Report for exercise 2 from group K

Tasks addressed:	5
Authors:	SONJA KRAFFT (03681252) LUDWIG-FERDINAND STUMPP (03736583) TIMUR WOHLLEBER (03742932)
Last compiled:	2023–10–27
Source code:	https://gitlab.lrz.de/00000000014A9334/mlcs-ex2-vadere

The work on tasks was divided in the following way:

SONJA KRAFFT (03681252)	Task 1	33%
	Task 2	33%
	Task 3	33%
	Task 4	33%
	Task 5	33%
LUDWIG-FERDINAND STUMPP (03736583)	Task 1	33%
	Task 2	33%
	Task 3	33%
	Task 4	33%
	Task 5	33%
TIMUR WOHLLEBER (03742932)	Task 1	33%
	Task 2	33%
	Task 3	33%
	Task 4	33%
	Task 5	33%

1 Introduction

Goal of this exercise

Through the experience from exercise one, it has become clear that the development of a simulation tool for even simple scenarios such as that of a single pedestrian with only a few obstacles can quickly become very complex. Therefore, the open source software Vadere is to be integrated and used as the basis of this exercise. Vadere enables the rapid development of complex scenarios via an extensive graphical and command line user interface. This allows a detailed inspection of performed simulations and thus enables a fast execution and evaluation of experiments. For this purpose, the spread of infectious diseases in human crowds will be simulated and evaluated in different scenarios.

Setting up the runtime and development environment

The following chapters both require a Java and a Python environment. We differentiate between:

1. Java runtime environment for running the Vadere[1] software (table 1). Mainly used in tasks 1 - 3.
2. Java development environment for building and running the Vadere software from its develpment repository[2] (table 2). Mainly used in tasks 4 - 5.
3. Python runtime & development environment for building and running the Plotly/Dash application[3] that visualizes the simulation results of an implemented SIR model in Vadere (table 3). Mainly used in tasks 4 - 5.

The three tables below list the required dependencies as well as the used code editors. Note, that these are not strict by default. For example for running the Vadere Software, any Java Runtime Environment later than Java 11 works.

OpenJDK Runtime Environment	build 17.0.1+12-39

Table 1: Java dependencies for running the Vadere software. Note that Java version 11 or above is required.

OpenJDK	17.0.1 2021-10-19
Maven	3.0
IntelliJ IDEA	2021.2.3 (Community Edition)

Table 2: Java dependencies and code editor for building and running the Vadere software. Note that Java version 11 or above is required.

Python	3.9.5
dash	2.0.0
dash_bootstrap_components	1.0.0
dash_core_components	2.0.0
dash_html_components	2.0.0
pandas	1.3.4
plotly	5.3.1
VSCode	1.62.2

Table 3: Python, libraries and code editor for Plotly/Dash application. Find the corresponding `requirements.txt` file at the provided code repository.

[1] `https://www.vadere.org/`
[2] `https://gitlab.lrz.de/vadere/vadere`
[3] `https://gitlab.lrz.de/00000000014A9334/mlcs-ex2-vadere/-/tree/main/SIRvisualization`

2 Individual Tasks

Report on task Task 1/5: Setting up the Vadere environment

Motivation

The first sub-task compares the cellular automaton (CA) developed in exercise one and Vadere as a simulation environment. For this purpose the following section evaluates the outcome of a customized version of the "Chicken Test" and the RiMEA Tests "Straight Line" and "Corner" scenario as defined in the RiMEA guidelines. As a movement model the Optimal Steps Model with its standart template is chosen which will be briefly introduced in the following section.

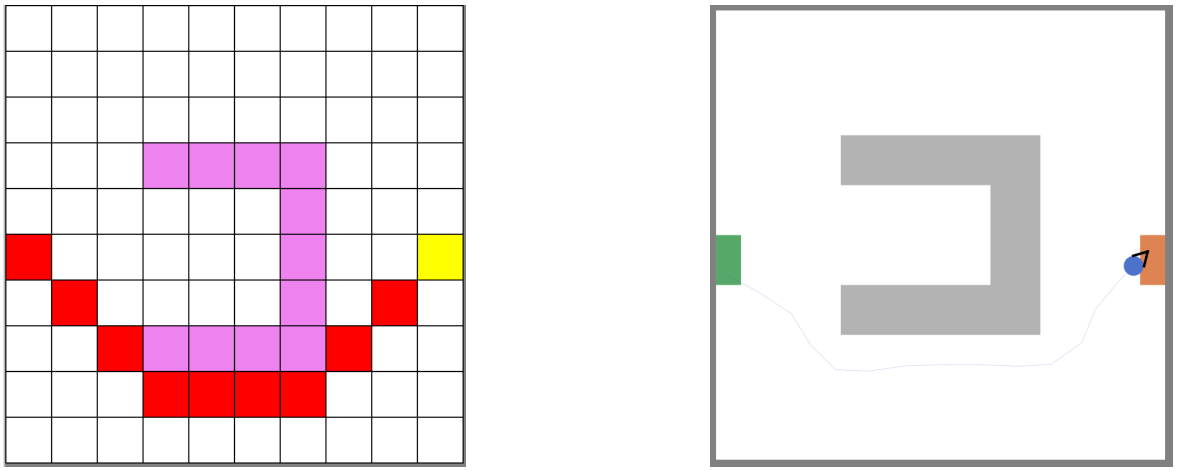
Optimal Steps Model (OSM)

The main advantage of the Optimal Step Model is the focus on imitating human movement behavior. This is achieved by recalculating the subsequent movement step for each simulation step. In this way, step length and walking speed can be continuously adapted to environmental influences such as obstacles or other pedestrians. This behavior, summarised under the term dynamic navigation field, subsequently ensures that the pedestrian decides anew for the optimal path in each time step. This results in a much more realistic pedestrian walking behavior, as it can simulate scenarios such as slowing walking speeds in crowds. Another clear difference to cellular automata is that there is no subdivision of the simulated world into discrete locations, instead OSM uses a continuous view of space. The continuous geometry of the space thus enables the agents to perform much more complex movements than being limited to axis-parallel or diagonal movements. These two factors result in a much more accurate approximation of the reality of movements in space. [4]

CA/OSM - Chicken Test:

The following section describes and discusses the differences and similarities of the simulations results for the chicken test scenario using the Cellular Automaton and Optimal Steps Model.

Please Note: The trajectory of a single pedestrian might be difficult to see due to preset settings of Vadere. Please ensure that you zoom in sufficiently on the images, they are not blank.



(a) Cellular automaton developed in exercise one.

(b) Optimal steps model in Vadere GUI.

Figure 1: Comparison "Chicken Test" implementations

[4] https://www.accu-rate.de/de/das-optimal-steps-model-2/

Observation:

The agent leaves the spawn area in the lower area of the source cell. The optimal path using the OSM emerges at the bottom around the obstacle, which is the same path as in the cellular automaton implementation. The trajectory of the pedestrian fluctuates somewhat over the entire course, even within long straights. This undulating motion can only be seen in Figure 1b due to distinctions in the accuracy of visualization. Nevertheless at any point in the simulation, the agent maintains a clear distance from the obstacle and arrives at the lower part of the target cell. The pedestrian gets therefore not stuck and passes the chicken test scenario in both simulations.

Discussion:

The scenario file in Vadere was set up as specified in the CSV file "Chicken Test" from exercise one. This means that the source cell and target cell are not exactly aligned in the centre of the 10x10 grid, but are offset slightly downwards. Therefore, it is not surprising that in both implementations, the optimal path turns out to be the path around bottom side of the obstacle.
Nevertheless, due to the continuous space of the OSM, it becomes clear that the movements of the agent is much more complex compared to the cellular automaton implementation. This can be particularly observed in slight deviations on the long straight at the bottom of the obstacle. Here, the pedestrian does not creep along the obstacle, but keeps a distance of about 1.5 times his physical dimensions at all times. In the cellular automation simulation as shown in Figure 1a cells with pedestrians were marked as completely occupied, it therefore appears as if the pedestrian is walking right along the obstacle.

CA/OSM - RiMEA Test 1 - Straight Line

(a) Cellular automaton developed in exercise one.

(b) Optimal steps model in Vadere GUI.

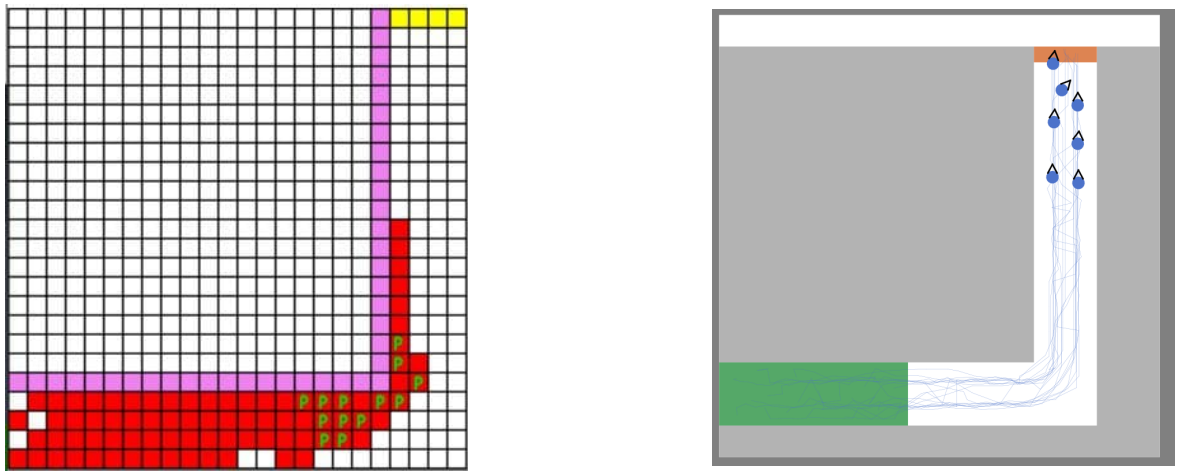
Figure 2: Comparison of the implementations of RiMEA Test 1 - straight line scenario

Observation:

The agent in the OSM as shown in Figure 2b leaves the source cell in the lower third and arrives at the target cell in the middle. Over the long straight, deviations upwards and downwards can be seen, yet the agent keeps its distance from the walls at all times.

Discussion:

The aim of this test is to prove that a predefined travel speed can be maintained over a period of time, taking into account minor inaccuracies resulting from body dimensions, reaction time and speed. According to the RiMEA Guideline the travel time should be within an interval of 26 to 34 seconds. The pedestrian in the cellular automaton needs 29 seconds for the distance which is almost equal to the 29,6 seconds needed using the OSM. This might be surprising since the trajectory of the OSM seem to be longer due to the fluctuations around around the middle of the aisle.
The experiment has shown that both implementations can complete the test with almost the same performance. As already mentioned in the previous Section 2, there are again considerable differences in the visualisation due to the difference between the continuous space in the OSM and the discretized simulation world using the CA.Nevertheless, the test indicates good aproximation of the reality even using the discrete spatial space of the cellular automaton.



(a) Cellular automaton developed in exercise one.

(b) Optimal steps model in Vadere GUI.

Figure 3: Comparison of the implementations of RiMEA test 6 - corner scenario

CA/OSM - RiMEA Test 6 - Corner

Observation:

All 20 pedestrians both implementations complete the test without colliding with obstacles or with each other. At various points in the simulation, certain pedestrians in the OSM decide to overtake the person in front of them. This can be seen for example in the movement to the right in Figure 3b where the penultimate pedestrian wanted to overtake the agent who was just arriving at the finish line. Moreover, most pedestrians in the OSM prefer a trajectory with enough space to the walls, which results in two main trajectories.

Discussion:

Comparing both implementations it becomes particularly apparent that the OSM can approximate the reality of the scenario better then the CA. Not only does it show delay caused by the congestion of pedestrians at the corner point but also does not suffer from the diagonal glitch at the corner point. Furthermore, the recalculation of the optimal path before each iteration for every pedestrian results in overtaking manoeuvres over different trajectories. In addition, the OSM results in two main routes, while the CA scenario in Figure 3a only indicates one main trajectory to the destination after the corner point.

