How Contagious are Airports? Using COVID-19 For Epidemiology Simulation

Christoph Uhl[1] and Joey Vijayam[2]

[1]*Christoph Uhl, Email: christoph.uhl@colorado.edu*
[2]*Joey Vijayam, Email: joseph.vijayam@colorado.edu*
[3]*CSCI 4314 Dynamic Models in Biology*

Abstract

The COVID-19 pandemic caused an unprecedented amount of talk about Epidemiology. Many people claim correctly that it is due to globalization that a virus such as COVID-19 can spread to every country on earth in just a few months. Air travel is an enormous contributor to globalization as it massively facilitates going to distant, hard-to-get-to places. In this paper, we investigated how much airports impact the spread of disease through simulating COVID-19 in the United States with only cities that contain primary airports. We found that without preventative measures, people in those cities are almost guaranteed to contract the virus.

1 Background

The COVID-19 epidemic has sparked countless discussions about the world's effectiveness in mitigating the spread of disease amidst a worldwide epidemic. During the months of February and March we witnessed the chaos associated with travel and its effects on the pandemic. Since this has been at the forefront of our thoughts lately, we have set out to examine the impact of air travel on disease spread. To do so, we have simulated a contagion starting at the same place and having the same characteristics as COVID-19, including fatality rate, type of spread (airborne, contact, etc), symptoms, and so on. However, we have limited the simulation to airports specifically in order to examine the role of airports and air travel in the spread of the disease.

Although we have much more advanced technology than we did a hundred years ago, the world is a much more connected place than it was a hundred years ago. Since the most prominent method of travel for people around the world is through airplanes and airports themselves are centers of contagion, pinpointing its effect on COVID-19 may help understand their significance during this pandemic. We believe that with the current COVID-19 pandemic, there is more epidemiology data available than for previous epidemics, and we would like to leverage this data to find meaningful insights on how airports spread disease. In addition, this perspective may help future governments and institutions prepare for the next epidemic.

Airports play an enormous role in connecting people across the world, millions of people travel for both leisure and business every day. In 2014 alone over 3.3 billion passengers traveled by air from 41,000 airports [1]. The impact of various epidemics due to air travel is often labeled as one of the root causes of epidemics turning into pandemics, but the extent of which it may be so is unclear. So, if we can get a better idea of how much airports actually enable viruses, we will be able to better simulate future virus spread which can lead to setting better protocols for preventing pandemics.

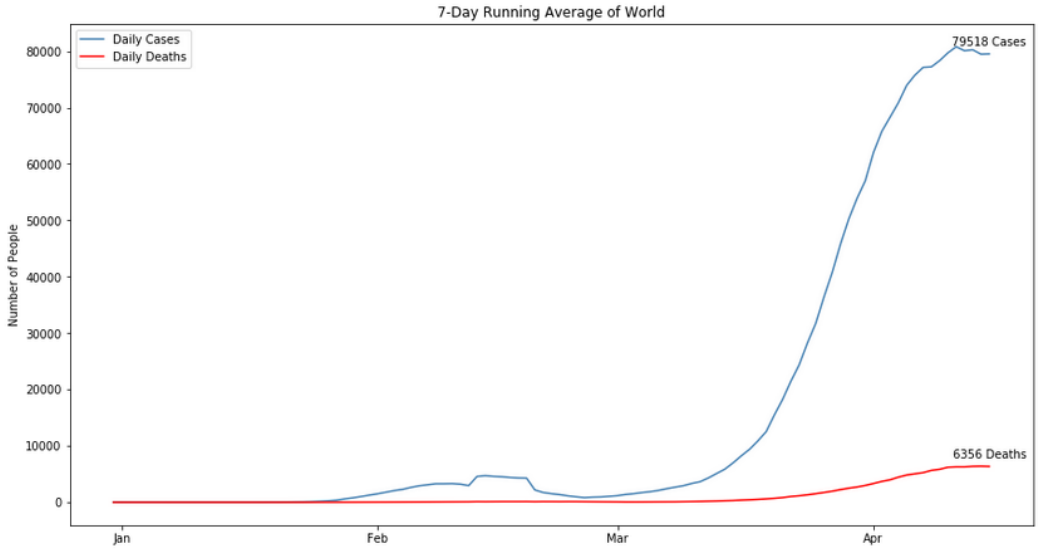


Figure 1. With data gathered from Johns Hopkins, this graph shows reported cases and deaths for COVID-19 during the first few months of 2020. By mid-April 2020, there were around 79,518 cases and 6,356 deaths reported in the U.S.

2 Methods

Onto our tests themselves for creating and performing our simulation. To perform our simulation, we gathered and parsed necessary data, formed a large graph containing smaller graphs for every city with nodes pertaining to each city's population, and ran our model with different variables to see the potential effect of travel on disease spread.

2.1 Data Collection

First we got all the cities that have airports. Then we used this dictionary to gather population information. The libraries involved for this process were Mwclient, which gave us web scraping abilities and BeautifulSoup, which allowed easy and efficient parsing through HTML text.

For data collection we found on Wikipedia a table listing all United States primary airports, which are airports with at least 10,000 enplanements per year. We then found another Wikipedia page that linked us to each state's list of cities along with their populations. To gain access to this data, we used the Mwclient library, which allows extraction of HTML of a given website. For all pages that we extracted HTML text from, we used the BeautifulSoup library, which provides an easy-to-use and efficient way of parsing through HTML. By building our own web scraper, we first collected names of cities with airports along with the state they're from, and then successfully scraped over 50 pages to gather city population data for 49 U.S. states. Oregon was not included in this list because it involved expanding the web scraper to different websites with varying representations of population data.

We structured our data in dictionaries wherein each city's statistics were saved as lists under its state, thereby acting as the dictionary key. Besides enabling an easy way to access desired information, this structure also acted to disallow duplicate names such as Portland, which is a city in both Oregon and Maine. We had now gathered and organized necessary data for our simulation.

We experienced some difficulty with deciding when to begin our simulation. Because the first confirmed case of COVID-19 occured in Seattle, Washington, we thought about starting the simulation there. However, the first death due to Coronavirus was from California in early February, much before when the first official Coronavirus case was reported in California. Nevertheless, our simulation works best when looking at the spread of disease within the country. Therefore, we

Table 1. Showcase of Data Collection. First 2 states by alphabetical order are shown, followed by their city names with airports and their respective populations. This table indicates our data (in dictionary form) describing all states and cities with airport data.

State	City	Population
Alabama	Birmingham	212237
Alabama	Dothan	65496
Alabama	Huntsville	180105
Alabama	Mobile	195111
Alabama	Montgomery	20576
Alaska	Anchorage	291826
Alaska	Aniak	501
Alaska	Bethel	6080
Alaska	Cordova	2239
Alaska	Dillingham	2329
Alaska	Fairbanks	31535

examined when the travel ban was implemented by the United States government prohibiting foreign travel to other nations. Since this was implemented on March 21, 2020, we looked at the number of cases in each county to decide where to start our simulation. Specifically we only set infected nodes in cities that are in counties with at least 800 confirmed cases (based on Johns Hopkins University data).

2.2 Graph Assembly

Secondly, we created our simulation by joining together many random graphs into one large network, using the NetworkX library on every step. Although we ideally would represent one person with one node, due to performance issues, we decided to have each node represent 1,000 people. Therefore, every infected node corresponds to 1,000 people being infected. Thus, an edge represents interaction between two groups of 1,000 homogenous people (either all are susceptible, infected, or recovered), indicating possible transmission from one node to another.

Each of the random graphs is built using the configuration model, a random network model that resembles community structure. This step seemed most applicable to us because we don't have full, accurate data on all cities in America, so using random graphs made the most sense for our simulation.

For accumulating all of the random graphs into one large graph (the base graph), we implemented some methods to keep track of our data for each city. We created an additional dictionary storing all cities, again ordered by state, with an ID-range-tuple to give the lowest (inclusive) and the highest node IDs (exclusive) that correspond to the particular city. This was to keep track of which nodes correspond to which city, and which city corresponds to which state.

Since we decided to represent the U.S. with an incomplete list of cities and transportation methods, we also understand that therefore our simulation model of the United States is incomplete. Further work could involve creating a more holistic model of the entire country where every city is represented.

2.3 Simulation Setup

Thirdly, we ran our simulation to analyze the effectiveness of air travel on the spread of the COVID-19 disease. The simulation can be explained in two parts: first is setup, in which all of the basic variables and node states are defined, and the second is running through each timestep and calculating the

spread.

To define the basic node states, we use a given dictionary that contains all of the infected cities (minimum of 800 confirmed cases), along with the number of nodes that should be infected for the given city (1 for every 1000 cases). Each node has a label of either 'S', 'I', or 'R'. At the start every node has an 'S' label, except for those pre-determined to be infectious, which have an 'I' label. The 'S' label indicates that the node would be susceptible to the disease, an 'I' label indicates that a node is infected, and an 'R' label indicates that the node has recovered. Simulating our COVID-19 disease spread assumes that once a person recovers from the virus, they cannot contract it again, which is currently not known to be true or untrue. To model recovery, we chose a recovery time for each node, meaning that instead of giving every node a probability to recover, we let every node stay infected for a certain period of time, but once over, the node is guaranteed to have recovered. In our case this period was 10 days. Based on this information, we performed the simulation in which there is a probability (depending on variable R) for transmission.

