Technische Universiteit
Eindhoven
University of Technology

Department of Mechanical Engineering

Simulation of a random network contact process

H.H.C.M. van Wesel
D&C 2017.046

Supervisors:
T. Taimre
B. Patch
I.J.B.F. Adan

April 25, 2017

Abstract

Contact processes are used to model a virus or infection spread on various systems. Most research is done on static systems that do not change structure or only react to the infection while the infection spread is modelled. This research focusses on systems or networks that change structure while an infection tries to spread through the system. These systems contain points or nodes that can be infected by other infected nodes if there is a connection or edge between them. The changing structure of the system consists of adding and removing nodes and edges between nodes.

A large part of the research is focused on programming a discrete event simulation to simulate random network contact processes. The program can simulate different random networks that can implement various ways to calculate the rates which determine the time and type of the next event. Here, an event is a change of structure in the network or related to the spread of infection throughout the network.

Using simulation, different random networks were simulated and the results analyzed. Results show that adding or removing nodes affects the spread of an infection as it provides extra healing. Further research into different variations of random networks, more complex random networks, and different aspects and models for infection spread is recommended.

Contents

Nomenclature

$\mathcal{S}_t$	system state at time t
$\mathcal{C}$	state space
d	dimension
$\mathbf{V}$	set of nodes
$n = \lvert\mathbf{V}\rvert$	number of nodes
$\mathbf{v}_t;\ \lvert\mathbf{v}_t\rvert = n$	vector of state of nodes at time t
$v_t(x);\ x \in \mathbf{V};$	state of node x at time t
$\mathbf{E}$	set of edges
$m = \lvert\mathbf{E}\rvert$	number of edges ($m = n \times (n-1)$ for directional edges and $m = n \times (n-1) \times 0.5$ for non-directional edges)
$\mathbf{e}_t;\ \lvert\mathbf{e}_t\rvert = m$	vector of state of edges at time t
$e_t(y_{i,j});\ y_{i,j} \in \mathbf{E};\ i,j \in \mathbf{V};\ i \neq j$	state of edge y at time t ($y_{i,j} = y_{j,i}$ if edges are non-directional)
$\mathbf{e}_t^a$	vector of active edges at time t ($y_{i,j} \in \mathbf{e}_t^a$ if $y_{i,j} = 1$)
$l = \lvert\mathbf{e}_t^a\rvert$	level (number of active edges)
$\mathbf{Q}$	state transition matrix
λ	rate of nodes added to system
μ	rate of nodes removed from system
α	rate of infection spread
β	rate of infection healing
p	rate of edge state change
p^+	rate of adding edge (change the state of an inactive edge to active)
p^-	rate of removing edge (change the state of an active edge to inactive)
s	rate of swapping 2 edge states
s^+	rate of swapping state of an active edge with state of an inactive edge
SI	(Susceptible, Infected) model
SIS	(Susceptible, Infected, Susceptible) model
SIR	(Susceptible, Infected, Recovered/Removed) model

Chapter 1

Introduction

Contact processes are used to model infection or virus spread on all kinds of networks. These networks are often modelled after social environments to study the effects of certain viruses or infections on the population of a particular area, like a district or city. Contact processes are also more recently used to model computer networks. However most research on contact processes is carried out on static networks that in some cases react to the infection or virus, for example as a result of quarantine measures.

In contrast, this research focusses on random networks that change structurally over time. Random graphs and Markovian random graphs especially form the modelling basis of these evolving random networks. When making these random networks increasingly more complex they could potentially be used to model real life situations where systems are constantly changing. As a particular example, consider an infection that spreads through a shopping mall where people come and go and are in (brief) contact with lots of different people over time while they are in the shopping mall, and thus part of the system. Another example can be a virus spreading through a peer to peer computer network where computers make and break connections with other computers constantly.

This work constitutes a preliminary investigation into random networks and especially using these networks in a contact process. At the moment Thomas Taimre, Daniel Gibbons and Brendan Patch are working on Markovian random graphs [13], and much of the theory about the random networks in this report is derived from their work. Also a report from Brendan Patch [10] is used to explain much of the theory of contact processes, since his report forms a good summary and insight into the original contact process defined by Harris [4].

This research combines the theory of contact processes and the current work on Markovian random graphs to program a simulation of random network contact processes in C++. The simulation is a discrete event simulation that calculates the next event to happen and updates

the random network accordingly. This simulation is then used to generate some preliminary results into the effects of the changing structure of random networks on the infection spread through these systems.

In the next chapter, background theory on contact processes and (Markovian) random graphs are given. It also presents a brief look into discrete event simulations as well as some more recent research being done into contact processes and the simulation thereof. In Chapter 3 the theory on random networks is expanded to networks that allow for nodes to be removed or added, and to use these random networks in a contact process. Also the different ways to determine the rates that both change the structure of the network over time and the infection of the network are described here.

Chapter 4 discusses how the random network contact process is simulated using a *C++* program. The structure of the programming is outlined as well as the method of choosing and implementing events. Also the input and output options, the structure of the system and how the system is updated is explained. Next, Chapter 5 describes some chosen random networks that are simulated. The results of these simulations are analyzed to look at the survival of an infection for a number of variations of the random networks.

Finally, in Chapter 6 recommendations are made into further research of random network contact processes, both expanding on the results from this research and for more complex random networks. Some recommendations to improve and change the simulation to simulate these more complex networks are stated. Chapter 7 briefly summarizes and concludes this research followed by the Bibliography and Appendices.

Chapter 2

Background

This chapter introduces the concepts and basic theory of contact processes, random graphs, Markovian random graphs and discrete event simulations. Moreover some of the more recent research into contact processes is briefly discussed at the end of this chapter.

2.1 Contact process

A contact process is an interacting particle system where points (or nodes) that are connected by edges have 'contact'. A contact process can be interpreted as a model for infection or virus spread where a node can be infected at a rate dependent on the number of infected nodes it is connected with through edges and an infected node is healed at a constant rate [8].

The original contact process is defined as a d dimensional lattice of nodes (or vertices) [4]. The set of nodes is $\mathbf{V}$ and the number of nodes is $n = |\mathbf{V}|$. Each node has contact (or edges) with its neighboring nodes over which a virus or infection can spread. A node can be susceptible or infected. This leads to a system state defined as: $\mathcal{S} = \mathbf{v}_t$ where $\mathbf{v}_t$ is an n dimensional vector of the state of nodes $v_t(x)$, with $x \in \mathbf{V}$. An infected node infects each neighboring node with exponential rate α and infected nodes are healed with exponential rate β. When healed the node becomes susceptible again. This infection spread model is also known as the SIS (Susceptible, Infected, Susceptible) model [12]. A node $v_t(x)$ is in state 0 when susceptible and in state 1 when infected. This leads to the following state space of the contact network: $\mathcal{C} = \{0, 1\}^n$.

A virus can only spread through a contact process if there is at least one node infected from the start. All nodes that are infected at time 0 are part of the subset $\mathbf{V}^I \subset \mathbf{V}$. If a vector $\mathbf{v}_t^I$ holds all node states with state 1 at time t, then the vector $\mathbf{v}_0^I$ holds the node states of all nodes in the set $\mathbf{V}^I$ at time 0.

For example, consider two neighbouring nodes. At time $t = 0$, the first node is infected, so $v_t(1) = 1$ and the second node is susceptible $v_t(2) = 0$. The system state $\mathcal{S} = [v_t(1), v_t(2)] = [1, 0]$. From this state, after some time node 1 is either healed with rate β or node 2 is infected with rate α.

Other infection spread models change the original contact process slightly. When using the SI (Susceptible, Infected) model [12], rate β is 0 because infected nodes cannot heal. When using the SIR (Susceptible, Infected, Recovered/Removed) model [12] the state space becomes: $\mathcal{C} = \{0, 1, 2\}^n$. A node is in state 2 when it has recovered from the virus or is removed from the system after 'dying' from the infection. In either case, nodes in state 2 cannot be infected again.

2.2 Random Graph

A random graph can be defined as a system with n points or nodes. Each of the nodes can be connected to other nodes if there is a line or edge between them. An edge can be added (or is active) between two nodes or is removed (inactive) with a certain probability [3]. So a random graph consists of a set of nodes $\mathbf{V}$ and a set of (active) edges $\mathbf{E}$.

2.3 Markovian random graph

A Markovian random graph or random network has a system state defined as a random graph that can change its system state according to a Markov process [13]. The change between states is time dependent and determined by transition rates. In this chapter, a random network with a fixed number of nodes is considered. A random network with a changing number of nodes will be discussed in Chapter 3.

A random network with a fixed number of nodes (n is constant) has m edges. The number of edges is dependent on the number of nodes: $m = n \cdot (n - 1)$ or $m = n \cdot (n - 1) \cdot 0.5$ if the edges are non-directional. An m dimensional vector $\mathbf{e}_t$ holds the state of all edges where $e_t(y)$ is the state of edge y. Each of the edges can either be active/added to system (state 1) or inactive/removed from system (state 0). This leads to a state space of $\mathcal{C} = \{0, 1\}^m$ and a system state of $\mathcal{S} = \mathbf{e}_t$. In this random graph, the system state changes whenever an edge is added or removed from the system. A random edge can change status at a rate p (or alternatively an edge is added with rate p_+ and removed with rate p_-). An edge (status) can also be swapped between nodes at rate s. A transition matrix $\mathbf{Q}_n$ can be defined to show how the random graph changes states. This transition matrix can be divided in sub-matrices $\mathbf{Q}_{i,j}$, where i is the level of the current system and j the level of the system after a transition takes place. The level of the random graph is the number of active edges. This leads to the following transition matrix:

$$\mathbf{Q}_n = \begin{pmatrix} \mathbf{Q}_{0,0} & \mathbf{Q}_{0,1} & & & & & \\ \mathbf{Q}_{1,0} & \mathbf{Q}_{1,1} & \mathbf{Q}_{1,2} & & & & \\ & \mathbf{Q}_{2,1} & \mathbf{Q}_{2,2} & \mathbf{Q}_{2,3} & & & \\ & & \ddots & \ddots & \ddots & & \\ & & & \mathbf{Q}_{m-1,m-2} & \mathbf{Q}_{m-1,m-1} & \mathbf{Q}_{m-1,m} \\ & & & & \mathbf{Q}_{m,m-1} & \mathbf{Q}_{m,m} \end{pmatrix}$$

The edge states can also be stored in a $n \times n$ matrix where the matrix indices correspond to the indices of each edge $e_t(i, j)$. For example the edge between node 1 and node 2 would be $e_t(1, 2)$ and $e_t(1, 2) = e_t(2, 1)$ if the edges are non directional. The values on the diagonal are always zero since no edges exist from one node to itself. These values will represent the state of the nodes later on. For a random network with 2 nodes the system states in matrix form and the transition rates between the states are shown in Figure 2.1. In this figure an edge status changes with rate p and the status between the two edges can be swapped with rate s.

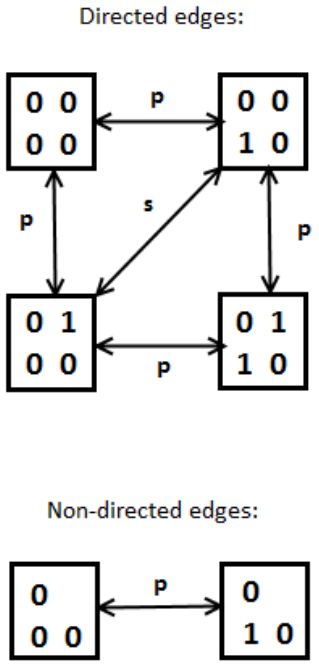

Figure 2.1: States in matrix form and transition rates for a random network with 2 nodes.

2.4 Discrete event simulation

A discrete event simulation is a simulation of a system or model that changes due to instantaneous events at certain times instead of continues changes over time [5]. Discrete event simulation can be activity, event or process oriented [9].

In an activity oriented simulation a very small time step is chosen and at each time step the system is evaluated to see if an event has occurred. Event oriented simulation time jumps from one event to the next event. Events may be scheduled where the next event is the first event to happen. Process oriented discrete event simulation groups related events into processes and simulation keeps track of the processes instead of each individual event.

Discrete event simulations are used in various fields like manufacturing, health care administration, military applications and logistics [2].

2.5 Relevant research

Since the introduction of the first contact process, it has been used for a lot of research on infection spread models. Some of the more recent research focusses on social contact networks that are often used as models for prediction of virus spread in real life situations. For example a study into public health interventions on a contact process modelled after Singapore [15]. Much research is focused on the social aspect of the contact process to model the process after real life social processes [11]. Other studies are more focussed on virus spread models or detection of a virus in a system. For example the latter can be useful research into better detection of virus spread in computer networks [7].

While some research is done on static contact processes (where the structure of the process does not change during simulation), other research allows for the model to change in response to the infection spread, like the birth and death of nodes [16] or edges to be swapped between nodes [14] [12]. Random networks are being used to create a random contact network but little research has been done on contact processes that continually change independently of the virus spread.

Almost all studies are using discrete event simulation to simulate their models. Depending on the model, different types of simulations are used like event based, agent based or time step simulations[6]. Parallel (or multi-threading) simulation is sometimes used in case of large models[15].

Chapter 3

Contact process on a random network

In this chapter a random network with a changing number of nodes is defined. This network is then used as a graph for a contact process leading to the random network contact process that is simulated in this research.

3.1 Random network with changing number of nodes

In Chapter 2 a Markovian random graph or random network has been introduced. Now a random network with a changing number of nodes is defined. Since the number of nodes n is now a variable from $n_{min} \geq 0$ to n_{max} which is possibly infinite, the number of edges m also becomes a variable since it depends on n. This also means that the matrix $\mathbf{Q}_n$ is different in size for different values of n. The system state and state space are still: $\mathcal{S}_t = \mathbf{v}_t, \mathbf{e}_t$ and $\mathcal{C} = \{0,1\}^m$. The transition matrix $\mathbf{Q}$ is defined as:

$$\mathbf{Q} = \begin{pmatrix} \mathbf{Q}_{n_{min}} & \mathbf{T}_{n_{min},n_{min+1}} & & & \\ \mathbf{T}_{n_{min+1},n_{min}} & \mathbf{Q}_{n_{min+1}} & \mathbf{T}_{n_{min+1},n_{min+2}} & & \\ & \ddots & \ddots & \ddots & \\ & & \mathbf{T}_{n_{max-1},n_{max-2}} & \mathbf{Q}_{n_{max-1}} & \mathbf{T}_{n_{max-1},n_{max}} \\ & & & \mathbf{T}_{n_{max},n_{max-1}} & \mathbf{Q}_{n_{max}} \end{pmatrix}$$

Here $\mathbf{Q}_n$ is the transition matrix for a random network with n nodes and $\mathbf{T}_{i,j}$ is the transition matrix from the random network with i nodes to a random network with j nodes.

For a system with $1 \leq n \leq 3$, $\mathbf{Q}$ has the following structure:

$$\mathbf{Q} = \begin{pmatrix} \mathbf{Q}_1 & \mathbf{T}_{1,2} & \\ \mathbf{T}_{2,1} & \mathbf{Q}_2 & \mathbf{T}_{2,3} \\ & \mathbf{T}_{3,2} & \mathbf{Q}_3 \end{pmatrix} = \begin{pmatrix} \mathbf{Q}_{1,0,0} & \mathbf{T}_{1,2} & & & & & \\ \mathbf{T}_{2,1} & \mathbf{Q}_{2,0,0} & \mathbf{Q}_{2,0,1} & & & & \\ & \mathbf{Q}_{2,1,0} & \mathbf{Q}_{2,1,1} & \mathbf{T}_{2,3} & & & \\ & & \mathbf{T}_{3,2} & \mathbf{Q}_{3,0,0} & \mathbf{Q}_{3,0,1} & & \\ & & & \mathbf{Q}_{3,1,0} & \mathbf{Q}_{3,1,1} & \mathbf{Q}_{3,1,2} & \\ & & & & \mathbf{Q}_{3,2,1} & \mathbf{Q}_{3,2,2} & \mathbf{Q}_{3,2,3} \\ & & & & & \mathbf{Q}_{3,3,2} & \mathbf{Q}_{3,3,3} \end{pmatrix}$$

Here the sub-matrices $\mathbf{Q}_{n,i,j}$ with indices n, i, j are the number of nodes, the level (number of active edges) of the current system and the level of the system if a transition in the sub-matrix takes place.

An example of a random network is a network where nodes are added with an (exponential) rate λ and removed with rate μ. The edges in this network are non-directional. Edges can change state between active (state 1) and inactive (state 0) with rate p. Finally an edge can swap its state with another edge at rate s. Each of these rates can be a constant rate or a function dependent on variables like the number of nodes or number of (active) edges. For a network with $1 \leq n \leq 3$ the states of the system are defined in the table below:

state	$\mathbf{e}_t$	n	m	l
1	[]	1	0	0
2	[0]	2	1	0
3	[1]	2	1	1
4	[0 0 0]	3	3	0
5	[1 0 0]	3	3	1
6	[0 1 0]	3	3	1
7	[0 0 1]	3	3	1
8	[1 1 0]	3	3	2
9	[1 0 1]	3	3	2
10	[0 1 1]	3	3	2
11	[1 1 1]	3	3	3

Table 3.1: States of random network with $1 \leq n \leq 3$.

The transition matrix for this random network is:

$$\mathbf{Q} = \begin{pmatrix}
0 & \lambda & & & & & & & & \\
\mu & 0 & p & \lambda & 0 & & & & & \\
\mu & p & 0 & 0 & \lambda & & & & & \\
 & \mu & 0 & 0 & p & p & p & & & \\
 & \frac{2}{3}\mu & \frac{1}{3}\mu & p & 0 & s & s & p & p & 0 \\
 & \frac{2}{3}\mu & \frac{1}{3}\mu & p & s & 0 & s & p & 0 & p \\
 & \frac{2}{3}\mu & \frac{1}{3}\mu & p & s & s & 0 & 0 & p & p \\
 & \frac{1}{3}\mu & \frac{2}{3}\mu & & p & p & 0 & 0 & s & s & p \\
 & \frac{1}{3}\mu & \frac{2}{3}\mu & & p & 0 & p & s & 0 & s & p \\
 & \frac{1}{3}\mu & \frac{2}{3}\mu & & 0 & p & p & s & s & 0 & p \\
 & 0 & \mu & & & & & p & p & p & 0
\end{pmatrix}$$

The colors of the rates in the transition matrix correspond with the colors from the sub-matrices $\mathbf{Q}_{n,i,j}$ and $\mathbf{T}_{i,j}$ determined earlier in this section.

3.2 Virus infection and spread models on the random network

The random network can be used as a graph for a contact process. The random network replaces the lattice of nodes that defined the contact process in Chapter 2. Now, instead of infected nodes always being able to infect their neighbors, the virus can spread from an infected node to any susceptible node in the network so long as there is an active edge between them. When edges are directional the edge from the infected node to the susceptible node has to be active. If the edges are non-directional the edge between nodes simply has to be active. From now on edges are considered non-directional unless mentioned.

Using the SIS infection spread model each node can be susceptible (state 0) or infected (state 1). Infected nodes can spread the infection with rate α and be healed with rate β. The state space for the random network contact process becomes: $\mathcal{C} = \{0,1\}^n \times \{0,1\}^m$. The system state is defined as: $\mathcal{S} = \mathbf{v}_t, \mathbf{e}_t$. In Figure 3.1 the states and state transitions of random network with a fixed number of nodes of 2 is shown. The state of nodes is added on the diagonal of the state matrices in the figure to account for the spread of a virus or infection.

For different infection models the random network changes slightly. The rate of healing β is zero when an SI model is used. The state space changes to $\mathcal{C} = \{0,1,2\}^n \times \{0,1\}^m$ for a SIR virus spread model where state 2 is the recovered/removed state of the node.

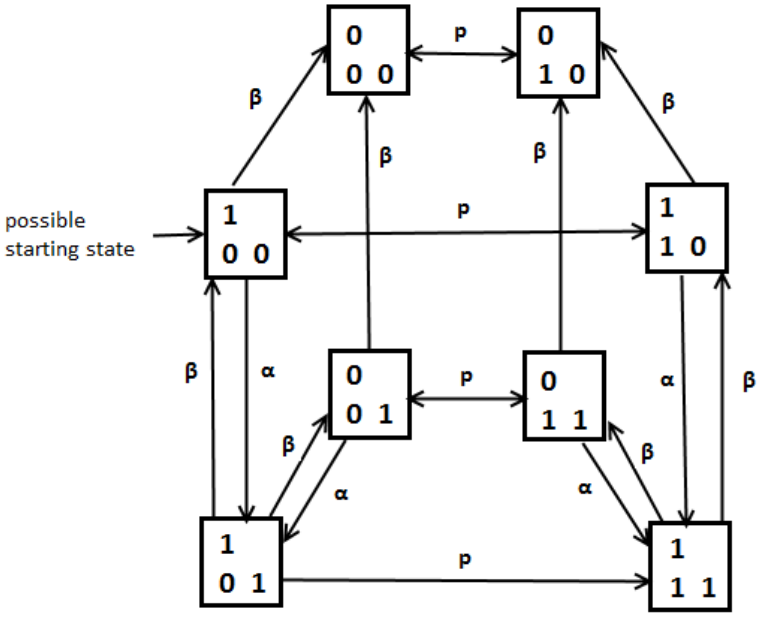

Figure 3.1: States in matrix form and transition rates for a random network contact process with 2 nodes.

3.3 Random network contact process rates

The random network contact process as defined above uses exponential rates to make transitions between states for both the random network itself and the infection spread. These rates can be constant or a function of one or more variables, like the number of nodes and (active) edges. In the simulation of the network, various rates can be used and different options for the calculations of rates are available as well. The rates that can be used are λ to add a node the system and μ to remove a node from the system. Rate p to change the state of an edge, or alternatively p^+ and p^- to add an edge (change inactive edge state to active) or remove an edge (change active edge state to inactive). Rate s to swap the state of an edge with another edge or s^+ to swap the state of an active edge with the state of an inactive edge. There are also the rates α to infect a node and β to heal a node. All rates have a base rate and dependent on the chosen simulation options the actual rates are calculated. For all simulation options regarding the rates of the random network, see appendix A.

Chapter 4

Random network simulation: implementation

In the previous chapter a random network contact process was defined where nodes and edges can be added and removed from the system while an infection or virus spreads through the network. A simulation program is developed to simulate the random network contact process, so results regarding the spread of the infection can be generated and analyzed. In this chapter the implementation and programming of the network simulation in C++ is discussed.

4.1 Simulation method

To simulate the random network, a discrete event simulation is chosen. As mentioned in Chapter 2, a discrete event simulation is a good solution for a random network, since it is a system that changes from one state to another after a certain time interval. The time interval between state changes is determined by the exponential rates that define how the network changes state as discussed in previous chapters, for example rate λ to add a node to the system or rate α to spread the infection from one node to another. For this reason a discrete event simulation method is chosen where the next event determines the time step.

Events are not scheduled, because the random network (and thus the rates that determine the next event) can change so much that rescheduling might be needed after each event. Instead the rates are used to calculate the time of the next event and which event happens at this time on a global or high level scope. This means that the choice of the time and event do not dictate which nodes and/or edges are chosen to implement the event, only the type of event is determined. Since each event has an exponential rate of occurring the rates of all possible events are added to determine the time of the next event. The type of event is chosen by a random number between 0 and the sum of the exponential rates. For example, if the next event can only be the spread of the infection with rate $\alpha = 1$ or the healing of a node with

rate $\beta = 1$, then the time of the next event is determined by an exponential rate of 2. The type of event is then determined by a random number between 0 and 2; if the random number is smaller than 1 the next event is the spread of an infection, else the next event is the healing of a node (the random number is between 1 and 2).

After the time and type of the next event is calculated the event is implemented on a local or low-level scope where some of the individual properties of the nodes and/or edges are used. For example, if the next event is to spread the infection, then first an infected node is selected at random. Second, a connected node to the chosen infected node is selected at random and if this node is susceptible the state of this node will change to infected. In this research the choice of these nodes and/or edges for any event is mostly random where only the status of the nodes and edges is taken into account.

This simulation method is chosen with keeping in mind that the random network can get increasingly more complex in terms of rate calculations and individual node and edge properties. While a purely high level approach might be a better option for very simple random networks (although nodes and edges still have to be chosen randomly) and a low level event scheduling method might be better for (very) complex networks, the choice to determine the time and type of event on a global scale and implement the event on a local scale was made so the simulation can be used for simple to increasingly more complex networks.

4.2 C++ implementation

The simulation is programmed in C++ to make use of an efficient and flexible language that is object oriented. Due to a lack of experience with programming in C++ the website www.learncpp.com was used to learn the programming language [1]. Visual Studio Express is used as the Integrated Development Environment (IDE). The IDE is used to program, compile and debug the simulation program. To better understand the implementation of the simulation a few concepts that are used when programming in C++ are described.

Files and headers:
When a program gets larger using multiple files makes it easier to develop a program. However to link these files together functions used from one file in another have to be declared in those files. This is called a forward declaration. Headers are files that are used for forward declarations so only the header file has to be linked to in a file that wants to use functions from other files. Program files have a .cpp extension and header files a .h extension. The simulation header files are also used to define data structures and classes that can then be included in every file that needs to use them.

Libraries:

Libraries are predefined files with code that can be used by including the appropriate headers in the files where they are needed. The IDE provides a large standard library that includes code for random number generators, structures like vectors, options for using input and output streams to data files and much more. Using the standard library is essential in programming a simulation (or anything else).

Data structures and classes:
Structures or structs in C++ are a collection or group of variables that define a single unit or object. For example a rate in the simulation has a variable that defines which type of rate it is, a variable that holds the base rate value and a variable that holds the actual rate value for the current system state. Classes are like structs but have the additional option to define functions inside the class to operate on the variables that are part of the class.

4.2.1 Overall structure

The general function of the program is to run (multiple) simulations where for each simulation run the program must choose the next event based on the transition rates, execute this event by choosing which nodes and/or edges are affected by this event and update the system state after the event accordingly. The program uses the following files to achieve this:

main.cpp:
This file is where the simulation starts and finishes. It iterates the simulation runs until they are all finished as specified by the options. Main.cpp will also create the data structures for the node and edge information and reserve memory to hold this information. If there is no input data file in the directory of the executable this file will call to create a new input file and closes the program.

simulation.cpp:
A simulation is initialized and started in this file. It will then iterate through choosing events until the stopping criteria are met. After choosing an event it will call the corresponding function in event.cpp to execute the event and update the system. Also the functions to update the rates using the new system state data are called from here.

event.cpp
In this file all the functions for every event can be found. When an event is determined the corresponding event function chooses which nodes and/or edges are going to carry out the event if available. Then the event is executed and if successful the node and edge data will be updated.

The file rng.cpp uses the Mersenne Twister 19937 generator to generate a random number whenever needed. This generator is available from the standard library. A separate file

rates.cpp is used to update the rates that will determine the next event. Figure 4.1 shows how these fundamental files relate to each other.

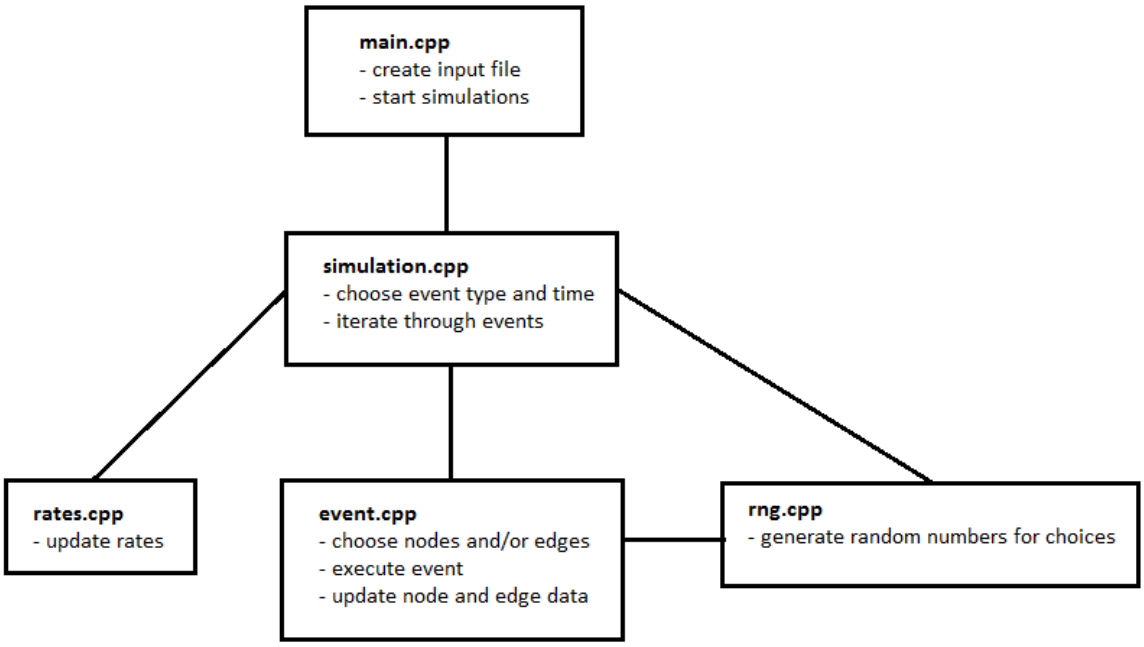

Figure 4.1: Links between important program files of simulation.

To simulate a random network, the program needs to store the state of the system (along with other data). Nodes are defined as a class in the node.h header file that holds the node information for an individual node and some general system data like the number of nodes, number of active edges and the number of infected nodes in the system. Node data is stored in a vector defined in the main.cpp file. Edge data is defined in a similar way in the datastructs.h header file. This file also defines other data structures that are used to store rate and simulation data. The options are also defined and stored in a class structure in the options.h header file to allow every file access to the options.

A complete list with all files and headers with short descriptions of their functions can be found in appendix C along with the actual code. Next a closer look into the implementation of some features of the simulation is given.

4.2.2 Input and output

For the input and output of options and data the use of data files is chosen. They are relatively easy to program and the output data can be imported into other programs like Matlab to analyze the results. Output is generated by function in the output.cpp file.

Since the simulation is capable of using a lot of different options, changing them constantly inside the IDE would be cumbersome and would lead to creating different executables for each variation of the simulation for use outside the IDE. Using a data text file based input method this is not a problem anymore. The program will create an input.dat file when it is run in a directory where no input.dat file is available. If input.dat already exists it uses this file to override the default options of the simulation to they ones specified in the file if these are valid. The input.dat file is written and read by functions from the input.cpp file. The generated input.dat file with the all the changeable options available and comments with all other valid options can be found in Appendix A.

4.2.3 Event choice

The time and type of the next event is determined by the rates in the simulation.cpp file. Since all rates are exponentially distributed in the simulation, the sum of the rates is determined. Using this sum and the random number generated, the time of the next event is calculated using the exponential distribution. After this a new random number is determined between 0 and the sum of the rates to determine the type of events. Using this random number a weighted choice from the values of the rates is made. After the chosen event is executed and the system updated the rates are updated using the new system state from function in the rates.cpp file. The updated rates can then be used to determine the next event.

4.2.4 Event implementation and system update

After an event is chosen it is executed by one of the functions in event.cpp. Each event has its own function. First nodes and/or edges have to be chosen for most events. For some events first eligible nodes or edges have to be preselected using a shortlist vector. Then they are chosen randomly using the random number generator. After the required nodes and/or edges are determined the system can be updated. For example this may be the removal of a node from the system that would lead to removing all edges connected to this node as well. One of the node variables that is used is the number of connected edges to each node. They will have to be updated too. Also global variables like the number of nodes and the number of active edges must be updated. After the system is updated accordingly the next event can be determined. To get a more in depth look into the implementation of each event the full C++ code is available in Appendix C.

Chapter 5

Random network simulation: results

After programming and testing the simulation, the program is used to generate results for various random networks. This chapter describes the chosen random networks and looks at the results from the simulations.

5.1 Simulation options

To take a look at the virus spread on a random network, three random network variations are chosen. The first is a fixed random network, the second a random network with linear rates and the last a random network with balanced rates. For each network the virus spread is analyzed. For all three networks and for each variation of those networks, the simulation starts with one node infected and the simulation will run for a number of events. This simulation run is then repeated 1000 times. For each simulation the results from the 'results.dat' file are used. The results from the simulations are analyzed using Matlab. All three network options will have three values or settings that are varied to see the effect of these variations on the survival chance of the virus after the specified number of events. The maximum fraction of infected nodes when the virus goes extinct and the average fraction of infected nodes when the virus survives are analyzed as well.

The fixed random network is a random network where the number of nodes and the number of edges is fixed for the duration of the simulation. This network is used to look at the virus spread on a network that does not change itself. Table 5.1 shows some of the defining options of this network (all options can be found in Appendix A). Here the maximum number of events is chosen sufficiently large for the network infection to reach steady state. The fraction of active edges is the desired fraction of active edges wanted in the network. This fraction can also be used to determine the chance for each edge to be active or inactive at the start of the

simulation. For example, a fraction of active edges of 1 means all edges are active at the start, and a fraction of active edges of 0.5 means each edge has a 0.5 chance to be active at the start.

number of nodes n	1000 (or variable)
fraction of active edges at start f_e	1 (or variable)
base rate for infecting/healing node α_b	1
infection/healing rate ratio r_α	1.5 (or variable)
infection rate α	linear function at $\alpha_b \cdot \frac{r_\alpha}{r_\alpha + 1} \cdot n_I$
healing rate β	linear function at $\alpha_b \cdot \frac{1}{r_\alpha + 1} \cdot n_I$
maximum number of events	10000

Table 5.1: Options for the fixed random network.

The survival chance, maximum infection fraction when extinct and average infection fraction after survival are looked at for variations in the ratio between infection and healing rate α and β, the fraction of edges that is active and the number of nodes. In Appendix A the used values for these variations can be found.

The random network with linear rates is a standard random network where nodes are added and removed, and edges can change and swap there states. All rates are a linear function of either the number of nodes or the number of edges. Some of the networks options can be found in Table 5.2. Again, the maximum number of events is chosen sufficiently large for the network infection to reach steady state.

The ratio of infection/healing will be varied for this network, once in case nodes can be added and removed from the network and once when this is not possible. Also the base rate of adding and removing nodes will be varied to see the effect on the infection spread. The full options for this network and the values of used variables can be found in Appendix A.