Report on task Task 2/5: Simulation of the scenario with a different model

Motivation

This sub-task outlines the effects of different movement models on the simulation results. For this purpose the three scenarios "Chicken Test", "straight line" and "corner scenario" should be evaluated and compared using the Social Force Model (SFM), the Gradient Navigation Model (GNM) and the results of the Optimal Steps Model (OSM) conducted in the previous section. The following is a very brief summary of the SFM and GNM movement models.

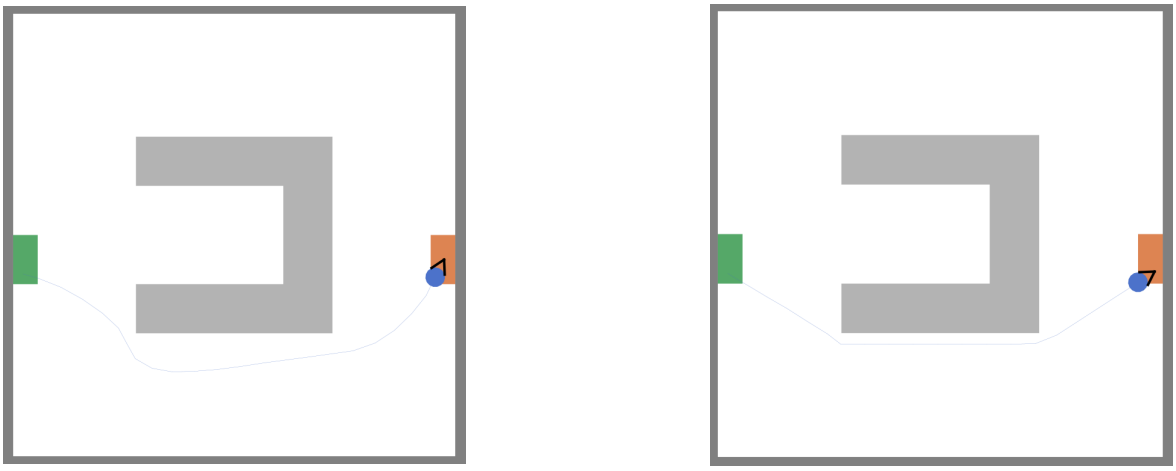
Social Force Model (SFM):

This model is a force-based model, which understands the behavior of pedestrians as the sum of different social forces. It should be noted that it is not a matter of forces physically acting on the pedestrian, but rather as the overlapping of different motivations of the individuals. These forces can be roughly divided into four types. A distinction is made between attractive forces such as the desire to reach a destination, which is usually a place or people of particular interest, and repulsive forces such as the distance to other agents and obstacles. [1]

Gradient Navigation Model (GNM):

This approach is an ordinary differential equation-based method which changes the velocity vector directly based on evaluating gradients of distance functions. The individual direction of movement of the agents is determined by a navigation vector that is a combination of floor field and local information. The advantage of this approach is essentially that it avoids the inertia of pedestrians by using only three differential equations. [2]

SFM/GNM/OSM - Chicken Test



(a) Social Force Model

(b) Gradient Navigation Model

Figure 4: Comparison implementations for the custom chicken test scenario.

Observation:

Figure 4 shows that both models, the SFM and GNM, are capable of passing the test. As in the OSM implementation in Figure 1b the optimal path is around the bottom side of the obstacle. Figure 4a shows that

the trajectory of the pedestrian has a sudden change in movement shortly before the bottom corner of the U-shape obstacle. This subsequently results in a curved trajectory away from the obstacle followed by a return to the closer proximity of the obstacle. The use of the GNM as shown in Figure 4b results rather in straight constant movements without such overshooting. This model also shows the smallest distance to the obstacle over the entire time frame. Compared to the OSM, the trajectories of the SFM and GNM appear to be much less oscillating over the entire course.

Discussion:

Notable in the evaluation of the experiment with the SFM as shown in Figure 4a is the small distance to the front lower corner and the subsequent overshooting away from it. Here, one could have expected a constant distance would be maintained at all times due to the even repulsive force of obstacles, but instead there is a time-delayed overreaction away from the corner point. This indicates that this model simulates a certain inertia in the change of movement. In contrast to this the GNM in Figure 4b shows an immediate change in direction, so that the course looks kink-like as a result of no inertia.
Furthermore, the use of the GNM shows that the goal was reached with the lowest number of iterations ($t_{SimStep} = 20$), which is plausible since straights should result in the shortest overall path. In contrast, the OSM needed the most iterations for this with ($t_{SimStep} = 23$). This is probably due to the slightly oscillating trajectory and the resulting extension of the walking distance.

SFM/GNM/OSM - RiMEA Test 1 - Straight Line

(a) Social Force Model (b) Gradient Navigation Model

Figure 5: Comparison implementations for the RiMEA Test 1 - straight line scenario.

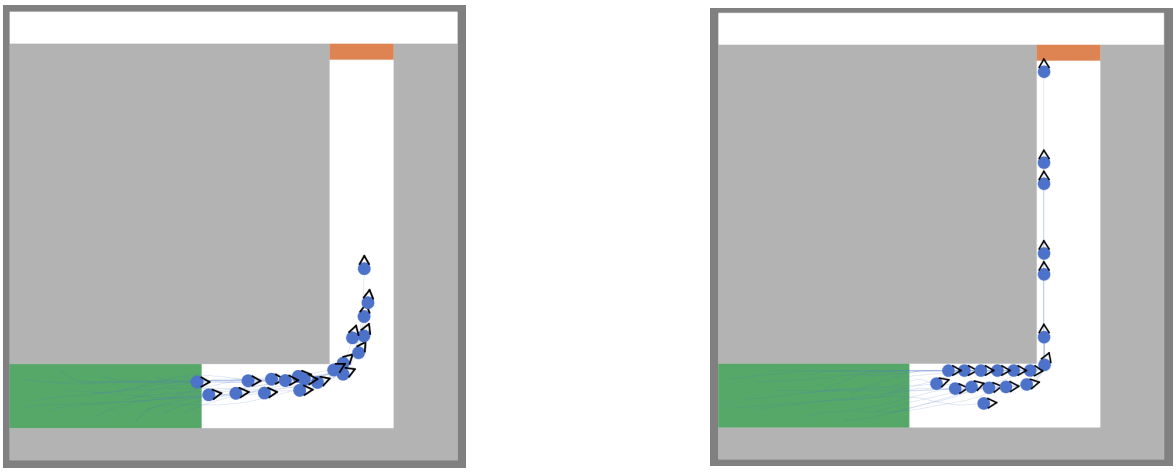
Observation:

The SFM in Figure 5a shows the trajectory of a pedestrian aligning himself centrally to the two walls within the first seven iterations. After that the agent moves with minor oscillating movements in the middle of the corridor until the target is reached in 28,4 seconds. In almost the same time (28,8 seconds) also the agent from the GNM scenario displayed in Figure 5b arrives at the target cell. In contrast to the pedestrian from the SFM fully centred to the aisle, however, the trajectory of the pedestrian is a straight line in the lower third of the corridor that remains constant over the entire time course.

Discussion:

As described in more detail in Section 2 the aim of this task is to analyze the walking behavior through a long corridor. For all three movement models the travel time is within the expected interval of 26 to 34 seconds. It is somewhat surprising that not the GNM with 72 simulation steps (=28.8 seconds) is the fastest model in this experiment, but instead the SFM with 71 iterations (=28.4 seconds). Here, one would assume that it should be the fastest model, since it connects source and target cell via a simple straight line. Since the deviation is so small, this is probably due to movement inaccuracies but should be negligible. In addition to that it is not suprising that the OSM requires the longest travel time for the required distance due to the oscillating trajectory.
Salient is that the GNM is the only model that does not try to align itself in the middle of the corridor in the initial iterations. A reason for this could be presumed in the initial spawn position of the pedestrian and the shortest way to the destination from there. In addition, with the SFM, one could assume that there is an equilibrium of the repulsive forces of the walls in the middle of the corridor, what could be the reason for the initial alignment movement.

SFM/GNM/OSM- RiMEA Test 6 - Corner



(a) Social Force Model

(b) Gradient Navigation Model

Figure 6: Comparison implementations for the RiMEA Test 6 - corner scenario.

Observation:

All three movement models are able to simulate 20 pedestrians moving around a corner without crossing the wall. The initially randomly placed pedestrians in the SFM shown in Figure 6a pass the corner with multiple agents at the the time and move into a main centred trajectory after the corner point. A different path emerges with the GNM as illustrated in Figure 6b where the pedestrians move in two rows towards the corner point, but there a kind of zip procedure takes place so that only one pedestrian goes around the corner at a time. Therefore a single trajectory close to the wall emerges after the corner point. Since there are no more congestion situations here, the distances between the agents increase after the corner point.

Discussion:

The aim of this test is to prove that the agents can manoeuvre around a corner without crossing it. The three models met this challenge in two different ways. The first approach was to traverse the corner in rows of two or more agents at a time. This approach has been shown to work for both the OSM and the SFM. The initial assumption that the SFM would result in the largest distances turned out to be wrong. Surprisingly, the distances were even smaller than with the OSM as shown in Figure 3b.
Furthermore, no real stop and go behaviour could be observed with the SFM, like in the OSM implementation. Another difference between the two models is the course of the main trajectory (=overlay of the trajectories taken by most pedestrians). Whereas with the OSM there were essentially two main trajectories parallel to each other, with the SFM there was a single broader trajectory. This is likely due to the fact that in this model, the agents are all equally repelled by the walls and, thus, the axis in the middle of the aisle turns out to be the optimal path.
The second approach shown in the GNM implementation in Figure 6b is surprising, as it would be a non-intuitive way for us humans to walk around the corner. This results in only one main trajectory with a small distance to the wall after the corner point. This unnatural, but nevertheless successful, behaviour is probably due to the fact that it is the shortest distance to the target cell after the corner point.

Report on task Task 3/5: Using the console interface from Vadere

Motivation

The main purpose of this task is to integrate and test Vadere via the command line interface. Integrating Vadere into your own development environment has the great advantage that you can use the already extensive tool as a black box and thus have access to a variety of different motion models without having to implement them yourself. Furthermore, it opens up the possibility of visualising simulation results in a proper way.

In order to build a first prototype that can make adjustments to a scenario from a vadere external development environment, a single pedestrian is to be placed in the corner of the RiMEA Test 6 corner scenario. The subsequent section describes the implemented modification pipeline in python.

Modification Pipeline

The flowchart in Figure 7 illustrates the workflow necessary to read in a scenario and pedestrian source file, combine them and run the simulation. As a first step it was necessary to understand the structure used by Vadere to store scenarios: Vadere creates for every scenario a nested JSON containing all information necessary to create and run a scenario. Such a `Scenario JSON` can be seen for example in Figure 8a. In order to place a pedestrian into the scenario it was necessary to understand where in the scenario file the information describing the pedestrian had to be stored. For this purpose, the structure was analysed in more detail with the help of an online JSON formatter. This offers the great advantage that nestings can be specifically hidden and, thus, a clear detailed analysis of the individual sub-areas can take place. The red highlighted areas show the hierarchy `Scenario > Topography > Dynamic Elements` from the top layer to the `dynamicsElements` list where the pedestrian information should be included.

