Report for exercise 2 from group A

Tasks addressed: 5
Authors: ALEX PASQUALI (03754113)
NICOLA GUGOLE (03753996)

Last compiled: 2021–11–18
Source code: `https://github.com/AlexPasqua/MLCMS-exercises`

The work on tasks was divided in the following way:

ALEX PASQUALI (03754113)	Task 1	50%
	Task 2	50%
	Task 3	50%
	Task 4	50%
	Task 5	50%
NICOLA GUGOLE (03753996)	Task 1	50%
	Task 2	50%
	Task 3	50%
	Task 4	50%
	Task 5	50%

Report on task 1, Setting up the Vadere environment

Concerning the setup, in order to use Vadere [5], it was necessary to download (or have already installed) Java 11. This is simply realizable by downloading from the Oracle's website [2] the correct Java Development Kit (JDK) installation package for the specific machine that is intended to be used for the task. Once this is done, it is necessary to install the package.
At this point to use Vadere it is necessary to download the software from the Vadere releases website [6] (specifically the "master branch" release instead of the stable one).
Now, after unzipping the downloaded directory, it is possible to open the Vadere GUI simply clicking on the file `vadere-gui.jar`. To test the software, 3 scenario have been run using the *Optimal Steps Model* (OSM) [11, 8, 12]: RiMEA's scenario 1 and scenario 6 [4] and the "chicken test".

RiMEA scenario 1 This test consists of a single pedestrian that has to walk directly towards a single target placed at the center of the grid [4].
The model loaded in Vadere to perform this test is again the OSM [11, 8, 12]. It is possible to select different models for running simulations by clicking on the tab "Model" in the GUI. There are already different models included in Vadere and they can be selected through the "Load template" button.
The scenario in object is very simple, since the pedestrian has only to walk in a straight line towards the target and there are no obstacles nor other pedestrians to avoid. For this reason, the final outcome of this simulation is similar to the one implemented from scratch in the first exercise, however there are some noticeable differences:

- The implementation in exercise 1 is much more discretized, both in time and space:
  - Every element in the scene occupies at least one whole cell and the shapes of every object are always squared or rectangular.
  - In exercise 1 the time step is fixed and the simulated time coincides with the real one. Instead in Vadere it is possible to simulate longer situations in a short time because it is possible to set a ratio between the simulation time step and the corresponding actual time in reality.
- In exercise 1 the walking speed is fixed and set to 1.0m/s, while in Vadere with OSM the speed is sampled from a distribution and is different for each pedestrian (or for multiple runs of the same type of scenario if the seed is changed).
- Concerning the trajectory instead, it tends to stay the same in both cases (exercise 1 and Vadere with OSM). This is probably due to the simplicity of the scenario, where there are no specific interactions to model. Anyway, in the Vadere implementation, it is possible to notice a small curve to the right when the pedestrian is approaching the target (Figure 1). This happens because the pedestrians are actually pointing towards the **center** of the target. The situation could be "corrected" by placing the pedestrian slightly more to the right at the start of the simulation, but the guidelines from RiMEA were followed literally and so the initial dispositions of the objects in the scene. In the exercise 1 implementation, instead, this does not happen because the pedestrian and the target have the same size and they are aligned perfectly .

Figure 1: Slight turn right in RiMEA scenario 1 run in Vadere with OSM [11, 8, 12]. The orange square is the target and the blue line is the pedestrian's trajectory.

RiMEA scenario 6 This scenario consists of 20 pedestrians moving around a left corner to reach a target. In this case the differences between the implementation of exercise 1 and the one in Vadere with OSM are the following:

- In exercise 1, the 20 pedestrians are places at random positions in the beginning of the corridor. This causes the pedestrians to start from different distances from the target, in some sense it is like the simulation had already started. In Vadere instead, it is possible to create a source that originates pedestrians from the same place in the map, making the simulation more realistic.
- The trajectories taken by pedestrians (Figure 2) are much more realistic using OSM compared to the ones in the exercise 1. Furthermore, it is possible to visualize them in the "Post-Visualization" in the GUI, while in the cellular automaton you could only try to track pedestrians moving at simulation time, making visual analyses much more difficult.
  Anyway, it is possible to observe in Figure 2 some weird behaviors in the final part of the trajectories. Some pedestrians that are on the outside try to go to the center of the corridor for no apparent reason, and some of them do the opposite. By closely and slowly observing the simulation, however, it is possible to notice that pedestrians leave the center of the corridor when others are near them, but often they try to return to the center as soon as possible while they could just go straight (and this is what probably would happen in a real situation).
- The overall time necessary to make all the pedestrians reach the target is roughly 29.3 seconds.

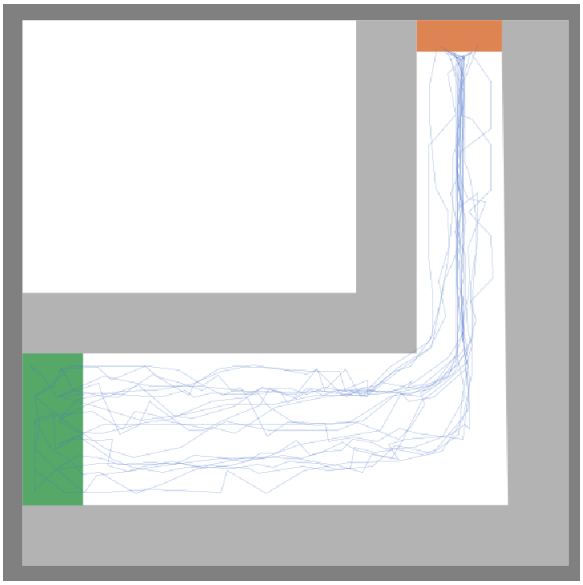


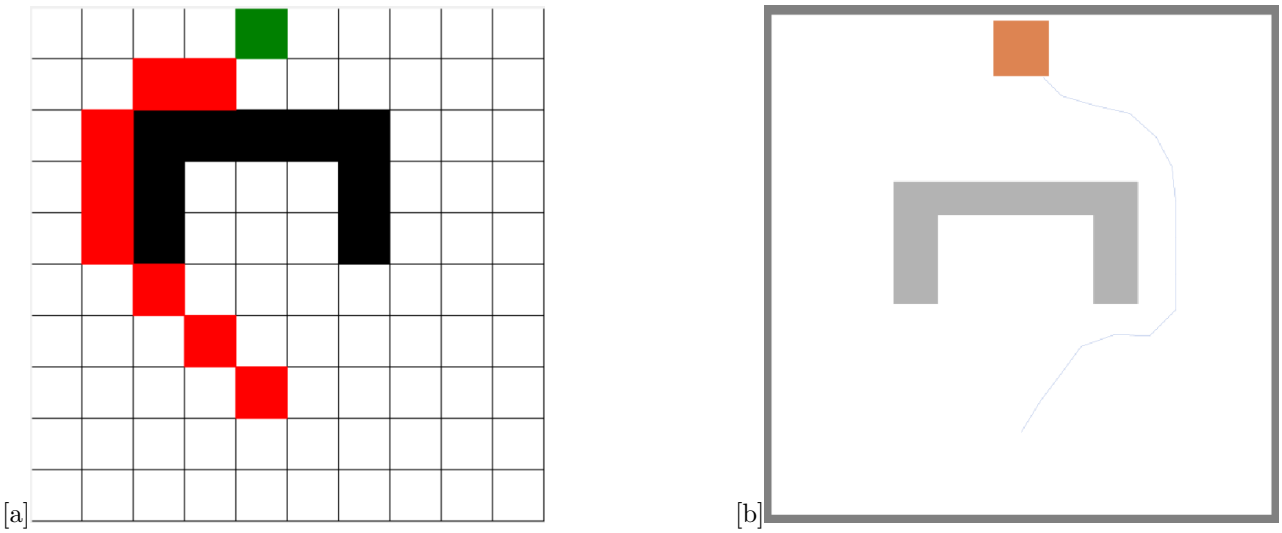
Figure 2: RiMEA scenario 6 run in Vadere with OSM [11, 8, 12].

Chicken test In this test there is one pedestrian in front of a target and they are separated by a U-shaped obstacle aimed to trap the pedestrian in case it goes straight towards the target.
In both the implementations of exercise 1 and Vadere with OSM the test is successfully passed, i.e. the pedestrian does not get trapped by the obstacle but it avoids it and reaches the target. However the trajectories of the two implementations differ significantly:

- Exercise 1: the trajectory is very squared, the pedestrian starts going in diagonal, perfectly at a 45° angle, then it coasts the border of the obstacle until it surpasses it and finally it goes 45° again to reach the target [1]. In Figure 3(a) is is possible to see a reconstruction of the trajectory taken by the pedestrian in the exercise 1, using Dijkstra's algorithm.
- Vadere with OSM: as it is possible to observe in Figure 3(b), the trajectory is more "rounded", but it looks less natural. A pedestrian, seeing such an obstacle, would go directly towards the external edge of it, instead here it goes towards the internal one, for then going right almost horizontally. The final third

[1] Towards the end the pedestrian might actually move horizontally/vertically (and not diagonally) depending on the position of the target, but anyway it is matter of a single move in the last time step.

of the trajectory [2] is a little more natural, even though it could be slightly more direct.
The overall time to reach the target is roughly 7.9 seconds.



[a] [b]

Figure 3: (a) The trajectory of the chicken test in exercise 1 using the Dijkstra's algorithm. (the trajectory has been recreated coloring in red the cells occupied by the pedestrian at each time step (from the bottom to the target (in green)). (b) the trajectory of the pedestrian running the OSM in Vadere.

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

This task consists of running the 3 scenarios of task1 (RiMEA scenario 1, RiMEA scenario 2 and the chicken test) with the *Social Force Model* (SFM) [10, 9] and the *Gradient Navigation Model* (GNM) [7, 8].

