Report for simSpread.py

1. **Abstract –** The following report addresses the features of simSpread.py, discusses what were the results observed after using the parameter values, and further uses those results to evaluate what was outcome of the simulation after it had been run. The parameter values used to execute the program were:

   a) Default values –
   - Population = 100
   - Infected people = 5
   - Neighborhood = Moore
   - Airports used in the simulation = 5

   b) First set of values –
   - Population = 20
   - Infected people = 2
   - Neighborhood = Von-neumann
   - Airports used in the simulation = 3

   c) Second set of values –
   - Population = 250
   - Infected people = 12
   - Neighborhood = Moore
   - Airports used in the simulation = 9

2. **Background –** The simulation was coded in the first place to show the spread of disease from one person to another, and further provide statistics through scatterplots that display how many people have been infected and how many are still immune to the disease. Afterwards, additional behaviors were later added in the code to state how many people have died from the illness, how many have recovered, and how many have travelled long distances using different airports whilst the spreading of this disease. To add, I had chosen the above 2 sets of values as parameters, in addition to the default values, to test and judge the difference of the results when the simulation program is run. I wanted to use extreme values, wherein the first set would carry out the simulation on small population of 20 people whilst using the Von-neumann neighborhood, whereas the second set would be carried out on 250 people and use Moore neighborhood instead; I wanted to interpret how many people get infected, how many die, how many recover and how many remain immune when the spread of disease is between a small sized population, and later, define a larger population size to carry out a comparison on how quickly, and how much does the disease spread when the count of individuals is higher.

3. **Methodology –** Firstly, I had made use of the default values of population, infected people, and total airports whilst choosing Moore/ Von-neumann neighborhood to test if the simulation actually progressed and worked correctly; I wanted to ensure that the program primarily asks the user to input the neighborhood as per as their choice, and once it is entered, it should implement the neighborhood on the grid accordingly and allow the individuals to move along the map as defined in the code. Without the input of the neighborhood, the simulation **will not** proceed.

Once done, python3 simSpread.py is called and the simulation is run using the default quantities of **100 (Population), 5 (InfectedPeople), and 5(AirportNumber).**

To carry out the comparisons, I used two extra sets of values, wherein I had made use of parameter sweeps for both times to run the simulation based on different values of <POPINIT> <INFECTEDINIT> <NEIGHBORHOOD> <totalAirports>

a. First simulation run using parameters as followed:

   Python3 simSpread.py 20 2 von-neumann 3

b. Second simulation run using parameters as followed:

   Python3 simSpread.py 250 12 moore 9

As briefed earlier, the set of values used above was to carry out an investigation on how the scatterplot output and displayed would differ when different range of values i.e. small and a large population number would be used; It can be concluded that the larger the size of the population, the more the disease would spread, and the more the people would get infected. However, you cannot ensure if the death or recovery rate of the individuals would always be a high ranged number, or lower than estimated.

➔ **<u>To run the simulation in all, the following extensions had been added into the python file:</u>**

(Extension-1) : The first snap of code below classifies a function for the movement of people across the grid, which is basically creating the neighborhoods variables. The second snap, in addition, just creates a function to prompt the user input of the neighborhood they choose, accept or decline the input, and provide an output either by running the movePeeps method, or asking for user input again if a wrong or invalid neighborhood has been entered using the if-else loop encoded in findingNeighbourhood method.

```
def movePeeps(cur, next, r, c):
    #print("Pos (", r, ",", c, ") has ", cur[r,c], " people")
    num_r, num_c = cur.shape

    global neighbourhood

    for peep in range(cur[r,c]):
        if neighbourhood == "moore":
            # Moore neighbourhood code: If the user input is "moore", the below code defines how the peeps move within the map (Can move North, South, East, West, NorthEast,
            #NorthWest, SouthEast and SouthWest)
            rMove = random.randint(-1,1)
            cMove = random.randint(-1,1)

        elif neighbourhood == "von neumann":
            # Von Neumann neighbourhood code: If the user input is "von neumann", the below code defines how the peeps move within the map (Can only move North, South, East, West)
            vonMovement = random.randint(-1, 1)
            vonDirection = random.randint(0, 1)

            if vonDirection == 0:
                rMove = vonMovement
                cMove = 0
            if vonDirection == 1:
                rMove = 0
                cMove = vonMovement

         #print("Move from (", r, ",", c, ") to (", r+rMove, "," , c+cMove, ")")
        if (r + rMove) > (num_r-1) or (r + rMove) < 0:
            rMove = 0
        if (c + cMove) > (num_c-1) or (c + cMove) < 0:
            cMove = 0
        next[r + rMove, c + cMove] +=1
    #    print("          (", r, ",", c, ") to (", r+rMove, "," , c+cMove, ")")
```

```
        ## Extension-1: Creating a method that sets neighbourhood according to the user input ##

def findingNeighbourhood():
    default_neighbourhood = ["moore", "von neumann"]

    global neighbourhood

    while not neighbourhood:
        neighbourhood_entered = input("\nPlease enter a neighbourhood to set (moore or von neumann): \n")
        if neighbourhood_entered.lower() in default_neighbourhood:
            neighbourhood = neighbourhood_entered
            print(f"Your neighbourhood has been set to : {neighbourhood_entered.title()}")
        else:
            print("The neighbourhood entered is invalid")

#find neighbourhood >>> Q1
findingNeighbourhood()                  ##### The findingNeighbourhood function is called to allow user to input a neighbourhood they want to choose amongst #####
                                        # The moore and Von Neumann neighbourhoods. As per as the user input, the grid movement of people within it would change #
                                        # Wherein if the neighbourhood "von neumann" is prompted, the people can move up, down, left or right. If the "moore" #
                                        # Neighbourhood is selected, the users can move up, down, left, right, or even diagonally. If NO NEIGHBOURHOOD is entered, #
                                        # The simulation would not proceed. #
```

(Extension-2): The below code was written to define extra behavior for individuals of death and recovery, in addition to becameInfected and notInfected which were already pre-defined in the code. When the simulation is run, amongst the total population, these 2 additional behaviors would be called to notify the users the count of deaths and recoveries that have occurred due to the illness.

```
        ##### Extension-2 : Adding additional behaviour of death and recovery #####

def death(becameInfected, r, c, prob):
    for peep in range(becameInfected[r,c]):
        if random.random() < prob:
            becameInfected[r, c] -= 1
            print(f">>> Infection death ({r}, {c})")
    return becameInfected
                                        ##### The functions death and reecovery are used to provide additional behaviour of individuals #####
                                        # Who have recovered, or died from the infection, if at all infected, otherwise are still categorized as #
                                        # Individuals who are uninfected by the disease. #
def recovery(becameInfected, notInfected, r, c, prob):
    for peep in range(becameInfected[r,c]):
        if random.random() < prob:
            notInfected[r, c] +=1
            becameInfected[r, c] -=1
            print(f">>> Recovered from infection ({r}, {c})")
    return becameInfected, notInfected
```

(Extension-3): The lines of codes seen in the screenshot below were used to provide overall results after the simulation had been completed. It would print the total population of the simulation, the total no.of people that had been infected and remained uninfected, the total deaths and the total recoveries that had been made at the end of the program.

```
        ##### Extension-3: Using print command to output additional information regarding the statistics of the simulation #####
    print("                                          <<< Statistics >>>                                                    ")
    print("Total population: ", POPINIT)
    print("Total no.of uninfected people: ", uninfected.sum().sum())
    print("Total no.of infected people: ", infected.sum().sum())
    print("Total no.of deaths: ", POPINIT+INFECTEDINIT-uninfected.sum().sum()-infected.sum().sum())

print("\n                                    ##### End of Disease-Spread-Simulation #####                                  \n")
```

(Extension-4): To create a boundary for restricting the spreading of disease, the below two sets of codes were implemented; the while loop had been produced along the “plt.axhline” code to create two horizontal lines on the scatterplots as the marks are output, acting as a restriction so that the values executed by the program do not go beyond these points.

```
                    ##### Extension-4: Creating a barrier for restricting the spread of disease #####

        while rpos == Rows_range or cpos == Columns_range:
            rpos = random.randint(0, num_r-1)
            cpos = random.randint(0, num_c-1)

        grid[rpos, cpos] += 1
        #print("Adding 1 to (", rpos, ",", cpos,")")  ##### The distributePop function creates a grid with 15 columns, and 10 rows based on the pre-defined variables #####
                                                      # COLMAX = 15, and ROWMAX = 10. The conditional loops create a barrier on the simulation plot that is shown, #
                                                      # Of the disease, acting as a boundary for the simulation values; The people move randomly accross the grid/map #
                                                      # Since random values are assigned using the random.randint function, and whenever the values are output,  #
                                                      # They would always be below number 9 and 11 in order to prevent the spread of the disease #
```

```
def plotGrids(Igrid, Ugrid):
    num_r, num_c = Igrid.shape
    Ix, Iy, Icount = buildScatter(Igrid)
    plt.scatter(Ix, Iy, s=Icount, c="r", alpha=0.5)
    Ux, Uy, Ucount = buildScatter(Ugrid)
    plt.scatter(Ux, Uy, s=Ucount, c="g", alpha=0.5)
    plt.axhline(Columns_range)                    ##### The axhline function is used to draw a horizontal line on the scatter, acting as a boundary line to restrict #####
    plt.axhline(Rows_range)                       #  The exceeding of values beyond this range. #
    plt.xlim = (-1, num_c)
    plt.ylim = (-1, num_r)
    plt.show()
```

(Extension-5): The transfer of people over long distances was made possible using the airport and transferPeople methods. The random.randint function outputs random numbers for the column and row values to be used, wherein these numbers further are accounted to print the airports between which the individuals have travelled.

```
        ##### Extension-5: Creating an airport to allow the movement of people along long distances #####

def airport(no_rows, no_cols, no_airports):
    rows, columns = Rows_range*np.ones(no_airports), Columns_range*np.ones(no_airports)
    while (Rows_range in rows) or (Columns_range in columns):
        rows, columns = np.random.randint(no_rows, size=no_airports), np.random.randint(no_cols, size=no_airports)
    airports = np.c_[rows, columns]
    return airports
                                                  ##### The airport function #selects a random number output from the list, but ensure that the size is smaller #####
                                                  # than the value of no_airports variable; if true, the person is transferred from the currentAirport to the #
                                                  # nextAirport using the transferPeople function #
def transferPeople(cur, next, r, c):
    currentAirport = np.array([r,c])
    for peep in range(cur[r,c]):
        nextAirport = currentAirport
        while (nextAirport == currentAirport).all():
            nextAirport = random.choice(airports)
        cur[r,c] -= 1
        next[nextAirport[0], nextAirport[1]] += 1
        print(f">>> Transferring person from {currentAirport} airport to {nextAirport} airport.")
    return next
```

```
if ([row, col] == airports).all(axis = 1).any() and \
        random.random() < 0.4 and \
        len(airports) > 1:
    transferPeople(infected, infected2, row, col)
    transferPeople(uninfected, uninfected2, row, col)
```

```
airports = airport(ROWMAX, COLMAX, totalAirports)
```

(Extension-6): The last extension was created to allow the use of parameters, to automate the results of the simulation as per as the values that have been chosen. If, the no.of arguments are too less, or not entered at all, the system on its own will make use of the default values of the arguments, whilst acknowledging the same to the user on the terminal, and carry out the simulation program towards its end.

```
        ##### Extension-6: Creating a code to generate data either automatically, or manually through user input, by implementing a parameter sweep driver script to run the
                #  simulation with varying parameters #

if len(sys.argv) < 5:
    print('\nArguement values entered are too less! System sets default values to: POPINIT = 100 , INFECTEDINIT = 5, totalAirports = 5\n'
        'Hint: Give own arguement values using "Python3 simSpread.py <PopulationValue> <InfectedNumber> <neighbourhoodName> <Airports>')
    POPINIT = 100
    INFECTEDINIT = 5
    neighbourhood = None            ##### The following if-else statement does this: IF the user does not input any parameter values, #####
    totalAirports = 5               # The terminal would notify the user that the system will now make use of its default parameter values #
else:                               # which are populationNumber = 100, InfectedPeople = 5, neighbourhoodSet = none (The user WILL BE ASKED AND HAS TO choose #
    arguement = sys.argv            # A neighbourhood amongst moore or von neumann to proceed further in the simulation, and the no.of total airports between transfers = 5 #
    POPINIT = int(arguement[1])
    INFECTEDINIT = int(arguement[2])
    neighbourhood = arguement[3] if arguement [3] in ["moore", "von neumann"] else None
    totalAirports = int(arguement[4])
```

4. **Results –** The following would be the results when:

(a) The simulation is run on default values –

-> The terminal looks as such and asks the user to choose either neighborhood to proceed further into the simulation.

```
fateh@fateh-virtual-machine:~/FOP/FOP_Programming_19977850$ python3 simSpread.py

Arguement values entered are too less! System sets default values to: POPINIT = 100 , INFECTEDINIT = 5, totalAirports = 5
Hint: Give own arguement values using "Python3 simSpread.py <PopulationValue> <InfectedNumber> <neighbourhoodName> <Airports>

Please enter a neighbourhood to set (moore or von-neumann):
```

- After the simulation has ended using the default values, the statistics shown are as such:

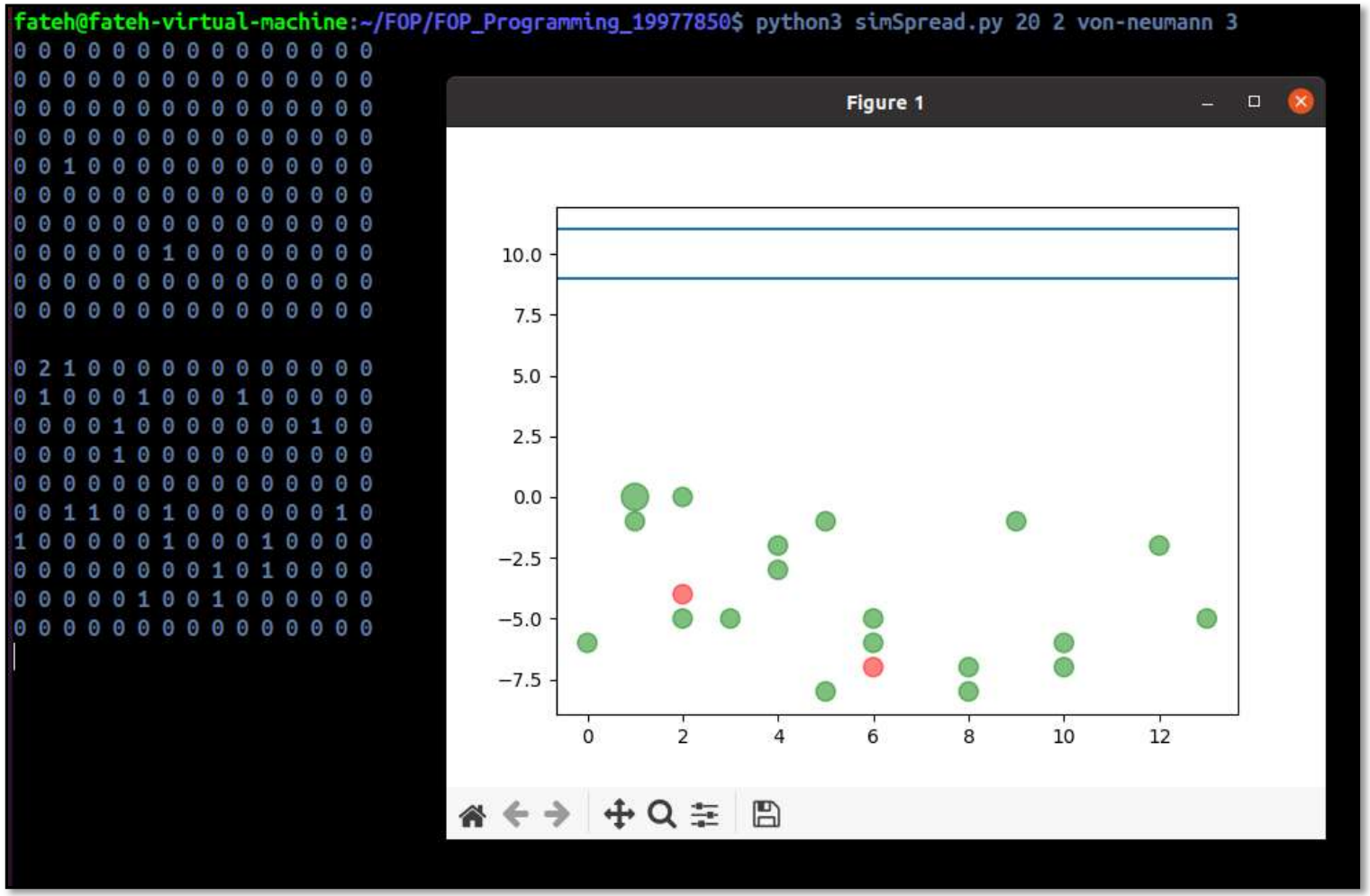
```
                                        <<< Statistics >>>
Total population:  100
Total no.of uninfected people:  94
Total no.of infected people:  0
Total no.of deaths:  11

                                    ##### End of Disease-Spread-Simulation #####
```

(b) Using first set of parameters –

-> The initial starting of the program would be as such:

- Note how the simulation has been called, using the parameters.

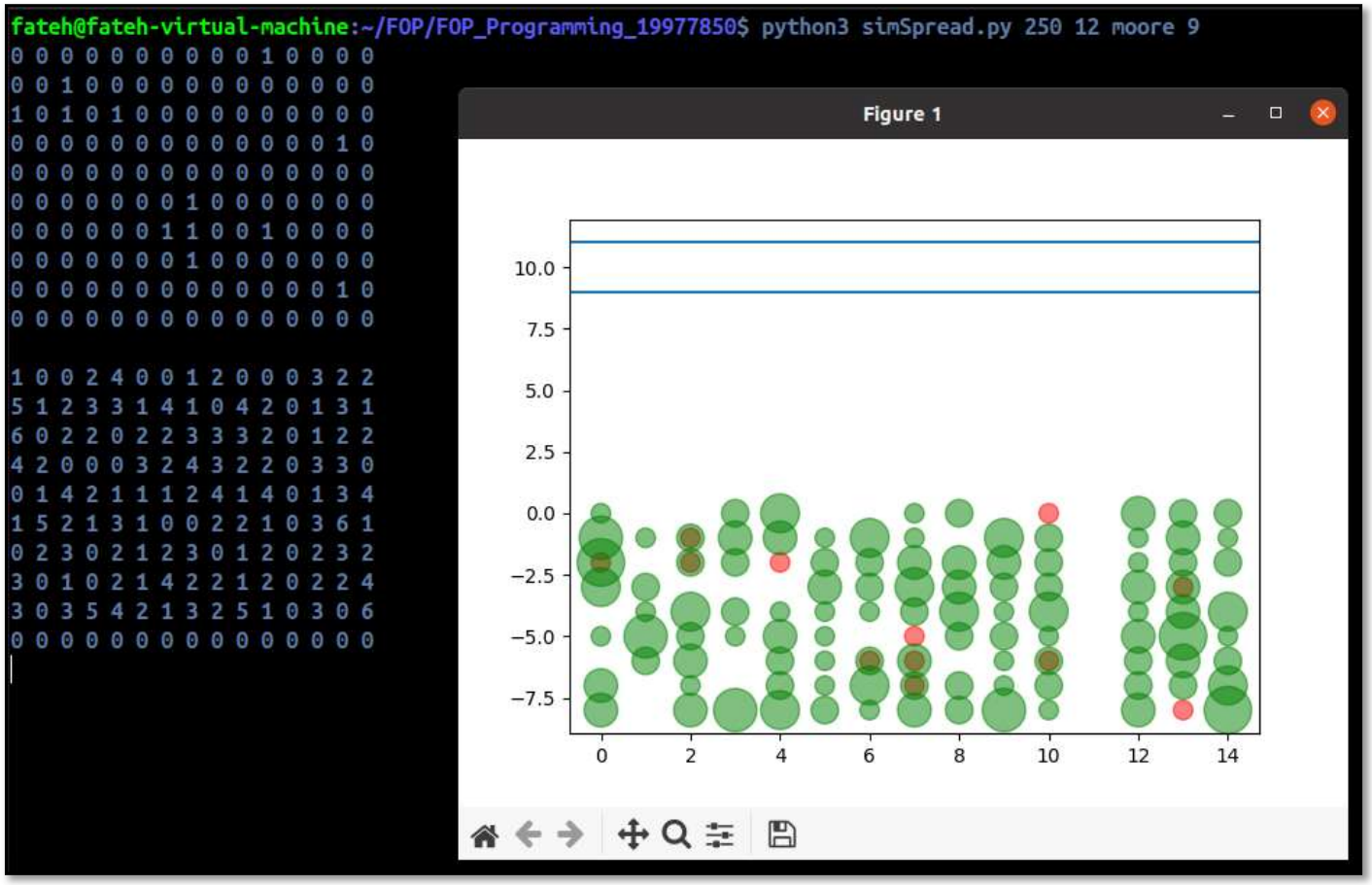


- End

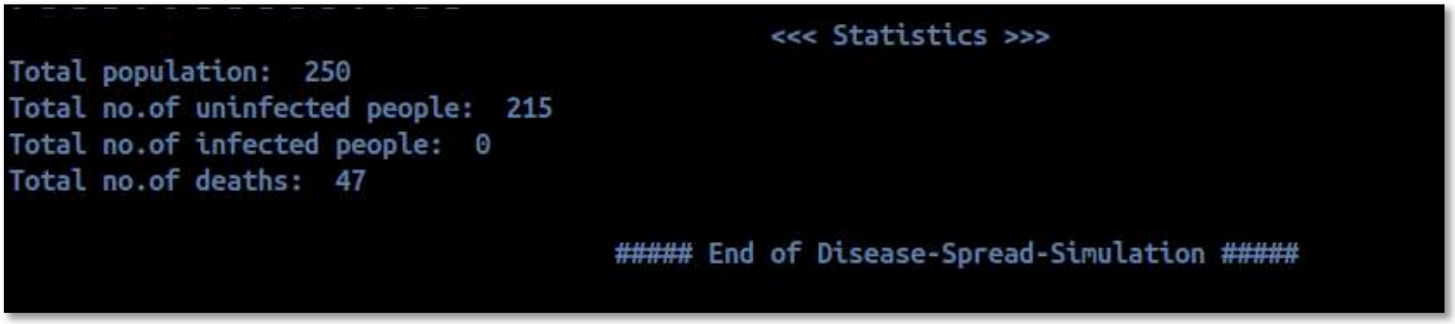

<<< Statistics >>>
Total population: 20
Total no.of uninfected people: 22
Total no.of infected people: 0
Total no.of deaths: 0
End of Disease-Spread-Simulation

(c) Using second set of parameters –

-> Start:



- End:



5. **Conclusion –** It can be expressed that as the population size of the simulation increased, the statistics values of all variables would increase proportionally; as the population increases, the total quantity of individuals that have remained uninfected would be of a higher number, total no.of infected people may or may not be higher than previously run simulations, and so would be the case for the no.of total deaths. Further investigations that we could carry out would have been, that for how long a treatment should be provided to the individuals that have been infected with the illness. Furthermore, we may also carry out an analysis that if a person has been infected with the disease and have a flight to catch, the simulation could be programmed in a way that it would delay the movement of the individual to a further date ahead. Lastly, a total cost of the treatment could be displayed to the people that have been infected with the sickness, to notify them about the total billing of all the medicines and care offered.