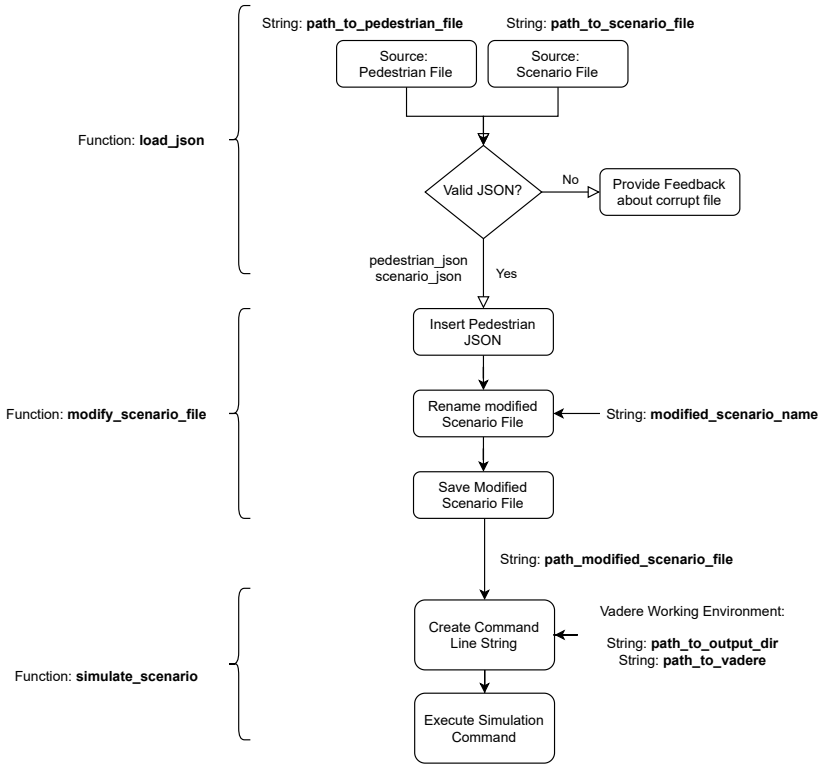


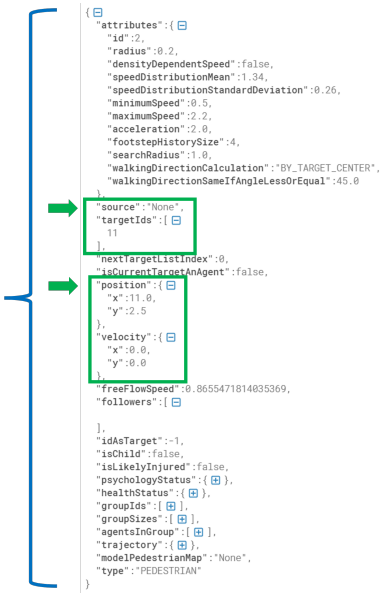
Figure 7: Flow chart to modify a existing scenario file by code

After the localization within the `Scenario JSON` it was necessary to understand the structure of a `Pedestrian JSON`, which contains all information necessary to describe a pedestrian. For this purpose, a pedestrian was placed at the desired position in the corner scenario and the `targetID` of the target was stored in the list field `targetIds`. This way the pedestrian gets a target assigned which has to reached. Such a `Pedestrian JSON` is illustrated in Figure 8b.

The next step was to read in and modify the unmodified corner scenario from task one. For this purpose, Python offers its own library developed for JSONs with which it is possible to read in a JSON file and store it temporarily as a Python object(dict). Since no further information are given about which attributes should by modified by the use of code the green highlighted areas could be possible examples. In order to maintain modular and clear structure the `pedestrian JSON` is stored and read in from an external `.json` file.

Then the modified scenario is saved in the `scenarios` folder under a different name. The last step was to generate the console command that starts the `vadere-console.jar` and execute it in the corresponding workspace.

(a) Shows the scenario file for the unmodified corner scenario from RiMEA Test 6.



(b) Shows the pedestrian information JSON which is inserted into the "dynamicsElements" list of the scenario JSON

Figure 8: Highlighted by red markings is the branch of the nested scenario JSON that contains the pedestrian information. Other branches are collapsed for display purposes.

Comparison Simulation GUI / CLI

By modifying the scenario, which in this case was the addition of a pedestrian in the corner area, no difference arises whether the simulation was started through the graphic interface or the console. The reason for this may be that the flag `useFixedSeed` of the scenario JSON is set to true in the branch `attributesSimulation`. This means that the same seed is used when the scenario is simulated again.

Comparison Target Attainment

It can be observed that the added pedestrian reaches the target cell in the 34th iteration. It is notable that some of the 20 randomly created pedestrians in the source area reach the target even before the added pedestrian. This is probably due to the fact that the initial velocity in the `DynamicElements` list is set to zero, so a certain inertia is to be expected until the walking speed is reached.

Report on task 4/5: Integrating a new model

Motivation

In the previous chapters, the main functionalities of the Vadere software have been analyzed and compared to our own *Cellular Automaton*. In this chapter, we want to integrate a new model, namely the SIR model, which is used to analyze the spread of a disease within a population.

Integrating the SIR model

For this, we are provided with a first implementation of the SIR model which for now only needs to be placed at the right package locations in the Vadere codebase. Figure 9 shows the UML diagram of the main components of the SIR model highlighted in green. The orange parts refer to later implementations, such as the decoupling of the infection rate and the simulation step length as discussed in this chapter, or the implementation of a *recovered* state in the next chapter.

The `AttributesSIRG` class specifies the parameters of the SIR simulation, namely the number of infected pedestrians at the beginning (`infectionsAtStart`), the infection rate (`infectionRate`) and a hard distance boundary to decide if pedestrians are close enough to potentially become infected (`infectionMaxDistance`). In order to keep track of the state of each pedestrian, the enumeration `SIRType` is added and differentiates between:

1. *susceptible*: Person is able to receive the disease.
2. *infected*: Person is infected and can transmit the disease to susceptible people.
3. *removed*: Person has had the disease and is dead, or is isolated until recovery.

An implementation of people who have recovered from the disease and can no longer be infected will be introduced in chapter 5. For this, an additional entry `ID_RECOVERED` is going to be added to the `SIRType` enumeration.

The `SIRGroupModel` class combines all functionalities of the SIR model and is responsible for assigning and updating the infection status within the population of the simulated scenario. The `topography` attribute of the likewise class name allows access to the dynamic pedestrian elements of the simulation. Most importantly, the `update` method, which implements the corresponding method of the `Model` interface, is called in between every simulated iteration and propagates a potential spread of the disease.

The `SIRGroupModel` class also makes use of the `SIRGroup` class, which implements the `Group` interface that is responsible for providing methods to interact with the group.

To be able to integrate the SIR model into the already existing Vadere project, one first needs to checkout the latest source code from the official Vadere repository[5]. When this has been done for this project, the commit id `59299e2c` of the `master` - branch has been the latest version, so we choose to implement our changes based on this version. For local development, the latest installation of the open-source Java development kit (*OpenJDK*, version 17.0.1 2021-10-19) has been downloaded and installed[6]. Building the project and the external dependencies, requires the Java package manager Maven to be installed[7]. Note that the Vadere software requires Maven version 3.0, as specified in their project Readme.

In order to use the implemented SIR model inside the Vadere simulation, one needs to specify it as a submodel inside the *Model* tab of the scenario customizer inside the Vadere GUI. Figure 10 highlights the required entries shown at the example of a `OptimalStepsModel`, which has been the selected locomotion model for the upcoming scenarios. One can see that the SIR model is added as a sub-model and that the values for parameters are specified.

[5] `https://gitlab.lrz.de/vadere/vadere`
[6] `https://jdk.java.net/17/`
[7] `https://maven.apache.org/download.cgi`

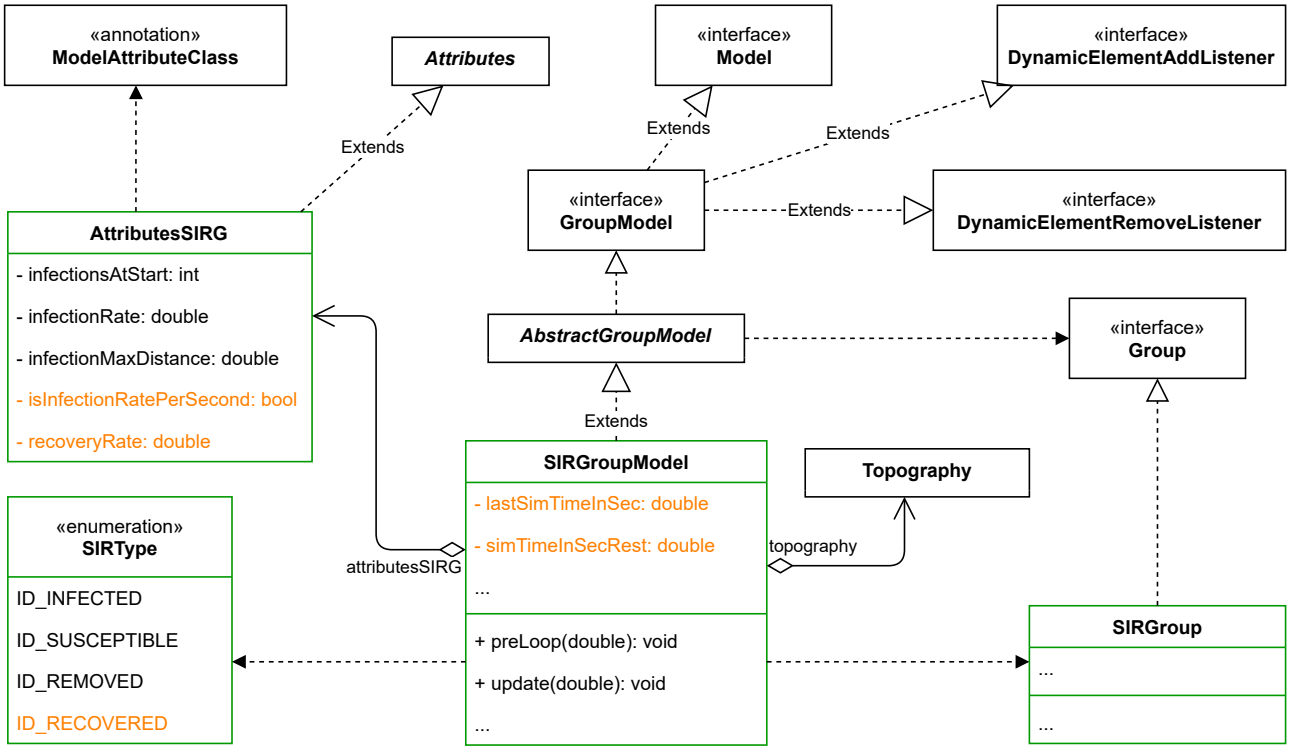


Figure 9: UML diagram of the implementation of the SIR model. Added parts are shown in green. Orange parts refer to later implementations such as the decoupling of infection rate and simulation step length or the implementation of an additional *recovered* state.

Extracting results of the simulation

To be able to analyze the simulation afterwards, we have to implement a way to extract the data about the infected pedestrians. For this reason, a custom output processor, namely the `FootStepGroupIDProcessor` has been implemented inside the `org.vadere.simulator.projects.dataprocessing.processor` package of the Vadere project. Figure 11 shows an UML diagram of the main functionalities.

The `FootStepGroupIDProcessor` extends the superclass `DataProcessor` and implements the interface `ModelFilter`. The `doUpdate()` method is called after every simulation step with the current `SimulationState` and allows to extract useful information about the simulation. For our SIR model, the assigned group id is of importance, since this number indicates if a pedestrian is susceptible, infected and later also recovered. The assigned id numbers match the implementation of the `SIRType` enumeration. During the call of `doUpdate()`, we first find and access the `SIRGroupModel`, then we iterate through the `SIRType` groups to then finally report information about each pedestrian. Here, we save the pedestrian id, the current simulation time as well as the group id which corresponds to the infection status.

Note that the current implementation of the `FootStepGroupIDProcessor` rounds the simulation time. However, the post visualization of Vadere does not work with rounded times, which is why the current version cannot visualize the infection groups after the simulation has been completed. To be still able to visualize the infection process of the simulation, one can turn on the coloring settings while the simulation is running. While a further iteration of the `FootStepGroupIDProcessor` might improve on this, the current status has been enough for the here described tasks, which is why no further adjustments have been taken at this point.

In order to write the extracted data from the `FootStepGroupIDProcessor` to a CSV file after the simulation has been finished, we need to add a new file at the `data output` tab of the simulation GUI in Vadere and link it to the just created `FootStepGroupIDProcessor`. Here, we set the `Data Key` to `EventtimePedestrianIdKey` as shown in figure 12. This allows us to analyze the infection process after the simulation has been finished by inspecting the content of the CSV file using a provided Plotly/Dash Python tool.

