Disease_Simulation_With_Airport-Cities_In_USA

May 17, 2020

[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 #for saving dictionary
import networkx as nx #for simulation
import copy #to make deep copies
```

1 Simulating USA COVID-19 Spread using Airport-cities

Author: Christoph Uhl, christophuhl07@gmail.com

The goal of this notebook is to simulate the COVID-19 virus's spread in the United States using random graphs to represent cities with primary airports (meaning an annual number of more than 10,000 enplanements).

This notebook includes the following sections:

1) Data gathering – Get names of cities with primary airports and their population
2) Simulation – Write the simulation using the configuration model to produce random graphs for cities and 1 node for every 1,000 people
3) Analysis – Analyzing graphs, and Conclusion
4) Conclusion – Final words on analysis, and future work

1.1 1. Data Gathering

1) Get list of US cities with primary airports

2) Get population of those cities

1.1.1 1.1 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.

[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.

[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.

[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 city), 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 territories
                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 city names, so choosing the right one is tough
                new_city = new_city[:new_city.find('/')]

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

[5]:
```
airport_cities_by_state
```

[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',
 'Mammoth Lakes',
 'Monterey',
 'Oakland',
 'Ontario',
 'Orange County',
 'Palm Springs',
 'Redding',
 'Sacramento',
 'San Diego',
 'San Francisco',
 'San Jose',
 'San Luis Obispo',
 'Santa Barbara',
 'Santa Maria',
 'Santa Rosa',
 'Stockton'],
'COLORADO': ['Aspen',
 'Colorado Springs',
 'Denver',
 'Durango',
 'Eagle',
 'Grand Junction',
 'Gunnison',
 'Hayden',
 'Montrose'],
'CONNECTICUT': ['Hartford', 'New Haven'],
'DELAWARE': ['Wilmington'],
'FLORIDA': ['Daytona Beach',
 'Fort Lauderdale',
 'Fort Myers',
 'Fort Walton Beach',
 'Gainesville',
 'Jacksonville',
```

```
 'Key West',
 'Melbourne',
 'Miami',
 'Orlando',
 'Panama City Beach',
 'Pensacola',
 'Punta Gorda',
 'Sanford',
 'Sarasota',
 'St. Augustine',
 'St. Petersburg',
 'Tallahassee',
 'Tampa',
 'West Palm Beach'],
'GEORGIA': ['Albany',
 'Atlanta',
 'Augusta',
 'Brunswick',
 'Columbus',
 'Savannah',
 'Valdosta'],
'HAWAII': ['Hilo',
 'Honolulu',
 'Kahului',
 'Kailua-Kona',
 'Kaunakakai',
 'Lanai City',
 'Lihue'],
'IDAHO': ['Boise',
 'Idaho Falls',
 'Lewiston',
 'Pocatello',
 'Sun Valley',
 'Twin Falls'],
'ILLINOIS': ['Belleville',
 'Bloomington',
 'Champaign',
 'Chicago',
 'Chicago',
 'Marion',
 'Moline',
 'Peoria',
 'Quincy',
 'Rockford',
 'Springfield'],
'INDIANA': ['Evansville', 'Fort Wayne', 'Indianapolis', 'South Bend'],
'IOWA': ['Cedar Rapids', 'Des Moines', 'Dubuque', 'Sioux City', 'Waterloo'],
```

```
'KANSAS': ['Garden City', 'Manhattan', 'Topeka', 'Wichita'],
'KENTUCKY': ['Cincinnati', 'Lexington', 'Louisville', 'Owensboro', 'Paducah'],
'LOUISIANA': ['Alexandria',
 'Baton Rouge',
 'Lafayette',
 'Lake Charles',
 'Monroe',
 'New Orleans',
 'Shreveport'],
'MAINE': ['Bangor', 'Portland', 'Presque Isle', 'Rockland'],
'MARYLAND': ['Baltimore', 'Hagerstown', 'Salisbury'],
'MASSACHUSETTS': ['Boston',
 'Hyannis',
 'Nantucket',
 'Provincetown',
 'Vineyard Haven',
 'Worcester'],
'MICHIGAN': ['Alpena',
 'Detroit',
 'Escanaba',
 'Flint',
 'Grand Rapids',
 'Hancock',
 'Iron Mountain',
 'Kalamazoo',
 'Lansing',
 'Marquette',
 'Muskegon',
 'Pellston',
 'Saginaw',
 'Sault Ste. Marie',
 'Traverse City'],
'MINNESOTA': ['Bemidji',
 'Brainerd',
 'Duluth',
 'Hibbing',
 'International Falls',
 'Minneapolis',
 'Rochester',
 'St. Cloud'],
'MISSISSIPPI': ['Columbus', 'Gulfport ', 'Jackson'],
'MISSOURI': ['Columbia', 'Joplin', 'Kansas City', 'Springfield', 'St. Louis'],
'MONTANA': ['Billings',
 'Bozeman',
 'Butte',
 'Great Falls',
 'Helena',
```

```
 'Kalispell',
 'Missoula',
 'Sidney'],
'NEBRASKA': ['Grand Island', 'Lincoln', 'Omaha'],
'NEVADA': ['Boulder City', 'Elko', 'Las Vegas', 'North Las Vegas', 'Reno'],
'NEW HAMPSHIRE': ['Lebanon', 'Manchester', 'Portsmouth'],
'NEW JERSEY': ['Atlantic City', 'Trenton', 'New York '],
'NEW MEXICO': ['Albuquerque', 'Hobbs', 'Roswell', 'Santa Fe'],
'NEW YORK': ['Albany',
 'Binghamton',
 'Buffalo',
 'Elmira',
 'Farmingdale',
 'Islip',
 'Ithaca',
 'New York',
 'New York',
 'Newburgh',
 'Niagara Falls',
 'Plattsburgh',
 'Rochester',
 'Syracuse',
 'Watertown',
 'White Plains'],
'NORTH CAROLINA': ['Asheville',
 'Charlotte',
 'Concord',
 'Fayetteville',
 'Greensboro',
 'Greenville',
 'Jacksonville',
 'New Bern',
 'Raleigh',
 'Wilmington'],
'NORTH DAKOTA': ['Bismarck',
 'Dickinson',
 'Fargo',
 'Grand Forks',
 'Minot',
 'Williston'],
'OHIO': ['Akron',
 'Cincinnati',
 'Cleveland',
 'Columbus',
 'Columbus',
 'Dayton',
 'Toledo',
```

```
 'Youngstown'],
'OKLAHOMA': ['Lawton', 'Oklahoma City', 'Tulsa'],
'OREGON': ['Eugene', 'Medford', 'North Bend', 'Portland', 'Redmond'],
'PENNSYLVANIA': ['Allentown',
 'Erie',
 'Harrisburg',
 'Latrobe',
 'Philadelphia',
 'Pittsburgh',
 'State College',
 'Wilkes-Barre',
 'Williamsport'],
'RHODE ISLAND': ['Block Island', 'Providence', 'Westerly'],
'SOUTH CAROLINA': ['Charleston',
 'Columbia',
 'Florence',
 'Greenville',
 'Hilton Head',
 'Myrtle Beach'],
'SOUTH DAKOTA': ['Aberdeen', 'Rapid City', 'Sioux Falls'],
'TENNESSEE': ['Chattanooga',
 'Knoxville',
 'Memphis',
 'Nashville',
 'Tri-Cities'],
'TEXAS': ['Abilene',
 'Amarillo',
 'Austin',
 'Beaumont',
 'Brownsville',
 'College Station',
 'Corpus Christi',
 'Dallas',
 'Dallas',
 'El Paso',
 'Harlingen',
 'Houston',
 'Houston',
 'Killeen',
 'Laredo',
 'Longview',
 'Lubbock',
 'McAllen',
 'Midland',
 'San Angelo',
 '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!

1.1.2 1.2 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.

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

[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_name):
            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))
    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, otherwise 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:
        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 doesn'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.findAll("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 are 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 column (if any)
                        #   that is represented with a 'th' tag, rather than a␣
↪'td' tag, so just reduce pop index by 1
                        populations = handle_new_city(populations,␣
↪get_city_population_tuple(alt_city, 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=contender_tag.
↪find("a").attrs['title'])
    return my_dict

city_pop_by_state = get_city_pop_dictionary(airport_cities_by_state)
```

