

A Declarative high-performance toolkit for modelling infectious diseases

DAAN SCHOUTEDEN

Promoter:

Prof. dr. Frank NEVEN

Supervisor:

Dr. Lander WILLEM

Co-promoter:

Prof. dr. Stijn VANSUMMEREN

A thesis submitted for the degree of
MSc Computer Science



Department of Computer Science
2020-2021

Contents

1	Introduction	1
2	Modelling infectious diseases	3
2.1	Concepts of infectious diseases	3
2.1.1	Transmission	4
2.1.2	Stages	4
2.1.3	Asymptomatic carrier	5
2.1.4	Immunity	6
2.1.5	Acute versus chronic	6
2.2	Mathematical models of infectious diseases	6
2.2.1	SIR model	7
2.2.2	SEIR model	8
2.2.3	Additional refinements	9
2.2.4	Deterministic and homogeneous	9
2.3	Computational model simulations	9
2.3.1	Individual-based models	9
2.3.2	Parameterization	10
3	Stride	12
3.1	Key aspects	12
3.1.1	Core concepts	12
3.1.2	Individuals	14
3.1.3	Contact pools	14
3.1.4	Configurations	16
3.2	Simulation	17
3.2.1	Initialisation	18
3.2.2	Health status update	19
3.2.3	Pool presences	19
3.2.4	Contact tracing	19
3.2.5	Contacts and transmissions	20
3.3	Data	25
3.3.1	Population	25
3.3.2	Contact pools	25
3.3.3	Contact matrix	30
3.4	Analysis	34
3.4.1	Parallelization	34

3.4.2	Code	34
3.4.3	Performance	36
3.5	First optimisation	38
3.5.1	Code architecture	42
3.5.2	Updating individuals	42
3.5.3	Infector	43
3.5.4	Solution	43
3.5.5	Results	44
4	Sampling	49
4.1	General idea	49
4.2	Preliminary insights	50
4.2.1	Infector runtimes	50
4.2.2	Reversed contact vector	53
4.3	Iterative intervals	54
4.3.1	Age intervals	54
4.3.2	Implementation	57
4.3.3	Correctness	57
4.3.4	Performance	59
4.4	Sampling with iteration	65
4.4.1	Implementation	65
4.4.2	Correctness	66
4.4.3	Performance	66
4.4.4	Observations	75
4.5	Full sampling	79
4.5.1	Implementation	80
4.5.2	Correctness	81
4.5.3	Performance	81
4.6	Adjusted full sampling	85
4.6.1	Correctness	90
4.6.2	Performance	90
4.7	Inf-to-sus	93
4.7.1	Double iterative intervals	95
4.7.2	General sampling	96
4.7.3	General sampling contacts	98
4.8	Summary	102
5	Domain specific language	105
6	Conclusion	106
7	Future work	107
	Bibliography	111
	A1 Nederlandse samenvatting	112

A2 Configurations	113
A3 Sampling same interval	115

Chapter 1

Introduction

When a new or already known disease emerges, it is crucial to know how it can be controlled and combated in order to save lives. The knowledge of these diseases is not something that is gained overnight. Extensive research, which consists of gathering and processing data over the course of years or even decades, is needed to completely understand the ins and outs of a disease. The data that has to be gathered and the methods that will be used for the processing are different depending on the disease. Today's technology offers a lot of possibilities for the collection of data with among other things e-questionnaires [1] and IoT-based data collection systems [2]. Likewise, there are lots of different processing options on the gathered data depending on what has to be achieved, with for example machine learning which can be used in cancer prognoses and predictions [3]. In general, diseases can be grouped in two different types: infectious and non-infectious. Next to finding a cure for either type, it is also of interest to know how infectious diseases spread and behave under certain circumstances. This information is necessary to get a better understanding of diseases or make predictions on how they will spread. For this type of research, computer models have been created to simulate the diseases' behaviours and how they roam through populations. They play a major role in disease outbreaks, such as the COVID-19 pandemic, by providing essential information and insights to governments and the World Health Organization (WHO) in their policy decision making [4]. Since time is of the essence when these policy decisions have to be made, the speed at which these models generate information is of great importance. This thesis aims to improve such a model, namely Stride, an individual-based simulator for the transmission of infectious diseases [5]. The first goal is to analyse the Stride code and investigate ways to increase its performance, thus fulfilling the need for fast information retrieval. When policy decisions are being considered, a multitude of scenarios are simulated on the models which may require them to run in different ways. Therefore, the other goal of this thesis is to increase the flexibility and extensibility of Stride by creating a domain specific language (DSL). The intent of this DSL is to make Stride's functionalities easier and more transparent to use on a more abstract level.

To start off this thesis, the key concepts of infectious diseases, how models work, and how these models can be used on those diseases is explained in Chapter 2. Then, once a general grasp is obtained on this material, the focus shifts to Stride. Chapter 3 examines the design and workings of Stride along with the data that is used. In Chapter 4 the

major optimisation of this thesis, which is through a sampling approach, is explained and evaluated. The DSL previously talked about and the process of creating it is explained in Chapter 5.

Chapter 2

Modelling infectious diseases

The purpose of this chapter is to get a grasp of how infectious diseases can be modelled to obtain relevant information through simulations. Therefore, a good understanding of infectious diseases and how they can spread is needed, and thus explained first. Then, the architectural design behind such models is described with regard to how Stride is designed. It must be stated that this chapter's intent is solely to present information collected from academic publications and literature that is considered relevant for this thesis. It does therefore not contain any sort of new personal research, neither is it an examination of the given content. The content of this chapter is based on [6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24].

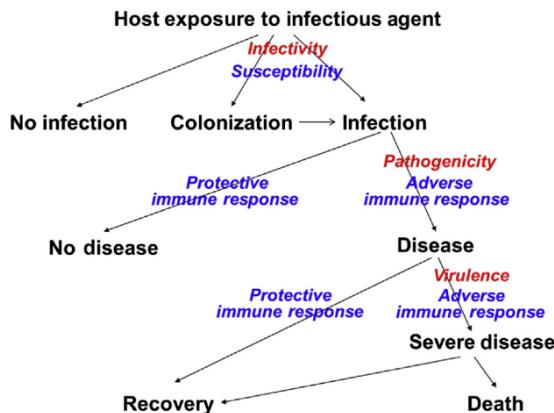


Figure 2.1: Potential outcomes of a host exposed to a pathogen (infectious agent) with depending factors of progression between stages. Figure copied from [6].

2.1 Concepts of infectious diseases

An infectious disease is an illness caused by any sort of pathogen, including bacteria and viruses, which is transferred from one host to another. Transmission of pathogens can occur in various ways, which is explained in Section 2.1.1, but does not necessarily mean that the new host will become ill. Figure 2.1 illustrates that exposure of a host to a

pathogen can result in different outcomes. These outcomes are dependent on numerous factors such as the host's susceptibility to infection and disease, environmental factors, and physical and social behaviour of the host. If a new pathogen reaches a new host, it starts to colonize: multiplication of the pathogen at an entry point such as the skin or the mucous membranes of the respiratory, digestive, or urogenital tract. Then the pathogen starts to invade and establish within the host's tissue and causes disruption within a host, but does not necessarily result in disease. Disease only occurs when a certain level of disruption has been reached which translates into symptoms and signs of illness. The final outcome of the disease varies in every situation, in which the immune system of the host plays a major role in. Depending on the host's immune system and the disease, the host may even become immune to the disease [6].

2.1.1 Transmission

An important trait of many infectious diseases is that an infected host can act as a source of exposure to others. As will be explained in Chapter 3, Stride focuses primarily on diseases such as measles, influenza, and COVID-19 which are mainly transmitted through means of contact. This thesis will thus concentrate on transmission through contact. For the sake of completeness however, we mention that there are five distinct means of transmission [7]:

1. Through contact, direct (skin or sexual contact) or indirect (infected fomite, blood or body fluid).
2. Ingesting contaminated food.
3. Inhaling contaminated air.
4. Through vectors (mosquito, tick, etc.).
5. Through pregnancy.

2.1.2 Stages

When a transmission of a pathogen has occurred, the exposed individual can become ill as discussed earlier. This does not mean that the recently infected person immediately becomes infectious to others or starts showing symptoms. Infectious diseases have different stages and the duration of those stages is different depending on the person and the type of infection, among other things. The time from exposure until the time of the first signs of symptoms of disease is called the *incubation period*. After the incubation period, the *period of clinical illness* starts which describes the duration between the first and last signs or symptoms. These periods are used to describe how the disease manifests itself to the infected person, but they do not say anything about the infectiousness of the infected person. The term to describe the time from exposure until the onset of infectiousness is the *latent period*. The latent period is followed by the *infectious period* which is the duration when the infected person can transmit pathogen. Figure 2.2 visualises these periods for different types of diseases. It should be noted that these periods are an indication and that they can vary from person to person. [6]

The most obvious pattern of these stages is when the incubating and latent period are the same, as well as the clinical ill and infectious period. An example of such a disease is Ebola, and is shown in Figure 2.2(a). Diseases where these patterns nicely follow up on each other are relative easy to treat and predict individually. Unfortunately there are also diseases where this is not the case.

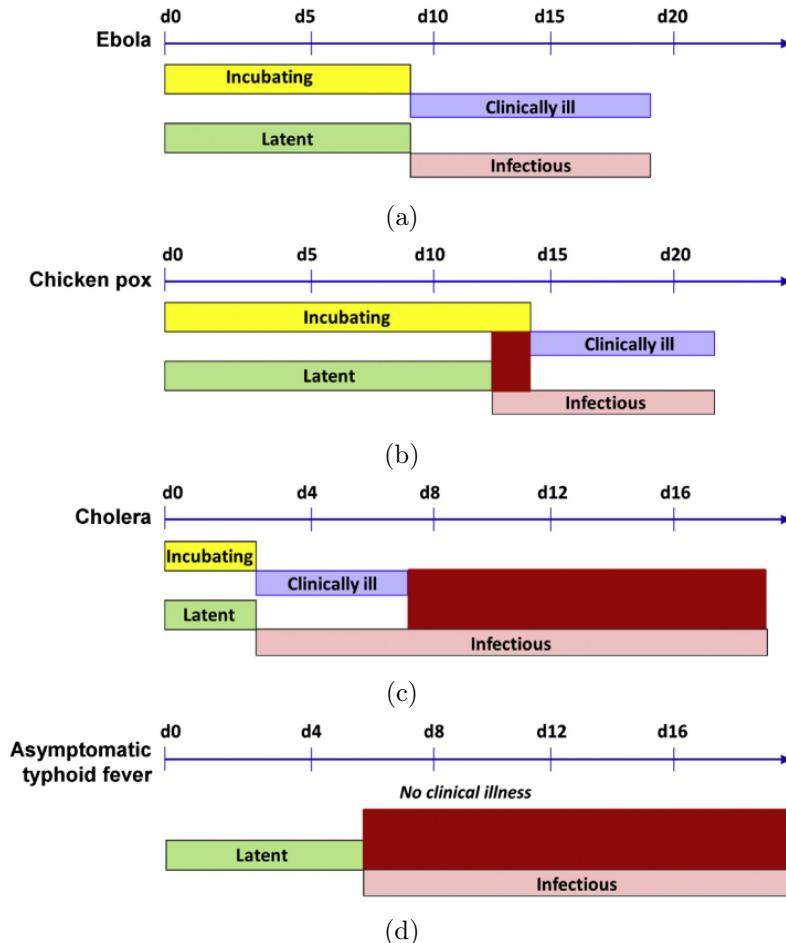


Figure 2.2: Stages of infectious diseases for different types of disease in function of the time since exposure (in days). The red bar displays when an infectious person is asymptomatic. Figure copied from [6].

2.1.3 Asymptomatic carrier

It is possible that a person is infectious to others, but yet shows no signs or symptoms of disease. Someone who satisfies these conditions is called an *asymptomatic carrier*, which can be divided in three different types of carrier [6]:

1. **Incubatory:** This occurs when the incubation period overlaps with the infectious period and is for example the case with chicken pox, shown in Figure 2.2(b).

2. **Convalescent:** In this case the clinically ill period ends before the infectious period. Cholera is such a disease and is visualised in Figure 2.2(c).
3. **Healthy:** This is the case when someone has no signs or symptoms, but still can infect others like with asymptomatic typhoid fever shown in Figure 2.2(d).

It is clear that asymptomatic carriers make it harder to contain a disease. When treating infectious diseases it is therefore important to know when the different stages appear in order to take the appropriate actions such as contact tracing and isolation [8].

2.1.4 Immunity

Prevention of a disease can be more important than a cure, especially when trying to control an outbreak. Being immune to a disease therefore offers the best solution in prevention from getting ill. The actual workings of immunity will not be explained as they do not add value to this thesis. The different ways to become immune, however, deserve some elaboration to give insight in how this can effect the modelling of infectious diseases.

The first case is called *innate* immunity, which describes the immunity someone is born with [9]. The second case is *adaptive* or *acquired* immunity, and is used to denote the cases of immunity that are gained after being born. Once born, immunity can be acquired in four different ways as shown in Figure 2.3. A distinction can be made between *active* and *passive*, and between *natural* and *artificial* immunity. Active immunity is gained when exposure to a pathogen triggers the immune system to produce antibodies for that pathogen, while passive immunity is gained when someone receives antibodies without being exposed to a pathogen. Active immunity can occur in a natural way when infected by a pathogen, or in an artificial manner when vaccinated. Likewise, passive immunity can happen in a natural way by receiving antibodies through breastfeeding or the placenta, or in an artificial way through globulin [11].

2.1.5 Acute versus chronic

The last thing that has to be discussed about infectious diseases is the different types of infection. The two most used terms to describe infections are *acute* and *chronic infections* and their key distinction is the duration of the infection. An acute infection lasts no more than six months, whilst a chronic infection lasts longer than six months. Also, the symptoms of acute infections develop rapidly over the course of days, in contrast to chronic infections where symptoms develop over weeks or months. Measles and influenza, the diseases Stride focuses on, are caused by acute infections. Therefore, whenever infectious diseases are mentioned in this thesis, they refer to those that are cause by acute infections [12].

2.2 Mathematical models of infectious diseases

A model is used as a representation of something and can be used by anyone in any field, with for example an architect who can build a scale model to represent their idea of a

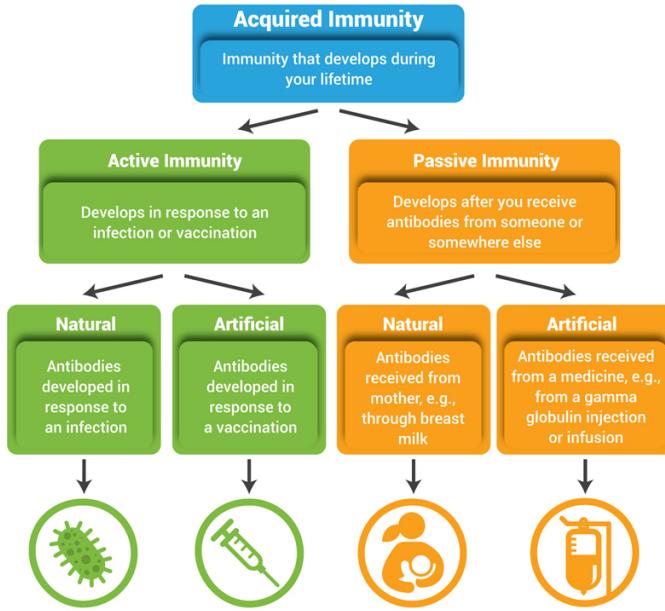


Figure 2.3: A visualization of the different types of acquired immunity. Figure copied from [10].

building. The architect's scale model can be just four walls and a roof, or the architect may have put incredible attention to detail in his work with by decorating every room. The difference in both models lies in the amount of work done by the architect. Likewise, a model to simulate infectious diseases can be a fast high-level program that produces little information, or a very complex program that needs a substantial amount of time to calculate detailed information of every individual. The next step in building such a model, is thus to understand how the recently gathered knowledge about infectious diseases can be applied and what trade-offs have to be made to get the desired results.

2.2.1 SIR model

Section 2.1.5 explained that the focus lies on infectious diseases caused by acute infections, and thus the different stages of infectious diseases from Section 2.1.2 are short periods of time. Section 2.1.1 also explained that the only type of transmission that will be considered is through contact. With this information, the most basic transmission model for a transmitted infectious disease can be used, called the SIR model which refers to its three compartments: susceptible, infected, and recovered. It is a system that consists of three coupled non-linear ordinary differential equations. The model makes use of very strict assumptions and is therefore not an accurate representation of the real world. There are numerous variations on the SIR model with different assumptions, but the basic assumptions are the following:

- A large, fixed population represented as one large group, so without subgroups, where no demographic events occur.

- The infection has no latent period, thus an individual becomes infectious as soon as they become infected.
- Once someone is recovered, they are forever immune.
- Every person can encounter every other person with the same probability each day (which has nothing to do with probabilities, but rather with an intuitive model).
- The outbreak is short lived.

Although the model makes these strong assumptions, it still provides a lot of value. With simple calculus a great deal of information can be extracted such as the reproductive number of the disease, the rate of recovery, and the rate of infection [13].

2.2.2 SEIR model

An aspect of the SIR model is that there is no latent period. However, Sections 2.1.2 and 2.1.3 showed that the latent period is an important part of infectious diseases. In order for a model to give a more accurate representation of infectious diseases, it thus needs to take this period into account. The SEIR model is a refinement of the SIR model that fulfills this need by adding the *exposed* state between the susceptible and infectious states. The exposed state represents the latent period when someone has been exposed to pathogen, but in which the amount of pathogen is too low in order for the host to infect others. Figure 2.4 illustrates how the exposed state gives the SEIR model a more accurate representation of infectious diseases [14].

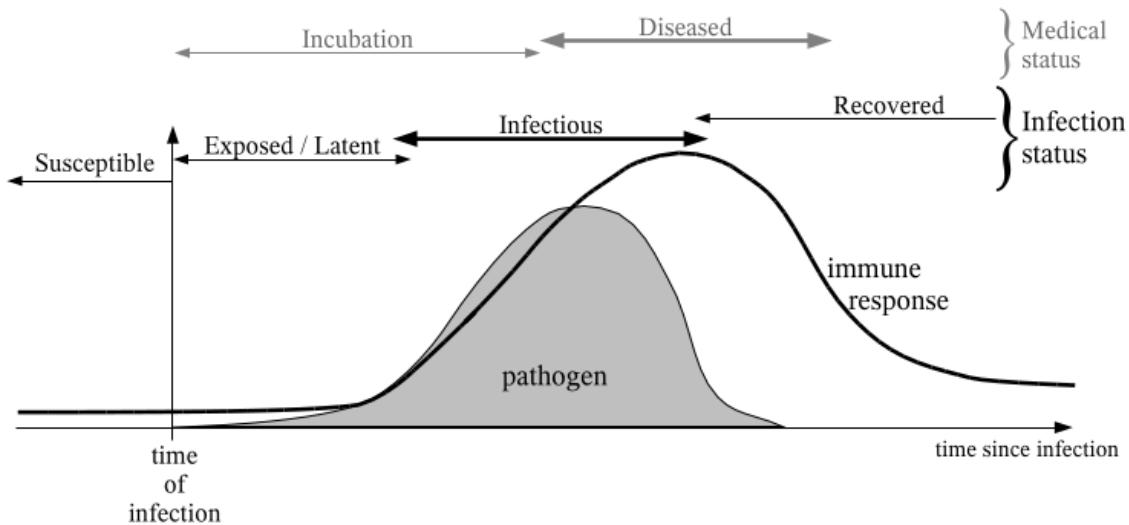


Figure 2.4: A generalized representation of an infectious disease over time, showing the different stages related to the dynamics of the pathogen and the host's immune response. Figure copied from [14].

2.2.3 Additional refinements

The SEIR model is more advanced than the SIR model, but it is far from perfect. Section 2.1.4 explained how immunity can be achieved in different ways. Further refinement of these models would thus be to incorporate the possibility of people being born with immunity. Since people do not necessarily become immune to a disease once they have recovered from it, another refinement would be to add the possibility of people becoming susceptible instead of immune after recovery. A third refinement could satisfy the lack of asymptomatic carriers, which is explained in Section 2.1.3, by adding the possibility of people being an asymptomatic carrier [14].

2.2.4 Deterministic and homogeneous

The models that have been explained could be seen as deterministic models. Given the same starting conditions, they would produce the same results every time. These models are limited to uniform and homogeneous populations, but still provide useful insights in infectious diseases [15]. However, their resemblance of the real world lacks the structure and stochasticity of every day life. In real life, people are divided in sub-populations such as their household, workplace, and school, where they meet different people every day. Every decision someone makes can have an impact on what their next decision will be or even what someone else their decisions will be. In order to simulate real life, different decision outcomes need to be taken into account. The simulation of such a decision can be seen as the collection of every possible outcome, for which every outcome has a probability that it will happen. This role of chance has a big impact when the number of infected people is relatively small or when considering a small (sub-)population [16].

2.3 Computational model simulations

Every simulation serves a certain purpose, and so computational models can be designed in different ways to serve different purposes. This last section of the introductory chapter explains, based on Stride, how a computational model can be build to simulate infectious diseases.

2.3.1 Individual-based models

Section 2.2.4 explained how the mathematical models give a more high-level representation of infectious diseases in a population. One of their concepts is that they make use of homogeneous populations, which states that everyone in the population is and acts completely the same. Their results show the transmission dynamics based on the amount of people in every compartment (susceptible, infected and recovered), which only tells something on a population-level [17]. These results thus do not tell anything about the individuals that get infected or who someone has contacts with. If such more detailed results are required, a different model design is needed.

These details cannot be gathered from a homogeneous population, because everyone is then the same and no distinctions can be made. A heterogeneous population is needed

which is when there are distinct individuals each with their own 'life'. These distinctions can be made based on age, places they visit, people they meet, and so on. Every individual has thus their own identity and information that needs to be captured as well as the interactions with everyone else. A model that fulfills these needs is an individual-based model of which the definition is given by Willem et al. [18, 19]:

"A computer simulation for the creation, disappearance and movement of a finite collection of interacting individuals or agents with unique attributes regarding spatial location, physiological traits and/or social behaviour."

In an individual-based model the population is designed in a bottom-up fashion. The population-level information is now the result of all the interactions between the individuals and their 'behaviour' [18]. Behaviour of individuals can also change based on for example different days of the week (weekday and weekend), or when feeling ill. It also allows for different environments to be created where every environment has its own type of behaviour [20].

Example 1. When at work, people interact very different than when they are at home. When at home, the contacts between people are closer than the contacts at work. As such, individuals in different environments will 'act' in different ways.

These environments also give the possibility to incorporate different stochastic interactions for each environment, which is very valuable when looking smaller populations as was stated in Section 2.2.4.

Example 2. An infectious person has a much higher probability of infecting members of their household than colleagues at work.

Additionally, these models can give the option to track every individual in as much detail as desired. This gives the possibility to create a 'history' for every individual as well as their network of contacts, thus providing new information in understanding different phenomena [18].

Overall, as stated by Murphy et al. [20], these models can be of use in countless situations to provide insights into the adaptation and function of systems. Another important asset of them is that they grant these insights when access to field data is dangerous, unavailable, or impossible to collect, with for example the COVID-19 pandemic that made collecting field data extremely difficult. Furthermore, the world of individual-based models is a growing community where open-source information and programs play an important role. This, together with the increasing computational power, ensures future improvements in this field [20, 21].

2.3.2 Parameterization

Next to creating a model, parameters that are passed to the simulation need to be established. It is obvious that parameters can greatly impact the results of a simulation and thus definitely need to be chosen carefully:

Example 3. Looking back at Examples 1 and 2, the different probabilities in every environment need to be defined with actual values. If the probability that someone is infected is slightly higher or lower can result in respectively more or less infected people in a simulation.

There are numerous ways to gather the required data with open-source data proving a valuable asset in these situations. In some cases it might be challenging because certain interactions may not exist or still need some research [22]. When parameters need to be estimated, there is a level of uncertainty that needs to be accounted for. Various techniques have been presented to give more accurate estimations on parameters [25, 23], with a more relevant study about transmission model parameters for seasonal influenza by Willem et al. [24]. What parameters and data are used in Stride will be seen in the next chapter.

Chapter 3

Stride

This chapter describes the main subject of this thesis, which is the individual-based simulator for the transmission of infectious diseases or Stride for short. Before diving into pseudo code, we will explain the reasoning behind it and discuss the most important concepts. Furthermore, we lay out and examine its workings, as well as the data that is being used.

3.1 Key aspects

Before diving into the details and code of Stride, we need a better understanding of the model. This is gained by first figuring out what the most important concepts of Stride were when it was designed, followed by the two most important aspects, namely the individuals and the contact pools. Lastly, the configurations will be discussed since they determine how the model should be run.

3.1.1 Core concepts

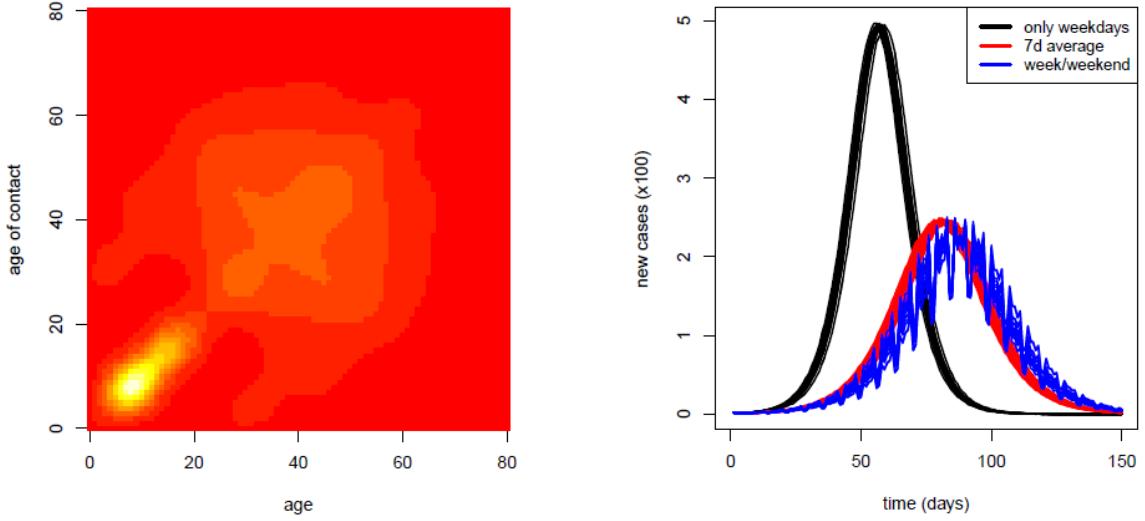
Computer models have proven to provide valuable information in the spread of infectious diseases and what results of countermeasures can be [4]. The paper in which Stride is presented by Elise Kuylen et al. [5] states the following:

"Several individual-based models for the transmission of infectious diseases, that take into account heterogeneous social mixing, have already been proposed, but only a few of these, such as FluTE [26] and FRED [27] are publicly available as open-source projects."

The way they describe Stride is as follows:

"An open-source individual-based model for infectious disease transmission, which explicitly takes into account age, type of day and context when simulating social contacts."

Stride's main goal is thus to give a more accurate representation of transmission of infectious diseases by making the simulations resemble more closely to real life. The way Stride does this is by heavily focusing on the social contacts and the different aspects that influence these, since social contacts are crucial for infectious disease transmission [5].



(a) Age-mixing patterns in which yellow indicates high contact rates and red low contact rates.

(b) Number of influenza cases over time using different scenarios regarding the type of days.

Figure 3.1: Stride's social mixing results, copied from [5].

Age

The first key aspect of social contacts in Stride is the age. The model works on a heterogeneous population in which every individual is different, including their age. In calculating the contacts between people, age has a big impact on who someone meets. Figure 3.1(a) shows Stride's resulting age-mixing pattern, which is a representation of the different contact rates based on age. This may seem odd at first, but the explanation behind the results is that people are mainly mixing with people of the same age. The diagonal line between the bottom-left and upper-right corner represents the contact rates between people of the same age, which also has the highest contact rates in each row and column. The bottom left yellow area is the result of children having contact with their peers at school, which also goes for the yellow area in the middle that represents people meeting with their colleagues. It is thus clear that age is a major influential factor in who an individual meets.

Type of day

Another important aspect Stride takes into account is the different type of days. A simulation in which every day is the same, does of course not give a good representation of every day life. Stride allows therefore for different type of days to be played out, such as weekdays, weekend days, and holidays. That fact that this is an important feature is confirmed in Figure 3.1(b), where the total number of infected cases is much higher when using only weekdays instead of different types of days. Likewise, the peak of number of new cases happens a lot earlier when using only weekdays and is significantly higher.

Social context

The last major aspect is the context of a social contact. Just as the age and the type of day, this is yet another very intuitive social trait. The context of contacts in Stride is viewed as the places where people can meet. Generally speaking, when in a household, the percentage of people one will have contact with is a lot higher than when at work regardless of age. How Stride handles these different social contexts will become clear during this chapter.

3.1.2 Individuals

Since Stride is an individual-based model, the most important parts are the individuals. Each person is defined by its own characteristics such as their age and health. A person's health status is based on the SEIR model, which is discussed in Section 2.2.2, and can be either susceptible, exposed, infectious, recovered or vaccinated/immunized. Section 2.1 explained how people can have different reactions on infections, hence it is a rather important feature for a model to take into account when emulating real life. Therefore, every person's health has its own durations for the different stages of infectious diseases from Section 2.1.2. How, when, and where individuals can have contact with one another depends on the contact pools in which each person can be present.

3.1.3 Contact pools

A person can be a member of four different *contact pools*. These pools represent the different contexts in which an individual can have contact with someone. People are only able to have contact with members of the same pool who are present at the same time. These types of pools are the household, school or workplace, and two communities. An overview of the pools of an individual is visualised in Figure 3.2, which also shows the days when someone can be present in each pool. An important rule is that an individual can only be a part of one pool per pool type.

Household

The most straightforward contact pool is the household. Everyone is a member of their own household and will be present in it every day of the week. There is however one exception to this rule, which is when someone is quarantined and is explained in more detail in Section 3.2.3.

School and work

Of the four types of pools someone can be a part of, there is one type which is not the same for everyone and that is the contact pool that represents an average individual's daytime occupation on a weekday. Just like in real life, what people do during a regular weekday is different for every person: children go to school and adults go to work. Following this logic, individuals younger than 18 years old should be a member of a school-pool and those 18 years and older of a work-pool. This might already be a solid representation for a model, but Stride makes a few extra distinctions here:

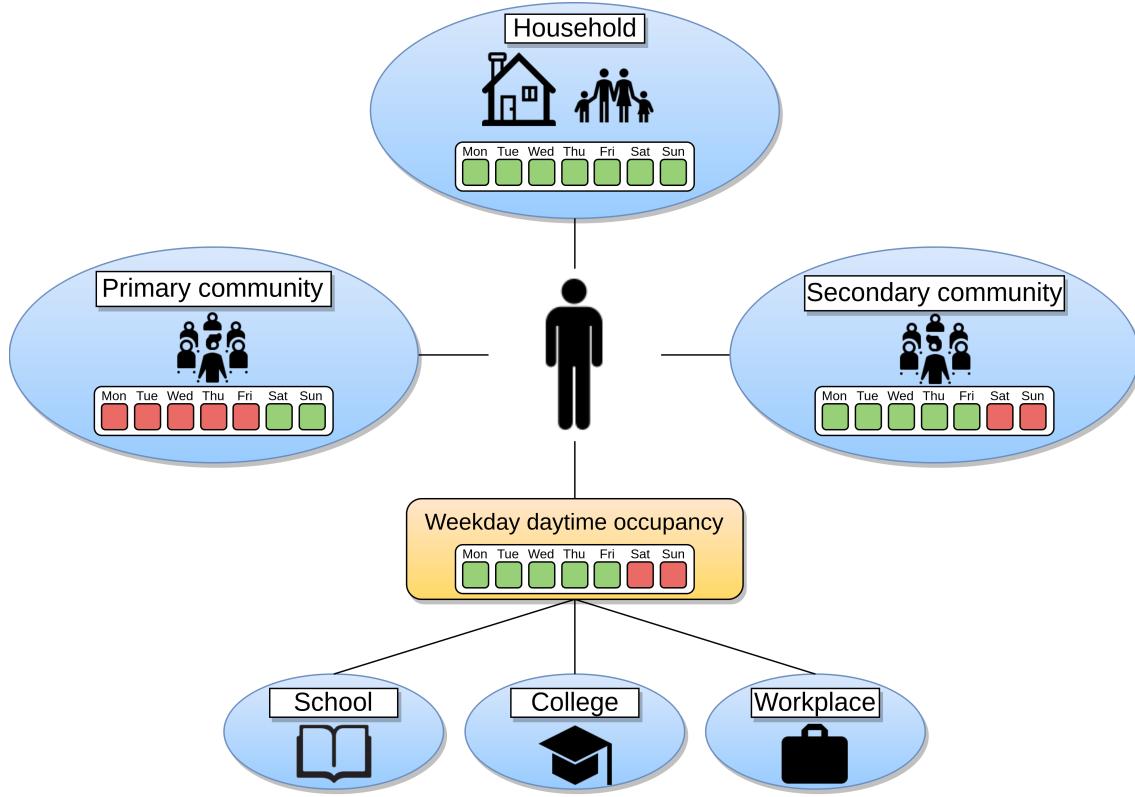


Figure 3.2: Overview of the different type of contact pools. The days colored in green and red in each pool respectively show when an individual can be present or not.

- Children, thus people younger than 18 years old, do have a school contact pool, but these pools do not only consist of children. Schools also need teachers who are of course adults. These teacher individuals are therefore not a part of a workplace contact pool, but instead of a school contact pool.
- Another distinction that can be made is the different types of school. This can be done in various ways, but Stride only separates K-12 schools and colleges. K-12 schools describe all the different types of education up until the age of 18 and colleges only contain (young) adults. Obviously, the school contact pools of the teachers and professors also make this distinction since they are in the same pool as their students.
- The workplace is the only pool of which an individual is not required to be a part of. A large part of the society in real life does not have a job or is retired. The individuals portraying these people are thus only a member of three contact pools.

The days when people can be present in their school or workplace pool are the weekdays from Monday to Friday, so weekend jobs do not exist in the simulation. However, as Section 3.1.1 explained that Stride takes different types of days into consideration, the holidays cause for people to skip their school or workplace pools on these days. People who feel ill also typically don't go to school or their work, so they might then also not appear in their contact pool as will be seen in Section 3.2.3.

Communities

The last types of contact pools are the communities, of which there are two: the primary and secondary community. These pools represent the times when someone is not at school or work, and not at home. They can be described as the places and people someone meets in their spare time such as when practising a sport or other hobbies. Those pools represent the same concept, but do not occur at the same time or consist of the same people. The primary community is used for the contacts that are being made in the weekend and on holidays thus people can only be present in them on Saturday, Sunday and on holidays. Correspondingly, the secondary community represents the people someone can meet on a regular weekday outside of their school or workplace pool. Just as someone who feels ill will not go to school or work, they can also decide to stay at home and not attend their primary and secondary community.

3.1.4 Configurations

The previously mentioned aspects give more information about the model itself. How the model must be used in a simulation is decided by the various customisable configurations. Covering every single configuration would be a little otiose, so only the ones that are considered relevant will be explained.

Population

Sections 3.1.2 and 3.1.3 showed how individuals are being represented and how they are divided in contact pools where they can have contact with each other. These individuals and their contact pools are configured in a population file which contains a row for every individual. Such a row consists of non-negative integers representing the individual's age and the ID of every pool type as follows:

(age, household, school, work, primary_community, secondary_community)

Since not everyone has to be a member of a school or workplace contact pool, it is possible for these values to be zero, indicating that they do not belong in one.

Holidays

As stated in Section 3.1.1, Stride makes a distinction between the type of days. Next to weekdays and weekend days, holidays are also being accounted for when deciding what kind of day the model has to simulate. These holidays are presented in a file that contains the information of all the days that have to be treated different than usual. This can be as detailed as needed and allows for distinctions to be made. There are for example days that are a national holiday and thus result in everyone not going to school or work (and their secondary community), but there are also holidays that only apply to schools such as summer break so that adults (not teachers) still need to go to work. Another similar example in Belgium is that colleges start somewhere mid September while K-12 schools start the first of September. In the holidays configuration file this is made possible by defining the ages to which a holiday applies.

Disease characteristics

Because the purpose of Stride is to model multiple infectious diseases, it needs a disease configuration file which describes the characteristics of the disease. This gives the possibility to simulate any disease on the basis of a couple traits. These traits consist of the basic reproduction number¹, the probabilities of being symptomatic or asymptomatic, how long someone can be infectious, etc.

Event logging

Subsequent sections will explain how Stride gives the possibility to run faster at the expense of generating less information. The main rule that defines this behaviour is the event logging rule. It determines what type of events need to be logged, such as the transmissions and contacts. The reason for this to have such an impact is that when contacts need to be logged, the model needs to calculate every possible contact. If only transmissions need to be logged, it suffices to only calculate the transmissions instead of every possible contact. How this effects the model will become clear in Section 3.2.5.

Age contact matrix

By now it is known how individuals and their contact pools are presented. These pools represent different social contexts in which people can have contact with each other. How people interact with one another in every pool is given by the age contact matrix file. This file contains a matrix with the probabilities of two people having contact with respect to their age and the pool type in which they find themselves in. Passing this matrix on to the simulation instead of defining the probabilities in the code, allows for flexible and customisable usage. The details of how this matrix is being used is described in Section 3.3.3.

Numerical values

At last, there are various configuration values that are plain numerals. For example the number of days that the model has to simulate, the date at which the simulation has to start, or the number of infected and immune people at the start. There is also a possibility for multi-threading by configuring the number of threads that have to be used, which will be talked about further along this chapter.

