CSE 6730 Project 2: Simulation of Virus Spread

Team Number:

2.24

Team Members:

Yue Yu (yueyu@gatech.edu)

Chenjun Tang (ctang90@gatech.edu)

Tianqi Liu (tliu318@gatech.edu)

Link:

https://github.gatech.edu/tliu318/COVID19-Sim

April 28, 2020

Introduction

Modeling and characterizing the spread of diseases play a vital role in guarding the safety of citizens and maintaining public security. Moreover, the development of simulation modeling techniques makes it possible to estimate the spread of contagious diseases as well as conduct risk assessments for various control measures. Recently, the outbreak of the coronavirus disease (COVID-19) poses a great threat to public health, as more than 700,000 cases have been confirmed in more than 100 countries, with more than 30,000 deaths . To tackle the COVID-19 crisis, understanding the spread patterns of COVID-19 is of great significance. For individuals, knowing the spread characteristics of COVID-19 enables them protect themselves better and decrease the risk of COVID-19 infection. For governments, they can make better public policies to offer guidance to people and reduce their anxiety . To conclude, it is urgent and necessary to model and understand the spread of COVID-19 disease.

Motivated by the above, in this project, we aim to simulate the spread of the disease by using several frameworks including cellular automata model, graph-based dynamic model and ordinary differentiation equation-based (ODE) model. Specifically, with the given in- formation about the disease, we employ a simulation model to analyze the dynamics of disease transmission to evaluate the effectiveness of some factors as well as epidemic control measures. Based on the above simulation models, our ultimate goal to investigate several questions: How did the virus get spread from a location to other locations? To what extent do the policies, for instance, social distancing and community quarantine, help controlling the spread of the virus?

Our project consisted of the three part: the first part focuses on Cellular Automata Model, the second part focuses on Graph-based information passing model and the third part concentrates on ODE-based model. More information is on the following pages.

Part_1_CA_simulation

April 28, 2020

[1]:
```
import cv2
import numpy as np
from numpy import random
import matplotlib.pyplot as plt
```

1 Part 1: A cellular automata model

In the first part of this tutorial, we'll apply the concept of cellular automata (CA) to the modeling of the spread of the COVID-19 disease. Cellular au-tomata are algorithms that describe the discrete spatial and temporal evolution of complex systems by applying local deterministic orprobabilistic transformation rules to the cells of a regular lattice.

1.1 Simple SEIR-CA model

In this part, the spread of COVID-19 is modeled in a geographically distributed population. This disease is assumed to be a derivation of the susceptible-exposed-infectious-recovered (SEIR) model. People are first covertly infected and they will only show symptoms after an incubation period. For simplicity, it is assumed that no persons will die during the simulation (related or unrelated to the illness). And a recovered person will never suffer from the illness again.
The simulation function of SEIR can be written as

$$\frac{dS}{dt} = -\beta\frac{SI}{N}$$

$$\frac{dE}{dt} = \beta\frac{SI}{N} - \sigma E$$

$$\frac{dI}{dt} = \sigma E - \gamma I$$

$$\frac{dR}{dt} = \gamma I$$

$$N = S + E + I + R$$

where S, E, I, R stands for the *Susceptible*, *Exposed*, *Infected*, *Recovered* people respectively.

- N is the total number of population,
- β controls how often a susceptible-infected contact results in a new exposure,

- γ is the rate an infected recovers and moves into the resistant phase,
- σ is the rate an exposed person becomes infective

1.2 Conceptual model

World: The world is simulated to be a $m \times n$ grid $G = G(t) \equiv (g_{ij}(t))$ of cells that evolve over discrete time (in days). Every cell of the grid is a person.

Cell: Every cell of G is in one of four possible colors, white, yellow, red or green. Each of them stands for a person in the corresponding states: 1. **White Cell, Susceptible (S)**:This person has never gotten the illness before. If this person comes in close contact with a sick person, he/she is at risk of catching the illness. 2. **Yellow Cell, Exposed (E)**: This person has been infected but is not showing symptoms. This person carries the virus and can infect its neighbor cells. After an incubation period of T days, this person will show symptoms and become Infected. 3. **Red Cell, Infected (I)**: This person has the illness and showing symptoms. 4. **Green Cell, Recovered (R)**: This person had the illness but has fully recovered. He/she cannot become sick again.

These states are associated with the following integers:

[2]:
```
# Health states
SUSCEPTIBLE = 0
EXPOSED = 1
INFECTED = 2
RECOVERED = 3

# Colors
BLACK = (0,0,0)
WHITE = (255,255,255)
RED = (0,0,255)
GREEN = (0,255,0)
YELLOW = (0,255,255)
```

1.3 Simulation of the Virus Spreading

The evolution rule of the SEIR-CA system in each iteration is defined as follows:

1. **Spreading**: The virus spreads from all carriers to their neighbors at rate β, which is uniform and independent for all positions. Healthy person (Susceptible) infected in this progress becomes a virus carrier (Exposed) but will not show symptoms during the incubation period.
2. **Develop**: The incubation period of virus carriers decrease by 1. If the incubation period reaches 0, that person starts to show symptoms (Infected).
3. **Recover**: Infected people will recover at rate γ. A recovered person will not be infected any more.

[3]:
```
# Infection rate,recovery rate and mortality rate
rate_S_E = 0.3 # beta
rate_I_R = 0.01 # gamma
```

According to the assumptions and rules above, the cell is defined with the following code. The incubation period is set to be 14 days.

[4]:
```
class Cell(object):
    def __init__(self, x, y, world, status = SUSCEPTIBLE, incub_period = 14):
        self.x = x
        self.y = y
        self.world = world
        self.status = status
        self.incub_period = incub_period

    def get_neighbors(self):
        neighbors = []
        for nx, ny in [(self.x + 1, self.y), (self.x - 1, self.y), (self.x,␣
↪self.y + 1), (self.x, self.y - 1)]:
            if 0 <= nx < self.world.x_count and 0 <= ny < self.world.y_count:
                neighbors.append((nx, ny))
        return neighbors

    def be_exposed(self):
        if self.status == SUSCEPTIBLE:
            fill_grid(self.x, self.y, YELLOW)
            self.status = EXPOSED
            self.world.exposed_cells.append(self)

    def cell_develop(self):
        if self.status == EXPOSED:
            self.incub_period -= 1
            if self.incub_period == 0:
                self.be_infected()
            self.world.exposed_cells.remove(self)

    def be_infected(self):
        if self.status != INFECTED:
            fill_grid(self.x, self.y, RED)
            self.status = INFECTED
            self.world.infected_cells.append(self)

    def be_recovered(self):
        if self.status != RECOVERED:
            fill_grid(self.x, self.y, GREEN)
            self.status = RECOVERED
            self.world.recovered_count += 1

    def expose_neighbors(self):
        neighbors = self.get_neighbors()
        for nx, ny in neighbors:
```

```
            neighbor_cell = self.world.cells[nx][ny]
            if random.random() < rate_S_E:
                neighbor_cell.be_exposed()
```

The world of the system is defined as below.

[5]:
```
class World(object):
    def __init__(self, x_count, y_count, hospital_capacity = 20):
        self.x_count = x_count
        self.y_count = y_count
        self.population = x_count * y_count
        self.cells = [[Cell(i, j, self) for j in range(y_count)] for i in␣
↪range(x_count)]
        self.infected_cells = []
        self.exposed_cells = []
        self.exposed_count = 0
        self.infected_count = 0
        self.recovered_count = 0

    def update(self):
        self.spread()
        self.develop()
        self.recover()
        self.count()

    def spread(self):
        for i in range(len(self.exposed_cells)):
            self.exposed_cells[i].expose_neighbors()
        for i in range(len(self.infected_cells)):
            self.infected_cells[i].expose_neighbors()

    def develop(self):
        for cell in self.exposed_cells:
            cell.cell_develop()

    def recover(self, rate_I_R = rate_I_R):
        for cell in self.infected_cells:
            if random.random() < rate_I_R:
                cell.be_recovered()
                self.infected_cells.remove(cell)

    def terminate(self):
        # terminates when no one is exposed of infected
        return len(self.exposed_cells) == 0 and len(self.infected_cells) == 0

    def count(self):
        # update the counts
```

```
        self.exposed_count = 0
        self.infected_count = 0
        self.exposed_cells = []
        self.infected_cells = []
        for i in range(self.x_count):
            for j in range(self.y_count):
                if self.cells[i][j].status == EXPOSED:
                    self.exposed_count += 1
                    self.exposed_cells.append(self.cells[i][j])
                elif self.cells[i][j].status == INFECTED:
                    self.infected_count += 1
                    self.infected_cells.append(self.cells[i][j])
```

Two functions are defined to help visualize the simulation.

[6]:
```
def draw_lines():
    for i in range(y_count):
        cv2.line(grid, (0, grid_size * (i + 1)), (width, grid_size * (i + 1)),␣
↪BLACK)
    for j in range(x_count):
        cv2.line(grid, (grid_size * (j + 1), 0), (grid_size * (j + 1), height),␣
↪BLACK)

def fill_grid(x, y, color = YELLOW):
    cv2.rectangle(grid, (x*grid_size+1, y*grid_size+1), ((x+1)*grid_size-1,␣
↪(y+1)*grid_size-1), color, -1)
```

1.4 Simulation with the simple SEIR-CA model

On the "zeroth day" ($t = 0$), the world is full of susceptible people and one of them (randomly picked) gets sick. This state is our initial condition. The simulation will terminate after 200 days. The animation of the simulation will not be shown in the pdf file. It can be found when running the block below.

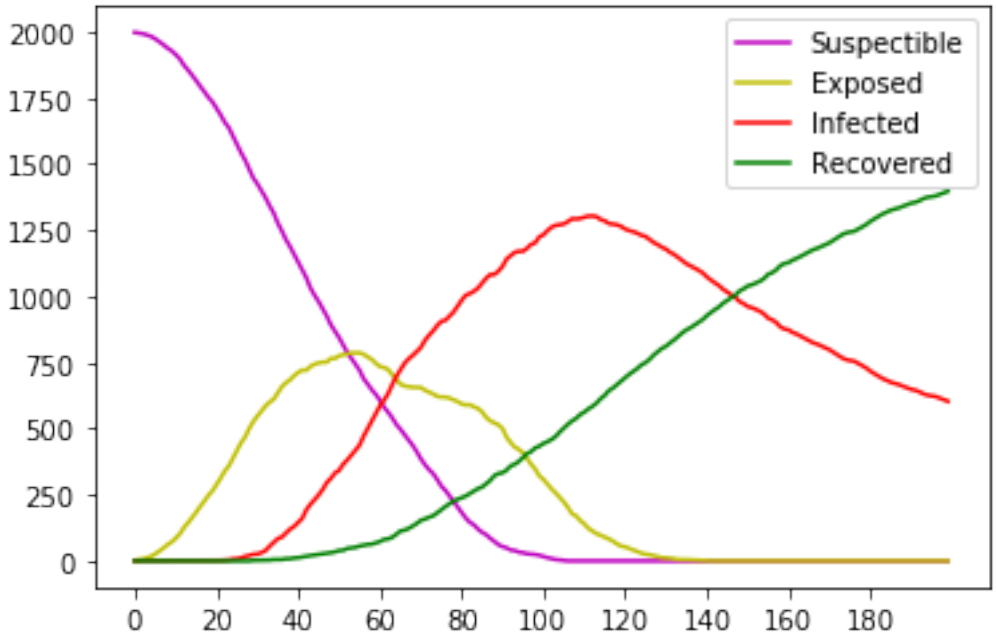
[7]:
```
cv2.namedWindow('grid')

x_count, y_count = 50, 40
grid_size = 20
width, height = x_count * grid_size, y_count * grid_size

grid = np.ones((height, width, 3), dtype = np.uint8)
grid[:,:,] = 255 * grid[:,:,]

draw_lines()
cv2.imshow('grid', grid)
W = World(x_count, y_count)
#Randomly initialize the first infected
```

```
rand_x = random.randint(0, x_count)
rand_y = random.randint(0, y_count)
W.cells[rand_x][rand_y].be_infected()
day = 0
max_iter = 200
S_list = [0] * max_iter
E_list = [0] * max_iter
I_list = [0] * max_iter
R_list = [0] * max_iter


while not W.terminate() and day < max_iter:
    cv2.imshow('grid', grid)
    W.update()
    # exposedNum = len(W.exposed_cells)
    # infectedNum = len(W.infected_cells)
    suspectibleNum = W.population - W.recovered_count - W.exposed_count - W.
 ↪infected_count
    S_list[day], E_list[day], I_list[day], R_list[day] = suspectibleNum, W.
 ↪exposed_count, W.infected_count, W.recovered_count
    day += 1
    if day in range(50, 60):
        print("Day {}, suspectible: {}, exposed: {}, infected:{}, recovered:{}"\
            .format(day, suspectibleNum, W.exposed_count, W.infected_count, W.
 ↪recovered_count))
    if cv2.waitKey(27)&0xFF==ord(' '):
        break
    cv2.waitKey(100)


cv2.imshow('grid', grid)
```

```
Day 50, suspectible: 869, exposed: 765, infected:332, recovered:34
Day 51, suspectible: 844, exposed: 775, infected:342, recovered:39
Day 52, suspectible: 816, exposed: 779, infected:365, recovered:40
Day 53, suspectible: 788, exposed: 784, infected:381, recovered:47
Day 54, suspectible: 766, exposed: 786, infected:399, recovered:49
Day 55, suspectible: 745, exposed: 786, infected:418, recovered:51
Day 56, suspectible: 721, exposed: 786, infected:437, recovered:56
Day 57, suspectible: 689, exposed: 781, infected:471, recovered:59
Day 58, suspectible: 668, exposed: 770, infected:500, recovered:62
Day 59, suspectible: 647, exposed: 762, infected:528, recovered:63
```

[8]:
```
x_axis = range(0, day)
y_axis = range(0, W.population)

plt.xticks(np.arange(0, day, step = day//10))
```

```
plt.plot(x_axis, S_list[:day], 'm', label='Suspectible')
plt.plot(x_axis, E_list[:day], 'y', label='Exposed')
plt.plot(x_axis, I_list[:day], 'r', label='Infected')
plt.plot(x_axis, R_list[:day], 'g', label='Recovered')
plt.legend(loc='upper right')
plt.show()
```



2 More Complex Model: Considering Mortality and Quarantine

The model above is not quite realistic. In real world many people died from COVID-19. Also, people showing symptoms will go to hospital and be quarantined to avoid contacts. Specifically, we add to extra parameter Q which stand for the number of infected population being quarantined in hospital. They also have a higher probability to recover. Then, the ODE formula can be expressed as

$$\frac{dS}{dt} = -\beta \frac{SI}{N}$$

$$\frac{dE}{dt} = \beta \frac{SI}{N} - \sigma E$$

$$\frac{dI}{dt} = \sigma E - (\gamma_1 + q + \mu) I$$

$$\frac{dR}{dt} = \gamma_1 I + \gamma_2 Q$$

$$\frac{dQ}{dt} = qI$$

- q is the proportion of population being self-quarantined.
- γ_1 is the rate an infected recovers and moves into the resistant phase,
- γ_2 is the rate an quarantined recovers and moves into the resistant phase,
- μ is the mortality rate related to disease.

2.1 Conceptual model

Now our conceptual model about cells need to be updated.

Cell: Every cell of G is in one of six possible colors, white, yellow, red, green, blue or black. Each of them stands for a person in the corresponding states: 1. **White Cell, Susceptible (S)**:This person has never gotten the illness before. If this person comes in close contact with a sick person, he/she is at risk of catching the illness. 2. **Yellow Cell, Exposed (E)**: This person has been infected but is not showing symptoms. This person carries the virus and can infect its neighbor cells. After an incubation period of T days, this person will become Infected. 3. **Red Cell, Infected (I)**: This person has the illness and showing symptoms. 4. **Green Cell, Recovered (R)**: This person had the illness but has fully recovered. He/she cannot become sick again. 5. **Blue Cell, Quarantined (Q)**: This person is quarantined in the hospital and will not infect others. 6. **Black Cell, Dead (D)**: This person died from the illness.

These states are associated with the following integers:

[9]:
```
# Health states
SUSCEPTIBLE = 0
EXPOSED = 1
INFECTED = 2
RECOVERED = 3
QUARANTINED = 4
DEAD = 5

#Infection rate and recovery rate
rate_S_E = 0.3 # beta
rate_I_R = 0.01 # gamma_1
rate_H_R = 0.15 # gamma_2
rate_I_D = 0.02 # mu

#Colors
BLACK = (0,0,0)
WHITE = (255,255,255)
RED = (0 ,0,255)
GREEN = (0,255,0)
YELLOW = (0,255,255)
BLUE = (255,153,51)
```

The evolution rules are also modified accordingly.

[10]:
```
class Cell(object):
    def __init__(self, x, y, world, status = SUSCEPTIBLE, incub_period = 14,␣
↪rate_S_E = rate_S_E):
        self.x = x
        self.y = y
        self.world = world
        self.status = status
        self.incub_period = incub_period
        self.rate_S_E = rate_S_E

    def get_neighbors(self):
        # this function returns the neighbor coordinates of the cell
        neighbors = []
        for nx, ny in [(self.x + 1, self.y), (self.x - 1, self.y), (self.x,␣
↪self.y + 1), (self.x, self.y - 1)]:
            if 0 <= nx < self.world.x_count and 0 <= ny < self.world.y_count:
                neighbors.append((nx, ny))
        return neighbors

    def be_exposed(self):
        if self.status == SUSCEPTIBLE:
            fill_grid(self.x, self.y, YELLOW)
            self.status = EXPOSED
            self.world.exposed_cells.append(self)

    def cell_develop(self):
        world = self.world
        if self.status == EXPOSED:
            self.incub_period -= 1
            if self.incub_period == 0:
                if world.hospital_count < world.hospital_capacity:
                    self.go_hospital()
                else:
                    self.be_infected()
                self.world.exposed_cells.remove(self)

    def be_infected(self):
        if self.status != INFECTED and self.status != RECOVERED:
            fill_grid(self.x, self.y, RED)
            self.status = INFECTED
            self.world.infected_cells.append(self)

    def be_recovered(self):
        if self.status != RECOVERED and self.status != SUSCEPTIBLE:
            fill_grid(self.x, self.y, GREEN)
            self.status = RECOVERED
            self.world.recovered_count += 1
```

```
    def go_hospital(self):
        world = self.world
        if world.hospital_count < world.hospital_capacity:
            fill_grid(self.x, self.y, BLUE)
            self.status = QUARANTINED
            world.hospital_cells.append(self)
            world.hospital_count += 1

    def die(self):
        fill_grid(self.x, self.y, BLACK)
        self.status = DEAD
        self.world.death += 1

    def expose_neighbors(self):
        neighbors = self.get_neighbors()
        for nx, ny in neighbors:
            neighbor_cell = self.world.cells[nx][ny]
            if neighbor_cell.status == SUSCEPTIBLE:
                if random.random() < self.rate_S_E:
                    neighbor_cell.be_exposed()


#Create the world of cells
class World(object):
    def __init__(self, x_count, y_count, hospital_capacity = 20, rate_S_E =␣
 ↪rate_S_E, incub_period = 14):
        self.x_count = x_count
        self.y_count = y_count
        self.population = x_count * y_count
        self.hospital_capacity = hospital_capacity
        self.cells = [[Cell(i, j, self, rate_S_E = rate_S_E, incub_period =␣
 ↪incub_period) for j in range(y_count)] for i in range(x_count)]
        self.infected_cells = []
        self.exposed_cells = []
        self.hospital_cells = []
        self.exposed_count = 0
        self.infected_count = 0
        self.hospital_count = 0
        self.recovered_count = 0
        self.death = 0

    def update(self):
        self.spread()
        self.develop()
        self.recover()
        self.fill_hospital()
```

```
        self.count()


    def spread(self):
        for i in range(len(self.exposed_cells)):
            self.exposed_cells[i].expose_neighbors()
        for i in range(len(self.infected_cells)):
            self.infected_cells[i].expose_neighbors()

    def develop(self):
        for cell in self.exposed_cells:
            cell.cell_develop()
        for cell in self.infected_cells:
            if random.random() < rate_I_D:
                cell.die()
                self.infected_cells.remove(cell)

    def recover(self, rate_I_R = rate_I_R):
        for cell in self.infected_cells:
            if random.random() < rate_I_R:
                cell.be_recovered()
                self.infected_cells.remove(cell)

        for cell in self.hospital_cells:
            if random.random() < rate_H_R:
                cell.be_recovered()
                self.hospital_cells.remove(cell)
                self.hospital_count -= 1


    def fill_hospital(self):
        for cell in self.infected_cells:
            if self.hospital_count == self.hospital_capacity:
                break
            cell.go_hospital()
            self.infected_cells.remove(cell)

    def terminate(self):
        return (self.recovered_count + self.hospital_count + self.death) == W.
↪population

    def count(self):
        self.exposed_count = 0
        self.infected_count = 0
        self.exposed_cells = []
        self.infected_cells = []
        for i in range(self.x_count):
```

```
            for j in range(self.y_count):
                if self.cells[i][j].status == EXPOSED:
                    self.exposed_count += 1
                    self.exposed_cells.append(self.cells[i][j])
                elif self.cells[i][j].status == INFECTED:
                    self.infected_count += 1
                    self.infected_cells.append(self.cells[i][j])
        self.hospital_count = len(self.hospital_cells)
```

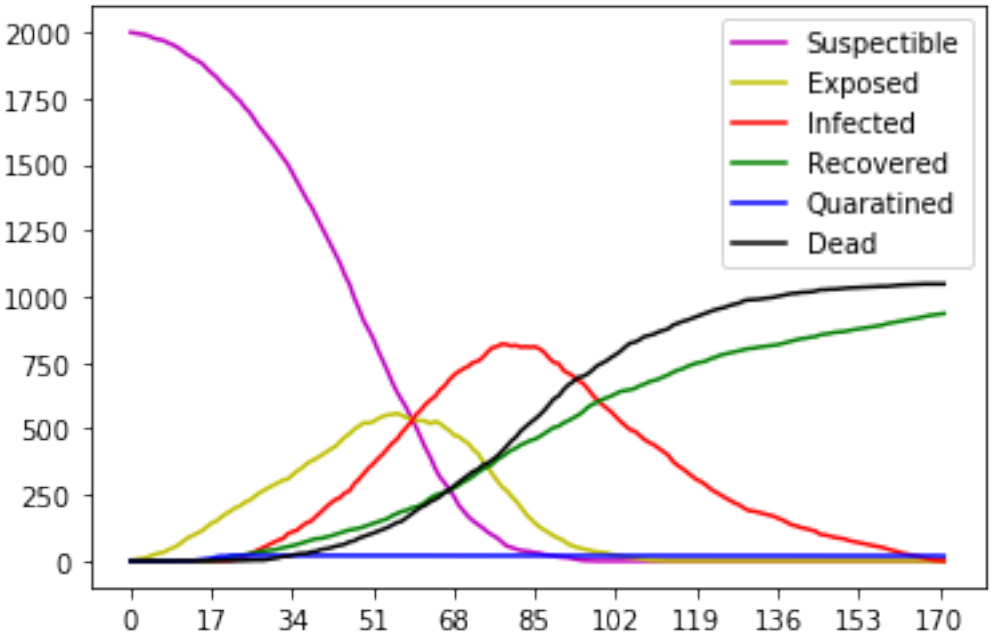
2.2 Simulation with the modified SEIR-CA model

[11]:
```
cv2.namedWindow('grid')

x_count, y_count = 50, 40
grid_size = 20
width, height = x_count * grid_size, y_count * grid_size
#Create a grid for visualization and set all blocks to be white (suspectible)
grid = np.ones((height, width, 3), dtype = np.uint8)
grid[:,:,] = 255 * grid[:,:,]

draw_lines()
cv2.imshow('grid', grid)
W = World(x_count, y_count)
#Randomly initialize the first infected
rand_x = random.randint(0, x_count)
rand_y = random.randint(0, y_count)
W.cells[rand_x][rand_y].be_exposed()
day = 0
max_iter = 200
S_list = [0] * max_iter
E_list = [0] * max_iter
I_list = [0] * max_iter
R_list = [0] * max_iter
H_list = [0] * max_iter
D_list = [0] * max_iter


while not W.terminate() and day < max_iter:
    cv2.imshow('grid', grid)
    W.update()
    suspectibleNum = W.population - W.recovered_count - W.exposed_count - W.
 ↪infected_count - W.hospital_count - W.death
    S_list[day], E_list[day], I_list[day], R_list[day], H_list[day],␣
 ↪D_list[day] = suspectibleNum, W.exposed_count, W.infected_count, W.
 ↪recovered_count, W.hospital_count, W.death
    day += 1
    if day in range(50, 60):
```

```
        print("Day {}, suspectible: {}, exposed: {}, infected:{}, recovered:{},␣
↪hospital: {}, death: {}"\
              .format(day, suspectibleNum, W.exposed_count, W.infected_count, W.
↪recovered_count, W.hospital_count, W.death))
    if cv2.waitKey(27)&0xFF==ord(' '):
        break
    cv2.waitKey(100)

cv2.imshow('grid', grid)
```

```
Day 50, suspectible: 907, exposed: 522, infected:330, recovered:129, hospital:
20, death: 92
Day 51, suspectible: 875, exposed: 521, infected:351, recovered:135, hospital:
20, death: 98
Day 52, suspectible: 840, exposed: 526, infected:367, recovered:142, hospital:
20, death: 105
Day 53, suspectible: 796, exposed: 538, infected:387, recovered:147, hospital:
20, death: 112
Day 54, suspectible: 758, exposed: 546, infected:404, recovered:152, hospital:
20, death: 120
Day 55, suspectible: 718, exposed: 552, infected:424, recovered:160, hospital:
20, death: 126
Day 56, suspectible: 675, exposed: 554, infected:445, recovered:171, hospital:
20, death: 135
Day 57, suspectible: 638, exposed: 558, infected:460, recovered:179, hospital:
20, death: 145
Day 58, suspectible: 608, exposed: 550, infected:485, recovered:187, hospital:
20, death: 150
Day 59, suspectible: 578, exposed: 542, infected:506, recovered:191, hospital:
20, death: 163
```

[12]:
```
x_axis = range(0, day)
y_axis = range(0, W.population)


plt.xticks(np.arange(0, day, step = day//10))
plt.plot(x_axis, S_list[:day], 'm', label='Suspectible')
plt.plot(x_axis, E_list[:day], 'y', label='Exposed')
plt.plot(x_axis, I_list[:day], 'r', label='Infected')
plt.plot(x_axis, R_list[:day], 'g', label='Recovered')
plt.plot(x_axis, H_list[:day], 'b', label='Quaratined')
plt.plot(x_axis, D_list[:day], 'k', label='Dead')
plt.legend(loc='upper right')
plt.show()
```



2.3 Sensitivity study with the modified SEIR-CA model

Now, we want to investigate the influence of different parameters on our model. We focus on the number of infected populaiton under different parameters. The simulation process is defined in the function below for parameter changes.

