Report for exercise 2 from group H

Tasks addressed:	5
Authors:	Muhammed Yusuf Mermer () Milena Schwarz () Antonia Gobillard () Nayeon Ahn () Hammad Basit ()
Last compiled:	2024–05–18
Source code:	https://github.com/Hammad-7/MLCMS_Exercises.git

The work on tasks was divided in the following way:

Muhammed Yusuf Mermer () **Project lead**	Task 1	20%
	Task 2	20%
	Task 3	20%
	Task 4	20%
	Task 5	20%
Milena Schwarz ()	Task 1	20%
	Task 2	20%
	Task 3	20%
	Task 4	20%
	Task 5	20%
Antonia Gobillard ()	Task 1	20%
	Task 2	20%
	Task 3	20%
	Task 4	20%
	Task 5	20%
Nayeon Ahn ()	Task 1	20%
	Task 2	20%
	Task 3	20%
	Task 4	20%
	Task 5	20%
Hammad Basit ()	Task 1	20%
	Task 2	20%
	Task 3	20%
	Task 4	20%
	Task 5	20%

Report on task 1, Setting up the Vadere environment

Task 1 includes downloading the Vadere [3] software, e.g. downloading the compiled JAR files from the official website. They can be run by navigating to the location where the files are saved via the command window. Then, the following line must be run in order to open the Graphical User Interface: `java -jar vadere-gui.jar`

In this task, we will take a look at three scenarios that were already highlighted in our automaton from exercise sheet 1. In Vadere, the Optimal Steps Model is used to run the scenarios. "The Optimal Steps Model (OSM) is a pedestrian locomotion model that operationalizes an individual's need for personal space." [6]

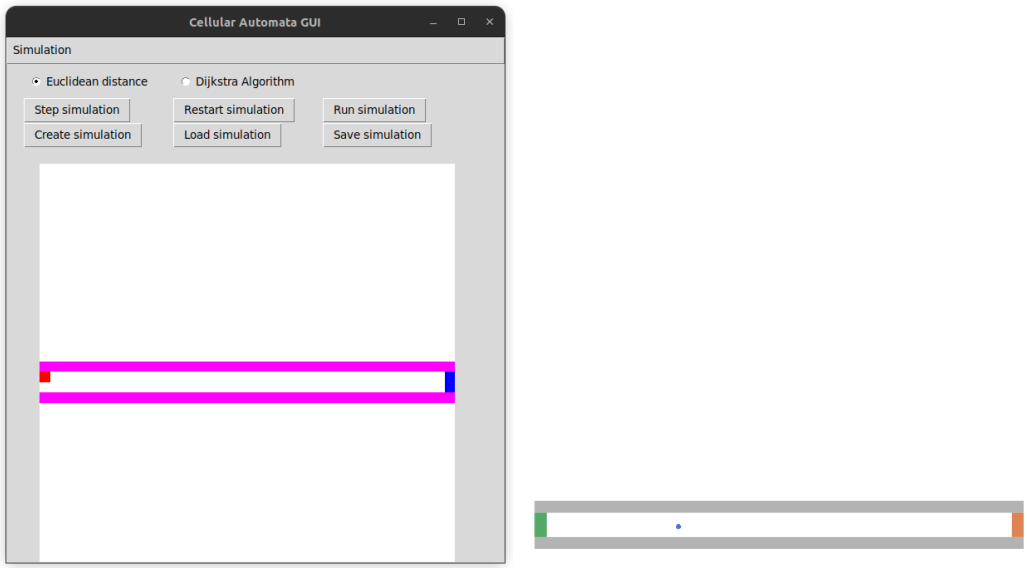
RiMEA scenario 1 (straight line):
Parameters of simulation 1

- height and width of the pedestrian: 0.4m
- corridor: 40m long

The Vadere software offers a larger variation of features than our automaton from exercise 1. The following tabs are available in the application:

- Simulation: settings e.g. for time can be found
- Model: where different templates can be loaded
- Psychology: not used for this exercise
- Topography: JSON structure of the current scenario
- Stimuli: not used for this exercise
- Data output: determine which files will be created when running a scenario
- Topography creator: GUI with different buttons to create your own scenario

After running a scenario, a new tab appears, called 'Post-Visualization'. It is useful to check upon any arbitrary time step of the simulation.

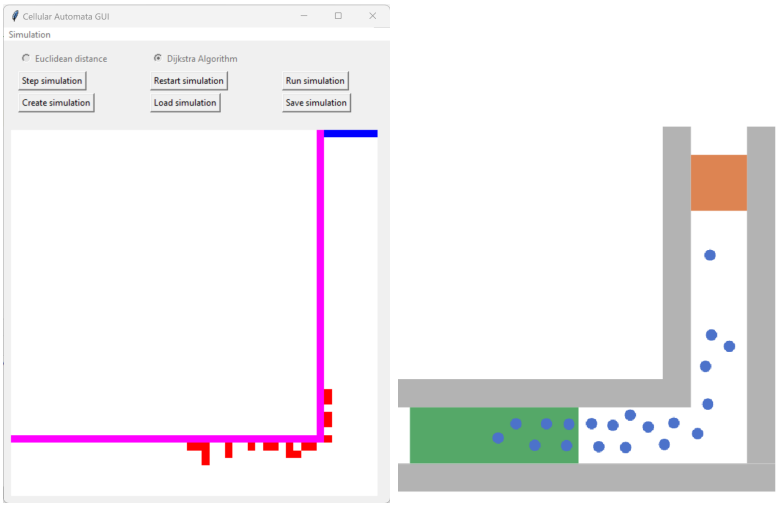


(a) Automaton simulation (b) Vadere simulation

Figure 1: RiMEA scenario 1

RiMEA scenario 6 (movement around a corner):
Of course, the Vadere software makes it easier to recreate scenarios. The pedestrians get spawned randomly

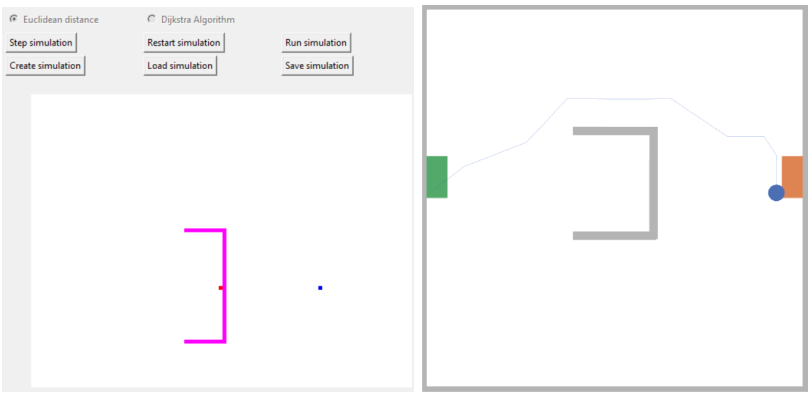
throughout the source which can simply be drawn out as a rectangle or another desired shape. The simulation for the RiMEA scenario 6 also differs: In our automaton, the pedestrians moved close to the border to take the shortest path to the target at the end of the corridor. The pedestrians in the OSM on the other side space out evenly, even if the path is a little longer that way.



(a) Automaton simulation (b) Vadere simulation

Figure 2: RiMEA scenario 6

Chicken Test:
In our automaton, we can only see the current state of the simulation, whereas in Vadere we have the option to display trajectories which allow us to see the exact path the pedestrian took to reach the target.



(a) Automaton simulation (b) Vadere simulation

Figure 3: Chicken Test

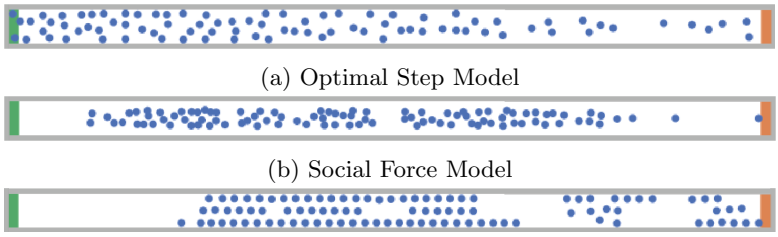
Report on task 2, Simulation of the scenario with a different model

In the second task, the same scenarios are used, however, two different models are considered together with the OSM.

- Social Force Model (SFM): "It is suggested that the motion of pedestrians can be described as if they would be subject to 'social forces'. These 'forces' are not directly exerted by the pedestrians' personal environment, but they are a measure for the internal motivations of the individuals to perform certain actions (movements)." [5]

- Gradient Navigation Model (GNM): "The model uses a superposition of gradients of distance functions to directly change the direction of the velocity vector. The velocity is then integrated to obtain the location." [4]

RiMEA scenario 1 (straight line):
For the SFM, pedestrians move closer together compared to OSM. The degree of compaction is distinctly higher: we can distinguish one single lane in the center of the corridor. That is not the case for the OSM for which the movement seems more disorganized. The GNM looks the most different compared to the two above. Pedestrians split into three distinctive rows and move along a grid, sometimes changing lanes. This model is the one that makes the movement here the most "organized".



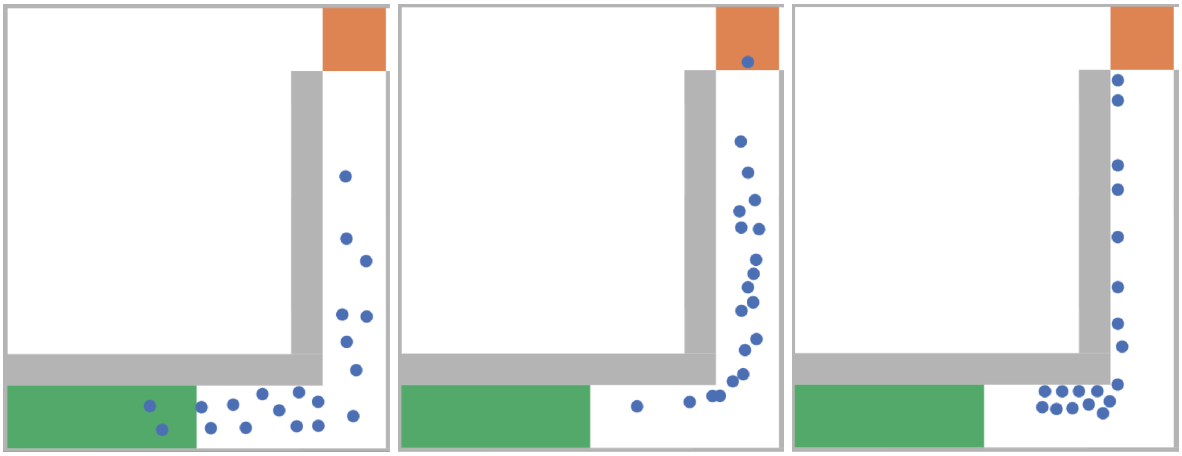
(a) Optimal Step Model

(b) Social Force Model

(c) Gradient Navigation Model

Figure 4: RiMEA Scenario 1 using different models

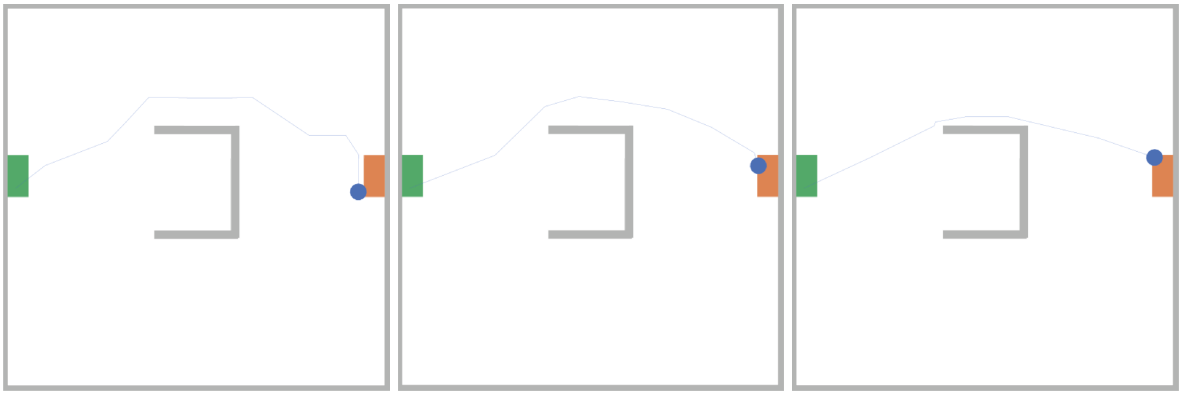
RiMEA scenario 6 (movement around a corner):
When moving around a corner, pedestrians behave in distinct ways for each of the models. In the OSM, they are spread out and move evenly toward the target. The SFM shows the pedestrians walking close to the middle of the corridor, whereas in the GNM, they move close to the wall, taking the shortest way to the target. Both of these models however do not make good use of the available space and therefore look less natural compared to the OSM.



(a) Optimal Step Model (b) Social Force Model (c) Gradient Navigation Model

Figure 5: RiMEA Scenario 6 using different models

Chicken Test:
All three of the models successfully pass the chicken test. However, differences in the pedestrian's movements can still be observed. The pedestrian in the OSM moves in an angular pattern around the obstacle. In the SFM, the pedestrian moves more naturally, leaving space between the obstacle and its walking path. The GNM shows similar behavior but leaves merely any space in between the wall and the pedestrian.



(a) Optimal Step Model (b) Social Force Model (c) Gradient Navigation Model

Figure 6: Chicken Test using different models

Report on task 3, Using the console interface from Vadere

In the current task, we engaged with the Vadere[3] simulation software via both console and graphical user interface (GUI), with a particular emphasis on the console interface. Utilizing the scenario file crafted for the RiMEA 6 scenario (corner) established in the initial task, we executed it through Vadere-console. This task involved a comparative analysis of output files generated by both the GUI and console interfaces to ascertain if they yield identical results. So, we conducted a comparative analysis of output files RiMEA 6 scenario using a Python script designed to compare the files by their byte size. Our findings showed that the output files were identically the same across all scenarios, indicating a consistent performance in the outputs generated. This consistency is crucial for the reliability of our simulations and suggests the scenarios are functioning as expected.

Vadere's design facilitates the extraction of actionable data for post-processing through the use of "output processors" within a scenario. These processors are seamlessly integrated into the user interface via the "Data output" tab in a scenario's settings. The default configuration generates a `postvis.traj` file, encapsulating comprehensive data such as positions, IDs, and targets for each pedestrian throughout the simulation.

To accomplish the objectives of this task, we modified the corner scenario file using Python, subsequently running Vadere on the altered scenario file. This modification process entailed the insertion of a single pedestrian without resorting to a source field.

Prior to any modification, it was imperative to understand that Vadere scenario files are essentially text files in JSON format. Through the GUI, adding a pedestrian is accomplished by selecting the "pedestrian" icon in the topography creator tab. A critical observation was the alteration in the `dynamicElements` list within the topography block of the scenario file after the inclusion of a pedestrian. It is crucial to note that unless a target ID (corresponding to a target object in the scenario) is assigned to the new pedestrian in the `targetIds` list, the pedestrian will remain static.

The procedural steps for this task were as follows:

- Retrieving the "corner scenario" file, as depicted in figure 7a.
- Programmatically adding a pedestrian at the designated location (marked by a red cross in figure 7a), ensuring the placement is clear of any obstacles, and appending this to the list of individual pedestrians.
- Saving the modified scenario file under a distinct name to distinguish it from the original.
- Executing Vadere-console.jar using the newly modified scenario file.

This approach not only demonstrates the adaptability of Vadere in accommodating changes through both GUI and console interfaces but also underscores the importance of precise manipulations in scenario files to achieve desired simulation outcomes.

The particular script is divided into two primary functions, `add_pedestrian` and `update_new_scenario`, and is executed in a main block.

Function: add_pedestrian:

- Purpose: Adds a pedestrian to a specified location in the scenario.
- Inputs:
  - scenario_path: Path to the existing scenario file.
  - x, y: Coordinates where the pedestrian will be placed.
- Process:
  - The scenario file is read as a JSON object.
  - A pedestrian object is created with the specified x and y coordinates. The type is set to 'PEDESTRIAN', and an empty targetIds list is initialized.
  - The script checks for the existence of dynamicElements in the scenario JSON structure. If present, the pedestrian object is appended to this list. If not, an error message is printed.
  - The pedestrian is assigned all target IDs from the scenario's targets, enabling movement towards these targets.

