



Homework III

H1N1 Epidemic

Networks dynamics and learning - Politecnico di Torino

Erich Malan s267475



The homework has been carried out alone but the extra point (5) has been carried out with
course student: Davide Bussone

Results Summary and Theory

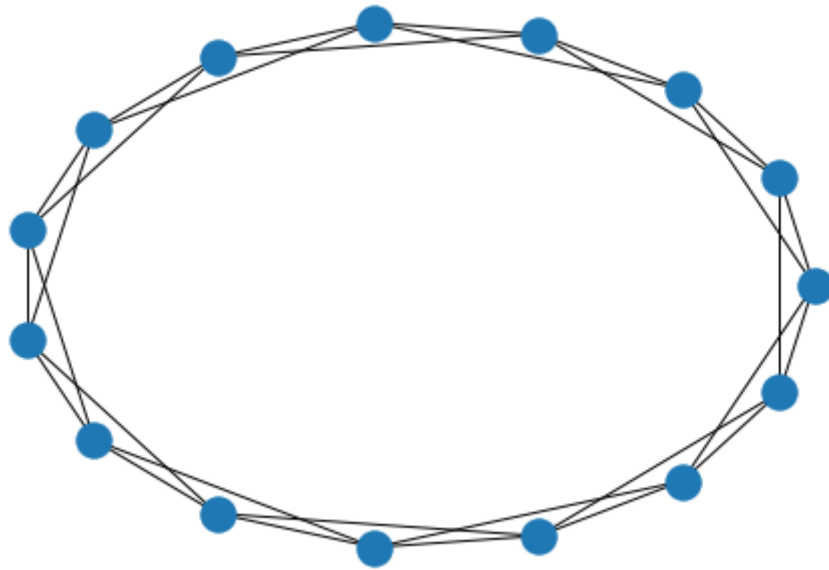
Introduction

The objective of the homework is to create pandemic models and comprehend the dynamic of the epidemic evolution depending on the hyper-parameters k , β and ρ , as well as the graph structure and the vaccination campaign. In the end an experimental solution (that comprehend an alternative type of graph definition and an alternative hyper-parameters search algorithm) is exploited in order to find another way to get similar or better results in a more random way trying to generalize the algorithm to reproduce as best as possible a general real case scenario.

Exercise 1

The objective of this exercise is to simulate a SIR epidemic on a symmetric k -regular undirected graph, this means that the possible state of a node is S (susceptible -> can be infected by a neighbor), I (infected -> can recover) or R (recovered) The graph creation exploit `nx.Graph`, and then every node is connected to the previous 2 nodes (using the sliding window to connect the beginning and the end of the array).

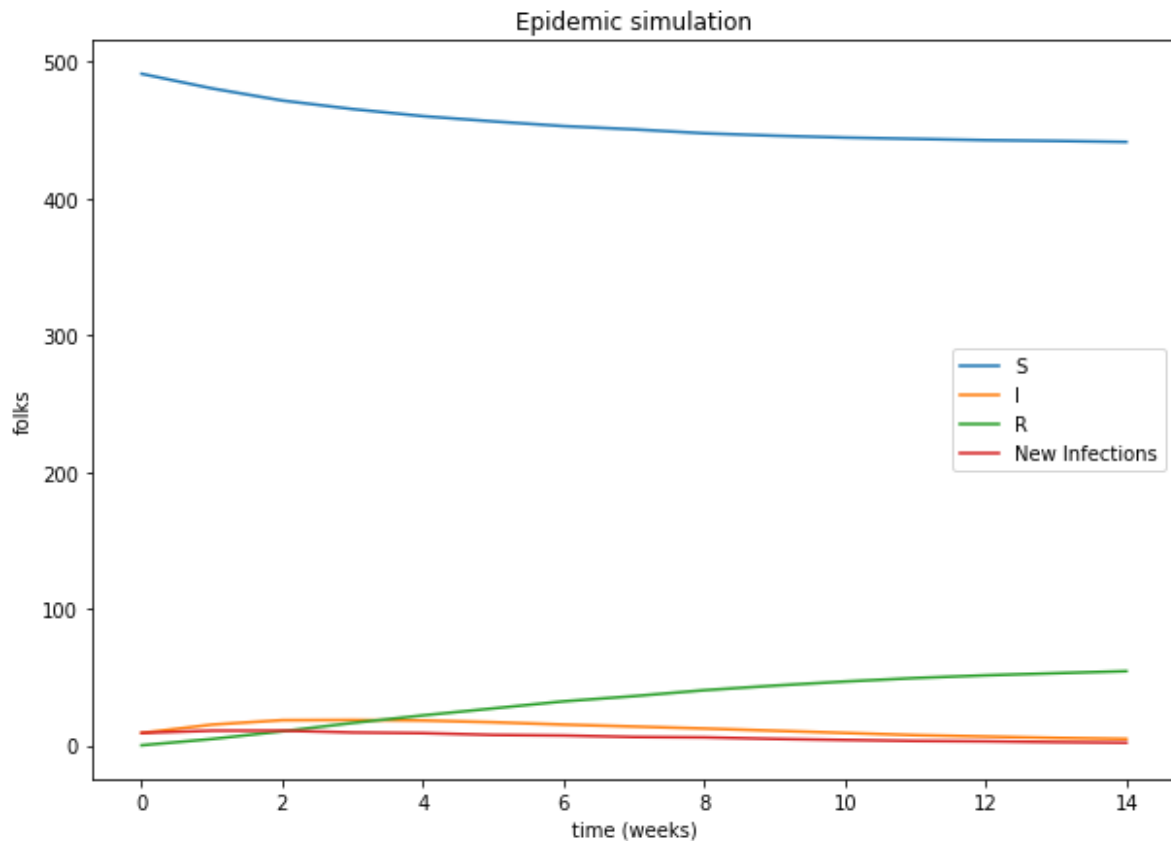
(Graph example with $N = 15$ nodes and $K = 4$ links)



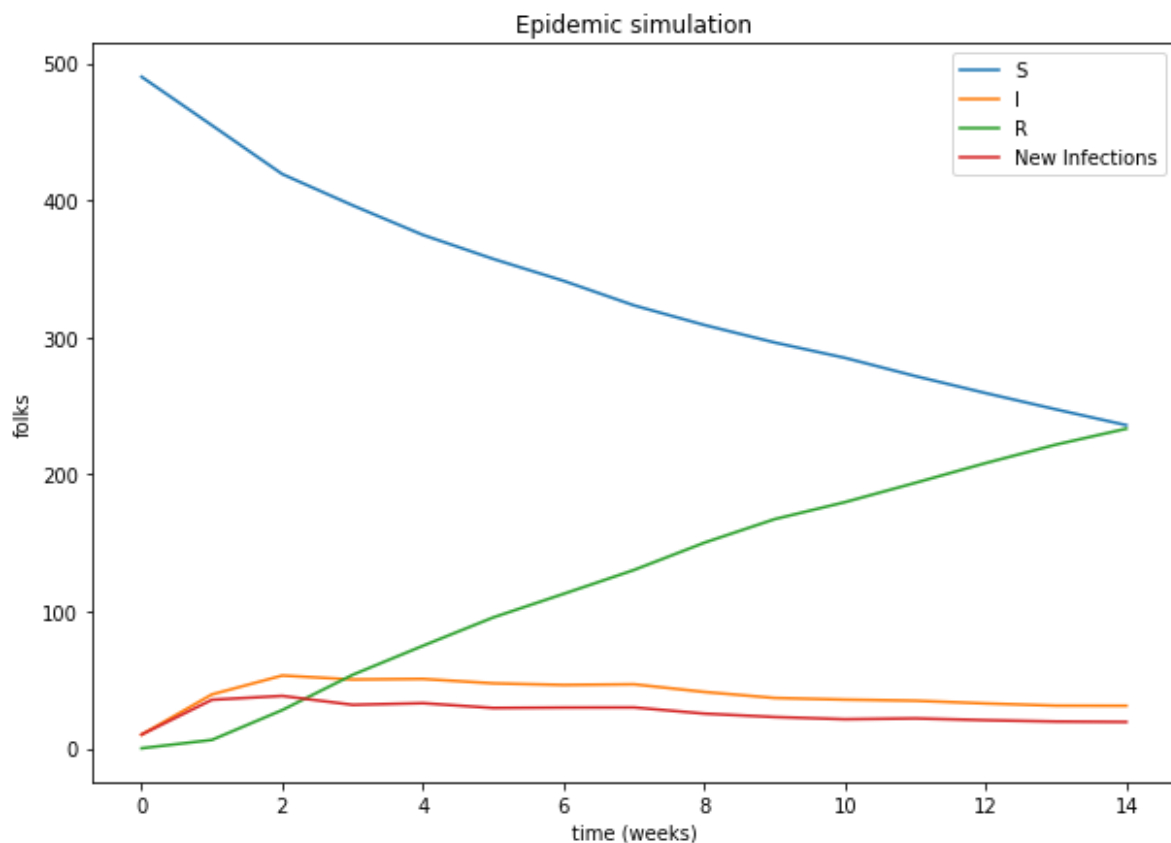
The simulation does not include a vaccination plan, the initial state is shaped that every node is in state 0 (susceptible) except for 10 random nodes over 500, which are infected. The simulation plot represent the average results of 10 simulations.

The infection probability parameter β is set to 0.3, while recovery (ρ) one is set to 0.7. The infection probability of a given node X_i is defined as

$\mathbf{P}(X_i(t+1) = I | X_i(t) = S) = 1 - (1 - \beta)^m$, where $m = \sum_j W_{j,i} * \delta_{X_j(t)}^I$, and k (the number of links of each node) is set to 4, creating a 4-regular graph.



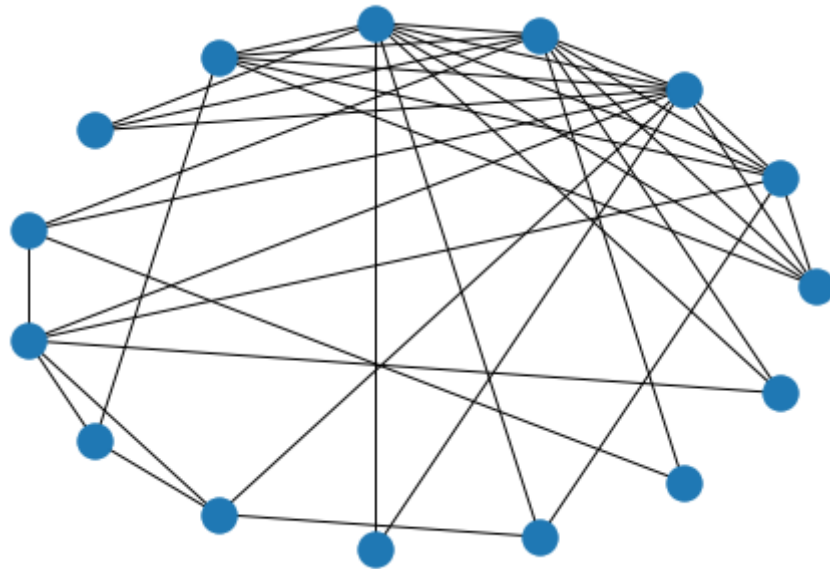
It can be easily noticed that in this case because of the graph structure (which is a 'faster and more strong flu spread' version of a circular graph) the simulation almost reaches the absorbing configuration where susceptible represent the $\approx 90\%$ of the population whereas the other $\approx 10\%$ is represented by recovered (or dead there is no such distinction in this case) people. Another attempt that tries some limit case simulates the pandemic with $\beta = 0.9$ (very high contagion rate).



In the plot of the simulation we can see that the susceptible decrease more rapidly as well as the recovered, but the infected as well as the newly week infections decrease from the second week, which means that we cannot state if the absorbing configuration will comprehend some susceptible in the end, even is very likely to happen (it can vary also because of dead bridges of $k > 1$ consecutive recovered nodes). We can state in the end that given this composition of the graph and $\rho = 0.7$ which is quite high, people recover very fast and is particularly difficult that every node gets infected.

Exercise 2

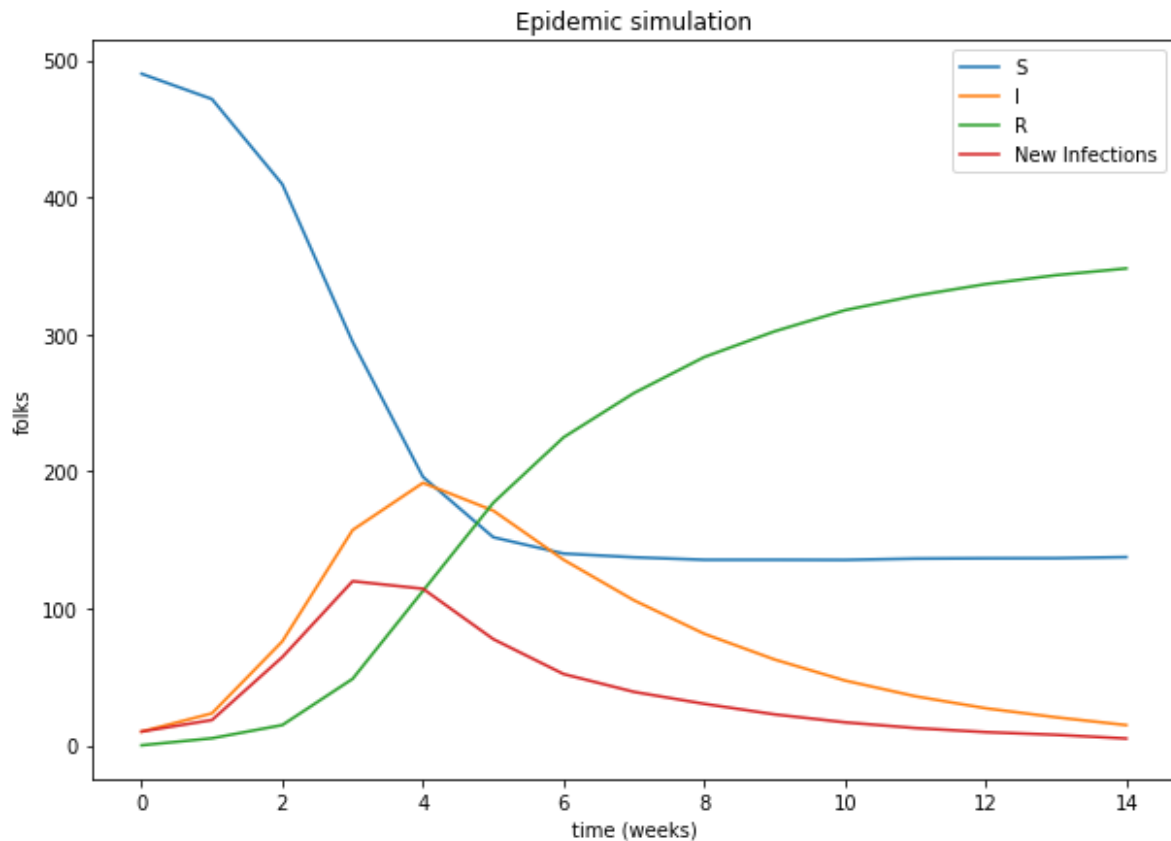
After having defined the class RandomGraphGenerator which creates a random graph with the preferential attachment model, which takes as input parameters k (the number of the initial k -complete graph, which defines the average degree $= \frac{k}{2}$), and n = number of nodes in the final graph. Here there is an example with $k = 5$, $n = 15$.



