Artificial Intelligence

American University *of* Armenia

Report

Minesweeper

Team:
Diana Sargsyan: diana_sargsyan@edu.aua.am
Mher Movsisyan: mher_movsisyan@edu.aua.am
Lilit Beglaryan: lilit_beglaryan@edu.aua.am

Supervisor
Prof. Monika Stepanyan

Contents

1 Abstract

Since 1995, several methods have been used to develop minesweeper agents, including the use of genetic algorithms, and other learning techniques and strategies. In this paper, in addition to discussing many available solutions to Minesweeper,we will suggest a Constraint Satisfaction Problem (CSP) approach using constraint propagation, logic inference, and search.
Keywords: minesweeper, constraint satisfaction problem,logic inference, forward-checking, depth-first search, random search, simulated annealing, hill climbing.

2 Introduction

Minesweeper which is designed for an n by m board to be played by a single agent aims to reveal the randomly located k mines within the shortest time possible [Bec15]. It is usually available by default in MS Windows and some other operating systems. In the process, to track the "mine" squares, the player can put flags on uncovered cells that might potentially contain mines. The agent plays the game by trying to uncover the "safe" cells and avoid the "mine" cells. Whenever the agent uncovers a "mine" square, the game is considered over. Otherwise, if the square revealed happens to be safe, an indicator number between "1" and "8" is displayed on it that shows the cumulative number of mines among the neighbouring squares of the clicked cell. Each cell is considered to be adjacent to eight cells, horizontally, vertically and diagonally, hence the maximum value of the indicator is "8". An indicator of "0" is not written explicitly on the square, rather it is marked by a blank cell and means that all the neighbors are "safe", while "8" implies that the cell is surrounded entirely by mines [Bec15].
The picture in Figure 1 shows that the cells with no neighbouring mines are left blank, indicator numbers "1","2","3","4" represent the corresponding number of mines in adjacent cells [enc22].

Minesweeper requires from our agent to be reasonable. The indicators hint the agent about the possible locations of the mines. The original version of Minesweeper does not let the agent lose in case it uncovered a mine cell for the first step. In case the agent clicked on a mine, the mouse pointer is moved to another randomly chosen square that does not have a mine instead of the selected mine square [Tho].

Structure of project paper: the paper consists of two main sections. The first section contains literature reviews of other existing solutions for minesweepers. The next section describes the combination of algorithms proposed by our group used to solve the problem.

Figure 1: **Example of Minesweeper board** [enc22]

3 Literature Review

The most well-known approaches to solving Minesweeper are Random Search (RS), Hill Climbing (HC), Simulated Annealing (SA) and Genetic Algorithms (GA).

1) Random Search (RS) [FM]

Random Search can be used to solve optimization problems by cutting down the time spent on considering all potential solutions to the problem and generating random solutions then picking the best of them. Minesweeper, not being an optimization problem can be brought to one, so that RS is applied to it. For that, a penalty function is used that for every indicator i, computes the squared difference between i and the number of hidden mines in an adjacency region of cell j, i.e. at each stage penalty function's value gets added by $(i-j)^2$.

Hence, whenever the board is solved correctly, i becomes equal to j for every cell(number of mines to be found nearby gets equal to the number of uncovered mines which implies that all safe cells have been uncovered on the board). And in that case, the error $(i-j)^2$ becomes 0 and penalty function obtains value of 0(See **Figure 2**.

2) Hill Climbing (HC) [FM]

The drawback of RS is that when encountering a locally correct solution, it does not continue to improve the solution further for the rest of the board. Instead, RS is starting a new random solution generation; hence RS helps to find only reasonable solutions in some parts of the board, but it is very rare to output a solution correctly for the whole board.

