CS1210 Computer Science I: Foundations
Project 2: Discrete Event Simulation, Part II
Due Wednesday, December 9 at 11:59PM
Updated Edition

Introduction

In this project we will be extending the DES system you implemented in the first project. You should feel free to start with your own code, or simply use the first project solution as your code base; either way is fine. Realize, however, that project 1 has not yet been graded, so starting with your own code may entail going back and making some corrections later on in the process.

We'll introduce the updates to the project 1 solution one idea at a time: my advice is to implement the first few ideas within the current single disease framework, then extend the simulator to multiple diseases, and finally address the required extensions to *Simulation*.*plot*() and the as-yet-to-be described rewrite of the *Sumulation*() class constructor.

Quarantine

One of the traditional public health interventions used to slow the spread of disease is simply for people to stay home. The word "quarantine" comes from the Italian, when forty days ("quaranta") of isolation were required before incoming ships' crew and passengers could come ashore in Venice during the Black Plague.

The recommended quarantine period of Q days should be a property of the disease, since each disease is contagious for a different number of days, I. Because an agent would need to develop symptoms (after E days) prior to knowing they were sick, some diseases are more amenable to quarantine, as they have relatively small E and larger I. For other diseases (those with larger E and smaller I), quarantine won't have much of an effect on reducing the effect on the population as a whole.

We'll add a method, *quarantine*(*self*, Q), to the *Disease*() class to start imposing a quarantine of $Q \leq I$ days for all those newly infected with the disease. Of course, not every agent is equally likely to be compliant with a quarantine order — healthcare workers are famous for working through their illness, although such misplaced heroics can actually make things worse for their weakened and susceptible patients. So the *Agent*() class should also be modified to reflect an individual agent's quarantine compliance probability, q. You'll need to modify the agent's update() method to ensure the quarantine goes into effect at the appropriate point during the course of an agent's illness, as long as an agent elects to comply according to their compliance probability q. If an agent honors the quarantine, they cannot infect anyone else until Q days later.

Compartmentalized Models

In the first project, every agent is created equal in all respects, even though they can presumably be customized with different quarantine compliance probabilities q or susceptibility parameter s. In more realistic models, agents are created of varying types, where it is the type of the agent that determines these values. Assume each agent, then, has a type variable (an integer, starting with 0) that identifies the type of that instance of class *Agent*(). We can use different types of agents to represent, *e.g.*, the elderly, the hospitalized, middle-aged working agents, stay-at-home parents, young adults, children, and/or infants. Aside from varying the values of these internal parameters, the agent types will interact with each other in different ways (*e.g.*, small children may be less likely to interact with older adults than with the middle aged). So in addition to individualized q and s parameters for agents of different types, we need to establish an additional contact probability parameter (default 1.0) that should be factored into the *Agent*.*infect*() method's calculations.

Because each type of agent should be capable of having a different contact probability with every other type of agent, we'll define an agent's contact probabilities, *cp*, as a list of probabilities, where each *cp*[*i*] modifies the probability that *self* will come into contact with an agent of type *i*. Thus what once relied only on *s*, *v* and *t* now must also factor in an agent's *cp*[*i*] when the susceptible agent is of type *i*. To make this work, you'll also need to modify *Simulation*(). *populate*() to allow one to specify the type of the agents being added and a corresponding vector of contact probabilities for that type of agent.

Vaccination Campaigns

In project 1, we had provided a mechanism for vaccinating an agent, but had not really made any use of this mechanism. For project 2, we modify the *Simulation*() class to support vaccination campaigns. A vaccination campaign is characterized by the probability that a given agent will be vaccinated, and a probability that the vaccine, if given, will be effective (this is modeled by the existing *v* parameter of an agent).

The *Simulation.campaign(self, time, disease, coverage, v)* method adds a tuple containing at least $(time, disease, coverage, v)$ to a new internal variable $Simulation.events$. When $Simulation.run()$ hits time step *time*, all agents in the Simulation that are not infected with *disease* are vaccinated with probability *coverage* and resulting vaccine effectiveness *v*. Note that multiple campaigns may take place in a single simulation.

We can handle quarantine orders to use the same kind of timed event idea. Define $Simulation.quarantine(self, time, disease, Q)$ to add a tuple containing at least $(time, disease, Q)$ to the $Simulation.events$ variable. In this way, we can model public health autorities' response to a pandemic situation.

Multiple Diseases

Finally, up to this point we have restricted ourselves to a single disease. In project 2, we need to lift this restriction to allow for more than one disease to be circulating at once. More specifically, we will modify the $Simulation.seed()$ method class to operate more like the $Simulation.campaign()$ method. By adding an additional parameter, *time*, we can inject new diseases into the simulation. Note that supporting multiple diseases will require significant updates to the code, since you will need to keep track of the course of each disease separately.

Extensions to Plot()

In the first project, you were asked to plot two lines showing the number of agents in state E and those in state I. In addition to I and E, modify plot() to also show the number of susceptible agents S (note that a susceptible agent who is vaccinated with effectiveness *v* should only contribute $v < 1$ to the total number of agents in state S; thus the effects of a vaccination campaign should be immediately obvious in the plot). Of course, now that there are multiple diseases, you will need to either label the lines accordingly, or create multiple plots, one per disease.

Interactive Simulation

One of the fun parts of running a simulation is the ability to explore "what if" scenarios, where the impact of different conditions can be simulated to better understand the implications of public health decisions. Should I close the schools? Should I order a quarantine? Should I order a vaccination campaign? Who should I vaccinate first? Those of you who grew up playing SimCity or the Sims can surely appreciate the usefulness of such a tool.

At this point, you should pretty much have all the components you need to deliver just this kind of simulation experience. To do so, you will need to write a top level function (not a method of any existing class) called *simulate*() that that supports interactive exploration of our disease simulator. The operation of this function is probably best introduced by example:

```
>>> simulate()
sim> new 500 0.001 [[1.0,0.5,0.5],[0.5,1.0,0.5],[0.5,0.5,1.0]]
sim> add 100 0
sim> add 100 1
sim> add 200 2
sim> disease influenza 0.95 2 7 0
sim> disease mumps 0.99 17 10 0.99
sim> seed 0 influenza 3
sim> seed 24 mumps 10
sim> quarantine 40 mumps 10
sim> campaign 25 influenza 0.9 0.85
sim> run
sim> plot influenza
sim> plot mumps
sim> bye
>>>
```

where the special *sim>* prompt indicates that whatever you type is interpreted in the context of our simulation.

Let's break down what these commands mean.

(1) The *new* command creates a new Simulation() object describing a $D = 500$ step simulation. The additional arguments on the same line are arguments used to create the instance of Simulation() *(note that you may well need to adjust some of the parameters to your existing __init__() method to make this simulation loop work appropriately).* Here, the mixing parameter is 0.001 and the 3x3 contact probability matrix (which suggests 3 different groups of agents) shows that agents are twice as likely to interact with other agents in their own group than agents from the other two groups.[1]

(2) The next three *add* commands add 100, 100 and 200 agents of type 0, 1 and 2, respectively *(note that you may well need to adjust some of the parameters to your existing __init__() method to make this simulation loop work appropriately).*

(3) The next two *disease* commands create new instances of the Disease() class that describe influenza and the mumps, respectively. The remaining arguments specify values for t, E, I and r, respectively *(note that you may well need to adjust some of the parameters to your existing __init__() method to make this simulation loop work appropriately).*

(4) The two *seed* commands ensure that 3 agents (selected at random) are initially infected with influenza (on day 0) and 10 agents (selected at random) are infected with the mumps on day 24 of the simulation *(note that you may well need to adjust some of the parameters to your existing __init__() method to make this simulation loop work appropriately).*

---

[1] Hint: note that there are no extraneous spaces in the list that specifies the contact matrix. Also, you will find the Python *eval*() function useful for handling that last list argument.

(5) The next *quarantine* command states that a 10 day quarantine for mumps will be established starting on day 40 of the simulation *(note that you may well need to adjust some of the parameters to your existing __init__() method to make this simulation loop work appropriately)*.

(6) The next *campaign* command states that 90% of currently uninfected patients will be vaccinated starting on day 25 with an influenza vaccine that is 85% effective *(note that you may well need to adjust some of the parameters to your existing __init__() method to make this simulation loop work appropriately)*.

(7) The *run* command runs the simulation to quiesence.

(8) The next two *plot* commands produce plots for their specified diseases.

(9) The *bye* command exits the interactive simulation, returning the most current (*i.e.*, the most recently created) simulation object.

Configuration File

While it is useful to interact with the simulator, it is also useful to be able to queue up a bunch of simulation commands in a file and simply read in the commands directly from the file. Assuming you have a Simulation() object, a config(self, filename) method that opens filename and reads in simulation commands like the ones shown above can be a useful shortcut. Note that your config method should handle all of the interactive commands except *new*, as being a method of Simulation() implies you already have a simulation object available.