Fast constraint satisfaction problem and learning based algorithm for solving minesweeper

Yash Pratyush Sinha[a], Pranshu Malviya[a], Rupaj Kumar Nayak[a,*]

[a]International Institute of Information Technology, Bhubaneswar, India 751003

Abstract

Minesweeper is a popular spatial-based decision making game that works with incomplete information. As an exemplary NP-complete problem, it is a major area of research employing various artificial intelligence paradigms. The present work models this game as Constraint Satisfaction Problem (CSP) and Markov Decision Process (MDP). We propose a new method named as dependents from independent set using deterministic solution search (DSScsp) for the faster enumeration of all solutions of a CSP based minesweeper game and improve the results by introducing heuristics. Using MDP, we implement machine learning methods on these heuristics. We train classification model on sparse data with results from CSP formulation. We also propose a new rewarding method for applying a modified deep Q-learning for better accuracy and versatile learning in minesweeper game. The overall results have been analyzed for different kinds of minesweeper games and their accuracies have been recorded. Results from these experiments show that the proposed MDP based classification model and deep Q-learning method are the best methods in terms accuracy for games with given mine densities.

Keywords: Constraint satisfaction problem, Markov decision process, minesweeper game, machine learning, Q-learning

1. Introduction

Minesweeper is a single-player game where the player has been given a grid minefield of size $p \times q$ containing n mines where each block of the grid contains at most one mine. These mines are distributed randomly across the entire minefield and their locations are not known to the player. The goal of the game is to uncover all the blocks which do not contain a mine (i.e., all safe blocks). If a block containing a mine is uncovered, the player loses the game. Whenever a safe block is uncovered, it represents the number of mines in its eight-neighbors containing mines. In the human version of

*Corresponding author

Email addresses: yashpratyushsinha@gmail.com (Yash Pratyush Sinha), pranshumalviya2@gmail.com (Pranshu Malviya), rupaj@iiit-bh.ac.in (Rupaj Kumar Nayak)

the game, when this number is zero the game automatically uncovers all of the eight-neighbors of such a block. Generally, first block uncovered by the player does not contain a mine. Analyzing the given pattern and arrangement of these boundary digits, the player has to figure out the position of the safest block that can be uncovered. The player can also mark a block with a flag if he/she decides that the block is mined. In this way the number of mines left to be discovered is updated and player can continue to play until all the free blocks are revealed. In some situation, if there exists more than one block with similar chances of containing a mine, but not with the surety of absence of mine, the player has to randomly choose the next block to uncover. As a result of this ambiguity, the player can lose the game even if he applies the most optimal strategy. As it is computationally difficult to make an optimal decision without considering all options, minesweeper is an NP-complete problem [1] which is an intriguing area of research. Many researchers have already described the formulation techniques that can be employed for this game and improved upon the previous ones as discussed in the next section.

2. Related Work

A strategy called single point strategy considers only one instance of the uncovered block and finds the safest move from its immediate neighbors. Its implementation model may vary but the single point computations are common practice in them. As a result, this strategy struggle with larger dimensional games or games with higher mine densities. Therefore, other strategies can be used to reconcile the limitations of single point techniques. CSP is another prominent way of formulating the minesweeper game in a mathematical form and several algorithms [2] exist to solve these CSPs. CSPs are the subject of intense research in both AI and operations research since not only it overcomes the problems in single point strategy, but the regularity in their formulation provides a common basis to analyze and solve problems of many unrelated families [3, 4]. MDP is another way of formulating games into state-action-reward form. This way has been extensively practiced nowadays to formulate games [5]. We also use such formulation and combines it with the CSP formulation to achieve even better results.

The minesweeper game is one of the basic ways of realization as a CSP [6]. The thesis by David Becerra [7] tests and talks about several different approaches taken to solve the minesweeper game and comes to the conclusion that CSP methods work as efficiently as any other formulations of the minesweeper game.

Our approach for solving CSPs is in line with the look-ahead method proposed by Dechter and Frost [8] that avoids unnecessary traversal of the search space to result in a more optimized method for dealing with CSPs.

The work of Bayer et al. [9] describes another approach that uses generalized arc consistency and higher consistencies (1-RC & 2-RC) to solve the CSP generated by the minesweeper to assist a person playing the game. Both of these approaches are inefficient when constraints are not very scalable for larger boards.

Nakov and Wei [10] describes the minesweeper game as a sequential decision-making problem called Partially Observable Markov Decision Problem (POMDP) and convert the game to an MDP, while also reducing the state space.

Approaches have been carried out like using belief networks by Bonet and Geffner [11] that consider a set of partially observable variables and considers the possible states as beliefs.

In the research by Castillo and Wrobel [12] authors described methods to learn the playing strategy for minesweeper not just by inferring from given state but also by making an informed guess that minimizes the risk of losing. They used Induction of Logic Program (ILP) techniques such as macros, greedy search with macros, and active inductive learning for the same. This is a very different and promising machine learning approach that is used in this research.

Sebag and Teytaud [13] have attempted to estimate the belief states in the minesweeper POMDP by simplifying the problem to a myopic optimization problem using Upper Confidence Bounds for Monte-Carlo Trees (UCB for Trees). While this method is reasonably accurate for small boards, it is too slow for larger board sizes making it inadequate. As described in research of Legendre et al. [14], authors have used heuristics with computing probabilities from CSP, called HCSP to study the impact of the first move and to solve the problem of selecting the next block to uncover for complex minesweeper grids on the basis of various strategies. By preferring the blocks which are closest to the frontier in case of a tie on the probability of mines, they got the better results for various sizes of minesweeper matrices. We have also used a variant of HCSP by using Manhattan distance for measuring closeness to the boundary.

