Final Report

Nadya Postolaki (posto018)
Fatima Rahman (rahma176)

May 11, 2020

Abstract

The game of Minesweeper is a widely known single-player computer game. The environment exist as a grid of cells in which, usually, a known amount of mines is hidden. The goal is for the player to reveal all cells that do not contain a mine without disrupting (revealing) the cells that do contain mines. Many have studied the game and have developed strategies for solvers to efficiently play the game with different levels of success. We continue the studies by comparing the strategies of existing solvers and attempting to implement the best of each into a theoretical Minesweeper solver of our own. This includes study of the initial move in an empty board and finding the success rates increases if an agent begins with a corner cell versus other types of cells, algorithms and heuristic strategies for increasing success rates including the AC-3 algorithm and naive approach to revealing cells and finding mines. We also approach the game as a type of CSP and delve into its NP-completeness, comparing and contrasting the success constraint propagation methods such as forward checking, backward propagation, and random selection.

1 Introduction: What is Minesweeper?

Though artificial intelligence research has tackled games such as Go, chess, and Sudoku in the past, our project aims to implement an AI to defeat the seminal Minesweeper. A single-player video game, Minesweeper has eluded both the young and old since its inception in the 1960s. Despite its deceptively simple premise, where all one has to do is dig around randomly placed mines, the game requires a degree of intuition and guesswork alongside logical thinking. One false click can trigger a series of explosions and ultimately your loss.

Using Stuart Russell and Peter Norvig's terminology[2], Minesweeper can be classified as a partially observable, deterministic, and static game. Though the environment is "known" in the sense that the game board operates by a strict set of rules, the player has no access to the full, unveiled Minesweeper board until the end, thus classifying it as "uncertain." But how does one play the game? First, let us visualize the game board:

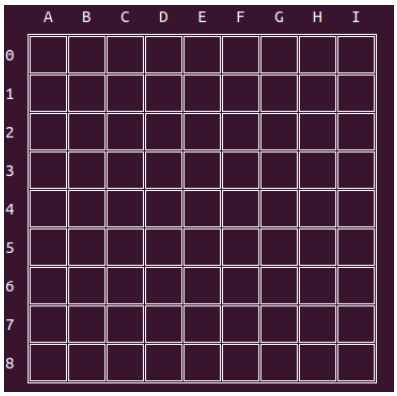

Figure 1: Blank Minesweeper Board

Before the player makes any moves, the game's initial state is a board (or "grid") of covered cells. Depending on one's chosen difficulty level (easy, medium, or hard), the board can be anywhere around 8x8 squares to 16x32 in most Minesweeper programs. Our project implementation will use a relatively easy size of 9x9.

The player can then pick any cell on the board, revealing a block of numbered and/or empty cells - the key is that the initial cell that is chosen to be revealed will never hit a mine, however the probability of a mine existing in any cells chosen at the first click will always be at 50 percent as the environment is not yet revealed partially nor in full. Once the first cell is chosen to be revealed, however, should the cell not contain a mine, an agent has two possible actions for all remaining cells on the board- flag/no action or revealing another cell. A flag or no-action, which we may refer to as no-op, is an action an agent may take when the probability of the cell containing a mine is 1. This is because the goal is to reveal all other cells whose probability of containing a mine is 0 without disrupting the mines that are hidden throughout the board. The agent may also reveal a cell, however should the cell contain a mine, the game is concluded and is considered a loss.

