Minesweeper

Christopher Nguyen
Vatche Kafafian

March 22, 2021

1 Overview

In this write-up, we are are going to discuss the components that allow us to build our own Minesweeper as well as the agents that we have made to play the game. These components that we will cover are the underlying representational functionalities that help us play the game, the inferences that update and store the knowledge of clues on the board as we play, the performance of our agents, and the efficiency of our program overall.

2 Representation

2.1 Displaying The Board

We displayed the board on a GUI through the utilization of Pygame and downloaded images. Behind the scenes, we first create a matrix with initialized values using a probability p that determines if a certain index in the matrix will contain a bomb or not. From this matrix that we created using a given dimension for the board, we take each piece or index in the matrix and place each piece with a corresponding image drawn on the board by translating the index to fit the screen size of the GUI. After getting the initial image of the board, we will constantly have to update the board rendering new images based off the decisions or click event actions performed by the agents. For example, the board is initially covered with filled blocks hiding the clues and bombs. The first random click will reveal one of these blocks and render a new image for this index appropriately using the predetermined cells surrounding it when first initializing the matrix. If the click is not a bomb, based on the amount of hidden bomb neighbors it has, it will render an image of a clue number displaying how many possible bomb neighbors there are to the agents playing the game. The agents themselves do not know where the location is. If the click is a bomb, we render a bomb image and no clue is given, but the game continues. As the game goes along, if the agent determines a location contains a bomb, the agent performs a right click event rendering a flag in that position marking it. Thus, our program covers all the representations based on possible actions that can be performed while playing Minesweeper.

2.2 Represent Information and Knowledge that clue cells reveal

We represent the information and knowledge that clue cells reveal through the use of dictionaries in Python. Each clue item in the dictionary has a key and a value consisting of five components. The key represents the index of the piece on the grid. The value consists of the clue, the number of safe neighbors, the number of neighbors with mines, the number of hidden neighbors, and a string value indicating if a basic strategy is applicable given the current knowledge of the position. We append a piece with these value to the dictionary once clicked on the board and remove them once they no longer have hidden neighbors. If a piece is clicked and it has a neighbor that is currently in the dictionary we append the clicked piece to the dictionary and update the clues for its neighbors. This allows us to keep a clean updated list of information on clue pieces that we are allowed to act on based on the current situation of the board at each iteration.

2.3 Inferred Relationships Between Cells

For each clue in the dictionary that cannot be determined with the basic strategy, we take its normal form and the normal form of each clicked neighbor of the clue in the dictionary. The normal form determines all the possible arrangements of a clue's hidden neighbors. For example if the clue minus the number of bombs around the clue is equal to 1 and it has two unclicked neighbors A and B, we determine the possible arrangements as [A is a bomb and B is not a bomb] or [A is not a bomb and B is a bomb]. We use this information in conjunction with other clues to logically find hidden neighbors that can be determined as safe or not safe. The information of each piece's normal form is then stored in their own list for later computing. The information is only used when the basic strategy is not applicable on the current representation of the board and if a piece has a hidden neighbor of three or less. If a piece has a hidden neighbor of three or less we compare the normal form between the original piece and each neighbor to see if our inference can be made, if not we continue the game with a random click. The logic of our inference will be discussed more in depth in the next section, while this just covers its representation.

