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

Table of Contents

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

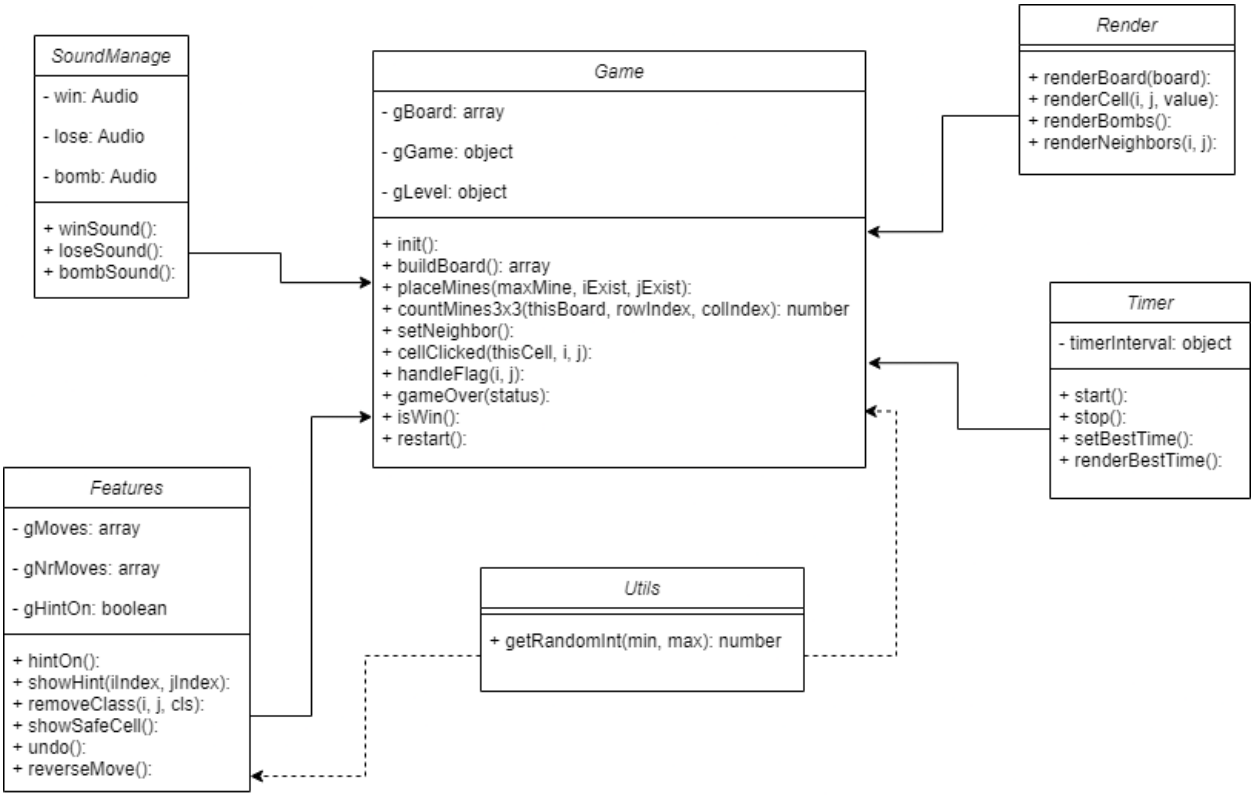

Figure 1

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.
- In JavaScript, a stack can be represented using an array, utilizing the pop and push methods.
- The gMoves is a stack used for the undo feature.

Figure 3

IV. Algorithm

1. Define the structure of the board game:

```
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 4

- gBoard: where to store the cells.
- gGame: define all information of this game.
- gLevel: define the level of this game, “Easy” is a default.

2. Initialize the game:

```
init() {
    gPlayer.innerText = NORMAL;
    this.gBoard = this.buildBoard();
    render.renderBoard(this.gBoard);

    let size = this.gLevel.size;

    gLives.innerText = size === 4 ? '❤️' : '❤️❤️❤️';
    gHints.innerText = size === 4 ? 'Hint: 💡' : 'Hints: 💡💡💡';
    gSafe.innerText = size === 4 ? '🫧' : size === 8 ? '🫧🫧' : '🫧🫧🫧';

    if (!this.gLevel.bestTime) return;
    timer.renderBestTime();
}
```

Figure 5

- init() is a beginning function of this game.
- The logic of this function:
- buildBoard() => renderBoard() => show the detail informations.

3. Build the board:

```
buildBoard() {
    let board = [];

    for (let i = 0; i < this.gLevel.size; i++) {
        board.push([]);

        for (let j = 0; j < this.gLevel.size; j++) {
            let cell = {
                minesAroundCount: null,
                isShown: false,
                isMarked: false,
                isMine: false,
                position: { i: i, j: j }
            }

            board[i].push(cell);
        }
    }

    return board;
}
```

Figure 6

- The time complexity: $O(n^2)$.
- Because the outer and inner loop is n => n*n = n^2.
- The outer loop will run each row of this board.
- In each row, it will first create an array, then it continues to run through all the subsequent columns. In each column, it will define that position as a cell and add the following information for that cell: minesAroundCount, isShown, isMarked, isMine, and position.

4. Render 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 7

- The time complexity: $O(n^2)$.
- It will loop each row; in each row it will loop each column.

- In each row, as it loops through the columns, each column will be a cell with the HTML tag <td>. Each cell will have a class named "cell cell(row-position)-(column-position)". The onclick event will call the function cellClicked, passing in the <td> element of that cell and its position. The oncontextmenu event will call the function handleFlag to assign a flag to the cell at that position.

5. **Render the best time:**

```
renderBestTime() {
    const bestTime = game.gLevel.bestTime;
    if (bestTime < 60)
        gBestTime.innerText = `😱 ${bestTime} seconds`;
    else if (bestTime > 60) {
        let mins = (bestTime / 60).toFixed(2);
        gBestTime.innerText = `😱 ${mins} minutes`;
    }
}
```

Figure 8

- Time complexity: O(1).
- It will get the value of bestTime stored in gLevel.
- If this time greater than 60, it will show the time with minutes format.
- If this time is less than, it will display the time in seconds.

6. Handle left clicking:

```
cellClicked(thisCell, i, j) {
    let cell = this.gBoard[i][j];

    if (cell.isShown || cell.isMarked) return;

    if (!this.gGame.isRun && this.gGame.isFirst) {
        this.gGame.isRun = true;
        this.gGame.isFirst = false;
        gPlayer.innerText = FIRE;

        timer.start();

        this.placeMines(this.gLevel.mines, i, j);
        this.setNeighbor();
    }
    // Hint feature
    else if (feature.gHintOn) {
        return feature.showHint(i, j);
    }
    // Click mine
    else if (cell.isMine) {
        if (this.gGame.isSound) soundManage.bombSound();

        cell.isShown = true;

        if (this.gLevel.lives > 1) {
            this.gLevel.lives--;

            feature.gMoves.push(cell);
            render.renderCell(i, j, MINE);

            let remainLives = this.gLevel.lives;

            switch (remainLives) {
                case 2:
                    gLives.innerText = '❤️ ❤️ 💔';
                    break;
                case 1:
                    gLives.innerText = '❤️ 💔 💔';
                default:
                    break;
            }
            gPlayer.innerText = SAD;
            return;
        }
        else if (this.gLevel.lives === 1) {
            render.renderBombs();
            this.gameOver('lost');
            return;
        }
    }

    // Main
    if (this.gGame.isRun) {
        if (cell.minesAroundCount > 0) {
            thisCell.innerHTML = cell.minesAroundCount;
            feature.gMoves.push(cell);
        } else {
            render.renderNeighbors(i, j);
            feature.gMoves.push(feature.gNrMoves.slice());
            feature.gMoves[feature.gMoves.length - 1].push(cell);
            feature.gNrMoves = [];
        }

        thisCell.classList.add('pressed');
        cell.isShown = true;
        if (this.isWin()) this.gameOver('win');
    }
}
```

Figure 9

- First, it will get the cell is clicked.

```
107 let cell = this.gBoard[i][j];
108
109 if (cell.isShown || cell.isMarked) return;
```

Figure 10

- If this cell is shown or marked, do not do anything.
- Seconds, handling if this is the first click.

```
111 if (!this.gGame.isRun && this.gGame.isFirst) {
112     this.gGame.isRun = true;
113     this.gGame.isFirst = false;
114     gPlayer.innerText = FIRE;
115
116     timer.start();
117
118     this.placeMines(this.gLevel.mines, i, j);
119     this.setNeighbor();
120 }
```

Figure 11

- Change the status of this game, run the timer.

```
6  start() {
7      let eTimer = document.querySelector('.timer');
8      this.timerInterval = setInterval(() => {
9          game.gGame.secsPassed++;
10
11         let timeArray = new Date(game.gGame.secsPassed * 1000).toString().split(':');
12         let minutes = timeArray[1];
13         let seconds = timeArray[2].split(' ')[0];
14         eTimer.innerText = `${minutes}:${seconds}`;
15     }, 1000);
16 }
```

Figure 12

Every second, it will update the display of the timer tag in HTML.

- Third, it will place all mines for this game.

```
placeMines(maxMine, iExist, jExist) {
    let countMine = 0;

    while (countMine < maxMine) {
        let size = this.gLevel.size;

        let i = util.getRandomInt(0, size);
        let j = util.getRandomInt(0, size);

        if (this.gBoard[i][j].isMine) continue;
        if (i === iExist && j === jExist) continue;

        this.gBoard[i][j].isMine = true;
        countMine++;
    }
}
```

Figure 13

- Time complexity: O(n)
- Get the max mine for this level.
- Random position for i and j.
- If this position is the first clicked, continue the algorithm.
- If this position existed the mine, continue the algorithm.
- Set the mine for this position.

- Fourth, in each cell, count the number of mines in the surrounding 3x3 area, then assign that number to the cell.

```
setNeighbor() {
    for (let i = 0; i < this.gBoard.length; i++) {
        for (let j = 0; j < this.gBoard.length; j++) {
            if (this.gBoard[i][j].isMine) continue;
            let mineCount = this.countMines3x3(this.gBoard, i, j);
            if (!mineCount) mineCount = '';
            this.gBoard[i][j].minesAroundCount = mineCount;
        }
    }
}
```

Figure 14

- Time complexity: $O(n^2)$.
- In each cell it will call countMines3x3, if the return is null, it set this cell is empty.
- If return is a number, set this number to this cell.

```
countMines3x3(thisBoard, rowIndex, colIndex) {
    let count = 0;

    for (let i = rowIndex - 1; i <= rowIndex + 1; i++) {
        if (i < 0 || i > thisBoard.length - 1) continue;

        for (let j = colIndex - 1; j <= colIndex + 1; j++) {
            if (j < 0 || j > thisBoard[0].length - 1) continue;

            if (i === rowIndex && j === colIndex) continue;

            let currentCell = thisBoard[i][j];
            if (currentCell.isMine) count++;
        }
    }

    return count;
}
```

Figure 15

- Time complexity: O(1).

- Because it always run 3 rows and 3 columns for each row.
- If will check each mine in area 3x3 is mine, increasing the count mine.

- Fifth, handle if this click is hint feature.

```
122 else if (feature.gHintOn) {
123     return feature.showHint(i, j);
124 }
```

Figure 16

```
34 showHint(iIndex, jIndex) {
35     for (let i = iIndex - 1; i <= iIndex + 1; i++) {
36         if (i < 0 || i > game.gBoard.length - 1) continue;
37 
38         for (let j = jIndex - 1; j <= jIndex + 1; j++) {
39             if (j < 0 || j > game.gBoard[0].length - 1) continue;
40             let curCell = game.gBoard[i][j];
41 
42             if (curCell.isShown || curCell.isMarked) continue;
43 
44             if (curCell.isMine) render.renderCell(i, j, MINE);
45             else if (curCell.minesAroundCount) render.renderCell(i, j, curCell.minesAroundCount);
46             else render.renderCell(i, j, ' ');
47 
48             let eCell = document.querySelector(`.cell${i}-${j}`);
49 
50             eCell.classList.add('hinted');
51 
52             setTimeout(render.renderCell, 1000, i, j, ' ');
53             setTimeout(this.removeClass, 1000, i, j, 'hinted');
54         }
55     }
56     this.gHintOn = false;
57 }
```

Figure 17

- Time complexity: O(1)
- Because it always runs area 3x3.
- First, in each cell in area 3x3, it will check if this cell is shown or marked, continue.
- Second, if this cell is the mine call renderCell() function to show mine cell.
- Third, if this cell has the number of mines around, call renderCell() function to show this number.

- Fourth, if this cell is empty, call renderCell() function to show ‘ ’ which is empty.
- Fifth, setTimeout to show this hint in 2 seconds.

- Sixth, handling if player click the mine.

