Report for exercise 1 from group G

Tasks addressed:	5
Authors:	Angelos Kafounis Caner Karaoglu Joaquin Gomez Sanchez
Last compiled:	2022–08–31
Source code:	https://github.com/joaquimgomez/ml-crowds-group-g

The work on tasks was divided in the following way:

Angelos Kafounis	Task 1	30%
	Task 2	30%
	Task 3	30%
	Task 4	30%
	Task 5	40%
Caner Karaoglu	Task 1	30%
	Task 2	40%
	Task 3	40%
	Task 4	30%
	Task 5	30%
Joaquin Gomez Sanchez	Task 1	40%
	Task 2	30%
	Task 3	30%
	Task 4	40%
	Task 5	30%

Report on task 1, Setting up the modeling environment

Environment and Project Files

For the development of project tasks we have created the environment and have used the tools listed in the Table 1, being the project robust to minor version changes of each element in the table. Also note that, not every entry of the table is needed for demonstration of the tasks, e.g. PyCharm IDE.

Python	3.8
NumPy	1.22.3
SciPy	1.8.0
Matplotlib	3.5.1
Jupyter Notebook	6.4.11
PyCharm Professional	2022.1

Table 1: Versions of used programming languages, environments, IDE's and libraries

Before we delve into the implementation details and scenarios, we have to define the project files we use and its dependencies. As mentioned, we used Python programming language for the implementation of the tasks. Matplotlib library has been used to visualize and create various figures that you are going to see throughout the report. We have two main source code files, `Automata.py` and `utils.py`, and one Jupyter Notebook file `Exercise1.ipynb`. Each file will be discussed separately in the following sections. As a side note, we have discarded `pygame` and other options in order to put our efforts in the main goals of the exercise, since we started with a version including `pygame`, but it required a specific large amount of development time.

`Automata` class methods

`Automata.py` consists of `Automata` class, which contains methods and data structures that implement the cellular automata architecture. `__init__()` method takes `config` as an argument, that can be a python dictionary or a path to a JSON file describing the scenario. In order to have an efficient performance, instead of one grid, we have three basic data structures: a list with the current position of the pedestrians, a list with the different targets (we contemplate the possibility that each pedestrian has a different target) and one list of obstacle coordinates. This method also initialize a `paths` dictionary, where each element element consists of the pedestrian's ID and the coordinates of the cell she/he has visited; and a dictionary `achievedTargets`, which indicates if the pedestrians' targets are reached or not. Except for the last three items, the other ones are given from an external `config`.

In Figure 1 we can see an example of `config` dictionary (or JSON file) used for this first task. It can be found, together with all the experiments and results in the Jupyter Notebook `Exercise1.ipynb`. We can see that the dictionary contains the name of the scenario, the dimensions of the grid, the list of the pedestrians (coding each pedestrian as `[id, x, y]`, the list of targets (coding each target as `[[list of ids], x, y]`, and the list of obstacles (coding each obstacle as `[x, y]`).

```
configTask1 = {
    "name": "Task1Scenario",
    "dimensions": {
        "width": 5,
        "height": 5
    },
    "pedestrians": [
        [1, 0, 2]
    ],
    "targets": [
        [[1], 4, 2]
    ],
    "obstacles": [
        [2, 2], [2, 1], [2, 3]
    ]
}
```

Figure 1: Content of a `config` dictionary/JSON.

Next we are going to describe the different basic methods of the class:

- `getDimensions()`: it returns the width and height of the grid.
- `getState()`: given the pedestrians, targets and obstacles in the class, as it has been explained, this method returns the current state of the grid. It works as follows: first, initialize a grid using the class scipy.sparse.dok_matrix, which is a really efficient data structure for handling sparse n-dimensional array-like tensors. For this method we pass height and width variables to it. Then, for each of elements lists that we have initialized at the `__init__()` method, assigning integers 1, 2 and 3 to the coordinates of pedestrians, obstacles and targets respectively, on the grid. These integers then will be used for determining which component type is occupying a particular grid location. Lastly, it returns the `grid` as a `ndarray`.
- `getStateWithPaths()`: this method gets the result from `getState()` and adds the path followed by the pedestrians, assigning to each cell of the path a 1.
- `getPaths()`: returns the `paths` dictionary.
- `neighbors()`: it returns all adjacent cells for a particular point in the grid. It also checks that all the returned neighbors are inside the grid using a inner function `validate()`.
- `euclideanDistance()`: given an origin and a target tuples with coordiantes, this method converts them into NumPy arrays and returns the Euclidian distance between these points using `np.linalg.norm()` method.
- `isTargetInTheNeighborhood()`: this method takes all neighbors of a pedestrian and its ID, and then it checks whether one of these neighborhood cells contains the pedestrian's target or not. If we are near to the target, then the pedestrian must stop. This method returns that the target has been achieved and the coordinates of the pedestrian's target. This information is latterly used by the operators to stop the pedestrians setting its instance in `achievedTargets` to True.
- `getUnreachableCells()`: it returns a list with all the unreachable cells, which can be both pedestrians or obstacles.

Following, we are going to introduce the basic operator and the simulate method, implemented as suggested in the exercise sheet.

`basicOperator()` and `simulate()` methods

The core part of this task 1 is the `basicOperator()` method, which resides in `Automata.py`. A more complicated operator, and its related methods, is going to be discussed in the following tasks.

`basicOperator()` starts with an outer loop which iterates on the `pedestrians` list. For each pedestrian we get their ID and their neighbors by invoking the `neighbors()` method by passing the coordinates of the pedestrian. Also, with `isTargetInTheNeighborhood()` we determine if target is achieved or not and we get the target's coordinates.

If we have that the pedestrian has reached the target, then the operator sets its value in the `achievedTargets` dictionary to true and continues with the next pedestrian. If it has not reached the target, we iterate over its neighbors, calculating the distance from these neighbors to the target (using the function `euclideanDistance()`), and discarding those that are unreachable cells (using the function `getUnreachableCells`). During the iteration over the neighbors we look for the one with the minimum distance. This one will be the next position of the pedestrian. As a last step, we save the previous position in the pedestrian path.

`simulate()` method is defined for simulating the cellular automata for a given operator, number of steps, an obstacle avoidance parameter and a pedestrians avoidance parameter. The last two parameters are used to enable/disable the avoidance of pedestrians and obstacles. If all pedestrians reach the target before the given number of steps, the simulator stops and returns the number of steps needed. If all pedestrians reach the target before the given number of steps, then the simulator stops and returns the number of steps needed. If not everyone reaches the target, then the simulator consume all the steps.

We also offer the method `simulateAndVisualize`, which is essentially the same as `simulate()`, but it prints on the screen the state of the automata for every step.

We will also discuss other methods when they came into play on the project. Also an operator with a cost function `operatorWithCostFunction()` will be discussed later.

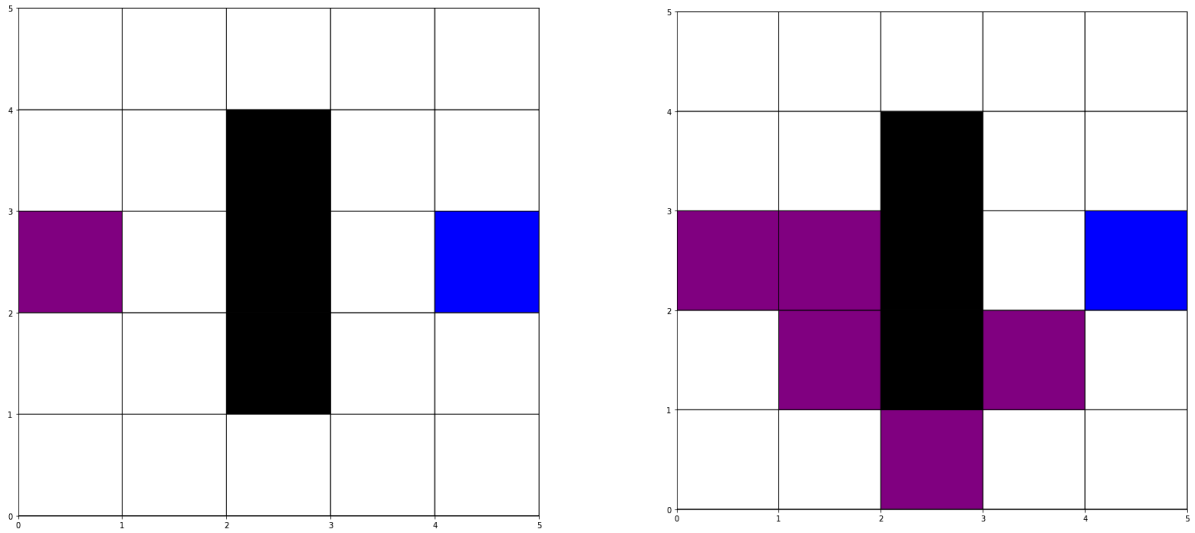
`Utils.py` methods

`Utils.py` file contains the following helper methods:

- `readScenarioFromJSONFilePath()`: this method reads a scenario from a JSON file and returns its elements as expected by the `Automata` class. It takes the JSON file path as an argument and returns all the elements of it, i.e., width, height, pedestrians, targets etc.
- `readScenarioFromJSON()`: this method reads a scenario from a dictionary and returns its elements as expected by the `Automata` class.
- `visualize()`: this method takes the state matrix of the grid and visualizes it using `matplotlib.pyplot's pcolor()` method.

Task results

As a visual example of the implementation, next we are going to show the results for the scenario described in Figure 1. This scenario consist of a 25 cells grid (5x5) with one pedestrian at (0, 2), one target at (4, 2) and also 3 obstacles at (2, 1), (2, 2) and (2, 3). As can be seen in the Notebook, we first visualize the initial state of the scenario (Figure 2a) and the final state with the followed path of the pedestrian (Figure 2b). We have that the pedestrian has purple color, the target blue color and the obstacles are black.



(a) Initial state

(b) Final state with paths

Figure 2: Initial state and final state with paths of the automata for the scenario from Figure 1.

To run this simulation the previously explained `basicOperator()` is used. We can appreciate also that we have taken into account the diagonal movements and that we are using the avoid obstacles option.

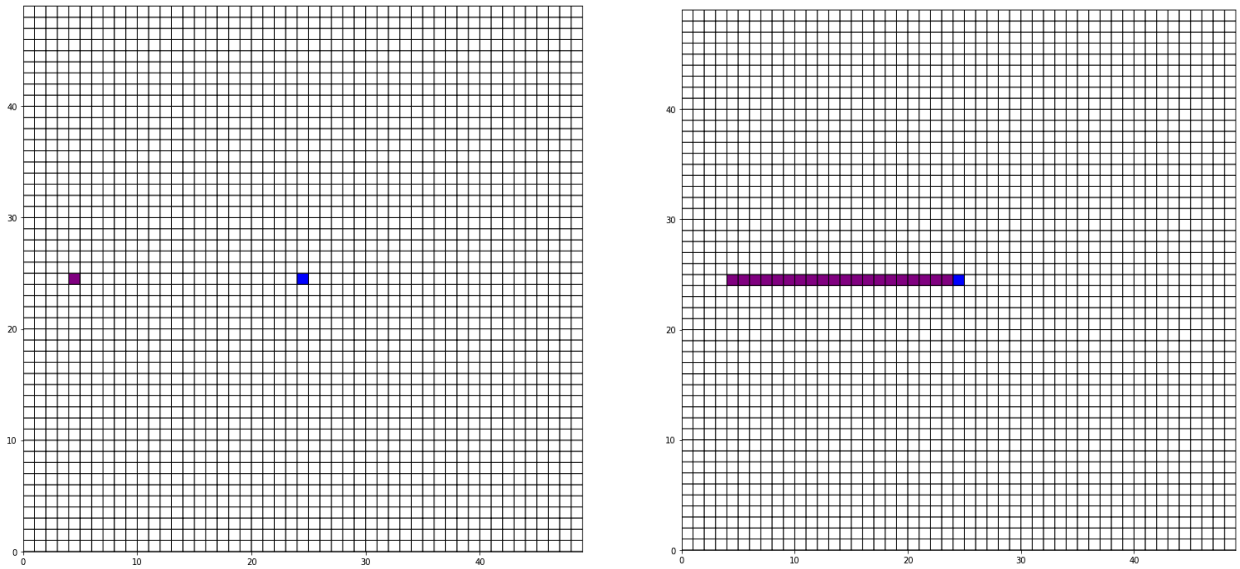
Report on task 2, First step of a single pedestrian

Task results

As stated in the task, we have a 50x50 cell grid structure. On this grid, we have one pedestrian at position (5, 25) and a target at (25, 25). We used for this simulation the same procedure as for the previous Task, but with the explained scenario described in its corresponding description of Figure 3.

```
configTask2 = {
    "name": "Task2Scenario",
    "dimensions": {
        "width": 49,
        "height": 49
    },
    "pedestrians": [
        [1, 4, 24]
    ],
    "targets": [
        [[1], 24, 24]
    ],
    "obstacles": []
}
```

Figure 3: Description of the scenario for the Task 2.



(a) Initial state

(b) Final state with path

Figure 4: Initial state and final state with path of the automata for the scenario from Figure 3.

As we can see on Figure 4, the pedestrian is 20 cells away from the target and indeed, we need 20 steps to reach the target as our simulation states.

Report on task 3, Interaction of pedestrians

Description and results

In this task, we define the pedestrian avoidance concept for our scenarios. Basically, this concept introduces that each pedestrian executes their movement, trying to stay away from other pedestrians. This rule can be given to an operator, i.e., passing to `basicOperator()` the argument `avoidPedestrians = True`.

Recall that `basicOperator()` method changes the grid state, by looking at the Euclidian distances of neighbors of each pedestrian to their respective targets, modifies the grid and saves paths for each pedestrian. For this task, we are using this operator function again but with the `avoidPedestrians` parameter setted to True.

As we discussed in previous tasks, operators iterate for each pedestrian, get every neighbor of every individual pedestrian, check if the target is in their neighborhood via `isTargetInTheNeighborhood()` method. If this is the case, the target is reached by this pedestrian, if not, procedure continues for this pedestrian.

Now we call the `getUnreachableCells()` method with `avoidPedestrians` argument. If that variable is `False`, this method only returns the obstacles list as unreachable cells. If pedestrian avoidance is enabled, obstacles and pedestrians are returned as unreachable cells.

As the Task 3 indicates, we created 50x50 grid and placed 5 equidistant pedestrians on the same circle, centered on (25, 25). This can be achieved with the following code depicted in Figure 5.

```
points = []
no_points = 5
for i in range(no_points):
    x = int(25.0 + 20 * math.cos(2 * math.pi * i / no_points))
    y = int(25.0 + 20 * math.sin(2 * math.pi * i / no_points))
    points.append((x,y))
```

Figure 5: Code for creating five equidistant pedestrians roughly on the same circle

The final pedestrian coordinates are (45, 25), (31, 44), (8, 36), (8, 13) and (31, 5).

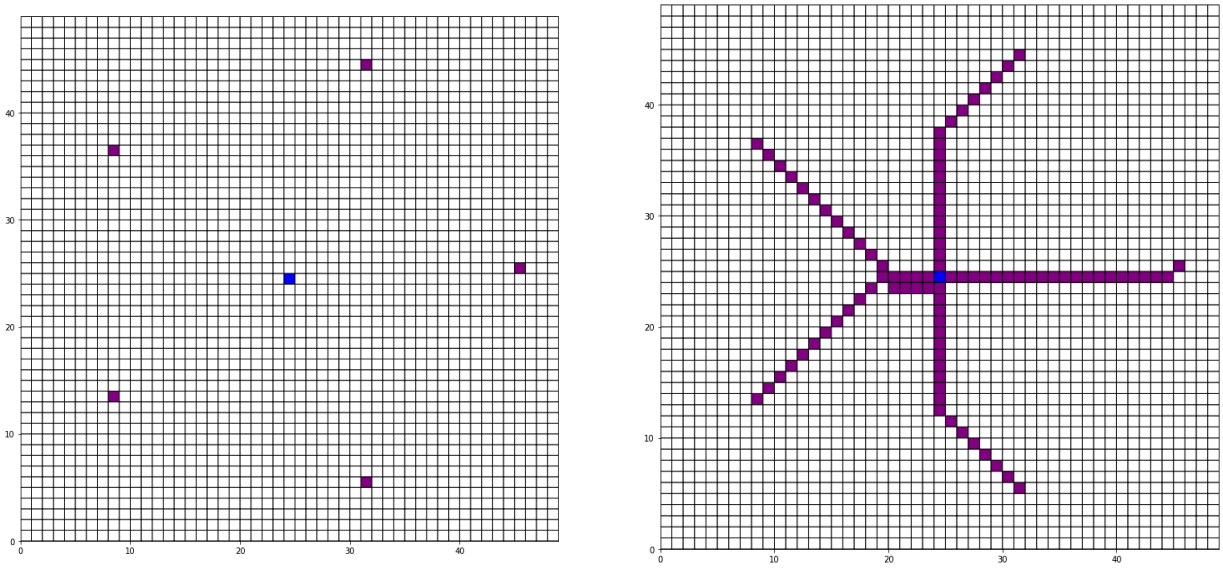
By creating these five points, we can then build the configuration dictionary and feed this as input to the code. The final description can be seen in Figure 6. Notice that the width and the heigh are 49 because we start counting from 0.

```
configTask3 = {
    "name": "Task3Scenario",
    "dimensions": {
        "width": 49,
        "height": 49
     },
    "pedestrians": [
        [1, 45, 25], [2, 31, 44], [3, 8, 36], [4, 8, 13], [5, 31, 5]
     ],
     "targets": [
        [[1,2,3,4,5], 24, 24]
     ],
     "obstacles": []
}
```

Figure 6: Structure for Task 3.

First, let us see what happens when we have **pedestrian avoidance**. We can do this by invoking `simulate()`. We simulate for 25 steps in order to see the complete results of the simulation. In spite of this parameter is an arbitrary value, notice that the simulation requires at least 21 steps in order to reach its end state.

In Figure 6 we can see the initial and final state with paths for the simulation avoiding pedestrians.



(a) Initial state

(b) Final state with paths

Figure 7: Initial state and final state with paths of the automata for the scenario from Figure 6 (avoiding pedestrians).

After we have visualized the paths of the pedestrians, we can easily conclude that every pedestrian have reached the target. They took 21, 20, 16, 16 and 19 steps to reach the target. Looking at the figure, we can also easily see the two pedestrians approaching to the target from top-left-corner and bottom-left-corner, are evading each other to not intersect in the same cells.

For a sanity check we can invoke the same `simulate()` function, but now without avoiding pedestrians by setting the argument to `False`. See Figure 8 for the results. In this case, they have needed the same number of steps mentioned before to reach the target.

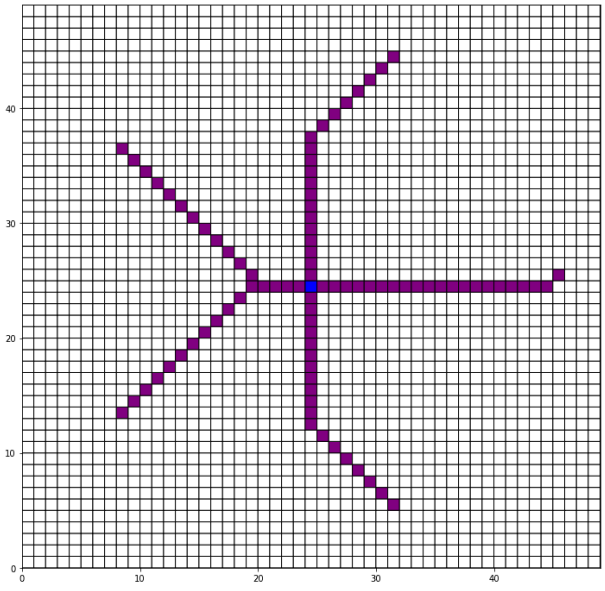


Figure 8: End state of the grid with paths for the scenario from Figure 6 file (without pedestrian avoidance).

Report on task 4, Obstacle avoidance

Up until now, we have worked with `basicOperator()` method, which acts only by depending the simple Euclidian distance metric for updating the state of the grid. In order to use a more complex operator and to use rudimentary obstacle avoidance for pedestrians by using a cost function, we are moving to more complicated operator method called `operatorWithCostFunction()` in `Automata.py` file. Interestingly, this method is very similar to the `basicOperator()` method with one single difference. From now on, distance metric control completely moved to the different function called `dijkstra()` which contains a variation of implementation of the Dijkstra algorithm.

`operatorWithCostFunction()` method

This method applies the "complex" operator to the current cellular automata state. The method uses a cost function, i.e. a grid flooded with distances by Dijkstra algorithm, to compute the next state of the cellular automata. It also allows for pedestrian and/or obstacle avoidance with parameters `avoidObstacles` and `avoidPedestrians`.

Method starts with a check: If every pedestrian has the same target, we can compute the distance matrix from `dijkstra()` method only once for all the pedestrians. We'll discuss the `dijkstra()` method following.

After that, just like we did in the `basicOperator()` method, for each pedestrian we compute its neighbors and its target. Get the minimum distanced reachable neighbor to the pedestrian's target and move the pedestrian to that cell. This outer loop is completely same as before. Lastly, we update the pedestrian's path and it's coordinates at the end.

`dijkstra()` method

`dijkstra()` method contains Dijkstra's algorithm for flooding the grid with the distances from all cells to the target cell. We pass the coordinates of the target cell as an argument. First we initialize the `distances` grid of dimension `(height, width)` with very large values (Python's infinity) in it and initialize its `target` coordinate with 0. In order to keep track the visited cells on the grid we have defined a set called `visited`. In an infinite loop, we define `currentCell` and `currentMinDist` variables and assigned them to `None` and `inf` respectively. Then, for every cell in the `distances` grid, if that cell is not in `visited` cells and its distance is less than `currentMinDist`, we keep assigning the current cell to that cell and `currentMinDist` to the distance of that cell in the `distances` matrix.

Termination condition for this loop is `currentCell`. If this is `None`, infinite loop ends. If not, we add this cell to the `visited` set.

Last part of the method is for every neighbor of this `currentCell`, if the neighbor is not in `getUnreachableCells()` and `avoidObstacles` condition is `False`; then we define the `newDistance` parameter by adding the `currentCell`'s distance with the `euclideanDistance(neighbor, currentCell)`. If this `newDistance` is smaller than the distance of the neighbor, we update the neighbor's distance with this `newDistance`. We have decided to add the Euclidean distance because we need some break for the directions different from the direct ones, e.g., the diagonals.

At the end we return the `distances` matrix. By this method we add high cost value to the obstacle cells. And also, we have implemented the rudimentary obstacle avoidance with this method.

Next, let us check what happens when we set/unset the obstacles avoidance option for two different test.

Bottleneck Scenario from RiMEA

For the first scenario, we tested the bottleneck figure as depicted in RiMEA, Guideline for Microscopic Analysis, p. 47, fig. 10. This scenario consists of two rooms connected by a corridor with 1/10 of the height of the rooms and a exit at the end of the second room. Both rooms are 10x10 and the first one has 150 persons in the left side.

For this scenario, since we need to specify velocities, we have used a especial version of the previously explained complex operator that will be introduce for the RiMEA tests. With or without velocities, the results are similar, but we decided to use the operator working with seconds and velocities in order to be consisten with the experiment description.

In Figure 9 the configuration of the scenario can be seen. We have created a grid as explained in RiMEA, considering some equivalences between cells and meters, and we have also created 150 pedestrians with a distribution of velocities in accordance with the experiment.

```
pedestrians = []
id = 1

# Create pedestrians
for i in range (1, 10):
    for j in range(2, 19):
        pedestrians.append([id, i, j, 0])
        id = id + 1
        if id == 151:
            break

# Distribute velocities
bunchSize = 25 # 150 pedestrians / 6 generations (20, 30, 40, 50, 60, 70)
for i in range(0, 150):
    if i < bunchSize: # 70s
        pedestrians[i][3] = random.uniform(1.07, 1.07)
    elif i < bunchSize*2: # 60s
        pedestrians[i][3] = random.uniform(1.27, 1.27)
    elif i < bunchSize*3: # 50s
        pedestrians[i][3] = random.uniform(1.39, 1.43)
    elif i < bunchSize*4: # 40s
        pedestrians[i][3] = random.uniform(1.46, 1.5)
    elif i < bunchSize*5: # 30s
        pedestrians[i][3] = random.uniform(1.52, 1.56)
    else: # 20s
        pedestrians[i][3] = random.uniform(1.6, 1.64)

configTask41 = {
    "name": "Task41Scenario",
    "dimensions": {
        "width": 71,
        "height": 22
    },
    "pedestrians": pedestrians,
    "targets": [
        [[i for i in range(1, 150)], 65, 10],
    ],
    "obstacles": [[20, i] for i in range(0, 22) if not i == 10 and not i == 11] + \
    [[i, 9] for i in range(21, 30)] + \
    [[i, 12] for i in range(21, 30)] + \
    [[30, i] for i in range(0, 22) if not i == 10 and not i == 11] + \
    [[51, i] for i in range(0, 22) if not i == 10 and not i == 11],
    "step": 0.5
}
```

Figure 9: Configuration for the Bottleneck Scenario from RiMEA

In Figure 10 we can appreciate the initial state and in Figure 11 the state after 15 seconds **without obstacle avoidance**.

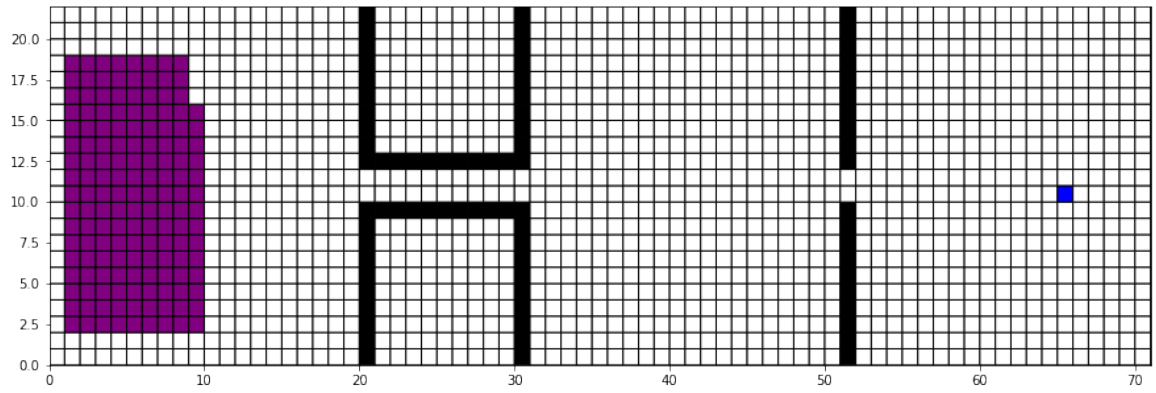


Figure 10: Bottleneck scenario initial grid state

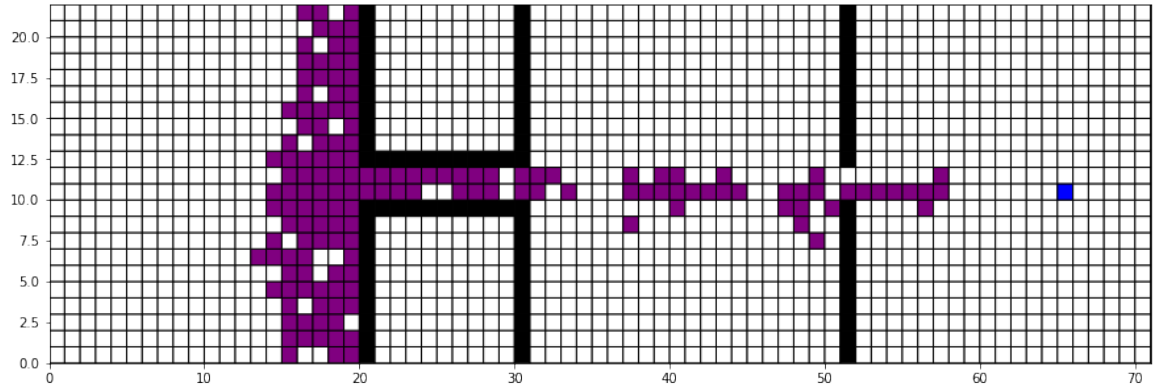
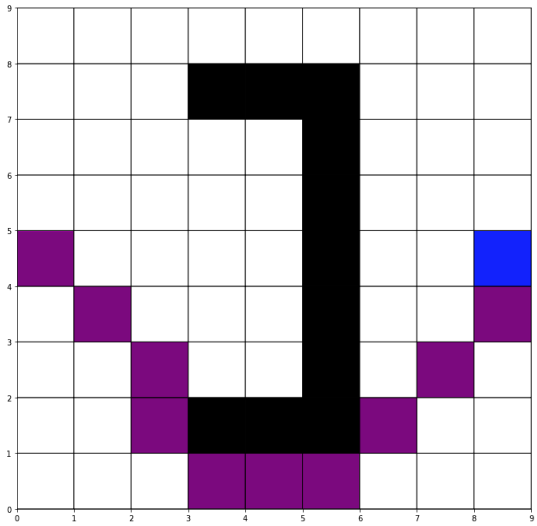


Figure 11: Bottleneck scenario after 15 simulation seconds **without obstacle avoidance**

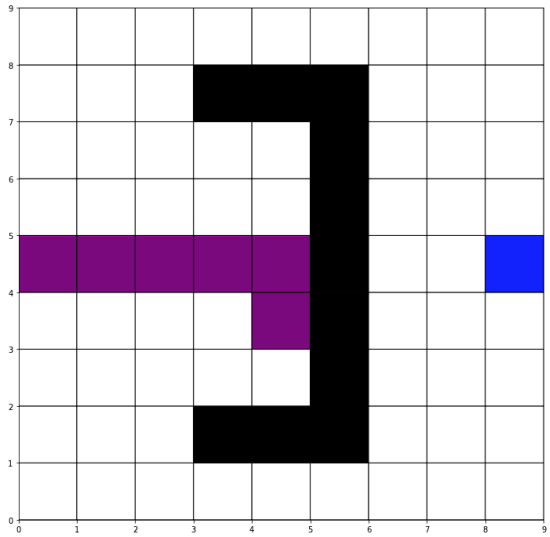
We can see that, without obstacle avoidance, all pedestrians try to cross the corridor causing a bottleneck at the entrance. In contrast with the same scenario, but with obstacle avoidance, when we don't have this option enabled the pedestrians do not follow a perfect line from the exit of the corridor to the target. This is cause due to the computation of the distance matrix, which does not take into account the obstacles and the pedestrians notice them when they are near and cannot continue.

"Chicken test" Scenario

We have implemented the chicken test scenario that was depicted in Figure 5 in the Exercise sheet. This consist of a square scenario with a U rotated 45 degrees to the left between a pedestrian and a target. For this simulation we have use the complex operator introduced at the begining of this task, without any special time or velocity addition.



(a) Final path after 20 steps for the chicken test **with obstacle avoidance**



(b) Final state with paths

Figure 12: Final path after 20 steps for the chicken test **without obstacle avoidance**

After simulating for 20 steps with and without obstacle avoidance we obtain the final paths of the Figure 12. As stated in the Exercise sheet, we can see that **without** obstacle avoidance feature, pedestrians stuck in front of the obstacle. If we enable the obstacle avoidance, we can see that the pedestrians successfully reach their target.

Report on task 5, Tests

Special version of methods: `operatorWithCostFunctionRiMEA()` and `simulateWithTime()`

Both `operatorWithCostFunctionRiMEA()` and `simulateWithTime` methods are similar to the ones previously introduced, but these ones incorporate the concept of velocity and time, required for this task.

In order to work with time, `simulateWithTime` (and the with-motion version `simulateAndVisualizeWithTime`) iterates over seconds instead of steps. For every second it updates, using an auxiliary function `updateAvailableSteps()`, the available steps of each pedestrian, i.e., the steps (meters) to move forward. Then it checks if someone can move, i.e., if someone has available steps, using the auxiliary function `someoneCanMove()`. If someone has available steps, then it applies the operator `operatorWithCostFunctionRiMEA()`. After applying the operator, it updates the running time (save on a `times` dictionary) for each pedestrian, using the auxiliary function `tikTak()`.

The operator `operatorWithCostFunctionRiMEA()` works the same way as `operatorWithCostFunction()`. The novelty is that it checks if the pedestrians have available steps (using auxiliary function `hasStepsAvailable()`) to proceed with its movement or not, it applies an absorbent behavior on the target (to avoid conflicts with other pedestrians trying to access the boundaries of the target).

These methods are supported by the data structures `pedestrainsSpeed`, `availableSteps` and `times`, created by the initialization method only if a speed for the pedestrians was specified in the 4th position array of each one. For the correct performance, we need to specify also the step size in the configuration as a new element of the dictionary/JSON.

TEST 1 - RiMEA Scenario 1 (Straight line)

For this first RiMEA test we have to define a straight line scenario with a pedestrian at the beginning and a target at the end. Establishing that 1 meter corresponds to 2 cells, we have that the 2x40 (m) corridor corresponds to 4x80 (cells) in our scenario. For a pedestrian speed of 1.33 m/s and a step size of 0.5, we should be able, as RiMEA states, to traverse the corridor in 26 up to 34 seconds.

Our simulation for 50 seconds finish after 31 seconds because the pedestrian achieved the target early. This is inside the aforementioned boundaries of traversing time, then we can conclude that our implementation is correct and the simulation is well done.

To recover the visualization, we suggest to execute the notebook cell corresponding to the task.

TEST 2 - RiMEA Scenario 4 (Fundamental diagram)

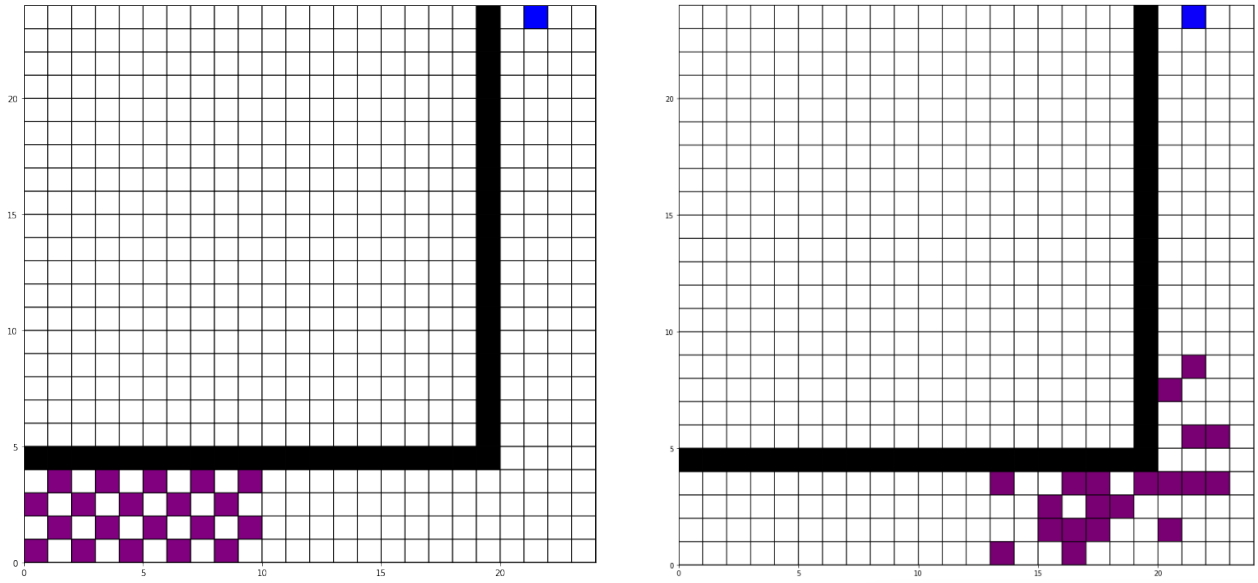
Due to a lack of time we have not been able to finish this test and produce some results to show here.

TEST 3 - RiMEA Scenario 6 (Movement around a corner)

For this test we have a corner, which turns to the left, scenario with 20 equally distributed pedestrians and same velocities. We should have that the pedestrians will successfully go around it without passing through walls.

Simulating for 80 seconds the pedestrians achieve the target correctly in 48 seconds, which seems correct since we set 0.5 m/s as a velocity and each human person takes 0.5. This taking into account that more than one pedestrian cannot occupy the same cell, since we have a relation of one cell, one pedestrian and pedestrian avoidance.

In Figure 13 can be seen the initial state of the automata and the state while they are turning to the left. We can appreciate in our simulation how the pedestrians near to the corner turn it rapidly than the others. This is something expected from a human behavior.



(a) Initial state

(b) Pedestrians turning the corner

Figure 13: Initial state and pedestrians turning the corner for the Test 3

TEST 4 - RiMEA Scenario 7 (Demographic parameters)

For the Scenario 7 we created 50 pedestrians of different pedestrians and we placed them in a random position (uniformly distributed) on the grid of 24x20 meters. The speed of the pedestrians varies depending on their age. The 50 pedestrians will walk with speeds compatible with Weidmann, as stated in RiMEA. In Figure 15 it can be seen the piece of code for assigning the distribution of velocities over the different pedestrians.

Our goal in this scenario is to show that the distribution of walking speeds in the simulation is consistent with the distribution in the table stated in RiMEA.

In the beginning, we placed 50 pedestrians randomly on the grid using the `np.random.randint` function from the Numpy library. We proceeded by adding IDs to each individual pedestrian. Finally, we need to distribute uniformly the speeds according to Figure 2 from Weidmann. For this purpose, we choose 9 pedestrians out of each age group (except from the 70 year old group that we has 5 instead of 9) to assign them a distance according to their age.

We are going to measure the walking speed of each pedestrian as the total distance travelled divided by the total time that they were travelling.

In Figure 14 we can observe the initial position of the pedestrians. At this point, the pedestrians are shuffled according to their age.

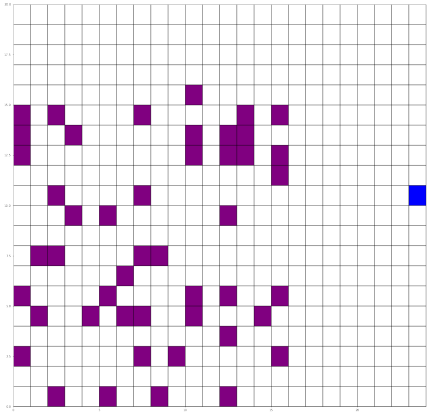


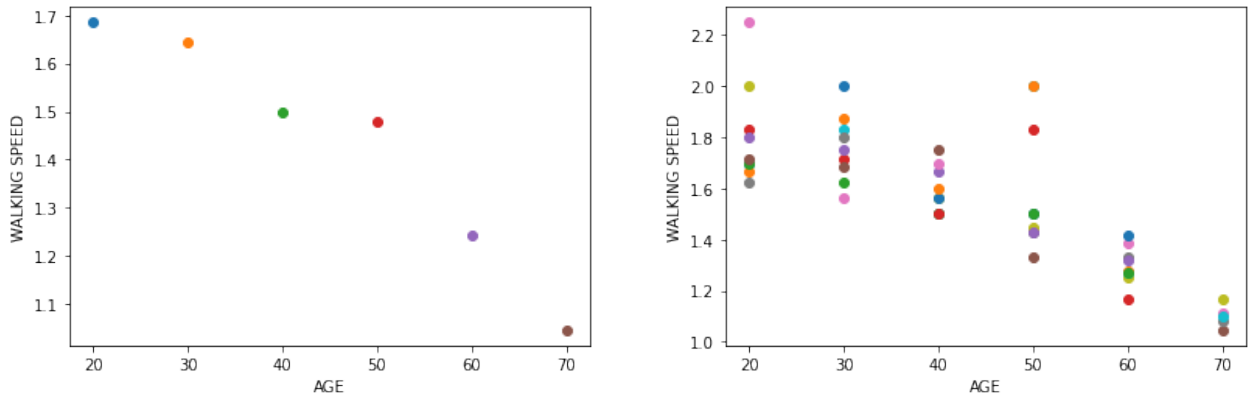
Figure 14: Initial position of the pedestrians before the simulation starts

```
ped = []

twenties = []
thirties = []
forties = []
fifties = []
sixties = []
seventies = []

# randomly place 50 pedestrians on the grid
ped = np.random.randint(16, size=(50, 2)).tolist()

# add Ids to the pedestrians
for i, p in enumerate(ped):
    p[0] = p[0]
    p[1] = p[1]
    p.insert(0, i+1)

# Finally, we add the speed according to their age
for i, p in enumerate(ped):
    if ((i+1) in list(range(1,10))):
        # 20 years old
        p.insert(4, random.uniform(1.6, 1.64))
        twenties.append(p[0])
    if ((i+1) in list(range(10,19))):
        # 30 years old
        p.insert(4, random.uniform(1.52, 1.56))
        thirties.append(p[0])
    if ((i+1) in list(range(19,28))):
        # 40 years old
        p.insert(4, random.uniform(1.46, 1.5))
        forties.append(p[0])
    if ((i+1) in list(range(28,37))):
        # 50 years old
        p.insert(4, random.uniform(1.39, 1.43))
        fifties.append(p[0])
    if ((i+1) in list(range(37,46))):
        # 60 years old
        p.insert(4, random.uniform(1.27, 1.27))
        sixties.append(p[0])
    if ((i+1) in list(range(46,51))):
        # 70 years old
        p.insert(4, random.uniform(1.07, 1.07))
        seventies.append(p[0])
```

Figure 15: Code for the distribution of the velocities over the pedestrians form the Test 4.

We ran the simulator and after 14 seconds all the pedestrian had reached the main target which is the blue cell in Figure 14. Lets now proceed to the calculation of their walking speed after the simulation. To display the results we used the matplotlib library.

The results regarding the average walking speed and the individual walking speed consist of two different plots (Figure 16). The first plot shows the average walking speed per age group. However, first we have to calculate the average walking speed of each age group and for this purpose we created the function `findAverageWalkingSpeed()` (available in the notebook) which gets as an argument an array with the IDs of all the pedestrians of an age group and calculates the walking speed by dividing the distanced covered by the time that they needed. On the other hand, the second plot of the figure bellow shows the individual walking speeds of each age group. In Figure 16b we can take a closer look and understand a bit more about the walking speeds of the pedestrians.



(a) Average Walking Speed of each age group

(b) Individual Walking speeds of each age group

Figure 16: Analysis of walking speeds based on different age groups

Observing Figure 16, we see that the average walking speeds are consistent with the distribution of the plot stated in RiMEA and left here for a quick review (Figure 17). Therefore, our experiment was successful and we showed that the distribution of walking speeds in the simulation is consistent with the distribution in the figure.

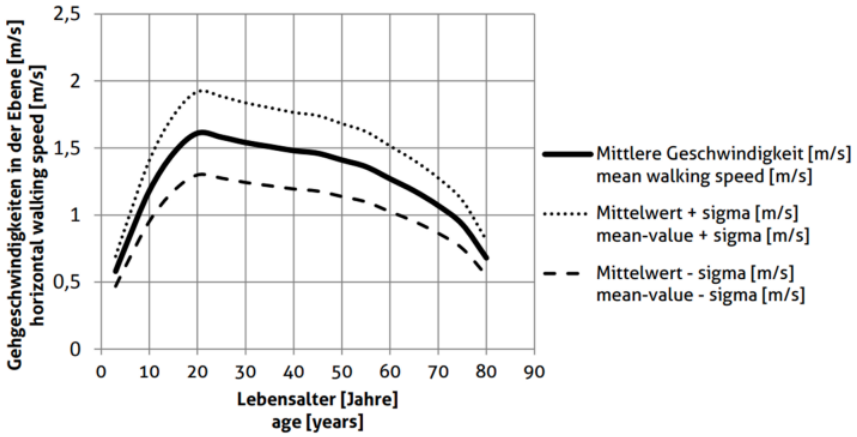


Figure 17: Figure 2 from Weidmann (RiMEA)

Literature