Politecnico di Torino

Corso di Laurea Magistrale in Data Science And Engineering

Network Dynamics and Learning

Homework III

Student	Email
Alejandra Solarte Uscategui	s321944@studenti.polito.it
Alejandro Mesa Gomez	s306638@studenti.polito.it
Anderson Estiver Alvarez Giraldo	s310468@studenti.polito.it
Juliana Cortes Mendivil	s314545@studenti.polito.it
Mateo Rivera Monsalve	s320923@studenti.polito.it

Contents

0.1 Individual contributions

Exercise	Student
2	Alejandra Solarte
1	Alejandro Mesa
2	Anderson Alvarez
1	Juliana Cortes
1	Mateo Rivera

1 1. Influenza H1N1 2009 Pandemic in Sweden

During the fall of 2009 there was a large pandemic of the H1N1-virus, commonly known as the swine-flu. During this pandemic it is estimated that about 1.5 million people in Sweden were infected. As an attempt to stop the pandemic and reduce excess mortality the government issued a vaccination program beginning in week 40 of 2009. During the weeks that followed they vaccinated more than 60% of the Swedish population.

In this homework, you will simulate the pandemic with the goal of learning the network-structure characteristics and disease-dynamics parameters of the pandemic in Sweden 2009. This task will be divided into 4 parts where the focus of each part is to:

1. Get started and learn how to:
   `a. Simulate a pandemic on a known graph`
   `b. Generate a random graph`
2. Simulate the disease propagation on a random graph without vaccination
3. Simulate disease propagation on a random graph with vaccination
4. Estimate the network-structure characteristics and disease-dynamics parameters for the pandemic in Sweden during the fall of 2009

All numbers regarding the H1N1 pandemic in Sweden during the fall of 2009 have been taken from the a report by the Swedish Civil Contingencies Agency (Myndigheten for samhallsskydd och beredskap, MSB) and the Swedish Institute for Communicable Disease Control (Smittskyddsinstitutet, SMI).

1.1 Preliminary parts

As a warm-up exercise we will start off by doing two preliminary parts. The first one will involve simulating an epidemic on a given graph, while the second part will be to generate a random graph with preferential attachment.

1.1.1 Epidemic on a known graph

In this part you will simulate an epidemic on a symmetric k-regular undirected graph with node set $V = \{1, 2, ..., n\}$ where every node is directly connected to the $k = 6$ nodes whose index is closest to their own modulo n. See Figure 1 for an example with 8 nodes. The graph that you will simulate the epidemic on will however contain $n = 500$ nodes.

Figure 1: Symmetric k-regular graph.

The disease propagation model that you will use to simulate the epidemic is a discrete-time simplified version of the SIR epidemic model. At any time $t = 0, 1, \ldots$ nodes are in state $X_i(t) \in \{S, I, R\}$, where S is susceptible, I is infected and R is recovered. Let $\beta \in [0, 1]$ be the probability that the infection is spread from an infected individual to a susceptible one (given that they are connected by a link) during one time step. Assuming that a susceptible node i has m infected neighbors, this means that the probability that individual i does not get infected by any of the neighbors during one time step is $(1-\beta)^m$. Thus, the probability that individual i becomes infected by any of its neighbors is $1-(1-\beta)^m$. Furthermore, let $\rho \in [0, 1]$ be the probability that an infected individual will recover during one time step. The epidemic is driven by the following transition probabilities

$$\mathbb{P}\left(X_i(t+1) = I \mid X_i(t) = S, \sum_{j\in V} W_{ij}\delta^I_{X_j(t)} = m\right) = 1-(1-\beta)^m$$

$$\mathbb{P}\left(X_i(t+1) = R \mid X_i(t) = I\right) = \rho$$

where $\sum_{j\in V} W_{ij}\delta^I_{X_j(t)}$ is the number of infected neighbors for node i.

Problem 1.1: You should simulate an epidemic on a symmetric $k-$regular graph $G = (V, E)$ with $|V| = 500$ nodes and $k = 6$. See Figure 1 for an example with $n = 8$ nodes and $k = 6$. Let $\beta = 0.25$ and $\rho = 0.6$. With one week being one unit of time, simulate the epidemic for 15 weeks. You can choose an initial configuration with 10 infected nodes selected at a random from the node set V, or make a different choice of initial configuration (In the latter case, please briefly discuss your motivation).

Do this $N = 100$ times and plot the following: * The average number of newly infected individuals each week. In other words, you should plot how many people *become* infected each week (On the average). * The average total number of susceptible, infected, and recovered individuals at each week. In other words, you should plot how many individuals *in total are* susceptible/infected/recovered at each week (On the average).

Hint: Since we use a fairly large amount of nodes for this simulation it is good idea to use *sparse matrices* for this and the following problems.

[1]:
```
import networkx as nx
import matplotlib.pyplot as plt
import seaborn as sns
import random
import numpy as np
import progressbar
import time
from itertools import product

%matplotlib inline
options = {
    'node_size': 500,
    'width': 1, # width of the edges,
    'node_color': '#ffffff',
    'node_shape': 'o',
    'edge_color': 'black',
    'font_weight':'normal'
}
sns.set()
```

[2]:
```
def custom_circulant_graph(n, k):
    #if condition returns False, AssertionError is raised:
    assert k % 2 == 0, "k must be even, i.e. k  {2i | i  N}"

    # Generate the circulant graph
    distances = [i for i in range(1, k // 2 + 1)]  # Set of distances (1, 2, ...
 ↪, k/2)
    return nx.circulant_graph(n, distances)
```

[3]:
```
# Set parameters
n = 500
k = 6
beta = 0.25
rho = 0.6
week_to_simulate = 15
initial_infected_nodes = np.random.choice(range(n), 10)

# Create the graph
G = custom_circulant_graph(n, k)

# Set initial configuration of the node state
color_map = []
for i in range(n):
    if i in initial_infected_nodes:
        G.nodes[i]['state'] = 'I'
        color_map.append('red')
    else:
        G.nodes[i]['state'] = 'S'
        color_map.append('lightblue')
```

[4]:
```
# From G, get the infected neighbors of the node i
def get_infected_neighbors(G, i):
    infected_neighbors = [neighbor for neighbor in G.neighbors(i) if G.
 ↪nodes[neighbor]['state'] == 'I']
    return infected_neighbors
```

[5]:
```
# Computes the probability that individual i becomes infected by any of its␣
 ↪neighbors or
# the probability that an infected individual will recover during one time step.
def P(G, i, beta = None, rho = None):
        return 1- (1 - beta) ** len(get_infected_neighbors(G, i)) if beta is not␣
 ↪None else rho
```

```
[6]: def EpidemicSimulation(G, beta, rho, time_limit):
         states = ['S', 'I', 'R']
         # We do not want to modify the original graph
         G = G.copy()

         # nodes_infected saves is a matrix of G.number_of_nodes() X time_limit
         # each row is associated to a node and each column to the state of the node
         #nodes_infected = np.zeros((G.number_of_nodes(), 1))
         nodes_infected = np.reshape([states.index(G.nodes[i]['state']) for i in G.
      ↪nodes], (-1, 1))

         # New infected
         new_nodes_infected = [nodes_infected.sum()]

         # Warning: G.nodes[i]['state'] saves the current state, not historical data
         for t in range(1, time_limit + 1):
             # We will be modifying the states of G, so we need a copy of the␣
      ↪iteration t - 1
             G_copy = G.copy()
             current_nodes_infected = np.copy(nodes_infected[:, -1])
             new_nodes_infected.append(0)

             # Let's make some transitions depending on the probabilities
             for i in G_copy.nodes:
                 new_state = G_copy.nodes[i]['state']
                 if new_state == 'S':
                     # 1 if i get infected, 0 otherwise
                     i_is_infected = np.random.binomial(1, P(G_copy, i, beta = beta))

                     if i_is_infected:
                         new_state = 'I'
                         new_nodes_infected[-1] += 1

                 elif new_state == 'I':
                     # 1 if i get recovered, 0 otherwise
                     i_is_recovered = np.random.binomial(1, P(G_copy, i, rho = rho))

                     if i_is_recovered:
                         new_state = 'R'

                 # Let's update the state of the node
                 G.nodes[i]['state'] = new_state
                 current_nodes_infected[i] = states.index(new_state)

             nodes_infected = np.append(nodes_infected, current_nodes_infected.
      ↪reshape(-1, 1), axis=1)
```

```
    return nodes_infected, new_nodes_infected
```

[7]:
```
def Simulations(G, beta, rho, time_limit = 15, n_simulations = 100,␣
 ↪use_progressbar = True):
    simulations = []
    # Create a progress bar
    if use_progressbar:
        bar = progressbar.ProgressBar(max_value=n_simulations)
    for s in range(n_simulations):
        simulations.append(EpidemicSimulation(G, beta, rho, time_limit))
        if use_progressbar:
            bar.update(s+1)
    if use_progressbar:
        bar.finish()
    return simulations
```

[8]:
```
# Extract and calculate the total number of S, I, and R nodes for each week
def calculate_average_states(simulations):
    time_limit = len(simulations[0][1])
    total_states = np.zeros((3, time_limit))  # For S, I, R
    for simulation in simulations: # Simulation has a tuple of matrix with 500␣
 ↪rows and 15 columns and a vector with the new infected
        states = simulation[0]  # Get the state history for each node
        for t in range(states.shape[1]): # states.shape[1] returns the number of␣
 ↪weeks of the simulation
            # summing the number of S,I,R by each week
            total_states[0, t] += np.sum(states[:, t] == 0)  # Susceptible (S)
            total_states[1, t] += np.sum(states[:, t] == 1)  # Infected (I)
            total_states[2, t] += np.sum(states[:, t] == 2)  # Recovered (R)

    # Calculate the average across all simulations
    average_states = total_states / len(simulations) # we divide by the number␣
 ↪of simulations length (simulations)
    return average_states
```

[9]:
```
def simplified_plot(data, title='', xlabel='', ylabel='', color='b', label=''):
    plt.plot(data, color=color, label=label)
    plt.title(title)
    plt.xlabel(xlabel)
    plt.ylabel(ylabel)
```

[10]:
```
nodes_infected, new_nodes_infected = EpidemicSimulation(G, beta, rho,␣
 ↪week_to_simulate)
```

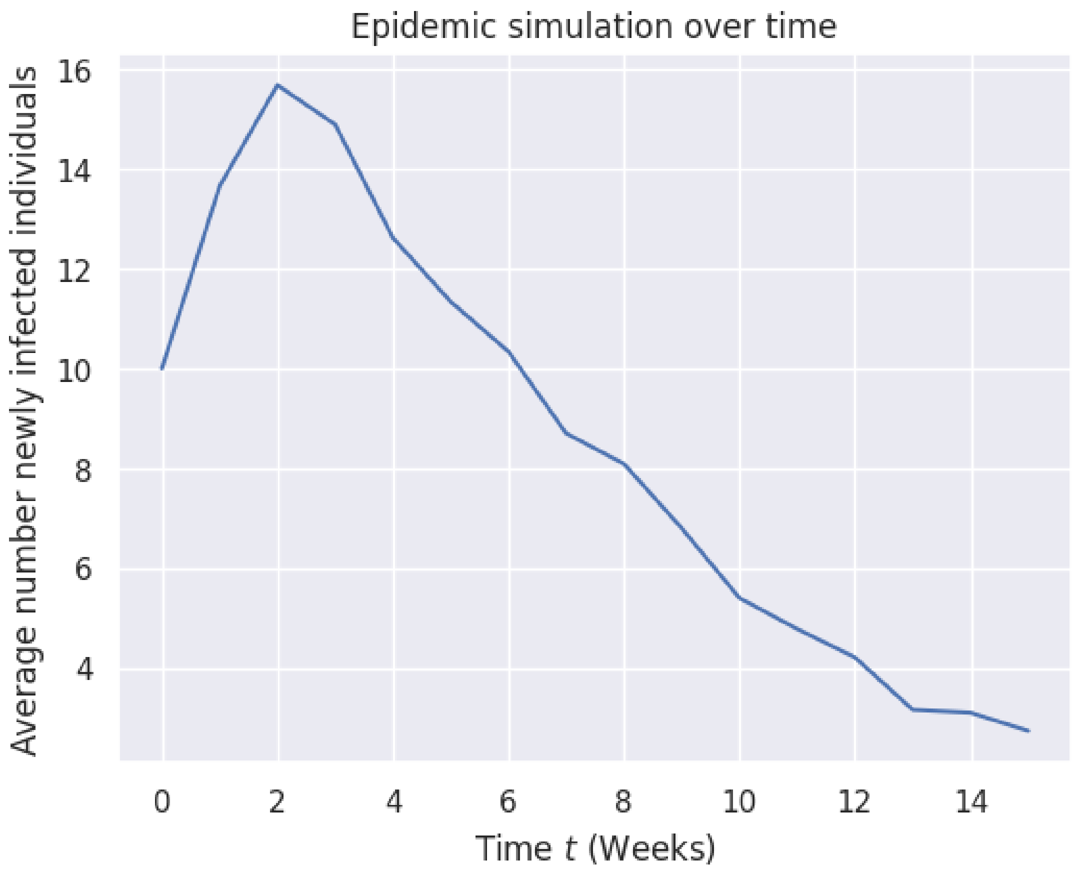
[11]:
```
S = Simulations(G, beta, rho, week_to_simulate)
```

```
100% (100 of 100)
|######################| Elapsed Time: 0:00:28 Time:  0:00:28
```

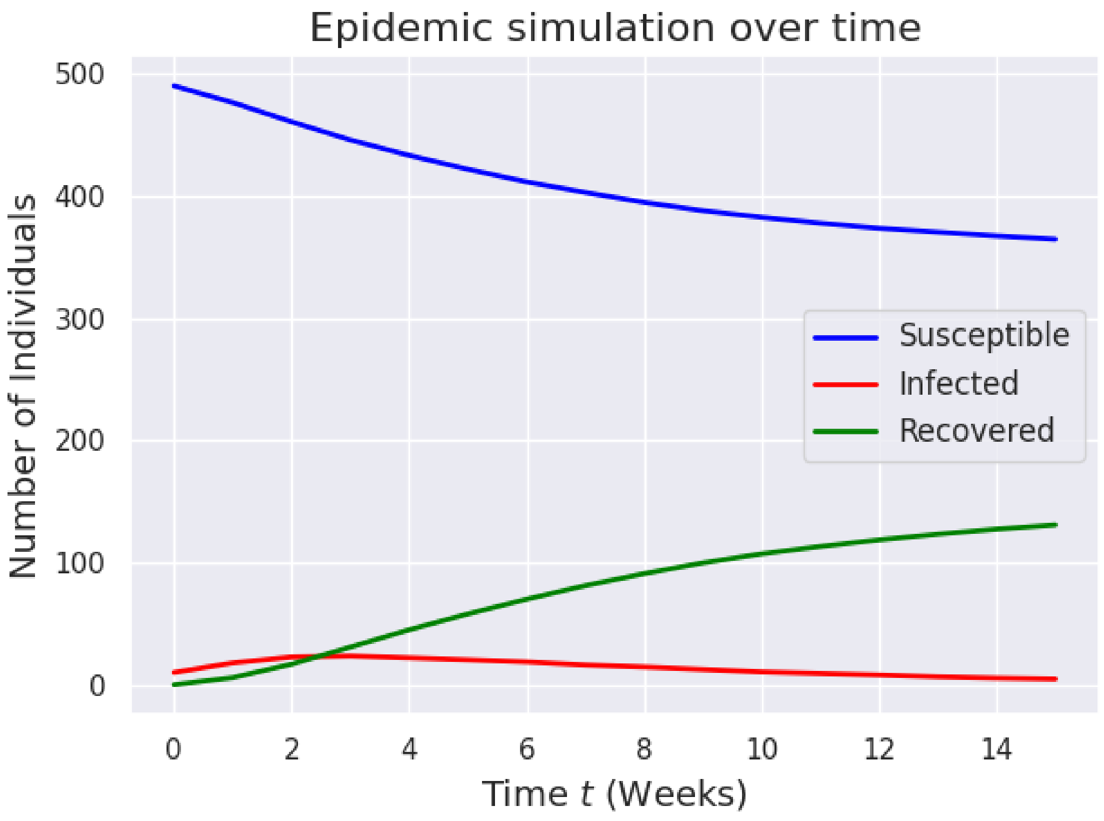
[12]:
```
average_new_nodes_infected = np.mean([simulation[1] for simulation in S], axis=0)
average_new_nodes_infected
```

[12]:
```
array([10.  , 13.65, 15.68, 14.89, 12.62, 11.34, 10.34,  8.7 ,  8.09,
        6.8 ,  5.4 ,  4.78,  4.21,  3.16,  3.1 ,  2.74])
```

[13]:
```
simplified_plot(average_new_nodes_infected,
                title="Epidemic simulation over time",
                xlabel='Time $t$ (Weeks)',
                ylabel='Average number newly infected individuals'
                )
plt.show()
```



[14]:
```
# Calculate the average states for S, I, R
average_states = calculate_average_states(S)

# Plotting the results

plt.plot(average_states[0], label='Susceptible', color='blue', linewidth=2)
plt.plot(average_states[1], label='Infected', color='red', linewidth=2)
plt.plot(average_states[2], label='Recovered', color='green', linewidth=2)

plt.title("Epidemic simulation over time", fontsize=16)
plt.xlabel("Time $t$ (Weeks)", fontsize=14)
plt.ylabel("Number of Individuals", fontsize=14)
plt.legend(fontsize=12)
plt.grid(True)
plt.tight_layout()

plt.show()
```



1.1.2 Generate a random graph

In this part you will generate a random graph according to the preferential attachment model. The goal is to have a randomly generated graph with average degree close to k. The idea is the following: at time t = 1 we start with an initial graph G1, that is complete with k +1 nodes. Then at every time t >= 2, create a new graph $G_t = (V_t; E_t)$ by adding a new node to G_{t-1} and connect it to some of the existing nodes V_{t-1} of G_{t-1} chosen according to some stochastic rule.

The rule by which the new node add links to the nodes of G_{t-1} is preferential attachment. This means that at every time-step t >= 2, every new node added at time t will have a degree $w_t(t) =$

$c = k/2$. Hence, it should add c undirected links to the existing graph G_{t-1}. It decides which of the nodes in V_{t-1} it should connect to based on some probability that is proportional to the current degree of the node it is connecting to. In other words, if we denote the new node n_t, the probability that there will be a link between node n_t and node i ϵ V_{t-1} is:

where W(t) is the adjacency matrix for the next time-step t and $w_i(t-1)$ is the degree of node i prior to adding the new node. Some care should be taken here so that you do not add multiple links to the same node.

You should also note that if k is odd, it is a bit trickier to generate the random graph such that the average degree will be k. If k happens to be an odd number, then c = k/2 will not be an integer. However, it is still possible to achieve an average degree of k when you add a large number of nodes. This can be done by alternating between adding [k/2] and [k/2] links when adding a new node to the graph.

Problem 1.2: Your goal is to, by using preferential attachment, generate a random graph of a large size (at least 900 nodes) with average degree k ϵ Z^+. Let the initial graph $G = (V_1, E_1)$ be a complete graph with $|V_1| = k_0 = k + 1$ nodes.

Note that the goal here is to implement a fairly general algorithm where it is very easy to change the average degree. It should be possible to change the average degree by only changing the value of k in your algorithm. This algorithm will then be used in Section 1.4.

```
[15]: # Function to generate a preferential attachment graph
      def preferential_attachment_graph(n, k):
          assert n > k + 1, "Number of nodes (n) must be greater than k + 1."

          # Initial complete graph with k + 1 nodes
          G = nx.complete_graph(k + 1)
          add_edges = [k // 2, (k + 1) // 2]

          # At every time t>=2 create a new graph by adding a new node
          for t in range(k + 2, n):
              G.add_node( t-1) #t-1 to prevent a jump in node name enumeration␣
       ↪n={4,5,6,___,8,9}

              # Calculate the degree of each existing node, serve as wi(t-1)
              degrees = dict(G.degree())

              # Select nodes to connect based on preferential attachment
              targets = set()

              # Alternate edge count for odd k
              num_edges = add_edges[(t - (k + 2)) % 2] if k % 2 else k // 2  # Fix to␣
       ↪add k/2 edges per new node

              # Loop to ensure no redundant edges to the same target
              while len(targets) < num_edges:
```

```
            # Probability of connecting to an existing node is proportional to␣
↪its current degree wi(t-1)
            chosen = random.choices(
                population=list(degrees.keys()),
                weights=list(degrees.values()),
                k=1
            )[0]
            if chosen != (t - 1):  # Avoid self-loops
                targets.add(chosen)

        # Add edges between the new node and selected targets
        for target in targets:
            G.add_edge(t, target)

    return G
```

[16]:
```
n = 900
k = 6
G = preferential_attachment_graph(n, k)

# Check average degree
average_degree = sum(dict(G.degree()).values()) / G.number_of_nodes()
print(f"Generated graph with average degree: {average_degree}")
```

```
Generated graph with average degree: 5.993333333333333
```

1.2 Simulate a pandemic without vaccination

In this part you will be using the graph generated in Section 1.1.2 and then simulate an epidemic on it. The disease propagation model is again the discrete-time version of the SIR epidemic model used in Section 1.1.1.

Problem 2: Using the methods developed in Section 1.1, generate a preferential attachment random graph $G = (V, E)$ with $|V| = 500$ nodes. The average degree should be $k = 6$. Let $\beta = 0.25$ and $\rho = 0.6$. With one week being one unit of time, simulate the epidemic for 15 weeks. You can choose an initial configuration with 10 infected nodes selected at a random from the node set V, or make a different choice of initial configuration (In the latter case, please briefly discuss your motivation).

Do this $N = 100$ times and plot the following:

- The average number of newly infected individuals each week. In other words, you should plot how many people *become* infected each week.
- The average total number of susceptible, infected, and recovered individuals at each week. In other words, you should plot how many individuals *in total are* susceptible/infected/recovered at each week.

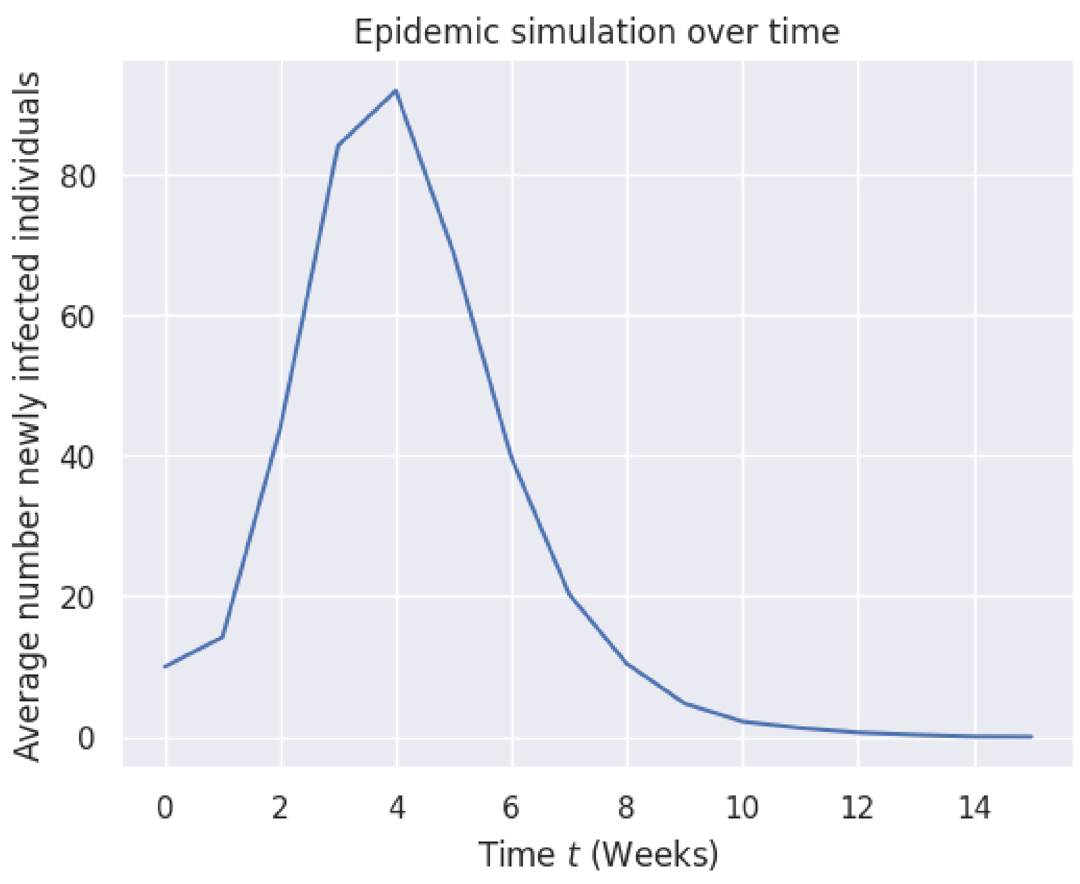
[17]:
```
# Parameters
n = 500
k = 6
```

```
beta = 0.25
rho = 0.6
week_to_simulate = 15
n_simulations = 100
initial_infected_nodes = np.random.choice(range(n), 10)
```

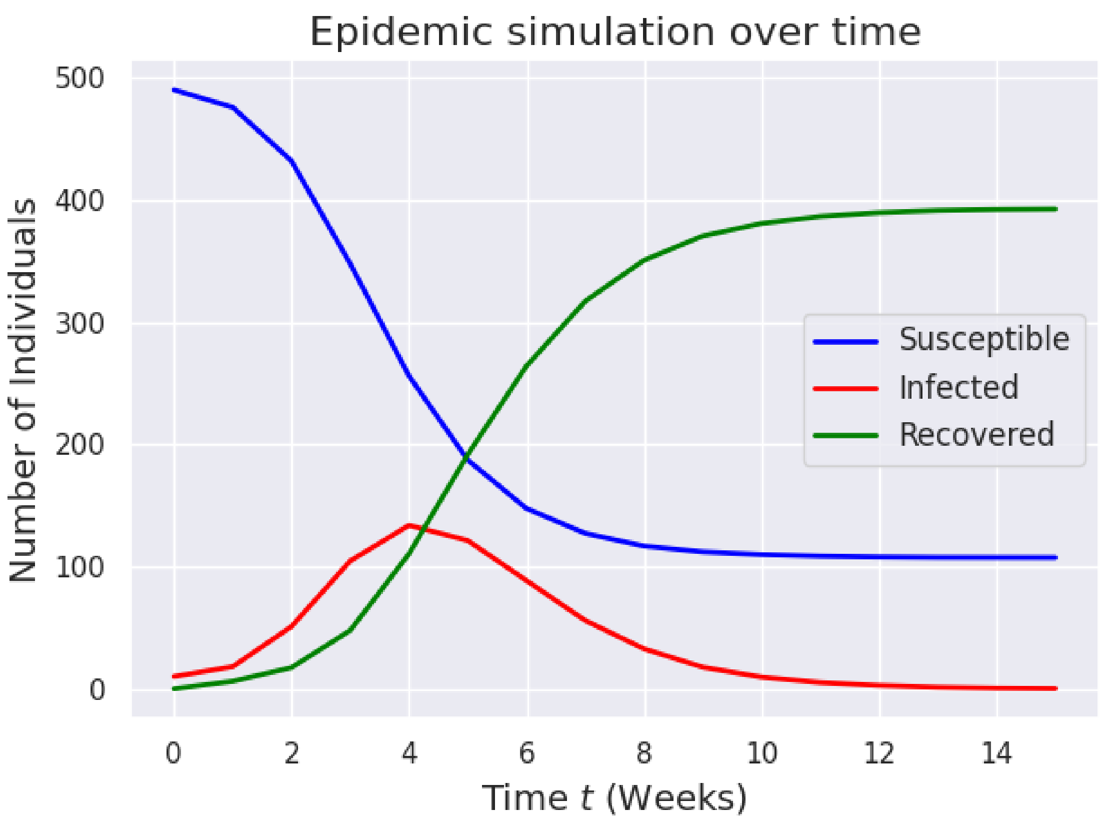
[18]:
```
# Generate graph
G = preferential_attachment_graph(n, k)

# Set initial configuration of the node state
color_map = []
for i in range(n):
    if i in initial_infected_nodes:
        G.nodes[i]['state'] = 'I'
        color_map.append('red')
    else:
        G.nodes[i]['state'] = 'S'
        color_map.append('lightblue')

# Run simulations
S = Simulations(G, beta, rho, time_limit = week_to_simulate,␣
 ↪n_simulations=n_simulations)

average_degree = sum(dict(G.degree()).values()) / G.number_of_nodes()
print(f"Generated graph with average degree: {average_degree}")

# Calculate the average number of new infected nodes
average_new_nodes_infected = np.mean([simulation[1] for simulation in S], axis=0)
```

```
100% (100 of 100)
|######################| Elapsed Time: 0:00:16 Time:  0:00:16

Generated graph with average degree: 5.988
```

[19]:
```
# Plotting average number of newly infected individuals each week.
simplified_plot(average_new_nodes_infected,
                title="Epidemic simulation over time",
                xlabel='Time $t$ (Weeks)',
                ylabel='Average number newly infected individuals'
                )
plt.show()
```



[20]:
```
#Plotting average total number of susceptible, infected, and recovered␣
↪individuals at each week
# Calculate the average states for S, I, R
average_states = calculate_average_states(S)

# Plotting the results

plt.plot(average_states[0], label='Susceptible', color='blue', linewidth=2)
plt.plot(average_states[1], label='Infected', color='red', linewidth=2)
plt.plot(average_states[2], label='Recovered', color='green', linewidth=2)

plt.title("Epidemic simulation over time", fontsize=16)
plt.xlabel("Time $t$ (Weeks)", fontsize=14)
plt.ylabel("Number of Individuals", fontsize=14)
plt.legend(fontsize=12)
plt.grid(True)
plt.tight_layout()

plt.show()
```



1.3 Simulate a pandemic with vaccination

In this part you will essentially do the same thing as before, but you will also try to take some action to slow down the epidemic. This is normally done using vaccination. Therefore, during each week, some parts of the population will receive vaccination. Once a person is vaccinated it cannot be infected. Furthermore, the vaccination is assumed to take effect immediately once given, i.e. if person a is vaccinated in week 10, then a is no longer susceptible during that week, and can therefore not infect any other individual.

You should once again simulate the disease propagation for 15 weeks, but you should now also distribute vaccination to the population. This should be done such that the total fraction of population that has received vaccination by each week is according to:

Vacc(t) = [0; 5; 15; 25; 35; 45; 55; 60; 60; 60; 60; 60; 60; 60; 60]:

Vacc(t) should be interpreted as: 55% of the population has received vaccination by week 7, and 5% received vaccination during week 7.

To simulate the actual vaccination you should, at the beginning of each week,find the correct number of individuals to vaccinate according to Vacc(t). You should then find individuals to vaccinate. These individuals should be selected uniformly at random from the population that has not yet received vaccination. This means that an infected individual might receive vaccination as well. The reason behind this is that some people were not able to tell whether they had the H1N1-virus or just the common cold. If an infected individual becomes vaccinated it is assumed that she will not be able to infect another individual. In other words, we assume that regardless of the state of an individual prior to the vaccination, once vaccinated the individual will not be able to become infected nor infect any other individuals.

Problem 3: Using the methods developed in Section, generate a random graph $G = (V, E)$ with $|V| = 500$ nodes. The average degree should be $k = 6$. Let $\beta = 0.25$ and $\rho = 0.6$. With one week being one unit of time, simulate the epidemic for 15 weeks. using the vaccination scheme Vacc(t) above.You can choose an initial configuration with 10 infected nodes selected at a random from the node set V, or make a different choice of initial configuration (In the latter case, please briefly discuss your motivation).

Do this $N = 100$ times and plot the following:

- The average number of newly infected individuals each week. In other words, you should plot how many people *become* infected each week.
- The average total number of susceptible, infected, recovered and vaccinated individuals at each week.

[21]:
```
# Getting the infected and not vaccinated neighbors of a node
def get_infected_neighbors(G, i):
    """
    Function to get infected  and not vaccinated neighbors of a node
    """
    return list(filter(lambda neighbor: G.nodes[neighbor]['state'] == 'I' and G.
↪nodes[neighbor]['vaccinated'] == 0, G.neighbors(i)))

def sample_individuals(size,total):
    numbers = list(range(total))
    random.shuffle(numbers) #random shuffle the list

    while numbers:  # while theres available numbers
        yield numbers[:size]  # return a sample
        del numbers[:size]   # delete the sampled numbers

def vaccinate_individuals(G, sampled_nodes):
    """
    Function to set a node as vaccinated
    """
    for node in sampled_nodes:
        G.nodes[node]['vaccinated'] = 1
```

In order to propperly simulate how the vaccination prevents people to get infected, first we need to build a function to sample which individuals shall recieve the vaccine.

`sample_individuals`, shuffles a list of the same size as nodes there are in the graph in order to get the number of people to sample. After that, `Vaccinate_individuals` can be called to update the vaccinated state of each sampled individual in the graph.

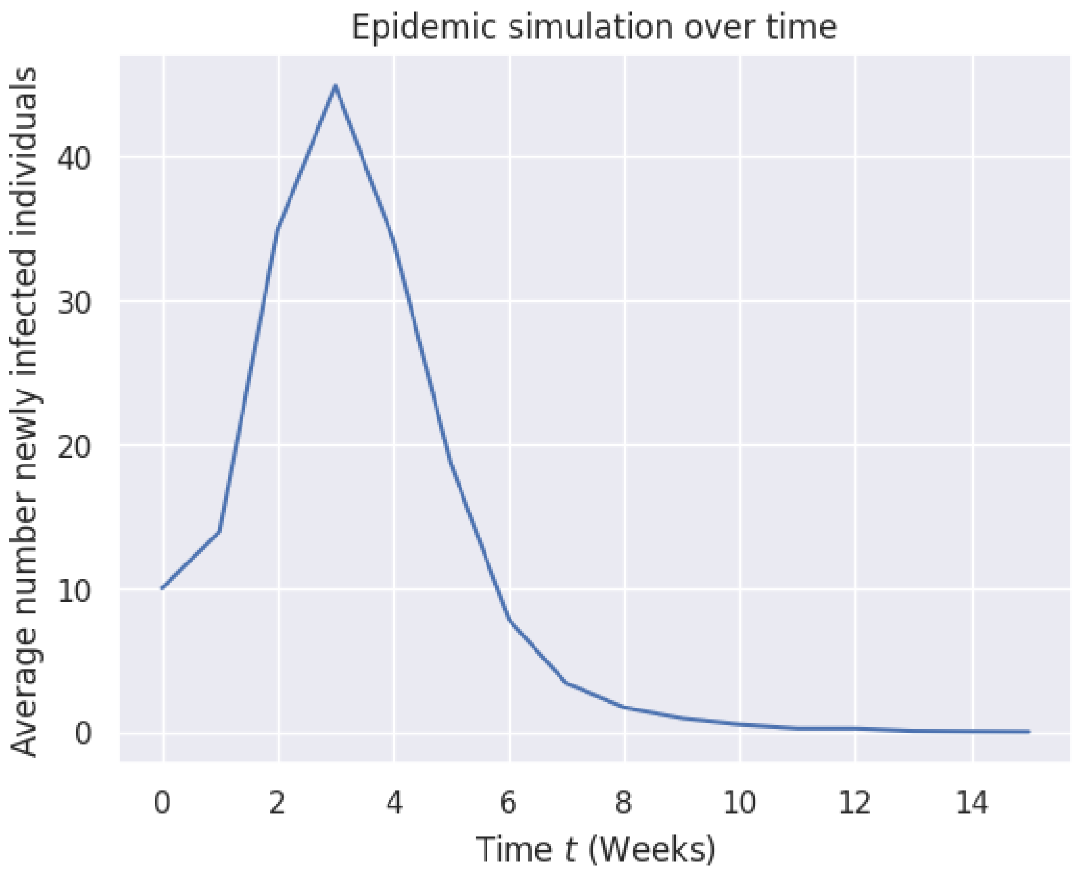
The previous `EpidemicSimulation` function is modified in order to recive the percentage of vaccination given in the problem sentence, if a susceptible person gets vaccinates, it passed inmediatelly to recovered, and `get_infected_neighbors` filters the vaccinated people in order to assure they will not be contagious if infected.

```
[22]: # Function to calculate infection/recovery probability
def P(G, i, beta=None, rho=None):
    infected_neighbors = get_infected_neighbors(G, i)
    if beta is not None:
        return 1 - (1 - beta) ** len(infected_neighbors)  # Infection probability
    elif rho is not None:
        return rho  # Recovery probability
    return 0

def EpidemicSimulation(G, beta, rho, time_limit, vacc_t):


    states = ['S', 'I', 'R']

    # We do not want to modify the original graph
    G = G.copy()

    ids = list(range(len(G.nodes)))
    random.shuffle(ids) #random shuffle the list

    new_vacc_nodes=[0]
    # nodes_infected saves is a matrix of G.number_of_nodes() X time_limit
    # each row is associated to a node and each column to the state of the node
    #nodes_infected = np.zeros((G.number_of_nodes(), 1))

    nodes_infected = np.reshape([states.index(G.nodes[i]['state']) for i in G.
 ↪nodes], (-1, 1))


    # New infected
    new_nodes_infected = [nodes_infected.sum()]

    # Warning: G.nodes[i]['state'] saves the current state, not historical data
    for t in range(1, time_limit + 1):

        current_nodes_infected = np.copy(nodes_infected[:, -1])
        new_nodes_infected.append(0)
        vacc_sample = int(np.round((vacc_t[t]-vacc_t[t-1])*(len(G.nodes)/
 ↪100),0))  #percentage of people to vaccinate

        ids_to_vacc = ids[:vacc_sample]
        del ids[:vacc_sample]
        new_vacc_nodes.append(vacc_t[t]*5)
        vaccinate_individuals(G, ids_to_vacc)
        # We will be modifying the states of G, so we need a copy of the␣
↪iteration t - 1
        G_copy = G.copy()
        # Let's make some transitions depending on the probabilities
        for i in G_copy.nodes:
            new_state = G_copy.nodes[i]['state']
            vaccinated = G_copy.nodes[i]['vaccinated']
            if new_state == 'S':
                # 1 if i got infected, 0 otherwise
                if not vaccinated:
                    i_is_infected = P(G_copy, i, beta = beta)> np.random.rand()
                else:
                    i_is_infected = 0

                if i_is_infected:
                    new_state = 'I'
                    new_nodes_infected[-1] += 1
                elif vaccinated:
                    new_state = 'R'

            elif new_state == 'I':
                # 1 if i get recovered, 0 otherwise
                i_is_recovered =  P(G_copy, i, rho = rho) > np.random.rand()

                if i_is_recovered:
                    new_state = 'R'

            # Let's update the state of the node
            G.nodes[i]['state'] = new_state
            current_nodes_infected[i] = states.index(new_state)

        nodes_infected = np.append(nodes_infected, current_nodes_infected.
↪reshape(-1, 1), axis=1)

    return nodes_infected, new_nodes_infected, new_vacc_nodes
# Function for running multiple simulations
def Simulations(G, beta, rho, time_limit = 15, n_simulations = 100,␣
↪use_progressbar=True, vacc_t=None):
    simulations = []
    if use_progressbar:
        bar = progressbar.ProgressBar(max_value=n_simulations)

    for s in range(n_simulations):
        simulations.append(EpidemicSimulation(G, beta, rho, time_limit, vacc_t))
        if use_progressbar:
            bar.update(s+1)
    if use_progressbar:
        bar.finish()

    return simulations
```

[23]:
```
# Generate graph
G = preferential_attachment_graph(n, k)

vacc_t = [0,5,15,25,35,45,55,60,60,60,60,60,60,60,60,60] #total
# Set initial state for nodes
for node in initial_infected_nodes:
    G.nodes[node]['state'] = 'I'
    G.nodes[node]['vaccinated'] = 0 #At week 0 nobody is vaccinated

for node in G.nodes():
    if node not in initial_infected_nodes:
        G.nodes[node]['state'] = 'S'  # Default state for non-infected nodes
    G.nodes[node]['vaccinated'] = 0 #At week 0 nobody is vaccinated

# Run simulations
S = Simulations(G, beta, rho, time_limit = week_to_simulate,␣
 ↪n_simulations=n_simulations, vacc_t= vacc_t)

# Calculate the average number of new infected nodes
average_new_nodes_infected = np.mean([simulation[1] for simulation in S], axis=0)
```

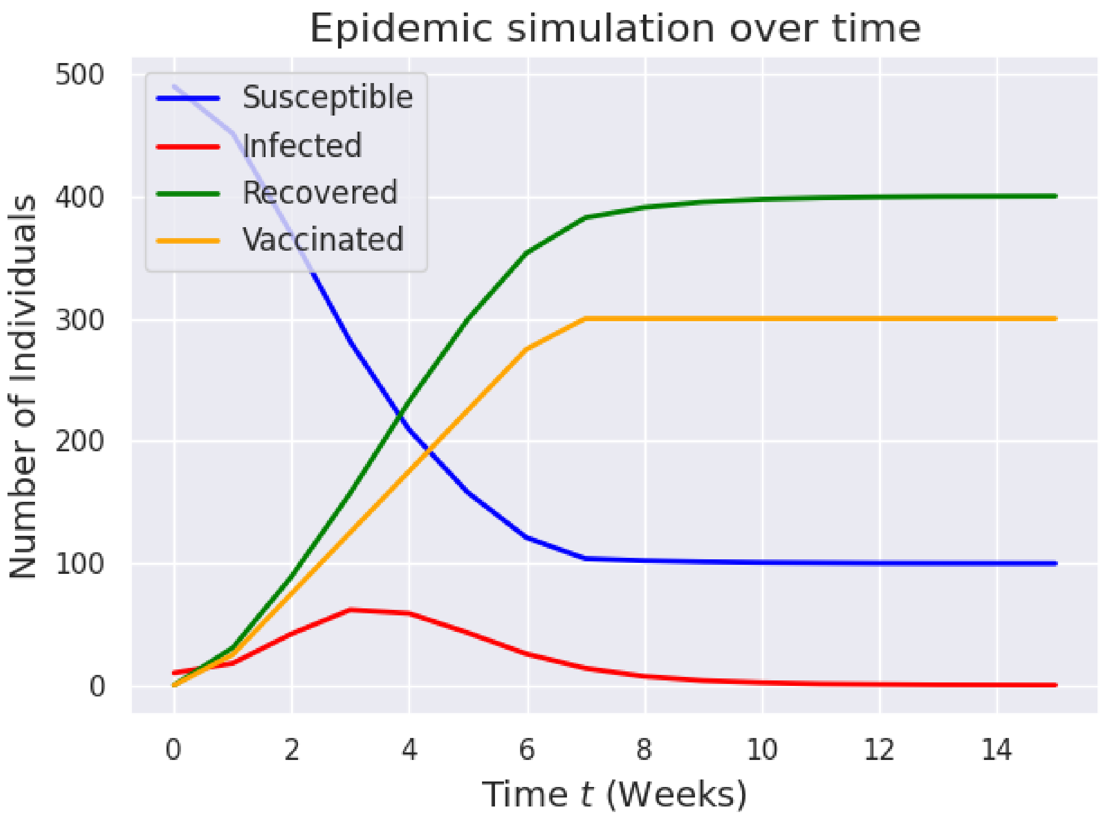
```
100% (100 of 100)
|######################| Elapsed Time: 0:00:14 Time:  0:00:14
```

[24]:
```
# Plotting average number of newly infected individuals each week.
simplified_plot(average_new_nodes_infected,
                title="Epidemic simulation over time",
                xlabel='Time $t$ (Weeks)',
                ylabel='Average number newly infected individuals'
                )
plt.show()
```



Comparing the results to the previous excercise, the infeccion peak gets reduced by half, however the behaviour of the curve remains the same.

[25]:
```
#Plotting average total number of susceptible, infected, and recovered␣
↪individuals at each week
# Calculate the average states for S, I, R
average_states = calculate_average_states(S)
average_new_vaccinated_nodes = np.mean([simulation[2] for simulation in S],␣
↪axis=0)

# Plotting the results

plt.plot(average_states[0], label='Susceptible', color='blue', linewidth=2)
plt.plot(average_states[1], label='Infected', color='red', linewidth=2)
plt.plot(average_states[2], label='Recovered', color='green', linewidth=2)
plt.plot(average_new_vaccinated_nodes, label='Vaccinated', color='orange',␣
↪linewidth=2)

plt.title("Epidemic simulation over time", fontsize=16)
plt.xlabel("Time $t$ (Weeks)", fontsize=14)
plt.ylabel("Number of Individuals", fontsize=14)
plt.legend(fontsize=12)
plt.grid(True)
plt.tight_layout()

plt.show()
```



In this graphic we can clearly see the contribution of vaccination, the recovered population grows rapidly, almost as an exponential and the susceptible population stabilizes in 100, this come from the fact that the infected population decreases so rapidly that a lot of them does not get in contact with a patient.

1.4 The H1N1 pandemic in Sweden 2009

In this part you will use all the previous parts in order to estimate the social structure of the Swedish population and the disease-spread parameters during the H1N1 pandemic. As mentioned before, during the fall of 2009 about 1.5 million people out of a total population of 9 million were infected with H1N1, and about 60% of the We will simulate the pandemic between week 42, 2009 and week 5, 2010. During these weeks, the fraction of population that had received vaccination was:

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

In order to not spend too much time running simulations, we will scale down the population of Sweden by a factor of 104 . This means that the population during the simulation will be $n = |\mathcal{V}| = 934$. For the scaled version, the number of newly infected individuals each week in the period between week 42, 2009 and week 5, 2010 was:

$$I0(t) = [1, 1, 3, 5, 9, 17, 32, 32, 17, 5, 2, 1, 0, 0, 0, 0]$$

where the first entry is the initial number of infected. The following algorithm will do a gradient-based search over the parameter space of k, β, and ρ in order to find the set of parameters that best matches the real pandemic.

Problem 4: Using the algorithm above, estimate the average degree k and the disease-spread parameters β and ρ for the pandemic. Once you have found the best estimate, report what parameters you got. You should also show the following plots:

- The average number of newly infected individuals each week according to the model (with your best parameters) compared to the true value of newly infected individuals each week.
- The total number of susceptible, infected, recovered and vaccinated individuals at each week according to the model.

[26]:
```
rng = np.random.default_rng(1234567890)
```

The cost function to minimize will be the mean root square error:

$$\text{RMSE} = \sqrt{\frac{1}{15}\sum_{t=1}^{15}(I(t) - I_0(t))^2}$$

where I(t) is the average number of newly infected individuals each week in the simulation and I0 (t) is the true value of newly infected individuals each week.

[27]:
```
def MRSE(I):
    I = np.array(I)
    I_0 = np.array([1, 1, 3, 5, 9, 17, 32, 32, 17, 5, 2, 1, 0, 0, 0, 0],␣
 ↪dtype=float)
    return np.sqrt(np.mean((I - I_0) ** 2))
def get_graph(n,k):
    G = preferential_attachment_graph(n, k)
    # Set initial state for nodes
    for node in initial_infected_nodes:
        G.nodes[node]['state'] = 'I'
        G.nodes[node]['vaccinated'] = 0 #At week 0 nobody is vaccinated

    for node in G.nodes():
        if node not in initial_infected_nodes:
            G.nodes[node]['state'] = 'S'  # Default state for non-infected nodes
        G.nodes[node]['vaccinated'] = 0 #At week 0 nobody is vaccinated
    return G
```

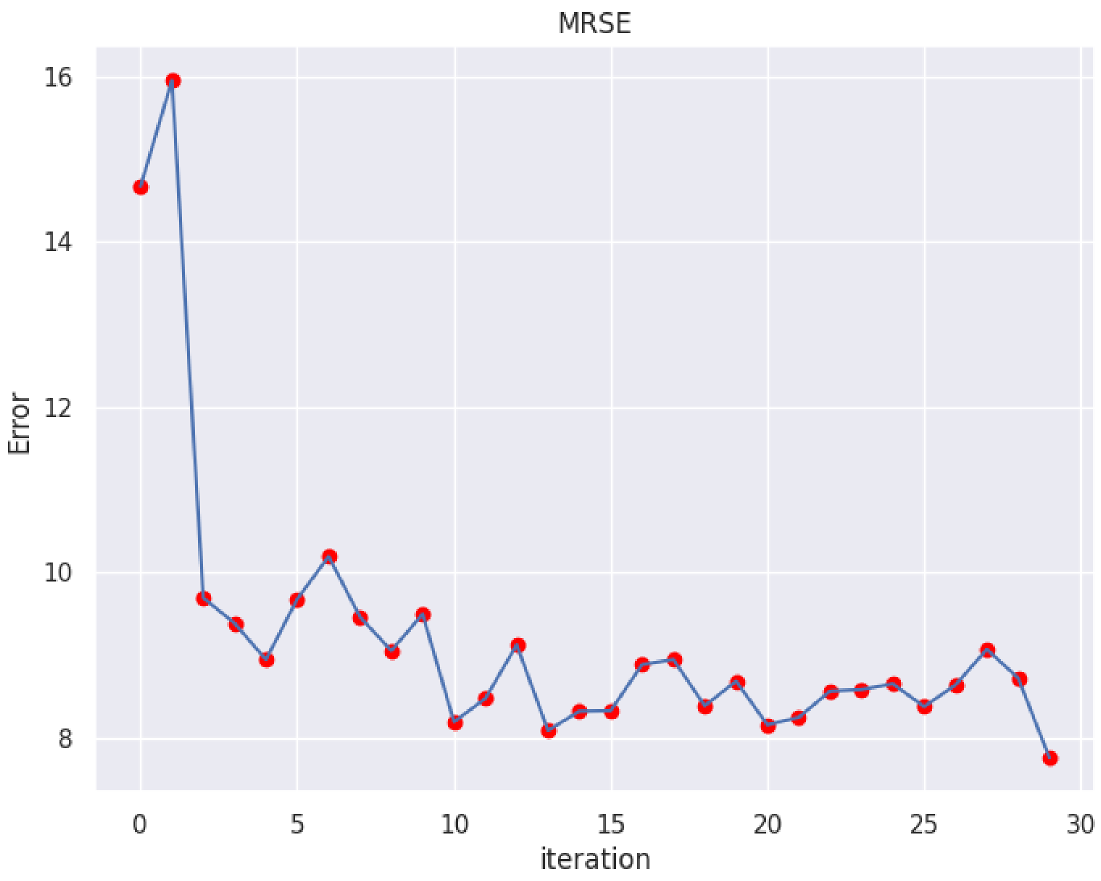
The gradient descent follows the algorithm proposed in the excersice, simulating 27 combinations for each iteration.

```
[28]: def gradient_descent( iterations = 1000, stopping_threshold = 1e-6):

          # Initializing weight, bias, learning rate and iterations
          n = 934
          k = 6
          beta = 0.3
          rho = 0.6

          delta_k = 1
          delta_beta = 0.1
          delta_rho = 0.1
          vacc_t = [0,5, 9, 16, 24, 32, 40, 47, 54, 59, 60, 60, 60, 60, 60, 60]
          set_k =    [k - delta_k, k, k + delta_k]
          set_beta = [beta - delta_beta, beta, beta + delta_beta]
          set_rho  = [rho - delta_rho , rho , rho + delta_rho ]
          permutations = [set_k, set_beta, set_rho]

          week_to_simulate = 15
          n_simulations = 10

          initial_infected_nodes = rng.choice(range(n), 10)

          current_weight = 0.1
          current_bias = 0.01
          iterations = iterations

          ks = []
          betas = []
          rhos = []
          costs = []
          previous_cost = None


          # Estimation of optimal parameters
          for i in range(iterations):


              #Run Simulations
              simulations  = dict({})
              bar = progressbar.
      ↪ProgressBar(max_value=len(list(product(*permutations))))
              for j,param in enumerate(product(*permutations)):
```

```
            S_i = Simulations(get_graph(n,param[0]), param[1], param[2],␣
↪time_limit = week_to_simulate, n_simulations=n_simulations,␣
↪use_progressbar=False, vacc_t=vacc_t)
            I_hat=  np.mean([simulation[1] for simulation in S_i], axis=0)

            error_i = MRSE(I_hat)
            simulations[error_i]=param

            bar.update(j+1)


        bar.finish()


        min_error = np.min(list(simulations.keys()))

        costs.append(min_error)
        ks.append(k)
        betas.append(beta)
        rhos.append(rho)



        # If the change in cost is less than or equal to
        # stopping_threshold we stop the gradient descent
        if previous_cost and abs(previous_cost-min_error)<=stopping_threshold:
            break


        if  previous_cost :
        #     if previous_cost > min_error:

                # Updating weights and bias
            k = simulations[min_error][0]
            beta = round(simulations[min_error][1],3)
            rho = round(simulations[min_error][2],3)
            # else:
            #     delta_beta/=2
            #     delta_rho/=2


        set_k =     [k - delta_k, k, k + delta_k]
        set_beta = [beta - delta_beta, beta, beta + delta_beta]
        set_rho   = [rho - delta_rho , rho , rho + delta_rho ]
        permutations = [set_k, set_beta, set_rho]

        # Printing the parameters for each 1000th iteration
        print(f"Iteration {i+1}/{iterations}: MRSE {previous_cost}, k \
        {k}, Beta {beta}, Rho {rho}, delta_b {delta_beta},  delta_rho␣
↪{delta_rho}")
        previous_cost = min_error

    return k, beta, rho, ks, betas, rhos,costs
```

[29]:
```
output = gradient_descent(iterations = 30, stopping_threshold=1e-3)
```

[30]:
```
# Visualizing the weights and cost at for all iterations
plt.figure(figsize = (8,6))
plt.plot(np.arange(0,len(output[-1])), output[-1])
plt.scatter(np.arange(0,len(output[-1])), output[-1], marker='o', color='red')
plt.title("MRSE")
plt.ylabel("Error")
plt.xlabel("iteration")
plt.show()
```



The error (MRSE) shows an overall decreasing trend in the early iterations, indicating that the model effectively adjusts its parameters at the beginning of the optimization. However, after approximately iteration 10, the error stabilizes between 7.5 and 10, exhibiting noticeable fluctuations.