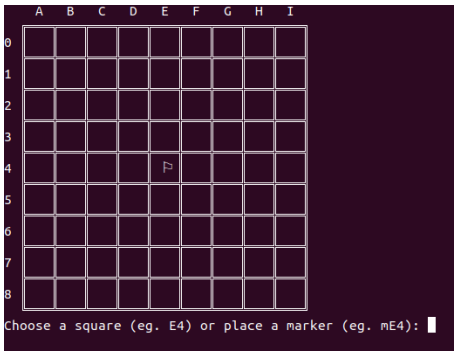

Figure 3: Placing a Marker

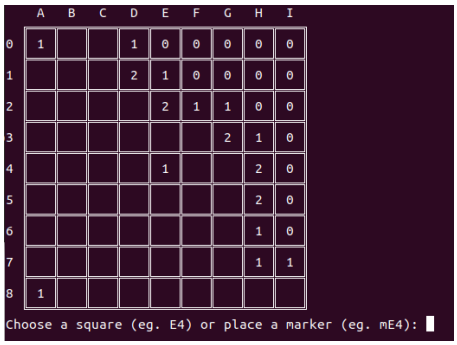

Figure 4: Partial Board Reveal

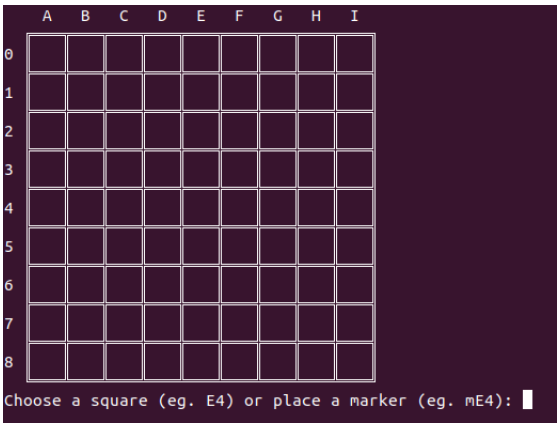

Figure 2: Instructions

Our code deviates from the classic implementation in that the player must type in a location they wish to either reveal or flag. If a cell is chosen to be revealed, the game checks whether the value of the cell is a bomb or not. If it is a bomb, an explosion is triggered and you will automatically lose the game. If not, the value of the cell is revealed by showing the amounts of mines surrounding that cell in particular. If the value of the cell is zero, then other surrounding blocks are also revealed.

The number (in the domain 1-8) signifies how many mines border the cell, in any direction (horizontal, diagonal, vertical). For example, if a cell is marked "1," there is only one mine that touches the cell. All other surrounding squares are not mines. Minesweeper's terminal test is either failure (i.e. explosion triggered) or the goal state, denoting the user's success in correctly identifying all mine sites and digging up all other cells. Not only is this game zero-sum, but it is also randomized. Once one replays the game, the board is wiped clean and the mine placement is once again rearranged.

In essence, we strive to implement an efficient take on Minesweeper by addressing it as a constraint satisfaction problem.

2 Analysis & Problem Characteristics

2.1 NP-Completeness

A majority of the Minesweeper-centric artificial intelligence studies we read through go off of Richard Kaye's seminal treatise "Minesweeper is NP-complete," which put forth the Minesweeper-consistency problem. Consistency in this case denotes a decision problem that verifies if a Minesweeper configuration is valid and consistent with its game parameters,[5] i.e. the mine layout in covered squares correlates with the uncovered, numbered squares. Proving Minesweeper's computational difficulty through a wire construct and Boolean logic gates, this NP-completeness makes the consistency verification take "polynomial-time with respect to the input size (i.e. number of total squares in the grid)"[3] on a nondeterministic machine. Brute force will simply not cut it.

Later theorists sought to either vehemently refute or expound on this claim, including the University of Warwick's Kasper Pederson, who claims, using a counting argument, Minesweeper assignments were P-complete and the SOLUTION problem doesn't have intuitive membership of the complexity class DP.[10] The Minesweeper counting problem, which "[counts] the number of legal and consistent assignments of the configuration," is also P-complete.[4] On the other hand, Hebrew University of Jerusalem's Elchanan Mossel proposes a Minesweeper game board of lattice Zd with a close to infinite state space, monotone in p, i.e. "set of winning probabilities is an interval [p(d),1] or (p(d),1]."[8] Given that the latter two are rarely cited in CSP implementations, we chose to focus on Kaye's conjectures above all else. However, it should be noted that Allan Scott et al. introduced a separate decision problem, Minesweeper inference, which "asks whether there exists at least one covered square whose contents can be safely inferred given a consistent configuration." Given that it only considers consistent graphs, inference is a "promise problem" based on propositional logic, and takes $O(n^2m^2)$*(poly time), and thus is co-NP complete.*[3][1]

2.2 Implementation through CSP

Constraint satisfaction problems, or CSPs, are a large focus on the Minesweeper game and are dependent on its NP-completeness. Minesweeper can be easily interpreted as a CSP as it fits the constraints of such problems. Each cell of the game can be represented with a boolean domain,[6] true and false or 1 and 0, where 0 or false can be the value of a cell with no mines/bombs and 1 or true will be the value held by the cell that does contain a mine. The minesweeper game was previously defined with rules that allow for a cell to reveal whether it contained a mine or not, and if it didn't, it also revealed the number of mines surrounding that particular cell and, at times, the cells neighboring it

as well. [12] The number a revealed cell, we will call n, implies that there are exactly n mines surrounding that cell, and by revealing other cells, a process of elimination can allow for the pinpointing of the location of a mine.[3] "We generate a constraint for every revealed square on the board, whose scope is the set of squares adjacent to the revealed square"; this is known as the "Sum" constraint. [6]

