American University of Armenia

Zaven & Sonia Akian College of Science and Engineering

Applying Constraint Satisfaction Problem and Reinforcement Learning on Minesweeper

Aram Leblebjian, Joseph Alakach

Supervisor: Monika Stepanyan

Spring 2023

Abstract

In this project, we study the effectiveness and the differences between two popular AI techniques, Constraint Satisfaction Problem (CSP) and Reinforcement Learning (RL), and applied both in solving the Minesweeper game. Our study aims to combine the CSP and RL algorithms and evaluate their ability to find the solution to a minesweeper game. Our results indicate that while RL can achieve satisfactory performance after long training, it is still inferior compared to CSP in solving the minesweeper. CSP is able to solve a configuration in a much shorter time, with fewer resources for implementation, and with a much smaller search space compared to RL. However, adding RL to solve a configuration when CSP gets stuck results in a much more powerful AI for minesweeper because of RL's probabilistic reasoning and uncertain decision-making. Our study contributes to the growing body of literature on the effectiveness of two different AI techniques and provides insights into their strengths and limitations.

Keywords: AI, Artificial Intelligence, Machine Learning, CSP, Constraint Satisfaction problem, RL, Reinforcement Learning

Acknowledgements

We would like to express our deepest gratitude to our supervisor, Monika Stepanyan, for her invaluable guidance and support throughout this project. Her insight and feedback have been invaluable to the success of this research. We also want to thank her for providing us with the necessary resources and tools to conduct our research effectively. We are grateful for her patience and encouragement, ensuring our work meets the highest standards.

Contents

1 Introduction

Minesweeper is a puzzle video game. In the game, mines are randomly scattered around the board, divided into square cells, also called tiles (Figure 1.1). Cells have three states: unexplored, explored, and flagged. Flagged cells are those marked by the player to indicate a potential mine location. At the start of the game, all the cells are unexplored. If a player explores a mined cell, the game ends, and the player loses. If a player explores a safe cell, it displays a number from 0 to 8, indicating the number of mines that surround it diagonally and adjacent to it. A blank cell can be numbered as 0, and all its neighboring cells will automatically be explored. Thus, a cell displaying the number 0 means no bombs are surrounding that cell. Also, a cell displaying the number 8 means eight bombs are surrounding that cell. The game's goal is to explore all safe cells without exploring a mined cell. Before the game starts, the player knows how many mines there are (the mine count), and the first cell to be explored by the player will never be a mine and will always be a safe cell.

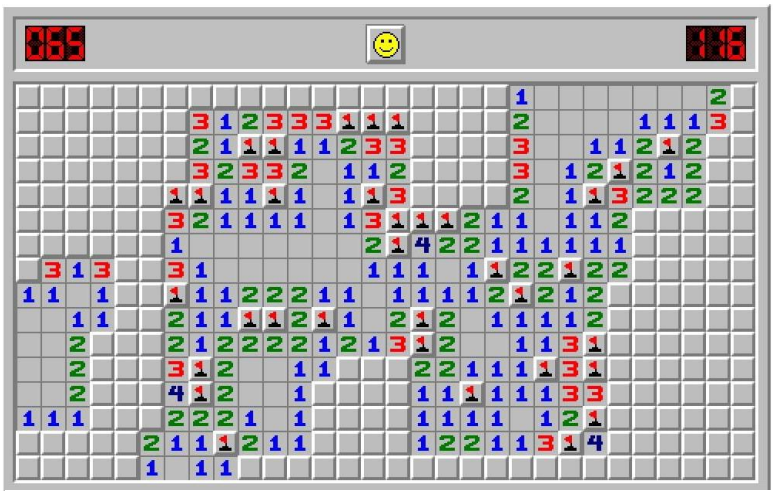

Figure 1.1

The history of minesweeper dates back to 1985, when Conway, Hong, and Smith developed a game called "relentless logic" (Figure 1.2). This game was the origin of the idea of the minesweeper. This game had a similar board with squares on it, and a player was located in

the top left corner cell, whose goal was to get to the bottom right cell without stepping on a mine. The player only knew the number of mines adjacent to their location. Using the same idea, in 1989, minesweeper was developed by Donner and was released by Windows (History of Minesweeper, n.d.).

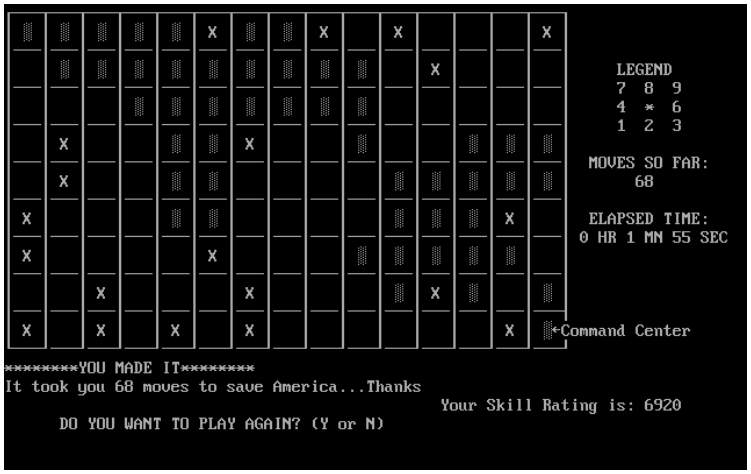

Figure 1.2

So now we are dealing with the problem of solving the game and reaching the goal state without exploring any wrong cells. We are interested in solving this problem because minesweeper can be represented as constraint satisfaction, search, or even logic problem related to the topics covered in Artificial Intelligence. Moreover, the minesweeper can be solved by applying different Reinforcement Learning techniques and algorithms.

Solving minesweeper using Constraint Satisfaction Problem or Reinforcement Learning strategies is an important problem in Artificial Intelligence research. Minesweeper has been widely used as a benchmark problem in AI research to evaluate the effectiveness of different algorithms. The skills and techniques required to solve minesweeper can be applied to other real-world problems involving uncertain decision-making. Using CSP with RL and understanding their strengths and weaknesses can help us to identify the most appropriate algorithm for a given problem and optimize its performance. Moreover, this can also contribute to fundamental research in AI by advancing our understanding of the principles of constraint satisfaction, probabilistic reasoning, and decision-making.

2 Background Information

Before diving deeper into our literature review and methodology, it is essential to provide some background information on the key concepts and algorithms used in the paper, such as CSP, backtracking, and reinforcement learning.

2.1 Constraint Satisfaction Problems (CSP)

In a Constraint Satisfaction Problem, a set of variables must be assigned values that satisfy a set of constraints, where each variable has a domain of possible values, and each constraint specifies the allowed combinations of values for a set of variables. The goal is to find a solution with consistent and complete assignments that assigns values to all variables to satisfy all constraints. A consistent assignment assigns the variables to values from their domain that do not violate any constraints. A complete assignment is where every variable is assigned. An example can be Figure 2.1, where each neighboring state of US must be assigned to a different color, and this can be done through CSP. One CSP-solving technique is the backtracking algorithm.

Figure 2.1

2.2 Backtracking

Backtracking is an algorithm used to solve problems by incrementally building a solution and backtracking to earlier steps when a solution is found invalid or impossible. It is commonly used in constraint satisfaction problems, where the goal is to find a solution that satisfies a set of constraints. Backtracking chooses a value for one variable at a time and backtracks when some other variable has no other values left to assign. The backtracking algorithm explores all possible solutions by constructing partial solutions and testing each one against the constraints. If a partial solution violates any constraint, the algorithm tries out another value or backtracks to the previous step if there are no more values to consider at that stage. The backtracking algorithm continues until a valid solution is found or all possible solutions have been explored. In Figure 2.2, the state in the red circle shows that assigning blue to that cell results in the country shown by the yellow arrow having no more color in its domain. That's why we backtrack and change the blue to orange.