The random network with balanced rates is a network where the sum of the rates for adding (rate λ) and removing (rate μ) nodes is constant while the ratio between the rates is linear balanced to the chosen balanced number of nodes. This means a higher ratio for adding nodes if $n < n_{balanced}$ and lower for $n > n_{balanced}$. For example, take the constant sum of rates λ and μ at 1. Then $\lambda = \mu = 0.5$ if $n = n_{balanced}$, $\lambda > \mu$ if $n < n_{balanced}$, and $\lambda < \mu$ if $n > n_{balanced}$. The same is valid for adding and removing edges except that they are balanced around the chosen fraction of active edges. The combined rate of infecting and healing nodes is also balanced to a fraction of $n_{balanced}$ called the infection factor. This means the infection and healing rates are still linear dependent on the number of infected nodes but the combined rate is equal to the base rate if the number of infected nodes is equal to the number of balanced

starting number of nodes n_{start}	100
min number of nodes n_{min}	0
max number of nodes n_{max}	200
number of total edges e_{max}	$n \cdot (n-1) \cdot 0.5$
fraction of active edges f_e	0.5
base rate for adding/removing nodes λ_b	1 (or variable)
base rate for changing edge status p_b	0.01
base rate for swapping edge status s_b	0.01
base rate for infecting/healing node α_b	10
infection/healing rate ratio r_α	4 (or variable)
add nodes rate λ	constant at $0.5 \cdot \lambda_b \cdot n_{start}$
remove nodes rate μ	linear function at $0.5 \cdot \lambda_b \cdot n$
change edge status rate p	linear function at $p_b \cdot e_{max}$
swap edge status rate s	linear function at $s_b \cdot e_{max}$
infection rate α	linear function at $\alpha_b \cdot \frac{r_\alpha}{r_\alpha + 1} \cdot n_I$
healing rate β	linear function at $\alpha_b \cdot \frac{1}{r_\alpha + 1} \cdot n_I$
maximum number of events	25000

Table 5.2: Options for the random network with linear rates.

nodes times the infection factor. In Table 5.3 some of the simulation options can be found.

The ratio between infection and healing rate will again be one of the variables. The infection/healing factor and the fraction of active edges will also be varied. Again, all the options for this network can be found in appendix A, as well as the used values for the variables.

starting number of nodes n_{start}	500
balanced number of nodes $n_{balanced}$	500
min number of nodes n_{min}	0
max number of nodes n_{max}	1000
number of total edges e_{max}	$n \cdot (n-1) \cdot 0.5$
fraction of active edges f_e	0.5 (or variable)
balanced number of edges $e_{balanced}$	$e_{max} \cdot f_e$
base rate for adding/removing nodes λ_b	1
base rate for changing edge status p_b	1
base rate for infecting/healing node α_b	1
infection/healing rate ratio r_α	3 (or variable)
infection/healing factor f_α	0.01 (or variable)
add node rate λ	$\begin{cases} \lambda_b \cdot 0.5 & \to n = n_{balanced} \\ \lambda_b \cdot (1 - 0.5 \cdot \frac{n - n_{min}}{n_{balanced} - n_{min}}) & \to n < n_{balanced} \\ \lambda_b \cdot (1 - 0.5 \cdot \frac{n - 2 \cdot n_{balanced} + n_{max}}{n_{max} - n_{balanced}}) & \to n > n_{balanced} \end{cases}$
remove node rate μ	$\begin{cases} \lambda_b \cdot 0.5 & \to n = n_{balanced} \\ \lambda_b \cdot 0.5 \cdot \frac{n - n_{min}}{n_{balanced} - n_{min}} & \to n < n_{balanced} \\ \lambda_b \cdot 0.5 \cdot \frac{n - 2 \cdot n_{balanced} + n_{max}}{n_{max} - n_{balanced}} & \to n > n_{balanced} \end{cases}$
add edge rate p^+	$\begin{cases} p_b \cdot (1 - 0.5 \cdot \frac{e}{e_{balanced}}) & \to e <= e_{balanced} \\ p_b \cdot (1 - 0.5 \cdot \frac{e - 2 \cdot e_{balanced} + e_{max}}{e_{max} - e_{balanced}}) & \to e > e_{balanced} \end{cases}$
remove edge rate p^-	$\begin{cases} p_b \cdot 0.5 \cdot \frac{e}{e_{balanced}} & \to e <= e_{balanced} \\ p_b \cdot 0.5 \cdot \frac{e - 2 \cdot e_{balanced} + e_{max}}{e_{max} - e_{balanced}} & \to e > e_{balanced} \end{cases}$
infection rate α	linear function at $\alpha_b \cdot \frac{r_\alpha}{r_\alpha + 1} \cdot \frac{n_I}{f_\alpha \cdot n_{balanced}}$
healing rate β	linear function at $\alpha_b \cdot \frac{1}{r_\alpha + 1} \cdot \frac{n_I}{f_\alpha \cdot n_{balanced}}$
maximum number of events	25000

Table 5.3: Options for the random network with balanced rates.

5.2 Result analysis

For the three random network simulations, the fraction of times the virus survives, the average fraction of infection at the end of the simulation if the virus survives and the maximum fraction of infection when the virus goes extinct are analyzed.

The fraction of times the virus survives indicates how likely the virus can survive after a long period of infection. It can also show in what range for values of certain variables the virus has any chance of survival. The average fraction of infection at the end of the simulation can show whether the whole or almost the entire network is infected or that some balance between infected and susceptible nodes is reached. The maximum fraction of infection when the virus goes extinct may show a minimum infection fraction that has to be reached by the infection to almost guaranty long term survival or even infinite survival.

All the simulation results in the form of graphs can be found in Appendix B. Below the results are discussed for each of the three random networks.

5.2.1 Fixed random network

For the fixed random network the infection/healing rate ratio, the active edge fraction and the number of nodes were varied.

Infection/healing rate ratio:
It looks like the infection has any chance of survival if the ratio between the infection and healing rate is 1 or greater. In Figure 5.1 it seems that for a ratio greater than 1 the survival chance goes to 1.

The average infection fraction after survival is already almost 1 at a rate of approximately 1.25 from being 0 for rates smaller than 1. The maximum infection extinction fraction peaks at a ratio of 1 at almost 0.12 and decreasing for higher ratios, meaning the virus is very likely to survive if it can infect 10% of the system or even much less for higher ratios.

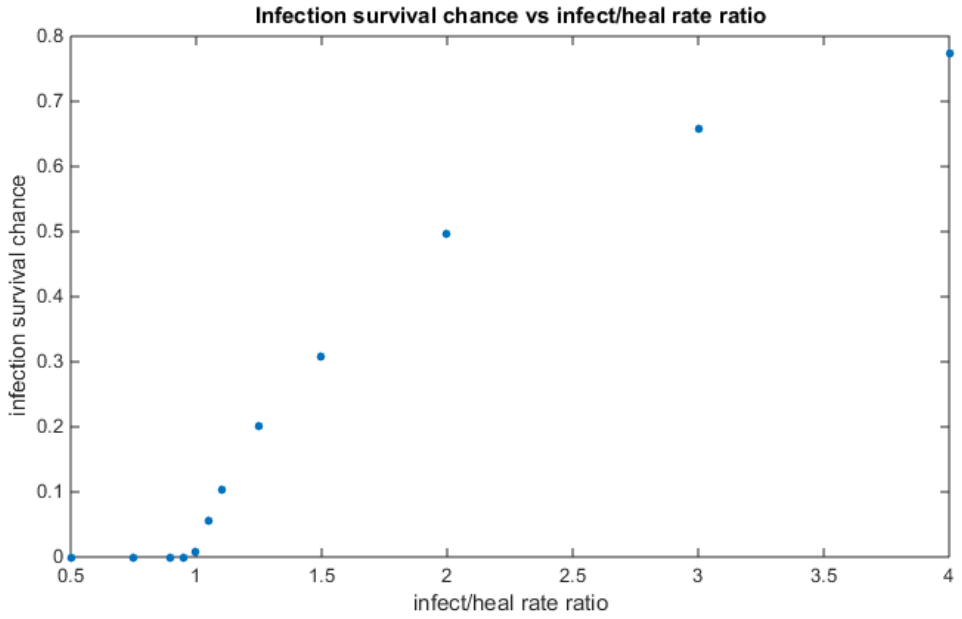

Figure 5.1: Infection survival chance vs infect/heal rate ratio for fixed random network.

Active edge fraction:
For active edge fractions between 0.1 and 0.9 there seems to be little to no change in survival chance and the maximum infection extinction fraction. The average infection survival fraction decreases a little for lower fractions but is still well above 0.98 at a fraction of 0.1. This means that even if 10% of the edges is active, the nodes have no trouble infecting each other with a very minimal difference between 10% and 100% active edges.

Number of nodes:
The number of nodes also has little effect on the survival chance where only the chance at 10 nodes seems a bit lower than the rest. The average infection survival fraction goes down significantly for a number of nodes lower than 100 but is still high at 0.85 for a 10 node system. The maximum infection extinction fraction however is 1 at 10 nodes. This means that even if the whole system is infected the chance of the virus going extinct will always be high in the long term. The fraction is already down below 0.2 at 100 nodes. The results show that there is virtually no difference in infection spread for a number of nodes at least higher than 500 and a very minimal difference between a number of nodes of 100 and higher.

5.2.2 Linear rates random network

For the linear rates random network the infection/healing rate ratio, the infection/healing rate ratio when there are nodes added or removed and the add/remove node bas rate fraction were varied.

Infection/healing rate ratio:
The virus has a survival chance from a ratio of 2.5 or higher. From this ratio of 2.5 the survival infection fraction seems to linearly increase, but is still well below 40% at a ratio of 5.5. The maximum infection extinction fraction is very high. Networks that are infected up to 90% still have a chance to go extinct. Given the lower survival infection fraction it is very unlikely a infection will not eventually die.

Infection/healing rate ratio without adding/removing nodes:
The difference between allowing adding and removing nodes or not is great. Now that it is not allowed, the infection has a survival chance from ratio of 1 instead of 2.5, just as the fixed random network. Also the survival infection fraction and maximum infection fraction when going extinct are very similar to fixed random results. The influence from adding and removing nodes seem great on virus spread while the influence from changing or swapping the status of edges is very small given the similarities with the fixed random network results.

Add/remove nodes base rate fraction:
The survival chance remains almost unchanged for the base rate from 0.01 to 0.5. The survival chance goes down fast after that to 0 at a base rate of 1.5 as shown in figure 5.2. The average infection fraction after survival decreases linear from 1 at a rate of 0.01 to 0 at a rate of 1.5. The maximum infection fraction when extinct is very low between a base rate of 0 and 0.5 while very high from 1 to 1.5, the rate after which no virus is able to survive anymore. Further research into area between 0.5 and 1 is recommended.

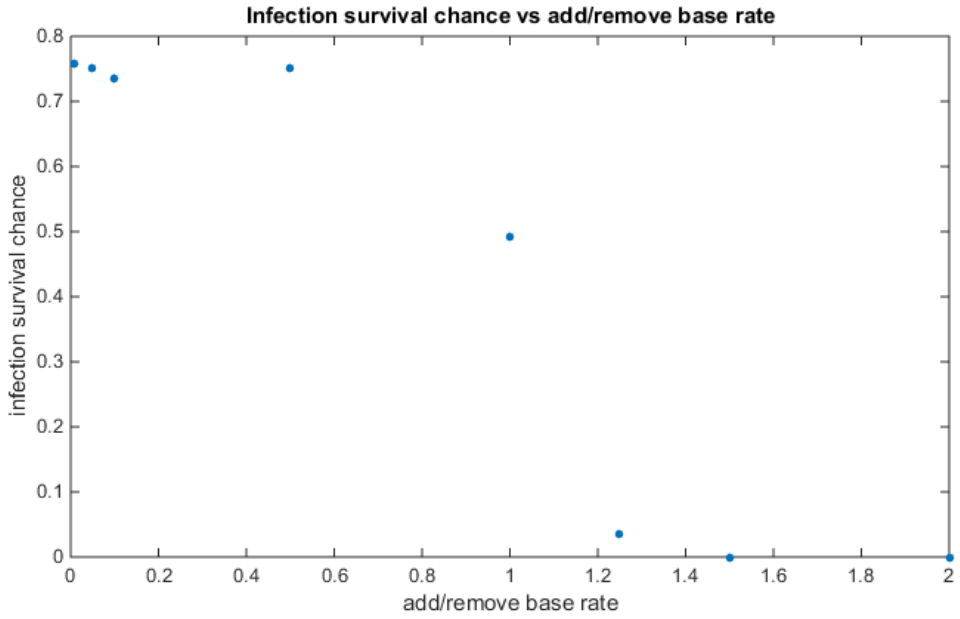

Figure 5.2: Infection survival chance vs add/remove base rate for linear rates random network.

5.2.3 Balanced rates random network

For the balanced rates random network the infection/healing rate ratio, the infection/healing rate fraction and the active edge fraction were varied.

Infection/healing rate ratio:
The survival chance increases almost linearly from 0 at ratio of 1 to 0.65 at a ratio of 3. The survival infection fraction also increases from 0 at a ratio of 1, but may find a balance of 65% survival change for higher ratios. There is a very low maximum extinction infection fraction for the whole range of investigated ratios.

Infection/healing rate factor:
The survival chance is going down for high virus rate factors. The survival infection fraction also decreases for higher factors while the maximum extinct infection fraction increases. But this fraction stays below 4%.

Active edge fraction:
Figure 5.3 shows a very low survival chance for active edge fractions below 0.005. For fractions above 0.01 the survival chance stays level at 45%. There is an almost linear increase of the survival infection fraction. The maximum extinction fraction is higher around 10% for very

low active edge fractions when there is still a survival chance, instead of lower than 3% for higher active edge fractions.

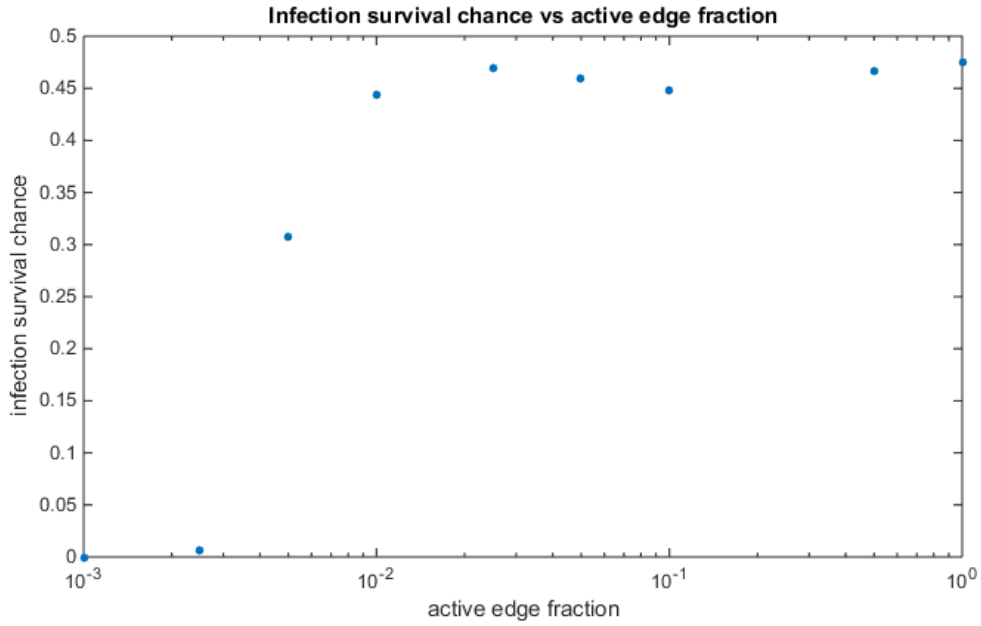

Figure 5.3: Infection survival chance vs active edge fraction for balanced rates random network.

Chapter 6

Recommendations

This research has only started scratching the surface of random network contact process simulations and analysis. Therefore recommendations can be found in this chapter on how and where to proceed further with research in terms of the random network contact processes themselves and the simulation program.

6.1 Further research into random network contact processes

The results into some variations of simple random networks show influences from primarily adding and removing nodes from the system, as well as a system that has so few active edges that it becomes difficult for infected nodes to infect susceptible nodes. The results from these simulations are just the beginning of exploring the influence of a random network on infection spread models. Only a few random networks have been analyzed for only a few variables. Also the results are just a first look into the infection spread on random networks. More in-depth analysis combined with more simulation results is recommended to get a better picture and more statistically proven results. Especially the effects of adding and removing nodes from a system seem to have much influence.

Further research into other variables for random networks is recommended as well. The effects of different ways to calculate rates can be investigated a lot more. Other infection spread models like SI and SIR may also be considered to see if they have significantly different effects.

Another option is to look into random networks that react to the virus spreading through the system. For example, nodes that are infected are more or less likely to be removed from the system than susceptible nodes. This will lead to more complex systems, so more complex random networks with individual node and edge properties are a logical next step to research into further. More complex systems may give different results as is already shown in studies where social aspects are modelled into existing static network contact processes that give

significant different results as to how a virus or infection spreads through the system.

6.2 Simulation

The current simulation has a lot of options for different random network contact processes to be analyzed, at least more than what is used in the result analysis this research. Thus the current simulation program can be used to further explore and analyze the spread of infections on relatively simple random networks as discussed above. However to go further into more complex random networks the simulation should be expanded. This simulation can be used as a basis for these more complex systems, but a few major alterations are recommended if the program is to be expanded to simulate more random networks with more options to choose from.

The most important change would be to switch from a data file input program to the use of a graphical user interface (GUI) from which all options can be chosen. A GUI could also lead to more control during simulations such as pausing and stopping the simulation and looking at partial results (say the results from the first 100 simulations from a 1000 simulation run).

A more in depth look into the used data structures and memory use may lead to a faster and more efficient simulation. This can be of great importance when large and more complex random networks are to be simulated. The current limit on the size of the network is around 20000 nodes due to memory. This will no doubt decrease when nodes and especially edge data increases.

Multi-threading could be added to the simulation program as well. This can significantly reduce simulation times if multiple simulations can be run simultaneously. Due to the nature of the chosen discrete event method using multi-threading within one simulation run seem not feasible. Calculating multiple events at the time will lead to much synchronization and system update problems since the occurrence and choice of events can depend heavily on the outcome of previous events.

Finally more output options could be added to the simulation if other aspects of the random network are of interest as well as other aspects of the infection spread.

Chapter 7

Conclusion

The theory for contact networks and Markovian random graphs or random networks was combined to look at the effects of infection spread on a system with a changing structure. A simulation was programmed that is able to simulate many runs. The program can simulate different random networks that primarily differ in size. It also has a lot of options for different calculations for transition rates. Using the simulation different random networks were simulated and the results analyzed.

The results from the simulations showed that besides the ratio between infection and healing rates, the infection spread on a random network is influenced by adding and removing nodes from the system. Not only an infected node can be removed from the system, giving the network effectively extra healing, also all edges connected to this node are removed from the system. New nodes added to the system need time to get active edges with other nodes. These effects of adding and removing nodes showed that the infection needs a relative higher spreading ratio compared to the healing ratio.

It is recommended to do more research into the effect of adding and removing nodes on a random network in relation to virus spread. Also more study into other effects of random networks and more complex networks is advised.

Bibliography

[1] C++ tutorial. `http://www.learncpp.com`.

[2] Theodore T. Allen. *Introduction to Discrete Event Simulation and Agent-based Modeling.* Springer, 2011.

[3] E.N. Gilbert. Random graphs. *The Annals of Mathematical Statistics*, 30(4):1141–1144, 1959.

[4] T.E. Harris. Contact interactions on a lattice. *The Annals of Probability*, 2(6):969–988, 1974.

[5] D.P. Kroese, Thomas Taimre, and Zdravko Botev. *Handbook of Monte Carlo methods.* Wiley, 2011.

[6] Chia-Tung Kuo, Da-Wei Wang, and Tsan sheng Hsu. Simple and efficient algorithms to get a finer resolution in a stochastic discrete time agent-based simulation. *Simulation and Modeling Methodologies, Technologies and Applications*, 256, 2012.

[7] Jinho Lee, John J. Hasenbein, and David P. Morton. Optimization of stochastic virus detection in contact networks. *Elsevier*, 2014.

[8] Thomas M. Liggett. *Interacting Particle Systems.* Springer, 1985.

[9] Norm Matloff. Introduction to discrete-event simulation and the simpy language, 2008.

[10] Brendan Patch. Contact processes, 2014.

[11] Ana Perisic and Chris T Bauch. A simulation analysis to characterize the dynamics of vaccinating behaviour on contact networks. *BMC Infectious diseases*, 9(77), 2009.

[12] Sven Van Segbroeck, Francisco C. Santos, and Jorge M. Pacheco. Adaptive contact networks change effective disease infectiousness and dynamics. *PLoS Computational Biology*, 6(8), 2010.

[13] Thomas Taimre, Daniel Gibbons, and Brendan Patch. Markovian random graph processes. Manuscript.

[14] Erik Volz and Lauren Ancel Meyers. Epidemic thresholds in dynamic contact networks. *J. R. Soc. Interface*, (6):233–241, 2009.

[15] Tianyou Zhang, Xuju Fu, Michael Lees, CHee Keong Kwoh, Gary Kee Khoon Lee, and Rick Siow Mong Goh. A contact-network-based simulation model for evaluating interventions under what-if scenarios in epidemic. *Proceedings of the 2012 Winter Simulation Conference*, 2012.

[16] Guanghu Zhu, Guanrong Chen, Xin jian Xu, and Xinchu Fu. Epidemic spreading on contact networks with adaptive weights. *Journal of Theoretical biology*, (317):133–139, 2013.

Appendices

Appendix A

Simulation options

A.1 random network variable values

tables A.1, A.2 and A.3 show the used values for the chosen variables for each random network that is simulated:

Infect/heal rate ratio	Active edge fraction	Number of nodes
0.5	0.1	10
0.75	0.2	50
0.9	0.3	100
0.95	0.4	250
1	0.5	500
1.05	0.6	750
1.1	0.7	1000
1.25	0.8	2500
1.5	0.9	5000
2		7500
3		10000
4		

Table A.1: values for the chosen variables for the fixed random network simulation

Infect/heal rate ratio	Infect/heal rate ratio without add/remove nodes	add/remove nodes base rate fraction
1	1	0.01
1.5	1.5	0.05
2	2	0.1
2.5	2.5	0.5
3	3	1
3.5	3.5	1.25
4	4	1.5
4.5		2
5		
5.5		

Table A.2: values for the chosen variables for the linear rates random network simulation

Infect/heal rate ratio	Infect/heal rate fraction	active edge fraction
1	0.005	0.001
1.25	0.01	0.0025
1.5	0.025	0.005
1.75	0.05	0.01
2	0.075	0.025
2.25	0.2	0.05
2.5	0.5	0.1
2.75	1	0.5
3		1

Table A.3: values for the chosen variables for the balanced rates random network simulation

A.2 Options description

All the possible input options with a description of these options can be found in the standard input file generated by the simulation when no input file is in the directory of the program:

```
;Input file
;'=text' are option categories. all options in this category are directly below
;options consist of name and value separated with a space. a blank line or &
    means the end of options in this category
;comments start with ';'

;double values have an precision of 3, so higher precision on numbers with more
     than 3 values after the dot will be lost

;general uptions:
=general
numberOfSimulations 1 ;integer number: sets the number of simulations
nBalanced 500    ;integer number: sets the balanced number of nodes, used by
    various types of rate calculations
nMax 1000    ;integer number: sets the maximum number of nodes, both used to
    reserve memory and used by variouse types of rate calculations, cannot be
    lower than 'nBalanced'
nMin 0       ;integer number: sets the minimum number of nodes, cannot be higher
     than 'nBalanced'
edgeFraction 0.5  ;double number: sets the fraction of balanced edges that are
    active, used by various rate functions and the balanced start for active
    edges
fixedSeed 0    ;integer number: can be used so every simulation will have the
    same results(provided the options stay the same as well)
&

;start of simulation options:
=startOfSimulation
n 10       ;integer number: sets the number of nodes at the start of the
    simulation, must be between nMax and nMin
activeEdges balanced  ;string (text): 'none' means a system with no active
    edges, 'full' means a system with all edges active and 'balanced' means a
    system where each edge has a chance to be active according to the
    edgeFraction from the general options
&

;stopping criteria options:
=stoppingCriteria
time 10      ;double number: this is the time at which the simulation will stop
events 10000     ;integer number: this is the maximum number of events until
    simulation will stop
virusExtinction true  ;boolean: when true the simulation will terminate early
    when no nodes are infected anymore
timeOn true    ;boolean: when true the simulation will terminate at the time
    limit if the maximum number of events is not reached
```

```
&

;file output options
=fileOutput
simulationData true ;boolean: when true this creates a data file with general
    information on each simulation run
eventlog true   ;boolean: when true this creates a data file with an eventlog
    of all events
nodeInfo false    ;boolean: when true this creates a data file with all node
    information after each event
edgeInfo false    ;boolean: when true this creates a data file with the edge
    matrix after each event
ratesInfo false   ;boolean: when true this creates a data file with all the
    rates used in the simulation after each event
virusSpread true  ;boolean: when true this creates a data file with the number
    of nodes, number of infected nodes and the fraction of infected nodes after
    each event
&

;console output options:
=consoleOutput
simulationData true   ;boolean: when true this shows general simulation data on
     the console
eventlog true   ;boolean: when true this shows an eventlog on the console
simulationPause false ;boolean: when true this will cause the simulation to
    pause after each simulation run
&

;base rate values settings
=rates
arNode 1    ;double number: sets the base rate for adding a node or the
    combined base rate of adding and removing a node when rates are dependent
arNode2 0   ;double number: sets the base rate for removing a node when
    independent from adding a node
arEdge 1    ;double number: sets the base rate for adding an edge or the
    combined base rate of adding and removing an edge when rates are dependent,
    also the rate for changing edge status
arEdge2 0   ;double number: sets the base rate for removing an edge when
    independent from adding an edge
swapEdge 1    ;double number: sets the base rate for swapping 2 edges (or just
    there status)
infection 1   ;double number: sets the base rate for infecting a node or the
    combined base rate of infecting and healing a node when rates are dependent
infection2 0    ;double number: sets the base rate for healing a node when
    independent from infecting a node
&

;
=arNodeRate
ratio 1     ;double number: the ratio between the base rate of adding and
```

```
    removing a node when dependent, input is ratio for adding a node to 1
    removing a node so input:1 is the used ratio
dependent true     ;boolean: when true the base rates of adding and removing a
    node are dependent and determined by the ratio
type constant    ;string (text): determines how rates are calculated/updated; '
    constant' means rates are equal to the base rates, 'n_linear' means the
    rates are linear dependent on the number of nodes, 'n_linear_add_only' means
     the rate for adding nodes is linear dependent on the number of nodes the
    rate for removing nodes is constant at base times nBalanced, '
    n_linear_remove_only' means the rate for removing nodes is linear dependent
    on the number of nodes the rate for adding nodes is constant at base times
    nBalanced
        ;'n_linear_balanced' means the rates are linear dependent and balanced
            to the base rate at nBalanced nodes in the system, '
            n_linear_balanced_add_only' means the add node rate is linear
            dependent and balanced to the base rate at nBalanced nodes in the
            system and the remove node rate is constant, '
            n_linear_balanced_remove_only'means the remove node rate is linear
            dependent and balanced to the base rate at nBalanced nodes in the
            system and the add node rate is constant, '
            constant_dependent_n_linear_balanced' means constant rates but when
            dependent on each other in such a way they favour reaching the
            balanced number of nodes nBalanced
addFactor 1    ;double number: not used in current simulation program
removeFactor 1     ;double number: not used in current simulation program
&

;
=arEdgeRate
changeStatus true ;boolean: when true the add and remove edge events are
    replaced by the change edge event that selects a random edge and changes its
     status
ratio 1      ;double number: the ratio between the base rate of adding and
    removing an edge when dependent, input is ratio for adding an edge to 1
    removing an edge so input:1 is the used ratio
dependent true     ;boolean: when true the base rates of adding and removing an
    edge are dependent and determined by the ratio
type constant    ;string (text): determines how rates are calculated/updated; '
    constant' means rates are equal to the base rates, 'e_linear' means the
    rates are linear dependent on the number of edges in the system (active and/
    or inactive), 'e_linear_balanced' means the rates are linear dependent and
    balanced to the base rate at the ideal fraction of active edges in the
    system, 'constant_dependent_e_linear_balanced' means constant rates but when
     dependent on each other in such a way they favour reaching the ideal
    fraction of active edges
addFactor 1    ;double number: not used in current simulation program
removeFactor 1     ;double number: not used in current simulation program
&

;
```

```
=swapEdgeRate
changeStatus true ;boolean: when true the event swap edge becomes swap edge
    status where two random edges swap their status instead of an active edge
    being swapped with an inactive one
type constant    ;string (text): determines how rates are calculated/updated; '
    constant' means rates are equal to the base rate, 'e_linear' means the rates
     are linear dependent on the number of (active) edges, 'e_linear_balanced'
    means the rates are linear dependent on the fraction of active edges (
    applies only to swap edge event)
swapFactor 1     ;double number: not used in current simulation program
&

=infectionRate
ratio 4        ;double number: the ratio between the base rate of infecting and
    healing a node when dependent, input is ratio for infecting a node to 1
    healing a node so input:1 is the used ratio
dependent true       ;boolean: when true the base rates of infecting and healing
     a node are dependent and determined by the ratio
infectionType constant     ;string (text): determines how rates are calculated/
    updated; 'constant' means rates are equal to the base rates, 'ni_linear'
    means the rates are linear dependent on the number of infected nodes, '
    ni_linear_balanced' means the rates are linear dependent and balanced to a
    fraction of nBalanced nodes in the system determined by the infection factor
    , 'ni_ne_linear' means the rates are linear dependent on the number of
    infected nodes times the number of edges connected to each infected node, '
    ni_ne_linear_balanced' means the same as 'ni_linear_dependent but than also
    dependent on the number of edges connected to each infected node
infectionFactor 0.2    ;double number: fraction to determine the fraction of
    nBalanced nodes to which the base rate is set for balanced types of rate
    calculations
&

; end of inputfile
```

A.3 default options for simulation

input file with the default options for the fixed random network:

```
;general uptions:
=general
numberOfSimulations 1000
nBalanced 1000
nMax 2000
nMin 0
edgeFraction 0.5
fixedSeed 0
&

;start of simulation options:
```

```
=startOfSimulation
n 1000
activeEdges full
&

;stopping criteria options:
=stoppingCriteria
time 10
events 10000
virusExtinction true
timeOn false
&

;file output options
=fileOutput
simulationData false
eventlog false
nodeInfo false
edgeInfo false
ratesInfo false
virusSpread false
&

;console output options:
=consoleOutput
simulationData false
eventlog false
simulationPause false
&

;base rate values settings
=rates
arNode 0
arNode2 0
arEdge 0
arEdge2 0
swapEdge 0
infection 1
infection2 0
&

;
=arNodeRate
ratio 1
dependent true
type constant
addFactor 1
removeFactor 1
&
```

```
;
=arEdgeRate
changeStatus true
ratio 1
dependent true
type constant
addFactor 1
removeFactor 1
&

;
=swapEdgeRate
changeStatus true
type constant
swapFactor 1
&

=infectionRate
ratio 1.5
dependent true
infectionType ni_linear
infectionFactor 0.2
&

; end of inputfile
```

input file with the default options for the linear rates random network:

```
;general uptions:
=general
numberOfSimulations 1000
nBalanced 100
nMax 200
nMin 0
edgeFraction 0.5
fixedSeed 0
&

;start of simulation options:
=startOfSimulation
n 100
activeEdges balanced
&

;stopping criteria options:
=stoppingCriteria
time 10
events 25000
virusExtinction true
timeOn false
&
```

```
;file output options
=fileOutput
simulationData false
eventlog false
nodeInfo false
edgeInfo false
ratesInfo false
virusSpread false
&

;console output options:
=consoleOutput
simulationData false
eventlog false
simulationPause false
&

;base rate values settings
=rates
arNode 1
arNode2 0
arEdge 0.01
arEdge2 0
swapEdge 0.01
infection 10
infection2 0
&

;
=arNodeRate
ratio 1
dependent true
type n_linear_remove_only
addFactor 1
removeFactor 1

;
=arEdgeRate
changeStatus true
ratio 1
dependent true
type e_linear
addFactor 1
removeFactor 1
&

;
=swapEdgeRate
changeStatus true
```

```
type e_linear
swapFactor 1
&

=infectionRate
ratio 4
dependent true
infectionType ni_linear
infectionFactor 0.2
&

; end of inputfile
```

input file with the default options for the balanced rates random network:

```
;general uptions:
=general
numberOfSimulations 1000
nBalanced 500
nMax 1000
fixedSeed 0
&

;start of simulation options:
=startOfSimulation
n 500
activeEdges balanced
&

;stopping criteria options:
=stoppingCriteria
time 10
events 10000
virusExtinction true
timeOn false
&

;file output options
=fileOutput
simulationData false
eventlog true
nodeInfo false
edgeInfo false
ratesInfo true
virusSpread true
&

;console output options:
=consoleOutput
simulationData false
eventlog false
```

```
simulationPause false
&

;base rate values settings
=rates
arNode 1
arNode2 0
arEdge 1
arEdge2 0
swapEdge 0
infection 1
infection2 0
&

;
=arNodeRate
ratio 1
dependent true
type constant_dependent_n_linear_balanced
addFactor 1
removeFactor 1
&

;
=arEdgeRate
changeStatus false
ratio 1
dependent true
type constant_dependent_e_linear_balanced
addFactor 1
removeFactor 1
&

;
=swapEdgeRate
changeStatus true
type e_linear
swapFactor 1
&

=infectionRate
ratio 3
dependent true
infectionType ni_linear_balanced
infectionFactor 0.01
&

; end of inputfile
```

Appendix B

Simulations results

B.1 fixed random network

variable: infect/heal rate ratio

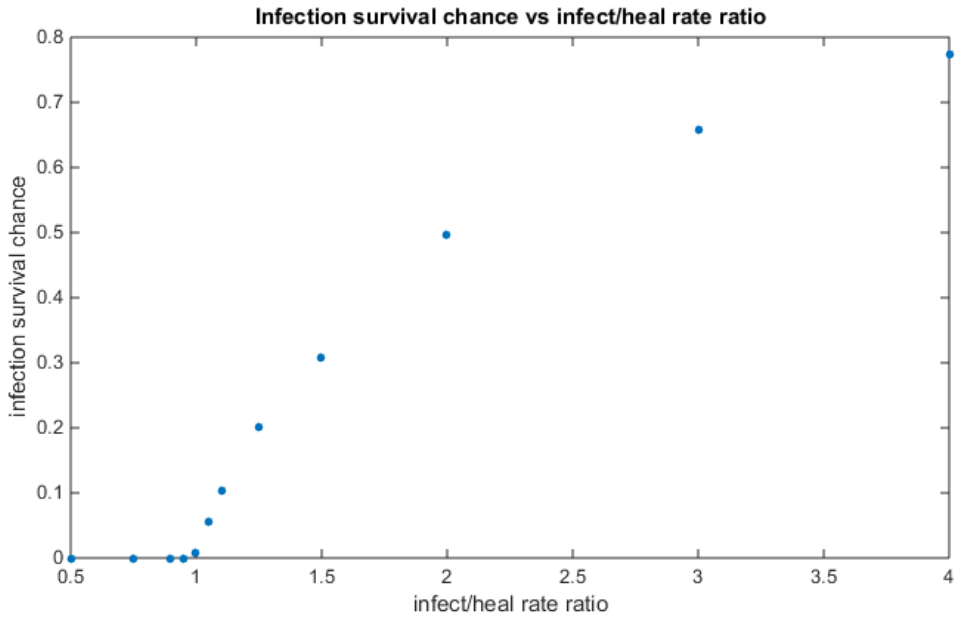

Figure B.1: infection survival chance vs infect/heal rate ratio for fixed random network

