Mohammadmahdi Ghahramanibozandan
2041608

Knowledge Representation and Learning Project
Porf. Luciano Serafini
University of Padova

Introduction

Minesweeper is a game where mines are hidden in a grid of squares. The objective is to discover all the mines by opening the cells. At each iteration you have to open a cell if the cell contains a mine you have lost, otherwise you will discover the number of mines in the surrounding cells, which can be used to decide the next cell to be opened.

The Problem to be Solved

Encode any game state in a set of formulas S and use a SAT to decide for every cell i, j if:

- **i, j is safe i.e., S |= ¬mine ij**
- **i, j is unsafe i.e., S |= mine ij**
- **If you cannot decide if i, j is safe or not.**

And then select the best safe cell and generate the next configuration.

Project Taxonomy

Part1: Logic Implementation

In this part, the variables are defined and the game axioms are implemented using Minisat22 from the solver module of pysat. The result is tested on a real test case.

Part2: Automatic Playing

Since automatic playing requires to access to the game, in this part the game environment is simulated and then we make the playing automatic.

Part3: Scalability Testing

In this part, the whole steps are put in a function and by changing the size of the game, we check the scalability of the game.

Part 1: Logic Implementation

- Define Binary Variables: For each cell, there are 11 possible states to take. Note that in order to encode the game easier, two extra rows and two extra columns are considered whose status are Disable and they are unchangeable during the game. The reason is that it makes the coding for edge cells easier.

$$I_0 => 0 : Zero\ bomb\ around$$
$$I_1 => 1 : One\ bomb\ around$$
$$I_2 => 2 : Two\ bombs\ around$$
$$I_3 => 3 : Three\ bombs\ around$$
$$I_4 => 4 : Four\ bombs\ around$$
$$I_5 => 5 : Five\ bombs\ around$$
$$I_6 => 6 : Six\ bombs\ around$$
$$I_7 => 7 : Seven\ bombs\ around$$
$$I_8 => 8 : Eight\ bombs\ around$$
$$D => 9 : Disable$$
$$B => 10 : Bomb$$

Part1: Logic Implementation

- Number of Variables: We define the game size as the number of rows, or the number of columns, of the game. If the size of the game is n, then we have $(n+2)*(n+2)$ cells in our table. To understand it better, note that we considered extra rows and columns in the previous definition. Now, since each cell can take 11 possible states, overally, the number of variables are $11*(n+2)*(n+2)$. In the first part, in which our aim is just to check the correctness of the implementation, we set $n=8$ and consequently we have 1100 binary variables.

Part1: Logic Implementation

- Axioms:

The first row is disable :

$$\bigwedge_{c=0}^{n+1} in(9,0,c)$$

The first column is disable :

$$\bigwedge_{r=0}^{n+1} in(9,r,0)$$

The last row is disable :

$$\bigwedge_{c=0}^{n+1} in(9,n+1,c)$$

The last column is disable :

$$\bigwedge_{r=0}^{n+1} in(9,r,n+1)$$

Original cells are not disabled :

$$\bigwedge_{r=1}^{n}\bigwedge_{c=1}^{n} \neg in(9,r,c)$$

Part1: Logic Implementation

To make the implantation capable of assigning **“Unknown”** status for those cells which are not yet surely safe or unsafe, we consider this axiom. Actually, we do not force each cell to have one state. However, they cannot take more than one values at the time.

There is atmost one state for each cell :

$$\bigwedge_{r=0}^{n+1} \bigwedge_{c=0}^{n+1} \bigwedge_{s,s' \in S, s \neq s'} \neg(in(s,r,c) \wedge in(s',r,c))$$

Part1: Logic Implementation

If a cell's status is **I_0**, then there must be no bomb in its surrounding (8 adjacent cells).

$$If\ I_0,\ then\ there\ is\ no\ bomb\ in\ surrounding:$$

$$\bigwedge_{r=1}^{n}\bigwedge_{c=1}^{n} in(0,r,c) \rightarrow \neg\left(\bigvee_{r'=r-1}^{r+1}\bigvee_{c'=c-1}^{c+1} in(10,r',c')\right)$$

Part1: Logic Implementation

If I_1, then there is EXACTLY 1 bomb in surrounding : (N^2 includes 8 adjacent cells for each pair of (r, c))

$$\bigwedge_{r=1}^{n}\bigwedge_{c=1}^{n} in(1,r,c) \rightarrow \bigwedge_{I\in N^2,|I|=n^2-1+1}\bigvee_{(r',c')\in I} in(10,r',c') \wedge \bigwedge_{I\in N^2,|I|=1+1}\bigvee_{(r',c')\in I} \neg in(10,r',c')$$

If I_2, then there are EXACTLY 2 bombs in surrounding : (N^2 includes 8 adjacent cells for each pair of (r, c))

$$\bigwedge_{r=1}^{n}\bigwedge_{c=1}^{n} in(2,r,c) \rightarrow \bigwedge_{I\in N^2,|I|=n^2-2+1}\bigvee_{(r',c')\in I} in(10,r',c') \wedge \bigwedge_{I\in N^2,|I|=2+1}\bigvee_{(r',c')\in I} \neg in(10,r',c')$$

Part1: Logic Implementation

If I_3, then there are EXACTLY 3 bombs in surrounding : (N^2 includes 8 adjacent cells for each pair of (r, c))

$$\bigwedge_{r=1}^{n}\bigwedge_{c=1}^{n} in(3,r,c) \rightarrow \bigwedge_{I\in N^2,|I|=n^2-3+1}\ \bigvee_{(r',c')\in I} in(10,r',c') \wedge \bigwedge_{I\in N^2,|I|=3+1}\ \bigvee_{(r',c')\in I} \neg in(10,r',c')$$

If I_4, then there are EXACTLY 4 bombs in surrounding : (N^2 includes 8 adjacent cells for each pair of (r, c))

$$\bigwedge_{r=1}^{n}\bigwedge_{c=1}^{n} in(4,r,c) \rightarrow \bigwedge_{I\in N^2,|I|=n^2-4+1}\ \bigvee_{(r',c')\in I} in(10,r',c') \wedge \bigwedge_{I\in N^2,|I|=4+1}\ \bigvee_{(r',c')\in I} \neg in(10,r',c')$$

