CS 440

PROFESSOR WES COWAN

Minesweeper

MARCH 22, 2021

NABHANYA NEB (NN291)

ARPITA RAY (AR1516)

I. Minesweeper Introduction

Our code uses a series of techniques to create an AI version of the common game known as Minesweeper. The objective of Minesweeper is to use educated guesses in order to reveal which cells in a square grid are mines, without actually blowing the mines up. This grid can be of any dimension, and can have any number of mines hidden, as long as it is less than the total number of cells. Our project is composed of four main components– the states, the game itself, the knowledge base, and updating information as the agent goes through the board. Each of these components is used in the basic agent program, the advanced agent program, and the extra credit tasks for both as well. They are modified accordingly. When a cell is revealed, it displays a clue value if it is safe, a 'B' if it was blown up, or an empty square if a mine was discovered.

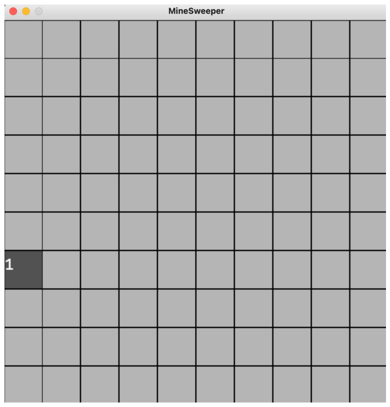

Fig. 1: Starting Position for Basic Agent

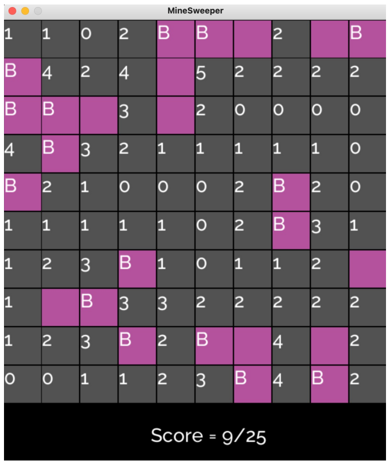

Fig. 2: Final Result for Basic Agent

II. Representation of the States

To start, we define a state object. This object is used to make up the game board. Each cell is a state, which is initialized with a row, column, and value attribute. The row and column represent its location on the board, and the value is used to hold whether it has been visited, and whether it is a mine or a safe spot. It keeps track of inferred relationships between this state and its neighbors, This object also holds the number of total neighbors, how many are mines, how many are safe, and whether or not they have been revealed. We also added a function to return a list of all the neighboring states. A neighboring state can either be above, below, to the left of, to the right of, or diagonal from the given state.

III. Representation of the Game

Next, we define a game object. This object is used to run the fundamentals of the game. The game holds the dimension, total number of mines on the board, and a 2D array of states which is referred to as the grid. We also wrote a function to randomly place n mines across the board. This function returns a list of n random tuples (r, c), taken from the list of all tuples, with no repeats.

IV. Representation of Knowledge

Representation of the knowledge base is done in two different ways for the basic and advanced agents. The basic agent uses a priority queue of tuples that hold cell locations. We used a Python heap queue for this. The priority is defined by the probability that the cell at that location is a mine. The lower the probability, the more likely that cell will be visited by the agent.

We used constraint programming to create the advanced agent. Our knowledge base here is a 2D Array in the form of an augmented matrix It has $dim * dim + 1$ columns. The amount of rows depends on how may equations are needed to solve the game board each round. Each column represents a cell, and each row represents an equation using those cells. The last column is all clue values. All of the values are either 0 or 1, depending on whether or not they show up in that row's equation.

The last column represents the clue value for that equation. For example, say we have a $3x4$ matrix as follows to represent cells a, b, c:

$$\begin{bmatrix} 1 & 0 & 1 & 2 \\ 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 \end{bmatrix}$$

This means $a + c = 2$, $a + b = 1$, and $b = 0$. The AI would solve that $a = 1$, $b = 0$, and $c = 1$. This means that b is not a mine, while a and c are. The steps taking to achieve this matrix and these solutions are discussed in a later section.

V. Updating Information

After that, we define a function to update all the lists, arrays, and priority queues used in our AI. This function helps keep track of which cells have been revealed, how many are mines, how many are safe spots, and the explosions. It also updates the all the values and the knowledge base that the AI uses to make decisions in later steps.

