Minesweeper Solved as a Constraint Satisfaction Problem

Carl Anderson, Andy Chen, Archit Kalla

December 18, 2021

Abstract

In this report, we explore the efficacy of Minesweeper as a Constraint Satisfaction Problem (CSP). We are particularly interested in understanding how to solve Minesweeper strictly logically and how effective it can solve boards at increasing mine densities. We discuss the method behind creating a CSP based agent to solve minesweeper and the basic strategies implemented in the design. We then analyze both the time it takes to solve boards and the win percentage at each mine density and compare it with other findings in the broader AI community.

1 Background

The popular game of Minesweeper is fairly simple. A player is presented with an n x m board of square tiles with a certain number of those squares containing hidden mines randomly scattered across the board. The objective is to clear or "sweep" the board of mines by revealing squares one at a time, with the win condition being a board where all squares of the board are either revealed or marked as containing a mine. The game is designed such that a player's first move is always guaranteed to be a safe one, meaning there will not be a mine revealed on the first move. After a square is revealed, if there are mines in the 9 squares surrounding it, the square will give the number of mines in the surrounding square, otherwise, the 9 surrounding squares are automatically cleared as well. This will typically generate a frontier of numbers to the uncleared patches of squares that contain mines. The player can use the numbers revealed to mark squares according to what they think contains a mine. This can be used to systematically "sweep" the mines until the game is completed where all non mine squares are cleared and mines have been marked, or the game is lost when a square containing a mine is revealed. There are situations that may arise depending on the game board mine density where there is not enough information to make a guaranteed safe move, thus making higher mine densities more difficult to solve and increasingly relying on random guessing rather than using the information on the board. Mine density is a metric we came up with to represent the difficulty of a given minesweeper game independent of how large the board is. The mine density of a given minesweeper game is simply the number of mines divided by the total number of squares.

Minesweeper has been shown to be NP-complete which means that the problem of minesweeper is Non-deterministic Polynomial-time computable (NP) as well as it is "complete" which means that it "includes" all other NP-problems [Kay00]. For general purposes, this means that a Minesweeper solution can be quickly verified, but the solution itself lacks a "quick" solution. Among the AI community, the game of minesweeper has been analyzed with many different approaches, some of which include, constraint satisfaction, greedy best first, and probability, where each method may use combinations of probabilistic, greedy, and CSP algorithms to beat the game of minesweeper. In this report we will cover different methods that have been used to solve minesweeper and then cover our observations when implementing a CSP Minesweeper solver.

2 Literature review

2.1 Constraint Satisfaction

In David Barreca's thesis he states, "Minesweeper fits naturally into the CSP framework. Each square will be represented by a variable with a binary domain of 0 and 1. A value of 0 indicates the

square is mine-free while a 1 denotes a cell containing a mine. Every time a square x is probed, x constrains the number of mines present in its neighbors" [Bec15]. To exploit the fact that minesweeper is simply a large constraint satisfaction problem, Barreca offers two methods to playing the game.

First Berreca uses a simple CSP method which involves identifying possible squares that can be mines and forming an equation in the form of A+B+C= 1 and each labeled square has its own equation in which the program will reduce the equations until it finds a 0 which indicates an square without a mine, typically involving backtracking to fully reduce the problem space [Bec15]. This method is also found in Yimin Tang's paper on Minesweeper as a CSP where Tang, et. all parse the minesweeper board and to equations, group the equations and perform reductions to find the next move [Tan18].

Berreca's other solution involving CSP involves using the concept of Coupled Subsets originally formulated by Chris Studholme [Stu]. Using the CSP method above, the coupled subset method handles large constraint spaces and as Berreca describes it, "To reduce the search space size, C is partitioned into coupled subsets. Two constraints are coupled if they share a common variable. As a result, backtracking search can solve each subset separately as an independent subproblem... If no safe move is discovered from the constraint solutions, the solver calculates a probability estimate for each square being mine-free based on the solutions found through backtracking search" [Bec15].

A common theme found in papers talking about CSP is that the method is not perfect. Variety of papers have found CSP to be fairly slow due to the backtracking and analysis of each constraint in the problem space. Although in Tang and Berreca's paper they mention doing stochastic choices first when one is needed in the problem space to speed up the reduction process, this comes at the cost of win rate. In further research done by Tang, et all. In further papers by Tang, he explored other options involving CSP based approaches however he had a similar problem where he had to sacrifice accuracy for speed which is also highlighted in Luis Gardea's paper [GKS15]. The general consensus however, is that CSP based methods are entirely feasible and have been done a multitude of times in the broader AI community.

