CSCI4511W Final Project: Applying Logic Inferencing into CSP Solver for Minesweeper Problem

Linh Duong (duong172)
Hinda Omar (omarx030)

December 2020

Abstract

Minesweeper a well-known computer game as well as an NP-problem which is popular among the Artificial Intelligence community. The problem we are solving in this project is the creation of an automated player for the game Minesweeper. Our chosen approach for an automated Minesweeper player is through the use of logic inferencing combined with backtracking search to provide an improved algorithm for CSP solver. Our improved algorithm is called Bonus CSP solver, which assumed the probability of randomly opening a new cell that is not a mine is 80%. We compared the basic CSP solver and the Bonus CSP solver based on each algorithm's runtime and memory usage. The experiment shows that the Bonus CSP solver performs significantly faster than the basic CSP solver, especially the runtime increase exponentially with the board size. However, the memory usage used by Bonus CSP solver is slighter larger than that of basic CSP solver due to more inferencing equations for more precise calculation of heuristics. Finally, we discuss about how Bonus CSP solver is unstable for a small board size.

1 Introduction

Minesweeper is a game that is familiar to many people as it is included in certain operating systems used on computers, most famously in Windows. Minesweeper is a single-player puzzle game in which the purpose is for the player to clear the playing board without touching any of the hidden mines. At the start of a game, the board consists of $n \times m$ untouched cells. The player clicks one of these cells to uncover it, revealing either a number or a mine. The number of the cell indicates the number of mines directly surrounding that cell. The player uses this information to decide which cells might be mines; they are able to "flag" suspected cells to keep track of mine cells. Upon discovering a mine, the player loses and the game ends. The goal is to uncover every non-mine cell, which results in a win for the player. The game is played using logic but there are

instances where it is impossible to deduce the content of any cell and therefore guessing is necessary.

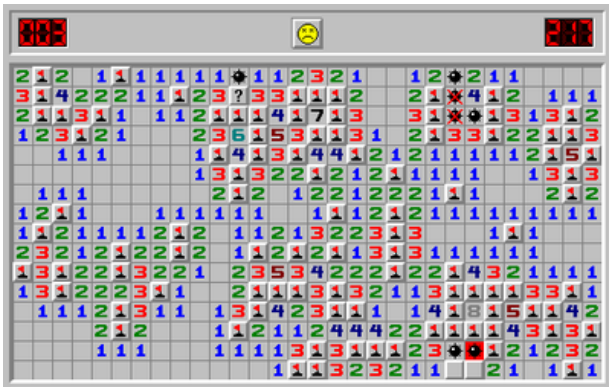

Figure 1: A loss state of a minesweeper game

For our project, we chose to focus on the creation of an automated player for Minesweeper. Having knowledge of the rules, we can construct some constraints for the problem and compute an agent that will search the gamespace to find the best moves based on the current state of the game. *Figure 1* represents an instance of a state of what the "board" looks like at a point in time, which includes the numbers, flags, and untouched squares on the board. However, this state represents a loss since the player has clicked on the mine. As a result, we decide to investigate an approach which allows us to solve this minesweeper problem.

Our approach to solve this is through the use of logic inferences combined with Constraint Solving (CSP) to improve the behavior of a normal CSP approach. Through out the paper, we discuss about its performance in terms of runtime, memory usage and optimality comparing to a normal CSP approach.

First of all, we discuss how the minesweeper game is generally solved. In particularly, we discuss about how "Logic Inference" can be used to make equations for CSP to be applied in the minesweeper game. The research paper "*A Minesweeper Solver Using Logic Inference, CSP and Sampling*"[10] from Shanghaitech University will give us an overlook of their general strategy for applying CSP to minesweeper game. This strategy is very important because it will act as a base strategy for other researchers to understand and apply CSP into solving minesweeper. Furthermore, it is also one of the key thing we decide to change to improve our CSP approach.

Second, after generalizing the problem and CSP approach, we then discuss about the advantages and limitations of their current approach. In the research paper [10] and in "*Optimistic heuristic for minesweeper*" [3] by Olivier Buffet and his colleagues, they both discuss about the downside of using current equations for CSP in the context of minesweeper problem. Additionally, in paper [3], the authors provide an alternative approach to improve the heuristics of

UCT-based algorithm. By combining the UCT and CSP solvers, they could drastically improve the performance and runtime of the algorithm.

