Final Report
Minesweeper's destroyer v1.0

MATH2020 – Discrete Mathematics

Prof. Tu Nguyen

Thursday, December 9th, 2021

Team BackUp

Vu Nguyen, Nguyen Hoang, Phuc Vu, Trung Pham

1. Introduction

1.1 The Minesweeper Game

Minesweeper is a popular game pre-installed in many operating systems. The board is divided into squares, and mines are scattered about randomly. The number in a cell indicates the number of mines nearby. The right mouse button can be used to flag squares that are suspected of being mines. The player wins by opening all the squares without triggering the mines.

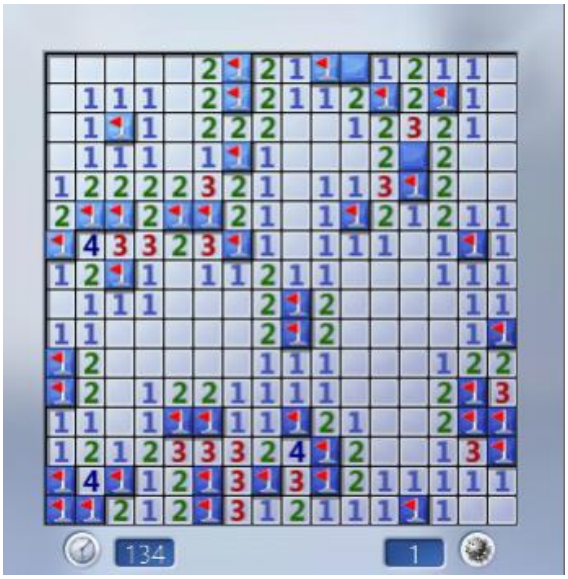

Figure 1. The Minesweeper Game

1.2 Project's objective

The main goal of the project is to implement a simple program to achieve an acceptable win rate in the hard mode of the game Minesweeper, using:

- Board reader: We can get a bitmap of all the pixels on the board by using the screenshot function. Next, we read the numbers on the computer screen by utilizing their different coloring.
- Logic: An algorithm to enumerate all possible configurations then using logic math to choose the best result.
- Board clicker: Use the Robot class in the standard library to click on the squares on the board.
- Probability: Sometimes luck is needed to win the game, we would then use probability to optimize our luck and secure victory

1.3 Mathematical background

The two main concepts of discrete math that is mostly used throughout the project are: Logic math and probability.

a. Logic Math

Logic math plays a huge role in our project. Initially, our idea to approach the problem was to describe the states and information of each square using logic math (as demonstrated in backtracking) and then solve these logic equations to find the roots (the safest square). To solve these equations, we first came to Prolog, a logic programming language which is able help us generates all the valid roots for these equations (see section 7 for more details). Yet, due to difficulties in implementing Prolog in Java, we finally came to the final solution of Backtracking with constraint instead, which turned out to work quite well.

b. Probability

A win in minesweeper is not a 100% win, there are cases requiring us or computer to play the next move with the highest probability, or even worse, 50/50 random guess; therefore, a backtracking algorithm with probability recorded for each move is an important part in this mine solver, especially in advanced mode. The probability of surrounding squares can be estimated based on open squares and the center square value which provides the information of the number of mines in the surrounding area.

1.4 Code Structure & what is new

Stage 1: Reading the screen

The program starts its calibration and detection routines, take screenshots and detects grid of minesweeper on the main screen, it then read and mark the current stage of the board on a 2D array (OnScreen) (see more on section 2.1). This whole module of scanning the grid and mark the status on a 2D array is inherited from the GitHub of Bai Li (luckytoilet)

Stage 2: Minesweeper solver

This stage contains two major parts: Flagging and finding a safe square

- **Flagging:** Scan the board throughout. If there is an opened (numbered) square with the number of unopened squares surrounding it equal to the current number of mines yet to discovered surrounding it (which is equal to number on the square - number of flags surrounding it) then we flag all the unopened squares surrounding it. This process can be found in the function “attemptFlagMine()” in the code.
- **Finding a safe square to click:** In this we would use the three method we mentioned, including straightforward method, backtracking with constrain and probability to find the square that is confirmed to be safe in all cases so that we can make our move on (see sections 2.2, 2.3, 2.4, 2.5 for more detail).

The interaction on the minesweeper grid, including clicking, chording (double-click), flagging, use some help from the Robot class in Java standard library. The idea was inspired from David Hill’s Minesweeper analyzer, and MSolver by Li Bai. Implementation, analyzing, optimization was done by our team.