We can notice that even if the average degree is c.a. 5 there is a node with 10 edges (top right) while at the bottom of the graph there is one with only 2 edges. In fact the graph is much more dense in the top right part, which coincide with the 5-complete starting graph.

Now we simulate the previous pandemic model over the new graph but this time with $k = 6$ average degree, filled with 500 nodes (same number as before), β and ρ are maintained to 0.3 and 0.6 respectively.

The resulting plot is the following:



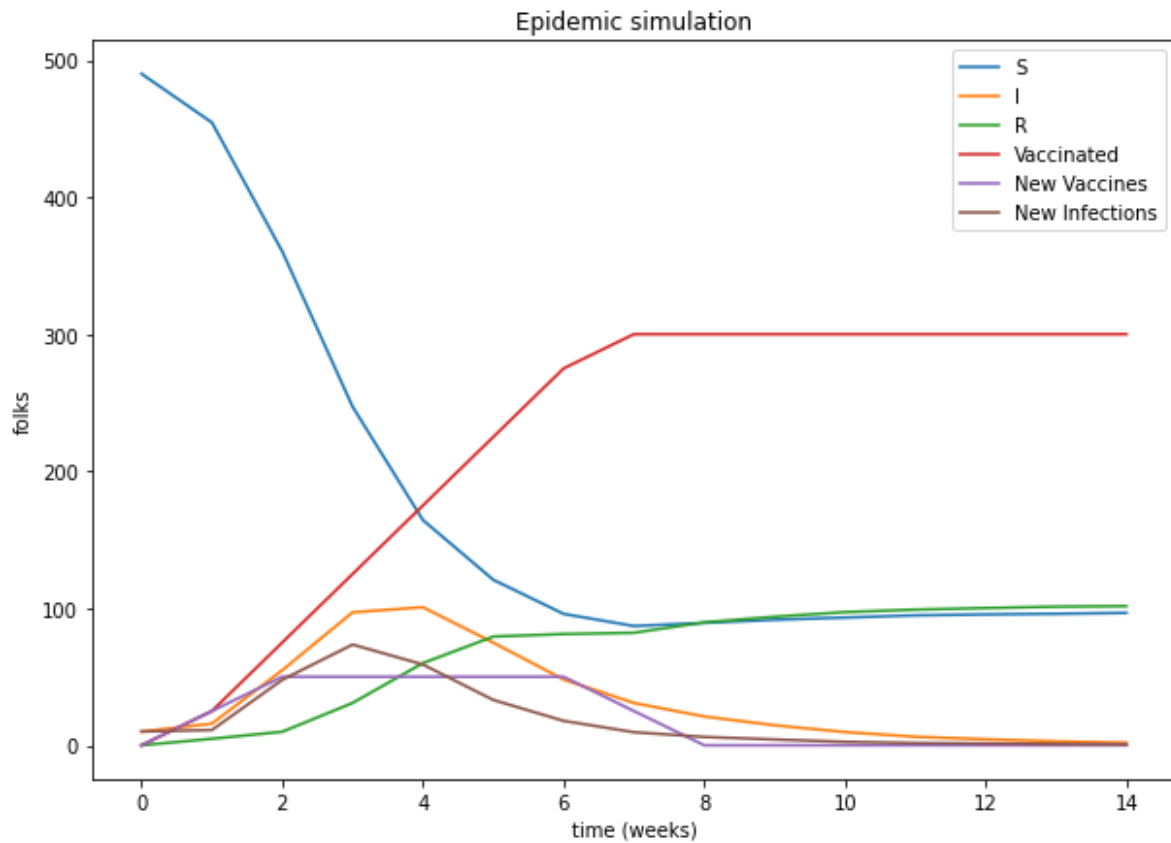
The infection rate grows faster and both the infection cases and the newly infections present a more defined bell shape, while the susceptible decrease drastically approximately to the 30% in 6 weeks, anyway the recovery rate is pretty high and in the end the new infections rate is ≈ 0 , which could lead towards a stable configuration.

Exercise 3

This time the previous experiment is repeated with the same conditions, expect for the vaccination plan that this time is given randomly to the population in a predefined time serie disposal (every state which is not already vaccinated could be picked up for the vaccine administration). The vaccination is supposed to work 100% instantaneously and independently on the state of the subject it won't be neither infected neither contagious. The cumulative vaccine distribution is

vacc = [0,5,15,25,35,45,55,60,60,60,60,60,60,60]

The respective resulting plot is:



We can state that the vaccination campaign made a huge difference, the infection rate decreases rapidly, and the final state of recovered + infected population is ≈ 20 (against 70/80% of the previous experiment).

Exercise 4

The exercise 4 aims to estimate the parameters k , β and ρ comparing the number of newly infected of the real H1N1 pandemic of 2009 in Sweden, decreasing the graph cardinality of 10^4 . The search is based on an iterative grid search over some subset of the total search space possible defined as $\forall \theta \in \text{parameters} \quad \mathbf{S}_{i^{\text{th}}-\text{space}} \subseteq \{\theta - \delta(\theta), \theta, \theta + \delta(\theta)\}$, at the beginning δk is set to 1 (the min possible k) while $\delta(\rho)$ and $\delta(\beta)$ are set to 0.1 (they will decrease until 0.025 in the last iterations). The grid search does not repeat the previous already searched experiments even if a new iteration could lead to different results (the results are very random even if each space configuration is tested 10 times and it is then averaged).

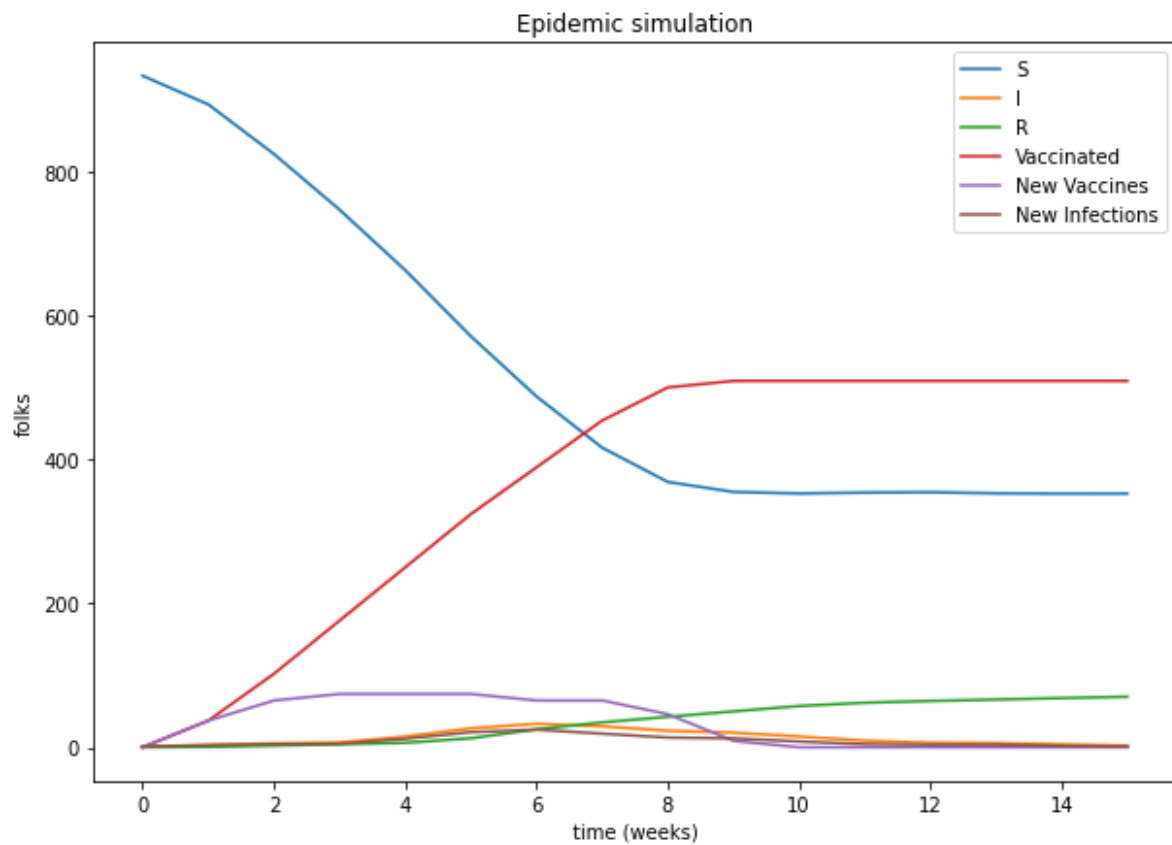
The performance is measured through the root mean square error (RMSE) between the simulation and the ground truth of newly infected.

The search space resulted to be:

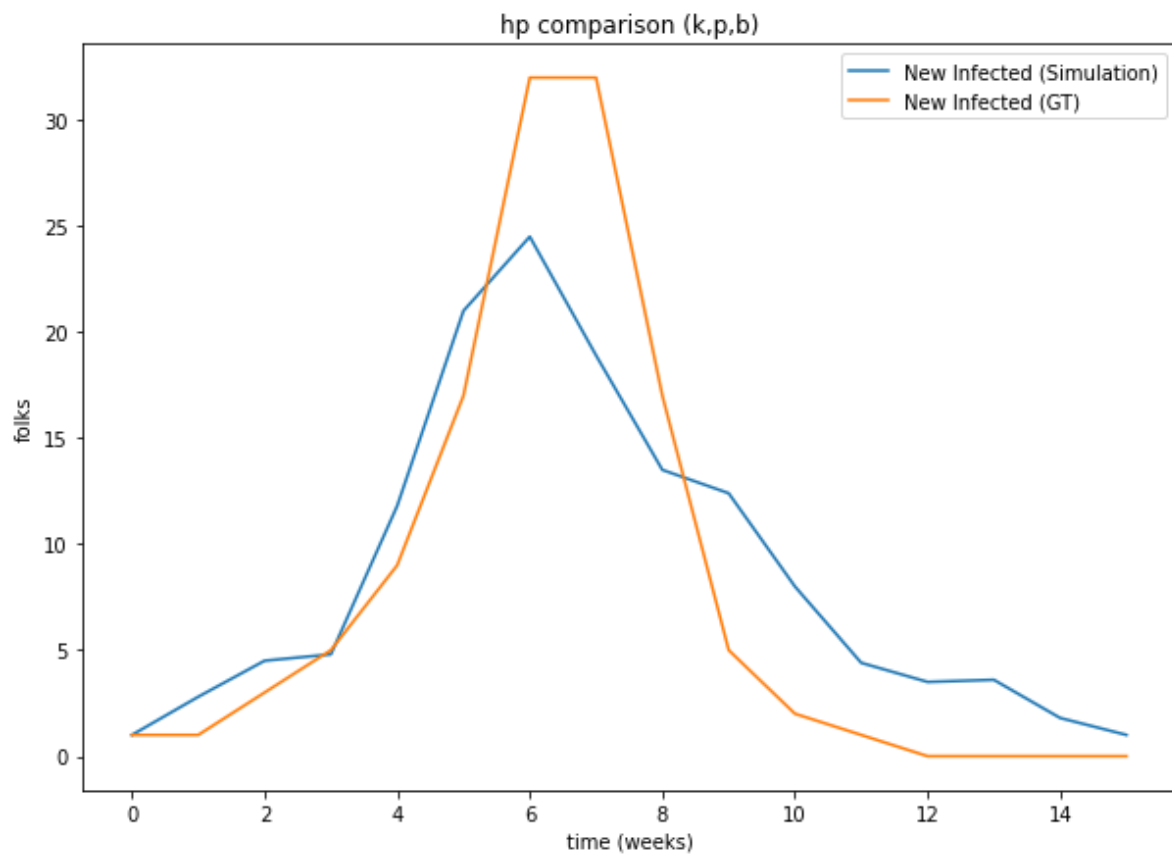
```
{'k': [9, 10, 11], 'b': [0.2, 0.3, 0.4], 'p': [0.5, 0.6, 0.7]}
{'k': [8, 9, 10], 'b': [0.1, 0.2, 0.3], 'p': [0.5, 0.6, 0.7]}
{'k': [9, 10, 11], 'b': [0.0, 0.1, 0.2], 'p': [0.5, 0.6, 0.7]}
{'k': [9, 10, 11], 'b': [0.05, 0.1, 0.15], 'p': [0.55, 0.6, 0.65]}
{'k': [9, 10, 11], 'b': [0.075, 0.1, 0.125], 'p': [0.575, 0.6, 0.625]}
```

while the best result occurred for $\{\beta : 0.1, k : 10, \rho : 0.6\}$ with an RMSE of 5.001.

The best result occurred for a previous run with parameters $\{\beta : 0.2, k : 11, \rho : 0.5\}$ which achieved the best RMSE of 2.86 (remark that this solution has been tested even in the experiment previously reported -step 1). The picture below represent the plot of SIR, new vaccines and new infections



While the next picture represents the plots comparing the ground truth and the best simulation new infections rate



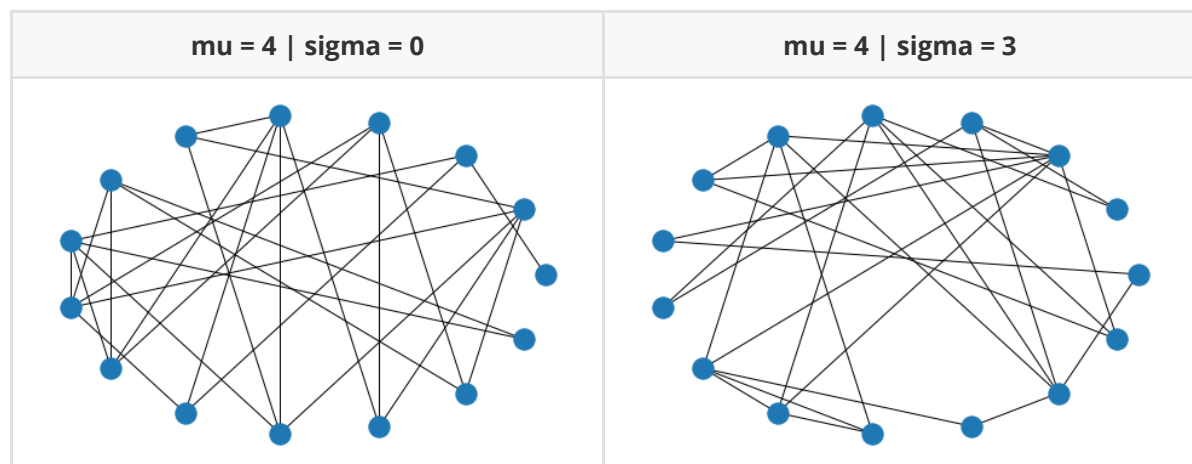
At the end of the day, after various runs we can state that k is in a detour of 10 ± 1 , $\rho = 0.55 \pm 0.1$, $\beta = 0.25 \pm 0.5$, based on this graph composition.