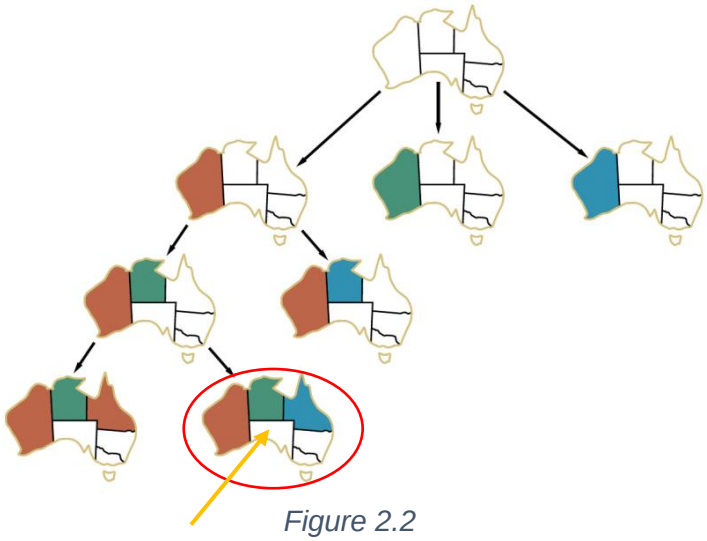

Figure 2.2

2.3 Reinforcement Learning

Reinforcement learning (RL) is a subfield of machine learning that focuses on training agents to learn to make decisions based on feedback from their environment. RL aims to learn a policy that maps states to actions such that the agent maximizes a cumulative reward signal over time (comi, 2018). The RL agent interacts with the environment by taking actions and observing the resulting

state transitions and rewards. The agent aims to learn a policy that maximizes the expected cumulative reward over time (comi, 2018).

The reward is a crucial component of RL, as it serves as the feedback mechanism that guides the agent's learning. After each action, the agent receives either a positive, negative, or neutral reward from the environment, indicating how good or bad the action was. The agent's goal is to maximize the sum of rewards of one epoch (one game) over time, which requires it to balance the effect of each action with long-term consequences (comi, 2018).

An example of RL can be the Mario game (Figure 2.3), where the agent (Mario) will learn to complete tasks, such as achieving a milestone or avoiding obstacles through rewards. The agent has the ability to perform four actions in each state of the game. In this example, the agent will receive a positive reward when getting a coin, avoiding the Goombas (the enemy mushrooms), a big positive reward when winning the game, a negative reward when being hit by a Goomba, and a big negative reward when it loses the game.

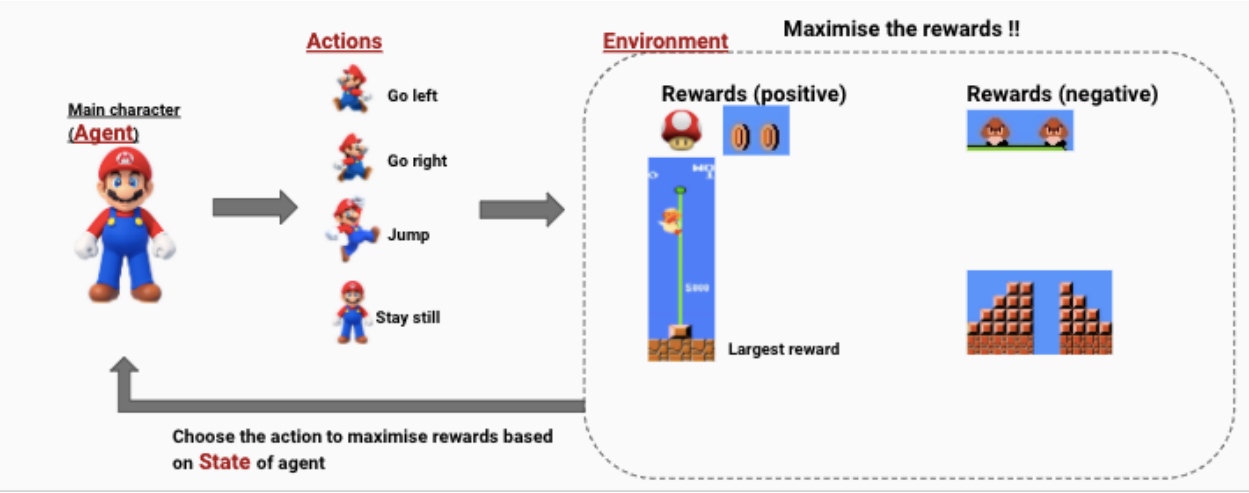

Figure 2.3

2.4 Q-Learning

Q-learning is a model-free reinforcement learning algorithm that learns the optimal policy for a given problem through trial and error. The algorithm estimates the expected reward for each state-action pair in a given environment and updates the values based on the observed rewards. It trains the function to learn which actions are better in each state and selects the optimal ones. The algorithm is particularly useful in problems with discrete action and state spaces, where the possible actions and states are limited. It represents the state of the problem as the rows of the Q-table and the columns as the actions the agent can take for each state (Figure 2.4) . When an agent starts interacting with the environment, the reward of taking a particular action for a particular state would be computed or updated based on the performance of that action in that state (Luu, 2023). Q-learning can converge to the optimal policy but may require many iterations, especially in large state spaces.

Q-Table		Actions					
		South (0)	North (1)	East (2)	West (3)	Pickup (4)	Dropoff (5)
States	0	0	0	0	0	0	0
	.	.	.	.	.	.	.
	.	.	.	.	.	.	.
	.	.	.	.	.	.	.
	328	-2.30108105	-1.97092096	-2.30357004	-2.20591839	-10.3607344	-8.5583017
	.	.	.	.	.	.	.
	.	.	.	.	.	.	.
	.	.	.	.	.	.	.
	499	9.96984239	4.02706992	12.96022777	29	3.32877873	3.38230603

Figure 2.4

The Bellman equation is used to update the Q-values by finding the relationship between the Q-values of each state and action pair and the Q-values of the actions and states that the agent has gone through. This enables the agent to estimate the Q-values of each action-state pair by considering both the rewards received in the past and the estimated rewards for acting in the future (Torres.AI, 2020).

Deep Q-learning is a variation of Q-learning that uses deep neural networks to estimate the Q-values of state-action pairs (Figure 2.5). The algorithm is based on a deep neural network that takes the current state of the environment as input and outputs the estimated Q-values for each possible action. The neural network is trained using a combination of supervised learning and reinforcement learning, where the targets for the Q-values are generated using the Bellman equation (Luu, 2023). Deep Q-learning is effective in solving problems with large state spaces, where a high-dimensional image represents the state.

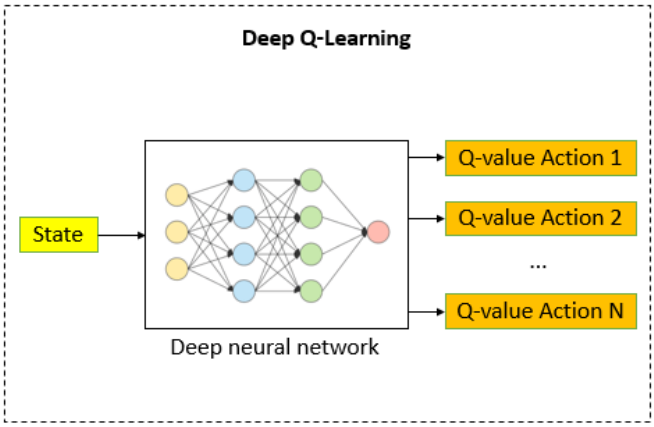

Figure 2.5

In case of minesweeper, Q-learning with Q-table is not feasible as Q-learning can be more suitable if the state space is small, because it can converge to the optimal policy faster than Deep Q-learning. However, as the minesweeper's state space is large and represented by a high-dimensional image, Deep Q-learning is more appropriate as it can handle high-dimensional input and can learn the optimal policy more efficiently. Moreover, Deep Q-learning has the potential to learn more complex patterns and features from the game board, which can result in better performance than Q-learning. The choice between Q-learning and Deep Q-learning would ultimately depend on the size and complexity of the state space, as well as the desired level of performance and computational resources available.

3 Literature Review

Minesweeper is a well-known problem in artificial intelligence, and much literature review has already been done on the topic. Numerous studies have explored different approaches to solving minesweeper, including constraint satisfaction problem (CSP), reinforcement learning (RL), logical algorithms, and heuristic search. These studies have examined the effectiveness and efficiency of different algorithms and their performance under various conditions, such as different board sizes and numbers of mines. Additionally, studies have investigated the cognitive processes involved in Minesweeper solving, such as pattern recognition, spatial reasoning, and decision-making under uncertainty. Some of the important research that has been helpful for us in studying the problem, identifying gaps and limitations in them, and addressing them will be explained now.

