CS 520: Intro to Artificial Intelligence **Rutgers: Fall 2020**

Project #2

October 31, 2020

Name: Parth Patel - pbp71, Nihar Prabhala - np642, Rohan Shah - ras513, Rumeet Goradia - rug5

Background

File Structure and How to Run

- common.py
- commonCSP.py
- commonProbability.py
- strategy1.py (Basic Agent)
- strategy2.py (Improved Agent)
- strategy3.py (Doubly Improved Agent)
- strategy4.py (Triply Improved Agent)
- testing.py (Driver Code)
- diffBoard.py (Hard Maze Generation)

To run our code in a single location, you can run *testing.py* with an input argument of either 1,2,3, or 4, all which correspond to the strategy of interest. The driver will run code from $dim = 10, 20, ..., 50$ and, along with mine densities of $0.1 * dim^2, 0.2 * dim^2, ..., 0.9 * dim^2$, will generate the graphs we have shown in this report.

To access our difficult board analysis, view the README.md.

common.py

This file creates the environment for our project. It serves as the base for our board and agent generation. It contains the necessary methods and attributes to ensure that each strategy is able to perform its tasks. The basis for these two classes is the *Cell* class. The *Cell* class contains attributes that focus on a certain cell as well as its neighbors.

Board

This class generates the playing board. We create a square board with dimension *dim* Thus, the board will hold dim^2 cells. The *mine density* indicates what percentage of the total cells on the board will be turned into mines. We implemented the option to either place these mines randomly, or use a set of predetermined mine locations indicated by the user. For testing the agent strategies, we used the randomly generated mines. For determining particularly difficult boards, we used the predetermined locations to set the mines. When inserting the mines into the board, the methods take care of creating clues on the game board for the Agent to access.

Agent

This class creates *Agent* objects that are used with the strategies we have implemented. This Agent will traverse the board according to the strategy, and will keep track of the different types of cells that it encounters. Thus, there are 3 lists maintained in this object: *Revealed Cells, Tripped Mines, and Identified Mines.* The Agent also maintains a list of preferable cells to traverse if a strategy requires it. Agent also contains a variety of helper methods, such as choosing random coordinates that have not been explored before, and checking to see if the game has been completed.

commonCSP.py

This file deals with the methods needed to interact with the *Knowledge Base* and create inferences.

Format of the Knowledge Base

Our Knowledge Base is a two-dimensional list that keeps track of all known information for a particular strategy. It is not used in Basic Agent, but it has extensive use in all the Improved Agents.

Formally, all variables (cells) are integers. These variables will be equal to zero (safe) or one(mine). The LHS of an equation can contain up to eight variables, and the RHS will be a value between zero and eight.

An equation is represented as a list of integers. The last element of each list is the clue value for that equation. All other elements of the list are variables represented by a unique value drawn from its cell's coordinates.

Method Descriptions

When we add new *equations* to the Knowledge Base, it needs to be thoroughly vetted. This means that we must see if the equation we are trying to insert is either in the Knowledge Base already, or if it can be simplified substantially before insertion. The simplified equation is then used to reduce the equations in the Knowledge Base and added to the Knowledge Base.

The "reduction" of an equation is done by determining if the equation is a subset of any of the equations in the Knowledge Base. Once this determination is made, we take the set difference between the two and store that as the equation we are reducing.

As we continue to take subsets and create simplified equations, we can finally check for inferences in our Knowledge Base. In particular, we check to see if the cells in an equation are either all safe, or all mines.

Sweeping a Mine or Two

To view the agents in action, refer to the attached GIF files. Similarly, we have included a few key scenarios for each agent to showcase their prowess in the Appendix.

Basic Agent: The Basics

Introduction

This strategy makes use of a few rules, but does not evaluate the entire board at once. Rather, it only looks to its neighbors and determines if there are any inferences that can be made at the time of reveal. If nothing can be deduced, we continue to reveal random cells.

In particular, our implementation of this strategy follows the following format: At the beginning of the game, we pick a cell on the board randomly. As the game continues, we check to make sure our picked cell has not been visited before. If the difference between a cell's clue and its mine neighbors is equal to the remaining unknown cells, then the remaining unknown cells are mines. The complement of this idea is implemented to determine if the remaining unknown cells are safe.

Run-Time and Storage

In Basic Agent, we have no need to create a complex data structure to store information. No inferences are drawn from known information across different cells. Thus, the total storage space needed, for this algorithm, is $O(n)$, where $n = dim^2$.

The time complexity is $O(n)$, as we are performing subtractions n times.

Results

The results from Basic Agent are as expected. Since no inferences are being made, we see that the strategy's ability to make viable decisions is hampered. Success rates hover at just above 20%, giving us a good baseline with which we compare our improved agents.

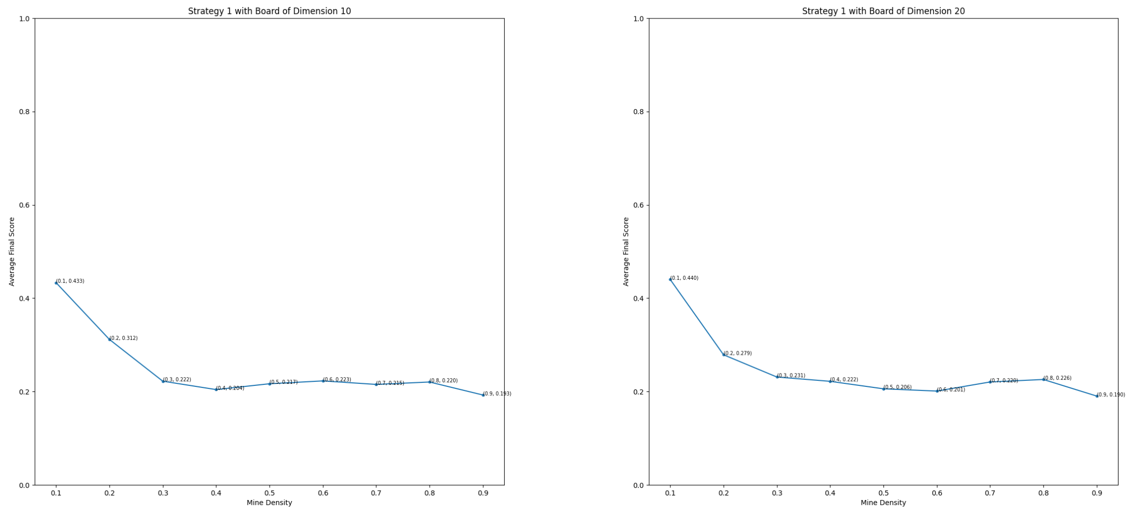

Figure 1: Dimensions 10 and 20 for Basic Agent

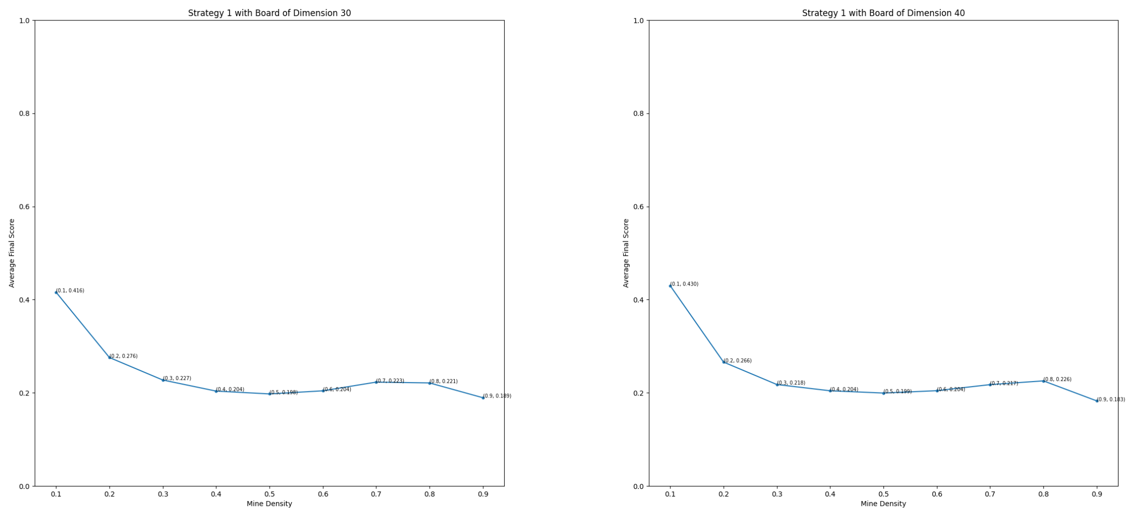

Figure 2: Dimensions 30 and 40 for Basic Agent

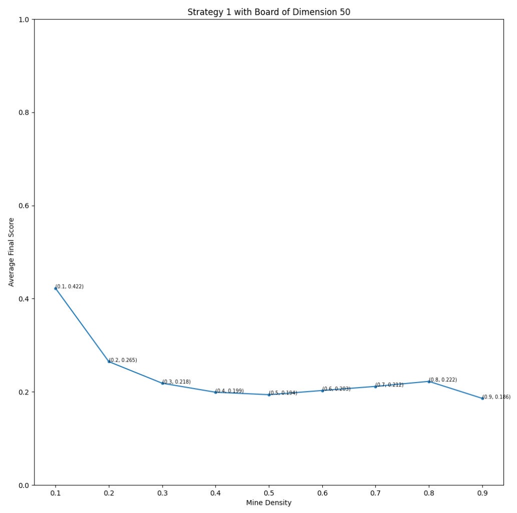

Figure 3: Dimension 50 for Basic Agent

Improved Agent

Introduction

With the Improved Agent, we implement the Knowledge Base in order to draw a variety of conclusions based on the cells we have revealed. This agent is presumed, and later shown, to be much better than the Basic Agent. This Agent, along with the Basic Agent, will serve as foundational building blocks for the remaining agents, as well as the generation of difficult Minesweeper boards.

