Quantitative Viral Dynamics Across Scales

Digital Workshop:

Epidemic Spreading on Networks

March 23rd, 2022

Francesco Bonacina [1], Benjamin Faucher [1], Chiara Poletto [1]

(1) INSERM, Sorbonne Université, Pierre Louis Institute of Epidemiology and Public Health, Paris, France

Introduction

The spread of directly transmitted infectious diseases is mediated by human-to-human contacts - e.g. face-to-face contacts for airborne infections, sexual contacts for HIV, gonorrhea or other sexually transmitted diseases. Accounting for the complexity of human contact patterns is thus essential to capture individuals' heterogeneities in transmission. Network theory provides a powerful mathematical framework to reach this goal. Mathematical tools from network theory can synthesize the empirical records of human contacts and allow realistic modelling of the spreading dynamics to assess the infection risk and quantify the impact of interventions.

During this workshop we will analyze an empirical network of face-to-face contacts reconstructed by the Sociopatterns.org collaboration. We will then learn how to model the spread of an infection on the top of the network. As an example, we will consider an infection that confers full immunity to the pathogen. Computer simulations of the network-based spreading model allow identifying the individuals who cover a major role in the spreading of the infection. This information is essential for risk assessment and enables the design of intervention strategies. We will study the example of vaccination. Assuming a limited amount of vaccines is available - i.e. not enough to vaccinate the whole population - we will use the model to identify optimal strategies for vaccines' allocation.

Workshop outline

1) Network analysis: we will analyze an empirical network of contacts collected by the Sociopatterns.org collaboration. We will consider the network of face-to-face contacts among 75 people in a geriatric hospital.

2) Epidemic on network: we will develop a computation model for the spread of an infection on the top of the network. We will use the SIR modelling framework as an example.

3) Role of the seed: we will analyze how the epidemic dynamics changes with the contact behaviour of the seeding node.

4) Vaccination: we will compare different vaccination scenarios where a proportion of the population is vaccinated.

Let's start!

The work presented here has been coded in Python 3.8.5, by using a few foundamental packages:

- `pandas`, version 1.2.3.
- `numpy`, version 1.19.2.
- `matplotlib`, version 3.3.4

In any case, if your Python version is > 3, you should have no problems.

Please run the following before going further.

In [ ]:
```
# Load a few packages
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import time
```

1) Analysis of a network of face-to-face interactions

Face-to-face contact network in a hospital

Contacts in specific settings can be recorded with active Radio Frequency Identification (RFID), small devices that exchange low-power radio packets among each other. By asking people to wear these devices it is possible to record their face-to-face close proximity interactions with high spatiotemporal resolution (Cattuto et al PLOS ONE 2010, Salathé et al PNAS 2010, Isella JTB 2010, Stehlé BMC Medicine 2011, Obadia et al PLOS Comput Biol 2015). Collected data are fully anonymized.

We will analyze a dataset collected with RFID and made freely available by the Sociopatterns.org collaboration (http://www.sociopatterns.org/). Sociopatterns.org is an interdisciplinary research collaboration, that since 2008 has deployed RFID in a

variety of settings, reconstructing the pattern of face-to-face contacts at museums, conferences, schools, workplaces, hospitals, Africa rural settings, among the others.

The dataset analyzed here was collected in a hospital ward. 75 individuals including medical doctors (MED), nurses (NUR), administrative staff (ADM) and patients (PAT) wore the RFID device for four days. The original dataset contains a temporal network of contacts sampled with a temporal resolution of 20 seconds.

More information on the deployment and the analysis of the data can be found in the associated publication: *P. Vanhems et al., Estimating Potential Infection Transmission Routes in Hospital Wards Using Wearable Proximity Sensors, PLoS ONE 8(9): e73970 (2013) https://journals.plos.org/plosone/article? id=10.1371/journal.pone.0073970 (https://journals.plos.org/plosone/article? id=10.1371/journal.pone.0073970)*.

Data were downloaded from the sociopatterns.org website: http://www.sociopatterns.org/datasets/hospital- ward-dynamic-contact-network/ (http://www.sociopatterns.org/datasets/hospital-ward-dynamic-contact- network/)

We aggregated the network into a weighted, static, undirected network. Individuals in the hospital are represented by nodes associated with a status: medical doctors (MED), nurses (NUR), administrative staff (ADM) and patients (PAT). A contact between node and i, j is encoded by an edge (tuple (i, j)) and a weight which represents the number of times this contact occurred during the four days. Note that a contact corresponds to an interaction of 20 seconds. Thus, a weight of, e.g., 4 between node i and j means that during the four days, the two nodes had an interaction of 4 x 20s = 80s in total.

Please load and check the dataset using the following code.

The `hwn` dataframe stores the network of contacts.

In [ ]:
```
hwn = pd.read_csv('./hospital_weighted_net.csv')
hwn.head()
```

The `ns` dataframe stores the status of each node.

In [ ]:
```
ns = pd.read_csv('./node_status.csv')
ns.head()
```

Challenge problem 1: Analysis of the weighted degrees

Here we will study the weigthed degree distribution of the network. The degree of a node i is the number of edges departing from node i. In a weighted network, the weighted degree (or strength) of node i is the sum of the weights of all edges of i.

- Compute a numpy array `nodes` of all nodes, a numpy array `edges` of all edges, the number of nodes `nn` and of edges `ne`.
- Compute a numpy array `weighted_degrees` of the weighted degree of all nodes.
- Compute the mean and the standard deviation of the weighted degree distribution. Plot its histogram.

In [ ]:
```
### Challenge problem 1
#   Fill here with your code!   ****************
```

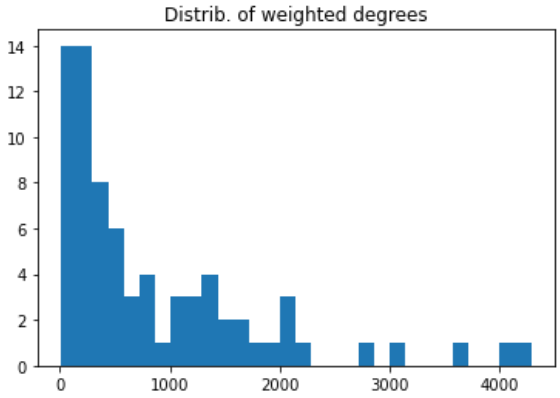
Challenge problem 1 - Output

Nodes, edges and size of the network:

- Nb of nodes: 75
- First three and last three elements of the list of nodes: array([0, 1, 2]) ... array([72, 73, 74])
- Nb of edges: 1139

Weighted degrees:

- mean: 865
- standard deviation: 957



Challenge problem 2: Analysis of the weighted degrees by status

- Define the numpy arrays `administratives`, `medicals`, `nurses` and `patients` containing the nodes of each status;
- Compute how many nodes there are for each status;
- Compute the mean and the standard deviation of the weighted degrees for each group of nodes with the same status;
- Plot the histograms of the weighted degrees for each group of nodes with the same status.

In [ ]:
```
### Challenge problem 2
#   Fill here with your code!   ****************
```

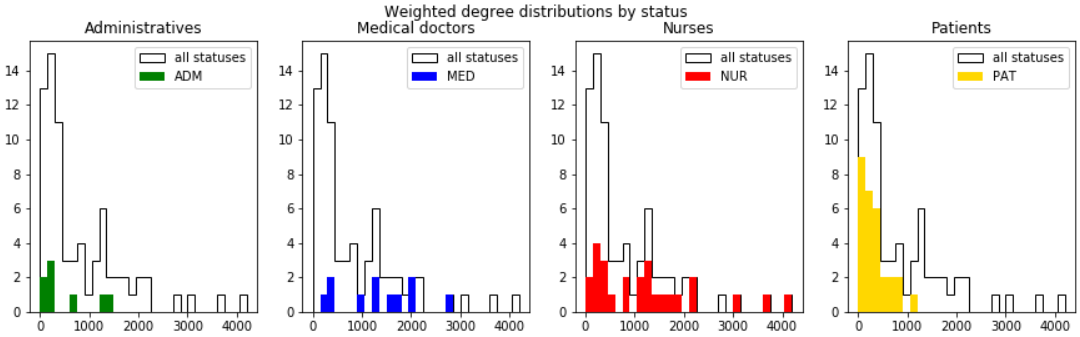
Challenge problem 2 - Output

Nodes by status:

- ADM: 8
- MED: 11
- NUR: 27
- PAT: 29
- total: 75

Weighted degrees by status (mean and standard deviation):

- ADM: 507 535
- MED: 1365 776
- NUR: 1356 1197
- PAT: 316 266



How networks are efficiently stored

A convenient way to work with networks is to define a list of lists that stores for each node its neighbours. Another list of lists can be used to store the weights of each link.

In the following we will work with the following variables:

- `neigbours` : a list of N lists, where N is the number of nodes. The j-th element of the i-th list is the j-th neighbour of the node i
- `weights` : a list of N lists, where N is the number of nodes. The j-th element of the i-th list is the weight of the edge (i, j)

In [ ]:
```
neighbours = [[] for n in nodes] # list of lists: the ith list will contain all the neighbours of node i
weights = [[] for n in nodes]    # list of lists: the ith list will contain the weights of edges of node i

for e in range(ne):

    i = hwn['i'].iloc[e]
    j = hwn['j'].iloc[e]
    w = hwn['weight'].iloc[e]
```

```
    neighbours[i].append(j)
    neighbours[j].append(i)
    weights[i].append(w)
    weights[j].append(w)
```

1.4. BONUS - network visualization

As a bonus we can visualize the network using the NetworkX package (https://networkx.org/). This is a python library to handle graphs.

Please, install the current release of `networkx` with `pip` from your terminal:

- `$ pip install --upgrade pip`
- `$ pip install networkx`

For help look at https://networkx.org/documentation/stable/install.html#

Then, import the package by running the following cell:

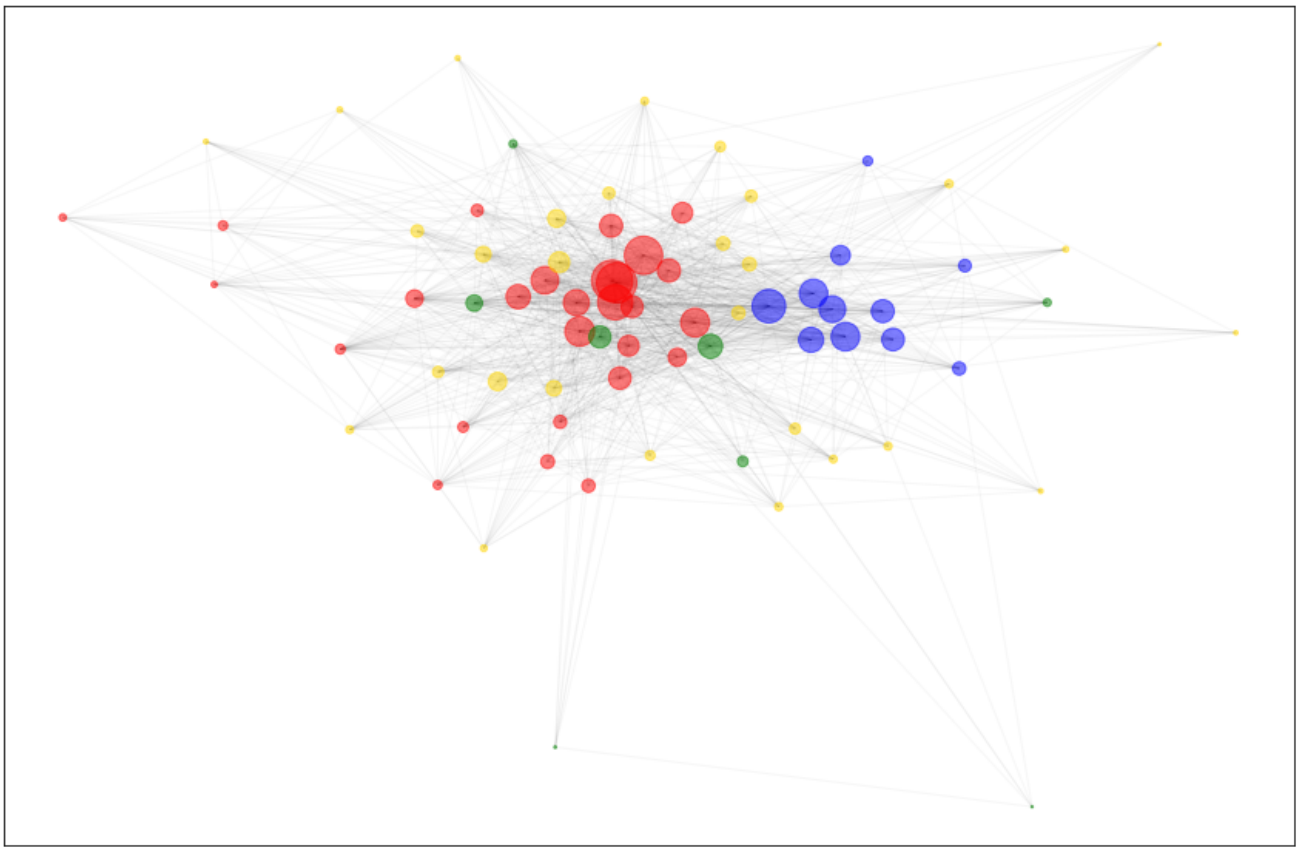
In [ ]:
```
import networkx as nx
```

We show here how to visualize the network that we are studying.

In [ ]:
```
# create the network
HWN = nx.Graph()

# add edges and weights
HWN.add_weighted_edges_from([(hwn['i'].iloc[c], hwn['j'].iloc[c], hwn['weight'].iloc[c]) for c in range(ne)])
```

Visualize the graph using the spring layout

In [ ]:
```
fig = plt.figure(figsize=(15,10))

# draw nodes
node_pos = nx.spring_layout(HWN, seed=123456)                    # set positions of nodes
colors = {'ADM':'green', 'MED':'blue', 'NUR':'red', 'PAT':'gold'} # define a color for each status
nx.draw_networkx_nodes(G=HWN,                                      # graph
                       pos=node_pos,                               # node positions, set using spring_layout()
                       nodelist=nodes,                             # list of nodes
                       node_size=[d/6 for d in weighted_degrees],  # list of sizes, proportional to the degrees
                       node_color=[colors[s] for s in ns['status']],# list of colors, set looking at the status
                       alpha=0.5)
# draw edges
nx.draw_networkx_edges(G=HWN,
                       pos=node_pos,
                       alpha=0.03)
plt.show()

# nurses: red
# patients: gold
# medical doctors: blue
# administratives: green
```



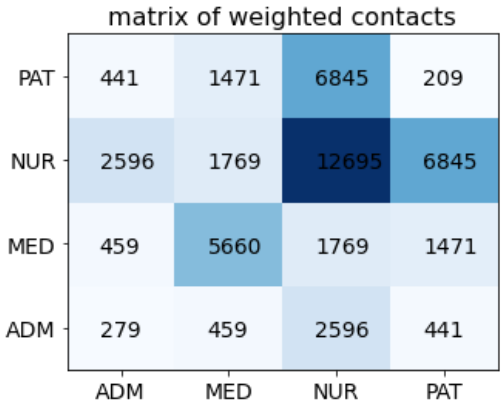
1.5. BONUS Challenge problem - matrix of weighted contacts among groups

We can also study how the different persons in the hospital interact with each other depending on their status.

Compute and visualize the weighted contact matrix according to nodes' status. Here the matrix element (a, b) encode the total number of contacts between the status a and the status b (which, in our problem, is proportional to the total duration of the contacts between nodes of status a and nodes of status b).

You can use the function `pcolormesh` of the `matplotlib.pyplot` library to draw a heatmap of the contact matrix.

In [ ]:
```
### BONUS Challenge problem 3
#   Fill here with your code!   ****************
```

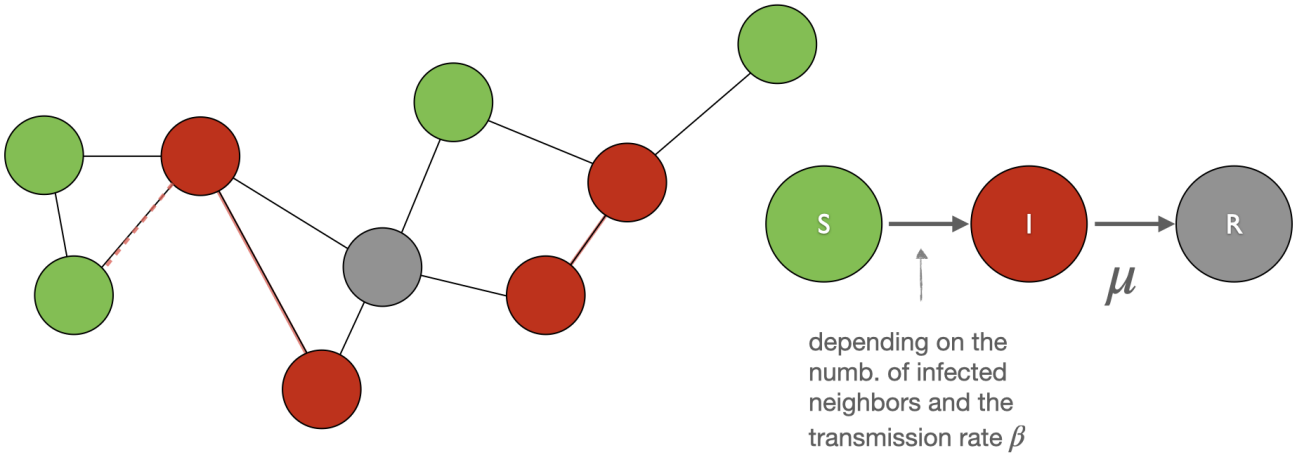


BONUS Challenge problem 3 - Output

2) Epidemic on network

The SIR model

We will now study how an epidemic spreads on the hospital network. To this end, we use here a discrete-time stochastic SIR model. To each individual (node of the network) we assign an infectious status among Susceptible (S), Infectious (I) or Recovered (R). Transmission occurs through the links of the network. We assume that a Susceptible individual, i, can be infected by each of its Infected neighbors, j, with a probability that depends on a transmissibility parameter (β), and the weight of the link (i, j). We can assume a binomial process, i.e. the probability per unit time a node j infects i is given by $1 - (1 - \beta)^{w_{ij}}$.



Summarizing, at each time step :

- If node i is Susceptible, it can be infected by an Infectious neighbor j with probability $1-(1-\beta)^{w_{ij}}$;
- If node i is Infectious, it can become Recovered with a probability μ. This gives an average infectious period of $\frac{1}{\mu}$
- A Recovered node cannot be infected anymore.

Importantly, at each time step, all nodes' transitions are synchronous. This means that the update of the infectious status of nodes must be done at the end of the time step.

Challenge problem 4: Implementation of the SIR() function

Implement the function `SIR()` that simulates the stochastic spread of an epidemic on a weighted network.

We propose you to implement a function that takes as **inputs** the following parameters:

- parameters which encode the topology of the weighted network
  - `nodes` : the list of nodes of the network, from 0 to the N-1, where N is the number of nodes
  - `neigbours` : the list of N lists, where the j-th element of the i-th list is the j-th neighbour of the node i
  - `weights` : the list of N lists, where the j-th element of the i-th list is the weight of the edge (i,j)
- the spreading parameters of the SIR model:
  - `beta` : float between 0 and 1, it is the transmissibility parameter
  - `mu` : float between 0 and 1 : the recovery rate
- other parameters in order to simulate different scenarios:
  - `initial_infected` : integer, the node that will be infected at time t = 0. You can put 0 as default argument
  - `initial_recovereds` : list of nodes already Recovered at time t=0. This parameter will be useful later to study vaccination. You can put an empty list as default argument.

The function will return 4 arrays as **outputs**:

- `time` : the timesteps ([1,2,3,...,t_{end}])
- `S` : the number of Susceptibles at each time step
- `I` : the number of Infectious at each time step
- `R` : the number of Recovered at each time step

Note that the 4 arrays must have the same length

The function must stop when there is no infected anymore.

In order to simulate stochastic dynamics, we will sample random numbers by using the `random` package.

In [ ]:
```
import random
```

In [ ]:
```
```

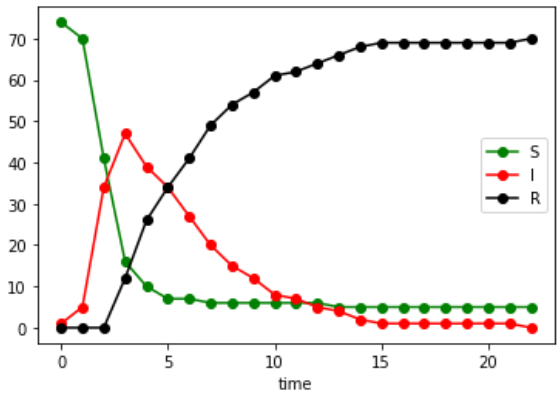
```
### Challenge Problem 4

def SIR(nodes, neighbours, weights, beta, mu, initial_infected=0, initial_recovereds=[]):

    '''
    INPUT
    nodes : list of nodes, from 0 to N-1, where N is the number of nodes
    neighbours : list of N lists. The i-th list contains all the neighbours of node i
    weights : list of N lists. The i-th list contains the weights of edges of node i
    beta : transmissibility parameter, that is the probability of transmission per unit time
    mu : recovery rate
    initial_infected : node that will be infected at the beginning
    initial recovereds : list of nodes that will be immunized at the beginning

    OUTPUT
    time,S,I,R: numpy arrays with time steps and series of number of infectd, susceptible, recovered
    '''

    # *****   Write here your code!   **********************************************


    # ***************************************************************************

    return np.array(time), np.array(S), np.array(I), np.array(R)
```

We can now call the function `SIR()` and run one single stochastic realization of an epidemic on the network. We can then plot the resulting trajectories of S(t), I(t) and R(t) in a single plot.

In [ ]:

```
# spreading parameters
mu = 0.3
beta = 0.005

# run simulation
t, S, I, R = SIR(nodes, neighbours, weights, beta, mu)

# plot results
plt.plot(t, S, 'g-o', label='S')
plt.plot(t, I, 'r-o', label='I')
plt.plot(t, R, 'k-o', label='R')
plt.legend(loc='center right')
plt.xlabel('time')
plt.show()
```

Problem 4 - output



3) Role of seeds

The spreading dynamics is critically affected by the transmission and recovery parameters and by initial conditions. In particular, the individual who seeds the infection covers a central role in the dynamics. Certain individuals may be central nodes of the network able to infect a larger number of nodes and generate larger epidemics. For the case of the hospital networks, we have seen that individuals with a different status (nurseys, patients, etc.) tend to have a different number of connections. We analyze here whether the status of the seeding node has an impact on the epidemic dynamics. To better highlight the role of node status, we average over different stochastic realizations obtained with the same seeding node and average over all seeding nodes with same status.

We provide here a piece of code that loop over different seeding nodes and stochastic realizations and then average the trajectories of Recovered so obtained. We also store the final Attack Rate for each stochastic run and seeding node and make the histogram. (We recall that the final Attack Rate is the total number of infected people at the end of the epidemic).

Challenge problem 5: Explore the role of seed

Analyze the impact of the status of the seeding node with different transmission and recovery parameters. Try: β = 0.001, 0.005, 0.1 with constant μ = 0.3.

Note: the variable `niter_per_seed` controls the number of stochastic simulations per seed. You test different values to see how this affects the stability of the results. Do not go over 500 simulations per seed not to be too long.

Before getting started

Before getting started with this we need to create a function to define the length of the arrays obtained from the SIR simulations. This is because, when we repeatedly run the SIR() function we simulate several epidemics that probably end at different times. This means that each time the SIR() function returns arrays (time, S, I, R) with different lengths. However, in order to average the results of different simulations, we need to set those arrays to the same length. In particular, in the following we will focus on the `R` array.

In [ ]:
```
def fix_R(R, tmax):
    '''
    Function to define the length of the R array according to tmax. R is the array obtained from the
    SIR simulation containing the total number of recovered people at each time step.
    '''

    lenR = len(R)
    if lenR>=tmax:   # if R is longer than tmax, we cut it
        Rfixed = R[:tmax]
    else:            # otherwise, we extend it
        Rfixed = np.append(R, [R[-1]]*(tmax-lenR))

    return Rfixed
```

Getting started

Set the parameters `beta` and `mu` and explore the dynamics

In [ ]:
```
# Challenge problem 5: set spreading parameters
beta =
mu =
tmax = 35   # maximum number of temporal steps that we will consider

# simulation parameters
niter_per_seed = 500
```

Run the simulations

In [ ]:
```
# check time of execution
start_time = time.time()

# matrix to store results of simulations
seed_results = np.zeros((nn,tmax))

# loop over each node as initial seed
for n in nodes:

    # matrix to store results of simulations for a given seed
    mat_tmp = np.zeros((niter_per_seed, tmax))

    # run several realizations for each seed
    for i in range(niter_per_seed):

        # simulate the spreading
        t, S, I, R = SIR(nodes, neighbours, weights, beta, mu, initial_infected=n)

        # fix R: we extend the R vector up to tmax
        R = fix_R(R, tmax)

        # save results
        mat_tmp[i,:] = R

    # average over the simulations of the seed
    Rseed = np.mean(mat_tmp, axis=0)

    # store results of the seed
    seed_results[n,:] = Rseed

print("--- %s seconds ---" % (time.time() - start_time))
```

Plot the results

In [ ]:
```
fig = plt.figure(figsize=(9,6))

# matrices of results by status
adm_results = seed_results[administratives,:]
med_results = seed_results[medicals,:]
nur_results = seed_results[nurses,:]
pat_results = seed_results[patients,:]
```

```
x = np.arange(0,tmax)

# plot averages evolutions by status
colors = {'ADM':'green', 'MED':'blue', 'NUR':'red', 'PAT':'gold'} # define a color for each status
plt.plot(x, np.mean(adm_results, axis=0), color=colors['ADM'], lw=3, label='ADM')
plt.plot(x, np.mean(med_results, axis=0), color=colors['MED'], lw=3, label='MED')
plt.plot(x, np.mean(nur_results, axis=0), color=colors['NUR'], lw=3, label='NUR')
plt.plot(x, np.mean(pat_results, axis=0), color=colors['PAT'], lw=3, label='PAT')

plt.xlabel('time')
plt.ylabel('Nb of recovered')
plt.legend(title=r'$\beta=$%4.3f, $\mu=$%2.1f'%(beta, mu), loc='lower right')
plt.title('SIR dynamics: seed dependency')

fig.tight_layout()
```

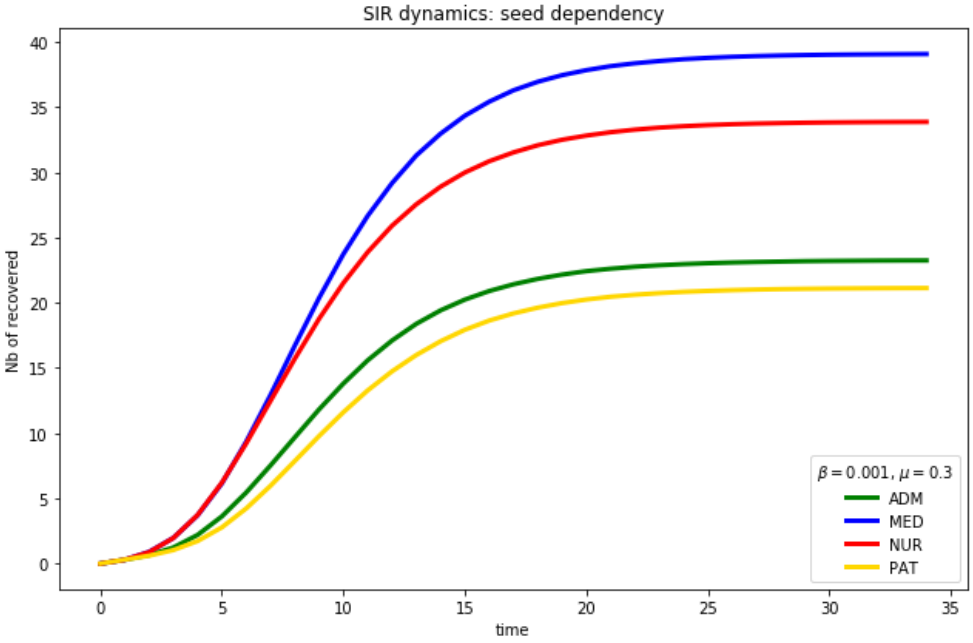
In [ ]:

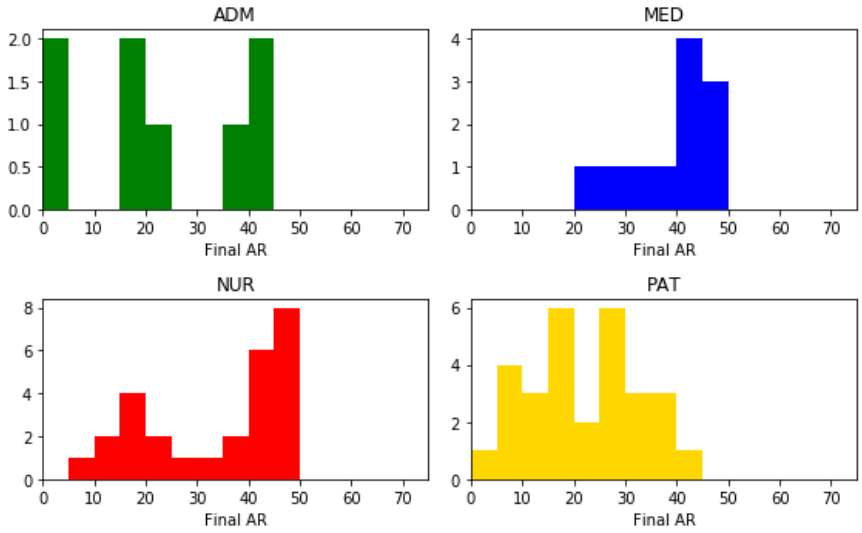
```
# histograms of final AR

ARadm = [adm_results[n,-1] for n in range(nadm)]
ARmed = [med_results[n,-1] for n in range(nmed)]
ARnur = [nur_results[n,-1] for n in range(nnur)]
ARpat = [pat_results[n,-1] for n in range(npat)]

fig = plt.figure(figsize=(8,5))
ax1 = fig.add_subplot(221)
ax2 = fig.add_subplot(222)
ax3 = fig.add_subplot(223)
ax4 = fig.add_subplot(224)

bins = np.arange(0,76,5)
ax1.hist(ARadm, bins=bins, color=colors['ADM'])
ax1.set_title("ADM")
ax1.set_xlabel("Final AR")
ax1.set_xlim((0,75))
ax2.hist(ARmed, bins=bins, color=colors['MED'])
ax2.set_title("MED")
ax2.set_xlabel("Final AR")
ax2.set_xlim((0,75))
ax3.hist(ARnur, bins=bins, color=colors['NUR'])
ax3.set_title("NUR")
ax3.set_xlabel("Final AR")
ax3.set_xlim((0,75))
ax4.hist(ARpat, bins=bins, color=colors['PAT'])
ax4.set_title("PAT")
ax4.set_xlabel("Final AR")
ax4.set_xlim((0,75))

fig.tight_layout()
```

Challenge problem 5 - outputs

Example of the output for one set of parameters

$\beta = 0.001$





4) Vaccination

The planning of a vaccination campaign needs to cope with limited resources and logistic constraints. When limited vaccines are available, models can help design optimal allocation strategies. We here analyze the impact of different choices regarding vaccine allocation.

We consider the case in which 23 vaccines are available, enough to cover ~30% of the population. Our assumption is that the vaccine has an effectiveness of 100%, so that vaccinated people are completely immunized. We assume vaccines are distributed before the start of the epidemic. Therefore, vaccinated individuals are modeled by setting their infectious status to Recovered at the beginning of the simulations.

We will compare a scenario without vaccination with different vaccination scenarios. We aim at identifying the strategy that better mitigate the epidemic by comparing the following strategies:

- the *naive* strategy, where we vaccinate 23 people at random;
- The *network-based* strategy, where we vaccinate the 23 persons with the highest number of contacts;
- Two *status-based* strategies, where we vaccinate 23 people according to their status. We will compare a strategy where we vaccinate 23 individuals randomly chosen among nurses and medical doctors with a strategy where we vaccinate 23 randomly chosen patients.

We expect the *network-based* strategy to be more effective. However the *status-based* strategies are more realistics, because in real life the information of the face-to-face contact network is in general not available. The information of the status of individuals is in general available and as we have seen so far could be used as a proxy for individuals' number of interactions.

In [ ]:
```
# spreading parameters
beta = 0.005
mu = 0.3

# simulation parameters
ndoses = 23
niter_per_seed = 10
tmax = 35
```

Run the simulation for the scenario without vaccines to be used as reference scenario

In [ ]:
```
# check time of execution
start_time = time.time()

# matrix to store results of simulation
mat_novax = np.zeros((niter_per_seed*nn, tmax))

c = 0   # simulation counter
# for each node as seed
for n in range(nn):

    # iterating several times for each seed
    for i in range(niter_per_seed):

        # simulations
        t, S, I, R = SIR(nodes, neighbours, weights, beta, mu, initial_infected=nodes[n])

        # fix R
        R = fix_R(R, tmax)

        # save results
        mat_novax[c,:] = R
        c += 1

print("--- %s seconds ---" % (time.time() - start_time))
```

Challenge problem 6: define the list of nodes to be vaccinated

Complete the code below by defining the list of nodes to be vaccinated for each strategy and then run simulations for these scenarios:

- 1: vaccination of 23 people at random;
- 2: vaccination of 23 patients at random;
- 3: vaccination of 23 nurses/medical doctors at random;
- 4: vaccination of the 23 nodes with the highest degree.

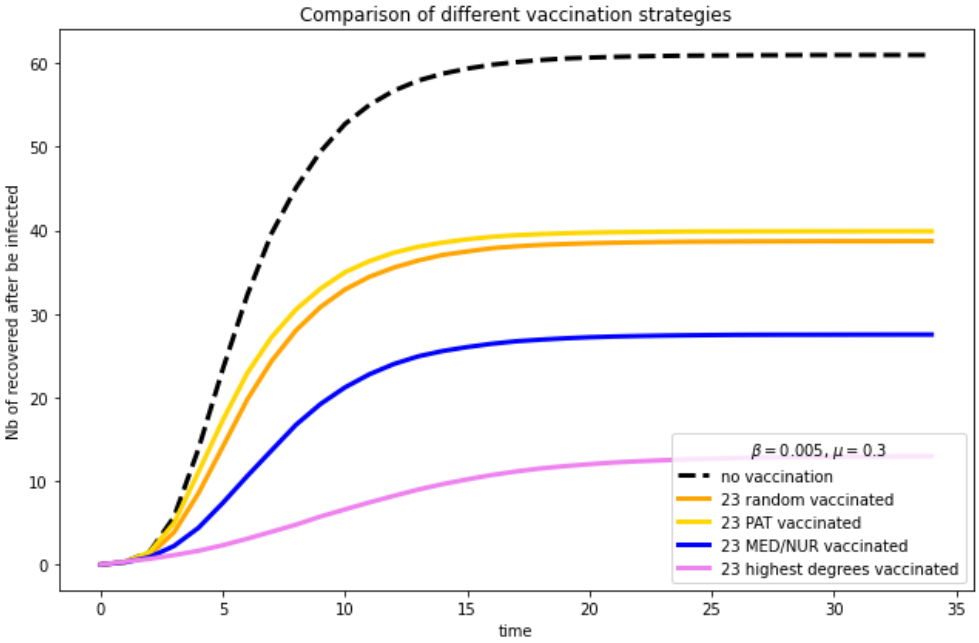
In [ ]:
```
### Challenge problem 6

# check time of execution
start_time = time.time()

# matrices to store results of simulations, one for each strategy
nsimulations = niter_per_seed*(nn-ndoses)
mat1 = np.zeros((nsimulations, tmax))
mat2 = np.zeros((nsimulations, tmax))
mat3 = np.zeros((nsimulations, tmax))
mat4 = np.zeros((nsimulations, tmax))

# define people to be vaccinated according to the 4 strategies
# ***** Fill here with your code!   *************************
vacc1 =
vacc2 =
vacc3 =
vacc4 =
notvacc1 =
notvacc2 =
notvacc3 =
notvacc4 =
# ************************************************************

c = 0   # simulation counter
# for each not vaccinated node as seed
for n in range(nn-ndoses):

    # iterating several times for each seed
    for i in range(niter_per_seed):

        # simulations
        t1, S1, I1, R1 = SIR(nodes, neighbours, weights, beta, mu, initial_infected=notvacc1[n],
                             initial_recovereds=vacc1)
        t2, S2, I2, R2 = SIR(nodes, neighbours, weights, beta, mu, initial_infected=notvacc2[n],
                             initial_recovereds=vacc2)
        t3, S3, I3, R3 = SIR(nodes, neighbours, weights, beta, mu, initial_infected=notvacc3[n],
                             initial_recovereds=vacc3)
        t4, S4, I4, R4 = SIR(nodes, neighbours, weights, beta, mu, initial_infected=notvacc4[n],
                             initial_recovereds=vacc4)

        # fix R
        R1 = fix_R(R1, tmax)
        R2 = fix_R(R2, tmax)
        R3 = fix_R(R3, tmax)
        R4 = fix_R(R4, tmax)

        # save results
        mat1[c,:] = R1
        mat2[c,:] = R2
        mat3[c,:] = R3
        mat4[c,:] = R4
        c += 1

print("--- %s seconds ---" % (time.time() - start_time))
```

Plotting the results

In [ ]:
```
fig = plt.figure(figsize=(9,6))

x = np.arange(0,tmax)

# epidemic without vaccination
plt.plot(x, np.mean(mat_novax, axis=0), color='k', ls='--', lw=3, label='no vaccination')

# different vaccination strategies
plt.plot(x, np.mean(mat1, axis=0)-ndoses, color='orange', lw=3, label='%d random vaccinated'%ndoses)
plt.plot(x, np.mean(mat2, axis=0)-ndoses, color=colors['PAT'], lw=3, label='%d PAT vaccinated'%ndoses)
plt.plot(x, np.mean(mat3, axis=0)-ndoses, color=colors['MED'], lw=3, label='%d MED/NUR vaccinated'%ndoses)
plt.plot(x, np.mean(mat4, axis=0)-ndoses, color='violet', lw=3, label='%d highest degrees vaccinated'%ndoses)

plt.xlabel('time')
plt.ylabel('Nb of recovered after be infected')
```

```
plt.legend(title=r'$\beta=$%4.3f, $\mu=$%2.1f'%(beta, mu), loc='lower right')
plt.title('Comparison of different vaccination strategies')

fig.tight_layout()
```

Problem 6 - output



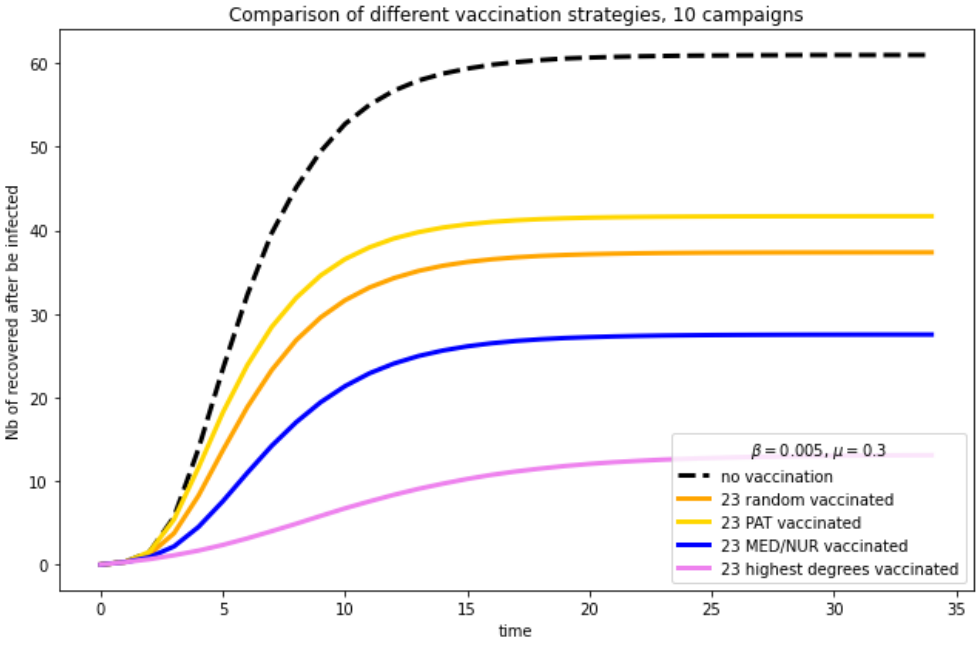
Challenge problem 7: simulate several vaccination campaigns

Repeat the simulations by averaging the results over 10 runs (10 different campaigns of vaccination). Again, we consider the same 4 vaccination strategies and peolpe to be vaccinated can change from a campaign to the next.

In [ ]:
```
ncampaigns = 10
```

In [ ]:
```
### Challenge problem 7
#   Fill here with your code!   ****************
```

Challenge problem 7 - output



In [ ]:

In [ ]:

In [ ]: