Homework 3

Network Dynamics, and Learning

01TXLSM

Fatemeh Ahmadvand

s301384

2022-2023

Homework(3) 2022-2023

1 Influenza H1N1 2009 Pandemic in Sweden

1.1 Preliminary parts

1.1.1 Epidemic on a known graph

Starting with a k-regular undirected network, each node is linked to the k nodes whose index is closest to the node's modulo in order to replicate a *SIR* epidemic model: The connectivity of the graph may be altered by changing the value of k. (e.g. $k = 1$ cyclic graph C_n, $k = n - 1$ fully connected graph). We simulate the epidemic model for t time units starting from an initial setup where i random nodes are infected (state I) and the other $n - i$ are susceptible (state S). This indicates that there is a chance that a S node will become infected with a probability $P(X_i(t + 1) = I | X_i(t) = S) = 1 - (1 - \beta)^m$ where m is the number of the node's infected neighbors. This means that each infected neighbor has a probability $\beta \in [0, 1]$ of infecting the S node.

Additionally, we define $\rho \in [0, 1]$ as the likelihood of an I node recovering (state R) during a time step $P(X_i(t + 1) = R | X_i(t) = I) = \rho$. With $|V| = 500$ nodes, $k = 4$, $\beta = 0.3$, and $\rho = 0.7$, we replicate the *SIR* epidemic on the graph by choosing 10 random nodes to act as the first infected nodes. We next run this simulation $N = 100$ times, counting the number of nodes in each state (i.e., S, I, R) and the number of S nodes that contract an infection at each time step throughout each simulation. The average behaviors are then plotted after averaging the findings across 100 simulations.

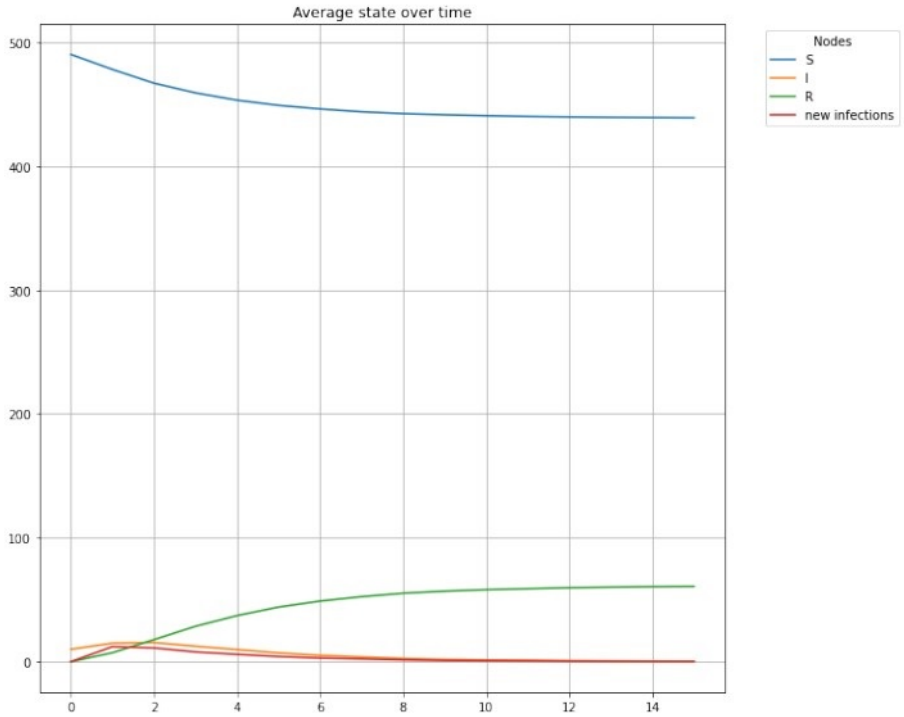


Figure 1: States behavior

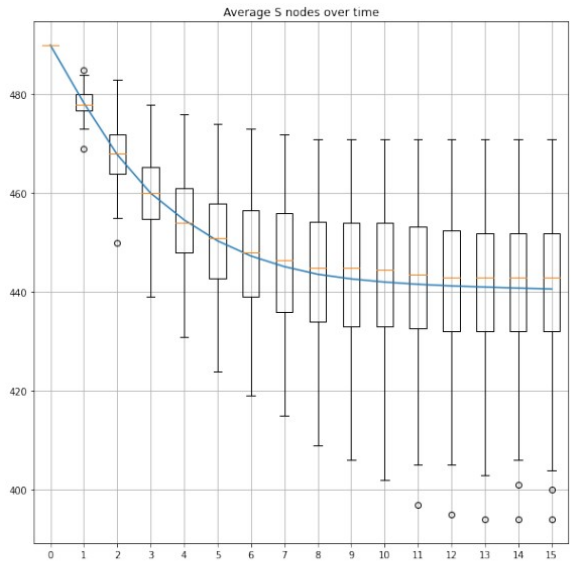


Figure 2: Susceptibles over time

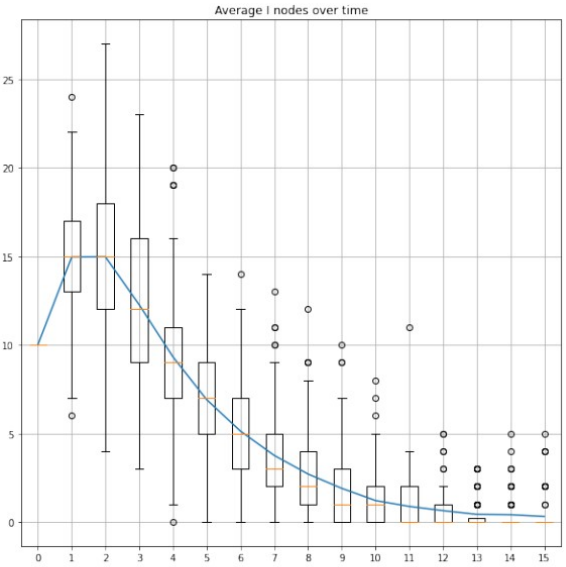


Figure 3: Infected over time

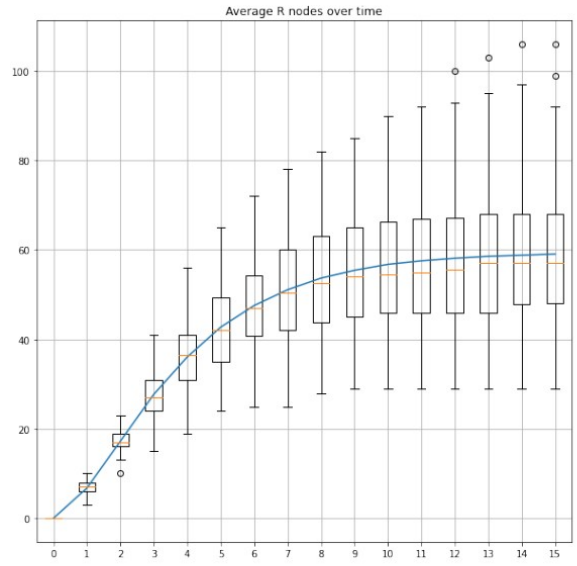


Figure 4: Recovered over time

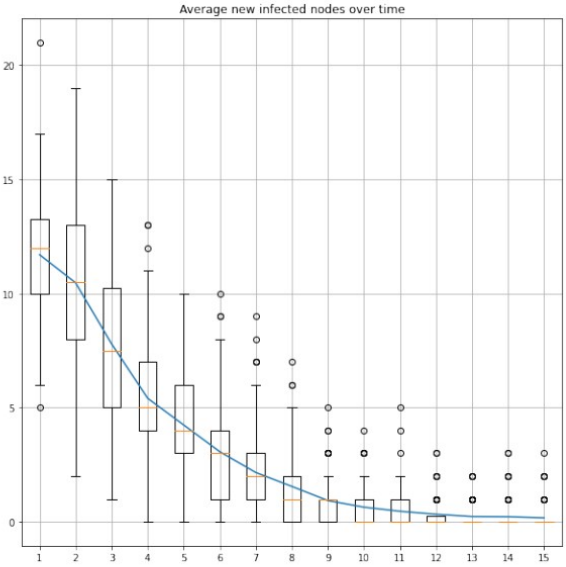


Figure 5: Average new infections

1.1.2 Generate a random graph

With an average degree of $k \in Z^+$ and using the preferential attachment model (PA), our objective is to create a random graph. To create a PA graph, we first build a complete graph with $k+1$ nodes. Then, we continue to add nodes with degrees of $c = k/2$, (alternating between $c = \lfloor k/2 \rfloor$ and $c = \lceil k/2 \rceil$ when k is odd) until we achieve the necessary number of nodes n. With probability $P(W_{n_t,i}(t) = W_{i,n_t}(t) = 1 | G_{t-1} = (V_{t-1}, E_{t-1})) = \dfrac{w_i(t-1)}{\sum_{j \in v} w_i(t-1)}, i \in V_{t-1}$, we connect each node we add to the graph to c previously current nodes. proportionate to the node's degree prior to the inclusion of the additional node n_t. We may get quite diverse outcomes by changing the value of k and the number of nodes in the final graph.

