AI approaches to solve Minesweeper problem

CS7IS2 Project (2020/2021)

Aishwarya Agarwal, Boris Flesch, Rui Xu, Niejun Yin

aiagarwa@tcd.ie, fleschb@tcd.ie, ruxu@tcd.ie, yinn@tcd.ie

Abstract. This article explores different AI approaches to solve the Minesweeper game. After a review of related work, we propose the following approaches: DFS, BFS (agent-enhanced), CSP and RL (Q-learning). We then evaluate the performance of these approaches using criteria such as win rate and number of steps. While DFS/BFS seems to provide the lowest performances, CSP clearly appears as a great solver for the Minesweeper game regarding our evaluation criteria.

Keywords: minesweeper, DFS, BFS, CSP, RL

1 Introduction

Many AI approaches could be considered as Minesweeper solvers. In this article, we propose a review of related work to evaluate and propose relevant approaches. Then, problem and algorithm section defines the environment and goal of our AI, while describing all approaches tested (i.e. DFS/BFS, CSP, RL). Finally, we provide results, evaluation of the results and conclusions drawn from our observations.

2 Related work

There are a lot of approaches that have been used to build an AI agent to solve minesweeper game puzzles. BFS/DFS, Reinforcement Learning, Constraint Satisfaction Problem are the few which we will be exploring in this project.

BFS is an uninformed search algorithm that expands all nodes at a given depth before expanding any node at a greater depth. While DFS works by always generating successor nodes of the newly expanded node until a certain depth is reached, and then backtracking to the next newly expanded node and generating one of its successor nodes to find the solution [1]. For instance, in the minesweeper problem, a 2-dimensional grid of nodes is initialized and among those m nodes are randomly chosen and set to be mines. The number of adjacent mines of the remaining nodes will also be calculated out. Then the initial state in the minesweeper problem is considered as the root node which will be given once the game starts, the successor nodes will be generated in a way that finds their adjacent cells. This traversal strategy leads to a BFS or DFS approach. Since BFS

and DFS both need to traverse all nodes in extreme cases, the time complexity is the same. But BFS will take a higher space complexity since it needs to store all nodes at a given depth before expanding any node at a greater depth while DFS only saves the path of nodes from the initial node to the current node.

Reinforcement learning is learning what to do — how to map situations to actions — so as to maximize a numerical reward signal[2]. The reinforcement learning uses rewards and punishments as signals for positive and negative behavior. The goal of reinforcement learning is to find the action/state which would lead the agent to get the maximum reward. MDPS are mathematical frameworks which frame the problem of learning from interaction to achieve a goal. The learner and decision maker is called the agent. The thing it interacts with, comprising everything outside the agent, is called the environment. The environment also gives rise to rewards, special numerical values that the agent seeks to maximize over time through its choice of actions[2].

A constraint satisfaction problem (CSP) can be stated as a problem that consists of a set of variables, a value need to be selected from a given finite domain, and to be assigned to each variable in the problem so that all constraints relating to the variables are satisfied [5]. By considering minesweeper as a CSP, Chris Studholme [6] developed an efficient algorithm that makes use of CSP techniques for both finding an appropriate next action and for calculating the probability of success when a guess has to be made. The results showed that the algorithm can have a win rate above 60% for a beginner lever minesweeper and a win rate that above 30% for an intermediate lever minesweeper. One of the latest approaches implemented CSP and compared with few other algorithms like Reinforcement Learning, regression etc. The work showed that CSP and reinforcement learning gives the best results in solving the puzzle[3].

From the previous work [7][8], the standard measurement of minesweeper solver is the win rate and steps when solving problems of different difficulty. And the biggest factor deciding the win rate is the mines density. In our project, we will also test the solvers' win rate and steps with different mines number. In order to make the result more comprehensive, we will also introduce one more indicator to show the performances of each solver.Also, according to Studholme [6], the probability of a mine in the middle of the board is higher than in a corner, thus, in the following experiment, we will always start at the left-up corner.