```
// Click mine
else if (cell.isMine) {
    if (this.gGame.isSound) soundManage.bombSound();

    cell.isShown = true;

    if (this.gLevel.lives > 1) {
        this.gLevel.lives--;

        feature.gMoves.push(cell);
        render.renderCell(i, j, MINE);

        let remainLives = this.gLevel.lives;

        switch (remainLives) {
            case 2:
                gLives.innerText = '❤️ ❤️ 💔';
                break;
            case 1:
                gLives.innerText = '❤️ 💔 💔';
            default:
                break;
        }
        gPlayer.innerText = SAD;
        return;
    }
    else if (this.gLevel.lives === 1) {
        render.renderBombs();
        this.gameOver('lost');
        return;
    }
}
```

Figure 18

- Time complexity: O(1)
- First, check if the player allow the sound, play the bombSound().
- Second, changing the status for this cell is shown.
- Third, checking the remaining lives, if greater than 1, decreasing these lives by 1. Push all information of this cell to the gMoves, then display for player this is a mine, display the remaining lives.

- Forth, if the remaining lives equals 1, call renderBombs() function to show all bombs in this game. Then set the status of is game to “lost”.

```
renderBombs() {
    for (let i = 0; i < game.gBoard.length; i++) {
        for (let j = 0; j < game.gBoard.length; j++) {
            if (game.gBoard[i][j].isMine) this.renderCell(i, j, MINE);
        }
    }
}
```

Figure 19

- Time complexity: $O(n^2)$.
- Display this cell is mine.

```
gameOver(status) {
    timer.stop();
    this.gGame.isRun = false;

    if (status === 'win') {
        gPlayer.innerHTML = WIN;
        timer.setBestTime();
        if (this.gGame.isSound) soundManage.winSound();
    } else if (status == 'lost') {
        gLives.innerHTML = this.gLevel.size === 4 ? '💔' : '💔💔💔';

        gPlayer.innerHTML = LOSE;
        if (this.gGame.isSound) soundManage.loseSound();
    }
}
```

Figure 20

- Time complexity: O(1)
- First, stop the timer, change the status isRun to false.
- Second, checking the status, if ‘win’. Display this status to the player, then call setBestTime() function and play the win sound.
- Third, if the status is ‘lost’. Change the lives displays, and the status display, and play the lost sound.

- Seventh, if this cell is not mine.

```
158 // Main
159 if (this.gGame.isRun) {
160     if (cell.minesAroundCount > 0) {
161         thisCell.innerHTML = cell.minesAroundCount;
162         feature.gMoves.push(cell);
163     } else {
164         render.renderNeighbors(i, j);
165         feature.gMoves.push(feature.gNrMoves.slice());
166         feature.gMoves[feature.gMoves.length - 1].push(cell);
167         feature.gNrMoves = [];
168     }
169 
170     thisCell.classList.add('pressed');
171     cell.isShown = true;
172     if (this.isWin()) this.gameOver('win');
173 }
```

Figure 21

- Time complexity is O(1).
- Checking the mine around count of this cell, if greater than 0, display this to player and push the information of this cell to gMoves.
- If the mine around count is 0, call renderNeighbors() function to display the neighbors until it reach the cell that has the mine around count. After that, push all neighbors cell to gMoves and the last position is this clicked cell.

```
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 22

- 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.

7. **Handle right clicking:**

```
176 handleFlag(i, j) {
177     let cell = this.gBoard[i][j];
178 
179     if (!this.gGame.isRun || cell.isShown && !cell.isMine) return;
180 
181     if (!cell.isMarked) {
182         render.renderCell(i, j, FLAG);
183         cell.isMarked = true;
184         if (this.isWin()) this.gameOver('win');
185     } else {
186         if (cell.isMine && cell.isShown) return;
187         render.renderCell(i, j, ' ');
188         cell.isMarked = false;
189     }
190 }
```

Figure 23

- Time complexity is O(1).
- If this clicked cell is not marked, display this cell with flag, change the status isMarked, checking is this marked reaching the wining requirement or not.

```
208 isWin() {
209     for (let i = 0; i < this.gBoard.length; i++) {
210         for (let j = 0; j < this.gBoard.length; j++) {
211             let cell = this.gBoard[i][j];
212             if (cell.isMine && !cell.isMarked) return false;
213             if (!cell.isMine && !cell.isShown) return false;
214         }
215     }
216     return true;
217 }
```

Figure 24

- Time complexity is $O(n^2)$.

- It will scan all cells in this board, if the cell is the mine but is not marked, return false; if the cell is not mine but is not shown, return false. After all looping, if nothing change return true.

- If this clicked cell is marked, addition algorithm to check if this cell is mine and is shown, do not do anything; then render this cell to remove the flag and change the status isMarked to fasle.

8. **Restart function:**

```
restart() {
    timer.stop();
    this.gGame.isRun = false;
    this.gGame.isFirst = true;
    this.gGame.shownCount = 0;
    this.gGame.markedCount = 0;
    this.gGame.secsPassed = 0;
    document.querySelector('.timer').innerHTML = '00:00';
    this.gLevel.hints = this.gLevel.size === 4 ? 1 : 3;
    this.gLevel.lives = this.gLevel.size === 4 ? 1 : 3;
    this.gLevel.safeClicks = 1;
    feature.gMoves = [];
    feature.gNrMoves = [];
    document.querySelector('.best-time').innerHTML = '';
    this.init();
}
```

Figure 25

- Time complexity is O(1).
- Stopping the timer.
- Change all information of this game to default.
- Display the timer to default.
- Display the features to default.
- Empty the stack.
- Remove the best time.

9. Undo feature:

```
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);
}

reverseMove(cell) {
    cell.isShown = false;

    if (cell.isMine) {
        game.gLevel.lives++;
        gLives.innerHTML = gLives.innerHTML === '💔💔💔' ? '💔💔💔' : '💔💔💔';
    }

    document.querySelector(`.cell${cell.position.i}-${cell.position.j}`).classList.remove('pressed');

    render.renderCell(cell.position.i, cell.position.j, '');
}
```

Figure 26

- Time complexity is O(n)
- First, if the previous move change status for more than one cell, run the loop for each cell change and call reverseMove() function for each cell.
- Second, if the previous move just change the status of one cell, call reverseMove() function for this cell.
- The reverseMove() function change the status isShown to fasle, if this cell is mine increase the lives and display it to player. Remove the class ‘pressed’ and render this cell to empty.

10. Show safe cell feature:

```
showSafeCell() {
    if (!game.gGame.isRun) return;
    if (game.gLevel.safeClicks === 0) return;

    game.gLevel.safeClicks--;
    let remains = game.gLevel.safeClicks;

    gSafe.innerHTML = remains === 1 ? '🔍' : remains === 2 ? '🔍🔍' : '❌';

    let safeCells = [];

    for (let i = 0; i < game.gBoard.length; i++) {
        for (let j = 0; j < game.gBoard[0].length; j++) {
            let cell = game.gBoard[i][j];
            if (cell.isMine || cell.isShown) continue;
            safeCells.push({ i: i, j: j })
        }
    }

    let randomIndex = safeCells[util.getRandomInt(0, safeCells.length)];

    document.querySelector(`.cell${randomIndex.i}-${randomIndex.j}`).classList.add('safe');

    setTimeout(this.removeClass, 2000, randomIndex.i, randomIndex.j, 'safe');
}
```

Figure 27

- Time complexity is $O(n^2)$.
- It will scan all the cell in this board, if the cell is mine or shown, continuously; if the cell is not mine or shown add this position to the safeCells array.
- Finally, random a position in safeCells and display it for the player, and setTimeout 2 seconds to hide this feature.

11. Show hint feature:

```
hintOn() {
    if (game.gLevel.hints === 0) return;
    if (!game.gGame.isRun) return;

    this.gHintOn = true;
    game.gLevel.hints--;

    if (game.gLevel.size === 4) {
        return gHints.innerText = 'Hint: ❌';
    }

    let remainHints = game.gLevel.hints;

    switch (remainHints) {
        case 2:
            gHints.innerText = 'Hints: 💡💡❌'
            break;
        case 1:
            gHints.innerText = 'Hints: 💡❌❌'
            break;
        default:
            gHints.innerText = 'Hints: ❌❌❌'
            break;
    }
}
```

Figure 28

- Time complexity: O(1).
- This is the algorithm to update the hint remaining for this game and display it for player.
- The algorithm to show the hint is described in **6. Handle left clicking.**

V. Implements

- The initialization of this game:

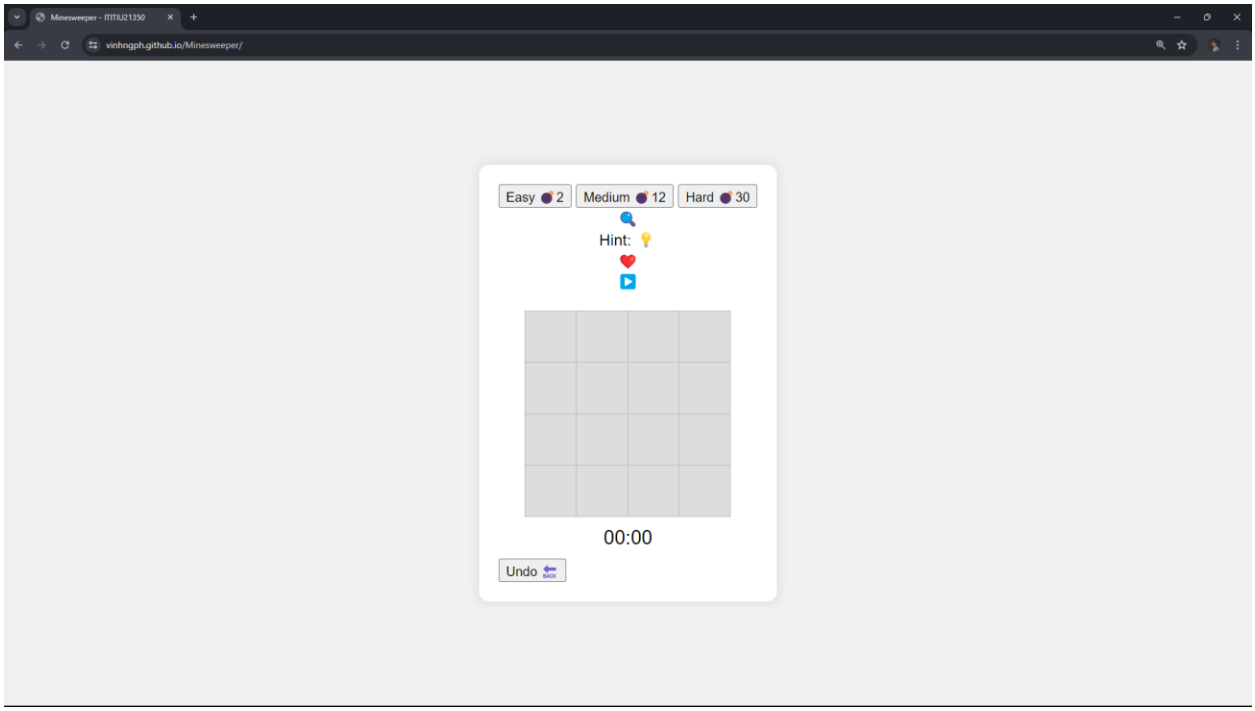

Figure 29

- Start the game:

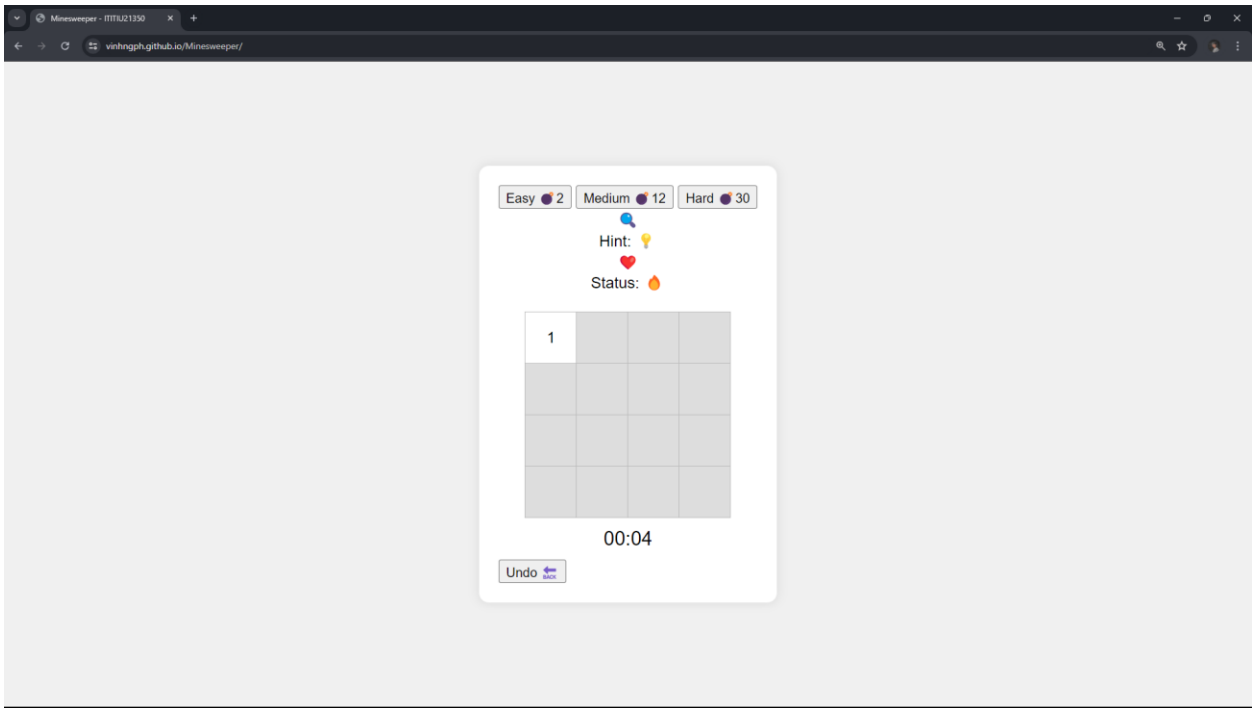

Figure 30

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

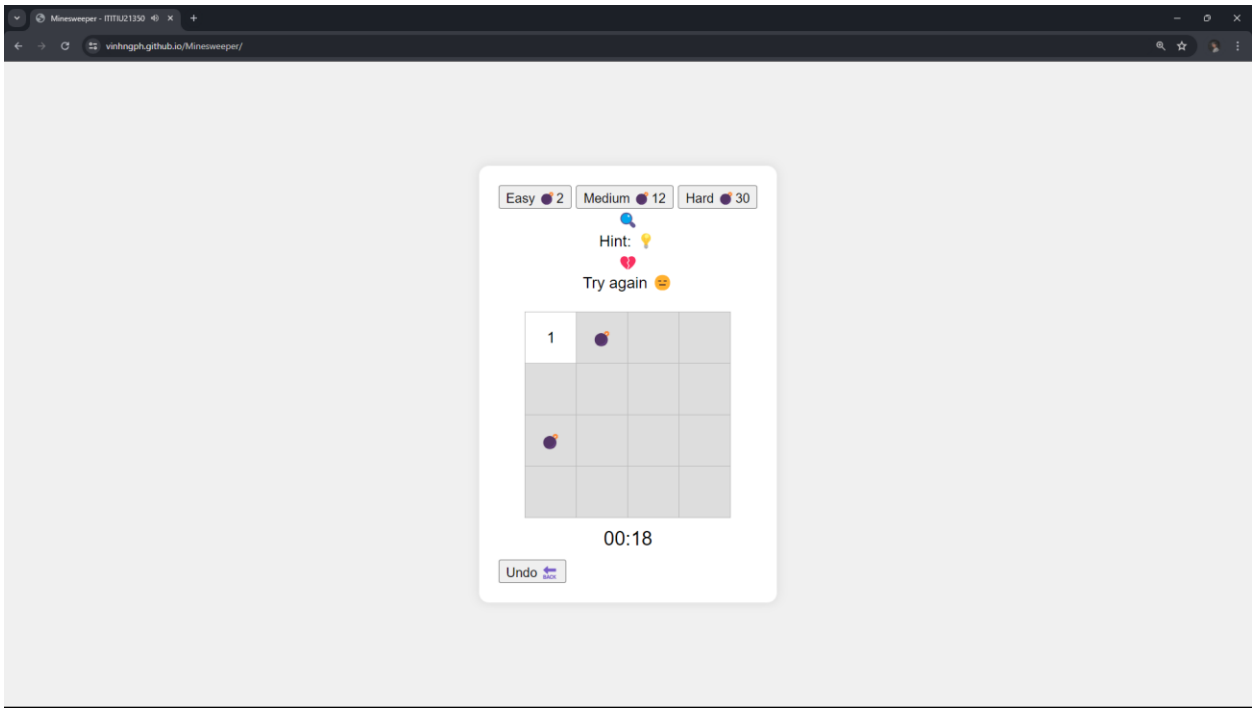

Figure 31

- Click "Try again 😑" to continue:

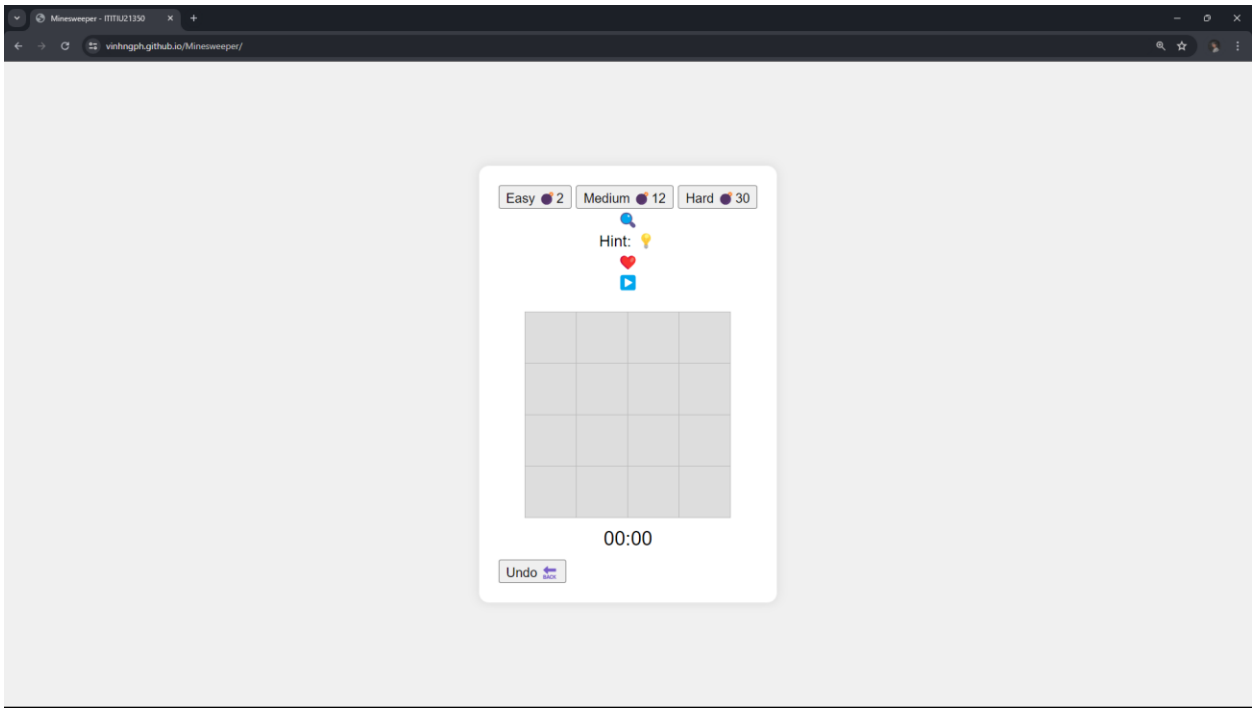

Figure 32

- 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:

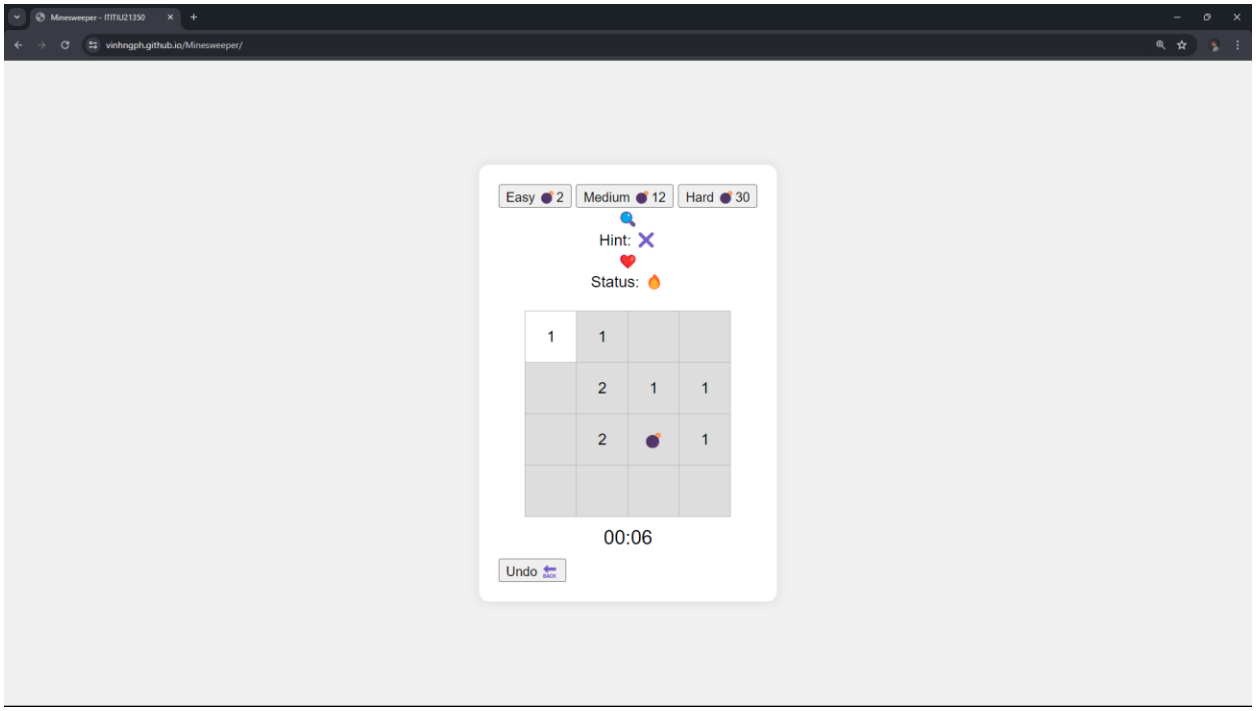

Figure 33

- 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:

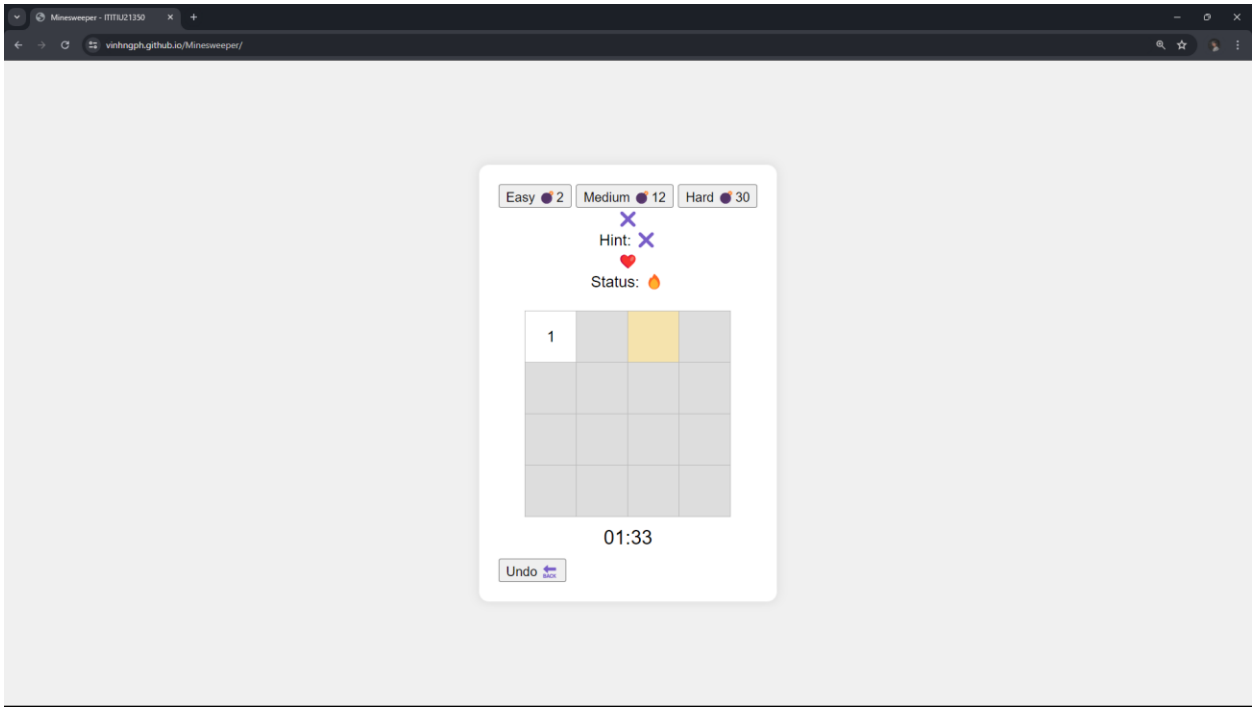

Figure 34

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

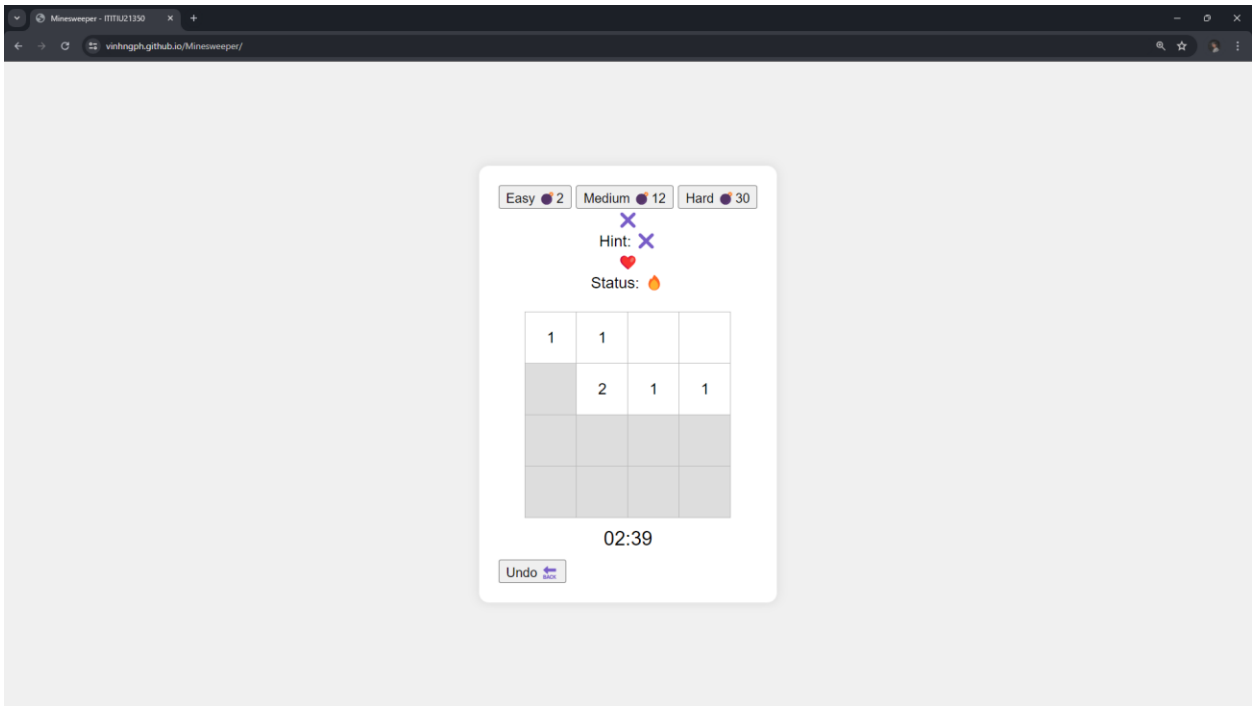

Figure 35

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

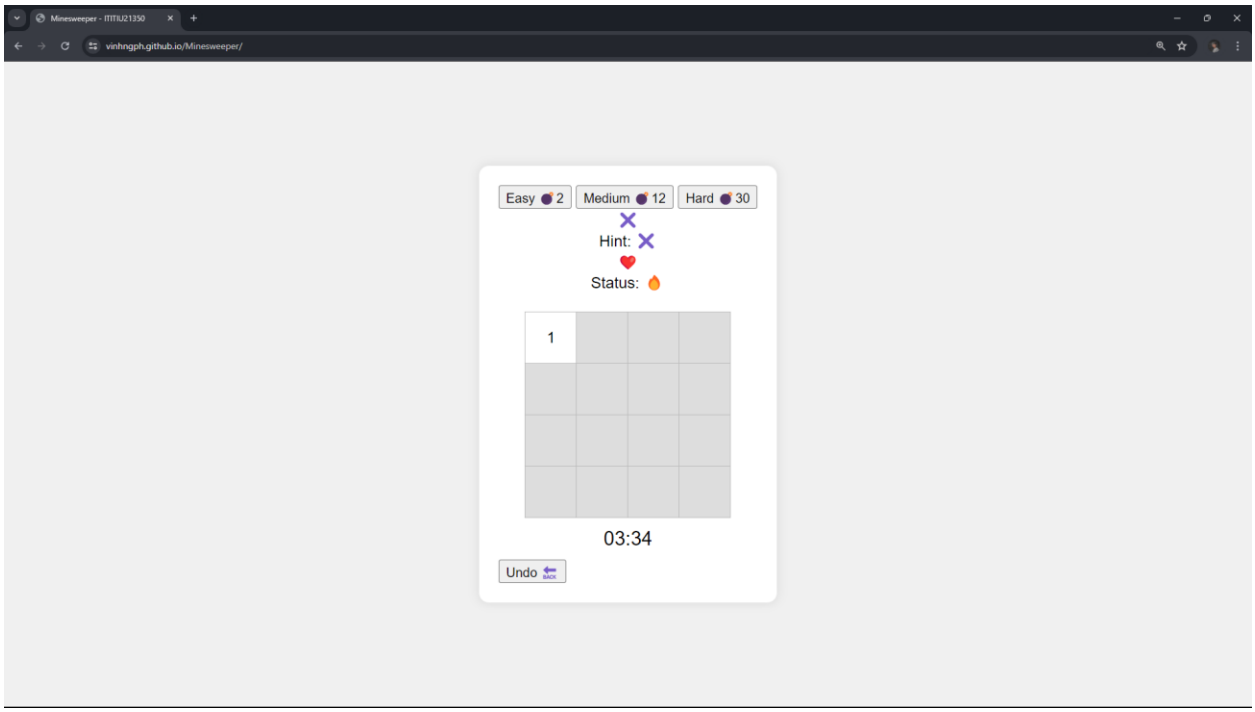

Figure 36

- 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:

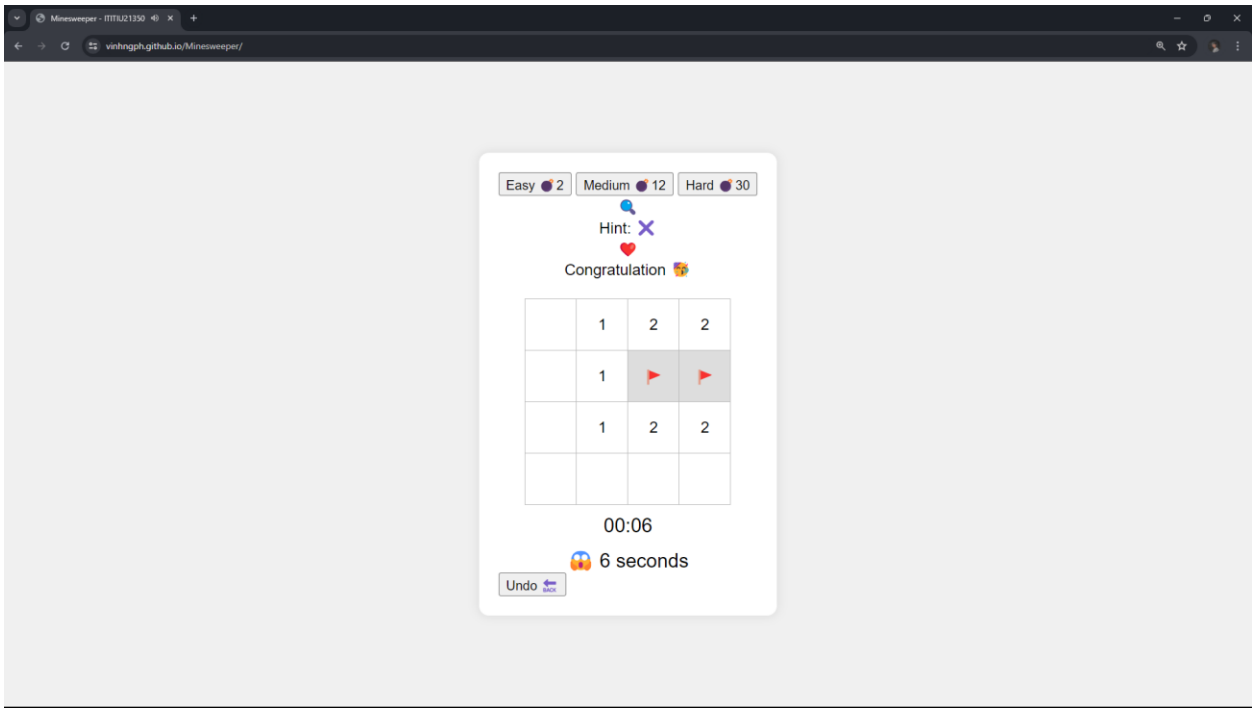

Figure 37

- 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”:

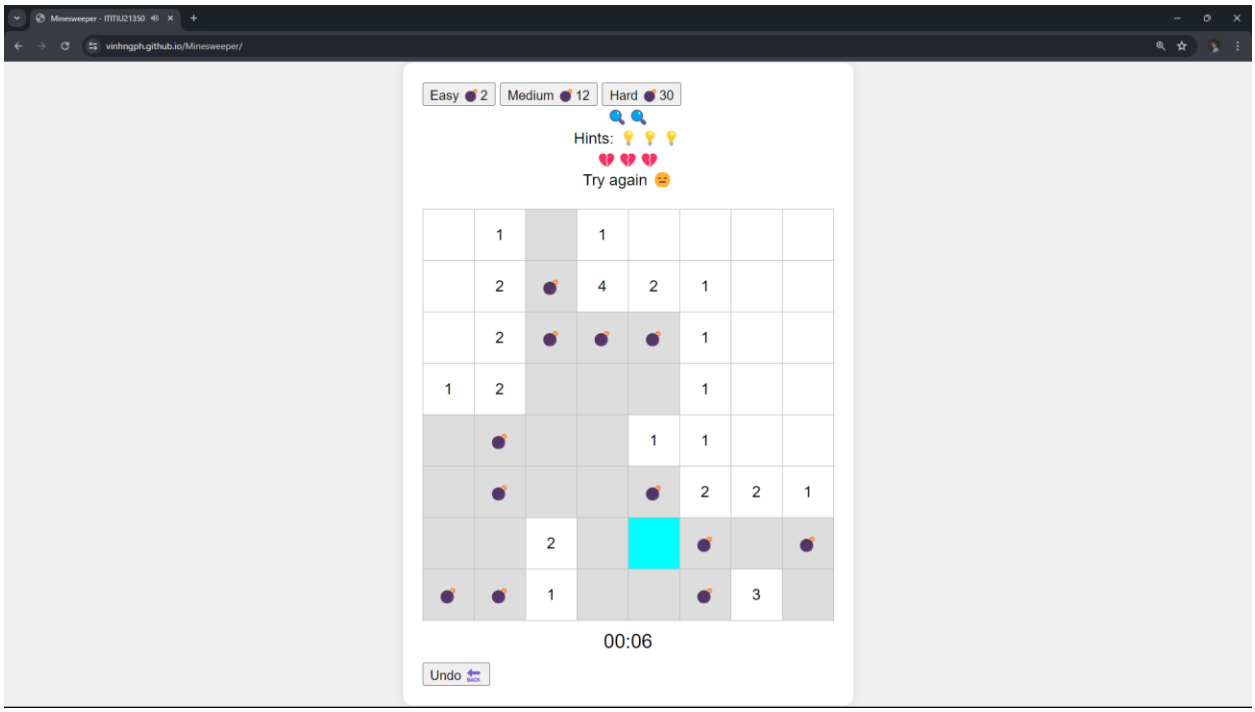

Figure 38

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

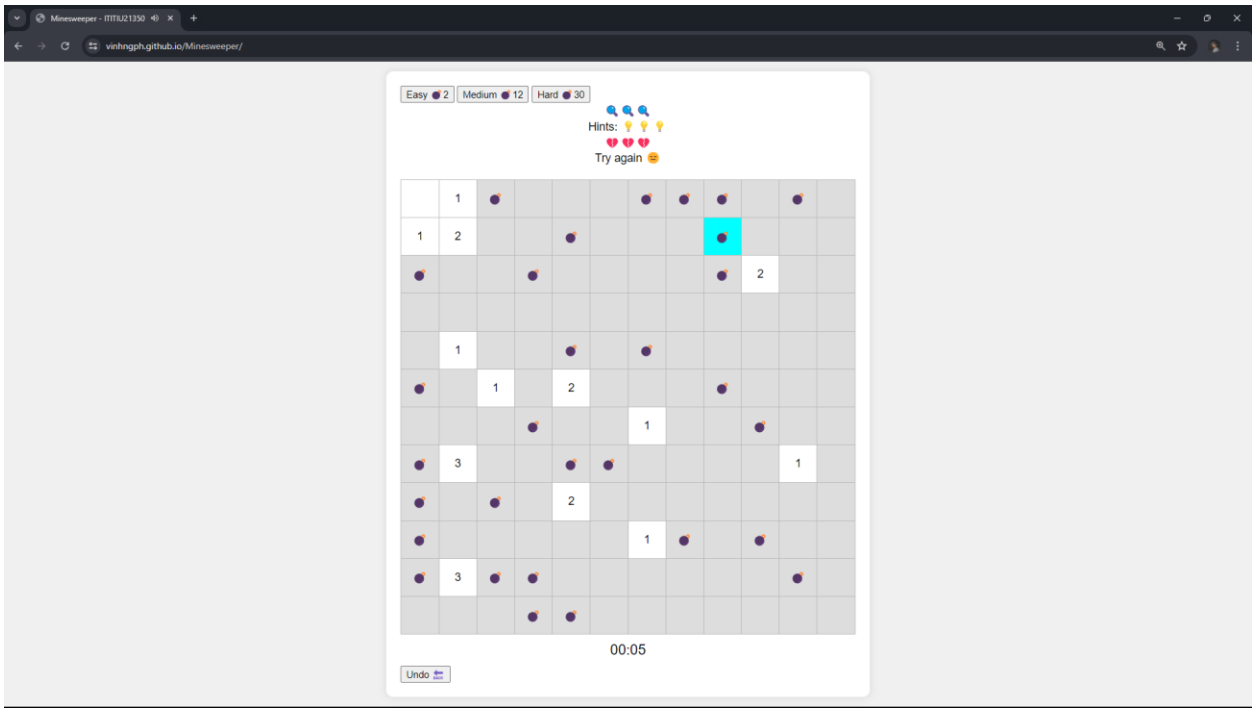

Figure 39

- 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.