C4M Workshops: Human Mobility and Epidemic Modelling

Introduction and Overview

Researchers use models of epidemics in order to be able to predict whether and how an epidemic can be contained. Using models of epidemics to arrive at conclusions with public health implications has a long history. As far back as the 18-th century, the Swiss mathematician and physicist Daniel Bernoulli modelled the spread of smallpox[2].

Modelling an epidemic requires modelling the disease – specifying to what extent it is infectious and how fast individuals can recover; and modelling human mobility – specifying how far and how fast people can travel. Once a model is built and validated, it can be used to predict outbreaks of various diseases using computer simulations.

For this project, you will be implementing an algorithm that illustrates the spread of a disease through different cities. Given a list of cities and the times to travel between them, your program will first calculate the shortest time from the initially infected city to every other city. Then you will simulate the spread of an epidemic through these cities.

In Part 1 of the project, you will be working on modelling human mobility using graphs. Although we will be using graphs to represent travel distance, graphs are useful in a variety of settings, such as bioinformatics and genomics. Part 1 will be completed as Exercise Set 4 (due Jul. 8, to be submitted on PCRS) and Exercise Set 5 (due Jul. 15, to be submitted on PCRS and MarkUs).

In Part 2 you will be modelling the spread of disease using a simple model. This will be completed as Exercise Set 6 (due Jul. 22, to be submitted on PCRS) and Assignment 2 (due Jul. 29, to be submitted on MarkUs). As before, the assignment is expected to be quite a bit more work than the exercises.

Note that initial inputs for Part 2 are available in `e5_tester.py`*, so that you can work on Part 1 and Part 2 from the very beginning.*

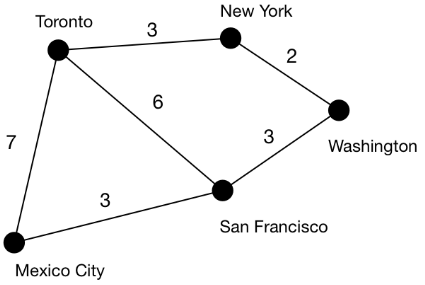
You should do the work assigned on PCRS on your own, but you are encouraged to complete the code submitted on MarkUs with a partner.

Part 1: Finding the Shortest Time to Get to a City

For Part 1, you will implement five helper functions, and submit them on PCRS. Then you will put them together to implement the full algorithm. Before starting to implement the helper functions, finish reading the full description of Part 1.

Your goal for Part 1 is to calculate the *shortest path* between two cities. Given the durations of the direct flights between different cities, you will figure out the shortest possible time (assuming zero layover time) that it could take to travel between two cities that are not connected by a direct flight. For example, you may have the following data about direct flights which can be represented by this picture.

```
Toronto:New York 3
New York:Washington 2
Washington:San Francisco 5
San Francisco:Mexico City 3
Toronto:Mexico City 7
Toronto:San Francisco 6
```



The data means, for example, that there is a 3-hour direct flight from Toronto to New York, **and** a 3-hour direct flight from New York to Toronto. We say that Toronto and New York are neighbours. On the other hand, there is no direct flight from Toronto to Washington (or vice-versa), so they are not neighbours.

For our program, we will store this direct-flight distance information in a dictionary where the keys will be the departure cities and the values will be lists of tuples. Each tuple will have an arrival city and the time of the direct flight. The dictionary that matches our example would look like this:

```
{'Toronto': [('Mexico City', 7), ('New York', 3), ('San Francisco', 6)],
'New York': [('Toronto', 3), ('Washington', 5)],
'Washington': [('New York', 2), ('San Francisco', 5)],
'San Francisco': [('Mexico City', 3), ('Toronto', 6), ('Washington', 5)],
'Mexico City': [('Toronto', 7), ('San Francisco', 3) ]}
```

Notice that all 12 flights (the 6 original flights in the data and the 6 return flights) are stored in the dictionary.

