THE UNIVERSITY OF SYDNEY

COMP2017 / COMP9017 Assignment 1

March 27th, 2020, 11:59pm AEDT (Week 5 Friday)

This assignment is worth 6% of your final assessment

Task Description

In this assignment, you will be implementing a modified version of the game "Minesweeper" called "ND-Minesweeper". You will write a set of functions that enables a user to define and then play the game.

Working on your assignment

Staff may make announcements on Ed (https://edstem.org) regarding any updates or clarifications to the assignment. The Ed resources section will contain a PDF outlining any notes/changes/-corrections to the assignment. You can ask questions on Ed using the assignments category. Please read this assignment description carefully before asking questions. Please ensure that your work is your own and you do not share any code or solutions with other students.

You can work on this assignment using your own computers or lab machines. You will need to submit to Ed via Git, which is covered elsewhere in this course. When you make a submission, your submission will be automatically compiled and run and you will receive feedback as to whether you passed the public test cases as well as a link that will enable you to inspect the output of your submission.

It is important that you continually back up your assignment files. You are encouraged to submit your assignment while you are in the process of completing it to receive feedback and to check for correctness of your solution.

Introduction

ND-Minesweeper is played on a n-dimensional grid of n-dimensional hypercubes, where n is a positive integer and the grid is of arbitrarily defined size. We will refer to each hypercube in the game grid as a "cell". Each cell can therefore be defined by a set of n integer coordinates $(k_1, k_2 \dots k_n)$ where $0 \leq k_i \leq x_i$ for $1 \leq i \leq n$ where x_i is the maximum coordinate (one less than the size) of the given i-th dimension of the grid. The dimensions of the grid do not necessarily have the same size as each other. The cell at coordinates $(0, 0 \dots 0)$ is always a "corner" of the grid. Each cell may contain a mine, which is hidden from the player. The game starts with all cells unselected. Every turn, the player selects a cell.

Example

Below is an example of a starting 2-dimensional ($n = 2$) grid, with maximum coordinates 5 and 3 (that is, $x_1 = 5$ and $x_2 = 3$, with size for 6 and 4 cells respectively) and some cell coordinates indicated. The locations of the hidden mines are indicated by asterisks (*) (their coordinates are $(1, 3)$ and $(2, 2)$). At this point, no cells have been selected by the player.

(0,3)	*				(5,3)
		*			
(0,1)					
(0,0)	(1,0)				(5,0)

The player proceeds to select a cell in the first turn.

If the cell contains a mine, the player loses and the game ends.

If the cell does not contain a mine, the game informs the player of the number of cells adjacent to the selected cell which contain mines. A pair of distinct cells are adjacent if the maximum difference between any pair of matching coordinates is 1. That is, if the coordinates of the (distinct) cells respectively are $(a_1, a_2 \dots a_n)$ and $(b_1, b_2 \dots b_n)$, then they are adjacent if and only if