Part1: Logic Implementation

If I_5, then there are EXACTLY 5 bombs in surrounding : (N^2 includes 8 adjacent cells for each pair of (r, c))

$$\bigwedge_{r=1}^{n}\bigwedge_{c=1}^{n} in(5,r,c) \rightarrow \bigwedge_{I\in N^2,|I|=n^2-5+1}\bigvee_{(r',c')\in I} in(10,r',c') \wedge \bigwedge_{I\in N^2,|I|=5+1}\bigvee_{(r',c')\in I} \neg in(10,r',c')$$

If I_6, then there are EXACTLY 6 bombs in surrounding : (N^2 includes 8 adjacent cells for each pair of (r, c))

$$\bigwedge_{r=1}^{n}\bigwedge_{c=1}^{n} in(6,r,c) \rightarrow \bigwedge_{I\in N^2,|I|=n^2-6+1}\bigvee_{(r',c')\in I} in(10,r',c') \wedge \bigwedge_{I\in N^2,|I|=6+1}\bigvee_{(r',c')\in I} \neg in(10,r',c')$$

Part1: Logic Implementation

If I_7, then there are EXACTLY 7 bombs in surrounding : (N^2 includes 8 adjacent cells for each pair of (r, c))

$$\bigwedge_{r=1}^{n}\bigwedge_{c=1}^{n} in(7,r,c) \rightarrow \bigwedge_{I\in N^2,|I|=n^2-7+1}\bigvee_{(r',c')\in I} in(10,r',c') \wedge \bigwedge_{I\in N^2,|I|=7+1}\bigvee_{(r',c')\in I} \neg in(10,r',c')$$

If I_8, then there are EXACTLY 8 bombs in surrounding : (N^2 includes 8 adjacent cells for each pair of (r, c))

$$\bigwedge_{r=1}^{n}\bigwedge_{c=1}^{n} in(8,r,c) \rightarrow \bigwedge_{I\in N^2,|I|=n^2-8+1}\bigvee_{(r',c')\in I} in(10,r',c') \wedge \bigwedge_{I\in N^2,|I|=8+1}\bigvee_{(r',c')\in I} \neg in(10,r',c')$$

Part1: Logic Implementation

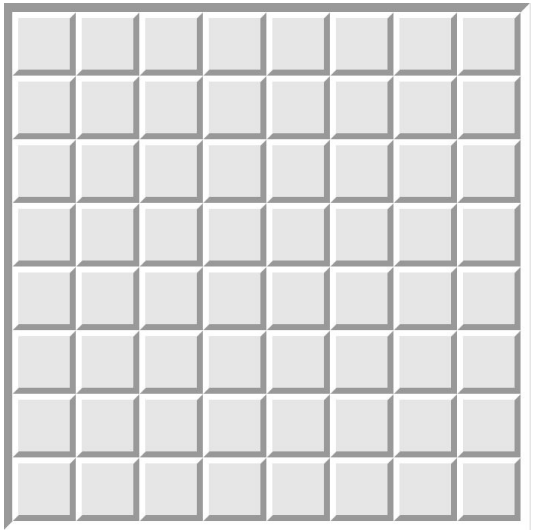
- How to assign **“Safe”**, **“Bomb”** and **“Unknown”** to cells?

  Consider the previously written axioms and the game configuration as a knowledge base(KB). Then for a given cell, i,j:

  - - If **KB ^ Bomb(i,j)** is **UNSAT**, then i,j is **Safe**.
  - - If **KB ^ ¬Bomb(i,j)** is **UNSAT**, then i,j is **Bomb**.
  - - If **None of the above**, then i,j is **Unknown**.

Part1: Logic Implementation

-Test on a real test case:



```
[
 [XX, XX, XX, XX, XX, XX, XX, XX, XX, XX],
 [XX, YY, YY, YY, YY, YY, YY, YY, YY, XX],
 [XX, YY, YY, YY, YY, YY, YY, YY, YY, XX],
 [XX, YY, YY, YY, YY, YY, YY, YY, YY, XX],
 [XX, YY, YY, YY, YY, YY, YY, YY, YY, XX],
 [XX, YY, YY, YY, YY, YY, YY, YY, YY, XX],
 [XX, YY, YY, YY, YY, YY, YY, YY, YY, XX],
 [XX, YY, YY, YY, YY, YY, YY, YY, YY, XX],
 [XX, YY, YY, YY, YY, YY, YY, YY, YY, XX],
 [XX, XX, XX, XX, XX, XX, XX, XX, XX, XX]
 ]
```

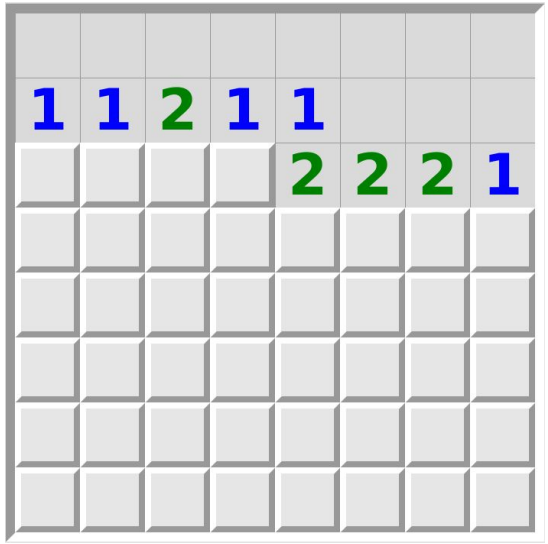
```
Cells Are All Unknown. Let's Make a Random Choice...
```

```
(1, 1):Unknown
(1, 2):Unknown
(1, 3):Unknown
(1, 4):Unknown
(1, 5):Unknown
(1, 6):Unknown
(1, 7):Unknown
(1, 8):Unknown
(2, 1):Unknown
(2, 2):Unknown
(2, 3):Unknown
(2, 4):Unknown
(2, 5):Unknown
(2, 6):Unknown
(2, 7):Unknown
(2, 8):Unknown
(3, 1):Unknown
(3, 2):Unknown
(3, 3):Unknown
(3, 4):Unknown
(3, 5):Unknown
(3, 6):Unknown
(3, 7):Unknown
(3, 8):Unknown
(4, 1):Unknown
(4, 2):Unknown
(4, 3):Unknown
(4, 4):Unknown
(4, 5):Unknown
(4, 6):Unknown
(4, 7):Unknown
(4, 8):Unknown
(5, 1):Unknown
```

```
(5, 2):Unknown
(5, 3):Unknown
(5, 4):Unknown
(5, 5):Unknown
(5, 6):Unknown
(5, 7):Unknown
(5, 8):Unknown
(6, 1):Unknown
(6, 2):Unknown
(6, 3):Unknown
(6, 4):Unknown
(6, 5):Unknown
(6, 6):Unknown
(6, 7):Unknown
(6, 8):Unknown
(7, 1):Unknown
(7, 2):Unknown
(7, 3):Unknown
(7, 4):Unknown
(7, 5):Unknown
(7, 6):Unknown
(7, 7):Unknown
(7, 8):Unknown
(8, 1):Unknown
(8, 2):Unknown
(8, 3):Unknown
(8, 4):Unknown
(8, 5):Unknown
```

