CPSC 217 Assignment 3

Due: Friday June 7, 2024 at 11:55pm
Weight: 7%
Sample Solution Length: 135 lines, including some comments (not including the provided code)

Individual Work:

All assignments in this course are to be completed individually. Use of large language models, such as ChatGPT, and/or other generative AI systems is prohibited. Students are advised to read the guidelines for avoiding plagiarism located on the course website. Students are also advised that electronic tools may be used to detect plagiarism.

Late Penalty:

Late assignments will not be accepted.

Submission Instructions:

Your program must be submitted electronically to the Assignment 3 drop box in D2L. It is your responsibility to ensure that your file has been uploaded successfully before the deadline. Save the confirmation email that D2L sends to you when your submission has been received successfully. You don't need to submit SimpleGraphics.py or the image files -- we already have them.

Description

In this assignment you are going to implement portions of the classic game Minesweeper. While the game dates back to the 1960s, it can still be played today on a variety of websites. A brief description of the game appears in the following paragraph. A more complete description is available on Wikipedia.

Minesweeper is a single player puzzle game. The player is presented with a rectangular game board that is broken down into a grid. Each location in the grid can either be a mine or an open space. At the beginning of the game, all of the locations are covered, so the player doesn't know which locations are mines and which locations are open spaces. The object of the game is to clear all of the open spaces (by left clicking on them) without clicking on a mine. When an open space is uncovered it reports the number of mines that are present in the 8 immediately adjacent spaces. This gives the player information that helps them successfully clear the remainder of the board. A square that the player has determined to be a mine can be marked as such by right clicking on it.

By default, the game starts in "easy" mode which uses a board with 9 rows, 12 columns and 10 mines. Running the command `python mine_sweeper.py medium` will start a game with 10 rows, 20 columns and 35 mines. You can also start the game in its most difficult mode by replacing `medium` with `hard`. Finally, there is a test mode that starts the game with a small board that contains only one mine. Be sure to test your program for each game mode.

I have written the user interface code (graphics and mouse) for the game. Your task is to write the functions that implement most of the game logic. These functions are described in the following sections. Note that you must follow the implementation instructions exactly. If your function has a different name, takes a different number of parameters, or returns a different value then my code will not be able to call it successfully, and the game will not work.

Part 1: Checking for a Visible Space

In this implementation of the game a two-dimensional list is used to represent the game board. Each element in the list is a two-character string. The first character in the string will either be a `C`, indicating that the board space is covered, a `V`, indicating that the board space is visible, an `M`, indicating that the space is covered and the player has marked the space as a mine, or a `?`, indicating that the space is covered and the player has marked the space as unknown. The second character in the string will either be a space which indicates that the space is empty, or a `*` which indicates that the space is a mine.

The user interface needs to know which spaces are covered and which spaces are visible in order to draw the board correctly. To facilitate this, you need to write a function named `isVisible` (notice the use of a lowercase i and an uppercase V). This function will take the game board, a row, and a column as its only three parameters. It will return `True` if the position indicated is visible. Otherwise it will return `False` indicating that the position is covered.

Hint: A position is visible if the first character in the string at the row and column indicated is a `V`. Otherwise it is covered.

This should be a really short function. My implementation is only 3 lines of code.

Run the Minesweeper game after implementing this function. While the game won't work yet, my automated tests will give you feedback on whether or not you have this function working. Do not proceed to Part 2 until this function passes all of my tests.

Part 2: Checking for a Mine

Every time the player left clicks on the board the game needs to check and see if that location holds a mine or an open space. If the space holds a mine then the player loses. Otherwise the game needs to reveal that space (and possibly additional adjacent spaces, as discussed in Part 6).

Write a function named `isMine`. The `isMine` function will take 3 parameters: the game board, a row within the board, and a column within the board. The function should return True if the indicated location in the board holds a mine. Otherwise it should return `False`. Like `isVisible`, this should be an approximately three-line function.

This function is also tested by my code when the program starts. Do not proceed to Part 3 until this function passes all of my tests.

Part 3: Adjacent Mines

When a square is revealed (and it doesn't contain a mine) the game tells the player how many mines are located in its (up to 8) immediately adjacent neighbours. Write a function named `numMines` that performs this task. Your function will take the game board, row, and column as its only parameters. It will return an integer between 0 and 8 as its only result.

Make sure that your function correctly handles cases at the edge of the board. In particular, note that the corner spaces of the board only have 3 neighbours, and the non-corner edge spaces only have 5 neighbours. Also, don't count the space you are on – only count the neighbours. My implementation of the `numMines` function is almost 30 lines of code without any comments, but shorter implementations are possible with a little bit of creativity.

Tests will also be run on this function when the game starts. Do not proceed to Part 4 until this function passes all of the tests.