2. Implementation

2.1 Reading the field

"Reading the field" exists for our bot to recognize the minesweeper grids and diverse types of squares of the game by applying the following two routines, which are the calibration routine and the detection routine. At the beginning of the game when we still have a completely empty grid, the calibration routine would be invoked to determine the location, the size of the grid square as well as the dimensions of the minesweeper's board. The basis or the main idea is to first take the screenshot of the board and then analyze it by applying the heuristic algorithm to get the exact location.

The Minesweeper's destroyer not only needs to locate exactly all the square's locations but also needs to have an ability to read or determine what is the number behind the pointed unopened square. To implement this "reading" function, it is compulsory to pass a small section (which is cropped from the original snapshot) of the previous screenshot, which the bot took from the Calibration, then insert it into the detection routine. Next, the routine would look at a few pixels and it would be able to determine which number behinds that square.

The detection routine principles are quite simple. We would try to compare colors between opened and unopened squares because each type of square has a distinct color.

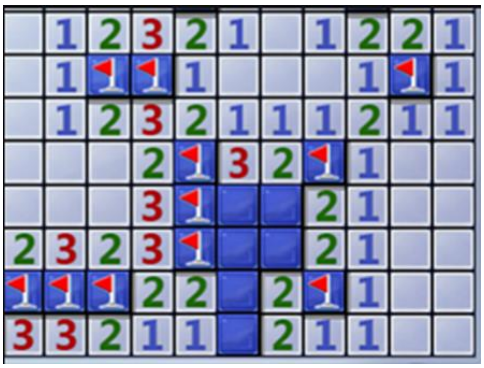

Figure 2.1 Opened and unopened squares having different colors

2.2 Straightforward algorithm

We would first approach the game with a simple brute-force algorithm.

In the beginning, when no square is uncovered, we made a random move. It is guaranteed that the first square we click on will not be a mine because the Minesweeper game is designed to make the 1st click safe.

Consider an opened square X that has a number on it. Look at the squares immediately surrounding X. If the number of unopened blocks is equal to the number on X, then we flag every unopened block surrounding it. If the number of flagged squares equals the number on X, then click on the unopened, unflagged squares around X.

Figure 2.2.1 Example of straightforward algorithm

For instance, look at Figure 2.2.1 The middle square with number 2 on it has already been two flags surrounding it, so we can click on all the other surrounding squares.

Figure 2.2.2 Minesweeper's beginner mode solved using straightforward algorithm

The straightforward algorithm is good enough for a beginner mode, but as we aim to conquer the highest level of the game, another approach must be considered.

2.3 Backtracking algorithm

We'd always start with the straightforward algorithm until it gets stuck, then we would need a better approach. Consider the following case:

Figure 2.3.1 One case where straight forward algorithm is stuck

Using the basic algorithm, we seem to be stuck. Up to now, we've only considered the information we have for a single square, which is not optimized.

If we only consider the lower "2" (one with red dot), it is obvious one of the two squares that we mark contains a mine in it, yet we do not know which one and this information is useless to us.

Yet when combine this information with the upper "2" the one with the green dot, we can see that the area that is covered with the white line is perfectly safe

Mathematically, if we mark the square as below and consider A, B, C, D respectively as the event of the square A, B, C, D contains a bomb. We can see our case as the following logic equation:

$$(A \cup B) \cap (A \cup B \cup C \cup D) = A \cup B$$

Which leaves C and D out of the result, which means C and D contain no mine.

Figure 2.3.2 The case is described under logic math perspective

More broadly, we can combine more information with the upper squares and mark them as follow:

Consider A, B, C, D, E, F respectively as the event of the square A, B, C, D, E, F contains a mine. With the known information of the adjacent numbered squares, we'd have the following:

$$\begin{cases} (A \cup B) \cap (A \cup B \cup C \cup D) \\ D \cup E \\ E \cup F \end{cases}$$

$$<=> \begin{cases} (A \cup B) => C, D \; are \; safe \\ E \; (knowing \; D \; is \; safe) \\ E \; (as \; a \; result \; of \; D \; is \; safe) \end{cases}$$

$$<=> \begin{cases} (A \cup B) \\ E \end{cases}$$

Figure 2.3.3 Taking into account information of upper squares

Hence there are 2 possible cases for us to consider:

Figure 2.3.4 There are mines in B and E

Figure 2.3.5 There are mines in A and E

In either case, we know that C, D, and F are safe, and E is surely mined, which open us to more information to solve the configuration above

Figure 2.3.6 Deduced safe squares

It is hard to make the computer think deductively like this. Yet from this process, one thing we can use to implement the mine solver algorithm: adjacent squares can provide information to each other.