[13]:
```
def modified_sim(hospital_rate = 0.002, rate_S_E = 0.1, incub_period = 14):
    x_count, y_count = 70, 60
    population = x_count * y_count
    W = World(x_count, y_count, hospital_capacity = hospital_rate * population,␣
↪rate_S_E = rate_S_E, incub_period = incub_period)
    #Randomly initialize the first infected
    rand_x = random.randint(0, x_count)
    rand_y = random.randint(0, y_count)
    W.cells[rand_x][rand_y].be_exposed()
    day = 0
    max_iter = 300
    S_list = [0] * max_iter
    E_list = [0] * max_iter
    I_list = [0] * max_iter
    R_list = [0] * max_iter
    H_list = [0] * max_iter
    D_list = [0] * max_iter

    while day < max_iter:
        W.update()
```

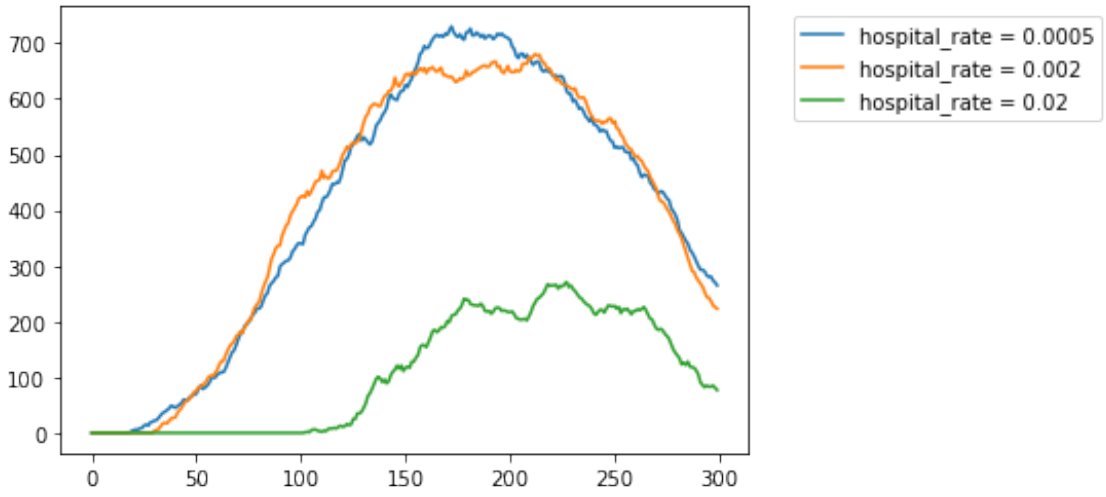
```
        suspectibleNum = W.population - W.recovered_count - W.exposed_count - W.
↪infected_count - W.hospital_count - W.death
        S_list[day], E_list[day], I_list[day], R_list[day], H_list[day],␣
↪D_list[day] = suspectibleNum, W.exposed_count, W.infected_count, W.
↪recovered_count, W.hospital_count, W.death
        day += 1

    return I_list
```

2.4 Hospital Availability Sensitivity Study

Three levels of Hospital Availability were investigated:
- Low: Few infected people are quarantined.
- Normal: Infected people are quarantined in hosiptal.
- High: Temporary hospitals are established in response to the disease.

The result shows that higher hospital availability (higher quarantine level) can effectively reduce the infected number.

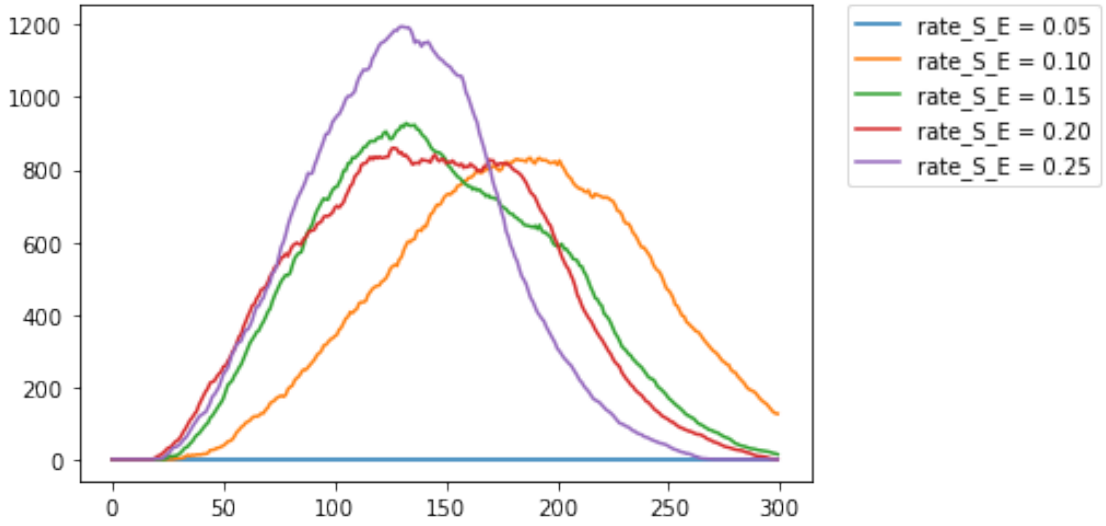
[14]:
```
for h_rate in (0.0005, 0.002, 0.02):
    I_list = modified_sim(hospital_rate = h_rate)
    plt.plot(I_list, label = 'hospital_rate = {}'.format(h_rate))
plt.legend(bbox_to_anchor=(1.05, 1), loc='upper left')
plt.show()
```



2.5 Infection Probability Sensitivity Study

Infection probablity can be different whether people keep social distancing or not.
The result shows that lower infection probability can flatten the infection curve.

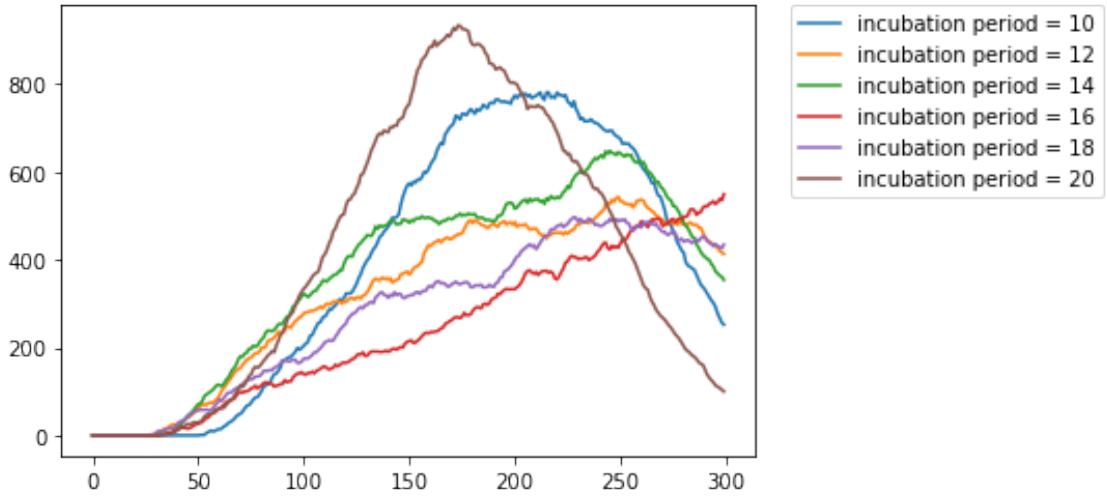
[15]:
```
for r_S_E in np.arange(0.05,0.3,0.05):
    I_list = modified_sim(rate_S_E = r_S_E)
    plt.plot(I_list, label = 'rate_S_E = {}'.format('%.2f'%r_S_E))
plt.legend(bbox_to_anchor=(1.05, 1), loc='upper left', borderaxespad=0.)
plt.show()
```



2.6 Incubation Period Sensitivity Study

We investigate how the average length of the incubation period affacts the spreading of the virus. In the simulation, we assumed all infected person will show symptoms exactly after the Kth days. The result doesn't show significant difference.

[16]:
```
for incub_p in np.arange(10,21,2):
    I_list = modified_sim(incub_period = incub_p)
    plt.plot(I_list, label = 'incubation period = {}'.format(incub_p))
plt.legend(bbox_to_anchor=(1.05, 1), loc='upper left', borderaxespad=0.)
plt.show()
```



2.7 Conclusions

The spreading speed of the virus is: - highly sensitive to the hospital availability - highly sensitive to the infection probability - not quite sensitive to the incubation period

Part_2_Graph_Basd_Simulation

April 28, 2020

Virus Simulation on Graph

Community interaction can be represented as graphs where a node represents a person and an edge represents an interaction. Initially the virus start from one person and he or she is coverly infected. SEIR model is used to implement this. At each iteration, covertly infected persons can spread the virus to their neighbors at a probablity of P. Once shown symptoms, a person (node) becomes isolated (remove all edges associated with it). The spreading of the virus is measured by the final infected populations.

Particularly We want to study the following variables:

- Infection Probability
- Degree of Graph
- Covert Period
- Quarantine Method

Next we investigate two type of graphs to observe if graph with the same average degree always end up the same outcomes:

- Connected Cavemen Graph (Ring of Cliques)
- Lancichinetti–Fortunato–Radicchi Benchmark Graph

Connected Cavemen graph resembles small community that consist of family of size K, and we have strong reason to believe that this is the graph topology to achieve lowest infection at known average degrees. LFR graph is another widely recongized graph representation of inter-community interactions. In this simulation, LFR graph is mainly investigated and tested against different set of parameters.

We also simulated the social distancing, quanrantine, aggressive quanrantine and the covert period affact the spread of the virus.

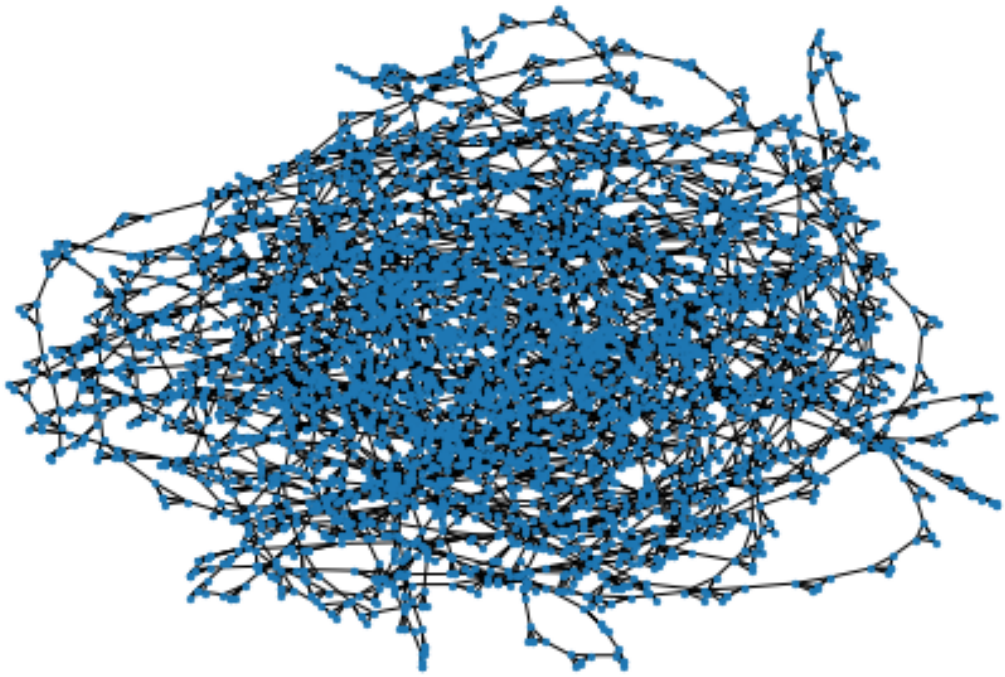
[19]:
```
import networkx as nx
import pandas as pd
import numpy as np
from enum import Enum
import random
from matplotlib import animation, rc
import matplotlib.pyplot as plt
from copy import deepcopy
import ipywidgets as widgets
from ipywidgets import interact, interactive, fixed, interact_manual
```

```
import matplotlib.pyplot as plt
import mplcyberpunk
```

Connected Cavemen Graph

Connected Cavemen Graph is a ring of cliques. Run the following code multiple times to generate a good layout.