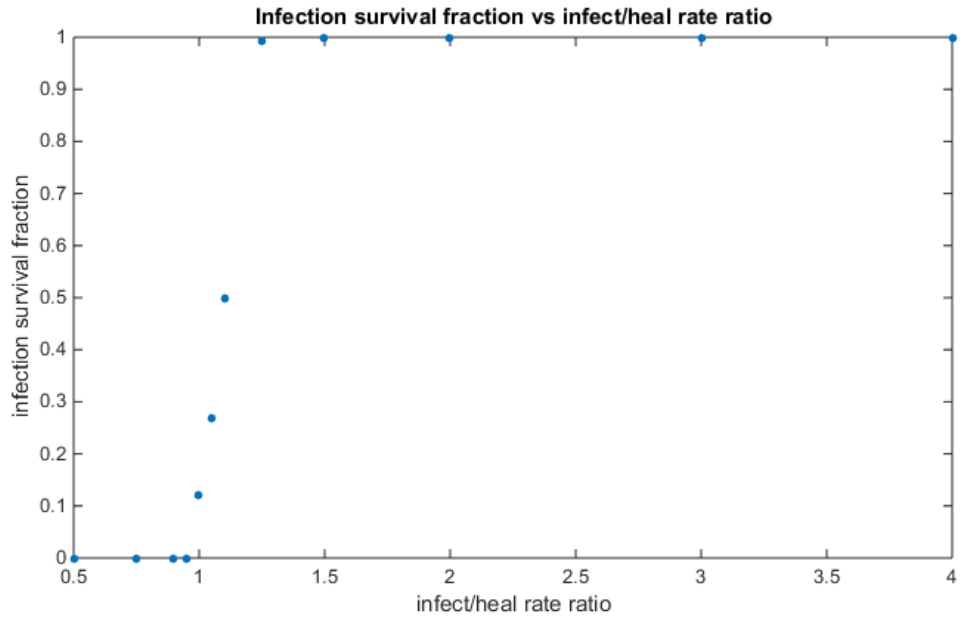

Figure B.2: infection survival fraction vs infect/heal rate ratio for fixed random network

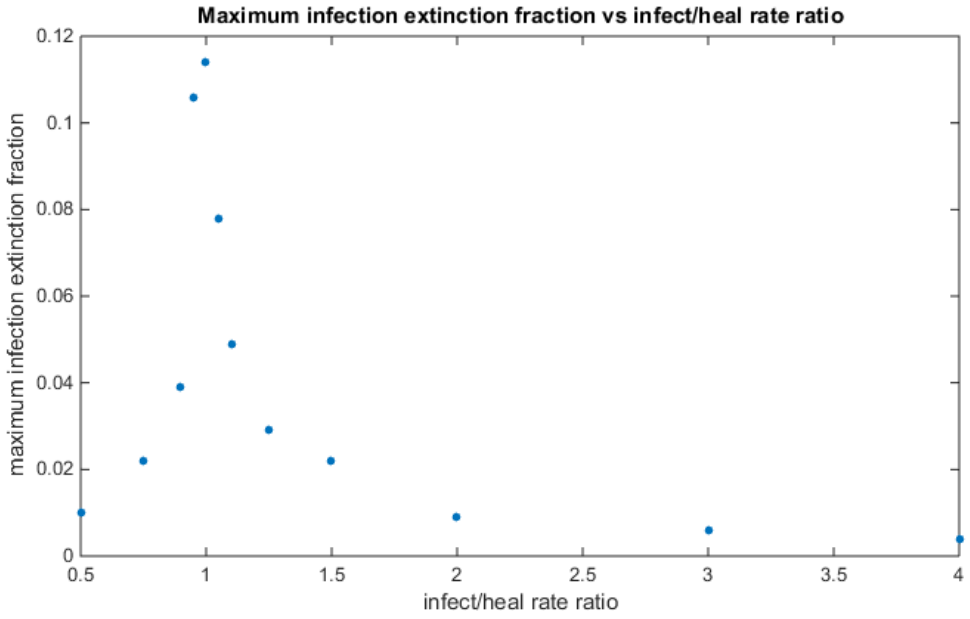

Figure B.3: maximum infection extinction fraction vs infect/heal rate ratio for fixed random network

variable: active edge fraction

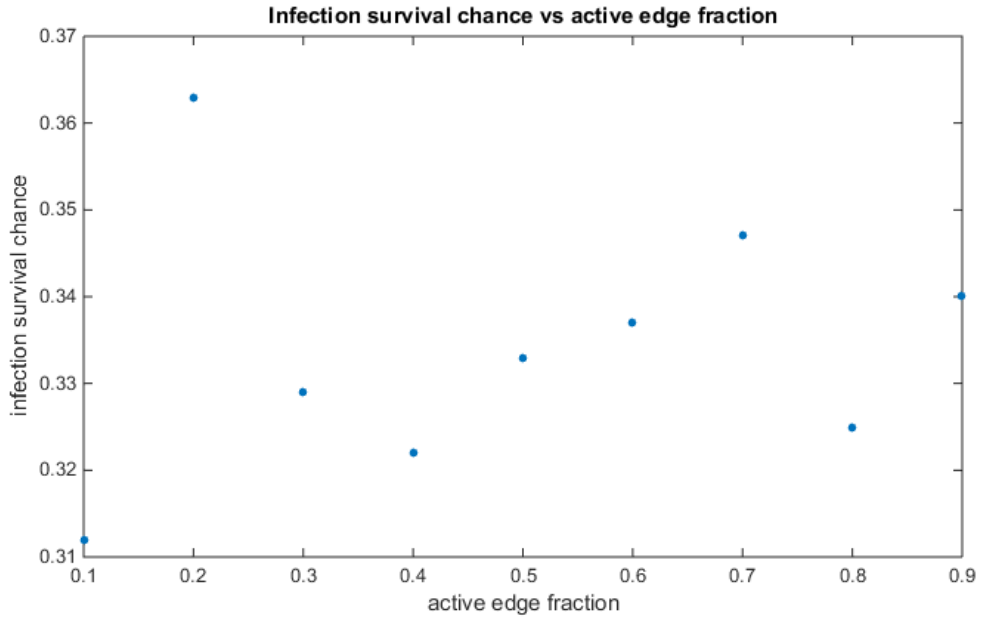

Figure B.4: infection survival chance vs active edge fraction for fixed random network

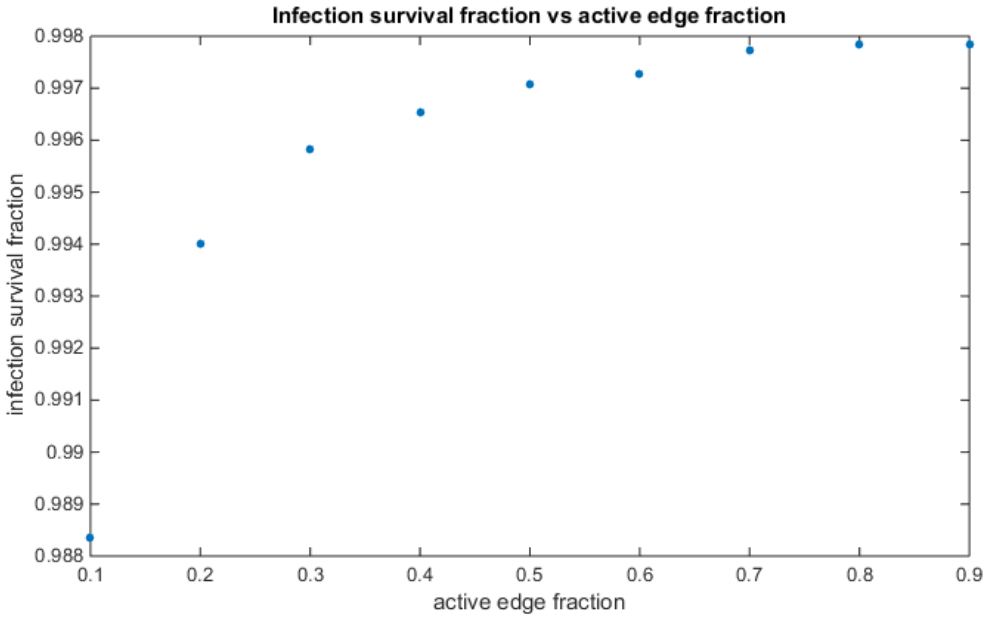

Figure B.5: infection survival fraction vs active edge fraction for fixed random network

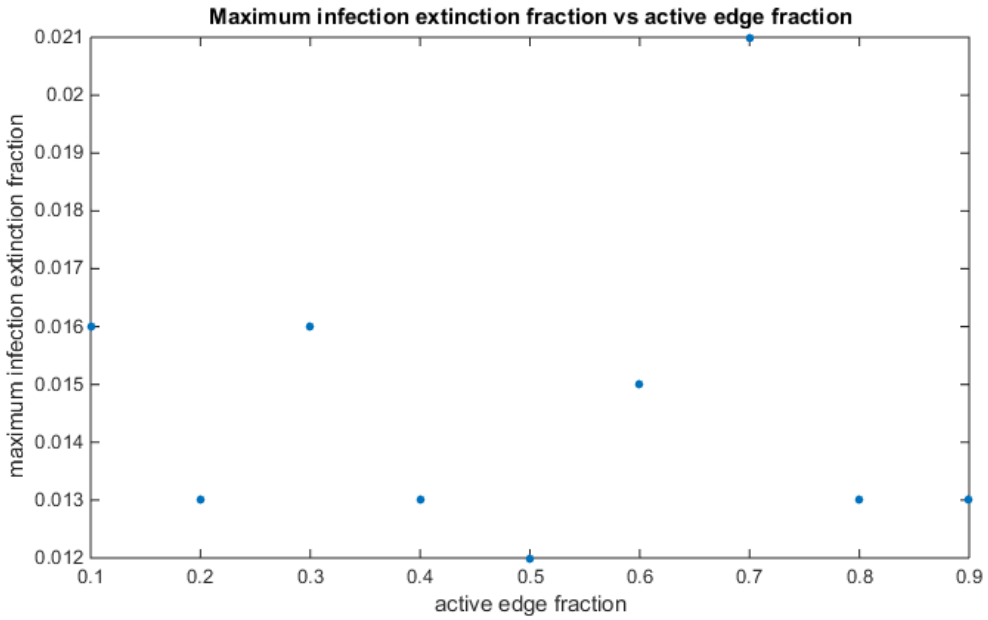

Figure B.6: maximum infection extinction fraction vs active edge fraction for fixed random network

variable: number of nodes

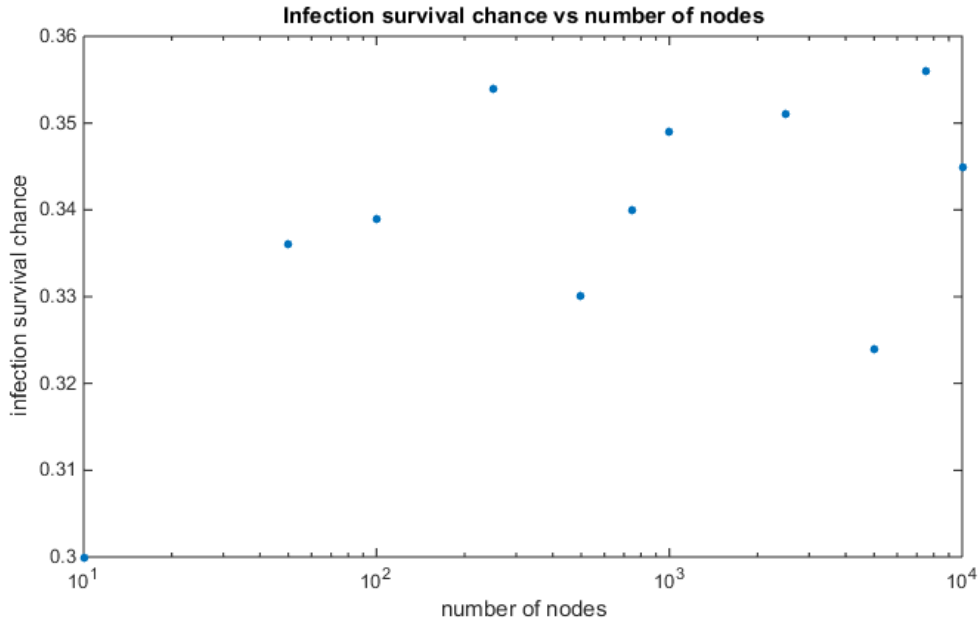

Figure B.7: infection survival chance vs number of nodes for fixed random network

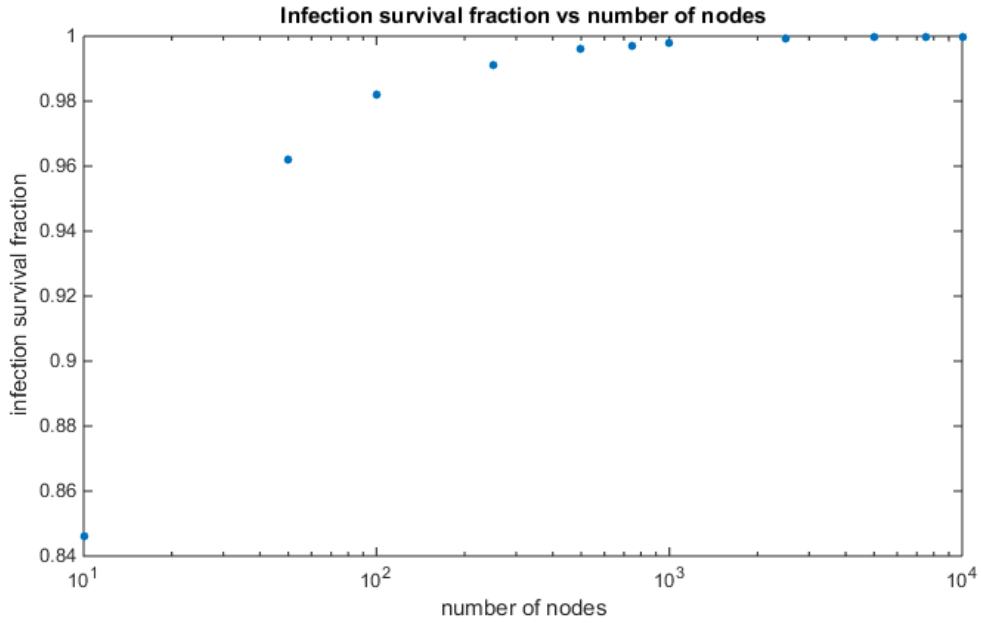

Figure B.8: infection survival fraction vs number of nodes for fixed random network

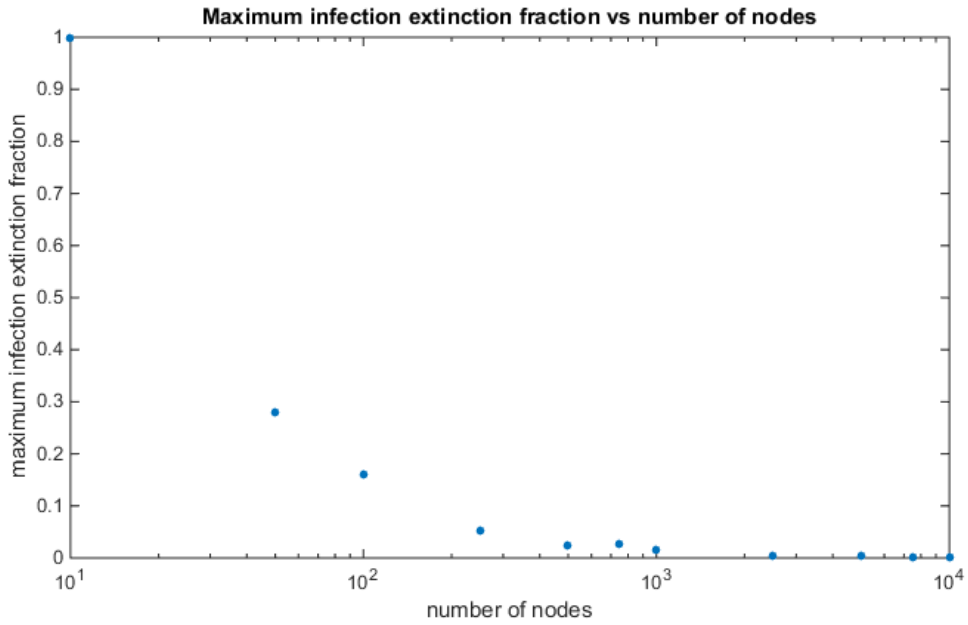

Figure B.9: maximum infection extinction fraction vs number of nodes for fixed random network

B.2 linear rates random network

variable: infect/heal rate ratio

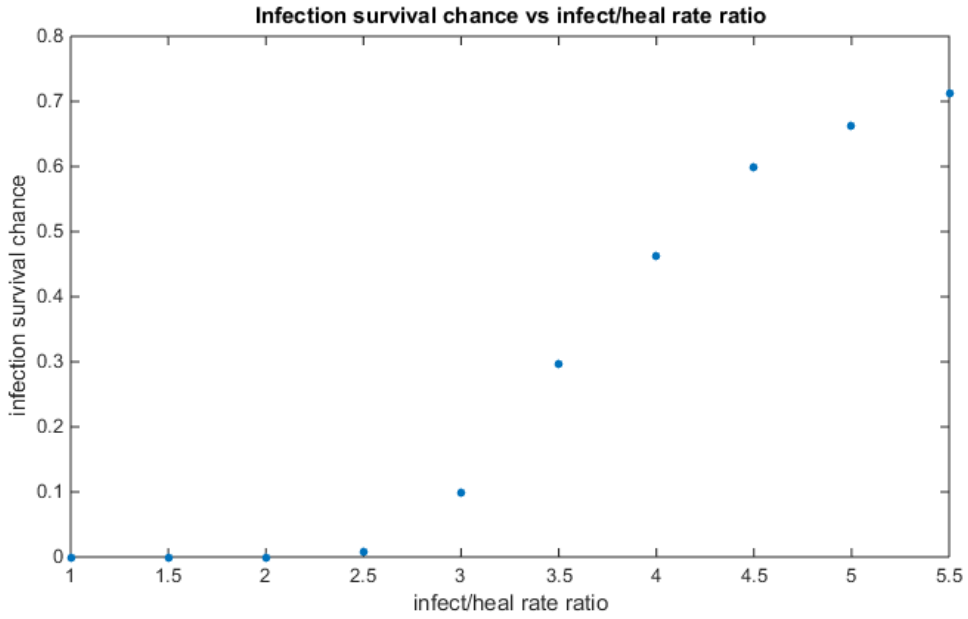

Figure B.10: infection survival chance vs infect/heal rate ratio for linear rates random network

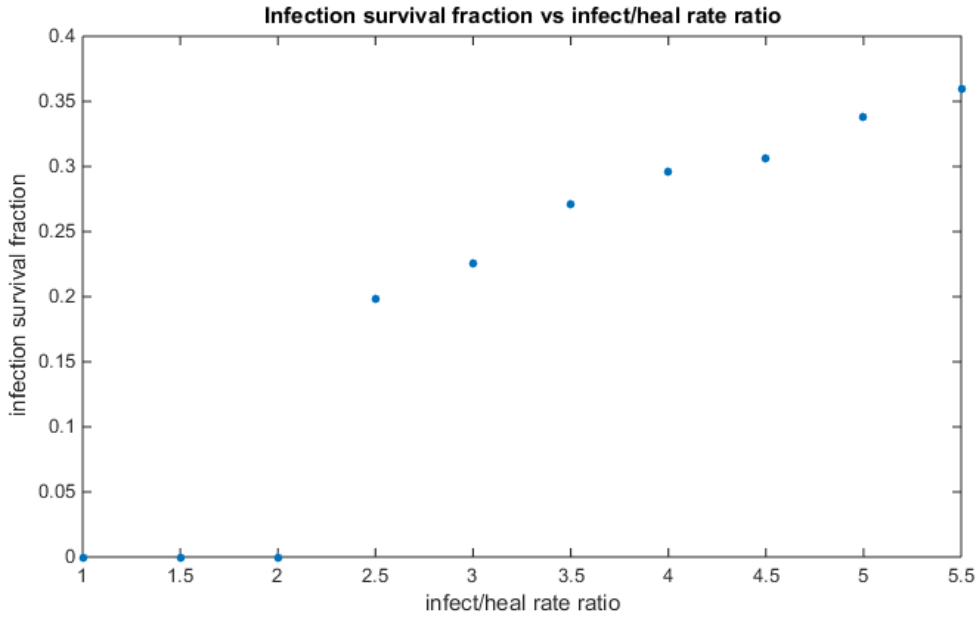

Figure B.11: infection survival fraction vs infect/heal rate ratio for linear rates random network

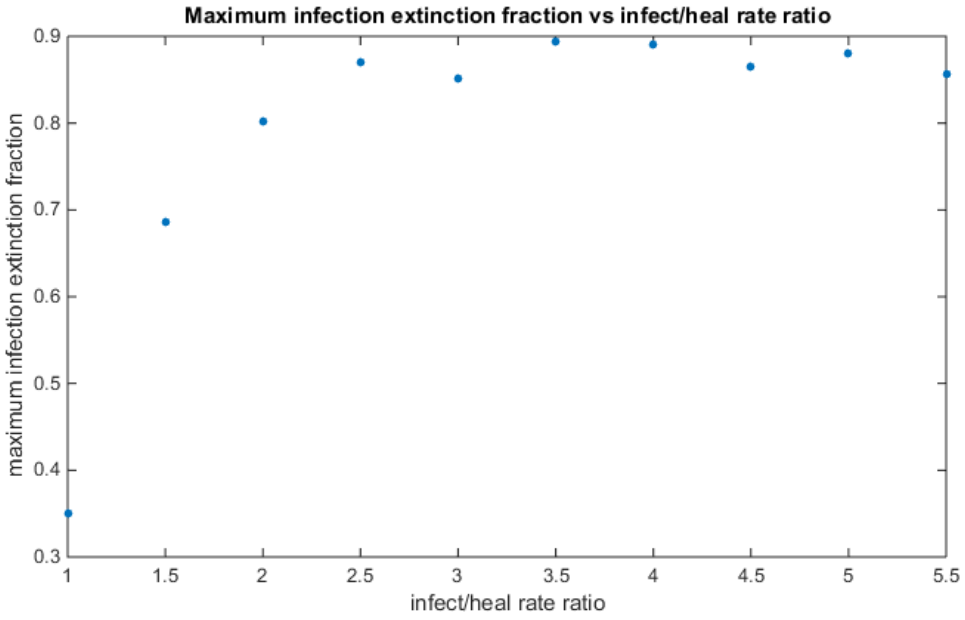

Figure B.12: maximum infection extinction fraction vs infect/heal rate ratio for linear rates random network

variable: infect/heal rate ratio without add/remove nodes

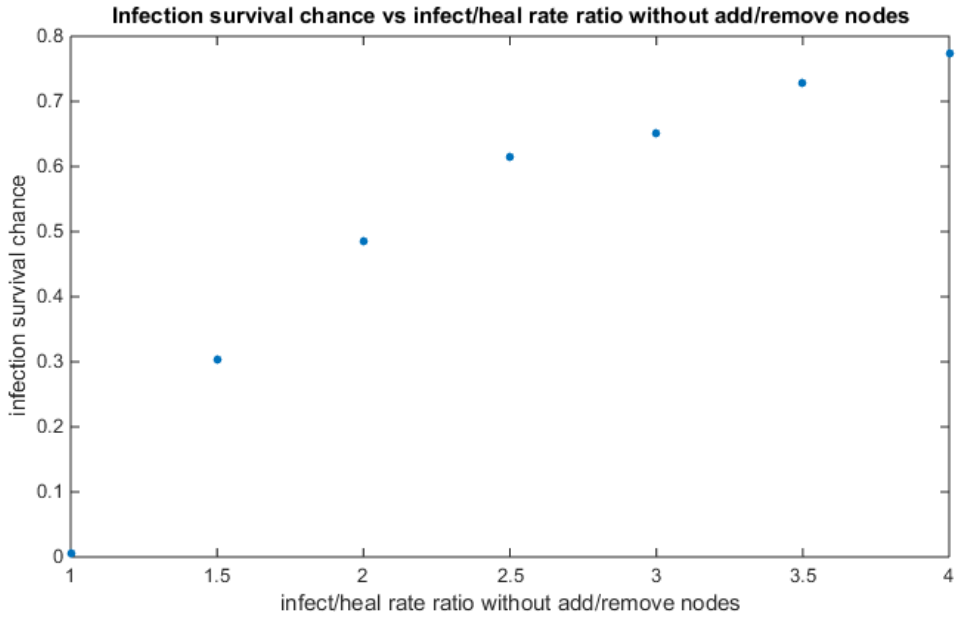

Figure B.13: infection survival chance vs infect/heal rate ratio without add/remove nodes for linear rates random network

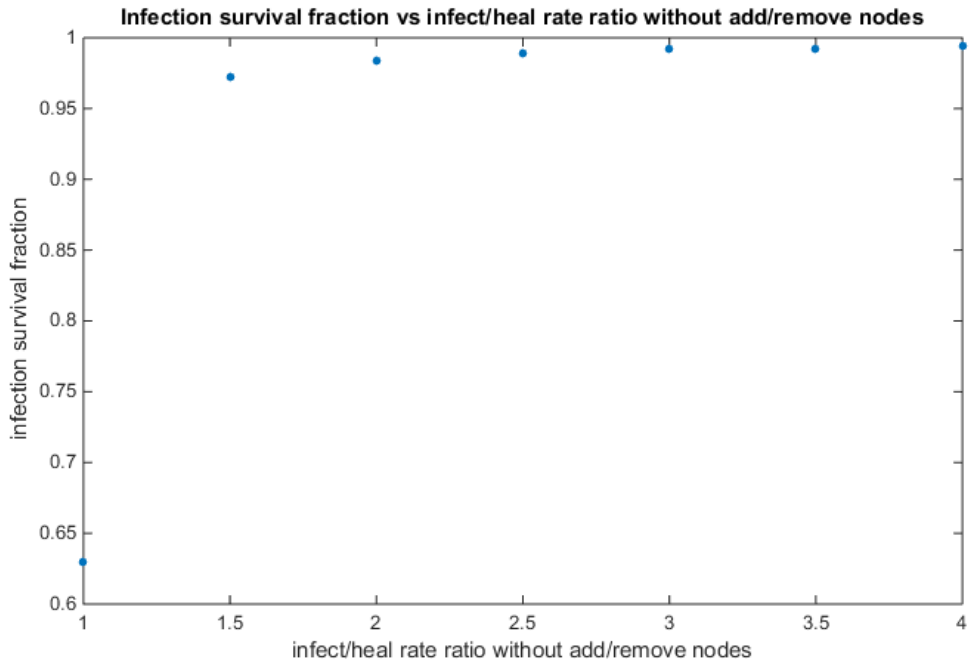

Figure B.14: infection survival fraction vs infect/heal rate ratio without add/remove nodes for linear rates random network

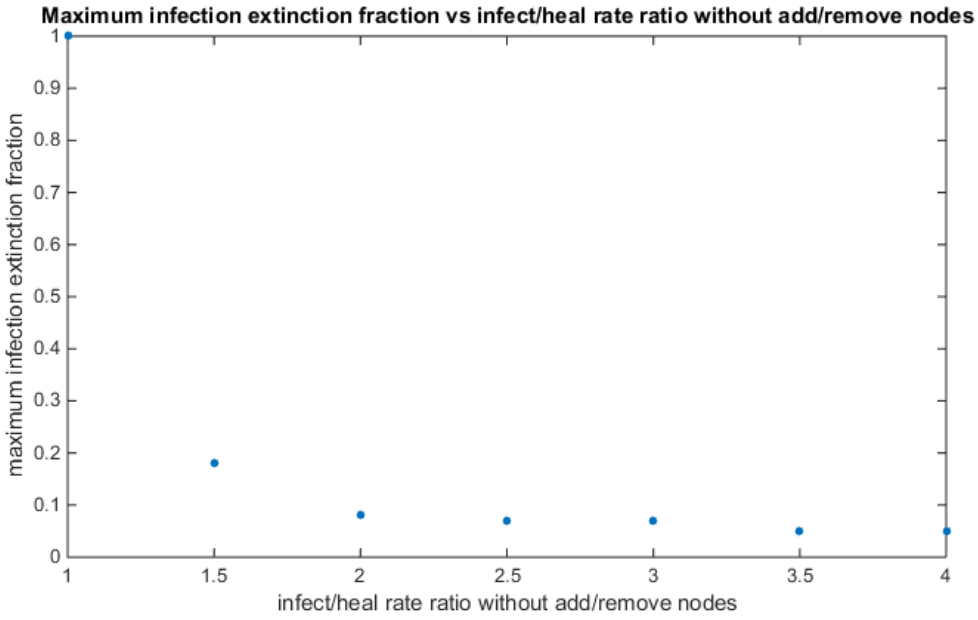

Figure B.15: maximum infection extinction fraction vs infect/heal rate ratio without add/remove nodes for linear rates random network

variable: add/remove base rate

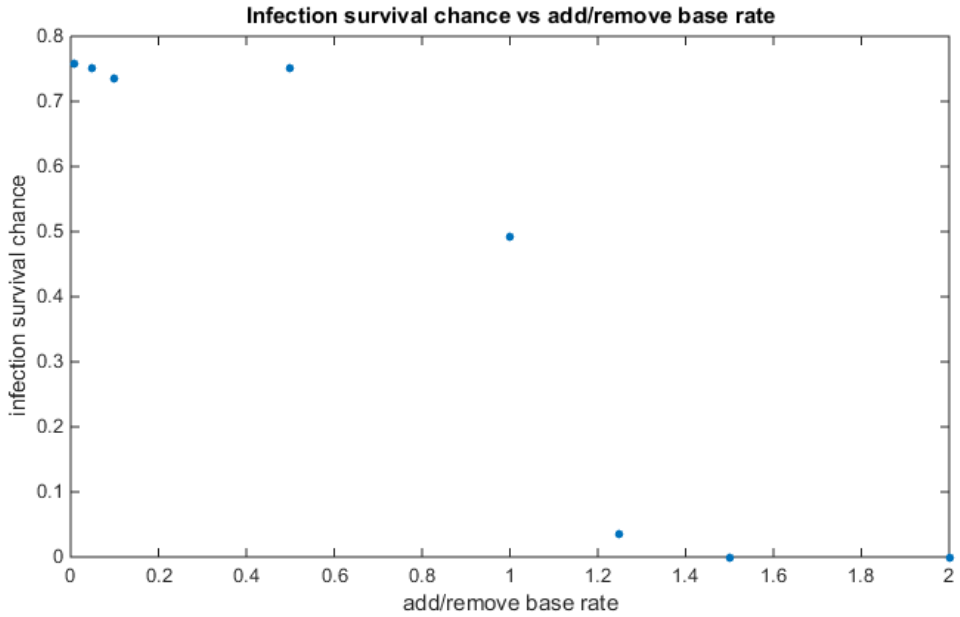

Figure B.16: infection survival chance vs add/remove base rate for linear rates random network

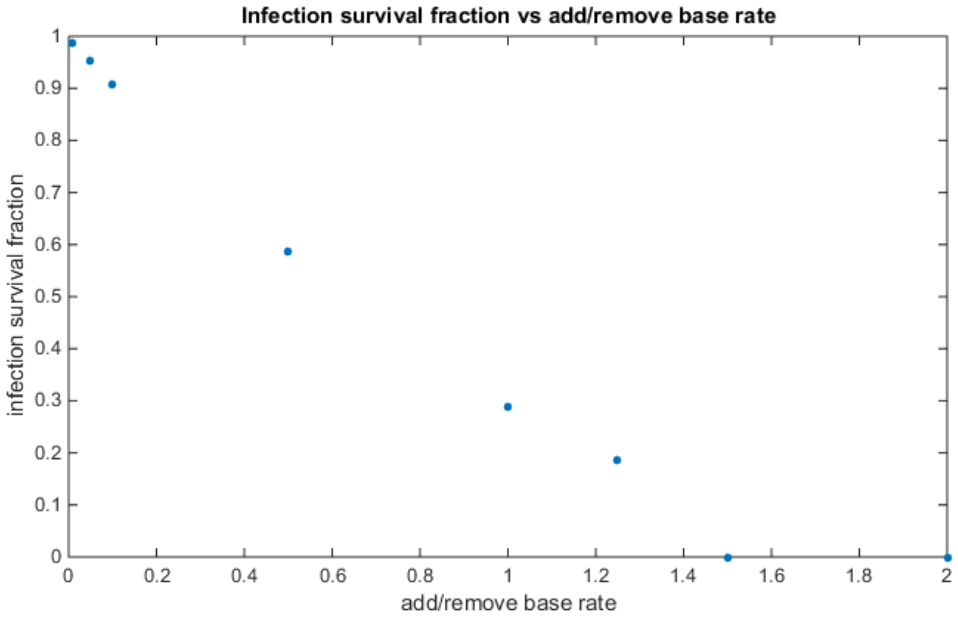

Figure B.17: infection survival fraction vs add/remove base rate for linear rates random network

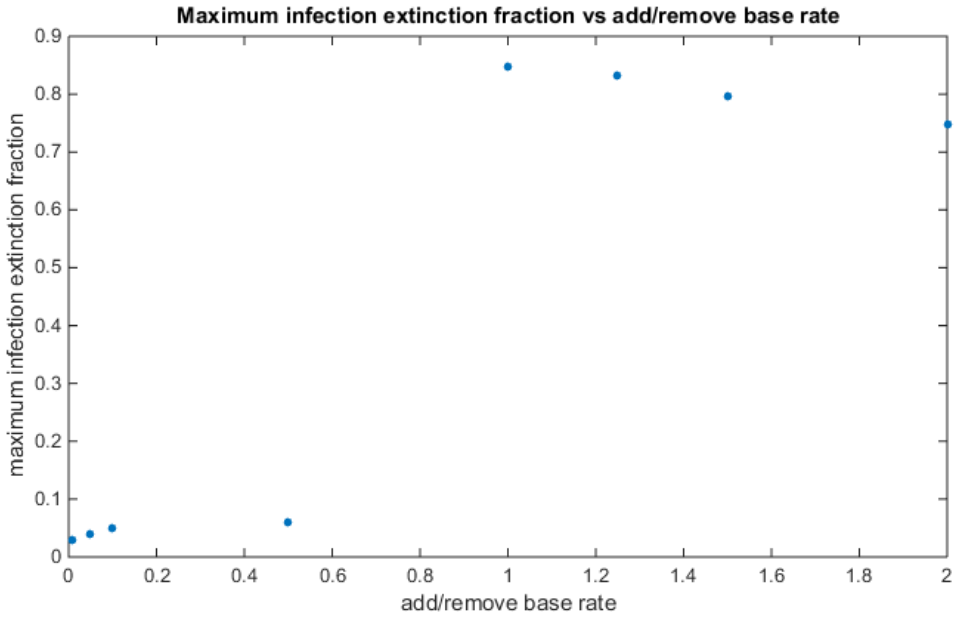

Figure B.18: maximum infection extinction fraction vs add/remove base rate for linear rates random network