3 Interference

For every iteration of the board, new clues are collected into the knowledge base. The knowledge base is a dictionary stored with revealed clues for clicked indexes that haven't been solved yet. This knowledge base is updated at every click or iteration on the board so that we know we have the correct information for all possible moves before executing a strategy. The clue's coordinates are the key and values that show if we can determine a coordinate's hidden neighbors as safe or not safe. If we can determine a coordinates status using the basic strategy, we mark it as "all neighbors are safe" or "all neighbors are bombs". If the basic strategy is applicable we will perform the appropriate action based on the string value of the status. If we cannot determine a coordinate's status, we mark it as "Unsure". If every coordinate in the knowledge base as a value of "Unsure", we use our advanced inference strategy. For each key in the dictionary, we set the piece object of each coordinate's normal form set. The normal form set is a logical expression that shows all the possible arrangements of a clue's hidden neighbors. This is stored in a list. We also do this for every neighbor that is a clue. Then we compare each key and its neighbor's normal form. We first do this through the use of heuristics. We have decided our heuristic to be the least occurring hidden coordinate in the normal form. For example, if one clue has inference1 = [A is a bomb and B is not a bomb] or [A is not a bomb and B is a bomb] and inference2 = [A is a bomb and B is not a bomb and C is not a bomb] or [A is not a bomb and B is a bomb and C is not a bomb] or [A is not a bomb and B is not a bomb and C is a bomb], C would be the least occurring coordinate between the two normal forms. Since this is the case, we will decide whether or not C is a bomb by comparing the two inferences. If there are any

contradictions between the two inferences, C has to be a bomb. Let's assume that C = 0 (C is not a bomb). This means that we can simplify inference2 = [A is a bomb and B is not a bomb] or [A is not a bomb and B is a bomb]. Then we compare inference1 and inference2. Since there are no contradictions between inference1 and inference2, our assumption that C is not a bomb is true. This comparison of normal forms is done for all keys in the dictionary. If we cannot find any inferences in our dictionary, we execute a random click on the board as our last possible option. We believe that our program covers all the possible inferences and strategies that can be made in most situations, however, it does not deduce everything that it can from the clues given on the board. We can improve it by using statistics in coalition with the given clues and positions of known pieces to improve the random click of the agent during the process of solving the board towards the end of its completion.

4 Decisions

Given the current state of the board and current knowledge state, we make decisions based off the index or positions of the pieces that we have stored in our boardStatus dictionary. The pieces we have stored in the dictionary are only the positions we can make a action for that have been already clicked. If we have found all the neighbors surrounding the piece we remove the piece from the dictionary since we can no longer make an action off it. In addition, if the piece is either a flag or a bomb we never put it in the dictionary because it does not provide any clues. This allows us to keep a clean dictionary filled with pieces that contain clues and which can be acted upon. This allows us to iterate through the elements in the dictionary and make a decision based off our basic strategy or inference method. When we iterate through the dictionary to make a decision each piece has a value that contains a value and string expression that tells us which strategy to apply. The values that the string can contain are "Unsure," "neighbors are bombs," and "neighbors are safe." First we iterate through the list and search all the positions if they contain the strings "neighbors are bombs" or "neighbors are safe," which indicates to implement the basic strategy. If we do not find these values and all the pieces contain a string value "unsure," we apply our inference method as our next course of action. If an inference cannot be made on any of the piece contained in boardStatus, the agent performs a random click as our last possible course of action. So in conclusion, our decision making is done by iterating through the dictionary that contains pieces, which are the only pieces that actions can be performed on in the current state of the board, and based on the string value and current status of the board we implement the appropriate strategy with the random click as our last possible option.

5 Performance

5.1 Play By Play Progression

In this section we are going to run through a play by play progression of our Minesweeper program with a dim = 9 and a p = .124. Throughout the steps we are going to comment on the decisions that our agent makes and discuss if we agree with the decision or if the decision surprised us. In addition, we are going to talk about how the agent is able to make these decisions.

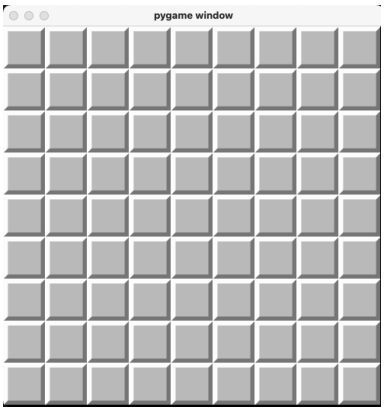

Figure 1: Initial Board

Initially, the board is covered with all hidden blocks so that the agent or user cannot see any clues or bombs as shown in figure 1 above.

