COVIDSafe

Report

January 3, 2023

1 Problem description

1.1 Introduction

COVIDSafe helps quickly identify and contact people who may have been exposed to COVID-19 (called 'close contacts'). The game will not protect you from viruses and will not warn you as soon as you get close to someone infected with a virus. You must try hard social distancing and access information about infected people.

1.2 Idea

COVIDSafe is a classic puzzle game. **The goal** of the game is to clear a rectangle grid without exposing viruses. The viruses are hidden in some of the squares in the grids. When each safe square is opened, a number indicating the number of viruses surrounding will appear and provide information to search for viruses.

1.3 Rules of the game

Find all viruses in the grid by carefully revealing cells. A revealed cell will indicate the number of adjacent viruses (horizontally, vertically, or diagonally). Cells with no adjacent viruses will be blank and automatically reveal their neighbors. Revealing a virus will cause you to lose the game. You win the game by revealing all non-virus cells.

1.4 Requirement

There are three default levels of difficulty: Beginner (9x9 grid with 10 mines), Intermediate (16x16 grid with 40 mines) and Expert (16x30 grid with 99 mines).

Our final product will be tested by Intermediate level game.

2 Solutions

2.1 Random

2.1.1 Introduction

The simplest method to solve this problem is to just randomly reveal cells

2.1.2 Algorithm

while not $finished$ **do**
  $cell \leftarrow randomlyChoose(unrevealedCells)$
  $reveal(cell)$

$finished \leftarrow checkFinished()$
end while

2.1.3 Result

After trying 3000 times, we could not win the game even once, which is unacceptable. Better approaches are needed.

2.2 Simple logic

2.2.1 Introduction

In this strategy, for each state, we will iterate through all clear boxes that contact at least one unrevealed box, try to mark its undecided neighbors by the result of the comparison between the number on the box and the sum of the number of marked viruses and the number of undecided boxes around it.

Concretely, let $knownViruses_{ij}$ be the number of viruses surrounding cell ij, $unrevealed_{ij}$ be the number of unrevealed cells on cell ij's neighborhood and $value_{ij}$ be the number on cell ij.

If:

$$value_{ij} == unrevealed_{ij}$$

then we can be certain that all unrevealed cells surrounding cell ij are containing viruses.

Moreover, if:

$$value_{ij} == knownViruses_{ij}$$

then all unrevealed neighbors must not contain viruses and hence can be safely revealed.

If we cannot mark any cells after iterating through all satisfied cells, a random cell will be revealed.

2.2.2 Algorithms

```
# Initial reveal
randomlyChoose(unrevealedCells)
while not finished do
    for cell in satisfiedCell do
        if value_ij == unrevealed_ij then
            toMark.add(cell) for cell in cell_ij's unrevealed neighborhood
        end if
    end for
    # Two for-loops are needed as we need the full context before
    # being able to mark cells as safe to open
    for cell in satisfiedCell do
        if value_ij == knownViruses_ij then
            toReveal.add(cell) for cell in cell_ij's unrevealed neighbors
        end if
    end for
    while toMark is not empty do
        cell ← toMark.pop()
        mark(cell)
    end while
    while toReveal is not empty do
```

cell ← *toReveal.pop*()
reveal(*cell*)
end while
if no cell was marked or revealed **then**
randomCell ← *randomlyChoose*(*unrevealedCells*)
reveal(*randomCell*)
end if
end while

2.2.3 Result

This simple method gives us about 58.25% winrate, which is not bad at all. However, there's room for improvement.

2.2.4 Problems

First, our algorithm has a lot of randomness, which may reduce its performance.

Moreover, in some situations, there is more than one possible permutation of the position of viruses. In those cases, the current approach fails to mark or reveal cells as there is no certainty that a cell is safe or not.

For example, consider the board below:

0	1	
0	2	
0	2	
0	1	

In the aforementioned example, we can see that the viruses can be either in the 1st and 3rd columns or in the 2nd and 4th columns. The algorithm cannot be sure about any cell so it will randomly open one.

2.3 Fixed initial cell

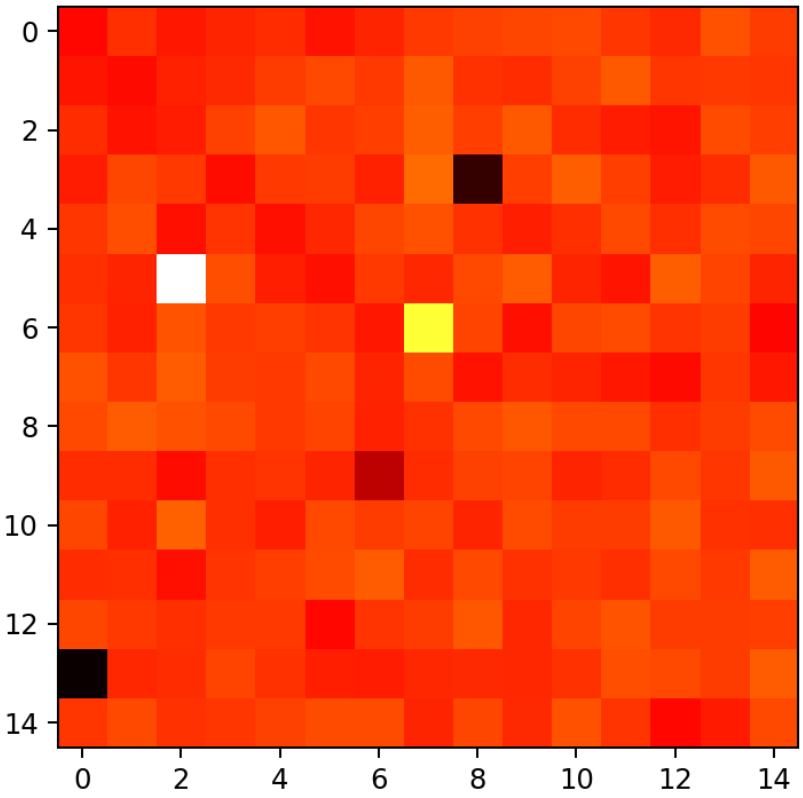
2.3.1 Introduction

This approach is identical to the above method, except instead of randomly choosing a cell to reveal at the beginning, we will use a fixed cell.

2.3.2 Result

We tested all 16x16 cells and choose square (i,j) with i $\in$ [1,16] and j $\in$ [1,16].

The results are shown in the following heatmap:



It can be easily seen that specifying the first cell to reveal doesn't make any difference to our result. In every cases, the winrates are always in range (58, 61).

2.4 CSP

2.4.1 Introduction

All grids in the game board are assigned with a corresponding variable, we set a variable's domain to be 1 if it is flagged, and 0 if it revealed and not a mine. For any unrevealed grid, a *Integervariable* with *lowerbound* = 0 and *upperbound* = 1 (which is equivalent to a *Booleanvariable*) is assigned.

Constraints:

The most important part in our CSP solver, is how much information that we can extract from visible grids in a game board, and translate these information to CSP constraints.

	1	2	3	4	5	6	7	8	9	...	16
1	0	0	0	1	a						
2	1	1	1	1	b						
3	g	f	e	d	c						
4											
5											
6											
7											
8											
9											
...											
16											

As an example, consider the 16x16 board shown above with a total of 40 viruses. We identify each square by its (row,column) coordinates.

In this example, the squares (1,1), (1,2) and (1,3) have already been played and have no virus in their respective surrounding squares. The squares (1,4), (2,1), (2,2) and (2,3) have been played too, and are surrounded by one virus each. At any point in the game, the set of squares that are still covered, not flagged as viruses, and adjacent to an uncovered square are the fringe. The simplest CSP formulation has boolean variables x_{ij}, denoting fringe squares at row i and column j. In our example, there are seven fringe variables: $x_{31}, x_{32}, x_{33}, x_{34}, x_{35}, x_{15}$ and x_{25}.

In the figure we show the number of viruses revealed in each square that has been uncovered. For each uncovered square adjacent to a fringe square, we have the constraint that the number revealed in that square is the sum of the viruses in the adjacent uncovered squares. This yields the following 5 constraints, one for each uncovered position adjacent to a fringe square.

Name	Square	Constraint
C1	(1,4)	$(x_{15}, x_{25}) \in (0,1)$ or $(1,0)$
C2	(2,4)	$(x_{15}, x_{25}, x_{35}, x_{34}, x_{33}) \in (0,0,0,0,1)$ or $(0,0,0,1,0)$ or (0,0,1,0,0) or (0,1,0,0,0) or (1,0,0,0,0)
C3	(2,3)	$(x_{32}, x_{33}), x_{34} \in (0,0,1)$ or $(0,1,0)$ or $(1,0,0)$
C4	(2,2)	$(x_{31}, x_{32}, x_{33}) \in (0,0,1)$ or $(0,1,0)$ or $(1,0,0)$
C5	(2,1)	$(x_{31}, x_{32}) \in (0,1)$ or $(1,0)$

As you can see from the structure of the constraints, they are all m of n constraints. That is, they all state that m out of the n variables are ones, the rest are zeros. You can use this fact in designing your constraint representation and in devising algorithms for checking constraints. The constraint on the total number of viruses is easy to express algebraically as

$$\sum_{ij \in (1...40)} x_{ij} = 40, x_{ij} \in (0,1)$$

where 20 is the total number of viruses in some instance of the problem.

A key step in setting up the CSP for a grid state is to assert all the constraints above. Of course, finding a single solution to the CSP containing the fringe variables is not enough. Before we uncover a square, we want to be certain that it is clear of viruses in all possible solutions to the CSP. Thus, unlike most CSP problems, which look for a single solution and stop, we examine all possible assignments of values to the fringe variables to find all the ones that satisfy the constraints. This set of possible solutions is then examined for squares that are viruses in every solution, or clear in every solution. Such squares can then be flagged and uncovered, respectively. In some situations, there will be no square that is clear in every solution. Then, there is no fringe square that can be safely uncovered. In this case, your program has to guess, by uncovering a square that might be clear.

Solution for problem:

We have constraints from above steps,

C1: a+b = 1
C2: e+d+a+b+c = 1
C3: f+e+d = 1
C4: g+f+e = 1
C5: g+f = 1

In this step, we use Google' OR-Tools to help solve the linear equations and simplify the constraints set. In this example, after applying OR-Tools solver for this set of constraints, we later get: g=0, f=1, e=0, d=0, c=0 and still left with a+b=1, which yells 2 solutions containing (a, b) = (0, 1) and (a, b) = (1, 0).

2.4.2 Algorithm

Uncover the upper-left square

Repeat

Find all fringe variables.

Assert the set of constraints on the fringe variables.

Simplify the set of constraints to get all possible solutions.

Flag known viruses and uncover safe squares.

If no squares can be uncovered, play a guess

Until game is won or lost.

2.4.3 Result

2.5 Logic and CSP

2.5.1 Introduction

Finding all solutions for the linear integer programming problem is very computationally intensive. Hence, in this approach, we combine both logic and CSP approaches. At first, logic will be used. When logic fails, we will use the aforementioned CSP method. If both methods fail, we have no choice but to choose a random cell to reveal.

2.5.2 Algorithm

```
# Initial reveal
randomlyChoose(unrevealedCells)
while not finished do
    # Use logic
    for cell in satisfiedCell do
        if value_ij == unrevealed_ij then
            toMark.add(cell) for cell in cell_ij's unrevealed neighborhood
        end if
    end for
    # Two for-loops are needed as we need the full context before
    # being able to mark cells as safe to open
    for cell in satisfiedCell do
        if value_ij == knownViruses_ij then
            toReveal.add(cell) for cell in cell_ij's unrevealed neighborhood
        end if
    end for
    while toMark is not empty do
        cell ← toMark.pop()
        mark(cell)
    end while
    while toReveal is not empty do
        cell ← toReveal.pop()
        reveal(cell)
    end while
    # Solve as a linear integer programming problem
    if no cell was marked or unrevealed then
        add cells to toMark and toReveal by solveAsCSP(currentState)
        while toMark is not empty do
            cell ← toMark.pop()
            mark(cell)
        end while
        while toReveal is not empty do
```

cell ← *toReveal.pop*()
reveal(*cell*)
end while
if no cell was marked or unrevealed **then**
randomCell ←randomlyChoose(unrevealedCells) reveal(randomCell)

2.5.3 Result

Using this method, we got a win rate of 71.2%, which is reasonably high and much higher than average human win rate (21.9%).

3 Problems and difficulty during the project execution

3.1 Create the game board

3.1.1 Problem

We don't have a lot of experience in making games. Making a game work takes us about a month.

3.1.2 Solution

As we are not the only ones who wants to make this game, we can follow some tutorials online to make our game work. It takes a while to understand the tutorials but after all, we made it.

3.2 Iteration between the game and the solver

3.2.1 Problem

A very simple way to have the solver and the game interact with each other is to have the solver contain the game instance or vice versa. However, even though we can limit the solver to only access the game state, it feels like "cheating" as nonetheless, the whole game instance - which also has the solution and the game mechanic - is contained inside the solver instance. Instead of letting the solver have full control of our game instance, we want to have the solver only use data provided by the game instance and in return give it commands. What each instance does with the provided information should not be accessible by the other one. However, implementing this criterion is not simple.

3.2.2 Solution

We struggled with this problem for about a week until one of our members found a way to do it. He makes the game instance and the solver instance interact via text files. Concretely, we run each instance in a different process, each creates a text file to output some information and receives data from the text file of the other one. Syncing them is also a problem but we used a trick to make it possible (see the code for more detail).

3.3 Programming error

3.3.1 Problem

As every programmer, we made errors when coding. Some are syntax errors, which were pretty easy to fix while others are logic errors, which took a lot of time to debug.

3.3.2 Solution

We stared at the screens, used debug mode, ran the code line by line, breakpoint by breakpoint, checking if every variables, functions, expression are correct.

3.4 Getting all solutions from ortools took too long

3.4.1 Problem

In many state, we had too many solutions, which makes getting them and iterating through them take too much time. For instance, consider the board below:

	1	2	3	4	5	6	7	8	9	...	16
1	x	x	x								
2	x	3	x								
3	x	x	x								
4						y	y	y			
5						y	2	y			
6						y	y	y			
7		z	z	z							
8		z	1	z							
9		z	z	z							
...											
16											

The board above yells 24 variables denoted as x, y, z and 4 constraints: $sum(x) = 3$, $sum(y) = 2$, $sum(z) = 3$ and $sum(x, y, z) < 40$

The 4th constraint will always be satified as if we take the sum of 3 aforementioned constrains, we will get $sum(x, y, z) = 6$

It can be seen that we can assign any 3 out of 8 neighbor x as viruses, assign any 2 out of 8 neighbor y cells as viruses and assign any cell out of 8 neighbor x cells as viruses. Hence, the number of solutions, in this case, will be: 'numSolution' $= C_8^3 C_8^2 C_8^1 = 56 * 28 * 8$, which is more than 10000.

3.4.2 Solutions

To solve the problem, we implement 2 solutions:

1. Set a time limit to ortools Solver, which still took a lot of time to stop.

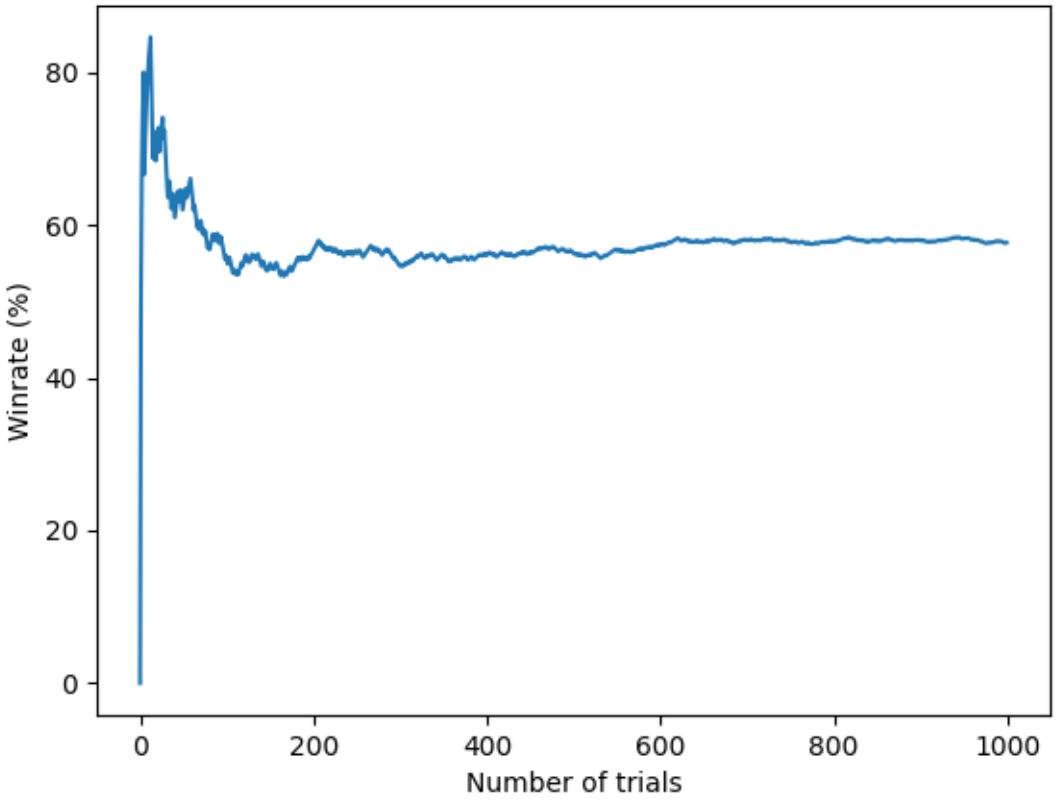
2. Require context to be enough to solve. We found out that if the board has a cell containing 0, it's likely to be solvable.

4 Appendix

4.1 About the validation of our result

For each test, we calculate the win rate (or the percentage that we solve the problem) after running the game 1000 times. Although the win rate is not fully converged, its fluctuation is minor ($\pm 1\%$) and should not make a noticeable difference in the result.

Here is the graph of the win rate with respect to the number of trials in one of our tests:



4.2 Reference and tools used in this project

1. AskPython. 2020. Create Minesweeper using Python From the Basic to Advanced
2. A handout from Rice University