David Chunho Lee

CS325 Portfolio Project: Analysis Component

**Playable game URL:**

https://repl.it/@davidchunholee/DavidMinesweeper#main.py

**Description**:

Minesweeper is a game played on a m * n grid where the goal is to uncover the entire field without revealing a mine. Initially, the mines are randomly generated, hidden, and spread throughout the board. The user starts off by seeing an empty board and has to uncover a cell. If the cell contains a mine, the game is over and the player loses. If it does not contain a mine, the user uncovers a number in that cell. The number displays how many mines are in its vicinity (adjacent, neighboring cell by 1 square). See Figure 1 for further explanation.

**Figure 1**:

| | | |
|---|---|---|
| * | | |
| | 3 | |
| * | | * |

*Description: In the table above, we can visualize a 3x3 grid. The user uncovers the center number '3' represents how many mines are within its 3x3 square. Mine is represented as a '*' (seen in top-left cell, bottom-left cell, bottom-right cell). In an actual game environment, the mines, '*', will be hidden from the player and the empty cells will be populated with numbers as well (when the player uncovers it), representing its own number depending on how many mines there are.*

**Rules**:

A 9x9 board is populated with randomly generated mines and calculated numbers that represent each cell and its surrounding mines. The user can enter their input by entering a row number and a column number, separated by a comma.

The following inputs are valid:

Input: '5, 6'

Input: '      7,                  4             '

Input: '9,1'

The following inputs are invalid:

Input: '0, 5'                    (number needs to be between 1-9)

Input: ‘5 3’ (no comma)

If the user uncovers a cell that is not a mine and is not next to a mine, that cell and its neighboring cells are also uncovered for the user. Once all cells are revealed (without the ones containing mines), the user wins.

**Analysis of verification algorithm**:

The minesweeper program initializes a m*n board (where m=9 and n=9) and produces 10 randomly placed mines onto the board via the ‘mine_locations’ method and ‘random’ module. Then, the ‘cell_value’ method is called to populate the non-mine cells with integers.

The status of the game is checked using the ‘game_status’ method. This is the important function that checks to see if the game is over. In the description, we mentioned that there were two paths to complete the game. The first way is to uncover a mine which results in a loss. The second way is to uncover the entire board (without hitting a mine) which results in a win. The ‘game_status’ method utilizes a ‘cells_checked’ variable that stores the number of cells that have been uncovered. We are playing this game on a 9x9 board, which holds 81 cells. There are 10 randomly placed mines in this game so we subtract 10 from 81 to get the cells that are available to be uncovered without resulting in a loss.

After every move that the player makes, ‘game_status’ checks and increments ‘cells_checked’ by seeing how many uncovered cells there are. If it is equal to 71, then the player wins since he/she has uncovered every non-mine slot. The function uses two nested for-loops as seen below:

```
def game_status(self):
    cells_checked = 0
    for row in range(n):
        for column in range(m):
            if self.board[row][column] != ‘-‘:
                cells_checked += 1
```

This has a runtime of O(m*n) where ‘m’ is the number of rows and ‘n’ is the number of columns on the board. It iterates over the entire board, or more specifically, every column ‘m’ per row ‘n’ resulting in Theta(m*n). This algorithm is probably most similar to a linear search algorithm or brute-force algorithm where the algorithm checks every cell. It is unlike linear search where the key may be towards the beginning which would result in earlier termination of the algorithm. It is similar to linear search/brute-force where they both have O(n) as an upper-bound where n is the size of input. In our grid/playable board, the size of input is m*n resulting in O(m*n) which is still a polynomial time algorithm.

When the board is initialized, there are mines placed randomly and the location of the mines are added as keys to a hash table (Python dictionary). The second path to complete the game is if the player uncovers a mine (resulting in a loss). The actual verification of the player's move or 'certificate' is done via the 'explore_cell' method. It checks if the keys (row,column) are in the hash table. Although in average case it may be O(1) to search for keys, we are interested in the worst-case of O(n) if there is any collision which still runs in polynomial time. See the snippet of code in 'explore_cell' below:

```
if (inputrow, inputcolumn) in self.mine_map.keys():
        self.clear_screen()
        print(f"Your choice {inputrow + 1}, {inputcolumn + 1} was a mine! Game Over")
        self.game_over = True
```

In a more generalized Minesweeper game, the decision question used to prove NP-Completeness is the question, "Given a rectangular grid partially marked with numbers and/or mines, some squares being left blank, to determine if there is some pattern of mines in the blank squares that give rise to the numbers seen[2]." Another way to think about this generalized problem is, "Given a grid, for each cell 'n' and mines 'm', can a board be created with integers representing that represent 'n' while accurately accounting for surrounding 'm' cells? To answer this, the cell_value method calculates all the values of the m*n playable board.

```
def cell_value(self):
        for mine in self.mine_map:
                (mine_row, mine_column) = mine
                for i in range(mine_row – 1, mine_row + 2):
                        for j in range(mine_column – 1, mine_column + 2):
                                … O(1) statements
```

Remember that we are proving for a generalized Minesweeper board, so instead of 10 mines, there are 'n' mines. For each 'n' mine, we iterate upon a 3x3 grid and increment by 1 for each adjacent mine, which again is done in polynomial time. With the way this code is setup, it only verifies a ((9x9)-1) grid instead of iterating over the entire grid. No matter how many mines we have in an infinite x infinite grid, we check a constant ((9x9)-1) grid. This comes up as O(n * i * j) where i=3; j=3. This can be substituted as O(9n) which is equivalent to O(n) asymptotically.

In order to prove that Minesweeper is NP-Complete, we have to be able to verify a certificate in polynomial time. As shown above, the verification algorithm for the decision questions are done in polynomial time and is therefore is in NP. The next step would be to show that every other problem in NP is reducible to it. To do this, we take a widely known NP-Complete problem to reduce it to Minesweeper.

Robert Kaye is a PhD mathematician that has published an article proving that Minesweeper is a NP-Complete puzzle[2]. In order to do so, he showed that the SAT-problem is reducible to minesweeper utilizing Boolean circuits built with logic gates. In order to do so, Kaye built a Minesweeper field represented as a 'wire' (part of a Minesweeper board) with adjacent cells x and x' surrounded by integer values that represent how many mines are surrounding that cell. The 'wire' is built such that all cells with an 'x' have the mine or do not contain the mine (can be vice versa) but not mixed. So these cells x and x' can be represented with True or False logic gates dependent on operators (numbers surrounding the board). The 'wire' also represents logic gates which are central to the CIRCUIT-SAT problem of verifying the truth of a certificate. The CIRCUIT-SAT problem can be reduced to this representation in polynomial time. Again, we know that the CIRCUIT-SAT is a well-known NP-Complete problem and with these logic 'wires' building up the Minesweeper board - we can visualize that this puzzle also fits into the NP-Complete space.

```
#####################################################################################
###################################### Sources ######################################
#####################################################################################
# Journal article for choosing NP-Complete Game
# 1. Kendall, Graham, Parkes, Andrew, and Spoerer, Kristian.
#    'A SURVEY OF NP-COMPLETE PUZZLES'. 1 Jan. 2008 : 13 – 34.
# 2. Kaye, Richardd. "Minesweeper Is NP-Complete." Springer-Verlag,
#    vol. 22, no. 2, 2000, pp. 9–15.,
#    www.minesweeper.info/articles/MinesweeperIsNPComplete.pdf.

# General ideas on how to build the project adapted from:
#    https://www.askpython.com/python/examples/create-minesweeper-using-python
#    https://medium.com/swlh/this-is-how-to-create-a-simple-minesweeper-game-in-
python-af02077a8de

# Idea of focusing on 3x3 grid instead of iterating over entire m*n board adapted
from:
#    https://www.youtube.com/watch?v=ptMMa-SDSeE
#####################################################################################
```