

Report for exercise 2 from group G

Tasks addressed: 5
Authors: Zhaozhong Wang (03778350)
Anastasiya Damaratskaya (03724932)
Bassel Sharaf (03794576)
Thanh Huan Hoang (03783022)
Celil Burak Bakkal (03712329)
Last compiled: 2024-11-14

The work on tasks was divided in the following way:

Zhaozhong Wang (03778350)	Task 1	20%
	Task 2	20%
	Task 3	20%
	Task 4	20%
	Task 5	20%
Anastasiya Damaratskaya (03724932)	Task 1	20%
	Task 2	20%
	Task 3	20%
	Task 4	20%
	Task 5	20%
Bassel Sharaf (03794576)	Task 1	20%
	Task 2	20%
	Task 3	20%
	Task 4	20%
	Task 5	20%
Thanh Huan Hoang (03783022)	Task 1	20%
	Task 2	20%
	Task 3	20%
	Task 4	20%
	Task 5	20%
Celil Burak Bakkal (03712329)	Task 1	20%
	Task 2	20%
	Task 3	20%
	Task 4	20%
	Task 5	20%

Report on task 1: Setting Up the Vadere Environment

Vadere Setup

Task 1 concentrates on recreating previous scenarios, like simulating a corridor with movement trajectory as a straight line or crowd movement around the corner, as well as the chicken test, where the U-shaped obstacle is placed between pedestrians and targets. In order to compare the results of both simulations, first with cellular automaton and then using Vadere software¹, we need to set up the graphical user interface of Vadere and recreate the scenarios. For that, after cloning the Artemis project, we ran all provided commands and opened the user interface. After successfully creating a new project, we created scenarios using the **Optimal Steps Model (OSM)** as a model template for Task 1. Next, we used **Topography Creator** to place obstacles, pedestrians, targets, and measuring points in the scenarios. We adjusted the IDs in the JSON files accordingly. After the visualization was completed, we ran the scenarios and saved *.scenario files to the **scenario/** folder of our project. To save the files, we chose a needed output file, right-clicked, and chose the option **Generate scenario**, where we selected a folder to save a created configuration file. For visualizing our results, we used **!Snapshot!** button in the **Post-Visualization** tab and created screenshots for the start and end positions of pedestrians in the simulation for each scenario.

Vadere vs. Cellular Automata

The main difference from the previous implementation of cellular automata from Exercise 1 is that pedestrians in Vadere do not have a body size the same as that of a cell. Seitz et al.[3] mentioned the direct dependence of crowd dynamics on pedestrian representation: in the case of cellular automata, the pedestrians are represented by the cells, which means the dimension of cells equals the dimension of pedestrians' torsos, and there cannot be any variation in size and shape among different individuals. This makes compression of the pedestrians impossible, which, on the other hand, should be expected in the dense crowd. Another issue of the cellular automata simulation is that pedestrians step from cell to cell, and it does not allow for movement in arbitrary directions. On the contrary, in Vadere, the pedestrians have a round form and are smaller than a cell size, which solves both named issues. It demonstrates a more realistic crowd dynamic and, therefore, produces more accurate results and predictions.

Optimal Steps Model (OSM)

Additionally, Vadere allows one to choose between different models for simulating crowd dynamics. OSM, used as a model template in Vadere for task 1, is based on the cellular automata in the sense that for choosing a movement direction, it uses attractive and repulsive potentials, and allows speed adjustments for maintaining a group structure[3]. At each time frame, OSM optimizes individual step choices for each pedestrian by evaluating potential future steps in order to minimize costs.

RiMEA Scenarios 1: Maintaining the specified walking speed in a corridor

RiMEA² scenario 1 focuses on evaluating if a person passing a corridor moves with the correct speed. To recreate a scenario from Exercise 1, Task 5, we placed obstacles in the form of a corridor and, in between, placed a blue pedestrian on the left and an orange target on the right, as demonstrated in the Figure 1. For better visualization, we saved the movement trajectory to thoroughly compare the simulation outcomes from both cellular automata and OSM model of Vadere.

Due to the limitations based on the same cell and pedestrian size in the cellular automata, the movement trajectory was a straight line for scenario 1. However, for OSM, the line is not straight, possibly due to optimizations after each step and the influence of the obstacle's walls nearby. Other than that, the configuration file of pedestrians in Vadere includes more attributes for the speed than in cellular automata (which only has **speed**), such as **densityDependentSpeed**, **speedDistributionMean**, **speedDistributionStandardDeviation**, **minimumSpeed**, and **maximumSpeed**. These new properties improve crowd-modeling simulations and future predictions since they can be flexibly adjusted for any scenario.

¹<http://www.vadere.org/releases/>

²https://rimea.de/wp-content/uploads/2016/06/rimea_richtlinie_3-0-0-_d-e.pdf

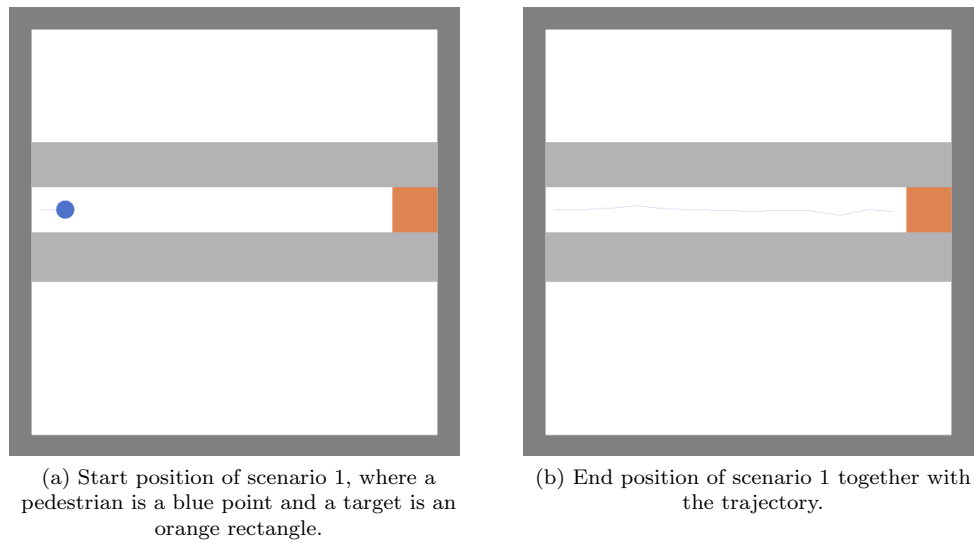


Figure 1: Visualization of scenario 1 using Vadere representing a corridor with a pedestrian and a target.

RiMEA Scenarios 6: Movement around a corner

RiMEA scenario 6 evaluates how the crowd passes the corner. For that, the target is set to be not absorbing to better depict congestion in the scenario after reaching a target. Similar to Exercise 1, Task 5, 20 blue pedestrians are placed in the upper left corner, aiming to reach an orange target located in the bottom right corner (Figure 2).

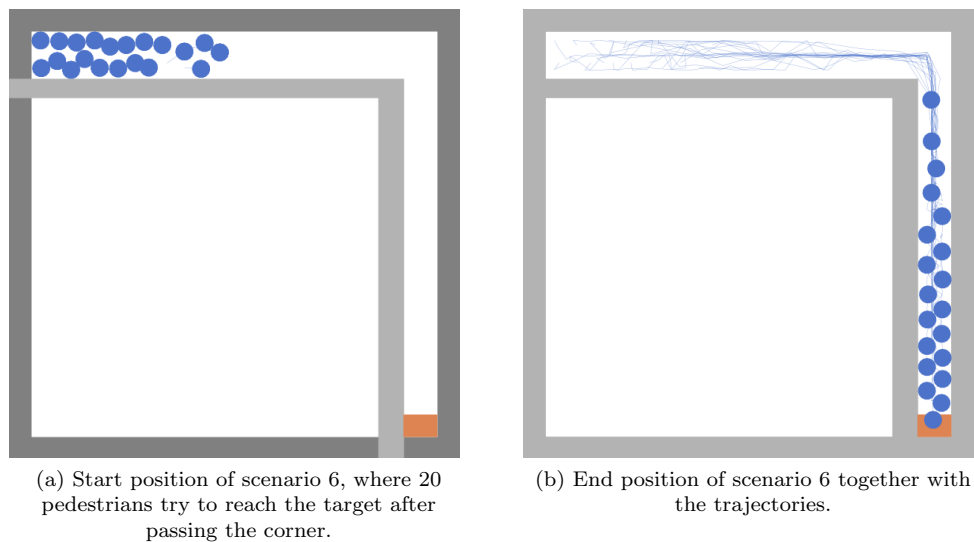


Figure 2: Visualization of scenario 6 using Vadere representing a corner scenario with multiple pedestrians.

The branching trajectory at the beginning of the scenario is similar to the cellular automata; however, in the case of OSM, the trajectories converge at the same place before the beginning of the corner, while in cellular automata simulation, the convergence happens first after entering the corner. Furthermore, since pedestrians are smaller than cells, the trajectory after entering a corner is not a straight line since pedestrians constantly try to optimize and reduce the distance to the target if possible.

One of the main observations was that the pedestrians continued to try to move closer, minimizing the space between each other as much as possible. Since the pedestrians in the cellular automata were not round and had

no ability to move closer, we could not observe this behavior. Interestingly, the last pedestrians in the crowd were still keeping their distance from each other in comparison to the nearest pedestrians to reach the target, which continued moving until it was not possible anymore. Nevertheless, neither OSM nor cellular automata permit the overlapping of pedestrians or pedestrians and obstacles.

Chicken Test

The chicken test is not directly a part of RiMEA guidelines but still provides an excellent evaluation of obstacle avoidance in the simulation. We located a U-shaped obstacle in the middle of the simulation, right before a pedestrian, to test how it would avoid the obstacle and reach its target (Figure 3).

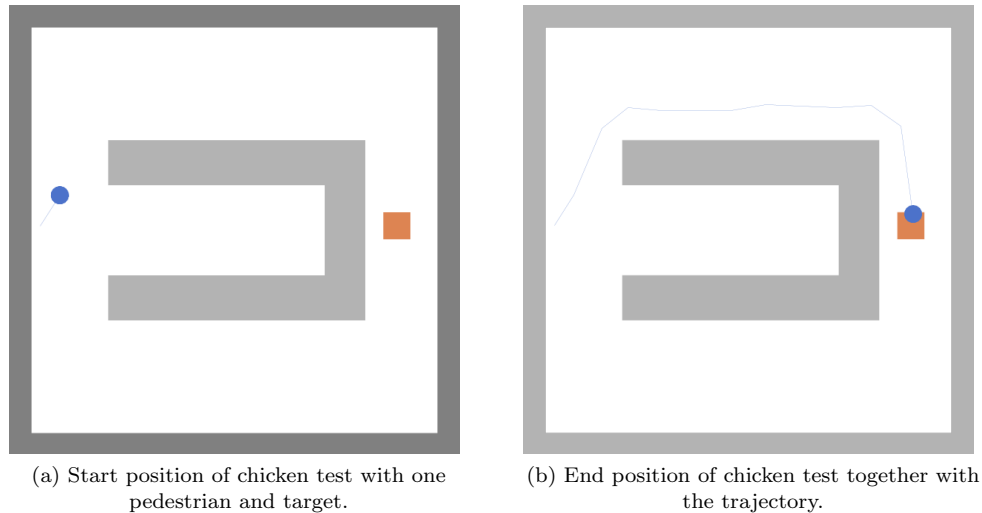


Figure 3: Visualization of chicken test.

In comparison to the cellular automata, which supports pedestrians and obstacles in the neighboring cells, pedestrians in the OSM constantly keep a distance from the obstacles in order to avoid collisions. The trajectory bends smoothly around one 'arm' of the U-shaped obstacle and, after completely passing the obstacle, proceeds directly toward the target.

In general, Vadere provides a more flexible user interface with many helpful features, such as showing trajectories, screenshots, and other visualization options, as well as the ability to choose between multiple models suitable for various simulation scenarios. Cellular automata is a simpler and more restricted version, which is computationally straightforward but lacks the fluidity of continuous models and path variability.

Report on task 2: Simulation With a Different Model

In Task 2, we aim to compare the simulation results of three scenarios using different models: the OSM from above, the Social Force Model (SFM), and the Gradient Navigation Model (GNM). SFM defines pedestrian behavior using several terms that describe the acceleration towards the desired velocity of motion, reflect the maintenance of distance between pedestrians and borders, and model attractive effects [2]. The GNM, on the other hand, changes the direction of the velocity vector, which is directly integrated to obtain the location, using a superposition of gradients of distance functions [1]. Similar to the cellular automata simulation, pedestrians want to reach their target in as few steps as possible and steer directly toward the direction of the steepest descent, avoiding overlapping with each other. Table 1 compares the results of three scenarios from Task 1, providing a better analysis of these two models.

Starting with scenario 1, we immediately notice the difference in the movement trajectory of OSM. In both models, pedestrians move in a straight trajectory toward their target: in the case of SFM, due to the force

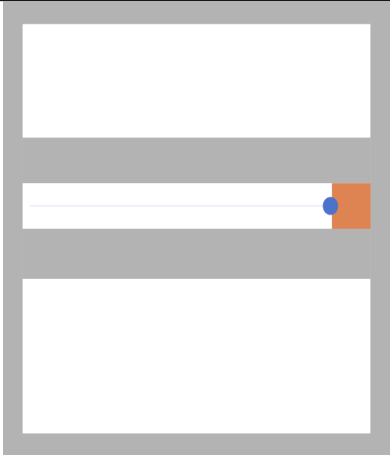
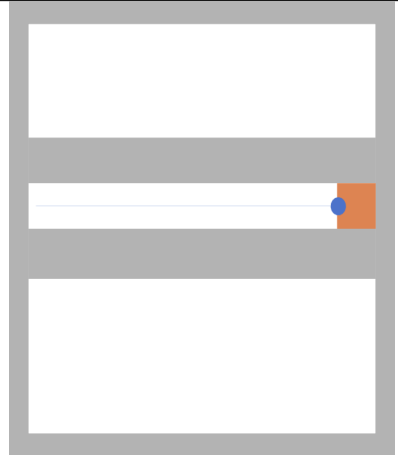
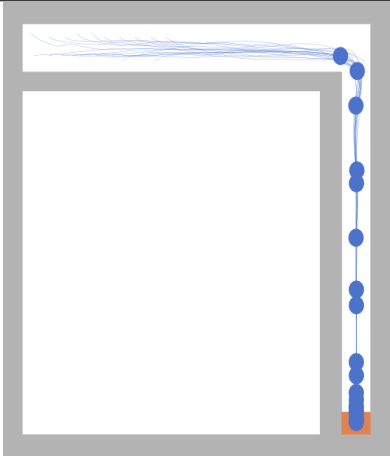
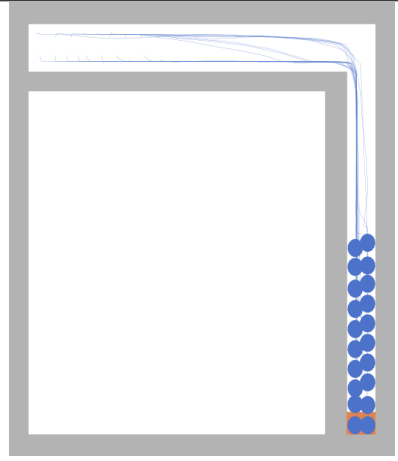
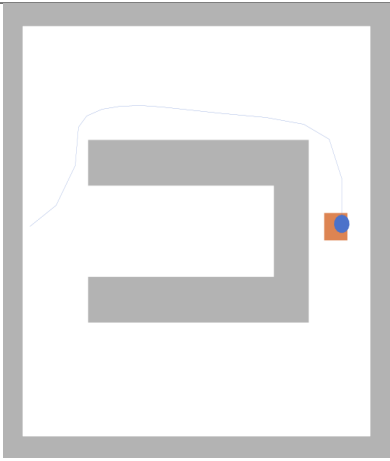
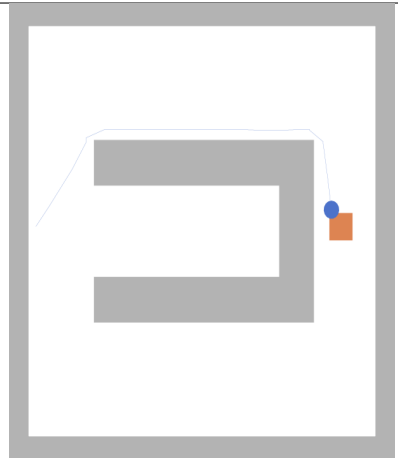
Scenario	Social Force Model	Gradient Navigation Model
RiMEA Scenario 1: Maintaining the specified walking speed in a corridor		
RiMEA Scenario 6: Movement around a corner		
Chicken Test with the U-shaped obstacle		

Table 1: Comparison between scenarios and models used.

aiming it forward, and in the case of GNM, due to following the path defined by the gradient field.

For the movement around the corner, all three models produce different results due to the interaction between pedestrians. By SFM, the trajectory curve is smoother to the corner than in OSM or GNM, but the simulation contains delays. Even after a longer wait, not all pedestrians formed an unbreakable queue to the target. Instead, they maintained a distance from each other. Due to the attractive force pulling pedestrians to the target, the trajectory after passing the corner is mostly straight. However, repulsive forces in the form of interaction with other pedestrians cause slight deviations from the straight line to the target. Furthermore, the SFM simulation produces overlapping between pedestrians, which should be avoided. On the contrary, in the GNM, since the pedestrians were placed in two lines before moving (Figure 2), two trajectory curves can be observed until the corner. After identifying the corner, the trajectories mostly converge to one, not in the middle as in OSM, but near the inner obstacle wall until there is no space anymore. To reduce the distance, they move to the right and press against each other. This behavior can also lead to slight overlaps between pedestrians. Only in OSM was no overlapping identified. The optimization after each step enables faster path corrections and efficient adjustments to obstacles or other pedestrians.

Lastly, the results of the chicken test also differ. In SFM, a pedestrian has a curved trajectory influenced by the repulsive forces from the U-shaped obstacle, which, after passing the first corner, decreases the distance to the obstacle until passing the second corner due to the attractive force. Even without obstacles in front, the pedestrian proceeds with the curved movement to avoid any possible collision with the obstacle. In GNM, the trajectory follows a contour of the obstacle, leaving a slight distance to it. Moving closer to the obstacle keeps the pedestrian within the lowest gradient path available without crossing into a high-potential zone and minimizes the potential gradient. Similar to the GNM trajectory, in OSM, the path closely follows the U-shaped contour but at a greater distance from the obstacles. The model ensures that pedestrians maintain a minimal distance while avoiding overlaps and collisions.

All three models have advantages and disadvantages when simulating pedestrian trajectories and should be used depending on the current goal of the simulation. SFM is, for example, suitable for capturing realistic social interactions through its force-based approach, where pedestrians respond to attractive and repulsive forces to maintain their personal space and avoid obstacles around them. We observed it in all scenarios with enough distance between pedestrians and obstacles. However, the interaction with other pedestrians guaranteed delays in reaching the target. GNM, on the other hand, navigates using potential fields that guide pedestrians along contours toward their targets, which ensures a straightforward, continuous trajectory but often lacks collision avoidance and ignores overlapping. OSM differs with its step-by-step optimization, which results in highly adaptive and efficient trajectories that minimize distance and avoid overlaps and collisions. However, the model may require higher computational resources due to constant movement adjustments. To summarize, SFM offers more realistic social behavior, while GNM is fast, simple, and smooth. OSM, on the other hand, achieves efficient and adaptable movement, though at a higher computational cost compared to the other two.

Report on task 3: Using the Console Interface From Vadere

Running Vadere Through Its Console Interface

Instead of simulating RiMEA Scenario 6 using the Vadere GUI, we run Vadere from the command line. During execution, the simulation's progress is printed in the console (see Figure 4). During execution, it checks JSON version compatibility, logs initialization details, and issues warnings about unused or overlapping targets. The simulation calculates floor fields and generates output files (`postvis.traj`, `overlaps.csv`, and `overlapCount.txt`) with progress updates. Once finished, it logs the successful completion of the simulation and the locations of output files.

The output files generated by using the command line are then compared to the output files produced by running the scenario in the GUI, using the `diff` command in the macOS terminal. Figure 5 shows the execution of the `diff` command. The folder `Console` contains all output files from simulating the scenario via the command line, while `GUI` holds all output files from running Vadere with the GUI. These respective folders can be

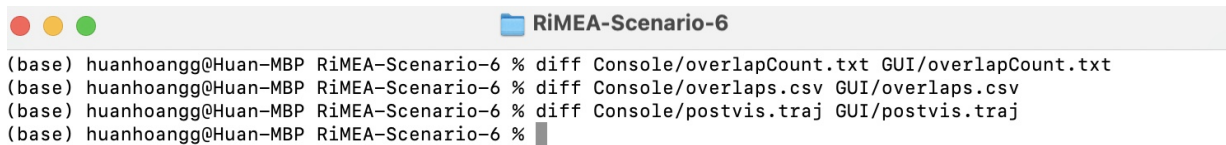
```

20:23:29,149 INFO ScenarioRun:147 - Initializing scenario: test...
20:23:29,152 INFO ScenarioRun:128 - scenario context initialized.
20:23:29,153 INFO ScenarioRun:229 - no mesh test.poly was found.
20:23:29,153 INFO ScenarioRun:229 - no mesh test_background.poly was found.
20:23:29,178 INFO PotentialFieldDistancesBruteForce:49 - solve floor field (PotentialFieldDistancesBruteForce)
20:23:29,181 INFO PotentialFieldDistancesBruteForce:106 - Progress: 0% -> 0/10201 [points]
20:23:29,189 INFO PotentialFieldDistancesBruteForce:106 - Progress: 10% -> 1021/10201 [points]
20:23:29,192 INFO PotentialFieldDistancesBruteForce:106 - Progress: 20% -> 2041/10201 [points]
20:23:29,195 INFO PotentialFieldDistancesBruteForce:106 - Progress: 30% -> 3061/10201 [points]
20:23:29,199 INFO PotentialFieldDistancesBruteForce:106 - Progress: 40% -> 4081/10201 [points]
20:23:29,203 INFO PotentialFieldDistancesBruteForce:106 - Progress: 50% -> 5101/10201 [points]
20:23:29,207 INFO PotentialFieldDistancesBruteForce:106 - Progress: 60% -> 6121/10201 [points]
20:23:29,211 INFO PotentialFieldDistancesBruteForce:106 - Progress: 70% -> 7141/10201 [points]
20:23:29,213 INFO PotentialFieldDistancesBruteForce:106 - Progress: 80% -> 8161/10201 [points]
20:23:29,215 INFO PotentialFieldDistancesBruteForce:106 - Progress: 90% -> 9181/10201 [points]
20:23:29,218 INFO PotentialFieldDistancesBruteForce:96 - floor field initialization time:37[ms]
20:23:29,221 INFO Topography:414 - Set PedestrianIdCount to start value: 101
20:23:29,238 INFO EikonalSolverDefaultProvider:23 - floor field initialization time:7[ms]
20:23:29,238 INFO Simulation:277 - preLoop finished.
20:23:56,703 INFO OutputFile:109 - Absolute file name/Users/huanhoangg/Documents/Code/vadere.v3.0.linux/Scenari
os/Demos/Test/vadere/output/test_2024-11-03_20-23-29.148/postvis.traj
20:23:56,728 INFO OutputFile:109 - Absolute file name/Users/huanhoangg/Documents/Code/vadere.v3.0.linux/Scenari
os/Demos/Test/vadere/output/test_2024-11-03_20-23-29.148/overlaps.csv
20:23:56,729 INFO OutputFile:109 - Absolute file name/Users/huanhoangg/Documents/Code/vadere.v3.0.linux/Scenari
os/Demos/Test/vadere/output/test_2024-11-03_20-23-29.148/overlapCount.txt
20:23:56,730 INFO Simulation:251 - Finished writing all output files
20:23:56,730 INFO Simulation:254 - Post-loop: before waitForTraci
20:23:56,730 INFO Simulation:261 - Post-loop: finished.
20:23:56,730 INFO ScenarioRun:207 - Simulation run finished.
20:23:56,730 INFO ScenarioRun:265 - Simulation of scenario test finished.

```

Figure 4: Console output snippet during simulation of the scenario using command line

found under the directory `scenario/task_3/output_comparison/without_new_pedestrian`. Since the `diff` command produces no output, the two sets of files are identical.



```

(base) huanhoangg@Huan-MBP RiMEA-Scenario-6 % diff Console/overlapCount.txt GUI/overlapCount.txt
(base) huanhoangg@Huan-MBP RiMEA-Scenario-6 % diff Console/overlaps.csv GUI/overlaps.csv
(base) huanhoangg@Huan-MBP RiMEA-Scenario-6 % diff Console/postvis.traj GUI/postvis.traj
(base) huanhoangg@Huan-MBP RiMEA-Scenario-6 %

```

Figure 5: Comparison of output files generated from running the scenario using the Vadere GUI and the command line

Adding Pedestrians

For this task, Python is used as the programming language, and the implementation is located in the file `scenario/task_3/add_pedestrian.py`. This file contains several functions, with `add_pedestrian` being the primary function that adds a new pedestrian with specified attributes to the scenario. Assuming that the scenario JSON file is correctly formatted and contains all required attributes without errors, the logic of the implementation is as follows:

1. The scenario JSON file is read from the specified path and converted into a Python dictionary using the function `read_scenario`.
2. For the attributes of the pedestrian, the user can pass the pedestrian ID, target IDs, x and y coordinates on the topology, and the updated scenario name as parameters to the function `add_pedestrian`. For this newly added pedestrians, only the attributes crucial for the RiMEA scenario 6's simulation are considered. Attributes such as `isLikelyInjured`, `psychologyStatus`, and others that do not affect the RiMEA simulation are excluded. The value for the attribute `freeFlowSpeed` is randomly generated through the `speed_inference` function based on a Gaussian distribution with the mean `speedDistributionMean` and standard deviation `speedDistributionStandardDeviation` taken from the scenario file.
3. If the pedestrian ID is invalid (already in use by other pedestrians, obstacles, or targets, as checked by `is_pedestrian_id_valid`), or if the target IDs are invalid (targets with specified IDs do not exist, as checked by `are_target_ids_valid`), or if the pedestrian's coordinates are inside obstacles (checked

by `is_coordinate_inside_obstacles`, implemented using the Ray-Casting algorithm), an exception is raised, prompting the user to choose different values for the parameters.

4. If the provided parameters are valid, the pedestrian is added to the scenario dictionary, namely to the `dynamicElements` list in the topography section of the scenario dictionary.
5. Finally, the updated dictionary is saved as a new scenario JSON file at the user-specified path using the `save_scenario` function.

When executing the `add_pedestrian.py` file using the command line, all the parameters are specified in the following order: `[input_file_path] [pedestrian_ID] [target_IDs] [x_coordinate] [y_coordinate] [scenario_name] [output_file_path]`. In our case, this would be:

```
python add_pedestrian.py "../task.1/corner_scenario6.scenario" 100 "3" 9.0 9.0 "task3"
"./corner_scenario6_updated.scenario"
```

This means the newly added pedestrian will be placed at coordinates (9.0, 9.0) (see Figure 6), with an ID of 100 and a target ID of 3. The updated scenario is saved under the name `corner_scenario6_updated.scenario`. There is also a Jupyter Notebook file called `Add_Pedestrian.ipynb` for those who are not familiar with specifying parameters via the command line and prefer an easier way to change the parameter values.

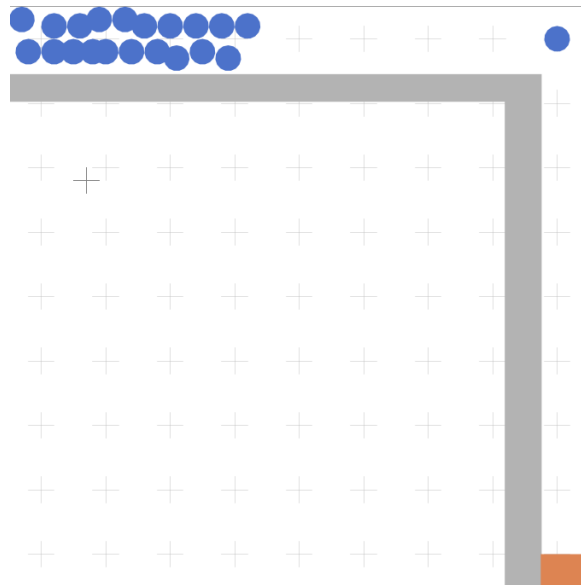


Figure 6: Updated scenario with the newly added pedestrian positioned at the top right corner

To compare the time it takes for pedestrians to reach the target, in contrast to Task 1, the target is set to absorbing. Since the newly added pedestrian is positioned closer to the target than the others, it understandably reaches and is absorbed by the target faster, specifically after 5.50 seconds. Among the original 20 pedestrians, the fastest takes 7.22 seconds, while the slowest takes 33.02 seconds.

As expected, executing the updated scenario, whether using the Vadere GUI or the command line, results in no differences in the output files. The `diff` command is again used to compare the outputs. The outputs can be found under the directory `scenario/task.3/output_comparison/with_new_pedestrian`.

Report on task 4: Integrating a New Model

SIRGroupModel and Its Related Classes

The class `SIRGroupModel` is responsible for the management of the SIRGroups in the SIR model. It is located in the package `org.vadere.simulator.models.groups.sir`. During each update step, it determines which `SIRGroup` a pedestrian should be assigned to or whether they should remain in the same group, based on its update method. There are three groups (`SIRGroups`) in the SIR Model, which are the susceptible group, the infected group, and the removed group (no usage yet, added later in subsequent tasks). They are determined by their group IDs, which are set to one of the constants in enum `SIRType`. Here are some of the key attributes of `SIRGroupModel` listed:

- `LinkedHashMap<Integer, SIRGroup> groupsById`: The list of mappings from group ids (keys) to SIRGroups (values).
- `AttributesSIRG attributesSIRG`: Provides the essential parameters for SIR model and infection propagation.
- `Topography topography`: Contains the essential attributes and methods for the management of the topography of the scenario.
- `int totalInfected = 0`: Total number of infected pedestrians.

One of the most important methods of `SIRGroupModel` is the `update` method. It is also the method we mainly focus on in this task and implement new features by changing it. This method is called in every update step of the simulation and it switches the states of pedestrians to infected or removed (no switch to removed state yet, added later) according to its own state, parameters in `AttributesSIRG` and the state of the pedestrian neighborhood around it in a certain distance.

It is also worth noting that the `initializeGroupsOfInitialPedestrians` method of `SIRGroupModel`, which is called in the `preLoop` method of `SIRGroupModel` to initialize the initial pedestrian groups, was not implemented in the original code. In our implementation, it first iterates over all pedestrians and assigns the number of `infectionsAtStart` pedestrians to the infected `SIRGroup` when the value of the `totalInfected` attribute is less than the value of `infectionsAtStart` attribute. At each assignment to infected `SIRGroup`, it increments the `totalInfected` by 1. If the `totalInfected` is equal to `infectionsAtStart`, then it assigns the pedestrians to susceptible `SIRGroup` in the loop.

There are two more files that are important for SIR model within the same package of `SIRGroupModel`, which are enum `SIRType` and class `SIRGroup`.

SIRType

The enum `SIRType` holds the following enumeration constants, which represent pedestrian states. They are represented by their ordinals:

- `ID_INFECTED`: Infected state. Represented by 0.
- `ID_SUSCEPTIBLE`: Susceptible state. Represented by 1.
- `ID_REMOVED`: Removed/recovered state. Represented by 2. (used later in subsequent tasks)

SIRGroup

The class `SIRGroup` represents one single group in the SIR model, which is an infected, susceptible, or removed group. The type of `SIRGroup` is determined by its group ID. It is set to one of the constants in `SIRType` representing these states. A `SIRGroup` provides the basic attributes (listed below) and the methods for basic operations like adding members (pedestrians) to the group and removing members from the group. Here are the key attributes of `SIRGroup` listed:

- `int id`: Group ID representing whether it is susceptible, infected or removed group. (0, 1 or 2).
- `int size`: Maximum number of pedestrians that can be contained in the group.
- `ArrayList<Pedestrian> members`: List of pedestrians that belong to this group.

AttributesSIRG

The class `AttributesSIRG` contains crucial parameters for the SIR model as attributes. It is located in the package `org.vadere.state.attributes.models`. The attributes of it are used in the `update` method of `SIRGroupModel` and are important for deciding how the infection will be propagated among the pedestrians. Here are the attributes of it:

- `int infectionsAtStart`: Default value is 0. This indicates how many pedestrians are infected at the beginning.
- `double infectionRate`: Default value is 0.01. It is the probability of the pedestrian getting infected by one of their neighbors after one simulation step (changed to "after 1 second" with modifications in the `update` method of `SIRGroupModel`).
- `double infectionMaxDistance`: Default value is 1. It is the maximum distance within which an infected pedestrian can infect another pedestrian.
- `double recoveryRate`: Default value is 0.01 (Not used yet, used later in subsequent tasks). It represents the probability of a pedestrian recovering after one simulation step (which was modified to 'after 1 second' in the `update` method of `SIRGroupModel`).

Output Processor

In order to analyze the change in the number of infected and susceptible pedestrians, the output data processor `FootStepGroupIDProcessor`, which is located in the package `simulator.projects.dataprocessing.processor`, is used. A new output file `SIRinformation.csv` is linked to it. `FootStepGroupIDProcessor` provides access to the internal data map for saving concrete data of type `Integer`, which is the group ID, with key type `EventtimePedestrianIdKey`. This key consists of the pedestrian ID and the event time, which is the exact time at which an event (infection in our case) occurred. This data map is written to the output file `SIRinformation.csv`. It has three columns: `pedestrianId`, `simTime`, and `groupId`. The first two columns, `pedestrianId` and `simTime`, represent the `EventtimePedestrianIdKey`, and the third column, `groupId`, represents the SIR group (state) the pedestrian is assigned to, e.g., susceptible or infected. Later, this `SIRinformation.csv` file will be used for visualizing the simulation results (plots of changing number of infected and susceptible pedestrians over time), using the Dash/Plotly app in the folder "visualization" of the repository.

Efficiency Improvement with LinkedCellsGrid

The class `LinkedCellsGrid<T extends PointPositioned>` (T is Pedestrian in our case) uses a grid structure to represent the surface and pedestrian positions, i.e. a 2D-array of type `GridCell<Pedestrian>`, where one can store multiple pedestrians in one cell. So, `LinkedCellsGrid` does not use numbers of type `double` for the coordinates of the positions of pedestrians. Instead, it calculates the discrete integer grid coordinates (x,y), i.e. the position in the grid, from 0 to (`gridSize - 1`) in both coordinates for the points with coordinate values of type `double`. It is located in the package `org.vadere.util.geometry`.

In the original update function of `SIRGroupModel`, the procedure for finding the neighbors within the maximum infection distance involves checking the distance between the current pedestrian and every other pedestrian. If the map is big and there are many pedestrians in the scenario, this method can be very inefficient. By using a grid-like structure to register the position of each pedestrian in the grid, we can efficiently get the grid cells that can lie within the maximum infection distance. Then it's sufficient to simply check if the pedestrians within those cells are within the maximum infection distance.

In the code, we first create an instance of the class `LinkedCellsGrid`, defining its size and side length. The choice of side length is arbitrary (in our case, 10), but generally, it should neither be too large nor too small. A very dense grid will result in a pedestrian having many valid neighboring grid cells, which can slow down the algorithm, as we need to iterate over these cells to get the list of valid neighbors. On the other hand, a very sparse grid won't filter out enough invalid neighbors, which also reduces the algorithm's efficiency. After that, we need to register the positions of the pedestrians in the grid. Since the positions of the pedestrians change over time, it's necessary to register the positions of the pedestrians in the grid at the beginning of each update

step and later remove them at the end of the update step to keep the grid always updated.

These operations are done using the `addObject` function and the `removeObject` function in `LinkedCellsGrid`. To obtain the valid neighbors using the grid structure, we utilize the `getObjects` function, which returns a list of valid neighboring pedestrians. This approach allows us to iterate over only the list of valid neighbors, rather than all pedestrians, significantly improving the efficiency of the procedure.

Static Scenario with 1000 Pedestrians (Without Recovered State)

In Figure 7, a static scenario with 1000 randomly distributed static pedestrians is shown. It is initialized with 990 susceptible and 10 infective pedestrians. This scenario is configured with the following key parameters:

Setup of the static scenario:

- Source and target overlapped with dimensions $50\text{m} \times 50\text{m}$
- Infections at the start: 10
- Infection rate: First 0.04, then 0.08
- Infection maximum distance: 2.0 m
- Random position spawning (`eventPositionRandom:true`)
- Disabled free space requirement (`eventPositionFreeSpace:false`)
- (`leavingSpeed:-1.0` in target): To make the output processor continue writing the group IDs even though none of the pedestrians are moving
- Main model for locomotion: `OptimalStepsModel`
- Sub-models: `SIRGroupModel`
- Target zone set to be a “waiter” to keep registering “infection events” with update frequency: 1.0

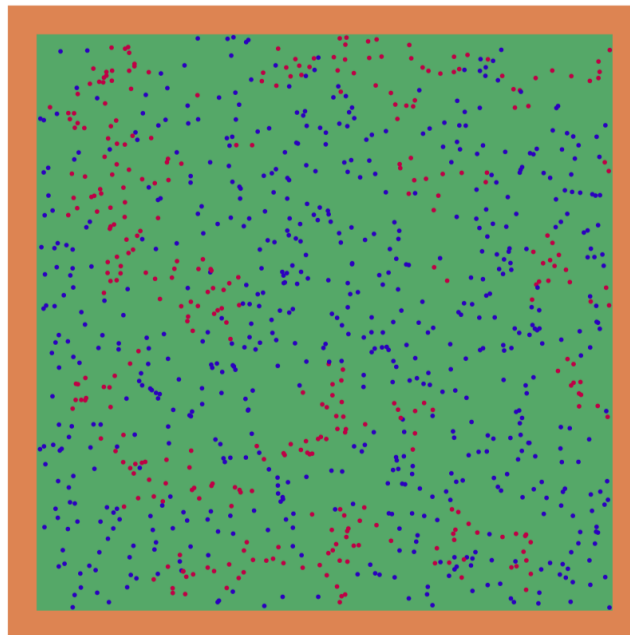


Figure 7: Static scenario with 1000 pedestrians, randomly distributed in the scenario. Some pedestrians are in infected state (red), and the rest are in susceptible state (blue). No recovered state.

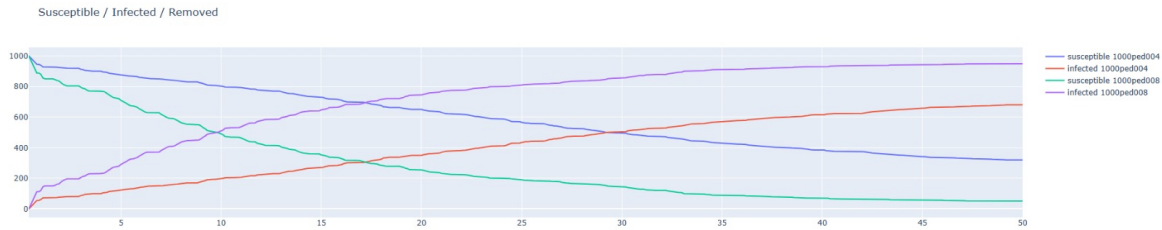


Figure 8: Plots of the development of number of susceptible and infected pedestrians over time with infection rates of 0.04 (blue as susceptible, red as infected) and 0.08 (green as susceptible, purple as infected)

The plots of static scenario is shown in Figure 8. Vertical axis represents number of pedestrians and horizontal axis represents time in seconds. From this plot, we can observe that in the scenario with an infection rate of 0.04 (represented by the red and blue lines), it takes approximately 30 seconds for half of the population to become infected. After the infection rate is increased to 0.08 (represented by the green and purple lines), it takes approximately 10 seconds for the same to happen. The points at which this occurs are marked at the intersection of the susceptible and infected pedestrian lines for both cases.

The scenario files of static scenario with 1000 pedestrians can be found in the folder `scenario.task_4.scenarios` under the names `1000ped_infection_rate_0_04` and `1000ped_infection_rate_0_08`, representing the cases with infection rate 0.04 and 0.08, respectively. The only difference between them is the infection rate.

Corridor Scenario

In Figure 9, a $40\text{m} \times 20\text{m}$ corridor scenario is shown. One source (green) and one target (orange) are placed on both left and right edges. In total, there are 2 sources and 2 targets. The source on the left side generates 100 pedestrians that move from left to right, with the target on the right side. Conversely, the source on the right side generates pedestrians that move from right to left toward the target on the left. The key parameters of the corridor scenario are as follows:

Setup of corridor scenario

- $40\text{m} \times 20\text{m}$ corridor, 2 sources and 2 targets (one on the left and one on the right, respectively). All sources and targets have the same size of $1\text{m} \times 20\text{m}$.
- Infections at the start: 10
- infection rate: 0.1
- Infection maximum distance: 2.0
- Main model for locomotion: `OptimalStepsModel`
- Sub-models: `SIRGroupModel`
- Enabled free space requirement (`eventPositionFreeSpace:true`)
- Random position spawning (`eventPositionRandom:true`)
- potential personal space width of pedestrian: 1.0

The plot of the corridor scenario can be seen in Figure 10. Vertical axis represents number of pedestrians and horizontal axis represents time in seconds. On the right edge of the plot there is a red box with a pair of numbers. Left part of the pair represents simulation time in seconds and the right part represents the number of infected pedestrians. It shows that there are 93 infected pedestrians at the end of the simulation. There are 10 infected pedestrians initially, due to the `infectionsAtStart` parameter being set to 10. This means $93 - 10 = 83$ pedestrians got infected in counter-flow.

The scenario file of the corridor scenario can be found in the folder with the path `scenario.task_4.scenarios`. It is named `corridor.scenario`.

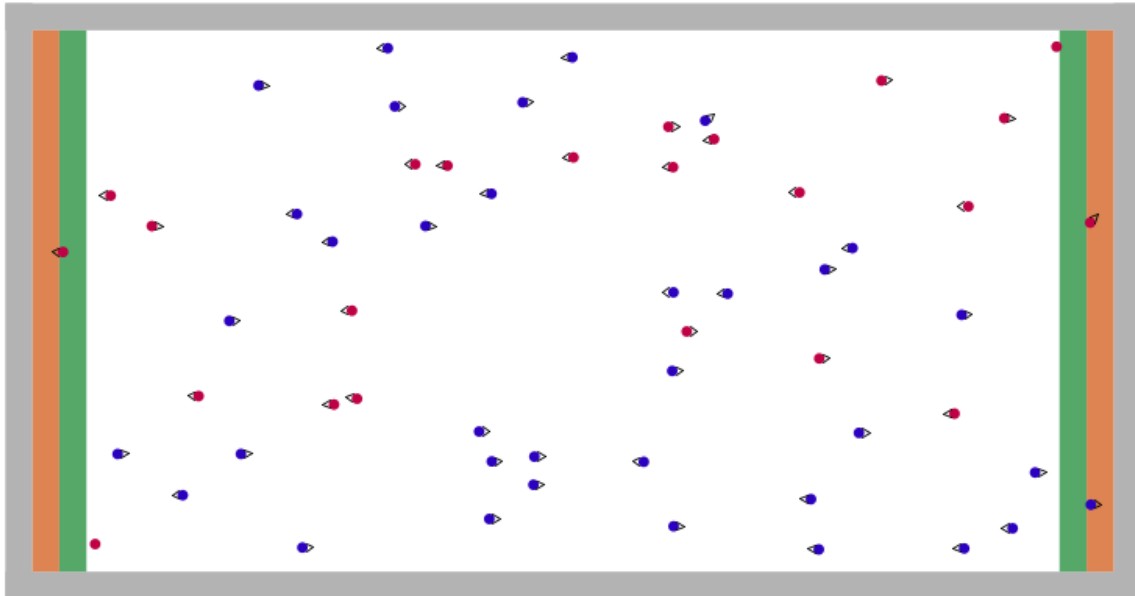


Figure 9: Screenshot of the simulation of the corridor scenario, with red representing infected pedestrians and blue representing susceptible pedestrians

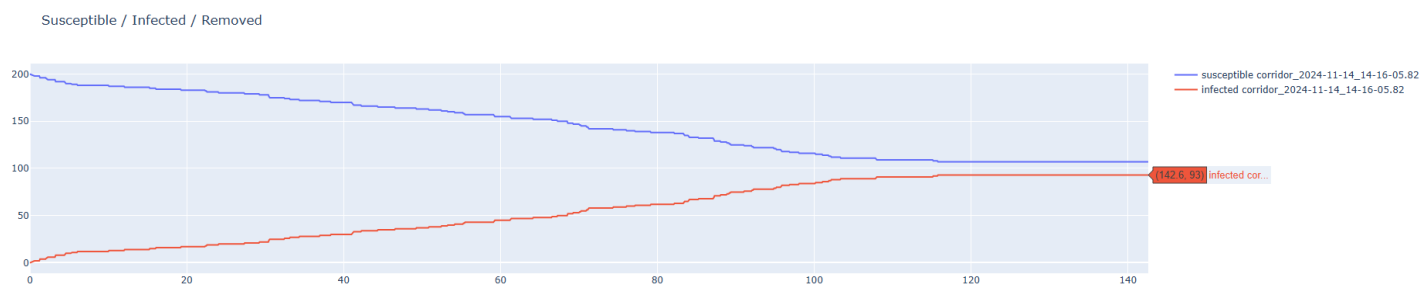


Figure 10: Plot showing the development of the number of susceptible and infected pedestrians over time in the corridor scenario, with blue representing susceptible pedestrians and red representing infected pedestrians

Decoupling the Infection Rate and the Time Steps

The following implementation of decoupling adjustment in the update method of `SIRGroupModel` not only solves the decoupling problem of infection rate but also decouples the recovery rate and time steps. Therefore, this modification and explanation of the method also applies to the recovery rate, providing us with both consistent infection and recovery rates at the same time. As a result, Task 5.2 is also addressed here.

In the original code, the infection rate depends on the step size of the simulation. For example, with each time step having a fixed length of 0.2s, the infection will propagate 5 times within one second. However, we expect the infection rate to be interpreted as “the probability of a pedestrian getting infected by one of their neighbors after one second”. To address this issue, we introduce two variables `last_simTime_int` and `current_simTime_int`, which record the integer part of the simulation time of the last time step and the current time step, respectively. Only if `current_simTime_int` is bigger than `last_simTime_int` do we execute the propagation of the infection (and later the process of recovery) because it means that we have passed the current second and arrived at the beginning of next second. Otherwise, if both variables are equal, then it means we are still in the same second, so we don’t do anything and just exit the current update step. By doing so, we ensure that the body part of the update function is only executed once per second. One thing to note is that we initialize `last_simTime_int` with the value -1 so that the update function will be executed only once during the first second (from time point 0 to time point 1).

Possible Extensions

Extension 1) Weather conditions: Weather conditions like rain, snow or sun could be implemented in the model. It would make the model more realistic, as we adjust our walking speed or path decisions according to the weather conditions in real life. The same logic can be applied to the pedestrians in the model. For example, if there are two pathways to reach a target, one being an outdoor pathway and the other an indoor one, pedestrians should prefer the indoor pathway to avoid getting wet. Or, if it is snowy, they should reduce their walking speed because the road might be icy and they could fall.

Extension 2) Public transportation: Public transport vehicles are among the crowded places where infectious diseases can spread the most in real life. Therefore, simulating this scenario would be a realistic choice, giving informative insights into infection propagation. For instance, pedestrians can be made to use a bus at a certain point along their path, and the bus (e.g., a large box with a distinct color) can move to a specific station at designated times or after a particular number of pedestrians have boarded.

Extension 3) Infection sensing: In real life, if we see someone coughing or sneezing excessively, we tend to become more cautious and keep a greater social distance from them compared to other people. Therefore, it would make the SIR model more realistic if we applied the same principle. For example, each pedestrian could have a sensing distance parameter to check for excessive sneezing or coughing in nearby pedestrians within that distance. If detected, the pedestrian could then increase their social distance.

Report on task 5: Analysis and Visualization of Results

Task 5.1: Implementation of Recovered State

In our implementation of the SIR model, we utilize the existing `ID_REMOVED` state to represent recovered individuals. While the code uses the term “removed” (following traditional SIR model terminology), these pedestrians don’t actually leave the simulation - they continue to move and interact with others, just like real-world recovered individuals who return to their daily activities while being immune to reinfection.

We ensured that recovered pedestrians behave realistically by implementing two crucial aspects: they cannot be infected again, and they cannot spread the infection to others. This is achieved through straightforward yet effective code logic:

When a pedestrian recovers from infection, we transition them to the recovered state using the following mechanism:

```

if (g.getID() == SIRType.ID_INFECTED.ordinal()) {
    if (this.random.nextDouble() < attributesSIRG.getRecoveryRate()) {
        elementRemoved(p);
        assignToGroup(p, SIRType.ID_REMOVED.ordinal());
        this.totalInfected -= 1;
    }
}

```

To implement immunity, we simply skip infection checks for recovered individuals:

```

if (g.getID() == SIRType.ID_REMOVED.ordinal()) {
    continue; // Skip infection checks for recovered pedestrians
}

```

In the simulation, recovered pedestrians remain active participants in the crowd dynamics. They maintain their ability to move naturally and avoid collisions with others, while being visually distinguishable from susceptible and infected individuals. This allows us to track the progression of immunity throughout the population.

Task 5.2: Making Recovery Time-Independent

The solution to this task is explained in the section Decoupling the Infection Rate and the Time Steps of Task 4. The solution provided in that section solves the decoupling problem for both infection and recovery rates.

In addition to the explanation in the mentioned section of Task 4, we can also note that the probability of recovery is now completely independent of external factors - it doesn't matter how many people are nearby, how long someone has been infected, or where they are in the simulation. This aligns with real-world scenarios where recovery often follows predictable patterns, independent of environmental conditions.

Recovered individuals maintain their normal movement patterns while being immune to reinfection, creating a realistic simulation of post-infection behavior in crowds. This approach allows us to accurately model how immunity spreads through a population over time, providing valuable insights into disease progression in crowded environments.

Task 5.3: Visualization Implementation

For visualizing the SIR model dynamics, we utilized and modified the pre-implemented Dash/Plotly application. The original visualization only tracked and displayed susceptible and infected populations, so we enhanced it to include the recovered state, providing a complete view of the epidemic progression.

Implementation Changes

The main modifications were made to the data processing and visualization logic in `utils.py`. We expanded the population tracking to include transitions from infected to recovered states:

```

# Added recovered state tracking in group_counts DataFrame
group_counts = pd.DataFrame(
    columns=['simTime', 'group-s', 'group-i', 'group-r'])

```

We implemented tracking of state transitions by monitoring each pedestrian's state changes over time. When an infected individual recovers, we now correctly update both the infected and recovered population counts:

```

elif g == ID_REMOVED and current_state == ID_INFECTED:
    # Handle transition from infected to recovered
    current_state = g
    group_counts.loc[group_counts['simTime'] > st, 'group-i'] -= 1
    group_counts.loc[group_counts['simTime'] > st, 'group-r'] += 1

```

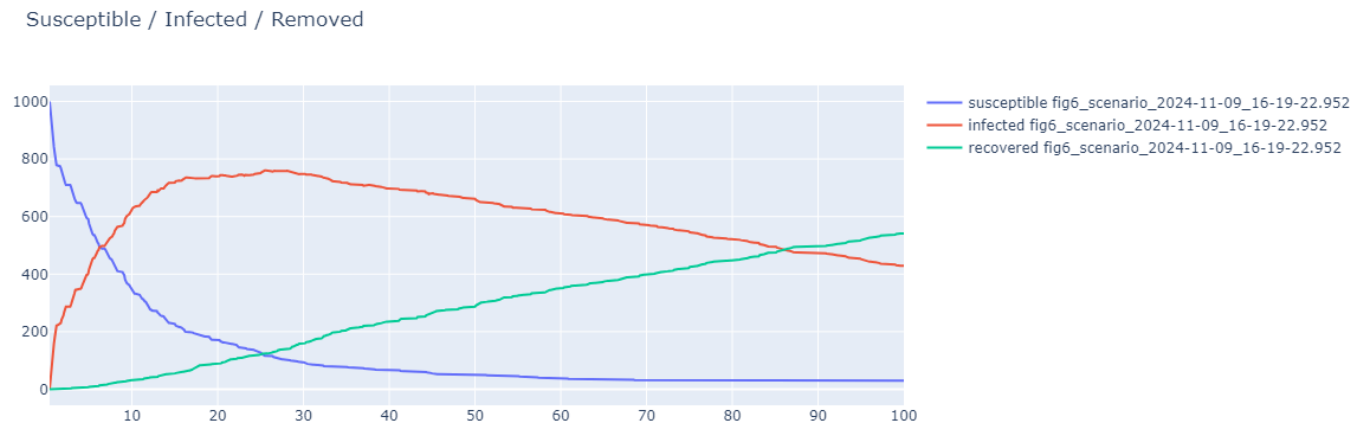


Figure 11: Plot showing the development of the SIR values over time. The visualization was updated to show the recovered pedestrians. This plot was for the 1000 pedestrian scenario.

Visualization Features

The enhanced visualization now displays three distinct line plots as seen in Figure 11. The **Blue** line represents the susceptible population, while the **Red** line represents the infected population and the **Green** line represents the recovered population

Tests

Test 1: Static 1000 Pedestrian Scenario

We implemented a large-scale static scenario to study infection spread in a dense, non-moving crowd. The scenario was configured with 1000 pedestrians, initially distributed as 10 infected and 990 susceptible individuals.

Scenario Setup The simulation was configured using the following key parameters:

- Source and target overlapped with dimensions $50\text{m} \times 50\text{m}$
- Random position spawning (`eventPositionRandom:true`)
- Disabled free space requirement (`eventPositionFreeSpace:false`)
- Static pedestrians (`leavingSpeed:-1.0` in target)
- Initial infection rate: 0.1
- Recovery rate: 0.01
- Infection maximum distance: 2.0m

Results Analysis Figure 12 shows a snapshot of the simulation

Visualization Features

The SIR dynamics over time are visualized in Figure 11. The results show the characteristic SIR model behavior:

- Rapid initial spread: The susceptible population (blue line) decreases sharply in the first 10 seconds
- Peak infection: The infected population (red line) reaches its maximum of approximately 750 individuals around $t=20\text{s}$

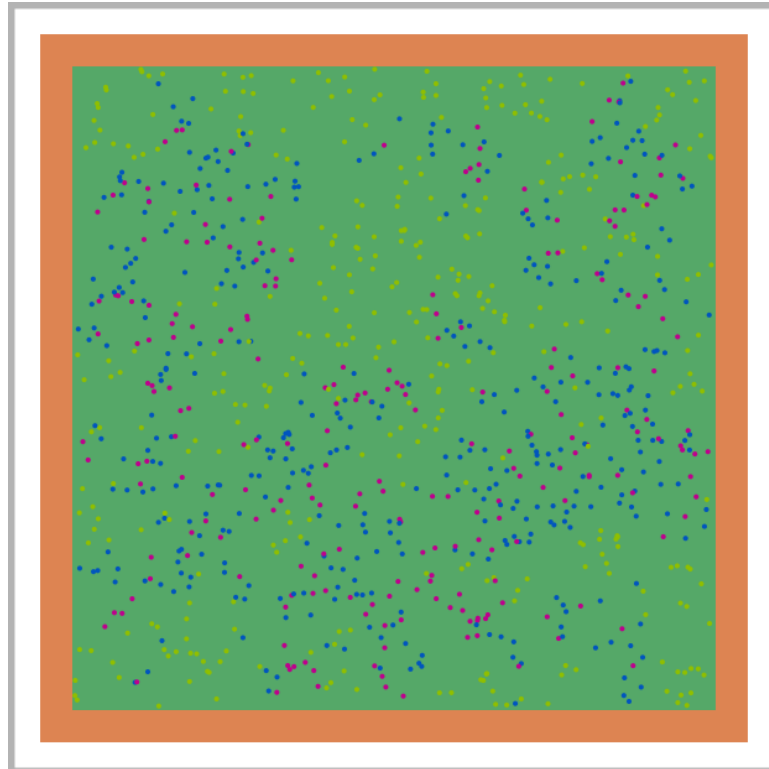


Figure 12: Static scenario with 1000 pedestrians, randomly distributed in the scenario. Some pedestrians are in infected state (blue), some are susceptible (red) and the rest are recovered (yellow).

- Recovery phase: The recovered population (green line) grows steadily, becoming the dominant group after $t=80s$

By $t=100s$, the population distribution stabilizes since people are either immune and have recovered or susceptible but the infection might not be able to reach them since the distance is too large or no one in their distance is infected. This test demonstrates the successful implementation of both the infection and recovery mechanisms in our enhanced SIR model, showing realistic epidemic progression in a static crowd scenario.

Test 2: Altering infection and recovery rate

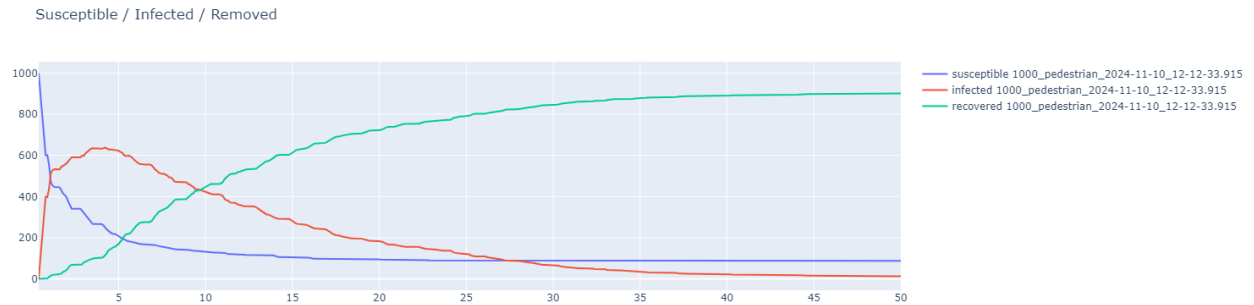
We tested the 1000 pedestrian scenario with different infection and recovery rates to study the changes in the SIR models as seen in Figure 13.

First, in Figure 13a, we wanted to study what would happen if the infection rate (0.2) is fairly larger than the recovery rate (0.1). What happened in the SIR model is that the infection spread rapidly, as seen in the red line at the start of the figure, and the susceptible population dropped. The recovered population then took some time to climb but did so slowly. This could be dangerous in real-life scenarios. If people do not implement precautionary measures and the infection spreads faster than individuals recover, it could lead to problems.

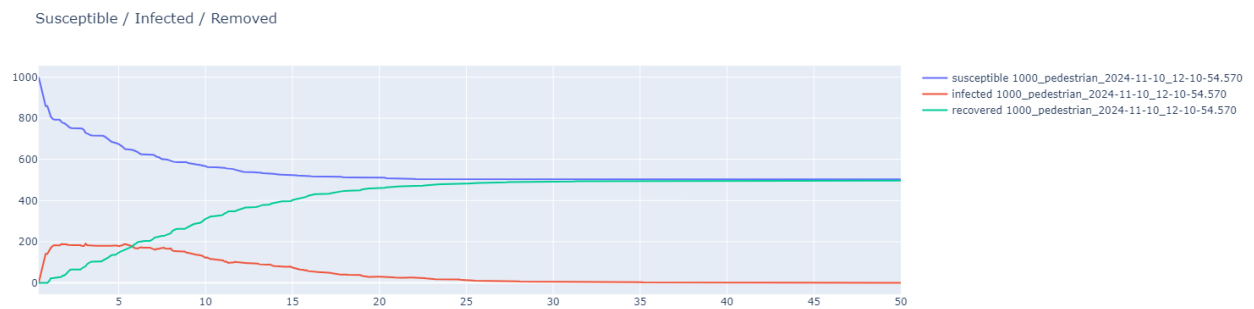
We also tested out the reversed case when the recovery rate (0.2) is higher than the infection rate (0.1). As seen in Figure 13b, this scenario is the complete opposite of the first case. The infections initially rose, as seen in the red line, but because the recovery rate was so high and people quickly became immune, there was not enough time for the infection to spread to other pedestrians.

The last case involved both rates being equal but higher than usual, at 0.2. As shown in Figure 13c, in this scenario, infections began to rise but faced more difficulty compared to the first case, where the infection rate increased rapidly. The green line, representing the recovery of pedestrians, gradually climbs as more individuals become immune.

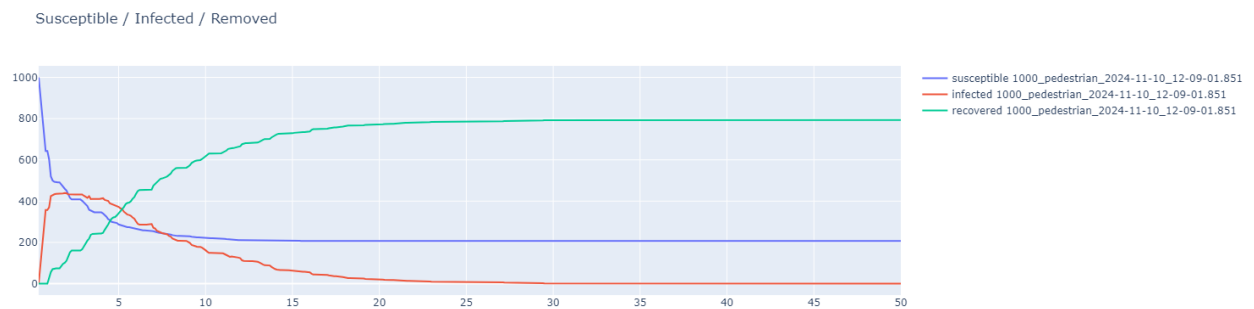
An interesting takeaway is that nearly all pedestrians in the end got infected in Figure 13a, whereas in Figure 13b, half of them never became infected, and in Figure 13c, only about 20% remained uninfected.



(a) Infection Rate 0.2, Recovery Rate 0.1



(b) Infection Rate 0.1, Recovery Rate 0.2



(c) Infection Rate 0.2 Recovery Rate 0.2

Figure 13: Three SIR plots with different infection and recovery rates in the 1000 pedestrian scenario. Blue line is susceptible, Red is for infected and Green for recovered

Test 3: Supermarket Scenario

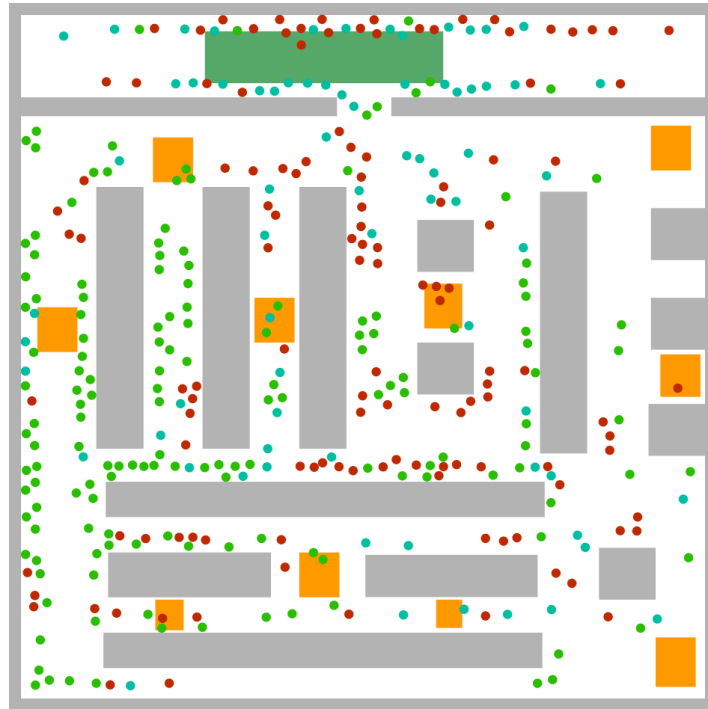


Figure 14: Layout of a supermarket, with several targets and target changers to specify the routing of pedestrians. They enter at the top, from the source, walk around in the supermarket and then leave through the same entrance again.

We designed a layout similar to a supermarket in Figure 14. It has one source generating pedestrians walking in with 10 targets that the pedestrians can go to. Each target has a target changer above it with a probability of 0.7 to change to another target. We ran the scenario with an infection rate and recovery rate of 0.01. We also set the **pedPotentialPersonalSpaceWidth** to 5.2 and infection maximum distance to 2. The results can be seen in Figure 15. Although we only started with 10 infected pedestrians with a low probability of infection, we can see that for the first third of the SIR graph (from 0 to roughly 80 seconds), everything was going okay. However, as soon as the supermarket started to get crowded and more pedestrians started to bump into each other, a sharp increase in infections can be seen in the middle section of the SIR graph.

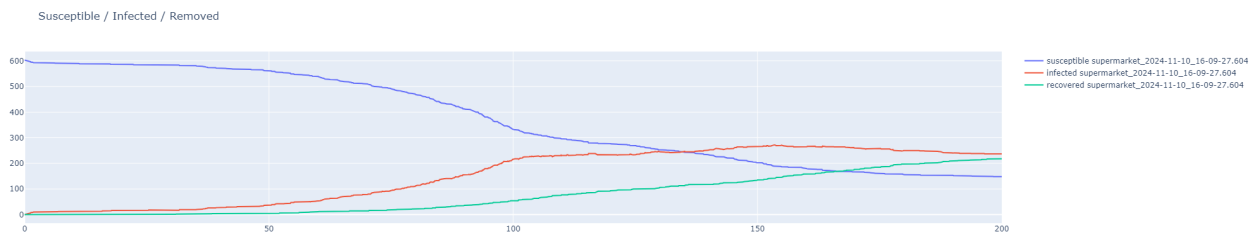


Figure 15: SIR of supermarket scenario with initial parameters. Blue line is susceptible, Red is for infected and Green for recovered

We wanted to know what would happen if we increase the **pedPotentialPersonalSpaceWidth** to 10.2 and keep the rest of the values as they are. The resulting SIR in Figure 16 shows no improvements in the number or speed of infections. It is still similar to Figure 15. This is likely because many people are getting into the supermarket. To address this, we kept all parameters the same but reduced the update frequency of the spanner. This adjustment should allow more room for people to enter and leave before additional individuals enter the

supermarket. The results were promising. In Figure 17, we can notice that the start of infections got pushed further down the timeline and more people entered and left without getting infected. This shows that while social distancing may be important, it cannot be effectively achieved unless we limit the number of pedestrians in the supermarket. In the first case, we had 600 people, but in the second, we only had 400. Moreover, the 400 pedestrians entered the supermarket more slowly, allowing those already inside to finish their shopping and leave, which resulted in fewer infections.

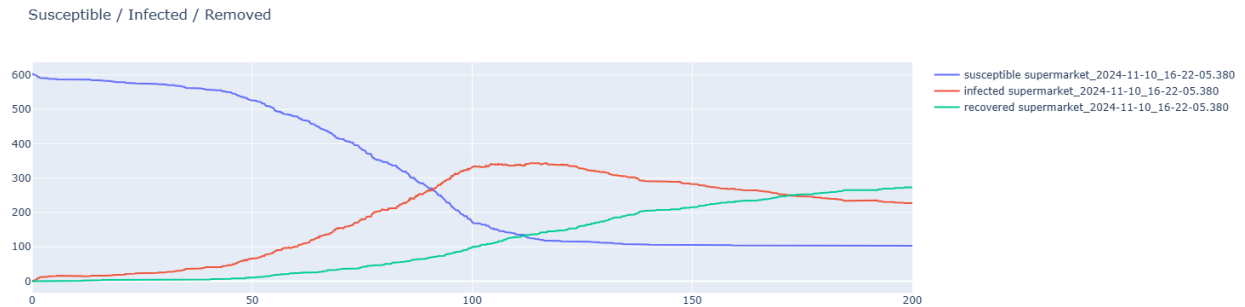


Figure 16: SIR of supermarket scenario after increasing **pedPotentialPersonalSpaceWidth**

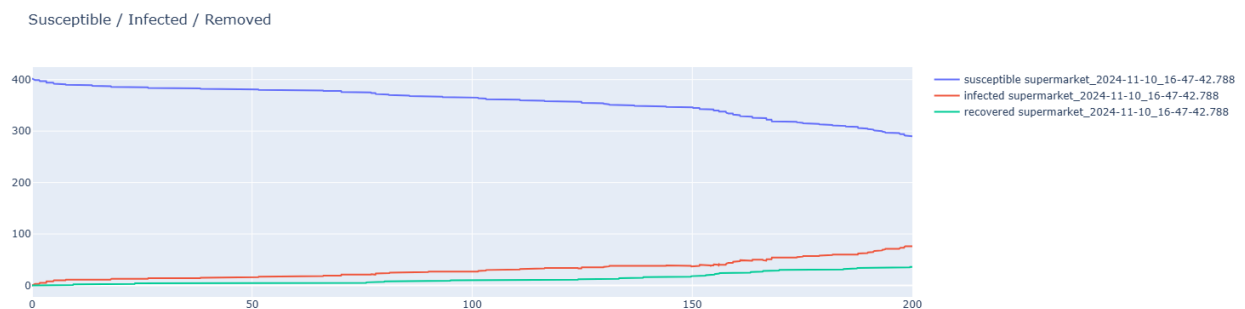


Figure 17: SIR of supermarket scenario after decreasing spawn frequency.

References

- [1] Felix Dietrich and Gerta Köster. Gradient navigation model for pedestrian dynamics. *Physical Review E*, 89(6):062801, 2014.
- [2] Dirk Helbing and Peter Molnar. Social force model for pedestrian dynamics. *Physical review E*, 51(5):4282, 1995.
- [3] Michael J Seitz and Gerta Köster. Natural discretization of pedestrian movement in continuous space. *Physical Review E—Statistical, Nonlinear, and Soft Matter Physics*, 86(4):046108, 2012.