Report for exercise 1 from group H

Tasks addressed:	5
Authors:	Milena Schwarz () Antonia Gobillard () Muhammed Yusuf Mermer () Nayeon Ahn () Hammad Basit ()
Last compiled:	2024–05–18
Source code:	https://github.com/Hammad-7/MLCMS_Exercises/tree/main/Exercise01

The work on tasks was divided in the following way:

Milena Schwarz () **Project lead**	Task 1	20%
	Task 2	20%
	Task 3	20%
	Task 4	20%
	Task 5	20%
Antonia Gobillard ()	Task 1	20%
	Task 2	20%
	Task 3	20%
	Task 4	20%
	Task 5	20%
Muhammed Yusuf Mermer ()	Task 1	20%
	Task 2	20%
	Task 3	20%
	Task 4	20%
	Task 5	20%
Nayeon Ahn ()	Task 1	20%
	Task 2	20%
	Task 3	20%
	Task 4	20%
	Task 5	20%
Hammad Basit ()	Task 1	20%
	Task 2	20%
	Task 3	20%
	Task 4	20%
	Task 5	20%

Report on task 1, Setting up the modeling environment

In the first task, the main goal is setting up the modeling environment which the following tasks depend on. We have undertaken a revision and enhancement of the 'ExampleGUI' program for better functionality and User Experience. We used the 'Place' method to design the window in `tkinter` [4]. The place method allows you to position a widget within a specified frame by directly entering x and y coordinate values. We used this method because it is useful to precisely position a widget and we will constantly change the size of the grid(widgets). (x= x-coordinate, y= y-coordinate) to set the position.

Here is a breakdown of the features and functionalities that are included:

1. Step Simulation Button: This feature allows users to incrementally advance through the simulation timeline. By pressing this button, the user can observe the progression of the simulation step by step.
2. Restart Simulation Button: Should there be a need to revert to the initial state of the simulation, users can utilize this button. Upon pressing, it resets the entire simulation to its default state.
3. Run Simulation Button: With the help of this feature, the user can watch the entire simulation without the need to repeatedly advance through the timeline themselves.
4. Create Scenario Button: Activating this button spawns an additional window, as depicted in the provided illustration. Within this window, users have the capability to:
   - Introduce pedestrian elements.
   - Incorporate obstacles.
   - Designate destinations.
5. Load Scenario Button: For users who have predefined simulation settings saved as JSON files, this button facilitates the importing of those scenarios into the current session.
6. Save Scenario Button: Post-simulation or during its configuration, users might want to preserve the current scenario for future reference or use. This button empowers users to save the present simulation scenario as a JSON file.

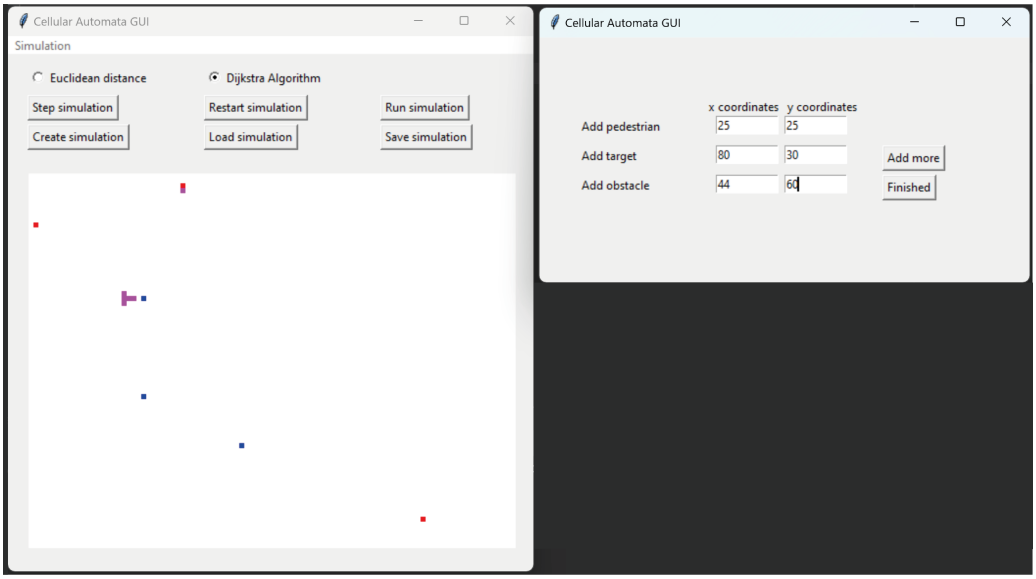
The `restart_scenario` function corresponding to the 'Restart Simulation' Button simply loads the file of the current running scenario again.
Within the function `run_simulation` the already existing function `step_scenario` was used. Between each function call, the program waits 0.05 seconds due to the usage of `time.sleep` [3].

In order for the user to intuitively create a new scenario, multiple options can be presented. We decided that this functionality should be a part of the Graphical User Interface (GUI). A consideration is to let the user draw pedestrians, targets, and obstacles with the mouse. The downside to this idea is that, especially with large grids, the input would be imprecise. Hence, we created a new window in which the user can write explicit integers.
The function `popupwin` creates and shows a simple user interface with multiple labels, entries, and buttons. The user can set a new scenario size, as well as add pedestrians, targets, and obstacles. All of this can happen due to the `add_more` function. The function adds one pedestrian, one target, and one obstacle at maximum. After clicking the corresponding button, the newly added objects can be seen in the main window, and the entries have been cleared with the `clear_entries` function in order to make it possible to add more of each desired object. In figure 1a new objects locations are set to be added. After clicking the "add more" button, the new objects can be seen on the left side of figure 1b in the main window. It includes error handling for numbers too small or too big for the scenario size, in which case an error messagebox, provided by `tkinter`, appears to explain to the user which inputs are possible, as can be seen in figure 1b.