VI. Inference

Whenever, the basic agent visits a state, it is marked as revealed, and a clue value is revealed. This value represents how many of its neighbors are mines. To do this, we first check if the current state is safe. If so, we count the number of mines surrounding it, and then update all of the variables for total neighbors, total mines, how many of those are revealed, total safes, and how many of those are revealed, in that state accordingly. If the total number of mines minus the number of revealed mines is the number of hidden neighbors, every hidden neighbor is a mine, so we reveal and push to the knowledge base accordingly. If the total number of safe neighbors minus the number of revealed safe neighbors is the number of hidden neighbors, every hidden neighbor is safe, so we reveal and push to the knowledge base accordingly. If the current spot is a mine, we blow it up.

The advanced agent, as aforementioned, uses a matrix with $dim * dim + 1$ columns and at-most $dim * dim$ rows. Every state on the game board is represented by a column in this matrix.

The method for discovering mines and safe spots is initially the same as the basic agent, but is further extended. This is described in the next paragraph. Before doing so, we update the matrix as follows. Every time the agent visits a safe state and new information is discovered, a corresponding equation is added to the matrix. The last value for this new row is the clue value that is revealed. For the rest of the row, the neighbors of the current state are represented with a 1 and everything else in the row are marked as 0 for this equation. If any safe spots are discovered, we mark everything in their column as 0. Every time the agent blows up or discovers mines, we mark everything in their column as 0. If any values in that column were previously a 1, the clue value for its row is subtracted by 1.

We perform these operations, based on the information we collect from the basic agent, at every step the advanced agent takes. Next, we check to see if we discovered any new safe spots or mines. We do so by iterating through the rows and checking if the clue value is 0. If this is the case, we find all the 1s and declare the corresponding states as safe, then update our knowledge. If the clue value is equal to the total number of 1s in the row, then every state in that row with a value of 1 is declared a mine. This advanced agent discovers new information that the basic agent cannot. The basic agent only uses local search while the advanced agent analyzes more data at once.

Our clever acronym for the basic agent is K.I.P. which stands for Knowledge is Power. Our clever acronym for the advanced agent is S.I.T.A. which stands for Subtraction is the Answer.

VII. Decisions

After all the information has been updated, the agent must decide where to go next. For the basic agent, this is an easy decision. It just pops off the next item off the priority queue because that is an inferred safe spot. The advanced agent does the same, since the priority queue was marked according to the additional matrix of equations. If the base is empty, it chooses a random location to go to. It does this by choosing a random tuple (r, c) of all covered spots.

VIII. Performance: Walk-through

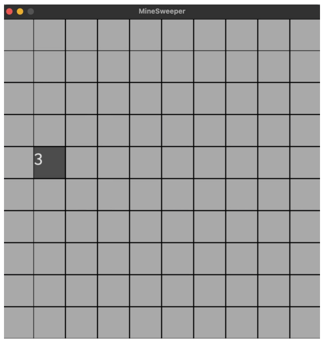

Fig. 3: Start of size 10 with 20 mines

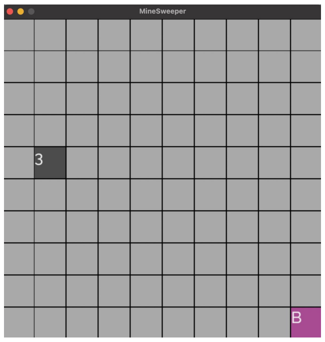

Fig. 4: First random blown up mine

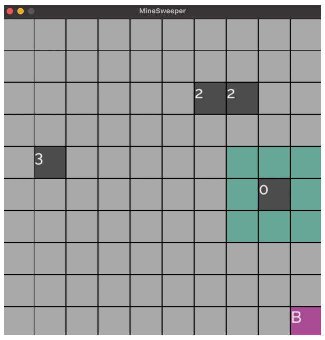

Fig. 5: 0 clue value = 8 safe spots uncovered

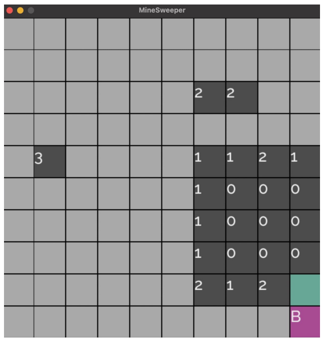

Fig. 6: Information used to make Advanced Inference

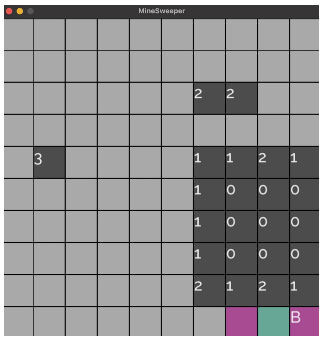

Fig. 7: Result of Advanced Inference at (9,7)

