Aalen University

Project Report
Course of Studies: Computer Science

Epidemics

by

Sascha Mößle, Tim Staudenmaier

75987, 75981

Supervising Professor: Prof. Dr. Thomas Thierauf

Submission Date: 28.2.2024

Statutory Declaration

I, **Sascha Mößle, Tim Staudenmaier**, hereby declare that I have wrote the information available in this work truthfully and independently by myself.

Furthermore, I assure that I have used no other than the specified sources and aids, that I have marked all quotations that I have used from other sources, and that the work in the same or similar was not yet part of a study or examination.

Location, Date Signature (Student)

Abstract

Epidemic diseases are an important area of study. Throughout human history there have been several instances of epidemics significantly that have significantly affected large parts of the world. A recent example would be the Corona virus, which brought most parts of the world to a standstill. The field of epidemics focuses on those contagious diseases that spread from person to person, such as Corona of influenza.

Depending on the characteristics of the virus, epidemics can have a very different progression. Some can spread explosively and infect the whole world without causing many casualties, while others spread very slowly but persist for a long time with a high mortality rate. An important part of studying these diseases is simulating how different diseases might affect the world. The simulation of such diseases can be accomplished using network graphs. This paper introduces a model that can be used for simulation of epidemics and explains how the program that allows to simulate a disease using that model is built.

Contents

List of Figures

1 Introduction

1.1 Motivation

An epidemic outbreak can have a major impact on the world. Past epidemics have shown that if we do not know how to respond to an epidemic outbreak and are not prepared for such cases, a new disease can wipe out large parts of the world. The black death caused the death of about 30% to 60% of all Europeans (75-200 million) during the 1300s [13]. To better understand such scenarios, simulations play an important role, as the study of real cases of epidemic outbreaks is difficult since there are only so many in the history of humans. Also, in the event of an outbreak, it is important to be able to simulate the next few days/weeks in order to accurately predict how the epidemic will evolve.

For this reason this work will discuss an approach to simulate such epidemics by modeling networks of people and then simulating the spreading of diseases with different characteristics in these networks.

1.2 Problem definition

A model of the social network in which the disease is spreading in is crucial to the simulation. Depending on the transmission method of the disease, this network may be highly connected in the case of a disease with airborne transmission or have only few connections for diseases that are sexually transmitted. In addition, different diseases can spread completely different in the same social network even if they have the same transmission method because the characteristics of the disease also play an important role in how it spreads. As suggested by Easley and Kleinberg [2] the transmission of computer viruses works in a similar way can therefore also be modeled using networks.

The networks that can be created using the app must be able to model different social networks. The most important part for the epidemics simulation is the modeling of the amount of contacts with other people each person has. Since each group can have a significant number of members an efficient method for creating networks with large amounts of nodes is required that still allows to model most social networks.

A method to visualize these networks in a clear way is needed. The visual representation must still be usable with large amounts of nodes (e.g. over 100,000 nodes). To make the visualization of the network more usable, some settings must be provided to modify the displayed network, such as hiding certain connections or nodes. It also needs ways to represent the current state of the network in respect to the spread of the diseases.

The app also needs to allow for creation of multiple diseases with different characteristics. To simulate various epidemic scenarios properties like the infectiousness, duration of illness or fatality need to be editable.

The app should be able to simulate multiple diseases at the same time. The simulation needs to take into account which humans have contact with each other and then simulate the spreading of the diseases according to the characteristics of each disease and group of people.

During the simulation the app will collect statistics that allow a review of key information after the simulation, such as the number of new infections over time.

2 Theory of modeling epidemics

The models used by the the app developed in this work are based on chapters 19-21 of the book "Networks Crowds and Markets" by Easley and Kleinberg [2].

2.1 Simplest model

The first model proposed by Easley and Kleinberg [2] uses a very simplistic representation of networks. The network is represented as a tree, with each layer representing the nodes that come into contact with infected nodes from the previous cycle. The root of the tree is the first person to contract the disease in the social network. During the first cycle the k nodes at a depth of 1 may or may not get infected with the disease, depending on the infectiousness of the disease. During the second cycle, each of these k nodes now comes into contact with k other nodes at a depth of 2. This means during the second cycle, $k \cdot k = k^2$ people are potentially at risk of infection. During each cycle, the number of people who come into contact with the disease increases by a factor of k, for a total of k^{cycle} people potentially exposed to the disease during each cycle. Figure 2.1 shows a possible network structure.

The book [2] also explains the concept of the reproductive number R_0 in relation to this network. R_0 is the expected number of new cases of the disease caused by a single infected individual. In the case of the tree network, this means $R_0 = pk$, where

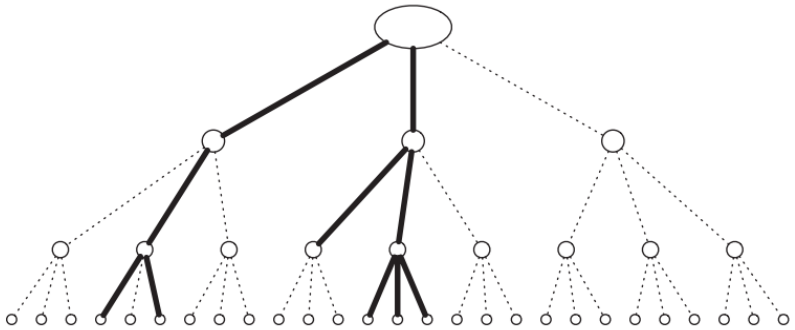


Figure 2.1: Tree representation of a network showing the spread of a disease (source: [2])

p is the infectiousness of the disease. If $R_0 > 1$, the number of cases will increase over time because each person infects more than one other person on average. Thus, there is a possibility that the disease will never die out. If $R_0 < 1$, the number of cases will decrease because on average each person will infect less than one other person, resulting in the disease dying out in a finite number of cycles. With this knowledge, the importance of R_0 in controlling an epidemic is clear. To prevent or stop an epidemic, the R_0 factor of a disease has to be less than 1.

2.1.1 Limitations

This model has several limitations. It assumes each person has contact with the same number of people, which is never the case in real social networks. There are always people who come into contact with more people than others. Also, each person can only infect others during the first cycle after being infected. It is not possible for a person to infect others during multiple cycles for longer lasting diseases. Further, it is not possible for a person to become infected a second time because there are no loops within the tree.

2.2 SIR Model

The SIR Model is a more advanced model that allows modeling of most social network structures by generalizing the contact structure. Easley and Kleinberg [2] define three stages each node can have:

- **S**usceptible: Node is not yet infected but susceptible to infection from its neighbors
- **I**nfectious: Node has caught the disease and has a probability to infect its neighbors
- **R**emoved: The node went through the full infection period and is removed from consideration for future cycles

The SIR Model uses a directed graph do indicate which nodes are neighbors and thus susceptible to infecting each other. The resulting graph does not have to be anti-symmetric it may also contain undirected edges.

In addition to the network the SIR Model uses two additional quantities to control the epidemic: p the probability an infected node infects a susceptible node and t_I the length of the infection period.

Initially some nodes are in the I state while all others are in the S state. After each

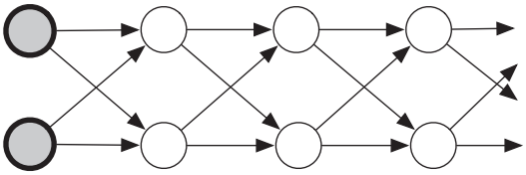


Figure 2.2: A network where the disease must pass through a narrow structure of nodes. (source: [2])

cycle all neighbors of the nodes in the I state are infected with a probability of p. After t_I steps a node is removed from the I state and can no longer infect others or be infected by others.

2.2.1 Reproductive Number

For these more complex networks calculating the reproductive number R_0 is not as trivial as for a tree based network. The book contains an example for a network shown in figure 2.2 in which even highly contagious diseases will die out.

Let d be a disease with a high contagiousness of $p = 0.8$ and an infection period of $t_I = 1$. Using the assumptions made for calculating R_0 in the tree network, this would result in $R_0 = 2 \cdot 0.8 = 1.6$, indicating that the disease is relatively likely never to die out. However, due to the structure of the network, the disease is going to die out relatively quickly. The probability of the disease not spreading during a cycle is $0.2^4 = 0.0016$, so on average the diseases will die out after $\frac{1}{0.0016} = 625$ cycles. This shows that in more complex networks, the structure of the network also plays a big role in the progression of the epidemic. It is not possible to calculate R_0 from the characteristics of the disease alone, the network structure must also be taken into account. Knowing this, it is impossible to accurately calculate R_0 in a highly complex network. It is however possible to observe the value of R_0 during the epidemic for previous cycles. Let R_0^k be the R_0 value for cycle k and I_k the number of infected nodes in cycle k. Then $R_0^k = \frac{I_k}{I_{k-1}}$, with this the evolution of the R_0 value can be observed, making it possible to estimate R_0 for future cycles and helping to understand whether the disease is currently dying out ($R_0 < 0$) or not ($R_0 > 1$). A simulation can also help to understand how the R_0 value of diseases will behave for a disease with certain quantities in a complex network.

2.2.2 Limitations

There are still some limitations to the SIR Model. Each person can only catch the disease at most once and the model does not allow for simulating multiple diseases

at the same time. However, the model allows most social networks to be represented, making it relatively easy to extend this network to include more complex diseases, e.g. with different levels of contagiousness depending on the time a node has been infected or a non-constant infection period.

2.3 SIS Model

The SIS Model is an extension to the SIR Model that also allows nodes to be reinfected multiple times. The **R**emoved state is exchanged with the **S**usceptible state, so nodes that are removed from the **I**nfected state are placed back in the **S**usceptible state. With this change, it is possible for diseases to survive an infinite amount of time in an SIS Model, as opposed to an SIR Model, where each simulation ends after a finite number of steps after the disease has burned through all nodes. The only way for the simulation in a SIS Model to end is if all infected nodes fail to transmit the disease to any of their neighbors t_I times.

2.4 Custom Model

2.4.1 Parameters

The model used by the developed app is a further extension of the SIR and SIS Models. It combines the SIR and SIS Models by moving nodes that finish the infection period either into the **R**emoved or **S**usceptible state depending on a quantity f. Each disease has a mortality rate f that determines whether a node is moved into the **R**emoved state and considered dead or moved back into the **S**usceptible state if it survived the infection. The **S**usceptible state is split into two substates: nodes that have never been infected before and nodes that have been put back into it after being infected at least once. This allows for different infectiousness values p_I for the first infection of a node and p_r for reinfections.

The t_I parameter will be split into two new parameters: t_{min} the minimum time an infection lasts before the node can be cured and t_ρ the probability a node is cured from its infection after the minimum time t_{min} has elapsed. This results in four states so far:

- Healthy: Susceptible nodes that were never infected before. They are at risk of being infected by their neighbors with a probability of p_I
- Infected: Nodes that are currently infected. The infection duration is least t_{min} cycles and the exact duration is determined by the probability t_ρ. After the

infection ended the node is moved into the cured state with probability $1 - f$ or into the deceased state with probability f.

- Cured: Nodes that were infected but survived. They are at risk of being infected by their neighbors with a probability of p_r
- Deceased: Nodes that died from the infection. They are not considered in future cycles and can not infect others anymore.

The infectiousness of a disease usually changes over the course of the infection. In most cases, an infected person is most likely to infect others during the first few days of contracting the disease. This should also be represented in the custom model. Thus, the values of p_I and p_r need to change with the time t_c for which a node has been infected. These time-dependent probabilities will be called p_I^t and p_r^t. A healthy node now has a probability of p_I^t to be infected by another node that has been infected since t cycles. Analogously, a cured node now has a probability of p_r^t.

Another part of epidemics that is not represented in the SIR or SIS Model are vaccines. During the course of an epidemic, vaccines can be used to reduce the fatality rate for infected persons and reduce the likelihood that vaccinated persons will contract the disease. To accommodate this in the custom model, two new node states are added:

- Vaccinated: Nodes that are vaccinated and now have a probability of getting infected of p_v^t and a fatality rate of f_v.
- Unvaccinated: Nodes that are not vaccinated and use the above mentioned probabilities p_I^t/p_r^t and f.

These two new states are not exclusive with the previous four states. Each node simultaneously has one of the two states, vaccinated or unvaccinated, and one of the previous four states healthy, cured, infected or deceased.

2.4.2 Network model

The network model is very similar to the one used by the SIR and SIS Model. However it does not allow for directed connections as there are very limited uses for those connections. Almost every human contact is bidirectional if for example one person gets close enough to another person to contract an air transmitted disease this transmission can always happen in both directions.

The network model used in this custom model organizes the nodes of one social circle in groups to more clearly organize the network for more visual clarity. Each group can be considered a small-world contact network were the nodes in each group are a localized part of the network that is highly connected with fewer connections to

other groups.

2.5 Characteristics of epidemics

2.5.1 Oscillating diseases

As Easley and Kleinberg [2] explain, diseases with certain characteristics can cause an oscillating number of infections. Consider a network with several highly connected groups that have have few connections between the groups. If a disease with a very high infection probability $p \geq 0.9$, a period of immunity $i > 0$ and a fatality rate close to zero breaks out in such a network, the umber of infections will oscillate. An example of how the number of infections might evolve over time can be seen in figure 9.8.

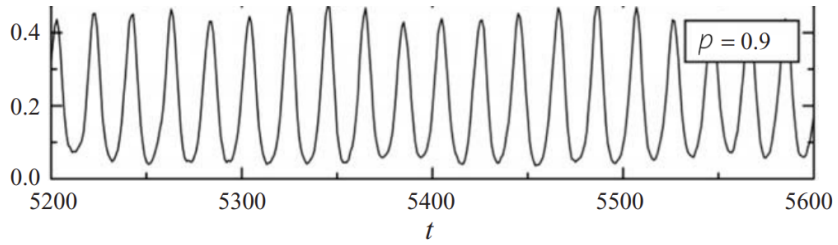


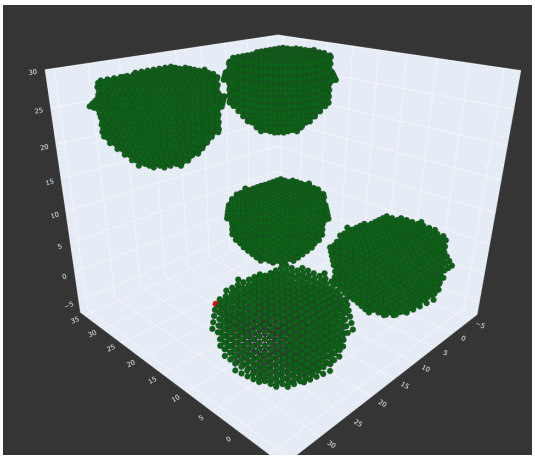
Figure 2.3: Amount of infections over time in a network of highly connected groups. The disease has a high infection rate and low fatality. (source: [2])

This happens because due to the high infectiousness of the disease, almost all nodes in a group that has at least one infected node will be infected within a few cycles. Since almost all nodes are infected after only a few cycles, there are only a few new infections in the next cycles due to the sparse set of available healthy nodes. If the initial wave started at cycle c, then at cycle $c' = c + t_i + i$ the nodes of the first wave are susceptible to infection again. This drastically increases the number of possible targets for new infections, which in turn increases the number of new infections again. This results in the oscillation of the number of new infections.

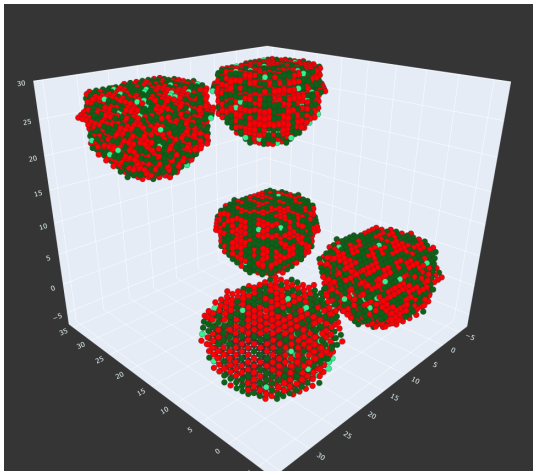
Such a scenario can be modeled with the created tool. The network consists of 5 groups with 2,000 nodes each. The groups have a lot of intra-group connections (5 per node) and only a small number of connections to the other groups. Each group is connected to two other groups with 1 edge per node and it is ensured that all groups have a path to all other groups.

The disease that will be simulated in this network has an infection rate of 0.2, a fatality rate of 0, a infection duration of 5 cycles and a immunity period of 3 cycles.

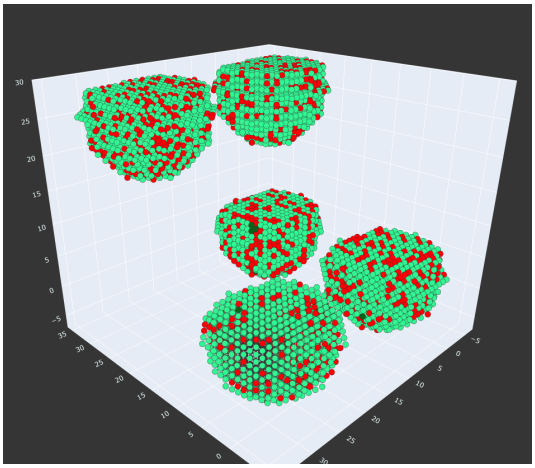
For simplicity the reinfection rate is the same as the initial infection rate and no vaccinations are used.



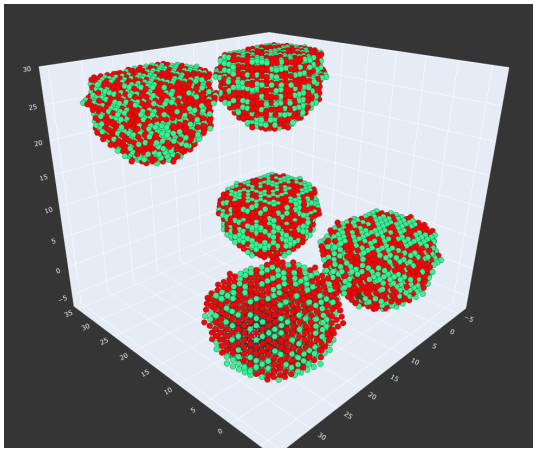
(a) Network after 0 Steps



(b) Network after 12 Steps, infections are at a maximum



(c) Network after 18 Steps, infection are at a minimum



(d) Network after 24 Steps, infection are at a maximum again

Figure 2.4: State of the network showing the oscillation of infected nodes. Red nodes are currently infected, dark green ones have never been infected and light green ones were previously infected but have recovered and have immunity.

Initially, 2 random nodes are infected with the disease. Figure 2.4 shows the state of the network after 0, 12, 18 and 24 cycles. The number of infections over time can be seen in figure 2.5 which clearly shows the oscillatory nature of this disease. Over time, the amplitude of the waves decreases, as the number of people infected at the same time and thus getting cured at the same time decreases, so the infections are no longer synchronized and each cycle has the same number of new nodes available to be infected. This happens because the infection rate was too low. The disease was not able to explosively infect all new nodes as soon as they became available, so some nodes remained uninfected for 2-3 cycles, shifting their cure time and breaking the oscillation. Increasing the infection rate to $p = 0.3$ prevents this, as the disease now spreads fast enough to infect every available node immediately. The result of this is shown in figure 2.6.

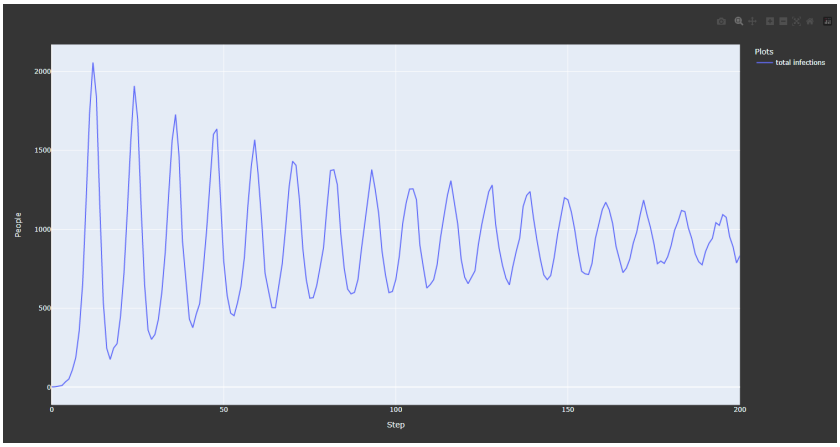


Figure 2.5: Amount of infections over time showing the oscillating nature with $p = 0.2$

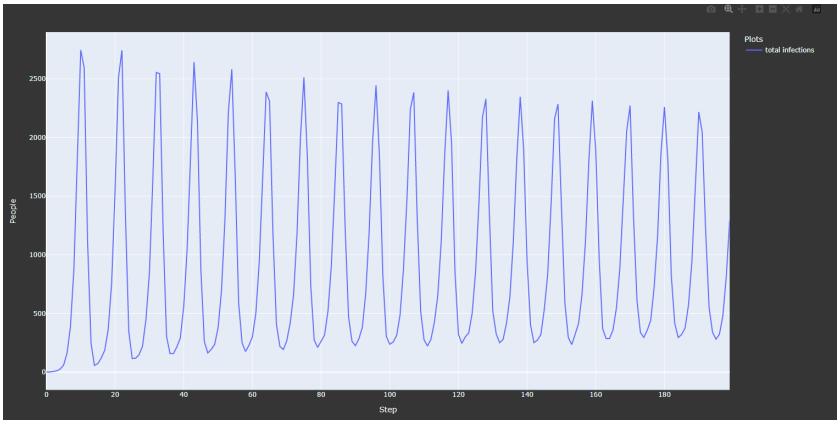


Figure 2.6: Amount of infections over time showing the oscillating nature with $p = 0.3$

Now the same network is used but the infection rate of the disease is decreased to 0.1. Because the disease is now not spreading as explosively as before it always has enough targets to infect until the previously infected nodes become cured again. Thus the amount of new infections is more consistent and does not oscillate as seen in figure 2.7.

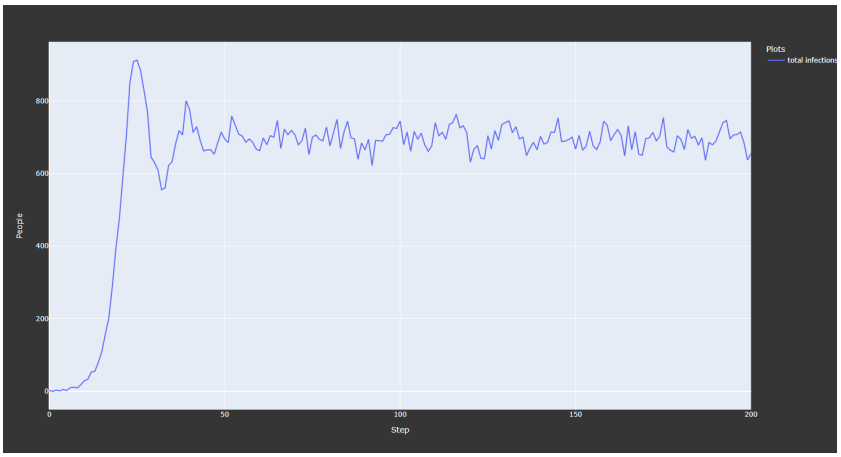


Figure 2.7: Amount of infections over time showing that no oscillations occur if with $p = 0.1$

Another way to break the oscillation is to keep $p = 0.3$ but decrease the duration of the infection to 2 cycles and remove the immunity period. Now the infected nodes become cured so fast that there are enough new nodes to infect for every cycle, again resulting in a relatively consistent amount of new infections as indicated in figure 2.8

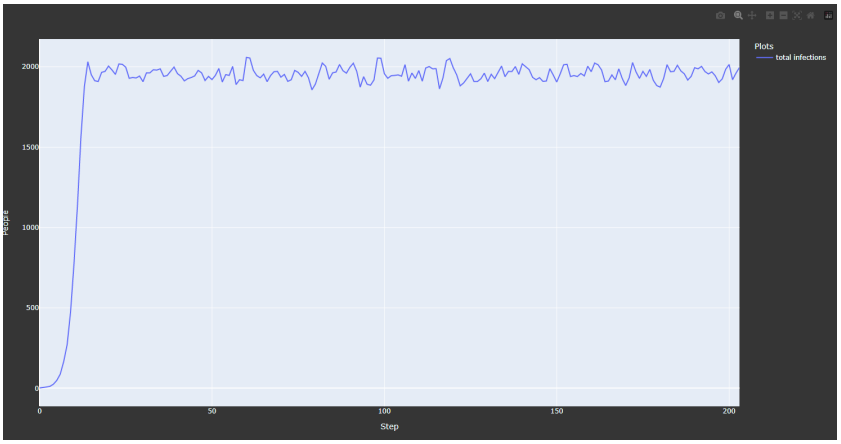


Figure 2.8: Amount of infections over time showing that no oscillations occur even with $p = 0.3$ if the duration is too short

2.5.2 Epidemics without diseases

Diseases are not the only thing that is capable of spreading through social networks. As previously mentioned computer viruses spread in a similar pattern. However, there are other completely different things that can spread in social networks in a similar way. Information, for example, can be modeled using a very similar approach to the one proposed for diseases. Information can be passed from person to person whenever they talk to each other, be it over a long distance through things like text messages or by meeting in person.

Other things that can spread through social networks include traditions or certain habits. Looking at the world, people within each country are much more likely to interact with other people from the same country than with people from other countries. This means each country can be modeled as a highly connected group of people with few connections to other countries. This model can be used to explain how each country has its own traditions that differ from other countries the further away they are. Traditions spread easily within a country, but the further away a country is, the less likely the tradition is to spread to that country. Because there are few connections to other countries, foreign traditions rarely spread to other countries and if they do, they are drowned out by the explosively spreading regional traditions of the other country.

These models can also be used to model genetic inheritance. As suggested by Easley and Kleinberg [2] this can be done by connecting parents to their offspring. Simulations can then be run to show the spreading of various genetic factors through the ancestry trees. This type of simulation itself offers many areas that can be expanded to create various simulations relevant to understanding human history and origins. It can also be used to indicate and visualize the existence of common ancestors and can give an idea of how many generations it takes to reach them. One such model that deals with genetics and common ancestors is the Wright-Fisher Model. Haller and Messer [6] go into more detail on how this model and the genetic simulation work. Genetic simulation and the Wright-Fisher Model are a large research area in themselves, there are several other works that discuss these topics, explain the mathematics behind those models, possible uses etc. [12] [11] [7].

These different applications shows how the use of epidemics and social networks goes way beyond modeling diseases and can help understand complex relations in various different areas of research.

3 Implementation

An app that allows for the simulation of epidemics is created. First, the required functionalities of the app have to be determined. It must be able to simulate different epidemic cases that require different social networks and diseases.

3.1 Network Editor

The app needs an editor that allows for the creation of different social networks. Since social networks can consist of a large number of people, this editor needs to allow the user to quickly create networks with a large number of nodes and connections. For this purpose there are three settings for a group of nodes:

- **Size**: The amount of nodes in the group
- **Intra group connections**: The amount of edges each node has to other nodes of the same group
- **Intra group connections delta**: The variation for the amount of edges each node has to other nodes of the same group. E.g. a connection amount of 5 with a delta of 3 would result in each node having between 2 and 8 connections.