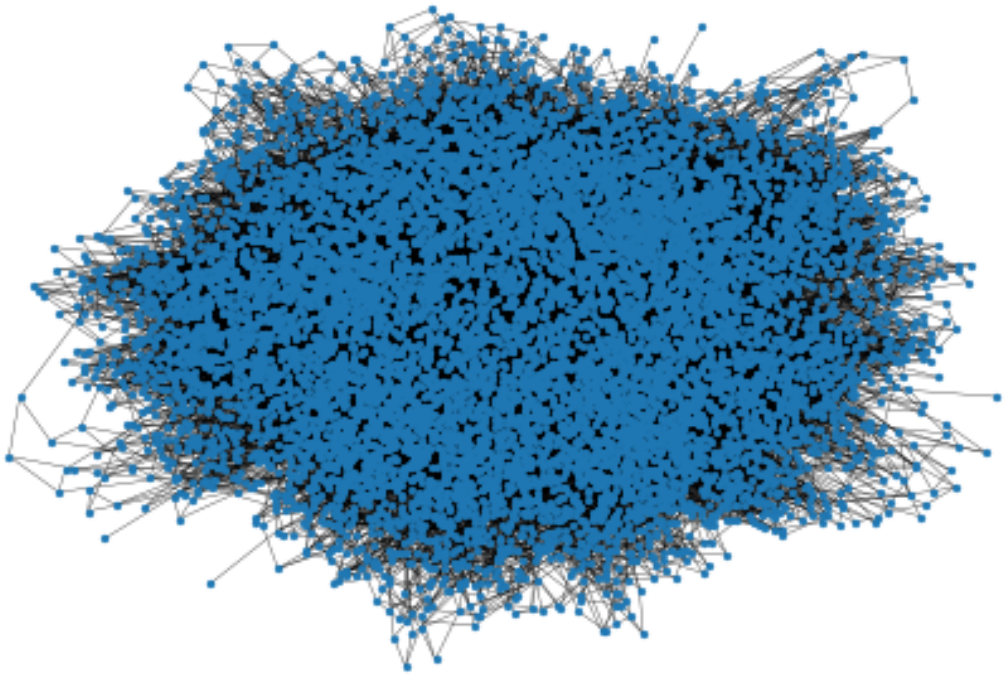
[56]:
```
ccg = nx.generators.community.connected_caveman_graph(1000,5)
ccg_pos = nx.layout.spring_layout(ccg)
nx.draw(ccg, node_size=5, pos=ccg_pos)
```



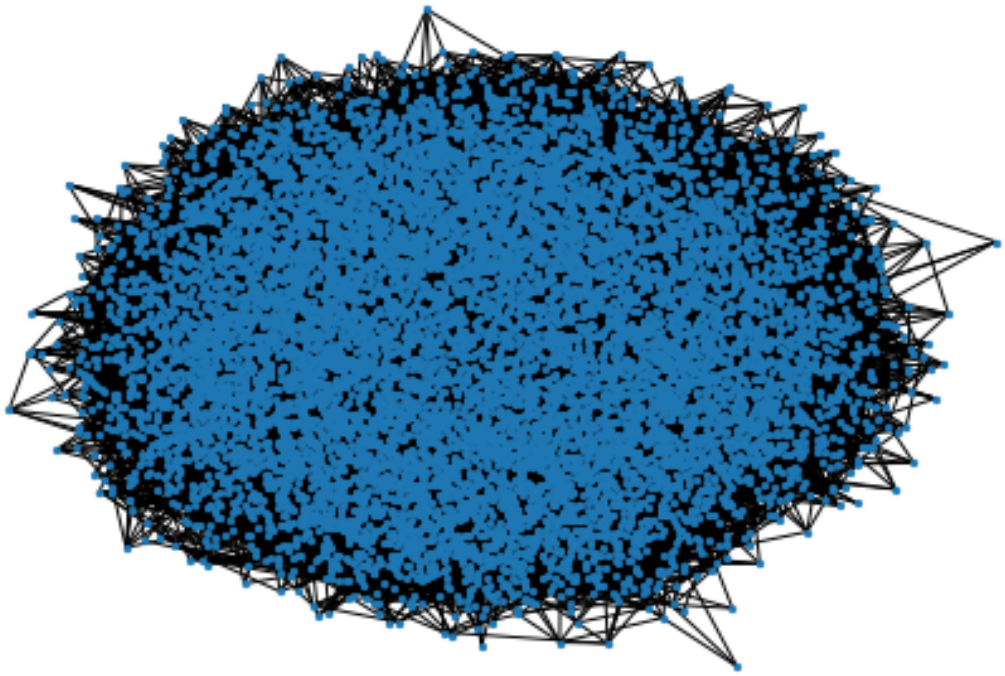
Lancichinetti–Fortunato–Radicchi Benchmark Graph

Connected Cavemen Graph is a ring of cliques. Run the following code multiple times to generate a good layout. A LFR benchmark with clustered layout looks like this:

[8]:
```
lfr5 = nx.generators.community.LFR_benchmark_graph(10000,3,1.1,0.
↪1,average_degree=5,max_degree=10,min_community=50,max_community=80,max_iters=500)
lfr5_pos = nx.layout.spring_layout(lfr5)
nx.draw(lfr5, node_size=5, pos=lfr5_pos, width=0.3)
```



[52]:
```
lfr7 = nx.generators.community.LFR_benchmark_graph(10000,3,1.1,0.
↪1,average_degree=7,max_degree=12,min_community=50,max_community=80,max_iters=500)
lfr7_pos = nx.layout.spring_layout(lfr7)
nx.draw(lfr7, node_size=5, pos=lfr7_pos)
```



Simulation

[14]:
```
class Condition(Enum):
  HEALTHY = 1
  COVERT = 2
  INFECTED = 3

class Quarantine(Enum):
  NONE = 1
  BASIC = 2
  AGGRESSIVE = 3

def initialize(G, covert_days=5, quarantine=Quarantine.BASIC, probability=0.1):
  G.graph['covert_days'] = covert_days
  G.graph['quarantine'] = quarantine
  G.graph['probability'] = probability
  for v in G:
    G.nodes[v]['condition'] = Condition.HEALTHY
    G.nodes[v]['color'] = 'Green'
    G.nodes[v]['covert_countdown'] = covert_days
  G.nodes[0]['condition'] = Condition.COVERT
  G.graph['nInfected'] = [1]
  G.graph['nCovert'] = [0]
```

```
def nextState(G):
  new_covertly = set()
  new_infected = set()
  for v in G.nodes():
    if  G.nodes[v]['condition'] == Condition.INFECTED:
      pass
    elif G.nodes[v]['condition'] == Condition.COVERT:
      if G.nodes[v]['covert_countdown'] <= 0:
        new_infected.add(v)
      else:
        G.nodes[v]['covert_countdown']-=1
    elif  G.nodes[v]['condition'] == Condition.HEALTHY:
      for neighbor in G.neighbors(v):
        if G.nodes[neighbor]['condition'] != Condition.HEALTHY:
          if random.random() < G.graph['probability']:
            new_covertly.add(v)
  G.graph['nInfected'].append(G.graph['nInfected'][-1] + len(new_infected))
  G.graph['nCovert'].append(G.graph['nCovert'][-1] + len(new_covertly))
  return [new_covertly, new_infected]

def updateGraph(G, new_covertly, new_infected):
  for v in new_covertly:
    G.nodes[v]['condition'] = Condition.COVERT
    G.nodes[v]['color'] = 'Yellow'
  for v in new_infected:
    G.nodes[v]['condition'] = Condition.INFECTED
    G.nodes[v]['color'] = 'Red'
    if G.graph['quarantine'] == Quarantine.AGGRESSIVE:
      for neighbor in frozenset(G.neighbors(v)):
        for neighbor2 in frozenset(G.neighbors(neighbor)):
          G.remove_edge(neighbor2, neighbor)
    elif G.graph['quarantine'] == Quarantine.BASIC:
      for neighbor in frozenset(G.neighbors(v)):
        G.remove_edge(neighbor, v)

def update(num):
    ax1.clear()
    [new_covertly, new_infected] = nextState(G)
    updateGraph(G, new_covertly, new_infected)
    colorMap = [G.nodes[v]['color'] for v in G]
    nx.draw(G, pos=pos, ax=ax1, node_size=10, node_color=colorMap)
    ax1.set_title("Day %d: "%(num+2), fontweight="bold")

    ax2.clear()
    ax2.plot(list(range(num+2)),G.graph['nInfected'][:num+2], color='R')
    ax2.plot(list(range(num+2)),G.graph['nCovert'][:num+2], color='Y')
```

```
    ax2.set_xlim(left=0,right=nDays)
    ax2.set_ylim(bottom=0, top=G.number_of_nodes())
    ax2.set_title("Cumulative Cases")

def simulate():
    for i in range(nDays):
        [new_covertly, new_infected] = nextState(G)
        updateGraph(G, new_covertly, new_infected)
```

Simulation on LFR Benchmark (Avg Degree = 5) with 10000 population

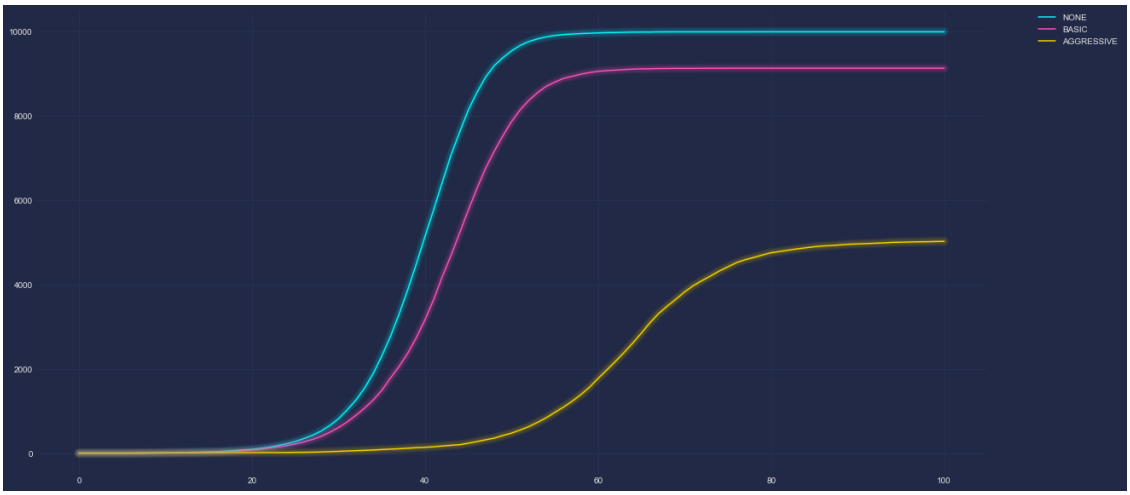
[12]:
```
G = deepcopy(lfr5) # Make a deep copy so that the original graph remain intact
pos = lfr5_pos
```

Quarantine Method Sensitivity Study

Three Quarantine methods were investigated: - None: No quarantine - Basic: When an individual becomes infected, isolate him or her from the community (Remove all edges) - Aggresive: When an individual becomes infected, isolate him/her and his or her neighbors from the community.

The result shows only aggressive quarantine is much more effective than the basic when infection probability is 0.1 and the covert days is 5.

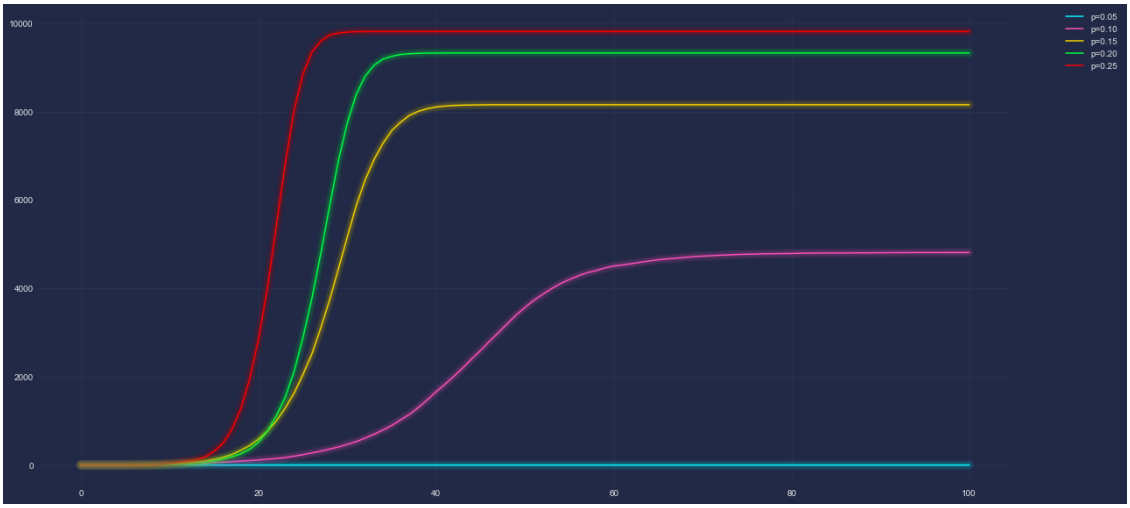
[44]:
```
plt.figure(figsize=(20,10))
for quarantine in Quarantine:
    G = deepcopy(lfr5)
    covert_days = 5
    infection_probability = 0.1
    nDays = 100
    initialize(G, covert_days, quarantine, infection_probability)
    simulate()
    plt.plot(G.graph['nInfected'], label=quarantine.name)

plt.legend(bbox_to_anchor=(1.05, 1), loc='upper left', borderaxespad=0.)
mplcyberpunk.make_lines_glow()
plt.show()
```



Infection Probability Sensitivity Study

We define the infection probability as the chance of contracting the virus per interaction with a infected person.

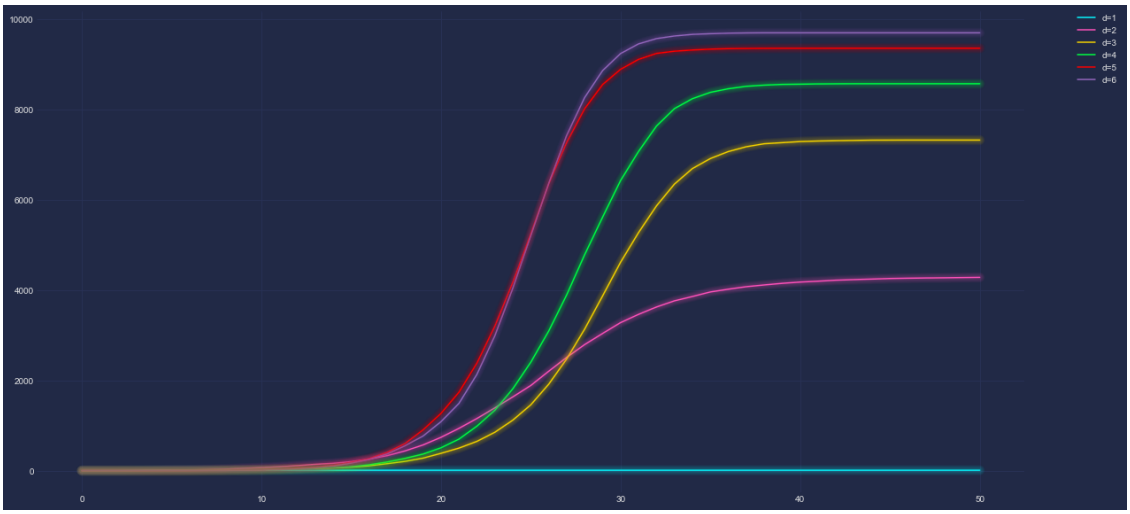
[45]:
```
plt.figure(figsize=(20,10))
for infection_probability in np.arange(0.05,0.3,0.05):
    G = deepcopy(lfr5)
    covert_days = 5
    nDays = 100
    quarantine=Quarantine.AGGRESSIVE
    initialize(G, covert_days, quarantine, infection_probability)
    simulate()
    plt.plot(G.graph['nInfected'], label='p={:.2f}'.
 ↪format(infection_probability))

plt.legend(bbox_to_anchor=(1.05, 1), loc='upper left', borderaxespad=0.)
mplcyberpunk.make_lines_glow()
plt.show()
```



Covert Period Sensitivity Study

Next we investigate how the average length of the covert period affacts the spreading of the virus. In the simulation, we assumed all infected person will show symptoms exactly after the Kth days.

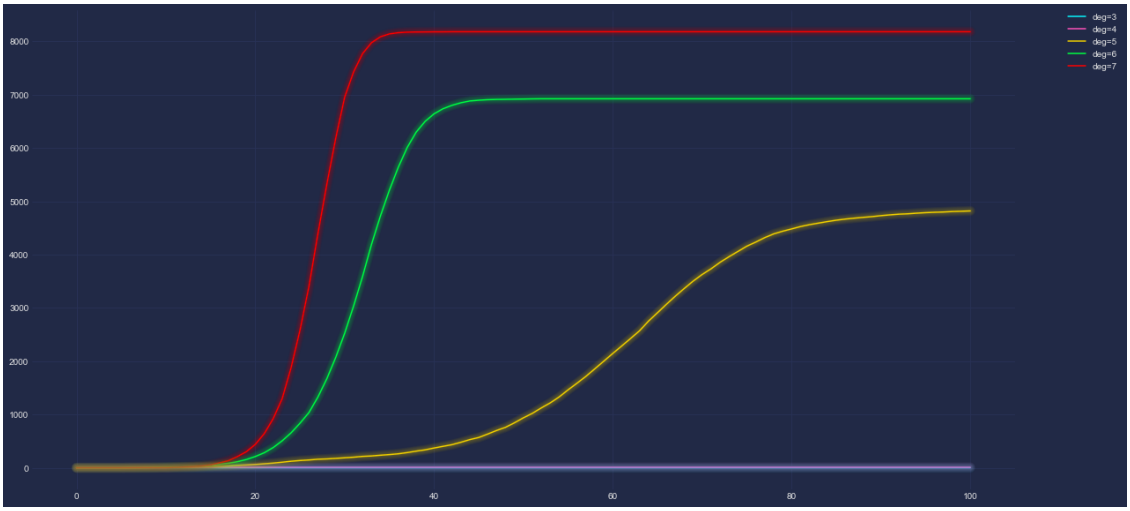
[40]:
```
plt.figure(figsize=(20,10))
for covert_days in range(1,7,1):
    G = deepcopy(lfr5)
    infection_probability = 0.2
    nDays = 50
    quarantine=Quarantine.AGGRESSIVE
    initialize(G, covert_days, quarantine, infection_probability)
    simulate()
    plt.plot(G.graph['nInfected'], label='d='+str(covert_days))

plt.legend(bbox_to_anchor=(1.05, 1), loc='upper left', borderaxespad=0.)
mplcyberpunk.make_lines_glow()
plt.show()
```



Average Degree of Graph Sensitivity Study

Finally we investigate how sensitive the spreading of the virus is regarding to the degree of the graph. From the result it is clear that the higher the average degree of the graph, the faster the virus spreads.

[48]:
```
plt.figure(figsize=(20,10))
for deg in range(3,8):
    G = nx.generators.community.LFR_benchmark_graph(10000,3,1.1,0.
 ↪1,average_degree=deg,max_degree=deg*2,min_community=50,max_community=80,max_iters=500)
    infection_probability = 0.1
    nDays = 100
    covert_days = 5
    quarantine=Quarantine.AGGRESSIVE
    initialize(G, covert_days, quarantine, infection_probability)
    simulate()
    plt.plot(G.graph['nInfected'], label='deg='+str(deg))
plt.legend(bbox_to_anchor=(1.05, 1), loc='upper left', borderaxespad=0.)
mplcyberpunk.make_lines_glow()
plt.show()
```



Animations of the spreading

We now show that the result of the spreading can vary greatly even for two graphs with the same average degree. We use an extreme graph case to show our observation, connected cavemen graph. As it is shown below, the two graph with same average degree and inital setup, ended up with completely different outcomes. In LFR, eventually the virus almost infected the whole network, and in connected cavemen, the virus stopped at a very early stage.

0.0.1 ANIMATION VIEWABLE IN PYTHON NOTEBOOK

LFR, degree of 5

[ ]:
```
G = deepcopy(lfr5) # Make a deep copy so that the original graph remain intact
pos = lfr5_pos
covert_days = 5
nDays = 45
quarantine=Quarantine.AGGRESSIVE
infection_probability = 0.2
fig, (ax1, ax2) = plt.subplots(1,2, figsize=(20,10))
initialize(G, covert_days, quarantine, infection_probability)
ani = animation.FuncAnimation(fig, update, frames=nDays, interval=1000,␣
↪cache_frame_data=False)
#rc('animation', html='jshtml')
#ani
```

Connected Cavemen, Clique of 5

[ ]:
```
G = deepcopy(ccg) # Make a deep copy so that the original graph remain intact
pos = ccg_pos
covert_days = 5
nDays = 45
quarantine=Quarantine.AGGRESSIVE
```

```
infection_probability = 0.2
fig, (ax1, ax2) = plt.subplots(1,2, figsize=(20,10))
initialize(G, covert_days, quarantine, infection_probability)
ani = animation.FuncAnimation(fig, update, frames=nDays, interval=1000,␣
↪cache_frame_data=False)
#rc('animation', html='jshtml')
#ani
```

Conclusions

The spreading speed of the virus is:

- highly sensitive to the infected probability
- highly sensitive to the quarantine method
- somewhat sensitive to the infected probability
- sensitive to the average degree of the graph (for same type of graph)

We further showed that the even two graphs with the same average degree, the spreading speed of the virus can vary greatly.

Part_3_ODE-based Simulation

April 28, 2020

[387]:
```
import numpy as np
from scipy.integrate import odeint
import matplotlib.pyplot as plt
from ipywidgets import interact
import networkx as nx
import pandas as pd
import random
import matplotlib.pyplot as plt
from copy import deepcopy
import ipywidgets as widgets
from ipywidgets import interact, interactive, fixed, interact_manual
from networkx.classes.function import *
import matplotlib.pyplot as plt
```

1 ODE-based Simulation Model

To model the spreading of the epidemic disease, ordinary differential equation models have been used. There are several kinds of models including SIR, SIS, SEIR and SEIRS, where each particular letter stands for one of the groups of the whole population (Susceptible, Infective, Exposed and Recovered), and the time evolution of these groups is modeled by the set of differential equations with several parameters as external factors.

To take a look into the ODE-based simulation model, we start at the basic SEIR model to model the dynamics of the number of four different kinds of people.

1.1 1. Simple SEIR model

The time evolution of the population compartments in ODE model is described by several nonlinear differential equations. For instance, the basic SEIR model can be written as several equations:

$$\frac{dS}{dt} = \mu(N - S) - \beta\frac{SI}{N}$$

$$\frac{dE}{dt} = \beta\frac{SI}{N} - (\mu + \sigma)E$$

$$\frac{dI}{dt} = \sigma E - (\mu + \gamma)I$$

$$\frac{dR}{dt} = \gamma I - \mu R$$

where S, E, I, R stands for the *Susceptible*, *Exposed*, *Infected*, *Recovered* people respectively.

- $N = S + E + I + R$ is the total number of population,
- β controls how often a susceptible-infected contact results in a new exposure,
- γ is the rate an infected recovers and moves into the resistant phase,
- σ is the rate an exposed person becomes infective,
- μ is the natural mortality rate (unrelated to disease).

[388]:
```
def seir_base_simulation(N = 1000,
                         I0 = 5,
                         R0 = 0,
                         E0 = 5,
                         beta = 0.8,
                         gamma = 0.1,
                         sigma = 0.3,
                         mu = 0.0001,
                         t = 100):

    # Total population, N.
    #N = N
    # Initial number of infected and recovered individuals, I0 and R0.
    #I0, R0 = 1, 0
    # Everyone else, S0, is susceptible to infection initially.
    S0 = N - I0 - R0 - E0
    # Contact rate, beta, and mean recovery rate, gamma, (in 1/days).
    # A grid of time points (in days)
    t = np.linspace(0, t, t)

    # The SIR model differential equations.
    def deriv(y, t, N, beta, gamma, sigma, mu):
        S, E, I, R = y
        beta = beta # * np.exp(-t/50)
        dSdt = mu * (N - S) - beta * S * I / N
        dEdt = beta * S * I / N - (mu + sigma) * E
        dIdt = sigma * E - (mu + gamma) * I
        dRdt = gamma * I - mu * R
        return dSdt, dEdt, dIdt, dRdt

    # Initial conditions vector
    y0 = S0, E0, I0, R0
    # Integrate the SIR equations over the time grid, t.
    ret = odeint(deriv, y0, t, args=(N, beta, gamma, sigma, mu))
    S, E, I, R = ret.T

    # Plot the data on three separate curves for S(t), I(t) and R(t)
```

```
    fig = plt.figure(facecolor='w', figsize = [9, 6])
    ax = fig.add_subplot(111,  axisbelow=True)
    ax.plot(t, S/N, 'b', alpha=0.6, lw=2.5, label='Susceptible')
    ax.plot(t, E/N, 'purple', alpha=0.6, lw=2.5, label='Exposed')
    ax.plot(t, I/N, 'r', alpha=0.6, lw=2.5, label='Infected')
    ax.plot(t, R/N, 'g', alpha=0.6, lw=2.5, label='Recovered')
    ax.set_xlabel('Time /days')
    ax.set_ylabel('Ratio of people')
    ax.set_ylim(0,1.1)
    ax.yaxis.set_tick_params(length=0)
    ax.xaxis.set_tick_params(length=0)
    ax.grid(b=True, which='major',  c='grey',lw=1, ls=':')
    legend = ax.legend()
    legend.get_frame().set_alpha(0.5)
    for spine in ('top', 'right', 'bottom', 'left'):
        ax.spines[spine].set_visible(False)
    plt.show()
```

2 Simulation with interaction.

By changing different values on different parameters, the shape of curve will be different.

[389]:
```
interact (seir_base_simulation
    , N=(1, 1000, 1)
    , I0=(1, 50, 1)
    , E0=(0, 100,1)
    , beta=(0, 1, 0.05)
    , gamma = (0, 0.5, 0.01)
    , sigma = (0, 1, 0.05)
    , mu = (0, 0.01, 0.001)
    , t = (0, 300, 10)
    );
```

```
interactive(children=(IntSlider(value=1000, description='N', max=1000, min=1), IntSlider(value=
```

3 2. More Complex Model: SIRS model with Quarantine

Apart from the existing modules in SIRS model, we also considers the effect of self-quaratine, the infected/exposed people will stay at home to avoid interacting with others. Specifically, we add to extra parameter Q_E, Q_I which stand for the number of exposed/infected population being quarantined respectively. Then, the ODE formula can be expressed as

$$\frac{dS}{dt} = \mu(N - S) - \beta\frac{SI}{N}$$

$$\frac{dE}{dt} = \beta\frac{SI}{N} - (\mu + \sigma + q)E$$

$$\frac{dI}{dt} = \sigma E - (\mu + \gamma + q)I$$

$$\frac{dR}{dt} = \gamma(I + Q_I) - \mu R$$

$$\frac{dQ_E}{dt} = -\sigma Q_E + qE$$

$$\frac{dQ_I}{dt} = -\gamma Q_I + \sigma Q_E + qI$$

- q is the proportion of population being self-quarantined.

[390]:
```
def seir_add_simulation(N = 1000,
                        I0 = 1,
                        R0 = 0,
                        E0 = 0,
                        beta = 0.8,
                        gamma = 0.1,
                        sigma = 0.3,
                        mu = 0.0001,
                        q = 0.05,
                        t = 100):

    # Total population, N.
    #N = N
    # Initial number of infected and recovered individuals, I0 and R0.
    #I0, R0 = 1, 0
    # Everyone else, S0, is susceptible to infection initially.
    QE0, QI0 = 0, 0
    S0 = N - I0 - R0 - E0 - QE0 - QI0

    # Contact rate, beta, and mean recovery rate, gamma, (in 1/days).
    # A grid of time points (in days)
    t = np.linspace(0, t, t)

    # The SIR model differential equations.
    def deriv(y, t, N, beta, gamma, sigma, mu, q):
        S, E, I, QE, QI, R = y
        dSdt = mu * (N - S) -beta * S * I / N
        dEdt = beta * S * I / N - (mu + sigma + q) * E
        dIdt = sigma * E - (mu + gamma + q) * I
        dRdt = gamma * I - mu * R + gamma * QI
        dQEdt = - sigma * QE + q * E
        dQIdt = - gamma * QI + sigma * QE + q * I
        return dSdt, dEdt, dIdt, dQEdt, dQIdt, dRdt
```

```
    # Initial conditions vector
    y0 = S0, E0, I0, QE0, QI0, R0
    # Integrate the SIR equations over the time grid, t.
    ret = odeint(deriv, y0, t, args=(N, beta, gamma, sigma, mu, q))
    S, E, I, QE, QI, R = ret.T

    # Plot the data on three separate curves for S(t), I(t) and R(t)
    fig = plt.figure(facecolor='w', figsize = [9, 6])
    ax = fig.add_subplot(111,  axisbelow=True)
    ax.plot(t, S/N, 'b', alpha=0.6, lw=2.5, label='Susceptible')
    ax.plot(t, (E+QE)/N, 'purple', alpha=0.6, lw=2.5, label='Exposed')
    ax.plot(t, (I+QI)/N, 'r', alpha=0.6, lw=2.5, label='Infected')
    ax.plot(t, R/N, 'g', alpha=0.6, lw=2.5, label='Recovered')
    #ax.plot(t, QE/N, 'hotpink', alpha=0.6, lw=2.5, label='Recovered')
    #ax.plot(t, QI/N, 'orange', alpha=0.6, lw=2.5, label='Recovered')
    ax.set_xlabel('Time /days')
    ax.set_ylabel('Ratio of people')
    ax.set_ylim(0,1.1)
    ax.yaxis.set_tick_params(length=0)
    ax.xaxis.set_tick_params(length=0)
    ax.grid(b=True, which='major',  c='grey',lw=1, ls=':')
    legend = ax.legend()
    legend.get_frame().set_alpha(0.5)
    for spine in ('top', 'right', 'bottom', 'left'):
        ax.spines[spine].set_visible(False)
    plt.show()
```

Simulation with interaction. By changing different values on different parameters, the shape of curve will be different. Compare our model with different q, it is clear that quarantine will reduce the total proportion of infected population.

[391]:
```
interact (seir_add_simulation
    , N=(1, 1000, 1)
    , I0=(1, 50, 1)
    , E0=(0, 100,1)
    , beta=(0, 1, 0.05)
    , gamma = (0, 0.5, 0.01)
    , sigma = (0, 1, 0.05)
    , mu = (0, 0.01, 0.001)
    , q = (0, 1, 0.05)
    , t = (0, 300, 10)
    );
```

```
interactive(children=(IntSlider(value=1000, description='N', max=1000, min=1), IntSlider(value=
```

4 3. More Complicated Case: Modeling the spread of the disease in meta-population level

The basic SEIR model (introduced in part 1) implicitly assumes a homogeneous infection network between individuals and thus can only model epidemic propagation in a single population. Here, we extend the SIR model to the metapopulation scenario.

4.1 Model Description

A **metapopulation** refers a group of separated sub-populations which interact at some level. For instance, we can model each city in USA as a sub-population, and all cities in USA together as a metapopulation. Therefore, apart from learning the dynamics in each small sub-populations, we can better model the population flows between different regions and capture the dynamic patterns on disease spreading.

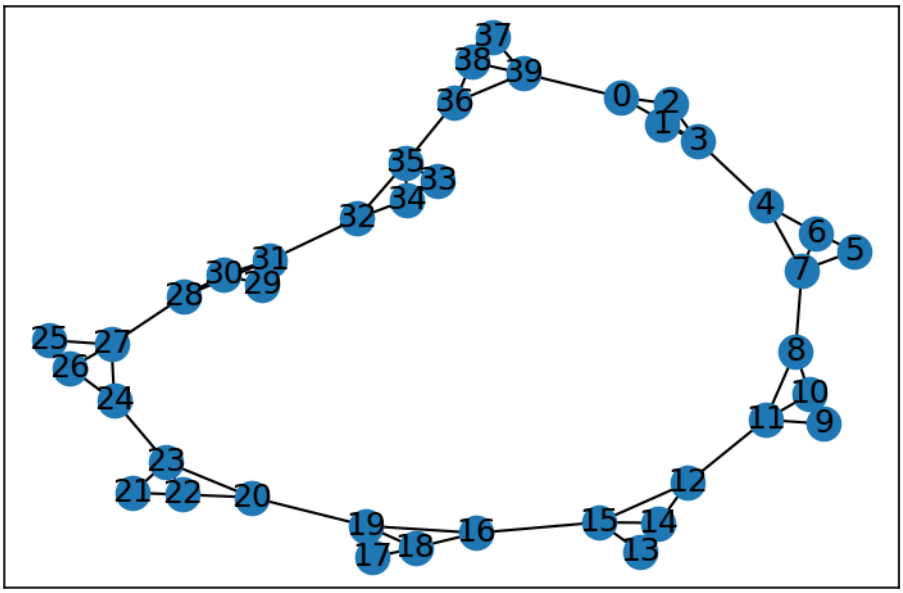
Specifically, in our problem, given a meta-population with N sub-populations, we denote the total number of individuals in sub-population n as P_n , and the numbers of individuals in the S, E, I, R states at time t as s_n, e_n, i_n, r_n, respectively. Between sub-populations n and m, we define the interaction strength as h_{nm}, which is the average volume of visitors from n to m in a unit time.

4.2 Graph Construction

To better model the dynamic of populations, i.e. h_{nm}, we can build a weighted graph $G = (V, E)$ over the n sub-populations. Here V is the set of nodes which corresponds to n sub-populations, i.e. $|V| = n$. E is the edge sets, and for each edge $e = (m, n) \in E$, there is a weight $w = h_{mn}$ that control the strength of the population flows between the nodes. Our model is graph-agnostic and can be applied to graphs with arbitrary size.

To begin with, we adopt the similar topology of the graph studied in the previous part. Here we use Connected Cavemen Graph, which is a ring of cliques as an example. We can run the following code multiple times to generate a good layout.

[392]:
```
from networkx.linalg.graphmatrix import *
l = 10 # (int) - number of cliques
k = 4 # (int) - size of cliques
G = nx.generators.community.connected_caveman_graph(l, k) ##
ccg_pos = nx.layout.spring_layout(G)
plt.figure(dpi =150)
nx.draw_networkx(ccg, node_size=150, pos=ccg_pos)
plt.show()
A = adjacency_matrix(G).toarray() # adjacency matrix of the graph
```



4.3 Modeling population flows

To model the mobility among different sub-populations, we need to simulate the population flow among them. In graph perspective, these can be regarded as the flow between different edge pairs. To calculate the population flow, there are several ways: - If the real life mobility data is availble, then we can use the corresponding real data to calculate the flow matrix H for simulation. - In many cases, if the real data is unavailable, then we can also do some simulations based on random walk theory on graphs.

Here, since it is difficult to collect the real-life mobility flow data, we aim to simulate the mobility based on node degree. Specifically, given a graph G with the adjacency matrix A, the ratio of the flow population from node m to n is in proportion to the node degree d_m, i.e. $\frac{h_{mn}}{p_m} \propto \frac{1}{d_m}$. In this case, we consider each neighbor nodes as homogeneous and can formally write the transition as

$$\hat{A} = D^{-1}A$$
$$H = (1-\delta)I + \delta\hat{A}$$

where $D = \text{diag}(d_1, d_2, ..., d_N)$ denotes the degree for each node, A is the adjacency matrix of the graph V, δ is a parameter that controls the strength of the flow, and h is the transition ratio matrix that determine *the proportion of* population among different sub-populations.

Also, we can consider the symmetric case where the the ratio of the flow population from node m to n is in related to both the degree of node m and n, as $1/\sqrt{d_m d_n}$, i.e. $\frac{h_{mn}}{p_m} \propto \frac{1}{\sqrt{d_m d_n}}$. In this case, we can formally write the transition as

$$\hat{A} = D^{-1/2}AD^{-1/2}$$

$$H = \text{normalize}\left((1-\delta)I + \delta\hat{A}\right)$$

Here we need to normalize H since the original sum of h for each row is not equal to 1. Note that $H_{mn} = h_{mn}/P_m$.

[393]:
```
def normalized_adjacency(A, symmetry = False):
    if symmetry:
        x = np.diag(1/np.sqrt(np.sum(A, axis = 0)))
        y = (x).dot(A).dot(x)
        z = np.diag(1/np.sum(y, axis = 0))
        return z.dot(y)

    else:
        x = np.diag(1/np.sum(A, axis = 0))
        return x.dot(A)

A_hat = normalized_adjacency(A, symmetry = False)
print('normalized matrix:', A_hat)
print('sum of each row (should be 1):', np.sum(A_hat, axis = 1)) # check that␣
 ↪every row of A sum to 1
```

```
normalized matrix: [[0.         0.         0.33333333 … 0.         0.
0.33333333]
 [0.         0.         0.5        … 0.         0.         0.        ]
 [0.33333333 0.33333333 0.         … 0.         0.         0.        ]
 …
 [0.         0.         0.         … 0.         0.5        0.5       ]
 [0.         0.         0.         … 0.33333333 0.         0.33333333]
 [0.25       0.         0.         … 0.25       0.25       0.        ]]
sum of each row (should be 1): [1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.
1. 1. 1. 1. 1. 1. 1. 1.
 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]
```

4.4 Calculating Mobility Flows

Then, in a metapopulation, a susceptible individual of sub-population n may contact with infectious individuals from several sources:

- the infectious *in the same sub-population* with a total number of i_n, which will result in

  $$\beta \cdot s_n \cdot i_n$$

  new infectious in n-th subpopulation where β is the infection rate;
- the infectious visitors *from other sub-populations*. The probability for an individual in m visiting n can be estimated by h_{mn}/P_m, so the new infectious in n totals

  $$\beta \cdot s_n \sum_{m \in \mathcal{N}_n} (h_{mn}/P_m)\, i_m$$

  where $\mathcal{N}_n$ is the neighborhood of n.

Also, since for each sub-population, there exists several people that come to other sub-populations, then the probability for an individual in m visiting n can be estimated by h_{mn}/P_m, so the new infectious from n totals

$$\beta \cdot \sum_{m \in \mathcal{N}_n} (h_{nm}/P_n)\, i_n$$

where $\mathcal{N}_n$ is the neighborhood of n.

Then, to sum up, the total number of new exposures from sub-population n caused by the three types of contacts is

$$\beta \cdot s_n \sum_{m \in \mathcal{N}_n} (h_{mn}/P_m - h_{nm}/P_n)\, i_m + \beta s_n \cdot i_n$$

Since we also need to consider the dynamic sub-populations, then we need to first calculate the population after flow transition as $s_n^{'} = \sum_m \frac{h_{mn}}{P_m} s_m$, $e_n^{'} = \sum_m \frac{h_{mn}}{P_m} e_m$, $i_n^{'} = \sum_m \frac{h_{mn}}{P_m} i_m$, $r_n^{'} = \sum_m \frac{h_{mn}}{P_m} r_m$. Note that $h_{nn}/P_n = 1 - \sum_{m \in \mathcal{N}_n} h_{nm}/P_n$, then, from the above analysis, we can rewrite the dynamic relationship of s_n, e_n, i_n, r_n as

$$\frac{\mathrm{d}s_n}{\mathrm{d}t} = \mu(P_n - s_n') - \beta s_n' \cdot i_n'$$

$$\frac{\mathrm{d}e_n}{\mathrm{d}t} = \beta s_n' \cdot i_n' - (\mu + \sigma) e_n'$$

$$\frac{\mathrm{d}i_n}{\mathrm{d}t} = \sigma \cdot e_n' - (\mu + \gamma) i_n'$$

$$\frac{\mathrm{d}r_n}{\mathrm{d}t} = \gamma \cdot i_n' - \mu \cdot r_n'$$

where s_n, e_n, i_n, r_n stands for the *Susceptible*, *Exposed*, *Infected*, *Recovered* people for sub-population n respectively. - $P_n = s_n' + e_n' + i_n' + r_n'$ is the total number of population, - β controls how often a susceptible-infected contact results in a new exposure, - γ is the rate an infected recovers and moves into the resistant phase, - σ is the rate an exposed person becomes infective, - μ is the natural mortality rate (unrelated to disease).

Note that the form of the ODE equation is very similar to the previous sub-population level, this is mainly due to the population mobility can be formulated as the 'biased random walk' on neighbors, and such walk can be modeled via a transition matrix from the analysis above. Actually the whole system now is more complicated and need to consider the dynamics for all sub-populations together.

4.5 Simulation Steps

For simulation the spread of the disease, we assume that the total population for each sub-population is the same for simplicity (but it surely can be changed), and the virus appears on sub-population #0. For different parameters, we add several assumptions which make sense under the real scenarios. Specifically, for parameter γ, we add a more strong assumtion that it varies with time. This is mainly because with the passage of the time, people get more aware of the disease and more powerful medication is likely to be introduced. Therefore, people can get better treatment and is easier to recover from the disease. In our experiments, we just use the linear function to

model the relation between γ and t. Specifically, let the initial γ be γ_0, then the γ can be written as

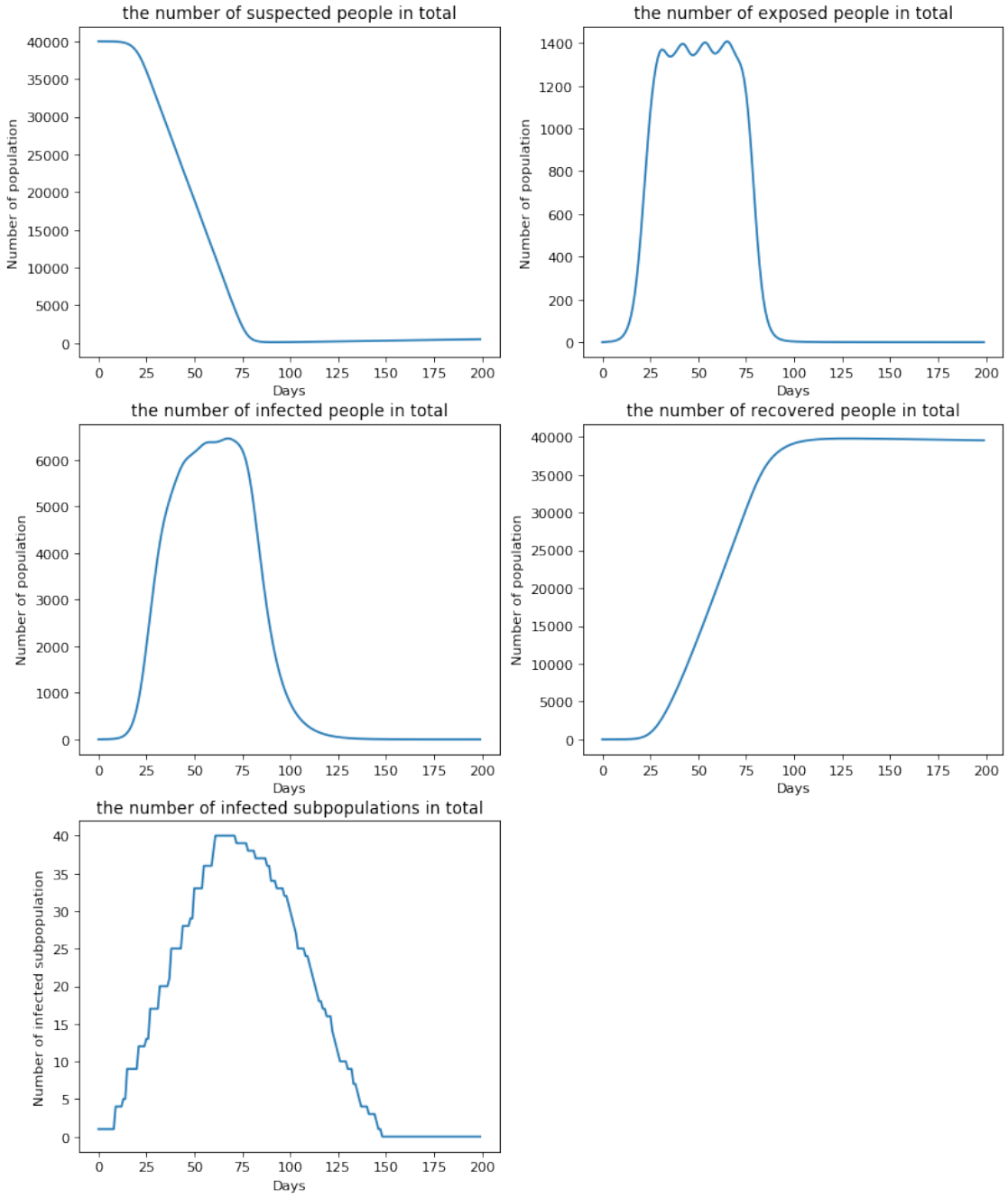
$$\gamma_t = \gamma_0(1 + \alpha t)$$

where α is a parameter controlling the increase speed of γ. Then, below is a simple case with $N = 1000$, $\alpha = 0.001$, $\beta = 0.8$, $\gamma_0 = 0.1$, $\sigma = 0.5$, $\mu = 0.0001$, $\delta = 0.8$.

[394]:
```
N = 1000 # number of sub-population
M = A_hat.shape[0] # number of sub-populations
I0 = np.zeros(M)#np.random.randint(size = M, low=10, high = 30) #0
I0[0] = 1
R0 = np.zeros(M)  #0
E0 = np.zeros(M) #0
beta = 0.8
gamma = 0.1
sigma = 0.5
mu = 0.0001
T = 200
delta = 0.9
A = np.eye(M) * delta + A_hat * (1-delta)
# Everyone else, S0, is susceptible to infection initially.
# QE0, QI0 = np.zeros(M), np.zeros(M) #0, 0
S0 = N - I0 - R0 - E0

# Contact rate, beta, and mean recovery rate, gamma, (in 1/days).
# A grid of time points (in days)
t = np.linspace(0, T, T)

# The SIR model differential equations.
def deriv(y, t, N, beta, gamma, sigma, mu, q, A):
    y = y.reshape(4, -1)

    S, E, I, R = y[0], y[1], y[2], y[3] # S, E, I, R: array with shape (M, )
    #print(S.shape,E,I,R)
    S = A.T.dot(S)
    E = A.T.dot(E)
    I = A.T.dot(I)
    R = A.T.dot(R)
    N = S + E + I + R
    gamma2 = gamma * (1+t*0.001)
    dSdt = mu * (N - S) - beta * S * I / N
    dEdt = beta * S * I / N - (mu + sigma) * E
    dIdt = sigma * E - (mu + gamma2) * I
    dRdt = gamma2 * I - mu * R
    return np.hstack([dSdt, dEdt, dIdt, dRdt])
    #return dSdt, dEdt, dIdt, dRdt

# Initial conditions vector
```

```
y0 = np.hstack([S0, E0, I0,  R0])
print('Input shape [should be %d]:'%(4*M), y0.shape)
# Integrate the SIR equations over the time grid, t.
ret = odeint(deriv, y0, t, args=(N, beta, gamma, sigma, mu, q, A))

ret = ret.T
print('Output shape [should be (%d, %d)]:'%(4*M, T), ret.shape)

S = ret[ : M]
E = ret[M : 2*M]
I = ret[2*M : 3*M]
R = ret[3*M : 4*M]
#S, E, I, R = ret.T

num_of_infected_sub_population = []
for i in range(T):
    num_of_infected_sub_population.append(len([x for x in I[:, i] if int(x) >=␣
 ↪1]))

plt.figure(figsize = [12,15], dpi = 80)
plt.subplot(3,2,1)
plt.plot(np.sum(S, axis = 0))
plt.xlabel('Days')
plt.ylabel('Number of population')
plt.title('the number of suspected people in total')
plt.subplot(3,2,2)
plt.plot(np.sum(E, axis = 0))
plt.xlabel('Days')
plt.ylabel('Number of population')
plt.title('the number of exposed people in total')
plt.subplot(3,2,3)
plt.plot(np.sum(I, axis = 0))
plt.xlabel('Days')
plt.ylabel('Number of population')
plt.title('the number of infected people in total')
plt.subplot(3,2,4)
plt.plot(np.sum(R, axis = 0))
plt.xlabel('Days')
plt.ylabel('Number of population')
plt.title('the number of recovered people in total')
plt.subplot(3,2,5)
plt.plot(num_of_infected_sub_population)
plt.xlabel('Days')
plt.ylabel('Number of infected subpopulation')
plt.title('the number of infected subpopulations in total')
plt.show()
```

```
Input shape [should be 160]: (160,)
Output shape [should be (160, 200)]: (160, 200)
```



4.6 Analysis

From the above analysis, we can see that the dynamics differs quit a lot for different δ. For instance, with the higher δ, the spread of the disease will slower, and the peak number of infected people will be smaller. Moreover, the number of the infected subpopulations will also increase a bit slower.

These phenomenons indicates that by reducing the travel among different sub-populations, the disease will be better controlled.

4.7 Modeling with interactions

To better illustrate the effect of different parameters, we try to build a GUI interface to simulate it from both macro and micro perspectives. By switching over 'target', you can choose the number of population that you would like to visualize. ### Macro Perspective: We observe the total number of four kinds of population (S, E, I, R) and plot them in a figure. ### Micro Perspective: We observe the number of four kinds of population (S, E, I, R) for each sub-population and plot them in a figure.

[395]:
```
def seir_network_simulation(N = 1000,
                            beta = 0.8,
                            gamma0 = 0.1,
                            sigma = 0.5,
                            alpha = 0.001,
                            mu = 0.0001,
                            T = 200,
                            delta = 0.9,
                            target = 1):

    #N = 1000 # number of sub-population
    M = A_hat.shape[0] # number of sub-populations
    I0 = np.zeros(M) # np.random.randint(size = M, low=10, high = 30) #0
    I0[0] = 1
    R0 = np.zeros(M)  #0
    E0 = np.zeros(M) #0
    beta = 0.8
    gamma = gamma0
    A = np.eye(M) * delta + A_hat * (1-delta)
    # Everyone else, S0, is susceptible to infection initially.
    # QE0, QI0 = np.zeros(M), np.zeros(M) #0, 0
    S0 = N - I0 - R0 - E0
    t = np.linspace(0, T, T)

    # The SIR model differential equations.
    def deriv(y, t, N, beta, gamma, sigma, mu, q, A):
        y = y.reshape(4, -1)

        S, E, I, R = y[0], y[1], y[2], y[3] # S, E, I, R: array with shape (M, )
        #print(S.shape,E,I,R)
        S = A.T.dot(S)
        E = A.T.dot(E)
        I = A.T.dot(I)
        R = A.T.dot(R)
        N = S + E + I + R
        gamma2 = gamma * (1+t*0.001)
```

```
        dSdt = mu * (N - S) - beta * S * I / N
        dEdt = beta * S * I / N - (mu + sigma) * E
        dIdt = sigma * E - (mu + gamma2) * I
        dRdt = gamma2 * I - mu * R
        return np.hstack([dSdt, dEdt, dIdt, dRdt])

    # Initial conditions vector
    y0 = np.hstack([S0, E0, I0,  R0])
    print('Input shape [should be %d]:'%(4*M), y0.shape)
    # Integrate the SIR equations over the time grid, t.
    ret = odeint(deriv, y0, t, args=(N, beta, gamma, sigma, mu, q, A))
    ret = ret.T
    print('Output shape [should be (%d, %d)]:'%(4*M, T), ret.shape)

    S = ret[ : M]
    E = ret[M : 2*M]
    I = ret[2*M : 3*M]
    R = ret[3*M : 4*M]

    # Plot the data on three separate curves for S(t), I(t) and R(t)
    fig = plt.figure(facecolor='w', figsize =  [8,6], dpi = 100)
    ax = fig.add_subplot(111,  axisbelow=True)
    if target == "All":
        plt.plot(t, np.sum(S, axis = 0), 'b', alpha=0.6, lw=2.5,␣
↪label='Susceptible')
        plt.plot(t, np.sum(E, axis = 0), 'purple', alpha=0.6, lw=2.5,␣
↪label='Exposed')
        plt.plot(t, np.sum(I, axis = 0), 'r', lw=2.5, label='Infected')
        plt.plot(t, np.sum(R, axis = 0), 'g', lw=2.5, label='Recovered')
        plt.title('Curve for all sub-populations')
    else:
        plt.plot(t, S[target], 'b', alpha=0.6, lw=2.5, label='Susceptible')
        plt.plot(t, E[target], 'purple', alpha=0.6, lw=2.5, label='Exposed')
        plt.plot(t, I[target], 'r', lw=2.5, label='Infected')
        plt.plot(t, R[target], 'g', lw=2.5, label='Recovered')
        plt.title('Curve for sub-population %d'%target)
    ax.set_xlabel('Time /days')
    ax.set_ylabel('Number of people')
    ax.yaxis.set_tick_params(length=0)
    ax.xaxis.set_tick_params(length=0)
    ax.grid(b=True, which='major',  c='grey',lw=1, ls=':')
    legend = ax.legend()
    legend.get_frame().set_alpha(0.5)
    for spine in ('top', 'right', 'bottom', 'left'):
        ax.spines[spine].set_visible(False)

    plt.show()
```

[358]:
```
interact (seir_network_simulation
    , N=(1, 1000, 1)
    , beta=(0, 1, 0.05)
    , gamma0 = (0, 0.5, 0.01)
    , alpha = (0, 0.1, 0.001)
    , sigma = (0, 1, 0.05)
    , mu = (0, 0.01, 0.001)
    , T = (0, 300, 10)
    , delta = (0.6, 1, 0.05)
    , target = ["All"] + [i for i in range(A_hat.shape[0])]
    );
```

```
interactive(children=(IntSlider(value=1000, description='N', max=1000, min=1), FloatSlider(valu
```

From the above visualization, it is clear that, for the nodes (i.e. subpopulations) near to the origin node (node #0), the peak number of infected people got higher, and the peak arrives early. This is mainly due to the graph structure, the propagation of virus takes some time. Also, with the passage of time, γ increases, therefore the increase of the infected population will be slower, making the peak becomes lower for these distant nodes.

5 Simulations on different random graphs

Apart from the small world graph, our method can also generalized to other types of social networks. One of the random scale-free network is Barabasi-Albert graph, which is generated using a preferential attachment mechanism.

Specifically, the Barabasi-Albert graph begins with an initial connected network of l nodes. New nodes are added to the graph one at a time. Each new node is connected to $k (k < l)$ existing nodes with a probability that is proportional to the number of links that the existing nodes already have. Formally, the probability that the new node m is connected to node n is

$$p_n = \frac{\deg(n)}{\sum_k \deg(k)}$$

where $\deg(k)$ is the degree of node k and the sum is made over all pre-existing nodes k. In this way, the such kind of graphs have power-law (or scale-free) degree distributions, with the degree distribution

$$P(k) \propto k^{-3}$$

Moreover, the average path length of the Barabasi-Albert graph increases approximately logarithmically with the size of the network as

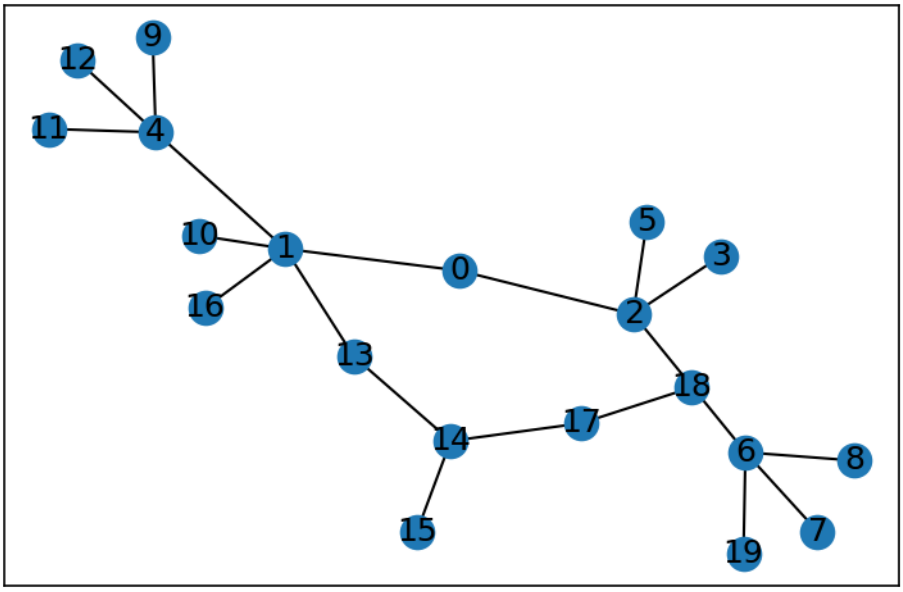
$$\ell \sim \frac{\ln N}{\ln \ln N}$$

In the simulation, we will try on different parameter k and observe the properties of the disease spread.

Reference: https://en.wikipedia.org/wiki/Barab%C3%A1si%E2%80%93Albert_model.

[399]:
```
from networkx.generators.random_graphs import *
l = 20 # (int) - number of cliques
k = 1 # (int) - size of cliques
G = barabasi_albert_graph(l, k)
ccg_pos = nx.layout.spring_layout(G)
plt.figure(dpi =150)
nx.draw_networkx(G, node_size=150, pos=ccg_pos)
print('A sample barabasi albert graph with l = 20, k = 2')
plt.show()
```

```
A sample barabasi albert graph with l = 20, k = 2
```



[397]:
```
def seir_network_simulation_barabasi_albert(N = 1000,
                          beta = 0.8,
                          gamma0 = 0.1,
                          sigma = 0.5,
                          alpha = 0.001,
                          mu = 0.0001,
                          T = 200,
                          delta = 0.9,
                          target = 1,
                          k = 3):
    G = barabasi_albert_graph(l, k)
    A = adjacency_matrix(G).toarray() # adjacency matrix of the graph
    A_hat = normalized_adjacency(A, symmetry = False)
    #N = 1000 # number of sub-population
    M = A_hat.shape[0] # number of sub-populations
```

```
I0 = np.zeros(M) # np.random.randint(size = M, low=10, high = 30) #0
I0[0] = 1
R0 = np.zeros(M)  #0
E0 = np.zeros(M) #0
beta = 0.8
gamma = gamma0
A = np.eye(M) * delta + A_hat * (1-delta)
# Everyone else, S0, is susceptible to infection initially.
# QE0, QI0 = np.zeros(M), np.zeros(M) #0, 0
S0 = N - I0 - R0 - E0
t = np.linspace(0, T, T)

# The SIR model differential equations.
def deriv(y, t, N, beta, gamma, sigma, mu, q, A):
    y = y.reshape(4, -1)

    S, E, I, R = y[0], y[1], y[2], y[3] # S, E, I, R: array with shape (M, )
    #print(S.shape,E,I,R)
    S = A.T.dot(S)
    E = A.T.dot(E)
    I = A.T.dot(I)
    R = A.T.dot(R)
    N = S + E + I + R
    gamma2 = gamma * (1+t*0.001)
    dSdt = mu * (N - S) - beta * S * I / N
    dEdt = beta * S * I / N - (mu + sigma) * E
    dIdt = sigma * E - (mu + gamma2) * I
    dRdt = gamma2 * I - mu * R
    return np.hstack([dSdt, dEdt, dIdt, dRdt])

# Initial conditions vector
y0 = np.hstack([S0, E0, I0,  R0])
print('Input shape [should be %d]:'%(4*M), y0.shape)
# Integrate the SIR equations over the time grid, t.
ret = odeint(deriv, y0, t, args=(N, beta, gamma, sigma, mu, q, A))
ret = ret.T
print('Output shape [should be (%d, %d)]:'%(4*M, T), ret.shape)

S = ret[ : M]
E = ret[M : 2*M]
I = ret[2*M : 3*M]
R = ret[3*M : 4*M]

# Plot the data on three separate curves for S(t), I(t) and R(t)
fig = plt.figure(facecolor='w', figsize =  [8,6], dpi = 100)
ax = fig.add_subplot(111,  axisbelow=True)
if target == "All":
```

```
        plt.plot(t, np.sum(S, axis = 0), 'b', alpha=0.6, lw=2.5,␣
↪label='Susceptible')
        plt.plot(t, np.sum(E, axis = 0), 'purple', alpha=0.6, lw=2.5,␣
↪label='Exposed')
        plt.plot(t, np.sum(I, axis = 0), 'r', lw=2.5, label='Infected')
        plt.plot(t, np.sum(R, axis = 0), 'g', lw=2.5, label='Recovered')
        plt.title('Curve for all sub-populations')
    else:
        plt.plot(t, S[target], 'b', alpha=0.6, lw=2.5, label='Susceptible')
        plt.plot(t, E[target], 'purple', alpha=0.6, lw=2.5, label='Exposed')
        plt.plot(t, I[target], 'r', lw=2.5, label='Infected')
        plt.plot(t, R[target], 'g', lw=2.5, label='Recovered')
        plt.title('Curve for sub-population %d'%target)
    ax.set_xlabel('Time /days')
    ax.set_ylabel('Number of people')
    ax.yaxis.set_tick_params(length=0)
    ax.xaxis.set_tick_params(length=0)
    ax.grid(b=True, which='major',  c='grey',lw=1, ls=':')
    legend = ax.legend()
    legend.get_frame().set_alpha(0.5)
    for spine in ('top', 'right', 'bottom', 'left'):
        ax.spines[spine].set_visible(False)

    plt.show()
```

[398]:
```
interact (seir_network_simulation_barabasi_albert
    , N=(1, 1000, 10)
    , beta=(0, 1, 0.05)
    , gamma0 = (0, 0.5, 0.01)
    , alpha = (0, 0.05, 0.001)
    , sigma = (0, 1, 0.05)
    , mu = (0, 0.01, 0.001)
    , T = (0, 300, 10)
    , delta = (0.6, 1, 0.05)
    , target = ["All"] + [i for i in range(1)]
    , k = (1, int(1/2), 1)
    );
```

```
interactive(children=(IntSlider(value=1000, description='N', max=1000, min=1, step=10), FloatSl
```

From the above simulation, we find that with the increase of k, the corresponding peak time almost remains same, but the peak value increases significantly when k is increased from 1 to 2 and 2 to 3. Then, the increase is not obvious. We explain such phenomenon as: at the first, the density of the Barabasi-Albert model increase very fast, and in this case and the spread of the virus among different regions will be weaker. In our real life, such 'edge' can be regarded as the transportstion tools (e.g. flights, trains), and by reducing these transportations, it will be useful to reduce the infection rate.

6 Conclusion

From the above, we aim to simulate the spread of the disease via using ODE-based SEIR on community-level and network-level disease spread, and justify the effect of social distancing (quarantine) and reducing the transportation tools. We give detailed analysis on several key parameter to better explain the role of several factors.

Split the work among the members of the team:

Yue Yu: Developed the ODE-based model, helped on final report.

Chenjun Tang: Developed the Cellular Automata model and helped on final report.

Tianqi Liu: Developed the Graph-based model and helped on final report.

All the group members take part in the literature survey, idea forming and result discussion actively.