HMW3

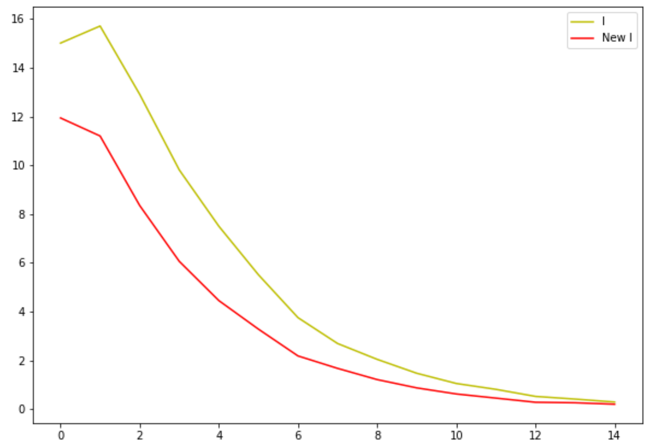
Antonio Dimitris Defonte

Collaborations
The results were compared with:
Eleonora Carletti

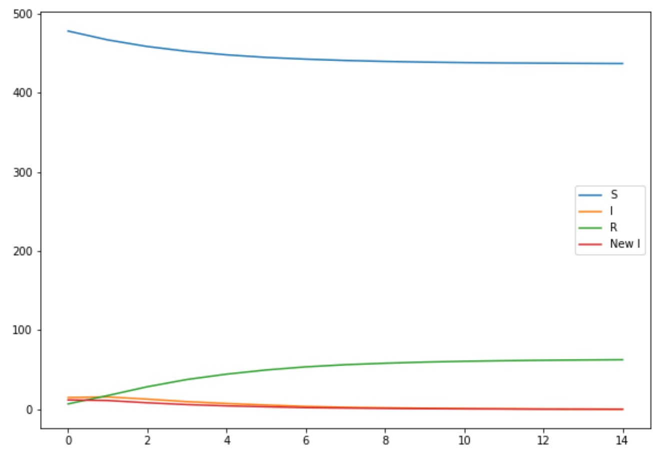
The Influenza H1N1 2009 Pandemic in Sweden

1.1 Epidemic on a known graph

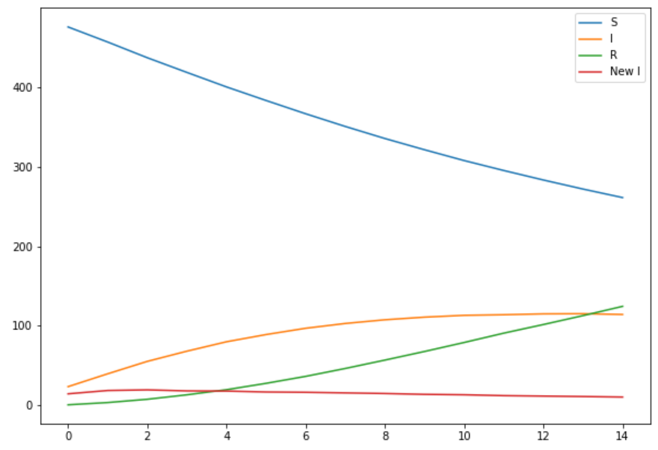
In this first section, it was asked to simulate an epidemic on a symmetric k-regular graph created with the *Networkx* built-in function *circulant_graph*. The *simulate* function takes as input the starting graph and the given parameters and it returns the average number of susceptible, infected, recovered and new infected individuals per week. This was accomplished by tracking with *Numpy* arrays, for each simulation, for each week the total number of nodes that are susceptible, infected, recovered or the new infected. After generating the initial node configuration by sampling a given number of nodes and assigning them the infected status, it is necessary to store both the old and updated node configuration in a *Numpy* array. Each node can have 3 states (SIR). This initial simulation will serve as the basic framework for the entire homework.
The graph must have 500 nodes, with an initial configuration of 10 infected ones. The parameters are as follow: $k = 4$, $\beta = 0.3$, $\rho = 0.7$, the number of simulations is set to 100 with 15 weeks. The parameter β indicates the probability that the infection spreads from an infected node to a susceptible one when they are connected by a link, while ρ is the probability that an infected individual will recover. Both are referred to a unit time step (a week in this case).
In the picture below we can see the trend of the average newly infected individuals each week together with the average number of infected individuals each week.



This other plot depicts the entire SIR epidemics with the different average node statuses per week.



To highlight the importance of the parameters, it is possible to notice how the plot changes by putting $\beta = 0.4$ and $\rho = 0.1$.



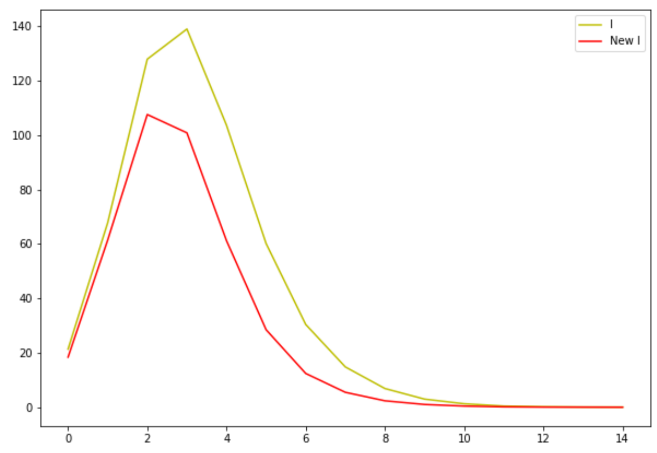
Even though the new infected rate per week has a similar pattern, the number of infected individuals grows much rapidly while that of the susceptible one decreases much faster.

1.2 Generate a random graph

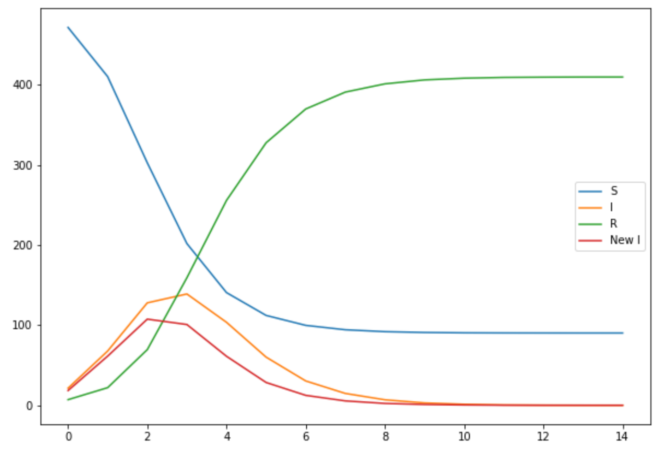
In this section it was asked to generate a random graph according to the *preferential attachment model*. This was accomplished with the *generate_rnd_graph_pa* function which takes as input the average degree k and the maximal number of time steps. At first a complete graph with $k+1$ nodes is generated, then we sample $c = k/2$ nodes according to their distribution and we attach a new node to them with undirected links. This is repeated until we reach the maximal number of time steps (or when we have a graph with a desired number of nodes). The parameter c is computed dynamically in order to adjust to the odd case as specified in the exercise: in a certain iteration t, if t is even we take the *floor* of $k/2$ and the *ceiling* otherwise.

2 Simulate a pandemic without vaccination

The objective of this section was to simulate a pandemic without vaccination on a random graph with *preferential attachment* with an average degree $k = 6$. The other parameters are as in the previous sections.
The first plot shows the average number of newly infected individuals each week together with the average number of infected individuals each week.



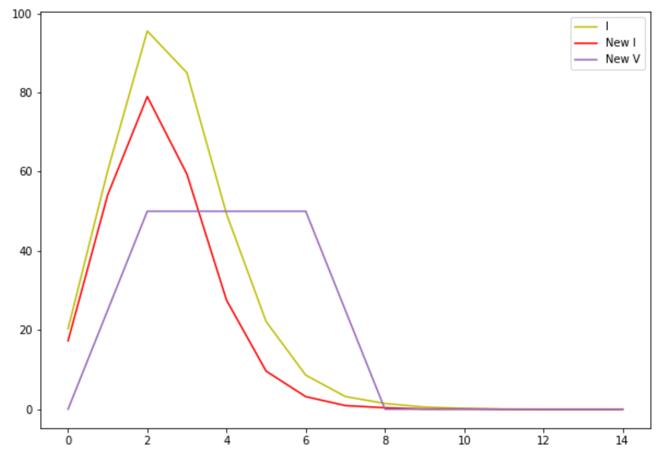
This other plot depicts the entire SIR epidemics with the different average node statuses per week.



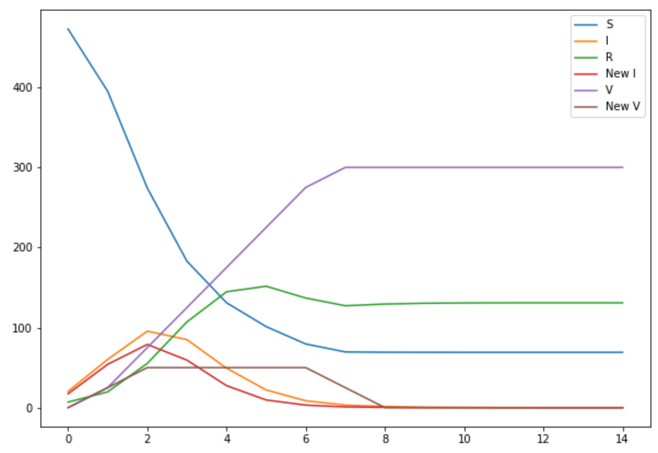
3 Simulate a pandemic with vaccination

In this section it was asked to simulate a pandemic with vaccination. This was accomplished with the *simulate_vaccination* function. Its inner workings are similar to that of the *simulate* function, but in this case we would like to track the vaccinated individuals. The percentage of individuals to vaccinate each week was obtained from the given weekly cumulative percentage of vaccinated people and, on top of the node states of the SIR model (susceptible, infected ad removed), a new node state was added in order to flag an individual as vaccinated in the node configuration (re-

covered people can be vaccinated). I also made possible to have vaccinated individuals in the first week (useful for the next sections) by sampling them at the beginning in the same fashion I also sampled the infected individuals.
In the following figure we can see that, with respect to the previous section, the average number of newly infected individuals per week is lower and the pandemic is dying earlier.



The following plot depicts the different average node statuses per week.



4 The H1N1 pandemic in Sweden 2009

In this section we had to estimate the social structure of the Swedish population and the disease-spread parameters during the H1N1 pandemic. This was accomplished by using 3 new nested functions.
The *search_best_params* function will perform a grid search with a given set of values for β, ρ and k by invoking the previously mentioned *simulate_vaccination* and *generate_rnd_graph_pa* functions and it will return the best combination in terms of RMSE. Each parameter needs to live in a certain

range (β, ρ must be between 0 and 1 since they are probabilities and k should be greater than 1) and this is ensured by the function *necessary_conditions*.
The function *update_parameters* will receive the result of the previous computations and if the RMSE is better it will update the parameters β, ρ and k before invoking again the *search_best_params* function. If the result is worse or if the parameters β, ρ and k are the same in 2 consecutive iterations this function return the current best configuration.
The outermost function *algorithm* changes the $\Delta\beta$, $\Delta\rho$ and Δk and it will return the best overall RMSE with its corresponding parameters.

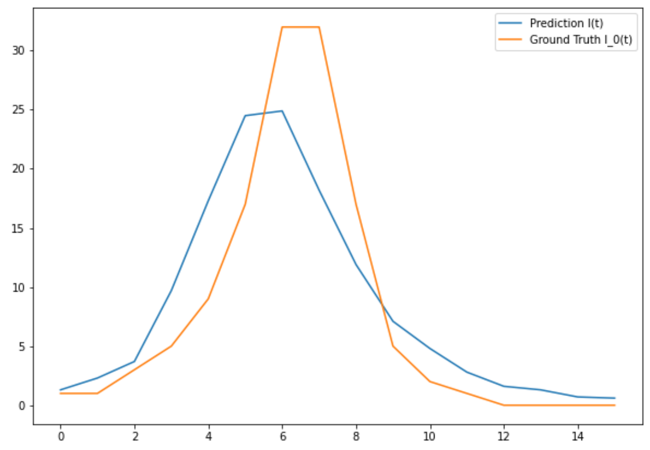
I started with the following parameters:

$$\beta = 0.3 \quad \rho = 0.5 \quad k = 10 \quad \Delta\beta = [0.2, 0.1, 0.05, 0.025] \quad \Delta\rho = [0.2, 0.1, 0.05, 0.025] \quad \Delta k = [3, 2, 1, 1]$$

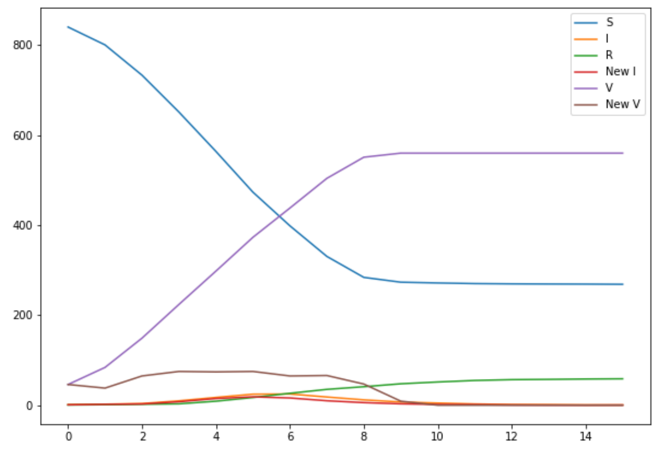
The best results are as follow:

$$\beta = 0.20 \quad \rho = 0.60 \quad k = 8 RMSE = 5.23$$

From the figures below we can see that in this model the infected people increase too fast with respect to the ground truth but overall there is a similar pattern.



The following plot depicts the different average node statuses per week.



5 Challenge (optional)

Graph structure

Networks with *small world* properties can model the small diameter and high clustering that are observed in real-world networks. In an epidemic scenario we would like to model the presence of people in clusters as triangles in the graph, with few long-distance connections. In this sense, this is similar to a scenario in which we have regions or localities where individuals are well connected in the same region, while there are only few long-distance connection between those regions.
This kind of graphs can be constructed in several ways, I tried two different but similar configurations as explained below.

