Report for exercise 1 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: 2024–10–31

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 Modeling Environment

Task 1 involves initializing the simulation environment, placing various elements like pedestrians, targets, and obstacles onto the grid, and providing the functionality for visualizing the grid. The main goal is to create a grid-based environment where different entities like pedestrians and targets can interact. In this task, we implemented the core functions that set up and display the simulation environment. This task sets the stage for further tasks by creating the crowd simulation's foundation.

The core components of this task are the `__init__` and `get_grid()` methods in the class `Simulation`. While `__init__` deals with general initialization of the simulation, `get_grid()` handles the placement of elements on the grid specifically. `__init__` also calls the `get_grid()`.

- `Simulation.__init__`: Initializes the grid with the given width and height values and sets each cell as an empty cell. It also acquires essential constituents of the simulation such as a list of pedestrians, positions of targets and obstacles, choice of algorithm for distance computation, and an indicator if targets should absorb the pedestrians or not from the simulation configuration class `SimulationConfig`. It also calls the `Simulation.get_grid()` method to place the elements (pedestrians, obstacles, and targets) according to the specified coordinates in `SimulationConfig`.
- `Simulation.get_grid()`: Places pedestrians, obstacles, and targets based on the given configuration. It does so by iterating over the list of pedestrians, obstacle positions, and target positions, which are already acquired and saved in the attributes of the `Simulation` class, and assigning them to their corresponding cells on the grid. In the end, the full state of the grid is returned.

Initialization of the Simulation Class

The `Simulation` class is initialized through its `__init__` method. This method sets up the grid based on the specified width and height, and it also acquires the list of pedestrians, targets, and obstacles from the simulation configuration class `SimulationConfig` and initializes them through calling `get_grid()` method. The configuration file provides the necessary parameters for the grid size and the elements' initial positions. `__init__` method initializes a grid of the specified size and fills it with the `empty` state (represented by the enum `el.ScenarioElement.empty`). After initializing the grid, the method calls `get_grid()` to populate the grid with pedestrians, targets, and obstacles.

Placing Elements on the Grid

Pedestrians, targets, and obstacles are placed on the grid using the `get_grid()` method. Each element is positioned according to its coordinates in the configuration file. This method helps render the first run of the simulation state in the GUI. Figure 1 shows an example of the grid visualization with a pedestrian and a target on a 5x5 grid.

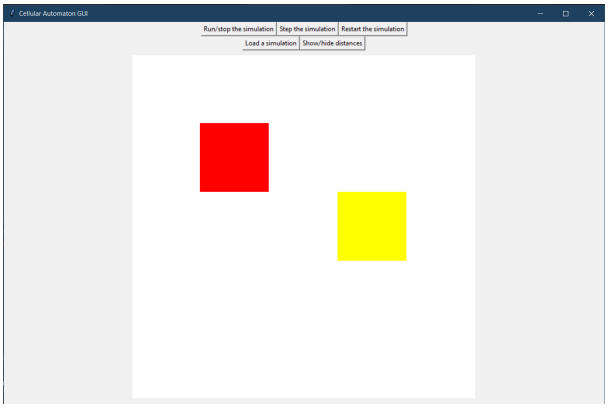


Figure 1: Grid visualization showing 1 pedestrian and 1 target in a 5x5 grid.

Report on task 2: First Step of a Single Pedestrian

Task 2 involves simulating the movement of a pedestrian toward a target using a naive distance computation. The pedestrian is initialized on a 50x50 cell grid, and the target is placed 20 cells away at the coordinates (25,25). The pedestrian moves one step closer in each simulation cycle towards the target, demonstrating a simple movement model without obstacles. This task is a stepping stone for more complex simulations, where pedestrians must navigate around obstacles or interact with one another.

The key methods implemented in this task are:

- `Simulation.get_neighbors()`: Returns the valid neighboring cells for a pedestrian's current position.
- `Simulation._compute_naive_distance()`: Returns the grid distance between pedestrians and the target.
- `Simulation.update()`: Updates the pedestrians' positions by selecting the neighboring cell that has the shortest distance to the target.

Code Explanation:

The `get_neighbors()` method is responsible for identifying the available moves for a pedestrian. It checks all adjacent cells and ensures the next move stays within the grid boundaries. It also shuffles the neighbor's order if the shuffle variable is `True`. The coordinates of the cells adjacent to the pedestrian's current position at (x,y) are defined as follows:

- `Left`: (x - 1, y),
- `Right`: (x + 1, y),
- `Up`: (x, y - 1),
- `Down`: (x, y + 1),
- `Up-Left (Diagonal)`: (x - 1, y - 1)
- `Up-Right (Diagonal)`: (x + 1, y - 1)
- `Down-Left (Diagonal)`: (x - 1, y + 1)
- `Down-Right (Diagonal)`: (x + 1, y + 1)

The `_compute_naive_distance_grid` method calculates the distance from each cell in the grid to the nearest target without considering obstacles. Given a tuple of target positions, the method creates a matrix of target coordinates and a grid of all cell positions. Using the SciPy[1] library, it then computes pairwise distances between each target and each cell in a single, efficient step. Each cell's minimum distance to any target is recorded, generating a complete distance grid representing the shortest path from each cell to the closest target. Cells occupied by obstacles are assigned an infinite distance, marking them as inaccessible to pedestrians.

The `update()` method performs one step of the simulation, updating the positions of all pedestrians based on their current distance to the target(s). For the current task, the `update()` method calculates the distance using the `_compute_naive_distance()` method to see which direction the target is and then checks the neighboring cells using the `get_neighbors()` method. Each pedestrian moves to the respective cell, which brings them closer to the target. In future tasks, the `update()` method will incorporate more features like pedestrian avoidance, such as pedestrian avoidance, obstacle avoidance, and target absorption. However, the implementation for Task 2 remains simple, as described above.

[1] `https://scipy.org`

25x25 Grid Scenario

A simulation was defined in `task2_scenario.json` with a grid of 50x50 cells (totaling 2500), a single pedestrian positioned at (5, 25), and a target located 20 cells away at (25, 25), as shown in Figure 2. The scenario was then simulated using our cellular automaton for 25-time steps, allowing the pedestrian to move toward the target and wait there at the end of the simulation. As seen in Figure 3, the pedestrian has successfully reached the target. The naive distance calculation was used since there were no obstacles.

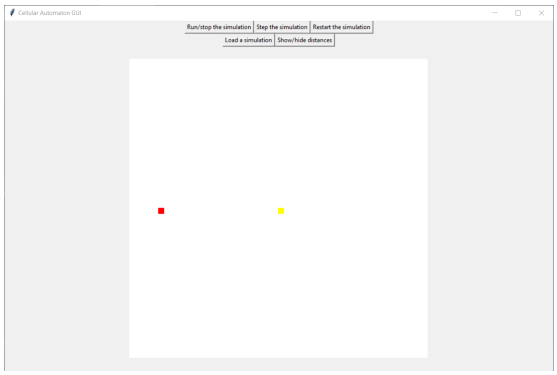


Figure 2: Cellular automaton on a 25x25 grid: Pedestrian at (5,25) and target at (25,25).

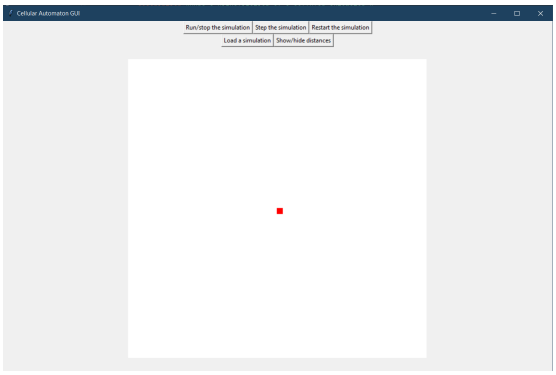


Figure 3: Cellular automaton after 25-step simulation: Pedestrian reaches the target (target is not absorbing in this scenario)

Report on task 3: Interaction of Pedestrians

In the third task, we mainly deal with three sub-tasks, which are speed adjustment, pedestrian avoidance, and making the target "absorbing" when it is specified in the configuration.

Scenario and Simulation

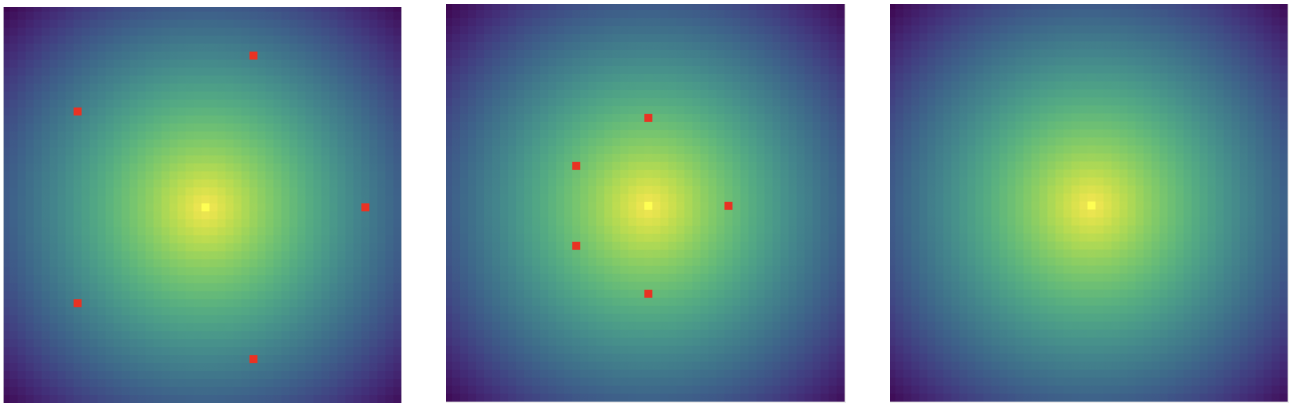
As demanded in task 3, we place five pedestrians equally spaced at angles of

$$\alpha = \frac{i}{5} \cdot 360^\circ,\ i = 0, \ldots, 4$$

in an Euclidean distance of 20 around a single absorbing target. Upon reaching the target, the pedestrians will be "absorbed" (removed from the scenario). We create a square scene, where length and width are both 50 units (cells) long (50x50), and use naive distance computation. The five pedestrians, all of which have a speed of 1, are placed at the following coordinates:

1. (45, 25)
2. (31, 44)
3. (9, 37)
4. (9, 13)
5. (31, 6)

Note that these coordinates were originally fractional numbers. However, because our cellular automaton operates on a grid with discrete integer coordinates, the coordinates are rounded to the nearest integers. This scenario is saved in the file `task3_scenario.json`. How the scenario progresses can be seen in Figure 4.
After running this scenario, we observe that pedestrians reach the target at **roughly** the same time, but not exactly simultaneously. This is because even though they are initially placed at the same Euclidean distance of 20, diagonal and non-diagonal (horizontal or vertical) steps do not have the same distance. While non-diagonal steps have a length of 1, diagonal steps have a length of $\sqrt{2}$. Therefore, pedestrians moving diagonally reach the target slightly faster than those moving horizontally or vertically.



(a) At the start of the simulation

(b) In the middle of the simulation. The distances from the pedestrians to the target are still roughly the same

(c) At the end of the simulation, all the pedestrians are "absorbed" into the target (removed from the scenario), leaving only the target on the gird

Figure 4: Different stages of the simulation

Speed Adjustment

To implement the specified speeds in configuration files, at each step of the pedestrian, the number of sub-steps we take is equal to the sum of the rounded speed and the rounded accumulated decimal part of the speed. For cases where the speed is a decimal number, we created a new accumulator attribute `speed_dec_acc` in the class `Pedestrian` to store the decimal part of the speed until it reaches or exceeds one. When this happens, the number of sub-steps to iterate in the sub-steps loop increments by one and the accumulator `speed_dec_acc` is then reduced by one after the loop starts.

For example, if the speed is 2.5, we round it to 2 and accumulate the decimal part 0.5 in the decimal accumulator `speed_dec_acc`, which is rounded to zero, rounding the total sub-steps to:

$$round(speed) + round(\texttt{speed\_dec\_acc}) = round(2.5) + round(0.5) = 2 + 0 = 2 \tag{1}$$

In the next step, with speed being 2.5 again, `speed_dec_acc` will become 1 after accumulating the additional decimal part 0.5 from this speed of 2.5. Therefore, the number of sub-steps will now be:

$$round(speed) + round(\texttt{speed\_dec\_acc}) = round(2.5) + round(1) = 2 + 1 = 3 \tag{2}$$

which is one more extra sub-step compared to the previous step. After entering the loop of sub-steps, we decrement the decimal accumulator `speed_dec_acc` by one and have

$$\texttt{speed\_dec\_acc} = \texttt{speed\_dec\_acc} - 1 = 1 - 1 = 0 \tag{3}$$

again.

Compensation of Diagonal Steps

To count the diagonal steps, we add an attribute `diagonal_encountered` to the `Pedestrian` class. In the `update()` method, we use the condition `(x_dif * y_dif != 0)` to check whether the product of the x and y positional differences is non-zero, indicating a diagonal step. If it is not zero, then it's diagonal and otherwise non-diagonal. We increment the `diagonal_encountered` by one whenever this condition is met.

Here, the idea is that diagonal movements should be stopped sometimes to compensate for the fact that a diagonal step has a distance of $\sqrt{2}$. Assuming there is probability p that a diagonal movement is be made (with probability *1-p*, it isn't), we expect:

$$\sqrt{2} \cdot p + 0 \cdot (1 - p) = 1, \quad \text{so} \quad p = \frac{1}{\sqrt{2}} \approx \frac{1}{1.414} \approx \frac{1}{1.5} = \frac{2}{3}$$

This suggests stopping every third diagonal movement. We check for this using the following condition: `(pedestrian.diagonal_encountered != 0 and pedestrian.diagonal_encountered % 3 == 0)`. If true, we do nothing in this sub-step, adding an extra sub-step and continuing to the next sub-step in the loop.

Absorbing Target

In the `update()` method, once a pedestrian enters a target from another cell, it is added to the list of targets entered in this update step `targets_already_entered`. This list helps other pedestrians avoid entering the same target (pedestrian avoidance). The pedestrian's position will be updated to the target's coordinates, and the pedestrian's old position will be marked as empty.

When the update step takes place, and the pedestrian is already on the target, it will be removed from the pedestrian's list. The target would also be added to the list `targets_already_entered` again with the same purpose.

Pedestrian Avoidance

For this task, a method called `get_next_move_position()` will consider neighboring cells and their distances to the targets in the current state of the grid and return the next move of the pedestrian. Since pedestrians update their positions immediately each time they move, neighboring cells occupied by other pedestrians will be excluded from consideration, ensuring that no two pedestrians occupy the same cell.

Report on task 4: Obstacle Avoidance

`Note:` The implementation of Dijkstra algorithm in this task is based on the template code generated by ChatGPT.

In Task 4, two different obstacle avoidance methods are implemented. The first method is simply to give the obstacle cells a very large value of distance-to-target (in our case: `np.inf`) in addition to the distance grid obtained via the naive method. At the beginning of our `get_next_move_position()` method of the class `Simulation`, we filter out the grid cells whose distance-to-target is not smaller than the current cell. By assigning a very high distance value to the target for obstacle cells, we ensure that the pedestrian never steps on an obstacle.

However, when testing this method in GUI simulation, one can encounter "deadlock" situations even if a path exists from the current position of the pedestrian to the target. This occurs because after neglecting the obstacle cells in the neighbors, there can be a situation where a path exists for the pedestrian to reach the target, but that path requires the pedestrian to first step on the neighboring cell whose Euclidean distance is greater than that of the current cell. As a result, the pedestrian will refrain from making this move and remain in place for the rest of the simulation. Such a situation is depicted in Figure 5.

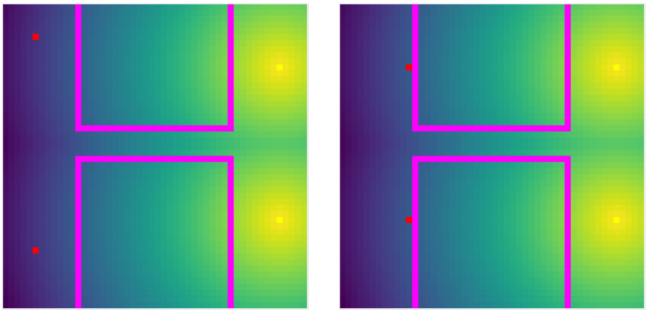


Figure 5: Visualization of the deadlock situation before passing the corridor.

In the chicken test scenario, since the pedestrian is only aware of the existence of an obstacle cell when adjacent to it, the pedestrian will fail to bypass the entire obstacle and instead walk into it, resulting in a deadlock, as shown in Figure 6.

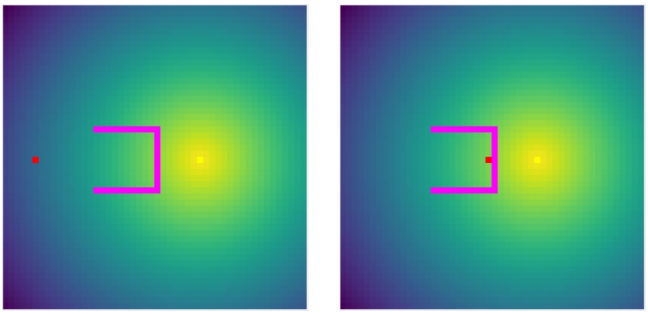


Figure 6: Visualization of the chicken test and a deadlock situation.

The second method is to utilize the Dijkstra algorithm to create a distance grid, which is not based on the Euclidean distance from the cell to the target but on the actual length of the path from the cell to the target. Here, in the context of a grid, we can treat each grid cell as a node, and its eight neighboring cells (2 vertical + 2 horizontal + 4 diagonal) are considered connected to the node by an indirect edge, whose cost is the Euclidean distance between two cell centers. Therefore, there are only two possible values for the edge cost: 1 or `np.sqrt(2)` ($\sqrt{2}$).

In our implementation using the built-in library of Python `heapq`[2], we first set the initial distance value of each cell to `np.inf` (infinity) and define eight possible directions that a pedestrian can make (2 vertical + 2 horizontal + 4 diagonal). There can be several targets on the grid, so we create a distance grid for each target and then keep only the smallest distance value that a cell receives from those distance grids. In the iteration for each target, we first set the distance value of the target cell to 0 and create a queue. Then we iteratively use `heapq.heappop()` function to pop the cell with the shortest distance value in the queue, update the value of its neighbors if necessary, and then add those neighbors to the queue. A valid neighbor should be within the grid's boundary while not being marked as an obstacle in the grid. When updating the distance value of a neighbor, we need to check if this neighbor is a diagonal neighbor or not because it will determine the distance value added to the neighbor. After the queue is empty, we get a distance grid that covers the distance from every cell to the target (unreachable cells will never be updated, so their distance values remain `np.inf`), and then we can proceed to the iteration for the next target.

Since the Dijkstra algorithm already considers the existence of obstacles while creating the distance grid, pedestrians will always be able to bypass the obstacles to reach the nearest target if they start from a position that allows them to do so. However, if they start from a cell that cannot reach any target, i.e., their initial distance value is `np.inf`, then our `get_next_move_position()` method will return the pedestrian's current position, essentially telling the pedestrian to remain in place throughout the simulation.
Figure 7 demonstrates the results of using the Dijkstra algorithm, where bright yellow cells represent those with a distance value of `np.inf`.

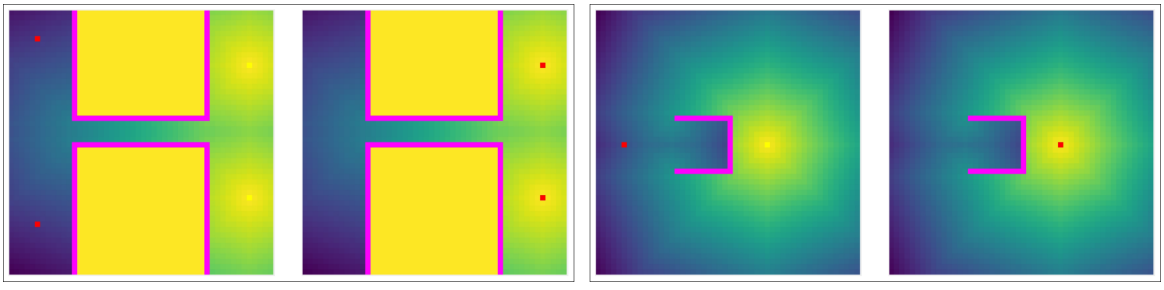


Figure 7: Visualization of the obstacle avoidance using the Dijkstra algorithm.

Report on task 5: RiMEA Tests

[2] `https://docs.python.org/3/library/heapq.html`

Task 5 concentrates on applying RiMEA guidelines[3] to test our implementation for crowd modeling and evacuation simulations, mainly focusing on pedestrian dynamics. These guidelines aim to document the evacuation time, prove the adequacy of the escape and rescue routes and their flexibility in case of unavailability due to an incident, as well as to identify congestion occurring by persons moving along the rescue routes. They are usually used to ensure the accuracy and reliability of evacuation models in safety-critical scenarios, such as public buildings, stadiums, or transportation hubs.

The guidelines contain provisional instructions for the validating simulation programs, which we use to evaluate our application. In Table 1, we list each test introduced by RiMEA and explain whether it is sufficient to be considered in our implementation of a cellular automaton.

In our simulation, we mainly focus on testing the scenarios 1, 4, 6, and 7. All the scenarios can be found in `configs/task5_scenarioX.json` files. The premovement time can be omitted since our scenario assumes that pedestrians start moving immediately upon recognizing a target, and all react uniformly. Varying premovement times might introduce unnecessary complexity without contributing to overall accuracy. Additionally, we need to consider the body size since it determines our cell size in the simulation and is also helpful for measurement purposes.

RiMEA Scenario 1: Straight Line

For the first test of maintaining the specified walking speed in a corridor, assume we have a corridor with only one pedestrian trying to cross it until it reaches the target at the other end of the corridor as depicted in Figure 8.

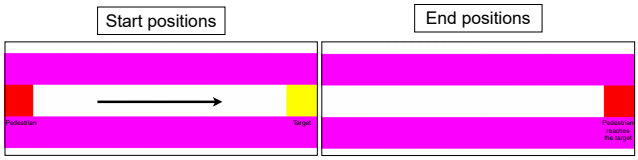


Figure 8: Visualization of the direct corridor with only one pedestrian and a target at the same high level.

Our task is to show that the pedestrian moves at the correct speed. In the configuration file, we defined the pedestrian's speed as 0.555. To have a better overview over time, we set a measuring point to compute the speed and density over the time steps. Figure 9 visualizes how a pedestrian's speed changes throughout the simulation, as well as its density. As a result, we demonstrate that the pedestrian achieves a speed of around 1.0 during odd time steps and a speed of around 0.0 during even time steps. The mean speed equals then approximately 0.5, which proves that the pedestrian moves at the correct speed throughout the simulation.

RiMEA Scenario 4: Plotting a Fundamental Diagram

`Note`: The the fundamental diagram in this scenario is based on the code generated by ChatGPT. You can find it in the folder named `extras` with the file name `fundamental_plot.py`.

Scenario and Visualization

Figures 10 and 11 provide visualizations of the initial and running states of our scenario. In this scenario, we created a 20x20 grid with 20 pedestrians, as well as one wide corridor of width five and one narrow corridor of width one, where no overtaking is possible. We place the target at the coordinates (17, 6), where pedestrians must pass wide and narrow corridor paths to reach it.

We place a measuring point just around the corner where these two corridors connect, i.e. at the coordinates (19,3). It is a vertical measuring point of size 1x5. Figure 12, showing the end state of the scenario, depicts the shape and location of our measuring point area at the top right side, drawn as a blue-striped vertical box.

[3] `https://rimea.de/wp-content/uploads/2016/06/rimea_richtlinie_3-0-0_-_d-e.pdf`

Test	Included in the implementation	Reasoning
1. Maintaining the specified walking speed in a corridor	Yes	See RiMEA Scenario 1: Straight Line about moving on a straight line.
2. Maintaining the specified walking speed upstairs	No	The stairs are not considered in the cellular automata.
3. Maintaining the specified walking speed downstairs	No	The stairs are not considered in the cellular automata.
4. Measurement of the fundamental diagram	Yes	See RiMEA Scenario 4: Plotting a Fundamental Diagram for plotting a fundamental diagram.
5. Premovement time	No	Premovement time is especially essential in order to avoid bottlenecks and situations where multiple pedestrians land on the same cell. The implementation already covers these cases, so the premovement is no longer necessary.
6. Movement around a corner	Yes	See RiMEA Scenario 6: Movement Around Corners about movement around corner.
7./8. Allocation of demographic parameters	Yes	See RiMEA Scenario 7: Demographic Parameters covering demographic parameter simulation.
9. Crowd of people leaving a large public space	Yes	Covered through the absorption functionality of the target. After a pedestrian reaches the target, it disappears, providing space for another pedestrian.
10. Allocation of escape routes	No	In our simulation, every pedestrian moves to the nearest target/exit. The allocation is not considered in this case due to simplicity reasons.
11. Choice of escape route	No	Only the closest target/exit is considered.
12. Effect of bottlenecks	Yes	Covered in Task 4, the bottleneck problem is a critical problem that influences pedestrian dynamics. The bottleneck occurs when a pathway narrows significantly, and the pedestrians must move through a restricted area, such as a corridor, and not get stuck on the way.
13. Congestion in front of a flight of stairs	No	The stairs are not considered in the cellular automata.
14. Choice of route	No	The stair scenario is not supported. Otherwise, a pedestrian always chooses the smallest distance to the target.
15. Movement of a large crowd of pedestrians around a corner.	No	Not considered due to simplicity reasons.

Table 1: List of RiMEA tests and reasoning which tests were included in the implementation of cellular automata.

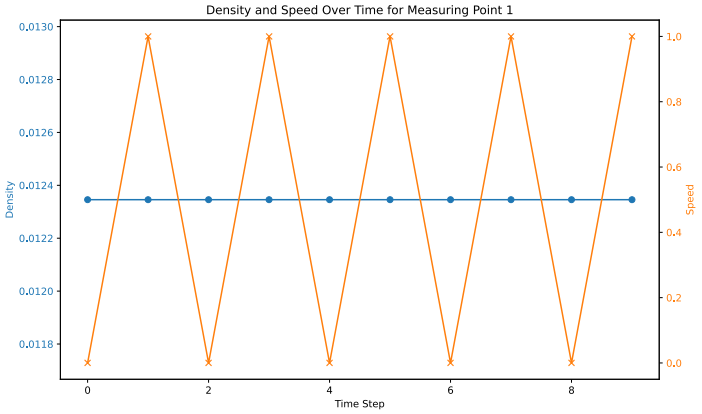


Figure 9: Plot diagram showing speed and density changes over the time steps.

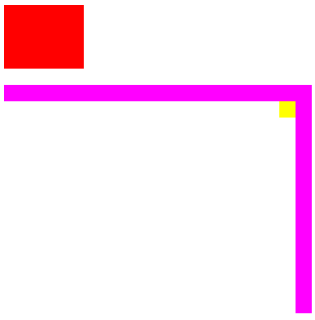


Figure 10: Visualization of the initial state of scenario 4

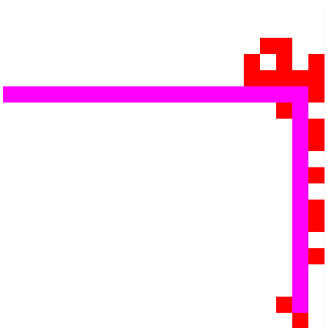


Figure 11: Visualization of the running state of scenario 4

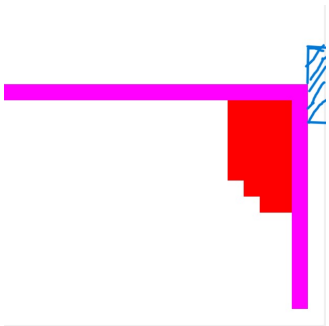


Figure 12: Fundamental Diagram

After the scenario ends and the measuring point collects the crucial data, we plot its fundamental diagram. Figure 13 demonstrates the fundamental diagram of flow and density. The flow is measured as the product of speed and density as follows in our code:

$$\text{flow} = \text{np.array}\left(\text{list}(\text{self.density\_data})\right) \times \text{np.array}\left(\text{list}(\text{self.speed\_data})\right)$$

From the fundamental diagram 13, we see that in the first half of the diagram, until the density is 0.6, the flow increases with density until it reaches a value of around 0.28. After that, increased density causes too much congestion, reducing the flow.

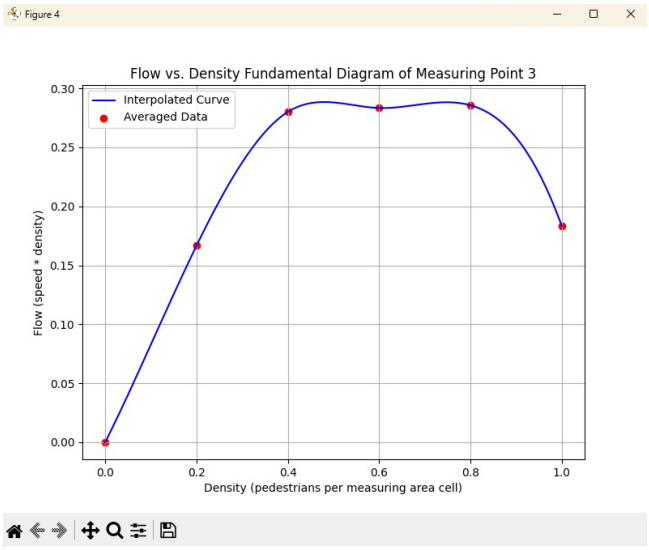


Figure 13: Fundamental Diagram

For scenario 4 of RIMEA and for the fundamental diagram. we implemented the `MeasuringPoint` class, which we explained in detail in the implementation explanation below.

Measuring points are significant for accurately analyzing and modeling pedestrian dynamics in a given space since they allow us to capture critical data at specific locations, like pedestrian density, speed, and flow rates. This information helps comprehend movement patterns, detect potential bottlenecks, and evaluate environmental changes' impact on crowd behavior.

When a measuring point is set and active, depending on its parameters such as delay and measurement time, it tracks the pedestrians passing by, saving the general amount and their speed over the provided time. In the following, we list some of the additional attributes of the `MeasuringPoint` class, except for `ID`, `upper_left`, `size`, `delay` and `measuring_time`:

- `density_data`: Stores density over the measuring period
- `speed_data`: Stores speed over the measuring period
- `measuring_active`: Indicates if the measuring period has started
- `peds_within_area`: Number of pedestrians in the measuring area
- `peds_to_track`: List of pedestrians IDs being tracked
- `peds_total_distances`: Total distance of pedestrians walked throughout the measuring period
- `mean_speed` and `mean_density`: Resulting mean of speed and density throughout the measuring time

Here, we list some additional auxiliary methods of the `Simulation` class, which are used for measuring point logic and will be explained in more detail in the explanation of the implementation section.

- `mp_reset_and_check_active`: Reset attributes of measuring points and check if they are active for the current step.
- `mp_compute_density`: Calculate pedestrian density in each measuring point area if it is active.
- `mp_compute_speed`: Compute the average speed of tracked pedestrians within active measuring points.
- `print_mp_result`: Print the results for each measuring point at the end of its measuring period.

Implementation Explanation

Acquiring Attributes

First of all, we receive the list of measuring points named `measuring_points` during initialization of the simulation through the method `__init__` of the class `Simulation`, where it is also stored as an attribute of this class.

Reset and Checking Activity Status

In each update step of the simulation, i.e. in each call of `update` method of the class `Simulation`, we first reset attributes of measuring points to zero or an empty list and check if they are active for the current step through the method `mp_reset_and_check_active` of the class `Simulation`. The reset attributes are `peds_within_area`, `peds_to_track` and `peds_total_distances`, which are also described above. Furthermore, being active for the current step means that the current step is between delay and the sum of delay and measuring time. Delay is the number of steps to wait before the start of the measuring, and measuring time is the number of steps in the measuring period.

Computation of Density

After the reset and activity check, we calculate pedestrian density in each active measuring point area with the method `mp_compute_density`. In this method, we iterate over all pedestrians and check for each measuring point if that pedestrian is located within its area with the method `is_within_area`. If the measuring point is active and the pedestrian also resides in it, we increase the number of pedestrians within the area `peds_within_area` of this measuring point by one and add the ID of that pedestrian to the list of pedestrians tracked by this measuring point `peds_to_track`. Afterward, we compute the density for each measuring point and add it to the list of densities `density_data`. We calculate the density through the following equation:

$$\text{density} = \frac{\text{peds\_within\_area}}{\text{size.width} \times \text{size.height} - \text{obstacles\_inside}}$$

Density is calculated as the number of pedestrians within the area of a measuring point divided by the number of cells in this area, excluding obstacle cells.

Computation of Speed

Just before the `update` method ends, i.e. after all grid updates and changes in this step are completed, we compute the average speed of tracked pedestrians within active measuring points using the method `mp_compute_speed` `mp_compute_speed`. For each active measuring point, we iterate over the list of pedestrians. If that measuring point tracks a pedestrian, i.e., pedestrian ID is in `peds_to_track`, we add the distance traveled by the pedestrian in this update step, i.e., `distances_made_this_step`, to the total distances made by the pedestrians in this measuring point, i.e., `peds_total_distances`. Finally, we append the average speed of tracked pedestrians to `speed_data`, which is computed as follows:

$$\frac{\text{peds\_total\_distances}}{\text{len(mp.peds\_to\_track)}}$$

We divide the total distance made by tracked pedestrians within the corresponding measuring point by the number of tracked pedestrians. If there are no tracked pedestrians by this measuring point, we directly append zero.

Computation of Mean Flow

We compute the mean pedestrian flow over the measurement period of a measuring point using the method `get_mean_flow`. It is computed as the product of the means of speed and density values it has registered, which is expressed as follows:

$$\text{mean\_flow} = \text{np.mean}\left(\text{np.array}\left(\text{list(self.density\_data)}\right) \times \text{np.array}\left(\text{list(self.speed\_data)}\right)\right)$$

Displaying Results

Finally, just before the `update` method ends, we print the results for each measuring point at the end of its measuring period with the method `print_mp_result`. In this method, we iterate over all measuring points and check if the current step is the end of its measuring period, which is expressed by the following condition:

$$\text{self.current\_step} == \text{mp.delay} + \text{mp.measuring\_time} - 1$$

If this is the case, we compute the mean pedestrian flow over the measurement period of this measuring point with the method `get_mean_flow`. After that, we print the key properties and results of the measuring point as [**ID**, **Mean Speed**, **Mean Density**, **Mean Flow**].

RiMEA Scenario 6: Movement Around Corners

Handling movement around corners is essential for crowd modeling simulations because it allows for a more realistic representation of pedestrian behavior in complex environments. Corners are standard features in urban spaces, buildings, and public areas, and pedestrians' movements around them can significantly impact crowd dynamics.

In Test 6, the main idea is to successfully navigate through the corners with a large amount of people without stumbling. For example, we placed 20 pedestrians in the upper left corner and set the target in the lower right corner. To reach the target, the pedestrians must form a queue and move one after the other. Similar to the bottleneck effect, the test is covered in our simulation and is illustrated in Figure 14.

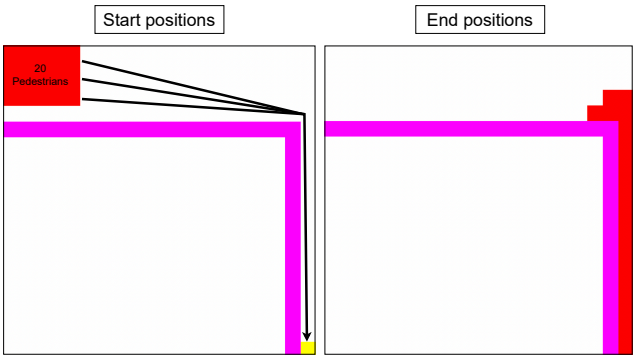


Figure 14: Visualization of crowd movement around the corner, where red squares represent pedestrians, the yellow square indicates the target, and purple squares represent the obstacles

RiMEA Scenario 7: Demographic Parameters

Provided with the approximate mean speed for each age, ranging from 5 to 80, we created a scenario where we put pedestrians between the given ages in the simulation at the same height (left: the youngest, right: the oldest). The targets were set with a distance of 5 between each other, positioned at the same level, without making them absorbing (Figure 15).

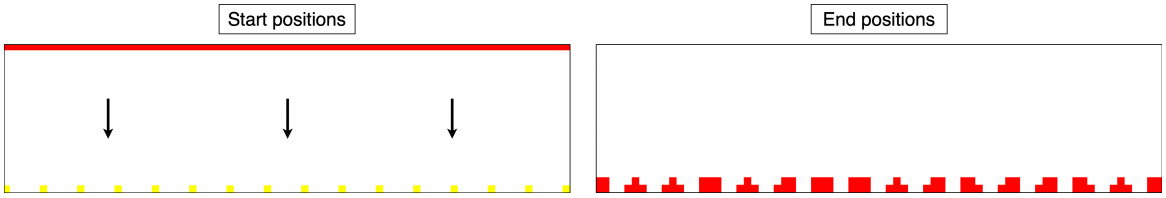
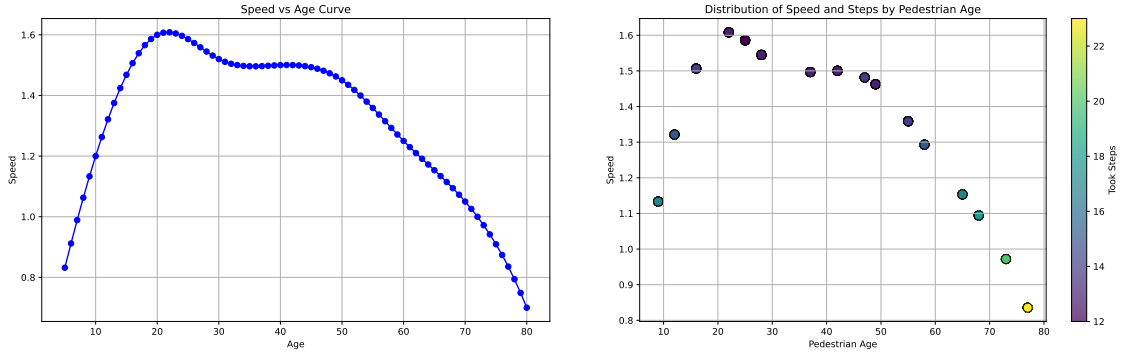


Figure 15: Visualization of testing demographic parameters in our simulation.

After the simulation, we analyzed the list of pedestrians who reached a target and then depicted the results in Figure 16 based on the number of steps taken. As expected, the pedestrians aged between 17 and 43 required the fewest steps to reach the target, while old pedestrians needed the most.



(a) General speed distribution over age

(b) Speed distribution over age after reaching the target

Figure 16: Speed distribution over age in general and in the simulation after reaching the target depending on the steps taken.