AI Term Project Report

1. Problem Description

Given the NTU campus map, the initial number of healthy/infected students, each student’s schedule, and the SEIR infection model, could we find the trend of the number of infected students on campus? Additionally, can we take advantage of the result to investigate some potential efficacies of various public health policies?

2. Importance

In recent months, COVID-19 makes people panic. By predicting the trend of the number of infected students, we can apply corresponding policies to prevent it. To be more specific, if we can simulate the trend of the number of infected students on our campus, we can apply the built model with different movement patterns and schedules of people to simulate the infection trend in our society. By observing the results of the simulation, we can adopt corresponding strategies to fight against COVID-19 so that it will cause the minimum loss, making our society stronger and healthier.

3. State of the art

In 《利用整合式傳染病監測軟體以協助流行性感冒的監測並提供公衛參考》[1], the researchers constructed an influenza simulation model based on daily-contact social networks and a multi-agent system to study transmission dynamics and to investigate the potential efficacies of various public health policies. However, though the method they've used called '分身點' to simulate the movement of people, which basically replicates several doppelgangers of a single person and puts them at different locations, making it easier to implement, however, the fidelity and accuracy were sacrificed.

In *Simulating SARS: Small-World Epidemiological Modeling and Public Health Policy Assessments* [2], the authors proposed a novel small-world model that makes use of cellular automata with the mirror identities (a miniature representation of frequently visited places to acknowledge human long-distance movement and geographic mobility) of daily-contact social networks to simulate epidemiological scenarios. Specifically, the model was used to simulate the dynamics of SARS transmission in Singapore, Taipei, and Toronto, and to discuss the effectiveness of the respective public health policies of those cities. In their system, they've constructed a large number of attributes for simulation, which makes the model more persuasive. Nevertheless, they've chosen their time step as the equivalent to one day in the real-world and only presented the analysis of relatively larger periods (degree of weeks to months), which is not the same as what we really concern: the real-time infection trend.

4. Goal

Based on the SEIR infection model, we aim to observe the real-time infection trend by simulating the movements and schedules of students on campus. Additionally, by simulating different situations and applying different health policies, we can evaluate their efficacy and put forth some suggestions on them.

5. Methodology and Experiments

In order to find the trend of infection, we've decided to simulate students' movements on campus. For implementation, we first constructed the campus map that the students are moving on, obtained theory values of the SEIR model, constructed the student agents, and visualized the real-time simulated situations.

5.1 Map Construction

When constructing the map, we've made some assumptions according to our personal experiences:

1. Each building on the map has single entrance.

```
class Point():
    def __init__():
        self.name # the name of this point (it can be a road on campus or a building name)
        self.position # the position of this point
        self.is_building #represent this point represents a building or not
```

2. In order to prevent students from going through the wall, our points are constructed at the intersection so that students only move straightly.
3. Each point will be connected by neighbor points, so it needs one more attributes.

```
class Point():
    def __init__():
        self.near_points
```

4. Only when students stay in the building will they be infected with COVID-19. Hence, we added an attribute to record the summation infection probability of the buildings.

```
class Point():
    def __init__():
        self.infect_prob
```

5.1.1 Point Defining

In order to avoid people moving through the wall on the campus map, we've defined numbers of intersection points on the map. The codes below are name of our points while the numbers in the comment are the indexes of POINTS list.

```
POINTS = ['管一', '舟山門', '小小福', '小木屋', '舟山路轉折點', # 0~4
          '舟山路轉折點', '長興街門', '男六', '側門', '大一女', # 5~9
          '共同', 'BICD', '正門', '椰林轉折點', '椰林轉折點', # 10~14
          '椰林轉折點', '椰林轉折點', '椰林轉折點', '椰林轉折點', '椰林轉折點', #15~19
          '椰林轉折點', '椰林轉折點', '文學院', '椰林轉折點', '椰林轉折點', #20~24
          '活大', '西門', '蒲葵轉折點', '蒲葵轉折點', '普通', #25~29
          '小福', '小椰林轉折點', '工綜', '水杉道轉折點', '鄭江樓轉折點', #30~34
          '博雅', '檀香道轉折點', '鄭江樓轉折點', '小椰林轉折點', '女九', #35~39
          '德田館', '新生', '檀香道轉折點', '後門路轉折點', '小椰林轉折點', #40~44
          '檀香道轉折點', '小椰林轉折點', '思亮館轉折點', '社科院', '霖澤館', #45~49
          '後門', '新體轉折點', '思亮館轉折點', '新體', '公館門', #50~54
          '新體門' #55
         ]
```

After defining the indexes of points, we've defined different kinds of attributes for the points. The code below is our definition of the **Point()** class:

```
class Point():
    def __init__(self, name, position, is_building,
                 near_points, dis_list, building_position, offset,
                 gcor_position, gcor_building_position, gcor_offset, unit_vec):
        self.name = name  # the name of points, it will be same with the names in POINTS list
        self.position = position  # the position on the map, which represents a numpy array [xxx, xxx]
        self.is_building = is_building  # a flag stands for that this point whether is a representative points for a building
        self.near_points = near_points  # a list to record which points are neighbor of this point
        self.dis_list = dis_list  # a list to record the distance with each points
        self.building_position = building_position  # if "is_building" is set, then this attribute will record the central posi
        self.offset = offset  # if "is_building" is set, then this attribute will record the offset of building's position
        self.infect_prob = 0  # the probability of infected for those who stays at the building
        self.gcor_position = gcor_position # geographical coordinate
        self.gcor_building_position = gcor_building_position # geographical coordinate
        self.gcor_offset = gcor_offset # geographical coordinate's offset
        self.unit_vec = unit_vec # a list records that the unitvector between neighboral points
```

Below are some functions that generate information of the attributes of points.

```
# The function to produce distance between two points by geographical coordinates
# Input : ┌─ i : the index of point
#         ├─ j : the index of point
#         └─ LOCATIONS : a list records the position of all points
#                        [k][0] for the latitude of point k
#                        [k][1] for the longitufe of point k
# Output : a distance between two points (meters)
def haversine(i,j,LOCATIONS):
    lon1 = LOCATIONS[i][1]
    lat1 = LOCATIONS[i][0]
    lon2 = LOCATIONS[j][1]
    lat2 = LOCATIONS[j][0]
    lon1, lat1, lon2, lat2 = map(radians, [lon1, lat1, lon2, lat2])
    # haversine公式
    dlon = lon2 - lon1
    dlat = lat2 - lat1
    a = sin(dlat/2)**2 + cos(lat1) * cos(lat2) * sin(dlon/2)**2
    c = 2 * asin(sqrt(a))
    r = 6371 # the radius of earth (km)
    return c * r * 1000
```

```
# The function to compute all pairs of shortest path
# A algorithm is called by Floyd_Warshall
# Input : ┌─ distance_list : a global list
#         └─ LOCATIONS : a list records the position of all points
#                        [k][0] for the latitude of point k
#                        [k][1] for the longitufe of point k
def Floyd_Warshall(distance_list,LOCATIONS):
    for i in range(0,len(LOCATIONS)):
        distance_list[i][i] = 0
    for k in range(0,len(LOCATIONS)):     # try every relay points
        for i in range(0,len(LOCATIONS)): # calcuate point i and point j
            for j in range(0,len(LOCATIONS)):
                if (distance_list[i][k] + distance_list[k][j] < distance_list[i][j]):
                    distance_list[i][j] = distance_list[i][k] + distance_list[k][j]
```

```
# The function to calculate the unit vector for one point to another point
# Input : ┌─ i : the index of start point
#         ├─ j : the index of end point
#         └─ LOCATIONS : a list records the position of all points
#                        [k][0] for the x coordinate of point k
#                        [k][1] for the y coordinate of point k
# Output : a unit vector for one point to another point ([xxx,xxx])
def unitvector(i, j, LOCATIONS):
    x1 = LOCATIONS[i][0]
    y1 = LOCATIONS[i][1]
    x2 = LOCATIONS[j][0]
    y2 = LOCATIONS[j][1]
    d = math.sqrt(pow(x2-x1,2) + pow(y2-y1,2))
    v = numpy.array([(x2-x1)/d, (y2-y1)/d])
    return v
```

```
# Some defined parameters
BUILDINGS = ['德田館', '文學院', '管一',
             '工綜', 'BICD', '社科院',
             '霖澤館', '新生', '博雅',
             '普通', '共同', '新體',
             '女九', '男六','大一女',
             '活大', '小福', '小小福'] #BICD生傳系館

BUILDINGS_LOCATION = numpy.array(
            [[10500,11300], [4850,9500], [6200,4000],
               [9200,10700], [5800,6950], [11200,13000],
               [13000,12800], [6850,11650], [4500,10628],
               [4700,10128], [5600,6600], [2900,14050],
               [7850,11550], [14700,8200], [2400,5850],
               [8700,9500], [5500,10128], [5200,6400]
              ])
BUILDINGS_LOCATION = BUILDINGS_LOCATION.astype(float)

BUILDINGS_OFFSET = numpy.array(
            [[200,300], [800,200], [600,400],
              [700,300], [300,300], [700,300],
              [300,400], [300,300], [400,200],
              [500,200], [200,150], [600,500],
              [300,150], [500,400], [250,250],
              [300,500], [150,200], [100,100]
             ])
```

```
BUILDINGS_OFFSET = BUILDINGS_OFFSET.astype(float)
LOCATIONS = numpy.array(
            [[6850,4600], [3300,4900], [5350,6000], [5600,6150], [6100,6400],
            [8800,7500], [12700,9100], [14400,7800], [10200,5700], [2800,6100],
            [6250,6600], [6250,6950], [1800,8550], [2500,8250], [3850,8250],
            [5750,8250], [6250,8250], [7350,8250], [8250,8250], [8700,8250],
            [2500,8750], [3850,8750], [4850,9050], [5750,8750], [7350,8750],
            [8700,8750], [1500,9850], [2500,9850], [3850,9850], [4700,9850],
            [5500,9850], [7350,9850], [9200,10050], [10200,10050], [12350,10000],
            [3850,10628], [10200,10700], [12350,10700], [7350,11300], [7850,11300],
            [10200,11300], [7350,11650], [10200,11800], [12350,11800], [7350,12500],
            [10200,12500], [7350,13000], [7850,13000], [10200,13000], [12350,12800],
            [12350,13400], [3850,13500], [7850,13500], [3850,14050], [1600,6200],
            [3850,14700]
            ])
LOCATIONS = LOCATIONS.astype(float)
                      .
                      .
                   omitted
                      .
                      .
near_list = [[3], [2,9], [1,3], [0,2,4], [3,5,10], #0~4
             [4,6,8,18], [5,7,34], [6], [5], [1,54], #5-9
             [4,11], [10,16], [13], [12,14,20], [13,15,21], #10-14
             [14,16,23], [11,15,17], [16,18,24], [5,17,19], [18,25], #15-19
             [13,21,27], [14,20,22,28], [21,23], [15,22,24,30], [17,23,25,31], #20-24
             [19,24,32], [27], [20,26,28], [21,27,29,35], [28,30], #25-29
             [23,29,31], [24,30,32,38], [25,31,33], [32,34,36], [6,33,37], #30-34
             [28,51], [33,37,40], [34,36,43], [31,39,41], [38,40], #35-39
             [36,39,42], [38,44], [40,43,45], [37,42,49], [41,45,46], #40-44
             [42,44,48], [44,47], [46,52], [45], [43,50], #45-49
             [49], [35,52,53], [47,51], [51,55], [9],#50-53
             [53]
            ]
```

5.1.2 Map Defining

So far, we've defined the **Point()** class in section 5.1.1. Next, we constructed a **Map()** class that records every point on it:

```
class Map():
    def __init__(self):
        self.point_list = [] # a list recording every point we've defined in section 5.1.1
    def add(self,point):
        self.point_list.append(point)
```

5.2 Theory Model

1. Probability of Infection:

Probability of infecting others:

In our simulation model, the probability of a student infecting others per minute is $P = p_0 * F * (1 - M * W) * S$, where p_0 is the basic probability of a student with COVID-19 infecting others and is calculated by the below methods:

$R_0 = C_{rate} * N_{meet} * I_{virus}$ [4]

- R_0 is the Basic reproduction number, there is plenty of estimation on the Internet, so we pick an average number of 3.5.
- C_{rate} is the contagious rate per contact.
- N_{meet} is the average number of students that a person may contact. Based on our personal experience, the number of people we contact is about 50 a day.
- I_{virus} is the serial interval time of COVID-19, which is approximately 3.4 days.

Thus, we calculate C_{rate} by $R_0/(N_{meet} * I_{virus})$, and get $C_{rate} = 0.0205$

- $C_{rate}$is the probability of a student might get infected after contact once. In our model, contact time is a continuous 2 classes and which is 2 hours in total.

Thus we calculated p by $(1-p)^{120} = 1 - C_{rate}$, and have $p = 1.70834 * 10^{-4}$.
$\Rightarrow P = 1.70834 * 10^{-4} * F * (1 - M * W) * S$

- $F = 1$ if the student is infectious, otherwise 0.
- $W = 1$ if the student is wearing a mask, otherwise 0.
- M is the protective efficiency of the mask preventing us from the virus.
- $S = 1$ if the student is symptomatic, otherwise 0.5.

Probability of being infected:

The probability of a student being infected every minute is $P_{total} * (1 - W * M)$.

- P_{total} is the sum of the P value for every student in the same buildings.
- $W = 1$ if the student is wearing a mask, otherwise 0.
- M is the ability of the mask preventing us from the virus.

2. Infection State Periods

1. The incubation period:

   We suppose that every person infected COIVD-19 will eventually become infectious, and the average incubation period is 5 days. The day one becomes infectious is picked by $N(5, 0.5)$ and will pick again if the number after rounding to nearest even is larger than 14 or smaller than 2.

```
def getRandomIncubationPeriod():
  d = round(np.random.normal(loc=5, scale=0.5))
    while d < 2 or d > 14:
      d = round(np.random.normal(loc=5, scale=0.5))
  return d
```

2. The latent period:

   The latent period will be 1~3 days before incubation. Thus we pick a random number by $N(2, 0.5)$ and will pick again if the number after rounding to nearest even is larger than 3 or smaller than 1. Then the latent period is the incubation period - the number.

```
def getRandomIncubationPeriod(incubation_period):
  d = round(np.random.normal(loc=2, scale=0.5))
    while d < 1 or d > 3:
      d = round(np.random.normal(loc=2, scale=0.5))
  return incubation_period - d
```

3. The infectious period:

   By studies, the infectious period is mostly 8~10 days after the latent period. Thus we pick a random number by $N(9, 0.5)$ and will pick again if the number after rounding to nearest even is larger than 10 or smaller than 8.

```
def getRandomIncubationPeriod(incubation_period):
  d = round(np.random.normal(loc=9, scale=0.5))
    while d < 8 or d > 10:
      d = round(np.random.normal(loc=9, scale=0.5))
  return incubation_period - d
```

4. The illness period:

By studies, the illness period is mostly 9~10 days after the incubation period if he/she is symptomatic. Thus we pick a random number by $N(9, 0.25)$ and will pick again if the number after rounding to nearest even is larger than 10 or smaller than 9.

```
def getRandomIncubationPeriod(incubation_period):
  d = round(np.random.normal(loc=9.5, scale=0.25))
    while d < 9 or d > 10:
      d = round(np.random.normal(loc=9.5, scale=0.25))
  return incubation_period - d
```

3. Probability of Showing Symptomatic:

Since we did not find any information online, we picked the probability of showing symptomatic in the case of cold which is 0.63.

4. Probability of Death:

We supposed that a person might die during the infectious period and the average days of the infectious period are **9** days. In addition, the probability of Death is $43.5/794 = 0.05479$, which is the number of death and infected these days.

Thus, the probability of death P_D that a student dies in one of these days can be calculated by $(1 - P_D)^9 = (1 - 43.5/794)$, so $P_D = 0.006241$.

5.3 Student Agent Construction

We first defined the states based on the SEIR model and our personal experiences, constructed the **Student()** for simulation based on the **Schedule()** class and the **HealthState()** class, and built students' movement and infection models.

5.3.1 States Defining

We defined two kinds of states for each student describing his/her current health state based on the SEIR model and his/her schedule of the whole week and created two classes to maintain them: the **HealthState()** class and the **Schedule()** class.

```
HEALTH_STATES = ['SUSCEPTIBLE', 'EXPOSED', 'INFECTIOUS', 'RECOVERED', 'DEAD']
SCHEDULE_STATES = ['IDLE', 'INCLASS', 'MOVING', 'NULL']
```

5.3.2 Schedule Class Construction

When constructing the schedule for each student, we made some assumptions according to our personal experiences:

1. Students only have classes between 8:10 and 18:25:

```
CLASS_START_TIME = ['08:10', '09:10', '10:20', '11:20', '12:20', '13:20', '14:20', '15:30', '16:30', '17:30', '18:25']
CLASS_END_TIME = ['09:00', '10:00', '11:10', '12:10', '13:10', '14:10', '15:10', '16:20', '17:20', '18:20', '19:15']
```

2. Students only enter the school at the following start points:

```
START_POINTS_ID = [
  12, # 正門
  26, # 西門
  50, # 後門
  8,  # 側門
  1,  # 舟山門
  6,  # 長興街門
  54, # 公館門
  55, # 新體門
  7,  # 男六
  9   # 大一女
]
```

3. The institutes of students only consist of the following:

```
INSTITUTES_NAME = ['資工', '外文', '工管', '機械', '生傳', '政治', '法律']
```

4. Each class is only held at the following buildings:

```
BUILDINGS_ID = [
  41, # 新生
  35, # 博雅
  29, # 普通
  10, # 共同
  53, # 新體
  99  # 系館
]
INSTITUTES_ID = [
  40, # 德田館
  22, # 文學院
  0,  # 管一
  32, # 工綜
  11, # BICD
  48, # 社科院
  49  # 霖澤館
]
```

5. The probabilities of students having a class at the buildings mentioned above are as follows:

```
BUILDINGS_PROB_WEIGTS = [
  0.15, # 新生
  0.15, # 博雅
  0.15, # 普通
  0.15, # 共同
  0.05, # 新體
  0.35  # 系館
]
```

6. The probability of students having a class at their own institutes' building is 0.5; otherwise they'll have a class at the buildings with the probabilities mentioned above.
7. Each class lasts for two continuous hours.
8. Each student only have lunch at the following locations randomly chosen:

```
RESTAURANTS_ID = [
  25, # 活大
  30, # 小福
  2,  # 小小福
  39, # 女九
  12, # 正門
  26, # 西門
  50, # 後門
  8,  # 側門
  1,  # 舟山門
  6,  # 長興街門
  54, # 公館門
  9   # 大一女
]
```

9. The probability of students having a class at each timestamp in CLASS_START_TIME is 0.5.
10. Students only have lunch at the three start timestamps around noon (11:20, 12:20, 13:20); otherwise, they don't have lunch at restaurants and nor is any restaurant be arranged.
11. Students don't have dinner on campus. When their last class of the day is over, they stay for 15~30 minutes on campus and then leave the campus.
12. Since there are no classes at the weekend, we only simulated the trend on weekdays.

The **Schedule()** class is constructed as follows:

```
class Schedule:
  def __init__(self, gender, instituteIdx):
    self.startPointID = self.getRandomStartPointID(gender) # student's start point ID
    self.destPointsID = [] # Destination points ID list of one day
    self.destTimes = [] # Timestamp list for each destination point of one day

    self.arrangeSchedule(instituteIdx) # The function to arrange each student's schedule
    self.arrangeRestaurant() # The function to arrange a restaurant for each student's lunch

    self.newDayInit(0) # The function to do some initialization at the beginning of a new day
```

The custom initialization function when a new day begins:

```
def newDayInit(self, day):
    self.startTime = Time.getRandomTimeStamp(Time.addMinutes(self.destTimes[day][0], -30), Time.addMinutes(self.destTimes[da
    self.endPointID = self.startPointID
    self.endTime = Time.getRandomTimeStamp(Time.addMinutes(self.destTimes[day][-1], 65), Time.addMinutes(self.destTimes[day]
    self.nextDestIdx = 0
    self.numDestPoints = len(self.destPointsID[day])
```

The pseudo code of the function to arrange each student's schedule:

```
def arrangeSchedule(instituteIdx):
    from (Monday) to (Friday):
        for (each timestamp) in (CLASS_START_TIME):
            if (student has a class at this timestamp (prob >= 0.5)):
                Add two continuous classes in the destPointsID list
```

The pseudo code of the function to arrange a restaurant for each student's lunch:

```
def arrangeRestaurant():
    from (Monday) to (Friday):
        Find the first class whose start time exceeds 12:20
        if (the class doesn't start at 12:20):
            Append a random restaurants into the destPointsID list
            Append the timestamp 12:20 into the destTime list
        elif (the class doesn't start at 13:20):
            Append a random restaurants into the destPointsID list
            Append the timestamp 13:20 into the destTime list
        elif (the class doesn't start at 11:20):
            Append a random restaurants into the destPointsID list
            Append the timestamp 11:20 into the destTime list
```

The pseudo code of some utility functions:

```
def getRandomStartPointID(gender):
    if gender == 'Male':
        return a random start point ID excluding '大一女'
    else:
        return a raondm start point ID excluding '男六'

def getRandomDestPointID():
    return a random ID from the BUIDLINGS_ID list
```

5.3.3 Health State Class Construction

Based on the SEIR model introduced in 5.2, we constructed the **HealthState()** class as follows:

```
class HealthState:
    def __init__(self, healthState):
        self.state = healthState # a state in HEALTH_STATE
        self.incubationPeriod = SEIR_Model.getRandomIncubationPeriod() # 潛伏期
        self.latentPeriod = SEIR_Model.getRandomLatentPeriod(self.incubationPeriod) # 潛藏期
        self.infectiousPeriod = SEIR_Model.getRandomInfectiousPeriod() # 感染期
        self.illnessPeriod = SEIR_Model.getRandomIllnessPeriod() # 發病期
        self.infectedDays = 1 if healthState == 'INFECTIOUS' else 0 # days since student has entered the INFECTIOUS state
        self.illnessDays = 0 # days that student has been symptomatic
        self.currentProb = SEIR_Model.ASYMPTOMATIC_TRANS_PROB if healthState == 'INFECTIOUS' else 0 # current probability th
        self.quarantined = False # if student is being quarantined
        self.wearingMask = True if random.random() <= SEIR_Model.WEARING_MASK_PROB else False # if student is wearing a mas
```

The pseudo code of the function to check student's state at the beginning of a new day:

```
def newDayCheckState():
    if state == 'SUSCEPTIBLE' or 'RECOVERED' or 'DEAD':
        return

    infectedDays += 1
    if student is symptomatic:
        illnessDays += 1

    Check whether the state is changed from 'EXPOSED' to 'INFECTIOUS'
    Check whether the state is changed from 'INFECTIOUS' to 'DEAD'
    Check whether the state is changed from 'INFECTIOUS' to 'RECOVERED'

    if the incubation period ends:
        if student becomes symptomatic:
            currentProb = SEIR_Model.SYMPTOMATIC_TRANS_PROB
            illnessDay = 1
            Determine whether the student is quarantined
        else:
            currentProb = SEIR_Model.ASYMPTOMATIC_TRANS_PROB

    elif student has conquered the virus and has no more symptoms:
        illnessDays = 0
        if quarantined:
            quarantined = False
```

5.3.4 Student Class Construction

The **student()** class is constructed as follows:

```
class Student:
  def __init__(self, healthState='SUSCEPTIBLE'):
    self.gender = 'male' if random.choice([0, 1]) == 0 else 'female'
    self.instituteIdx = random.randint(0, len(INSTITUTES_NAME)-1) # index of student's institute
    self.healthState = HealthState(healthState)
    self.scheduleState = 'NULL' # a state in SCHEDULE_STATE
    self.schedule = Schedule(gender=self.gender, instituteIdx=self.instituteIdx)
    self.currentPointID = self.schedule.startPointID # the ID of current point which the student stands on currently; if stu
    self.nextPointID = -1 # the ID of the next point on the path that student is moving to
    self.currentPosition = (MAP.point_list[self.schedule.startPointID].position).copy() # shallow copy
    self.currentSpeed = 0.0 # the speed at which the student is currently moving
    self.currentDirection = np.array([0.0, 0.0]) # the direction the student is currently moving towards to
```

The custom initialization function called at the beginning of a new day:

```
def newDayInit(self, day):
    self.schedule.newDayInit(day)
    self.healthState.newDayCheckState()
```

5.3.5 Movement & Infection Model Construction

When constructing the movement & infection model, we made some assumptions.

About movement:

1. Students move at a speed which is a constant MOVING_SPEED plus a uniform random value from -50 to 50.

2. Students only move when they have a class/have lunch/is going to leave campus in 15 minutes; otherwise, they simply stay at the current point.
3. Students only move in straight line and always follow the shortest path to the next destination when moving.
4. Students stay in the current building when their next class is held in the same building.
5. When students are in class, they are randomly distributed in the building.

About Infection:

1. Students get infected only when they are in the same building as an infectious student is, i.e. students won't be infected when they are moving.
2. The probability of infection when several infectious students are in the same building can be summed up. For example, when the probability of transferring the virus equals *p*, and two infectious students are in the same building, the probability of a susceptible student staying at the same building becomes *2p*.
3. The change of the health states from 'EXPOSED' to 'INFECTIOUS', 'INFECTIOUS' to 'RECOVERED', and 'INFECTIOUS' to 'DEAD' only happens at the beginning of a new day.

The pseudo-code of the model for simulating student's movement is constructed as follows:

```
def move(day):
    delta_distance = the distance each time unit student moves
    while delta_distance > 0:
        if delta_distance >= left distance to the next point on the path:
            delta_distance -= left distance to the next point on the path
            Update currentPointID and currentPosition
            Find the next point on the path to the destination
            Update currentDirection
            if student has reached the destination building/end point:
                break
        else:
            Set currentPoint to -1 (which means not standing on a point)
            Update currentPosition
```

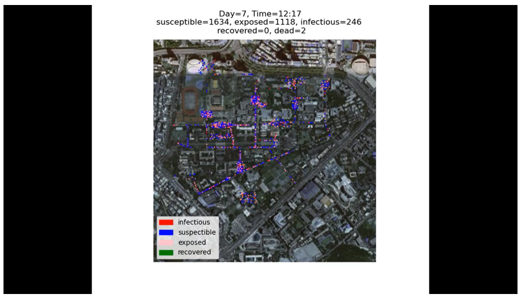
The pseudo-code of the model for simulating student's action at each time unit is constructed in the following functions:

```
def Action(CURRENT_TIME, day):
    if healState == 'DEAD' or student is quarantined:
        return

    if student reaches the campus:
        Change scheduleState to 'IDLE'
        Initialize the currentPointID and currentPosition

    if scheduleState == 'IDLE':
        if student has a class in 15 minutes or is going to leave campus:
            Change scheduleState to 'MOVING'
            Find the next point on the path to the next destination
            Set up currentSpeed and currentDirection

    elif scheduleState == 'MOVING':
        move(day)
        if student has reached the destination building/end point:
            if student has reached the destination building:
                Change scheduleState to 'INCLASS'
                Set currentPosition, currentSpeed, and currentDirection

                if healthState == 'INFECTIOUS':
                    Update the summation infectoin probability of the building
                    Determine whether student is infected at this time unit

            else: # student is going to leave campus
                Change scheduleState to 'NULL'
                return

    elif scheduleState == 'INCLASS':
        if CURRENT_TIME in CLASS_END_TIME:
            if student has a class in 15 minutes:
                if the next class is held at another building:
                    Find the next point on the path to the next destination building
                    Set up currentPosition and currentDirection

                    if healthState == 'INFECTIOIS':
                        Update the infect_prob of the building

            else:
                Change scheduleState to 'IDLE'

                if healthState == 'INFECTIOUS':
                    Update the infect_prob of the building

        else: # in class
            if infect_prob of the building > 0
                Determine whether student is infected at this time unit based on infect_prob and whether he/she is wearing
```

5.4 Visualizing

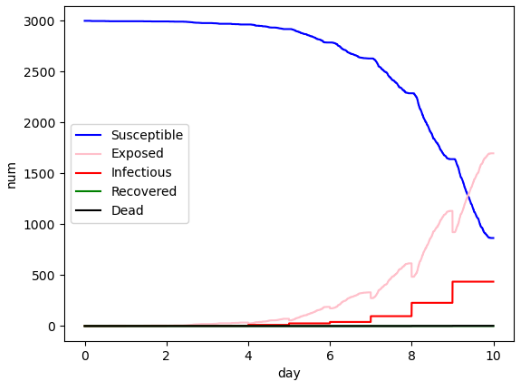
We used the python package "matplotlib" to visualize our simulation. The picture on the right side is a screenshot of one of our cases of the simulations. We'll provide the simulation videos in the appendix.

In our videos, each dot represents a student, while the color of the dot represents his/her state of health. The text showing on the top of shows the realtime infection condition.



In addition to animation, we recorded the number of students in each healthy state at different timestamps and drew some graphs as shown on the right.

In the graph, each color of the line represents the trend of the total number of students with that health state.



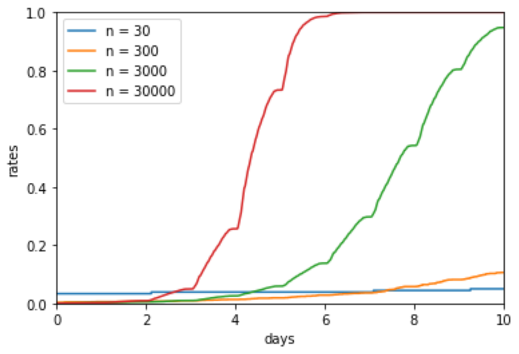
6. Results and Analysis

6.1 Realtime Simulation

We recorded the realtime simulation of different situations described in the following sections to obtain the in-time trend of infected students. All videos are provided in the appendix.

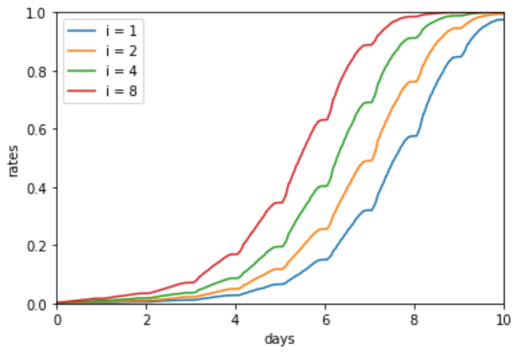
6.2 Various Number of Total Students

By comparing the different number of total students, we found out that the rate of infectious students over total students grows faster as the number of total students increases. In our opinion, the bigger the number of students is, the denser the buildings are, resulting in the higher probability of students getting infected.



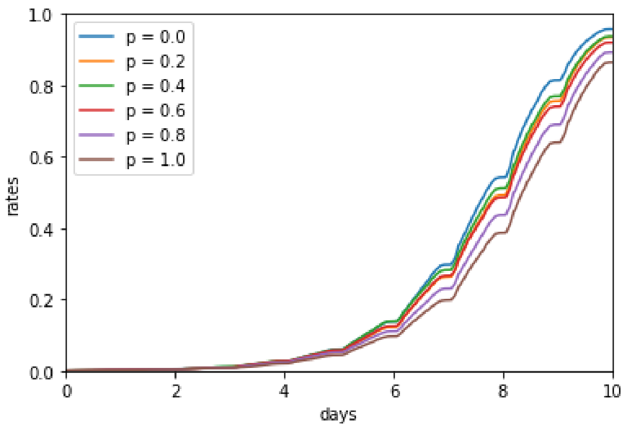
6.3 Various Number of Initial Infectious Students

By comparing different numbers of infectious students at the beginning, we found out that the more infectious students there are at the beginning, the faster the infected rate climbs.



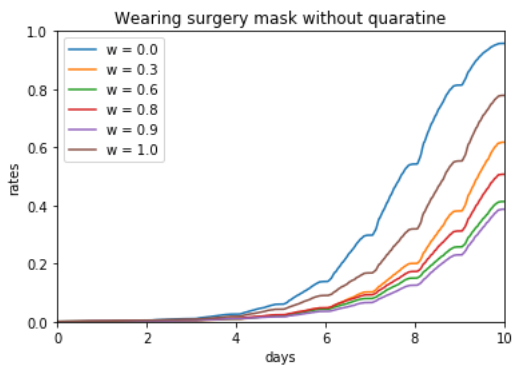
6.4 Various Policy: Quarantine

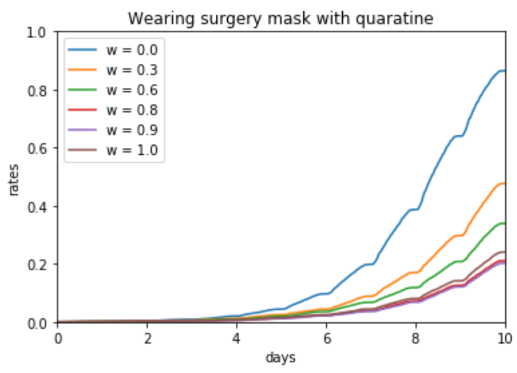
A person with symptoms will face a probability to be quarantined. However, by the simulations below, we found out that quarantining people after he/she showed symptoms is too slow. Even all of them were quarantined, people without symptoms could still have the ability to infect the rest of the students.

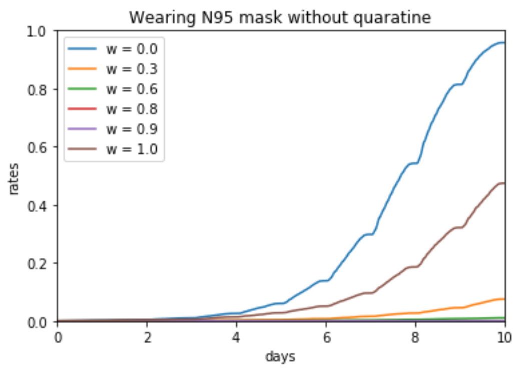


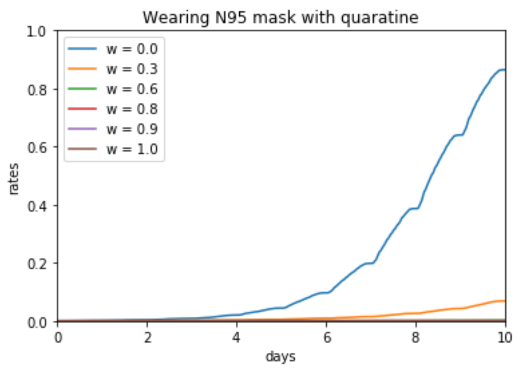
6.5 Various Policies: Different Masks

From [5], we obtained the protective efficiency of the surgical mask to be 0.292 while the N95 mask to be 0.915. Thus, we ran simulations on different probabilities of people wearing masks under four situations (every symptomatic person will be quarantined or all of them will not be quarantined) x (surgery mask or N95 mask).









In our expectation, the rate will increase as the rate of people wearing masks decreases. However, the results were different. After researching in the situations, we found out that the first batch of infectious students is more important. If there was an epidemic prevention flaw, no matter how high the mask-wearing rate was, it couldn't slow down the trend significantly. As a result, we think that it is of more importance to lock down the school rather than requiring students to wear masks if there are infectious students on campus.

7. Conclusion

After simulating several conditions, we found out that:

1. It's crucial for us to avoid going to dense places, and thus social distancing indeed works.
2. The best time to prevent the outbreaks is before the infectious number grows since the more infectious patients there are, the quicker the infectious rate climbs.
3. Wearing masks does help, and the N95 mask can significantly reduce the infectious rate than the surgical mask.
4. Quarantine the symptomatic people is not enough; those who have met the patients should also be quarantined.
5. Whether the outbreak occurs or not sometimes depends on our luck :)

8. Appendix

Simulation videos:

https://drive.google.com/drive/folders/1J03-oILftvFV8iIN3jQEJecRfY-EubrJ?usp=sharing

Github:

https://github.com/asdzx5812/AI_Final

9. References

[1] 《利用整合式傳染病監測軟體以協助流行性感冒的監測並提供公衛參考》行政院衛生署疾病管制局九十五年度科技研究發展計畫 https://www.cdc.gov.tw/uploads/files/24309cc6-96a4-42b7-84ab-9f10794081f7.pdf

[2] *Simulating SARS: Small-World Epidemiological Modeling and Public Health Policy Assessments*, Journal of Artificial Societies and Social Simulation vol. 7, no. 4 http://jasss.soc.surrey.ac.uk/7/4/2.html

[3] https://theconversation.com/how-long-are-you-infectious-when-you-have-coronavirus-135295

[4] http://www.publichealth.org.tw/upload/files/第三堂課　新冠肺炎之傳播分析(1).pdf

[5] https://journals.sagepub.com/doi/pdf/10.1177/153567601001500204?fbclid=IwAR1o6ynjqvgKXCoEcEtdhW8ObYjLprakFNuCMPXV_oHzQWTDnac2jC0SsG0