This has given us an idea of a backtracking implementation:

- **Step 1:** We can save a list of the squares that we have partial information about - unknown squares that is adjacent to a known square (numbered square),
- **Step 2:** Mark each of the square to be a mine or not, checking all configuration possible
- **Step 3:** Comparing each of the configurations with the numbered neighboring squares to see whether it is valid. If it is not, we move on to the next configuration.
- **Step 4:** If it is a valid solution, we add it to our list of solutions
- **Step 5:** Consider our list of solutions, make a move on the squares that are safe, Flagged the squares that we are sure that contains a mine, and update the screen reader. A move is considered safe if in all valid solutions it is empty (does not contain a mine). On the contrary, a square is surely a mine if in all valid solutions it contains a mine.

This algorithm has time complexity in worst cases of 2^n, with n is the number of unknown squares with partial information, which is good enough for us to try. Note that we will update configuration every time we make a move (finding out more information) so that our root is always the best root possible.

2.4. Splitting the board into regions of squares

A typical board in hard mode would be a 16x30 matrix, which means there is a possibility of cases where we must take into account a large number of squares to backtrack.

Considering the following case:

Figure 2.4 One case where segregation is useful

The region on the left alone has 23 squares with partial information. This means that the worst cases of the backtracking algorithm can be up to 2^{23} and more.

However, if we look closely, we can see that the squares that are marked in different colors give heuristic no information about each other. More generally, we found that 2 squares give information about each other only if they share an open square (a numbered square). This gives us an idea of splitting the partially informed squared into regions and then backtracking on them. We declare that a square belongs to a specific region if it shares at least a common numbered square with a square that is already in those regions.

Applying this method can largely reduce the complexity of the algorithm. In the case above, by splitting into regions, the complexity then would be $2 + 2^6 + 2 + 2^5 + 2 + 2^4 + 2 + 2^4 = 136$, which is a lot less than 2^{23}

2.5. Using probability to maximize the chance of winning

When there is not much information on the board, for example at the start of the game, the probability would be used for the algorithm to make the best prediction given limited information. Consider the case:

Figure 2.5.1 An example requires probability

First, the number 3 in the middle gives us a clue that all 3 squares around the middle 3 are mines. Given those 3 squares, we cannot directly infer which next square should be marked with 100% certainty and that comes to the important role of probability.

As mentioned above, the backtracking algorithm generates all possible outcomes for the next move:

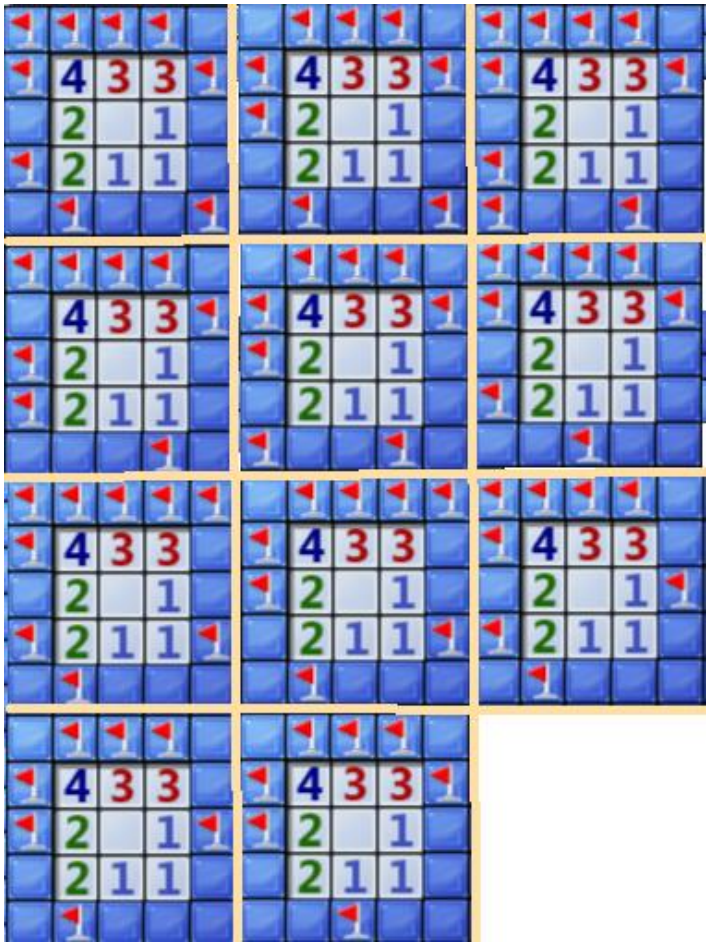

Figure 2.5.2 Backtrack and keep track of probability

All 11 possible outcomes are independent so that the probability that a square contains a mine is the sum of the probability of that event in each outcome.