In the research given by Buffet et al. [15], authors combined the two methods i.e., UCB for trees and HCSP as described in [13] and [14] respectively to improve the performance of UCB to be used not only for small boards but also for big minesweeper boards. This improves performance by a lot.

Recent works by Couetoux et al. [16] have tried to solve minesweeper by directly treating it as a Partially Observable MDP (POMDP). However, as pointed out by Legendre et al., minesweeper is actually a Mixed Observability MDP (MOMDP) and several improvements can be made to the current solvers to increase performance.

In the research of Gardea et al. [17] the authors have taken an approach to utilize different machine learning and artificial intelligence techniques like linear and logistic regression and reinforcement learning. Q-learning [18] is a reinforcement-learning algorithm which has been used extensively in games [5, 19] to solve MDPs. We have also used a modified version of Q-learning for solving above MOMDP efficiently.

In this paper, we have described our methods of creating an automatic solver for minesweeper using several techniques by gradually improving upon its previous ones. We have also used a modified version of Q-learning for solving above MOMDP efficiently. In Section 2, we describe how the game is formulated into CSP and then in an MDP. We describe techniques to play the game using both the CSP & MDP formulation. Also, we have described three ways to obtain the probabilities of each block containing a mine as well as a method to find blocks about which we can deterministically say if they contain a mine. After that, we give several methods, both hand-crafted and machine learning based to choose blocks in case that there is no deterministic solution. In Section 4, we described the implementation of these methods and present the results graphically and comparison table with existing popular methods. We conclude the paper in Section 5.

3. Formulations

Prior to the detailed explanation of algorithms and machine learning techniques that we used, in this section, we are going to describe how the minesweeper game is first realized as a playable mathematical form. These forms and notations then allowed us to design and improve upon the existing algorithms, particularly for this game for better results. This paper particularly uses two different formats, as expressing the game as a CSP or as an MDP.

3.1. CSP formulation

CSP [2] is a natural and easy way for the formulation of minesweeper, as it accurately captures the intuition of finding which blocks may or may not contain a mine. We use the fact that each uncovered block shows the number in its 8-neighbor region in order to build a CSP from the minesweeper game. These numbers put a constraint on the covered blocks in the neighbor that how many of them should have a mine. (We also know the total number of mine present in the game. This information will be used as heuristics discussed later in the paper.) If we treat the covered blocks as boolean variables (with 0 representing a safe block and a 1 representing a mine) we can use the uncovered numbers as constraints on such variables. For example, a two uncovered block will constrain the sum of all of the variables in it's eight neighbors to two. Assuming the uncovered number of every block at (i, j) represented by $\text{num}_{i,j}$ and the boolean variables represented by $\text{mine}_{i,j}$, the constraints can be written as (adaptively for blocks on the border):

$$num_{i,j} = \sum_{k=i-1}^{i+1} \sum_{l=j-1}^{j+1} mine_{k,l}, \;\; if(k \neq i \;\&\; l \neq j). \tag{3.1}$$

For several (i, j) we will obtain a set of the equation in the form of equation (3.1) that can rewritten in a matrix format as follows:

$$AM = N, \tag{3.2}$$

where M is the vector denoting a list of uncovered variables, N is the list of all applicable $\text{num}_{i,j}$ and A is the binary coefficient matrix obtained from equation (3.1) of size $m \times n$.

3.2. MDP formulation

Markov Decision Process (MDP) is formulating the minesweeper game as $5-tuple$ of $(S, A, T, R.(\cdot, \cdot), \gamma)$ where

- S is set of all the possible states of the game that represents all the possible board configurations at any time. It includes a special state s_{init} which is the initial state of the game as well as the final states s_{lose} or s_{win} which represents the state when the game is lost or the state of winning the game respectively.

- A is the set of all actions that can be performed at any time on a state s to go to some next state s'. For minesweeper, this represents all the covered blocks that can be uncovered for going to the current board state to the state after the block has been uncovered.

- T is the set of transition triplets (s, a, s') which is the probability of going to the next state s' from the current state s by doing the action a. In our consideration, the probability for a being 1 means the player is surely going to lose.

- R is the reward function $R(s, a)$ representing the reward to be received after performing action a on a state s. Our model rewards an action a according to the number of blocks that are deterministically uncovered after it is performed. It is so, because we want to encourage the model to choose the action which would uncover the maximum number of blocks with the least chance of loosing the game.

- γ is the discount factor according to which the future rewards are reduced.

A POMDP is a more general way of formulating the game as it captures the intuition that the entire state is not observable. POMDPs are solved by using belief states, which are generally computed using simulations or other tree search methods with $b_{s,a} = b(s, a, s')$, where, $b_{s,a}$ is the belief on state s on taking action a and observing state s'. This entire method, however, overlooks the fact that a part of the state is fully observable making the problem a Mixed Observability MDPs (MOMDPs). MOMDPs are generally easier and faster to simplify and solve than POMDPs. Our method uses this fact while ignoring the non-observable part of this MOMDP in the belief state to further simplify this problem. Since we neglect both the hidden part of the state and the future observation in this belief, we call it as a sub-state denoted by $s_a = b(s, a)$. Here b is a function which take a section of the board as a subset, based on the action. This sub-state is dependent on the current state and the action to be performed only. The details of how and what sub-states have been used are given in section 3.2.

4. Solving Method

This section contains strategies to find the next best block to uncover. Our solution is based on a two-step process. The first step is called Deterministic Solution Search (DSS), which evaluates solutions for the backbone variables (whose value is the same in all solutions), if they really exist for equation (3.2). DSS collects all deterministic variables which satisfy the constraints in the equation (3.2) and assigns them to be uncovered. The other non-deterministic variables are ignored.