3.2 Simulation

The next step in gaining insight into Stride is to figure out the simulation. This section will thoroughly explain how the model works and every important part will be covered in detail. A conceptual overview of the simulation flow with its most important components is illustrated in Figure 3.3. Stride begins by initialising the simulation, which is explained in Section 3.2.1. Then the simulation commences at the starting date, moving forward in

¹Simplified explanation: the average number of people someone infects, which describes the infectiousness of a disease.

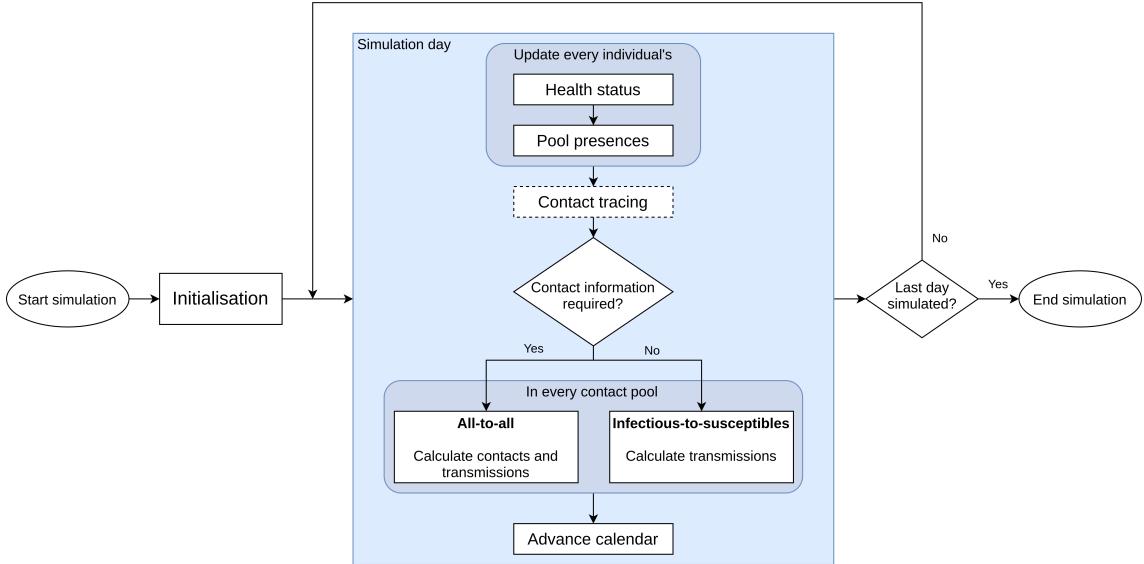


Figure 3.3: Conceptual overview of a Stride simulation.

discrete time-steps of one day until the configured number of days have been processed. All of these days consist of the same pattern which is the following:

1. Update the health status of every individual (Section 3.2.2).
2. While updating everyone's health, also determine in which pool they will be present (Section 3.2.3).
3. There is a possibility to perform contact tracing if this is set to be active (Section 3.2.4).
4. Make calculations to determine who has contact with each other and who gets infected (Section 3.2.5).

There is an additional step, universal testing, which will not be talked about in this thesis, because it was added after the start of the thesis and thus has not been examined. This testing step is similar to the contact tracing step, which is a very small part of the simulation that comes right after the contact tracing step and, as will be discussed in Section 3.4.3, is therefore insignificant.

3.2.1 Initialisation

Obviously, the initial step of the model is the initialisation. It uses the configurations from Section 3.1.4 to set up all the necessary segments before starting the simulation. The first segment that gets initialised is the random number generator manager, which handles all the random numbers and distributions. Next is the population that gets created by generating every individual one by one from the population file, followed by the construction of all the contact pools. Then the calendar is built with the holidays file and the given start date, after which most of the numerical configuration values get processed. Thereafter are

the contact matrix and disease characteristics read in and stored.

The last step in the initialisation process is to 'seed' all the random elements of the simulation. Individuals need to behave in different ways, not only regarding contacts, but also regarding the disease. Every individual's health therefore gets randomly created by using uniform distributions to determine how long the different stages of the disease will be, which are discussed in Section 2.1.2, and whether they will be symptomatic or not. The other seeding that gets done is randomly selecting the individuals that will be infected or immune at the start of the simulation.

3.2.2 Health status update

When an individual has not been infected, this step is skipped because their health status does not change then. If someone is infected, however, the simulator will look at their health characteristics that were randomly assigned in the initialisation. On the basis of these characteristics, it determines how the person's health will be in the upcoming day. The different outcomes are that the person becomes or remains infectious, symptomatic, infectious and symptomatic, or that the infection stops.

3.2.3 Pool presences

We now know that individuals belong to a pool and can be present or not, but not how this is stored by our model. Every pool is represented by its unique Id, the contact pool type, and its members. It does not keep track of which members are present during the current day. If someone is present or not is only being stored by the person itself. Thus, after a person's health status is updated, their presence in every pool gets evaluated.

There are various factors that affect in which pool someone will be present. The most straightforward one is what type of day the current day is, which can be a regular weekday, weekend day, or holiday. When a person is symptomatic and thus feels ill, there is a chance that they will not attend other contact pools except their household.

Furthermore, there are factors which are not a part of the standard model, but they are additional features that can be passed on through the configurations. A possible feature is to close schools and workplaces when, for example, the WHO wants to test if this will help to reduce an outbreak. Another element that can be turned on is the ability for a person to work from home and the ability for the 'government' of the population to enforce telework. The last non-standard factor is the possibility of quarantine, which translates in the person isolating themselves from every pool, with also being able to be isolated from their household and thus not be present in it.

3.2.4 Contact tracing

If the configurations have indicated that on the current day there will not be contacts traced, then this step is skipped, which also goes for universal testing. For the sake of completeness, this step will only be briefly explained here, because it does not belong to

the standard simulation flow. Even when it is incorporated in the simulation, it does not have a big impact on the run time as will be seen in Section 3.4.3.

The contact tracing step will iterate over the entire population and check who is infected and just recently 'felt' symptoms in order to perform the contact tracing, because people who do not feel symptoms would normally not know they are infected. Infected individuals their contacts of the previous days have a probability that they will be traced, which can result in them being put into quarantine. This probability resembles the fact that people in real life do not perfectly know who they have been in contact with and is a numerical configuration value.

3.2.5 Contacts and transmissions

The final and most important part of a simulated day is calculating every contact and transmission. This is done by iterating over the different pool types to see which pool is active on the current day, so for example the workplace, K-12 school, college, and secondary community contact pools get skipped when it is a national holiday. Then, all the pools of an active pool type are iterated over and get processed to determine the contacts and transmission. There are two algorithms that can be used to do these calculations, for which the choice is made at run time, depending on the need for information about contacts. If information about contacts is needed, the *all-to-all* algorithm will be selected. When the simulation only needs to handle transmissions and infections, the *infectious-to-susceptibles* algorithm is used.

All-to-all

The first algorithm, of which the pseudo code is shown in Algorithm 1, compares everyone of a pool with each other and determines for each member pair if they will have contact and if someone infects the other. Since everyone gets compared with everyone, this algorithm will be called *all-to-all* from now on. For every member pair in a contact pool, of which both individuals are present, the probability that they have contact is calculated. Then, a random number is drawn, based on the contact probability, to decide if the individuals have contact. When contact has been made, the next step is to look at the health statuses of the pair. If one of them is infectious while the other is susceptible, the probability of transmission is calculated. Similar to determining contact, the incident of the infectious person transmitting the disease to the susceptible one is decided by a random number based on the transmission probability. The susceptible person then becomes infected by the infectious one.

Infectious-to-susceptibles

Section 3.4.3 will explain how the all-to-all algorithm becomes a lot slower depending on the number of people in a contact pool. Therefore, an optimised approach has been designed to be a lot quicker in exchange for producing less information. This other algorithm will be called the *infectious-to-susceptibles* (inf-to-sus for short), because it only compares every infectious person in a pool with every susceptible one to determine who gets infected. At

Algorithm 1 Pseudo code of the all-to-all algorithm.

Input: $P_1 \dots P_N$ ▷ All members of the pool

```

1: for  $i \leftarrow 1$  to  $N$  do ▷ Iterate over all members
2:    $P_1 \leftarrow P[i]$ 
3:   if  $P_1$  not present then
4:     continue to next  $i$ 
5:   for  $j \leftarrow i$  to  $N$  do ▷ Iterate over members starting from  $i$ 
6:      $P_2 \leftarrow P[j]$ 
7:     if  $P_2$  not present then
8:       continue to next  $j$ 

9:    $C_{prob} \leftarrow$  probability  $P_1$  and  $P_2$  have contact
10:  if random number, given  $C_{prob}$ , says there is contact then
11:    Register contact
12:    if  $P_1$  or  $P_2$  susceptible and other infectious then
13:       $T_{prob} \leftarrow$  probability infection happens
14:      if random number, given  $T_{prob}$ , says there is transmission then
15:        Infect the susceptible one

```

first, the pool gets sorted based on the health status of every member, which is explained in the next section, and it is counted how many infectious members the pool contains. If there are no infectious people in the pool, the algorithm immediately stops. However, if there are infectious members of the pool, the next step is to compare the infectious ones with the susceptibles. For every pair of an infectious and a susceptible person, who are both present, the probability for them to have contact and the probability of transmission is calculated and multiplied with each other. This probability result is then used by a random number generator to decide if the infectious person infects the susceptible one, after which the susceptible person becomes infected. Note that this algorithm only generates information about the transmissions and limits itself to contacts between infectious and susceptible persons. As such, it cannot be used to calculate contacts over the entire population, which the all-to-all method can.

Sorting on health status

Line 10 of Algorithm 2 may look like it performs a redundant check by confirming that p_1 is infectious, but this is due to the way the contact pools are sorted on line 1. Algorithm 3 shows the pseudo code of this sortation based on the health statuses. A visual representation of a list of contact pool members before and after sorting on health statuses can be seen in Figure 3.4. Everyone who is immune is placed together at the end of the list and the susceptibles are arranged right in front of them. The remaining people with the health status exposed, infected, or recovered are put together with no specific order amongst them. Inf-to-sus therefore needs line 10 in order to separate the infectious from the rest.

Algorithm 2 Pseudo code of the infectious-to-susceptibles algorithm.

Input: $P_1 \dots P_N$ ▷ All members of the pool

- 1: sort $P[]$ on health status ▷ Explained in Algorithm 3
- 2: $N_{inf} \leftarrow$ number of infected members
- 3: **if** $N_{inf} = 0$ **then**
- 4: **return**
- 5: $i_{sus} \leftarrow$ index of first susceptible member
- 6: $i_{imm} \leftarrow$ index of first immune member
- 7: **for** $i \leftarrow 1$ to $(i_{sus} - 1)$ **do** ▷ Iterate over possible susceptibles
- 8: $P_1 \leftarrow P[i]$
- 9: **if** P_1 not present or not infectious **then**
- 10: continue to next i
- 11: **for** $j \leftarrow i_{sus}$ to $(i_{imm} - 1)$ **do** ▷ Iterate over susceptibles
- 12: $P_2 \leftarrow P[j]$
- 13: **if** P_2 not present **then**
- 14: continue to next j
- 15: $CT_{prob} \leftarrow$ probability of contact AND transmission between P_1 and P_2
- 16: **if** random number, given CT_{prob} , says there is contact and transmission **then**
- 17: P_1 infects P_2

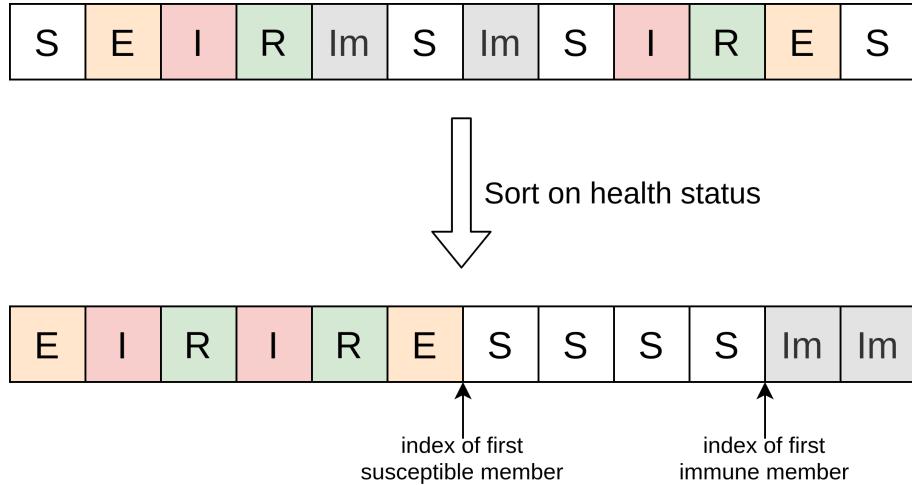


Figure 3.4: Visual representation of a list of contact pool members before and after being sorted on health status, where every square represents a member. The letters indicate the health status of a member: Susceptible, Exposed, Infectious, Recovered, or Immune.

Algorithm 3 Pseudo code of the function that sorts the members of a contact pool based on their health statuses.

Input: $P_1 \dots P_N, i_{imm}$ \triangleright All members of the pool, index of first immune member
Output: $P[]sorted, i_{imm}, i_{sus}$

```

1:  $i_{sus} \leftarrow 1$                                  $\triangleright$  Index of first susceptible member
2: for  $i \leftarrow 1$  to  $i_{imm}$  do                   $\triangleright$  Iterate over the pool members
3:    $member \leftarrow P[i]$ 
4:   if member is immune then
5:      $i_{imm} \leftarrow i_{imm} - 1$ 
6:      $swapped \leftarrow false$ 
7:     while  $i_{imm} > i$  and not  $swapped$  do
8:        $member_2 \leftarrow P[i_{imm}]$ 
9:       if  $member_2$  is immune then
10:         $i_{imm} \leftarrow i_{imm} - 1$ 
11:      else
12:        SWAP( $member, member_2$ )                 $\triangleright$  Switch positions in  $P[ ]$ 
13:         $swapped \leftarrow true$ 

14:   else if member not susceptible then
15:     if  $i_{sus} < i$  then
16:        $member_2 \leftarrow P[i_{sus}]$ 
17:       SWAP( $member, member_2$ )                 $\triangleright$  Switch positions in  $P[ ]$ 
18:        $i_{sus} \leftarrow i_{sus} + 1$ 
19: return  $P[ ], i_{imm}, i_{sus}$ 

```

Contact probability

Calculating the probability that two people have contact with each other takes a lot of factors into account, so only an abstract explanation of this will be given here. The pseudo code of the function that does these calculations is given in Algorithm 4. If the type of pool is a household, the contact probability is always 0.999. Otherwise, the first step is to retrieve the contact rates for both individuals in the specific pool type, which is further explained in Section 3.3.3. This number is a representation of how many contacts someone has in a specific pool type by looking at their age. These numbers are selected from the contact matrix, which is a configuration file, as seen in Section 3.1.4, based on the type of pool and the age of each individual. The lowest contact rate between the two individuals is then chosen as the definite number. This number is then multiplied by a contact adjustment factor based on the type of pool, which can be altered through the configurations. The adjustment factor is used, for example, to implement social distancing or intensify the contacts. The adjusted number of contacts is then transformed to the contact probability by dividing it by the total number of members in the pool minus one.

Algorithm 4 Pseudo code of the function that calculates the contact probability between two individuals in a contact pool.

Input: P_1 , P_2 , $type$, and N \triangleright Individuals, type of pool and total number of members
Output: $prob$ \triangleright Probability of P_1 and P_2 having contact

```

1: if type = Household then
2:   return 0.999

3: contactsp1 ← contact rate in pool( $type$ ) for age( $P_1$ )
4: contactsp2 ← contact rate in pool( $type$ ) for age( $P_2$ )
5: contacts ← min(contactsp1, contactsp2)

6: factor ← contact adjustment factor based on  $type$ 
7: contacts ← contacts * factor
8: prob ← contacts/( $N - 1$ )

9: prob ← min(prob, 0.999)                                     $\triangleright$  Probability never higher than 0.999
10: return prob

```

Transmission probability

The probability of an infectious person transmitting their disease to a susceptible person when making contact depends on the combination of two things:

1. The transmission probability of the disease itself, which is defined in the disease characteristics configurations.
2. The relative transmission factor from the infected to the susceptible. This is based of a factor that represents the susceptibility if the susceptible person is younger than

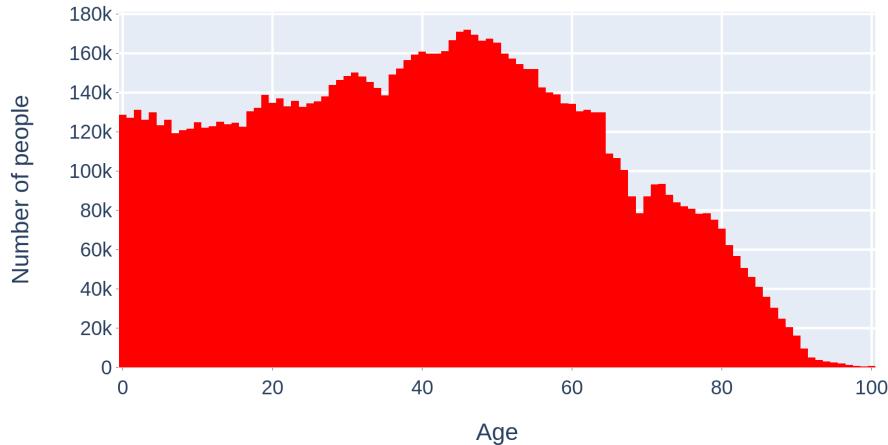


Figure 3.5: Age distribution of the 11M population.

18 years old and the transmissibility if the infected is asymptomatic.

3.3 Data

Now that we have a better understanding of how our model works, we want to know what data it uses and how this data looks like. Because Stride can be run with different configurations and data, our goal here is not to focus on the data that has been used and examine it. Rather, we want to gain insight in the data that we will be using for the rest of this thesis, so that we can keep this knowledge in mind when analysing results. Eventually, our objective is to improve Stride in general, not just for a specific set of data.

3.3.1 Population

The first type of data we will look at is the population, for which Stride primarily uses the census data for Belgium from Eurostat². For the research and tests of this thesis we used three different population representations, based on the 2011 census, which are also being used by the main researchers of Stride: 600k, 3M, and 11M. These names are also an indication of the number of people every population contains, which is respectively 600 thousand, 3 million, and 11 million. 11M is the most accurate representation of the Belgian population and is naturally used for simulations regarding Belgium. 600k and 3M are smaller versions of the 11M which we only used when testing our optimisations. Therefore, we only discuss the 11M population in this section, of which the age distribution is shown in Figure 3.5.

3.3.2 Contact pools

All of the action in a Stride simulation occurs in the contact pools. This is where individuals have contact with one another and transmit the disease. Table 3.1 presents the general

²<https://ec.europa.eu/eurostat>

	Household	School	Workplace	Primary community	Secondary community
Number of pools	4,859,837	131,756	566,759	22,000	22,000
Min. pool size	1	6	1	51	44
Max. pool size	6	50	1000	1431	1437
Avg. pool size	2.3	20	7.8	500	500

Table 3.1: General statistics of the contact pools of the 11M population.

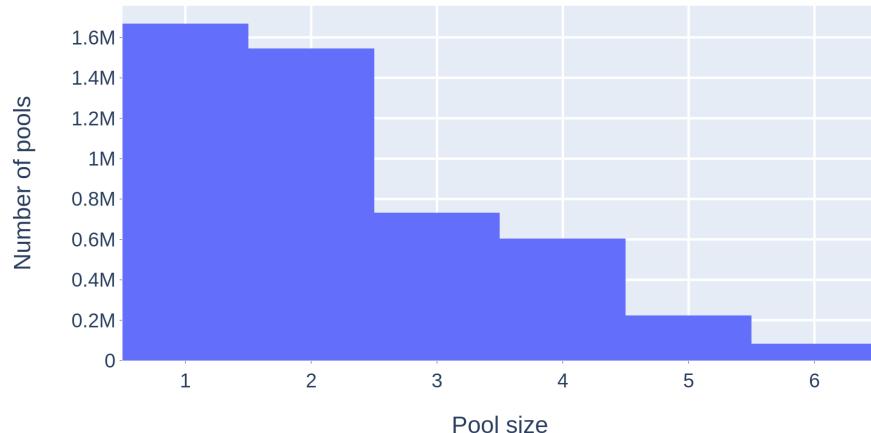


Figure 3.6: Distribution of the household pool sizes of the 11M population.

statistics of the different pool types next to each other. We see that there are vastly more household pools than any other type, but they are very small with a maximum of 6 and an average of 2.3 people per pool. The primary and secondary communities seem to be very similar and their average sizes are noticeably large. Workplaces can contain up to a thousand people, but are disproportionate compared to the other pool types with an average of 7.8 people. The school pools do not show any remarkable traits, but we will see that its general statistics are somewhat misleading. This information gives us a first view of the contact pools, however, a more profound analysis for every specific pool type is definitely needed.

Household

Since the total number of people in a household lies between one and six, there is probably not much variation possible amongst the household pools. Figure 3.6, which displays the distribution of household pool sizes, confirms this statement.

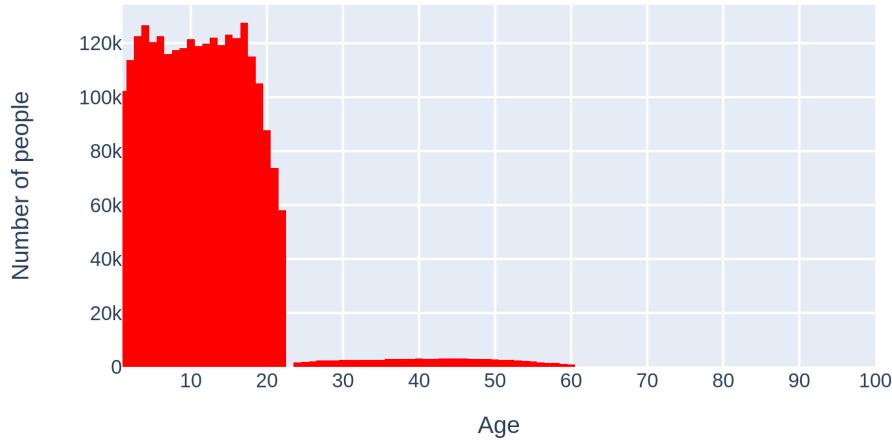


Figure 3.7: Age distribution of all the individuals that have a school contact pool in the 11M population.

School

Looking at the entire population, 24% of the people have a school contact pool and only individuals younger than 60 years old can be a member, which Figure 3.7 shows us. In contrast with the general statistics, the distribution of school pool sizes in Figure 3.8 shows that the school pool sizes are not evenly distributed and display a peculiar feature. We notice three different size 'clusters': pools with size 1 to 10, 14 to 24, and 50. There is not much information that we can use from our 11M population to interpret these results, except for the age of the individuals in each cluster. Further inspection of the school contact pools with size 50 show that every individual is at least 18 years old. It is thus safe to assume that these only represent the college contact pools, which is also a logical explanation for the relative large pool sizes. The other two clusters do not show any age-specific traits, so we cannot say with certainty that these are only K-12 schools. There are also a couple of school contact pools with sizes outside of the cluster sizes.

Workplace

Just like with schools, not everyone has a workplace pool. 40% of the population has a workplace contact pool and only people between the ages 17 to 77 can be a part of one, which is shown in Figure 3.9. We also noticed how the workplace contact pools can contain one to a thousand people with an average size of 7.8. The distribution of workplace contact pool sizes displayed in Figure 3.10, gives us insight into why this average size is so low. We clearly see that there is a major difference in the number of workplace pools with small sizes compared to the rest.

In order to get a more detailed view of the number of workplace pools per size, we split them up in multiple age distribution ranges in Figure 3.11. The different ranges that we have chosen display more precisely the number of pools per pool size. We only notice that in Figure 3.11(b) there still is a big difference between the sizes smaller than 20 and

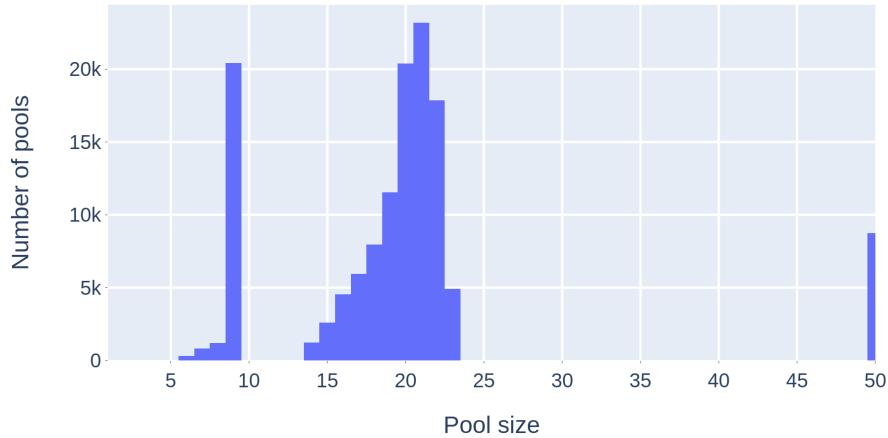


Figure 3.8: Distribution of the school pool sizes of the 11M population.

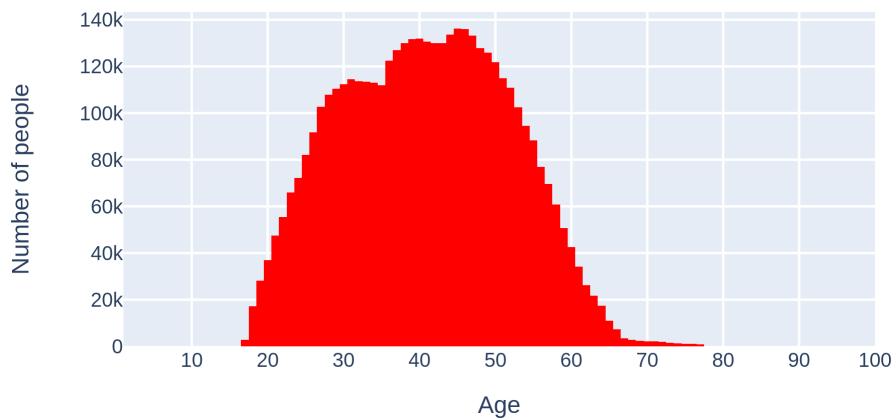


Figure 3.9: Age distribution of all the individuals that have a workplace contact pool in the 11M population.

the rest. Following this, we now look for 'clusters' of pool sizes, or pool size ranges, in which the pool sizes in the same range have approximately the same number of pools. The ranges that we assume, looking at Figure 3.11, are: [1,9], [10,19], [20,49], [50,249], and [250,1000]. The statistics for these ranges in Table 3.2, confirm that our ranges contain pool sizes with similar number of pools. The range [1,9] contains, as expected, 94% of all the workplace pools, but only consists of 60.5% of the working individuals. So, although there are not that many workplace contact pools with more than nine people, they still account for 39.5% of all the individuals with a workplace pool.

Primary & secondary community

When we looked at the general statistics of the pool types, we noticed that the primary and secondary community contact pools had very similar results. Figures 3.12 and 3.13 demonstrate that these types of pools are nearly identical and both have a normal pool

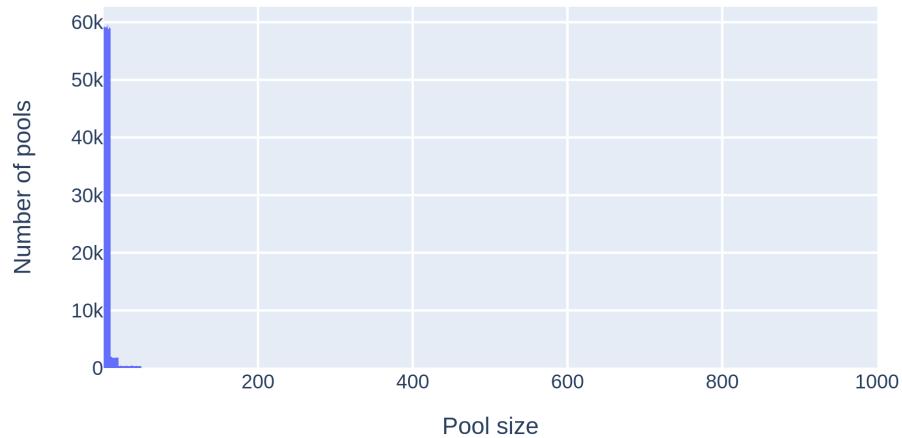


Figure 3.10: Distribution of the workplace pool sizes in the 11M population.

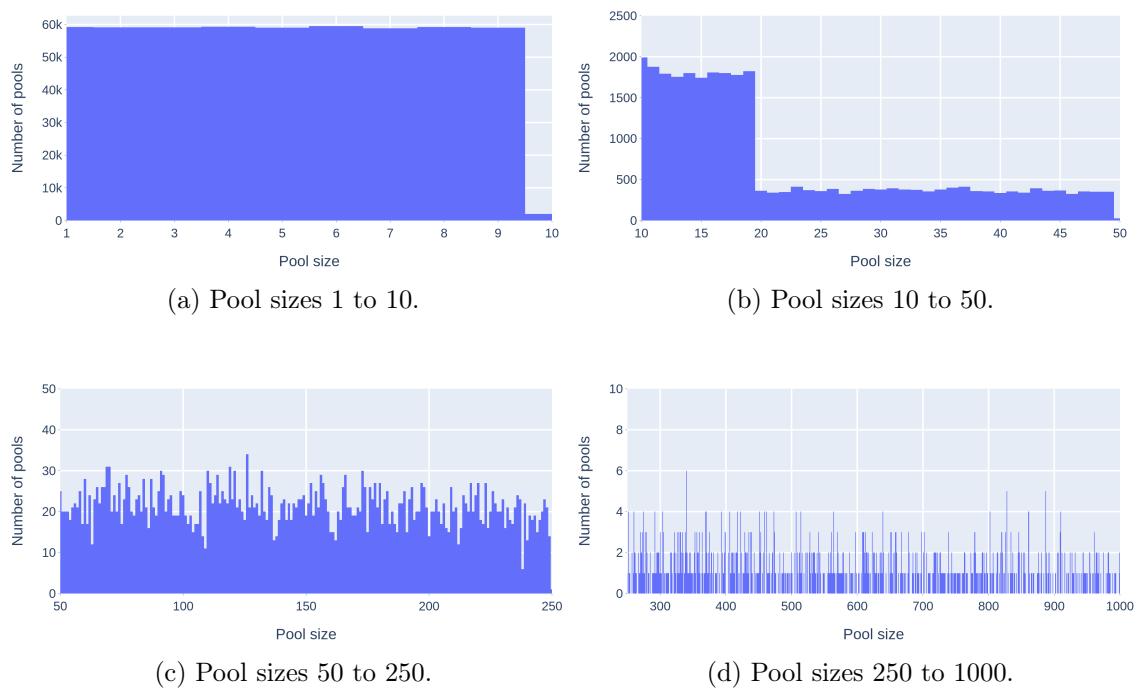


Figure 3.11: Distributions of the workplace pool sizes from Figure 3.10 for different pool size ranges.

Pool size range	1 - 9	10 - 19	20 - 49	50 - 249	250 - 1000
Minimum number of pools (for a pool size)	58,827	1,744	324	6	1
Maximum number of pools (for a pool size)	59,547	1,992	414	34	6
Number of pools	532,518	18,169	10,983	4,286	803
% of pools	94.0	3.2	1.9	0.8	0.1
Number of people	2,661,724	262,427	377,612	629,345	467,110
% of working people	60.5	6.0	8.6	14.3	10.6

Table 3.2: Statistics for the different workplace pool size ranges. The pool sizes in every size range have approximately the same amount of pools.

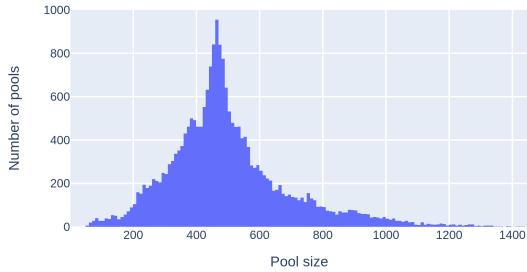


Figure 3.12: Distribution of the primary community pool sizes of the 11M population.

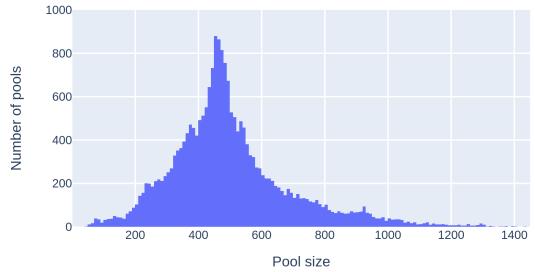


Figure 3.13: Distribution of the secondary community pool sizes of the 11M population.

size distribution with a peak around size 500.

3.3.3 Contact matrix

In Section 3.2.5 we learned how Stride calculates contacts based on the configurated contact matrices, which was explained in Section 3.1.4. For every type of contact pool there exists a matrix which contains the contact rate based on age. The matrices for a school, workplace, primary community, and secondary community are respectively displayed in Figures 3.14, 3.15, 3.16, and 3.17. These rates indicate the average number of contacts an individual has in a particular contact pool with respect to their own age. Algorithm 4 shows how the contact probability gets calculated between two individuals. The probability of two people having contact in a household is always 0.999, therefore, we do not need a contact matrix for the household contact pool type.

The data for these contact rates originates from SOCRATES, which is an online interactive tool³, created by Willem et al. [28], for the sharing of social contact data. The

³<http://www.socialcontactdata.org/socrates/>

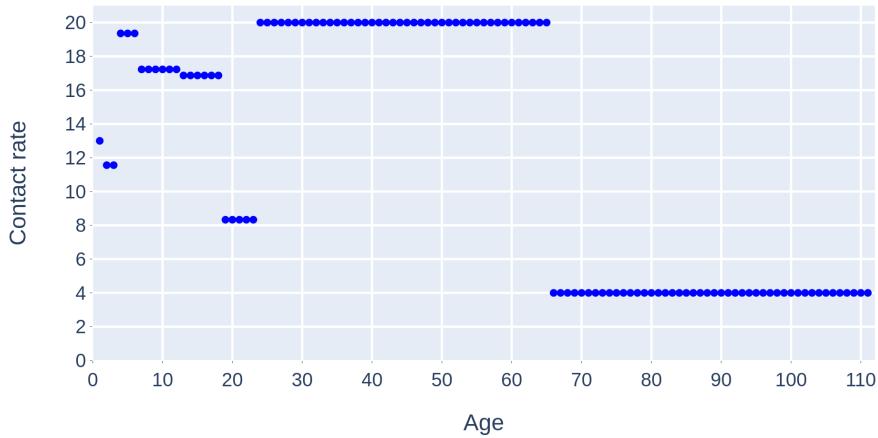


Figure 3.14: Contact rates per age at school.

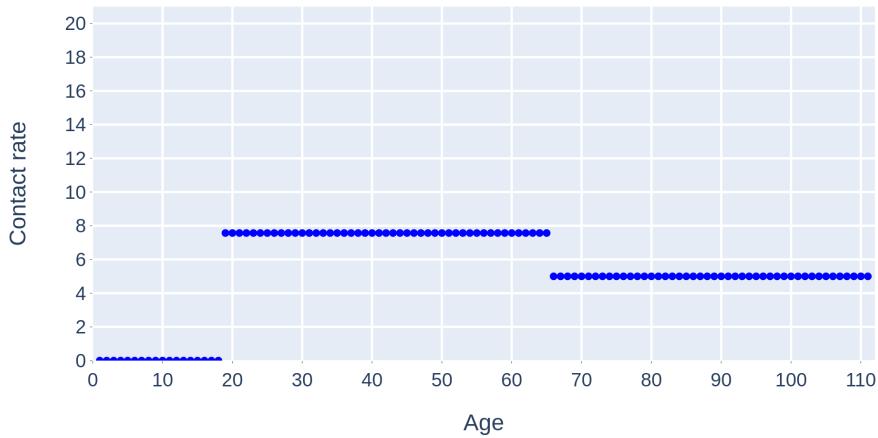


Figure 3.15: Contact rates per age at work.

data they present are 2D matrices in which the contact rate in a pool is based on the ages of the person who initiates contact and the one who receives it. These matrices for our contact pool types are visualised in Figure 3.18, which are different than the ones we presented. The contact rates that Stride uses are actually 1D matrices or just *vectors*, which are based on those 2D matrices. These matrices have been transformed into our vectors so that contact rate look-ups are faster and easier to use. If we would use the actual matrices, with dimensions 112x112, the look-ups would take substantially more time than with our vectors. Thus, from now on we will refer to these 1D matrices as the *contact vectors*. Furthermore, in the 2D matrices we again clearly see that people have the most contact with people who are the same age, with an exception in the workplace. This exception has an intuitive explanation, which is that people at work have contact with everyone disregarding age and that people at work are mostly between the ages of 18 to 65 years old.

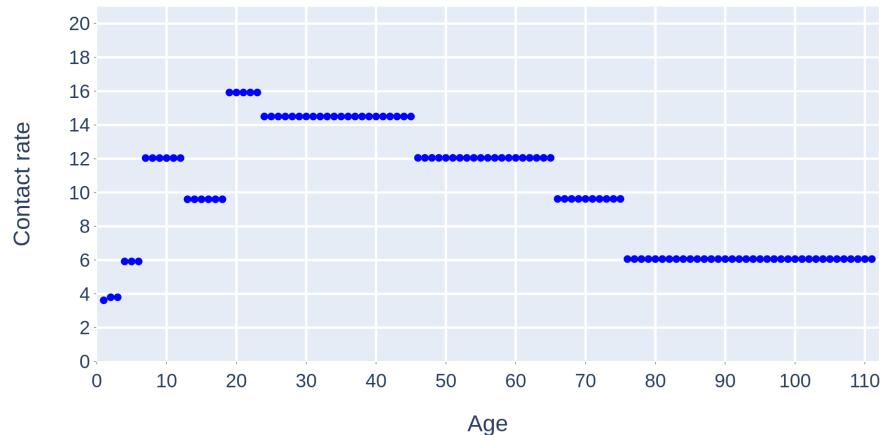


Figure 3.16: Contact rates per age in a primary community.