Exercise 5 (Extra)

This exercise has been carried out with Davide Bussone. The referring literature is reported at the end.

The initial idea considered the population density in Sweden, which presents 3 main poles in Stockholm (900k inh.), Goteborg (45k) and Malmo (35k).

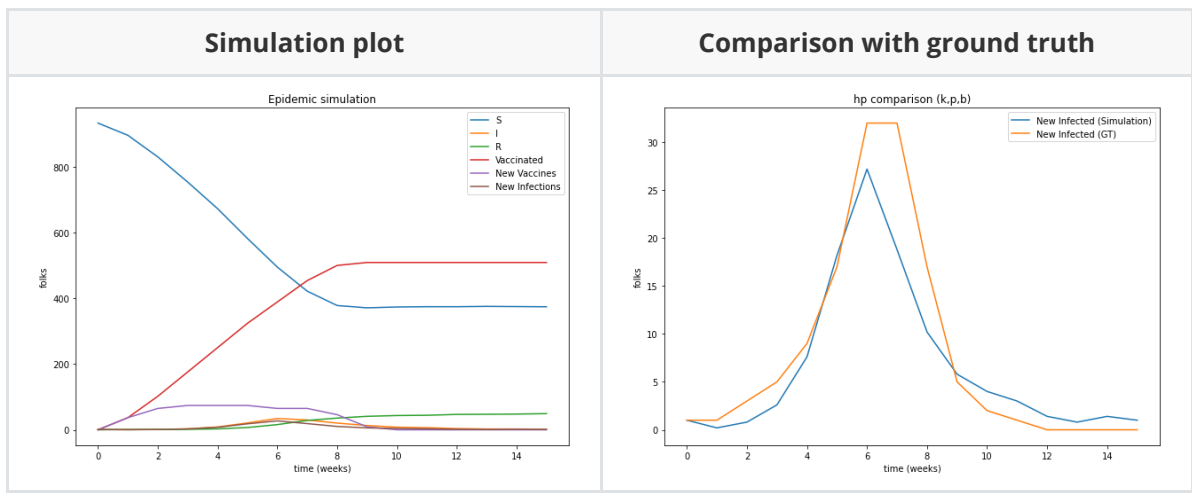
The idea was to recreate the population/relationship graph through a Random Graph generated with the configuration model. The main problem was the configuration, in the literature proposed we have not been able to find relevant data to reconstruct the graph. We decided in the end to generate a random normal distribution based on $\mathbf{N}(\mu, \sigma)$ which denotes the mean and standard deviation of the degree of nodes. Some fixes are apported to balance the integer approximations to preserve both mean and variance. The edge probability is not uniform but is distributed accordingly to the remaining residual attachments, in order to try to recreate hubs-nodes connected together, which could represent the nodes laying in the 3 biggest cities. For $N = 15$ nodes we have two toys examples below with same mean and different std.



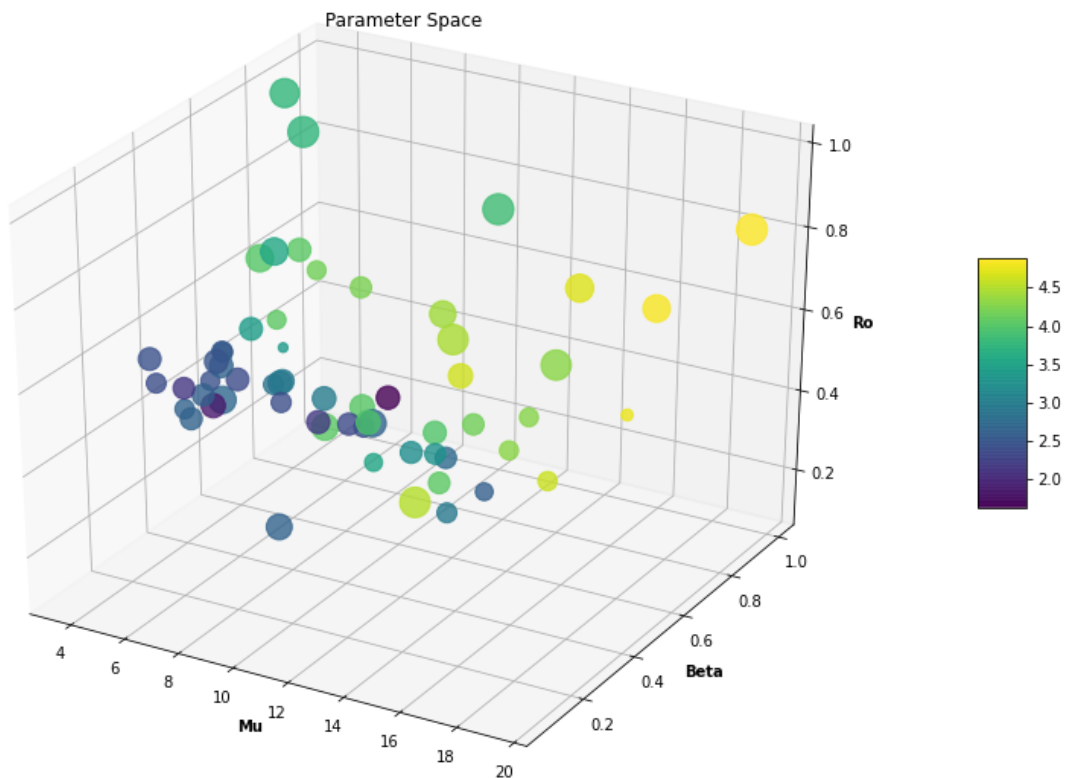
```
{'mu': [2, 20], 'sigma': [0, 15], 'p': [0, 1], 'b': [0, 1]}
{'mu': [4.4467343841782805, 18.252306530818036], 'sigma': [1.334728053506542,
13.636881974686524], 'p': [0.3821947039233371, 0.9137387241768485], 'b':
[0.05288051450365905, 0.7956994761963191]}
{'mu': [4.4467343841782805, 18.252306530818036], 'sigma': [4.531042186152833,
10.736264418073713], 'p': [0.3821947039233371, 0.6680939354334684], 'b':
[0.05288051450365905, 0.4980400296225941]}
{'mu': [5.407918070496258, 18.252306530818036], 'sigma': [4.531042186152833,
8.34263659408625], 'p': [0.5499228090866055, 0.6680939354334684], 'b':
[0.05288051450365905, 0.2971580303401039]}
{'mu': [5.407918070496258, 12.644504475606002], 'sigma': [5.594062370979486,
8.34263659408625], 'p': [0.5499228090866055, 0.6671306227311536], 'b':
[0.13454508278256605, 0.2971580303401039]}
```

best result obtained

```
4.125833249175249 {'mu': 12.570699358634222, 'sigma': 7.855656426839305, 'p':
0.6099537124145691, 'b': 0.2971580303401039}
```

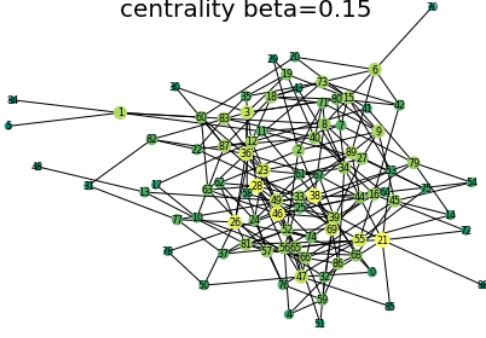
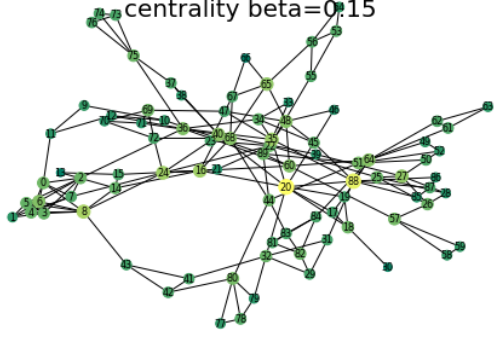


While the solution over the search space are



While mu, beta and rho are presented, it must be said that the color represent the logarithm of the average of RMSEs obtained while the size of each point represents the standard deviation. We can notice how the scatter plot is more dense near the blue (better score area) which means that the random search focused in that sub-cube, but the best result possible is almost into the green area which seems to be isolated from the other best results. At the end of the day we can state that the average of the connection is c.a. 6 for both random graphs, but an higher variance is preferred, moreover the beta and rho parameters are almost stable for both algorithms near 0.3 and 0.6 respectively.

PageRank Centrality $\beta = 0.15$ (Toy example 90 nodes, $\mu = 5$, $\sigma = 2$)

Configuration Model	Small World
<p>centrality beta=0.15</p> 	<p>centrality beta=0.15</p> 

The first proposal concerns a configuration model, its code is showed in the class RGConfModel. In it, the degree of each node is predefined, rather than having a probability distribution from which the given degree is selected. A normal distribution is chosen for the degree sequence, since, in configuration model, this sequence is not obliged to follow a Poisson distribution. The mean and the standard deviation of the nodes are denoted by μ and σ , respectively. Normal distribution seems to be more suitable to describe the Sweden Pandemic case, since the huge number of nodes inside the graph and the fact that each item is scaled by a factor of 10^4 . Such a distribution involves negative numbers, to avoid them, every degree less than one is cast to 1.

$$P_{ij} = \frac{Res(j)}{\sum_{k=1}^n Res(k)} \quad \forall k \in V$$

After the decision of the distribution, two stubs are selected, then connected to create an edge, according to preferential attachment's strategy, in order to let higher degree nodes to be connected quicker. This process is repeated until nodes available are not finished. \ The model was not so realistic towards the Sweden Pandemic situation. So, a more suitable model is represented by a small world graph, whose aim is to combine links due to geographic proximity with few long-distance connection.

The Sweden territory is highly shaped by a low people/ m^2 density, but there is a high variance of density accordingly to the territory, for example, there are a lot of small villages and few big cities. In particular it is possible to identify the presence of three main poles in Sweden, represented by the cities of Stockholm (900k), Goteborg(45k) and Malmoe(35k), so we decided to approximate the decreasing trend accordingly to the logic: Stockholm $\approx 10\%$ of Sweden pops, Goteborg is $\approx 5\%$ and so on following the rule $\frac{N}{10^j} \quad \forall j = 1, 2 \dots K_{top}$.

Of course after some iteration the number of nodes is too small so we decided to stop increasing j as soon as

$$\max_j \sum_{j=1}^{K_{top}} \frac{N}{10^j}$$

$$\text{s.t. } j > 0, \quad \frac{N}{10^j} \geq 5$$

(we considered 5 as the minimum size of a reasonable subgraph), only the last subgraph could be smaller to fill the gap with N. The idea was that an uniform random number higher than 75% is chosen to be used into the subgraph, while the remaining residual degrees are intended to be available for the perturbation of subgraphs. To ensure the connectivity we add one extra link between each subgraph without decreasing the residuals, moreover some techniques of balancing (because of integer casting) as well as self-loop and double connection avoidance are implemented.

The normal distribution is kept also for the small world random graph. Below it is reported a toy example of this type of random graph with a low number of nodes. Then, some parameters must be optimized:

- μ : mean of the normal distribution
- σ : standard deviation of the normal distribution
- β : $(1-\beta)^m$ measures of the probability that a susceptible nodes surrounded by m-infected nodes do not take the virus.
- ρ : probability that an infected person will get rid of the virus during one time step.

The exploited method is an iterative random search, so a number of iterations is fixed and the hyperparameters are chosen among uniformly sampled reasonable values. This kind of method might lead to a very promising solutions as well as a very bad ones, with respect to the previous iterative algorithm. It starts from the whole space and suffers less of local minima stuck, performs a faster space search which is not limited by more fixed length intervals which in this case is particularly needed because we have 4 instead of 3 parameters to be optimized (curse of dimensionality vs Bussone&Malan 0-1). Additionally to decrease the search space without being limited to restricted local areas we choose over all the 5 best solutions the largest space extended by a factor of $\frac{6}{5}$ validating the result to be compliant with the lower bound k_- and the upper bound k_+ . To be exhaustive the search space at each iteration is considering the 5 best solutions and a parameter θ :

$$[\max(\min(\theta) - (\frac{\max(\theta) - \min(\theta)}{10}), k_-), \min(\max(\theta) + (\frac{\max(\theta) - \min(\theta)}{10}), k_+)]$$

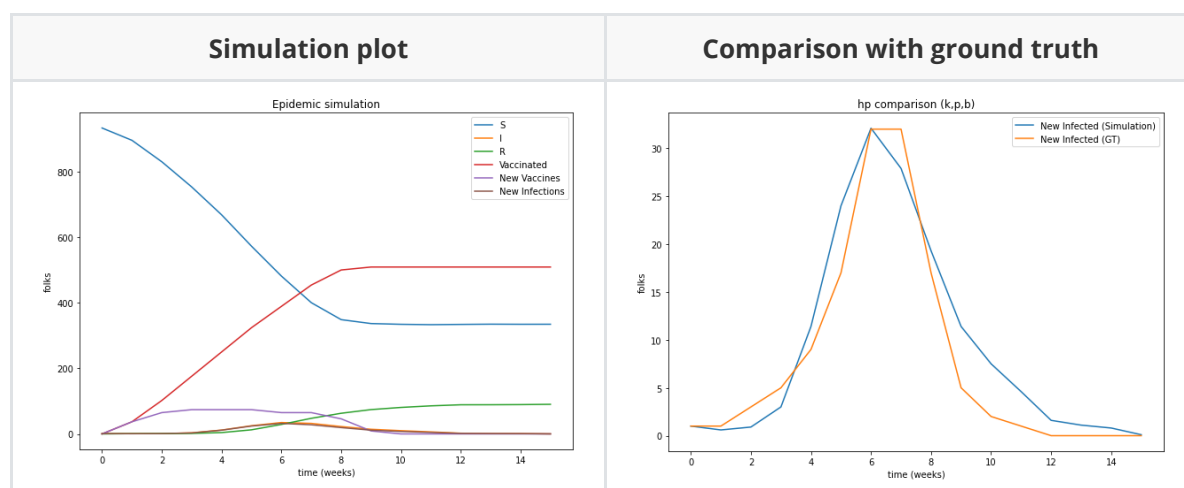
In general these are the key differences with respect to the algorithm in section 4.

