Report for exercise 1 from group H

Tasks addressed:	5
Authors:	Taiba Basit (03734212) Zeenat Farheen (03734213) Fabian Nhan (03687620)
Last compiled:	2021–04–28
Source code:	https://github.com/Combo1/MLCMS/tree/main/Exercise1

The work on tasks was divided in the following way:

Taiba Basit (03734212)	Task 1	33%
	Task 2	33%
	Task 3	33%
	Task 4	33%
	Task 5	33%
Zeenat Farheen (03734213)	Task 1	33%
	Task 2	33%
	Task 3	33%
	Task 4	33%
	Task 5	33%
Fabian Nhan (03687620)	Task 1	33%
	Task 2	33%
	Task 3	33%
	Task 4	33%
	Task 5	33%

1 INTRODUCTION

In our cellular automaton, each cell is in state X_i, where $X_i = 0, 1, 2, 3$. The symbols for a state as interpreted as:

```
0 : Empty Cell
1 : There is a pedestrian in this cell
2 : There is an obstacle in this cell
3 : There is a target in this cell
```

To build our cellular automaton, we came up with a modular approach of building 3 classes namely Cell, Board and Simulation.

1.1 Simulation of scenario

Method update_board() in class Board contains the main logic for simulation. The following considerations have been taken into account for the simulation:

1. This method works in case of multiple targets, multiple pedestrians and multiple obstacles.
2. We consider the Moore neighbours of a pedestrian cell.
3. To calculate the next move of the pedestrian cell, we calculate the Euclidean distance of the neighbours of pedestrian cells to the nearest target.
4. We also take into consideration the speeds of different pedestrian, where we consider a speed of 1m/s to be equivalent of moving the pedestrian by 1 cell in 1 time step.
5. In the presence of multiple targets, the pedestrian tries to move towards the target that is closest to it in terms of Euclidean distance.
6. Once the target is in the neighborhood of the pedestrian, it stops moving and waits there.

1.1.1 Algorithm

The following algorithm is implemented in the method update_board()

1. Let P be a pedestrian from the list of pedestrians
2. Fetch the nearest target for P. Let it be T
3. Fetch the list of valid Moore neighbours of P
4. For each valid neighbour N:
   (a) Check if N is in empty state i.e 0
   (b) If yes, then calculate the Euclidean distance N to T
   (c) If the distance is less than minimum dist, update minimum dist to new distance, and set N as the next move of P. Let the next move of P be P'
5. Move P to P'
   (a) Set state of P to empty or 0
   (b) Set state of P' to pedestrian or 1
6. Repeat steps 1-5 for all pedestrians till the nearest target of each pedestrian is in their neighborhood.

1.2 Calculating nearest target for pedestrian

Method set_nearest_target() in class Board performs the function of assigning nearest target to all pedestrians.

1.2.1 Possible Scenarios

The following scenarios are possible:

- Multiple pedestrians single target : In this case, all pedestrians move towards the same target.
- Single pedestrian multiple targets : In this case, the pedestrian moves towards the target that is closest to it in terms of Euclidean distance.
- Multiple pedestrians multiple targets : In this case, each pedestrian moves towards the target that is closest to it terms of Euclidean distance.

1.2.2 Algorithm

The function implements the following algorithm:

1. Let P be a pedestrian from a list of pedestrians
2. For each T in the list of targets:
   (a) Calculate Euclidean distance of P from T
   (b) If this distance is less than minimum distance. Update the minimum distance. Set T to be the nearest target of P
3. Repeat steps 1-2 for all pedestrians

1.3 Finding out Moore neighbours

Method check_neighbour(check_row, check_column, speed=1) in class Board implements the logic for finding out the nearest Moore neighbours of the cell positioned at coordinates (check_row, check_column) and having a default speed of 1 m/s.

1.3.1 Possible Scenarios

The following scenarios are possible:

- The current cell lies on the border of the board : In this case, only the cells surrounding the current cell at depth 1 that lie inside the board are considered as valid neighbours.
- The current cell lies in the middle of the board : In this case all cells surrounding the current cell at depth 1 are considered as valid neighbours.