Next, we discuss about alternative approaches for solving minesweeper game. While [3] provides an improvement of CSP algorithm by combining CSP with heuristics, the research paper "*Upper Confidence Tree-Based Consistent Reactive Planning Application to MineSweeper*" [8] by Sebag, M. and Teytaud, O. will compare and analyze 3 different approaches: Rejection Method, Constraint Solving and Markov-Chain Monte-Carlo Approaches. By providing some insights of these 3 approaches, paper [8] then provides variants of CSP approaches and modifies them into Upper Confidence Tree-Based Consistent Reactive Planning (UC-CRP) so that it would be suitable for solving an ***expert*** minesweeper game. In other words, this algorithm will focus on the difficulty of the game rather than how quick it can solve the game.

Finally, after discussing about different approaches of solving minesweeper game, we decided to choose one algorithm provided in [3], [8] and combined with our change in logic inference used by [10] to improve the chosen algorithm. We will collect the data and analyze them in terms of:

a) Runtime

b) Memory

c) Optimality

Our experiment will be conducted under a wide range of variables (board size and mine density) and repetitions to ensure the reliability of the experiment.

After examining existing approaches relating to the Minesweeper problem, we will then describe our chosen approach as well as our experiment design. We will experiment by running the automated player with different algorithms and then display and analyze our results. We will attempt to determine the better algorithm in terms of runtime, memory usage, and accuracy in playing the game.

2 Related Work

2.1 Minesweeper is NP-Complete

Minesweeper is an NP-complete game. Richard Kaye discusses the NP-completeness of Minesweeper in [4]. In this paper, Kaye describes what NP is, what it means for a problem to be in NP, and why the Minesweeper problem is NP-complete. In [7], Rodríguez-Achach and Coronel-Brizio calculate the probably of winning a game of (one-dimensional) Minesweeper taking into account the size of the board and number of mines on the board.

2.2 Solving Minesweeper

2.2.1 Inference in Minesweeper

In "*A Minesweeper Solver Using Logic Inference, CSP and Sampling*" [10], Yimin Tang and his colleagues discussed about the use of logic inference in

minesweeper puzzle, which was illustrated in Figure 2. According to their strategy, they define that each point on the board can be a value from 0-9 or a mine. If it's not a mine, then they have equation $x1 + x2 + x3 = 2$. And if there is another equation $x2 + x3 = 1$ then the new equation will replaced the old one. The procedure is repeated until no more equations can be generated for that same point on the board.

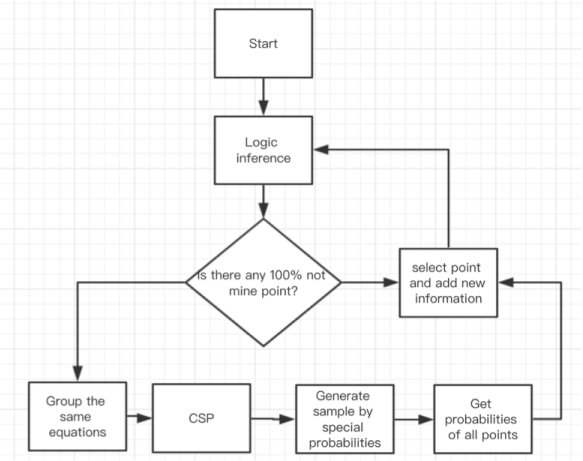

Figure 2: Logic Inference in Minesweeper

While [10] using logic inference for minesweeper, Vomlel, J. and Tichavsky, P. proposed an approximate probabilistic inferencing based on Canonical Polyadic tensor (CPT) decomposition in their research paper "*An Approximate Tensor-Based Inference Method Applied to the Game of Minesweeper*" [11]. In their strategy, they defined each field to have 2 variables. One variable is binary corresponding to state of that field (state 1 if there is a mine and 0 otherwise). Another variable conveys the state of neighbors positions as its parents. In other words, it conveys the number of its neighbors with a mine. Its CPT is defined by additional functions and observations based on corresponding states variables can be removed from the networks of functions if the states are known. The overall state is 1 mean the game is over and 0 otherwise.

