VinUniversity

Minesweeper's destroyer v1.0

BackUp team – Vu Nguyen, Nguyen Hoang, Phuc Vu, Trung Pham

VinUniversity College

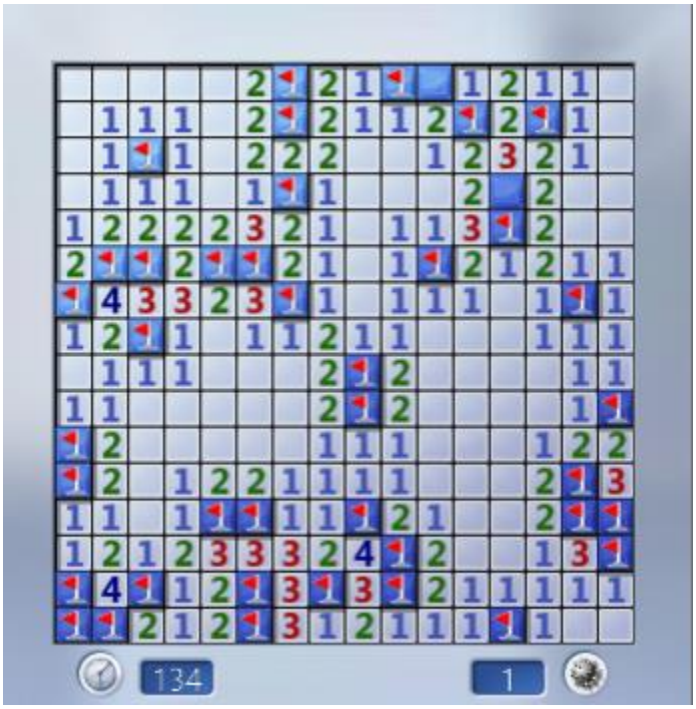
MATH2020 – Discrete Mathematics

Prof. Tu Nguyen

Thursday, November 18th, 2021

Introduction

Minesweeper is a popular game pre-installed in many operating systems. The board is divided into cells, 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 cells that are suspected of being mines. The player wins by opening all the cells without triggering the mines.



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 blocks on the board.
- Probability: Sometimes luck is needed to win the game, we would then use probability to optimize our luck and secure victory

Reading the field

Basically, this step is to teach our bot to recognize the minesweeper grids and different types of squares of the game by applying the following two routines, which are calibration routine and the detection routine.

Beforehand, our team using the library Java AWT (Abstract window toolkit) to implement the Graphical User Interface (GUI) in Java simply because the bot needs to click onto the game's window by itself.

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 taking the screenshot of the board and then analyze it by applying the heuristic algorithm to get the result.

to the initiator is loop-erased.

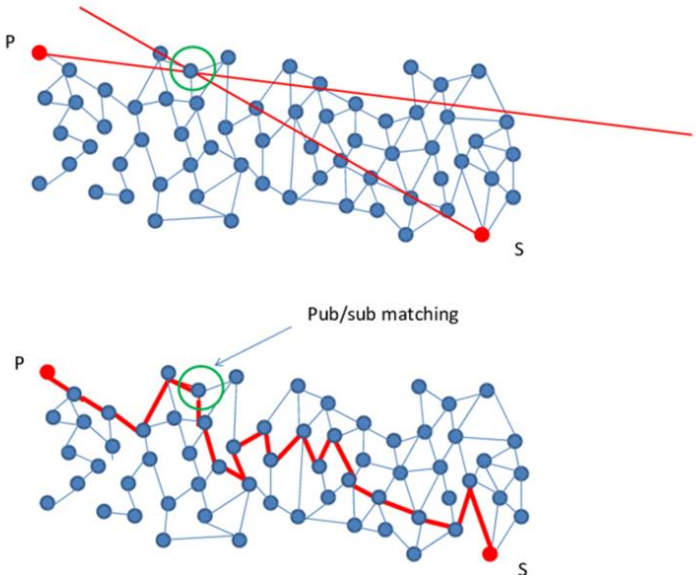
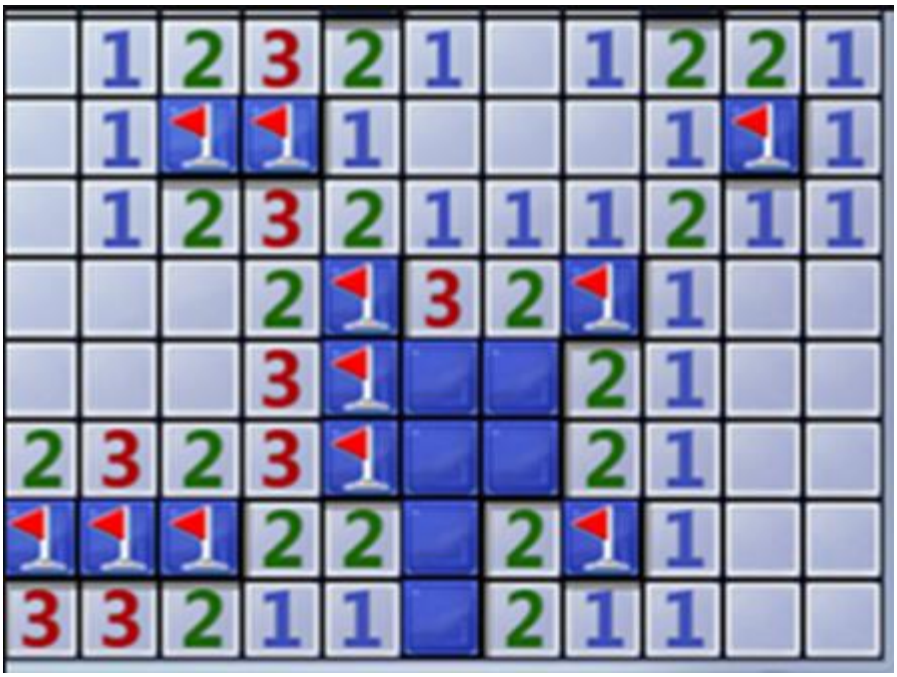


Illustration of heuristic leading to disseminate the information

The reason why the heuristic algorithm is chosen in this specific case is to optimize the speed because it is believed to be much faster and more efficient than most of the traditional methods. After running through all of the aforementioned stages, the bot is now can determine where all the squares are and will be ready for the next routine.