1.3.2 Algorithm

The following algorithm is implemented:

1. Let lower bound of search boundary be -speed and upper bound of search boundary be speed+1 of cell P
2. For each neighbouring cell N falling within this boundary, it is considered a valid neighbour if
   (a) N is not the current cell itself i.e. N != P
   (b) The row coordinate of N is not < 0 or $>=$ total number of rows in board
   (c) The column coordinate of N is not < 0 or $>=$ total number of columns in board
3. If a valid neighbour is a Target cell at depth 1 i.e state = 3, then return empty list
4. Else, repeat step 2 for all neighbours and append all valid neighbours to a list and return it.

2 TASK

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

1. **Basic Visualization**

   To visualize the cellular automaton grid structure, we have used the method _generate_board() in class Board. It sets up the data member _grid which is initialized with a N*N Cell instances , where N is the size of the grid input by the user. The method iterates over each cell and sets the state and position of the Cell according to the provided input_data read from input file.

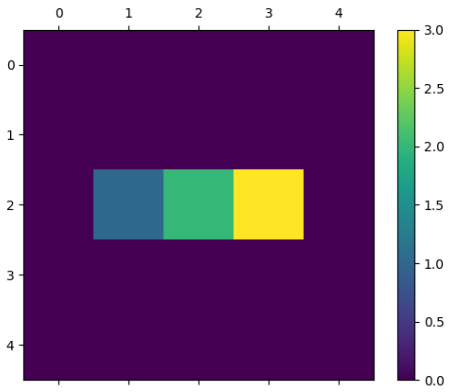

Figure 1: Basic Visualization

2. **Adding pedestrians in cells**

   Pedestrians can be added to the grid by placing '1' in the desired location in input text files. This is represented in the code with the state of Cell being set to 1. The pedestrians are represented by the color blue,as shown in figure 1.

3. **Adding targets in cells**

   Targets can be added to the grid by placing '3' in the desired location in input text files. This is represented in the code with the state of Cell being set to 3. The targets are represented by the color yellow, As shown in figure 1.

4. **Adding obstacles by making certain cells inaccessible**

   Obstacles can be added to the grid by placing '2' in the desired location in input text files. This is represented in the code with the state of Cell being set to 2.The obstacles are represented by the color green, As shown in figure 1.

5. **Simulation of the scenario (being able to move the pedestrians)**

   To simulate the movement of pedestrian towards target, we give an input scenario text file with comma separated values of 0,1,2,3 representing the state of a cell. The input represents a square grid of size N*N. Simulation is implemented using Matplotlib's animation and pyplot. We have used FuncAnimation method of class animation, to simulate the scenario. Method update_board() is invoked to generate the simulated state of the grid as shown in figure 2.

```
fig, ax = plt.subplots()
img = ax.matshow(input_file)
plt.colorbar(img)
ani = animation.FuncAnimation(fig,update,data_gen, interval=800,
                              save_count=50)
plt.show()
HTML(ani.to_html5_video())
```

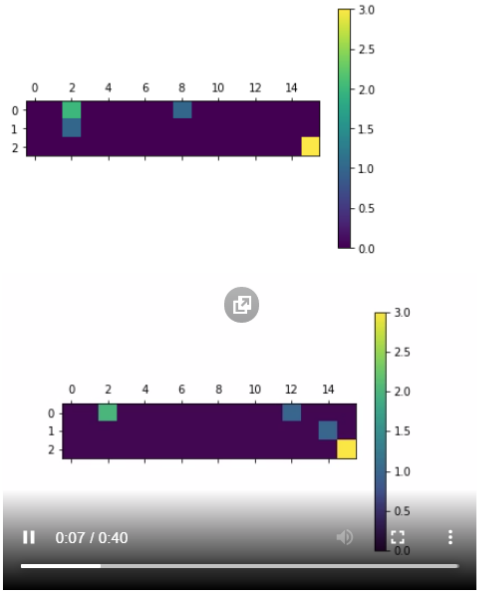

Figure 2: Simulation of the scenario

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

We provide a input text file with the scenario of 50 by 50 cells (2500 in total), a single pedestrian at position (5, 25) and a target 20 cells away from them at (25, 25).