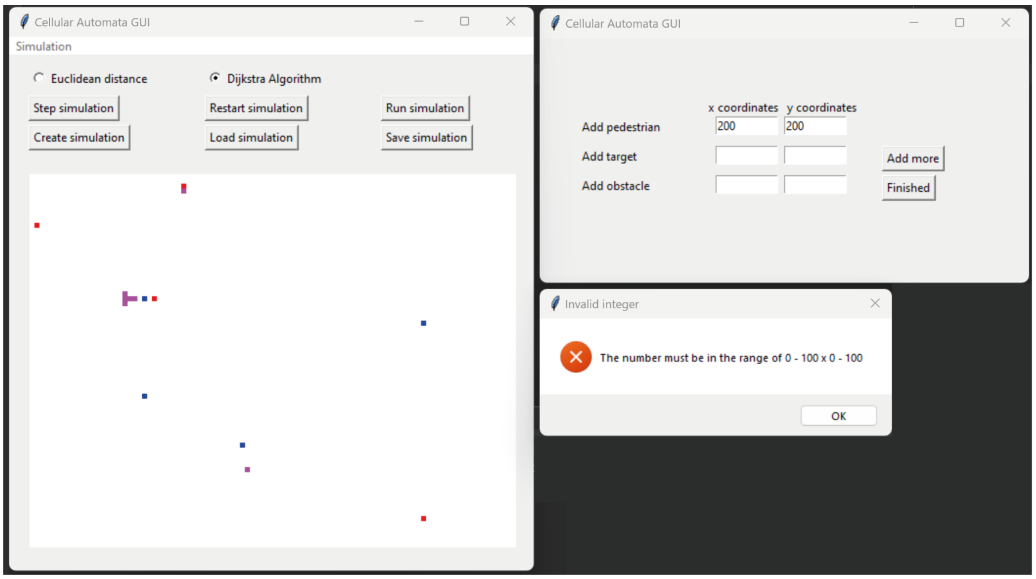
This choice of User Interface provides various advantages: the user can see changes fast and problems with entries can easily be related to the last input. Additionally, there is no need to continuously change the Interface, e.g. with a plus button, in order to fit more cells to add multiple objects of the same kind at once. After the creation of the new scenario, the finish button closes the additional window with the help of the `close_win`

function, and the user can save the scenario.

The `load_scenario` and `save_scenario` functions benefit from the `filedialog` of `tkinter`. This enables the user to select any JSON file on the computer as well as save new scenarios with the desired title and location.



(a) Create simulation GUI

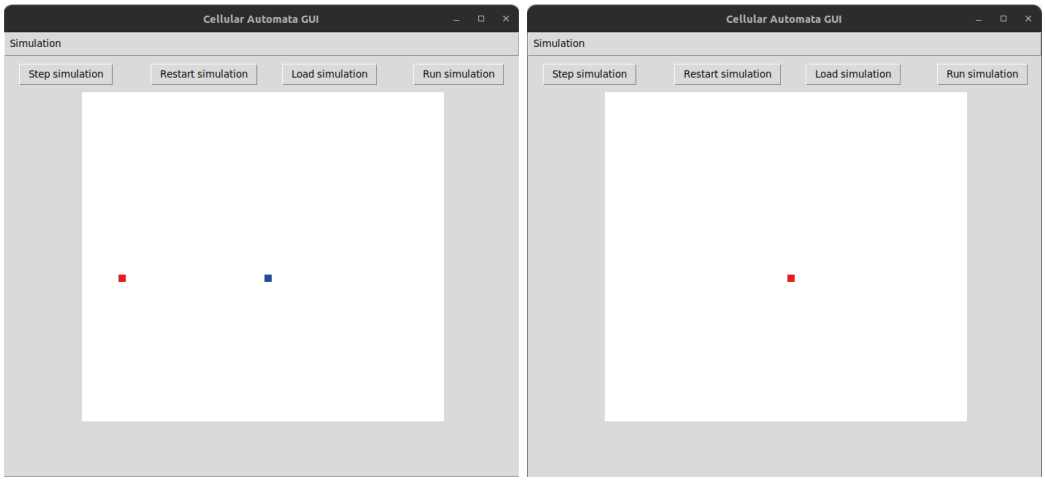


(b) Error Messagebox

Figure 1: The GUI for creating a new simulation includes entry fields and buttons to add pedestrians, targets, and obstacles to the current grid

Report on task 2, First step of a single pedestrian

For this task, we provide an input JSON file with a scenario consisting of 50 by 50 cells (2500 in total). The pedestrian is at the coordinate (5,25), whereas the target is 20 cells away at the coordinate (25,25).



(a) Initial configuration of the scenario 2 (b) Final configuration of the scenario 2

Figure 2a shows the initial configuration of scenario 2 on the grid. Figure 2b shows the final output after simulating the scenario for 25 time steps. The pedestrian reaches the target in 20 time steps and waits there for 5 seconds until the simulation is completed.

Report on task 3, Interaction of pedestrians

For this scenario, 5 pedestrians are now placed at roughly equal distances from the centered target. The input file for this task is the `scenario3.json`. We take a grid of 80 by 80 cells (6400 in total) and have the target centered at the coordinate (40,40). The pedestrians are placed roughly at 30 meters distances at the following coordinates from the target:

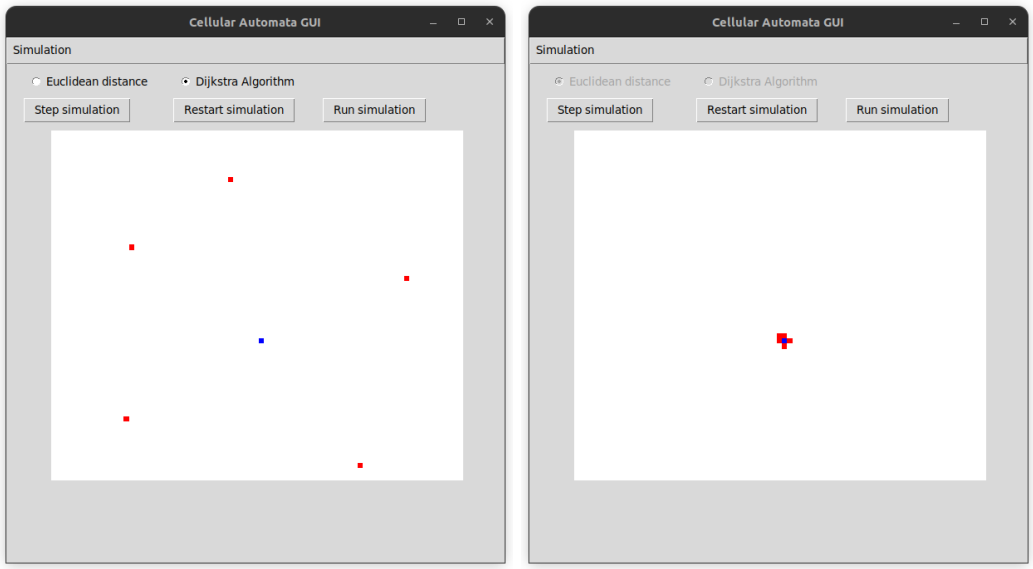
- (15, 22)
- (34, 9)
- (68, 28)
- (59, 64)
- (14, 55)