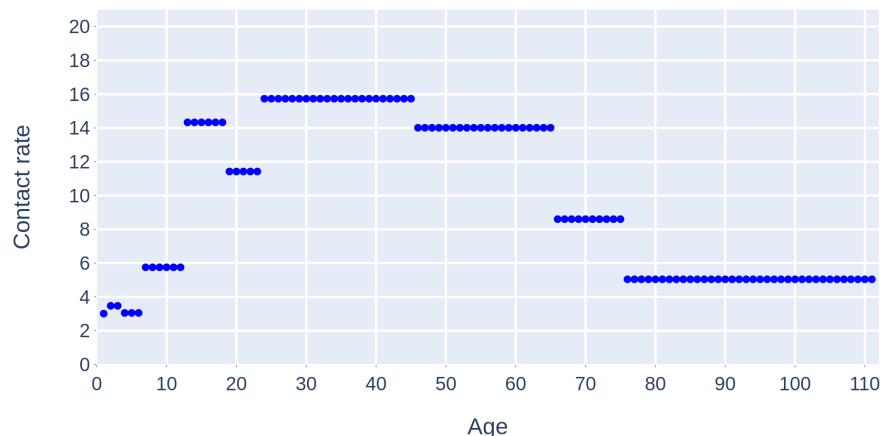


Figure 3.17: Contact rates per age in a secondary community.

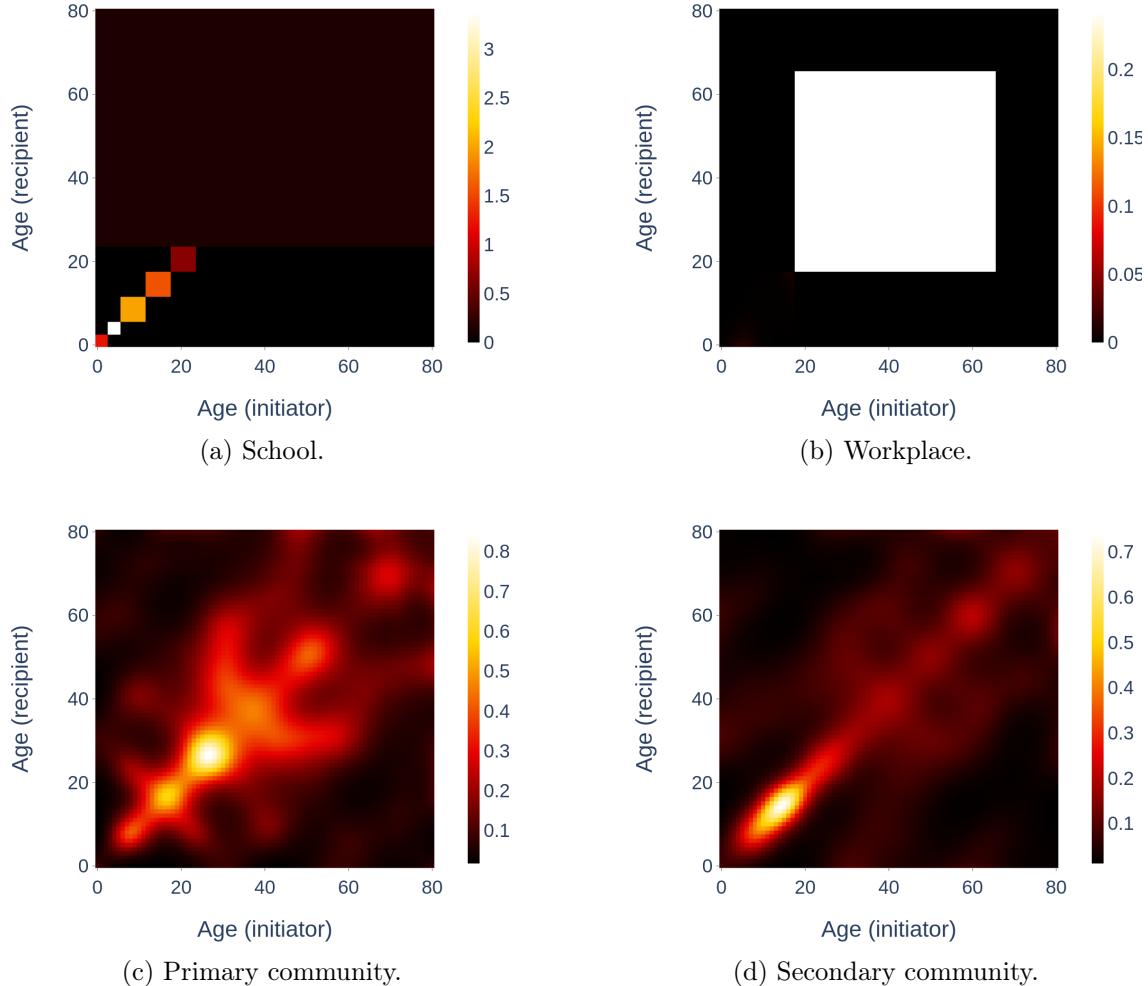


Figure 3.18: 2D contact matrix heatmaps showing the contact rates based on the ages of the contact initiator and recipient.

3.4 Analysis

Now that we have a complete understanding of Stride, we can dig into the code and start to analyse it.

3.4.1 Parallelization

A well-known optimisation technique in programming is called *parallelization*, which is the act of executing multiple parts of the program simultaneously. This can only be applied when a program allows for its code to be run in parallel. There are various parallelization techniques and each has its own pros and cons, so which one is most suitable depends on the situation. Stride also uses such a technique and it is called OpenMP⁴.

OpenMp In computer science, a thread (of execution) is a term used to describe the smallest sequence of program instructions that can be run independently [29]. OpenMP is an implementation of multithreading, which is the act of executing multiple threads concurrently and it is managed by the CPU (central processing unit). A thread runs on a core of a CPU and such a core can execute multiple threads concurrently. A core can only execute one thread at a time, so when there are multiple threads that need to be executed by a core, some need to wait their turn. If we were to use multithreading on a single core, the program would become slower instead of faster, because the threads would still be run sequentially and there would be overhead from the CPU who has to manage the threads. CPUs nowadays have multiple cores which allows us to divide the threads on the cores. When we now want to use OpenMP, we need to check the number of cores our CPU has. We then use this number to tell our program the maximum number of threads it can use for the most optimal run time.

An important aspect of OpenMP is that CPU cores share their memory. How this memory sharing exactly works depends on the CPU, but these details do not belong to the scope of this thesis. However, this can become an issue when working with a lot of data which, as we will see, is the case for Stride.

3.4.2 Code

In Section 3.2 we explained how a Stride simulation works, which was also visualised in Figure 3.3. Every part of the simulation process will we now examine separately while we simultaneously think about possible ways to improve the code that we are looking at.

Initialisation

We have seen that the initialisation consists of reading the configurations and setting up the simulation accordingly. Most of these configurations are values that have an effect on how the simulation has to run and what information it has to produce, but they only take one hundredth of a second and can thus be ignored. The first major portion is reading the 11M population file, which translates to reading 11 million rows and creating an individual

⁴<https://www.openmp.org/>

for every single row. This is done in sequence by reading row after row and creating a person every time and storing it in the population. Reading this file could be done in parallel, but trying to store the individuals concurrently in the population would probably cause more harm than good due to the threads that need to store something in the same memory.

After the population and contact pools have been created, it is turn to the seeders to determine the individuals' health characteristics, the people who are immune, and the people who start infected. These seeders are already optimised by using OpenMP where possible.

Updating individuals

The next step in our code is the update of every individual. This consists of updating a person's health status, as seen in Section 3.2.2, and then immediately updating the pools in which the person will be present for the upcoming day, as seen in Section 3.2.3. Stride iterates over the entire population and updates every individual separately, but this iteration is implemented with the possibility to use OpenMP. This, together with the fact that the updates are very small functions, indicates that there will probably be not much room to improve Stride here.

Contact tracing

As we know from Section 3.2.4, it is possible to use a contact tracing feature in Stride. This code section iterates twice over the population without OpenMP, so this could already be a potential optimisation by parallelizing the iterations. However, because this part of the code does not belong to the standard simulation, we give this section the lowest priority when making improvements. Besides these iterations the contact tracing performs some operations that need to be done and cannot be changed or improved.

Calculating contacts and transmissions

Section 3.2.5 explained the core of Stride, which is how contacts and transmissions get calculated. There are two algorithms for this: all-to-all and inf-to-sus. Only one of these can be used when simulating a day and this is determined at runtime. They are both implemented in a template called *infector* and how Stride executes this for every pool is shown in Algorithm 5, which consists of pseudo code combined with the actual OpenMP calls for completeness. We iterate over all the contact pool types and check if the type of pool is active in the current day and if not, we skip this type of pool. If we are, for example, simulating a holiday, we skip the workplace type. If the type of pool is active, we then iterate over every pool of that type and send them to the infector. During this thesis, several questions and concerns arose about this implementation, which we will now discuss:

1. **We calculate the contacts and transmissions for every pool of every type in one large iteration, how does this affect the results?**

The order in which we do all these calculations does not matter that much. When someone becomes exposed, it always takes at least a day before they get infected or

start feeling symptoms. Therefore, they will not be able to infect someone in the same iteration or change their behaviour.

2. When we use OpenMP, is there a possibility that problems occur regarding memory read and write operations? For example when two pools are being processed by infector and they consist of overlapping individuals.
The parallelization only applies to the inner for-loop which iterates over all the pools of a particular type. We have seen that people can only be a member of one pool per pool type, so this rule prevents such issues.
3. At line 10 in Algorithm 5 the population is passed on to infector. Why would this be necessary and does this present problems regarding OpenMP?
The population is needed in infector because it gets used when calculating the contact probabilities, although we did not explicitly show this in Algorithm 4. This does not cause any issues because the population is wrapped in a shared pointer, as we will see in Section 3.5.2.
4. We learned that the contacts and transmissions depend on randomness. Since randomness has to be created via random number generators, would this not cause problems when using OpenMP?

In Section 3.2.1 we discussed how random number generator (RNG) managers are the first thing that get created in the initialisation step. Stride creates such RNG managers for every thread and passes these on to infector, as we see in line 10 of Algorithm 5, which takes care of any issues regarding random number generators with multithreading.

Algorithm 5 Pseudo code of how 'infector' is used on every pool in a simulation day.

```

1: #pragma omp parallel num_threads(m_num_threads)
2: {
3:     thread_num ← OMP_GET_THREAD_NUM()
4:     for each type in pooltypes[] do                                ▷ Iterate over pool types
5:         if type not active then
6:             continue                                              ▷ Ignore this type and go to the next
7:         #pragma omp for schedule(static)
8:         for each pool in pools[type] do                          ▷ Iterate over every pool of type
9:             INFECTOR(pool, population, rng_handler[thread_num], ...)
10: }
```

3.4.3 Performance

A first code examination of Stride taught us more about the different components of the program and how parallelization is already used to improve the simulation speed. Our next step is to run the simulation with different parameters and measure the performance. All simulations in this chapter and the next use the configurations described in Appendix A2 unless stated otherwise.

VSC All of the following tests in this chapter and Chapter 4 ran on the VSC⁵ (Vlaams Supercomputer Centrum, or in English Flemish Supercomputer Centre) to get the most optimal results with the least amount of external interference. Our tests ran on computer nodes with a Xeon Gold 6140 CPU@2.3 GHz or Xeon Gold 6240 CPU@2.6 GHz, both with 18 cores; 192 GB RAM; 200 GB SSD local disk. All-to-all simulations have memory and virtual memory peaks of respectively 9.5 GB and 10.2 GB, while inf-to-sus only has peaks of 3.1 GB memory and 3.3 GB virtual memory. Memory bandwidth and latency measurements can be found on the VSC website⁶.

For our performance tests we divide the model in three sections and measure the time of each section as well as the total time. The sections are: updating individuals, contact tracing, and calculating contacts and transmissions (which we will now refer to as infector). With our first test, we want to see how much time Stride spends per section each day for both infector algorithms. The results for these simulations using all-to-all and inf-to-sus are visualised in Figure 3.19. The average results per section are listed in Table 3.3. The small runtime differences, such as the updating and tracing sections for both algorithms, are due to low-level system details and are negligible for the rest of this thesis.

We notice from our results is that the inf-to-sus is a major runtime optimisation compared to all-to-all, which is all due to the difference in the infector speed. When Stride uses all-to-all, almost all of its time is spent in the infector, so this is where we should look for improvements first. The ups and downs that we see in both graphs are caused by the type of day, five weekdays and 2 weekend days, where the weekdays take a little bit more time. This is a logical effect because the weekdays have more active pools (household, work, K-12 school, college, and secondary community) than weekend days (household, primary community).

Inf-to-sus displays a somewhat normal distribution regarding the infector time, with a peak around day 50. The explanation for this behaviour is that the infector only gets executed for a pool when there is at least one person infectious in that pool. Figure 3.20 shows the number of people who are infected per simulation day. The number of people who are infected clearly matches these changes in runtime of the infector. The first day of this simulation also takes a lot more time than the following days, which is due to the sorting of the individuals based on their health status. At the start, these pools are initialised without sorting the members and thus it takes more time to sort them. When they get sorted the next day, there only need to be a few changes made on average which causes the sorting to be much faster.

The total runtime of the entire simulation using all-to-all lasted 1 hour 51 minutes and 55 seconds, of which the initialisation took approximately 1 minute and 40 seconds. The inf-to-sus simulation obviously needed the same amount of time for the initialisation, but ran in a total time of 5 minutes and 49 seconds. Inf-to-sus only does calculations that

⁵<https://www.vsccentrum.be/>

⁶https://vlaams-supercomputing-centrum-vscdocumentation.readthedocs-hosted.com/en/latest/leuven/tier2_hardware/memory_bandwidth_and_latency_tier2.html

	Updating	Tracing	Infector	Total
all-to-all	1.08	0.15	64.33	65.56
inf-to-sus	1.10	0.13	0.89	2.12
speedup	/	/	72.28	30.92

Table 3.3: Average runtime (in seconds) per section for simulations run on 11M without holidays and 1 thread.

are absolutely necessary and has therefore very little room for improvement compared to all-to-all. Our focus will thus not lie on the initialisation, because it is only a small percentage of the simulation when using all-to-all. Contact tracing needs the least amount by far, so we ignore this part from now on and will only mention it sometimes for the sake of completeness.

Parallelization

The next test that we will perform is to see how the implemented parallelization affects the runtimes. For this part we will only focus on the parts of a simulation day that use OpenMP, which are the updating of individuals and the calculations of contacts and transmissions. Figure 3.21 reveals how the updating of individuals becomes slower when using more threads instead of faster. The implementation of OpenMP, which is supposed to be an improvement, has the opposite effect. We see a similar trend when looking at the all-to-all infector in Figure 3.22. Here we notice that using two threads is the slowest and that using more threads than becomes gradually faster, although it is still much slower than the simulation without parallelization. The inf-to-sus infector exhibits rather strange behaviour regarding parallelization, which we can see in Figure 3.23. Using multiple threads is faster in the beginning and at the end of the simulation, but is slower in between. Inf-to-sus has, just like all-to-all, the slowest times when using two threads and starts to become faster if more threads are used.

In order to be more accurate, the average times per simulation day for all of these parallelization tests are displayed in Table 3.4. This confirms our statements about updating individuals and the all-to-all infector parallelization, but shows that multithreading is in fact faster on average for the inf-to-sus infector. However, the total simulation time is still higher when using multiple threads because of the updating of individuals.

3.5 First optimisation

In this section we examine the parallelization problem in depth and present our first major optimisation.

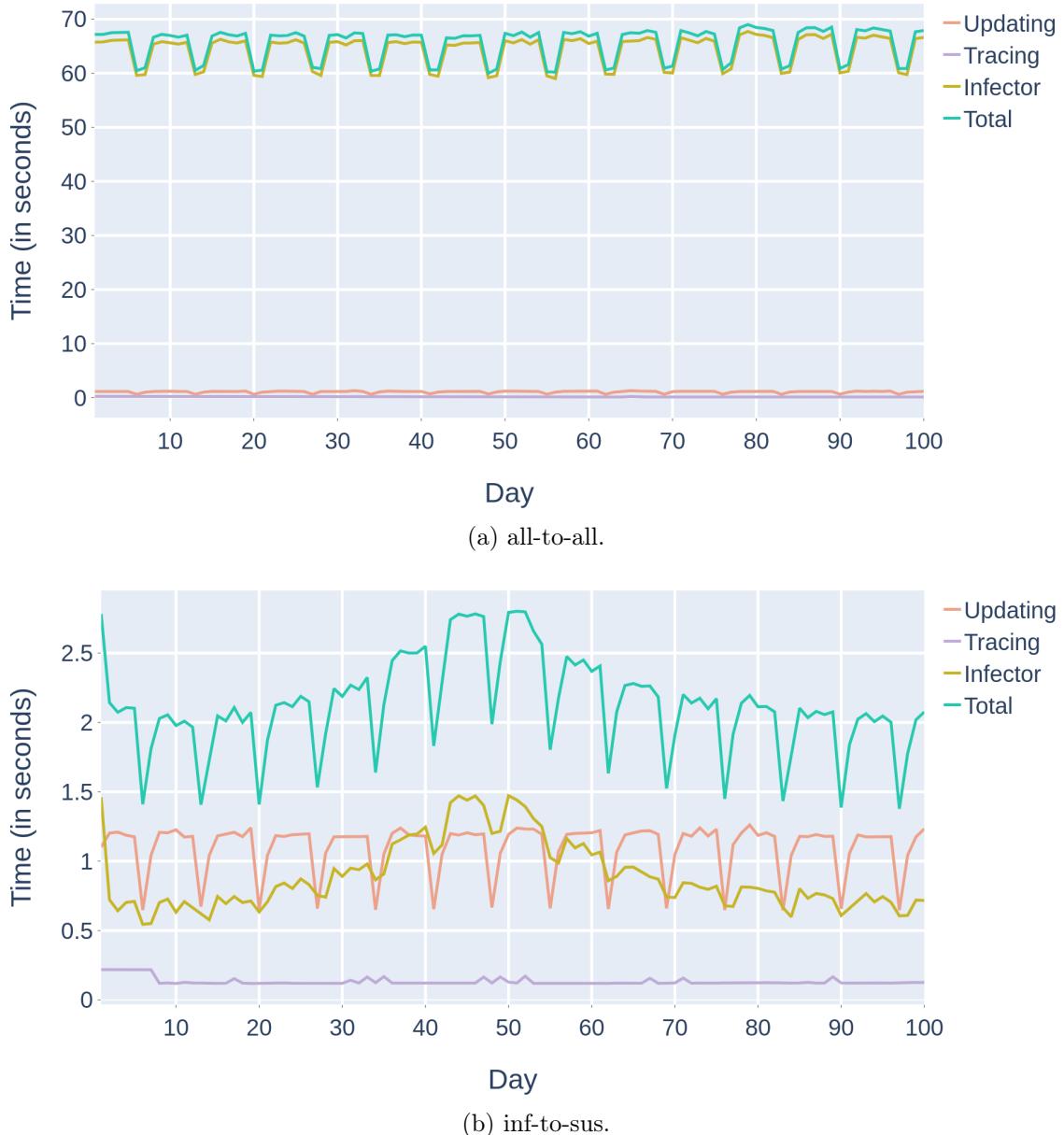


Figure 3.19: Section times per day, in seconds, for every section, including the total times for simulating a day. Simulations run on 11M without holidays using 1 thread.

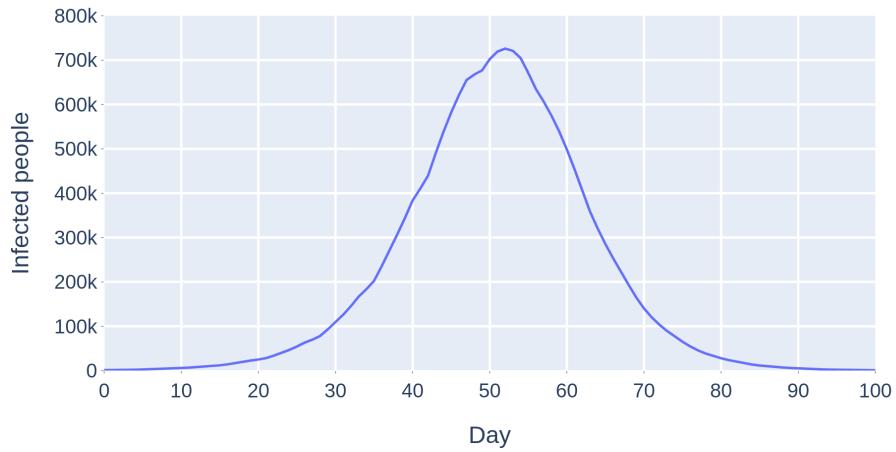


Figure 3.20: Number of infected people per simulation day. Simulation run on 11M without holidays using 1 thread.

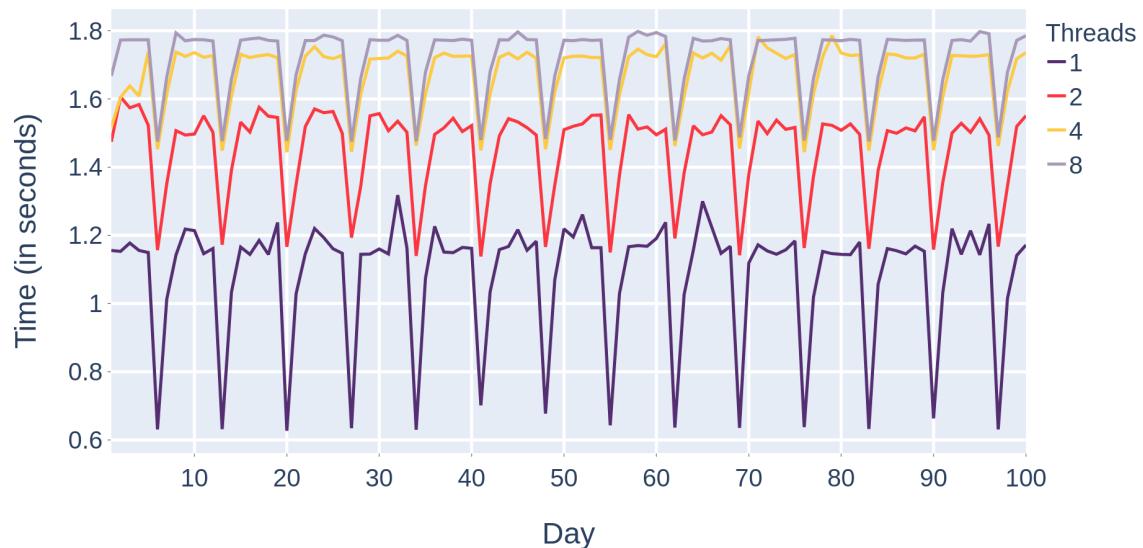


Figure 3.21: Time per simulation day for updating individuals using different number of threads. Simulation run on 11M without holidays.

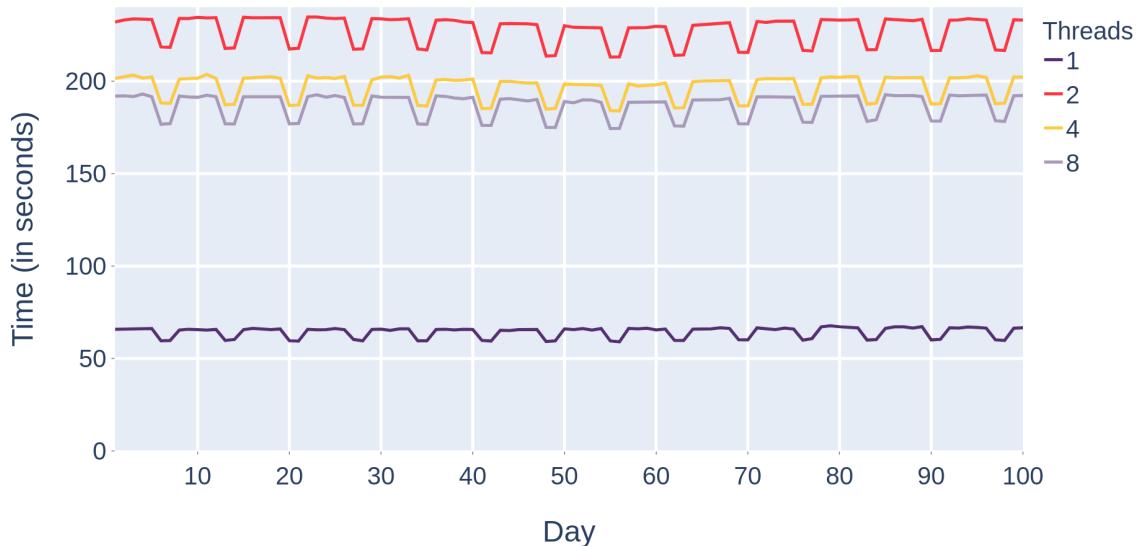


Figure 3.22: Time per simulation day for the infector using all-to-all using different number of threads. Simulation run on 11M without holidays.

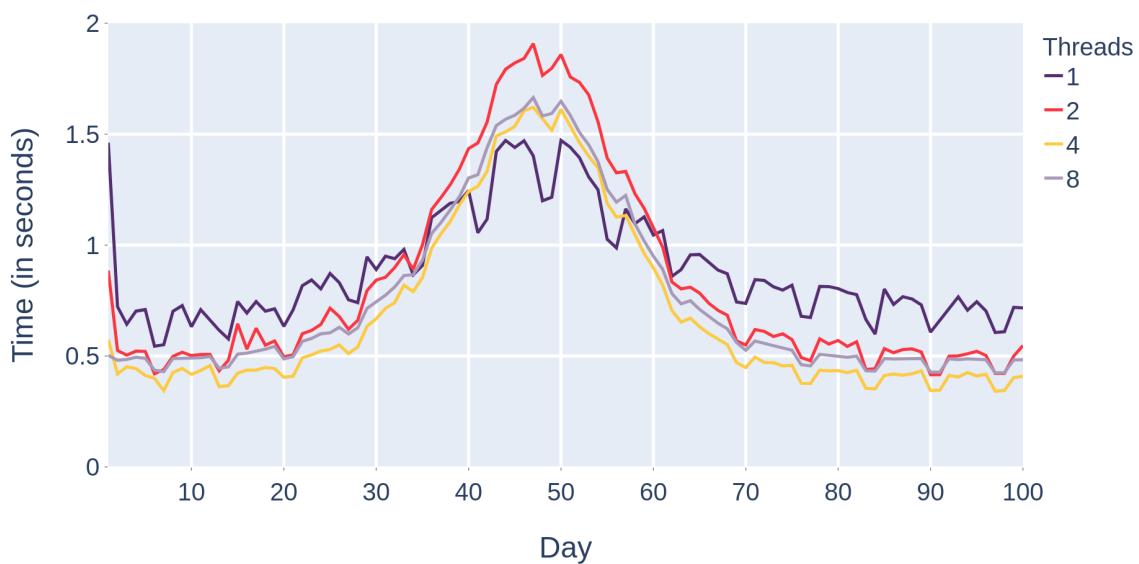


Figure 3.23: Time per simulation day for the infector using inf-to-sus using different number of threads. Simulation run on 11M without holidays.

Threads	Updating	Infector	Total	Threads	Updating	Infector	Total
1	1.08	64.33	65.56	1	1.10	0.89	2.12
2	1.45	227.93	229.50	2	1.52	0.83	2.48
4	1.67	197.04	198.85	4	1.66	0.69	2.49
8	1.72	187.15	189.00	8	1.67	0.75	2.57

(a) All-to-all.

(b) Inf-to-sus.

Table 3.4: Average runtime (in seconds) per section for simulations run on 11M without holidays.

3.5.1 Code architecture

Before we go into details about Stride, we need to disclose the architecture of the code. We will need this information when we need to contemplate about what causes these performance issues.

Population

Representing 11 million individuals is, needless to say, a lot of data that has to be stored and managed. Stride uses a segmented vector for its population, which is a container that stores objects, which are individuals in our case, almost contiguously in a chain of blocks. The block size, which is set to 512, determines how many individuals get stored consecutively. It is designed this way to have better performance regarding CPU caching.

Person

Every individual is represented in a *Person* object. This object contains various information such as its ID and age, but also of which pools they are a member and in which they are currently present. Information about their health is stored in another *Health* object. This health object stores the person's health status as well as their health characteristics.

Shared pointer

The *Population* is wrapped in a *std :: shared_ptr*. This is a smart pointer designed to facilitate the managing of storage of an object pointer when there are multiple owners that manage its lifetime [30]. The underlying structure has a reference counter which keeps track of all instances of the shared pointer and deletes the memory source and itself when this counter reaches zero [31].

3.5.2 Updating individuals

First we look at the updating of individuals, which needs more time to complete the more threads that are being used. In Sections 3.2.2 and 3.2.3 we discussed how every individual gets updated at the start of a simulation day regarding their health status and their pool

presences. This is implemented with a single iteration over the entire population in which it calls the 'update' function of a Person. The examination of this code does not present any visible errors. A possible reason for the multithreading problem could be related to the memory management done by the CPU. If we want to solve this, there is a strong possibility that we need to do extensive research on this topic and maybe redesign a great portion of the code, which lies outside the scope of this thesis. For this reason and because this section requires only a small portion of the total simulation time, we will ignore this section for now.

Let it be stated that we only made assumptions here by looking at the code. We did not perform tests on this section because of the little impact it would have on the total runtime. There is always the possibility to turn off the multithreading for this section, which would still result in acceptable runtimes.

3.5.3 Infector

We learned that it costs substantially more time for the infector when we use multiple threads and that the inf-to-sus infector displays a strange phenomenon, as seen in Figure 3.23, where multithreading can be both faster and slower depending on the day. We already said that the curve of the inf-to-sus infector graph is connected to the number of people who are infected. When there are only a few infected people, the infector from Algorithm 2 then can skip a lot of pools that do not contain an infected member. When multiple threads are being used in the days with a few infections, the infector for inf-to-sus is faster than without multithreading. For these reasons we can conclude that the parallelization with OpenMP works well to distribute the infector calls amongst the threads, as seen in Algorithm 5, and that the fault probably lies in the inf-to-sus code.

3.5.4 Solution

After detailed examination of both infector algorithms and performing tests, we noticed that most of the infector time was spent for calculating the contact probabilities. When we looked at the code for this function, for which the pseudo code is shown in Algorithm 4, we noticed that the population is also a parameter. The reason that this function cost so much time, was because the *shared_ptr* of this population was being passed by value. In C++, when a parameter is passed by value, a copy of that parameter is created in memory and passed on instead of the original. This is useful when we want to make sure that the function does not change the original data. However, this means that in our case a copy of the *shared_ptr* of the population needs to be created every time we want to calculate a contact probability. Let it be clear that this is not the same as copying the population itself, but only the *shared_ptr*.

A copy of a normal pointer in C++ would not cause much trouble, because it is only a reference number to the data it points to. When a *shared_ptr* or a copy of it gets created or destroyed, the reference counter of the shared pointer needs to be updated, which causes some overhead [31]. The calculation of the contact probability is called for every pair of individuals in all-to-all and for every infectious and susceptible pair in inf-to-sus in

a pool. Since there is a gigantic amount of pools that need to be passed on to infector, this results in innumerable contact probabilities that need to be calculated and thus results in a lot of overhead. A secondary issue is that the updating of the *shared_ptr* reference counter is an atomic operation, which means that when a thread needs to update it, it blocks other blocks from accessing this reference counter. Because of this, all the other threads have to wait their turn so that they can update the counter before continuing their thread of execution. This means that there is even more overhead when we use multiple threads for the infector [31].

The solution for all of this is to change the code so that the *shared_ptr* population is passed by reference instead of by value. This will only create a reference to the shared pointer instead of creating a new one. When a value is passed by reference, the function that gets called then has the power to change the original value, so we need to be cautious when we do this. Because of this, we examined the code of the contact probability function and saw that there were only read operations on the population data, so we can safely say that our changes will not result in malicious code.

3.5.5 Results

The changes that we made only affect the infector and the total runtime. The comparisons between the original version and our solution without multithreading is shown in Figure 3.24. Figure 3.24(a) shows us that the infector and total runtime of all-to-all is now almost twice as fast with our change than before. Inf-to-sus only shows a small increase in speed in Figure 3.24(b). The results in Table 3.5 confirm that we have an average total speedup per simulation day by 1.81 and 1.04 when using respectively all-to-all and inf-to-sus. Figure 3.25 visualizes the new ratios of the sections of the optimised Stride simulations using all-to-all and inf-to-sus. The averages of the sections are displayed in Table 3.6. All-to-all spends significantly less time in infector than before, while the differences in speed for inf-to-sus are very little.

We stated that multithreading would benefit even more if we pass the *shared_ptr* population by reference instead of by value. Figure 3.26 proves that we were correct and that we can now get better times with parallelization. In Table 3.7 we see that the speedups for all-to-all become larger the more threads that are being used. Although we improved Stride considerably, there is still a problem with all-to-all multithreading. Using eight threads on inf-to-sus is also slower than four. These issues might all have to do something with the memory, as mentioned before. For this reason, and because the primary researchers of Stride do not use parallelization, we will focus from now on only on using Stride with only thread.

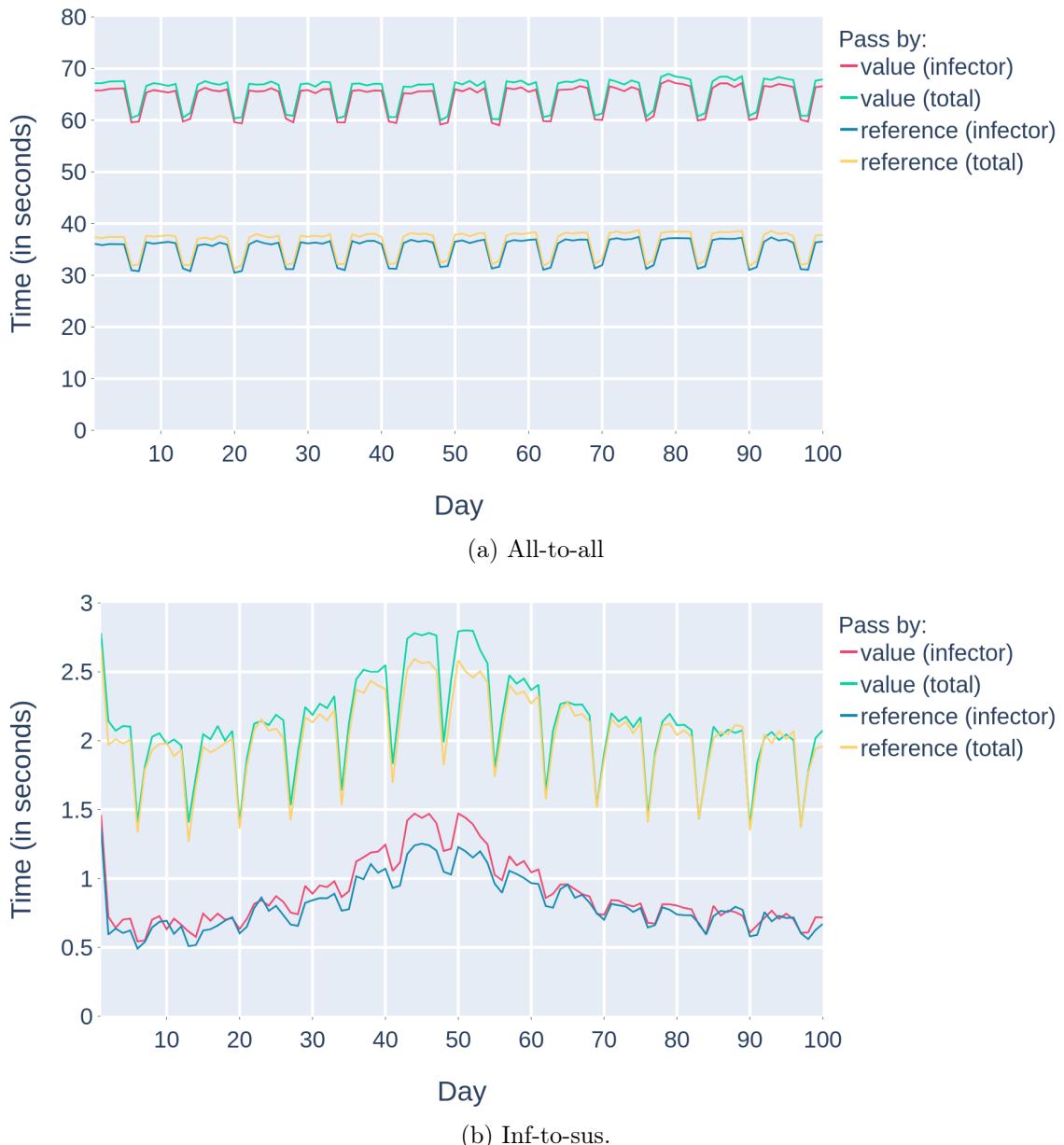


Figure 3.24: Comparison of the infector and total runtimes of passing the population by value and by reference without parallelization. Simulation run on 11M without holidays.