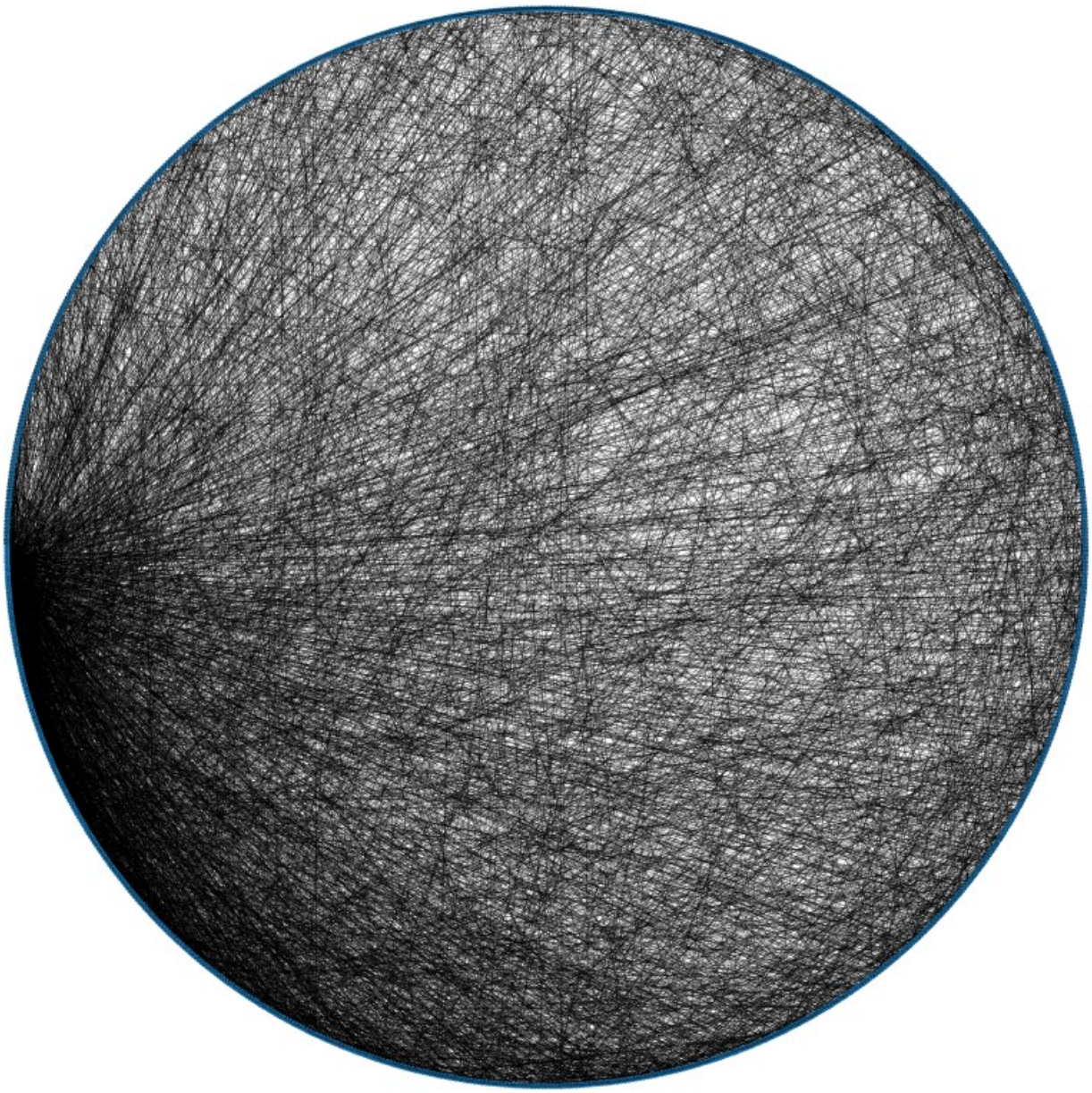


Figure 6: Preferential attachment random graph with $k = 10, |\mathcal{V}| = 900$

1.2 Simulate a pandemic without vaccination

Our objective is to model a *SIR* epidemic on a *PA* graph with $|V| = 500$ nodes, $k = 6$, $\beta = 0.3$, and Similar to the previous example, we start with 10 randomly chosen infected nodes and generate a new random network for every simulation in order to simulate 15-time steps for $N = 100$ simulations. We continue to keep an eye on how the states behave over time as well as the number of people that become sick each week.

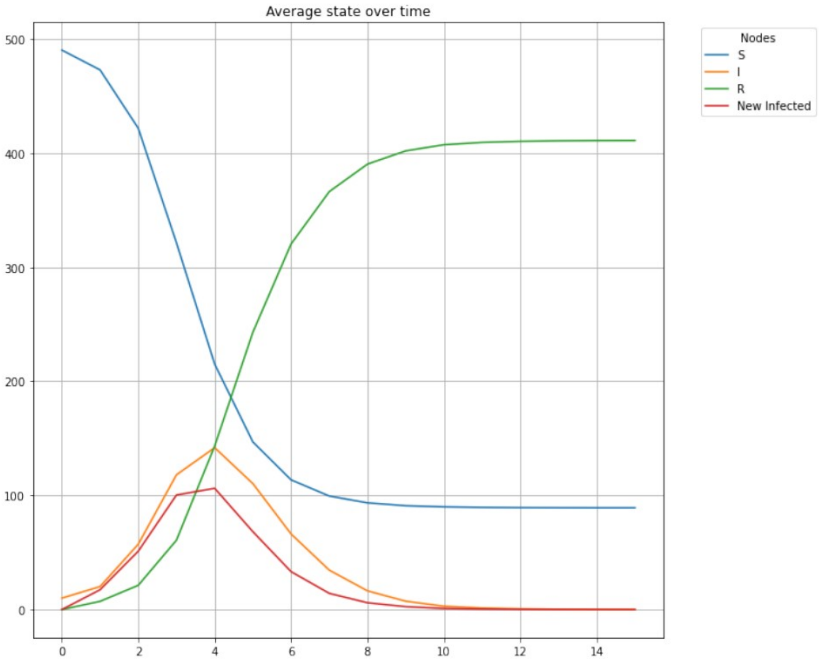


Figure 7: States behavior

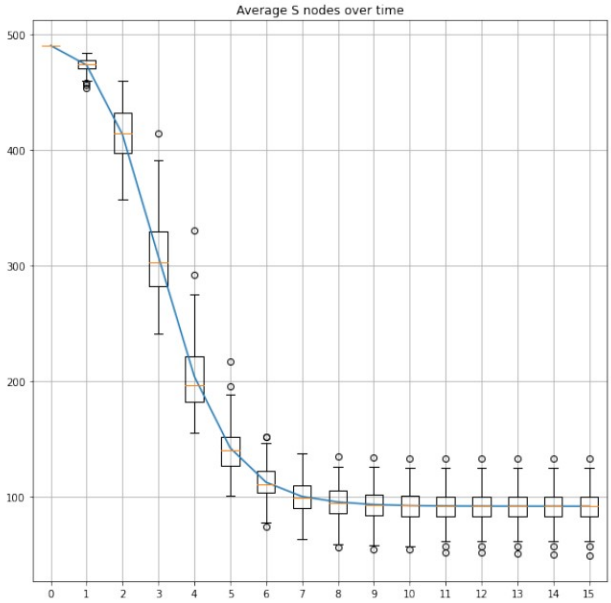


Figure 8: Susceptibles over time

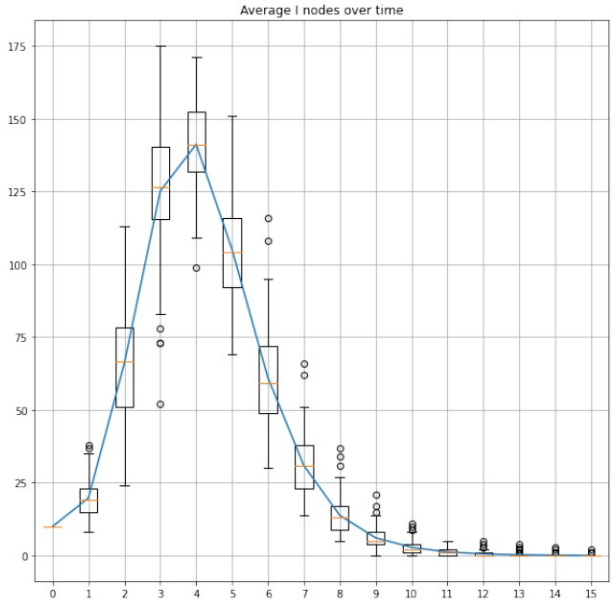


Figure 9: Infected over time

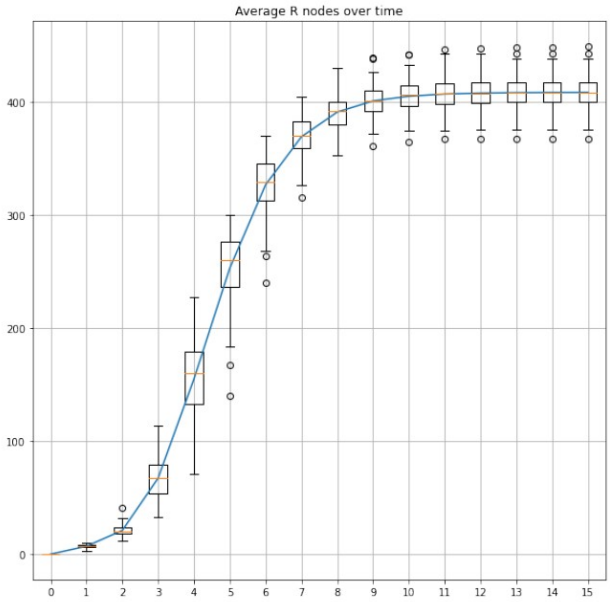


Figure 10: Recovered over time

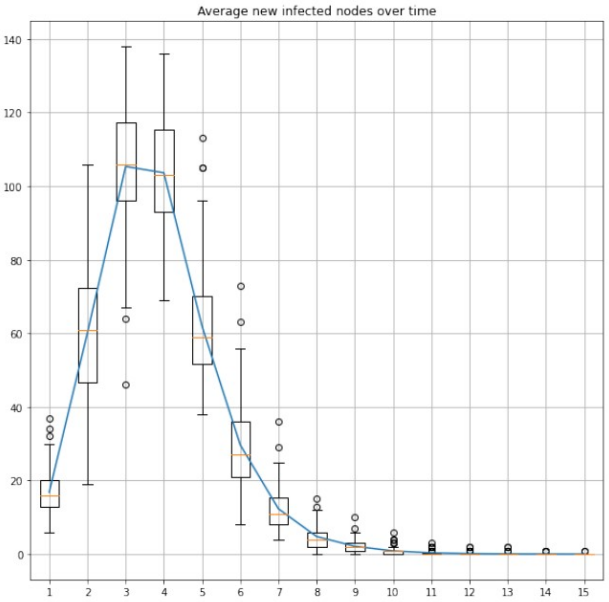


Figure 11: Average new infections

1.3 Simulate a pandemic with vaccination

In this case, we simulate a similar epidemic as before, with the addition of a vaccinated state V. Any node, regardless of its present status, can get a vaccination.

- A vulnerable node will lose the capacity to get an infection from its neighbors if it receives a vaccination.

- A vaccine will prevent an infected node from infecting its neighbors, but the node will still be able to recover. (i.e., change its state from I to R)

- A recovered node receiving a vaccination won't have any effect on the simulation at all.

Vaccinations will be administered in accordance with a policy that specifies the proportion of the population to be immunized each week, and they will take immediate effect (i.e., infected nodes can not contribute to infections in a week t if they get vaccinated in the same week). To make our simulation more accurate, we'll use the following policy.

$$Vacc(t) = [0, 5, 15, 25, 35, 45, 55, 60, 60, 60, 60, 60, 60, 60, 60]$$

where Vacc(t) represents the proportion of the population that has received vaccinations as a whole (e.g., 0% infected in the first week, reaching a plateau of 60% from week 8 on). With the exception of the vaccine mechanism, which precludes some nodes from participating in infection contacts during the simulations, we conduct the epidemic simulation in a fairly similar manner to previous simulations. The overall number of recovered nodes and the number of infected nodes can both be anticipated to decrease over the course of the 15 weeks (i.e. fewer infections).