Figure 3a shows the initial setup of the grid with the pedestrians and the targets.

When employing solely the Euclidean distance as the metric, it was observed that elements moving diagonally reached the target more quickly compared to those moving along horizontal or vertical paths. This expedited arrival occurs due to pedestrians covering a greater distance (approximately $\sqrt{2}$ units) while traversing diagonally, as opposed to only 1 unit when moving vertically or horizontally.

To address this discrepancy, we introduced the concept of waiting time. Pedestrians are made to wait longer to simulate a real-life scenario by incorporating a waiting period equivalent to $\sqrt{2}$ seconds when they move diagonally. Conversely, a waiting duration of 1 second is imposed when they move in horizontal or vertical directions. This adjustment equalizes the arrival times of all pedestrians at the target, ensuring that they reach it almost simultaneously, regardless of their chosen path.

Result: Figure 3b shows the configuration of scenario 3 just before the pedestrians are absorbed by the target. All the pedestrians reach roughly at equal times as they are placed at roughly equal distances from the center.



(a) Initial configuration of scenario 3

(b) Final configuration of scenario 3 before being absorbed by target

Report on task 4, Obstacle avoidance

For previous parts, Euclidean distance was a very good method to implement distance for determining the motion of the pedestrians as there is no limitation such as obstacles or height on the direct path to influence to direction of the pedestrians. They just follow the shortest path to the target point without any angle problem. However in real life:

> "Nothing is as easy as it looks" - Murphy's Law

There are obstacles that we need to consider. For part 4, using simply Euclidean distance is not fine, as one cannot go through objects and needs another path. Let us first take a look into what will happen when we use rudimentary obstacle avoidance.

Rudimentary obstacle avoidance employed the calculation of Euclidean distance. In this method, pedestrians would navigate towards to the targets linearly. However, we advanced it so that pedestrians will not traverse through objects, but this does not mean that they will just stop without trying other ways. They can look at their surrounding environment and walk around the obstacle if the barrier is not that big.

To do this, we have improved the `update_euclidean_move` function within `scenario_elements.py`. Normally, at the next update, the Euclidean distance needs to be shorter than the current one. However, our current code only tries to look at the shortest distance among the neighbours, it does not compare them with the current one. Therefore, when pedestrians are stopped by an obstacle, even though their movement will increase their distance to the obstacle compared to the current point, pedestrians can pass to the shortest distanced neighbour of its current location, and if there is no other object in front of them, they can just beat around the object with this method. This algorithm is visualised in figure 4.

(a) When pedestrian encounters an object

(b) Even though it is longer in distance, the pedestrian tries to use its neighbours to pass the object

(c) Pedestrian passes object successfully

Figure 4: Rudimentary obstacle avoidance using Euclidean distance

Although it could be seen as a solution, it only works for small objects that pedestrians can skip with one step. Hence, if the object is bigger and cannot be passed via one movement, the pedestrians will turn back to their first location where they first encountered the object. Then, they will twist around left and right points without successfully reaching the targets. This can be seen in figure 5.

(a) A pedestrian encounters an object

(b) Even though it is longer in distance, the pedestrian tries to use its neighbours to pass the object.

(c) The pedestrian could not pass the object and starts to twist around left-right points and cannot reach any target

Figure 5: Rudimentary obstacle avoidance fails

If we use this implementation on the chicken test scenario in which there is a U-shaped obstacle between the pedestrian and the target, we will see similar results. The pedestrian, trapped inside this box-shaped obstacle, is unable to exit or proceed towards the target. It changes its place but nevertheless, it cannot get out of the box to reach the target point, as depicted in figure 6.

(a) Initial setup of the chicken test

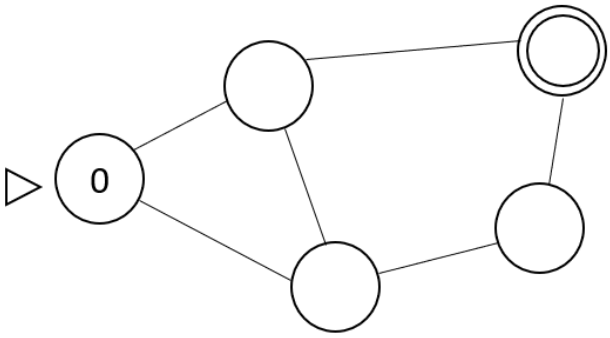
(b) The pedestrian makes contact with the obstacle

(c) The pedestrian slides to the right but still cannot pass through the object. Therefore, it will start to twist inside the object

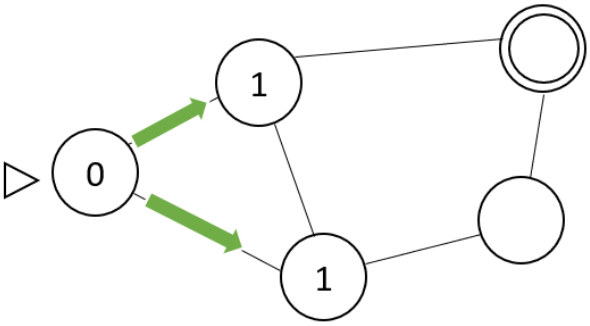
Figure 6: Chicken test fails for rudimentary obstacle avoidance

As we can indicate from these results, using rudimentary obstacle avoidance generally is not effective. Therefore, we needed to focus and explore much more effective algorithms.

