CMPEN351 Final project

Name: Chin Shiang Jin

Email : sjc6393@psu.edu

1 – Writeup for end users:

Descriptive Project Title - A simple Minesweeper

Project Summary, Game Rules, and User Interfaces

This project will be using the Bitmap Display to set up a simple Minesweeper game.

At the beginning of the game, the player can select the three difficulty levels. The easy level will have a size of 6 x 6 tiles and contains 3 mines. The normal level will have a size of 8 x 8 tiles and contains 6 mines. The hard level will have a size of 10 x 10 tiles and contain 10 mines. These tiles will be presented on the bit map display on a 2D grid. The content of the tiles will be hidden at the start of the game. Tiles that do not contain mine are safe.

The goal of the game is to uncover all the tiles which do not contain a mine. If the player reveals a tile hiding a mine (with symbol *), the player loses. If he managed to reveal all safe tiles without revealing the mine, he wins. The safe tile will contain a hint number showing how many mines is(are) around in the adjacent tiles.

The player will be entering a tile of his/her choice through the I/O window using numbers representing the tiles. For an example of 8 x 8 tiles, 1 -8 are the first row, 9 -16 are the second row, etc. Tile numbers for tiles that are not yet revealed will be shown on the Bitmap Display for the player's reference. If the player manages to reveal tile which does not contain any mine in adjacent tiles (the tile will display hint number 0), the adjacent tiles of the tile with hint number 0 will be automatically revealed as well.

After winning or loosing a particular game, the player can choose to replay the game by entering integer 1 when prompted. This will clear the Display, reset all the arrays used and bring the player back to the beginning of the game.

Mars configuration details

Set up the Bit Map Display with the following settings:

2 – Writeup for project details (technical):

Architectural description

This project will use a 100 words array to represent the tile's content. The actual array capacity used will be dependent on the game difficulty level selected by the users (36 for the easy game, 64 for the normal game, 100 for the hard game).

During the initial set-up, the program will prompt the player to select the difficulty levels (0 – easy, 1 – normal, 2 – hard). After that the program will read the values of the difficulty table, filling "variables" such as Numtiles (Number of tiles in one row/column), LayoutSize (Total number of tiles in the layout), SizeTiles (pixel size for a tile on the bit-map display), NumMines (number of mines) and GameCount (number of tiles revealed required to win the game) for the game.

After that, the program will generate n non-repeating random numbers (from 0 to LayoutSize) and store a special value on the corresponding array position. While storing, the procedure should increment the content of adjacent tiles by 1 indicating there is one mine nearby. Once the array is ready, a loop is used to read the tile numbers from a queue to reveal its content.

If there is no number in the queue, a prompt will be displayed asking the player to start entering the tile number they want to reveal. The program will read the integer number entered by the player, check if the number has been entered before by checking the corresponding index of the TileVisited array record. If the number has not been entered before, it will be placed into the queue. The corresponding TileVisited array index position will be updated to record the entry.

If there is an item in the queue, the program will retrieve the number from the queue and reveal the tile content on the Bitmap Display. If the tile content is of hint number 0, the program will get the adjacent tile numbers of the tile, check if the numbers have been entered before, and enter the number into the queue if haven't. For every tile revealed, the GameCount variable will be decreased by 1.

If the tile content is of hint number 9, this means the tile contains a mine, the program will display the game losing the message. If all the safe tiles are revealed without revealing the mine(s), the program will then display the game-winning message. Otherwise, if the player accidentally reveals a mine.

The flow charts for the programs are as follow:

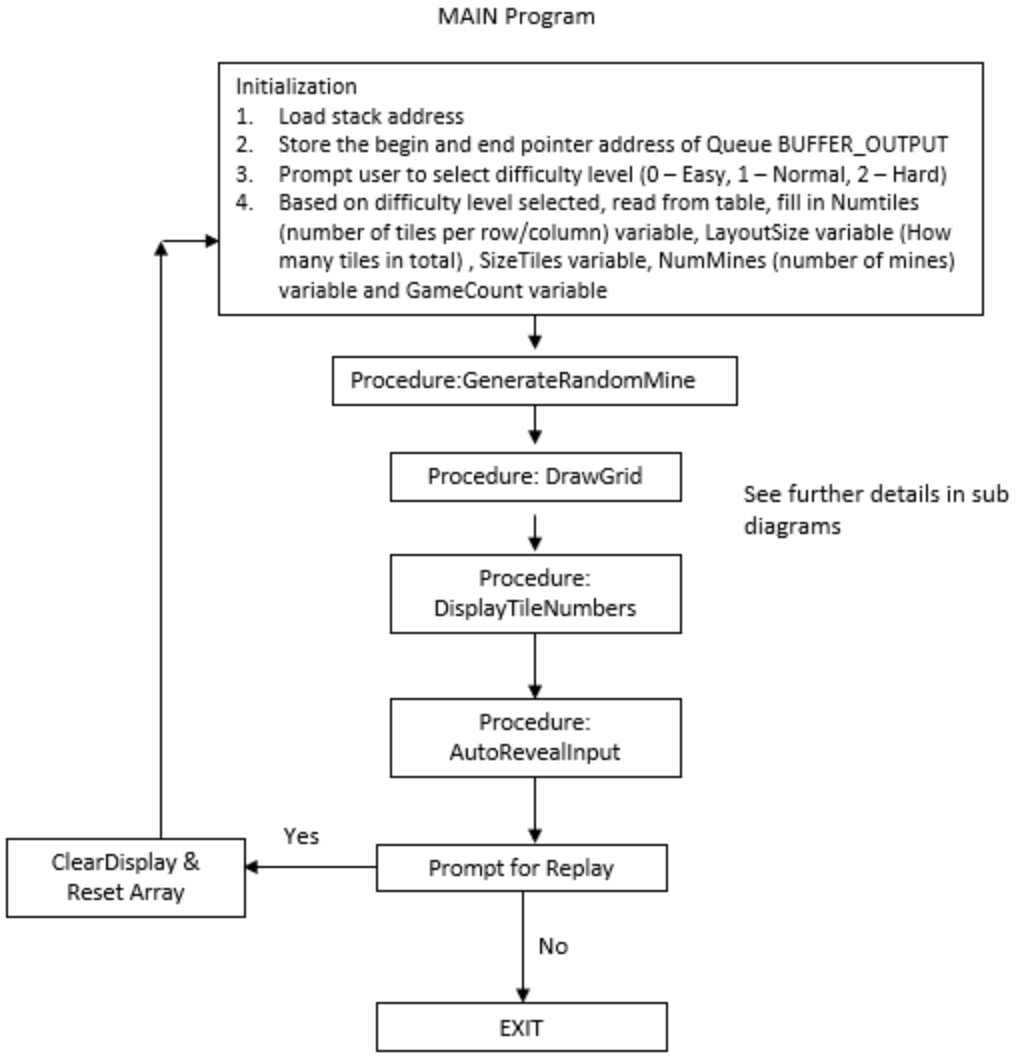


Figure 1: Flow chart of the main procedure

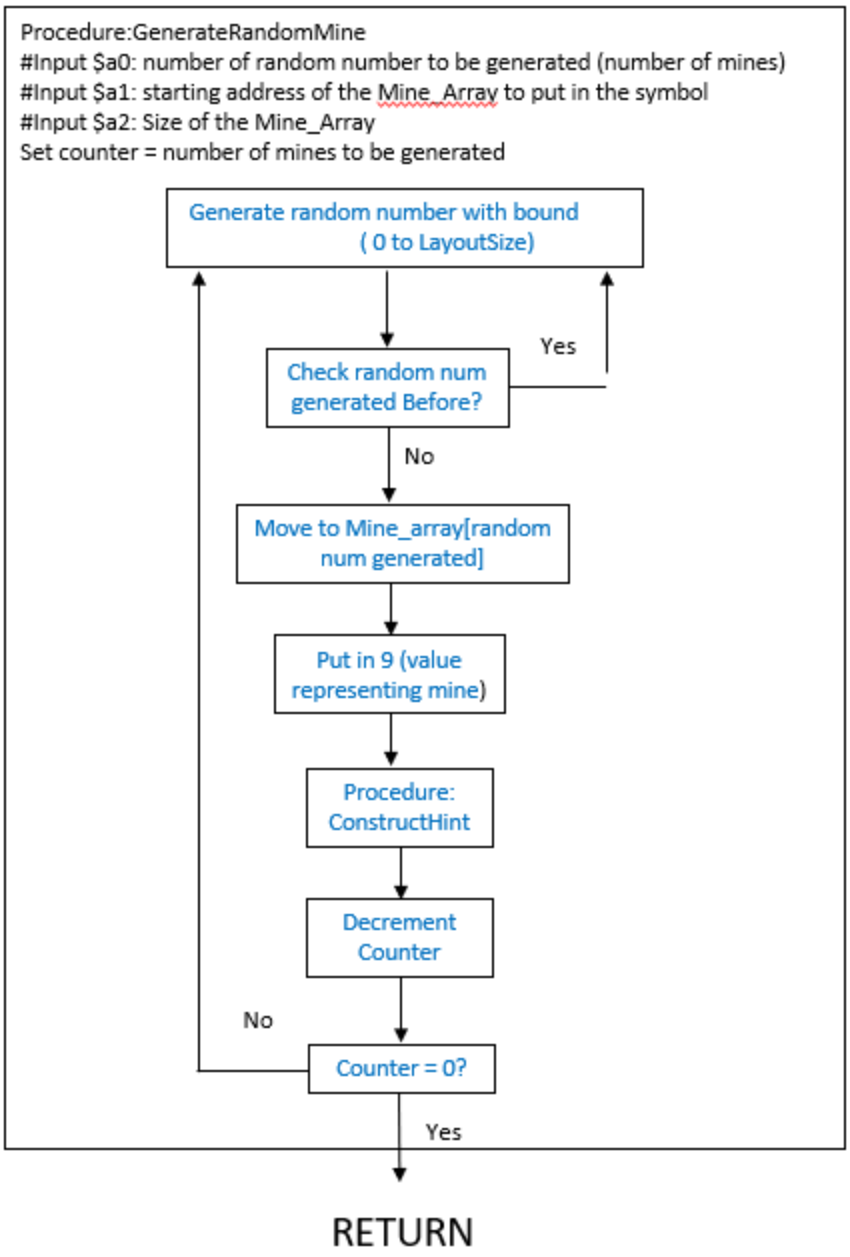


Figure 2: Flowchart for Procedure GenerateRandomMine

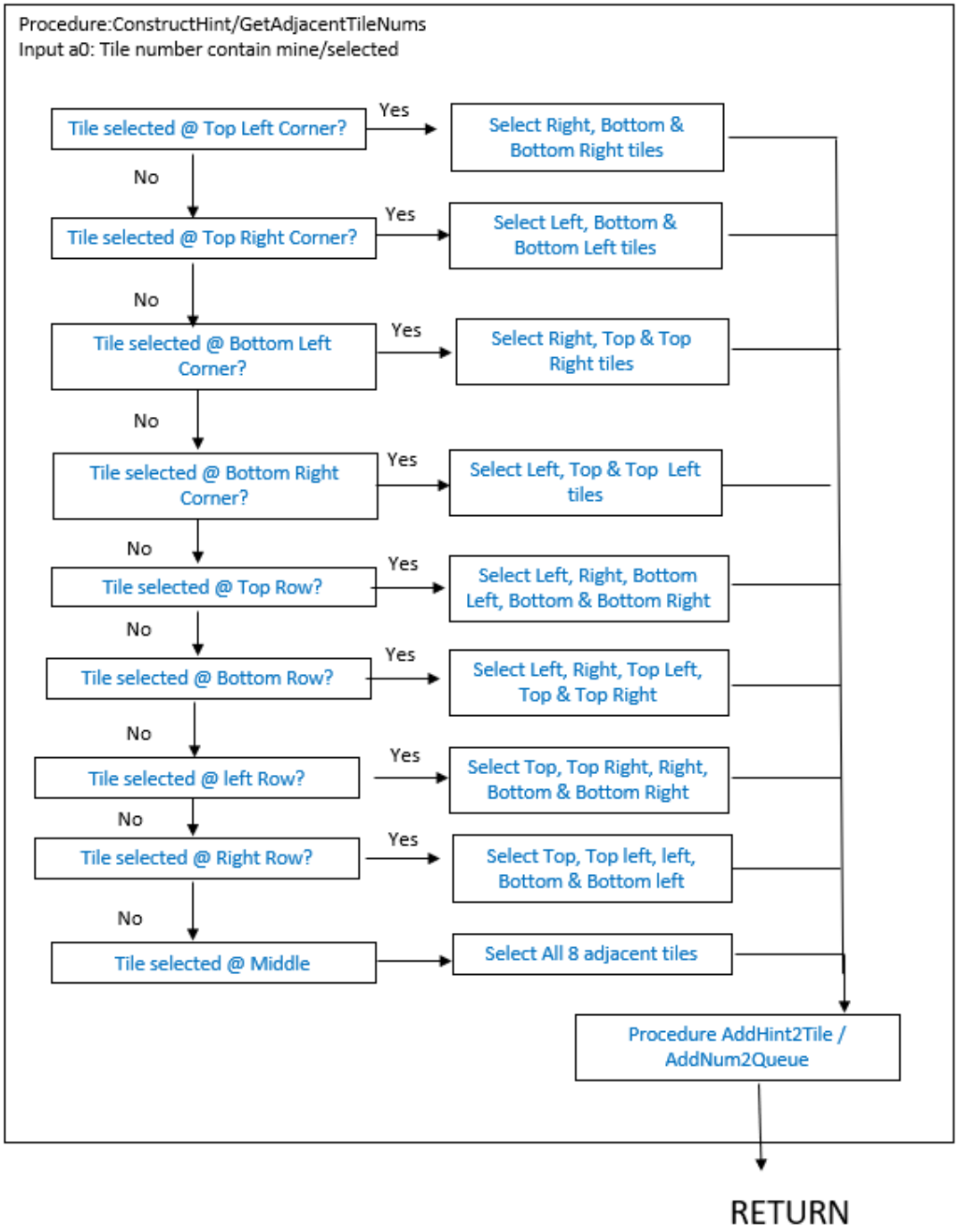


Figure 3: Flow Chart for ConstructHint/ Get AdjacentTileNums, both have identical structures

Note for the ConstructHint procedure, it will need to cater for corner cases and edge cases. For example, for a tile in the middle that contains a mine, all 8 adjacent tile content will add 1 to indicate there is one mine nearby. But for the top left, top right, bottom left and bottom right corners, only 3 adjacent tiles' content need to add 1. Refer to the Pseudocode attached at the end of this report. This is similar to the AdjacentTileNums procedure.

