Report for exercise 2 from group J

Tasks addressed:	5
Authors:	Jia Long Ji Qiu (03770323) Jiabo Wang (03770274) Yilun Liu (03761939)
Last compiled:	2022–11–17
Source code:	https://github.com/jialongjq/mlcms-2

The work on tasks was divided in the following way:

Jia Long Ji Qiu (03770323)	Task 1	1/3%
	Task 2	1/3%
	Task 3	1/3%
	Task 4	1/3%
	Task 5	1/3%
Jiabo Wang (03770274)	Task 1	1/3%
	Task 2	1/3%
	Task 3	1/3%
	Task 4	1/3%
	Task 5	1/3%
Yilun Liu (03761939)	Task 1	1/3%
	Task 2	1/3%
	Task 3	1/3%
	Task 4	1/3%
	Task 5	1/3%

Report on task 1, Setting up the Vadere environment

General overview of Vadere

Making first a general comparison between Vadere and the cellular automaton of the first exercise, there are various aspects that should be highlighted.

First of all, the interface of Vadere offers a wide variety of options that are very intuitive and makes the set up of any scenario much easier. The visualisation is similar to the one of the cellular automaton, since it consists of cells as well, with the plus of the elements being able to occupy larger sizes.

Moreover, when it comes to the visualisation of a simulation, it is really convenient that the results are saved as an output, so the user can check them again at any time. The same goes for the progress bar, which allows the visualisation frame by frame of the results.

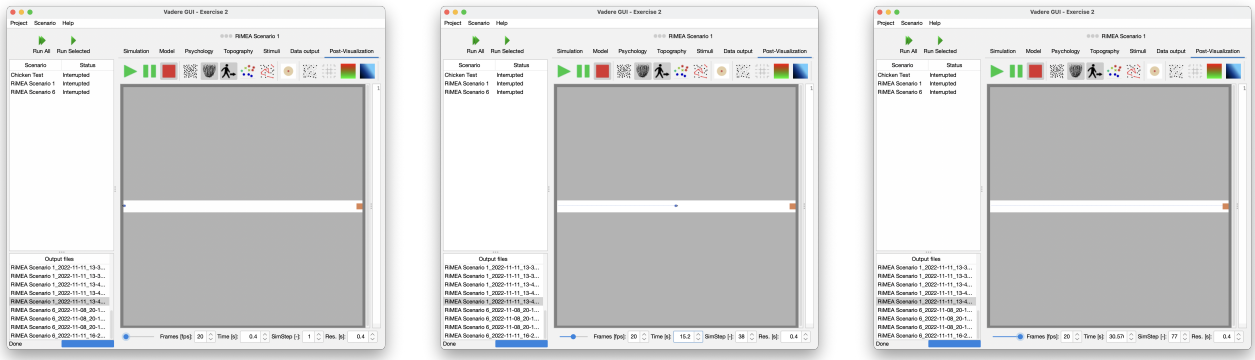
One last feature to highlight would be the file system. Vadere allows both loading and saving files for different purposes, which can be for example loading or saving a whole project, add predefined scenarios, changing parameters of a simulation such as the model, the psychology, the topology...

RiMEA scenario 1 (straight line)

It should be recalled that the RiMEA scenario 1 defined a 2 m wide and 40 m long corridor, with a pedestrian in one side and its target in the other one, with the purpose of proving that a pedestrian with a defined walking speed will cover the distance in the corresponding time period.

This scenario has been set up with the Vadere environment as shown in Figure 1a. For proper visualisation purposes, both width and height have been set to 40 m, and two obstacles of dimensions 40x19 have been placed to leave a 2 m wide space in the middle. The pedestrian was defined with a free flow speed of 1.33 m/s, so the travel time should lie in the range of 26 to 34 seconds. The pre-movement time is ignored since the pedestrian has its target assigned manually before running the simulation. The standard template Optimal Steps Model has been set for this execution, as well as for the rest of the scenarios of this task (RiMEA scenario 6 and chicken test).

As expected, by looking at the results of the simulation (Figure 1c), the travel time was around 30 seconds. Compared with the results of the cellular automaton created in exercise 1, the behaviour of the pedestrian is roughly the same (as it should be), since the only thing the pedestrian has to do is to walk in a straight line.



(a) At the beginning of the simulation (b) In the middle of the simulation (c) At the end of the simulation

Figure 1: Different stages of the simulation of the scenario 1

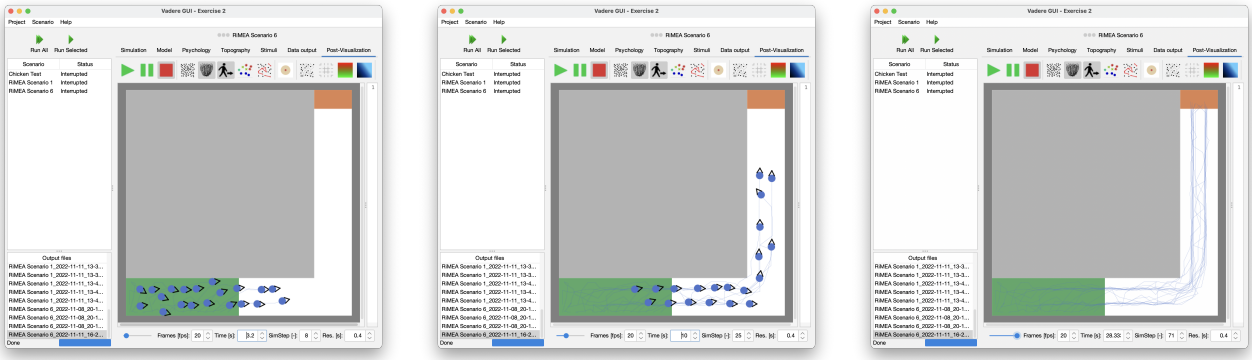
RiMEA scenario 6 (corner)

The RiMEA scenario 6 describes a group of 20 pedestrians moving towards a corner which turns to the left, in order to show that they can go around it without passing through walls.

This scenario has been created with the Vadere environment as shown in Figure 2a. As required in the statement, both height and width of the scenario have been set to 12 m. On the one hand, the obstacle has been placed in the top left part of the scenario and has dimensions of 10x10 m. On the other hand, the corridor has a width of 2 m and both horizontal and vertical part has 12 m of length. Finally, the 20 pedestrians are uniformly distributed in the first 6 m of the horizontal part of the corridor (in the green part) as demanded in the statement.

As expected, by looking the scenario at the end of the simulation (Figure 2c), all the 20 pedestrians reached the target without passing through walls.

Finally, comparing the results with those obtained in the cellular automaton of exercise 1, the pedestrians have roughly the same behaviour, since the model that has been set in this execution is the Optimal Steps Model (OSM), the pedestrians will always choose the optimal step that brings them to the target taking into account the obstacles.



(a) At the beginning of the simulation (b) In the middle of the simulation (c) At the end of the simulation

Figure 2: Different stages of the simulation of the scenario 6

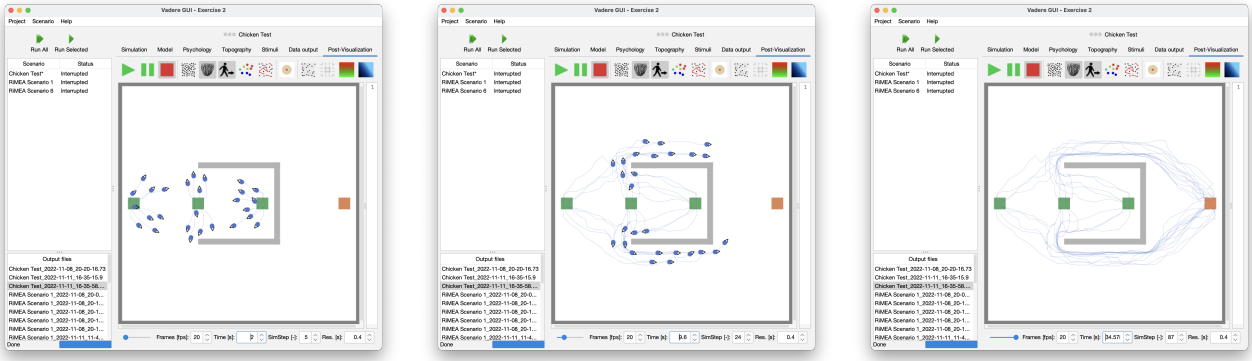
Chicken test

The chicken test consists in a scenario with pedestrians in one side and their target in the other, with a U-shaped obstacle in the middle obstructing the straight path between them. The point of this simulation is testing whether the pedestrians can avoid the obstacles and reach the target or they get stuck at some point and cannot reach any target.

This scenario has been created with the Vadere environment as shown in Figure 3a. And the scenario is composed by a set of obstacles forming a vertical U shape and the pedestrians placed in 3 different places: outside, at the beginning and inside of the U form obstacles. The target is placed at the same level of the pedestrians, so in the case there weren't obstacles, the optimal path for the pedestrians to reach the target is a straight line, but since there are obstacles, pedestrians cannot follow the straight line path so they have to flip the U form obstacles and then reach the target.

As it can be seen in the Figure 11a, all the pedestrians from the different places have reached to the target turning around the U-shaped obstacle.

Comparing the results with those obtained in the cellular automaton of exercise 1, the pedestrians have roughly the same behaviour if the algorithm chosen in the exercise 1 is Dijkstra or the Fast Marching Method. If the basic algorithm is chosen in the exercise 1, the pedestrians does not perform in the same way, since in the basic algorithm is not implemented the obstacle avoidance, and therefore the pedestrians get stuck in the U-shaped obstacle.



(a) At the beginning of the simulation (b) In the middle of the simulation (c) At the end of the simulation

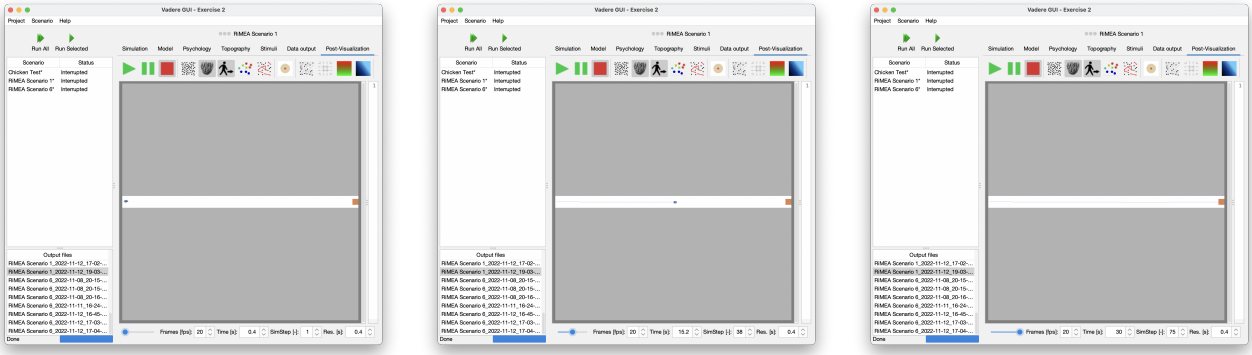
Figure 3: Different stages of the simulation of the chicken test

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

In this task, the three scenarios from the previous task will be re-run with other two different models: the Social Force Model and the Gradient Navigation Model. Both models will be compared to the Optimal Steps Model, the one used in the previous task, and the differences and similarities observed during the simulations will be analysed and reported in the following subsections. At the end, a joint analysis will be done for all three models.