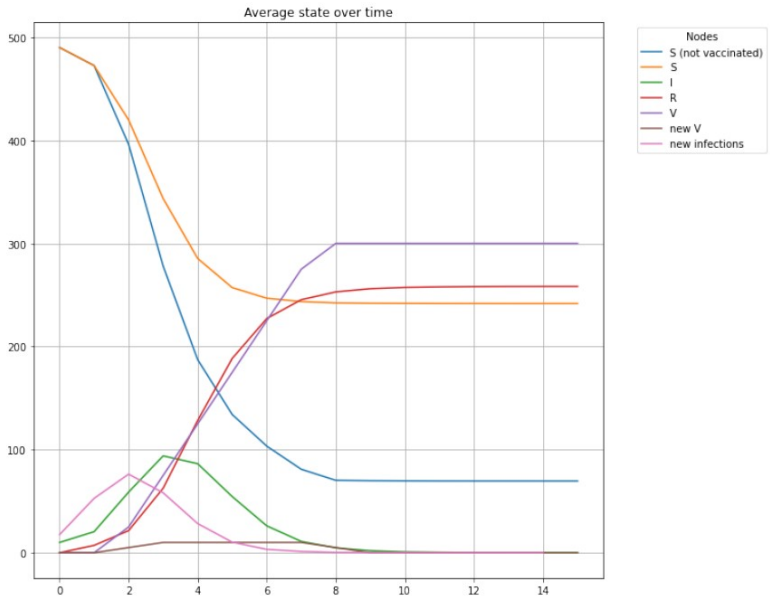


Figure 12: States behaviour

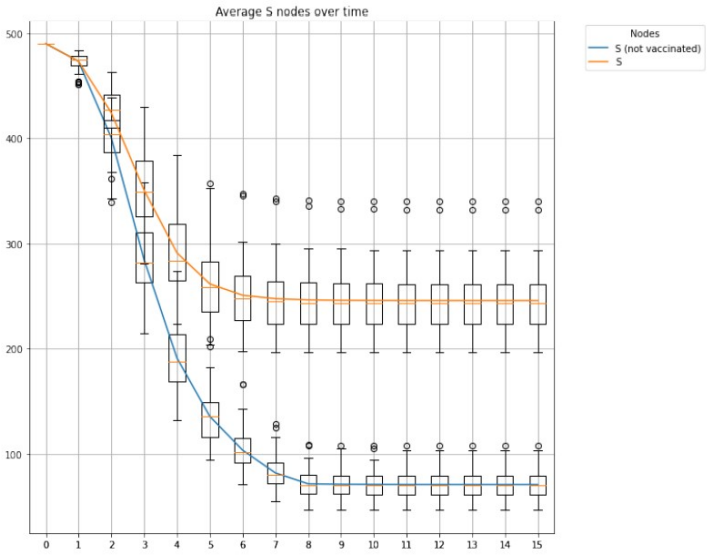


Figure 13: Susceptibles over time

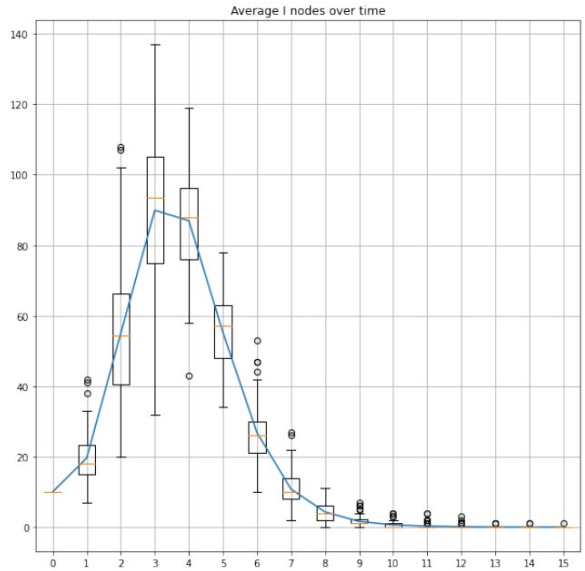


Figure 14: Infected over time

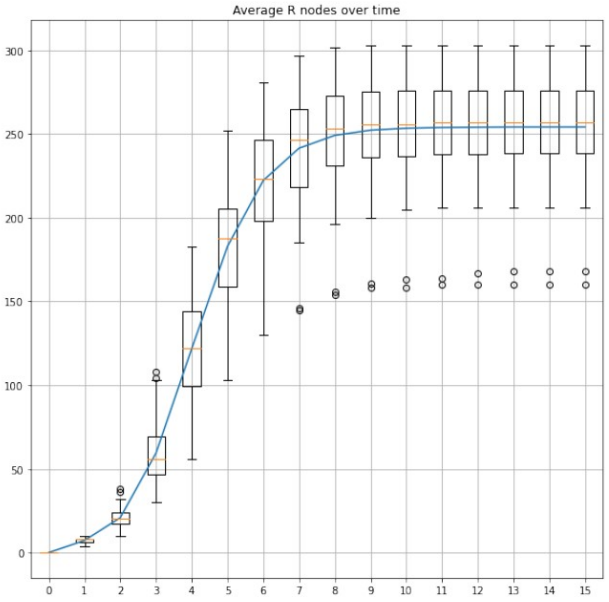


Figure 15: Recovered over time

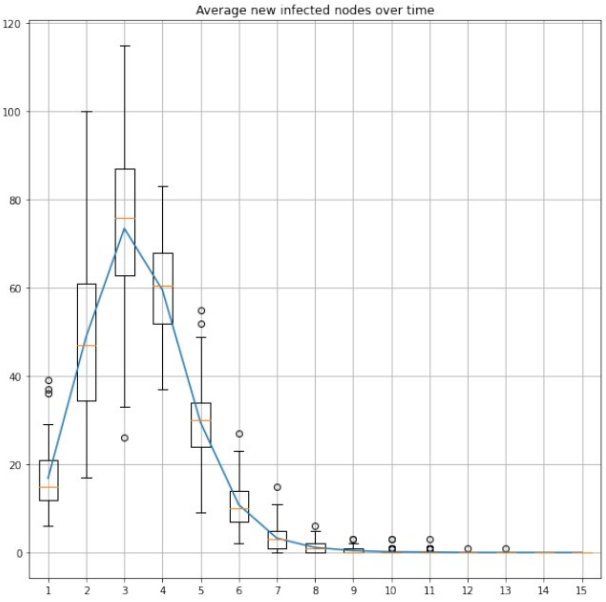


Figure 16: Average new infections

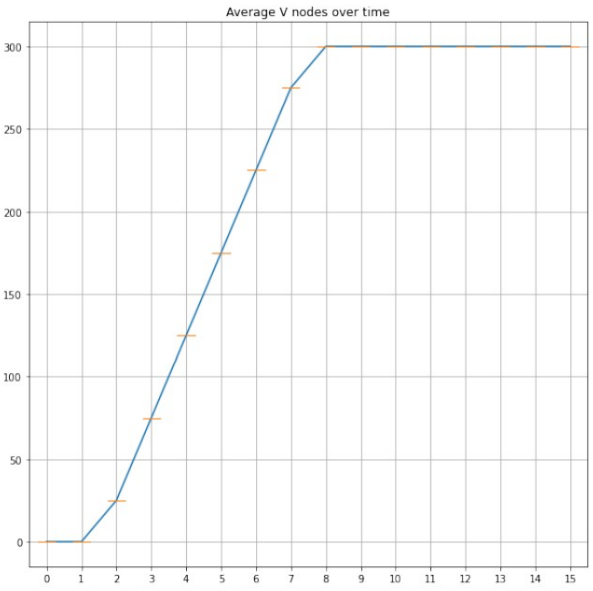


Figure 17: Vaccinated over time

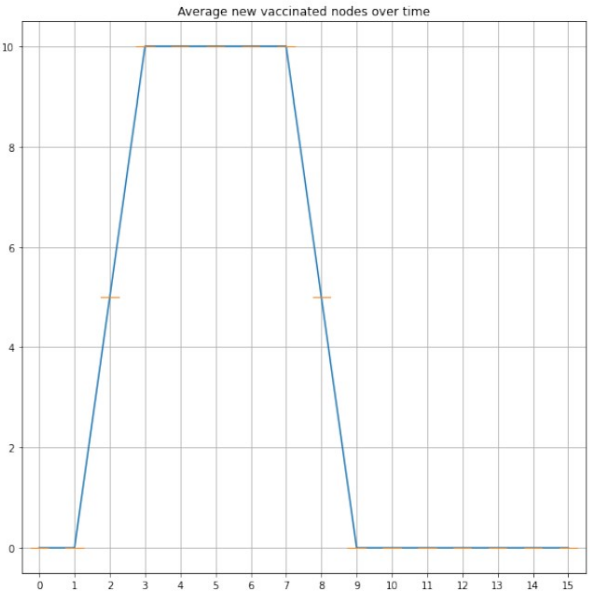


Figure 18: Average new vaccinated

1.4 The H1N1 pandemic in Sweden 2009

By simulating the 15 weeks between the 42nd of 2009 and the 5th of 2010, we will attempt to construct a model based on the 2009 H1N1 pandemic in Sweden using a lower-scale model (scaled down by a factor of 104) collected from real data. We will simulate the pandemic using the model created in point 3 and utilizing the vaccine schedule (derived from data from the H1N1 pandemic).

$$Vacc(t) = [5, 9, 16, 24, 32, 40, 47, 54, 59, 60, 60, 60, 60, 60, 60, 60]$$

By running 10 simulations with the given set of parameters k, β,ρ and calculating the root mean

squared error (*RMSE*) between the average of the 10 simulations and the "true value" indicated by our objective in this scenario will be to best estimate the infections behavior of the real (scaled) pandemic.

$$I_{true}(t) = [1, 1, 3, 5, 9, 17, 32, 32, 17, 5, 2, 1, 0, 0, 0, 0]$$

In order to prevent a pointless simulation, we will choose 1 node at random to be infected at the beginning of each simulation(chosen outside of the set of 5% vaccinated nodes to avoid a useless simulation). 5% of the population will be randomly picked to get the vaccine. We will look for the k, β,ρ parameter values that provide the lowest *RMSE* in an effort to best imitate the genuine epidemic. By investigating each parameter's neighborhood, we attempt to imitate a gradient descent in order to determine the ideal set of parameters.

$$k \in k - \Delta k, k, k + \Delta k$$
$$\beta \in \beta - \Delta\beta, \beta, \beta + \Delta\beta$$
$$\rho \in \rho - \Delta\rho, \rho, \rho + \Delta k$$

The center of our interval is moved to the new local minimum by simulating each combination of parameters and investigating the values of *RMSE* that occur (as an average of 10 iterations). By fixing a seed in the graph-generating process, which results in the construction of the exact same graph for equal values of k, the problem's significant internal variance is reduced. We have 2 alternative termination conditions since we detect a sizable degree of variation between runs for the same set of parameters:

- If the local minimum is the same as the one previously discovered, we remain in that region and lower the value of Δs by a factor of $\gamma = 0.5$ to shrink the neighborhood, which could produce more accurate *RMSE*s. We can reduce intervals up to 2 times: if the minimum hasn't changed after these two iterations, we have discovered a local minimum in the *RMSE* space.
- We can complete our investigation of the *RMSE* space if the best *RMSE* discovered thus far hasn't changed in ten different neighborhoods. This indicates that we may have come across simulations with a particular set of parameters that were unusually lucky and unlikely to be surpassed again.

Using the *PA* graph described in point 2, we observe that, on average, the *RMSE* constantly converges to values in the range $[4, 4.5]$, almost ever deviating from values below 4 and hardly ever failing to fall below 5. but from many experiments, it seems that there are a few different areas of the *RMSE* space that yield loss values around 4, most consistently when $k \sim 6$, $\beta \sim 3$, and $\rho \sim 0.6$. Here are a few results from experiments on the previously defined model:

Figure 19: RMSE: 3.73 with: k=5, β=0.35, ρ=0.55

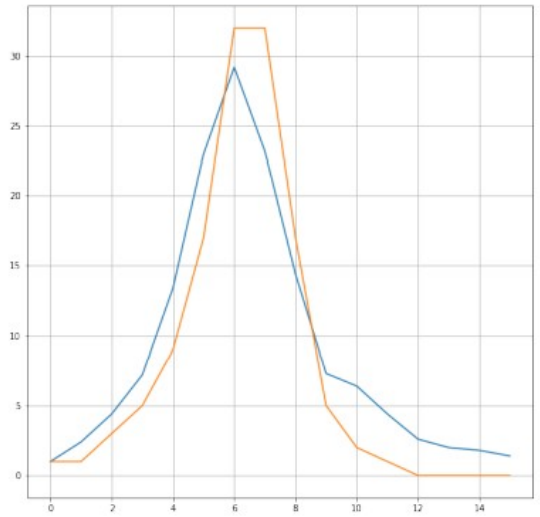


Figure 20: Simulation v Reference

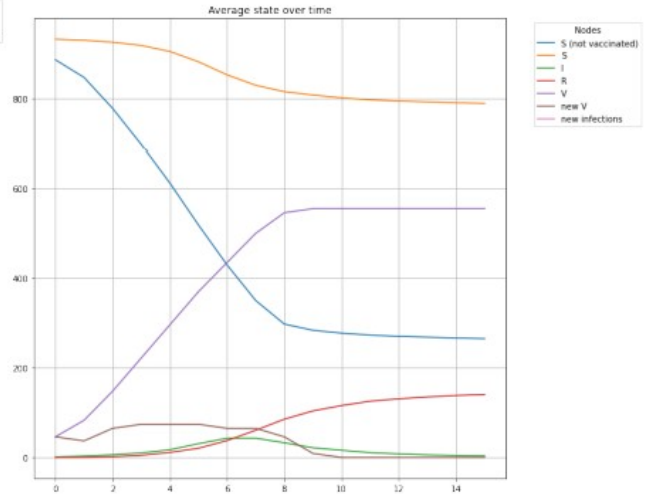


Figure 21: States

Figure 22: RMSE: 4.25 with: k=22, β=0.9, ρ=0.1

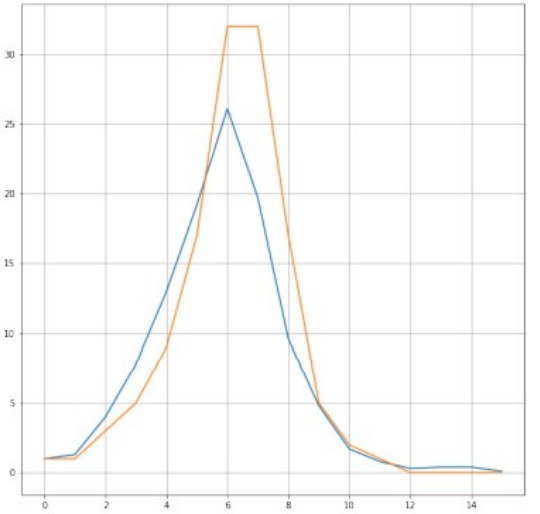


Figure 23: Simulation v Reference

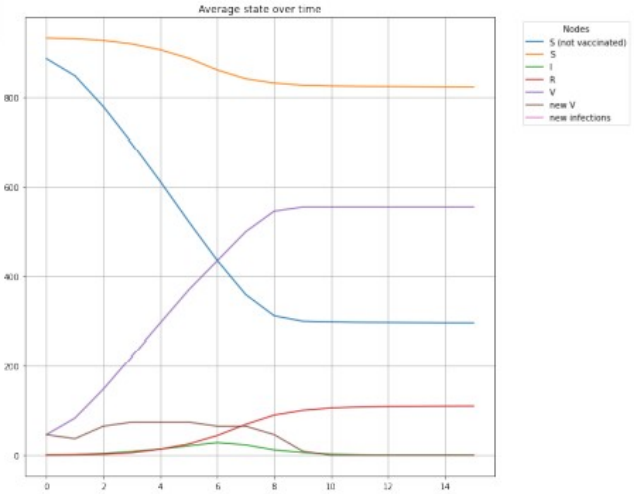


Figure 24: States

Figure 25: RMSE: 4.54 with: k=6, β=0.3, ρ=0.45

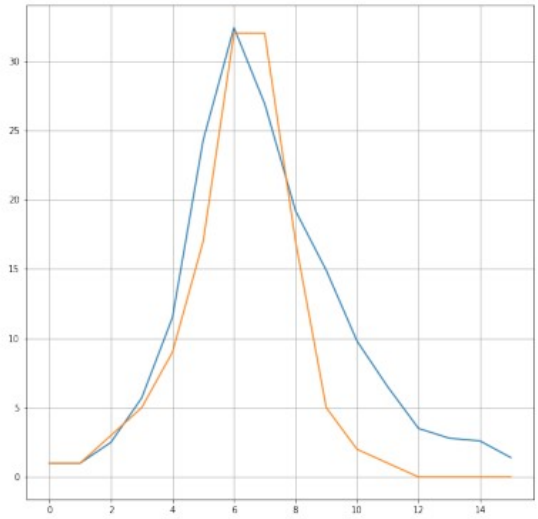


Figure 26: Simulation v Reference

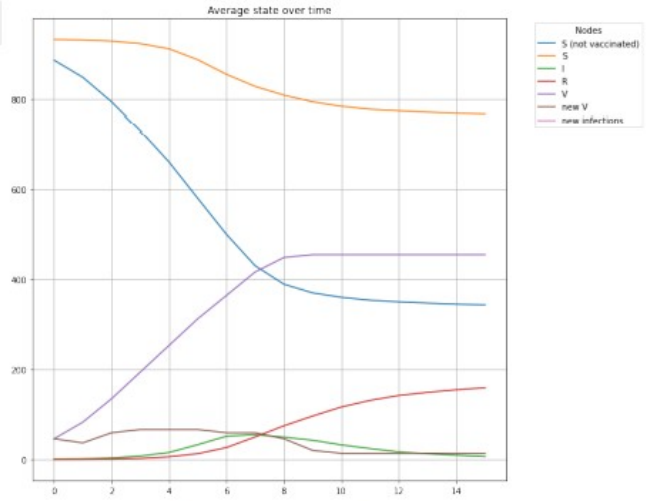


Figure 27: States

Figure 28: RMSE: 4.71 with: k=7, β=0.3, ρ=0.7

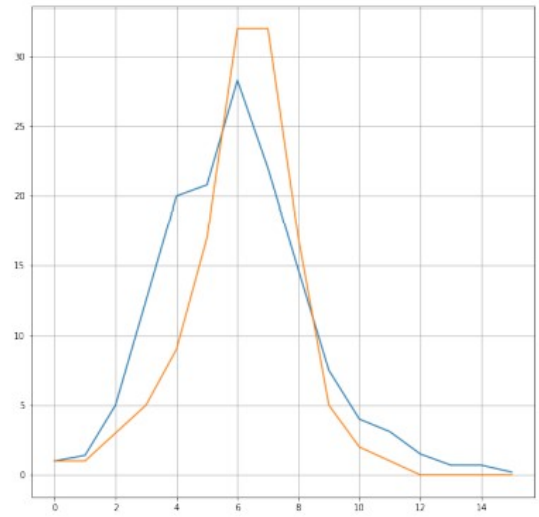


Figure 29: Simulation v Reference

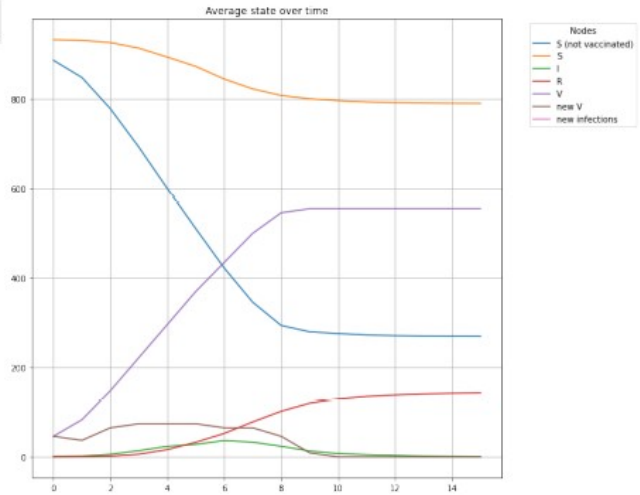


Figure 30: States

1.5 Challenge (optional)

In this section, I will propose some further analysis built upon the model of point 4. As a first experiment, I tried using the same model with a different type of random graph, specifically using Erd˝os–R´enyi (ER) random graph and the Configuration Model (*CM*). Again, we try to reduce variance by fixing a seed when generating the graphs.

1.5.1 Different random graphs

1.5.1.1 Erdo˝s–R´enyi

To generate an *ER* random graph, I generated a list of all possible edges between a couple of nodes and selected at random, with probability $p = \frac{k}{n-1}$, in order to obtain $n-1$ an average degree of k. Additionally, I made the decision to prevent isolated nodes by adding a random edge to nodes with degree $w_i = 0$, as it makes it logical that entirely isolated nodes are very unlikely to be found in our population-simulation model. Even though It is an unlucky event, the occurrence shouldn't have a significant impact on the simulation.

1.5.1.2 Configuration Model

I constructed a vector of degrees with length $|V|$ in accordance with a distribution with a mean value of k to create a *CM* random graph. Since we cannot determine the true degree distribution from the supplied data, we will adopt a uniform distribution in this situation. The degree distribution may be a valuable parameter to tweak in order to acquire various outcomes. Again we will avoid the possibility of having nodes with degree $w_i = 0$ because of the reasons explained above. For each node i with residual degree $w_i > 0$, we will then select at random a number of nodes equal to w_i having residual degree $w_j > 0$ which we will link with node i: In addition, we set the degree of the node and reduce the residual degree by 1 for the chosen nodes to account for the newly added linkages.

