Analysis for Project II: Minesweeper

Ilana Zane, William Bidle, Rakshaa Ravishankar

March 13, 2020

1) 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?

Our minesweeper board was generated by a two dimensional array with randomly placed mines. We created a main board and an agent board. Our main board displayed the contents of every cell (i.e. whether it was a clue, safe cell, or a mine) and the agent board displayed what agent could see, which was only updated after a new cell was uncovered or an inference was made. We used these two boards to demonstrate what every cell contained and how our agent was progressing throughout the game. For our basic algorithm, we had our program sort through its knowledge base and add any definite mines our 'mine fringe', a list of cells to immediately flag to make sure that we do not select any of those cells as our next move. We would then have our program check its 'safe fringe', a list that contained all safe cells, to choose one of those cells as our next spot. The two fringes were populated whenever we checked the neighbors of a cell. Our knowledge based contained the coordinates of every uncovered cell, their associated clue and the amount of neighbors that cell has. We would use the knowledge base to make basic inferences about our current move and then continue on with the game. Our inference algorithm was represented as a matrix that was populated with linear equations associated with every uncovered cell on the board. We used linear algebra operations to simplify our equations and make inferences across them. If we had uncovered any definite information, such as whether a cell was safe or a mine, we would update our mine fringe or safe fringe with this new information and update our knowledge base.

2) 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?

When we uncover a new clue, whether it was chosen through inference or a random move, we determine whether it is a clue or a mine. If it is a mine no new information is uncovered. If the uncovered cell presents us with a clue, we count the amount of neighbors this cell has and perform our basic inferences through a series of conditional statements. If we see that the number of neighbors is equal to the value of the clue we know that all neighbors are mines and those neighbors can be added to the mine fringe. If the number of surrounding mines minus the value of the clue equals zero we know that all surrounding neighbors are safe and can be added to the safe fringe. If we are unable to successfully determine whether or not a cell is a mine or a safe cell, we add our newly uncovered cell to our knowledge base. The knowledge base contains all such cells along with their associated coordinates, their

clue value, and the number of neighbors they have. As our agent progresses, we updated our knowledge base to keep track of how many neighbors a cell has as new cells are revealed across the board. We also continuously update our mine fringe and safe fringe. Our more advanced algorithm is able to make inferences across several clues. If our basic logic fails, meaning that were unable to successfully determine the nature of an uncovered cell, we model the board through a matrix that contains linear equations. For every uncovered clue, there is an associated linear equation. In every linear equation, we use the number "1" to represent an uncovered cell and a "0" to represent every unopened cell/mine. We have a vector that contains the values of every clue that is represented in our matrix. Once our augmented matrix is populated with all equations, we reduce the equations until the matrix is as close as possible to a reduced row echelon form. From there, we store our reduced linear equations into our knowledge base where we analyze the information we have through conditional statements in order to determined whether or not we have definitively discovered a cell that is safe or is a mine. These conditional statements are the same ones that are used when analyzing our basic algorithm. If at any point the sum of the variables in a reduced equation are equal to 0, then we know that each variable and its respective coordinate should be updated in the knowledge base as a safe spot. We also check to see if the number of variables in our reduced equation matches the value of the clue. If they are equal this means that all of the variables in that equation are mines. We know that our inference algorithm can deduce everything that it can because for a given clue all of the possible equations and moves are considered across the board. As seen in Figure 1, our agent that uses inference makes less random moves than the agent without inference because our inference agent is able to deduce more information from the board. We also watched through a few solutions step by step to see if our agent missed any crucial moves. We failed to find any anomalies or surprising moves. Thus, we can say that our inference agent is working.

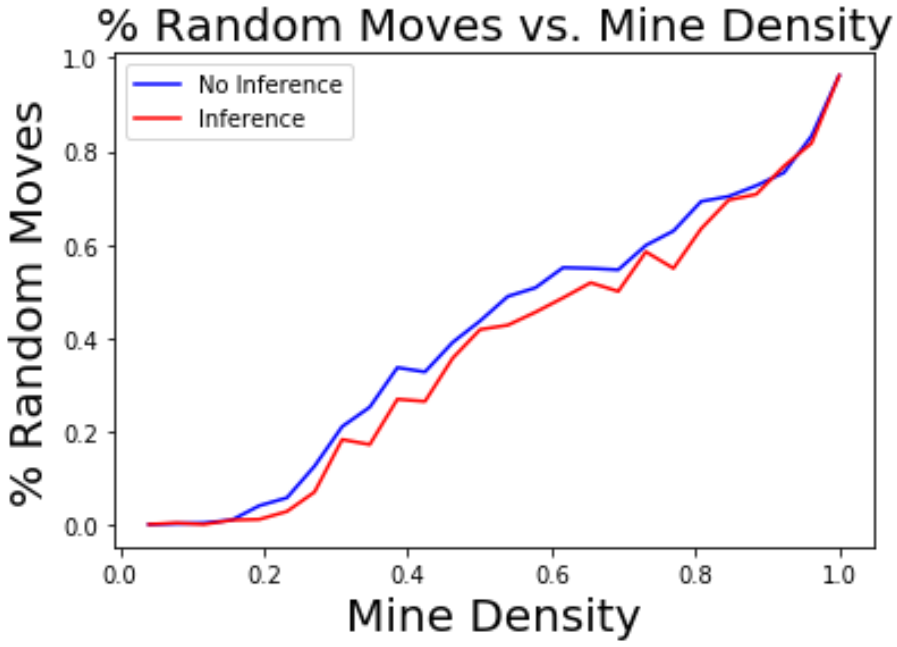

Figure 1

3) 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?

To see a play by play progression of our agent solving the board watch StepByStepAnalysis.mov (We also included MinesweeperVideo.mov that shows the inference agent solves a 15x15 maze with low density and high density). The movie displays our advanced agent solving the board of size 15x15 and a density of 0.4. Below are snapshots from the movie that displays instances where our basic logic makes an inference from one clue and another where our advanced agent makes an inference across multiple clues. Figures 2 and 3 display how the basic algorithm makes a decision. In Figure 2 the yellow circle represents a flag for the cell with the clue of 4 and the red circle represents all of the neighbors of that clue. Whenever the neighbor of a clue is uncovered or marked as a mine, a parameter in our knowledge base temporarily decrements the value of the clue. In this instance, as soon as the neighbor of 4 is marked as a flag, the clue value decrements to 3. Our basic logic was then able to see that the updated clue of 3 has three remaining neighbors and can infer that all of them are mines that need to be flagged (flags are in the green circle). In Figures 4 and 5 our advanced logic was able to make an inference that may not be intuitive to the average player. When making an inference for the cell with clue 2 (in the yellow circle) our agent realizes that the cell with clue 1 (right above clue 2) must have a mine within the blue circle. This leaves the remaining mine of clue 2 in the red circle. With this information the agent can come to the conclusion that the flag must go within the red circle. In Figure 4 we see that a flag is correctly revealed within the red circle. Our advanced agent thus utilized information from two different clues to come to this conclusion. Overall, our advanced agent did not make any surprising moves as it followed every step in our program as expected, deducing as much as it could with what it was given. The only moments where a mine was uncovered was through selecting a random cell when no more inferences could be made.

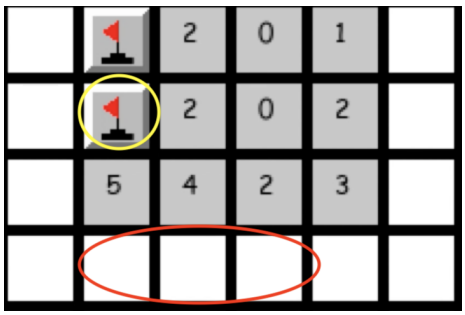

Figure 2

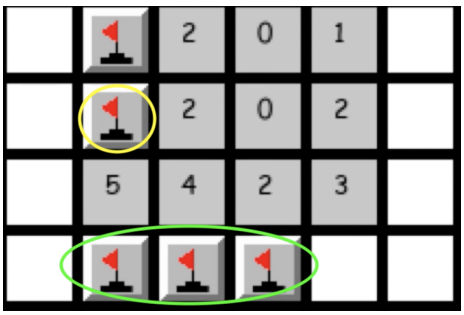

Figure 3

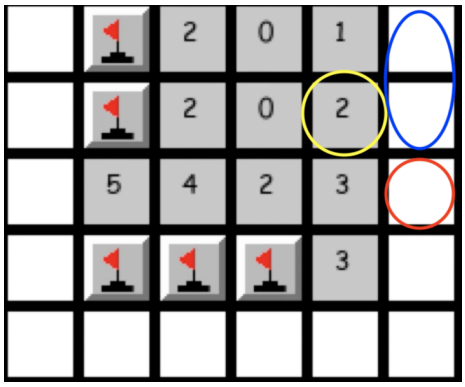

Figure 4

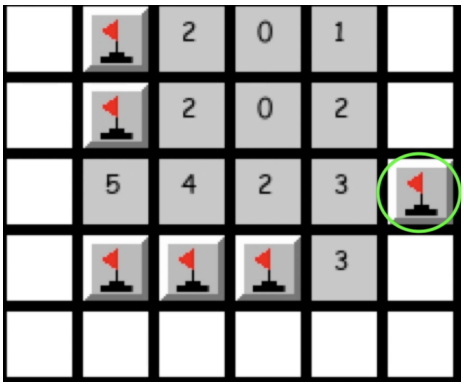

Figure 5

4) 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?

For a board of size 20x20 we ran our simple algorithm and our inference algorithm multiple times at different densities to find their average score at each density, as seen in Figure 6. We anticipated that when the mine density was closer to 0.0 our basic algorithm and our advanced algorithm would have similar performances since these mazes would be able to be solved with basic logic. We also anticipated that as the mine density approached 1.0, the chances of our algorithms selecting a mine was much higher. When a mine is selected no other new information is revealed and our knowledge base cannot be populated with useful information. Without enough information the algorithms resort to random moves, further increasing our chances of selecting another mine. As seen in Figure 6, our inference algorithm generally returns a better score than our basic algorithm. This is because after a certain density, around 0.15 and beyond, our inference algorithm has enough information to analyze multiple clues at the same time and make more inferences than our basic algorithm. The chances of our inference algorithm to select a mine upon making a random move is lower, so the random move is likely to uncover useful clues that can be added to our knowledge base.
Our minesweeper game becomes "hard" at a mine density of above 0.6. At this density, both algorithms have a score of approximately 0.5 and the score increasingly becomes lower tending towards a score of 0.0. Our basic algorithm, on average, does not return a better score than our inference algorithm. However, at a low mine density and high mine density,

the scores between the algorithms are very close, but our basic algorithm has a better time complexity throughout– finishing its analysis of clues much faster than the inference algorithm, as seen in Figure 7. Because of this, we conclude that only at a very low and very high mine density does our basic algorithm perform better.

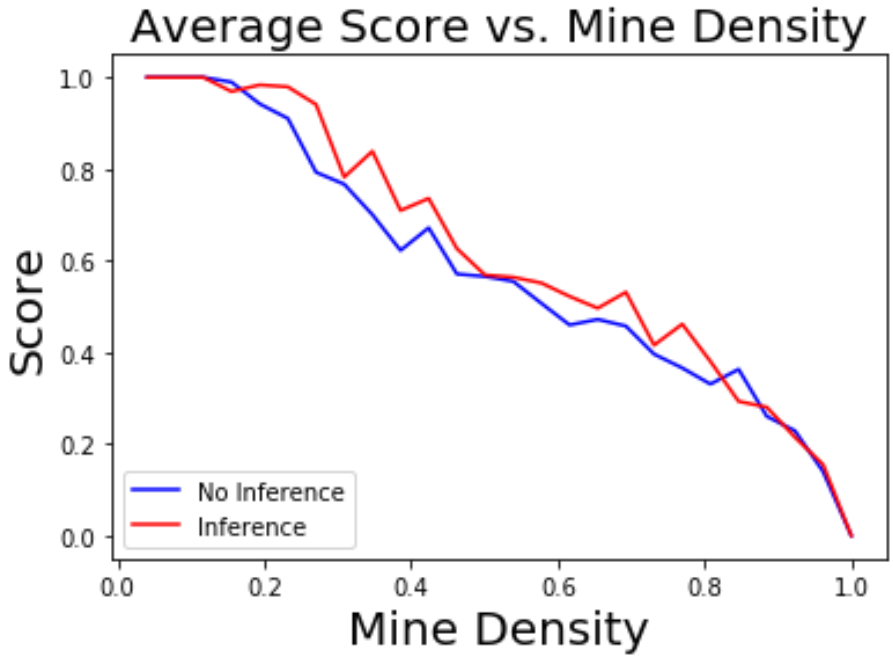

Figure 6

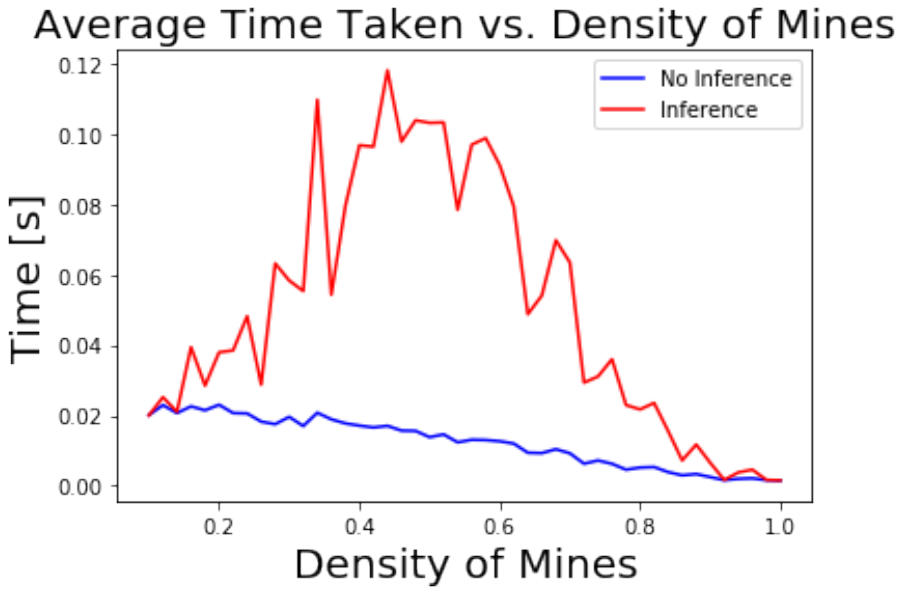

Figure 7

5) 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?

Space and time constraints arose while we were implementing our inference algorithm. As the board size increased our inference algorithm took longer to run because we had to analyze more clues. This is a problem specific constraint because with any game of minesweeper as the board size increases, the time it will take to run through all of the clues and to draw comparisons will increase. However, this issue could have been improved through our implementation. For our inference algorithm we chose to represent our knowledge base as a matrix that stores linear equations for all of the uncovered cells in our board. In order to decrease the time it takes for our inference algorithm to draw comparisons we could have only taken cells that are adjacent to our current position into consideration. If we had done this our matrix would've been populated with equations that were relevant only to our current position. When we take the entire board into consideration for every clue our equations are of size $\dim^2$, so it would be better to have a matrix that only contains the equations for immediate cells.

6) 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.

After adding the number of mines that are in the environment to the agent's knowledge base, we checked the performance. Figure 8 below shows the result for our advanced algorithm with and without the additional information. It can be seen that there is little to no improvement in in the overall score achieved for any density, which is fairly expected. By giving the agent this additional information, we can improve our last resort random choice when nothing else can be deduced from inference. For a given clue, the probability that its neighbors contain a mine is equal to the clue divided by the number of neighbors. For example, a clue of 1 with three neighbors would result in a probability of one-third for each cell to be a mine. For the rest of the cells in the maze the probability of a random move would be changed to the total mines remaining divided by the number of cells left. We would then check if the probability of a cell containing a mine due to one clue would be less than the overall probability that a cell contained a mine and choose the one with the lower value. By doing this, we would ultimately improve our random pick method. Even though there were no overall improvements to the score, there was a way to improve the performance with the new information. If, by chance, the algorithm already identified all of the mines in the maze and either flagged or set them off then the game could end there and a final score would be given. Not having to go through every cell to complete the game allows for a better run time. While a benefit, this only occurred towards the end of almost every game (if at all) and would therefore only speed up the runtime by a small amount.

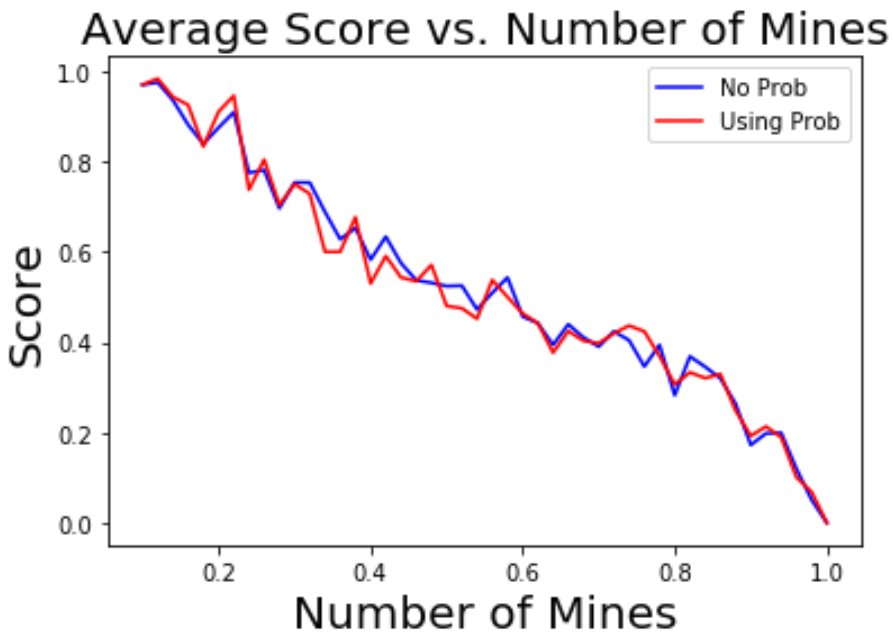

Figure 8

☺Member Contributions☺
everyone contributed equally