An Example

The elements of the Knowledge Base are lists that represent equations that are generated once we reveal a cell. Each equation either denotes the determination of a cell type and/or a description of a cell's neighbors and the cell's clue. It is best explained through an example:

A	B	C
D	R=1	E
F	G	H

In the example above, R denotes a cell that has been *revealed*, while all the other cells are unknown. When a cell is revealed, it can either be safe or a mine. In our example, the cell is safe and reveals a clue of 1. This means that out of neighbors of R, one must be a mine. We can postulate this information in the form of two equations:

$$A + B + C + D + E + F + G + H = 1 \tag{1}$$
$$R = 0 \tag{2}$$

If, for example, we revealed cell H, and it had a clue of 1, we would introduce the following equation:

$$G + R + E = 1 \tag{3}$$
$$H = 0 \tag{4}$$

Since we know R is safe, this would get simplified to:

$$G + E = 1 \tag{5}$$

H is now known, so the original equation would get simplified to:

$$A + B + C + D + E + F + G = 1 \tag{6}$$

Using the previously inferred information about G and E, we can write the entire Knowledge Base as the following:

$$\begin{aligned} A + B + C + D + F + G + E &= 1 \\ G + E &= 1 \\ A + B + C + D + F + 1 = 1 \iff A + B + C + D + F &= 0 \\ G + E &= 1 \\ R &= 0 \\ H &= 0 \end{aligned}$$

Now, since we know that A, B, C, D, F all equal 0, all of those cells must be safe. This cascading effect is explored through checking for inferences.

Implementation

As we have discussed in commonCSP.py, we need to make sure that our Knowledge Base only inserts equations that have already been reduced to purely new information. Similarly, when including this new information in our Knowledge Base, we can use it to update the information already at hand. This creates a variety of recursive-nests that we have implemented in our code.

When we first reveal a cell, we check its type and create a corresponding assignment equation. If the cell is safe, then we use its clue and its neighbors to construct a formula like the one shown above. These two equations are passed, respectively, as arguments to the *addEq* method.

In *addEq*, we reduce these equations. As mentioned above, this means that we check to see if the equation passed as an argument has any valid subsets compared to the information in the Knowledge Base. This is done using *reduceEq*. After we have determined that the equation is in its simplest form, we check to see if it is a subset of any of the equations within the Knowledge Base, which is done using *reduceKB*. Finally, we add it to the Knowledge Base if no new information comes from the new equation.

After reducing a new equation and the Knowledge Base, a decision must be made on what cell to reveal or identify as a mine. We do this by using our *checkForInference* function. For every equation in the Knowledge Base, *checkForInference* checks if the clue value is zero or if it is equal to the number of variables in the equation. If the clue value is zero, it can be inferred that all of the variables in that equation are safe and the Knowledge Base is updated with that information. Also, these inferred safe mines are added to a queue to be revealed next for more information. If the clue value is equal to the number of variables in the equation, it is inferred that all variables in this equation are mines and the Knowledge Base is updated accordingly. This follows the rules established in Basic Agent, but with the information gleaned from our Knowledge Base, we are able to make these inferences more frequently. The Knowledge Base possesses information about cells that are not direct neighbors of any given cell. Therefore the scope of our knowledge is extended further than what we had in the Basic Agent. In the case that an inference cannot be made, we apply a random selection of unrevealed coordinates to explore.

Algorithm 1 Pseudo-code summary of ReduceEq and ReduceKB

```
Require: X is a set, Y is a multiset
  $i = 0$
  while $i < len(Y)$ do
    if $X \subseteq Y_i$ then
      $X \leftarrow X \setminus Y_i$
      $i \leftarrow i + 1$
    end if
  end while
  $i = 0$
  while $i < len(Y)$ do
    if $Y_i \subseteq X$ then
      $Y_i \leftarrow Y_i \setminus X$
      $i \leftarrow i + 1$
    end if
  end while
```

Run-Time and Storage

The run-time complexity of this algorithm is based entirely on the Knowledge Base. By looking at *reduceKB* and *reduceEq*, we can find an effective bound for its run-time. In our calculations, we say that $n = dim^2$.

In the context of our calculation, the *isSubset* function in Python runs in constant time. This is because *a.isSubset(b)* computes in *len(a)* time where the maximum length of an equation a is 9 *(8 variables + 1 clue value)*. Then, *reduceEq* runs in $O(n)$ time because each equation is reduced by at most $O(n)$ equations already present in the Knowledge Base. Finally, *reduceKB* runs in $O(n^2)$ time. This is because *reduceKB* reduces each equation in the Knowledge Base, and if there are n equations, in the worst case, then this operation takes $O(n)$ time per iteration. Then, the Knowledge Base must be recursively reduced again by the equations that had been just been reduced resulting in $O(n^2)$ run-time for *reduceKB*. Finally, *reduceKB* is called once per cell in the minefield, resulting in a total run-time of $O(n^3)$ for the Improved Agent.

The storage cost of this algorithm is defined by the size of the Knowledge Base. Since the representation of our Knowledge Base is $O(n)$ space complexity.

Results

With the Improved Agent, we see an immediate increase in the success rate across different mine densities (where mine densities are defined as $0.1 * dim^2, 0.2 * dim^2, ..., 0.9 * dim^2$). As the dim increases, we see that the average success rate increases as well. This attests to the idea of modelling our Minesweeper as a Constraint Satisfaction Problem. By modelling the cells and their neighbors as equations, and by keeping track of information that we learn about, we can make inferences that allow us to solve our Minesweeper games more effectively.