Multiple forms of CSPs exist for Minesweeper such as Coupled Subsets CSP (CSCSP) which, as seen in Studholme's study, dynamically maintains a set of constraints, C. The CSCSP then partitions the set of constraints C, and performs backtracking searches as needed. Another form of CSP is Connected Components CSP (C3P); the CSP "divides search space into subsets of variables that share constraints... [but frames] the partition as a connectivity problem [which] allows for the solver to utilize existing algorithms to find connected components".[3] These two CSP implementations were compared by David Becerra in his focus on Algorithmic Approaches to Playing Minesweeper, and how a human or even AI can use logic to solve any given Minesweeper puzzle.

Shanghai Tech approached the Minesweeper problem as a propositional satisfiability problem with an infinite state space, which also follows the NP-complete theorem previously mentioned by Kaye. It groups propositional logic through algebraic systems of equations and uses sampling to get the number of mines that satisfy conditions in an equation set- further reducing a sample set in which a mine may be located,[15] however a study performed in Stanford found that when training a Minesweeper solver a CSP model works best compared to others such those of Shanghai Tech. The study compared solvers using "linear and logistic regression, reinforcement learning, as well as constraint satisfaction problems",[7] and by doing so, determined the dependability of CSP models. Similarly to that of Shanghai Tech, sum-variable methods were used to break n-ary constraints into unary and binary constraints, which further reduced the sample set where a mine or bomb may be located.

3 Strategies for Playing Minesweeper

Assuming all games played for Minesweeper exist on a 9x9 board and only 10 bombs are hidden around the board, one begins with the knowledge that there are 81 cells to begin with and each cell has the probability of 0.1235 of containing a mine - this changes for all cells as they are revealed. Of those 81 cells, there are three different types of cells: corner, edge, and central. Corner cells only have three total neighbors, edge cells have five, and central cells are surrounded completely with eight total neighbors. Neighbors are defined as any cells adjacent to an existing cell- they may be left, right, above, below, or diagonal from the current cell. Choosing carefully which cells to reveal initially and throughout the game may have a significant impact on the success rate for any agent.

3.1 The First Move

Often times players will choose a cell at random to be revealed with the intuition that since the probability of all cells at the start of the game will always be equal, in this case 0.1235, and there is no need for much thought when making that initial move, however it was found that impact of the initial action in an empty board aids in maximizing the success rate, specifically, probing of the corner cells. The idea behind this is that since there are only three adjacent cells, the probability of those surrounding the corner cells are impacted more significantly in that the probabilities of those blocks are likely to change more drastically. Equation 1 and figures a through f in the study shows that the upper-bound on the success rate of any corner-first strategy is higher than that of edge-first or central-first strategies- not beginning the game with the revealing of a corner cell first increases the probability of immediate death in the second move significantly. [4] So implementing this idea in a minesweeper solver in theory would increase its success rate immediately.

3.2 Algorithmic Strategies

Variables in CSPs, which we defined the Minesweeper game was a form of, are arc-consistent if every value in its domain satisfies the variable's binary constraints. [2] In some cases it is difficult to determine the arc-consistency of a certain cell in Minesweeper as part of the environment is still unknown to the agent, and there are cases where all cells have the same or similar probabilities of containing a mine and arc consistency has no effect on the domains of any surrounding cell. With this idea we may consider using the AC-3 algorithm.

Algorithm 1 AC-3 Pseudocode

```
1: while queue is not empty do
2:     (X_i, X_j) ←      function REMOVE-FIRST(queue).
3:     if          function REVISE(csp, X_i, X_j)  then
6:         if size of D_i = 0 then return false
7:             for  each X_k in X_i.NeighborsX_j do
8:                 add (X_k, X_i)to queue
9:
10:        return true =0
   function REVISE(csp, X_i, X_j)
2:     revised ← false
       for  each x in D_i do
4:         if no value y in D_jallows(x, y) to satisfy the constraint between X_i and X_j
   then
               deletex from D_i
6:             revised ← true
       return revised
```

In the case of our original Minesweeper program, we theorized that the AC-3

algorithm would be beneficial in the aid of determining what cells to open up next. It would consider the different arcs and probabilities of each cell then discard any inconsistencies for possible locations of mines and would run each time the environment updated. Unfortunately this theory is yet to be tested as we were unable to implement the algorithm into the original Minesweeper code that was put together, however with more time and assistance, we would like to continue with testing the theory in the future.

Other heuristic strategies were studied in depth like the naive approach which is to minimize the probability of uncovering a mine for a given configuration. The approach is though at maximizing the probability of opening at least one more block with minimum probability of uncovering a mine within one step. [4] This can be used alongside a probability calculator for surrounding cells. Should a single central cell be revealed with the value 1, it is known that there is a probability of 0.125 for each of the surrounding cells and it is far less "dangerous" to reveal cells around the cell marked as 1 versus a cell in a similar predicament marked anywhere from 2 through 7. (This goes to say that if a cell is marked 8, then there is a probability of 1 for each surrounding cell, meaning that all cells around it contain mines.)

4 Experiment Design & Implementation

Through our literature deep dive, we've learned Kaye's Minesweeper NP-completeness proposal is a core component of Minesweeper, given that every CSP solver we've researched thus far takes this facet (and Kaye's consistency theory) heavily into account. All CSP implementations we've seen thus far have a few generic commonalities in construction, including the variable set representing every square in the game's grid, each variable's domain being a Boolean (0 and 1 representing "safe" and "mine" squares). The general constraints include the total variable sum being the number of mines and the "sum of variables adjacent to any uncovered square must be equal to the number in the square." [7] What differs between these CSP executions is the constraint propagation methodology (backtracking vs forward checking, as well as value ordering heuristics). This basic framework (consistency and constraint definitions) gives us a good idea of how to implement our own CSP solver, and through our experiment we will enact the differing consistency-enforcement methods for this problem and compare and contrast their win rates.

4.1 Repl.It Attempt

Our initial Minesweeper game that we attempted to study was based heavily on existing code provided through the coding repository *Repl.It* [13] in a tutorial with an in-depth explanation on how to code it in Python. The main components we used were that the grid was represented as a set of arrays (two-dimensional Python lists), where each array represented an individual row and each index of the arrays represented a column of the grid, the game was always

Figure 5: First Move- Value 0

Figure 6: First Move- Value 1

randomized and utilized the random library to do so, and the game is won as soon as all cells that do not include mines are revealed, which is done with a function that consistently checks the whole board after every move that there are exactly ten uncovered cells left and no explosion was triggered.

We attempted to write a solver using the information found from dives in archives of preexisting solvers and theories, this included the implementation and adaption of AIMA code such as backtracking, AC-3, and forward-checking, however this became increasingly difficult as each cell must be checked and the value must be updated after each move a player makes.

Initially the idea was to set up the empty board with probabilities for each cell. Since the board was defined to be 9x9 with 10 mines, there are a total of 81 cells and each cell of an empty board had the probability of approximately 0.1235 of containing a mine. The next step would be to reveal a single corner cell as it was found in a study that this maximizes the rate of success, and we decided on keeping it simple for our case that the top left corner cell always be the first cell to be revealed. This of course did not always work (as seen in **Figure 9**) due to the placement of the mines being random; there is never a probability of 0 that a cell may contain a mine (in this case that it certainly won't) to safely reveal a cell in a completely empty board where no cell has yet been opened. After revealing the first cell and no mine was disturbed, the next idea was to recalculate probabilities for the whole environment in which the cells are not revealed. Some cases included that only a single block was revealed with the numbers 0, 1, 2, or 3.

In the case that 3 was revealed, it was always the simplest in determining which surrounding cells contained mines because it implied that all adjacent cells were in fact mines, making the probabilities of those adjacent cells equal to 1 and all remaining cells had the new probability of approximately 0.091. **Figure 8** gives a visual of where mines would be located when the initial move reveals the cell's value to be 3. When either 1 or 2 were the revealed values, **Figure 6** and **Figure 7** respectively, it simply changed the probability of all other cells possibly containing mines to 0.125. Since the rest of the environment contained all equal probabilities, arc-consistency would not have applied and

Figure 7: First Move- Value 2

Figure 8: First Move- Value 3

the AC-3 algorithm would not have helped in determining placement of cells when values 1, 2, and 3 were revealed and the solver would have to once again choose a random cell to open.

In the case that the cell revealed did not contain a mine and did not have any neighboring cells that contained mines (value was 0), **Figure 5**, the game would automatically reveal all other cells around it until cells that had a mine(s) surrounding them were revealed with their values. These cases were occasionally easy to work with and in other times, difficult. Should the revealed cells contain "corners" (where the cells create a corner of their own and the value on the inside of the corner is 1, such as cell C4 or G3 in **Figure 5**), the probability that such a corner contains a mine is 1 and again the probabilities of surrounding cells are reevaluated. This however became problematic as the probabilities of cells surrounding revealed cells impacted each other in different weights depending on their probabilities, and it began to be difficult to calculate those values for the solver to accurately make better decisions. We could have opted to simply pick another random cell, however the problem still existed that probabilities of each cell impacted the probabilities of others when some cells were revealed in the environment, so no algorithms would be of use since we did not know how to combat this issue.

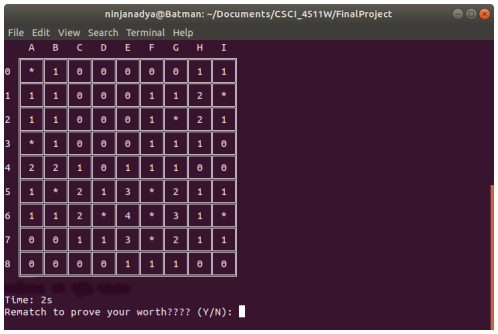

Figure 9: First Move- Immediate Loss

Upon further searches, we found that there did exist algorithms that could potentially be of use when calculating probabilities of all surrounding cells, however time was limiting in the attempt to adapt those algorithms to fit the already existing Python Minesweeper game we have written.

4.2 Final Approach

After facing limitations with our Repl.It Minesweeper rendition [13] and troubles adapting AIMA code to our object oriented setting, we ultimately settled on going off of the University of Toronto's Minesweeper code, formulated by Chris Studholme. Based on our research, we came up with a core system for the problem at hand:

- **Goal**: Given Minesweeper has an NxM board with K mines, we aim to uncover all M x N - K mine-free squares.
- **Mine Density**: Though other implementations tend to vary this, we chose to stick to Studholme's original 20% mine density for simplicity's sake.
- **Variables**: Every square in the game's grid, the domain 0 ("safe") and 1 ("mine"), i.e. a Boolean. The sum of all the variables is the total number of mines.
- **Constraints**: Every safe square yields a "sum constraint" over its 8 neighbors, meaning all variables adjacent to the uncovered square sum up to the number in the square. If all of a covered square's neighbors' values are known, the constraint can be simplified to an empty list and thrown out. [6]

$$x_{i,j} = \sum_{k,l} x_{k,l} \tag{1}$$

insert some sum constraint picture here

Equation 1 better illustrates this basic constraint, where $x_{i,j}$ in this case is a known constant, while $k \in \{i+1,i,i-1\}$ and is within the board's row and column bounds. These constraints, while proving effective for problem solving, require storing up to 70 tuples, which is hardly memory efficient. Studholme's basic implementation also includes what he terms an "Equation Strategy," which allows for further simplification of variables "by noting that one constraint's variables may be a subset of another constraint's variables" [7]. For example, knowing the constraints a+b+c+d=4 and a+b=2, it can be assumed c+d=2. After the non-trivial constraints have been simplified as much as possible, Studholme uses "coupled subsets" to divide the constraints on the basis of a common variable. [12] After enumerating solutions to all constraint subsets, we can analyze the solutions — In the case a square variable is 1 or 0 in all solutions, we can either probe or flag said square with certainty. However, we still have many squares

left over, and we will need to use a probability-based "crap shoot" algorithm for certain situations where solutions can't be used and "there is no possibility of new information ever making the guess easier or eliminating it entirely... it's a 50/50 guess" [12]. In essence, we are using a highly modified backtracking algorithm with value ordering using the Minimum-Remaining-Values (MRV) heuristic. The more contrained a variable, the earlier they are assigned. "Each variable is tested with a 0 first, then a 1. Each assignment is checked for consistency... if the assignment is inconsistent, then backtracking occurs" [7]. Once all the variables are assigned, a possible solution is found. The algorithm then backtracks again to find the next solution.