Both papers discussed 2 different inference methods, however, we will use the logic inference for our research paper because it is simple enough for applying CSP strategy. On the other hand, the CPT inference are useful for reducing a computational load, which would be very beneficial if our minesweeper puzzle becomes more complicated, for instance, if the field has more than 8 neighbors.

2.2.2 CSP strategy

In "*Minesweeper as a Constraint Satisfaction Problem*"[9], Studholme explained 7 steps of CSP strategy to solve the Minesweeper puzzle:

Step 1: All positions are represented as boolean values, 0 if there is no mine and 1 if there is a mine. The constraints are the sum of neighboring variables (for example $x1 + x2 + ...$). If neighboring variables are known, constraints can be simplified. If all neighboring variables are known, the constraints will be empty and discarded. CSP strategy maintains the set of constraints by adding and remove the constraints.

Step 2: Using logic inferencing to simplify the constraints. For example, given $x1 + x2 + x3 + x4 = 2$ and $x2 + x3 = 1$, then we will have $x1 + x4 = 1$ and $x2 + x3 = 1$. By simplifying these, some constraints may become trivial and CSP strategy can return to step 1 again.

Step 3: Given a set of non-trivial constraints, these constraints will grouped into smaller coupled subsets where 2 coupled constraints have common variables. Backtracking algorithm is used on these coupled subsets to find all possible solutions.

Step 4: Once all possible solutions have been found, the solutions will be analyzed to see if there is a case where the location of the mine is obvious. If there is such instances, then the location is marked and it can move on to step 5. If there is no such instances, then the new constraints can be added to the constraints set and it is immediately return to step 1 for a new round of simplifying and expanding set of constraints.

Step 5: The marked mine in step 4 is only a guess because it provides no new information added. Thus another 50/50 guess to provide new information for the other fields.

Step 6: The probability that a field has no mine is calculated by taking the number of solutions where that variable has a value of 0 and dividing it by the total number of solutions found for that particular coupled set of constraints. The variable with the highest probability of being clear is considered the "best guess" among the constrained squares for the next move.

Step 7: When a field was guessed, CSP checks if there are any unconstrained corner field remaining and if there are some, a random one is chosen. If there are none, a random unconstrained, noncorner, edge field is the next best choice. If none of these are available, then an interior unconstrained field needs to be chosen.

2.3 Alternative approaches

There have been many varying approaches in regards to the Minesweeper problem. In [1], Andrew Adamatzky approaches solving Minesweeper by constructing a cellular automaton to determine which squares are mines.

While [9] provides a simple CSP strategy approach, many other researches attempt to improves this strategy. A research paper "*Optimistic heuristic for minesweeper*" provides combining UCT with CSP to improve the behavior of UCT in long term aspects of the minesweeper problem. While the research paper "*Upper Confidence Tree-Based Consistent Reactive Planning Application to Minesweeper*"[8] provides UCT-based consistency reactive planning application to minesweeper game. This paper also discuss and provide pseudo code for:

a) rejection method
b) Constraint Solving (CSP) and
c) Markov-Chain Monte-Carlo Approaches to the minesweeper game.

This paper also discusses about how UCT can combine with approaches above to remove some constraints in minesweeper games.

Additionally, Jean Me hat and Tristan Cazenave also discussed about the use of UCT and Nested monte-Carlo Search in Single Player General Game Playing[5]. In particularly, this paper proves to have improved performance between their new search with the old traditional CSP searches.

In the research paper [10] and in "*Optimistic heuristic for minesweeper*" [3] by Olivier Buffet and his colleagues, they both discuss about the downside of using current equations for CSP in the context of minesweeper problem. Additionally, in paper [3], the authors provide an alternative approach to improve the heuristics of UCT-based algorithm. By combining the UCT and CSP solvers, they could drastically improve the performance and runtime of the algorithm.

While [3] provides an improvement of CSP algorithm by combining CSP with heuristics, the research paper "*Upper Confidence Tree-Based Consistent Reactive Planning Application to MineSweeper*"

[8] then provides variants of CSP approaches and modifies them into Upper Confidence Tree-Based Consistent Reactive Planning (UC-CRP) so that it would be suitable for solving an ***expert*** minesweeper game. In other words, this algorithm will focus on the difficulty of the game rather than how quick it can solve the game.