3.1 Logical Solvers

3.1.1 Logic & CSP

The game's goal is to uncover all the cells on the board that do not contain a bomb and flag all the cells that indicate a danger or a bomb. As there may be cases where even the algorithm or the human solver needs a guess to continue with the board, a particular board's correct solution is not guaranteed. A Stanford University study suggests two ways of modeling AI algorithms that solve the game; one using a set of logic rules and the other through a constraint satisfaction problem (Stanford University, nd).

1- Logic Rules.

According to Stanford University's solution, the set of logical rules to solve the minesweeper is defined as this:

- A cell is either explored, unexplored, or flagged.
- There are two types of cells: explored and unexplored.
- If a cell is explored, it has a numerical value indicating the number of mines surrounding it.
- If a cell is explored and the number of its unexplored neighbors equals its value, mark all its unexplored neighbors as mine. The explanation is that all the other explored ones don't have any bombs, indicating that the unexplored ones are bombs (Stanford University, nd).
- If the number of flags neighboring a particular explored cell equals its value, then explore its unexplored neighboring cells because that cell already has all its flags assigned (Stanford University, nd).
- If an explored cell has the value of 0, then explore its neighboring unexplored cell.

2- Backtracking Methodology.

The backtracking algorithm to find valid arrangements on the board is as follows:

- If a cell is mine in every arrangement with its covered neighbors, flag it, meaning assign its value to 1.
- If a cell is not a mine in every arrangement with its covered neighbors, it is safe, and you can cover it, assigning the value to 0.
- In an arrangement where neither of the first two rules defined above can be selected, the probability of a cell having mine is equal to the probability of it being safe. Therefore,

move forward and backtrack to define it after getting more information from the cells surrounding that cell (Stanford University, nd).

3- CSP Approach.

A solution through CSP defines the variables as the covered cells, each with domain 0 and 1, where 0 corresponds to a safe cell where you can explore it, and 1 corresponds to a mine.

The constraints of the CSP will be:

- The sum of all the variables, in the end, must be equal to the number of mines, as a variable can either take 0 (safe) or 1 (bomb). Therefore, adding all the ones will result in the total number of mines on that board. The board can't be covered by more or fewer bombs than the actual amount that the board has (Stanford University, nd).
- The sum of the variables neighboring an uncovered cell must be equal to the number in that cell, in the end, representing its mines (Stanford University, nd).

The results shown in Figures 3.1 were not as good as expected. Here, the timeout percentage is the percentage of the time taken to solve a configuration in the total amount of time dedicated to solving that problem. In general, if one attempts running logical and CSP approaches separately, the higher the difficulty of the configuration, the lower the win percentage; each technique alone is facing issues pushing it to failure.

Beginner: (9x9, 10 mines)	Win Percentage	Timeout Percentage	Average Percent Uncovered	Number of Trials
Logic AI	69%	0%	83%	1000
CSP AI	77%	0%	99%	30
Intermediate: (16x16, 40 mines)	Win Percentage	Timeout Percentage	Average Percent Uncovered	Number of Trials
Logic AI	37%	0%	65%	1000
CSP AI	60%	60%	31%	10
Advanced: (24x24, 99 mines)	Win Percentage	Timeout Percentage	Average Percent Uncovered	Number of Trials
Logic AI	9%	0%	49%	100
CSP AI	0%	70%	7%	10

Figure 3.1

Moreover, there also are cases where logical algorithms can fail. Here is an example (Figure 3.2). Here, only two mines left for the board to be solved. It is easy for a human that knows the game to know that the mines would be the two middle squares. However, the logic wouldn't be able to get that solution. That is because each number shown on the explored square has one more unexplored square than its value. Therefore, in this case, the logical agent will only have to guess the solution.

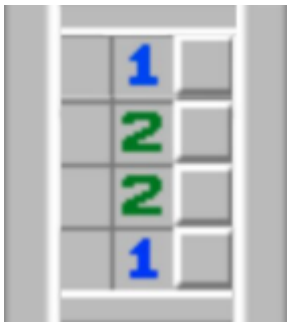

Figure 3.2

Stanford University's study met with the greatest success when they combined both approaches into one solver. This algorithm applies logic if it can certainly find safe squares. Only when logic can't find a solution does the CSP step in. Therefore, the combined solver is as fast as logical rules and as accurate as CSP, achieving high accuracy (Figure 3.3). Also, the timeout percentage is almost zero in most configurations because, in the combined approach, CSP will

only step in when logic fails and will not try to solve the whole configuration alone, which takes more time.

Beginner: (9x9, 10 mines)	Win Percentage	Timeout Percentage	Average Percent Uncovered	Number of Trials
Logic AI	69%	0%	83%	1000
CSP AI	77%	0%	99%	30
Combination AI	95%	0%	97%	1000
Intermediate: (16x16, 40 mines)	**Win Percentage**	**Timeout Percentage**	**Average Percent Uncovered**	**Number of Trials**
Logic AI	37%	0%	65%	1000
CSP AI	60%	60%	31%	10
Combination AI	84%	4%	81%	50
Advanced: (24x24, 99 mines)	**Win Percentage**	**Timeout Percentage**	**Average Percent Uncovered**	**Number of Trials**
Logic AI	9%	0%	49%	100
CSP AI	0%	70%	7%	10
Combination AI	70%	0%	90%	30

Figure 3.3

However, there remains a problem in their "Average Percent Uncovered," where we can see that, on average, the map is covered around 90% before losing, as their solver is solving the board starting from easy choices to hard ones till it encounters a dead end. Here, our approach will have a specific type of solver that will have to be used only for end games, where even the hard configurations will be solved, and this will be done by applying our reinforcement learning method to it. Reinforcement Learning will summarize the end choices for that configuration leading to a win with identifying patterns between the cells.

3.1.2 CSP Using GAC, 2-RC, & 3-RC.

The constraint system laboratory of the University of Nebraska-Lincoln studied a more interesting constraint-based approach for solving the minesweeper. Again, their approach was that every square (variable) on the board would be marked safe or mine (1 or 0). Every safe variable has a value on it (sum_variable) that represents the number of mines in its eight surrounding squares. In Figure 3.4, in the case of C1, exactly 2 out of {A, B, C, D, E, I, J} will be mines, and in the case of C2, exactly 3 out of {D, E, F, G, H, I, J} will be mines.

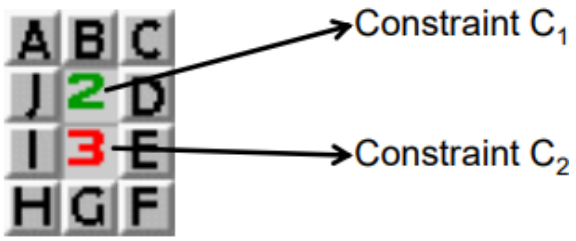

Figure 3.4

To solve a configuration, they defined three levels of consistency between the variables called in increasing order of complexity: GAC, 2-RC, and 3-RC. GAC (generalized arc consistency) ensures the consistency of every constraint, meaning it moves around the board using only one variable's guaranteed surrounding flags and safe squares without comparing its needs with any other explored variables (Bayer, 2006). On the other hand, 2-RC (2-relational consistency) ensures the consistency of every combination of 2 constraints. Likewise, the 3-RC ensures the consistency of every combination of 3 constraints. As the level of consistency rises, the cost will increase, but we will get a more concrete and accurate solution to the problem. They always applied the algorithms in the increasing order of their complexity (Bayer, 2006).

Figure 3.5 is a random configuration where they started the game.

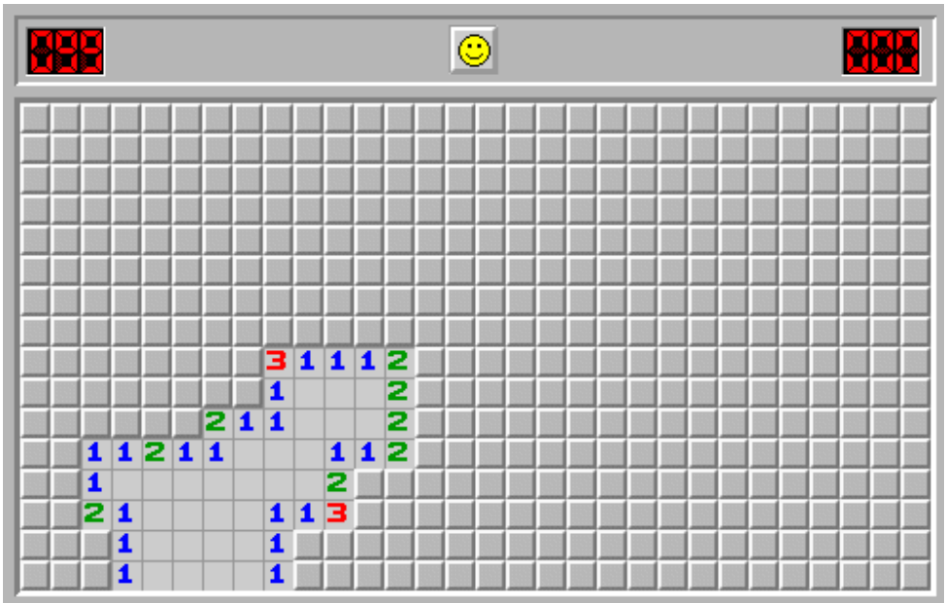

Figure 3.5

Applying the GAC on it will result in Figure 3.6, where no GAC is left for the configuration to continue.

