A The first program

We want to convert the individual-based model described in the sections 2 and A.1 to a simulation program. To do this, we will first construct a design for the simulation program that can be used during programming. This first design of the program structure will be a 'simple' one. We will often use very simple and straightforward methods, which are usually not the most efficient. The reason to start the program in a 'simple' way is to prevent mistakes and to prevent unnecessary optimizations. When we have a simple working code, we will start optimizing the code step-by-step, mostly focussed on the running time and memory usage. When we have optimized the code to our satisfaction, we will give an overview of the final program structure.

The road to the construction of the final code will be shown in the following appendices:

- appendix A: the program structure for the simple program
- appendix B: the optimization of the program
- appendix C: the program structure of the final program

In this appendix we will show the first simple design of the simulation program. First we will give an overview of the input parameters in appendix A.1. Next, in appendix A.2, we will give the structure of the program in an Unified Modeling Language (UML) diagram. In appendix A.3 a brief overview is given of what the main program will look like. Lastly, in appendix A.4 we will discuss the functions introduced in the UML diagram and we will give a short explanation for each of these functions.

A.1 Input

An overview of the input parameters can be found in table 1, where we will use the following notations:

$i \in \{S, I_1, I_2, T\}$	the infection stage
$g \in \{m, f\}$	the gender
$h \in \{c, s\}$	the type of partnership

	Parameter	Description	number
Network	N_0	mean population size	1
	x	the ratio male : female (1 : x)	1
	homo	heterosexual (0) or homosexual (1) population	1
	$\mu(i)$	rate death	4
	$\rho(h)$	rate partnership formation	2
	$\sigma(h)$	rate partnership separation	2
	$Z(g, h)$	partnership capacity distributions	4
Infection	$\beta(g, h, i)$	transmission rate ($i \in \{I_1, I_2\}$)	8
	$\alpha(i)$	rate enter stage $i \in \{I_2, T\}$	2
	nr_infected(i)	the number of initial individuals in infection stage $i \in \{I_1, I_2, T\}$	3
Spin-Up	*spin_up_nr_events*	the number of events the network needs to remain stable	1
	spin_up_tol	the tolerence in which the network needs to remain stable	1
	spin_up_max_events	the maximum number of events of the spin-up	1
Rest	*max_time*	the maximum time	1
	max_events	the maximum number of events	1
	nr_save	number of times to save output	1

Table 1: Input parameters for first program.

These input parameters can be chosen in such a way that different populations can be formed, with the generalizations introduced in 'on' or 'off':

- In case of a homosexual population, the population will consist of only males. The parameter *homo* = 1, $x = 0$ and all female parameter will be equal to zero: $\beta(i, f, h) = 0$ and $Z(f, h) = 0$ for all i and h.

- In case only one type of partnership is wanted, the parameters of one of the types of partnership can simply be equal to zero.

- In case less infection stages are wanted, the rate to enter certain stages can simply be equal to zero.

- In case no variation is wanted in the partnership capacity of individuals, the distributions $Z(g, t)$ can be chosen such that it will always return the same value.

We have chosen to take a constant number of four infection stages (S, I_1, I_2, T) and two partnership types (casual and steady) in the program. If we want less infection stages or types we simply put their parameters to zero, as explained before. We realise that taking a constant number is a weakness of the program, but it makes the programming a lot easier. Making the number of infection stages and the number of partnership types a variable, would be a great extension of the program. But since we do not have immediate plans to consider more stages or types, and

four infection stages and two partnership types is for now more than enough, we do not give this a very high priority.

A.2 The UML diagram

We have chosen for object-oriented programming. Therefore the the program will include classes. A *class* is a kind of blue print from which objects are created. For example we will create a class 'Individual' which says that each individual will need a name, gender etc. Each individual will be created according to this class and it will become an *object* of type 'Individual'.

To give a complete overview of the classes in our program, we create an UML (Unified Modeling Language) diagram. In an UML diagram each class is displayed as a rectangle containing three compartments. The first compartments shows the class name, the second compartment its attributes and the third compartment its functions. The attributes and functions of a class can be either private ($-$) or public ($+$).

In the individual-based program we create the classes Population, Individual and Partnership. The class Population has many attributes. Most importantly it consists of a list of references to all the individuals in the population. But it has also attributes such as birth rate, death rate, ratio male : female etc. To create a population according to the class Population, the input parameters of the program are used. All the individuals in the population are created according to the class Individual. Each individual has attributes such as a name, gender, maximum number of binding sites, a list of references to its partnerships etc. The list of references to partnerships, just mentioned as an attribute of an individual, consist of partnerships created according to the class Partnership. Each partnership has two attributes: a list of references to the individuals in that partnership and the type of partnership. A complete overview of the classes, with all its attributes and functions, is shown in the UML diagram in fig. 1.

The association between the classes is shown by lines. Each line also has two numbers with denotes the numbers of objects associated. For example each population consist of zero or more individuals ($0..*$) and each individual belongs to precisely one population (1).

Besides the previously discussed classes Population, Individual and Partnership there are other classes shown in the UML diagram. The classes g,h and i are made to create enumerations. With enumerations a set of strings are bound to constant values. For example the gender g can either be male m or female f, but with enumeration a value can be bound to these two strings: $m = 0$ and $f = 1$. The reason for using enumerations is to avoid errors from using invalid strings and it makes it easier to add or change values in the future.

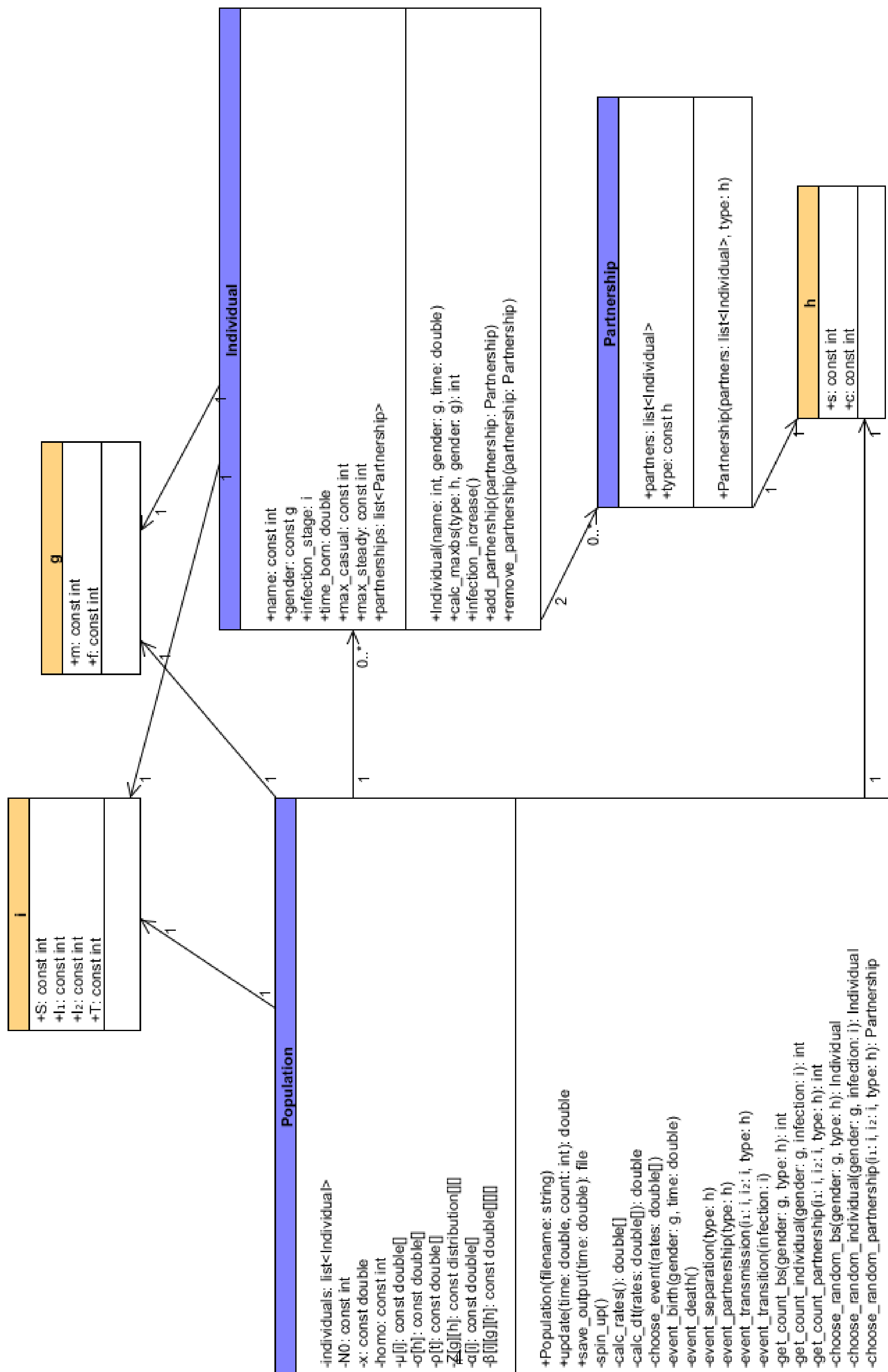
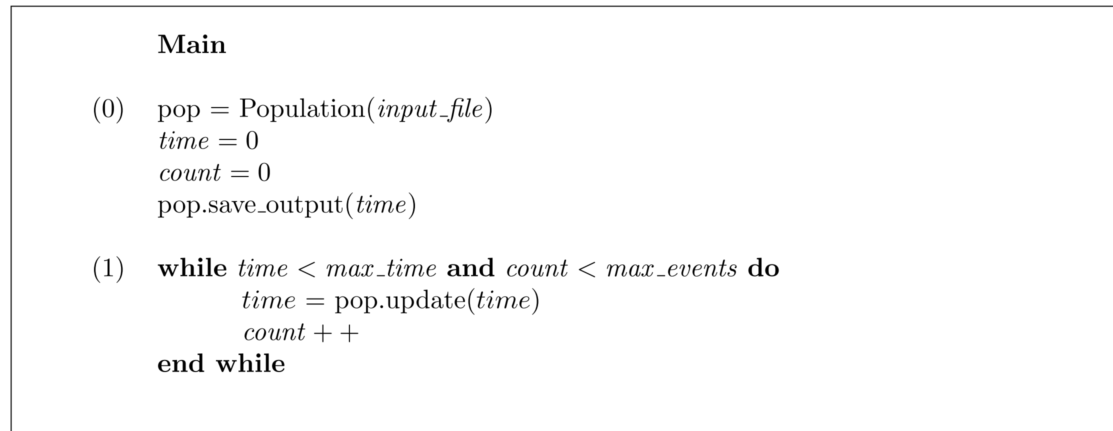


