Assignment 2 - Inference-Informed Action

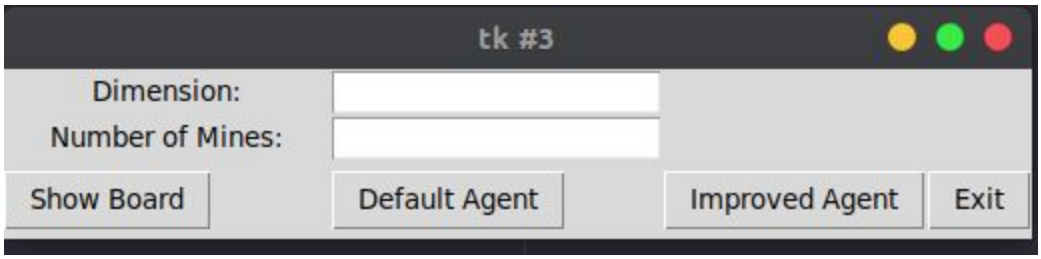
By Tatsat Vyas, Sri Jay Adarsh Gogineni, Andy Guo

Introduction

This project requires implementing the game minesweeper. In this game we are presented a blank grid of cells. Each turn the player must choose one of these cells and inside each cell contains either a mine or a safe space. If the player chooses a mine, they will lose the game. If the player chooses a free space, it will reveal a number that indicates how many mines are in its adjacent tiles. If the number is a 0, it will open up all the adjacent cells until it hits a number cell. In order to win this game, the player must keep choosing these safe tiles, using the number tiles as clues, until all the mines are marked and all the safe tiles are uncovered. The goal of this project is to create an algorithm that will play this game by giving the program a specific set of directions that the program must follow to help it determine the choice it makes.

Questions and Write up

Representation: How did you represent the board in your program, and how did you represent the information / knowledge that clue cells reveal? How could you represent inferred relationships between cells?

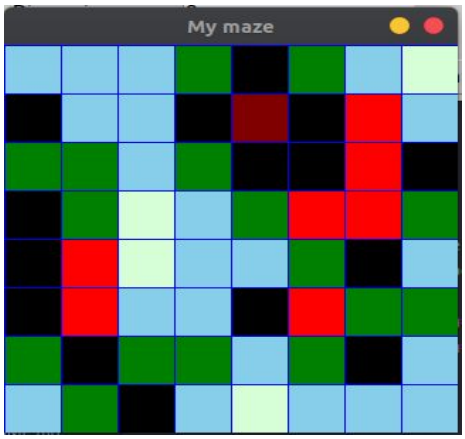


The user will first determine the dimension of the board and enter the amount of mines they want. The dimension will be used to find the total number of cells there are by doing dimension times dimension. The number of mines will equal the total number of -1s

that will be stored into the array and the rest will be stored with 0s, then it will be shuffled in the array. By changing the information stored in the array to a matrix we are able to mark the cells surrounding the mines with numbers that indicate how many mines are around it. To do this we will traverse through the matrix and at every -1 cell it will add 1 to those neighboring cells. The matrix after creating the minesweeper board will look something like this:

```
[[ 1  1  1  2 -1  2  1  0]
 [-1  1  1 -1  5 -1  3  1]
 [ 2  2  1  2 -1 -1  3 -1]
 [-1  2  0  1  2  3  3  2]
 [-1  3  0  1  1  2 -1  1]
 [-1  3  1  1 -1  3  2  2]
 [ 2 -1  2  2  1  2 -1  1]
 [ 1  2 -1  1  0  1  1  1]]
```

We use the graphics library to help us visualize the events happening while our algorithm selects its path. Each number in the current matrix will get converted to a different rgb number. For example, the cells with mines, which are marked -1 right now it will be stored into another matrix in colors as [0,0,0] which will be translated to black. The other representations in our GUI: 0 is light green color, 1 is set to sky blue, 2 is set to green, 3 is set to red, 4 is set to purple, 5 is set to maroon, 6 is set to turquoise, 7 is set to pink, and 8 is set to gray. Now our board will have different colors corresponding to if its a mine or a safe cell that contains a number corresponding to the adjacent mine cells. For example, the above matrix will be colored in this way:



Inference: When you collect a new clue, how do you model / process / compute the information you gain from it? In other words, how do you update your current state of knowledge based on that clue? Does your program deduce everything it can from a given clue before continuing? If so, how can you be sure of this, and if not, how could you consider improving it?

