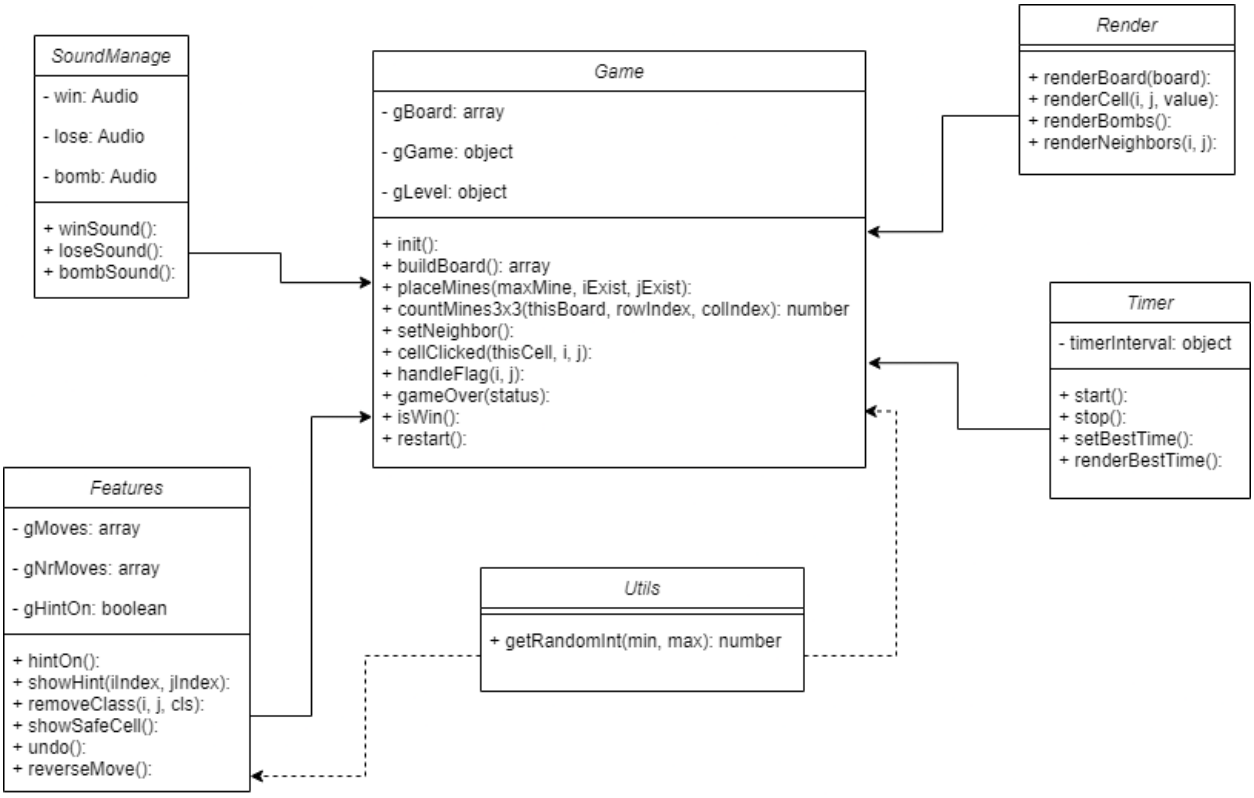


Figure 1

Class descriptions:

A. "Game" class:

The Game class serves as the central component orchestrating the game's core functionalities. It maintains the game board (gBoard), the game object (gGame), and the game level (gLevel). The class is responsible for initializing the game, constructing the board, placing mines, counting mines in a 3x3 grid, setting neighbors, handling cell clicks, managing flags, determining win conditions, and restarting the game.

B. "Render" class:

The Render class is responsible for the visual representation of the game state. It provides methods to render the entire board, individual cells, bombs, and neighbors.

C. "SoundManage" class:

The SoundManage class handles the audio feedback for the game. It includes attributes for win, lose, and bomb sounds, and methods to play these sounds.

D. "Timer" class:

The Timer class manages the timing aspects of the game. It can start and stop the timer, set the best time, and render the best time.

E. “Features” class:

   The Features class encompasses additional functionalities such as hints, move tracking, and undo operations. It manages the game's move history and provides methods for showing hints, safe cells, and reversing moves.

F. “Utils” class:

   The Utils class provides utility functions to support other classes. It includes a method to generate random integers within a specified range.

III. Data structures

1. Array:

To create the game board, I used a 2D array, which effectively represents the grid and the distribution of mines. This structure allows for efficient access and manipulation of the grid's cells, ensuring smooth gameplay and accurate mine placement.

```
class Game {
    constructor() {
        this.gBoard = [];
        this.gGame = {
            isRun: false,
            isFirst: true,
            shownCount: 0,
            markedCount: 0,
            secsPassed: 0,
            isSound: true,
        }
        this.gLevel = {
            size: 4,
            mines: 2,
            lives: 1,
            hints: 1,
            bestTime: + localStorage.bestTimeEasy,
            safeClicks: 1,
        }
    }
```