1.5.1.3 Results

We observe through several trials that the excessive variation between simulations problem still exists, but the two models behave somewhat differently:

- We routinely obtain less accurate simulations when using the *ER* random graph, with an *RMSE* of $[5, 5.5]$, which seldom ever falls below 5.

- We constantly obtain *RMSE* values in the range $[3.5, 4]$, seldom higher or below, while using a *CM* random graph.

1.5.1.4 ER simulations

Figure 31: RMSE: 4.86 with: k=9, β=0.3, ρ=1

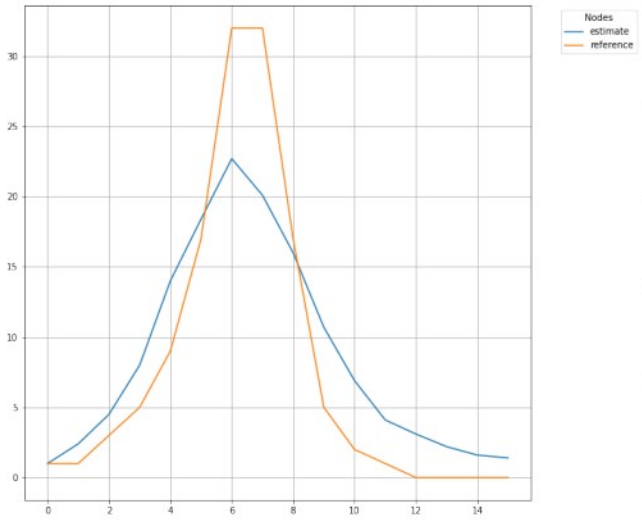


Figure 32: Simulation v Reference

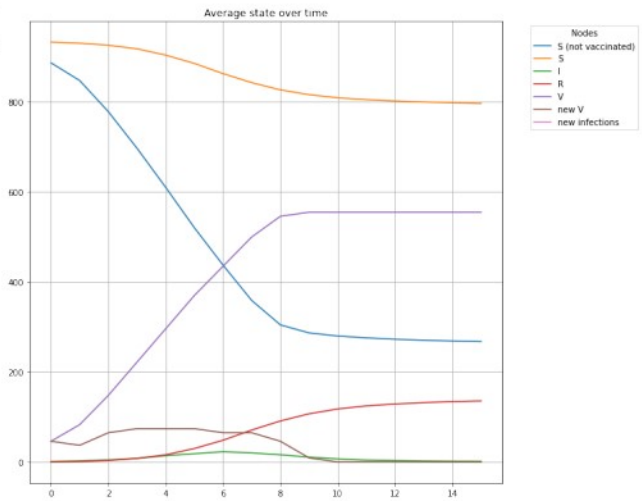


Figure 33: States

Figure 34: RMSE: 5.01 with: k=8, β=0.35, ρ=0.9

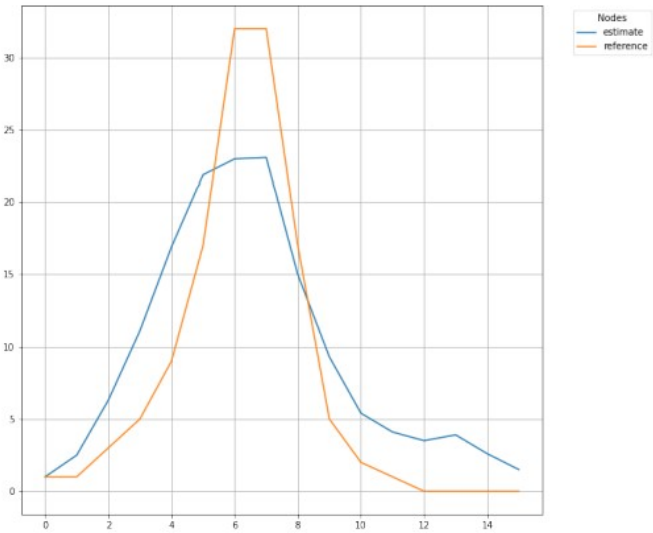


Figure 35: Simulation v Reference

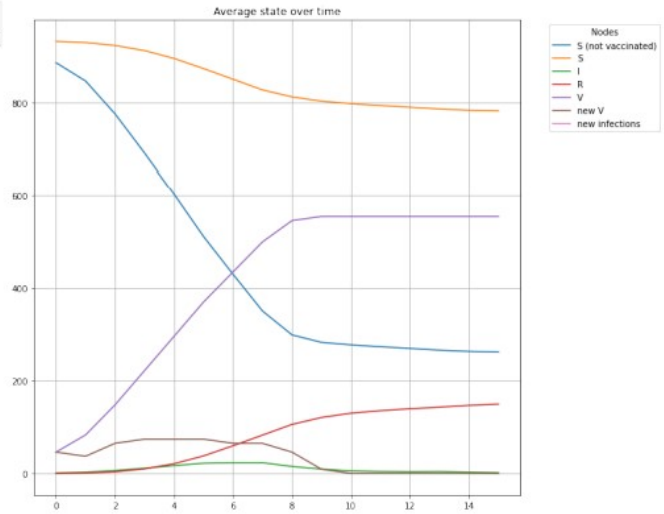


Figure 36: States

1.5.1.5 CM simulations

Figure 37: RMSE: 3.51 with: k=15, β=0.15, ρ=0.9

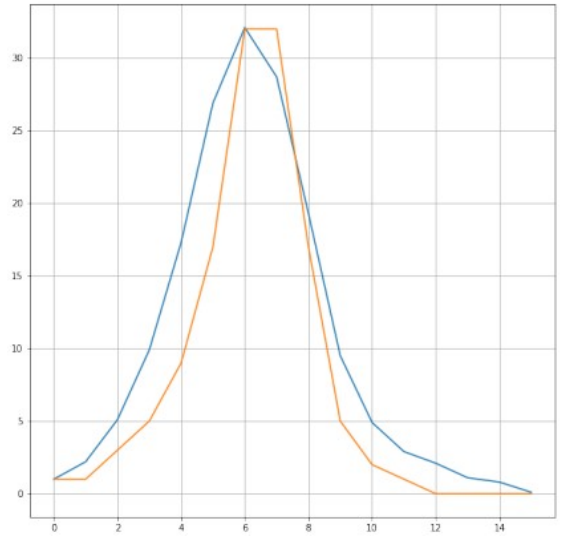


Figure 38: Simulation v Reference

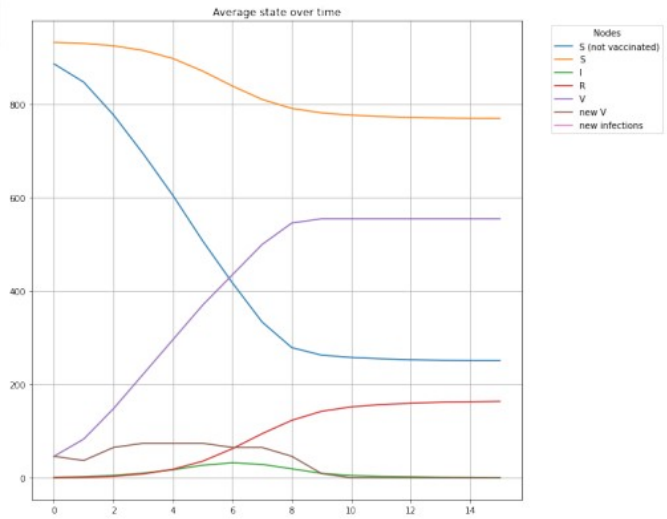


Figure 39: States

Figure 40: RMSE: 3.64 with: k=8, β=0.3, ρ=0.7

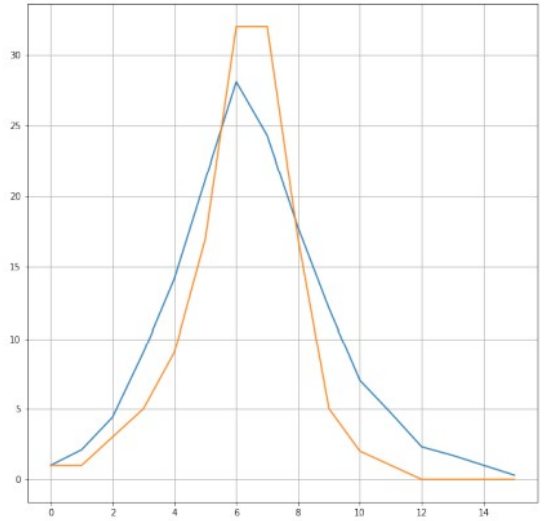


Figure 41: Simulation v Reference

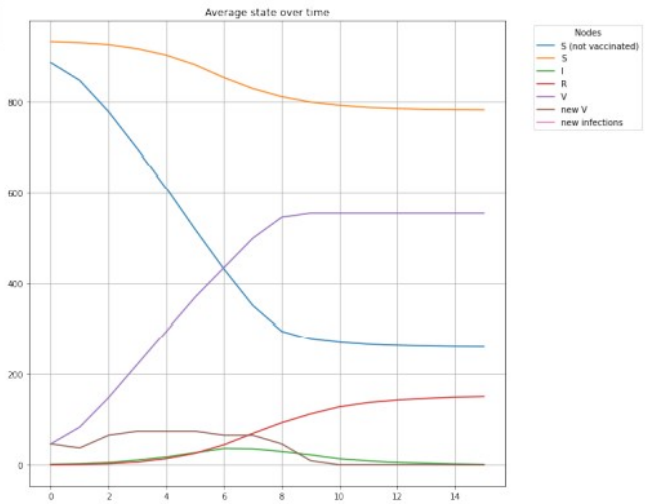


Figure 42: States

1.5.1.6 Conclusions

Given the relatively small difference between the 3 models, we cannot make definite conclusions, as variance is still playing a key role in the model. Also when changing the random graph model, we cannot find a definite set of parameters yielding the best *RMSE*, but we can find intervals for the different parameters consistently yielding similar values of *RMSE*.

1.5.2 Different exploration algorithm

Another approach we could take to improve the results is to consider a different exploration policy. To try and exit local minima, I decided to randomly recenter the neighborhood of parameters by choosing a random $k \in [1, 20]$, $\beta \in [0, 1]$ and $\rho \in [0, 1]$, while also "boosting" the Δs by setting $\Delta k = 5$, $\Delta \beta = 0.3$, $\Delta \rho = 0.3$. By effectively restarting the algorithm ($|repetitions| = 2$) our goal is to try to find another set of parameters that yield better local minima, thus may better simulate the reference values. This approach greatly increases the time needed to perform the simulations and allows to analyze of a much more diverse set of parameters, but it does not really improve the best *RMSE* values of the simulations. The issue again is that variance plays an important role and it's really difficult to overcome in this scenario. In general, we can conclude that the solution space as a function of k, β, ρ has many local minima circumscribed to a few regions of parameters in which the *RMSE* assumes similar values, but because of the considerable amount of the variance between simulations even for the same set of parameters, it's not possible to define which local minima better estimates the real pandemic. To further improve the chances of obtaining more accurate results, one could also try to increase the number of simulations in order to get a better average estimate, but since this approach is extremely computationally demanding, it would be better to further optimize the algorithm first: for example, by parallelizing the algorithm it could be possible to perform more simulations over time, hopefully reducing the amount of variance for a single set of parameters, thus better estimating which set of parameters yields a better average *RMSE*.

2 Coloring

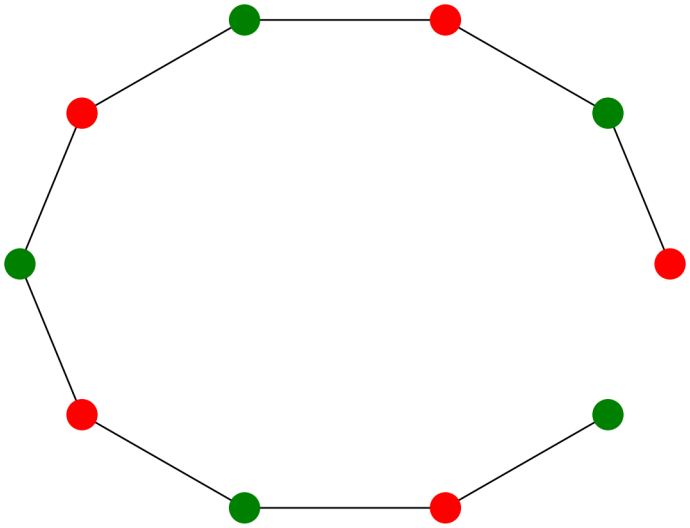
In general, this project has two parts. In the first part, we want to define our graph with ten nodes in such a way that these nodes are placed behind each other in a sequential (linear) arrangement, and we want to implement the coloring algorithm on it by using two colors, green and red. The point that we should pay attention to is that in the coloring algorithm, two nodes with two similar colors should not be placed together.

After defining the graph, we define a matrix by which we specify the relationships. For example, according to the matrix, we see that the first node is only connected to the second node, but the second node is connected to both the first node and the third node. Then we converted the matrix into a "*NumpyArray*" so that we can convert it into a graph using the "*networkx*" library. At first, we added the nodes and then we considered the initial black color for them. And after that, we implemented the coloring algorithm using red and green colors.

We have defined a function *assignColor*() that we will call later. In this way, we have a *node_colors* variable that we use when we want to display the graph in the part where we want to display the graph, so that for each node in the nodes that we defined in the graph if the color of that node was black, the function We call *assignColor*() and finally append the color obtained for the node in the *node_colors* variable.

In general, the work of the *assignColor*() function is that it takes a copy of the variable *color_list* = ['*red*','*green*'] and considers it as *safe_color_list*, and by checking the color of the nodes, if the color of the node is not black, the color of that node which is in *color_list*, it deletes and assigns a color to the next node from inside the list. This procedure is repeated recursively. For example, if our node

was green, it removes green from the list of colors and changes the next node to red. This action is implemented using a backtracking dfs we used this method by using two colors and ten nodes and we color our graph and finally, we saved the graph as a PDF file. Just because the position of the nodes is not important here, we are placed together in a circular manner. whose output is as shown below.

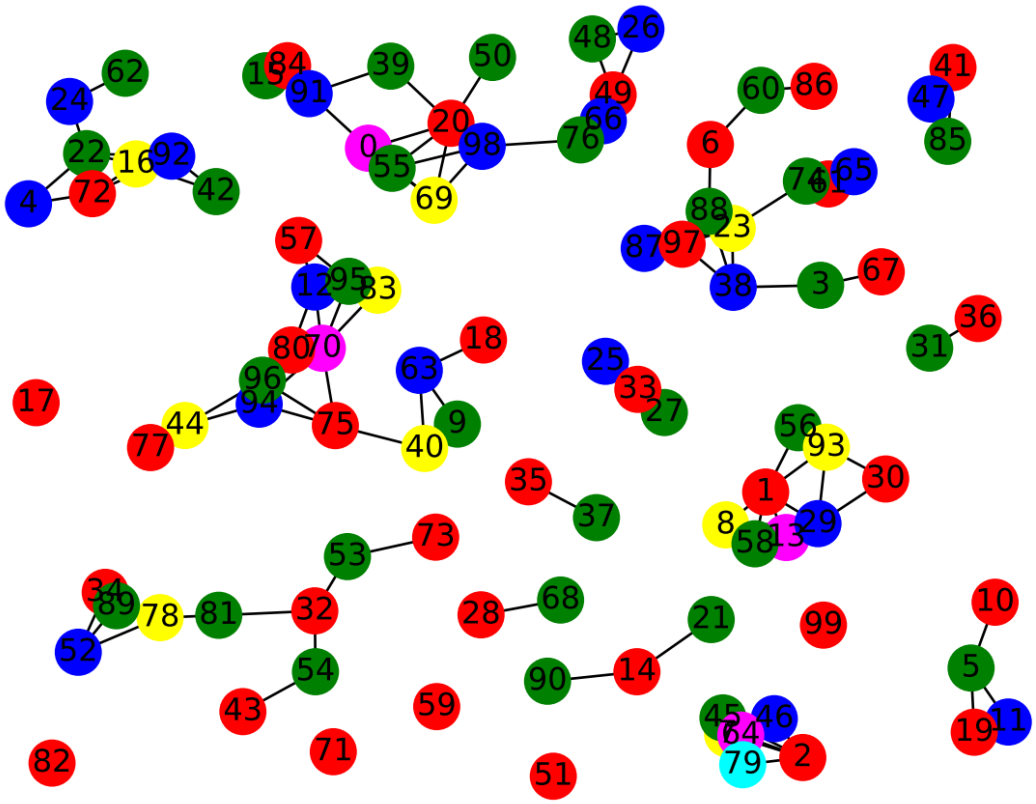


In the second part, our work is done based on the *wifi.mat* and *cords.mat* input files. We used the Numpy library to load the input files. This method is not a conventional method for loading *.mat* files.) We define graph G based on the adjacency matrix between the nodes and then add the nodes. In this section, we set the default color of the nodes to red and define the *color_list* variable with eight colors. Then we define the *greedy_coloring_algorithm* function, which takes a graph and colors as input and performs the coloring operation.

First, in this function, we store the nodes as a list in the nodes variable, and then we shuffled it so that the random ordering that was mentioned in the project definition happens. Now, for the nodes that are in this network, we stored their neighbors in a *dict_neighbors* dictionary and stored the keys whose numbers are the node numbers in the *nodes_neighbors* variable.

Then we defined a $forbidden\_colors$ list, which continues to work for each neighbor if the value of $network.node.data()[1].keys()$ becomes zero, otherwise, the neighbor adds that node in the $forbidden\_color$ variable, which is $forbidden\_color['color']$ becomes the main $forbidden\_color$, which finally appends $forbidden\_color$ in $forbidden\_colors$.

In general, it implements the same dfs algorithm but uses a greedy algorithm. But the distribution of colors is done since if we wanted to randomly choose between colors, each node could have other colors that were not in its neighbors, but here they are fixed in the order of selection. While in the first part, the probability of choosing was only between two colors, the probability of choosing the first color is 0.5, and the probability of choosing the next color is 1 because only one color remains. And it stays that way.


48
26
62
84
39
50
24
91
20
49
60
86
41
22
16
92
0
98
76
66
6
47
85
4
72
42
55
69
74
65
88
23
87
97
38
3
67
57
12
95
33
36
31
80
70
18
25
63
96
33
17
94
27
44
75
9
56
93
77
40
35
30
1
37
29
8
58
13
53
73
34
89
78
81
32
28
68
21
99
10
52
54
14
90
5
43
59
45
46
19
11
71
64
2
82
79
51

```
import scipy
import numpy as np # To input the adjacency matrix as a 2-D array
import networkx as nx # To use graph features
import matplotlib.pyplot as plt # To display a graphical view of the graph
from scipy.sparse import csr_matrix
from tqdm import tqdm
from epidemics_func import PreferentialAttachment, Infection, InfectionV, Recovery, Epidemic,
                           Epidemicw, Vaccinated, GradientOptimizer
import random
```

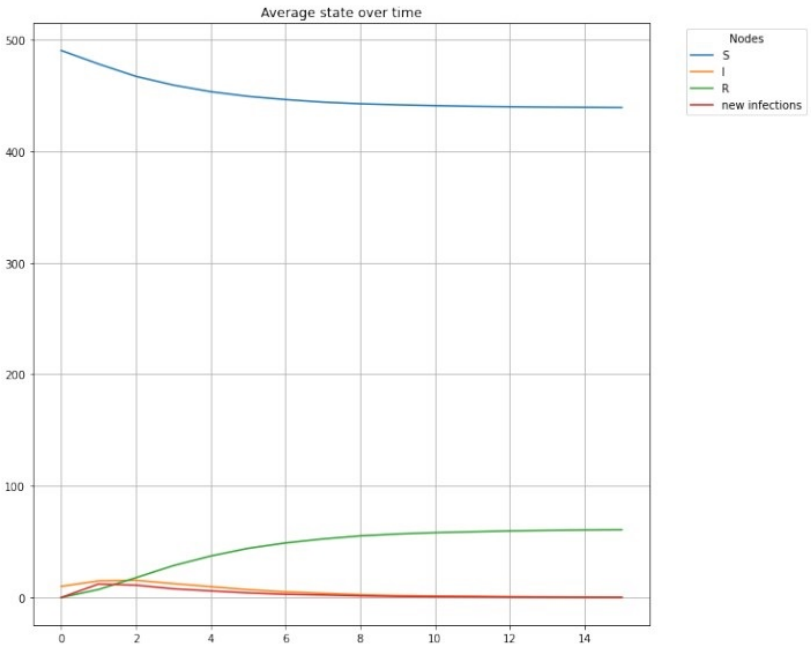
Problem 1

```
#init
n=500
states = []

#we create thr k-regilar matrix by shifting row elements in each row of adjacancy matrix
W=[]
v = np.zeros(n)
v[1] = 1
v[2] = 1
v[-1] = 1
v[-2] = 1
for i in range(n):
    W.append(v)
    v = np.roll(v, 1)

W=np.array(W).astype(int)

#sparse
W = csr_matrix(W)
```

```
states, newinfections = Epidemicw(W, n = 500, init_inf_nodes = 10, beta = 0.3, ro = 0.7, num_simul = 100, steps = 15)
```

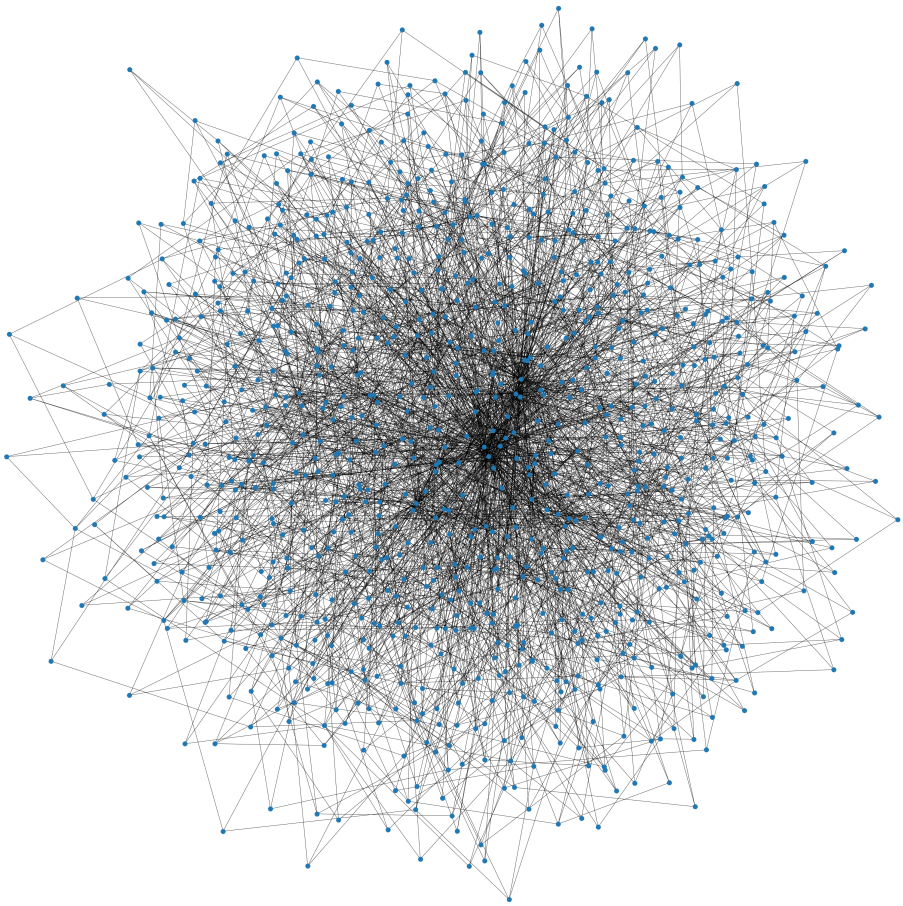
```
100%|██████████| 100/100 [01:03<00:00,  1.58it/s]
```

```
plt.figure(0, figsize=(10, 10))
plt.title("Average Number of nodes in each state over time")
plt.grid()
plt.plot(range(16), np.mean(states, axis = 0)[:, 0], label = "S")
plt.plot(range(16), np.mean(states, axis = 0)[:, 1], label = "I")
plt.plot(range(16), np.mean(states, axis = 0)[:, 2], label = "R")
plt.plot(range(16), np.mean(newinfections, axis = 0), label='New infections')
plt.legend(title='States', bbox_to_anchor=(1.05, 1), loc='upper left')
plt.show()
```



```
G = PreferentialAttachment(6, 1000)
```

```
plt.figure(figsize=(50,50))
nx.draw(G, with_labels = False)
```

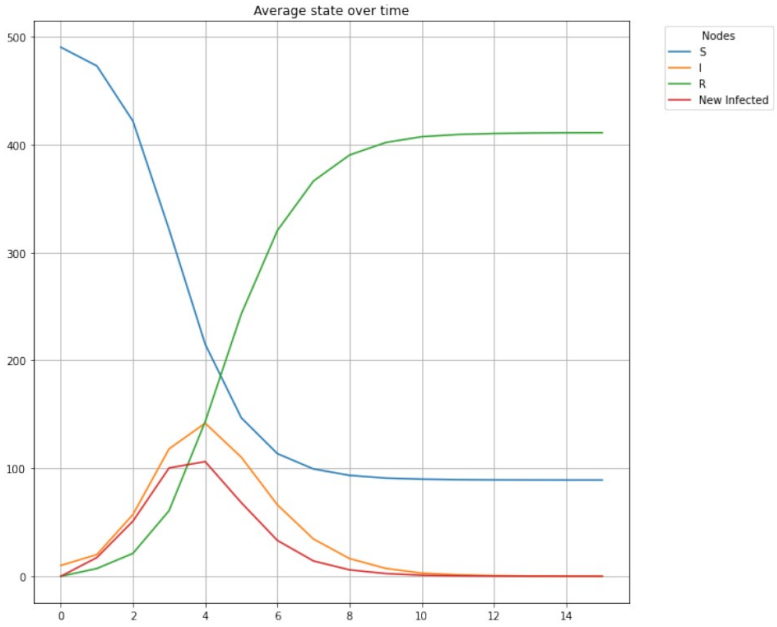


Problem 2

```
#following the algorithm used in previous section we generate our random graph
G = PreferentialAttachment(k = 6, n_nodes = 500)
W = nx.adjacency_matrix(G)
statess, newinfectionss = Epidemicw(W, n = len(G), init_inf_nodes = 10, beta = 0.3, ro = 0.7, num_simul = 100, steps = 15)
```

```
100%|██████████████████████████████████████████████████████████████████████| 100/100 [00:30<00:00,  3.30it/s]
```

```
plt.figure(0, figsize=(10, 10))
plt.title("Average Number of nodes over time")
plt.grid()
plt.plot(range(16), np.mean(statess, axis = 0)[:, 0], label = "S")
plt.plot(range(16), np.mean(statess, axis = 0)[:, 1], label = "I")
plt.plot(range(16), np.mean(statess, axis = 0)[:, 2], label = "R")
plt.plot(range(16), np.mean(newinfectionss, axis = 0), label='New infections')
plt.legend(title='States', loc='upper right')
plt.show()
```



Problem 3

```
#following the algorithm used in previous section we generate our random graph
vaccine_schedule = [0, 5, 15, 25, 35, 45, 55, 60, 60, 60, 60, 60, 60, 60, 60]
statesv, newinfectionsv, mean, rsme = Epidemic(k=6, n = 500, init_inf_nodes = 10, beta = 0.3, ro = 0.7, num_simul = 100, steps =
```

```
100%|██████████| 100/100 [00:30<00:00,  3.28it/s]
```

```
plt.figure(0, figsize=(10, 10))
plt.title("Average state over time")
plt.grid()
plt.plot(np.mean(statesv, axis = 0)[:, 1], label = "S")
plt.plot(np.mean(statesv, axis = 0)[:, 2], label = "I")
plt.plot(np.mean(statesv, axis = 0)[:, 3], label = "R")
plt.plot(np.mean(statesv, axis = 0)[:, 4], label = "V")
plt.plot(np.mean(newinfectionsv, axis = 0), label = "New Infections")
plt.legend(title='Nodes', bbox_to_anchor=(1.05, 1), loc='upper left')
plt.show()
```

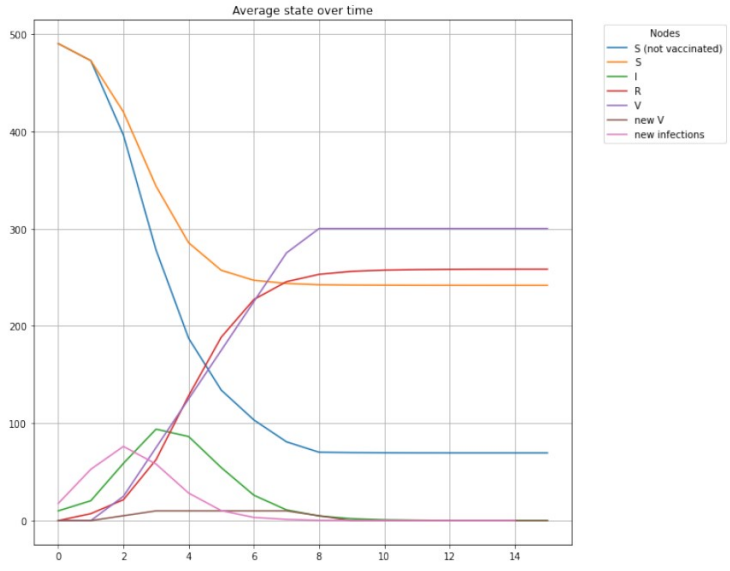
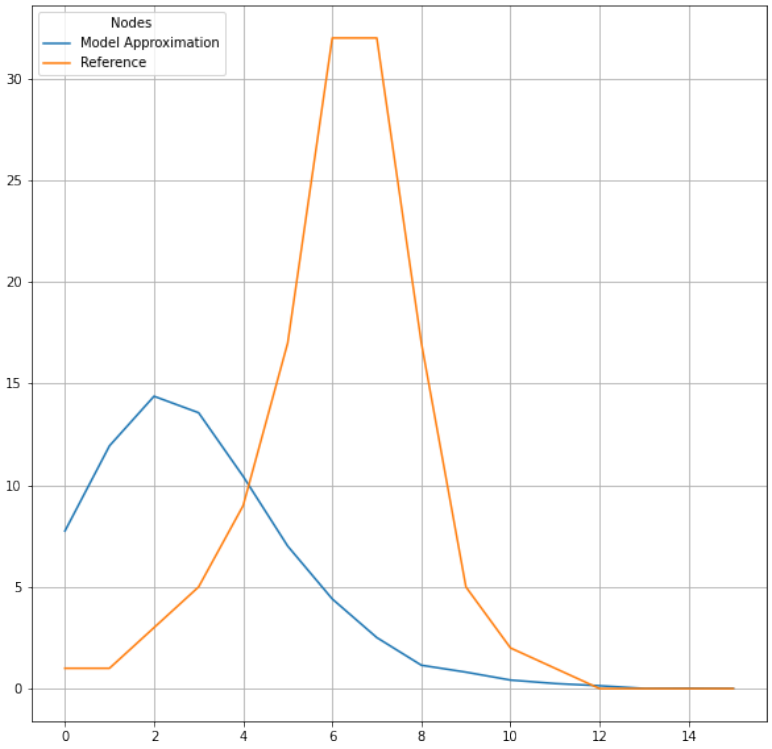