2.2 Greedy Best First

The greedy-best first method of beating minesweeper is simpler than constraint satisfaction. It relies on calculating some heuristic by way of a heuristic function, in this case calculating the chance that each game square is a bomb, and then choosing the best choice first or the square that is least likely to be a bomb. Rendyanto explains in their 2019 paper that we can define a set of logical rules which can be used to assign probabilities that each hidden square is a bomb [Ren19]. Once we have determined the probability that each square hides a bomb we simply make the next move which is least likely to expose a bomb.

The first two rules are quite simple and are as follows; If an exposed square has an equal number of unexposed unflagged adjacent squares as it's number, all of the unexposed adjacent unflagged squares have a 1 probability of being bombs and can be flagged. If an exposed square has the same number of unexposed flagged adjacent squares as it's number, all unexposed unflagged adjacent squares have a 0 probability of being bombs and can be flipped [Ren19].

The final rule is we need to calculate the probability of a square if it is neither 0 nor 1. This can be done using what is mine density calculation, which will become our greedy heuristic. The mine density of each square is the chance that square contains a mine given the information we have. This is calculated by iterating over each revealed square, dividing that squares number less the number of flagged squares surrounding it by the number of unrevealed squares surrounding that square. This rule actually encompasses our first two rules, e.g. if a revealed square of number 3 has 3 unrevealed squares surrounding it and no flags, we would assign the probability of each of those squares to be (1 - 0) / 1 = 1. We always take the maximum heuristic number calculated for each unrevealed square through this method and then pick the largest of the calculated heuristic over all squares [Maz11]. If there is a tie between multiple squares that are of values less than 1 we can increase our odds of guessing correctly by recalculating the heuristic except instead of taking the maximum mine density from each revealed

square we can multiply them together and then take the maximum. Finally, if there are no flipped neighbors for a square we automatically assign it the probability of 0.5.

2.3 Basic, Probabilistic, and Tank-Solver Approach

The basic minesweeper solving algorithm is pretty simple, reveal a square and upon having the numbers revealed, continue to employ set strategies to solve the board. The basic strategies that solvers employ are simply logical calculations on whether a mine can or can't be in a square based off of the numbers revealed around it. This works for simple boards with low mine density, but as mine density and also board size increases, this strategy will lose effectiveness.

This is where the tank-solver algorithm becomes useful. This algorithm will enumerate all possible mine configurations of the board or a section of it using the revealed numbers, then will choose the next square to expand based on which squares in the possible configurations don't ever have a mine [Li17]. The reason this isn't used as the first strategy used in solving boards from the start is that this strategy is very computationally infeasible at large board sizes, thus is only employed when more progress has been made on the board. This approach will get our solver farther, but it will inevitably run into another issue.

There are situations that arise where no amount of logical calculation can be done to guarantee that the AI will solve a minesweeper board without clicking a mine. In cases where there is a clear 50-50 on where the mine could be, truly nothing else can be done and it is left up to chance [Bec15]. In other cases, the board configuration can reveal clues about where mines could likely be. This situation commonly arises in boards of higher mine density, and probability can be employed to maximize the chances of correctly solving a board while minimizing the risk of failing.

In such cases, a common probabilistic strategy used by minesweeper solver AIs is expanding upon the tank solver algorithm and calculating the mine probability of each square then selecting the best one [Ren19], similar to the greedy approach except with the use of Tank-Solver. After enumerating all possible mine configurations of the entire board or a just section of the board (as part of the tank solver algorithm), the solver counts the total number of times the potential configurations place a mine on each square. After doing so, the solver will select the square(s) with the least count, meaning that the square(s) has the last potential configurations where a mine is placed, to expand next.

These probabilities can be further narrowed down to be resolved with logical decisions with the consideration of the remaining mine count on a game board. Some uncertain probabilistic situations, once factoring into consideration that there may only be 1, 2, or however many mines left, become a certain and solvable by logic [Li17].

2.4 Literature Review Conclusion

Based on the methods we have analyzed, there are various combinations of methods we can employ to create a minesweeper agent. Each method discussed above offers a different benefits and drawbacks which when creating our solution we will need to be aware of. We can confidently say that minesweeper has been exhaustively researched, however this does not mean that more efficient methods do not exist.

3 Design approach

The basic strategy when playing minesweeper is to eliminate the possible mine locations based on the numbers that are given on the frontier. First, you would check if there are any existing flags touching a numbered square. If the number of flagged squares touching the numbered square is equal to the number on the square, you would clear the remaining uncleared squares. Combining multiple numbered squares and possible options you have an effective strategy to solve minesweeper. However as the mine density increases, playing by this strategy may result in incomplete information, thus a common improvement on the base strategy is to start in a corner. This is to reduce the likelihood of "getting stuck", or run out of information to make a guaranteed safe move. Because the first move is

always safe, you are less likely to get "stuck" when selecting a corner.

Treating Minesweeper as a CSP seems like a logical first step in creating a Minesweeper solver. Each of the numbered squares can be treated as constraints for where mines are.

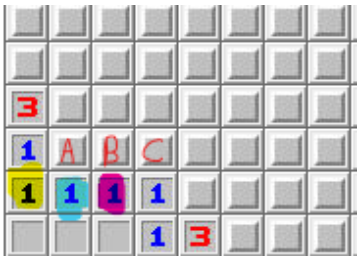

Figure 1: Simple constraint example

For example, in Figure 1 above we can see the 3 squares that could potentially be mines. Looking at the numbered square highlighted in yellow we can make the constraint A=1. For the blue highlighted square we can say A+B=1, and for the pink highlighted square we can make the constraint A+B+C =1. We can further reduce the equations, so for the pink highlighted square, given that A=1 we can reduce the constraint to 1+B=1 -¿ B=0, and given this the pink highlighted numbered square can be reduced to 1+0+C=1 -¿ C=0. Any equation that is equal to 0 means that the square is open, where on the other hand squares with mines are denoted with 1.

In more complex cases we can exploit this reduction. For example lets say Constraint 1 (C1) : A+B+C = 2, C2: A+B=1, and C3: B+C=1. In this example we can reduce the constraints to one variable we can say for certain is a mine, by reduce the scope of the constraint. Using C1 and C2 we get (A,B,C),(A,B) =¿ (C). Thus, C1 -C2: A+B+C -A -C = 2-1 =¿ C=1. We can now say that square C is a mine allowing us to select square A and B safely.

However there are edge cases were this reduction leaves us with a multivariable reduction (a case where A+B=X where X¿0). In this case, there is no additional information for the CSP method to obtain, otherwise known as a contradiction in the constraints. This calls for a need to take a random guess to get new information. In CSP code there are prints to the terminal which alert us when the solver will make a random guess due to these contradictions. In Figure 2, it is obvious that there are no safe moves with the available information and hence the solver makes a random guess (in this case it makes multiple random guesses).

