Report for exercise 1 from group A

Tasks addressed:	5
Authors:	MELIH MERT AKSOY (03716847) ATAMERT RAHMA (03711801) ARDA YAZGAN (03710782)
Last compiled:	2022–05–11
Source code:	https://github.com/mlcms-SS22-Group-A/mlcms-SS22-Group-A/tree/exercise-1

The work on tasks was divided in the following way:

MELIH MERT AKSOY (03716847)	Task 1	33%
	Task 2	33%
	Task 3	33%
	Task 4	33%
	Task 5	33%
ATAMERT RAHMA (03711801)	Task 1	33%
	Task 2	33%
	Task 3	33%
	Task 4	33%
	Task 5	33%
ARDA YAZGAN (03710782)	Task 1	33%
	Task 2	33%
	Task 3	33%
	Task 4	33%
	Task 5	33%

Report on task 1/5: Setting up the modeling environment

python	3.10
numpy	1.21.5
pygame	2.1.2
matplotlib	3.3.4

Table 1: Software versions.

$$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Figure 1: Input state matrix example.

```
0,0,0,0,0
0,1,0,0,0
0,0,0,3,0
0,0,0,0,0
0,0,0,0,0
```

Figure 2: Input text content for state matrix example.

```
1.5
```

Figure 3: Input text content for pedestrian speeds.

In table 1 we list the versions of the software we use in this project.

We use two `.txt` files to read user input. First we need to read the initial state of the state space, a two by two matrix with entries $\{0, 1, 2, 3\}$ which corresponds to the entries on the exercise sheet $\{E, P, O, T\}$ respectively. Here as usual, the state symbols represent:

- $0 == E$: empty cell
- $1 == P$: there is a pedestrian in this cell
- $2 == O$: there is an obstacle in this cell
- $3 == T$: this cell is a target for the pedestrians in this scenario

Theoretically there can be multiple targets in a scenario, but for simplicity we will assume only a single target cell is given by the user in this exercise. This condition is also checked when the user input is parsed, and a proper error message is raised if that is not the case.

For the basic visualization we need the initial state of the state space. We read this required parameter from a text file (specific for each task/scenario) in the `./input` directory which must reside on the root path of the project. Figure 2 shows a valid example input text file. Its corresponding (parsed) matrix is depicted with figure 1. This matrix is created by reading the input file using `numpy.loadtxt`. Additionally we parse this matrix and get a list of pedestrians, obstacles and targets (there should be a single target in the array, we check this condition). From the same example we would get the following lists:

- $pedestrian\_list = [[1, 1]]$
- $obstacle\_list = []$

- $target\_list = [2, 3]$

For our modeling software we set $1m^2$ for each cell, and 1 second for each time step. These are hardcoded and cannot be changed. However the user can provide speed values as meters per second for each pedestrian in an additional text file called `speeds.txt`. Example input text may look like figure 3.

In order to simulate a given scenario to the next time step, we need a proper update function that updates the positions of the pedestrians in the state matrix. In our update scheme we make it possible to move towards one of the eight different directions if there are no obstacles or pedestrians, which can be seen on figure 4.

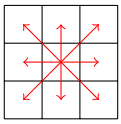


Figure 4: All possible movements from a cell.

Figure 5 explains a simplified version of the loop that we use to update the position of the pedestrians in each time step. One important decision that we make in our implementation is to sort the pedestrians in order of increasing distances to the target cell before updating their positions. Then we iterate this sorted list and update their positions in the state matrix one by one. Reason behind this decision is to avoid pedestrians stumble upon each other and go to a cell that has a less utility. This might for example occur if two pedestrians stand next to each other at positions $[0, 0]$ and $[0, 1]$ with a target cell at $[0, 2]$. In order to move them efficiently we first update the pedestrian at position $[0, 1]$ and then the pedestrian at $[0, 0]$. Otherwise one pedestrian would stay at its position since it cannot move into a cell which is already occupied. So we use the cells strictly, meaning only one pedestrian can occupy a cell at a time. In our implementation each pedestrian is an object and has some properties that we update or initialize such as `speed`, `target_reached` and `carry`. The property `speed` is constant and cannot be changed during the simulation. Other property `target_reached` of type boolean indicates whether the pedestrian has reached the target cell or not. The property `carry` is used to store the additional possible movement left from the previous time steps. Pedestrians do not have a position in the cell. For simplicity we assume that the pedestrians are always standing in the middle of a cell. This makes the distance calculations and the implementation of the update algorithm much simpler.

```
# sort pedestrian_list
timestep = 0
while not finished:
    # draw the scenario with pygame
    for pedestrian in pedestrian_list:
        if not pedestrian.target_reached:
            pedestrian.carry = update(...)
            pedestrian.update_distance_to_target(...)
    # sort pedestrian list
    timestep += 1
```

Figure 5: Main computation loop to update the position of the pedestrians.

When we call the update function it is checked whether the pedestrian can move to the middle of the desired cell. If the distance from the middle of the cell that the pedestrian is currently standing to the middle of the cell that the pedestrian wants to go is smaller than the speed plus the carry value of this pedestrian, then it is moved to the middle of that particular cell. Distance between two neighbouring cells may either be 1 meter or 1.4 meters (for diagonal cells). Figure 6 illustrates the update function called in the main computation loop. In the function we first find the available neighbours (empty neighbouring cells) that the pedestrian can move to. We use the `distance_matrix` to store the Euclidean distances (1) of the cells to the target. We fill this matrix in the `update_distances` function for the neighbouring cells if the entries not filled yet. This way we avoid some redundant computation. Finally the closest neighbour to the target (w.r.t. Euclidean distance between the middle of the cells) is chosen and the pedestrian is tried to be moved to this particular cell in the function `move`. As explained in the previous paragraph, if the sum of `speed` and `carry` values is not sufficient to move to the center of the desired cell, then the pedestrian is not moved to that cell. But we store this additional possible

movement in the variable `movement_left`. Else if the pedestrian can move to the desired cell, then we recursively call the update function since the pedestrian may make another move depending on the `movement_left` variable.

$$d(neighbour_{i,j}, target_{k,l}) = \sqrt{(i-k)^2 + (j-l)^2} \tag{1}$$

In the end we return the remaining available movement value, and in the main loop we store this in the `carry` attribute of the pedestrian. For more detail about our implementation of the cellular automaton please see our code-repository in our github page.

```
update(..., pedestrian, distance_matrix):
    available_neighbours = get_neighbours(...)
    if len(available_neighbours) == 0:
        return pedestrian.carry
    update_distances(...) # updates cell distances to the target (distance matrix)
    # choose the neighbour with the smallest distance to the target
    did_move, movement_left = move(...)
    if did_move:
        # set target_reached property if the pedestrian is on the target
        return update(...) # recursive call
    return movement_left
```

Figure 6: The evolution function.

