Document version 2.0 (mid-project clarifications and revisions)
~~Document version 1.1 (initial release + starter tile description revision)~~

CS-3510 — Spring 2021

Course Project

Designing Efficient Minesweeper Algorithms

Due date: April 22nd, 11:59 PM

0 Clarifications and Revisions

Optimization goal and evaluation metric

Your goal should be to *minimize the number of tiles opened until all bombs are discovered.*

This should also be the evaluation metric for comparing the **performance** of your two algorithms: *how many tiles does the algorithm need to mine before identifying all bomb squares?*

Non-deterministic States

- Your algorithms may need to make a random choice, and so you may hit a bomb square. The game should **not** stop in that case. You should continue until all bomb tiles are identified, trying to minimize the number of mined tiles.
- You should not assume that any 0 tiles will be automatically opened for you. Given that you try to minimize the number of opened tiles, there is no point in mining a tile if you are certain that it is a 0 tile.

Starting Square

The starting square is now guaranteed to be a tile with the highest number on the board. It may not be the only tile with this number.

Trivial Algorithms

Generally, any algorithm that does not make use of the information provided by numbered tile squares is trivial. A couple of examples of trivial algorithms include:

- Mining tiles randomly.
- Not trying to minimize the number of mined tiles.

Differentiating Your Two Algorithms

The two algorithms should be different enough so that they produce clearly different results in terms of running time and/or number of mined tiles (performance).

1 Problem Description

You will work on an algorithm design project for the game Minesweeper. You may work in groups of **up to three students.**

Minesweeper is a puzzle game with a simple set of rules. The single player is presented with an *n*-by-*m* rectangular grid of covered squares that he or she can choose to reveal. Within each square can be either a dangerous *landmine*, a helpful *number*, or *nothing at all*. The player's job is to uncover all safe squares and leave all landmines untouched.

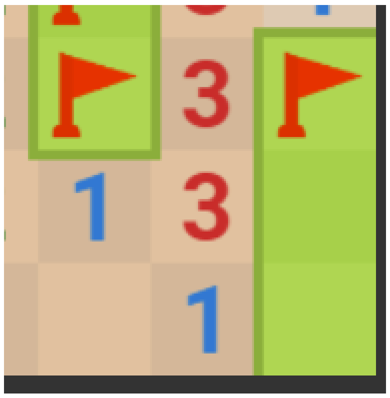


Figure 1.1: Numbers, flags, and undeterminable squares.

A *landmine*, once uncovered, will end the game for the player. In almost all implementations of Minesweeper, the player can set an unvalidated flag on the square to indicate its danger. These flags are for the player to use to mark dangerous squares.

Numbers in a square indicate the number of nearby landmines hidden in the eight neighboring (horizontal, vertical, and diagonal) squares on the grid. Considering groups of nearby numbers can help determine the location of a landmine. It is not always possible to guarantee the location of a landmine; guessing may sometimes be unavoidable.

Nothing at all is equivalent to having '0' in a tile; there are no nearby landmines in the eight surrounding tiles.

Solving a Minesweeper puzzle is a co-NP-complete problem — that is, a solution to a Minesweeper board configuration is the complement of a related problem.

2 Project Requirements

Your group's job is to design two (2) algorithms for solving Minesweeper puzzles. You may consider using all algorithm design techniques learned in class, as well as other algorithmic approaches that we will not have time to cover in class (e.g., randomized algorithms, genetic algorithms, etc). These two algorithms should be different in terms of approach.

For each of the two algorithms, you will provide an **implementation** as well as an analysis and comparison of runtimes and performance in a PDF **project report**. You may look online for guidance, but you should cite your references. Code and analysis should be your own.

2.1 Algorithm Implementation Details

Your algorithms will take as input a file representing Minesweeper board description. As output, the algorithms should return the presumed locations of all landmines on the on board. A board description is a json file in the following format:

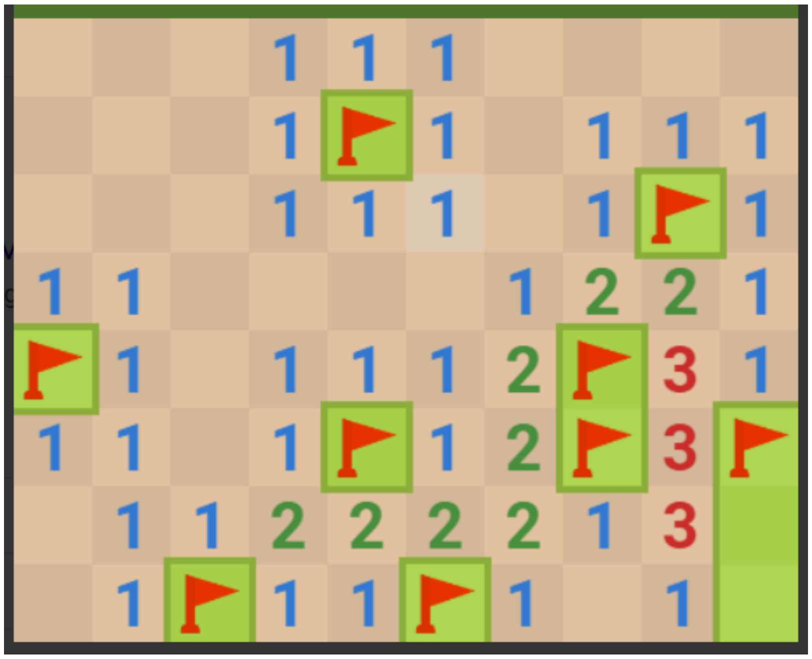
```
{
    dim:      "<int height>,<int width>"
    bombs:    "<int num_bombs>",
    safe:     "<int x>,<int y>",
    board:    "<string<0-9>height * width board_state>"
}
```

- `height` and `width` are integers representing the height and width of the board. Note the origin (0, 0) is in the top left of the board.
- `bombs` is an integer representing the number of bombs on the board.
- `safe` is a pair of integers separated by a semicolon. It represents the x and y coordinate of a zero square. This will allow you to discover an "island" of non-mines that will inform your algorithm's start.
- `board` is a `height * width` length string of values 0 - 9 (the number of adjacent bombs -- 9 represents a bomb in that location). The x, y coordinate board value is represented by the y * width + x position value in the string.

Your two algorithms and other code can be implemented in any language you are comfortable with. Python is recommended. If you choose an "exotic" language (ie not Python, Java, JavaScript, C/C++), you should notify `arnoldwang@gatech.edu` before proceeding to make sure that we will be able to run your program. Programs should be annotated so as to be understandable. Statistics to be mentioned in your report (explained in detail later) should be printed for each problem instance.

2.2 Example Board Configuration

Consider the following board:



A potential JSON representation:

```
{
    "testCase": {
        "dim" : "10,8",
        "bombs": "10",
        "safe": "0,1",
        "board": "000111000000001910111000111010111000012219101112931110191293901122221390191191011"
    }
}
```

2.3 Project Report

Within your project report, you will explain your two algorithms (and show their correctness, if they can provably find the right solution). Your group will also analyze and compare results for the two algorithms. One metric you should assess is the runtime complexity of your algorithm. Another metric is the number of squares you have "probed" in order to deduce the location of all bombs. This metric cannot be lower than the "bomb-density" — the number of bombs over the number of tiles.

You should include plots comparing your two algorithms in terms of: 1) the runtime and 2) the "fraction of squares revealed" metric. These plots should visualize how performance changes as grid size increases and as bomb density increases. (Bomb density being the ratio of the number of bombs over grid size.)

Your algorithms may involve some element of randomness, necessitating multiple trials for each problem instance. In these cases, meaningful averages and percentiles of the performance metrics for your algorithm should be reported as well. **Do not worry about building a GUI.**

A minimal checklist of items to include in your project report:

- ❏ Algorithm A
    - ❏ Description
    - ❏ Justification of correctness
    - ❏ Asymptotic, worst-case runtime analysis
- ❏ Algorithm B
    - ❏ Description
    - ❏ Justification of correctness
    - ❏ Asymptotic, worst-case runtime analysis
- ❏ Experimental plots with two curves (one curve for each algorithm)
    - ❏ Runtime vs grid area (from 100 to 1000 squares in steps of 100)
    - ❏ Runtime vs bomb density (from 2% to 20% in steps of 2%)
    - ❏ Performance ("fraction of squares revealed") vs grid area
    - ❏ Performance ("fraction of squares revealed") vs bomb density

3 Grading

Grading will be based primarily on **completion** of the previous tasks (rather than algorithm performance or runtime). To recap, assignment tasks include two instances of algorithm design and a complete project report.

We reserve the right to give partial credit in cases where one or both algorithms are trivial (e.g., random guessing).

4 Submission Procedure

You will need to submit a single ZIP/tar archive for the entire project. The archive should include code, your project report as a PDF file, and a README plain-text (.txt) file.

The README file must contain:

- Your name (or names for group projects), email addresses, date of submission
- Names and descriptions of all files submitted
- Instructions for compiling and running your programs (if applicable)
- Any known bugs or limitations of your program that we should know about

You should name the submitted file based on the names of the group members. For example, if the team consists of Alpha Beta, Gamma Delta, and Epsilon Zeta, the filename should be: **`AlphaBeta-GammaDelta-EpsilonZeta.zip`**

An example submission may look like as follows:

```
AlphaBeta-GammaDelta-EpsilonZeta.zip
| -- minesweeper-3510/
     | -- minesweeper-3510.py
     | -- <supporting_file.py>
     | -- ...
| -- README.txt
| -- algorithm.pdf
```

Only one member of each group needs to submit the actual code and documentation. The other group members can submit a simple text file in which they mention their partner's name that has submitted the actual assignment and the corresponding filename.

5 Resources

- ➔ You can play Minesweeper at https://minesweeperonline.com/.
- ➔ A couple of papers to fuel your design process:
  - ◆ David Becerra – Algorithmic Approaches to Playing Minesweeper
  - ◆ Jan Cicvárek – Algorithms for Minesweeper Game Grid Generation
  - ◆ Don't forget to cite your sources if you decide to borrow an idea.