Basic agent: In our basic algorithm we follow these set of rules given to us. For each cell there is a number associated with it, this is our clue. By subtracting the number of revealed mines to the number given in a cell, we can assume that every hidden neighbor is a mine. For a given cell if the total number of safe cells, 8 minus the clue, minus the number of revealed safe cells is the number of hidden neighbors then every neighbor is considered safe. If there is no cell that can be identified as safe or a mine from these given set of rules then the algorithm will randomly pick a cell to reveal. This will require us to store the following information: the safe cells, the mine cells and the number of hidden cells around each identified cell.

To do this, we created a new matrix that will store all the information about the revealed cells that the program discovers while traversing through the matrix. Each time it picks a matrix it will reveal it as a mine or a safe cell. If it is a safe cell the new matrix will obtain the number 9 at the same spot that the original matrix has and if it was a mine then the new matrix will have a -1 stored at that spot. To check the surrounding cells it will check to see if the new matrix has been revealed or not by determining if it has a 9 or a -1. By applying this we can use the two basic identification rules given to us. If the first two set of rules fails then we will randomly choose a cell in our board and continue from there until the whole board is revealed.

One way to improve this algorithm is how we are randomly choosing from the matrix. As you can see we select randomly without previous knowledge of the matrix that we had. To improve this we could randomly choose that were adjacent to the ones that we have already discovered previously. This way we will be able to compare and update more with the knowledge of the previously determinable cells. Since our algorithm chooses randomly from without considering the previously determined cells, our odds of randomly selecting a mine will always be increasing since the number of safe tiles is always

decreasing. By considering the previously determined safe cell, it will reduce the chance that we will randomly select a cell that has a mine.

Decisions: Given the current state of the board, and a state of knowledge about the board, how does your program decide which cell to search next? Aside from always opening cells that are known to be safe, you could either a) open cells with the lowest probability of being a mine (be careful - how would you compute this probability? or b) open cells that provide the most information about the remaining board (what could this mean, mathematically?). Be clear and precise about your decision mechanism and how you implemented it. Are there any risks you face here, and how do you account for them?

Custom agent: The custom agent works on the concept of linear algebra. It basically creates a system of equations for each bordered box. The bordered box is basically any box that has one or more undiscovered neighbors. It then creates a system of equation for each of the undiscovered neighbors and puts it equal to the number in the box - discovered mines around.

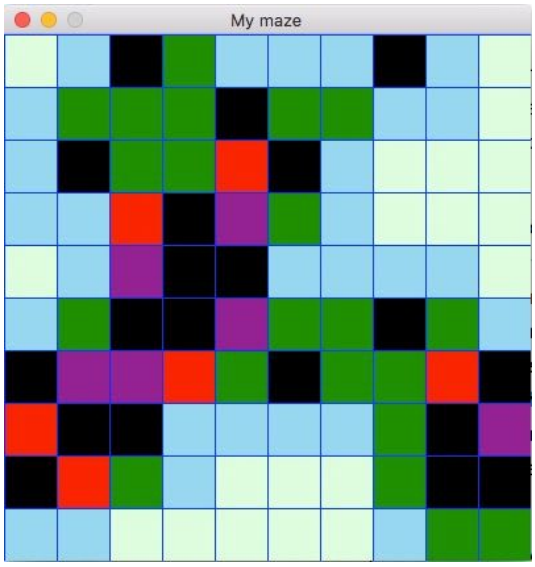
For example, if a box has 3 undiscovered boxes, 2 identified mines and the number in the box is 4 then the equation would be:

A + B + C = 4 -2 = 2

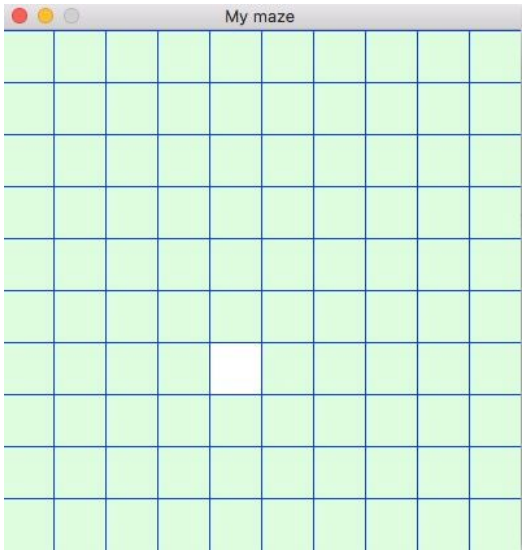
It does this for all the boxes and then it compares how many of the neighbors does each box have in common with other. If two boxes have all common but one uncommon each then you can identify that both of them are mines or one of them is mine and one is safe. This can be determined because of the binary condition of each undiscovered box. Each box can either be 0 or 1. This does not always work since at the beginning of the program there is not enough known value to do the system of equations with. In that case we keep making random guesses. This is the one thing that hurts the performance of our agent.

Performance: For a reasonably-sized board and a reasonable number of mines, include a play-by-play progression to completion or loss. Are there any points where your program makes a decision that you don't agree with? Are there any points where your program made a decision that surprised you? Why was your program able to make that decision?

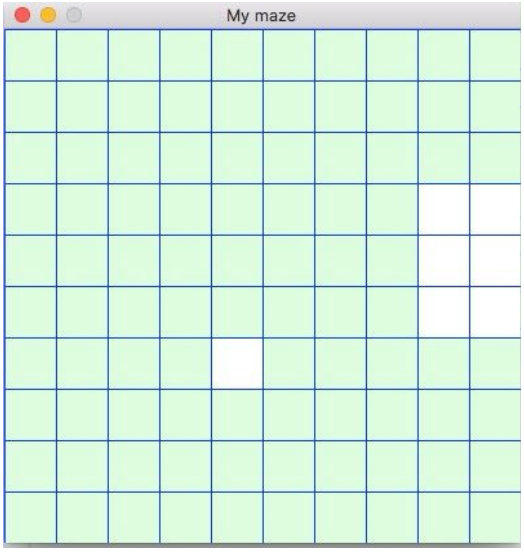
Basic agent:



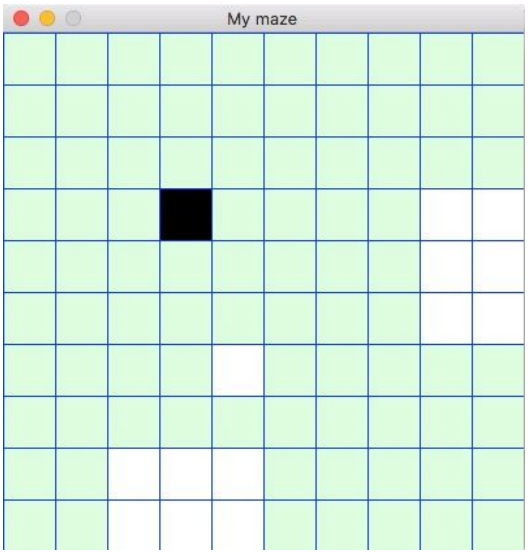
This is the board at the start of the default agent. As you can see this is a 10 x 10 board with 20 bombs shown (the black tiles). And the other colors of the board represents the number of adjacent mine cells next to it.



The first iteration of the algorithm selects a cell in the matrix and updates it to a new matrix the cell that was selected is colored white meaning it was a safe tile.



Iteration 2. Since the previous selected we were unable to determine if it was a safe or a mine surround it, the algorithm will randomly select a new cell in the matrix. Using the two rules given to us the algorithm continues using it until it cannot determine any more safe cells. Now it must randomly choose a new cell again. This process will be repeated again and again.