B.3 balanced rates random network

variable: infect/heal rate ratio

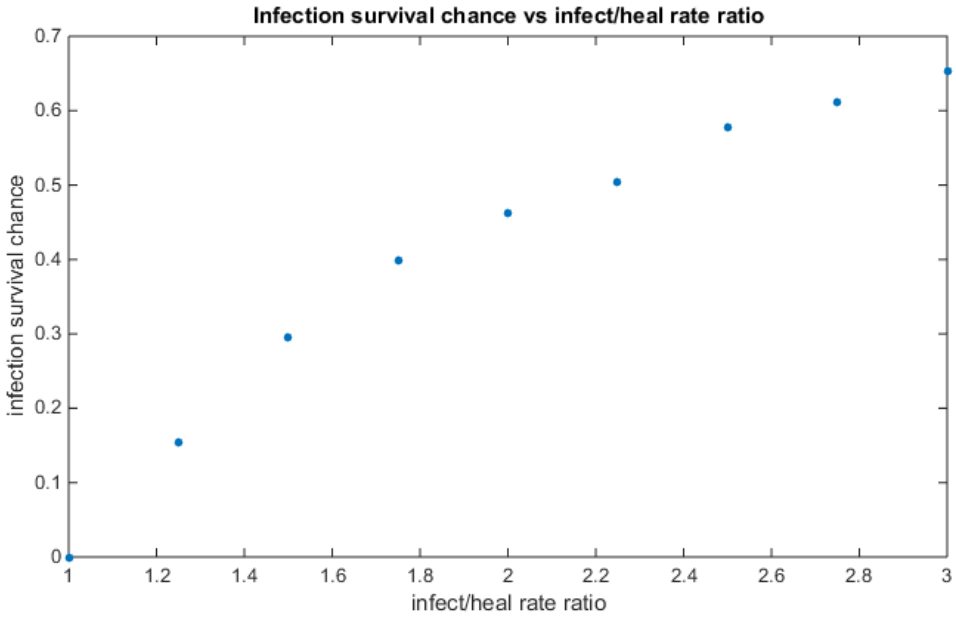

Figure B.19: infection survival chance vs infect/heal rate ratio for balanced rates random network

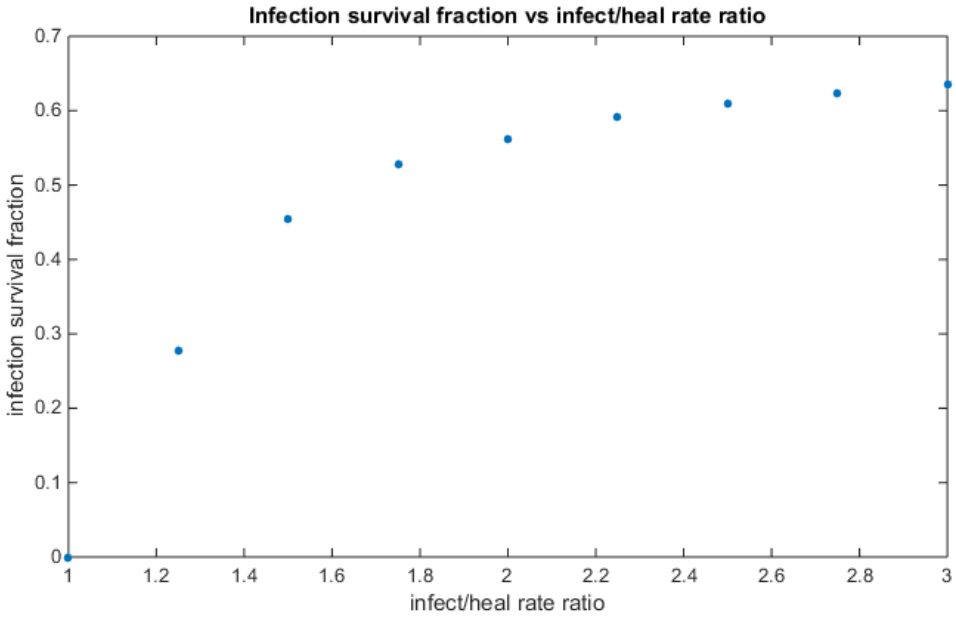

Figure B.20: infection survival fraction vs infect/heal rate ratio for balanced rates random network

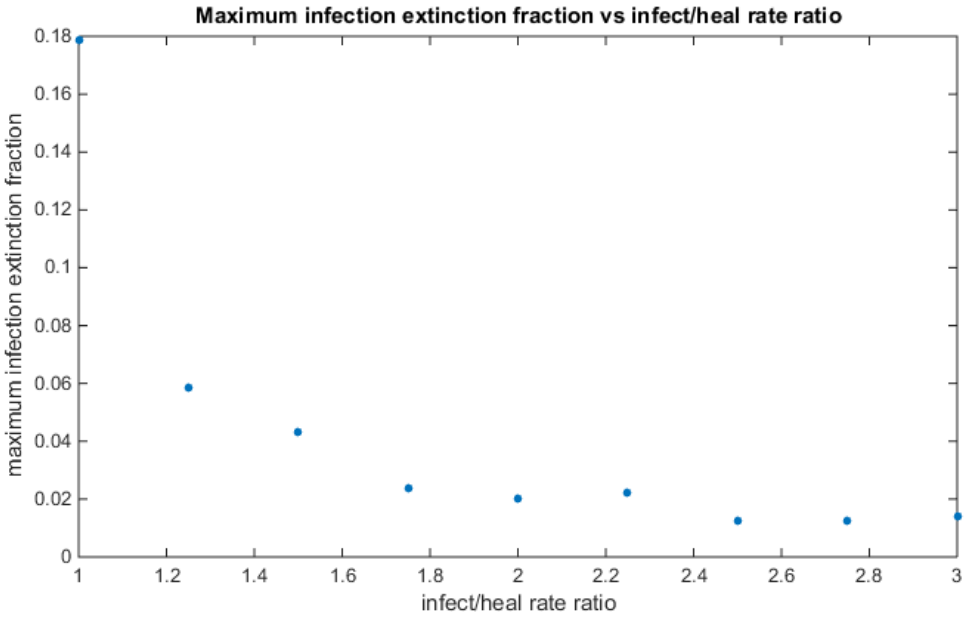

Figure B.21: maximum infection extinction fraction vs infect/heal rate ratio for balanced rates random network

variable: infect/heal rate fraction

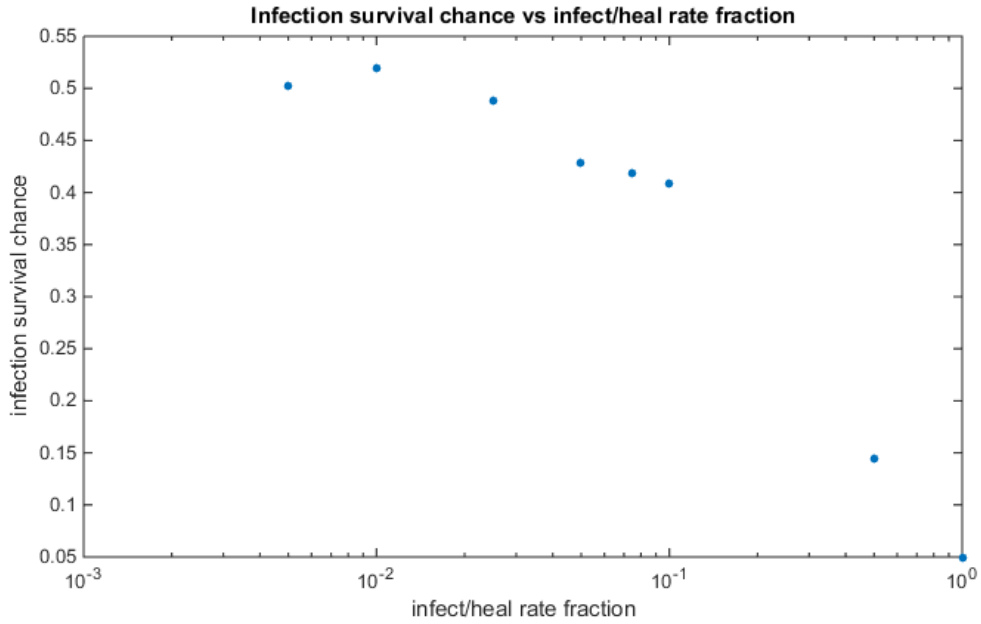

Figure B.22: infection survival chance vs infect/heal rate fraction for balanced rates random network

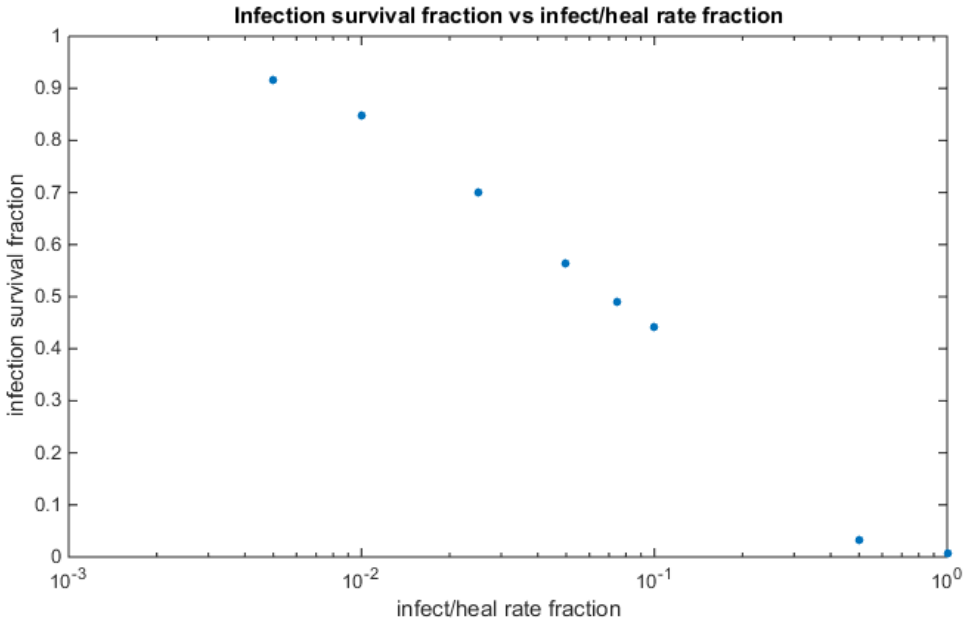

Figure B.23: infection survival fraction vs infect/heal rate fraction for balanced rates random network

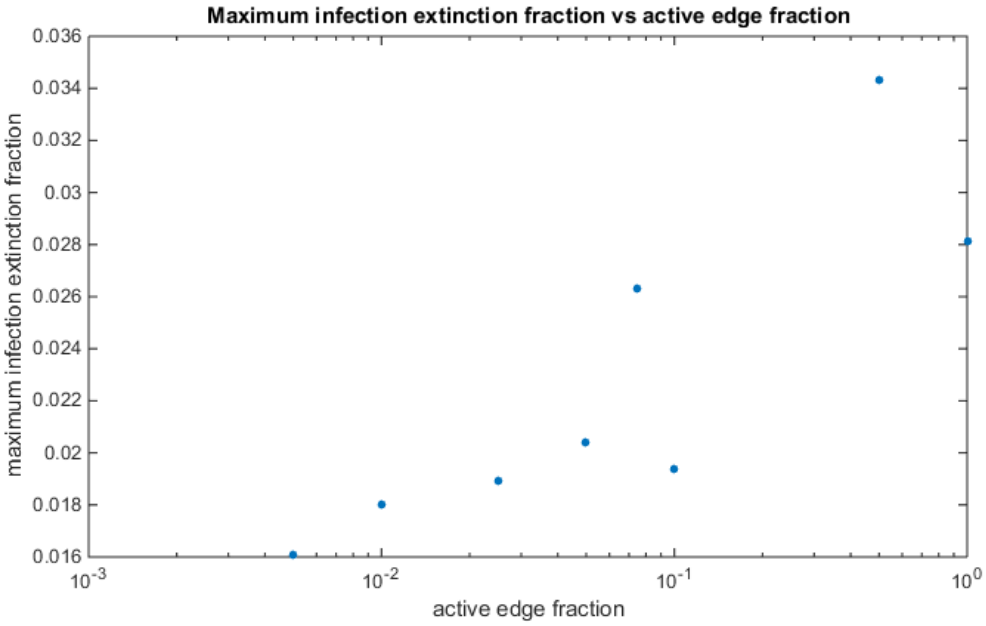

Figure B.24: maximum infection extinction fraction vs infect/heal rate fraction for balanced rates random network

variable: active edge fraction

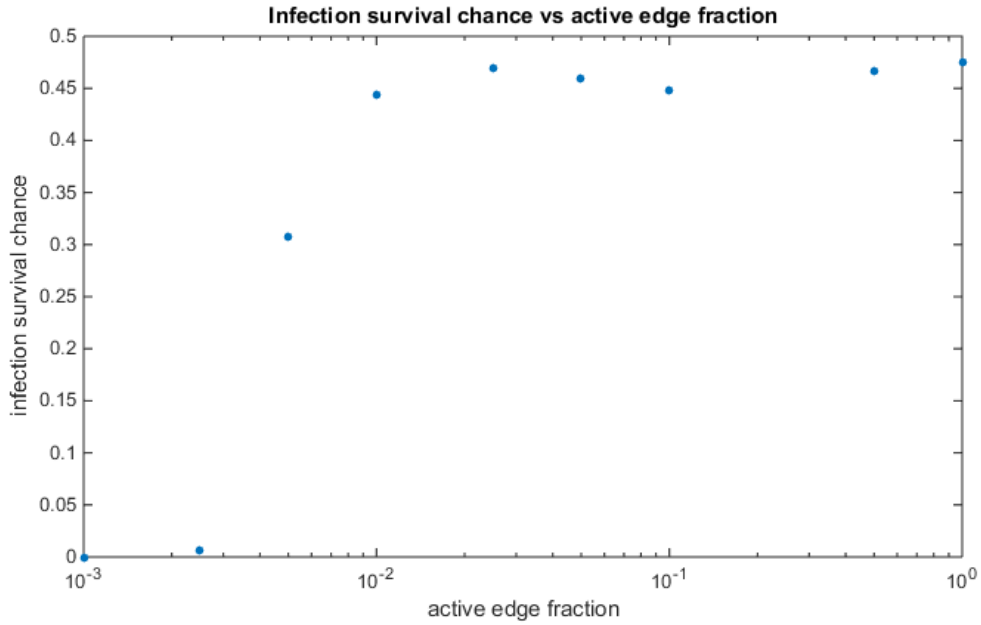

Figure B.25: infection survival chance vs active edge fraction for balanced rates random network

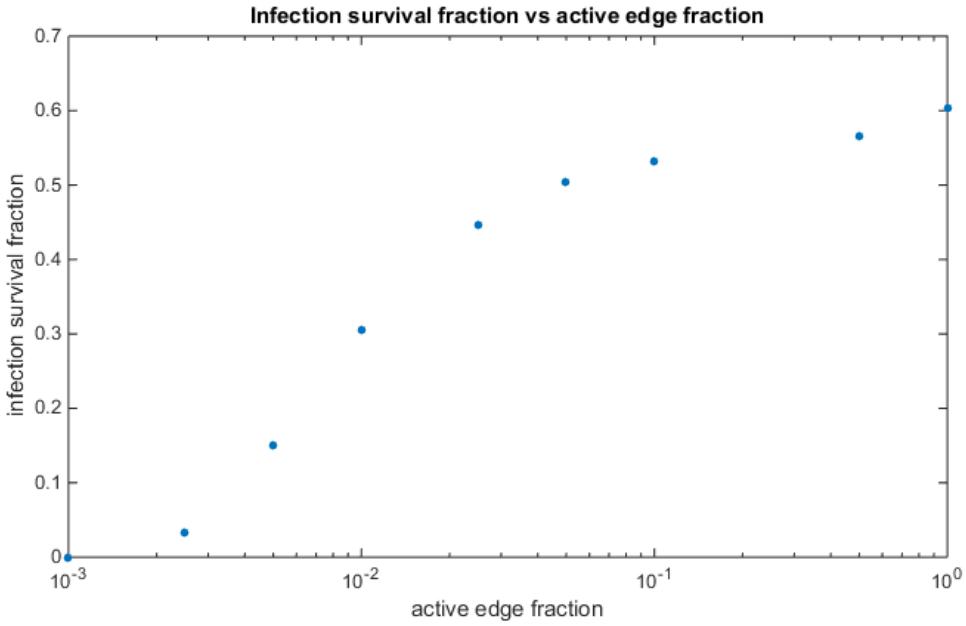

Figure B.26: infection survival fraction vs active edge fraction for balanced rates random network

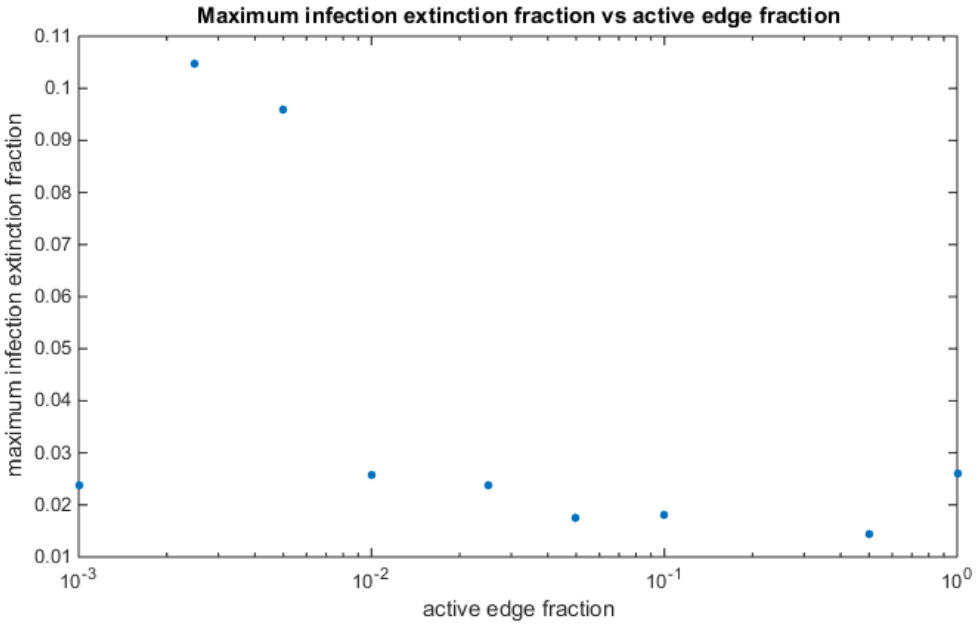

Figure B.27: maximum infection extinction fraction vs active edge fraction for balanced rates random network

Appendix C

C++ code

C.1 Code file descriptions

main.cpp: This file is where the simulation starts and finishes. It iterates the simulation runs until they are all finished as specified by the options. Main.cpp will also create the data structures for the node and edge information and reserve memory to hold this information. If there is no input data file in the directory of the executable this file will call to create a new input file and closes the program.

simulation.cpp: A simulation is initialised and started in this file. It will then iterate through choosing events until the stopping criteria are met. After choosing an event it will call the corresponding function in event.cpp to execute the event and update the system. Also the functions to update the rates using the new system state data are called from here.

event.cpp: In this file all the functions for every event can be found. when an event is determined the corresponding event function chooses which nodes and/or edges are going to carry out the event if available. Then the event is executed and if successful the node and edge data will be updated.

rng.cpp: The Mersenne Twister random generator is initiated and seeded here and this file has several functions that use the random number generator used to choose between events, nodes or edges for example.

rates.cpp: The rates for the random network are calculated here using the current system state. It also contains a function to initiate the rates that are going to be used in the simulation

input.cpp: this file has functions to read and write an input.dat data file. When reading it updates the options data to the user specified input in the input.dat files if these are

valid.

output.cpp: this file writes various output files to both the console application as .dat files. The options determine which output is generated except for results.dat which is always generated.

Node.h and Node.cpp: Node.h defines the class structure that holds data for each node that will be created by the program. Node.cpp initiates the values for the global (or static) variables used by the Node class like the number of nodes, number of active edges and number of infected nodes.

datastructs.h and datastructs.cpp: datastructs.h defines several struct and class structures to hold data created by the program like edge information, rates and simulation data. Datastructs.cpp initiates the values for simulation data such as simulation and event number.

options.h and options.cpp: Options.h defines a class that holds all option values that can be set by the input.dat files. Options.cpp creates one instance of the options class that will be used in the rest of the program to get the chosen options (or set the options according to the input file readings by input.cpp)

simulation.h, event.h, rng.h, rates.h, input.h and output.h: These header files are only used to forward declare (some of) their functions of the corresponding .cpp files so they can be used by other files.

C.2 Complete code

C.2.1 main.cpp:

```
// main.cpp : Defines the entry point for the console application.
//

#include "stdafx.h"

// headers included with compiler
#include <iostream>
#include <fstream>
#include <cstdint>

// headers
#include "simulation.h"
#include "input.h"
#include "output.h"
#include "options.h"
#include "Node.h"
#include "datastructs.h"
#include "event.h"
```

```
void reserveDataStructures(NodeVector_t &nodeList, NodeVector_t &nodeListInactive,
    EdgeMatrix_t &edgeData)
{
  // reserve memory for the active and inactive node lists
  nodeList.reserve(options.GetGeneral_nMax());
  nodeListInactive.reserve(options.GetGeneral_nMax());

  // reserve memory for edge matrix
  edgeData.reserve(options.GetGeneral_nMax());
  for (int32_t irow = 0; irow < options.GetGeneral_nMax(); ++irow)
  {
    EdgeVector_t edgeVector;
    edgeData.push_back(edgeVector);
    edgeData[irow].reserve(options.GetGeneral_nMax());
  }
  edgeData.clear();

  // reserve memory for shorlists used in event.cpp to preselect nodes/edges for (
      weighted) random choice
  reserveShortlist();
}

void clearDataStructures(NodeVector_t &nodeList, NodeVector_t &nodeListInactive,
    EdgeMatrix_t &edgeData)
{
  nodeList.clear(); //clear nodelist for next simulation
  nodeListInactive.clear();
  edgeData.clear();
}

int main()
 {

  if (std::ifstream("input.dat"))
  {
    removeOutput();

    readInput();

    options.printInput();

    openOutput();
    printOptions();
    printTitleResults();



    // define a vector to hold nodes and a vector to hold inactive nodes.
    NodeVector_t nodeList;
    NodeVector_t nodeListInactive;

    // define a matrix to hold edge information.
    EdgeMatrix_t edgeData;

    // reserve memory for all data structures
    reserveDataStructures(nodeList, nodeListInactive, edgeData);
```

```
    system("pause");

    int32_t simulationNumber = 1;

    options.GetGeneral_numberOfSimulations();

    do
    {
      simulation(nodeList, nodeListInactive, edgeData);
      clearDataStructures(nodeList, nodeListInactive, edgeData);
      ++simulationNumber;
    } while (simulationNumber <= options.GetGeneral_numberOfSimulations());

    if(!options.GetConsoleOutput_simulationPause()) system("pause");
  }

  else
  {
    createInput();

    system("pause");
  }


  return 0;
}
```

C.2.2 simulation.cpp:

```
// simulation.cpp : runs 1 complete simulation.
//

#include "stdafx.h"

// headers included with compiler
#include <cstdint>
#include <vector>
#include <iostream>
#include <numeric>

// headers
#include "Node.h"
#include "event.h"
#include "rng.h"
#include "rates.h"
#include "datastructs.h"
#include "output.h"
#include "options.h"

// creates the inital nodes lists filled with nodes
void createInitialNodes(NodeVector_t &nodeList, NodeVector_t &nodeListInactive)
{
  for (int32_t count = 0; count < Node::GetN(); count++)
  {
    nodeList.push_back ( Node() );
  }
```

```
}

// creates the initial edge matrix
void createInitialEdgeData(EdgeMatrix_t &edgeData)
{
  for (int32_t irow = 0; irow < Node::GetN(); ++irow)
  {
    EdgeVector_t edgeVector;
    edgeData.push_back(edgeVector);
    for (int32_t icol = 0; icol < Node::GetN(); ++icol)
    {
      edgeData[irow].push_back( Edge() );
    }
  }
}

// full system
void createFullSystem(EdgeMatrix_t &edgeData, NodeVector_t &nodeList)
{
  for (int32_t irow = 0; irow < Node::GetN(); ++irow)
  {
    for (int32_t icol = 0; icol < Node::GetN(); ++icol)
    {
      if (irow != icol)
      {
        edgeData[irow][icol].SetStatus(true);
        Node::IncreaseE();
      }
    }

    nodeList[irow].SetNumberOfEdges(Node::GetN() - 1);
  }
}

//balanced system
void createBalancedSystem(EdgeMatrix_t &edgeData, NodeVector_t &nodeList)
{
  double balance = options.GetGeneral_edgeFraction();
  for (int32_t irow = 0; irow < Node::GetN(); ++irow)
  {
    for (int32_t icol = 0; icol < irow; ++icol)
    {
      if (getRandom01() < balance)
      {
        edgeData[irow][icol].SetStatus(true);
        edgeData[icol][irow].SetStatus(true);
        nodeList[irow].IncreaseNumberOfEdges();
        nodeList[icol].IncreaseNumberOfEdges();
        Node::IncreaseE();
      }
    }
  }
}

void createInfectedSystem(NodeVector_t &nodeList)
{
  if (Node::GetN != 0)
  {
```

```
    int32_t value = getUniformInteger(nodeList.size());
    nodeList[value].ChangeStatus(NodeStatus::INFECTED);
    Node::IncreaseNInfected();
  }
}

bool getStoppingCriteria()
{
  if (options.GetStoppingCriteria_timeOn())
  {
    if (SimulationValues::GetTime() >= options.GetStoppingCriteria_time()) return
        false;
  }

  if (SimulationValues::GetEventNumber() >= options.GetStoppingCriteria_events())
      return false;

    if (options.GetStoppingCriteria_virusExtinction())
  {
    if (Node::GetNInfected() == 0) return false;
  }

  return true;
}

// function to determine time of next event
double getTime(const RateVector_t &rates)
{
  double sum(0.0);

  for (uint32_t it(0); it < rates.size(); ++it)
  {
    sum += rates[it].value;
  }

  return getExponentialRandomNumber(sum);
}

RateType chooseEvent(const RateVector_t rates)
{
  int32_t value = getWeightedUniformReal(rates);
  return rates[value].type;
}

void determineHighestInfectionFraction()
{
  double fraction = static_cast<double>(Node::GetNInfected()) / Node::GetN();
  if (fraction > SimulationValues::GetHighestInfectionFraction())
    SimulationValues::SetHighestInfectionFraction(fraction);
}

// Executes the simulation process
int32_t simulation(NodeVector_t &nodeList, NodeVector_t &nodeListInactive,
    EdgeMatrix_t &edgeData)
{
  // set random generator
  setRandomGenerator();
```

```
SimulationValues::SetTime(0);
SimulationValues::IncreaseSimulationNumber();
SimulationValues::SetEventNumber(0);
SimulationValues::SetHighestInfectionFraction(0);

Node::SetN(options.GetStartOfSimulation_n());       // set n to starting number of
    nodes
Node::SetE(0);
Node::SetNInfected(0);
Node::SetIDGenerator(0);

Result result;    // can be used to check results from events (error codes or
    different outcomes etc)
RateVector_t rates; // vector with all the rates used to determine time of next
    event and event choice

createInitialNodes(nodeList, nodeListInactive);
createInitialEdgeData(edgeData);

// turn some or all edges active based on starting options and edge fraction option
if (options.GetStartOfSimulation_activeEdges() == "balanced") createBalancedSystem(
    edgeData, nodeList);
else if (options.GetStartOfSimulation_activeEdges() == "full") createFullSystem(
    edgeData, nodeList);

// virus infection on starting system
createInfectedSystem(nodeList);

// initiate and update rates for first event
initiateRates(rates);
updateNodeRates(rates);
updateEdgeRates(rates);
updateVirusRates(rates, nodeList);

if (rates.size() == 0)
{
  std::cout << "WARNING: no events selected, time is set to max time and events is
      set higher than max events\n";
  SimulationValues::SetTime(options.GetStoppingCriteria_time());
  SimulationValues::SetEventNumber(options.GetStoppingCriteria_events() + 1);
}

determineHighestInfectionFraction();

// output text
printTitle();
printBaseRates(rates);
printStartOfSimulation(nodeList, nodeListInactive, edgeData);
printNodeData(nodeList, nodeListInactive);
printEdgeData(edgeData);



// execute events until a certain time limit is reached
while (getStoppingCriteria())
{
  double ratesSum = 0;
  for (uint32_t it = 0; it < rates.size(); it++)
```

```
    ratesSum += rates[it].value;

  if (ratesSum <= 0)
  {
    SimulationValues::SetTime(options.GetStoppingCriteria_time());
    std::cout << "No events can occur anymore, end of simulation\n";
  }
  else
  {
    SimulationValues::IncreaseEventNumber();

    SimulationValues::SetTime(SimulationValues::GetTime() + getTime(rates)); // get
        time of next event

    RateType eventType = chooseEvent(rates); // choose event

    switch (eventType)
    {
    case RateType::ADD_NODE: // add node event
    {
      result = addNode(nodeList, edgeData);
      updateNodeRates(rates);
      updateEdgeRates(rates);
      updateVirusRates(rates, nodeList);
      break;
    }
    case RateType::REMOVE_NODE: // remove node event
    {
      result = removeNode(nodeList, nodeListInactive, edgeData);
      updateNodeRates(rates);
      updateEdgeRates(rates);
      updateVirusRates(rates, nodeList);
      break;
    }
    case RateType::CHANGE_EDGE: // change edge status event
    {
      result = changeEdge(nodeList, edgeData);
      updateNodeRates(rates);
      updateEdgeRates(rates);
      updateVirusRates(rates, nodeList);
      break;
    }
    case RateType::ADD_EDGE:
    {
      result = addEdge(nodeList, edgeData);
      updateNodeRates(rates);
      updateEdgeRates(rates);
      updateVirusRates(rates, nodeList);
      break;
    }
    case RateType::REMOVE_EDGE:
    {
      result = removeEdge(nodeList, edgeData);
      updateNodeRates(rates);
      updateEdgeRates(rates);
      updateVirusRates(rates, nodeList);
      break;
    }
```

```
      case RateType::SWAP_EDGE_STATUS:
      {
        result = swapEdgeStatus(nodeList, edgeData);
        updateNodeRates(rates);
        updateEdgeRates(rates);
        updateVirusRates(rates, nodeList);
        break;
      }
      case RateType::SWAP_EDGE:
      {
        result = swapEdge(nodeList, edgeData);
        updateNodeRates(rates);
        updateEdgeRates(rates);
        updateVirusRates(rates, nodeList);
        break;
      }
      case RateType::INFECT_NODE:
      {
        result = infectNode(nodeList, edgeData);
        updateNodeRates(rates);
        updateEdgeRates(rates);
        updateVirusRates(rates, nodeList);
        break;
      }
      case RateType::HEAL_NODE:
      {
        result = healNode(nodeList, edgeData);
        updateNodeRates(rates);
        updateEdgeRates(rates);
        updateVirusRates(rates, nodeList);
        break;
      }
      }

      determineHighestInfectionFraction();

      printEventLog(result, nodeList, eventType); // generate output
      printNodeData(nodeList, nodeListInactive);
      printEdgeData(edgeData); // generate output
      printRatesInfo(rates);
      printVirusSpread();
    }
  }

  printResults(nodeList);
  printEndOfSimulation(nodeList, nodeListInactive, edgeData); // generate output

  if (options.GetConsoleOutput_simulationPause()) system("pause");

  return 0;
}
```