After the experiments, three different figures are portrayed:

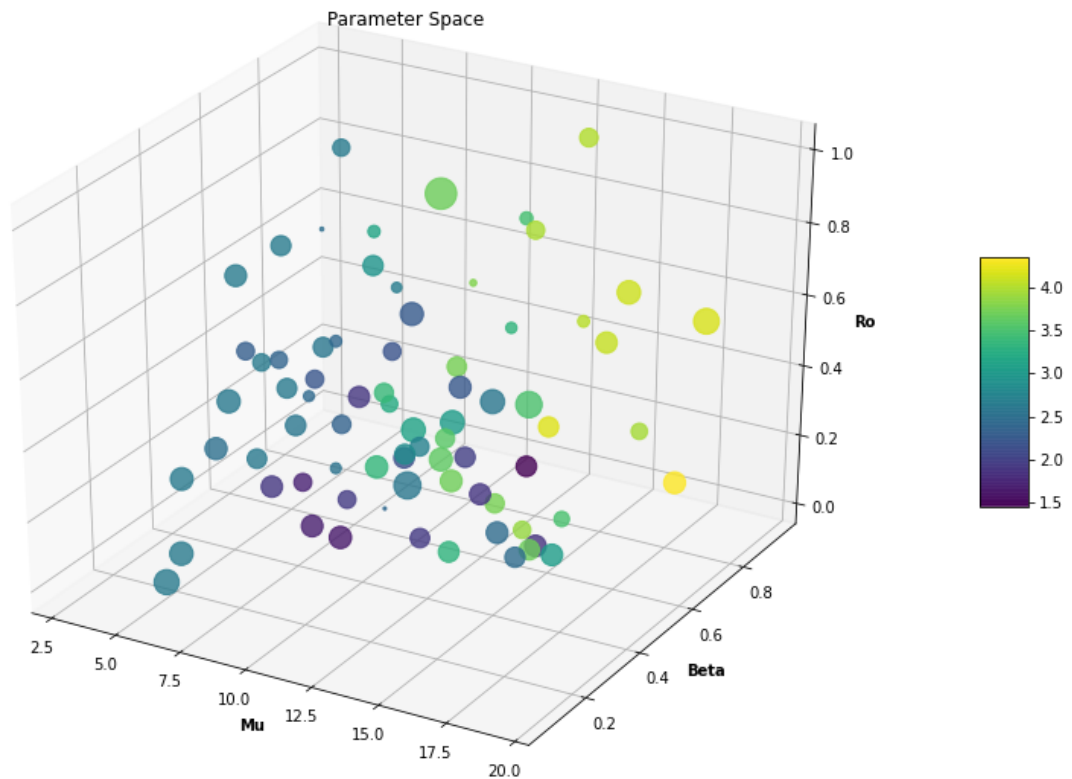
- The first one represents the evolution of the SIR model with vaccination;
- The second one illustrates the comparison between the predicted and true new infected;
- The last one builds the parameter space in which the optimization of μ , β and ρ is showed. The size of the dots represents the variance, while the colour (in Viridis' scale) is indicator of the RMSE error. Of course, the best set of hyperparameters is recognized by the violet dot, which is the one with the lowest error.

As a final comment, in terms of RMSE, there is a slight improvement.

```
RMSE = 3.2980107640818885
{'mu': 15.75720498241178, 'sigma': 6.0586868741065665, 'p': 0.35481668248356474, 'b': 0.4426809802135522}
```



While the solution over the search space are



References:

https://www.researchgate.net/figure/Characteristics-of-the-study-area-Sweden-Left-Population-density-of-counties_fig1_228780371

https://www.pnas.org/content/99/suppl_1/2566

https://www.wikiwand.com/en/2009_swine_flu_pandemic_in_Europe

<https://www.folkhalsomyndigheten.se/contentassets/1d7096c2b65d45b499c924d76333272c/influenza-in-sweden-2009-2010.pdf>

https://en.wikipedia.org/wiki/Demographics_of_Sweden

Code

```
import networkx as nx
import numpy as np
from numpy.random import choice, rand
import numpy as np
import matplotlib.pyplot as plt
from scipy.sparse import lil_matrix, csr_matrix, find
from IPython.core.display import display, HTML
np.random.seed(42)
```

```
def infection(m, beta):
    infection_probability = 1 - (1 - beta)**m
    return 1 if np.random.rand(1)[0] <= infection_probability else 0

def recovery(p):
    return 1 if np.random.rand(1)[0] >= p else 0
```

```

def init_stats(duration, state):
    st = np.zeros([duration, state.shape[0]])
    stats = lil_matrix(st, shape=(duration, state.shape[0]), dtype=int) #faster
    sparse matrix creation/population
    stats[0, :] = state
    return stats

def group_sum(stats, row): #given the stathistics matrix it group elements with
the same value and count value occurency
    previous_state = np.array(find(stats.getrow(row)))[1:]
    zeros = stats.shape[1]-previous_state.shape[1]
    count = {0:zeros}
    for idx in range(previous_state.shape[1]):
        node = previous_state[0, idx]
        value = previous_state[1, idx]
        count[value] = count.get(value, 0) + 1
    return count

def count_new_state_increment(stats, moment, key=1): #performs the difference
between two time periods (considering only the new cases)
    split = stats[moment-1:moment+1, :]
    cii = 0
    for n in range(split.shape[1]):
        if split[1, n] == key and split[0, n] != key: #check if the previous was
not already in that state
            cii += 1
    return cii

def epidemic_simulation(G, model, state, duration, beta, p, vaccination=None):
    stats = init_stats(duration, state)
    if vaccination:
        vaccinated = np.array([])
        for moment in range(1, duration): #for the duration of the study case
            if vaccination:
                #choose a percentual of vaccines (accordingly to the array given)
                vaccines = int((vaccination[moment]-vaccination[moment-1])/100
*state.shape[0])
                #choose randomly subjects of vaccination
                to_be_vaccinated = choice([x for x in range(state.shape[0]) if x not
in vaccinated], vaccines, replace=False)
                for vac in to_be_vaccinated:
                    stats[moment:, vac] = 3 #sets the vaccinated state
                vaccinated = np.concatenate([vaccinated, to_be_vaccinated])
            for node in G.nodes:
                if stats[moment, node] == 3: #vaccinated (skip)
                    continue
                node_state = stats[moment-1, node]
                if node_state == 0: #susceptible
                    neighbors = [n for n in G.neighbors(node)]
                    infected_neighbors = [stats[moment-1, x] for x in
neighbors].count(1) #counts the infected neighbors
                    stats[moment, node] = infection(infected_neighbors, beta) #check if
the subject gets infected
                elif node_state == 1: #infected
                    recovered = recovery(p)

```

```

        stats[moment,node] = node_state + recovery(p)
        if recovered > 0: #recovered
            stats[moment:,node] = 2
    else:
        continue

    stats = csr_matrix(stats) #faster sparse matrix computation
    return stats

def multiple_simulation(G, model, state, duration, beta, p, num, vaccination=None,
plot=True):
    stats = epidemic_simulation(G, model, state, duration, beta, p, vaccination)
    #initial simulation
    for i in range(1,num):
        stats =
np.column_stack((stats,epidemic_simulation(G,model,state,duration,beta,p,vaccinat
ion))) # num - 1 simulations
    ev_rate = compute_evolution(stats,duration,vaccination) #get the stats matrix
and compute the evolution of cases
    if plot:
        duration = range(duration)
        return plot_avg_state(ev_rate,duration,vaccination)
    else:
        return ev_rate

def compute_evolution(stats,duration,vaccination=None):
    initial_step = group_sum(stats[0,0],0) #start (t_0)
    initial_state = np.zeros(4 + (2 if vaccination else 0)) #initial state in t_0
    for k,v in initial_step.items():
        initial_state[k] = v
    initial_state[-1] = initial_step[1]
    # initial state = [susceptible,infected,recovered,newly_infected] or if
vaccination given
    # initial state =
[susceptible,infected,recovered,newly_vaccinated,newly_infected]
    evolution_rate = initial_state
    for moment in range(1,duration):
        if vaccination:
            avg_state = np.zeros(6)
        else:
            avg_state = np.zeros(4)
        for simulation in range(0,stats.shape[1]):
            count = group_sum(stats[0,simulation],moment)
            for k,v in count.items():
                avg_state[k] += v
            avg_state[-1] +=
count_new_state_increment(stats[0,simulation],moment)
            if vaccination:
                avg_state[-2] +=
count_new_state_increment(stats[0,simulation],moment,key=3)
            avg_state = avg_state / stats.shape[1]
            evolution_rate = np.column_stack((evolution_rate, avg_state))
    return evolution_rate

def plot_avg_state(evolution_rate,duration,vaccination=None):
    #evolution_rate = compute_evolution(stats,duration,vaccination)
    fig, ax= plt.subplots(figsize=(10,7))

```

```

ax.plot(duration, evolution_rate[0,:], label='S')
ax.plot(duration, evolution_rate[1,:], label='I')
ax.plot(duration, evolution_rate[2,:], label='R')
if vaccination:
    ax.plot(duration, evolution_rate[3,:], label='Vaccinated')
    ax.plot(duration, evolution_rate[-2,:], label='New Vaccines')
ax.plot(duration, evolution_rate[-1,:], label='New Infections')
ax.set(xlabel='time (weeks)',ylabel='folks', title='Epidemic simulation')
ax.legend(loc='best');
return evolution_rate
#print(evolution_rate)

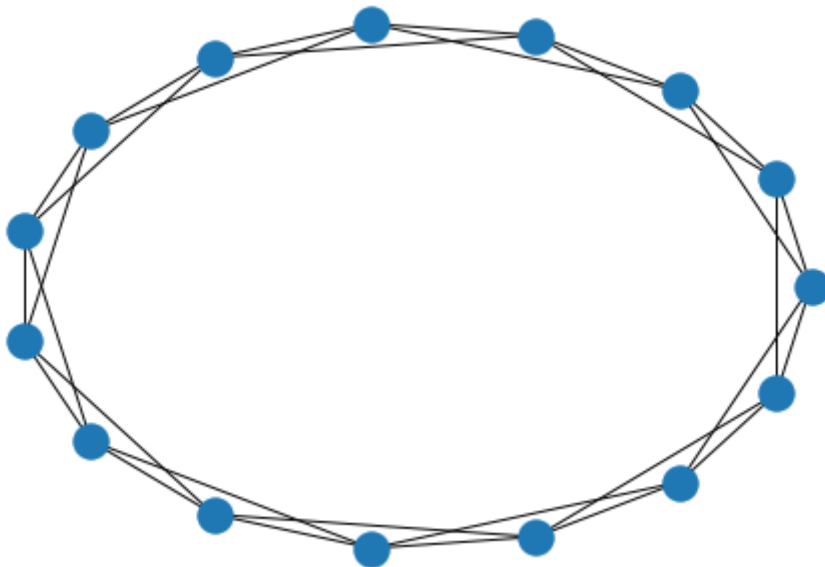
```

```

N = 15
nodes = range(N)
edges = []
for index,node in enumerate(nodes):
    edges.append([node,nodes[index-1]])
    edges.append([node,nodes[index-2]])

G = nx.Graph()
G.add_nodes_from(nodes)
G.add_edges_from(edges)
nx.draw_circular(G)

```



```

# Init
N = 500
nodes = range(N)
edges = []
for index,node in enumerate(nodes):
    edges.append([node,nodes[index-1]])
    edges.append([node,nodes[index-2]])

G = nx.Graph()
G.add_nodes_from(nodes)

```



```

G.add_edges_from(edges)
#nx.draw_circular(G)
status = np.zeros(N)
beta = 0.3
p = 0.7
simulation_time = 15 #weeks
attempts = 100
SIR = {'S':0, 'I':1, 'R':2}

init_infected = choice(nodes,10)
for inf in init_infected:
    status[inf] = 1
print(init_infected)

```

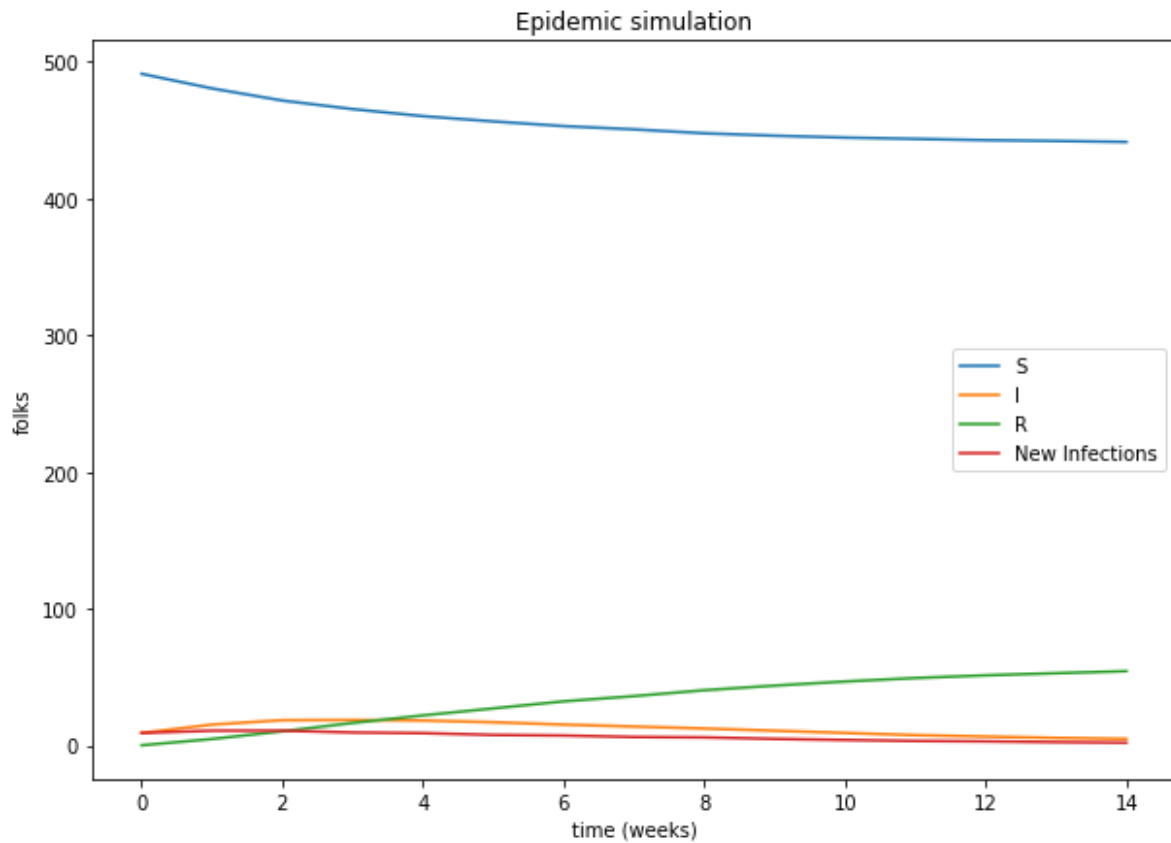
```
[102 435 348 270 106  71 188  20 102 121]
```

```
multiple_simulation(G,SIR,status,simulation_time,beta,p,attempts)
```

```

array([[491.  , 480.33, 471.46, 465.21, 460.04, 456.2 , 452.76, 450.37,
        447.5 , 445.74, 444.37, 443.42, 442.47, 441.91, 441.18],
       [ 9.  , 15.09, 18.35, 18.48, 18.1 , 16.84, 15.12, 13.7 ,
        12.24, 10.63,  9.  ,  7.41,  6.38,  5.38,  4.64],
       [ 0.  ,  4.58, 10.19, 16.31, 21.86, 26.96, 32.12, 35.93,
        40.26, 43.63, 46.63, 49.17, 51.15, 52.71, 54.18],
       [ 9.  , 10.67, 10.7 ,  9.35,  8.87,  7.56,  7.13,  6.06,
        5.77,  4.67,  3.88,  3.17,  2.87,  2.28,  1.9 ]])

```



```
# Init
N = 500
nodes = range(N)
edges = []
for index, node in enumerate(nodes):
    edges.append([node, nodes[index-1]])
    edges.append([node, nodes[index-2]])

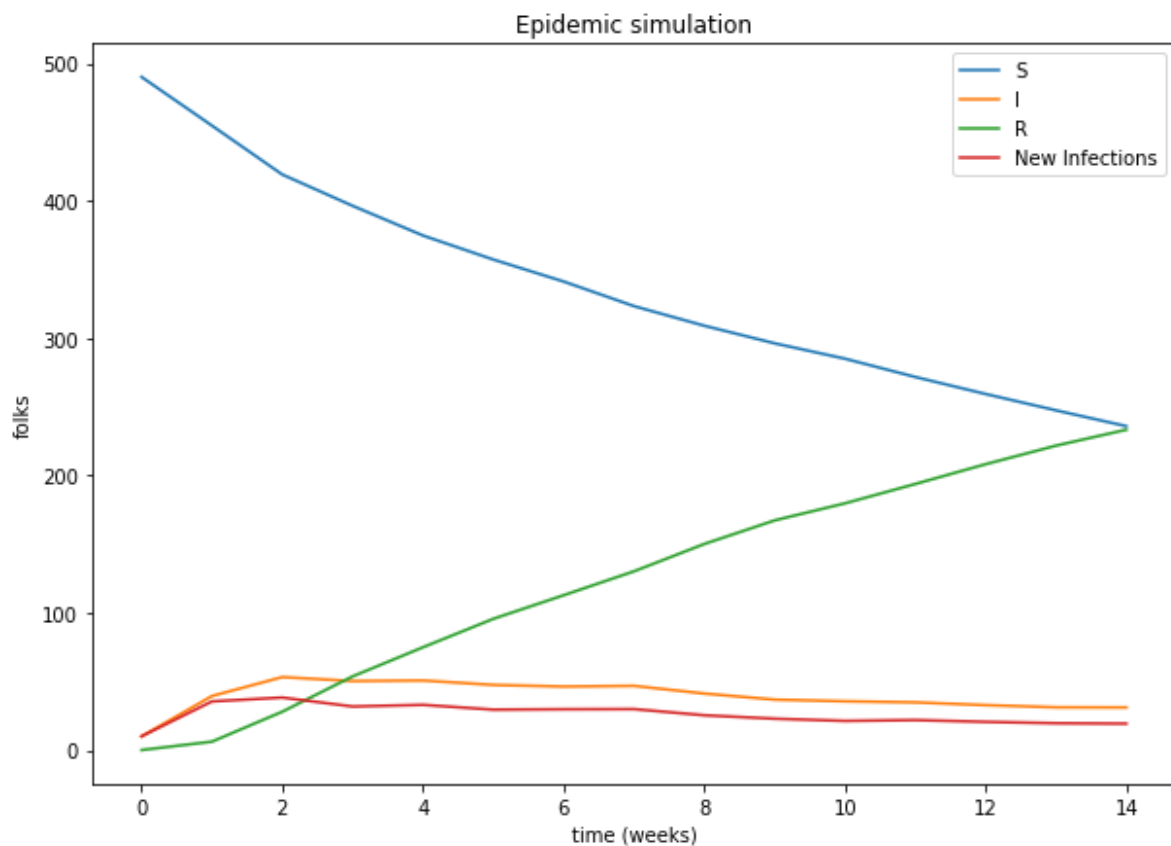
G = nx.Graph()
G.add_nodes_from(nodes)
G.add_edges_from(edges)
#nx.draw_circular(G)
status = np.zeros(N)
beta = 0.9
p = 0.6
simulation_time = 15 #weeks
attempts = 10
SIR = {'S':0, 'I':1, 'R':2}

init_infected = choice(nodes,10)
for inf in init_infected:
    status[inf] = 1
print(init_infected)
```

```
[423  19 433 246 284 130 138 255 449 167]
```

```
multiple_simulation(G,SIR,status,simulation_time,beta,p,attempts)
```

```
array([[490. , 454.6, 419. , 396.2, 374.6, 357.1, 341.1, 323.2, 308.9,
       296.1, 284.9, 271.6, 259.2, 247.3, 235.8],
      [ 10. ,  39.3,  53.1,  50.2,  50.6,  47.4,  46.2,  46.7,  41.1,
        36.7,  35.5,  34.7,  32.7,  31.1,  31. ],
      [  0. ,   6.1,  27.9,  53.6,  74.8,  95.5, 112.7, 130.1, 150. ,
       167.2, 179.6, 193.7, 208.1, 221.6, 233.2],
      [ 10. ,  35.4,  38.2,  31.7,  32.9,  29.4,  29.7,  29.8,  25.3,
       22.8,  21.2,  21.8,  20.5,  19.5,  19.2]])
```



```
class RandomGraphGenerator():
    def __init__(self,n,k):
        self.G = nx.complete_graph(k+1)
        self.avg_degree = np.floor(k/2)
        self.need_balance = False if k % 2 == 0 else True
        for node in range(len(self.G),n):
            self.preferential_attachment()

    def addNode(self):
        node = len(self.G.nodes)
        self.G.add_node(node)
        grade=self.avg_degree
        if self.need_balance:
```

```

        grade += len(self.G) % 2
    return node, grade

    def preferential_attachment(self):
        node, grade = self.addNode()
        degrees = np.array([self.G.degree(x) for x in self.G.nodes])
        attachment_probabilities = degrees / np.sum(degrees)
        neighbors = np.random.choice(self.G.nodes, size=int(grade),
        replace=False, p = attachment_probabilities)
        for n in neighbors:
            self.G.add_edge(node, n)
        return

    def print(self, *param):
        print(param)

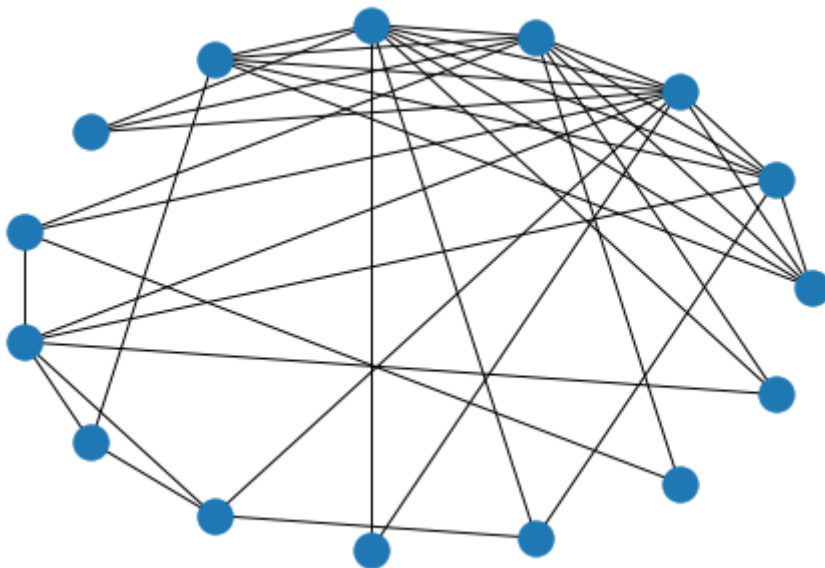
```

```

rg = RandomGraphGenerator(15,5)
print(np.mean(np.array([rg.G.degree(x) for x in rg.G.nodes])))
nx.draw_circular(rg.G)

```

5.066666666666666



```

N = 500
k = 6
rg = RandomGraphGenerator(N, k)
status = np.zeros(N)
beta = 0.3
p = 0.7
simulation_time = 15 #weeks
attempts = 100
SIR = {'S':0, 'I':1, 'R':2}

init_infected = choice(rg.G.nodes, 10, replace=False)

```

```

for inf in init_infected:
    status[inf] = 1
print(init_infected)
multiple_simulation(rg.G,SIR,status,simulation_time,beta,p,attempts)

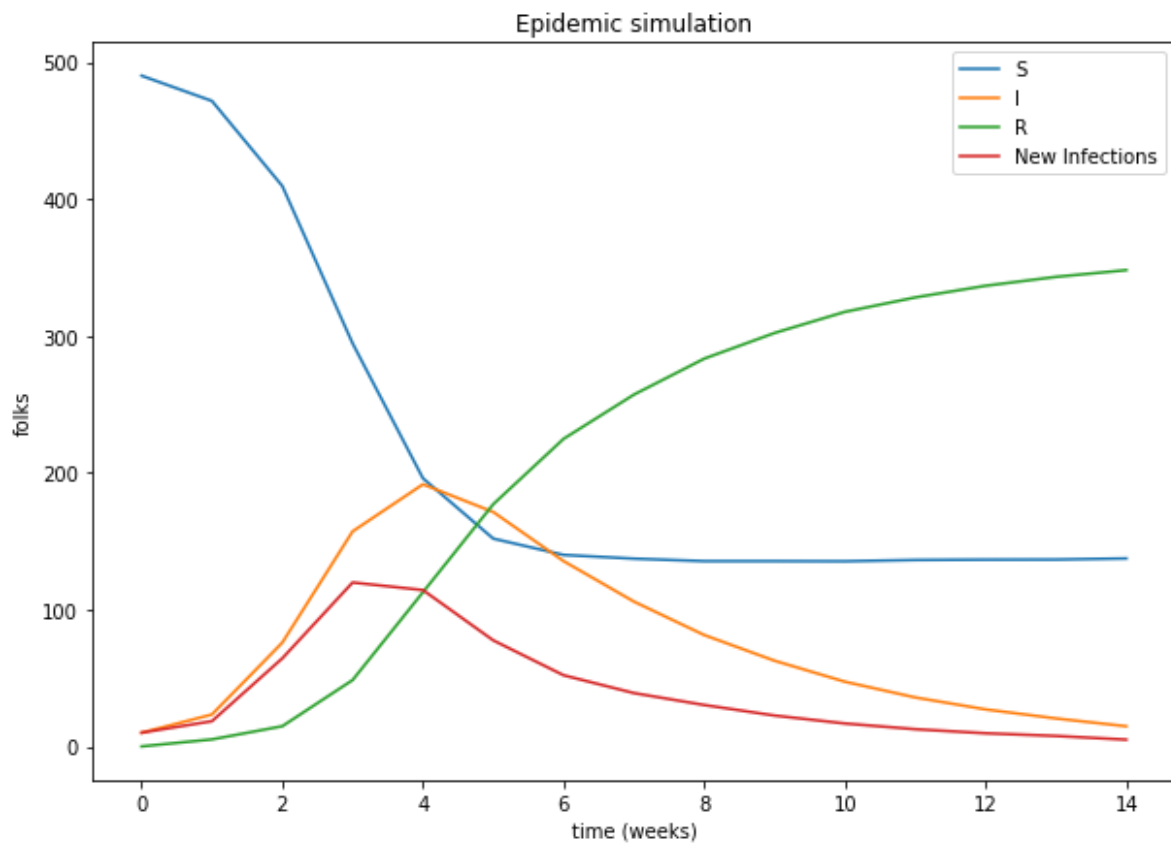
```

```
[ 18  21 295 132 221 216 320 211 467 140]
```

```

array([[490. , 471.58, 409.47, 294.44, 195.86, 151.8 , 139.9 , 137.18,
       135.35, 135.35, 135.26, 136.22, 136.45, 136.53, 137.32],
      [ 10. ,  23.3 ,  75.79, 157.08, 191.49, 171.24, 135.37, 105.79,
       81.34,  62.58,  47.24,  35.75,  27.01,  20.42,  14.71],
      [  0. ,   5.12,  14.74,  48.48, 112.65, 176.96, 224.73, 257.03,
      283.31, 302.07, 317.5 , 328.03, 336.54, 343.05, 347.97],
      [ 10. ,  18.42,  64.3 , 119.76, 114.23,  77.47,  51.97,  38.96,
       30.23,  22.54,  16.74,  12.56,   9.59,   7.63,   4.93]])

```



```

vacc = [0,5,15,25,35,45,55,60,60,60,60,60,60,60,60]
N = 500
k = 6
rg = RandomGraphGenerator(N,k)
status = np.zeros(N)
beta = 0.3