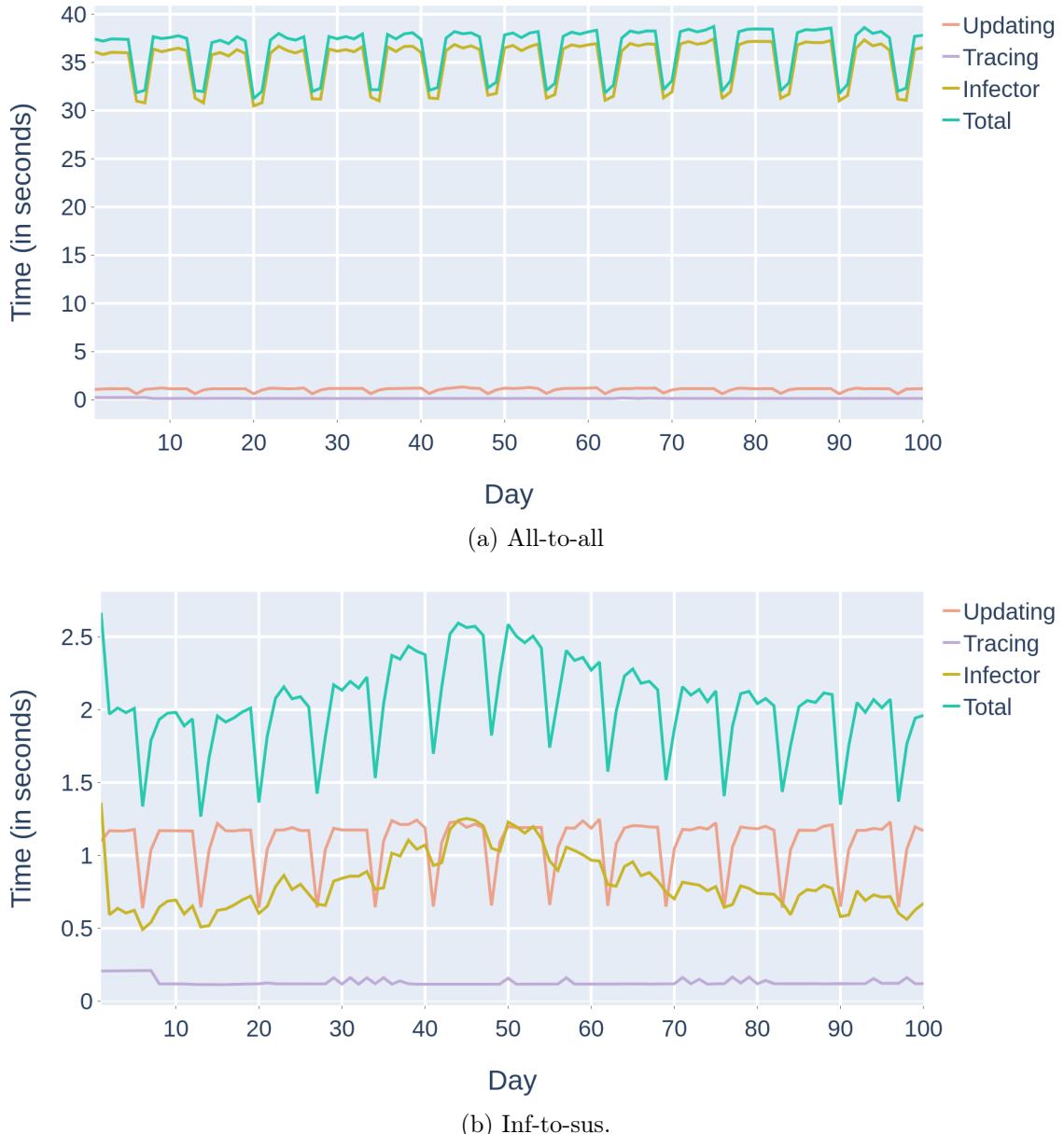


Figure 3.25: Section times per day, in seconds, for every section, including the total times for simulating a day using the pass by reference optimisation. Simulations run on 11M without holidays using 1 thread.

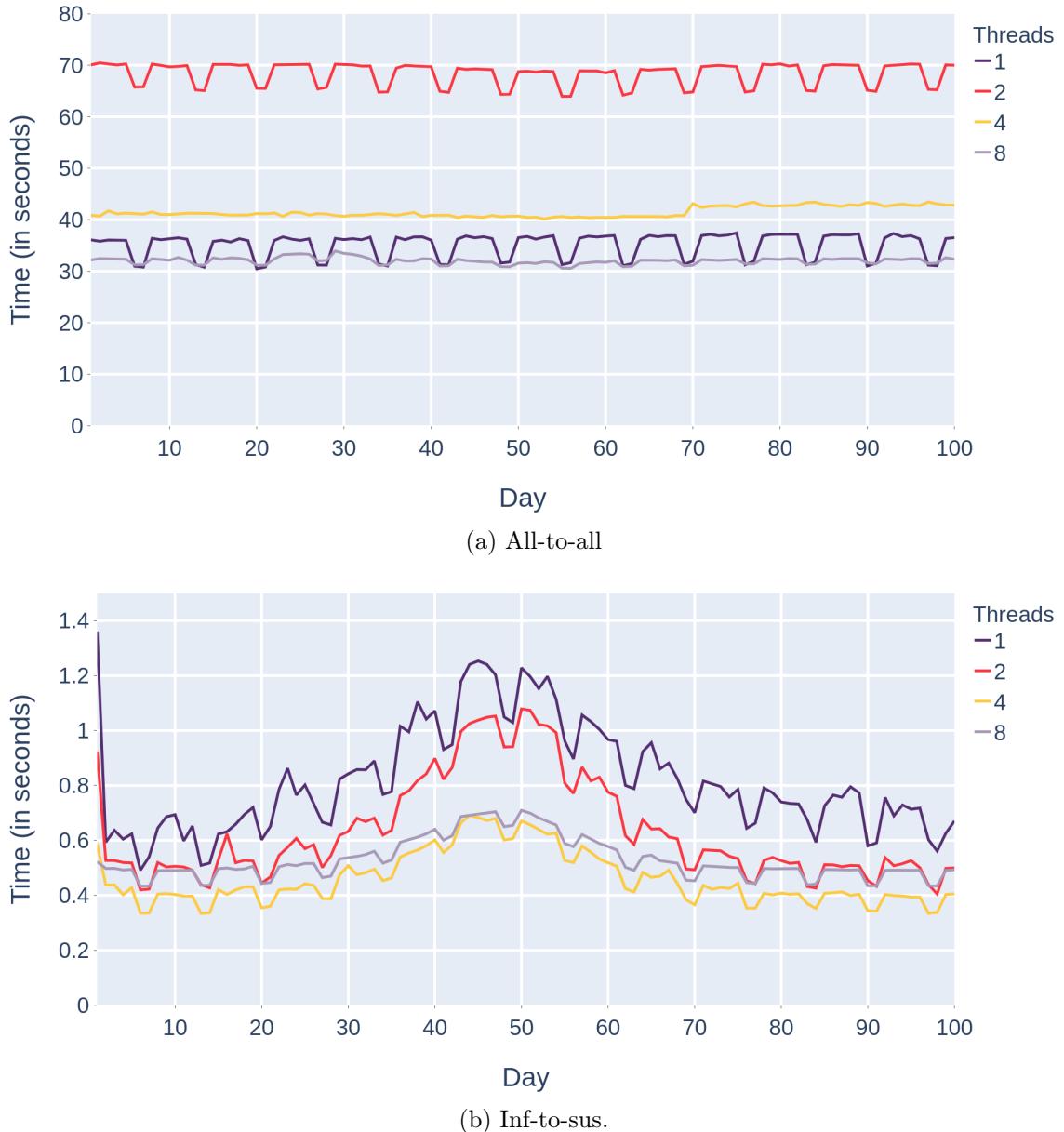


Figure 3.26: Time per day for the infector that passes the *shared_ptr* population by reference, using different number of threads. Simulation run on 11M without holidays.

	Infector	Total
pass by value	64.33	65.56
pass by reference	35.08	36.31
speedup	1.83	1.81

(a) All-to-all

	Infector	Total
pass by value	0.89	2.12
pass by reference	0.82	2.04
speedup	1.09	1.04

(b) Inf-to-sus.

Table 3.5: Average runtime (in seconds) per section of Stride before and after the optimisation with passing by reference. Simulations run on 11M without holidays and 1 thread.

	Updating	Tracing	Infector	Total
all-to-all	1.08	0.15	35.08	36.31
inf-to-sus	1.09	0.13	0.82	2.04

Table 3.6: Average runtime (in seconds) per section of Stride with the passing by reference optimisation. Simulations run on 11M without holidays and 1 thread.

Threads	Method	Infector	Total
1	by value	64.33	65.56
	by reference	35.08	36.31
	speedup	1.83	1.81
2	by value	227.93	229.50
	by reference	68.40	70.13
	speedup	3.33	3.27
4	by value	197.04	198.85
	by reference	41.50	43.35
	speedup	4.75	4.59
8	by value	187.15	189.00
	by reference	32.03	33.82
	speedup	5.84	5.89

(a) All-to-all

Threads	Method	Infector	Total
1	by value	0.89	2.12
	by reference	0.82	2.04
	speedup	1.09	1.04
2	by value	0.83	2.48
	by reference	0.63	2.43
	speedup	1.32	1.02
4	by value	0.69	2.49
	by reference	0.46	2.24
	speedup	1.50	1.11
8	by value	0.76	2.57
	by reference	0.53	2.33
	speedup	1.43	1.10

(b) Inf-to-sus.

Table 3.7: Comparison of the average runtimes (in seconds) per section before and after the pass by reference optimisation, depending on the number of used threads. Simulations run on 11M without holidays.

Chapter 4

Sampling

The optimised version of Stride from the previous chapter will be our new standard from now on. We also discussed that the most room for improvement lies in the infector algorithm of our model when using all-to-all and that we primarily focus on Stride without parallelization. Our previous optimisation was the result of a solution for an implementation issue, for which programming expertise is required to notice and solve it. This chapter presents an improvement by redesigning the infector algorithm, which requires less detailed programming knowledge to understand.

4.1 General idea

We learned that the all-to-all algorithm calculates contacts and transmissions in a contact pool, by comparing every individual with every other individual inside the pool. Algorithm 1 showed that this is done with a double for-loop: iterate over the pool members and for every person in this iteration, also iterate over the remaining members. In computer science, when we want to indicate the time it takes for an algorithm or program to run, we use time complexity. This describes an estimation of the amount of operations that are performed. If we now consider the calculation of contact and transmission between two individuals as one operation, all-to-all has time complexity $\mathcal{O}(n^2)$ with n being the pool size. This means that the larger pools have a quadratic times bigger impact than the small ones. Of course these operations cost more time when there is contact than when there is not, but this gives us a general indication.

Example 4. A pool with 10 people would need $10^2 = 100$ operations, while a pool that consists of 100 people requires $100^2 = 10,000$ operations.

All-to-all uses the double for-loop so it can compare every pair of individuals and determine if they have contact with each other and if they transmit the disease. The quadratic complexity of this algorithm is thus caused by the need to compare everyone with everyone inside a pool. When two people have contact, the model registers this and continues to calculate the transmission. If two people do not have contact, they are ignored and nothing happens or gets registered about the pair. These calculations can be seen as 'wasted' time because they do not contribute anything to the simulation or results. The idea behind our optimisation is to minimize this wasted time by only doing the calculations that are

necessary.

Our approach to limit the useless calculations is by taking a random sample of the people in a pool for which an individual is guaranteed to have contact with. Then, we only compare the individual with the people in the sample to calculate transmissions. This algorithm would need to iterate over the pool members only once, which would result in time complexity $\mathcal{O}(n * k) \approx \mathcal{O}(n)$ with n being the pool size and k the sample size. It would thus in theory have a linear time complexity which is much more efficient than the original all-to-all with quadratic time complexity.

In order for us to use this approach, we need a way to determine the sample sizes. Consider the all-to-all algorithm where we want to calculate the contacts for an individual. We loop over the pool members and calculate for every person if there is contact, which results in yes or no. If we now look at this from another point of view, every contact calculation is a Bernoulli trial that can result in a success or failure where the probability of success equals the contact probability [32]. This means that for one individual the algorithm needs to compute multiple Bernoulli trials, which in turn can be seen as a binomial distribution for which the definition is the following [33]:

"The binomial distribution refers to the number of successes in n independent trials with a common success probability p in each."

This applies to our approach where n is the number of people with whom we compare an individual and, as we already said, p is the contact probability. Thus, if these two variables are known, we can calculate for an individual the sample size $k \sim B(n, p)$. However, the issue with this is that the probability of two people having contact, is based on those two people. This means that we still would need to calculate the contact probability for every pair of individuals and be back to square one. The next sections will explain how we implemented our approach despite this obstacle.

4.2 Preliminary insights

Before we start to discuss and evaluate improvements, we continue to examine the performance for the purpose of optimising the all-to-all infector. Therefore, the data, and figures in this section only contain results from Stride simulations using all-to-all without parallelization.

4.2.1 Infector runtimes

Since our focus lies on the all-to-all infector, it is in our best interest to fully understand how the infector has such a big impact as we saw in Section 3.5.

Pool size runtimes

In the previous section we explained how the original algorithm, in theory, should have a quadratic time complexity in function of the pool size. Figure 4.1 shows the average execution time of the all-to-all infector per pool size, where we can clearly see the quadratic

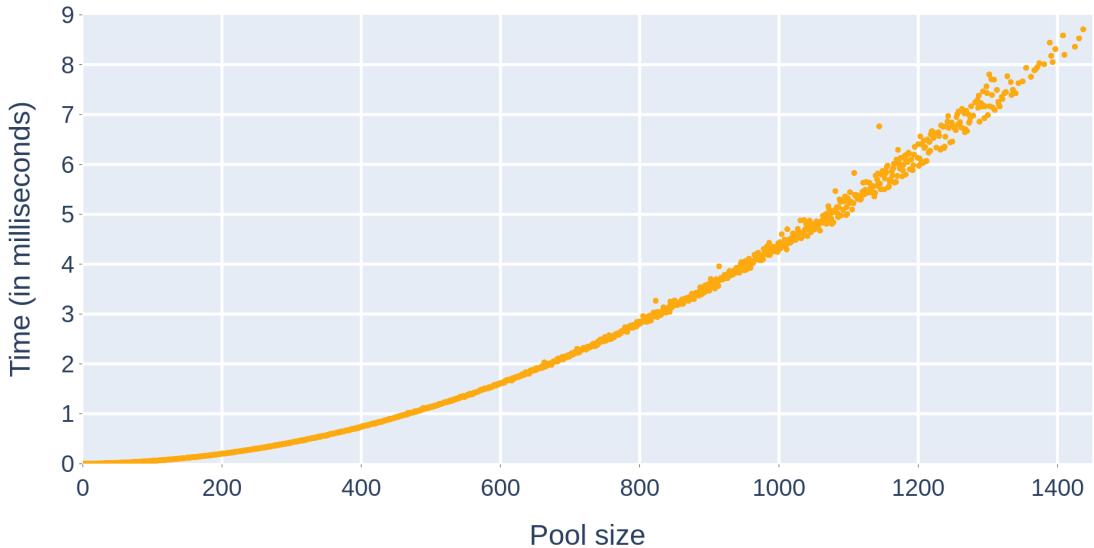


Figure 4.1: Average infector runtime (in milliseconds) per pool size using all-to-all. Simulation run on 11M without holidays using 1 thread.

curve that we talked about. Thus larger pool sizes have a much greater effect on the simulation time than the smaller ones. However, as we discussed in Section 3.3.2, there are far more pools that contain only a small amount of people.

The average runtime per pool size gives us a representation of how the pool size affects the infector, but they do not illustrate the actual infector runtimes in a simulation. As we know, contact pools are only active on particular days depending on the contact pool type. This results in different runtime results if we would measure only one simulation day. In order to give the best representation of the infector in a simulation, we compute the total runtimes for the infector for seven consecutive days without any holidays. The averages of these combined totals give us the best general indication of how pool sizes impact the infector time in a simulation. These results are displayed in Figure 4.2 and indicate that the small pools count for only a very small percentage of the total time it takes to calculate the contacts and transmissions for all the pools.

Pool type runtimes

Let it be noted that Figures 4.1 and 4.2 only present the results of measuring the time it takes for the all-to-all algorithm to calculate the contacts and transmissions for a pool with microsecond precision. The graph of Figure 4.2 reminds us of the pool size distributions of the primary and secondary community contact pools, which were discussed in Section 3.3.2, but we cannot say this with certainty. In order to get the most complete understanding of the infector runtimes, we also need to know how the different pool types influence the simulation runtime.

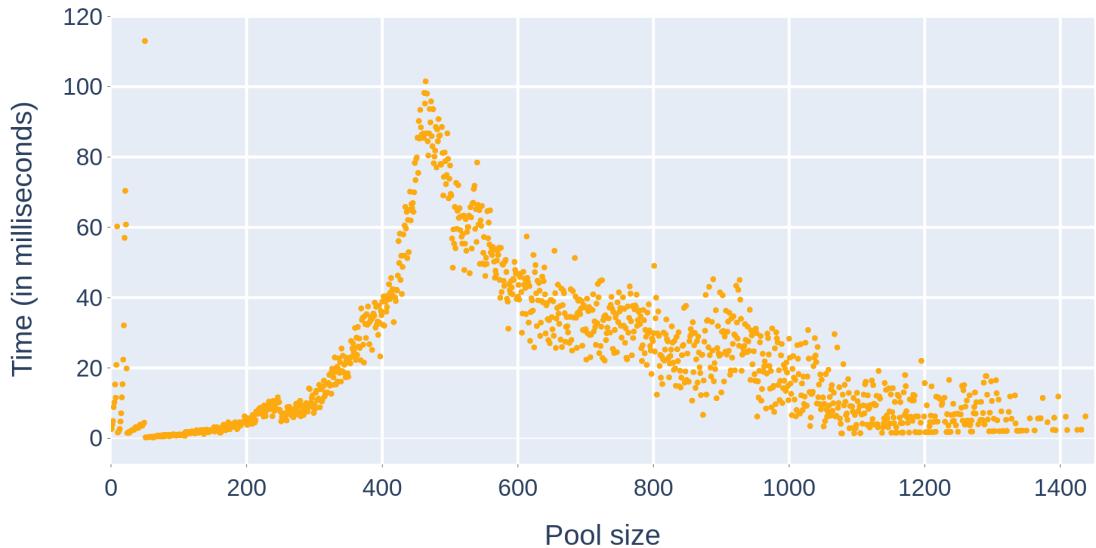


Figure 4.2: Total infector runtime (in milliseconds) per pool size using all-to-all on a day, based on the average of a regular 7-day week. Simulation run on 11M without holidays using 1 thread.

Figure 4.3 displays these infector runtimes for every pool type and makes a distinction between two different measurements per pool type. *Algorithm only* represents the total time spent in the infector algorithm for all the pools of a pool type. *Total time* gives a more complete overview of the simulation time per pool type, by measuring the time it takes to iterate over all of the pools of a certain type and call infector on them. This distinction needs to be made, because otherwise we could get the wrong impressions of how the time is distributed. For example, the household pools only spend 0.03 seconds in total in the algorithm, but it takes our model 1.72 seconds to do these calculations for every household. This demonstrates that there is definitely some overhead that we cannot change, which is not surprising since Stride needs to iterate over almost five million households every day.

What is most noteworthy about these results, are the proportions between the community pools and the rest. Both of them contain only 22,000 pools, but they account for the vast majority of the total infector runtime. This proves yet again that the larger pools are the main reason the simulation has suboptimal runtimes and that, in theory, our sampling approach will greatly improve this because of its linear time complexity. The difference between the primary and secondary community is due to the way we calculate the impact of every pool, which takes the average of a regular 7-day week. On a day in which they are active, the primary community needs approximately 29 seconds and the secondary community 30 seconds. There is thus not much difference between the runtimes of those two types of contact pools.

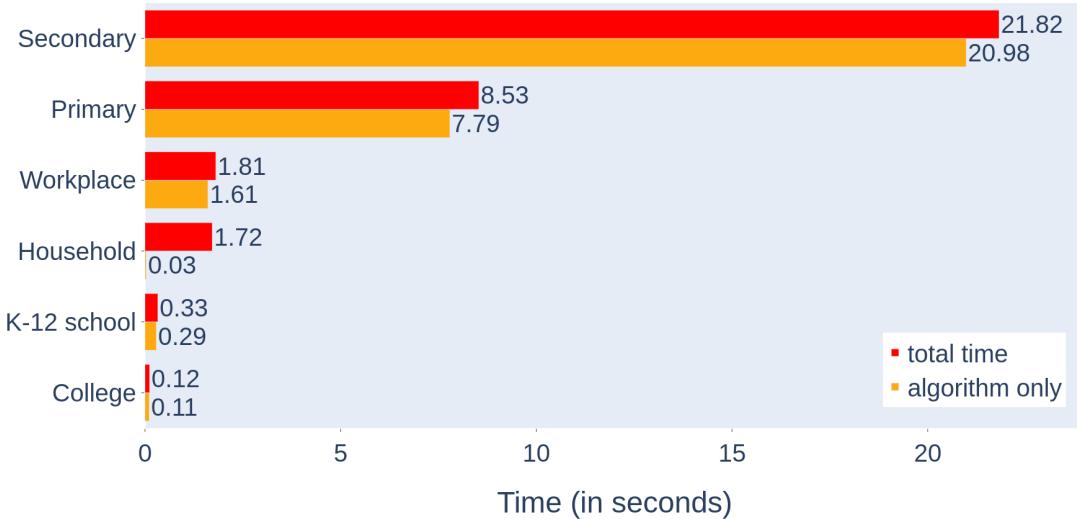


Figure 4.3: Total infector runtimes (in seconds) per pool type using all-to-all on a regular day, based on a regular 7-day week. *Total time* represents the time it takes to call infector on all of the pools and *algorithm only* represents the total time spend in the infector for a pool type. Simulation run on 11M without holidays using 1 thread.

4.2.2 Reversed contact vector

Stride has a built-in feature designed to assist the programmers when developing. It tests the program on the basis of different parameters to see if the results, for example the number of infected people, are correct. Because a lot of randomness is involved in a simulation, these results can differ every time. In order to determine if the results are 'correct', they need to lie within certain margins. This tool also verifies that the results are correct when using parallelization, which is necessary to catch multithreading issues such as race conditions [34]. This has already proven its worth during this thesis, but we will see in the upcoming sections that it does not always succeed in catching erroneous results. Therefore, we introduce another tool that will act as a benchmark in the upcoming sections, which is the reversed contact vector. We know that Stride uses the contact vectors from Section 3.3.3 to calculate the contact probabilities that are being used by the infector algorithms. The reversed contact vectors are the reconstructions of those original ones and are built up from the results of a simulation.

Figure 4.4 shows the reversed contact vectors of Stride for every pool type and compares them with the original contact vectors. We can clearly see that they are not an exact match, but this is due to the adjustment factors that are taken into account when calculating the contact probabilities, as shown in Algorithm 4. The original contact vectors start from the age of one, while Stride starts from age zero, which causes the reversed vectors to shift a little bit to the left. The K-12 school and college contact pools use the school contact vector for their contact probabilities and are therefore both compared to this vector.

Figures 4.4(a), 4.4(b) and 4.4(c) show that the reversed contact rate for some ages are zero. The reason for this is the way these reversed vectors are built. The results that our tool uses, are generated by an actual Stride simulation. They are saved in a file that contains all the logged events among which the contacts, transmissions, etc. These events are saved line by line and specify the IDs of both individuals in an event, their ages, the probability that the event would happen, and many more. If every single event in a simulation would be logged, the size of the log file would be several gigabytes per day for the 600K population. To determine how much needs to be logged, a parameter can be set for the number of people that need to be tracked. These people are then randomly chosen when initialising the simulator and only their events are logged.

In order to get the best results for these reversed contact vectors, we set the number of infected people in the beginning to zero. This leads to no infections during the entire simulation so individuals would not change their behaviour because they are infected. We set the number of people that will be logged to 50,000 and only simulate 28 days for these results, which is enough to get accurate results for every pool type. A side effect of this method is that some ages for which there are not that many individuals in a particular pool type, the results may show that the contact rate is zero. This is due to the random selection of people that need to be tracked, which causes some ages to not be represented in a pool type. An example of this are the professors of a college, ages 23 to 65, for which Figure 4.4(b) shows they do not have any contacts which is not true. The other possibility for this to happen is that there are no individuals in the population that meet the criteria for a certain age and pool type. All in all the reversed contact vectors still provide enough data to evaluate the model. In the upcoming sections we will ignore the original contact rates and only use the reversed vectors from Figure 4.4 as reference point.

4.3 Iterative intervals

Our general idea is to use a sampling approach to optimise the all-to-all infector algorithm. However, as we discussed, the main issue for this is the contact probability between two individuals, which is calculated based on the two individuals their age. In order to use our approach, this issue first must be resolved. This section explains a solution for the contact probability calculations, which also happens to be a major improvement already.

4.3.1 Age intervals

Since our problem is about the calculation of the contact probabilities, this is where we should look for the solution. Algorithm 4 shows how this calculation uses the contact rates of two persons and takes the minimum of the two rates to determine the probability. If we want to take a sample from a pool with the binomial distribution we talked about, we would need a contact probability that applies to everyone in the pool. Unfortunately, this is not possible because the contact rates are different depending on age. However, if we look at the contact vectors in Section 3.3.3, we notice that the contact rates are the same for a lot of ages. This means that the contact probability is the same in a lot of

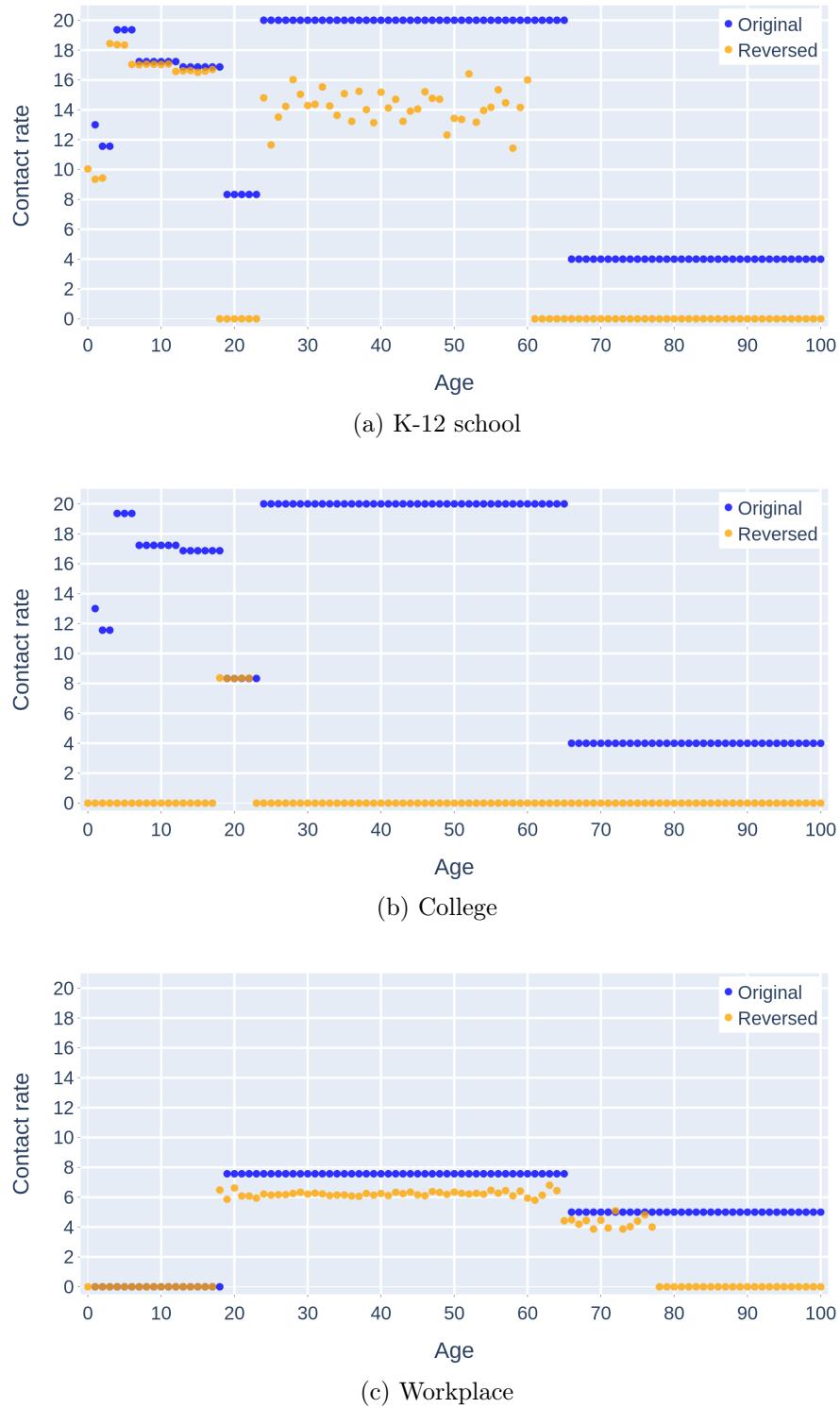
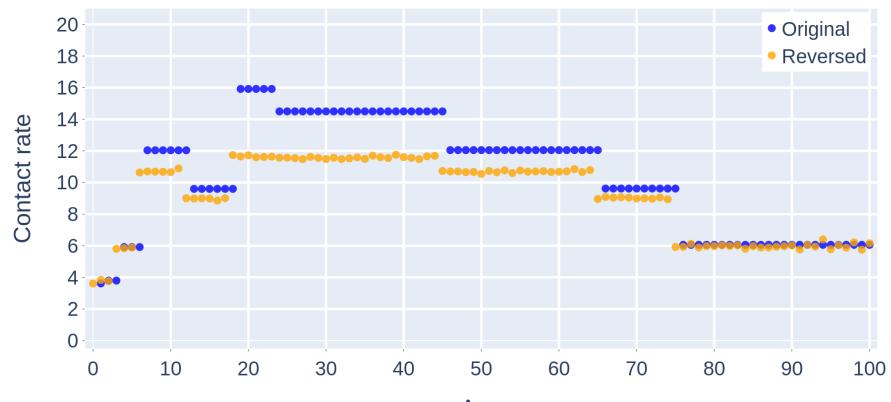
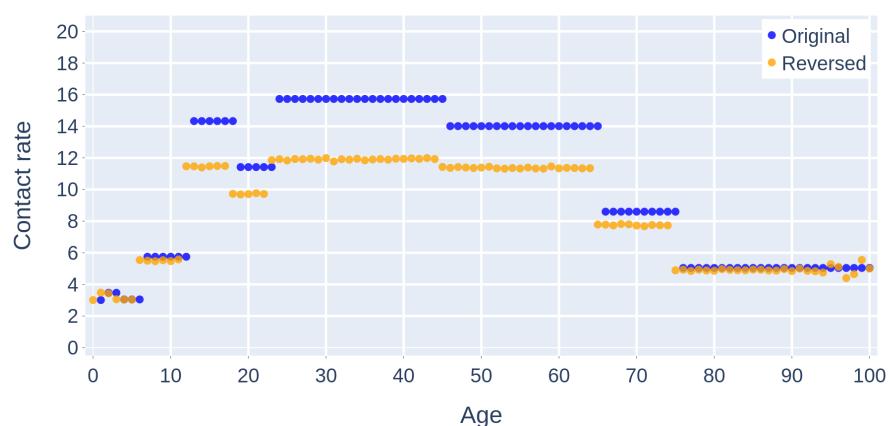


Figure 4.4: Comparison of the reversed contact rates with the original contact rates from Section 3.3.3.



(d) Primary community



(e) Secondary community

Figure 4.4: Comparison of the reversed contact rates with the original contact rates from Section 3.3.3.

cases. Thus, we could use the same contact probability for different pairs of individuals and still get correct results. Following this logic, we split up the contact pools in groups, where everyone in a group has the same contact rate. Then, if our model were to calculate the contacts for a pool, it would only need to calculate one contact probability for every pair of groups. Although the infector would still have a quadratic time complexity since everyone still gets compared with everyone, but it would reduce the runtime by reducing the number of contact probabilities that need to be calculated.

We now need to determine for every pool type how the pool members will be divided in groups. For this we look at the contact vectors from Section 3.3.3 to decide the ages with the same contact rate. Because the ages with the same contact rate lie in age intervals for every pool type, we can simply use these age intervals to represent a 'same contact rate group'. Table 4.1 shows the age intervals for every contact pool type with their corresponding contact rates. The first age intervals for the primary and secondary community have two contact rates, because age one has a different rate (the upper one) than ages two and three. Since these rates are almost the same, we choose to not split them up. However, for the school contact type we have a separate interval for age one, because its rate is much more different. Another reason for splitting these up or not, is the way these ages are distributed in the pools. In a school contact pool, almost everyone except the teacher has the same age, while a community pool can contain all the possible ages. This results in less intervals for the community pools, which will be beneficial for our sampling approach in the next sections.

4.3.2 Implementation

Now we can use the age intervals to implement our improved algorithm. Algorithm 6 presents the pseudo code for the sorting of pool members in these different intervals, which also returns the number of members in every interval. This sorting function will also be used for the upcoming approaches in the next sections. The implementation of our approach that uses the age intervals is presented in Algorithm 7. The household pools still use the original all-to-all algorithm, since they contain maximum six persons. The rest of the pools use our intervals approach, where every intervals gets compared with every other interval including itself. For every pair of intervals, only one contact probability is calculated. Then, the algorithm iterates over both intervals and uses the contact probability for every pair of members from two intervals to see if they have contact and transmit the disease. Because this algorithm still compares everyone with everyone in a pool by iterating over the intervals and their members, we will refer to this all-to-all infector approach as *iterative intervals* from now on.

4.3.3 Correctness

Obviously, we need to evaluate our approach and see if our model still produces correct results. For this we use the techniques discussed in Section 4.2.2, which are the built-in testing tool and the reversed contact vector. The testing tool only tells us if the results are valid, which is the case. We then continue to compute the reversed contact rates, for which the results are displayed in Figure 4.5. They show that the iterative intervals approach

#	Age interval	Rate
1	[1]	13.00
2	[2, 3]	11.56
3	[4, 6]	19.36
4	[7, 12]	17.23
5	[13, 18]	16.87
6	[19, 23]	8.33
7	[24, 65]	20.00
8	[66, ∞ [4

(a) School (K-12 school and college).

#	Age interval	Rate
1	[0, 18]	0
2	[19, 65]	7.57
3	[66, ∞ [5.00

(b) Workplace

#	Age interval	Rate
1	[0, 3]	3.62
		3.80
2	[4, 6]	5.92
3	[7, 12]	12.04
4	[13, 18]	9.60
5	[19, 23]	15.92
6	[24, 45]	14.50
7	[46, 65]	12.05
8	[66, 75]	9.62
9	[75, ∞ [6.06

(c) Primary community.

#	Age interval	Rate
1	[0, 3]	3.01
		3.47
2	[4, 6]	3.05
3	[7, 12]	5.75
4	[13, 18]	14.33
5	[19, 23]	11.42
6	[24, 45]	15.73
7	[46, 65]	14.01
8	[66, 75]	8.60
9	[75, ∞ [5.04

(d) Secondary community.

Table 4.1: Age intervals for the different contact pool types with their corresponding contact rates.

Algorithm 6 Pseudo code of the function that sorts the members in age intervals.

Input: $P_1 \dots P_N$, type ▷ All members of the pool and type of pool
Output: $P[]sorted$, $S[]$ ▷ Sorted members and interval sizes

```

1:  $intervals[ ] \leftarrow AGE\_INTERVALS(type)$            ▷ Maximum ages per interval (ascending)
2:  $N_{intervals} \leftarrow SIZEOF(intervals[ ])$              ▷ Number of intervals
3:  $i_{unsorted} \leftarrow 1$                                 ▷ Index of first unsorted member
4:  $S[ ] \leftarrow [N_{intervals}]$                           ▷ Interval sizes, zero at start

5: for  $i_{interval} \leftarrow 1$  to  $(N_{intervals} - 1)$  do          ▷ Iterate over the intervals
6:    $age_{interval} \leftarrow intervals[i_{interval}]$            ▷ Maximum age of the interval
7:   for  $i_{member} \leftarrow i_{unsorted}$  to  $N$  do           ▷ Iterate over the unsorted members
8:      $age_{member} \leftarrow AGE(P[i_{member}])$ 
9:     if  $age_{member} \leq age_{interval}$  then
10:      if  $i_{member} > i_{unsorted}$  then
11:        SWAP( $P[i_{unsorted}], P[i_{member}]$ )
12:       $i_{unsorted} ++$ 
13:       $S[i_{interval}] ++$ 
14:  $S[N_{intervals}] \leftarrow N - i_{unsorted} + 1$            ▷ Remaining unsorted members

15: return  $S[ ]$ 

```

has almost identical reversed contact rates as the original all-to-all algorithm. Therefore, we conclude that iterative intervals is a valid all-to-all algorithm.

4.3.4 Performance

At last, we need to examine the iterative intervals approach to see if it is indeed an optimisation. The first performance test that we will discuss is the runtime of the infector, which is the vast majority of the total runtime in an all-to-all simulation. Figure 4.6 shows the infector runtimes of the iterative intervals approach compared to the original algorithm, which clearly indicates that our new approach is a big improvement. Table 4.2 confirms that we achieve an average speedup of 1.85 on the infector and 1.80 on the total simulation runtime.

Next to the total runtime, we are also interested in how our optimisation affects the infector runtime regarding the pool type and size. In Section 4.2.1 we showed how the average infector runtime has a quadratic curve in function of the pool size. Because the infector algorithm now works different depending on the pool type, we need to examine the runtimes separately for every pool type. Figure 4.7 shows these average infector runtime results of the iterative intervals approach compared to the original algorithm. Here we can see that our new approach still has a quadratic curve, but that it becomes increasingly faster the larger the pool. The differences for the K-12 school and college pool types are very little and will be discussed more in Section 4.4.3.

Algorithm 7 Pseudo code of the all-to-all iterative intervals infector.