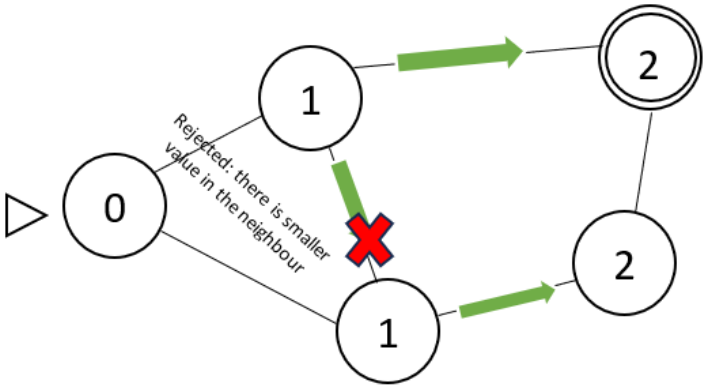
Dijkstra's algorithm is well-suited to our objectives. In the Dijkstra algorithm, we initially assigned the cost of target points as 0, since one does not need to spend time and energy to reach these points. Then, we look at the nearest points and assign them a cost of 1. And then, we take a look at these points' neighbours and give them cost as 2. At each iteration, we try to look at the neighbours of the point with the lowest cost that has unchecked neighbours and the increment the for its neighbors by 1. This procedure is summarized in figure 7 for better understanding.



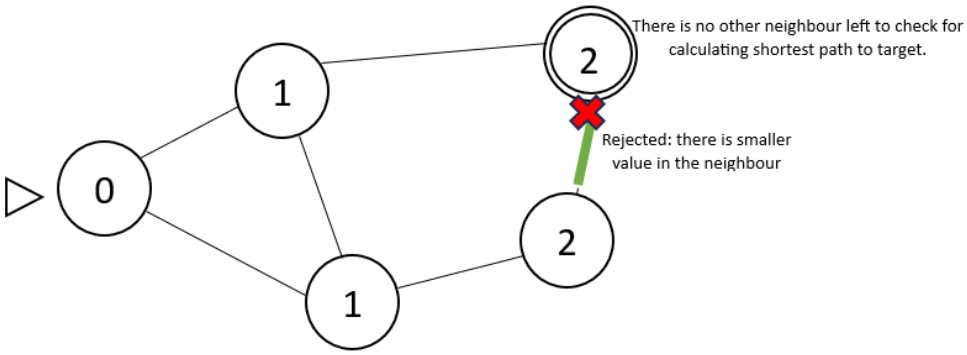
(a) Initial cost at the target point is 0, the cost of the other points is still unknown



(b) Looking at the neighbours of the point with the smallest cost. In this situation, it is the target point.



(c) Looking at the neighbours of the point with the smallest cost. We already looked at the target point, therefore it is time to take a look at the next smallest points.

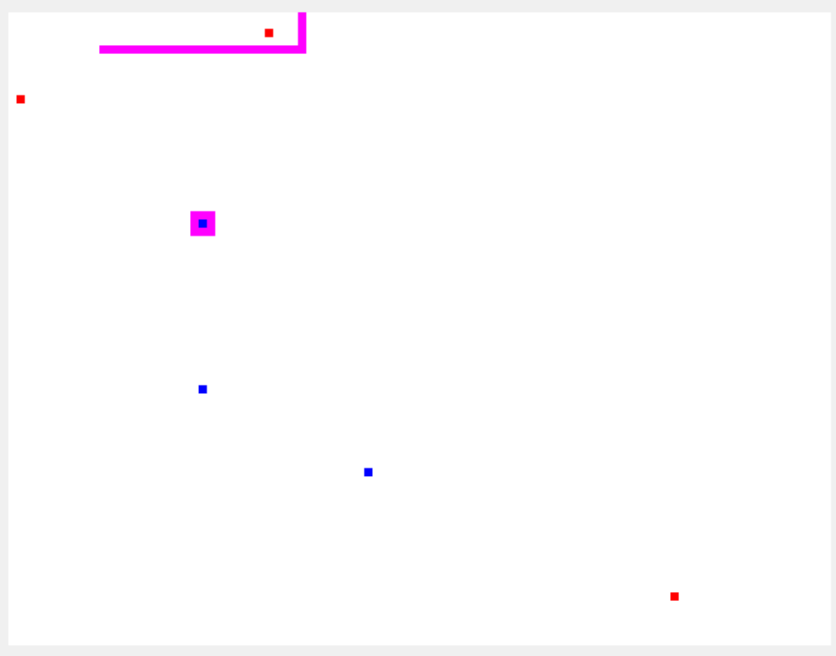


(d) The end result

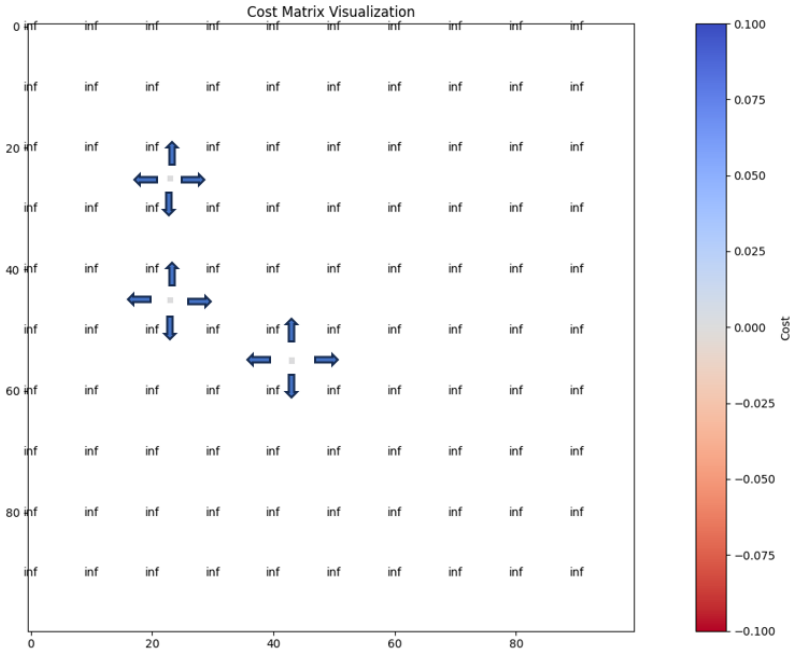
Figure 7: Example of how to use Dijkstra's algorithm

Given that our navigation domain for pedestrians is in 2D plane, we used Dijkstra's algorithm in a matrix framework. For each integer-valued coordinate point on the grid, we calculated the cost with the same method.

We provided an example for the scenario in figure 8:

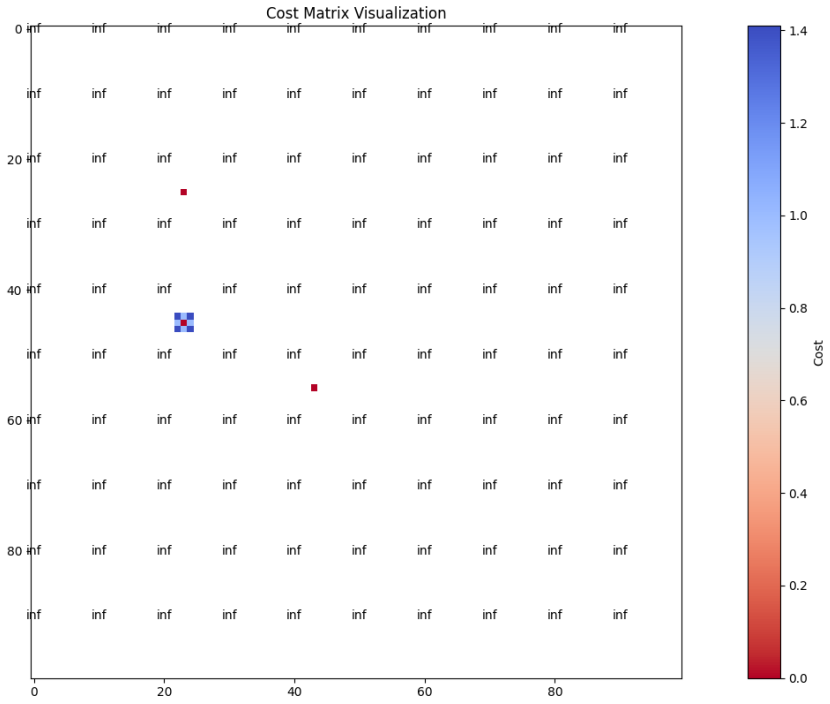


(a) The setup of the scenario. There are 3 pedestrians and 3 targets but one of the targets is fully covered by an obstacle.

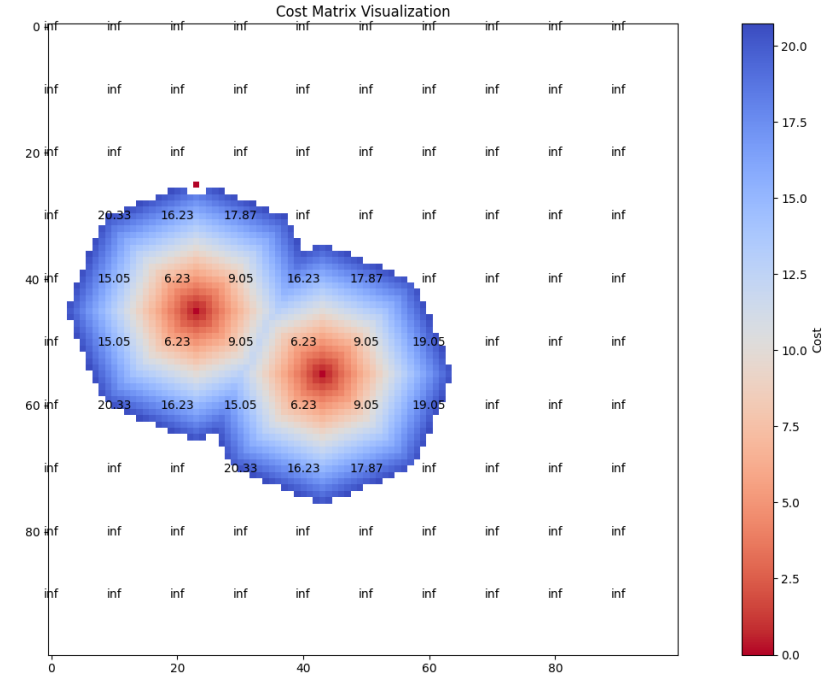


(b) Visualization of `cost_matrix`: Starting from target points (whose costs are 0), we will look at their neighbours and determine their respective costs.

Figure 8: Visualization of `cost_matrix` for a random test scenario

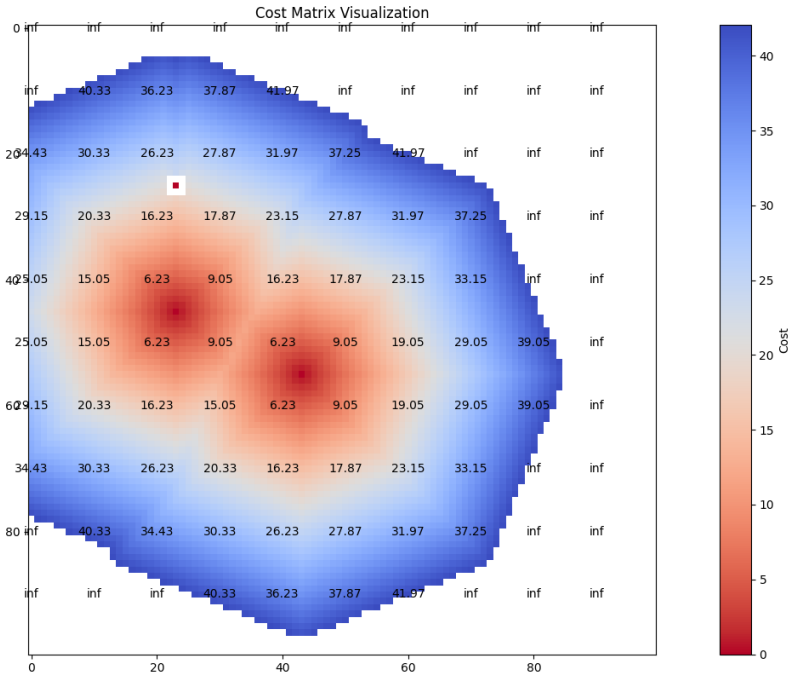


(c) Visualization of `cost_matrix`: Our matrix starts to expand. Even though the cost is very small near the target point, currently `matplotlib` displays them in blue. This will qualitatively change with the gain of more cost values.