In this way, we do not have to go for any other methods to find the safest block each time. This method proves to be very effective as it has a worst-case time complexity to $\mathcal{O}(n^2m)$. The steps to find deterministic variables are given in Algorithm 1.

Now, if Algorithm 1 returns a non-empty vector there exists some determined blocks which can directly uncover them or flag.

But if it is found that there is no such deterministic variable, we move to the next step. The primary approach of a player is to select the next move based on current

Algorithm 1 : DSS(A,M,N)

$Detrmined \leftarrow$ *empty list*
$loops \leftarrow$ Any significanly large number (say 10)
for $loops$ number of times **do**
    for Each row i in A **do**
        $Variables \leftarrow$ *Number of variables in* A_i
        if $Variables$ is 0 **then**
            Delete row A_i from A
        else if N_i is 0 **then**
            for every variable $var_{i,j} = 1$ in A_i **do**
                $Reduce(A, N, j, 0)$ using Algorithim 2
                Add variable M_j with value 0 to to $Determined$
            end for
        else if $Variables = N_i$ **then**
            for every variable $var_{i,j} = 1$ in A_i **do**
                $Reduce(A, N, j, 1)$ using Algorithim 2
                Add variable M_j with value 1 to to $Determined$
            end for
        end if
    end for
end for
Return $Determined$

Algorithm 2 : Reduce(A,N,j,value)

if $value$ is 0 **then**
    Set column j of A to 0
else if $value$ is 1 **then**
    Reduce value of N_j by 1
    Set column j of A to 0
end if

knowledge of the state and thus we estimate the probabilities of each block being safe or unsafe.

4.1. From CSP

We propose several approaches to find a set of feasible solutions for the CSP using equation (3.2) by which the probabilities for non-deterministic variables are estimated. One can generate a solution set either by using simple backtracking method or by our proposed DSScsp algorithm where, both the algorithms require traversing of a tree.

The backtracking approach constructs a pruned tree where the path for every deepest leaf node is one possible solution. The tree is traversed recursively to gather solutions that satisfy the given set of constraints or equation (3.2). This is implemented mainly to compare with our proposed method.

Our proposed method DSScsp, as given in Algorithm 3, uses DSS (Algorithm 1), at each step to quickly reduce the backtracking search space. The depth of the tree is drastically reduced as several variables are resolved in a single node due to DSS. This leads DSScsp being randomized and much faster than simple backtracking in general. The $Reduce$ function in Algorithm 2 is used in both Algorithm 1 and Algorithm 3.

Algorithm 3 : DSSCSP(A,M,N)

Static $S \leftarrow$ empty list
if A is a Zero-Matrix **then**
    M is a possible solution, add it to S
end if
$i = argmax_{i \in M}(\sum_{j}^{M_i} j)$; where M_i is i^{th} column in M
A', M', N' = A, M, N
M'_i = 0
if $A'M' = N'$ is possible **then**
    $Reduce(A', N', i, M'_i)$
    Run $DSS(A', M', N')$ for further reduction
    $DSSCSP(A', M', N')$
end if
A'', M'', N'' = A, M, N.
M''_i = 1

if $A''M'' = N''$ is possible **then**
    $Reduce(A'', N'', i, M''_i)$
    Run $DSS(A'', M'', N'')$ for further reduction
    $DSSCSP(A''', M'', N'')$
end if
Return S

Using one of these ways, we now have a set of solutions, or a solution matrix S that satisfy the given constraint equation (3.2). In order to limit the time taken to find these equations, we have employed a strategy of just computing a random large enough subset of all the possible solutions over the full set of solutions. We do this by setting

a maximum number of solutions, a maximum depth as well as a maximum number of iterations. The guarantee of the sample being random comes from the fact that, in DSScsp we assign values 1 or 0 randomly.

This set of solutions allows us to compute the probabilities of a block containing a mine.

$$P^\top = \left[\frac{\sum_{i=1}^{s} S_{1,i}}{s}, \ \frac{\sum_{i=1}^{s} S_{2,i}}{s}, \ \ \cdots \ , \frac{\sum_{i=1}^{s} S_{m,i}}{s} \right] \tag{4.1}$$

where, S represents the binary solution matrix and s is the number solutions generated. In S matrix, each row corresponds to a unique solution to equation (3.2) i.e., $S_{i,j} = 1$ only if block j contained a mine in solution i.

The simplest way to proceed with the game after obtaining probabilities is to uncover the block with the least probability of having a mine. But there are ways to improve. We developed several heuristics in addition to these probabilities that play a crucial role and employed machine learning algorithms to have faster execution with more accuracy. However, in minesweeper, even the supposedly best move picked by any heuristic, might result in losing the game.

4.2. Heuristic approaches

We have built heuristics which consider the problem using several other parameters and formulations like size of the board, total number of mines, flags used, the location of blocks etc. which may affect the results. We have used machine learning to find the best heuristics with these parameters.

4.2.1. Manhattan Distance

Legendre et al.[14] consider a heuristic that the blocks closer to the edges of the board are likely to be more informative about blocks involved in equation (3.2) as compared to the farther ones. We consider the same and use Manhattan distance between block at (i, j) to the nearest edge of the board. We also relax the rule of choosing the block with minimum probability. Instead, we choose a list of blocks B_s such that $\forall m \in B_s : P_s \leq \min(P) + 0.05$ where, $P^\top$ is given in (4.1) and $\min(P)$ is the minimum probability in P. Now that we have a reduced list of safe blocks, we choose the one with the minimum Manhattan distance from the minefield boundary. We uncover this block now.