Part1: Logic Implementation

Let’s randomly choose the cell (1,1):



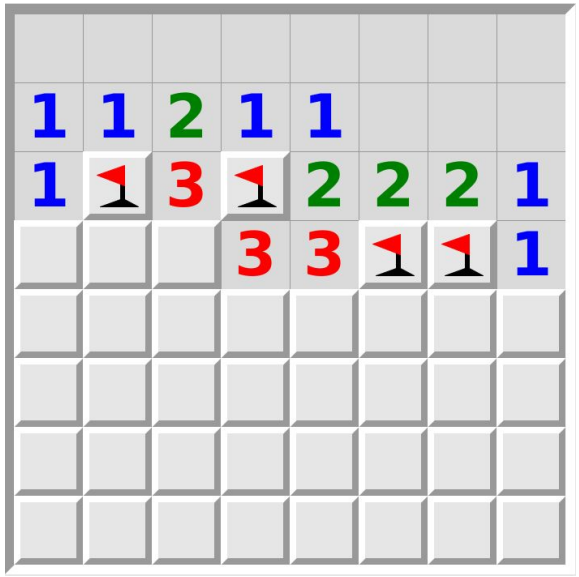

```
[
 [XX, XX, XX, XX, XX, XX, XX, XX, XX, XX],
 [XX, I0, I0, I0, I0, I0, I0, I0, I0, XX],
 [XX, I1, I1, I2, I1, I1, I0, I0, I0, XX],
 [XX, YY, YY, YY, YY, I2, I2, I2, I1, XX],
 [XX, YY, YY, YY, YY, YY, YY, YY, YY, XX],
 [XX, YY, YY, YY, YY, YY, YY, YY, YY, XX],
 [XX, YY, YY, YY, YY, YY, YY, YY, YY, XX],
 [XX, YY, YY, YY, YY, YY, YY, YY, YY, XX],
 [XX, XX, XX, XX, XX, XX, XX, XX, XX, XX]
 ]
```

```
(3, 1):Safe
(3, 2):Bomb
(3, 3):Safe
(3, 4):Bomb
(4, 1):Unknown
(4, 2):Unknown
(4, 3):Unknown
(4, 4):Safe
(4, 5):Safe
(4, 6):Bomb
(4, 7):Bomb
(4, 8):Safe
(5, 1):Unknown
(5, 2):Unknown
(5, 3):Unknown
(5, 4):Unknown
(5, 5):Unknown
(5, 6):Unknown
(5, 7):Unknown
(5, 8):Unknown
(6, 1):Unknown
(6, 2):Unknown
```

```
(6, 2):Unknown
(6, 3):Unknown
(6, 4):Unknown
(6, 5):Unknown
(6, 6):Unknown
(6, 7):Unknown
(6, 8):Unknown
(7, 1):Unknown
(7, 2):Unknown
(7, 3):Unknown
(7, 4):Unknown
(7, 5):Unknown
(7, 6):Unknown
(7, 7):Unknown
(7, 8):Unknown
(8, 1):Unknown
(8, 2):Unknown
(8, 3):Unknown
(8, 4):Unknown
(8, 5):Unknown
(8, 6):Unknown
(8, 7):Unknown
(8, 8):Unknown
```

Part1: Logic Implementation

Flag the Bombs and select the Safe cells:



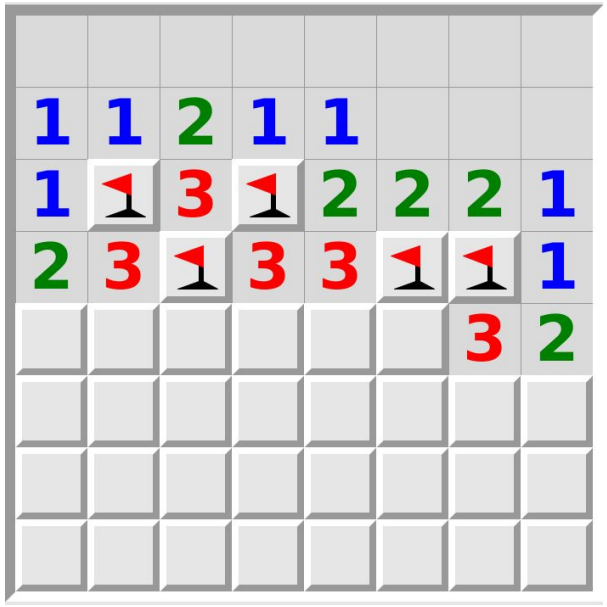

```
[
 [XX, XX, XX, XX, XX, XX, XX, XX, XX, XX],
 [XX, I0, I0, I0, I0, I0, I0, I0, I0, XX],
 [XX, I1, I1, I2, I1, I1, I0, I0, I0, XX],
 [XX, I1, BB, I3, BB, I2, I2, I2, I1, XX],
 [XX, YY, YY, YY, I3, I3, BB, BB, I1, XX],
 [XX, YY, YY, YY, YY, YY, YY, YY, YY, XX],
 [XX, YY, YY, YY, YY, YY, YY, YY, YY, XX],
 [XX, YY, YY, YY, YY, YY, YY, YY, YY, XX],
 [XX, YY, YY, YY, YY, YY, YY, YY, YY, XX],
 [XX, XX, XX, XX, XX, XX, XX, XX, XX, XX]
 ]
```

```
(4, 1):Safe
(4, 2):Safe
(4, 3):Bomb
(5, 1):Unknown
(5, 2):Unknown
(5, 3):Unknown
(5, 4):Unknown
(5, 5):Unknown
(5, 6):Unknown
(5, 7):Safe
(5, 8):Safe
(6, 1):Unknown
(6, 2):Unknown
(6, 3):Unknown
(6, 4):Unknown
(6, 5):Unknown
(6, 6):Unknown
(6, 7):Unknown
(6, 8):Unknown
(7, 1):Unknown
(7, 2):Unknown
(7, 3):Unknown
(7, 4):Unknown
(7, 5):Unknown
(7, 6):Unknown
(7, 7):Unknown
(7, 8):Unknown
```

```
(8, 1):Unknown
(8, 2):Unknown
(8, 3):Unknown
(8, 4):Unknown
(8, 5):Unknown
(8, 6):Unknown
(8, 7):Unknown
(8, 8):Unknown
```