```

```

p = 0.7
simulation_time = 15 #weeks
attempts = 100
SIR = {'S':0, 'I':1, 'R':2}

init_infected = choice(rg.G.nodes,10, replace=False)
for inf in init_infected:
    status[inf] = 1
print(init_infected)
multiple_simulation(rg.G,SIR,status,simulation_time,beta,p,attempts,vacc)

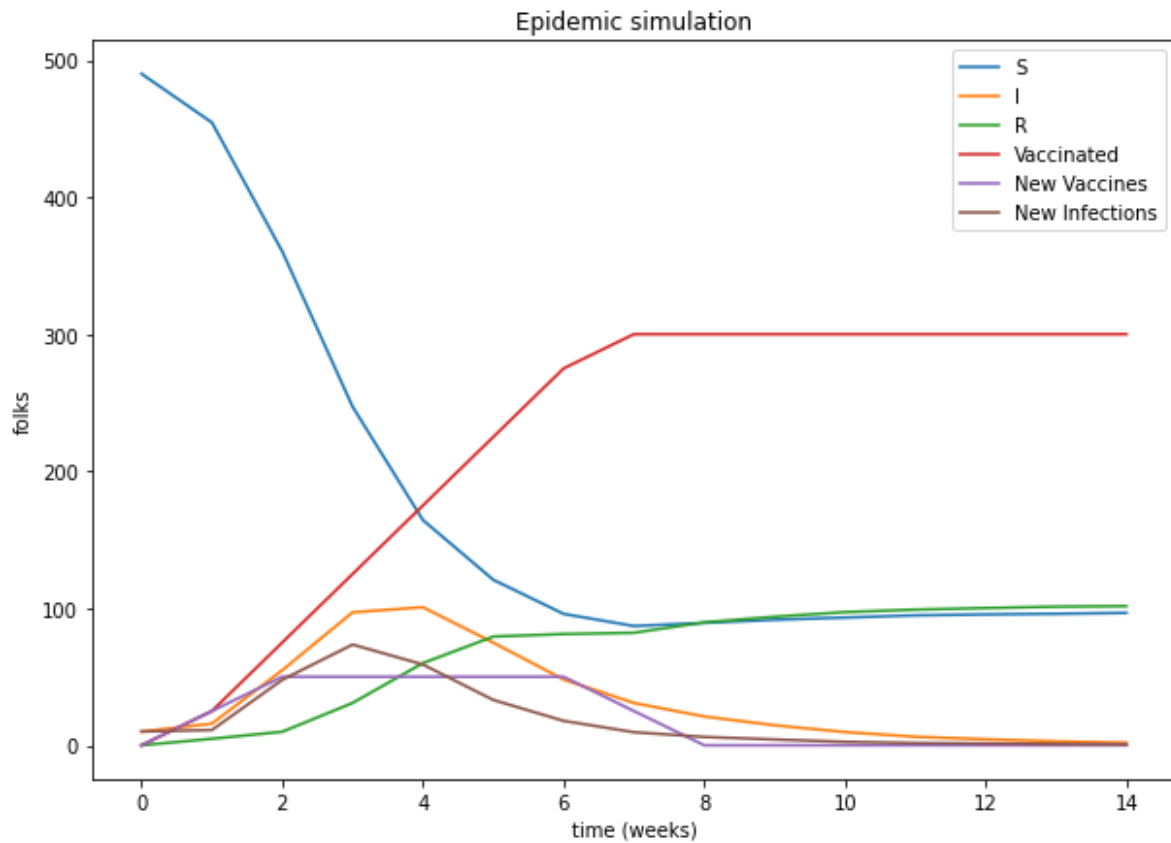
```

```
[378 497 401 365 345 69 214 235 191 84]
```

```

array([[490. , 454.46, 360.39, 247.07, 164.32, 120.77, 95.87, 87.03,
      89.29, 91.59, 93.2 , 94.81, 95.49, 95.97, 96.62],
      [ 10. , 15.78, 54.77, 97.03, 100.73, 74.9 , 47.96, 30.88,
      21. , 14.59, 9.61, 6.2 , 4.36, 2.87, 1.85],
      [ 0. , 4.76, 9.84, 30.9 , 59.95, 79.33, 81.17, 82.09,
      89.71, 93.82, 97.19, 98.99, 100.15, 101.16, 101.53],
      [ 0. , 25. , 75. , 125. , 175. , 225. , 275. , 300. ,
      300. , 300. , 300. , 300. , 300. , 300. , 300. ],
      [ 0. , 25. , 50. , 50. , 50. , 50. , 50. , 25. ,
      0. , 0. , 0. , 0. , 0. , 0. , 0. ],
      [ 10. , 11.13, 47.77, 73.51, 58.92, 33.08, 17.82, 9.52,
      6.08, 4.24, 2.51, 1.64, 1.33, 0.96, 0.5 ]])

```



```

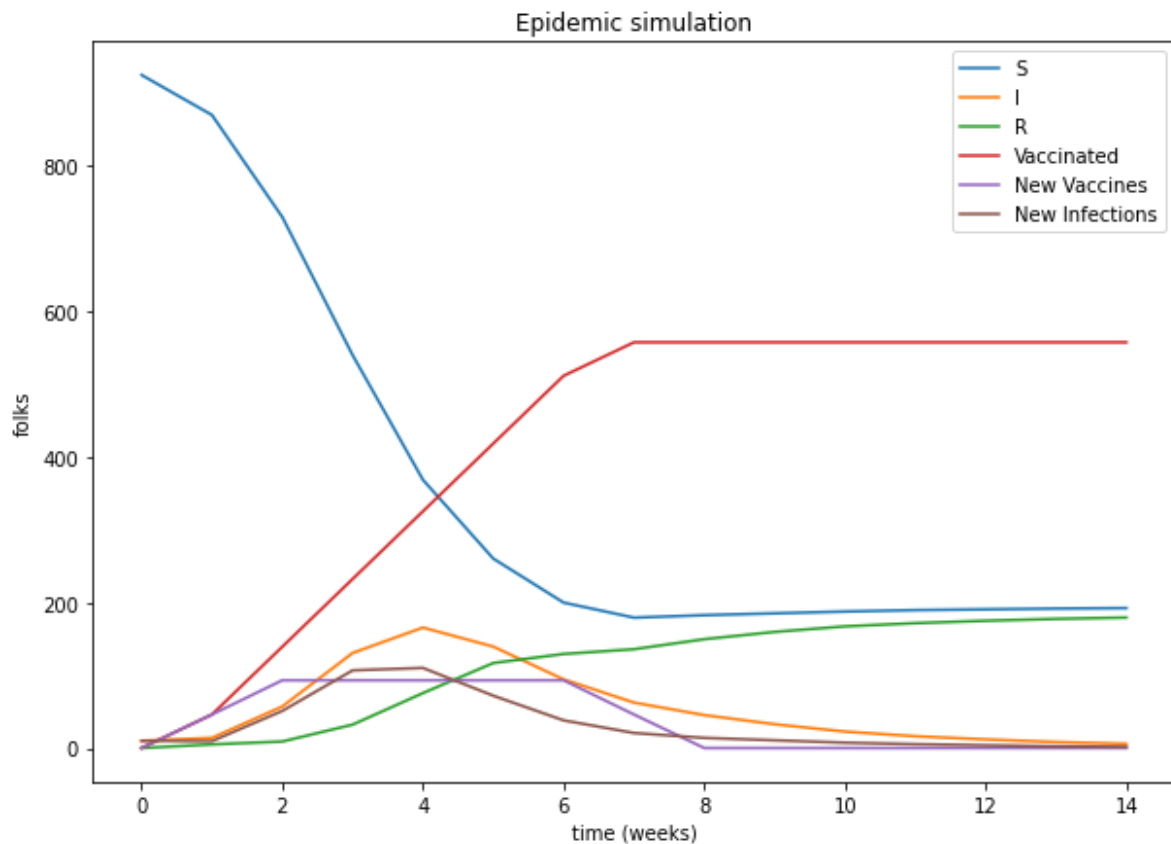
vacc = [0, 5, 15, 25, 35, 45, 55, 60, 60, 60, 60, 60, 60, 60, 60]
N = 934
k = 6
rg = RandomGraphGenerator(N, k)
status = np.zeros(N)
beta = 0.3
p = 0.7
simulation_time = 15 #weeks
attempts = 100
SIR = {'S':0, 'I':1, 'R':2}

init_infected = choice(rg.G.nodes, 10, replace=False)
for inf in init_infected:
    status[inf] = 1
print(init_infected)
multiple_simulation(rg.G, SIR, status, simulation_time, beta, p, attempts, vacc)

```

```
[679 408 922 813 890 521 110 804 720 266]
```

```
array([[924. , 869.06, 729.09, 539.43, 368.2 , 260.07, 199.64, 178.98,
       182.38, 184.92, 187.39, 189.34, 190.38, 191.46, 192.11],
      [ 10. ,  13.95,  56.96, 130.47, 165.38, 139.22,  94.24,  62.37,
       45.19,  32.65,  22.57,  16.22,  11.88,   8.25,   5.78],
      [  0. ,   4.99,   8.95,  32.1 ,  75.42, 116.71, 129.12, 135.65,
      149.43, 159.43, 167.04, 171.44, 174.74, 177.29, 179.11],
      [  0. ,  46. , 139. , 232. , 325. , 418. , 511. , 557. ,
      557. , 557. , 557. , 557. , 557. , 557. , 557. ],
      [  0. ,  46. ,  93. ,  93. ,  93. ,  93. ,  93. ,  46. ,
       0. ,   0. ,   0. ,   0. ,   0. ,   0. ,   0. ],
      [ 10. ,   9.42,  50.95, 106.63, 109.96,  71.83,  37.98,  20.64,
      13.91,  10.62,   7.34,   5.27,   3.81,   2.38,   1.59]])
```



```
from sklearn.model_selection import ParameterGrid
N = 934
vacc = [5,9,16,24,32,40,47,54,59,60,60,60,60,60,60,60]
status = np.zeros(N,dtype=np.int32)
init_infected = choice(range(status.shape[0]),1)
status[init_infected] = 1
simulation_time = 16
attempts = 10
SIR = {'S':0, 'I':1, 'R':2, 'V':3}
NI_true = [1,1,3,5,9,17,32,32,17,5,2,1,0,0,0,0]
k0, b0, p0 = 10, 0.3, 0.6
dk, db, dp = 1, 0.1, 0.1
paramatrix = np.array([[k0,b0,p0],[dk,db,dp]])
```



```

def RMSE(i,i_true):
    return np.sqrt(1/i.shape[0]*sum((i-i_true)**2))

def
search_pandemic(tested_hp,N,model,params,i_true,state,duration,num,vaccination=None,best_res=None):
    grid = ParameterGrid(params) # creates a grid to perform gridsearch over the
subset of parameters k,b,p
    best_res = best_res
    for params in grid:
        # check if the params have already been tested, if so skip the
computation to spare time and computation
        if(params in tested_hp):
            continue
        else:
            tested_hp.append(params)
            rg = RandomGraphGenerator(N,params['k']) #generate the graph with the new
k
            ev_rate =
multiple_simulation(rg.G,model,state,simulation_time,params['b'],params['p'],attempts,vacc,plot=False) #performs the simulation
            newly_inf = ev_rate[-1,:]
            result = RMSE(newly_inf,i_true) #compute RMSE
            if not best_res or result < best_res[0]: #check if it is the best result
and in case it stores it
                best_res = [result,ev_rate,params]
            new = True
    return best_res, tested_hp

def plot_newly_infected(duration,i,i_true):
    ig, ax= plt.subplots(figsize=(10,7))
    ax.plot(duration, i, label='New Infected (Simulation)')
    ax.plot(duration, i_true, label='New Infected (GT)')
    ax.set(xlabel='time (weeks)',ylabel='folks', title='hp comparison (k,p,b)')
    ax.legend(loc='best');

def
parameters_search(N,paramatrix,model,i_true,state,duration,num,vaccination=None):
    tested_hp = [] # every possible result computed
    best_res = None
    while True:
        params = validate_params(paramatrix) #checks params, lower bound, upper
bount, dtype
        print(params) #print the params that are going to be tested
        # search the pandemic parameters over the subset, grid search + average
of 10x simulations for each set
        sc_stat_par,tested_hp =
search_pandemic(tested_hp,N,SIR,params,NI_true,status,simulation_time,attempts,vacc,best_res)
        if not best_res or sc_stat_par[0] < best_res[0]: #if new or it achieved a
better score
            paramatrix[0,:] = sc_stat_par[2]['k'],sc_stat_par[2]
['b'],sc_stat_par[2]['p'] #change the k0,p0,b0 center parameters
            best_res = sc_stat_par
            elif sc_stat_par[2] == best_res[2] and paramatrix[1,1] >= 0.05:

```

```

        #if equals reduce the step dp,db (dk is already minimum) down to
        0.025 then it stops so it does not cycle unendly
        paramatrix[1,1:] /= 2
        elif sc_stat_par[0] >= best_res[0]: # stops the search
            break

    plot_avg_state(best_res[1],range(duration),vaccination)
    plot_newly_infected(range(duration),best_res[1][-1,:],i_true)
    print(best_res[0],best_res[2])

def validate_param(vals,lb=0,ub=1,integ=False):
    valval = []
    for v in vals:
        if v < lb: #lower bound
            v = lb
        elif v > ub: #upper bound
            v = ub
        if integ: #dtype
            v = int(v)
        else: #round the result to avoid ugly params as 0.9999999999999999
            v = np.round(v, 3)
        valval.append(v)
    return valval

def validate_params(paramatrix): #construct the validated params dictionary
    k0, b0, p0 = paramatrix[0,:]
    dk, db, dp = paramatrix[1,:]
    params = {
        'k': validate_param([k0-dk, k0, k0+dk],2,934,True),
        'b': validate_param([b0-db, b0, b0+db]),
        'p': validate_param([p0-dp, p0, p0+dp])
    }
    return params

```

```

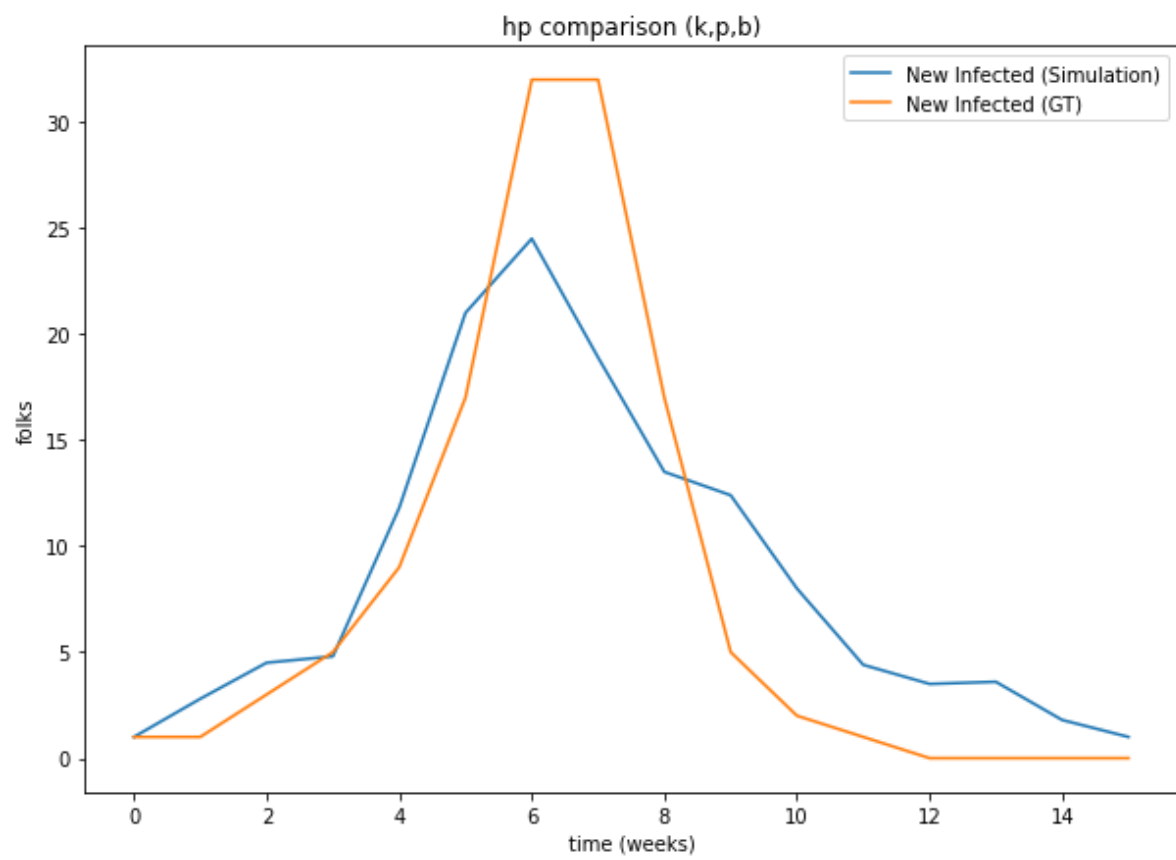
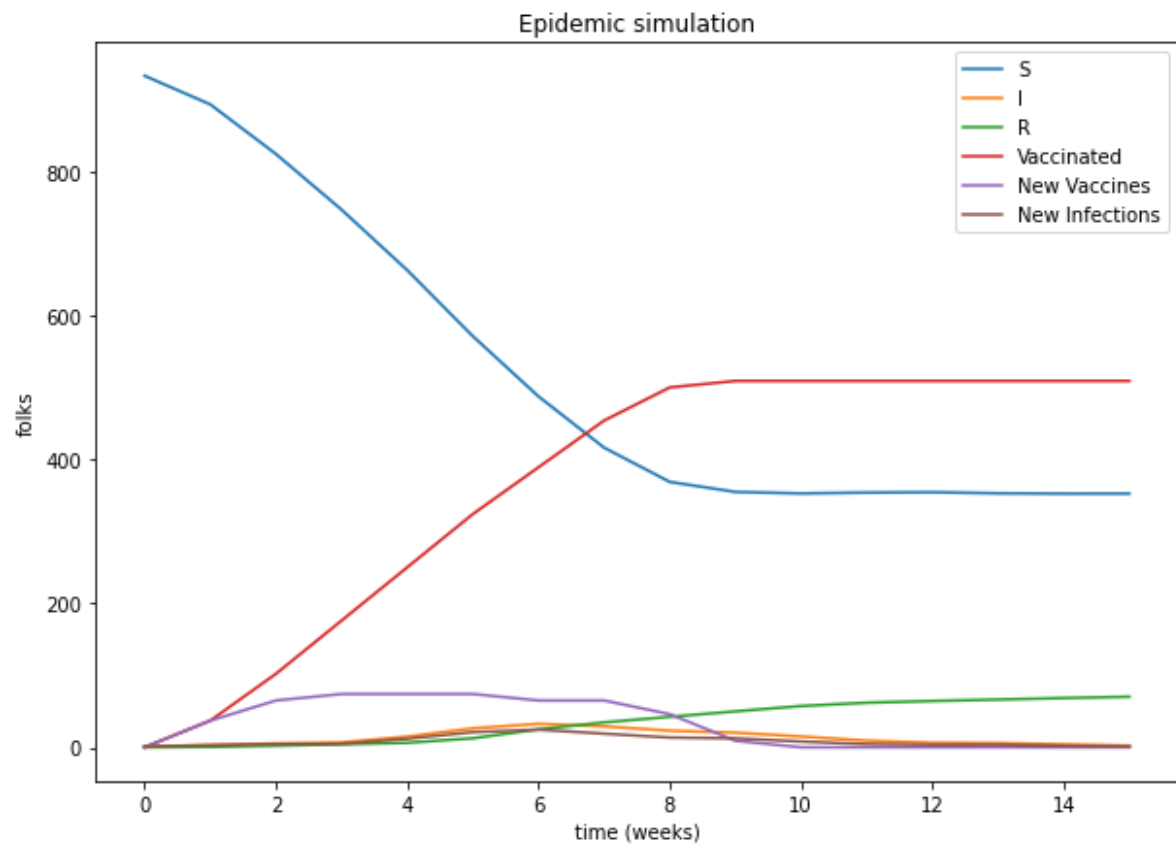
parameters_search(N,paramatrix,SIR,NI_true,status,simulation_time,attempts,vacc)

```

```

{'k': [9, 10, 11], 'b': [0.2, 0.3, 0.4], 'p': [0.5, 0.6, 0.7]}
{'k': [8, 9, 10], 'b': [0.1, 0.2, 0.3], 'p': [0.5, 0.6, 0.7]}
{'k': [9, 10, 11], 'b': [0.0, 0.1, 0.2], 'p': [0.5, 0.6, 0.7]}
{'k': [9, 10, 11], 'b': [0.05, 0.1, 0.15], 'p': [0.55, 0.6, 0.65]}
{'k': [9, 10, 11], 'b': [0.075, 0.1, 0.125], 'p': [0.575, 0.6, 0.625]}
5.0078064060025325 {'b': 0.1, 'k': 10, 'p': 0.6}

```



2.8613807855648994 {'b': 0.19999999999999998, 'k': 11, 'p': 0.5}

Exercise 5 Code

```
import networkx as nx
import numpy as np
from numpy.random import choice, rand
import numpy as np
import matplotlib.pyplot as plt

#from mpl_toolkits.mplot3d import Axes3D
from mpl_toolkits import mplot3d
from scipy.sparse import lil_matrix, csr_matrix, find
np.random.seed(42)
```

Previous code

```
def infection(m,beta):
    infection_probability = 1 - (1-beta)**m
    return 1 if np.random.rand(1)[0] <= infection_probability else 0

def recovery(p):
    return 1 if np.random.rand(1)[0] >= p else 0

def init_stats(duration,state):
    st = np.zeros([duration,state.shape[0]])
    stats = lil_matrix(st,shape=(duration,state.shape[0]),dtype=int) #faster
    sparse matrix creation/population
    stats[0,:] = state
    return stats

def group_sum(stats,row):
    previous_state = np.array(find(stats.getrow(row)))[1:]
    zeros = stats.shape[1]-previous_state.shape[1]
    count = {0:zeros}
    for idx in range(previous_state.shape[1]):
        node = previous_state[0,idx]
        value = previous_state[1,idx]
        count[value] = count.get(value,0) + 1
    return count

def count_new_state_increment(stats,moment, key=1):
    split = stats[moment-1:moment+1,:]
    cii = 0
    for n in range(split.shape[1]):
        if split[1,n] == key and split[0,n] != key:
            cii += 1
    return cii

def epidemic_simulation(G,model,state,duration,beta,p,vaccination=None):
    stats = init_stats(duration,state)
    if vaccination:
        vaccinated = np.array([])
    for moment in range(1,duration):
        if vaccination:
```

```

        vaccines = int((vaccination[moment]-vaccination[moment-1])/100
*state.shape[0])
        to_be_vaccinated = choice([x for x in range(state.shape[0]) if x not
in vaccinated], vaccines, replace=False)
        for vac in to_be_vaccinated:
            stats[moment:,vac] = 3
        vaccinated = np.concatenate([vaccinated,to_be_vaccinated])
    for node in G.nodes:
        if stats[moment,node] == 3:
            continue
        node_state = stats[moment-1,node]
        if node_state == 0: #susceptible
            neighbors = [n for n in G.neighbors(node)]
            infected_neighbors = [stats[moment-1,x] for x in
neighbors].count(1)#group_sum(stats,moment,1,neighbors)
            stats[moment,node] = infection(infected_neighbors,beta)
        elif node_state == 1: #infected
            recovered = recovery(p)
            stats[moment,node] = node_state + recovery(p)
            if recovered > 0: #recovered
                stats[moment:,node] = 2
        else:
            continue

    stats = csr_matrix(stats) #faster sparse matrix computation
    return stats

def multiple_simulation(G,model,state,duration,beta,p,num,vaccination=None,
plot=True):
    stats = epidemic_simulation(G,model,state,duration,beta,p,vaccination)
    for i in range(1,num):
        stats =
np.column_stack((stats,epidemic_simulation(G,model,state,duration,beta,p,vaccinat
ion)))
    ev_rate = compute_evolution(stats,duration,vaccination)
    if plot:
        duration = range(duration)
        return plot_avg_state(ev_rate,duration,vaccination)
    else:
        return ev_rate

def compute_evolution(stats,duration,vaccination=None):
    initial_step = group_sum(stats[0,0],0)
    initial_state = np.zeros(4 + (2 if vaccination else 0))
    for k,v in initial_step.items():
        initial_state[k] = v
    initial_state[-1] = initial_step[1]
    evolution_rate = initial_state
    for moment in range(1,duration):
        if vaccination:
            avg_state = np.zeros(6)
        else:
            avg_state = np.zeros(4)
        for simulation in range(0,stats.shape[1]):
            count = group_sum(stats[0,simulation],moment)
            for k,v in count.items():
                avg_state[k] += v

```

```

        avg_state[-1] +=
count_new_state_increment(stats[0,simulation],moment)
        if vaccination:
            avg_state[-2] +=
count_new_state_increment(stats[0,simulation],moment,key=3)
            avg_state = avg_state / stats.shape[1]
            evolution_rate = np.column_stack((evolution_rate, avg_state))
        return evolution_rate

def plot_avg_state(evolution_rate,duration,vaccination=None):
    #evolution_rate = compute_evolution(stats,duration,vaccination)
    ig, ax= plt.subplots(figsize=(10,7))
    ax.plot(duration, evolution_rate[0,:], label='S')
    ax.plot(duration, evolution_rate[1,:], label='I')
    ax.plot(duration, evolution_rate[2,:], label='R')
    if vaccination:
        ax.plot(duration, evolution_rate[3,:], label='Vaccinated')
        ax.plot(duration, evolution_rate[-2,:], label='New Vaccines')
    ax.plot(duration, evolution_rate[-1,:], label='New Infections')
    ax.set(xlabel='time (weeks)',ylabel='folks', title='Epidemic simulation')
    ax.legend(loc='best');
    return evolution_rate
    #print(evolution_rate)

def RMSE(i,i_true):
    return np.sqrt(1/i.shape[0]*sum((i-i_true)**2))

def
search_pandemic(tested_hp,N,model,params,i_true,state,duration,num,vaccination=None,best_res=None):
    grid = ParameterGrid(params)
    best_res = None
    for params in grid:
        if(params in tested_hp):
            continue
        else:
            tested_hp.append(params)
            rg = RandomGraphGenerator(N,params['k'])
            ev_rate =
multiple_simulation(rg.G,model,state,simulation_time,params['b'],params['p'],attempts,vacc,plot=False)
            newly_inf = ev_rate[-1,:]
            result = RMSE(newly_inf,i_true)
            if not best_res or result < best_res[0]:
                best_res = [result,ev_rate,params]
    return best_res, tested_hp

def plot_newly_infected(duration,i,i_true):
    ig, ax= plt.subplots(figsize=(10,7))
    ax.plot(duration, i, label='New Infected (Simulation)')
    ax.plot(duration, i_true, label='New Infected (GT)')
    ax.set(xlabel='time (weeks)',ylabel='folks', title='hp comparison (k,p,b)')
    ax.legend(loc='best');

```

```

def
parameters_search(N,paramatrix,model,params,i_true,state,duration,num,vaccination
=None):
    tested_hp = []
    best_res = None
    while True:
        params = validate_params(paramatrix)
        print(params)
        sc_stat_par, tested_hp =
search_pandemic(tested_hp,N,SIR,params,NI_true,status,simulation_time,attempts,va
cc,best_res)
        if not best_res or sc_stat_par[0] < best_res[0]:
            paramatrix[0,:] = sc_stat_par[2]['k'],sc_stat_par[2]
['b'],sc_stat_par[2]['p']
            best_res = sc_stat_par
        elif sc_stat_par[2] == best_res[2] and paramatrix[1,1] >= 0.1:
            paramatrix[1,1:] /= 2
        elif sc_stat_par[0] > best_res[0]:
            break

        plot_avg_state(best_res[1],range(duration),vaccination)
        plot_newly_infected(range(duration),best_res[1][-1,:],i_true)
        print(best_res[0],best_res[2])

def validate_param(vals,lb=0,ub=1,integ=False):
    valval = []
    for v in vals:
        if v < lb:
            v = lb
        elif v > ub:
            v = ub
        if integ:
            v = int(v)
        valval.append(v)
    return valval

def validate_params(paramatrix):
    k0, b0, p0 = paramatrix[0,:]
    dk, db, dp = paramatrix[1,:]
    params = {
        'k': validate_param([k0-dk, k0, k0+dk],2,934,True),
        'b': validate_param([b0-db, b0, b0+db]),
        'p': validate_param([p0-dp, p0, p0+dp])
    }
    return params

```

Configuration model on a normal distribution

```

class RGConfModel():
    def __init__(self,N,mu,sigma=None):
        self.N = N
        self.mu = mu
        self.sigma = sigma if sigma else mu/4
        self.WCM = np.zeros((N,N))
        self.available_targets = np.arange(N)
        self.residual_degrees = None

