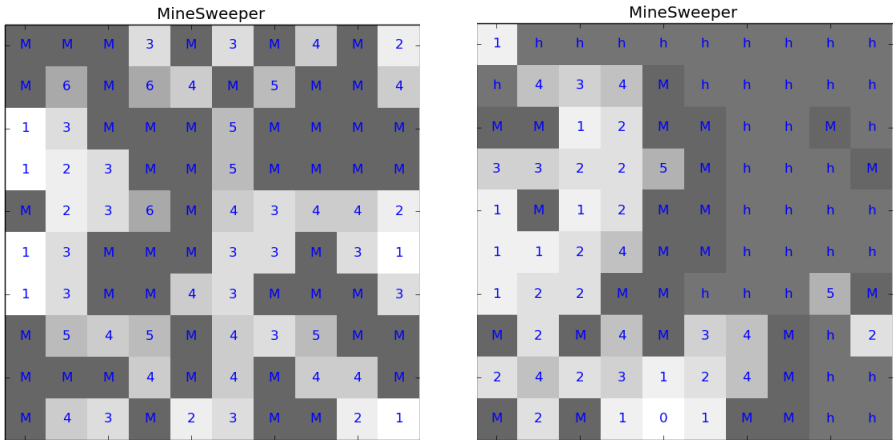
CS440 - Spring 2021 - Project 2

Gal Zandani (gz113) and Vaishnavi Manthena (vm504)

March 22, 2021

Representation

The board is represented by a 2-dimensional numpy array with a given dimension and a randomly generated num from 1 to the number of locations in the game for the number of mines in the game. Each location is specified as either 'M' for a mine, an integer from 0 to 8 for a safe location with that many number of neighbor mines, or for the game board 'h' for hidden locations on the board.



The knowledge board is represented by a list of tuples, for each tuple there is a (key, value) combination so that the key is a list of certain hidden locations on the board and the value is the updated clue number of mines in that list of locations. We get this number from the clue number at a certain point, at first, and update that as we get more information about identified safe locations and mine locations. Whenever we identify a location we also reveal it in the game currently being played. So, the entire information we know is located in the knowledge base and the game currently being played. We represent inferred relationships between cells by:
Say M(i,j) = 1, if there is a mine at (i,j), else it is 0

$$S(i,j) => \sum M(m,n)\ for\ all\ (m,n)\ which\ are\ neighbors\ of\ (i,j)\ = C(i,j)$$

S(i,j) is a revealed safe cell. C(i,j) is the clue number at this safe cell.

Inference

When we collect a new clue in the game, we add that information to our knowledge base in the following way.

If we collect information about a safe location on the board, we add a new tuple to the knowledge base, like the one described above, with the key as a list of the hidden neighbors of the known safe location and the value as the clue number given from the known safe location subtracted by the number of its mine neighbors known in the game so far. Then, we check for certain simplifications we can make. So, we remove found safe location from all lists of locations (there is one list/key per each rule) and not change the corresponding updated clue numbers or values (this is because a safe location has a value of zero and does not contribute to the mine number).

If we collect information about a mine location on the board, we only update the current data in the knowledge base. We check for certain simplifications we can make. So, we remove the found mine from all lists of locations and subtract 1 from the updated clue number of only the rules it appears at.

Additionally, for the two cases, we check whether there is a rule such that the length of the list of locations on the board equals to the updated clue number. If there is such a case, we can conclude all of the locations in the list are mines. Also, if there is any rule whose value (updated clue number) changed to zero, we conclude that all locations in the corresponding list are safe.

When we get a clue about a safe cell or a mine, as mentioned before, our process of updating the knowledge base involves going through each rules and substituting our new knowledge. In this process we deduce information about cells that are safe or cells that are mines only if this can explicitly be inferred from each individual rule. However, we do not deduce everything possible as we do not analyze different rules at once when we update our KB.

Example of what we infer when we update KB:
A+B+C = 2
If we add 'A=0' (A is safe), then we deduce that B and C are mines.

Another example of what we infer when we update KB:
A+B+C = 1
If we add 'A=1' (A is a mine), then we deduce that B and C are safe.

Example of what we can't infer when we update KB:
A+B+C = 1

Now say we add: A+C+D = 2 to the KB.
We cannot deduce that D-B = 1 and therefore D = 1 and B = 0.