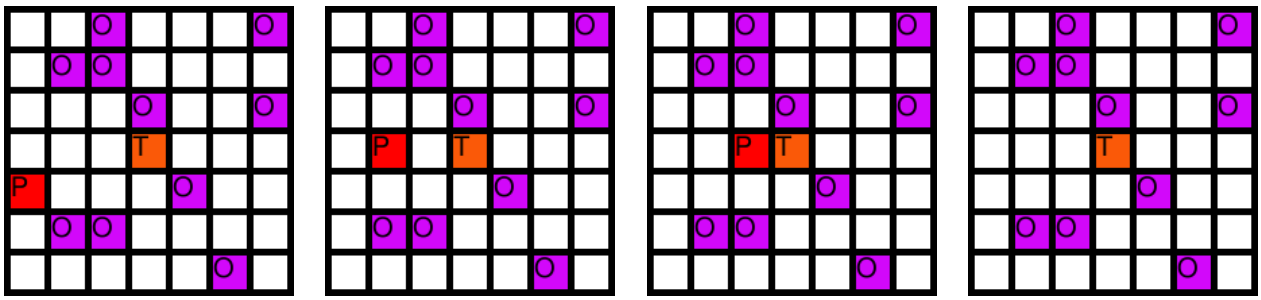
Figure 7 shows an example scenario simulated by our software, visualized using the `pygame` module [2]. The setup is as follows:

- There are obstacles in the scenario (depicted in purple in the figure),
- single pedestrian at position $(5, 1)$ depicted in red,
- single target at position $(4, 4)$ depicted in orange.

The following properties are also same in the other scenarios in this report if not stated otherwise.

- Pedestrian has the speed 1.6 meters per second.
- Each cell covers $1m^2$ area.
- From time step t to time step $t + 1$ corresponds to a second in real life.

One can see the placed pedestrian, obstacles and the target in the simulation. In each time step the pedestrian is able to move to the cell that is closest (in terms of Euclidean distance) to the target with speed 1.6 meters per second where each time step simulates a second. If not stated otherwise we will assume that a pedestrian has a speed of 1.6 meters per second in this report. This is the average of the fast walking pace according to a recent study [1].



(a) timestep 0 (b) timestep 1 (c) timestep 2 (d) timestep 3

Figure 7: `pygame` visualization of a scenario using Euclidean distance [TASK 1]

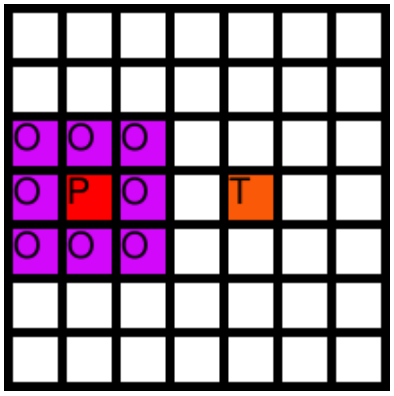
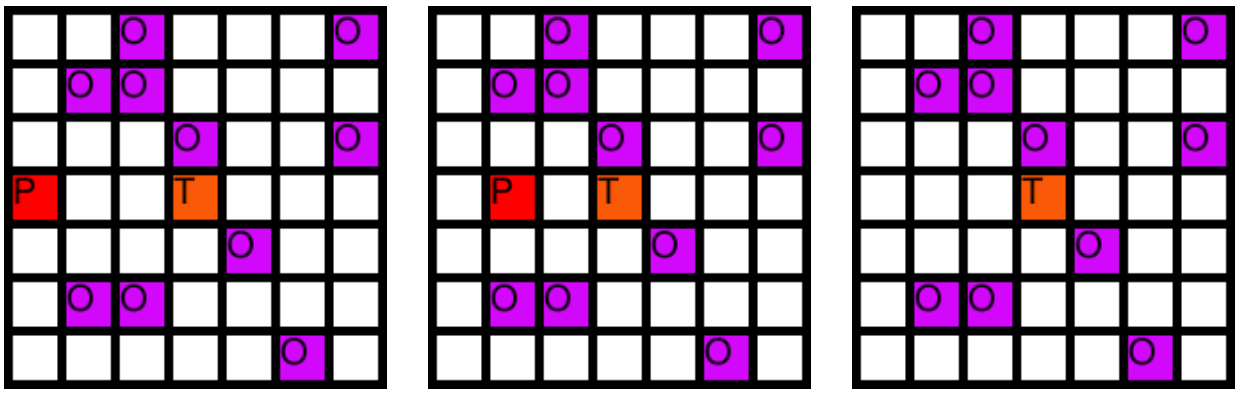


Figure 8: `pygame` visualization of another scenario where the pedestrian is trapped [TASK 1]

Figure 8 shows that a pedestrian can be trapped, i.e. adding obstacle cells makes these cells inaccessible throughout the simulation.

In some scenarios a pedestrian is able move more than one cells in a single time step. This occurs when the sum of `speed` and `carry` values of the pedestrian is larger than the distance between those cells. The scenario in figure 9 is very similar to the scenario in figure 7, but the pedestrian is now closer to the cell $[3, 1]$. Consequently it needs to travel a shorter distance (diagonals cells are 1.4 meters apart, others are 1 meter apart) and this results a larger `carry` value for the pedestrian. So in the next time step the pedestrian can move 2 meters because the sum of `carry` and `speed` at this time step exceeds 2. That is the reason that it moves 2 cells at a single time step and reaches the target at time step 2, whereas the simulation in figure 7 took 3 time steps to finish.



(a) timestep 0 (b) timestep 1 (c) timestep 2

Figure 9: `pygame` visualization of a scenario using Euclidean distance [TASK 1]

In task 2 and 3 Euclidean distance is used as the distance measure for finding the 'best' neighbouring cell to move. In this context this means the cell which has the smallest Euclidean distance to the target cell. We use the Dijkstra algorithm (our implementation) for tasks 4 and 5, which we will explain in further detail in our report for task 4.

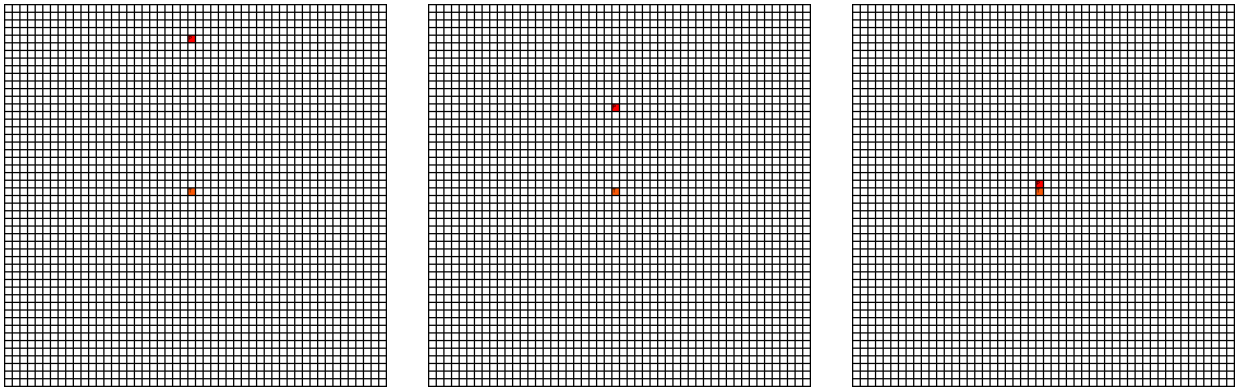
Report on task Task 2/5: First step of a single pedestrian

The initial structure of this scenario is explained in detail in the following:

- The cellular automaton consists of 50x50 grid (2500 in total).
- A single pedestrian is located at position $(5, 25)$.

