Article

A cellular network model for crowd and pedestrian simulations

Chiara Piccolroaz [1], Dominik Childs [2]

[2] Hochschule München - University of Applied Sciences; dchilds@hm.edu

Version January 29, 2025 Submitted to Modeling and Simulation Assignment

Keywords: cellular networks, pedistrian streams, modeling, simulation, optimization

In this study, we developed a cellular network model to simulate pedestrian movements. The two central aspects of our model are the utility and the priority functions, as they respectively determine where and when each agent will move towards its target through a space with other obstacles and agents. The general utility is calculated using the euclidean distance that implements a correction mechanisms for diagnoal fields. Dynamic influences that are supposed to prevent the forming of crowds and make it less attractive to move close to walls are added to the utility seperatly. The priority function is calculated based on the predicted time each agent needs to move to the next best field. In order to evaluate the proposed model, it was tested it against four scenarios taken from the RiMEA e.V. guidelines. These tests include observing agents while moving without being obstructed, navigating around a u-shaped obstacle, evacuating a room with two and four doors and walking along a corridor. The results indicate that the model is functioning correctly, with outcomes mostly aligning with expectations.

1. Introduction

The term cellular automaton (CA) refers to a mathematical model used in various mathematical and scientific fields to represent spatially discrete dynamic systems. More precisely, it represents changes within a space structured as a grid of cells over discrete time steps. When these changes are cosidered sequentially we refer to cellular networks (CN). In computer science, cellular automata are often utilized for tasks such as simulating parallel computing, designing algorithms, and enhancing cryptographic methods. In physics and biology, they are used to model physical or natural processes such as fluid dynamics, phase transitions, disease spread, population growth, and the formation of biological patterns.
In this study, we focus on cellular networks and pedestrian streams. More concrete, we aim to develop a cellular network model that simulates the movement characteristics of pedestrians as realistically and correctly as possible.

The primary objective is to create a model capable of approximating the behaviors of pedestrian streams. This includes implementing an utility and priority function that determinates the next best position and time for each agent movement. We calculate this function based on Euclidean distance 3.1 and use the Dijkstra algorithm to find the shortest path 3.1.2, adjusting utility values according to the steps effort caused by field orientation 3.1.3 and the avoidance behavior caused by the proximity to other agents or obstacles 3.1.4.

The secondary objective is to assess whether the model is correct. This assessment includes utilizing five test cases from those proposed by RiMEA e.V. and analyzing key metrics such as time, speed, distance, density and flow of individual agents or the entire crowd. The model was tested against four scenarios outlined in the RiMEA e.V. guidelines. These scenarios include navigating around a U-shaped obstacle, walking with a clear line of sight, evacuating a room using two and four doors, and traversing a corridor to reach a target.

With an event-driven implementation of a neural network model, this study contributes to simulating the movements of pedestrian streams in different scenarios. Such simulations are generally useful for designing buildings with efficient evacuation routes, planning large events, providing optimal infrastructure support. In this study, we will use our model to simulate a real-worlds experiment, such as the one presented in section 3.5.5.

The results indicate that the model is functioning correctly, with outcomes largely aligning with expectations. Yet, the "chicken test" reveals a denser crowd around the enclosure edges, likely due to the Dijkstra algorithm prioritizing the optimal path and creating a queue as individuals wait to occupy the most advantageous positions. Similarly, the RiMEA 4 test provides results that diverge from those suggested by the Weidmann diagram.

2. Data and Methods

In order to enhance the understanding and replicability of this study, this section presents the main characteristics and assumptions underlying our cellular network model (2.1). Additionally, it describes the data (2.2) and methods (2.3) used to simulate pedestrian streams.

2.1. Characteristics and Assumptions of the Cellular Network Model

As mentioned in the introduction, our study focuses on cellular networks. These networks have three main characteristics: the cells dividing the grid field, the states these cells can present, and the rules determining the agents' movements. These characteristics are summarized in Figure 1, which shows our self-implemented visualization tool just before a simulation is run.