Note: however we do try to combine all information in the KB in a later step when we are trying to choose a cell to explore in the advanced agent. The particular function we use to improve our inference later on is 'queryKB'.

How we might improve it: solving systems of linear equations by using numpy arrays and mathematical functions provided by python.

Decisions

First our implementation would use any knowledge that was inferred from a previous update to the knowledge base. This information is basically a list of cells that are identified as safe and a list of cells that were identified as mines. Note that these cells have not been revealed yet, only identified.

So, first we explore all the identified safe cells in the game and update KB accordingly and also gather simple info that can be deduced from the KB, with these updates, for next iterations.

Then, we update the KB based on all the safely identified mines accordingly and also gather simple info that can be deduced from the KB, with these updates, for next iterations.

However, if we don't have such clues we would iterate through all the current hidden cells which are involved in the rules in the KB. For each hidden cell we do the following:

First, query the KB by assuming the cell is a mine. If the KB is not satisfiable then we know that the cell is safe. On the other hand if the KB was satisfiable, we query the KB by assuming that the cell is safe. Now, if the KB is not satisfiable then we know that the cell is a mine. On the other hand if the KB was satisfiable in both cases (when we assume the cell is a mine and when we assume its safe), then we cannot conclude anything from this cell.

In case we loop through all such hidden cells and cannot conclude anything about them, we randomly pick a hidden cell to explore next.

The way we query the knowledge base is that we create a copy of the current knowledge base and a copy of the current state of the game.
Then, if we assumed the given location is a mine, we mark it as a mine ('M') in the copied game and update the copied knowledge base accordingly. If we assumed the current location is a safe location, we mark it as safe ('C') in the copied game and update the copied knowledge base accordingly.
The conclusions we get from those updates of the copied knowledge base, we update the copied game with, in order to have a better understanding of the current state of the game which helps us make more better conclusions. Specifically, according to those updates, if we can conclude some of the basic safe and mine cells we update these values in the duplicate game to prevent extra assignments.
So, we construct a tree of assignments of the hidden locations in the copied game so we can analyze certain possibilities for their values. We analyzed

those assignments through checking if they are satisfiable or contradicts with the current knowledge base so we can draw some conclusions. The overall idea of our 'queryKB' is to specify wether a particular assumption will lead to a contradiction or not using the constraint satisfaction problem and a DFS traversal of the variable assignment tree. Here we try to choose hidden Locations currently in the knowledge base (these are the hidden locations which may violate constraints). Constraints are the rules in the knowledge base in compliance to the current state of the game. The domain of all values is either 0 (assigning it as safe) or 1 (assigning it as mine). We use the 'isSatisfiable' function to plug in variable assignments into the knowledge base and look for contradictions.

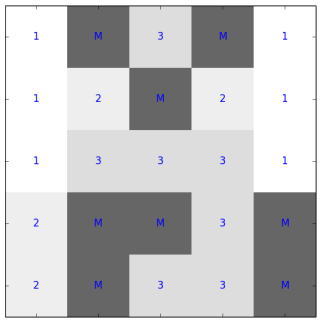
Performance 1

We generated a game with a mine density of 0.3 in a 5x5 board. The final score of this game is 0.75.
List of steps which reveal random locations: 1-4, 8, 11, 14
List of steps which reveal identified safe cells: 5, 6, 7, 10, 12, 16, 18, 19, 20, 21, 22, 24
List of steps which reveal an identified mine: 9, 13, 15, 16, 17, 23 The program uncovers first the safe locations and after that the mine locations, so there can be a step in the game that the knowledge base confirmed a certain location should be a mine and uncover it in the game only after it has finished uncovering all of the known safe locations.

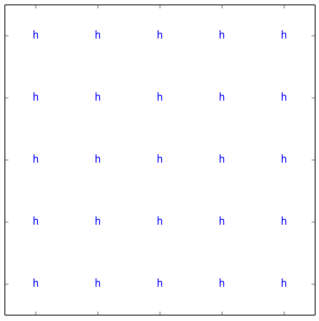
An example of a step where the program makes a decision that we do not agree with is in steps 10-13. When it reveals the mine in step 11, we must know that there is a mine located where it is revealed in step 13. But, before uncovering that mine, it chooses to uncover the safe location in step 12. The program is able to make that decision since the knowledge base might conclude there is a mine in a specific location but decides to reveal the safe location first. The goal of the game is actually safely identifying the mines on the board and revealing them.