Top Left	Top	Top Right
Left	Selected Tile	Right
Bottom Left	Bottom	Bottom Right

Table 1: Reference for adjacent tiles

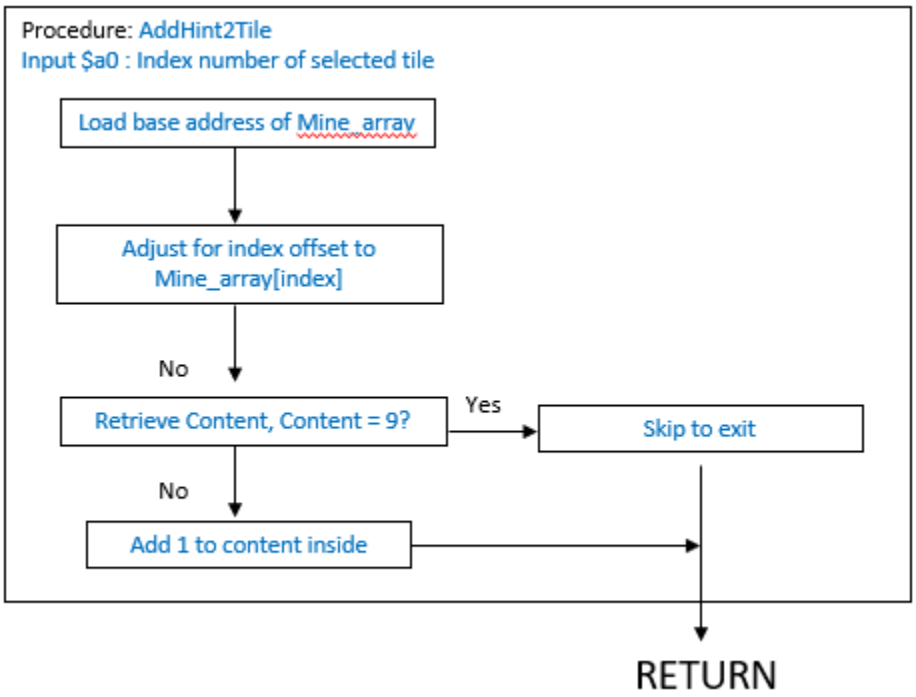


Figure 4: Procedure AddHint2Tile

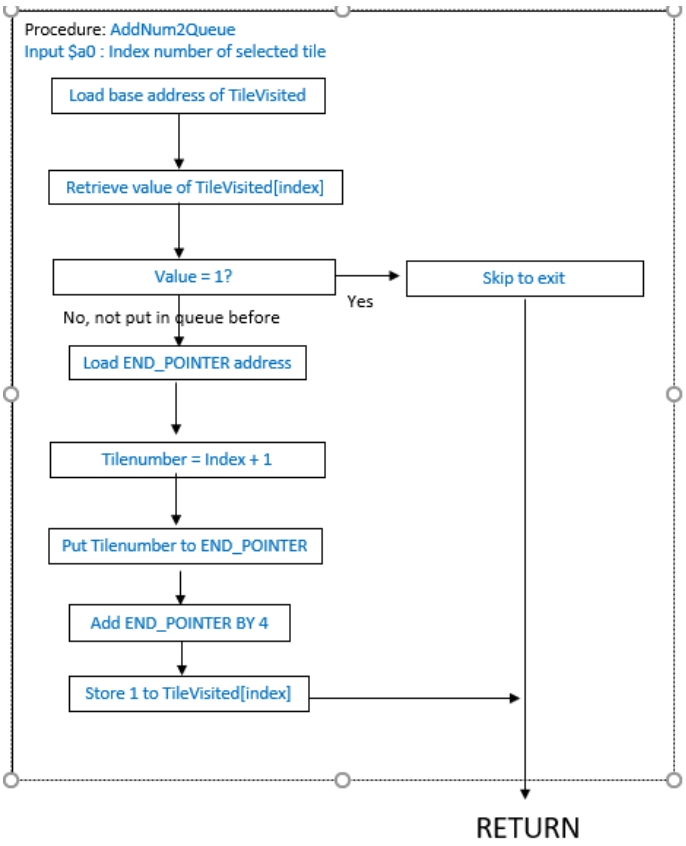


Figure 5: Procedure AddNum2Queue

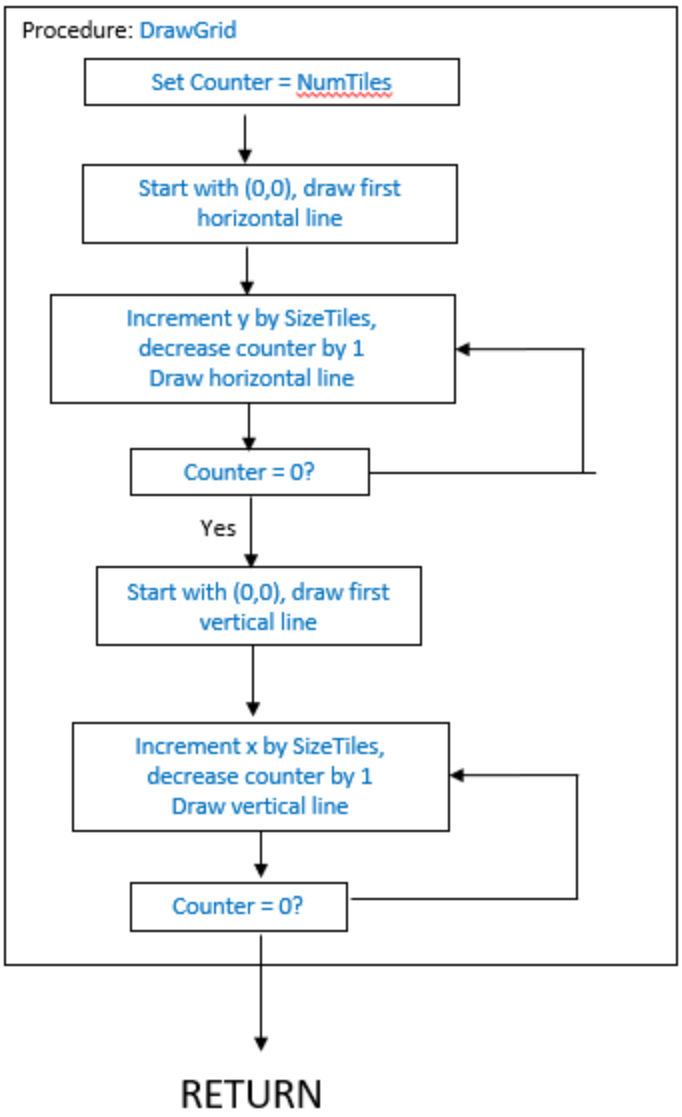


Figure 6: Procedure DrawGrid

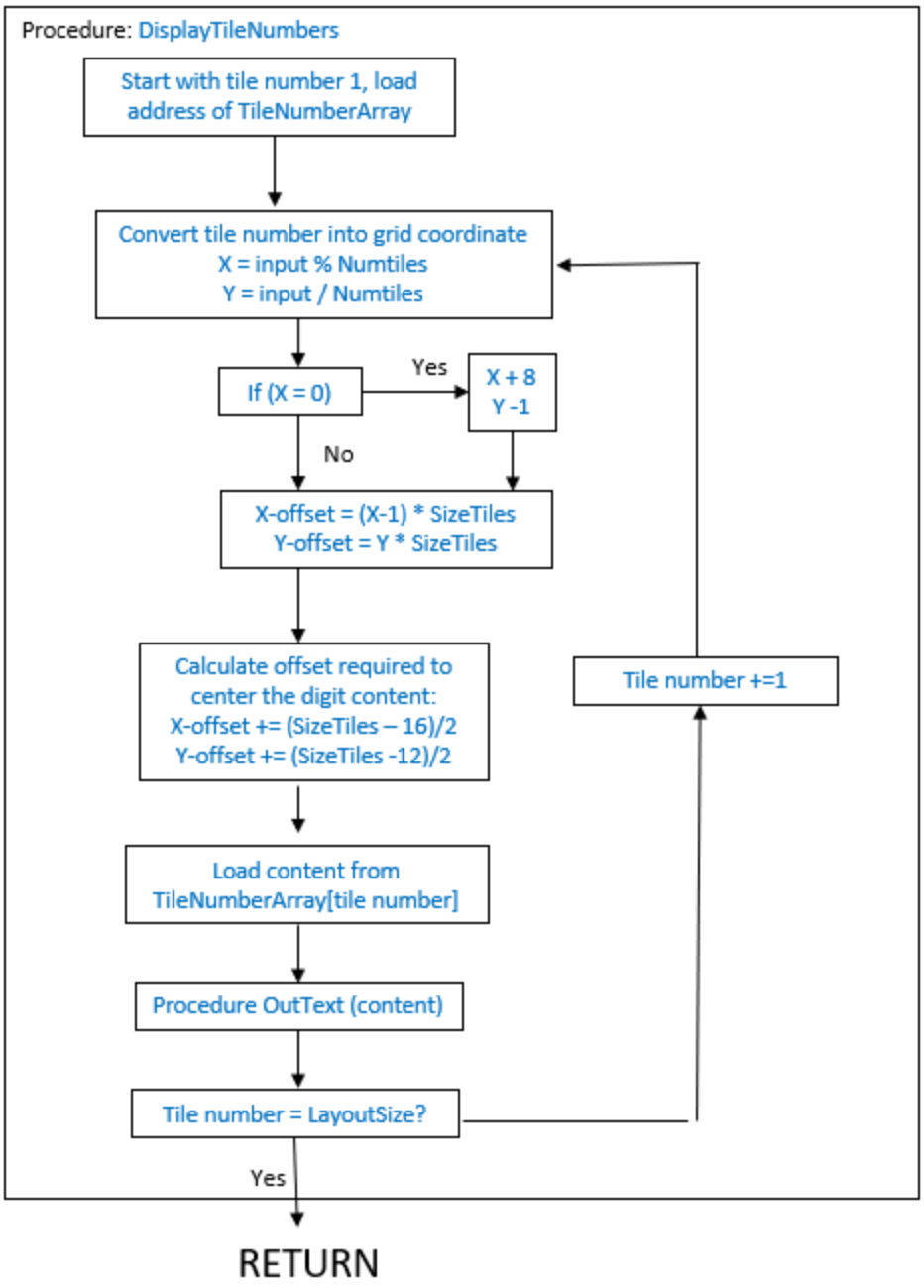


Figure 7: Procedure DisplayTileNumbers

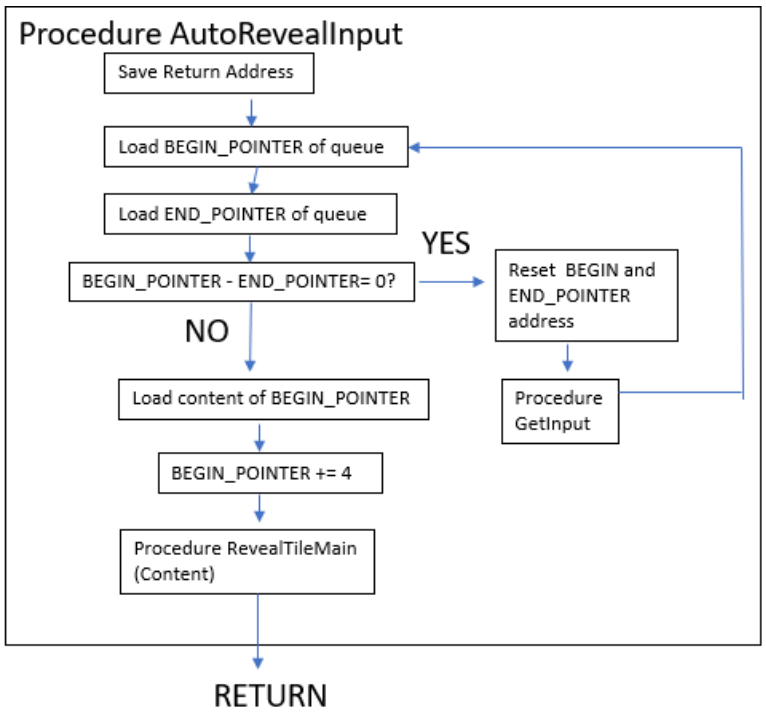


Figure 8: Procedure AutoRevealInput

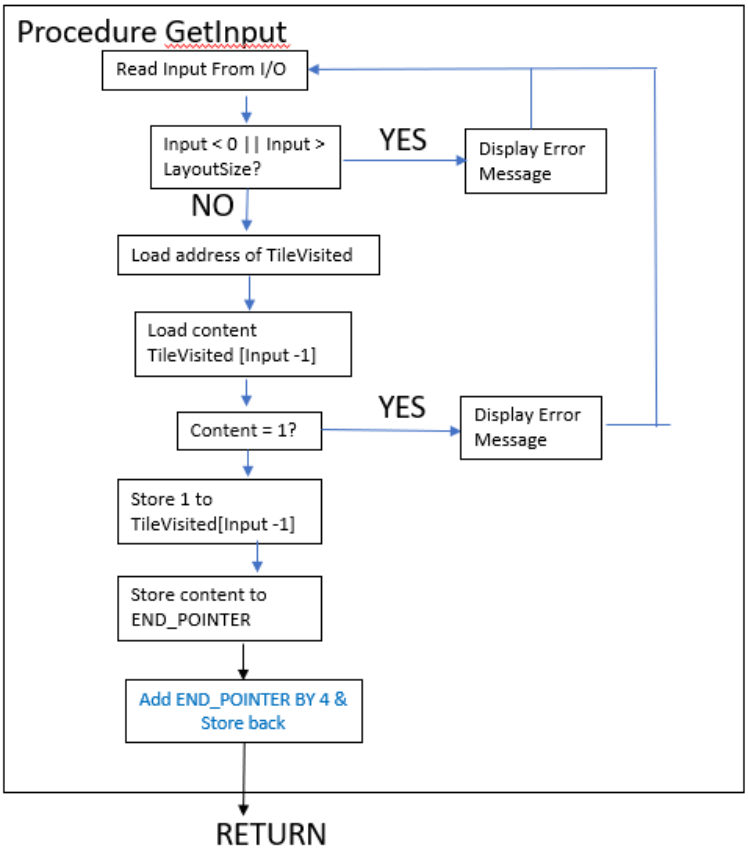


Figure 9: Procedure GetInput

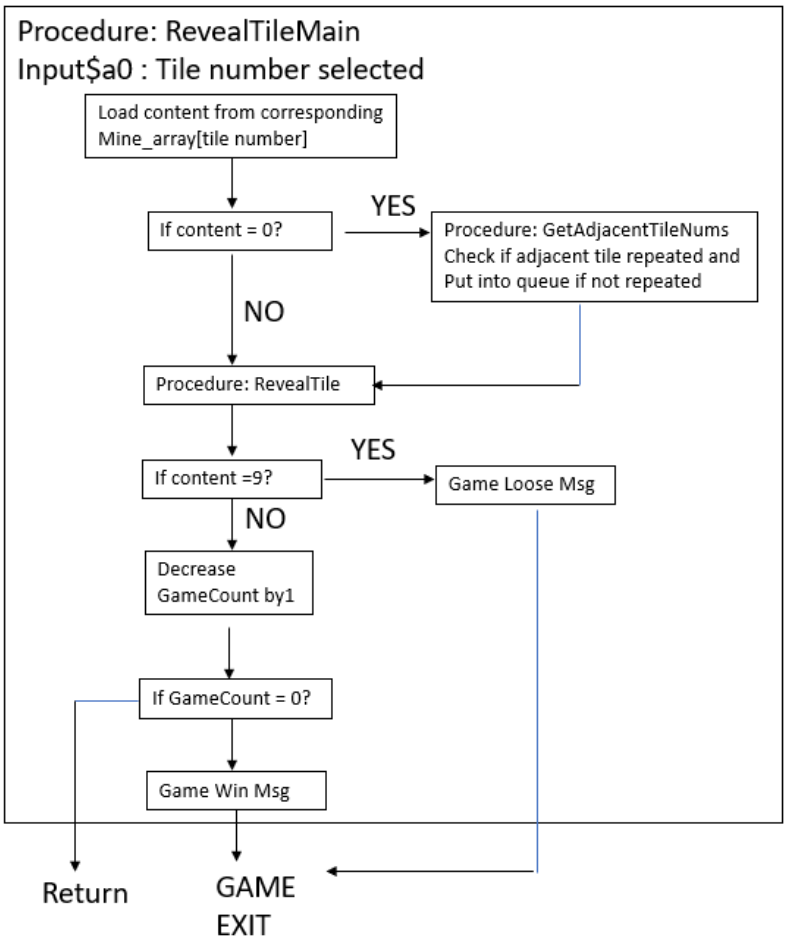


Figure 10: Procedure RevealTileMain

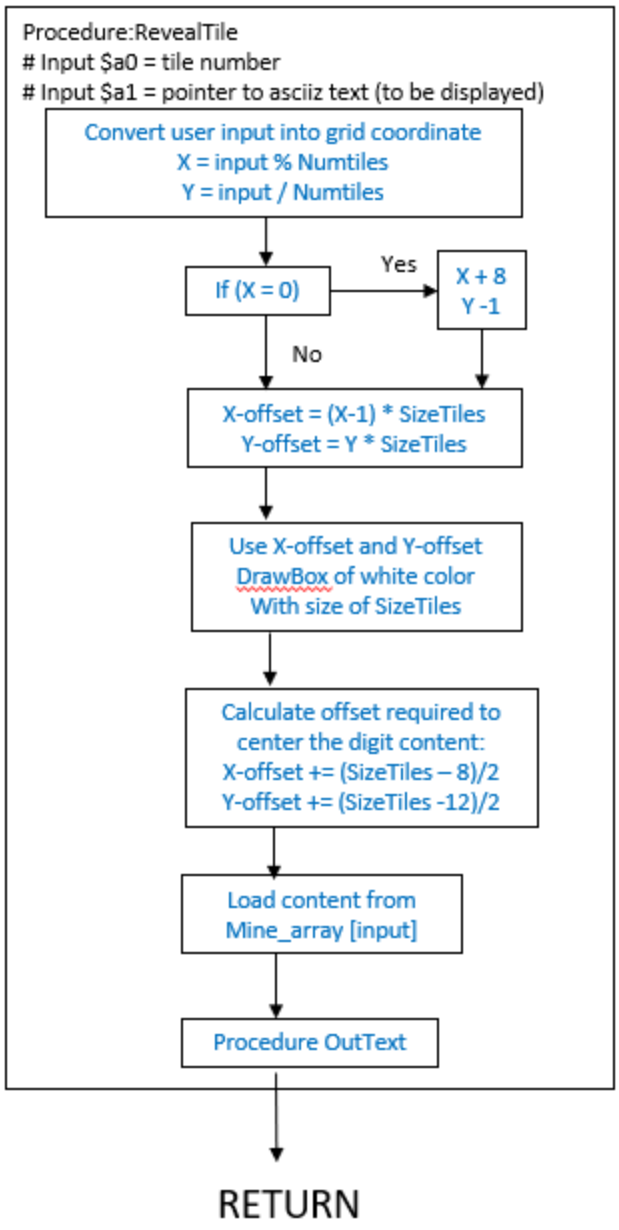


Figure 11: Procedure RevealTile

Test/verification/build process

The building and testing process of the project will be divided into the following stage:

- First build – To build the basic text version of the game using I/O input. Key procedures required are as follow:
    - Procedure to randomly generate a non-repeating tile number that contains a mine
        - **Test**: Try to limit the bound of random number generated (0 to n) to be the same as the number of random numbers generated (n numbers), and confirm the first n index positions of the Mine_array are filled

    - Procedure to fill in the array (will be named as Mine_array) which stores the content of all mine tiles. This includes the procedure to fill in the hint given the tile number contains the mine.

        - **Test**: After this, at the breakpoint, check the content of Mine_array, the mine tile will have a value of 9. All adjacent tiles will have a value + 1 for each mine nearby.
        - **Test**: Look at the corner cases and edge cases to ensure the hint is added correctly.

    - Procedure to get the user input from the I/O window entry, and check the content of selected tile in the Mine_array. If the selected tile index by the user contains a mine, display the loose message. If all the non-mine index positions are checked, display the winning message.
        - **Test**: Use a smaller size array (say 10) with 5 mines. First, try to enter the number that contains mine to see if the loose message got displayed. Then enter all 5 numbers that don't have mine to see if they win message got displayed.