Function: update_new_scenario

- Purpose: Saves the updated scenario as a new JSON file.
- Inputs:
  - scenario_path: Original scenario file path.
  - new_scenario: The updated scenario object with the added pedestrian.
- Process:
  - Generates a timestamp to append to the new scenario file name, ensuring uniqueness.
  - Creates a new file path by appending '_updated' and the timestamp to the base name of the original file.
  - Writes the updated scenario JSON to this new file. If an error occurs during writing, it is printed to the console.

Main Execution Block:

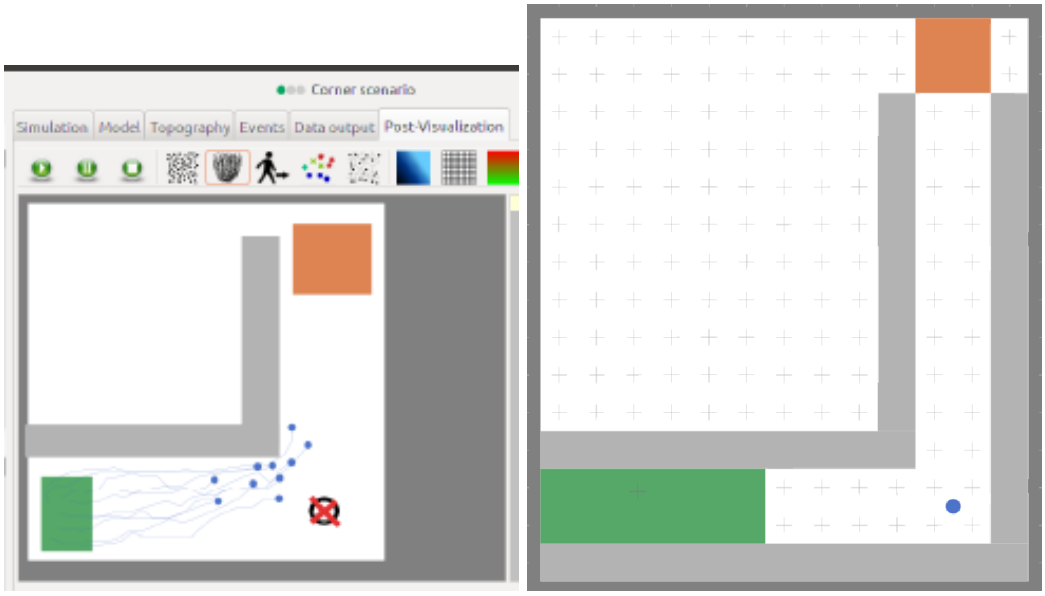
- Sets the scenario_path to the location of the original scenario file.
- Calls `add_pedestrian` to add a pedestrian to the specified coordinates.
- Calls `update_new_scenario` to save the updated scenario to a new file.

Instructions for Running the Updated Scenario:
To run the new scenario in the console and then open it in the GUI, the following commands are provided:

```
java -jar vadere-console.jar scenario-run --output-dir [your output folder]
                                          --scenario-file [your scenario file folder]
java -jar vadere-gui.jar,
```

then navigate to the scenario tab and open the updated scenario file.

This script streamlines the process of modifying simulation scenarios in Vadere, particularly for scenarios that require specific placement of pedestrians and subsequent visualization and analysis in both the console and GUI interfaces.
The result can be seen in figure 7b.



(a) Setup of the scenario (b) New pedestrian placement

In our simulation task, we introduced a new pedestrian into the scenario. The addition of this new pedestrian, positioned closer to the target, would cause different outputs of the scenario. The scenario output results minorly differed in `overlaps.csv`, particularly 'distance-PID3' and 'overlaps-PID3'. We could observe that the first ones to arrive were the added pedestrians that took 9 seconds to get to the target, due to their closer proximity. The last ones of the pedestrians originated from the standard starting point and took 30.293 seconds to reach the target. This outcome was achieved by the OSM model.

Report on task 4, Integrating a new model

After having used the Vadere software with its existing features, we have to implement a new feature by adding the SIR (Susceptible-Infectious-Removed) model into the source code [7]. Firstly, we need to set up our development environment. We used IntelliJ IDEA as our IDE and the latest Java development kit (OpenJDK). We copied the SIR files provided to us on Moodle and pasted them at the proper locations as specified by the first line of each file. In order to run it, we first have to rebuild the software and also do a clean maven build. **Figure 8** shows the UML diagram representing all the new classes that have been included for the SIR model. We go into more details for each file now:

- **AbstractGroupModel**: An abstract that serves as the base for implementing various group models in the Vadere software.
  `Methods:`
  - *registerMember(Pedestrian ped, T currentGroup):* An abstract method for registering a pedestrian to a specified group.
  - *getNewGroup(int size):* An abstract method for obtaining a new group with the given size.
  - *getNewGroup(final int id, final int size):* An abstract method for obtaining a new group with the specified ID and size.
  - *assignToGroup(Pedestrian pedestrian):* An abstract method for assigning a pedestrian to a group.
- **AttributesSIRG**: This class extends the `Attributes class` and is used to define the attributes specific to the SIR model.
  `Methods:`
  - *getInfectionsAtStart():* Returns the value of `infectionsAtStart` attribute.
  - *getInfectionRate():* Returns the value of `infectionRate` attribute.
  - *getInfectionMaxDistance():*Returns the value of `infectionMaxDistance` attribute.

Figure 8: UML diagram representing SIR class

- **FootStepGroupIDProcessor**: This class extends `DataProcessor<EventtimePedestrianIdKey, Integer>` and implements the `ModelFilter` interface. This class is a data processor designed to extract and process information related to group IDs based on footstep data.
  `Methods:`
  - *doUpdate(SimulationState state):* It updates the processor based on the simulation state. For each group, it iterates over group members and footstep data to extract and associate group IDs with pedestrianIds and event times.
  - *toStrings(EventtimePedestrianIdKey key):* It converts the data associated with a given key into a string array.
- **SIRType**: This file defines enumeration for different types of individuals within the context of the SIR model which consists of `ID_INFECTED`, `ID_SUSCEPTIBLE` and `ID_REMOVED`.
- **SIRGroup**:This file represents a SIR Group.
  `Methods:`
  - *getId():* Returns group ID.
  - *equals(Group o):* Compares with another group based on their IDs.
  - *getMembers():* Returns a list of pedestrians who are members of the group.
  - *getSize():*Returns the size of the group
  - *isMember(Pedestrian ped):* Checks if a pedestrian is a member of the group.
  - *isFull():* Checks if the group is full.
  - *getOpenPersons():* Returns the number of remaining spots in the group.
  - *addMember() and removeMember():* Adds or remove a pedestrian
  - *setPotentialFieldTarget():* Sets the potetntial field target for the group.
  - *getPotentialFieldTarget():* Returns the potetntial field target for the group.
- **SIRGroupModel**:This file implements the SIR Group model.
  `Methods:`
  - *initialize():* Initializes the SIR group model with required attributes.
  - *setPotentialFieldTarget() and getPotentialFieldTarget():* Sets or returns the potential field target for the group model.
  - *getFreeGroupId():* Determines a free group ID based on infectionRate and total infected pedestrians.
  - *getGroup():* Reutrns the SIR group of the pedestrian.
  - *isMember(Pedestrian ped):* Checks if a pedestrian is a member of the group.
  - *registerMember():* Registers a pedestrian as a member of a specific SIR group.
  - *getNewGroup():* Overloaded function to return either a free group ID or create a new one.
  - *assignToGroup():* Overloaded function to assign a pedestrian to a specific SIR group or a free SIR group.
  - *elementAdded() and elementRemoved():* Adds or remove a pedestrian from a group.
  - *update():* Updates the SIR groups and simulate infections.