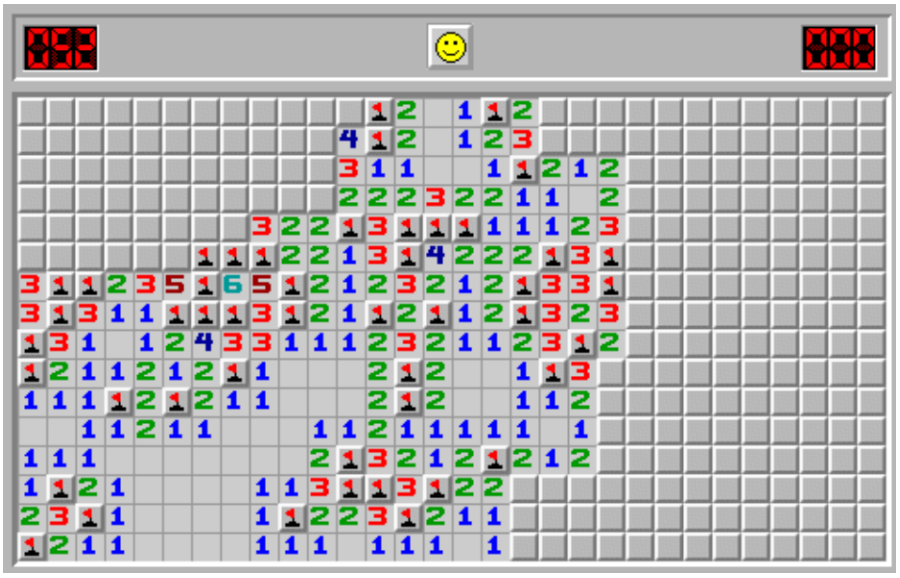

Figure 3.6

Therefore, pass to the 2-RC algorithm, resulting in Figure 3.7.

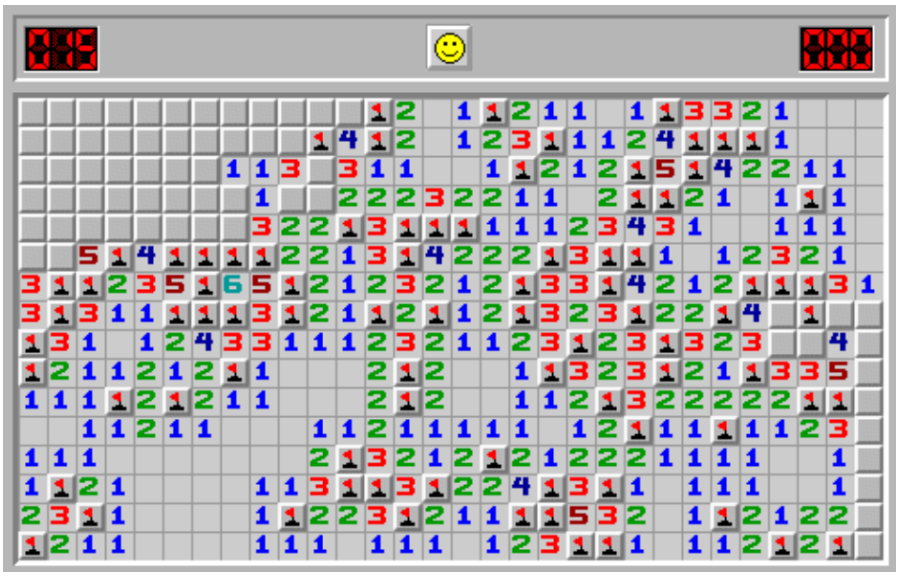

Figure 3.7

And finally, as solving every combination of two constraints is over, we move on to 3-RC, resulting in Figure 3.8.

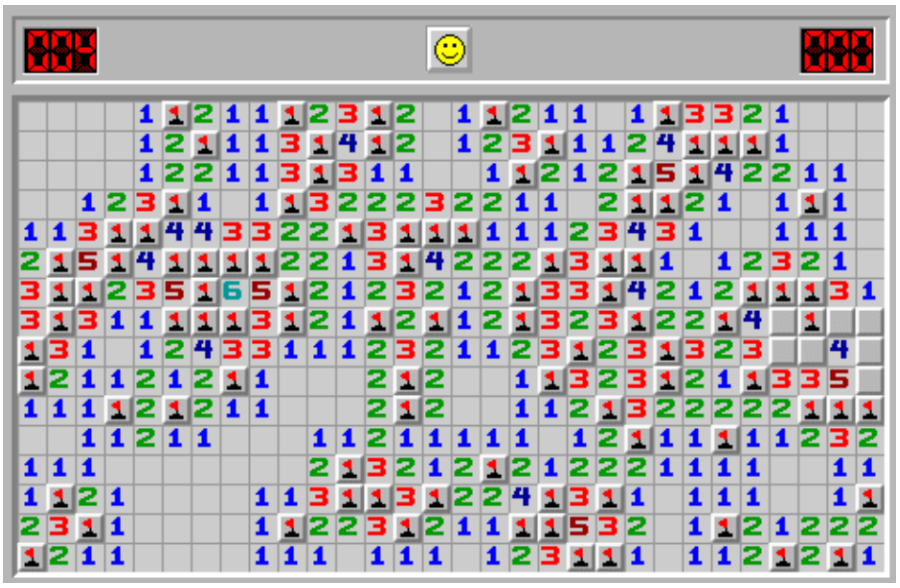

Figure 3.8

And this cannot be continued using 3-RC. Therefore, either one can use 4-RC, which will take far more time for the algorithm to solve or can take a guess and continue.

Moreover, Figure 3.9 is an example showing what levels of consistency can define a square, whether it is a flag or safe. Blue is used for GAC, green for 2-RC, and yellow for 3-RC. Here, you can decide the fate of the blue cells by using one variable (cell). Greens can be decided by ensuring the consistency of two constraints. Blue can be decided by ensuring the consistency of three constraints.

Figure 3.9

Figure 3.10 is an example where GAC cannot continue to decide between solving the configuration, and 2-RC must step in.

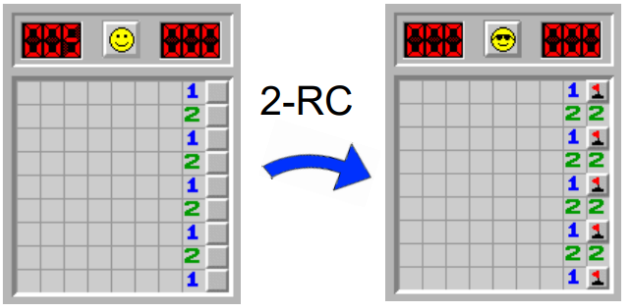

Figure 3.10

Also, figure 3.11 is a case where the consistency of 2 constraints would not be enough, and 3-RC would step in.

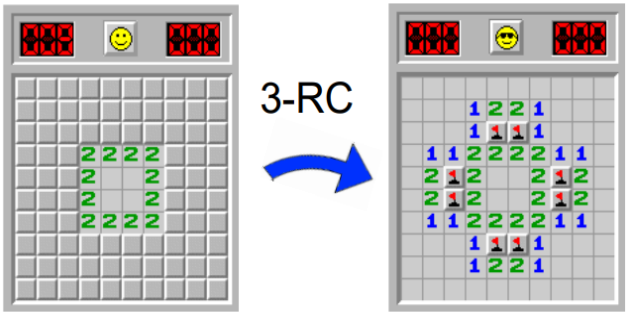

Figure 3.11

Although not stopping with 3-RC and continuing with 4-RC, 5-RC may result in the end solving the whole configuration, in some bad Minesweeper games, there may be cases where you can't find the guaranteed correct solution and need to take one of the cells with 0.5 probability (Figure 3.12).