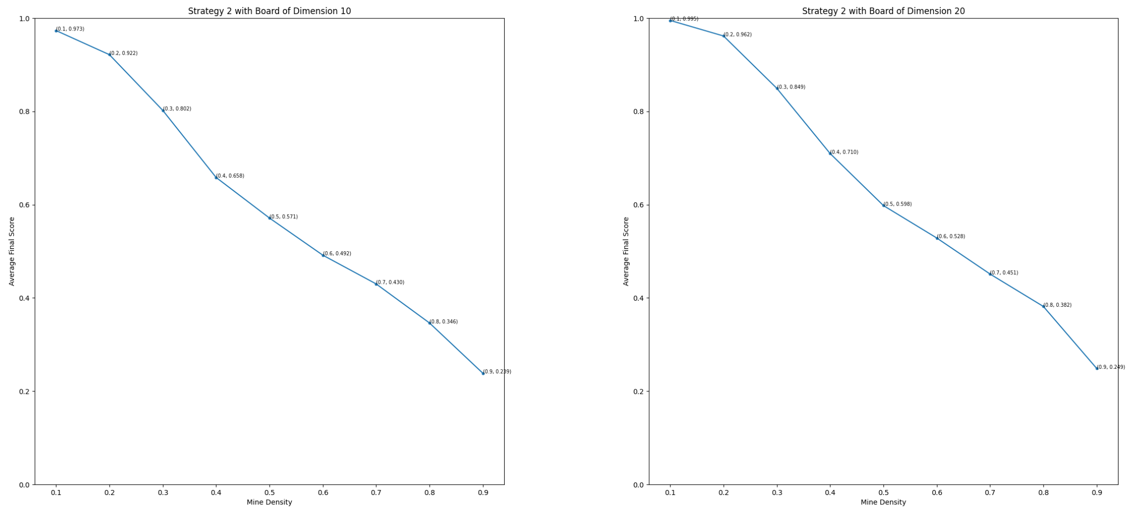

Figure 4: Dimensions 10 and 20 for Improved Agent

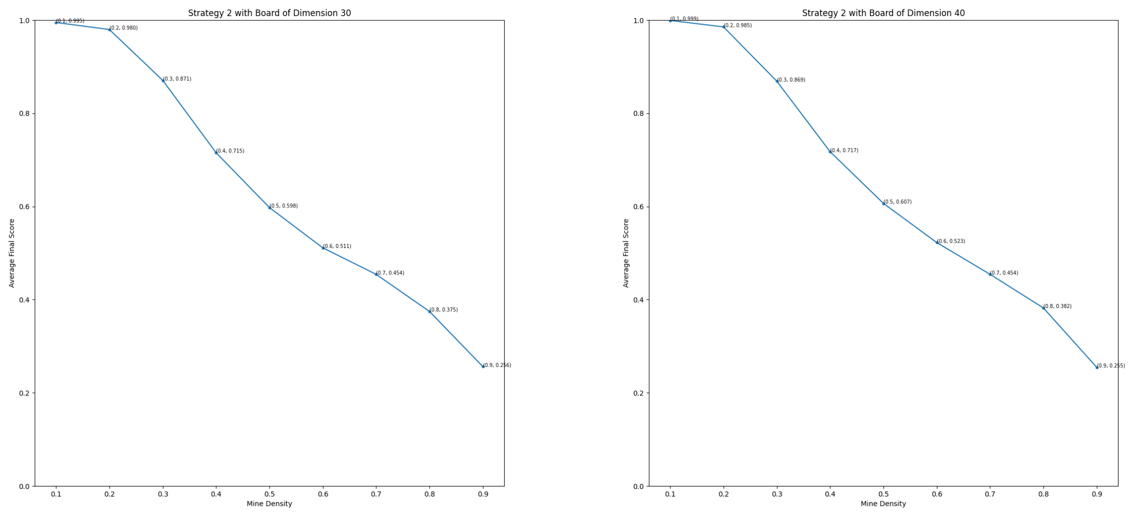

Figure 5: Dimensions 30 and 40 for Improved Agent

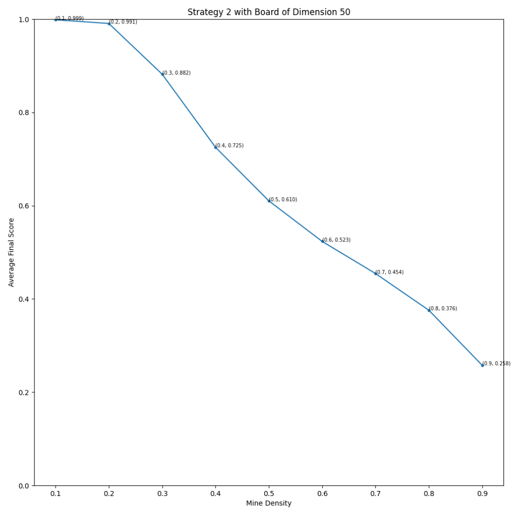

Figure 6: Dimension 50 for Improved Agent

Strategy 3: Doubly Improved Sweeper

Introduction

With the Doubly Improved Agent, we calculate the probability that any given cell on the board is a mine before every decision this Agent makes. We effectively simulate all possible configurations of the board based on the variables in our Knowledge Base and draw conclusions accordingly. This Agent also uses the same equation format that Improved Agent uses for storage in the Knowledge Base. As such, we reused and re-purposed many of the helper functions like *addEq* in this implementation. This Agent includes the introduction of new functions to perform these simulations: *calculateVariableProbabilities*, *findValidConfigs*, *createVariableGraph*.

Implementation

The core part of the Doubly Improved Agent is completed using the *findValidConfigs* function. In order to calculate the probability that an unknown cell is a mine, we must consider all possible configurations of the board with the current variables, and check how many of those configurations are valid (i.e. do not contradict any information already in the Knowledge Base) and subsequently in how many of those valid configurations the given cell is a mine. So, using a recursive tree-search implementation, the *findValidConfigs* function finds all possible combinations of the variables in our Knowledge Base as mines or safe cells. For each recursive step, we first simulate adding the a variable as a safe cell to our Knowledge Base, and then simulate adding the same variable as a mine to our Knowledge Base; we return the sum of total number of valid configurations from each respective branch. Once a leaf node is reached (i.e. there are no more variables to consider) and the calculated combination of variable values is valid, then the function tracks which variables have been simulated as mines and increments their "mine count" in a dictionary. In this dictionary, the keys are the variables and the values are the number of times they have been simulated as mines in valid configurations.

Calculating probabilities for any cell, then, becomes straightforward: divide the value for this variable in the mine count dictionary by the total number of valid configurations. If the probability of a specific variable being a mine is 0, then that variable is stored in an array specifically for safe variables. Similarly, if the probability of a specific variable being a mine is 1, then that variable is stored in an array specifically for mine variables. All other variables are sorted based on their probabilities in the original list of variables. Since we know that all safe variables will be safe regardless of future variable revelations (and the same for mine variables), we check all coordinates corresponding to safe variables and identify all coordinates corresponding to mine variables before re-calculating probabilities for the minefield.

Initially, when we first tried to implement this, we were running into extraordinary run-times for higher dimension values. The recursion tree was getting far too deep and taking enormous amounts of computing power when trying to consider all possible configurations of a board given a set of unknown cells. Instead, we were able to reduce our search space of valid configurations handsomely.

We implement two different pruning methods. The first pruning method simply involves checking after each variable value simulation if the configuration is still valid; if it is not valid, we do not have to go further down this branch. Similarly, the second pruning method reduces the number of sub-branches to be explored by using the current simulated Knowledge Base to apply inferences similar to those in the Basic Agent, so that we are able to eliminate branches that do not match these variable assignments. Moreover, to decrease the maximum depth of our search tree, we organize the variables into a graph of connected components in the *createVariableGraph* function. Each connected component is composed of variables that appear in equations in the original Knowledge Base with each other; this concept of "neighbors" is based on the fact that a specific variable's assignment as a mine or a cell will only impact the potential values of other variables that share an equation with that variable. We are able to optimize the storage costs of the Knowledge Base by implementing the *thinKB* function. This function looks to see whether a cell has exhausted all of the information it could possibly give us. Namely, if all of the neighbors of a cell have been revealed, we have concrete information about the neighbor, and the clue no longer provides any valuable additional information; so, we remove it from the Knowledge Base. In addition, each connected component is matched with a "relevant Knowledge Base", which is a subset of the original Knowledge Base whose equations apply directly to the variables in the respective connected component.

Run-Time and Storage

The most time-consuming aspect of the Doubly Improved Agent is undoubtedly the calculation of all valid configurations, and thus, to analyze the run-time of this strategy, we must analyze the run-time of the *findValidConfigurations* function. Before implementing the above discussed optimizations, this function had a worst-case run-time of $O(n^2 * 2^n)$. This is due to the fact that this function's run-time is dependent on the number of variables passed into the function, and the maximum (theoretical) number of variables is n. In addition, each pass of this function involves copying the Knowledge Base, which could have a maximum of n equations, and the *reduceKB* function has a worst-case run-time of $O(n^2)$. Even after optimizations, the worst-case run-time for this function remains $O(n^2 * 2^n)$, as all variables would remain in the same connected component. Since the *strategy3* function can run a total of n times, the total run-time of this algorithm is $O(n^3 2^n)$.

The maximum number of equations in the Knowledge Base, as stated before, is n, and the maximum number of variables is also n. Therefore, the storage cost of this algorithm is simply $O(n)$.

Results

Overall, it is clear that the Doubly Improved Agent consistently outperforms the Basic Agent across all board dimensions and mine densities. This shows us that we were able to draw several effective conclusions about the board by simulating the mine configurations for sets of unknown cells on the board. By calculating so many different valid configurations, we are able to effectively calculate probabilities. This gives a much larger Knowledge Base from which to draw information from to make our next decision. With the additional information, we are able to more often make the correct decision in exploring the board.