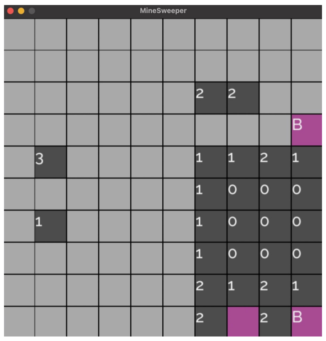

Fig. 8: Second random blown up mine

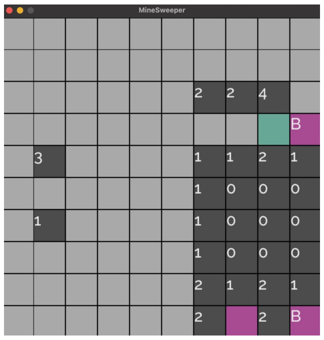

Fig. 9: Information used to make Basic Inference

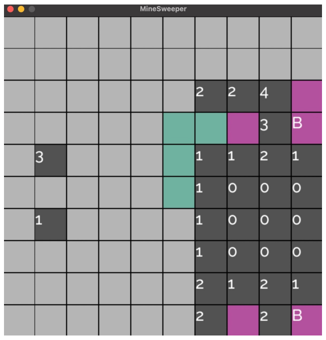

Fig. 10: Result of Basic Inference at (3,7) (2,9)

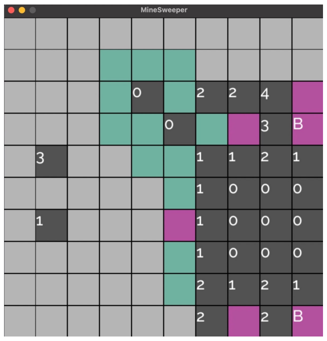

Fig. 11: Result of next Advanced Inference at (6,5)

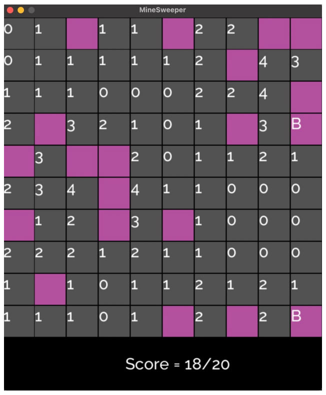

Fig. 12: End: Score 18/20

Step by step images of the advanced agent with a dimension of 10 and 20 random mines are shown above. Figure 3 shows the random location where the agent starts. Figure 4 shows the next step, chosen randomly, which unfortunately happens to be a blown up mine. Figure 5 shows a later step in which the agent found a location with a clue value of 0. Since all the neighbors are safe, the agent proceeds with visiting each of the neighbors to gather more information. Figure 6 shows the board's status before the agent has to make its first advanced inference. $(8,8)$ has a clue value of 2, and $(8,9)$ is a safe spot. Thus, $(9,7)$ must be a mine, since $(9,9)$ is also a mine and the rest of the neighbors have been visited and marked as safe. The basic agent would not be able to do this because it does not keep track of that many states and all of their neighbors at once. Figure 7 shows the result of this. Figure 8 shows a random spot chosen, which happens to be a mine. Figure 9 shows the information used to make a basic-level inference, and it uncovers that $(3,7)$ and $(2,9)$ are guaranteed mines, which is shown in Figure 10. Figure 11 shows the result of the next advanced inference, done similarly to the method explained for Figure 7. Finally, we fast-track to the end. The advanced agent discovered 18 of the 20 mines. 12 of these were done with the basic-method, and 6 were done with the advanced-method. The only two that were blown up were chosen randomly.

There aren't any points where our advanced agent program makes a decision that we don't agree with. The only mines that it blows up are locations that are randomly selected when no further inferences can be made. For the basic agent, there are be better decisions that could be made but these issues are solved by our advanced agent. Our program does not surprise us anywhere, and it actually performs better than we would've on our own. It scored well and performs exactly how we intended it to.

IX. Performance: Plots

Plots for the different agents are shown below.

