Minesweeper Solver Analysis

Jacob Slagle

January 18, 2019

Introduction

Minesweeper Solver is a personal project I started to practice my coding skills. Specifically I wanted to take a problem, develop an algorithm to solve it, and create a project around it with all of the trappings. The game minesweeper was suitably complex: a good,efficient algorithm would take a bit of thinking, but not so much that I couldn't also focus on code design and increasing my Python knowledge. I also like the poetry of taking something which I've used as a distraction in the past and using it to improve myself.

The Problem

Simply put, the problem is to solve a game of Minesweeper (I assume the reader is familiar with the mechanics of the game). In more detail, I seek an algorithm that, if possible, wins a game of Minesweeper without ever risking losing the game, i.e. never makes a move unless its a sure move. It suffices to look at a freeze frame of the game and determine with certainty which spaces must contain mines, and so should be flagged (right-clicked), and which spaces must not contain mines, and so should be revealed (left-clicked). Its unnecssary to find all such spaces, because an algorithm that can find at least one new space each time can be applied repeatedly.

What if there are no spaces that can be determined to be mined or free of mines (henceforth "free" for short)? Indeed, this is the case at the beginning of the game before any spaces have been revealed. However every game of minesweeper I've played, including the one I've coded, ensures that the first square selected is free as well as all surrounding squares. As it turns out, sometimes it's impossible to make any certain moves in the middle of the game either. In this case all we can do is guess, and I will discuss below how to make a good guess. For now though, we consider this: when possible, how do I determine which moves I can make with certainty.

Theory and Algorithms

Here I use the pronoun "we" as in formal mathematics.

As a starting point we consider what exactly it means to beat a game of minesweeper. To win we must place flags on exactly those squares which contain mines and reveal the rest of the squares- in other words we must find the right "placement" of flags on the board. Now suppose we are playing a particular game and furthermore we are looking at a freeze frame of the game where some of the squares are revealed. The revealed squares in this frame contain numbers which tell us how many mines are adjacent to it. Since we aim to place flags on mines and only mines, these number hints are constraints on the way flags can be placed. With an eye towards satisfying these constraints, we make the following definition:

Definition 1. A placement is **satisfactory** if every revealed square s, the number hint h in s is equal to the number of flags surrounding s.

Observe that the correct placement of flags on mines is itself a satisfactory placement. It follows that if a square is flagged in *every* satisfactory placement, it must contain a mine. Therefore we say placing a flag on such a square is a **certain flag** (a flag placed with certainty). Likewise, if a square is *unflagged* in every mine placement, we know it must be free (i.e. it does not contain a mine). Thus we say revealing is a **certain reveal**. A **certain move** is a certain flag or a certain reveal. Consider the value of a certian move over an uncertain move: if a reveal is uncertain, there are satisfactory placements in which the revealed square contains a mine, and if one of those satisfactory placements is the actual mine placement then revealing the square loses the game. Uncertain flags, while not immediately fatal, can lead to errors down the line. Hence we only want to make certain moves. This line of reasoning lends itself to an algorithm idea: generate all satisfactory placements, check which squares are consistently flagged or unflagged across each placement and make certain moves accordingly. Let's assume for the moment we have a function satisfactoryPlacemnts(G) which takes a game of minesweeper G and generates all satisfactory placements.

function SOLVE(G) ▷ Version 1
  $Mines \leftarrow$ game board
  $Free \leftarrow$ game board
  for P in satisfactoryPlacements(G) **do**
    $Mines \leftarrow Mines \cap P$
    $Free \leftarrow Free - P$
  end for
  flagAll($Mines$)
  revealAll($Free$)
end function

We just need to implement satisfactoryPlacements(). We start with a brute force approach: go over all mine placements and yield the ones that are satisfactory. Mine placements are just subsets of the game board B, i.e. elements of the powerset $\mathcal{P}(B)$.

function SATISFACTORYPLACEMENTS(G) ▷ Version 1

```
    for P in 𝒫(B) do
        if P is satisfactory then
            yield P
        end if
    end for
end function
```

This method checks $|\mathcal{P}(B)| = 2^{|B|}$ placements. We can check if a placement is satisfactory in linear time, and since the placements are bound by the board in size, we have that satisfactoryPlacements() has an $O(|B|2^{|B|})$ runtime. The other parts of solve(). The other work in solve is relatively insignificant and so this also the runtime for solve. If you've ever played minesweeper though it might occur to you that there's unnecessary about looking at the entire board, since in a sense we only have information about the unrevealed squares which are adjacent to numbered squares. In sense these are the only squares we can really do anything with. We state this formally with the following claim:

Claim. *If a square is not adjacent to a revealed square, it is not a certain move.*

Proof. Suppose s is a square not adjacent to any revealed square. Choose a satisfactory placement P- for instance P could be the actual mine placement . Set $P_1 = P - \{s\}$ and $P_2 = P \cup \{s\}$. Because s is not adjacent to any revealed squares, P_1 contains the same number of flags around each revealed square as P does and so P_1 is also a satisfactory placement. By the same token, P_2 is a satisfactory placement. Because P_1 is a satisfactory placement that does not contain s, s is not a certain flag. Because P_2 is a satisfactory placement that contains s, s is not a certain reveal. □

We only want to "play" on squares that have the potential to be certian moves, and so with the above claim in mind we say a square is **in-play** if it is unrevealed and adjacent to a revealed square. We will refer to all in-play squares collectively as the **perimiter** because they circumscribe the revealed part of the board.We adapt the above algorithm simply by substituting perimiter wherever game board was before. This version of the algorithm yields the same result, but we will not bother with the details of the proof.

```
function SOLVE(G)                                          ▷ Version 2
    Mines ← perimiter(G)
    Free ← perimter(G)
    for P in satisfactoryPlacements(G) do
        Mines ← Mines ∩ P
        Free ← Free − P
    end for
    flagAll(Mines)
    revealAll(Free)
end function
function SATISFACTORYPLACEMENTS(G)                         ▷ Version 2
    for P in 𝒫(perimiter(G)) do
```

```
        if P is satisfactory then
            yield P
        end if
    end for
end function
```

This reduces the runtime to $O(|Per|2^{|Per|})$ where of course Per is the perimiter. While this is still exponential, the perimiter is *much* smaller than the board except in pathological cases. In practice though (i.e. when using the the algorithms outlined here), the revealed part of the board is a 2-dimensional contiguous blob and so it's perimiter is 1-dimensional. Waving our hands, we might say the perimiter is roughly bounded by, say, four times the square root of the board size. This makes a huge difference given that it acts as an exponent in the runtime