Part 4: Creating the Board

Write a function named `createBoard`. Your function will take 3 parameters: The number of rows in the board, the number of columns in the board, and the number of mines that should be placed at random locations on the board. Your function should return a two-dimensional list where every element is a 2-character string. The first character should be `C` in every location to indicate that the space is covered. The second character should be an `*` for the number of mines indicated and a space in all other cases. The mines should appear at random positions within the board. Your function should return the two-dimensional list as its only result.

Hint: I think that the easiest way to write this function is to begin by creating a board where every space is covered and empty, represented by the string `"C "`. Then replace some occurrences of `"C "` with `"C*"`, ensuring that you don't put two mines on top of each other. My implementation of `createBoard` is about 15 lines of code without any comments.

Once you have completed this function, the game board should draw and you should be able to reveal squares on it. However, the game won't be able to realize that you have won until you complete the `hasWon` function described in the next section, and the game will be much easier to play once you complete the `reveal` function in the final part of the assignment.

Part 5: Has the Player Won?

I have provided an implementation of the `hasWon` function that always returns `False`. You need to replace my implementation with a version that correctly determines whether or not the player has cleared all of the non-mine spaces from the board (in which case it should return `True`). It is actually easier to describe the situations where the user hasn't won. The user has not won if:

- There are any spaces on the board where the space has been marked as a mine but the space doesn't contain a mine
- There are any spaces on the board that are covered that do not contain a mine
- There are any spaces on the board that are marked as unknown, indicated by a `?`

If none of these situations apply then you should conclude that the user has won.

My implementation of `hasWon` is 10 lines of code (without any comments).

Once you have this function implemented the game should work insomuch as the player can win or lose, but it won't be fun to play because it's too much work to click on all of the blank spaces that aren't close to a mine. Implement the last function so that you have a better version of the game (and to finish the assignment).

Part 6: Revealing the Rest of the Board

In most implementations of Minesweeper clicking a location on the board that is adjacent to zero mines reveals all of the adjacent squares that can trivially to be determined not to contain a mine. While this functionality isn't necessary for the game, it makes it faster and more enjoyable to play.

I have provided an implementation of the `reveal` function that only reveals one square without revealing the appropriate neighbours. Your task is to replace my implementation of the function with an implementation that provides the better behaviour. The better behaviour can be accomplished using the following algorithm. If you prefer, you are also welcome to use a different algorithm (including a recursive solution), provided that it generates correct results. Regardless of the algorithm used, your implementation of `reveal` must continue to take to the game board, row, and column as its only parameters. The reveal function should not return a result.

```
If the square in question has any mines in its immediately neighbouring squares then
    Reveal the current square and do nothing else
Otherwise
    Create two new, empty, lists, one to hold row values and one to hold column values
    Add the row provided as a parameter to the row list
    Add the column provided as a parameter to the column list
    As long as there is at least one item in each list
        Remove an item, r, from the row list and the corresponding item, c, from the column list
        Make the location at row r and column c visible
        If location (r, c) is not adjacent to any mines
            For each of its (up to 8) neighbours
                If the neighbour is currently covered
                    Add the neighbour's row to the row list and its column to the column list
```

My implementation of this algorithm was approximately 40 lines of code including some blank lines and comments. You might be able to come up with a way to write it more compactly than I did.

Additional Requirements:

- You must <u>not</u> modify any of the code that I have provided except for:
    - The body of `hasWon`
    - The body of `reveal`
    - The order in which the tests are applied (if you choose to tackle the assignment in a different order then I have presented it in this write-up – <u>this is not recommended</u>)
- All lines of code that you write must be inside functions (except for the function definitions themselves and any constant definitions).
- You must create and use the functions described previously in this document.
- Do <u>not</u> define one function inside of another function.
- You must make appropriate use of loops. In particular, your program must work for game boards of various sizes.

- Include appropriate comments on each of your functions. All of your functions should begin with a comment that briefly describes the purpose of the function, along with every parameter and every return value.
- Your program must <u>not</u> use global variables (except for constant values that are never changed, and in this assignment you may not even find that you want any global constants).
- Your program must use good programming style. This includes things like appropriate variable names, good comments, minimizing the use of magic numbers, etc. Your program should begin with a comment that includes your name, student number, and a brief description of the program.

Grading:

This assignment will be graded on a combination of functionality and style. A base grade will be determined from the general level of functionality of the program (Does it create the board correctly? Does it correctly identify the number of mines adjacent to a square? Does it reveal the board correctly when a square with 0 mines adjacent to it is clicked?). The base grade will be recorded as a mark out of 12.

Style will be graded on a subtractive scale from 0 to -3. For example, an assignment which receives a base grade of 12 (A), but has several stylistic problems resulting in a -2 adjustment will receive an overall grade of 10 (B+). Fractional grades will be rounded to the closest integer.

Total Score (Out of 12)	Letter Grade
12	A
11	A-
10	B+
9	B
8	B-
7	C+
6	C
5	C-
4	D+
3	D
0-2	F

Looking for an A+? Create a recursive solution to `reveal` that does <u>not</u> include a loop or any lists (other than the game board).