Social Force Model (SFM)

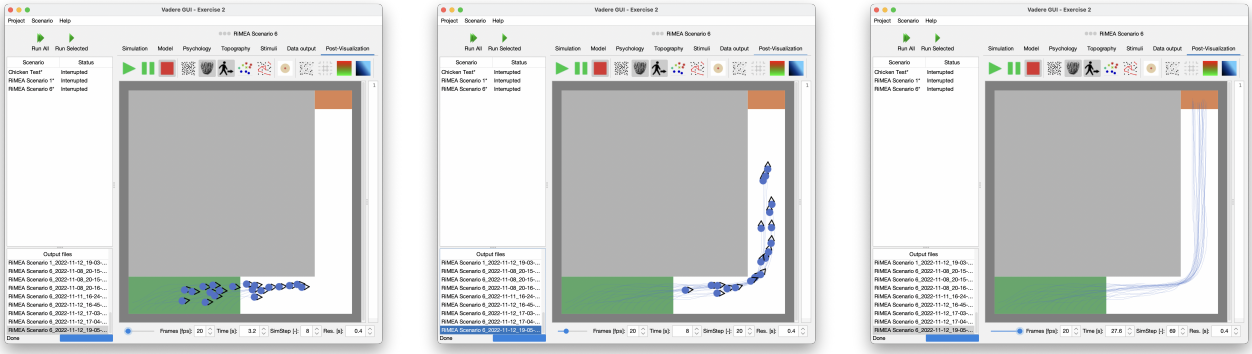
(a) RiMEA scenario 1: in this first case, with Social Force Model the pedestrian does not follow a straight line path anymore, instead, as we can see in the Figure 4c, it moves downwards 3 positions in the first half of the simulation and later it goes up 1 position before reaching the target. With Optimal Step Model, the pedestrian follows a straight line path to reach the target. The travel time in both simulations are nearly the same (30 s with SFM and 30,57 s with OSM). As they behave nearly in the same way and it is a very simple scenario, it is hard to know why the SFM does not follow a strictly straight line.



(a) At the beginning of the simulation (b) In the middle of the simulation (c) At the end of the simulation

Figure 4: Different stages of the simulation of the scenario 1 using the Social Force Model

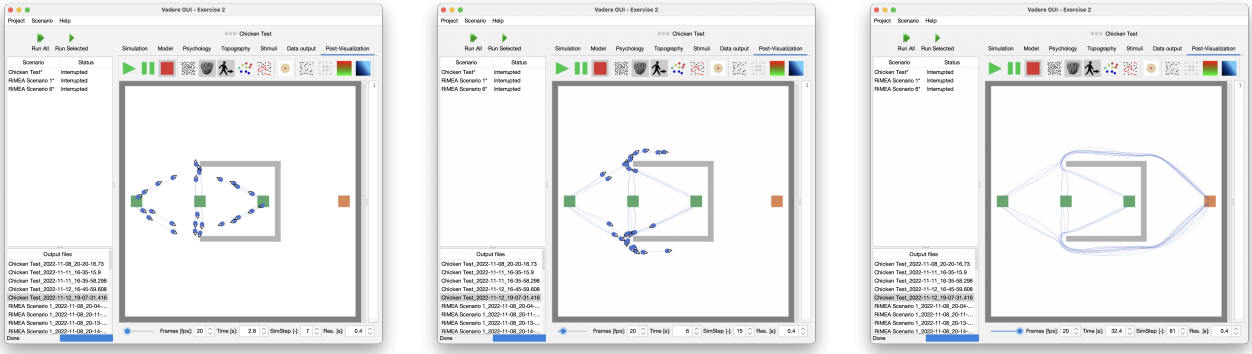
(b) RiMEA scenario 6: with the Social Force Model, the behavior of the pedestrians are more “ordered” and natural. All the pedestrians tend to follow the optimal/shortest path to reach the target. As we can see in the Figure 10b, the paths are more smooths and all of them looks similar to each other and there is a sensation of order, whereas in the Optimal Step Model the paths are a little bit chaotic and pedestrians behaves more “nervous” trying to reach the target: some of them stops or collide with obstacles at certain point when it is not required. The consequence of that order of the pedestrians can be observed rather on the travel time to the target. It is a little bit faster with the SFM (27.6 s) compared to the OSM (28.33 s).



(a) At the beginning of the simulation (b) In the middle of the simulation (c) At the end of the simulation

Figure 5: Different stages of the simulation of the scenario 1 using the Gradient Navigation Model

(c) Chicken test: of course, with the Social Force Model, the pedestrians can also avoid the obstacles and reach the target. As said in the previous case, the behavior of the pedestrians are more direct, ordered and natural in SFM than in OSM. It seems like the pedestrians know from the beginning where they have to go to reach the target, whereas in OSM although all of the pedestrians can avoid the obstacles and reach the target, some of them during the path collide with each other and obstacles and their paths are not as smoothly as the paths of the SFM (Figure **??**, there are some big changes of directions from a pedestrian. The consequences also are reflected in the travel time with SFM - 32.4 s and with OSM - 34.57 s.

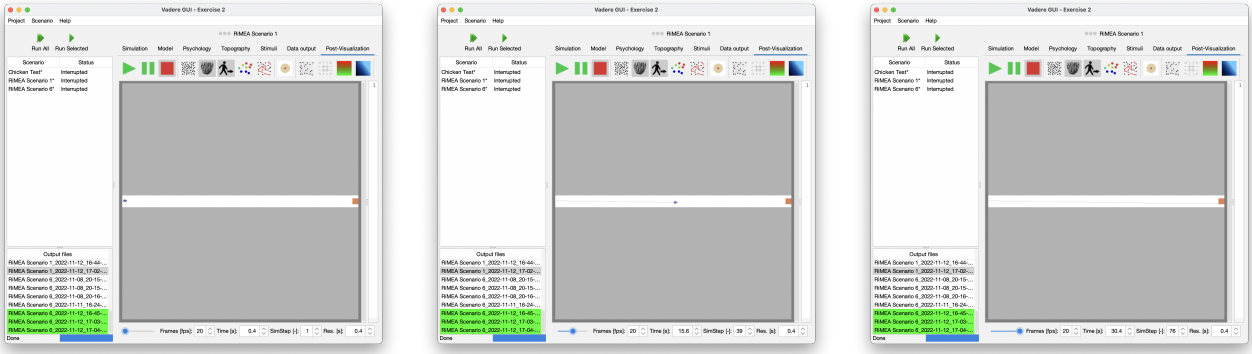


(a) At the beginning of the simulation (b) In the middle of the simulation (c) At the end of the simulation

Figure 6: Different stages of the simulation of the chicken test using the Social Force Model

Gradient Navigation Model (GNM)

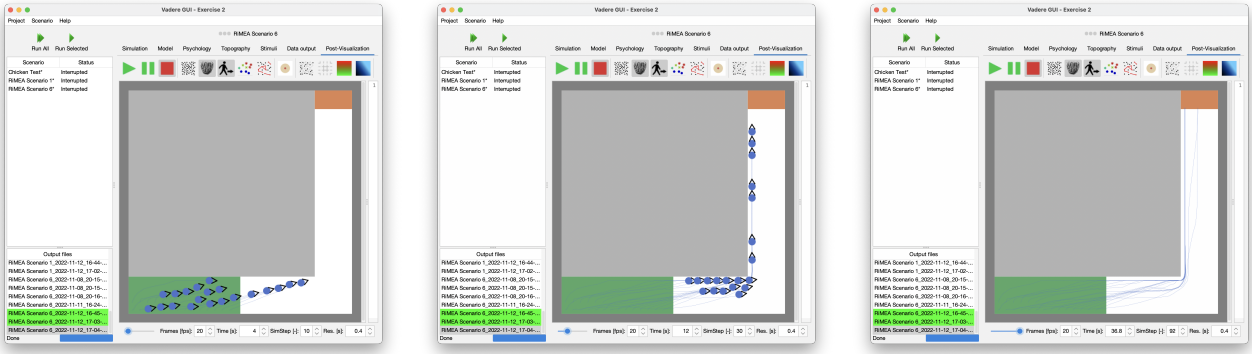
(a) RiMEA scenario 1: the pedestrian does not follow a straight line anymore, instead, it moves downwards until the target is reached. The travel time in both simulations are roughly the same, being the simulation using GNM slightly faster, but negligible. This new behaviour is due most probably to how a target surface works: by looking at the results using OSM (Figure 1), it can be observed that the pedestrian turns up right before reaching the target, instead of keeping a completely straight path, so it is possible that the target itself can be reached through different points inside it. By taking this into account, for this case the pedestrian might have selected a lower point in the target, and therefore there might have been some anticipation while selecting the path. However, since not much information is given, this conclusion cannot be ensured.



(a) At the beginning of the simulation (b) In the middle of the simulation (c) At the end of the simulation

Figure 7: Different stages of the simulation of the scenario 1 using the Gradient Navigation Model

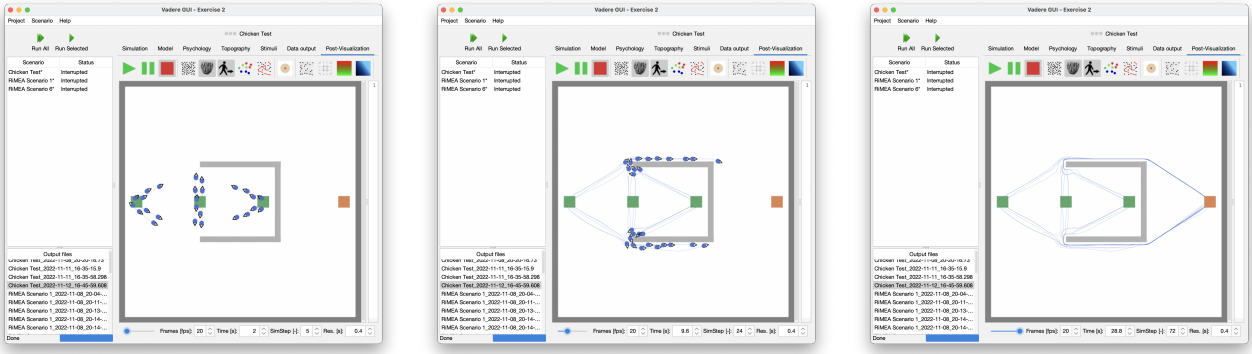
(b) RiMEA scenario 6: for this scenario, except for two pedestrians, all of them ended up doing the same uniform path as the one who reached the target first, without passing through walls. All pedestrians seemed to be more organised, as if they already knew the path they should follow beforehand. Furthermore, whenever a pedestrian had any other pedestrian obstructing their path, their behaviour consisted in reducing their speeds and waiting until they were able to continue (with exceptions, as commented before), rather than taking another path, as the pedestrians in the simulation using OSM did (Figure 2). Due to this behaviour, the travel time increased about 8 seconds.



(a) At the beginning of the simulation (b) In the middle of the simulation (c) At the end of the simulation

Figure 8: Different stages of the simulation of the scenario 6 using the Gradient Navigation Model

(c) Chicken test: in the chicken test, all pedestrians were able to turn around the U-shaped obstacle, by taking uniform paths as well. Their behaviour were similar to the described in the scenario 6, but this time different yet symmetrical paths were taken, depending on the starting point of each pedestrian inside the source (either slightly upper or lower). There were some exceptions as well, some pedestrians chose to avoid the obstructing pedestrians rather than waiting. This time, however, the travel time was reduced by around 6 seconds, most probably because in the simulation using OSM, the scattering of so many pedestrians in such narrow space for taking other paths caused a blockage, something that does not happen with GNM.