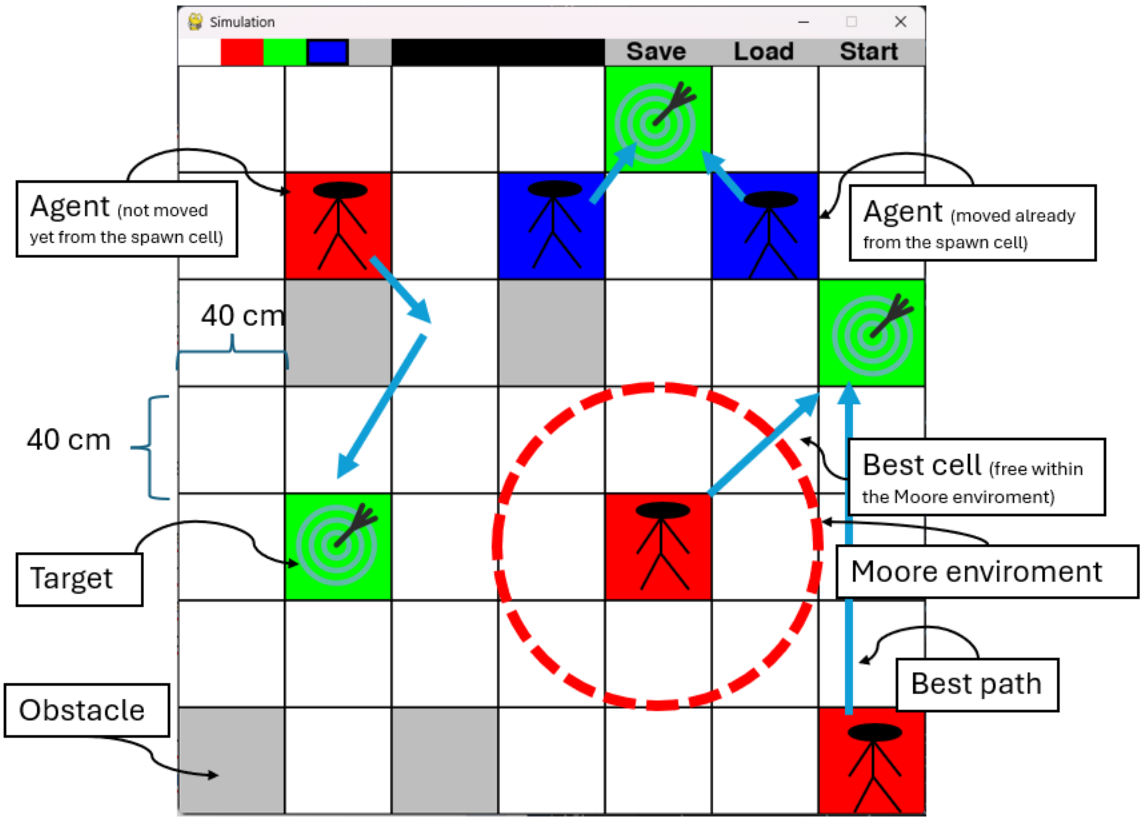

Simulation
Save
Load
Start
Agent (not moved yet from the spawn cell)
Agent (moved already from the spawn cell)
40 cm
40 cm
Best cell (free within the Moore enviroment)
Target
Moore enviroment
Best path
Obstacle

Figure 1. Characteristics and assumptions of the cellular network model. In red and blue the agents, in grey the obstacles, and in green the targets

We decided to represent the environment as a 2D grid in which each cell has a size of 40 cm, as, on average, individuals cover approximately 40 cm per step as mentioned in our lecture.

We decided that each cell can either be free or occupied. Specifically, we differentiate the types of occupation as follows: target (Fieldtype.TARGET), agent (Fieldtype.SPAWN and Fieldtype.OCCUPIED), or obstacle (Fieldtype.OBSTACLES).

In our model, there must be at least one target and one agent in the grid. There can be none, one, or several obstacles. The agents spawn in specific cells and attempt to move towards their targets with each time step. Once an agent reaches its target, it disappears. Each agent has a free-flow speed representing how quickly they move when the path is unobstructed. This attribute is addressed in more detail in Section 2.2.2.

We have established the following key rules to govern the direction and time at which each agent moves towards its target.

First, each agent moves to the nearest target on the grid.

Second, the agents move sequentially to their next cell at each time step. The time step is determined by our priority function described in Section 3.2, which takes into account the speed and desired field of each agent within the grid.

Third, the next cell must be a free cell within the Moore enviroment and with an hiher utility value. Free means that the cell is not currently occupied by other agents or obstacles. The Moore environment refers to the directly surrounding cells of an agent (8 cells if the agent is not at the edges or corners of the grid). The utility value is determined by our utility function described in Section 3. We implemented this using Euclidean distance and Dijkstra's algorithm. In order to mimic the real world, we decided that the agents should avoid obstacles and other agents on their path. We therefore corrected the utility values to take into account factors such as the influence of the current position relative to the desired cell and the proximity of other agents or obstacles.

2.2. Data for Simulating the Pedistrian Streams in the Cellular Network Model

In order to simulate pedestrian streams using our cellular network model, we need information about the positions of agents, targets, obstacles, and the so-called free-flow speed of agents. Since this information is not provided, we implemented two functions that randomly generate these data.

2.2.1. Element positions: Random selection of free cells & Runs Test

The function *draw_random_position_from_free_ones* returns a randomly chosen cell from the available free ones for the specified number of agents, targets, and obstacles. These cells will represent the positions of these elements at the beginning of the simulation.

To check for randomness in the data, we conducted a Runs test. This is a nonparametric statistical test that counts the number of runs and compares this to the expected number of runs, calculating a z-score to measure the deviation. If the z-score is beyond a critical value, it suggests non-randomness in the data. As shown in 2.2.1, the z-score is greater than the critical value, preventing us from rejecting the null hypothesis.

```
Runs Test:
Z-score: 0.603053454834441
Critical value: 1,96
P-value: 0.05
```

Listing 1: Results of the Runs Test for the random position of the grid elements

2.2.2. Free-flow speeds: Normal distribution & Kolmogorov–Smirnov Test

The function *draw_normal_distributed_random_numbers* returns an array with normally distributed random numbers for the specified number of agents. These numbers will represent the free-flow speed of the agents at the beginning of the simulation. Since the agents' speeds follow a normal distribution, we established the mean at 1.34 meters per second and the standard deviation at 0.26 meters per second, aligning with values recognized by the modeling and simulation community. Regarding the range for generated values, we set the minimum free-flow speed to 0.5 meters per second and the maximum to 12 meters per second, based on the peak performance of the fastest human.

To assess the fit of the generated free-flow speeds against a hypothesized normal distribution, we conducted the Kolmogorov-Smirnov test. This is a non-parametric test for the equality of continuous one-dimensional probability distributions that can be used to determine whether a sample comes from a given reference probability distribution or whether two samples come from the same distribution [1].

The function uses a sample of normally distributed free-flow speeds and a significance level α of 5%. This represents the risk of Type I error, or the probability of rejecting the null hypothesis when it is true. The Kolmogorov-Smirnov test on trimmed data gives a p-value of approximately 0.104, which is greater than the significance level, indicating that the data likely follows the hypothesized distribution (see 2.2.2). The relatively low p-value results from trimming values between 0.5 and 12. Without trimming, the p-value significantly increases from 0.105 to 0.807 (see 2.2.2).

```
Kolmogorov-Smirnov Test:
Statistic: 0.001214382815212578
P-value: 0.10463708581011633
Average of all values 1.3408679606058023
```

Listing 2: Results of the Kolmogorov-Smirnov Test for the trimmed normal distribution

```
Kolmogorov-Smirnov Test:
Statistic: 0.0006400527271357426
P-value: 0.8069909383548901
Average of all values 1.3400618604178296
```

Listing 3: Results of the Kolmogorov-Smirnov Test for the untrimmed normal distribution

The average of generated values around 1.34 as well as the free-flow speeds that are plotted in figure 2 confirm the adherence to the normal distribution for both trimmed and untrimmed data.

