CS 440: Assignment 2 - Inference-Informed Action

16:198:440

This project is intended to explore how data collection and inference can inform future action and future data collection. This is a situation frequently confronted by artificial intelligence agents operating in the world - based on current information, they must decide how to act, balancing both achieving a goal and collecting new information. Additionally, this project stresses the importance of formulation and representation. There are a number of roughly equivalent ways to express and solve this problem, it is left to you to decide which is best for your purposes.

1 Background: MineSweeper

In the game minesweeper, you are presented with a square grid landscape of cells. Hidden in some of the cells are 'mines'. At every turn, you may select a cell to uncover. At this point, one of two things will happen: if there is a mine at that location, it explodes and you lose the game; if there is not a mine at that location, it reveals a number, indicating the number of adjacent cells where there are mines. If the cell reveals 0, all the surrounding 8 cells are empty of mines. If the cell reveals 8, all 8 adjacent cells must have mines. For any value in between, for instance 4, you know that half of the adjacent cells have mines, but you cannot be sure by this clue alone which half are dangerous and which half are safe. *Note: Common implementations of MineSweeper have other features, for instance instantly opening cells that are obviously clear to save you some time. We are ignoring that for the purpose of this assignment, as well as making some other small changes. Read carefully.*

The goal of the game is to identify the locations of all the mines (if possible); by collecting clues and information, you can begin to infer which cells are dangerous and which are safe, and use the safe cells to collect more information. This process is iterated until, hopefully, all cells are either uncovered, marked as clear, or marked as mined.

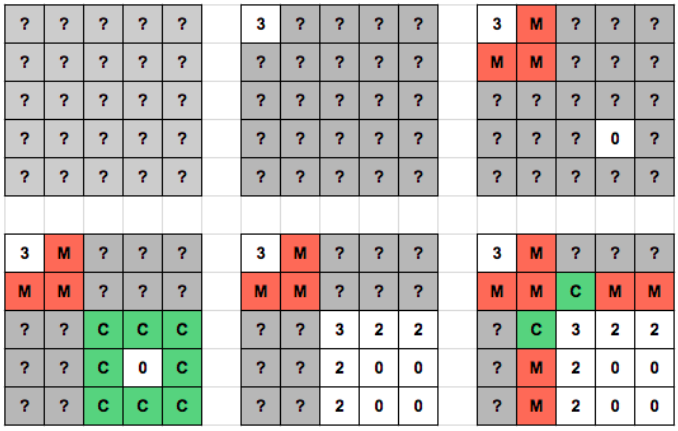

Figure 1: In the first board, all cells are unknown. The first move reveals a 3, in which case we may infer that all three surrounding cells have mines. We mark them as such, and know not to ever search those cells. The second move reveals a 0, in which case we know that all 8 surrounding cells must be clear. We mark them as clear, and know that we can search them without worry. Collecting data from the clear cells, we are able to infer the locations of more mines.

The goal of this project is to write a program to play MineSweeper - that is, a program capable of sequentially deciding what cells to check, and using the resulting information to direct future action.

2 Program Specification

There should effectively be two parts to your program, the **environent** representing the board and where the mines are located, and the **agent**. When the agent queries a location in the environment, the environment reports whether or not there was a mine there, and if not, how many of the surrounding cells are mines. The agent should maintain a knowledge base containing the information gained from querying the environment, and should not only be able to update its knowledge base based on new information, but also be able to perform inferences on that information and generate new information.

- The environment should take a dimension d and a number of mines n and generate a random $d \times d$ boards containing n mines. The agent will not have direct access to this location information, but will know the size of the board. *Note: It may be useful to have a version of the agent that allows for manual input, that can accept clues and feed you directions as you play an actual game of minesweeper in a separate window.*
- In every round, the agent should assess its knowledge base, and decide what cell in the environment to query.
- In responding to a query, the environment should specify whether or not there was a mine there, and if not, how many surrounding cells have mines.
- The agent should take this clue, add it to its knowledge base, and perform any relevant inference or deductions to learn more about the environment. If the agent is able to determine that a cell has a mine, it should flag or mark it, and never query that cell. If the agent can determine a cell is safe, it's reasonable to query that cell in the next round.
- Traditionally, the game ends whenever the agent queries a cell with a mine in it - a final score being assessed in terms of number of mines safely identified.
- However, extend your agent in the following way: if it queries a mine cell, the mine goes off, but the agent can continue, using the fact that a mine was discovered there to update its knowledge base (but not receiving a clue about surrounding cells). In this way the game can continue until the entire board is revealed - a final score being assessed in terms of number of mines safely identified out of the total number of mines.

This last modification allows the game to 'keep going' and avoids the situation where you accidentally find a mine early in the game and terminate before the game gets interesting.

You may either do a GUI or text based interface - the important thing for the purpose of the project is the representation and manipulation of knowledge about the mine field. This will draw on the material discussed in a) Search, b) Constraint-Satisfaction Problems, and c) Logic and Satisfiability. You must implement a basic MineSweeper agent, described in the next section, and your own agent as an improvement on the basic one.

As usual, you must create your own code for solving the main issues and algorithmic problems in this assignment, but you are welcome to use external libraries for things like visualizations, etc.

2.1 A Basic Agent Algorithm for Comparison

Implement the following simple agent as a baseline strategy to compare against your own:

- For each cell, keep track of
  - whether or not it is a mine or safe (or currently covered)

  - if safe, the number of mines surrounding it indicated by the clue
  - the number of safe squares identified around it
  - the number of mines identified around it.
  - the number of hidden squares around it.

