User Guide

Overview

diseaseSim.py's purpose is to simulate the outbreak of a disease based on a number of user defined variables. The variables that can be defined are:

- *Neighbourhood type* – this will either be 'Moore' or 'Von Neumann' cellular automata neighbourhoods. This will determine the 'rules' of the simulation and what the population is allowed to interact with and how they will behave each timestep.
- *Grid type* – this will instruct the program to either [1. Run using a default area (see *columns* and *rows*)] or [2. Search the local directory for a 'barriers.csv' file that can be used to manually determine the 'grid space' and rows and columns (see 'barriers.csv' definition)].
- *Starting population* – which must be an integer greater than or equal to 0. This variable will populate the simulation and assign 'people' to a random location in the *grid*. It is determined by the user entering a value and is then recalculated when the user specifies *starting infected* and *starting immune*.
- *Starting infected* – this must be another integer greater than or equal to 0. It will determine how many of the *Starting population* gets converted into an 'infected person'.
- *Starting immune* – this must be another integer greater than or equal to 0. This variable will determine how many of the *Starting population* gets converted into an 'immune person'.
- *Timesteps* – this must be another integer greater than or equal to 0. It determines the number of calculations each simulation should perform according to the neighbourhood type. Each timestep is equivalent to one calculation of the type.
- *Probability of infection* – this must be a float less than 1 and greater than 0. It determines the chance of each 'person' infecting another 'person' if they are standing next to one another (on the same 'grid space').
- *Probability of recovery* – this must be another float less than 1 and greater than 0. It determines the chances of an 'infected person' recovering the next *timestep.*
- *Probability of death* – this must be another float less than 1 and greater than 0. It determines the chances of an 'infected person' dying. In other words, this will decrease the 'total population'. (Note: please see the bugs at the bottom of the page).
- *Columns* – this is an integer greater than or equal to 0. It is the quantity which defines the horizontal axis of the 'grid space'.
- *Rows* – this is an integer greater than or equal to 0. It is the quantity which defines the vertical axis of the 'grid space'.
- *Printing timesteps* – this is a char input (either 'y' or 'n') and will determine whether the 'grid space' will be displayed on the screen.

Definitions

Barriers: This refers to a special cell in the 'grid space' that prevents 'people' from entering into and moving through.

barriers.csv: This refers to a specified file that exists in the program's local directory. It contains 1s and 0s arranged so as to determine the 'grid space'. The 1s define the cell location for 'Barriers' whereas the 0s define the location of vacant cells that the 'People' can move into and through.

Grid space: This refers to the arrays of cells (calculated by the number of [*columns* - 1] multiplied by the number of [*rows* – 1]) that store either 'Barriers' or 'People' when the simulation runs.

People: This simply refers to the sum of the *starting population* + *infected population* + *immune population (*which are recalculated each timestep) as a collective group.

Default Operating Instructions

1) Run the program using python 3 in a terminal window.
2) Input the specified parameters according to the instructions on screen when prompted.
3) Upon completion of the parameter inputs, you will be given the option to run the simulation, quit the program, or change the parameters if you wish.
4) If you do not wish to change the inputs or quit, simply start the simulation.
5) The simulation will begin to run and after it is finished it will prompt the user if they would like a copy of the results.
6) A 'summary_of_results.txt' file will be written to the program's local directory containing information about the simulation as well as the parameters that were utilised.
7) The program will stop running.

Parameter Sweep Operation Instructions

By running the program using the 'params.sh' shell script the user is able to bypass the console inputs and perform multiple simulations in quick succession. The script takes a total of 14 console inputs which are described (in italics) below and will use 'barriers.csv' as the specified 'grid space'.

1. Neighbourhood type (*M* or *V*)
2. Starting population (*integer >= 0*)
3. Starting infected population (*integer >= 0*)
4. Starting immune population (*integer >= 0*)
5. Number of timesteps (*integer >= 0*)
6. Probability of infection [starting] (*float*)
7. Probability of infection [stopping] (*float*)
8. Probability of infection [stepping] (*float*)
9. Probability of recovery [starting] (*float*)
10. Probability of recovery [stopping] (*float*)
11. Probability of recovery [stepping] (*float*)
12. Probability of death [starting] (*float*)
13. Probability of death [stopping] (*float*)
14. Probability of death [stepping] (*float*)

1) So, (as a recommended example for the parameters), you could type this into the console:

```
sh params.sh M 100 5 1 10 0.1 0.5 0.1 0.1 0.5 0.1 0.1 0.5 0.1
```

2) By executing the above command the shell script will start the program and run 125 separate experiments all in quick succession.
2a) The script will create a new directory 'Simulation_Parameter_Sweep_*Date_Time*' and copy all relevant files into it.
2b) It will then create another directory inside 'trial_stats' where it will move the respective experiment's 'diseaseSim_Trial_*Number*.txt' output file in to once it has been created. Each individual experiment will be logged when completed in the terminal window and will output this individual statistics file into the 'trial_stats' directory.
2c) As the program continues to run, it will update a 'summary_of_results.txt' file back in the 'Simulation_Parameter_Sweep_*Date_Time*' directory. This serves as a single txt file that contains all of the experiment's data in 'trial_stats'.
2d) Finally as the program continues to run and eventually ends, it will update a 'greater_statistics.txt' file in the 'Simulation_Parameter_Sweep_*Date_Time*' directory. This 'greater_statistics.txt' file calculates and writes collective statistics using 'summary_of_results.txt' as an input for it's calculations. This file continuously updates in case the program stops or crashes.
3) The shell command will finish and the program will stop generating files and data.

Code Summary

Please note that this is just a brief summary of some of the code.

The program can be broken down into three main parts. The first part of the program defines all of the functions that are used. The second part is the code that executes if the program is not using a parameter sweep, contrary to the third section which executes if there is a script being utilised.

The first part of the code consists of all of the functions. The biggest three are all functions called when writing statistics files at the end of the program. These are the 'summary()', 'output()', and 'statsprint()' functions. Although these three don't necessarily have to be defined as functions (since they could have been easily implemented into the main body of the code) I felt that in order to keep the code somewhat structured and organised it would have be better to have them as defined functions in order to improve its readability. [Note: This section ends at '#Start of Program Code']

The second part of the code relies on system inputs (via the terminal) in order to execute.
The print() and input() functions are implemented to communicate with the user so that he or she can define the variables that are required to run the program. Although I have not written in input exceptions for all possible inputs, (please see known bugs), I have chosen to do the main ones (such as if the user inputs a character or float instead of an integer and vice versa).
One of the biggest parts of this section of the code is the '#confirmation screen selection' which determines what input parameter is updated based on what the user selects to change. Despite this part increasing the usability and accessibility of the program, I am rather disappointed with it since it looks very messy and I do not think it is the most efficient way to accomplish the task. Furthermore, it is not necessarily needed for the simulation to run properly and is more of a novelty feature rather than a core one.
Despite this however, I am pleased with how the rest of the 'non-parameter sweep section' of the code works. This includes how the core code builds the game world (e.g. reading the csv file) as well as the 'end screen options' for generating an output file.

The third portion of the code uses the parameter sweep inputs as a basis for the variables. I found this part of the code rather difficult to implement since it involved the use of a .sh file. Although I could have chosen to implement another (start, stop, step) looping procedures for the number of people in the simulation and different timesteps, I chose to only implement loops for the probability variables to reduce the runtime of the parameter sweep.
Furthermore, it would be relatively simple to run a parameter sweep and compare the difference between (for example) a population of 100 against a population of 500. All you would need to do is run the program with different population parameters and compare their 'greater_statistics.txt' files.
The read and write system I chose to implement in the program is not at all efficient but it does create an intuative directory structure and a method for outputting the files. Since I am updating the 'greater_statistics.txt' file each time, if the program crashed I would still have access to the results (up until that point). Similarly, the 'summary_of_results.txt' file also updates continuously as the program runs. This ensures data security and (although it would be much faster and efficient to calculate the averages at the end of the program), it makes sense to calculate it on the go in case someone wanted to stop the program once they had achieved their desired statistical outcomes.

Known bugs:

One bug in the program (that is fixable) is how the program defines the uninfected population:

'total uninfected' = *starting population – starting infected – starting immune.*

If *starting population* is given a value of 0, or if *starting infected* or *starting immune* are greater than *starting population,* due to negative values, this has the potential to break the simulation.

Another bug (that is also fixable) is how the program interacts and takes in the values of the 'barriers.csv' file. If the 'barriers.csv' file does not contain a valid 'grid space' the program will still try to use 'barriers.csv' which can potentially ruin the simulation.

Another bug that is observable is if the user does not input the exact parameters he or she wants as well as the correct parameter types (as described in the instructions above) into their parameter sweep, the script will fail to run with the program.

A final bug (that is fixable) is if the user does not properly define a variable in the parameter sweep. For example, if the user gives the parameter sweep a negative integer, the program will still attempt to run and (if it can) not produce an accurate simulation. If the value will not be taken by the diseaseSim.py, the parameter sweep will simply fail.