An example of the steps where the program made a decision that surprised us is step 8. It chooses a random cell to reveal. However, it chooses a cell that is near safe cells that have a reasonable clue number. Amongst all the other hidden cells available for exploration it chooses a cell next to safe cells with high clue numbers. This was a little surprising since many of the other hidden cells may have resulted in a exploring a safe cell and therefore gaining more information about the environment. The algorithm works this way since it chooses a random safe cell. This problem is fixed to some extent in the better decisions extra credit.

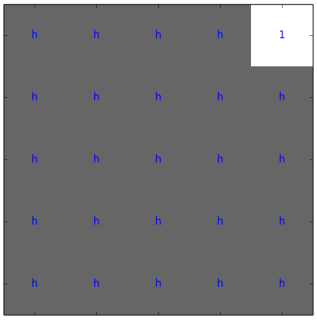
Solution



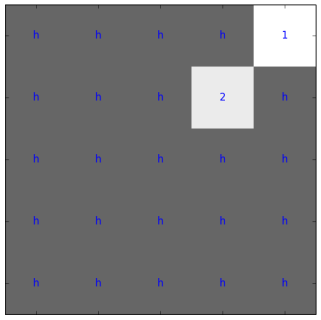
Initial Game



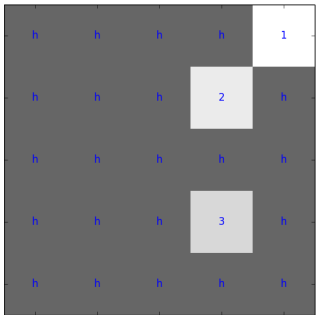
Step 1



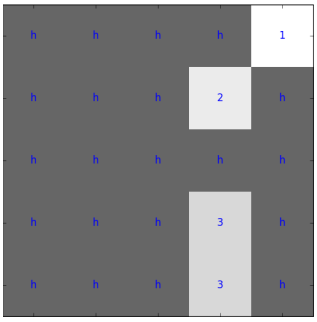
Step 2



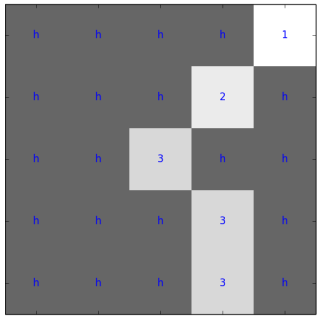
Step 3



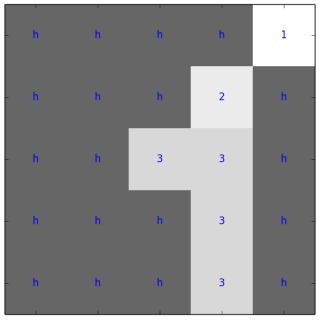
Step 4



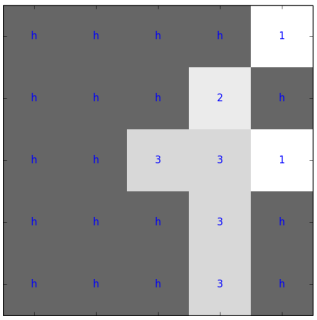
Step 5



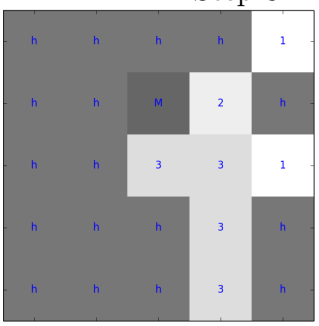
Step 6



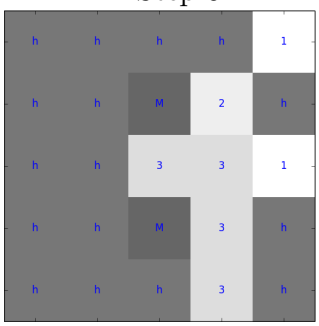
Step 7



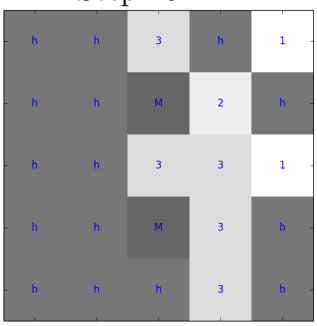
Step 8



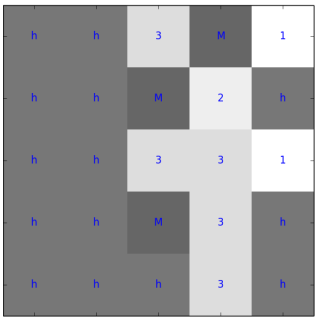
Step 9



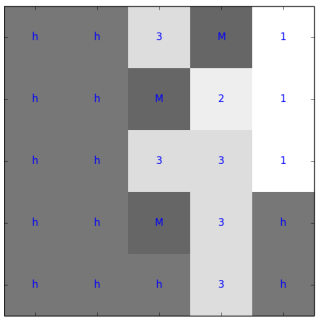
Step 10



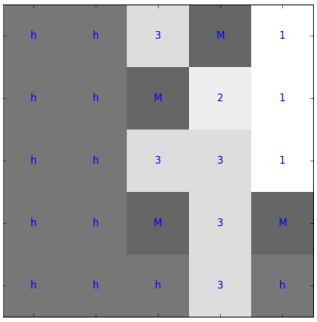
Step 11



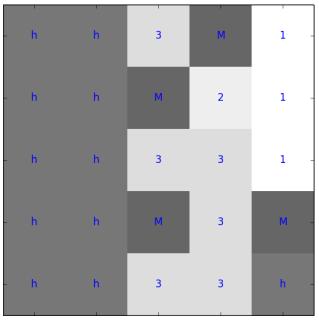
Step 12