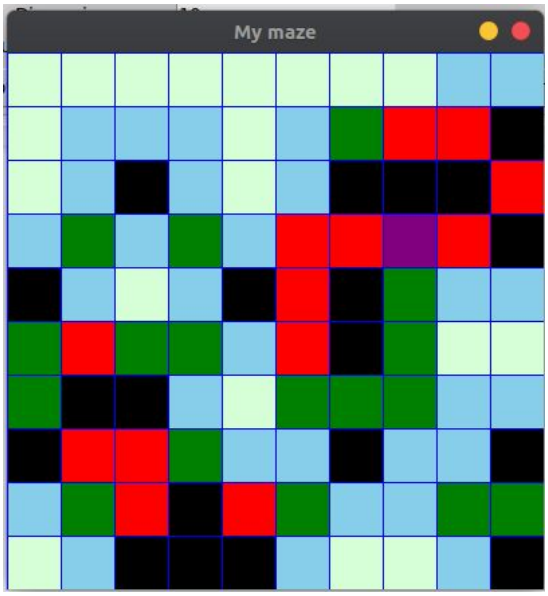


In iteration 3, we select a cell and determine all the cells that we're able to determine to be safe. However, this stops as soon as we cannot determine any more safe cells

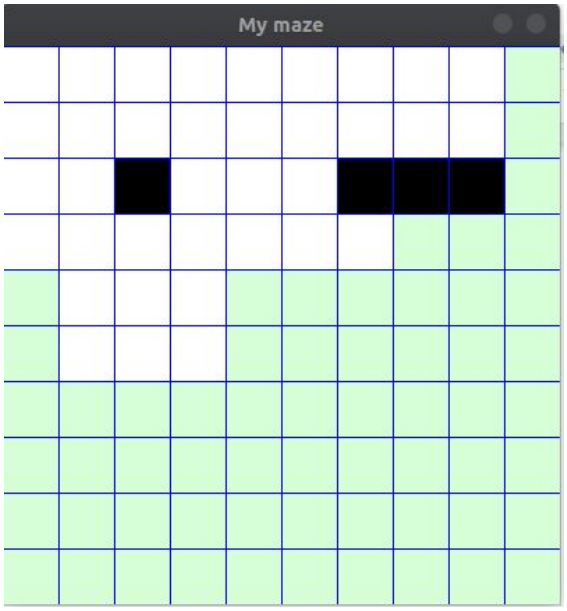
In iteration 4, we randomly selected a mine cell. Then the algorithm will select a new cell and continue from there. Until it reaches the original matrix. However the basic agent will lose here since it selected a mine cell.

Since the basic agent randomly selects any undiscovered cells. Originally the chances of selecting a mine was 20% however after iteration 3 the chance of selecting a mine goes up to ~23%. This is because the chances of randomly selecting a mine went from 20/100 to 20/87. Therefore the longer we play the game our odds of landing on a mine cell is always increasing.

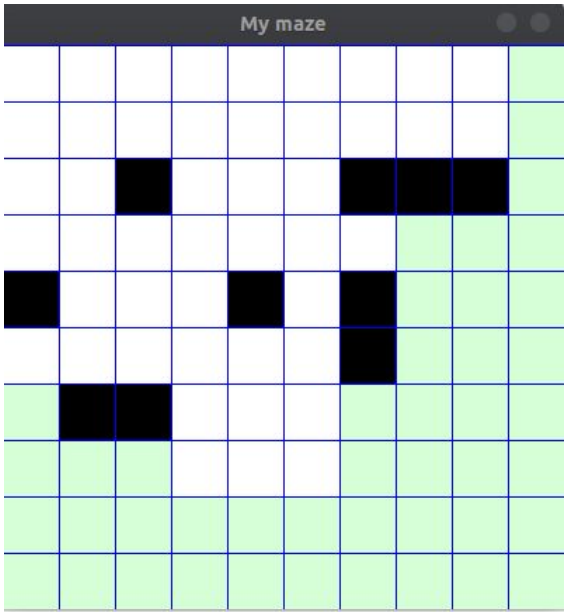
Custom agent:



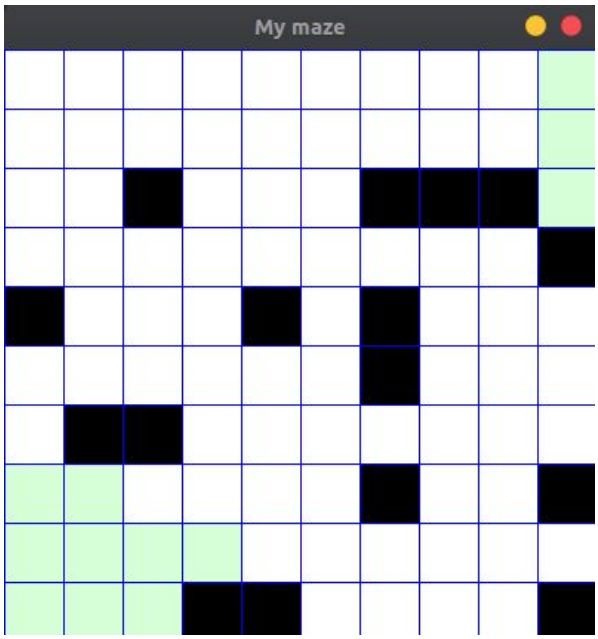
This is the board at the start of the custom agent. As you can see this board contains a 10 x 10 matrix with 20 number of mines.



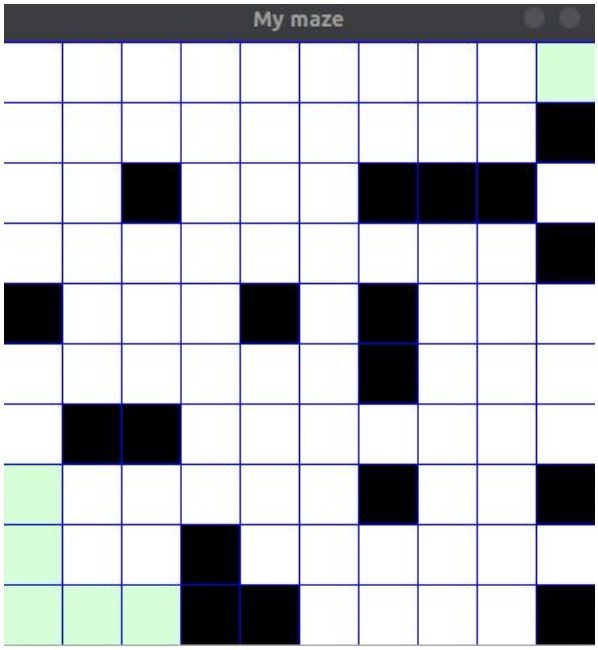

```
<-------found mines------>
[(2, 6), (2, 7)]
<-------found mines------>
[(2, 8)]
<-------found mines------>
[(2, 2)]
<------now doing smart pick---------->
```




```
<------now doing smart pick---------->
<-------found mines------>
[(4, 4)]
<-------found mines------>
[(4, 0)]
<-------found mines------>
[(4, 6), (5, 6)]
<------now doing smart pick---------->
<-----found mine using smart pick--------->
(6, 2)
<-----found mine using smart pick--------->
(6, 1)
```




```
<-------found mines------>
[(7, 6)]
<-------found mines------>
[(3, 9)]
<-------found mines------>
[(9, 4)]
<-------found mines------>
[(7, 9)]
<-------found mines------>
[(9, 9)]
<------now doing smart pick---------->
<-----found mine using smart pick--------->
(9, 3)
```



