CS311 Report
Assignment 0 - Simulation Using Java

B Siddharth Prabhu
200010003@iitdh.ac.in

Devdatt N
200010012@iitdh.ac.in

9 August 2022

1 Introduction

Here we attempt to calculate the approximate time an infiltrator would take to cross a border into a defending country. The border, embedded with probability governed sensors pose a challenge to the infiltrator. Therefore, we must take an algorithmic approach to the problem by varying the probability and width parameters.

2 Assumptions & Considerations

Different elements of the scenario have been modeled as classes, included in the same folder as our main program, `Main.java` . Below are the assumptions and considerations we've taken into account during the coding process:

2.1 Border

The border between the AC and DC consists of a rectangular (discretized) grid of sensor cells, which has infinite length, and width W. We consider the initial position of an infiltrator to be one of the cells in the first layer of the border. (Considering it to be outside the border at $t = 0$ would just translate the generated graph, and wouldn't change the trends.)

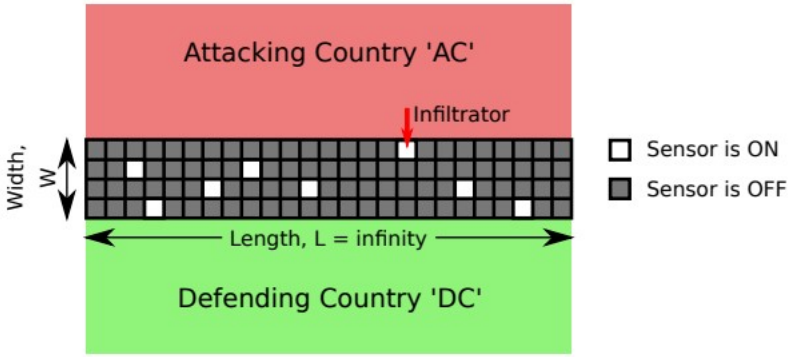

Figure 1: Illustration of the scenario

2.2 Sensors and Infiltrator

The infiltrator is initially placed on a sensor at depth zero. (The terms 'width' and 'depth' are used interchangeably here.) Each sensor is ON or OFF based on a probability 'p' ($0 < p < 1$). Things to consider include:

1. Moving left or right has no payoff, since L is infinity and the sensor states are all based on a probability value that's independent of the decisions of other sensors.
2. Moving upwards would be counteractive to crossing the border. It would be better to stop and wait at the current sensor until a favourable scenario exists (such that the current sensor is off AND one of the three adjacent sensors of the next level is also off.

An initial approach would be to render the states of only the 8 'visible' sensors, that are reachable from the current cell. This is already a significant reduction from having to render the entire $L \times W$ grid. However, we can do better. Since we only ever move forward, we can just simulate the current cell, 'W', and the three 'forward' cells, which we call 'A', 'S', and 'D', due to their QWERTY intuitiveness. At the final layer, only 1 cell ('W') has to be generated. Also, the infiltrator itself keeps track of the extent to which it has crossed the border, as a class attribute.

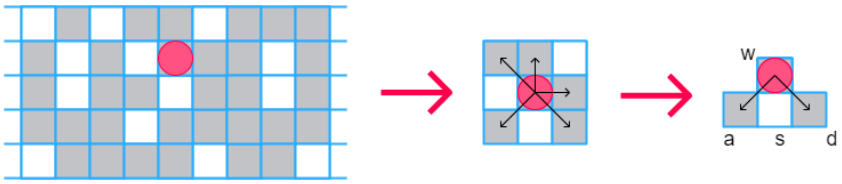

Figure 2: Evolution of our approach

2.3 Environment Variables

- `p_arr` and `width_arr` contain different values of 'p' and 'W' respectively, that we loop over to obtain different values for infiltration time 't'.
- `num_trials` is responsible for the number of trials done for each (p, W) pair. However (in its current form), the Java and Python programs don't communicate with each other. So, the .py program is set up for `num_trials = 5`
- The `Clock` class handles time-tracking for each trial of the experiment.
- The averages over the trials are calculated in the Python file, and are then processed to plot the required graph.

3 Code Compilation and Execution

To get started, we check if java is available using `java -version` from the command line.

We use the following command line arguments to run and test our program :

1. `javac *.java`
2. `java Main`
3. `python3 graphplot.py`

As an alternative, we have also included a bash script which executes all the arguments which can be run with the command line argument `bash run.sh` .

In our program's case, separate inputs for widths and p don't have to be entered since all combinations are stored within arrays within. Output is configured to be saved in the `output.txt` file where each line is in the format (p,w,t) where 'p', 'W' are parameters, and 't' is time taken.

We have used WSL(Windows Subsystem for Linux) to emulate a Linux environment, so to open the generated graph from the WSL command line we use `wslview graph.png` .

4 Graphical Representation

We have generated a three dimensional graph with p, width and time on the x,y and z axes respectively using matplotlib. The obtained graph is as below:

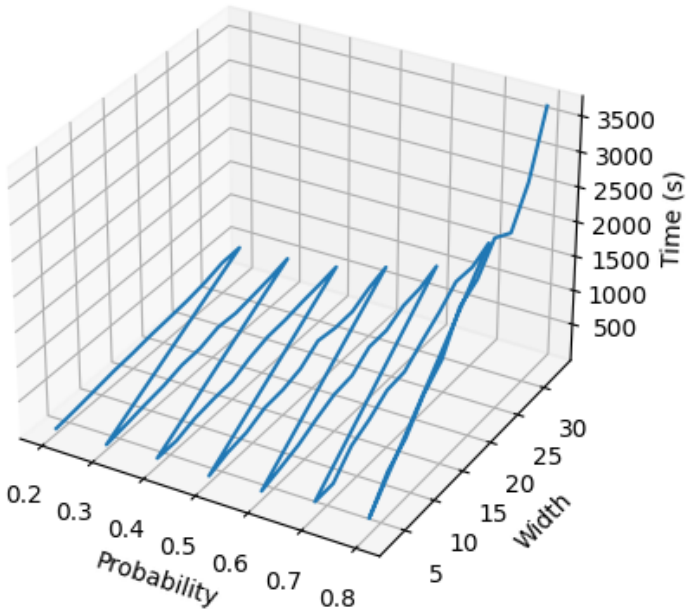

Figure 3: Variation of time with p,w combinations

5 Observations

As we can see, average time to cross the border increases as probability 'p' of the sensor switching on and width increases. For a uniform 'p', as width increases the rate of change of time taken to cross the border remains largely the same.

However, at high widths and changing 'p', the differential of time taken with respect to width can be seen as more exponential. This is to be expected since it is almost impossible to cross the border as 'p' approaches 1.

6 Conclusion

Apart from a rather strange downward slant around `width = 30` and `p = 0.8`, the graph remains as expected with time to cross the border increasing along with (p,w) increasing. This might be because true randomness is hard to achieve and even though we have tested each combination five times, we need a higher number of trials to have an effect of choosing a truly random number.

Nevertheless, we have managed to deliver a rather optimal algorithm to generate the time taken by ignoring the unfruitful left and right moves.