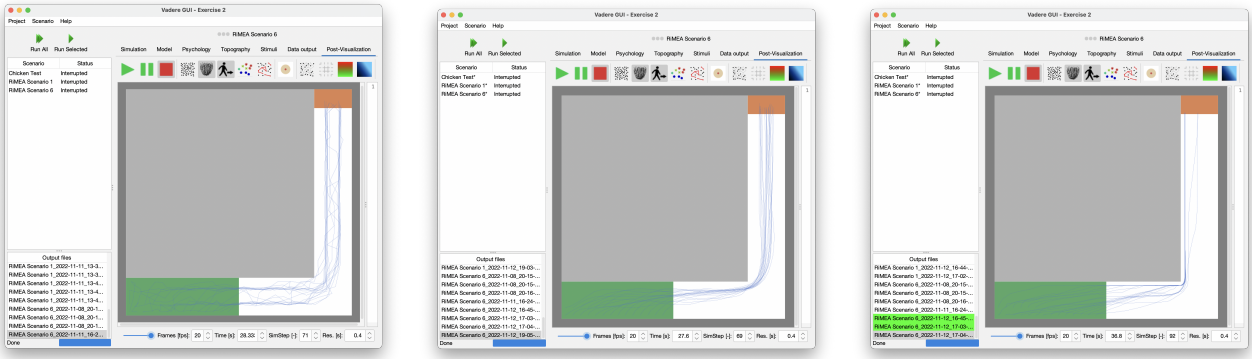


(a) At the beginning of the simulation (b) In the middle of the simulation (c) At the end of the simulation

Figure 9: Different stages of the simulation of the chicken test using the Gradient Navigation Model

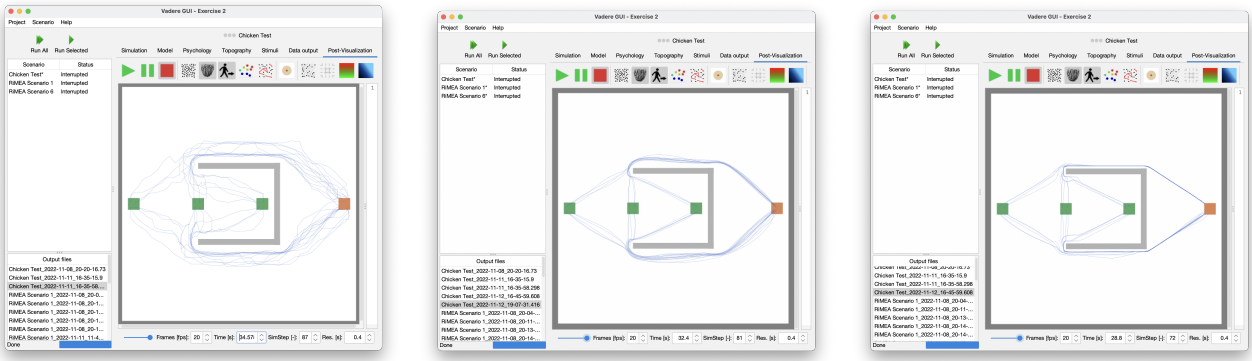
Joint analysis

After looking through the results of the simulation of the previous three scenarios using the different models, it can be concluded that the change of models affected specially the way the paths were distributed. The directions taken by the pedestrians in each scenario were similar, but the main difference lain in the level of strictness in terms of "path bifurcation" they had to follow while traversing it, meaning that the more strict the model, the less the pedestrians were allowed to take other paths in order to avoid other pedestrians. The OSM is the less strict, whereas the GNM is the most, as can be noticed in the comparisons in Figures 10 and 11.



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

Figure 10: Comparison of the scenario 6 between the 3 models



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

Figure 11: Comparison of the chicken test between the 3 models

Report on task 3, Using the console interface from Vadere

Running Vadere from command line

The first job asked in this task is to run the corner scenario (Scenario 6) by calling Vadere from the command line. The command used to run it is the following:

java -jar vadere-console.jar scenario-run

–scenario-file "/path/to/the/file/scenariofilename.scenario"

–output-dir="/path/to/output/folders"

The output files produced by running the scenario from the console were the same as the ones produced by running it in the graphical user interface. This has been checked by using the command *diff* with the corresponding files to compare as inputs (Figure 14).

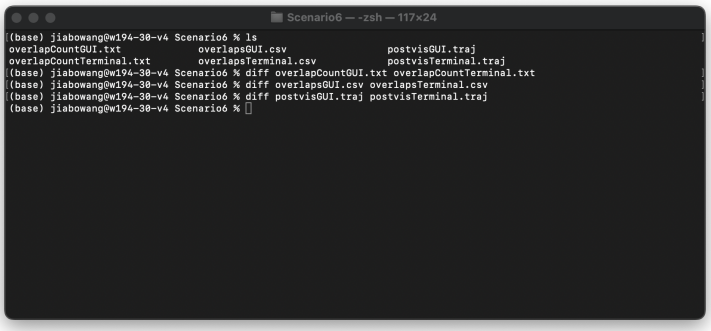


Figure 12: Command line of making diff with the output files

Adding pedestrians programmatically

Knowing that the scenario file represents a JSON file, in order to add pedestrians programmatically, the script *add_pedestrian.py* has been implemented. This script reads the scenario file, modifies the data by adding a pedestrian and saves it in JSON format. It has the following methods:

- **read_scenario(path)**: this method receives as a parameter the path where the scenario file is saved. The scenario is a JSON file, where the data is stored in a dictionary. The function of this method is to open the file, read it and return the data stored in a dictionary.
- **add_pedestrian(data)**: this is the main method of the script. It takes as a parameter the data containing the information of the scenario and all it has to do is to add a pedestrian with all its attributes in the position (11.5, 1.5) as shown in the Figure 13 of the scenario and return the data.
- **save_scenario(path, data)**: finally, this method saves the modified data in the path of the parameter.

Both input path and output path of the scenarios files are hard-coded and takes the "corner scenario" (scenario 6) as demanded in the statement.

After executing this newly modified scenario file using both the console and the GUI, a comparison between the respective outputs has been done as in the previous section, and no differences have been observed.

The added pedestrian is located nearer to the target than the ones from the source, for this reason, it takes him shorter to reach the target (12.8 s). The fastest one from the source needs 14.4 s and the last one reaches after 29.76 s.

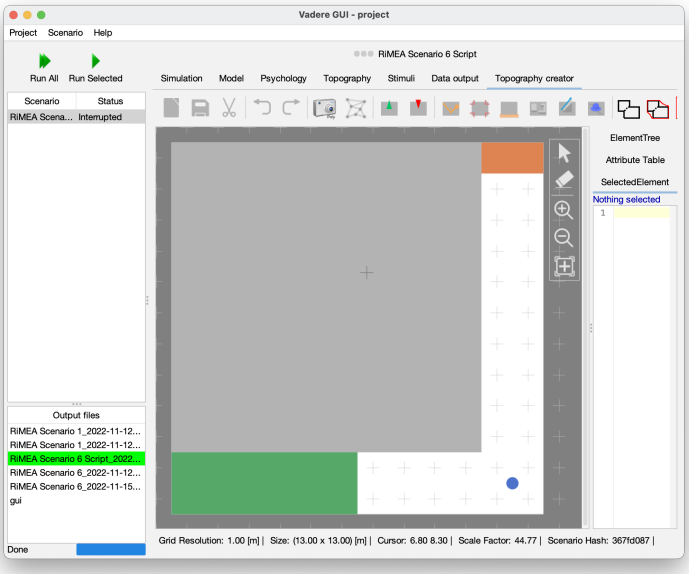


Figure 13: Scenario 6 with a new added pedestrian in the corner

Report on task 4, Integrating a new model

Integration of the SIR Model into Vadere

The integration of SIR model into Vadere at the beginning can be achieved by placing and replacing all the provided file within SIRGroupModel.zip into the appropriate package folders as the belonging location being indicated by the first line of each file.

- The files SIRGroup.java, SIRGroupModel.java and SIRType.java within folder "sir" should be moved into a newly created folder at the directory VadereSimulator/src/org/vadere/simulator/models/groups/sir;
- AbstractGroupModel.java moved into VadereSimulator/src/org/vadere/simulator/models/groups;
- AttributesSIRG.java moved into VadereState/src/org/vadere/state/attributes/models;
- FootStepGroupIDProcessor.java moved into VadereSimulator/src/org/vadere/simulator/projects/ dataprocessing/processor.

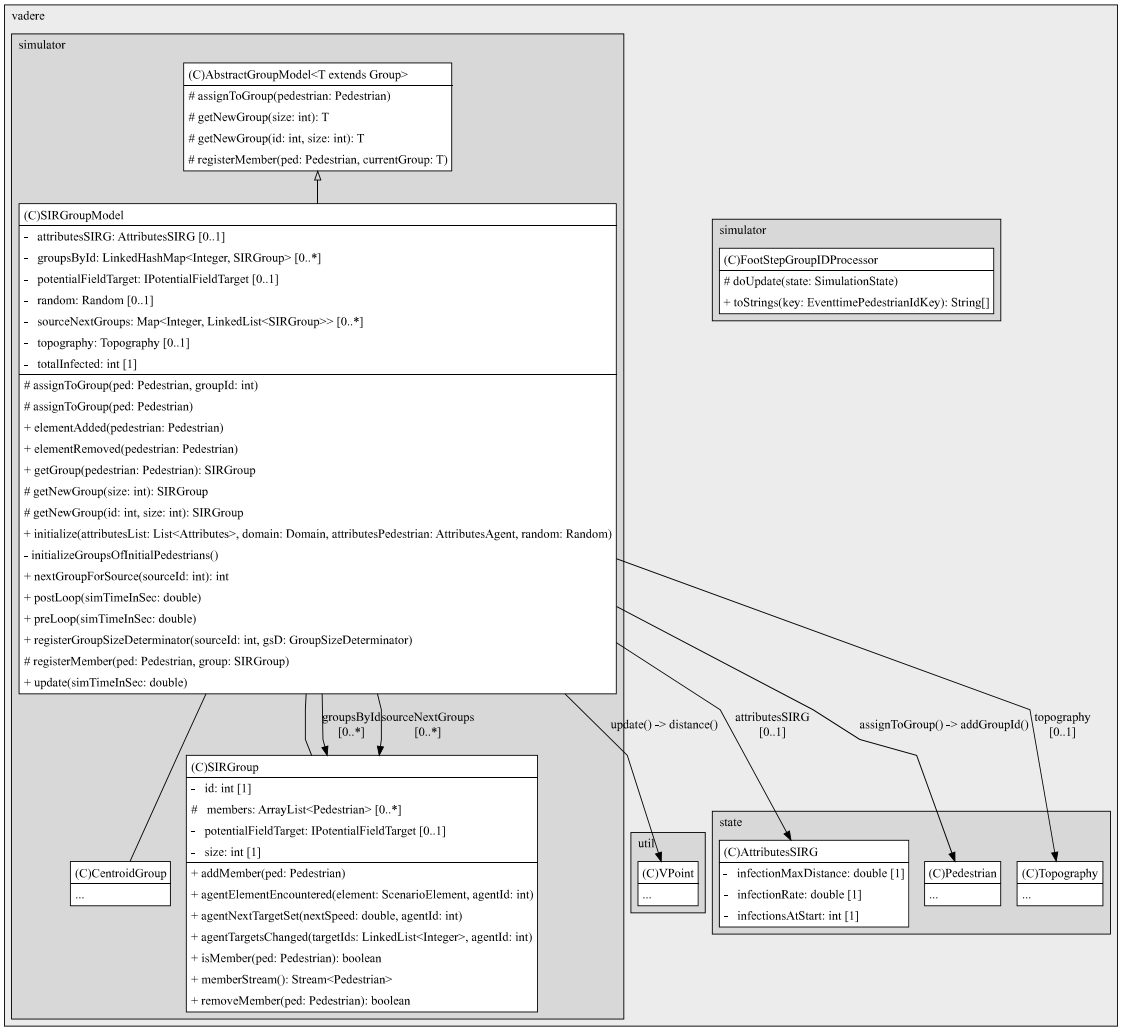


Figure 14: A concise UML class diagram showing the structure of SIR model and related classes

A more detailed explanation of each class follows below:

– SIRGroup: this class represents a list of pedestrians which can be identified with an id.

