Fossil Sweeper Lab

Let’s do some recursion

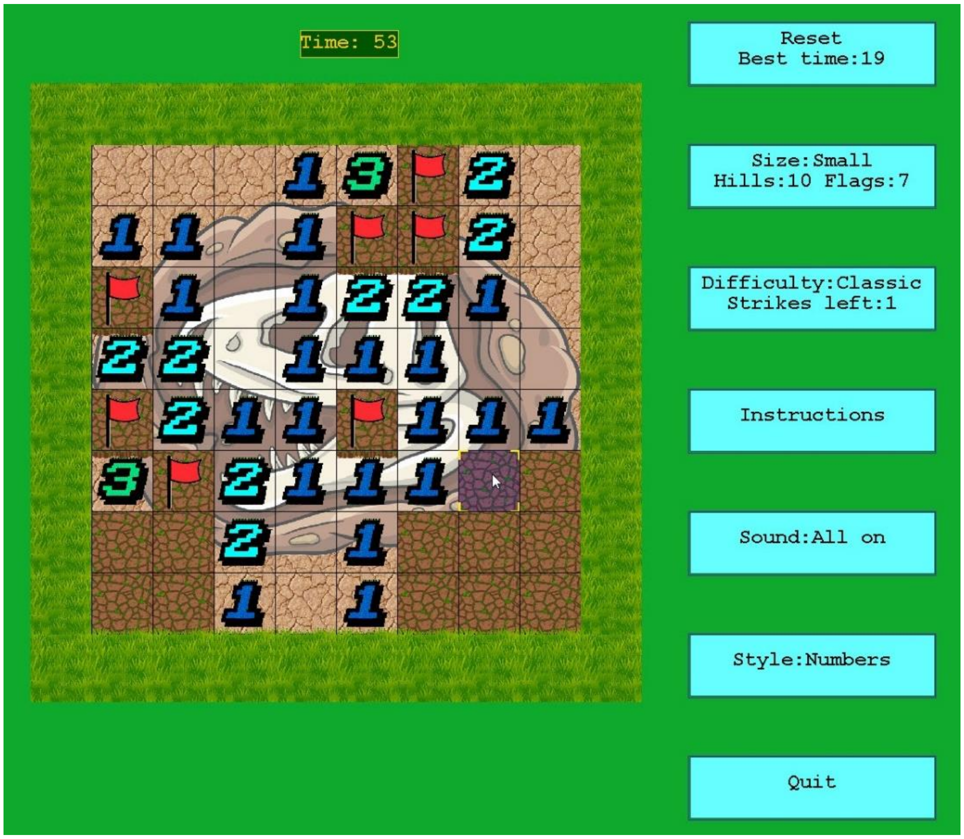
Fossil Sweeper is a clone of a classic mine sweeper puzzle game.

Game Instructions:

- Uncover as much of the fossil as possible without disturbing the ant hills.
- Left mouse-click or SPACE to uncover a plot of land. The number of ants seen represent the number of adjacent plots that contain an ant hill.
- Uncovering a plot of land with an ant hill will disturb the ants.
- Right mouse-click or ENTER to mark a suspected ant hill with a flag.

Win the game by successfully marking all ant hills without disturbing them.

The difficulty option grants you several strikes: these represent the number of ant hills you can disturb before losing the game. You can set the rules to a classic mine-sweeper game by setting the difficulty to Classic, which only allows for one strike and reveals one open spot for you on the game board.



Time: 53

Best time:19

Size:Small
Hills:10 Flags:7

Difficulty:Classic
Strikes left:1

Sound:All on

Style:Numbers

Student assignment:

Most of the game is done, but there are two methods within `Utilities.java` that need completion – `populateBoardValues` and `revealEmpties`:

`void populateBoardValues(Button [][] gameBoard)`

The argument `gameBoard` is a 2-D array of Button objects. Each button has a data-field called `numAnts`. If `numAnts` is storing the value `GamePanel.ANTHILL` (9), it represents an anthill, or a mine in traditional minesweeper. Otherwise, `numAnts` should store an integer that represents the number of adjacent cells that have an anthill: something between 0 and 8.

Assume `gameBoard` is not null and is populated with buttons whose `numAnts` data-field is either set to 0 or `GamePanel.ANTHILL` (9).

For each cell in `gameBoard` that does not contain 9, this method will set the `numAnts` data-field in that cell to an integer that represents the number of adjacent cells for which the `numAnts` value is `GamePanel.ANTHILL` (9). In other words, for each cell that does not contain an anthill, this will change the value to the number of adjacent cells that are anthills.

Given a buttons array that is passed in with the following values for `numAnts`:

0	0	9	0
0	9	0	0
0	0	9	9
0	0	0	0

will change the array to this:

1	2	9	1
1	9	4	3
1	2	9	9
0	1	2	2

With this example, note that the cell at (row:0, col:1) will have its `numAnts` data-field set to 2 because there is an anthill to its right at (0, 2) and another one beneath it at (1, 1). Likewise, for the cell at (1, 0), it will have its `numAnts` data-field set to 1 because it is only adjacent to one anthill, which is to its right at (1, 1).

`void revealEmpties(Button [][] gameBoard, int row, int col)`

Assume `gameBoard` is not null, and row and col are valid indices of `gameBoard`.

The idea of this method is that the client has clicked on a cell at (row, col). If that button is a cell that is not adjacent to any anthills, then we need to open-up the board by revealing every adjacent space next to (row, col) that are also not adjacent to any anthills. These are buttons for which their `numAnts` data-field is storing zero (0), which we will call these "zero-cells".

Likewise, any adjacent cell next to a zero-cell should also be revealed. Given a Button object called b, we can reveal it by calling `b.setClicked(true);`

This should only be done for cells that have not already been clicked. This can be seen by calling `b.hasBeenClicked()` which returns a `boolean` (true or false).

Let’s say we have a board with the following values for `numAnts`:

```
0 1 1 1 0 0
0 2 9 2 0 0
0 2 9 2 0 0
0 1 1 2 1 1
0 0 0 1 9 1
0 0 0 1 1 1
```

There are anthills for each cell where `numAnts` stores a 9
The other values represent the number of adjacent cells with an anthill.

So the anthills are at `(1,2)`, `(2,2)` and `(4,4)`.

Initially, assume that no cells have been clicked on (not revealed to the client). We might visualize that as a board of all “X” values.

```
X X X X X X
X X X X X X
X X X X X X
X X X X X X
X X X X X X
X X X X X X
```

The client will click on this green cell at `(0,4)`

If we were to call `revealEmpties(gameBoard, 0, 4)`, the client has clicked on the cell at row:0, col:4. Since it’s `numAnts` data-field contains a zero, we want to call `hasBeenClicked` at all of the zero-cells that are around it, as well as the non-zero cells that are adjacent to the zero-cells we find in that area.

```
X X X 1 0 0
X X X 2 0 0
X X X 2 0 0
X X X 2 1 1
X X X X X X
X X X X X X
```

Note: there is a compact recursive solution for the `revealEmpties` method. The efficiency might initially seem terrifying, but for the size of the boards we are working with it will not pose a problem.

To help with the process of opening up all adjacent cells at a particular (row, col), you are provided with a helper method:

```
//pre:  gameBoard!=null, row and col are valid indices of gameBoard
//post: for any unclicked space adjacent to (row, col) that does not
//      contain zero, reveal it
void revealAdjacentSpaces(Button[][]gameBoard, int row, int col)
```

This can be called to good effect with a recursive solution that is structured like this:

```
public static void revealEmpties(Button[][]gameBoard, int row, int col)
{
   //terminating cases go here

   gameBoard[row][col].setClicked(true);
   //reveal any adjacent space next to a 0-space

   revealAdjacentSpaces(gameBoard, row, col);
   //recursive calls go here

}
```

To test your `revealEmpties` method, run the driver program. You will see a button called `Difficulty`: if you select the option `Classic`, the program will seek out one large open area and reveal the spaces with your method. If you see an area of zero-ant tiles open in addition to each of their adjacent spaces, you have the method done correctly.