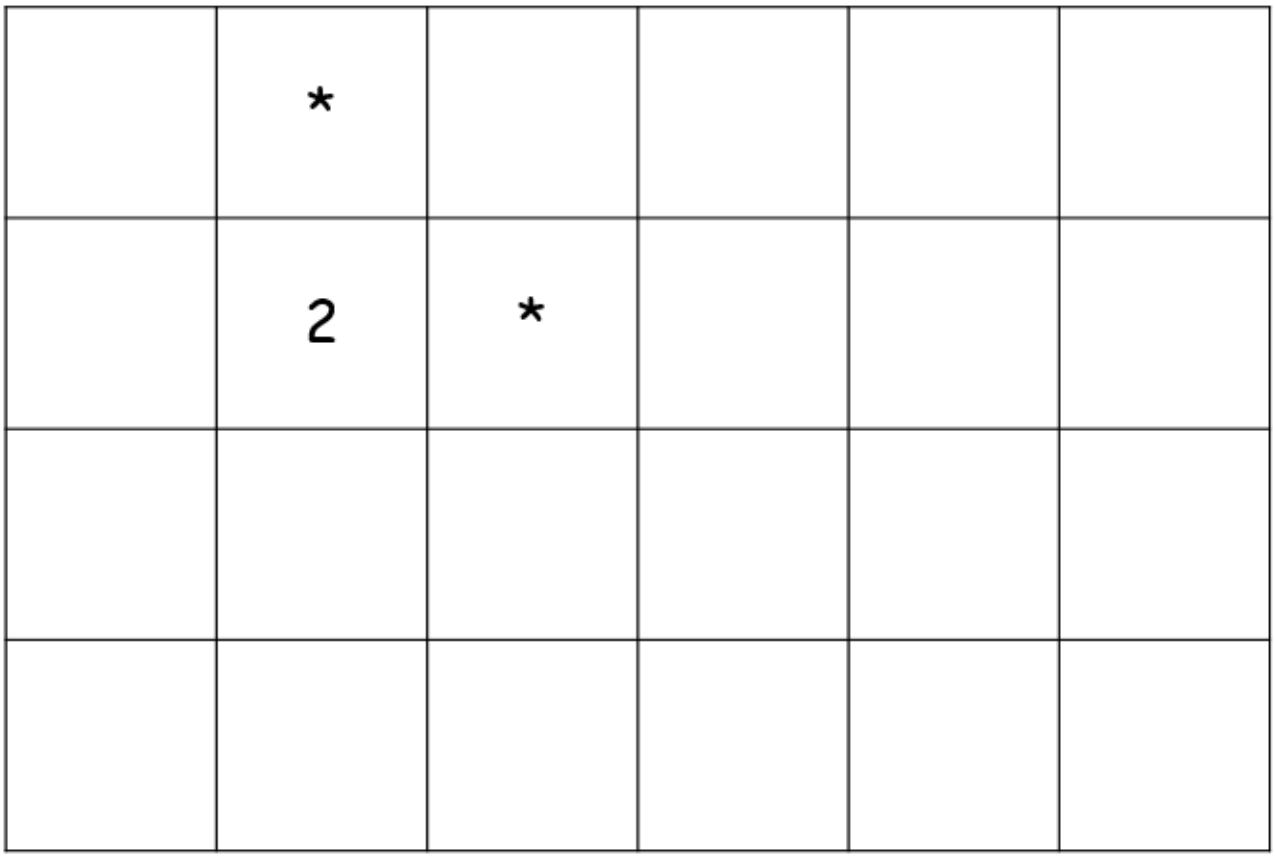
$$a_i - b_i = \begin{cases} -1 & \text{or} \\ 0 & \text{or} \\ 1 & \end{cases}$$

for all $1 \leq i \leq n$. Note that this means cells which are "diagonal" to the selected cell are adjacent. If no adjacent cells contain mines, the game automatically selects all adjacent cells recursively until all cells selected this way have a non-zero number of cells adjacent which contain mines.

Cells at the boundary of the grid (i.e. at least one coordinate k_i is 0 or x_i) are only adjacent to those cells within the boundaries of the grid, there is no wrapping to the other side.

When all cells that do not contain mines have been selected, the player wins and the game ends.

Using the example grid above, a player has made a move of selecting $(1, 2)$. The adjacent cells are $(0, 3)$, $(1, 3)$, $(2, 3)$, $(0, 2)$, $(2, 2)$, $(0, 1)$, $(1, 1)$, $(2, 1)$. There are two adjacent mines, so the player is informed of this number, and no cells are expanded recursively. The grid after this action is shown below.



The player then selects the cell $(4, 1)$. As this cell has no adjacent mines, cells are automatically selected recursively outwards until they contain at least one adjacent mine. Please note that cells

$(0, 3)$ and $(2, 3)$ have remained unselected because they were not reached by the recursive selection. Note that the recursive selection also stops when a boundary is reached. The grid after this action is shown below.

	*		1	0	0
1	2	*	1	0	0
0	1	1	1	0	0
0	0	0	0	0	0

At this point, if the player selects the cells $(0, 3)$ and $(2, 3)$, they win the game. If the player selects either $(1, 3)$ or $(2, 2)$, they lose the game.

Task

Each cell in the ND-Minesweeper grid will be represented by the following struct. The entire grid is represented by an array of such structs.

```
struct cell {
    int mined;
    int selected;
    int num_adjacent;
    struct cell * adjacent[MAX_ADJACENT];
    int coords[MAX_DIM];
    int hint;
}
```

MAX_ADJACENT is a constant integer that will be available as a preprocessor **#define** when your code is tested, representing the maximum space to store cells adjacent to the given cell. This may be less than the actual number of adjacent cells depending on the dimensionality of the game; however you are always guaranteed enough space to store pointers to adjacent cells.