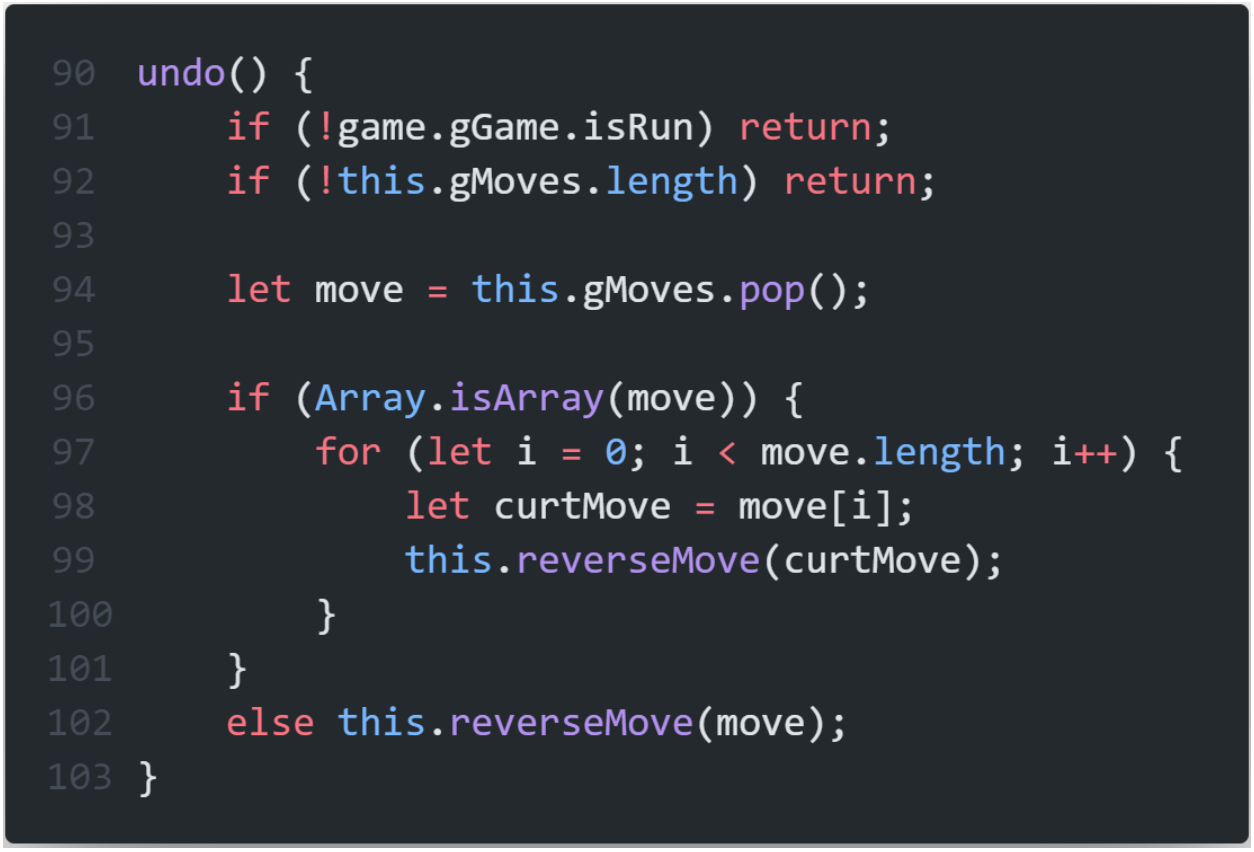
Figure 2

2. Stack:

I implemented an undo feature using a stack data structure. This allows players to revert their moves, enhancing the user experience by providing a way to correct mistakes and explore different strategies without restarting the game.

```
class Features {
    constructor() {
        this.gMoves = [];
        this.gNrMoves = [];
        this.gHintOn = false;
    }
```

Figure 3



```
undo() {
    if (!game.gGame.isRun) return;
    if (!this.gMoves.length) return;

    let move = this.gMoves.pop();

    if (Array.isArray(move)) {
        for (let i = 0; i < move.length; i++) {
            let curtMove = move[i];
            this.reverseMove(curtMove);
        }
    }
    else this.reverseMove(move);
}
```

Figure 4

IV. Algorithm

1. Build the board:

```
renderBoard(board) {
    let size = board.length;

    let boardHTML = '';

    for (let i = 0; i < size; i++) {
        boardHTML += '<tr>';

        for (let j = 0; j < size; j++) {
            let className = `cell cell${i}-${j}`;
            boardHTML += `<td class="${className}" onclick="cellClicked(this, ${i}, ${j})" oncontextmenu="handleFlag(${i}, ${j}); return false;"></td>`;
        }

        boardHTML += '</tr>';
    }

    document.querySelector('.game-board>table').innerHTML = boardHTML;
}
```

Figure 5

- Because the outer and inner loop is n. So n*n = n^2.
  ⇨ Time complexity: O(n^2).
- This algorithm will generate the rows corresponding to the level of this game.
- It also has the function to check if user click the cell, what action will run.

2. Render the neighbors:

```
renderNeighbors(i, j) {
    for (let rowIdx = i - 1; rowIdx <= i + 1; rowIdx++) {
        if (rowIdx < 0 || rowIdx > game.gBoard.length - 1) continue;

        for (let colIdx = j - 1; colIdx <= j + 1; colIdx++) {
            if (colIdx < 0 || colIdx > game.gBoard[0].length - 1) continue;
            if (rowIdx === i && colIdx === j) continue;

            let currCell = game.gBoard[rowIdx][colIdx];
            if (currCell.isMarked || currCell.isShown) continue;
            feature.gNrMoves.push(currCell);

            this.renderCell(rowIdx, colIdx, currCell.minesAroundCount);
            currCell.isShown = true;
            let elCell = document.querySelector(`.cell${rowIdx}-${colIdx}`);
            elCell.classList.add('pressed');
            if (!currCell.minesAroundCount) this.renderNeighbors(rowIdx, colIdx);
        }
    }
}
```

Figure 6

- The time complexity of each individual operation is typically considered to be constant time, denoted as O(1). This means that the time it takes to perform these operations does not depend on the size of the input.
- Therefore, in the worst case, where the outer and inner loops iterate over a maximum of 9 times, the overall time complexity of the renderNeighbors function can be approximated as O(1) * 9, which simplifies to O(1).
- In other words, the time complexity of the renderNeighbors function is constant, regardless of the size of the game board or the number of cells being rendered.