Figure 1: UML diagram of the program structure

A.3 The main program

The main program will consist of making the initial population with the constructor function Population() of the class Population. To create the initial population we will use the input parameters saved in a file. After the population is made, the function update() of the class Population will be repeatedly called until the maximum time or the maximum number of events has been reached. For explanation of the function update(), the constructor function Population() and all other functions shown in the UML diagram see the appendix A.4.



Main

(0) pop = Population(*input_file*)
time = 0
count = 0
pop.save_output(*time*)

(1) **while** *time* < *max_time* **and** *count* < *max_events* **do**
    time = pop.update(*time*)
    count + +
end while

A.4 The functions

In this section we will explain the functions introduced in the UML diagram. The functions have been divided into their respective classes. For some functions their actions are dependent on whether a homosexual or a heterosexual population is considered. Therefore in some functions the two different cases are discussed separately.

A.4.1 Class Population

Population(filename: string)

This is the constructor function for the class Population. A file is given as input which consists of all the needed parameters, such as the number of individuals, the number of initially infected individuals, the ratio male : female ($1 : x$) etc. We construct the population in the following steps:

- Create an empty population, but set its attributes according to the input file
- Create the individuals for the population
- Create a stable network with the use of a 'spin-up' (**spin_up()**)
- Introduce the infection

All in all, the constructor function creates a population with a stable network and a newly introduced infection.

spin_up()

The partnerships of the initial population are formed during a 'spin-up'. It lets the program run without infection until the degree distribution stabilizes. It results in a population with a stable network. The spin-up is explained in more detail in section 4.1.

update(time: double): double

In this function the following takes place:

- The new rates at which the events occur are calculated (**calc_rates()**)
- The time till the next event is calculated (**calc_dt()**) and the time will be updated
- An event is chosen and takes place (**choose_event()**)
- Every so many events certain results are saved to a file (**save_output()**)

The updated time is given as output of this function.

calc_rates(): double[]

This function calculates the rates λ_i at which the events occur. See section 3.6 for a complete overview of the events and their corresponding rates for a heterosexual population.

For a homosexual population the events and their rates are very similar to that of a heterosexual population. The only differences are that all the rates of the events concerning females are equal to zero and four different transmission are considered: from a male in stage I_1 to a susceptible male and from male in stage I_2 to a susceptible male in both steady and casual partnerships.

The output of this function is all the calculated rates.

calc_dt(rates: double[]): double

This function calculates the time till the next event. As input the rates λ_i are given. We have assumed that the events occur according to a Poisson process (see section 2.3). A random number r is generated and the time till the next event dt will be calculated:

$$dt = \frac{-\log(r)}{\sum_i \lambda_i}$$

As output the time step dt is given.

choose_event(rates: double[])

Every event occurs with a certain probability that can be calculated with the rates that are given as input. A random event is chosen according to the calculated probabilities, and the chosen event takes place.

save_output(time: double[]): file

This function calculates the wanted output of the simulation program and saves it in one or multiple files. The time is also saved in the output so that it is clear which results belong to which time. To start we want to save the number of individuals of gender g in infection stage i, for all g, i. We could later extend this output to save for example the entire network structure.

event_birth(gender: g, time: double)

A new individual of gender g is created and is entered in the population list. The individual will have the name nr and infection stage S. Every time a new individual has entered the population, we add $nr = nr + 1$ such that every individual has an unique name. For every newborn individual the number of steady and casual binding sites are calculated according to the distributions $Z(g, c)$ and $Z(g, s)$. Also the time that the individual is born is saved as an attribute for the individual. Since a newborn individual has no partners, its list of partnerships will be empty.

event_death(gender: g)

A random individual of gender g is removed from the population. Not only must the reference to the individual be removed from the list of the population, but also all references to the partnerships it is involved in.

event_separation(type: h)

A random partnership of type h is chosen. This chosen partnership will be removed and references to this partnership, in the list of partnerships of both individuals involved, will be deleted.

event_partnership(type: h)

In case of a heterosexual population:
A random free binding site of type h belonging to male is chosen, and a random free binding site of type h belonging to a female is chosen. We need to check if these two binding sites can form a partnership. If a partnership already exist between these two individuals, two new random free binding sites will be chosen. This will be repeated until a partnership is formed.

In case of a homosexual population:
Two random free binding sites of type h are chosen. If a partnership does not yet exist between these two individuals and the two binding sites do not belong to the same individual, a partnership is formed. Otherwise two new random binding sites are chosen.

event_transmission(i_1: i, i_2: i, type: h)

In case of a heterosexual population:
A random partnership of type h is chosen where the male is in infection stage i_1 and the female in infection stage i_2. However one individual needs to be infectious and the other susceptible. In this partnership transmission occurs and the individual in infection stage S transits to infection stage I_1.

In case of a homosexual population:
A random partnership of type h is chosen where one individual is in infection stage $i_1 \in \{I_1, I_2\}$

and the other is in infection stage $i_2 = S$. In this partnership transmission occurs and the individual in infection stage S transits to infection stage I_1.

event_transition(infection: i)

A random individual is chosen in infection stage $i \in \{I_1, I_2\}$. Its infection stage is changed from I_1 to I_2 or from I_2 to T, depending on its current infection stage.

get_count_bs(gender: g, type: h): int

This function counts the number of free binding sites of type h belonging to individuals of gender g. For g and h the string *None* can be chosen if there is no preference. For example get.count(m, *None*) will give the number of all male free binding sites (both steady and casual).

get_count_individual(gender: g, infection: i): int

This function counts the individuals of gender g in infection stage i. Again g and i can be *None*.

get_count_partnership(i_1: i, i_2: i, type: h): int

In case of a heterosexual population:
This function counts the number of partnerships of type h with the male in infection stage i_1 and the female in infection stage i_2.

In case of a homosexual population:
This function counts the number of partnerships of type h with one individual in infection stage i_1 and the other in infection stage i_2.