- There is only a single target at coordinates $(25, 25)$.
- The distance between the pedestrian and the target is 20 cells (20 meters in physical distance since each cell is implemented as 1m x 1m squares).

If we do the calculation, we know that the pedestrian should reach the target in about 12.5 seconds. Since a time step corresponds to a second, we expect our simulation to finish at time step 13 since time steps are discrete values. Figure 10 includes snapshots of the scenario simulated by our software at different time steps. As one can see, the pedestrian approaches the target cell and reaches the target at time step 13.



(a) timestep 0 (b) timestep 6 (c) timestep 12

Figure 10: `pygame` visualization of the scenario on task 2 using Euclidean distance with a single pedestrian at $(5, 25)$, target at $(25, 25)$ [TASK 2]

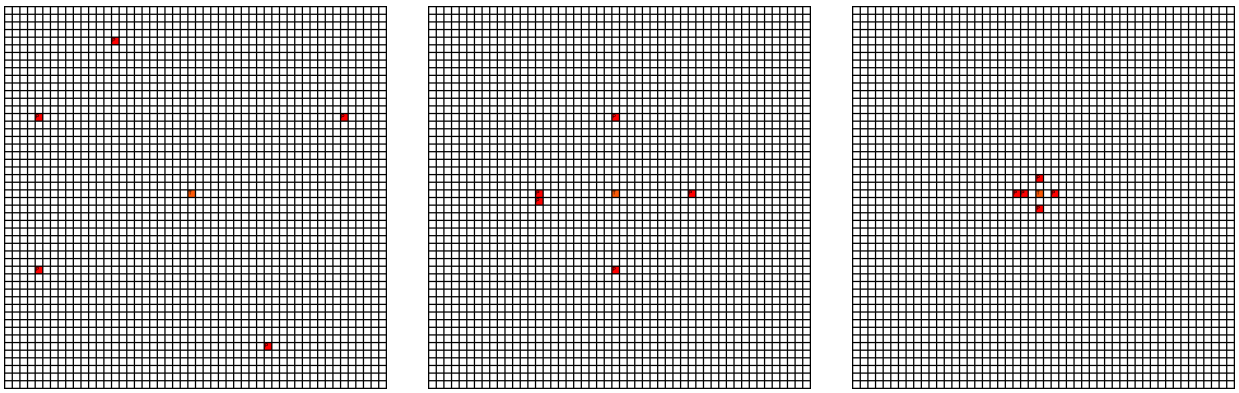
Report on task Task 3/5: Interaction of pedestrians

Pedestrian avoidance tried to be added as an additional cost to the neighbouring cells. However the behaviour becomes unnatural as the pedestrians come close to the target cell. We decided to hard-code the pedestrian avoidance by making the neighbouring cells inaccessible if there is already a pedestrian inside.

In our implementation the target cell is "absorbing" which means the pedestrians disappear when they reach the target cell. Figure 11 contains snapshots of the time steps 0, 9 and 14 of the scenario simulated by our software, as well as here follows a brief description of the initial state of the scenario in this task:

- The cellular automaton consists of 50x50 grid (2500 in total), since this task asks us to use the same basic structure from task 2.
- There are 5 pedestrians at positions $(5, 15)$, $(15, 45)$, $(45, 35)$, $(35, 5)$ and $(15, 5)$.
- There is only a single target at position $(25, 25)$, which is the center of the grid.
- As requested, all pedestrians have the exact same euclidean distance ~ 22.3607 meters to the target cell and they are positioned in a way so that they line roughly on a circle.

One can see that the pedestrians approach the target cell as expected, and the pedestrians on the left of the target cell come close to each other. Since no cost function is taken into account for the pedestrian avoidance, they just avoid stepping onto each other. Finally all pedestrians approach the target cell and enters the target cell at time step 16. At the end of the simulation all pedestrians reach the target cell at time step 15 except the leftmost standing pedestrian in (c). The pedestrian cannot enter the desired neighbouring cell (diagonal cell) at time step 9 depicted in (b) because the cell is already occupied with the other pedestrian. That is why his movement is inefficient in time step 9. At time step 16 all pedestrians reach or "absorbed" to the target cell.



(a) timestep 0 (b) timestep 9 (c) timestep 14

Figure 11: `pygame` visualization of the circle pedestrian scenario on task 3 using Euclidean distance. Pedestrians at $(5, 15)$, $(15, 45)$, $(45, 35)$, $(35, 5)$, $(15, 5)$ and target at $(25, 25)$. All pedestrians have the Euclidean distance ~ 22.3607 meters to the target cell [TASK 3]