2.3.1 Reinforcement Learning

Minesweeper is a partially observable Markov Decision Problem. Nakov and Wei show that, however, it can be defined as a fully observable Markov Decision Problem with an appropriate state space[6]. In their paper "*Minesweeper, #Minesweeper*", Nakov and Wei use the fact that the Minesweeper problem is NP-complete to define a corresponding (counting) problem, which they name #MINESWEEPER, and prove that it is #P-complete. They define the #MINESWEEPER problem as such:

> *#MINESWEEPER: Given a particular minesweeper configuration find the number of the possible mine distributions over the covered squares.*

After describing model counting, #MINESWEEPER and #P-completeness, the authors discuss several algorithms and explain that they are using an implementation of DDP ("Decomposing Davis-Putnam") algorithm. They then define the minesweeper problem in the reinforcement learning framework and describe the state space problem: the fact that there is a very large state space, making it difficult to solve any problem that isn't small enough. They list techniques to cut the search space without sacrificing an exact solution.

2.3.2 Adapted UCT

Auger, Couëtoux, and Teytaud, in their paper [2] use the case of the fully observable MDP to prove consistency in infinite action space, among other factors. Upper Confidence Trees (UCT) has been shown to provide consistent solvers for games, in the finite case. The authors adapt UCT to be consistent in the more general finite case, including infinite action and state spaces. [2] They describe the MDP setting (including a state space and action space, tree nodes, etc) and define in detail the adapted UCT algorithm.

2.3.3 TeddySweeper

In [13], Wu, Lin, Yen, and Chen propose a Minesweeper solver named TeddySweeper. Single Point Strategy and Equation Strategy are common strategies used to solve while playing Minesweeper by a human player, while a computer can work with an additional strategy, CSP. [13] However, they say that CSP would consider all the possible mine configurations, making it inefficient due to the exponential number of possible configurations when the number of untouched cells is high. The authors' proposed method of finding mines, which they call TeddyMethod, calculates the minefield configurations of each cell (using a formula which they provide). The calculations also consider the probability of a cell being a mine. Their method implements a divide-and-conquer algorithm, which can not always be applied (there are some situations for which it would not work), but generally it reduces the number of combinations to calculate.[13]

3 Approach

3.1 Description of approach

Our approach to solve this minesweeper problem is to apply the logic inferences described from [10] into the CSP from [12] for minesweeper.

For a basic CSP approach:

Our agent keep track of 2 lists, a list of all cells which it think as having mines ($mineVariables$) and a list of all safe cells ($nonMineVariables$). The agent will play games in 3 phases recursively: Basic solver, Subset solver and Random opening solver. In **Basic solver**, its task is to solve the board without any equations from logic inferences. To do this, it will have 3 choices of actions:

- Click all non-mine-cells: the agent will click a non-mine cell. When the cell open, it will form a constraint equations for the variable which including its neighbors and equation value. Additionally, since the open non-mine cell can't be a mine, it will remove from all constraint equations from other variables.

- Flag all mines: If the number of variable is equal to the value of equation (i.e $A + B + C = 3$), then the agent flags all the cells and

removes the cell from constraint equations of other variables.

- Check the equation lists for mine cells and non mine cells: the agent yields more mine and non-mine variables for the agent to flag or open. For each step of removing the non-mine variables and mine variables, the agent will check its knowledge base to see if there is any equation with all non-mine variables (i.e $\sum_{i=1}^{n} a_i = 0$) or if there is any equation with all mine variables (i.e $\sum_{i=1}^{n} a_i = n$)

In **Subset Solver**, its task is to break down the subsets of equations when the basic solver is no longer able to yield more equations. This agent combines the subsets from knowledge base and start to solve them (i.e if $a_1 + a_2 + a_3 = 2$ and $a_1 + a_2 = 1$ then a_3 must be a mine)

In **Random opening solver**, its task is randomly open on a cell to make and deduce new equations when subset solver cannot deduce anymore equations. For instance, if $a_1 + a_2 + a_3 + a_4 = 3$ and $a_2 + a_3 = 1$, then subset solvers cannot deduce, resulting in agent need to randomly open a new cell. The next open cell need to be based on the heuristic calculate from the probability of any cell having a bomb. The smaller the value of heuristic, the more likely for the cell to be opened. The probability is calculated by equation:

$$P(isMine(A_i)) = \frac{\sum X_i \times \frac{n!}{i!(n-i)!}}{\sum \frac{n!}{i!(n-i)!}}$$

After recursively running these 3 phases, a knowledge base can be formed. In short, to solve the minesweeper problem, the agent will need to use the following rules:

- Use basic solvers for checking mine cells and non mine cells. If it is guarantee that the cell is mine, or non-mine, then the agent can flag or open it respectively. If nothing else can be deducted, proceed to subset solver.