3. Counting mines:

```
76 countMines3x3(thisBoard, rowIndex, colIndex) {
77     let count = 0;
78 
79     for (let i = rowIndex - 1; i <= rowIndex + 1; i++) {
80         if (i < 0 || i > thisBoard.length - 1) continue;
81 
82         for (let j = colIndex - 1; j <= colIndex + 1; j++) {
83             if (j < 0 || j > thisBoard[0].length - 1) continue;
84 
85             if (i === rowIndex && j === colIndex) continue;
86 
87             let currentCell = thisBoard[i][j];
88             if (currentCell.isMine) count++;
89         }
90     }
91 
92     return count;
93 }
```

Figure 7

- The term n of inner and outer loop is the same which is 3.
- So 3x3 = 0.

  ⇨ Time complexity: O(1).

V. Implements

- The initialization of this game:

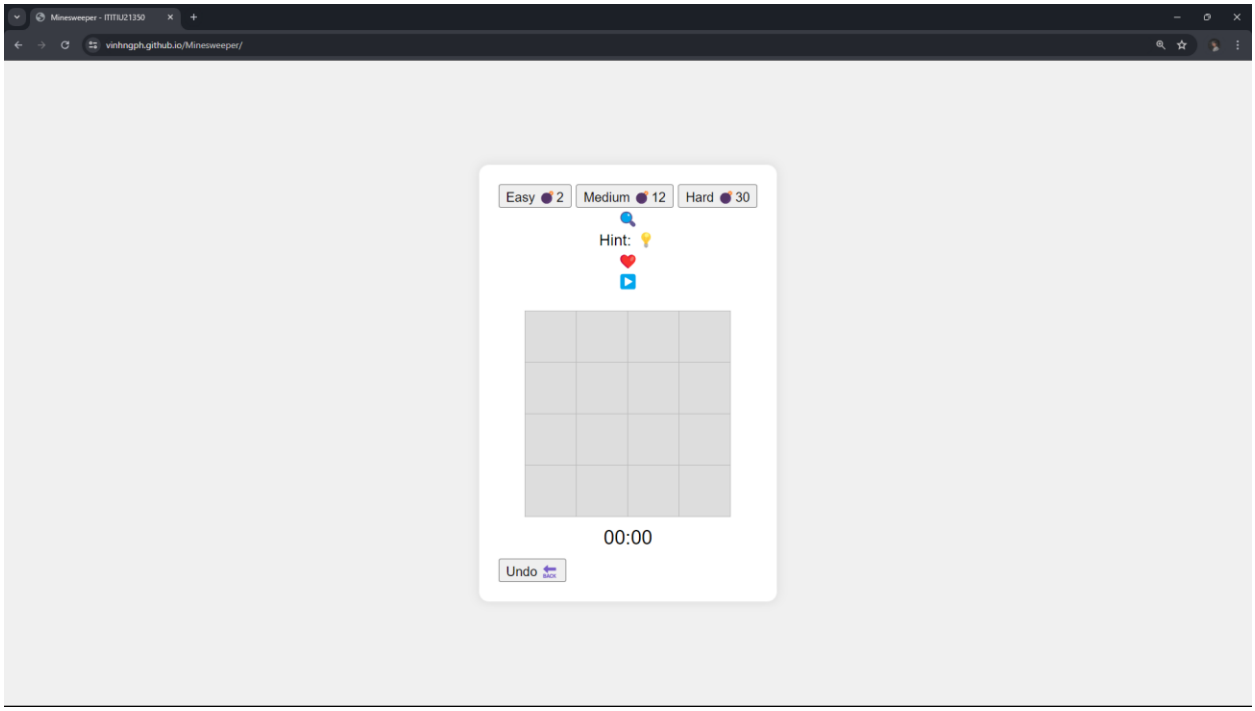


Figure 8

- Start the game:

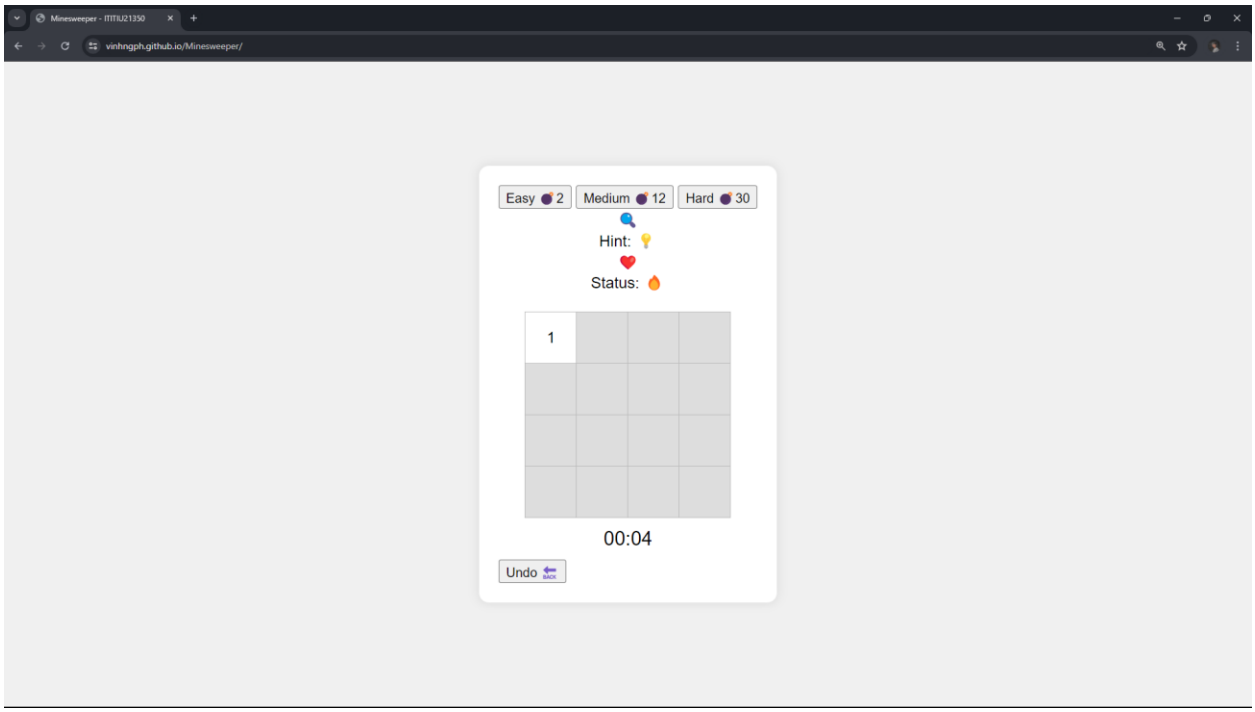


Figure 9

- Game over because in easy you just have 1 live to play:

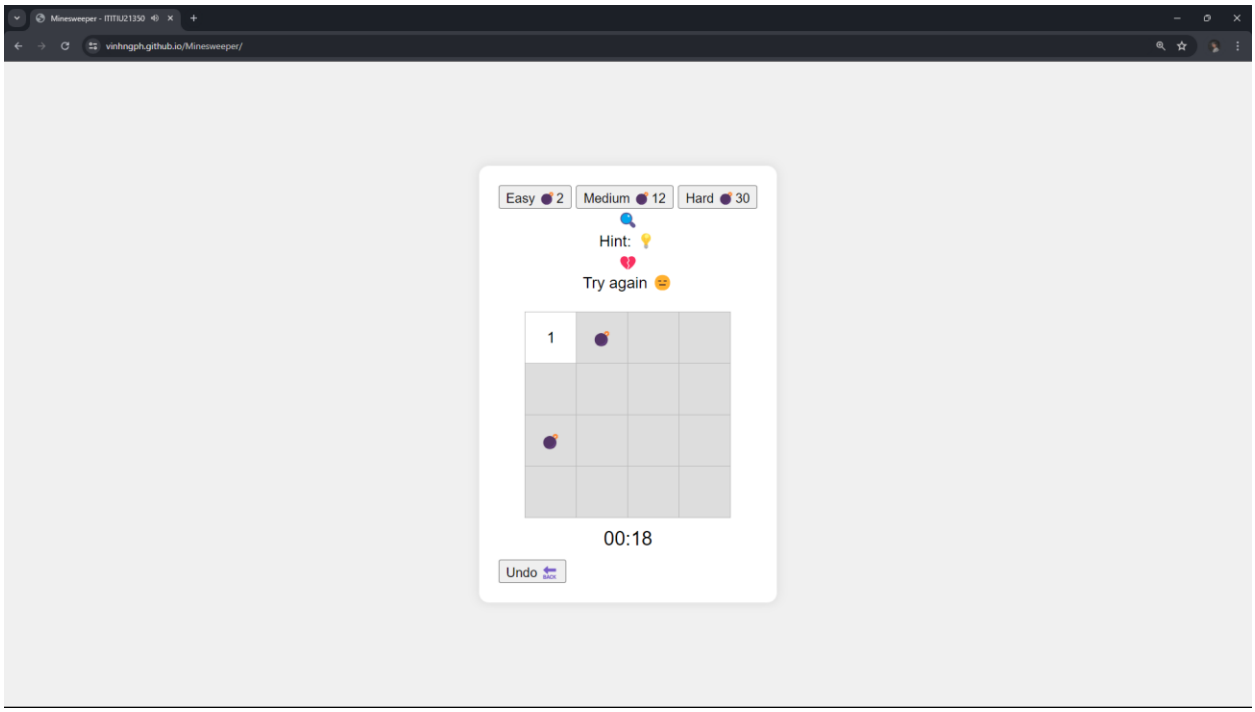


Figure 10

- Click "Try again 😑" to continue:

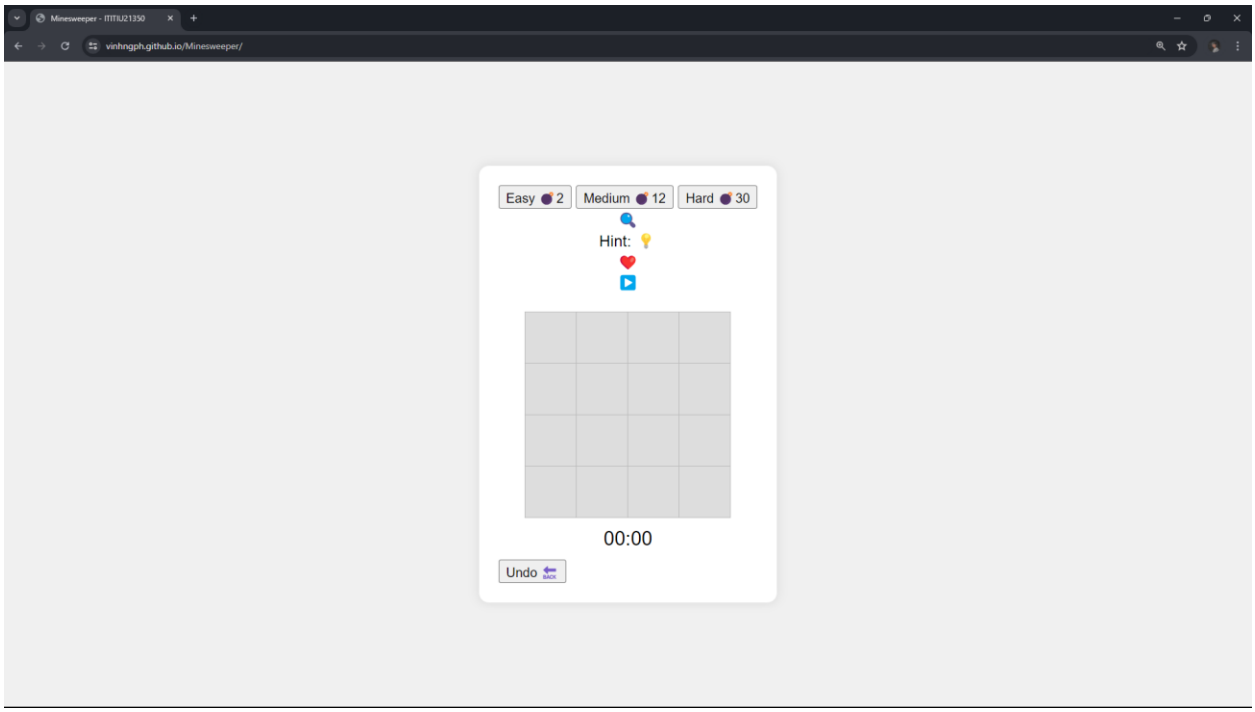


Figure 11

- When I use “Hint”, the total hints will be decrease by 1. This feature whill show all cell in around 3x3 then disappear in 2 seconds:

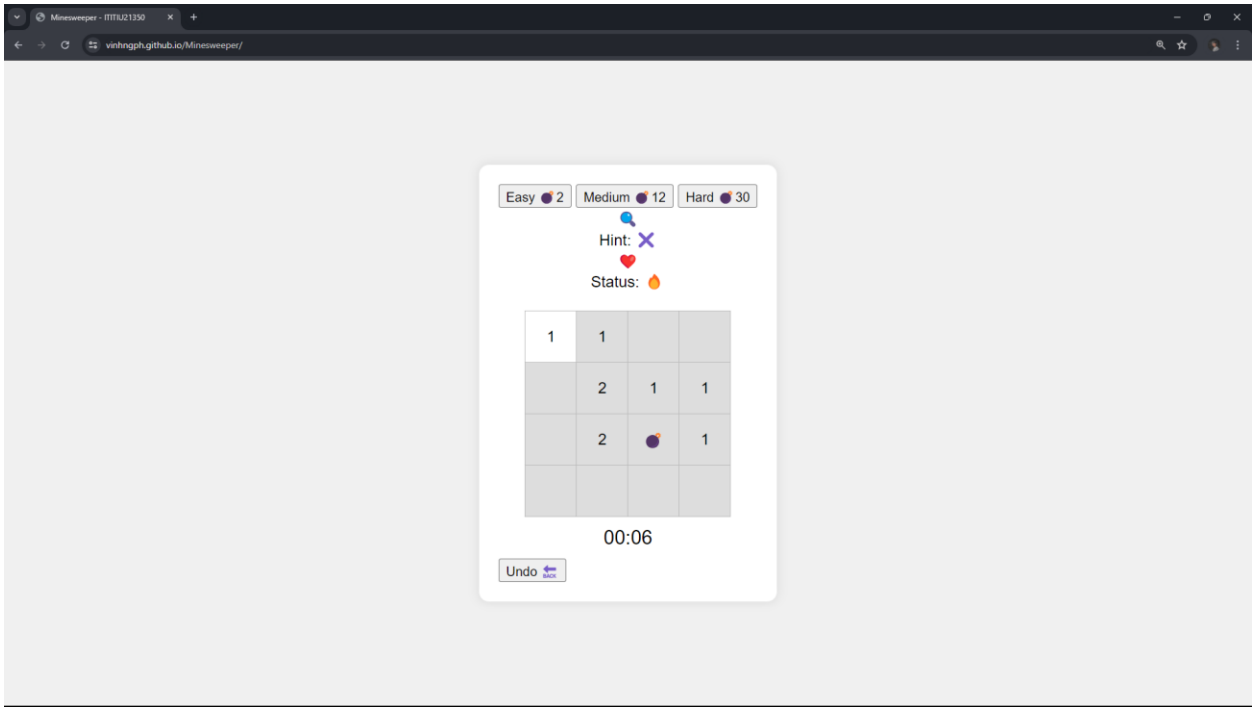


Figure 12

- When I use “Safe Click”, it will find all safe sell and avoid cell has mine, then it will random to show 1 cell for player:

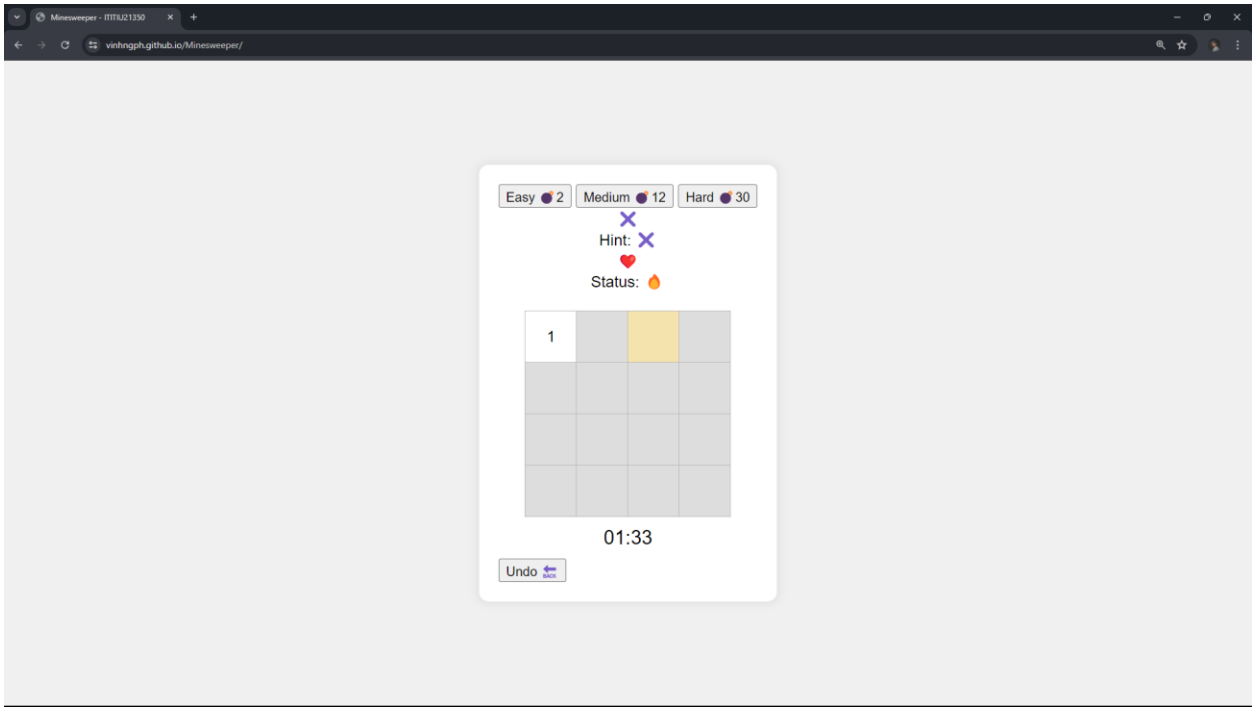


Figure 13

- After clicking the safe cell, I can unlock more cells:

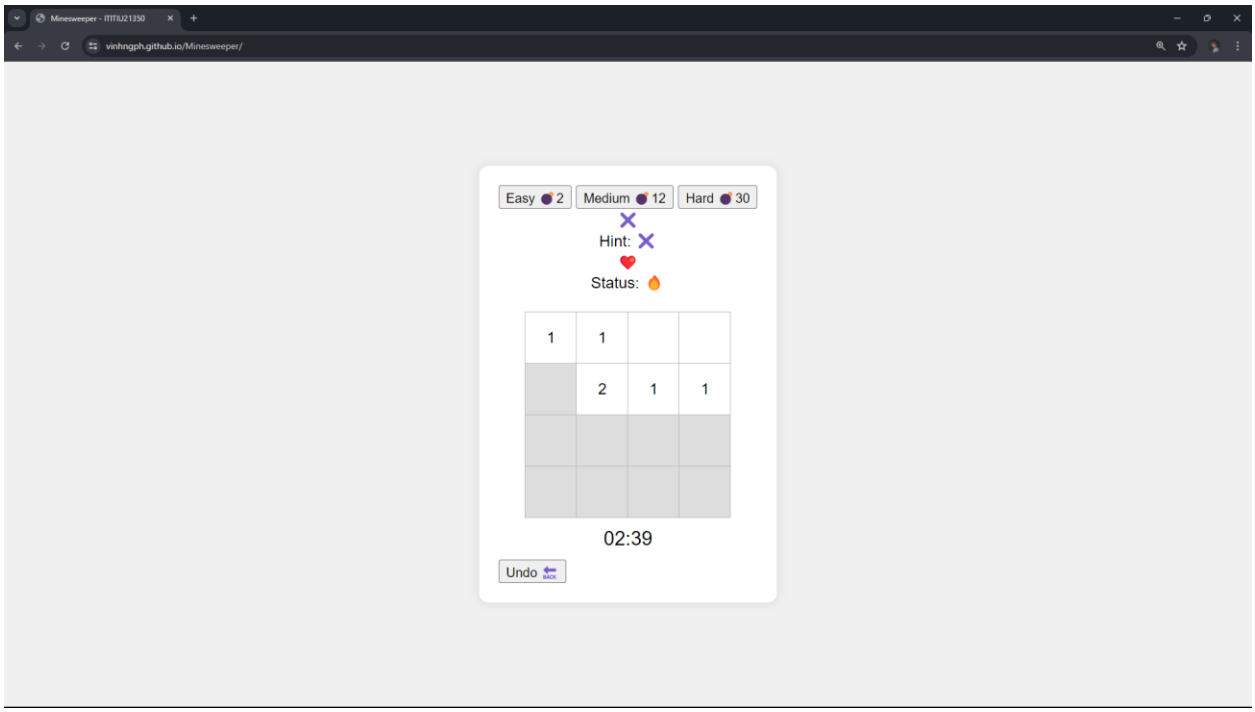


Figure 14

- I also can click “Undo” to comback the previous action:

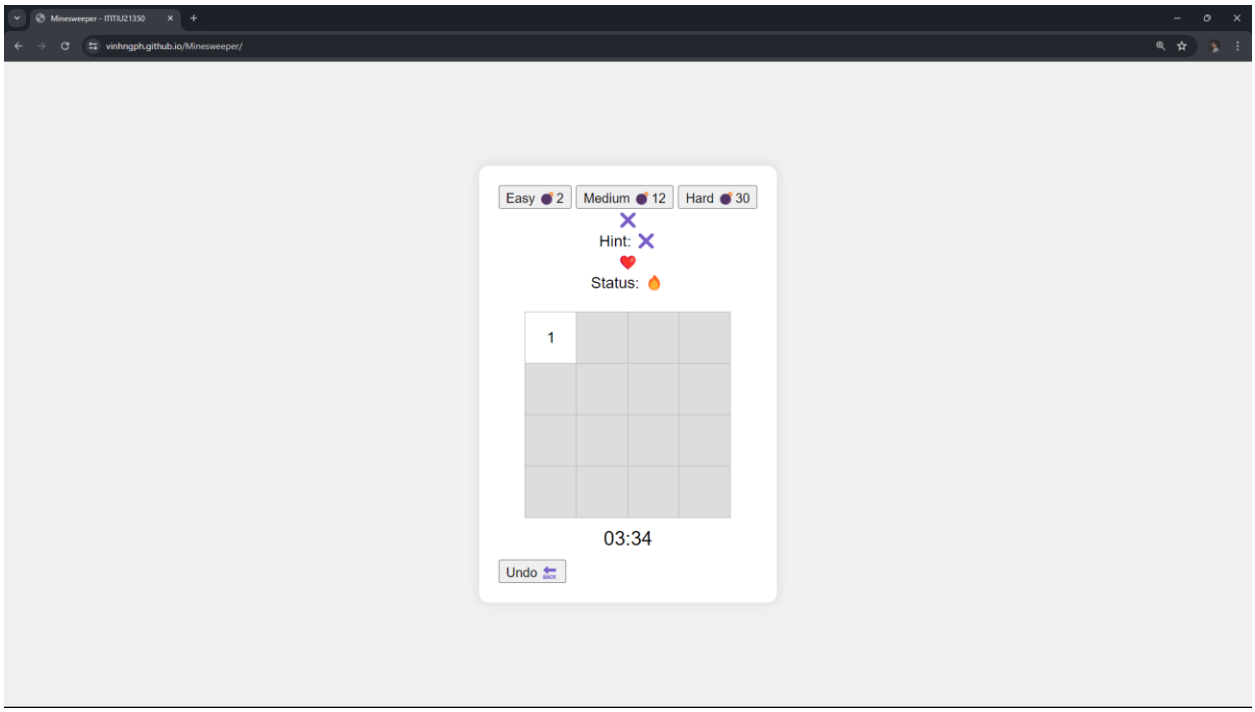


Figure 15

- If I unlock all safe cell and marked true mine cells, I will receive a message that I win that game, then compare with the previous best time if it less than, updating

it:

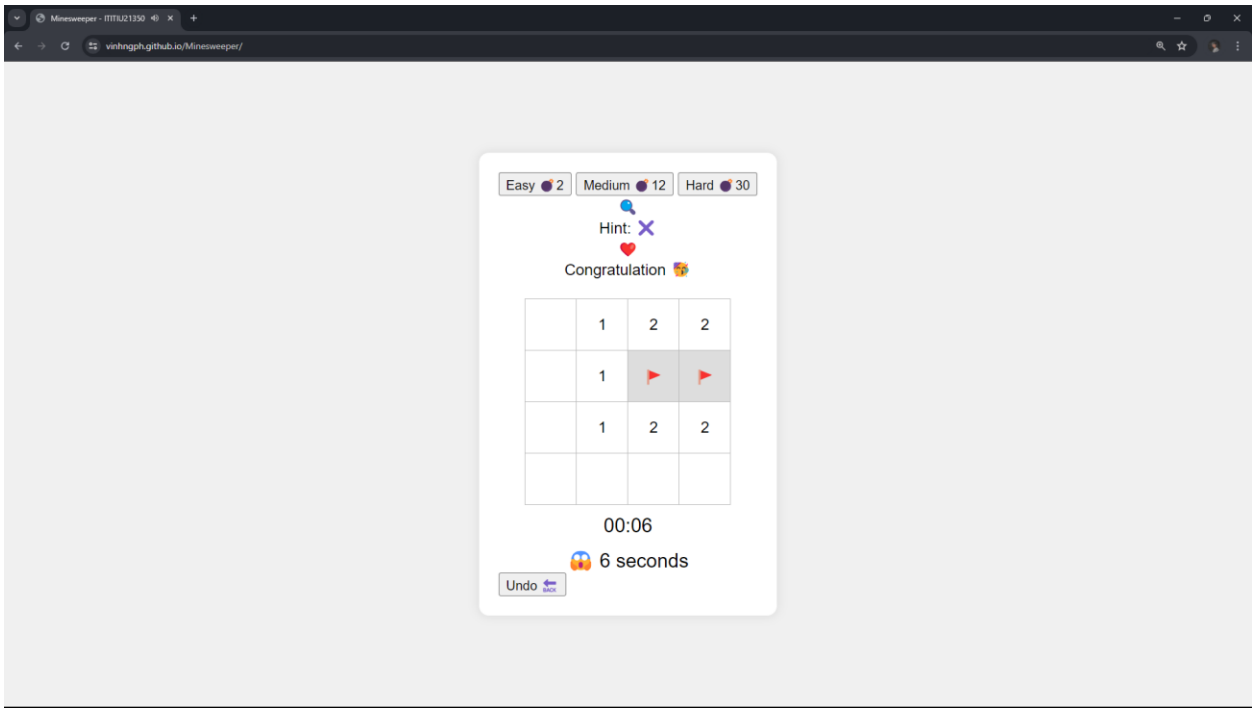


Figure 16

- With the same of algorithms, in medium level I have the table 8x8 and 12 mines. Increase the time to use “Hints” and “Safe cell”:

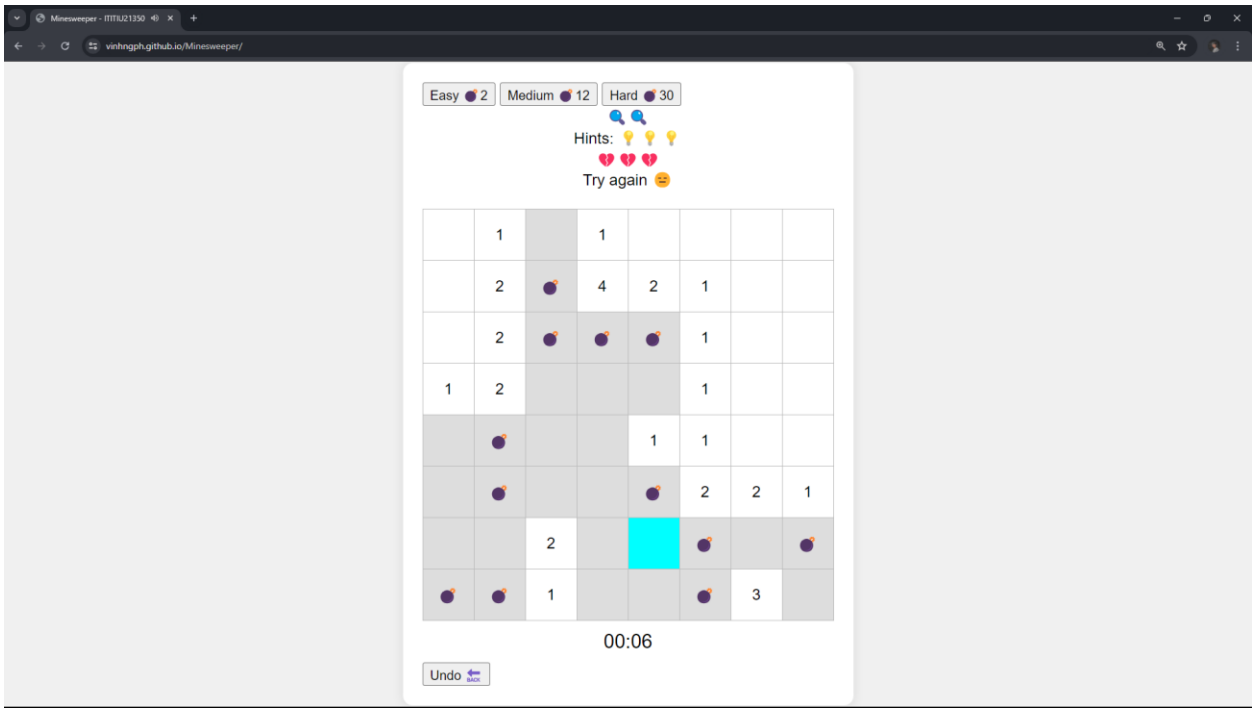


Figure 17

- And this is the hard mode with 30 mines with table size 12x12:

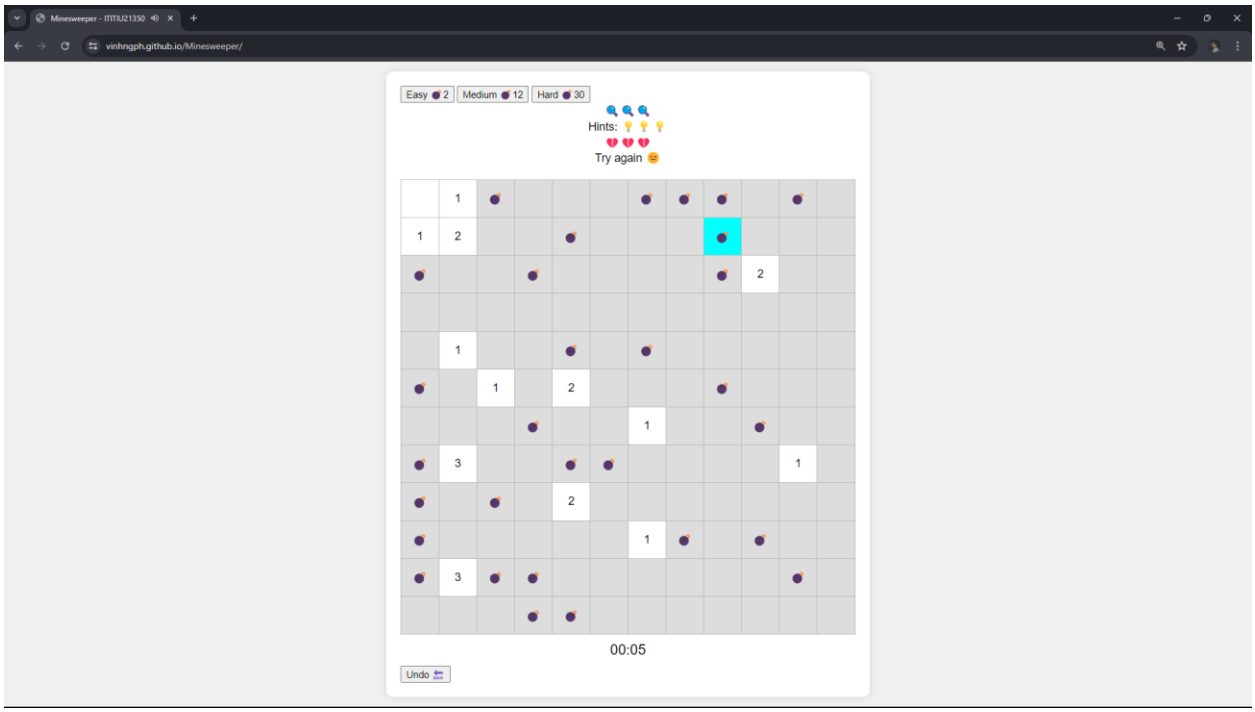


Figure 18

- Finally, this project was deployed in: https://vinhngph.github.io/Minesweeper/

VI. Conclusion

In conclusion, the development of the Minesweeper game as part of the Algorithms & Data Structures course has been an enlightening and rewarding experience. By choosing the web environment as the primary platform, I was able to utilize various web technologies to create a fully functional and accessible game. This project provided a practical application of the theoretical concepts covered in the course, particularly in the areas of algorithm design and data structure implementation.