Report on task Task 4/5: Obstacle avoidance

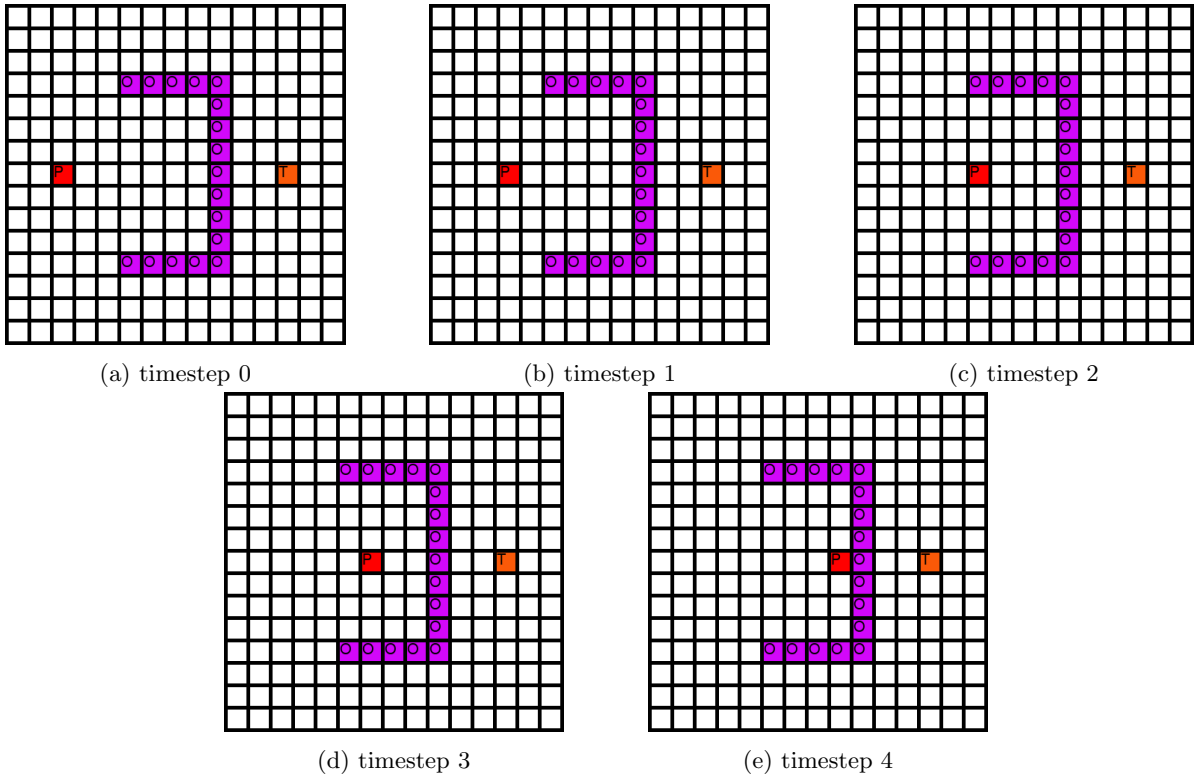
For obstacle avoidance we changed our update (evolution) function scheme. In tasks 1, 2 and 3 we have used the Euclidean distance as a distance measure and a pedestrian always tried to move to the neighbouring cell which has the smallest Euclidean distance to the target. To save some redundant computation we always pass a parameter called `distance_matrix` to the update function and fill its required entries when needed. The Euclidean distance is however cannot handle obstacles in front of the target. Figure 13 includes snapshots of the scenario `chicken-test` simulated by our software. In this simulation we have used the Euclidean distance. The setup of the scenario `chicken-test` has the following initial properties:

- 15 by 15 grid ($15m^2$ area).
- There is a single pedestrian at position $(8, 3)$,
- single target at position $(8, 13)$.
- A wall (filled with obstacle cells) can be seen in the figure (purple color).

This illustrates the problem that the Euclidean distance has when a wall-like object is placed between a pedestrian and the target. Since the Euclidean distance to the target is smaller on the cells which are to the right of the pedestrian, the pedestrian always wants to move towards right. After time step 4 the pedestrian stays at its place since the Euclidean distance of the neighbouring cells to the target are greater than the Euclidean distance of the cell that it is standing in. In this case we had to manually interrupt the simulation.

```
def dijkstra(state_matrix, target):
    # initialize distance matrix D with infinity in each entry
    # set zero distance to the target cell entry
    # build empty priority queue P
    # insert the target cell to the P with key 0
    while not Q.empty():
        cell = Q.get()
        neighbouring_cells = get_neighbours(...) # does not include obstacles
        for neighbour in neighbouring_cells:
            dist = distance(cell, neighbour)
            if D(neighbour) > dist + D(cell):
                D[neighbour] = dist + D(cell)
                # insert the neighbour to the P with key D(cell) + dist
    return D
```

Figure 12: Dijkstra's algorithm. It computes a-priori the `distance_matrix` that we use in the update (evolution) function.



(a) timestep 0 (b) timestep 1 (c) timestep 2

(d) timestep 3 (e) timestep 4

Figure 13: `pygame` visualization of the chicken scenario on task 4 with Euclidean distance [TASK 4]

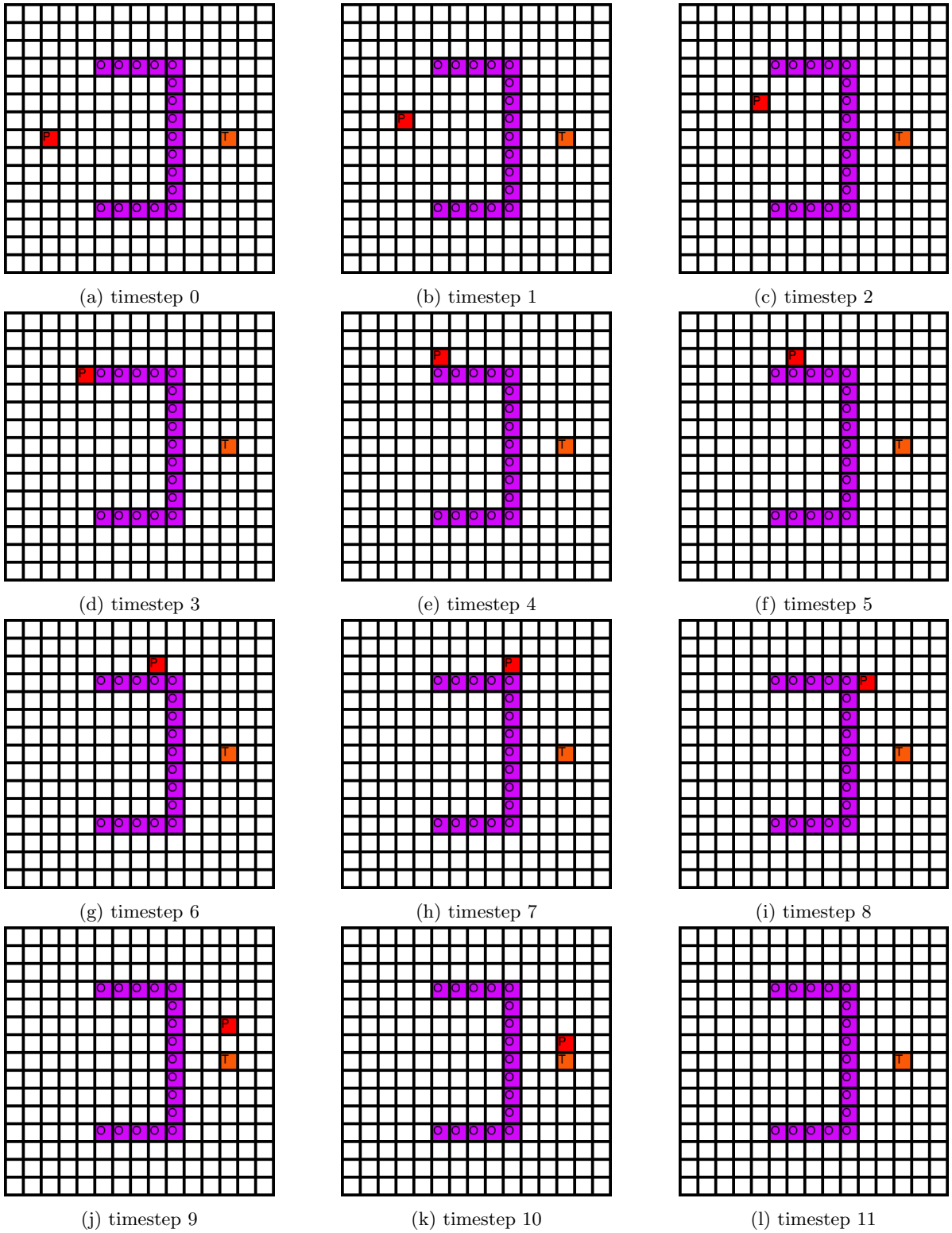
The Euclidean distance needs to be changed so that it takes account of the obstacles around it. Instead, we have implemented the Dijkstra's shortest path algorithm for rudimentary obstacle avoidance. Figure 12 shows how the `distance_matrix` is computed using the Dijkstra's algorithm in our implementation. Since we do not include the obstacles in the distance computation, their corresponding matrix entries remains as infinite values.