Dijkstra's Shortest Path Algorithm

It is possible to figure out the shortest path from Toronto to every other city as follows:

1. Add the the departure city (e.g., Toronto) to a list of "unvisited" cities.
2. While there are still unvisited cities left in the list:
   (a) Remove the unvisited city that is closest to the departure city,
   (b) Add that city to a list of "visited" cities,
   (c) Add the neighbours of that city (i.e., other cities that one can get to directly) to the "unvisited" list.

There are two things to consider when implementing Step (2)

- When you're picking the "closest" city, keep in mind that there could be more than one route to get to that city. You'll want to make sure that every city in the unvisited list has a distance associated with it, and that you update that distance every time you add a new city to the "visited" list.
- When you're "visiting" a city, don't do step (c) for any neighbours that you've seen already. If it's in the "visited" list, you've already found the shortest route to that city. If it's in the "unvisited" list, just check to see if this path is shorter than the one you've got listed so far (see the previous point).

A good thing to do now would be to trace through this algorithm on a piece of paper (using a pen and writing down the visited and unvisited lists) using the example data. Then check your result by looking at the diagram. There is also a video explaining the algorithm posted on the C4M website.

C4M Workshops: Human Mobility and Epidemic Modelling

Start Coding with Some Helper Functions

Now that you have a basic understanding of the overall algorithm, it is time to implement some helper functions that you will need. In the Exercise Set 4, you will find the following functions that you must implement.

Name	Basic Description
`get_closest(unvisited)`	`unvisited` is a list of tuples of city name and distance. Return the tuple from this list for which the distance is smallest.
`find_city(city, city_list)`	Return the index in `city_list` of the first tuple that contains `city`.
`get_all_cities(distances)`	Return a sorted list of all the cities that appears in the distance dictionary `distances`.
`process_line(line)`	Return a tuple containing the information from a string of the format `"first city name:second city name distance"`.
`build_direct_distances(lines)`	From a list of lines (each of which are the format required as input to `process_line`), return a dictionary that contains all the direct-flight distances between pairs of cities.

This is only a basic description of each function. More details are given in the docstrings and the problem descriptions on PCRS. Because the error messages in PCRS can be obscure and frustrating, you will find it better to initially write your code in Pyzo. You may want to skip ahead to the next section and download the starter code so you can open it in Pyzo. If you do this, don't forget to copy your work back into PCRS to check the test cases (and to submit it for Exercise 4 credit.)

This is the end of the work to submit for Exercise Set 4.

Combining your completed functions into a Program

Once you have correctly completed the five functions, you are ready to combine them into a full program to calculate the shortest time from a single (presumably infected) city to every other city. The first thing you should do is go back and revisit the section of this handout that describes Dijkstra's algorithm. That's what you'll be implementing. If you don't thoroughly understand the algorithm, watch the video or come to see us in office hours. You are welcome (even encouraged) to work in pairs for E5.

Next, download the file `dijkstra_starter.py`, open it in Pyzo and read it. At this point, you will have completed many of the helper functions, and submitted them on PCRS. You should also include them in your MarkUs submission. Now you have only two functions to complete — `visit_next` and `visit_all`. Using the comments in the code as a guide, complete these functions to implement the full Dijkstra's algorithm.

Initially test your code by creating a direct-flight distances dictionary by hand (like the one in this handout) and then calling `visit_all` using different cities as the departure city. Next, add to your program to read a file like `"cities.txt"` and use your helper function to build the direct-flight distance dictionary from the file.

Dijkstra's algorithm is probably the most complex algorithm that you will be implementing this summer. Your implementation likely won't work on the first try. Trace your code with the sample input in Pyzo, and compare the visited and unvisited lists that your code computes at each step with the visited and unvisited lists that you expect (you can also see the video for the values that Michael obtains at each step). When you notice a discrepancy, find its source. Use the Pyzo debugger. Help is also available during office hours

Name your program `dijkstra_cities.py` and submit it as Exercise 5 on MarkUs. On the C4M website, we provide `cities.txt` and `cities2.txt`, as well as a tester program. **Please paste the output of the tester program in your submission.**

For E5, on PCRS, you will complete a function that is analguous to `visit_all`. Implementing it first should help you with completing the MarkUs portion of the assignment.