1. As explained in [3], the basic example consists in starting from a *k-regular* undirected graph. If $k \geq 4$ we can ensure a large number of triangles and thus a high clustering coefficient. We can then add l additional undirected links, with l a binomial random variable with parameters $\frac{nk}{2}$ (n is the number of nodes) and $p \in [0, 1]$. The pair of nodes that are connected by the new edges are chosen uniformly at random.

2. In the other example, I started as before form a *k-regular* undirected graph, but instead of adding new edges, I performed a *rewiring* operation. This kind of graph, as explained in [2], is generated form the *Watts-Strogatz* model. For each node i, we take the $k/2$ rightmost edges that are rewired with a certain probability $p \in [0, 1]$. For the rewiring operation, the existing edge (i, j) is substituted with (i, q) where q is a node chosen uniformly at random from all the nodes of the graph. It is important to avoid self loops and edge duplication.

Algorithm

The new algorithm shares a similar structure to that of the previous section, but, as explained below, some building blocks were improved in order to better explore solutions and escape local minima.
Since both the *small world* network models require an additional parameter p, an exhaustive search in the parameter space with a *grid search* would be much time demanding. I implemented a *random search* in which each parameter x is chosen uniformly at random in the interval $[x - \Delta x, x + \Delta x]$

(for the parameter k only integers are considered). This process is repeated for a certain number of times and the parameter configuration with the best RMSE is saved.
Another important contribution is the exploration of worsening solutions. This can be though as a simplified version of the *Simulated Annealing* algorithm as explained in [4]. Instead of updating the parameters when a new best solution is encountered, this new algorithm accepts also worsening solution with a certain criterion, in order to escape local minima.
If the RMSE is not improving, with probability $z \in [0, 1]$ the algorithm will accept the worsening solution, otherwise it exits this current block and continues with the management of the *Deltas* for the different parameters and it will enter a new cycle. z is constructed as follows:

$$z = \exp\left(-\frac{rmse - RMSE}{T}\right)$$

where $rmse$ is the score of the current worsening solution, $RMSE$ is the current best score and T is the current temperature. T is dynamically updated with a so-called *cooling schedule*, which in this case is as follows:

$$T = T0 * \alpha^i$$

with $T0$ the initial temperature (5 i this case), $\alpha \in (0, 1)$ (0.95 in this case) and i the local iteration counter. The cooling schedule is a delicate part of the algorithm, in this case I decided to update the temperature at each iteration (usually it is maintained constant for a certain number of iterations, the *plateau*).
Such formulation returns a lower probability to accept worsening solutions is they are too much distant from the current best one. Another interesting aspect is that as the iterations go by, the temperature decreases and so does z. This will allow at the beginning a higher chance of exploring more solutions and a lower probability to accept worsening solutions as the iteration counter grows.
After all of this, the higher block of the algorithm initializes the parameters with the current best ones, selects the new *Deltas* for the parameters (each time are smaller) and restarts with the various iterations.

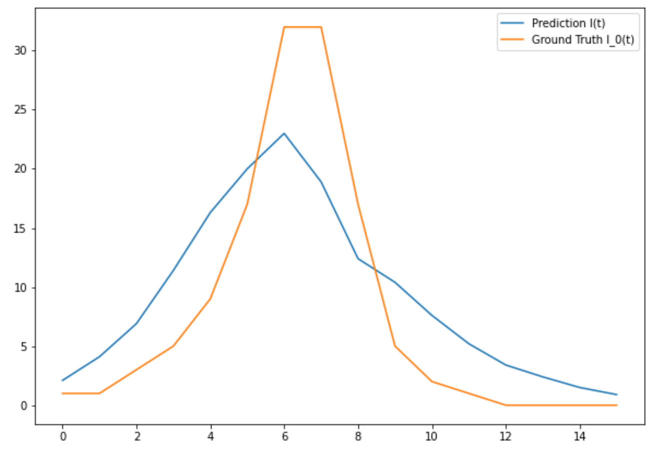
Results

Starting parameters and plot for the model with the basic *small world* graph:

$$\beta = 0.3 \quad \rho = 0.5 \quad k = 10 \quad p = 0.5$$

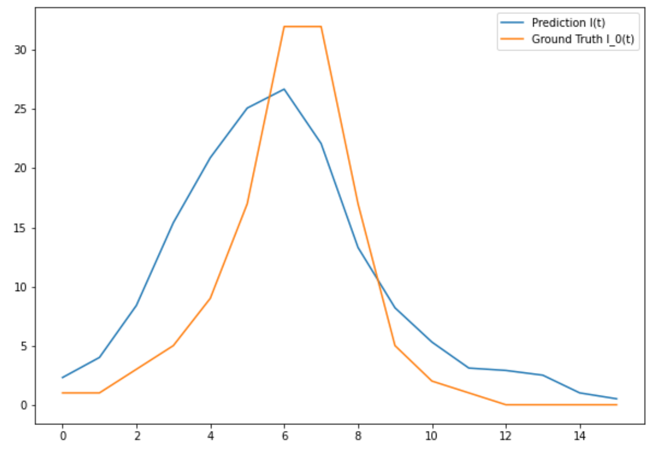
$$\Delta\beta = [0.2, 0.1, 0.05, 0.025] \quad \Delta\rho = [0.2, 0.1, 0.05, 0.025] \quad \Delta k = [3, 2, 1, 1] \quad \Delta p = [0.2, 0.1, 0.05, 0.025]$$

$$RMSE = 5.79$$



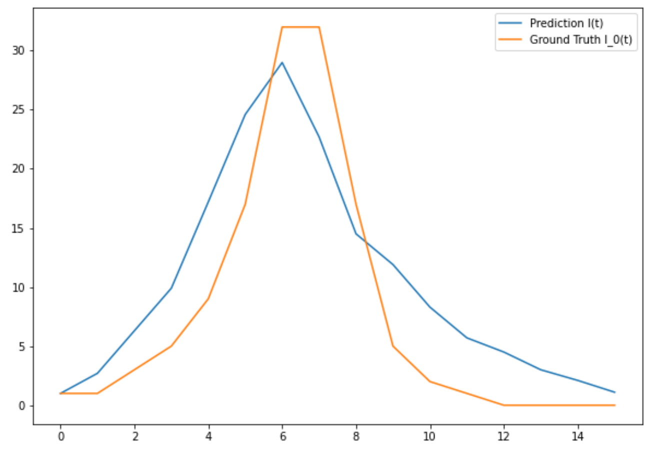
Starting parameters and plot for the *small world* model with rewiring:

$$\beta = 0.2 \quad \rho = 0.5 \quad k = 8 \quad p = 0.5$$
$$\Delta\beta = [0.2, 0.1, 0.05, 0.025] \quad \Delta\rho = [0.2, 0.1, 0.05, 0.025] \quad \Delta k = [3, 2, 1, 1] \quad \Delta p = [0.2, 0.1, 0.05, 0.025]$$
$$RMSE = 5.60$$



Starting parameters for the *preferential attachment* graph:

$$\beta = 0.2 \quad \rho = 0.5 \quad k = 8$$
$$\Delta\beta = [0.2, 0.1, 0.05, 0.025] \quad \Delta\rho = [0.2, 0.1, 0.05, 0.025] \quad \Delta k = [3, 2, 1, 1]$$
$$RMSE = 5.05$$



As we can see from the figures above, the *preferential attachment* model performed better than the *small world* models, reaching an *RMSE* equal to 5.05. It is the only model that was able to bring a small improvement with respect to the previous section.
As we can see from [1] , the demographics of Sweden concentrates the population in urban areas, in particular in few cities and metropolitan areas, while the rest of the territory (with some exceptions) has a low population density. As an improvement to the random graphs I used, it could be employed a network that models the high clustering of the individuals in the south of Sweden and a low link connectivity between individuals in the low population density areas.

References

[1] Sweden. https://en.wikipedia.org/wiki/Sweden#Demographics.

[2] Watts–Strogatz model. https://en.wikipedia.org/wiki/Watts-Strogatz_model.

[3] Fabio Fagnani Giacomo Como. Lecture Notes of Network Dynamics and Learning.

[4] R. Tadei, F. Della Croce, and A. Grosso. *Fondamenti di Ottimizzazione*. Esculapio, 2005.

Code for the assignment

[1]:
```
import networkx as nx
import numpy as np
import scipy as sp
from random import choice, sample
import sys
import math
import matplotlib.pyplot as plt
%matplotlib inline

np.random.seed(0)
```

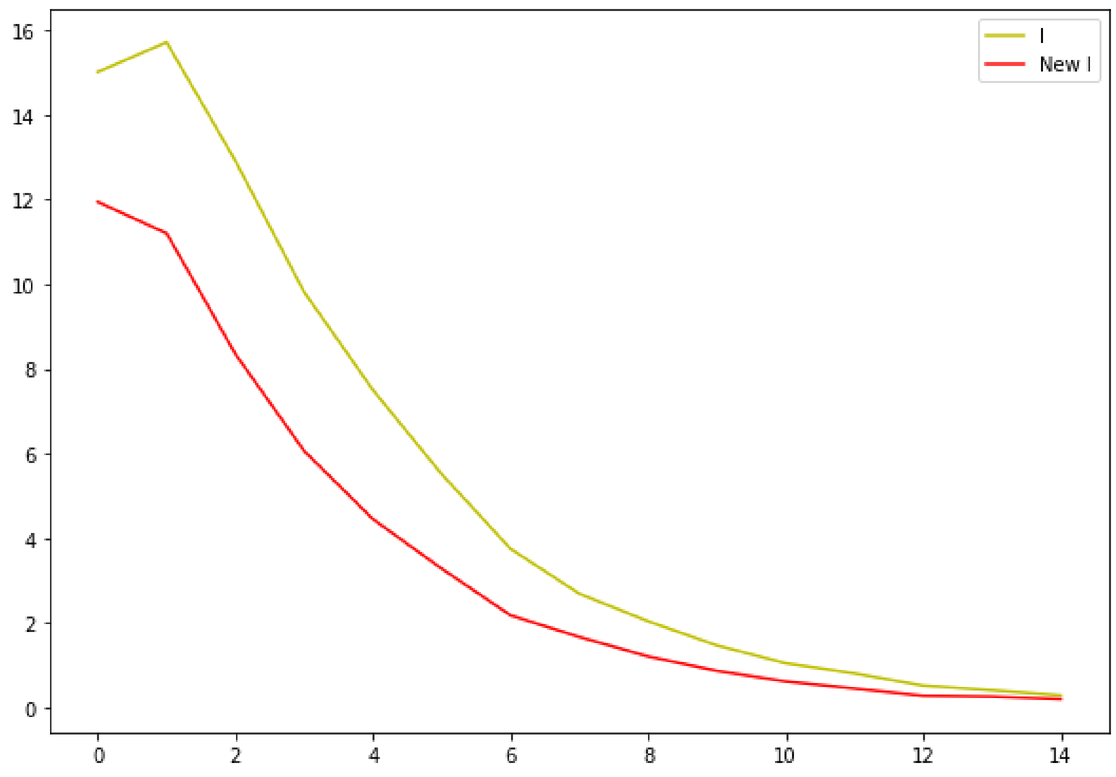
1 Epidemic on a known graph

Problem 1.1

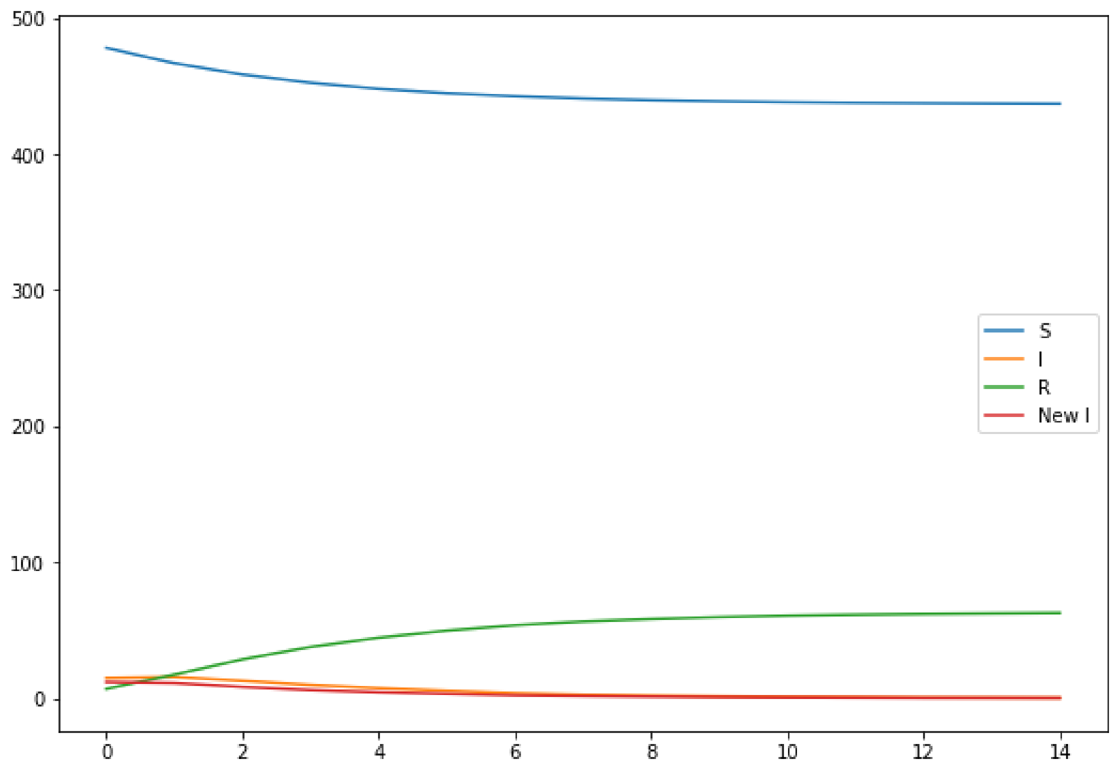
[2]:
```
n_nodes = 500
k_regular = nx.circulant_graph(n_nodes, [1,2])
#nx.draw_circular(k_regular, with_labels=True)
```

[4]:
```
def simulate(graph, n_nodes, initial_infected, n_simulations, n_weeks, beta, ro):

  # Vectors for storing a particular type of node:
  # i^th row corresponds to the i^th simulation,
  # j^th column corresponds to the j^th week
  susceptible_nodes = np.zeros((n_simulations, n_weeks), dtype=int)
  infected_nodes = np.zeros((n_simulations, n_weeks), dtype=int)
  recovered_nodes = np.zeros((n_simulations, n_weeks), dtype=int)
  new_infected_nodes = np.zeros((n_simulations, n_weeks), dtype=int)

  for simulation in range(n_simulations):
    # The node configuration has a column for the old and the updated␣
↪configuration
    nodes_configuration = np.zeros((n_nodes, 2), dtype=int)
    #initial_infected_nodes = sample(graph.nodes(), initial_infected)
    initial_infected_nodes = np.random.choice(graph.nodes,␣
↪size=(initial_infected,) ,replace=False)

    for node in initial_infected_nodes:
      nodes_configuration[node, 0] = 1

    for week in range(n_weeks):

      for node in graph.nodes:
        if nodes_configuration[node, 0] == 0:
          m = 0
          # Count neighbours that are I
          for neighbour in graph.neighbors(node):
            if nodes_configuration[neighbour, 0] == 1:
              m+=1
          if np.random.rand() < (1-beta)**m:
            nodes_configuration[node, 1] = 0
            susceptible_nodes[simulation, week] += 1
          else:
            nodes_configuration[node, 1] = 1
            infected_nodes[simulation, week] += 1
            new_infected_nodes[simulation, week] += 1

        elif nodes_configuration[node, 0] == 1:
          if np.random.rand() < ro:
            nodes_configuration[node, 1] = 2
            recovered_nodes[simulation, week] += 1
          else:
            nodes_configuration[node, 1] = 1
            infected_nodes[simulation, week] += 1

        else:
          nodes_configuration[node, 1] = 2
          recovered_nodes[simulation, week] += 1
      #print(weekly_susceptible_nodes, weekly_infected_nodes,␣
 ↪weekly_recovered_nodes, weekly_new_infected_nodes)

      nodes_configuration[:, 0] = nodes_configuration[:, 1]
      #print(infected_nodes)
      #print(infected_nodes.shape)
      #sys.exit()

  infected_nodes_avg = np.mean(infected_nodes, axis=0)
  susceptible_nodes_avg = np.mean(susceptible_nodes, axis=0)
  recovered_nodes_avg = np.mean(recovered_nodes, axis=0)
  new_infected_nodes_avg = np.mean(new_infected_nodes, axis=0)

  return infected_nodes_avg, susceptible_nodes_avg, recovered_nodes_avg,␣
 ↪new_infected_nodes_avg
```

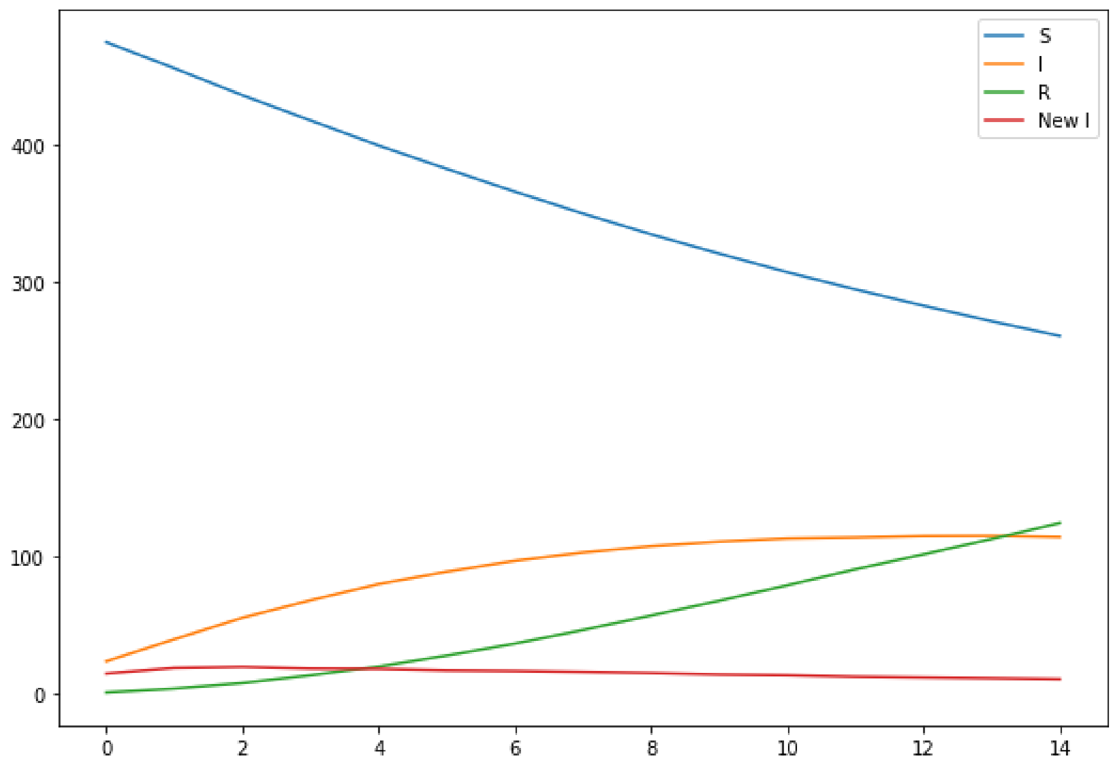
```
[ ]: np.random.seed(0)
     infected_nodes_avg, susceptible_nodes_avg, recovered_nodes_avg,␣
      ↪new_infected_nodes_avg = simulate(k_regular, n_nodes, 10, 100, 15, 0.3, 0.7)
     fig, ax = plt.subplots(figsize=(10,7))
     ax.plot(infected_nodes_avg, 'y', label='I')
     ax.plot(new_infected_nodes_avg, 'r', label='New I')
     ax.legend(loc='best')
     plt.savefig("I_newI_ex1_1.png")
     plt.show()
```



```
[ ]: fig, ax = plt.subplots(figsize=(10,7))
     ax.plot(susceptible_nodes_avg, label='S')
     ax.plot(infected_nodes_avg, label='I')
     ax.plot(recovered_nodes_avg, label='R')
     ax.plot(new_infected_nodes_avg, label='New I')
     ax.legend(loc='best')
     plt.savefig("SIR_ex1_1.png")
     plt.show()
```



```
[ ]: np.random.seed(0)
     infected_nodes_avg, susceptible_nodes_avg, recovered_nodes_avg,␣
      ↪new_infected_nodes_avg = simulate(k_regular, n_nodes, 10, 100, 15, 0.4, 0.1)
     fig, ax = plt.subplots(figsize=(10,7))
     ax.plot(susceptible_nodes_avg, label='S')
     ax.plot(infected_nodes_avg, label='I')
     ax.plot(recovered_nodes_avg, label='R')
     ax.plot(new_infected_nodes_avg, label='New I')
     ax.legend(loc='best')
     plt.savefig("SIR_ex1_1_change.png")
     plt.show()
```



Problem 1.2

```
[6]: def generate_rnd_graph_pa(avg_deg, max_time):

       initial_nodes = avg_deg + 1
       rnd_graph_pa = nx.complete_graph(initial_nodes)
       #nx.draw_circular(rnd_graph_pa, with_labels=True)

       for t in range(initial_nodes, max_time):

         # compute dinamically c in order to adjust for odd case:
         # take floor when t is even
         # take ceil when t is odd
         if t % 2 == 0:
           c = int(math.floor(avg_deg/2))
         else:
           c = int(math.ceil(avg_deg/2))
         #print(c)

         # compute deg distribution at time t-1
         degrees = np.array([d for n, d in rnd_graph_pa.degree()])
         deg_distr = degrees/sum(degrees)
```

```
        neighbors = np.random.choice(np.array(rnd_graph_pa.nodes()), p=deg_distr,␣
  ↪size=c, replace=False)

        # add new node with edges
        node = max(rnd_graph_pa.nodes())+1
        rnd_graph_pa.add_node(node)
        for neigh in neighbors:
            rnd_graph_pa.add_edge(node, neigh)

    return rnd_graph_pa
```

[ ]:
```
# checking if it works
k = 10
rnd_graph_pa = generate_rnd_graph_pa(k, 900)
#nx.draw_circular(rnd_graph_pa, with_labels=True)
degrees = np.array([d for n, d in rnd_graph_pa.degree()])
#print(degrees)
print("Average degree: ", np.mean(degrees))
print("Number of nodes: ", rnd_graph_pa.number_of_nodes())
```

```
Average degree:  10.0
Number of nodes:  900
```

[ ]:
```
# checking if it works also with odd numbers
k = 11
rnd_graph_pa = generate_rnd_graph_pa(k, 900)
#nx.draw_circular(rnd_graph_pa, with_labels=True)
degrees = np.array([d for n, d in rnd_graph_pa.degree()])
#print(degrees)
print("Average degree: ", np.mean(degrees))
print("Number of nodes: ", rnd_graph_pa.number_of_nodes())
```

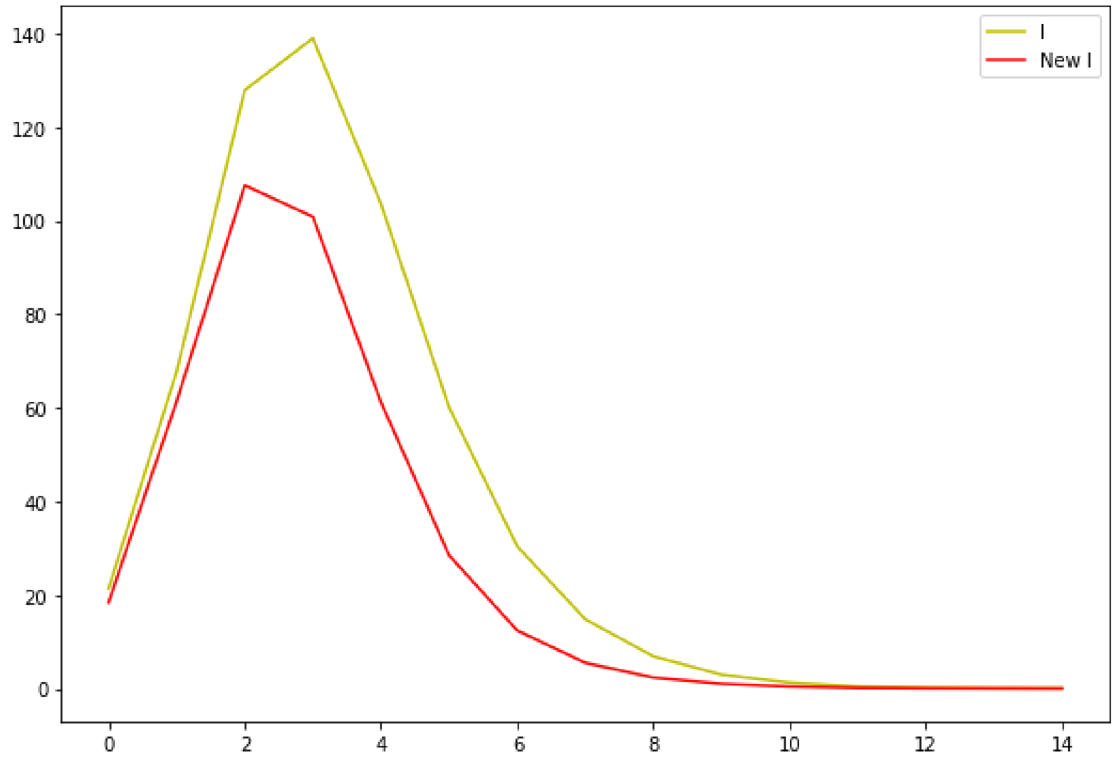
```
Average degree:  11.0
Number of nodes:  900
```

2 Simulate a pandemic without vaccination

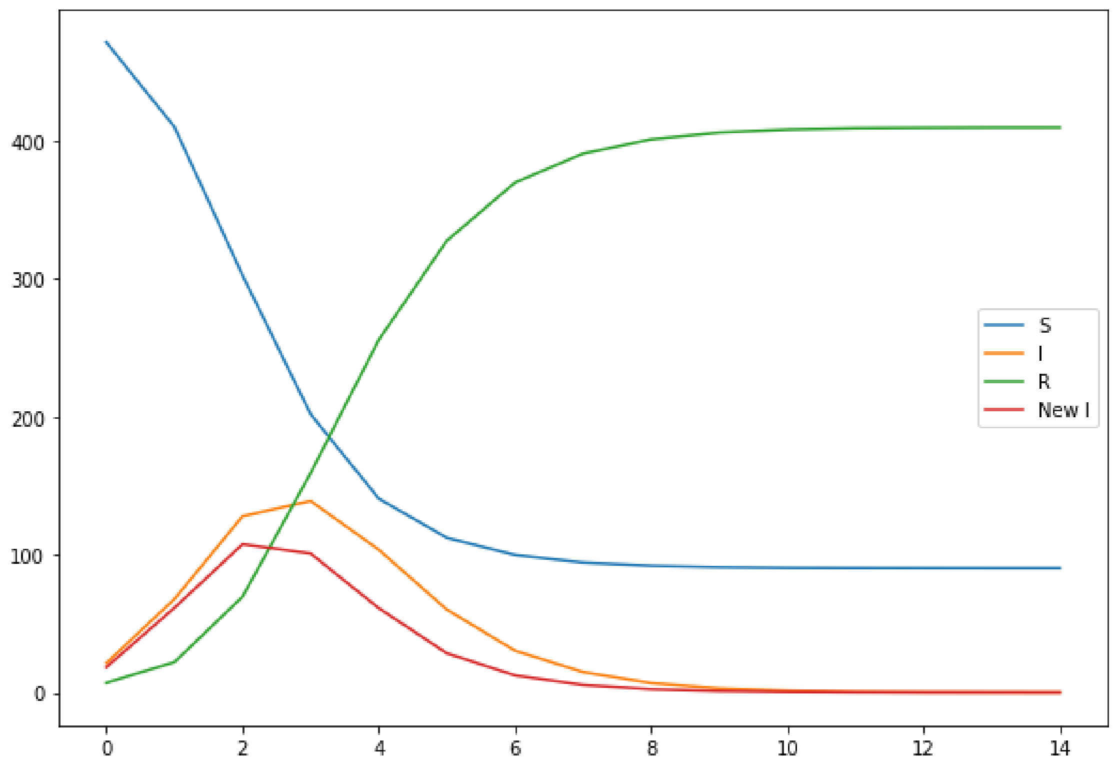
Problem 2

[22]:
```
np.random.seed(0)
k = 6
max_nodes = 500
n_weeks = 15
beta = 0.3
ro = 0.7
initial_infected = 10
n_iterations = 100
```

```
rnd_graph_pa = generate_rnd_graph_pa(k, max_nodes)

infected_nodes_avg, susceptible_nodes_avg, recovered_nodes_avg,␣
 ↪new_infected_nodes_avg = simulate(rnd_graph_pa, max_nodes, initial_infected,␣
 ↪n_iterations, n_weeks, beta, ro)

fig, ax = plt.subplots(figsize=(10,7))
ax.plot(infected_nodes_avg, 'y', label='I')
ax.plot(new_infected_nodes_avg, 'r', label='New I')
ax.legend(loc='best')
plt.savefig("I_newI_ex2.png")
plt.show()
```