Here's the first 5 states with their cities with airports and population:

[8]:
```
for key in list(city_pop_by_state.keys())[:5]:
    print("{}: {}".format(key, city_pop_by_state[key]))
```

```
ALABAMA: [('Birmingham', 212237), ('Dothan', 65496), ('Huntsville', 180105),
('Mobile', 195111), ('Montgomery', 205764)]
ALASKA: [('Anchorage', 291826), ('Aniak', 501), ('Bethel', 6080), ('Cordova',
2239), ('Dillingham', 2329), ('Fairbanks', 31535), ('Galena', 470), ('Homer',
5003), ('Juneau', 31275), ('Kenai', 7100), ('Ketchikan', 8050), ('Kodiak',
6130), ('Kotzebue', 3201), ('Nome', 3598), ('Sitka', 8881), ("St. Mary's", 507),
('Unalakleet', 688), ('Unalaska', 4376), ('Valdez', 3976), ('Wrangell', 2369)]
ARIZONA: [('Bullhead City', 39540), ('Flagstaff', 65870), ('Mesa', 439041),
('Page', 7247), ('Phoenix', 1445632), ('Tucson', 520116), ('Yuma', 93064)]
ARKANSAS: [('Little Rock', 193524), ('Fort Smith', 88209), ('Fayetteville',
76899), ('Texarkana', 29919)]
CALIFORNIA: [('Arcata', 17231), ('Bakersfield', 347483), ('Burbank', 103340),
```

```
('Fresno', 494665), ('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
Obispo', 45119), ('Santa Barbara', 88410), ('Santa Maria', 99553), ('Santa
Rosa', 167815), ('Stockton', 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.

[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.

1.2 2. 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

1.2.1 2.1 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.

2.1.1 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.

[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: 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: MICHIGAN
Now doing: MINNESOTA
Now doing: MISSISSIPPI
Now doing: MISSOURI
Now doing: MONTANA
Now doing: NEBRASKA
Now doing: NEVADA
Now doing: NEW HAMPSHIRE
Now doing: NEW JERSEY
Now doing: NEW MEXICO
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
```

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.

[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.

2.1.2 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.

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

The code below may take a while.

[13]:
```
cumul_graph = nx.MultiGraph() #base graph, needds to be a multigraph for now,␣
 ↪once all is accumulated 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: MICHIGAN
```

```
Now doing: MINNESOTA
Now doing: MISSISSIPPI
Now doing: MISSOURI
Now doing: MONTANA
Now doing: NEBRASKA
Now doing: NEVADA
Now doing: NEW HAMPSHIRE
Now doing: NEW JERSEY
Now doing: NEW MEXICO
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
```