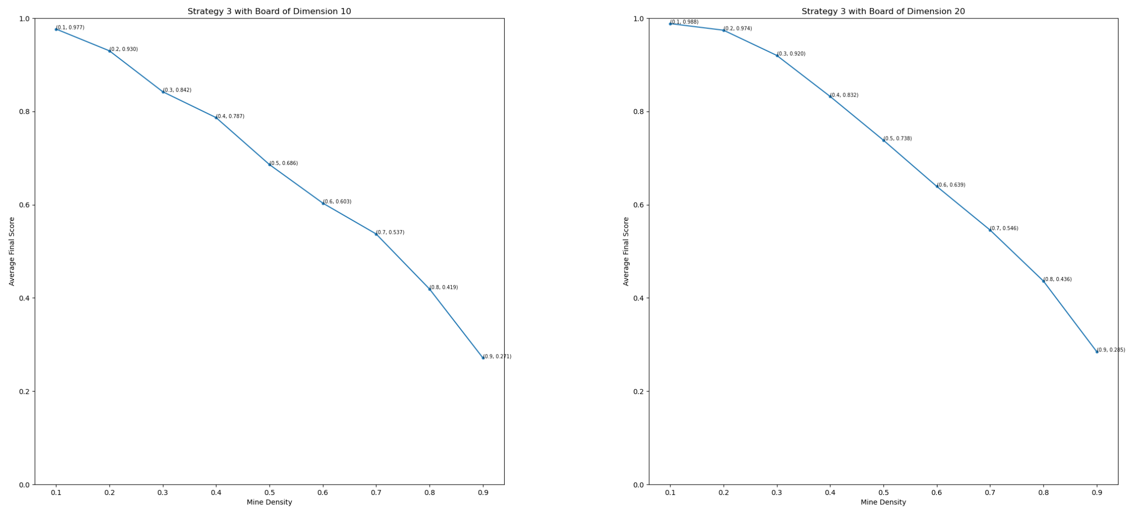

Figure 7: Dimensions 10 and 20 for Doubly Improved Agent

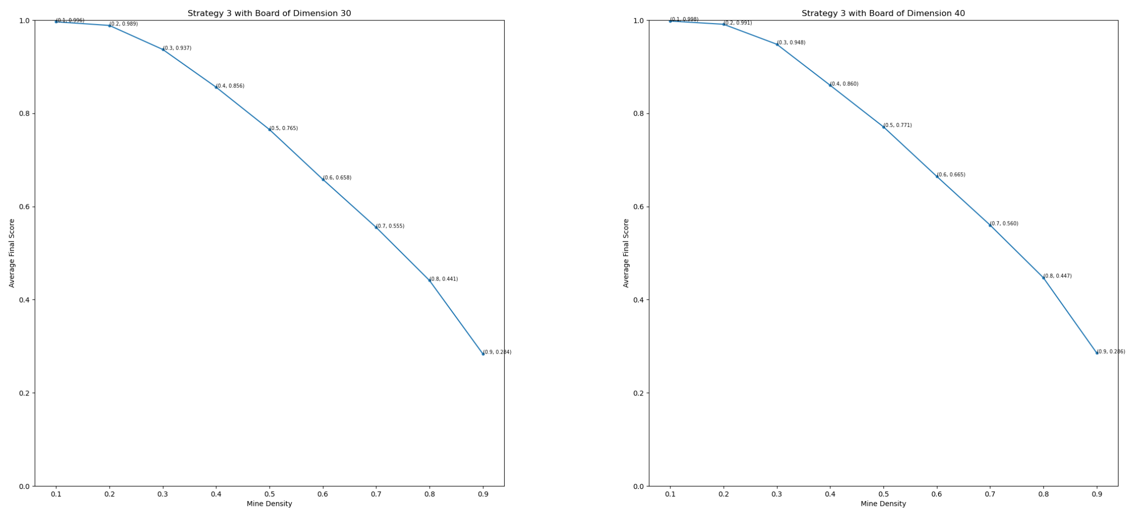

Figure 8: Dimensions 30 and 40 for Doubly Improved Agent

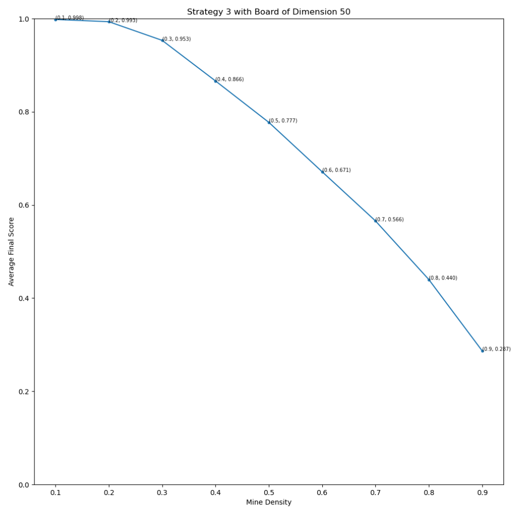

Figure 9: Dimension 50 for Doubly Improved Agent

Strategy 4: Triply Improved Sweeper

Introduction

The Triply Improved Sweeper is an amalgamation of the first three agents. By combining a probabilistic model and a logical model, the agent makes informed decisions about which cells to pick next and which cells to flag as mines.

Implementation

Much like the Doubly Improved Agent, the Triply Improved Agent implements a function called *calculateVariableProbabilities*. In this method, however, there lies one key difference: a threshold value. In this implementation, we do not bind the Agent rigorously to strict definitions of 0% probability for a variable to be declared a safe cell. Rather, if the Agent is able to deduce that the probability of a certain cell being a safe cell is less than 12.5%, our Agent adds it to a list of safe variables. (Because we cannot risk incorrectly identifying a mine, the required probability for a variable to be declared a mine remains 100%).

The safe cells are then all checked by the Agent without probability recalculation, but since there is a small chance that they are not actually safe, we employ a check to ensure that the correct type of equations are being added when checking these variables. The Agent also identifies all the cells in the identified mines list returned by *calculateVariableProbabilities* as mines. Before recalculating all probabilities, we employ a version of the *checkForInference* function from the Improved Agent to remove any variables whose values can be directly inferred. Then probabilities are recalculated for the entire set of remaining variables, and this process then repeats. Once there are no more cells that can be probabilistically inferred to be safe or mines, we choose the safest remaining cell.

The *checkForInference* function is also called once the Agent checks a cell outside of the aforementioned loop. The main benefit, then, comes from the combination of employing this function and choosing the safest cell to explore next.

Run-Time and Storage

The run-time for this algorithm will effectively be the same as the Doubly Improved Agent. If there is nothing that can be inferred from the Knowledge Base at a given point in time, we have to run the *findValidConfigs* function on the entire set of variables. This will take time on the order of $O(n^2 2^n)$ for the same reasons as the Doubly Improved Agent. In this scenario, if there are n cells to traverse through, the total run-time will be $O(n^3 2^n)$.

In terms of storage costs, this will also be on the same order as the Doubly Improved Agent. We are not storing any significant amounts of data. The data is merely being grouped together to help make more assumptions about the board. We are still storing graphs, relevant Knowledge Bases, and the overarching Knowledge Base (the dominant storage cost) in the same way, so our storage cost is also $O(n)$.

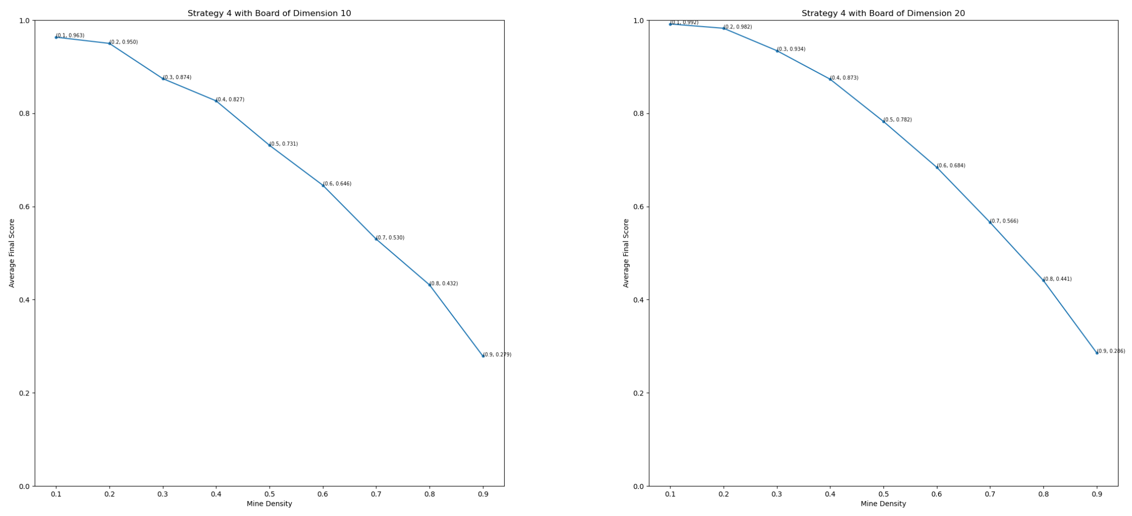

Figure 10: Dimensions 10 and 20 for Triple Improved Agent

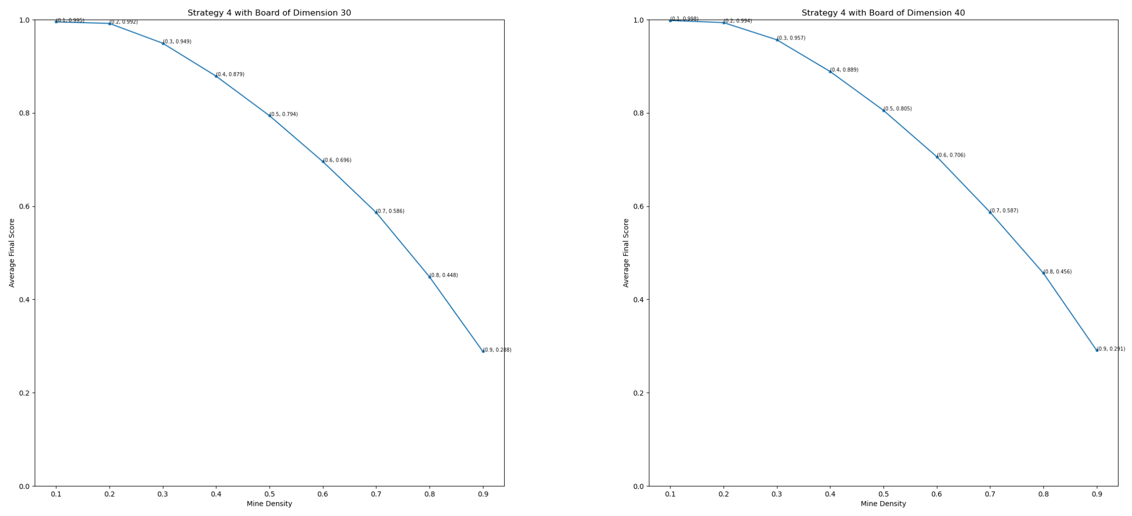

Figure 11: Dimensions 30 and 40 for Triple Improved Agent

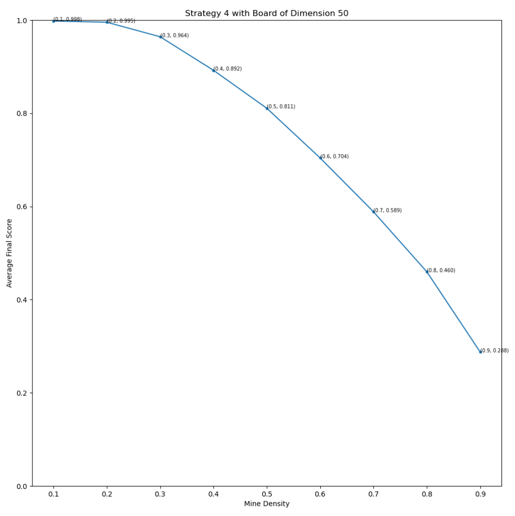

Figure 12: Dimension 50 for Triple Improved Agent

Remarks and Conclusion

In this section, we seek to compare all four Agents across different *dim* values and different mine densities. It is immediately apparent that the Basic Agent is the most lackluster of the four. As we have mentioned, this agent is unable to make basic inferences across the entire board. When the Basic Agent considers a cell, it only looks to its neighbors to determine if the clue has been satisfied.