insert crapshoot picture

Algorithm 2 Studholme's 'CSP Strategy' Pseudocode

```
    Input: Board after first click
    Output: Solved Board
    Global Variables: ConstraintList (initially empty)
    Local Variables: 3 SolutionSet variables (LargestNumOfConstraints,
    LargestNumOfSolutions, & LargestNumofEquations) all set to 0
    function CSPSTRATEGY(Board)
2:      initialize ConstraintList through double for loop through Board
        while !DONE(Board) do                          ▷ while game isn't over
4:          SIMPLIFY_CONSTRAINTS(ConstraintList)
            Subsets = COUPLE_CONSTRAINTS(ConstraintList)
6:          if len(Subsets) ≤ 0 then
                all unknown squares separated (by mines) from known squares
8:              i.e. no problems to solve
            else
10:             for subset in Subsets do
                    get # of variables in largest subset
12:                 ENUMERATE_SOLUTIONS(subset)          ▷ solves subset
            CRAPSHOOT(ConstraintList)
    return Board =0
```

Though the "CSP Strategy" provided countless interesting results, we decided a comparison to other constraint propagation strategies was in order. We decided to try implementing our own varying propagation methods within a simple backtracking search (such as forward checking and generalized arc consistency), as well as going off of Stanford University students Trevor Howard and Pavan Mehrortra's "fringe square" theory, where the only variables are "fringe squares," i.e. every "covered square adjacent to uncovered [squares]" [14].

As a refresher, the forward checking form of inference establishes arc consistency for any variable X by deleting from any unassigned variable (connected to X) Y's domain values inconsistent with X. For our implementation, we checked constraints with only one uninstantiated variable and kept track of all constraint pairs and pruned variables. Though we opted not to value order, this algorithm

ended up being surprisingly efficient. As we looped through, we tested each value in a variable's domain and used a list "vals" within our for loop to keep track of all the assigned values of other corresponding variables in the constraint's scope, then used our CONSTRAINT_CHECK() function to see if all values assigned pass the constraints placed on the squares, otherwise the value assigned will be pruned.

Algorithm 3 Forward Checking Pseudocode

```
    Inputs: ConstraintList
    Local Variables: PrunedList (initially empty)
    function ForwardChecking(ConstraintList)
        for constraint in ConstraintList do
3:          if constraint only has 1 assigned var then
                variable = constraint's unassigned var
                for domain in variable's domain list do
6:                  assign domain to variable
                    vals = []
                    for var in constraint's scope do
9:                                  ▷ loop through all associated values in constraint
                        append var's assigned value to vals
                    if !CONSTRAINT_CHECK(vals) then
12:                     prune domain value from variable domain
                        append domain value to PrunedList
                        if variable's domain list empty then return False, PrunedList
15:
                return True, PrunedList
```

Our Generalized Arc Consistency (GAC) method, also known as 1-Relational Consistency, is similar to the University of Nebraska-Lincoln's Constraint Systems Laboratory's Minesweeper application. This algorithm "considers every constraint independently, and attempts to infer the values of the unexplored squares in the scope of the constraint from the values of the explored ones" [6]. A variable is only considered arc-consistent if all the values in its domain satisfy its binary constraints, i.e. constraints with only one other variable.

Finally, we attempted the "fringe square" technique, where the only variables were the covered squares neighboring every uncovered square, given that the "fringe square" is not a potential mine. [14] The domain remained the same, however the sum constraint was only employed when there were less than 20 squares. In this case, the sum of an uncovered square's surrounding squares is the uncovered square's number, minus the number of adjacent flags. In this case, the number of variables was minimized, moreso improving storage memory over anything else.

insert fringe square picture

Algorithm 4 GAC Pseudocode

```
Inputs: ConstraintList
Local Variables: PrunedList (initially empty), Queue (initially goes off of ConstraintList
function GAC(ConstraintList)
    while ConstraintList not empty do
        constraint = REMOVE_FIRST(ConstraintList)
4:      for var in constraint's scope list do
            for domain_val in var's domain list do
                if constraint doesn't support domain_val then
                    prune domain_val from variable domain
8:                  append domain_val to PrunedList
                    if variable's domain list empty then return False, PrunedList
                    elseadd constraints with the changed variable to Queue
            return True, PrunedList
```

4.3 Results

We used three primary metrics to gauge a CSP winning strategy's success: Average CPU time, win ratio, and average percent of board uncovered. We performed 50 trials on each method with 3 levels of difficulty — "Easy" denotes an 8x8 board with 9 mines, "Intermediate" refers to a 16x16 board with 40 mines, and "Advanced" concerns a 24x24 board with 99 mines. We kept the mine density constant.