Input: $P_1 \dots P_N$, $type$ ▷ All members of the pool and type of pool

```

1: if  $type = household$  then
2:   original all-to-all ▷ Algorithm 1
3: else
4:    $S[] \leftarrow \text{SORT\_INTERVALS}(P[], type)$  ▷ Algorithm 6
5:    $N_{\text{intervals}} \leftarrow \text{SIZEOF}(S[])$  ▷ Number of intervals

6:   for  $interval_1 \leftarrow 1$  to  $N_{\text{intervals}}$  do ▷ Iterative intervals
7:     for  $interval_2 \leftarrow interval_1$  to  $N_{\text{intervals}}$  do ▷ Iterate remaining intervals
8:        $C_{prob} \leftarrow \text{CONTACT\_PROBABILITY}(P[interval_1], P[interval_2])$  ▷ Algorithm 4

9:       for each member  $P_1$  in  $interval_1$  do
10:         if  $P_1$  not in pool then
11:           continue to next  $P_1$ 
12:         for each member  $P_2$  in  $interval_2$  do
13:           if  $P_2$  not in pool OR  $P_1 = P_2$  then
14:             continue to next  $P_2$ 
15:             if random number <  $C_{prob}$  then
16:               register contact
17:               if  $P_1$  or  $P_2$  susceptible and other infectious then
18:                  $T_{prob} \leftarrow$  transmission probability
19:                 if random number <  $T_{prob}$  then
20:                   infect the susceptible one

```

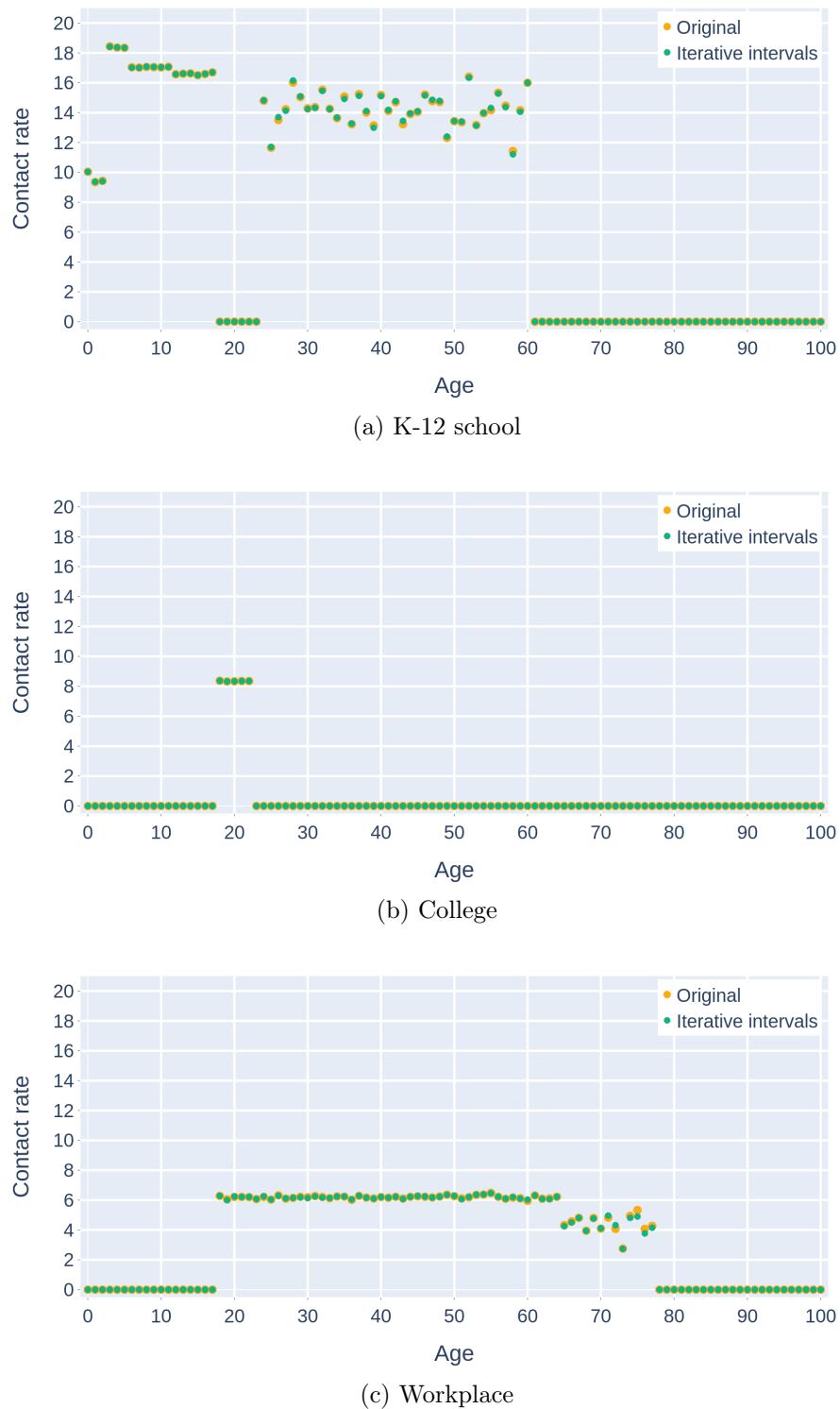
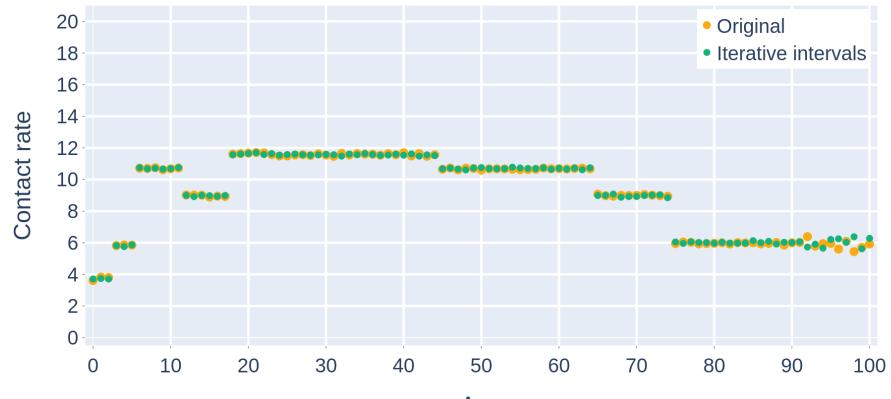
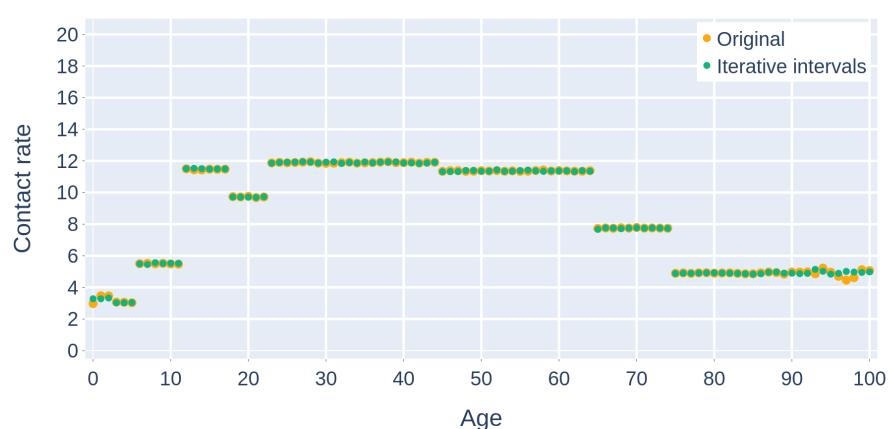


Figure 4.5: Comparison of the iterative intervals reversed contact rates with the original all-to-all reversed contact rates from Section 4.2.2.



(d) Primary community



(e) Secondary community

Figure 4.5: Comparison of the iterative intervals reversed contact rates with the original all-to-all reversed contact rates from Section 4.2.2.

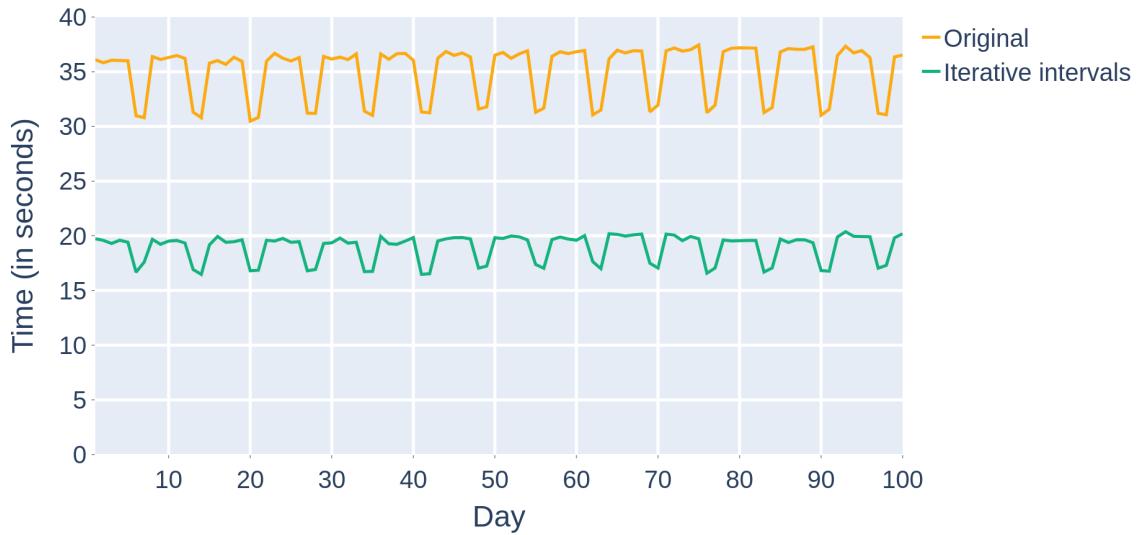


Figure 4.6: Comparison of the infector runtimes between the original and iterative intervals approaches.

	Infector	Total
original	35.08	36.31
iterative intervals	18.92	20.16
speedup	1.85	1.80

Table 4.2: Comparison of the average daily runtimes between the original and iterative intervals approach.

In Section 4.2.1 we discussed the runtime of every pool type on an average day based on a 7-day week, which was also shown in Figure 4.3. This taught us which pool types have a larger impact on the general runtime than others. Now we are only interested in how our approach affects the actual runtime of every pool type. Figure 4.8 shows the total runtimes for every pool type on an active day using different approaches, where we can see the impact of our approach on every pool type. The optimisations are most beneficial to the community pools and the workplace, while the household now takes more time to compute. This is due to the extra if-test in Algorithm 7 and is detrimental to its runtime. Altogether, the iterative intervals approach is a major optimisation for the all-to-all infector by only eliminating the amount of contact probability calculations.

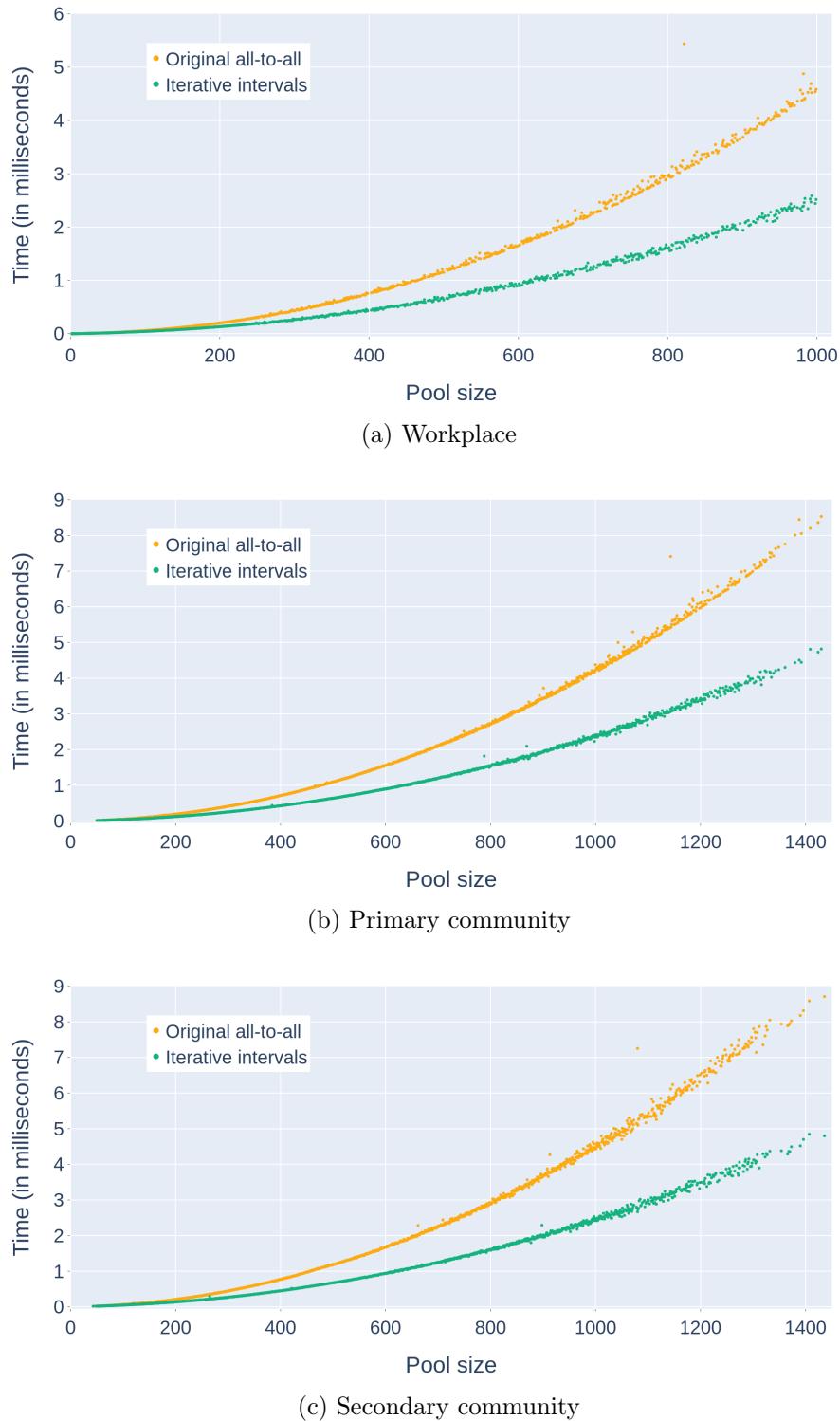


Figure 4.7: Average infector runtimes (in milliseconds) per pool size using the original all-to-all and iterative intervals. Simulation run on 11M without holidays using 1 thread.

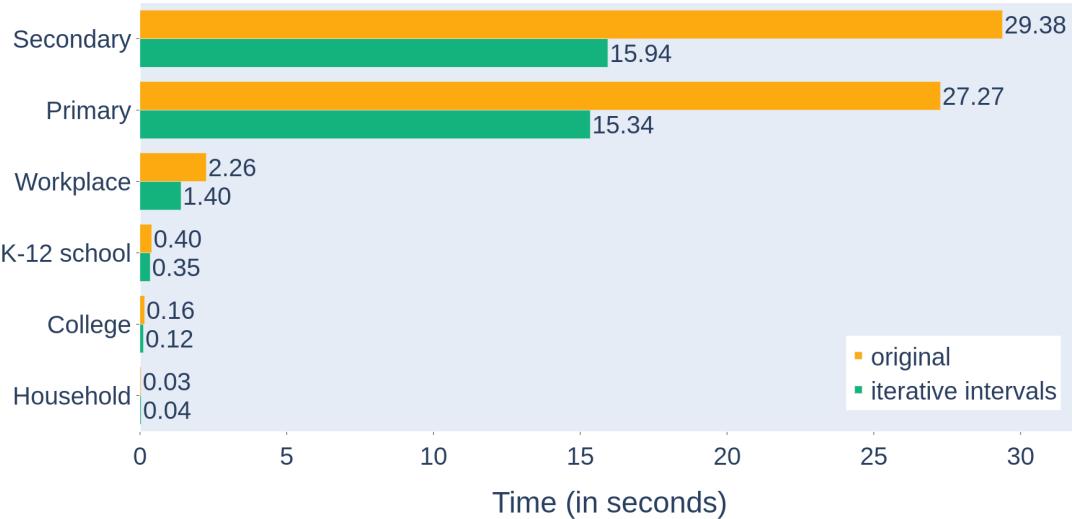


Figure 4.8: Comparison of the total infector runtimes (in seconds) per pool type on an active day. Simulation run on 11M without holidays using 1 thread.

4.4 Sampling with iteration

Recall that the issue of our general sampling idea was the contact probability. For every member in a pool we want to calculate a sample size with a binomial distribution that needs to know the pool size and the probability. What we have learned from iterative intervals can now be used to implement the sampling idea. We again divide our pool in these intervals after which the intervals get compared in pairs. Then, we can calculate the sample size, which is the number of contacts that someone will have with the members in another interval. However, this raises a question about the contact calculations for members in the same interval. Calculating if two members have contact with each other happens only once, which is also the case for calculating contacts between two intervals. Every pair of intervals is only considered once to calculate the contacts between their members. If we now want to use the sampling approach for the members inside an interval, taking a sample for every member would cause the algorithm to consider two individuals twice. To avoid this problem, we will not use the sampling approach when comparing members in the same interval, but instead use the iterative intervals approach. Calculating contact for people in the same interval will thus calculate one contact probability and then compare everyone with everyone. Since our new approach uses sampling combined with the iterative intervals, we refer to it from now on as the *sampling with iteration* approach.

4.4.1 Implementation

The pseudo code of the sampling with iteration approach is given in Algorithm 8. Just like the iterative intervals, the household type pools still use the original all-to-all method because their maximum size is six individuals, while the rest of the pools start off with

dividing their members in the age intervals. Then, all of the intervals get compared once with each other. Like we said, the contacts between the people in the same interval are calculated in the iterative intervals way, where a double for-loop compares everyone with each other and only one contact probability is calculated for all of them. The comparisons with the other intervals work by iterating over the members of one interval, where for every person P_1 a sample size gets calculated by the binomial distribution based on the number of people in the other interval, $interval_2$, and the contact probability between the members of the two intervals. This sample size indicates the number of people from $interval_2$ that P_1 will have contact with. If the sample size is zero we can just skip P_1 and continue to the next one, because there are no contacts for the current P_1 with members in $interval_2$. If the sample size is equal to the number of people in $interval_2$, it means that P_1 will have contact with everyone in the interval. When this happens, we do not need to sample and can just iterate over the entire interval and handle the contacts and transmissions between P_1 and everyone in $interval_2$. If the sample size is smaller than the size of $interval_2$, we need to determine who will have contact with P_1 . To select these contacts, we 'draw' a person from $interval_2$ based on a random number. If this randomly selected individual is present in the pool and if we have not already drawn him for the same P_1 , we register their contact and calculate the transmission as we know.

4.4.2 Correctness

Again we need to make sure that this approach is also correct. The first step is to run the built-in testing tool to see if the results, such as the number of infected people, lie within a certain interval and that it can also be used with parallelization. The sampling with iteration algorithm passed this test, so we then continue to compute the reversed contact vectors, which are displayed in Figure 4.9. The reversed rates of the original all-to-all algorithm are very similar to the reversed rates of the sampling with iteration approach. Because this new approach passed both tests, we conclude that it is a valid all-to-all algorithm.

4.4.3 Performance

Now that we have verified the correctness of sampling with iteration, we want to know how it performs compared to our other two approaches. Figure 4.10 shows the total infector runtime per day for the different algorithms, which proves that sampling with iteration is even faster than iterative intervals. Table 4.3 reveals that our latest algorithm achieves an average total speedup of 2.50, which is due to the 2.63 average speedup in the infector.

Our general idea from Section 4.1 explains that the improvements we want to make will have the most impact on the larger pools because of the linear time complexity. Hence, Figure 4.12 shows the average infector runtime regarding the pool size for the workplace and community pools. The community pools show that our sampling with iteration approach has a somewhat linear curve, as we would expect, and that it is therefore a massive optimisation the larger the pool size. However, Figure 4.12(a) shows that the workplace pools have a quadratic curve, similar to the iterative intervals algorithm. This is because the workplace has only three different age intervals where the [0,18]-interval is empty on

Algorithm 8 Pseudo code of the all-to-all sampling with iteration infector.

Input: $P_1 \dots P_N$, type ▷ All members of the pool and type of pool

```

1: if type = household then
2:   original all-to-all ▷ Algorithm 1
3: else
4:    $S[ ] \leftarrow \text{SORT\_INTERVALS}(P[ ], \text{type})$  ▷ Algorithm 6, returns interval sizes
5:    $N_{\text{intervals}} \leftarrow \text{SIZEOF}(S[ ])$  ▷ Number of intervals

6:   for interval1  $\leftarrow 1$  to  $N_{\text{intervals}}$  do ▷ Iterative intervals
7:     for interval2  $\leftarrow \text{interval}_1$  to  $N_{\text{intervals}}$  do ▷ Iterate remaining intervals
8:        $C_{\text{prob}} \leftarrow \text{CONTACT\_PROBABILITY}(\text{interval}_1, \text{interval}_2)$  ▷ Algorithm 4

9:       if interval1 = interval2 then
10:         iterative intervals ▷ Algorithm 7
11:       else
12:         for each member  $P_1$  in interval1 do
13:           if  $P_1$  not in pool then
14:             continue to next  $P_1$ 
15:           sample  $\leftarrow \text{BINOMIAL}(S[\text{interval}_2], C_{\text{prob}})$  ▷ Binomial distribution
16:           if sample = 0 then
17:             continue to next  $P_1$ 
18:           else if sample =  $S[\text{interval}_2]$  then
19:             iterate over interval2 and register contact with everyone
20:           else
21:             drawsall  $\leftarrow 0$  ▷ Number of draws
22:             drawsgood  $\leftarrow 0$  ▷ Number of good draws
23:             drawn_members  $\leftarrow []$ 

24:             while drawsgood < sample AND drawsall <  $S[\text{interval}_2]$  do
25:                $P_2 \leftarrow$  random member of interval2
26:               if  $P_2$  in drawn_members then
27:                 draw next  $P_2$ 
28:                 add  $P_2$  to drawn_members[ ]
29:                 ++ drawsall
30:               if  $P_2$  not in pool then
31:                 draw next  $P_2$ 
32:                 ++ drawsgood
33:               register contact
34:               if  $P_1$  or  $P_2$  susceptible and other infectious then
35:                  $T_{\text{prob}} \leftarrow$  transmission probability
36:                 if random number <  $T_{\text{prob}}$  then
37:                   infect the susceptible one

```

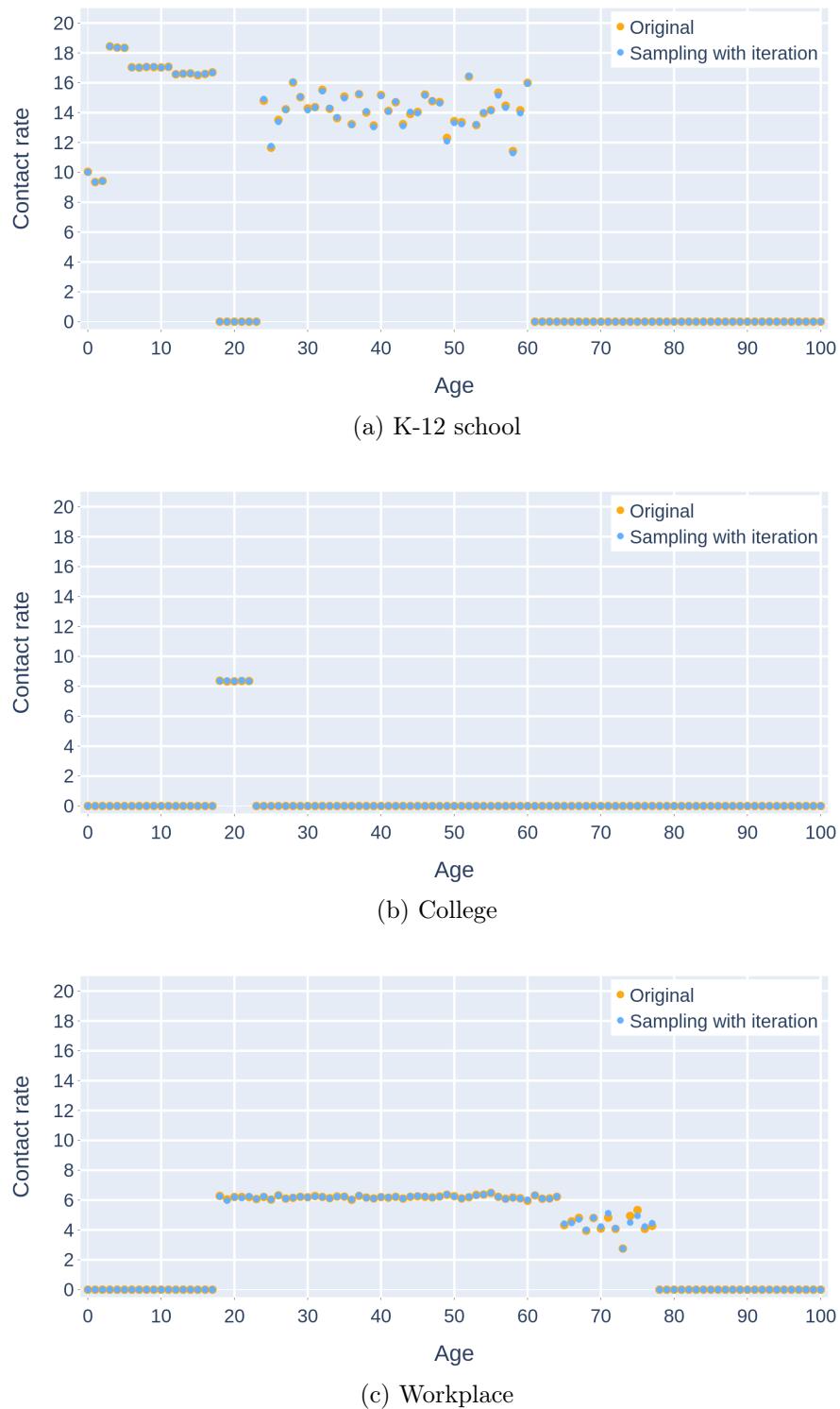
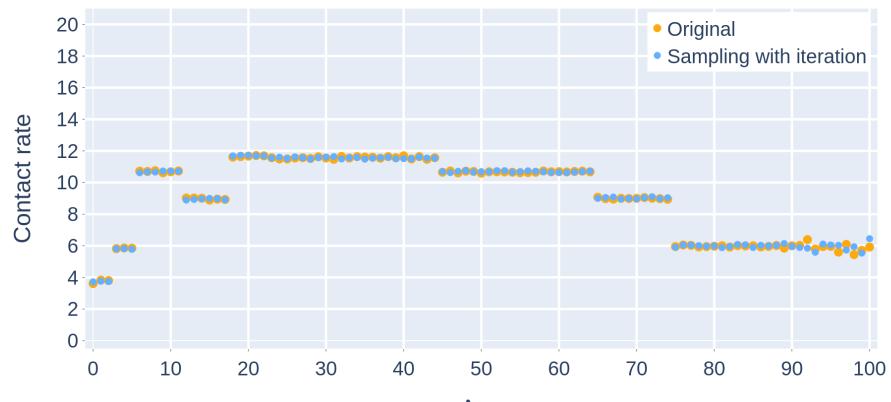
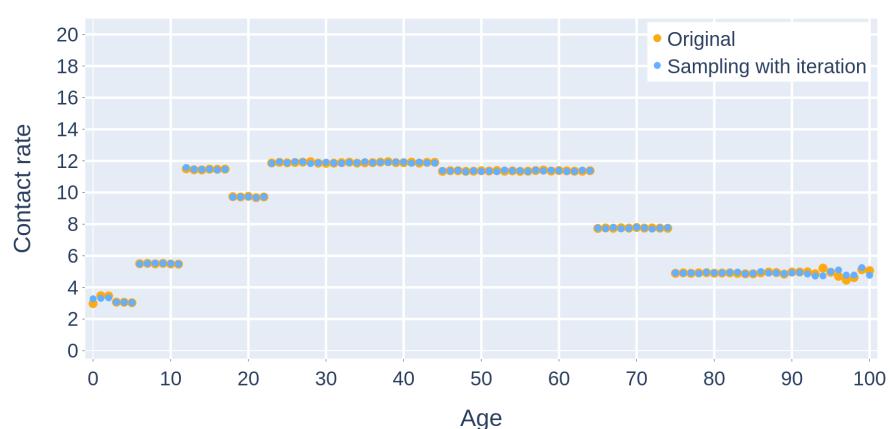


Figure 4.9: Comparison of the sampling with iteration reversed contact rates with the original all-to-all reversed contact rates from Section 4.2.2.



(d) Primary community



(e) Secondary community

Figure 4.9: Comparison of the sampling with iteration reversed contact rates with the original all-to-all reversed contact rates from Section 4.2.2.

	Infector	Total
original	35.08	36.31
sampling with iteration	13.35	14.55
speedup	2.63	2.50

Table 4.3: Comparison of the average daily runtimes between the original and sampling with iteration approach.

most occasions. Secondly, almost everyone resides in the [19,65]-interval according to the workplace age distribution in Figure 3.9. This causes workplace pools to spend most of their time calculating contacts between people in the same interval, which uses the iterative intervals method. Sampling with iteration has thus barely any effect on the workplace contact calculations. The community pools have nine different intervals that are all evenly distributed, which supports this explanation.

The impact of our optimisation for every pool type is presented in Figure 4.11, which shows that the community pools are the only ones that benefit from our new algorithm. The rest of the pools are all slower than the iterative intervals or even the original all-to-all approach. To further investigate this, a more detailed view on these pools is needed. Figure 4.13 displays the average infector runtimes only for the pools that contain 300 people or less. The primary community in Figure 4.13(b), as well as the secondary community in Figure 4.13(c), both show that the sampling with iteration has the worst results for smaller pools. The reason for this is that the algorithm has extra overhead because of the binomial distribution calculation and the random member draws. When the intervals contain only a few individuals, the overhead takes more time than simply comparing everyone with everyone like the iterative intervals method does. Only pools with around 275 people achieve the fastest time with the sampling with iteration approach, but it is already faster than the original one for pools with a size of at least 150 individuals. Figures 4.13(d) and 4.13(e) also show that the K-12 school and college pools lose speed using sampling with iteration, for which the reason is also the predominant overhead in smaller pools and intervals.

In the end we can conclude that our sampling with iteration algorithm is a major optimisation compared to the original and iterative intervals all-to-all algorithms for larger pools. The time complexity of this algorithm is hard to define, because it is not completely linear due to the contact calculations between people in the same interval which still has a quadratic time complexity. Also, a double for-loop is used to iterate over the intervals which also results a quadratic time complexity in function of the number of intervals. However, this is negligible if there are not that many intervals as is the case for our different pool types. Altogether, this new approach is more efficient the larger the pools, as long as their members can be somewhat evenly divided in multiple age intervals.

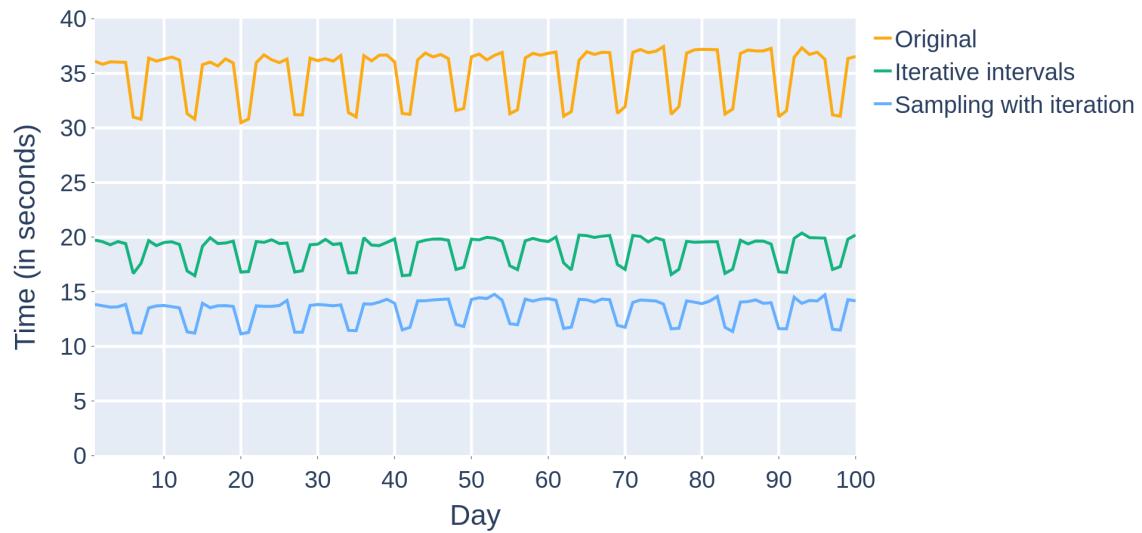


Figure 4.10: Comparison of the infector runtimes between the different approaches.

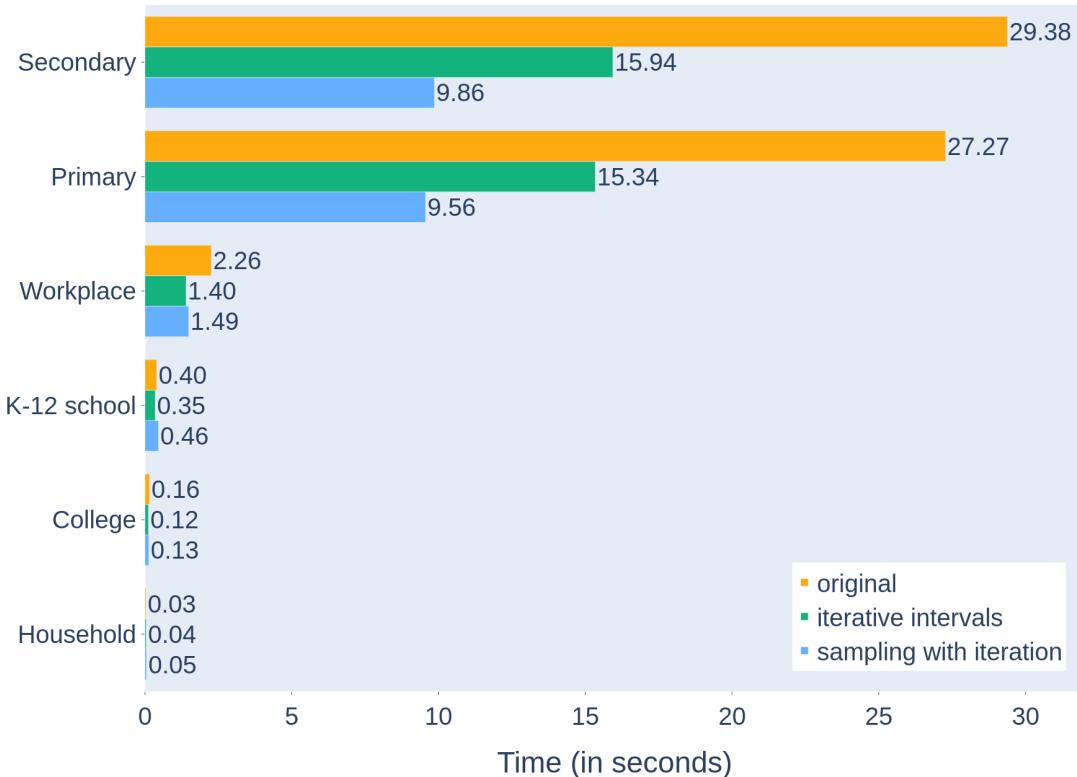


Figure 4.11: Comparison of the total infector runtimes (in seconds) per pool type on an active day. Simulation run on 11M without holidays using 1 thread.

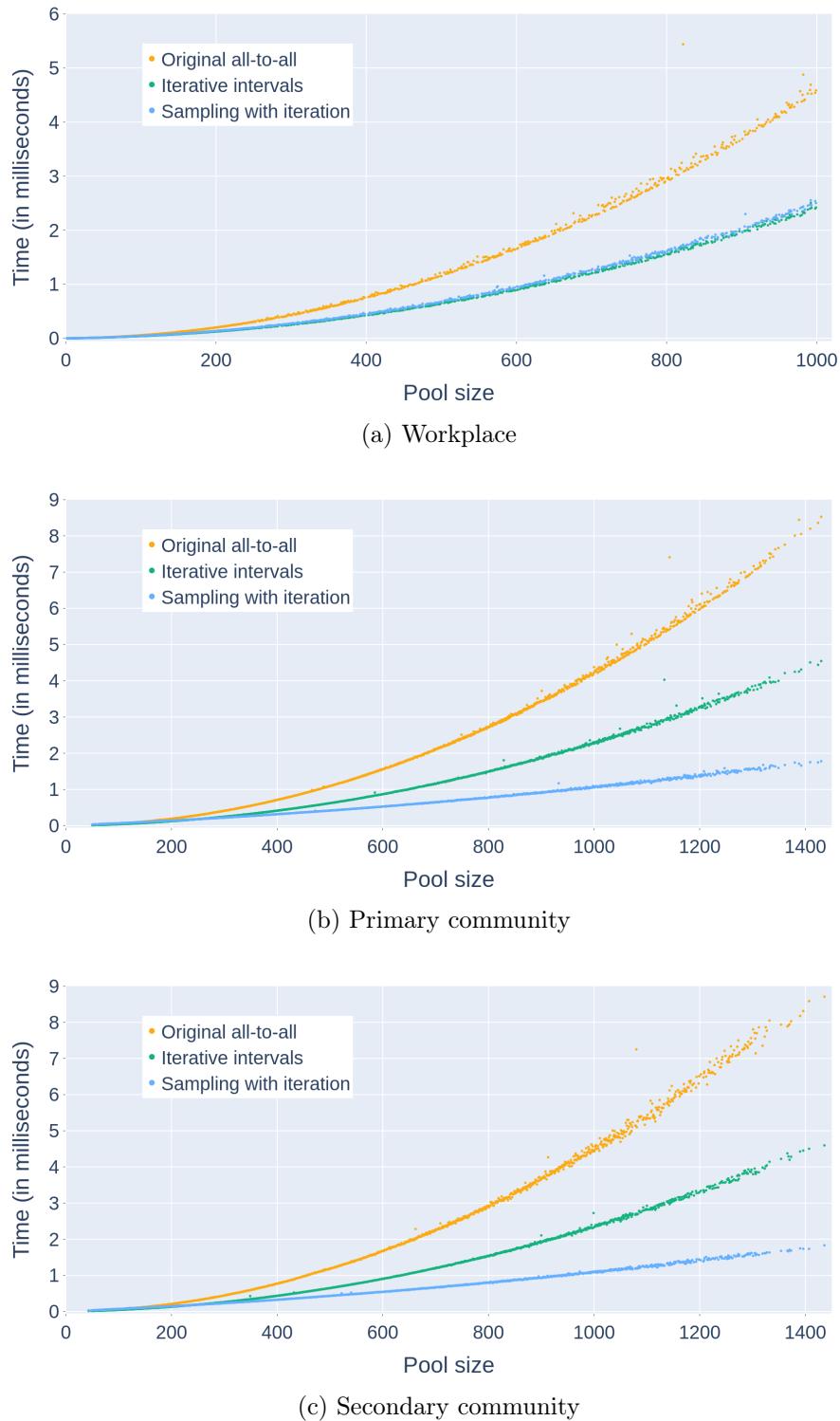


Figure 4.12: Average infector runtimes (in milliseconds) per pool size using different approaches. Simulation run on 11M without holidays using 1 thread.