- The second build – build the graphic components required for the game and test it individually. Components required are as follow:
    - Procedure to draw the gridlines
        - Requires procedure drawing a horizontal line and procedure drawing a vertical line)
            - Which further requires procedure drawing the dot,
        - **Verify** by running this procedure alone, should see the gridlines drawn according to the grid arrangement of the game
    - Procedure to draw a white box (revealing the tile content as the default background is black)
        - Requires procedure drawing box
        - Which further requires procedure drawing the horizontal line
        - And procedure drawing the dot
        - **Verify** by running this procedure alone and confirm corresponding white box at the selected tile number is drawn
    - Procedure to draw the digit output (representing tile numbers and the hint for the tile)

        - Modify the OutText procedure in previous lab 7 by allowing the procedure to take input for the color selected.
        - **Verify** by running this procedure to check if the digit got displayed properly

- The third build – integrate the graphic components to the text version of the game
    - Create RevealTile procedure to reveal the tile content selected by the user
        - **Conduct Full Test.** Run the procedure, input user-selected tile number, verify on the Bitmap on visual expected

- Fourth Build – Incorporated advanced features for the game
    - Allow users to select the difficulty level
        - Rerun Test for third build for each difficulty level. The game should perform as expected.
    - Display tile number on the layout for the player to select the tile
        - Verify through the Bitmap display during initialization
    - Deal with the exceptions
        - If number input out of bound/outside range, skip the count and display the warning message
        - If a number entered before, skip the count and display the warning message
        -
- Fifth Build – Automatically reveal adjacent tile if the selected tile has zero value.
    - Start by building a text version of the automatic reveal procedure
        - Utilizing the flow charts from Figures 3, 5, 8, 9, 10. Replace the call to RevealTile procedure in figure 10 with a simple print integer syscall to print the tile number selected.
    - For the test, load Mine_Array content with all 0. Load the initial setting for Numtiles, SizeTile, LayoutSize. Set NumMines = 0 and GameCount = 58.
        - Run the AutoRevealInput loop, verify that after the player enters a valid tile number, all the tile numbers are automatically printed by the procedure.
        - And there is no repeating tile number(s) printed.
    - Incorporate the automatic reveal procedure into the fourth build.
        - Run the full test to ensure the game performed as expected.
            - The mines are generated randomly
            - The adjacent tiles of hint number 0 were revealed automatically
            - Revealing tile contain mines will result in the game loose message
            - Revealing all tiles not containing mines will result in a game win message.

Debug/issues

The main change from the initial design is the removal of Keyboard Polling. It was complicated than I initially thought because the program uses two digits to represent the tile numbers. If we were to poll the player input using Keyboard, then we need to design a procedure to combine two numbers entered by the player together to form the tile number digits. The complexity of the feature outweighs the potential benefit of it, as with the automatic reveal loop available, the player now doesn't need to input all the tile numbers one by one to reveal them. Instead, the player should wait for the automatic reveal loop to reveal tiles adjacent to hint number 0 before entering the next choice.

Another challenge that was complicated is to write the procedure to add hints to tiles adjacent to the mine and add tile numbers adjacent to the queue. Although both have identical code structures, they differ in the final procedure (AddHint2Tile / AddNum2Queue). Hence I was unable to figure out a way to select a different final procedure from the main procedure. I also avoided using a fixed table to deal with special cases (like corner tiles, top/bottom rows, left/right columns) as I will need the procedure to identify the special cases given different grid layout (6x6 or 8x8 or 10x10).

One particular feature that I am particularly proud of is the use of a queue to automatically reveal the tile adjacent to hint number 0. The queue was implemented by declaring a BUFFER_OUTPUT array and two pointers (BEGIN_POINTER_OUTPUT) and (END_POINTER_OUTPUT).

- The new number will be added to the address stored in END_POINTER_OUTPUT, where the address will be increment by a word distance (4 bytes) and stored back to the END_POINTER_OUTPUT.
- The numbers in the array will be extracted starting from the address stored in BEGIN_POINTER_OUTPUT, where the address will be increment by a word distance (4 bytes) and stored back to move to the next number.
- Hence a First In First Out (FIFO) queue was able to implement.
- This allows the program to utilize the GetAdjacentTileNums procedure to get the adjacent tile numbers of tile with hint number 0, and store it in the queue to be processed (revealed) one by one.
- With the use of a TileVisited array to store the entry history of tile numbers, the program was able to automatically reveal adjacent tiles while preventing redundant tile numbers from being stored in the queue.

3 – Source Code

Refer attached file.

Appendix 1: Pseudocode for Procedure ConstructHint

Input: Tile number generated by random number generator which will contain a mine

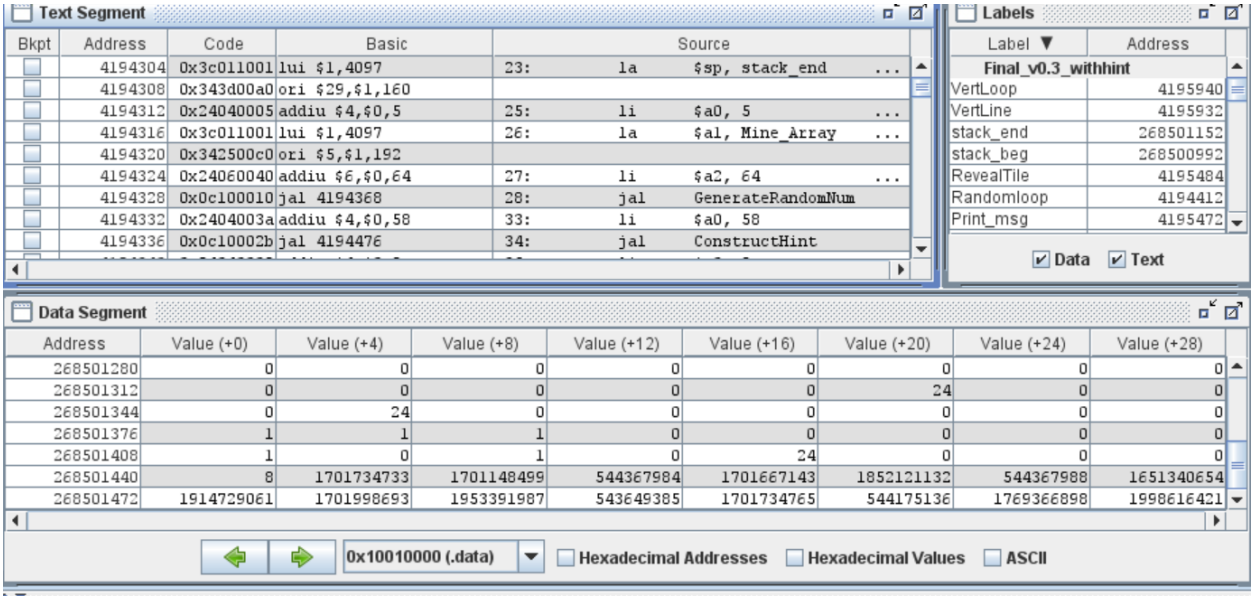
Input2: Total number of tiles in one row (consider putting this as variable for easier input) – Numtiles