C.2.3 event.cpp:

```
// event.cpp : adds node to system.
//
```

```cpp
#include "stdafx.h"

// headers included with compiler
#include <cstdint>
#include <vector>
#include <iostream>

// headers
#include "Node.h"
#include "rng.h"
#include "rates.h"
#include "datastructs.h"
#include "options.h"

static std::vector<int32_t> shortlist;
static std::vector<int32_t> shortlist2;

//external functions:
void reserveShortlist()
{
  shortlist.reserve(options.GetGeneral_nMax());
  shortlist2.reserve(options.GetGeneral_nMax());
}

Result addNode(NodeVector_t &nodeList, EdgeMatrix_t &edgeData)
{
  Result result;

  if (Node::GetN() >= options.GetGeneral_nMax())
  {
    result.code = ErrorCode::MAX_NODES_REACHED;
    return result;
  }

  // update system
  nodeList.push_back(Node());
  Node::IncreaseN();

  result.node1 = Node::GetN() - 1;
  result.node1ID = nodeList[Node::GetN() - 1].GetID();


  EdgeVector_t edgeVector(Node::GetN() - 1);
  edgeData.push_back(edgeVector);

  for (int32_t irow(0); irow < Node::GetN(); ++irow)
  {
    edgeData[irow].push_back( Edge() );
  }

  if (!nodeList.size() == Node::GetN()) result.code = ErrorCode::UPDATE_ERROR;
  if (!edgeData.size() == Node::GetN()) result.code = ErrorCode::UPDATE_ERROR;
  else result.code = ErrorCode::OK;

  return result;
}
```

```
Result removeNode(NodeVector_t &nodeList, NodeVector_t &nodeListInactive, EdgeMatrix_t
    &edgeData)
{
  Result result;

  // choose node to remove

  int32_t value;
  if (Node::GetN() == 0)
  {
    result.code = ErrorCode::NO_NODE_AVAILABLE;
    return result;
  }
  else if (Node::GetN() <= options.GetGeneral_nMin())
  {
    result.code = ErrorCode::MIN_NODES_REACHED;
    return result;
  }
  else if (Node::GetN() == 1) value = 0;
  else value = getUniformInteger(nodeList.size());

  result.node1 = value;
  result.node1ID = nodeList[value].GetID();

  //update system
  //decrease number of edges for all nodes connected to node to be removed
  for (int32_t iter(0); iter < Node::GetN(); ++iter)
  {
    if (edgeData[value][iter].GetStatus())
    {
      nodeList[iter].DecreaseNumberOfEdges();
      Node::DecreaseE();
    }
  }

  if (nodeList[value].GetStatus() == NodeStatus::INFECTED) Node::DecreaseNInfected();

  nodeListInactive.push_back(nodeList[value]);
  nodeList.erase(nodeList.begin() + value);

  Node::DecreaseN();

  edgeData.erase(edgeData.begin() + value);
  for (int32_t iter(0); iter < Node::GetN(); ++iter)
  {
    edgeData[iter].erase(edgeData[iter].begin() + value);
  }

  if (!nodeList.size() == Node::GetN()) result.code = ErrorCode::UPDATE_ERROR;
  if (!edgeData.size() == Node::GetN()) result.code = ErrorCode::UPDATE_ERROR;
  else result.code = ErrorCode::OK;

  return result;
}

Result changeEdge(NodeVector_t &nodeList, EdgeMatrix_t &edgeData)
{
  Result result;
```

```
  // choose edge by selecting 2 nodes
  int32_t node1;
  if (Node::GetN() < 2)
  {
    result.code = ErrorCode::NO_EDGE_AVAILABLE;
    return result;
  }
  else if (Node::GetN() == 2) node1 = 0;
  else node1 = getUniformInteger(nodeList.size());

  int32_t node2;
  if (Node::GetN() == 2) node2 = 1;
  else
  {
    do
    {
      node2 = getUniformInteger(nodeList.size());
    } while (node1 == node2);
  }

  result.node1 = node1;
  result.node1ID = nodeList[node1].GetID();
  result.node2 = node2;
  result.node2ID = nodeList[node2].GetID();

  // update system
  if ( edgeData[node1][node2].GetStatus() )
  {
    edgeData[node1][node2].SetStatus(false);
    edgeData[node2][node1].SetStatus(false);
    nodeList[node1].DecreaseNumberOfEdges();
    nodeList[node2].DecreaseNumberOfEdges();
    Node::DecreaseE();
    result.code = ErrorCode::OK;
  }
  else
  {
    edgeData[node1][node2].SetStatus(true);
    edgeData[node2][node1].SetStatus(true);
    nodeList[node1].IncreaseNumberOfEdges();
    nodeList[node2].IncreaseNumberOfEdges();
    Node::IncreaseE();
    result.code = ErrorCode::OK;
  }

  return result;
}

Result addEdge(NodeVector_t &nodeList, EdgeMatrix_t &edgeData)
{
  Result result;

  //choose edge
  for (int32_t it(0); it < Node::GetN(); ++it)
  {
    if (nodeList[it].GetNumberOfEdges() < Node::GetN() - 1)
      shortlist.push_back(it);
```

```
  }

  int32_t node1;
  int32_t numberNodesAvailable = shortlist.size();

  if (numberNodesAvailable == 0)
  {
    result.code = ErrorCode::NO_EDGE_AVAILABLE;
    return result;
  }
  else if (numberNodesAvailable == 1) node1 = shortlist[0];
  else node1 = getUniformInteger(shortlist);

  shortlist.clear();

  for (int32_t it(0); it < Node::GetN(); ++it)
  {
    if (!edgeData[node1][it].GetStatus() && it != node1)
      shortlist.push_back(it);
  }

  int32_t node2;
  numberNodesAvailable = shortlist.size();
  if (numberNodesAvailable == 0)
  {
    result.code = ErrorCode::NO_EDGE_AVAILABLE;
    return result;
  }
  else if (numberNodesAvailable == 1) node2 = shortlist[0];
  else node2 = getUniformInteger(shortlist);

  shortlist.clear();

  result.node1 = node1;
  result.node1ID = nodeList[node1].GetID();
  result.node2 = node2;
  result.node2ID = nodeList[node2].GetID();

  // update system
  edgeData[node1][node2].SetStatus(true);
  edgeData[node2][node1].SetStatus(true);
  nodeList[node1].IncreaseNumberOfEdges();
  nodeList[node2].IncreaseNumberOfEdges();
  Node::IncreaseE();
  result.code = ErrorCode::OK;

  return result;
}

Result removeEdge(NodeVector_t &nodeList, EdgeMatrix_t &edgeData)
{
  Result result;

  //choose edge
  for (int32_t it(0); it < Node::GetN(); ++it)
  {
    if (nodeList[it].GetNumberOfEdges() > 0)
      shortlist.push_back(it);
```

```
  }

  int32_t node1;
  int32_t numberNodesAvailable = shortlist.size();

  if (numberNodesAvailable == 0)
  {
    result.code = ErrorCode::NO_EDGE_AVAILABLE;
    return result;
  }
  else if (numberNodesAvailable == 1) node1 = shortlist[0];
  else node1 = getUniformInteger(shortlist);

  shortlist.clear();

  for (int32_t it(0); it < Node::GetN(); ++it)
  {
    if (edgeData[node1][it].GetStatus() && it != node1)
      shortlist.push_back(it);
  }

  int32_t node2;
  numberNodesAvailable = shortlist.size();
  if (numberNodesAvailable == 0)
  {
    result.code = ErrorCode::NO_EDGE_AVAILABLE;
    return result;
  }
  else if (numberNodesAvailable == 1) node2 = shortlist[0];
  else node2 = getUniformInteger(shortlist);

  shortlist.clear();

  result.node1 = node1;
  result.node1ID = nodeList[node1].GetID();
  result.node2 = node2;
  result.node2ID = nodeList[node2].GetID();

  // update system
  edgeData[node1][node2].SetStatus(false);
  edgeData[node2][node1].SetStatus(false);
  nodeList[node1].DecreaseNumberOfEdges();
  nodeList[node2].DecreaseNumberOfEdges();
  Node::DecreaseE();
  result.code = ErrorCode::OK;

  return result;
}

Result swapEdgeStatus(NodeVector_t &nodeList, EdgeMatrix_t &edgeData)
{
  Result result;

  // choose edges by selecting 4 nodes
  int32_t node1;
  if (Node::GetN() < 3)
  {
    result.code = ErrorCode::NO_EDGE_AVAILABLE;
```

```
    return result;
  }
  else node1 = getUniformInteger(nodeList.size());

  int32_t node2;
  do
  {
    node2 = getUniformInteger(nodeList.size());
  } while (node1 == node2);

  int32_t node3 = getUniformInteger(nodeList.size());

  int32_t node4;
  if  (node3 == node1)
  {
    do
    {
      node4 = getUniformInteger(nodeList.size());
    } while (node4 == node2 || node4 == node3);
  }
  else if (node3 == node2)
  {
    do
    {
      node4 = getUniformInteger(nodeList.size());
    } while (node4 == node1 || node4 == node3);
  }
  else
  {
    do
    {
      node4 = getUniformInteger(nodeList.size());
    } while (node4 == node3);
  }

  if (edgeData[node1][node2].GetStatus() == edgeData[node3][node4].GetStatus())
  {
    result.code = ErrorCode::EDGES_HAVE_SAME_STATUS;
    return result;
  }
  else
  {
    result.node1 = node1;
    result.node1ID = nodeList[node1].GetID();
    result.node2 = node2;
    result.node2ID = nodeList[node2].GetID();
    result.node3 = node3;
    result.node3ID = nodeList[node3].GetID();
    result.node4 = node4;
    result.node4ID = nodeList[node4].GetID();
  }

  // update system
  if (edgeData[node1][node2].GetStatus())
  {
    edgeData[node1][node2].SetStatus(false);
    edgeData[node2][node1].SetStatus(false);
    nodeList[node1].DecreaseNumberOfEdges();
```

```
    nodeList[node2].DecreaseNumberOfEdges();

    edgeData[node3][node4].SetStatus(true);
    edgeData[node4][node3].SetStatus(true);
    nodeList[node3].IncreaseNumberOfEdges();
    nodeList[node4].IncreaseNumberOfEdges();

    result.code = ErrorCode::OK;
  }
  else
  {
    edgeData[node1][node2].SetStatus(true);
    edgeData[node2][node1].SetStatus(true);
    nodeList[node1].IncreaseNumberOfEdges();
    nodeList[node2].IncreaseNumberOfEdges();

    edgeData[node3][node4].SetStatus(false);
    edgeData[node4][node3].SetStatus(false);
    nodeList[node3].DecreaseNumberOfEdges();
    nodeList[node4].DecreaseNumberOfEdges();

    result.code = ErrorCode::OK;
  }

  return result;
}

Result swapEdge(NodeVector_t &nodeList, EdgeMatrix_t &edgeData)
{
  Result result;

  // choose edges by selecting 4 nodes:

  if (Node::GetN() < 3)
  {
    result.code = ErrorCode::NO_EDGE_AVAILABLE;
    return result;
  }
  // choose first node for first edge
  for (int32_t it(0); it < Node::GetN(); ++it)
  {
    if (nodeList[it].GetNumberOfEdges() > 0)
      shortlist.push_back(it);
  }

  int32_t node1;
  int32_t numberNodesAvailable = shortlist.size();

  if (numberNodesAvailable == 0)
  {
    result.code = ErrorCode::NO_EDGE_AVAILABLE;
    return result;
  }
  else if (numberNodesAvailable == 1) node1 = shortlist[0];
  else node1 = getUniformInteger(shortlist);

  shortlist.clear();
```

```
for (int32_t it(0); it < Node::GetN(); ++it)
{
  if (edgeData[node1][it].GetStatus() && it != node1)
    shortlist.push_back(it);
}
// choose second node for first edge
int32_t node2;
numberNodesAvailable = shortlist.size();
if (numberNodesAvailable == 0)
{
  result.code = ErrorCode::NO_EDGE_AVAILABLE;
  return result;
}
else if (numberNodesAvailable == 1) node2 = shortlist[0];
else node2 = getUniformInteger(shortlist);
  shortlist.clear();

// choose node 3, first node of second edge
for (int32_t it(0); it < Node::GetN(); ++it)
{
  if (nodeList[it].GetNumberOfEdges() < Node::GetN() - 1)
    shortlist.push_back(it);
}
  int32_t node3;
numberNodesAvailable = shortlist.size();
  if (numberNodesAvailable == 0)
{
  result.code = ErrorCode::NO_EDGE_AVAILABLE;
  return result;
}
else if (numberNodesAvailable == 1) node3 = shortlist[0];
else node3 = getUniformInteger(shortlist);
  shortlist.clear();

// choose node 4, second node of second edge
for (int32_t it(0); it < Node::GetN(); ++it)
{
  if (!edgeData[node3][it].GetStatus() && it != node3)
    shortlist.push_back(it);
}

int32_t node4;
numberNodesAvailable = shortlist.size();
  if (numberNodesAvailable == 0)
{
  result.code = ErrorCode::NO_EDGE_AVAILABLE;
  return result;
}
else if (numberNodesAvailable == 1)
{
  if ((node3 == node1 && shortlist[0] == node2) || (node3 == node2 && shortlist[0]
      == node1))
  {
    result.code = ErrorCode::NO_EDGE_AVAILABLE;
    return result;
  }
  else  node4 = shortlist[0];
}
```

```
  else
  {
    do
    {
      node4 = getUniformInteger(shortlist);
    } while ((node3 == node1 && node4 == node2) || (node3 == node2 && node4 == node1)
        || node4 == node3);
  }

  shortlist.clear();

  // give result the chosen node positions and ID
  result.node1 = node1;
  result.node1ID = nodeList[node1].GetID();
  result.node2 = node2;
  result.node2ID = nodeList[node2].GetID();
  result.node3 = node3;
  result.node3ID = nodeList[node3].GetID();
  result.node4 = node4;
  result.node4ID = nodeList[node4].GetID();

  // update system

  edgeData[node1][node2].SetStatus(false);
  edgeData[node2][node1].SetStatus(false);
  nodeList[node1].DecreaseNumberOfEdges();
  nodeList[node2].DecreaseNumberOfEdges();

  edgeData[node3][node4].SetStatus(true);
  edgeData[node4][node3].SetStatus(true);
  nodeList[node3].IncreaseNumberOfEdges();
  nodeList[node4].IncreaseNumberOfEdges();

  result.code = ErrorCode::OK;

  return result;
}

Result infectNode(NodeVector_t &nodeList, EdgeMatrix_t &edgeData)
{
  Result result;

  /* Choose infected and susceptible node */

  // choose infected node:
  // create a list with positions of infected nodes
  for (int32_t it(0); it < Node::GetN(); ++it)
  {
    if (nodeList[it].GetStatus() == NodeStatus::INFECTED)
      shortlist.push_back(it);
  }

  int32_t node1;
  int32_t numberNodesAvailable = shortlist.size();

  if (numberNodesAvailable == 0) // no infected nodes
  {
    result.code = ErrorCode::NO_NODE_AVAILABLE;
```

```
  return result;
}
else if (numberNodesAvailable == 1) node1 = shortlist[0]; // select infected node1
else node1 = getUniformInteger(shortlist); // select a random infected node1

shortlist.clear();

// choose susceptible node:
// if no nodes are connected to infected node1
if (nodeList[node1].GetNumberOfEdges() == 0)
{
  result.code = ErrorCode::NO_EDGE_AVAILABLE;
  return result;
}

// create a list with all node positions connected to infected node1
for (int32_t it(0); it < Node::GetN(); ++it)
{
  if (edgeData[node1][it].GetStatus() && it != node1)
    shortlist.push_back(it);
}

int32_t node2;
// if the rate of infection spread is dependent on all nodes connected to infected
    nodes (ne) all connected nodes to node1 will be used
if (options.GetInfectionRate_infectionType() == "ni_ne_linear" || options.
    GetInfectionRate_infectionType() == "ni_ne_linear_balanced")
  {
    numberNodesAvailable = shortlist.size();
    if (numberNodesAvailable == 1)
    {
      if (nodeList[shortlist[0]].GetStatus() == NodeStatus::SUSCEPTIBLE)
        node2 = shortlist[0];
      else
      {
        result.code = ErrorCode::ALREADY_INFECTED;
        shortlist.clear();
        return result;
      }
    }
    else
    {
      node2 = getUniformInteger(shortlist); // select a random node2

      if (nodeList[node2].GetStatus() != NodeStatus::SUSCEPTIBLE)
      {
        result.code = ErrorCode::ALREADY_INFECTED;
        shortlist.clear();
        return result;
      }
    }
  }

else
{
  // create list with all susceptible nodes positions connected to infected node 1
  for (uint32_t it(0); it < shortlist.size(); ++it)
  {
```

```
      if (nodeList[shortlist[it]].GetStatus() == NodeStatus::SUSCEPTIBLE)
        shortlist2.push_back(shortlist[it]);
    }

    numberNodesAvailable = shortlist2.size();
    if (numberNodesAvailable == 0) // no (susceptible) nodes connected to infected
        node 1
    {
      result.code = ErrorCode::ALREADY_INFECTED;
      shortlist.clear();
      return result;
    }
    else if (numberNodesAvailable == 1) node2 = shortlist2[0]; // select (susceptible)
         node2
    else node2 = getUniformInteger(shortlist2); // select a random (susceptible) node2
  }

  shortlist.clear();
  shortlist2.clear();

  // give chosen node positions and ID to result
  result.node1 = node1;
  result.node1ID = nodeList[node1].GetID();
  result.node2 = node2;
  result.node2ID = nodeList[node2].GetID();


  /* update system */

  nodeList[node2].ChangeStatus(NodeStatus::INFECTED);
  Node::IncreaseNInfected();

  result.code = ErrorCode::OK;
  return result;
}

Result healNode(NodeVector_t &nodeList, EdgeMatrix_t &edgeData)
{
  Result result;

  // select infected node
  for (int32_t it(0); it < Node::GetN(); ++it)
  {
    if (nodeList[it].GetStatus() == NodeStatus::INFECTED)
      shortlist.push_back(it);
  }

  int32_t node1;
  int32_t numberNodesAvailable = shortlist.size();

  if (numberNodesAvailable == 0)
  {
    result.code = ErrorCode::NO_NODE_AVAILABLE;
    return result;
  }
  else if (numberNodesAvailable == 1) node1 = shortlist[0];
  else node1 = getUniformInteger(shortlist);
```

```
  shortlist.clear();

  result.node1 = node1;
  result.node1ID = nodeList[node1].GetID();

  // update system
  nodeList[node1].ChangeStatus(NodeStatus::SUSCEPTIBLE);
  Node::DecreaseNInfected();

  result.code = ErrorCode::OK;

  return result;
}
```

C.2.4 rng.cpp:

```
// event.cpp : adds node to system.
//

#include "stdafx.h"

// headers included with compiler
#include <cstdint>
#include <vector>
#include <random>
#include <iostream>

// headers
#include "datastructs.h"
#include "options.h"

static std::random_device rd;
static std::mt19937 mt(rd());

void setRandomGenerator()
{
  if (options.GetGeneral_fixedSeed() > 0) mt.seed(options.GetGeneral_fixedSeed());
  else mt.seed(rd());
}

//set random generator

double getRandom01()
{
  std::uniform_real_distribution<double> uniformReal(0, 1);
  return uniformReal(mt);
}

double getExponentialRandomNumber(double mean)
{
  std::exponential_distribution<double> exponential(mean);
  return exponential(mt);
}

int32_t getUniformInteger(const int32_t &numberOfValues)
{
```

```
  std::uniform_int_distribution<int32_t> uniformInt(0, numberOfValues - 1);
  return uniformInt(mt);
}

int32_t getUniformInteger(const std::vector<int32_t> &valueList)
{
  std::uniform_int_distribution<int32_t> uniformInt(0, valueList.size() - 1);
  return valueList[uniformInt(mt)];

}

int32_t getWeightedUniformReal(const RateVector_t &rates)
{
  double sum(0);
  for (uint32_t it(0); it < rates.size(); ++it)
  {
    sum += rates[it].value;
  }

  std::uniform_real_distribution<double> uniformReal(0, sum);

  double uniformValue = uniformReal(mt);

  double halfValue = uniformValue / 2.0;

  int32_t chosenValue(0);
  double cumlativeWeight = rates[0].value;

  while (cumlativeWeight < uniformValue)
  {
    ++chosenValue;
    cumlativeWeight += rates[chosenValue].value;
  }

  //std::cout << "sum and choice: " << sum << ", " << chosenValue << "\t";
  return chosenValue;
}
```

C.2.5 rates.cpp:

```
// event.cpp : adds node to system.
//

#include "stdafx.h"

// headers included with compiler
#include <cstdint>
#include <vector>

// headers
#include "datastructs.h"
#include "options.h"
#include "Node.h"

//internal functions
void calculateRatesRatio(double &rates1, double &rates2, const double &ratio)
```

```
{
  double fraction = rates1 / (ratio + 1.0);
  rates1 = fraction * ratio;
  rates2 = fraction;
}

// extern
void initiateRates(RateVector_t &rates)
{
  // clear rates from previous simulation
  rates.clear();

  // add/remove node rates
  if ((options.GetRates_arNode() + options.GetRates_arNode2()) != 0)
  {
    double addNodeRate = options.GetRates_arNode();
    double removeNodeRate = options.GetRates_arNode2();
    double arNodeRatio = options.GetArNodeRate_ratio();

    if (options.GetArNodeRate_dependent()) calculateRatesRatio(addNodeRate,
        removeNodeRate, arNodeRatio);

    // initiate add and remove node rates
    if (addNodeRate > 0)
      rates.push_back({ RateType::ADD_NODE, addNodeRate, addNodeRate });

    if (removeNodeRate > 0)
      rates.push_back({ RateType::REMOVE_NODE, removeNodeRate, removeNodeRate });
  }

  // add/remove edge rates
  if ((options.GetRates_arEdge() + options.GetRates_arEdge2()) != 0)
  {
    double addEdgeRate = options.GetRates_arEdge();
    double removeEdgeRate = options.GetRates_arEdge2();
    double arEdgeRatio = options.GetArEdgeRate_ratio();

    if (options.GetArEdgeRate_changeStatus())
    {
      if (addEdgeRate > 0)
        rates.push_back({ RateType::CHANGE_EDGE, addEdgeRate, addEdgeRate });
    }
    else
    {
      if (options.GetArEdgeRate_dependent())
        calculateRatesRatio(addEdgeRate, removeEdgeRate, arEdgeRatio);

      if (addEdgeRate > 0)
        rates.push_back({ RateType::ADD_EDGE, addEdgeRate, addEdgeRate });

      if (removeEdgeRate > 0)
        rates.push_back({ RateType::REMOVE_EDGE, removeEdgeRate, removeEdgeRate });
    }
  }

  //swap edge rate (to be added)
  if (options.GetRates_swapEdge() != 0)
  {
```

```
    double swapRate = options.GetRates_swapEdge();

    if (options.GetSwapEdgeRate_changeStatus()) rates.push_back({ RateType::
        SWAP_EDGE_STATUS, swapRate, swapRate });
    else rates.push_back({ RateType::SWAP_EDGE, swapRate, swapRate });
  }

  // infection spread and heal rate
  if ((options.GetRates_infection() + options.GetRates_infection2()) != 0)
  {
    double infectionRate = options.GetRates_infection();
    double healRate = options.GetRates_infection2();
    double infectionRatio = options.GetInfectionRate_ratio();

    if (options.GetInfectionRate_dependent()) calculateRatesRatio(infectionRate,
        healRate, infectionRatio);

    if (infectionRate > 0)
      rates.push_back({ RateType::INFECT_NODE, infectionRate, infectionRate });

    if (healRate > 0)
      rates.push_back({ RateType::HEAL_NODE, healRate, healRate });
  }
}

void updateNodeRates(RateVector_t &rates)
{
  // add node rates
  if (options.GetArNodeRate_type() == "n_linear" || options.GetArNodeRate_type() == "
      n_linear_add_only")
  {
    for (uint32_t it = 0; it < rates.size(); ++it)
    {
      if (rates[it].type == RateType::ADD_NODE)
      {
        if (Node::GetN() == 0)
          rates[it].value = rates[it].baseValue;
        else rates[it].value = rates[it].baseValue * Node::GetN();
      }
    }
  }
  else if (options.GetArNodeRate_type() == "n_linear_balanced" || options.
      GetArNodeRate_type() == "n_linear_balanced_add_only")
  {

    for (uint32_t it = 0; it < rates.size(); ++it)
    {
      if (rates[it].type == RateType::ADD_NODE)
      {
        if (options.GetGeneral_nBalanced() >= options.GetGeneral_nMax())
          rates[it].value = rates[it].baseValue;
        else if (options.GetGeneral_nBalanced() <= 0)
          rates[it].value = 0;
        else
        {
          double nSlope = (0.0 - rates[it].baseValue) / (options.GetGeneral_nMax() -
              options.GetGeneral_nBalanced());
          double nIntercept = 0.0 - (nSlope * options.GetGeneral_nMax());
```

```
          rates[it].value = nSlope * Node::GetN() + nIntercept;
        }
      }
    }
  }
  else if (options.GetArNodeRate_type() == "n_linear_remove_only")
  {
    for (uint32_t it = 0; it < rates.size(); ++it)
    {
      if (rates[it].type == RateType::ADD_NODE)
      {
        if (Node::GetN() == 0)
          rates[it].value = rates[it].baseValue;
        else rates[it].value = rates[it].baseValue * options.GetGeneral_nBalanced();
      }
    }
  }

  // remove node rates
  if (options.GetArNodeRate_type() == "n_linear" || options.GetArNodeRate_type() == "
      n_linear_remove_only")
  {
    for (uint32_t it = 0; it < rates.size(); ++it)
    {
      if (rates[it].type == RateType::REMOVE_NODE)
        rates[it].value = rates[it].baseValue * Node::GetN();
    }
  }
  else if (options.GetArNodeRate_type() == "n_linear_balanced" || options.
      GetArNodeRate_type() == "n_linear_balanced_remove_only")
  {

    for (uint32_t it = 0; it < rates.size(); ++it)
    {
      if (rates[it].type == RateType::REMOVE_NODE)
      {
        if (options.GetGeneral_nBalanced() >= options.GetGeneral_nMax())
          rates[it].value = 0;
        else if (options.GetGeneral_nBalanced() <= 0)
          rates[it].value = rates[it].baseValue;
        else
        {
          double nSlope = (rates[it].baseValue - 0.0) / (options.GetGeneral_nBalanced
              () - options.GetGeneral_nMin());
          double nIntercept = 0.0 - (nSlope * options.GetGeneral_nMin());
          rates[it].value = nSlope * Node::GetN() + nIntercept;
        }
      }
    }
  }
  else if (options.GetArNodeRate_type() == "n_linear_add_only")
  {
    for (uint32_t it = 0; it < rates.size(); ++it)
    {
      if (rates[it].type == RateType::REMOVE_NODE)
      {
        if (Node::GetN() == 0)
          rates[it].value = rates[it].baseValue;
```

```
        else rates[it].value = rates[it].baseValue * options.GetGeneral_nBalanced();
      }
    }
  }

  // dependent rates
  if (options.GetArNodeRate_type() == "constant_dependent_n_linear_balanced" &&
      options.GetArNodeRate_dependent())
  {
    for (uint32_t itadd = 0; itadd < rates.size(); ++itadd)
    {
      if (rates[itadd].type == RateType::ADD_NODE)
      {
        for (uint32_t itrem = 0; itrem < rates.size(); ++itrem)
        {
          if (rates[itrem].type == RateType::REMOVE_NODE)
          {
            double baseRate = rates[itadd].baseValue + rates[itrem].baseValue;

            if (Node::GetN() == options.GetGeneral_nBalanced())
            {
              rates[itadd].value = baseRate * 0.5;
              rates[itrem].value = baseRate * 0.5;
            }
            else if (Node::GetN() <= options.GetGeneral_nBalanced())
            {
              double fraction = (static_cast<double>(Node::GetN()) - options.
                  GetGeneral_nMin()) / (options.GetGeneral_nBalanced() - options.
                  GetGeneral_nMin());
              rates[itadd].value = baseRate * (1 - (0.5 * fraction));
              rates[itrem].value = baseRate * 0.5 * fraction;
            }
            else
            {
              double fraction = (Node::GetN() - (2.0 * options.GetGeneral_nBalanced())
                   + options.GetGeneral_nMax())
                / (options.GetGeneral_nMax() - options.GetGeneral_nBalanced());
              rates[itadd].value = baseRate * (1 - (0.5 * fraction));
              rates[itrem].value = baseRate * 0.5 * fraction;
            }
          }
        }
      }
    }
  }
}

void updateEdgeRates(RateVector_t &rates)
{
  double eMax = (Node::GetN() * Node::GetN() - Node::GetN()) * 0.5;

  // if there are no edges in system (n < 2)
  if (eMax <= 0)
  {
    for (uint32_t it = 0; it < rates.size(); ++it)
    {
      if (rates[it].type == RateType::CHANGE_EDGE)
        rates[it].value = 0;
```

```
    if (rates[it].type == RateType::ADD_EDGE)
      rates[it].value = 0;

    if (rates[it].type == RateType::REMOVE_EDGE)
      rates[it].value = 0;

    if (rates[it].type == RateType::SWAP_EDGE_STATUS)
      rates[it].value = 0;

    if (rates[it].type == RateType::SWAP_EDGE)
      rates[it].value = 0;
  }
  return;
}

// change edge status rates
if (options.GetArEdgeRate_type() == "e_linear")
{
  for (uint32_t it = 0; it < rates.size(); ++it)
  {
    if (rates[it].type == RateType::CHANGE_EDGE)
      rates[it].value = rates[it].baseValue * eMax;
  }
}

// add edge rates
if (options.GetArEdgeRate_type() == "e_linear")
{
  for (uint32_t it = 0; it < rates.size(); ++it)
  {
    if (rates[it].type == RateType::ADD_EDGE)
      rates[it].value = rates[it].baseValue * (eMax - Node::GetE());
  }
}
else if (options.GetArEdgeRate_type() == "e_linear_balanced")
{

  for (uint32_t it = 0; it < rates.size(); ++it)
  {
    if (rates[it].type == RateType::ADD_EDGE)
    {
      double eBalanced = eMax * options.GetGeneral_edgeFraction();
      if (eBalanced >= eMax)
        rates[it].value = rates[it].baseValue;
      else if (eBalanced <= 0)
        rates[it].value = 0;
      else
      {
        double nSlope = (0.0 - rates[it].baseValue) / (eMax - eBalanced);
        double nIntercept = 0.0 - (nSlope * eMax);
        rates[it].value = nSlope * Node::GetE() + nIntercept;
      }
    }
  }
}

// remove edge rates
```

```
if (options.GetArEdgeRate_type() == "e_linear")
{
  for (uint32_t it = 0; it < rates.size(); ++it)
  {
    if (rates[it].type == RateType::REMOVE_EDGE)
      rates[it].value = rates[it].baseValue * Node::GetE();
  }
}
else if (options.GetArEdgeRate_type() == "e_linear_balanced")
{

  for (uint32_t it = 0; it < rates.size(); ++it)
  {
    if (rates[it].type == RateType::REMOVE_EDGE)
    {
      double eBalanced = eMax * options.GetGeneral_edgeFraction();
      if (eBalanced >= eMax)
        rates[it].value = 0;
      else if (eBalanced <= 0)
        rates[it].value = rates[it].baseValue;
      else
      {
        double nSlope = rates[it].baseValue / eBalanced;
        double nIntercept = 0.0 - (nSlope * 0);
        rates[it].value = nSlope * Node::GetE() + nIntercept;
      }
    }
  }
}

// dependent rates
if (options.GetArEdgeRate_type() == "constant_dependent_e_linear_balanced" &&
    options.GetArEdgeRate_dependent())
{
  for (uint32_t itadd = 0; itadd < rates.size(); ++itadd)
  {
    if (rates[itadd].type == RateType::ADD_EDGE)
    {
      for (uint32_t itrem = 0; itrem < rates.size(); ++itrem)
      {
        if (rates[itrem].type == RateType::REMOVE_EDGE)
        {
          double baseRate = rates[itadd].baseValue + rates[itrem].baseValue;
          double eBalanced = eMax * options.GetGeneral_edgeFraction();

          if (Node::GetE() <= eBalanced)
          {
            double fraction = Node::GetE() / eBalanced;
            rates[itadd].value = baseRate * (1 - (0.5 * fraction));
            rates[itrem].value = baseRate * 0.5 * fraction;
          }
          else
          {
            double fraction = (Node::GetE() - (2 * eBalanced) + eMax) / (eMax -
                eBalanced);
            rates[itadd].value = baseRate * (1 - (0.5 * fraction));
            rates[itrem].value = baseRate * 0.5 * fraction;
          }
```

```
          }
        }
      }
    }
  }

  // swap edge status rates
  if (options.GetSwapEdgeRate_type() == "e_linear")
  {
    for (uint32_t it = 0; it < rates.size(); ++it)
    {
      if (rates[it].type == RateType::SWAP_EDGE_STATUS)
        rates[it].value = rates[it].baseValue * eMax;
    }
  }

  // swap edge rates
  if (options.GetSwapEdgeRate_type() == "e_linear")
  {
    for (uint32_t it = 0; it < rates.size(); ++it)
    {
      if (rates[it].type == RateType::SWAP_EDGE)
        rates[it].value = rates[it].baseValue * Node::GetE();
    }
  }
  else if (options.GetSwapEdgeRate_type() == "e_linear_balanced")
  {

    for (uint32_t it = 0; it < rates.size(); ++it)
    {
      if (rates[it].type == RateType::SWAP_EDGE)
        rates[it].value = rates[it].baseValue * (Node::GetE() / eMax);
    }
  }
}

void updateVirusRates(RateVector_t &rates, const NodeVector_t &nodeList)
{
  // infection rates
  if (options.GetInfectionRate_infectionType() == "ni_linear")
  {
    for (uint32_t it = 0; it < rates.size(); ++it)
    {
      if (rates[it].type == RateType::INFECT_NODE)
        rates[it].value = rates[it].baseValue * Node::GetNInfected();
    }
  }
  else if (options.GetInfectionRate_infectionType() == "ni_linear_balanced")
  {
    for (uint32_t it = 0; it < rates.size(); ++it)
    {
      if (rates[it].type == RateType::INFECT_NODE)
        rates[it].value = (rates[it].baseValue /
          (options.GetInfectionRate_infectionFactor() * options.GetGeneral_nBalanced()
              )) * Node::GetNInfected();
    }
  }
```

```
  else if (options.GetInfectionRate_infectionType() == "ni_ne_linear")
  {
    for (uint32_t it = 0; it < rates.size(); ++it)
    {
      if (rates[it].type == RateType::INFECT_NODE)
      {
        int32_t connectedNodes = 0;
        for (int32_t it2 = 0; it2 < Node::GetN(); ++it2)
        {
          if (nodeList[it2].GetStatus() == NodeStatus::INFECTED)
            connectedNodes += nodeList[it2].GetNumberOfEdges();
        }
        rates[it].value = rates[it].baseValue * connectedNodes;
      }
    }
  }
  else if (options.GetInfectionRate_infectionType() == "ni_ne_linear_balanced")
  {
    for (uint32_t it = 0; it < rates.size(); ++it)
    {
      if (rates[it].type == RateType::INFECT_NODE)
      {
        int32_t connectedNodes = 0;
        for (int32_t it2 = 0; it2 < Node::GetN(); ++it2)
        {
          if (nodeList[it2].GetStatus() == NodeStatus::INFECTED)
            connectedNodes += nodeList[it2].GetNumberOfEdges();
        }
        rates[it].value = (rates[it].baseValue /
          (options.GetInfectionRate_infectionFactor() * options.GetGeneral_nBalanced()
              )) * connectedNodes;
      }
    }
  }


  // heal rates
  if (options.GetInfectionRate_infectionType() == "ni_linear" || options.
      GetInfectionRate_infectionType() == "ni_ne_linear")
  {
    for (uint32_t it = 0; it < rates.size(); ++it)
    {
      if (rates[it].type == RateType::HEAL_NODE)
        rates[it].value = rates[it].baseValue * Node::GetNInfected();
    }
  }
  else if (options.GetInfectionRate_infectionType() == "ni_linear_balanced" || options
      .GetInfectionRate_infectionType() == "ni_ne_linear_balanced")
  {
    for (uint32_t it = 0; it < rates.size(); ++it)
    {
      if (rates[it].type == RateType::HEAL_NODE)
        rates[it].value = (rates[it].baseValue /
          (options.GetInfectionRate_infectionFactor() * options.GetGeneral_nBalanced()
              )) * Node::GetNInfected();
    }
  }
}
```