some list format here average CPU time, win ratio, timeout percentage wonder if there's a way to hybridize GAC studholme's CSP strategy

5 Conclusion

Through our research, we discovered a number of vastly differing Minesweeper solvers, many using algorithmic or statistical alternatives to CSP. A group at Stanford University did a comparison of solvers that utilize linear and logistic regression methods, reinforcement learning, and a CSP implementation. The former two methods, though out of our scope when formulating this project, proved far less effective than the CSP model. The first supervised learning method used "the local board configuration to classify an uncovered square," and the classifier uses a feature extractor that extracts non-separable data. This data trained a logistic regression to predict uncovered square behavior. The Stanford group's second approach included modified Q-learning while modeling Minesweeper as a Markov Decision Problem (MDP), which was moderately successful but had a high time complexity. [7] Similarly, University of California-Berkeley's Preslav Nakov and Zile Wei also treated Minesweeper as a partially observable MDP, using a CNF formula to compute the state transition proba-

bility and state value function. [11] Others at Université Paris-Sud used Upper Confidence Trees (UCT) and Direct Policy Search (DPS, an anytime algorithm). [9] At the AAAI-17 Workshop, Chen et al. used search algorithms with the PSEQ heuristic to probe game board cells. [4] Ultimately we chose not to further delve into these alternative sources given that they give little help with constructing a Minesweeper CSP, i.e. the subject of our final project (though they did spark our curiosity for future side projects).

References

[1] Iris van Rooij Allan Scott, Ulrike Stege. Minesweeper may not be np-complete but is hard nonetheless, 2011. The Mathematical Intelligencer.

[2] Stuart J. Russell, Peter Norvig. Artificial intelligence. In Marcia Horton, editor, *A Modern Approach Third Edition*, chapter 1, pages 8–9. Pearson Education, New Jersey, 2010.

[3] David Becerra. Algorithmic approaches to playing minesweeper, 2015. https://dash.harvard.edu/bitstream/handle/1/14398552/BECERRA-SENIORTHESIS-2015.pdf?sequence=1.

[4] Shiteng Chen Chong Zu Zhaoquan Gu Jinzheng Tu, Tianhong Li. Exploring efficient strategies for minesweeper, 2017. https://www.aaai.org/ocs/index.php/WS/AAAIW17/paper/download/15091/14775.

[5] Richard Kaye. Minesweeper is np-complete, 2000. http://simon.bailey.at/random/kaye.minesweeper.pdf.

[6] Josh Snyder Ken Bayer and Berthe Y. Choueiry. An interactive constraint-based approach to minesweeper, unknown. https://www.aaai.org/Papers/AAAI/2006/AAAI06-344.pdf.

[7] Ryan Silva Luis Gardea, Griffin Koontz. Training a minesweeper solver, 2015. http://cs229.stanford.edu/proj2015/372$_{r}eport.pdf$.

[8] Elchanan Mossel. The minesweeper game: Percolation and complexity, 2002. https://www.stat.berkeley.edu/ mossel/publications/mine$_{s}weeper.pdf$.

[9] Woanting Lin Olivier Teytaud Olivier Buffet, Chang-Shing Lee. Optimistic heuristics for minesweeper, 2012. https://hal.inria.fr/hal-00750577v2/document.

[10] Kasper Pedersen. The complexity of minesweeper and strategies for game playing, 2003-2004. https://pdfs.semanticscholar.org/2083/3f71d74ff26ffd18979796cf1cbc8b3d92e6.pdf.

[11] Zile Wei Preslav Nakov. Minesweeper, minesweeper, 2003. https://www.researchgate.net/profile/Preslav$_{N}akov/publication/228613592_{M}inesweeper_{M}inesweeper/links/$
$Minesweeper.pdf$.

[12] Chris Studholme. Minesweeper as a constraint satisfaction problem, 2007. http://www.cs.us.es/cursos/ia1-2007/trabajos/trabajo-2/minesweeper-toronto.pdf.

[13] ThomasS1. How to program minesweeper in python, 2018. https://repl.it/talk/learn/How-to-program-MineSweeper-in-Python-fully-explained-in-depth-tutorial/9397.

[14] Pavan Mehrortra Trevor Howard.

[15] Yanpeng Hu Yimin Tang, Tian Jiang. A minesweeper solver using logic inference, csp and sampling, 2018. https://arxiv.org/abs/1810.03151.