One possible explanation for this instability is that the number of newly infected individuals per week is very low compared to the total population involved in the study. This results in weak and noisy gradient updates, making the optimization process less stable. Additionally, when changes in the data are small, the estimated gradients may not provide enough information for significant parameter adjustments, leading to erratic parameter updates. Another contributing factor might be the effect of randomness in the data, where small variations significantly impact the optimization due to the low number of new infections per week.

The preferential attachment graph used in our model generation lacks a fixed random seed, meaning that each execution produces a slightly different network topology. This randomness introduces an additional source of variability in the optimization process.

Since the graph structure affects the interactions between nodes, different runs may lead to slight variations in the computed gradients, impacting the stability and convergence of the optimization. This could contribute to the observed fluctuations in the error metric (MRSE) across iterations.

[31]:
```
lowest_err = min(output[-1])
lowes_idx = output[-1].index(lowest_err)
print(f"K: {output[3][lowes_idx]} Beta: {output[4][lowes_idx]} Rho:␣
↪{output[5][lowes_idx]} MRSE: {lowest_err}")
```

```
K: 6 Beta: 0.1 Rho: 0.2 MRSE: 7.760074097584378
```

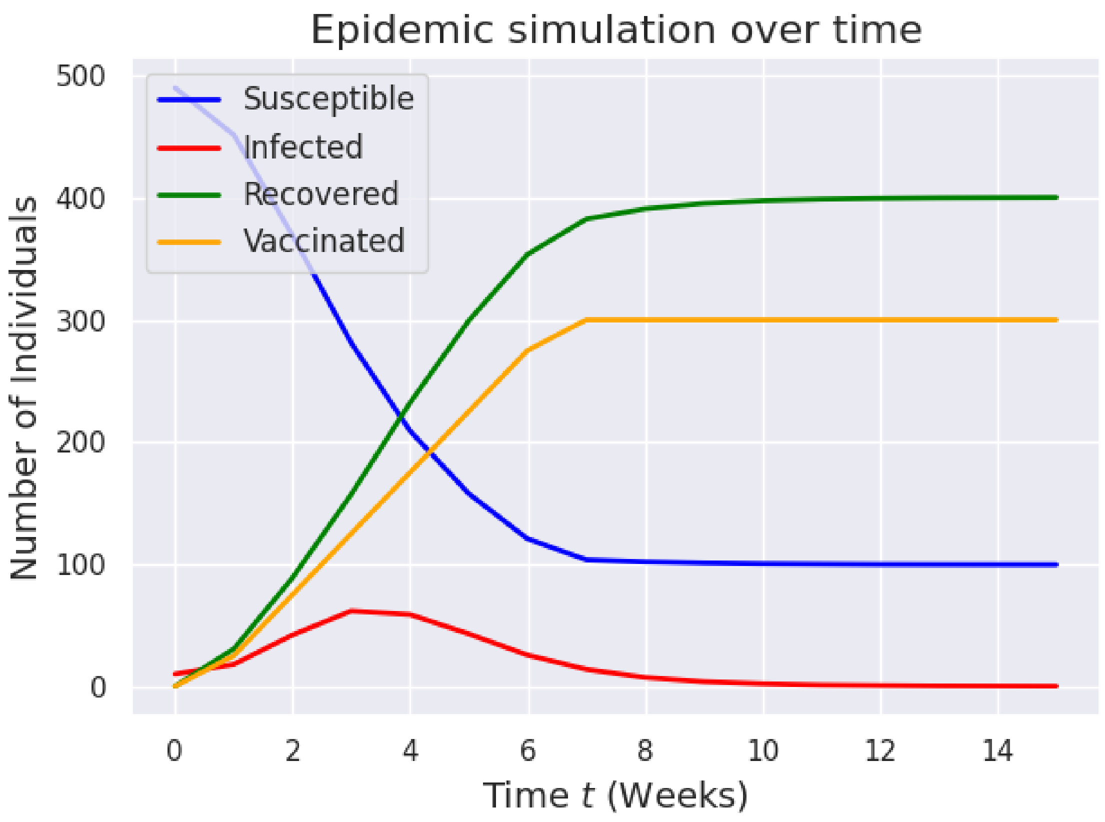
Since it was impossible to arrive to convergence, we chose the smallest MRSE as the best possible aproach, wich got the following parameters.

[32]:
```
s=  Simulations(get_graph(934,output[3][lowes_idx]),
                output[4][lowes_idx],
                output[5][lowes_idx],
                time_limit = 15,
                n_simulations=10,
                use_progressbar=True,
                vacc_t=[0,5, 9, 16, 24, 32, 40, 47, 54, 59, 60, 60, 60, 60, 60,␣
↪60]
               )
```

```
100% (10 of 10)
|########################| Elapsed Time: 0:00:01 Time:  0:00:01
```

[33]:
```
#Plotting average total number of susceptible, infected, and recovered␣
↪individuals at each week
# Calculate the average states for S, I, R
average_states = calculate_average_states(S)
average_new_vaccinated_nodes = np.mean([simulation[2] for simulation in S],␣
↪axis=0)

# Plotting the results
```

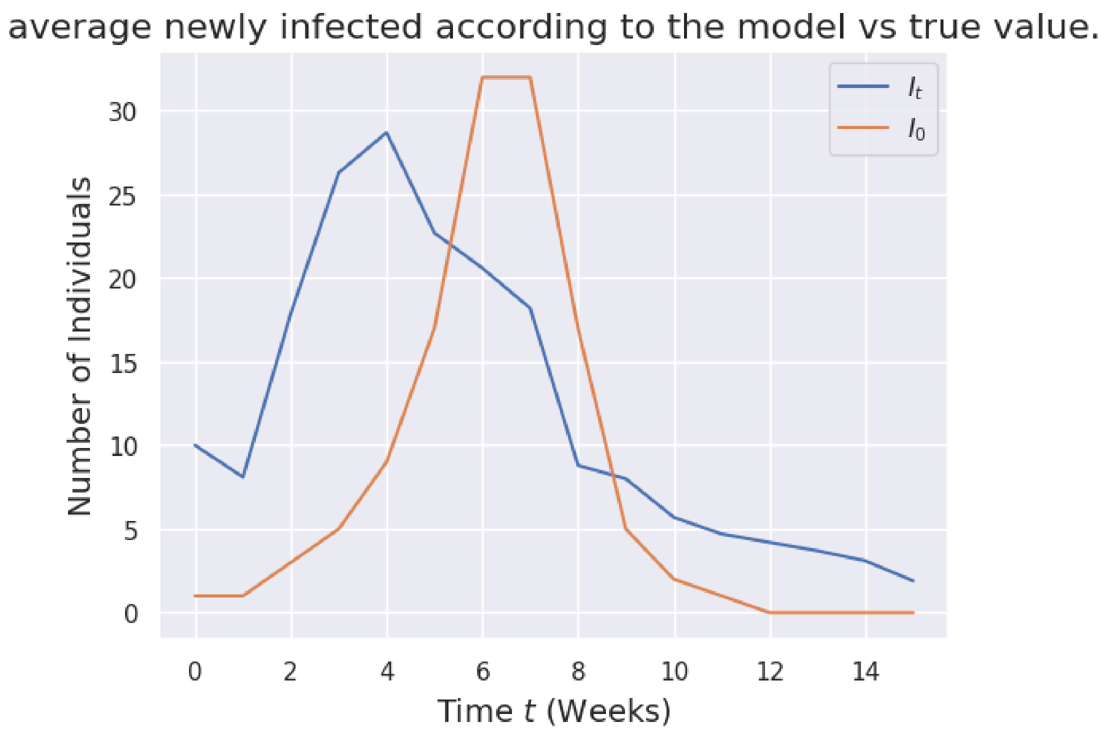
```
plt.plot(average_states[0], label='Susceptible', color='blue', linewidth=2)
plt.plot(average_states[1], label='Infected', color='red', linewidth=2)
plt.plot(average_states[2], label='Recovered', color='green', linewidth=2)
plt.plot(average_new_vaccinated_nodes, label='Vaccinated', color='orange',␣
 ↪linewidth=2)

plt.title("Epidemic simulation over time", fontsize=16)
plt.xlabel("Time $t$ (Weeks)", fontsize=14)
plt.ylabel("Number of Individuals", fontsize=14)
plt.legend(fontsize=12)
plt.grid(True)
plt.tight_layout()

plt.show()
```



The simulation as we can see got descent results comparable with the previous excercise.

[37]:
```
plt.plot(np.mean([simulation[1] for simulation in s], axis=0), label='$I_t$')
plt.plot([1, 1, 3, 5, 9, 17, 32, 32, 17, 5, 2, 1, 0, 0, 0, 0],label='$I_0$')
plt.title("average newly infected according to the model vs true value.",␣
 ↪fontsize=16)
plt.xlabel("Time $t$ (Weeks)", fontsize=14)
plt.ylabel("Number of Individuals", fontsize=14)
plt.legend()
```

```
[37]: <matplotlib.legend.Legend at 0x7ea4ac1e01d0>
```



Even if the error was still a little bit high, we can see that the fit at least got values close to the real values

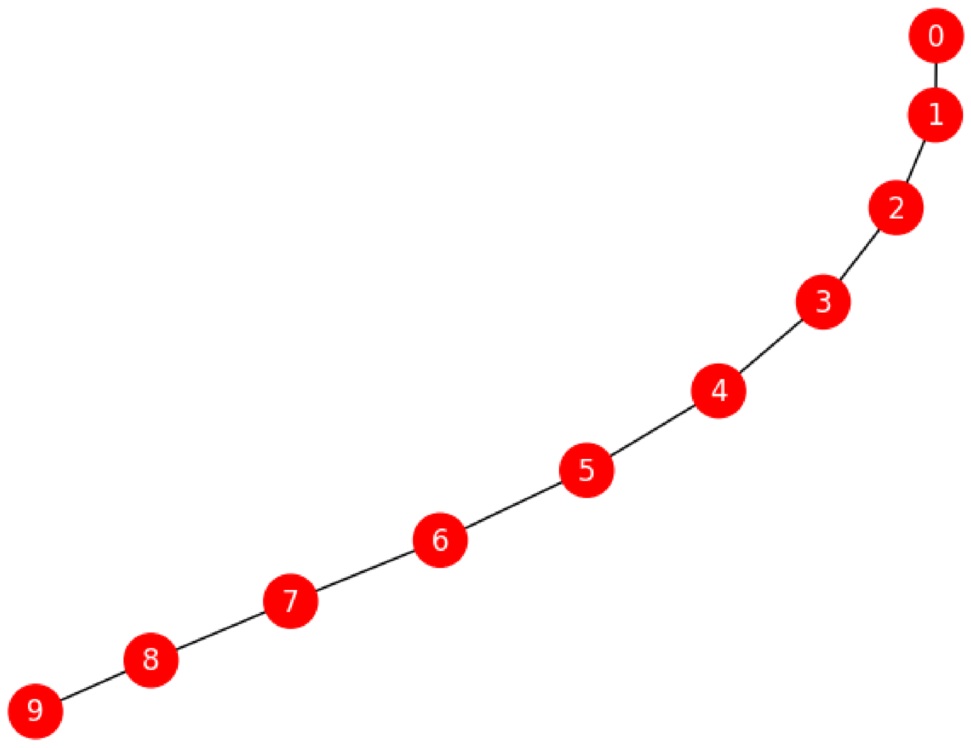
2 Coloring

In this example, study a line graph with 10 nodes.

Denote the i-th node state by Xi(t) and the set of possible states by C = {red,green}. At initialization, each node is red, i.e, Xi(t) = red for all i = 1,. . . ,10. Every discrete time instance t, one node I(t), chosen uniformly at random, wakes up and updates its color.

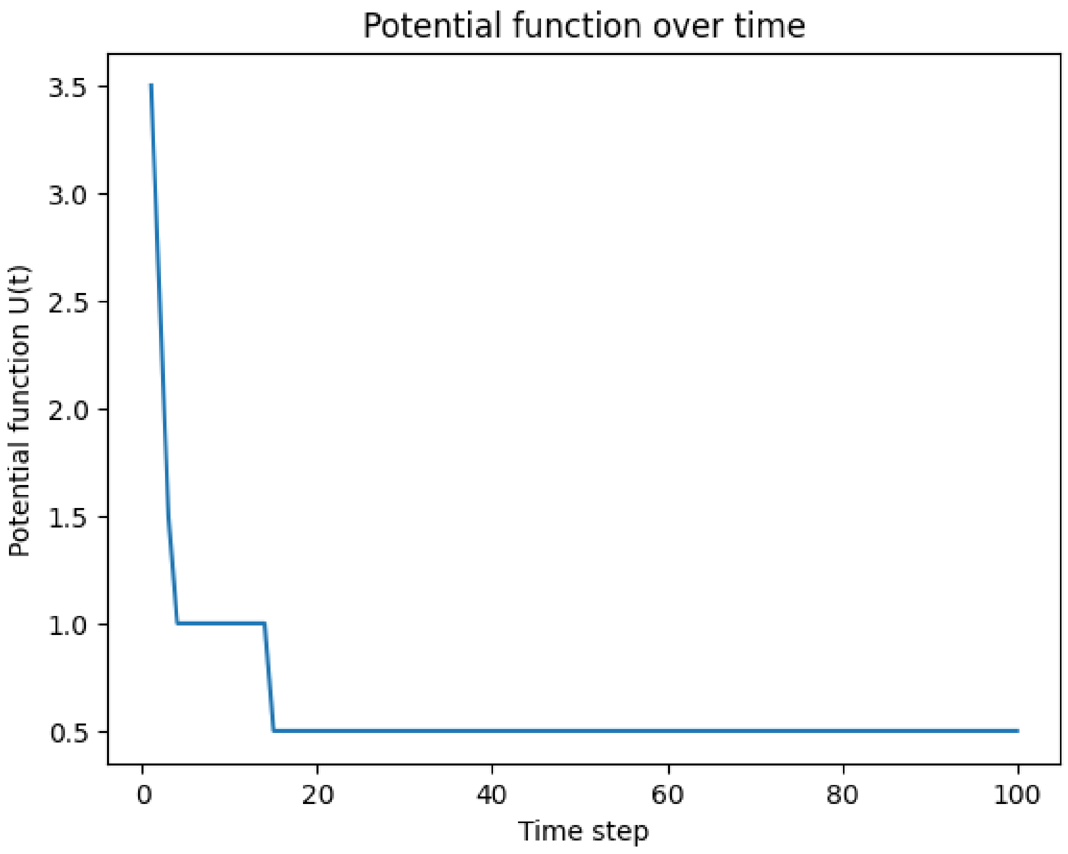
```
[ ]: import networkx as nx
     import matplotlib.pyplot as plt
     import numpy as np
     import random as rn
```

```
# Create an undirected line graph with 10 nodes
G = nx.path_graph(10)

#Possible colors= (red,green)
C=['red','green']
#At the beginning all the nodes are red
nx.set_node_attributes(G, C[0], 'state')

# Initialize weights
for u, v in G.edges():
    G[u][v]['weight'] = 1

colors= [G.nodes[node]["state"] for node in G.nodes]

# Draw the graph
nx.draw(G, with_labels=True, node_color=colors, node_size=500,␣
 ↪font_color='white')
plt.show()
```



```
[ ]: #Time to wake up is randomly and discrete
t=100

def cost(s, X_j):

        #calculate the cost
        #What is the cost of remain at the same color that the neighbor?

        if X_j==s:
            cost=1
        else:
            cost=0

        return cost

def probabilities(n, i,a,C):
     numerator= np.exp(-n*sum(G[i][j]['weight']*cost(a, G.nodes[j]['state']) for␣
 ↪j in G.neighbors(i)))
     denominator= sum(np.exp(-n*sum(G[i][j]['weight']*cost(s, G.
 ↪nodes[j]["state"]) for j in G.neighbors(i))) for s in C)
     return numerator/denominator

#Calculate the utility
def utility(G):
    return 1/2*sum(G[i][j]["weight"]*cost(G.nodes[i]["state"], G.
 ↪nodes[j]["state"]) for i,j in G.edges())

def coloring(t):

    U=[]

    for i in range(1,t+1):
        n=i/50

        #Choose a random node
        node=rn.choice(list(G.nodes))
        print("The chosen node is ", node, "with color ", G.nodes[node]["state"])

        #Calculate the probabilities
        p_red = probabilities(n, node, C[0],C)
        p_green = probabilities(n, node, C[1],C)

        #Choose the biggest probability and update the color
        if p_red>p_green:
            G.nodes[node]["state"]=C[0]
        else:
            G.nodes[node]["state"]=C[1]
        print(" The node ", node, " is now", G.nodes[node]["state"])

        U.append(utility(G))

    # Plot the potential function over time
    plt.plot(range(1, t + 1), U)
    plt.xlabel('Time step')
    plt.ylabel('Potential function U(t)')
    plt.title('Potential function over time')
    plt.show()
```

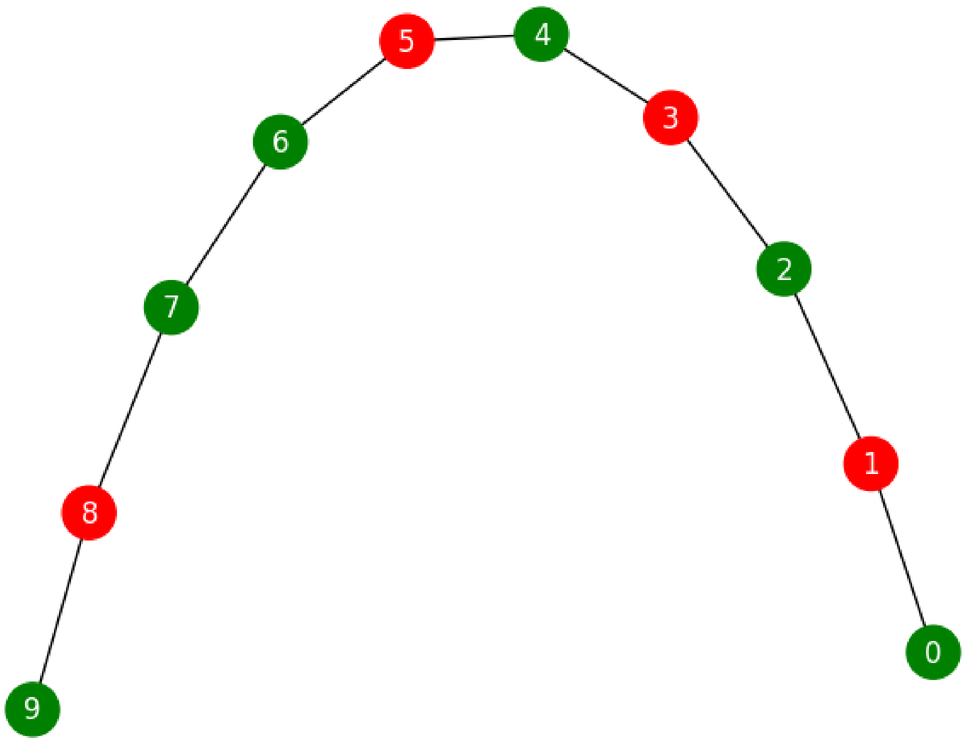
```
[ ]: coloring(100)
```



The potential function is directly proportional to the cost of remaining the same color as a neighbor. As the nodes update their colors at each step of the simulation to ensure that all neighbors have different colors, the utility function, which measures how similar the colors of the neighbors are, tends to decrease.

```
[ ]: colors= [G.nodes[node]["state"] for node in G.nodes]
     print(colors)
     # Draw the graph
     nx.draw(G, with_labels=True, node_color=colors, node_size=500,␣
      ↪font_color='white')
     plt.show()
```

```
['green', 'red', 'green', 'red', 'green', 'red', 'green', 'green', 'red',
'green']
```



2.1 Exercise b

Next, we use the coloring algorithm for the problem of assigning wifi-channels to routers.

The adjacency matrix of a network of 100 routers is given in wifi.mat and the routers' coordinates are given in coord.mat. Here, a link between two nodes means that the two routers are able to interfere with each other. The set of possible states is C = {1 : red,2 : green,3 : blue,4 : yellow,5 : magenta,6 : cyan,7 : white,8 : black}, where colors represent frequency bands

```
[ ]: from scipy.io import loadmat
     import scipy.io
```

```
[ ]: adjency_matrix= loadmat('wifi.mat')
     adjency_matrix=adjency_matrix["wifi"]
```

```
[ ]: adjency_matrix
```

```
array([[0, 0, 0, ..., 0, 0, 0],
       [0, 0, 0, ..., 0, 0, 0],
       [0, 0, 0, ..., 0, 0, 0],
       ...,
       [0, 0, 0, ..., 0, 0, 0],
       [0, 0, 0, ..., 0, 0, 0],
       [0, 0, 0, ..., 0, 0, 0]], dtype=uint8)
```

```
[ ]: routers_coordinates= loadmat("coords.mat")
     routers_coordinates=routers_coordinates['coords']
```

```
[ ]: routers_coordinates
```

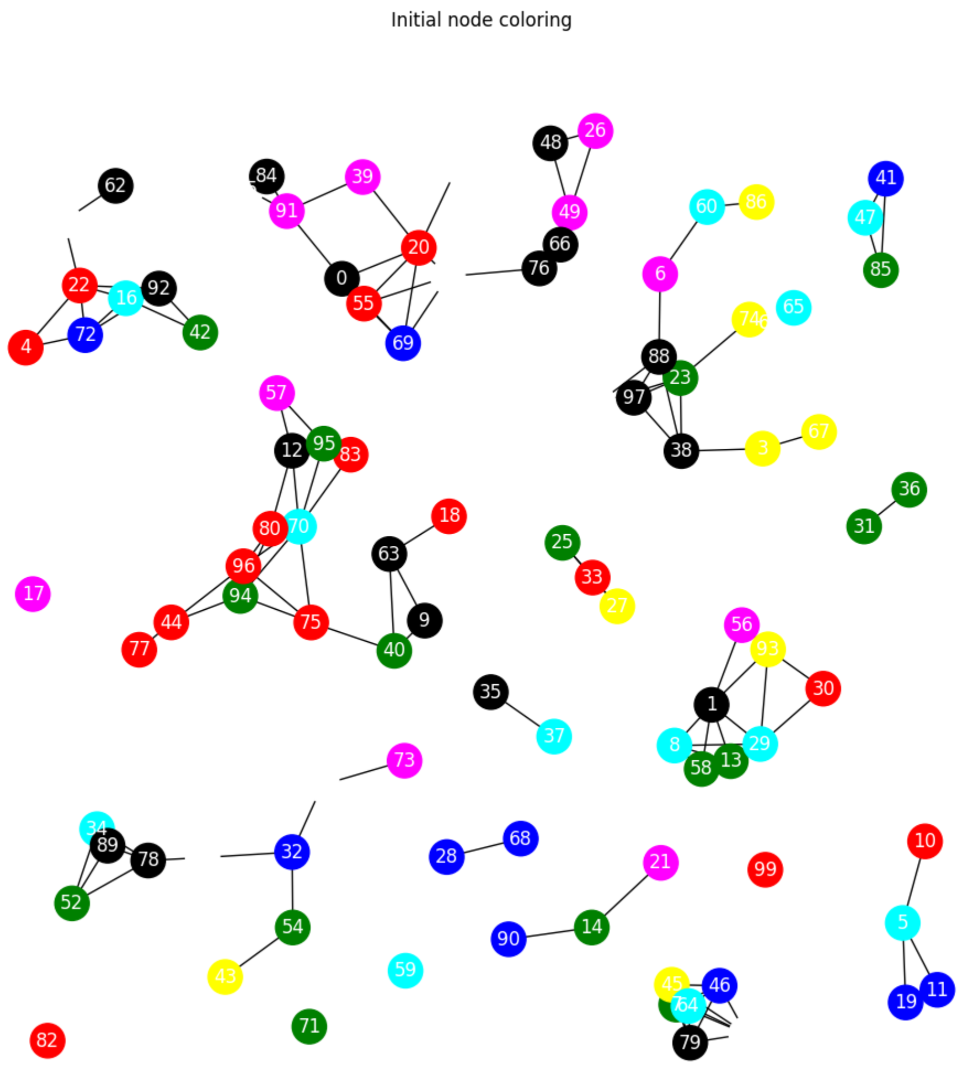
```
[ ]: # Create the graph

     def create_initial_graph():
         G1 = nx.from_numpy_array(adjency_matrix)

         #Possible states
         C={"red":1, "green":2, "blue":3, "yellow":4, "magenta":5, "cyan":6, "white":
      ↪7, "black":8}
         colors= list(C.keys())
         #It is necesarry to assign a value to the color in order to calculate the␣
      ↪cost funciton later

         #Arbitrary initial condition
         for i in range(len(G1.nodes)):
             G1.nodes[i]["state"]=rn.choice(list(C.keys()))

         print(G1.nodes[2])

         return G1, C, colors
```

```
[ ]: G1, C, colors =create_initial_graph()
```

```
{'state': 'white'}
```

```
[ ]: #Initial image
     plt.figure(figsize=(10, 10))
     pos = {i: (routers_coordinates[i, 0], routers_coordinates[i, 1]) for i in␣
      ↪range(G1.number_of_nodes())}
     node_colors = [G1.nodes[node]['state'] for node in G1.nodes]
```

```
nx.draw(G1, pos, with_labels=True, node_color=node_colors, node_size=500,␣
 ↪font_color='white')
plt.title('Initial node coloring')
plt.show()
```