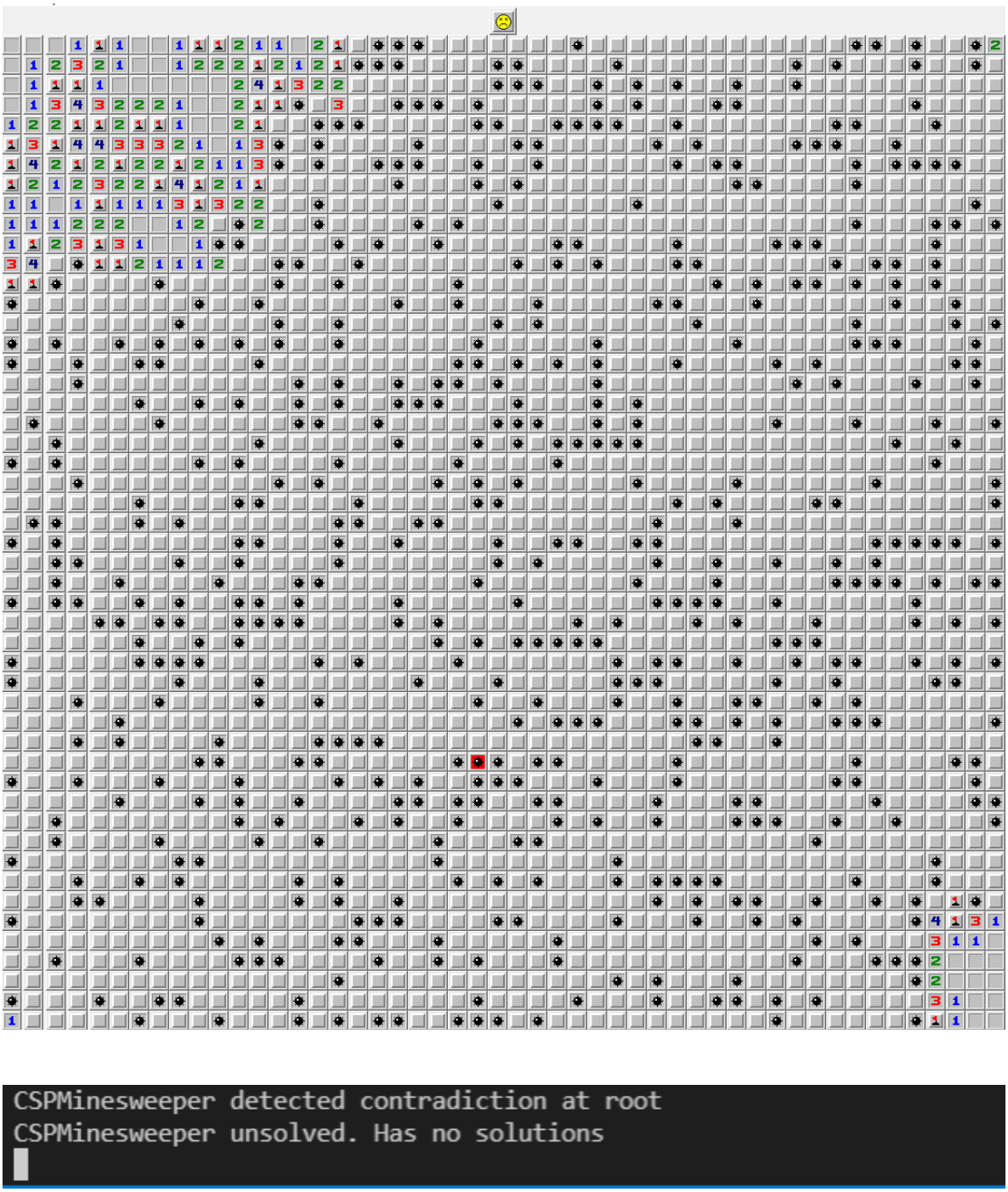

Figure 2: Not enough information (Contradiction)

A key thing to note in this solver is that it only treats the board logically, not probabilistically, hence the random guesses are completely random, taking nothing in the already discovered board into account.

4 Implementation

We were able to find both a python version of minesweeper and a CSP solver for minesweeper. The CSP solver was built on the concepts stated above. The code itself is fairly complex. The code treats

each square as an object and uses the objects methods to perceive the game state. Then using the methods mentioned above, the solver finds mines and clears spaces. The solver also uses a backtracking method that gets called to add new information to the constraints list. This cycle of perceiving the game board, reducing known constraints, clearing spaces, and backtracking can each be treated like a "move". much like how a human would solve minesweeper.

5 Experiments and Results

To run our experiment, we used a Minesweeper game software written in Python using their built in CSP based solver algorithm. For the game board size for our tests, we set the board dimensions to 30 x 16 which is the standard size for an expert game board in Microsoft's Minesweeper. We added code to the software to time the runtime of the algorithm for each time it's executed on a game board. We chose to run the solver algorithm 100 times for each level of mine density ranging from 0.0 to 0.3 in increments of 0.01, logging the success or failure of the run, and also the runtime of the algorithm if the solver successfully solves the board in a csv output file.

After running our tests, we took the data that was generated and plotted it on a graph as the success rate of the solver algorithm as a function of mine density. We then graphed the total runtime of the solver run as a function of mine density. After carefully reviewing the test results, we decided to repeat the experimental trial a second time due to what we believe was an outlier in the data that did not accurately reflect what we were testing, and graphed that second trial data as well. After some discussion, we ran the experiment a third time, and graphed the success rate in the same way as before, but the runtime data was graphed only using the average runtime data of successful solves instead.