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

[14]:
```
70707
```

[15]:
```
city_nodes_by_state
```

[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),
 ('Flagstaff', 1308, 1373),
 ('Mesa', 1373, 1812),
 ('Page', 1812, 1819),
 ('Phoenix', 1819, 3264),
 ('Tucson', 3264, 3784),
 ('Yuma', 3784, 3877)],
'ARKANSAS': [('Little Rock', 3877, 4070),
 ('Fort Smith', 4070, 4158),
 ('Fayetteville', 4158, 4234),
 ('Texarkana', 4234, 4263)],
'CALIFORNIA': [('Arcata', 4263, 4280),
 ('Bakersfield', 4280, 4627),
 ('Burbank', 4627, 4730),
 ('Fresno', 4730, 5224),
 ('Long Beach', 5224, 5686),
 ('Los Angeles', 5686, 9478),
 ('Mammoth Lakes', 9478, 9486),
 ('Monterey', 9486, 9513),
 ('Oakland', 9513, 9903),
 ('Ontario', 9903, 10066),
 ('Palm Springs', 10066, 10110),
 ('Redding', 10110, 10199),
 ('Sacramento', 10199, 10665),
 ('San Diego', 10665, 11966),
 ('San Francisco', 11966, 12771),
 ('San Jose', 12771, 13716),
 ('San Luis Obispo', 13716, 13761),
 ('Santa Barbara', 13761, 13849),
 ('Santa Maria', 13849, 13948),
 ('Santa Rosa', 13948, 14115),
 ('Stockton', 14115, 14406)],
'COLORADO': [('Aspen', 14406, 14412),
 ('Colorado Springs', 14412, 14857),
 ('Denver', 14857, 15520),
 ('Durango', 15520, 15537),
 ('Eagle', 15537, 15543),
 ('Grand Junction', 15543, 15603),
 ('Gunnison', 15603, 15608),
 ('Montrose', 15608, 15627)],
'CONNECTICUT': [('Hartford', 15627, 15751), ('New Haven', 15751, 15880)],
```

```
'DELAWARE': [('Wilmington', 15880, 15950)],
'FLORIDA': [('Daytona Beach', 15950, 16018),
 ('Fort Lauderdale', 16018, 16200),
 ('Fort Myers', 16200, 16282),
 ('Fort Walton Beach', 16282, 16304),
 ('Gainesville', 16304, 16437),
 ('Jacksonville', 16437, 17340),
 ('Key West', 17340, 17364),
 ('Melbourne', 17364, 17446),
 ('Miami', 17446, 17916),
 ('Orlando', 17916, 18201),
 ('Panama City Beach', 18201, 18214),
 ('Pensacola', 18214, 18266),
 ('Punta Gorda', 18266, 18286),
 ('Sanford', 18286, 18346),
 ('Sarasota', 18346, 18403),
 ('St. Augustine', 18403, 18417),
 ('St. Petersburg', 18417, 18682),
 ('Tampa', 18682, 19074),
 ('West Palm Beach', 19074, 19185)],
'GEORGIA': [('Atlanta', 19185, 19683),
 ('Augusta', 19683, 19879),
 ('Columbus', 19879, 20073),
 ('Savannah', 20073, 20218),
 ('Albany', 20218, 20293),
 ('Valdosta', 20293, 20349)],
'HAWAII': [('Honolulu', 20349, 20686),
 ('Hilo', 20686, 20729),
 ('Kahului', 20729, 20755),
 ('Lihue', 20755, 20761),
 ('Kaunakakai', 20761, 20764),
 ('Lanai City', 20764, 20767)],
'IDAHO': [('Boise', 20767, 20993),
 ('Idaho Falls', 20993, 21054),
 ('Pocatello', 21054, 21109),
 ('Twin Falls', 21109, 21158),
 ('Lewiston', 21158, 21190)],
'ILLINOIS': [('Belleville', 21190, 21234),
 ('Bloomington', 21234, 21310),
 ('Champaign', 21310, 21391),
 ('Chicago', 21391, 24086),
 ('Marion', 24086, 24103),
 ('Moline', 24103, 24146),
 ('Peoria', 24146, 24261),
 ('Quincy', 24261, 24301),
 ('Rockford', 24301, 24453),
 ('Springfield', 24453, 24569)],
```

```
'INDIANA': [('Indianapolis', 24569, 25436),
 ('Fort Wayne', 25436, 25703),
 ('Evansville', 25703, 25820),
 ('South Bend', 25820, 25921)],
'IOWA': [('Cedar Rapids', 25921, 26047),
 ('Des Moines', 26047, 26250),
 ('Dubuque', 26250, 26307),
 ('Sioux City', 26307, 26389),
 ('Waterloo', 26389, 26457)],
'KANSAS': [('Wichita', 26457, 26847),
 ('Topeka', 26847, 26973),
 ('Manhattan', 26973, 27029),
 ('Garden City', 27029, 27055)],
'KENTUCKY': [('Lexington', 27055, 27350),
 ('Louisville', 27350, 27947),
 ('Owensboro', 27947, 28004),
 ('Paducah', 28004, 28029)],
'LOUISIANA': [('Alexandria', 28029, 28076),
 ('Baton Rouge', 28076, 28305),
 ('Lafayette', 28305, 28425),
 ('Lake Charles', 28425, 28496),
 ('Monroe', 28496, 28544),
 ('New Orleans', 28544, 28887),
 ('Shreveport', 28887, 29086)],
'MAINE': [('Portland', 29086, 29152),
 ('Bangor', 29152, 29184),
 ('Presque Isle', 29184, 29193),
 ('Rockland', 29193, 29200)],
'MARYLAND': [('Baltimore', 29200, 29820),
 ('Hagerstown', 29820, 29859),
 ('Salisbury', 29859, 29889)],
'MASSACHUSETTS': [('Boston', 29889, 30506),
 ('Nantucket', 30506, 30516),
 ('Provincetown', 30516, 30518),
 ('Worcester', 30518, 30699)],
'MICHIGAN': [('Alpena', 30699, 30709),
 ('Detroit', 30709, 31422),
 ('Escanaba', 31422, 31434),
 ('Flint', 31434, 31536),
 ('Grand Rapids', 31536, 31724),
 ('Hancock', 31724, 31728),
 ('Iron Mountain', 31728, 31735),
 ('Kalamazoo', 31735, 31809),
 ('Lansing', 31809, 31923),
 ('Marquette', 31923, 31944),
 ('Muskegon', 31944, 31982),
 ('Saginaw', 31982, 32033),
```