Output: content of adjacent tiles + 1

```
If(number = 1 )                                  # Border case top left corner
        Add 1 to array[number]                   #add 1 to the adjacent tile on the right
        Add 1 to array[number + Numtiles]        #add 1 to the adjacent tile on bottom
                                                 right corner
        Add 1 to array[number + Numtiles -1]     #add 1 to the adjacent tile on bottom

Else if (number = Numtiles)                      #Border case top right corner
        Add 1 to array[number - 2]               #add 1 to the adjacent tile on the left
        Add 1 to array[number + Numtiles -2]     #add 1 to the adjacent tile on bottom
                                                 Left corner
        Add 1 to array[number + Numtiles -1]     #add 1 to the adjacent tile on bottom

Else if (number = (Numtiles * (Numtiles -1) +1)  #Border case bottom left corner
        Add 1 to array[number]                   #add 1 to the adjacent tile on the right
        Add 1 to array[number – Numtiles]        #add 1 to the adjacent tile on top
                                                 right corner
        Add 1 to array[number – Numtiles -1]     #add 1 to the adjacent tile on the top

Else if (number = (Numtiles*Numtiles)            #Border case bottom right corner
        Add 1 to array[number -2]                #add 1 to the adjacent tile on the left
        Add 1 to array[number – Numtiles -2]     #add 1 to the adjacent tile on top left
        Add 1 to array[number – Numtiles -1]     #add 1 to the adjacent tile on top

Else if (number < Numtiles)                      #Border case: Top row (Note top
                                                 #corners will be captured by previous
                                                 #if/else statement
        Add 1 to array[number]                   #add 1 to adjacent tile on the right
        Add 1 to array[number -2]                #add 1 to adjacent tile on the left
        Add 1 to array[number + Numtiles – 2]    #add 1 to adjacent tile on the bottom left
        Add 1 to array[number + Numtiles – 1]    #add 1 to adjacent tile on the bottom
```

```
        Add 1 to array[number + Numtiles]          #add 1 to adjacent tile on the bottom
                                                   right

Else if (number > (Numtiles * (Numtiles -1))       #Border case: Bottom row (Note bottom
                                                   #corners will be captured by previous
                                                   #if/else statement
        Add 1 to array[number]                     #add 1 to adjacent tile on the right
        Add 1 to array[number -2]                  #add 1 to adjacent tile on the left
        Add 1 to array[number - Numtiles – 2]      #add 1 to adjacent tile on the top left
        Add 1 to array[number - Numtiles – 1]      #add 1 to adjacent tile on the top
        Add 1 to array[number - Numtiles]          #add 1 to adjacent tile on the top right

Else if(number % Numtiles = 0)                     #Border case: right column (Note top
                                                   #and bottom corner will be captured by
                                                   #previous if/else statement
        Add 1 to array[number – 2]                 #add 1 to adjacent tile on the left
        Add 1 to array[number – Numtiles -2]       #add 1 to adjacent tile on top left
        Add 1  to array[number – Numtiles -1]      #add 1 to adjacent tile on top
        Add 1 to array[number + Numtiles -1]       #add 1 to adjacent tile on bottom
        Add 1 to array[number + Numtiles -2]       #add 1 to adjacent tile on bottom left

Else if (number % Numtiles = 1)                    #Border case: left column (Note top
                                                   #and bottom corner will be captured by
                                                   #previous if/else statement

        Add 1 to array[number]                     #add 1 to adjacent tile on the right
        Add 1 to array[number – Numtiles]          #add 1 to adjacent tile on top right
        Add 1  to array[number – Numtiles -1]      #add 1 to adjacent tile on top
        Add 1 to array[number + Numtiles -1]       #add 1 to adjacent tile on bottom
        Add 1 to array[number + Numtiles]          #add 1 to adjacent tile on bottom right
Else                                               #Rest of the case: for tile in the middle
        Add 1 to array[number]                     #add 1 to adjacent tile on the right
        Add 1 to array[number – Numtiles]          #add 1 to adjacent tile on top right
        Add 1  to array[number – Numtiles -1]      #add 1 to adjacent tile on top
        Add 1 to array[number – Numtiles -2]       #add 1 to adjacent tile on top left
        Add 1 to array[number – 2]                 #add 1 to adjacent tile on the left
        Add 1 to array[number + Numtiles -2]       #add 1 to adjacent tile on bottom left
        Add 1 to array[number + Numtiles -1]       #add 1 to adjacent tile on bottom
        Add 1 to array[number + Numtiles]          #add 1 to adjacent tile on bottom right

Procedure to add 1 to the tile
Input1: index number of tile array

        Load base address of Mine_array
        Multiply index number by 4                 #Each array_position is 4 bytes apart
        Base address + index_offset
```

Retrieve content in the address
If (content = 24)
Skip
Else
Content += 1
Put the content back to the address
Jump back



Example Output verification