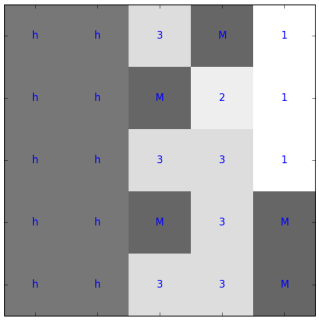
Step 13



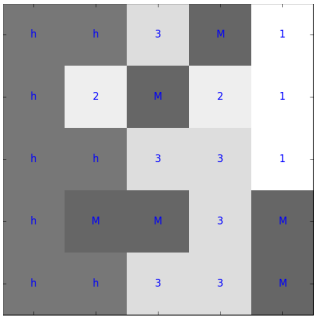
Step 14



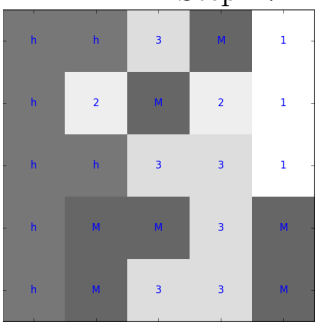
Step 15



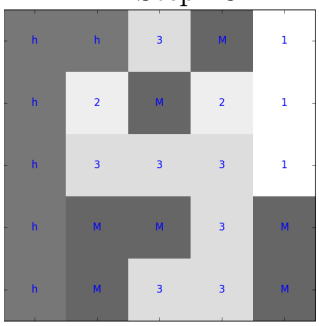
Step 16



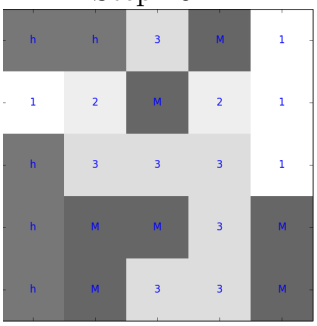
Step 17



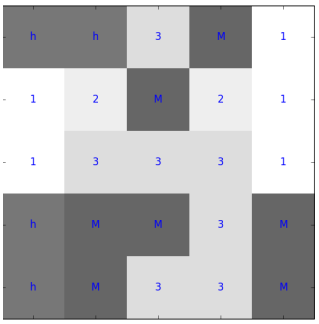
Step 18



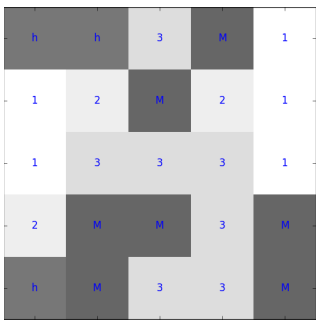
Step 19



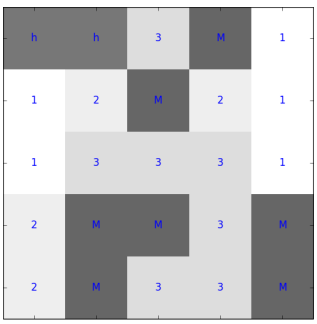
Step 20



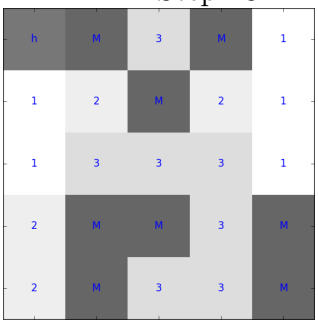
Step 21



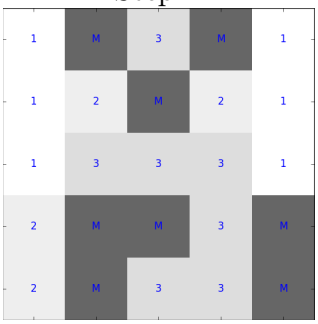
Step 22



Step 23

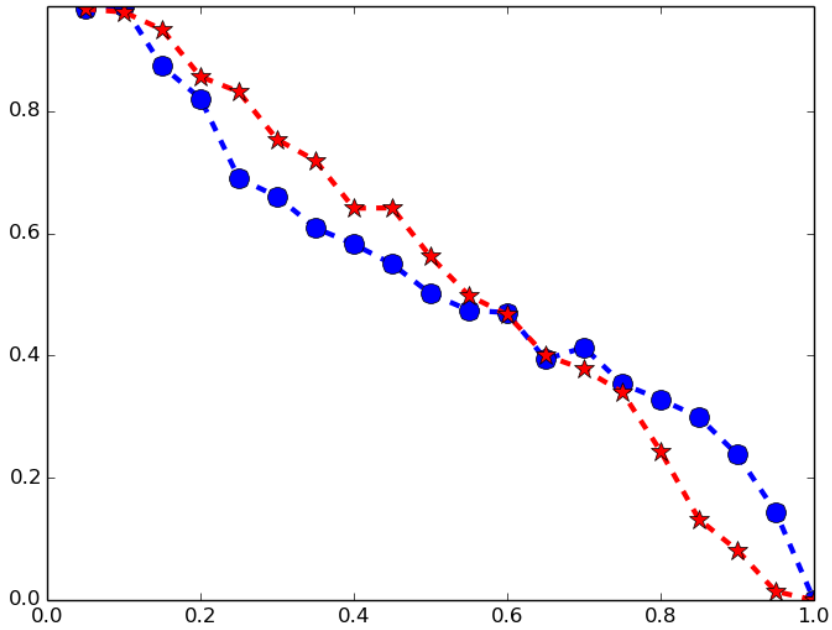


Step 24



Performance 2

Figure 1: Mine Density vs Average Final Score for Basic Agent and Advanced Agent



Index: Blue- basic agent and red- advanced agent. X-axis: Mine Density in the game, $0.05 \leq q \leq 1.0$. Y-axis: Average Final Score.

Our improved agent performs better than the basic agent between mine densities 0 and 0.6. At very low densities it behaves similar to basic agent. After density 0.65, it either performs the same or less well than the basic agent. Although at greater mine densities the average score is still poor due to the fact that with many mines the agent will be obliged to choose random positions, the basic agent performs comparatively better, because with greater mine densities it might be easier to directly infer mine and safe cells through simple techniques. This is because when there are many mines, it would be easier to come up with conclusions about mines through straightforward inferences. Our advanced agent, on the other hand, spends time querying the knowledge base and maintains data structures like all the safe cells identified (but not yet revealed) and all the mine cells identified

(but not yet revealed). The implementation issues of maintaining these data structures, like when to reveal the cells in these structures, or do we reveal the mines first or the safe cells, and more might effect the efficiency of the advanced agent in the cases where there are many mines.

The improved average score of the advanced agent during smaller to medium mine densities may be due to the fact that in these case we may not get a lot of opportunities to use straightforward inferencing techniques of the basic agent. It might take a few random hidden cells to be revealed before we can apply such straightforward techniques. Whereas, the advanced agent uses querying the knowledge base and proof by contradiction to combine the various rules in the knowledge base. This kind of inferencing may give more information about the current state of the game in situations of average mine densities (not too high or not too low). So, advanced agent is able to figure more things out frequently during medium mine densities.