Part1: Logic Implementation

Flag the Bombs and select the Safe cells:

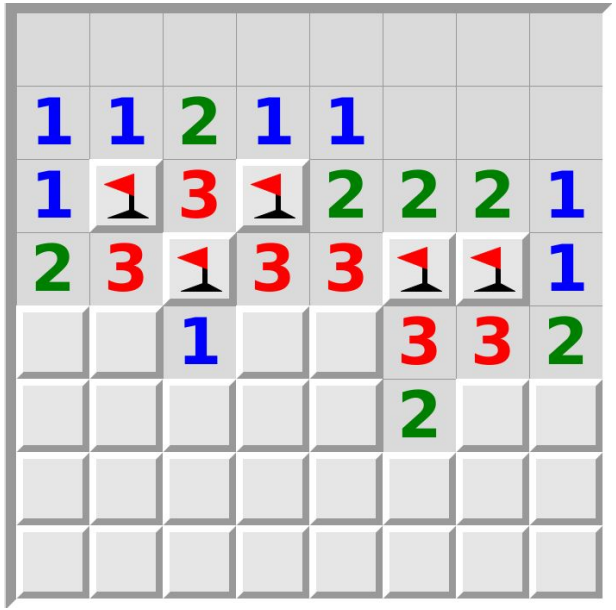



```
[
 [XX, XX, XX, XX, XX, XX, XX, XX, XX, XX],
 [XX, I0, I0, I0, I0, I0, I0, I0, I0, XX],
 [XX, I1, I1, I2, I1, I1, I0, I0, I0, XX],
 [XX, I1, BB, I3, BB, I2, I2, I2, I1, XX],
 [XX, I2, I3, BB, I3, I3, BB, BB, I1, XX],
 [XX, YY, YY, YY, YY, YY, YY, I3, I2, XX],
 [XX, YY, YY, YY, YY, YY, YY, YY, YY, XX],
 [XX, YY, YY, YY, YY, YY, YY, YY, YY, XX],
 [XX, YY, YY, YY, YY, YY, YY, YY, YY, XX],
 [XX, XX, XX, XX, XX, XX, XX, XX, XX, XX]
 ]
```

```
(5, 1):Unknown
(5, 2):Unknown
(5, 3):Safe
(5, 4):Unknown
(5, 5):Unknown
(5, 6):Safe
(6, 1):Unknown
(6, 2):Unknown
(6, 3):Unknown
(6, 4):Unknown
(6, 5):Unknown
(6, 6):Safe
(6, 7):Unknown
(6, 8):Unknown
(7, 1):Unknown
(7, 2):Unknown
(7, 3):Unknown
(7, 4):Unknown
(7, 5):Unknown
(7, 6):Unknown
(7, 7):Unknown
(7, 8):Unknown
(8, 1):Unknown
(8, 2):Unknown
(8, 3):Unknown
(8, 4):Unknown
(8, 5):Unknown
(8, 6):Unknown
(8, 7):Unknown
(8, 8):Unknown
```

Part1: Logic Implementation

Flag the Bombs and select the Safe cells:




```
[
 [XX, XX, XX, XX, XX, XX, XX, XX, XX, XX],
 [XX, I0, I0, I0, I0, I0, I0, I0, I0, XX],
 [XX, I1, I1, I2, I1, I1, I0, I0, I0, XX],
 [XX, I1, BB, I3, BB, I2, I2, I2, I1, XX],
 [XX, I2, I3, BB, I3, I3, BB, BB, I1, XX],
 [XX, YY, YY, I1, YY, YY, I3, I3, I2, XX],
 [XX, YY, YY, YY, YY, YY, I2, YY, YY, XX],
 [XX, YY, YY, YY, YY, YY, YY, YY, YY, XX],
 [XX, YY, YY, YY, YY, YY, YY, YY, YY, XX],
 [XX, XX, XX, XX, XX, XX, XX, XX, XX, XX]
 ]
```

```
(5, 1):Bomb
(5, 2):Safe
(5, 4):Safe
(5, 5):Bomb
(6, 1):Unknown
(6, 2):Safe
(6, 3):Safe
(6, 4):Safe
(6, 5):Safe
(6, 7):Safe
(6, 8):Bomb
(7, 1):Unknown
(7, 2):Unknown
(7, 3):Unknown
(7, 4):Unknown
(7, 5):Unknown
(7, 6):Unknown
(7, 7):Unknown
(7, 8):Unknown
(8, 1):Unknown
(8, 2):Unknown
(8, 3):Unknown
(8, 4):Unknown
(8, 5):Unknown
(8, 6):Unknown
(8, 7):Unknown
(8, 8):Unknown
```