Where i_1, i_2 and h can be *None*.

get_random_bs(gender: g, type: h): Individual

This function returns a random free binding site of type h belonging to an individual of gender g. Where g and h can be *None*. It returns the individual to which the binding site belongs.

get_random_individual(gender: g, infection: i): Individual

This function returns a random individual of gender g in infection stage i. Where g and i can be *None*.

get_random_partnership(i_1: i, i_2: i, type: h): Partnership

In case of a heterosexual population:
This function returns a random partnership of type h with the male in infection stage i_1 and the female in infection stage i_2.

In case of a homosexual population:
This function chooses a random partnership of type h with one individual in infection stage i_1 and the other in infection stage i_2.

Where i_1, i_2 and h can be *None*.

A.4.2 Class Individual

Individual(name: int, gender: g, time: double)

This is the constructor function of the class Individual. A new individual is created with the name, gender and the time it was born given as input. The maximum number of casual and steady binding sites for this new individual are calculated. The infection stage for a newborn is always susceptible and it will have zero partners.

calc_maxbs(type: h, gender: g): int

This function calculates the number of binding sites of type h for a newborn individual of gender g. The number of binding sites of type h for gender g are distributed according the random variable $Z(g, h)$.

This function returns the chosen number of binding sites of type h.

infection_increase()

This function increases the infection stage of the individual to the next infection stage.

add_partnership(partnership: Partnership)

The partnership given as input will be added to the list of partnerships of the individual.

remove_partnership(partnership: Partnership)

The partnership given as input will be removed from the list of partnerships of the individual.

A.4.3 Class Partnership

Partnership(partners: list<Individual>, type: h)

This is the constructor function of the class Partnership. A partnership of type h is formed between the list of individuals given as input.

B Optimizing

We have constructed a design for the simulation program that we have used during the programming of the first version, which can be found in appendix A. We talk about a 'first version' of the program, because we started the programming very straightforward and try to optimize and improve the code step by step. This optimization process will be described in this appendix.

First in appendix B.1 we will explain the optimization approach and the reasoning for this approach. In sections B.2 - B.5 we discuss the optimization of certain functions. In appendix B.7 we consider other data-structures and compare them to the current data-structure.

B.1 Start: simple program

We started programming in the most simple way as explained in appendix A. But the most simple way to program does not mean it is also the most efficient way. One of the reasons for keeping the level of difficulty of programming to a minimum is to prevent mistakes. If we start with a simple working code and optimize the code step by step, instead of programming the more efficient (and likely more difficult to program) code in one step, we decrease the chances of mistakes. Another advantage is that starting with a simple code and optimizing this code step by step, we will prevent premature optimization and we will be able to thoroughly investigate which parts of the code are in most need of optimization. Optimizing the code can either mean decreasing the running time or the memory usage of the code, although we will mostly focus on the running time. By using the Python Profilers we can see how often and for how long various parts of the program are executed [18]. This way we can see which functions take up the most of the running time, we can try optimize these functions, and restart profiling the code. We will continue this process until we have the opinion that no further optimization is possible or needed. We have to keep in mind while optimizing the functions, that the code has to remain user friendly and easy to read. Therefore we will sometimes need to ask ourself: is the optimization worth it?

For the first, simple program we have followed the guidelines of the program structure explained in appendix A. To understand the future optimizations, it is important to know that every time an event has occurred the program recalculates

- $N(g, i)$ = the number of individuals of gender g and in certain infection stage
- $P(h, i_1, i_2)$ = the number of partnerships of type h and the individuals involved in infection stages i_1, i_2
- $F(g, h)$ = the number of free binding sites of type h belong to an individual of gender g

by going through the entire population and counting the individuals, partnerships or free binding sites that satisfy the given criteria. When choosing a random individual, partnership or free binding site satisfying certain criteria we generate a random number between 1 and the total number of individuals, partnerships or free binding sites satisfying the criteria. Then we will go through the population and count the number of individuals, partnerships or free binding sites satisfying the criteria until we reach the generated random number.

These are of course not the most efficient methods, but now that we have a working code let us see where we can start optimizing.

B.2 Optimizing: calculating N, P, F

Using Python Profilers, we see that the functions **get_count_bs()**, **get_count_partnership()** and **get_count_individual()** (see appendix A.4.1) are very time consuming. This was to be expected because of the way it is programmed now: we go through the entire population and have to access attributes of each individual. For example to calculate $N(g,i)$ we have to access for each individual the attributes that represent its gender and its infection stage. Accessing an objects attributes is not very expensive to do, but since we have to do this for each individual in the population, the cumulative time can become very high for large populations. We also have to repeat the functions very often. Every time an event has occurred N, P and F have to be recalculated. Therefore optimizing these functions can greatly increase the speed of our code.

We will try to calculate N, P and F faster after each event by taking into account which event has occurred and the consequences the event has on N, P and F. For example if an individual dies, we need to subtract this individual from N, its free binding sites from F and its partnerships from P. Updating N, P and F after each event prevents the need to recalculate N, P and F by going through the entire population.

We however keep the function **get_count_bs()**, **get_count_partnership()** and **get_count_individual()** in our code so it is possible to check if the updating has gone well. Updating N, P and F is more error prone than the functions. Therefore the functions are an excellent way to check if we have made no mistakes with programming this faster method.

B.3 Optimizing: choose random individual, partnership or free binding site

Now that we have optimized our previous most expensive functions, we will check what the new most time-consuming functions are. Next we see that other time consuming functions are **get_random_bs()**, **get_random_partnership()** and **get_random_individual()**. For these functions we also have to go through the population and access the objects attributes until the chosen random individual, partnership or free binding site is reached.

Suppose we need to chose a random individual to take part in a partnership. The reason we can not just chose a random individual from the list of individuals is that we need to take into account the criteria the individual needs to satisfy. For example the gender of the individual can be one of the criteria. We also need to take into account that individuals have different numbers of free binding sites and some individuals may not even have any more free binding sites. Therefore each individuals has a certain chance of being chosen.

One way to prevent going through the population to chose a random free binding site is to make a list of references to all individuals with free binding sites. The frequency the individual is in the list is equal to the number of free binding sites the individual has. Therefore the more free binding sites the individual has, the larger the chance the individual gets chosen. However take into account that the free binding sites have the criteria gender g and type h. Therefore the free binding sites are sorted into $2 \cdot 2 = 4$ lists.

We can do the same for the individuals and partnerships. The individuals sorted on infection stages, so in 4 different lists. The partnerships sorted on type and infection stages of both individuals, so $2 \cdot 4 \cdot 4 = 32$. But as you can see, with this new way, we have to store many

lists of references to objects and we also have to update them after every event. Therefore this method is unfortunately not yet as fast as we wish, but it does give us the possible optimization described in the next section.

B.4 Optimizing: remove from list

The most expensive action is now a build-in python function: the removal of an element from a list. This is caused by the many lists that we haved introduced in the previous optimization step, which we need to keep up-to-date. Deleting an item from a list is a $\mathcal{O}(n)$ operation with the python build-in remove function. The time-complexity of various operations that we discuss in this section are obtained from [21]. Removing the last item in the list is only a $\mathcal{O}(1)$ operation. Therefore it is cheaper to swap the item you want to remove with the item at the end of the list. Since the order of the items in the list is of no importance to us, swapping causes no problem. To swap the items we will need to know the index value of the item that we want to remove from the list. We can find its index value with the build-in python function, but that will cost $\mathcal{O}(n)$. A cheaper way is to save the index values, or *hash*-values, as a an attribute of the object. This way we know the places where all the references of the object are saved, and when you have the index-value, it is only $\mathcal{O}(1)$ to access the reference.

To summarize we propose the following procedure to delete a reference to an object from a list:

- access the index value of the object (saved as an attribute of the object)
- swap the reference on this index in the list with the last reference in the list
- delete the last reference in the list
- update the index-values of the objects of both references

This may seem like a lot of work, but most of these operations cost only $\mathcal{O}(1)$. Therefore for large populations this proposed removal procedure will save a lot of time.

B.5 Optimizing: calculating the rates

The most time-consuming function is now **calc_rates()**. This function is called after every event, to recalculate the rates. One option to optimize this function, and for other functions as well, is to use Cython. Cython makes it possible to write C extensions for Python, which gives the combined power of Python and C [20]. We have written a C-extension for the function **Calc_Rates()** and see the function is approximately twice as fast as the original function. A great advantage of the use of Cython is that it is an optimization tool that can be used to speed up different functions in the code.