4.2.2. Supervised Learning

This method follows a machine learning approach in which we define the given situation as a $state$ and uncovering a block as a $action$. It is a binary classification model that is trained on the different states and their corresponding actions taken in minesweeper game. We collect the data from the games played randomly by the previous versions. The aim is to classify the $action$ as safe or unsafe for the current $state$. We divide the collected data as x_{train} and y_{train} as follows.

- x_{train}: The input part x that describes the information we know from the current $state$ and an $action$. It consists of the following:
  - Size of the board (p, q),

- Number of mines,
- Coordinates of the covered blocks (i, j): These are the coordinates of the covered blocks in the neighbor (variable boundary) of uncovered blocks. We also consider three random covered blocks in current $state$,
- Probability (P): The probability is the same as obtained from DSScsp algorithm if the covered block lies in the variable boundary in the current $state$. Otherwise, its value is assigned to 0.5 as there is no information given for that block. Noted that the assignment 0.5 describes the chance of having mines as 50%,
- Size of M,
- Variable with minimum probability ($index(p_{min})$): It is the index ($action$) of the covered block from the variable boundary with minimum probability of having mine,
- Score(corner/edge/middle): It is the additional heuristic we use in this method. We take the probability of a block to have no neighboring mine into account so that more blocks can uncover and we get better and more detailed view of the game from the new $state$, In other words, we are considering the next immediate $state$ of the minesweeper game just after we click a block. This score varies as per the orientation of the new block on the board. If f is the number of flags already used and l is the number of uncovered blocks left,

$$\begin{aligned} Score(M_{ij} = 0 \mid (i,j) = corner) &= 1 - \left(\frac{m-f}{l}\right)^4 \\ Score(M_{ij} = 0 \mid (i,j) = edge) &= 1 - \left(\frac{m-f}{l}\right)^6 \\ Score(M_{ij} = 0 \mid (i,j) = middle) &= 1 - \left(\frac{m-f}{l}\right)^8 \end{aligned} \tag{4.2}$$

  As corner block will have the maximum score, it will uncover the larger area of the board that eventually increase chances of winning the game.

y_{train}: The output part consists of a single column corresponding to each data point in x_{train} as y. Its value is 1 if the block turns out to be safe or else it is 0.

We then propose to train two machine learning models with the above data. One is xgBoost and the other is neural Network.

XGBoost- Single-pass Classification: Here, we have implemented binary classification of above state-action pair represented as x_{train} into classes of win and loss as y_{train}. We use single-pass eXtreme Gradient Boost (XGBoost) [20] classifier to do this. It is an ensemble technique that is used to build a predictive tree-based model for handling sparse data where the model is trained in an additive manner. Here, new models are created that predict the residuals or errors of prior models and then added together to make the final prediction. So, instead of training one strong learning algorithm XGBoost trains several weak ones in a sequence until there is no further

improvement. The objective of XGBoost is based on loss function $L(\theta)$ (we use logistic for binary classification) for prediction and a regularization part $R(\theta)$ (we use L2 regularization) that depends on the number of leaves and their prediction score in the model that control its complexity and avoids overfitting. Let the prediction y_i^{pred} for a data-point x_i be,

$$y_i^{pred} \ = \ \sum_j \theta_j x_{ij}. \tag{4.3}$$

Here θ_j are learned by the model from data. Considering mean squared error, loss function for actual value y_i will be,

$$L(\theta) \ = \ \sum_i (y_i \ - \ y_i^{pred})^2. \tag{4.4}$$

Hence, the objective function for XGBoost mathematically is,

$$Obj(\theta) \ = \ L(\theta) \ + \ R(\theta). \tag{4.5}$$

We optimize the above objective function by training from the given data and the model tunes its parameter accordingly. When the model is trained with the data, at every new $state$ we test the above data as input to and the model returns a predicted score y^{pred}, such that $0 \leq y^{pred} \leq 1$ where, larger the value of y^{pred}, more is the chance of that block to be safe. Hence for each $state$, we select the block with the maximum score, to uncover and proceed to another state.

Neural Network: Here, classification is done by iteratively generating a small batch of data and training the model with it. This method allows the model to learn and adapt according to the randomly generated data. The model is a simple multi-layer neural network for binary classification that takes x as its input and returns y as output. Table 1 describes the neural network layers, number of neurons in them and the corresponding activation functions.

We used Relu activation function for all layers except output layer that uses Sigmoid activation function as y between 0 and 1. Here, the loss was evaluated using binary cross-entropy for its logistic behavior and optimizer as Adam. These configurations were done after experimental validation. Our model of iterative classification was inspired by the experience-replay training methods. In this, however, we just train the model iteratively in given number of episodes by generating the training data using the previously made model. Algorithm 4 describes the iterative process of training the neural network.

Table 1: Neural Network

Layer	**Number of Neurons**	**Activation Function**
Input	sizeof(x)	ReLU
Hidden1	sizeof(x)	ReLU
Hidden2	5	ReLU
Hidden3	5	ReLU
Output	1	sigmoid

Algorithm 4 : Iterative Classification

Import $weights$ from current $model$
$n_{episodes} \leftarrow$ number of episodes
$batch \leftarrow 10$
for each $episode$ from 1 to $n_{episodes}$ **do**
    $x_{train}, y_{train} \leftarrow make\_data(batch)$
    Train model with x_{train} and y_{train}
    Save updated $model$ and $weights$
end for

4.2.3. Q-Learning

We use the MDP formulation as described in section 2.2 as a heuristic. As stated there, we have avoided state space explosion by using sub-states instead of belief states. The advantage of using sub-state over belief state is that sub-state can be calculated only using the current state and the expected action. Belief states, on the other hand, need to be calculated by trying to predict the observation, often requiring expensive tree search methods to be computed. It should be pointed out that while our actual formulation is a MOMDP, we have ignored the unknown belief part to improve on computability. In our formulation, these sub-states are of a fixed size $sub \times sub$ with the center of the sub-state representing the block to be uncovered (i.e., the action). We then use deep Q-learning to solve the MDP formulated earlier by learning a Q-function to predict expected discounted reward and we then choose the action with the maximum expected discounted reward.