The same `chicken-test` scenario results in the states in figure 14 when the Dijkstra's algorithm is used to compute the `distance_matrix` a-priori. Initially the pedestrian starts to move to the upper-right cell at (b), which actually has greater Euclidean distance to the target than the right cell. But the Dijkstra's algorithm updated the distance matrix while constructing the path from the target cell to all the other cells except the obstacle cells. Therefore the distance stored in the `distance_matrix` is smaller for the upper-right cell. In later time steps the pedestrian takes a proper path and manages to move around the obstacle and reach the target. The `chicken-test` scenario is a perfect example to show how Dijkstra can manage simple obstacle avoidance tasks that the Euclidean distance cannot.

Scenario shown in figure (10) of the `RiMEA` guidelines [3] (bottleneck) is simulated using our Dijkstra implementation. Snapshots of time steps from 0 to 11 and 57 to 62 are included in figure 15. The scenario took 62 time steps in total. As one can see in (a) the initial state of the scenario is as follows:

- There are two 10 by 10 meters rooms with a 1 meter wide and 5 meters long corridor connecting them.
- 50 pedestrians are placed at the first 5 meters of the room, covering a $50m^2$ area in total.
- Single target located at the right of the scenario at position $(6, 25)$.
- All pedestrians have the speed 1.6 meters per second.
- 1 second between time step t and $t + 1$.

It is also important to mention again that we update the pedestrians one by one in the evolution function, and we update them in order of increasing distance to the target – in this case distance computed by the Dijkstra algorithm, not Euclidean!



(a) timestep 0 (b) timestep 1 (c) timestep 2

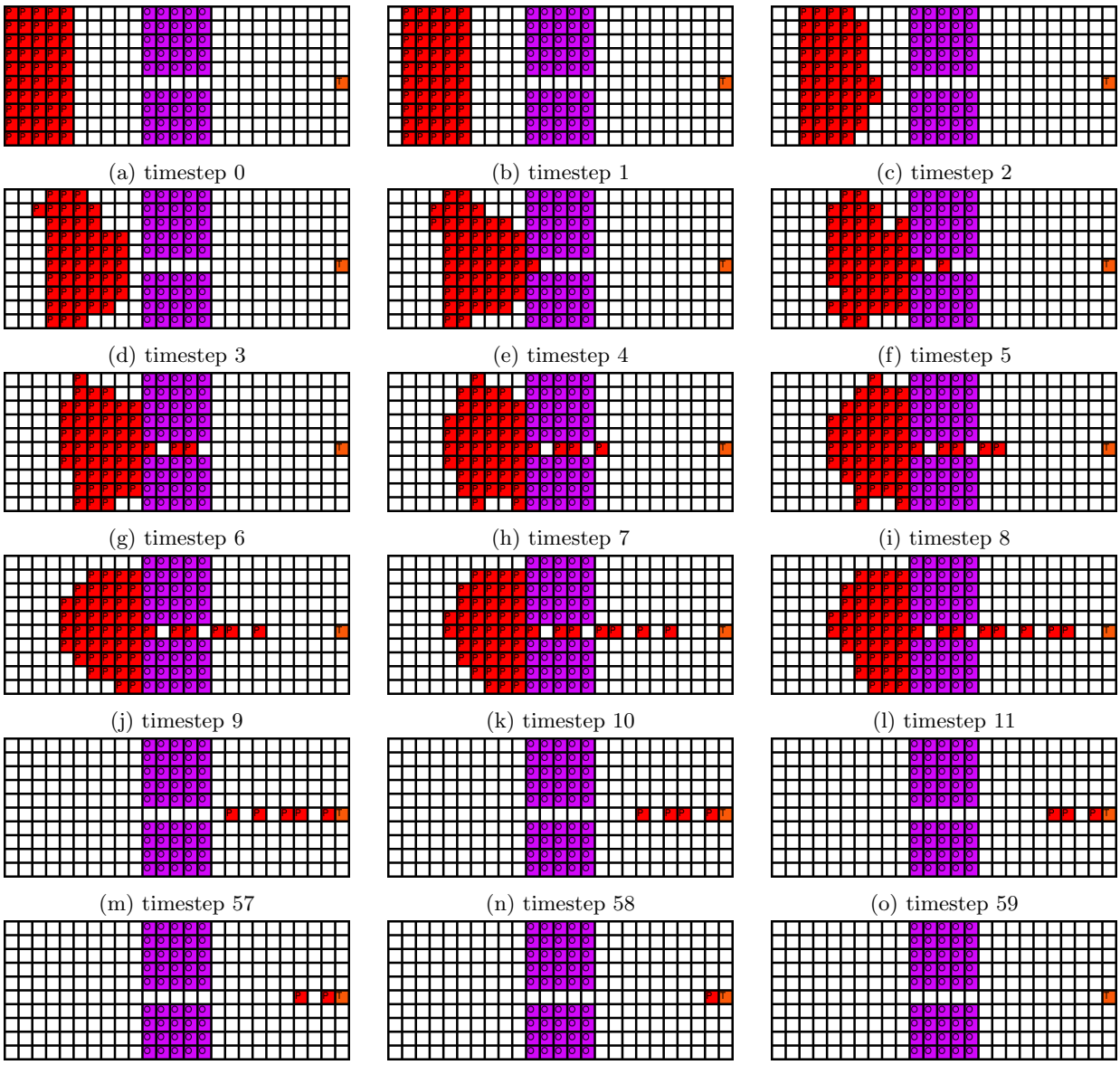
(d) timestep 3 (e) timestep 4 (f) timestep 5

(g) timestep 6 (h) timestep 7 (i) timestep 8

(j) timestep 9 (k) timestep 10 (l) timestep 11

Figure 14: `pygame` visualization of the chicken scenario on task 4 using distance matrix computed by Dijkstra algorithm [TASK 4]