- Solve subsets by breaking down equations the repeat the basic solvers steps. If no more subsets can be deducted, then proceed to random open solver

- Click on random cell to open based on the heuristic and probability of that cell being a mine

It is worth noting that the risk of randomly opening the cell is high when the mine density is high.

For a Bonus CSP approach: This approach is similar to the CSP solver above. However, instead of the agent can only know whether the cell is mine or not after opening it, the agent will have 80% of randomly opening a cell and get it correct.

3.2 Software and code used

We decided to use the code provided by Aditya Vyas. The code for minesweeper game can be found on his Github repository [12]. In his repository, a code for basic agent, normal CSP agent, chain CSP agent and improved bonus CSP agent is provided for solving minesweeper games. Furthermore, in his code, he had also defined the environment for the minesweeper game. As there is already existing code for minesweeper environment and a few algorithms to solve the game, we could easily modify this code to fit our particular designed problem and chosen algorithms in order to collect the data and analysis. The necessary code and modified code can be seen in 8.

4 Experiment

4.1 Experiment Design

Our experiments involved with 2 chosen algorithms discussed in our previous sections. We had modified the code to be run our 2 chosen algorithms on the same generated minesweeper board to keep a fair initialization properties. To measure time, we imported ***timeit*** package from python 3:

```
from timeit import default_timer as timer
start = timer()
    # running the algorithm
end = timer()
print(end - start)
```

This experiment was repeated for 3 trials on a wide range of different board size (10x10, 11x11, 12x12, 13x13, 14x14, 15x15) with 2 different mine density (0.1 and 0.5). The relationship between board size/density with time was determined.

To collect the data for memory usage, we decided to use a SSH with VmPeak.

```
ps -u <studentID>
grep VmPeak /proc/<PID>/status
```

These command lines will be done simultaneously on 2 SSH windows of the same lab machine. Since the time taken for Bonus CSP to play the minesweeper game is small (less than 1 second), we decided to included a pause of 20 seconds before the program is terminated. We compared the memory usage for 2 chosen algorithms on a wide range of board size (10x10, ... , 18x18, 19x19, 20x20). To keep it simple, we decided only experiment it with $minedensity = 0.5$

Finally, to compare the optimality of each algorithm, we ran the program on each algorithm repeatedly over 3 trials run. The number of success and fail would be recorded for each algorithm and the probability of solving the problem can be calculated. The comparison of optimality can be calculated based on the comparison between the probability of solving the game.

4.2 Experiment Results

Table 1, 2, 3 show the results of running the CSP and Bonus CSP algorithms, comparing them based on average runtime and memory usage. The average is calculated over 3 trials for each board size and algorithms.

5 Analysis

Table 1 shows the result of the runtime for both CSP and Bonus CSP over different board size. It is clearly that the Bonus CSP solver is faster than the basic CSP solver. Especially, the time taken for CSP solver to solve the minesweeper problem is linearly proportional to its board size while the time taken for Bonus CSP solver to solve the problem is exponentially proportional to its board size. This is understandable because the Bonus CSP Solver has a higher change of randomly opening a new cell correctly, in which the change is increase to 80%. Therefore, the Bonus CSP solver is likely to make more new equations easily and deduce more equations to smaller subset to inference whether the current cell is a mine or not.

On the other hand, it is worth noting that the average time is calculated over at least 3 trials run. We notice that the average time taken for Bonus CSP solver to solve the small board size (shown in 11x11 and 12x12 in Table 1) is slightly faster than 10x10 board size. This implies that the bonus CSP algorithm may not be stable to be run on smaller size board. Perhaps, an explanation for this is because the probability for opening a correct cell is assumed to be increase to 80%, which affect the heuristic for that current cell. However, the bigger the board size, the less difference between the probability calculated by CSP solver and by Bonus CSP solver. As a result, as the board size increase, the Bonus CSP algorithms is likely to have a more precise calculation of the heuristic and the probability of whether the cell is a mine or not.