In order to proceed, the bot 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 actually quite simple. Basically, we would try to compare colors between opened and unopened squares.



It is clear that all types of squares would have a different color. However, we've found that some close-to-identical color such as the unopened square and number 1 are quite the similar. But it can be separated from each other by comparing the variance of the patch from the average color for the patch.

Straightforward algorithm

We would first approach the game with a fairly simple brute-force algorithm. Consider an opened block X that has a number on it. The straightforward algorithm is as follow:

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. Next, if the number of unopened blocks is equal to the number on X, then we flag every block surrounding it.

Look at the blocks immediately surrounding X: if the number of flagged blocks equals the number on X, then click on the unopened, unflagged blocks around X. For instance, look at the middle block with number 2 on it:

In this case, there has already been two flags surrounding it, so we can click on all the other surrounding blocks.

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.

Backtracking algorithm

We'd always start with the straightforward algorithm until it gets stuck, then we would bring in the big gun

Consider the following case:

Using the basic algorithm, we seem to be stuck. However, this case is not insolvable, if we think a little out of the box. 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 square with a two (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.

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:

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

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

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

Hence there are 2 possible cases for us to consider:

- There are mines in B and E

- There are mines 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

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 number square to see whether it is a valid proof
- **Step 4:** 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

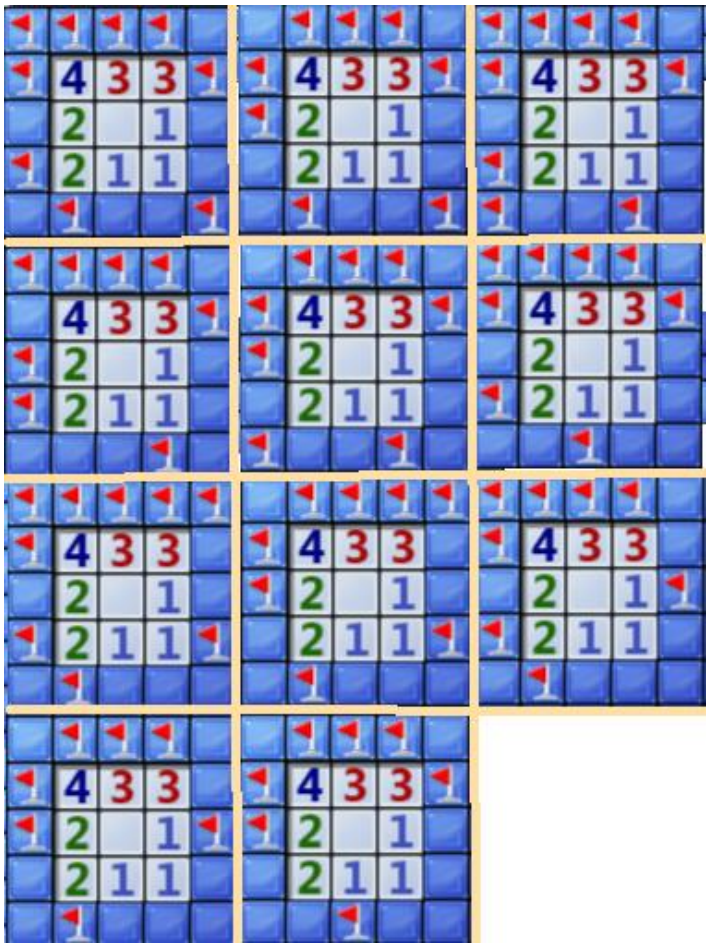
This algorithm has time complexity of 2^n , with n is the number of unknown squares with partial information, which is good enough for us to try. Note that we would update configuration every time, we make a move (finding out more information) so that our root is always the best root possible

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:

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 Tank algorithm generates all possible outcomes for the next move:



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 Mij 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:

$$M_j = \sum Mij$$

Let Xj be discrete random variable such that:

$X_j = 1$, square j-th contains a mine

$X_j = 0$, otherwise

⇒ 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:

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.

Implementation plan

Phase	Name	Description	Start Date	End Date
1	Planning	Brainstorm for ideas Write proposal	October 15th	October 21st
2	Define requirements	Identify main functions Brainstorm optimal algorithms	October 22nd	November 14th
3	Implementation	Write pseudocode Optimize algorithms Implement each function	November 15th	November 28th
4	Testing and modifications	Each member tests the program on their computers and make changes to algorithm and code (if necessary) based on the program's performance	November 28th	December 5th
5	Report	Write report Make presentation	November 15th	December 5th

Reference

Some sources that we researched and refer to when coming up with our implementation:

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.

Mishchenko, A. (2021, March 17). *Creating advanced minesweeper solver using Logic Programming*. DEV Community. Retrieved November 18, 2021, from https://dev.to/krlove/creating-advanced-minesweeper-solver-using-logic-programming-2ppd?fbclid=IwAR1KvNqSVUZ4_7tAuE2OWki-zGZmONXTXPezxKBmPxhjy4kPBS-VakPij2s.

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