Correct colour visualization: Currently, the colours are not visualised correctly for different groups (infected, susceptible, removed). In order to do that, we modified the class `SimulationModel.java` to include different colours for different groups. Figure 9 shows the highlighted code that has been changed to include the colours for different groups. We chose red for infected, blue for susceptible, and green for removed. This change only allows us to see the groups during the simulation and does not work for post-visualization.

```java
public Color getGroupColor(@NotNull final  Pedestrian ped) {
    if (ped.getGroupIds().isEmpty() || (!ped.getGroupSizes().isEmpty() && ped.getGroupSizes().getFirst() == 1)) {
        return config.getPedestrianDefaultColor();
    }
    int groupId = ped.getGroupIds().getFirst();
    Color c = colorMap.get(groupId);
    //different colors for different SIR groups
    Color infected = new Color( r: 255,  g: 0,  b: 0);
    Color susceptible = new Color( r: 0,  g: 0,  b: 255);
    Color removed = new Color( r: 0,  g: 255,  b: 0);
    colorMap.put(0,infected);
    colorMap.put(1,susceptible);
    colorMap.put(2,removed);
    if (c == null) {
        c = new Color(Color.HSBtoRGB(random.nextFloat(),  saturation: 1f,  brightness: 0.75f));
        colorMap.put(groupId, c);
    }
    return c;
}
```

Figure 9: Modified code for different groups

Increasing Efficiency in Distance Computation:

If we look into the `LinkedCellsGrid.java` file we can see that `getObjects` is used to retrieve pedestrians near to an infected pedestrian. This function basically looks at every cell of the `grid`. If there is any object in that cell, it will then compare distances to the center. In the end, this function adds pedestrians who are nearer or equal to the radius to the `result` variable, which will be used later with the `infectionRate` attribute to decide whether selected pedestrians are infected or not.

While this method gives correct results, it does not necessarily mean that it cannot be advanced and made more efficient. If the number of pedestrians increases for each group, it can be observed that the number of computations increases. For n number of pedestrians, if we look at the whole grid for them, we would get $O(n * width * height)$ as time complexity, which will be computationally intensive.

When we searched for a better solution, we found out that the `Quadtree` data structure is the best way to implement this function as it is on a map.

Basically, as explained by AJT, `Quadtree` uses both Binary Trees and Recursion to achieve its purpose. While, only leaf nodes store relevant data, root nodes and internal nodes store information to get these data. Internal nodes contain 4 pointers to point to directions of the current point: Northeast, Northwest, Southeast, and Southwest. When someone wants to reach a location, they need to go through quadrant nodes until they get to the leaf node.

So compared to looking at the whole grid, it is more efficient to go through nodes and find the point that we want to reach. It has a time complexity of $O(log(n))$ for searching, which proves that compared to the previous method, it is much more efficient.

However, this method also has flaws. In some situations, if points are near to each other they may cause too much recursion of nodes, as can be seen in figure 11e. However, it can be still said that in most of the situations with spread pedestrians, `Quadtree` data structure helped us to improve the efficiency.

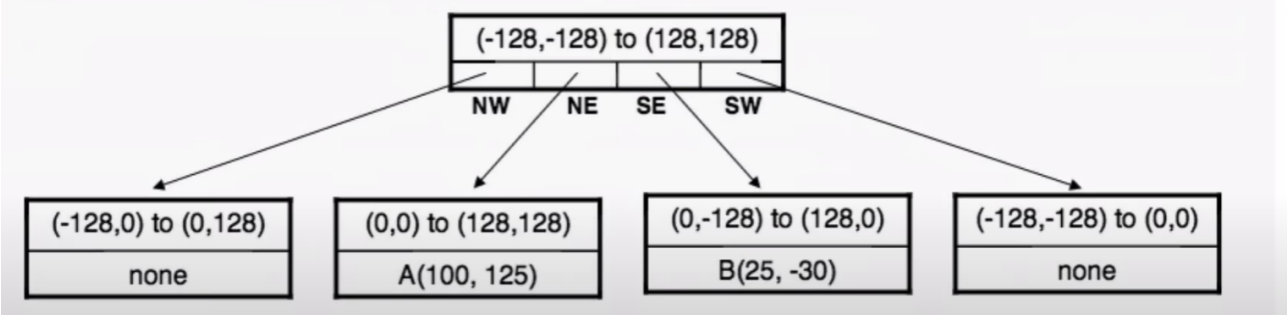
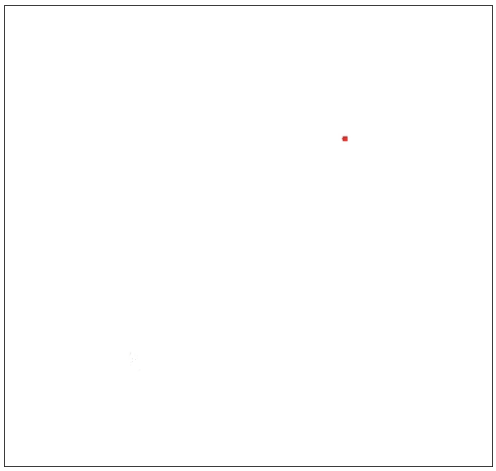
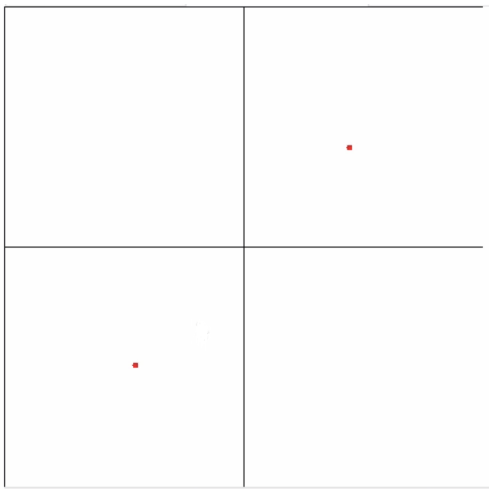


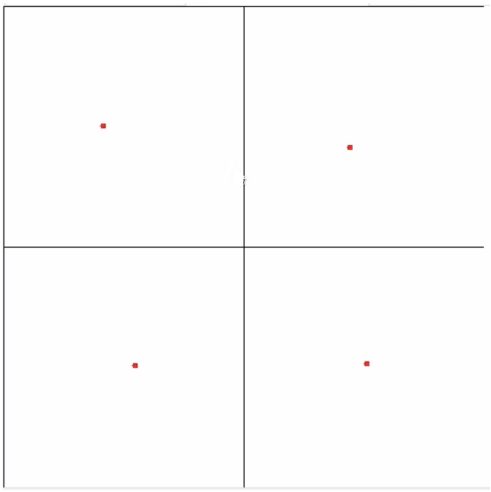
Figure 10: How `QuadTree` works. Pointer for 4 sections. Internal nodes do not store information for data



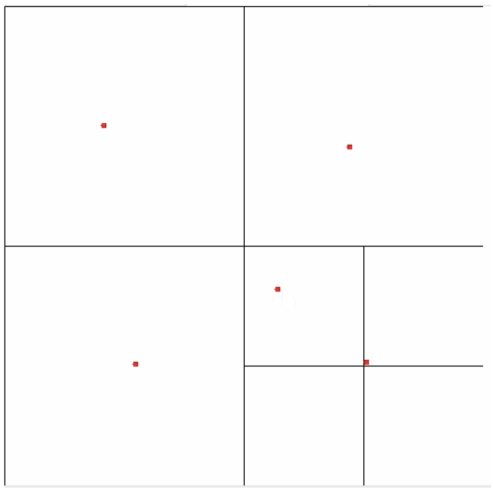
(a) If there is only one point, there is no need to separate the area into places. One node is enough.



(b) If we put another point to another place, `QuadTree` will divide the total area into 4 pieces. The root node will have a pointer to these two points. The other 2 leaf nodes will be empty.

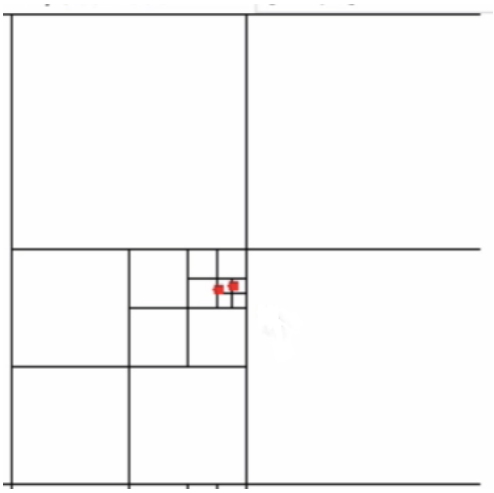


(c) If we keep putting a point into an empty node, it will not scale the tree again



(d) When we put the fifth point properly, the root node will have a pointer to another internal node which contains a pointer to 4 leaf nodes.

Figure 11: `QuadTree` Visualization



(e) Flaw of `QuadTree`. If we put two points too close to each other, it may cause unnecessary creation of internal nodes while only 2 of them have pointers to data available leaf nodes.

Figure 11: `QuadTree` Visualization

Decoupling simulation step size from infection rate: In the current implementation, the infection rate depends on the simulation step size. However, this is not desirable as a change in simulation step size also changes the infection behaviour. For this reason, it is desirable to decouple them. Although it is very hard to perfectly decouple them because simulation always depends on some degree of time discretisation, we try to decouple them as far as possible. Figure 12 includes the highlighted lines of code that we have added to try and achieve this aim. We added a new attribute `elapsedTime` to check and allow for simulation to happen only when `1` second has passed from the previous simulation. This reduces the effect of `simulation time step` and makes the simulation behaviour more realistic.

```
public void update(final double simTimeInSec) {
    // check the positions of all pedestrians and switch groups to INFECTED (or REMOVED).
    DynamicElementContainer<Pedestrian> c = topography.getPedestrianDynamicElements();

    //checking elapsed time for simulation
    elapsedTime+=simTimeInSec - prevTime;

    // Try to discretize the time to 1 second to reduce the dependency on simTimeInSec
    if(elapsedTime>=1.0){
        elapsedTime = 0;
        prevTime = simTimeInSec;
        if (c.getElements().size() > 0) {
            for(Pedestrian p : c.getElements()) {
                // loop over neighbors and set infected if we are close
                if(getGroup(p).getID() == SIRType.ID_INFECTED.ordinal())
                    continue;
```

Figure 12: Decoupling simulation time step from infection rate

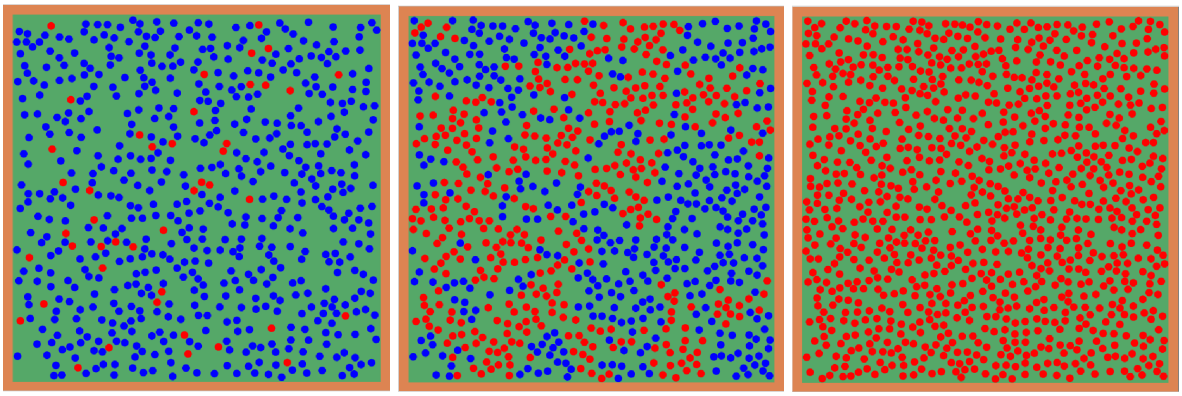
Static experiments: After making all the changes to the code, we ran some experiments with our modified code as mentioned in the exercise sheet:

- **Scenario 1**: A static scenario with 1000 pedestrians randomly distributed in the scenario. For this scenario, we started with the following attributes:
  - `infectionsAtStart`: 0

- `infectionRate`: 0.05
- `infectionsMaxDistance`: 1.0

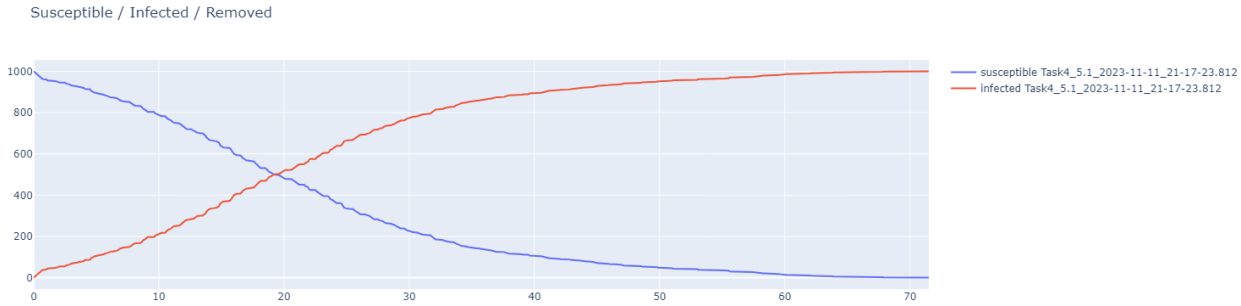
Figure 13 shows the simulation progress from start to end with pedestrians slowly getting infected over time until all the pedestrians are infected at the end. We used the *Dash/Plotly visualization* tool to visualize the simulation results. Figure 14a shows the graph of pedestrians getting infected where half of the pedestrians got infected in around `19` seconds.

- **Scenario 2**: Now we have to double the `infectionRate` from that of the previous experiment. So we set the `infectionRate` to `0.1` while keeping the other attributes the same and run the experiment again. This time, for half of the pedestrians to get infected it took around `7` seconds which is less than half of the time it took in the first setting. Figure 14b shows the graph with both settings. We can easily see that it took way less time for pedestrians to get infected when the `infectionRate` was increased.

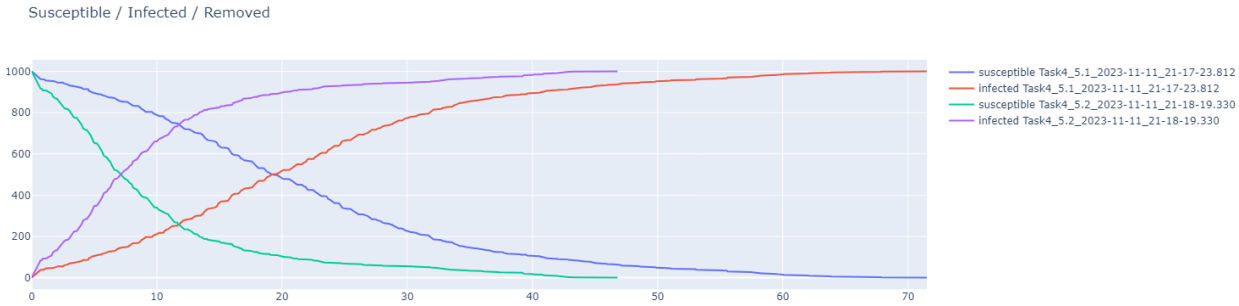


(a) start (b) middle (c) End

Figure 13: Simulation Progress



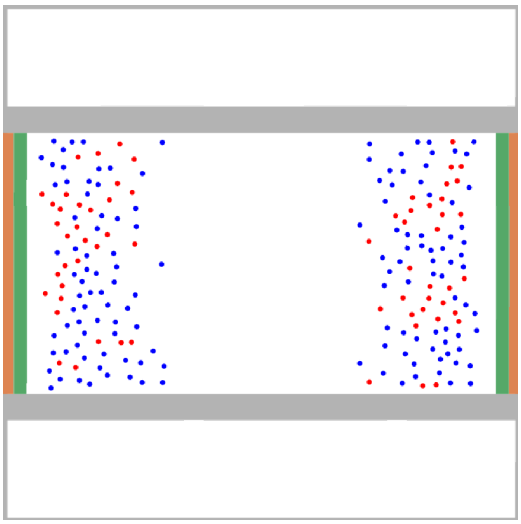
(a) Infection Rate: 0.05



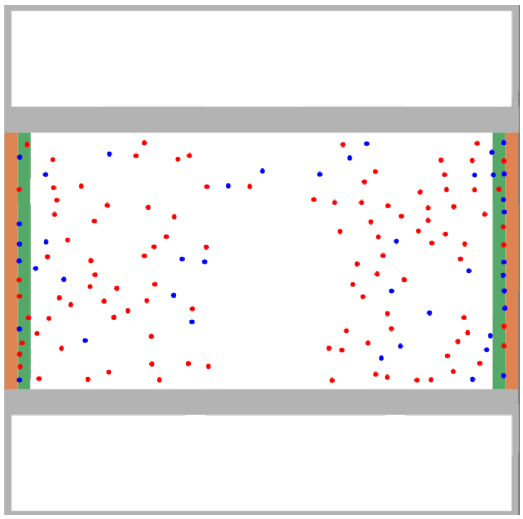
(b) Comparison between infection rates 0.05 and 0.1

