As I mentioned to Steven Teleki, I had completed this assignment about a year and half ago when I was applying for a position at Cognitive Scale. At the time the explanation of how I arrived at my design decisions was not a part of the assignment. I will now attempt to explain those design decisions. Thankfully I included sufficient comments that a quick review of the code reminded me of how the code works.

I chose to write the code in Python. I was becoming more comfortable with it having used it for about a year and a half. I also wanted the practice.

I saw that the printing of the field was going to be one of the challenging portions of the exercise especially as the space ship moves around and also when a mine that is on the edge of the field is detonated. The field would constantly be expanding and contracting with each movement and potentially with a detonation of a mine.

**Maintaining the Center of the Field**

I needed to figure out the field expansion and contraction algorithms so that I could correctly print the field, placing the ship at the center and printing only the minimum size field. I did this by example:

Ship Movement

1. For the following three fields:

. A .

. . .

B . C

A . .

.

.

A

When we move north, we must “add” two rows to the top side.

. . .

. . .

. A .

. . .

B . C

. . .

. . .

A . .

.

.

.

.

A

1. However, if the field looks like this (the southernmost mine is one row inside the trailing edge):

. . .

. A .

. . .

B . C

. . .

When we move north, we can “subtract” one row from the bottom and “add” one row to the top.

. . . **+1**

. . .

. A .

. . .

B . C

**-1**

1. And finally, for:

. A .

. . .

B . C

. . .

. . .

After moving north we “remove” two rows from the bottom:

. A .

. . .

B . C

So it seems like the algorithm has something to do with how far a mine is from the trailing edge.

If closest mine to trailing edge is on the edge:

Add two rows/columns to total number of rows/columns

If leading edge is north side or west side,

add 2 to the y coordinate (north)/x coordinate (west) of all mines

else: nothing

If closest mine to trailing edge is distance of 1 away from edge:

Add 1 to the x coordinate (east/west) or y coordinate (north/south) of all mines

If closest mine to trailing edge is distance >= 2

Remove two rows/columns from total number of rows/columns

If trailing edge is north side or west side:

add 2 to the y coordinate (north)/x coordinate (west) of all mines

Ship Destroys a Mine:

1. Starting position:

. A . . .

B . . . .

. . . C .

Ship fires alpha pattern (destroying A and C). The one empty row on the top has a matching empty row on bottom. They can both be trimmed. However the left and write margins are cannot be trimmed because there is not an equal number of columns that can be removed.

B . . . .

1. Starting position:

. A . . .

B . . . .

. . C . .

Ship fires gamma pattern (destroying A and C). The field can’t be trimmed from any of it’s four sides.

. A . . .

B . . . .

. . . . .

1. Starting position:

A . .

. . .

. . .

Ship fires alpha pattern. (Special condition) There are no mines left, therefore number of columns and number of rows = 1.

.

So after a fire pattern is executed the, the “margins” of the field may be able to be trimmed, as long as you can trim an equal amount from top and bottom -OR- from the left and right.

**Classes**

The first decision was how to represent the field, the spaceship, and the mines. My first inclination was to have a multidimensional list for the field where I imported the supplied field directly into the NxM list. This would combine the field and the mines. This was good, I had combined two of the elements, and printing would be trivial. However, this would force me to add and delete entire rows or columns from the array as the spaceship made movements within the field, and potentially whenever mines were destroyed. That sounded like more trouble than it was worth.

So, I tried to boil down the problem to what I really needed to track. What I came up with was:

1. Number of rows and number of columns (2 integers)
2. The location of the mines (n tuples)
   1. During printing of a field, any location that didn’t correspond with a mine would be treated as empty space
3. I didn’t need to track anything about the spacecraft other than its position, which is co-located with the center of the field.
   1. The center of the field could be deduced from the dimensions provided in 1) and 2) and therefore there was no need to create a separate entity representing the spacecraft.

Field

With these conclusions, I decided to combine the field the mines and spacecraft into a single class ‘Field’ with two integers and a storage container for n tuples. Excellent, I am now storing only the things I care about thus reducing my memory footprint, and I don’t have to deal with adding and removing rows and columns from a multidimensional list.

The coordinates for the mines would need to be stored as a pair of values and would need to be searchable. A natural choice was a dictionary:

dict<key: (x,y); value: char>

Even though the center of the field could always be deduced by the dimensions of the field, I decided it would be clearer in the code to refer directly to the center coordinates, so I included as member objects ‘center\_x’ and ‘center\_y’.

The constructor for the class Field could take a list of lines read from the file provided as input to the program. I can parse each row and find the location of all the mines, the number of columns would the longest line and the number of rows would the number of lines in the file.

Condensing functions together

When looking at the functions I have created I am often looking to see if there is a way to combine them because of similar function. The functions Field:North(); Field:South(); Field:East(); Field:West() are all doing similar things and infact are all operating off the same algorithm I have outlines above. However, I decided to leave them separate functions for clarity. I was having enough bugs associated with row/column incrementing that I though combining the functions would only make it harder to trace those if and when I made another mistake.

Error handling

1. There was no guarantee the field defined in the Field file was already minimally sized. Therefore I included in my design a Field::Trim() function that could handle trimming an arbitrary number of columns or rows if needed.
2. The assumed delimiter was a “space”. I accounted for cases where the spaces were omitted from the field file.
   1. I also included at the top of the Field.py file, a global variable for the file ‘\_FIELD\_FILE\_DELIMITER’. This could be set to which ever delimetter was used in a Field file.

**Testing**

Test Cases

I first used all the examples provided in the assignment to confirm the result I was producing was the same as those described in the examples.

1. Simple successful run
2. Successful run
3. Complete but script too long (too many instructions)
4. Missed a mine
5. Script to short

I then created some edge cases for both the fields and scripts.

1. Script/Field file names are not valid
2. A complicated field with an ‘a’ value mine at the start
3. Boundary condition with last mine being 'a' on last step
4. No delimiter between each cuboid in field file and
5. A script that contains every single missile fire pattern and every single direction to move including a blank line (implied falling) the field is also not uniform on the right side (rows do not have equal lengths) at the beginning
6. Mines have value of 'A' - testing the method of using numbers instead of characters to track the value of the mine. 'A' -> 'z'

Unit Testing

One area I focused on with my unit testing was what if the Field file passed into my program had some odd/bad data.

Shifting the coordinates of the mines as the number of rows or columns increase or decrease is an important part of my algorithm. During my unit testing of Field:Adjust\_Mine\_Coordinates(), I remember finding a bug that had somehow not been covered by my functional testing.

**Future Growth/Development/Improvements**

1. Ship class
   1. A ship class that takes any set setoff instruction definitions
      1. In the future North could mean move 2 cuboids in positive y direction
      2. New firing patters could be defined
   2. Field would still have the basic North, South, East, West movements that would always be defines as one movement
      1. The instruction definitions passed to class Ship would have to be in terms of those basic instructions
      2. Field would now accept a list of coordinates for a fire solution, rather than have them defined in the Field.Fire() function.
   3. I think the Ship class would accept a field object in as a default parameter in it constructor. The field would have been created outside the Ship class because it requires access to the Field file. Field makes sense as a component of Ship since the ship has exist in a particular environment i.e. a defined field.
2. Figure out what the issue is with relative/absolute paths
3. What happens if the field this is supplied has an even number of rows or columns?
   1. I could blindly add a row or column (functionally this be the same as adding a row to the bottom or to the right side). This would not change the coordinate of the mines in the inertial reference frame, but could be considered to be changing the coordinate of the mines in a reference frame centered on ownship. However, because ownship’s position is defined to be the center of the field, there was no center to the field.