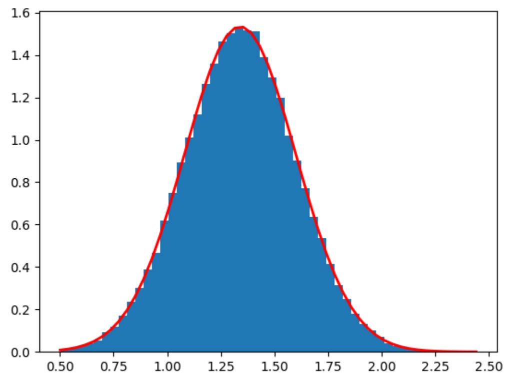


Figure 2. The Probability Density Function (PDF) of the Normal Distributions averaging at about 1.34

2.3. Methods for Simulating the Pedistrian Streams in the Cellular Network Model

The source code of our study can be found at the following GitLab repository. This was implemented entirely in the Python programming language due to its good adaptability and interpretability. These advantages allowed us to seamlessly implement a Model-View-Controller (MVC) architectural pattern and integrate useful packages for analysing and visualising our simulations.

MVC architecture is a widely used design pattern that divides an application into three interconnected components, namely *Model*, *View*, and *Controller* (Figure 3). The *Model* represents the data and business logic, managing data, responding to queries, and updating the controller on any changes. The *View* presents data to the user and handles user interactions, sending input to the controller for processing. The *Controller* acts as an intermediary between the model and view, processing user input, updating the model as necessary, and refreshing the view. This separation of concerns promotes modularity, making the software easier to develop, test, and maintain.

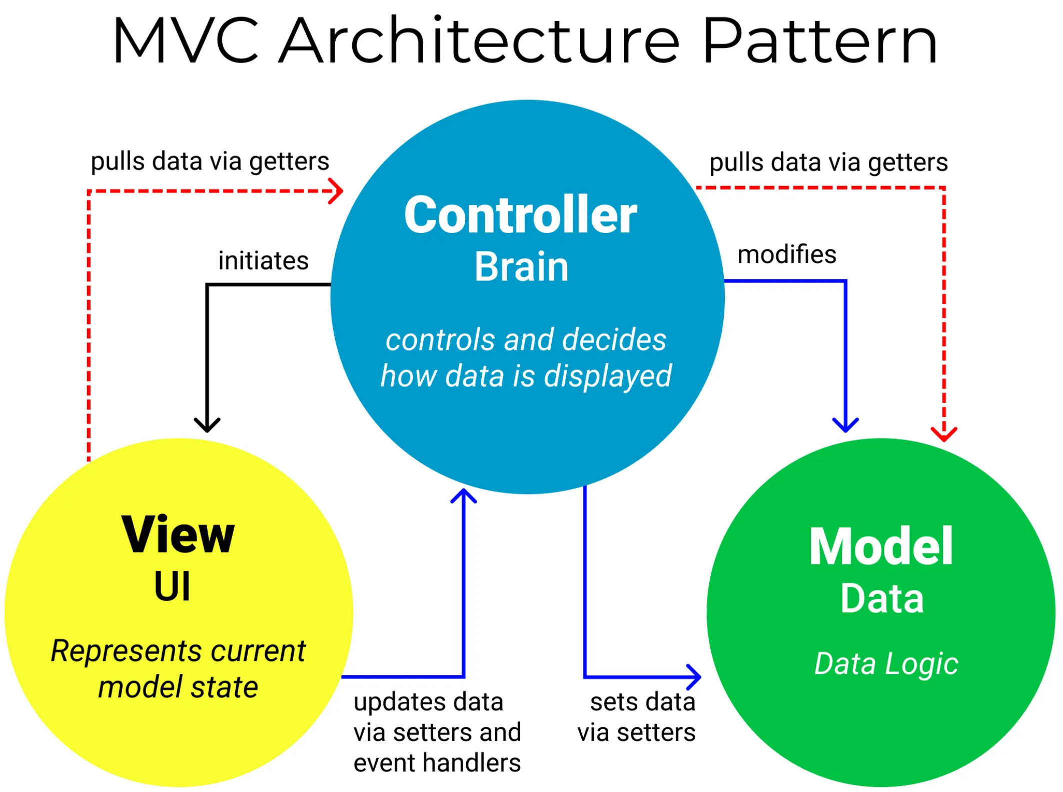


Figure 3. The Model-View-Controller (MVC) software pattern

In addition to standard packages such as *scipy*, *numpy*, *pandas* and *six*, we used four packages to compute and visualise our simulation results. The packages *matplotlib* and *pygame* played a key role in developing a visual and interactive representation of the agents' movements. The first is a comprehensive library for creating static, animated and interactive visualisations, while the second is designed for writing video games with modules for computer graphics and sound (Figure 1). Similarly, the *networkx* and *seaborn* packages helped to visualise simulation metrics and results. The first is a tool for creating, manipulating and exploring complex networks of nodes and edges (Figure 5b), while seaborn is a statistical data visualisation library we used to create heat maps and line graphs.

3. Results

This chapter is organized into three main sections that cover the central aspects of our study. The first present the calculation of key metrics, including the utility function (Ch. 3.1), priority function (Ch. 3.2), and movement data (Ch. 3.3). The second detail the implementation of the MVC architectural pattern to create our cellular network model (Ch. 3.4). The third present the results of five tests selected from the RiMEA guidelines (Ch. 3.5).

3.1. Utility Function: Calculating the most utile movements

This section focuses on the utility function, specifically how we decided to calculate it (Ch. 3.1.1 and 3.1.2) and adjust it (Ch. 3.1.3 and 3.1.4) for our model. The utility function is a fundamental variable in our study, as it determines the movements of each agent towards their destinations.

Theoretically, there are two main approaches to defining and understanding this function.

The first approach is a physical analogy, in which targets are described as protons and agents as electrons. In this analogy, agents move to the next cell as if attracted by targets and repelled by other agents or obstacles. However, this analogy has been criticized because, in real pedestrian scenarios, there are no physical forces acting directly on or between people.

The second approach, derived from psychological and neuroscientific theories, suggests that agents try to find the most favorable positions or paths within the space they can reach on the grid. This approach is a more appropriate representation of the problem, where possible movements are assessed, and the one that maximizes some form of benefit or minimizes some form of cost for the agents is chosen. In our model, the benefit is achieved by reaching the goal in as few steps as possible, while the costs are incurred through collisions with other agents or obstacles.

3.1.1. Euclidian Distance: Elementar Method to calculate the Utility

The first and more elementary method of computing the utility function is to use a scalar function, denoted as Φ, whose gradient can be utilized to determine the potential of any field. A straightforward and intuitive way to implement a potential function is through the Euclidean distance. By analogy with physics, the potential of each cell is calculated as the negative Euclidean distance to a given target. Figure 4 shows the parallelism between the Euclidean distances of each cell and the potential these cells have to a given target.