Figure 14: Visualization of different infection rates

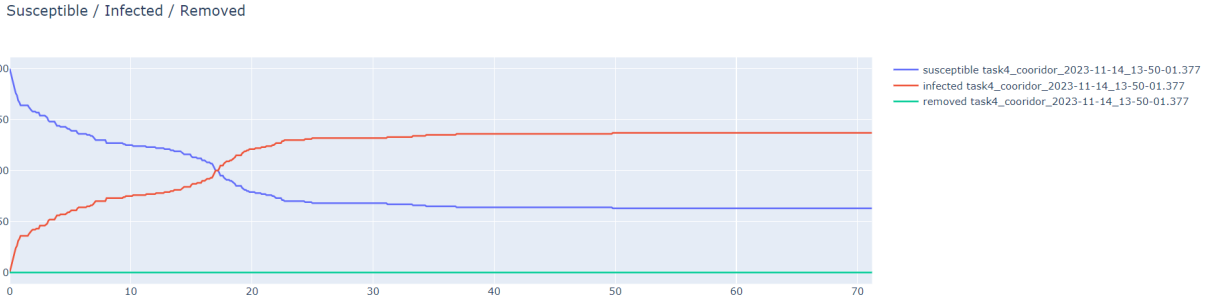
Corridor Scenario: We implement a 40m x 20m corridor scenario where 100 pedestrians move from right to left and another 100 move from left to right. With an infection rate of 0.1, it can be seen from Figure 15c that not all pedestrians get infected. Out of 200 people, only **137** of them are infected, and the acceleration of infection decreases significantly after the contact of the different directioned groups ends. If we increase the number of pedestrians, length of the corridor, or `infectionRate`, it can be observed that the number of infections will increase.



(a) Pedestrians in different directions approach each other. Some of them are infected



(b) Distribution of susceptible and infected pedestrians after collision ends



(c) Pedestrian Groups (Infected & Susceptible) / Time Graph. At the beginning and at the contact time between different directioned pedestrians, acceleration of infection is high compared to other points.

Figure 15: Corridor Scenario of 40m × 20m

Possible Extensions:
In order to make our model more realistic, here are some suggested extensions that can be added:

- ***Age Groups:*** Every demography has people of different age groups with different immunity strengths. For example, younger people tend to have better immunity and faster recovery rates than older people and hence they are less susceptible to getting infected. Adding the age parameter to `Pedestrian` and giving different `infectionRate` and `recoveryRate` would make our model more robust and realistic.
- ***Adding vaccinated groups:*** Another nice extension to our SIR model would be to add a certain percentage of the population to be `vaccinated` by adding this attribute to the `Pedestrian`. The vaccinated groups would be significantly less susceptible to getting infected or having a faster or greater chance of recovery in case they are infected.
- ***Incubation period for exposed individuals:*** Adding another group of `exposed` individuals can also be beneficial regarding realism. Exposed individuals who have been infected do not start to show the symptoms right away and have lower chances of infecting others. Adding the `exposed` group with an `incubationPeriod` time which is the time after which they actually get sick and start showing symptoms can make our model more realistic.

Report on task 5, Analysis and visualization of results

In order for us to add the recovered class and visualize them properly, we had to change the following files:

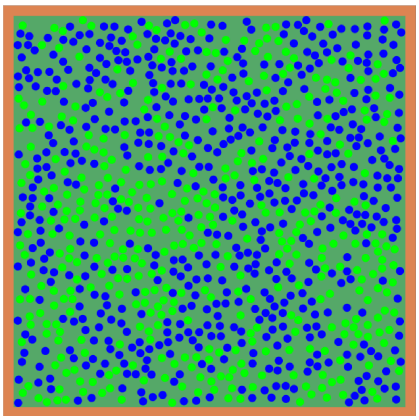
- **AttributesSIRG:**
  - New method `getRecoveredfixedRate()` which returns the new value of `recoveredfixedRate` set arbitrarily at 0.2 (the probability for an INFECTED individual to become recovered/removed).
- **SIRType:**
  - Added the type `ID_REMOVED` for the removed group.
- **SIRGroupModel:**
  - To include the removed class, we had to change the method `getFreeGroupId()` to include the new group of recovered individuals. Figure 16 shows the change in the function. It was a simple fix to include the recovered class with a similar logic as the infected group. We also had to change the `Update function` to add infected individuals to the recovered class based on a certain probability.
- **utils.py:**
  - We changed the function `create_folder_data_scatter(folder)` in the provided SIRVisualization utils file (`DASH/plotly` visualizer) to allow us to visualize the recovered class.

To test our implementation we conducted multiple experiments on our modified software. The first test we conduct is the same as in figure 13 from task 4, with the only difference that pedestrians now can recover after being infected. The end frame for task 5 can therefore look like the ones in figure 17, with different distributions of blue and green dots for susceptible and recovered pedestrians.
Figure 18 shows a comparison of different infection and recovery rates. In subfigure 18a, a low infection rate (0.05) and a high recovery rate (0.2) are used. Only 202 pedestrians become infected over time and a constant value is established after 70 time steps. In subfigure 18b, the infection rate is doubled (0.1), while the recovery rate stays the same (0.2). In comparison, a drastic difference can be seen: the susceptible and removed lines almost close the gap in between with a total number of 473 infected and later removed pedestrians after 60 time steps. Subfigure 18c changes the recovery parameter to 0.1, and the infection parameter stays at 0.1. Here, the initial spike in infections is much higher, and a total of 790 infections after 86 time steps. Finally, subfigure 18d is created with an infection rate of 0.05 and a recovery rate of 0.1. It looks the closest to the second parameter settings, however, the lines intersect, and the infections reach a total of 630 after 103 time steps.

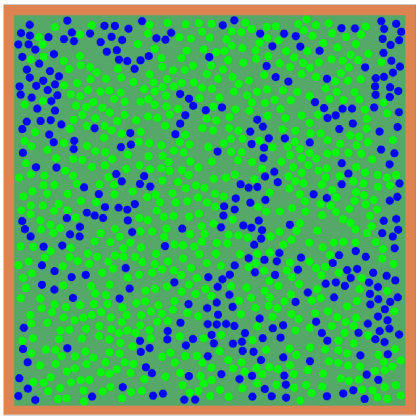
(a) getFreeGroupId() changes in SIRGroupModel

(b) Changes in update method of SIRGroupModel

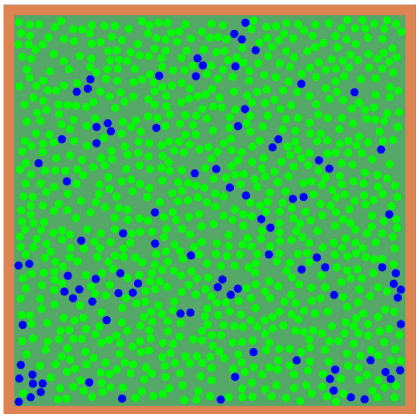
Figure 16: Code changes in SIRGroupModel to include recovered class.