The other Agents perform much better in comparison. The Improved Agent has a larger decrease in success as we see an increase in mine density when compared to the Doubly and Triply Improved Agents. This can be attributed to the agent's inability to choose cells that have the lowest probability of being a mine. It will randomly choose a cell if it cannot make any inferences.

The Doubly and Triply Improved Agents display a common theme in statistics: Diminishing Returns. The two agents perform roughly the same, with the Triply Improved Agent edging out its predecessor minimally.

Tests have also shown that the detection rate for mines across the Improved Agents linearly with mine density. However, the Basic Agent performs about the same, but still relatively poorly, for all mine densities. Additional testing in Hard Maze Generation showed that all strategies' detection rates decrease for boards where mines are clumped up. A analysis of this trend can be found in the next section.

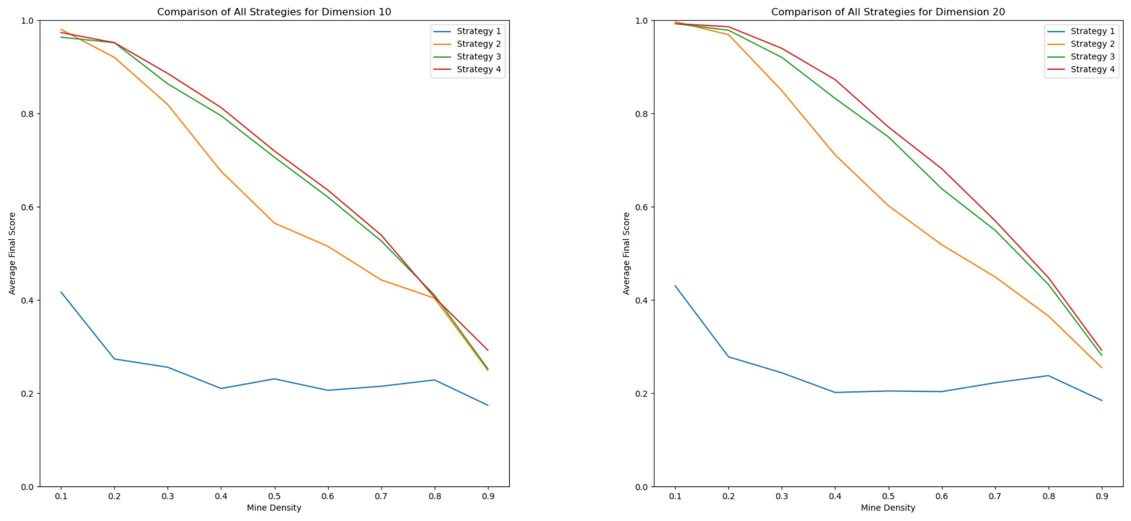

Figure 13: Dimensions 10 and 20 for All Agents

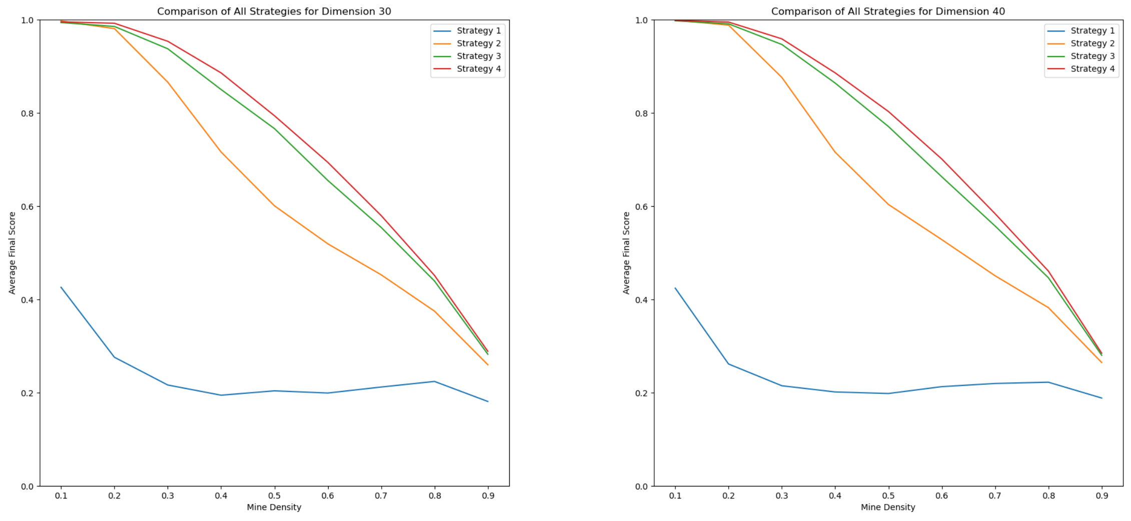

Figure 14: Dimensions 30 and 40 for All Agents

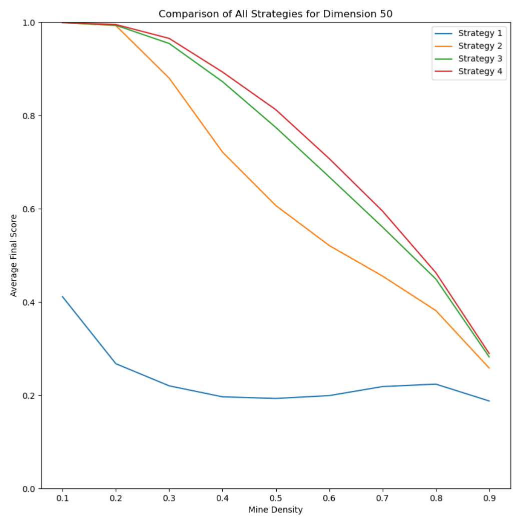

Figure 15: Dimension 50 for All Agents

Hard Maze Generation

Introduction

A difficult board can be defined as one that offers the lowest success rate on average. Using this definition, we constructed a local search that generates different mine configurations in order to force the average success rate downward. By doing so, we utilize the Improved Agent. In order to produce failing boards, we need to force the agent to make as many guesses as possible. This means that the cells do not reveal a lot of information about locations of the mines. Rather than being able to make a deduction, the agent is repeatedly forced to guess, thus making it more likely to hit a mine.

Intuitively, we arrive at a scenario where the mines must be placed in clusters. In this configuration, mines hidden within clusters can only be accessed by being guessed directly. We quickly tested and confirmed this theory by brute forcing all mine configurations for boards of dimensions 3x3 and 4x4. This can be explained by looking at an example:

A	B	C
D	M	M
F	M	M

In this board, we see that the mine in the bottom right corner has no opportunity to be discovered. Eventually, the agent will discover the mine at (1,1), (1,2), and (2,1). However, since the agent has no information on the total amount of mines in the system, it will automatically pick (2,2) in order to complete the game, thus forcing a tripped mine. This intuition is something that has been verified in our testing. By creating clusters of mines, the average success rate decreases.

Generation

To generate these types of boards, we use a local search algorithm that creates different mine configurations. In a sense, it is a hill-climbing algorithm, in that we are hunting for an optima by exploring all the current state's neighbors.

In the context of this problem, we consider the search space to be all configurations of a given board with given dimension and mine density. The "neighbor" state of a configuration on a board are the possible locations the mine can move to, discussed below. It would be too computationally taxing to test every possible location of every mine on the board.

We do this by first randomly generating a board, and then looping through each of the mines inside of the board. We consider a relaxed version of the mine's possible movements: rather than having the mine consider a maximum of 8 possible neighbors, we consider only vertical and horizontal movements. The rationale here is that these movements still cover the entire board. While we consider a particular mine, we hold all the others in place. When this mine has been moved to a neighbor that produces a lower success rate than we established in the baseline, we keep track of that location, and then move on to another mine. Each new mine configuration is worse than the previous configuration, thus following suit with the absence of backtracks in a local search algorithm.

If the success rate of the board does not continue to decrease at 1% or more, then we allow the algorithm to continue its search up to n times, where n is determined by the user. In our testing, we noticed that the decrease in the success rate slowed significantly as time went on. As our returns continued diminishing, we had to determine a threshold to stop iterating through potential mine boards.

As we discussed in class, if we were to let this algorithm run infinitely, we would be able to obtain a global optima. In our case, we have constraints that make this concept infeasible. Thus, we bounded our local search to this differential.