Table 2 shows the result of the runtime for both CSP solver and Bonus CSP solver on a higher density. It is clearly that the higher the density, the longer the time taken to solver the problem when comparing with Table 1. This is understandable because the more mines on the board, the more equations need to be deducted. This is because it is more difficult to deduct $\sum_{i=1}^{n} a_i = 0$ since there is higher mine density, thus less likely to guarantee that the cell is a non-mine unless more equations are added to knowledge base. However, it is worth noting that the average runtime for Bonus CSP is more stable for smaller size board comparing to when mine density = 0.1. Perhaps, an explanation for this is because since there are more equations added to the knowledge base, the agent doesn't depend on randomly open the new cell because it can deduce more smaller subsets based on existing knowledge base.

Finally, Table 3 shows the result of memory usage for both CSP solver and Bonus CSP solver. One interesting thing we notice from it is that the memory usage for both CSP and Bonus CSP solvers are the same for the board size of $n = 10, 11, 12, ...17$. Perhaps, one explanation for this is because there are equal

numbers of inferenced equations in CSP solver and Bonus CSP solver. We also notice that this Table 3 combines with Table 1 may imply one important thing, the Bonus CSP solver may be unstable for solving the smaller board size.

On the other hand, we also realize that the memory usage for Bonus CSP solver is slight larger than that of the CSP solver. This implies that there are more logic references deducted from Bonus CSP than that of CSP solver, which results in more memory to calculated calculated the probability and stored the heuristics for the equations.

In short, from the experiments, we have deducted that:

1. Bonus CSP solver is faster than basic CSP solver
2. Bonus CSP solver uses slightly more memory usage than basic CSP solver
3. Bonus CSP solver is unstable when being run on a small board size ($n < 17$)

6 Conclusion

In this paper, we discussed the various methods and approaches to solving the game Minesweeper. Though we discussed alternative approaches in related work, our focus here was on using inference and solving minesweeper as a constraint satisfaction problem. We used existing code by [12] with two agents, the "CSP Agent" and "Bonus CSP Agent", in order to run the two algorithms and compare them. Both CSP Agent and Bonus CSP Agent use the logic inference, however, Bonus CSP Agent has additional information in the knowledge base which increase the probability of choosing a cell that is not a mine up to 80%. The result of the experiment shows Bonus CSP Agent is faster than CSP Agent, however it slightly use more memory usage than the CSP Agent. Additionally, we also note that the Bonus CSP Agent is unstable in a simple, smaller board size yet display a significant power when being run on a bigger and more complex board size.

In the future, it might be worthwhile to apply the CSP with another search algorithm such as UCT as well as other general gameplay algorithms for comparison. To do this, we would have to write our own code or heavily modify the existing code.

7 Teamwork

Linh

- Researched papers to use and summarized them
- Designed and ran the experiment and did the analysis
- Worked on sections: Introduction, Related Work, Approach, Experiment, Analysis

Hinda

- Researched papers to use and summarized them
- Worked on sections: Introduction, Related Work, Conclusion

References

[1] A. Adamatzky. How cellular automaton plays minesweeper. *Applied Mathematics and Computation - AMC*, 85:127–137, 09 1997.

[2] D. Auger, A. Couëtoux, and O. Teytaud. Continuous upper confidence trees with polynomial exploration – consistency. In H. Blockeel, K. Kersting, S. Nijssen, and F. Železný, editors, *Machine Learning and Knowledge Discovery in Databases*, pages 194–209, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[3] O. Buffet, C.-S. Lee, W.-T. Lin, and O. Teytuad. Optimistic heuristics for minesweeper. In R.-S. Chang, L. C. Jain, and S.-L. Peng, editors, *Advances in Intelligent Systems and Applications - Volume 1*, pages 199–207, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[4] R. Kaye. Minesweeper is np-complete. *The Mathematical intelligencer*, 22(2):9–15, 2009.

[5] J. Méhat and T. Cazenave. Combining uct and nested monte-carlo search for single-player general game playing. *IEEE Transactions on Computational Intelligence and AI in Games*, 2, 01 2011.

[6] P. Nakov and Z. Wei. Minesweeper, minesweeper. 04 2003.