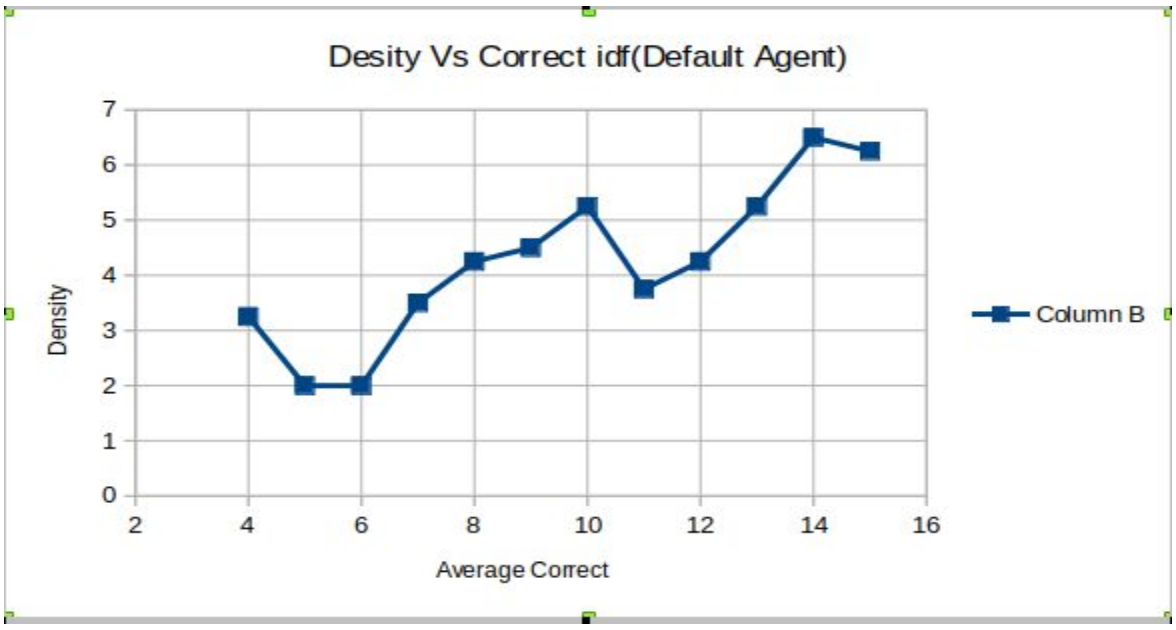

```
<-------found mines------>
[(8, 3)]
<-------found mines------>
[(1, 9)]
<------now doing smart pick---------->
```

Next Click: Ends the game

```
<------now doing smart pick---------->
<------Steped on a mine------>
<-------found mines------>
[(7, 0)]
```

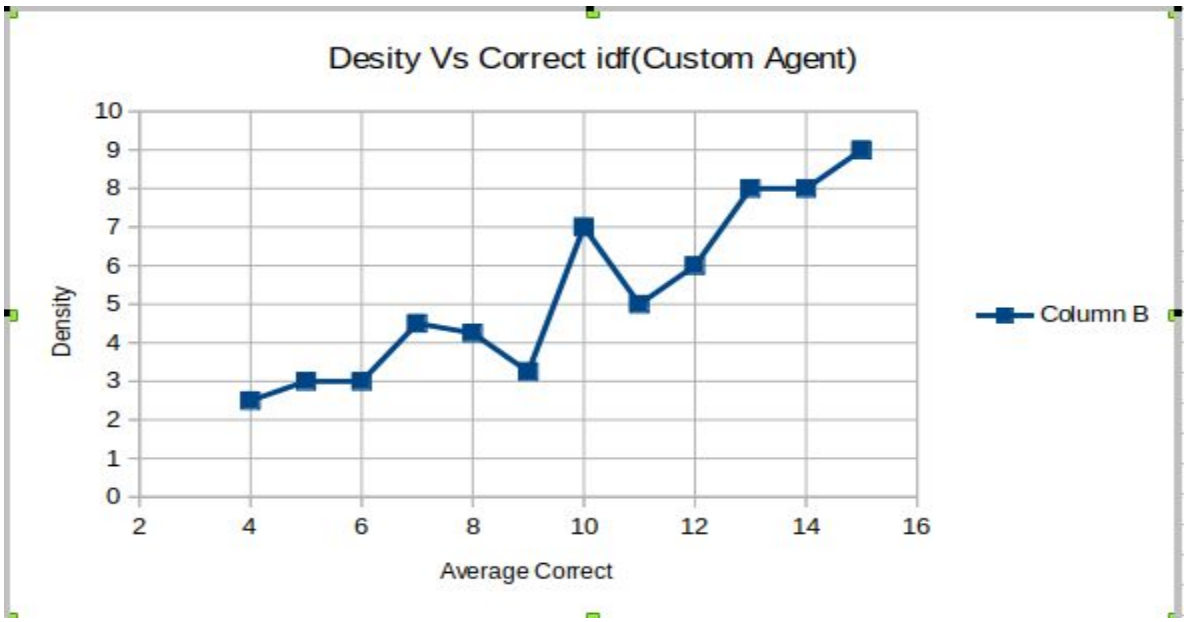
As we can see above, the agent starts at a random location in the grid, and goes through the algorithm. When it finds locations that are bombs, they are marked and outputted onto the console window as “found mines” and the location of the mines are displayed. This algorithm keeps track of all the mines it found, as well as all the mines it has stepped on. Even if the algorithm steps on a mine, it will continue forward, using the information available to it, constantly backtracking and seeing what the next best choice is as well as where the mines are.

