Comparing Algorithms in Solving Minesweeper

CS7IS2 Project (2020/2021)

Vaibhav Gusain, Sachin Arvind Gade, Jorge González Escribano, Vardan Kaushik

Trinity College Dublin
gusainv@tcd.ie, gades@tcd.ie, gonzaljo@tcd.ie, kaushikv@tcd.ie

Abstract. This project focused on solving Minesweeper games by using different artificial intelligence models. What makes this problem interesting to solve is that first, Minesweeper is NP-complete, making this problem especially difficult to solve, and second, in some instances, no move could win the game with sure certainty. For these reasons, we implement four different models and study their performance: logic inference, Q-learning, constrained solution problem, and DFS-BFS search. The research obtained interesting results by comparing all algorithms and showed that the constrained solution problem is the best algorithm to solve Minesweeper.

Keywords: Artificial Intelligence, Minesweeper, Logic Inference, Q-Learning, CSP, DFS, BFS

1 Introduction

Minesweeper is a single-player puzzle arcade game. It is one of the most popular games which almost every person has played during his/her early childhood. The rules of the game are pretty simple, we have to clear all of the boxes without exposing any of the mines using the clues provided by the neighbouring boxes. For each block user can perform 3 actions namely: click the box, flag it as a mine, flag it as "might be mine". Although the rules for the game seem pretty straightforward, yet some people still find it difficult to solve the minesweeper board, and as the game progresses the board gets complicated, and with one wrong move the game can end in a loss. The fact that despite having such simple rules the game is yet so complicated for many people attracted us to the problem, and we wanted to try different searching and learning techniques to the game and compare how they perform among each other. In this study, we will be comparing different search techniques and learning techniques for solving the minesweeper and will also compare them against a simple "logic and inference algorithm" which we will use as a baseline model. In the rest of the paper, we will go through the background research we did about different algorithms then we'll talk about the different experiments we conducted and their results followed by the conclusion. For the minesweeper game, we will be using [1] as our primary framework for simulating the game for our different agents. We chose

this repository because it provided out-of-the-box support for simulating the game, and also provided API endpoints that our agent can use to easily interact with the game.
The Group presentation is present as - https://drive.google.com/file/d/1pWh4tp4pbOw1GZ6P9S22C9sO9rPB6e4r/view?usp=sharing

2 Related Work

Minesweeper is a game with a predefined set of rules and logic hence it can be solved using a basic logic and inference method. This basic solver tries to solve the board just by looking at the cells. The first move is made randomly since it is not possible to find the exact non-mine cell. 4 basic rules are followed by the basic solver:

- If the cell contains 0, then click all the neighbouring cells
- If the cell contains 8, the flag all the neighbouring cells
- If the number in the cell is the same as the mines found in the neighbouring cells the click remaining cells
- If the number in the cell is the same as the undiscovered cells found in the neighbouring cells the flag remaining cells

While using the above basic solver, there will be a time when it will be impossible to find the safe cells and then the algorithm will have to make a random choice.

CSP based approaches are considered the state of the art when dealing with Minesweeper games. Much work has been done that compares this solution with multiple other algorithms but still concluding that CSP is the preferred approach

For another algorithm to solve minesweeper game we used DFS and BFS. DFS and BFS are searching algorithm which actually searches for the safe nodes in the minesweeper. The difference between DFS/BFS and Q learning is that in DFS/BFS, the algorithm knows the basic logic of the game; it's to check the number of a nodes in minesweeper and determine the location of the mine. There are eight possible moves available to the algorithm from a reveled or selected node of the minesweeper. DFS/BFS creates a tree like structure in minesweeper by going from the selected node to the neighboring nodes. From there the algorithm tries to deduce safe nodes or mines. If the algorithm finds safe nodes, it clicks on then and if the algorithm finds a mine, it flags them. DFS and BFS differs in how they move to their neighboring nodes; DFS moving to a neighboring node from a parent node tries to evaluate the neighboring node and then moves to the neighboring nodes of the child node whereas BFS moving to a neighboring node from a parent node tries to evaluate this node and then moves to the neighboring nodes of the parent node.

Up Till now, we have discussed the approaches in which we use the rules of the game to move our agent. But we also wanted to see whether it is possible to train such an agent, which could beat Minesweeper without learning any rules of the game explicitly. So we tried training a reinforcement learning agent, which

would learn to beat the game based on the reward structure provided to it rather than learning the explicit rules of the game. In such approaches, the problem is divided into states and possible actions. States can be thought of as a state of the agent in the game, for our example, the current state of the board will be equal to the state of the agent and action can be thought of as legal moves our agent can do in that state. The outcome of the state action pair is the next state. We wanted to try two different approaches namely: Q learning and deep Q learning. In Q learning, we calculate the Q value for every state and every possible action for that state using the formula:

Fig. 1. Here st is the current state, at is current action, rt is the reward for each state st, y is the discount factor.

