C4M: Human Mobility and Epidemic Modeling

Overview

Using models of epidemics to determine if and how an epidemic can be contained has a surprisingly long history. As far back as the 18th century, the Swiss mathematician and physicist Daniel Bernoulli modelled the spread of smallpox [2]. Modeling an epidemic requires modeling the disease: specifying to what extent it is infectious, how fast individuals can recover, and how people travel from place to place. Once a model has been 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 time it takes to travel between them, your program will first calculate the shortest time from the city with the first infection to every other city. You will then simulate the spread of an epidemic through these cities.

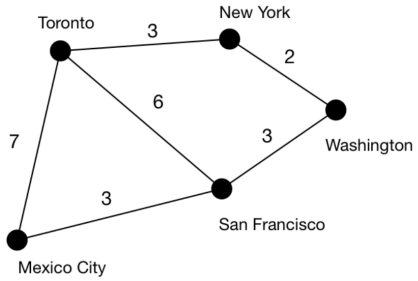
Part 1: Representing Travel Times Between Cities

Your goal for Part 1 is to write functions that will help you read data about flights from a file and build a dictionary with information about direct flights between cities.

Introduction to Graphs

Consider the contents of file `cities.txt` below, which represents data about a flight network shown in the picture on the right:

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



The following statements are true about this example, amongst others:

- There is a 3-hour direct flight from Toronto to New York

- There is a 3-hour direct flight from New York to Toronto
- There are no direct flights from Toronto to Washington
- A person could either get from Toronto to Washington via New York or via San Francisco

In computer science, we call structures like these graphs; note that graphs are used in many other settings, such as bioinformatics and genomics. Using the terminology associated with graphs, the cities are **nodes** and the flights are **edges**. When two nodes are connected by a single edge, the nodes are considered to be **neighbors** of one another; for example, Toronto and New York are neighbors in this example, while Toronto and Washington are not neighbors.

For our program, we will store this flight information in a dictionary. The keys will represent departure cities. The values will be lists of tuples, with each tuple including an arrival city and a time it takes to get there. 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', 2)],
'Washington': [('New York', 2), ('San Francisco', 3)],
'San Francisco': [('Mexico City', 3), ('Toronto', 6), ('Washington', 3)],
'Mexico City': [('Toronto', 7), ('San Francisco', 3) ]}
```

Notice that each pair of neighbors appears twice in this dictionary, with each city in the pair appearing as either a departure or arrival city. This gives us a total of 12 total flights.

Task: Implement Helper Functions

You will need to implement the following functions to complete this part of the assignment, all of which can be found in Part 1 of `project2.ipynb`:

Name	Basic Description
`get_closest(...)`	Returns the tuple from a list of city-distance tuples for which the distance is smallest.
`find_city(...)`	Returns the index in `city_list` of the first tuple that contains `city`.
`process_line(...)`	Returns a tuple containing the information from a string of the format `"{first city}:{second city} distance"`.
`build_adjacent_distances(...)`	Returns a dictionary that contains all the direct-flight distances between pairs of cities.
`get_all_cities(...)`	Returns a sorted list of all the cities that appear in the distance dictionary `city_to_city_dist`.

Part 2: Finding the Shortest Time to Get to a City

Your goal for Part 2 is to calculate the **shortest path** between two cities. Given a dictionary with the durations of the direct flights between different cities, you will figure out the shortest possible time that it could take to travel from one city to any other city (assuming zero layover time). You will do this using Dijkstra's algorithm.

Dijkstra's Algorithm

Suppose that we want to find the shortest path from Toronto to every other city. We will keep track of two lists: one of `visited` cities and one of `unvisited` cities. Each list will contain a tuples with a city name and the distance from our starting point to the departure city. We will initialize the list of `unvisited` cities with our starting point as its only element, and we will initialize the list of `visited` cities to be empty:

```
unvisited = [('Toronto', 0)]
visited = []
```

Note that we set the distance between Toronto and Toronto to be zero since no flight is required.

While the list of `unvisited` cities is not empty, the algorithm proceeds as follows:

1. Remove the city from `unvisited` that is closest to the departure city (in this example, Toronto) and add it to the list of `visited` cities.
2. For each neighbor of the city that was added to `visited`:
   (a) If the neighbor is already in `visited`, you have already found the shortest route to that city and do not have to do anything for that neighbor.
   (b) If the neighbor was not in `visited`, calculate the time it takes to get to the neighbor via the city you just added to `visited`. This time should be (the time it took to get to the city you recently moved to `visited`) + (the time it takes to go from that city to the neighbor via a direct flight).
      i. If the neighbor is not already in `unvisited`, add it to that list with the time you just computed.
      ii. If the neighbor is already in `unvisited`, check to see if the time you just computed is shorter than the one you have in the list so far so far. If the new one is shorter, update that tuple accordingly. Note that there could be more than one route to get to a given city, and that there may be ties.

A step-by-step walkthrough of this algorithm being applied to the earlier example can be found on the C4M website. Before you try to write code for this, go through this process step-by-step using a pen and paper and then check your result by looking at the diagram.

Task: Implement Helper Functions

You will need to implement the following functions to complete this part of the assignment, all of which can be found in Part 2 of `project2.ipynb`:

Name	Basic Description
`visit_next(...)`	Returns the tuple from a list of city-distance tuples for which the distance is smallest.
`visit_all(...)`	Returns the list of shortest distances from `city` to every city in the dictionary `city_to_city_dist`.
`build_shortest_distances(...)`	Returns a dictionary of the shortest distances between any pair of cities.

Dijkstra's algorithm is the most complex algorithm that you will implement in this course, so do not worry if your implementation does not work on the first try. To help you get started, we provide an analogue problem involving a patient queue that you should complete first. Once your code is working for that problem, utilize a similar flow to implement Dijkstra's algorithm for our cities graph. Before you test your code using the `city_to_city_dist` dictionary we provide, you might want to test your code by creating an even simpler dictionary with 3 cities. Remember the various techniques we discussed for testing your code: inspecting intermediate results with `print()`, using a debugger, etc.

Part 3: Simulating the Spread of Disease

Your goal for Part 3 is to write functions that will help you simulate the spread of a disease.

Calculating Transition Probabilities

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. Suppose we let $f(A \to B)$ roughly represent the likelihood of a person moving from city A to city B. We will define it using a **power law**, which has been shown to be appropriate for modeling human mobility [1]:

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

α is a constant that we will provide as a parameter to the simulation; the larger alpha gets, the smaller the likelihood of transition. 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 — would be 1.

The problem with this definition is that we want our simulation to maintain the same total

population. In other words, we do not want the same person moving from A to B **and** moving from A to C **and** staying in A. To avoid this problem, we will not use the above equation for our probabilities. We will instead normalize those values by dividing each one according to the sum of the likelihoods that people will travel to all 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 follows (note the use of P vs. f):

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

Now, the probability of staying in the city A is not one, but rather:

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

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 recompute the denominator.

Task: Implement Helper Functions

You will need to implement the following functions to complete this part of the assignment, all of which can be found in Part 3 of `project2.ipynb`:

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

In order to store the transition probabilities for repeated use in our program, we will build a dictionary with almost the same form as the one returned by `build_adjacent_distances()` except that instead of holding the distance from one city to another, the second element will be P(departure city $\to$ destination city). The function `build_transition_probs()` will return this dictionary. The first parameter to `build_transition_probs()` is a dictionary of the shortest distances. 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.

Part 4: Progressing the Simulation

Your goal for Part 4 is to write the code that will advance your simulation by an incremental time step with some degree of randomness so that you get different results each time.

Task: Counting 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 yet another dictionary. The keys will be city names, and the values will be the number of people who are currently sick 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 0. There is nothing to do here since you have already written `init_zero_sick_population()`. 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 a given time step and the probability of each one travelling somewhere else, we are almost ready to work out how many sick people will be in each city at the next time step. To do this, we will make use of a function called `choice()` from the package `numpy`. `choice()` requires three inputs: (1) a list of items to choose from, (2) a parallel list indicating the probability of choosing each item in the first list, and (3) the number of items that should be chosen with replacement. The function uses random selection, so it will produce a different result almost every time. Here is some sample code:

```
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, p=probs, size=n_sick))
print(chosen_dests)
```

In the function `time_step()`, you should use `get_cities()`, `get_probabilities()`, and `choice()` to 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, they leave the old city. It is also important to remember that the sick person who travels to a new city should not arrive there until the next time step, so you should only update the dictionary after you have worked out the final destinations for all the sick people from all the cities.

Task: Recovering or Infecting Others

At this point, we can work out where each sick person goes during a time step. We now need to extend `time_step()` so that we know how many people recover and how many people infect others after each iteration. You will use the constants β and γ to modify the likelihood of these occurrences, where β is the probability of recovery and γ is the probability of an

individual infecting another person in the same city. It might be tempting to take the number of sick people in a city and multiply it by β to determine how many people recover. This approach would work if the number of sick people is large; if we have a small population, however, multiplying by β will give a number less than 1 and we will end up with nobody sick if we round down. Instead, use `choice()` to randomly decide if each sick individual will recover or remain sick, and use `choice()` once more to decide if each sick individual will infect someone or not infect anyone.

Part 5: Putting It All Together

Now that you have written all of the helper functions, it is finally time to run your simulation! Download `cities.txt` and put it in your working directory. In Google Colab, you can find your working directory by clicking the `Files` icon on left side of the screen. You can then either drag-and-drop the file into that window or click `Upload to Session Storage` at the top to bring up a File Dialog. We have provided the main program that will call upon your helper functions to run the simulation, so as long as you upload `cities.txt` and all of your helper functions work, your simulation should be ready to go.

Here is an interesting run of the simulation with $\alpha = 2, \beta = 0.3, \gamma = 0.3$. As you can see, we start with one infected individual in Toronto and the disease spreads over time, but everyone eventually 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}
```

Remember that you might get different results if you run the simulation since there is some randomness associated with how people travel, get sick, and recover. Run a simulation on

your own using a different set of parameters that you think might be interesting. Save the result as a text file called `simulation_output.txt`.

Submission Checklist

- ☐ `project2.ipynb` (as a Google Colab link): Your completed Jupyter notebook that passes all of the provided tests.
- ☐ `simulation_output.txt`: A text file that contains the printout of an interesting simulation.

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