- SIRGroupModel: this class extends the AbstractGroupModel class and defines the behaviour of a set of SIR groups (which are susceptible or infected in the first instance). It manages the infection process through the method *update()*, which consists in iterating through all pedestrians in the simulation and, for each infected neighbor pedestrian within the specified infection distance, a draw is made and if the pedestrian is unlucky enough, it will get infected. Infected pedestrians are moved from the susceptible to the infected SIR group as a consequence.
- SIRType: this enummeration class represents the possible states of a pedestrian in a SIR Model: infected, susceptible or removed.
- AbstractGroupModel: this abstract class implements the **GroupModel** class.
  - **registerMember(Pedestrian ped, T currentGroup)**: function to register the Pedestrian ped to the T currentGroup, it does not check whether the pedestrian is already a member of another group, so the caller has to make sure of that.
  - **getNewGroup(int size)**: function to get a new group just by specifying the size of that group.
  - **getNewGroup(final int id, final int size)**: function to get a new group with the id and the size specified by the parameters.
  - **assignToGroup(Pedestrian pedestrian)**: function to assign a the Pedestrian pedestrian to a group.
- AttributesSIRG: this class represents the different parameters of the infection behavior.
  - *infectionsAtStart*: the initial number of pedestrian infected.
  - *infectionRate*: the probability such as one pedestrian can become infected if he/she is in contact with an infected pedestrian.
  - *infectionsMaxDistance*: the maximum range where a pedestrian can become infected if an infected pedestrian is located inside this range.

  The three methods that this class have are the getters of the parameters described before.
- FootStepGroupIDProcessor: data output processor to maintain the SIR information, that is the infection status. It is important to store this information for a post simulation analysis of the results.
  - **doUpdate(SimulationState state)**: for each foot step of the trajectory it stores a new output composed of the current time, the pedestrian ID and its group ID.

Analysis of the simulations

In order to be able to analyze the simulations, a new processor (FootStepGroupIDProcessor) has been added. Also a new file, SIRinformation.csv, has been added to the output. This file is linked with the FootStepGrouIDProcessor and contains the group information about the SIR model. As explained before in the class FootStepGroupIDProcessor, the data stored is composed by the pedestrian ID, its group ID and the simulation time of that moment.

Correction of the SIR groups colors

The S, I, R groups are not visualized correctly in the current setup and a good and quick visualization of the different groups of the simulation is fundamental when we have to explain and analyze the behavior of them. So, changes have been done in the class SimulationModel.java in order to set the correct colors and be able to corretly visualize these colors. We have defined the following colors:

- **Infected pedestrian**: red (255, 0, 0) RGB color
- **Recovered pedestrian**: green (0, 255, 0) RGB color
- **Susceptible pedestrian**: blue (0, 0, 255) RGB color

Firstly, in the class SimulationModel (Listing 1) we have added as attributes the colors mentioned before.

```
protected final Color infected_color = new Color(255, 0, 0);
protected final Color recovered_color = new Color(0, 255, 0);
protected final Color susceptible_color = new Color(0, 0, 255);
```

Listing 1: new attributes of the SimulationModel class

And then, in the constructor class of the class, we have the default coloring agent and finally adding the colors to the mapping (colorMap) between the group ID and the group coloring as we can see in the Listing 2.

```
public SimulationModel(final T config) {
  super(config);
  this.config = config;
  this.config.setAgentColoring(AgentColoring.GROUP); //added: to visualize corretly the group colors
  this.colorMap = new ConcurrentHashMap<>();
  this.colorMap.put(-1, config.getPedestrianDefaultColor());
  this.colorMap.put(0, infected_color);      //added: color
  this.colorMap.put(1, susceptible_color);   //added: color
  this.colorMap.put(2, recovered_color);     //added: color
  this.random = new Random();
}
```

Listing 2: constructor of the SimulationModel class

Finally, remark that all the changes done before just enables to visualize the groups during simulation, the visualization does not work in the post visualization.

Efficiency improvement of the infection process

As said before, the infection process is managed through the method *update()* of the class *SIRGroupModel*. This method consists in a double for loop: all pedestrians checking for all pedestrians to determine if there are any infected neighbors within the infection radius, which is very inefficient. To make it even worse, all the work done for iterating through the neighbors of a certain pedestrian can end up being useless, since that pedestrian could have already been infected and this condition is not checked until the very end of the inner loop.

As shown in Listing 3, two improvements have been made to this method: instead of checking if a pedestrian belongs to the susceptible group at the very end of the inner loop, check it before going through all its neighbors; and instead of looping through all pedestrians in the inner loop, only loop through those within the radius of infection. This second improvement takes advantage of the *LinkedCellsGrid* class, concretely the *getObjects()* method, which given a position and a radius, loops only through the cells within the given radius and returns a list of objects of the requested class, which is *Pedestrian* in this case.

```
public void update(final double simTimeInSec) {
  // Check the positions of all pedestrians and switch groups to INFECTED (or REMOVED).
  DynamicElementContainer<Pedestrian> c = topography.getPedestrianDynamicElements();
  LinkedCellsGrid<Pedestrian> grid = c.getCellsElements();
  double infectionRadius = attributesSIRG.getInfectionMaxDistance();
  double infectionRate = attributesSIRG.getInfectionRate();
  if (c.getElements().size() > 0) {
    for(Pedestrian p : c.getElements()) {
      // Only check for susceptible pedestrians, skip already infected ones
      SIRGroup g = getGroup(p);
      if (g.getID() == SIRType.ID_INFECTED.ordinal())
        continue;
      // Get neighbors within a radius of infectionRadius from the position of the Pedestrian p
      VPoint position = p.getPosition();
      List<Pedestrian> neighbors = grid.getObjects(position, infectionRadius);
      // Try to infect p with probability infectionRate with as many attempts as infected neighbors
    are there
      for (Pedestrian n : neighbors) {
        if (p == n || getGroup(n).getID() != SIRType.ID_INFECTED.ordinal())
          continue;
        if (this.random.nextDouble() < infectionRate) {
          elementRemoved(p);
          assignToGroup(p, SIRType.ID_INFECTED.ordinal());
        }
      }
    }
  }
}
```

Listing 3: Improved version of the infection process

Testing and results

Once fixed all the problems mentioned before, now it is possible to run some scenarios with the modified Vadere software and analyze some results. In particular, we are going to analyze the behavior of the pedestrians in 2 different scenarios:

- **Scenario 1**: a fairly large static scenario of 50m x 50m with a source spawning 1000 pedestrians, and a target exactly over of the source, which is not absorbing. The 1000 pedestrians are randomly created, by setting spawnAtRandomPositions as true and useFreeSpaceOnly as false. Over the 1000 pedestrians, 10 are infective and 990 are susceptible. The objective of this test is to see how the susceptible and infective pedestrians change over time.
- **Scenario 2**: a dynamic corridor scenario of 40m x 20m, where one group of 100 susceptible pedestrians moves from left to right, and another group of 100 infected pedestrians moving from right to left. The pedestrians are created over time (not all at the same time) so the attribute useFreeSpaceOnly is set as true. The objective of this test is to see how many pedestrians get infected in this counter-flow.

Let's first explain about the first scenario, where two experiments have been done:

- the first one with the following attributes: The infection rate has been set to 0.05 ("infectionRate": 0.05) with 10 infected pedestrians infected at the beginning ("infectionsAtStart": 10) and the maximum infection distance is set to 1.0 ("infectionMaxDistance": 1.0). The simulation will be executed with a simulation time ratio of 1.0 ("realTimeSimTimeRatio" = 1.0) and the time step will be 0.5 ("simTimeStepLength"= 0.5). After simulating this scenario, half of the population takes 26.1 s to become infected.

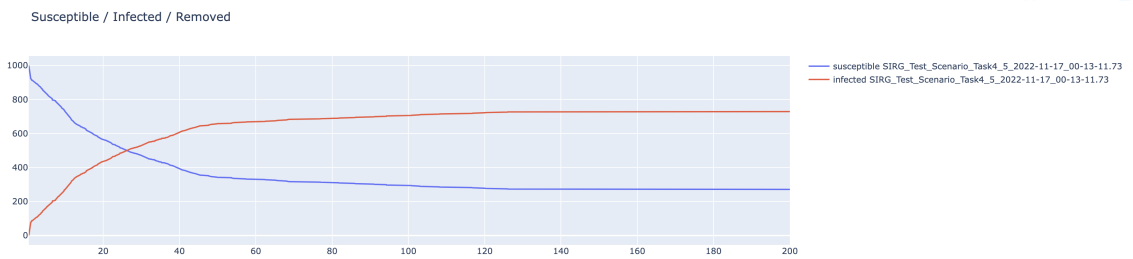


Figure 15: Dash/Ploty view of a single simulation - time (x) and pedestrians (y)

- the second one with the same attributes as the first one, with the only difference that the infection rate is increased to 0.1. As expected, after increasing the infection rate, the infection is spreading faster and in fact, half of the population become infected in a shorter time, just 5.7 s.

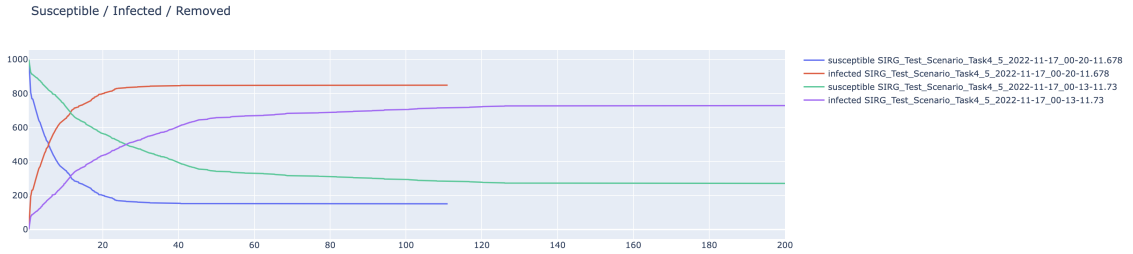
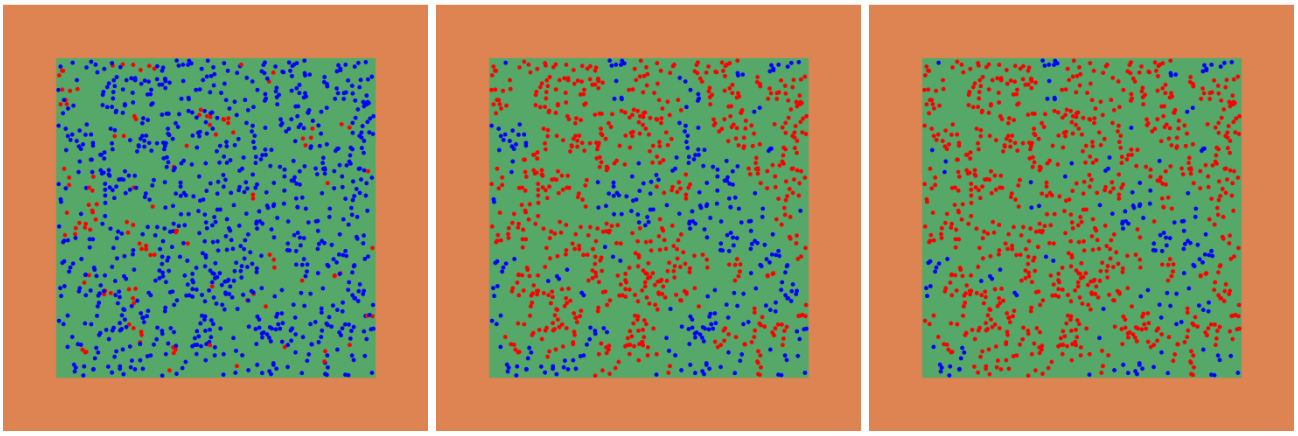


Figure 16: Dash/Ploty view of the multiple simulation - time (x) and pedestrians (y)

As shown in the Figures 15 and 16 of the Dash/Ploty views, a better graphically of how the infection is spreading can be seen. Obviously, the scenario with 0.1 infection rate delivers a faster infection spreading and becomes establish much quicker than the scenario with a smaller infection rate. Finally, in the figure 17 we can see an example of the simulation of the first scenario in the GUI.