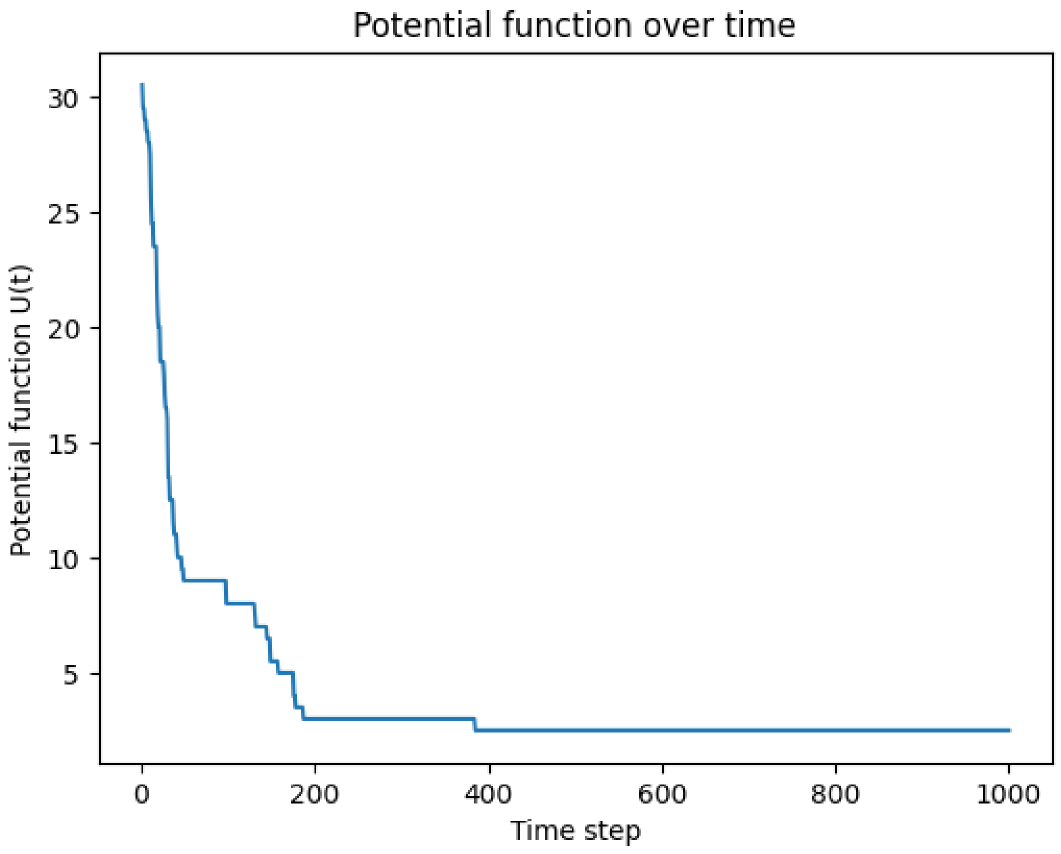
[ ]:
```
#Cost function
def cost(s,X_j):

    if X_j==s:
        cost=2
    elif(abs(X_j-s)==1):
        cost=1
    else:
        cost=0
    return cost

#Probabilities function

def probabilities(n, i,a,C):
     numerator= np.exp(-n*sum(G1[i][j]['weight']*cost(C[a], C[G1.
↪nodes[j]['state']]) for j in G1.neighbors(i)))
     denominator= sum(np.exp(-n*sum(G1[i][j]['weight']*cost(C[s],C[ G1.
↪nodes[j]["state"]]) for j in G1.neighbors(i))) for s in C.keys())
     return numerator/denominator

#Utility function
def utility(G1):
    return 1/2*sum(G1[i][j]["weight"]*cost(C[G1.nodes[i]["state"]], C[G1.
↪nodes[j]["state"]]) for i,j in G1.edges())
```

```
[ ]: np.exp(-1), np.exp(-1000), np.exp(-500)
```

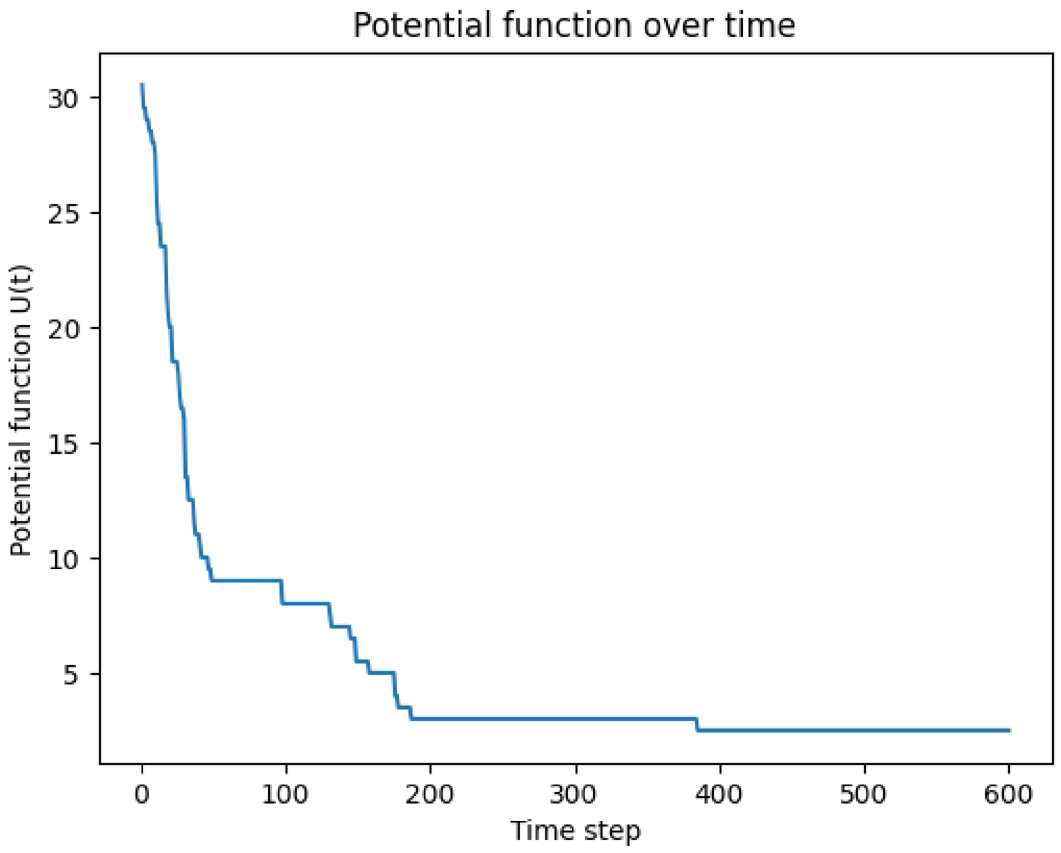
```
(0.36787944117144233, 0.0, 7.124576406741286e-218)
```

```
[ ]: t=1000
def bands(t):

    U=[]
    for i in range(1,t+1):

        #Choose a random node
        node=rn.choice(list(G1.nodes))
        print("The chosen node is ", node, "with color ", G1.
↪nodes[node]["state"])

        #update n
        n=i/50

        #Calculate the probabilities
        p_red = probabilities(n, node, colors[0],C)
        p_green = probabilities(n, node, colors[1],C)
        p_blue = probabilities(n, node, colors[2],C)
        p_yellow = probabilities(n, node, colors[3],C)
        p_magenta = probabilities(n, node, colors[4],C)
        p_cyan = probabilities(n, node, colors[5],C)
        p_white = probabilities(n, node, colors[6],C)
        p_black = probabilities(n, node, colors[7],C)

        #Choose the biggest probability and update the color
        probs=[p_red, p_green, p_blue, p_yellow, p_magenta, p_cyan, p_white,␣
↪p_black]
        G1.nodes[node]["state"]=colors[np.argmax(probs)]
        print(" The node ", node, " is now", G1.nodes[node]["state"])

        U.append(utility(G1))

    # Plot the potential function over time
    plt.plot(range(1, t + 1), U)
    plt.xlabel('Time step')
    plt.ylabel('Potential function U(t)')
    plt.title('Potential function over time')
    plt.show()

    print(len(U))
    plt.plot(range(1, 600 + 1), U[:600])
    plt.xlabel('Time step')
    plt.ylabel('Potential function U(t)')
    plt.title('Potential function over time')
    plt.show()
```

[ ]:
```
bands(1000)
```



```
1000
```

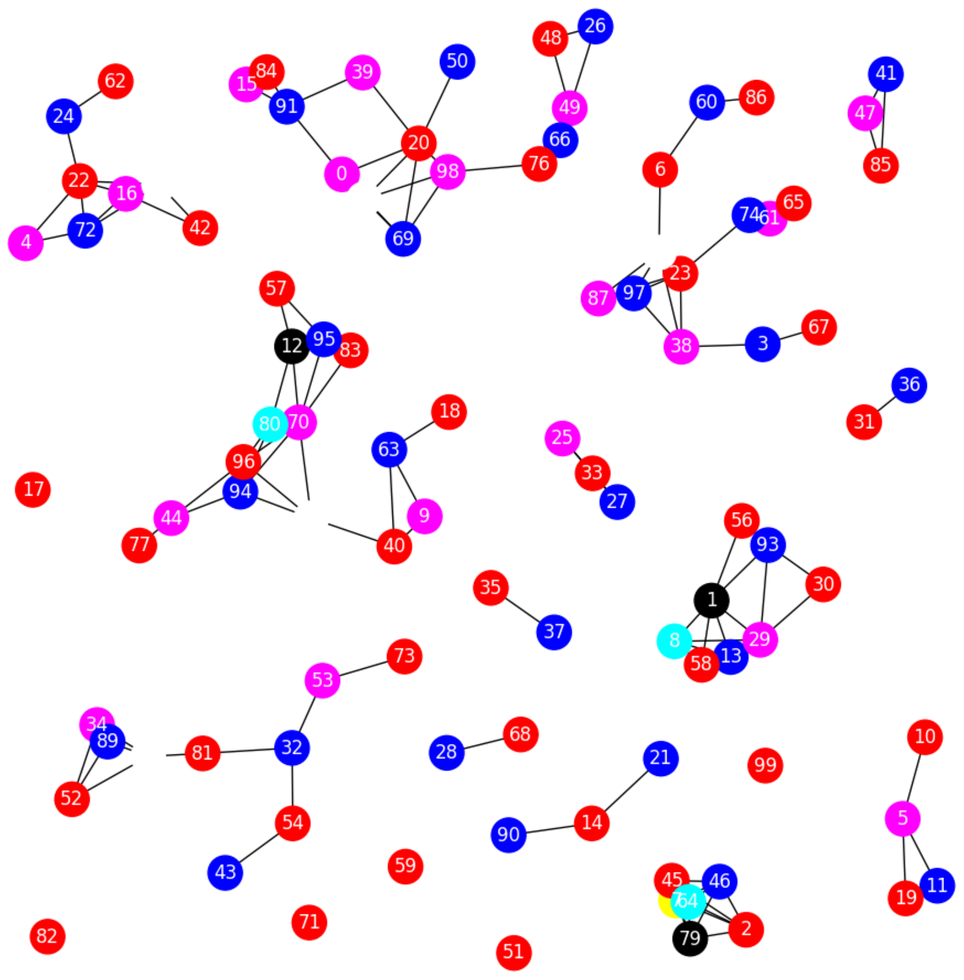


We can observe that between 200-400 the potential function arrives to its minimum value and remain in this until the end of the simulation.