A network then consists of any number of groups, each with different settings. To allow connections between these groups, there are similar settings available for each pair of groups:

- **Inter group connections**: The amount of edges each node has to other nodes of the other group
- **Inter group connections delta**: The variation for the amount of edges each node has to other nodes of the other group. E.g. a connection amount of 5 with a delta of 3 would result in each node having between 2 and 8 connections.

Using these properties the user is able to quickly generate big networks to model most social situations. To visualize the structure of the network there will also be a 3D view of the current network which can be updated after each change. Chapters 4 and 5 explain the implementation of the feature for generating and visualizing

networks in more detail.

3.2 Disease Editor

To model the diseases that spread in the network, the app contains a tab where the desired characteristics for each disease can be set. These include the properties discussed for the custom model in section 2.4 as well as some other properties required for visualizing the network:

- **Name**: Name of the disease, will be shown in the legend when displaying the network
- **Color**: Color nodes infected with this disease will have
- **Fatality rate**: f, the chance a node is moved into the deceased state after the infection period is over
- **Vaccinated fatality rate**: Fatality rate for vaccinated nodes
- **Infection rate**: p_I, the chance an unvaccinated node gets infected by a neighbor
- **Reinfection rate**: p_r, the chance a previously infected node gets infected again by a neighbor
- **Vaccinated infection rate**: p_v, the chance a vaccinated node gets infected by a neighbor
- **Minimum duration**: t_{min}, the minimum cycles an infection lasts
- **Cure chance**: t_ρ, chance a node gets cured (or dies) each cycle after it has been infected for t_{min} cycles
- **Immunity period**: Amount of cycles a node is immune after being cured
- **Infectiousness factor**: Decrease of infection rates with each cycle the node has been infected. E.g. with a initial $p_I = 0.5$ and a factor $I = 0.9$ the infection rate after x cycles is $p = p_I \cdot I^x$.
- **Initial infections**: Amount of nodes infected with this disease in cycle 0, the start of the simulation

3.3 Simulation

To simulate diseases, the app contains two different tabs. One where the network is displayed in 3D and the current state of each node is indicated by colors. The other tab contains only text for each group, describing how many nodes are infected, deceased, etc. per group. The simulation without visualizing the graph allows for faster simulation of large numbers of steps. In networks with a large number of nodes (over 100,000, depends on computing power of the system) building the graph visualization can take a long time making the simulation with the visual representation almost unusable while the text based simulation is still able to simulate multiple steps per second.

Both simulations include buttons to advance one step, automatically advance steps over time, reset the simulation and save the statistics collected during the simulation. The simulation which displays the networks also contains buttons to modify the display of the network, e.g. to hide specific node groups or edges so only the areas of the network that the user wishes to observe are displayed. Chapter 6 explains the implementation of this feature in more detail.

3.4 Statistics

The last tab of the app can be used to view the statistics collected during simulations. It displays the individual statistics in a coordinate system and contains functions to split the statistics by different parameters, e.g. to show only infections with a certain disease/all diseases or of only a certain group. The graph can be viewed as the individual value of each step as well as the cumulative value up to each step.

3.5 Frameworks

The app is constructed using python. For the UI QT [1] is used, this is further explained in chapter 8. For the display of the graphs, the plotly [8] library is used along with the dash implementation it provides for creating webpages. The graphs are then be displayed on the dash webpage which is embedded in the QT app. The implementation of the website is explained in chapter 7. For saving a project the JSON file format is used for the network and Pickle files for the statistics. The saved project files contain all necessary information, like the groups of the network, diseases and stats of previous simulations to allow closing the app after saving without losing any progress.

4 Network Generation

The developed tool provides a user interface for generating networks to simulate epidemics. To create these networks, different groups of people can be defined. Each group consists of n nodes, the members, each of which has a certain number of edges to other nodes of the same group. The number of edges per node is determined by a mean μ and a delta δ. Each node will then have an edge count between $\mu - \delta$ and $\mu + \delta$.

Let $g1$ and $g2$ be two different groups. Then in addition to the above mentioned edges within a group, it is also possible to specify the number of edges between the nodes of the two groups with an average and delta value, in the same way as described above.

To create a network graph that meets all of these requirements multiple algorithms are needed.

4.1 Creating edges within a group

Let g be a group of n nodes, each of which needs between $\mu - \delta$ and $\mu + \delta$ edges. Then the first step is to generate a sequence of n integer numbers within this range. This sequence represents the degree each node should have at the end.

4.1.1 Randomly adding new edges

This algorithm was the first iteration for creating the networks, it has two issues.

First, not every sequence of degrees is graphic, e.g. not every sequence of degrees has a corresponding simple graph. A simple graph is a graph consisting only of undirected edges and no loops.

For example, let s be a sequence of random integers with sum m. If m is not even, s is definitely not graphic, because in a simple graph without loops, every edge increases the total degree of all nodes by two. Thus it is impossible to have a total degree of all nodes that is not even. This is not checked in the above algorithm 1, which would

Algorithm 1 Adding random edges

```
nodes ←nodes with less than required degree
while nodes is not empty do
    n1, n2 ← two unconnected nodes
    connect n1 and n2
    if n1 has required degree then
        remove n1 from nodes
    end if
    if n2 has required degree then
        remove n2 from nodes
    end if
end while
```

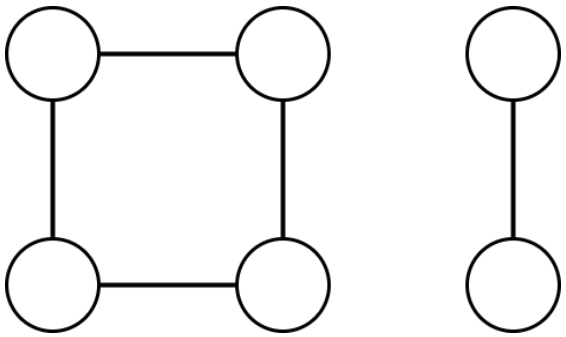


Figure 4.1: Graph created by random algorithm 1 with $\mu = 2$ and $\delta = 0$

cause the *nodes* list to contain only one node at the end, making it impossible to select two unconnected nodes from it.

The second issue is that by adding edges randomly it is possible to end up in a situation, where all remaining edges are already connected but have still not reached the required degree.

Figure 4.1 shows one such situation. In this graph each node needs to have a degree of exactly two. In theory it is possible to create such a graph, but the random algorithm 1 may end up in the situation shown in the figure. Here the only two nodes remaining in the *nodes* list are the two on the right which are already connected. From this state it is impossible to create a graph that satisfies the degree requirements.

The second iteration of the algorithm uses a more methodical approach to adding the edges to solve the above issues.

4.1.2 Erdos-Gallai Theorem

To solve the first problem of degree sequences that are not graphic, the Erdos-Gallai Theorem is used. It provides one of the two known approaches to solving the graph realization problem, i.e. it gives a necessary and sufficient condition for a finite sequence of natural numbers to be the degree sequence of a simple graph.

$$\sum_{i=1}^{k} d_i \leq k(k-1) + \sum_{i=k+1}^{n} \min(k, d_i) \tag{4.1}$$

for all integers k with $1 \leq k \leq n$, where $d_1 \geq d_2 \geq \ldots \geq d_n$ are non-negative integers.

A sequence of degrees is graphic if and only if $d_1 + d_2 + \ldots + d_n$ is even and the above equation holds for every k. The inequality ensures that the sum of the first k terms of the degree sequence does not exceed the theoretical maximum number of edges $(k(k-1))$ plus the sum of the remaining edges $\sum_{i=k+1}^{n} \min(k, d_i)$ for the remaining vertices.

For this tool, every degree sequence needs to be graphic, because it should always be possible to generate a graph for the user's input. This means that if the Erdos-Gallai Theorem shows that a sequence is not graphic, it must be changed until it is. This is done by simply decrementing a random number of the degree sequence by one.

If the Erdos-Gallai Theorem failed because the sum was not even, the sum will be even after decrementing once. By decrementing random numbers of the sequence, only the left side of the inequality gets smaller, because k never changes. This means after decrementing enough times, the theorem will always be fulfilled. Thus, any sequence of degrees can be changed to be graphic.

The python implementation of this algorithm can be seen in 4.1.

```
1 def _erdos_gallai(self) -> bool:
2     if sum(self.deg_seq) % 2 != 0:
3         return False
4     n = self.size
5     for k in range(1, n + 1):
6         if sum(self.deg_seq[:k]) > k * (k - 1) + sum(min(k, d)
              for d in self.deg_seq[k + 1 :]):
7             return False
8     return True
```

Listing 4.1: Erdos-Gallai Theorem in Python

4.1.3 Havel-Hakimi algorithm

After creating a graphic degree sequence using the Erdos-Gallai Theorem, with all degrees within the $\mu - \delta$ and $\mu + \delta$ range, this degree sequence needs to be converted to a network graph.

Algorithm 2 Havel-Hakimi algorithm

deg_seq $\leftarrow$ graphic sequence of degrees
nodes $\leftarrow$ list of nodes, with same length as *deg_seq*
while $sum(deg\_seq) > 0$ **do**
  $n \leftarrow nodes.pop(0)$
  $d \leftarrow deq\_seq.pop(0)$
  targets $\leftarrow$ n nodes with highest degree from *nodes*
  for t in *targets* **do**
    connect t and n
    $deg\_seq[t] \leftarrow deg\_seq[t] - 1$
  end for
end while

Using the Havel-Hakimi algorithm 2 a simple graph can be constructed for every graphic sequence of degrees. This algorithm always finds a correct solution as proven by Erdős et al. [3].

Python implementation

The Havel-Hakimi algorithm is implemented in the class HavelHakimi. This class has properties for the sequence of degrees `deg_seq`, list of node ids `node_id_seq` and a dictionary which nodes are connected `edges` (each key node id has an undirected edge to all its value node ids).

First the `node_id_seq` is sorted to be non-increasing and the `node_id_seq` is shuffled to assign random degrees to each node. Then the Havel-Hakimi algorithm is used:

```
1 def _connect_nodes(self):
2     while sum(self.deg_seq) > 0:
3         node = self.node_id_seq.pop(0)
4         deg = self.deg_seq.pop(0)
5         targets = self._get_highest_n_nodes(deg)
6         for target in targets:
7             self.deg_seq[self.node_id_seq.index(target)] -= 1
8         self.edges[node] = targets
9         self._sort_sequence()
```

Listing 4.2: Havel Hakimi Algorithm in Python

The function in listing 4.2 implements the above algorithm 2. After each iteration, the degree sequence is sorted so it is non-increasing again. When sorting the `deg_seq` it is important that the `node_id_seq` is altered in the same way so each node id still corresponds to the same degree, this function is shown in 4.3.

```
1 def _sort_sequence(self):
2     self.node_id_seq = [x for _, x in sorted(zip(self.deg_seq,
          self.node_id_seq), reverse=True)]
3     self.deg_seq.sort(reverse=True)
```

Listing 4.3: Sorting degrees and node ids

4.2 Creating edges between groups

Randomly creating edges between two groups leads to similar problems as described in section 4.1. Therefore, a modified version of the Havel-Hakimi algorithm and the Erdos-Gallai theorem is used.

4.2.1 Creating the degree sequence

Let $g1$ and $g2$ be two disjoint groups of size n and m. If $n \neq m$, it may not be possible for every node to have a degree between $\mu - \delta$ and $\mu + \delta$. Let $n = 5$ and $m = 10$ with $\mu = 5$, $\delta = 0$. If every node in $g2$ has a degree of 5, all nodes in $g1$ would have a degree of 10. If every node in $g1$ has a degree of 5, the average degree of the nodes in $g2$ would be only 2.5.

So if $n \neq m$ one group may have lower degrees than the minimum or higher degrees than the maximum for certain μ and δ. In this work the solution where the bigger group may have a lower degree will be used.

First, the degree sequence for the smaller group $g1$ is created. The sequence consists of n integer numbers between $\mu - \delta$ and $\mu + \delta$. The degree sequence of the $g2$ must have the same sum as the sequence of $g1$, otherwise the sequences are not graphic because each added edge decreases the sum of the degree sequences of $g1$ and $g2$ by one and the Havel-Hakimi algorithm finishes only when both sequences reach 0.

Let $s1$ be the sum of the degree sequence for $g1$. An algorithm is needed to generate a sequence of m integer numbers with sum $s1$, where the numbers should be as close to or inside the desired range $\mu - \delta$ to $\mu + \delta$ as possible.

The algorithm 3 first generates a random sequence of integers in the range $\mu - \delta$ to

Algorithm 3 Sequence generation

Input: $(g1, m, g1\_deg\_seq, \mu, \delta)$
Output: $g2\_deg\_seq$

```
s1 ← sum(g1_deg_seq)
seq ← sequence of m random integers between μ − δ and μ + δ
while sum(seq)! = s1 do
    if sum(seq) > s1 then
        indices ← []
        for deg in seq do
            if deg > μ − δ then
                indices push deg
            end if
        end for
        if length of indices == 0 then
            for deg in seq do
                if deg > 0 then
                    indices push deg
                end if
            end for
        end if
        i ← random entry of indices
        seq[i] ← seq[i] − 1
    else
        indices ← []
        for deg in seq do
            if deg < μ + δ then
                indices push deg
            end if
        end for
        i ← random entry of indices
        seq[i] ← seq[i] + 1
    end if
end while
```

$\mu + \delta$. If the sum of this sequence is too big, it decrements random entries until all degrees are equal to $\mu - \delta$. If this is still too large, random entries are decremented further until the required sum is reached. If the initial sum of the sequence is too small, random entries are incremented until the required sum is reached. It is always possible to reach the sum without any element of the sequence being greater than $\mu + \delta$, because the length of this sequence will always be smaller than or equal to the sequence of the first group. Thus, the maximum sum of the $g1_{seq}$ is $n \cdot (\mu + \delta)$ and the maximum sum of $g2_{seq}$ is $m \cdot (\mu + \delta)$. Since $n \leq m$ the following always holds: $n \cdot (\mu + \delta) \leq m \cdot (\mu + \delta)$. So the sequence of length m generated by the algorithm 3 will never be bigger than the maximum of degrees within the range $\mu - \delta$ to $\mu + \delta$.

The python implementation of this algorithm is shown in listing 4.4.

```
1  def _create_sequence_with_sum(self, size: int, _sum: int):
2      seq: List[int] = np.random.randint(self.min_deg, self.max_deg
           + 1, size=(size)).tolist()
3      while sum(seq) != _sum:
4          if sum(seq) > _sum:
5              if len(choices := [x for x in seq if x >
                   self.min_deg]) == 0:
6                  choices = [x for x in seq if x > 0]
7              seq[seq.index(random.choice(choices))] -= 1
8          else:
9              seq[seq.index(random.choice([x for x in seq if x <
                   self.max_deg]))] += 1
10     return seq
```

Listing 4.4: Algorithm for creating a sequence with specific sum

Since the second degree sequence is always constructed so that the two sequences together are graphic, the Erdos-Gallai Theorem is theoretically not needed here. It is still used to check the result in case there are any implementation errors that produce a non-graphic sequence, though it should always return true if the implementation has no problems. The modified implementation of the Erdos-Gallai Theorem can be seen in listing 4.5. This implementation tests the inequality for two sets of values: n, the size of $g1$, with $g2\_deg\_seq$ and m, the size of $g2$ with $g1\_deg\_seq$. This combination of values is used because the nodes of $g2$ have to be connected to the n nodes of $g1$ and vice versa. If the second degree sequence is constructed using the above algorithm 3, this function will always return true.

```
1  def _erdos_gallai(self) -> bool:
2      for n, seq in zip([self.size1, self.size2], [self.deg_seq2,
           self.deg_seq1]):
3          for k in range(1, n + 1):
4              if sum(seq[:k]) > k * (k - 1) + sum(min(k, d) for d
                   in seq[k + 1 :]):
5                  return False
6      return True
```

Listing 4.5: Modified Erdos-Gallai Theorem for connecting to disjunct groups

4.2.2 Modified Havel-Hakimi algorithm

The Havel-Hakimi algorithm from section 4.1.3 is modified to use two disjoint groups of nodes that it connects. The implementation of the modified algorithm can be seen in listing 4.6. Before this algorithm is used, the two degree sequences are constructed and sorted to be non-increasing. The two node id sequences are also created and shuffled.

```
1  def _connect_nodes(self):
2      while sum(self.deg_seq1) + sum(self.deg_seq2) != 0:
3          if self.size1 <= self.size2:
4              node = self.node_id_seq1.pop(0)
5              deg = self.deg_seq1.pop(0)
6              targets = self._get_highest_n_nodes(deg,
                   self.deg_seq2, self.node_id_seq2)
7              for target in targets:
8                  self.deg_seq2[self.node_id_seq2.index(target)] -=
                       1
9          else:
10             node = self.node_id_seq2.pop(0)
11             deg = self.deg_seq2.pop(0)
12             targets = self._get_highest_n_nodes(deg,
                   self.deg_seq1, self.node_id_seq1)
13             for target in targets:
14                 self.deg_seq1[self.node_id_seq1.index(target)] -=
                       1
15         self.edges[node] = targets
16         self._sort_sequence()
```

Listing 4.6: Modified Havel-Hakimi algorithm

The algorithm creates the edges starting from the smaller group. Let d be the highest degree of the smaller group. Then a node from the smaller group with degree d is selected for `node`. After that d nodes with the highest degrees are selected from the list of nodes of the **other** (bigger) group. This is the main difference from the original Havel-Hakimi algorithm, which selects the nodes from the same group. Then the `node` from the smaller group is connected to each selected node from the larger group and the degrees for all nodes `node` is connected to are decremented.

5 Network Display

Originally, it was intended to display the networks in 2D, with a 3D representation being considered a nice to have feature. During testing we found that a 2D representation does not provide a clear view of the network. Even for relatively small networks (~200 nodes) the 2D representation was very confusing. For this reason, the 2D representation was scrapped and only the 3D one was implemented. In 3D, even networks with thousands of nodes provide a clear view of the different groups and nodes.

The Python package plotly [8] is used to display the networks in 3D. `plotly.graph_objs` provides the `Scatter3d` function, which is used to draw both the nodes and edges. For the nodes the function takes in three lists of coordinates, one for each axis x, y and z. For the edges it also requires these three lists, but here the first coordinate is the start point for an edge, the second the end point, followed by a None entry and then the next edge coordinates. Currently, the only information that is generated for graphs is the number of nodes in each group and lists of which node ids must be connected with edges. This information must be translated into coordinates for plotly to be able to display the network.

5.1 Displaying the nodes

All nodes belonging to the same group should be arranged in a sphere. Each sphere thus represents one group and the spheres must have enough space between them so that the groups can be easily distinguished, as shown in Figure 5.1.

To distribute the spheres in the coordinate system, it is first divided into cubes. Let n_{max} be the number of nodes in the biggest group. To determine the side length s of the cubes, it is necessary to know the radius r of the smallest sphere that can fit n_{max} nodes.

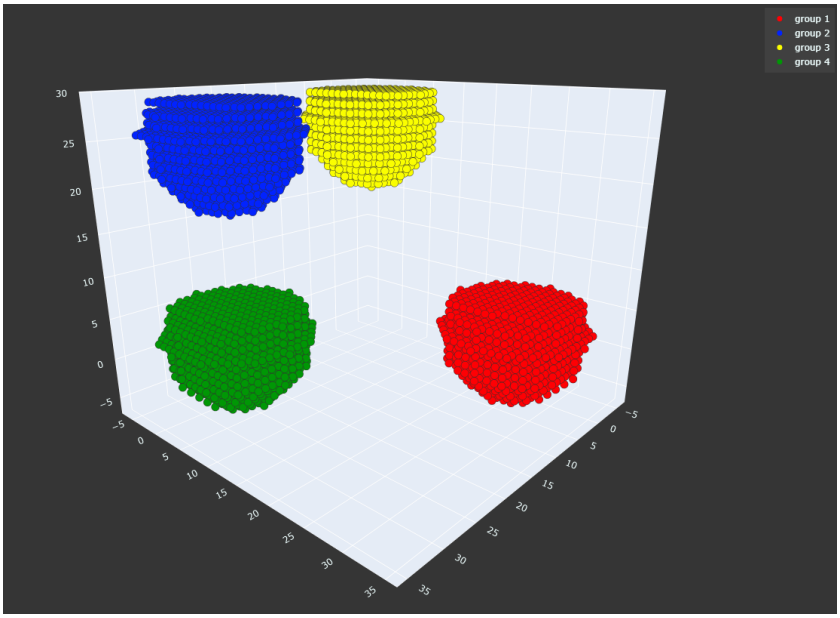


Figure 5.1: Network containing 4 groups with 2000 nodes per group

5.1.1 Creating a sphere

The nodes are placed only at coordinates $x, y, z \in \mathbb{N}$. To create a sphere that can fit n_{max} nodes, the radius needs to be known before the exact coordinates are calculated. The problem with this is that the only way to get the exact radius for a sphere that can fit n_{max} nodes that are all at coordinates $x, y, z \in \mathbb{N}$ is to use the algorithm depicted in 4. This algorithm is inefficient because it computes all spheres until r is

Algorithm 4 Calculating exact radius

Input: (n_{max})
Output: r
  $r \leftarrow 0$
  $n \leftarrow 0$
  while $n < n_{max}$ **do**
    $r \leftarrow r + 1$
    coords $\leftarrow$ calculate coordinates for all nodes in sphere with radius r
    $n \leftarrow$ length of coords
  end while

big enough. This can be improved by estimating the number of nodes in a sphere of radius r instead of computing the exact number. The number of nodes is estimated using the formula 5.1.

$$\frac{4 \cdot \pi \cdot r^3}{3} \tag{5.1}$$

This estimation incurs an error. 5.2 shows the best currently known formula for calculating this error which was discovered and proven by Zheng [14].

$$\sum_{\substack{x \in z^3 \\ |x| \leq r}} P(x) = O_{\epsilon,P}(r^{\upsilon+84/63+\epsilon}) \tag{5.2}$$

The true error bound is suspected to be $O_\epsilon(R^{1+\epsilon})$ but this is currently not possible to prove. For the context of this work the simpler estimation shown in formula 5.3 is sufficient.

$$O_\epsilon(r^{21/16+\epsilon}) \tag{5.3}$$

Let r_e be the resulting radius required for n_{max} nodes using the estimation. If n_{max} is close to the boundary between two radii, then the resulting radius may be too large or too small due to the introduced error. For this reason, the resulting radius is decremented by one and the exact calculation is then used to find the correct required radius, as shown in algorithm 5. This will always find the smallest radius for a sphere containing at least n_{max} points.

Algorithm 5 Calculating exact radius

Input: (n_{max})
Output: r
  $r \leftarrow 0$
  $n \leftarrow 0$
  while $n < n_{max}$ **do**
    $r \leftarrow r + 1$
    $n \leftarrow$ estimate amount of nodes in sphere with radius r
  end while
  $r \leftarrow r - 1$
  while len(coords) $< n_{max}$ **do**
    coords $\leftarrow$ calculate exact coords of nodes for radius r
  end while

The formula 5.4 represents the inequality for points inside or on the surface of a sphere centered at the origin in three-dimensional space, as proven by Gui and Moradifam [5]. A point is inside the sphere if this inequality is satisfied.

$$x^2 + y^2 + z^2 \leq r^2 \tag{5.4}$$

This means the bigger one of the values x, y, z becomes, the smaller the other must be to satisfy the inequality. Using this knowledge, it is possible to construct an algorithm (shown in 6) that efficiently computes all triples of x, y, z values that satisfy the inequality. The order of z, x, y in the algorithm is important, so a half full sphere creates points that fill the sphere from the bottom to the top. Starting with z and $y = 0$, transforming the formula to x results in $|x| \leq \sqrt{(r^2 - z^2)}$, which

corresponds to all possible values of x with respect to z and r. This range corresponds to $-a, a$ in the algorithm. Solving for y then results in $|y| \leq \sqrt{(r^2 - z^2 - x^2)}$, which yields all possible values for y with respect to x, z, r. This coincides with $-b, b$ in the algorithm.

Algorithm 6 Calculating lattice points

Input: r
Output: *coords*
```
coords ← []
for z := −r to r do
    a ← ⌊√(r² − z²)⌋
    for x := −a to a do
        b ← ⌊√(a² − x²)⌋
        for y := −b to b do
            coords ← (x, y, z)
        end for
    end for
end for
```

The Python implementation of this function takes in an amount of points n and returns a list of triples of length n containing the resulting coordinates and the radius r of the sphere.

5.2 Arranging the spheres

After calculating the coordinates and radii for each sphere (one for each group), the biggest radius r_{max} is known. Using this, the side length of the cubes can be calculated. To create some space between the spheres, this side length s is calculated as $s = 2 * r_{max} * 1.25$ to add an empty space of 12.5% of the sphere diameter on each side.

The cubes are always created within a larger cube, e.g. the number of cubes is $c = x^3$. x is the side length of the bigger cube in cubes (e.g. a side length of $x = 3$ cubes results in $c = 27$ smaller cubes). In order not to spread out the spheres unnecessarily, the smallest number of cubes should be created. The minimum required number of cubes is the number of spheres/groups. To get the next higher value for c, x can be calculated by $x = \lceil \sqrt[3]{num\_groups} \rceil$.

Using this the algorithm 7 computes the coordinates of bottom left corner of each cube.

For each sphere a random cube is selected and its offsets are added to the coordinates of all points within the sphere, resulting in the final coordinates for all nodes in the

Algorithm 7 Calculating cube offsets

Input: num_group, r_{max}
Output: *offsets*
$s \leftarrow \lceil 2 * r_{max} * 1.25 \rceil$
$o \leftarrow \lceil 2 * r_{max} * 0.125 \rceil$
$c \leftarrow \lceil \sqrt[3]{num\_groups} \rceil$
for $x, y, z := 0$ to c **do**
offsets $\leftarrow (x \cdot s + o, y \cdot s + o, z \cdot s + o)$
end for

network.

5.3 Creating the scatter graph

In the final graph it must be possible to toggle the visibility of each group. To achieve this, the coordinates for each group are stored separately and only the active ones are added to the coordinate lists for the `Scatter3d` call.

```
for group in self.network.groups:
    if group.id not in self.hidden_groups:
        x, y, z = zip(*self.group_coords[group.id])
        aXn.extend(x)
        aYn.extend(y)
        aZn.extend(z)
        colors.extend([group.color] * len(x))
trace1 = go.Scatter3d(
    x=aXn,
    y=aYn,
    z=aZn,
    mode="markers",
    name="nodes",
    marker=dict(
        symbol="circle",
        size=6,
        color=colors,
        line=dict(color="rgb(50,50,50)", width=0.5),
    ),
    uirevision="0",
    showlegend=False,
)
```

Listing 5.1: Creation of the node trace

The code in 5.1 creates the trace that displays all active nodes. Whenever the visibility of a group is toggled, this trace must be rebuilt. It is important to note that

the coordinates for the nodes in the x,y,z arrays contain the nodes in ascending order by their id. This means that nodes of group 0 come first starting with node 0, then group 1, and so on. This is important for creating color sequences based on the node states, which is discussed in section 6.1.

5.4 Displaying the edges

The edges are displayed using a second `Scatter3d` with the `lines` mode. For this the x,y and z arrays need the format of [xStart, xEnd, None] for each line. This means that a coordinate array for 100 lines would have a length of 300.

The current format in which the edges are represented contains only the node ids that are connected by edges. This representation was created in section 4.1.3. To translate these node ids to the coordinates of the nodes calculated in section 5.1, a map is created when the coordinates are calculated for each node. This map stores the node id as a key and the index of its coordinates, in the coordinate array, as value. This map can be used to calculate the start and end coordinates for all edges of visible groups, which is done by looking up the coordinates of the start and end nodes by using their ids and the map. The python code that realizes this can be seen in listing 5.2.

```
1  for edge in edges:
2      _from, to = edge[0], edge[1]
3      if not (_from in node_id_map and to in node_id_map):
4          continue
5      from_ind = node_id_map[_from]
6      to_ind = node_id_map[to]
7      aXe.extend([node_coords_x[from_ind], node_coords_x[to_ind],
           None])
8      aYe.extend([node_coords_y[from_ind], node_coords_y[to_ind],
           None])
9      aZe.extend([node_coords_z[from_ind], node_coords_z[to_ind],
           None])
```

Listing 5.2: Creation of the edge coordinate arrays

The trace is then created using the `Scatter3d` call from listing 5.3.

```
1  trace2 = go.Scatter3d(
2              x=aXe,
3              y=aYe,
4              z=aZe,
5              mode="lines",
6              uirevision="0",
7              line=dict(color="rgb(125,125,125)", width=1),
8              hoverinfo="none",
```

```
9               showlegend=False,
10          )
```

Listing 5.3: Creation of the edge trace

6 Visual Simulation

To simulate epidemics in a network, each node needs to store its state. As explained in 2.4 each node can have one of the following states: healthy, cured, infected, vaccinated, deceased. The current state is stored in the node together with the disease it is infected with in case of the state `infected`. The number of cycles the node has been infected for and the total number of times the node has been infected are also stored. Each node also contains all nodes it is connected to.

With this information the simulation can be performed using the algorithm in 8.

Algorithm 8 Simulate epidemics

```
Input: nodes
  for node in nodes do
    if node is not vaccinated then
      vaccinate according to vaccination chance of group
    end if
    if node is infected then
      if infection time > disease duration then
        kill or cure node according to rates of disease
      else
        infection time ← infection time + 1
        for each connected node that is not infected do
          infect node according to rate of disease
          (depending on vaccination status and previous infections)
        end for
      end if
    end if
  end for
```

6.1 Displaying the status

To indicate the current state of the simulation, the nodes are colored according to their state. Each state has a configurable color, with infected having different colors for each disease. Since nodes may have several states at the same time (eg.

vaccinated and infected), each state has a priority and only the color of the state with the highest priority is displayed. A lower number indicates a higher priority.

1. Infected/Deceased
2. Vaccinated
3. Healthy/Cured

This priority list can be used to generate a sequence of colors from all node states. This sequence is generated for the nodes in ascending order of their ids. This is important because the `Scatter3d` graph requires the colors in the same order as the coordinates, which are also in ascending order as mentioned in section 5.3. The color of the graph can then be updated using the `update_trace` method as seen in listing 6.1.

```
1 fig.update_traces(selector=dict(name="nodes"),
      marker=dict(color=state_colors))
```

Listing 6.1: Updating the colors of the graph

7 Dash Website

The Python package plotly [8] is used to display the network. Usually this package allows for the creation of figures that are then displayed in Jupyter notebooks, an extra window or saved as PNGs. However, if only plotly is used, there is no way to edit a created figure without completely regenerating it, which would then open a new window that displays the figure. For the use case of this app, the figures must be able to be updated within the same window so the simulation and different settings for viewing the networks are possible.

To accomplish this, the figures plotly creates can be hosted on a webserver, which is then able to update the network that the website displays. Plotly offers a built-in website framework called dash. The dash app is created using the statement in listing 7.1. The CSS of the website uses the dash bootstrap CSS as a base and font awesome [4] is used for the icons. The assets folder is set because the website uses some custom CSS located in that folder, that overrides some of the default dash bootstrap CSS.

```
1 app = Dash(
2     external_stylesheets=[dbc.themes.BOOTSTRAP,
          dbc.icons.FONT_AWESOME],
3     assets_folder=os.getcwd() + "/assets",
4     suppress_callback_exceptions=True,
5 )
6 app.layout = dash.html.div(id="page-content")
7 app.run()
```

Listing 7.1: Instatiate a new Dash app

After calling `app.run()`, this app automatically starts a flask server that hosts the website at `http://localhost:8050`. To specify what the website displays, the `layout` property of the `app` is set to the desired HTML components. Plotly provides HTML components as Python classes as well as some custom bootstrap components that are more advanced HTML elements like a sidebar. In this case, a simple div with the id `page-content` is used. The children of this div are then changed depending on which page needs to be displayed.

The webapp consists of three different pages:

- **View**: Displays a 3D network and offers some controls to alter the appearance

- **Simulation**: Displays the same network as the view does with the same controls and offers functionality to run the simulation in that network.
- **Stats**: Shows stats collected during the simulation with controls for which stats to display

To change between these pages, three routes are created for the app so each page has its own URL: `/view`, `/sim` and `/stats`. To display the correct page for each URL, a callback is used to change the content of the `page-content` div. This callback is invoked whenever any URL of the webserver is accessed, it is shown in listing 7.2.

```
@callback(
            Output("page-content", "children"),
            [Input("url", "pathname")],
)
def display_page(pathname):
    if pathname == "/view":
        html_view.reset()
        return html_view.layout
    elif pathname == "/sim":
        sim_view.reset()
        return sim_view.layout
    elif pathname == "/stats":
        stats_view.reset()
        return stats_view.layout
    else:  # if redirected to unknown link
        return "404"
```

Listing 7.2: Create callback for page content

7.1 Network View

The website for the network view consists of a collapsible sidebar on the left that provides the controls for the appearance of the network. The rest of the page is used to display the network. Figure 7.1 depicts the UI of this page.

The sidebar contains buttons to visually disable or enable the grid, intra/inter group edges, group/status color or groups of nodes. The slider can be used to reduce the number of displayed nodes, which can help to improve performance for large networks. Each of these settings uses a callback to update the network. Listing 7.3 contains such a callback for toggling the grid. This callback updates the style of the button to change the background color, depending on whether the button is now enabled or disabled. The `update_graph_grid` edits the necessary settings in the plotly figure to enable or disable the grid and then returns the updated figure. The two return values of the `toggle_grid` function are passed to the specified outputs,

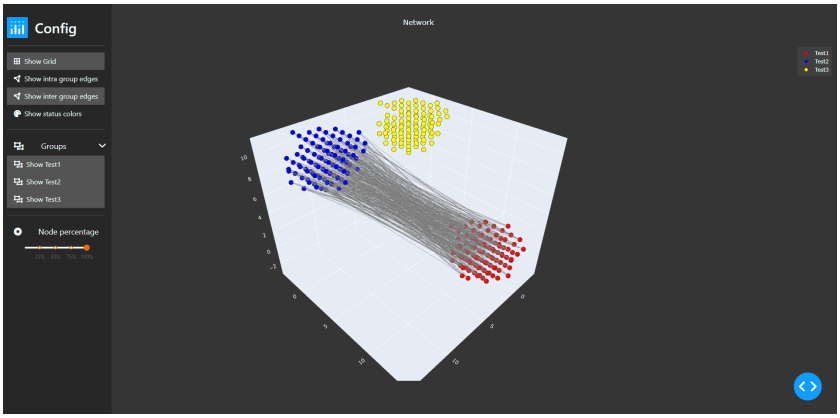


Figure 7.1: UI for the network view webpage

which contain the UI element ids for the button and network figure. By using these outputs, dash ensures that the changes are reflected on the website without having to reload it. Similar callbacks are created for the other buttons and the slider.

```
@callback(
    [
        Output(self.id_factory("grid-button"), "style"),
        Output(self.id_factory("live-graph"), "figure",
            allow_duplicate=True),
    ],
    [Input(self.id_factory("grid-button"), "n_clicks")],
    prevent_initial_call=True,
)
def toggle_grid(_):
    self.sidebar.show_grid = not self.sidebar.show_grid
    return {
        "background-color": self.ENABLED_COLOR
        if self.sidebar.show_grid
        else self.BACKGROUND_COLOR
    }, self.update_graph_grid(self.sidebar.show_grid)
```

Listing 7.3: Callback for toggling the grid

The plotly figure for the network, which makes up the rest of the webpage, offers some controls for viewing the 3D network. The camera can be rotated by holding down the left mouse button or moved by holding down the right mouse button. The zoom can be changed with the mouse wheel and in the top right corner there are buttons for these controls as well as a button to download the current view as a PNG. Chapter 5 explains how the network figure is created.

7.2 Simulation view

The simulation view contains the same UI as the network view. Its class inherits from the network view. Then 5 new buttons are added to control the simulation: reset simulation, show log output, advance one step, enable auto advancing (1 step per second) and save statistics. Using these five buttons as well as the sidebar to control the appearance of the network, the simulation can be run, updating the graph with new colors for the current state of each node after each step. This is done in the callback of the advance step button, as shown in listing 7.4, which executes one simulation step and then computes the new color sequence and log output. The updated graph and log console text are then returned to the outputs to update the website.

```
1  @callback(
2      Output(self.id_factory("live-graph"), "figure",
           allow_duplicate=True),
3      Output("log-console-content", "children",
           allow_duplicate=True),
4      Input(self.id_factory("step"), "n_clicks"),
5      prevent_initial_call=True,
6  )
7  def step(_):
8      with self.sim_mutex:
9          self.sim.simulate_step()
10         color_map, _ = self.sim.create_color_seq()
11         if self.show_logs:
12             return (
13                 self.graph.update_status_colors(color_map),
14                 self.sim.stats.get_log_text_html(),
15             )
16         else:
17             return self.graph.update_status_colors(color_map), ""
```

Listing 7.4: Callback for advancing simulaton by one step

The reset button opens a popup that requires confirmation from the user before the simulation is reset. The save button also opens a popup that allows the user to enter a name for the statistics of this simulation.

7.3 Stats view

The stats website displays statistics saved during simulations. When the website is opened for the first time, a popup window is displayed, allowing the user to select which statistics are to be displayed. This website contains a sidebar similar to the

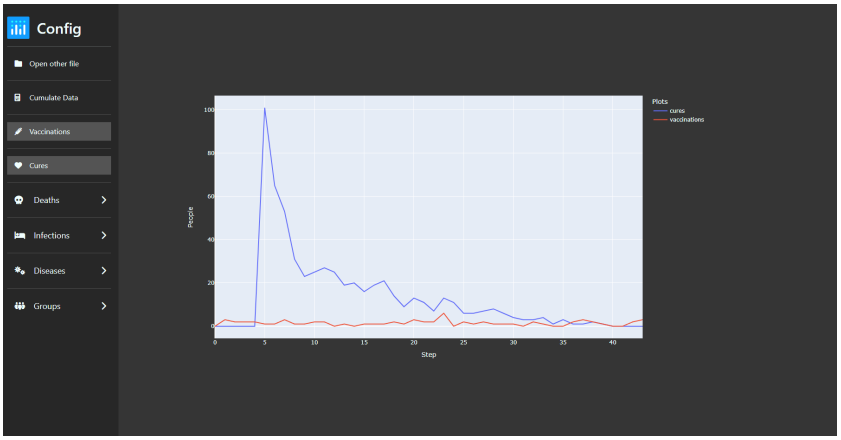


Figure 7.2: UI for the stats webpage

previous view and simulation pages. The sidebar contains buttons to control graphs for which statistics are displayed. The UI is shown in figure 7.2.

The sidebar contains the following buttons:

- Open another file: Brings the popup to select another stat file back up
- Cumulate data: Cumulates the graphs to show total up until each step instead of individual values of each step
- Display vaccinations, cures, deaths or infections data. Deaths and infections are split into total, vaccinated and unvaccinated
- Groups and diseases buttons allow to split the data to show only the values for a specific group/disease or the total for all diseases/groups

These buttons use callbacks to update the graphs similar to the ones for the network view.

7.4 Updating the data

Endpoints are created to update the network or available stats files that are displayed. The dash server allows the creation of listeners on URLs that are called when other applications send HTML POST requests to those URLs. To update the data, two endpoints are exposed, one for updating the network and one for updating the stat files. The creation of these endpoints is shown in listing 7.5. After a POST request is received, the JSON data it contains is decoded and stored for the stats or decoded, built and stored for networks. Then the view is reset to refresh all open

webpages. Finally, a response is sent for either success (200) or failure (400).

```
@app.server.route("/update-data", methods=["POST"])
def update():
    try:
        json_data = request.get_json()
        project.network = Network.from_dict(json_data)
        project.network.build()
        graph.update_network(project.network)
        html_view.reset()
        sim_view.reset()
        stats_view.reset()
        return make_response(jsonify({"status": "OK"}), 200)
    except Exception as e:
        return make_response(jsonify({"status": {str(e)}}), 400)

@app.server.route("/update-stats", methods=["POST"])
def update_stats():
    try:
        json_data = request.get_json()
        stats = SimStats.from_dict(json_data["stats"])
        project.stats[json_data["filename"]] = stats
        stats_view.reset()
        return make_response(jsonify({"status": "OK"}), 200)
    except Exception as e:
        return make_response(jsonify({"status": {str(e)}}), 400)
```

Listing 7.5: Endpoints for updating data

8 Qt

Qt is a framework that streamlines the development of high-performance, visually appealing and feature-rich GUI applications. It offers support for multiple languages such as C++, Python, and JavaScript, allowing developers to choose their preferred language. Thanks to the documentation and the large user base, there is a lot of support material to reference [1].

With the Qt Designer, developers can create GUI designs effortlessly. Designs can be created via drag-and-drop, which accelerates designing in Qt. It also enables developers to visualize what their application will look like in real-time [1].

The feature-rich and vast knowledge base, coupled with its user-friendly design, made it an easy choice to use in this project to develop the applications.

8.1 Qt Designer

This is a tool from the Qt framework that allows designing and building a GUI via drag-and-drop. It uses the what-you-see-is-what-you-get approach, where how it looks in the designer will be the same in the real application [1].

Figure 8.1 shows the Qt Designer application, which consists of a main window to which widgets can be dragged from the left sidebar to place them in the application. If a layout is selected for the widgets, Qt will automatically fit them according to the layout, so ensuring that the widgets are properly aligned. In applications, it is important to know the hierarchy of the widgets. Qt Designer makes this process easy to figure out, the object window (on the top right) shows the hierarchy tree of the widgets. Making a widget a child of another widget can be achieved by dragging it into the parent widget.

Not only can the content and placement of the widgets be adjusted, but it is also possible to set several properties of these widgets within Qt Designer (on the bottom right). This helps to see the changes immediately without having to start the GUI application. This is especially useful when changing the style or the layout of certain widgets, as these changes are immediately reflected in the designer.

Qt Designer makes Qt an attractive choice for developers looking to create a GUI

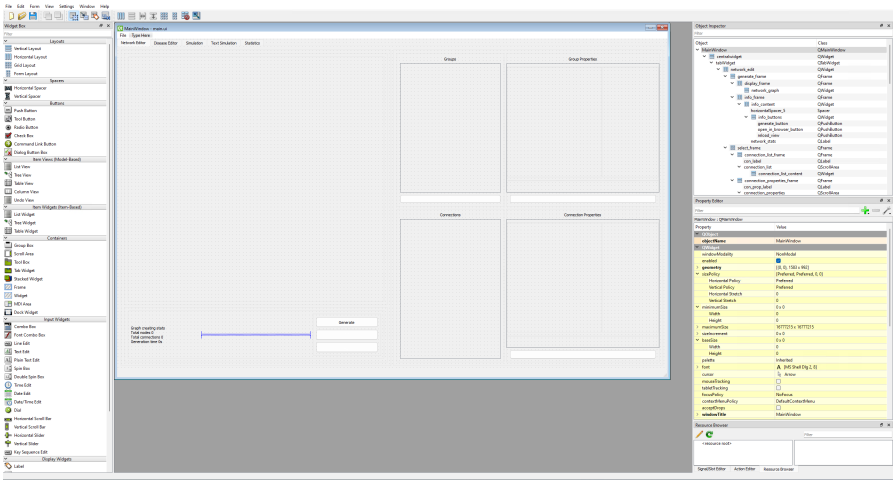


Figure 8.1: Qt Designer

application. The real-time visualization makes creating a GUI easy and efficient, making it possible to try out different looks of the application, resulting in a modern-looking design with minimal effort.

8.2 PyQt

Qt supports multiple programming languages to allow developers to choose their preferred language. This project was developed in Python, so PyQt was used. PyQt is a module that can be easily installed into the Python environment using pip. It contains all the add-ons that the C++ version of Qt also has. The add-ons can be used by importing the corresponding Python module.

To develop an application in PyQt, the GUI can either be created using only the Python commands or with the help of the Qt Designer. The GUI created in the Qt Designer is saved in a *.ui* file that can be loaded in the PyQt application [10]. Listing 8.1 shows how to load a `.ui` file into a simple application.

Once a design is loaded into the application, it is easy to access the widgets that were created in the Qt Designer. To access and modify widgets created in the Qt Designer, they can be referenced using the ID associated with the widget in the designer. In Listing 8.1, the widget created in the Qt Designer has the ID *myLabel*. Depending on the type of widget, different functions are available, in this case the widget is a label, which contains the *setText* function to change the text it displays. Other widgets, such as a frame widget, offer different functions.

After a design is loaded, it is possible to add more widgets to it. For example, if

buttons need to be created dynamically. This allows developers to create the default design of their application in the Qt Designer and dynamically change the content or arrangement of the widgets in the application. The ease of access to the objects created by the Qt Designer helps to improve the readability and simplicity of the application code.

```
1  from PyQt5 import QtWidgets, uic
2  import sys
3  class GuiApplication(QtWidgets.QMainWindow):
4      def __init__(self):
5          super(GuiApplication, self).__init__()
6          uic.loadUi('basic.ui', self)
7          self.myLabel.setText('Changed text')
8          self.show()
9
10 app = QtWidgets.QApplication(sys.argv)
11 window = GuiApplication()
12 app.exec_()
```

Listing 8.1: Simple Qt GUI application

8.3 Signals and Slots

In GUI programming, it is often necessary to execute functions when, for example, a button is pressed. Other frameworks use callbacks to achieve this functionality; Qt uses the signals and slots mechanism. The signal is emitted by an object when its internal state changes. The slot is the function that will be executed once the corresponding signal is emitted. A signal can have multiple slots connected to it, which are executed one after the other. The signal-slot mechanism is independent of the GUI event loop and is executed immediately after a signal is emitted [1].

Each signal has a *.connect(Slot slot)* function that is used to connect the signal to a slot. For example, a button in Qt has three signals: *clicked*, *pressed*, and *released*. In order to connect a function to the button, the following command has to be used: *example_button.clicked.connect(slot_function)*. Now, when the button is clicked by the user in the GUI, the *slot_function* is executed [10].

In this project, the signals and slots feature is mainly used to connect button presses and allow different objects to communicate with each other. The reason why signals should be used for objects to communicate with each other is explained in section 8.3.1. Listing 8.2 contains a simple example from the project, that allows the user to start the network generation with a button press. To be able to start the generation process, the *clicked* signal of the button *generate_button* is connected to the method *start_generating*. When the user presses the button, the slot is executed and therefore the generation process is started.

```
class UiDisplayGroup(QObject):
    ...
    def connect_signals(self):
        self.generate_button.clicked.connect(self.start_generating)
        ...
    def start_generating(self):
        # Start the thread to generate the network.
```

Listing 8.2: Signals and Slots example from the project

8.3.1 Thread communication

In general, it is bad practice to have a thread modify GUI widgets to ensure separation of concerns. If this thread were killed prematurely, it could result in an unexpected GUI state that could lead to errors. Or, if the main thread changed the structure of the GUI widgets before the thread could add the desired widgets, this would also lead to errors. To solve this issue, the signals and slots mechanism is ideal for this situation. Instead of letting a thread handle the widget creation, the thread just sends the widget data back to the main thread, which then adds or modifies the widgets. This way, the integrity of the GUI can be maintained because the main thread retains control over widget manipulation, ensuring consistent and expected behavior [1].

An example of this approach is shown in listing 8.3. It demonstrates how the text simulation updates the label showing the current simulation speed. To achieve this functionality, three different classes were used:

UiTextSimulationTab: This runs in the main thread and changes the widgets.

SimulationWorker: This class executes the simulation in a separate thread.

WorkerSignals: This class only holds the signals that the threads can send.

First, an object for the class *WorkerSignals* is created and stored as *worker_signals* in the *UiTextSimulationTab*. The signals of this class are assigned to slots in the *connect_signals* method. Here, the *update_control_label_signal* is connected to the method *update_control_labels*. When a user presses the buttons that changes the simulation speed, a signal is sent that executes the *change_speed* method that has been connected to the slot. This method changes the simulation speed according to the action sent as parameter. After changing the speed, this method sends the signal *update_control_label_signal*. This signal contains the current simulation speed. The simulation speed sent by the signal is received by *update_control_labels*, which is executed in the main thread and can safely update the label containing the information. This way, only the main thread will change widgets.

```
class WorkerSignals(QObject):
    update_control_label_signal: pyqtSignal = pyqtSignal(float)
    ...
class SimulationWorker(QThread):
    ...
    def change_speed(self, action: str):
        # Changes the simulation speed according to the received
           action.
        self.signals.update_control_label_signal.emit(self.simulation_speed)
class UiTextSimulationTab:
    def __init__(self, parent: QtWidgets.QMainWindow):
        ...
        self.worker_signals = WorkerSignals()
        ...
    def connect_signals(self):
        # Connecting the button Signals and other signals from
           the worker_signals
        self.worker_signals.update_control_label_signal.connect(self.update_contro
    def update_control_labels(self, simulation_speed: float):
        # Updating the labels containing the simulation speed
           information.
```

Listing 8.3: Thread safety example

8.4 QSS

Like HTML, Qt supports a style sheet to specify how widgets should look. QSS was inspired by CSS (Cascading Style Sheets) and is therefore very similar in terminology and syntax. A style sheet can be set for each widget either directly in the Qt Designer, in the code or in a separate file. If widgets are created dynamically and are not present in the Qt Designer, it is not possible for the Qt Designer to set the style for these widgets. In this case, either a style sheet file is needed, or the style must be set in the program itself [1]. The style sheet file has the advantage that it contains all the styles in one place rather than scattered throughout the program, which improves maintainability. The syntax for a style sheet is shown in listing 8.4. In this example, the padding and spacing for the *QCheckBox* are changed. When a *QCheckBox* is checked, it will display an image with the specified width and height.

```
QCheckBox{
    padding: 0px;
    spacing: 0px;
}
QCheckBox::indicator:checked {
    image: url(assets:selected.png);
    width: 24px;
```

```
8       height: 24px;
9   }
```

Listing 8.4: QSS example

To load a style sheet, it must first be loaded and then set as the style in the main application. PyQt5 has a method *setStyleSheet* for every widget in Qt, which can be used to set individual styles for different widgets. It is also possible to set the style for the main application, which is used by all widgets in that application but is overridden by individual widget styles. Listing 8.5, shows how to set the style sheet as the main window style after successfully reading it [10].

```
1  class UiNetworkEditor(QtWidgets.QMainWindow):
2      ...
3      def load_window(self):
4          ...
5          with open("qt/NetworkEdit/style_sheet.qss", mode="r",
               encoding="utf-8") as fp:
6              self.stylesheet = fp.read()
7          self.setStyleSheet(self.stylesheet)
```

Listing 8.5: Loading the QSS file

8.5 Abblication views

The GUI application of the created app, contains five tabs/views, each with a different use:

Network Editor In this view, the user can create a network consisting of nodes and connections and view it in a 3D space.

Disease Editor Allows to create the disease that are used for the simulation in the created network.

Simulation The Simulation view embeds the webpage used for simulation. It displays the network and its state during the simulation, allowing the user to observe the behavior of the disease in the network.

Text Simulation If the network is too big or a large number of simulation cycles are required, which makes using the visual simulation impractical, this text-based simulation can be used instead. It provides the same functionality for controlling the simulation and saving statistics as the visual simulation.

Statistics This view embeds the webpage used to display the saved statistics. It can be used to view graphs for the various stats saved during simulations to help analyze the results.

9 Experiments

In this chapter some experiments will be conducted. These experiments are based on the basic principles such as R_0 value introduced in chapter 2. The goal is to recreate different situations using the developed tool and run a simulation with the described parameters. These simulations can then be used to support or refute the assumptions made about how the diseases should theoretically behave in these scenarios.

9.1 Importance of R_0

The role of the reproductive number R_0 was discussed in section 2.2.1. It is the deciding factor for whether a disease is dying out ($R_0 < 1$) or able to live indefinitely ($R_0 > 1$). Calculating R_0 in complex social networks is very difficult, because the structure of the networks has a large impact on R_0 in addition to the characteristics of the disease. For these experiments, a very rough estimate of $R_0 = e_\mu \cdot p \cdot t_I$ is used, as this is sufficient to estimate whether the disease should die out in the experiment or live for a long time. Let e_μ be the average number of connections per node and t_I the time it takes for an infection to be cured, let N be the total number of nodes in the network and E the total number of edges, then $e_\mu = \frac{E}{N}$. p is the probability that a node will become infected if one of its neighbors has the disease.

9.1.1 The network

The network used in these experiments contains 3 groups of 5000 nodes each. Group 1 has 5 intra-group edges for each node, group 2 has 4 and group 3 has 3 intra-group edges. Between each pair of groups there are 2 edges per node. The resulting network can be seen in figure 9.1, it contains 15,000 nodes in total.

Let N_i be the number of nodes in group i, then the number of edges can be calculated using the following equation:

$$E = \frac{n_1}{2} \cdot 5 + \frac{n_2}{2} \cdot 4 + \frac{n_3}{2} \cdot 3 + \frac{n_1 + n_2}{2} \cdot 2 + \frac{n_1 + n_3}{2} \cdot 2 + \frac{n_2 + n_3}{2} \cdot 2 \qquad (9.1)$$

It is important to note, that each edge appears in both the nodes it connects, so the

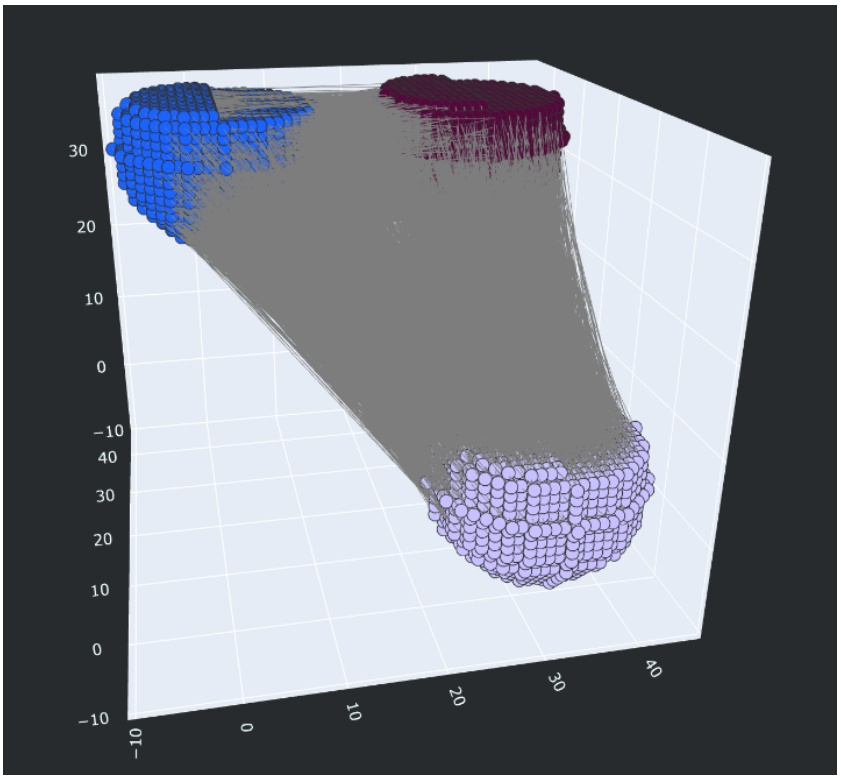


Figure 9.1: Structure of the network used for the R_0 experiments

number of edges is half of the sum of the degrees, which is why the number of nodes is divided by 2 in the equation. Using this equation, it can be calculated that the network contains 60,000 edges. This leads to an average amount of connections per node of $e\mu = \frac{60{,}000}{15{,}000} \cdot 2 = 8$. But the different groups must also be considered in isolation. If the R_0 value within a group of nodes is greater than one, the disease is likely to never die out in that group and the nodes in that group will constantly spread the disease to the other groups. An example of this is shown in the next section.

9.1.2 Experiment with $R_0 < 1$

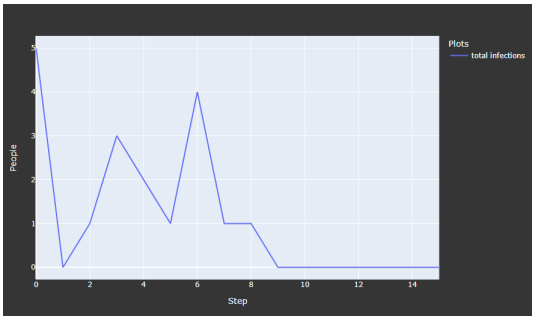
Since this experiment uses the same network as the following experiment, which will show an example of $R_0 > 1$, the factor e_μ is static and cannot be changed. This makes p the only deciding factor for R_0, it can be calculated using $p = \frac{R_0}{e_\mu \cdot t_I}$, so in this case with $t_I = 5$ and $R_0 < 1$, $p < \frac{1}{40}$. For cases with R_0 close to 1, there is still a good chance that the disease will die out even if $R_0 < 1$. $p = 0.024$ is chosen for an estimated $R_0 = 0.96$ to ensure that the disease dies out in a finite number of steps.

The parameters for the disease of this experiment are:

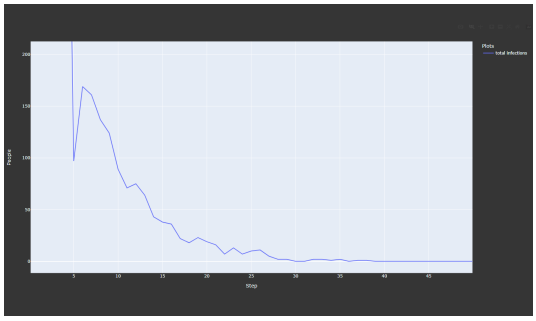
- (vaccinated) fatality rate: 0
- (re-, vaccinated) infection rate: 0.024
- infection period t_I: 5
- initial infections: 5
- cure chance: 1
- immunity period: 0
- infectiousness factor: 1

Running the simulation shows that the disease is never able to spread to a large number of people and dies out after only ~9 steps. This can also be seen in the first graph of figure 9.2, which shows the number of new infections per cycle. Increasing the initial number of infected nodes to 5,000 shows that even with such a large number of infections the disease still dies out after only ~40 steps

This experiment supports the theory that for a $R_0 < 1$ the disease will die out in a finite amount of steps, even for R_0 values close to 1.



(a) Infection counts with 5 initial infections



(b) Infection counts with 5000 initial infections, showing that the disease still dies out after ~40 steps

Figure 9.2: Experiment with an estimated R_0 of 0.96 and $p = 0.024$

9.1.3 Experiment with $R_0 > 1$

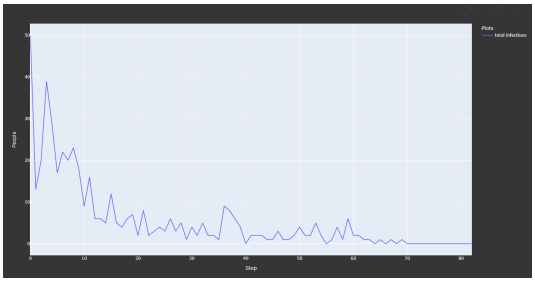
This experiment uses the same network as the previous one, so p is again the deciding factor for R_0. $p = \frac{R_0}{e_\mu \cdot t_I}$ so for this case with $R_0 > 1$, $p > \frac{1}{40}$. The theory created said that it should be sufficient to have a p big enough to ensure that $R_0 > 1$ in only one of the groups. Since group 1 has the most connections, $R_0^1 > 1$ if $p > \frac{1}{9 \cdot 5} = 0.0222$. Since the closer R_0 to 1, the higher the chance that the disease will randomly die out, $p = 0.03$ is chosen, resulting in a $R_0^1 = 1.35$ and $R_0 = 1.2$, where R_0^1 is big enough for the disease to have a decent chance of surviving indefinitely.

The properties that were changed from the previous experiment are:

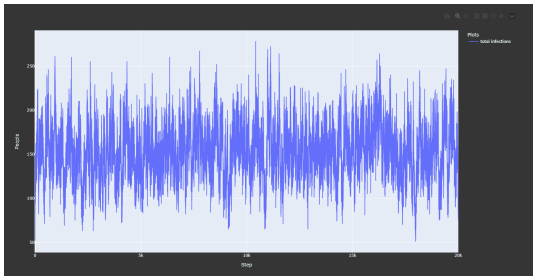
- (re-, vaccinated) infection rate: 0.032
- initial infections: 50

Running the simulation shows that the disease consistently dies out at about 70 steps. The left graph in figure 9.3 shows the number of infections. The expectation was that the disease would have a decent chance of surviving, but even after 50 attempts it never survived. This can be explained by the network structure: the nodes in group 1 have 9 connections, but 4 of them are "worth less" because the nodes it can spread to in the other groups are not able to spread the disease to another 9 new nodes, but only 7 or 8 depending on the group. This means that when calculating R_0, the nodes in group 1 cannot be considered to have 9 connections, somewhere around 8 would be more accurate. This makes $R_0^1 = R_0 = 1.2$ which is not high enough for a decent chance of indefinite survival. This shows how difficult it can be to calculate R_0 in complex networks, because there is no formula that can take the network structure into account.

Increasing p to 0.036 is enough to make the disease live indefinitely with $R_0 = 1.44$.



(a) Infection counts with 50 initial infections, $R_0 = 1.2$ and $p = 0.032$.



(b) Infection counts with 150 initial infections, $R_0 = 1.44$ and $p = 0.036$

Figure 9.3: Experiment showing that diseases with R_0 close to 1 still have a decent chance of dying out. Although it is theoretically possible even for the disease with R_0 to survive indefinitely, it is statistically improbable because the chance it dies out each cycle is too big.

After running the simulation for 20,000 cycles, the disease is still alive and infecting new nodes. This supports the theory that with $R_0 > 1$ there is a probability > 0 that the disease never dies out. Another option is to increase the connections within group 1, which increases R_0^1 so that the disease is able to survive within group 1. This again shows that in addition to viewing the network as a whole, each part of the network must also be viewed in isolation. If the disease is able to survive in a subset of the network, it will never die out and will constantly spread to other parts of the network, even though it would never be able to survive in the other parts of the network.

Changing the network

To prove the above assumption that the network structure does have an effect on R_0, this experiment uses the same disease as the previous one with $p = 0.036$ and 150 initial infections. The network will have only half as many edges, group 1 has 2 intra-group connections for each node, group 2 has 2 and group 3 has 2 intra-group connections. Between each pair of groups there is 1 connection per node. This new network has 30,000 edges, resulting in e_μ = 4. Thus the new estimated $R_0 = 2 \cdot 0.375 = 0.75$ which is smaller than 1 so the expectation is that the disease will die out even though it has the same infection rate as before.

After 20 cycles, the disease has died out, supporting the theory that the network structure, and thus e_μ, has an effect on the spreading of diseases and the value R_0. Graph 9.4 shows the number of infections until the disease died out.

To support the theory from earlier that each group must also be viewed in isolation when calculating R_0, the network is changed again. Group 1 has 8 intra-group

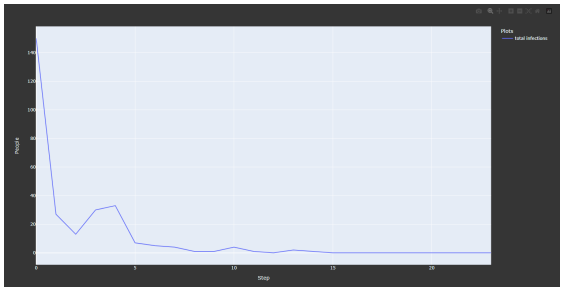


Figure 9.4: Amount of infections with the same diseases ($p = 0.036$) but a network with half as many connections

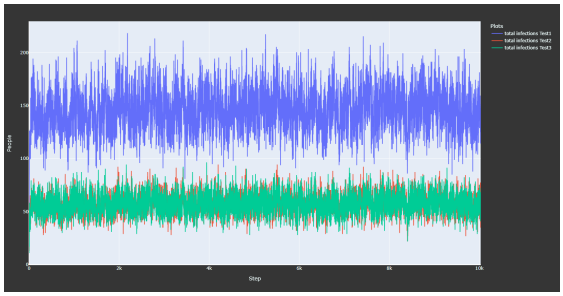


Figure 9.5: Infections per group with $p = 0.036$ and $R_0 = 0.96$, the disease never dies out

connections for each node, group 2 has 1 and group 3 has 1 intra-group connection. Between each pair of groups there is 1 connection per node, for a total of 40,000 edges, giving $e_\mu = 5.\overline{3}$ and $e_\mu^1 = 8$ for connections within group 1. The same $p = 0.036$ is used, resulting in $R_0 = 0.96$ while $R_0^1 = 1.44$. Running the simulation again shows that even though $R_0 < 1$ the disease manages to survive indefinitely. $R_0^1 = 1.44$ means that the disease has a chance of never dying out within group 1 and is constantly spreading from group 1 to the other groups. Figure 9.5 shows the number of infections per group. Group 1 has significantly more infections and keeps the disease alive, while groups 2 and 3 have similar, lower amounts of infections. This experiment shows how difficult it is to estimate R_0 for complex networks, as the network may contain highly connected subgroups where the disease can live longer and spread to the rest of the network again.

9.2 Multiple Diseases

Interesting dynamics can be observed when multiple diseases are introduced into the same social network. Cases with two diseases, where one or both of the diseases have a $R_0 < 1$, do not present an interesting scenario, since once one disease has

died out, the problem is reduced to one disease. But if both (or more) diseases have a $R_0 > 1$, then they will compete against each other for survival, assuming that a person can only be infected by one diseases at a time, e.g. because they stay at home until they are cured, so they cannot be infected by another disease during that time. Since a disease must constantly infect new nodes to stay alive, the diseases can wipe out other diseases by infecting all the nodes themselves, leaving no nodes available to infect for other diseases.

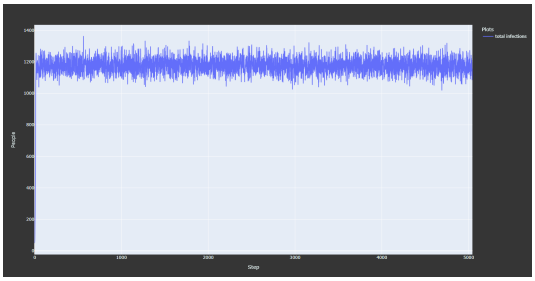
For diseases with a similar R_0, the two diseases will each have ~50% of infections. Now consider a case where one disease d_1 has $R_0^1 = 5$ and the second d_2 has $R_0^2 = 2$. Looking at the diseases in isolation, both are theoretically able to survive, since both $R_0^1 > 1$ and $R_0^2 > 1$. Looking at the situation where both diseases are in the same network, d_1 will infect significantly more people than d_2, about 2.5 times as many. This means that d_1 will have $\frac{5}{7}$ of the total infections during the first wave. Since d_1 has many more infected nodes during the first wave, that makes it even easier for it to spread than d_2. At the boundaries where susceptible nodes have connections to nodes infected with d_1 and also to nodes infected with d_2, d_1 will infect $\frac{5}{7}$ of those nodes. This will slowly reduce the number of nodes infected with d_2 until none are left and d_2 has died out, even though its $R_0^1 > 1$. This means that in networks with more than one disease, the one with the highest R_0 value is the most likely to survive.

9.2.1 Experiment

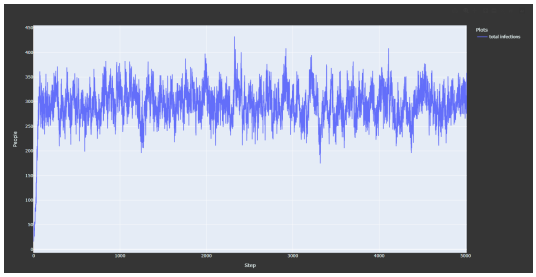
This theory can again be supported by conducting an experiment using the developed simulation tool. The network will be the same as in the previous experiment, consisting of 3 groups with more intra-group connections than inter-group connections. The exact structure is explained in section 9.1.1.

Two diseases are used for this simulation:

- (vaccinated) fatality rate: 0
- (re-, vaccinated) infection rate: 0.06 for Disease 1 / 0.04 for Disease 2
- infection period t_I: 5
- initial infections: 50
- cure chance: 1
- immunity period: 0
- infectiousness factor: 1



(a) Infection counts for disease 1 with 50 initial infections and $p = 0.06$



(b) Infection counts for disease 2 with 150 initial infections and $p = 0.04$

Figure 9.6: Experiment to show both diseases never die out on their own

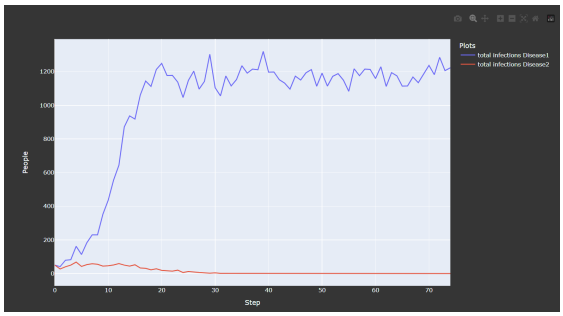


Figure 9.7: Infections per disease with $p = 0.06$ and $p = 0.04$, disease 2 dies out after 40 steps even though it did not die out when the two diseases were simulated in isolation

First, the experiment is run with the two diseases separately to show that they never die out on their own. The results can be seen in figure 9.6, which shows the number of new infections up to cycle 5000, indicating that the individual diseases have not died out by then.

Now both diseases are used simultaneously. After 40 cycles, d_2 has died out because the new infections of d_1 have increased so much that there are not enough nodes available for d_2 to infect. This increase in infections with d_1 and decrease in infections with d_2 is shown in figure 9.7.

This supports the theory that in networks with multiple highly infectious diseases the one with the highest R_0 value is the one most likely to survive.

9.3 Small-World Phenomenon

The small-world phenomenon is also known as six degrees of separation. The theory is that in an arbitrarily large social network the length of the path from one person to any other person is, on average, only 6 steps. To support this theory, Stanley Milgram conducted an experiment in the 1960s [9]. In this experiment, Milgram gave a letter to a random source person in Nebraska, USA, and told them to deliver it to a target person in Massachusetts, USA. The source person was given only basic information about the target, such as address and occupation and each person was allowed to send the letter only to someone they knew on a first-name basis, with the goal of getting the letter closer to its target. Each person in the chain was given the same information. After many iterations of this experiment, the average number of persons it took to deliver the letter to its target was between 5 and 6 persons, hence the name of the *Six Degrees of Separation* principle.

This experiment does not always find the shortest path, because people forward the letter only to the person they think is closest to the target. Unknown to them, there may be another person they know who is much closer to the target, making for a shorter path that is never discovered. To find the true shortest path, each participant would have to forward the letter to all of their friends, keeping track of which path the letter took. Then, when all letters have arrived at the target, the true shortest path is that of the letter that required the least number of people to arrive.

9.3.1 Experiment

For this experiment a network with only one group is created. This group contains 10,000 nodes, with each node having 6 connections. Although this network is not completely accurate to a real situation. Usually a person's network of friends is tightly coupled, a person's friends usually know each other and their friends know the original person etc. Also, a person tends to know more people who live in an area close to them and fewer people who live farther away. This leads to a highly connected network with only short connections and a few random longer connections. The network used for this experiment uses only random edges, which does not represent the fact that the geographically closer people are, the more connections they have. However in the current day, the internet allows for many more long-distance connections, which makes the used network used more valid in this context.

To show that each node can be reached in only six steps, a disease with an infection rate of 1 and an infection duration of 1000 is used. Initially, only one node is infected. Figure 9.9 shows the network after 6 cycles.

After cycle 7, all nodes are infected, i.e. they could all be reached from the starting

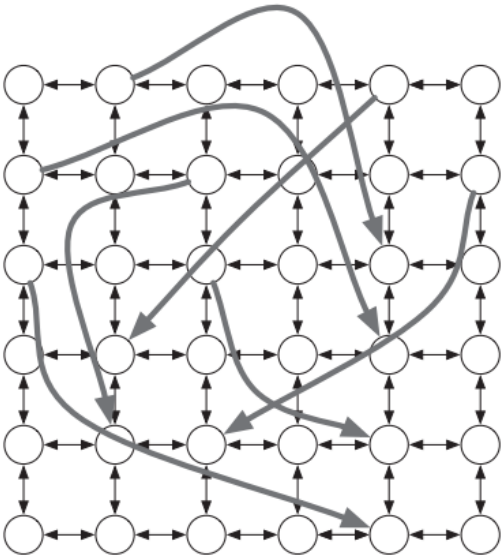


Figure 9.8: True structure of a network showing the friends of persons. The geolocically closer people are the higher the amount of connections. There are only few connections over longer range. (source: [2])

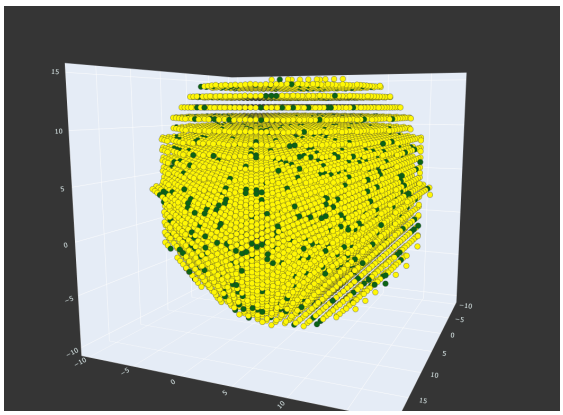


Figure 9.9: Network after 6 cycles, yellow nodes have a shortest connection with 6 or less steps to the starting node, green ones have a shotest connection with more than 6 edges to the starting node

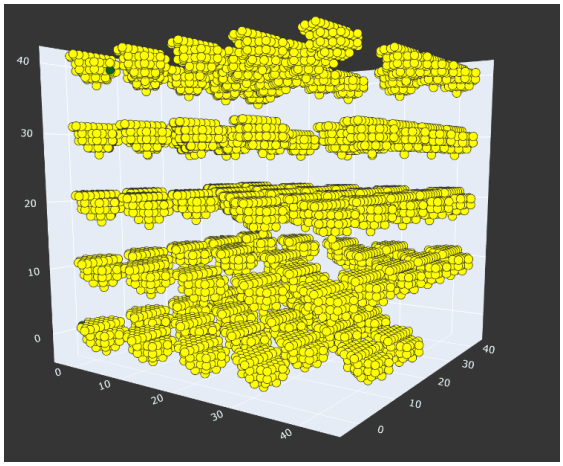


Figure 9.10: Network after 6 cycles, yellow nodes have a shortest connection with 6 or less steps to the starting node, green ones have a shotest connection with more than 6 edges to the starting node

node in a maximum of 7 steps. The exact numbers of new infections per step are: 6, 30, 150, 717, 2893, 5306 and 897. Using these values, the average shortest path length can be calculated, which is ~5.5964. This coincides with the findings of Milgram [9], whose experiments showed that the average length is somewhere between 5 and 6.

To reduce the limitation of the used network in terms of random connections, another network structure could be used. This more closely models the existence of highly connected local friend groups, where everyone knows each other and a few random connections to people from other friend groups that are farther away. This network uses a large number of groups with relatively few members each. In this case, 100 groups of 5-10 people are used, where each node is connected to 4 others of the same group and each group of people is connected to 3-4 other groups, with only 0-1 edges per node. This again results in a network where each node has an average of 6 connections.

The same disease is used in this network and the state of the network after cycle 6 can be seen in figure 9.10.

This shows that in this network it is also possible to reach all other nodes in only six steps, even though there are only a few long-distance connections and a lot of tightly coupled small groups. The average path length in this experiment was ~5.6621.

10 Summary and Outlook

10.1 Achieved Results

In chapter 2 the theory of epidemics was introduced. The two main parts of epidemics were discussed, networks and diseases. Three existing models and their respective limitations were explained, a simple tree-based model, the SIR Model and the SIS Model. These were combined and extended to create the custom model used in this work. This custom model was explained in detail in section 2.4.

An app was created that uses the custom model to allow for simulations of epidemics. The app includes functions for modeling social networks, creating diseases, running simulations with the created networks and diseases and collecting statistics about the simulation. The general functionality of the app has been outlined in chapter 3 and its implementation is described in chapters 4, 5 and 6, which explain the process and algorithms involved in creating the app.

After creating the app some experiments were conducted to support the theories discussed in chapter 2. These experiments were performed using the developed app. The results of the simulations coincided with the theories of how the diseases are expected to behave. This indicated the importance of factors like the reproductive number R_0 in epidemics. The findings of these experiments and theories can be used to better prepare for real-world scenarios of epidemics. They help to understand the behavior of diseases and their lifetime in complex social networks.

10.2 Outlook

10.2.1 Extensibility of the Results

The network editor can be extended to allow precise control over which nodes are connected to which. This requires a solution that still allows fast creation of large networks while providing such fine control over the connections.

For very large networks, the computation can take a long time. Tests showed that networks with ~100,000 nodes took almost 30 minutes to generate. A big

factor for this long time is that Python is an interpreted language, which makes such calculations relatively slow compared to the same calculations in a compiled language like C++. The app could be reimplemented in C++ which would reduce the computing time, but since C++ does not provide a package like plotly, a custom implementation for displaying the graphs is necessary, which can be time consuming to implement.

10.2.2 Transferability of the Results

The results achieved using the simulation can be applied to real-world scenarios. These simulations can help to predict how an epidemic will evolve over time. This can be used to test how certain factors affect the spreading of the disease to determine what countermeasures are necessary to suppress the disease and ensure the safety of the population. One example might be reducing the number of connections in the network, which could be achieved in the real world by implementing social distancing measures. Another factor that can be tested is whether a vaccine that reduces the infection probability by a certain amount is sufficient to make the disease die out or what percentage of people would need to be vaccinated to achieve this.

Without tools to simulate the behavior of diseases, it is almost impossible to predict what will happen, making it difficult to decide on the necessary countermeasure until it might be too late. This is why tools for simulation are necessary in the field of epidemics.

Bibliography

[1] The Qt Company. *QT UI Framework*. 2024. URL: https://www.qt.io/ (visited on 01/16/2024).

[2] David Easley and Jon Kleinberg. *Networks, crowds, and markets*. Cambridge, England: Cambridge University Press, June 2012.

[3] Péter L. Erdős, István Miklós, and Zoltán Toroczkai. *A simple Havel-Hakimi type algorithm to realize graphical degree sequences of directed graphs*. 2010. arXiv: 0905.4913 [math.CO].

[4] Inc. Fonticons. *Font Awesom*. 2024. URL: https://fontawesome.com/ (visited on 01/16/2024).

[5] Changfeng Gui and Amir Moradifam. "The Sphere Covering Inequality and Its Applications". In: *Inventiones mathematicae* 214 (Dec. 2018). DOI: 10.1007/s00222-018-0820-2.

[6] Benjamin C Haller and Philipp W Messer. "SLiM 3: Forward Genetic Simulations Beyond the Wright–Fisher Model". In: *Molecular Biology and Evolution* 36.3 (Jan. 2019), pp. 632–637. ISSN: 0737-4038. DOI: 10.1093/molbev/msy228. eprint: https://academic.oup.com/mbe/article-pdf/36/3/632/27980602/msy228.pdf. URL: https://doi.org/10.1093/molbev/msy228.

[7] Richard R. Hudson. "Generating samples under a Wright–Fisher neutral model of genetic variation ". In: *Bioinformatics* 18.2 (Feb. 2002), pp. 337–338. ISSN: 1367-4803. DOI: 10.1093/bioinformatics/18.2.337. eprint: https://academic.oup.com/bioinformatics/article-pdf/18/2/337/48850502/bioinformatics_18_2_337.pdf. URL: https://doi.org/10.1093/bioinformatics/18.2.337.

[8] Plotly Technologies Inc. *Collaborative data science*. 2015. URL: https://plot.ly.

[9] Jon Kleinberg. "The small-world phenomenon: An algorithmic perspective". In: *Proceedings of the thirty-second annual ACM symposium on Theory of computing*. 2000, pp. 163–170.

[10] *PyQt - Python Wiki*. URL: https://wiki.python.org/moin/PyQt (visited on 02/10/2024).

[11] Paula Tataru et al. “Statistical Inference in the Wright–Fisher Model Using Allele Frequency Data”. In: *Systematic Biology* 66.1 (Aug. 2016), e30–e46. ISSN: 1063-5157. DOI: 10.1093/sysbio/syw056. eprint: https://academic.oup.com/sysbio/article-pdf/66/1/e30/24194717/syw056.pdf. URL: https://doi.org/10.1093/sysbio/syw056.

[12] Tat Dat Tran, Julian Hofrichter, and Jürgen Jost. “An introduction to the mathematical structure of the Wright–Fisher model of population genetics”. In: *Theory in Biosciences* 132.2 (June 2013), pp. 73–82. ISSN: 1611-7530. DOI: 10.1007/s12064-012-0170-3. URL: https://doi.org/10.1007/s12064-012-0170-3.

[13] LIZZIE WADE. “From Black Death to fatal flu, past pandemics show why people on the margins suffer most”. In: (2020).

[14] Fan Zheng. *Gauss Sphere Problem with Polynomials*. 2013. arXiv: 1312.0108 [math.NT].