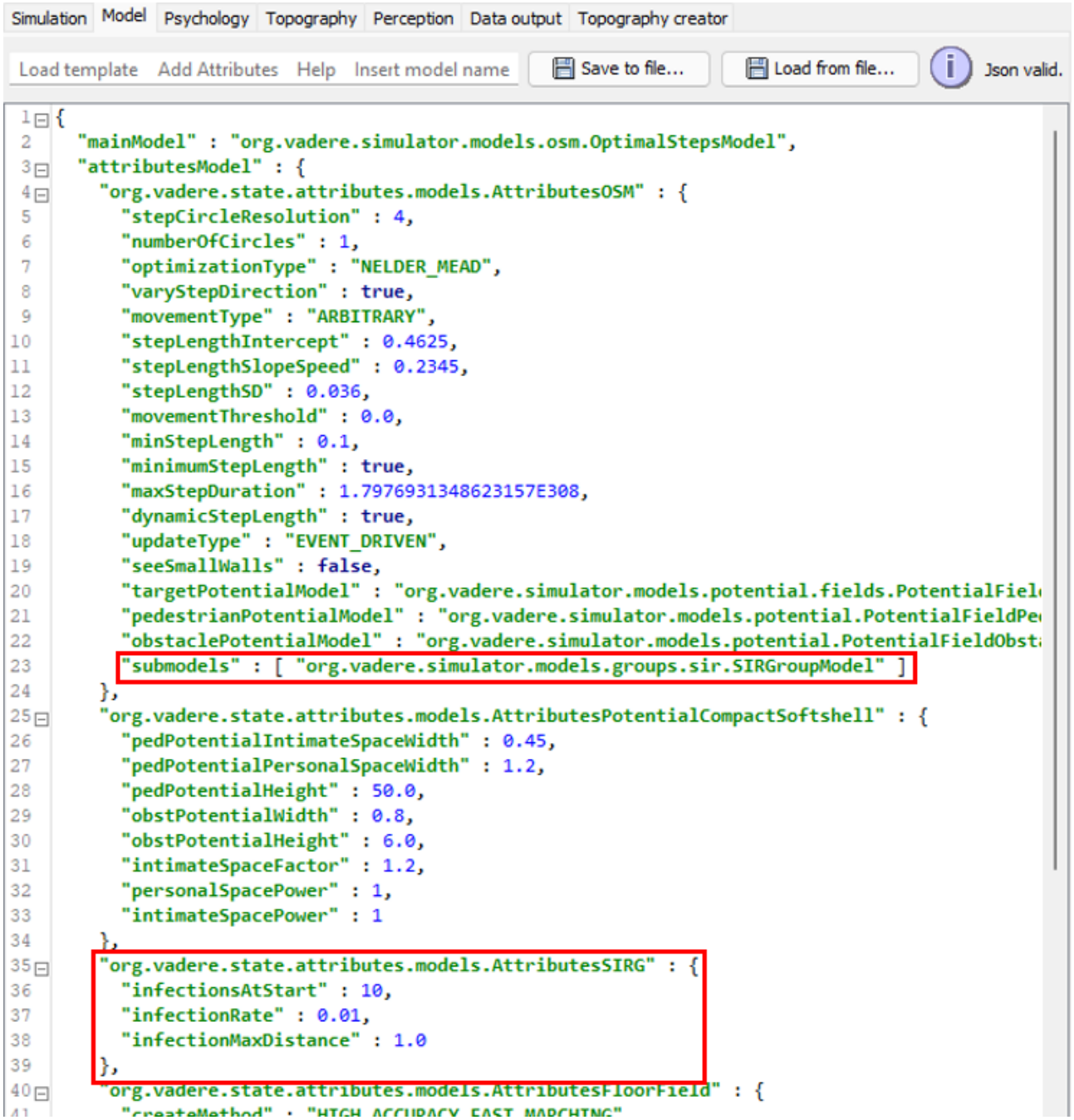


Figure 10: Screenshot of the Vadere GUI. Adding the SIR model as a submodel of the `OptimalStepsModel` and specifying the attributes.

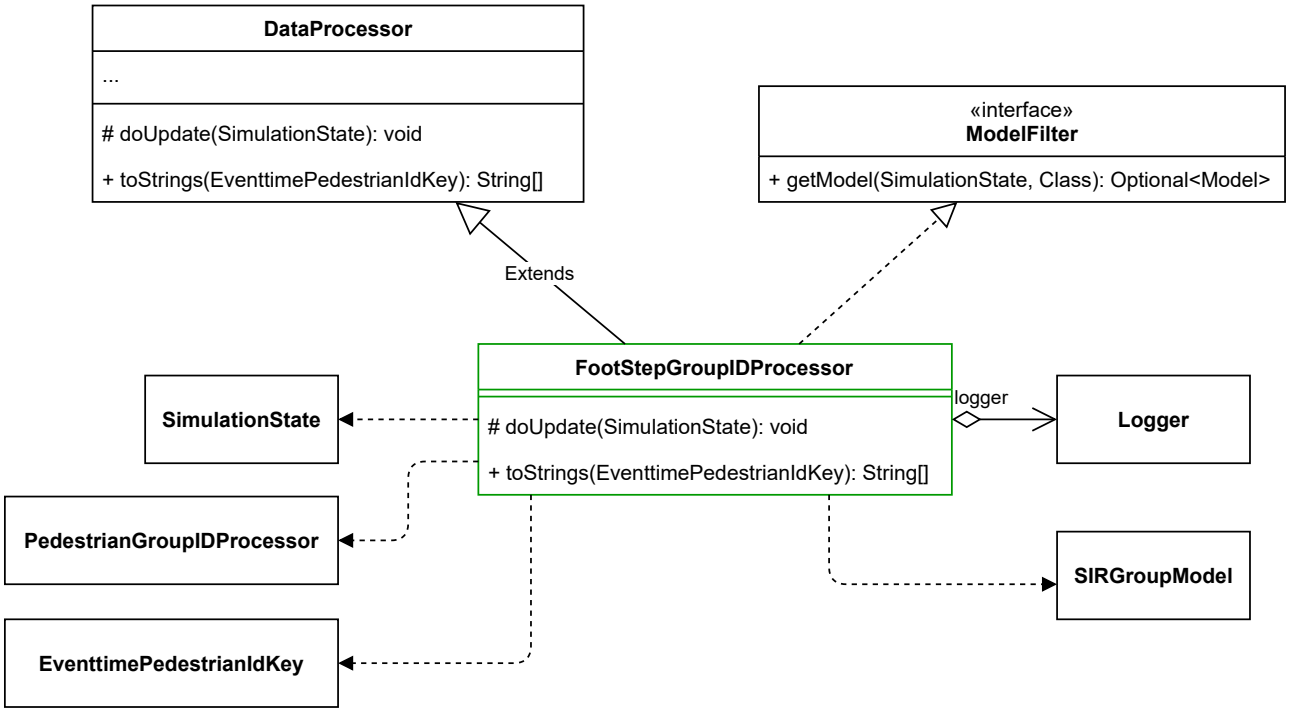


Figure 11: UML diagram of main parts of `FootStepGroupIDProcessor`. Added parts are shown in green.

Improve efficiency of distance computation for neighbors

The method `SIRGroupModel.update()` is responsible for simulating the spread of the disease during each simulation step. It makes use of the attributes `infectionRate` and `infectionMaxDistance` to check, which yet uninfected pedestrians will become infected in the next time step. For this purpose, it is necessary to find pedestrians which are within a fixed maximum distance to another already infected pedestrian. There are multiple ways to implement this check, with varying time complexity each.

Current implementation

The current implementation of the distance computation for neighbors is not very efficient. For all infected pedestrians, we check all other pedestrians and then explicitly compute the distance between each pair of infected and uninfected pedestrians to then again reduce the number of neighbors to only meet the maximum distance value. Due to this nested loop structure, the time complexity of the function is $O(n^2)$ with n being the number of simulated pedestrians. Algorithm 1 shows the pseudo-code for the current implementation.

Algorithm 1 Finding pedestrians within a fixed radius of another pedestrian - $O(n^2)$

Require: $maxDistance \geq 0$, $pedestrians$
  for each $pedestrianA \in pedestrians$ **do**
    for each $pedestrianB \in pedestrians$ **do**
      if $pedestrianA \neq pedestrianB \wedge distance(pedestrainA, pedestrianB) < maxDistance$ **then**
        SIMULATEINFECTION($pedestrianA$, $pedestrianB$)
      end if
    end for
  end for

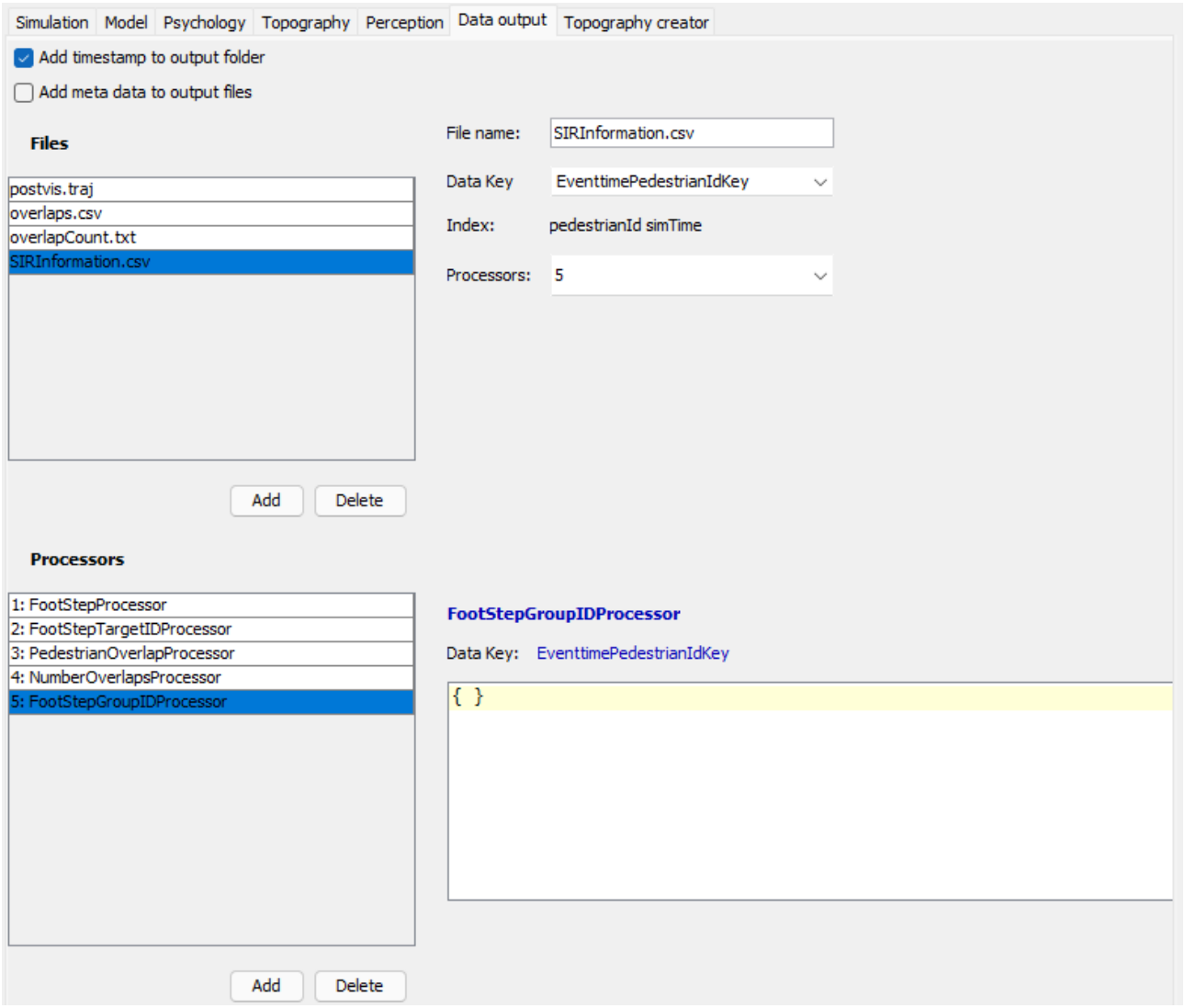


Figure 12: Writing the data of the `FootStepGroupIDProcessor` to a file `SIRInformation.csv`.