Social Force Model (SFM)

- **RiMEA scenario 1:** in this simple test, the trajectory stays straight forward towards the target
- **RiMEA scenario 6:** in this case, the behavior exhibited using SFM is more natural than the one of OSM. All pedestrians tend to follow the optimal/shortest trajectory and they tend to be less repulsed by others. Furthermore, with SFM the paths are more smooth, while with OSM there are some hard angles and squared-like stretches.
  In Figure 4(a) and 4(b) it is possible to see the different trajectories between OSM and SFM. The ones of SFM look more natural especially in a running/panic/evacuation situation.
  Another thing noticeable observing the simulation, more then the trajectories, is that the crowd movement is more fluid in SFM because in OSM some pedestrian tend to give way to others in an excessive manner, stopping when it is not actually very necessary.
  The overall time necessary to make all the pedestrians reach the target is 25.2 seconds [3] (with OSM it was 29.3 seconds).
- **Chicken test:** the test is of course passed also with SFM. Comparing to OSM, the trajectory is again a bit more direct, but anyway very similar. It is possible to compare the 2 trajectories visually in Figure 5(a) and 5(b).
  The overall time to reach the target is 8.3 seconds (roughly 4 tenths of a second slower than OSM).

Gradient Navigation Model (GNM):

- **RiMEA scenario 1:** also using this model, the trajectory stays straight towards the target, as can be expected, given the simplicity of the scenario.

[2] After the vertical section, when the pedestrian surpasses the obstacle and starts going diagonally towards the target.
[3] It is worth noticing that, towards the end, only one single pedestrian, slower than the others, remains alone and makes the overall time increase significantly.

- **RiMEA scenario 6:** all the pedestrian really try to walk along the shortest path to the target (in Figure 4(c) it is possible to observe this line marked more heavily that runs next to the wall). This has the effect of "packing" the pedestrian together and this causes some slowdowns, especially for the pedestrian on the inside/tight line, who get their way continuously cut by the others that were on the outside.
  Overall the movements look still more natural than the ones obtained with OSM, but less so compared to the ones coming from an execution of the same scenario with SFM, in fact some slowdowns would be avoidable if some pedestrian would use the available space on the outside of the trajectory.
  The overall time necessary to make all the pedestrians reach the target is 27.6 seconds, making this model slower than SFM but quicker than OSM in the RiMEA scenario 6.
- **Chicken test:** the GNM takes a very directed trajectory to the target, it looks close to the shortest path that Dijkstra's algorithm could find. It is possible to observe this by comparing Figure 5(c) and Figure 3(a).
  The overall time to reach the target is 8.0 seconds (OSM: 7.9s, SFM: 8.3s).

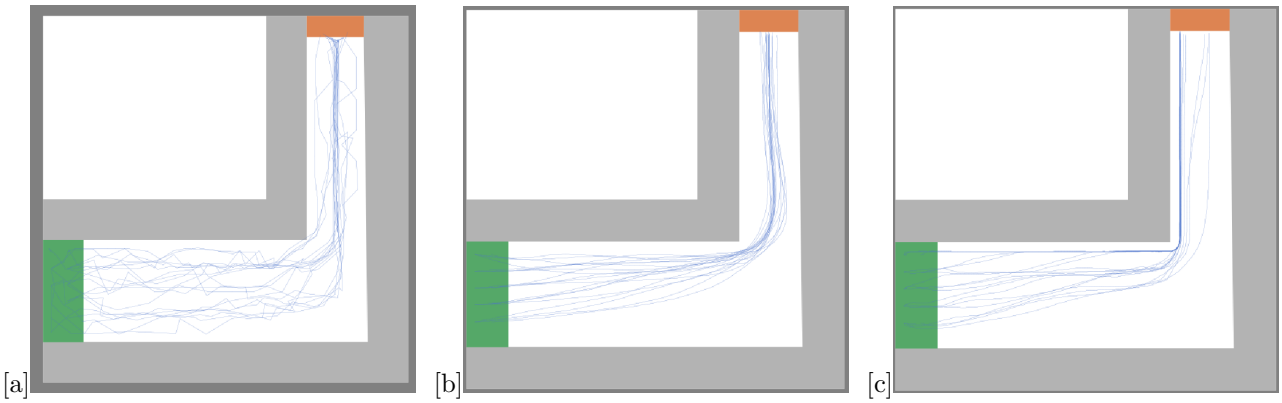


Figure 4: The trajectories of the RiMEA scenario 6 run with OSM (a), with SFM (b) and with GNM (c).

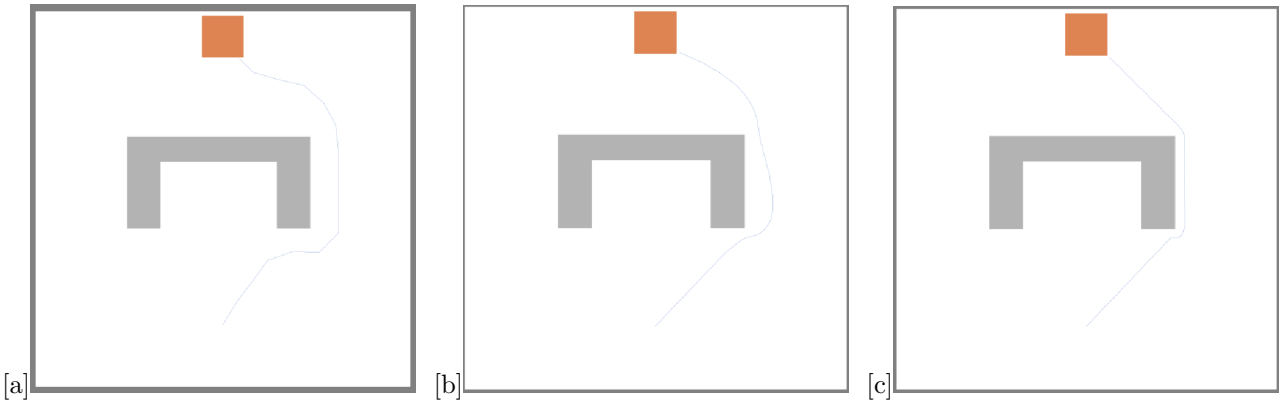


Figure 5: The trajectory of the chicken test with OSM (a), with SFM (b) and with GNM (c).

Report on task 3, Using the console interface from Vadere

Running Vadere from console By running Vadere from console, instead of GUI, there are, as expected, no differences in the output files. This check has been performed using the `diff` command of the Linux bash comparing file by file the outputs of two different runs of the same scenario, one with GUI and the other with console.

Adding pedestrians programmatically In order to be able to insert pedestrians programmatically, the script `add_pedestrian.py` has been created. It is a Python script that contains 3 functions, the main one being `add_pedestrian`.
This function performs the following steps [4]:

- **Get the scenario to modify:** the function takes as parameter either the path to the scenario file or the scenario itself in the form of a dictionary [5]. If the path is passed, the scenario is read with the help of `json.load()`.
- **Get the target id for the new pedestrian:** if the target id is not explicitly passed to the function, check if there is only one single target in the scenario and in that case set its id as the pedestrian's target id [6], otherwise leave this field blank (the pedestrian won't move in this case [7]).
- **Create the pedestrian:** it is simply a dictionary with the same structure (nested lists and dictionaries) present in the JSON file of the scenario under the key `dynamicElements` [8]. Therefore, its structure and its keys are fixed and they have to mimic what found in the JSON file in order to be compatible, while the values associated to each key are passed to `add_pedestrian` as parameters [9].
- **Add the new pedestrian to the pedestrians list:**
  - The list is obtained with: `scenario['scenario']['topography']['dynamicElements']`
  - Add the new pedestrian (namely `ped`):
    `scenario['scenario']['topography']['dynamicElements'].append(ped)`
- **Save the output scenario:** with the help of `json.dump()`. This updated scenario file now contains the new pedestrian and can be loaded into Vadere.

Add pedestrian to RiMEA scenario 6 At this point, it is possible to add a pedestrian in the desired position of the RiMEA scenario 6 simply by calling `add_pedestrian` (described in previous paragraph). An example of a call to this function is shown in Figure 6(a), while the initial scenario for this task is displayed in Figure 6(b).
The newly added pedestrian, starting from a location nearer to the target, reaches the destination first in 5.4 seconds, while the second fastest takes 8.9 seconds.
This test, since the base scenario is the one used in task 1, has been run with OSM.

[4] Only the main ones are reported in this report for a better clarity
[5] In this case it means that a scenario file has already been read with json and the result of `json.load()` itself (i.e. a dictionary) is passed to the function.
[6] This is an implementation choice, it would have been possible not to set the target id automatically in any case, but it was preferred to do it since most of the times each pedestrian has a target id, otherwise it wouldn't move.
[7] If this is not the intended behavior, then the target ids must be passed explicitly.
[8] In the scenario's JSON file, under the key `dynamicElements` it is the list of the pedestrians (the ones that are not originated from a source, to be precise). The pedestrian dictionary created in the function `add_pedestrian` has the same structure of each of the elements in that list.
[9] Some values that were not in the interest of the tasks of this exercise are actually hard-coded and not passed as parameters.

[a]
```
add_pedestrian(
    scenario_path="../task1/scenarios/rimea6.scenario",
    out_scen_name="task3",
    output_path="scenarios/task3.scenario",
    position=(8, 2),
    targetIds=[5]
)
```

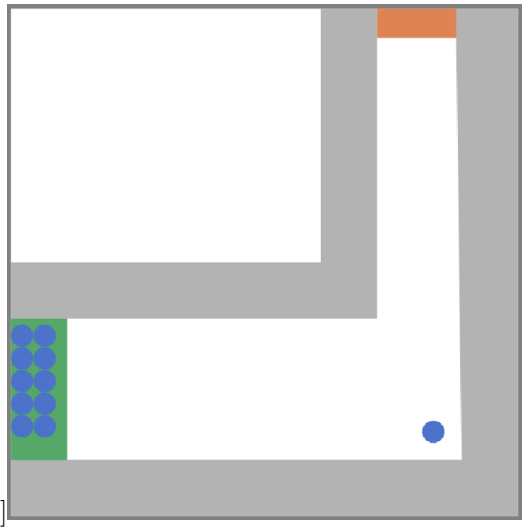
[b]


Figure 6: (a) An example of a call to the function `add_pedestrian`. (b) The initial scenario for this task. The isolated pedestrian has been added programmatically by a call like the one in (a).

Report on task 4, Integrating a new model

To integrate the proposed `SIR model` in Vadere the Moodle files have been positioned in the correct folders, as pointed out in the files first row of code. After re-building the software (using **IntelliJ 2021.2.3 (Community Edition)** and **Java JDK 11.0.13**) one can execute the modified Vadere software by executing the `VadereApplication` class. This original implemented model lacks some features and optimalities: it is composed only by *infected* and *susceptible* pedestrians, as well as not being properly colored nor properly initialized.

Figure 7 gives an overview of the SIR model implementation in Vadere. Going more into detail:

- **AttributesSIRG:** class to set some general parameters regarding the infection behavior.
  - *infectionsAtStart:* how many pedestrians have to be already infected at the start of the simulation.
  - *infectionRate:* how probable it is for a susceptible pedestrian to become infected if he/she is in contact with an infected pedestrian.
  - *infectionMaxDistance:* range in which a susceptible pedestrian gets the risk of being infected if another infected pedestrian is in that range.
- **SIRGroup:** class to maintain an active counting and membership for different pedestrian groups. Each group, *Infected, Susceptible and Recovered* in the proposed case, will keep a list of members.
  - *id:* to uniquely identify a pedestrian group.
  - *members:* list containing the pedestrians in the group. `addMember` and `removeMember` are responsible for the update of the list.
- **SIRType:** `enum` [1] to conveniently store the possible pedestrian states.
- **SIRGroupModel:** main class, responsible for keeping the pedestrians divided in groups, updating them by actually performing infections (and later on possible recoveries).
  - *groupsById:* a `LinkedHashMap` to store the various present groups, identified by their id.
  - *attributesSIRG:* instance containing the aforementioned simulation parameters.
  - *totalInfected:* a counter to keep track of the overall number of currently infected.
  - `getFreeGroupId:` creates groups for storing the infected and susceptible pedestrians, fulfilling also the *attributesSIRG* initial constraints.
  - `initializeGroupsOfInitialPedestrians:` assigns pedestrians in the scenario to groups at the beginning.

  - `assignToGroup:` subroutine to assing a particular pedestrian to a group.
  - `elementAdded:` wrapper of `assignToGroup`.
  - `elementRemoved:` removes a pedestrian from a group.
  - `update:` method which implements the model core. The method goes through every pedestrian, checks for infected nearer than maximal infective range. For every infected pedestrian found in the range a probability coin is tossed and if the pedestrian is unlucky then he/she becomes infected.

- **FootStepGroupIDProcessor:** output processor to maintain the infection status in the resulting file, useful for later analysis and plotting. For a correct processing it is fundamental to assert this processor as the desired output processor, otherwise no infection status information will reach the post simulation.
  - `doUpdate:` at each time step writes a new output row composed of the current time step, the pedestrian id and its group id.

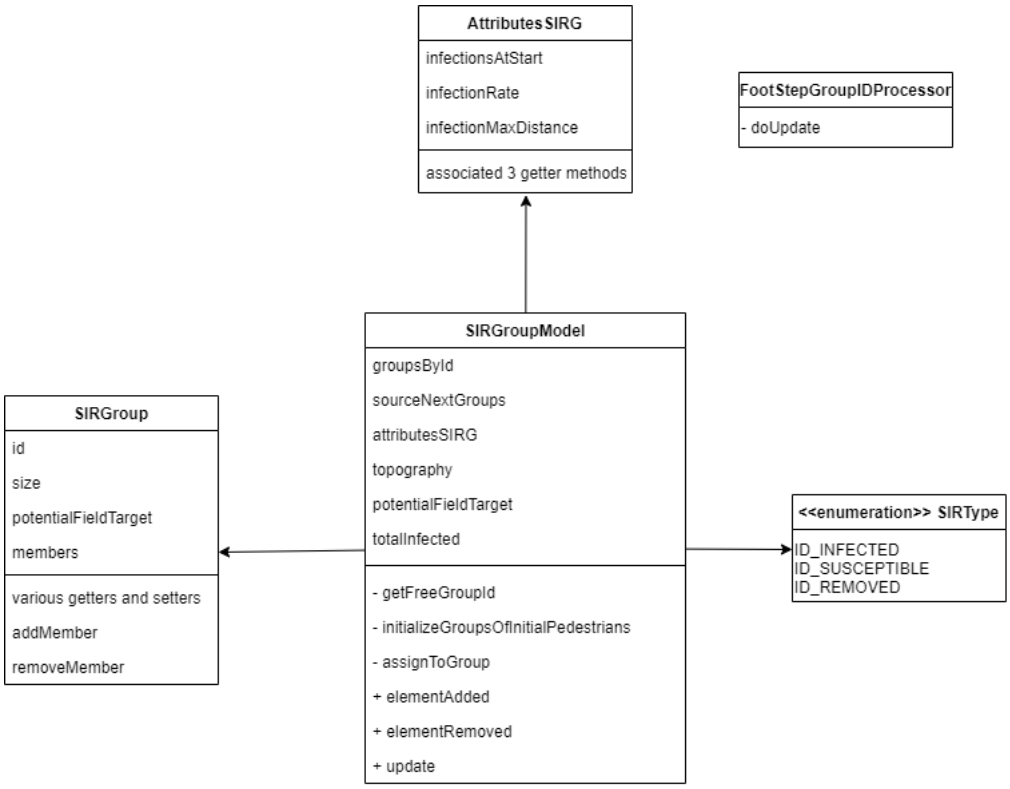


Figure 7: slim version for the SIR model UML Class Diagram

After successfully integrating the SIR model into Vadere, some changes are needed for a fully working (and more efficient) implementation of such model. Namely the need for initial group assignment to pedestrians, the need for a correct coloring for better visualization and finally the need for a more efficient search for neighboring pedestrians during the `update`.

- *Initial group assignment:* without the initial assignment the future simulation updates call are useless and the simulation would not be respecting the constraints imposed by *attributesSIRG*. In **Figure 8** it is possible to see the implementation inside a light blue box. A for loop iterates over all the pedestrians, calling indirectly the `getFreeGroupId` function for group assignment.

```java
private void initializeGroupsOfInitialPedestrians() {
    // get all pedestrians already in topography
    DynamicElementContainer<Pedestrian> c = topography.getPedestrianDynamicElements();

    if (c.getElements().size() > 0) {
        //  to INFECTED or SUSCEPTIBLE groups.
        for(Pedestrian ped : c.getElements()){
            assignToGroup(ped);
        }
    }
}
```

Figure 8: initial group assignment snippet

- *Correct coloring visualization:* the SIR groups are not visualized correctly in the initial setup, even after correct initial group assignment. A good and immediate visualization is absolutely fundamental when analysing behaviours, even though this coloring is not an issue neither for the output processor nor for the visualization scripts. To solve the problem a couple of lines have been added to `VadereGui/src/org/vadere/gui/components/model/SimulationModel.java`. More precisely, looking at **Figure 9**, the changes consist in adding the basic colors for the three different groups (orange box), changing the default simulator coloring agent (green box) and finally adding the mapping between group id and group coloring (red box). It is important to remark that this solution solves the problem regarding the simulation only, leaving the post visualization groups all identical in color as proposed in **Figure 10**.

```java
public abstract class SimulationModel<T extends DefaultSimulationConfig> extends DefaultModel {

    public final T config;
    private ConcurrentHashMap<Integer, Color> colorMap;
    private Random random;
    protected final Color infectiveColor = new Color( r: 255,  g: 0,  b: 0);     // red    for INFECTIVE
    protected final Color susceptibleColor = new Color( r: 255,  g: 255,  b: 0); // yellow for SUSCEPTIBLE
    protected final Color recoveredColor = new Color( r: 0,  g: 255,  b: 0);     // green  for RECOVERED


    /unchecked/
    public SimulationModel(final T config) {
        super(config);
        this.config = config;
        this.config.setAgentColoring(AgentColoring.GROUP); // to correctly visualize the groups colors
        this.colorMap = new ConcurrentHashMap<>();
        this.colorMap.put(-1, config.getPedestrianDefaultColor());
        // add the desired colors, coupled with the groups ids (0 - infected, 1 - susceptible, 2 - recovered)
        this.colorMap.put(0, getInfectiveColor());
        this.colorMap.put(1, getSusceptibleColor());
        this.colorMap.put(2, getRecoveredColor());
        this.random = new Random();
    }
```

Figure 9: simulation color correction snippet

- *Efficient search for neighboring pedestrians during the* `update`*:* originally the function was not efficient. For each pedestrian every other pedestrian would have been taken into consideration as plausible infective agent. In reality only those pedestrians who are both infectious and close enough to the current pedestrian are plausible for infection. This is anyway not the only inefficiency: all pedestrians are in fact checked

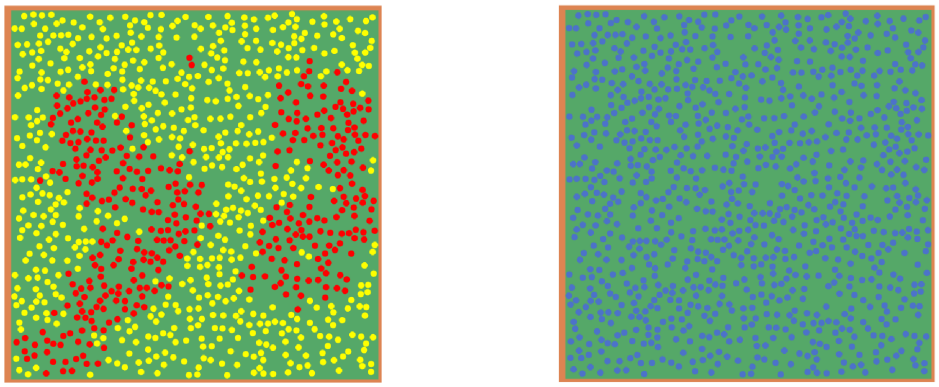


Figure 10: Group coloring during and after the simulation

for a change of state even if at the current state of the model only susceptible pedestrians are capable of changing state. Therefore it would be a good idea to act only on susceptible pedestrians and no others. Both these situations have been solved as can be appreciated looking at the white box in **Figure 11**. If the current pedestrian is infected than no need to check its neighbours. If he/she is not infected, than, by having information on the cell elements, the current pedestrian will now look for infectious pedestrians only at its surroundings in the max infective range. This is possible thanks to the use of the `LinkedCellsGrid` class, which effectively implements a grid augmenting the position of generic objects (the pedestrian in this case), for faster access. Particularly useful resulted to be the `getObjects` method inside of the class, a method which exploits the constructed grid to get all the required neighbours of a given pedestrian. The method only needs the pedestrian position as `VPoint` object and the `radius` of search.

```java
@Override
public void update(final double simTimeInSec) {
    // check the positions of all pedestrians and switch groups to INFECTED (or REMOVED).
    DynamicElementContainer<Pedestrian> c = topography.getPedestrianDynamicElements();
    if (c.getElements().size() > 0) {
        for (Pedestrian p : c.getElements()) {
            // loop over neighbors and set infected if we are close

            // if p is already infected then no need to execute
            if (getGroup(p).getID() == SIRType.ID_INFECTED.ordinal())
                continue;
            // loop ONLY on the real neighbours
            List<Pedestrian> pNeighbours = c.getCellsElements().getObjects(p.getPosition(),
                    attributesSIRG.getInfectionMaxDistance());

            for (Pedestrian p_neighbor : pNeighbours) {
                if (p == p_neighbor || getGroup(p_neighbor).getID() != SIRType.ID_INFECTED.ordinal())
                    continue;
                if (this.random.nextDouble() < attributesSIRG.getInfectionRate()) {
                    SIRGroup g = getGroup(p);
                    if (g.getID() == SIRType.ID_SUSCEPTIBLE.ordinal()) {
                        elementRemoved(p);
                        assignToGroup(p, SIRType.ID_INFECTED.ordinal());
                    }
                }
            }
```

Figure 11: simulation optimization snippet

Once all the due fixes are in place it is possible to experiment with the modified Vadere software. In

particular a couple of experiments have been carried out for this task:

- a static scenario, where **1000** pedestrians are present and **10** are initially infected. The staticness is ensured by keeping the source on top of the target and making the target non-absorbing. The scenario is tested with different infection rate to state and appreciate the change in dynamic behaviour.
- a dynamic corridor scenario of `40mx20m` where a group of 100 susceptible pedestrians which are created overtime (the source attribute `useFreeSpaceOnly` is set to `true`) is moving from right to left and an automatically generated group of 100 infected pedestrians goes from left to right. The automatic generation is possible thanks to the code produced in **Task 3**, which is used as foundation for the script `create_scenario_task4.5_2.py`, adding the 100 infected pedestrians on the left of the corridor. The objective is to grasp how many people get infected in the encounter between the two groups.

Regarding the *static scenario*, two experiments have been carried out:

- `realTimeSimTimeRatio:0.1, simTimeStepLength:0.2` with the following infection parameters: `infectionsAtStart:10, infectionRate:0.01, infectionMaxDistance:1.0`. An example of simulated behaviour can be appreciated in **Figure 12** where half of the population is infected in **47** seconds.
- `realTimeSimTimeRatio:0.1, simTimeStepLength:0.2` with the following infection parameters: `infectionsAtStart:10, infectionRate:0.02, infectionMaxDistance:1.0`. A similar behaviour happens to the one previously shown in **Figure 12**, with the difference standing obviously in the infection spreading faster. Half of the population is infected in fact in only **21** seconds, less than half of the previously reported time.

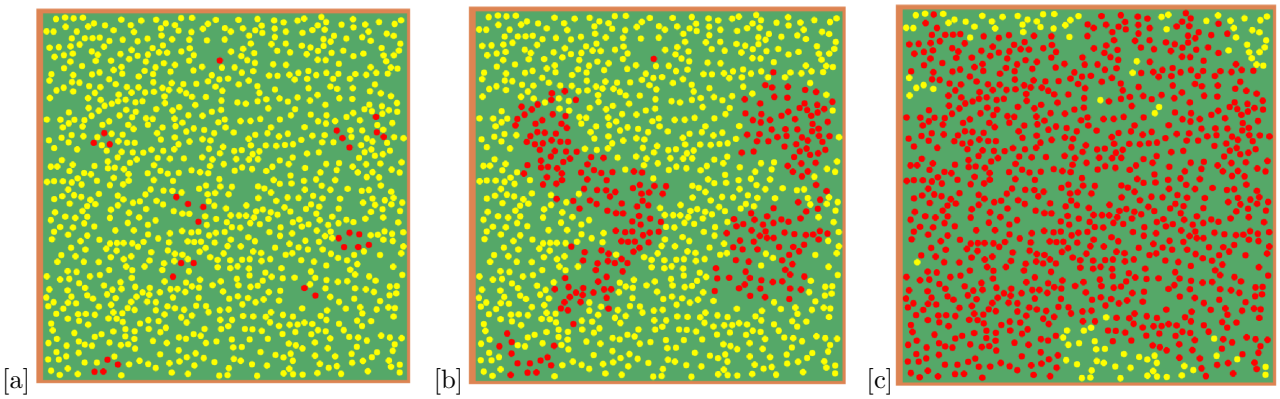


Figure 12: Simulation evolution from left to right.

Using the *Dash/Plotly* visualization utility it is possible to better graphically describe the flow of infection, both for the single simulation and the multiple simulations (**Figure 13**). The graphic immediately shows how the higher infection rate delivers a much faster infection spreading, with the susceptible getting all infected in a visibly shorter time.

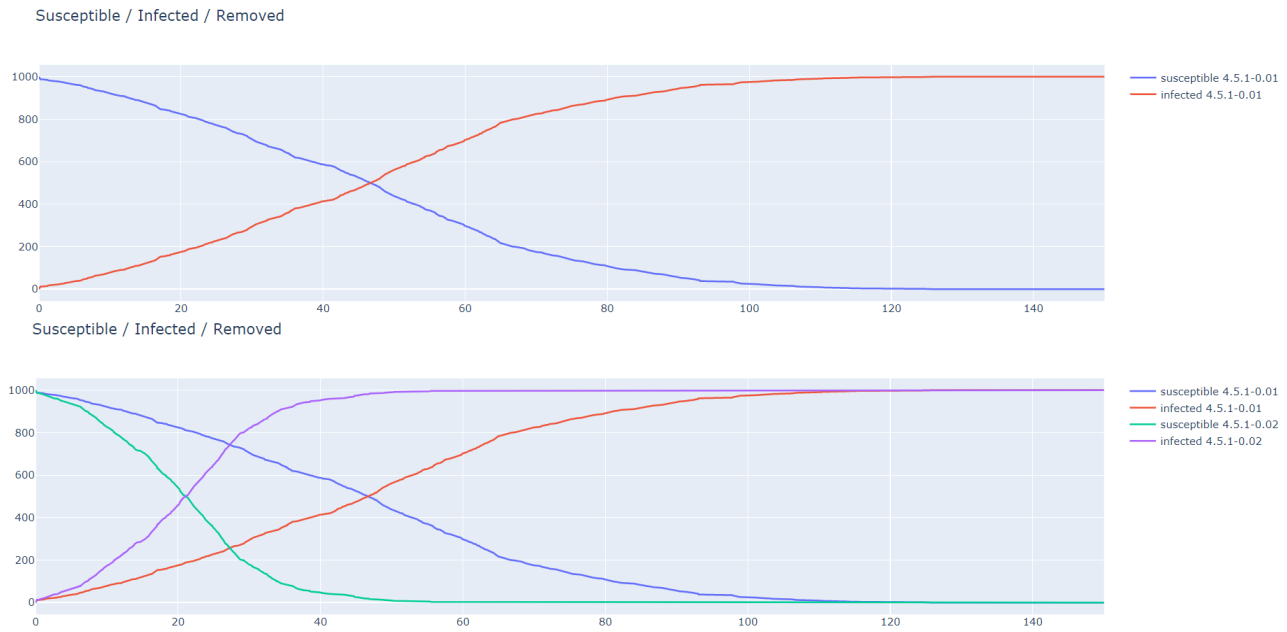


Figure 13: Single (above) and multiple (below) simulation - time on the x, pedestrians on the y

Regarding the *corridor scenario*, the two groups run into each other and interact in the middle of the scenario. Both a simple notebook (`analyze_corridor_counterflow_infections.ipynb`) and the *Dash/Plotly* (**Figure 14**) are able to identify that during the interaction **21** healthy pedestrians get infected. The scenario has the following parameters: `realTimeSimTimeRatio:1.0, simTimeStepLength:0.2, infectionsAtStart:0, infectionsRate:0.1, infectionsMaxDistance:1.0`.
Figure 15 portrays an evolution of the simulated scenario, showing the interaction between the groups. The groups get slowed down during the interaction because of the higher density in the area, simulating well a real world scenario. In the *Dash/Plotly* figure the interaction moment is easily identifiable, with the infection curve rising and then stabilizing again once the two groups get past each other.

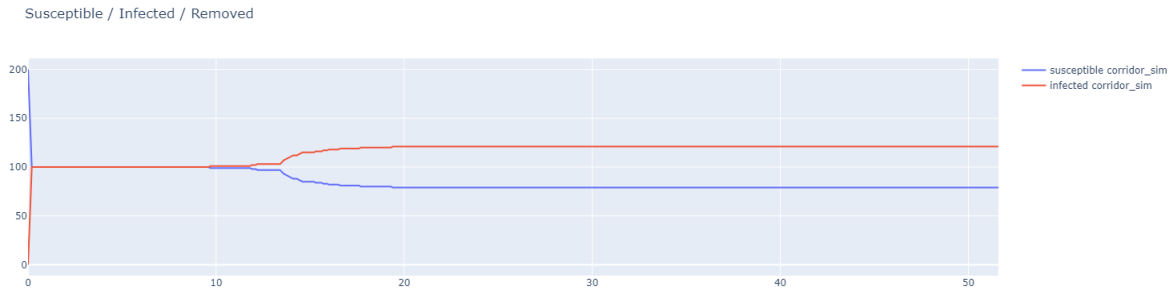


Figure 14: Dash visualization of corridor scenario

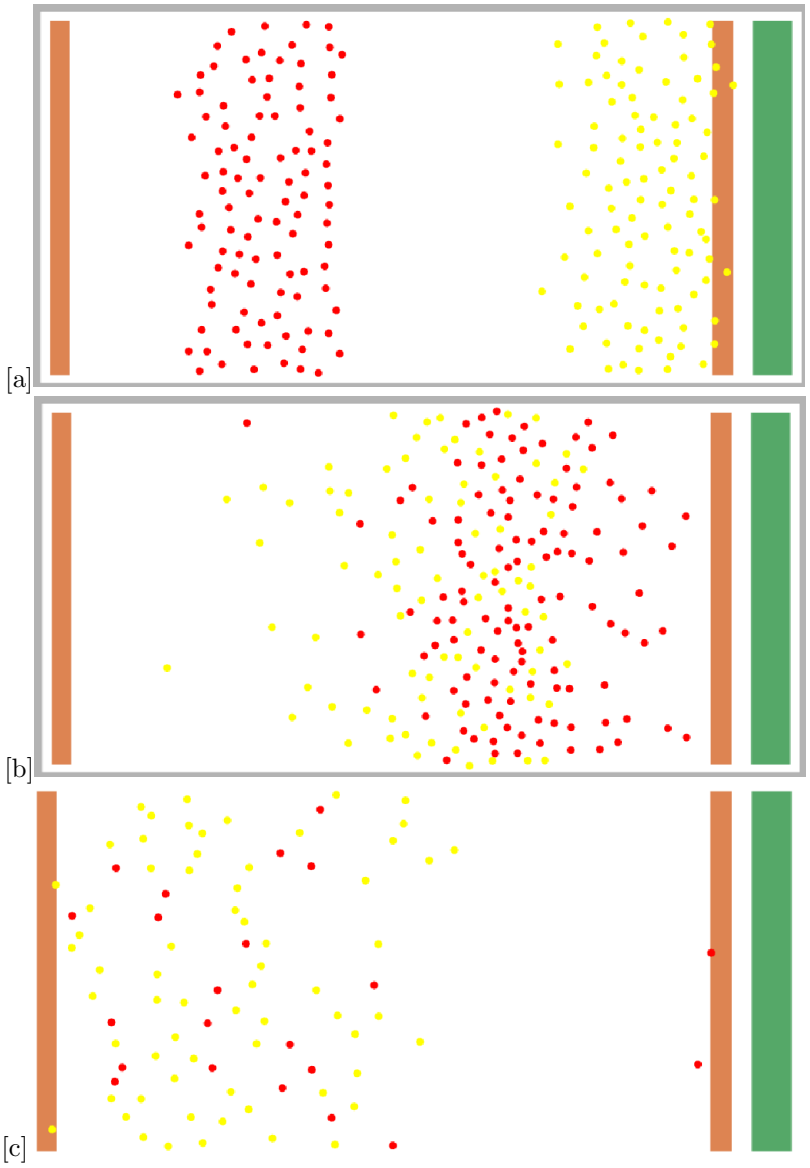


Figure 15: Simulation evolution from left to right.

Even though the simulation behaviour seems to be realistic, an unnatural mechanism is still present. If one were to change the `simTimeStepLength` parameter then a different result comes out of the *corridor scenario*, as well as in any other scenario. The problem stands in the fact that the `SIRGroupModel` update function, responsible for new infections (and later on for new recovered pedestrians), is called at every simulation time step, making it so that the bigger the time step the littler the updates and the probability of new infections, and viceversa. A decoupling between the simulated step length and the `SIRGroupModel` update is possible and should therefore be present.
A possibility for decoupling could be changing the `SIRGroupModel` update frequency such that it becomes independent from the simulated time step. Exploiting the simulated time step range (in Vadere it stands betwen 0.0 and 1.0 excluded), the proposed solution is to update the `SIRGroupModel` only **once per second**, keeping therefore a perhaps higher discretization but giving back nevertheless a detached update from the simulated time step granularity. It is relevant to notice that the *once per second* is an a priori decision which can be modelled to be whatever frequency seems to best fit, the proposed solution is simply an example of which frequency could be chosen.
The solution implementation is relatively simple: `SIRGroupModel` class is not changed at all, the code changes earlier on in the execution hierarchy. The `SIRGroupModel update` method is in fact called by the `Simulation` class via `updateLocomotion` method, which calls periodically (at every simulated time step) for an update of

every model. The proposed solution makes the execution of an update of `SIRGroupModel` possible only if a second has passed from the last call, as can be appreciated in the purple box of the code snippet in **Figure 16**. The effect of this implementation is that no matter what are the `realTimeSimTimeRatio, simTimeStepLength` the simulated behaviour will be definitely more realistic and coherent with the passing time.

```
private void updateLocomotionLayer(double simTimeInSec) {
    for (Model m : models) {
        List<SourceController> stillSpawningSource = this.sourceControllers.stream().filter(s -> !s.isSourceFinished(simTimeInSec))
        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 the model is a SIR simulation then simulate infection/recovery spreading by passing how many real
            // time steps are simulated into a simulated time step for a more realistic and coherent spread.
            // This implementation assumes that a real time step happens at a discretization level of ONE SECOND
            if(m instanceof SIRGroupModel) {
                double simTimeStepLength = this.attributesSimulation.getSimTimeStepLength();
                // execute only one update per second passed
                if((int)(simTimeInSec - simTimeStepLength) < (int) (simTimeInSec))
                    m.update(simTimeInSec);
            }
            else {
                m.update(simTimeInSec);
            }
```

Figure 16: decoupling of `SIRGroupModel` update

At the end of the task, it is interesting to discuss possible extensions that such a malleable model can be adapted to have. Regarding the addition of new states:

- *Dead:* with every probability not the most original idea but nevertheless important and realistic (nowadays more than ever!) in the proposed scenario. Infections can of course lead to death, creating a new pedestrian state where the pedestrian would become static (*no target id*) and non-infective as well as non - infectable.
- *Vaccinated:* once again an ever more realistic feature to represent realistically an infection scenario. It could be in fact plausible to sustain that after a certain period of time vaccines can become available against a certain pathology. Infected, susceptible and recovered could all become vaccinated with a certain rate of possibility. In the vaccinated state pedestrians have a much lowered infection rate, a higher recovery rate and also a much lower death rate, implying the previously proposed rate is already implemented.
- *Immune:* someone is luckier than others, genetics plays an important role in having a different response to pathology exposure. This fact is absolutely present in the human nature, implying the utility of such a state. This state would be assigned at the start of the simulation, and only in that moment, with a certain rate. A pedestrian in the immune state cannot ever become susceptible, infected or recovered.

Another possible idea to make the simulation more realistic and dynamic could be the addition of *aging/pedestrians having different ages to start with.* In reality we have in fact different ages, which makes us on average differently susceptible to illnesses. At an implementation point of view the pedestrian age could be chosen from a distribution and the age would act as a weight on the infection/recovery/death rate. Going more in depth, an aging mechanism could be added so that as time passes the pedestrians get older, therefore dynamically changing their infection/recovery/death rate.

Report on task 5, Analysis and visualization of results

After having integrated the SIR model into Vadere in the previous task, now it has been expanded with new features related to a new state that a pedestrian can assume: **recovered**.
A pedestrian can switch to recovered only from infected, and once reached this state, it cannot change (i.e. it cannot be re-infected or become susceptible to infection).
To add this feature, different steps were necessary and the following modifications have been brought to the following classes/enums:

- `SIRType`: the unused item `ID_REMOVED` has been changed to `ID_RECOVERED`.
- `AttributesSIRG`: the `recoveryRate` attribute has been added, which is the equivalent of `infectionRate` but for pedestrians' recoveries.

- `SimulationModel`:
  - three attributes of type `Color` were added to represent the colors associated with different pedestrians' states: `infectiveColor` (red), `susceptibleColor` (yellow) and `recoveredColor` (green).
  - in the constructor, the parameter that determines the color of the pedestrians has been changed from the target they are pointing to the group they belong to (susceptible, infected or recovered). Furthermore the desired colors for the different groups have been added to the attribute `colorMap`. The updated constructor is shown in Figure 17.
  - the "getter" methods for the newly added attributes (`infectiveColor`, `susceptibleColor` and `recoveredColor`) were added.
- `SIRVisualization` module:
  - `app.py`: very few changes were necessary to this file, in fact we only changed the title of the plot adding the word "Recovered" in the function `update_figure`. Within this function, `create_folder_data_scatter` is called. It is a function contained in `SIRVisualization.utils.py` and this has indeed been modified.
  - `utils.py`:
    - function `file_df_to_count_df`: the attribute `ID_RECOVERED` has been added and the function's body (showed in Figure 18) has been changed in order to consider the recovered pedestrians. In particular these operations are performed: for every pedestrian, save a dataframe [3] called `simtime_group` containing the simulation time and the group id (susceptible/infected/recovered) at that time for the current pedestrian and save the current state as the group to which the pedestrian belongs at the beginning of the simulation (first time step). Then save another dataframe (`group_counts`) containing the again simulation time and then 3 columns - one for each group - containing the number of pedestrians belonging to that group at that time. Finally, start iterating over the rows of the first dataframe (i.e. over the time steps) and:
      - if the pedestrian passes from susceptible to infected, save the current state to infected and for all the next time steps increase the number of infected people and decrease the one of the susceptible people in `group_counts`.
      - if then it passes from infected to recovered, increase the number of recovered people for all the next time steps and decrease the one of infected people. Since the state cannot change from "recovered", exit the iteration over the time steps and go to the next pedestrian.

```
30      public SimulationModel(final T config) {
31          super(config);
32          this.config = config;
33          this.config.setAgentColoring(AgentColoring.GROUP); // to correctly visualize the groups colors
34          this.colorMap = new ConcurrentHashMap<>();
35          this.colorMap.put(-1, config.getPedestrianDefaultColor());
36          // add the desired colors, coupled with the groups ids (0 - infected, 1 - susceptible, 2 - recovered)
37          this.colorMap.put(SIRType.ID_INFECTED.ordinal(), getInfectiveColor());
38          this.colorMap.put(SIRType.ID_SUSCEPTIBLE.ordinal(), getSusceptibleColor());
39          this.colorMap.put(SIRType.ID_RECOVERED.ordinal(), getRecoveredColor());
40          this.random = new Random();
41      }
```

Figure 17: The updated constructor of `SimulationModel`. At line 33 the argument of `config.setAgentColoring` has been changed from `AgentColoring.TARGET` to `AgentColoring.GROUP`. Lines 37, 38 and 39 set the desired colors for the 3 states in the `colorMap`.

```python
for pid in pedestrian_ids:
    # dataframe containing, for each time step in the simulation, the group to which the pedestrian (pid) belongs
    simtime_group = df[df['pedestrianId'] == pid][['simTime', 'groupId-PID5']].values
    current_state = simtime_group[0][1]     # save current pedestrian state (susceptible / infected / recovered)
    col_name = 'group-'
    col_name += 's' if current_state == ID_SUSCEPTIBLE else ('r' if current_state == ID_RECOVERED else 'i')
    group_counts.loc[group_counts['simTime'] >= 0, col_name] += 1
    # iterate over the simulation time steps
    for (st, g) in simtime_group:
        # if the pedestrian passes from infected to recovered
        if g == ID_RECOVERED and current_state == ID_INFECTED:
            current_state = ID_RECOVERED
            # for each future time step, increase the number of recovered and decrease the one of infected
            group_counts.loc[group_counts['simTime'] >= st, 'group-r'] += 1
            group_counts.loc[group_counts['simTime'] >= st, 'group-i'] -= 1
            break   # the pedestrian cannot change state once it's recovered
        # if the pedestrian passes from susceptible to infected
        elif g == ID_INFECTED and current_state == ID_SUSCEPTIBLE:
            current_state = ID_INFECTED
            # for each future time step, increase the number of infected and decrease the one of susceptible
            group_counts.loc[group_counts['simTime'] >= st, 'group-i'] += 1
            group_counts.loc[group_counts['simTime'] >= st, 'group-s'] -= 1

    # original implementation
    # 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
    #         break

return group_counts
```

Figure 18: The body of the function `file_df_to_count_df`. The commented lines show the code previous to the modifications introduced to take the recovered pedestrians into account.

To test the implementation of this new state ("recovered"), the same test presented for the previous task has been performed: 1000 pedestrians are placed on a grid, 990 susceptible and 10 infected. They do not move but they can infect each other according to the SIR model with the addition of the "recovered" state.
This test has been run with different configurations for the infection rate and the recovery rate. Some plots are available on **Figure 20** and some screenshots of the execution of the scenarios are present in **Figure 19**. The different rates configurations are:

1. Infection rate equal to recovery rate (actual value: 0.01): here the infection is well contrasted by the recoveries, in fact in spreads pretty slowly among pedestrian. The infection is finally blocked and by the end of the run the situation stabilizes having slightly less than 400 pedestrians that have not been infected. The final number of recovered is 600 ca., meaning that more than half of the population has been infected.

2. Infection rate double than the recovery rate (actual values: 0.02 and 0.01): in **Figure 20(b)** it is possible to see how, by the end of the simulation, almost everyone had been infected, because the number of recovered people is near 1000 (i.e. the whole population). Recall that in order to be recovered, one must have been infected first.

3. Infection rate half of the recovery rate (actual values: 0.01 and 0.02): in this third case, the infection starts spreading, but it gets controlled by the fast recovery rate of the pedestrians. By the end, only slightly more than 200 people (20% of the population) got infected.

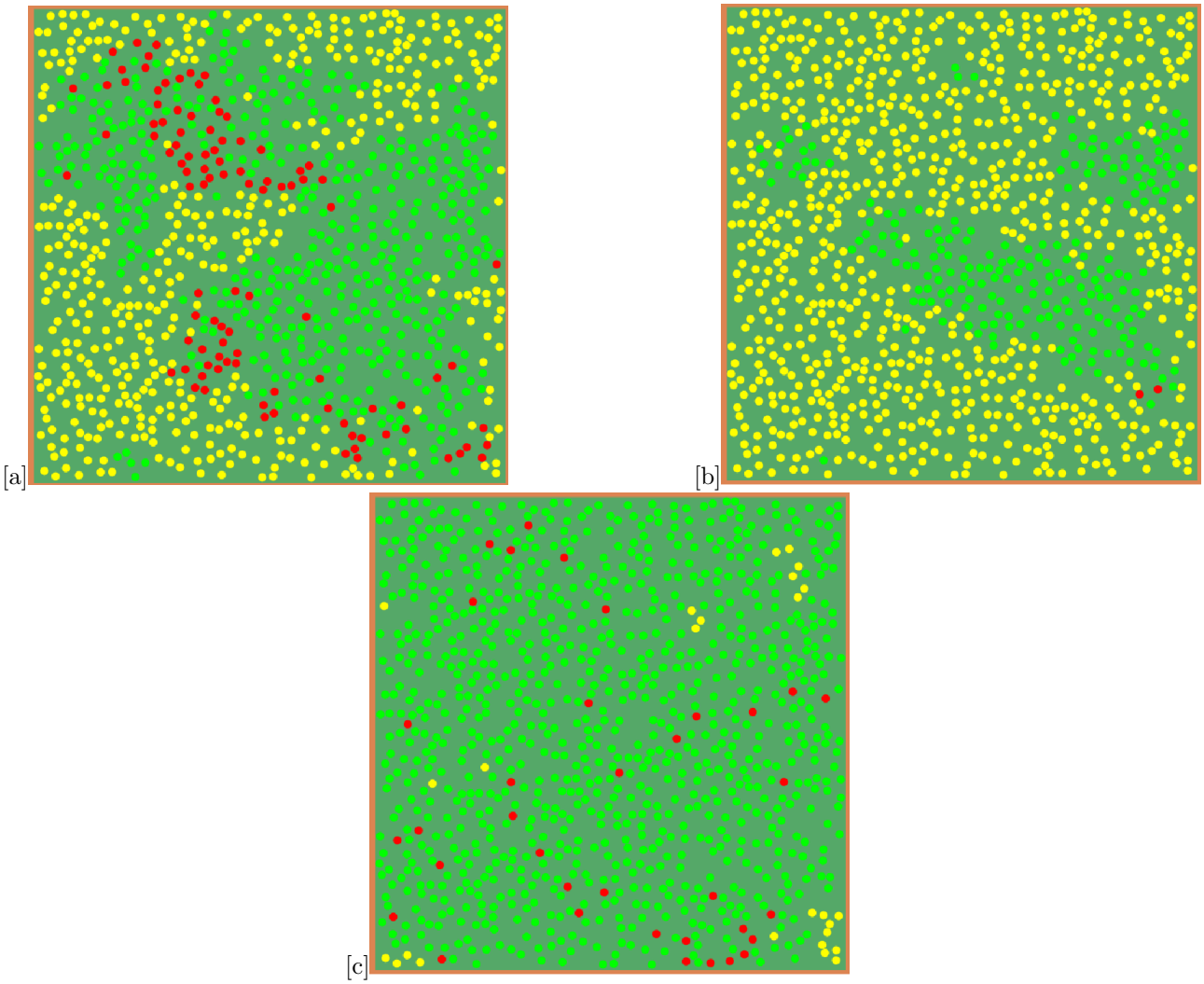


Figure 19: Screenshots of the scenarios while run with different infection and recovery rates, namely: (a) infection rate = 0.1 and recovery rate = 0.1; (b) infection rate = 0.1 and recovery rate = 0.2; (c) infection rate = 0.2 and recovery rate = 0.1

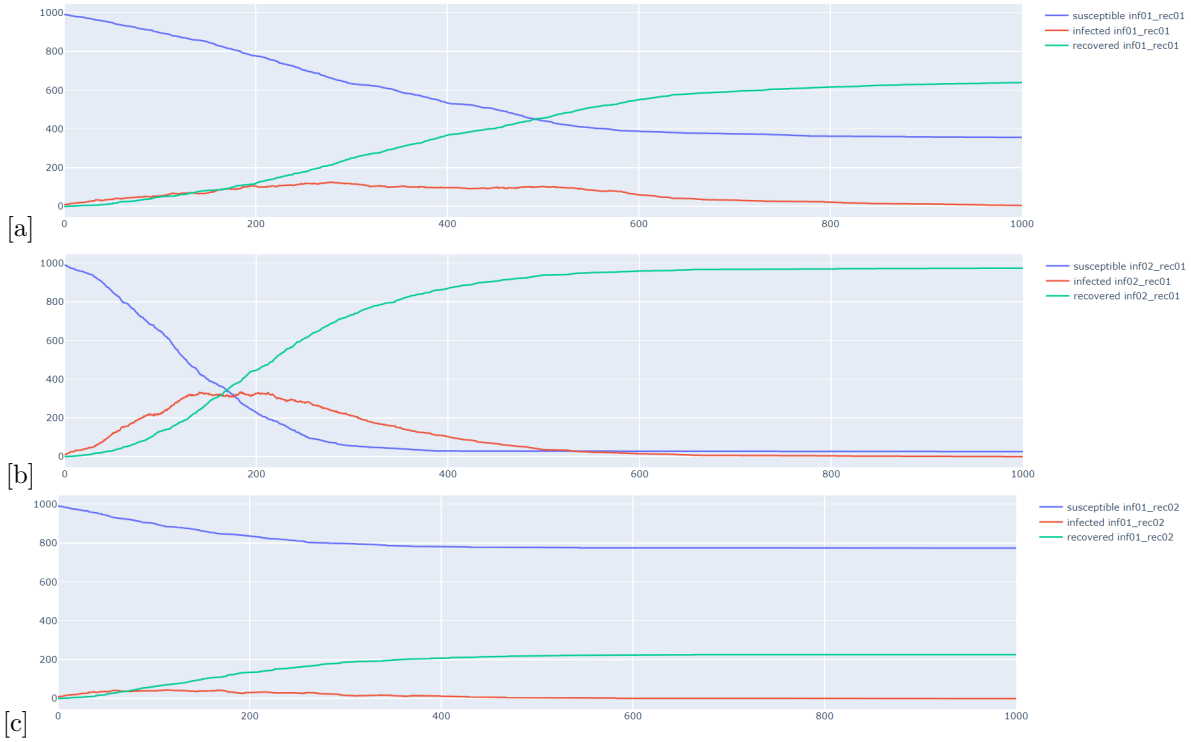


Figure 20: Evolution of the number of susceptible, infected and recovered pedestrian with (a) infection rate = 0.1 and recovery rate = 0.1; (b) infection rate = 0.2 and recovery rate = 0.1; (c) infection rate = 0.1 and recovery rate = 0.2

Eventually one last test was carried on to verify how the model behaves in a real world scenario of infection: a supermarket. The supermarket is in fact a plausible site of disease spreading, where careful considerations should be made to avoid gatherings or a too reduced interpersonal space. The proposed supermarket scenario is the one proposed in **Figure 21(a)**.

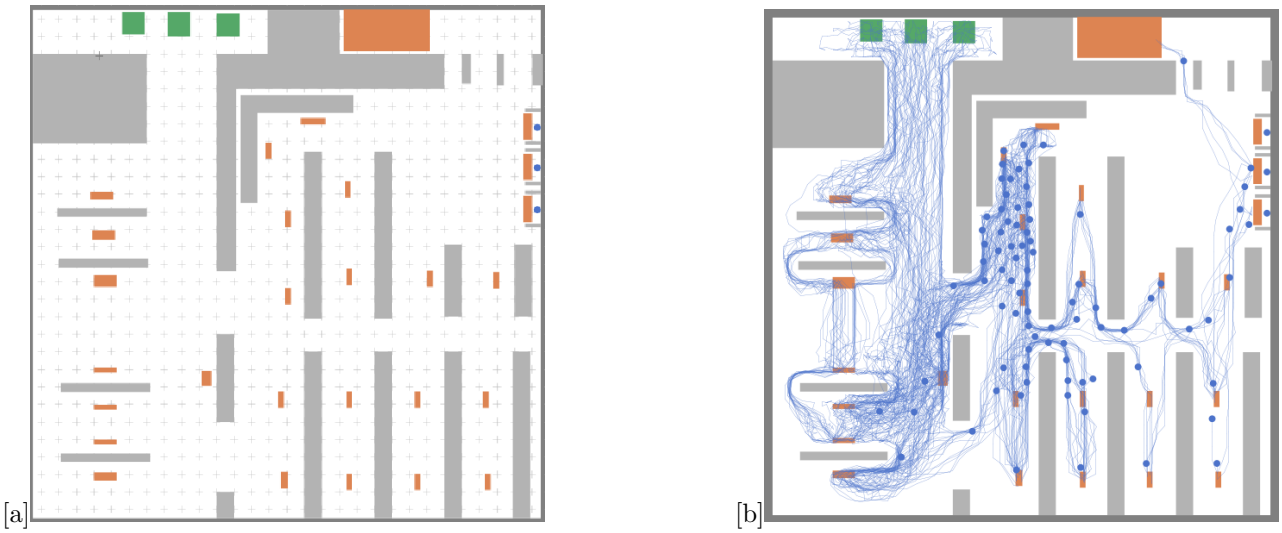


Figure 21: On the left the supermarket scenario, on the right a depiction of a possible crowded moment

Making the scenario as realistic as possible is fundamental for a meaningful test. The pedestrian sources have been modified in their standard parameters so that they spawn pedestrians in groups of **3** for a certain time period (the first **150** seconds out of the total **400** simulated) following a spawning probability dictated by the `NegativeExponentialDistribution` parameter value. This was done to assure that not all pedestrians are

spawned immediately (customers do not enter all at once in the supermarket, except for the American Black Friday), but that instead there is a continuous flow into the supermarket, not too empty and not too hectic, as a normal day would be.

To make the scenario more realistic the three sources have been given three different routes to take, visiting some aisles and then going to one of the cash desks (where the cashiers are on the top right). They finally end their shopping by exiting through the bigger target on top. This was done to assure not all pedestrians follow the unrealistic rule of having the same exact supermarket route.

Two different tests are proposed in the following paragraphs:

- **TEST A:** a scenario where the disease is not highly infective (**infectionRate: 0.02**) and the recovery rate is fairly low (**recoveryRate: 0.001**, more improbable, it would in fact be a very strange disease if in the course of a single shopping trip people get both infected and cured). The analysis shows the differences if no regulation on interpersonal distance is applied vs if **2.0** meters have to be kept between pedestrians when possible (as happened for a certain period in the current pandemic in Italy).
- **TEST B:** a scenario with a much more infective disease (**infectionRate: 0.10**) and the same recovery rate. The analysis shows how in such a scenario the social distancing is not enough because of the presence of bottlenecks in the supermarket, asking also for an entrance flow management (e.g. changing the spawned pedestrians per source from **3** to **2**).

Regarding TEST A, **Figure 22** shows the development of the simulation through some snapshots. As a reminder the simulation has the following parameters:

- **infectionsAtStart:** 5
- **infectionRate:** 0.02
- **infectionMaxDistance:** 1.0
- **recoveryRate:** 0.001
- **pedPotentialPersonalSpaceWidth:** 0.9
- **pedestrians per source per spawning:** 3

The simulation shows how the infection spreads quite fast among the customers, with the infected being able to exploit the gathering shown in **Figure 22 (a) and (b)**. The infectivity of the disease is not too high and therefore not all customers get infected.

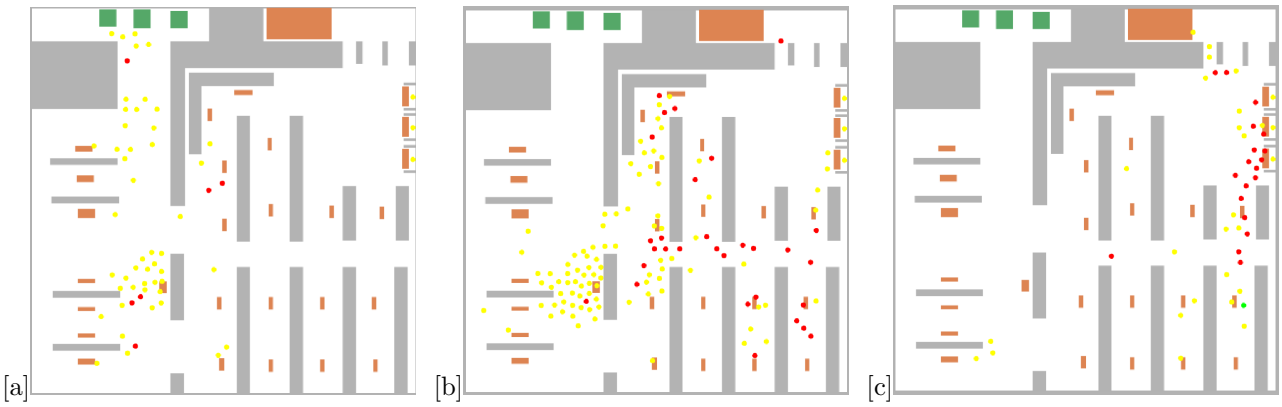


Figure 22: Screenshots taken from the first test, having a **0.9** interpersonal space

A solution to the infective behaviour stands in applying some rules we are nowadays used to, for example *Social Distancing*. What is shown in **Figure 23** is the development of a comparative simulation where the parameters are:

- **infectionsAtStart:** 5

- **infectionRate:** 0.02
- **infectionMaxDistance:** 1.0
- **recoveryRate:** 0.001
- **pedPotentialPersonalSpaceWidth: 2.0**
- **pedestrians per source per spawning:** 3

Therefore one can notice how only the personal space width has changed, forcing the pedestrians to keep their distances when possible. The effect can be visually appreciated, with the infections being drastically reduced, showing the effectiveness of this approach.

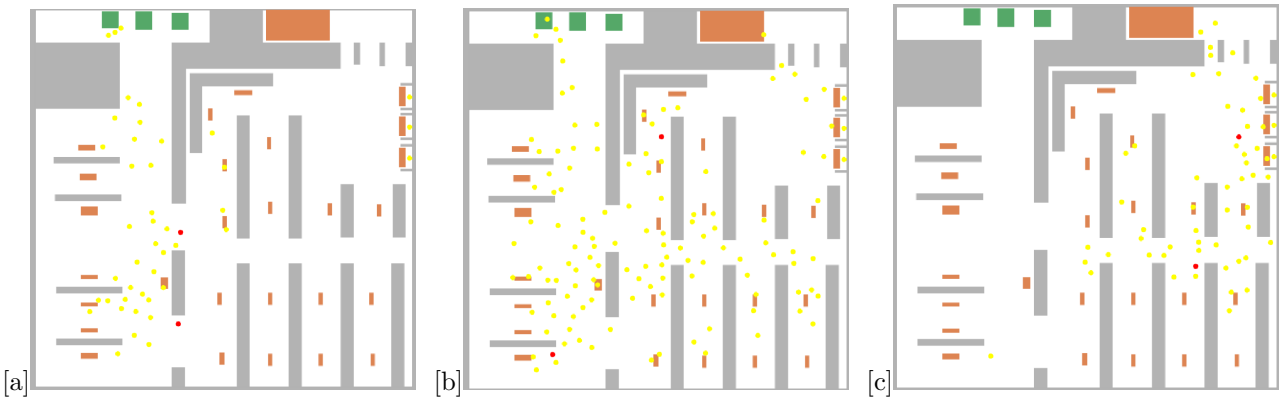


Figure 23: Screenshots taken from the first test, having a **2.0** interpersonal space

Eventually, in **Figure 24**, a graphical comparison between the two simulations for TEST A is shown through *Dash/Plotly*. In particular `test_1_1` represents the first simulation and `test_1_2` the second one, remarking again the effectiveness of increasing the social distancing in having less infections.

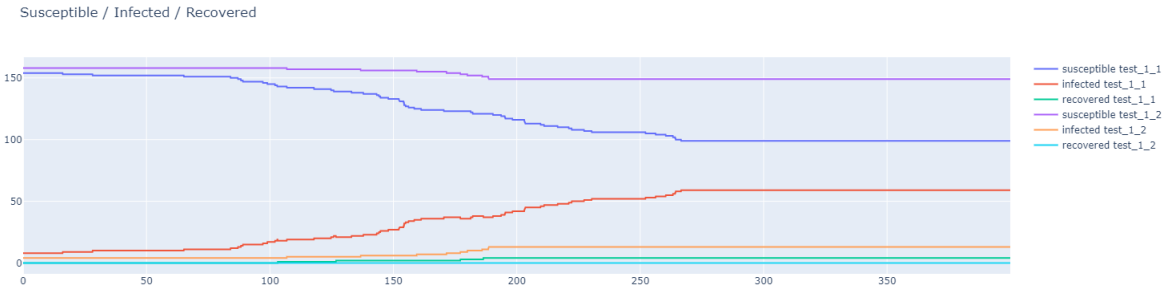


Figure 24: Graphic visualization using *Dash/Plotly* of the first test

Moving on to TEST B, a different scenario is portrayed. Here a much higher infection rate is present, causing the social distancing to be less efficient. The test proposes a change in the entry flow of the supermarket which in combination with social distancing proves to be working better. **Figure 25** refers to the TEST B scenario where no precautions are taken into consideration. The simulation parameters are:

- **infectionsAtStart:** 0
- **infectionRate:** 0.10
- **infectionMaxDistance:** 1.0
- **recoveryRate:** 0.001

- **pedPotentialPersonalSpaceWidth: 1.0**
- **pedestrians per source per spawning:** 3

The simulation shows how the infection spreads incredibly fast among the customers, reaching almost all the customers and also infecting one of the poor cashiers. The higher number of infected in this experiment gives rise also to some recovered popping up somewhere in the simulation. One example is the cashier who has a miracolous infection and recovery during a single working shift, impressive!
Such a situation should not happen and this leads to the second part of the TEST B.

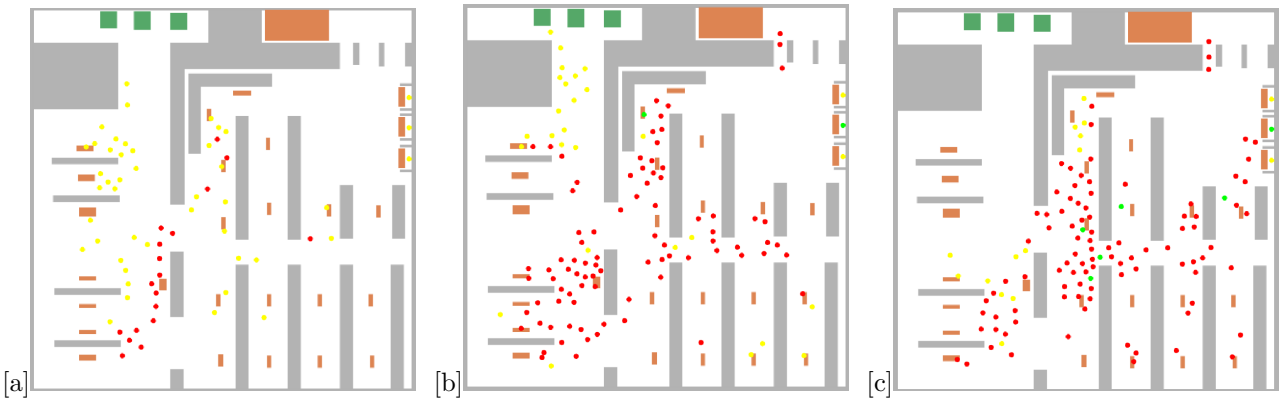


Figure 25: Screenshots taken from the second test, having a **1.0** interpersonal space

A possible mitigation of the infective behaviour stands in applying as previously mentioned some rules. What is shown in **Figure 23** is the development of a comparative simulation where the parameters are:

- **infectionsAtStart:** 0
- **infectionRate:** 0.1
- **infectionMaxDistance:** 1.0
- **recoveryRate:** 0.001
- **pedPotentialPersonalSpaceWidth: 2.0**
- **pedestrians per source per spawning:** 2

It is possible to notice how the combination of both a higher inter-person space and a slower entrance flow gives a considerably better result, with littler infections. It is interesting to notice that infections are anyhow existent. This is due to the higher infection rate: if the infection rate is high it is possible to get infected even in a smaller gathering time window, well simulating a more aggressive disease. The responsibility is not only on that argument, but also partly on Vadere: the software simulation has the chance to spawn infected pedestrians, and having a higher infection rate helps this process.

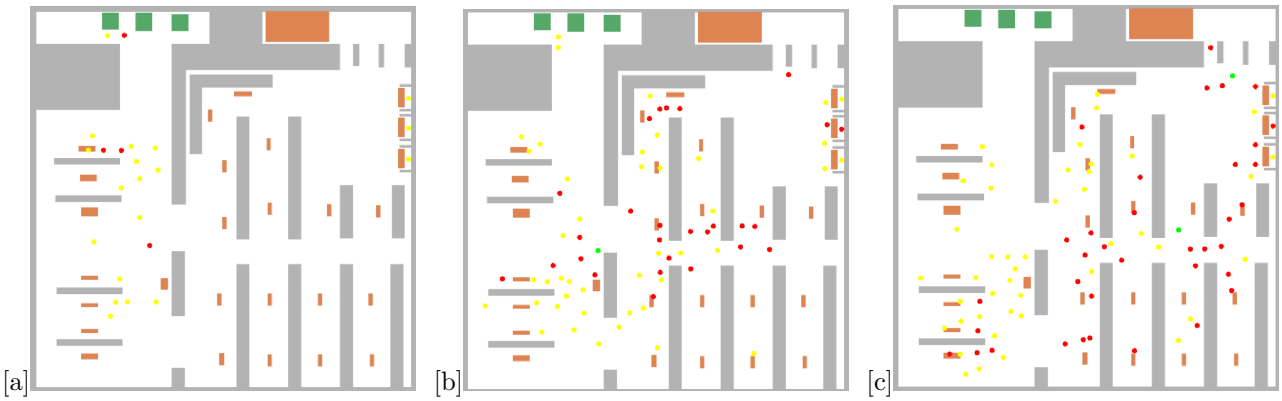


Figure 26: Screenshots taken from the second test, having a **2.0** interpersonal space and lower entrance rate

Finally, as for the TEST A, a *Dash/Plotly* graphic in **Figure 27** is shown to better portray the comparison. In particular `test_2_1` represents the first simulation and `test_2_2` the second one.
It is important to remark that the simulation where all precautions are taken into consideration effectively spawns a lower amount of pedestrians (lower entrance flow, same execution time), therefore the plot has to be analyzed with that in mind. The graphic does nevertheless show how a combination of precautions does help the decrease in spreading even against a more aggressive disease.

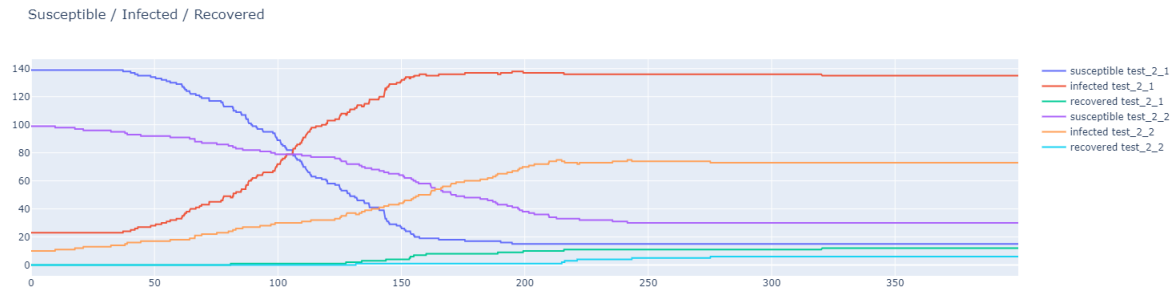


Figure 27: Graphic visualization using *Dash/Plotly* of the second test

References

[1] Java's `enum` type. `https://docs.oracle.com/javase/tutorial/java/javaOO/enum.html`.

[2] Oracle's jdk download webiste. `https://www.oracle.com/de/java/technologies/javase/jdk11-archive-downloads.html`.

[3] Pandas' dataframe. `https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.html`.

[4] Rimea guidelines. `https://rimeaweb.files.wordpress.com/2016/06/rimea_richtlinie_3-0-0_-_d-e.pdf`.

[5] Vadere. `https://www.vadere.org/`.

[6] Vadere releases webiste. `http://www.vadere.org/releases/`.

[7] Felix Dietrich and Gerta Köster. Gradient navigation model for pedestrian dynamics. *Physical Review E*, 89:062801, 06 2014.

[8] Felix Dietrich, Gerta Köster, Michael Seitz, and Isabella Sivers. Bridging the gap: From cellular automata to differential equation models for pedestrian dynamics. *Journal of Computational Science*, 5, 09 2014.

[9] Dirk Helbing, Illés Farkas, and Tamás Vicsek. Simulating dynamic features of escape panic. *Nature*, 407:487–490, 09 2000.

[10] Dirk Helbing and Peter Molnar. Social force model for pedestrian dynamics. *Physical Review E*, 51, 05 1998.

[11] Michael Seitz and Gerta Köster. Natural discretization of pedestrian movement in continuous space. *Physical review. E, Statistical, nonlinear, and soft matter physics*, 86:046108, 10 2012.

[12] Isabella Sivers and Gerta Köster. Dynamic stride length adaptation according to utility and personal space. *Transportation Research Part B: Methodological*, 74, 04 2015.