MAX_DIM is a constant integer that will be available as a preprocessor **#define** when your code is tested, representing the maximum space to store coordinates representing a cell. The actual number of coordinates required is equal to the number of dimensions of the game; however you are always guaranteed enough space to store coordinates.

mined is an integer that is either 0 (no mine present at this cell) or 1 (mine present).

selected is an integer that is either 0 (not selected) or 1 (selected). It represents whether the cell has been selected, whether by the player or automatically.

num_adjacent is an integer equal to the number of cells adjacent to this one.

adjacent is an array of pointers to the structs representing all adjacent cells to this one, in arbitrary order.

coords is an array of integers representing the coordinates of the current cell.

hint is an integer that represents the number of adjacent mined cells to the current cell.

Implement the following functions for your ND-Minesweeper game. Do not write any **main()** function; your code will be tested by directly calling the functions you implement.

Initialisation

```
void init_game(struct cell * game, int dim, int * dim_sizes, int num_mines,
int ** mined_cells);
```

This function will be called first, exactly once at the start of the game.

dim provides you with the number of dimensions of this game, and **dim_sizes** is an array of **dim** integers representing the size of the grid (i.e. number of cells) in each respective dimension.

mined_cells is a **num_mines** sized array of **dim** sized integer arrays, which represents a **num_mines** sized array of coordinates of cells that contain mines.

Using the example grid above, the parameters would be: **dim = 2**, **dim_sizes = {6,4}**, **num_mines = 2**, **mined_cells = {{1,3}, {2,2}}**.

Your function must write a **struct cell** for each cell in the grid to the array **game**, which is guaranteed to have enough memory for the number of cells in the grid. You do not have to add structs into the array in any defined order, but there must not be any empty array entries between structs. For a suggestion on how to order the **game** array, see the Notes and Hints section at the end of this document. However you must ensure the following values for struct fields:

All cells must be initialised to **selected = 0**.

Mined cells must have **mined = 1**, all others **mined = 0**.

adjacent must contain the correct pointers to all the **struct cell**s in the **game** array representing adjacent cells of a given cell. This depends on where you store the corresponding **struct cell**s in the array **game**, which is up to you. You must also ensure the **num_adjacent** field contains

the number of adjacent cells. The **adjacent** array may contain up to **MAX_ADJACENT** elements as described above, but you may not need to use all the space. All your pointers should be placed at the start of the array, without empty entries.

coords must contain the coordinates of the cell that this **struct cell** represents. The coordinates array must represent the dimensions in the same order as defined in **dim_sizes**. As above, this array has space for up to **MAX_DIM** elements but may not all need to be used.

hint does not need to, but can, be defined by this function. It is only required to take on a correct value later (see below).

Gameplay

```
int select_cell(struct cell * game, int dim, int * dim_sizes, int * coords);
```

This function will be called only after **init_game** has been called exactly once. It will be called exactly once for each move of the player.

coords is a **dim** sized array of integers representing the coordinates of the cell that the player has selected. **dim_sizes** has the same meaning as for **init_game**.

game is the same array that you created in **init_game** or modified in previous calls of **select_cell**. You must now modify your array based on the player's selection.

If the player has selected a cell with a mine, return 1 and mark the cell as selected. The game has been lost, and there will be no further function calls for this **game** array.

If the player has selected a cell without a mine, you need to ensure that the **hint** field of the relevant **struct cell** in the array contains the total number of mines in adjacent cells, before you return. You always still mark the cell as selected.

Furthermore, if the selected cell has 0 adjacent mines, you must recursively automatically select adjacent cells in all dimensions. If the adjacent cells also have 0 adjacent mines, continue the recursion. Only stop when there is a non-zero number of mines in adjacent cells to a particular cell. All recursively selected cells must also have the correct value for number of adjacent mines stored in the **hint** field of the corresponding **struct cell**.