Another option to speed up the calculation of the rates is to not recalculate all the rates after every event. The happening of an event will not change all the rates of the events, but most likely only a few. If we update only the changed rates, by taking into account what changes has been made in the population, this may reduce the needed calculation time.

We have tried both methods. The first method, using Cython for the function **calc_rates()**, made the function approximately twice as fast. But the second method, only recalculating the changed rates in stead of all, was an even faster method. Thus we went for the second method.

However we kept the function **calc_rates()** in our code, so we are able to check during programming if the recalculations of the rates go well.

B.6 Optimizing: enumerations

A surprisingly expensive function is a build-in Python function that gets the value or string of an enumeration. As explained before in appendix A.2 we use enumerations to bind a string to a value. For example the enumeration 'gender' makes an object that binds the string *male* to the value 0. This is a very well-defined method, but unfortunately the program needs to ask the enumeration many times for the value or string of the object.

Therefore we have decided that we will not store enumeration objects, but values. For example if an individual is male, its gender will not be the gender object (that binds *male* to 0) but simply have value 0. The population will get a new attribute *genders* that will consist of the following list:

$$genders = [male, female]$$

If we want to know what the value 0 for a gender represents in the population, we simply ask the string on index 0 in the list *genders*. It also works the other way around: if we want to know the value of the string *male*, we ask for its index value in the list.

We will replace all the enumerations (for gender, infection stage and partnership type) with this new method. This new method is a lot less elegant, but also a lot less time consuming.

B.7 Optimizing: other data-structures

We have now reached a point in the optimization process where no longer one function stands out in the running time. Many functions now share the 'title' of the most time-consuming function, such as the creation and removal of individuals and partnerships from the population. The total running time for the program is however still very high, too high for large populations. We might consider that the choice of object-oriented programming may not have been the fastest and most memory friendly option.

Therefore we are going to discuss and compare three different ways to program the population. We will mostly compare the running time of the three methods, since that is at the moment the biggest problem of the code. First we will discuss each of the methods and its advantages and disadvantages. Then we will time certain operations on all three methods and see which is the best method to program the population.

B.7.1 Lists of objects

This method is our current idea: the original idea in appendix A, altered with the optimization steps in appendices B.2 - B.4. In this method we make objects and put them into a lists. Lists are not very memory friendly, but an array can not hold objects, so that is not an option. A very big advantage of this method is that we work with objects, which is very user friendly and it makes it very easy to alter the code.

A disadvantage is the requirement to keep and update the many lists. But let us not forget that the reason for introducing the many lists, was to be able to pick a random individual, partnership or free binding site with certain requirements for a very low cost. Choosing a random

item in the list is only a $O(1)$ operation, updating the lists is however more expensive.

To make deleting a reference from a list as cheap as possible, we use the previous explained 'swapping' method (appendix B.4) and save the index-value as an attribute of the object.

B.7.2 Array and bisection

For the second method we make use of arrays in stead of lists. Arrays are much more memory friendly then lists, since they do not change size. With an array you allocate the needed memory size beforehand. Since you need to allocate the memory, you already need to guess how large the population size will become. You need to save some extra space since the size of the population will change over time. When your population size grows larger then the allocated memory, you will receive an error and the program will stop. While with a list you can easily remove or append rows, this is not the case with arrays.

Bisection
To search for a random individual, partnership or binding site we use the bisection method. Assume for example that we want a random individual in infection stage I_2. We need to keep track of the total number of individuals in infection stage I_2 already encountered in the population. For example:

name	infection stage	total I_2
1	S	0
2	I_2	1
3	S	1
4	I_1	1
5	I_2	2
6	S	2
⋮		
100000	S	42

In this example there are 42 individuals in infection stage I_2. We chose a random integer r between 1 and 42. When using the left bisection method to find r we find the index of the first individual that has the total number equal to r. In other words the r-th individual in infection stage I_2.

An advantage is that the bisection-method is very fast, which makes finding the random individual or partnership very fast. However disadvantages are that the 'total column' needs to be updated if there are any changes. Illustrating this with our previous example: If the random chosen individual in infection stage I_2 transits to the next infection stage, the total number of individuals in infection stage I_2 already encountered in the population will be one less from that individual on. Therefore we need to subtract one many times.

B.7.3 Pandas

Pandas is a Python library providing high-performance, easy-to-use data structures and data analysis tools for the Python programming language [19]. This method is by far the most easy and user-friendly. In the previous two methods, you needed to update the lists (first method) or the column of total numbers (second method) when an event has occurred. With the method using Pandas there is no need to keep track and update such data. Since pandas is specialized in

big data, when we want a random individual, partnership or binding site with certain properties, we simply ask the library to construct a sublist of all that satisfy the properties. Then we can simply choose a random element from that list.

B.7.4 Performance data-structures

We will now test all three methods on two operations:

- Choosing a random individual in a certain infection stage
- Removing a given individual (and all the updating that may be needed in the first two methods)

Choosing random individual		
1000×	$N = 10.000$ time (sec)	$N = 100.000$ time (sec)
List	0.027	0.037
Array	0.091	0.131
Pandas	5.683	18.234

Table 2: The time (in seconds) needed to choose a random individual in a certain infection stage a thousand times. Calculated for both populationsizes 10.000 and 100.000, and for the three methods described in appendices B.7.1 - B.7.3.

Remove individual		
1000×	$N = 10.000$ time (sec)	$N = 100.000$ time (sec)
List	0.024	0.037
Array	0.420	2.652
Pandas	5.538	35.621

Table 3: The time (in seconds) needed to remove a given individual from the population a thousand times. Calculated for both populationsizes 10.000 and 100.000, and for the three methods described in appendices B.7.1 - B.7.3.

We can see from the calculated times in table 2 and table 3 that Object-Oriented programming and storing the references to the objects in lists seems to be the fastest option of the three.

B.8 Optimizing: other programming language

There is one big optimization option we have not yet mentioned: choosing another programming language. Although the assignment was to write the program in Python, it is still very interesting to see how another language would fare.

Python is know to be very slow, because it is an *interpreted* language. The program runs on the CPU, but for the CPU to understand the code it needs to be translated to *machine* code. Unlike *compiled* languages such as C, interpreted languages do not directly compile to machine

code. Therefore Python tends to be 10-100 times slower than C. [22]

Unfortunately we do not have the time to rewrite the code in another programming language. But it is definitely something to keep in mind for future projects. Python is easy to read and write, but you pay for it with its slow speed. Therefore it is advisable to not use Python for a program with a large running time, such as ours.

B.9 End: times final program

We have considered a lot of optimization steps and have come to the final program. We will measure the execution time of this program for the baseline parameter values used in section 5.1 and investigate how the population size influences its performance. The results are shown in table 4. During the optimization steps we have tried to ensure that the costs of the events do not depent on the size of the population, for example by saving the index values and introducing the 'swap method'. We see in the measured times that the mean time per event does indeed not increase in the same degree as the population size. However it does increase some when the population size grows in size, telling us that the costs of the events are not completely independent of the population size.

Please note that increasing the population size increases the number of events that have to take place to let the same number of years pass in the population. Therefore increasing the population size does increase the total runtime of the program. The number of events, and therefore the runtime of the program, that have to occur to let a certain amount of years pass in the population strongly depend on the parameter values. We have only varied the population size in table 4 and kept the other parameters constant. But we expect that changing the other parameter values will mostly change the number of events, not the mean time per event. Therefore table 4 will still be informative.

Performance			
		N	
	1.000	10.000	100.000
execution time	89.403	948.725	9548.464
nr of events	727.967	7.610.427	75.529.017
mean time per update	0.000107	0.000123	0.000135
mean time per event birth	0.0000751	0.0000745	0.0000822
mean time per event death	0.000109	0.000109	0.000114
mean time per event partnership formation	0.0000826	0.0000841	0.0000918
mean time per event partnership separation	0.000120	0.000119	0.000129
mean time per event transmission	0.0000711	0.0000668	0.0000670
mean time per event transition	0.0000612	0.0000582	0.0000578

Table 4: The execution times (in seconds) and number of events for a run of 5.000 years. With the input parameters set to table 7 (in the main text) and scenario $(n_f, n_m) = (4, 4)$.