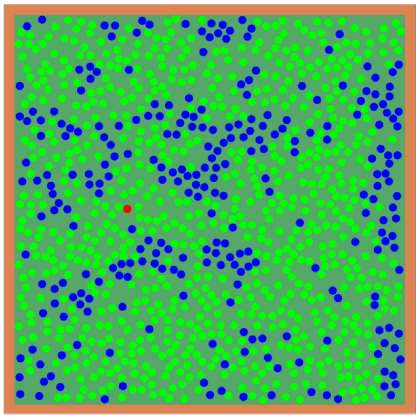
(a) I: 0.05 - R: 0.2



(b) I: 0.1 - R: 0.2

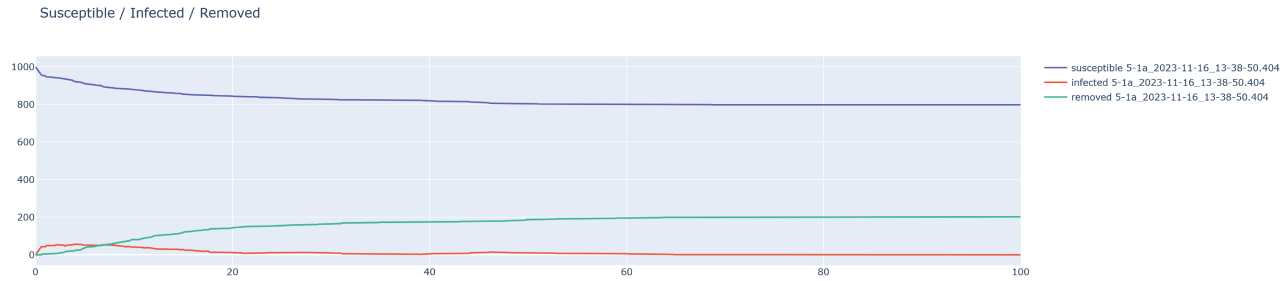


(c) I: 0.1 - R: 0.1

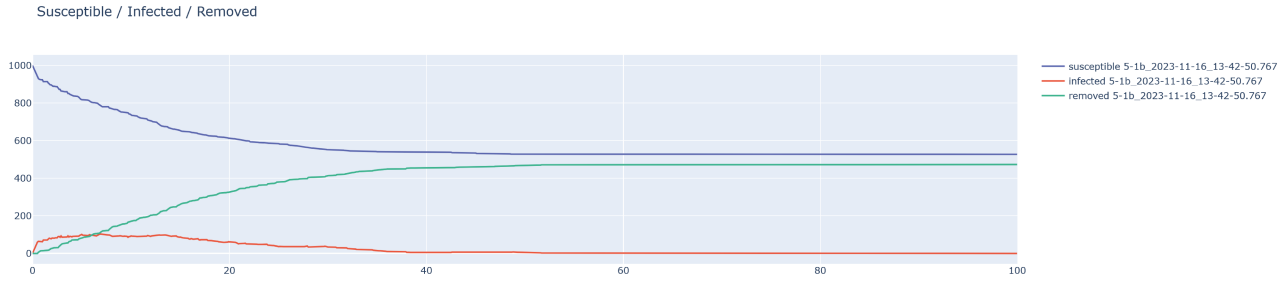


(d) I: 0.05 - R: 0.1

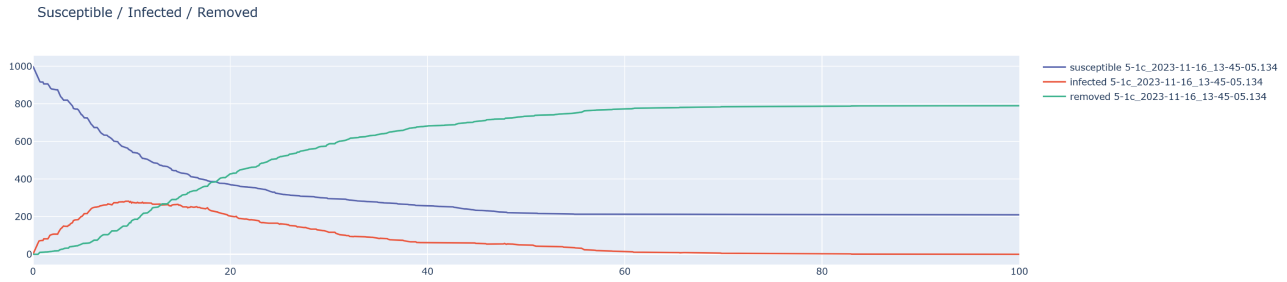
Figure 17: Final simulation picture for different infection (I) and recovery (R) rates



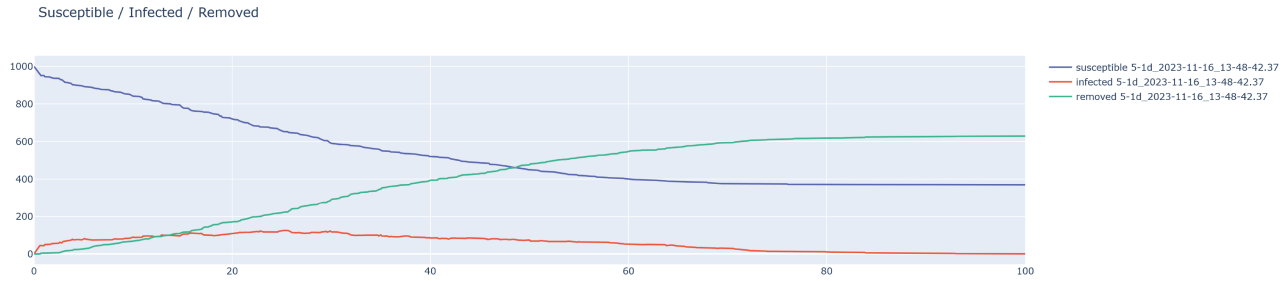
(a) I: 0.05 - R: 0.2



(b) I: 0.1 - R: 0.2



(c) I: 0.1 - R: 0.1



(d) I: 0.05 - R: 0.1

Figure 18: Comparison of different infection (I) and recovery (R) rates

Supermarket simulation

To create a fairly realistic scenario of a supermarket, we took inspiration from the supermarket model available on smartdraw. To comply with the requirement of having a supermarket of at least 30x30m, however, all dimensions have been roughly doubled compared to the previous model, which may affect the realism of the simulation.

(a) Supermarket model

(b) Pedestrians enter the supermarket

(c) Pedestrians are walking around the supermarket

(d) Some pedestrians are returning however, some of them are stuck inside of a crowd

Figure 19: Supermarket Scenario for `pedPotentialPersonalSpaceWidth` = 0.5

To get pedestrians to take different paths, 3 sources of 50 pedestrians have been implemented (150 in total for the scenarios), each group having different target orders (routings) inside the supermarket before leaving. For all the scenarios, the following parameters are fixed and set:

- `infectionsAtStart` : 10
- `infectionRate` : 0.1
- `infectionMaxDistance` : 1.0
- `recoveredfixedRate` : 0.05

To see what happens when increasing the `pedPotentialPersonalSpaceWidth`, 4 tests have been implemented with different values for this parameter: 0.5, 1.5, 2.5, and 3.5.

Regarding the paths of the pedestrians, we notice the following during the scenarios: the lower the value of `pedPotentialPersonalSpaceWidth` is, the more they get stuck in the corridors as different pedestrians headed in different directions hinder each other's motion. Some pedestrians even cannot complete their objective in a

time frame of 500 seconds.

Regarding the spread of the infection, referring to the SIR visualisations, both 0.5 (see figure 21a) and 1.5 (see figure 21b) meters produce a high initial spike in infections, in which it subsides faster for a value of 1.5. While there is a difference in the total number of infections, it is a rather small amount. Adding another meter to achieve a `pedPotentialPersonalSpaceWidth` of 2.5 results in a drastic difference, however, as can be seen in figure 21c. The initial infection spike is less than 50, and the line of susceptible and removed pedestrians does not intersect, i.e. less than half of the supermarket visitors become infected. Figure 21d shows an interesting result: The graph does not show even fewer infections like it was to be expected. Despite the efforts to increase the distance between pedestrians, the limited space, particularly at the entrance, leads to crowded conditions, facilitating the transmission of infections.