```
 ('Sault Ste. Marie', 32033, 32047),
 ('Traverse City', 32047, 32061)],
'MINNESOTA': [('Minneapolis', 32061, 32485),
 ('Rochester', 32485, 32604),
 ('Duluth', 32604, 32691),
 ('St. Cloud', 32691, 32759),
 ('Hibbing', 32759, 32774),
 ('Bemidji', 32774, 32789),
 ('Brainerd', 32789, 32802),
 ('International Falls', 32802, 32807)],
'MISSISSIPPI': [('Columbus', 32807, 32830), ('Jackson', 32830, 33003)],
'MISSOURI': [('Kansas City', 33003, 33490),
 ('St. Louis', 33490, 33804),
 ('Springfield', 33804, 33978),
 ('Columbia', 33978, 34108),
 ('Joplin', 34108, 34161)],
'MONTANA': [('Billings', 34161, 34265),
 ('Bozeman', 34265, 34302),
 ('Butte', 34302, 34335),
 ('Great Falls', 34335, 34393),
 ('Helena', 34393, 34421),
 ('Kalispell', 34421, 34440),
 ('Missoula', 34440, 34506),
 ('Sidney', 34506, 34511)],
'NEBRASKA': [('Omaha', 34511, 34957),
 ('Lincoln', 34957, 35237),
 ('Grand Island', 35237, 35288)],
'NEVADA': [('Boulder City', 35288, 35303),
 ('Elko', 35303, 35321),
 ('Las Vegas', 35321, 35904),
 ('North Las Vegas', 35904, 36120),
 ('Reno', 36120, 36345)],
'NEW HAMPSHIRE': [('Lebanon', 36345, 36358),
 ('Manchester', 36358, 36467),
 ('Portsmouth', 36467, 36487)],
'NEW JERSEY': [('Trenton', 36487, 36571), ('Atlantic City', 36571, 36610)],
'NEW MEXICO': [('Albuquerque', 36610, 37155),
 ('Hobbs', 37155, 37189),
 ('Roswell', 37189, 37237),
 ('Santa Fe', 37237, 37304)],
'NEW YORK': [('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)],
'NORTH CAROLINA': [('Charlotte', 46478, 47350),
 ('Raleigh', 47350, 47819),
 ('Greensboro', 47819, 48113),
 ('Fayetteville', 48113, 48322),
 ('Wilmington', 48322, 48444),
 ('Concord', 48444, 48538),
 ('Greenville', 48538, 48631),
 ('Asheville', 48631, 48723),
 ('Jacksonville', 48723, 48795),
 ('New Bern', 48795, 48825)],
'NORTH DAKOTA': [('Fargo', 48825, 48949),
 ('Bismarck', 48949, 49022),
 ('Grand Forks', 49022, 49078),
 ('Minot', 49078, 49125),
 ('Williston', 49125, 49152),
 ('Dickinson', 49152, 49174)],
'OHIO': [('Akron', 49174, 49372),
 ('Cincinnati', 49372, 49674),
 ('Cleveland', 49674, 50057),
 ('Columbus', 50057, 50949),
 ('Dayton', 50949, 51089),
 ('Toledo', 51089, 51365),
 ('Youngstown', 51365, 51429)],
'OKLAHOMA': [('Oklahoma City', 51429, 52078),
 ('Tulsa', 52078, 52478),
 ('Lawton', 52478, 52570)],
'PENNSYLVANIA': [('Allentown', 52570, 52691),
 ('Erie', 52691, 52788),
 ('Harrisburg', 52788, 52837),
 ('Latrobe', 52837, 52845),
 ('Philadelphia', 52845, 54426),
 ('Pittsburgh', 54426, 54728),
 ('Wilkes-Barre', 54728, 54768),
 ('Williamsport', 54768, 54798)],
'RHODE ISLAND': [('Providence', 54798, 54976), ('Westerly', 54976, 54998)],
'SOUTH CAROLINA': [('Charleston', 54998, 55118),
 ('Columbia', 55118, 55247),
 ('Florence', 55247, 55284),
 ('Greenville', 55284, 55342),
 ('Myrtle Beach', 55342, 55369)],
'SOUTH DAKOTA': [('Sioux Falls', 55369, 55550),
```

```
 ('Rapid City', 55550, 55625),
 ('Aberdeen', 55625, 55653)],
'TENNESSEE': [('Chattanooga', 55653, 55820),
 ('Knoxville', 55820, 55998),
 ('Memphis', 55998, 56644),
 ('Nashville', 56644, 57270)],
'TEXAS': [('Houston', 57270, 59582),
 ('San Antonio', 59582, 61093),
 ('Dallas', 61093, 62434),
 ('Austin', 62434, 63384),
 ('El Paso', 63384, 64067),
 ('Corpus Christi', 64067, 64392),
 ('Laredo', 64392, 64652),
 ('Lubbock', 64652, 64905),
 ('Amarillo', 64905, 65104),
 ('Brownsville', 65104, 65287),
 ('Killeen', 65287, 65432),
 ('McAllen', 65432, 65574),
 ('Waco', 65574, 65710),
 ('Midland', 65710, 65846),
 ('Abilene', 65846, 65967),
 ('Beaumont', 65967, 66086),
 ('College Station', 66086, 66199),
 ('Tyler', 66199, 66303),
 ('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.

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

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

[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.

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

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

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

With that we are done with part 1!