So, we believe the minesweeper becomes hard with higher mine densities. The advanced agent we constructed is able to work out things better than the basic agent in most of the mine densities presented. The advanced agent gets a better final score, meaning it could solve games more careful and detect most of the mines in the game. When there are more mines on the board, both of the agents have a higher probability of choosing a random location on the game which would be a mine. In that case, it's harder to draw inferences from the knowledge base if the locations revealed in the game are mines and lead to lower final scores.

Efficiency

Here we described some of the space or time constraints you run into in implementing this program and they are all implementation specific constraints. We believe most of the time constraints occurred in the DFS traversal of the variable assignment tree while querying the knowledge base.

Early termination: We could terminate as soon as we find a subset of variable assignments which is automatically impossible, and hence prune out the rest of the variable assignment tree. To some extent we took care of this by considering partial assignments and see if these could conclude an obvious contradiction at this earlier stage. However, we could probably consider more ways to detect contradictions at this point instead of just checking for basic contradictions. By basic contradictions I mean like if of mines exceeds clue value or clue value exceeds number of neighboring locations.

Unit clauses: If we know that some location has to be a safe or has to be a mine (in order to prevent contradiction) then we can automatically assign corresponding values (0's and 1's) instead trying out different values before coming down to this assignment. We took care of this in our code after realizing this technique.

Variable Ordering: Choose a location that is involved with the most number of rules in the KB earlier. So, we prioritize the locations based on the number of rules they are in. Greater this number, greater the priority of the location. We did not incorporate this into our code, but it is a way could improve on time/space constraints.

Value ordering: Assign 0 or 1 to a location based on which option will add the fewest constraints to the other variables. We did not incorporate this into our code, but it is a way could improve on time/space constraints.

The only specific space constraint was to store the knowledge base and variable assignment tree. We tried to simplify the knowledge base based on every clue to only keep the useful information.

Bonus Questions

Clever Acronym

The clever acronym we chose for the advanced agent we constructed is 'reductio ad absurdum' agent. This means 'reduction to absurdity' in latin which is a synonym for 'proof by contradiction.'

Global Information

In this case we can add one additional information/rule to our knowledge base at the very beginning. This rule would be that all the locations withing the mine will add up to the number of total mines.

Illustration: Suppose M(i,j) = 1 if (i,j) has a mine, 0 otherwise.

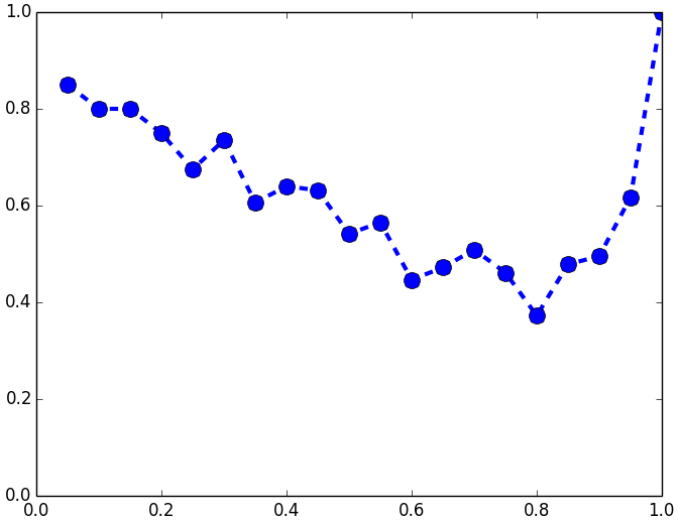
The rule we have to add to KB at the start:

((i,j) tuples for all location in game, total number of mines), which is basically saying:

$$\sum M(i,j) \; for \; all \; (i,j) \; in \; game \; should \; equal \; to \; total \; number \; of \; mines.$$

Therefore, whenever we have that rule updating such that the length of the list of locations equals the number of mines not explored then we know all the locations in that list are mines, so we can identify them safely and finish the game. Also, while updating this rule in the knowledge base (based on new information), if we ever notice that this rule's value becomes zero, we can safely conclude that the remaining locations in the list are safe and end the game.