Further, if we let M_{ij} be an event such that:

$M_{ij} = 1$ when in i^{th} outcome, square j^{th}contains a mine

$M_{ij} = 0$, otherwise

Let M_j denotes the number of outcomes that j^{th} square contains a mine, then:

$$M_j = \sum M_{ij}$$

Figure 2.5.3 Calculate probability and make decision

Let X_j be discrete random variable such that:

$X_j = 1$, square j^{th} contains a mine

$X_j = 0$, otherwise

Therefore, the probability of j^{th} square not containing a mine is:

$$P(X_j = 0) = 1 - P(X_j = 1) = 1 - \frac{M_j}{total\ outcomes}$$

In this case, the next marked square would be the square with the smallest number (2) and that prediction stands an $1 - \frac{2}{11} = 81.82\%$ of being correct.

However, some cases do not necessarily guaranty a 100% win. Consider the case below:

Figure 2.5.4 An example requires a 50/50 random guess

The numbers 3 and 1 at the bottom right corner indicate that there is a mine among 2 unknown squares. In such a situation, it is almost impossible to come up with an algorithm with more than 50% accuracy other than a random guess.

3. Discussion: another approach using logic programming

Although, we have mentioned that it is hard to describe the information given by numbered squares, it is not impossible. During our research, we have found prolog or logic programming languages in general are built to solve problems like this. Logic programming is a programming paradigm which is largely based on formal logic. Any program written in a logic programming language is a set of sentences in logical form, expressing facts and rules about some problem domain (Aral & Gelfond, 1994). Briefly speaking, Logic programming allows us to solve problems in a declarative way.

Consider the following case

Figure 3. Example of states describe in prolog

Logic programming allows us to take each piece of information into account and generate all possible roots for us. From Logic programming perspective, the problem is stated as follow:

- The left most square “1” states that there is either a mine in cell A or cell B
- The square “2” states there are 2 mines in either (cell A and B) or (Cell A and C) or (Cell B and C)
- The right most square “1” states that there is either a mine in Cell B or Cell C

All this information would be taken into account with operation AND (conjunction). Prolog would then check and output all the configurations that meet our listed requirements. This could have replaced our backtracking algorithm to find valid solutions as it is much simpler and easier to code. However, as it is prolog is not strongly supported on Java, it made the code quite buggy and hard to debug, yet it is still a nice approach for a minesweeper solver.

4. Result

The solver works quite well with the winning percentage of 43% in 50 game with the best time of 22 seconds

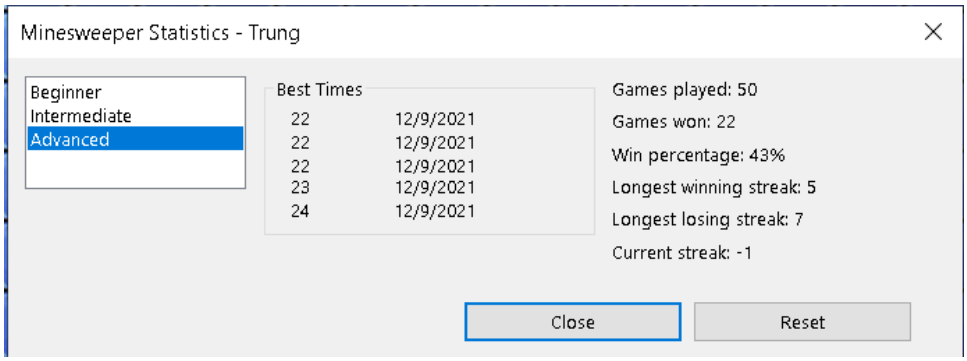

Figure 4.1 Statistic of Minesweeper destroyer

The cause of losing is most likely because minesweeper is still a game of luck. In many cases, the game put us in a 50-50 situation where there is no logical path forward. In other cases, the probability solver just guesses wrong. To illustrate, this is one of the most common pattern Minesweeper usually pulls in the end game:

Figure 4.2 A case where luck is required

Overall, the performance of the solver is acceptable, especially if we consider 2.4% win rate of all game in expert mode and 39% win rate of the best player on “minesweeper.online”, a popular community of minesweeper.

Reference

David, Hill. (n.d.). *Minesweeper player, solver and analyser in javascript*. GitHub. Retrieved November 18, 2021, from https://github.com/DavidNHill/JSMinesweeper?fbclid=IwAR35zzHIeki9VuDRiR3ycjxHKZDHBxC3xFwGpj1hdBM9R7Yix8cGxP7sZgA#readme.

Li, B. (n.d.). *MSolver: A small, self-contained minesweeper solver.* GitHub. Retrieved November 18, 2021, from https://github.com/luckytoilet/MSolver.

Aral, C.; Gelfond, M. (1994). *Logic programming and knowledge representation*. The Journal of Logic Programming. 19–20: 73–148. doi:10.1016/0743-1066(94)90025-6

Minesweeper Online. 2021. Minesweeper Statistic. [online] Available at: https://minesweeper.online/statistics.