1.2.2 2.2 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

[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 from 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):
```

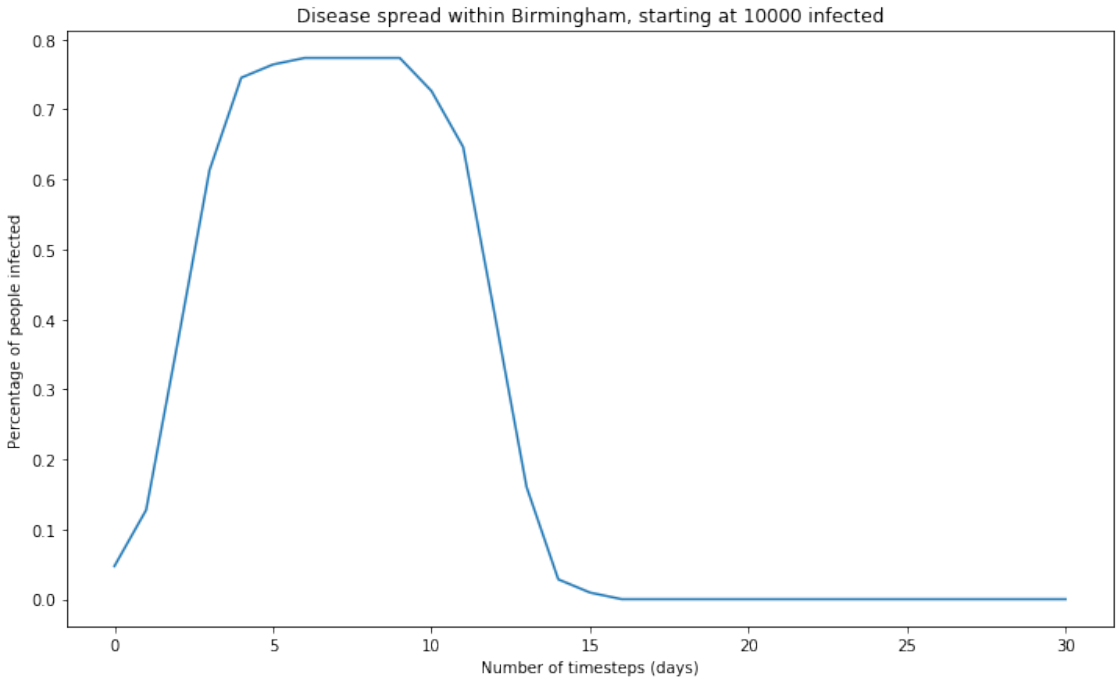
```
#                 print(states[neighbor])
                if states[neighbor] == 'S':
                    if np.random.rand() < p:
                        states[neighbor] = 'I'
                        newly_infected.append(neighbor)

        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.

[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)
```

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

[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()
```



1.2.3 2.3 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.

[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)
```

[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)
```

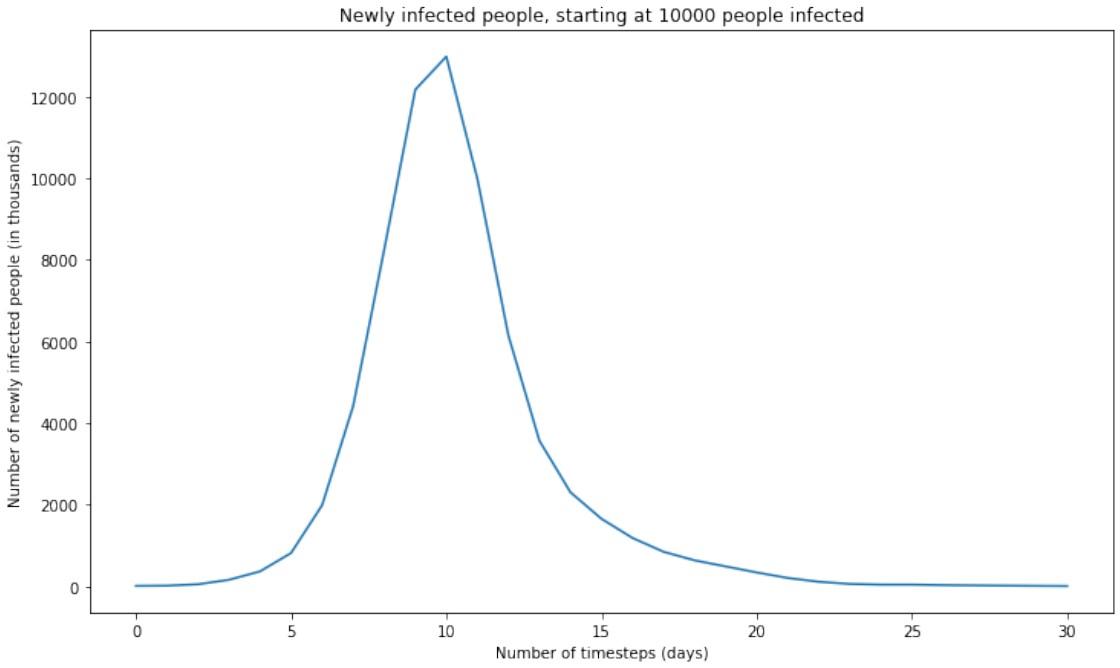
```
            flying_nodes = [str(x) for x in np.random.
 ↪choice(range(city[1],city[2]), num_flights, replace = False)]
            if len(flying_nodes) >= 1:
                nodes_to_connect.append(flying_nodes)
#     print(nodes_to_connect)
    new_edges = []
    num_connectors = len(nodes_to_connect)
    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]

            #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 from 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
```

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

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

```
plt.show()
```



1.2.4 2.4 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.

2.4.1 Reading in the data

[372]:
```
#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()
```

[372]:

	Countyname	ST_Name	FIPS	ST_ID	dt	Confirmed	Deaths	Population	\
0	Autauga	Alabama	1001	1	2020-01-22	0	0	55869	
1	Autauga	Alabama	1001	1	2020-01-23	0	0	55869	
2	Autauga	Alabama	1001	1	2020-01-24	0	0	55869	
3	Autauga	Alabama	1001	1	2020-01-25	0	0	55869	
4	Autauga	Alabama	1001	1	2020-01-26	0	0	55869	

	IncidenceRate	NewCases

```
0            0.0       NaN
1            0.0       0.0
2            0.0       0.0
3            0.0       0.0
4            0.0       0.0
```

2.4.2 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.