In (c) the pedestrians start to move towards the corridor. Due to the bottleneck at the corridor entry, the pedestrians standing behind cannot move efficiently and therefore have to "wait". In our implementations we have set the `carry` property of a pedestrian to 0 if it does not move. This is a bit similar to a real life scenario



(a) timestep 0 (b) timestep 1 (c) timestep 2

(d) timestep 3 (e) timestep 4 (f) timestep 5

(g) timestep 6 (h) timestep 7 (i) timestep 8

(j) timestep 9 (k) timestep 10 (l) timestep 11

(m) timestep 57 (n) timestep 58 (o) timestep 59

(p) timestep 60 (q) timestep 61 (r) timestep 62

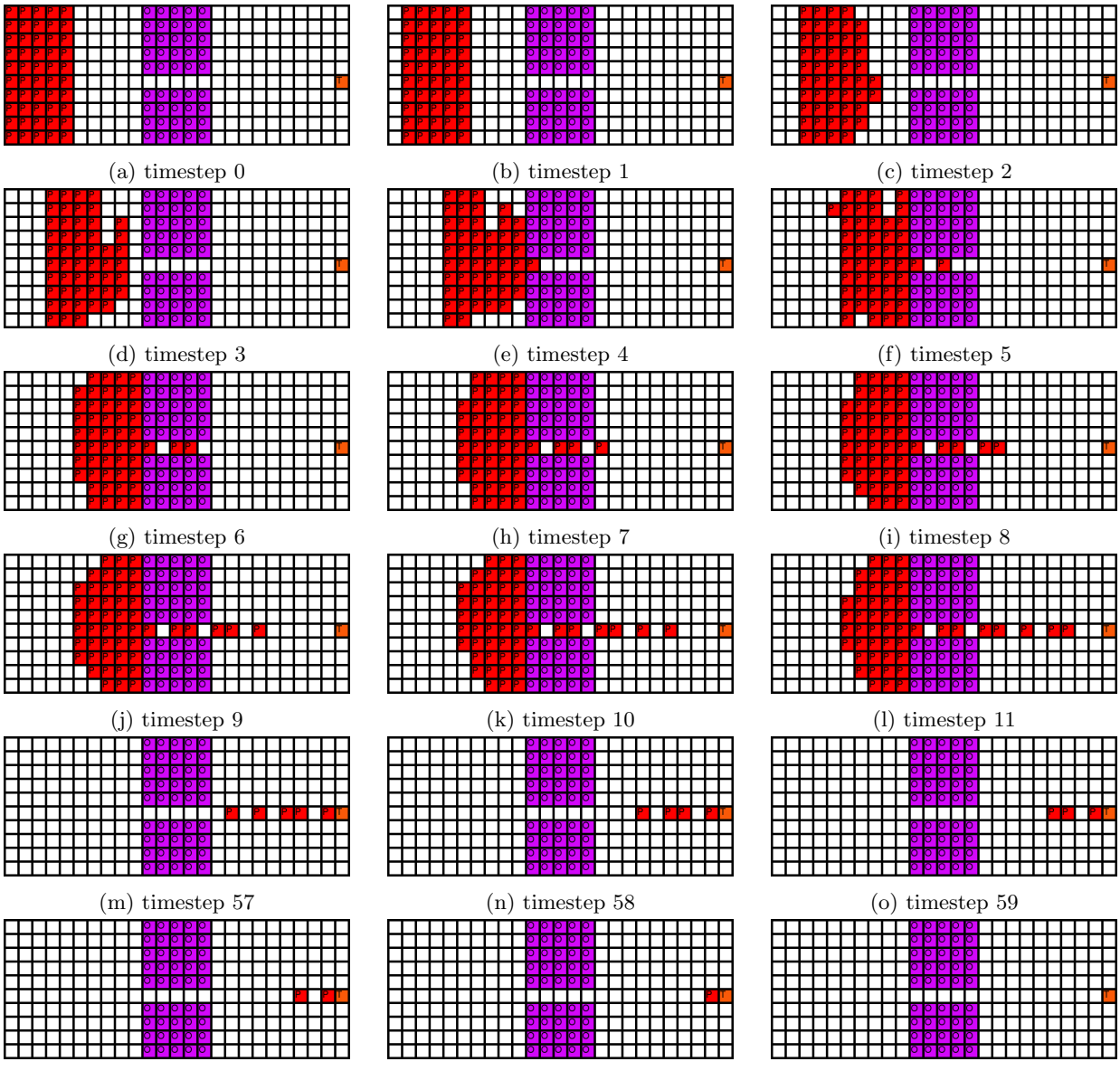
Figure 15: `pygame` visualization of the bottleneck scenario on task 4 using distance matrix computed by Dijkstra algorithm [TASK 4]

since one cannot move at a constant speed and keep its momentum in a crowded area. When the pedestrians enter the corridor they also cannot fill the corridor efficiently and there are some 1 meter long spaces between them while they pass through the corridor. This may also because of the lost `carry` property of some of the pedestrians. After 53 seconds there are no pedestrians in the left room. In about 1 minute all pedestrians left the room – reached the target.

If obstacle avoidance is not implemented (Euclidean distance is used instead of the Dijkstra to compute the distances), all pedestrians are still able to reach the target after about 1 minute. In figure 16 we see the simulation at the same time steps as the previous one, as well as with the same initial state and configurations. All pedestrians managed to exit the room successfully when Euclidean distance is used as well. And it again took 62 seconds as before. So why in the `chicken-test` scenario the Euclidean distance has failed and here it did not? We argue the reason behind this is the position of the target cell. If we consider the neighbouring cells of the entry of the corridor, we notice that the entry has the smallest Euclidean distance than its neighbouring cells that are positioned left relative to the entry cell. This would not be the case if the target is location at

position $(1, 25)$ – at the top-right corner of the room. The pedestrian at position $(1, 10)$ in (f) would never go downwards towards the corridor for example, because the cells downwards would have a greater Euclidean distance to the target than the cell $(1, 25)$. In summary the Euclidean distance does not fail in the `bottleneck` scenario but the simulation fails (infinite loop) when it is used in the `chicken-test` scenario. As a conclusion we can say that using Dijkstra algorithm to compute the distances of the neighbouring cell to the target is generally better for obstacle avoidance.

Instead of using only Dijkstra distances as the cost in the update scheme and avoiding obstacles by eliminating them from the valid neighbours list for each pedestrian, it is also possible to implement an additional cost function to reproduce the effect of obstacle avoidance. The implementation could be similar to one that is explained in the case of pedestrian interaction: Penalize the neighbouring cells of a pedestrian according to their distance to obstacles, if the distance to obstacles is less than a user-defined minimum distance, otherwise do not penalize the cell at all. This cost function then can be simply added to the basis cost function, which only contains the distances calculated by Dijkstra algorithm to add the feature of obstacle avoidance.



(a) timestep 0 (b) timestep 1 (c) timestep 2

(d) timestep 3 (e) timestep 4 (f) timestep 5

(g) timestep 6 (h) timestep 7 (i) timestep 8

(j) timestep 9 (k) timestep 10 (l) timestep 11

(m) timestep 57 (n) timestep 58 (o) timestep 59

(p) timestep 60 (q) timestep 61 (r) timestep 62

Figure 16: `pygame` visualization of the bottleneck scenario on task 4 using Euclidean distance (no obstacle avoidance) [TASK 4]

Report on task Task 5/5: Tests

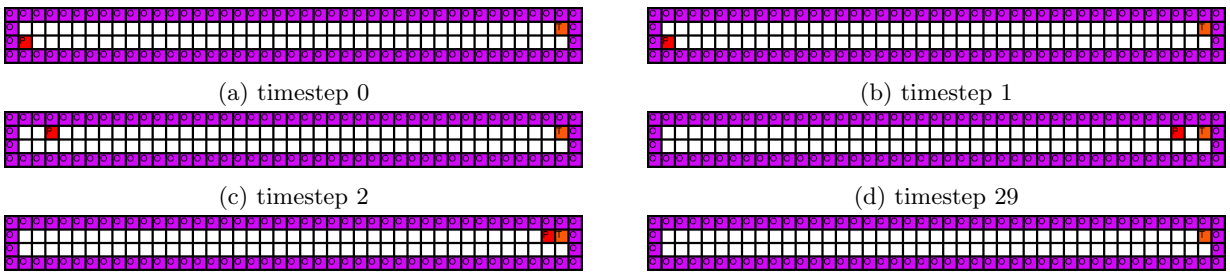
Test 1 (scenario 1) – Straight Line

First test in this task is about proving that a person in a 2 meters wide and 40 meters long corridor can reach the target in the corresponding time period [3]. In the initial setup one can see in figure 17:

- 4 by 43 grid cells.
- Single pedestrian at position $(3, 2)$,
- single target at position $(2, 42)$, 40 meters away from the pedestrian.
- The speed of the pedestrian is 1.33 meters per second.
- Obstacle cells surround the area.

The expected time for the pedestrian to reach the target can be analytically calculated as 30.075 seconds. As one can see in (f) the pedestrian is already absorbed in the time step 31 which means it reaches the target at this time step. The simulation result is in line with the physical calculation since time steps can only simulate discrete seconds. Also this is in the range of the recommended range in the `RiMEA` guidelines, which is 26 to 34 seconds.

In order to pass test 1 we do not need to implement any additional features. Since this is a simple test (not computationally expensive) we also did not neglect any scenario specific properties. Due to limitations of the cellular automaton our simulation cannot capture a realistic visualization. For example one could argue that the pedestrians movements is too deterministic towards the target, however in reality a person may not be able to strictly follow a straight line when trying to reach a target.



(a) timestep 0 (b) timestep 1

(c) timestep 2 (d) timestep 29

(e) timestep 30 (f) timestep 31

Figure 17: `pygame` visualization of the RiMEA scenario 1 (straight line) using Dijkstra [TASK 5]

Test 2 (scenario 4) – Fundamental Diagram

In this `RiMEA` test scenario the main aim is to get the basic relation between flow of pedestrians and the pedestrian density. The initial setup of the test scenario is described in the following:

- A hall/corridor that has width 500 and height 10, note here that in RiMEA guidelines the test is run at a hall with 1000 meters wide, however applying this measure to the simulation software throws an error of "Window size too big!", which is the cause of this size shrink.
- Pedestrians are randomly placed in the hall according to the currently tested density value.
- A single target is placed at position $(4, 250)$, at the middle of the corridor.
- Two measuring lines are added at positions $(\_, 249)$ and $(\_, 251)$ to be able to measure the speed of pedestrians closing to the target both from the left and the right side of the corridor.

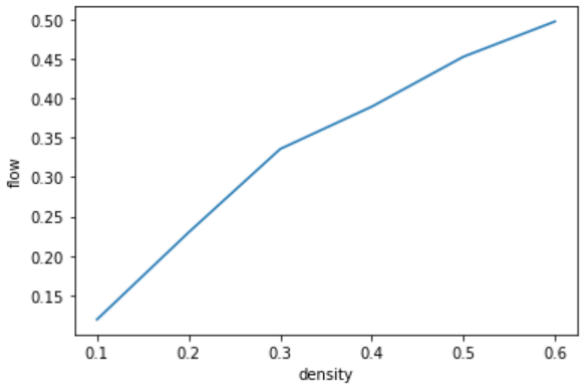


Figure 18: Flow density diagram [TASK 5]

The simulation is run for 6 different density values $\{1, 2, 3, 4, 5, 6\}$, which indicate the percentage of pedestrians in each row, i.e a density value of 1 means that approx. 10% of each row is filled with pedestrians. The simulation then rests for 10 time-steps and starts measuring the current speed of each pedestrian that has reached the measuring line. This measuring lasts for 60 seconds and the simulation is stopped afterwards to calculate the average speed of pedestrians in this period. After getting the result of this calculation, using the relation $flow = density * speed$, flow value for the currently specified density value is obtained. In Figure 18 one can see the graphical relation between the flow value and the density value of pedestrians.

Test 3 (scenario 6) – Movement Around a Corner

Third test in this task is about the ability of the pedestrians to move around a corner successfully – without passing through walls (obstacle cells in purple color). This corresponds to scenario 6 in the `RiMEA` document. In the initial setup of the scenario we have in figure 19:

- 14 by 13 grid cells.
- 12 pedestrians placed at positions from $(12, 1)$ to $(12, 6)$ and from $(13, 1)$ to $(13, 6)$, at the beginning of the corridor.
- A single target is placed in position $(1, 12)$, at the end of the corridor.

Our simulation software already included obstacle avoidance from task 4. As one can see in the figure, the pedestrians are able to move around the corner in the first 11 time steps and then reach the target successfully without moving through any obstacle cell. Evaluation of this test is therefore also positive without any extra features.