Part1: Logic Implementation

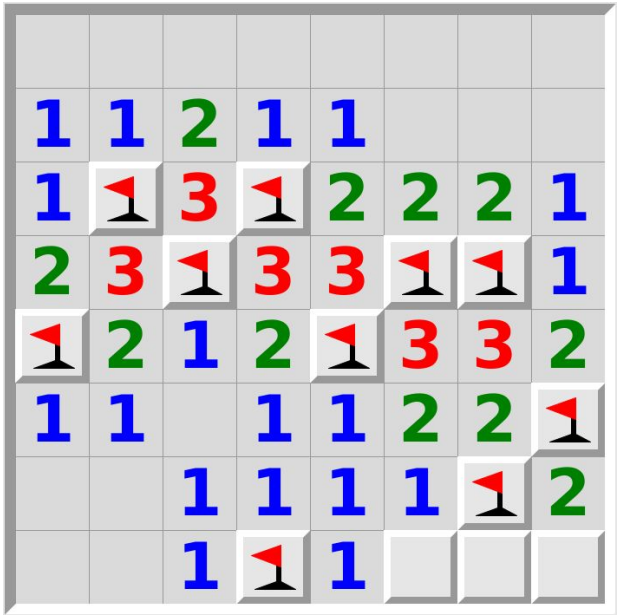
Flag the Bombs and select the Safe cells:

```
[
 [XX, XX, XX, XX, XX, XX, XX, XX, XX, XX],
 [XX, I0, I0, I0, I0, I0, I0, I0, I0, XX],
 [XX, I1, I1, I2, I1, I1, I0, I0, I0, XX],
 [XX, I1, BB, I3, BB, I2, I2, I2, I1, XX],
 [XX, I2, I3, BB, I3, I3, BB, BB, I1, XX],
 [XX, BB, I2, I1, I2, BB, I3, I3, I2, XX],
 [XX, I1, I1, I0, I1, I1, I2, I2, BB, XX],
 [XX, I0, I0, I1, I1, YY, YY, YY, YY, XX],
 [XX, I0, I0, I1, YY, YY, YY, YY, YY, XX],
 [XX, XX, XX, XX, XX, XX, XX, XX, XX, XX]
 ]
```

```
(7, 5):Safe
(7, 6):Safe
(7, 7):Bomb
(7, 8):Safe
(8, 4):Bomb
(8, 5):Safe
(8, 6):Unknown
(8, 7):Unknown
(8, 8):Unknown
```

Part1: Logic Implementation

Flag the Bombs and select the Safe cells:



```
[
 [XX, XX, XX, XX, XX, XX, XX, XX, XX, XX],
 [XX, I0, I0, I0, I0, I0, I0, I0, I0, XX],
 [XX, I1, I1, I2, I1, I1, I0, I0, I0, XX],
 [XX, I1, BB, I3, BB, I2, I2, I2, I1, XX],
 [XX, I2, I3, BB, I3, I3, BB, BB, I1, XX],
 [XX, BB, I2, I1, I2, BB, I3, I3, I2, XX],
 [XX, I1, I1, I0, I1, I1, I2, I2, BB, XX],
 [XX, I0, I0, I1, I1, I1, I1, BB, I2, XX],
 [XX, I0, I0, I1, BB, I1, YY, YY, YY, XX],
 [XX, XX, XX, XX, XX, XX, XX, XX, XX, XX]
 ]
```

```
(8, 6):Safe
(8, 7):Safe
(8, 8):Safe
```

Part1: Logic Implementation

Flag the Bombs and select the Safe cells:

```
[
 [XX, XX, XX, XX, XX, XX, XX, XX, XX, XX],
 [XX, I0, I0, I0, I0, I0, I0, I0, I0, XX],
 [XX, I1, I1, I2, I1, I1, I0, I0, I0, XX],
 [XX, I1, BB, I3, BB, I2, I2, I2, I1, XX],
 [XX, I2, I3, BB, I3, I3, BB, BB, I1, XX],
 [XX, BB, I2, I1, I2, BB, I3, I3, I2, XX],
 [XX, I1, I1, I0, I1, I1, I2, I2, BB, XX],
 [XX, I0, I0, I1, I1, I1, I1, BB, I2, XX],
 [XX, I0, I0, I1, BB, I1, I1, I1, I1, XX],
 [XX, XX, XX, XX, XX, XX, XX, XX, XX, XX]
 ]
```

```
You win...
```

Part1: Logic Implementation

- Results:

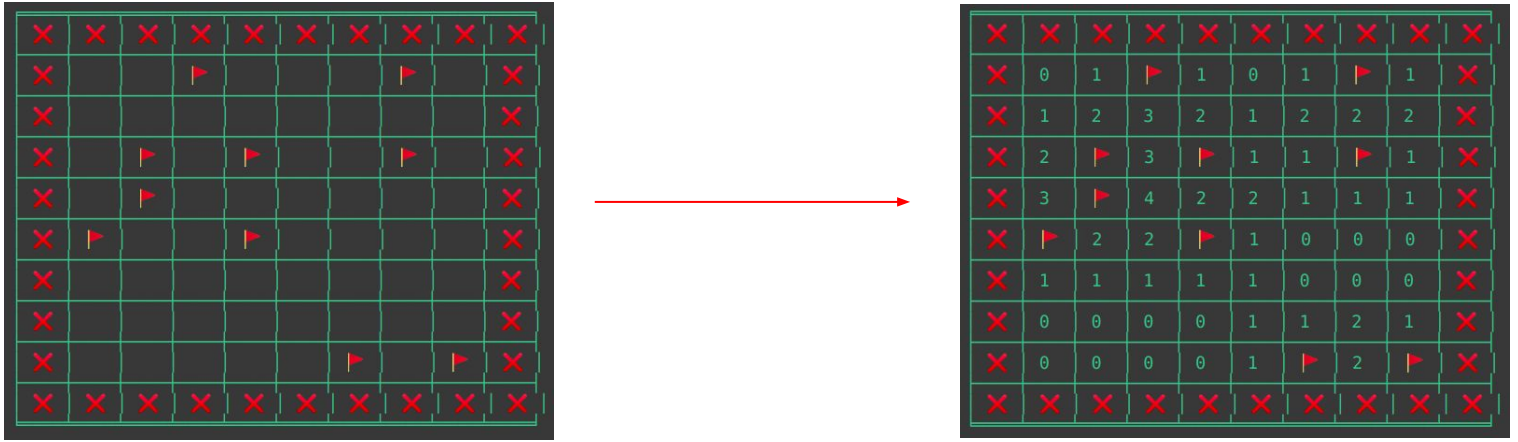
- The implementation is **correct** and it **never** makes mistake.
- We may **lose** the game, since somewhere we need to make a **random** choice.
- Updating the configuration **manually** is time-consuming.

Part2: Automatic Playing

We concluded that updating the game configuration manually is time-consuming and avoid us analysing the implementation. One would suggest to automatically flag bombs and select safe cells. However, it first requires to simulate the game environment. Using applications to play this game, the environment is **semi-observable** for us and we cannot make the game automatic. By game simulation we deal with this issue.

Part2: Automatic Playing

- Game Environment Simulation: We randomly distribute the specific number of bombs in the cells and then, accordingly assigne the real state to each cell. Then, we claim that this environment is our game and our implementation should reconstruct it from an empty configuration, using axioms and the current configuration of the game.