Though most recovery indicators would use probability values to indicate who has recovered and who hasn't (leading to death), and how fast each patient recovered, we instead decided to use a recovery time for patients. This is due to the lack of adequate data concerning patient recovery for COVID-19, and because we didn't deem it necessary for our study. Since this simulation mainly tracks the spread of COVID-19 to populations in cities with airports around the United States, we did not find it necessary to detail the recovery probabilities for each patient, and we understand that this leads to less accurate recovery data. Since the recovery times for COVID-19 patients hovers around 10 days, this is the metric we used as the recovery time parameter for our simulation.

Our simulation can only be effective if air travel can effectively be incorporated into our model. Therefore, in order to simulate flights, we implemented a few steps at each time-step of our model. First, we chose a random node within each city. Next, we randomly chose a certain number of other cities to connect the chosen node to. This indicates that the infection will spread to the cities chosen. Lastly, we remove all created edges.

3 Results

We set out to examine the effectiveness of airports on the spread of disease. Our graphs showcase the discrepancies between the real world and simulation data. Our simulation shown in Figure (2) indicates that if we were to abide by the R_0 value indicated for COVID-19 according to the Center for Disease Control (giving us a value of 5.7)[**1**], our simulation shows the importance of preventative measures with COVID-19. As can be seen with various values for R_0 ($R_0 = 2.0$, $R_0 = 4.0$, $R_0 = 6.0$) we get similar slopes wherein around 80 percent of the population becomes infected, see related work for more on this. It is much different compared to the real cases, where much less than 10 percent of the population gets infected.

Of course there are a few assumptions that are made for the model, and these best showcase the importance of preventative measures with treating the COVID-19. Firstly, we have assumed that everyone who gets in contact with someone who has the disease will get the disease. We divided the R_0 group by the mean degree, and used this value to determine the probability for infection rates in our simulation. As we know, this does not reflect real world information. Though it's unclear to what extent, we know that not everyone who is in contact with someone with COVID-19 receives the disease. Next, it also assumes that Secondly, and most importantly, our data does not take into account various measures that have been set in place to prevent the spread of the disease. Since people are ordered to stay in their homes, flight travel is banned, and every citizen is encouraged to have the least amount of contact with people as possible, the spread of the disease has been greatly limited. Our random graph is based on normal community structure, but it varies from what we see in the real because preventative measures are being taken to limit the spread of the disease.

It is important to note that the percentage of infection is less than 100 percent because the total

population is not taken into account for this simulation. Since we are only taking into account the populations of cities with airports, the infected rate peaks at about 80 percent.

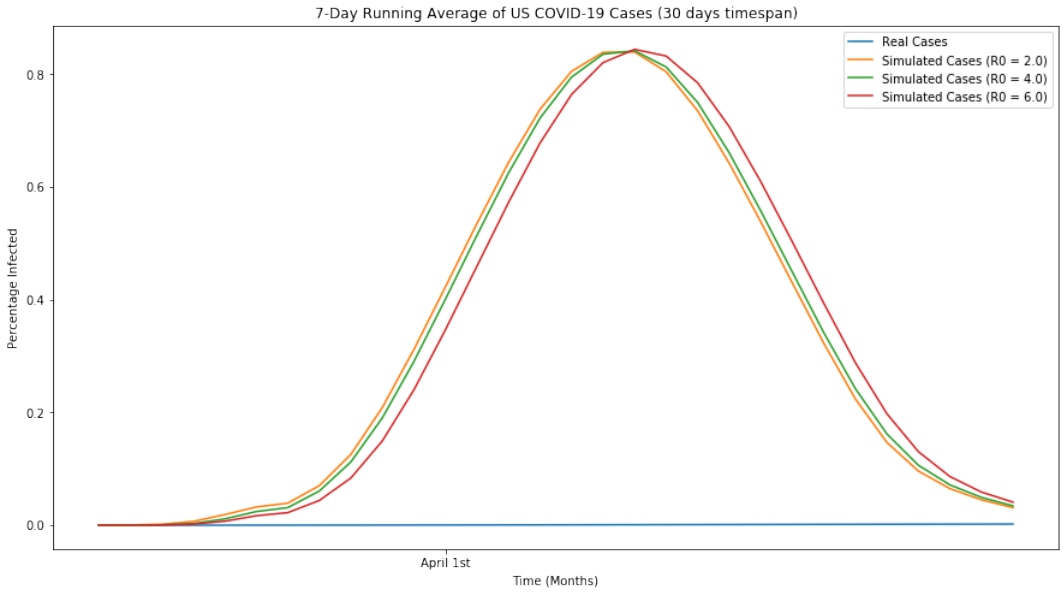


Figure 2. Number of people infected, un-scaled simulated data. In blue line, real cases, and simulated cases with R_0 shown with values 2.0, 4.0, and 6.0. Time in months shown on x-axis, percentage Infected shown on the y-axis (0.2 = 20 percent)

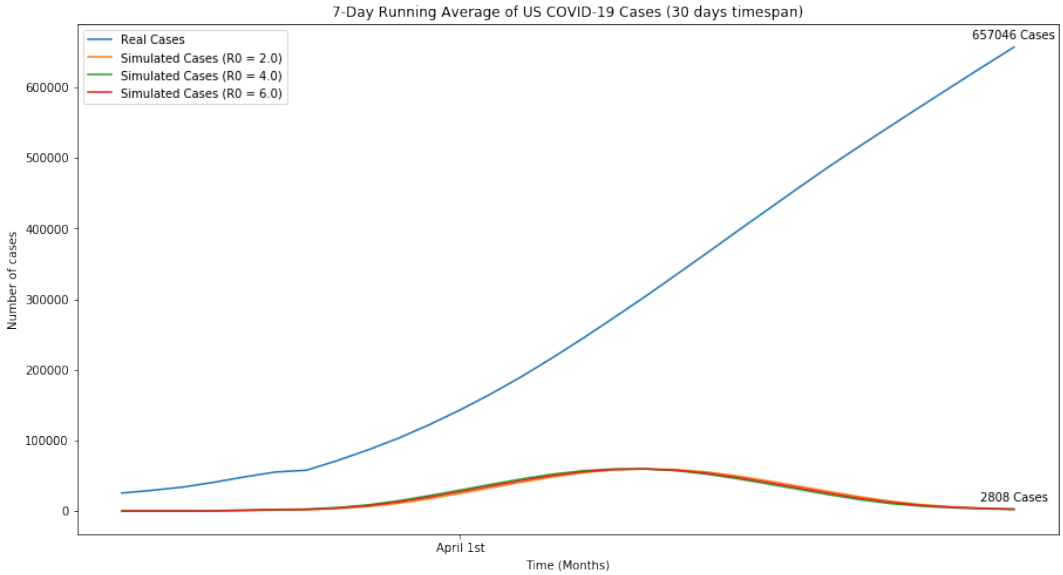


Figure 3. Number of people infected, un-scaled simulated data. In blue line, real cases, and simulated cases shown with R_0 shown with values 2.0, 4.0, and 6.0. Time in months shown on x-axis, number of cases shown on y-axis. Total 2808 cases simulated and 657046 real cases.

4 Related Work

In our search for related work we have been mostly empty-handed. We believe this is due to the sensitive nature of the COVID-19 pandemic. However, some works that we were able to use in our project are related to Network Science.

When it came to finding an effective way to write a epidemiology simulation, we heavily relied on Aaron Clauset's lecture notes [**2**] in which he outlined the concept and noteworthy information about epidemiology.

When it came to choosing the best random model for building community structure, we found that previous work has determined multiple good reasons for using the configuration model. Originally, we intended to use the Erdös-Rényi model, however after reading Lee Bernick's paper on "Modeling Human Networks Using Random Graphs"[**3**], we realized that the configuration model is the way to go. We had also discussed using the Watts-Strogatz model due to its small-world-like behavior, but decided against it when it came to our attention that it was not a scale-free network. A scale-free network has a degree distribution that follows a power-law distribution. In other words, most nodes have a low degree, and only few nodes have a very high degree. This is a phenomenon that occurs in social networks, and one that brings out community structure in a graph by causing 'hubs' to form. Since we wanted to best model community structure in our random graph, we decided that the configuration model with its scale-free attribute would be the best choice.

For our simulation, we needed to find a probability of the disease spreading from an infected node to a susceptible node p. Since we simulated COVID-19 spread, we wanted to base this probability on the estimated R_0 value of COVID-19. R_0 gives the expected number of people that an infected person spreads their disease to. In the case of COVID-19, we found that current estimates of R_0 value lie around 5.7, with a 95% Confidence Interval being 3.8-8.9 [**1**]. For our purposes, we didn't stick exactly to these estimated values, as our conversion from R_0 to p is given by

$$p = \frac{R_0}{\hat{k}} \tag{1}$$

which means that p quickly becomes greater than 1 if we have a low mean degree $\hat{k}$. Furthermore, since we used the configuration model to create our random graphs, $\hat{k}$ was fairly small at 1.9. When R_0 is greater than 1.9, we get that the probability of transmission $p > 1$. In which case we reset $p = 1$ so we can actually run the simulation.

5 Project Contributions

This project was worked on by Joey Vijayam and Christoph Uhl. Joey Vijayam focused on writing the essay and preparing for the presentation. Christoph focused on gathering data and finding the right model for our simulation. We worked together on integrating the model with the data collected and interpreting the results for the data. In addition, all data collection, presentation planning and execution, model simulations, and paper formation was worked on by both of us, with both of us putting equal hours towards this project.