In Q-learning, we train a model to learn a Q-function given its inputs. The form of Q-function used in our application is given in the equation (4.6). The Q-function will take the same amount of time for every board configuration irrespective of the size of the board. This makes it such that the limitation here is the number of possible actions which is the number of uncovered blocks. This Q-learning can find the next action in $\mathcal{O}(pq)$ where p and q are the dimensions of the given board. computation. The transition rule of Q-learning (SARSA) is given as:

$$Q(s_a, a) = R(s_a, a) + \gamma \max_{a'} Q(s'_{a'}, a') \tag{4.6}$$

Here, s_a represents the sub-state of state s for choosing any action a, s' and a' represents the state at after choosing action a.

However, the problem exists that the direct sub-state is not very representative of the expected reward for any action. For this, we use a score as defined in equation (4.7) to represent every covered block in the sub-state. The score is designed such that it retains information of its own safe probability as well as its location. For a number to represent location we have used the formula in equation (4.2).

$$Sc_{i,j} = \alpha \times P_{i,j} + (1 - \alpha) \times score_{i,j} \tag{4.7}$$

Here $P_{i,j}$ represent the probability of a mine being present at location (i, j), $\text{score}_{i,j}$ represent the location score and α is a variable to be chosen such that the $\text{Sc}_{i,j}$ represents the best possible score. We have used any invalid values to represent an uncovered

block. In order to decrease bias in α, we have defined it as a linear function of board dimension (p & q) and mine ratio $\frac{n}{p \times q}$ as

$$\alpha = \theta_1 \times p + \theta_2 \times q + \theta_3 \times \frac{n}{p \times q} + \theta_4. \tag{4.8}$$

To obtain an α which would give us a good representation of score, we obtained win ratios of different values of α by playing with the heuristic given in section 4.2.1 on the vector of Sc instead of P. We then performed linear regression with the α which maximized win ratio as the target to obtain the values of θ_1 to θ_4. By using the method described above, we can obtain scores $Sc_{i,j}$ for every block (i, j). Using these scores, we can find a sub-state of size $sub \times sub$ for each action a where each element of the sub-state is the score in Sc. We then defined the immediate reward $R(s_a, a)$ for any action a as in equation (4.9).

$$R(s_a, a) = \frac{\text{number of newly opened blocks because of action } a}{(p \times q) - n}. \tag{4.9}$$

We then ran several simulations to find the net discounted expected reward $Q(s_a, a)$ for several action a and sub-state s_a pairs. A neural network of the configuration given in Table 2 was then trained to predict the Q-function. As done in the section 4.2.2, both single pass training and iterative training methods were used. We then choose the action which maximizes the expected discounted reward $Q(s_a, a)$.

Table 2: Network for Q-function

Layer	**Number of Neurons**	**Activation Function**
Input	$sub \times sub$	ReLU
Hidden1	$sub \times sub$	tanh
Hidden2	$sub \times sub$	linear
Hidden3	$sub \times sub$	linear
Output	1	tanh

5. Experiment

For the experiment, we implemented the minesweeper game from scratch along with a playable interface in C++. We then built the various versions of the solver in which every version (except version 6.5 and 3.0) is an improvement over previous versions with a new algorithm or tuned parameters. The versions are numbered as given in Table 3.

By creating several different versions of the solver we were able to see the performance improvement caused by each subsequent algorithm.

5.1. Formulating as CSP

All versions from 2.0 to 4.5 were implemented in C++ for faster performance. Backtracking with DSScsp was implemented iteratively to avoid stack overflow as version 2.0. Version 2.5 is the limited traversal of the search tree by limiting the depth and

Table 3: Versions with algorithm description

Version Number	Algorithm
1.0	Backtracking
2.0	1.0+DSS
2.5	1.0 limited + DSS
3.0	DSScsp + DSS
3.5	DSScsp limited + DSS
4.0	3.0 + Manhattan Distance
4.5	3.5 + Manhattan Distance
5.0	4.5 + Supervised Learning using Single-pass Classification
5.5	4.5 + Supervised Learning using Iterative Approach
6.0	4.5 + Q-Learning using Iterative Classification
6.5	4.5 + Q-Learning using Single-pass Classification **??**

width to be traversed in the search tree. The width is limited by limiting the number of solutions to 100. The depth is limited by limiting the number of swaps, or the number of edges starting from the root node of the tree to 1000. We choose the number of solutions as 100 as it gives reasonably accurate results in much less time.

Version 3.0 is the implementation of DSScsp Algorithm 3 along with Algorithm 1 to find the solution set. This method finds the same solution set as simple backtracking. Hence, it has the same accuracy as that in Version 2.0. But, it is much faster in terms of time taken to solve the minesweeper game.

Version 3.5 is the limited traversal of the search tree traversed in Version 3.0. The time taken is reduced by limiting the depth and breadth of the search tree. The limitations are chosen such that accuracy and amount of time are similar in case of version 3.5 and version 2.5. We have set the limit to the number of solutions to 100 and limit to the depth of the tree to 300, such that the accuracy of Version 3.5 remains almost similar to version 3.0 while reducing the total amount of time.

Versions 4.0 and 4.5 are the extensions of versions 3.0 and 3.5 respectively. They include the heuristic rules defined in section 4.2.1.

5.2. Formulating as MDP

We used machine learning method to perform classification that utilizes the information like $state - action$ pair and their corresponding $result$. We denote this information as one data-point. The data was collected by playing games using version 4.0, with x-dimension of board i.e., p, varying from 5 to 30 and y-dimension i.e., q, as $0.5 \times p$ to p. For each board, we generated at most 100 different mine distribution possible in it. The mine ratios for these boards varied from 5% to 30%.

5.2.1. Classification

For version 5.0, we collected this ample amount of data of around 23.3 million and trained it using XGBoost classifier. We used the XGBoost library in Python to implement this model and saved the trained model as a binary file. This binary file is then accessed in C++ to play the game with version 5.0.

For version 5.5, we collected a batch of data of more than 0.7 million samples similar to the previous version and trained the binary neural network (Table 1). We saved the updated neural network model and then again generated a new batch of data to repeat this process for 100 episodes. With each iteration, the weights got updated and the model learned to classify dynamically. We built the neural network model using keras [21]. As keras is a Python library, we used another library keras2cpp [22], which allowed us to use the trained keras model in C++ as a function.

5.2.2. Deep Q-Learning

In the versions 6.0 and 6.5, the first step in order to build the sub-state was to get parameters as in equation (4.8) for a good value of α. This was done by first simulating several games on different values of α as described in section 4.2.3. We varied α from 0 to 1 with increments of $1/30$ for board configuration with the larger dimension 5 to 10 and 20 with mine ratio from 0.5 to 0.25 playing 100 games in each. We then ran linear regression as in equation (4.8) with α which gave the maximum win ratio with respect to the board size configuration and mine ratio. The regression gave an r-squared value of 0.27 and thus was accurate.

On obtaining α we get the score for each block using equation (4.7) which we could then used to build sub-states of $sub = 3$. For running the simulation to obtain discounted rewards several games were played where the net reward and the action-sub-state pair was recorded. Each individual reward was calculated as given in section 4.2.3.

For the discount, we have used a linearly increasing discount over an exponentially increasing discount so as to punish actions which may lead to a loss severely. This converges as all our rewards are between $0 - 1$, but the discounting factor can increase to any number.

To obtain the simulations, as in section 4.2.2, we have used manhattan distance as base heuristic. We simulated various games with 5 to 20 and 30 with mine ratios from 0.05 to 0.30 and playing 200 different games on each configuration. For each game, we recorded the sequence of a sub-state-reward pair, where the reward was the full discounted reward calculated after the game was over and sub-state is a vector of length 9 denoting the sub-state.

Since the sub-state captures information from both the state and action, a neural network of the configuration in the table (2) built using keras and was trained to learn the sub-state vector-reward mapping with mean squared error as the loss and $rmsprop$ as the weight optimizer. Using this as the first training pass, we then used the model obtained above to get more reward-sub-state pairs from the model obtained after training as described above and trained the model in an iterative model. The iterative training was done on 50 episodes of the larger dimension of 5 to 20 and 30 with mine ratios from 0.05 to 0.20 and playing 5 games on each configuration. This gave us the iterative pass trained model.

5.3. Results

In this section, we study the performance of each version by simulating them on the several minesweeper boards. In this way, we can have exact performance statistics

for each version for their interpretation and comparison. We compare the win ratios of these methods and their dependencies on mine ratio, a number of blocks and dimensional ratio of the board. The performance data is collected by simulation of games where board size is changing from 5 to 15 with dimensional ratio 0.5 (rectangular-board) to 1 (square-board). For all these configurations of the boards, mine ratio is set to vary from 5% to 25% with time out of 5 seconds for each game. As the win accuracy almost converges to 0 after decreasing the density of mines reaches around 25% in any minesweeper board, we can get an illustrative view of this logistics behavior.

Since the version 1.0 is a simple backtracking, it takes a lot of time to execute and that is why we do not consider it for comparison with other algorithms. In fig. 1, we compare the win accuracy of version 2.0 with versions up to 3.5 with respect to different mine ratios. We can interpret from the figure that, version 2.0 performs the worst, as it takes more than 5 seconds (computationally very slow) to solve that causes frequent times out. The accuracy of versions 2.5 and 3.5 are very close to that of 2.0 and 3.0 due to limited traversal. Overall, version 3.0, performs much better than others such as version 2.0, version 2.5 and version 3.5.

This difference of time taken is also illustrated in fig. 5 which shows how much slower is version 2.0 as compared to version 3.0. Noted that with very high mine percentages, version 2.0 begins to lose fast (as the games become harder) making it less accurate. The same figure also shows that the limited versions such as version 2.5 and version 3.5 are much faster than the others, with a very low difference in accuracy as seen from fig. 1. Also, the negative values of time obtained as a result of cubic regression. We compared the performance of version 3.0 with other versions to analyze the effects of adding heuristics in fig. 2 for same minesweeper boards. The heuristic in section 4.2.1 is experimentally proved to be useful for better results as we notice an improvement in version 4.0 over 3.0 which is also evident from the fig. 1.

If we talk about MDP formulations, one of the heuristics, we use is the number of mines left for a state in a game that results in further improvement in the accuracy. Now, as the number of total mines n is increased for a board, that is, mine ratio increase, the effect of left mines on the result decreases. Hence, versions from 5.0 to 6.5 also produce similar results as previous versions and this was also seen in the fig. 2 with mine ratio above 19%. Overall, due to over-fitting, version 5.0 and 6.5 outperforms all other versions with a visible difference with 5.5 and 6.5 following them.