C The final program

During the optimization in appendix B and the alterations in section 4 we have made a lot of changes to the program. Therefore we will give a new overview of the program, to make it easier to read and understand the new code. The program structure will be explained in a similar way as appendix A. First the input-parameters will be given in appendix C.1. The UML-diagram will be shown in appendix C.2. We will clarify the UML-diagram by explaining the attributes in appendix C.3 and the functions in appendix C.4.

C.1 Input parameters

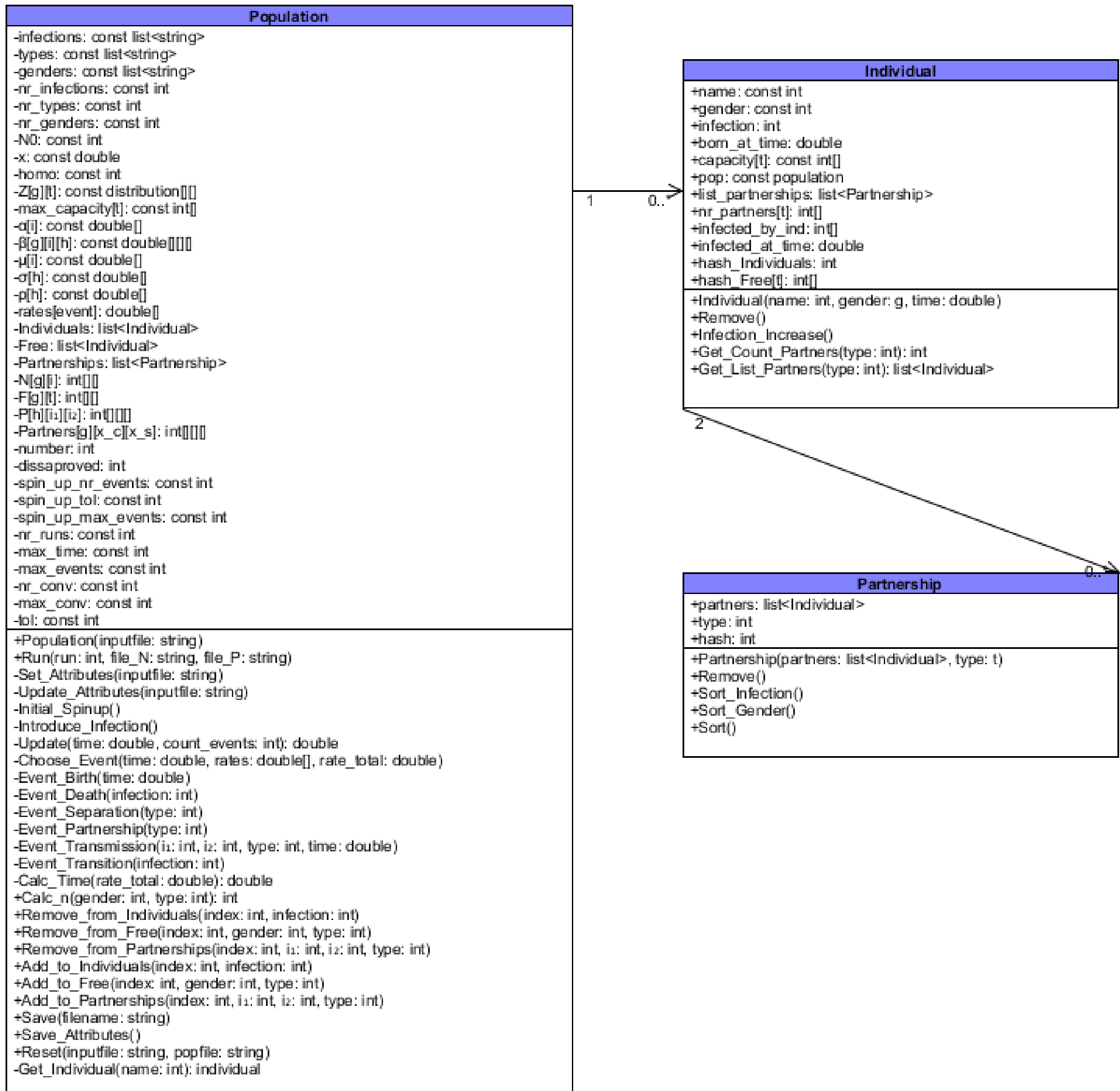
The program has the input parameters shown in table 5.

	Parameter	Description	number
Network	N_0	mean population size	1
	x	the ratio male : female (1 : x)	1
	homo	heterosexual (0) or homosexual (1) population	1
	$L(i)$	the mean sexual active lifetime of an individual in infection stage i	4
	$d_P(h)$	the mean duration of a partnership of type h	2
	$\theta(h)$	the mean number of lifetime partners of type h	2
	$Z(g,h)$	partnership capacity distribution of type h for individuals of gender g	4
Infection	$\beta(g,h,i)$	transmission rate ($i \in \{I_1, I_2\}$)	8
	$d_I(i)$	the mean duration of infection stage $i \in \{I_1, I_2\}$	2
	$nr_infected(i)$	the number of initial individuals in infection stage $i \in \{I_1, I_2, T\}$	3
Spin-Up	*spin_up_nr_events*	the number of events the network needs to remain stable	1
	spin_up_tol	the tolerence in which the network needs to remain stable	1
	spin_up_max_events	the maximum number of events of the spin-up	1
Stop	*nr_runs*	the number of simulation runs	1
	{ *max_time*	the maximum time per run	1
	{ *max_events*	the maximum number of events per run	1
	{ *nr_conv*	the number of events the infection needs to remain stable	1
	{ *tol*	the tolerence in which the infection needs to remain stable	1
	{ *max_conv*	the maximum number of events of the stabilization of the infection	1
Save	*nr_save*	number of times to save certain output per run	1
	nr_save_pop	number of times to save entire network per run	1

Table 5: Input parameters for final program.

C.2 UML-diagram

Figure 2: the UML-diagram



C.3 Attributes

C.3.1 Population

Constant Attributes Population		
Enumerations	*infections*	list of the possible infection stages
	types	list of the possible partnership types
	genders	list of the possible genders
	nr_infections	the number of possible infection stages
	nr_types	the number of possible partnership types
	nr_genders	the number of possible genders
Network	N_0	the initial population size before spin-up
	x	the ratio male : female (1 : x)
	homo	the population is heterosexual (0) or homosexual (1)
	$\mu(i)$	the death rate of an individual in infection stage i
	$\sigma(h)$	the dissolution rate of a partnership of type h
	$\rho(h)$	the rate of the formation of a partnership of type h
	$distrib(g, h)$	the partnership capacity distributions, a list of cumulative probabilities
	$max_capacity(h)$	the maximum partnership capacity for type h
Infection	$\alpha(i)$	the transition rate of infection stage i to $i + 1$
	$\beta(g, i, h)$	the transmission rate of an individual of gender g in in infection stage i in a partnership of type h
	$nr_infected(i)$	the number of initial infected individual in infection stage i
Spinup	*spin_up_nr_events*	the number of events that the spin-up needs to be stable
	spin_up_tol	the tolerance of the spin-up
	spin_up_max_events	the maximum number of events of the spin-up
Stop	*nr_runs*	the number of runs
	max_time	the maximum time in years for each run
	nr_events	the maximum number of events for each run
	tol	the tolerance in which the infection needs to be stable
	nr_conv	the maximum number of events the infection needs to be stable
	max_conv	the maximum number of events the infection needs to be stable

	Variable Attributes Population	
Lists of ref	*Individuals*(i)	list of references to individuals in the population, sorted on infection stage.
	Partnerships(h, i_0, i_1)	list of references to partnerships in the population, sorted on type and infection stages of the individuals.
	Free(g, h)	list of references to individuals with a free binding site, sorted on type of binding site and the gender of individual. (there is one reference for each free binding site)
Count of lists	$N(g, i)$	array with the number of individuals of gender g in infection stage i
	$P(h, i_0, i_1)$	array with the number of partnerships of type h between individuals in infection stages i_0 and i_1
	$F(g, h)$	array with the number of free binding sites of type h belonging to an individual of gender g
Rest	*rates*	list of the rates of all the possible events
	Partners(g, x_c, x_s)	array with the number of individuals of gender g with x_c casual partners and x_s steady partners
	number	the 'name' of the next individual that enters the population
	dissaproved	the number of dissaproved partnerships
	infected	the cumulative number of infected individuals