- If, for a given cell, the total number of mines (the clue) minus the number of revealed mines is the number of hidden neighbors, every hidden neighbor is a mine.
- If, for a given cell, the total number of safe neighbors (8 - clue) minus the number of revealed safe neighbors is the number of hidden neighbors, every hidden neighbor is safe.
- If a cell is identified as safe, reveal it and update your information.
- If a cell is identified as a mine, mark it and update your information.
- If no hidden cell can be conclusively identified as a mine or safe, pick a cell to reveal at random.

2.2 An Improved Agent

The algorithm described for the basic agent is a weak inference algorithm based entirely on local data and comparisons - it is effectively looking at a single clue at a time and determining what can be conclusively said about the state of the board. This is useful, and should be quite effective in a lot of situations. But not every situation - frequently multiple clues will interact in such a way to reveal more information when taken together.

Your improved agent should model the knowledge available, and use methods of inference to combine multiple clues to draw conclusions when possible or necessary. Note that 'knowledge' to model includes potentially: a) whether or not the square has been revealed, b) whether or not a revealed cell is a mine or safe, c) the clue number for a revealed safe cell, and d) inferred relationships between cells.

3 Questions and Writeup

Answer the following questions about the design choices you made in implementing this program, both from a representational and algorithmic perspective.

- 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?
- 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?
- Decisions: Given a 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?

- 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?
- 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?
- 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?
- 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.

4 Bonus: Dealing with Uncertainty

Suppose that your mine detector has a certain false negative rate ($p_{\mathbf{neg}}$) at which it mistakenly reports a mined cell to be empty. It additionally has a false positive rate ($p_{\mathbf{pos}}$) at which it mistakenly reports an empty cell to be mined. This means that you cannot always trust the clue number / mine counts to be accurate.

How could you adapt your solve to each of the following cases (separately)? Try to implement at least one, and experiment with what kind of performance hit results, in terms of your ability to safely clear the board.

i) You know in advance $p_{\mathbf{neg}} = 0, p_{\mathbf{pos}} = 0.2$.

ii) You know in advance $p_{\mathbf{neg}} = 0.2, p_{\mathbf{pos}} = 0$.

ii) You know in advance $p_{\mathbf{neg}} = 0.2, p_{\mathbf{pos}} = 0.2$.

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.

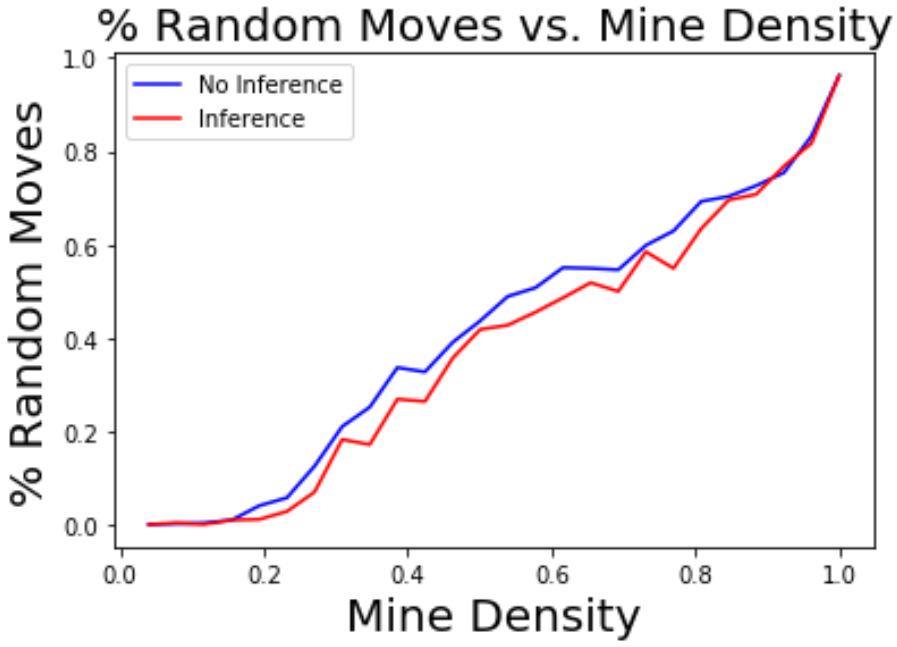

Figure 1

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.

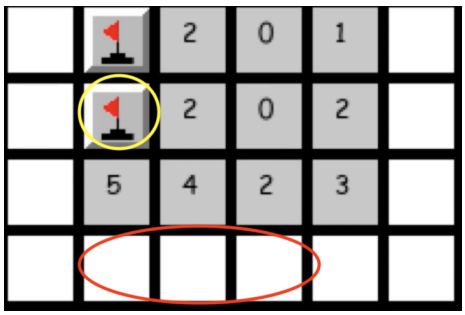

Figure 2

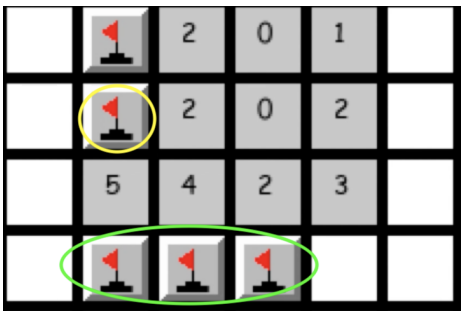

Figure 3

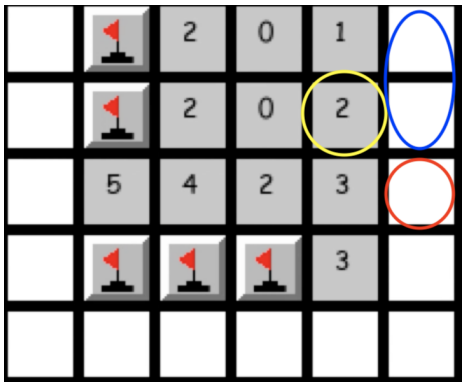

Figure 4

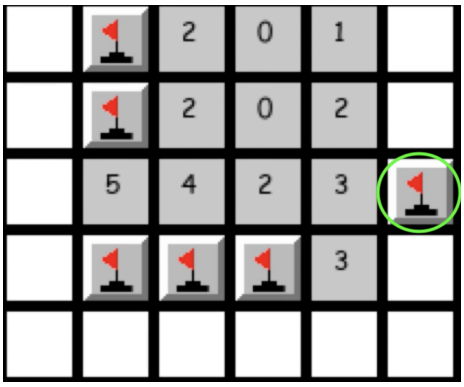

Figure 5

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.

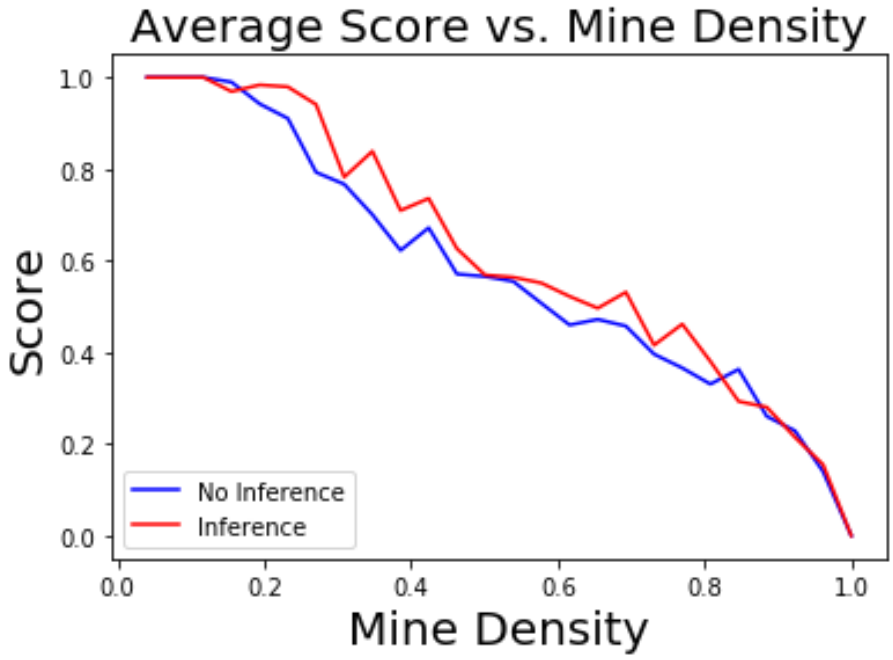

Figure 6

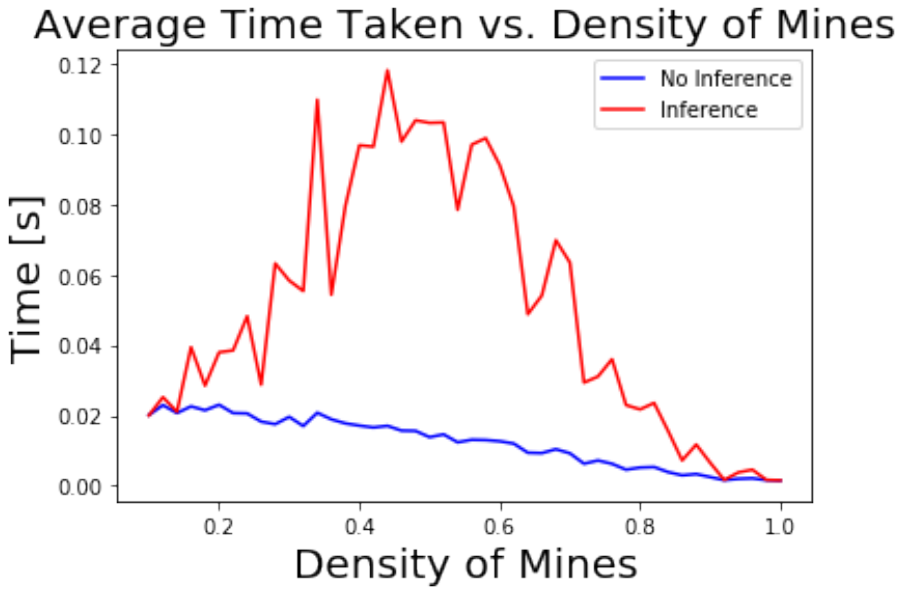

Figure 7

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.

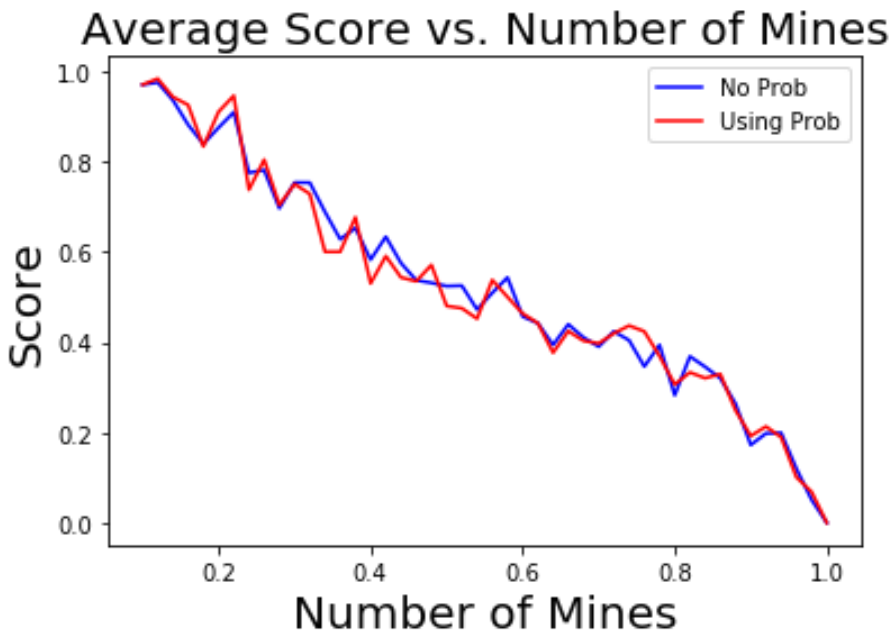

Figure 8

☺Member Contributions☺
everyone contributed equally

Minesweeper 2.5

Ilana Zane, William Bidle, Rakshaa Ravishankar

April 12, 2020

1 Computation

1.1 Methodology

We utilized the advanced agent from Project 2 and built upon it. The agent from Project 2 would go through any obvious moves first, and then try to make any inferences it could for the board. If nothing at all could be determined from this step, it then needed to access the probabilities of our 'active unknowns,' which are cells that have some information tied to them (i.e. an adjacent cell has been revealed). To get these probabilities, we utilized the reduced matrix that our inference agent generated for us and created a list of potential solutions. We obtained the potential solutions by taking the dot product of our matrix with vectors of 1's and 0's (indicating a variable being safe (0) or a mine (1)) and checked whether or not the result was equal to our answer vector, which contains revealed clues. From these solutions, we determined the probabilities by averaging how often a variable appeared as a mine (see Figure 5).

1.2 Reducing Variables

In general, if we have n variables per equation and each of these are either safe (0), or a mine (1), then our program will generate 2^n possible solutions that we need to check. For larger boards, we reach a very unreasonable number of solutions that the computer will have to simulate. If our board was even 10x10, the 100 cells would result in 100 variables per equation, leading to 2^{100} potential solutions that we need to check. However, this number can be drastically reduced. We only need to simulate potential solutions that include our 'active unknowns,' and not include any 'inactive unknowns.' In our situation, inactive unknowns correspond to any 0 columns in our reduced matrix, which tells us that no information has been revealed about that cell and its neighbors. This reduces our potential solutions and we end up simulating a number of solutions within a reasonable amount of time (see Figure 3).

1.3 Approximation

Even after this reduction, there were still scenarios where equations have a significant amount of variables. The result would be a very slow run time due to a large amount of computations. We found that any situation with more than 18 variables per equation led to our program taking a long time to compute possible solutions for just one step in the game. In order to solve this problem, we set a limit to when our program would calculate exact probabilities and when it would begin to estimate. If our equations exceeded 18 variables (i.e. greater than 2^{18} possible vectors to check), we would only simulate from a random selection of 2^{18} vectors in the total amount. For example, if we had 20 variables, we would randomly select 2^{18} of the 2^{20} total vectors to then continue and find the dot

product for possible solutions. This would yield approximate probabilities, however since our sample size is relatively large, we would end up with values that were close to the exact ones. The figure below shows an example where we simulate the exact probability vs. the approximate probability for a system of equations containing 20 variables (the X-axis is the variable number and the Y-axis is probability value). Even though we are only looking at 2^{18} out of the 2^{20} vectors in our approximation, the probability results seen are close to the exact answer. In some places we can see that the approximate is even the same value as the exact, so not only would our approximation help with space/time constraints, but it would also keep accuracy in the results.

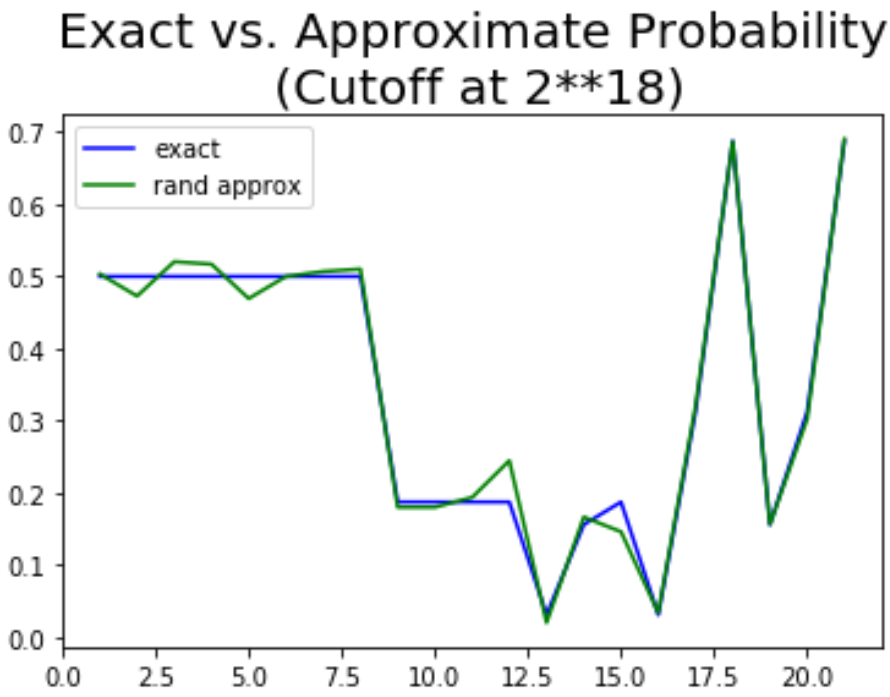

Figure 1

1.4 Example

1.4.1 Generating Solutions

We have created a step by step explanation as to how we calculated general probabilities using the example provided in the Project 2.5 Assignment PDF.

```
[2] [A] [1]
[B] [C] [D]
[E] [3] [F]
```

Figure 2

Based on the given variables, our inference agent creates a system of linear equations (our augmented matrix) and the associated answer vector that contains all clue values.

Immediately our inference agent determines that cell B is a mine and D is safe. The equations are created by looking at each uncovered cell and adding its clue value into our answer vector. In the assignment's example, the cell of clue 1 has remaining neighbors A and C (D has been reduced to be safe). The cell with clue 2 has remaining neighbors A and C as well, and since B is a mine, our clue value is reduced to 1. Therefore the two equations for cells 1 and 2 are the same so we reduced the redundant equation to just A+C=1. The clue value for 3 is reduced to 2 because B is a mine, so the resulting equation is C+E+F=2. The full linear equations are then generated into an augmented matrix, where active variables in the equation are represented by 1s and inactives by 0. As explained in the approximation section of our analysis, if there are any columns of our matrix that consist of only zeros, we exclude those columns from our matrix to prevent our program from analyzing cells that are already declared as safe/mine or are inactive unknowns. For our example columns 2 and 4 are removed to create a simpler matrix. The final reduced matrix is now in the form we need, with A+C=1 and C+E+F=2, without any extraneous variables.

```
For the system of equations:
A + 0B + C + 0D + 0E + 0F = 1
0A + 0B + C + 0D + E + F = 2

We represent this by:
[1 0 1 0 0 0] = [1]
[0 0 1 0 1 1] = [2]

And it can first be reduced to:
[1, 1, 0, 0] = [1]
[0, 1, 1, 1] = [2]

Since we had a zero column for the 2nd and 4th variables (i.e. B and D)
```

Figure 3

1.4.2 Generating Probability

For our six variable example above, if we were to include the inactive unknowns (zero columns), we would end up with 2^6 (64) potential vector solutions that we would need to check. However, if we eliminate these inactive unknowns from our equations, then we will reduce this total amount. Since B and D give us no helpful information (we already deduced what we could from them), we can safely look at only the remaining 4 variables A,C, E and F. This then leads to only 2^4 (16) potential solutions that we need to check. The generated solutions can be seen below.

```
The possible solutions for the reduced matrix are:
[[0 0 0 0]
 [0 0 0 1]
 [0 0 1 0]
 [0 0 1 1]
 [0 1 0 0]
 [0 1 0 1]
 [0 1 1 0]
 [0 1 1 1]
 [1 0 0 0]
 [1 0 0 1]
 [1 0 1 0]
 [1 0 1 1]
 [1 1 0 0]
 [1 1 0 1]
 [1 1 1 0]
 [1 1 1 1]]
```

Figure 4

We check if these solutions are valid by taking the dot product between them and our matrix. If the resulting vector is equal to our clue vector (in our case it would be [1, 2]), then we have a valid solution. Of the 16 possible vectors, only 3 yielded valid solutions to our system of equations as seen below. The final step would be to take the average value of each column (thus each variable), and that average value would tell us how likely that spot is to be a mine. As expected, we find that A has probability $\frac{1}{3}$ to be a mine, and the rest have probability $\frac{2}{3}$ to be a mine.

```
Now take the dot product of each potential solution with the system of equations
We find that the valid solutions are:
[0 1 0 1]
[0 1 1 0]
[1 0 1 1]

And the corresponding probabilities are:
[0.3333333333333333, 0.6666666666666666, 0.6666666666666666, 0.6666666666666666]
```

Figure 5

2 Basic Cost Agent

2.1 Data Analysis

The above plot shows the results of % Average Cost vs. Mine Density for our agent from Project 2 (in blue) and our agent that minimizes cost (in red). In general, as we increase the mine density of a board, we expect to see the percentage of mines stepped on increase (i.e. the cost increase) for any agent. Both agents can solve the game with basic logic when the mine density is low around 0.0 - 0.2 and both struggle to obtain enough information when the density is higher around 0.8 - 1.0. Therefore, we expect each of their performances to be similar in these regions and this is exactly seen in Figure 6. When the mine density is within the range of 0.2 - 0.8, we see that our cost agent ends up stepping on approximately 5% fewer mines than our agent from the previous project. Instead of picking randomly when we are stuck with no obvious moves, which is exactly what the Project 2 agent does, we use our simulated probabilities and make a decision based on these. Since we are now

accurately predicting what the probabilities of each active unknown cell is, we end up with more information about the board than if we just pick randomly. This results in a better chance to select a safe square than if we were to pick randomly.

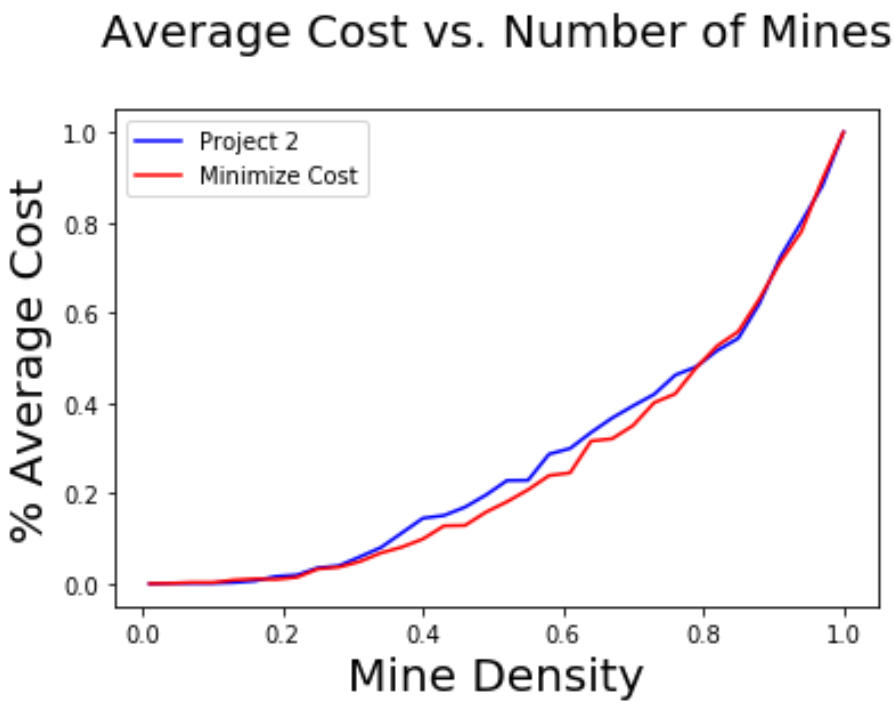

Figure 6

2.2 When and What to Pick

The methodology behind the cost agent was fairly simple, looking only at the cell with the lowest probability of being a mine and making our decision based on its probability value. In general, the probability that a completely random (i.e. inactive unknown) square is a mine is exactly $\frac{1}{2}$. Since we have no information about how many mines are left and we know the only possibilities are either mine or safe, then the $\frac{1}{2}$ probability is the only possibility for a completely random move. By utilizing this information, if the lowest probability for an active known square being a mine is less than $\frac{1}{2}$, the obvious choice for our cost agent is to pick that square since we have less of a chance of stepping on a mine. If the lowest probability is greater than $\frac{1}{2}$, then again, the obvious choice is to pick randomly from the inactive unknowns, since our chance of stepping on a mine is smaller. In the event that we have a tie, that is the lowest probability that an active unknown is a mine and the probability an inactive unknown is a mine, are both $\frac{1}{2}$, then we must break the tie. We decided that it would be best to reveal the active unknown square, as we would end up revealing more information about the board. Any information revealed around an active unknown square would potentially lead to more information about other surrounding active unknown squares and we would potentially be able to open up more relevant information for the game. Choosing an inactive unknown would only add the list of active unknowns and not progress our game in a helpful way.

3 Basic Risk Agent

3.1 Data Analysis

The agent for minimizing risk shows an improvement for a mine density greater than 0.2 and up until 0.8, which is similar to what we saw in the cost agent. In this region, the minimizing risk agent tends to make definite moves about 5% more often than the Project 2 agent does. This occurs because this agent tries to minimize the number of uncertain moves (risk) made in the game by utilizing information it gains from simulating moves and looking ahead.

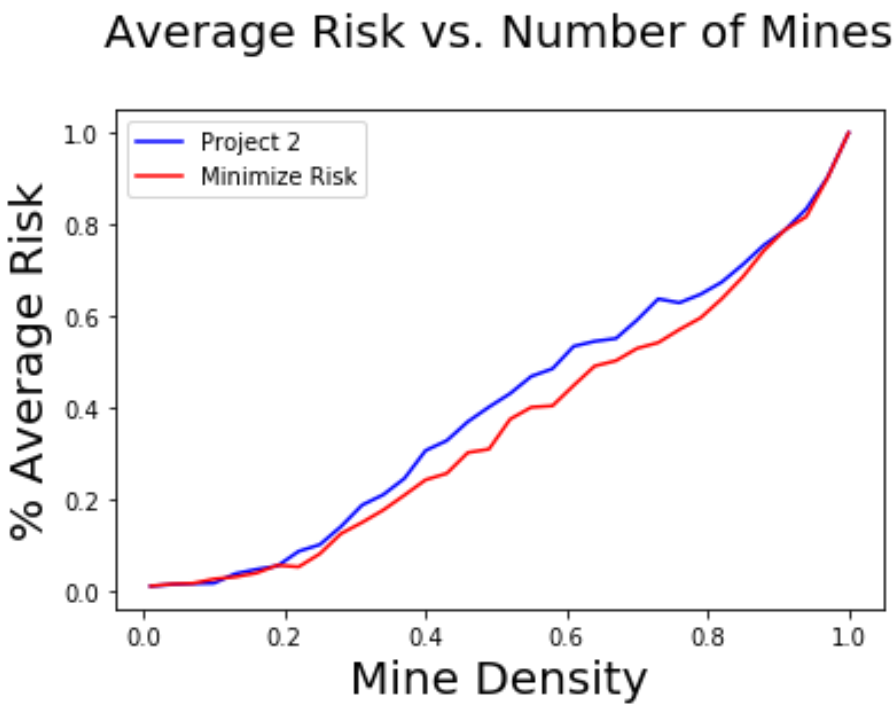

Figure 7

3.2 Methodology

Unlike the cost agent, which simply looks at the lowest probability of an active unknown being a mine, the risk agent has to look a step ahead. The risk agent simulates the outcome of an active unknown by marking it first as a mine and then as a safe, determining the resulting number of definite moves we uncover in each scenario. This was achieved by creating a temporary knowledge base with the current active unknowns and checking the possible neighbors of these cells. The agent then followed the basic rules set forth in the previous project to determine any obvious safe or mine cells and kept track of them in the temporary knowledge base. The results get stored into the variables R and S, where R is the number of squares the agent could work out if the possible move were a mine and S is the number of squares the agent could work out if the possible move were safe. We then insert both values into Equation 1 to find the expected number of squares that can be worked out when opening the current possible move.

$$qR + (1 - q)S \tag{1}$$

Where q is the probability of the possible move being a mine. This equation then tells

us the expected number of moves we will be able to immediately determine for a given cell. After simulating this value for each of our active unknowns, we then choose the cell that has the highest expected value to be revealed. The calculation of risk and simulation of sample possible moves is only necessary if nothing can be determined by the knowledge base after inferences are made, the safe and mine fringes have been traversed and emptied and moves that minimize cost are made. The "improved" decision making is in place of making a random move, since it chooses a cell with a greater probability of the rest of the board being solved, taking into account whether or not the cell could be a mine.

4 Bonus

We had expected that each agent would be better at minimizing its respective quantity, as that was the main priority for each of them. Surprisingly, we found that the risk minimizing agent is better when it comes to minimizing both cost and risk. Even though there isn't a significant difference between the two, the risk agent outperforms the cost agent by a very small percentage when minimizing cost. In hindsight, we think this result makes sense because the risk agent looks a step ahead in the game, whereas the cost agent only uses information from the immediate state of the board. Additionally, the minimizing risk agent works to minimize the number of uncertain moves and perhaps hits less mines overall for this reason.

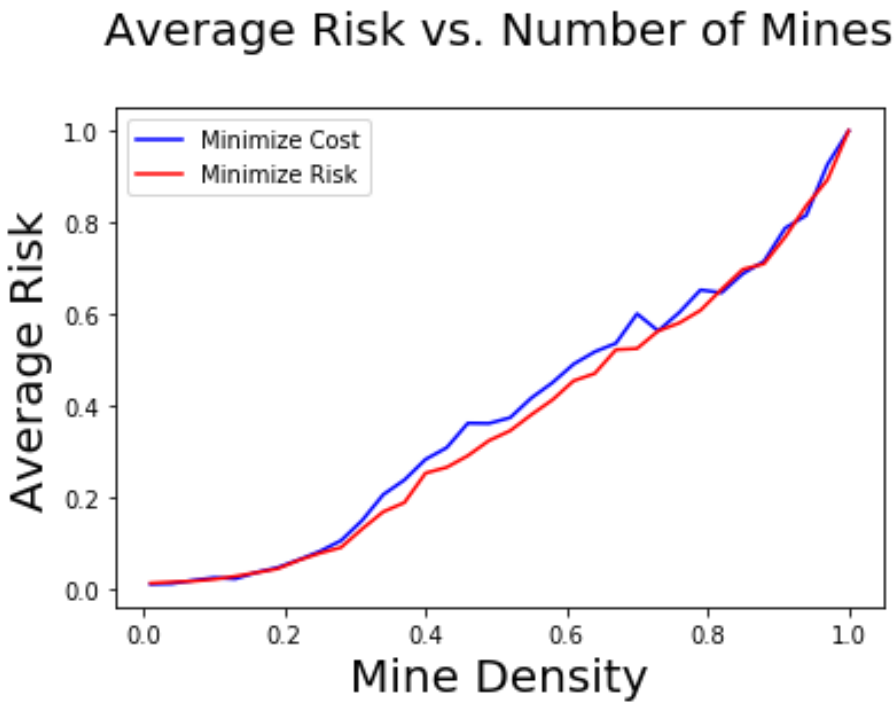

Figure 8

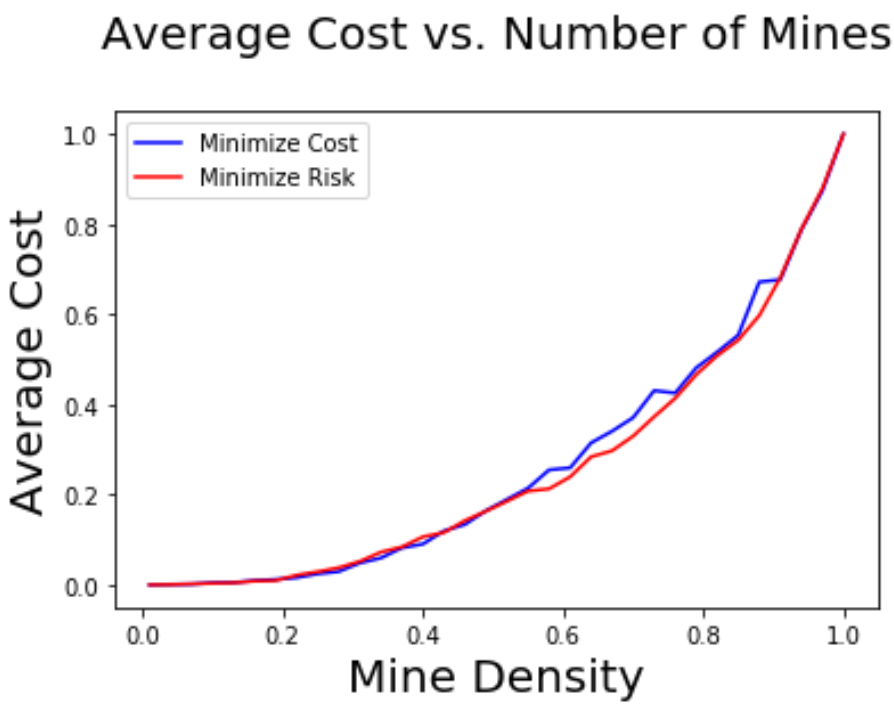

Figure 9

5 Improved Cost Agent

5.1 Basic/Improved Cost

For our improved agent we decided to build upon the basic cost agent that we previously created. As explained before, our basic cost agent currently assigns each active unknown a probability that it is a mine based on information we gathered from our inference step. It will then analyze the cell with the lowest probability and make a decision based off of the probability value. Our improved cost agent takes this same logic a step further by utilizing the concept of conditional probability in order to look a step further into the game. Therefore, when faced with similar scenarios, the two agents will tend to behave differently and make different decisions, even if the overall outcome is the same.

5.2 Simulation

In order to simulate conditional probabilities, we first needed to analyze all of the valid solutions that were previously generated. For example, if our system of linear equations contained variables [A,B,C] we would simulate clicking on A and find the probabilities that B and C are mines given A. We would continue this process for ${}^{n}P_{2}$ permutations, where n is the total number of variables in our equation, thus calculating all conditional probabilities are calculated for each variable. If we were to calculate the probability of B being a mine given A, we would use the following equation:

$$P(B\ mine|A) = P(A\ mine) * P(B\ mine|A\ mine) + P(A\ safe) * P(B\ mine|A\ safe) \quad (2)$$

With the list of valid solutions we have generated we can find the probability of A being a mine and the probability of A being safe. We then look for solutions where A

is strictly a mine— from those solutions we determine the probability of B being a mine. Solutions where A is strictly a safe are found and we determine the probability of B being a mine based on those solutions. All four of these probabilities are combined to find the total conditional probability for each variable being a mine given each of the remaining variables. Each variable will have n -1 possible conditional probabilities associated with it, and from this point we select the smallest probabilities from this set. Once the smallest probability has been selected for each variable we compare probabilities across our variables and select the variable that is associated with the smallest conditional probability— this is the next cell to be uncovered. However, if there is more than one variable that shares the smallest conditional probability we have to resolve the tie. We do this by replacing those tied conditional probabilities with their original probabilities (i.e. the probabilities that would have been used by our basic cost agent).

5.3 Comparison

Figure 10 shows that our improved cost performs slightly better than our basic cost agent. This occurs when the density is within 0.4 and 0.9, because outside of this region both methods can solve the game with basic strategies. Even though both agents attempt to minimize the cost of the next move, the advanced agent ultimately outperforms the basic one. By looking ahead and utilizing Equation 2 to find conditional probabilities of moves, the advanced agent ultimately steps on fewer mines.

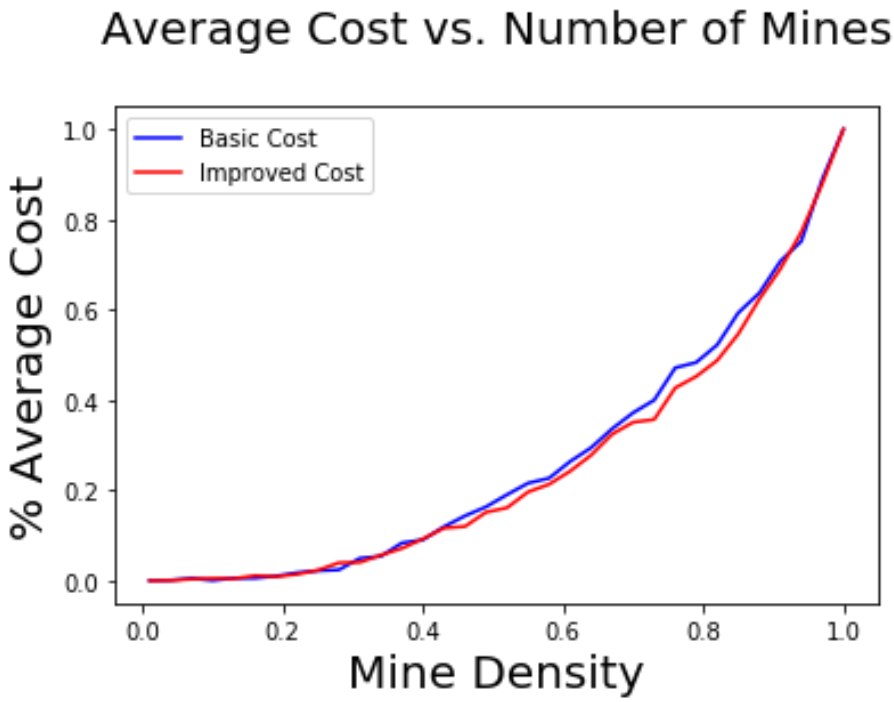

Figure 10

5.4 Things to do Differently

We could definitely make our improved cost agent better. Even though it performs better than the basic cost agent, it only simulates one step into the future. One of the biggest issues we had while doing this project was handling the vast amount of potential solutions that would occasionally arise for some boards. As we explained in Question 1, there were

some areas where we could simplify and make approximations to speed up our calculations. This ended up working well for the basic cost and risk agents, however, when we moved to the advanced cost agent similar problems arose. On top of the large amount of immediate probabilities we needed to simulate for each active unknown, we then also had to calculate every permutation of conditional probabilities. Since we didn't make any approximations for calculating conditional probabilities, we were only able to simulate one step into the future.

Member Contributions: All members contributed equally