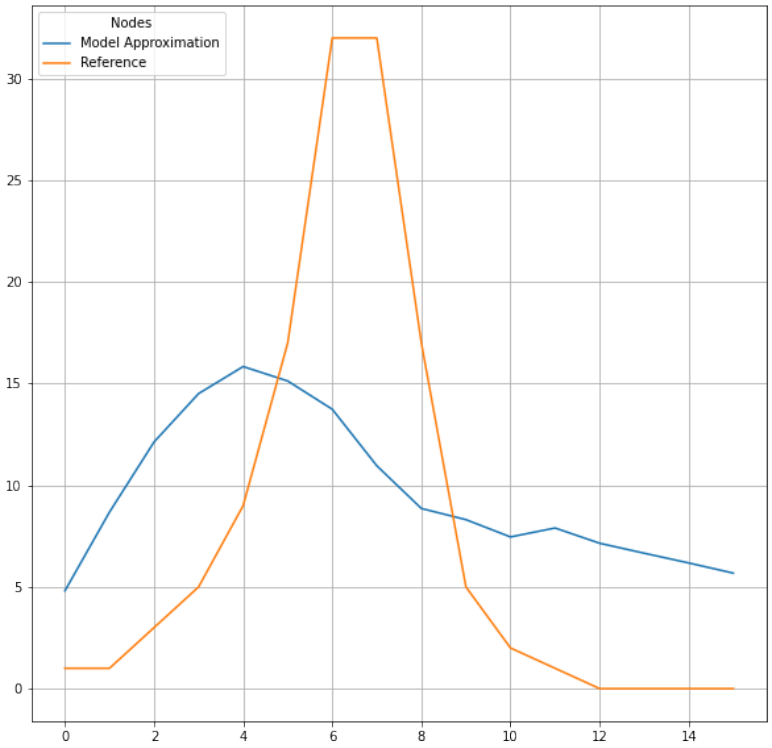
Figure 12: States behaviour

Problem 4

```
best = GradientOptimizer(k=10, beta=0.3, ro=0.6, step_k=2, step_beta=0.1, step_ro=0.1)
```

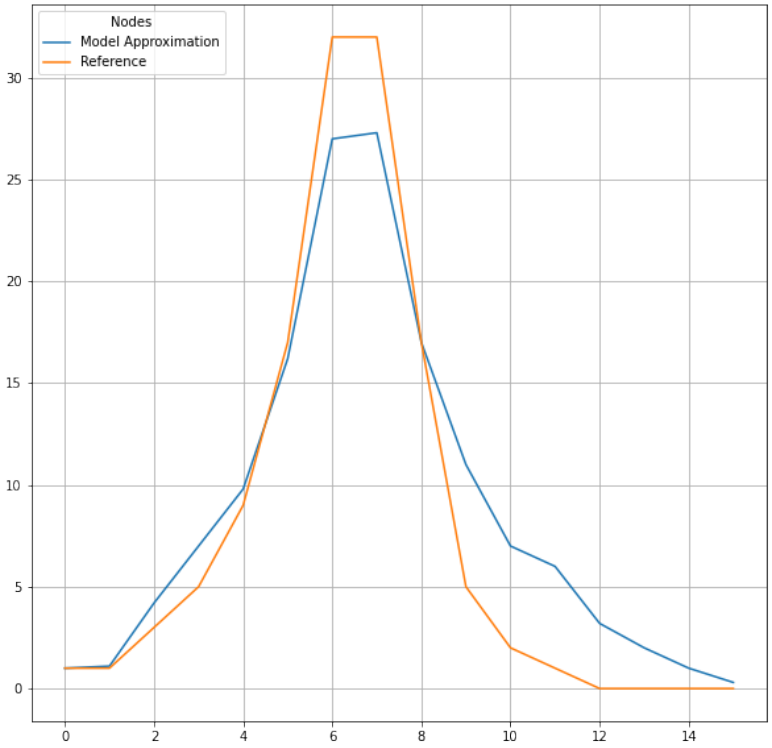



```
best = GradientOptimizer(k=4, beta=0.3, ro=0.6, step_k=2, step_beta=0.1, step_ro=0.1)
```



```
best = GradientOptimizer(k=8, beta=0.3, ro=0.6, step_k=2, step_beta=0.1, step_ro=0.1)
```

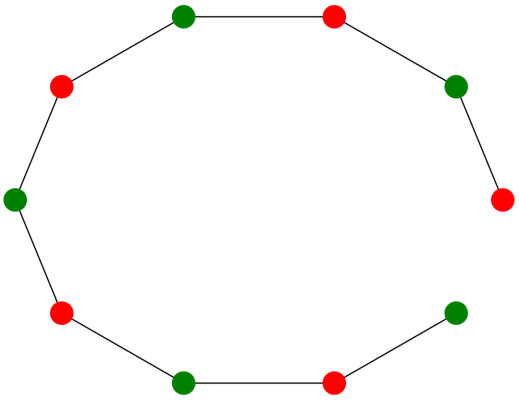
```
  0%|                                                                                | 0/100 [00:00<?, ?it/s]
Starting Parameters with k=8, beta=0.3, ro=0.6
100%|████████████████████████████████████████████████████████████████████████████████| 100/100 [00:34<00:00,  2.93it/s]
100%|████████████████████████████████████████████████████████████████████████████████| 100/100 [00:33<00:00,  2.99it/s]
100%|████████████████████████████████████████████████████████████████████████████████| 100/100 [00:33<00:00,  2.98it/s]
100%|████████████████████████████████████████████████████████████████████████████████| 100/100 [00:34<00:00,  2.91it/s]
100%|████████████████████████████████████████████████████████████████████████████████| 100/100 [00:33<00:00,  2.99it/s]
100%|████████████████████████████████████████████████████████████████████████████████| 100/100 [00:33<00:00,  2.99it/s]
100%|████████████████████████████████████████████████████████████████████████████████| 100/100 [00:34<00:00,  2.93it/s]
100%|████████████████████████████████████████████████████████████████████████████████| 100/100 [00:33<00:00,  2.95it/s]
100%|████████████████████████████████████████████████████████████████████████████████| 100/100 [00:33<00:00,  2.99it/s]
100%|████████████████████████████████████████████████████████████████████████████████| 100/100 [00:33<00:00,  3.01it/s]
100%|████████████████████████████████████████████████████████████████████████████████| 100/100 [00:33<00:00,  2.99it/s]
Best RMSE located : 9.88 at: k=4 beta=0.2 ro=0.8
```



2 Coloring

```python
import numpy as np # To input the adjacency matrix as a 2-D array
import networkx as nx # To use graph features
import matplotlib.pyplot as plt # To display a graphical view of the graph
import random
```

```python
# we used back_tracking DFS to solve this problem
def problem_one():
    G = nx.Graph()  # creating Graph G
    numofnodes = 10  # number of nodes
    adj_matrix = [
        [0, 1, 0, 0, 0, 0, 0, 0, 0, 0],
        [1, 0, 1, 0, 0, 0, 0, 0, 0, 0],
        [0, 1, 0, 1, 0, 0, 0, 0, 0, 0],
        [0, 0, 1, 0, 1, 0, 0, 0, 0, 0],
        [0, 0, 0, 1, 0, 1, 0, 0, 0, 0],
        [0, 0, 0, 0, 1, 0, 1, 0, 0, 0],
        [0, 0, 0, 0, 0, 1, 0, 1, 0, 0],
        [0, 0, 0, 0, 0, 0, 1, 0, 1, 0],
        [0, 0, 0, 0, 0, 0, 0, 1, 0, 1],
        [0, 0, 0, 0, 0, 0, 0, 0, 1, 0],
    ]  # adjacency matrix for line like graph
    adj_matrix = np.array(adj_matrix)
    # print(adj_matrix)
    G = nx.from_numpy_array(adj_matrix)  # creating graph with adjacency matrix
    G.add_nodes_from(G.nodes(), color='black')  # Initializing graph with black-colored nodes
    color_list = ['red', 'green']

    def assignColor(node_):
        safe_color_list = color_list.copy()
        for nd in G.adj[node_].keys():
            if G.nodes[nd]['color'] != 'black':
                safe_color_list.remove(G.nodes[nd]['color'])
        G.nodes[node_]['color'] = safe_color_list[0]
        for nd_ in G.adj[node_].keys():
            if G.nodes[nd_]['color'] == 'black':
                assignColor(nd_)

    node_colors = []
    for nd in G.nodes():
        if G.nodes[nd]['color'] == 'black':
            assignColor(nd)
        node_colors.append(G.nodes[nd]['color'])
    pos = nx.circular_layout(G)  # Set position layout
    nx.draw(G, pos, node_color=node_colors)
    plt.axis('off')  # To prevent showing X-Y axes
    plt.show()  # Displays the graph
    plt.savefig("problem2(a).pdf", format="pdf", bbox_inches="tight")
```



```python
def problem_two():
    data = np.loadtxt('wifi.mat')  # Loading wifi.mat file with numpy
    pos = np.loadtxt('coords.mat')
    data = np.array(data)  # Converting to Numpy array as adjacency matrix
    pos = np.array(pos)
    data = (np.rint(data)).astype(int)  # showing edges between nodes by 1 as an integer number
    G = nx.Graph()  # creating Graph G
    numofnodes = 100  # number of nodes
    G = nx.from_numpy_array(data)  # creating graph with adjacency matrix
    G.add_nodes_from(G.nodes(), color='red')
    color_list = ['red', 'green', 'blue', 'yellow', 'magenta', 'cyan', 'white', 'black']

    def greedy_coloring_algorithm(network, colors):
        nodes = list(network.nodes())
        random.shuffle(nodes)  # step 1 random ordering
        for node in nodes:
            dict_neighbors = dict(network[node])
            # gives names of nodes that are neighbors
            nodes_neighbors = list(dict_neighbors.keys())

            forbidden_colors = []
            for neighbor in nodes_neighbors:

                len(network.nodes.data()[1].keys())
                if len(network.nodes.data()[neighbor].keys()) == 0:
                    # if the neighbor has no color, proceed
                    continue
                else:
                    # if the neighbor has a color,
                    # this color is forbidden

                    forbidden_color = network.nodes.data()[neighbor]
                    forbidden_color = forbidden_color['color']
                    forbidden_colors.append(forbidden_color)
            # assign the first color
            # that is not forbidden
            for color in colors:
                # step 2: start everytime at the top of the colors,
                # so that the smallest number of colors is used
                if color in forbidden_colors:
                    continue
                else:
                    # step 3: color one node at the time
                    network.nodes[node]['color'] = color
                    break

    greedy_coloring_algorithm(G, color_list)
    colors_nodes = [data['color'] for v, data in G.nodes(data=True)]
    nx.draw(G, pos, node_color=colors_nodes, with_labels=True)
    plt.axis('off')  # To prevent showing X-Y axes
    plt.show()  # Displays the graph
    plt.savefig("problem2(b).pdf", format="pdf", bbox_inches="tight")


if __name__ == '__main__':
     problem_one()
   # problem_two()

#%%
```