C.3.2 Individual

	Attributes Individual	
Constant	*name*	the name of the individual
	gender	the value of the gender of the individual
	capacity(h)	the partnership capacity of type h
	pop	the population to which the individual belongs
	born_at_time	the time at which the individual is born
Variable	*infection*	the value of the infection stage of the individual
	list_partnerships	list of all partnerships in which the individual is involved
	nr_partners(h)	the number of partnerships of type h in which the individual is involved
	infected_by_ind	the list of the name and infection stage of the individual that infected it this individual
	infected_at_time	the time at which the individual gets infected
	hash_Individuals	the index number of the individual in the population list *Individuals*
	hash_Free(h)	the list of index numbers of the free binding sites of type h in the population list *Free*

C.3.3 Partnership

	Attributes Partnership	
Constant	*partners*	the list of partners involved in the partnership
	type	the value of the type of the partnership
Variable	*hash*	the index value of the partnership in the population list *Partnerships*

C.4 Functions

C.4.1 Population

Population(inputfile: string)
The constructor function of the class Population. A population is created that is stable without infection and a newly introduced infection:

- First the function sets the attributes (except infection attributes α and β) and creates an empty population (**Set_Attributes()**)
- Second it makes an initial population, according to the input parameters, but without partnerships (**Initial_Spinup()**)
- Next partnerships are created by letting the program run without infection, until the degree distribution stabilizes (**Spin_up()**)
- Save this stabilized population so we can re-use this population if we want to do multiple runs with infection (**Save()**)
- Now that we have a population that is stable without infection, we introduce the infection (**Introduce_Infection()**)
- Finally we must reset the attributes involving infection (α and β) (**Update_Attributes()**)

Set_Attributes(inputfile: string)
This function reads the inputfile and sets all attributes according to the input parameters. The attributes α and β that involve infection are set equal to zero. The population itself is empty.

Update_Attributes(inputfile: string)
This function reads the inputfile and updates the attributes α and β which involve infection.

Initial_Spinup()
This function creates the initial population of size N_0 before the Spin-up, with all individuals in infection stage S and without partnerships. The number of males and females are decided by the population attributes x and *homo*.

Spin_Up()
This function runs the code without infection until the degree distribution stabilizes. In other words, until the number of individuals of gender g with n partnerships of type h stabilizes, for all possible g, n and h (see for further explanations section 4.1).

Introduce_Infection()
It infects random individuals in the population, independent of their gender. The number of individuals it infects and to what infection stage goes according the population attribute $nr_infected(i)$.

Reset(inputfile: string, popfile: string)
This function sets the attributes according to the inputfile and creates an empty population. The population is created according to the population saved in the *popfile*. This should be a stable population without infection, created by the previously explained spin-up. We introduce the infection, and just like the constructor function, this leaves us with a population that is stable without infection, and a newly introduced infection.

Run(run: int, file_N: string, file_Partners: string)
This is the main run of the program. It lets events occur (**Update()**) and keeps track of the time and the number of events that have passed. Every once in a while (according to the attributes *nr_save* and *nr_save_pop*) it saves either only certain output, or the entire network structure. This function either stops if the maximum time has been reached, or if the infection stabilizes (see for further explanations section 4.3).

Update(time: double, count_events: int): time: double
This function updates the population by letting an event occur:

- Fist calculate the total rate from the population attribute *rates*
- Calculate the time to the next event (**Calc_Time()**) and update the time
- Choose and event and let that event occur (**Choose_Event()**)

Choose_Event(time: double, rates: array, rate_total: double)
This function chooses an event according to the population attribute *rates* and lets the chosen event take place.

Save(file)
Saves the entire population to the file. From every individual the attributes *name*, *gender*, *infection*, *capacity*, *born_at_time*, *partners* separated by types, *infected_by_ind*, and *infected_at_time* are saved.

Save_Attributes()
This function saves some of the input parameters to a file. This is only needed once, since the input parameters remain the same for all runs. Therefore the file-name can remain the same, and is not needed to give as an input parameter of the function.

Event_Birth(gender: int, time: double)
This function creates a new individual of the given gender with the constructor function **Individual()** of the class Individual.

Event_Death(infection: int)
This function chooses a random individual with the given infection. The chosen individual is removed with the function **Remove()** of the class Individual.

Event_Separation(type: int)
This function chooses a random partnership of the given type. The chosen partnership will be removed from the population with the function **Remove()** of the class Partnership.

Event_Partnership(type: int)
This function choses two random free binding sites of either two males (in case of a homosexual population) or of both a male and a female individual (in case of a heterosexual population). When the two individuals are chosen to form a partnership, we need to check if the partnership is allowed. If either both the free binding sites belong to the same individual (which is possible in a homosexual population) or the two individuals are already involved in a partnership (no matter what type of partnership) the partnership will be disapproved.
When the partnership is disapproved, no partnership will be formed and a next event will be

chosen. We also keep track of the number of disapproved partnership by adding one to the population attribute *disapproved*. If the partnership is approved, the partnership will be formed with the constructor function **Partnership()** of the class Partnership.

Event_Transmission(infection_0: int, infection_1: int, type: int, time: double)
This function chooses a random partnership of the given type with:

- the individuals in the given infection stages (in case of a homosexual population)
- the male in the given infection stage infection_0 and the female in infection stage infection_1 (in case of a heterosexual population)

In this random partnership transmission of infection will take place, therefore the susceptible individual in the partnership will move to the next infection stage with the function **Infection_Increase()** of the class Individual. This function also saves, as attributes of the individual that gets infected, the name and current infection stage of the individual it was infected by and the time at which the individual gets infected.

Event_Transition(infection: int)
This function chooses a random individual in the given infection stage and increases this chosen individual infection stage with the function **Infection_Increase()** of the class Individual.

Calc_Time(rate_total: double): dt: double
This function calculates the time till the next event.

Calc_n(gender: int, type: int): n: int
This function calculates the partnership capacity of the given type for an individual of the given gender. The partnership capacities are calculated according to the distribution saved in the population attributes $distrib(g, h)$.

Remove_from_Individuals(index: int, infection: int)
This function removes the reference at the given index from the population list $Individuals(i)$ for i the given infection, and takes care of all the necesarry updates:

- Remove the individual from the list
  (by swapping the individual with the last individual in the list, and then removing the individual from the end of the list)
- Update the *hash* value of both individuals
- Update $N(i)$
- Update *rates*

Remove_from_Free(index: int, gender: int, type: int)
This function removes the reference at the given index from the population list $Free(g, h)$ for g the given gender and h the given type, and takes care of all necesarry updates:

- Remove the individual from the list
  (with the swapping method)
- Update the *hash_Free* value of both individuals

- Update $F(g, h)$
- Update *rates*

Remove_from_Partnerships(index: int, type: int, infection_0: int, infection_1: int)
This function removes the reference at the given index from the population list *Partnerships*(h, i_0, i_1) with h the given type and i_0 and i_1 the given infection stages, and takes care of all necesarry updates:

- Remove the partnership from the list
  (with the swapping method)
- Update the *hash* value of the partnership
- Update $P(h, i_0, i_1)$
- Update *rates*

Add_to_Individuals(ind: individual, infection: int)
This function adds the reference of the given individual at the end of the population list *Individuals*(i) for i the given infection,and takes care of all necesarry updates:

- Add individual to end of the list
- Update *hash* value of the individual
- Update $N(i)$
- Update *rates*

Add_to_Free(ind: individual, gender: int, type: int)
This function adds the reference of the given individual at the end of the population list *Free*(g, h) for g the given gender and h the given type, and takes care of all the necesarry updates:

- Add individual to end of the list
- Update *hash_Free* value of the individual
- Update $F(g, h)$
- Update *rates*

Add_to_Partnerships(partnership: partnership, type: int, infection_0: int, infection_1: int)
This function adds the reference of the given partnership to the population list *Partnerships*(h, i_0, i_1) with h the given type and i_0 and i_1 the given infection stages, and takes care of all the necesarry updates:

- Add partnership to end of the list
- Update *hash* value of the partnership
- Update $P(h, i_0, i_1)$
- Update *rates*

Get_Individual(name: int): ind: individual
This function finds and returns the individual with the given name.

