Report for exercise 1 from group J

Tasks addressed:	5
Authors:	Wenbin Hu (03779096)
	Yilin Tang (03755346)
	Mei Sun (03755382)
	Daniel Bamberger (03712890)
Last compiled:	2023–05–08
Source code:	https://github.com/HUWENBIN2024/TUMCrowdModelingGroupJ/tree/main/ex1

The work on tasks was divided in the following way:

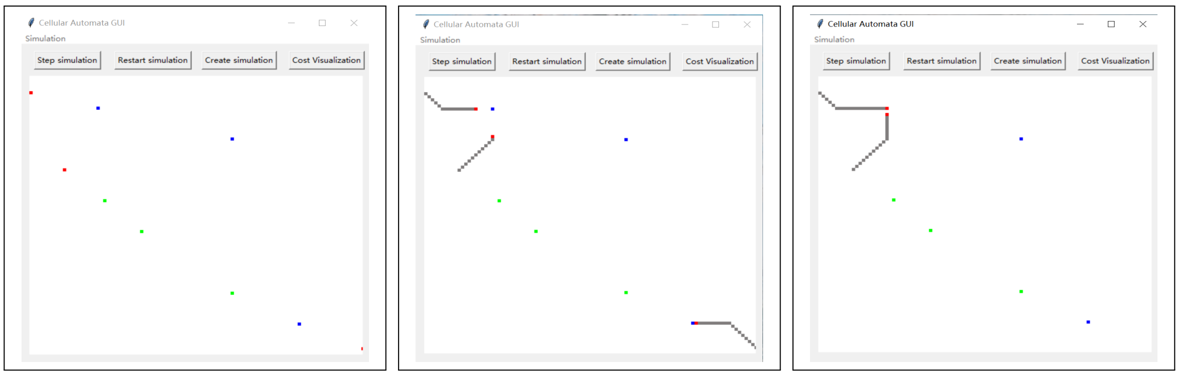
Wenbin Hu (03779096)	Task 1	25%
	Task 2	25%
	Task 3	25%
	Task 4	25%
	Task 5	25%
Yilin Tang (03755346)	Task 1	25%
	Task 2	25%
	Task 3	25%
	Task 4	25%
	Task 5	25%
Mei Sun (03755382)	Task 1	25%
	Task 2	25%
	Task 3	25%
	Task 4	25%
	Task 5	25%
Daniel Bamberger (03712890)	Task 1	25%
	Task 2	25%
	Task 3	25%
	Task 4	25%
	Task 5	25%

Report on task 1, Setting up the modeling environment

We created a graphical visualization of the environment for this project. It consists of a squared grid and it is possible to specify the size, as well as the length of the side of each cell. There are three main objects: *gui*, *scenario*, and *pedestrian*. This was practically implemented in Python, using the Tkinter package for the graphical aspects of the project.

The ***gui*** is implemented by Tkinter Canvas. It implements the user interface design. The GUI mainly consists of two main parts: buttons and canvas. Fig. 1 shows the GUI.

```
python main.py --json_path ../scenarios/task1.json --iter 100
```



(a) (b) (c)

Figure 1: [a] shows the beginning of the simulation; [b] shows the simulation after 10 time steps; [c] shows the end of the simulation.

As Fig. 1 shows, the window consists of 4 buttons and a canvas.
Table 1 shows the functions of these buttons.

Button	Function
Step simulation	Simulate the pedestrians walking 1 step.
Restart simulation	Reset the scenario.
Create simulation	Demonstrate the entire process from the start state to the end state.
Cost Visualization	Shows the cost in canvas.

Table 1: The functions of buttons

Code and implementation The whole project is implemented in Python and Tkinter is used to manage the graphical aspects. The grid is implemented through a class inheriting from Tkinter's Canvas. The ***scenario*** has three main elements: pedestrian(red), obstacle(green), target(blue).
Table 2 shows the elements and property of scenario:

Element	Property	Description
pedestrian	pedestrian_positon	The coordinates of pedestrian.
	speed	Speed The speed of pedestrian.
	path	Path The list contains all position.
	pedestrian_cost	The cost imposed on a pedestrian by other pedestrians.
target	target_position	The coordinates of target.
	target_distance_grid	The distance between the relevant cell and its nearest target.
obstacle	obstacle_position	The coordinates of obstacle.

Table 2: The elements and properties of scenario

Finally, the MainGUI class implements the visualization of the interface and the feasibility of commands, the Pedestrian class implements all the pedestrians' behaviors, and the Scenario class realizes the transformation of the scenario and loads the scenario from the json file. Their main methods are:

Class	Method	Description
MainGUI	step_scenario	Moves the simulation forward by one step, and visualizes the result.
	load_scenario	Load a specific scenario described by a json file.
	parse_args.distance_mode	defualt = Dijkstra mode ; if –iter distance = Euclidean distances mode.
Pedestrian	get neighbors	Compute all neighbors in a 9-cell neighborhood of the current position.
		-return: A list of neighbor cell indices (x,y) around the current position.
	update_step	Moves to the cell with the lowest distance to the target.
Scenario	update_target_grid	Computes the shortest distance from every grid point to the nearest target cell.
		-returns: The distance for every grid cell, as a np.ndarray.
	update_step	Updates the position of all pedestrians.
	target_grid_to_image	Creates a colored image based on the distance to the target stored in self.target_distance_grids.
	to_image	Creates a colored image based on the ids stored in self.grid.

Table 3: The main method of gui

Change the scenario without changing the code It is possible to change the scenario without changing the code in our project. We can change it by CLI as below. In the command, we specify a json file containing coordinate information to load the coordinates of pedestrians, obstacles, and targets.

```
# Simulation

python main.py --json_path <path of a json file> --iter <# of steps> --distance_mode <
    dijkstra or euclidean> --r_max <r max param>

An example:

python main.py --json_path ../scenarios/sc0.json --iter 100 --distance_mode dijkstra --r_max
    2
```

Here is an example of a json file that defines the "shape"(length and width) of the canvas and contains the coordinate information of pedestrians, obstacles, and targets:

```
# Json

{
    "shape": [200, 200],
    "targets": [
                    [50, 20],
                    [180, 60]
                ],
    "pedestrians": [
                    [[31, 2]     , 2.3],
                    [[0, 0]      , 1.8],
                    [[199,59]    , 2.1]
                ],
    "obstacles": [
                    [20, 10],
                    [21, 10]
                ]

}
```

Report on task 2, First step of a single pedestrian

Task 2 consists in defining a scenario with 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).
This scenario can be executed in our cellular automaton step by step or in one fell swoop with the Run button. Specifically, each cell has a side of 1 meter and the walking speed of the pedestrian is 1m/s, therefore, each step along the cardinal axes is going to cover 1 meter and take 1 second to be accomplished, while each diagonal step is going to cover 1.4 meters in 1.4 seconds. As expected and shown in Fig. 2, in the experiments the pedestrian reaches the target in 20 seconds.
The target is placed at the bottom, therefore, when the pedestrian reaches the destination, it will be covered by the pedestrian.

```
python main.py --json_path ../scenarios/task2.json --iter 100
```

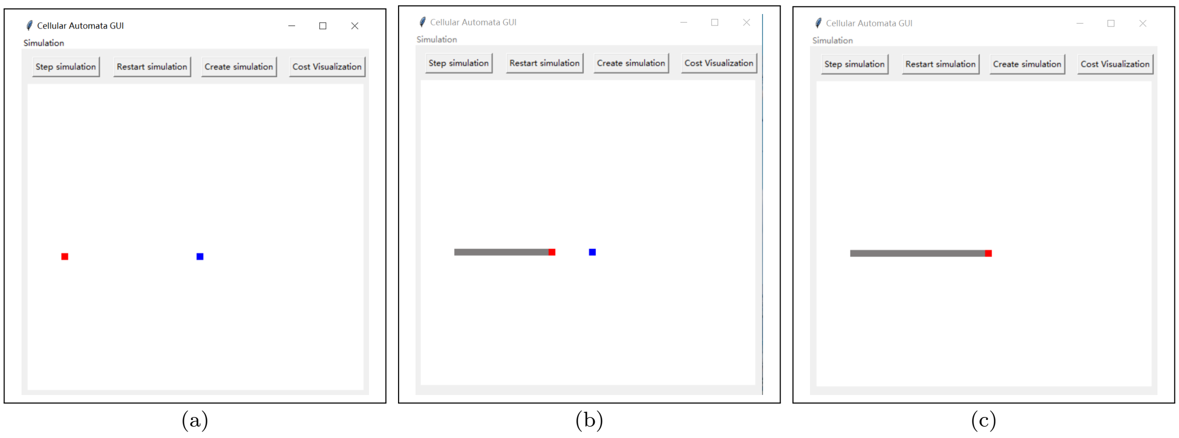


Figure 2: [a] The beginning of the simulation; [b] In the middle of the simulation; [c] The end of the simulation.

Report on task 3, Interaction of pedestrians

Pedestrians Interaction

Naturally, there exists repulsive forces among pedestrians, which obeys empirical observations and theoretical social force models. One of crucial goals of crowd modelling and simulation is to avoid crush and stampede in real scene. Without pedestrians interaction implementation, pedestrians will probably overlap each other, which is shown in Figure 3.

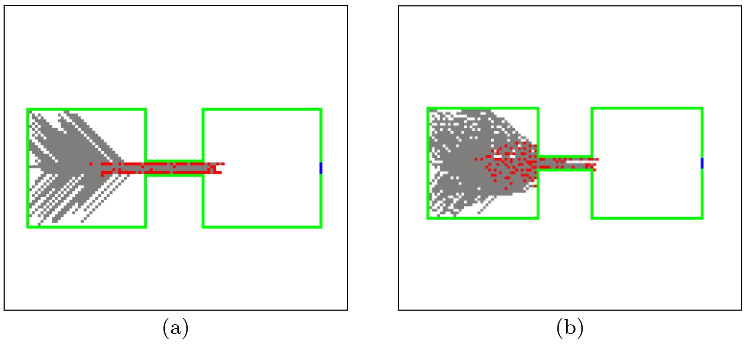


Figure 3: [a] demonstrates a pedestrians-overlap effect. [b] shows the result after we implemented the pedestrian repulsive force.

Following the provided instruction, the repulsive force between two individuals (i, j) is simulated as:

$$C(r_{i,j}) = \mathbb{I}(r_{i,j} < r_{max}) \cdot \exp(\frac{1}{r^2 - r_{max}^2}) \tag{1}$$

, where $I(\cdot)$ is the indicator function, r_{max} is the threshold distance for a force effect, and $r_{i,j}$ is the distance of individual (i, j). When implementing it, we need consider the running complexity. It is not stable and not efficient to iterate all the pedestrians for each individual-received force during stepping. Our method is conducting a preprocessing before scenario updates. For each grid, we assign its resultant force by summing up the force by its neighbourhood 3, where the neighbourhood is defined as 2.

$$\{(x, y) \mid (x - x_p)^2 + (y - y_p)^2 < r_{max}^2; x, y, x_p, y_p \in [0, width) \cap N\} \tag{2}$$

,where (x_p, y_p) is the coordinate of the grid p.

$$C_p = T + \beta \sum_{n \in N(p)} C(\|n - p\|_2) \tag{3}$$

, where C_p restores the received resultant force of grid p, T is cost function by the target-grid distance, β is a hyperparameter to adjust the interaction influence, and $\|n - p\|_2$ represents the distance of grid n, p.

Speed Control

In real situation, people have different walking speeds. As a result, it is important that we need to conduct a speed control. The difficulty of the speed implementation lies on two aspects: (I) The desired moving distance is larger than the original step length. (II) Vise versa. For (I), we do a while loop to complete a multi-standard-step move. For (II), we accumulate the desired step length for next time step, and the pedestrian doesn't move. The idea is trivial while the implementation needs circumspection.

Experimental Setting

In this task, the goal is to place 5 pedestrians equally spaced around a target with 30-50m distance. We chose to place the Pedestrians at a distance of 40m around the target. Therefore we need a scenario with a width and height of at least 81. Therefore we chose for the scenario to be 101m x 101m in size, with Target Cell $T = (50, 50)$

For the pedestrians to be distributed approximately at angle $\frac{360°}{5} = 72°$ to each neighbor at 40m distance, we trivially start to place the first Pedestrian P_0 at 40m distance vertically at cell $x_0 = 50$ and $y_0 = 10$ and rotate this position by $i * 72°$ around the target for pedestrian i.

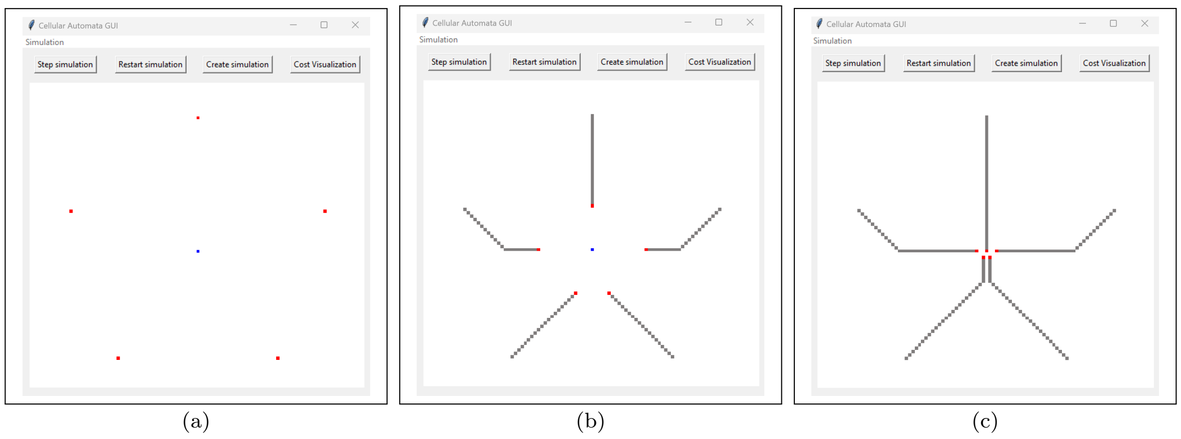


Figure 4: Three different moments of task3. [a]: The initial setup; [b]: The pedestrian state in the middle of the simulation; [c]: The pedestrian state at the end of the simulation.

$$\begin{pmatrix} x_i \\ y_i \end{pmatrix} = \begin{pmatrix} \cos\left(i * \frac{2\pi}{5}\right) & -\sin\left(i * \frac{2\pi}{5}\right) \\ \sin\left(i * \frac{2\pi}{5}\right) & \cos\left(i * \frac{2\pi}{5}\right) \end{pmatrix} \begin{pmatrix} x_0 - x_T \\ y_0 - y_T \end{pmatrix} + \begin{pmatrix} x_T \\ y_T \end{pmatrix}$$

With x_T, y_T being the position of the target cell it can be see that in order to be rotated correctly around the target, the first pedestrians' position translated to be rotated relative to $(0, 0)$, rotated by $72°$ or $\frac{2}{5}\pi$ and then shifted back to be relative to the target. After that we round to the nearest discrete cell position. This yields the following starting positions:

$$(50, 10), (12, 38), (26, 82), (74, 82), (88, 38)$$

All these 5 pedestrians are assigned a specific equal speed v = 1.41m/s.

Experimental Result

For the test scenario the simulation ran as shown in 4. Theoretically, all the 5 pedestrians should arrive the target at the same time. However, the result shows that pedestrians arrive the target with a slight different time. This is caused by the imprecision of the our speed implementation. Though it is not exactly accurate, our speed is fundamentally correct, and the flaw is out of human perception.

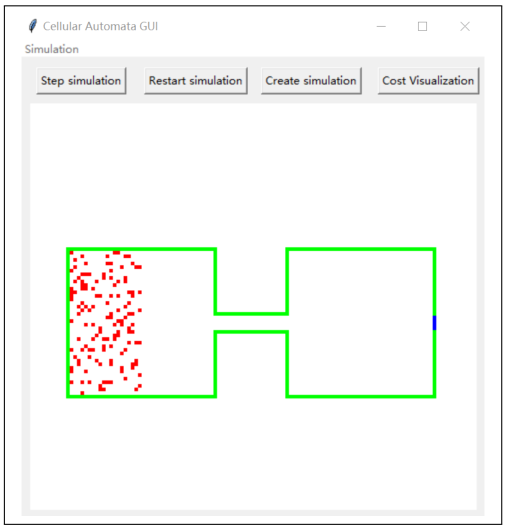
If you want to reproduce the result, you can run this line of code:

```
python main.py --json_path ../scenarios/task3.json --iter 100 --distance_mode euclidean --
      r_max 2
```

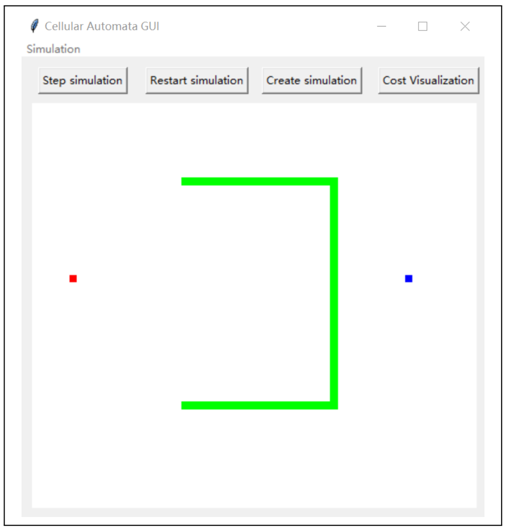
Report on task 4, Obstacle avoidance

Task 4 consists of different implementations of obstacle avoidance functions. We constructed the environment consisting of two different scenarios to test the performance of these algorithms. The bottleneck and the chicken test. For both scenarios, 150 pedestrians were distributed randomly and uniformly within the respective areas. The initial state of both scenarios is shown in Fig. 5.

```
python main.py --json_path ../scenarios/task4_bottleneck.json
python main.py --json_path ../scenarios/task4_chickentest.json
```



(a)

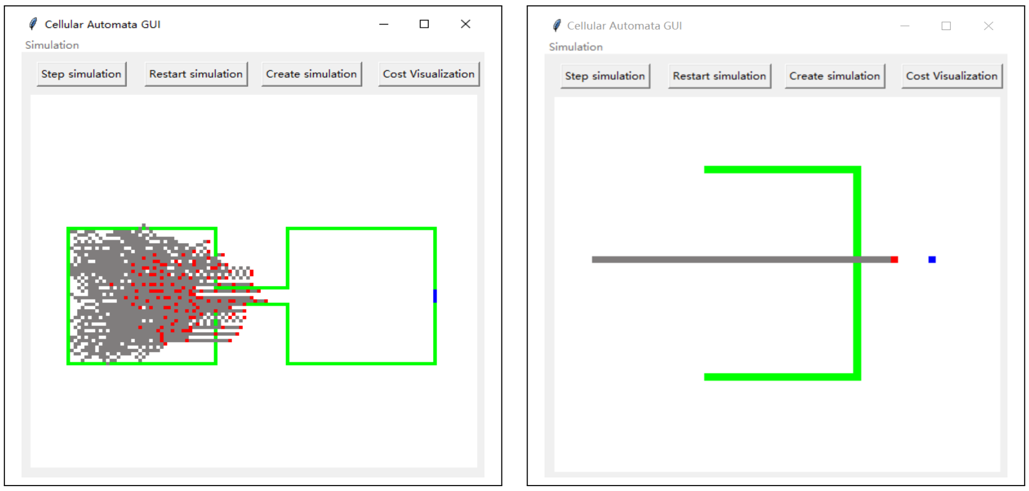


(b)

Figure 5: [a] The initial state of bottleneck scenario; [b] The initial state of chicken test scenario

Without Obstacle Avoidance Without any kind of obstacle avoidance implemented, all pedestrians move with no prior knowledge of obstacles, but only consider direction towards the nearest target. The pedestrians will consider other pedestrians and obstacles as blank cells, therefore it is inevitable that they step on one another and that obstacles are unable to act as barriers. For example, in the case depicted in Fig. 6, in the chicken test scenario the pedestrians would just step on the obstacle and other pedestrians as they were not there at all, as well as the bottleneck scenario, the pedestrians would not pass through the bottleneck but simply go straight towards the target.

```
python main.py --json_path ../scenarios/task4_bottleneck.json --iter 100 --distance_mode
    euclidean --r_max 2
python main.py --json_path ../scenarios/task4_chickentest.json --iter 100 --distance_mode
    euclidean --r_max 2
```



(a) (b)

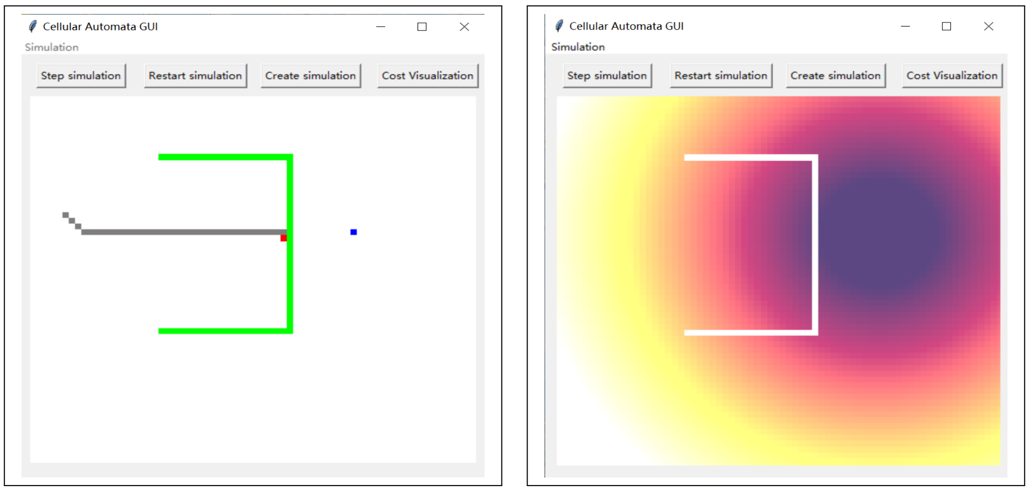
Figure 6: [a] The bottleneck scenario without obstacle avoidance; [b] The chicken test scenario without obstacle avoidance

With Rudimentary Obstacle Avoidance The rudimentary obstacle avoidance implemented in this task consists of making the pedestrians simply not step on any obstacles or other pedestrians. This is simply achieved by assigning a infinity number to obstacle grids when calculating the Euclidean distance cost. It can be shown that it just needs one line of code.

```
distances[self.grid == self.NAME2ID['OBSTACLE']] = np.inf
```

For the chicken test, with this kind of obstacle avoidance, the pedestrian (red) will start walking towards the target (blue) until it gets trapped by the obstacles (green), and at this point it will stay there until the simulation ends. Fig. 7 shows the ends with the pedestrian stuck in the trap and the cost of simulation with rudimentary obstacle avoidance. Command line for result reproduction.

```
python main.py --json_path ../scenarios/task4_chickentest.json --iter 100 --distance_mode
    euclidean --r_max 2
```



(a) (b)

Figure 7: [a] The pedestrian stuck in the trap; [b] The cost of simulation with rudimentary obstacle avoidance

Obstacle Avoidance with Dijkstra Algorithm It is not easy to simply build a cost field for avoiding obstacles, since the situation is too complicated and our step-algorithm just do a greed algorithm. To achieve a long-term benefit, we come up with the Dijkstra Algorithm. Dijkstra Algorithm works on a graph, so we build an undirectional graph. Inside the graph, each vertex represents a grid, and each edge represents a path connecting neighbours, which can be illustrated in Figure 8. Since the diagonal and vertical/horizontal path length are different, which are $\sqrt{2}$m and 1m respectively. To reach a more realistic simulation, we set the weight of a diagonal edge and a vertical/horizontal edge as $\sqrt{2}$ and 1 respectively.

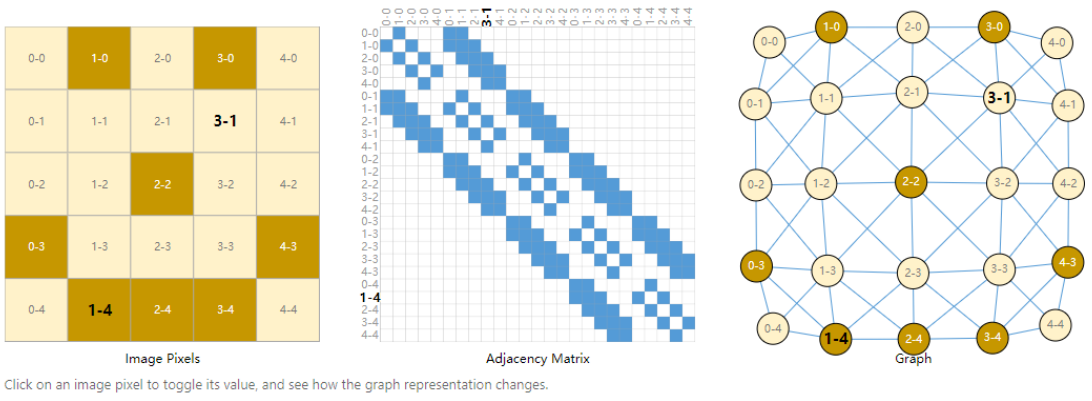


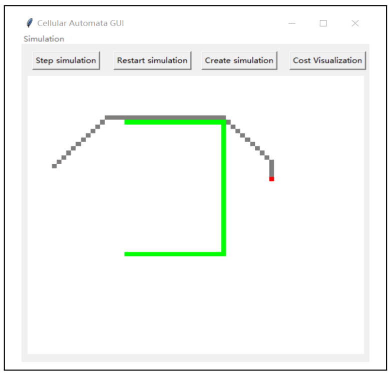
Image Pixels
Adjacency Matrix
Graph
Click on an image pixel to toggle its value, and see how the graph representation changes.

Figure 8: It illustrates how we construct an undirectional graph. [2]

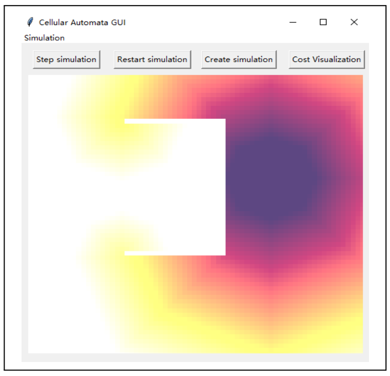
For reproduction, please run this line of code:

```
python main.py --json_path ../scenarios/task4_chickentest.json --iter 100 --distance_mode
    dijkstra --r_max 2
```

We demonstrate our results of 2 scenarios to the effectiveness of our method, which are **Chicken Test** and **Bottleneck Test**. As shown in Fig. 9, the **Chicken Test** is completed successfully since the red pedestrian can circumnavigate the U-shaped obstacle. The cost of simulation with Dijkstra algorithm as shown in Fig. 9(b), shows a clear eight-maned star pattern because each cell's distance is calculated on the basis of the eight neighbors' distances, rather than a real representation of distance towards the target. For the **Bottleneck Test**, both the rudimentary obstacle avoidance and the more sophisticated Dijkstra implementation can achieve reaching the objective. The difference is which path they choose. As shown in Fig. 10 this algorithm makes it so that the optimal path is promptly found, making all the Pedestrians form a nice queue without moving to unnecessary cells.



(a)

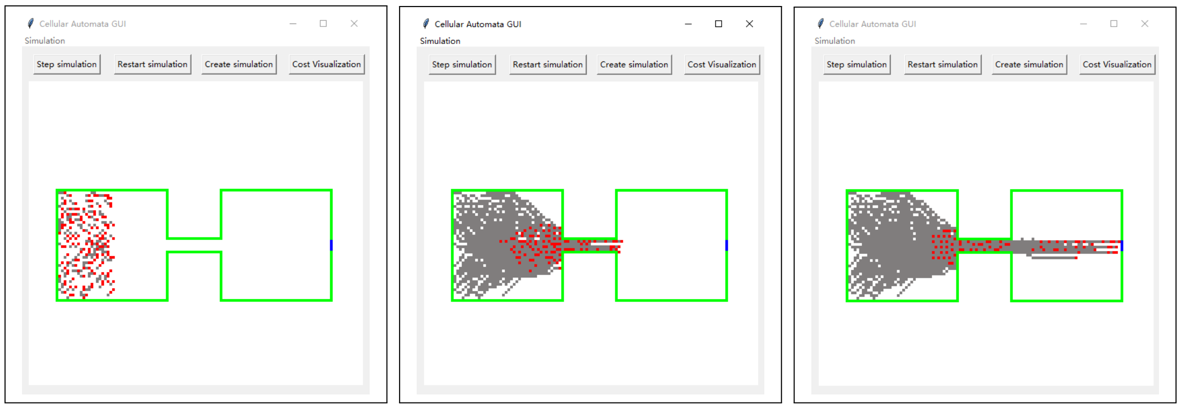


(b)

Figure 9: Chicken test executed with the Dijkstra algorithm. [a]: The observed path of pedestrians; [b]: The cost of simulation with Dijkstra algorithm obstacle avoidance

For reproduction, please run this line of code:

```
python main.py ——json_path ../scenarios/task4_bottleneck.json ——iter 200 ——distance_mode
    dijkstra ——r_max 2
```



(a) (b) (c)

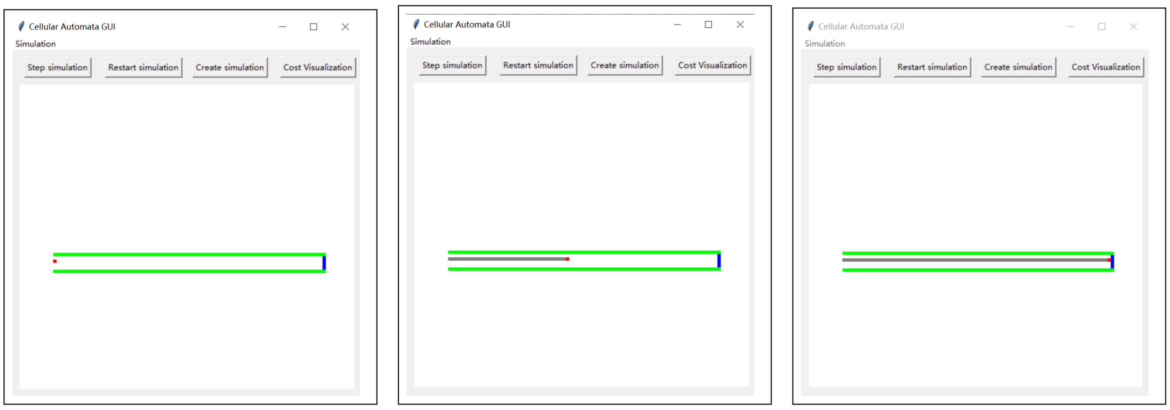
Figure 10: Bottleneck test executed with the Dijkstra algorithm. [a]: The pedestrian state after 2 steps; [b]: The pedestrian state in the middle of the simulation; [c]: The pedestrian state near the end of the simulation

Report on task 5, Tests

The RiMEA [1] guidelines are used to validate the implementation of modeling human crowd by defining the specific scenarios.

TEST 1- RiMEA scenario 1: straight line, ignore premovement time
Scenario description: one pedestrian with a walking speed of 1.33m/s in a 2m wide and 40m long corridor that ends with targets. Here, we define each cell as 0.25 m^2, and the update between two graphical is implemented by calling sleep to simulate a passing time step, in addition, we also ignore the premovement time as it is the time span between the activation of the signal and the start of evacuation of an individual person.
Under this scenario, the expected travel time should lie in the range of 26 to 34 seconds. Our experiment shows that the simulation time is 27.05 s and the pedestrian reaches the target with 80-time steps. This exactly satisfies the requirement of the scenario 1 test.

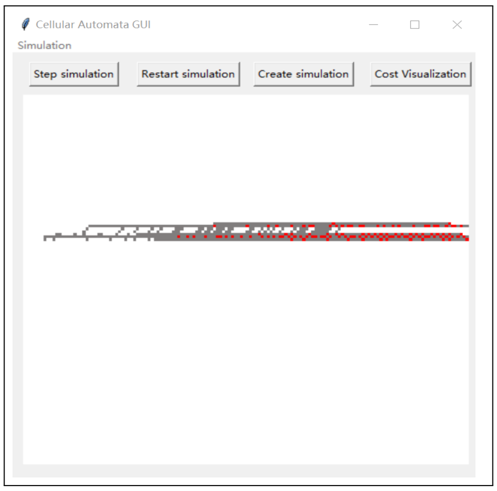
The following figure illustrates 3 stages of the simulation.



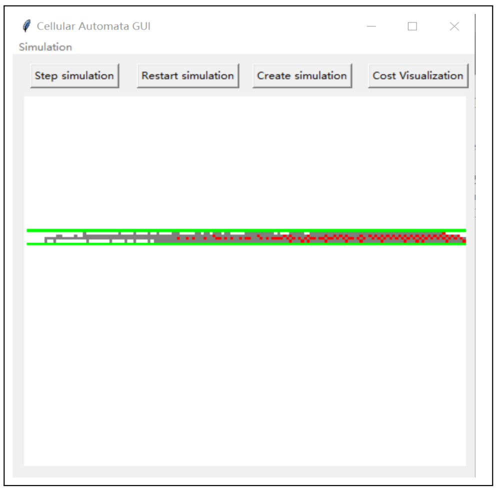
(a) (b) (c)

Figure 11: Three different moments of the RiMEA scenario 1. [a] Initial setup for scenario 1; [b] The pedestrian state in the middle of the simulation; [c] The pedestrian state near the end of the simulation

TEST 2- RiMEA scenario 4: fundamental diagram, density vs. velocity or density vs. flow
Scenario description: This scenario requires a corridor 1000m long and 10m wide with different pedestrians densities.($0.5p/m^2, 1p/m^2, 2p/m^2, 3p/m^2, 4p/m^2, 5p/m^2, 6p/m^2,$).But, unfortunately, we couldn't load such a large number of cells and pedestrians to implement the simulation after waiting a long time.So, we decided to change to an approximation version for this scenario.We scale to a smaller area size (50m*50m)while maintaining the same pedestrian densities. Figure 12 shows the partial simulation for test2. Without obstacles(Figure 12(a)), We noticed that the direction in which the crowd was walking started to become mixed. This is easily reminiscent of the real world, where serious crowd crush can very easily occur under such a situation. Besides, it takes a long to time for all the pedestrians move to the safety area. From Figure 12(b), the obstacles present in the area proved to be effective diversions, allowing the walkers to move in an orderly fashion towards the safety zone.



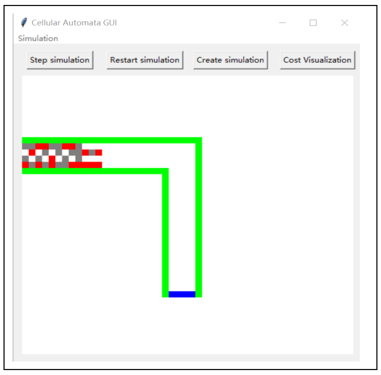
(a)

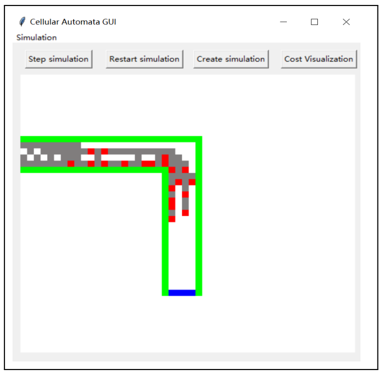


(b)

Figure 12: [a]Without obstacles; [b] Add obstacles

TEST 3- RiMEA scenario 6: Movement around a corner
Scenario description: There are 20 pedestrians in a corridor and moving around a corner to reach the target located at the end of the corner.
Under this scenario, all the pedestrians should turn right successfully without passing through walls. This is implemented by the obstacle avoidance mechanism.


(a)


(b)

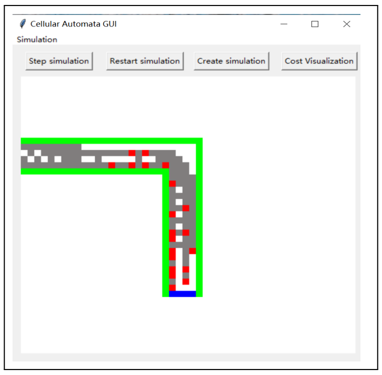
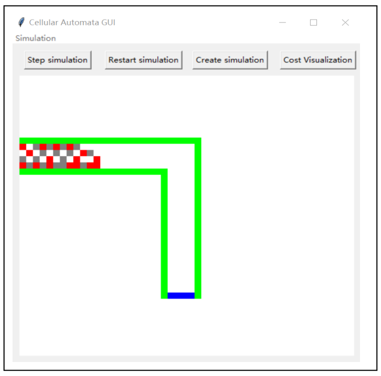
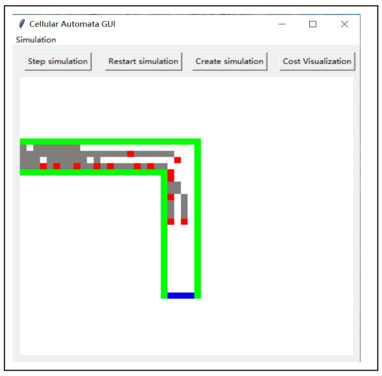

(c)

Figure 13: Three different moments of the simulation with Dijkstra algorithm.


(a)


(b)

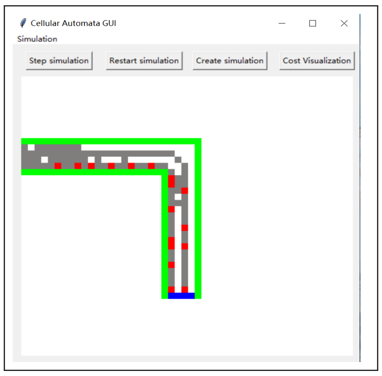

(c)

Figure 14: Three different moments of the simulation with Euclidean algorithm.

From Fig. 13 and Fig. 14, we could find that both of these two algorithms turn around the corner without passing through walls as expected. But the slightly different thing is that, in the case of the Euclidean algorithm (Fig. 13), the pedestrians are more careless. This is caused by the property of the Euclidean algorithm. It just considers the distance between the candidate update cell and the target. Thus, it is the simple and easiest algorithm.
As for the Dijkstra algorithm (Fig. 14), it is a well-known algorithm used to find the shortest path between a starting node and all other nodes in a graph (breadth-first). Every pedestrian computes the best path for itself and tries to follow it as much as possible. Since we take the person into account which means that overtaking is not possible. So, when the optimal path is occupied by other pedestrians, they will typically wait until they are able to move forward along their desired path.
By comparing the travel time of this simulation. The Euclidean algorithm takes 39s to reach the targets, however, it only takes 30 s with the Dijkstra algorithm. Therefore, the Dijkstra algorithm is more efficient than Euclidean.

TEST 4- RiMEA scenario 7: demographic parameters
Scenario description: Select 50 pedestrians with different walking speeds based on the speed distribution in Figure 2 of RiMEA guideline. We randomly sample the speed in a range of 0.6 m/s to 1.6 m/s. Concretely, the speed for every age interval is shown as follows.

- 3-10 years old: speed $\in [0.6, 1.2) m/s$
- 11-20 years old: speed $\in [1.2, 1.6) m/s$
- 21-50 years old: speed $\in [1.4, 1.6) m/s$
- 51-70 years old: speed $\in [1.1, 1.4) m/s$
- 71-80 years old: speed $\in [0.7, 1.1) m/s$

We show the three different stages of the simulation. Fig.15(a) is the initial setup. Fig. 15(b) illustrates pedestrians with higher speeds are able to move towards their destination more quickly, while those with lower speeds are still on their way. Fig. 15(c) display all the pedestrians with lower speed reach the target.

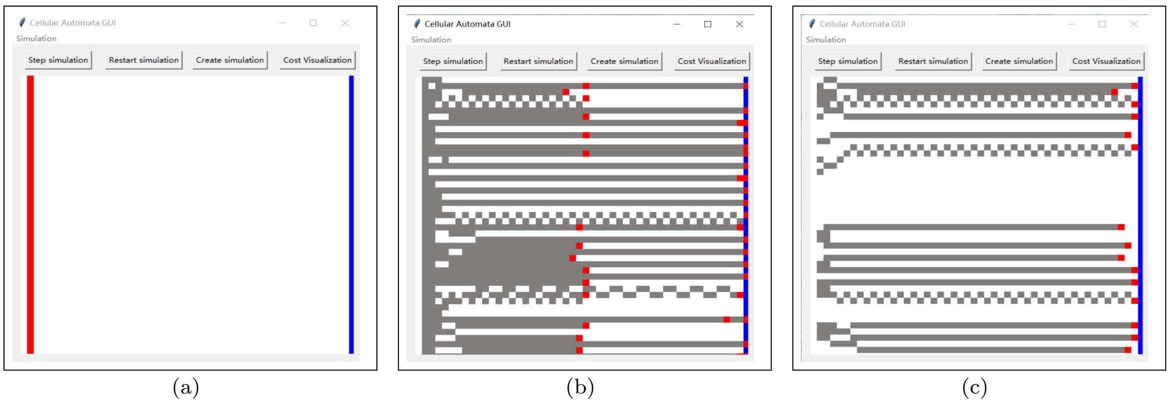


Figure 15: Pedestrians walking speed distribution.

References

[1] G Rimea. Richtlinie für mikroskopische entfluchtungsanalysen [guidelinefor microscopic evacuation analysis] version 3.0, 2016.

[2] Benjamin Sanchez-Lengeling, Emily Reif, Adam Pearce, and Alexander B. Wiltschko. A gentle introduction to graph neural networks. *Distill*, 2021. https://distill.pub/2021/gnn-intro.