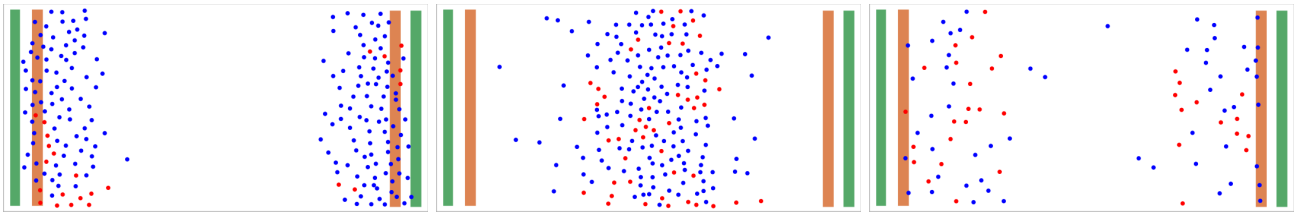
(a) At the beginning of the simulation (b) In the middle of the simulation (c) At the end of the simulation

Figure 17: Different stages of the simulation of the first scenario

Regarding the second scenario, a dynamic corridor scenario of 40m x 20m with the following parameters: the infection rate has been set to 0.05 ("infectionRate": 0.05) with 10 infected pedestrians infected at the beginning ("infectionsAtStart": 10) and the maximum infection distance is set to 1.0 ("infectionMaxDistance": 1.0). The simulation will be executed with a simulation time ratio of 1.0 ("realTimeSimTimeRatio" = 1.0) and the time step will be 0.5 ("simTimeStepLength"= 0.5). And the attribute useFreeSpaceOnly as true.

In this case, to make it more realistic, there are infected pedestrians in both groups. Because it is not realistic to have just infected pedestrians in one side and susceptible pedestrians on the other side.

An example of the simulation of this scenario is shown in the following figure 18



(a) At the beginning of the simulation (b) In the middle of the simulation (c) At the end of the simulation

Figure 18: Different stages of the simulation of the second scenario

As expected, in the middle of the simulation (Figure 18b), the number of the infected pedestrians have increased because there is a counter-flow between non-infected and infected pedestrians of the two groups. And at the end of the simulation (Figure 18c) it can be observed that the number of infected pedestrians are much more than at the beginning of the simulation.

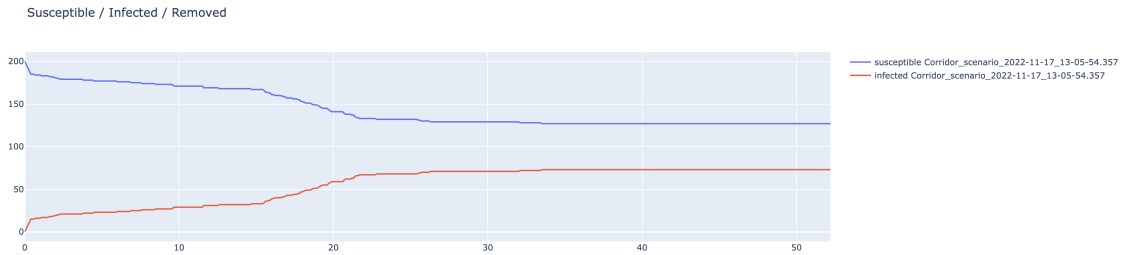


Figure 19: Dash/Ploty view of the scenario 2 simulation - time (x) and pedestrians (y) with 10 infected pedestrians at start

In the Figure 19 we can see a better evolution of the number of infected and susceptible pedestrians during the simulation. The interaction of the two groups can be identified during the periods of 15.5s and 22.5s, where the number of infected pedestrians increases from 34 to 67, so 23 healthy pedestrians become infected during the counter-flow. After the counter-flow, the number of susceptible and infected pedestrians is again stabilised. The number of infected pedestrians does not surpass the number of susceptible pedestrians because the initial number of infected pedestrians is not very big (10), if we increase this parameter, for example to 50, a much bigger number of pedestrians become infected at the end as we can see in the next Figure 20 and the number of infected pedestrians is increased much more during the counter-flow. The same effect can be achieved by increasing the infection rate.

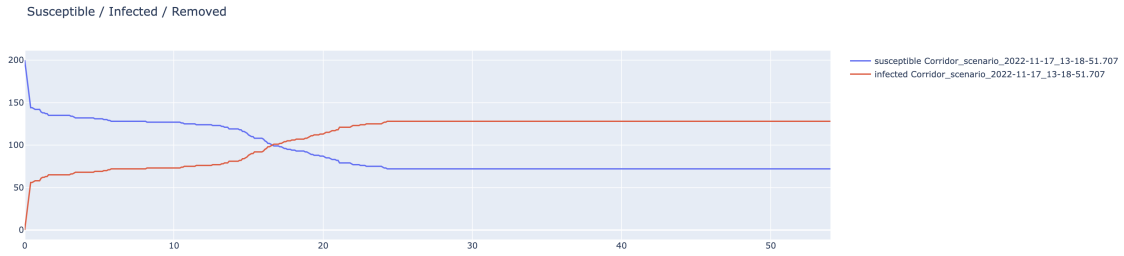


Figure 20: Dash/Ploty view of the scenario 2 simulation - time (x) and pedestrians (y) with 50 infected pedestrians at start

Infection rate and time step decoupling

A way to decouple the infection rate and the time step would be to fix the *update()* method to execute once every certain time independently of the time step size.

This can be done by implementing an accumulator/repeater: for every invokation of *updateLocomotionLayer()*, which is the method that calls *update()*, instead of calling *update()* every time, depending on whether the time step is greater or less than the fixed time, the *update()* method may be called many times at once or once every a certain number of calls. This allows the fixed time to be as flexible as possible: for example, given a time step of size 0.5, if the fixed time is 2 seconds, the infection process should be executed once every four calls of *updateLocomotionLayer()*; if the fixed time is 0.25 seconds, it should be executed twice every one call.

The implementation of this decoupling is shown in Listing 4. A private attribute *accumulatedTime* has been added for keeping track of the accumulated time. The while loop is enough for treating both cases. This should only affect the SIR model, therefore an if condition of the model being an instance of such class has been added.

```
private double accumulatedTime = 0.0;

// ...

private void updateLocomotionLayer(double simTimeInSec) {
  for (Model m : models) {
    List<SourceController> stillSpawningSource = this.sourceControllers.stream().filter(s -> !s.
    isSourceFinished(simTimeInSec)).collect(Collectors.toList());
    int pedestriansInSimulation = this.simulationState.getTopography().getPedestrianDynamicElements
    ().getElements().size();
    int aerosolCloudsInSimulation = this.simulationState.getTopography().getAerosolClouds().size();

    // Only update until there are pedestrians in the scenario or pedestrian to spawn or aerosol
    clouds persist
    if (!stillSpawningSource.isEmpty() || pedestriansInSimulation > 0 || aerosolCloudsInSimulation >
     0) {
      if (m instanceof SIRGroupModel) {
        accumulatedTime += this.attributesSimulation.getSimTimeStepLength();
        // fixedTime can be any positive value
        double fixedTime = 1.0;
        while (accumulatedTime >= fixedTime) {
          m.update(simTimeInSec);
          accumulatedTime -= fixedTime;
        }
      }
      else m.update(simTimeInSec);

      if (topography.isRecomputeCells()) {
        // rebuild CellGrid if model does not manage the CellGrid state while updating
        topographyController.update(simTimeInSec); //rebuild CellGrid
      }
    }
  }
}
```

Listing 4: Decoupled version of the *updateLocomotionLayer()* method

Possible extensions of the model beyond

It is broadly known (specially nowadays) that there are a lot of factors involved in the propagation, recovery and symptoms of a disease: indoor spaces might increase the infection ratio, pedestrians with healthy habits might take faster to recover from a disease or have less symptoms... Therefore, many features could be added as extensions of the model beyond and would help to make it more realistic and customizable, some examples are:

- Face masks and indoor spaces: a pedestrian might wear a face mask. Face masks could reduce the infection ratio. Pedestrians staying for too long in indoor spaces might constantly increase the infection ratio.
- Vaccinated: a vaccinated pedestrian might recover faster from a disease than a non-vaccinated one. It could also reduce the ratio of being infected (or even dying). In addition, the effectiveness of a vaccine might change over time.
- Age: the age of a pedestrian might affect the recovery period and even the death ratio. It would be different for every age group, depending on the characteristics of the disease. For example, kids might be less susceptible than elders. Elders might have a higher death ratio.

Report on task 5, Analysis and visualization of results

Adding the recovered state

For the recovered state representing recovered pedestrians, which cannot get re-infected and cannot infect susceptible persons, since the removed state will not be considered in this exercise, it has been substituted for this new state. The overall changes are the following:

- *ID_RECOVERED* added to the enumeration class *SIRType*.
- *recoveryRate* attribute added to the *AttributesSIRG* class.
- Added an implementation for infested pedestrians (Figure 5). Now, for infested pedestrians, there is a chance of recoveryRate to recover from the disease. Recovered pedestrians do not do anything.
- For the Dash/Plotly visualization, the *SIRvisualization* app has been changed accordingly (Figures 6 and 7. This new version takes into account the added recovered state.

```
public void update(final double simTimeInSec) {
  // Check the positions of all pedestrians and switch groups to INFECTED (or REMOVED).
  DynamicElementContainer<Pedestrian> c = topography.getPedestrianDynamicElements();
  LinkedCellsGrid<Pedestrian> grid = c.getCellsElements();
  double infectionRadius = attributesSIRG.getInfectionMaxDistance();
  double infectionRate = attributesSIRG.getInfectionRate();
  if (c.getElements().size() > 0) {
    for(Pedestrian p : c.getElements()) {
      // Only check for infected and susceptible pedestrians
      // Infected pedestrians might get recovered
      SIRGroup g = getGroup(p);
      if (g.getID() == SIRType.ID_INFECTED.ordinal()) {
        double recoveryRate = attributesSIRG.getRecoveryRate();
        if (this.random.nextDouble() < recoveryRate) {
          elementRemoved(p);
          assignToGroup(p, SIRType.ID_RECOVERED.ordinal());
        }
      }
      // Susceptible pedestrians might get infected
      else if (g.getID() == SIRType.ID_SUSCEPTIBLE.ordinal()) {
        // Get neighbors within a radius of infectionRadius from the position of the Pedestrian p
        VPoint position = p.getPosition();
        List<Pedestrian> neighbors = grid.getObjects(position, infectionRadius);
        // Try to infect p with probability infectionRate with as many attempts as infected
    neighbors are there
        for (Pedestrian n : neighbors) {
          if (p == n || getGroup(n).getID() != SIRType.ID_INFECTED.ordinal())
            continue;
          if (this.random.nextDouble() < infectionRate) {
            elementRemoved(p);
            assignToGroup(p, SIRType.ID_INFECTED.ordinal());
          }
        }
      }
    }
  }
}
```