In the graphs we have generated for developing a difficult board, we were only to generate plots for *dim* up to 15. Solving the Improved Agent for each possible mine location is very taxing computationally and doing so for so many different mine densities is even more taxing. Anecdotally (while this is obviously not as concrete as real data), we noticed that the general pattern of success rate decreasing by 10-15 percent generally was upheld.

Algorithm 2 Pseudocode summary of Hard Board Generation

```
Require: dim is size of board, RND is the random restart limit, findNeighbors returns all valid horizontal
  and vertical neighbors
  $X \leftarrow Board(dim)$
  $baseScore \leftarrow BasicAgent(X)$
  $counter \leftarrow 0$
  while $counter < RND$ do
    for all $mines$ in $X.mineList$ do
      $neighbors \leftarrow findNeighbors(mine, dim)$
      $mineList, lowerScore \leftarrow findWorstConfig()$
      $delta \leftarrow (lowerScore - baseScore)/(baseScore)$
      if $delta > 0.01$ then
        $counter \leftarrow counter + 1$
      else
        $baseScore \leftarrow lowerScore$
      end if
    end for
  end while
```

Run-Time and Storage

A run-time analysis of this algorithm hinges on the run-time of the Improved Agent. By constantly checking the success rate of boards, we can say that this algorithm is definitely some factor multiplied by the total computational complexity of the Improved Agent.

Thus, we consider the total run-time based on the characteristics of the algorithm. We run the Basic Agent 15 times per new mine configuration. Each mine checks 4 potential neighbors. The computational worst case would represent the best case for our algorithm. If we were able to achieve consistent decreases in the success rate, the algorithm would run n times. Finally, since we allow for x random restarts, this leads to a worst case run-time of $O(xn(0.4n * 4 * n^2)) = O(xn^4)$ where $n = dim^2$. More likely than not, we would leave x as a constant (1), and we would see that the algorithm does not consistently decrease the success rate, leading to a lighter $O(n^3)$ run-time.

Storage costs are similar to the Improved Agent.

Results

As dimension increases, we see a decrease in the difference between the success rate and the transformed worst boards. This can be attributed to the fact that there is more information as dim increases. Similarly, due to the limitations of our computers, we could not randomly restart as frequently as would be required to push these towards their lower limit.

In the boards generated, we see a multitude of clumps to varying degrees. The locations of these clumps are variable, but there seems to be a tendency for these clumps to form at the edges/borders of the board. As mines get more packed together, less information is available and the Agent has to rely on guessing more often.

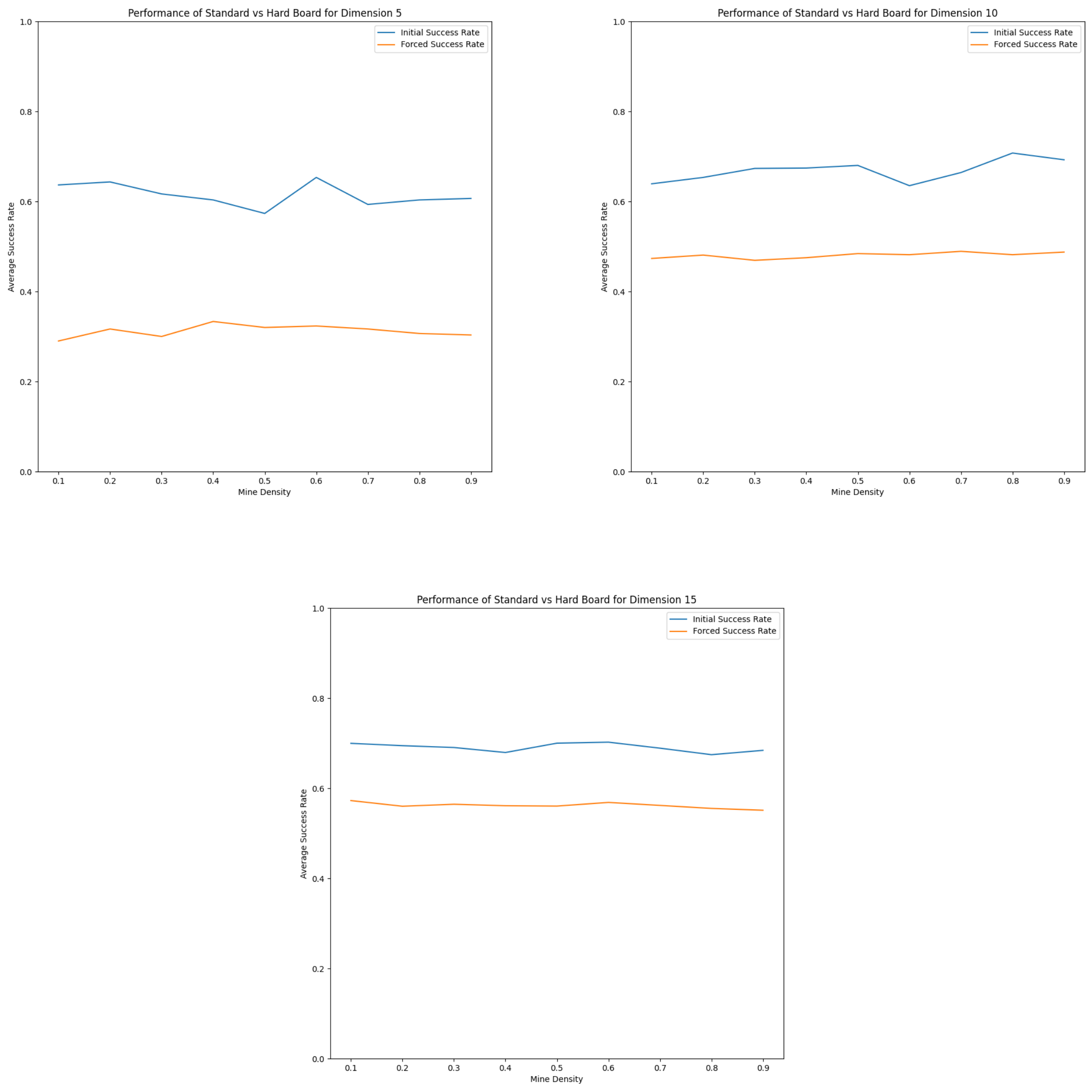

Figure 16: Difficult Boards for Different Dimensions

Appendix

Visualization Credit

The visualizations for the mine fields are not completely original work. Much of the logic behind creating these graphics comes from @jakevdp's blog post from 2012. We believe that these visualizations were not a core component of our project, and thus we felt justified in utilizing similar code from this blog post for our purposes. These visualizations were also used to generate the GIF images referenced at the beginning of this report. No plagiarism is intended.

Division of Work and Attestation

In terms of brainstorming potential implementations for each agent, all four of us equally contributed ideas that constituted our final submission. In terms of actual coding, Rohan and Rumeet took the lead in writing the basis for the entire project, including the Board, Agent, and Cell representations, as well as a few helper methods. Parth and Rohan collaborated heavily for the Basic Agent and the Improved Agent, with Parth taking the lead on a majority of the CSP-related functions. Rumeet and Nihar collaborated heavily for the Doubly Improved Agent and Triply Improved Agent, with Nihar taking the lead on a majority of the probability calculating functions and Rumeet taking the lead on a majority of the optimizations. Rohan and Nihar worked together to compose the code for the difficult board generation. Nihar generated the tests for all strategies, as well as the visualizations for the 10x10 boards for each strategy (including the GIF images).

Play-by-Play Visualizations: Specific Scenarios

As previously mentioned, please refer to the GIF images attached to this project submission for an overview of each type of agent's progression through a 10x10 board.

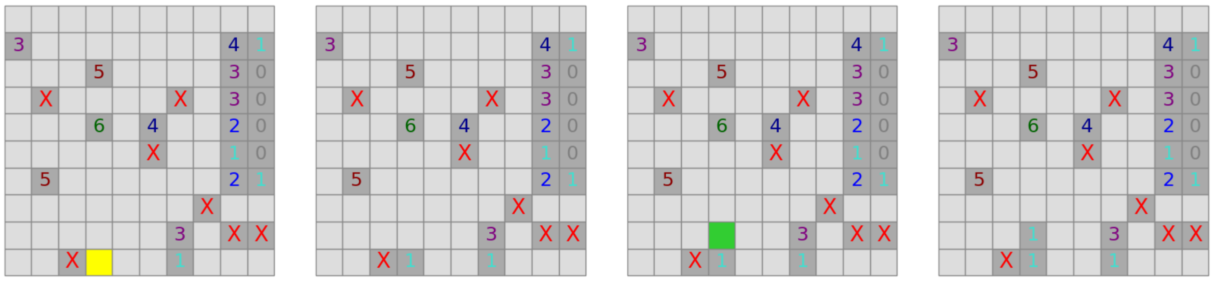

Figure 17: Basic Agent. The yellow cell is randomly picked. Because its clue is satisfied, the green cell is inferred.