[373]:
```
#Correctly capitalizing all the state names to match the JHU data
states = [string.capwords(state) for state in city_nodes_by_state]
```

[374]:
```
df_us = df_us[df_us["ST_Name"].isin(states)]
```

[375]:
```
dtIndex = pd.DatetimeIndex(df_us['dt'])
df_us["day"] = dtIndex.day
df_us["month"] = dtIndex.month
```

2.4.3 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.

[376]:
```
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)]
```

[376]:
```
         Countyname     ST_Name   FIPS  ST_ID          dt  Confirmed  Deaths  \
223591       Nassau    New York  36059     36  2020-03-21       1234       4
223707     New York    New York  36061     36  2020-03-21       7530     180
227071  Westchester    New York  36119     36  2020-03-21       1387       0
356411         King  Washington  53033     53  2020-03-21        934      74

        Population  IncidenceRate  NewCases  day  month
223591     1356924          90.94     480.0   21      3
223707     8336817          90.32    2379.0   21      3
227071      967506         143.36     296.0   21      3
356411     2252782          41.46     141.0   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.

[377]:
```
city_nodes_by_state["NEW YORK"]
```

[377]:
```
[['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]]
```

[378]:
```
city_nodes_by_state["WASHINGTON"]
```

[378]:
```
[['Bellingham', 68038, 68128],
 ['Pasco', 68128, 68202],
 ['Seattle', 68202, 68946],
 ['Spokane', 68946, 69165],
 ['Walla Walla', 69165, 69197],
 ['Wenatchee', 69197, 69231],
 ['Yakima', 69231, 69324]]
```

[379]:
```
starter_cities_by_state  = {"NEW YORK": [("New York", 9), ("White Plains", 1)],␣
 ↪"WASHINGTON": [("Seattle", 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 ($934 \approx 1000$).

1.2.5 2.5 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.

2.5.1 Writing final simulation code

[412]:
```
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 = False)]

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 ***** #

    #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 probability 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)
        graph.add_edges_from(new_edges)
        #go through each infected node, check if their recovery time is 0, if␣
↪yes, remove them from 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)*1000) #mutiply by 1,000 because␣
↪one node indicates 1K people
        graph.remove_edges_from(new_edges)
```

```
    return cases_over_time
```

2.5.2 Running final simulation

[413]:
```
all_sims = []
for r_naught in [0.1, 0.25, 0.5, 1.0, 2.0, 4.0, 6.0]:
    infs = final_simulation(base_graph, starter_cities_by_state,␣
 ↪city_nodes_by_state, ro = r_naught, timesteps=50)
    all_sims.append((r_naught, infs))
```

Now that we have ran the simulation for a number of different R_0s, let's analyze!

1.3 3. Analysis

For the analysis, there are only 2 steps:

1) Make a plotting function, we need a plotting function to compare simulated and real data.

2) Run + Analyze Plots, this is where the analysis takes place

1.3.1 3.1 Plotting function

The plotting function (`plot_spread()`) below takes all the simulation data, all the John's Hopkins University data, a list of states to plot (only works for JHU data), start/end dates, a boolean to indicate whether or not to plot percentage of cases, and a number to indicate how many days to make a rolling-average for (smooths out the data curve). It expects the simulation data to be in form of a list of tuples where the first element gives the simulation's R_0 and the second element gives the number of infected people throughout the timesteps of the simulation.

Please note that the start date should more or less be fixed to the 21st of march, if the simulation data is to line-up with the real data, as the cities and the number of nodes to start with the infection within them are pre-determined based on values from that date.

[414]:
```
def get_end_date(num_days, start_date = "2020-03-21"):
    #given a start date and the number of days, get the end date of period
    #  leverages Covid-19 John's Hopkins University data

    if num_days == -1:
        return df_us['dt'].unique()[-1]

    unique_dates = list(df_us['dt'].unique())
    start_date_index = unique_dates.index(start_date)
    return unique_dates[start_date_index+num_days]
```

[415]:
```
def plot_spread(all_sim_data, pltDf, states = ['All'], start_date =␣
 ↪'2020-03-21', end_date = '', plot_percentage = False, ra_length = 7):
    #plots number of cases over time for specified states

    US_population = 304000000 #approximate population of the states for which␣
 ↪the simulation runs
    simulation_population = base_graph.number_of_nodes()*1000
```

```
    if states[0] != "All":
        pltDf = pltDf[pltDf["ST_Name"].isin(states)]

    all_dates = list(pltDf[pltDf['dt'] >= start_date]['dt'].unique())

    if end_date != '':
        all_dates = all_dates[:all_dates.index(end_date)]
    else:
        all_dates = pltDf['dt'].unique()

    all_days_real = []
    ra_real = []
    label_indices = []
    label_names = []
    num_to_month = {1: 'Jan', 2: 'Feb', 3: 'Mar', 4: 'April 1st', 5: 'May 1st'}

    all_data = []
    all_plot_labels = []
    max_real_val = [-1, -1] #index, val
    for days_since_outbreak, date in enumerate(all_dates):
        tempDf = pltDf[pltDf['dt'] == date]
        cases_day = tempDf['Confirmed'].sum()
        all_days_real.append(cases_day)

        #for ra_length day running average
        ra = []
        for i in range(ra_length):
            try:
                ra.append(all_days_real[days_since_outbreak-i])
            except:
                pass
        actual_ra_dp = np.mean(ra)
        ra_real.append(actual_ra_dp)
        if max_real_val[1] < actual_ra_dp:
            max_real_val[0] = days_since_outbreak
            max_real_val[1] = actual_ra_dp

        #set up label indices
        if tempDf.iloc[0]['day'] == 1:
            label_indices.append(days_since_outbreak)
            label_names.append(num_to_month[tempDf.iloc[0]['month']])

    all_data.append(ra_real)
    all_plot_labels.append("Real Cases")

    max_val = [-1, -1, -1] #sim_data_index, data_point_index, value
```

