Report for exercise 1 from group C

Tasks addressed:	5
Authors:	Muhammad Abdul Moeed (03768384)
	Aleksi Kääriäinen (03795252)
	Danqing Chen (03766464)
	Julia Xu (03716599)
	Joseph Alterbaum (03724310)
Last compiled:	2024–09–27

The work on tasks was divided in the following way:

Muhammad Abdul Moeed (03768384)	Task 1	20%
	Task 2	20%
	Task 3	20%
	Task 4	20%
	Task 5	20%
Aleksi Kääriäinen (03795252)	Task 1	20%
	Task 2	20%
	Task 3	20%
	Task 4	20%
	Task 5	20%
Danqing Chen (03766464)	Task 1	20%
	Task 2	20%
	Task 3	20%
	Task 4	20%
	Task 5	20%
Julia Xu (03716599)	Task 1	20%
	Task 2	20%
	Task 3	20%
	Task 4	20%
	Task 5	20%
Joseph Alterbaum (03724310)	Task 1	20%
	Task 2	20%
	Task 3	20%
	Task 4	20%
	Task 5	20%

Report on task 1, Setting up the modeling environment

1 Modeling environment setup

Task 1 focuses on writing the initialization function of the Simulation class and set up the grid data structure for the simulation of pedestrians. The initialization function gets as input the configuration information for the simulation and these values are copied into the corresponding class attributes. This includes information about the pedestrians, targets, obstacles, grid size, distance computation etc The grid is initialized in the `get_grid` method of the Simulation class. It is a 2-D array which can be described as below:

$$\text{grid} = \{0,1,2,3\}^{w \times h}$$

Where w and h denote the width and height respectively. 0,1,2 and 3 each represent the state of the corresponding cell numerically with the following mapping:

1. '0' represents the state E (empty)
2. '1' represents the state T (target)
3. '2' represents the O (obstacle)
4. '3' represents the P (pedestrian)

This returned array is implemented as a numpy array which is used by the `CellularAutomatonGUI` class to visualize the state of the system.

There is a comment we want to make here. First, the function `get_grid()` in the Simulation class first initializes the pedestrians one by one, then the obstacles and then target. There is an implicit assumption here that pedestrian state is given priority over the other two states. If a pedestrian, obstacle and target all are defined to be on the same cell, then the function `get_grid()` will place the pedestrian value (i.e. '3') there and hence the pedestrian will be visualized on that cell.

Each scenario is created using a config JSON file which contains the list of pedestrians, targets, obstacles, grid size etc. The visualization specifics can be configured in another JSON file where things like the colors of elements, size of cells and simulation time step can be specified.

Report on task 2, First step of a single pedestrian

2 First step of a single pedestrian

In this task we implemented the first naive movement of a single pedestrian toward a single target. On a high level, this is implemented as follows:

1. Find the list of neighbours for the pedestrian. This is done using the `_get_neighbours()` method of the Simulation class
2. Find the best neighbour using a score. Here the score is the euclidean distance of the pedestrian from the target cell.
3. Update the position of each pedestrian in the `update()` method in sequence depending on the `perturb` flag.

The `_get_neighbours()` method lists all the valid neighbour positions of a pedestrian. For a pedestrian at location (x,y), it will look at the locations $(x+1, y)$, $(x, y+1)$, $(x-1, y)$, $(x, y-1)$, $(x+1, y+1)$, $(x-1, y-1)$, $(x+1, y-1)$ and $(x-1, y+1)$. Note that these include the diagonal neighbours as well. The returning list of neighbours will only include the positions will lie inside of the simulation grid.

The `update()` method takes one step of the simulation. One update step of the simulation can comprise of multiple calls of the `step()` method. For this, we have introduced an additional floating point attribute of the `Pedestrian` class named as `distance_to_cover`. This variable keeps a track of the euclidean distance that has been travelled and hence can cater to the speed of the pedestrian. The following algorithm provides pseudo code of how the update function works

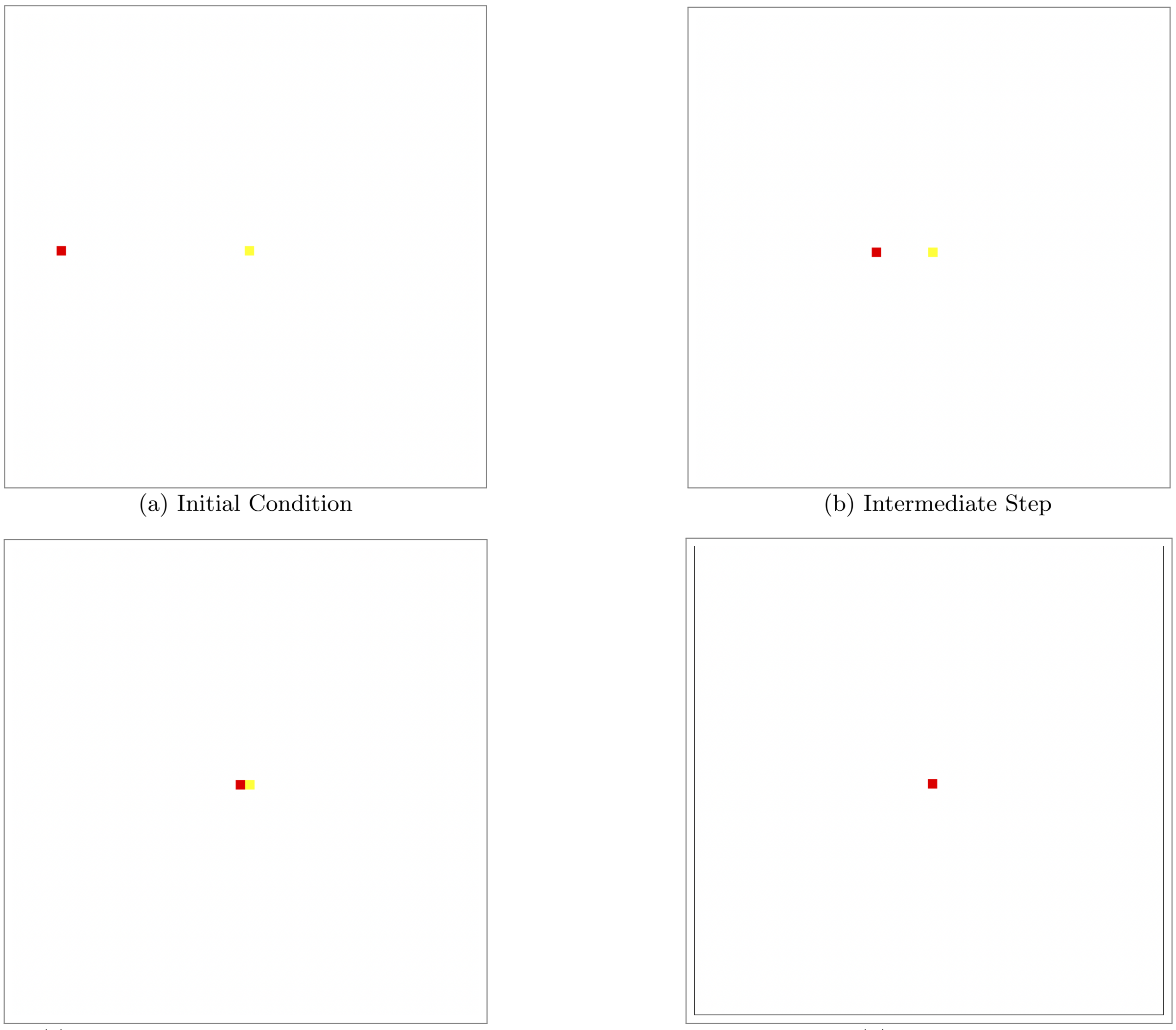
Algorithm 1 Update Step of the Simulation

```
function UPDATE($perturb$)                                    ▷ performs one step of the simulation
    if perturb == TRUE then
        shuffle_pedestrians()                                 ▷ randomly choose pedestrians
    end if
    for each $ped$ in $pedestrians$ do
        $starting\_distance\_to\_cover \leftarrow ped.distance\_to\_cover$
        $ped.distance\_to\_cover \leftarrow ped.distance\_to\_cover + ped.speed$   ▷ speed in cells per time step
        while $ped.distance\_to\_cover >= 1$ and $ped$ $not$ $removed$ do
            $step(ped, starting\_distance\_to\_cover)$        ▷ only step if necessary
        end while
    end for
    $pedestrians \leftarrow$ remove the ones that reached target
    $finished \leftarrow$ true if all pedestrians reached target
    return $finished$
end function
```

The `step` function used in the above algorithm is a helper function we used to move the pedestrian by one time unit. Note that multiple time units can exist in one step of the simulation. For example if the speed is 2 then that means that for each simulation step, the pedestrian should move on average a distance of two, which equates to roughly 2 calls of the inner step function.

Experimental Configuration. To test the implementation of this task, the following experiment was conducted. A 50×50 grid, a single pedestrian at $(5, 25)$ and target at $(25, 25)$. The pedestrian was given the speed of 1. We used Naive distance computation i.e. the euclidean distance for the decision. For each state, the update function lists down the valid neighbours and picks the one that minimizes the euclidean distance to the target. The pedestrian only moves one cell at a time. The GUI can be seen in figure 1. Note that in the end state, the pedestrian is at the target but only the target is being visualized. The colors of target and pedestrian are given in a separate json file. For this experiment, color yellow represents target and red represents the pedestrian.

Figure 1: First Step of a single pedestrian with speed 1



(a) Initial Condition

(b) Intermediate Step

(c) Intermediate step just before target

(d) End state

Report on task 3, Interaction of pedestrians

3 Pedestrian Interaction

In the third task the goal was to implement speed adjustment for pedestrians. In our case speed means the ability to define how long of a distance a pedestrian is able to move in each time window. One unit is the orthogonal distance between two cells. The distance each pedestrian moves in a time window is calculated with the Pedestrian class attributes `speed` and `distance_to_cover`, where `speed` is a floating point value indicating the distance that can be moved in a time window. `distance_to_cover` is also a floating point number, and it's purpose is to act as a running count to keep track of movement already made in a time window. Lines 6-10 in Algorithm 1 show our implementation of speed. In the algorithm, the value of `distance_to_cover` is evaluated continuously in a while-loop, and if the value is larger than 1, i.e. the pedestrian can move at least one cell, then the function `step()` is called. The while-loop enables a pedestrian to move more than once per each time window, or none, if the value of `distance_to_cover` is not large enough.

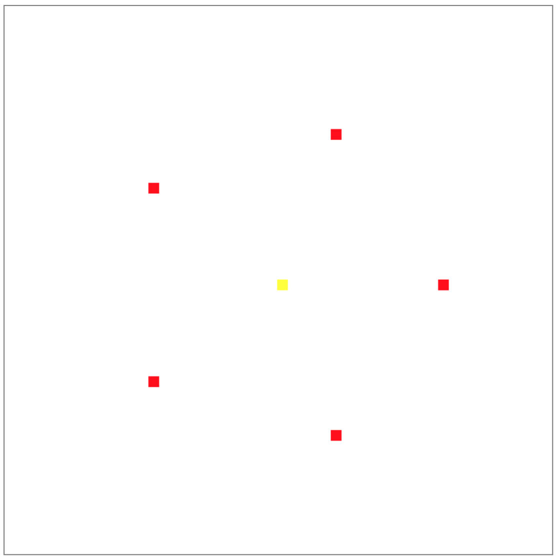
Pedestrian avoidance was also implemented in this task. Pedestrian avoidance is the behavior where the pedestrians preemptively alter their path as not to collide with other pedestrians. Pedestrian avoidance is implemented with the function:

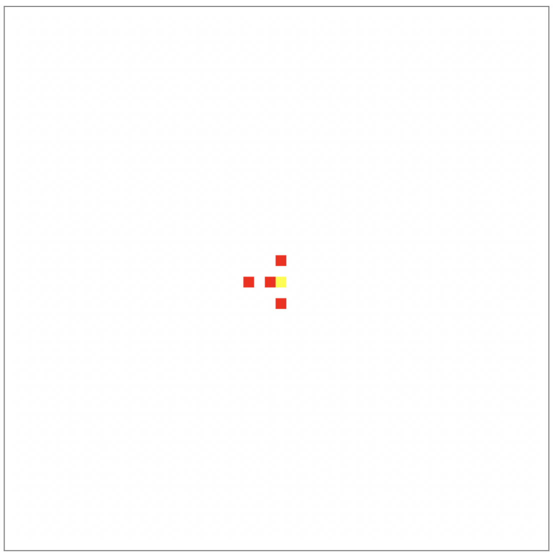
$$c(r) = \begin{cases} \exp\left(\frac{1}{r^2 - r_{max}^2}\right), & \text{if } r < r_{max} \\ 0 & \texttt{otherwise} \end{cases} \tag{1}$$

The cost is a function of r, where r is the euclidean distance between two pedestrians. Changing the value of r_{max} changes the range where pedestrians actively avoid each other. On a pedestrian position update, the cost is calculated for each pedestrian in the simulation, i.e. if there are 5 pedestrians in the simulation, the value of $c(r)$ is calculated 4 times, once for each distance. The actual value of the pedestrian avoidance is thus the sum of these values. In our implementation, r_{max} is set to 1.1, which allows pedestrians to move and reside side-by-side in the neighboring cells, but never residing in the same cell.

The final objective in this task was to define a scenario which shows that pedestrians can move in arbitrary directions in the grid and with the correct speed. The setup for this scenario can be observed in figure 2a. A single target is placed in the middle of the grid. The target is 'absorbing', i.e. pedestrians are removed from the grid on reaching the target. Five pedestrians are placed on the grid roughly at the same distance from the target. The pedestrians are placed in angles $\frac{2\pi \cdot i}{5}$ for $i = 0, 1, 2, 3, 4$ from the target and given speed of 1.

Figure 2: The setup and end state of the circle test


(a) Setup for circle test


(b) State after first pedestrian has reached target

Running the simulation for the given scenario shows that the pedestrians are able to reach the target at roughly the same time, as can be verified from figure 2b. The first pedestrian to reach the target was the right-most pedestrian. At the point when the first pedestrian was absorbed, the other pedestrians have gathered near the target, and will be absorbed in the next couple of update steps. It is noteworthy that in our implementation, only a single pedestrian can be absorbed by the target in a single time window. This means that the earliest time point when all 5 pedestrians are absorbed is 4 ticks after the first absorption. Keeping that in mind, and running the scenario, it can be verified that the time point when all pedestrians are absorbed is exactly 4 ticks after the first absorption.

Report on task 4, Obstacle avoidance

4 Obstacle avoidance

For obstacle avoidance, the utility function is designed to assess the "cost" or "penalty" associated with moving to any given position in the grid, which includes the consideration of obstacles.

The utility function effectively integrates both static obstacles and dynamic pedestrian considerations into the path finding algorithm. By assigning an infinite penalty to obstacles and additional penalties based on proximity to other pedestrians.

The following pseudo code describes the ‘utility‘ function.

Algorithm 2 Utility Function for Pathfinding

```
function UTILITY($pos$)                                        ▷ Calculates utility based on position
    if grid[$pos.x$][$pos.y$] == OBSTACLE_VALUE then
        return $\infty$                                        ▷ Return a high penalty for obstacles
    end if
    $distances \leftarrow$ ComputeDistanceGrid(targets)
    $dist\_to\_target \leftarrow$ GetDistance($distances, pos$)
    $r\_max \leftarrow 1.1$                                    ▷ Maximum radius for pedestrian avoidance
    $ped\_avoidance\_factor \leftarrow 0$
    for each $ped$ in $pedestrians$ do
        $distance \leftarrow$ EuclideanDistance($pos, ped.get\_position()$)
        if $distance < r\_max$ then
            $ped\_avoidance\_factor \leftarrow ped\_avoidance\_factor +$ Cost($distance, r\_max$)
        end if
    end for
    return $dist\_to\_target + ped\_avoidance\_factor$
end function
```

Obstacle Check: The first operation inside the utility function checks if the given position is an obstacle by comparing the grid value at the position `pos` with a predefined `OBSTACLE_VALUE`. If the position is an obstacle, the function returns infinity, indicating that this position is highly undesirable due to its impassability.

Distance Calculation: The function calculates the distance from the current position to all potential targets using a helper function `ComputeDistanceGrid`. This function generates a grid where each cell's value represents the shortest path distance from that cell to the nearest target, providing essential data for the utility computation.

Pedestrian Avoidance: The utility function initializes a pedestrian avoidance factor to zero. It then iterates over each pedestrian to check the distance from the current position to the pedestrian's position. If this distance is less than a specified radius `r_max`, a cost is added to the pedestrian avoidance factor. This cost is computed to increase the undesirability of positions closer to other pedestrians, effectively modeling the tendency of pedestrians to maintain personal space.

4.1 Bottleneck Scenario

Experimental Configuration. A grid of 50 x 50 dimensions is set up for the bottleneck scenario. Originally, as depicted in Figure 3a, the scenario involved 30 pedestrians occupying the left half the room. In our simulation, given the cell size of 1m, we utilize 50 pedestrians, each with a movement speed of 1m per step. The configuration of the walls (depicted in purple) is designed such that the sole pathway to the target passes through a designated corridor. In the graph, the yellow cell stands for target while the red represent pedestrians.

Figure 3: Stages of the bottleneck scenario using Naive pathfinding

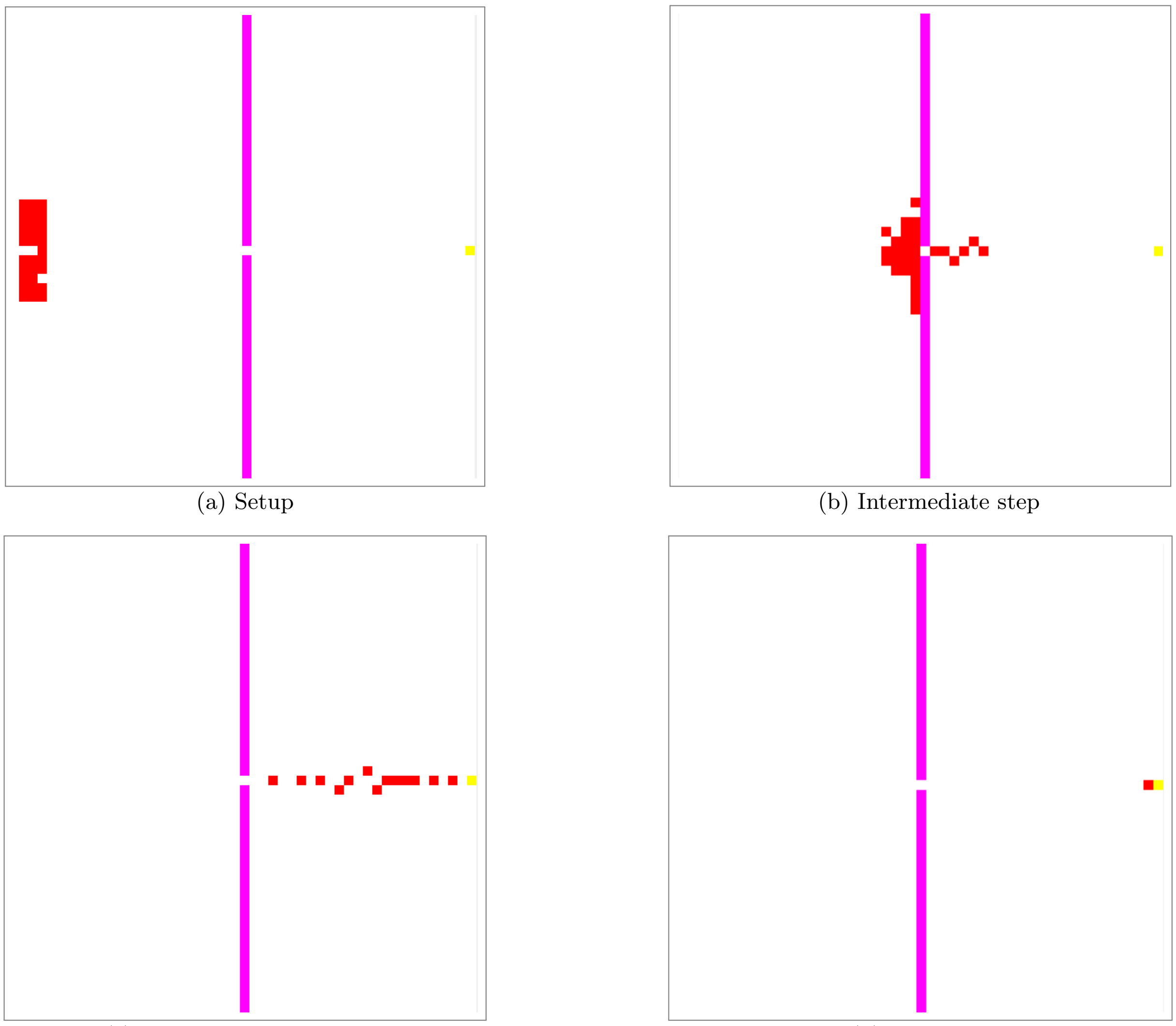

(a) Setup (b) Intermediate step (c) Crossing the bottleneck (d) End state

Figure 3 shows the movements of the movements of the pedestrian through the bottleneck barrier using the Naive path finding. As we can see, the pedestrian can find their way through the corridor to reach the target. The corridor serves to effectively line up pedestrians, helping to alleviate congestion at the exit. This effective behavior from the Euclidean distance algorithm stems from its ability to bypass other pedestrians, which effectively act as dynamic obstacles. Basic obstacle avoidance is integrated into the algorithm, ensuring that any movement is towards a direction that is superior to the current position. However, if obstacles block the direct Euclidean path and there are no alternative cells available to move closer to the target, the Euclidean distance algorithm would fail. In Section 4, the test demonstrates this failure and underscores the necessity for alternative algorithms such as Dijkstra's.

4.2 Dijksttra's algorithm Implementation on Grid.

Path finding for pedestrians in simulations in the last 3 exercises relied on Euclidean distance—a straight-line approach to the target. This method works fine in open spaces, but it hits a snag when pedestrians encounter obstacles that block their path. Since Euclidean distance doesn't account for such barriers, it falls short in complex environments. Enter Dijkstra's algorithm, a game-changer for navigating around obstacles. Unlike the straight-line method, Dijkstra's algorithm calculates a cost matrix that keeps a tally of the price to reach the target from each node, steering clear of any blockages by assigning them a prohibitively high cost. This ensures that when our simulated pedestrians plot a course, they automatically route around obstacles, choosing the path of least resistance to reach their destination. In essence, Dijkstra's transforms our grid from a field of hurdles to an obstacle-free landscape, making for a much smarter and obstacle-aware path finding process.

The `_compute_dijkstra_distance_grid` method applies Dijkstra's algorithm to a grid for pedestrian pathfinding, avoiding obstacles and finding the shortest paths to target positions. The steps are as follows:

1. Initialize a distance grid with infinity values, denoting all cells as initially unreachable.
2. Populate a priority queue with target positions, setting their initial distances to zero.
3. Define possible movement directions, considering both orthogonal and diagonal moves.
4. Iteratively process the queue, updating distances for neighboring cells not obstructed by obstacles.
5. For each cell, calculate new distances. If a cell's new distance is lower, update it and push it back into the queue.
6. Once all cells have been processed, return the distance grid containing the shortest paths to targets.

This approach ensures an efficient path finding solution that dynamically circumvents obstacles on the grid. Pseudo code is provided here 3.

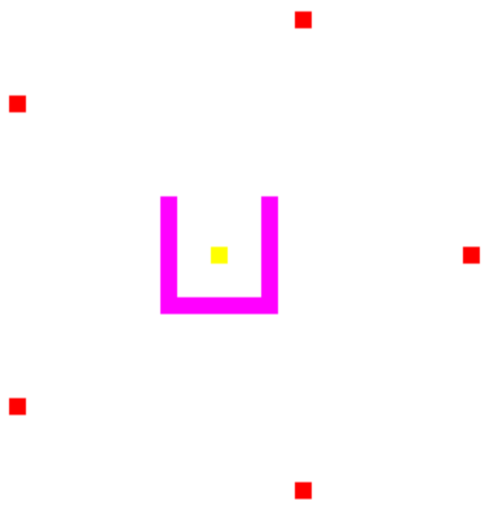
Algorithm 3 Compute Dijkstra Distance Grid

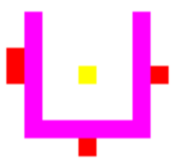
```
function ComputeDijkstraDistanceGrid($targets$, $width$, $height$, $grid$)
    $distances \leftarrow$ array of $width \times height$, initialized to $\infty$
    $pq \leftarrow$ empty priority queue
    for all $target$ in $targets$ do
        if $0 \leq target.x < width$ and $0 \leq target.y < height$ then
            $distances[target.x][target.y] \leftarrow 0$
            add $(0, (target.x, target.y))$ to $pq$
        end if
    end for
    $directions \leftarrow [(0,1), (1,0), (0,-1), (-1,0), (1,1), (-1,-1), (1,-1), (-1,1)]$
    while $pq$ is not empty do
        $(current\_dist, (x, y)) \leftarrow$ pop from $pq$
        if $current\_dist > distances[x][y]$ then
            continue
        end if
        for all $(dx, dy)$ in $directions$ do
            $nx \leftarrow x + dx$
            $ny \leftarrow y + dy$
            if $0 \leq nx < width$ and $0 \leq ny < height$ then
                if $grid[nx][ny]$ is not an obstacle then
                    $new\_dist \leftarrow current\_dist + 1$
                    if $new\_dist < distances[nx][ny]$ then
                        $distances[nx][ny] \leftarrow new\_dist$
                        add $(new\_dist, (nx, ny))$ to $pq$
                    end if
                end if
            end if
        end for
    end while
    return $distances$
end function
```

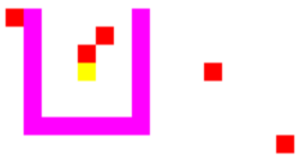
4.3 Chicken Test

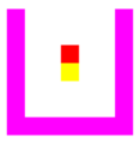
Experimental Configuration. For the chicken test scenario, we constructed a 9x9 matrix. We placed a U-shaped barrier midway(depicted with purple) along the route connecting the pedestrian(depicted with red) and the target(depicted with yellow). The walking pace of the pedestrian was set to a constant 1 meter per move, with the target designed to be absorbent. Scenario files were prepared to test both the straight-line Euclidean distance method (Naive path finding method) and Dijkstra's algorithm. The starting conditions for both experiments, depicted in Figure 4, are identical.

Figure 4: Scenarios using Naive and Dijkstra's algorithms


(a) Chicken Test Setup


(b) The end of the scenario with Naive path finding method


(c) The intermediate state of the scenario using Dijkstra's algorithm


(d) The end state of the scenario using Dijkstra's algorithm, the target is reached by all pedestrians

The pathways generated using both the Euclidean distance (Naive path finding method) and Dijkstra's algorithm are depicted in Figure 4a. While the Euclidean distance algorithm manages to find a route in the bottleneck scenario, it does not succeed in reaching the target when faced with a more intricate obstacle on the pedestrian's route. As shown in Figure 4b, the pedestrian remains stuck in its current location, unable to find a reachable direction where the direct distance to the target is shorter than from its present position.

For the second chicken test scenario, Dijkstra's algorithm is applied when there is a direct obstacle blocking the path between some of the pedestrian and the target. The paths resulting from this application are illustrated in Figure 4c, capturing a moment mid-simulation, and in Figure 4d, showing the moment the pedestrian reaches the absorbable target. The pedestrian evaluates its next move by comparing the Dijkstra scores of the neighboring cells, effectively navigating around the obstacle.

Report on task 5, RiMEA tests

5 RiMEA tests

In the following, the simulation software is tested with four scenarios from the RiMEA-guideline on Microscopic Evacuation Analysis. Therefore, in each scenario, their corresponding assumptions and its implementation are described. The results are then discussed and evaluated regarding cde verification, as well as the comparison of mathematical model & software implementation. Optionally, aspects about model validation and the comparison of software implementation & knowledge about reality are elucidated.

5.1 Test 1/RiMEA scenario 1

5.1.1 Test Setup

In this section, it is to be tested if a pedestrian is moving in a straight line with correct speed. To test a movement of straight line a corridor is introduced. The RiMEA guidelines present a scenario based on average walking speed of a pedestrian which will act as the base for this test's parameters. Table 1 shows a summary of this test's parameters: a pedestrian is walking along a corridor of two meters in width and 40 meters in length. The correct speed is set to the average velocity of 1.33 m/s. The desired outcome is a movement without turns [1].

Parameter	Value
Corridor width	2 m
Corridor length	40 m
Body dimension	40 cm
Walking speed	1.33 m/s

Table 1: Setup of parameters for testing the speed of a straight line-walking pedestrian

The premovement time can be ignored for the tests below since the pedestrian's speed is set once and not changed during a simulation. In other words: for simplification, a pedestrian cannot change its speed after it started walking. The body size (e.g., the cell size in meters), however, is needed to complete the tests since the body's dimensions are smaller than the corridor's length and width.

5.1.2 Test Implementation

To correspond to the scales of the RiMEA guidelines the scale is set to one cell simulating 40 cm (body's dimension) in reality. Therefore, the corridor of 2x40m dimensions is scaled to 5x100 cells. One step in the simulation represents one second in reality. From this scaling it follows that the desired walking speed of 1,33 m/s is translated to 3,325 cells per step in the simulation.

The simulation is set up so that a pedestrian is put into the left side of the corridor while a target is put on the same horizontal lane but on the right side of the corridor. Figure 5 shows an example of the test simulation setup.

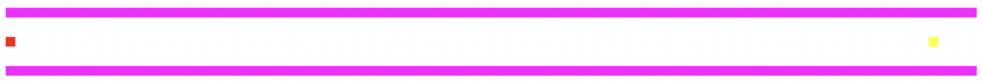


Figure 5: Test 1 Setup: 5x100 cells corridor (purple); pedestrian (red) on the left; target (yellow) on the right

To test the first scenario with both distance computations, this simulation is run twice. Once with setting the distance computation to *"naive"* and once with *distance_computation = "dijkstra"*. This can be done in the configuration file of *task5_1*. The next sub-chapter shows the results of these tests.

5.1.3 Evaluation

By running the simulation, both distance computations successfully cross the corridor in a straight line towards the target. Figure 6 shows an exemplary output towards the end of simulating the straight line walking test scenario.

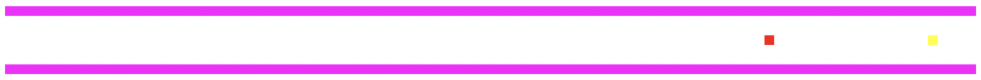


Figure 6: Simulation of straight line walking towards the end.

5.2 Test 2/RiMEA scenario 4

In this section, the main task is to plot a fundamental diagram. The given scenario shown in figure 7 describes a general setup for measurement of fundamental diagram given by the RiMEA guidelines. Its goal is to create a relation between density and flow of a given traffic sequence. Measurement points are introduced to track objects passing them during a given time frame and the flow is calculated by multiplying the density of tracked sequence with its mean speed. A set of density (0.5, 1, 2, 3, 4, 5, 6 [person per square meter]) is to be tested to show its effect on the flow.

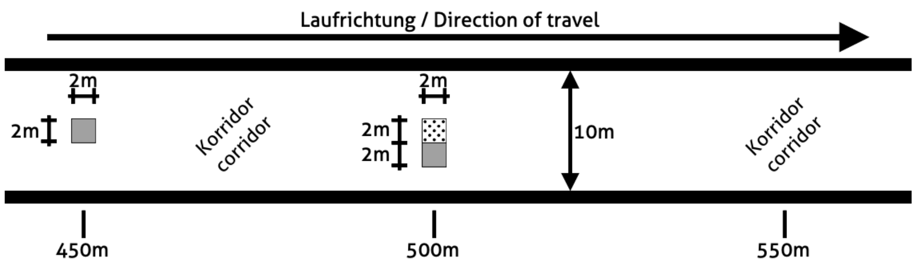


Figure 7: Description of the measurement scenario of Test 4 in the RiMEA-guidelines (measurement points in grey and dotted) [1]

5.2.1 Test Setup

For simplification, the dimensions of the given scenario are modified and reduced to fit our simulation comfortably regarding visualization and aspects of computation time/size. The parameters of this test's setup are summarized in table 2.

Parameter	Value
Cell size	1 m
Body dimension	1 m
Walking speed range	1.2 - 1.4 m/s
Corridor length	20 m
Corridor width	1 m
Number of measurement points	1
Measuring time	60 s
Delay time	10 s
Size of measurement point	20 x 1 m (width x height)
Densities	0.5, 1, 2, 3, 4, 5, 6 [person/m$\hat{2}$]

Table 2: Setup of parameters of one simulation for testing the flow-density values.

To test our simulation with different densities, the corridor measurements are simplified and instantiated with a width of one cell and a length of 20 cells. Furthermore, the scaling is set to one meter per cell here to reduce the complexity of scales through calculations. The passenger's body dimensions are set to the corridor's width (1 cell or 1 meter) to prohibit passengers overtaking each other. Additionally, the number of measurement points is reduced to one (instead of 3 as described in the guideline) and instantiated in a way that it takes the same dimensions as the passageway. This can be justified by the fact that passengers are walking behind each other in a line due to the corridor's dimensions. This way - at any given time - the passengers would have to pass the measurement point anyways to get to the target on the other side of the hall.

5.2.2 Test Implementation

Considering the computation size/time, all density-simulations are run on one single configuration file. The higher the density, the larger is the computation time since more pedestrians have to reach the target.

A configuration file containing all densities (generate_configs.py task_5_2) is created for visualization. Figure 8 shows a pleasant visualization of "crowding factor" given different densities. However, running the simulation with this configuration would simply take too long.

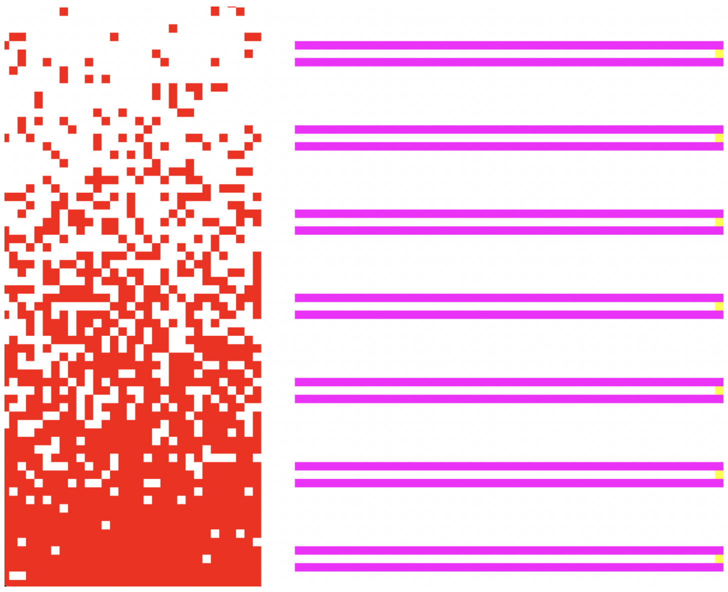


Figure 8: Graphics of the simulation starting with 7 lanes (one for each density). Upper lanes have lower density and lowest lane has the highest density of 6 P/m$\hat{2}$

Therefore another configuration file (generate_configs.py task_5_2_by_density) is created where only one density is being simulated at once. The changing parameters based on the specific density are the measuring time, pedestrian count and density since a higher density within the same geographical frame leads to more pedestrian being computed and therefore needing more time for all pedestrians to have reached the target.

Each configuration file relates to one density with its measurement point's ID being he same value as its corresponding density. Figure 9 shows an exemplary simulation of the test setup of densities 0.5, 1 and 2.

Figure 9: Start of simulations for densities 0.5, 1, and 2.

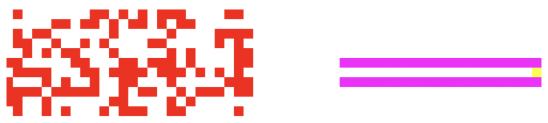
(a) Simulation start of density 0.5.

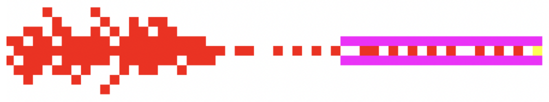
(b) Simulation start of density 1.

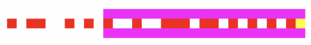
(c) Simulation start of density 2.

The process of the simulation is being displayed in figure 10. It shows how the pedestrians are being spawned into the field and their movement towards the corridor during the simulation. Every pedestrian is successfully walking towards the target via the passageway and no one is detouring. This is guaranteed through careful choices of location spawning.

Figure 10: (a) Start, (b) mid, and (c) end of the simulation for density 6.


(a) Simulation start of density 6.


(b) Simulation of density 6 after roughly 50 seconds/steps.


(c) Towards the end of the simulation of density 6.

A new attribute *measured_pedestrians* is being introduced in the class *Measurement Point* to keep track of collected pedestrians throughout different simulations. Therefore, each configuration file results in one simulation collecting data about the tracked density and flow of a given setup. Here, density is simply the number of pedestrians having passed the measurement point (without duplicates) and flow is being calculated in each step of the simulation by taking its current density and multiplying it with its current mean speed.

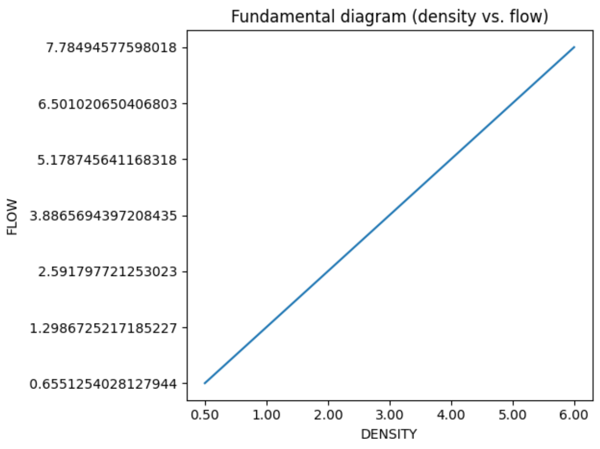
5.2.3 Evaluation

After simulating all scenarios with the given densities, the projected data is collected in a CSV file which is then used to plot the fundamental diagram.

Figure 11: Results of simulation with non-restricting measuring time of 300s.

ID	DENSITY	FLOW
0	0.5	0.6551254028127944
1	1.0	1.2986725217185227
2	2.0	2.591797721253023
3	3.0	3.8865694397208435
4	4.0	5.178745641168318
5	5.0	6.501020650406803
6	6.0	7.78494577598018

(a) Density and flow values of all non-restricted density-simulations.



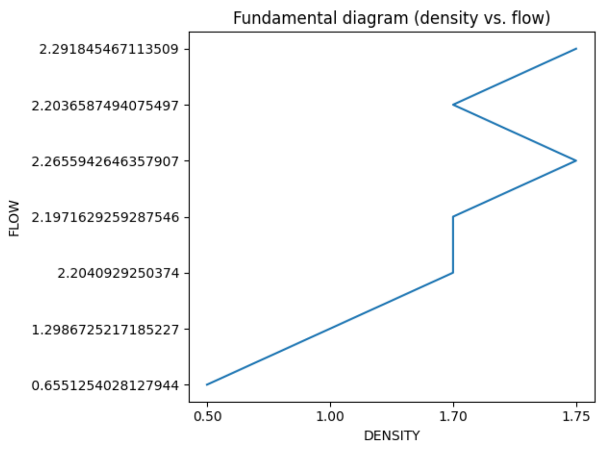
(b) Fundamental diagram of all densities with non-restricting measuring time.

Two sets of simulations have been run to showcase a difference of density and flow. To show, that the densities are being correctly displayed, figure 11 shows that our test implementation computes the correct densities. This graph shows that a non-restraining measuring time leads to identical densities and higher flow values the higher densities reach. The non-restricting measuring time is set high enough (e.g. 300s) to guarantee that the measurement point tracks all pedestrians walking through.

Figure 12: Results of simulation with restricting measuring time of 60s.

ID	DENSITY	FLOW
0	0.5	0.6551254028127944
1	1.0	1.2986725217185227
2	1.7	2.2040929250374
3	1.7	2.1971629259287546
4	1.75	2.2655942646357907
5	1.7	2.2036587494075497
6	1.75	2.291845467113509

(a) Density and flow values of all time-restricted density-simulations.



(b) Fundamental diagram of all densities with limited measuring time of 60s and delay time of 10s.

The fundamental diagram in figure 12b shows the output of a limited measuring time of 60 seconds and a delay time of 10 seconds. Due to the limited time a one-person-only-lane can only provide a density up to 1.75 person per square meter and a max flow of roughly 2.3. The figure clearly shows that densities starting from 2 contract in flow while densities below that threshold still hold their desired densities. Another interesting finding is the zig-zag movement in the right side of the figure. Densities of 2, 3 and 5 show the same resulting density of 1.7 after a limited measuring time while simulations on density 4 and 6 show the same resulting value of 1.75 density post-simulation.

In conclusion, in our case, densities higher than 1 person per square meter lead to a buffered flow with a maximum in reached real density of 1.75. This is not surprising due to the low/narrow space this scenario is using for the corridor and measurement point dimensions.

5.3 RiMEA-Test 6 - Movement around a corner

5.3.1 Test Scenario

In this scenario, the simulation software should achieve smooth movement around a corner specified by the following: "*Twenty persons moving towards a corner which turns to the left will successfully go around it without passing through walls*" [1]. Furthermore, a graphic is provided including measures and the starting range of pedestrians, cp. Fig. 13.

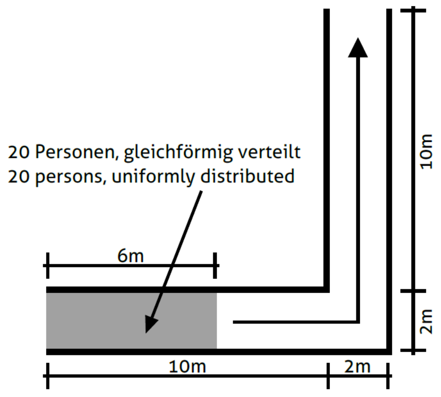

20 Personen, gleichförmig verteilt
20 persons, uniformly distributed
6m
10m
2m
10m
2m

Figure 13: Description of the corner & measurements in Test 6 of the RiMEA-guideline [1]

5.3.2 Test implementation

While the test scenario is rather straightforward, a few assumptions were made in the implementation of the test to account for the usage of a cellular automaton. First of all, we use a 12m x 12m grid in combination with two wall obstacles defining the corridor. The targets are placed as a line at the end (top right) of the corridor, shown in yellow in Fig. 14, while the walls are visualised in pink and pedestrians are shown in red. Since the RiMEA-guideline gave no constraints, walls are assumed to have a thickness of 1 cell and are placed on both sides of the corridor. Furthermore, all pedestrians are initialized with the same speed of 1m/s. Due to the cellular automaton, pedestrians occupy exactly one cell which will become important when discussing the impact of the cell size on the implementation results.

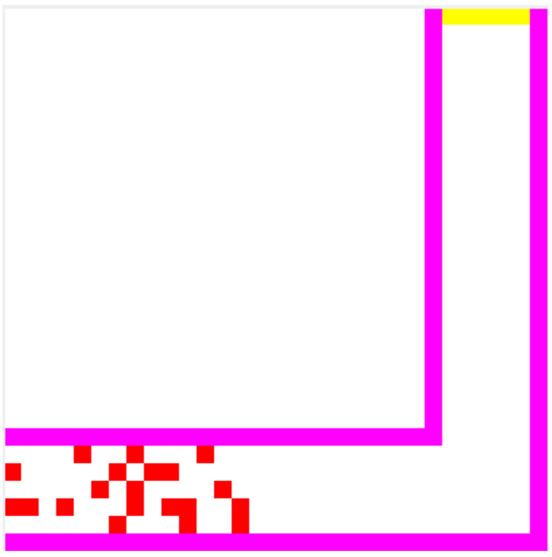


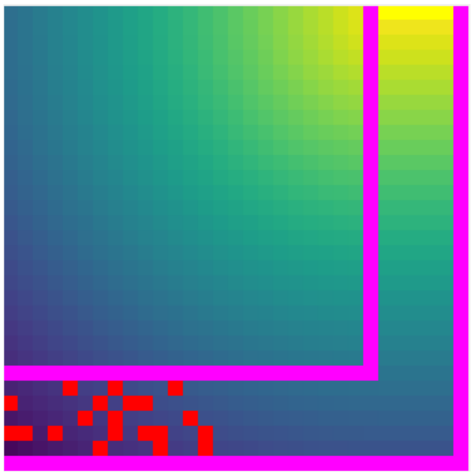
Figure 14: Setup of RiMEA-Test 6 within the simulation's GUI (default cell size 0.4m)

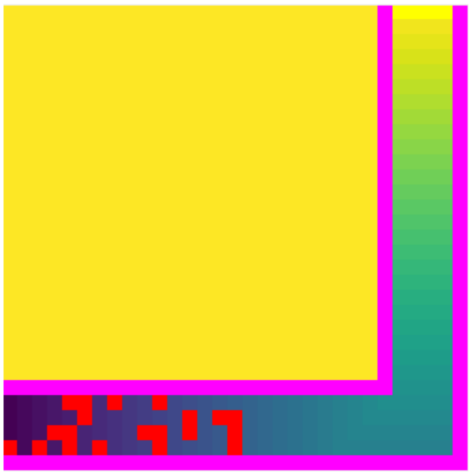
The automatic creation of configuration files for this scenario is realised in the *RIMEA_Test_6_corner*-function of *generate_configs.py*. The main feature is the invariance to changing cell sizes, as all positions (pedestrian starting area, targets & obstacles) are computed relative to the globally fixed cell size. Also, the grid is made one cell bigger than required to absorb the right wall of the corridor. Thereby, the resulting parameters for the default cell size of 0.4m are a 31 x 31 grid (30 + 1) and a corridor width of 5 cells. After the creation of both walls, 20 pedestrians are spawned in the designated 6m x 2m area at the lower left, corresponding to a 5 x 15 cells in the default setup. We achieve the uniform distribution of pedestrians at the start by randomly sampling cells from the starting area. As a result, all of the features described in the RiMEA-guideline were implemented for Test 6.

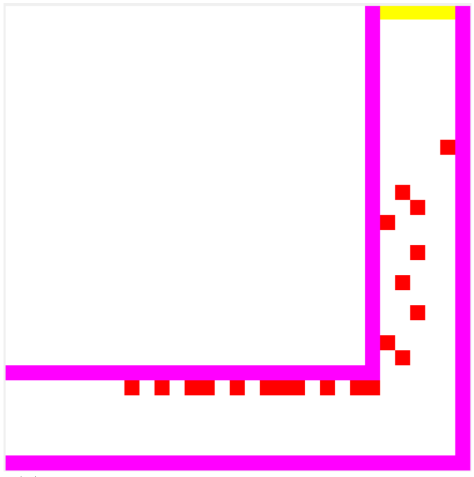
5.3.3 Results

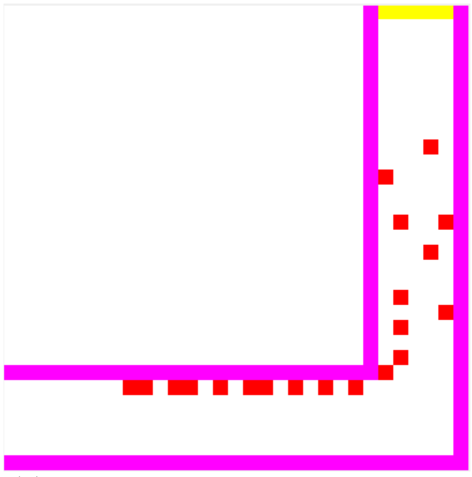
After implementation of the configuration files, the test was conducted several times with two variable parameters - distance measure and cell size. Firstly, we evaluated the behaviour of naive distance computation & Dijkstra's algorithm for the default cell size (0.4m). The corresponding initial configuration can be seen in Fig. 15. Both configurations achieved the desired result when running the simulation: All pedestrians went around the corner, did not pass through walls and eventually reached the target area where they were absorbed. The distance grids shown in Fig. 15a & Fig. 15b seem reasonable since the naive calculation just measures how far the pedestrian is from the target while the upper left part of the Dijkstra distance grid seen in Fig. 15b shows that this part is unreachable. This is exactly the desired "distance" for the area outside of the corridor. Regarding code verification, the simulation shows behaviour in line with the depicted distance grids so no bigger implementation errors are expected.

Figure 15: Visualisation of distance function for default setup of RiMEA-Test 6


(a) Distance grid of default initial setup using the naive method

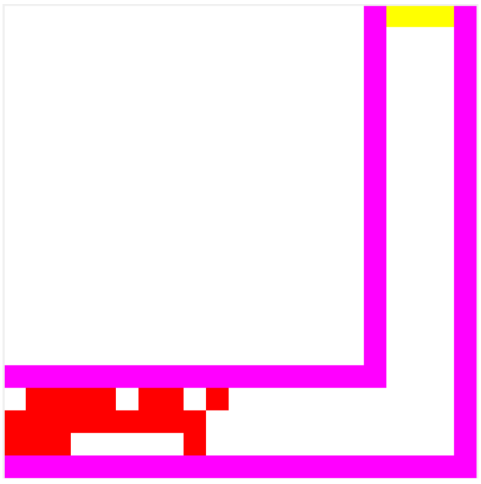

(b) Distance grid of setup using Dijkstra's algorithm

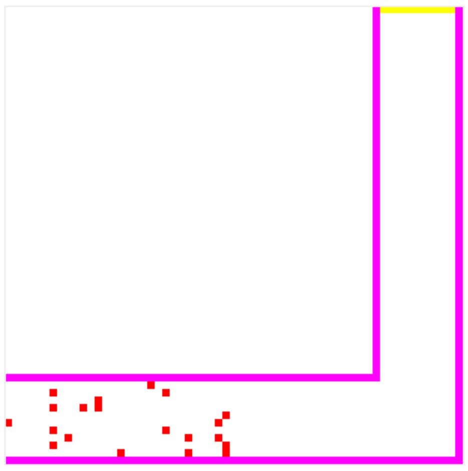

(c) Intermediate step of simulation with naive/Euclidean distance


(d) Intermediate step of simulation with Dijkstra algorithm

The second part of our testing concerned changing cell sizes within the simulation. The two initial configurations are shown in Fig. 16, where Fig. 16a uses a cell size of 0.6m & Fig. 16b a cell size of 0.2m. Again both setups achieved the desired behaviour of not passing through walls and going around the corner smoothly. The results regarding the movement in itself are the same as described in the first part of this section but two interesting observations regarding the cell size and model validation can be made nonetheless. Firstly, a cell size bigger than 0.67m is invalid, since in that case the cellular automaton cannot store enough pedestrians in the specified starting area. This is related to the assumption of one pedestrian per cell or equivalently that a pedestrian is as big as one cell. In this edge case, the mismatch between the mathematical model of discretisation and the real world becomes obvious. On the other hand, although a very small cell size does not impose problems in computation it is not a valid model either. The same assumption of one pedestrian per cell means in this case that a human occupies a spatial are of 0.2m x 0.2m which is much too small for a regular human body. So in conclusion, the cell size has an upper limit from which any cellular automaton is invalid but also a lower limit where the mismatch between model of humans and reality becomes large.

Figure 16: Setup for test 6 with different cell sizes


(a) Initial setup with cell size 0.6m


(b) Initial setup with cell size 0.2m

5.4 RiMEA-Test 7 - Allocation of demographic parameters

5.4.1 Test Description

RiMEA-Scenario 7 tests the behavior of the simulation in the context of specifically distributed pedestrian speeds. The exact specification is the following: "*Select a group consisting of adult persons in accordance with Figure 2 and distribute the walking speeds over a population of 50 persons. Show that the distribution of walking speeds in the simulation is consistent with the distribution in the table.*" [1]. In addition, a graphic, cp. Fig. 17, is given showing the distribution of walking speeds over the age based on a publication by Weidmann [2].

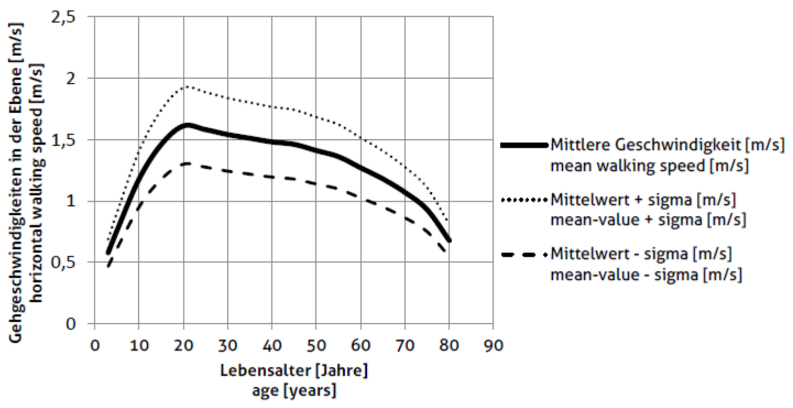


Figure 17: "Walking speed in the plane as a function of age based on Weidmann" [1]

5.4.2 Test implementation

Since the scenario description is rather general, the possible range of complexity in implementation is broad. The pedestrians could for example be simulated as one crowd moving freely within a space, separated from each other by obstacles or even in separate iterations. Moreover, the group of 50 people could be randomly sampled or picked by hand. We chose to simulate the pedestrians separated by walls and not moving freely to make tracking of covered distances easier. In our case, the distribution was given as the speed's mean for 75 different ages which led to the decision to randomly sample 50 pedestrians (avoiding duplication of ages). The RiMEA-guidelines [1] state on page 16 that if no data is provided, the RiMEA standard population consisting of 50% men & women whose age is distributed normally around the age of 50 with a standard deviation of 20 years should be used. We decided to ignore this standard population distribution since it has no impact on

our model achieving the right speeds distributed over age. If a model can achieve the right speed for a uniform distribution over the age it is sensible to assume it does so on a Gaussian distribution over the age as well.
For our simulation setup we used the naive cost function since the distance grid is equal to Dijkstra's algorithm if each pedestrian runs in their own 1-cell-corridor. Initially, a three cell corridor was tested to allow for diagonal movement of each pedestrian. Since our speed adjustment algorithm does not necessarily consider diagonal movement and the grid becomes very big, the corridor width of one cell was chosen instead. In the setup a corridor for each age between 5 and 80 is initialised and if a pedestrian of the corresponding age is sampled it is placed in that corridor. The corridors are ordered ascending such that the leftmost column relates to a 5-year old pedestrian and the rightmost to an 80-year old pedestrian. The automatic creation of configuration files for scenario 7 is implemented in the *RIMEA_Test_7_demographic*-function of *generate_configs.py* and the specific initial configuration used in the following test is depicted in Fig. 18 (note that obstacles are grey in this visualisation for better readability). After sampling randomly from the provided speed distribution (CSV-file), the speeds in [m/s] are converted to [cell/step] with the implicit timing assumption of 1s per step and the globally set cell size (cp. ch. 5.3.2). Afterwards, the pedestrians are initialised into the grid in ascending order of age.

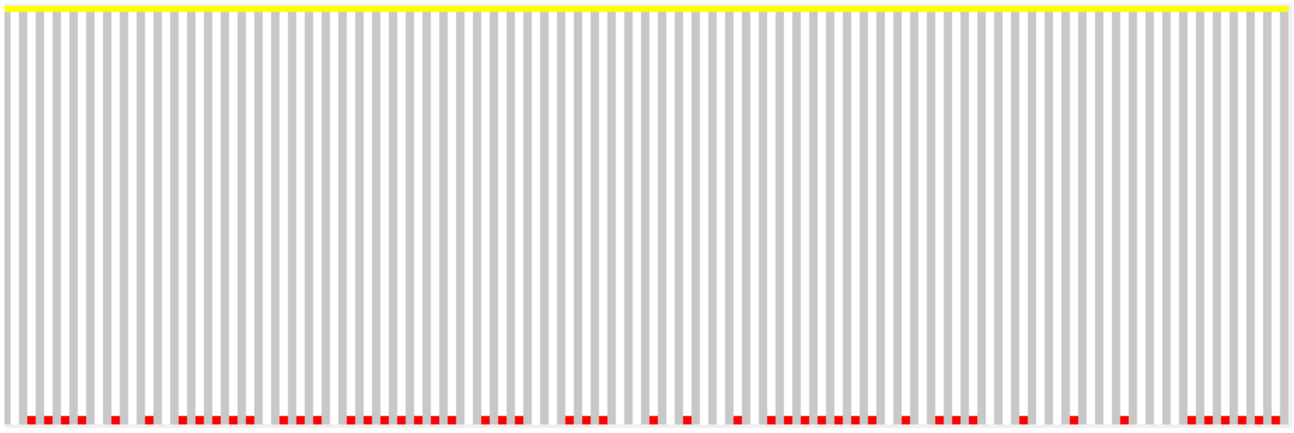


Figure 18: Test setup for RiMEA-scenario 7

To test if the simulated behaviour is similar to the given distribution, we chose to derive the speed from the distance covered by pedestrians in a certain amount of time. Therefore, we stepped the simulation 12 times/12 seconds (until the fastest objects reached the target) and measured the covered distance in cells for each pedestrian manually. The final state of the simulation can be seen in Fig. 19, where three pedestrians just reached the target in yellow. Since the speed distribution by Weidmann was already in .csv-format, we used Excel to calculate the resulting simulated speed in [cells/step] & [m/s] accordingly. Moreover, we calculated the deviation from simulated and desired speed for each age in [%]. Although manual retrieval of the covered distances is in general not the most efficient method, we argue that within the scope of this project it is a valid choice since implementation of automatic stopping & retrieval of covered cells/resulting speeds for each sampled pedestrian takes much longer. Nonetheless, there is massive room for improvement since automatic retrieval of pedestrian states would allow for arbitrarily large corridor lengths and even free movement of samples in a grid without obstacles.

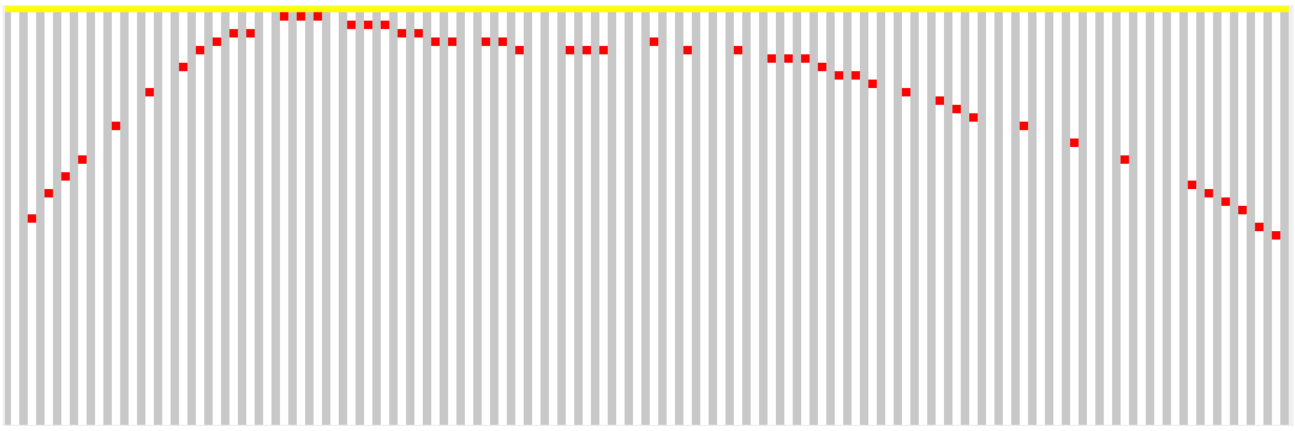


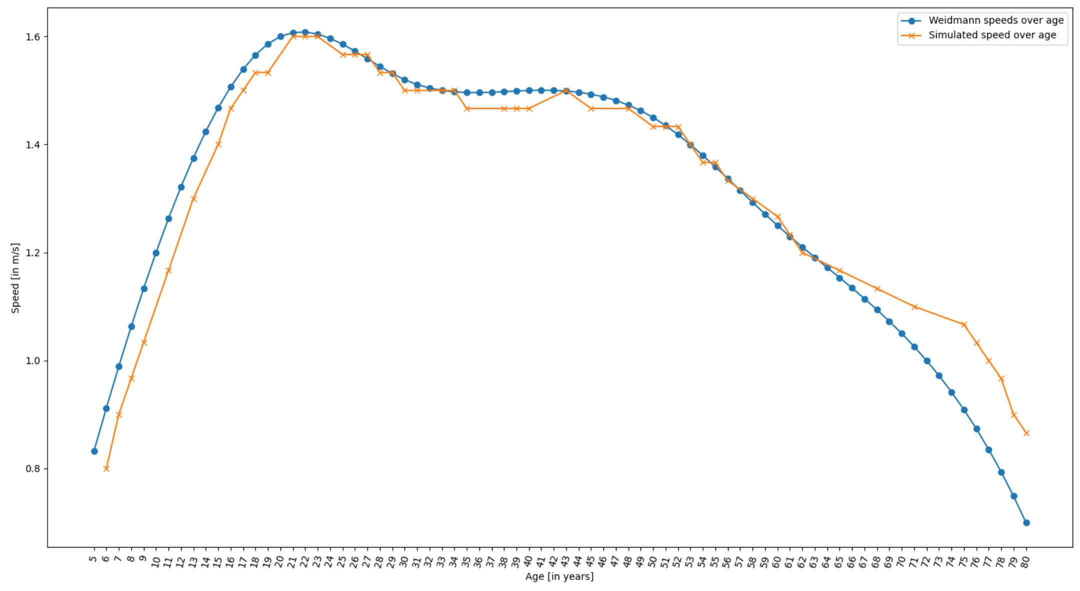
Figure 19: Final state of simulation of test 7

5.4.3 Results

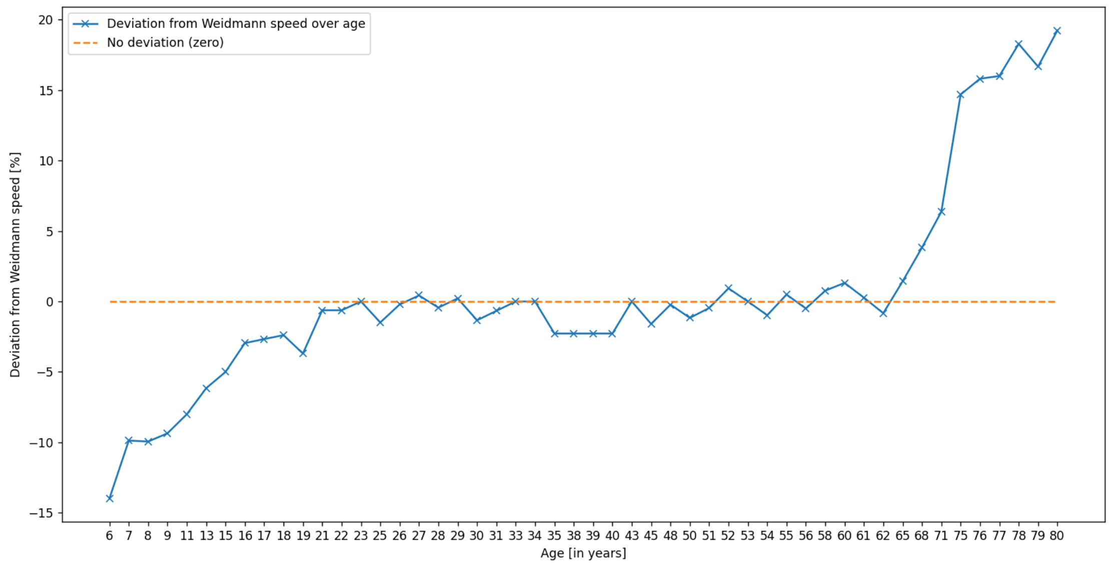
The particular results for our iteration of RiMEA-scenario 7 are stored in the repository at *analysis/RiMEA_scenario_7.csv*. The first colum "*ID*" serves as an index, while the third column "*Weidmann_speed*" stores the given distribution from *configs/rimea_7_speeds.csv* for all sampled ages. Measured and calculated data from our simulation, like the number of covered cells in 12 steps or the resulting speed in [m/s] for each age, can be found in columns four to seven. In order to discuss the results from simulation with our cellular automaton, we implemented the script *result_visualisation.py* in the *src*-folder of the repository. This script plots two figures given the .csv-file with results and the .csv-file with the desired speeds. Firstly, the distribution of simulated and desired speeds over the age is visualised, which can be seen in Fig. 20a. Moreover, the deviation of the simulated speed from the desired speed in % is plotted over all existing age samples, shown in Fig. 20b.
A visual inspection of the simulated speeds (orange curve in Fig. 20a) show that the distribution is roughly the same as the desired speeds (blue curve in Fig. 20a). The only major visual discrepancy occurs at ages over 65 where the simulated speeds start to be significantly higher than the desired values. For a more detailed analysis we will have a look at the deviations from desired speeds (in %) rather than the visual structure of the distribution. Considering Fig. 20b, we can see that for a large portion of the age scale the deviation lies under 5% in both directions (slower/faster). In exact terms this holds for ages bigger than 13 and smaller than 68. While for ages under 13 the simulated speed tends to be lower than the desired speed, it is permanently higher for ages over 68. This is especially interesting in comparison with the distribution of desired speeds in Fig. 20a since the desired speed for both age ranges is similar and varies around 1 m/s.

After describing the results in the previous segment, we now want to discuss them and propose possible reasons for the observed behaviour. Although not perfectly, we conclude that the scenario was passed successfully since the overall distribution of sampled speeds matched the desired distribution. The deviation scatters around 0 for most ages which is expected considering the rather strong discretisation assumption in a cellular automaton. One challenging observation is, that for the same speed values of around 1 m/s the speed deviated in two different directions. In our speed adjustment implementation the distance is accumulated over several time steps until a full cell can be overcome, which produces over- & undershooting a desired distance quite regularly. However, this is not a good explanation for the shown behaviour in ages under 13 and over 68 since we would expect the speed adjustment to be either too slow or too fast for the same absolute value of desired speed. While finding the exact error in implementation or modelling of the system would require more extensive tests, e.g., distributions for even lower speeds or with more samples, a few possible directions can be given. First of all, the performance of the speed adjustment is highly dependent on the length of the covered distance. While for rather short grids (here 50 cells corresponding to 20 m) the over- & undershooting does not compensate, for longer distances the deviations will become smaller. Furthermore, the assumed cell size has extensive impact on the speed since a big cell size produces stronger deviations in each step and a small step size is more precise when estimating the distance that has to be moved. Both effects have more characteristics which depend on the surrounding model assumptions which will not be discussed in-depth here, e.g., a small step size is suboptimal if the pedestrian can move only one cell forward in each time step. Another reason for deviations of the desired speeds in general is the consideration of diagonal movement. While we accumulated movement smaller than the cell size over several steps, we did not account for the distance mismatch when moving in a diagonal direction. This could be improved by changing the speed adjustment implementation. In conclusion, the differing behaviour for the same desired speed could not be explained in detail and it would be interesting to see if this is reproducible for larger populations or different distributions. Overall the test was successful since the distribution was matched within a certain range.

Figure 20: Plots of results from simulation of RiMEA-scenario 7



(a) Plot of absolute simulated speeds and distribution from Weidmann over age



(b) Plot of deviation of simulated speed and distribution from Weidmann over age

5.5 Discussion – conducted RiMEA-scenarios

As introduced in the lecture there is a difference between code verification and model validation. While the first describes comparing mathematical model against software implementation to check the right behaviour of the implementation, the latter means analysing software implementation and knowledge about reality to see if the mathematical model was correct. The RiMEA-guidelines state that testing the components of simulation software is the first integral step towards verifying a simulation program in general [1]. All four tests that were required in Exercise 5 are part of this component testing. After conducting all scenarios successfully, our implementation is partly verified as a correct realisation of the underlying mathematical model/cellular automaton. However, this does not validate the cellular automaton as a correct mathematical model of pedestrian movement in real life. Moreover, the conducted tests were all rather straightforward and are only an excerpt of the 15 tests that the RiMEA-guideline provides in total.

References

[1] RiMEA. (2016) *Guideline for Microscopic Evacuation Analysis.*, RiMEA e.V., 3.0.0 edition.

[2] Weidmann U., *Transporttechnik der Fußgänger*, Schriftenreihe des Institut für Verkehrsplanung, Transporttechnik, Strassen- und Eisenbahnbau Nr. 90, S.35-46, Zürich, Januar 1992.