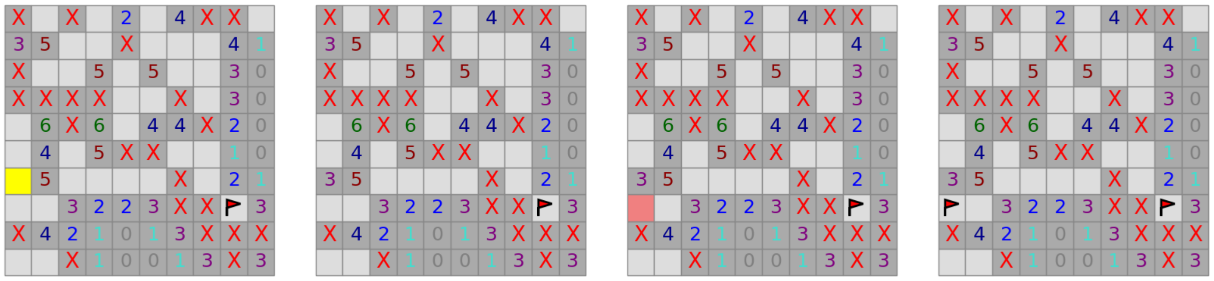

Figure 18: Basic Agent. The yellow cell is randomly picked. Since its clue matches the amount of its neighbors unrevealed, it can mark all of them as mines.

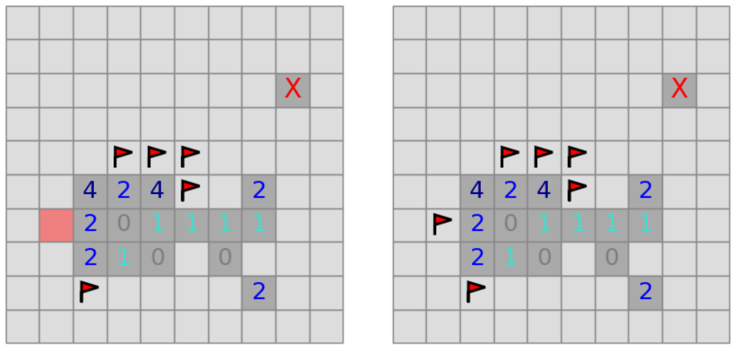

Figure 19: Improved Agent. The red cell is inferred to be a mine by the algorithm. Since the clue at position (5,3) is satisfied, its neighbors will be safe. This leads the clue at position (5,2) to be satisfied only if (5,1) is a mine and (4,1) is a mine.

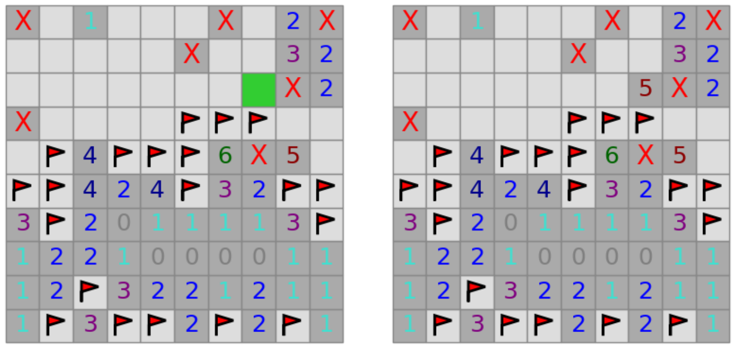

Figure 20: Improved Agent. The green cell is inferred by looking at cells (0,8) and (1,8). Since one of the two remaining neighbors of cell (0,8) must be a mine, the remaining neighbor of (1,8) must be safe.

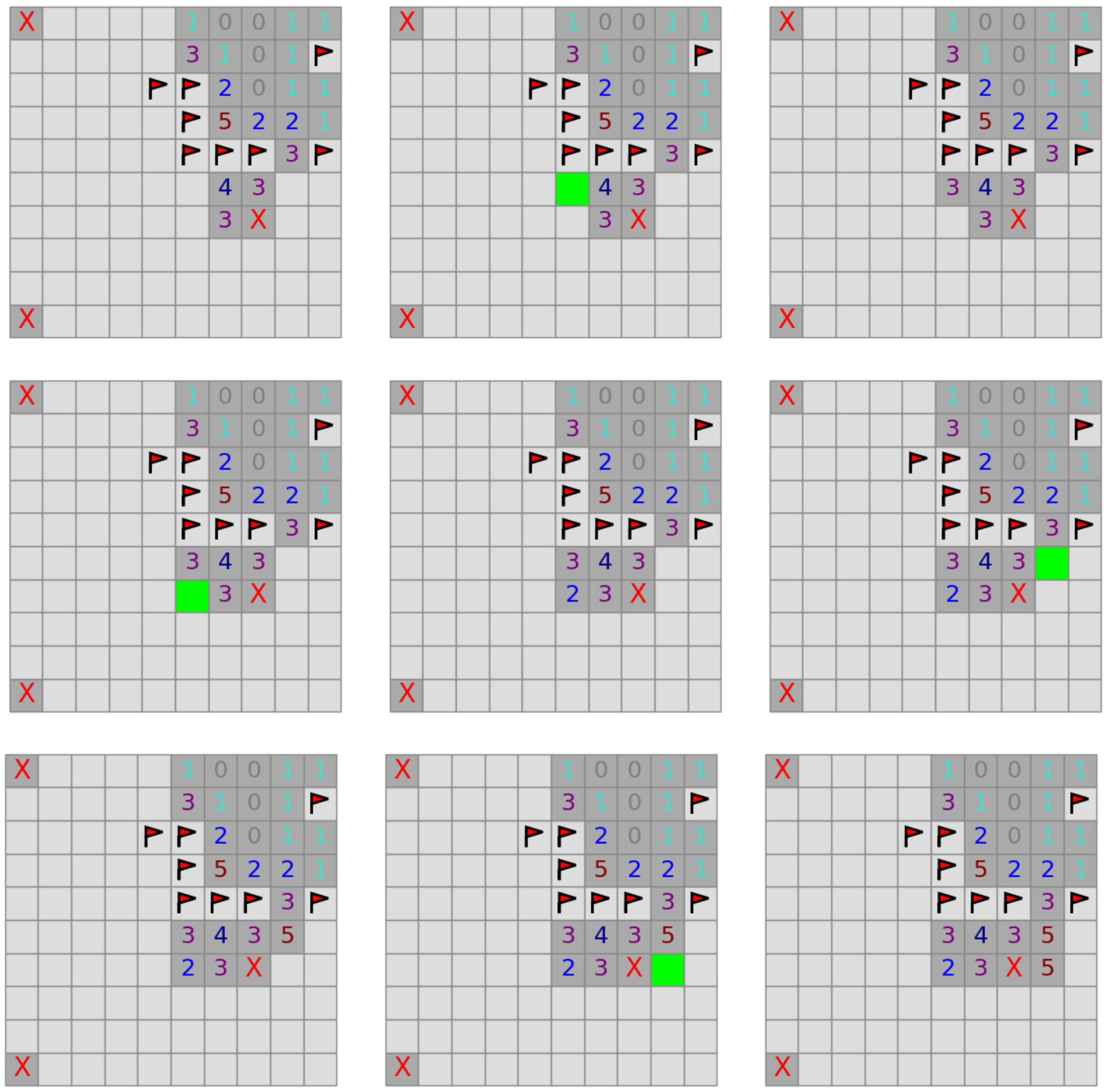

Figure 21: The Doubly Improved Agent is able to make deductions across a variety of different cells. It continues to cascade and find safe cells from newly revealed cells.

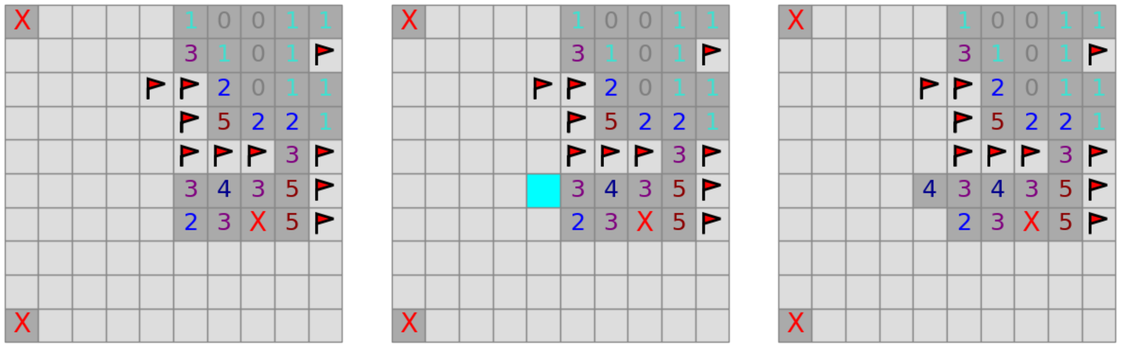
Figure 22: The Doubly Improved Agent uses probability to determine if the blue cell is safe. In this case, the agent properly deduced that the cell is safe.