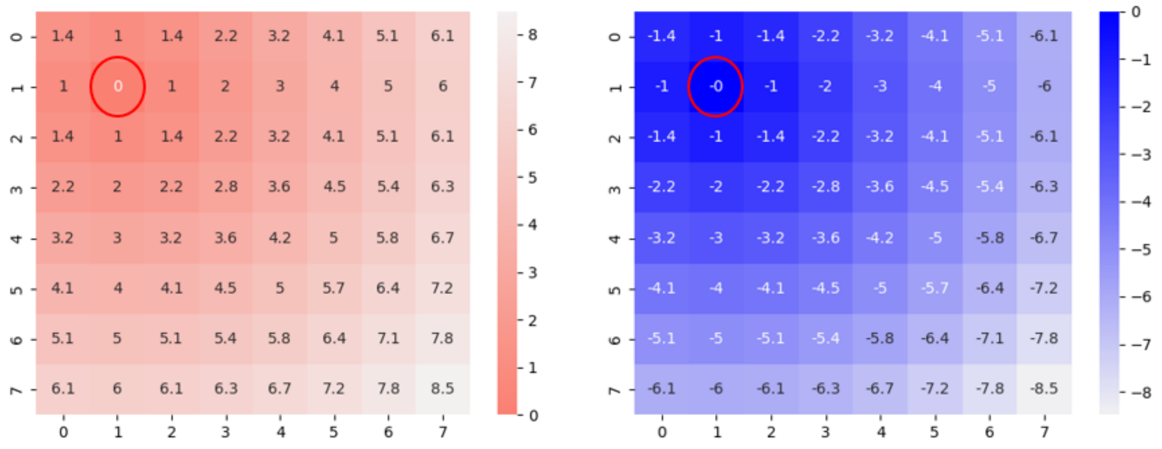
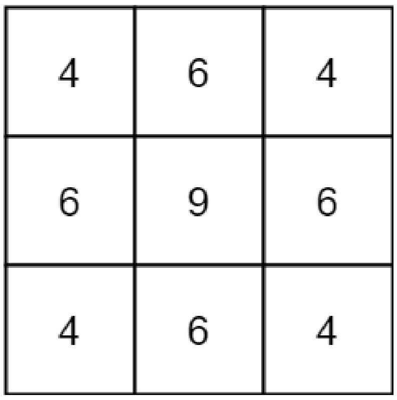


Figure 4. The Paralleleism between euclidian distances (left) and field potentials (right). The target is at coordinates (1, 1)

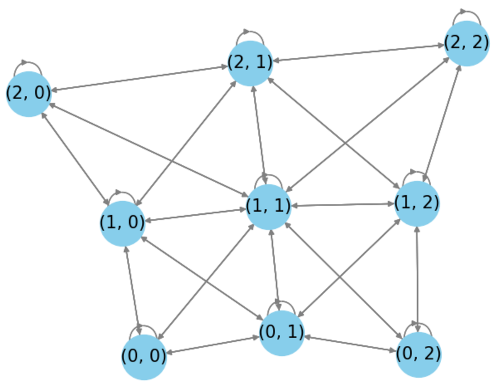
3.1.2. Dijkstra's algorithm : Complex Method to calculate the Utility

The second and more complex method of computing the utility function is to use the Dijkstra's algorithm. This finds the shortest path from a starting node to all other nodes in a graph with non-negative edge weights. It begins by setting the distance to the starting node as zero and all other nodes as infinity. The algorithm repeatedly visits the unvisited node with the smallest known distance, updates the distances to its neighboring nodes if a shorter path is found, and marks the node as visited. This process continues until all nodes have been visited or the target node is reached. The algorithm ensures that it visits all nodes that are closer to the source node than the farthest target node, resulting in the shortest paths from the starting node to the target node and any other nodes within this range, which can be inefficient for large graphs. While multiple shortest paths may exist, Dijkstra's algorithm identifies one such path. For a comprehensive explanation of how the algorithm works refer to [2].



4	6	4
6	9	6
4	6	4

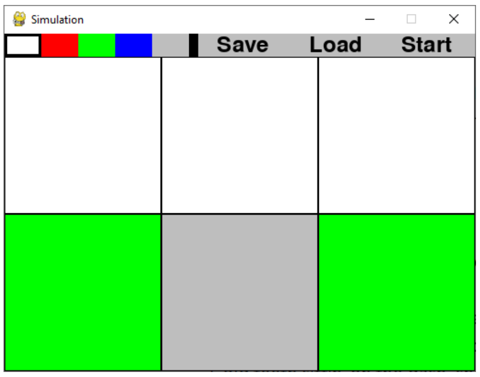
(a) 3x3 Map with 9 Fields, each field annotated with it's respective outgoing edge count



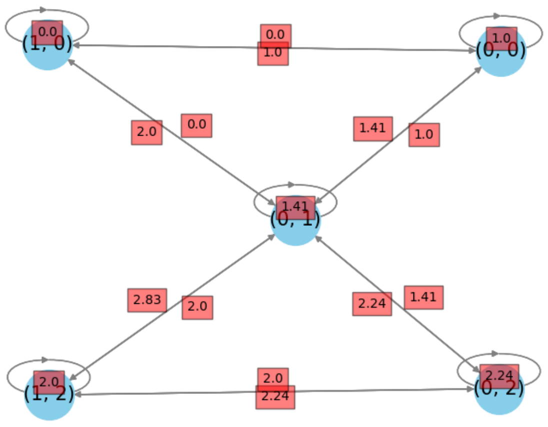
(b) Visualisation of the fully connected directed graph of a 3x3 Map

Figure 5. Generating a fully connected graph from a 3x3 Map

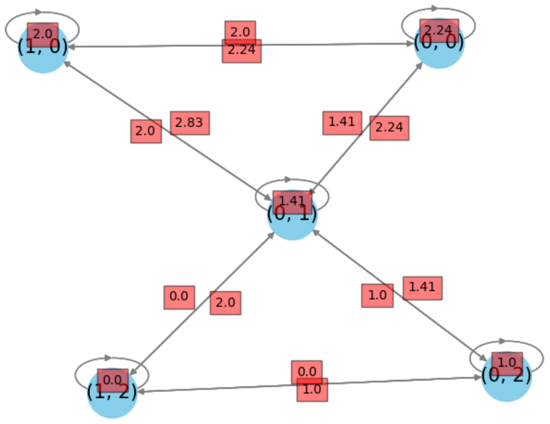
To extract the shortest path from an agent's current position to its target, we need a weighted graph with multiple edges. The complexity increases with multiple targets. In a 3x3 matrix map, each field can have up to 9 edges, including staying on the same field. This results in a graph with 9 nodes and 49 edges for one target. With multiple targets, each field's edge weights vary based on their distance to each target. With two targets, the number of edges doubles to 98; with three targets, it triples, and so on. As we explained explain in section 3.1.2 this makes initialization costly for many targets, but afterward, only fields/nodes affected by the agent's moves will need to be updated. Parallelising the weight assignment for different targets mitigates the initialisation time, reducing it from minutes to seconds. The distribution of edges accross fields within the map is show in figure 5a. The weights for each edge are positive, unlike the negative utility values. In this example, we use the absolute value of the utility by removing the sign. This is valid because the highest negative number, closer to 0, is equally distant from 0 in the positive direction. Since lower weights are preferred, we still find the shortest path even when using the absolute values of the utility. As the utility is also influenced by the agents' repelling fields, we need to update the graphs weights as the agents move. These dips adjust the initial weights, which represent the combined influence of agents' dip values, the fixed euclidean distance, and static obstacle dips. An example is provided in figure 6a. We used NetworkX's built-in method to find the shortest path to the specified target instead of implementing our own algorithm.



(a) Visualisation of a map with two targets and one obstacle



(b) Euclidean distances to target in the bottom left corner



(c) Euclidean distances to target in the bottom right corner

3.1.3. Step effort: Correcting the Personal Utility based on the Cells Orientation

A major limitation of calculating personal utility with these two approaches is that the orientation of cells neighboring an agent's current position is ignored. In other words, the step effort each agent must expend to reach a cell is not taken into account when calculating its utility. However, agents have to take "larger steps" to move diagonally compared to moving along the grid axes.

Under the assumption presented in **??** that a person can cover an average distance of 40 cm with one step, each agent in our model will require one step to move to a cell along the same axis and approximately 1.414 steps to move to a cell diagonally from its current position. This value is calculated using the Pythagorean Theorem between an agent's current cell $f1$ and any other cell $f2$ in its Moore environment, as expressed in equation 1.

$$d(f1, f2) = \sqrt{(f1.\text{col} - f2.\text{col})^2 + (f1.\text{row} - f2.\text{row})^2} = \sqrt{(1)^2 + (1)^2} \approx 1.414 \tag{1}$$

Thus, the first corrective mechanism we implement in our model's utility function is to multiply the personal utility of agents by factors of 1 or 1.414 according to the number of steps required to reach each of their adjacent cells. As a result (Figure 7), the potential of these cells no longer corresponds to the simple negative Euclidean distance to the target but is instead weighted depending on their orientation to the agent's current cell.

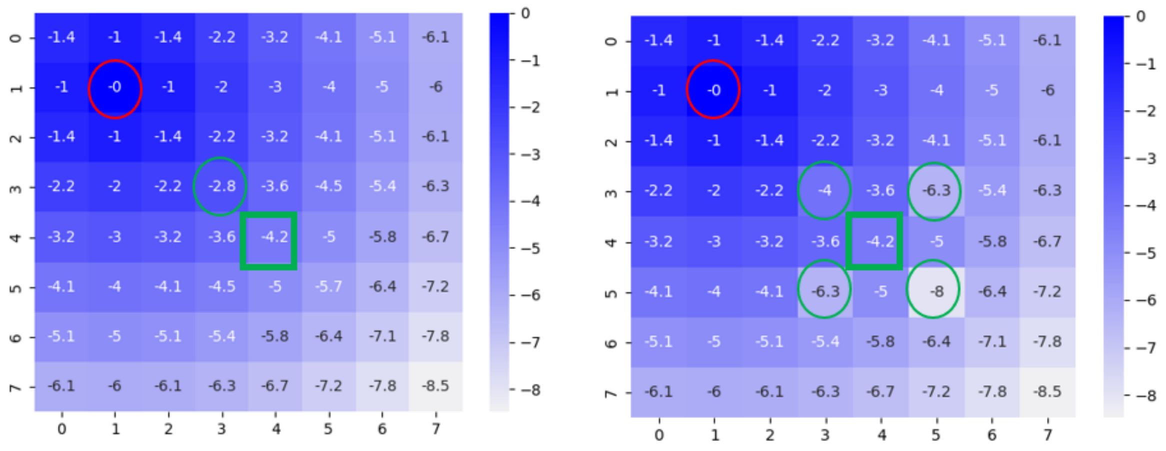


Figure 7. The Difference between field potentials (left) and corrected field potentials (right) due to the step effort. In this example both are calculated as negative euclidian distance. The target is at coordinates (1,1) and the agent at coordinates (4,4)

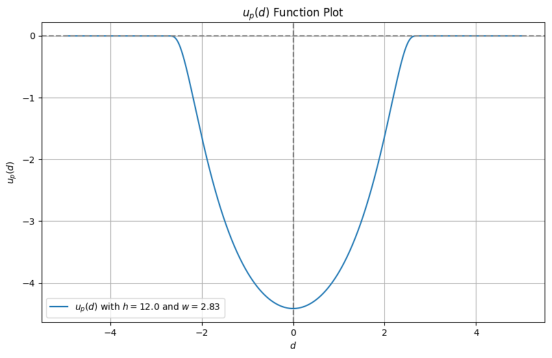
3.1.4. Dip: Correcting the personal utility the Influence of other Agents and Obstacles on the Utility

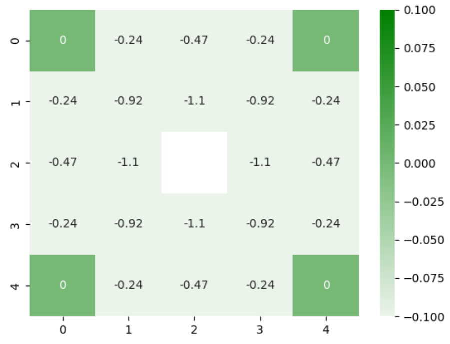
Another limitation of calculating personal utility with the presented approaches is the presence of other agents or obstacles around an agent's current position is ignored. In practice, people tend to choose less direct paths to their target to avoid getting too close to others. By analogy with physics, agents and obstacles are considered to have a repelling field around them, which we refer to as a dip.

The dip implemented in our model extends across two rings of surrounding cells and negatively influences the utility of these cells for the agents around them. The formula 2, inspired by Friedrich's well-known mollifier in [3], relies on the parameters w (width) and h (height) to determine a dip's strength and range. In Figure 8a the negative height represents the valley of the function and the width indicates the outer edge of the repelling field. This is given by $\sqrt{2^2+2^2}$ for a degree of adjacent fields connected to the current field equal to 2.

$$u_p(d) = \begin{cases} -h\exp\left(\frac{1}{(d/w)^2-1}\right) & \text{if } |d| < w \\ 0 & \text{else} \end{cases} \tag{2}$$

Thus, the second corrective mechanism we implement in the utility function is to subtract the dip values from the utility values of each cell. As shown in Figure 8b, these dip values depend on the proximity to an agent or obstacle. For cells outside this repelling field range, the dip values equal zero. For cells within the repelling field range of multiple agents or obstacles, the dip values are summed, making these cells unattractive for other agents as they represent a crowded or obstructed path. The correctness of this dip function is enforced by an accompanying unit test. This test calculates the supposed dips exerted by the agent on the surrounding cells and compares these results to values known to be correct.

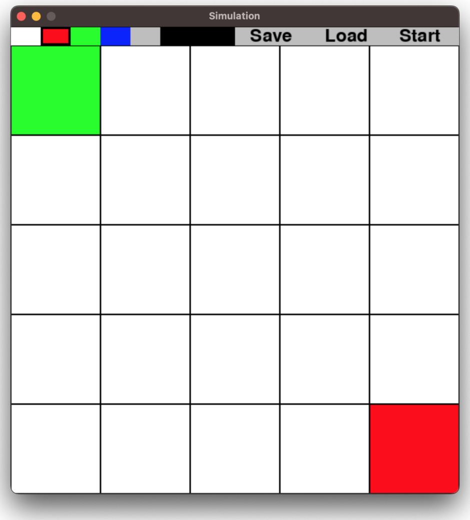




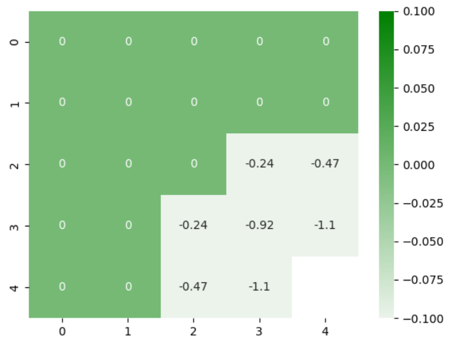
(a) The illustrated mathematical dip function **(b)** The illustrated repelling field around the agent

Figure 8. Illustration the dip: an agent's repelling field positioned at coordinates (2,2) of the grid

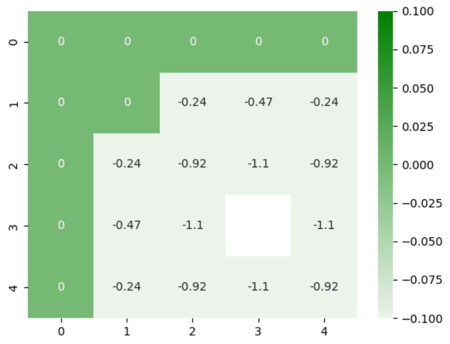
To illustrate how an agent moves through a grid while carrying its repelling field, we place one agent in the lower right corner and a target in the upper left corner. The agent will choose the shortest and most efficient path to reach the target, moving diagonally to head directly towards it. This change in utility will effect other agents decision making, but not the agent carrying this particular field.



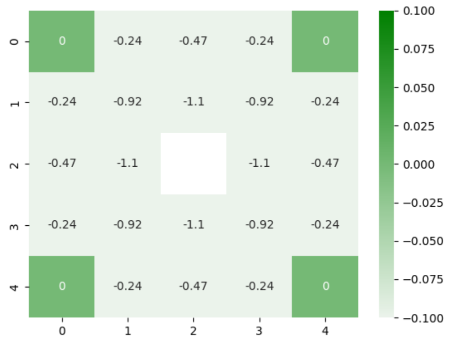
(a) Illustration of the initial situation



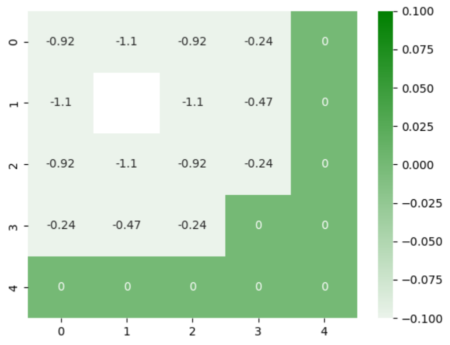
(b) Step 0: The agent spawns here.



(c) Step 1



(d) Step 2



(e) Step 3: There is no last step, as the agent reaches its target after this one.

Figure 9. The agent is surrounded by a repelling field, making all affected fields less desirable for other agents

249 ### *3.2. Priority Function: Determinating a realisting and correct correct movement order*

250 This section focuses on the priority function, specifically on the logic we used to implement it. 251 The priority function is fundamental for determining the order in which the agents are moved and for 252 calculating the most relevant simulation metrics, such as velocity, density, and flow (Ch. 3.3).

As we will describe below, each agent has a *priority value* which is updated before every move and corresponds to the necessary time to move from its starting field to a desired one. More concretely, the *priority value* at a time t for an agent A is calculated as the sum of its previous priorities (base time (BT)) and the predicted time to move from its current to the desired field (predicted move time (PMT)). In other words, this is the sum of the PMT between each pair of fields on one agent's path, starting from its spawn field and ending at its desired field. As shown in Figure 10 (right), the PMT between two fields is calculated as the distance an agent needs to walk between two fields (dist) divided by its free-flow speed (ffs). As described in section 3.3.2 and 2.2.2, the distance can be either 40 cm or approximately 56 cm, while the free-flow speed results from a normally distributed function with a mean of 1.34 m/s.

The reason for calculating the priority values as summarised in equation 3 and setting the simulation time accordingly is that these values provide a realistic sequence for the agents' movements and give accurate results when calculating time-based metrics. The realistic aspect derives from the fact that the last PMT determines the time that each agent would need to move from its current position to a desired field, allowing us to prioritise the agent that would actually move faster in a real-life scenario. The accuracy aspect is supported by the observation that using other variables to order the agents, such as their creation time or free flow speed, will provide less accurate results in our model.

$$UtilityValue_{At} = BT_{At} + PTM_{(field_{At}, field_{At+1})} = \sum_{0}^{t} PTM_{(field_{At}, field_{At+1})} = \sum_{0}^{t} \frac{dist_{(field_{At}, field_{ASt+1})}}{ffs_A} \tag{3}$$

However, this method is only valid if the desired field is exactly the one that the agent will enter when it is its turn to move, e.g. if the movement of one agent does not affect the decision of the next most utile field for other agents. Such scenarios will still occur in our model, since agents can block or apply dip onto the desired fields of other agents. In such cases, the faster agent will have its PMT confirmed and its *priority value* updated accordingly, while the slower agent will have its PMT and BT recalculated. In the case the slower agent is completely blocked (Figure 10 (centre and right)), its PMT will be set to infinity. As soon as its desired field becomes free, its PMT will be calculated again as usual (figure 10 (right)), and its BT will be updated to reflect the priority time of the agent that occupied that field as it allowed the slower agent to move again.

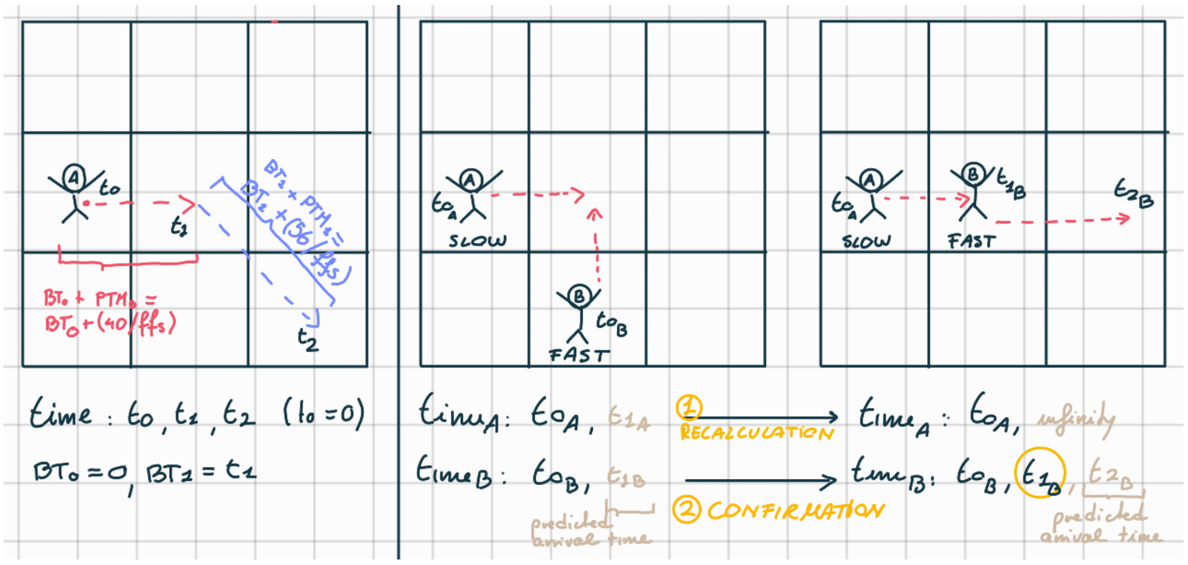


Figure 10. Priority calculation with (left) and without (right) free-line of sight.

3.3. Time, Distance, Velocity, Density, Flow: Calculating relevant movement metrics

In this section, we introduce key metrics utilized to characterize pedestrian flows. The accompanying figure 11 illustrates the three possible movements an agent can make during each simulation time step (left) and provides a concrete example for the four metrics related to these movements, including time, distance, velocity, density and flow (right).

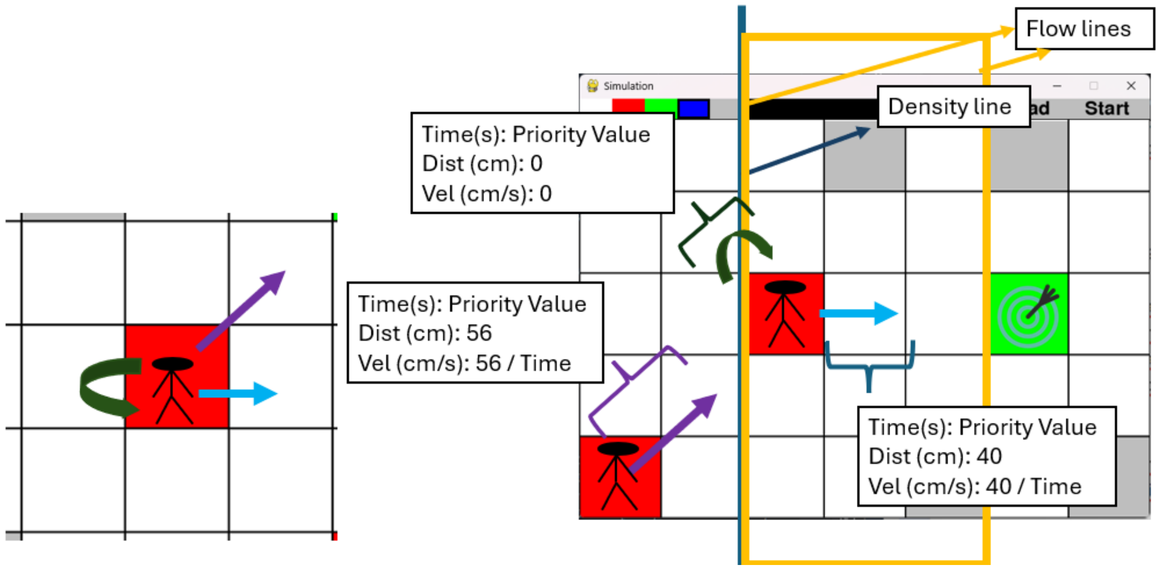


Figure 11. Three possible agents movements (left) and the four key metrics utilized to characterize pedestrian flows: distance, velocity, distance, density (right)

3.3.1. Time

The variable time refers the current simulation time. At the start of the simulation, the value is set to 0 and is updated with each agent's movement. This corresponds to the lowest *priority value* of the agent among those still on the grid. Its calculation was described in section 3.2. By doing this, the simulation unit time, the agent movements and the updating of the following simulation metrics are synchronised, ensuring that the simulation results are easy to interpret, realistic, and accurate. As The final simulation time is determined by the *priority value* of the last agent entering its target.