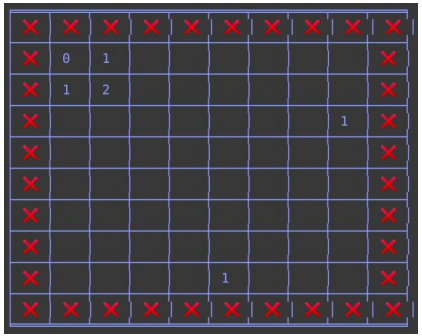
Part2: Automatic Playing

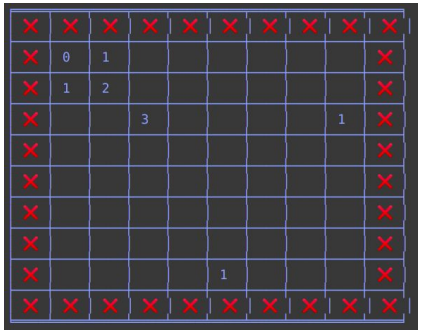
- Generate the Next Configuration: By flagging bombs and reveal the status of safe cells, we generate the next configuration of the game.

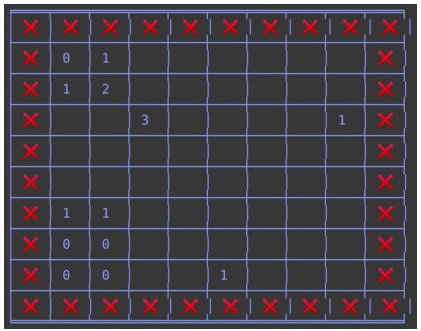
```
def generate_new_configuration(initial_matrix, status):
  safes = [key for key in status if status[key]=='Safe']
  bombs = [key for key in status if status[key]=='Bomb']
  for sf_idx in safes:
    initial_matrix[sf_idx[0]][sf_idx[1]] = game[sf_idx[0]][sf_idx[1]]
    if game[sf_idx[0]][sf_idx[1]] == BB:
      print(Fore.RED + "you lose...")
      return (False,initial_matrix)
  for bmb_idx in bombs:
    initial_matrix[bmb_idx[0]][bmb_idx[1]] = BB
  return (True,initial_matrix)
```

Part2: Automatic Playing

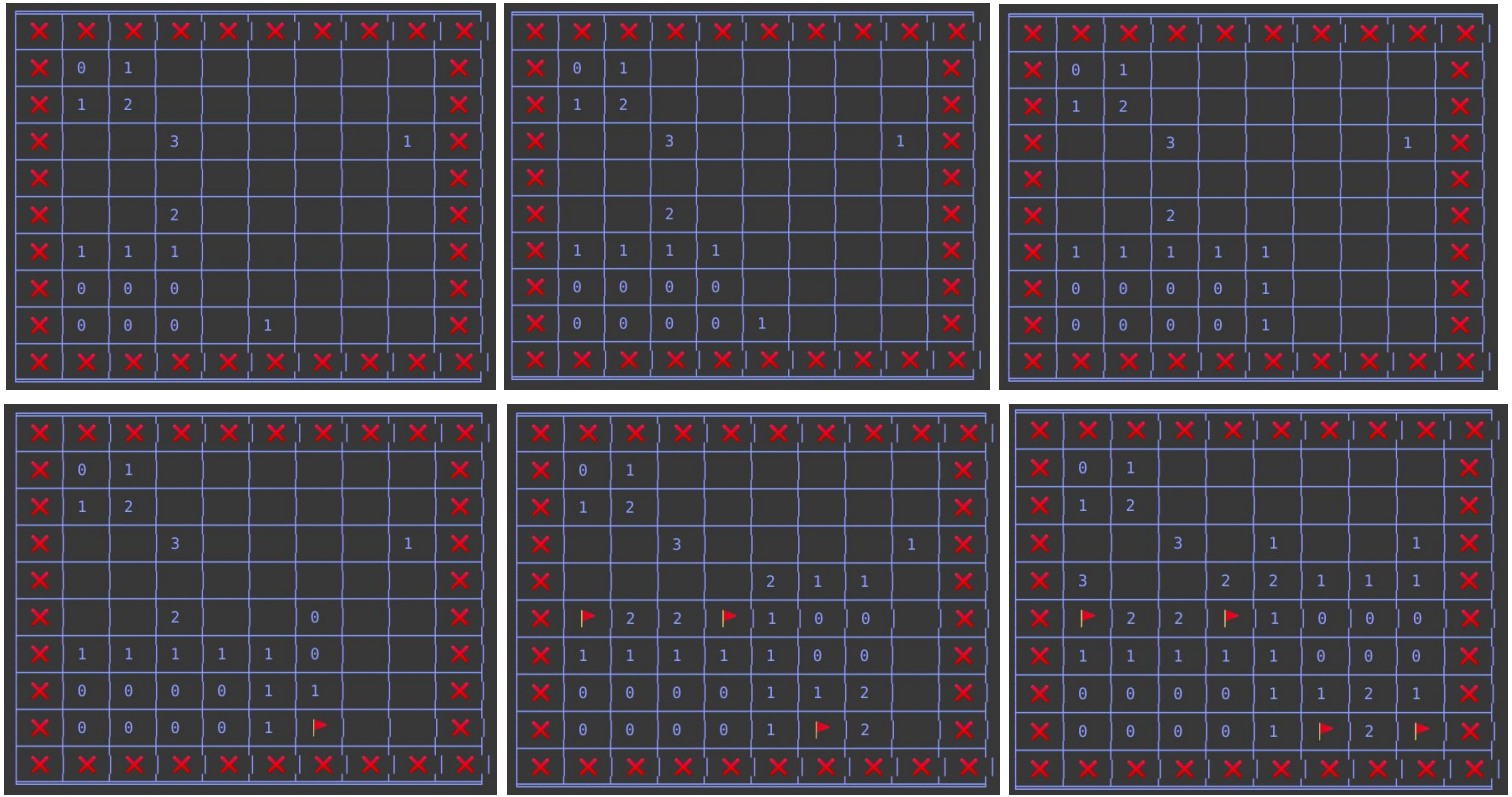
```
[
 [XX, XX, XX, XX, XX, XX, XX, XX, XX, XX],
 [XX, YY, YY, YY, YY, YY, YY, YY, YY, XX],
 [XX, YY, YY, YY, YY, YY, YY, YY, YY, XX],
 [XX, YY, YY, YY, YY, YY, YY, YY, YY, XX],
 [XX, YY, YY, YY, YY, YY, YY, YY, YY, XX],
 [XX, YY, YY, YY, YY, YY, YY, YY, YY, XX],
 [XX, YY, YY, YY, YY, YY, YY, YY, YY, XX],
 [XX, YY, YY, YY, YY, YY, YY, YY, YY, XX],
 [XX, YY, YY, YY, YY, YY, YY, YY, YY, XX],
 [XX, XX, XX, XX, XX, XX, XX, XX, XX, XX]
]
```



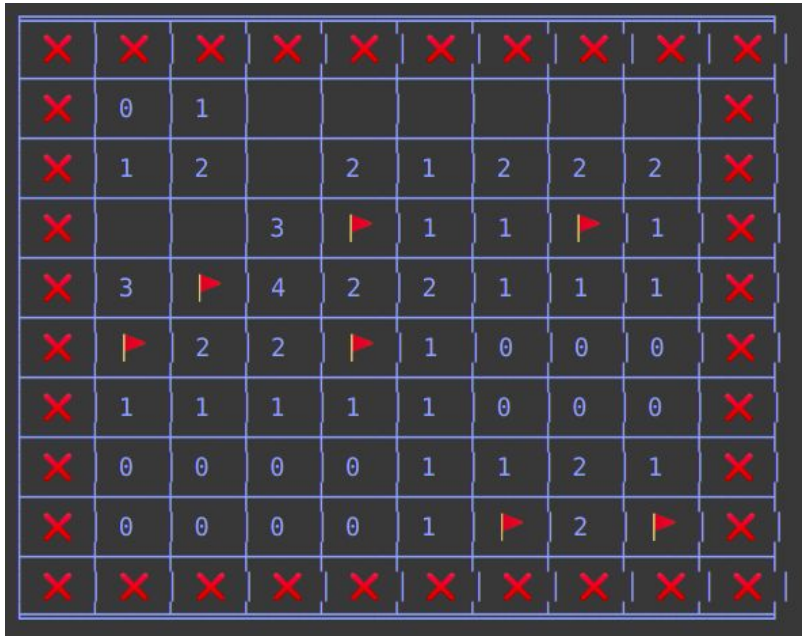


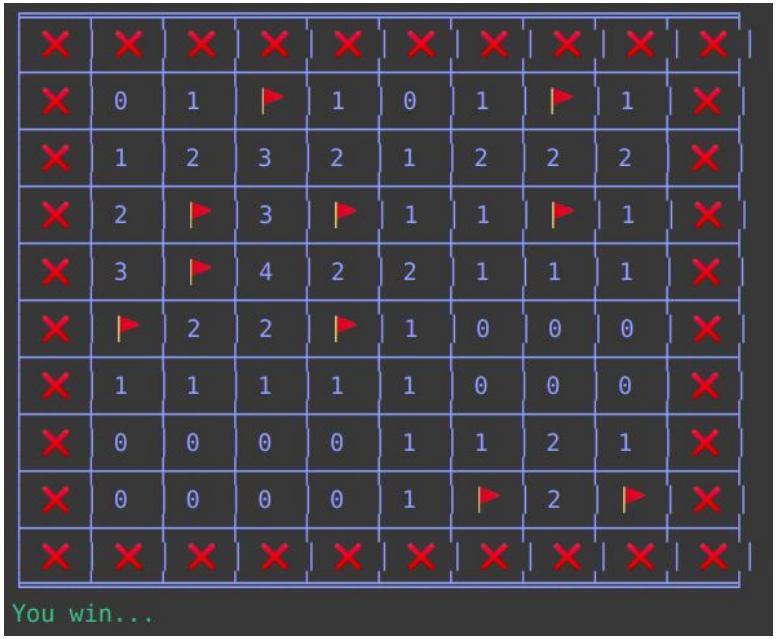


Part2: Automatic Playing



Part2: Automatic Playing



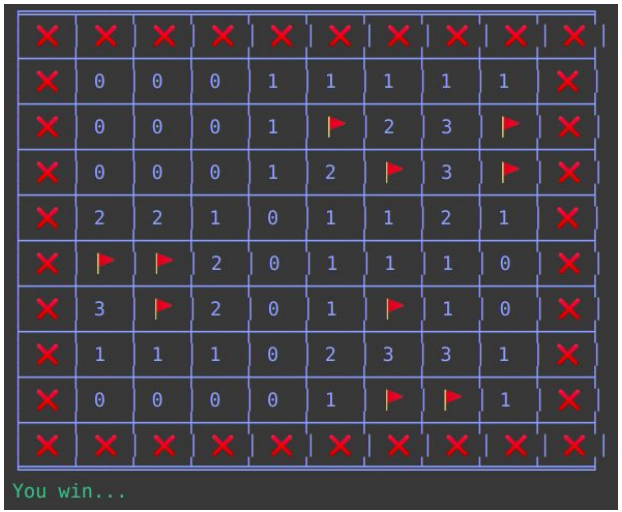



```
1 initial_matrix == game
```

```
True
```

Part2: Automatic Playing

We then test the automatic playing on a real game. Below, you can see the one taken from an application and its solution obtained by our implementation. We skip showing the step-by-step changing and report only the final state obtained by our implementation:



Part3: Scalability Testing

In this part we put everything into a function and attempt to call it on nine games which are different in size to test the scalability of our implementation. The size of these games are *4x4, 5x5, 6x6, 7x7, 8x8, 9x9, 10x10, 15x15, 20x20.*

In the next slide, we will see the results for *15x15* and *20x20* games.

Game : 4 X 4, 3 *bombs*

Game : 5 X 5, 5 *bombs*

Game : 6 X 6, 7 *bombs*

Game : 7 X 7, 8 *bombs*

Game : 8 X 8, 9 *bombs*

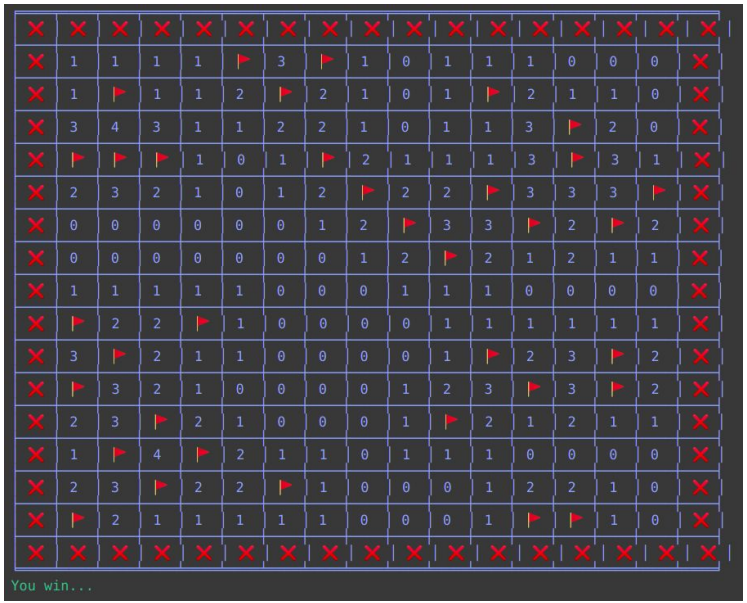
Game : 9 X 9, 10 *bombs*

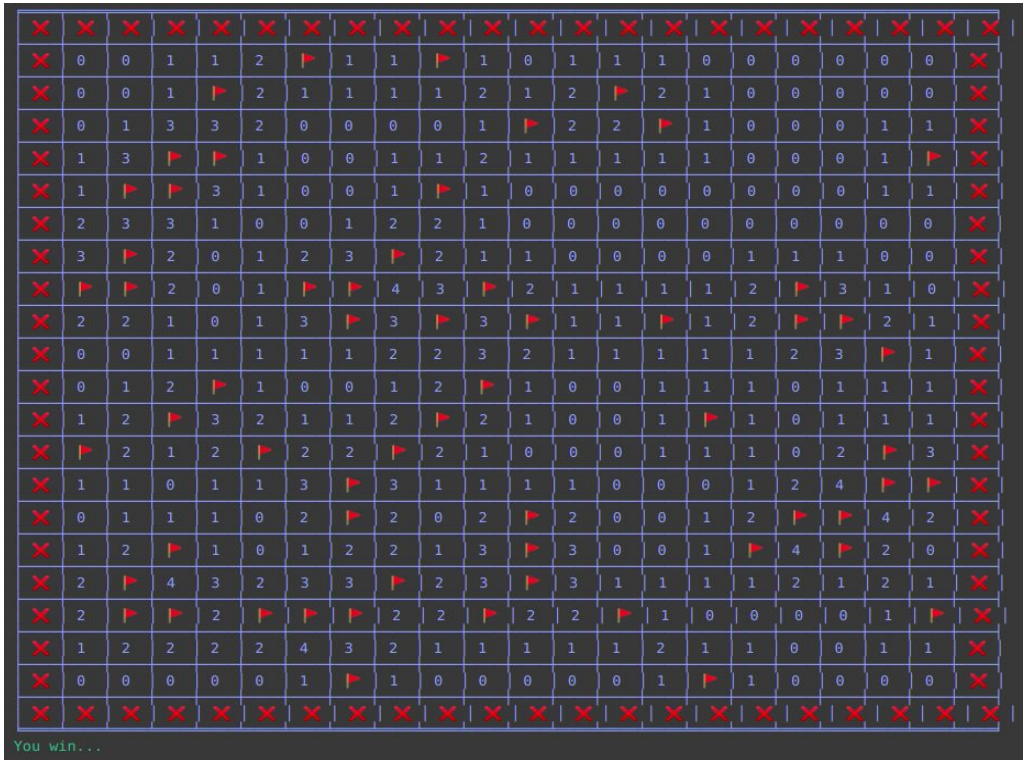
Game : 10 X 10, 13 *bombs*

Game : 15 X 15, 35 *bombs*

Game : 20 X 20, 60 *bombs*

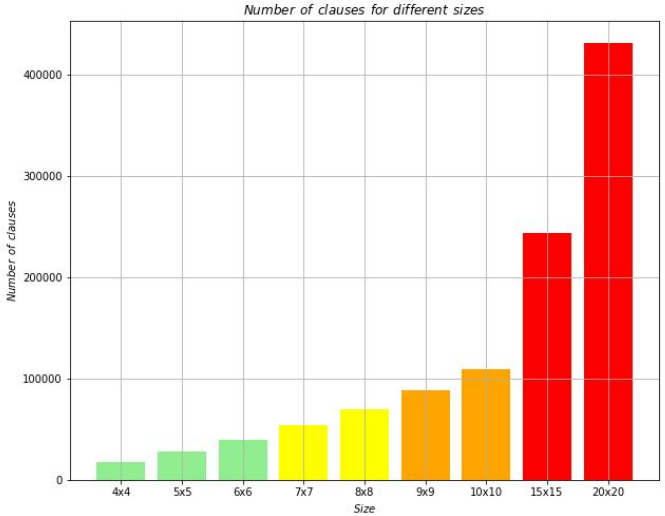
Part3: Scalability Testing

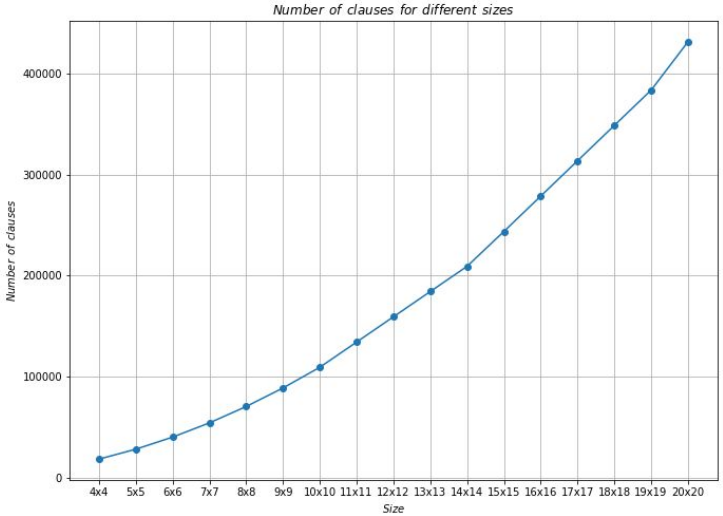




Part3: Scalability Testing

Here we plot the change in the number of clauses with respect to the game size:





Part3: Scalability Testing

According to the results, we can approximate the curve for the change in the number of clauses versus the game size:

$$NC \approx 1100 * size^2$$

It means that for a *100x100* game, we will have approximately *11 Million* clauses. This amount of clauses would be handled easily by Pysat package and we can expand our game to such a dimension. The reason this project avoids testing the model on such a huge game environment is that Colab Notebook cannot visualize the result properly and there is no purpose of doing this.

Thank for your attention