Listing 5: New version of the *update()* method, with a implementation of the recovered state

```
def file_df_to_count_df(df,
                        ID_SUSCEPTIBLE=1,
                        ID_INFECTED=0,
                        ID_RECOVERED=2):
    """
    Converts the file DataFrame to a group count DataFrame that can be plotted.
    The ID_SUSCEPTIBLE and ID_INFECTED specify which ids the groups have in the Vadere processor
    file.
    """
    pedestrian_ids = df['pedestrianId'].unique()
    sim_times = df['simTime'].unique()
    group_counts = pd.DataFrame(columns=['simTime', 'group-s', 'group-i', 'group-r'])
    group_counts['simTime'] = sim_times
    group_counts['group-s'] = 0
    group_counts['group-i'] = 0
    group_counts['group-r'] = 0

    for pid in pedestrian_ids:
        simtime_group = df[df['pedestrianId'] == pid][['simTime', 'groupId-PID5']].values
        current_state = ID_SUSCEPTIBLE
        group_counts.loc[group_counts['simTime'] >= 0, 'group-s'] += 1
        for (st, g) in simtime_group:
            if g != current_state and g == ID_INFECTED and current_state == ID_SUSCEPTIBLE:
                current_state = g
                group_counts.loc[group_counts['simTime'] > st, 'group-s'] -= 1
                group_counts.loc[group_counts['simTime'] > st, 'group-i'] += 1
            elif g != current_state and g == ID_RECOVERED and current_state == ID_INFECTED:
                current_state = g
                group_counts.loc[group_counts['simTime'] > st, 'group-i'] -= 1
                group_counts.loc[group_counts['simTime'] > st, 'group-r'] += 1
                break
    return group_counts
```

Listing 6: New version of the *file_df_to_count_df()* method, which now counts the added recovered group

```
def create_folder_data_scatter(folder):
    """
    Create scatter plot from folder data.
    :param folder:
    :return:
    """
    file_path = os.path.join(folder, "SIRinformation.csv")
    if not os.path.exists(file_path):
        return None
    data = pd.read_csv(file_path, delimiter=" ")

    print(data)

    ID_SUSCEPTIBLE = 1
    ID_INFECTED = 0
    ID_RECOVERED = 2

    group_counts = file_df_to_count_df(data, ID_INFECTED=ID_INFECTED, ID_SUSCEPTIBLE=ID_SUSCEPTIBLE,
     ID_RECOVERED=ID_RECOVERED)
    # group_counts.plot()
    scatter_s = go.Scatter(x=group_counts['simTime'],
                           y=group_counts['group-s'],
                           name='susceptible ' + os.path.basename(folder),
                           mode='lines')
    scatter_i = go.Scatter(x=group_counts['simTime'],
                           y=group_counts['group-i'],
                           name='infected ' + os.path.basename(folder),
                           mode='lines')
    scatter_r = go.Scatter(x=group_counts['simTime'],
                           y=group_counts['group-r'],
                           name='recovered ' + os.path.basename(folder),
                           mode='lines')
    return [scatter_s, scatter_i, scatter_r], group_counts
```

Listing 7: New version of the *create_folder_data_scatter()* method, which now adds the scatter plot of the recovered group

Testing of the new setup

Before proceeding with the different proposed tests, a simulation of a simple scenario has been run to check that everything works fine.

This scenario is similar to the static scenario with 1000 pedestrians, but it is smaller and only has 100 pedestrians to make it simpler. The infection rate has been set to 0.1 and there will be 10 infected pedestrians at start. The newly implemented infection/recovery process will be executed with a fixed time of 1 second, and the time step will be 0.5.

First, by setting the recovery rate to 0.0, the expected result of the simulation is that the number of infected pedestrians will grow up until whether there are no more susceptible pedestrians remaining, or the remaining susceptible pedestrians cannot be infected because they are out of the range of any infected pedestrian.

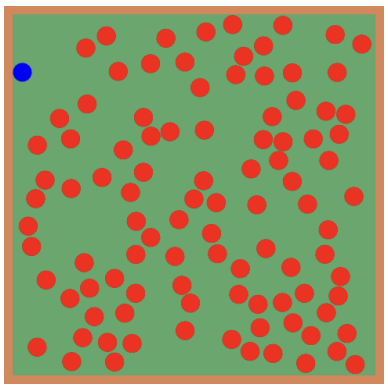


Figure 21: End of the simulation of a simple scenario

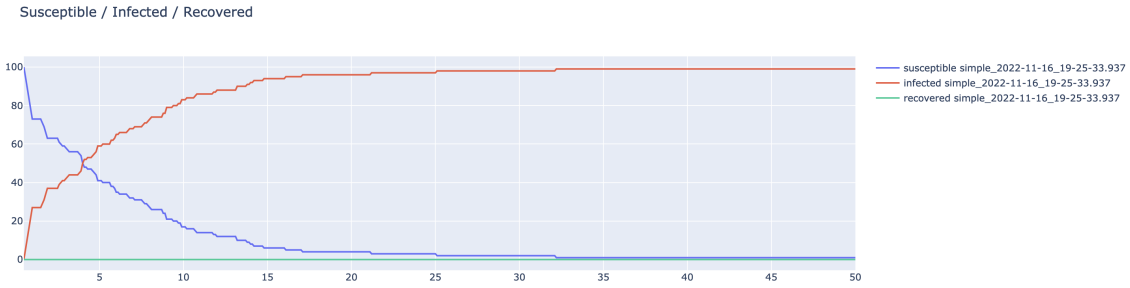
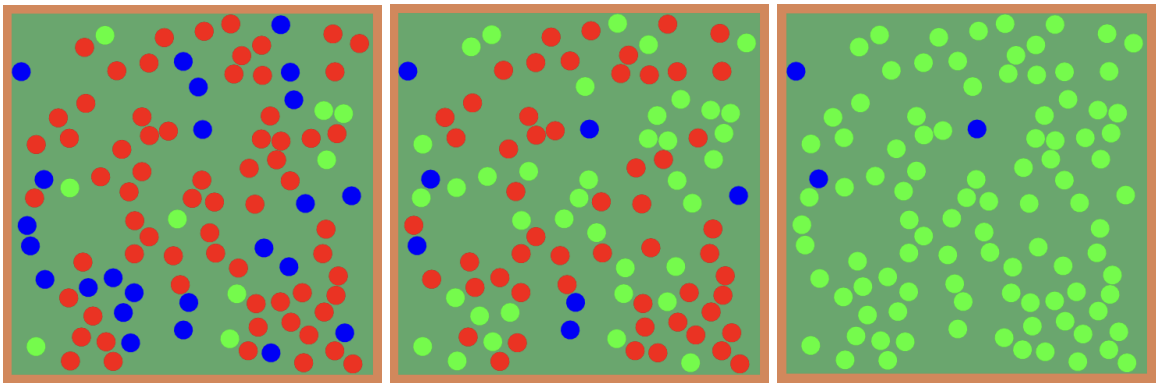


Figure 22: Evolution of the simulation of a simple scenario

As observed in Figure 21, in the end of the simulation there was only one susceptible pedestrian left, who did not get infected because it was out of range of the other infected pedestrians. A representation of the evolution of the infection progress is shown in Figure 22. This behaviour was the expected, since no recovered pedestrians are being introduced.

Next, the recovery rate will be set to 0.05. This means that infected pedestrians will eventually recover from the disease, thus the number of infected pedestrians will not grow up as quickly as in the previous case. At a given point, the number of infected pedestrians will only decrease until all of them get recovered.



(a) At the beginning of the simulation (b) In the middle of the simulation (c) At the end of the simulation

Figure 23: Different stages of the simulation of a simple scenario with recovery rate of 0.05

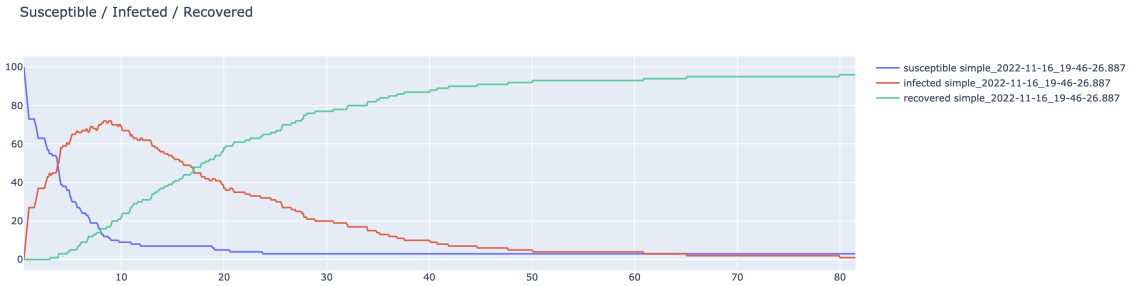


Figure 24: Evolution of the simulation of a simple scenario with a recovery rate of 0.05

As seen in Figures 23 and 24, there were many more infected than recovered pedestrians at the beginning of the simulation. Later on, infected pedestrians became recovered until no infected pedestrian remained. More pedestrians remained susceptible than in the previous case, since there is the possibility of a pedestrian being able to recover before it infects all its neighbors. The implementation seems to be correct so everything is ready for the next tests.

Test 1

This first test asks to construct a fairly large scenario with a source spawning 1000 pedestrians over a target, so that they keep still during the simulation. The aim of this test is to visualize how the susceptible, infective and recovered numbers change over time.

For this, a source with dimensions 25x25 has been placed with 1000 static pedestrians (Figure 25. The first values set for the infection and recovery rates are going to be 0.1 and 0.05, respectively. As asked, the simulation begins with 10 infected and 990 susceptible pedestrians at the start.

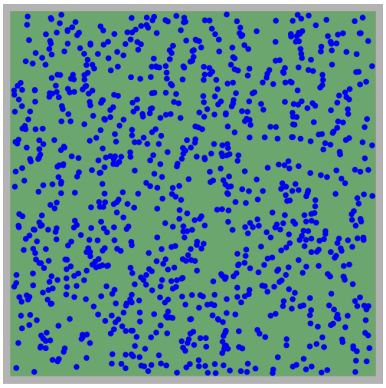


Figure 25: Setup of the scenario for the test 1

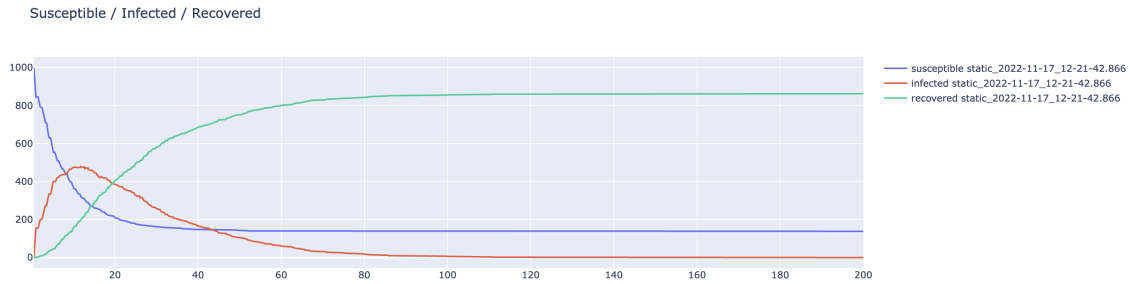


Figure 26: Evolution of the scenario for the test 1

The result of this simulation is displayed in Figure 26. It is practically identical to the results obtained with the introductory test (Figure 24), with the difference that there are ten times more pedestrians.