Then, in a closer observation we can see that the potential funciton is decreasing because the probability of have the same color that the neighbour or to the next to this is decreasing at each time.

```
[ ]: plt.figure(figsize=(10, 10))
     pos = {i: (routers_coordinates[i, 0], routers_coordinates[i, 1]) for i in␣
      ↪range(G1.number_of_nodes())}
     node_colors = [G1.nodes[node]['state'] for node in G1.nodes]
     nx.draw(G1, pos, with_labels=True, node_color=node_colors, node_size=500,␣
      ↪font_color='white')
     plt.title('Final state coloring')
     plt.show()
```

Final state coloring



2.2 Exercise c

Evaluate what happens for different choices of $\eta(t)$, i.e., constant (with small and large values), or other increasing functions $\eta(t)$ etc. Comment on what you observe.

```
[ ]: G2, C2, colors2= create_initial_graph()
```

```
{'state': 'cyan'}
```

```
[ ]: #Put n as a parameter of the function

     def bands2(t, e, state):
```

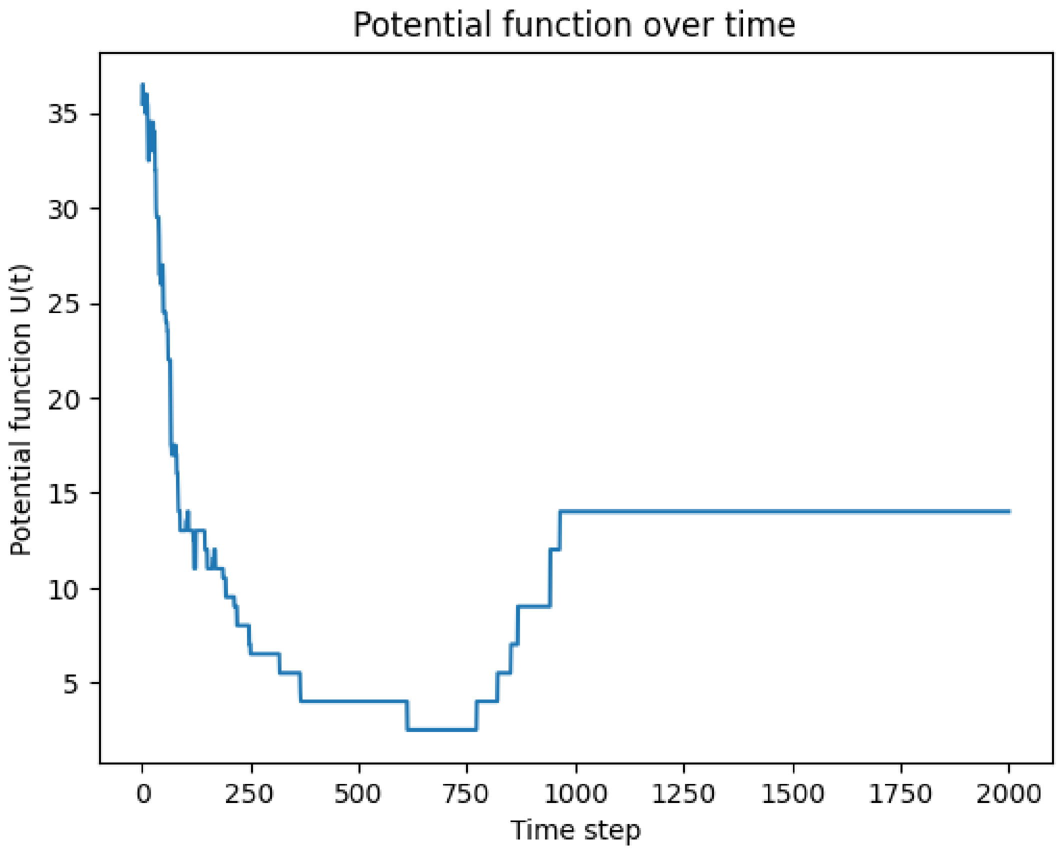
```
    U=[]

    for i in range(1,t+1):
        #Choose a random node
        node=rn.choice(list(G2.nodes))
        #print("The chosen node is ", node, "with color ", G1.
↪nodes[node]["state"])

        #update n
        if state=="static":
            n=e
        else:
            n=i/e

        #Calculate the probabilities
        p_red = probabilities(n, node, colors2[0],C2)
        p_green = probabilities(n, node, colors2[1],C2)
        p_blue = probabilities(n, node, colors2[2],C2)
        p_yellow = probabilities(n, node, colors2[3],C2)
        p_magenta = probabilities(n, node, colors2[4],C2)
        p_cyan = probabilities(n, node, colors2[5],C2)
        p_white = probabilities(n, node, colors2[6],C2)
        p_black = probabilities(n, node, colors2[7],C2)

        #Choose the biggest probability and update the color
        probs=[p_red, p_green, p_blue, p_yellow, p_magenta, p_cyan, p_white,␣
↪p_black]
        #print(np.argmax(probs))
        G2.nodes[node]["state"]=colors2[np.argmax(probs)]
        #print(" The node ", node, " is now", G1.nodes[node]["state"])

        U.append(utility(G2))

     # Plot the potential function over time
    plt.plot(range(1, t + 1), U)
    plt.xlabel('Time step')
    plt.ylabel('Potential function U(t)')
    plt.title('Potential function over time')
    plt.show()
```

There are strong changes in the graph

```
[ ]: G2, C2, colors2= create_initial_graph()
     bands2(2000, 1, 0)

     #n equal to the time
```

```
{'state': 'white'}
```

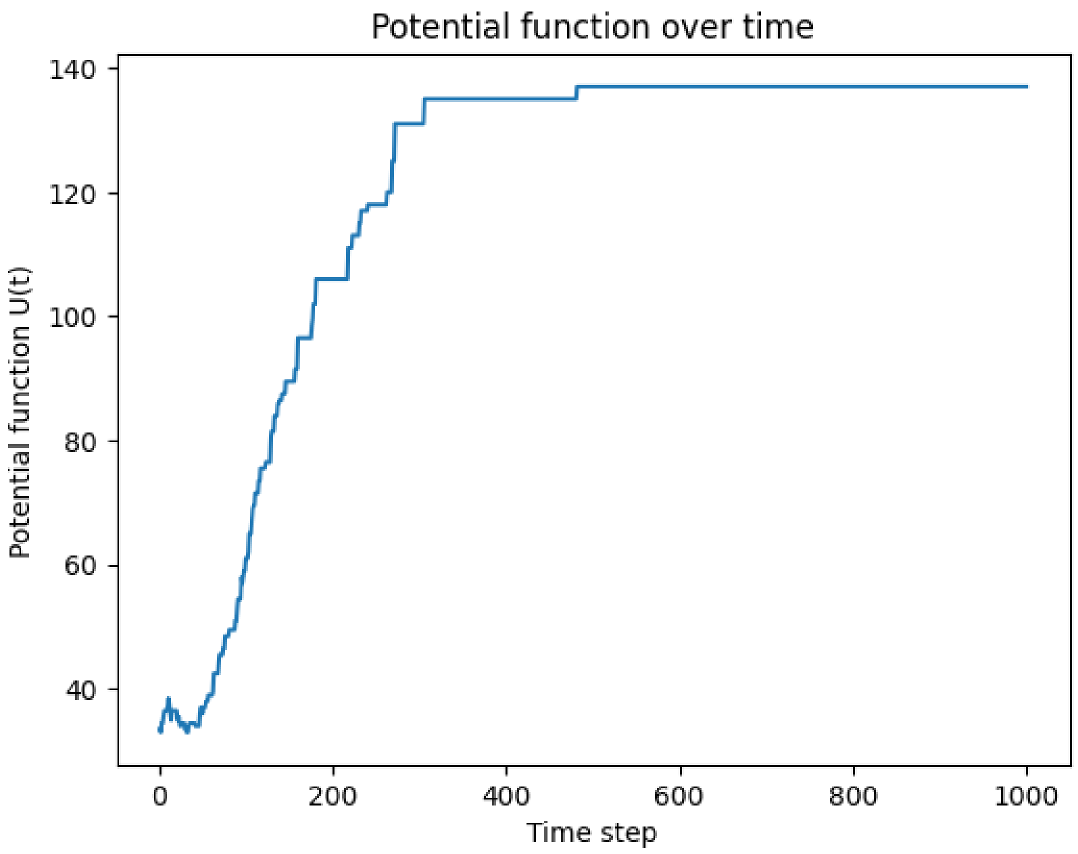
```
C:\Users\alejandrs\AppData\Local\Temp\ipykernel_4044\2069156484.py:17:
RuntimeWarning: invalid value encountered in scalar divide
  return numerator/denominator
```



If n is equal to the time it is more present the local minimums in the graph, the potential function doesn't decrease gradually, instead arrives to the final value after have having lower values.

```
[ ]: G2, C2, colors2= create_initial_graph()
     bands2(1000,0, "static")

     #Without n
```

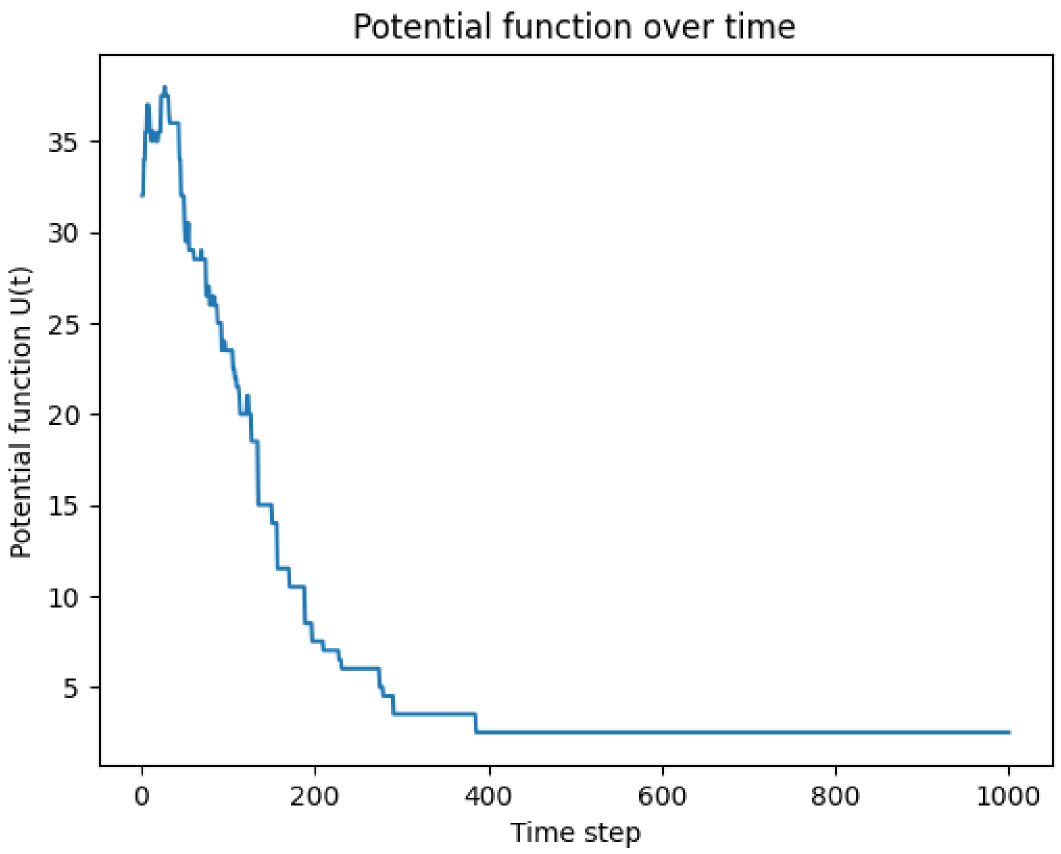
```
{'state': 'cyan'}
```



If n equals 0, the probabilities are the same for all colors, making it impossible to solve the problem.

```
[ ]: G2, C2, colors2= create_initial_graph()
     bands2(1000, 1000, 0)
```

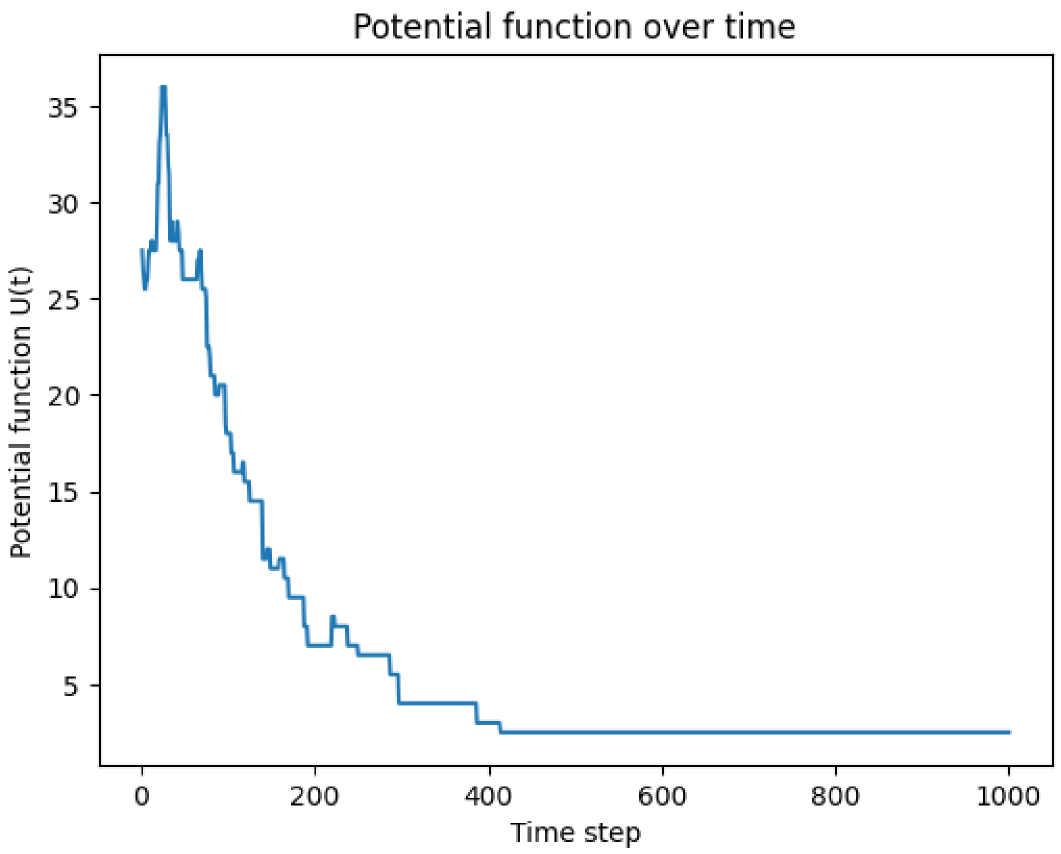
```
{'state': 'red'}
```



When n takes smaller values, most of the noise in the potential function occurs at the beginning, approximately within the first 100 time steps. After that, the decreasing behavior becomes more linear.

```
[ ]: G2, C2, colors2= create_initial_graph()
     bands2(1000, 100000, 0)
```

```
{'state': 'red'}
```



The smaller the value of n, the lower the noise in the function.