Here comes the approach called Hill Climbing which assists in overcoming this problem. Like the RS, initially, Hill Climbing generates a random solution. Later,

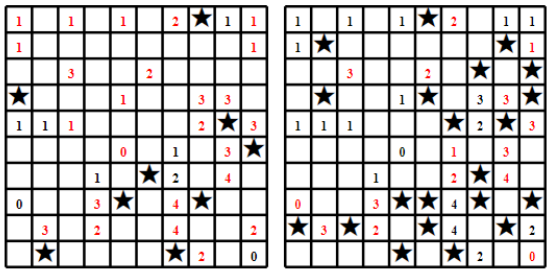

Figure 2: **Example of random search in Minesweeper** [FM]
In the left picture of Figure 2, indicators are randomly generated; some are truly signifying the number of mines surrounding the cells, while others are incorrect. The penalty value of the generated configuration on the left is $60 = 8*(1-0)^2+4*(2-1)^2+2*(2-0)^2+(3-0)^2+4*(3-1)^2+2*(3-2)^2+3*(4-2)^2+(0-1)^2$ After 1000000 random generations, the picture on the right was obtained with a lower penalty function value, 15(calculated in the same way, red colored indicators were generated wrongly, subtract from them the uncovered mines, raise to power 2 and add the errors).

instead of regenerating new potential solutions, HC makes small changes locally with the hope of improving it. If the change does not result in a desirable improvement, HC returns to the previous state.

Figure 3 contains an example of the HC approach to Minesweeper. At each iteration, HC removes or adds a mine in a random location, and if the change minimizes the penalty function, we keep it. If not, we discard the change and go back. After 1414 iterations, updating the position of any mine does not yield a better solution so HC stops there. It outputs not the correct solution but a relatively good one(with a penalty of 5) that tends to the best solution. The advantage of HC over RS is its speed in reaching a comparatively good solution.

3) Simulated Annealing [FM]

HC having an advantage over RS, still has drawbacks. It finds only a local optimum(a configuration that optimizes the penalty function's value). However, if locally HC is still looking for a better solution, it halts without trying to find the global optimum. In the **Figure 4**, HC could halt on the height A since it cannot find a better value in the surroundings, while we need to find D.

Simulated Annealing(SA) comes to resolve this issue. Like the two mentioned approaches above, SA also starts with a random configuration of the board and keeps a temperature value(t) at each iteration. t is decreasing as SA proceeds further to the next iterations.

Unlike the HC, which completely discards bad configurations in the local context of the board, SA assigns probability values to each locally bad state $E(-c/t)$ where t is

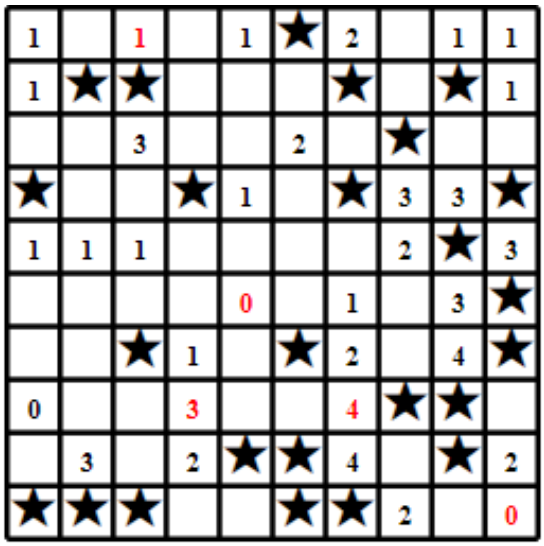

Figure 3: **Example of Hill Climbing in Minesweeper** [FM]