3 Problem and algorithm

3.1 Minesweeper rules, goal of the AI and environment

Minesweeper is a puzzle game that consists of a rectangular grid of tiles which have randomly distributed hidden mines. Usually the standard board size is

16x16 with 40 mines, but for the evaluation and analysis of AI algorithms we will consider different board sizes and number of mines starting with board size 8x8 with 1 mine. It is a single player game where the player clicks on the tile to uncover it which can either be a mine or blank or an integer which indicates the number of adjacent tiles that have a hidden mine. If the tile uncovered is blank then it indicates there is no mine adjacent to that uncovered tile. If the uncovered tile contains a mine then the player loses and the game ends. Using this limited information, the goal of the AI agent is to uncover all the tiles that do not contain mines or flag all the tiles that contain mines.

For this project, our goal has been to focus on AI approaches. Therefore we chose to fork an existing Minesweeper game written in Python by "*cash*", which is available at the following link: https://github.com/cash/minesweeper. This repository provides a few features meant to be used for AI applications over Minesweeper and offers a simple UI to visualise the evolution of the game when the AI is playing.

However, the behaviour of the game did not totally match the one expected for our project. We therefore had to modify the behaviour of the game/rules and add a set of features including mines flagging, automatic discovery of adjacent nodes, reveal of cells one by one instead of reveal by groups (i.e. if neighbour cells are blank), methods for evaluation purpose (e.g. win rate and performance of the AI) and a few other adjustments. For that purpose, we used a first basic approach with DFS described below.

3.2 Depth First Search(DFS)/Breadth First Search(BFS)

The DFS approach has been chosen essentially testing our Minesweeper game code base and adding the aforementioned set of essential features. Indeed, the goal was to develop a set of common features usable with all AI approaches described below. As we chose DFS as a first basic approach to observe and modify that Python code appropriately, this initial approach is not meant to provide particularly relevant results.

Basically, BFS and DFS have similar ideas in search. They will query adjacent nodes. The difference is that BFS will expand all nodes at a given depth and then expand any deeper nodes. In our project, we added an AI action decision-making process for BFS, implemented a decision-making algorithm based on BFS to solve the minesweeper problem.

The main idea of decision-making is to evaluate whether the node we traversed is safe. We will use a quite straightforward method that compares the number of adjacent mines which is given by the game and the unrevealed adjacent nodes number and revealed mines number. If the number of unrevealed mines is equal to the unrevealed adjacent nodes, then all the unrevealed adjacent nodes are mines. If all the adjacent mines are revealed, then the left unrevealed adjacent nodes are safe and will all be added to safe_nodes queue. Otherwise, the unrevealed adjacent nodes are uncertain.

Algorithm 1: BFS Algorithm for Minesweeper

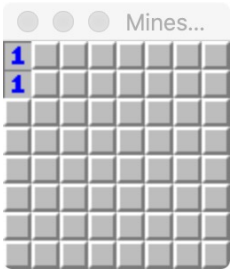
```
Function BFS(current_location):
    for ( node in current_location.adjacents() ) {
        if node is safe then
            safe_queue.add(nodes) ;
        else if node is uncertain then
            uncertain_queue.add(nodes) ;
        else
            label the node as mine;
        end
    }
    if safe_queue not empty then
        BFS(safe_queue.pop()) ;
    else
        BFS(uncertain_queue.pop());
    end
End Function
```

3.3 Constraint Satisfaction Problem(CSP)

Minesweeper can be formulated as a CSP, and according to the definition of CSP, the CSP approach is designed as:

- Variables: all the cells
- Domains: mine or non-mine
- Constraints: The number of mines in a variable's neighbors equals the value of the cell

For example, if we have the situation of Figure 1, We can build the constraints by the Table 1:



Variables	Domain	Constraints
(0,0)	non-mine	[(1,0),(1,1)]=1
(0,1)	non-mine	[(1,0),(1,1),(0,2),(1,2)]=1

Fig. 1: Example for revealed cells **Table 1:** Constraints equations

The typical method for solving CSPs is backtracking search, but here in our agent, we adopted an inference method for the agent to maintain and update the constraint equations, then make an action decision based on the internal constraints. For solving the CSP, the agent keeps 2 lists:

- non_mine_variables: a list of cells which be considered as non-mines
- mine_variables: a list of cells which be considered as mines

The agent plays the minesweeper in two processes, the action decision process is responsible for finding a cell for the next click, the constraints creation process is responsible for creating new constraints after one cell is revealed.

Action decision process

- **Click non-mine variables**: By default, the cell (0, 0) will be clicked as the agent doesn't have any constraints to infer non-mines and mines initially. For other situations, the agent will click cell from the non_mine_variables list which can guarantee the action will not reveal a mine.
- **Constraints reduction**: There will be a time when the non_mine_variables list is empty, then we need to infer more information from the constraints and the information may not be that direct, a reduction method can help make the constraints simpler and clearer. For example, if we have 2 constraints equations like Table 3.3 where one constraints elements is a subset of the other one, we can simplify the constraints by the following formula:

$$\begin{cases} A + B = a \\ A + B + C = b \end{cases} \implies \begin{cases} A + B = a \\ C = b - a \end{cases} \tag{1}$$

  So, here we can simplify the second constraints equation to $[(0, 2), (1, 2)] = 0$, and it is easily to find both (0, 2) and (1,2) are non-mine cells.
- **Flag new mine-variables and find non-mine variables**: After reducing the constraints, The agent search for new non-mines and mines based on the following two types of equations:
  - A + B + C = 0, when the constraint value is zero, which means all the variables in the equation are non-mines, we can add these variables into non_mine_variables list.
  - A + B + C = 3, when the constraint value equals the number of variables in the equation, it means all the variables are mines, we need to add them to the mine_variables list and flag them.
- **Random choice**: There could be a scenario where no constraints equation can be reduced and the non-mine variables list is empty, which means the current information is not sufficient enough to make further inferences. The agent has to click a cell randomly, for making this random choice less risky, we roughly calculate the risk for unrevealed neighbors for all revealed cells based on an accumulation of the number of a revealed cell minus the number of flagged mines in its neighbors then divided by the unrevealed cells of its neighbors, and return the less risky cell as next action. Due to the inaccuracy of this heuristic method, this step still is considered a random choice.

Constraints creation process

- **Form the constraints equation**: When a cell is opened, a constraints equation is formed by the cell's neighbors and number. For example, the

number of opened cell is a, that means we have a number of mines in its neighbors, then the equation is calculated by the following way: the sum of unrevealed and unflagged variables equals a minus number of flagged mines in the neighbors

- **Remove variable from constraints equations**: Once we open a cell safely, which means the cell must be a non-mine variable, we can remove this variable from the constraints equations list.

The pseudocodes of two processes

Algorithm 2: CSP Agent Action Decision Process

```
Reveal the first cell according to the config;
if There is any non-mine variable remain then
    Pop a variable from the mine_variables list and click the coordinate
     of that variable;
else
    Reduce constraints and find new non-mine cells and mine cells;
    Flag new mines that we found;
    if Any non-mine variable remain after reducing constraints: then
        Pop a variable from the mine_variables list and click the
         coordinate of that variable;
    else
        Click a random cell with heuristic;
    end
end
```

Algorithm 3: CSP Constraints Creation Process

```
if All safe cells are revealed or all mines are flagged: then
    AI wins the game. ;
else if A mine was clicked: then
    AI lost the game ;
else
    Get the value of last variable we clicked;
    Build constraints by last variable;
    Remove the last variable for other constraints;
    Try to find new non_mines and mines;
end
```

3.4 Reinforcement Learning (Q-learning)

In the previous works, RL was implemented by randomly uncovering the cell and if the cell has mine then reward as -1 and if cell is not mine then reward as +1. Using this algorithm the model was trained over approximately 100000 iterations and based on that the agent played the game. But this model suffers from a hugely large state space. The algorithm must visit each board state under

a random sampling of mine configurations in order to accurately approximate the probability of whether each cell is safe [4]. Hence, we tried evaluating a different approach using Reinforcement Learning to check whether we could get better results. We formulated the minesweeper game as MDP and using q-learning we tried to get the best possible state which an agent can move into to win the game. We calculate the reward for each cell based on the revealed/non-revealed cells in its neighborhood. The main elements of the game are as given below:
Environment : Minesweeper Game Grid
Action: Reveal the cell, Flag the cell
States: 1-9, blank, Unrevealed
Q-value iteration takes the following form:

$$Q(s,a) \leftarrow Q(s,a) + (R(s,a,s') + \gamma max(Q(s',a'))) \tag{2}$$

But in the minesweeper game, we are just concerned with the immediate reward whether a particular move will uncover a mine on a specific board configuration. We do not need the long term reward [4]. In order to approximate the immediate reward, we remove the γmax(Q(s',a')) term, leaving the equation as:

$$Q(s,a) \leftarrow Q(s,a) + (R(s,a,s')) \tag{3}$$

where s current state; a action; Q(s,a)q value of a particular state and action; R(s,a,s')Reward; learning rate.
To start the game agent will click the cell as passed in game configuration. Then the q-value for each of the neighbors of the revealed cell will be calculated. If the q-value of at least one of the neighbors is greater than or equal to 0, then the agent will reveal the cell with maximum q-value otherwise q-values of all the unrevealed cells will be calculated and agent will reveal the cell with maximum q-value. The reason for calculating the q-values for only neighbors first is for the better and faster performance but if all the neighbors have q-values as negative i.e. no cell without risk then all the unrevealed cells are checked if there exist any cell with better q-value. Also, based on minimum q-value, it is checked that if the number of exposed cells adjacent to the cell with minimum q-value are equal to total number of neighbors-1, then the cell is flagged. The game will continue till the agent reveals all the non-mines cells.

Algorithm 4: Calculate Reward

```
initializing reward as 0 for all adjacent cells to the cell: do
    if adjacent cell is exposed: then
        if adjacent cell value > 0 then
            Reward += adjacent cell value ⋆(-10);
        else
            Reward += 10;
        end
    else
        Reward +=0;
    end
end
```

Algorithm 5: Update q value

```
Reveal the first cell according to the config;
for all adjacent cells to the revealed cell: do
    if adjacent cell is not revealed and flagged: then
        value(adjCell)=(1-α)⋆value(adjCell) + α ⋆reward;
end
if value of at least one adjacent cell is greater than or equal to 0: then
    Reveal an adjacent cell with maximum value;
else
    if revealed adjacent cells count is 1 less than the total adjacent cells:
     then
        Flag the cell with minimum value;
    for all the unrevealed cells in the grid: do
        if cell is not revealed and flagged: then
            value(adjCell)=(1-α) ⋆ value(adjCell) + α ⋆reward;
    end
    Reveal the cell with maximum value;
end
```

4 Results

In the experimental process of performance testing, our strategy is to define game configurations with 9 different difficulty levels, and use different algorithms to conduct 100 confrontation tests with each configuration. In each of the 100 tests, we considered the following indicator:

- Win Rate: Percentage of wins in 100 games
- Degree of Completion: Average degree of completion in 100 games

$$completion = \frac{\sum(revealed\_num/cell\_num)}{round\_time} \tag{4}$$

- Average Steps: Average number of steps in the winning round

4.1 Win Rate

Figure 2 shows us the win rates of different solvers in three games with different difficulty levels. We can clearly see that as the density of landmines increases, the success rate of these three algorithms will decrease to varying degrees, which is in line with expectations. Because the difficulty of the game is positively correlated with the density of mines.

Among them, we can see that when the difficulty of the game is very easy, the success rate of CSP and RL is very high, almost reaching 100%, while BFS is also quite high, which is about 60%. But as the difficulty of the game increases, the success rate of BFS has dropped the most. While CSP can still maintain a 20% win rate when the mine density reaches 10%.