To reduce the number of infections, we want to introduce several ideas: First of all, only a specified amount of pedestrians should be allowed inside a supermarket depending on its size. We made calculations for what could be the best number of people for `pedPotentialPersonalSpaceWidth` = 3.5. It can be seen from figure 20, that when **119** people were allowed to enter the supermarket, less than 50% of the people got infected due to a smaller crowd at the entrance.

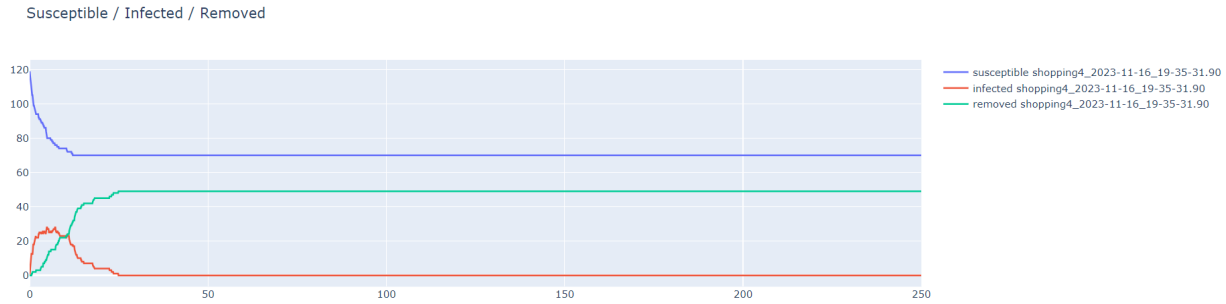
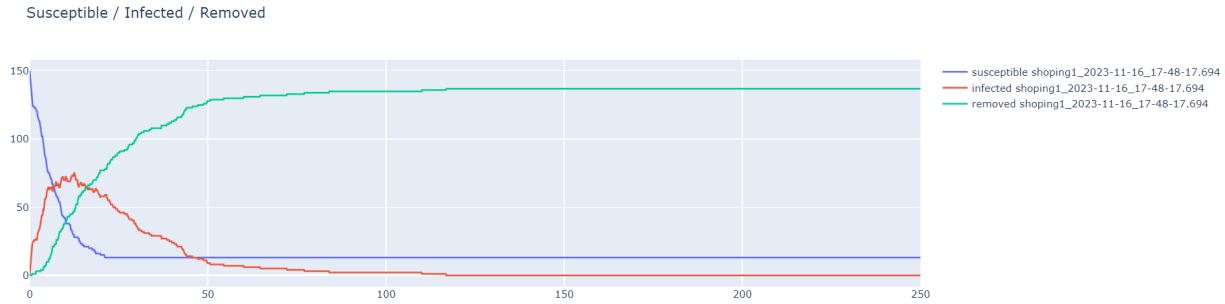


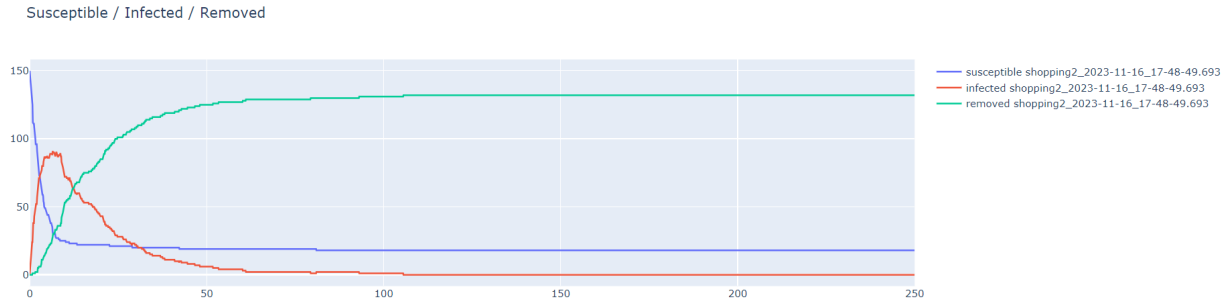
Figure 20: 119 people are allowed to enter the supermarket

While waiting outside, the people need to retain distance as well. This can be realized using drawn-on lines in the parking lot as well as using shopping carts for more flexible queues. Both of these methods have been used during the height of the COVID-19 pandemic. [2, 1]

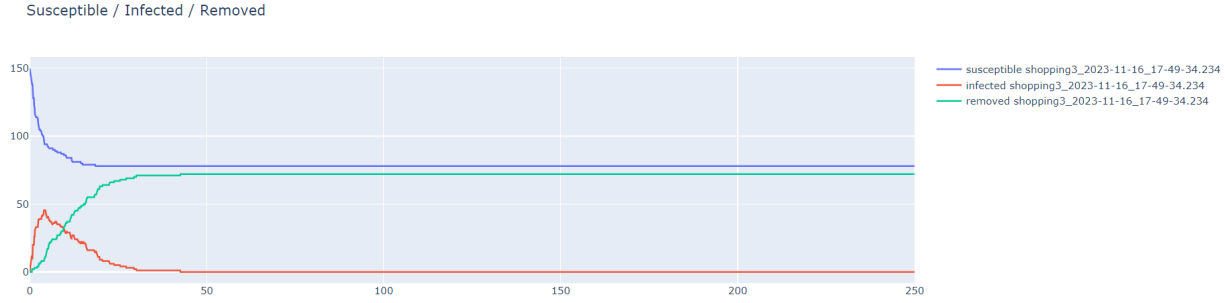
Additionally, specific routes can be marked inside the supermarket to achieve 'one-way streets' to avoid the need to close the gap between pedestrians even for a short amount of time.



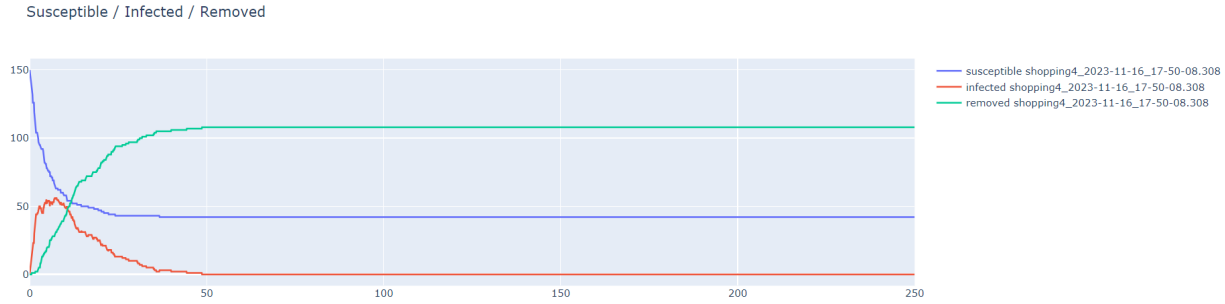
(a) `pedPotentialPersonalSpaceWidth` = 0.5



(b) `pedPotentialPersonalSpaceWidth` = 1.5



(c) `pedPotentialPersonalSpaceWidth` = 2.5



(d) `pedPotentialPersonalSpaceWidth` = 3.5

Figure 21: Comparison of different `pedPotentialPersonalSpaceWidth` in market scenario

References

[1] Wired Aarian Marshall. As Cities Reopen, Expect to Wait in Lots of Lines. https://www.wired.com/story/cities-reopen-expect-wait-lines/. Last accessed 16.11.2023.

[2] BBC. Coronavirus: Customers queue for hours as Ikea reopens 19 shops. https://www.bbc.com/news/business-52874615. Last accessed 16.11.2023.

[3] Benedikt Kleinmeier, Benedikt Zönnchen, Marion Gödel, Gerta Köster. Vadere: An Open-Source Simulation Framework to Promote Interdisciplinary Understanding. In: Collective Dynamics, vol. 4, 2019.

[4] Felix Dietrich and Gerta Köster. Gradient navigation model for pedestrian dynamics. *Physical Review E*, 89(6), jun 2014.

[5] Dirk Helbing and Péter Molnár. Social force model for pedestrian dynamics. *Physical Review E*, 51(5):4282–4286, may 1995.

[6] Christina Maria Mayr and Gerta Köster. Social distancing with the optimal steps model, 2022.

[7] https://github.com/pedestrian-dynamics-HM/vadere.