Improved implementation using `LinkedCellsGrid`

The package `org.vadere.util.geometry` includes a class `LinkedCellsGrid` that models a grid augmenting the standard position properties of a generic object for faster access. This class implements a method `getObjects()` that takes a `VPoint` object and a radius to return a list of objects on the instantiated `LinkedCellsGrid` that are within the specified radius. This method is implemented in a way that achieves a complexity of $O(1)$ for the fixed radius check, which improves our overall complexity of the `update` method to $O(n)$. The improvement comes from no longer iterating over all n pedestrians but only over the neighboring pedestrians (which are constant in n for a fixed radius). Algorithm 2 shows the pseudo-code for the improved implementation. The interested reader can take a look at the actual implementation in the method `SIRGroupModel.update()` of the package `org.vadere.simulator.models.groups.sir` to see how it has been implemented.

To verify the correct implementation, a static infection scenario like the one described in figure 14 has been simulated before and after the simulation and the resulting infection graphs have been compared. There was no difference to be detected.

Algorithm 2 Finding pedestrians within a fixed radius of another pedestrian - $O(n)$ version

Require: $maxDistance \geq 0$, $pedestrians$
  $pedestrianCells \leftarrow LinkedCellsGrid(pedestrians)$
  for each $pedestrianA \in pedestrians$ **do**
    $neighboringPedestrians \leftarrow pedestrianCells.$getObjects($pedestrianA.pos$, $maxDistance$)
    for each $pedestrianB \in neighboringPedestrians$ **do**
      if $pedestrianA \neq pedestrianB$ **then**
        simulateInfection($pedestrianA$, $pedestrianB$)
      end if
    end for
  end for

Infection test scenarios

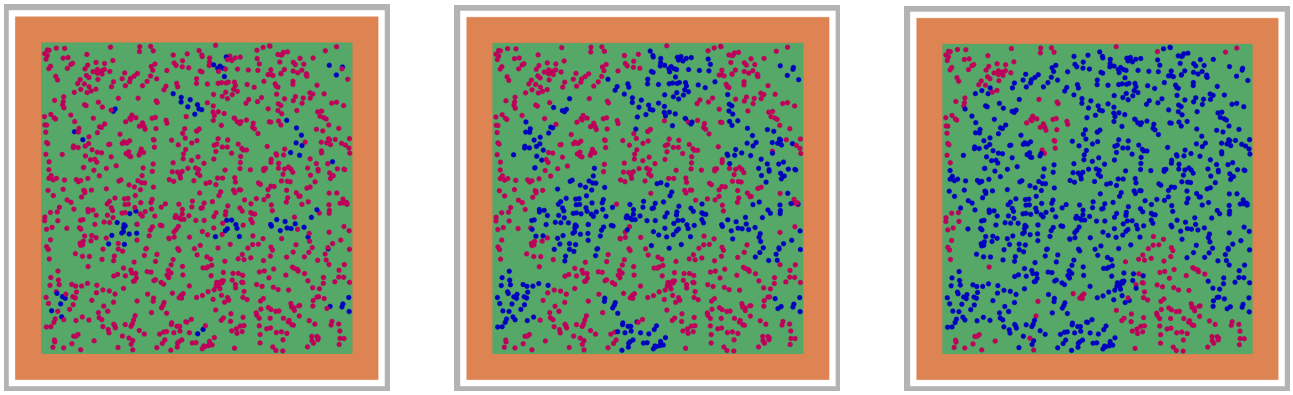
In order to validate the functionality of the implemented SIR model, it is tested with two simple scenarios. After the simulation has finished, a Plotly/Dash Python tool is used to display the resulting infection curves over time.

Infection test scenario #1 - static test

The first test scenario is a 30.00 m x 30.00 m squared room with 1000 pedestrians, randomly distributed in the scenario. Note that the pedestrians are not moving and the target is set not to absorb the pedestrians.

Figure 13 shows the spread of the infection with increasing simulation time. One can see that the amount of infected pedestrians is increasing over time. Using the python plotting tool, one can then visualize the number of infected and susceptible pedestrians over the simulation time as shown in figure 14. Note that the sum of infected and susceptible pedestrians is always equal to the total population since we have not yet implemented a *recovered* state. Looking at the intersection of both lines, one can then see the point of time when half of the population is infected. For the simulated scenario with a infection rate of 0.01, this point is reached after 94.1 s. In addition, one can observe that the change in infected pedestrians is first increasing and then decreasing again.

One interesting aspect of SIR models is to analyze the influence of the infection rate on the growth of the infection. For this reason, the same static experiment is run with a infection rate of 0.02, which is twice as high as the previous value. The comparison is shown in figure 15. One can see that the higher infection rate of 0.02 leads to a stronger increase in the number of infected pedestrians. Here, half of the population is infected after 48.8 s. This is 45.3 s earlier than with an infection rate of 0.01.



(a) Iteration 21 / 200 s. (b) Iteration 77 / 200 s. (c) Iteration 183 / 200 s.

Figure 13: Static infection test scenario. 1000 pedestrians are randomly distributed in a 30 m x 30 m squared room. There are 10 initially infected pedestrians, the infection rate is 0.01 and there is a maximum infection distance of 1.0 m. The scenario has been simulated for 200 s. Infected pedestrians are colored in blue, susceptible pedestrians in red.

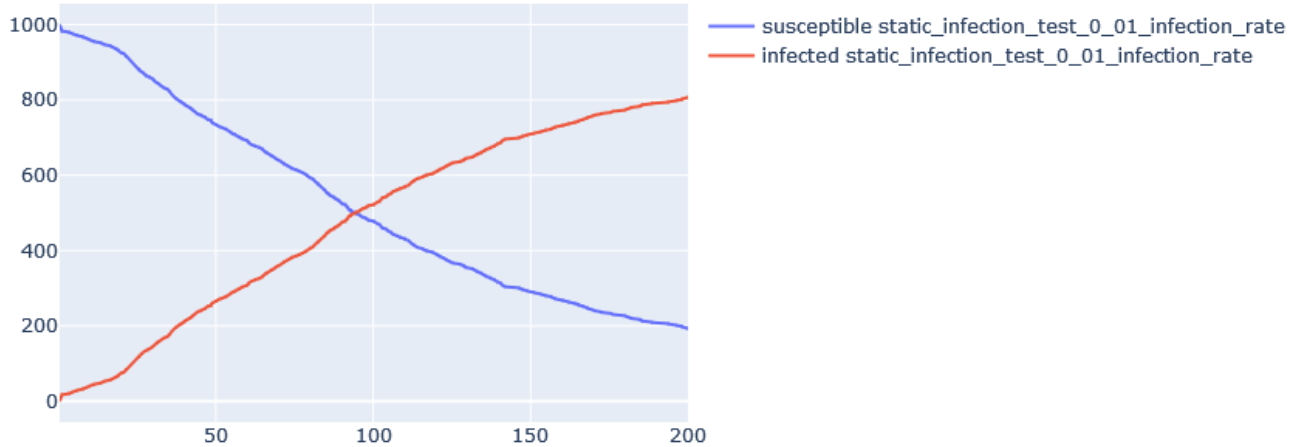


Figure 14: Visualization of the static infection scenario of 1000 randomly distributed pedestrians in a 30.00 m x 30.00 m squared room. 10 pedestrians are infected at the very beginning, the infection rate is 0.01 and the maximum infection distance is set to 1.0 m. The x-axis of the plot shows the iteration time in seconds while the y-axis shows the total numbers of pedestrians belonging to each group.

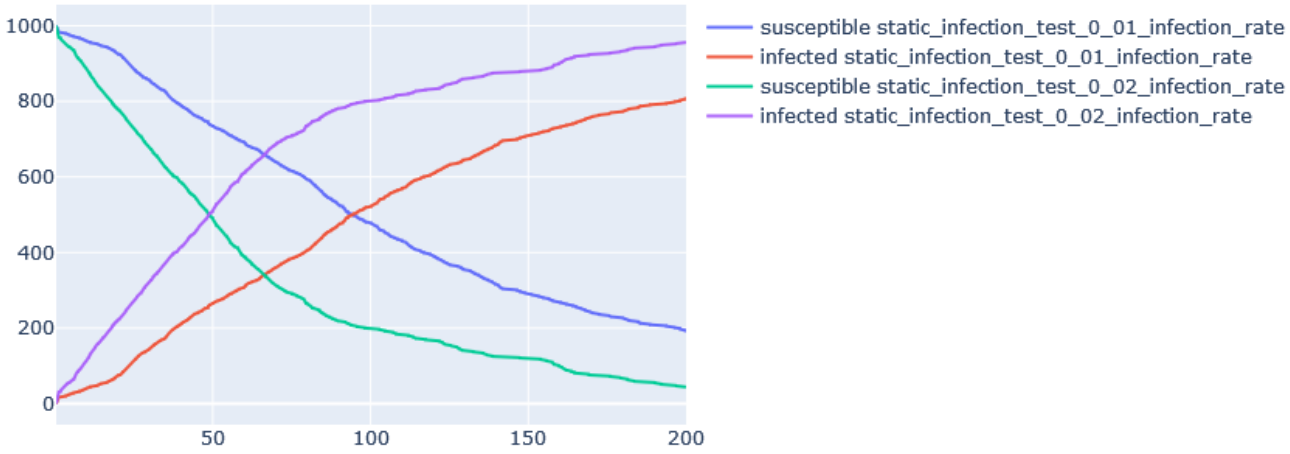


Figure 15: Visualization of the static infection scenario of 1000 randomly distributed pedestrians in a 30.00 m x 30.00 m squared room. 10 pedestrians are infected at the very beginning, the maximum infection distance is set to 1.0 m. Shown are two simulated scenarios with the different infection rates of 0.01 and 0.02. The x-axis of the plot shows the iteration time in seconds while the y-axis shows the total numbers of pedestrians belonging to each group.

Infection test scenario #2 - corridor test

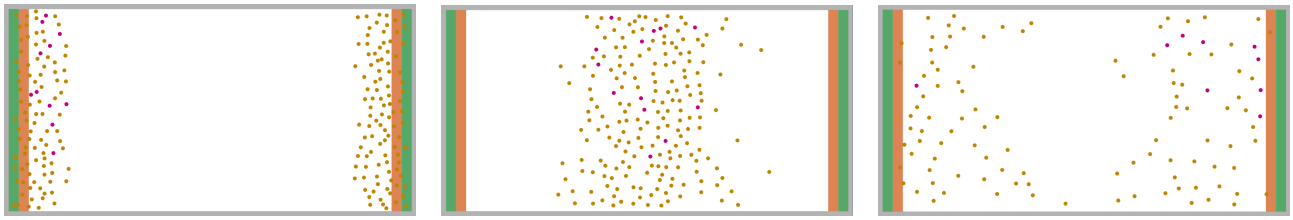
After the first test with static pedestrians has been simulated and the results analyzed, a dynamic scenario is tested next. It consists of a corridor of 40 m x 20 m, where one group of 100 pedestrians moves from left to right, and another 100 from right to left. In order to not spawn all the pedestrians at the same time, the simulation parameter `useFreeSpaceOnly` is set to `true`. The pedestrians are automatically created from the source ground so we do not need to specify the position of each pedestrian individually. After the simulation, the infection graph is visualized using the Plotly/Dash tool and the total number of infected pedestrians in this counter-flow scenario analyzed. Figure 16 shows the simulation at three different time steps.

Note that unfortunately, the balance of infected pedestrians on the left and the right is not equal. However using the current implementation of the SIR model, one cannot precisely define, which pedestrians are going to be infected at the very beginning. What one could do is to use the python tool presented in the last chapter to manually assign group ids corresponding to the infection status. However, it would be more comfortable to be able to specify this from within the GUI. The subsection "Possible extensions of the current SIR model" lists this point and a few others as an idea for future improvement of the implemented SIR model in Vadere.

After the simulation has been completed, one can take a look at the generated infection graph to determine the total number of infected pedestrians (see Figure 17). Additionally taking a look at figure 16 we can see that at the very beginning only the pedestrians on the left are infected. Although the initial number of infection is set to 10, we basically start with 11 infected pedestrians since one is infected during the creation process. Figure 16c shows that only one pedestrian from the original right side has become infected in this counter-flow scenario. All other five infections have happened on the left side only. This sums up to a total of 16 infected pedestrians at the end of the simulation process.