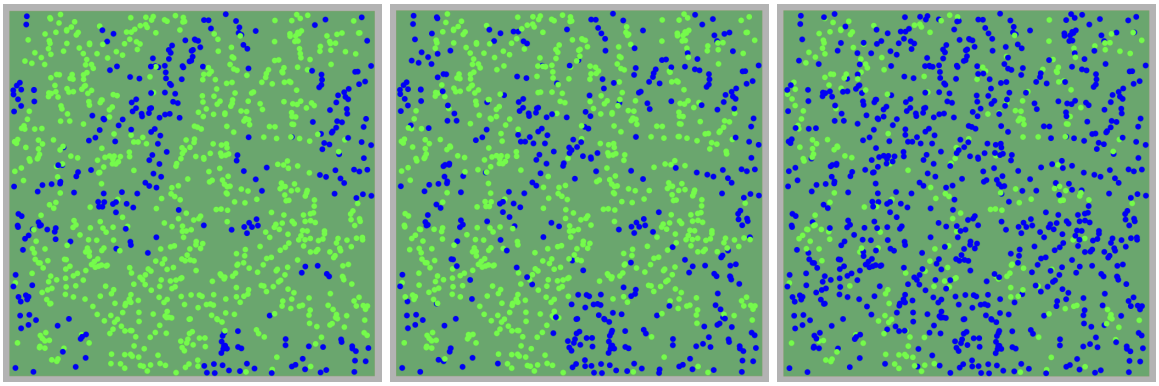
Again, since pedestrians can recover now, the number of infected pedestrians does not grow as quickly as before, without this new state. Around 175 pedestrians remained susceptible whether because there were not infected pedestrians in their neighborhood at any time, or because there were infected neighbors but they recovered from it before spreading it.

It should be highlighted that even though at a given point the number of infected pedestrians stopped growing, it does not mean that no more pedestrians were being infected. In fact, there were, but it kept decreasing because fewer and fewer susceptible pedestrians remained from time to time, until no more pedestrians could be infected, while already infected pedestrians kept recovering from the disease.

Test 2

The second test asks to experiment with the infection and recovery rates in the same scenario as before, and see how the evolution of the disease changes when these parameters are changed.

Since the behaviour when the recovery rate is smaller than the infection rate has already been studied in the previous test, for the next simulations the recovery rates will be equal, higher or much higher than the infection rate, which will be kept fixed in 0.1.



(a) recoveryRate = 0.1 (b) recoveryRate = 0.2 (c) recoveryRate = 0.5

Figure 27: End of different simulations with different recovery rates

By comparing the results of the simulations using different recovery rates as shown in Figures 27 and 28, it can be concluded that the higher the recovery rate is, the weaker the disease becomes, meaning that it disappears earlier. Therefore, the number of remaining susceptible pedestrians increases in relation to this parameter.

Going one step further, the relation between the recovery rate and the total number of recovered pedestrians has been studied as well. By taking a look at the plot displayed in Figure **??**, it is remarkable how in the interval from recovery rate 0.2 to 0.3 there is a major impact on the spread of the disease. From the 0.3 ratio on, the decrease relaxes until the 0.9 ratio, where a slight increase can be noticed. This increase, however, might have been impacted by the execution of the program, but it is also a proof of the ratio not being so relevant when it is already close to 1.0.

In conclusion, it has been observed that the higher the recovery rate was, the weaker the disease became. This increase in the recovery rate resulted in a continuous decrease in the total number of recovered pedestrians, but rather than being a constant decrease, there are intervals where a major decrease can be noticed, which in this case was between the rates 0.2 and 0.3.

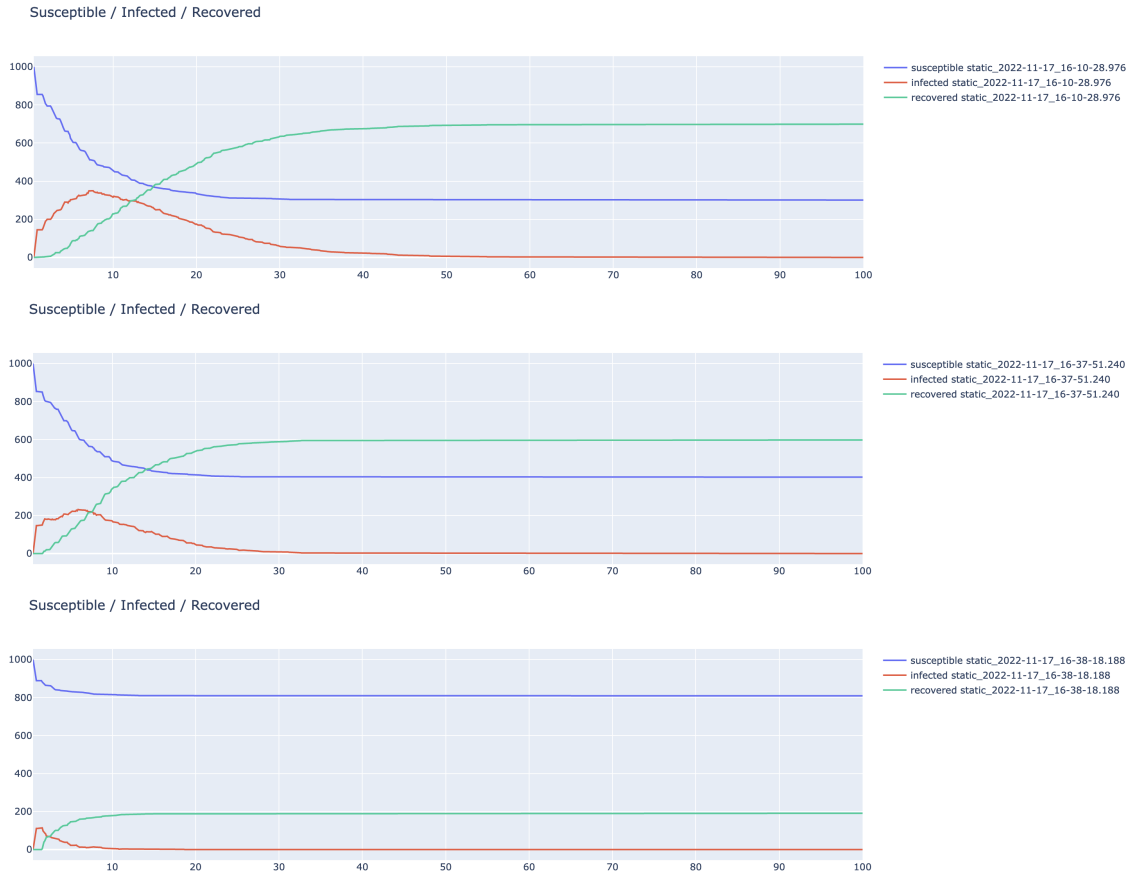


Figure 28: From top to bottom, evolution of the simulation with recovery rates 0.1, 0.2 and 0.5

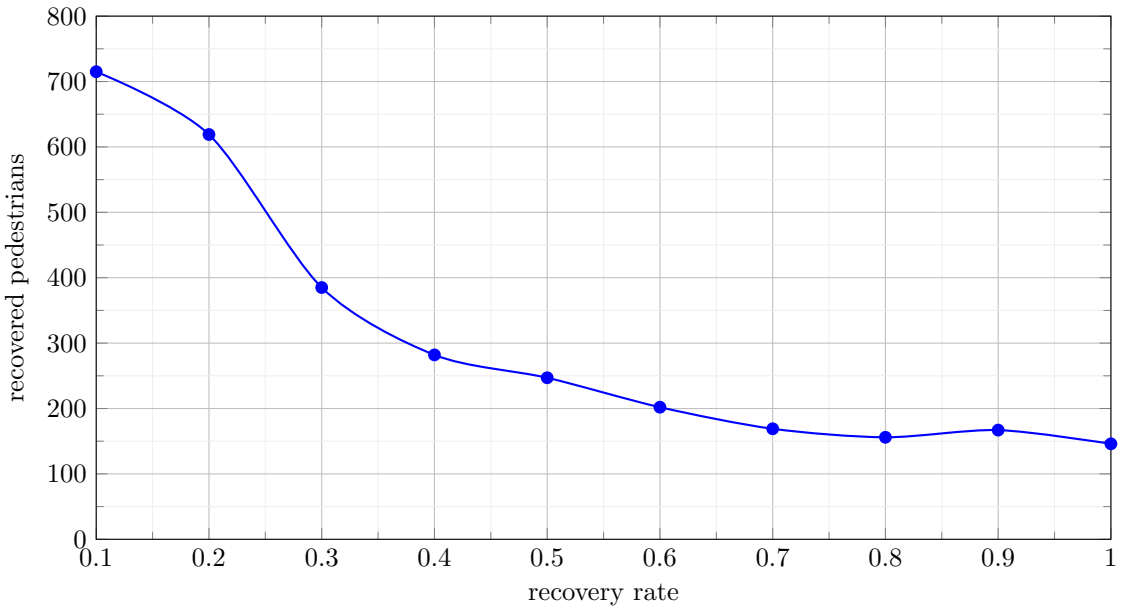


Figure 29: Relation between the recovery rate and the number of recovered pedestrians

Test 3

For the third test, we implemented a supermarket scenario (Figure 30) for testing the actions of customers and the result of social distancing by using different pedPotentialPersonalSpaceWidth within the process of simulation. A 50x50 scenario with 11 different kind of merchandise, 7 cashiers, a passage for customers that did not buy anything, and 1 exist as targets is manually constructed first.

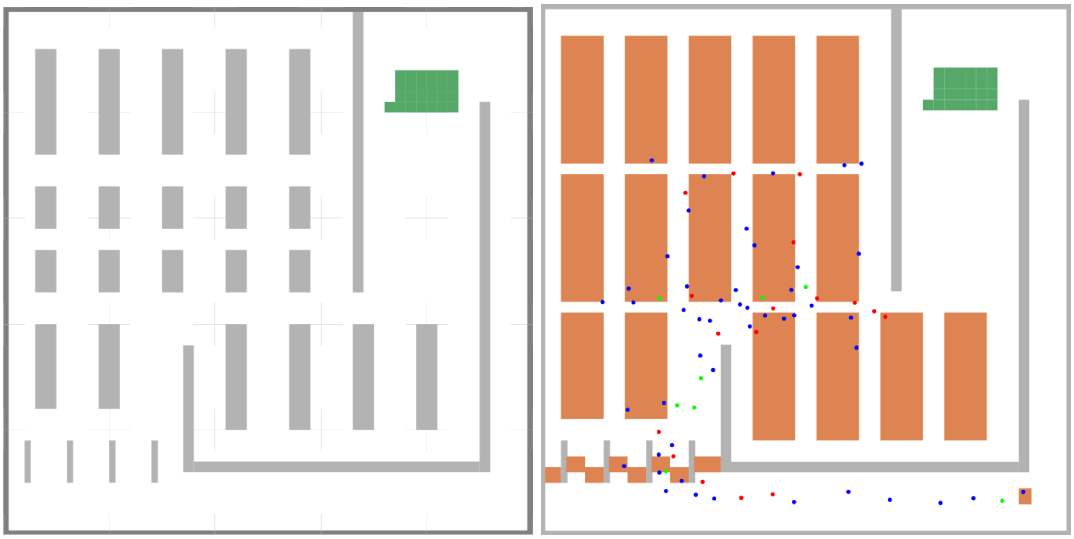


Figure 30: The implemented supermarket topography and simulation result, showing obstacles representing shelves and walls, target areas representing different departements such as groceries and beverage products, etc., and automatically added sources of customers with different shopping list.