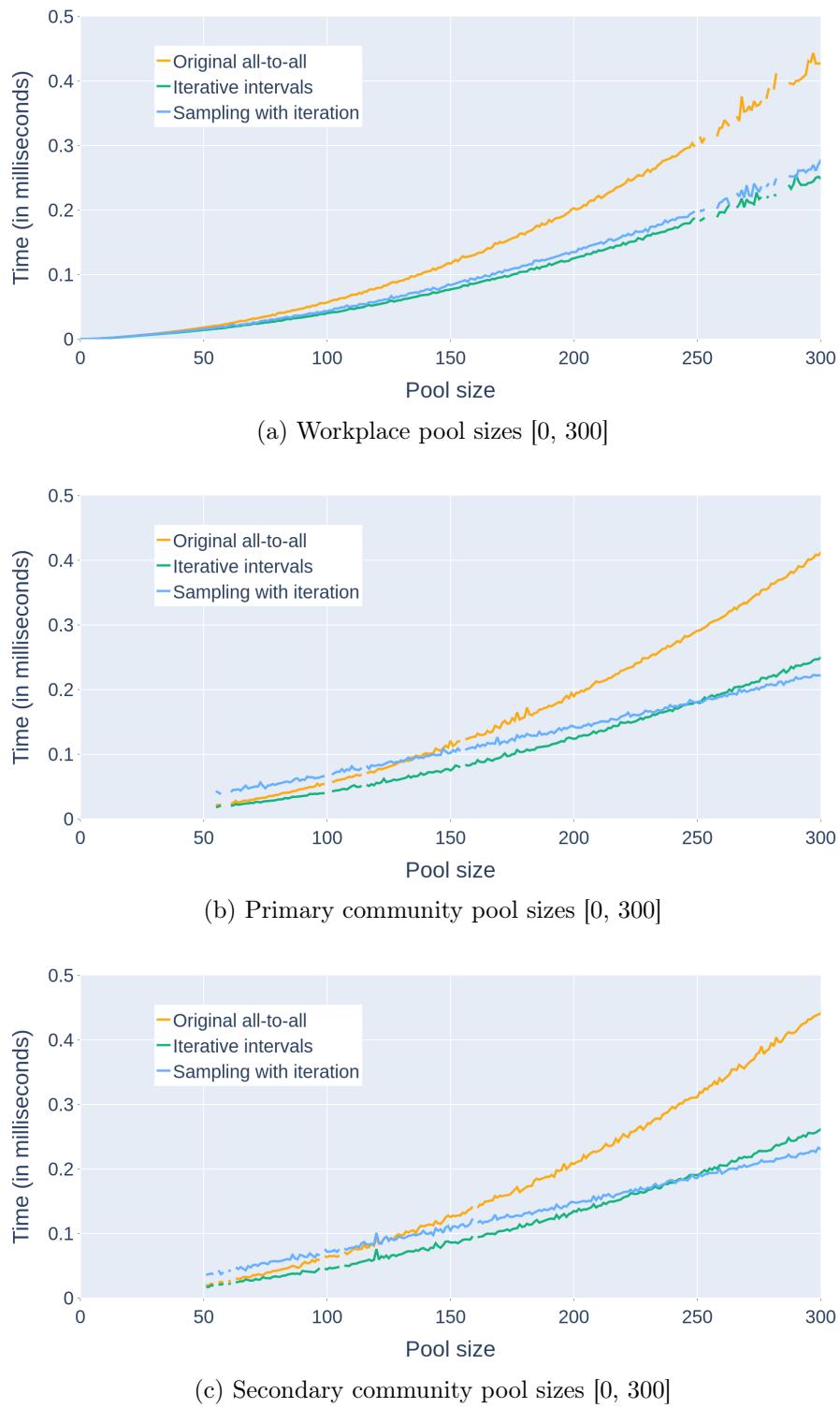
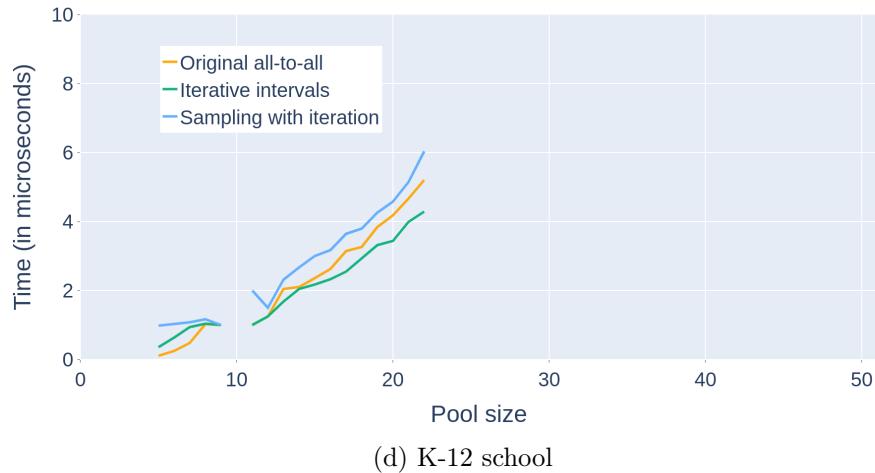
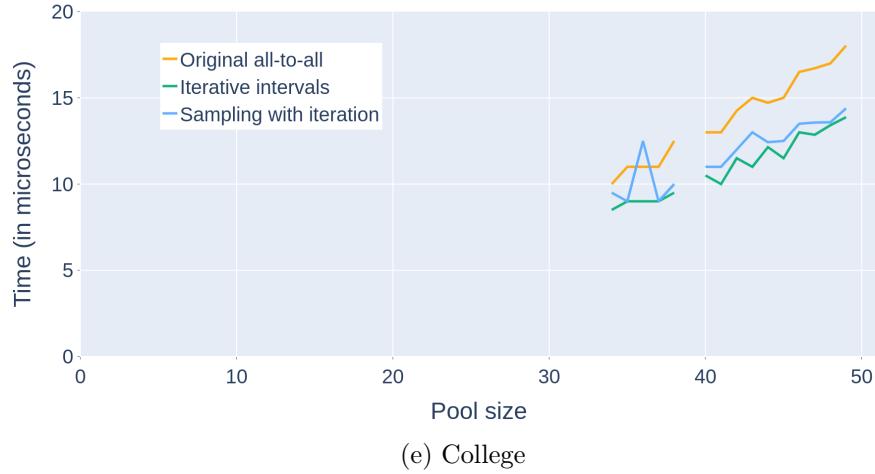


Figure 4.13: Average infector runtimes per pool size using different approaches. Simulation run on 11M without holidays using 1 thread.



(d) K-12 school



(e) College

Figure 4.13: Average infector runtimes per pool size using different approaches. Simulation run on 11M without holidays using 1 thread.

4.4.4 Observations

In the process of creating the sampling with iteration approach as is, different implementations and techniques were examined in order to create the most optimal algorithm. These variations of the actual algorithm gave insights that are worth mentioning here.

Pool size threshold

Because we want to achieve the best runtimes, we want to minimise the parts where our sampling with iteration acts slower than the rest. As we learned from Figure 4.13, the original algorithm is faster for the pools with maximum 150 people. Therefore, we adjusted Algorithm 8 so that pools with 150 individuals or less make use of the original all-to-all algorithm instead of only the household pools. After testing this theory, the results did not improve compared to only separating the household pools from the rest and were even worse most of the times. The most plausible explanation for this is *branch prediction*, which is an optimisation technique for the processor [35]. When there are multiple branches in a program, such as in an if-then-else-structure, the processor tries to guess which branch it will need to execute and commences to execute the instructions of a particular branch, which is called *speculative execution* [36]. The execution then becomes faster on a low-level, because it does not have to wait for the result of the condition, which is slow, to follow the conditional jump to the instructions. However, if the speculations turn out to be incorrect, the already performed computations are discarded and the right instructions get executed and causes the execution to slow down by a couple of clock cycles. The more mispredictions, the slower the execution of course.

Our sampling with iteration algorithm only uses the original method on household pools. Algorithm 5 showed us how the infector is called for all pools, which iterates over the pools of a particular type before continuing to the pools of the next type. Thus when the processor predicts the branch, it is only wrong when the previous pool was a household pool and the next one is a different type or vice versa. Therefore, the branch prediction technique works very well with how we implemented our algorithm. When we changed the if-test to check for pools with less than 150 people, it caused a lot of mispredictions because subsequent pools in the iteration have different sizes. The household, K-12 school and college pools never contain more than 150 individuals, thus they are not affected by this. However, the workplace and both community pools can have very large sizes, which results in a lot of mispredictions. Therefore, this overhead is the cause that implementing such a threshold based on the pool size makes the execution more time-consuming. A solution to this is to sort the pools based on their sizes when initialising the simulation, but this has not been examined because it would most likely not significantly increase performance.

Sorting at initialisation

Every non-household pool gets sorted in age intervals every time their contacts and transmissions need to be calculated. Since we use a population in which the individuals always remain the same age and the pools contain the same people, this sorting procedure can

easily be implemented at the start of the simulation in the initialisation. This has been tested, but did not result in faster execution times. Also, the simulation can also switch between all-to-all and inf-to-sus from one day to another based on what needs to be calculated. For these reasons, it is therefore best to use the sampling with iteration as originally described, which gives our method also the possibility to work with dynamic populations and pools.

One sample size

This new approach, for which the pseudo code is given in Algorithm 8, shows that a sample size gets calculated for every member P_1 when iterating over interval_1 . This means that the binomial distribution that uses a random number generator is used for every P_1 , which is a rather costly operation compared to, for example, a simple if-test in terms of execution time. Computing the sample size only once for an entire pair of intervals would thus reduce these costs. After examining this implementation, it could be concluded that it indeed produced faster results and that it passed the built-in testing tool. However, the reversed contact rates did not match the original ones as can be seen in Figure 4.14. The K-12 school, college, and workplace reversed contact vectors were very similar to the original ones because they do not contain that many different non-empty intervals. On the other hand, the primary and secondary community pools their reversed contact rates were reasonably different than the original rates. The only difference regarding our actual sampling with iteration is that there is less randomness involved due to computing only one sample size. The explanation for these results is the law of large numbers [37], which states that the outcome is closer to the expected outcome the more trials are performed. Therefore, determining only one sample size has a very big impact on the contacts between two intervals. Normally, a sample size gets calculated multiple times which thus evens out the odd results. The reason that this passed the built-in test, is because in the end, the law of large numbers is applicable because a lot of sample sizes have been calculated. The reversed contact vectors are thus essential in testing our different approaches.

Storing drawn individuals

When a sample size is determined for a member P_1 , the contacts get randomly selected one by one. Once a random contact has been chosen, the algorithm needs to check that this person did not already have contact in the same iteration with P_1 . The persons that P_1 already had contact with in the same iteration are stored in a `std::vector` [38] by their indexes. For every P_1 a new, empty vector gets created with a size that is the same as the number of people in the other interval. Searching if an element exists in such a vector has a linear time complexity in function of the size of the vector. Our algorithm already optimises this by storing the indexes subsequently, starting at the first element. Checking if an index already appears in the vector is then performed by only checking the elements of the vector that are not empty, which reduced the total infector runtime by several seconds. Another option that has been tested is by replacing the `std::vector` with a `std::unordered_set` [39], which is a hash map that has a constant time complexity for lookups. The daily total infector runtime was, however, a little bit slower than the current approach. The reason for this is that the sample sizes are very small and are rarely higher than five, which only happens

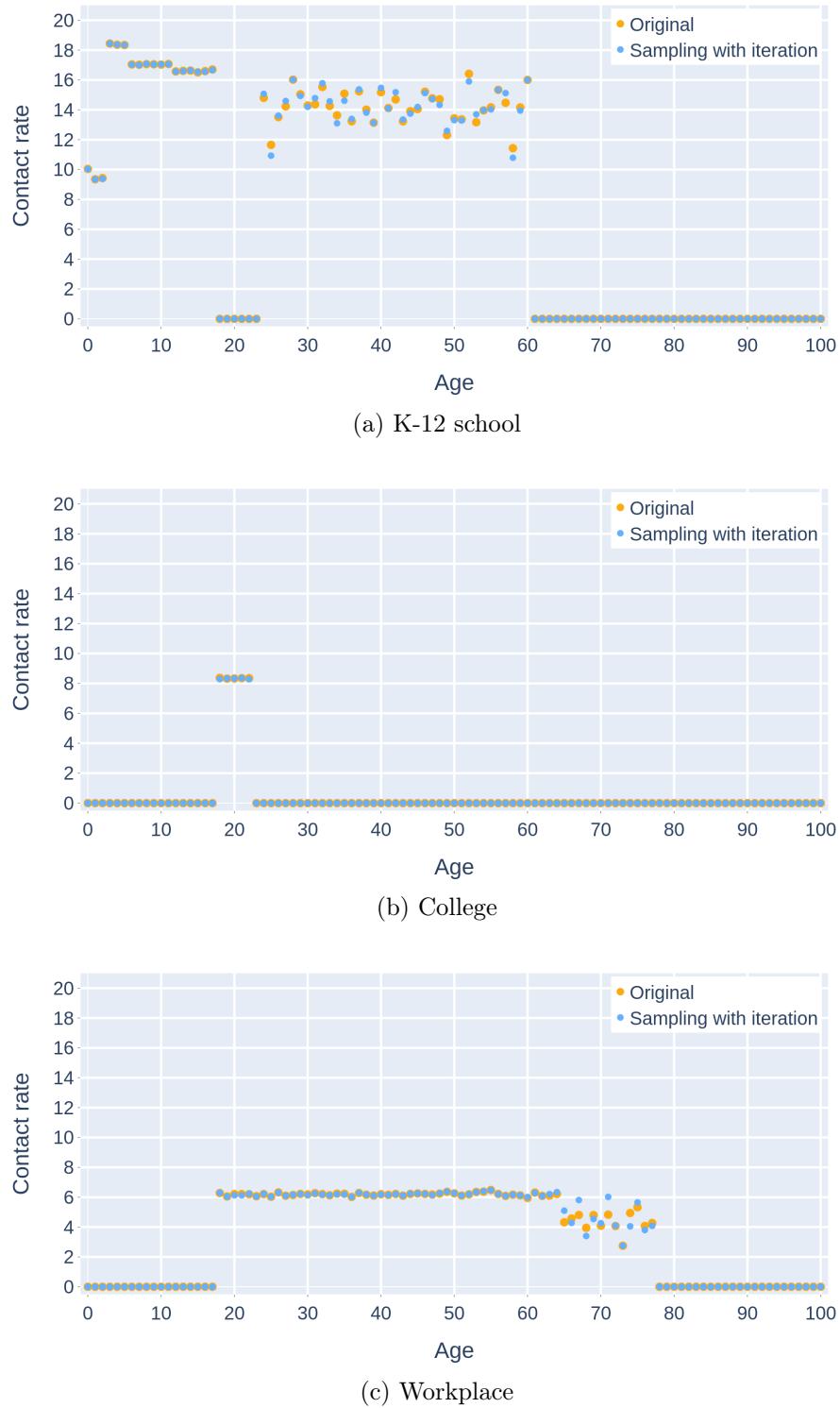
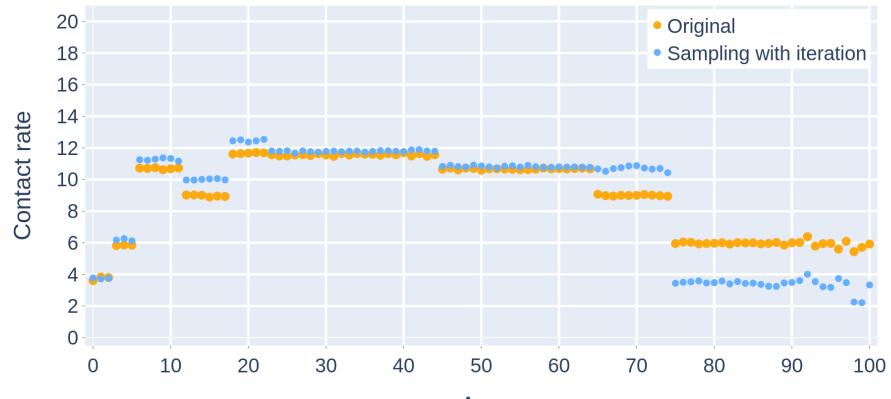
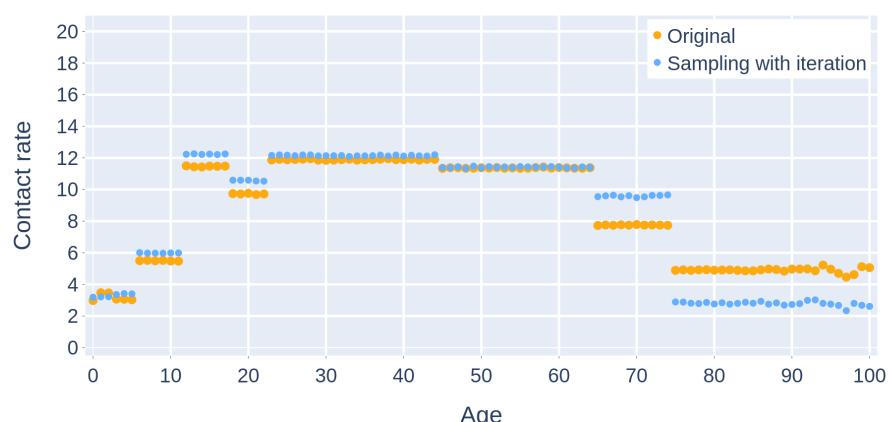


Figure 4.14: The reversed contact rates of sampling with iteration using one sample size per interval compared to the original all-to-all reversed contact rates from Section 4.2.2.



(d) Primary community



(e) Secondary community

Figure 4.14: The reversed contact rates of sampling with iteration using one sample size per interval compared to the original all-to-all reversed contact rates from Section 4.2.2.

if the interval is very large. Although the hash map has a constant lookup time, it has more overhead than the regular vector. Since most of the lookups are executed when there are zero or one indexes stored, the overhead from `std::unordered_set` causes it to be slower.

The creation of a vector also comes with some overhead, hence we tried to minimize this overhead by creating the vector only once per pair of intervals. After every P_1 this vector needs to be emptied for the next one, which also has some overhead. Examining these results indicated that emptying the vector costs more time than deleting and creating a new vector. Another option to reduce the time is by creating a smaller vector, since only a small part of it gets actually used. This is not entirely true, because it is possible for almost an entire interval to be absent due to illness, which would cause the vector to fill up if those absent members are drawn. Choosing for example halve the original size to create the vector and only increasing its size when necessary is probably also not an option, because this would cause overhead due to extra tests to handle the vector.

4.5 Full sampling

We learned from sampling with iteration that the algorithm did not improve the infector runtime for workplace pools, because most of the contact calculations were between people in the same age interval. In order to improve this, the algorithm needs an update on how to compute these same interval calculations and, since sampling has such good results, we want to use sampling for these computations. As we already discussed in the previous section, there is no straightforward way to implement this and still produce correct results. Examples 5 and 6 present ideas that have been considered while also explaining why they would not work.

Example 5. Consider an age interval with N members for which we need to calculate the contacts between the members of said interval. We could iterate over the members, starting from member 1 up to N . For every member i , we calculate the sample size based on $N - i$ members and only randomly select contacts between members $i + 1$ till N . There are two reasons that this approach would not work:

1. Starting at member 1, the rest of the members are all considered as a possible contact. If we then move on to member 2, member 1 is not considered as a contact anymore and will thus have much less contacts than member N and result in incorrect results.
2. We have learned that sampling is ineffective when the intervals are small because of the overhead that comes with it. Because we gradually decrease our 'interval' from which we draw contacts, this will cause our approach to perform worse than just comparing everyone with each other.

Example 6. Consider again the age interval with N members for which we want to calculate the contacts between its members. To prevent our interval to decrease in size, we now iterate over our members and keep the interval size the same. Every member i will thus have a sample size based on N and will select contacts from all N members in the interval. This results in everyone being considered the same amount of times. If we then look at the pair of individuals that have been considered, we see that the contact between

member i and member j has been considered twice, namely once when calculating contacts for member i and once when calculating contacts for member j . Thus, the algorithm would have calculated twice the amount of contacts that should have been calculated. In order to solve this, we simply divide the contacts by two. Unfortunately, there are also various reasons that this would not work:

1. Calculating twice the amount of contacts would of course significantly decrease the speedup.
2. When we delete half of the contacts, how do we decide which ones that need to be nullified?
3. If we already calculated contacts for i , among which j , what needs to be done if we draw i when calculating contacts for j ?
4. Implementing this, would require us to keep track of all the contacts, so that we can delete half of them at the end. Storing this and then selecting half of the contacts would create significant overhead and maybe even result in even worse runtimes.

A solution to our problem should therefore lie somewhere in the middle between these two examples. We do not want to calculate redundant contacts, because this would increase the execution time. On the other hand, only calculating the necessary contacts might be impossible to achieve with sampling. Therefore, the solution that we present might calculate the same contacts twice, but it lets us iterate over the entire interval without needing to select the 'real' contacts at the end. This is achieved by calculating the contact probability in another way, so that it can be used in the binomial distribution when computing sample sizes while always considering the entire interval as possible contacts. The method for calculating the contact probability in Algorithm 4 is slightly adjusted by replacing line 8 with the following:

$$p = \frac{2 - \sqrt{4 - 4\frac{r}{n-1}}}{2}, \text{ with interval size } n \text{ and contact rate } r$$

The reasoning behind this and the prove that it should work are presented in Appendix A3. Let it be noted that this contact probability is only used for the contact calculations in the same interval, thus comparing two different intervals still uses the original method. Because this algorithm uses sampling in all occasions, except for the household pools, it will be referred to from now on as the *full sampling* approach.

4.5.1 Implementation

Algorithm 9 shows the pseudo code of full sampling, where the only difference with the sampling with iteration approach is the case when $interval_1$ is equal to $interval_2$. When our model now calculates the contacts between people in the same age interval, it iterates over the interval's members and calculates a sample size based on the number of people in the interval and the adjusted contact probability. It then operates the same as regular sampling by keeping track of who has already been drawn and randomly selects people in the entire interval. An additional issue is solved by adding P_1 as an 'already drawn'

member, to prevent someone from having contact with themselves. We also discussed the problem of people having contact twice with each other in the same pool on the same day. When this happens, both contacts are handled and registered as valid independent events. If a transmission occurs, it will only happen the first time because the second time none of the members is susceptible, which thus does not interfere with the normal simulation flow.

4.5.2 Correctness

We also need to verify that this algorithm produces correct results. Like with the previous approaches, the built-in testing tool will be the first step to check full sampling. However, unlike iterative intervals and sampling with iteration, the full sampling approach fails some of the tests. Although it passed the majority of tests, this means that our algorithm produces invalid results and cannot be used as an optimisation for the all-to-all infector. To get a better understanding of why some tests failed, we examine the reversed contact rates in Figure 4.15. These also show that our new approach produces an excessive amount of contacts for the workplace and especially the K-12 school pools, while having valid rates for both community pools. Furthermore, it can also be seen that the difference between the original and full sampling approach is bigger the higher the original contact rate.

4.5.3 Performance

Although the full sampling algorithm is wrong, we are still interested in its results. Figure 4.16 shows that it is faster than the original and iterative intervals approach. Compared to sampling with iteration, it is remarkably slower on weekdays but faster on weekends. The speedup for the infector and total runtime compared to the original all-to-all, presented in Table 4.4, is respectively 2.48 and 2.37 which is less than sampling with iteration that has speedups of 2.63 and 2.50.

Since full sampling performs worse than expected, we further examine its results even though it is incorrect, because we can still gain valuable insights. In Figure 4.17 we clearly see why full sampling is both better and worse than sampling with iteration on different occasions. It needs significantly more time to compute the contacts in workplaces and even more time for K-12 schools. Nonetheless, the primary and secondary community pools slightly benefit from this faulty algorithm. If we look at Figure 4.18, we see that full sampling has the best runtimes for all larger pools, and that it has a linear time complexity for both communities as well as the workplace. So what we tried to achieve, namely speeding up same interval computations with sampling, has proven to be effective.

Because the total runtime for the workplace is the worst of all while having such good results for larger pools, we also examine the results for smaller pool sizes in Figure 4.19 for all types except the household. It can be seen that full sampling performs substantially worse on the K-12 school and college pools. For both community pools the algorithm performs slightly worse than sampling with iteration for smaller pools. This, together with the effects on both school pools is due to the overhead that sampling has. Full sampling becomes the fast for workplace pools with approximately 150 people. For the community pools it has the best runtimes for pools with at least 250 people and performs better than

Algorithm 9 Pseudo code of the all-to-all full sampling infector.

Input: $P_1 \dots P_N$, type ▷ All members of the pool and type of pool

```

1: if  $type = household$  then
2:   original all-to-all ▷ Algorithm 1
3: else
4:    $S[ ] \leftarrow \text{SORT\_INTERVALS}(P[ ], type)$  ▷ Algorithm 6, returns interval sizes
5:    $N_{\text{intervals}} \leftarrow \text{SIZEOF}(S[ ])$  ▷ Number of intervals

6:   for  $interval_1 \leftarrow 1$  to  $N_{\text{intervals}}$  do ▷ Iterative intervals
7:     for  $interval_2 \leftarrow interval_1$  to  $N_{\text{intervals}}$  do ▷ Iterate remaining intervals

8:       if  $interval_1 = interval_2$  then
9:          $C_{\text{prob}} \leftarrow \text{SPECIAL\_CPROB}(interval_1)$  ▷ Appendix A3
10:        for each member  $P_1$  in  $interval_1$  do
11:          if  $P_1$  not in pool then
12:            continue to next  $P_1$ 
13:             $sample \leftarrow \text{BINOMIAL}(S[interval_1], C_{\text{prob}})$  ▷ Binomial distribution
14:            if  $sample = 0$  then
15:              continue to next  $P_1$ 
16:            else if  $sample = S[interval_2]$  then
17:              iterate over  $interval_2$  and register contact with everyone
18:            else
19:               $drawn\_members \leftarrow []$ 
20:              add  $P_1$  to  $drawn\_members$ 
21:               $draws_{\text{all}} \leftarrow 1$  ▷ Number of draws, already 'drawn'  $P_1$ 
22:               $draws_{\text{good}} \leftarrow 0$  ▷ Number of good draws

23:              while  $draws_{\text{good}} < sample$  AND  $draws_{\text{all}} < S[interval_1]$  do
24:                 $P_2 \leftarrow$  random member of  $interval_1$ 
25:                if  $P_2$  in  $drawn\_members$  then
26:                  draw next  $P_2$ 
27:                  add  $P_2$  to  $drawn\_members[ ]$ 
28:                   $\text{++ } draws_{\text{all}}$ 
29:                  if  $P_2$  not in pool then
30:                    draw next  $P_2$ 
31:                     $\text{++ } draws_{\text{good}}$ 
32:                    register contact
33:                    if  $P_1$  or  $P_2$  susceptible and other infectious then
34:                       $T_{\text{prob}} \leftarrow$  transmission probability
35:                      if random number  $< T_{\text{prob}}$  then
36:                        infect the susceptible one
37:                    else
38:                      regular sampling between two different intervals ▷ Algorithm 8
```

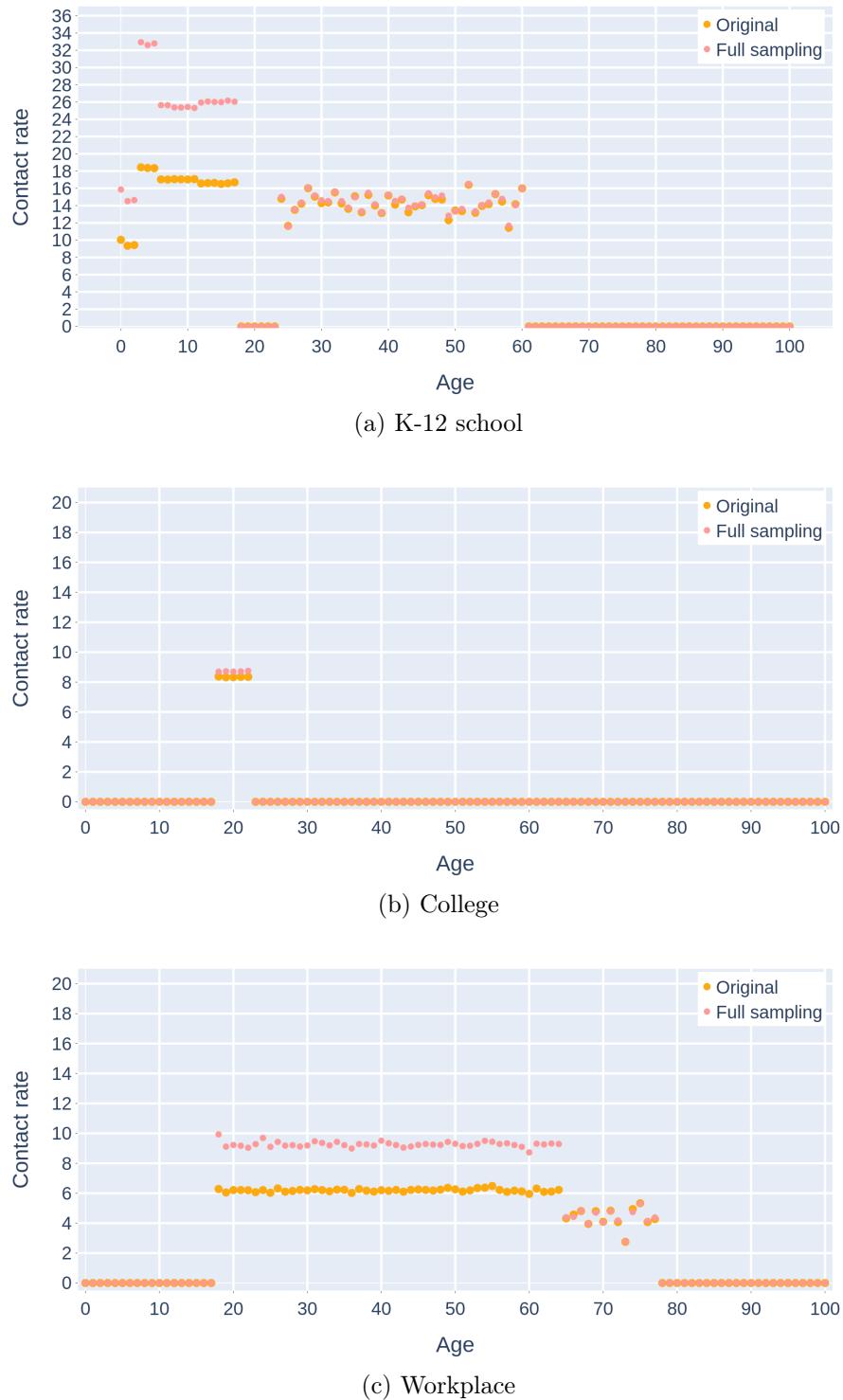


Figure 4.15: The reversed contact rates of full sampling compared to the original all-to-all reversed contact rates from Section 4.2.2.

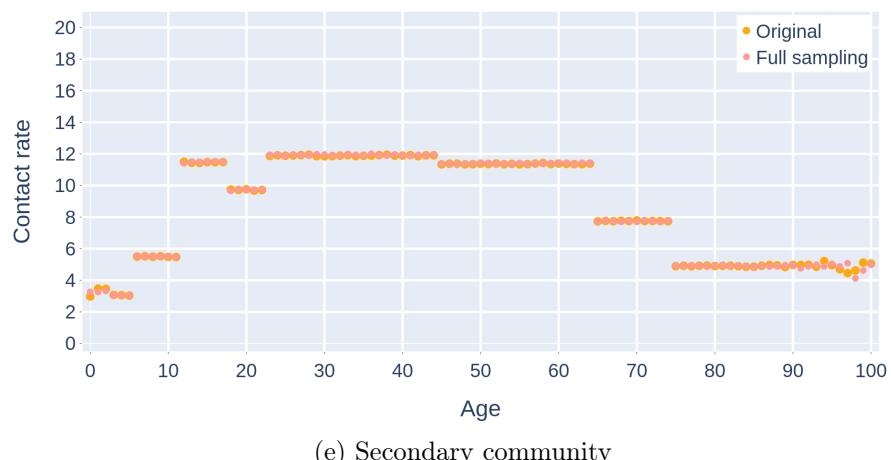
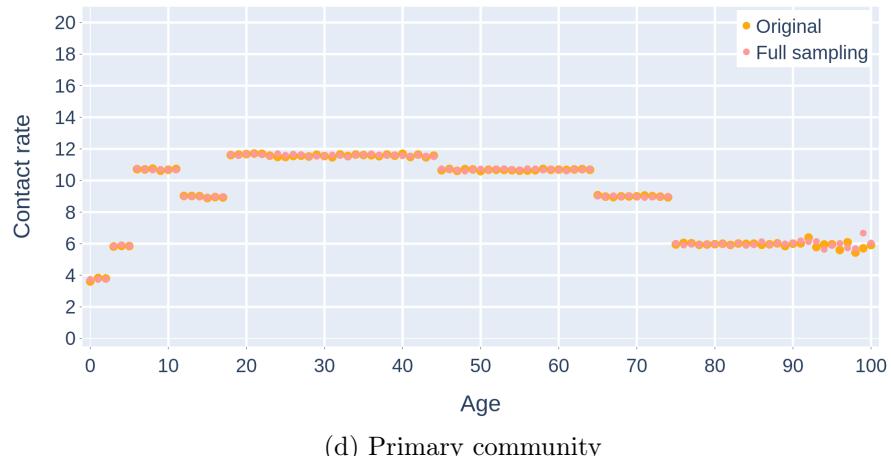


Figure 4.15: The reversed contact rates of full sampling compared to the original all-to-all reversed contact rates from Section 4.2.2.

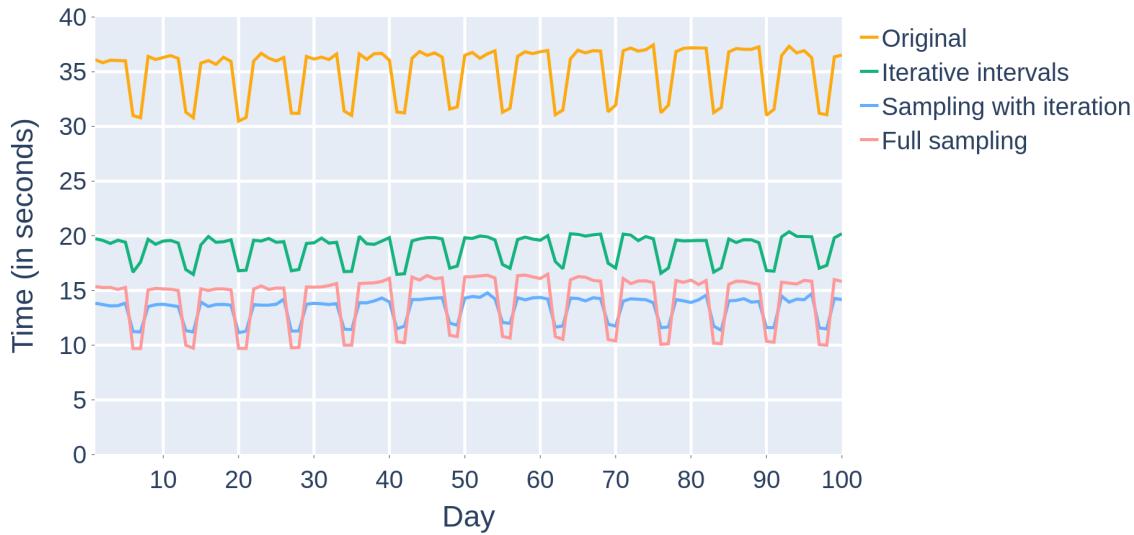


Figure 4.16: Comparison of the infector runtimes between full sampling and the other approaches.

	Infector	Total
original	35.08	36.31
full sampling	14.16	15.30
speedup	2.48	2.37

Table 4.4: Comparison of the average daily runtimes between the original and full sampling approach.

the original for pools that contain 150 members or more. The algorithm thus works as expected for larger pools, but fails to perform on smaller pools.