Here Y and rt are the hyperparameters, as y determines the amount of contribution previous experiences will have on the current action for our agent and rt is the reward function, it should be high if the agent did a correct action and low if the agent did the wrong action. Q values for the starting state for all the actions are zero, and the goal of the agent is to reach the goal state while maximizing the reward. Such agents learn the Q Values for each possible state during the training time. Whereas in deep Q learning instead of using the formula provided by equation 1 above, we try to predict the Q value for each state using the deep neural network and we then choose an action that has the highest probability. Such networks are trained using an MSE loss. However, talking much about them would be out of the scope of this paper, but the reader can take a look at them in [2]. We decided to implement Q learning as we didn't have any compute power to run deep Q learning on our machines.

3 Problem Definition and Algorithm

Minesweeper is a well-known computer game. The game starts by showing a blank grid of squares. Some squares contain mines while the remaining squares are safe. The task is to find all the mines without detonating any of them. If the mine detonates then the game ends and the player losses the game. If there is no mine then the square will contain a number telling the number of mines present in the eight immediately adjacent squares.

3.1 Logic Inference - Basic Solver

For the minesweeper game, our agent can have 13 possible states for each of the cells on the board namely:

- 0-8 is the number of mines in the surrounding.
- 9 is a flagged field.
- 10 is the questioned field.
- 11 is the undiscovered field
- 12 is a mine field.

Logic inference uses basic game rules to solve the problem. As explained previously, the first click is random since there is nothing to check. Once the cells are opened, it follows the rules and tries to solve the problem. The pseudo code can be written as:

Algorithm 1 Logic inference - Basic Solver

```
Open random starting cell
while game not finished do
    for each cell in the grid do
        if cell = 0 then
            Click all the neighbouring cells
        end if
        if cell = 8 then
            Flag all the neighbouring cells
        else
            if cell = number of mines in the surrounding then
                Click all the remaining neighbouring cells
            end if
            if cell = number of unexplored cells in the surrounding then
                Click all the remaining neighbouring cells
            else
                Randomly click leaving mines cells
            end if
        end if
    end for
end while
```

3.2 Q Learning

As defined previously Q learning agent learns to play the game with a reward structure defined in eqn[1] which is based on the current state of the game, rather than learning the rules of the game.

By using the information above, along with the information about the dimension of the board, we can define a linear state of the system. This can be defined as an nD array of size (nRows,nColumns,13), where nRows and nColumns are the number of rows and columns of the game board and then the corresponding nth dimension can tell what is the state of each of the cells. For simplicity, we can convert this nd array into a 1d array. Now we need to first train our agent for it to learn what actions it should take when it is in a particular state i.e we need

to train it to learn the Q values of the agent. An epsilon greedy approach was followed to train an agent with the epsilon parameter set to 0.8, this approach was chosen to make sure the agent explores all of the action for a particular state before finalising the q values. The Q values for any state were calculated using the linear equation W*X + B where X is the current state, W and B were the ridge regression coefficient and intercept parameters and were initialised with the value of 1e-5. While the agent explores the environment we keep track of the states along with the max possible reward for that state. Then we train a ridge regression model with input as possible states and output as the reward value. Once the model is trained W is set to the coeff value of the model and b is set to the intercept value of the model. This model was trained recursively after an "n" number of events in the environment and it was also another hyper-parameter for our agent. Once the model is trained then the model parameters W and b were stored in a .pkl file. During the inference phase, we can calculate the Q values using the pre-trained W and b values and choose the action with maximum Qvalue.

3.3 Constraint Satisfaction Problem

A Minesweeper game can be defined as a Constraint Satisfaction Problem in which the variables are the undiscovered cells, the domain is whether there is a mine in such cell (1) or not (0) and the constraints are that the sum of the variables around a discovered cell must be equal to the number inside that cell.

$$
\begin{aligned}
X &= \{\text{undiscovered cells}\} \\
D &= \{1, 0\} \\
\sum x &= n_a \ \forall \ x \text{ with constraint c}
\end{aligned}
$$

This CSP also has the peculiarity that each time a decision is made (that is, each time a cell is opened), the set of constraints in the problem change as new cells of the board are discovered. Algorithm 2 shows the pseudocode of the algorithm used to solve this problem.