Decouple infection rate and time step

In the original implementation of the SIR model, the infection rate (`infectionRate`) and, therefore, also the simulation results depend on the step size of the simulation (`simTimeStepLength`). This is because during every call of the `SIRGroupModel.update()` method, this infection rate is used to check for a spread of the infection from one infected pedestrian to another susceptible one. Since this method is called during every single simulation step, the observed growth with respect to time depends on how many simulated steps are happening during a fixed period of time. This dependency is visualized in figure 18a. The observed coupling is not very convenient for reasons of comparison to other simulation settings since one would need to both change the infection rate as well as the step size of the simulation. In the following, we are presenting and evaluating



(a) Iteration 3 / 100 s. (b) Iteration 16 / 100 s. (c) Iteration 33 / 100 s.

Figure 16: Corridor infection test scenario. Counter-flow in a 40 m x 20 m corridor with 100 pedestrians from each side. 10 initially infected pedestrians, infection rate of 0.01 and a maximum infection distance of 1.0 m. Simulated for 100 s. Infected pedestrians are colored in red, susceptible pedestrians in yellow.

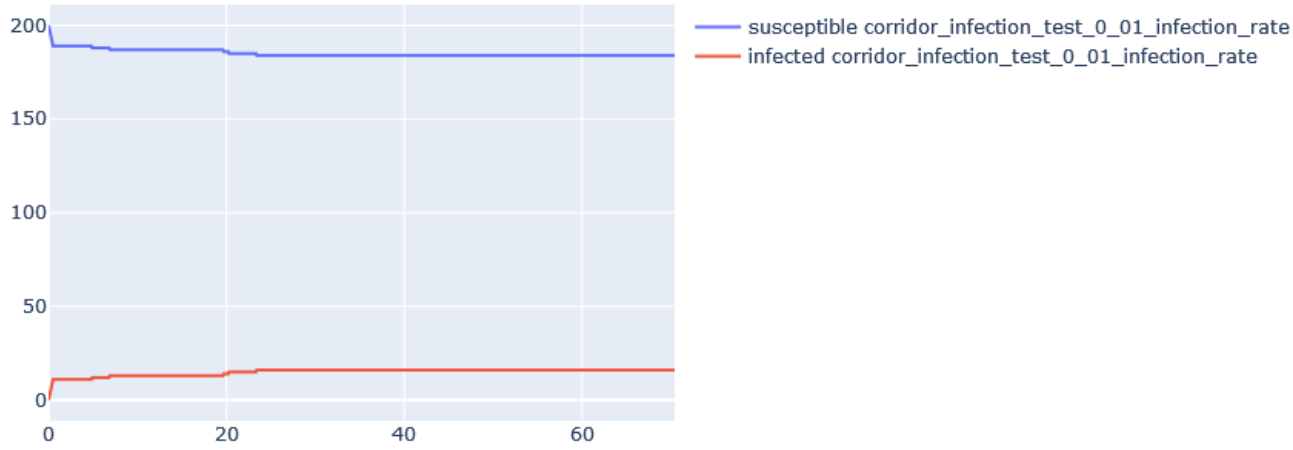


Figure 17: Visualization of the corridor infection scenario of 100 pedestrians on each side of a 20.00 m x 40.00 m corridor. 10 pedestrians are infected at the very beginning, the maximum infection distance is set to 1.0 m and the infection rate is 0.01. After the simulation, a total of 16 pedestrians are infected. The x-axis of the plot shows the iteration time in seconds while the y-axis shows the total numbers of pedestrians belonging to each group.

infectionRatePerSecond [1/s]	simTimeStepLength [s/it]	infectionRatePerTimeStep [1/it]
0.025	0.4	0.01
0.025	0.8	0.02

Table 4: Two settings of the same infection rate with different time steps length. One can see that the difference in time step length is corrected by choosing a infection rate per time step accordingly.

two possible solutions to this coupling problem.

Simulation step based decoupling

A first and simple approach[8] to solve the coupling issue is to:

1. Treat the infection rate as a rate per second.
2. Multiply the infection rate with the step size of unit "seconds per simulation step" to get the corresponding infection rate per simulation step.
3. Use the resulting infection rate per simulation step as before in the `SIRGroupModel.update()` method.

Apart from the described unit conversion, one could leave the `SIRGroupModel.update()` method as it is originally implemented which is why this approach is likely to be the simplest one. Table 4 shows two settings with different time step lengths. Looking at the results of this decoupling approach as visualized in figure 18b one can see a great improvement over the case without the decoupling. However, the decoupling is still not perfect. This is because two times a probability of 0.01 is not equal to once a probability of 0.02 in our infection use case, since infected pedestrians are able to infect others (exponential growth). This is why the numbers of infections for the 0.4 time step length are still rising quicker than the 0.8 time step length case.

Seconds based decoupling

Another approach which decouples the infection rate from the step length of the simulation is to only check for a spread of infections for each fully passed second[9]. This is different to the original implementation in `SIRGroupModel.update()` where it is done for every simulation step. For this approach as well, the attribute `infectionRate` is interpreted as a rate per seconds. During the `SIRGroupModel.update()` method, one then only needs to count the number of total passed seconds since the last iteration and run this many iterations of infection rounds at once (see pseudo-code at algorithm 3. Figure 18c shows the results of the decoupling for two tested simulation step length of 0.4 s and 0.8 s. Comparing the results to 18b, one can clearly see the advantage of this decoupling method.

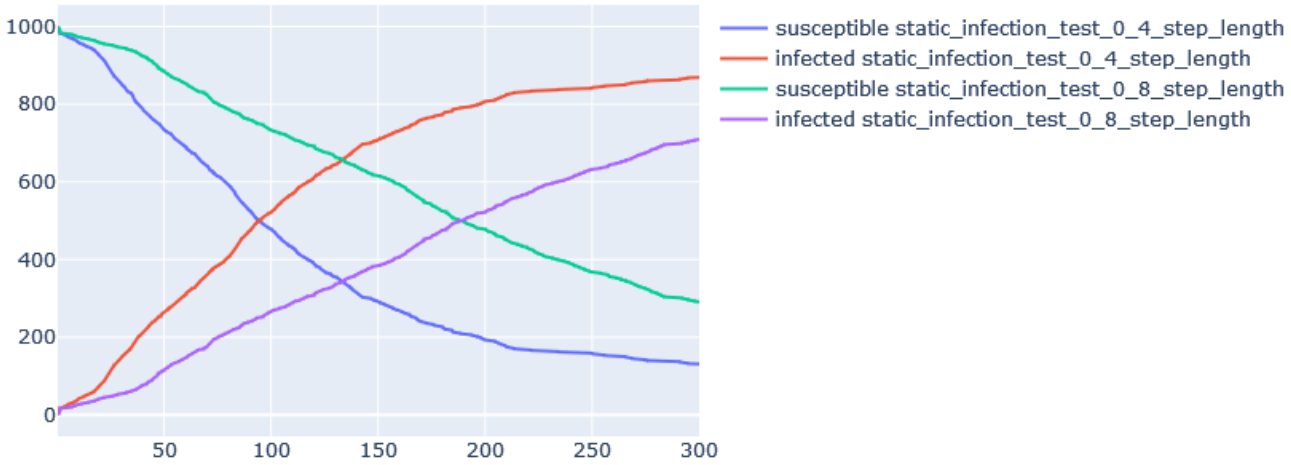
Algorithm 3 Execute function for every passed second

Require: $lastCallTime$, $lastRest$, $thisCallTime$
  $delta \leftarrow thisCallTime - lastCallTime + lastRest$
  $numIterations \leftarrow (int)\ delta$
  $lastRest \leftarrow delta - (int)\ delta$
  $lastCallTime \leftarrow (int)\ thisCallTime$
  for $k \in \{1, ..., numIterations\}$ **do**
    InfectionTest
  end for

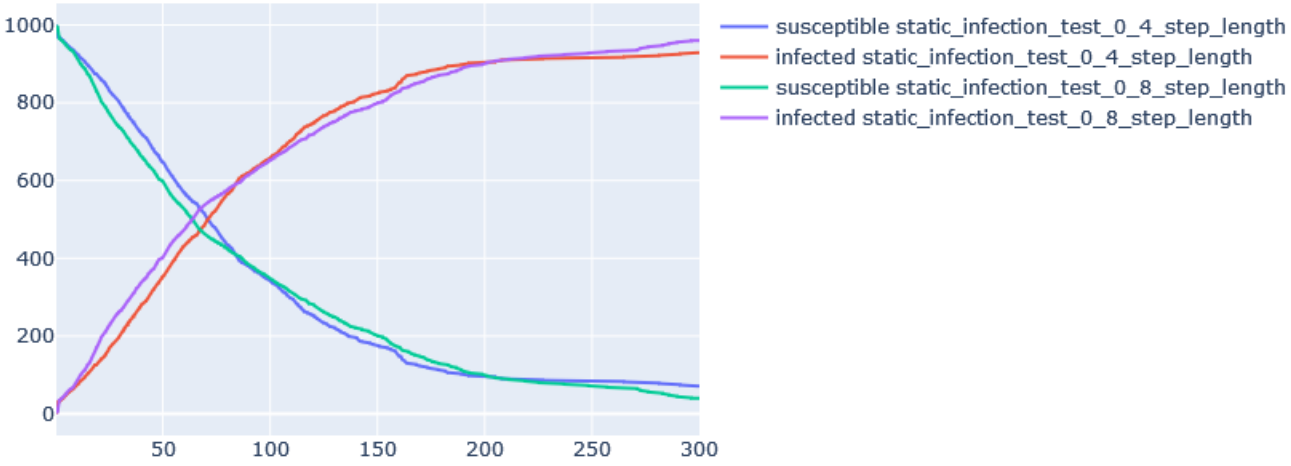
In order to make the above described decoupling optional, a flag `isInfectionRatePerSecond` has been added as an attribute to the `SIRModelAttributes` class. Setting this flag to `true` enables the decoupling and interpretes the `infectionRate` as a rate per seconds. Leaving this flag at `false` resets the original behaviour where the `infectionRate` is treated every iteration step of the simulation.

[8] https://gitlab.lrz.de/00000000014A9334/mlcs-ex2-vadere/-/blob/main/scenarios/task_4_6_decoupling/time_step_based_implementation/SIRGroupModel.java

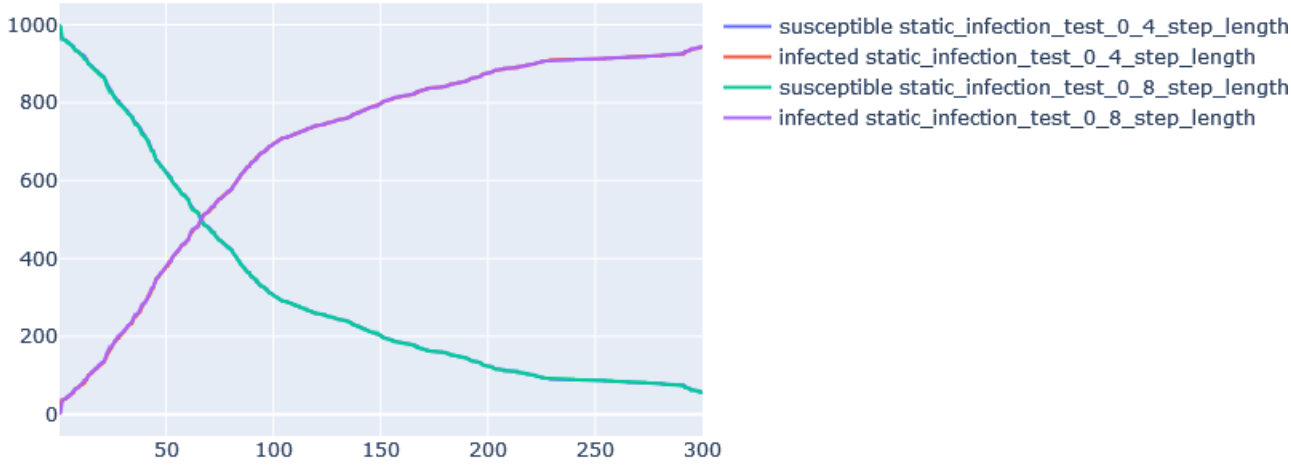
[9] https://gitlab.lrz.de/00000000014A9334/mlcs-ex2-vadere/-/blob/main/SIRGroupModel_task4/sir/SIRGroupModel.java



(a) Without decoupling of infection rate and step time length.