4.6 Adjusted full sampling

Examining the correctness of full sampling in Section 4.5.2 taught us that it produces too much contacts for workplace and K-12 school pools and little bit too much for college pools. The primary and secondary communities, on the other hand, generate a valid amount of contacts. Calculating the reversed contact rates does not consider the duplicate contacts that full sampling generates, which can thus result in such high rates. As we know, the K-12 school pools have the smallest pools excluding the household ones. Because these only contain up to 21 individuals, the probability of selecting duplicate contacts is therefore much higher than with larger pools, such as the communities. The pool size distribution of workplace pools, which was discussed in Section 3.3.2, showed us that 94% of these pools contain one to nine individuals. Following the explanation of K-12 schools,

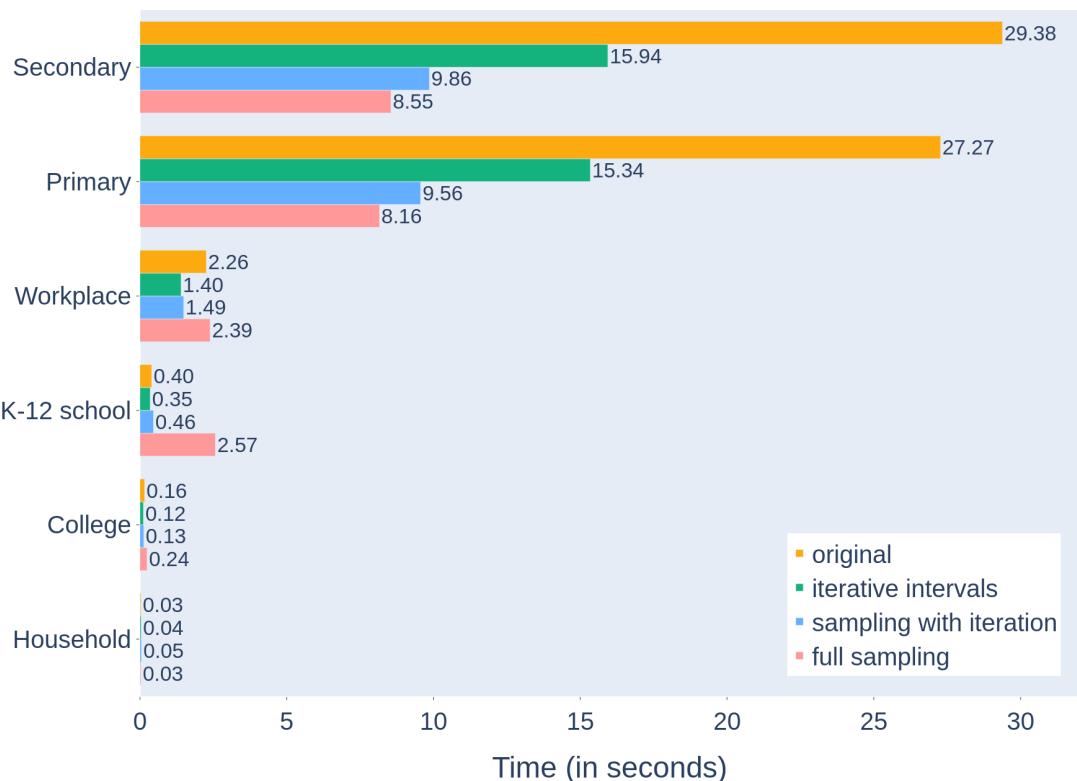


Figure 4.17: Comparison of the total infector runtimes (in seconds) per pool type on an active day. Simulation run on 11M without holidays using 1 thread.

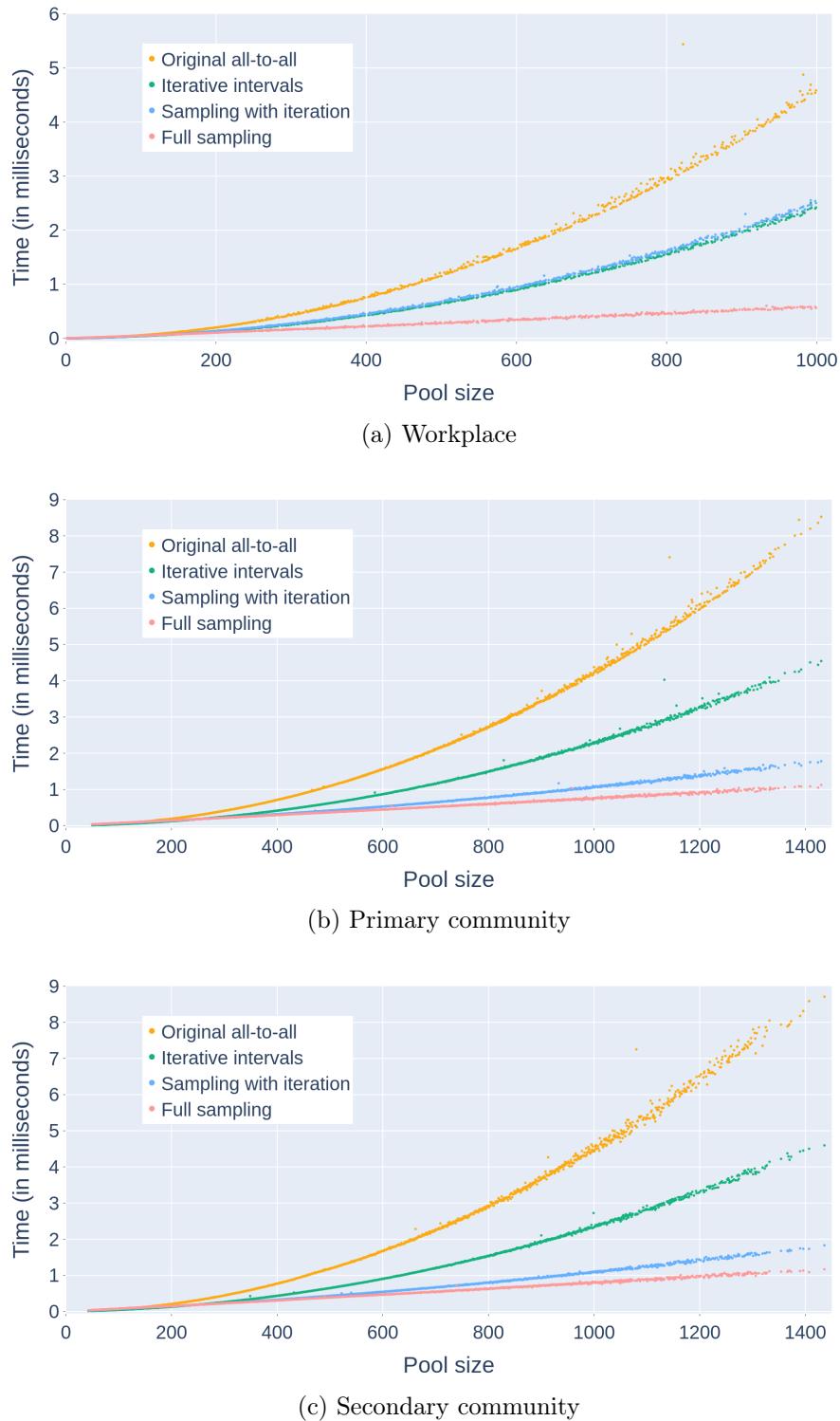


Figure 4.18: Average infector runtimes (in milliseconds) per pool size using different approaches. Simulation run on 11M without holidays using 1 thread.

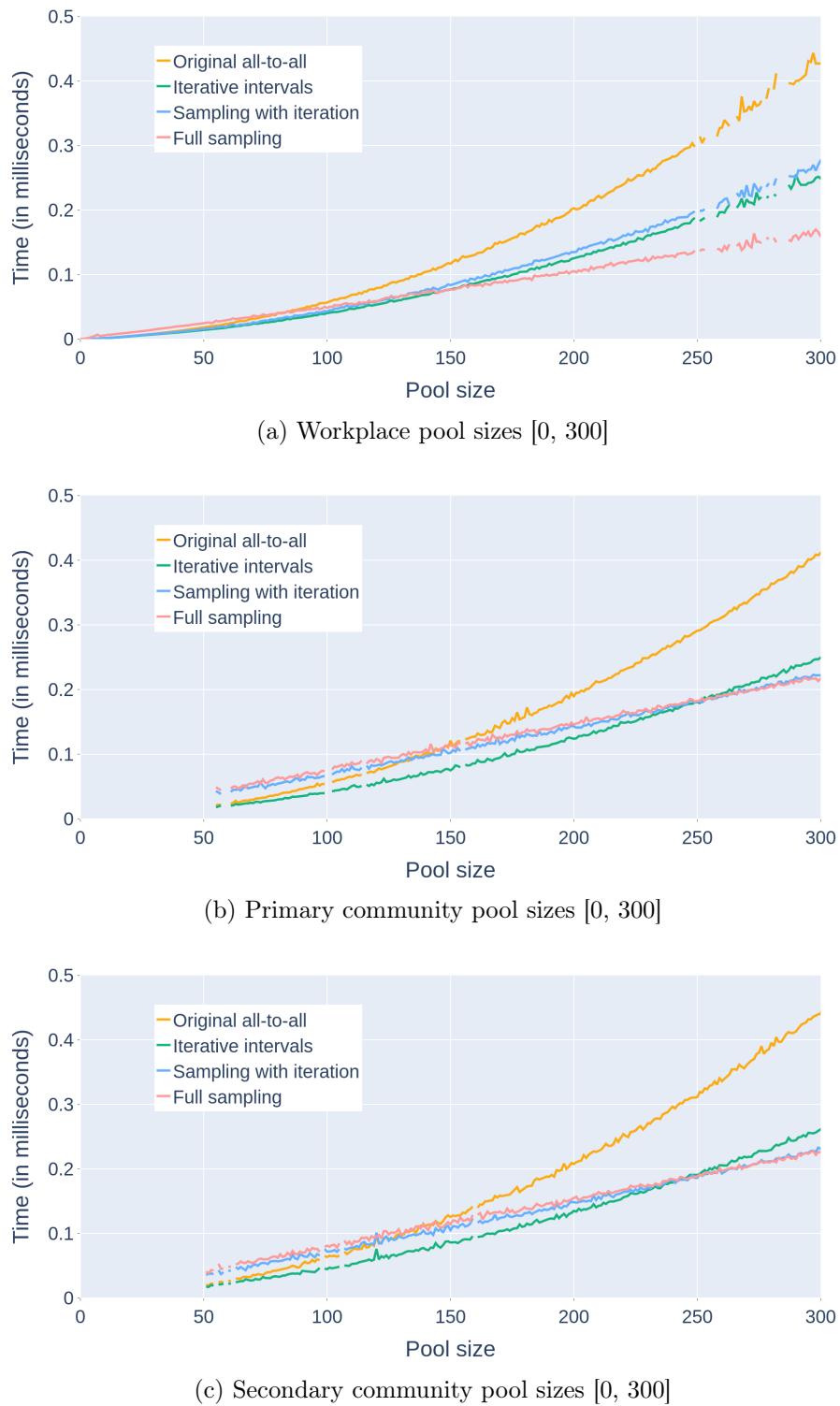
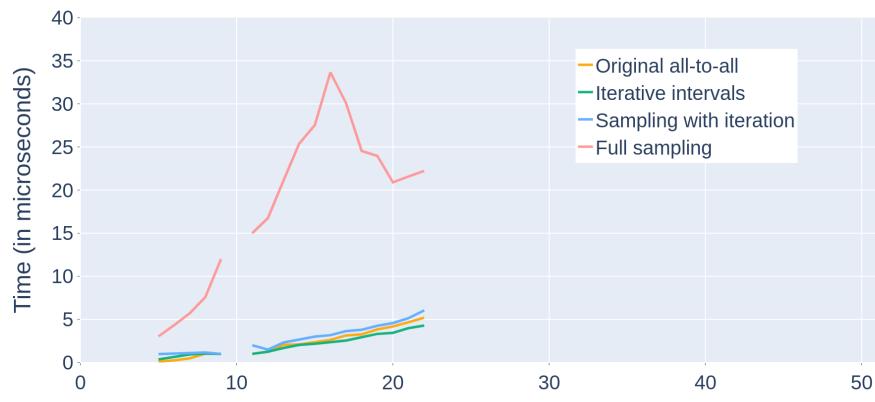
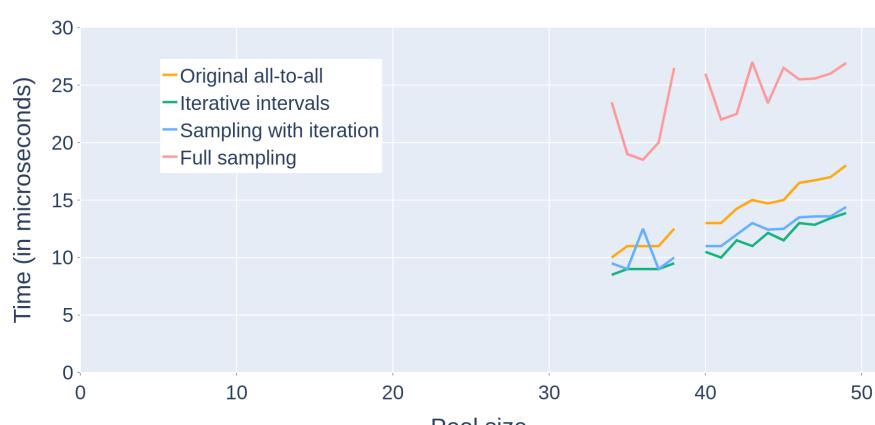


Figure 4.19: Average infector runtimes per pool size using different approaches. Simulation run on 11M without holidays using 1 thread.



(d) K-12 school



(e) College

Figure 4.19: Average infector runtimes per pool size using different approaches. Simulation run on 11M without holidays using 1 thread.

the reversed contact rates of the workplace are logically much higher than they are supposed to be, despite the fact that workplace pools have a maximum size of a thousand people. Most of the community pools have a larger size, which explains why their reversed contact rates were valid. Since duplicate contacts are handled as only one contact, it results in less diverse contacts, which in turn gives the disease less chance to be transmitted to new people. This clarifies why full sampling is incorrect and why the built-in tests failed.

Reducing the number of duplicate contacts could thus correct the full sampling algorithm. We just pointed out that those duplicates occur more the smaller the interval. Therefore, using full sampling only on larger pools that have a small chance of generating redundant contacts could solve the problem. Since Section 4.5.3 revealed that our approach becomes faster than the original for community pools with at least 150 people, we set this as threshold to decide if a pool uses the original algorithm or full sampling. The adjusted approach uses thus the same implementation as described in Algorithm 9, but changes the if-test that checks for a household pool to an if-test that checks if the pool has less than 150 members.

4.6.1 Correctness

To check if our adjusted full sampling approach works, we execute the built-in tests that it previously failed. This time, the algorithm passes every test which indicates that its results can be considered somewhat valid. The reversed contact rates in Figure 4.20 also show that our adjustment results in our algorithm to produce the right amount of contacts. Even though the algorithm now passes the tests, we know that its results are not 100% correct because the duplicate contacts can still happen and cause for less transmissions. Because we only apply our algorithm to pools with at least 150 members, duplicate contacts have a smaller probability to occur. Setting the threshold even higher would naturally increase the correctness of the results. With our current threshold of 150, we can say that the adjusted full sampling approach is a valid all-to-all infector algorithm as long as we accept the fact that its results are nearly 100% correct.

4.6.2 Performance

Now that we created a legitimate full sampling algorithm, we are interested in how it performs. Figure 4.21 presents promising results where this new approach achieves the best results out of all of the valid infector algorithms. Table 4.5 confirms that full sampling now reaches the fastest average infector and total runtimes and has thus the best speedup for these with respectively 2.83 and 2.67.

Because we set out pool size threshold at 150, the K-12 school and college pools only use the original all-to-all. Figure 4.22 shows that this is indeed the case and that their infector runtimes are slightly higher than the original, which is due to the extra if-test of course. The most noticeable difference can be seen for the workplace, which has now by far the best runtimes using the adjusted full sampling method. The community pools both have lost a small amount of time compared to the erroneous full sampling, but they now still execute faster than sampling with iteration by more than a second per day. In Section

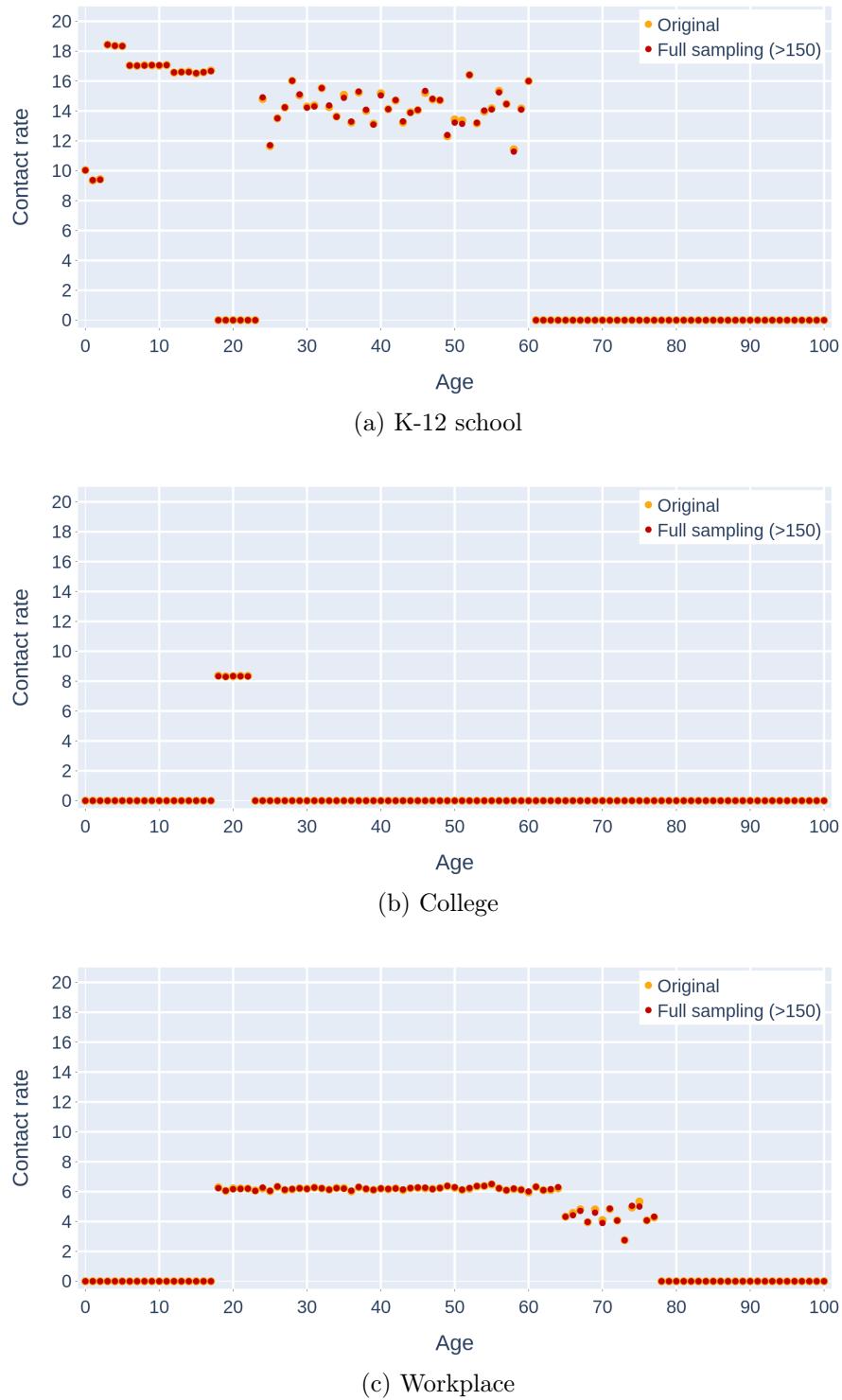


Figure 4.20: The reversed contact rates of full sampling on pools with 150+ people compared to the original all-to-all reversed contact rates from Section 4.2.2.

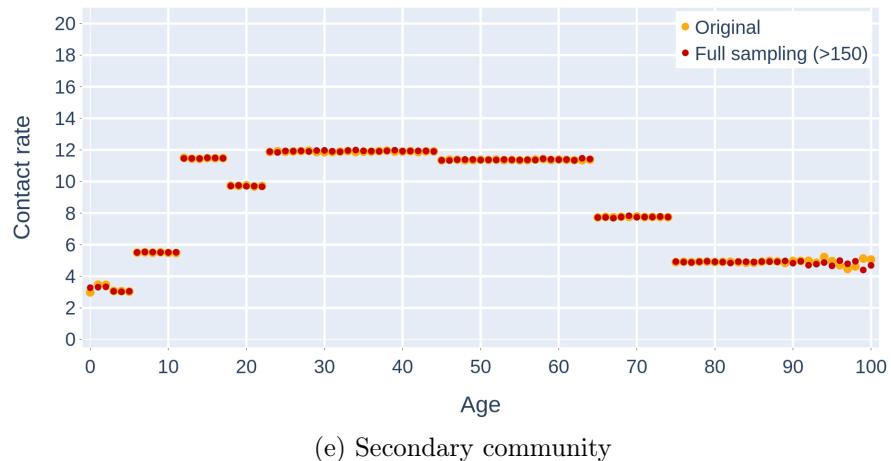
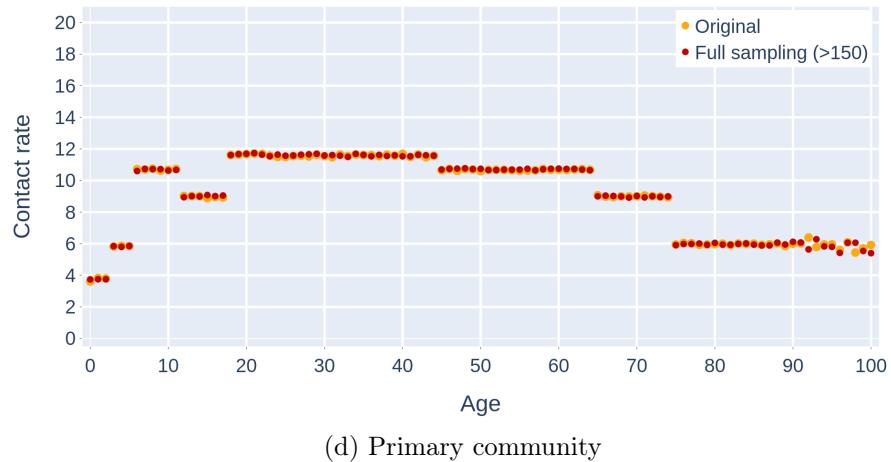


Figure 4.20: The reversed contact rates of full sampling on pools with 150+ people compared to the original all-to-all reversed contact rates from Section 4.2.2.

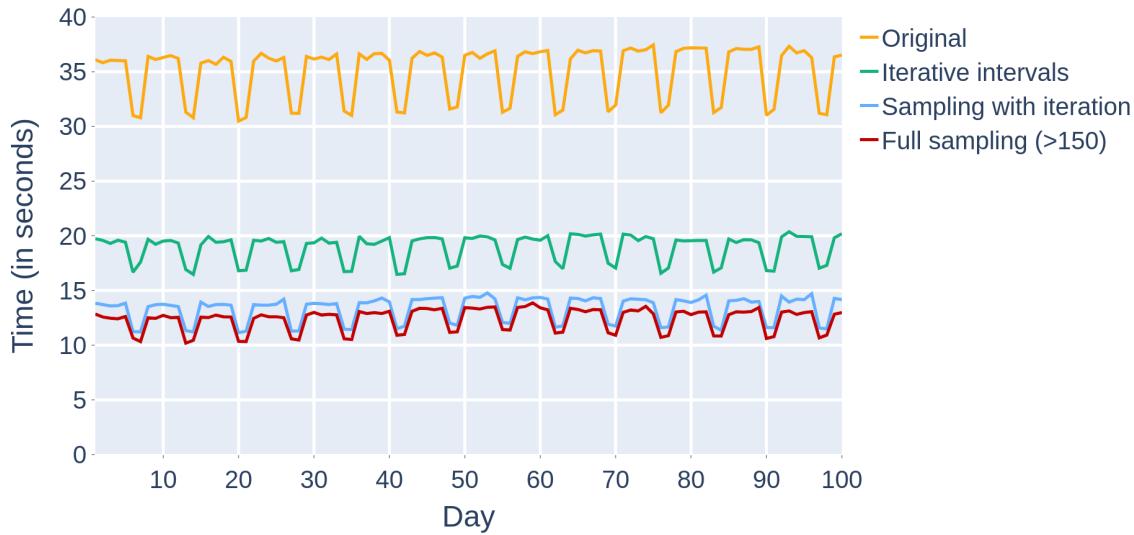


Figure 4.21: Comparison of the infector runtimes between the adjusted full sampling and the other valid approaches.

	Infector	Total
original	35.08	36.31
full sampling (>150)	12.37	13.58
speedup	2.84	2.67

Table 4.5: Comparison of the average daily runtimes between the original and the adjusted full sampling approach.

4.5.3 we already discussed how full sampling performs per pool type in function of the pool sizes. If we were to visualize these results again, we would see that every pool type follows the curve of the original all-to-all and that pools with at least 150 members follow the full sampling line.

4.7 Inf-to-sus

Since our new approaches are decent optimisations for the all-to-all infector, we are curious on how the inf-to-sus infector would be affected by these. As we know, this algorithm sorts the individuals based on their health status before comparing the infectious to the susceptibles in order to determine the transmissions. A pool that contains no infectious members on a certain day cannot produce any transmissions and is therefore skipped for that day. These runtimes are strongly correlated with the number of infectious people on a day. This gives the infector and total runtimes of the inf-to-sus its typical curve as displayed in Figure 3.19(b). If we want to implement our algorithms, there are some

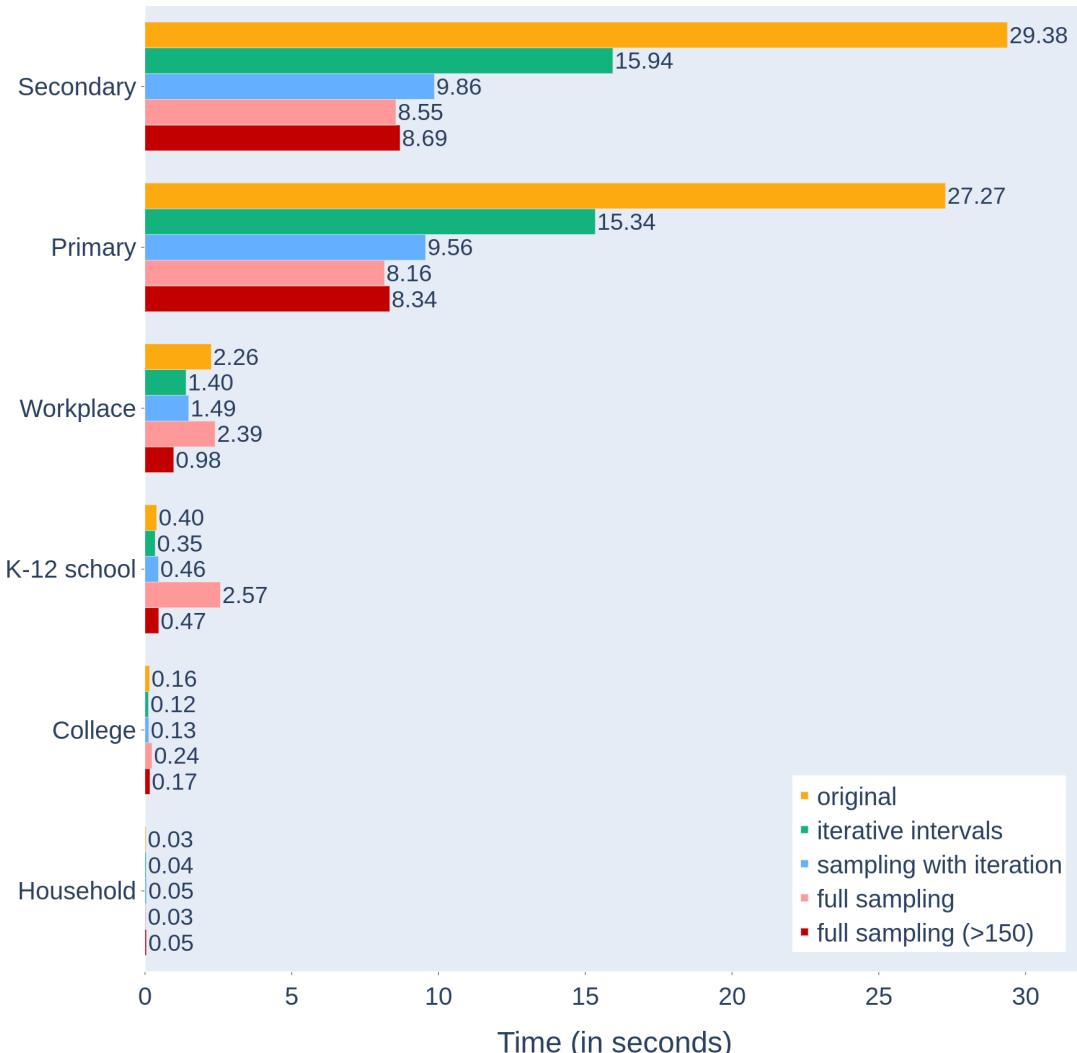


Figure 4.22: Comparison of the total infector runtimes (in seconds) per pool type on an active day. Simulation run on 11M without holidays using 1 thread.

options that need to be considered. Because only infectious get paired with susceptibles, we would need to sort our pool in age intervals twice: once for the susceptibles and once for the infectious individuals. This can turn out to be a rather costly implementation because of how the algorithm then would work. First, it needs to sort the pool members based on health status, after which it is known if there are infectious people among them or not. Recall from Section 3.2.5 that after this sorting, the susceptibles are put together as well as the immune members, but that the rest of the members are all put together. Sorting the susceptibles in age intervals thus would work the same as usual, but sorting the infectious individuals require a slightly different approach. Here we could choose to only sort the infectious members in intervals or sort the susceptible, recovered, and exposed members together. For our tests, we choose to only sort the infectious in intervals and place them together.

4.7.1 Double iterative intervals

The optimised algorithms now need to be converted so that they produce the same results as the original inf-to-sus infector. Iterative intervals is the most straightforward method to convert, because we can simply compare every infectious interval with every susceptible interval. Then for every infectious interval, the probability of contact can be calculated as we always did. Section 3.2.5 also taught us how the transmission probability is based on the health characteristics of the infected and the age of the receiving person. We determine the transmission probability separately for every infectious person based on the ages of every susceptible interval, so that we only need to calculate it once per infectious person for an entire susceptible interval, which reduces the number of computations. This produces correct results, because the age intervals never contain members that are both older and younger than 18 years old. The transmission and contact probability can then easily be combined to give the actual probability that inf-to-sus needs.

Sorting infectious members

We discussed why the inf-to-sus algorithm runtimes rely heavily on the number of infectious people. Thus, pools with only a few infectious members, have substantially faster runtimes than pools that contain numerous infectious people. Following this logic, if we would use our iterative intervals approach on a pool with only a couple of infectious members, the overhead of sorting the infectious people and handling these intervals could result in worse runtimes instead of better. Figure 3.20 showed us the number of infections per day in our simulations, which can differ a lot. Since the most amount of individuals are infected around day 50, we present the average number of infectious people per pool size on this simulation day in Figure 4.23. Here we can see that indeed only approximately 10% of the members are infectious that day. Therefore, next to the iterative intervals algorithm that sorts both infectious and susceptibles, we will also test this approach by only sorting the susceptibles and iterate over the infectious people as presented in Algorithm 2.

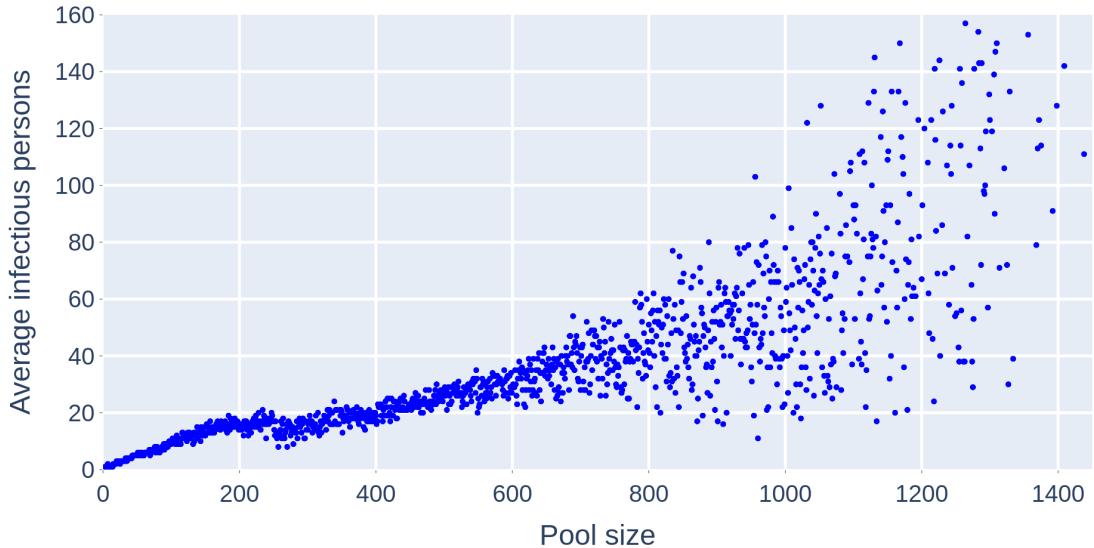


Figure 4.23: Average number of infectious people per pool size on day 50 of a Stride simulation on population 11M, using the configurations in Appendix A2.

Results

Both algorithms pass the tests of the built-in tool, so we can conclude that they produce correct results. Figure 4.24 shows the infector runtimes of the original inf-to-sus compared to our two approaches, which gives the impression that they are all very similar. Table 4.6 presents the average infector and total runtimes for these three approaches. Here we can see that the iterative intervals approach is only slightly faster than the original when it only sorts the susceptible members. This confirms our theory that the overhead plays a big role in the optimisations when there are only a few infectious people in a pool. The speedup of the iterative intervals approach that only sorts the susceptibles is 1.025 and thus not very significant. This also proves our statement that the inf-to-sus infector has very little room for improvements. Our optimised algorithm can possibly be improved if the health sorting and age interval sorting are combined in one general sorting function. We do not further examine this, because it would most likely have very little impact.

4.7.2 General sampling

Since iterative intervals produced better results, we are interested to see how the sampling approaches perform. Looking at how sampling could be used for the inf-to-sus infector, we notice that neither the sampling with iteration nor (adjusted) full sampling approach are applicable. They both have their own way of dealing with the calculations between members of the same interval. However, the members of infectious and susceptible intervals are mutually exclusive, so there can never occur calculations between members of the same interval. Because of this, we can simply use a general sampling algorithm that iterates over the infectious people and takes samples out of every susceptible interval. The binomial distribution can then be used with the combined probability as we described earlier, so

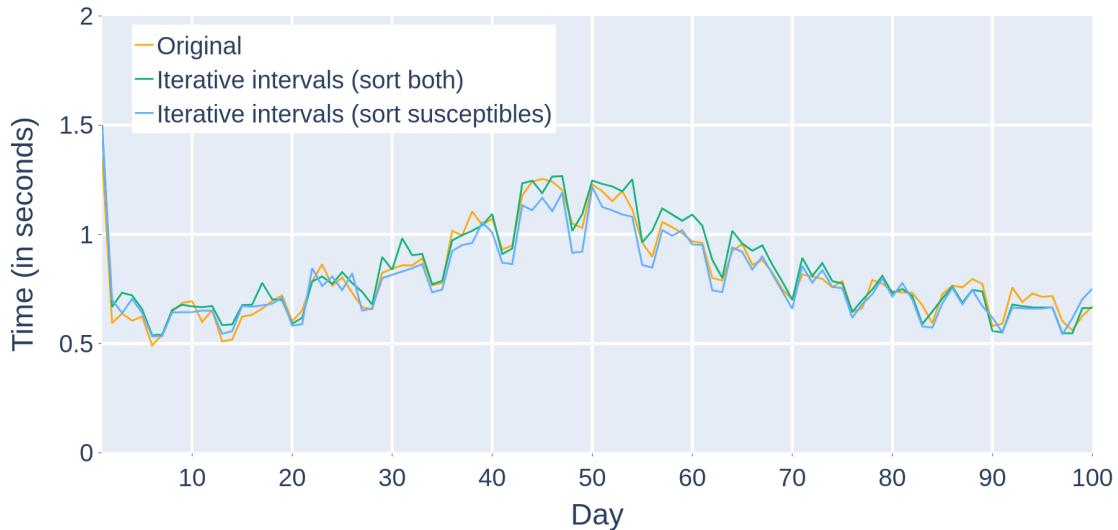


Figure 4.24: Comparison of the infector runtimes between the original inf-to-sus algorithm and both iterative intervals approaches.

	Infecto	Total
original	0.82	2.04
iterative intervals (sort both)	0.84	2.08
iterative intervals (sort susceptibles)	0.80	2.02

Table 4.6: Average infector and total runtimes per simulation day of the original inf-to-sus algorithm, compared to both iterative intervals approaches. Simulation run on 11M population.

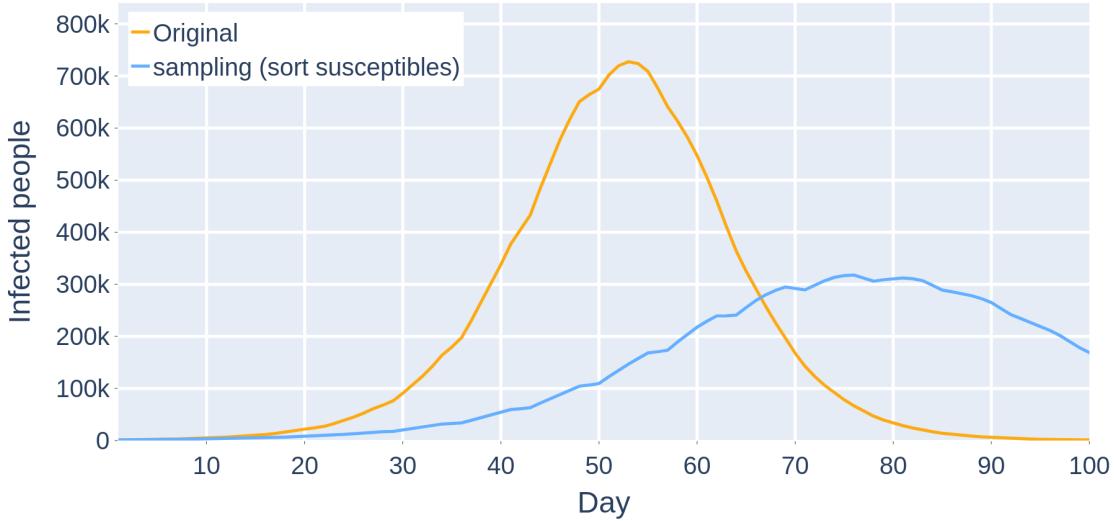


Figure 4.25: The number of infected people per day using different approaches. Simulations run on 11M population.

that the sample size represents the number of people that will become infected. Again, we implement this algorithm twice: one approach that sorts both infectious and susceptible members and one that only sorts the susceptible ones.

