Lab 3: Spread of memes on a network

Urlich Icimpaye, Jakob Häggström, Oscar Jacobson

May 2022

1 Simple meme spread model

The following model was made from these rules and states:

- Resting : With probability p a resting person will discover a new meme by themselves and become a sharer.
- Sharer: With probability q a sharer will pick one person completely at random from the population. If that person is resting then they will now become a sharer. However, if the person they pick is bored, then the sharer will lose interest and become bored too
- Bored: With probability r a bored person will pick one person completely at random from the population. If that person is resting then the bored person will now become resting, otherwise they will continue to be bored. The simulation was done with $N = 1000$ People where there was 1 sharer and one bored initially. While in another example there was 100 sharers and 100 bored initially.

1.1 Examples of two different behaviours

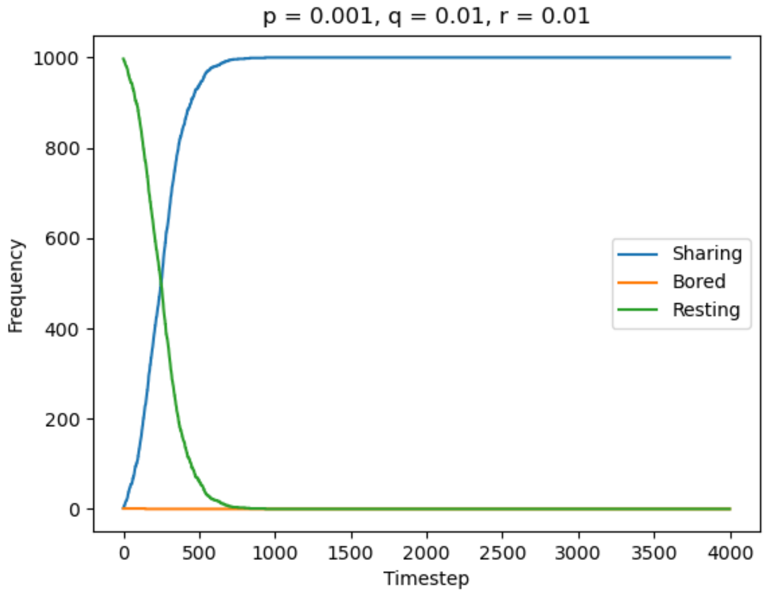

Figure 1: Result of the model with one bored and one spreading initially.

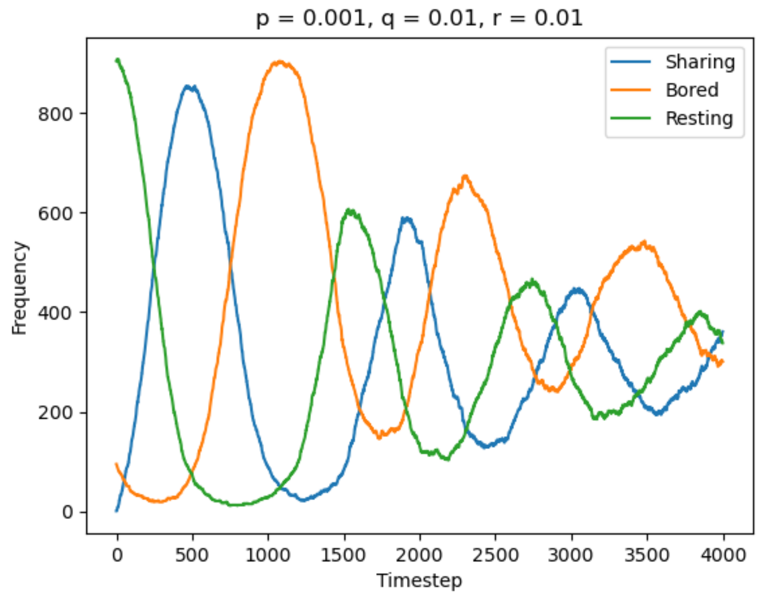

Figure 2: Result of the model with 100 bored and one spreading initially.

Figure 1 shows that the number of bored and number of resting dies out. The reason why bored dies out is likely due to the fact that there are 998 resting initially, so it's most likely that a bored will find someone resting and therefore die out because they also become resting. While the sharer will also likely only find people resting. Which means that the sharer won't likely be able to find a bored person and become bored. But when setting 100 sharers and 100 bored initially, the three states oscillates as seen in 2. This is because sharers manages to find bored, which makes them bored, but when bored nodes finds nodes that are resting they turn into resting which in turn makes it possible to return to a sharer with probability p. It's also shown in figure 2 that the number of sharers decreases as the number of bored increases, while the number of bored decreases when the number of resting increases. Which is expected considering the rules mentioned above.

1.2 Phase transitions

To generate the heat maps the model was simulated 20 times while varying the parameter p between $[0, 0.001]$, while the other parameters are fixed as such $q = 0.01$ $r = 0.01$. Each simulation was done for 2000 time steps. Note that there has been a typo on the y-axes in figures 3 and 4 where it says Q, it should be p. The heat map was also generated for two initial conditions. One configuration where the number of sharers and bored was set to one initially. And another where they're both set to 100.

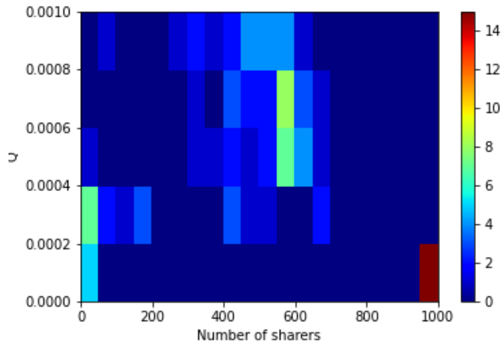

Figure 3: heatmap when varying p with initial conditions set as 100 sharers and 100 bored

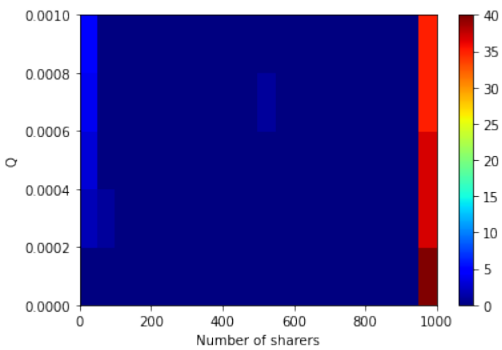

Figure 4: heatmap when varying p with initial conditions set as one sharer and one bored

In figure 4 the same behaviour observed in 1 since most runs gives between 950 and 1000 sharers, while some runs has zero sharers. This is again likely due to the fact that it's unlikely that a sharer finds a bored person. While the other initial condition shows that, when p is at it's largest the number of sharers decreases because it's also likely that a sharer find a bored person since there are more bored people initially than in 4. This means that the number of sharers can increase with low competition from the bored nodes as seen in figure 3.

2 Meme model on real- and lattice network

The meme model was applied to a real and lattice network, but each node can only spread its state to its neighbours. A typical result looked like the following:

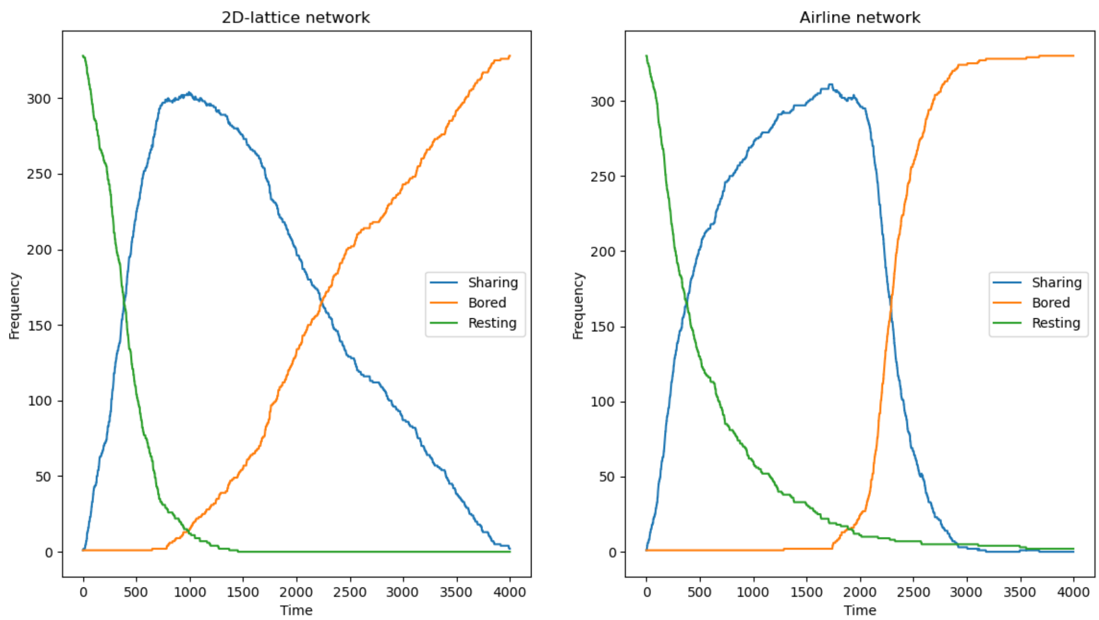

Figure 5: Result of the meme model on a 2D lattice - and real network with parameters $r, q = 0.01$ and $p = 0.001$.

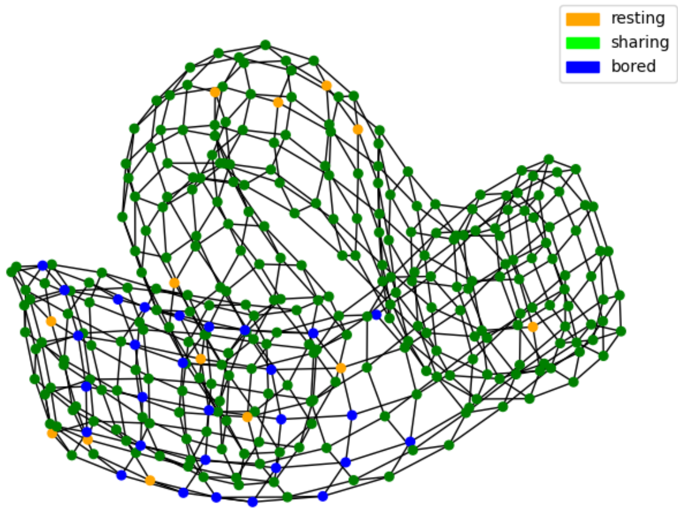

Figure 6: Image of the lattice network after 1000 steps.

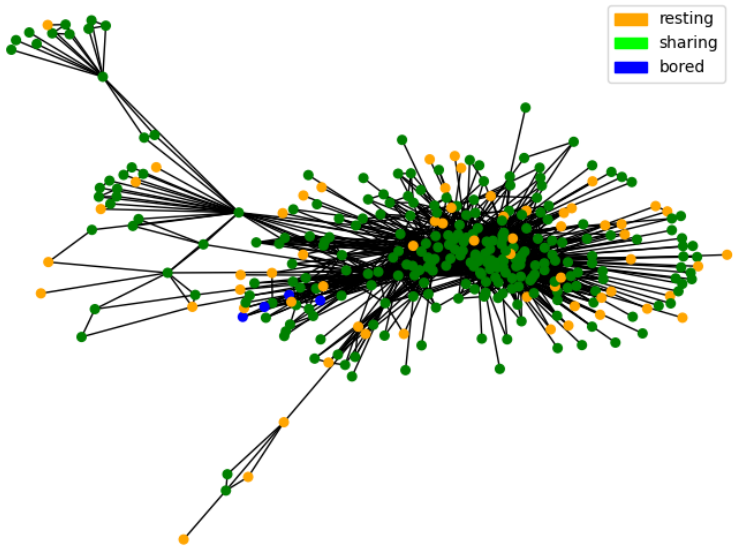

Figure 7: Image of the airlines network after 1000 steps

As seen in 5, both simulations seems to show similar characteristics. The number of sharers shows a significant increase, and reaches its maxima as the number of bored exceedes the number of resting. This means that theres a higher likelyhood that the sharing people finds bored people than resting, which makes the population of bored people increase. Thereafter, the dynamic of number of sharers was examined for different values of p.

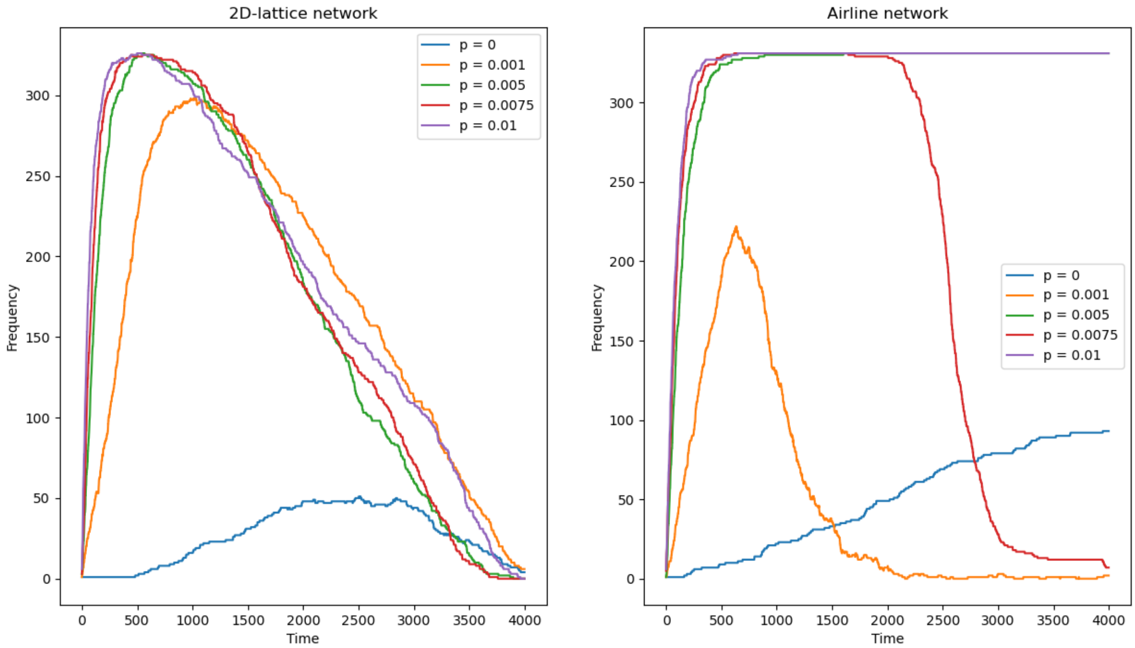

Figure 8: Number of sharers for different values of the p parameters.

As seen in 8, the increase of the p parameter seems to shift the curve to the left and or lowering it. In both networks a lower p seems to speed up the increasing trend of sharers. The airlines network maximum of sharers seem to exist over longer periods of times than the lattice network. This is most likely due to the structural differences in the networks. As seen in figure 6-7, the used networks differs significantly in the number neighbours per node.

The airlines network has a few nodes that are heavily connected, and the majority only has a few neighbours, whereas in the lattice network each node has four neighbours. This leads to the fact that a sharer "hitting" a bored node is generally harder in the airlines network compared to the lattice network. The cause for this is the fact that the nodes with more connections has a lower chance to find individual bored nodes in its neighbouring reach. As the model is built on an random choice of neighbours the same neighbour can be chosen over and over. The probability is therefore lower since a highly connected node has many neighbours. So when the number of sharers is high, then the system has to generally wait longer in order to find a bored node and transition into a bored spread. Though if an highly connected node becomes a "bored-spreading" state, then the shift from sharing to bored happens fast in the network since there are many neighbours to spread to.

Appendix

```
from networkx.algorithms import community
import networkx as nx
import numpy as np
import matplotlib.pyplot as plt
```

```python
import seaborn as sns


state_dict = {'resting': 0, 'sharing': 1, 'bored': 2}

def generate_mesh(N):

   mesh = [state_dict['resting'] for i in range(N)]

   mesh[0] = state_dict['bored']
   mesh[1] = state_dict['sharing']
   # mesh[1:100] = [state_dict['bored'] for i in range(100)]
   # mesh[100] = state_dict['sharing']

   return mesh

#e.g. p = 0.001, q = 0.01, r = 0.01)
def resting_or_sharing(p):
   return int(np.random.choice([state_dict['resting'], state_dict['sharing']], /
       size = 1, p = [1 - p, p]))

def sharer_or_bored(q,mesh, ind, N):

   choice = np.random.choice([0 , 1], size = 1, p = [1 - q, q])

   if choice:

      while True:
         rand_person = int(np.random.randint(0, N, 1))
         if ind!=rand_person:
            break

      if mesh[rand_person] == state_dict['resting']:
         mesh[rand_person] = state_dict['sharing']

      elif mesh[rand_person] == state_dict['bored']:
         mesh[ind] = state_dict['bored']


def bored_or_resting(r,mesh,ind, N):
   choice = int(np.random.choice([0 , 1], size = 1, p = [1 - r, r]))

   if choice:
      while True:
         rand_person = int(np.random.randint(0, N, 1))
         if ind!=rand_person:
            break
      if mesh[rand_person] == state_dict['resting']:
         mesh[ind] = state_dict['resting']


def get_stats(mesh, N):

   mesh_temp = np.array(mesh)

   nr_sharing = len(mesh_temp[mesh_temp == state_dict['sharing']])
   nr_bored = len(mesh_temp[mesh_temp == state_dict['bored']])
   return [nr_sharing, nr_bored, N - nr_sharing - nr_bored]
```

```
def run_sim(N , num_steps, p, q, r):
    mesh = generate_mesh(N)
    stats = []
    for step in range(num_steps):
        #print(mesh)
        for ind,person in enumerate(mesh):
            if person == state_dict['resting']:
                mesh[ind] = resting_or_sharing(p)
            elif person == state_dict['sharing']:
                sharer_or_bored(q,mesh, ind, N)
            else:
                bored_or_resting(r,mesh,ind, N)
        stats.append(get_stats(mesh, N))
    return stats
if __name__ == '__main__':
    p = 0.001
    q = 0.01
    r = 0.01
    N_persons = 1000
    num_steps = 4000
    n_sharers = np.linspace(0, 1000, 50)
    # tot_result_list = []
    # iter = 0
    # totiter = len(q_list) * 50
    # for j, q in enumerate(q_list):
    # for i in range(50):
    # stats = run_sim(N_persons , num_steps, p, q, r)
    # tot_result_list.append([stats[-1][0], q])
    # iter += 1
    # print(f"{100 *round(iter / totiter, 3)}% done")
    # tot_result_list = np.array(tot_result_list)
    # plt.figure()
    # plt.hist2d(tot_result_list[:,0], tot_result_list[:,1], bins=20, /
```

```
        cmap=plt.cm.jet)
    # plt.colorbar()
    # plt.xlabel('Number of sharers')
    # plt.ylabel('Q')
    # plt.show()

    stats = run_sim(N_persons , num_steps, p, q, r)
    stats = np.array(stats)

    ticks = list(range(num_steps))

    plt.plot(ticks, stats[:,0], label = 'Sharing')
    plt.plot(ticks, stats[:,1], label = 'Bored')
    plt.plot(ticks, stats[:,2], label = 'Resting')
    plt.xlabel('Timestep')
    plt.ylabel('Frequency')
    plt.title(f'p = {p}, q = {q}, r = {r}')
    plt.legend()
    plt.show()
```

Listing 1: The code for the simple meme spread model

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.patches as mpatches
import networkx as nx
from celluloid import Camera
import random
import os

MAIN_DIR = os.path.split(os.path.abspath(__file__))[0]
fig_dir = os.path.abspath(os.path.join(MAIN_DIR,'..', 'figs'))
data_dir = os.path.abspath(os.path.join(MAIN_DIR,'..', 'data'))

state_dict = {'resting': 0, 'sharing': 1, 'bored': 2}
colour_list = ['orange', 'green', 'blue']
color_dict = {'resting': np.array([255, 165, 0]), 'sharing': np.array([0, 255, /
    0]), 'bored': np.array([0, 0, 255])}
legend = [mpatches.Patch(color = color_dict[state] / 255, label = f"{state}") for /
    state in color_dict]


def generate_real_network():

    df = open(os.path.join(data_dir, 'airlines.txt'), 'r')
    num_vertices = 0

    for line in df:
        line_list = line.strip().split()
        if line_list[0] == '*Vertices':
            num_vertices = int(line_list[1])
            break

    G = nx.empty_graph(num_vertices)
    reading_edges = False
    for line in df:
        line_list = line.strip().split()
        if not reading_edges:
            if line_list[0] != "*Edges":
                continue
            else:
```

```python
            reading_edges = True
            continue
      else:
         G.add_edge(int(line_list[0]), int(line_list[1]))

   node_list = list(G.nodes)

   for node in node_list:
      if len(list(G.neighbors(node))) == 0:
         G.remove_node(node)

   node_list = list(G.nodes)

   print(f"Number of nodes: {len(node_list)}")
   number_nodes = len(node_list)

   indicies = [i for i in range(number_nodes)]

   ind1 = indicies[random.randint(0, number_nodes - 1)]

   indicies.pop(indicies.index(ind1))

   ind2 = indicies[random.randint(0, number_nodes - 2)]

   attr_dict = { node : {"state" : 0} for node in node_list}
   attr_dict[node_list[ind1]]["state"] = 1
   attr_dict[node_list[ind2]]["state"] = 2

   nx.set_node_attributes(G, values=attr_dict)

   return G

def generate_lattice_network(N):

   indicies = [i for i in range(N)]

   ind1 = indicies[random.randint(0, N - 1)]

   indicies.pop(indicies.index(ind1))

   ind2 = indicies[random.randint(0, N - 2)]


   G = nx.grid_2d_graph(int(N / 10), 10, periodic = True)

   node_list = list(G.nodes)

   attr_dict = { node : {"state" : 0} for node in node_list}
   attr_dict[node_list[ind1]]["state"] = 1
   attr_dict[node_list[ind2]]["state"] = 2


   nx.set_node_attributes(G, values=attr_dict)

   return G




# e.g. p = 0.001, q = 0.01, r = 0.01)
```

```
def resting_or_sharing(p, G, node):
    new_state = int(np.random.choice([state_dict['resting'], /
         state_dict['sharing']], size = 1, p = [1 - p, p]))
    G.nodes[node]["state"] = new_state

def sharer_or_bored(G, old_state, q, node):

    choice = np.random.choice([0 , 1], size = 1, p = [1 - q, q])

    if choice:


        neighbor = list(G.neighbors(node))
        num_neigh = len(neighbor)
        if num_neigh > 1:
            ind = random.randint(0, num_neigh - 1)
        else:
            ind = 0

        rand_person = neighbor[ind]

        if old_state.nodes[rand_person]["state"] == state_dict['resting']:
            G.nodes[rand_person]['state'] = state_dict['sharing']

        elif old_state.nodes[rand_person]["state"]== state_dict['bored']:
            G.nodes[node]['state'] = state_dict['bored']


def bored_or_resting(G, r, old_state , node):
    choice = int(np.random.choice([0 , 1], size = 1, p = [1 - r, r]))

    if choice:

        neighbor = list(G.neighbors(node))
        ind = random.randint(0, len(neighbor) - 1)
        rand_person = neighbor[ind]
        if old_state.nodes[rand_person]["state"] == state_dict['resting']:
            G.nodes[rand_person]['state'] = state_dict['resting']


def get_stats(G):

    nodes = G.nodes

    nr_sharing = 0
    nr_bored = 0
    nr_resting = 0

    for node in nodes:

        if nodes[node]["state"] == state_dict['sharing']:
            nr_sharing += 1

        elif nodes[node]["state"] == state_dict['bored']:

            nr_bored += 1

        else:
            nr_resting += 1

```

```
    return [nr_sharing, nr_bored, nr_resting]



def plot_mesh(G, camera, tick):
    node_colours = []
    for i in G.nodes():
        node_colours.append(colour_list[G.nodes[i]['state']])
    nx.draw(G, with_labels=False, node_color = node_colours, node_size=30, /
        edge_color='black', linewidths=1, font_size=15)
    plt.legend(handles = legend)
    plt.title(f"T = {tick}")
    camera.snap()

def run_sim(N , num_steps, p, q, r, record = False, assignment = 1):



    if assignment == 1:
        G = generate_lattice_network(N)
    else:
        G = generate_real_network()


    if record:
        fig = plt.figure()
        camera = Camera(fig)
        plot_mesh(G, camera, 0)
    else:
        camera = None

    stats = []

    for step in range(num_steps):
        #print(mesh)

        node_list = G.nodes
        old_state = G.copy()


        for node in node_list:

            if old_state.nodes[node]["state"] == state_dict['resting']:

                resting_or_sharing(p, G, node)

            elif old_state.nodes[node]["state"] == state_dict['sharing']:

                sharer_or_bored(G, old_state, q, node)

            else:

                bored_or_resting(G, r, old_state, node)

        stats.append(get_stats(G))
        if record and step % 10 == 0:
            plot_mesh(G, camera, step)


    return stats, G, camera
```

```
if __name__ == '__main__':


    p_list = [0.001]
    q = 0.01
    r = 0.01

    N_persons = 332
    num_steps = 1000
    record = False
    assignment = 2

    sharer_lat = []
    sharer_net = []
    for p in p_list:
        stats1, G1, camera = run_sim(N_persons , num_steps, p, q, r, record=record, /
            assignment = 1)
        stats1 = np.array(stats1)
        stats2, G2, camera = run_sim(N_persons , num_steps, p, q, r, record=record, /
            assignment = 2)
        stats2 = np.array(stats2)
        sharer_lat.append(stats1[:,0])
        sharer_net.append(stats2[:,0])

    if record:
        animation = camera.animate()
        animation.save(os.path.join(fig_dir, 'sim.gif'))

    node_colours1 = []
    node_colours2 = []

    for i in G1.nodes():
        node_colours1.append(colour_list[G1.nodes[i]['state']])

    for i in G2.nodes():
        node_colours2.append(colour_list[G2.nodes[i]['state']])


    plt.figure()
    nx.draw(G1, with_labels=False, node_color = node_colours1, node_size=30, /
        edge_color='black', linewidths=1, font_size=15)
    plt.legend(handles = legend)
    plt.show()

    plt.figure()
    nx.draw(G2, with_labels=False, node_color = node_colours2, node_size=30, /
        edge_color='black', linewidths=1, font_size=15)
    plt.legend(handles = legend)
    plt.show()


    time = list(range(num_steps))
    fig, (ax1, ax2) = plt.subplots(1,2)

    # for i, p in enumerate(p_list):
```

```
# ax1.plot(time, sharer_lat[i], label = f"p = {p}")
# ax2.plot(time, sharer_net[i], label = f"p = {p}")

# ax1.set_xlabel("Time")
# ax1.set_ylabel("Frequency")
# ax1.set_title("2D-lattice network")
# ax1.legend()
# ax2.set_xlabel("Time")
# ax2.set_ylabel("Frequency")
# ax2.set_title("Airline network")
# ax2.legend()
# plt.show()

# ax1.plot(list(range(num_steps)), stats1[:,0], label = 'Sharing')
# ax1.plot(list(range(num_steps)), stats1[:,1], label = 'Bored')
# ax1.plot(list(range(num_steps)), stats1[:,2], label = 'Resting')
# ax1.set_xlabel("Time")
# ax1.set_ylabel("Frequency")
# ax1.set_title("2D-lattice network")
# ax1.legend()
# ax2.plot(list(range(num_steps)), stats2[:,0], label = 'Sharing')
# ax2.plot(list(range(num_steps)), stats2[:,1], label = 'Bored')
# ax2.plot(list(range(num_steps)), stats2[:,2], label = 'Resting')
# ax2.set_xlabel("Time")
# ax2.set_ylabel("Frequency")
# ax2.set_title("Airline network")
# ax2.legend()
# plt.show()


# n_sharers = np.linspace(0, 1000, 50)

# tot_result_list = []
# iter = 0
# num_iter = 20
# totiter = len(p_list) * num_iter

# for j, p in enumerate(p_list):

# for i in range(num_iter):
# stats = run_sim(N_persons , num_steps, p, q, r)
# tot_result_list.append([stats[-1][0], q])
# iter += 1
# print(f"{100 *round(iter / totiter, 3)}% done")
```

Listing 2: The code for the 2D lattice