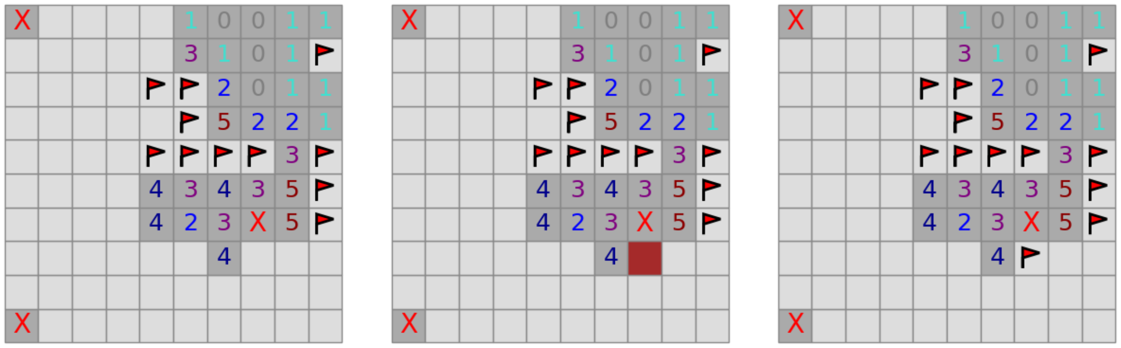
Figure 23: The Doubly Improved Agent deduces from a cell's clue that the dark red cell is a mine

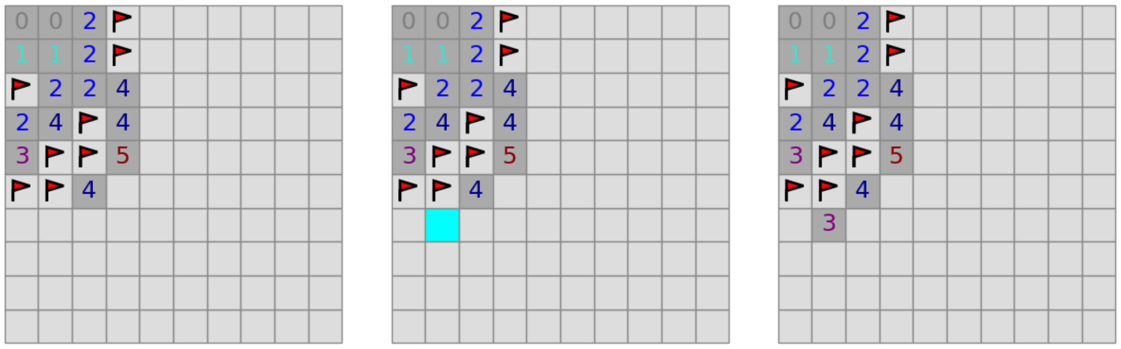
Figure 24: The Triply Improved Agent determines that this cyan cell has the lowest probability of being a mine outside the ”safe” threshold. The relevant clues imply larger probabilities for all other unknown cells (at least 1/2), while the cyan cell has a probability of 1/4 at this stage of the board.

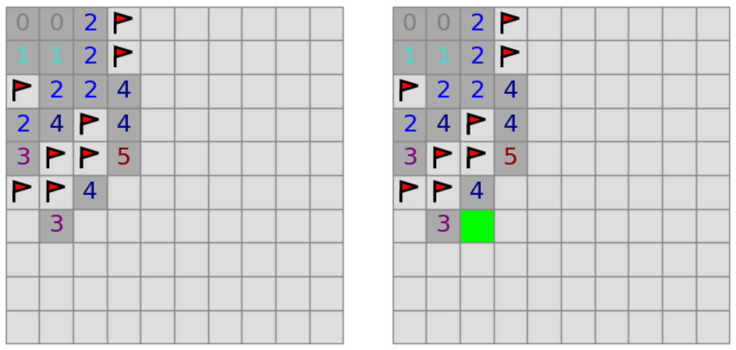

Figure 25: The Triply Improved Agent infers that this green cell is safe because out of all possible configurations for this cell, it was a mine in less than 1/8 of the valid configurations.

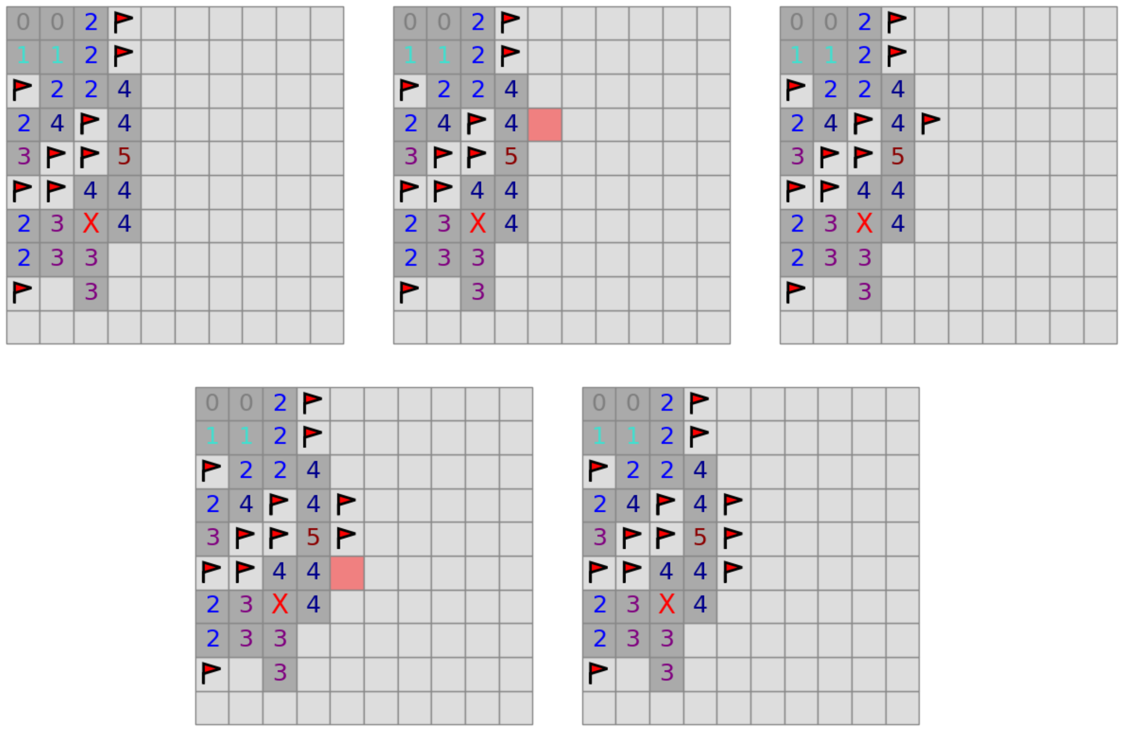

Figure 26: The Triply Improved Agent is able to deduce from the 5 clue that the rest of its neighbors are all mines, and flags them as such.

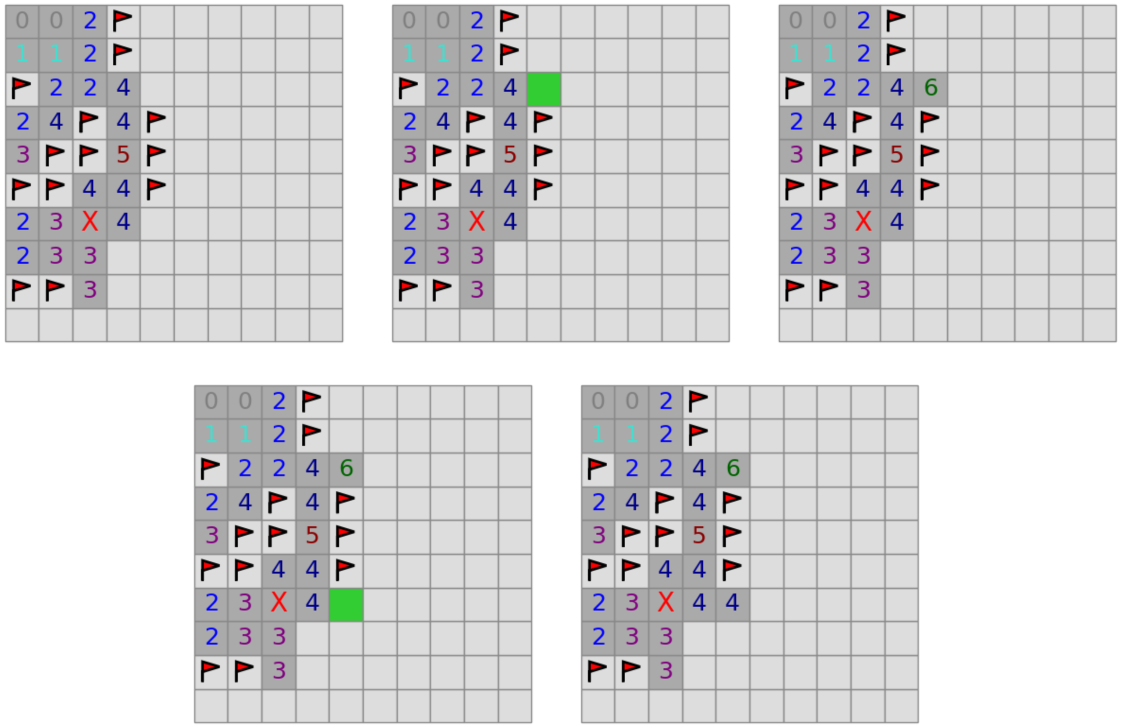

Figure 27: The Triply Improved Agent is able to deduce that the first green square is safe. The clue to the bottom left (4) is satisfied, meaning that the first green is safe. The agent then reveals that cell. The next green cell is also able to be inferred safe. The clue to the top left (4) is satisfied, with three flagged mines and one tripped mine. Note: inferred safes are revealed first; hence, why the clue of 6 does not flag the rest of its neighbors as mines.