Supplementary Material

Supplementary material for this paper is our code (Python Notebook) used for this project. Screenshots of our notebook are attached to the end of this paper.

References

[1] Sanche S, et al., *High contagiousness and rapid spread of severe acute respiratory syndrome coronavirus 2*, Emerg Infect Dis., 2020, https://doi.org/10.3201/eid2607.200282.

[2] Clauset A., *Network Epidemiology*, University of Colorado Boulder, CSCI 3352, Biological Networks, http://tuvalu.santafe.edu/~aaronc/courses/3352/csci3352_2020_L8.pdf.

[3] Bernick L., *Modeling Human Networks Using Random Graphs*, MIT Mathematics, 2018, http://www-math.mit.edu/~apost/courses/18.204_2018/Lee_Bernick_paper.pdf.

In [1]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
%matplotlib inline

import mwclient #for getting wikipedia data
from bs4 import BeautifulSoup #for extracting desired information from html data
from bs4 import Comment #for looking through comments

import string #for string processing
import locale #for turning string to number
import re #for string processing

import json
import networkx as nx
import copy
```

Airport contagion

Things we are trying to accomplish in this notebook are:

- Run a simulation of COVID-19 spreading with set locations (airports) as the only places on earth
- Get the same number of cases across the US out of the simulations as observed with real data
- Find and use an adequate metric with which to determine if the simulation is close enough

The goal is to use the empirical infectiousness of COVID-19 and simulate the spread of the virus over time, based on 24 timesteps (1 for every hour) in a day, where at every step, we have a chance equal to the infectiousness that another person gets the disease.

To determine the impact of air travel on disease spread, we measure the following:

- Only United States impact
- Dealing with continued income of infected people from outside U.S., either
  - start with when the US closed its international airtravel (January 31st)
  - OR start from the beginning (with the first 4 cases, and randomly adding some to account for income)
- Measure only the states

Getting US cities with primary airports

Using the MediaWiki API with the mwclient library, we can get the parsed data of a wikipedia page. With that, we gain access to information in the tables of the wiki page. If we didn't do this, we would just see the code that should populate the table, not the table after it is parsed.

Getting US cities with primary airports

Using the MediaWiki API with the mwclient library, we can get the parsed data of a wikipedia page. With that, we gain access to information in the tables of the wiki page. If we didn't do this, we would just see the code that should populate the table, not the table after it is parsed.

In [2]:
```
site = mwclient.Site("en.wikipedia.org")
result = site.api('parse', prop='text', pageid=4718801) #using pageid here, because I couldn't get
the call to work with the title
airports = result['parse']["text"]["*"]
```

Since the data is in html form, we can use BeautifulSoup to extract desired information.

In [3]:
```
soup = BeautifulSoup(airports, 'html.parser')
```

After looking at the data a bit, I noticed that the information we're after is in the third table of the page, so we first get the third table tag, and then we go through each row, keeping only the city name.

In order convert the cities to population count, we need to keep the cities ordered by their state. This also helps dealing with duplicate names like Portland, which is a city in both Oregon and Maine.

In [4]:
```
#get the table that has contains all of the primary airport names
airport_table = soup.findAll("table")[2]

# find all rows, which are indicated by the 'tr' tag, then get the first element (which is the cit
y), indicated by the 'td' tag
#   furthermore, we need to check if 'td' tag exists and if it does, if it has a 'cite' tag,
#   which indicates that the current row is actually just indicating a new state (in which case we
make a new key in the dict),
#   otherwise we can add a state
airport_cities_by_state = {}
current_state = ""
for row in airport_table.findAll("tr"):
    if(row.find('td')):
        if row.find('td').find("cite"):
            current_state = row.find('td').text
            if current_state == "AMERICAN SAMOA": #this indicates that we are now covering US terr
itories
                break
            else:
                airport_cities_by_state[current_state] = []
        else:
            new_city = row.find('td').text[:-1] #leave out the last character since it's a newline
```

In [4]:
```
#get the table that has contains all of the primary airport names
airport_table = soup.findAll("table")[2]

# find all rows, which are indicated by the 'tr' tag, then get the first element (which is the cit
y), indicated by the 'td' tag
#   furthermore, we need to check if 'td' tag exists and if it does, if it has a 'cite' tag,
#   which indicates that the current row is actually just indicating a new state (in which case we
make a new key in the dict),
#   otherwise we can add a state
airport_cities_by_state = {}
current_state = ""
for row in airport_table.findAll("tr"):
    if(row.find('td')):
        if row.find('td').find("cite"):
            current_state = row.find('td').text
            if current_state == "AMERICAN SAMOA": #this indicates that we are now covering US terr
itories
                break
            else:
                airport_cities_by_state[current_state] = []
        else:
            new_city = row.find('td').text[:-1] #leave out the last character since it's a newline
character

            #check for edge cases
            if new_city.find(', ') >= 0:
                #handles hawaii city names that include island name
                new_city = new_city[:new_city.find(', ')]

            if new_city.find('/') >= 0:
                #this case is tricky, so simply don't worry about it, the '/' indicates multiple c
ity names, so choosing the right one is tough
                new_city = new_city[:new_city.find('/')]

            airport_cities_by_state[current_state].append(new_city)
```

In [5]:
```
airport_cities_by_state
```

Out[5]:
```
{'ALABAMA': ['Birmingham', 'Dothan', 'Huntsville', 'Mobile', 'Montgomery'],
 'ALASKA': ['Anchorage',
  'Anchorage',
  'Anchorage',
  'Aniak',
  'Barrow',
  'Bethel',
```

In [5]:
```
airport_cities_by_state
```

Out[5]:
```
{'ALABAMA': ['Birmingham', 'Dothan', 'Huntsville', 'Mobile', 'Montgomery'],
 'ALASKA': ['Anchorage',
  'Anchorage',
  'Anchorage',
  'Aniak',
  'Barrow',
  'Bethel',
  'Cordova',
  'Deadhorse',
  'Dillingham',
  'Fairbanks',
  'Galena',
  'Homer',
  'Juneau',
  'Kenai',
  'Ketchikan',
  'King Salmon',
  'Kodiak',
  'Kotzebue',
  'Nome',
  'Petersburg',
  'Sitka',
  "St. Mary's",
  'Unalakleet',
  'Unalaska',
  'Valdez',
  'Wrangell',
  'Yakutat'],
 'ARIZONA': ['Bullhead City',
  'Flagstaff',
  'Grand Canyon',
  'Mesa',
  'Page',
  'Peach Springs',
  'Phoenix',
  'Tucson',
  'Yuma'],
 'ARKANSAS': ['Fayetteville', 'Fort Smith', 'Little Rock', 'Texarkana'],
 'CALIFORNIA': ['Arcata',
  'Bakersfield',
  'Burbank',
  'Fresno',
  'Long Beach',
  'Los Angeles',
```

```
 'San Antonio',
 'Tyler',
 'Waco',
 'Wichita Falls'],
'UTAH': ['Cedar City', 'Ogden', 'Provo', 'Salt Lake City', 'St. George'],
'VERMONT': ['Burlington'],
'VIRGINIA': ['Charlottesville',
 'Lynchburg',
 'Newport News',
 'Norfolk',
 'Richmond',
 'Roanoke',
 'Washington',
 'Washington'],
'WASHINGTON': ['Bellingham',
 'Friday Harbor',
 'Pasco',
 'Pullman ',
 'Seattle',
 'Seattle ',
 'Spokane',
 'Walla Walla',
 'Wenatchee',
 'Yakima'],
'WEST VIRGINIA': ['Charleston', 'Clarksburg', 'Huntington', 'Morgantown'],
'WISCONSIN': ['Appleton',
 'Eau Claire',
 'Green Bay',
 'La Crosse',
 'Madison',
 'Milwaukee',
 'Mosinee',
 'Rhinelander'],
'WYOMING': ['Casper',
 'Cody',
 'Gillette',
 'Jackson Hole',
 'Laramie',
 'Rock Springs']}
```

And there it is! But looking at the list of airport cities, we see some duplicates, like Anchorage. This simply means there is more than one primary airport in that city. For the purpose of this project, we don't care. So we may need to remove duplicate values, but for now let's move on to getting city populations!

Getting city population with MediaWiki API

Getting city population with MediaWiki API

Background:

After spending some time looking for the ideal database containing all US cities with their estimated population, I came out empty handed. Since the ideal didn't exist, we now are stuck with the next best thing, which is to go to the state specific list-of-cities Wikipedia article and query that.

There is a Wikipedia article that links to all of these, based on the state, so it seems that using their links is the way to go. This is the article I'm referring to.

In [6]:
```
#used to convert strings like "1,000" to integer 1000
locale.setlocale(locale.LC_ALL, 'en_US.UTF-8');
```

In [7]:
```
def process_state(state_name):
    #capitalize the string and replace whitespaces with underscores
    return string.capwords(state_name.replace(" ", "_"), sep = "_")

def is_city_list(contender_str):
    lc = contender_str.lower()
    if lc.find("cities") >=0 or lc.find("municipalities") >= 0 or lc.find("_incorporated") >= 0:
        return True
    return False