[ ]:
```
fig, ax = plt.subplots(figsize=(10,7))
ax.plot(susceptible_nodes_avg, label='S')
ax.plot(infected_nodes_avg, label='I')
ax.plot(recovered_nodes_avg, label='R')
ax.plot(new_infected_nodes_avg, label='New I')
ax.legend(loc='best')
plt.savefig('SIR_ex2.png')
plt.show()
```



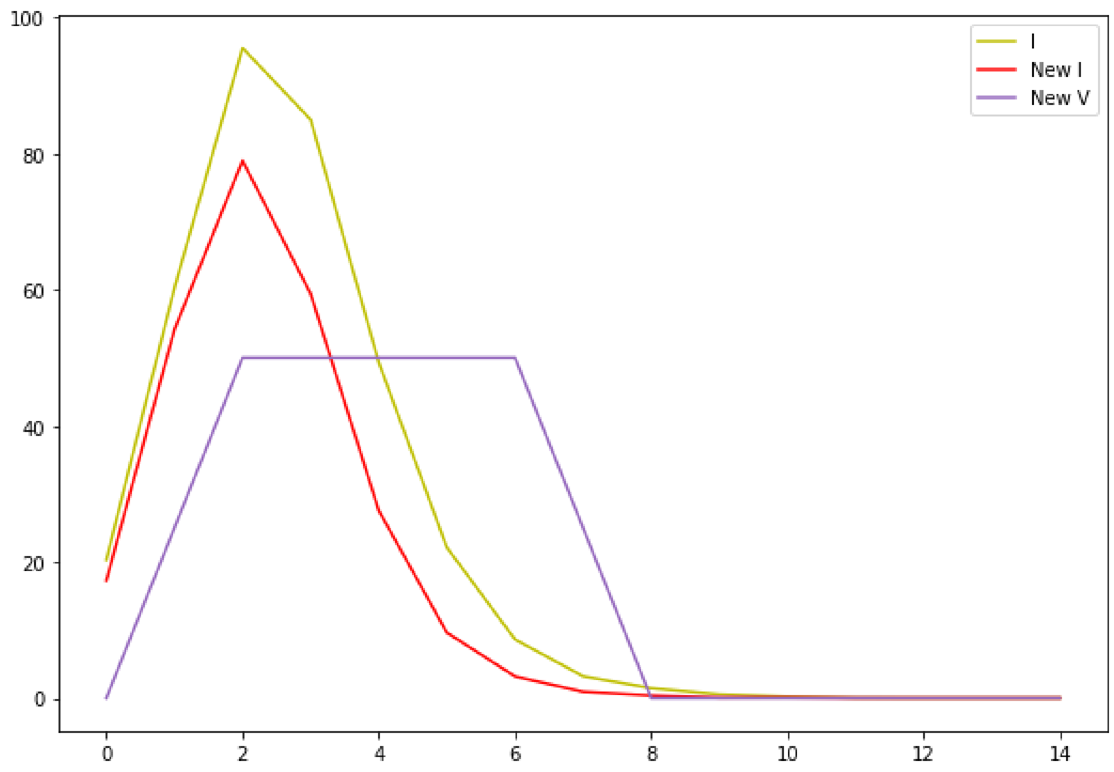
3 Simulate a pandemic with vaccination

Problem 3

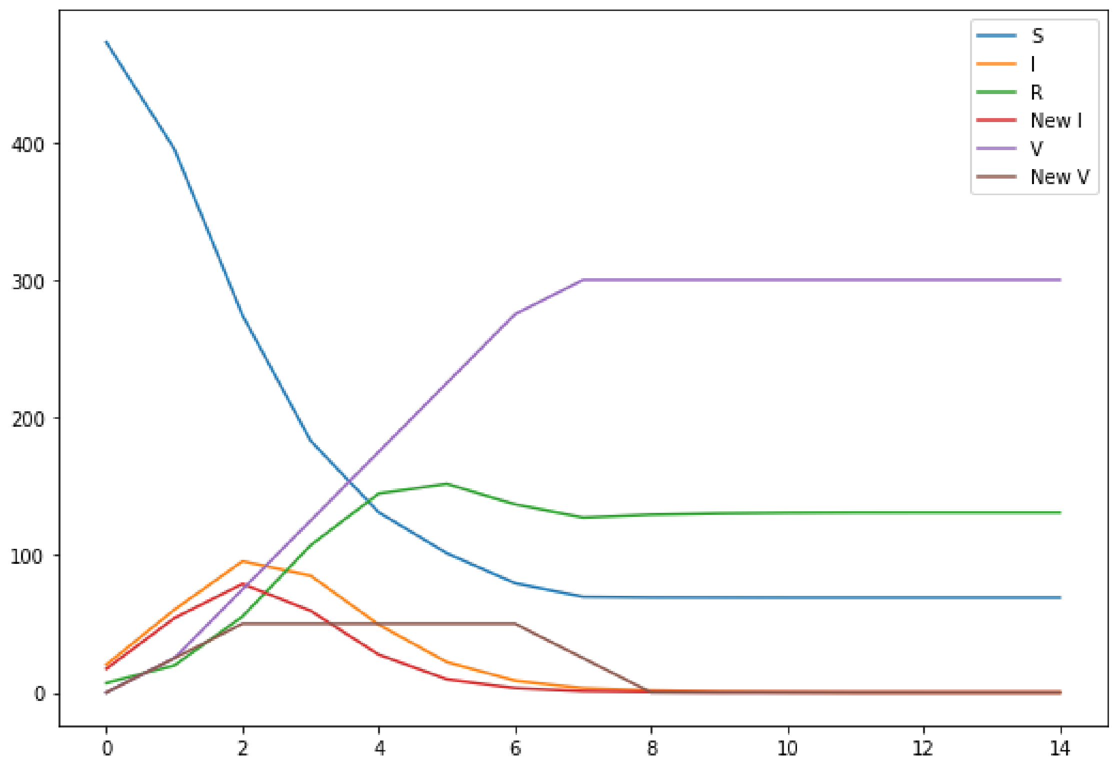
[8]:
```
def simulate_vaccination(graph, n_nodes, initial_infected, n_simulations,␣
 ↪n_weeks, beta, ro, vacc_t):

    # Vectors for storing a particular type of node:
    # i^th row corresponds to the i^th simulation,
    # j^th column corresponds to the j^th week
    susceptible_nodes = np.zeros((n_simulations, n_weeks), dtype=int)
    infected_nodes = np.zeros((n_simulations, n_weeks), dtype=int)
    recovered_nodes = np.zeros((n_simulations, n_weeks), dtype=int)
    new_infected_nodes = np.zeros((n_simulations, n_weeks), dtype=int)

    vacc_perc = n_nodes * np.array(vacc_t) / 100
    #print(vacc_perc)
    vacc_perc = vacc_perc.astype(int)
    #print(vacc_perc)
    weekly_vacc_perc = vacc_perc.copy()
    weekly_vacc_perc[1:] = weekly_vacc_perc[1:]-weekly_vacc_perc[:-1]

    for simulation in range(n_simulations):
```

```
  # The node configuration has a column for the old and the updated␣
↪configuration
  nodes_configuration = np.zeros((n_nodes, 2), dtype=int)
  # modifying this for the next execise in order to have more than 0 vaxx␣
↪nodes at week 0
  initial_infected_vacc_nodes = np.random.choice(graph.nodes,␣
↪size=(initial_infected + weekly_vacc_perc[0], ), replace=False)
  initial_infected_nodes = initial_infected_vacc_nodes[:initial_infected]
  initial_vacc_nodes = initial_infected_vacc_nodes[initial_infected: ]

  for node in initial_infected_nodes:
    nodes_configuration[node, 0] = 1
  for node in initial_vacc_nodes:
    nodes_configuration[node, 0] = 3

  for week in range(n_weeks):

    if weekly_vacc_perc[week] != 0:
      vaccinated_nodes = np.random.choice(np.argwhere(nodes_configuration[:,␣
↪0]!=3).flatten(), size=weekly_vacc_perc[week], replace=False)
      for node in vaccinated_nodes:
        nodes_configuration[node, 0] = 3

    for node in graph.nodes:
      if nodes_configuration[node, 0] == 0:
        m = 0
        # Count neighbours that are I
        for neighbour in graph.neighbors(node):
          if nodes_configuration[neighbour, 0] == 1:
            m+=1
        if np.random.rand() < (1-beta)**m:
          nodes_configuration[node, 1] = 0
          susceptible_nodes[simulation, week] += 1
        else:
          nodes_configuration[node, 1] = 1
          infected_nodes[simulation, week] += 1
          new_infected_nodes[simulation, week] += 1


      elif nodes_configuration[node, 0] == 1:

        if np.random.rand() < ro:
          nodes_configuration[node, 1] = 2
          recovered_nodes[simulation, week] += 1
        else:
          nodes_configuration[node, 1] = 1
          infected_nodes[simulation, week] += 1
        elif nodes_configuration[node, 0] == 2:
          nodes_configuration[node, 1] = 2
          recovered_nodes[simulation, week] += 1

        else:
          nodes_configuration[node, 1] = 3

      nodes_configuration[:, 0] = nodes_configuration[:, 1]

  infected_nodes_avg = np.mean(infected_nodes, axis=0)
  susceptible_nodes_avg = np.mean(susceptible_nodes, axis=0)
  recovered_nodes_avg = np.mean(recovered_nodes, axis=0)
  new_infected_nodes_avg = np.mean(new_infected_nodes, axis=0)


  return infected_nodes_avg, susceptible_nodes_avg, recovered_nodes_avg,␣
 ↪new_infected_nodes_avg, vacc_perc, weekly_vacc_perc
```

```
[ ]: np.random.seed(0)
k = 6
max_nodes = 500
n_weeks = 15
beta = 0.3
ro = 0.7
initial_infected = 10
n_iterations = 100
vacc_t = [0, 5, 15, 25, 35, 45, 55, 60, 60, 60, 60, 60, 60, 60, 60]

rnd_graph_pa = generate_rnd_graph_pa(k, max_nodes)

infected_nodes_avg, susceptible_nodes_avg, recovered_nodes_avg,␣
 ↪new_infected_nodes_avg, vacc_perc, weekly_vacc_perc =␣
 ↪simulate_vaccination(rnd_graph_pa, max_nodes, initial_infected, n_iterations,␣
 ↪n_weeks, beta, ro, vacc_t)

fig, ax = plt.subplots(figsize=(10,7))
ax.plot(infected_nodes_avg, 'y', label='I')
ax.plot(new_infected_nodes_avg, 'r', label='New I')
ax.plot(weekly_vacc_perc, 'tab:purple', label='New V')
ax.legend(loc='best')
plt.savefig("I_newI_vacc_ex3.png")
plt.show()
```



[ ]:
```python
fig, ax = plt.subplots(figsize=(10,7))
ax.plot(susceptible_nodes_avg, label='S')
ax.plot(infected_nodes_avg, label='I')
ax.plot(recovered_nodes_avg, label='R')
ax.plot(new_infected_nodes_avg, label='New I')
ax.plot(vacc_perc, label='V')
ax.plot(weekly_vacc_perc, label='New V')
ax.legend(loc='best')
plt.savefig("SIR_ex3.png")
plt.show()
```