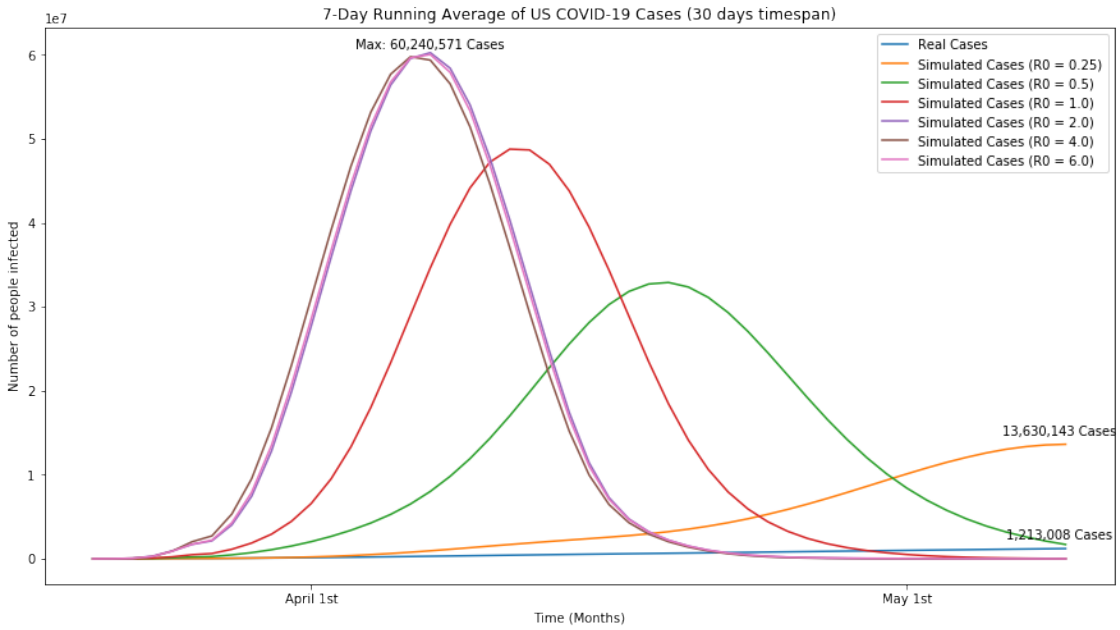
```
    for i, cur_sim_data in enumerate(all_sim_data):
        r_naught = cur_sim_data[0]
        sim_data = cur_sim_data[1]
        ra_sims = []
        all_days_sims = []
        for days_since_outbreak, date in enumerate(all_dates):
            sims_day = sim_data[days_since_outbreak]
            all_days_sims.append(sims_day)
            ra = []
            for ii in range(ra_length):
                try:
                    ra.append(all_days_sims[days_since_outbreak-ii])
                except:
                    pass
            actual_ra_dp = np.mean(ra)
            ra_sims.append(actual_ra_dp)
            if actual_ra_dp > max_val[2]:
                max_val[0] = i
                max_val[1] = days_since_outbreak
                max_val[2] = actual_ra_dp

        all_data.append(ra_sims)
        naught_str = str(r_naught)
        num_decimals = naught_str[::-1].find(".")
        all_plot_labels.append("Simulated Cases (R0 = {r_naught:.
↪{num_decimals}f})".format(r_naught=r_naught, num_decimals=num_decimals))


    ### 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)
    ylabel = "Number of people infected"

    for i, ra in enumerate(all_data[:]):
        if plot_percentage:
            if i == 0:
                ylabel = "Percentage Infected"
                cplot = ax.plot([cases/US_population for cases in ra], label =␣
↪all_plot_labels[i])
                cycord = cplot[0].get_ydata()[max_real_val[0]]
                cxcord = cplot[0].get_xdata()[max_real_val[0]]
                ax.annotate('Max Real Percentage: {:,.4f}'.format(cycord),
                        xy=(cxcord, cycord),
                        xytext=(-5, 5),  # 5 points vertical offset
                        textcoords="offset points",
```

```
                        ha='center', va='bottom')
            else:
                cplot = ax.plot([cases/simulation_population for cases in ra],␣
↪label = all_plot_labels[i])
                if i == max_val[0]+1:
                    cycord = cplot[0].get_ydata()[max_val[1]]
                    cxcord = cplot[0].get_xdata()[max_val[1]]
                    ax.annotate('Max Simulated Percentage: {:,.4f}'.
↪format(cycord),
                                xy=(cxcord, cycord),
                                xytext=(0, 1),  # 5 points vertical offset
                                textcoords="offset points",
                                ha='center', va='bottom')
        else:
            cplot = ax.plot(ra, label = all_plot_labels[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=(-5, 5),  # 5 points vertical offset
                        textcoords="offset points",
                        ha='center', va='bottom')
            if i == max_val[0]+1:
                cycord = cplot[0].get_ydata()[max_val[1]]
                cxcord = cplot[0].get_xdata()[max_val[1]]
                ax.annotate('Max: {:,.0f} Cases'.format(cycord),
                        xy=(cxcord, cycord),
                        xytext=(0, 1),  # 5 points vertical offset
                        textcoords="offset points",
                        ha='center', va='bottom')

    ax.set_title(title)
    ax.set_ylabel(ylabel)
    ax.set_xlabel("Time (Months)")
    ax.set_xticks(label_indices)
    ax.set_xticklabels(label_names)
    ax.legend()

    plt.show()
```

1.3.2 3.2 Plotting and Analyzing

Now let's finally look at some plots!