def get_name_and_population_indices(column_names):
    #this will take the first mention of population in the expected table headers as the index for
population
    #  more recent population information tend to come first
    name_index = -1
    name_found = False
    pop_index = -1
    pop_found = False
    for i, col_name in enumerate(column_names):
        if not name_found and ("name" in col_name or "city" in col_name or "municipality" in col_n
ame):
            name_index = i
            name_found = True
        if not pop_found and ("pop." in col_name or "population" in col_name or "estimate" in col_
name or "2020 census" in col_name):
            pop_index = i
            pop_found = True
        if pop_found and name_found:
            return name_index, pop_index
```

In [7]:
```
def process_state(state_name):
    #capitalize the string and replace whitespaces with underscores
    return string.capwords(state_name.replace(" ", "_"), sep = "_")

def is_city_list(contender_str):
    lc = contender_str.lower()
    if lc.find("cities") >=0 or lc.find("municipalities") >= 0 or lc.find("_incorporated") >= 0:
        return True
    return False

def get_name_and_population_indices(column_names):
    #this will take the first mention of population in the expected table headers as the index for
population
    #  more recent population information tend to come first
    name_index = -1
    name_found = False
    pop_index = -1
    pop_found = False
    for i, col_name in enumerate(column_names):
        if not name_found and ("name" in col_name or "city" in col_name or "municipality" in col_n
ame):
            name_index = i
            name_found = True
        if not pop_found and ("pop." in col_name or "population" in col_name or "estimate" in col_
name or "2020 census" in col_name):
            pop_index = i
            pop_found = True
        if pop_found and name_found:
            return name_index, pop_index
    return name_index, pop_index

def get_indices(soups):
    #given list of soup, extract the index of the correct table
    #  utilizes get_name_and_population_indices
    pni = -1; ppi = -1
    for i, soup in enumerate(soups):
        columns = [col.text.lower() for col in soup.findAll("th")]
        pni, ppi = get_name_and_population_indices(columns)
        if pni >= 0 and pni <= 3 and ppi >= 0:
            return i, pni, ppi
    return 0, pni, ppi #randomly choose the first table

def get_city_population_tuple(name, pop):
    try:
        return (name, locale.atoi(pop))
    return 0, pni, ppi #randomly choose the first table

def get_city_population_tuple(name, pop):
    try:
        return (name, locale.atoi(pop))
    except ValueError:
        #issue is square brackets at the end of the string
        cutoff = pop.find("[")
        return (name, locale.atoi(pop[:cutoff]))

def get_soup(page_name):
    result = site.api('parse', prop='text', page=page_name)
    wiki_html = result['parse']['text']['*']
    soup = BeautifulSoup(wiki_html)
    redirect = soup.findAll("div", {"class": "redirectMsg"})
    if redirect:
#         print("Redirecting")
        return get_soup(redirect[0].find('a').text)
    return soup

def text_processing(text):
    # Strip leading and tailing whitespaces, remove newline and extraneou characters
    text = re.sub(r"\n|†|‡|\*|\[*.]", "", text)
    text = text.strip()
    return text

def handle_new_city(populations, candidate_tuple):
    #check if new city is already in the list, if it is, keep the one with higher population, othe
rwise just append new city
    no_duplicate_tuple = True
    for i, (name, pop) in enumerate(populations):
        if candidate_tuple[0] == name:
            no_duplicate_tuple = False
            if pop > candidate_tuple[1]:
                #do not add repeating cities, keep only the largest population
                break
            else:
                populations[i] = candidate_tuple
    if no_duplicate_tuple:
        populations.append(candidate_tuple)
    return populations


def get_city_populations(cities, page_name):
    cur_soup = get_soup(page_name)
    table_tags = cur_soup.findAll("table")
    if not table_tags:
```

```
def get_city_populations(cities, page_name):
    cur_soup = get_soup(page_name)
    table_tags = cur_soup.findAll("table")
    if not table_tags:
        table_tags = cur_soup.findAll("table", {"class": "wikitable sortable"})
        if not table_tags:
#             print("No table_tags")
            return None

    tindex, nindex, pindex = get_indices(table_tags)
    table = table_tags[tindex]

    if nindex == -1 and pindex == -1:
#         print("Could not find either name column nor population column")
        return None
    if nindex == -1:
#         print("Could not find name column")
        return None
    if pindex == -1:
#         print("Could not find population column")
        return None

    populations = []
    for row in table.findAll("tr")[1:]: #start from the third entry because the first is set-up
        subtractor = 0
        if row.find("th"):
            #this means we need to subtract 1 from the indices, as they assumed all columns to be
use the 'td'
            subtractor = 1

        possible_city_tags = row.findAll("td") #not a precise way to find the city names, but it d
oesn't need to be, since we are checking base on our list of cities

        #Check if findAll returns a list or none
        if possible_city_tags:
            city_tag = possible_city_tags[nindex-subtractor]
            city = text_processing(city_tag.text)
            if city in cities:
                #using a try/except statement in order not to have to find("[") in all strings
                populations = handle_new_city(populations, get_city_population_tuple(city, row.fin
dAll("td")[pindex-subtractor].text))
            else:
                try:
                    #if city name is not found in a 'td' tag, we can look at alternative tags
                    alt_city_tag = row.find("th") #as in the case of california, the city names ar
e under 'th' tags
        if row.find("th"):
            #this means we need to subtract 1 from the indices, as they assumed all columns to be
use the 'td'
            subtractor = 1

        possible_city_tags = row.findAll("td") #not a precise way to find the city names, but it d
oesn't need to be, since we are checking base on our list of cities

        #Check if findAll returns a list or none
        if possible_city_tags:
            city_tag = possible_city_tags[nindex-subtractor]
            city = text_processing(city_tag.text)
            if city in cities:
                #using a try/except statement in order not to have to find("[") in all strings
                populations = handle_new_city(populations, get_city_population_tuple(city, row.fin
dAll("td")[pindex-subtractor].text))
            else:
                try:
                    #if city name is not found in a 'td' tag, we can look at alternative tags
                    alt_city_tag = row.find("th") #as in the case of california, the city names ar
e under 'th' tags
                    alt_city = text_processing(alt_city_tag.text)
                    if alt_city in cities:
                        # from the currrently encountered problem cases, it is only the first colu
mn (if any)
                        #   that is represented with a 'th' tag, rather than a 'td' tag, so just r
educe pop index by 1
                        populations = handle_new_city(populations, get_city_population_tuple(alt_c
ity, row.findAll("td")[pindex-subtractor].text))
                except:
                    continue
    return populations

def get_city_pop_dictionary(cities_dict):
    link_soup = get_soup("Lists_of_populated_places_in_the_United_States")
    my_dict = {}
    for state in cities_dict.keys():
        for sibling in link_soup.find(id = process_state(state)).parent.find_next_siblings(limit=
4)[1:]:
            for contender_tag in sibling.findAll("li"):
                if is_city_list(str(contender_tag)) or len(sibling.findAll('li')) == 1:
                    my_dict[state] = get_city_populations(cities=cities_dict[state], page_name=con
tender_tag.find("a").attrs['title'])
    return my_dict

city_pop_by_state = get_city_pop_dictionary(airport_cities_by_state)
```

```
('Long Beach', 462257), ('Los Angeles', 3792621), ('Mammoth Lakes', 8234), ('Monterey', 27810), ('
Oakland', 390724), ('Ontario', 163924), ('Palm Springs', 44552), ('Redding', 89861), ('Sacramento
', 466488), ('San Diego', 1301617), ('San Francisco', 805235), ('San Jose', 945942), ('San Luis Ob
ispo', 45119), ('Santa Barbara', 88410), ('Santa Maria', 99553), ('Santa Rosa', 167815), ('Stockto
n', 291707)]
```

Unfortunately gathering the data for Oregon was not possible at this time, due to the distributed layout of population information for the state (and only that state). So let's remove it.

In [119]:
```
del city_pop_by_state["OREGON"]
try:
    city_pop_by_state["OREGON"]
except KeyError as e:
    print("KeyError:", e)
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-119-ded924c0280f> in <module>
----> 1 del city_pop_by_state["OREGON"]
      2 try:
      3     city_pop_by_state["OREGON"]
      4 except KeyError as e:
      5     print("KeyError:", e)

KeyError: 'OREGON'
```

As we can see it throws an error when referencing Oregon, so it is no longer in the dictionary.

Now that we have done this, let's move on to the simulation.

The Simulation

To run the simulation, we need to do the following:

1) Get a random graph for all of the cities, with the number of nodes being proportional to the population size

2) Get a simulation running on just one of these random graphs

3) Write a flight feature -- Select a random node in all the random graphs, and connect them all for one timestep

4) Find starting points of infection based on John's Hopkins University's publicly available data

5) Run the simulation

6) Evaluation

5) Run the simulation

6) Evaluation

1. Setting up the basic graph

The basic graph of the simulation is a cumulation of many individual random graphs. But since it takes some time to create random graphs for all of the cities, let's just create them once and save them. That way we can simply load in the basic graph at the beginning of the simulation.

1.1 Making the random graphs

The first step is straightforward; making the graphs. Here we use the configuration model in order to resemble community structure, where each node represents 1,000 people.

The code below may take a while.

In [10]:
```
city_graphs_by_state = {}
for state in city_pop_by_state:
    print("Now doing:", state)
    city_graphs_by_state[state] = []
    for city in city_pop_by_state[state]:
        n = int(city[1]/1000)
        if n <= 1:
            continue
        for i in range(9):
            try:
                sequence = nx.random_powerlaw_tree_sequence(n, tries=10**i)
                break
            except:
                continue
        G = nx.configuration_model(sequence)
        city_graphs_by_state[state].append((city[0], G))
```