```

```

self.G = None
self.init_graph_distribution()
self.init_graph()

def init_graph(self):
    #self.print(self.residual_degrees)
    for i in range(self.N):
        # a target is available if its residual in degree is positive
        available_targets = np.argwhere(self.residual_degrees > 0).flatten()
        if available_targets.shape[0] == 0:
            break
        # randomly connect the link to the targets
        targets = np.random.choice(available_targets, self.residual_degrees[i],
[ self.available_targets/np.sum(self.residual_degrees)])
        self.residual_degrees[i] -= targets.shape[0]
        for target in targets:
            if self.residual_degrees[target] < 1:
                available_targets =
self.available_targets[np.argwhere(self.residual_degrees > 0)].flatten()
                if available_targets.shape[0] == 0:
                    break
            #self.print(available_targets)
            target = np.random.choice(available_targets,1)
            self.WCM[i, target] = 1
            self.WCM[target, i] = 1
            self.residual_degrees[target] -=1
self.G = nx.from_numpy_array(self.WCM, create_using=nx.Graph)

def init_graph_distribution(self):
    s = np.random.normal(self.mu, self.sigma, self.N) #Select a number through a
normal distribution, given a mean and a standard deviation
    n = s.astype(np.int32) #Cast it to integer
    #self.print(np.sum(s)/s.shape[0], np.std(s,
ddof=1), np.sum(n)/n.shape[0], np.std(n, ddof=1))
    n[n<1] = 1 #Check if the number of edges is suitable
    nodes_to_add_remove = int(sum(s) - sum(n)) #adds randomly some connection in
order to minimize the int cast error
    ex = np.random.choice(range(self.N), abs(nodes_to_add_remove)) #Inside all
the possible nodes, select a number of nodes to add (for example, in N=100 nodes,
take 8 nodes)
    for i in ex: #Iterate over the selected node
        if nodes_to_add_remove > 0:
            n[i] += 1
        else:
            if n[i] > 2:
                n[i] -= 1
            else:
                while True:
                    new_index = np.random.randint(self.N)
                    if n[new_index] > 2:
                        n[new_index] -= 1
                    break
    #self.print(np.sum(n)/n.shape[0], np.std(n, ddof=1))
    self.residual_degrees = np.array(n)

def print(self, *param):
    print(param)

```



```

def draw(self):
    nx.draw(self.G)

def draw_circular(self):
    pos = nx.circular_layout(self.G)
    nx.draw(self.G, pos)

```

Iterative Random Search functions

```

from sklearn.model_selection import ParameterGrid
from numpy import linspace

def RMSE(i,i_true):
    return np.sqrt(1/i.shape[0]*sum((i-i_true)**2))

def plot3D(results):
    dataset = [x[2] for x in results]
    x = [x['mu'] for x in dataset]
    y = [y['b'] for y in dataset]
    z = [z['p'] for z in dataset]
    size = [30*s['sigma'] for s in dataset]
    color_score = np.log([x[0]+1 for x in results])

    fig,ax = plt.subplots(figsize=(15,10))
    ax = plt.axes(projection = '3d')
    ax.grid(b = True, color = 'grey',
            linestyle = '-.', linewidth = 0.3,
            alpha = 0.2)
    my_cmap = plt.get_cmap('viridis')
    sctt = ax.scatter3D(x, y, z,
                       alpha = 0.8,
                       c = color_score,
                       cmap = my_cmap,
                       s = size,
                       marker = 'o')

    plt.title("Parameter Space")
    ax.set_xlabel('Mu', fontweight = 'bold')
    ax.set_ylabel('Beta', fontweight = 'bold')
    ax.set_zlabel('Ro', fontweight = 'bold')
    fig.colorbar(sctt, ax = ax, shrink = 0.3, aspect = 5)

    # show plot
    plt.show()

def plot_newly_infected(duration,i,i_true):
    ig, ax= plt.subplots(figsize=(10,7))
    ax.plot(duration, i, label='New Infected (Simulation)')
    ax.plot(duration, i_true, label='New Infected (GT)')
    ax.set(xlabel='time (weeks)',ylabel='folks', title='hp comparison (k,p,b)')
    ax.legend(loc='best');

```

```

def extend_interval(x1,x2,ub=1,lb=0,quantity=None):
    if not quantity:
        quantity = np.abs(x2-x1)/10
    return validate_param([(x1-quantity),(x2+quantity)],ub=ub,lb=lb)

def choose_random_params(params):
    ps = {}
    for k,vs in params.items():
        ps[k] = np.random.uniform(vs[0],vs[1])
    return ps

def
iterative_random_search_simulation(k,q,prev_res,attempts,N,model,i_true,state,dur
ation,vaccination=None):
    #prev_res, k best results, q random space points

    if prev_res:
        #print([x[2] for x in prev_res])
        results = prev_res[:]
        best_ress = sorted(results, key=lambda x: x[0])[:k]
        par_sp = [x[2] for x in best_ress]
        params_space = {
            'mu': extend_interval(min(x['mu'] for x in par_sp),max(x['mu'] for x in
par_sp),ub=20,lb=2,quantity=1),
            'sigma': [min(x['sigma'] for x in par_sp),max(x['sigma'] for x in
par_sp)],
            'p': extend_interval(min(x['p'] for x in par_sp),max(x['p'] for x in
par_sp)),
            'b': extend_interval(min(x['b'] for x in par_sp),max(x['b'] for x in
par_sp)),
        }
    else:
        results = []
        params_space = {
            'mu': [2,20],
            'sigma': [0,15],
            'p': [0,1],
            'b': [0,1],
        }
    print(params_space)
    for point in range(q):
        params = choose_random_params(params_space)
        print(params)
        CMG = RGSmallWorldModel(N,params['mu'],params['sigma']).G
        ev_rate =
multiple_simulation(CMG,model,state,simulation_time,params['b'],params['p'],attem
pts,vacc,plot=False)
        newly_inf = ev_rate[-1,:]
        result = RMSE(newly_inf,i_true)
        print(result)
        results.append([result,ev_rate,params])

    return results

```

```

def
research_random(iter,N,k,q,attempts,model,i_true,state,duration,vaccination=None)
:
    results = None ## define for first iteration
    for i in range(iter):
        results =
iterative_random_search_simulation(k,q,results,attempts,N,model,i_true,state,dura
tion,vaccination)

    best_res = sorted(results, key=lambda x: x[0])[0]
    #print(best_res)
    plot_avg_state(best_res[1],duration,vaccination)
    plot_newly_infected(duration,best_res[1][-1,:],i_true)
    print(best_res[0],best_res[2])
    plot3D(results)

```

Small World model base onf the Configuration over RN Distribution

```

import time

class RGSsmallWorldModel():
    def __init__(self,N,mu,sigma=None):
        start = time.time()
        self.N = N
        self.mu = mu
        self.percentage = 10
        self.counter = 1
        self.sigma = sigma if sigma else mu/4
        self.WCM = lil_matrix(np.empty((N,N)))
        self.available_targets = np.arange(N)
        self.residual_degrees = None
        self.G = None
        self.nodes_count = 0
        self.init_graph_distribution()
        self.init_graph()
        end = time.time()
        #self.print("time graph creation",end - start)

    def init_graph(self):
        while self.nodes_count < self.N: # while the number of filled nodes is lower
than N
            #new_nodes = np.random.normal(self.mu, self.sigma, 1).astype(int)
            new_nodes = int(self.N * self.percentage / (100 * self.counter)) #93
(10%), 46 (5%), 31 (3%), 23 (2.5%), etc
            if new_nodes >= 5:
                self.counter += 1
            if self.nodes_count + new_nodes > self.N:
                new_nodes = self.N - self.nodes_count
            elif new_nodes < 1:
                break

```

```

nodes = np.arange(self.nodes_count, self.nodes_count + new_nodes)

new_graph_res_deg = self.residual_degrees[nodes[0]:nodes[-1]].copy()
new_graph_res_deg[new_graph_res_deg<1] = 0
if new_nodes <= 2:
    new_sg = nx.Graph()
    new_sg.add_nodes_from(nodes)
    if new_nodes == 2:
        new_sg.add_edge(nodes[0], nodes[1])

else:
    new_sg = self.create_sub_graph(nodes)

if self.G and self.G.number_of_edges() > 0:
    ### additional link to ensure the strongly connection
    norm_degrees = np.array([self.G.degree(x) for x in
self.G.nodes])/(self.G.number_of_edges()*2)
    old_graph_node = choice(self.G.nodes, 1, [x for x in norm_degrees])[0]
    if new_nodes > 2:
        new_graph_node =
choice(nodes, 1, self.normalize_weights(new_graph_res_deg))[0] # new sub graph node
random accordingly to degree distribution
    else:
        new_graph_node = nodes[0]
    self.G = nx.compose(self.G, new_sg)
    self.G.add_edge(old_graph_node, new_graph_node)
    #self.residual_degrees[old_graph_node] -= 1
    #self.residual_degrees[new_graph_node] -= 1
    self.WCM[old_graph_node, new_graph_node] = 1
    self.WCM[new_graph_node, old_graph_node] = 1

else:
    self.G = new_sg

self.nodes_count += new_nodes

#self.draw_circular()
self.WCM = nx.adjacency_matrix(self.G)
self.perturbate_graph()

def create_sub_graph(self, nodes):
    sg = nx.Graph()
    sg.add_nodes_from(nodes)
    for node in nodes:
        if(self.residual_degrees[node] < 1): #saturated node
            continue
        sub_graph_degrees = self.residual_degrees[nodes[0]:nodes[-1]].copy()
#select the nodes that are part of the subgraph
        sub_graph_degrees[self.nodes_count - node] = 0 # avoid self loop
        for _, x in sg.edges(node): # avoid double connections
            if x in nodes:
                sub_graph_degrees[x-nodes[0]] = 0
        if np.sum(sub_graph_degrees[sub_graph_degrees>0]) == 0: #saturated
neighborhood end of cycle
            break

```

```

        num_links =
np.min(np.random.randint(int(self.residual_degrees[node]*3/4), self.residual_degrees[node])) # min (randint(res degree), available_neighbors)
        num_links = num_links if num_links > 0 else 1
        targets = choice(nodes, num_links, self.normalize_weights(sub_graph_degrees))
# choose a random number of connections [1,max_degree] between the normalized
probability of neighbors degrees
        if targets.shape[0] == 0: #no targets
            continue
        #self.residual_degrees[node] -= targets.shape[0]
        for target in targets:
            if self.residual_degrees[target] < 1:
                sub_graph_degrees = self.residual_degrees[nodes[0]:nodes[-1]] #select
the nodes that are part of the subgraph
                if np.sum(sub_graph_degrees) == 0: #saturated neighborhood
                    break
                target = choice(nodes, 1, self.normalize_weights(sub_graph_degrees))[0]
                self.residual_degrees[node] -=1
                self.residual_degrees[target] -=1
                sg.add_edge(node, target)
                self.WCM[node, target] = 1
                self.WCM[target, node] = 1
        #self.draw(sg, title=f'{self.nodes_count}_{nodes.shape[0]}')
        return sg

def perturbate_graph(self):
    for i in range(self.N):
        if(self.residual_degrees[i] < 1):
            continue
        # a target is available if its residual in degree is positive
        available_targets_pos = np.argwhere(self.residual_degrees > 0).flatten() #
np.argwhere(self.residual_degrees > 0).flatten()
        available_targets = self.residual_degrees_copy.copy()
        available_targets[available_targets<0] = 0
        if(np.sum(available_targets)<1):
            break
        for _, x in self.G.edges(i): # avoid double connections
            if x in available_targets_pos:
                available_targets[x] = 0
        if available_targets.shape[0] == 0:
            break
        # randomly connect the link to the targets
        targets =
np.random.choice(available_targets_pos, self.residual_degrees[i], self.normalize_weights(self.residual_degrees))
        for target in targets:
            if self.residual_degrees[target] < 1:
                available_targets =
self.available_targets[np.argwhere(self.residual_degrees > 0)].flatten()
                if available_targets.shape[0] == 0:
                    break
                target = np.random.choice(available_targets, 1)[0]
                self.residual_degrees[i] -=1
                self.WCM[i, target] = 1
                self.WCM[target, i] = 1
                self.residual_degrees[target] -=1
                self.G.add_edge(i, target)
        #self.G = nx.from_numpy_array(self.WCM.todense(), create_using=nx.Graph)

```

```

def init_graph_distribution(self):
    s = np.random.normal(self.mu, self.sigma, self.N) #Select a number through a
normal distribution, given a mean and a standard deviation
    n = s.astype(np.int32) #Cast it to integer
    n[n<1] = 1 #Check if the number of edges is suitable
    nodes_to_add_remove = int(sum(s) - sum(n)) #adds randomly some connection in
order to minimize the int cast error
    ex = np.random.choice(range(self.N), abs(nodes_to_add_remove)) #Inside all
the possible nodes, select a number of nodes to add (for example, in N=100 nodes,
take 8 nodes)
    for i in ex: #Iterate over the selected node
        if nodes_to_add_remove > 0:
            n[i] += 1
        else:
            if n[i] > 2:
                n[i] -= 1
            else:
                while True:
                    new_index = np.random.randint(self.N)
                    if n[new_index] > 2:
                        n[new_index] -= 1
                        break
    self.residual_degrees = np.array(n)
    self.residual_degrees_copy = self.residual_degrees.copy()

def print(self, *param):
    for i,x in enumerate(param):
        print(f"{i}) {x}")

def normalize_weights(self, arr):
    w = arr.copy()
    w[w<0] = 0
    den = np.sum(w)
    den = den if den > 0 else 1
    return [arr/den]

def draw(self, gr=None, title=None):
    if not gr:
        gr = self.G
    fig = plt.figure()
    title = title if title else "graph"
    fig.suptitle(title, fontsize=20)
    nx.draw(gr)

def draw_circular(self, gr=None, title=None):
    if not gr:
        gr = self.G
    fig = plt.figure()
    title = title if title else "graph"
    fig.suptitle(title, fontsize=20)
    pos = nx.circular_layout(self.G)
    nx.draw(gr, pos)

```

```

def draw_centrality(G):
    n_nodes = len(G)
    W = nx.adjacency_matrix(G)
    W = W.toarray()
    w = (W @ np.ones(n_nodes))
    D = np.diag(w)
    P = np.linalg.inv(D) @ W
    n_iter = 50
    x_0 = np.ones(n_nodes)/n_nodes
    tol = 1e-5
    Pt = P.T
    z = np.zeros(n_nodes)
    alpha = 0.85
    for n in range(n_iter):
        new_alpha = np.power(alpha,n)
        P_new = np.linalg.matrix_power(Pt,n)
        z += (new_alpha*P_new) @ x_0
    z = (1-alpha)*z

    pos = nx.spring_layout(G)
    fig = plt.figure()
    fig.suptitle("centrality beta=0.15", fontsize=20)
    nx.draw(test_graph.G, pos,with_labels=True,nodelist=G.nodes,node_size = [d*7000
    for d in z],node_color=z,font_size=8,cmap=plt.cm.summer,)

```