4 The H1N1 pandemic in Sweden 2009

Problem 4

[23]:
```
def RMSE(infected_nodes_avg, infected_ground_truth):
  #print(infected_nodes_avg)
  #print(infected_ground_truth)
  return np.sqrt(1/infected_nodes_avg.shape[0] * np.sum((infected_nodes_avg -␣
 ↪infected_ground_truth)**2))

def necessary_conditions(beta_set, ro_set, k_set):
  for i in range(3):
    # check if the values of beta are between 0 and 1
    if beta_set[i] < 0:
      beta_set[i] = 0.0
    elif beta_set[i] > 1:
      beta_set[i] = 1.0

    # do the same for ro
    if ro_set[i] < 0:
      ro_set[i] = 0.0
    elif ro_set[i] > 1:
      ro_set[i] = 1.0
```

```python
    # avg deg at least 2
    if k_set[i] <=1:
      k_set[i] = 2


def search_best_params(k0, beta0, ro0, d_k, d_beta, d_ro, n_nodes, n_weeks,␣
↪n_iterations, vacc_t, infected_ground_truth):
  # construct the 3 parameters spaces
  beta_set = np.array([beta0 - d_beta, beta0, beta0 + d_beta])
  ro_set = np.array([ro0 - d_ro, ro0, ro0 + d_ro])
  k_set = np.array([k0 - d_k, k0, k0 + d_k])

  necessary_conditions(beta_set, ro_set, k_set)

  # we have 3 parameters with 3 possible values each,
  # we will have to compute 3x3x3 times the result and
  # choose the best one for updating the parameters
  initial_infected = infected_ground_truth[0]
  best_res = {'beta':-1, 'ro':-1, 'k':-1, 'rmse':None}
  best_epidemic = None

  for beta in beta_set:
    for ro in ro_set:
      for k in k_set:
        rnd_graph_pa = generate_rnd_graph_pa(k, n_nodes)
        epidemic = simulate_vaccination(rnd_graph_pa, n_nodes, initial_infected,␣
↪n_iterations, n_weeks, beta, ro, vacc_t)
        rmse = RMSE(epidemic[0], infected_ground_truth)
        #print(beta, ro, k)
        #print(rmse)
        #print()
        #print(epidemic[0], rmse)
        if best_res['rmse'] is None or rmse < best_res['rmse']:
          best_res['beta'] = beta
          best_res['ro'] = ro
          best_res['k'] = k
          best_res['rmse'] = rmse
          best_epidemic = epidemic
        #print()
  #print(best_res)
  return best_res, best_epidemic[0], best_epidemic[1], best_epidemic[2],␣
↪best_epidemic[3], best_epidemic[4], best_epidemic[5]
```

```
[24]: def update_parameters(k0, beta0, ro0, d_k, d_beta, d_ro, n_nodes, n_weeks,␣
       ↪n_iterations, vacc_t, infected_ground_truth):

        stop_sim = False
        global_res = None

        #print(d_k, d_beta, d_ro)

        while stop_sim is False:
          res = search_best_params(k0, beta0, ro0, d_k, d_beta, d_ro, n_nodes,␣
       ↪n_weeks, n_iterations, vacc_t, infected_ground_truth)

          rmse = res[0]['rmse']
          beta = res[0]['beta']
          ro = res[0]['ro']
          k = res[0]['k']

          print(res[0])
          #print(res[1])

          # initial value
          if global_res is None:
            global_res = res

          #print(beta0, ro0, k0, beta, ro, k)
          #print(rmse, global_res[0]['rmse'])

          # if same stop
          if (beta, ro, k) == (beta0, ro0, k0):
            # if same and rmse lower save new value
            if rmse < global_res[0]['rmse']:
              global_res = res
            stop_sim = True

          # if not improving stop and parameters are different
          elif rmse > global_res[0]['rmse']:
            stop_sim = True

          # update initial parameters
          else:
            global_res = res
            k0 = k
            beta0 = beta
            ro0 = ro

        print()
        return global_res
```

```
[25]: def algorithm(k0, beta0, ro0, n_nodes, n_weeks, n_iterations, vacc_t,␣
       ↪infected_ground_truth):
        # Choose a set for each of the delta,
        # in this case 4 values for each one
        delta_beta_set = [0.2, 0.1, 0.05, 0.025]
        delta_ro_set = [0.2, 0.1, 0.05, 0.025]
        delta_k_set = [3, 2, 1, 1]
        best_res = None

        for i in range(len(delta_beta_set)):
          d_k = delta_k_set[i]
          d_beta = delta_beta_set[i]
          d_ro = delta_ro_set[i]

          print(f"k0={k0}, beta0={beta0}, ro0={ro0}, delta_k={d_k},␣
       ↪delta_beta={d_beta}, delta_ro={d_ro}")
          res = update_parameters(k0, beta0, ro0, d_k, d_beta, d_ro, n_nodes, n_weeks,␣
       ↪n_iterations, vacc_t, infected_ground_truth)

          #print(res[1])
          if best_res is None or res[0]['rmse'] < best_res[0]['rmse']:
            best_res = res

          #print(res[0]['rmse'], best_res[0]['rmse'])

        return best_res
```

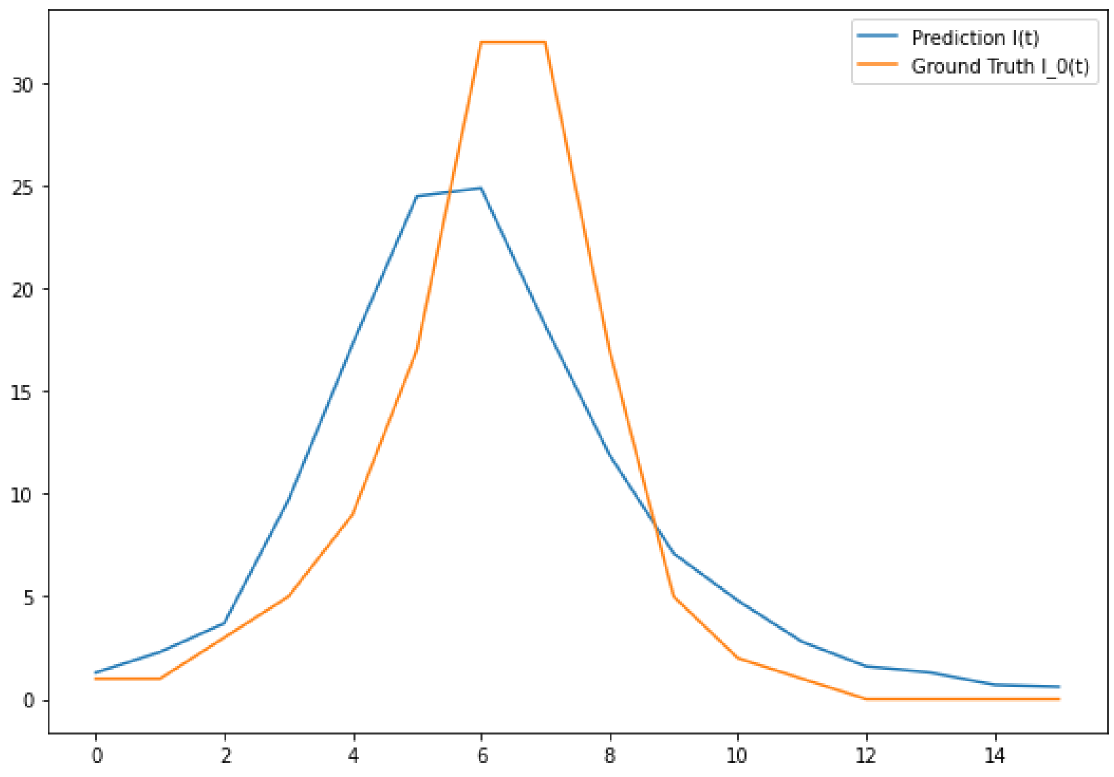
```
[ ]: np.random.seed(0)
     vacc_t = [5, 9, 16, 24, 32, 40, 47, 54, 59, 60, 60, 60, 60, 60, 60, 60]
     infected_ground_truth = [1, 1, 3, 5, 9, 17, 32, 32, 17, 5, 2, 1, 0, 0, 0, 0]
     n_nodes = 934
     n_weeks = len(vacc_t)
     n_iterations = 10
     k0 = 10
     beta0 = 0.3
     ro0 = 0.5

     best_res, best_infected_nodes, best_susceptible_nodes, best_recovered_nodes,␣
      ↪best_new_infected_nodes, best_vacc_perc, best_weekly_vacc_perc = algorithm(k0,␣
      ↪beta0, ro0, n_nodes, n_weeks, n_iterations, vacc_t, infected_ground_truth)
```

```
k0=10, beta0=0.3, ro0=0.5, delta_k=3, delta_beta=0.2, delta_ro=0.2
{'beta': 0.09999999999999998, 'ro': 0.5, 'k': 13, 'rmse': 6.9589869952457875}
{'beta': 0.09999999999999998, 'ro': 0.7, 'k': 16, 'rmse': 5.859500831982192}
{'beta': 0.09999999999999998, 'ro': 0.49999999999999994, 'k': 13, 'rmse':
6.0903612372337985}
```

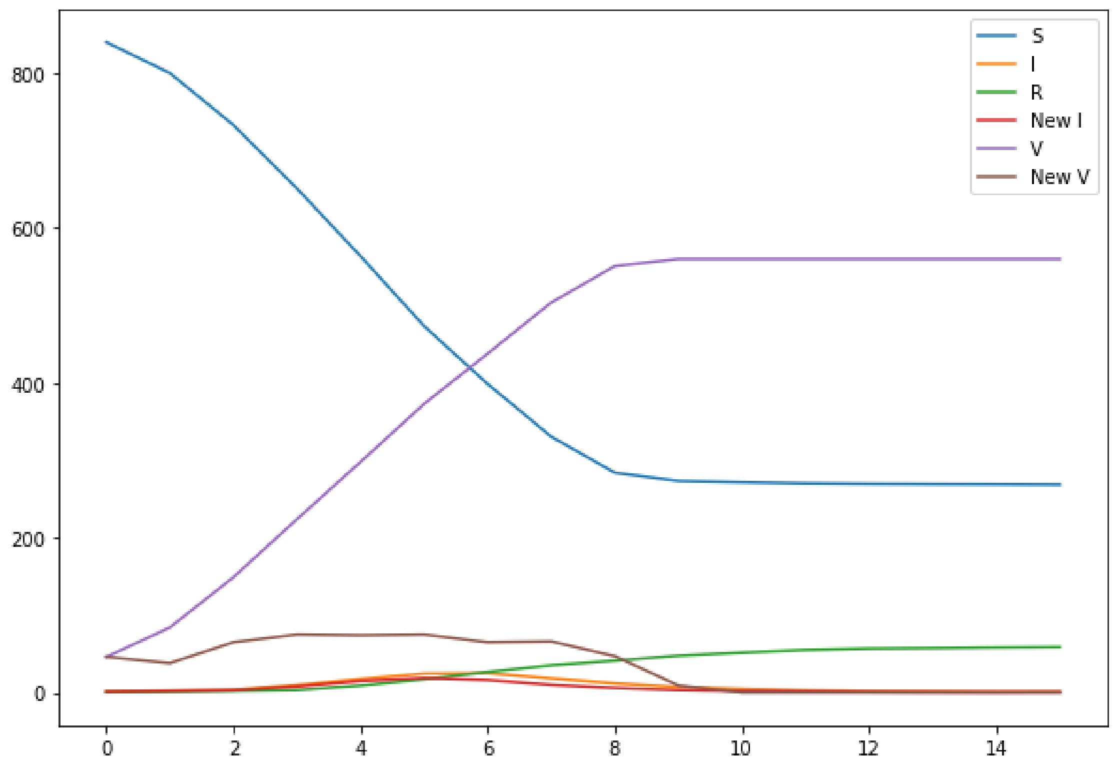
```
k0=10, beta0=0.3, ro0=0.5, delta_k=2, delta_beta=0.1, delta_ro=0.1
{'beta': 0.19999999999999998, 'ro': 0.6, 'k': 8, 'rmse': 5.225837253493454}
{'beta': 0.19999999999999998, 'ro': 0.5, 'k': 6, 'rmse': 7.152490824880519}

k0=10, beta0=0.3, ro0=0.5, delta_k=1, delta_beta=0.05, delta_ro=0.05
{'beta': 0.25, 'ro': 0.5, 'k': 10, 'rmse': 21.196815798605225}
{'beta': 0.2, 'ro': 0.55, 'k': 10, 'rmse': 10.554264540933206}
{'beta': 0.15000000000000002, 'ro': 0.5, 'k': 9, 'rmse': 6.4331660168225095}
{'beta': 0.15000000000000002, 'ro': 0.5, 'k': 8, 'rmse': 6.644405541506328}

k0=10, beta0=0.3, ro0=0.5, delta_k=1, delta_beta=0.025, delta_ro=0.025
{'beta': 0.27499999999999997, 'ro': 0.5, 'k': 9, 'rmse': 32.97848730612124}
{'beta': 0.27499999999999997, 'ro': 0.525, 'k': 8, 'rmse': 17.069710015111564}
{'beta': 0.27499999999999997, 'ro': 0.55, 'k': 7, 'rmse': 6.497739991720198}
{'beta': 0.27499999999999997, 'ro': 0.5750000000000001, 'k': 8, 'rmse':
6.288083968904996}
{'beta': 0.24999999999999997, 'ro': 0.6000000000000001, 'k': 7, 'rmse':
7.748951541982954}
```

[ ]:
```python
print(f"Best rmse: {best_res}")
fig, ax = plt.subplots(figsize=(10,7))
ax.plot(best_infected_nodes, label='Prediction I(t)')
ax.plot(infected_ground_truth, label='Ground Truth I_0(t)')
ax.legend(loc='best')
plt.savefig('pred_vs_GT.png')
plt.show()
```

```
Best rmse: {'beta': 0.19999999999999998, 'ro': 0.6, 'k': 8, 'rmse':
5.225837253493454}
```



```
[ ]: fig, ax = plt.subplots(figsize=(10,7))
     ax.plot(best_susceptible_nodes, label='S')
     ax.plot(best_infected_nodes, label='I')
     ax.plot(best_recovered_nodes, label='R')
     ax.plot(best_new_infected_nodes, label='New I')
     ax.plot(best_vacc_perc, label='V')
     ax.plot(best_weekly_vacc_perc, label='New V')
     ax.legend(loc='best')
     plt.savefig('SIR_ex4.png')
     plt.show()
```



5 Challenge (optional)

[10]:
```
def small_world_binomial(n_nodes, k, p):
  # define the initial k_regular graph
  k_neighbours = int(k/2)
  graph = nx.circulant_graph(n_nodes, list(range(1, k_neighbours+1)))
  l = np.random.binomial(n_nodes*k/2, p, 1)
  for i in range(int(l)):
    res = np.random.choice(np.array(graph.nodes()), size=2, replace=False)
    a = res[0]
    b = res[1]
    new_edge = (a, b)
    while (new_edge in graph.edges()):
      res = np.random.choice(np.array(graph.nodes()), size=2, replace=False)
      a = res[0]
      b = res[1]
      new_edge = (a, b)
    graph.add_edge(a, b)

  return graph
```

[11]:
```
def small_world_rewiring(n_nodes, k, p):
  # define the initial k_regular graph
  k_neighbours = int(k/2)
  graph = nx.circulant_graph(n_nodes, list(range(1, k_neighbours+1)))

  # now iterate through the existing edges
  for edge in graph.edges():
    # with probability p rewire the current edge
    if np.random.rand() < p:
      graph.remove_edge(edge[0], edge[1])
      # to avoid self loops remove current tail of the edge
      other_nodes = list(graph.nodes())
      other_nodes.remove(edge[0])
      # choose uniformly random the new head
      new_end_node = int(np.random.choice(other_nodes, size=1 ,replace=False))
      new_edge = (edge[0], new_end_node)
      #print(edge, new_edge)
      # to avoid adding existing edges, check if the new edge already exists
      while (new_edge in graph.edges()):
        # in this case, choose again a new head
        new_end_node = int(np.random.choice(other_nodes, size=1 ,replace=False))
        new_edge = (edge[0], new_end_node)
        #print("Correction ", new_edge)
      graph.add_edge(new_edge[0], new_edge[1])
      #print()

  return graph
```

[12]:
```
class meta_heuristics:
  def __init__(self, graph_type, k0, beta0, ro0, p0, d_k_set, d_beta_set,␣
↪d_ro_set, d_p_set, n_searches, n_nodes, n_weeks, n_iterations, vacc_t,␣
↪infected_ground_truth, temperature, alpha):
    self.k0 = k0
    self.beta0 = beta0
    self.ro0 = ro0
    self.p0 = p0
    self.delta_k_set = d_k_set
    self.delta_beta_set = d_beta_set
    self.delta_ro_set = d_ro_set
    self.delta_p_set = d_p_set
    self.n_searches = n_searches
    self.n_nodes = n_nodes
    self.n_weeks = n_weeks
    self.n_iterations = n_iterations
    self.vacc_t = vacc_t
    self.infected_ground_truth = infected_ground_truth
    self.temperature = temperature
    self.alpha = alpha
    self.graph_type = graph_type


  def __RMSE(self, infected_nodes_avg):
    return np.sqrt(1/infected_nodes_avg.shape[0] * np.sum((infected_nodes_avg -␣
↪self.infected_ground_truth)**2))

  def __necessary_conditions(self, beta, ro, k, p):
    # check if the values of beta are between 0 and 1
    if beta < 0:
      beta = 0.0
    elif beta > 1:
      beta = 1.0

    # do the same for ro
    if ro < 0:
      ro = 0.0
    elif ro > 1:
      ro = 1.0

    # do the same for p
    if p is not None:
      if p < 0:
        p = 0.0
      elif p > 1:
        p = 1.0

    # avg deg at least 2
    if k <= 1:
      k = 2

    return beta, ro, p, k


  def __search_best_params(self, d_k, d_p, d_beta, d_ro):

    initial_infected = self.infected_ground_truth[0]
    best_res = {'beta':-1, 'ro':-1, 'k':-1, 'p':-1, 'rmse':None}
    best_epidemic = None

    for param_set in range(self.n_searches):
      beta = np.random.uniform(self.beta0 - d_beta, self.beta0 + d_beta)
      ro = np.random.uniform(self.ro0 - d_ro, self.ro0 + d_ro)
      k = np.random.randint(self.k0 - d_k, self.k0 + d_k)
      if p0 is not None:
        p = np.random.uniform(self.p0 - d_p, self.p0 + d_p)
      else:
        p = None
      beta, ro, p, k = self.__necessary_conditions(beta, ro, k, p)

      #print(beta, ro, k, p)

      if self.graph_type == "small_world_rewiring":
        rnd_graph = small_world_rewiring(self.n_nodes, k, p)
      elif self.graph_type == "small_world_binomial":
        rnd_graph = small_world_binomial(n_nodes, k, p)
      elif self.graph_type == "preferential_attachment":
        rnd_graph = generate_rnd_graph_pa(k, self.n_nodes)
      else:
        print(f"Graph {self.graph_type} not impemented")
        sys.exit()

      #print(len(rnd_graph_pa.nodes()))
      epidemic = simulate_vaccination(rnd_graph, self.n_nodes, initial_infected,␣
↪self.n_iterations, self.n_weeks, beta, ro, self.vacc_t)
      rmse = self.__RMSE(epidemic[0])
      #print(beta, ro, k)
      #print(rmse)
      #print()
      #print(epidemic[0], rmse)
      if best_res['rmse'] is None or rmse < best_res['rmse']:
        best_res['beta'] = beta
        best_res['ro'] = ro
        best_res['k'] = k
        best_res['p'] = p
        best_res['rmse'] = rmse
        best_epidemic = epidemic
      #print()

    #print(best_res)
    return best_res, best_epidemic[0], best_epidemic[1], best_epidemic[2],␣
↪best_epidemic[3], best_epidemic[4], best_epidemic[5]


  def __update_parameters(self, d_k, d_p, d_beta, d_ro):
    stop_sim = False
    global_res = None
    iteration_counter = 0

    #print(d_k, d_beta, d_ro)

    while stop_sim is False:
```

```
    res = self.__search_best_params(d_k, d_p, d_beta, d_ro)

    rmse = res[0]['rmse']
    beta = res[0]['beta']
    ro = res[0]['ro']
    k = res[0]['k']
    p = res[0]['p']

    print(res[0])
    #print(res[1])

    # initial value
    if global_res is None:
      global_res = res

    #print(beta0, ro0, k0, beta, ro, k)
    #print(rmse, global_res[0]['rmse'])

    #print(iteration_counter)

    # if same stop
    if (beta, ro, k) == (self.beta0, self.ro0, self.k0):
      # if same and rmse lower save new value
      if rmse < global_res[0]['rmse']:
        global_res = res
      stop_sim = True

    # if not improving stop and parameters are different
    elif rmse > global_res[0]['rmse']:
      self.temperature = self.temperature * (self.alpha ** iteration_counter)
      prob = np.exp(-(rmse - global_res[0]['rmse'])/self.temperature)
      #print(rmse, global_res[0]['rmse'])
      #print(prob)
      if np.random.rand() < prob:
        print("worsening solution")
        self.k0 = k
        self.beta0 = beta
        self.ro0 = ro
        self.p = p
      else:
        print("exit")
        stop_sim = True

    # update initial parameters
    else:
      global_res = res
      self.k0 = k
```

```
        self.beta0 = beta
        self.ro0 = ro
        self.p = p

      iteration_counter += 1

    print()
    return global_res


  def algorithm(self):
    best_res = None
    for i in range(len(self.delta_beta_set)):
      d_k = self.delta_k_set[i]
      if delta_p_set is not None:
        d_p = self.delta_p_set[i]
      else:
        d_p = None
      d_beta = self.delta_beta_set[i]
      d_ro = self.delta_ro_set[i]

      print(f"k0={self.k0}, beta0={self.beta0}, ro0={self.ro0}, p0={self.p0}␣
↪delta_k={d_k}, delta_beta={d_beta}, delta_ro={d_ro}, delta_p={d_p}")
      res = self.__update_parameters(d_k, d_p, d_beta, d_ro)

      #print(res[1])
      if best_res is None or res[0]['rmse'] < best_res[0]['rmse']:
        best_res = res
        self.beta0 = res[0]['beta']
        self.fk0 = res[0]['k']
        self.ro0 = res[0]['ro']

      #print(res[0]['rmse'], best_res[0]['rmse'])

    return best_res
```

[ ]:

```
np.random.seed(0)
delta_beta_set = [0.2, 0.1, 0.05, 0.025]
delta_ro_set = [0.2, 0.1, 0.05, 0.025]
delta_p_set = [0.2, 0.1, 0.05, 0.025]
delta_k_set = [3, 2, 1, 1]
vacc_t = [5, 9, 16, 24, 32, 40, 47, 54, 59, 60, 60, 60, 60, 60, 60, 60]
infected_ground_truth = [1, 1, 3, 5, 9, 17, 32, 32, 17, 5, 2, 1, 0, 0, 0, 0]
n_nodes = 934
n_weeks = len(vacc_t)
n_iterations = 10
```

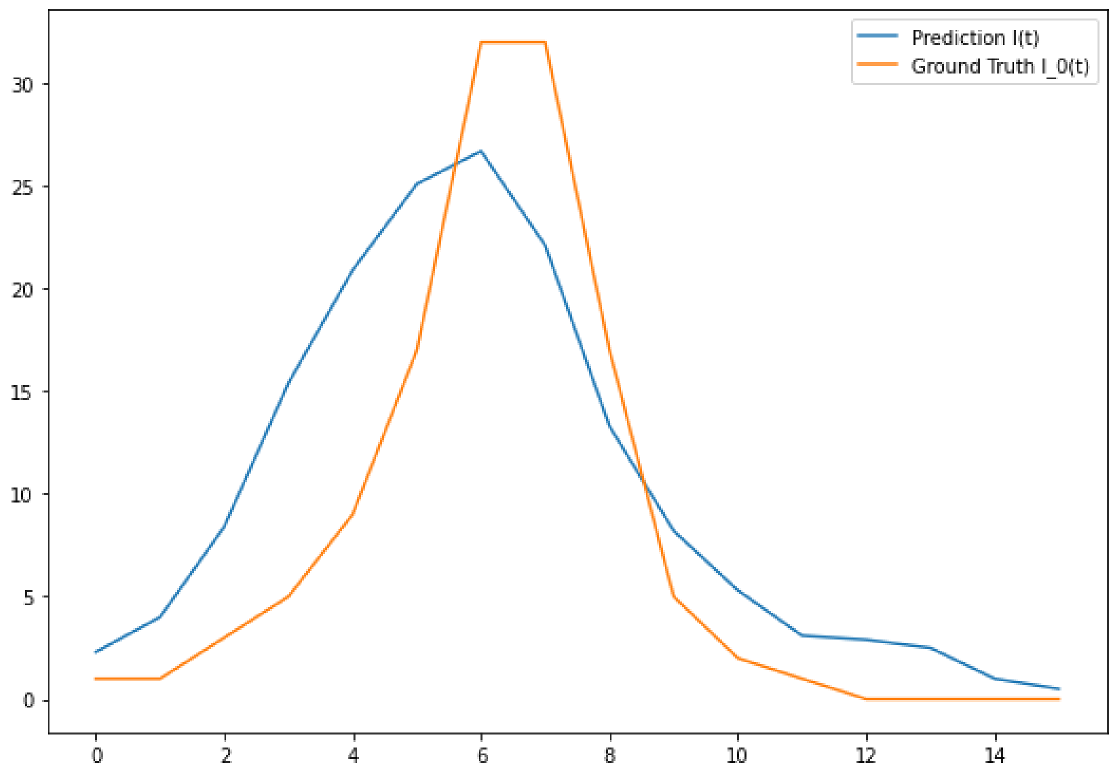
```
n_searches = 27
k0 = 10
beta0 = 0.3
ro0 = 0.5
p0 = 0.5
temperature = 5
alpha = 0.95
graph_type = "small_world_rewiring"

MH = meta_heuristics(graph_type, k0, beta0, ro0, p0, delta_k_set,␣
 ↪delta_beta_set, delta_ro_set, delta_p_set, n_searches, n_nodes, n_weeks,␣
 ↪n_iterations, vacc_t, infected_ground_truth, temperature, alpha)
best_res, best_infected_nodes, best_susceptible_nodes, best_recovered_nodes,␣
 ↪best_new_infected_nodes, best_vacc_perc, best_weekly_vacc_perc = MH.algorithm()
```

```
k0=10, beta0=0.3, ro0=0.5, p0=0.5 delta_k=3, delta_beta=0.2, delta_ro=0.2,
delta_p=0.2
{'beta': 0.4073581211425818, 'ro': 0.5964781506528471, 'k': 7, 'p':
0.6092453258601562, 'rmse': 7.410845093509916}
{'beta': 0.36553150762275805, 'ro': 0.634426409606624, 'k': 7, 'p':
0.5175441354862839, 'rmse': 7.6321687612368745}
worsening solution
{'beta': 0.3156005836501107, 'ro': 0.738447375619915, 'k': 9, 'p':
0.6953099781193698, 'rmse': 6.897191457397713}
{'beta': 0.22331747241281205, 'ro': 0.682543934687289, 'k': 11, 'p':
0.5635453305518296, 'rmse': 7.815649365215919}
worsening solution
{'beta': 0.17332819793111937, 'ro': 0.5069727525263379, 'k': 13, 'p':
0.4096977645923652, 'rmse': 6.216862150635158}
{'beta': 0.1497784845307706, 'ro': 0.6422700302337059, 'k': 12, 'p':
0.4735843209196353, 'rmse': 9.078133067982646}
exit

k0=13, beta0=0.17332819793111937, ro0=0.5069727525263379, p0=0.5 delta_k=2,
delta_beta=0.1, delta_ro=0.1, delta_p=0.1
{'beta': 0.18175646945886026, 'ro': 0.5618149431667976, 'k': 12, 'p':
0.5691462523426483, 'rmse': 7.532927717693832}
{'beta': 0.16056463516059932, 'ro': 0.5477412930017129, 'k': 12, 'p':
0.5862595472544709, 'rmse': 7.430679645900501}
{'beta': 0.18958578541021143, 'ro': 0.5954025893169249, 'k': 12, 'p':
0.5356664250150269, 'rmse': 6.939290309534543}
{'beta': 0.25016826543494786, 'ro': 0.6631646106650013, 'k': 11, 'p':
0.5207201842387308, 'rmse': 6.810791070059337}
{'beta': 0.3152832039000596, 'ro': 0.6910969695075632, 'k': 9, 'p':
0.5017488643010717, 'rmse': 6.978448968073063}
worsening solution
{'beta': 0.2818277533493956, 'ro': 0.7832458827310612, 'k': 10, 'p':
```

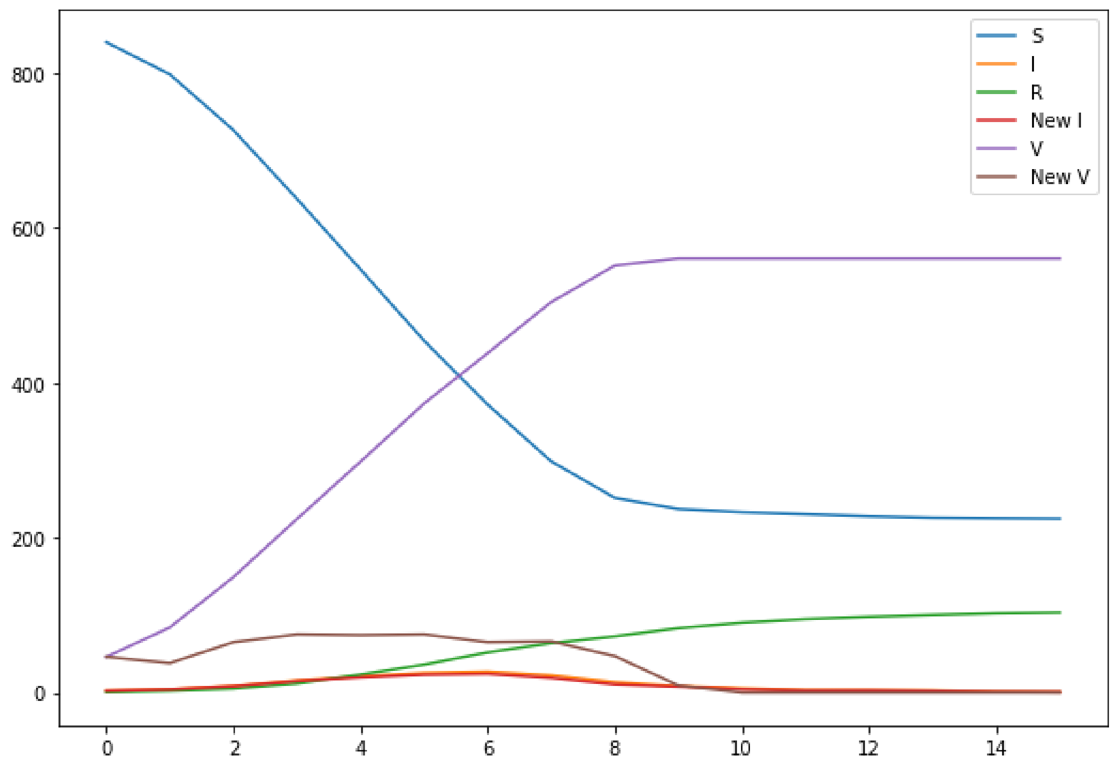
```
0.5091849883021219, 'rmse': 6.377009487212639}
{'beta': 0.3482917943894223, 'ro': 0.8682736949250796, 'k': 9, 'p':
0.5617582139862098, 'rmse': 6.571529502330489}
worsening solution
{'beta': 0.3938855256325323, 'ro': 0.7820468478620197, 'k': 8, 'p':
0.522468933813982, 'rmse': 6.619053935420077}
exit

k0=9, beta0=0.3482917943894223, ro0=0.8682736949250796, p0=0.5 delta_k=1,
delta_beta=0.05, delta_ro=0.05, delta_p=0.05
{'beta': 0.3742244323658531, 'ro': 0.8772494164574987, 'k': 8, 'p':
0.5475205429560178, 'rmse': 6.165529174369382}
{'beta': 0.37922669935531367, 'ro': 0.9236693012733077, 'k': 8, 'p':
0.5378878672821641, 'rmse': 6.863217175640007}
exit

k0=8, beta0=0.3742244323658531, ro0=0.8772494164574987, p0=0.5 delta_k=1,
delta_beta=0.025, delta_ro=0.025, delta_p=0.025
{'beta': 0.3908430695814459, 'ro': 0.8885242412923492, 'k': 8, 'p':
0.5004734664514993, 'rmse': 5.790455508852477}
{'beta': 0.37018851948194, 'ro': 0.8638464369318437, 'k': 8, 'p':
0.5055501545494141, 'rmse': 6.047985201701472}
worsening solution
{'beta': 0.35197946239549766, 'ro': 0.8401996676625256, 'k': 8, 'p':
0.5002753409940934, 'rmse': 6.716630479637836}
exit
```

[ ]:
```python
print(f"Best rmse: {best_res}")
fig, ax = plt.subplots(figsize=(10,7))
ax.plot(best_infected_nodes, label='Prediction I(t)')
ax.plot(infected_ground_truth, label='Ground Truth I_0(t)')
ax.legend(loc='best')
plt.savefig('small_world_rewiring.png')
plt.show()
```

```
Best rmse: {'beta': 0.3908430695814459, 'ro': 0.8885242412923492, 'k': 8, 'p':
0.5004734664514993, 'rmse': 5.790455508852477}
```



```
[ ]: fig, ax = plt.subplots(figsize=(10,7))
     ax.plot(best_susceptible_nodes, label='S')
     ax.plot(best_infected_nodes, label='I')
     ax.plot(best_recovered_nodes, label='R')
     ax.plot(best_new_infected_nodes, label='New I')
     ax.plot(best_vacc_perc, label='V')
     ax.plot(best_weekly_vacc_perc, label='New V')
     ax.legend(loc='best')
     plt.savefig("SIR_small_world_rewiring.png")
     plt.show()
```



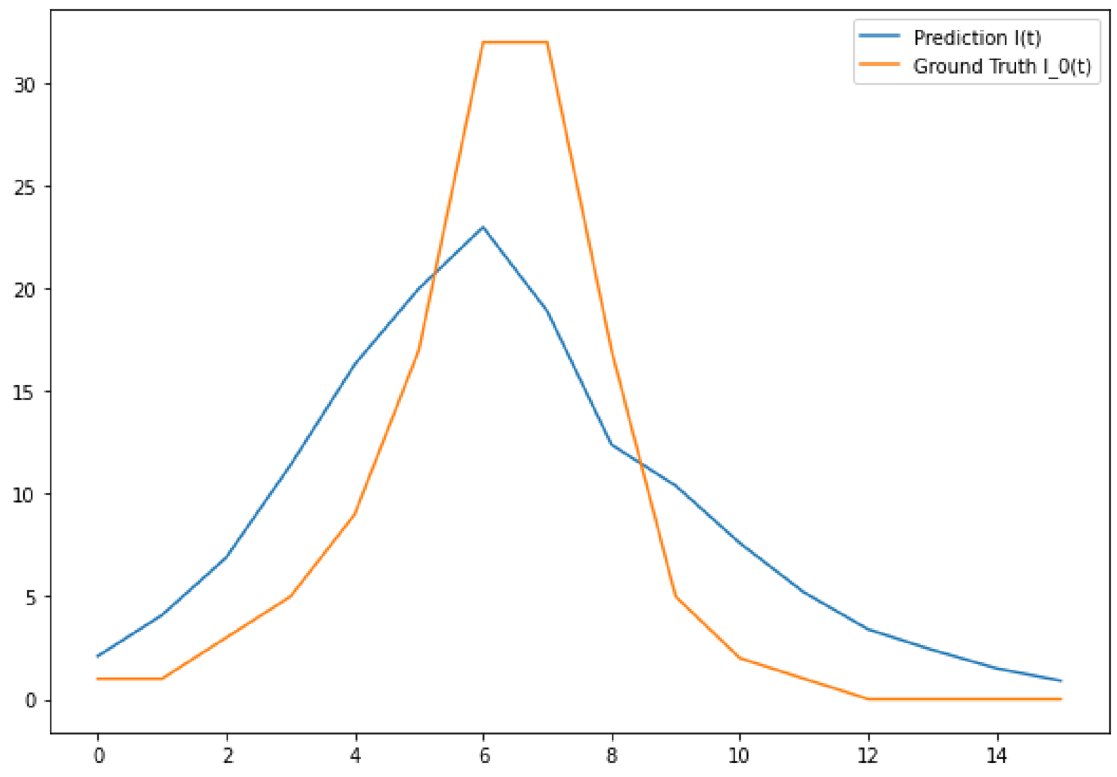
```
[ ]: np.random.seed(0)
delta_beta_set = [0.2, 0.1, 0.05, 0.025]
delta_ro_set = [0.2, 0.1, 0.05, 0.025]
delta_p_set = [0.2, 0.1, 0.05, 0.025]
delta_k_set = [3, 2, 1, 1]
vacc_t = [5, 9, 16, 24, 32, 40, 47, 54, 59, 60, 60, 60, 60, 60, 60, 60]
infected_ground_truth = [1, 1, 3, 5, 9, 17, 32, 32, 17, 5, 2, 1, 0, 0, 0, 0]
n_nodes = 934
n_weeks = len(vacc_t)
n_iterations = 10
n_searches = 27
k0 = 8
beta0 = 0.2
ro0 = 0.5
p0 = 0.5
temperature = 5
alpha = 0.95
graph_type = "small_world_binomial"

MH = meta_heuristics(graph_type, k0, beta0, ro0, p0, delta_k_set,␣
 ↪delta_beta_set, delta_ro_set, delta_p_set, n_searches, n_nodes, n_weeks,␣
 ↪n_iterations, vacc_t, infected_ground_truth, temperature, alpha)
```

```
best_res, best_infected_nodes, best_susceptible_nodes, best_recovered_nodes,␣
 ↪best_new_infected_nodes, best_vacc_perc, best_weekly_vacc_perc = MH.algorithm()
```

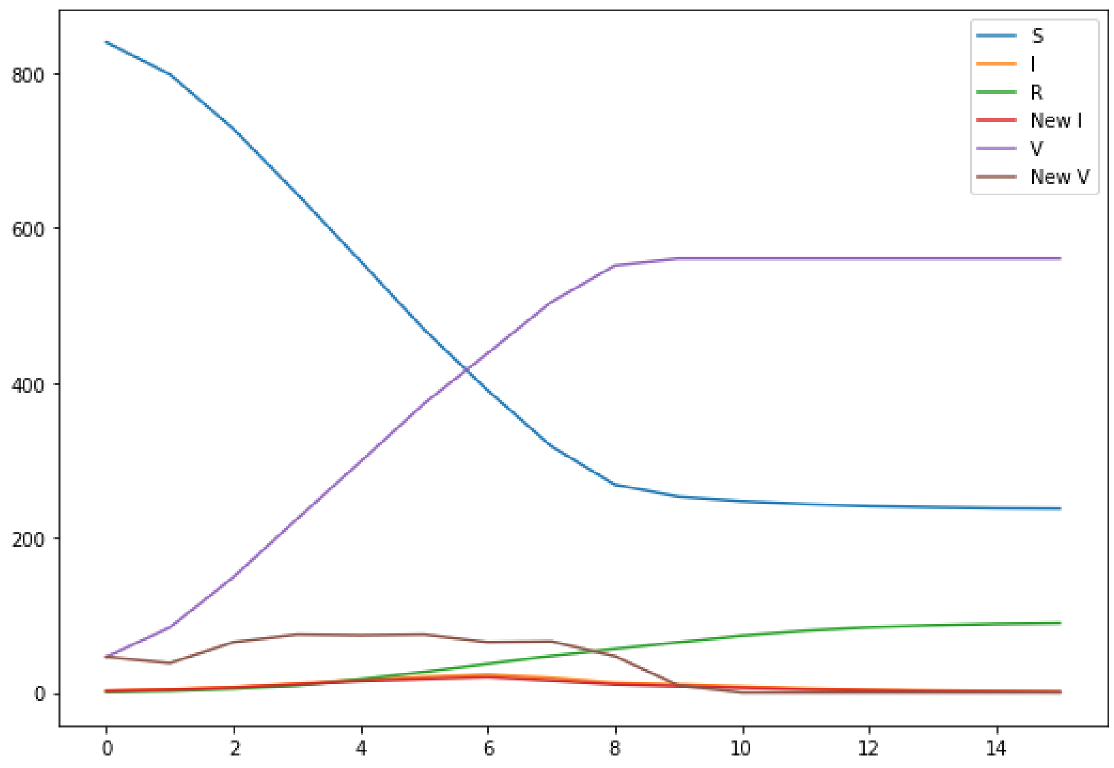
```
k0=8, beta0=0.2, ro0=0.5, p0=0.5 delta_k=3, delta_beta=0.2, delta_ro=0.2,
delta_p=0.2
{'beta': 0.36791648447417336, 'ro': 0.6461479030963413, 'k': 5, 'p':
0.535732909024775, 'rmse': 7.1037402120291535}
{'beta': 0.3424571421660668, 'ro': 0.5974406925336546, 'k': 5, 'p':
0.43447419664587134, 'rmse': 7.223010106596833}
worsening solution
{'beta': 0.5272988135403038, 'ro': 0.5442114319676661, 'k': 3, 'p':
0.6248006609747869, 'rmse': 7.206290654698852}
worsening solution
{'beta': 0.6686435627482853, 'ro': 0.5646614392998326, 'k': 3, 'p':
0.6904627143911153, 'rmse': 7.627950248920086}
worsening solution
{'beta': 0.8637025962628522, 'ro': 0.5923941667067458, 'k': 2, 'p':
0.4798030459856115, 'rmse': 8.285604986963836}
exit

k0=3, beta0=0.36791648447417336, ro0=0.6461479030963413, p0=0.5 delta_k=2,
delta_beta=0.1, delta_ro=0.1, delta_p=0.1
{'beta': 0.42670547542787235, 'ro': 0.7254885405258931, 'k': 4, 'p':
0.5737653026884635, 'rmse': 6.9935237899073455}
{'beta': 0.4832054580020049, 'ro': 0.7303020664019133, 'k': 4, 'p':
0.5454080460741083, 'rmse': 6.765168142773689}
{'beta': 0.5038569038980861, 'ro': 0.7690260726919167, 'k': 5, 'p':
0.42618981187177757, 'rmse': 6.226706593376631}
{'beta': 0.5372015547906455, 'ro': 0.7972541657692065, 'k': 5, 'p':
0.41647225465578136, 'rmse': 6.4216430919197}
worsening solution
{'beta': 0.5188172873948305, 'ro': 0.8255216378907844, 'k': 4, 'p':
0.47646221908978315, 'rmse': 6.355066876123335}
worsening solution
{'beta': 0.513770208460311, 'ro': 0.8456519982166356, 'k': 4, 'p':
0.46958801784837745, 'rmse': 6.716583953171433}
worsening solution
{'beta': 0.45979848129881556, 'ro': 0.871096052823779, 'k': 5, 'p':
0.4689210189034712, 'rmse': 6.711603757672231}
worsening solution
{'beta': 0.5477049695665205, 'ro': 0.8073561758973489, 'k': 4, 'p':
0.43670397947763334, 'rmse': 5.599051258918783}
{'beta': 0.5753680351204783, 'ro': 0.8570274204077686, 'k': 5, 'p':
0.41827279194495165, 'rmse': 6.203426472523069}
exit

k0=4, beta0=0.5477049695665205, ro0=0.8073561758973489, p0=0.5 delta_k=1,
delta_beta=0.05, delta_ro=0.05, delta_p=0.05
{'beta': 0.515106317859018, 'ro': 0.8192484055333848, 'k': 4, 'p':
0.4781201192585485, 'rmse': 5.9400126262492075}
{'beta': 0.5543142626522024, 'ro': 0.8388518677125119, 'k': 4, 'p':
0.5109258721005352, 'rmse': 6.645534214794172}
worsening solution
{'beta': 0.5110043116319962, 'ro': 0.8347563799739605, 'k': 4, 'p':
0.5449217942210297, 'rmse': 6.363911925851897}
worsening solution
{'beta': 0.5595798818370535, 'ro': 0.8749688359897495, 'k': 4, 'p':
0.4503673546100499, 'rmse': 5.987904474855958}
worsening solution
{'beta': 0.5499926619463921, 'ro': 0.8936242760616038, 'k': 4, 'p':
0.47032462854147905, 'rmse': 5.8784032695962605}
{'beta': 0.5386809866888, 'ro': 0.9299542384982689, 'k': 4, 'p':
0.5350487301191029, 'rmse': 6.17555665507167}
exit

k0=4, beta0=0.5499926619463921, ro0=0.8936242760616038, p0=0.5 delta_k=1,
delta_beta=0.025, delta_ro=0.025, delta_p=0.025
{'beta': 0.5272055592871108, 'ro': 0.8734509083099983, 'k': 4, 'p':
0.5237257386410775, 'rmse': 6.3664157891234225}
{'beta': 0.5483152891262983, 'ro': 0.8570731671949027, 'k': 4, 'p':
0.4901296999364391, 'rmse': 6.74736059507716}
exit
```