```
Now doing: ALABAMA
Now doing: ALASKA
Now doing: ARIZONA
Now doing: ARKANSAS
Now doing: CALIFORNIA
Now doing: COLORADO
Now doing: CONNECTICUT
Now doing: DELAWARE
Now doing: FLORIDA
Now doing: GEORGIA
```

```
Now doing: OHIO
Now doing: OKLAHOMA
Now doing: PENNSYLVANIA
Now doing: RHODE ISLAND
Now doing: SOUTH CAROLINA
Now doing: SOUTH DAKOTA
Now doing: TENNESSEE
Now doing: TEXAS
Now doing: UTAH
Now doing: VERMONT
Now doing: VIRGINIA
Now doing: WASHINGTON
Now doing: WEST VIRGINIA
Now doing: WISCONSIN
Now doing: WYOMING
```

For each one thousand people in a city, we add one node. Because of this, there is a chance that some states won't have any cities represented as a graph. Let's see if this is true.

In [11]:
```
for state in city_graphs_by_state:
    if len(city_graphs_by_state[state]) <= 1:
        print(city_graphs_by_state[state])
```

```
[('Wilmington', <networkx.classes.multigraph.MultiGraph object at 0x000001E292DC7828>)]
[('Burlington', <networkx.classes.multigraph.MultiGraph object at 0x000001E293BF5320>)]
```

There are only two states for which we only have 1 city represented, however there are no states for which we have 0 cities represented. So this seems to work well.

1.2 Cumulating all random graphs

In order to start the simulation with one basic graph, we need to first accumulate all of the random graphs.

What we need to watch out for in this process, is keeping track of which nodes correspond to which city, corresponds to which state. To keep track of this, we need to create another dictionary, just like the ones before, but instead of storing the population or the random graph as the second element of the city-tuple, we will store an ID-range-tuple (min, max) that gives the lowest (inclusive) and the highest node ID (exclusive) that correspond to the particular city.

In [12]:
```
city_nodes_by_state = copy.deepcopy(city_graphs_by_state)
```

The code below may take a while.

In [13]:
```
cumul_graph = nx.MultiGraph() #base graph, needds to be a multigraph for now, once all is accumula
ted will change to normal graph
node_counter = 0
```

1.2 Cumulating all random graphs

In order to start the simulation with one basic graph, we need to first accumulate all of the random graphs.

What we need to watch out for in this process, is keeping track of which nodes correspond to which city, corresponds to which state. To keep track of this, we need to create another dictionary, just like the ones before, but instead of storing the population or the random graph as the second element of the city-tuple, we will store an ID-range-tuple (min, max) that gives the lowest (inclusive) and the highest node ID (exclusive) that correspond to the particular city.

In [12]:
```
city_nodes_by_state = copy.deepcopy(city_graphs_by_state)
```

The code below may take a while.

In [13]:
```
cumul_graph = nx.MultiGraph() #base graph, needds to be a multigraph for now, once all is accumula
ted will change to normal graph
node_counter = 0
for state in city_graphs_by_state:
    print("Now doing:", state)
    for i, city in enumerate(city_graphs_by_state[state]):
        num_cur_city_nodes = city[1].number_of_nodes() #city[1] gives graph for that city
        city_nodes_by_state[state][i] = (city[0], node_counter, node_counter+num_cur_city_nodes)
        cumul_graph = nx.disjoint_union(cumul_graph, city[1])
        node_counter += num_cur_city_nodes
```

```
Now doing: ALABAMA
Now doing: ALASKA
Now doing: ARIZONA
Now doing: ARKANSAS
Now doing: CALIFORNIA
Now doing: COLORADO
Now doing: CONNECTICUT
Now doing: DELAWARE
Now doing: FLORIDA
Now doing: GEORGIA
Now doing: HAWAII
Now doing: IDAHO
Now doing: ILLINOIS
Now doing: INDIANA
Now doing: IOWA
Now doing: KANSAS
Now doing: KENTUCKY
Now doing: LOUISIANA
Now doing: MAINE
Now doing: MARYLAND
Now doing: MASSACHUSETTS
```

```
Now doing: NEW YORK
Now doing: NORTH CAROLINA
Now doing: NORTH DAKOTA
Now doing: OHIO
Now doing: OKLAHOMA
Now doing: PENNSYLVANIA
Now doing: RHODE ISLAND
Now doing: SOUTH CAROLINA
Now doing: SOUTH DAKOTA
Now doing: TENNESSEE
Now doing: TEXAS
Now doing: UTAH
Now doing: VERMONT
Now doing: VIRGINIA
Now doing: WASHINGTON
Now doing: WEST VIRGINIA
Now doing: WISCONSIN
Now doing: WYOMING
```

In [14]:
```
cumul_graph.number_of_nodes()
```

Out[14]:
```
70707
```

In [15]:
```
city_nodes_by_state
```

Out[15]:
```
{'ALABAMA': [('Birmingham', 0, 212),
  ('Dothan', 212, 277),
  ('Huntsville', 277, 457),
  ('Mobile', 457, 652),
  ('Montgomery', 652, 857)],
 'ALASKA': [('Anchorage', 857, 1148),
  ('Bethel', 1148, 1154),
  ('Cordova', 1154, 1156),
  ('Dillingham', 1156, 1158),
  ('Fairbanks', 1158, 1189),
  ('Homer', 1189, 1194),
  ('Juneau', 1194, 1225),
  ('Kenai', 1225, 1232),
  ('Ketchikan', 1232, 1240),
  ('Kodiak', 1240, 1246),
  ('Kotzebue', 1246, 1249),
  ('Nome', 1249, 1252),
  ('Sitka', 1252, 1260),
  ('Unalaska', 1260, 1264),
  ('Valdez', 1264, 1267),
  ('Wrangell', 1267, 1269)],
 'ARIZONA': [('Bullhead City', 1269, 1308),
  ('Wichita Falls', 66303, 66407),
  ('San Angelo', 66407, 66507),
  ('Longview', 66507, 66588),
  ('Harlingen', 66588, 66653)],
 'UTAH': [('Cedar City', 66653, 66683),
  ('Ogden', 66683, 66768),
  ('Provo', 66768, 66884),
  ('St. George', 66884, 66963),
  ('Salt Lake City', 66963, 67157)],
 'VERMONT': [('Burlington', 67157, 67199)],
 'VIRGINIA': [('Charlottesville', 67199, 67244),
  ('Lynchburg', 67244, 67309),
  ('Newport News', 67309, 67489),
  ('Norfolk', 67489, 67734),
  ('Richmond', 67734, 67944),
  ('Roanoke', 67944, 68038)],
 'WASHINGTON': [('Bellingham', 68038, 68128),
  ('Pasco', 68128, 68202),
  ('Seattle', 68202, 68946),
  ('Spokane', 68946, 69165),
  ('Walla Walla', 69165, 69197),
  ('Wenatchee', 69197, 69231),
  ('Yakima', 69231, 69324)],
 'WEST VIRGINIA': [('Charleston', 69324, 69371),
  ('Huntington', 69371, 69417),
  ('Morgantown', 69417, 69447),
  ('Clarksburg', 69447, 69462)],
 'WISCONSIN': [('Appleton', 69462, 69532),
  ('Eau Claire', 69532, 69593),
  ('Green Bay', 69593, 69695),
  ('La Crosse', 69695, 69746),
  ('Madison', 69746, 69954),
  ('Milwaukee', 69954, 70550),
  ('Mosinee', 70550, 70554),
  ('Rhinelander', 70554, 70561)],
 'WYOMING': [('Casper', 70561, 70616),
  ('Cody', 70616, 70625),
  ('Gillette', 70625, 70654),
  ('Laramie', 70654, 70684),
  ('Rock Springs', 70684, 70707)]}
```

You may notice that these are all multi graphs. Since we do not need them or the cumulative graph to be multigraphs, let's turn the cumulative graph into a simple graph. We can do this, because none of the edges use the multi-graph features, meaning we are only freeing up space when doing this.

In [16]:
```
simp_graph = nx.Graph(cumul_graph)
```

cumulative graph into a simple graph. We can do this, because none of the edges use the multi-graph features, meaning we are only freeing up space when doing this.

In [16]:
```
simp_graph = nx.Graph(cumul_graph)
```

In [17]:
```
simp_graph.number_of_nodes()
```

Out[17]: 70707

Now that we've accumulated all of the graphs, let's save the cumulated graph. Let's also save the city_nodes_by_state dictionary, as re-running all of the code would take a while. It may also be a good idea to free up the space that the city_graphs_by_state dictionary along with all of the graph objects are taking up.

In [18]:
```
#save graph
nx.write_gml(simp_graph, "base_graph.gml")
nx.write_gml(cumul_graph, "base_multi_graph.gml")
```

In [63]:
```
json_dict = json.dumps(city_nodes_by_state)
with open("city_nodes_by_state.json", 'w') as f:
    f.write(json_dict)
```

In [362]:
```
#free space up
del city_graphs_by_state
```

With that we are done with part 1!

2. Writing the basic simulation

The basic simulation should have the following parts:

1) Read in base graph and give each node a label of either S, I, or R (Susceptible, Infected, or Recovered)

2) Start out with certain number of infected people

3) Perform timesteps, in each there's a probability (depending on R_0) for transmission, and to recover, nodes have to stay infected for a given number of days

In [14]:
```
def basic_simulation(graph, num_base_infected = 3, ro = 2.0, recover_time = 10, timesteps = 100):
    # ***** SETUP ***** #
    #set up beginning states -> all nodes have state 'S'
    num_nodes = graph.number_of_nodes()
    num_edges = graph.number_of_edges()
```


2. Writing the basic simulation

The basic simulation should have the following parts:

1) Read in base graph and give each node a label of either S, I, or R (Susceptible, Infected, or Recovered)

2) Start out with certain number of infected people

3) Perform timesteps, in each there's a probability (depending on R_0) for transmission, and to recover, nodes have to stay infected for a given number of days

In [14]:
```
def basic_simulation(graph, num_base_infected = 3, ro = 2.0, recover_time = 10, timesteps = 100):
    # ***** SETUP ***** #
    #set up beginning states -> all nodes have state 'S'
    num_nodes = graph.number_of_nodes()
    num_edges = graph.number_of_edges()
    node_list = list(graph.nodes)
    vals = ['S']*num_nodes
    states = dict(zip(graph.nodes, vals))
    infected = {}

    mean_degree = 2*num_edges/num_nodes
    p = mean_degree/ro
    if p > 1:
        p = 1.0

    #next, randomly chose <num_base_infected> number of nodes, and change their state to 'I'
    origin_infected = np.random.choice(node_list, num_base_infected, replace=False)
    for node in origin_infected:
        states[node] = 'I'
        infected[node] = recover_time #number of steps until an infected node is recovered

    num_infected_over_time = [num_base_infected]

    nx.set_node_attributes

    # ***** SIMULATE ***** #
    for day in range(timesteps):
        #go through each infected node, check if their recovery time is 0, if yes, remove them fro
m the dict and add them to the recovered list
        newly_recovered = []
        newly_infected = []
        for inf_node in infected:
            infected[inf_node] -= 1
            if infected[inf_node] == 0:
```

```
    for node in origin_infected:
        states[node] = 'I'
        infected[node] = recover_time #number of steps until an infected node is recovered

    num_infected_over_time = [num_base_infected]

    nx.set_node_attributes

    # ***** SIMULATE ***** #
    for day in range(timesteps):
        #go through each infected node, check if their recovery time is 0, if yes, remove them fro
m the dict and add them to the recovered list
        newly_recovered = []
        newly_infected = []
        for inf_node in infected:
            infected[inf_node] -= 1
            if infected[inf_node] == 0:
                #Node is recovered!
                states[inf_node] = 'R'
                newly_recovered.append(inf_node)
                continue
            for neighbor in graph.neighbors(inf_node):
#                 print(states[neighbor])
                if states[neighbor] == 'S':
                    if np.random.rand() < p:
                        states[neighbor] = 'I'
                        newly_infected.append(neighbor)

        for rec_node in newly_recovered:
            del infected[rec_node]

        for inf_node in newly_infected:
            infected[inf_node] = recover_time

        num_infected_over_time.append(len(infected))
    return num_infected_over_time, num_nodes
```

Since we haven't yet set up connections between the cities on the main graph, let's make the basic simulation run on just one city. Using Birmingham as an example.

In [29]:
```
n = int(212237/1000) #population from Birmingham, divided by 1000
sequence = nx.random_powerlaw_tree_sequence(n, tries=500000)
test_graph = nx.configuration_model(sequence)
```

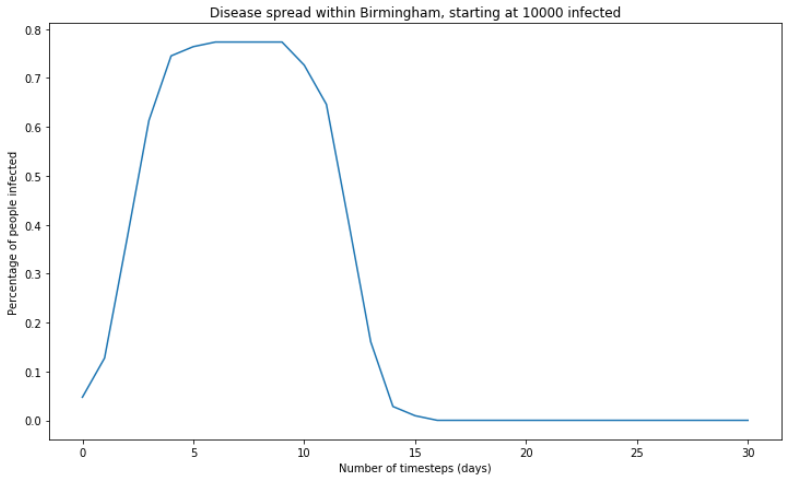
In [32]:
```
infs, pop = basic_simulation(test_graph, num_base_infected = 10, timesteps = 30)
```

Since we haven't yet set up connections between the cities on the main graph, let's make the basic simulation run on just one city. Using Birmingham as an example.

In [29]:
```
n = int(212237/1000) #population from Birmingham, divided by 1000
sequence = nx.random_powerlaw_tree_sequence(n, tries=500000)
test_graph = nx.configuration_model(sequence)
```

In [32]:
```
infs, pop = basic_simulation(test_graph, num_base_infected = 10, timesteps = 30)
```

In [38]:
```
fig,ax = plt.subplots(figsize=(12,7))
ax.plot([inf/pop for inf in infs])
ax.set_title("Disease spread within Birmingham, starting at 10000 infected")
ax.set_xlabel("Number of timesteps (days)")
ax.set_ylabel("Percentage of people infected")
plt.show()
```



3. Adding flights to the Simulation

3. Adding flights to the Simulation

Since we would like to examine the impact of airtravel on the spread of the virus, we need a way to simulate flights. The way that we will do that in this case is as follows. At each timestep:

1) Choose a random node within each city

2) Randomly choose a certain number of other cities to connect the chosen node to (for all chosen nodes)

3) Remove all created edges

The tricky part here is correctly referencing nodes based on their city.

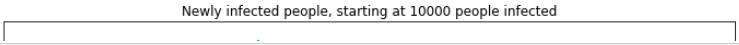
In [52]:
```
base_graph = nx.read_gml("base_graph.gml")
with open('city_nodes_by_state.json', 'r') as f:
    city_nodes_by_state = json.load(f)
```

In [61]:
```
def get_new_edges(proportion_fly):
    nodes_to_connect = []

    for state in city_nodes_by_state:
        for city in city_nodes_by_state[state]:
            #city gives name, min, max
#             print(city)
            num_city_nodes = city[2]-city[1]
            num_flights = int(num_city_nodes*proportion_fly)
            flying_nodes = [str(x) for x in np.random.choice(range(city[1],city[2]), num_flights,
replace = False)]
            if len(flying_nodes) >= 1:
                nodes_to_connect.append(flying_nodes)
#     print(nodes_to_connect)
    new_edges = []
    num_connectors = len(nodes_to_connect)
    for i in range(num_connectors):
        for cur_node in nodes_to_connect[i]:
            #first choose a city to connect to, then choose a node within that city to connect to
            city_index = np.random.randint(0, num_connectors)
            if city_index == i:
                if city_index+1 == num_connectors:
                    city_index -= 1
                else:
                    city_index += 1
            city = nodes_to_connect[city_index]
                    --
            city = nodes_to_connect[city_index]

            #now get the node within that city to connect to
            connecting_node = np.random.choice(city)
            new_edges.append((cur_node, connecting_node))
    return new_edges

def simulation(graph, num_base_infected = 100, ro = 2.0, recover_time = 10, proportion_fly = 0.1,
timesteps = 100):
    # ***** SETUP ***** #
    #set up beginning states -> all nodes have state 'S'
    num_nodes = graph.number_of_nodes()
    num_edges = graph.number_of_edges()
    node_list = list(graph.nodes)
    vals = ['S']*num_nodes
    states = dict(zip(graph.nodes, vals))
    infected = {}

    mean_degree = 2*num_edges/num_nodes
    p = mean_degree/ro
    if p > 1:
        p = 1.0

    #next, randomly chose <num_base_infected> number of nodes, and change their state to 'I'
    origin_infected = np.random.choice(node_list, num_base_infected, replace=False)
    for node in origin_infected:
        states[node] = 'I'
        infected[node] = recover_time #number of steps until an infected node is recovered

    new_infected_over_time = [num_base_infected]

    # ***** SIMULATE ***** #

    for day in range(timesteps):
        new_edges = get_new_edges(proportion_fly)
        graph.add_edges_from(new_edges)
        #go through each infected node, check if their recovery time is 0, if yes, remove them fro
m the dict and add them to the recovered list
        newly_recovered = []
        newly_infected = []
        for inf_node in infected:
            infected[inf_node] -= 1
            if infected[inf_node] == 0:
                #Node is recovered!
                states[inf_node] = 'R'
                newly_recovered.append(inf_node)
```

```
    for day in range(timesteps):
        new_edges = get_new_edges(proportion_fly)
        graph.add_edges_from(new_edges)
        #go through each infected node, check if their recovery time is 0, if yes, remove them fro
m the dict and add them to the recovered list
        newly_recovered = []
        newly_infected = []
        for inf_node in infected:
            infected[inf_node] -= 1
            if infected[inf_node] == 0:
                #Node is recovered!
                states[inf_node] = 'R'
                newly_recovered.append(inf_node)
                continue
            for neighbor in graph.neighbors(inf_node):
#                 print(states[neighbor])
                if states[neighbor] == 'S':
                    if np.random.rand() < p:
                        states[neighbor] = 'I'
                        newly_infected.append(neighbor)

        for rec_node in newly_recovered:
            del infected[rec_node]

        for inf_node in newly_infected:
            infected[inf_node] = recover_time

        new_infected_over_time.append(len(newly_infected))
        graph.remove_edges_from(new_edges)
    return new_infected_over_time, num_nodes
```

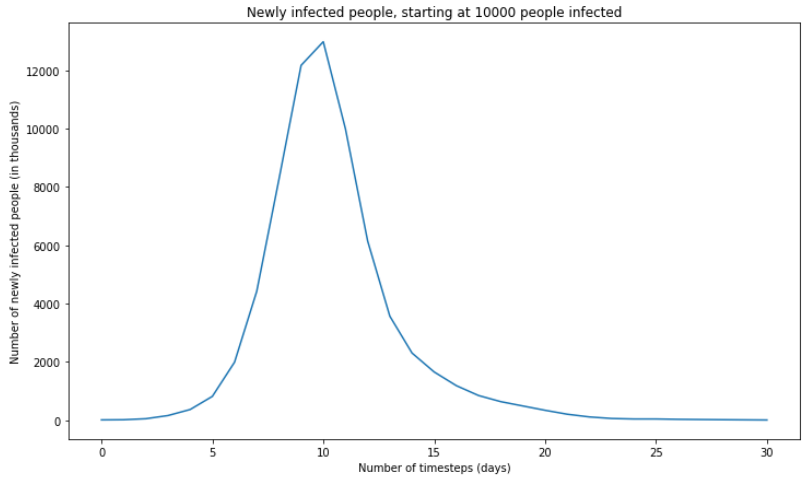
In [62]:
```
base_infected = 10
infs, pop = simulation(copy.deepcopy(base_graph), num_base_infected=base_infected, timesteps = 30)
```

In [63]:
```
fig,ax = plt.subplots(figsize=(12,7))
ax.plot(infs)
ax.set_title("Newly infected people, starting at " +  str(base_infected*1000) + " people infecte
d")
ax.set_xlabel("Number of timesteps (days)")
ax.set_ylabel("Number of newly infected people (in thousands)")
plt.show()
```



In [62]:
```
base_infected = 10
infs, pop = simulation(copy.deepcopy(base_graph), num_base_infected=base_infected, timesteps = 30)
```

In [63]:
```
fig,ax = plt.subplots(figsize=(12,7))
ax.plot(infs)
ax.set_title("Newly infected people, starting at " +  str(base_infected*1000) + " people infecte
d")
ax.set_xlabel("Number of timesteps (days)")
ax.set_ylabel("Number of newly infected people (in thousands)")
plt.show()
```



4. Finding starting points of infection

To find the starting points, we need to read in the data, and then figure out which cities have around 1000 or more cases. This will allow us to place infected nodes in the correct cities, thus simulating more accurately the spread of COVID-19.

4.1 Reading in the data

4. Finding starting points of infection

To find the starting points, we need to read in the data, and then figure out which cities have around 1000 or more cases. This will allow us to place infected nodes in the correct cities, thus simulating more accurately the spread of COVID-19.

4.1 Reading in the data

In [66]:
```
#The link below was attained by going through JHU's COVID-19 visualization github.
file_url = 'https://raw.githubusercontent.com/govex/COVID-19/master/data_tables/JHU_USCountymap/df_Counties2020.csv'
df_us = pd.read_csv(file_url)
df_us.drop(labels = 'Unnamed: 0', axis=1, inplace = True)
df_us.head()
```

Out[66]:

	Countyname	ST_Name	FIPS	ST_ID	dt	Confirmed	Deaths
0	NaN	American Samoa	60	NaN	2020-01-22	0	0.0
1	NaN	Guam	66	NaN	2020-01-22	0	0.0
2	NaN	Northern Mariana Islands	69	NaN	2020-01-22	0	0.0
3	NaN	Puerto Rico	72	NaN	2020-01-22	0	0.0
4	NaN	Virgin Islands	78	NaN	2020-01-22	0	0.0

4.2 Data wrangling

Keep only the data for the states we have represented in our simulation. And for later purposes, add a day and month column.

In [67]:
```
#Correctly capitalizing all the state names to match the JHU data
states = [string.capwords(state) for state in city_nodes_by_state]
```

In [68]:
```
df_us = df_us[df_us["ST_Name"].isin(states)]
```

In [69]:
```
dtIndex = pd.DatetimeIndex(df_us['dt'])
df_us["day"] = dtIndex.day
df_us["month"] = dtIndex.month
```

4.3 Finding cities to start infection

To determine where to start the virus, we will look at the number of confirmed cases on the 21st of March, the day that the US close


```
df_us["day"] = dtIndex.day
df_us["month"] = dtIndex.month
```


4.3 Finding cities to start infection

To determine where to start the virus, we will look at the number of confirmed cases on the 21st of March, the day that the US close its international borders.

In [70]:
```
cases = df_us.loc[(df_us["dt"] == "2020-03-21") & (df_us["Confirmed"] > 0)]

sig_cases = [num_cases for num_cases in cases["Confirmed"] if num_cases>=800]
cases[cases["Confirmed"].isin(sig_cases)]
```

Out[70]:

	Countyname	ST_Name	FIPS	ST_ID	dt	Confirmed	Deaths	day	month
193789	Nassau	New York	36059	36.0	2020-03-21	1234	4.0	21	3
193790	New York	New York	36061	36.0	2020-03-21	7530	45.0	21	3
193819	Westchester	New York	36119	36.0	2020-03-21	1387	0.0	21	3
194901	King	Washington	53033	53.0	2020-03-21	934	74.0	21	3

The above shows the coounties with more than 800 confirmed cases as of March 21st.

From this we need to find out which cities are in those counties. So let's look at the cities in New York, and go through by hand. This is possible since there are only 4 total counties to look at.

In [71]:
```
city_nodes_by_state["NEW YORK"]
```

Out[71]:
```
[['Albany', 37304, 37401],
 ['Binghamton', 37401, 37448],
 ['Buffalo', 37448, 37709],
 ['Elmira', 37709, 37738],
 ['Ithaca', 37738, 37768],
 ['New York', 37768, 45943],
 ['Newburgh', 45943, 45971],
 ['Niagara Falls', 45971, 46021],
 ['Plattsburgh', 46021, 46040],
 ['Rochester', 46040, 46250],
 ['Syracuse', 46250, 46395],
 ['Watertown', 46395, 46422],
 ['White Plains', 46422, 46478]]
```

In [72]:
```
city_nodes_by_state["WASHINGTON"]
```

```
['Ithaca', 37738, 37768],
['New York', 37768, 45943],
['Newburgh', 45943, 45971],
['Niagara Falls', 45971, 46021],
['Plattsburgh', 46021, 46040],
['Rochester', 46040, 46250],
['Syracuse', 46250, 46395],
['Watertown', 46395, 46422],
['White Plains', 46422, 46478]]
```

In [72]:
```
city_nodes_by_state["WASHINGTON"]
```

Out[72]:
```
[['Bellingham', 68038, 68128],
 ['Pasco', 68128, 68202],
 ['Seattle', 68202, 68946],
 ['Spokane', 68946, 69165],
 ['Walla Walla', 69165, 69197],
 ['Wenatchee', 69197, 69231],
 ['Yakima', 69231, 69324]]
```

In [73]:
```
starter_cities_by_state  = {"NEW YORK": [("New York", 9), ("White Plains", 1)], "WASHINGTON": [("S
eattle", 1)]}
```

By going through the cities one by one, we find that New York City (New York county), White Plains (Westchester county) and Seattle (King county) will be our starter cities. Since we do not have a city in Nassau county, but New York City is very close to Nassau county, and we cannot simply overlook 1234 confirmed cases in the county, we will add an extra infected node to New York City.

Since one node represents a thousand people in our simulation, we will start the simulation with 9 infected nodes in New York City ($7530 + 1234 \approx 9000$), and 1 infected node for both White Plains ($1387 \approx 1000$) and Seattle ($934 \approx 1000$).

5. Running the simulation

To run the simulation, we need to first set it up to appropriately take in the starter infections, and collect relevant information.

5.1 Writing final simulation code

In [138]:
```
def choose_nodes_by_city(city, num_choices):
    #given a city (represented by [name, min_node, max_node]) and a number of nodes to choose,
    #  return a random <num_choices> nodes of the particular city
    return [str(x) for x in np.random.choice(range(city[1],city[2]), num_choices, replace = Fals
e)]

def get_new_edges(proportion_fly, city_nodes_by_state):
    nodes to connect = []
```


5. Running the simulation

To run the simulation, we need to first set it up to appropriately take in the starter infections, and collect relevant information.

5.1 Writing final simulation code

In [138]:
```
def choose_nodes_by_city(city, num_choices):
    #given a city (represented by [name, min_node, max_node]) and a number of nodes to choose,
    #  return a random <num_choices> nodes of the particular city
    return [str(x) for x in np.random.choice(range(city[1],city[2]), num_choices, replace = Fals
e)]

def get_new_edges(proportion_fly, city_nodes_by_state):
    nodes_to_connect = []
    for state in city_nodes_by_state:
        for city in city_nodes_by_state[state]:
            #city gives name, min, max
#             print(city)
            num_city_nodes = city[2]-city[1]
            num_flights = int(num_city_nodes*proportion_fly)
            flying_nodes = choose_nodes_by_city(city, num_flights)
            if len(flying_nodes) >= 1:
                nodes_to_connect.append(flying_nodes)

    new_edges = []
    num_connectors = len(nodes_to_connect)
    for i in range(num_connectors):
        for cur_node in nodes_to_connect[i]:
            #first choose a city to connect to, then choose a node within that city to connect to
            city_index = np.random.randint(0, num_connectors)
            if city_index == i:
                if city_index+1 == num_connectors:
                    city_index -= 1
                else:
                    city_index += 1
            city = nodes_to_connect[city_index]

            #now get the node within that city to connect to
            connecting_node = np.random.choice(city)
            new_edges.append((cur_node, connecting_node))
    return new_edges

def final_simulation(graph, city_infections_by_state, city_nodes_by_state, ro = 2.0, recover_time
= 10, proportion_fly = 0.1, timesteps = 100):
    # ***** SETUP *****
```

```
            #now get the node within that city to connect to
            connecting_node = np.random.choice(city)
            new_edges.append((cur_node, connecting_node))
    return new_edges

def final_simulation(graph, city_infections_by_state, city_nodes_by_state, ro = 2.0, recover_time
= 10, proportion_fly = 0.1, timesteps = 100):
    # ***** SETUP ***** #

    #Basic variables
    num_nodes = graph.number_of_nodes()
    num_edges = graph.number_of_edges()
    node_list = list(graph.nodes)
    mean_degree = 2*num_edges/num_nodes

    #R0 gives the expected number of people infected per infected person person, so to get a prob
ability of infection,
    #  we take the average degree and divide it by R0.
    p = ro/mean_degree
    if p > 1:
        p = 1.0

    #set up beginning states -> all nodes have state 'S', no nodes are infected
    vals = ['S']*num_nodes
    states = dict(zip(graph.nodes, vals))
    infected = {}

    #Override nodes based on starting infections
    for state in city_infections_by_state:
        for city in city_infections_by_state[state]:
            #find city in list of cities of current state
            state_city_index = 0
            for i,candidate_city in enumerate(city_nodes_by_state[state]):
                if candidate_city[0] == city[0]:
                    state_city_index = i
                    break

            for inf_node in choose_nodes_by_city(city_nodes_by_state[state][i], city[1]):
                states[inf_node] = 'I'
                infected[inf_node] = recover_time

    cases_over_time = [len(infected)]

    # ***** SIMULATE ***** #
    for day in range(timesteps):
        new_edges = get_new_edges(proportion_fly, city_nodes_by_state)
```

```
    cases_over_time = [len(infected)]

    # ***** SIMULATE ***** #
    for day in range(timesteps):
        new_edges = get_new_edges(proportion_fly, city_nodes_by_state)
        graph.add_edges_from(new_edges)
        #go through each infected node, check if their recovery time is 0, if yes, remove them fr
om the dict and add them to the recovered list
        newly_recovered = []
        newly_infected = []
        for inf_node in infected:
            infected[inf_node] -= 1
            if infected[inf_node] == 0:
                #Node is recovered!
                states[inf_node] = 'R'
                newly_recovered.append(inf_node)
                continue
            for neighbor in graph.neighbors(inf_node):
                if states[neighbor] == 'S':
                    if np.random.rand() < p:
                        states[neighbor] = 'I'
                        newly_infected.append(neighbor)

        for rec_node in newly_recovered:
            del infected[rec_node]

        for inf_node in newly_infected:
            infected[inf_node] = recover_time

        cases_over_time.append(len(infected))
        graph.remove_edges_from(new_edges)
    return cases_over_time
```

5.2 Running final simulation

In [199]:
```
all_sims = []
for r_naught in [2.0, 4.0, 6.0]:
    infs = final_simulation(base_graph, starter_cities_by_state, city_nodes_by_state, timesteps=3
0)
    all_sims.append((r_naught, infs*1000))
```

6. Evaluation

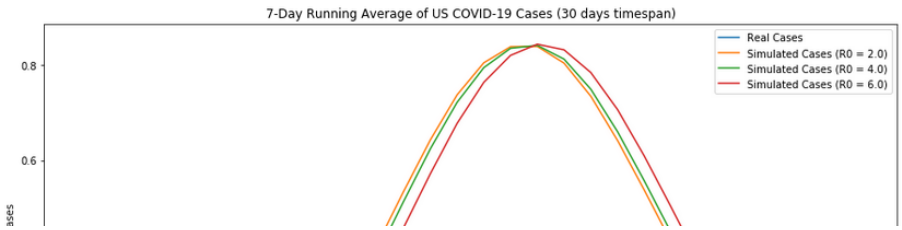
To evaluate our simulation, we need to compare our simulated data to the real world data. The function below should do just that.

```
    all_plot_labels.append("Real Cases")

    for r_naught, sim_data in all_sim_data:
        ra_sims = []
        all_days_sims = []
        for days_since_outbreak, date in enumerate(all_dates):
            sims_day = sim_data[days_since_outbreak]
            all_days_sims.append(sims_day)
            ra = []
            for i in range(ra_length):
                try:
                    ra.append(all_days_sims[days_since_outbreak-i])
                except:
                    pass
            ra_sims.append(np.mean(ra))
        all_data.append(ra_sims)
        all_plot_labels.append("Simulated Cases (R0 = {:.1f})".format(r_naught))

    ### Plotting Stuff ###

    fig,ax = plt.subplots(1,1,figsize=(15,8))

    title = "{:d}-Day Running Average of US COVID-19 Cases (30 days timespan)".format(ra_length)
#, state_string[:-2])

    ax.set_title(title)
    for i, ra in enumerate(all_data[:]):
        if i == 0:
            cplot = ax.plot([cases/US_population for cases in ra], label = all_plot_labels[i])
        else:
            cplot = ax.plot([cases/simulation_population for cases in ra], label = all_plot_label
s[i])
#         if i <= 1:
#             cycord = cplot[0].get_ydata()[-1]
#             cxcord = cplot[0].get_xdata()[-1]
#             ax.annotate('{:.0f} Cases'.format(cycord),
#                     xy=(cxcord, cycord),
#                     xytext=(0, 5),  # 5 points vertical offset
#                     textcoords="offset points",
#                     ha='center', va='bottom')
#     splot = ax.plot([cases for cases in ra_sims], color = 'red', label = "Simulated Cases")
#     sycord = splot[0].get_ydata()[-1]
#     sxcord = splot[0].get_xdata()[-1]
#     ax.annotate('{:.0f} Cases'.format(sycord),
#             xy=(sxcord, sycord),
#             xytext=(0, 5),  # 5 points vertical offset
```


```
    ax.set_title(title)
    for i, ra in enumerate(all_data[:]):
        if i == 0:
            cplot = ax.plot([cases/US_population for cases in ra], label = all_plot_labels[i])
        else:
            cplot = ax.plot([cases/simulation_population for cases in ra], label = all_plot_label
s[i])
#         if i <= 1:
#             cycord = cplot[0].get_ydata()[-1]
#             cxcord = cplot[0].get_xdata()[-1]
#             ax.annotate('{:.0f} Cases'.format(cycord),
#                     xy=(cxcord, cycord),
#                     xytext=(0, 5),  # 5 points vertical offset
#                     textcoords="offset points",
#                     ha='center', va='bottom')
#     splot = ax.plot([cases for cases in ra_sims], color = 'red', label = "Simulated Cases")
#     sycord = splot[0].get_ydata()[-1]
#     sxcord = splot[0].get_xdata()[-1]
#     ax.annotate('{:.0f} Cases'.format(sycord),
#             xy=(sxcord, sycord),
#             xytext=(0, 5),  # 5 points vertical offset
#             textcoords="offset points",
#             ha='center', va='bottom')
```


```
    ax.set_ylabel("Number of cases")
    ax.set_xlabel("Time (Months)")
    ax.set_xticks(label_indices)
    ax.set_xticklabels(label_names)
    ax.legend()

    plt.show()
```

In [203]:
```
plot_spread(all_sims, df_us, states = states, start_date = "2020-03-21", end_date = "2020-04-19")
```



```
    ax.set_title(title)
    for i, ra in enumerate(all_data[:]):
        if i == 0:
            cplot = ax.plot([cases/US_population for cases in ra], label = all_plot_labels[i])
        else:
            cplot = ax.plot([cases/simulation_population for cases in ra], label = all_plot_label
s[i])
#         if i <= 1:
#             cycord = cplot[0].get_ydata()[-1]
#             cxcord = cplot[0].get_xdata()[-1]
#             ax.annotate('{:.0f} Cases'.format(cycord),
#                     xy=(cxcord, cycord),
#                     xytext=(0, 5),  # 5 points vertical offset
#                     textcoords="offset points",
#                     ha='center', va='bottom')
#     splot = ax.plot([cases for cases in ra_sims], color = 'red', label = "Simulated Cases")
#     sycord = splot[0].get_ydata()[-1]
#     sxcord = splot[0].get_xdata()[-1]
#     ax.annotate('{:.0f} Cases'.format(sycord),
#             xy=(sxcord, sycord),
#             xytext=(0, 5),  # 5 points vertical offset
#             textcoords="offset points",
#             ha='center', va='bottom')

    ax.set_ylabel("Number of cases")
    ax.set_xlabel("Time (Months)")
    ax.set_xticks(label_indices)
    ax.set_xticklabels(label_names)
    ax.legend()

    plt.show()
```

In [203]:
```
plot_spread(all_sims, df_us, states = states, start_date = "2020-03-21", end_date = "2020-04-19")
```