C.2.6 input.cpp:

```
// simulation.cpp : runs 1 complete simulation.
//

#include "stdafx.h"

// headers included with compiler
#include <cstdint>
#include <vector>
#include <iostream>
#include <fstream>
#include <string>
#include <numeric>

// headers
#include "options.h"

std::vector<std::string> stringSplit(std::string str, std::string delimiters)
{
  std::vector<std::string> result;
  size_t current;
  size_t next = -1;
  do
  {
    current = next + 1;

    next = str.find_first_of(delimiters, current);
    if (str.substr(current, next - current) != "␣")
    {
      result.push_back(str.substr(current, next - current));
    }

    if (result.size() >= 2) break;
  } while (next != std::string::npos);

  return result;
}

bool checkDoubleValue(std::string &currentvalue, std::string &strvalue)
{
  strvalue = strvalue.substr(0, strvalue.length() - 1 ); //remove last number due to
      possible rounding error

  if (currentvalue.length() >= strvalue.length())
  {
    std::string strcomp = currentvalue.substr(0, strvalue.length());
    return (strcomp == strvalue);
  }
  else
  {
    std::string strcomp = strvalue.substr(0, currentvalue.length());
    return (strcomp == currentvalue);
  }
}

bool checkIntegerValue(std::string &currentvalue, std::string &strvalue)
{
```

```
  //std::cout << currentvalue << " equals " << strvalue << "\n";
  if (currentvalue.length() > strvalue.length())
  {
    if (currentvalue.at(strvalue.length()) == '.')
    {
      std::string strcomp = currentvalue.substr(0, strvalue.length());
      return (strcomp == strvalue);
    }
    else return(strvalue == currentvalue);
  }
  else return (strvalue == currentvalue);
}

void general(std::string &strInput)
{
  std::vector<std::string> currentoption = stringSplit(strInput, "␣\t;");

  if (currentoption.size() > 1)
  {
    if (currentoption[0] == "numberOfSimulations")
    {
      int32_t value = atoi(currentoption[1].c_str());
      std::string strvalue = std::to_string(value);

      if (checkIntegerValue(currentoption[1], strvalue))
      {
        if (value < 1)
        {
          options.SetGeneral_numberOfSimulations(0);
          std::cout << "general:␣\"" << currentoption[0] << "␣" << currentoption[1] <<
               "\"␣is␣to␣low,␣";
          std::cout << currentoption[0] << "␣is␣set␣to␣1\n";
        }
        else options.SetGeneral_numberOfSimulations(value);
      }
      else
      {
        std::cout << "general:␣\"" << currentoption[0] << "␣" << currentoption[1] << "
            \"␣is␣not␣valid␣input,␣";
        std::cout << currentoption[0] << "␣is␣set␣to␣default\n";
      }
    }

    else if (currentoption[0] == "nBalanced")
    {
      int32_t value = atoi(currentoption[1].c_str());
      std::string strvalue = std::to_string(value);

      if (checkIntegerValue(currentoption[1], strvalue))
      {
        options.SetGeneral_nBalanced(value);
      }
      else
      {
        std::cout << "general:␣\"" << currentoption[0] << "␣" << currentoption[1] << "
            \"␣is␣not␣valid␣input,␣";
        std::cout << currentoption[0] << "␣is␣set␣to␣default\n";
      }
```

```
    }

    else if (currentoption[0] == "nMax")
    {
      int32_t value = atoi(currentoption[1].c_str());
      std::string strvalue = std::to_string(value);

      if (checkIntegerValue(currentoption[1], strvalue))
      {
        if (options.GetGeneral_nBalanced() > value)
        {
          options.SetGeneral_nMax(2 * options.GetGeneral_nBalanced());
          std::cout << "general:␣n␣max␣was␣to␣low,␣n␣max␣is␣set␣to␣2␣times␣n␣balanced\
              n";
        }

        else options.SetGeneral_nMax(value);
      }
      else
      {
        std::cout << "general:␣\"" << currentoption[0] << "␣" << currentoption[1] << "
            \"␣is␣not␣valid␣input,␣";
        std::cout << currentoption[0] << "␣is␣set␣to␣default\n";
        if (options.GetGeneral_nBalanced() > options.GetGeneral_nMax())
        {
          options.SetGeneral_nMax(2 * options.GetGeneral_nBalanced());
          std::cout << "general:␣default␣n␣max␣was␣to␣low,␣n␣max␣is␣set␣to␣2␣times␣n␣
              balanced\n";
        }
      }
    }

    else if (currentoption[0] == "nMin")
    {
      int32_t value = atoi(currentoption[1].c_str());
      std::string strvalue = std::to_string(value);

      if (checkIntegerValue(currentoption[1], strvalue))
      {
        if (options.GetGeneral_nBalanced() < value)
        {
          options.SetGeneral_nMin(0);
          std::cout << "general:␣n␣min␣was␣to␣high,␣n␣min␣is␣set␣to␣0\n";
        }
        else options.SetGeneral_nMin(value);
      }
      else
      {
        std::cout << "general:␣\"" << currentoption[0] << "␣" << currentoption[1] << "
            \"␣is␣not␣valid␣input,␣";
        std::cout << currentoption[0] << "␣is␣set␣to␣default\n";
        if (options.GetGeneral_nBalanced() < options.GetGeneral_nMin())
        {
          options.SetGeneral_nMax(0);
          std::cout << "general:␣default␣n␣min␣was␣to␣high,␣n␣min␣is␣set␣to␣0\n";
        }
      }
    }
```

```
  else if (currentoption[0] == "edgeFraction")
  {
    double value = atof(currentoption[1].c_str());
    std::string strvalue = std::to_string(value);

    if (checkDoubleValue(currentoption[1], strvalue))
    {
      if (value <= 0)
      {
        options.SetGeneral_edgeFraction(0.0001);
        std::cout << "general: edge fraction was to low, edge fraction is set to
            0.0001\n";
      }
      else if (value > 1)
      {
        options.SetGeneral_edgeFraction(1.0);
        std::cout << "general: edge fraction was to high, edge fraction is set to 1\
            n";
      }
      else options.SetGeneral_edgeFraction(value);
    }
    else
    {
      std::cout << "general: \"" << currentoption[0] << " " << currentoption[1] << "
          \" is not valid input, ";
      std::cout << currentoption[0] << " is set to default\n";
    }
  }

  else if (currentoption[0] == "fixedSeed")
  {
    int32_t value = atoi(currentoption[1].c_str());
    std::string strvalue = std::to_string(value);

    if (checkIntegerValue(currentoption[1], strvalue))
    {
      if (value < 0)
      {
        options.SetGeneral_fixedSeed(0);
        std::cout << "general: fixed seed was to low, fixed seed is set to 0\n";
      }
      else options.SetGeneral_fixedSeed(value);
    }
    else
    {
      std::cout << "general: \"" << currentoption[0] << " " << currentoption[1] << "
          \" is not valid input, ";
      std::cout << currentoption[0] << " is set to default\n";
    }
  }
}
else
{
  std::cout << "general: \"" << currentoption[0] << " " << currentoption[1] << "\"
      is not valid input, ";
  std::cout << currentoption[0] << " is set to default\n";
}
```

```
}

void startOfSimulation(std::string &strInput)
{
  std::vector<std::string> currentoption = stringSplit(strInput, "␣\t;");

  if (currentoption.size() > 1)
  {
    if (currentoption[0] == "n")
    {
      int32_t value = atoi(currentoption[1].c_str());
      std::string strvalue = std::to_string(value);

      if (checkIntegerValue(currentoption[1],strvalue))
      {
        if (value <= 0)
        {
          options.SetStartOfSimulation_n(1);
          std::cout << "start␣of␣simulation:␣\"" << currentoption[0] << "␣" <<
              currentoption[1] << "\"␣is␣to␣low,␣";
          std::cout << currentoption[0] << "␣is␣set␣to␣1\n";
        }
        else if (options.GetGeneral_nMin() > value)
        {
          options.SetStartOfSimulation_n(options.GetGeneral_nMin());
          std::cout << "start␣of␣simulation:␣n␣was␣to␣low,␣n␣is␣set␣to␣n␣min\n";
        }
        else if (options.GetGeneral_nMax() < value)
        {
          options.SetStartOfSimulation_n(options.GetGeneral_nMax());
          std::cout << "start␣of␣simulation:␣n␣was␣to␣high,␣n␣is␣set␣to␣n␣max\n";
        }
        else options.SetStartOfSimulation_n(value);
      }
      else
      {
        std::cout << "start␣of␣simulation:␣\"" << currentoption[0] << "␣" <<
            currentoption[1] << "\"␣is␣not␣valid␣input,␣";
        std::cout << currentoption[0] << "␣is␣set␣to␣default\n";
        if (options.GetGeneral_nMin() > options.GetStartOfSimulation_n())
        {
          options.SetStartOfSimulation_n(options.GetGeneral_nMin());
          std::cout << "start␣of␣simulation:␣default␣n␣was␣to␣low,␣n␣is␣set␣to␣n␣min\n
              ";
        }
        else if (options.GetGeneral_nMax() < options.GetStartOfSimulation_n())
        {
          options.SetStartOfSimulation_n(options.GetGeneral_nMax());
          std::cout << "start␣of␣simulation:␣default␣n␣was␣to␣high,␣n␣is␣set␣to␣n␣max\
              n";
        }
      }
    }

    else if (currentoption[0] == "activeEdges")
    {
      if (currentoption[1] == "none") options.SetStartOfSimulation_activeEdges("none")
          ;
```

```
      else if (currentoption[1] == "balanced") options.
          SetStartOfSimulation_activeEdges("balanced");
      else if (currentoption[1] == "full") options.SetStartOfSimulation_activeEdges("
          full");
      else
      {
        std::cout << "infection␣rate:␣\"" << currentoption[0] << "␣" << currentoption
            [1] << "\"␣is␣not␣valid␣input,␣";
        std::cout << currentoption[0] << "␣is␣set␣to␣default\n";
      }
    }
  }
  else
  {
    std::cout << "start␣of␣simulation:␣\"" << currentoption[0] << "␣" << currentoption
        [1] << "\"␣is␣not␣valid␣input,␣";
    std::cout << currentoption[0] << "␣is␣set␣to␣default\n";
  }
}

void stoppingCriteria(std::string &strInput)
{
  std::vector<std::string> currentoption = stringSplit(strInput, "␣\t;");

  if (currentoption.size() > 1)
  {
    if (currentoption[0] == "time")
    {
      double value = atof(currentoption[1].c_str());
      std::string strvalue = std::to_string(value);

      if (checkDoubleValue(currentoption[1],strvalue))
      {
        if (value < 0)
        {
          std::cout << "stopping␣criteria:␣\"" << currentoption[0] << "␣" <<
              currentoption[1] << "\"␣is␣negative,␣";
          std::cout << currentoption[0] << "␣is␣set␣to␣default\n";
        }
        else if (value == 0)
        {
          options.SetStoppingCriteria_time(value);
          std::cout << "WARNING:␣stopping␣criteria:␣\"" << currentoption[0] << "␣" <<
              currentoption[1] << "\"␣is␣zero,␣";
          std::cout << "simulation␣will␣not␣have␣any␣events␣if␣time␣limit␣is␣on\n";
        }
        else options.SetStoppingCriteria_time(value);
      }
      else
      {
        std::cout << "stopping␣criteria:␣\"" << currentoption[0] << "␣" <<
            currentoption[1] << "\"␣is␣not␣valid␣input,␣";
        std::cout << currentoption[0] << "␣is␣set␣to␣default\n";
      }
    }
    else if (currentoption[0] == "events")
    {
      int32_t value = atoi(currentoption[1].c_str());
```

```
      std::string strvalue = std::to_string(value);

      if (checkIntegerValue(currentoption[1], strvalue))
      {
        if (value < 0)
        {
          std::cout << "stopping␣criteria:␣\"" << currentoption[0] << "␣" <<
              currentoption[1] << "\"␣is␣negative,␣";
          std::cout << currentoption[0] << "␣is␣set␣to␣default\n";
        }
        else if (value == 0)
        {
          options.SetStoppingCriteria_events(value);
          std::cout << "WARNING:␣stopping␣criteria:␣\"" << currentoption[0] << "␣" <<
              currentoption[1] << "\"␣is␣zero,␣";
          std::cout << "simulation␣will␣not␣have␣any␣events\n";
        }
        else options.SetStoppingCriteria_events(value);
      }
      else
      {
        std::cout << "stopping␣criteria:␣\"" << currentoption[0] << "␣" <<
            currentoption[1] << "\"␣is␣not␣valid␣input,␣";
        std::cout << currentoption[0] << "␣is␣set␣to␣default\n";
      }
    }

    else if (currentoption[0] == "virusExtinction")
    {
      if (currentoption[1] == "true") options.SetStoppingCriteria_virusExtinction(true
          );
      else options.SetStoppingCriteria_virusExtinction(false);
    }
    else if (currentoption[0] == "timeOn")
    {
      if (currentoption[1] == "true") options.SetStoppingCriteria_timeOn(true);
      else options.SetStoppingCriteria_timeOn(false);
    }
  }
  else
  {
    std::cout << "stopping␣criteria:␣\"" << currentoption[0] << "␣" << currentoption
        [1] << "\"␣is␣not␣valid␣input,␣";
    std::cout << currentoption[0] << "␣is␣set␣to␣default\n";
  }
}

void fileOutput(std::string &strInput)
{
  std::vector<std::string> currentoption = stringSplit(strInput, "␣\t;");

  if (currentoption.size() > 1)
  {
    if (currentoption[0] == "simulationData")
    {
      if (currentoption[1] == "true") options.SetFileOutput_simulationData(true);
      else options.SetFileOutput_simulationData(false);
    }
```

```
    else if (currentoption[0] == "eventlog")
    {
      if (currentoption[1] == "true") options.SetFileOutput_eventlog(true);
      else options.SetFileOutput_eventlog(false);
    }

    else if (currentoption[0] == "nodeInfo")
    {
      if (currentoption[1] == "true") options.SetFileOutput_nodeInfo(true);
      else options.SetFileOutput_nodeInfo(false);
    }

    else if (currentoption[0] == "edgeInfo")
    {
      if (currentoption[1] == "true") options.SetFileOutput_edgeInfo(true);
      else options.SetFileOutput_edgeInfo(false);
    }

    else if (currentoption[0] == "ratesInfo")
    {
      if (currentoption[1] == "true") options.SetFileOutput_ratesInfo(true);
      else options.SetFileOutput_ratesInfo(false);
    }
    else if (currentoption[0] == "virusSpread")
    {
      if (currentoption[1] == "true") options.SetFileOutput_virusSpread(true);
      else options.SetFileOutput_virusSpread(false);
    }
  }
  else
  {
    std::cout << "file␣output:␣\"" << currentoption[0] << "␣" << currentoption[1] << "
        \"␣is␣not␣valid␣input,␣";
    std::cout << currentoption[0] << "␣is␣set␣to␣default\n";
  }
}

void consoleOutput(std::string &strInput)
{
  std::vector<std::string> currentoption = stringSplit(strInput, "␣\t;");

  if (currentoption.size() > 1)
  {
    if (currentoption[0] == "simulationData")
    {
      if (currentoption[1] == "true") options.SetConsoleOutput_simulationData(true);
      else options.SetConsoleOutput_simulationData(false);
    }

    else if (currentoption[0] == "eventlog")
    {
      if (currentoption[1] == "true") options.SetConsoleOutput_eventlog(true);
      else options.SetConsoleOutput_eventlog(false);
    }

    else if (currentoption[0] == "simulationPause")
    {
```

```
      if (currentoption[1] == "true") options.SetConsoleOutput_simulationPause(true);
      else options.SetConsoleOutput_simulationPause(false);
    }
  }
  else
  {
    std::cout << "console output: \"" << currentoption[0] << " " << currentoption[1]
        << "\" is not valid input, ";
    std::cout << currentoption[0] << " is set to default\n";
  }
}

void rates(std::string &strInput)
{
  std::vector<std::string> currentoption = stringSplit(strInput, " \t;");

  if (currentoption.size() > 1)
  {
    if (currentoption[0] == "arNode")
    {
      double value = atof(currentoption[1].c_str());
      std::string strvalue = std::to_string(value);

      if (checkDoubleValue(currentoption[1], strvalue))
      {
        if (value < 0)
        {
          std::cout << "rates: \"" << currentoption[0] << " " << currentoption[1] << "
              \" is negative, ";
          std::cout << currentoption[0] << " is set to 0\n";
          options.SetRates_arNode(0);
        }
        else options.SetRates_arNode(value);
      }
      else
      {
        std::cout << "rates: \"" << currentoption[0] << " " << currentoption[1] << "\"
            is not valid input, ";
        std::cout << currentoption[0] << " is set to default\n";
      }
    }

    if (currentoption[0] == "arNode2")
    {
      double value = atof(currentoption[1].c_str());
      std::string strvalue = std::to_string(value);

      if (checkDoubleValue(currentoption[1], strvalue))
      {
        if (value < 0)
        {
          std::cout << "rates: \"" << currentoption[0] << " " << currentoption[1] << "
              \" is negative, ";
          std::cout << currentoption[0] << " is set to 0\n";
          options.SetRates_arNode2(0);
        }
        else options.SetRates_arNode2(value);
      }
```

```
    else
    {
      std::cout << "rates: \"" << currentoption[0] << " " << currentoption[1] << "\"
          is not valid input, ";
      std::cout << currentoption[0] << " is set to default\n";
    }
  }

  else if (currentoption[0] == "arEdge")
  {
    double value = atof(currentoption[1].c_str());
    std::string strvalue = std::to_string(value);

    if (checkDoubleValue(currentoption[1], strvalue))
    {
      if (value < 0)
      {
        std::cout << "rates: \"" << currentoption[0] << " " << currentoption[1] << "
            \" is negative, ";
        std::cout << currentoption[0] << " is set to 0\n";
        options.SetRates_arEdge(0);
      }
      else options.SetRates_arEdge(value);
    }
    else
    {
      std::cout << "rates: \"" << currentoption[0] << " " << currentoption[1] << "\"
          is not valid input, ";
      std::cout << currentoption[0] << " is set to default\n";
    }
  }

  else if (currentoption[0] == "arEdge2")
  {
    double value = atof(currentoption[1].c_str());
    std::string strvalue = std::to_string(value);

    if (checkDoubleValue(currentoption[1], strvalue))
    {
      if (value < 0)
      {
        std::cout << "rates: \"" << currentoption[0] << " " << currentoption[1] << "
            \" is negative, ";
        std::cout << currentoption[0] << " is set to 0\n";
        options.SetRates_arEdge2(0);
      }
      else options.SetRates_arEdge2(value);
    }
    else
    {
      std::cout << "rates: \"" << currentoption[0] << " " << currentoption[1] << "\"
          is not valid input, ";
      std::cout << currentoption[0] << " is set to default\n";
    }
  }

  else if (currentoption[0] == "swapEdge")
  {
```

```
    double value = atof(currentoption[1].c_str());
    std::string strvalue = std::to_string(value);

    if (checkDoubleValue(currentoption[1], strvalue))
    {
      if (value < 0)
      {
        std::cout << "rates: \"" << currentoption[0] << " " << currentoption[1] << "
            \" is negative, ";
        std::cout << currentoption[0] << " is set to 0\n";
        options.SetRates_swapEdge(0);
      }
      else options.SetRates_swapEdge(value);
    }
    else
    {
      std::cout << "rates: \"" << currentoption[0] << " " << currentoption[1] << "\"
          is not valid input, ";
      std::cout << currentoption[0] << " is set to default\n";
    }
  }

  else if (currentoption[0] == "infection")
  {
    double value = atof(currentoption[1].c_str());
    std::string strvalue = std::to_string(value);

    if (checkDoubleValue(currentoption[1], strvalue))
    {
      if (value < 0)
      {
        std::cout << "rates: \"" << currentoption[0] << " " << currentoption[1] << "
            \" is negative, ";
        std::cout << currentoption[0] << " is set to 0\n";
        options.SetRates_infection(0);
      }
      else options.SetRates_infection(value);
    }
    else
    {
      std::cout << "rates: \"" << currentoption[0] << " " << currentoption[1] << "\"
          is not valid input, ";
      std::cout << currentoption[0] << " is set to default\n";
    }
  }

  else if (currentoption[0] == "infection2")
  {
    double value = atof(currentoption[1].c_str());
    std::string strvalue = std::to_string(value);

    if (checkDoubleValue(currentoption[1], strvalue))
    {
      if (value < 0)
      {
        std::cout << "rates: \"" << currentoption[0] << " " << currentoption[1] << "
            \" is negative, ";
        std::cout << currentoption[0] << " is set to 0\n";
```

```
          options.SetRates_infection2(0);
        }
        else options.SetRates_infection2(value);
      }
      else
      {
        std::cout << "rates: \"" << currentoption[0] << " " << currentoption[1] << "\"
            is not valid input, ";
        std::cout << currentoption[0] << " is set to default\n";
      }
    }
  }
  else
  {
    std::cout << "rates: \"" << currentoption[0] << " " << currentoption[1] << "\" is
        not valid input, ";
    std::cout << currentoption[0] << " is set to default\n";
  }
}

void arNodeRate(std::string &strInput)
{
  std::vector<std::string> currentoption = stringSplit(strInput, " \t;");

  if (currentoption.size() > 1)
  {
    if (currentoption[0] == "ratio")
    {
      double value = atof(currentoption[1].c_str());
      std::string strvalue = std::to_string(value);

      if (checkDoubleValue(currentoption[1], strvalue))
      {
        if (value <= 0)
        {
          std::cout << "add/remove node rate: \"" << currentoption[0] << " " <<
              currentoption[1] << "\" is zero or negative, ";
          std::cout << currentoption[0] << " is set to default\n";
        }
        else options.SetArNodeRate_ratio(value);
      }
      else
      {
        std::cout << "add/remove node rate: \"" << currentoption[0] << " " <<
            currentoption[1] << "\" is not valid input, ";
        std::cout << currentoption[0] << " is set to default\n";
      }
    }

    else if (currentoption[0] == "dependent")
    {
      if (currentoption[1] == "true") options.SetArNodeRate_dependent(true);
      else options.SetArNodeRate_dependent(false);
    }

    else if (currentoption[0] == "type")
    {
      if (currentoption[1] == "constant") options.SetArNodeRate_type("constant");
```

```
      else if (currentoption[1] == "n_linear") options.SetArNodeRate_type("n_linear");
      else if (currentoption[1] == "n_linear_add_only") options.SetArNodeRate_type("
          n_linear_add_only");
      else if (currentoption[1] == "n_linear_remove_only") options.SetArNodeRate_type(
          "n_linear_remove_only");
      else if (currentoption[1] == "n_linear_balanced") options.SetArNodeRate_type("
          n_linear_balanced");
      else if (currentoption[1] == "n_linear_balanced_add_only") options.
          SetArNodeRate_type("n_linear_balanced_add_only");
      else if (currentoption[1] == "n_linear_balanced_remove_only") options.
          SetArNodeRate_type("n_linear_balanced_remove_only");
      else if (currentoption[1] == "constant_dependent_n_linear_balanced") options.
          SetArNodeRate_type("constant_dependent_n_linear_balanced");
      else
      {
        std::cout << "add/remove␣node␣rate:␣\"" << currentoption[0] << "␣" <<
            currentoption[1] << "\"␣is␣not␣valid␣input,␣";
        std::cout << currentoption[0] << "␣is␣set␣to␣default\n";
      }
    }

    else if (currentoption[0] == "addFactor")
    {
      double value = atof(currentoption[1].c_str());
      std::string strvalue = std::to_string(value);

      if (checkDoubleValue(currentoption[1], strvalue))
      {
        options.SetArNodeRate_addFactor(value);
      }
      else
      {
        std::cout << "add/remove␣node␣rate:␣\"" << currentoption[0] << "␣" <<
            currentoption[1] << "\"␣is␣not␣valid␣input,␣";
        std::cout << currentoption[0] << "␣is␣set␣to␣default\n";
      }
    }

    else if (currentoption[0] == "removeFactor")
    {
      double value = atof(currentoption[1].c_str());
      std::string strvalue = std::to_string(value);

      if (checkDoubleValue(currentoption[1], strvalue))
      {
        options.SetArNodeRate_removeFactor(value);
      }
      else
      {
        std::cout << "add/remove␣node␣rate:␣\"" << currentoption[0] << "␣" <<
            currentoption[1] << "\"␣is␣not␣valid␣input,␣";
        std::cout << currentoption[0] << "␣is␣set␣to␣default\n";
      }
    }
  }
  else
  {
```

```
    std::cout << "add/remove␣node␣rate:␣\"" << currentoption[0] << "␣" <<
        currentoption[1] << "\"␣is␣not␣valid␣input,␣";
    std::cout << currentoption[0] << "␣is␣set␣to␣default\n";
  }
}

void arEdgeRate(std::string &strInput)
{
  std::vector<std::string> currentoption = stringSplit(strInput, "␣\t;");

  if (currentoption.size() > 1)
  {
    if (currentoption[0] == "changeStatus")
    {
      if (currentoption[1] == "true") options.SetArEdgeRate_changeStatus(true);
      else options.SetArEdgeRate_changeStatus(false);
    }
    else if (currentoption[0] == "ratio")
    {
      double value = atof(currentoption[1].c_str());
      std::string strvalue = std::to_string(value);

      if (checkDoubleValue(currentoption[1], strvalue))
      {
        if (value <= 0)
        {
          std::cout << "add/remove␣edge␣rate:␣\"" << currentoption[0] << "␣" <<
              currentoption[1] << "\"␣is␣zero␣or␣negative,␣";
          std::cout << currentoption[0] << "␣is␣set␣to␣default\n";
        }
        else options.SetArEdgeRate_ratio(value);
      }
      else
      {
        std::cout << "add/remove␣edge␣rate:␣\"" << currentoption[0] << "␣" <<
            currentoption[1] << "\"␣is␣not␣valid␣input,␣";
        std::cout << currentoption[0] << "␣is␣set␣to␣default\n";
      }
    }

    else if (currentoption[0] == "dependent")
    {
      if (currentoption[1] == "true") options.SetArEdgeRate_dependent(true);
      else options.SetArEdgeRate_dependent(false);
    }

    else if (currentoption[0] == "type")
    {
      if (currentoption[1] == "constant") options.SetArEdgeRate_type("constant");
      else if (currentoption[1] == "e_linear") options.SetArEdgeRate_type("e_linear");
      else if (currentoption[1] == "e_linear_balanced") options.SetArEdgeRate_type("
          e_linear_balanced");
      else if (currentoption[1] == "constant_dependent_e_linear_balanced") options.
          SetArEdgeRate_type("constant_dependent_e_linear_balanced");
      else
      {
        std::cout << "add/remove␣edge␣rate:␣\"" << currentoption[0] << "␣" <<
            currentoption[1] << "\"␣is␣not␣valid␣input,␣";
```

```
        std::cout << currentoption[0] << " is set to default\n";
      }
    }

    else if (currentoption[0] == "addFactor")
    {
      double value = atof(currentoption[1].c_str());
      std::string strvalue = std::to_string(value);

      if (checkDoubleValue(currentoption[1], strvalue))
      {
        options.SetArEdgeRate_addFactor(value);
      }
      else
      {
        std::cout << "add/remove edge rate: \"" << currentoption[0] << " " <<
            currentoption[1] << "\" is not valid input, ";
        std::cout << currentoption[0] << " is set to default\n";
      }
    }

    else if (currentoption[0] == "removeFactor")
    {
      double value = atof(currentoption[1].c_str());
      std::string strvalue = std::to_string(value);

      if (checkDoubleValue(currentoption[1], strvalue))
      {
        options.SetArEdgeRate_removeFactor(value);
      }
      else
      {
        std::cout << "add/remove edge rate: \"" << currentoption[0] << " " <<
            currentoption[1] << "\" is not valid input, ";
        std::cout << currentoption[0] << " is set to default\n";
      }
    }
  }
  else
  {
    std::cout << "swap edge rate: \"" << currentoption[0] << " " << currentoption[1]
        << "\" is not valid input, ";
    std::cout << currentoption[0] << " is set to default\n";
  }
}

void swapEdgeRate(std::string &strInput)
{
  std::vector<std::string> currentoption = stringSplit(strInput, "  \t;");

  if (currentoption.size() > 1)
  {
    if (currentoption[0] == "changeStatus")
    {
      if (currentoption[1] == "true") options.SetSwapEdgeRate_changeStatus(true);
      else options.SetSwapEdgeRate_changeStatus(false);
    }
```

```
    else if (currentoption[0] == "type")
    {
      if (currentoption[1] == "constant") options.SetSwapEdgeRate_type("constant");
      else if (currentoption[1] == "e_linear") options.SetSwapEdgeRate_type("e_linear"
          );
      else if (currentoption[1] == "e_linear_balanced") options.SetSwapEdgeRate_type("
          e_linear_balanced");
      else
      {
        std::cout << "swap edge rate: \"" << currentoption[0] << " " << currentoption
            [1] << "\" is not valid input, ";
        std::cout << currentoption[0] << " is set to default\n";
      }
    }

    else if (currentoption[0] == "swapFactor")
    {
      double value = atof(currentoption[1].c_str());
      std::string strvalue = std::to_string(value);

      if (checkDoubleValue(currentoption[1], strvalue))
      {
        options.SetSwapEdgeRate_swapFactor(value);
      }
      else
      {
        std::cout << "swap edge rate: \"" << currentoption[0] << " " << currentoption
            [1] << "\" is not valid input, ";
        std::cout << currentoption[0] << " is set to default\n";
      }
    }
  }
  else
  {
    std::cout << "swap edge rate: \"" << currentoption[0] << " " << currentoption[1]
        << "\" is not valid input, ";
    std::cout << currentoption[0] << " is set to default\n";
  }
}

void infectionRate(std::string &strInput)
{
  std::vector<std::string> currentoption = stringSplit(strInput, " \t;");

  if (currentoption.size() > 1)
  {
    if (currentoption[0] == "ratio")
    {
      double value = atof(currentoption[1].c_str());
      std::string strvalue = std::to_string(value);

      if (checkDoubleValue(currentoption[1], strvalue))
      {
        if (value <= 0)
        {
          std::cout << "infection rate: \"" << currentoption[0] << " " <<
              currentoption[1] << "\" is zero or negative, ";
          std::cout << currentoption[0] << " is set to default\n";
```

```
      }
      else options.SetInfectionRate_ratio(value);
    }
    else
    {
      std::cout << "infection␣rate:␣\"" << currentoption[0] << "␣" << currentoption
          [1] << "\"␣is␣not␣valid␣input,␣";
      std::cout << currentoption[0] << "␣is␣set␣to␣default\n";
    }
  }

  else if (currentoption[0] == "dependent")
  {
    if (currentoption[1] == "true") options.SetInfectionRate_dependent(true);
    else options.SetInfectionRate_dependent(false);
  }

  else if (currentoption[0] == "infectionType")
  {
    if (currentoption[1] == "constant") options.SetInfectionRate_infectionType("
        constant");
    else if (currentoption[1] == "ni_linear") options.SetInfectionRate_infectionType
        ("ni_linear");
    else if (currentoption[1] == "ni_linear_balanced") options.
        SetInfectionRate_infectionType("ni_linear_balanced");
    else if (currentoption[1] == "ni_ne_linear") options.
        SetInfectionRate_infectionType("ni_ne_linear");
    else if (currentoption[1] == "ni_ne_linear_balanced") options.
        SetInfectionRate_infectionType("ni_ne_linear_balanced");
    else
    {
      std::cout << "infection␣rate:␣\"" << currentoption[0] << "␣" << currentoption
          [1] << "\"␣is␣not␣valid␣input,␣";
      std::cout << currentoption[0] << "␣is␣set␣to␣default\n";
    }
  }

  else if (currentoption[0] == "infectionFactor")
  {
    double value = atof(currentoption[1].c_str());
    std::string strvalue = std::to_string(value);

    if (checkDoubleValue(currentoption[1], strvalue))
    {
      if (value <= 0)
      {
        std::cout << "infection␣rate:␣infection␣factor␣was␣to␣low,␣edge␣fraction␣is␣
            set␣to␣default\n";
      }
      else if (value > 1)
      {
        options.SetInfectionRate_infectionFactor(1.0);
        std::cout << "infection␣rate:␣infection␣factor␣was␣to␣high,␣edge␣fraction␣is
            ␣set␣to␣1\n";
      }
      else options.SetInfectionRate_infectionFactor(value);
    }
    else
```

```
      {
        std::cout << "infection␣rate:␣\"" << currentoption[0] << "␣" << currentoption
            [1] << "\"␣is␣not␣valid␣input,␣";
        std::cout << currentoption[0] << "␣is␣set␣to␣default\n";
      }
    }
  }
  else
  {
    std::cout << "infection␣rate:␣\"" << currentoption[0] << "␣" << currentoption[1]
        << "\"␣is␣not␣valid␣input,␣";
    std::cout << currentoption[0] << "␣is␣set␣to␣default\n";
  }
}


//extern
void createInput()
{
  std::ofstream input;
  input.open("input.dat");

  input << ";Input␣file\n;'=text'␣are␣option␣categories.␣all␣options␣in␣this␣category␣
      are␣directly␣below\n";
  input << ";options␣consist␣of␣name␣and␣value␣separated␣with␣a␣space.␣a␣blank␣line␣or
      ␣&␣means␣the␣end␣of␣options␣in␣this␣category\n";
  input << ";comments␣start␣with␣';'\n\n";
  input << ";double␣values␣have␣an␣precision␣of␣3,␣so␣higher␣precision␣on␣numbers␣with
      ␣more␣than␣3␣values␣after␣the␣dot␣will␣be␣lost\n\n";


  input << ";general␣uptions:\n";
  input << "=general\n";
  input << "numberOfSimulations␣" << options.GetGeneral_numberOfSimulations() << "\t";
  input << ";integer␣number:␣sets␣the␣number␣of␣simulations\n";
  input << "nBalanced␣" << options.GetGeneral_nBalanced() << "\t\t";
  input << ";integer␣number:␣sets␣the␣balanced␣number␣of␣nodes,␣used␣by␣various␣types␣
      of␣rate␣calculations\n";
  input << "nMax␣" << options.GetGeneral_nMax() << "\t\t";
  input << ";integer␣number:␣sets␣the␣maximum␣number␣of␣nodes,␣both␣used␣to␣reserve␣
      memory␣and␣used␣by␣variouse␣types␣of␣rate␣calculations,␣cannot␣be␣lower␣than␣'
      nBalanced'\n";
  input << "nMin␣" << options.GetGeneral_nMin() << "\t\t\t";
  input << ";integer␣number:␣sets␣the␣minimum␣number␣of␣nodes,␣cannot␣be␣higher␣than␣'
      nBalanced'\n";
  input << "edgeFraction␣" << options.GetGeneral_edgeFraction() << "\t";
  input << ";double␣number:␣sets␣the␣fraction␣of␣balanced␣edges␣that␣are␣active,␣used␣
      by␣various␣rate␣functions␣and␣the␣balanced␣start␣for␣active␣edges\n";
  input << "fixedSeed␣" << options.GetGeneral_fixedSeed() << "\t\t";
  input << ";integer␣number:␣can␣be␣used␣so␣every␣simulation␣will␣have␣the␣same␣
      results(provided␣the␣options␣stay␣the␣same␣as␣well)\n";
  input << "&\n\n";


  input << ";start␣of␣simulation␣options:\n";
  input << "=startOfSimulation\n";
  input << "n␣" << options.GetStartOfSimulation_n() << "\t\t\t";
```

```
input << ";␣integer␣number:␣sets␣the␣number␣of␣nodes␣at␣the␣start␣of␣the␣simulation,␣
    must␣be␣between␣nMax␣and␣nMin\n";
input << "activeEdges␣" << options.GetStartOfSimulation_activeEdges() << "\t";
input << ";␣string␣(text):␣'none'␣means␣a␣system␣with␣no␣active␣edges,␣'full'␣means␣a
    ␣system␣with␣all␣edges␣active";
input << "␣and␣'balanced'␣means␣a␣system␣where␣each␣edge␣has␣a␣chance␣to␣be␣active␣
    according␣to␣the␣edgeFraction␣from␣the␣general␣options\n";
input << "&\n\n";


input << ";␣stopping␣criteria␣options:\n";
input << "=stoppingCriteria\n";
input << "time␣" << options.GetStoppingCriteria_time() << "\t\t\t";
input << ";␣double␣number:␣this␣is␣the␣time␣at␣which␣the␣simulation␣will␣stop\n";
input << "events␣" << options.GetStoppingCriteria_events() << "\t\t";
input << ";␣integer␣number:␣this␣is␣the␣maximum␣number␣of␣events␣until␣simulation␣
    will␣stop\n";
if (options.GetStoppingCriteria_virusExtinction()) input << "virusExtinction␣true\t"
    ;
else input << "virusExtinction␣false\t";
input << ";␣boolean:␣when␣true␣the␣simulation␣will␣terminate␣early␣when␣no␣nodes␣are␣
    infected␣anymore\n";
if (options.GetStoppingCriteria_timeOn()) input << "timeOn␣true\t\t";
else input << "timeOn␣false\t\t";
input << ";␣boolean:␣when␣true␣the␣simulation␣will␣terminate␣at␣the␣time␣limit␣if␣the
    ␣maximum␣number␣of␣events␣is␣not␣reached\n";
input << "&\n\n";


input << ";␣file␣output␣options\n";
input << "=fileOutput\n";
if (options.GetFileOutput_simulationData()) input << "simulationData␣true\t";
else input << "simulationData␣false\t";
input << ";␣boolean:␣when␣true␣this␣creates␣a␣data␣file␣with␣general␣information␣on␣
    each␣simulation␣run\n";
if (options.GetFileOutput_eventlog()) input << "eventlog␣true\t\t";
else input << "eventlog␣false\t\t";
input << ";␣boolean:␣when␣true␣this␣creates␣a␣data␣file␣with␣an␣eventlog␣of␣all␣
    events\n";
if (options.GetFileOutput_nodeInfo()) input << "nodeInfo␣true\t\t";
else input << "nodeInfo␣false\t\t";
input << ";␣boolean:␣when␣true␣this␣creates␣a␣data␣file␣with␣all␣node␣information␣
    after␣each␣event\n";
if (options.GetFileOutput_edgeInfo()) input << "edgeInfo␣true\t\t";
else input << "edgeInfo␣false\t\t";
input << ";␣boolean:␣when␣true␣this␣creates␣a␣data␣file␣with␣the␣edge␣matrix␣after␣
    each␣event\n";
if (options.GetFileOutput_ratesInfo()) input << "ratesInfo␣true\t\t";
else input << "ratesInfo␣false\t\t";
input << ";␣boolean:␣when␣true␣this␣creates␣a␣data␣file␣with␣all␣the␣rates␣used␣in␣
    the␣simulation␣after␣each␣event\n";
if (options.GetFileOutput_virusSpread()) input << "virusSpread␣true\t";
else input << "virusSpread␣false\t";
input << ";␣boolean:␣when␣true␣this␣creates␣a␣data␣file␣with␣the␣number␣of␣nodes,";
input << "␣number␣of␣infected␣nodes␣and␣the␣fraction␣of␣infected␣nodes␣after␣each␣
    event\n";
input << "&\n\n";
```

```
input << ";console␣output␣options:\n";
input << "=consoleOutput\n";
if (options.GetConsoleOutput_simulationData()) input << "simulationData␣true␣\t";
else input << "simulationData␣false\t";
input << ";boolean:␣when␣true␣this␣shows␣general␣simulation␣data␣on␣the␣console\n";
if (options.GetConsoleOutput_eventlog()) input << "eventlog␣true\t\t";
else input << "eventlog␣false\t\t";
input << ";boolean:␣when␣true␣this␣shows␣an␣eventlog␣on␣the␣console\n";
if (options.GetConsoleOutput_simulationPause()) input << "simulationPause␣true\t";
else input << "simulationPause␣false\t";
input << ";boolean:␣when␣true␣this␣will␣cause␣the␣simulation␣to␣pause␣after␣each␣
    simulation␣run\n";
input << "&\n\n";


input << ";base␣rate␣values␣settings\n";
input << "=rates\n";
input << "arNode␣" << options.GetRates_arNode() << "\t\t";
input << ";double␣number:␣sets␣the␣base␣rate␣for␣adding␣a␣node␣or␣the␣combined␣base␣
    rate␣of␣adding␣and␣removing␣a␣node␣when␣rates␣are␣dependent\n";
input << "arNode2␣" << options.GetRates_arNode2() << "\t\t";
input << ";double␣number:␣sets␣the␣base␣rate␣for␣removing␣a␣node␣when␣independent␣
    from␣adding␣a␣node\n";
input << "arEdge␣" << options.GetRates_arEdge() << "\t\t";
input << ";double␣number:␣sets␣the␣base␣rate␣for␣adding␣an␣edge␣or␣the␣combined␣base
    ␣rate␣of␣adding␣and␣removing␣an␣edge␣when␣rates␣are␣dependent,␣also␣the␣rate␣for␣
    changing␣edge␣status\n";
input << "arEdge2␣" << options.GetRates_arEdge2() << "\t\t";
input << ";double␣number:␣sets␣the␣base␣rate␣for␣removing␣an␣edge␣when␣independent␣
    from␣adding␣an␣edge\n";
input << "swapEdge␣" << options.GetRates_swapEdge() << "\t\t";
input << ";double␣number:␣sets␣the␣base␣rate␣for␣swapping␣2␣edges␣(or␣just␣there␣
    status)\n";
input << "infection␣" << options.GetRates_infection() << "\t\t";
input << ";double␣number:␣sets␣the␣base␣rate␣for␣infecting␣a␣node␣or␣the␣combined␣
    base␣rate␣of␣infecting␣and␣healing␣a␣node␣when␣rates␣are␣dependent\n";
input << "infection2␣" << options.GetRates_infection2() << "\t\t";
input << ";double␣number:␣sets␣the␣base␣rate␣for␣healing␣a␣node␣when␣independent␣
    from␣infecting␣a␣node\n";
input << "&\n\n";


input << ";\n";
input << "=arNodeRate\n";
input << "ratio␣" << options.GetArNodeRate_ratio() << "\t\t\t";
input << ";double␣number:␣the␣ratio␣between␣the␣base␣rate␣of␣adding␣and␣removing␣a␣
    node␣when␣dependent,␣";
input << "input␣is␣ratio␣for␣adding␣a␣node␣to␣1␣removing␣a␣node␣so␣input:1␣is␣the␣
    used␣ratio\n";
if (options.GetArNodeRate_dependent()) input << "dependent␣true\t\t";
else input << "dependent␣false\t\t";
input << ";boolean:␣when␣true␣the␣base␣rates␣of␣adding␣and␣removing␣a␣node␣are␣
    dependent␣and␣determined␣by␣the␣ratio\n";
input << "type␣" << options.GetArNodeRate_type() << "\t\t";
input << ";string␣(text):␣determines␣how␣rates␣are␣calculated/updated;␣'constant'␣
    means␣rates␣are␣equal␣to␣the␣base␣rates,␣";
input << "'n_linear'␣means␣the␣rates␣are␣linear␣dependent␣on␣the␣number␣of␣nodes,␣";
```

```
input << "'n_linear_add_only' means the rate for adding nodes is linear dependent on
    the number of nodes the rate for removing nodes is constant at base times
    nBalanced, ";
input << "'n_linear_remove_only' means the rate for removing nodes is linear
    dependent on the number of nodes the rate for adding nodes is constant at base
    times nBalanced\n";
input << "\t\t\t\t;'n_linear_balanced' means the rates are linear dependent and
    balanced to the base rate at nBalanced nodes in the system, ";
input << "'n_linear_balanced_add_only' means the add node rate is linear dependent
    and balanced to the base rate at nBalanced nodes in the system and the remove
    node rate is constant, ";
input << "'n_linear_balanced_remove_only'means the remove node rate is linear
    dependent and balanced to the base rate at nBalanced nodes in the system and the
    add node rate is constant, ";
input << "'constant_dependent_n_linear_balanced' means constant rates but when
    dependent on each other in such a way they favour reaching the balanced number of
    nodes nBalanced\n";
input << "addFactor " << options.GetArNodeRate_addFactor() << "\t\t";
input << ";double number: not used in current simulation program\n";
input << "removeFactor " << options.GetArNodeRate_removeFactor() << "\t\t";
input << ";double number: not used in current simulation program\n";
input << "&\n\n";


input << ";\n";
input << "=arEdgeRate\n";
if (options.GetArEdgeRate_changeStatus()) input << "changeStatus true\t";
else input << "changeStatus false\t";
input << ";boolean: when true the add and remove edge events are replaced by the
    change edge event that selects a random edge and changes its status\n";
input << "ratio " << options.GetArEdgeRate_ratio() << "\t\t\t";
input << ";double number: the ratio between the base rate of adding and removing an
    edge when dependent, ";
input << "input is ratio for adding an edge to 1 removing an edge so input:1 is the
    used ratio\n";
if (options.GetArEdgeRate_dependent()) input << "dependent true\t\t";
else input << "dependent false\t\t";
input << ";boolean: when true the base rates of adding and removing an edge are
    dependent and determined by the ratio\n";
input << "type " << options.GetArEdgeRate_type() << "\t\t";
input << ";string (text): determines how rates are calculated/updated; 'constant'
    means rates are equal to the base rates, ";
input << "'e_linear' means the rates are linear dependent on the number of edges in
    the system (active and/or inactive), ";
input << "'e_linear_balanced' means the rates are linear dependent and balanced to
    the base rate at the ideal fraction of active edges in the system, ";
input << "'constant_dependent_e_linear_balanced' means constant rates but when
    dependent on each other in such a way they favour reaching the ideal fraction of
    active edges\n";
input << "addFactor " << options.GetArEdgeRate_addFactor() << "\t\t";
input << ";double number: not used in current simulation program\n";
input << "removeFactor " << options.GetArEdgeRate_removeFactor() << "\t\t";
input << ";double number: not used in current simulation program\n";
input << "&\n\n";


input << ";\n";
input << "=swapEdgeRate\n";
```

```
  if (options.GetSwapEdgeRate_changeStatus()) input << "changeStatus true\t";
  else input << "changeStatus false\t";
  input << ";boolean: when true the event swap edge becomes swap edge status where two
      random edges swap their status instead of an active edge being swapped with an
      inactive one\n";
  input << "type " << options.GetSwapEdgeRate_type() << "\t\t";
  input << ";string (text): determines how rates are calculated/updated; 'constant'
      means rates are equal to the base rate, ";
  input << "'e_linear' means the rates are linear dependent on the number of (active)
      edges, ";
  input << "'e_linear_balanced' means the rates are linear dependent on the fraction
      of active edges (applies only to swap edge event)\n";
  input << "swapFactor " << options.GetSwapEdgeRate_swapFactor() << "\t\t";
  input << ";double number: not used in current simulation program\n";
  input << "&\n\n";


  input << "=infectionRate\n";
  input << "ratio " << options.GetInfectionRate_ratio() << "\t\t\t\t";
  input << ";double number: the ratio between the base rate of infecting and healing a
      node when dependent, ";
  input << "input is ratio for infecting a node to 1 healing a node so input:1 is the
      used ratio\n";
  if (options.GetInfectionRate_dependent()) input << "dependent true\t\t\t";
  else input << "dependent false\t\t\t";
  input << ";boolean: when true the base rates of infecting and healing a node are
      dependent and determined by the ratio\n";
  input << "infectionType " << options.GetInfectionRate_infectionType() << "\t\t";
  input << ";string (text): determines how rates are calculated/updated; 'constant'
      means rates are equal to the base rates, ";
  input << "'ni_linear' means the rates are linear dependent on the number of infected
      nodes, ";
  input << "'ni_linear_balanced' means the rates are linear dependent and balanced to
      a fraction of nBalanced nodes in the system determined by the infection factor, "
      ;
  input << "'ni_ne_linear' means the rates are linear dependent on the number of
      infected nodes times the number of edges connected to each infected node, ";
  input << "'ni_ne_linear_balanced' means the same as 'ni_linear_dependent but than
      also dependent on the number of edges connected to each infected node\n";
  input << "infectionFactor " << options.GetInfectionRate_infectionFactor() << "\t\t";
  input << ";double number: fraction to determine the fraction of nBalanced nodes to
      which the base rate is set for balanced types of rate calculations\n";
  input << "&\n\n";

  input << "; end of inputfile\n";

  input.close();

  std::cout << "input file created! run program again for results.\n";
}

void readInput()
{
  std::ifstream input;
  input.open("input.dat");

  while (input)
  {
```

```
// read stuff from the file into a string and print it
std::string strInput;
getline(input, strInput);

if (!strInput.empty() && strInput[0] != ';')
{

  if (strInput[0] == '=')
  {
    if (strInput == "=general")
    {
      do
      {
        getline(input, strInput);
        if (strInput[0] != ';' && strInput[0] != '␣' && strInput[0] != '&' && !
            strInput.empty())
        {
          general(strInput);
        }
      } while (!strInput.empty() && strInput[0] != '&');
    }

    else if (strInput == "=startOfSimulation")
    {
      do
      {
        getline(input, strInput);
        if (strInput[0] != ';' && strInput[0] != '␣' && strInput[0] != '&' && !
            strInput.empty())
        {
          startOfSimulation(strInput);
        }
      } while (!strInput.empty() && strInput[0] != '&');
    }

    else if (strInput == "=stoppingCriteria")
    {
      do
      {
        getline(input, strInput);
        if (strInput[0] != ';' && strInput[0] != '␣' && strInput[0] != '&' && !
            strInput.empty())
        {
          stoppingCriteria(strInput);
        }
      } while (!strInput.empty() && strInput[0] != '&');
    }

    else if (strInput == "=fileOutput")
    {
      do
      {
        getline(input, strInput);
        if (strInput[0] != ';' && strInput[0] != '␣' && strInput[0] != '&' && !
            strInput.empty())
        {
          fileOutput(strInput);
        }
```

```
  } while (!strInput.empty() && strInput[0] != '&');
}

else if (strInput == "=consoleOutput")
{
  do
  {
    getline(input, strInput);
    if (strInput[0] != ';' && strInput[0] != '␣' && strInput[0] != '&' &&
        strInput.empty())
    {
      consoleOutput(strInput);
    }
  } while (!strInput.empty() && strInput[0] != '&');
}

else if (strInput == "=rates")
{
  do
  {
    getline(input, strInput);
    if (strInput[0] != ';' && strInput[0] != '␣' && strInput[0] != '&' &&
        strInput.empty())
    {
      rates(strInput);
    }
  } while (!strInput.empty() && strInput[0] != '&');
}

else if (strInput == "=arNodeRate")
{
  do
  {
    getline(input, strInput);
    if (strInput[0] != ';' && strInput[0] != '␣' && strInput[0] != '&' &&
        strInput.empty())
    {
      arNodeRate(strInput);
    }
  } while (!strInput.empty() && strInput[0] != '&');
}

else if (strInput == "=arEdgeRate")
{
  do
  {
    getline(input, strInput);
    if (strInput[0] != ';' && strInput[0] != '␣' && strInput[0] != '&' &&
        strInput.empty())
    {
      arEdgeRate(strInput);
    }
  } while (!strInput.empty() && strInput[0] != '&');
}

else if (strInput == "=swapEdgeRate")
{
  do
```

```
            {
              getline(input, strInput);
              if (strInput[0] != ';' && strInput[0] != '␣' && strInput[0] != '&' && !
                  strInput.empty())
              {
                swapEdgeRate(strInput);
              }
            } while (!strInput.empty() && strInput[0] != '&');
          }

          else if (strInput == "=infectionRate")
          {
            do
            {
              getline(input, strInput);
              if (strInput[0] != ';' && strInput[0] != '␣' && strInput[0] != '&' && !
                  strInput.empty())
              {
                infectionRate(strInput);
              }
            } while (!strInput.empty() && strInput[0] != '&');
          }
        }

      }
    }
}
```

