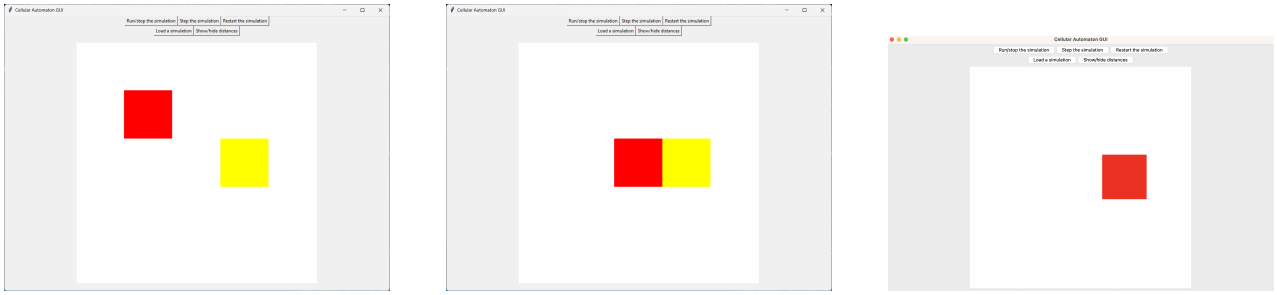
Report for exercise 1 from group A

Tasks addressed:	5
Authors:	Hirmay Sandesara (03807348) Yikun Hao (03768321) Yusuf Alptigin Gün (03796825) Fatih Özlügedik (03744764) Subodh Pokhrel (03796731)
Last compiled:	2025–05–13

The work on tasks was divided in the following way:

Hirmay Sandesara (03807348)	Task 1	100%
	Task 2	60%
	Task 3	0%
	Task 4	0%
	Task 5	30%
Yikun Hao (03768321)	Task 1	0%
	Task 2	0%
	Task 3	0%
	Task 4	100%
	Task 5	0%
Yusuf Alptigin Gün (03796825)	Task 1	0%
	Task 2	40%
	Task 3	100%
	Task 4	0%
	Task 5	0%
Fatih Özlügedik (03744764)	Task 1	0%
	Task 2	0%
	Task 3	0%
	Task 4	0%
	Task 5	10%
Subodh Pokhrel (03796731)	Task 1	0%
	Task 2	0%
	Task 3	0%
	Task 4	0%
	Task 5	60%

Report on task 1, Setting up the modeling environment



(a) Initial position (b) First step (c) Second step

Figure 1: Three images side by side with small gaps

Updated and built upon in '**src/simulation.py**' file:

- **Simulation.__init__()**: We facilitated the initialisation function of the Simulation class to initialise the environment based on the configuration that the user provides through the selection of JSON file via 'el.SimulationConfig' object. Next, we initialized the grid first with an empty cell. Following this, we populated it with obstacles, targets, and initial positions of pedestrians based on the given input configuration via a JSON file. Storing the list of pedestrian objects in the class object and computing the initial distance grid.
- **Simulation.get_grid**: Modified to return the current state of the simulation grid for visualisation.

Figure 1a is the result we see after using the specified configuration, 'configs/toy config.json'.

Report on task 2, First step of a single pedestrian

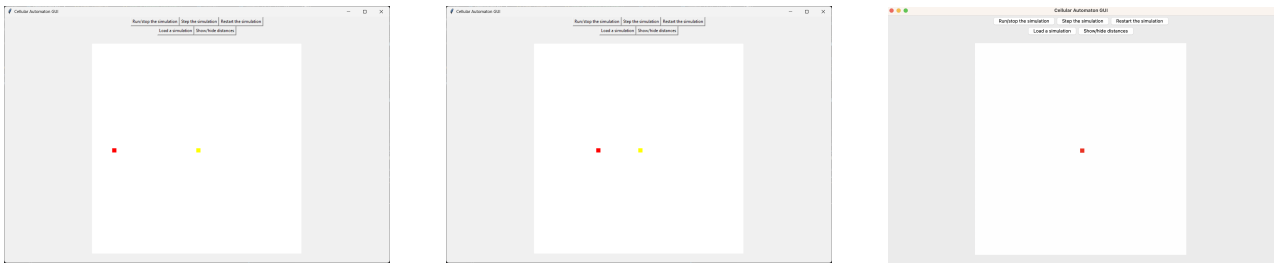
To finish task 2, we updated and built upon in the '**src/simulation.py**' file:

- **Simulation._get_neighbors()**: The `_get_neighbors()` function for the Simulation class was implemented to find and return all neighbors of a particular cell in the grid. This includes the diagonal ones, totaling 8 neighbors for a given position. The function also ensures all neighbors are within grid boundaries and returns a list containing all neighboring positions. In our implementation, we assume all these neighbors are movable to, and diagonal movements don't get blocked by obstacles.
  All of the mentioned functionalities are achieved through simple list and loop operations. Firstly, we create an empty list called neighbors. Also, two variables called `neighbor_x` and `neighbor_x`. These two variables hold the x and y coordinates of the position sent to the function as an argument. Using two for loops together with both ranging from -1 to 1 (inclusive), the code finds all possible 8 neighboring positions by adding the loop indexes at the time to the initial positions. The two small details here are that we don't add the initial positions, as this information would be present in the update function anyway, and we check if this possible neighboring position is within the bounds of the grid. Then, the function adds the new position to the neighbors list. After all positions are added, the function shuffles the list if the shuffle variable is true, and returns the neighbors list containing all neighboring positions.
- **Simulation.update()**: The update function for the Simulation class was implemented to achieve walking behaviours of the pedestrians. It ensures that updates keep occurring as long as there are pedestrians on the grid. This initial version of the update function deals with the most basic pedestrian movement, which is a straight line walk to the target with no obstacles.
  Firstly, the function computes the distance of all cells to the target by using the `_compute_distance_grid` function and gets the list of pedestrians, both of which are available in the class. These are stored in the distances and pedestrians variables, and the pedestrians list is shuffled if the perturb variable is true. Then, we start calculating the position of the pedestrians for time $\mathbf{t + 1}$, assuming the initial time is $\mathbf{t}$. This simulates one update step for the grid. An empty list of updated positions is created, and all pedestrians in the list are looped through one by one. For each of them, their current distance to the target

is stored in the `current_distance` variable, using their position in the grid to find their distance from the already computed distances. Then, this initial position and the distance of the pedestrian are assumed to be the best they can achieve, meaning if no smaller distance to the target is found in the neighboring cells, the pedestrian won't change its position. To find if this is the case, the `_get_neighbors` function is used together with a for loop to go through all neighbors of the pedestrian. For each neighbor, the distance of the neighboring cell to the target is calculated using its position, and is checked to see if it's smaller than the initial distance the pedestrian has. If so, this neighboring cell is deemed the best distance and position since the pedestrian would benefit (get closer to the target) from walking in this direction. With the loop, the function is able to find which of the 9 cells (the initial cell plus 8 neighboring cells) is the closest to the target, so that the pedestrian would walk to it. When all neighbors are checked, if the initial position isn't the best position, meaning there's a better cell the pedestrian can move to, the scenario element of the initial cell is turned to empty (E) while the scenario element of the best neighboring cell is turned to a pedestrian (P). When the scenario elements are assigned, the new position of the pedestrian, which could also be the initial position, is added to the updated positions list. Since the simulation scenario for the task requires the pedestrian to stay at the target, this version of the update function returns False, so when the target is reached, the pedestrian stays there, and the simulation continues until stopped.

Updated and built upon in '**src/generate_configs.py**' file:

- **Task_2 Simulation**: A function is added to generate the mentioned JSON configuration for task 2 with a 50x50 grid, one target at (25, 25), naive distance calculation, and one pedestrian at (5, 25). Images from the simulation are given in 2. The first image shows the initial configuration, with a pedestrian at (5, 25) and the target at (25, 25). The pedestrian then walks towards the target in a straight line, which is shown by the intermediate steps of the simulation. In the end, the pedestrian reaches the target and stays there, as the closest distance to the target cell is the target itself.



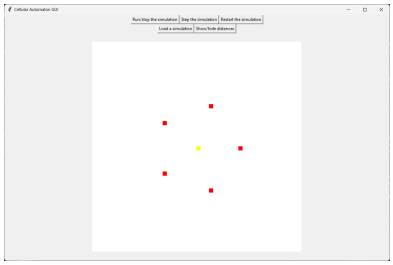
(a) Initial position (b) Tenth step (c) Twentieth step

Figure 2: Simulation Steps of Task 2 Scenario

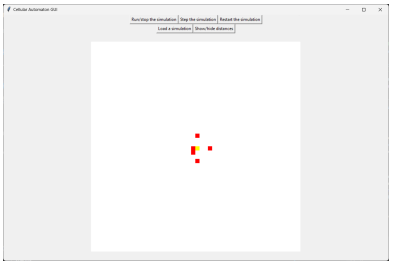
Report on task 3, Interaction of pedestrians

- **Pedestrian Avoidance**: To implement a rudimentary pedestrian avoidance, simple changes to the update function suffice to achieve correct simulation results. The first thing to do is to make sure the pedestrians don't step on other pedestrians (P) cells. Since the update function was designed to loop through all pedestrians and their neighbors, we can add simple if-else statements to check if a neighbor is a cell the pedestrian would move to. At the start of the loop for checking the neighbors of a pedestrian, we also check if the neighboring cell is a pedestrian. If so, then the action of checking the distance of the cell to the target compared to the initial cell isn't done. Simply, the code acts as if that cell doesn't exist in the calculation, making it impossible for a pedestrian to step onto a cell that isn't empty or the target. As a side note, even though the pedestrian would technically be able to step onto an obstacle with this setup, since there aren't any obstacles in the scenarios up to this point, this doesn't create any problems. A more complex avoidance is implemented in part 4 for complex scenarios.
- **Simulation Scenario**: For the given simulation scenario, we have to make the targets absorb the pedestrian. This is done by checking whenever a pedestrian moves to a neighboring cell (i.e., the initial cell has more distance to the target than a neighboring cell), if the moved cell is the target. If it's so, we don't change the scenario elements for that cell and don't add the pedestrian to the list of updated positions.

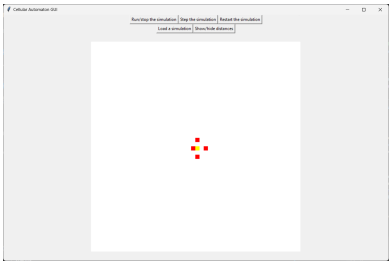
This makes pedestrians "leave" the scenario when absorption is true. Otherwise, the update function logic works as before. As a final change, for every update step, we return the length of the updated positions list. This ends when there are no positions to be updated, i.e., when all pedestrians have reached the target.
To simulate the scenario, the initial orientation shown in 3.a was chosen. In a 50x50 grid, the pedestrians are placed in a circular formation far away from the target. These pedestrians are roughly equally far away from the target in Euclidean distance. Simulating the scenario, we see that the pedestrians reach the target in similar times, but in different steps of the simulation. One pedestrian reached the target in the eighth step, another in the ninth, and the other 3 in the tenth step. This is because even though the distance to the target in Euclidean distance is the same for all pedestrians, the cellular automata design doesn't allow all pedestrians to walk in a similar pattern. Pedestrians more diagonally placed to the target reach it faster since they've the ability to walk diagonally; while pedestrians placed in a more straighter direction towards the target reach it slower since they have fewer diagonal moving patterns. Moving in a diagonal pattern lets pedestrians traverse a greater distance, even though the speed of all pedestrians is the same. To counter this issue, we need to implement speed adjustments. Also, in the seventh step of the simulation, the pedestrian avoidance feature can be seen in action. When two pedestrians reaching the target from opposite diagonal directions have to cross the same cell, the pedestrian that is first in the list occupies the cell first, making it so the other pedestrian has to move to another cell.



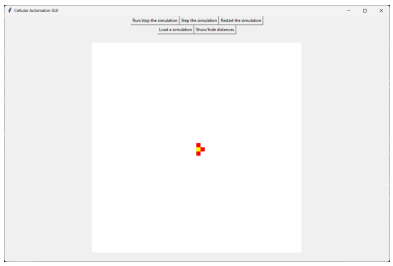
(a) Initial orientation



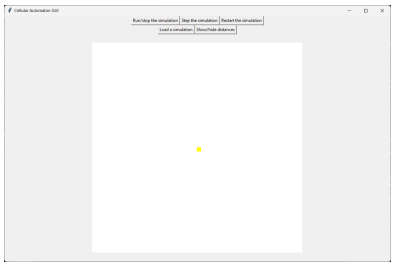
(b) Seventh step



(c) Eighth step



(d) Ninth step

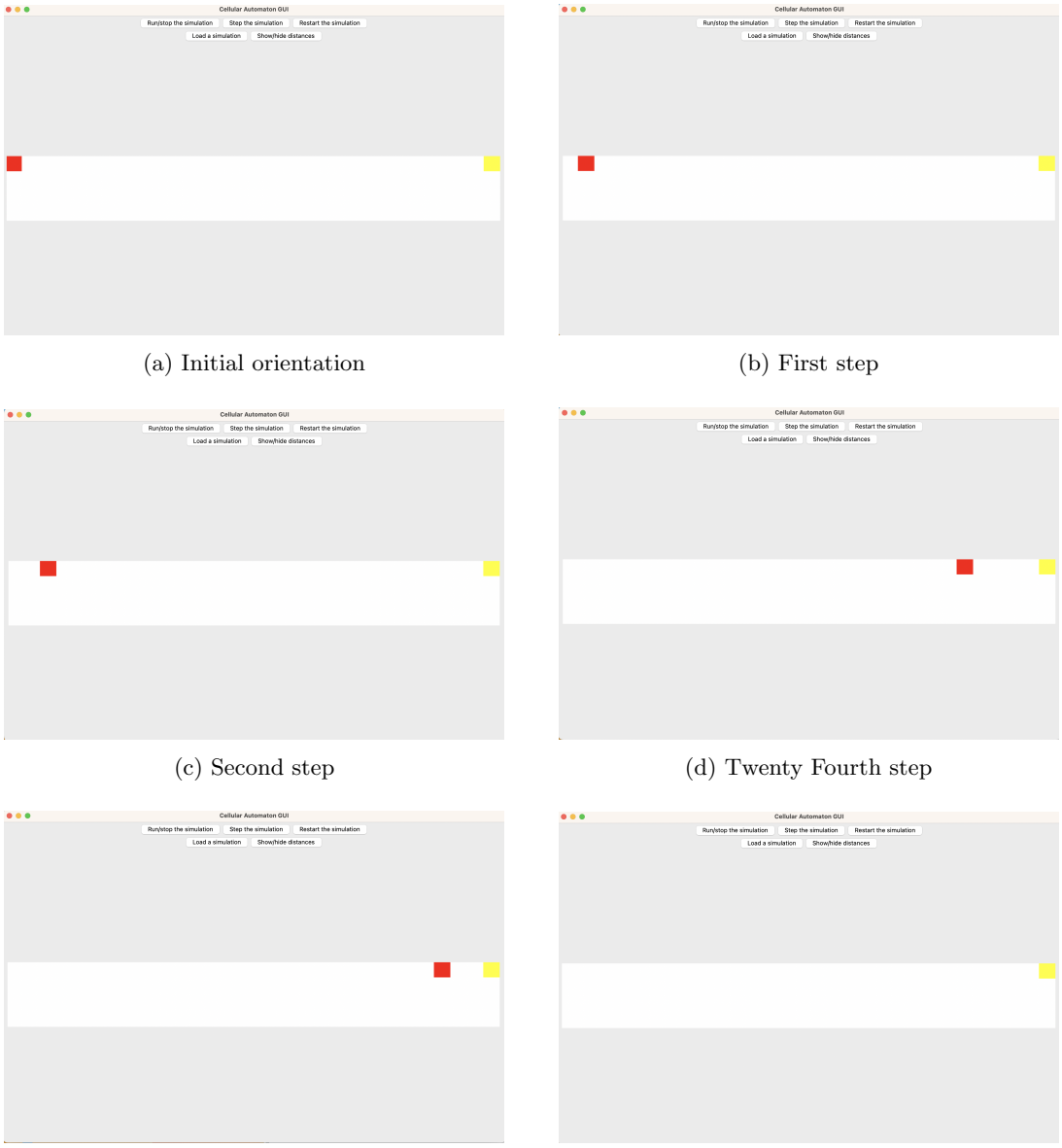


(e) Twelfth step

Figure 3: Simulation Steps of Task 3 Scenario

- **Speed adjustments**: For speed adjustments, we need to make sure that pedestrians can travel according to their speed. But in the automata, we can just make the pedestrians go cell by cell. So, a mechanism to keep track of pedestrian movements is necessary. We've to make sure that when a pedestrian has speed **X** and the distance to the next cell is **1.5X**, the pedestrian shouldn't move to the next cell, but tracking should be done for the supposed **X** distance the pedestrian travelled. We do this by adding an attribute to the class called "progress". All pedestrians start with a progress of 0, and in each update step, they add their speed to their progress, simulating their movement in the grid. Now, when the best position for a pedestrian to move to is found in the update function, instead of immediately calculating movements, we check if the pedestrian can actually move into that cell; i.e., if they've made enough movement progress for a movement in the cellular automata. When the distance to the next best cell is smaller than the pedestrians' progress, the movement is initiated, since the pedestrian has traversed enough distance to go to that cell. Now, also, when a movement occurs, we have to check if multiple movements can occur. Because in cases where the pedestrian's speed is high, meaning it can traverse multiple cells at the same time, we don't want to update the pedestrians' location by just one cell, but by the number of cells it can

move. To achieve this, a while loop is used. In the loop, by checking if the pedestrians' progress is greater than the traversable distance, we achieve multiple cell movement.
Here, a helper function called `find_next_neighbor` is needed. This is because, going forward in the automata, we have to keep checking neighboring cells to calculate the possible forward trajectories of the pedestrians. This function takes all the information at the time, containing pedestrians, occupied cells, distances of the grid, best possible distance at the time, and the best possible position at the time, to return the neighbor with the smallest distance to the target and its position. In the loop, for each iteration, this function is used to calculate the next possible trajectory. Then, the distances between the cells in the trajectory are deducted from the pedestrians' progress. Only when the pedestrian doesn't have enough progress for another forward movement, the loop is stopped, and the current best position and the distances are assigned to the necessary variables.
The remainder of the update function works the same, with only a small difference of using the function `find_next_neighbor` for also checking the first neighboring cells, making the code more modular. Now, with functionalities of multiple cell movement or no movement depending on the speed of the pedestrian, we can simulate pedestrian movements more accurately. Some example scenarios are given below:



(a) Initial orientation

(b) First step

(c) Second step

(d) Twenty Fourth step

(e) Twenty Fifth Step

(f) Twenty Eight Step

Figure 4: Simulation Steps of Scenario with Pedestrian Speed 1.04 and Straight line Movement

- **Scenario 1**: In this scenario, we've a pedestrian at (0,0) with speed 1.04 and a target at (29,0), as shown in the initial configuration of 4. The pedestrian has to walk in a straight line to the target, but this is with more than an initial speed of 1, 1.04. Here, we see the effects of the "progress" mechanism for speed. In the first couple of steps of the scenario, the pedestrian walks one cell at a time. But in the background, it's extra speed of 0.04 is accumulated in the class, keeping track to know when it has to move more than one cell. In the twenty-fifth step, there's enough accumulation of the pedestrian progress that it moves 2 cells at a time. This allows it to reach the target destination in a total of 28 steps, one less than that of 29 if it only had a speed of 1. This mechanism can intuitively be thought of as the pedestrian walking from cell to cell, but with each step, it's going a bit further in each cell. So, for example, a pedestrian starting its walk at the start of the initial cell, with speed 1.04, goes to 0.04 cm (or given distance measurement) farther in the next cell, instead of going from the start of the initial cell to the start of the next cell.

(a) Initial orientation (b) First step (c) Second step

(d) Third step (e) Fifth Step (f) Sixth Step

Figure 5: Simulation Steps of Scenario with Pedestrian Speed 1 and diagonal movement

- **Scenario 2**: In this scenario, we've a pedestrian at (0,0) with speed 1 and a target at (4,4), as shown in the initial configuration of 5. This time, the pedestrian will walk in a diagonal line, since the maximum distance it can traverse in one cell movement is $\sqrt{2}$. But as it can be seen in the first step of the simulation, the pedestrian doesn't move, since with speed 1, it can't traverse $\sqrt{2} \sim 1.41$ distance. It has to accumulate progress for 2 update iterations, and then it'll move one step diagonally. In total, since the distance from the initial cell to the target cell is around ~ 5.65, with speed 1, it takes the pedestrian 6 iterations to reach it. This can also be seen from the images.

Report on task 4, Obstacle avoidance

- **Obstacle avoidance**: Until now, all pedestrians could move freely to the target without obstacles. The next step is obstacle implementation. To fulfill this requirement, pedestrians should not be able to move to obstacle grid. This is implemented via the shown function 1 in `Simulation.update`.

Listing 1: Pedestrian Obstacle Avoidance

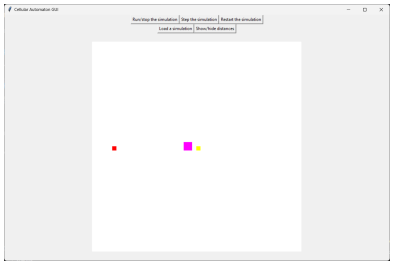
```
# Avoid non-empty cells (except targets)
if self.grid[neighbor.x, neighbor.y] not in [el.ScenarioElement.empty, el.
    ScenarioElement.target]:
     continue
```

Another requirement is that the obstacles are not allowed to overwrite targets. This is implemented via the shown function 2 in `Simulation.__init__`.

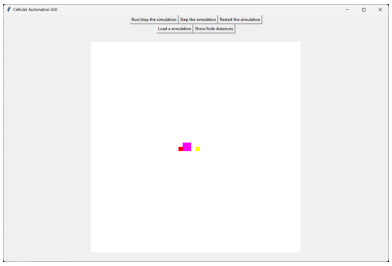
Listing 2: Overwritting Avoidance

```
for position_obstacle in config.obstacles:
    if self.grid[position_obstacle.x, position_obstacle.y] != el.ScenarioElement
        .target:
        self.grid[position_obstacle.x, position_obstacle.y] = el.ScenarioElement
            .obstacle
    continue
```

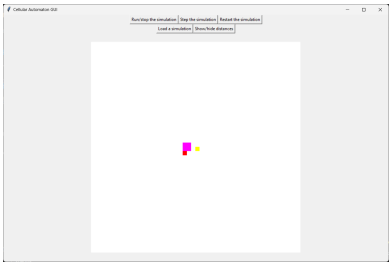
The testing scenario and results are shown below as 6. In 50x50 grid, the target is placed exactly to the middle and the pedestrian is placed 20 grids to the left of the target grid. There is a squared obstacle placed near to the target, and the pedestrian has to circumvent the obstacle to reach the target. The goal could be achieved via the naive distance algorithm. From the figures, it could be observed that the pedestrian reached the obstacle at the 17^{th} step, then it takes 3 steps to circumvent the obstacle. Finally, the pedestrian manages to reach the target at the 21^{th} step.



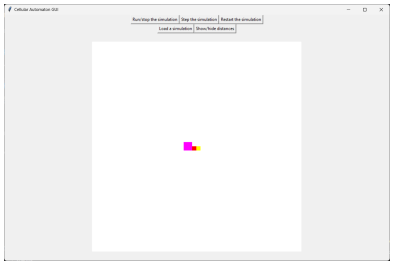
(a) Initial position



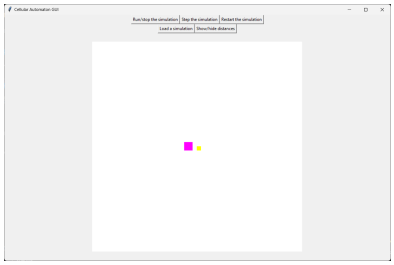
(b) 17^{th} step



(c) 18^{th} step



(d) 20^{th} step



(e) 21^{th} step

Figure 6: Simple Obstacle Avoidance with Naive Distance Algorithm

- **Dijkstra algorithm**: To compute the shortest distance between each grid cell and its closest target cell, we implemented the Dijkstra function to iteratively propagate distance values throughout the grid and flood the cells with distance values. The core idea of the Dijkstra algorithm is, starting from target cells, by always selecting unvisited neighbors, interactively expanding the shortest known-path from the target to all other nodes. The implementation is Simulation._compute_dijkstra_distance_grid function.
  Simulation._compute_dijkstra_distance_grid: To fulfill the need to expand only the shortest known-path, heapq library is introduced. This library in Python provides an implementation of heap queue algorithm, specifically min-heap, ensuring the popped out value is always the smallest. The algorithm starts by initializing all grid cells to distance infinity, then assigns the value 0 to all target cells. Then the main loop starts, all target cells are added to priority queue, then iteratively pop out all cells and expand their neighbors. Before expanding neighbors, check if current distance is smallest entry of the cell grid. This is to avoid redundant processing repeated and outdated entries, ensuring the efficiency and correctness of the algorithm. By expanding neighbors, we define orthogonal movement(up, down, left, right) with step distance 1 and diagonal movements with step distance $\sqrt{2}$. Skip the cell if it is an obstacle, so that the distances to obstacles remain infinity. Notice that keeps the expanded value only if the value is smaller than neighbor's current value. The algorithm ends when all cells in priority queue are popped out and returns a 2D distance array, containing the shortest distances from the grid cell to nearest target.
  Visualization of distance to the target is shown as 7. In this map, warmer colors indicate shorter distances

to the target. Upon visualizing the resulting distance map, we noticed that the color grids radiated from target cells are not in perfect circles. This is because movement is restricted to discrete orthogonal and diagonal steps of one grid unit. Therefore, only target cells aligned along the eight primary directions — namely north, northeast, east, southeast, south, southwest, west, and northwest — can be reached via a straight line. The remaining points require combined directional movements, resulting in non-linear paths.

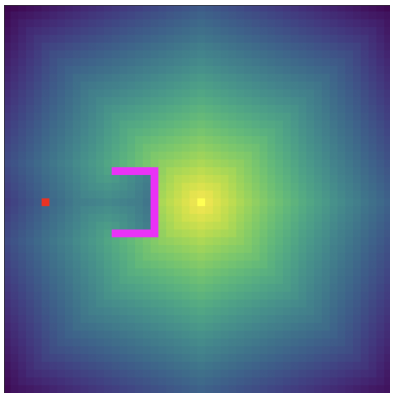
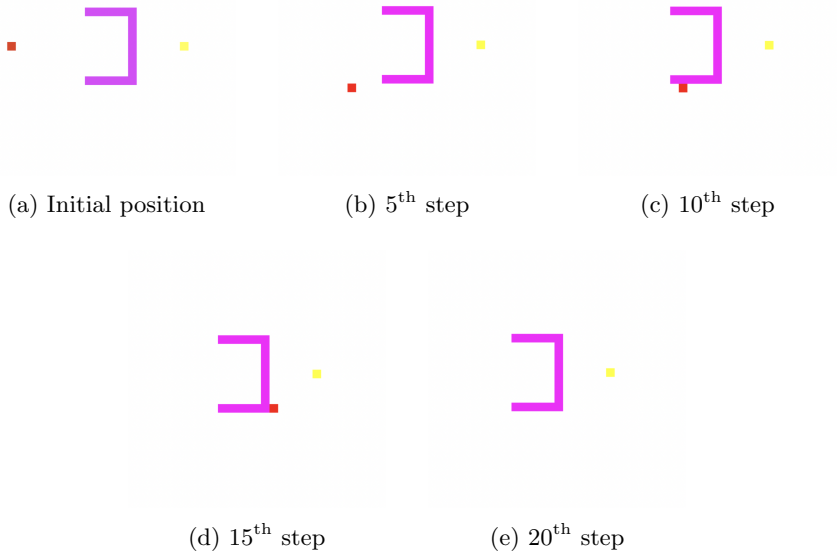


Figure 7: Task 4 distance color map (chicken). The color map is not in perfect circle.

- **Simulation Scenario - chicken test**: For given scenario of chicken test, we test if a pedestrian can avoid the U-shape obstacle and reach the target. In a 50x50 grid, the target is placed exactly in the middle and the pedestrian is placed 20 grids to the left of the target, separated by an U-shaped obstacle facing the pedestrian. The five figures below (8) shows the simulation result, which ends in 20 steps. Step difference of neighbouring figures is 5. As shown in the test, the pedestrian circumvented the obstacle and successfully reached the target location. The result indicates the well implementation of Dijkstra algorithm. As the cells within U-shape obstacle is flooded relatively later and assigned higher distance values, the pedestrian navigate around the obstacle instead of get trapped inside, a scenario that might occur with greedy algorithm such as naive distance algorithm.



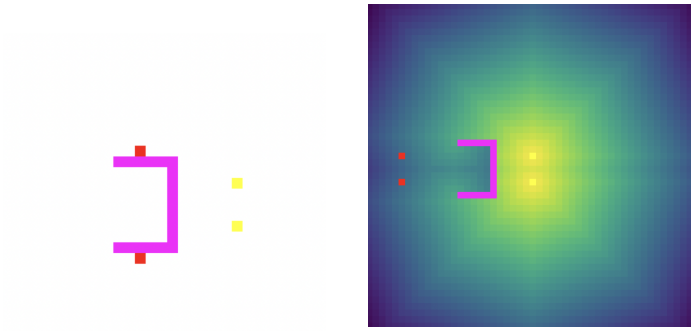
(a) Initial position (b) 5^{th} step (c) 10^{th} step

(d) 15^{th} step (e) 20^{th} step

Figure 8: Simulation Steps of Task 4 Chicken Scenario

- **Simulation Scenario - chicken test - variant**: To briefly test the Dijkstra algorithm in more complex

scenarios, we introduce a variant of the chicken test. The difference lies within, that there are two targets and two pedestrians. Those two sets of pedestrians and targets have a vertical grid difference of 4 grids. The results are shown as below.
Within 9a, two pedestrians (red) and two targets (yellow) are placed with a vertical offset. A U-shaped obstacle forces the pedestrians to find separate paths toward their respective targets. 9b is a distance color map with binary targets. The results indicate a successful implementation of Dijkstra algorithm with multiple pedestrians and multiple targets.

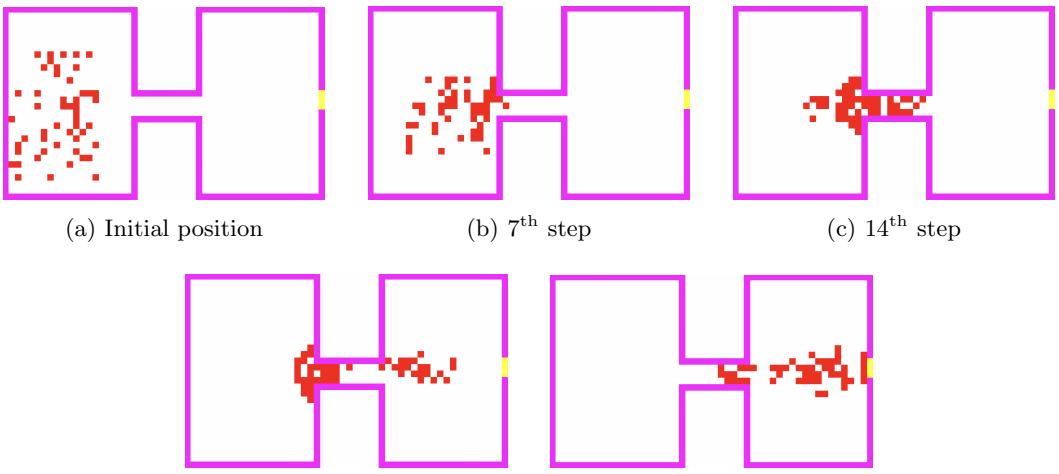


(a) 17^{th} step of the test

(b) Task 4 distance color map (chicken - variant)

Figure 9: Results of chicken test - variant

- **Simulation Scenario - bottleneck**: The bottleneck test is modeled after RiMEA Guideline Test 12 "Effect of Bottlenecks", though in a simplified form. The simulation space is a 50×30 grid, where two separate rooms are connected by a central corridor. On the far right of the space, there is a 3-cell-wide exit embedded in the wall. A total of 50 pedestrians are randomly initialized in the left room and must reach the exit by passing through the corridor.
  As shown in 10 below, all pedestrians are able to reach the target after 49 steps. As the simulation progresses, congestion forms near the entrance of the corridor, while the area on the right side remains mostly clear. It could be observed that the bottle creates a delay in pedestrian flow, and congestion gets more and more severe as the pedestrian population grows. Enlarging bottleneck width might mitigate such phenomena.



(a) Initial position (b) 7^{th} step (c) 14^{th} step

(d) 21^{th} step (e) 28^{th} step

Figure 10: Simulation Steps of Task 4 Bottleneck Scenario

Report on task 5, RIMEA Tests

RiMEA-Guidelines provide standards for evacuation analyses of complex buildings and ensure that fundamental questions about evaluation analysis are answered[1]. The simulation-based analysis determines evacuation time, pedestrian flow, and congestion. Cellular automata, on the other hand, can be used to investigate models in natural science, mathematics, computer science,[2], and as seen in the above section, even model and simulate human crowds. As such, RiMEA guidelines can be used to check how the model behaves in predetermined test cases. In this section, we evaluate our model in four different RiMEA test scenarios:

In this section, we discuss the findings of the implementation of four different test cases to evaluate the performance of our modelled cellular automaton. These tests follow the specifications in the RiMEA guidelines:

Test 1 The main objective of Test 1 is to verify that a pedestrian in a 2m wide and 40 m long corridor with a defined walking speed will cover the distance in a certain period of time. To do this, we create a scenario (see Figure 11 with the exact conditions to simulate the behaviour of an automaton.

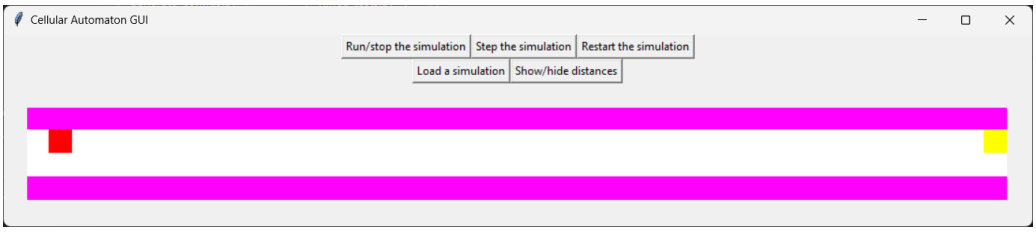


Figure 11: Test setup with pedestrian (red) and target (yellow)

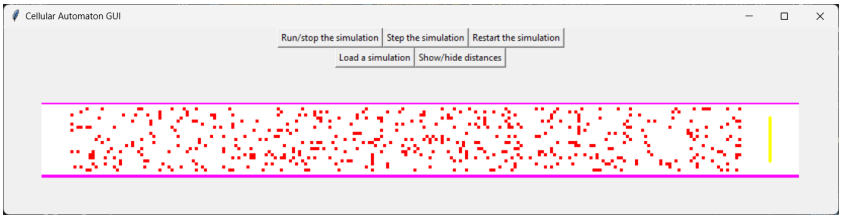
The expected travel time for the pedestrian with a typical speed (between 2.5 km\h and 5.1 km \h) is in between 26 to 34 seconds. In our case, a pedestrian with a speed 1.33 m\h completed the walk in a total of 31 seconds (see Figure 12); which is within the expected range.

```
Running RiMEA Test 1: Straight Line Movement
Successfully loaded configuration from configs/task_5_rimea_test1.json
Successfully converted config dictionary to object with attributes
Test completed in 31 steps
PASSED: Travel time within expected range (26-34 steps)
```

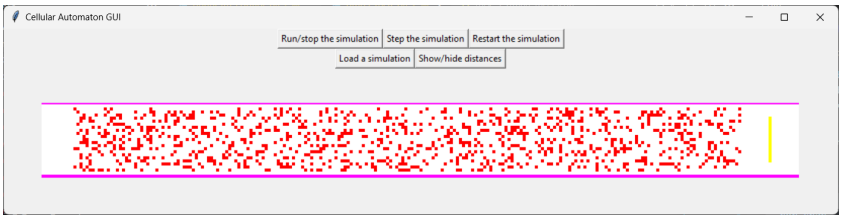
Figure 12: Result of Test 1

Although the guidelines include a pre-movement (reaction) time as a model parameter, we omit it here without loss of validity. This can be explained by the fact that Test 1 has a 12-s timing margin, so an extra 1-s reaction time has a negligible effect. Similarly, in Test 4, the measurement begins recording only after a 10-second transient period, by which point any reaction-time effects have already dissipated. Meanwhile, in Test 6, there is no explicit time dependence as we only examine the success of the model in the corners. In Test 7 scenarios, we consider the distribution of the results rather than the absolute values, and therefore, the reaction time does not play a significant role. This is because the reaction time would affect the arrival time of each demographic group in the same way. On the other hand, as pedestrians move to the adjacent cells in every subsequent step, their speed is inherently connected to the grid size; therefore, we have to model the body size.

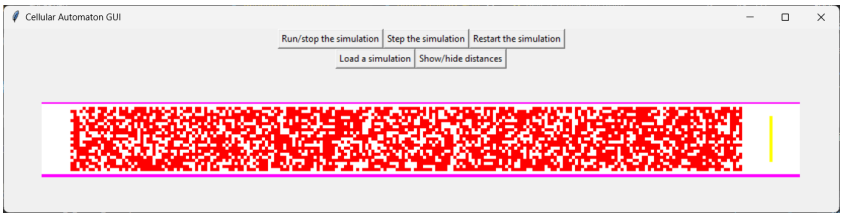
Test 4 This test examines the speed of pedestrian flow at a given measurement point for different crowd densities. We simulated the system for density values ranging between 0.5 P/m2 and 6 P/m2. In order to calculate the speed of the pedestrians at a 'stable' level, the measurement only starts 10 seconds after the start of the simulation. The setup of the simulation with various densities can be seen in Figure **??**.



(a) Density = 0.5 ped/m



(b) Density = 1 ped/m



(c) Density = 3 ped/m

Three measurement areas are defined in the corridor where, as the pedestrian passes, their speed is recorded. In the end, the average of this value is taken as the speed of the moving crowd. The main objective of this test is to establish the basic relationship between density and pedestrian speed as it exists in the measurement area. The expected result is a decreasing speed with the increase in crowd density with the value tending towards zero[3]. As seen in Figure 14, the speed does indeed go down towards zero with increasing density. However, the speed values are much slower than expected (and seen in the GUI). The cause of this is thought to be an error in the measurement in the simulation.

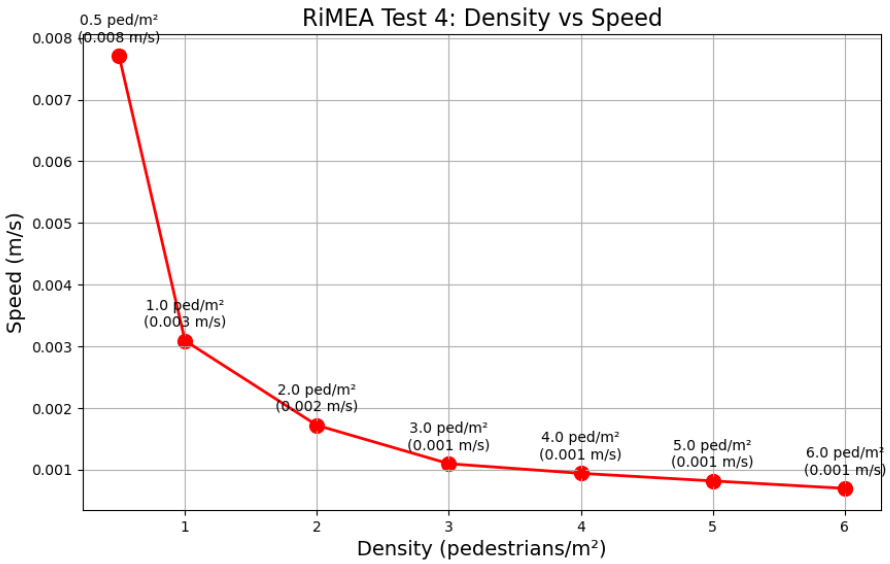


Figure 14: Result of Test 4 (Density vs Speed of pedestrians)

Test 6 In this test, we check if the pedestrian in the simulation could cope with the obstacles. In particular, we test whether the simulated pedestrian could successfully turn the corner. For this purpose, we construct a horizontal corridor with a sharp vertical turn as seen in Figure 15. The pedestrian was placed 4m before the start of the connection to the perpendicular segment of the corridor and simulated to move towards the target at the top of the vertical corridor. During the simulation, the number of successful turns is calculated. A turn is considered successful if the pedestrian does not make a forbidden move, i.e., crossing the boundaries of the road defining the corner.

Figure 15: Result of Test 4 (Density vs Speed of pedestrians)

In the implementation, Bresenham's line algorithm is used to check for violations. The line of movement from (previous_x, previous_y) to (current_x, current_y) is looked at to see if it crossed any obstacle, and if so, it is considered to be a wall violation. Using this method, the result seen in Figure 16 is achieved. It shows that the five pedestrians moved into walls at some point. This is an obstacle avoidance issue.

```
Simulation completed in 26 steps
Pedestrians that reached the target: 20 out of 20
Unique positions used in corner area: 25
Wall violations detected: 5
FAILED: Issues detected in corner navigation
- 5 potential wall violations
```

Figure 16: Result of Test 4 (Density vs Speed of pedestrians)

Task 7 In this task, we measure the time required for pedestrians of different age groups to reach a target location. These pedestrians have different walking speeds. As seen in Figure 17, they first line up vertically and move horizontally to a fixed target as the simulation starts. The ages and speeds are loaded from file *rimea_7_speeds*. The comparison of age vs walking speed can be seen in Figure 18.

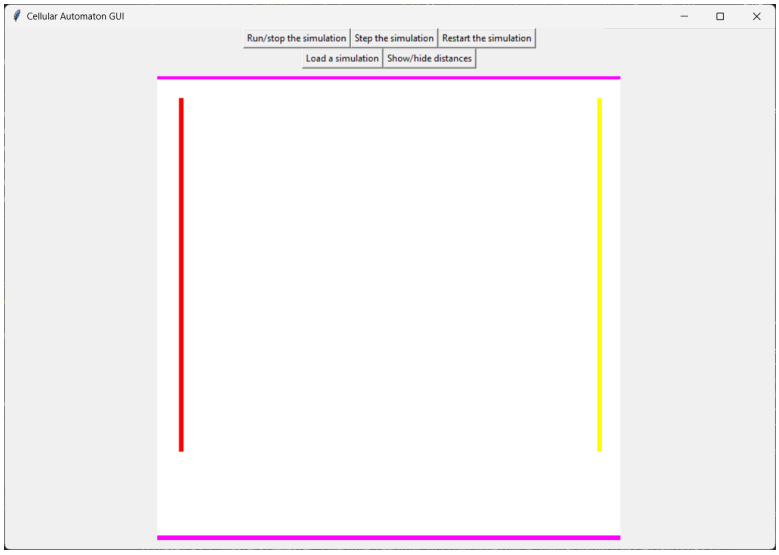


Figure 17: Initial position of pedestrians and targets

Figure 18: Age vs speed

Based on Figure 18, we expect some pedestrians to reach the target faster and some slower. The young and the old ones are expected to take the most time, resulting in a graph inverse to the one seen. This is indeed true and can be seen in Figure 19. Although Figure 19 is not an exact mirror of 18, it is clearly visible that the relation between speed and time taken with age is coherent.

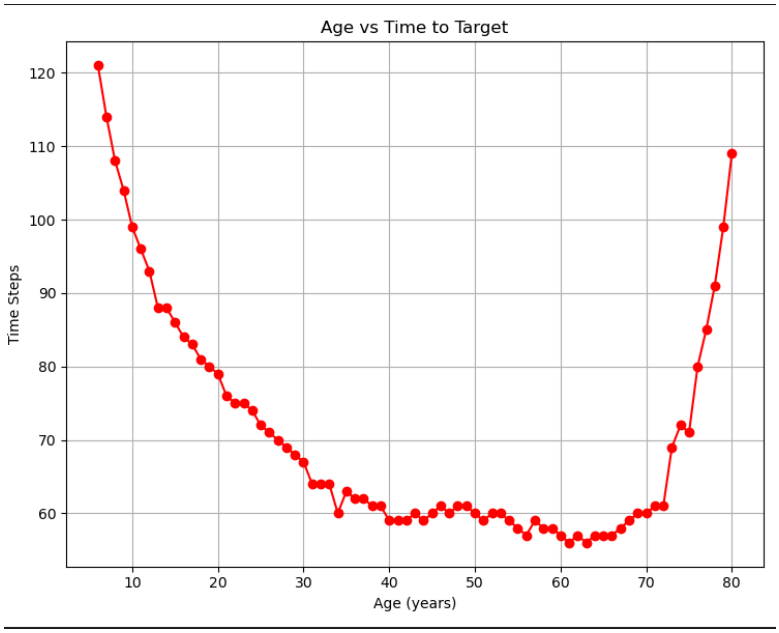


Figure 19: Age vs time taken to reach target

References

[1] Christian Rogsch, Hubert Klüpfel, Rainer Könnecke, and Andreas Winkens. Rimea: A way to define a standard for evacuation calculations. In Ulrich Weidmann, Uwe Kirsch, and Michael Schreckenberg, editors, *Pedestrian and Evacuation Dynamics 2012*, pages 455–467, Cham, 2014. Springer International Publishing.

[2] Tommaso Toffoli and Norman Margolus. *Cellular Automata Machines: A New Environment for Modeling*. 04 1987.

[3] L. D. Vanumu, K. Ramachandra Rao, and G. Tiwari. Fundamental diagrams of pedestrian flow characteristics: A review. *European Transport Research Review*, 9:49, 2017.