We also analyze the relationship between win ratio of these algorithms with a number of blocks in the board, $p \times q$ in fig. 4 as well as dimensional ratio, $\frac{p}{q}$ in fig. 3 and find that it does not affect the overall accuracy of any versions as such.

The mine configuration of size 9×9 with 10 mines is considered as a standard beginner size board. The accuracy obtained by algorithms presented in this paper compared to [15] is given in the Table 4.

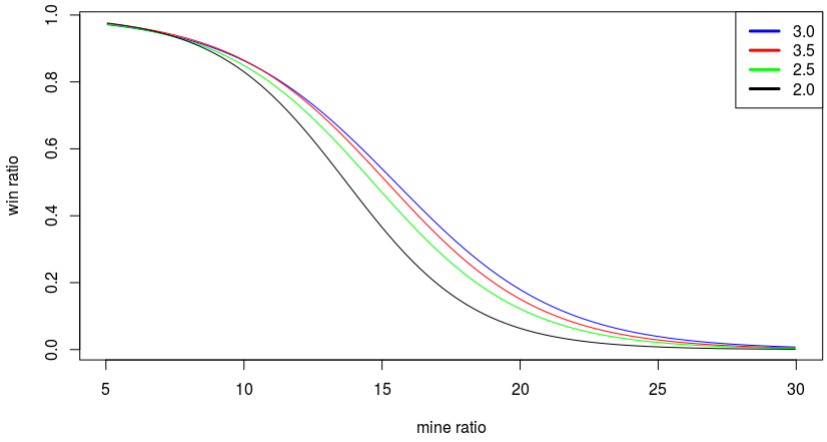

Figure 1: Mine ratio vs. win ratio in CSP based algorithms

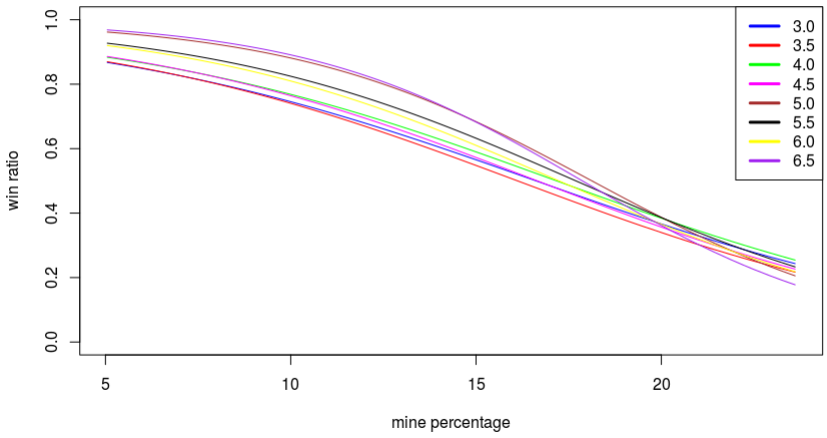

Figure 2: Mine ratio vs. win ratio for all versions in table (3)

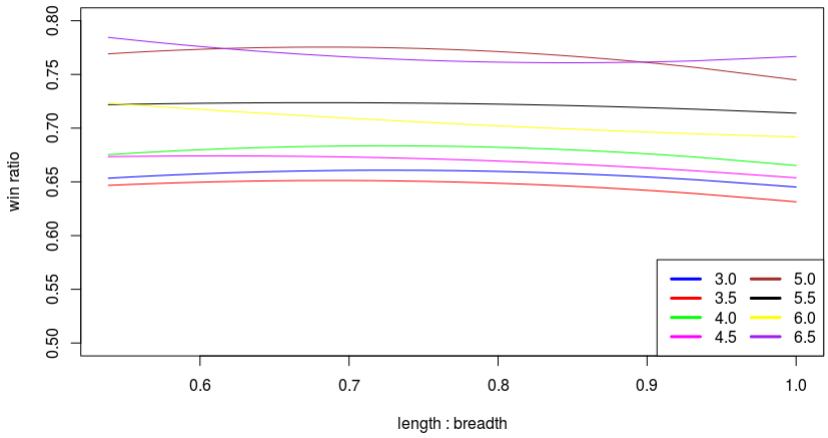

Figure 3: Win ration vs ratio of length to breadth ($p : q$) of the minesweeper board. Lower ratios indicate more “rectangularly” as opposed to higher ratios representing ”squareness”

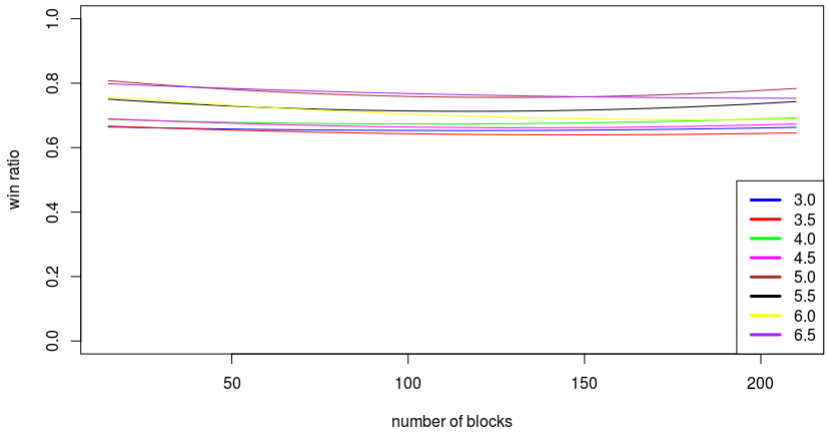

Figure 4: Win ratio vs Number of total blocks in minefield. ($p \times q$)