(b) Simulation step based decoupling.



(c) Seconds based decoupling.

Figure 18: Visualization of the static infection scenario of 1000 randomly distributed pedestrians in a 30.00 m x 30.00 m squared room. 10 pedestrians are infected at the very beginning, the maximum infection distance is set to 1.0 m and the infection rate is 0.01. Shown are two simulated scenarios with the different step sizes of 0.4 s and 0.8 s. The x-axis of the plot shows the iteration time in seconds while the y-axis shows the total numbers of pedestrians belong to each group.

Possible extensions of the current SIR model

The following tries to motivate for possible extensions of the current implementation of the SIR model apart from the already implemented *recovered* state in the next chapter.

Smooth infection distance boundary

In the current implementation, the parameter `infectionMaxDistance` acts as a hard decision boundary on which neighboring pedestrians are considered to possible become infected with a constant chance equal to the value of the `infectionRate`. This hard decision boundary in distance and the fixed infection rate within that boundary is not very realistic. Instead, one could remove the fixed distance boundary and at the same time make the infection rate dependent on the distance to the currently infection person. One could even go one step further and additionally model the particles of the infection vapor one by one.

Manually define who is infected at the beginning of the simulation

Right now, there is a parameter `infectionsAtStart` that defines the start population of infected pedestrians. However, one currently cannot choose manually which pedestrians are infectious. In order to increase the possibility of recreated simulation scenarios, it would be useful to be able to assign the infected status on a pedestrian by pedestrian basis, right from within the Vadere GUI.

Model infection as an interaction of two pedestrians

The current implementation of the infection rate as set by the parameter `infectionRate` combines two probabilities at once:

1. probability of an already infected person A to spread the disease. This depends on factors such as if the pedestrian is wearing a face mask and for how long the pedestrian has already been infected.
2. probability of another yet uninfected person B to become infected. This depends on factors such as if the person is wearing a mask and others such as the age of the person or the condition of the immunity system are also plausible.

Instead of combining both probabilities into one single infection rate, one could decouple these as described above in order to achieve greater degree of realism. This idea could be combined in a model of the disease-causing particles in the air, where factors of pedestrian A define the amount and velocity of spread particles and factors of pedestrian B account for a resistance against these.

Infection rate orientation dependent

To improve the realism of the SIR model, one could think of making the infection rate dependent on the orientation. The infection probability is likely to be lower when the infection pedestrian is standing in front of the other pedestrian while looking into the same direction.

Considering an additional incubation period

Typically, there is a time between becoming infected and being able to infect others, the so-called incubation period. The Susceptible-Exposed-Infectious-Removed (SEIR) model incorporates this delay by adding an additional "exposed" status.

Summary

While there is still a lot of room for improvement, the performed tests have shown that the already implemented version of the SIR model can be used to simulate the spread of an infection in various scenarios and to analyze

the impact of the infection rate.

Report on task 5/5, Analysis and Visualization of Results

Motivation

In reality, people do not stay infected forever, but eventually recover from a disease. In order to improve the model, an additional state is required that is implemented and tested as described in this section.

Setup

For this task, the setup of task 4 is used. In particular, the SIR model is integrated and the `SIRType`, `AttributesSIRG`, and `SIRGroupModel` classes were modified. Furthermore, the previous visualization with Plotly/Dash is adapted by modifying the `SIRvisualization.utils` class in order to visualize the additional recovered state.

Implementation of recovered state

The additional state, *recovered*, represents people that have been infected, have recovered from the disease, and consequently are immune. In the SIR model, one can easily add such states by adding a type in the `SIRType` class.

Implementation of recovery rate

In order to decide if a person transitions from *infected* to *recovered*, an independent probability has to be added. This recovery rate is included in the `AttributesSIRG` class such that it is accessible and modifiable in the simulation. The transition is implemented in the `SIRGroupModel` class and is based on the implementation of the transition from *susceptible* to *infected*. For each time step, a decision is made whether an infected pedestrian recovers and is assigned to their new group depending on the recovery rate.

Visualization of recovered state

Having a new state, also requires to adapt the implementation of the visualization with Plotly/Dash. Therefore, the observation of recovered people is plotted in addition to the group of susceptible and infected people. Because all people in this simulation go through a predefined process from susceptible via infected to recovered, one simply can decrease the number of infected people and increase the number of recovered people if a person recovers in a specific time step.

Tests

The implementation is tested by the following three tests. For each test, we start with a description of the scenario and an assumption for the results before executing it. Then, our observations are described and results and limitations are discussed.

Test 1: Implementation of recovery

Description of Scenario:

This scenario consists of a 30 m x 30 m squared field and 1000 randomly distributed, non-moving pedestrians of which 10 are infected and 990 are susceptible. The construction of the scenario is illustrated in Figure 19. The simulation is executed with Vadere and parameters are set by using the Vadere GUI. The experiment is

observed for 100 iterations with 10 infections at start and the same value for infection and recovery rate. Due to the fact, that pedestrians to not move in this scenario, time is not an important factor and both rates are set to 0.05.

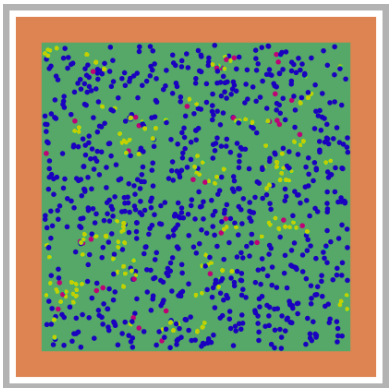


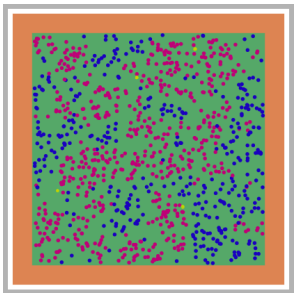
Figure 19: Scenario of Test 1 in the first few time steps: 1000 randomly distributed pedestrians; Distinction between susceptible (blue), infected (turquoise), and recovered (pink) pedestrians.

Assumption:

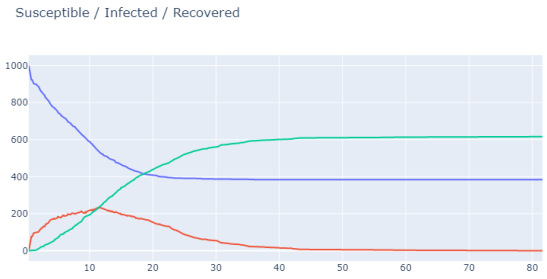
In the beginning, people start to get infected before the number of recoveries increases. Finally, the majority of people will be recovered because most people will have been infected at some point. After the simulation, the number of infected people should be zero.

Observation:

The observations are represented in Figure 20. As expected, the number of infected people rapidly increases in the beginning but decreases after about 10 time steps. The descent is based on two facts: 1. people recover and change their state to *recovered*, 2. recovered people do no infect others. Compared to Task 4, there are a lot less people that have ever been infected even though the infection rate is higher in this scenario. Aside from that, the number of recovered people has its strongest increase when the number of infected people is the highest and the curve flattens until every infected person recovers.



(a) Visualization of the final state with the Vadere GUI: distinction between susceptible (blue), infected (turquoise), and recovered (pink) pedestrians



(b) Visualization of Test 1 with Plotly/Dash: development of the number of susceptible (blue), infected (red) and recovered (turquoise) people

Figure 20: Observations of Test 1

Discussion:

Even though this is a simple scenario, the development of the curves for infected and recovered pedestrians reflect realistic results. One is able to observe that only infected people have a transition to the *recovered* state. In addition to that, the recoveries make the curves stagnate a lot earlier than in the previous simulation because the infection and the recovery rate are the same and the infections and recoveries are not independent. In reality, flattening the curve of infected people is particularly important because less high excursions result in a smaller number of people who simultaneously depend on medical treatment. Even though, the results seem to be realistic, there are some limitations of this scenario. The number of pedestrians is limited and there is no movement which leads to a saturation of the curve of infected pedestrians.

Test 2: Experiment with infection and recovery rate

Description of Scenario:

In Test 1, we chose the same value for the infection rate and the recovery rate. For the second test, the effects of different infection and recovery rates are to be observed with the same scenario as Test 1. In order to be able to compare results, we choose two values $a = 0.01$ and $b = 0.05$. Apart from the scenario of Test 1, we create three additional scenarios by combining each of the values a and b for the infection and the recovery rate. This results in the following three cases:

1. $a = 0.05$ and $b = 0.05$
2. $a = 0.01$ and $b = 0.01$
3. $a = 0.05$ and $b = 0.01$
4. $a = 0.01$ and $b = 0.05$

Assumption:

Case#2 has similar results than Case#1 but there are fewer pedestrians that will ever be infected. In Case#3, there are a lot of infections before the curves flatten. For Case#4, only few people get infected because the recovery rate outruns the infection rate such that infection are avoided early on.

Observation:

The results meet the expectations and are visualized in Figure 21. One can clearly see that there is a higher number of infected people if the infection rate is higher. Only a higher recovery rate is able to damp the curve of infected pedestrians. Figure 21c shows that for a smaller recovery rate, the curves of susceptible and infected people become more similar to those of Task 4.

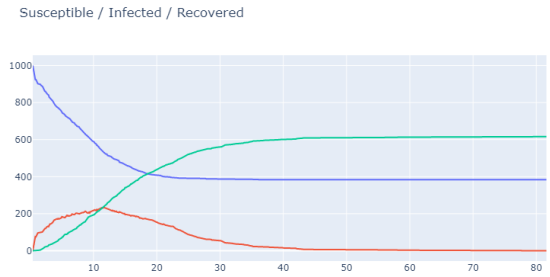
Discussion:

The values for a and b were chosen arbitrarily such that the results are meaningful. Case#1 and Case#2 indicate what happens if the common value for both rates is higher or smaller. If a and b become more similar, Case#3 and Case#4 degenerate to Case#1 or Case#2 depending on the chosen value. If a and b diverge, similar results as for Case#3 and Case#4 are to be observed.

Test 3: Supermarket scenario

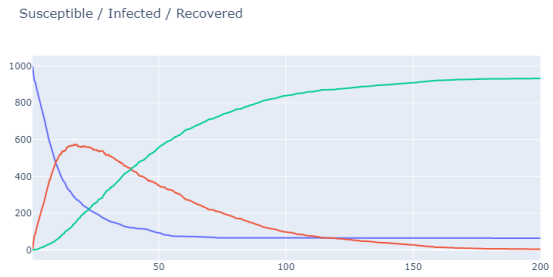
Description of Scenario:

In reality, people are not static but move around and spread disease not only to their neighbors but everyone who they meet on their way. Therefore, a supermarket scenario is to be created where customers enter a

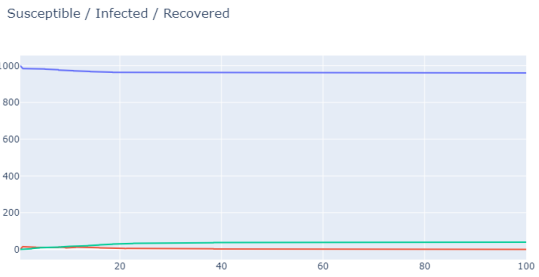


(a) Case#1: higher value for infection rate and recovery rate

(b) Case#2: smaller value for infection rate and recovery rate



(c) Case#3: higher value for infection rate and smaller value for recovery rate



(d) Case#4: smaller value for infection rate and higher value for recovery rate

Figure 21: Observations of Test 2: Visualization of four cases with Plotly/Dash: development of the number of susceptible (blue), infected (red) and recovered (turquoise) people

store, wander around before they eventually leave again. As a starting point, we chose to copy and adapt the supermarket scenario provided in the Vadere project[10]. Starting the simulation without modifications reveals that customers tend to block each other such that customers do not leave the store. The removal of some obstacles allows customers to walk past each other. Additional, customers are able to enter the supermarket with entrances in three different corners. Other factors also have an influence on the infections and are therefore examined in the experiment. The overall construction is illustrated in Figure 22.