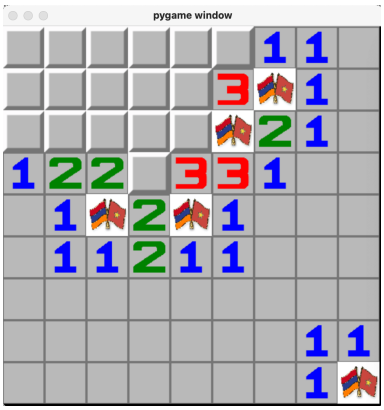

Figure 2: Step 1

After the first click on the Initial board, the first click leads to the board transforming into Figure 2. Since the first click was a white space the board expanded until all the white spaces were surrounded by clues like borders separating them from the hidden spaces. In addition, to run faster and to lower the time constraints, the agent also computes to see if any neighbors can be determined to be safe or bombs based off our basic strategy. The agent in this situation found neighbors that can be determined safe or a bomb and updated the board before next iteration saving several separate clicks into one iteration. As we see for index (4,2), (4,4), (2,5), (1,6), and (8,8), after the first click the agent evaluated the clues given in the knowledge base and updated the neighboring pieces along with it to save time. We agree with the decisions that the agent made in this situation. The only thing that surprised us is how the agent locates and updates everything almost immediately and catches clues or positions that we ourselves do not recognize until the agent performs the action.

Figure 3: Step 2

For click 2 the board transforms from step 1 to step 2. We notice that the board makes multiple updates again. First we see that the agent marks the index (2,4) with a flag. This move surprised us because we were wondering why the agent made that decision. However, after dissecting the board a little further, we see that the clue at (3,5) needed one more bomb and had one more hidden neighbor so that flag was justified. After updating that flag and looking at the other clues we see that multiple clues have neighbors that can be determined safe. For example, at index (4,3), (1,5), and (3,4), we can determine that their safe neighbors are safe after marking the flag and updating the clues to lead us to transform the board into figure 3. We see how again how the agent is able to perform multiple actions in one click derived from the knowledge base given to it, saving us time from performing more iterations for each move.

Figure 4: Step 3

Then on the third click, we see that index (2,2) is safe based off its neighboring neighbors (1,3) and (2,3) and the agent clicks it making the right decision that we would make. In addition, after marking (2,2) safe the agent went even further and surprised us by updating its knowledge base immediately after placing the one. Because of this immediate update, the agent is able to determine another update that it can make off the first click and marks index (2,1) as a flag based off of index (3,2) now only having one more hidden neighbor and one unidentified mine, performing two updates off one click.

Figure 5: Step 4

Then on the fourth click we know that index (1,1) and (2,0) are safe due to the clues given by index (2,2) and (3,0). After updating the index (1,1) and (2,0) as safe the agent updates the surrounding clues and is able to make another decision to mark (1,0) as safe due to the clue given at (1,1) after marking it as safe. This allows us transform our step 3 board into step four.

Figure 6: Step 5

Finally, for the last click, the agent was able to determine index (0,1) as a bomb do to the clue given by the piece at (0,2). After updating that clue for its neighboring pieces and marking it as flag. The agent was able to also determine the last position as safe due to the clues given by the pieces at (1,0) and (1,1), thus completing the board.

Throughout this play by play the agent operated and made decisions appropriately and did not make any decisions that we did not agree with. If anything the agent met expectations and performed beyond them by undergoing multiple updates throughout a single iteration by constantly tapping into its knowledge base and immediately updating it, allowing for more decisions to be made instantly. In the next section, we will evaluate how our algorithm performs against the simple baseline algorithm.

5.2 Improved Agent V.S. Basic Agent

In this section, we will evaluate the difference in overall success between our basic agent and our improved agent.

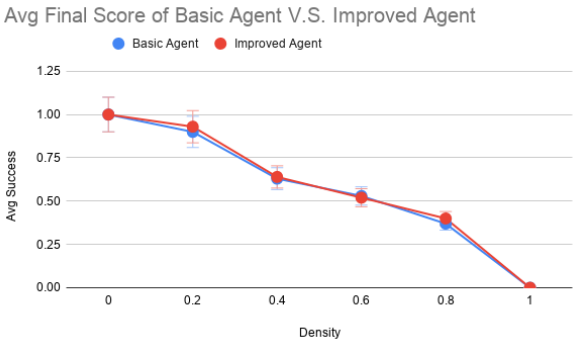

Figure 7: Basic Agent V.S. Improved Agent

As shown above, we have a plot that shows the Average final score of the basic agent versus our improves agent. We compared them on a fixed board with the dim = 9. The avg final score of each density was computed by taking the average of ten runs for each density where we took the number of successfully marked bombs divided by the total number of bombs. The graph makes sense and agrees with our intuition after comparing the two algorithms. There is only a very minor improvement with our algorithm because our algorithm builds upon the basic strategy by attempting to find inferences from multiple clues. In order for there to be a significant difference, the agent has to have the opportunity to act on scenarios that yield the possibility to come up with inferences to build or create normal forms for the pieces. The minesweeper becomes hard when the bomb density of the board increases which means we have to make more guesses. It is hard because there is a lack of clues, but more bombs. This leads to our agent making more random clicks than applying any strategy, which leads the agent to hit more bombs. Now in comparison to the basic agent the improved agent beats the simple algorithm whenever there is the possibility of making inferences based on multiple surrounding clues. The basic algorithm has to guess, while our improved agent is able make inferences and come up with a approach to either mark a bomb or determine a position as safe. The simple algorithm is not better than our algorithm because ours is built on the basic algorithm but it also has the added feature of solving multi-clue inferences. Our algorithm is able to work things out that the basic agent cannot whenever there are multiple clues by using normal form comparisons on surrounding neighbors, thus improving the amount of options or strategies when it comes to solving the game. Although the improved agent may not make these inferences as frequently as we would like it to, it still offers a greater approach or better percentage of success in comparison to the basic agent.

6 Efficiency

While implementing this program, we ran into both space and time constraints. These issues are mainly due to how we implemented the program. One space constraint we run into when implementing this program includes the size of the dictionary. Sometimes there are clues in the dictionary that cannot utilized in the current situation of the board and at some iterations they can be considered as wasted space because they do not help with the current state of the board. However, we do not get rid of these clues because these clues can have the potential to become useful as the board opens up. So to improve upon this issue, we can make a data set with situations, where a clue would not be considered helpful on the current state on the board and move it to a waitlist. While we move the other pieces with helpful clues into a smaller list speeding our process of iterating through the dictionary to perform an action. Another problem arises when implementing our inference based strategy. We loop through the entire dictionary risking the possibility of wasting time by generating normal forms for clues that will not produce an inference. This can possibly be improved by finding a way to produce normal forms for neighbors that have a high likelihood of producing an inference. This would probably have to be done by coming up with specific scenarios that can produce inferences and then checking if the clues we are given match those scenarios. Overall, the major issues that we deal with are implementation based and we can mainly improve on them by creating data sets of certain scenarios to further truncate our query and speed the process of our inference based strategy as well as basic strategy to further the efficiency of our agent overall.

7 Clever Acronym

The clever acronym that we came up with is S.M.I.L.E., which stands for sweeping multi-inference logging excavator. We named it this because we are sweeping through a board of mines using inferences that involve a combination of multiple clues that are logged into a dictionary. We then use the inferences to excavate the board for mines to avoid. Thus, allowing us to S.M.I.L.E. when avoiding the bombs and finding safe neighbors.