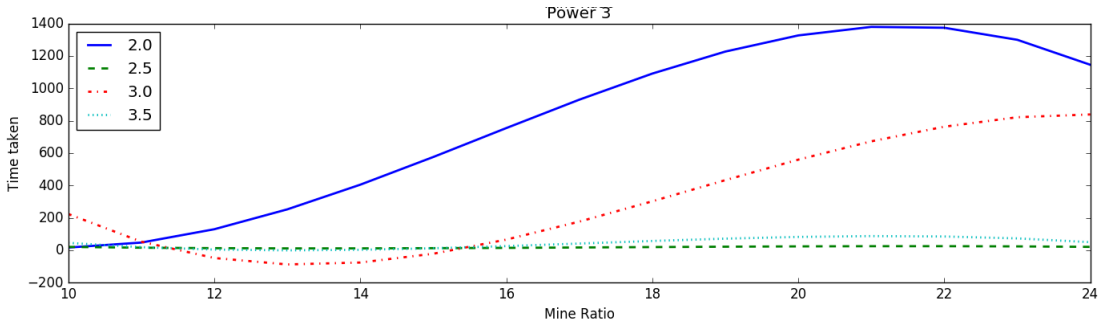

Figure 5: Time vs mine percentage; Part of the graph goes to negative time due to cubic regression limits

Table 4: Standard Beginner Board Accuracy

Version Number	**Accuracy**
2.0	82.14
2.5	77.34
3.0	82.20
3.5	82.20
4.0	86.24
4.5	86.22
5.0	92.02 (max)
5.5	89.21
6.0	87.58
6.5	91.41
OH [15]	89.9

6. Conclusion

In this paper, we have used the realization of the minesweeper game as an MDP and as a CSP to create several kinds of solving methods. We introduced DSScsp (Algorithm 3), which can enumerate all solutions of the CSP much faster than backtracking while being just as accurate. We also introduce the concept of limited traversal which obtains a subset of all possible solutions and showed that it gives nearly as accurate results as full traversal, while being much faster. We improved preexisting heuristics (Version 4.0) and introduced new ones which used machine learning and deep Q-learning (Version 5.0 & version 6.5) on the MDP formulation of the game which we showed to be better than that of existing methods (Table (4)). We also showed that deep Q-learning is marginally better than machine learning (classification) methods in general, but not always as can be seen in the Table (4). Overall, we claim that our method of deep Q-learning (Version 6.5) can play minesweeper to a high degree of accuracy while still being fast enough to play boards as large as 200 total blocks.

References

[1] R. Kaye, Minesweeper is np-complete, In The Mathematical Intelligencer 22 (2000) 9–15.

[2] R. H. Yap, Constraint processing by rina dechter, morgan kaufmann publishers, 2003, hard cover: Isbn 1-55860-890-7, xx+ 481 pages, Theory and Practice of Logic Programming 4 (5-6) (2004) 755–757.

[3] M. Fellows, T. Friedrich, D. Hermelin, Constraint satisfaction problems: Convexity makes all different constraints tractable, In Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence (2011).

[4] N. Bouhmala, A variable depth search algorithm for binary constraint satisfaction problems (2015).

[5] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, M. Riedmiller, Playing atari with deep reinforcement learning, https://arxiv.org/abs/1312.5602 0 (2013) 1.

[6] C. Studholme, Minesweeper as a constraint satisfaction problem (2000).

[7] D. Becerra, Algorithmic approaches to playing minesweeper (2015).

[8] R. Dechter, D. Frost, Backjump-based backtracking for constraint satisfaction problems, In Artificial Intelligence. v.136 (2002).

[9] K. Bayer, J. Snyder, B. Y. Choueiry, An interactive constraint-based approach to minesweeper, July, 2006, pp. 16–20.

[10] P. Nakov, Z. Wei, MINESWEEPER, 2003.
URL `http://www.cs.berkeley.edu/\begingroup\let\relax\relax\endgroup[Pleaseinsert\PrerenderUnicode{}intopreamble]zile/CS294-7-Nakov-Zile.pdf`

[11] B. Bonet, H. Geffner, Belief tracking for planning with sensing: Width, complexity and approximations, In Journal of Artificial Intelligence Research. 50 (2014) 923–970.

[12] L. P. Castillo, S. Wrobel, Learning minesweeper with multirelational learning, in: Proceedings of the 18th International Joint Conference on Articial Intelligence, 2003, pp. 533–538.

[13] M. Sebag, O. Teytaud, Combining myopic optimization and tree search: Application to minesweeper, In Learning and Intelligent Optimization (2012).

[14] M. Legendre, K. Hollard, O. Buffet, A. Dutech, Minesweeper: Where to probe?, RR-8041, INRIA (2012).

[15] O. Buffet, C. S. Lee, W. Lin, O. Teytaud, Optimistic Heuristics for MineSweeper, In International Computer Symposium, Hualien, Taiwan, 2012.

[16] A. Couetoux, M. Milone, O. Teytaud, Consistent belief state estimation, with application to mines, In International Computer Symposium (2012).

[17] L. Gardea, G. Koontz, R. Silva, Training a minesweeper solver, An Autumn CS 229 (2015).

[18] C. J. Watkins, P. Dayan, Q-learning, Machine learning 8 (3-4) (1992) 279–292.

[19] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, et al., Human-level control through deep reinforcement learning, Nature 518 (7540) (2015) 529–533.

[20] T. Chen, C. Guestrin, Xgboost: A scalable tree boosting system (2016).

[21] F. Chollet, keras, https://github.com/fchollet/keras (2015).

[22] P. Plonski, keras2cpp, https://github.com/pplonski/keras2cpp (2016).