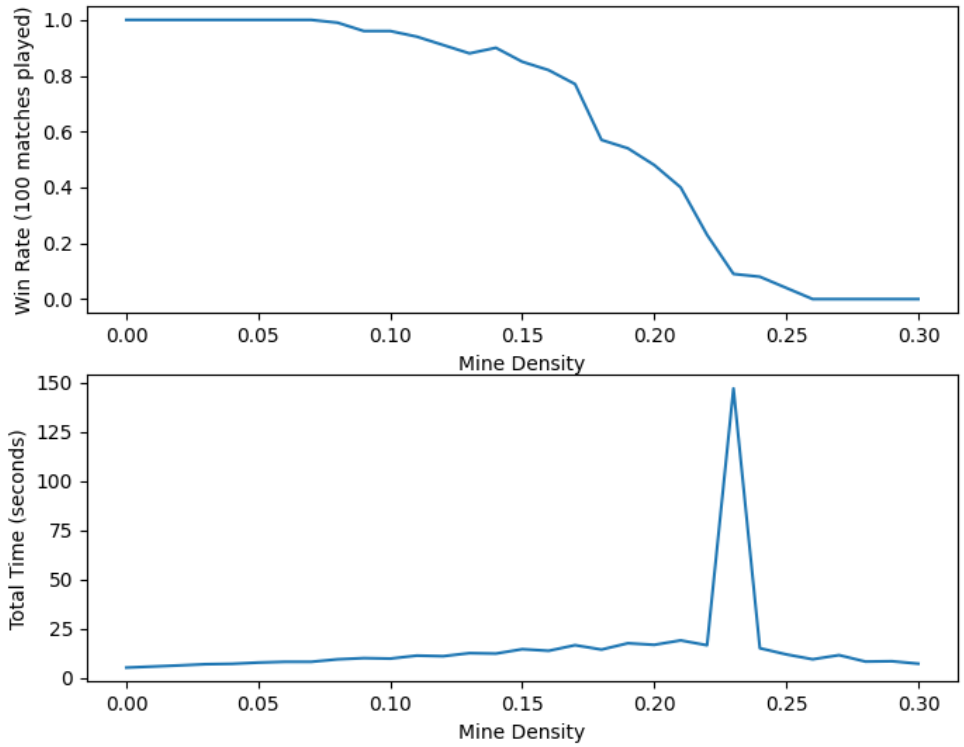

Figure 3: Graphed results of first trial run.

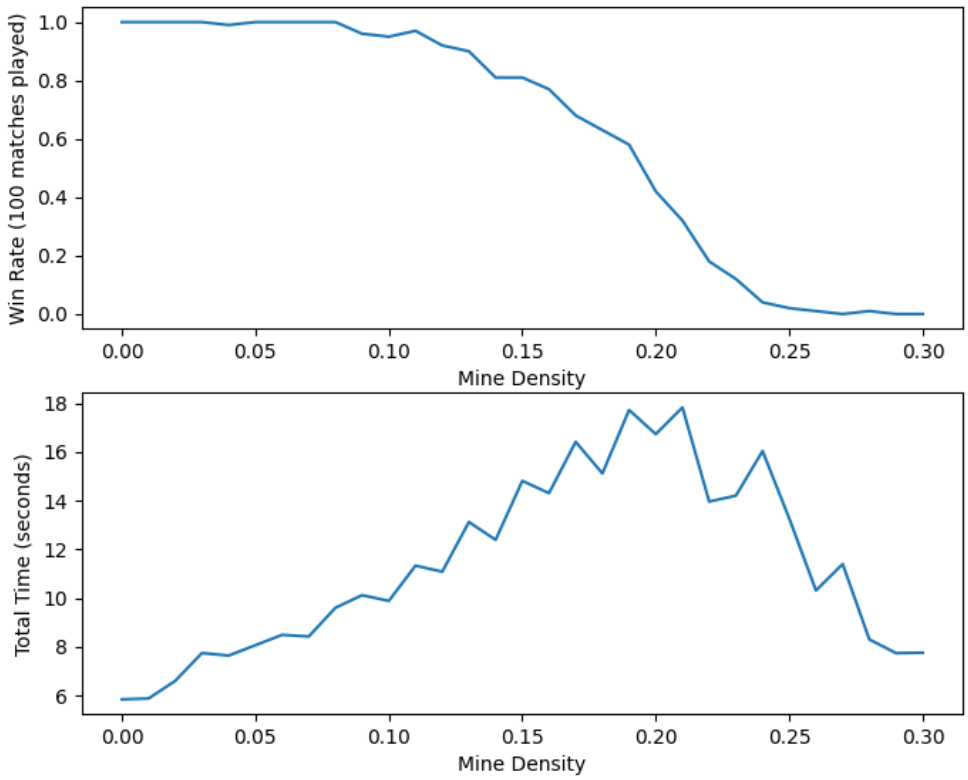

Figure 4: Graphed results of second trial run.

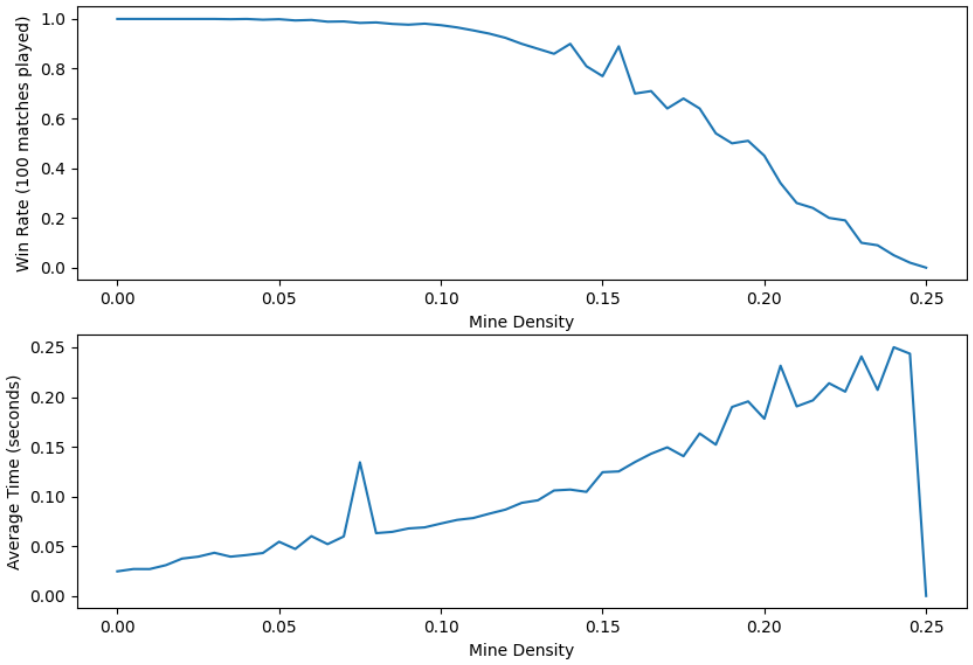

Figure 5: Graphed results of third trial run.

6 Analysis

When the solver success rate is plotted on a graph as a function of mine density, the resulting graph appears to resemble a reverse logistic curve centered at around 20% mine density. In our first results graph was a large spike in the runtime at around 23% mine density which was around 700% the size of the rest of the measurements. We were not able to recreate this among any other tests we did and found no other cases of an extraordinary runtime for a seemingly random mine density. Given that we couldn't recreate this issue it was simply chalked up to a random hardware issue and ignored for the remainder of our testing and analysis.

For our second trial, we simply re-ran our first experiment and got very similar graphs without the enormous random spike in the time portion present in the first trial. The total runtime graph had a smaller scale, allowing for better analysis of the data. The data shows a clear upwards trend until a mine density of about 20%, then a downwards trend from there. 20% is also the center of the reverse logistic shape of the success graph.

In the total runtime graph of trial 2, the lower scale also reveals a consistent "spiky" up and down shape as the curve progresses along the x-axis. Our only possible explanation for this is that this may possibly be due to some very specific scenarios related to mathematical patterns of mine distribution with specific quantities of mines, if such a thing exists, and how that relates to how the solver algorithm runs.

In the third trial, we graphed the success rates the same, but only plotted the average runtimes of all the successful solver algorithm runs. Our success rate graph was very similar to the graphs in our other 2 trials, but our average runtime graph had a linear increase with the values towards the end of the x-axis being sporadic due to fewer success times to be averaged and plotted. The maximum average runtime the algorithm took was about 0.28 seconds.

We then compared our results with those from another similar experiment. The results of that experiment being shown in figure 6 below.