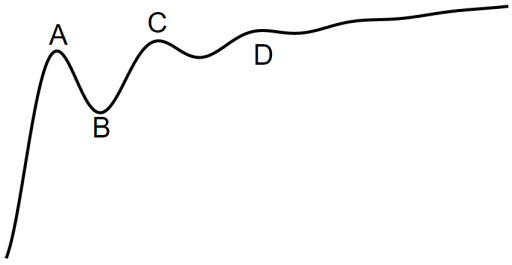

Figure 4: **Example of Simulated Annealing in Minesweeper** [FM]

the temperature value of that iteration and c is the difference between the previous and current penalty values. Following this probabilistic approach, initially, when t is very high, $(-c/t)$ is very high, and therefore all initial changes are accepted blindly(similar to RS). When the temperature is medium, and the state is a local optimum, SA will not most likely go downwards, but it will try to explore for a better change locally. An example can better explain this. In **Figure 4**, in the beginning, when t is very high, SA will fastly progress, going over the valley between A and C in search of a higher top. When it is already in the valley between C and B(here, the t value will be lower), it will not likely go back to the lower A-C valley. We can see similarities between SA and RS when initially the temperature value is higher and between SA and HC when the temperature gets lower and lower, and SA only tries to optimize locally.

4) Genetic Algorithm [FM]

Another approach called Genetic Algorithm(GA), the idea of which is derived from natural selection in biology, is used to overcome the shortcuts of HC and RS algorithms. Here, instead of starting with one random solution and trying to improve it in further iterations, GA keeps a pool of possible solutions called "population". Each member is assigned a "fitness" score. And in later iterations, the members with the highest scores are crossed together(like in the natural selection process), and we get successors that are the "fittest." In generating new configurations(successors) of the board, random mutations are possible, similar to RS, and SA, where the following configuration results from mutating the current state. GA is similar to HC in that its fitness function can be the inverse of the penalty function of HC since the smaller the penalty value, the higher the fitness of that state. The cross over of the two fittest solutions A1 ans A2 among the population is accomplished in the following way: randomly take a line crossing the new empty board, one side of that line will be populated by one part of A1, and the other side will be taken from A2. And mutations might apply to the child. In **Figure 5**, see the children obtained after the given number of cross-overs. The penalty value decreases as GA generates fitter solutions.

We decided to implement the game of Minesweeper, defining it as a Constraint Satisfaction Problem (CSP). CSP is a problem representation form where states must satisfy predefined constraints [Bec15]. CSP consists of

- Variables
- Domain of the variables (sets of possible values)
- A set of constraints (a set of limitations imposed on the variables)

The goal is to assign values to the variables so that no constraint is violated and all variables have assigned values, making the solution complete and consistent [Bec15].

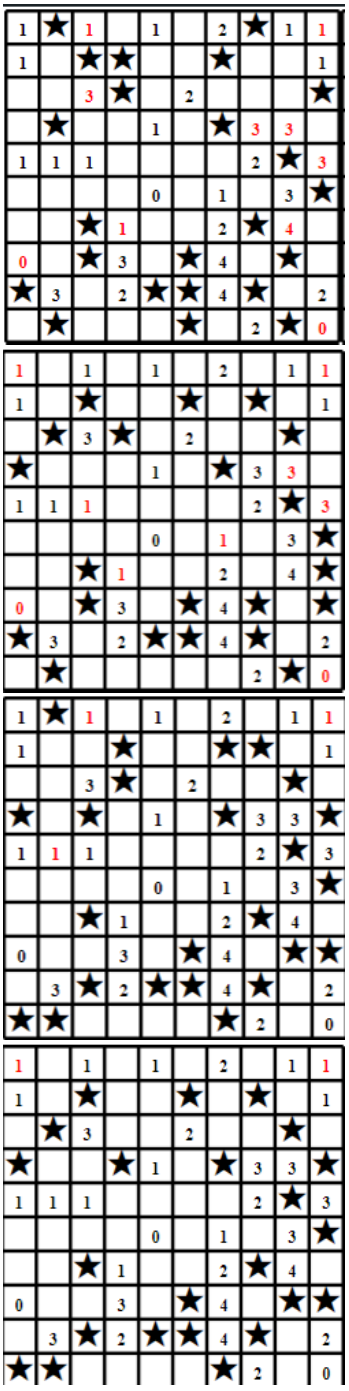

Figure 5: **Example of Genetic Algorithm in Minesweeper: after 1000,2000,5000,10000 steps** [FM]