C.2.7 output.cpp:

```
// output.cpp : prints data to screen (prints data to output file later)
//

#include "stdafx.h"

// headers included with compiler
#include <cstdint>
#include <iostream>
#include <fstream>
#include <vector>

// headers
#include "datastructs.h"
#include "Node.h"
#include "options.h"

static std::ofstream eventlog;
static std::ofstream results;
static std::ofstream simulationData;
static std::ofstream nodeInfo;
static std::ofstream edgeInfo;
static std::ofstream ratesInfo;
static std::ofstream virusspread;


void removeOutput()
```

```
{
  if (std::ifstream("results.dat")) std::remove("results.dat");
  if (std::ifstream("simulationdata.dat")) std::remove("simulationdata.dat");
  if (std::ifstream("eventlog.dat")) std::remove("eventlog.dat");
  if (std::ifstream("nodeinfo.dat")) std::remove("nodeinfo.dat");
  if (std::ifstream("edgeinfo.dat")) std::remove("edgeinfo.dat");
  if (std::ifstream("ratesinfo.dat")) std::remove("ratesinfo.dat");
  if (std::ifstream("virusspread.dat")) std::remove("virusspread.dat");
}

void openOutput()
{
  results.open("results.dat");
  if (options.GetFileOutput_simulationData()) simulationData.open("simulationdata.dat"
      );
  if (options.GetFileOutput_eventlog()) eventlog.open("eventlog.dat");
  if (options.GetFileOutput_nodeInfo()) nodeInfo.open("nodeinfo.dat");
  if (options.GetFileOutput_edgeInfo()) edgeInfo.open("edgeinfo.dat");
  if (options.GetFileOutput_ratesInfo()) ratesInfo.open("ratesinfo.dat");
  if (options.GetFileOutput_virusSpread()) virusspread.open("virusspread.dat");
}

void printOptions()
{
  if (options.GetFileOutput_simulationData())
  {
    simulationData << "\nOptions:\n";

    simulationData << "General:\n";
    simulationData << "Number␣of␣Simulations:␣" << options.
        GetGeneral_numberOfSimulations() << "\n";
    simulationData << "n␣balanced:␣" << options.GetGeneral_nBalanced() << "\n";
    simulationData << "n␣max:␣" << options.GetGeneral_nMax() << "\n";
    simulationData << "n␣min:␣" << options.GetGeneral_nMin() << "\n";
    simulationData << "edge␣fraction:␣" << options.GetGeneral_edgeFraction() << "\n";
    simulationData << "fixed␣seed:␣" << options.GetGeneral_fixedSeed() << "\n\n";

    simulationData << "Start␣of␣simulation:\n";
    simulationData << "n:␣" << options.GetStartOfSimulation_n() << "\n";
    simulationData << "active␣edges:␣" << options.GetStartOfSimulation_activeEdges()
        << "\n\n";

    simulationData << "Stopping␣criteria:\n";
    simulationData << "time:␣" << options.GetStoppingCriteria_time() << "\n";
    simulationData << "events:␣" << options.GetStoppingCriteria_events() << "\n";
    if (options.GetStoppingCriteria_virusExtinction()) simulationData << "virus␣
        extinction:␣true\n";
    else simulationData << "virus␣extinction:␣false\n";
    if (options.GetStoppingCriteria_timeOn()) simulationData << "time␣limit␣on:␣true\n
        \n";
    else simulationData << "time␣limit␣on:␣false\n\n";

    simulationData << "File␣output:\n";
    if (options.GetFileOutput_simulationData()) simulationData << "simulation␣data:␣
        true\n";
    else simulationData << "simulation␣data:␣false\n";
    if (options.GetFileOutput_eventlog()) simulationData << "eventlog:␣true\n";
    else simulationData << "eventlog:␣false\n";
```

```
    if (options.GetFileOutput_nodeInfo()) simulationData << "node info: true\n";
    else simulationData << "node info: false\n";
    if (options.GetFileOutput_edgeInfo()) simulationData << "edge info: true\n";
    else simulationData << "edge info: false\n";
    if (options.GetFileOutput_ratesInfo()) simulationData << "rates info: true\n";
    else simulationData << "rates info: false\n";
    if (options.GetFileOutput_virusSpread()) simulationData << "virus spread: true\n\n
        ";
    else simulationData << "virus spread: false\n\n";

    simulationData << "Console output:\n";
    if (options.GetConsoleOutput_simulationData()) simulationData << "simulation data:
        true\n";
    else simulationData << "simulation data: false\n";
    if (options.GetConsoleOutput_eventlog()) simulationData << "eventlog: true\n";
    else simulationData << "eventlog: false\n";
    if (options.GetConsoleOutput_simulationPause()) simulationData << "simulation
        pause: true\n\n";
    else simulationData << "simulation pause: false\n\n";

    simulationData << "Rates:\n";
    simulationData << "add/remove node: " << options.GetRates_arNode() << "\n";
    simulationData << "add/remove node 2: " << options.GetRates_arNode2() << "\n";
    simulationData << "add/remove edge: " << options.GetRates_arEdge() << "\n";
    simulationData << "add/remove edge 2: " << options.GetRates_arEdge2() << "\n";
    simulationData << "swap edge: " << options.GetRates_swapEdge() << "\n";
    simulationData << "infection/heal: " << options.GetRates_infection() << "\n\n";
    simulationData << "infection/heal 2: " << options.GetRates_infection2() << "\n";

    simulationData << "Add/remove node rate:\n";
    simulationData << "ratio: " << options.GetArNodeRate_ratio() << "\n";
    if (options.GetArNodeRate_dependent()) simulationData << "dependent: true\n";
    else simulationData << "dependent: false\n";
    simulationData << "type: " << options.GetArNodeRate_type() << "\n";
    simulationData << "add factor: " << options.GetArNodeRate_addFactor() << "\n";
    simulationData << "remove factor: " << options.GetArNodeRate_removeFactor() << "\n
        \n";

    simulationData << "Add/remove edge rate:\n";
    if (options.GetArEdgeRate_changeStatus()) simulationData << "change status: true\n
        ";
    else simulationData << "change status: false\n";
    simulationData << "ratio: " << options.GetArEdgeRate_ratio() << "\n";
    if (options.GetArEdgeRate_dependent()) simulationData << "dependent: true\n";
    else simulationData << "dependent: false\n";
    simulationData << "type: " << options.GetArEdgeRate_type() << "\n";
    simulationData << "add factor: " << options.GetArEdgeRate_addFactor() << "\n";
    simulationData << "remove factor: " << options.GetArEdgeRate_removeFactor() << "\n
        \n";

    simulationData << "Swap edge rate:\n";
    if (options.GetSwapEdgeRate_changeStatus()) simulationData << "change status: true
        \n";
    else simulationData << "change status: false\n";
    simulationData << "type: " << options.GetSwapEdgeRate_type() << "\n";
    simulationData << "swap factor: " << options.GetSwapEdgeRate_swapFactor() << "\n\n
        ";
```

```
    simulationData << "Infection␣rate:\n";
    simulationData << "ratio:␣" << options.GetInfectionRate_ratio() << "\n";
    if (options.GetInfectionRate_dependent()) simulationData << "dependent:␣true\n";
    else simulationData << "dependent:␣false\n";
    simulationData << "infection␣type:␣" << options.GetInfectionRate_infectionType()
        << "\n";
    simulationData << "infection␣factor:␣" << options.GetInfectionRate_infectionFactor
        () << "\n\n";
  }
}

void printTitleResults()
{
  results << "results:\n";
  results << "sim\ttime\tn\tn_infected\tevents\thighest_infection_fraction\n";
}

void printTitle()
{
  if (options.GetFileOutput_eventlog())
    eventlog << "\n\nSIMULATION␣" << SimulationValues::GetSimulationNumber() << ":\n";

  if (options.GetFileOutput_virusSpread())
    virusspread << "SIMULATION␣" << SimulationValues::GetSimulationNumber() << ":\n";
    virusspread << "n\tn_inf\tfraction\n";

  //diagnostics
  if (options.GetFileOutput_simulationData())
    simulationData << "\n\nSIMULATION␣" << SimulationValues::GetSimulationNumber() <<
        ":\n";

  if (options.GetFileOutput_nodeInfo())
    nodeInfo << "\n\nSIMULATION␣" << SimulationValues::GetSimulationNumber() << ":\n";

  if (options.GetFileOutput_edgeInfo())
    edgeInfo << "\n\nSIMULATION␣" << SimulationValues::GetSimulationNumber() << ":\n";

  if (options.GetFileOutput_ratesInfo())
    ratesInfo << "\n\nSIMULATION␣" << SimulationValues::GetSimulationNumber() << ":\n"
        ;

    // console output
  std::cout << "\n\nSIMULATION␣" << SimulationValues::GetSimulationNumber() << "␣
      results:\n";
}

void printBaseRates(const RateVector_t &rates)
{
  if (options.GetFileOutput_ratesInfo())
  {
    ratesInfo << "Base␣rates:\n";

    for (uint32_t it(0); it < rates.size(); ++it)
    {
      switch (rates[it].type)
      {
      case RateType::ADD_NODE:
      {
```

```
      ratesInfo << "add node: " << rates[it].baseValue << "\t";
      break;
    }
    case RateType::REMOVE_NODE:
    {
      ratesInfo << "remove node: " << rates[it].baseValue << "\t";
      break;
    }
    case RateType::CHANGE_EDGE:
    {
      ratesInfo << "change edge: " << rates[it].baseValue << "\t";
      break;
    }
    case RateType::ADD_EDGE:
    {
      ratesInfo << "add edge: " << rates[it].baseValue << "\t";
      break;
    }
    case RateType::REMOVE_EDGE:
    {
      ratesInfo << "remove edge: " << rates[it].baseValue << "\t";
      break;
    }
    case RateType::SWAP_EDGE_STATUS:
    {
      ratesInfo << "swap edge status: " << rates[it].baseValue << "\t";
      break;
    }
    case RateType::SWAP_EDGE:
    {
      ratesInfo << "swap edge: " << rates[it].baseValue << "\t";
      break;
    }
    case RateType::INFECT_NODE:
    {
      ratesInfo << "infect node: " << rates[it].baseValue << "\t";
      break;
    }
    case RateType::HEAL_NODE:
    {
      ratesInfo << "heal node: " << rates[it].baseValue << "\t";
      break;
    }
    }
  }

  ratesInfo << "\nInitial rates:\n";

  for (uint32_t it(0); it < rates.size(); ++it)
  {
    switch (rates[it].type)
    {
    case RateType::ADD_NODE:
    {
      ratesInfo << "add node: " << rates[it].value << "\t";
      break;
    }
    case RateType::REMOVE_NODE:
```

```
      {
        ratesInfo << "remove node: " << rates[it].value << "\t";
        break;
      }
      case RateType::CHANGE_EDGE:
      {
        ratesInfo << "change edge: " << rates[it].value << "\t";
        break;
      }
      case RateType::ADD_EDGE:
      {
        ratesInfo << "add edge: " << rates[it].value << "\t";
        break;
      }
      case RateType::REMOVE_EDGE:
      {
        ratesInfo << "remove edge: " << rates[it].value << "\t";
        break;
      }
      case RateType::SWAP_EDGE_STATUS:
      {
        ratesInfo << "swap edge status: " << rates[it].value << "\t";
        break;
      }
      case RateType::SWAP_EDGE:
      {
        ratesInfo << "swap edge: " << rates[it].value << "\t";
        break;
      }
      case RateType::INFECT_NODE:
      {
        ratesInfo << "infect node: " << rates[it].value << "\t";
        break;
      }
      case RateType::HEAL_NODE:
      {
        ratesInfo << "heal node: " << rates[it].value << "\t";
        break;
      }
      }
    }

    ratesInfo << "\nEvent rates:\n";
  }
}

void printStartOfSimulation(const NodeVector_t &nodeList, const NodeVector_t &
    nodeListInactive, const EdgeMatrix_t &edgeData)
{
  if (options.GetFileOutput_simulationData())
  {
    simulationData << "start of simulation:\n\n";
    simulationData << "initial nodes:\n";

    for (uint32_t count = 0; count < nodeList.size(); count++)
    {
      simulationData << "Node ID = " << nodeList[count].GetID() << "\n";
    }
```

```
    simulationData << "\nnodeList␣size␣=␣" << nodeList.size() << "\n";
    simulationData << "nodeListInactive␣size␣=␣" << nodeListInactive.size() << "\n";
    simulationData << "Edge␣data␣matrix␣size␣=␣" << edgeData.size() << "\n\n";


  }

  if (options.GetConsoleOutput_simulationData())
  {
    std::cout << "start␣of␣simulation:\n";
    std::cout << "nodeList␣size␣=␣" << nodeList.size() << "\n";
    std::cout << "nodeListInactive␣size␣=␣" << nodeListInactive.size() << "\n";
    std::cout << "Edge␣data␣matrix␣size␣=␣" << edgeData.size() << "\n\n";
  }
}

void printVirusSpread()
{
  virusspread << Node::GetN() << "\t" << Node::GetNInfected() << "\t";
  if (Node::GetN() == 0) virusspread << "0\n";
  else virusspread << static_cast<double>(Node::GetNInfected()) / Node::GetN() << "\n"
      ;
}

void printEdgeData(const EdgeMatrix_t &edgeData)
{
  if (options.GetFileOutput_edgeInfo())
  {
    edgeInfo << "Event␣" << SimulationValues::GetEventNumber() << ":\n";
    for (uint32_t irow = 0; irow < edgeData.size(); ++irow)
    {
      for (uint32_t icol = 0; icol < edgeData.size(); ++icol)
      {
        if (edgeData[irow][icol].GetStatus())
          edgeInfo << "1␣";
        else
          edgeInfo << "0␣";
      }
      edgeInfo << "\n";
    }
    edgeInfo << "\n";
  }
}

void printNodeData(const NodeVector_t &nodeList, const NodeVector_t &nodeListInactive)
{
  if (options.GetFileOutput_nodeInfo())
  {
    nodeInfo << "Event␣" << SimulationValues::GetEventNumber() << ":\n";
    nodeInfo << "Nodes␣in␣system:\n";
    for (uint32_t count = 0; count < nodeList.size(); count++)
    {
      nodeInfo << "Node␣ID␣=␣" << nodeList[count].GetID() << "\t";
      nodeInfo << "number␣of␣edges␣=␣" << nodeList[count].GetNumberOfEdges() << "\t";
      nodeInfo << "Node␣status␣=␣";
      switch (nodeList[count].GetStatus())
      {
```

```
      case NodeStatus::SUSCEPTIBLE: { nodeInfo << "susceptible\n"; break; }
      case NodeStatus::INFECTED: { nodeInfo << "infected\n"; break; }
      case NodeStatus::HEALED: { nodeInfo << "healed\n"; break; }
      }
    }

    nodeInfo << "\nNodes removed from system:\n";
    for (uint32_t count = 0; count < nodeListInactive.size(); count++)
    {
      nodeInfo << "Node ID = " << nodeListInactive[count].GetID() << "\n";
    }

    nodeInfo << "\n";
  }
}

void printRatesInfo(const RateVector_t rates)
{
  if (options.GetFileOutput_ratesInfo())
  {
    for (uint32_t it(0); it < rates.size(); ++it)
    {
      switch (rates[it].type)
      {
      case RateType::ADD_NODE:
      {
        ratesInfo << "add node: " << rates[it].value << "\t";
        break;
      }
      case RateType::REMOVE_NODE:
      {
        ratesInfo << "remove node: " << rates[it].value << "\t";
        break;
      }
      case RateType::CHANGE_EDGE:
      {
        ratesInfo << "change edge: " << rates[it].value << "\t";
        break;
      }
      case RateType::ADD_EDGE:
      {
        ratesInfo << "add edge: " << rates[it].value << "\t";
        break;
      }
      case RateType::REMOVE_EDGE:
      {
        ratesInfo << "remove edge: " << rates[it].value << "\t";
        break;
      }
      case RateType::SWAP_EDGE_STATUS:
      {
        ratesInfo << "swap edge status: " << rates[it].value << "\t";
        break;
      }
      case RateType::SWAP_EDGE:
      {
        ratesInfo << "swap edge: " << rates[it].value << "\t";
        break;
```

```
      }
      case RateType::INFECT_NODE:
      {
        ratesInfo << "infect␣node:␣" << rates[it].value << "\t";
        break;
      }
      case RateType::HEAL_NODE:
      {
        ratesInfo << "heal␣node:␣" << rates[it].value << "\t";
        break;
      }
      }
    }

    ratesInfo << "\n";
  }
}

void printEventLog(const Result result, const NodeVector_t nodeList, const RateType &
    eventType)
{
  if (options.GetFileOutput_eventlog())
  {
    eventlog << SimulationValues::GetEventNumber() << ":\tTime:␣" << SimulationValues
        ::GetTime() << "\t";

    switch (eventType)
    {
      case RateType::ADD_NODE:
      {
        eventlog << "Event:␣add␣node\t\t";
        eventlog << "n:␣" << Node::GetN() << "\t";
        eventlog << "e:␣" << Node::GetE() << "\t";
        eventlog << "n_infected:␣" << Node::GetNInfected() << "\t";
        if (result.code == ErrorCode::OK)
        {
          eventlog << "result:\tsucces\t";
          eventlog << "node␣position(ID):\t" << Node::GetN() << "(" << result.node1ID
              << ")\t";
        }
        else if (result.code == ErrorCode::NO_NODE_AVAILABLE) eventlog << "result:\tno
            ␣node␣available\t";
        else if (result.code == ErrorCode::MAX_NODES_REACHED) eventlog << "result:\
            tmaximum␣nodes␣reached\t";
        else eventlog << "result:\tfailure\t";


        break;
      }
      case RateType::REMOVE_NODE:
      {
        eventlog << "Event:␣remove␣node\t";
        eventlog << "n:␣" << Node::GetN() << "\t";
        eventlog << "e:␣" << Node::GetE() << "\t";
        eventlog << "n_infected:␣" << Node::GetNInfected() << "\t";
        if (result.code == ErrorCode::OK)
        {
          eventlog << "result:\tsucces\t";
```

```
      eventlog << "node␣position(ID):\t" << result.node1 << "(" << result.node1ID
          << ")\t";
    }
    else if (result.code == ErrorCode::NO_NODE_AVAILABLE) eventlog << "result:\tno
        ␣node␣available\t";
    else if (result.code == ErrorCode::MIN_NODES_REACHED) eventlog << "result:\
        tminimum␣nodes␣reached\t";
    else eventlog << "result:\tfailure\t";

    break;
  }
  case RateType::CHANGE_EDGE:
  {
    eventlog << "Event:␣change␣edge␣\t";
    eventlog << "n:␣" << Node::GetN() << "\t";
    eventlog << "e:␣" << Node::GetE() << "\t";
    eventlog << "n_infected:␣" << Node::GetNInfected() << "\t";
    if (result.code == ErrorCode::OK)
    {
      eventlog << "result:\tsucces\t";
      eventlog << "edge␣position(ID):\t" << result.node1 << "," << result.node2 <<
          "␣";
      eventlog << "(" << result.node1ID << "," << result.node2ID << ")\t";
    }
    else if (result.code == ErrorCode::NO_EDGE_AVAILABLE) eventlog << "result:\tno
        ␣edge␣available\t";
    else eventlog << "result:\tfailure\t";

    break;
  }
  case RateType::ADD_EDGE:
  {
    eventlog << "Event:␣add␣edge\t\t";
    eventlog << "n:␣" << Node::GetN() << "\t";
    eventlog << "e:␣" << Node::GetE() << "\t";
    eventlog << "n_infected:␣" << Node::GetNInfected() << "\t";
    if (result.code == ErrorCode::OK)
    {
      eventlog << "result:\tsucces\t";
      eventlog << "edge␣position(ID):\t" << result.node1 << "," << result.node2 <<
          "␣";
      eventlog << "(" << result.node1ID << "," << result.node2ID << ")\t";
    }
    else if (result.code == ErrorCode::NO_EDGE_AVAILABLE) eventlog << "result:\tno
        ␣edge␣available\t";
    else eventlog << "result:\tfailure\t";

    break;
  }
  case RateType::REMOVE_EDGE:
  {
    eventlog << "Event:␣remove␣edge\t";
    eventlog << "n:␣" << Node::GetN() << "\t";
    eventlog << "e:␣" << Node::GetE() << "\t";
    eventlog << "n_infected:␣" << Node::GetNInfected() << "\t";
    if (result.code == ErrorCode::OK)
    {
      eventlog << "result:\tsucces\t";
```

```
      eventlog << "edge␣position(ID):\t" << result.node1 << "," << result.node2 <<
          "␣";
      eventlog << "(" << result.node1ID << "," << result.node2ID << ")\t";
    }
    else if (result.code == ErrorCode::NO_EDGE_AVAILABLE) eventlog << "result:\tno
        ␣edge␣available\t";
    else eventlog << "result:\tfailure\t";
    break;
  }
  case RateType::SWAP_EDGE_STATUS:
  {
    eventlog << "Event:␣swap␣edge␣status\t";
    eventlog << "n:␣" << Node::GetN() << "\t";
    eventlog << "e:␣" << Node::GetE() << "\t";
    eventlog << "n_infected:␣" << Node::GetNInfected() << "\t";
    if (result.code == ErrorCode::OK)
    {
      eventlog << "result:\tsucces\t";
      eventlog << "edge␣1␣position(ID):\t" << result.node1 << "," << result.node2
          << "␣";
      eventlog << "(" << result.node1ID << "," << result.node2ID << ")\t";
      eventlog << "edge␣2␣position(ID):\t" << result.node3 << "," << result.node4
          << "␣";
      eventlog << "(" << result.node3ID << "," << result.node4ID << ")\t";
    }
    else if (result.code == ErrorCode::NO_EDGE_AVAILABLE) eventlog << "result:\tno
        ␣edge␣available\t";
    else if (result.code == ErrorCode::EDGES_HAVE_SAME_STATUS) eventlog << "result
        :\tedges␣had␣same␣status:␣no␣swap␣needed\t";
    else eventlog << "result:\tfailure\t";
    break;
  }
  case RateType::SWAP_EDGE:
  {
    eventlog << "Event:␣swap␣edge\t";
    eventlog << "n:␣" << Node::GetN() << "\t";
    eventlog << "e:␣" << Node::GetE() << "\t";
    eventlog << "n_infected:␣" << Node::GetNInfected() << "\t";
    if (result.code == ErrorCode::OK)
    {
      eventlog << "result:\tsucces\t";
      eventlog << "edge␣1␣position(ID):\t" << result.node1 << "," << result.node2
          << "␣";
      eventlog << "(" << result.node1ID << "," << result.node2ID << ")\t";
      eventlog << "edge␣2␣position(ID):\t" << result.node3 << "," << result.node4
          << "␣";
      eventlog << "(" << result.node3ID << "," << result.node4ID << ")\t";
    }
    else if (result.code == ErrorCode::NO_EDGE_AVAILABLE) eventlog << "result:\tno
        ␣edge␣available\t";
    else eventlog << "result:\tfailure\t";
    break;
  }
  case RateType::INFECT_NODE:
  {
    eventlog << "Event:␣infect␣node\t";
    eventlog << "n:␣" << Node::GetN() << "\t";
    eventlog << "e:␣" << Node::GetE() << "\t";
```

```
        eventlog << "n_infected:␣" << Node::GetNInfected() << "\t";
        if (result.code == ErrorCode::OK)
        {
          eventlog << "result:\tsucces\t";
          eventlog << "node␣positions(ID):\t" << result.node1 << "," << result.node2
              << "␣";
          eventlog << "(" << result.node1ID << "," << result.node2ID << ")\t";
        }
        else if (result.code == ErrorCode::NO_EDGE_AVAILABLE) eventlog << "result:\tno
            ␣edge␣available\t";
        else if (result.code == ErrorCode::NO_NODE_AVAILABLE) eventlog << "result:\tno
            ␣node␣available\t";
        else if (result.code == ErrorCode::ALREADY_INFECTED) eventlog << "result:\tall
            ␣connected␣nodes␣already␣infected\t";
        else eventlog << "result:\tfailure\t";
        break;
      }
      case RateType::HEAL_NODE:
      {
        eventlog << "Event:␣heal␣node\t";
        eventlog << "n:␣" << Node::GetN() << "\t";
        eventlog << "e:␣" << Node::GetE() << "\t";
        eventlog << "n_infected:␣" << Node::GetNInfected() << "\t";
        if (result.code == ErrorCode::OK)
        {
          eventlog << "result:\tsucces\t";
          eventlog << "node␣position(ID):\t" << result.node1 << "(" << result.node1ID
              << ")\t";
        }
        else if (result.code == ErrorCode::NO_NODE_AVAILABLE) eventlog << "result:\tno
            ␣node␣available\t";
        else eventlog << "result:\tfailure\t";
        break;
      }
    }

    eventlog << "\n";
  }

  if (options.GetConsoleOutput_eventlog())
  {
    std::cout << SimulationValues::GetEventNumber() << ":\tTime:␣" << SimulationValues
        ::GetTime() << "\t";

    switch (eventType)
    {
      case RateType::ADD_NODE:
      {
        std::cout << "Event:␣add␣node\t\t";
        std::cout << "n:␣" << Node::GetN() << "\t";
        break;
      }
      case RateType::REMOVE_NODE:
      {
        std::cout << "Event:␣remove␣node\t";
        std::cout << "n:␣" << Node::GetN() << "\t";
        break;
      }
```

```
      case RateType::CHANGE_EDGE:
      {
        std::cout << "Event: change edge\t";
        std::cout << "n: " << Node::GetN() << "\t";
        break;
      }
      case RateType::ADD_EDGE:
      {
        std::cout << "Event: add edge\t\t";
        std::cout << "n: " << Node::GetN() << "\t";
        break;
      }
      case RateType::REMOVE_EDGE:
      {
        std::cout << "Event: remove edge\t";
        std::cout << "n: " << Node::GetN() << "\t";
        break;
      }
      case RateType::SWAP_EDGE_STATUS:
      {
        std::cout << "Event: swap edge status\t";
        std::cout << "n: " << Node::GetN() << "\t";
        break;
      }
      case RateType::SWAP_EDGE:
      {
        std::cout << "Event: swap edge\t\t";
        std::cout << "n: " << Node::GetN() << "\t";
        break;
      }
      case RateType::INFECT_NODE:
      {
        std::cout << "Event: infect node\t";
        std::cout << "n: " << Node::GetN() << "\t";
        break;
      }
      case RateType::HEAL_NODE:
      {
        std::cout << "Event: heal node\t";
        std::cout << "n: " << Node::GetN() << "\t";
        break;
      }
    }

    std::cout << "\n";


  }
}

void printResults( const NodeVector_t &nodeList)
{
  results << SimulationValues::GetSimulationNumber() << "\t";
  results << SimulationValues::GetTime() << "\t";
  results <<  Node::GetN() << "\t";
  results <<  Node::GetNInfected() << "\t";
  results << SimulationValues::GetEventNumber() << "\t";
  results << SimulationValues::GetHighestInfectionFraction() << "\n";
```

```
  std::cout << "end␣of␣simulation␣time␣=␣" << SimulationValues::GetTime() << ",\t";
  std::cout << "number␣of␣infected␣nodes:␣"<< Node::GetNInfected() << "\n";
}

void printEndOfSimulation(const NodeVector_t &nodeList, const NodeVector_t &
    nodeListInactive, const EdgeMatrix_t &edgeData)
{
  if (options.GetFileOutput_simulationData())
  {
    simulationData << "\nend␣of␣simulation:\n";
    simulationData << "ID␣list␣of␣nodes␣in␣system\n";
    for (uint32_t count = 0; count < nodeList.size(); count++)
    {
      simulationData << "Node␣ID␣=␣" << nodeList[count].GetID() << "\n";
    }

    simulationData << "\nID␣list␣of␣nodes␣removed␣from␣system\n";
    for (uint32_t count = 0; count < nodeListInactive.size(); count++)
    {
      simulationData << "Node␣ID␣=␣" << nodeListInactive[count].GetID() << "\n";
    }

    simulationData << "\nnodeList␣size␣=␣" << nodeList.size() << "\n";
    simulationData << "nodeListInactive␣size␣=␣" << nodeListInactive.size() << "\n";
    simulationData << "Edge␣data␣matrix␣size␣=␣" << edgeData.size() << "\n\n";
  }

  if (options.GetConsoleOutput_simulationData())
  {
    std::cout << "\nEnd␣of␣simulation:\n";
    std::cout << "nodeList␣size␣=␣" << nodeList.size() << "\n";
    std::cout << "nodeListInactive␣size␣=␣" << nodeListInactive.size() << "\n";
    std::cout << "Edge␣data␣matrix␣size␣=␣" << edgeData.size() << "\n\n";
  }
}
```