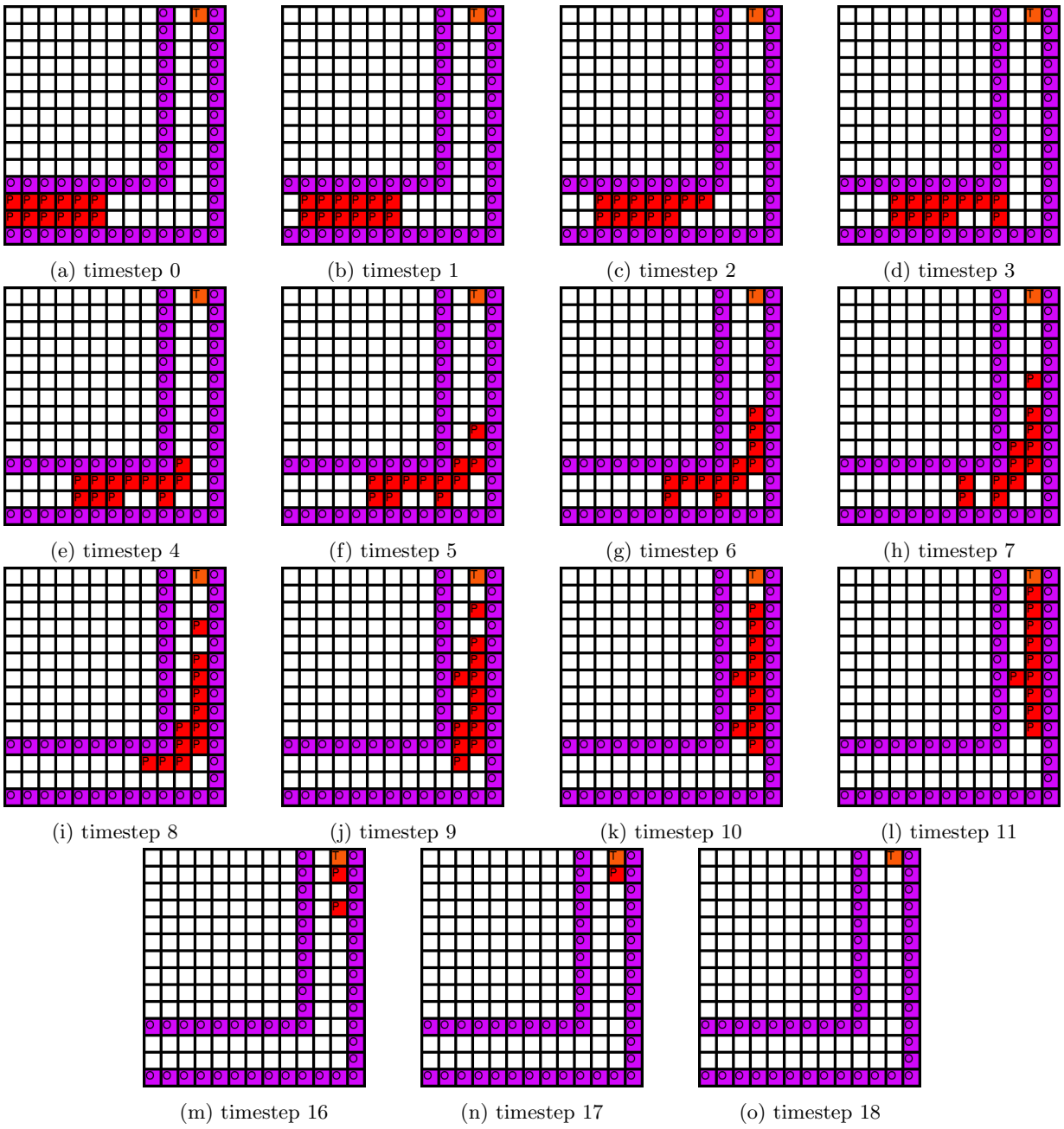
For this scenario we had to reduce the number of pedestrians to 12 because in our simulation software each cell can hold a fixed number of pedestrians – only a single pedestrian per cell. In the `RiMEA` guidelines however the original scenario includes 20 uniformly distributed persons within the same area. We think this does not affect the ability to move around a corner (main purpose of this test) hence this can be ignored.

Test 4 (scenario 7) – Demographic Parameters

In scenario seven of `RiMEA` the speed conservation of people of different ages is observed. In figure 20 the averages of the speeds according to the ages are given with maximum and minimum values. For simplicity the simulation is run for only the people of the following ages instead of continuous values: $\{5, 10, 20, 30, 40, 50, 60, 70, 80\}$.

The start state of the simulation consists of one target, no obstacles and pedestrians with a density of 0.3. A dictionary is used to respect the given speeds in the figure, where the average speeds for each age is stored. After that, the speeds are sampled from a Gaussian distribution with these given means. The same measuring scheme as in scenario four is used to assign the average rates to people of the same age.

The measured speeds are relatively smaller than the speeds in the figure from `RiMEA` guidelines. The main reason for that is the given speed to the pedestrians in our project is actually more similar to the maximum



(a) timestep 0 (b) timestep 1 (c) timestep 2 (d) timestep 3

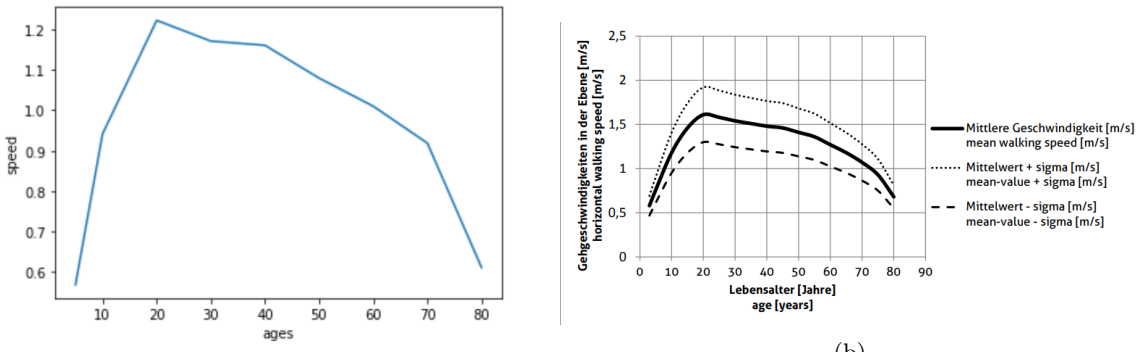
(e) timestep 4 (f) timestep 5 (g) timestep 6 (h) timestep 7

(i) timestep 8 (j) timestep 9 (k) timestep 10 (l) timestep 11

(m) timestep 16 (n) timestep 17 (o) timestep 18

Figure 19: `pygame` visualization of the RiMEA scenario 6 (movement around a corner) using Dijkstra [TASK 5]

speed they can reach throughout the simulation. Because of the pedestrian density the maximum possible speed is hard to reach, and therefore the measured speeds are relatively lower. However the main trend of the given curve in the figure from `RiMEA` guidelines is still preserved by the simulation.



(a) (b)

Figure 20: age-speed functions. (a) is our results from the test case, (b) is from `RiMEA` guidelines [TASK 5]

References

[1] Murtagh, Elaine M., Mair, Jacqueline L., Aguiar, Elroy, Tudor-Locke, Catrine, Murphy, Marie H. "Outdoor Walking Speeds of Apparently Healthy Adults: A Systematic Review and Meta-analysis". In: 51 (Jan. 2021), pp. 125–141.

[2] pygame community. *Pygame:Pygame is a set of Python modules designed for writing video games. Version 2.1.1.* available at: `https://www.pygame.org/docs/`. May 2022.

[3] RiMEA. *Guideline for Microscopic Evacuation Analysis 3.0.0 edition.* www.rimea.de. 2016.