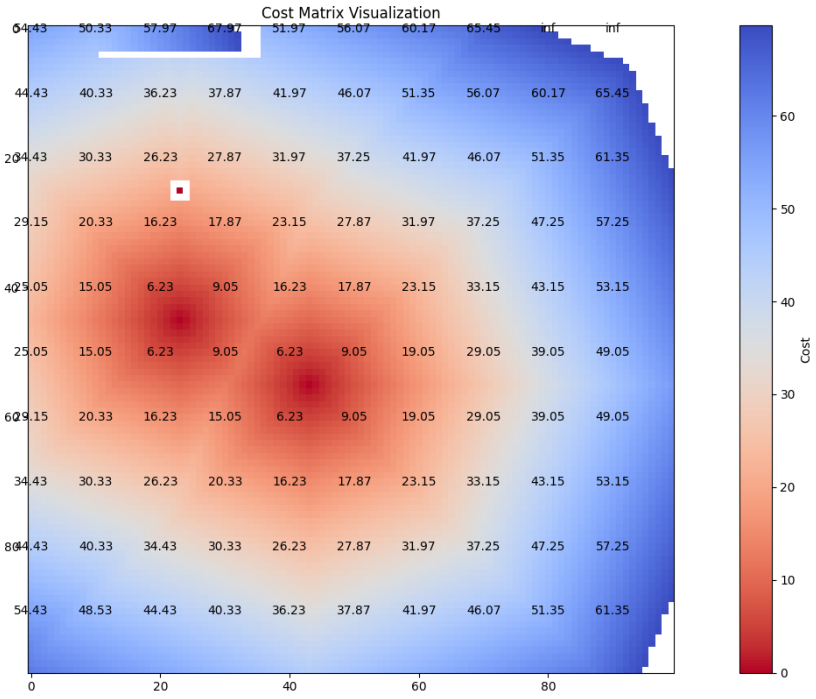


(d) Visualization of `cost_matrix`: It can be seen that there is no expansion from point (23,25), pictured in red, because obstacles are not considered as neighbours and thus, you cannot reach that target point from anywhere.

Figure 8: Visualization of `cost_matrix` for a random test scenario

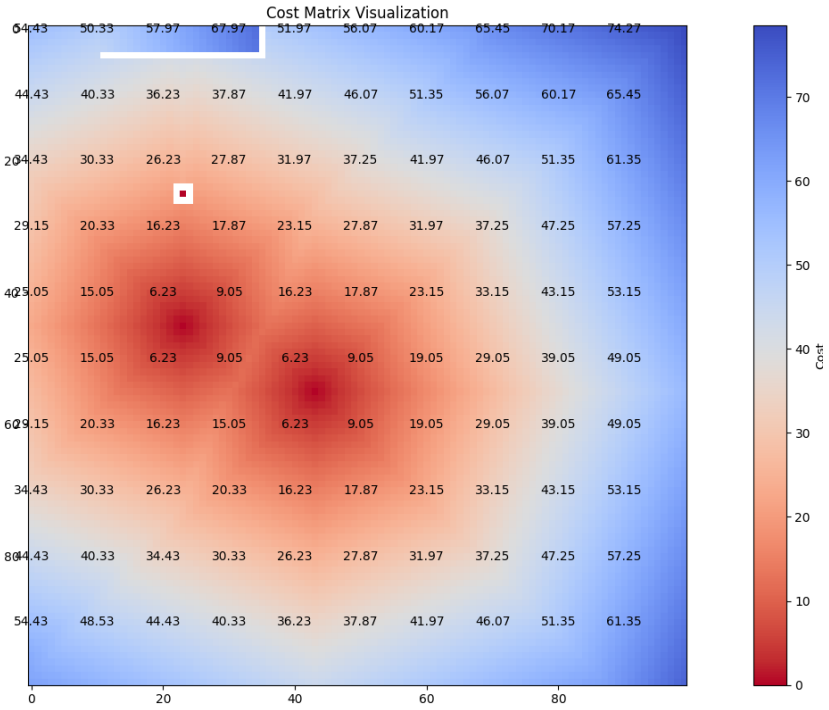


(e) Visualization of `cost_matrix`: The expansion continues to large distances and cost increases as well. Therefore, points close to the target point become more red rather than blue.



(f) Visualization of `cost_matrix`: There is an L-shaped obstacle on top of the grid, therefore Dijkstra's algorithm needs to work its way around starting from the open side on the left. This is why points further inside of the L-shaped obstacle have a larger cost than the ones outside.

Figure 8: Visualization of `cost_matrix` for a random test scenario



(g) The final `cost_matrix` using Dijkstra's algorithm

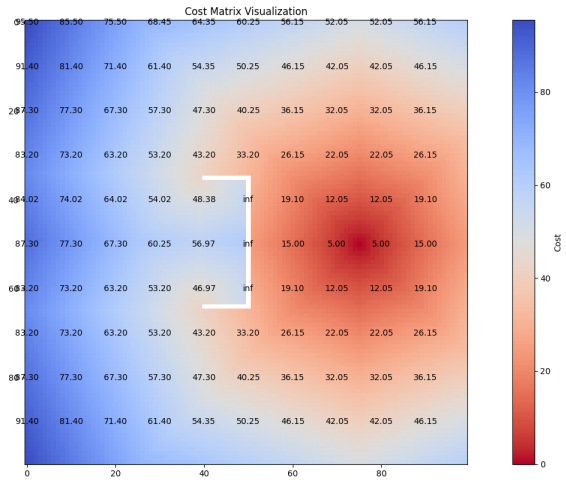
Figure 8: Visualization of `cost_matrix` for a random test scenario

This method works fine, but it is relatively slow compared to Euclidean distance, especially for large matrices. This was primarily due to one big reason. After looking at the neighbours of a point, our algorithm is not only looking at the next unobserved point with the lowest cost, but it looks at the cost of all neighbours of neighbours recursively. Therefore, with this method, we are looking at the same point several times which decreases our algorithm's performance significantly.

Subsequently, following our research and thinking, we found a way to increase performance, we can use `PriorityQueue` [1] whose time complexity is $O(n \cdot logn)$, compared to our first approach of time complexity $O(n^2)$.

Basically, `PriorityQueue` stores the smallest element (in our situation it refers to the cost) at the top of the queue so that it can pop when we want to use it.

You can see the chicken test implementation in figure 9.



(a) `cost_matrix` of Chicken Test

(b) The pedestrian does not move inside of the box but tries to go around it

(c) To reach the target with the shortest path, the pedestrian goes close to the obstacle.

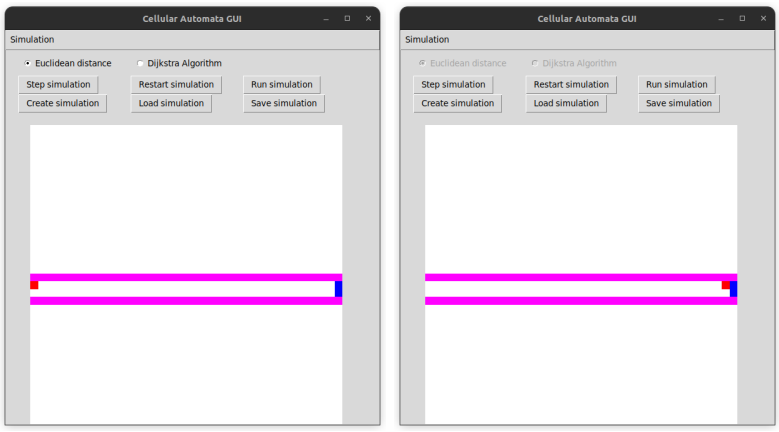
(d) Pedestrian tries to come close again to the border line in order to reach the target as quickly as possible using the shortest path.

Figure 9: Chicken Test using Dijkstra's shortest path algorithm

Report on task 5, Tests

The guideline "Richtlinie für Mikroskopische Entfluchtungsanalysen" (RiMEA, translation: "Policy for microscopic evacuation analysis") provides several test scenarios, four of which we will further investigate.

TEST1: RiMEA scenario 1 (straight line):
According to the given test case, 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.
I have initialized one pedestrian at the coordinate (0,20) with the target being at (0,40) and run the simulation.



(a) Starting position (b) Final Position

Figure 10: RiMEA Scenario 1 (straight line)

10a shows the initial configuration of task 1. 10b shows the final position of the pedestrian being absorbed by the target. The pedestrian reaches the target in `39.0 seconds` with a speed of `1.0 m/s`.

TEST2: RiMEA scenario 4 (fundamental diagram: density vs. velocity or density vs. flow):

For this scenario, we had a corridor of 5x275 meters and put different densities of people here for testing. We put different numbers of pedestrians in this corridor; however, as we increased the density, our computers could not handle the simulation. Like in the Figure 11.

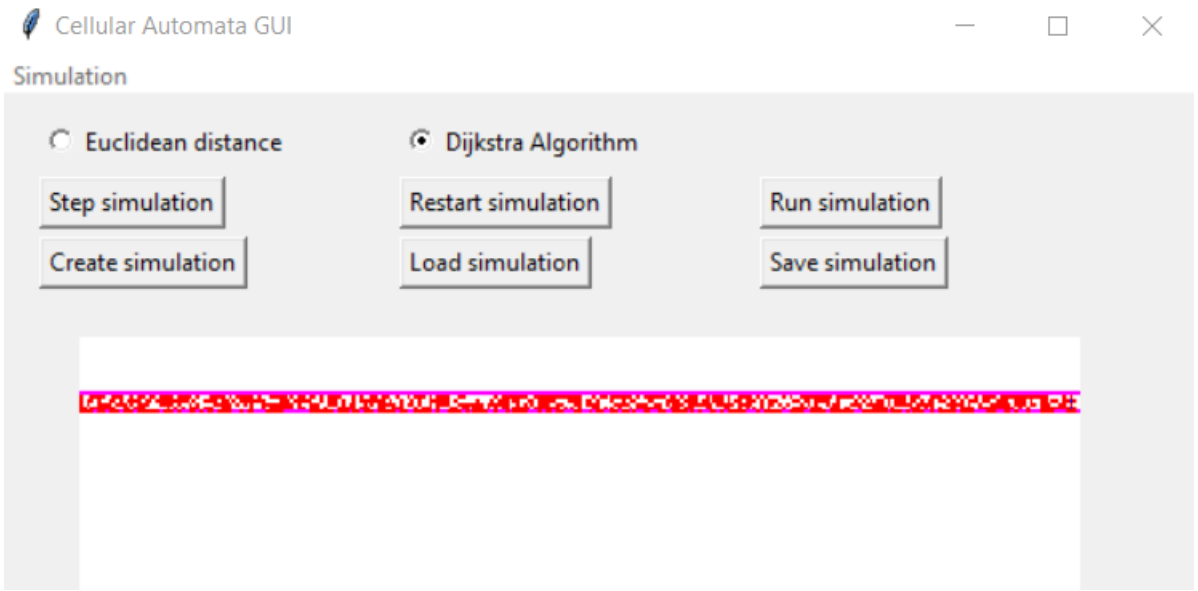


Figure 11: 1370 people put inside of a corridor

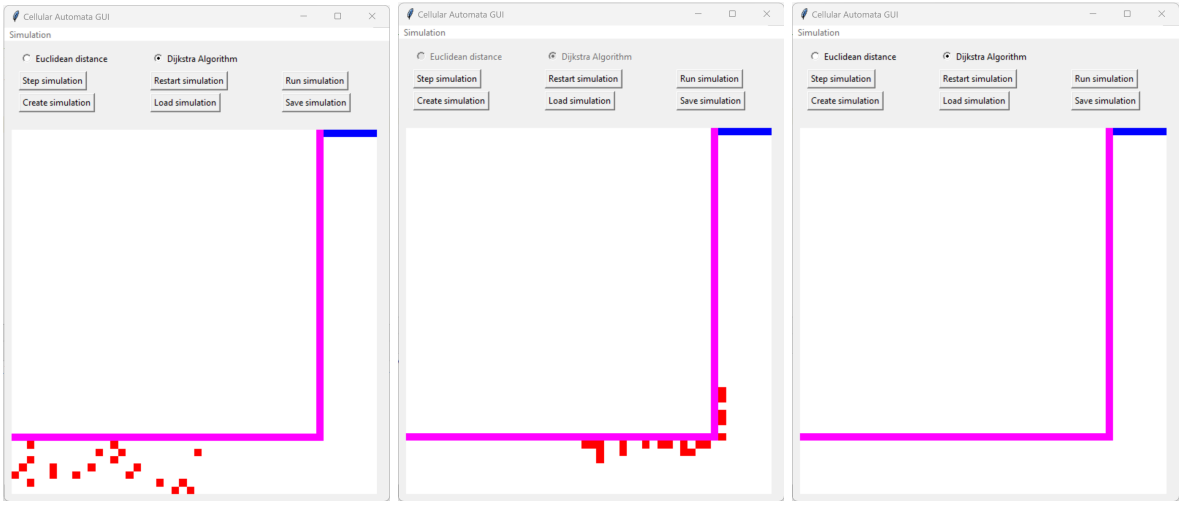
Therefore, instead of using all control points, we used only the ones near to target points with a much smaller number of pedestrians. We got some results, but they were not as expected. Our implementations were fine in other scenarios but in this one, we got erroneous values.

Basically, at the control points, our code tries to find the average speed values of pedestrians going through that point and we store these values inside of `self.speeds225_18, self.speeds250_17, self.speeds250_18` variables, depending on their location. After getting the average speed for each second, we store this infor-

mation inside of `total_speeds225_18, total_speeds250_17, total_speeds250_18` attributes. In the end, we just take the average of the average speeds at each second. Then, print them on the console. Although the results were erroneous most probably due to some mistakes in our implementation, the simulation seemed to be running correctly.

To avoid creating many pedestrians with random attributes by hand, we created and used the file `pedestrian_generator.py`.

TEST3: RiMEA scenario 6 (movement around a corner):
In the RiMEA test scenario 6, a 2m wide corridor with a corner is to be modeled, which can be seen in figure 12a. The file `RiMEA6.json` provides the corresponding setup, where 4 grid cells equal 1 square meter. Figure 12b and 12c show the process of the pedestrians walking to the right side before moving up to the target in order to avoid the obstacle. Without the obstacle, the pedestrians are able to move diagonally and thus reach the target faster.



(a) Starting position (b) Process (c) Target position

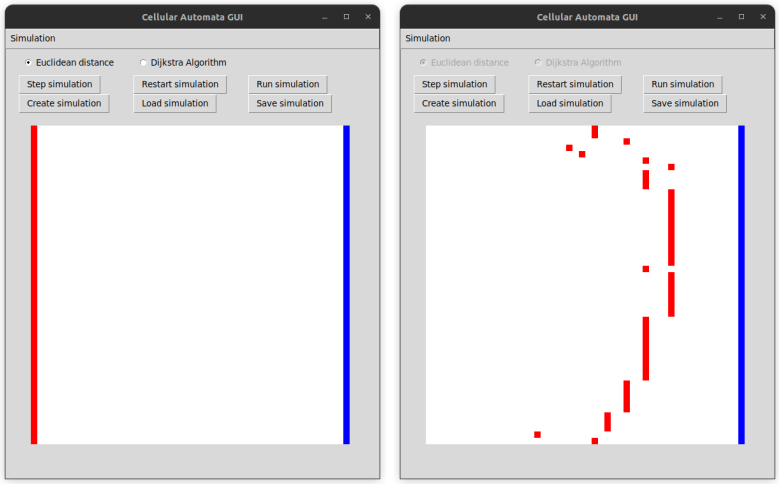
Figure 12: In RiMEA scenario 6, the pedestrians move around a corner

TEST4: RiMEA scenario 7 (demographic parameters):
In this test, we have to consider 50 pedestrians with varying walking speeds distributed according to walking speed distributions described in Figure 2 of the RiMEA document [2]. In order to do this we generated speeds for different age groups sampled from the estimates based on the figure, as outlined in Table 1.

Age group	Number of pedestrians	Min Speed [m/s]	Max Speed [m/s]
0-10	5	0.60	1.20
10-20	5	1.20	1.60
20-40	10	1.50	1.60
40-50	10	1.40	1.50
50-60	10	1.30	1.40
60-70	5	1.10	1.30
70-80	5	0.60	1.10

Table 1: Speed distribution as per age

Figure 13a demonstrates the initial position of all the pedestrians starting from the same column with targets being located at the other end. After running the simulation for some time, we can see that in figure 13b different age groups form different movement groups corresponding to their walking speeds. The outcome aligns with the movement patterns outlined in the RiMEA document.[2] The statistical values have also been saved in the `statistics.json` file for further investigation.



(a) Starting position (b) Process

Figure 13: The different age groups can be seen forming a movement group

References

[1] `https://docs.python.org/3/library/queue.html`. Last accessed 01.11.2023.

[2] `https://rimeaweb.files.wordpress.com/2016/06/rimea_richtlinie_3-0-0_-_d-e.pdf`. Last accessed 01.11.2023.

[3] `https://docs.python.org/3/library/time.html`. Last accessed 01.11.2023.

[4] `https://docs.python.org/3/library/tkinter.html`. Last accessed 01.11.2023.