4 Method

4.1 Background

Our group implemented Minesweeper in Python programming language. We formulated and solved the Minesweeper game as a constraint satisfaction problem (CSP), while also using logical inference. We assume the board is ten by ten and the number of mines is 10. The variables are the 100 squares, each of which is either a "mine" cell or a "safe" cell, indicated by values of -1 or 0 respectively (the domain for each variable is {-1,0}). The constraints keep the relationship between each cell and its neighbouring cells. Each constraint is associated with a marker number called indicator(a value from {0,...,8}) that shows the number of neighboring "mine" squares of an explored cell. The sum of the absolute values("mine"- (-1), "safe"-0) of the variables in a constraint of an explored square should add up to its indicator (i.e., the number of mines around the square should be equal to its indicator).

4.2 Algorithm

The algorithm has multiple components which come into play at different times based on the state the game is in, and are not necessarily executed sequentially. Firstly, the game starts by the agent choosing to uncover the square (1, 1). If the indicator of the constraint put on that square is not "0", the agent uncovers the corner (n, n). If the same happens to that corner too, the agent opens the third, sometimes even the fourth corner. The initial openings of the corners are done only when the agent doesn't have a certainly "safe" square to uncover. This initial component is not performed if more than 12 iterations have passed, and is only here as an alternative to randomly picking a starting point. Choosing the corners is a childhood tradition of ours. Note that we can uncover a "mine" in the first iteration of the game, which differs from the classical version of Minesweeper where a random "safe" square gets revealed instead, and showcases the beautiful cruelty of this world.

Secondly, if there are uncovered squares that have an indicator value of "0", the algorithm first chooses to open its adjacent covered squares, since we are sure that all of them are "safe". If there are uncovered squares whose indicator value is equal to the number of adjacent covered squares, this means all the hidden neighbours are "mines". So it chooses to flag those covered squares as "mines" and decrements the adjacent indicators of the flagged square by one for each flagged square. Once these two have done their job, the algorithm iterates over each square, checking if a close-by square (with a Manhattan distance of 2 or less) is a square whose constraints form a superset of its constraints. In the case that we find a square the constraint set of which forms a superset of the first square's constraint set, we decrement the second square's indicator by the indicator of the first square and remove the constraints of the first square (the intersection part) from the second square's constraint set.

This way enforcing consistency over the constraint space becomes faster since we are shrinking the space size.

Finally, after exhausting all the inference steps possible using the previous components, the agent searches through all possible permutations of mine placements in the remaining covered squares. We refer to each of these permutations as universes. If a universe consistent with all constraints is hypothesized by the agent, the mine locations are logged for later use. Once the agent is done accounting for all possible universes, the number of possible universes where a square had a mine will be used to determine which square is best to uncover. The agent takes the square with minimum number of mines in the multiverse(the number of possible universes where there was a mine in that square) since it is the most likely to be a "safe" square. This way we minimize the chance of hitting a "mine", whether or not the search resulted in finding certainly safe squares.

4.3 Time complexity

Because of the exhaustive search at the end, the worst case time complexity is $O(2^r)$ where r is the number of covered squares left. If the initial components of the algorithm succeed in inferring the entire board, the worst case complexity becomes $O(n^4)$ where n is the dimension of the board (10 in our case).

5 Results

The algorithm was tested for randomly generated instances of size 10 by 10, and the correct solution was found in 104 of 130 cases. All of the cases where the agent exploded on a mine were during the first few iterations of the game, when the agent is required to choose a corner and has no percepts to process. We also tried various board sizes (8x8, 14x14, 16x16, 20x20) and mine quantities (8, 14, 16, 20, 44), during which we observed no significant difference in the success rate when $\frac{n}{k} = 1$, n being the length of the board, k being the number of mines. However when having a mine quantity of 44 and a board size of only 16x16 ($\frac{n}{k} = 2.75$), the game became over-constrained and the agent started the exhaustive search too early, spending too much time searching for a good move (in the multiverse context). Here are some examples on how the game proceeds:

Click on **Minesweeper Demo** to see our demonstration of the game.

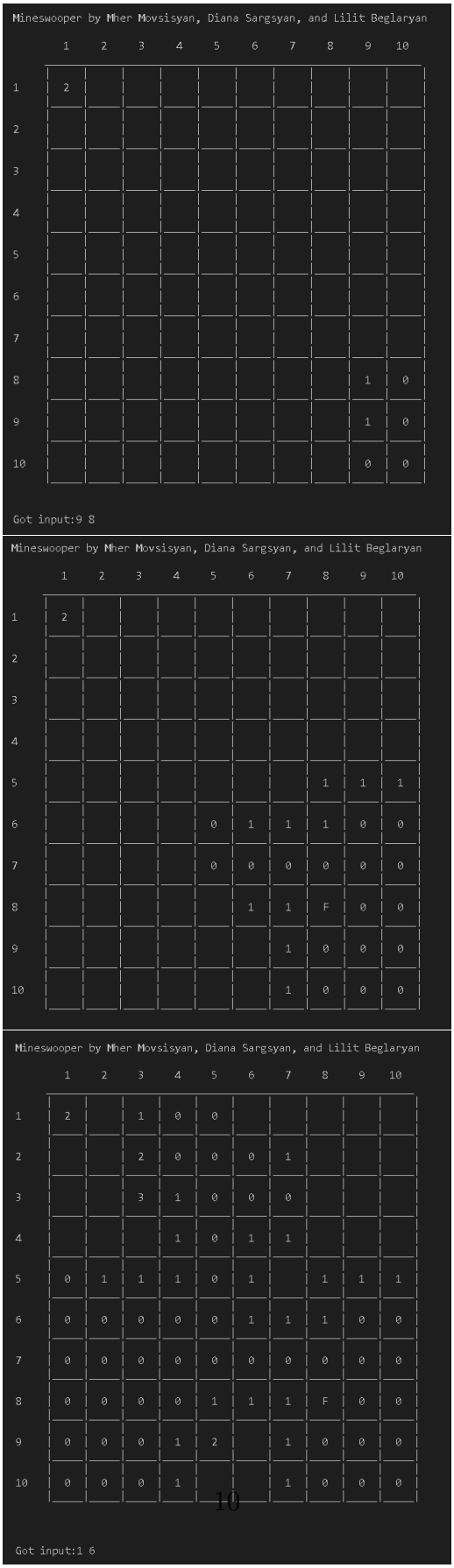

Figure 6: **Example of Minesweeper in Minesweeper: after three steps** [Mov]

References

[Bec15] David J Becerra. "Algorithmic approaches to playing minesweeper". In: (2015). URL: https://dash.harvard.edu/bitstream/handle/1/14398552/BECERRA-SENIORTHESIS-2015.pdf?sequence=1&isAllowed=y.

[enc22] Wikipedia the free encyclopedia. *Minesweeper board*. Wikipedia, the free encyclopedia, 2022. URL: https://en.wikipedia.org/wiki/Minesweeper_%28video_game%29.

[FM] Informatics Faculty of Mathematics and University of Warsaw Mechanics. *Lecture 11: Solving hard problems*. URL: https://www.mimuw.edu.pl/~erykk/algods/lecture11.html.

[Mov] Mher Movsisyan. *Mineswooper*. URL: https://github.com/MovsisyanM/Mineswooper. (accessed: 06.12.2022).

[Tho] Thowarth. URL: http://web.stanford.edu/class/archive/cs/cs221/cs221.1192/2018/restricted/posters/thowarth/poster.pdf.