The goal of Test 3 is to find relations between the number of infections and factors like the personal space width or number of customers. In order to make draw reasonable conclusions, other factors like the infection rate and recovery rate are fixed with the values 0.2 and 0.005, respectively. These values are chosen because they lead to clear results when running the simulation for 1000 iterations.

The experiment is divided into four cases based on two values for two factors. The first parameter is the number of customers that depends on the number of pedestrians that are distributed by each entrance. All three entrances receive the value d as `distributionParameters`. A higher value results in a smaller number of customers that enter the store. The second parameter is `pedPotentialPersonalSpaceWidth` p that determines the distance that customers keep to each other. Since the distance of infections has a maximum of 1.0m, the values for the personal space width is chosen to be either below or above this value. The experiment tests the following four cases:

1. $d = 20.0$ and $p = 0.5$
2. $d = 40.0$ and $p = 0.5$
3. $d = 20.0$ and $p = 1.5$
4. $d = 40.0$ and $p = 1.5$

Assumption:

With a reduced number of customers there are less infections because there are fewer contacts. If people keep a distance of at least 1.0m, they do not get infected.

Observation:

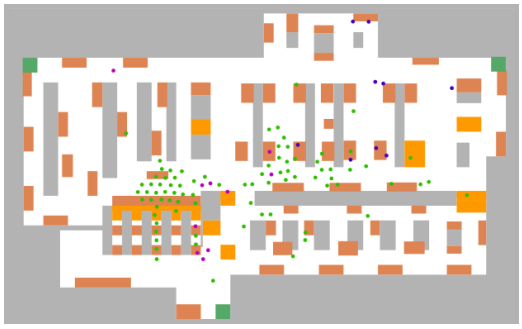
Our observations are represented in Figure 22 for the time step $t = 800$. One is able to see that a reduced number of customers does not only avoid customers to block each other's ways but also reduces the number of infections. In Case#1, only 9% (9 out of 102) of the customers stay susceptible, compared to 15% (7 out of 47) in Case#2. Even though, this factor has an effect on the number of infections, it mainly prevents the supermarket from getting too crowded. A more effective factor with respect to a reduction of infections is to increase the personal space width. Because the infection is possibly passed on to the pedestrians within a range of 1.0m, setting the personal space width above this threshold successfully help to let customers stay susceptible. This becomes obvious when comparing Case#1 with Case#3 or Case#2 with Case#4. A majority of people has not been infected in the latter cases (blue coloring indicates that a person is susceptible). However, if we choose different values for the personal space width, other problems occur for this scenario. Increasing the personal space width leads to a crowded supermarket where customers block each other's ways until nobody is able to enter or exit the supermarket. Decreasing the personal space width while allowing only a few of customers, makes them leave the store quickly and prevents a high number of infections. Anyhow, a higher value for the personal space width is desirable because it is independent of the number of contacts. For a smaller value, all customers could be interested in the same goods and infect each other while they stay susceptible if they do social distancing.

The best results, where almost no customer gets infected, are achieved by reducing the number of customers while having a minimum personal space width of 1.0m guaranteed (see: Case#4).

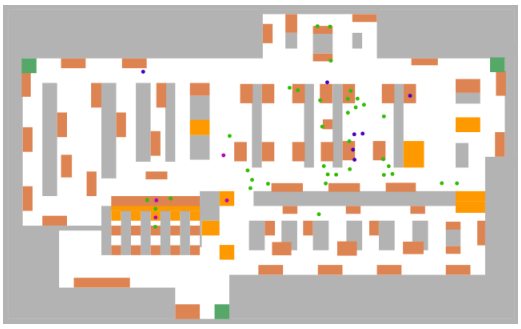
[10] `https://gitlab.lrz.de/vadere/vadere`

Discussion:

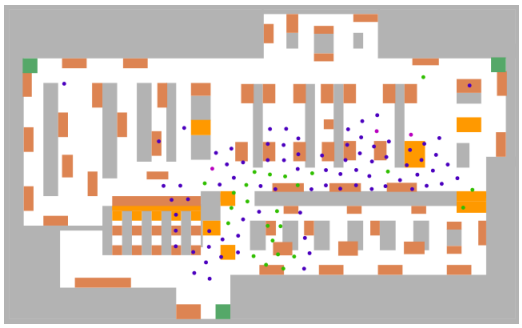
The simulation of the supermarket scenario encounters a variety of limitations. Firstly, infected people only enter the store in the beginning. At this point, there is no implementation that regularly produces infected customers who enter the store. Furthermore, customers would not recover in a supermarket when they have been infected after entering. In reality, people would enter the store either susceptible, infected, or recovered. During their stay, customers possibly have one transition, either from susceptible to infected or from infected to recovered. But in our case, they are able to go through the whole process from susceptible to recovered.



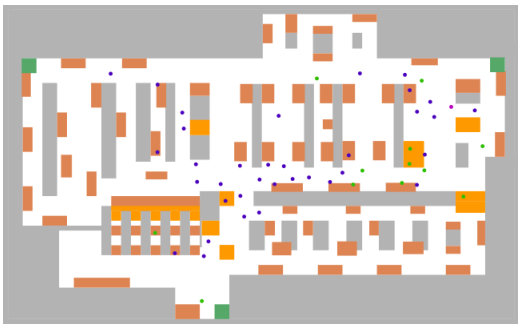
(a) Case#1: space width below 1.0m and a higher number of customers lead to many infections and people do not stay susceptible



(b) Case#2: space width below 1.0m and a smaller number of customers still lets the majority get infected



(c) Case#3: space width above 1.0m and a higher number of customers do not lead to many infections and people stay susceptible



(d) Case#4: space width above 1.0m and a smaller number of customers almost completely avoids infections and people stay susceptible

Figure 22: Observations of Test 3: Visualization of four cases (t=800) with the Vadere GUI: development of the number of susceptible (blue), infected (red) and recovered (green) people

Bonus Tests: Bottleneck Scenario

Description of Tests:

Another interesting scenario contains bottlenecks where people are moving and are possibly forced to reduce their distance to each other. In order to further test our implementation, we design three tests out of this scenario. All tests are to be compared with a reference setting. We consider the personal space width p, the infection rate i and the recovery rate r.

1. Reference: $p = 1.5$m, $i = 0.05$ and $r = 0.01$
2. Test#1 - effect of social distancing: $p = 0.5$m, $i = 0.05$ and $r = 0.01$
3. Test#2 - effect of a small infection rate: $p = 1.5$m, $i = 0.01$ and $r = 0.01$
4. Test#3 - effect of a high recovery rate: $p = 1.5$m, $i = 0.05$ and $r = 0.05$

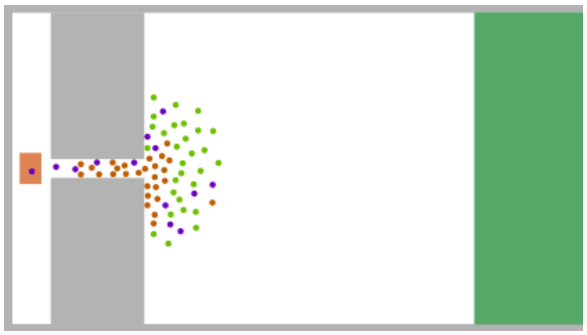
The Vadere project provides a bottleneck scenario where pedestrians pass a 2m wide bottleneck. 10 infected and 90 susceptible pedestrians start on the right side of a bottleneck and aim to reach a target on the other side (see Figure 23). The distance between source and bottleneck is reduced and the SIR model is applied to the provided scenario. In addition to that, parameters are adapted to the corresponding test case.

Assumption:

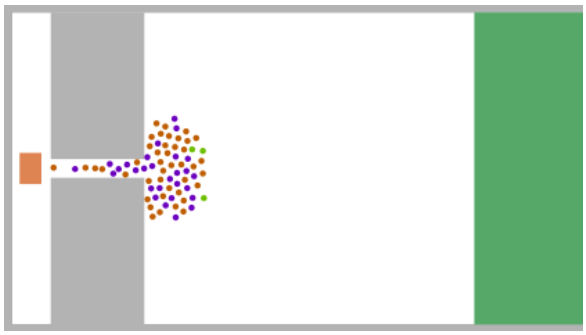
A majority of people is able to stay susceptible if they do social distancing. A small infection rate and a high recovery rate effectively avoid that a lot of people get infected.

Observation:

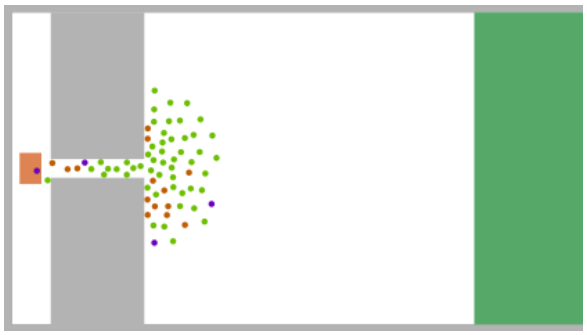
Figure 23 visualizes results of the reference setting and the three tests at time step 40.



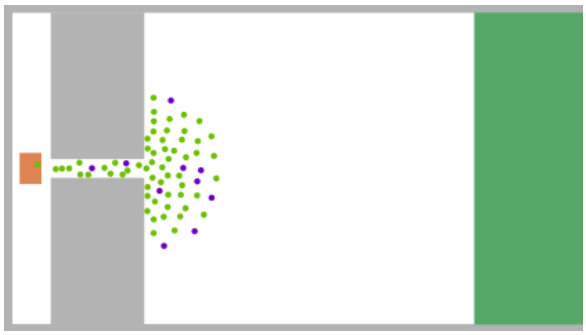
(a) Reference: space width above 1.0m leads to a slow contamination where a minority stays susceptible



(b) Test#1: space width below 1.0m leads to a fast contamination



(c) Test#2: space width above 1.0m and a small infection rate lead to a clear reduction of infections



(d) Test#3: space width above 1.0m and a high recovery rate prevents that the majority of people gets infected

Figure 23: Observations of the Bonustests: Visualization of four cases (t=40) with the Vadere GUI: development of the number of susceptible (green), infected (orange) and recovered (violet) people

Test 1: Figures 23a & 23b
As expected, the majority of people will get infected if they abstain from social distancing. But surprisingly, a desired distance of 1.5m does not guarantee that people will stay susceptible. While pedestrians are waiting until they are able to pass the bottleneck they infect each other, resulting in the fact that mainly infected pedestrians reach the goal on the other side of the bottleneck.

Test 2: Figures 23a & 23c
Since social distancing seems to have a minor effect on the number of infections in this setting, we need more effective measures in order to help pedestrians stay susceptible. When comparing Figures 23a & 23c, one is able to see that in the latter case, susceptible pedestrians are able to pass the bottleneck. This indicates that a reduced infection rate has a higher effect regardless of the distance between single pedestrians. Even for a personal space width below 1.0m, similar results are achieved.

Test 3: Figures 23a & 23d
With a high recovery rate, almost nobody gets infected because infectious people recover fast enough such that they cannot transmit diseases. This way, the majority of people reaches the target without getting infected. Compared to a small infection rate (see Figure 23c), a high recovery rate is even more effective.

Discussion:

Observing the results reveals that social distancing does not work well for situations where people do not stay distant. These bottleneck situations occur for small passages and entrances where crowds are built and people tend to ignore desired distances in order to reach a target. For these cases, more effective measures are required. While higher recovery rates can only be achieved by medications, small infection rates have multiple possible causes. In particular for these situations, measures like wearing a face mask are necessary in order to reduce the infection rate.

Interestingly, a smaller distance does not necessarily have a negative effect on the number of infections as opposed to situations like the supermarket scenario. If people reduce their distance to each other, more people fit through the passage resulting in a better flow. This way, people do not have to wait that long and reduce the amount of time where they are exposed to infectious people.

References

[1] Peter Molnar Dirk Helbing. Social force model for pedestrian dynamics. 1998.

[2] Gerta Köster Felix Dietrich. Gradient navigation model for pedestrian dynamics. 2014.