[416]:
```
plot_spread(all_sims[1:], df_us, states = states, end_date = get_end_date(50),␣
↪plot_percentage = False)
```



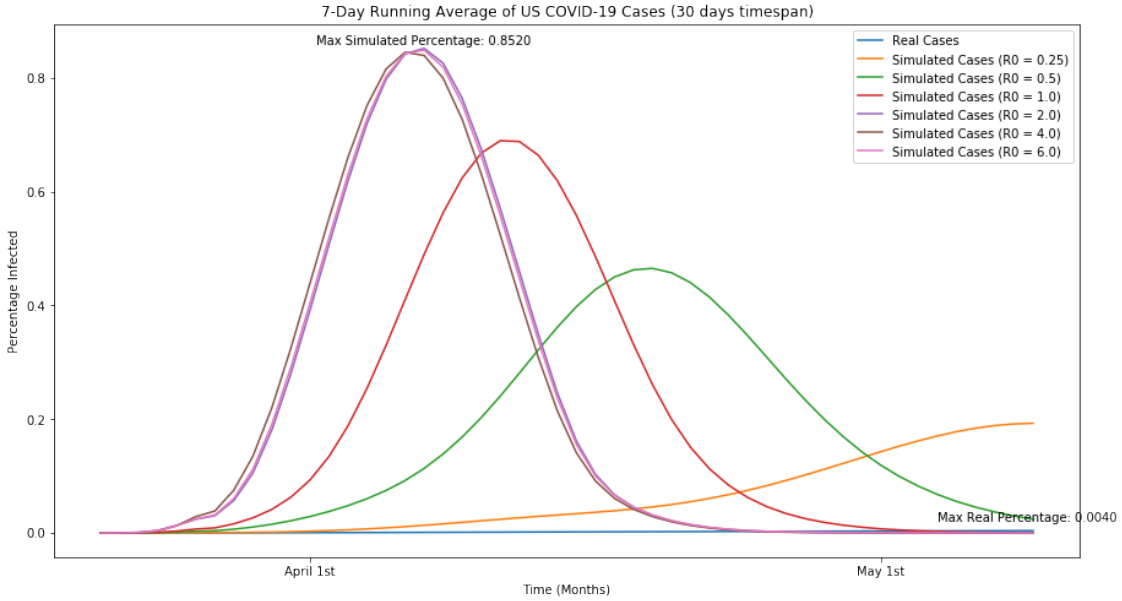
The plot shows the straight number of cases. We can see that the simulated cases are much larger than the real cases, which is likely due to preventative actions taken against the virus. Interestingly enough, when we increase R_0, for $R_0 \geq 2.0$, there are no significant changes visible on the graph. This has to do with how R_0 was used in the simulation, specifically in how we found the probability that an infected node spreads the disease to a neighboring node. We divided R_0 by $\hat{k}$, the average number of neighbors of all nodes in the graph. This works mathematically, since R_0 is defined as the expected number of transmissions per caase (for COVID-19 $R_0 \approx 5.7$). But once we run the simulation with an $R_0 > \hat{k}$ (in our case ≈ 2), this 'probability' grows above 100%, which is not possible, so we set it back to 100%. So essentially, once $R_0 > \hat{k}$, the probability of transmitting the disease to a neighbor is 100%, meaning that there is no more chance involved in getting the disease outside of the network's underlying structure.

The slight variantions in the plots of R0, then, are caused by the random edges that are created every timestep of the simulation, and act as flights between cities. This is because the underlying network is the same for all of the simulations, just those flights are randomized.

So although once we raise R_0 over ≈ 2 the simulated plots don't change anymore, we see that what the simulation predicts is that **a lot** more people should get infected. This means that if the real world was more similar to the simulated world, we would see a lot more cases in the real world. Of course, the two worlds have many differences, but in the terms of how the disease spreads, although it is a 100% transmission rate in the simulation, that isn't too far off from the real world, since an $R_0 = 5.7$ is incredibly high. The way that the simulation represents connections is by personal-contact, but in the real world we do not see this anymore; through the preventative actions taken (quarantine, home-office, forced closure of restaurants), the social network of the current real world should be vastly different from what it normally is, and that is what the simulation's network tries to resemble.

But since the numbers might be different from the percentage calculations, let's what the graph would look like if we plotted the percent population infected.

[417]:
```
plot_spread(all_sims[1:], df_us, states = states, end_date = get_end_date(50),␣
↪plot_percentage = True)
```



This curve is very similar to the previous, except that the percentage of real cases is even smaller in comparison to the simulated curves. This is because in the simulation only represents $\approx$ 70 million people, instead of the 304 million people estimated to live in the states that the simulation runs on (population of mainland US - Oregon + Hawaii + Alaska).

1.4 4. Conclusion

As discussed in the analysis, the simulation differs mainly in the underlying network. Having used the configuration model, we attempted to show how business-as-usual social networks are structured, but since the social network has a different structure under preventative measures for COVID-19, the configuration model might not be the best at simulating the pandemic.

Furthermore, the way that disease spreading was simulated is by calculating a probability of transmission, which indicated how likely a node was to get infected if a neighboring node had the disease – at every timestep. So even if the neighboring node had the disease and it didn't get spread to you at the first timestep, at the second timestep (which in our case indicates the second day), the dice get rolled again. A more accurate simulation would take into the total life-span of a node's infection, and based on this and the estimated R_0, give the probability of transmission.

All in all, this project shows information gathering in the form of a web-scraper traversing a few links and extracting html table information, setting up a base network to run the simulation on using the configuration model, and lastly a simple network epidemiology simulation that returns the number of currently infected people.

Future work should include:

1) Reconsiderations on using the configuration model to resemble city and social structure accurately for COVID-19 times,

2) Making the simulation more accurate, by changing the way the rate of transmission is calculated, or possibly using a different or new method of simulating transmission,

3) Representing fewer people with a node, while adding more nodes to the network, and lastly

4) Representing the United States more holistically in the simulation (only ~70 million people in the simulation)

Aside from these suggestions, if one would like to simulate exactly COVID-19 in the United States, not just with cities with primary airports, it would be absolutely necessary to get information on all of the cities in the country.

Thank you for reading! Please email me with any questions: christophuhl07@gmail.com

[ ]: