MINESWEEPER SIMULATOR REPORT

Members:
Nguyen Minh Trang - ITDSIU19020
Phan Vo Phuong Tung - ITDSIU19025
Le Thi Thu Tra - ITDSIU19058
Tran Huu Phuc - ITDSIU19013

Github link: https://github.com/nmtrang/minesweeper

THIS PAGE IS INTENTIONALLY LEFT BLANK

TABLES OF CONTENT

I. INTRODUCTION

1. Brief introduction to Minesweeper game

Minesweeper is a single-player puzzle video game. The objective of the game is to clear a rectangular board containing hidden "mines" or bombs without detonating any of them, with help from clues about the number of neighboring mines in each field. The game originates from the 1960s, and it has been written for many computing platforms in use today. It has many variations and offshoots.[1]

2. Brief introduction to Minesweeper Simulator project

As described in the section above, the team members had been working on this project in order to rebuild the game based on its rules. With the help of Data Structures and Algorithms knowledge, we were able to establish the game from scratch using some of the most common data structures and algorithms.

II. OBJECTIVES

1. Objectives with respect to the game

By walking through and analyzing the game, we were able to break the big picture into small sections and examine each of them. This has helped us to understand why and how the creators come up with the first draft and build up a whole smooth and logical game flow from the ground up.

2. Objectives with respect to the course

With the understanding of data structures and algorithms, we could spend time analyzing bits of the segment of the game and reasoning up from there to build the whole game. Therefore, this project helps us to fully understand the importance of Data Structures and Algorithms in various

[1] Wikipedia: Minesweeper (video game)

Computer Science fields. Specifically, how each piece of algorithm works, why it works that way and when to use it to give the most optimal solution to multiple problems.

III. CONTRIBUTIONS

Member	Main tasks
Nguyen Minh Trang	Leading, keeping track of individual's tasks; design classes; analyzing algorithms; writing report
Phan Vo Phuong Tung	Create main user interface; design classes; analyzing algorithms
Le Thi Thu Tra	Design classes; observing and giving solutions to fix bugs; coming up with new feature to the game
Tran Huu Phuc	Observing and giving solutions to fix bugs; coming up with new solution to write algorithms

IV. DEVELOPMENT

1. What is the big picture?

Play the game online along with the big picture explanation
First off, let us walk through the game rule:

The rule of the game is simple, the number on a block shows the number of mines adjacent to it and you have to flag all the mines.

Gameplay:

The objective is to clear all the "mines" or bombs from a 16x16 minefield. There are 40 in total.

To get data on where the bomb is, left click to reveal the cells. A cell with a number uncovers the number of neighboring cells (the 8 most coordinate ones encompassing it) containing bombs, in spite of the fact that itself will not contain a bomb. A cell that does not contain a bomb in its coordinate neighbor cells (the 8 most coordinate ones encompassing it) is a purge cell, and when clicked on, will uncover the whole locale of all purge cells until a cell with a number shows up. Utilizing this data also figures work to dodge the bombs.
To mark a cell you think may be a bomb, right-click on the cell and a flag will appear. You've got 40 flags added up to, one for each bomb. You'll be notified after you have utilized all your flags with a number of how numerous banners you have got cleared out within the lower cleared out corner.
The game is won when the player has successfully identified all the cells that contain bombs and the game is lost when the player clicks on a cell which contains a bomb.
The player can “unflag” a cell by right clicking the cell again. The player can undo any number of moves for any type of move, which includes clicking on flagged cells, empty cells, and neighbor cells.

2. Analyzing and planning

By fully grasping the game rule, we came up with what is needed for the simulator creation.

First off, the main GUI for the players, it should contain resources like images such as numbers, flag, bomb, wrong flag marked, empty cell, mine cell; buttons with some click events as *“what will happen if the player left-click at a random cell, what will happen if right-click is triggered, what will pop up, what is the condition to win the game and what is not?”*

Secondly, because we had analyzed the action involved in the game as mentioned above, we thought of these must-have functions: `undo()` and `find_empty_cells()`. Meanwhile, those remaining functions will be added during the coding time.

3. **Classes design** (full UML view)

So, how did we design and organize classes?
Depending on every objects in the game, we extracted the most essential elements in the game and packaged them into these classes:

a. `Game.java`
- This is the entry point and UI design of the game
- It is built with the layout of the Java Swing
- It handles basic user input, specifically, clicking
- It contains some action events involved in the game (mostly click events)

Note: Given that the grid consists of individual cells, all of which can be thought of as objects, with each cell sharing certain basic properties/functionalities so it will make sense to do some type of class inheritance in the program.

b. `Cell.java` (parent class)
- This is where we store properties of the most basic cell like whether it's covered, flagged or what type of that cell: is it BOMB cell? Is it BOMB NEIGHBOR cell? Or is it EMPTY cell?

b.1. `BombCell` (extends Cell)
b.2. `NeighborOfBombCell` (extends Cell)
b.3. `EmptyCell` (extends Cell)
- How is this `EmptyCell` initiated? It has 2 ways of creation:
  - Whenever the game starts to begin, we have all the empty cells, of course
  - When the player starts to play the game, some cells will be covered by some sort of numbers or flagged, some other cells will be left as remaining empty cells (outside of bomb ones).

c. `Board.java`

- Combines every different small classes into one big view and is used as a main controller.
- Handles user actions (either left or right click) and performs or displays desired moves/information accordingly using paint and event listeners
    - Stores all images of these different types of cells in a map and displays images according to cells clicked
    - Contains **recursive function** to clear a bomb-free region of empty cells and is called when user clicks on an empty cell
- Stores user moves whenever user clicks the board, and handles the "undo" functionality whenever the user desires to remove a move

d. `CellType.java` and `ImageName.java` (enums)

- Enums used to represent the different types of cells and images in a cleaner, more readable way
- CellType has `Bomb, BombNeighbor, Empty`
- ImageName has `Empty, Covered, Marked, Wrongmarked, Bomb`

e. `Input.java and CustomMap.java`

- Has two variable: size of map and number of mines
    - If the user choose 8x8 map: return a map with width and height is equal 8 and the number of mines is 10
    - If the user choose 16x16 map: return a map with width and height is equal 16 and the number of mines is 10
    - If the user choose Custom (new feature):
        - Open the JFrame CustomMap: let the user input the value of size (width size always equal height size) and input the number of mines
        - Cannot set mines equal 0
        - Cannot set set size lower than 6

Some terms:

Neighbor bomb cells: 

Bomb cell: 

Empty cell: 

Covered cell: 

Flag: 

Wrong-marked Flag: 

4. Algorithms implementation

a. Key design concepts

- **2D Array**

Since minesweeper takes place on a 16x16 grid (default), we use a 2D array of "Cell" objects, which we think of as a grid, to store the state of each cell in that grid, in relation to which cells are bombing included when empty or adjacent cells (ie "number" cells). Cell was a parent class we created that stores the properties of each cell and handles basic cell behavior based on its type. It is used every time a new game is started in which the 2-D matrix is randomly filled with bomb cells, number cells (cells that are neighbors to the bomb cells that tell how many bombs are nearby) or empty cells . Processing different user actions for a particular cell by using a double for loop representing the rows and columns of the 2D array, or calling individual elements of the 2D array to initialize cell objects. Our use of a 2D matrix was extremely helpful in organizing my entire game board structure and finding user actions indicating the desired locations to handle the logic of the game.

```
protected static Cell[][] gameBoard;
```

```
gameBoard = new Cell[n_size][n_size];
```

- **Collection**

Players can "undo" their moves when they hit the "undo" button, unless hitting a bomb, which then the game ends. We fully take advantage of the LIFO property of Stack so it would be best for us to implement that principle to store the players' move in order to undo the most recent step by popping it out of the Stack. In other words, removing the player's move from Stack and returning the state of the cell depending on the player's last click. The other function of Stack - `push()` is also used to add steps to the Stack whenever the player left-click a number of empty cells, and right-click to flag/unflag a cell, which shows the player's most recent move. The player can undo any number of moves for any type of move, which includes clicking on flagged cells, empty cells, and neighbor cells, but for bomb cells, the `undo()` function is disabled (because that is plainly cheating, of course!) by "clearing" the Stack.

```
private Stack gameSteps = new Stack();
```

- **Recursion**

We utilize recursion to reveal and clear a mine-free region of cells. In other words, when a player clicks on a cell that has no bombs with adjacent cells also bomb-free so all adjacent cells are automatically cleared and the region with bomb-free cells is revealed up until the cell that is connected to a bomb.

b. Algorithms explanation

- `find_empty_cells()`

Our aim is to find neighboring empty cells of a cell by giving its x and y position which is the cell that the player left-click.

- First off, we need to mark that cell to “not-being-covered-anymore” state (this is obvious!) with the method of a cell - `flipUp()`. From this situation, let’s forget the fact that the clicked cell might be a bomb cell. We are just talking about the situation where there are adjacent empty cells.

```
gameBoard[x][y].flipUp();
```

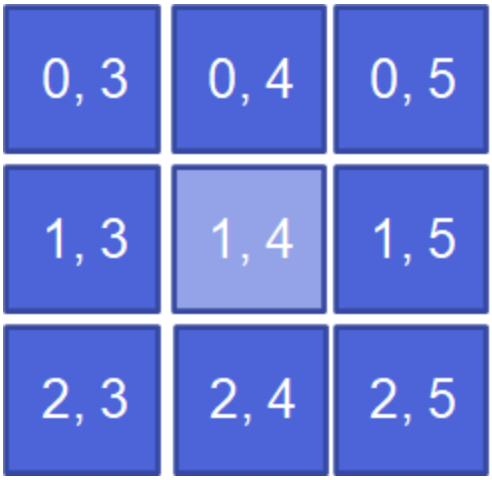
- Secondly, we need to store or push that step into Stack

```
gameSteps.push(x * n_size + y);
```

Inside `Cell.java`, we had initially created a function called `flipUp()` just to set the state of a cell to “not-being-covered”. So when we call that function inside of `find_empty_cells()`, we want to set the state of that specific x and y that was parsed in the parameter to what has just been mentioned above.

```
public void flipUp() {
   this.isCovered = false;
}
```

Inside the for loop, it will find all the neighboring cells of that empty cell.
For example: we have a current cell that has x = 1 and y = 4, it will check all neighbor cell of 1,4 like this:



For each cell, it will get the cell type and put it in `typeOfCell` with `getCellType()`. After that, it will check if the cell type is Empty, Covered and is not Marked by flag. If true, it will continue to find the neighboring empty cells of that covered empty cell. This algorithm utilize the power of recursion

```
public void find_empty_cells(int x, int y) {

   gameBoard[x][y].flipUp();
   gameSteps.push(x * n_size + y);
   for (int dx = -1; dx <= 1; dx++) {
     for (int dy = -1; dy <= 1; dy++) { //set bounds
        if ((dx != 0 || dy != 0) && x + dx < n_size && y + dy <
n_size && x + dx >= 0 && y + dy >= 0) {

           CellType typeOfCell = gameBoard[x + dx][y +
dy].getCellType();

            if (typeOfCell == CellType.Empty && gameBoard[x +
dx][y + dy].isCoveredCell() && !gameBoard[x + dx][y +
dy].isMarkedCell()) {
               find_empty_cells(x + dx, y + dy); // recursive
            }
        }
```

```
      }
    }


 }
```

- undo()

Let's not consider the situation that there is no step stored in the gameStep Stack. In the other situation, our goal is to pop out the top most layer of the stack which represents the most recent step.

- First off, pop it out and store it inside a new integer.
- Secondly, load the most recent step's cell according to that new integer and the size of the cell.

```
if (!gameSteps.empty()) {
   int i = (Integer) gameSteps.pop(); //gets most recent game step

   //corresponding cell to the game step
   // this also holds the position of the last-clicked cell
   Cell cell = gameBoard[i / n_size][i % n_size];
```

- Thirdly, the situation is divided into two:

  - What if the corresponding cells are flagged and covered?

In this situation, we just simply reduce the remaining number of flags shown in the status bar if the state of that cell is "already-marked". Otherwise, add the number up.

```
//Handle flagged cells situation, which are covered
   if (cell.isCoveredCell()) { // if cell is covered
```

```
      cell.changeWhetherMarked();
      if (cell.isMarkedCell()) {
         minesLeft--;
       } else {
            minesLeft++;
            if (!inGame) {
               inGame = true;
            }
       }
   } else if (cell.getCellType() == CellType.BombNeighbor) {
        cell.isCovered = true;
   }
```

- What if the corresponding cells are empty ones?

Clearly, before the player clicks that cell and finds out there are a whole bunch of empty cells popping up, it is a covered cell so we must reset its state first.

- While the stack still holds steps, we process it. In order to load the next previous step, we did as mentioned above, in this case, `cellNext`
- We have 2 situations going on here: that cell is a bomb neighbor cell or another empty cell. Through pseudo-code, we can easily handle this.
    - If it is a `BombNeighbor` cell, push it back to the step because we don't want to process it until we hit the *undo* button again.
    - If it is another empty cell, simply reset its covered state like we previously did to it.

Note: But why is this important? Why do we have to check for the `BombNeighbor` cell and re-check the `Empty` cell? Visualizing this makes it better to explain.

Suppose we had played 2 moves, we first received **1 `NeighborBomb` cell** and **3 `Empty` cells**. The gameSteps Stack can be visualized like this:

Order of pushing in	Position (x,y)	Cell type
4	(5,9)	Empty
3	(5,8)	Empty
2	(5,7)	Empty
1	(2,2)	NeighborBomb

Therefore, if we don't re-check the empty cell, whenever we want to undo all the empty cells, we have to click undo n number of times corresponding to n number of empty cells in order to undo all of those, rather than **clicking once and erasing all the empty cells.**

```
   if (cell.getCellType() == CellType.Empty) {
      cell.isCovered = true;
      while (!gameSteps.empty()) {
          int j = (Integer) gameSteps.pop();
          Cell cellNext = gameBoard[j / n_size][j % n_size];
          if
(cellNext.getCellType().equals(CellType.BombNeighbor)) {
             gameSteps.push(j);
             break;
          } else {
               cellNext.isCovered = true;
          }
      }
   }
```

V. SELF-EVALUATE

1. Strengths:

- Members agree on and set team goals based on outcomes and results, rather than just on the amount of work being done. A clear plan can then be set about how the members are going to achieve these objectives, as a group, as well as each individual's contribution. This provides the team with clear direction and gives the team something to aim for collectively.
- Team members are always happy to assist others when they need a helping hand with work.
- Everyone is unique and will be able to offer their own experiences and knowledge that others may not possess. So, members help each other a lot in designing the realistic database, as well as the coding part.

2. Weaknesses:

- Some first few weeks were chaos in organizing and planning the project because some tasks might clash with each other, which leads to disconnected pieces of work.
- Some members do not know how to use git correctly and effectively. This affects not only other people but also the whole working style such as sending code outside of git/github, different variable/method names, wrong indentation, messy excessive code.
- Coding convention is inconsistent. Some members like naming things depending on whether they think it is beautiful. This broke the whole project or at least, code files do not seem like they are connected.

VI. CONCLUSION

- The project helps us know how to build a proper game with add-on features like undo… Also how to apply recursion, linked list into classes. This is not our perfect application and we have to improve many more in the future.
- The project was designed simple, with just letting the user choose the size of the board and play the game based on their choice.

What needs to be improved?

- More advanced features other than undo and choose menu like score system, time system.
- Knowing how to apply some design patterns into the code
- Knowing how convert the Java code into Javascript
- Knowing how to use Github more flexible

VII. REFERENCES

https://www.w3schools.com/java/java_recursion.asp
https://www.geeksforgeeks.org/stack-class-in-java/
https://www.microsoft.com/vi-vn/p/microsoft-minesweeper/9wzdncrfhwcn?activetab=pivot:overviewtab
https://www.instructables.com/How-to-play-minesweeper/
https://minesweeper.online/game/697203744
https://github.com/nmaguirre/minesweeper
https://stackoverflow.com/questions/32735206/java-png-exception

THE END