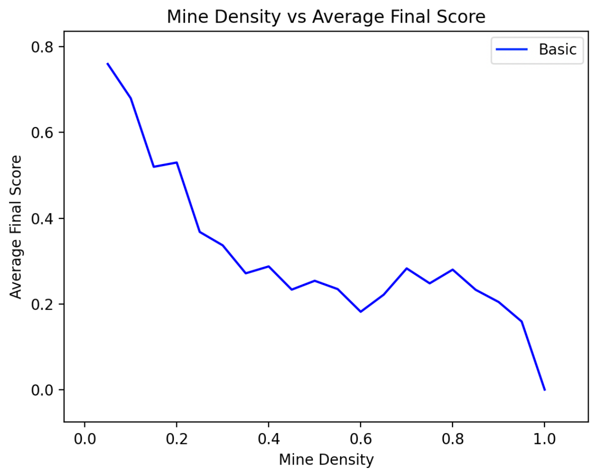

Fig. 13: Basic agent's success rate

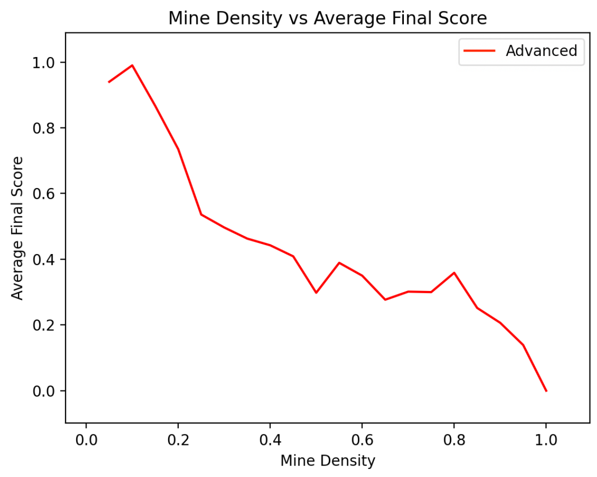

Fig. 14: Advanced agent's success rate

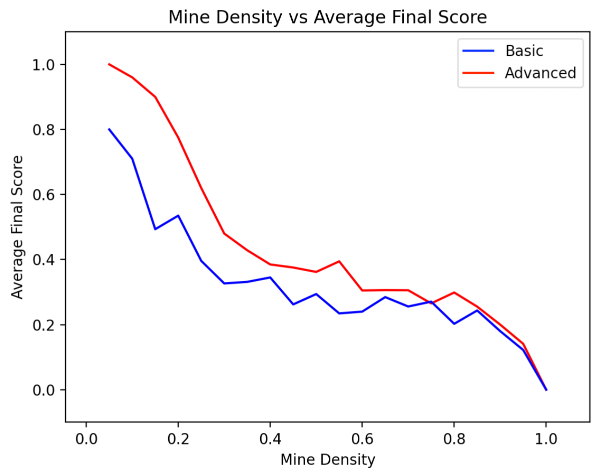

Fig. 15: Success Rates Compared

These graphs agree with our intuition. Minesweeper gets hard when the mine density gets high, approximately around 0.75. The advanced agent is consistently performing better than the basic agent. The advanced algorithm performs significantly better until the density reaches 0.75. At this point, a vast majority of the board is a mine, so it makes it difficult to have a fully comprehensive knowledge base for either of the agents, and the success rates start to get closer together.

X. Efficiency

In every program, the coders will run into time and space constraints. There are some problem-constraints that have to be worked around, but there are also some implementation-constraints that we would like to improve on.

The first time constraint is that with every step, the agent must update its knowledge base. This one is problem specific. In order to make the best decisions possible, the knowledge base must be as up to date as possible. Using old knowledge would create a lower success rate because knowledge can grow and change. However, doing this for every state that the agent visits does take time, and it would not be possible to go through the entire knowledge base with every single step, especially for bigger dimensions. Our computers would probably heat up and crash. The next time constraint is going through each neighbor of each cell and making sure the value for all the variables are updated. This is an implementation-based constraint. We would like to improve on this by finding a way to automatically update multiple at once without missing any. Additionally, if a neighbor is a neighbor of the current cell, finding some way to keep track of this and update accordingly would help as well.

The first space constraint is the size of the advanced agent's matrix. This is a problem-constraint, assuming using a matrix is part of the problem, because we must have a minimum of $dim * dim + 1$ columns, because we need one column to represent each cell. The next space constraint is using multiple data structures to hold all the information. This one is implementation specific. We have different lists for each type of cell, when it would be better to hold all types of cells in one concise and informative data structure.

XI. Running the Game

Finally, we have a function to run the game. This function initializes the UI and game, holds the agent's current location, and chooses a random starting spot. Then, it runs a while loop until all the cells have been revealed. This is essentially our main function. It chooses the next intelligent cell to reveal, removes that from the knowledge base, and keeps track of the score. The score is the total number of mines that have been discovered, over the total number of mines on the board.

XII. Extra Credit

Minesweeper performs better if the agent does not choose random states when the queue is empty, and instead uses logic. To implement this, we use probabilities to determine the likeliness that the remaining hidden states are mines. For example, if one of the hidden state's neighbors has a clue value of 5, then the hidden state is marked with a probability of $1/5$. If another neighbor has a clue value of 4, then the probability would be $1/5$ divided by $1/4$, which is $4/5$. Therefore, the chance of it being a mine is increased, and the agent is less likely to visit here. This is done for all the hidden states. The plots for this implementation are shown below.