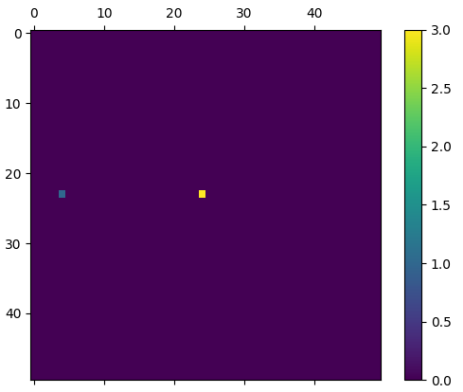

Figure 3: Initial configuration of pedestrian

Figure 3 shows the initial state of the grid. After running the simulation, the final state of the grid is displayed in figure 4 We observe that the pedestrian has moved from it's position to the target in 20 time steps and waits there as expected. The core method used to implement this test case is Board.update_board(), which is described above in Simulation of Scenario.

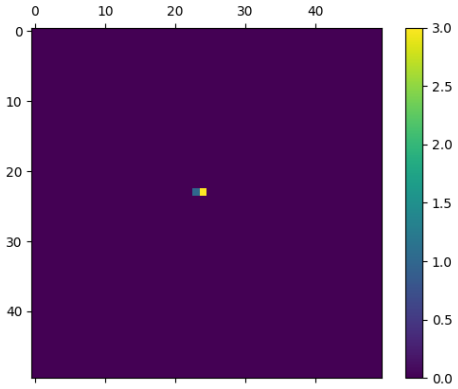

Figure 4: Final configuration of pedestrian

Report on task 3/5, Interaction of pedestrians

The input file for this task contains a 70 * 70 matrix. Here we consider that one cell has the dimensions of 1x1 m. The pedestrians are denoted by 1 and the target is denoted by 3. The location of the target is (35,35). The pedestrians are located at (10,17),(35,6),(60,23),(55,59),(9,53).

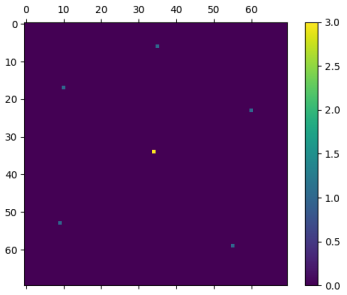

Figure 5: Initial configuration of pedestrian

The configuration of the pedestrian around the target can be seen in the figure below:

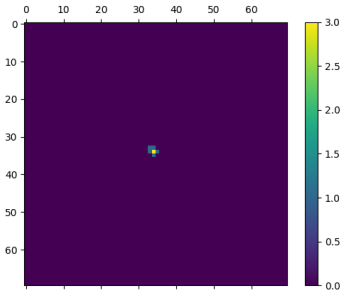

Figure 6: Final configuration of pedestrian

Result : All the pedestrians reach the target at roughly the same time because they are travelling at the same speed and are at the approximately equal distance from the target.The detailed algorithm is explained in section 1.

Report on task 4/5, Obstacle avoidance

For the Dijkstra algorithm we imagined the board to be the graph, the cells to be the nodes and neighboring cells (vertical, horizontal and diagonal adjacent cells) to be edges. Here is the initial configuration of board:

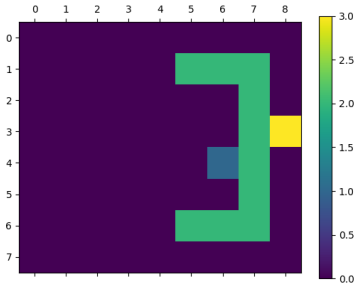

Figure 7: Initial scenario

The algorithm works as follows:

1. For each node we save a distance and a visited value. The distance value is initialized with infinity for every node but the target cell. The visited value is initialized with False for every node. Then we start the algorithm from the target cell.
2. As long as there are unvisited nodes, choose the one with the lowest distance, which was not visited and is not an obstacle.
3. Set the visited value to True and for each neighbor of this node check if their distance value is higher than the current node's distance value + 1. If that is the case then replace their distance value with the current node's distance value + 1.
4. Repeat steps 2-3

We end up with a matrix as shown in figure 8, which contains the amount of steps needed from the every cell in the grid to the target cell.

```
b.dijkstra()
```

```
[[10, 9, 8, 7, 6, 5, 4, 3, 3],
 [10, 9, 8, 7, 6, inf, inf, inf, 2],
 [10, 9, 8, 7, 7, 7, 8, inf, 1],
 [10, 9, 8, 8, 8, 8, 8, inf, 0],
 [10, 9, 9, 9, 9, 9, 9, inf, 1],
 [10, 10, 9, 8, 8, 8, 9, inf, 2],
 [11, 10, 9, 8, 7, inf, inf, inf, 3],
 [11, 10, 9, 8, 7, 6, 5, 4, 4]]
```