C.2.8 Node.h:

```
#ifndef NODE_H
#define NODE_H

// headers from compiler
#include <cstdint>

enum class NodeStatus
{
  SUSCEPTIBLE,
  INFECTED,
  HEALED,
  TOTAL
};

class Node
{
```

```
private:
  static int32_t s_nIDGenerator;
  static int32_t s_n;
  static int32_t s_e;
  static int32_t s_nInfected;

  int32_t m_nID;
  NodeStatus m_status;
  int32_t m_numberOfEdges;

public:
  Node()
  {
    m_nID = ++s_nIDGenerator;
    m_status = NodeStatus::SUSCEPTIBLE;
    m_numberOfEdges = 0;
  }

  // static member functions
  static int32_t GetN() { return s_n; }
  static void IncreaseN() { ++s_n; }
  static void DecreaseN() { --s_n; }
  static void SetN(int32_t input) { s_n = input; }

  static int32_t GetE() { return s_e; }
  static void IncreaseE() { ++s_e; }
  static void DecreaseE() { --s_e; }
  static void SetE(int32_t input) { s_e = input; }

  static int32_t GetNInfected() { return s_nInfected; }
  static void IncreaseNInfected() { ++s_nInfected; }
  static void DecreaseNInfected() { --s_nInfected; }
  static void SetNInfected(int32_t input) { s_nInfected = input; }

  static void SetIDGenerator(int32_t input) { s_nIDGenerator = input; }

  // rest
  int32_t GetID() const { return m_nID; }
  int32_t GetNumberOfEdges() const { return m_numberOfEdges; }
  NodeStatus GetStatus() const { return m_status; }

  void IncreaseNumberOfEdges() { ++m_numberOfEdges; }
  void DecreaseNumberOfEdges() { --m_numberOfEdges; }
  void ChangeStatus(NodeStatus status) { m_status = status; }

  void SetNumberOfEdges(int32_t numberOfEdges) { m_numberOfEdges = numberOfEdges; }
};

#endif#pragma once
```

C.2.9 Node.cpp:

```
// Node.cpp : functions for the Node class.
//

#include "stdafx.h"
```

```
// headers included with compiler
#include <cstdint>

// headers
#include "Node.h"

int32_t Node::s_nIDGenerator = 0;
int32_t Node::s_n = 0;
int32_t Node::s_e = 0;
int32_t Node::s_nInfected = 0;
```

C.2.10 datastructs.h:

```
#ifndef DATASTRUCTS_H
#define DATASTRUCTS_H

// headers from compiler
#include <cstdint>
#include <vector>

// headers
#include "Node.h"

enum class RateType
{
  ADD_NODE,
  REMOVE_NODE,
  CHANGE_EDGE,
  ADD_EDGE,
  REMOVE_EDGE,
  SWAP_EDGE_STATUS,
  SWAP_EDGE,
  INFECT_NODE,
  HEAL_NODE,
  TOTAL
};

enum class ErrorCode
{
  OK,
  NO_NODE_AVAILABLE,
  NO_EDGE_AVAILABLE,
  ALREADY_INFECTED,
  UPDATE_ERROR,
  MAX_NODES_REACHED,
  MIN_NODES_REACHED,
  EDGES_HAVE_SAME_STATUS,
  TOTAL

};

struct Rate
{
  RateType type;
  const double baseValue;
```

```
  double value;
};

struct Result
{
  ErrorCode code;
  int32_t node1;
  int32_t node2;
  int32_t node3;
  int32_t node4;
  int32_t node1ID;
  int32_t node2ID;
  int32_t node3ID;
  int32_t node4ID;

};

class Edge
{
private:
  bool m_status;

public:
  Edge() { m_status = false; }
  Edge(bool status) { m_status = status; }

  bool GetStatus() const { return m_status; }
  void SetStatus(bool status) { m_status = status; }
};

class SimulationValues
{
private:
  static int32_t s_eventNumber;
  static double s_time;
  static int32_t s_simulationNumber;
  static double s_highestInfectionFraction;

public:
  static int32_t GetEventNumber() { return s_eventNumber; }
  static void IncreaseEventNumber() { ++s_eventNumber; }
  static void SetEventNumber(int32_t input) { s_eventNumber = input; }

  static double GetTime() { return s_time; }
  static void SetTime(double input) { s_time = input; }

  static int32_t GetSimulationNumber() { return s_simulationNumber; }
  static void IncreaseSimulationNumber() { ++s_simulationNumber; }

  static double GetHighestInfectionFraction() { return s_highestInfectionFraction; }
  static void SetHighestInfectionFraction(double input) { s_highestInfectionFraction =
       input; }
};

typedef std::vector<Node> NodeVector_t;
typedef std::vector<Rate> RateVector_t;
typedef std::vector<Edge> EdgeVector_t;
typedef std::vector<EdgeVector_t> EdgeMatrix_t;
```

```
#endif
#pragma once
```

C.2.11 datastructs.cpp:

```
// datastructs.cpp : functions for the Node class.
//

#include "stdafx.h"

// headers included with compiler
#include <cstdint>

// headers
#include "datastructs.h"

int32_t SimulationValues::s_eventNumber = 0;
double SimulationValues::s_time = 0;
int32_t SimulationValues::s_simulationNumber = 0;
double SimulationValues::s_highestInfectionFraction = 0;
```

C.2.12 options.h:

```
#ifndef OPTIONS_H
#define OPTIONS_H

// headers from compiler
#include <cstdint>
#include <iostream>
#include <string>

class Options
{
private:
  // General options
  int32_t general_numberOfSimulations;
  int32_t general_nBalanced;
  int32_t general_nMax;
  int32_t general_nMin;
  double general_edgeFraction;
  int32_t general_fixedSeed;

  // Start of Simulation options
  int32_t startOfSimulation_n;
  std::string startOfSimulation_activeEdges;

  // Stopping Criteria options
  double stoppingCriteria_time;
  int32_t stoppingCriteria_events;
  bool stoppingCriteria_virusExtinction;
  bool stoppingCriteria_timeOn;
```

```
  // File output options
  bool fileOutput_simulationData;
  bool fileOutput_eventlog;
  bool fileOutput_nodeInfo;
  bool fileOutput_edgeInfo;
  bool fileOutput_ratesInfo;
  bool fileOutput_virusSpread;

  // Console output options
  bool consoleOutput_simulationData;
  bool consoleOutput_eventlog;
  bool consoleOutput_simulationPause;

  // rates options
  double rates_arNode;
  double rates_arNode2;
  double rates_arEdge;
  double rates_arEdge2;
  double rates_swapEdge;
  double rates_infection;
  double rates_infection2;

  // Add/remove node rate options
  double arNodeRate_ratio;
  bool arNodeRate_dependent;
  std::string arNodeRate_type;
  double arNodeRate_addFactor;
  double arNodeRate_removeFactor;

  // Add/remove edge rate options
  bool arEdgeRate_changeStatus;
  double arEdgeRate_ratio;
  bool arEdgeRate_dependent;
  std::string arEdgeRate_type;
  double arEdgeRate_addFactor;
  double arEdgeRate_removeFactor;

  // swap edge rate options
  bool swapEdgeRate_changeStatus;
  std::string swapEdgeRate_type;
  double swapEdgeRate_swapFactor;

  // swap edge rate options
  double infectionRate_ratio;
  bool infectionRate_dependent;
  std::string infectionRate_infectionType;
  double infectionRate_infectionFactor;

public:
  Options()
  {
    // General options
    general_numberOfSimulations = 1;
    general_nBalanced = 500;
    general_nMax = 1000;
    general_nMin = 0;
    general_edgeFraction = 0.5;
```

```
    general_fixedSeed = 0;
    // Start of Simulation options
    startOfSimulation_n = 10;
    startOfSimulation_activeEdges = "balanced";
    // Stopping Criteria options
    stoppingCriteria_time = 10.0;
    stoppingCriteria_events = 10000;
    stoppingCriteria_virusExtinction = true;
    stoppingCriteria_timeOn = true;
    // File output options
    fileOutput_simulationData = true;
    fileOutput_eventlog = true;
    fileOutput_nodeInfo = false;
    fileOutput_edgeInfo = false;
    fileOutput_ratesInfo = false;
    fileOutput_virusSpread = true;
    // Console output options
    consoleOutput_simulationData = true;
    consoleOutput_eventlog = true;
    consoleOutput_simulationPause = false;
    // rates options
    rates_arNode = 1.0;
    rates_arNode2 = 0;
    rates_arEdge = 1.0;
    rates_arEdge2 = 0;
    rates_swapEdge = 1.0;
    rates_infection = 1.0;
    rates_infection2 = 0;
    // Add/remove node rate options
    arNodeRate_ratio = 1.0;
    arNodeRate_dependent = true;
    arNodeRate_type = "constant";
    arNodeRate_addFactor = 1.0;
    arNodeRate_removeFactor = 1.0;
    // Add/remove edge rate options
    arEdgeRate_changeStatus = true;
    arEdgeRate_ratio = 1.0;
    arEdgeRate_dependent = true;
    arEdgeRate_type = "constant";
    arEdgeRate_addFactor = 1.0;
    arEdgeRate_removeFactor = 1.0;
    // swap edge rate options
    swapEdgeRate_changeStatus = true;
    swapEdgeRate_type = "constant";
    swapEdgeRate_swapFactor = 1.0;
    // swap edge rate options
    infectionRate_ratio = 4.0;
    infectionRate_dependent = true;
    infectionRate_infectionType = "constant";
    infectionRate_infectionFactor = 0.2;
}

// General options
void SetGeneral_numberOfSimulations(int32_t input) { general_numberOfSimulations =
    input; }
void SetGeneral_nBalanced(int32_t input) { general_nBalanced = input; }
void SetGeneral_nMax(int32_t input) { general_nMax = input; }
void SetGeneral_nMin(int32_t input) { general_nMin = input; }
```

```
void SetGeneral_edgeFraction(double input) { general_edgeFraction = input; }
void SetGeneral_fixedSeed(int32_t input) { general_fixedSeed = input; }
// Start of Simulation options
void SetStartOfSimulation_n(int32_t input) { startOfSimulation_n = input; }
void SetStartOfSimulation_activeEdges(std::string input) {
    startOfSimulation_activeEdges = input; }
// Stopping Criteria options
void SetStoppingCriteria_time(double input) { stoppingCriteria_time = input; }
void SetStoppingCriteria_events(int32_t input) { stoppingCriteria_events = input; }
void SetStoppingCriteria_virusExtinction(bool input) {
    stoppingCriteria_virusExtinction = input; }
void SetStoppingCriteria_timeOn(bool input) { stoppingCriteria_timeOn = input; }
// File output options
void SetFileOutput_simulationData(bool input) { fileOutput_simulationData = input; }
void SetFileOutput_eventlog(bool input) { fileOutput_eventlog = input; }
void SetFileOutput_nodeInfo(bool input) { fileOutput_nodeInfo = input; }
void SetFileOutput_edgeInfo(bool input) { fileOutput_edgeInfo = input; }
void SetFileOutput_ratesInfo(bool input) { fileOutput_ratesInfo = input; }
void SetFileOutput_virusSpread(bool input) { fileOutput_virusSpread = input; }
// Console output options
void SetConsoleOutput_simulationData(bool input) { consoleOutput_simulationData =
    input; }
void SetConsoleOutput_eventlog(bool input) { consoleOutput_eventlog = input; }
void SetConsoleOutput_simulationPause(bool input) { consoleOutput_simulationPause =
    input; }
// rates options
void SetRates_arNode(double input) { rates_arNode = input; }
void SetRates_arNode2(double input) { rates_arNode2 = input; }
void SetRates_arEdge(double input) { rates_arEdge = input; }
void SetRates_arEdge2(double input) { rates_arEdge2 = input; }
void SetRates_swapEdge(double input) { rates_swapEdge = input; }
void SetRates_infection(double input) { rates_infection = input; }
void SetRates_infection2(double input) { rates_infection2 = input; }
// Add/remove node rate options
void SetArNodeRate_ratio(double input) { arNodeRate_ratio = input; }
void SetArNodeRate_dependent(bool input) { arNodeRate_dependent = input; }
void SetArNodeRate_type(std::string input) { arNodeRate_type = input; }
void SetArNodeRate_addFactor(double input) { arNodeRate_addFactor = input; }
void SetArNodeRate_removeFactor(double input) { arNodeRate_removeFactor = input; }
// Add/remove edge rate options
void SetArEdgeRate_changeStatus(bool input) { arEdgeRate_changeStatus = input; }
void SetArEdgeRate_ratio(double input) { arEdgeRate_ratio = input; }
void SetArEdgeRate_dependent(bool input) { arEdgeRate_dependent = input; }
void SetArEdgeRate_type(std::string input) { arEdgeRate_type = input; }
void SetArEdgeRate_addFactor(double input) { arEdgeRate_addFactor = input; }
void SetArEdgeRate_removeFactor(double input) { arEdgeRate_removeFactor = input; }
// swap edge rate options
void SetSwapEdgeRate_changeStatus(bool input) { swapEdgeRate_changeStatus = input; }
void SetSwapEdgeRate_type(std::string input) { swapEdgeRate_type = input; }
void SetSwapEdgeRate_swapFactor(double input) {swapEdgeRate_swapFactor = input; }
// swap edge rate options
void SetInfectionRate_ratio(double input) { infectionRate_ratio = input; }
void SetInfectionRate_dependent(bool input) { infectionRate_dependent = input; }
void SetInfectionRate_infectionType(std::string input) { infectionRate_infectionType
     = input; }
void SetInfectionRate_infectionFactor(double input) { infectionRate_infectionFactor
    = input; }
```

```
// General options
int32_t GetGeneral_numberOfSimulations() { return general_numberOfSimulations; }
int32_t GetGeneral_nBalanced() { return general_nBalanced; }
int32_t GetGeneral_nMax() { return general_nMax; }
int32_t GetGeneral_nMin() { return general_nMin; }
double GetGeneral_edgeFraction() { return general_edgeFraction; }
int32_t GetGeneral_fixedSeed() { return general_fixedSeed; }
// Start of Simulation options
int32_t GetStartOfSimulation_n() { return startOfSimulation_n; }
std::string GetStartOfSimulation_activeEdges() { return
    startOfSimulation_activeEdges; }
// Stopping Criteria options
double GetStoppingCriteria_time() { return stoppingCriteria_time; }
int32_t GetStoppingCriteria_events() { return stoppingCriteria_events; }
bool GetStoppingCriteria_virusExtinction() { return stoppingCriteria_virusExtinction
    ; }
bool GetStoppingCriteria_timeOn() { return stoppingCriteria_timeOn; }
// File output options
bool GetFileOutput_simulationData() { return fileOutput_simulationData; }
bool GetFileOutput_eventlog() { return fileOutput_eventlog; }
bool GetFileOutput_nodeInfo() { return fileOutput_nodeInfo; }
bool GetFileOutput_edgeInfo() { return fileOutput_edgeInfo; }
bool GetFileOutput_ratesInfo() { return fileOutput_ratesInfo; }
bool GetFileOutput_virusSpread() { return fileOutput_virusSpread; }
// Console output options
bool GetConsoleOutput_simulationData() { return consoleOutput_simulationData; }
bool GetConsoleOutput_eventlog() { return consoleOutput_eventlog; }
bool GetConsoleOutput_simulationPause() { return consoleOutput_simulationPause; }
// rates options
double GetRates_arNode() { return rates_arNode; }
double GetRates_arNode2() { return rates_arNode2; }
double GetRates_arEdge() { return rates_arEdge; }
double GetRates_arEdge2() { return rates_arEdge2; }
double GetRates_swapEdge() { return rates_swapEdge; }
double GetRates_infection() { return rates_infection; }
double GetRates_infection2() { return rates_infection2; }
// Add/remove node rate options
double GetArNodeRate_ratio() { return arNodeRate_ratio; }
bool GetArNodeRate_dependent() { return arNodeRate_dependent; }
std::string GetArNodeRate_type() { return arNodeRate_type; }
double GetArNodeRate_addFactor() { return arNodeRate_addFactor; }
double GetArNodeRate_removeFactor() { return arNodeRate_removeFactor; }
// Add/remove edge rate options
bool GetArEdgeRate_changeStatus() { return arEdgeRate_changeStatus; }
double GetArEdgeRate_ratio() { return arEdgeRate_ratio; }
bool GetArEdgeRate_dependent() { return arEdgeRate_dependent; }
std::string GetArEdgeRate_type() { return arEdgeRate_type; }
double GetArEdgeRate_addFactor() { return arEdgeRate_addFactor; }
double GetArEdgeRate_removeFactor() { return arEdgeRate_removeFactor; }
// swap edge rate options
bool GetSwapEdgeRate_changeStatus() { return swapEdgeRate_changeStatus; }
std::string GetSwapEdgeRate_type() { return swapEdgeRate_type; }
double GetSwapEdgeRate_swapFactor() { return swapEdgeRate_swapFactor; }
  // swap edge rate options
double GetInfectionRate_ratio() { return infectionRate_ratio; }
bool GetInfectionRate_dependent() { return infectionRate_dependent; }
std::string GetInfectionRate_infectionType() { return infectionRate_infectionType; }
```

```
double GetInfectionRate_infectionFactor() { return infectionRate_infectionFactor; }


void printInput()
{
  std::cout << "\nCurrent input:\n";

  std::cout << "General:\n";
  std::cout << "number of simulations: " << general_numberOfSimulations << "\n";
  std::cout << "n balanced: " << general_nBalanced << "\n";
  std::cout << "n max: " << general_nMax << "\n";
  std::cout << "n min: " << general_nMin << "\n";
  std::cout << "edge fraction: " << general_edgeFraction << "\n";
  std::cout << "fixed seed: " << general_fixedSeed << "\n\n";

  std::cout << "Start of simulation:\n";
  std::cout << "n: " << startOfSimulation_n << "\n";
  std::cout << "active edges: " << startOfSimulation_activeEdges << "\n\n";

  std::cout << "Stopping criteria:\n";
  std::cout << "time: " << stoppingCriteria_time << "\n";
  std::cout << "events: " << stoppingCriteria_events << "\n";
  if (stoppingCriteria_virusExtinction) std::cout << "virus extinction: true\n";
  else std::cout << "virus Extinction: false\n";
  if (stoppingCriteria_timeOn) std::cout << "time limit on: true\n\n";
  else std::cout << "time limit on: false\n\n";

  std::cout << "File output:\n";
  if (fileOutput_simulationData) std::cout << "simulation data: true\n";
  else std::cout << "simulation data: false\n";
  if (fileOutput_eventlog) std::cout << "eventlog: true\n";
  else std::cout << "eventlog: false\n";
  if (fileOutput_nodeInfo) std::cout << "node info: true\n";
  else std::cout << "node info: false\n";
  if (fileOutput_edgeInfo) std::cout << "edge info: true\n";
  else std::cout << "edge info: false\n";
  if (fileOutput_ratesInfo) std::cout << "rates info: true\n";
  else std::cout << "rates info: false\n";
  if (fileOutput_virusSpread) std::cout << "virus spread: true\n\n";
  else std::cout << "virus spread: false\n\n";

  std::cout << "Console output:\n";
  if (consoleOutput_simulationData) std::cout << "simulation data: true\n";
  else std::cout << "simulation data: false\n";
  if (consoleOutput_eventlog) std::cout << "eventlog: true\n";
  else std::cout << "eventlog: false\n";
  if (consoleOutput_simulationPause) std::cout << "simulation pause: true\n\n";
  else std::cout << "simulation pause: false\n\n";

  std::cout << "Rates:\n";
  std::cout << "add/remove node: " << rates_arNode << "\n";
  std::cout << "add/remove node 2: " << rates_arNode2 << "\n";
  std::cout << "add/remove edge: " << rates_arEdge << "\n";
  std::cout << "add/remove edge 2: " << rates_arEdge2 << "\n";
  std::cout << "swap edge: " << rates_swapEdge << "\n";
  std::cout << "infection/heal: " << rates_infection << "\n";
  std::cout << "infection/heal 2: " << rates_infection << "\n\n";
```

```
    std::cout << "Add/remove node rate:\n";
    std::cout << "ratio: " << arNodeRate_ratio << "\n";
    if (arNodeRate_dependent) std::cout << "dependent: true\n";
    else std::cout << "dependent: false\n";
    std::cout << "type: " << arNodeRate_type << "\n";
    std::cout << "add factor: " << arNodeRate_addFactor << "\n";
    std::cout << "remove factor: " << arNodeRate_removeFactor << "\n\n";

    std::cout << "Add/remove edge rate:\n";
    if (arEdgeRate_changeStatus) std::cout << "change status: true\n";
    else std::cout << "change status: false\n";
    std::cout << "ratio: " << arEdgeRate_ratio << "\n";
    if (arEdgeRate_dependent) std::cout << "dependent: true\n";
    else std::cout << "dependent: false\n";
    std::cout << "type: " << arEdgeRate_type << "\n";
    std::cout << "add factor: " << arEdgeRate_addFactor << "\n";
    std::cout << "remove factor: " << arEdgeRate_removeFactor << "\n\n";

    std::cout << "Swap edge rate:\n";
    if (swapEdgeRate_changeStatus) std::cout << "change status: true\n";
    else std::cout << "change status: false\n";
    std::cout << "type: " << swapEdgeRate_type << "\n";
    std::cout << "swap factor: " << swapEdgeRate_swapFactor << "\n\n";

    std::cout << "Infection rate:\n";
    std::cout << "ratio: " << infectionRate_ratio << "\n";
    if (infectionRate_dependent) std::cout << "dependent: true\n";
    else std::cout << "dependent: false\n";
    std::cout << "infection type: " << infectionRate_infectionType << "\n";
    std::cout << "infection factor: " << infectionRate_infectionFactor << "\n\n";
  }

};

extern Options options;
#endif
#pragma once
```

C.2.13 options.cpp:

```
// simulation.cpp : runs 1 complete simulation.
//

#include "stdafx.h"

//headers
#include "options.h"


Options options;
```

C.2.14 simulation.h:

```
#pragma once
#ifndef SIMULATION_H
#define SIMULATION_H

#include <cstdint>

#include "Node.h"
#include "datastructs.h"

int32_t simulation(NodeVector_t &nodeList, NodeVector_t &nodeListInactive,
    EdgeMatrix_t &edgeData);

#endif
```

C.2.15 event.h:

```
#ifndef ADDNODE_H
#define ADDNODE_H

// headers from compiler
#include <cstdint>
#include <vector>

// headers
#include "datastructs.h"
#include "Node.h"

void reserveShortlist();

Result addNode(NodeVector_t &nodeList, EdgeMatrix_t &edgeData);
Result removeNode(NodeVector_t &nodeList, NodeVector_t &nodeListInactive, EdgeMatrix_t
     &edgeData);
Result changeEdge(NodeVector_t &nodeList, EdgeMatrix_t &edgeData);
Result addEdge(NodeVector_t &nodeList, EdgeMatrix_t &edgeData);
Result removeEdge(NodeVector_t &nodeList, EdgeMatrix_t &edgeData);
Result swapEdgeStatus(NodeVector_t &nodeList, EdgeMatrix_t &edgeData);
Result swapEdge(NodeVector_t &nodeList, EdgeMatrix_t &edgeData);
Result infectNode(NodeVector_t &nodeList, EdgeMatrix_t &edgeData);
Result healNode(NodeVector_t &nodeList, EdgeMatrix_t &edgeData);

#endif
#pragma once
```

C.2.16 rng.h:

```
#ifndef RNG_H
#define RNG_H

// headers from compiler
#include <cstdint>

// headers
```

```
#include "datastructs.h"

void setRandomGenerator();
double getRandom01();
double getExponentialRandomNumber(double mean);
int32_t getUniformInteger(const int32_t &numberOfValues);
int32_t getUniformInteger(const std::vector<int32_t> &valueList);
int32_t getWeightedUniformReal(const RateVector_t &rates);

#endif
#pragma once
```

C.2.17 rates.h:

```
#ifndef RATES_H
#define RATES_H

// headers from compiler
#include <cstdint>

// headers
#include "datastructs.h"

void initiateRates(RateVector_t &rates);
void updateNodeRates(RateVector_t &rates);
void updateEdgeRates(RateVector_t &rates);
void updateVirusRates(RateVector_t &rates, const NodeVector_t &nodeList);

#endif
#pragma once
```

C.2.18 input.h:

```
#ifndef INPUT_H
#define INPUT_H

// headers from compiler
#include <cstdint>
#include <vector>

void createInput();
void readInput();


#endif
#pragma once
```

C.2.19 output.h:

```
#ifndef OUTPUT
#define OUTPUT

// headers from compiler
#include <cstdint>
#include <vector>
#include <iostream>
#include <fstream>

// headers
#include "datastructs.h"
#include "Node.h"

void removeOutput();
void openOutput();
void printOptions();
void printTitleResults();
void printTitle();
void printBaseRates(const RateVector_t &rates);
void printStartOfSimulation(const NodeVector_t &nodeList, const NodeVector_t &
    nodeListInactive, const EdgeMatrix_t &edgeData);
void printVirusSpread();
void printEdgeData(const EdgeMatrix_t &edgeData);
void printNodeData(const NodeVector_t &nodeList, const NodeVector_t &nodeListInactive)
    ;
void printRatesInfo(const RateVector_t rates);
void printEventLog(const Result result, const NodeVector_t nodeList, const RateType &
    eventType);
void printResults(const NodeVector_t &nodeList);
void printEndOfSimulation(const NodeVector_t &nodeList, const NodeVector_t &
    nodeListInactive, const EdgeMatrix_t &edgeData);

#endif
#pragma once
```