Part 2: Simulating the Spread of Disease

In Part 2, you will write a simulation of the spread of disease. In the same way that we completed some helper functions for Exercise 4 and then put them together into a full program for Exercise 5. We will start this part by writing some individual helper functions. Exercise Set 6 has four functions for you to write and submit on PCRS. As before, you might find it helpful to develop these functions in Pyzo where you can use the debugger and then copy your solution into PCRS to test it and submit.

Name	**Basic Description**
`get_cities(city_prob_pairs)`	From a list of tuples of cities and probabilities, return a list of cities in the same order.
`get_probabilities(city_prob_pairs)`	From a list of tuples of cities and probabilities, return a list of probabilities in the same order.
`init_zero_sick_population(cities)`	Return a dictionary whose keys are the values in the list `cities`,and whose values are all 0
`build_transition_probs(shortest_distances, alpha)`	Build a dictionary of the probability of a person moving from one city to any other. (More details below.)

Staying Put or Moving to Another City: Implementing build_transition_probs

Eventually our simulation will proceed in time steps. At each time step each individual in each city will either stay in the city, or move to another city. The probability of moving to another city is given by an equation that looks pretty scary at first but it isn't really as complicated as it seems.

Suppose we let $f(A \to B)$ roughly represent the likelihood of moving from city A to city B. We would define it as:

$$f(A \to B) = \frac{1}{(1 + [\text{distance from A to B}])^{\alpha}}$$

α is a constant that we will provide as a parameter to the simulation. Notice that as the distance from A to B gets larger, the chance of someone moving from A to B gets smaller. Also notice that $f(A \to A)$ (the probability of staying in city A if we used this definition), would be 1. This formula is called a *power law.* Empirical research[1] indicates that we power laws are appropriate for modelling human mobility.

The problem with this definition is that we want our simulation to maintain the same total population. (We don't want the same person moving from A to B and **also** moving from A to C and **also** staying in A.) In order to avoid this problem, we won't use the probabilities as we have defined them above. Instead we will *normalize* them by dividing each one by the sum for all the cities. Suppose our list of cities is A through Z. We will calculate the final probability of a person moving from city A to city B as:

$$P(A \to B) = \frac{f(A \to B)}{f(A \to A) + f(A \to B) + f(A \to C) + ... + f(A \to Z)}$$

Notice that as long as the departure city is A, the denominator is the same for every destination city. When you implement this in your program, you should take advantage of this and not be recomputing the same denominator.

Now the probability of staying in the city A is:

$$P(A \to A) = \frac{1}{f(A \to A) + f(A \to B) + f(A \to C) + ... + f(A \to Z)}$$

In order to store these probabilities for repeated use in our program, we will build a dictionary with almost the same form as the one returned by `build_direct_distances` except that instead of holding the distance from one city to another, the second element will be $P(\text{departure city} \to \text{destination city})$ as we have defined it above. The function `build_transition_probs` will return this dictionary. The first parameter to `build_transition_probs` is a dictionary of the shortest distances[1] and the second is α.

Once you have completed these four functions correction on PCRS, you are finished E6 but don't stop now. Start early on putting it all together to complete the entire project. You should do E6 alone on PCRS but you are encouraged to work in pairs for the final project.

Building the Dictionary of shortest distances.

Start by downloading `simulation_starter.py` and copying in your functions from `dijkstra_cities.py` and from E6..

Our function `build_transition_probs` had a parameter that was the dictionary of shortest distances between any pair of cities. Your next task is to complete a function `build_shortest_distances` that will create this dictionary. For each key, the value will be a list of tuples where the first element is a destination city name and the second element is the shortest distance from the key (that departure city) to the destination city. The lists should have every city in the simulation including the departure city itself — with a distance of 0. You might notice that this dictionary has the same basic format as the one that held the direct-flight distances. In this case, the values must include every city in the simulation.

Write and test this function. Make use of the function `visit_all` that you completed as part of E5 and copied into the starter code.

How Many Sick People in Each City?

The simulation will be based on time steps. The interesting data that changes at each time step is how many inhabitants are sick in each city. We will represent this with, you guessed it, another dictionary. The keys will be city names and the values will be the number of currently sick people in that city. At the beginning of the simulation, this dictionary will have a key for every city and all the values except one will be zero. There is nothing to do here since you've already written `init_zero_sick_population(cities)` on PCRS. Notice how the starter code uses this function and then immediately resets the number of sick people in Toronto to 1.

Now that we know how many sick people are in a city at one time step, and the probability of each one travelling somewhere else, we are almost ready to be able to work out how many sick people will be in each city at the next time step. To do this we make use of a function called `choice` from the package `numpy`. Here is some sample code that you should play with in the Pyzo shell which already has the `numpy` package available. Repeatedly call the last two lines in this fragment to see what happens.

[1] You will write a function to build and return this dictionary in the last part of the assignment. For now, just hardcode a dictionary of the correct format to test your function.

```
from numpy.random import choice  # only needs to be done once
probs = [.8, .15, .05]           # these must sum to 1
cities = ["TO", "NYC", "DF"]     # you must have the same number of options as probabilities
n_sick = 6                       # the number of sick people

chosen_dests = list(choice(cities, size=n_sick, p=probs))
print(chosen_dests)
```

Notice that to call `choice` you will need a pair of parallel lists. One will have the city names and other other the probabilities (in the same order). Use your functions `get_cities` and `get_probabilities` that you submitted on PCRS for E6.

Notice how this produces a convenient list of the destination city for each of our sick individuals from **one** source city. You'll need to repeatedly do this calculation for each source city (using the correct transition probabilities for that city). But what do you do with the result?

Writing the function time_step

You need to use this list returned from `choice` to correctly update the dictionary that records how many sick people are in each city at each step. Remember that when a sick person travels to a new city, he leaves the old city. Also important is that the moving sick person doesn't arrive at the new city until the next time step. So if you are calculating the destination for sick people in a loop going one city and a time, you shouldn't add the newly arriving people into the cities until you've finished the loop. So once you worked out the final destinations for all the sick people (from all the cities) you should update the dictionary.

Recovering or Infecting Others

At this point we can work out, for each sick person, **where** they go during a time step. We also need to work out how many recover and how many infect others.You will use the constants β and γ in your Python code. The constants can be adjusted for different runs of the simulator. Each individual in each city will recover with probability β and each individual will infect one other person in the same city with probability γ.

It might be tempting to try to take the number of sick people in a city and multiply it by β to determine how many of these people recover. This approach would work if the numbers of sick people were large, but if we have a small population (for example only 1 sick person in a city), then multiplying by β will give a number less than 1 and if we round down, we will always end up with nobody sick. Instead, use a similar approach as you used to decide where the `n_sick` people should travel. This time use the numpy `choice` command on each time step to decide randomly if each sick individual will recover or remain sick. You can use numpy `choice` one more time to decide if each sick individual will infect someone or not infect anyone on each time step.

C4M Workshops: Human Mobility and Epidemic Modelling

Running the Simulation

Here is an interesting run of the simulation, with $\alpha = 2, \beta = 0.3, \gamma = .3$. As you can see, we start with one infected individual in Toronto, then the disease spreads, but eventually everyone recovers on Day 24.

```
Day 0 {’Mexico City’: 0, ’New York’: 0, ’Toronto’: 1, ’San Francisco’: 0, ’Washington’: 0}
Day 1 {’Mexico City’: 0, ’New York’: 0, ’Toronto’: 2, ’San Francisco’: 0, ’Washington’: 0}
Day 2 {’Mexico City’: 0, ’New York’: 0, ’Toronto’: 1, ’San Francisco’: 0, ’Washington’: 1}
Day 3 {’Mexico City’: 0, ’New York’: 0, ’Toronto’: 0, ’San Francisco’: 0, ’Washington’: 2}
Day 4 {’Mexico City’: 0, ’New York’: 0, ’Toronto’: 0, ’San Francisco’: 0, ’Washington’: 2}
Day 5 {’Mexico City’: 0, ’New York’: 2, ’Toronto’: 0, ’San Francisco’: 0, ’Washington’: 2}
Day 6 {’Mexico City’: 0, ’New York’: 2, ’Toronto’: 0, ’San Francisco’: 0, ’Washington’: 2}
Day 7 {’Mexico City’: 0, ’New York’: 1, ’Toronto’: 0, ’San Francisco’: 0, ’Washington’: 2}
Day 8 {’Mexico City’: 0, ’New York’: 4, ’Toronto’: 0, ’San Francisco’: 0, ’Washington’: 0}
Day 9 {’Mexico City’: 0, ’New York’: 6, ’Toronto’: 0, ’San Francisco’: 0, ’Washington’: 0}
Day 10 {’Mexico City’: 0, ’New York’: 6, ’Toronto’: 0, ’San Francisco’: 0, ’Washington’: 0}
Day 11 {’Mexico City’: 0, ’New York’: 6, ’Toronto’: 0, ’San Francisco’: 0, ’Washington’: 2}
Day 12 {’Mexico City’: 0, ’New York’: 7, ’Toronto’: 0, ’San Francisco’: 0, ’Washington’: 1}
Day 13 {’Mexico City’: 0, ’New York’: 5, ’Toronto’: 0, ’San Francisco’: 0, ’Washington’: 0}
Day 14 {’Mexico City’: 1, ’New York’: 4, ’Toronto’: 0, ’San Francisco’: 0, ’Washington’: 0}
Day 15 {’Mexico City’: 0, ’New York’: 1, ’Toronto’: 0, ’San Francisco’: 0, ’Washington’: 1}
Day 16 {’Mexico City’: 2, ’New York’: 0, ’Toronto’: 0, ’San Francisco’: 0, ’Washington’: 1}
Day 17 {’Mexico City’: 3, ’New York’: 0, ’Toronto’: 0, ’San Francisco’: 0, ’Washington’: 0}
Day 18 {’Mexico City’: 1, ’New York’: 0, ’Toronto’: 0, ’San Francisco’: 0, ’Washington’: 0}
Day 19 {’Mexico City’: 2, ’New York’: 0, ’Toronto’: 0, ’San Francisco’: 0, ’Washington’: 0}
Day 20 {’Mexico City’: 3, ’New York’: 0, ’Toronto’: 0, ’San Francisco’: 0, ’Washington’: 0}
Day 21 {’Mexico City’: 0, ’New York’: 0, ’Toronto’: 0, ’San Francisco’: 0, ’Washington’: 1}
Day 22 {’Mexico City’: 0, ’New York’: 0, ’Toronto’: 0, ’San Francisco’: 0, ’Washington’: 2}
Day 23 {’Mexico City’: 0, ’New York’: 0, ’Toronto’: 0, ’San Francisco’: 0, ’Washington’: 2}
Day 24 {’Mexico City’: 0, ’New York’: 0, ’Toronto’: 0, ’San Francisco’: 0, ’Washington’: 0}
Day 25 {’Mexico City’: 0, ’New York’: 0, ’Toronto’: 0, ’San Francisco’: 0, ’Washington’: 0}
Day 26 {’Mexico City’: 0, ’New York’: 0, ’Toronto’: 0, ’San Francisco’: 0, ’Washington’: 0}
```

In your submission, include another interesting run, with different parameters.

References

[1] Gonzalez, M.C., Hidalgo, C.A. and Barabasi, A.L., 2008. Understanding individual human mobility patterns. Nature, 453(7196), pp.779-782.

[2] Körner, T. 2009. Mathematics and Smallpox. Gresham College Lecture.
http://www.gresham.ac.uk/lectures-and-events/mathematics-and-smallpox