Because we would like to simulate the real trajectories of customers travelling through different shelves buying different sort of commodities, we implemented Python code to automatically add sources of customers with different shopping lists, each of which is sampled randomly from the list of all options. A source for customers buying nothing is also added.

Considering for simulating the varying total number of customers in the supermarket from time to time within the opening hours, the starting and ending time of generating customers of each sources is also randomly chosen within a range, which makes the total rate of adding customers varies through time, i.e., only a few sources generates customers at the beginning and the end, whereas all sources are adding customers in the middle of simulation.

In order to see the result in a more clear and efficient manner, we set the attributes of SIR model as following: the infection rate has been continued setting to 0.05 ("infectionRate": 0.05) with 10 infected pedestrians infected at the beginning ("infectionsAtStart": 10) and the maximum infection distance is set to 1.0 ("infectionMaxDistance": 1.0). The recovery rate is set to 0.005 ("recoveryRate": 0.005) as to keep the number of infected customers not decreasing too much within the supermarket. The simulation will be executed with a simulation time ratio of 1.0 ("realTimeSimTimeRatio" = 1.0) and the time step will be 0.5 ("simTimeStepLength"= 0.5). And the attribute useFreeSpaceOnly as true.

For changing the pedPotentialPersonalSpaceWidth, we tested it using value of 1.0, 3.0 and 5.0, with each of 25 sources generating 50 pedestrians throughout time in total. SIR result is shown in Figure 31:

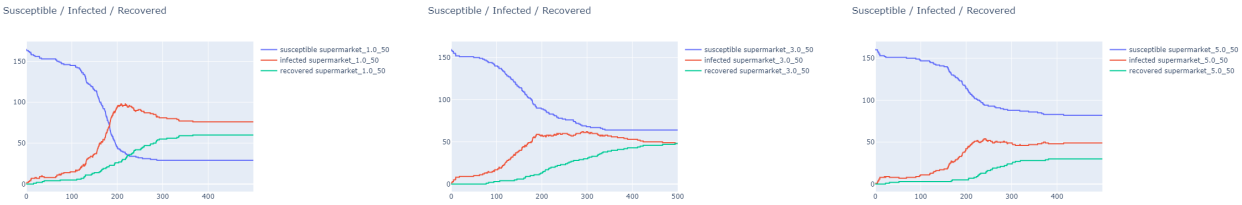


Figure 31: The simulation result with each sources generatng 50 pedestrians in total, and pedPotentialPersonalSpaceWidth = 1.0, 3.0 and 5.0

From which we can see that such social distance does help a little to reduce the number of infections. The final ratio of customers being infected has reduced from around 80% to 50% with the social distance increased five-times. Yet it is still too crowded for further reduction in infection cases within the supermarket, and increasing social distance also caused more time in serving all the customers.

Considering restricting the total amount of customers entering into the supermarket from the beginning. We reduce the number of pedestrians generated by each sources from 50 to 25 and 10. The result is shown in Figure 32 and Figure 33:

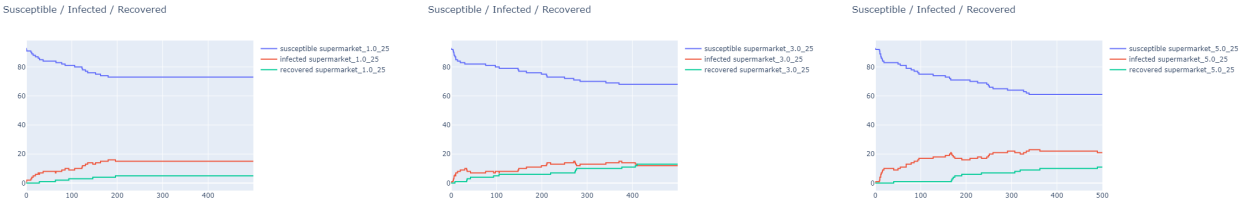


Figure 32: The simulation result with each sources generatng 25 pedestrians in total, and pedPotentialPersonalSpaceWidth = 1.0, 3.0 and 5.0

Figure 33: The simulation result with each sources generatng 10 pedestrians in total, and pedPotentialPersonalSpaceWidth = 1.0, 3.0 and 5.0

As it is shown in the graphs, restricting total customer flow is far more effective than simply forcing a larger social distance. With half traffic flow, the ratio of infected cases for the social distance 1.0 has rapidly dropped around 20% instead of the original 80% (especially considering that half of them is the infected cases introduced at the beginning, which means the actual ratio is around 10%); and a one-fifth customer flow has even managed to keep the number of infected staying the same as the number of pedestrians infected set at the beginning (10), with only less than 5 customers get themselves infected within our supermarket. And conducting social distance policy even worsened the case, since customers cannot get themselves into shelves and cashiers they want to reach as rapidly as before, which caused everyone spending much more time in the supermarket and increased their probabilities of getting infected.

Therefore, conducting social distance policy can reduce the infection cases, whereas controlling the amount of customers entering into our supermarket is far more effective. This can be achieved in the real world by

extending opening hours, or providing discounts during less populous hours to encourage customers to visit when the store is not busy.

Bonus Test 1

Here, we test the case when half of the cashiers were closed. This should increase the total amount of time customers spend within the supermarket, especially in the case of high customer traffic. And the situation could even get worse, because the waiting area for cashiers has the most density of customers, further increase the probability of getting infected.

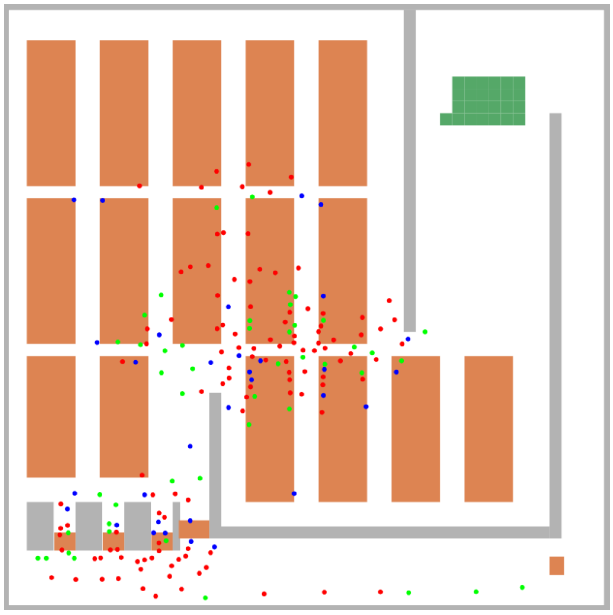


Figure 34: Closing half of the cashiers makes too many customers gathering around the cashier area

Figure 35: The simulation result with removing half cashiers, each sources generating 100 pedestrians in total, and pedPotentialPersonalSpaceWidth = 1.0, 3.0 and 5.0

As Figure 35 shows, closing half cashiers has significantly increased cases of infections, caused an increase in total time customers spent in the supermarket, and also makes social distance become way less effective than before.

Bonus Test 2

Here, we test the case when 1/3 of the shelves were removed. This should also increase the total amount of time customers spend within the supermarket, especially in the case of high customer traffic. The situation should similar to removing cashiers, as it increase the time for customers to find the commodities they wish to get.

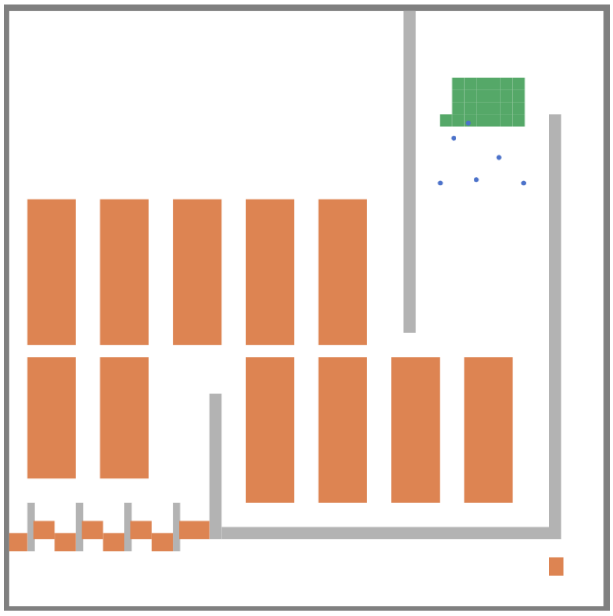


Figure 36: The scenario of removing 1/3 of the shelves

Figure 37: The simulation result with closing 1/3 of shelves, each sources generating 100 pedestrians in total, and pedPotentialPersonalSpaceWidth = 1.0, 3.0 and 5.0

As Figure 37 shows, removing shelves also increased cases of infections, caused an increase in total time customers spent in the supermarket, and makes social distance become way less effective than before as well. The case is even worse than reducing cashiers, as during the process of customers fetching products they are actually passing by and interacting with more people than waiting in the cashier zone.

Bonus Test 3

Here, we would like to test our supermarket's full capacity of serving customers. The amount of customers generated by each sources is set to 200, and number of sources also increased from 25 to 49. We are using the topology scenario same as the one in test 3.

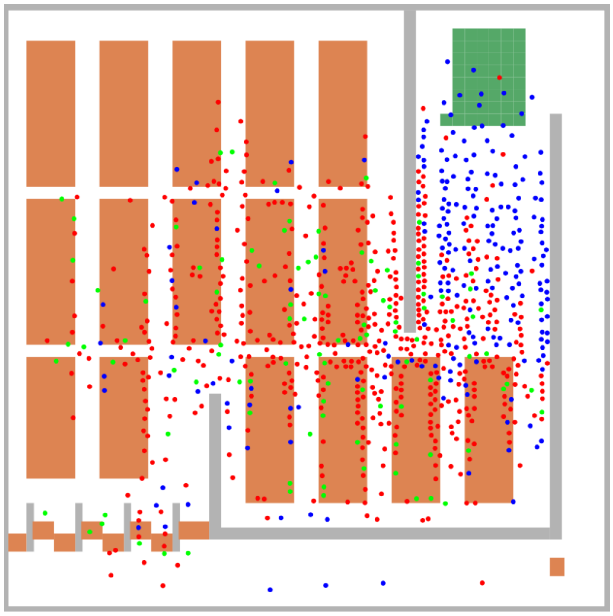


Figure 38: The extreme scenario with four times in total amount of customers

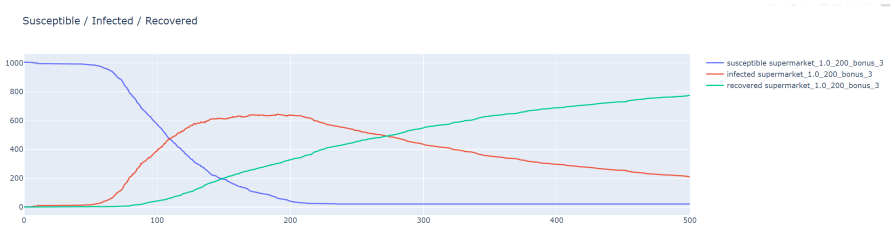


Figure 39: A tremendous amount of infections has happened

As Figure 38 and 39 shows, We can observe that a huge crowd of pedestrians gathered in the middle of the supermarket. With this number of crowds, as one can expect a tremendous amount of infections has happened.