Optional: Check functions
There are many population attributes we need to keep updating when an event has occured, such as the attributes $N(g, i)$, $P(h, i_0, i_1)$, $F(g, h)$ and $Partners(g, t_0, \dots, t_n)$. Therefore there are a few functions with only the function to check the correctness of these updates:

- **Calc_N()**
- **Calc_F()**
- **Calc_P()**
- **Calc_Partners**

These functions go through the entire population and calculate the true distributions N, F, P and *Partners*. Comparing these distribution to the population attributes will check if the updates of the attributes have gone correctly.

The function **Check()** will check the entire population. This function can for example be called at the end of the run to see if all is correct. It checks the correctness of the population attributes N, F, P, *Partners*, *Individuals*, *Free*, *Partnerships*, the individual attributes *nr_partners*, *hash_Individual*, *hash_Free* and the partnership attribute *hash*.

Optional: Calc_Rates()
This function calculates the rates of each possible event and saves them in the population attribute *rates*. This function is not used by the program, but remains in the code for possible checking during programming.

C.4.2 Individual

Individual(pop: population, name: int, gender: int, infection: int, time: double)
The constructor function of the class Individual. It creates a new individual with the given gender, given infection and born at the given time belonging to the given population. A newly constructed individual has no partnerships and is given the name *number* (the population attribute). When the name is used, add one to the population attribute *number* such that each individual has a unique name. The partnership capacities *capacity*(h) are calculated with the population function **Calc_n()** for all types.

A lot of population attributes need to be updated when a new individual enters the population:

- Add the new individual to *Individuals*(i) (**Add_to_Individuals()** from the class Population)
- Add its free binding sites to $Free(g, h)$ for all h (**Add_to_Free()** from the class Population)
- Add one to $Partners(g, 0, 0)$, since the new individual has no partners

Remove()
This function removes the individual from the population, in other words removes all references to the individual. This consists of the following steps:

- Dissolve all partnerships the individual is involved in (**Remove()** from the class Partnership)
- Remove its binding sites from *Free*(g, h) for all h (**Remove_from_Free()** from the class Population)
- Remove the individual from *Individuals*(i) (**Remove_from_Individuals()** from the class Population)

Infection_Increase()
This function increases the infection stage of the individual. Changing the individuals attribute *infection* is however not enough. The following population attributes also need to be updated:

- Remove all partnerships of this individual from the list *Partnerships*(h, i_0, i_1) (**Remove_from_Partnerships**), with the old infection stage of the individual.
- Add all partnerships of this individual to the list *Partnerships*(h, i_0, i_1) (**Add_to_Partnerships**), with the new infection stage of the individual.
- Remove the individual from *Individual*(i) (**Remove_from_Individuals**), with i the old infection stage
- Add the individual to *Individuals*(i) (**Add_to_Individuals**), with i the new infection stage.

Get_Count_Partners(type: int): count: int
This function counts and returns the number of partnerships of the given type that the individual is involved in.

Get_List_Partners(type: int): list_partners: list
This function returns the list of all the names of the partners of the individual, that are involved in a partnership of the given type with the individual.

C.4.3 Partnership

Partnership(ind_0: individual, ind_1: individual, type: int)
The constructor function of the class Partnership. It forms a partnership of the given type between the given individuals. The partnership needs to be added to the individuals attributes *list_partnerships*, such that each individual has a reference to the partnership. Since each individual gains a partner, we need to add one to the attribute *nr_partners*(h) of each individual. Also the following population attributes need to be updated:

- Remove one free binding site of each individual from *Free*(g, h) (**Remove_from_Free**)
- Subtract one from *Partners*(g, x_c, x_s) for each individual, with x_c and x_s its old number of partners
- Add one to *Partners*(g, x_c, x_s) for each individual, with x_c and x_s its new number of partners
- Add the partnership to *Partnerships*(h, i_0, i_1) (**Add_to_Partnerships**)

Remove()
The function removes a partnership from the population and performs all the necessary updates. The partnership needs to be removed from the individuals attributes *list_partnerships*. Since each individual loses a partner, we need to subtract one from the attribute *nr_partners*(h) of each individual. Also the following population attributes need to be updated:

- Add one free binding site of each individual to $Free(g, h)$ (**Add_to_Free**)
- Subtract one from $Partners(g, x_c, x_s)$ for each individual, with x_c and x_s its old number of partners
- Add one to $Partners(g, x_c, x_s)$ for each individual, with x_c and x_s its new number of partners
- Remove the partnership from $Partnerships(h, i_0, i_1)$ (**Remove_from_Partnerships**)

Sort_Infection(): sorted_list : list<individuals>
This function returns a list of the individuals in the partnership sorted on infection stage values.

Sort_Gender(): sorted_list : list<individuals>
This function returns a list of the individuals in the partnership sorted on gender values.

Sort(): sorted_list : list<individuals>
This function returns a list of the individuals in the partnership sorted on either infection stage values (in case of a homosexual population) or gender values (in case of a heterosexual population).

D Results

The figures illustrating the results of the simulation model for the baseline parameter values, that are shown in section 5.3 in tables 9 - 16.

D.1 Degree Distribution

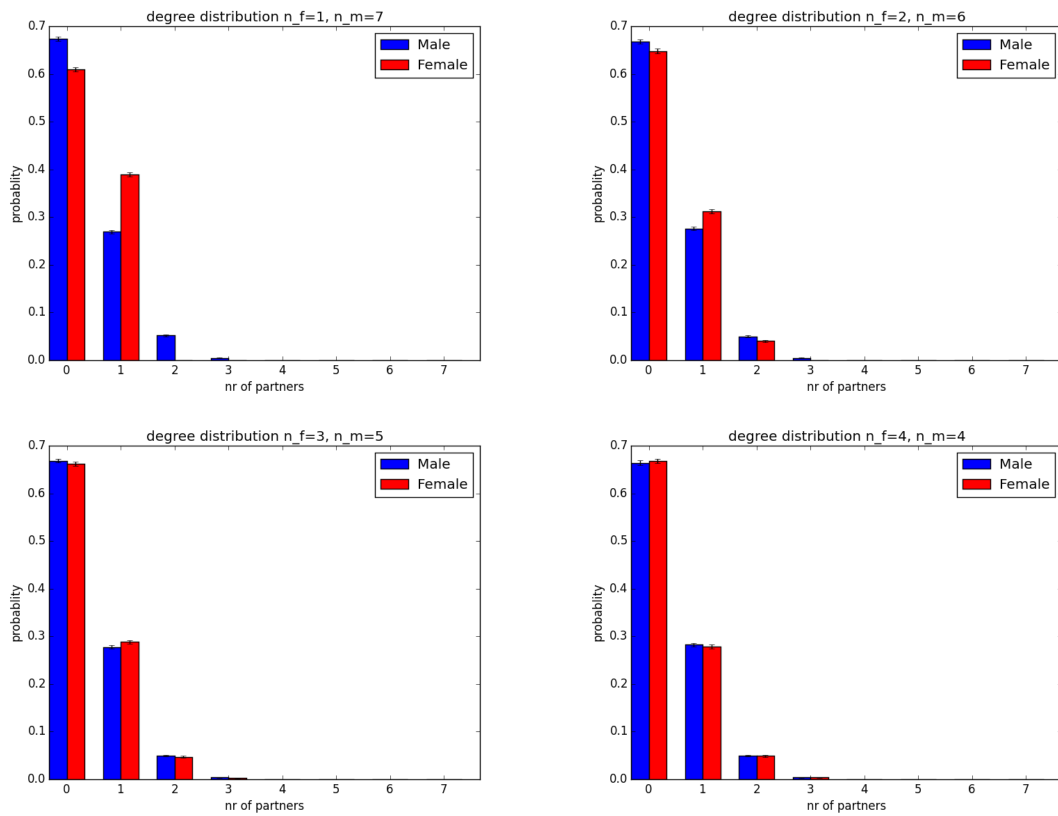


Figure 3: The degree distributions (mean and 95% confidence interval) with $N_0 = 1.000$, for $(n_f, n_m) = (1,7), (2,6), (3,5)$ and $(4,4)$. The used parameter values are the baseline parameter values in table 7.