Figure 2: Mine Density vs Average Final Score for Advanced Agent Using Global Information



Index: Blue- advanced agent using global information. X-axis: Mine Density in the game, $0.05 \leq q \leq 1.0$. Y-axis: Average Final Score.

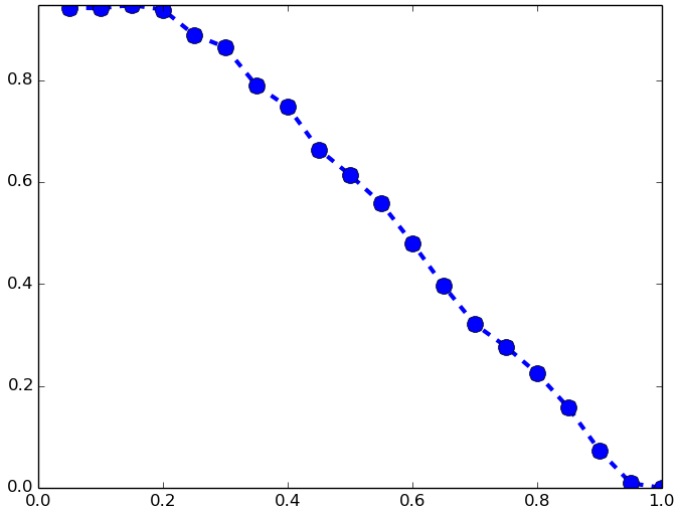
The graph shows that the final scores using global information in the knowledge base for the advanced agent helps to generate better final scores in most of the games and performs well even with very large mine densities. For example, we can observe that for mine density of 1.0 that average final scores are 1.0, means that for every board generated with this mine density the advanced agent was able to solve the whole game successfully and find all the mines. For low to average densities also, the performance is pretty good, because once some information is added to the knowledge base we can come to a point where we can identify a large number of cells as safe or as mines.

Better Decisions

We used a method that take the current game and output a hidden cell that was given the least probability of being a mine. We used a dictionary with keys as the hidden locations and values as probabilities.

We assigned probabilities as follows: If there is a neighboring safe cell then the corresponding probability for this hidden cell will be: (of unidentified mines around mines around the safe cell)/(of unidentified cells around the safe cell). If there are multiple safe cells around the hidden cell, then the final probability of the hidden cell will the maximum of all such probabilities. Finally we return the key in the dictionary with lowest value/probability.

Figure 3: Mine Density vs Average Final Score for Advanced Agent Using Improved Cell Selection



Index: Blue- advanced agent using improved cell selection. X-axis: Mine Density in the game, $0.05 \leq q \leq 1.0$. Y-axis: Average Final Score.

The graph shows that the final scores using improved cell selection in the knowledge base for the advanced agent helps to generate better final scores especially for the lower mine densities. This mechanism chooses the random cell which has the lowest probability of being a mine from the information we have in the knowledge base at a given time. The plot overall demonstrates better final scores that the plot of the original advanced agent, so the selection mechanism improves random selection.

Summary

We both worked on the logic and code for the problems in this project, we brain stormed ideas, and agreed on the solutions we got it. We met on Zoom to write together the code for this project to generate the games, the two agent algorithms, and improvements for the advanced agent. We collaboratively found solutions to issues and bugs we faced while testing and running the code. We both were running test cases on the code outside of our Zoom meetings (generating graphs, generating pictures of games, running different games with different inputs). In this report, we wrote the performance 1 and bonus questions sections together. Mostly it was completely collaborative but the below are some individual things we did:

Gal: Got the latex document ready with the relevant sections, visualized and generated the graphs for some questions, debugged the code and found solutions for the issues, offered ideas or more efficient ways of implementations while working on the code together, and added the pictures to the report. Wrote the representation and inference sections and added on to performance 2 and decisions sections.

Vaishnavi: Debugging parts of advanced agent, thinking about how to implement the advanced agent with constraint satisfaction and how to implement variable assignment tree traversal. Added on to the following parts of the writeup: representation, inference. Wrote the following parts of the writeup: decisions, performance 2, efficiency.

We both wrote some methods of the code, debugged separately, and contributed to the answers on this report. This write up was written by both Gal and Vaishnavi, each one of us was responsible for some questions and we added on to each other's answers.

I read and abided by the rules laid out, I have not used anyone else's work for our project, my work is only my own and my partner's.
Gal
Vaishnavi