Figure 8: Dijkstra Distance Matrix

To implement the movement of the pedestrians we iterated over all the pedestrians on the board and searched if there is a neighboring cell, whose distance value is lower than the distance value of the current cell and is neither pedestrian, obstacle or target. If there is such a cell update the position of the pedestrian and move on the next pedestrian in the list, until we reach a certain amount of steps. The code snippets can be found inside in class Board in methods dijkstra(self) and update_board_dijkstra(self).

To visualize the implementation we used matplotlib's animation and embedded the animation inside of the jupyter notebook. The final results after the simulation is shown in figure 9

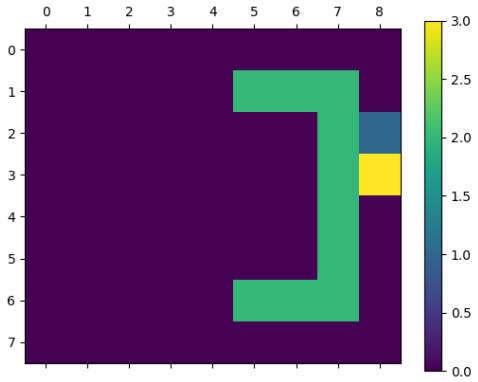

Figure 9: Final position after simulation

Report on task 5/5, Tests

RiMEA scenario 1 (straight line, ignore pre movement time):
The test case is: It is to be proven that a person in a 2 m wide and 40m long corridor with a defined walking speed will cover the distance in the corresponding time period.

For this task we created a board of 50 * 50, with a corridor in the center of width 5m. We have taken a width of 5m instead of 2m for the pedestrians to visually identifiable and a length of 50m instead of 40m. We have initialized 3 pedestrians at (22,0),(24,0) and (26,0) with speed or 1 cell/timestep, 2 cell/timestep and 3 cell/timestep. The information for speeds for different pedestrians is to be specified in a separate text file containing the (row,column,speed) for each pedestrian.It also contains 5 targets at (22,49), (23,49),(24,49),(25,49) and (26,49).

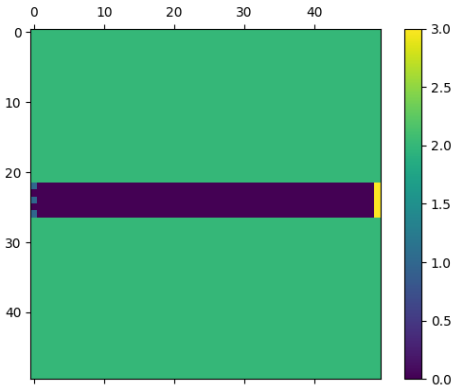

Figure 10: Initial scenario

On running the simulation, it can be easily seen that the person is covering the distance based on their walking speed. The image below shows this positions of pedestrian during the simulation:

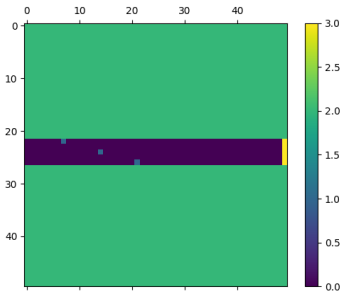

Figure 11: During simulation

As we run our simulation, we can see that all the pedestrians move at their defined speed and reach the target in the corresponding time period.

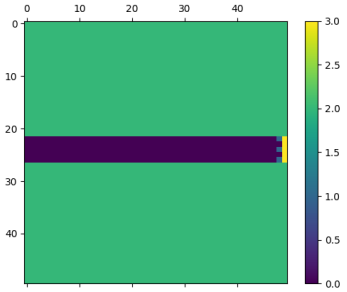

Figure 12: Final position after simulation

RiMEA scenario 4 (fundamental diagram, be careful with periodic boundary conditions): For this scenario we defined a 500/100m cells long corridor (1 cell = 0.2m) with a height of 50/10m cells. For the different scenarios we used 0.4P/m2, 1P/m2, 3P/m2. We stopped the simulation at a certain point since the computation on the notebook was to heavy. We did not create any measuring points, but it appears that groups build according to their speed level. Due to the nature how we implemented the dijkstra algorithm the pedestrian move diagonally upwards, which should not be an issue since diagonal moves are as expensive as horizontal/vertical movements and they still end up at the target at the optimal amount of steps. However as we implemented this test case the pedestrians just seem to build a bar-shaped group without huge differences across different pedestrian density configurations.

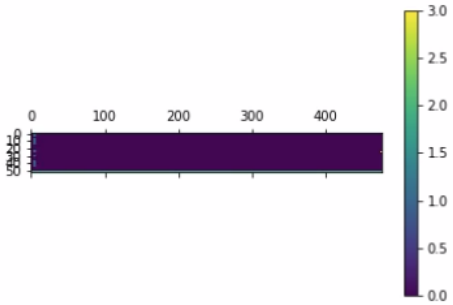

Figure 13: Initial position before simulation

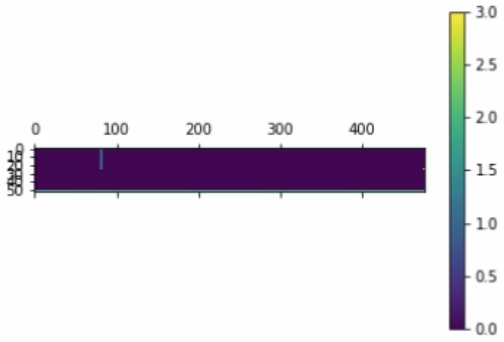

Figure 14: Running pedestrians

RiMEA scenario 6 (movement around a corner):
The test case states that: Twenty persons moving towards a corner which turns to the left will successfully go around it without passing through walls.

For this scenario we created a board of size 12 *12. The corner is present at the bottom right of the board. We created 9 pedestrians instead of 20 because the number of pedestrian does not affect the test case and also based on the board size the number of pedestrian is proportional to it.

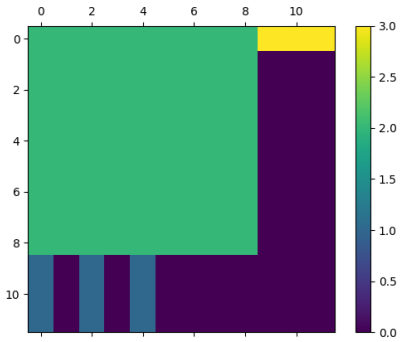

Figure 15: Initial position before simulation

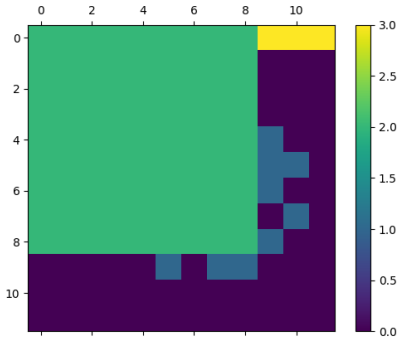

Figure 16: Pedestrians turning around the corner

After running the simulation, it can be clearly seen that the pedestrians were easily able to take turn at the corner without passing through the walls (figure 16) and were reach the target as shown in figure 17.

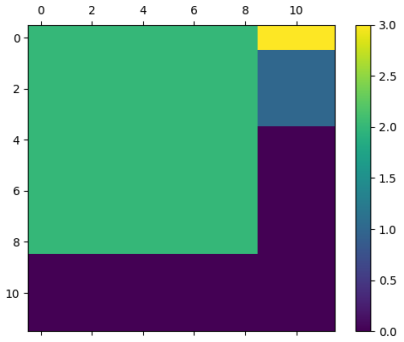

Figure 17: Final position after simulation

RiMEA scenario 7 (demographic parameters, visual comparison of figure and results is sufficient): For the RiMEA scenario 7 we used a 60 * 60 grid (1 cell = 0.2m). Each pedestrian can walk between 0.6m/s-1.6m/s, we simulated that by making each pedestrian able to move between 3-8 cells in one time step. We generated the data by randomly generating a position and for the speed distribution we artificially tailored the speed such that we get a similar result to the curve in Figure 2 of RiMEA. In the simulation you can see that very early on that faster pedestrians move towards the top of the pack quickly since the distance to cover is not that big. Each speed group is also developing their own movement group, which could be avoided by making an even bigger field where the speed differences are smaller between the groups. The speed of the pedestrians can be found down below, since we have not actual probands there is no age axis, but you can see that the speed approximately corresponds to the distribution of the RiMEA scenario.

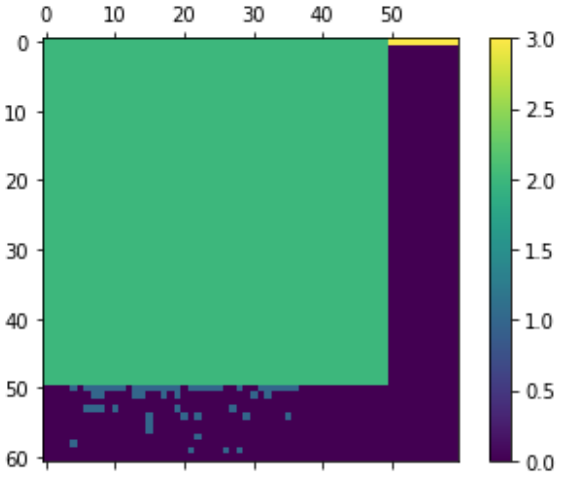

Figure 18: Initial position before simulation

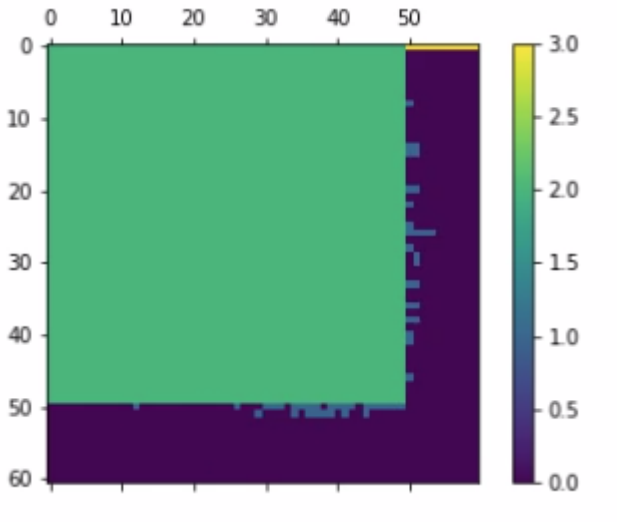

Figure 19: Running Pedestrians

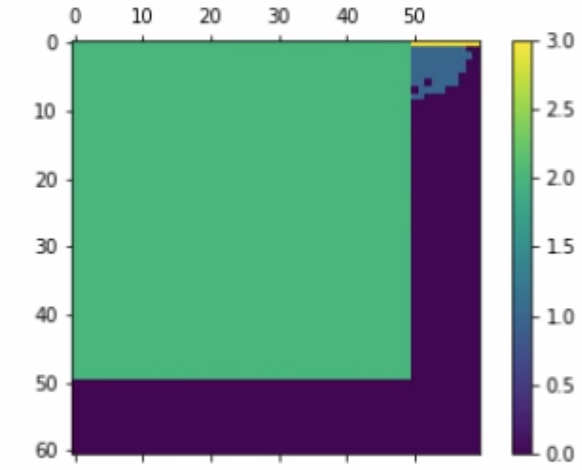

Figure 20: Final position after simulation

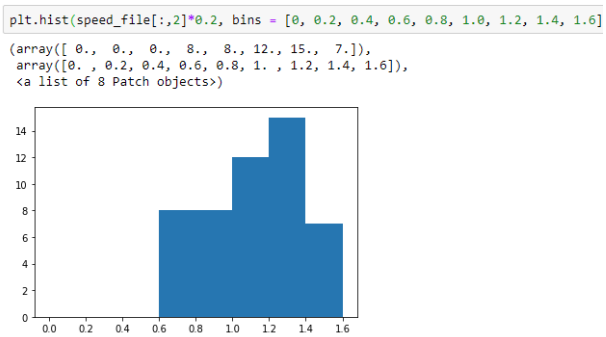

Figure 21: Pedestrian speed