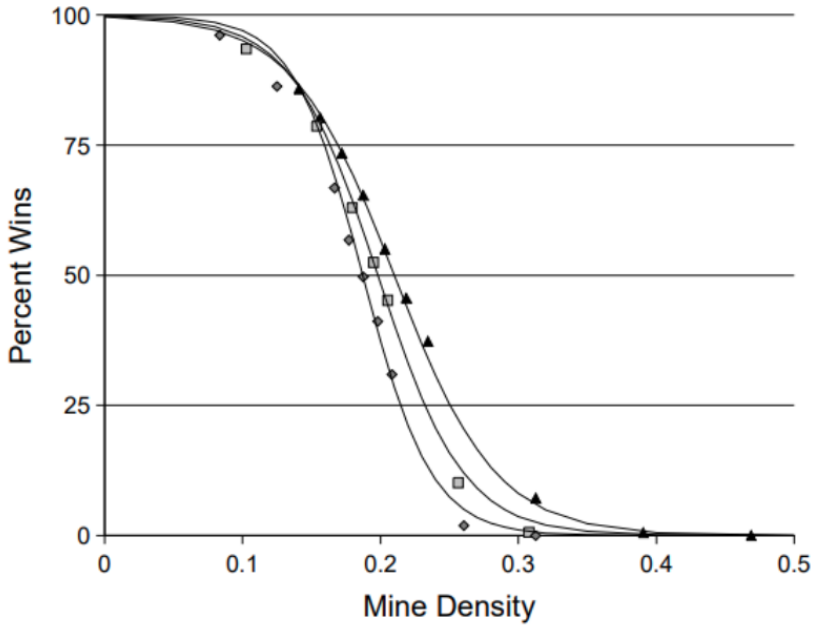

Figure 6: Win ratio as a function of mine density for the three standard difficulty levels. Triangles for the beginner level(8x8 with 10 mines), squares for the medium(16x16 with 40 mines), and diamonds for the expert level(30x16 with 99 mines). Data points fitted to logistic curves [Stu].

In figure 6, the curve with the diamonds represents the experiment with expert (30 x 16 with 99 mines 20% density) games, which is the board size we ran our experiment on. By comparing the success rate graphs of our results with the graph in figure 6, the trends all seem to be consistent: the results follow a reverse logistic curve centered at around 20% mine density.

7 Conclusion

In all, we found that minesweeper can be solved using a constraint based approach in normal cases with reasonable success. In vanilla minesweeper, the expert board has 20% mine density. Our results found that using only the information present on the board, our solver has an approximate 45% chance at solving the board. It fares well considering that the solver only works logically, and with an average runtime of 0.30 seconds per board and 20 seconds for 100 runs, performs relatively quickly as well. An important take away that we had was that the reduction method that the CSP solver uses was very similar to the resolution with refutation that was exercised in class. The method that you can combine the known constraints (sentences) and reduce them really opened our eyes to how the fundamentals are applied in real AI algorithms.

In the future, improvements can be made to the solver algorithm and experimentation process. Firstly, the solver's random guessing upon encountering a situation where no logical choice can be revised to use probabilistic guesses as there are algorithms to estimate the likelihood of a mine at certain spaces with the CSP as a base for them. This is seen in Tang's paper on their implementation of a minesweeper solver [Tan18]. Additionally, more tests on different testing conditions can be run to fully understand the limitations of the CSP solver. Similar to what's seen in Studholme's paper, he ran tests that changed the board shape to see if a column like board vs a rectangular board changed the probability of success in a strict CSP minesweeper solver. These are just 2 ways that we could continue to expand on and improve our algorithm and experimentation.

References

[Bec15] David Becerra. Algorithmic approaches to playing minesweeper. *FAS Theses and Dissertations*, pages 100–107, 2015.

[GKS15] Luis Gardea, Griffin Koontz, and Ryan Silva, 2015.

[Kay00] Richard Kaye. Minesweeper is np-complete. *The Mathematical Intelligencer*, pages 9–15, 2000.

[Li17] Bai Li. How to write your own minesweeper ai, Feb 2017.

[Maz11] Marina Maznikova. Minesweeper solver, 2011.

[Ren19] Hafidh Rendyanto. Perfect minesweeper ai using greedy and probability calculation, 2019.

[Stu] Chris Studholme. Minesweeper as a constraint satisfaction problem. Last accessed 5 November 2021.

[Tan18] Yimin Tang. A minesweeper solver using logic inference, csp and sampling. *ArXiv:1810.03151 [Cs]*, 2018.