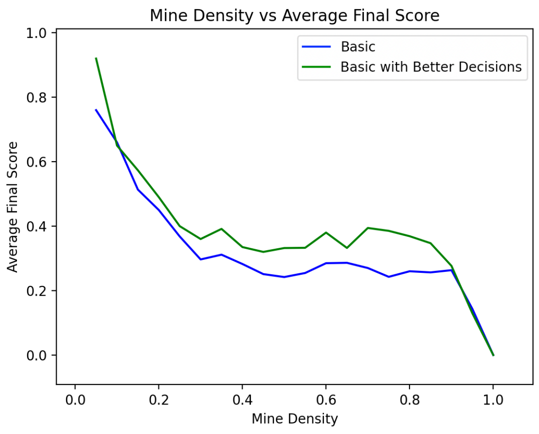

Fig. 16: Plot for basic agent with and without better decisions

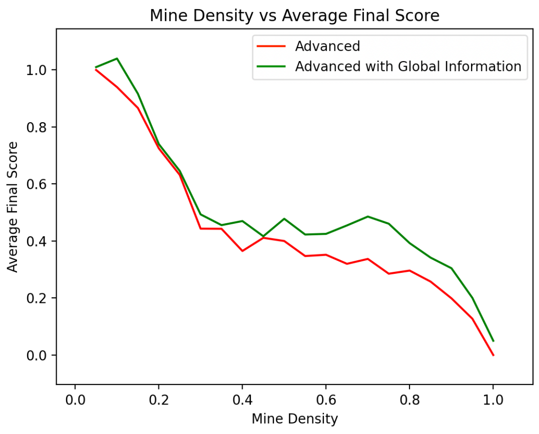

Fig. 17: Plot for advanced agent with and without better decisions

For the other part of the Extra Credit, we build off of the decision making mentioned above. The probabilities factor in the total mines that have yet to be discovered. It uses the method from above, and then if there are remaining spots that do not have any revealed neighbors, the probability is set to $1/\text{mines remaining}$. This allows for a further improved calculation. The plots for this implementation are shown below. These plots have the highest success rates.

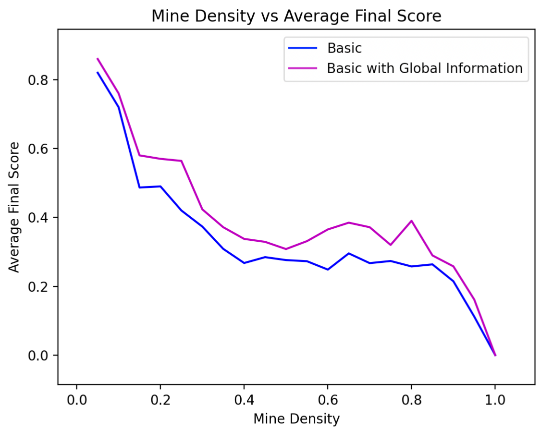

Fig. 18: Plot for basic agent with and without global information

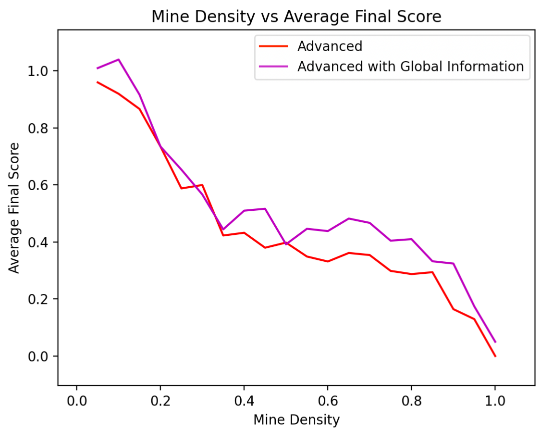

Fig. 19: Plot for advanced agent with and without global information

Acknowledgements

Nabhanya Neb (nn291):
We did both the code and the report together using Zoom. Because of this, we did not have to split anything up and we both contributed equally while working on call. We alternated who was coding/typing to make it fair.

Arpita Ray (ar1516):
The work was evenly divided because we did all of the code on screen-share. We took turns on who would be the person typing each time. For the report, we used Overleaf and did the write up the same way. We worked together on it and alternated who wrote.

Honor Statement:
We have read and abided by the rules of the project, we have not used anyone else's work for the project, and we did all of the work by ourselves.