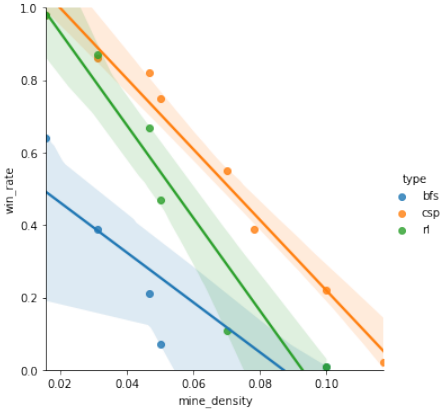


Fig. 2: Win Rate Comparison

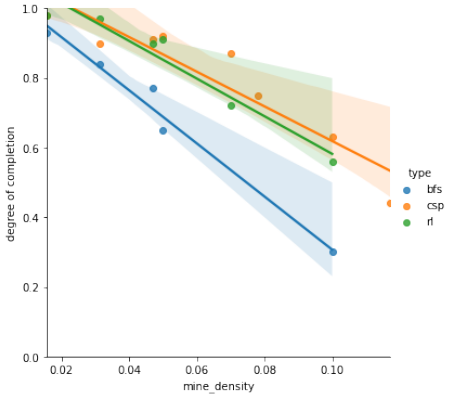


Fig. 3: Degree of Completion

4.2 Degree of Completion

We also calculated the degree of completion of each game for comparison. We define the degree of completion as the ratio of the number of grids explored by the algorithm (i.e. marked as safe) to the total number of grids. The more grids the algorithm explores safely, the closer it is to the victory of the game.
We can see from 3 that although the completion rate of the three algorithms is decreasing as the game difficulty increases, CSP always has the highest completion rate, RL has the second-highest completion rate, and BFS is always the lowest.

4.3 Average Steps

Game Configuration		**BFS**	**CSP**	**RL**
size	mines			
8 * 8	1	60	36	62
8 * 8	2	61	45	61
8 * 8	3	60	49	60
10 * 10	5	94	83	95
10 * 10	7	-	85	93
10 * 10	10	-	85	-
16 * 16	20	-	224	-
16 * 16	30	-	224	-
16 * 16	40	-	-	-

Table 2: Average Steps in the winning round

Table 2 shows the average number of steps of three algorithms under the condition that the win rate exceeds 20%. When the winning rate is less than 20%, we suppose that the algorithm has failed in the game and no longer calculates the average number of steps. The average steps can reflect the speed of winning the game to a certain extent.
We can see that under any game difficulty, CSP maintains a relatively low number of steps, that is to say, in the three algorithms, CSP can complete the game faster.

5 Conclusion

From the results we can conclude that CSP approach is most successful when compared to Reinforcement Learning and BFS/DFS approaches for the minesweeper game. This could be because in CSP approach the agent makes as minimum guesses as possible. The agent aims to reveal or flag cells are based on the certainty. The agent makes a random choice when the current information is not sufficient enough to make inferences. But like Reinforcement Learning and BFS/DFS algorithms, we can observe that as the number of mines increases the performance of the CSP algorithm decreases. This could be because as there are more mines in the grid, whenever a random choice is made the probability of clicking a cell that has mine increases.
In future we can explore backtracking approach for CSP and anlayze if the results are improved. Another area we can look into is integrating the q-learning algorithm with CSP to make a little informed guess and reduce the risk of clicking a mine when there are large number of mines.

References

1. Korf, Richard E. "Depth-first iterative-deepening: An optimal admissible tree search." Artificial intelligence 27, no. 1 (1985): 97-109.
2. Sutton, Richard S., and Andrew G. Barto. Reinforcement learning: An introduction. MIT press, 2018.
3. Tang, Yimin, Tian Jiang, and Yanpeng Hu. "A Minesweeper Solver Using Logic Inference, CSP and Sampling." arXiv preprint arXiv:1810.03151 (2018).
4. Gardea, L., Koontz, G. and Silva, R., Training a Minesweeper Solver.
5. Brailsford, Sally C., Chris N. Potts, and Barbara M. Smith. "Constraint satisfaction problems: Algorithms and applications." European journal of operational research 119, no. 3 (1999): 557-581.
6. Studholme, Chris. "Minesweeper as a constraint satisfaction problem." Unpublished project report (2000).
7. Nakov, Preslav, and Zile Wei. "Minesweeper, Minesweeper." Unpublished Manuscript, Available at: http://www. minesweeper. info/articles/Minesweeper (Nakov, Wei). pdf (2003).
8. Castillo, Lourdes Pena, and Stefan Wrobel. "Learning minesweeper with multirelational learning." In INTERNATIONAL JOINT CONFERENCE ON ARTIFICIAL INTELLIGENCE, vol. 18, pp. 533-540. LAWRENCE ERLBAUM ASSOCIATES LTD, 2003.