The recursive algorithm is summarised in the pseudocode below. Note that it does not include any statements setting the **hint** or any other fields.

```
function select_recursion(CELL):
    if (CELL has no adjacent mines):
        for all cells x adjacent to CELL:
            if x is not selected:
                select x
                select_recursion(x)
```

If all cells except those that contain mines have been selected (by the player or automatically), return 2 and ensure all selected cells are correctly marked as selected. The game has been won, and there will be no further function calls for this **game** array.

Otherwise, return 0. There may be future function calls for this **game** array.

If the player selects a cell that is out of bounds, already selected or otherwise invalid, you should return 0 and do nothing else.

Notes and Hints

Your submission must not print any output to standard output or standard error.

Please note you may not use variable length arrays. You may use dynamic memory, but this is not necessary to complete this assignment.

You may be familiar with popular versions of the Minesweeper game that include features such as guaranteeing the first move is mine-free, and having "flags". For simplicity, such features are deliberately excluded from this assignment.

Your **`game`** array does not need to contain cells in any specified order. However, we suggest you use **row or column major order** to transform n-dimensional coordinates to a linear array index easily. There are many online resources available describing how this can be done.

Submission and Mark Breakdown

You are provided with a scaffold containing three files, **`minesweeper.c`** and **`minesweeper.h`** and **`params.h`**. The functions that you need to implement are in **`minesweeper.c`**. Please place all of your code in this file, or in included files as only this file will be passed for compilation.

The file **`params.h`** contains definitions for the constants **`MAX_ADJACENT`** and **`MAX_DIM`** (see above for details) that may change depending on the test case. Please ensure your code includes **`params.h`** (**`minesweeper.h`** already does this for you). The scaffold contains default values for the constants but you may change the values in the testing of your own code. However, any modifications to **`params.h`** in your submission will be overwritten by the testing code. You should use other files for code that you wish to form part of your submission.

Submit your assignment via Git to Ed for automatic marking. The marking program will check the validity of your **`game`** array by printing out a description of its contents, including the coordinates of your cells and their struct fields. If this output does not match expected output, you will be informed of the difference.

This checking output will produce text rows in the following format, one per cell in the array:

`cell at <coordinates> mined 0/1 selected 0/1 hint <hint value> adjacent to <number of adjacent cells> cells: <coordinates of all adjacent cells separated by spaces>`

Note that the hint field in the output will only be printed if it is required to be correct (that is, if the cell has been selected).

Your code will be compiled with the following options. Please note you may not use variable length arrays.

```
-Wall -Werror -Werror=vla -g -std=c11 -lm -fsanitize=address
```

The mark breakdown of this assignment follows (**6 marks total**).

- **4 marks** for correctness, assessed by automatic test cases on Ed. Some test cases will be hidden and will not be available.
- **2 marks** from manual marking. After the deadline, teaching staff will assess your submission and provide feedback. Manual marking will assess the style and structure of your code.

Warning: Any attempts to deceive or disrupt the marking system will result in an immediate zero for the entire assignment. Negative marks can be assigned if you do not follow the assignment problem description or if your code is unnecessarily or deliberately obfuscated.

Academic Declaration

By submitting this assignment you declare the following:

I declare that I have read and understood the University of Sydney Student Plagiarism: Coursework Policy and Procedure, and except where specifically acknowledged, the work contained in this assignment/project is my own work, and has not been copied from other sources or been previously submitted for award or assessment.

I understand that failure to comply with the Student Plagiarism: Coursework Policy and Procedure can lead to severe penalties as outlined under Chapter 8 of the University of Sydney By-Law 1999 (as amended). These penalties may be imposed in cases where any significant portion of my submitted work has been copied without proper acknowledgment from other sources, including published works, the Internet, existing programs, the work of other students, or work previously submitted for other awards or assessments.

I realise that I may be asked to identify those portions of the work contributed by me and required to demonstrate my knowledge of the relevant material by answering oral questions or by undertaking supplementary work, either written or in the laboratory, in order to arrive at the final assessment mark.

I acknowledge that the School of Computer Science, in assessing this assignment, may reproduce it entirely, may provide a copy to another member of faculty, and/or communicate a copy of this assignment to a plagiarism checking service or in-house computer program, and that a copy of the assignment may be maintained by the service or the School of Computer Science for the purpose of future plagiarism checking.