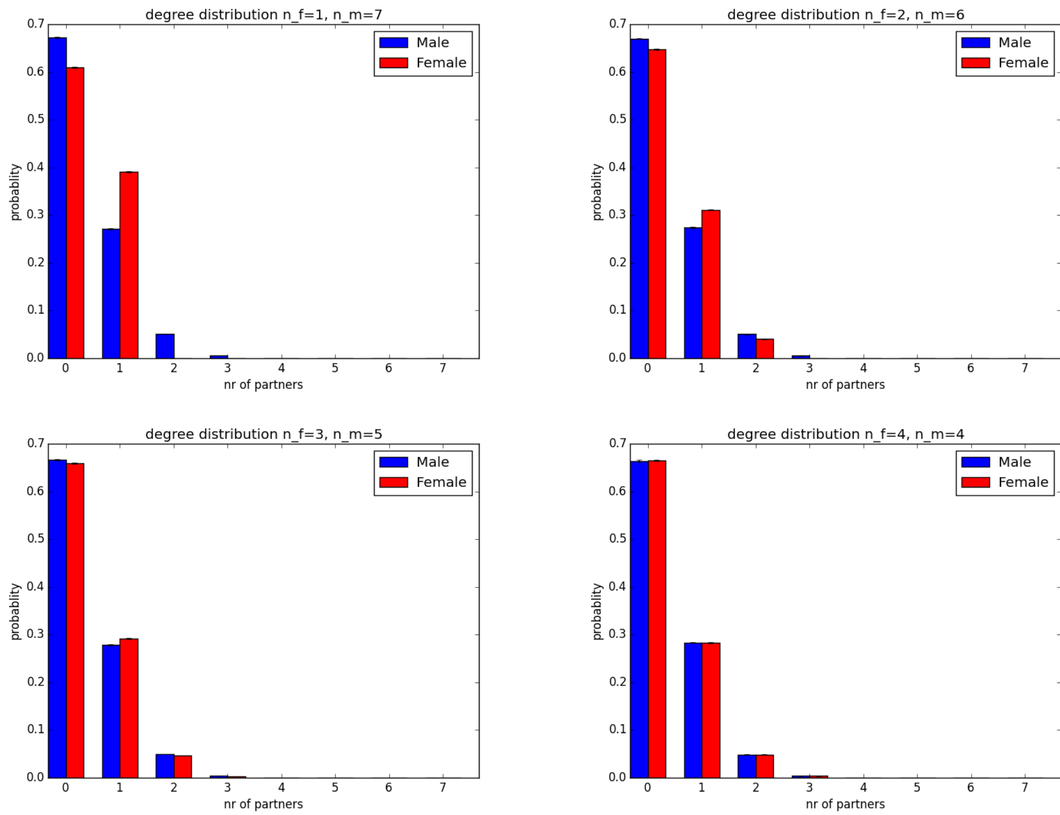


Figure 4: The degree distributions (mean and 95% confidence interval) with $N_0 = 10.000$, for $(n_f, n_m) = (1, 7), (2, 6), (3, 5)$ and $(4, 4)$. The used parameter values are the baseline parameter values in table 7.

D.2 Degree Variance and Mean

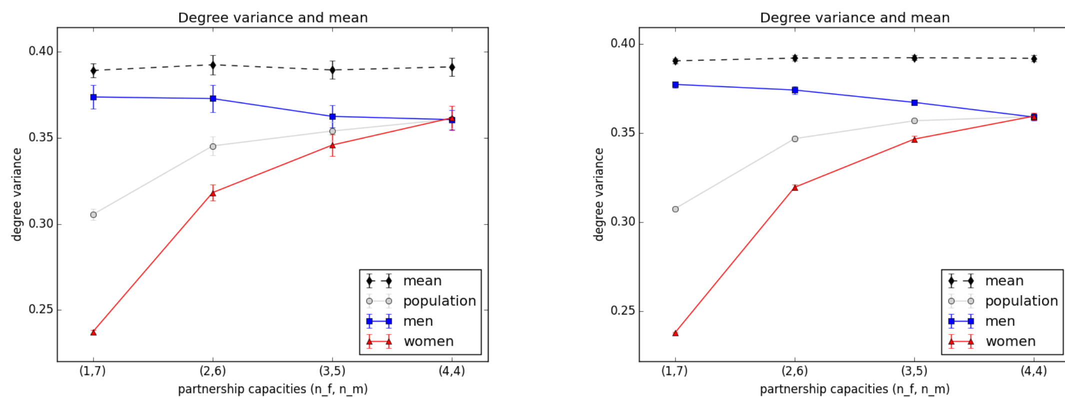


Figure 5: Degree Variance and Mean (mean and 95% confidence interval) for $(n_f, n_m) = (1,7),(2,6),(3,5)$ and $(4,4)$, with $N_0 = 1.000$ (left) and $N_0 = 10.000$ (right). The used parameter values are the baseline parameter values in table 7.

D.3 Concurrency Index

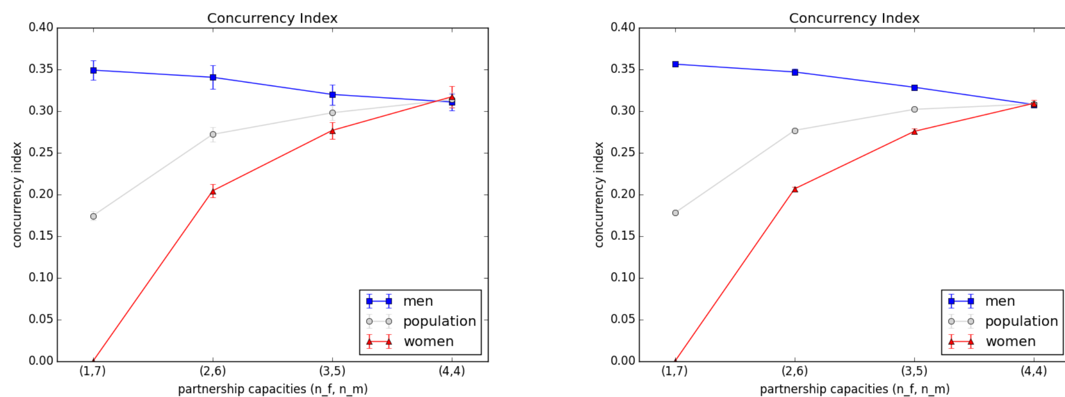


Figure 6: Concurrency index (mean and 95% confidence interval) for $(n_f, n_m) = (1,7),(2,6),(3,5)$ and $(4,4)$, with $N_0 = 1.000$ (left) and $N_0 = 10.000$ (right). The used parameter values are the baseline parameter values in table 7.

D.4 Endemic Level

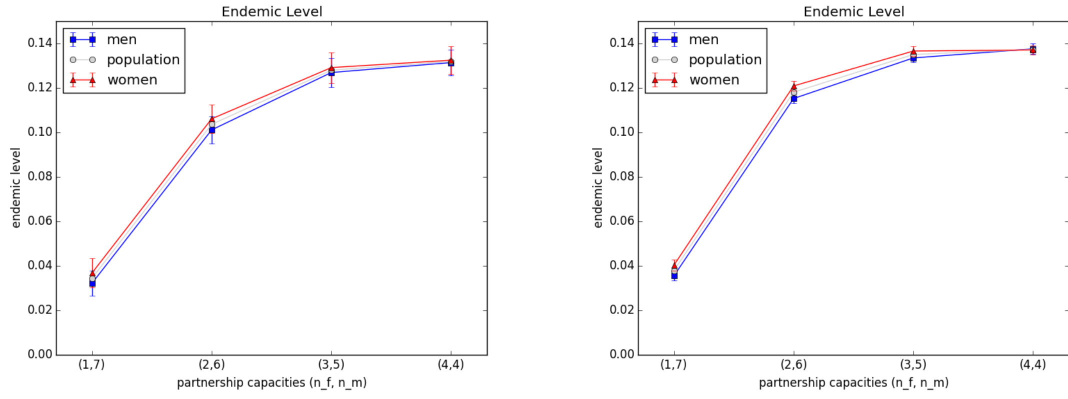


Figure 7: Mean endemic level (mean and 95% confidence interval) for $(n_f, n_m) = (1,7), (2,6), (3,5)$ and $(4,4)$, with $N_0 = 1.000$ (left) and $N_0 = 10.000$ (right). The used parameter values are the baseline parameter values in table 7.