Building the graph In this problem, undiscovered cell (variable) surrounding a discovered cell (constraint value) is connected with every other variable surrounding the same constraint value.

Solving trivial constraints When the number of variables having the same constraint is equal to the value of the constraint, all those variables are ensured to be 1 (contains a mine). This can be easily inferred from Equation 1. Following is an example where this case can be seen(Fig 2):

Algorithm 2 CSP Solve

```
1: Open random starting cell
2: while game not finished do
3:     Generate constraint graph
4:     Find trivial constraints
5:     if Trivial constraints found then
6:         Perform actions
7:     else
8:         Simplify constraints
9:         Find trivial constraints
10:        if Trivial constraints found then
11:            Perform actions
12:        else
13:            Search all possible legal configurations
14:            if A variable is the same across all solutions then
15:                Actuate on cell
16:            else
17:                Discover cell with least chance of being bomb
18:            end if
19:        end if
20:    end if
21: end while
```

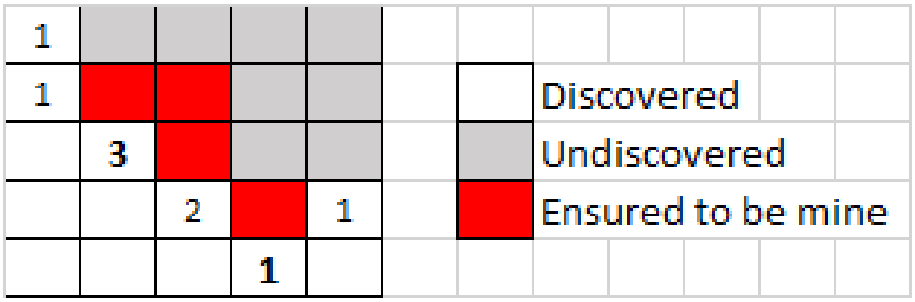

Fig. 2. Trivial constraints ingame

Simplifying constraints Given two constraints A, B with values n1, n2 and sets of dependent variables S1 and S2 respectively, if A is a strict superset of B, A can be simplified as follows:

$$S_1 = S_1 - S_1$$
$$n_1 = n_1 - n_2$$

In the following in-game picture, this simplification can be seen by presenting the variables as part of an equation:

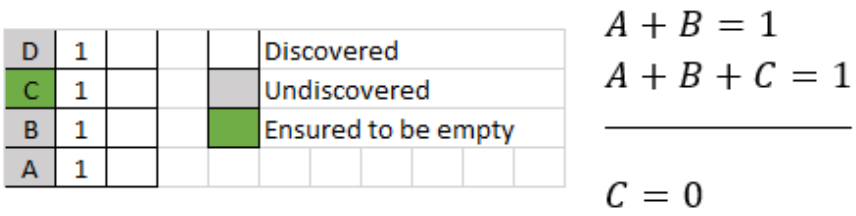

Fig. 3. Ingame example of constraint simplification

Search all possible legal configuration If the two previous methods fail, a recursive search is performed for all the possible variable configurations that satisfy the constraints. Filtering is employed to improve the search time. When the search finishes, from all the possible configurations, the variables are checked to see if there is any of them that gets the same value assigned across all of them, which means that such value is the real one for sure.

Random choice In some games, it may be the case that there is no move that will be successful with 100% certainty. In these cases, based on the possible solutions obtained from the previous step, the cell that had a value of 1 assigned in the least of them is discovered, as it is the cell that has the least chance of losing the game.

3.4 DFS and BFS

DFS/BFS algorithm first clicks on a node randomly. This could result in the game being lost in the first move as the selected node can be safe as well as mine. In case the node is safe, then all the nodes which are not revealed and have adjacent mines greater than zero are added in a queue. For DFS, the addition of the nodes are done at the starting of the queue while for BFS, the addition of the nodes are done at the end of queue. All nodes are popped from the beginning of the queue.

The first element of the queue is evaluated first. By evaluation, I mean we check all the surrounding nodes and if the number of adjacent nodes is equal to the number of adjacent mines then all these nodes are marked as mine, otherwise we check for safe nodes. If the number of unrevealed adjacent nodes are greater than the number of adjacent mines and if all of the mines are marked, then the rest of the nodes are marked are safe and then clicked on. From here new neighboring nodes are added to the queue based on the type of the algorithm. Sometimes, the algorithm cannot find any neighboring nodes to evaluate as safe node or mine node, this is when the queue becomes empty and there are no more nodes to evaluate and in this case another unrevealed node is clicked on randomly. This could either make a new DFS/BFS tree or could click on a mine and lose the game. This goes on till either the game is won or lost.

4 Experimental Results

To evaluate the algorithms, we ran every algorithm on a game with 10X10 grid and 12 mines, 200 times. Every time we run the algorithm, the game creates new grid of 10X10 with 12 mines placed randomly.

4.1 Logic Inference - Basic Solver

The Logic inference algorithm was able to win 55 games out of 200. The win rate is 27.5%. Since, first choice is made random, there are chances of getting mine in first click. Also, sometimes the game reaches at the point where is is difficult for the algorithm to make safe click, at that time the algorithm uses random click which increases the chances of getting mine.

4.2 Q Learning

The Q learning algorithm was able to only win 5 games out of 200. Which gives us the win rate of our reinforcement learning agent just 2.5%. However we should also consider the fact that this agent was only trained for 1e7 iteration, and no prior information of rules were provided to this agent as compared to the other approaches in the study. We believe if we would have trained the agent for some more iterations it would have outperformed other algorithms in the study.

4.3 CSP

In the results it can be seen that this model achieves a 73% of win rate and that, for those games in which a random choice has to be made, it can win 65% of them.

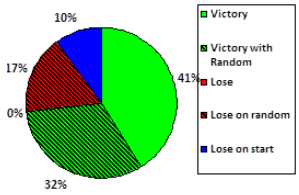

Fig. 4. CSP Result

4.4 DFS and BFS

BFS algorithm wins 148 times out of 200 whereas DFS algorithm wins 133 times out of 200. This makes the win rate of BFS as 79% and the win rate of DFS 78%. Since the location of the mines are random and so is the clicking of nodes at start so the results are not consistent every time the code is run. Moreover, for all 200 games, BFS does 26200 number of evaluation over 1.8 seconds and DFS does 27400 number of evaluations over 1.9 seconds. DFS is not only slow and does more evaluations than BFS but it also gives lower win rate, which is consistent for most of the game simulations. This makes BFS a better algorithm than DFS for minesweeper game.

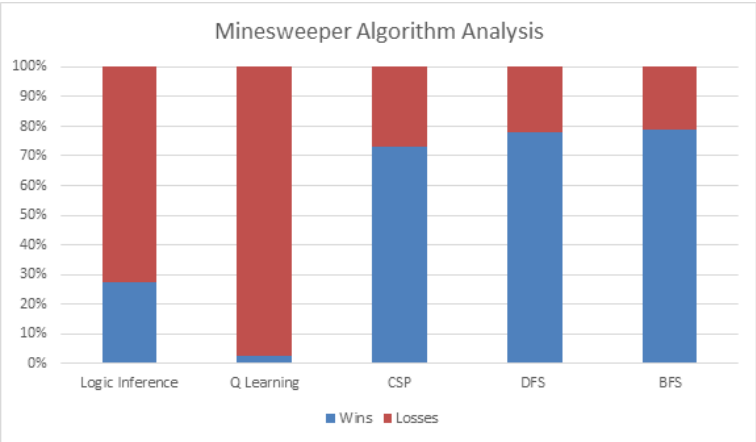

Fig. 5. Minesweeper Algorithm Analysis

5 Conclusions

From our research we can safely see that uninformed search were able to outperform CSP's, Logic inference and the Q learning method. However, one dis-

advantage of uninformed search is that the algorithm often runs into a deadlock situation in which there are no more evaluable nodes. To get out of the deadlock, we again have to randomly select an unrevealed node which often times select a mine and we loses the game. We believe if one needs a solution for minesweeper in a very short frame of time then they should opt for CSP's as they will provide the optimal solution 80% of time. However, if one has time they should try and run more training iteration on the Q learning agent as that agent has the potential to beat win rate of all the other algorithms mentioned in the research.

References

1. Kaye, R. Minesweeper is NP-complete. The Mathematical Intelligencer 22, 9–15 (2000). https://doi.org/10.1007/BF03025367
2. Yimin Tang, Tian Jiang, & Yanpeng Hu. (2018). A Minesweeper Solver Using Logic Inference, CSP and Sampling.
3. Minesweeper framework https://github.com/duguyue100/minesweeper
4. DeepQ Learning https://arxiv.org/abs/1704.03732
5. Q Learning https://web.stanford.edu/class/cs234/slides/lecture6.pdf
6. Minesweeper Q learning, Reference code for minecraft with Q learning https://github.com/HaniAlmousli/minesweeperRLDemo
7. Minesweeper game using BFS and DFS https://programmersought.com/article/41763978569/
8. Minesweeper Basic Solver https://github.com/aditya1702/Machine-Learning-and-Data-Science/tree/master/Minesweeper%20AI%20Bot