[ ]:
```
print(f"Best rmse: {best_res}")
fig, ax = plt.subplots(figsize=(10,7))
ax.plot(best_infected_nodes, label='Prediction I(t)')
ax.plot(infected_ground_truth, label='Ground Truth I_0(t)')
ax.legend(loc='best')
plt.savefig('small_small_world_binomial.png')
plt.show()
```

```
Best rmse: {'beta': 0.5477049695665205, 'ro': 0.8073561758973489, 'k': 4, 'p':
0.43670397947763334, 'rmse': 5.599051258918783}
```



```
[ ]: fig, ax = plt.subplots(figsize=(10,7))
     ax.plot(best_susceptible_nodes, label='S')
     ax.plot(best_infected_nodes, label='I')
     ax.plot(best_recovered_nodes, label='R')
     ax.plot(best_new_infected_nodes, label='New I')
     ax.plot(best_vacc_perc, label='V')
     ax.plot(best_weekly_vacc_perc, label='New V')
     ax.legend(loc='best')
     plt.savefig("SIR_small_world_binomial.png")
     plt.show()
```



```
[ ]: np.random.seed(0)
     delta_beta_set = [0.2, 0.1, 0.05, 0.025]
     delta_ro_set = [0.2, 0.1, 0.05, 0.025]
     delta_p_set = None
     delta_k_set = [3, 2, 1, 1]
     vacc_t = [5, 9, 16, 24, 32, 40, 47, 54, 59, 60, 60, 60, 60, 60, 60, 60]
     infected_ground_truth = [1, 1, 3, 5, 9, 17, 32, 32, 17, 5, 2, 1, 0, 0, 0, 0]
     n_nodes = 934
     n_weeks = len(vacc_t)
     n_iterations = 10
     n_searches = 27
     k0 = 8
     beta0 = 0.2
     ro0 = 0.5
     p0 = None
     temperature = 5
     alpha = 0.95
     graph_type = "preferential_attachment"

     MH = meta_heuristics(graph_type, k0, beta0, ro0, p0, delta_k_set,␣
      ↪delta_beta_set, delta_ro_set, delta_p_set, n_searches, n_nodes, n_weeks,␣
      ↪n_iterations, vacc_t, infected_ground_truth, temperature, alpha)
```

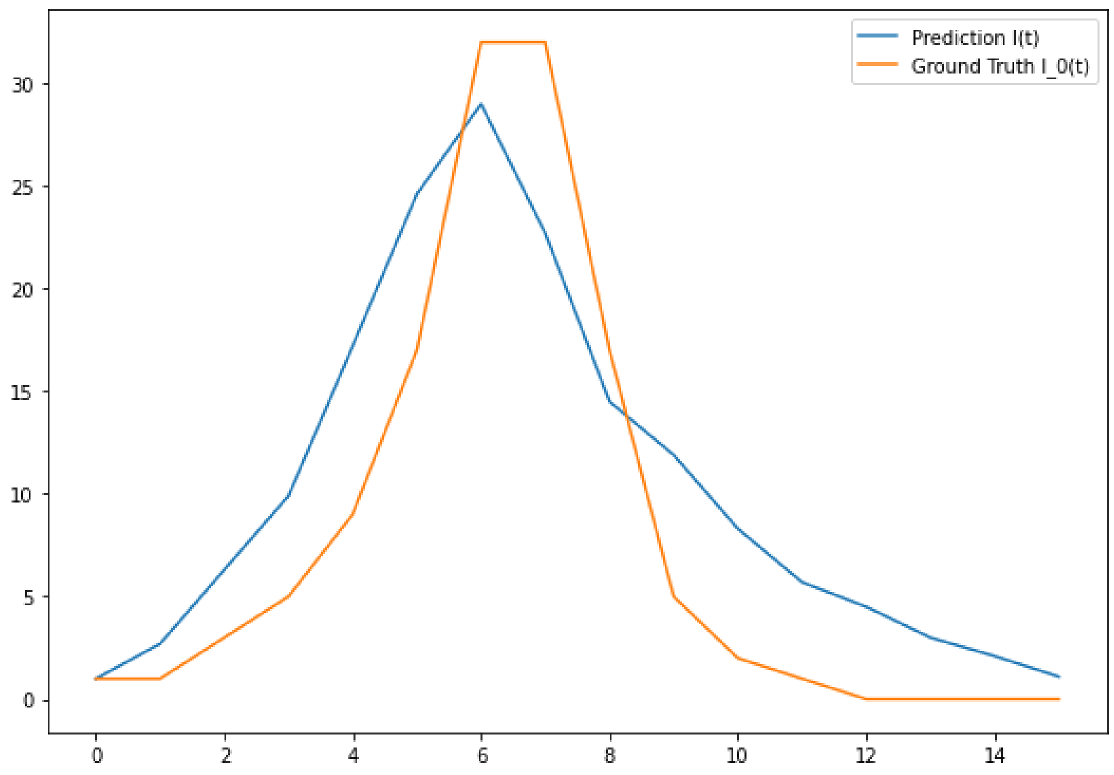
```
best_res, best_infected_nodes, best_susceptible_nodes, best_recovered_nodes,␣
 ↪best_new_infected_nodes, best_vacc_perc, best_weekly_vacc_perc = MH.algorithm()
```

```
k0=8, beta0=0.2, ro0=0.5, p0=None delta_k=3, delta_beta=0.2, delta_ro=0.2,
delta_p=None
{'beta': 0.2280712104712786, 'ro': 0.5815358451693049, 'k': 6, 'p': None,
'rmse': 7.783275338827479}
{'beta': 0.41846414058353176, 'ro': 0.440095605562391, 'k': 4, 'p': None,
'rmse': 5.985294478970939}
{'beta': 0.28021827982488046, 'ro': 0.2636516199793778, 'k': 4, 'p': None,
'rmse': 6.989545764926358}
exit

k0=4, beta0=0.41846414058353176, ro0=0.440095605562391, p0=None delta_k=2,
delta_beta=0.1, delta_ro=0.1, delta_p=None
{'beta': 0.42007172437223267, 'ro': 0.340145655005204, 'k': 3, 'p': None,
'rmse': 6.404051452010672}
{'beta': 0.4139360677859195, 'ro': 0.4166863581829779, 'k': 3, 'p': None,
'rmse': 7.781588848043824}
exit

k0=3, beta0=0.42007172437223267, ro0=0.340145655005204, p0=None delta_k=1,
delta_beta=0.05, delta_ro=0.05, delta_p=None
{'beta': 0.3854647633270771, 'ro': 0.2960097367462295, 'k': 3, 'p': None,
'rmse': 6.907197333216997}
{'beta': 0.35081716375574057, 'ro': 0.3219314705558319, 'k': 3, 'p': None,
'rmse': 7.206247289678589}
worsening solution
{'beta': 0.39584003290841685, 'ro': 0.2719777213041259, 'k': 3, 'p': None,
'rmse': 6.202318115027639}
{'beta': 0.3852817310108174, 'ro': 0.2843707186081308, 'k': 3, 'p': None,
'rmse': 7.626966959414469}
exit

k0=3, beta0=0.39584003290841685, ro0=0.2719777213041259, p0=None delta_k=1,
delta_beta=0.025, delta_ro=0.025, delta_p=None
{'beta': 0.38391038874677974, 'ro': 0.2949373967721126, 'k': 3, 'p': None,
'rmse': 5.739120141624499}
{'beta': 0.3791227035641068, 'ro': 0.27834098910702826, 'k': 3, 'p': None,
'rmse': 6.9973655757006155}
worsening solution
{'beta': 0.3919570148792955, 'ro': 0.2944240723967337, 'k': 3, 'p': None,
'rmse': 6.907830701457586}
worsening solution
{'beta': 0.3908144218642722, 'ro': 0.31813048948590805, 'k': 3, 'p': None,
'rmse': 7.174477332879378}
worsening solution
```

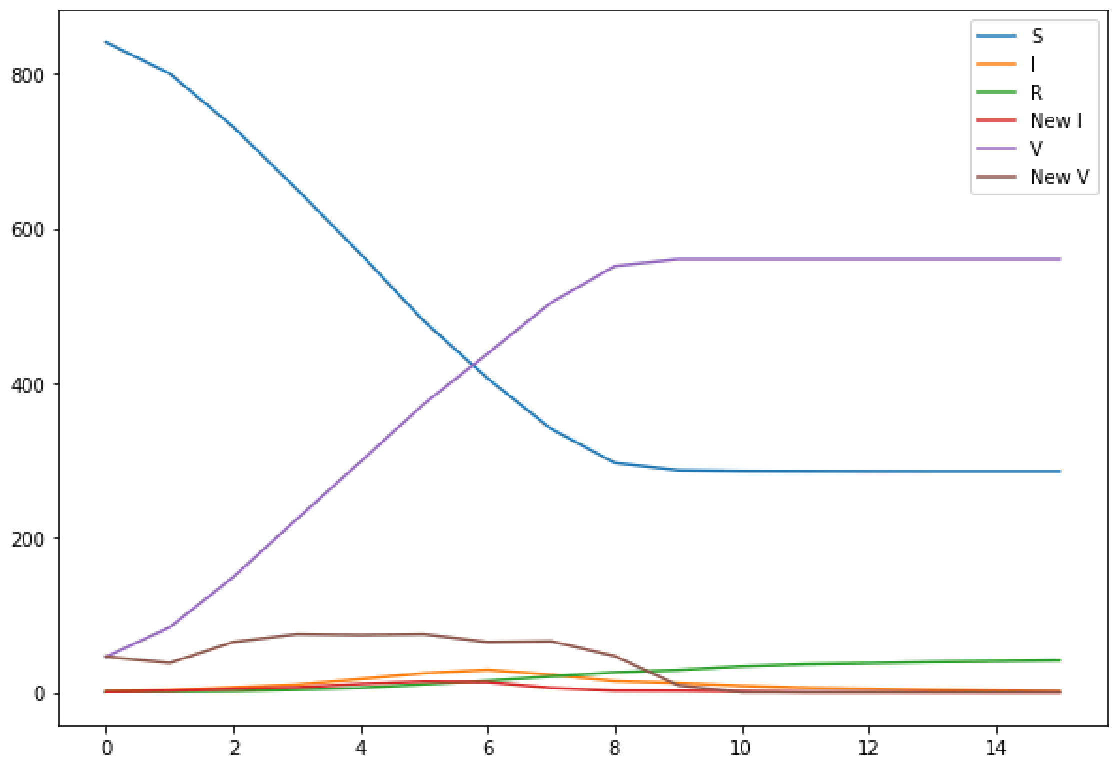
```
{'beta': 0.392081551330391, 'ro': 0.31543947042796067, 'k': 3, 'p': None,
'rmse': 6.694447325956042}
worsening solution
{'beta': 0.41126032896072845, 'ro': 0.3100674064588306, 'k': 3, 'p': None,
'rmse': 5.950682733939022}
worsening solution
{'beta': 0.4235426194656072, 'ro': 0.3019533742034916, 'k': 3, 'p': None,
'rmse': 6.49942305131771}
worsening solution
{'beta': 0.4182936552492103, 'ro': 0.3172397356209809, 'k': 3, 'p': None,
'rmse': 5.054638958422253}
{'beta': 0.4050018912845084, 'ro': 0.3121142479464371, 'k': 3, 'p': None,
'rmse': 8.198322999248079}
exit
```

```
[ ]: print(f"Best rmse: {best_res}")
     fig, ax = plt.subplots(figsize=(10,7))
     ax.plot(best_infected_nodes, label='Prediction I(t)')
     ax.plot(infected_ground_truth, label='Ground Truth I_0(t)')
     ax.legend(loc='best')
     plt.savefig('preferential_attachment_challenge.png')
     plt.show()
```

```
Best rmse: {'beta': 0.4182936552492103, 'ro': 0.3172397356209809, 'k': 3, 'p':
None, 'rmse': 5.054638958422253}
```



```
[ ]: fig, ax = plt.subplots(figsize=(10,7))
     ax.plot(best_susceptible_nodes, label='S')
     ax.plot(best_infected_nodes, label='I')
     ax.plot(best_recovered_nodes, label='R')
     ax.plot(best_new_infected_nodes, label='New I')
     ax.plot(best_vacc_perc, label='V')
     ax.plot(best_weekly_vacc_perc, label='New V')
     ax.legend(loc='best')
     plt.savefig("SIR_preferential_attachment_challenge.png")
     plt.show()
```


S
I
R
New I
V
New V
800
600
400
200
0
0
2
4
6
8
10
12
14