[7] M. Rodríguez-Achach, H. F. Coronel-Brizio, A. R. Hernández-Montoya, R. Huerta-Quintanilla, and E. Canto-Lugo. The one-dimensional minesweeper game: What are your chances of winning? *International Journal of Modern Physics C*, 27(11):1650127, 2016.

[8] M. Sebag and O. Teytaud. Upper confidence tree-based consistent reactive planning application to minesweeper. In Y. Hamadi and M. Schoenauer, editors, *Learning and Intelligent Optimization*, pages 220–234, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[9] C. Studholme. Minesweeper as a constraint satisfaction problem. *Unpublished project report*, 2000.

[10] Y. Tang, T. Jiang, and Y. Hu. A minesweeper solver using logic inference, csp and sampling. *arXiv preprint arXiv:1810.03151*, 2018.

[11] J. Vomlel and P. Tichavský. An approximate tensor-based inference method applied to the game of minesweeper. In L. C. van der Gaag and A. J. Feelders, editors, *Probabilistic Graphical Models*, pages 535–550, Cham, 2014. Springer International Publishing.

[12] A. Vyas. Machine learning and data-science/minesweeper ai bot.

[13] T. Wu, C.-N. Lin, S. Yen, and J.-C. Chen. Teddysweeper: A minesweeper solver. 2013.

8 Appendix

8.1 Basic CSP Solver Function

```
class CSPAgent():
  def play(self):
    non_mine_variables.append(env.variable_ground[0, 0])

    while(True):
      self._basic_solver()

      # Condition to end the game
      all_flags_equal_to_mines =
      list(zip(*np.where(self.env.mines))) ==
        list(zip(*np.where(self.env.flags)))
      all_clicked = np.all(self.env.clicked)

      # If we are stuck at a game position, where no new
      # moves can be made then we just click randomly.
      if self.game_stuck:
      self.game_stuck = False
      self._click_random_square()

      if self.env.mine_hit:
        self.game_won = False
        return

      if all_clicked:
        if all_flags_equal_to_mines:
          self.game_won = True
        else:
          self.game_won = False
          return

        if self._check_solvable_csp():
          self._resolve_subsets()

        # If everything fails, then click randomly
          if self._check_solvable_csp():
            self._click_random_square_with_heuristic()
```

8.2 Runtime measurement for CSP and Bonus CSP Function

```
env = Environment(n = 10, mine_density = 0.5,
      end_game_on_mine_hit = False, visual = False)
env.generate_environment()

agent1 = CSPAgent(env = env,
                  end_game_on_mine_hit = False)
agent2 = BonusCSPAgent(env = env,
                       end_game_on_mine_hit = False)

start = timer()
agent1.play()
end = timer()
print("CSP: %20.12f" %(end - start))

start = timer()
agent2.play()
end = timer()
print ("Bonus CSP: %0.12f" %(end - start))
```

8.3 Tables

Board size	Average Runtime	
	CSP $\pm 0.001(seconds)$	Bonus CSP $\pm 0.000001(seconds)$
10x10	0.183494482	0.000079378
11x11	0.241266758	0.000077523
12x12	0.306558129	0.000079211
13x13	0.371913170	0.000082412
14x14	0.614038556	0.000087775
15x15	0.716012936	0.000086963

Table 1: Average runtime of CSP and Bonus CSP for $minedensity = 0.1$

Board size	Average Runtime	
	CSP $\pm 0.001(seconds)$	Bonus CSP $\pm 0.000001(seconds)$
10x10	2.464043397457	0.000098150224
11x11	3.477617460924	0.000101075818
12x12	5.084935908206	0.000107647851
13x13	6.961196067743	0.000129998662
14x14	9.378334048204	0.000126498441
15x15	12.518584681985	0.000128227286

Table 2: Average runtime of CSP and Bonus CSP when $minedensity = 0.5$

Board size	Memory Usage (kB)	
	CSP	Bonus CSP
10x10	2796908	2796908
11x11	2796908	2796908
12x12	2796908	2796908
13x13	2796908	2796908
14x14	2796908	2796908
15x15	2796908	2796908
16x16	2796908	2796908
17x17	2796908	2796908
18x18	2815004	2814996
19x19	2869284	2869388
20x20	2936508	2936632

Table 3: Memory usage of CSP and Bonus CSP when $minedensity = 0.5$