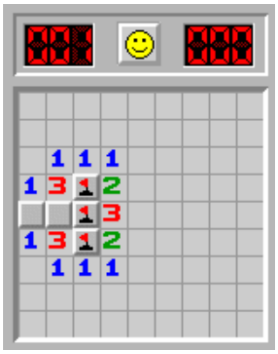

Figure 3.12

Although this approach works quite well, as they're saving the constraint, it consumes both more memory and time when comparing the

constraints of two or more cells together. Our approach will also solve this limitation, which will be explained in our methodology.

3.1.3 Modified CSP

Finally, another study by some students discusses their steps in solving the problem using CSP. According to them, using regular CSP is impossible to get to a valid solution, as in the case of the complete and consistent assignment for a regular CSP, multiple assignments may exist to get to a goal state (Gil & Chai, nd.). But in minesweeper, each configuration can have only one correct solution, as the mine locations are predefined and do not change as we progress. Therefore, they can't take a step as always to check if it is false and backtrack. We must check if taking a step may result in a loss without actually taking the step.

Moreover, when a square is explored, assuming it is not a mine, it will bring forward a set of new constraints. Therefore, the new CSP algorithm must have the ability to have new constraints and erase the satisfied ones (to search through the unsatisfied ones faster), and that will lead to having more than one game state, which will look like a different CSP (the variables and the constraints will change in some subset of moves). In other words, by satisfying the existing constraints, we can move from one CSP to another, and then add the newly generated constraints and continue solving with those new ones (Gil, & Chai, nd.).

As defined previously in other cases, the variables will be the unexplored and unflagged squares adjacent to explored squares. The domain will be {0, 1}, where 1 represents a mine, and 0 represents a safe square. The constraints are for the explored squares that still have unexplored squares in their surroundings.

An example that would explain this can be shown in Figure 3.13. This will be represented as:

X = {a, b, c, d, e}

D = {0,1}

C = {(a+b=2),(a+b+c+d+e=3),(d+e=1),(e=1)}

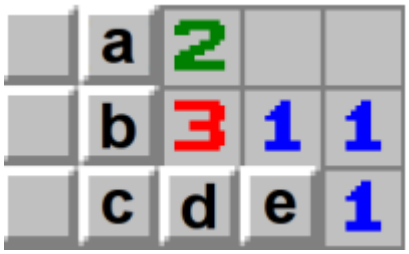

Figure 3.13

As a+b=2, then both are 1s (a=1, b=1) and must be flagged, then e=1 mine at e, and that can infer to d+e=1 ➔ d=0 no mine at d, then to the second constraint 1+1+c+0+1=3 ➔ c=0 no mine at c. This part of the CSP is solved, and when c and d are explored, we will get a set of new constraints from them, and they will be our set of variables. This solution will be part of our solution modified a bit to satisfy our needs in terms of time and accuracy.

3.2 Reinforcement Learning

3.2.1 Deep Learning Model

A study was conducted at Loughborough University by Muhammad Hamza Sajjad from the Computer Science Department about neural network learners for minesweeper. He used a pre-built deep learning library called Deeplearning4j, where he used a normal neural network for easy configurations and trained convolutional neural networks for more difficult configurations (Sajjad, 2022). Initially, he started inputting the whole board into the model, where all the cells on the board are already explored (revealed or flagged). However, this is not reinforcement learning, as their classifier would learn to classify whether each cell on the board could be mine or could be explored using the patterns it learned from the training, not by playing the game (like RL).

However, as he advanced in research, he found that even though representing this problem as a classification problem is not wrong, there were much better results when he identified the problem as a regression problem, where each unexplored cell has a continuous value between 0 and 1, indicating the confidence percentage of the prediction.

Moreover, inputting the whole board to the model, say n*m board, would result in the model learning and solving only for that board (Sajjad, 2022). Therefore, to create one model and generalize that model for every other difficulty and board size, another approach was suggested. A window will be created, say a 5x5 window, over the whole board, and each time that board will move on the board, updating the results of each cell inside the window (Sajjad, 2022). For example, for a 6x6 board size, the input of the neural network will be four 5x5 windows of the board, and each cell's values will be updated more than once. This generalizes their model for any board size and difficulty. However, he had to train the model for different board sizes so that even if the model could be applied to different board sizes, it would be useful and solve the more difficult configurations (Sajjad, 2022).

He trained the model for 240,000 epochs, which takes a lot of resources, and his win percentage during the training was around 45 percent. However, he experimented and added more layers, in the end, getting a 91%-win rate for beginner, 72% for intermediate, and 21% for expert (Sajjad, 2022).

Our model is a Reinforcement Learning model and not a Deep Learning one. Therefore, by less training and computational power, we tried to produce some good results. (Although, again, it took a lot for us to train).

3.2.2 State Spaces of problem formulation.

Another study was conducted by Anav Mehta from Cupertino high school, CA, USA, where he studied applying different reinforcement learning algorithms for problems that can be solved by Constraint satisfaction algorithms. In the case of minesweeper, a particularly interesting approach was his state representation approach, where he had three different state representations of the board's cells with its surrounding neighboring eight cells.

The first representation's state space size is N*M*10, where N is board’s number of Rows and M the number of columns, where each cell has ten fields to fill, where the first eight fields will have either 0 or 1, indicating a mine in the cell's corresponding neighboring cell, the ninth field indicates no mines in the neighbors, and the tenth field defines the cell is unknown (Mehta, 2019).

The second representation's state space size is N*M*2, where each cell has two fields. The first field takes values between 0 and 8, indicating the cell's number of surrounding flags, and the second field represents whether the cell is unexplored or flagged by 0 and 1, respectively (Mehta, 2019).

Finally, the third representation is via image, and the state is N*M*1, where each cell's corresponding field's 0 to 8 values represent the number of mines surrounding it, and the value 9 is the unknown representer (Mehta, 2019).

He chose the second representation after training on all three as the results were better. The reward system is (N*M) for a win, (-N*M) for loss, progress (+1), and nonprogress (-0.5), and this was exactly our representation of the rewards in the beginning before we started training and realized that training the game on the whole board will require large computation power and training for long times. That's why we changed the state space and represented it as the third state

representation (the simplest one) where we care about a cell's surrounding mines, not their particular locations.

4 Methodology

As we came across different ideas and approaches during our literature review, we identified some gaps and limitations to the discussed existing research and worked on filling these gaps. We used some of the concepts from the approaches discussed in literature review, combined them, modified them, and came up with our method to solve different configurations of the minesweeper. We combined two different implementations of minesweeper solvers to get one ultimate and unique solver. One uses problem-solving techniques, including CSP, backtracking, and logic, and the other is a reinforcement learning model, which uses techniques of pattern recognition and decision-making under uncertainty.

4.1 Constraint Satisfaction Problem Formulation

In our implementation of CSP, as already explained, a cell is either explored, unexplored, or flagged. We have three methods of exploring or flagging a cell. Our variables are the unexplored cells on the board that have at least one explored neighboring cell. Their domains are either assigning it to a flag or exploring it. Each cell has different constraints in the three methods, explaining how each variable can be assigned a value.

4.1.1 First Method

The first method is similar to the GAC from section 3.1.2 but modified with the logic from section 3.1.1. The two constraints of exploring or flagging an unexplored cell are as follows :

- If a cell is explored and the number of its unexplored neighbors equals its value, mark all its unexplored neighbors as mine.
- If the number of flags neighboring a particular explored cell equals its value, then explore its unexplored neighboring cells because that cell already took all its flags.

As the first explored cell in any configuration cannot be a flag, the algorithm automatically explores the left upper cell of the board. Then, this first algorithm moves around the board, tries to find an explored cell with some unexplored neighbors, and applies the above rules to that cell. If an unexplored cell is either explored or flagged, as discussed in section 3.1.3, this brings some new changes to its explored neighbors, and those neighbors are added back to the to-be-checked list of the first algorithm. And therefore, this brings forward our modified version of backtracking, where it won't waste any more memory storing the constraints. The backtracking does not backtrack to check whether this change had any effect on its previously checked cells, but simply, it just goes and checks whether this change had any effect only on its neighboring cells, and this can make the algorithm run much faster than with the regular backtracking.

An example of the solution of the method is in Figure 4.1, where the cell in blue gets its two flags because it has only two unexplored cells, and none of its explored cells had any flags.

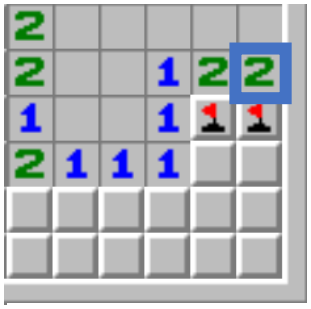

Figure 4.1

The continuation of that example is shown in Figure 4.2, where the cell in blue's surrounding flag equals its value (1=1). Therefore, all its unexplored cells are being explored now.

Figure 4.2

However, there may (and will for hard configurations) come a time when this first algorithm will finish all its checks and will not be

able to bring changes to the map considering each cell alone, as its rules wouldn't be satisfied for any of those cells. Therefore, the algorithm heads to its second method.

4.1.2 Second Method

The second method in the algorithm is similar to the 2-RC method from section 3.1.2, where it uses the neighbors of each cell together to get a piece of new information on the board. Any new change will add the neighbors of the changed cell to the to-be-checked list, and the algorithm will head back to its first method, as the run time of the 2-RC is much higher than the average run time of GAC.

Figures 4.3, 4.4, 4.5, 4.6 will show in order a case where the first method cannot be applied anymore, and the algorithm will head to the second method until it gets a useful change in the map that leads the algorithm to head back to the first method to continue solving the configuration. In Figure 4.3, the neighboring cells of the blue cell will have one bomb. Therefore, the bomb of the red cell will be one of the blue's neighbors, and we can explore the other cells of the red cell, but we did not get anything important from the result. We continue with the second method in Figure 4.4, and again the blue cell will have one bomb in one of its two neighbors. Therefore, we can see that the red cell's first bomb will be in one of blue's neighbors, and the other remaining neighbor of the red cell is a flag. But again, we did not get anything useful for the first method and continued

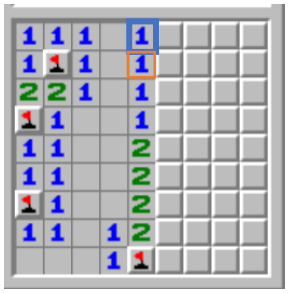

Figure 4.3

Figure 4.4

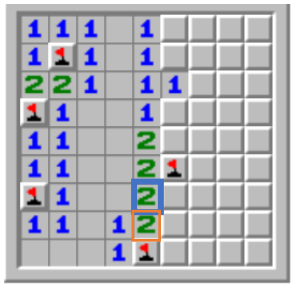

Figure 4.5

Figure 4.6

with the second method. Finally, in Figure 4.5, as the blue cell got its one bomb, therefore, one of its other two neighboring cells will be a bomb, and we can get from that that the red cell's other bomb will be one of the neighbors of the blue cell's, and we explore its remaining cells, and we explored a cell whose value is 1 (Figure 4.6), which we can continue with that again using the first method.

Another example of how the second method works is in Figures 4.7-4.8 in order, where the neighboring cells of the blue cell will have one bomb, and the neighboring cell of the red cell will have four bombs. As the blue and red cells share two cells, and after eliminating those two cells, the red cell will have three other neighbors, which means each of those neighbors will be flagged because there can be at most one bomb in the shared two cells, and the last flag of the red cell will be one of the shared cells, meaning blue's leftmost cell (the yellow cell) will have to be explored as blue's one flag will be one of the shared cells. After flagging and exploring some of the cells using the second method, the algorithm will return to the first method. If neither of the first two methods brings any new change to the map, the third method will be applied to the configuration.

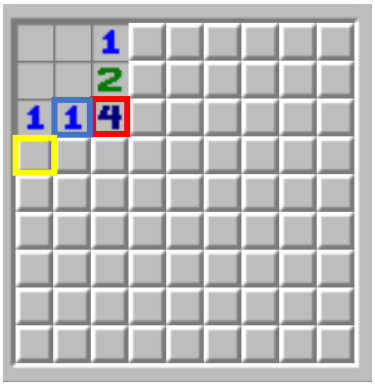

Figure 4.7

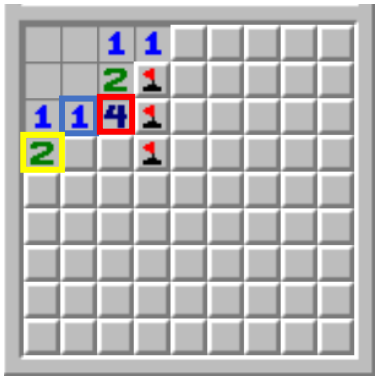

Figure 4.8

4.1.3 Third Method

The third method starts searching the board for unexplored cells with at least one explored neighboring cell. Whenever it finds one, it assigns a flag to that cell and goes on and applies the same minesweeper solver first and second methods on it, trying to solve the board while a cell is

assigned to a flag. However, while the third level assigns a flag to a cell, afterward, the game cannot explore a cell; it just assumes that that particular cell is explored and continues flagging other cells according to that. The goal of not exploring a cell is that even if the first assigned flag is wrong, you will not lose, but that wrong assignment may lead you to explore a cell that is a bomb, which will result in a loss. That's why the algorithm continues and tries to find something wrong or errors in the configuration, resulted by flagging the initial cell (error meaning will be explained later using Figures 4.9-4.10). This will make the algorithm backtrack to the flag-assigned cell and explore it, as the result of flagging it leads to an error, meaning that cell must be explored. If the algorithm does not find any errors by assigning a flag to that particular cell, it will backtrack, remove that assigned flag, search for another unexplored cell with explored neighbors, and apply the same steps.

In Figure 4.9, We cannot apply the first two CSP methods on the board, so we go to the third method. Flagging the yellow cell results in the green cell getting all its bombs and exploring its remaining cells (the one in black). This results in the red cell having five flags in its surrounding five remaining cells. Finally, this makes the blue cell with value 2 have three flags surrounding it, which is a contradiction, meaning, assigning a flag to the yellow cell was wrong, meaning the yellow cell has to be explored.

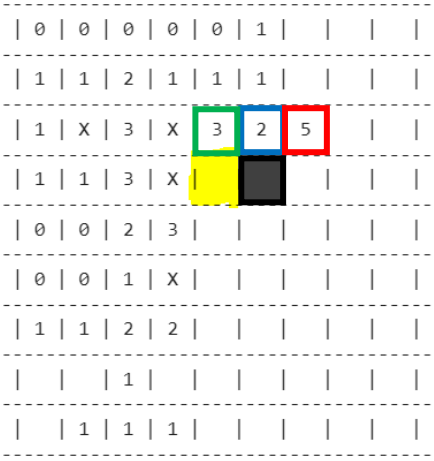

Figure 4.9

Another Example can be seen in Figure 4.10, where assigning a flag to the yellow cell and exploring the red cell's remaining two cells (as its value is 3 and it already got all its three flags) will end up with the green cell with value 5 only having four flags in its surrounding unexplored cells, meaning a mistake. That's why we backtrack to the origin point, remove the flagged cell's flag, and explore it.

Figure 4.10

Finally, if none of our three methods change the map, the algorithm applies our Reinforcement Learning model on the board to predict a cell's value. If the prediction is right, the algorithm returns to the first method; otherwise, it is a lost game. Therefore, we can say that our algorithm does not make any wrong move while solving the configuration unless it gets stuck and chooses to explore or flag a cell using the RL model. The game is a win when the map is covered with the number of flags equal to the amount dedicated to that map, and the remaining cells are uncovered without exploring a bomb cell.

4.2 Reinforcement Learning Formulation

We initially aimed to apply Reinforcement Learning (RL) to solve the Minesweeper game. During the research, we started with Q-learning, a widely used RL algorithm. However, we soon encountered a challenge due to the large state space of the minesweeper. With a 9x9 (easy configuration) board consisting of 81 cells, the number of possible board configurations becomes huge (2^{81}), making it impractical to use Q-learning directly. As a result, we transitioned to Deep Q-Learning, which employs a neural network to approximate the Q-values and can handle large state spaces more effectively.

Although our goal was to develop the Deep Q-Learning and train it on the whole easy board, we found from some research that we need at least 60,000 epochs of training, and we didn't have the computational power to do that. Therefore, to not just have a non-trained method and to further enhance the training efficiency of RL, we devised a strategy to feed only a small part of the board into the RL agent. This way, instead of training the RL agent on the entire board, which would require significantly more training epochs, we decided to focus on training the agent on partial board configurations. These partial boards are obtained by pre-played minesweeper games leveraging CSP techniques, choosing the cell to be explored from the cells with at least three neighboring explored or flagged cells, and then taking the neighboring 25 cells of it (Figure 4.11).

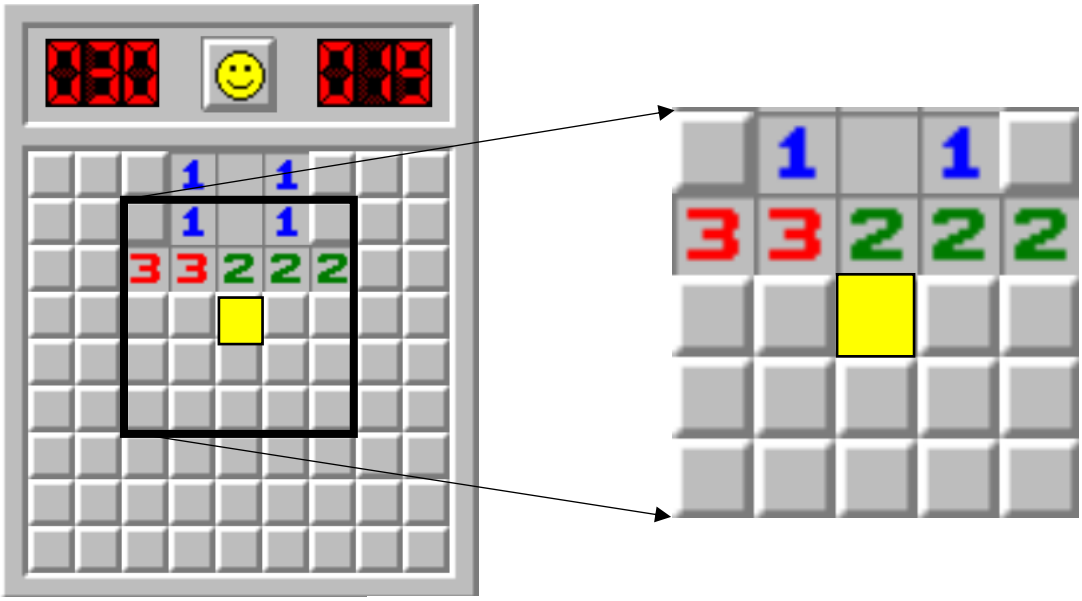

Figure 4.11

The CSP approach is crucial in preparing the training data for RL. First, we apply CSP algorithms to the board. For the cases where CSP algorithms do not find a solution and get stuck, we use one of the unexplored cells of the half-solved board as the center of the 5x5 mini board to train to find the best action for that cell using RL. In these cases, RL steps in to provide a potential

solution and learn from the outcome, bridging the gap where CSP struggles. This hybrid approach allows us to exploit the strengths of both CSP and RL, combining the logical reasoning capabilities of CSP with the learning and exploration capabilities of RL.

Employing CSP as a precursor to RL training has several advantages. Firstly, it reduces the burden on RL to explore the entire state space of the game, which is computationally expensive and time-consuming. By training RL on partial boards, we can significantly reduce the training time while still enabling the agent to learn effective strategies. Additionally, using CSP helps provide a more structured and meaningful training set for RL. The solutions obtained from CSP represent logical deductions and patterns that can guide RL towards optimal actions, allowing RL to leverage the knowledge acquired through CSP reasoning.

4.2.1 Network Architecture

Our network consists of three convolutional layers followed by two dense layers (Figure 4.11). The convolutional layers were designed to simulate an actual player's behavior when exploring the board, and each convolutional layer utilized 3x3 kernels to capture the local information from the neighboring cell of the unexplored cell, helping the agent extract relevant features and patterns from the board (Dumane, 2020). The use of convolutional layers was essential and well-suited for capturing the patterns in our grid-like structures. RELU (Rectified Linear Unit) activation function was applied after each convolutional layer to bring non-linearities into the network and enable it to learn complex states and representations (Dumane, 2020).

Following the CNN layers, we added two fully connected dense layers. The first one uses RELU activation for the network to learn the non-linear relationships of the extracted features from the CNNs, and the second one has a linear activation to produce a single output, which is the Q-

value for each possible action on that state, which in our case is two (Dumane, 2020). The network's specific architecture and parameters, including the 3x3 kernels and the two types of layers, were determined through research and experiments.

```
model = Sequential([
            Reshape((5, 5, 1), input_shape=(25,)),
            Conv2D(64, (3,3), activation='relu', padding='same'),
            Conv2D(64, (3,3), activation='relu', padding='same'),
            Conv2D(64, (3,3), activation='relu', padding='same'),

            Flatten(),
            Dense(64, activation='relu'),
            Dense(n_actions, activation='linear')])

model.compile(optimizer=Adam(lr=lr), loss='mse')
```

Figure 4.11

As our state representation is a 5x5 matrix which is a part of the whole minesweeper board, we reshape the matrix to a 5x5x1 shape where each element of this array contains a number that represents the cell's type. The (-2) values mean unexplored cell, (-1) means it contains a mine, and (0-8) represents the number of bombs surrounding that cell.

Our reward system was crucial in guiding the agent's learning process. When the agent takes an action that reveals a correct cell or doesn't reveal a mined cell, a (+1) reward will be assigned. Conversely, if the agent opens a mined cell or incorrectly doesn't reveal a safe cell, a (-1) reward will be assigned. Those positive and negative rewards will lead the agent to learn to avoid choices that lead to negative outcomes and prioritize actions that contribute to the game's progress.

4.2.2 Hyperparameters

With research and experiments, we carefully selected the hyperparameters for effective learning and performance. Firstly, we started training with a learning rate (lr) of 0.001. However, the agent wasn't learning from the random moves done at the beginning of the game, and we tried to increase and decrease the lr until we ended up with 0.0005. The learning rate determines the step size at which the network updates the weights during the training (Brownlee, 2019). That's why small lr means more precise updates of the board without tending to update the board faster to achieve optimality (Brownlee, 2019).

We used the epsilon-greedy strategy for the agent to perform random actions in the beginning and explore different states of the game and update the board accordingly. Initially, we set the epsilon value to 1, which allows the agent to explore the environment fully. After each move, the epsilon is multiplied by 0.9985, which is the epsilon decrement, encouraging the agent to do fewer random moves and more using the knowledge learned by the random moves.

4.2.3 Training

During training, we used a replay buffer that trains the models and updates the parameters using batches of size 64. We trained the model for 7000 epochs, where the training data was taken from the unsolved CSP configurations, and although the model required more training, we were limited with computational power. Each game was represented as an epoch. The Bellman equation was used to calculate the target Q-values to update the parameters. To improve the overall efficiency of training, we used Adam Optimizer, which uses adaptive gradient descent algorithms, adapting the learning rate for each network parameter based on their historical gradient (Brownlee, 2017). The loss function was MSE (Mean Squared Error), which calculates the squared difference

between the predicted and target Q-values. If MSE gets minimized, the Q-values tend to make accurate decisions.

Throughout the training process, we monitored the agent's performance and calculated the win rate after each epoch. We didn't need to calculate the average reward of each game as each game is only taking one action so the reward will be either (+1) or (-1). Overall, our applied techniques and practices worked well for the agent's learning process, acquiring the necessary knowledge to explore or flag a cell.

5 Results & Analysis

We created a minesweeper game using Python from scratch to test our solvers. We made the game so that each time it generates a board with randomly placed bombs. So, we generate and use the tests by ourselves. The game has three configurations (levels), easy, medium, and hard. The easy level has ten bombs randomly placed on the board, which has the size of 9x9 cells. The medium level has 40 bombs randomly placed on the board, which has the size of 16x16 cells. The expert level has 85 bombs randomly placed on the board, which has the size of 16x30 cells. Thus, the difficulty increases by increasing the number of bombs and map size at each level.

Initially, we started by training the RL agent and tried different experiments to optimize the performance of the agent. One important aspect we focused on was the choice of learning rate. We trained the model using these three different learning rates: 0.001, 0.0005, and 0.0001. After analyzing the results, we found that the learning rate 0.0001 achieves the highest win rate. This indicates that a lower learning rate allowed the agent to make more accurate and effective updates to its learning policy during training.

Moreover, we explored different values for epsilon and its decrement. After experimenting with smaller epsilon values with smaller decrements, and bigger epsilon values with other different decrements, we found that the best decision is letting the model perform at least 1500 random moves in the beginning. Therefore, we chose the epsilon to be 1 and the decrement to be 0.9985. In this case, the epsilon value will start from 1 and decrease until it gets approximately 0.1 after 1500 iterations. This way, the agent will effectively explore the game in the first 1500 iterations while learning and using some of its knowledge from different states.

After training the model on 7000 epochs with the important parameters used as mentioned above, our win rate, which measures the proportion of successful games solved by the agent, has reached 75%, indicating the success of the agent in learning to make correct decisions in just 7000 epochs. This means that with this small amount of training, our model went from random choices to actually predicting the correct decision in 75% of the game cases, leading to a huge increase in win rate from only the CSP model to the mixed model.

Therefore, we ran the algorithm for 500 iterations on each difficulty level to test our solver algorithm. The following table shows the experiments we conducted on the solvers to evaluate their performances. We tested the performance of the first method of the CSP alone. Then, we added to it the other two methods. Finally, we added to the CSP methods the RL model, which is our final representative model.

	Only CSP First Method	Only Three CSP Methods	CSP Methods With RL
Easy	25%	57%	90%
Medium	10%	38%	72%
Hard	1%	20%	47%

Table 5.1

Table 5.1 presents interesting findings regarding the win rates obtained from running the algorithm for 1000 iterations on each difficulty level. We can see that relying only on the first method yielded unsatisfactory results (25%) for the easy board. By applying all three CSP methods together, we get a 33% increase in the win rate percentage point. However, this result shows the percentage that the CSP methods can solve without making any guesses, so whenever there are no constraints left to satisfy, we assume that the game is lost without any more moves.

Finally, the most remarkable improvement in win rate was observed during the CSP and RL combined model. We witnessed a 90%-win rate for the easy board, 75% for medium, and 47%

for large. This proves the benefits of using logical models with Reinforcement Learning, showing the significant contribution of RL in fulfilling the limitations of CSP. This also improves and enhances the model's ability to solve more complex boards, which are bigger, and with more mines.

Another Experiment was observing the interaction between the CSP and RL (Table 5.2), noting how often CSP deferred to RL for decision-making. It was observed that out of 1000 games, on average, RL was used 2.4 times a game in easy mode, 2.2 times in medium, and 1.7 times in hard. This indicates that CSP successfully solved a significant portion of the game scenarios independently, showcasing its robustness in handling Minesweeper puzzles. However, whenever CSP got stuck, and RL took charge, we examined around 72% accuracy in decision-making. However, it is worth noting that RL did a wrong move in an average of 28% of cases. We know that this value can be decreased by increasing the epsilon decrement, letting the model do more random moves at the beginning of the training, exploring different states, and letting the model train for more epochs.

	Applied RL	RL Revealed Correctly	RL Revealed Wrongly
Easy	2432	74%	26%
Medium	2290	71%	29%
Hard	1792	69%	31%

Table 5.2

Finally, we conducted an experiment to analyze the percentage of the minesweeper board that was successfully solved and uncovered before that game led to a loss (Table 5.3). This provides the progress made by our model when they were failing. We got that, on average, for 1000 games, our CSP model covers around 18% of the board before losing, excluding the winning cases, and the combined model covers 38% of the board before losing. This shows why the board fails

sometimes after applying RL more than three or four times on it, which is logical, considering the correct move probability of 75%.

	Average Percentage Covered Before Losing Only CSP	Average Percentage Covered Before Losing The Combined Model
Easy	14%	37%
Medium	17%	38%
Hard	20%	38%

Table 5.3

6 Future Work & Conclusion

There are several avenues for future work that can further enhance performance and expand the scope of our methodology.

Firstly, explore more and experiment with RL's input state space. As shown in our case, each cell has only two Q-values (N*M) as we have two actions), which can explain the number of bombs surrounding that cell. These two Q-values are getting updated regularly for that particular cell. Another good approach can be for each cell to get 20 Q-values (N*M*10), where each of the first nine Q-values represents the number of flags surrounding that cell, and the tenth Q-value shows whether the number of flags is unknown. In this case, instead of only one neuron connected to each cell, there will be ten neurons. Although this approach is experimental, it may produce good results as more neurons may optimize the training and enable the network to absorb more information.

Secondly, adding domain-specific heuristics into the RL training process, which leads the agent to explore and have pre-information about the state before taking action, may enhance the agent's decision-making capabilities.

Thirdly, at the beginning of the training phase, train the agent using only deep learning, where the whole open board will be the input of the model, and the model will learn the patterns from the board. Then, after understanding the model a bit about the game, continue training the

same model using RL. This may result in faster training for RL and better performance, although training the deep learning model duration may result in no less overall training time.

Fourthly, modify our RL model, where not only will the middle cell of the 5x5 board be explored or flagged, but the model output will be the solution of all the cells of the 25 cells of the input board. For this to be possible, we must increase the convolutional layers, which require more training resources.

In conclusion, our research has demonstrated the effectiveness of employing the Constraint Satisfaction Problem (CSP) and Reinforcement Learning (RL) approaches in solving Minesweeper games. We found that the combination of CSP and RL offers a powerful approach to solving the minesweeper, using RL's ability to learn patterns and CSP's ability to transform the environment into a logically solvable one, achieving much better results than applying each method alone. CSP acts as a preliminary reasoning means to narrow down the possible solutions and generate training data. At the same time, RL serves as a learning mechanism that can handle the complexities and uncertainties of the game. This combination enables us to tackle minesweeper efficiently, leveraging logical inference (CSP) and learned strategies (RL) to solve the game fully and effectively.

References

Bayer, K., Snyder, J., & Choueiry, B. (2006). *A Constraint-Based Approach to Solving Minesweeper* [Article]. Constraint Systems Laboratory - University of Nebraska-Lincoln. Retrieved November 10, 2022, from https://digitalcommons.unl.edu/cgi/viewcontent.cgi?article=1169&context=cseconfwork

Brownlee, J. (2017, June 3). *Gentle introduction to the adam optimization algorithm for deep learning*. MachineLearningMastery. https://machinelearningmastery.com/adam-optimization-algorithm-for-deep-learning/

Brownlee, J. (2019, January 25). *Understand the impact of learning rate on neural network performance*. MachineLearningMastery. https://machinelearningmastery.com/understand-the-dynamics-of-learning-rate-on-deep-learning-neural-networks/

Comi, M. (2022, June 8). *How to teach an AI to play games: Deep Reinforcement Learning*. Medium. https://towardsdatascience.com/how-to-teach-an-ai-to-play-games-deep-reinforcement-learning-28f9b920440a

Dumane, G. (2020, March 2). *Introduction to convolutional neural network (CNN) using tensorflow*. Medium. https://towardsdatascience.com/introduction-to-convolutional-neural-network-cnn-de73f69c5b83

Gil, & Chai. (n.d.). *Solving Minesweeper Using CSP* [Article]. Retrieved November 10, 2022, from https://www.cs.huji.ac.il/w~ai/projects/old/MineSweeper.pdf

History of minesweeper. Minesweeper. (n.d.). http://www.minesweeperonline.net/history.php

Luu, Q. T. (2023, May 2). *Q-learning vs. deep Q-learning vs. Deep Q-Network*. Baeldung on Computer Science. https://www.baeldung.com/cs/q-learning-vs-deep-q-learning-vs-deep-q-network

Mehta, A. (2019). *Reinforcement Learning for Constraint Satisfaction Game Agents*. Square Space.

https://static1.squarespace.com/static/5a63b41dd74cff19f40ee749/t/6155ac413fa14c4e7ae3e755/1633004611390/anav+mehta.pdf

Sajjad, M. H. (2022, May 3). Neural network learner for minesweeper. https://arxiv.org/pdf/2212.10446.pdf

Stanford University. (n.d.). *Minesweeper AI*. Retrieved November 10, 2022, from http://web.stanford.edu/class/archive/cs/cs221/cs221.1192/2018/restricted/posters/thowarth/poster.pdf

TORRES.AI, J. (2020, June 11). *The bellman equation*. Medium. https://towardsdatascience.com/the-bellman-equation-59258a0d3fa7