Results

Strangely enough, the approach that only sorts the susceptibles fails the built-in tests, while the approach that sorts both members passed them. Figure 4.25 shows the number of infected people per day using the failed approach compared to the original, which is far less than what it should be. We do not have a sound explanation for these results and examining the code did not indicate that there are programming errors. The infector and total average runtimes of these algorithms are displayed in Figure 4.26. The correct sampling approach has the same curve as the original, but is significantly higher which means it performs worse. This can be confirmed by the average runtimes presented in Table 4.7. As we know, the sampling method produces a lot of overhead and has therefore the worst times for small intervals. This explains its bad results, because the infectious intervals are relatively small most of the time. The simulation using the faulty algorithm has a different curve that is due to the correlation of the number of infected people per day. Using sampling on the inf-to-sus infector did thus not result in an optimisation.

4.7.3 General sampling contacts

The general sampling approach left us with some unanswered questions, therefore, we further investigate this by implementing another approach. We create a new algorithm that follows the same sampling methods as before, but this one samples contacts like the all-to-all infector instead of transmissions. This could provide insights why the previous

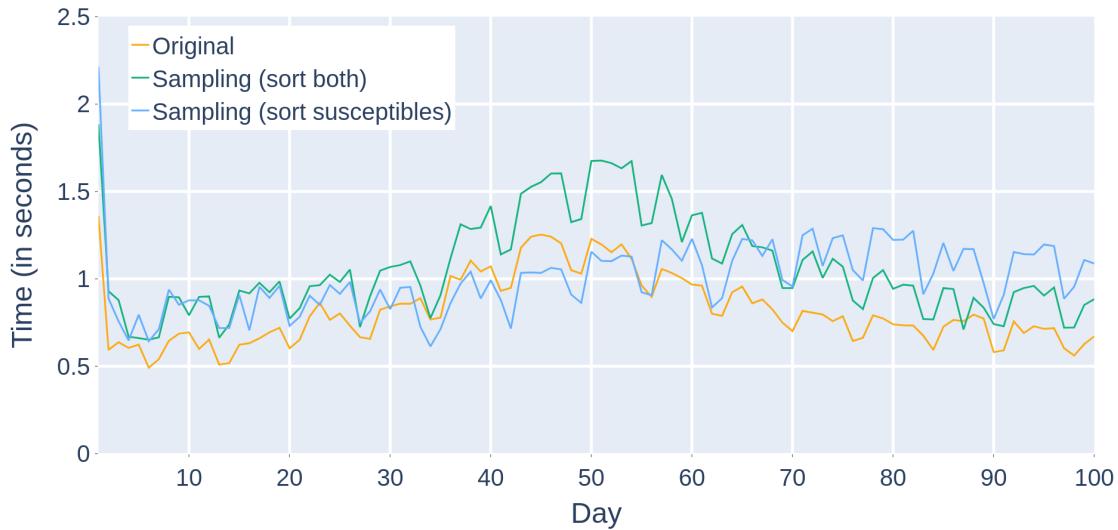


Figure 4.26: Comparison of the infector runtimes between the original inf-to-sus algorithm and both sampling approaches.

	Infector	Total
original	0.82	2.04
sampling (sort both)	1.06	2.39
sampling (sort susceptibles)	0.99	2.32

Table 4.7: Average infector and total runtimes per simulation day of the original inf-to-sus algorithm, compared to both sampling approaches. Simulation run on 11M population.

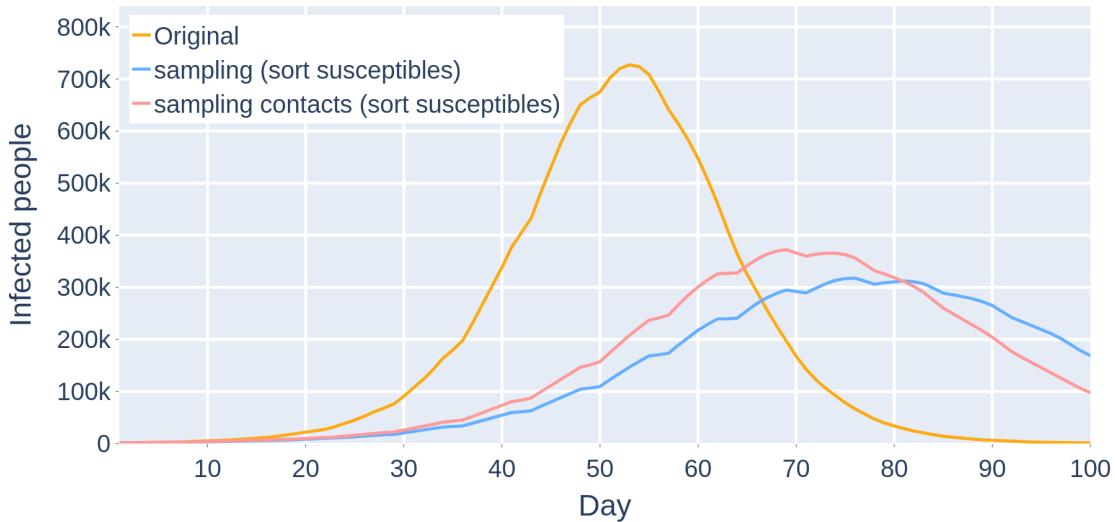


Figure 4.27: The number of infected people per day using different approaches. Simulations run on 11M population.

sampling algorithm failed or even point out code errors that were missed during examination. This new approach thus compares the infectious with the susceptibles and determines the contacts by a sample size that is based on the susceptible intervals and the contact probabilities between them and the infectious. We again implement this algorithm twice, once by only sorting the susceptibles and once by sorting both susceptible and infectious members.

Results

Just like the previous transmission sampling approaches, the one that only sorts the susceptibles fails the built-in tests while the other one passes them. However, Figure 4.27 presents the number of infected people per day of both faulty algorithms, which shows that sampling contacts leans more to the correct curve. There is also no apparent explanation for these phenomena, which still leaves us wondering about what causes these results. Section 4.4.4 discussed how one sample size for an entire interval produced erroneous data due to the less randomness. This might also be the case for these algorithms, although the approaches that sort both the infectious and susceptible members in intervals produce valid results despite that they should have the same level of randomness.

Figure 4.28 shows that sampling contacts when sorting both member types performs almost as good as the the original inf-to-sus, but becomes slower the more infectious people in a pool. Table 4.8 confirms that sampling contacts is indeed slower than the original and is, therefore, not an optimisation for the inf-to-sus infector. The faulty contacts sampling algorithm behaves the same as the transmissions sampling algorithm, which is caused by the erroneous number of infected people.

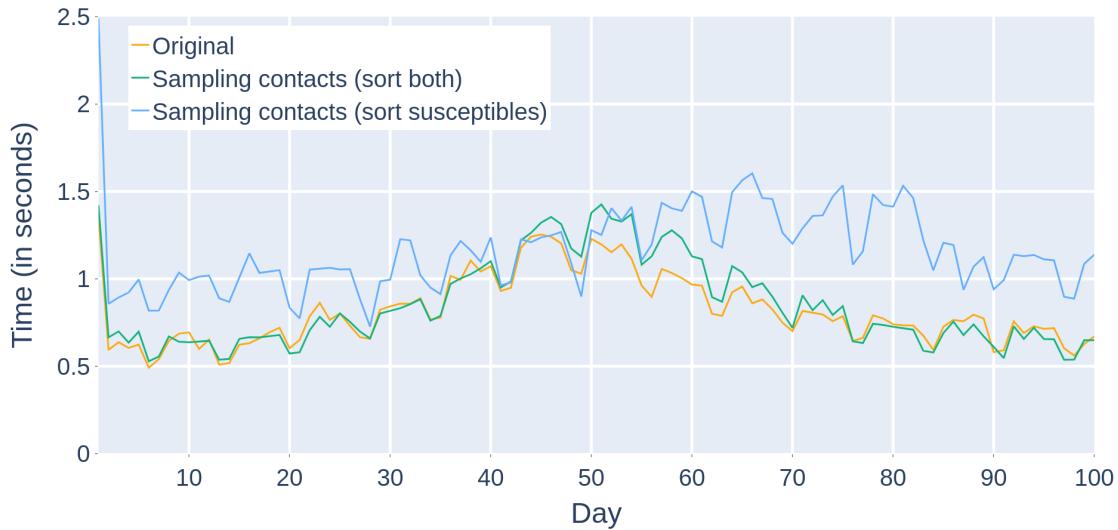


Figure 4.28: Comparison of the infector runtimes between the original inf-to-sus algorithm and both contact sampling approaches.

	Infector	Total
original	0.82	2.04
sampling contacts (sort both)	0.85	2.06
sampling contacts (sort susceptibles)	1.16	2.58

Table 4.8: Average infector and total runtimes per simulation day of the original inf-to-sus algorithm, compared to both contact sampling approaches. Simulation run on 11M population.

4.8 Summary

This chapter presented and examined various approaches to optimise the infector algorithms, with the emphasis on the all-to-all one. Figure 4.29 presents the infector runtimes of all of the valid all-to-all algorithms since the beginning of the thesis, including the original that passed the shared pointer of the population by value. The total runtime to simulate a day is almost identical, therefore, we do not also need to show these results. Figure 4.22 presented the impact of these approaches on the different pool types. Here it could be seen that the sampling approaches are by far the fastest to calculate the contacts for the workplace, and the primary and secondary community pools. K12-school and college pools benefit most from the iterative intervals algorithm. These results can all be explained based on the impact of the algorithms depending on the pool sizes, which was visualised in Figures 4.18 and 4.19. They prove that the sampling approaches have an approximately linear time complexity, which makes them faster the larger the pool compared with the original and iterative intervals algorithms that have a quadratic time complexity. They also showed that the opposite is true the smaller the pool, because the sampling approaches produce extra overhead.

Table 4.9 presents the average infector and total runtimes for a simulation day of all the valid all-to-all algorithms. From this we can conclude that we reduced the average time needed to calculate the contacts and transmissions by 5.20 times since the beginning of this thesis, and by 2.84 times compared to the improved original algorithm. The total time to simulate a day can be computed by our optimised algorithms with a speedup of 4.83 compared to the initial Stride model, and 2.67 compared to the improved original algorithm.

Since Stride is used primarily on the 11M population that represents Belgium, we are also interested in how our optimisations affect the entire simulation. The simulation times using all the valid all-to-all approaches are displayed in Table 4.10. Here, day 1-100 represents the total time it takes to execute all the calculations for all simulation days combined. The total simulation time shows how long a simulation actually ran from start to finish including the initialisation and other small operations. It can be concluded from these results that our optimisations can achieve a speedup of 4.47 compared to the Stride simulations at the start of this thesis, and 2.52 compared to the original pass-by-reference model.

Regarding the inf-to-sus simulations, the only actual improvement is the pass-by-reference solution that is discussed in Section 3.5. Using the iterative intervals approach we also achieved a very small speedup of 1.025 for the infector. Since this relatively small, further examination is required to confirm that this is a consistent optimisation. Because it has little impact on the simulation runtimes, we leave the inf-to-sus as is.

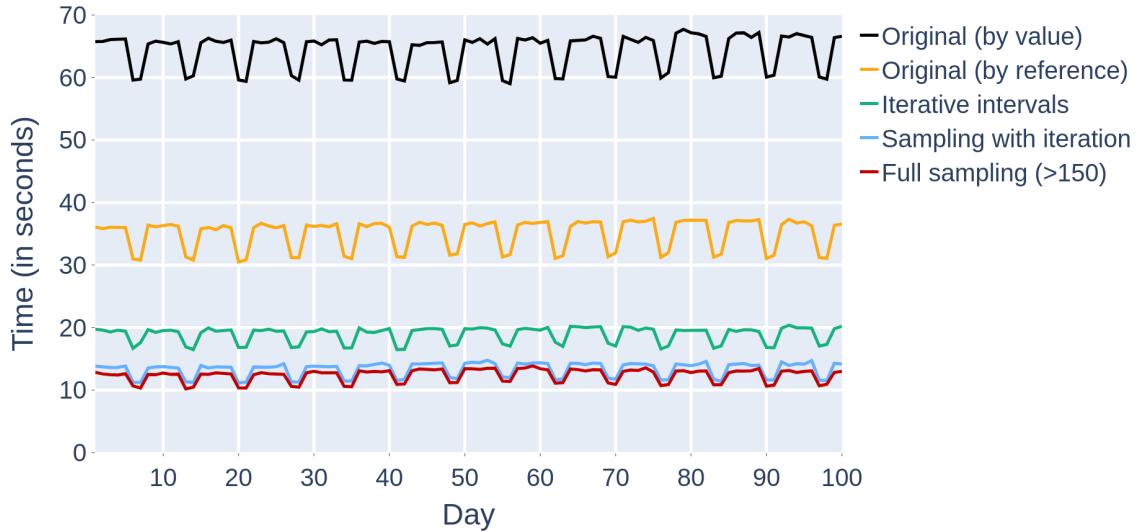


Figure 4.29: The infector runtimes of all the valid all-to-all approaches. Simulation run with the configurations in Appendix A2, on 11M population using one thread.

Approaches	Time	Infector		Total	
		value	reference	Time	value
original (by value)	64.33			65.56	
original (by reference)	35.08	1.83		36.31	1.81
iterative intervals	18.92	3.40	1.85	20.16	3.25
sampling with iteration	13.35	4.82	2.63	14.55	4.51
full sampling (>150)	12.37	5.20	2.84	13.58	4.83

Table 4.9: The average runtime (in seconds) of the infector per simulation day and total simulation day average of the valid all-to-all approaches. Simulations are run with the configurations in Appendix A2, on 11M population using one thread. Speedups are calculated in comparison with the initial Stride model, used at the beginning of the thesis (original by value), and its optimisation from Section 3.5 (original by reference).

Approaches	Day 1-100			Total simulation		
	Time	Speedup vs.		Time	Speedup vs.	
		value	reference		value	reference
original (by value)	1:50:51			1:51:55		
original (by reference)	1:01:57	1.79		1:03:01	1.78	
iterative intervals	0:34:57	3.17	1.77	0:36:03	3.10	1.75
sampling with iteration	0:25:35	4.33	2.42	0:26:39	4.20	2.36
full sampling (>150)	0:23:58	4.63	2.58	0:25:03	4.47	2.52

Table 4.10: Runtimes (in h:mm:ss) of all days combined and the total simulation that includes all overhead such as the initialisation, of the valid all-to-all approaches. Simulations are run with the configurations in Appendix A2, on 11M population using one thread. Speedups are calculated in comparison with the initial Stride model, used at the beginning of the thesis (original by value), and its optimisation from Section 3.5 (original by reference).

Chapter 5

Domain specific language

Chapter 6

Conclusion

Chapter 7

Future work

- Updating of individuals optimisations
- Sampling with iteration: divide large intervals in sub intervals
- Fine-tune full sampling
- 2D contact matrices with intervals

Bibliography

- [1] Hunter Louise. Challenging the reported disadvantages of e-questionnaires and addressing methodological issues of online data collection. *Nurse Researcher (through 2013)*, 20(1):11–20, 09 2012. Copyright - Copyright RCN Publishing Company Sep 2012; Last updated - 2016-04-09.
- [2] Hongtao Li, Feng Guo, Wenyin Zhang, Jie Wang, and Jinsheng Xing. (a,k)-anonymous scheme for privacy-preserving data collection in iot-based healthcare services systems. *Journal of Medical Systems*, 42, 02 2018.
- [3] Konstantina Kourou, Themis P. Exarchos, Konstantinos P. Exarchos, Michalis V. Karamouzis, and Dimitrios I. Fotiadis. Machine learning applications in cancer prognosis and prediction. *Computational and Structural Biotechnology Journal*, 13:8–17, 2015.
- [4] Michael T. Meehan, Diana P. Rojas, Adeshina I. Adekunle, Oyelola A. Adegbeye, Jamie M. Caldwell, Evelyn Turek, Bridget M. Williams, Ben J. Marais, James M. Trauer, and Emma S. McBryde. Modelling insights into the covid-19 pandemic. *Paediatric Respiratory Reviews*, 35:64–69, 2020.
- [5] Elise Kuilen, Sean Stijven, Jan Broeckhove, and Lander Willem. Social contact patterns in an individual-based simulator for the transmission of infectious diseases (stride). *Procedia Computer Science*, 108:2438–2442, 2017. International Conference on Computational Science, ICCS 2017, 12-14 June 2017, Zurich, Switzerland.
- [6] J.M. Seventer and N.S. Hochberg. Principles of infectious diseases: Transmission, diagnosis, prevention, and control. *International Encyclopedia of Public Health*, pages 22–39, 10 2016.
- [7] Kenrad E Nelson et al. Epidemiology of infectious disease: general principles. *Infectious Disease Epidemiology Theory and Practice*. Gaithersburg, MD: Aspen Publishers, pages 17–48, 2007.
- [8] Ken T D Eames and Matt J Keeling. Contact tracing and disease control. *Proceedings. Biological sciences*, 270(1533):2565—2571, December 2003.
- [9] Stuart E. Turvey and David H. Broide. Innate immunity. *The Journal of allergy and clinical immunology*, 125(2 Suppl 2):S24–S32, Feb 2010. 19932920[pmid].

- [10] Acquired immunity. <https://clinicalinfo.hiv.gov/en/glossary/acquired-immunity>. Accessed: 2021-03-07.
- [11] Immunity types. <https://www.cdc.gov/vaccines/vac-gen/immunity-types.htm>. Accessed: 2021-03-07.
- [12] Acute infections. https://mpkb.org/home/pathogenesis/microbiota/acute_infections. Accessed: 2021-03-05.
- [13] Howard (Howie) (Georgia Institute of Technology. Mathematics Department) Weiss. The sir model and the foundations of public health. *Materials matemàtics*, pages 1–17, 2013.
- [14] Matt J. Keeling and Pejman Rohani. *Modeling Infectious Diseases in Humans and Animals*. Princeton University Press, 2008.
- [15] Yu. A. Kuznetsov and C. Piccardi. Bifurcation analysis of periodic seir and sir epidemic models. *Journal of Mathematical Biology*, 32(2):109–121, Jan 1994.
- [16] Rui-Xing Ming, Jiming Liu, William K. W. Cheung, and Xiang Wan. Stochastic modelling of infectious diseases for heterogeneous populations. *Infectious Diseases of Poverty*, 5(1):107, Dec 2016.
- [17] Joël Mossong, Niel Hens, Mark Jit, Philippe Beutels, Kari Auranen, Rafael Mikolajczyk, Marco Massari, Stefania Salmaso, Gianpaolo Scalia Tomba, Jacco Wallinga, Janneke Heijne, Małgorzata Sadkowska-Todys, Magdalena Rosinska, and W. John Edmunds. Social contacts and mixing patterns relevant to the spread of infectious diseases. *PLoS Medicine*, 5(3):e74, March 2008.
- [18] Lander Willem, Frederik Verelst, Joke Bilcke, Niel Hens, and Philippe Beutels. Lessons from a decade of individual-based models for infectious disease transmission: a systematic review (2006-2015). *BMC Infectious Diseases*, 17(1), September 2017.
- [19] Willem Lander. *Agent-based models for infectious disease transmission : exploration, estimation & computational efficiency*. PhD thesis, University of Antwerp, 2015.
- [20] Kilian J. Murphy, Simone Ciuti, and Adam Kane. An introduction to agent-based models as an accessible surrogate to field-based research and teaching. *Ecology and Evolution*, 10(22):12482–12498, 2020.
- [21] Zhen Z. Shi, Chih-Hang Wu, and David Ben-Arieh. Agent-based model: A surging tool to simulate infectious diseases in the immune system. *Open Journal of Modelling and Simulation*, 02(01):12–22, 2014.
- [22] Eric Bonabeau. Agent-based modeling: Methods and techniques for simulating human systems. *Proceedings of the National Academy of Sciences*, 99(suppl 3):7280–7287, 2002.
- [23] Joke Bilcke, Philippe Beutels, Marc Brisson, and Mark Jit. Accounting for methodological, structural, and parameter uncertainty in decision-analytic models. *Medical Decision Making*, 31(4):675–692, June 2011.

- [24] Nele Goeyvaerts, Lander Willem, Kim Van Kerckhove, Yannick Vandendijck, Germaine Hanquet, Philippe Beutels, and Niel Hens. Estimating dynamic transmission model parameters for seasonal influenza by fitting to age and season-specific influenza-like illness incidence. *Epidemics*, 13:1–9, December 2015.
- [25] Lander Willem, Sean Stijven, Ekaterina Vladislavleva, Jan Broeckhove, Philippe Beutels, and Niel Hens. Active learning to understand infectious disease models and improve policy making. *PLoS Computational Biology*, 10(4):e1003563, April 2014.
- [26] Dennis L. Chao, M. Elizabeth Halloran, Valerie J. Obenchain, and Ira M. Longini, Jr. Flute, a publicly available stochastic influenza epidemic simulation model. *PLOS Computational Biology*, 6(1):1–8, 01 2010.
- [27] John J. Grefenstette, Shawn T. Brown, Roni Rosenfeld, Jay Depasse, Nathan T.B. Stone, Phillip C. Cooley, William D. Wheaton, Alona Fyshe, David D. Galloway, Anuroop Sriram, Hasan Guchu, Thomas Abraham, and Donald S. Burke. Fred (a framework for reconstructing epidemic dynamics): An open-source software system for modeling infectious diseases and control strategies using census-based populations. *BMC Public Health*, 13(1), 2013. Copyright: Copyright 2013 Elsevier B.V., All rights reserved.
- [28] Lander Willem, Thang Van Hoang, Sebastian Funk, Pietro Coletti, Philippe Beutels, and Niel Hens. Socrates: an online tool leveraging a social contact data sharing initiative to assess mitigation strategies for covid-19. *BMC Research Notes*, 13(1):293, Jun 2020.
- [29] Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, 1979.
- [30] C++ shared pointer documentation. http://www.cplusplus.com/reference/memory/shared_ptr/. Accessed: 2021-04-24.
- [31] How to: Create and use shared_ptr instances. <https://docs.microsoft.com/en-us/cpp/cpp/how-to-create-and-use-shared-ptr-instances>. Accessed: 2021-04-24.
- [32] Athanasios Papoulis and S Unnikrishna Pillai. *Probability, random variables, and stochastic processes*. Tata McGraw-Hill Education, 2002.
- [33] David Wypij. Binomial distribution. In *Wiley StatsRef: Statistics Reference Online*. American Cancer Society, 2014.
- [34] S.H. Unger. Hazards, critical races, and metastability. *IEEE Transactions on Computers*, 44(6):754–768, 1995.
- [35] Raj Parihar. Branch prediction techniques and optimizations, 2009.
- [36] Jun Sawada and Warren A. Hunt. Processor verification with precise exceptions and speculative execution. In Alan J. Hu and Moshe Y. Vardi, editors, *Computer Aided Verification*, pages 135–146, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg.

BIBLIOGRAPHY

- [37] Kai Yao and Jinwu Gao. Law of large numbers for uncertain random variables. *IEEE Transactions on Fuzzy Systems*, 24(3):615–621, 2016.
- [38] C++ vector documentation. <http://www.cplusplus.com/reference/vector/vector/>. Accessed: 2021-05-18.
- [39] C++ unordered_set documentation. http://www.cplusplus.com/reference/unordered_set/unordered_set/. Accessed: 2021-05-18.

Appendix A1

Nederlandse samenvatting

Appendix A2

Configurations

Parameter	Value	Additional info
age_contact_matrix_file	contact_matrix_flanders _conditional_teachers.xml	
event_log_level	'All' or 'Transmission'	respectively for all-to-all and inf-to-sus
event_output_file	false	
disease_config_file	disease_covid19_age.xml	
global_information_policy	NoGlobalInformation	
holidays_file	holidays_none.csv	adjusted so that contact tracing occurs every day
immunity_profile	None	
immunity_rate	0.8	
num_days	100	
num_participants_survey	10	
num_threads	1	
output_cases	true	
output_persons	false	
output_summary	false	
population_file	pop_belgium11M_c500 _teachers_censushh.csv	
population_type	default	
rng_seed	2020	
r0	11	
seeding_age_max	99	
seeding_age_min	1	
num_infected_seeds	1200	
start_date	2020-03-02	
stride_log_level	info	
track_index_case	false	
vaccine_link_probability	0	
vaccine_profile	Random	
vaccine_rate	0.8	
detection_probability	0.5	

Table A2.1: Configurations to run general Stride simulations throughout this thesis.

Appendix A3

Sampling same interval

The following pages present the reasoning and prove for sampling in the same interval, for which all credit is due to prof. dr. Stijn Vansumeren.

Sampling a contact relation within the same age interval

Stijn Vansumeren

December 4, 2020

Assume that we are given a set D of n persons, all with the same age. Further assume that we know the contact rate r , that is, we know that on average, every person in D will meet with r other people in D .

Our goal is to use a sampling-based algorithm to derive a *contact relation* over D . Formally a *contact relation over D* is a binary relation $R \subseteq D \times D$ such that:

- if $(a, b) \in R$ then $a \neq b$ (people don't meet themselves); and
- if $(a, b) \in R$ then also $(b, a) \in R$ (having a contact is symmetric).

The following algorithm allows us to derive a (random) contact relation R over D . For now, assume that $p \in [0, 1]$ is given. We will determine its correct value below.

Sampling algorithm.

1. Initialize R to be empty.
2. For each x in D :
 - (a) Draw $K_x \sim B(n - 1, p)$
 - (b) Let Y_x be a sample of K_x elements from $D \setminus \{x\}$.
 - (c) For every $y \in Y_x$, add (x, y) and (y, x) to R

Here, $B(n - 1, p)$ is the Binomial distribution with parameters $n - 1$ and p .

It is straightforward to verify that the resulting relation R will be valid a contact relation.¹

¹For the purpose of simulating epidemics we of course don't want to actually build R , but instead immediately process the pairs (x,y) and (y,x) of R as soon as they are generated. By explicitly building R in these notes, we can reason over the correctness of the sampling algorithm.

Let ρ_a denote the expected number of contacts that a person $a \in D$ has in the generated contact relation R . Our goal now is to derive a value for p such that ρ_a equals the contact rate r . Hereto, we reason as follows.

The probability space. We will need to reason over the probability space (Ω, P) , where the set Ω of possible outcomes is the set of all possible runs of the algorithm. Concretely, each outcome $\omega \in \Omega$ is hence a tuple of the form

$$\omega = (R^\omega, (K_x^\omega)_{x \in D}, (Y_x^\omega)_{x \in D}),$$

that records the contact relation R^ω constructed by the run ω , as well as the family of natural numbers $(K_x^\omega)_{x \in D}$ chosen during the run and the family of samples $(Y_x^\omega)_{x \in D}$.

In what follows, if $x \in D$ then we write K_x for the discrete random variable that, given outcome $\omega \in \Omega$ returns the actual value K_x^ω chosen in line (2.a). The random variables R and Y_x are defined similarly. Furthermore, if x and b are elements of D then we overload notation and denote by the expression $x \in Y_b$ the event consisting of all outcomes ω for which $x \in Y_b(\omega)$. That is,

$$(x \in Y_b) = \{\omega \in \Omega \mid x \in Y_b(\omega)\}.$$

For a given event $\alpha \subseteq \Omega$, $\mathbf{1}_{\{\alpha\}}$ denotes the indicator random variable, i.e., the binary random variable such that

$$\mathbf{1}_{\{\alpha\}}(\omega) = \begin{cases} 1 & \text{if } \omega \in \alpha \\ 0 & \text{otherwise.} \end{cases}$$

Counting the number of contacts in one particular outcome. Consider an arbitrary outcome $\omega = (R^\omega, (K_x^\omega)_{x \in D}, (Y_x^\omega)_{x \in D})$. What is the number of contacts that person a has in R^ω ? Because R^ω is symmetric by definition, the number of contacts of a person a is simply the number of times that a occurs in the first column in R^ω , i.e., it is the number of tuples in $(a, b) \in R^\omega$ with $b \in D \setminus \{a\}$. Note that such a tuple (a, b) may have been added to R^ω because $b \in Y_a^\omega$, or because $a \in Y_b^\omega$, or both. Therefore, the number of times that a occurs in the first column in R^ω equals

$$\begin{aligned} & \sum_{b \in Y_a^\omega} 1 + \sum_{\substack{b \in D \setminus \{a\} \\ a \in Y_b^\omega}} 1 - \sum_{\substack{b \in D \setminus \{a\} \\ a \in Y_b^\omega, b \in Y_a^\omega}} 1 = |Y_a^\omega| + \sum_{\substack{b \in D \setminus \{a\} \\ a \in Y_b^\omega}} 1 - \sum_{\substack{b \in D \setminus \{a\} \\ a \in Y_b^\omega, b \in Y_a^\omega}} 1 \\ &= K_a(\omega) + \sum_{b \in D \setminus \{a\}} \mathbf{1}_{\{a \in Y_b\}}(\omega) - \sum_{b \in D \setminus \{a\}} \mathbf{1}_{\{a \in Y_b, b \in Y_a\}}(\omega). \end{aligned}$$

Determining the expected number of occurrences. The expected number of occurrences of a is then

$$\rho_a = E[K_a + \sum_{b \in D \setminus \{a\}} \mathbf{1}_{\{a \in Y_b\}} - \sum_{b \in D \setminus \{a\}} \mathbf{1}_{\{a \in Y_b, b \in Y_a\}}] \quad (1)$$

$$= E[K_a] + \sum_{b \in D \setminus \{a\}} E[\mathbf{1}_{\{a \in Y_b\}}] - \sum_{b \in D \setminus \{a\}} E[\mathbf{1}_{\{a \in Y_b, b \in Y_a\}}] \quad (2)$$

$$= E[K_a] + \sum_{b \in D \setminus \{a\}} P(a \in Y_b) - \sum_{b \in D \setminus \{a\}} P(a \in Y_b, b \in Y_a) \quad (3)$$

$$= (E[K_a] + \sum_{b \in D \setminus \{a\}} P(a \in Y_b) - \sum_{b \in D \setminus \{a\}} P(a \in Y_b) P(b \in Y_a)) \quad (4)$$

$$= (n-1)p + \sum_{b \in D \setminus \{a\}} p - \sum_{b \in D \setminus \{a\}} pp \quad (5)$$

$$= (n-1)p + (n-1)p - (n-1)p^2 \quad (6)$$

$$= 2(n-1)p - (n-1)p^2. \quad (7)$$

Here, (2) is due to linearity of expectation; (3) is because $E[\mathbf{1}_{\{\alpha\}}] = P(\alpha)$ for every indicator variable $\mathbf{1}_{\{\alpha\}}$ and every event α ; (4) is because Y_a and Y_b are independent random variables; and (5) is because:

- $K_a \sim B(n-1, p)$ by definition of the algorithm; therefore K_a follows a Binomial distribution and as such it is known that $E[K_a] = (n-1)p$;
- $P(b \in Y_a) = P(a \in Y_b) = p$, as we will formally demonstrate further below.

Determining p . We want ρ_a to equal the contact rate r .

$$\begin{aligned} \rho_a &= r \\ \iff 2(n-1)p - (n-1)p^2 &= r \\ \iff p^2 - 2p + \frac{r}{n-1} &= 0 \end{aligned}$$

As such, we can determine p by finding the roots of a quadratic polynomial. There are at most two such roots, given by

$$p_1 = \frac{2 + \sqrt{4 - 4\frac{r}{n-1}}}{2} \quad \text{and} \quad p_2 = \frac{2 - \sqrt{4 - 4\frac{r}{n-1}}}{2}.$$

Note, however, that these roots only exist if the quantity under the square root is positive, i.e., if

$$\begin{aligned} 4 - 4\frac{r}{n-1} &\geq 0 \\ \iff 1 - \frac{r}{n-1} &\geq 0 \\ \iff 1 &\geq \frac{r}{n-1} \\ \iff n-1 &\geq r \end{aligned}$$

This is expected, since we can never hope to achieve a contact rate of r if D has strictly less than $r+1$ persons. (Every person in D can meet with at most $n-1$ persons.)

Further note that, of the two possible roots p_1 and p_2 , it suffices to consider only p_2 . Indeed, we are only interested in those values of p that lie within the interval $[0, 1]$. Note that $p_1 \geq 1$, and when $p_1 = 1$ (which is achieved when $r = n-1$) then $p_2 = p_1 = 1$ (i.e., in that case there is only a single root). Therefore, the correct value of p is

$$p = \frac{2 - \sqrt{4 - 4\frac{r}{n-1}}}{2}.$$

Why the probability that $b \in Y_a$ is p . We next prove formally that, for any $a \in D$ and any $b \in D \setminus \{a\}$ the probability that $b \in Y_a$ in step (2.b) of the algorithm, is p .

For the sake of the development, fix $a \in D$ and $b \in D \setminus \{a\}$. Note that, in particular, this hence implies that $n \geq 2$.

We will need the following known equalities.

- If X is a random variable such that $X \sim B(n-1, p)$, then the probability that X equals a specific value $k \in \mathbb{N}$ is as follows.

$$P(X = k) = \begin{cases} \binom{n-1}{k} p^k (1-p)^{n-1-k} & \text{if } 0 \leq k \leq n-1 \\ 0 & \text{otherwise} \end{cases} \quad (8)$$

- For any real values $x, y \in \mathbb{R}$ and any natural number $m \in \mathbb{N}$ it holds that

$$(x+y)^m = \sum_{l=0}^m \binom{m}{l} x^l y^{m-l} \quad (9)$$

Now note that $P(b \in Y_a, K_a = k)$ is the probability that $b \in Y_a$ and the value of K_a drawn in step (2.a) is k . Equivalently, $P(b \in Y_a, K_a = k)$ is the probability that both (i) $b \in Y_a$ and (ii) $|Y_a| = k$.

Then the quantity that we seek, $P(b \in Y_a)$, is simply $P(b \in Y_a, K_a)$, marginalized over the possible values of K_a :

$$P(b \in Y_a) = \sum_{k \in \mathbb{N}} P(b \in Y_a, K_a = k) \quad (10)$$

$$= \sum_{k \in \mathbb{N}} P(b \in Y_a | K_a = k) P(K_a = k) \quad (11)$$

$$= \sum_{k=0}^{n-1} P(b \in Y_a | K_a = k) P(K_a = k) \quad (12)$$

The second equality is by definition of conditional probability; the third is because $K_a \sim B(n - 1, p)$ and therefore $P(K_a = k) = 0$ for $k \geq n$ by (8).

Let us next calculate the conditional probability $P(b \in Y_a | K_a = k)$. This is the probability that $b \in Y_a$ when $K_a = |Y_a| = k$. I.e., it is the probability that b is in a random k -sized subset of $D \setminus \{a\}$. Now note:

- there are $\binom{n-1}{k}$ possible subsets of $D \setminus \{a\}$ of size k ;
- when $k = 0$, there is no subset of $D \setminus \{a\}$ that contains b , since the only subset of size 0 is the empty set.
- when $k \geq 1$ there are $\binom{n-2}{k-1}$ possible sets $Y_a \subseteq D \setminus \{a\}$ of size k that contain b .

Indeed, observe that every set $Y_a \subseteq D \setminus \{a\}$ that contains b is of the form $Y_a = \{b\} \cup Y'_a$ with $Y'_a \subseteq D \setminus \{a, b\}$. If Y_a is of size k , then Y'_a is of size $k - 1$. Therefore, the number of k -sized sets Y_a with $b \in Y_a$ equals the number $k - 1$ sized subsets of $Y'_a \subseteq D \setminus \{a, b\}$, of which there are $\binom{n-2}{k-1}$.

Combining these three points, we conclude that, for every $0 \leq k \leq n - 1$:

$$P(b \in Y_a | K_a = k) = \begin{cases} 0 & \text{if } k = 0 \\ \frac{\binom{n-2}{k-1}}{\binom{n-1}{k}} & \text{if } 1 \leq k \leq n - 1. \end{cases} \quad (13)$$

We now continue our derivation of $P(b \in Y_a)$, and plug in (8) and (13) in

(12):

$$P(b \in Y_a) = \sum_{k=0}^{n-1} P(b \in Y_a \mid K_a = k) P(K_a = k) \quad (14)$$

$$= \sum_{k=1}^{n-1} \frac{\binom{n-2}{k-1}}{\binom{n-1}{k}} \binom{n-1}{k} p^k (1-p)^{n-1-k} \quad (15)$$

$$= \sum_{k=1}^{n-1} \binom{n-2}{k-1} p^k (1-p)^{n-1-k} \quad (16)$$

$$= \sum_{l=0}^{n-2} \binom{n-2}{l} p^{l+1} (1-p)^{n-1-(l+1)} \quad (17)$$

$$= p \sum_{l=0}^{n-2} \binom{n-2}{l} p^l (1-p)^{n-2-l} \quad (18)$$

$$= p(p + (1-p))^{n-2} \quad (19)$$

$$= p 1^{n-2} \quad (20)$$

$$= p \quad (21)$$

Here, (17) is by re-indexing with $l = k - 1$, and (19) is by application of (9) with $x = p$, $y = 1 - p$ and $m = n - 2$.