Performance: For a fixed, reasonable size of board, plot as a function of mine density the average final score (safely identified mines / total mines) for the simple baseline algorithm and your algorithm for comparison. This will require solving multiple random boards at a given density of mines to get good average score results. Does the graph make sense / agree with your intuition? When does minesweeper become ‘hard’? When does your algorithm beat the simple algorithm, and when is the simple algorithm better? Why?



The above graph tries to examine the performance of the default algorithm. The x-axis represents the number of correctly identified and the y-axis represents the density of the grid. As you can see the number of correctly identified mines increases, however, there is a part where the algorithm does not perform as expected. Because of the randomness of the

algorithm, the number of correctly identified mines depends on the number of times the algorithm makes a random choice. This can be seen by the fluctuations in the graph. The correctly identified is not a constant increase but a variation of how lucky the algorithm gets.



The above graph is the graph of the customs agent. As you can see the performance is clearly better than the default agent. There is still a factor of randomness involved since it does make a random choice when the algorithm can not make any robust predictions. Therefore, by making a board to have more mines will influence the performance of the algorithm rather it will select a free cell or a cell that will contain mines in them. Due to randomness by increasing the number of mines while having the same dimension will also increase the mine and make it 'harder' since the custom agent will use probably to reduce the chance of it selecting a mine while the basic agent will continue to select randomly as the number of safe cells will decrease resulting in a higher chance of it selecting a mine cell.

Efficiency: What are some of the space or time constraints you run into in implementing this program? Are these problem specific constraints, or implementation specific constraints? In the case of implementation constraints, what could you improve on?

One constraint we have in our program runs into is that the algorithm never considers the total number of mines that was in the board. This means after each step, our algorithm will go through the whole matrix over and over again. This is to help us determine which cell has the lowest 'probability' that it will have a mine in it. If the algorithm was able to be able to determine the number of mines it will save us time since our current custom algorithm does not consider this principle. If we were able to determine the number of mines we can skip the step of the algorithm going through the whole mine and have it select from the other tiles. And at the very end of the if the number of predicted mine cells equals the total mines given we can skip any guessing steps and conclude that all the remaining undiscovered cells will be considered safe. Therefore with this one knowledge it will drastically improve the space and time of our program.

Improvements: Consider augmenting your program's knowledge in the following way - tell the agent in advance how many mines there are in the environment. How can this information be modeled and included in your program, and used to inform action? How can you use this information to effectively improve the performance of your program, particularly in terms of the number of mines it can effectively solve? Re-generate the plot of mine density vs expected final score for your algorithm, when utilizing this extra information.

There are ways to improve the agent's knowledge, if we know the number of mines there are in the environment. So far, we have been working under the assumption that there are unlimited number of mines in the game. Because of this, we aren't able to confirm the exact probability of the mine being in that specific spot. If we know the total number of mines, new information becomes available to us. We can then go through the entire bordering cells of what has been clicked, and calculate the total probability that there is a mine in each of the bordering cells. We know the number of neighboring mines each cell has and we know the total number of mines, so we can assign each cell a probability between 0 to 1 of there being a mine in that specific cell. If we have these probabilities, it's a matter of choosing the smallest probability of the mine being in that specific spot and picking that one as the next best choice to explore. This would then open up more bordering cells and then we could run the algorithm again to calculate new probabilities based on the number of mines remaining. This could drastically improve the performance

of the agent, because each time there is a decision to be made, the agent will make the most informed choice always. This is due to it having an empirical probability assigned to it. When probabilities are in the picture, the agent will always make the best choice. However, since minesweeper is a game of luck in the end of the day, there will always be cases where the agent can't make an informed decision even with the probabilities because there might be times where all the bordering cells have equal probabilities. When this happens, the agent will be forced to guess. However, despite this, this version of the agent will almost always give the best results.