3.3.2. Distance

In our model the distance describes the number of centimeters each agent travels from its starting position to a desired location. At the beginning of the simulation, the distance is set to 0 for all agents. During the simulation, this value is updated by adding the distance covered with the most recent movement to the cumulative distance previously traveled, thereby providing information on both partial and total distances. As illustrated in figure 11 (right), an agent can move three different distances depending on the orientation of the target field relative to the current one. These distances are 0 cm if the agent remains in the current field, 40 cm if the agent moves along the axis directions, and approximately 56 cm if the agent moves to a diagonal field. These values are derived from the cell length of 40 cm, as specified in section 2.1, and are calculated using the Pythagorean theorem.

3.3.3. Velocity

Similarly to distance, velocity describes the speed each agent requires to reach a desired field starting from their spawn field. This value is calculated individually for each agent, beginning at 0 at the start of the simulation and being updated after each movement until the target is reached. It is determined by dividing the distance covered from the starting field to the current field by the current simulation time.

3.3.4. Density

The pedestrian density describes the number of agents passing a fictive line on the grid as they move from their starting positions to their targets. This is calculated by dividing the number of agents by the height of the grid. It is crucial to strategically position this line between the agents' starting positions and their target destinations to better visualize the agents' velocity stabilizing as they approach their target and queue to enter it.

3.3.5. Flow

Similarly to density, pedestrian flow is calculated by measuring the time it takes for each agent to travel from a first virtual line to a second virtual line. The first one corresponds to the density line described above and the second is positioned further along the corridor.

3.4. Model: Structuring and Implementing a Cellular Network Model for pedestrian streams

In the following section, we delve into the structural 3.4.1 and logical **??** implementation of our cellular network model.

3.4.1. Model Structure

As discussed in section 2.3, this model adheres to the MVC architecture pattern. The MVC pattern comprises three distinct yet interacting components, namely *Controller*, *Model*, and *View*. Figure **??** provides a summary of how this architecture is implemented in our source code.

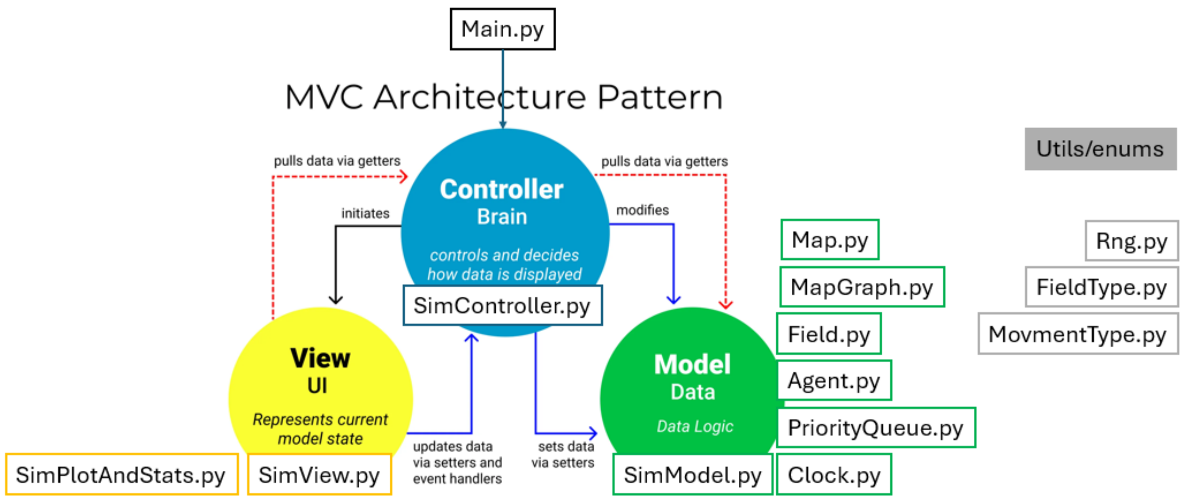


Figure 12

The *main.py* file is the primary file that users must consider when running the simulation. It contains crucial variables necessary for the simulation, such as the size and shape of the grid, the number of desired agents, obstacles, targets, and the algorithm used for moving these agents. When starting this file, the specified variables are collected, the three MVC components are initialized, and a for-loop is entered. This loop is responsible for maintaining our GUI visible until the entire simulation is finished. By the first loop the GUI, represented as a 2D cell grid with interactive buttons and cells, will be displayed to the users. At this point, the user can select the desired buttons and cells to assign spawn agents (red), obstacles (grey), or targets (green) to the desired fields. Optionally, the positions can also be generated randomly by setting the corresponding variable in the *main.py* file. After specifying at least one agent and one target, the simulation can be initialized and then started.

The coordination of the simulation initialization and start is managed by the *Controller* component, which utilizes functions implemented in the *View* and *Model* components. A more detailed explanation of the Controller's underlying logic is provided in Figure 13.

The *View* component contains the *simViewer.py* and *simPlotAndStats.py* files. The *SimViewer.py* file is responsible for enabling intuitive user interaction with the simulation environment and rendering all simulation data calculated by the *Model* component on behalf of the *Controller* component. Following the same logic, the *simPlotAndStats.py* file generates heat maps and line diagrams, which are used for visualizing the simulations's characteristics and metrics.

The *Model* component includes the *SimMode.py* file and several classes we created to best implement our cellular neural network. Among these, the most important are *Map.py*, *Field.py*, and *Agents.py*, as they contain the fundamental functions used by the *Controller.py* to calculate the data and send these to the *View* component to be visualized.

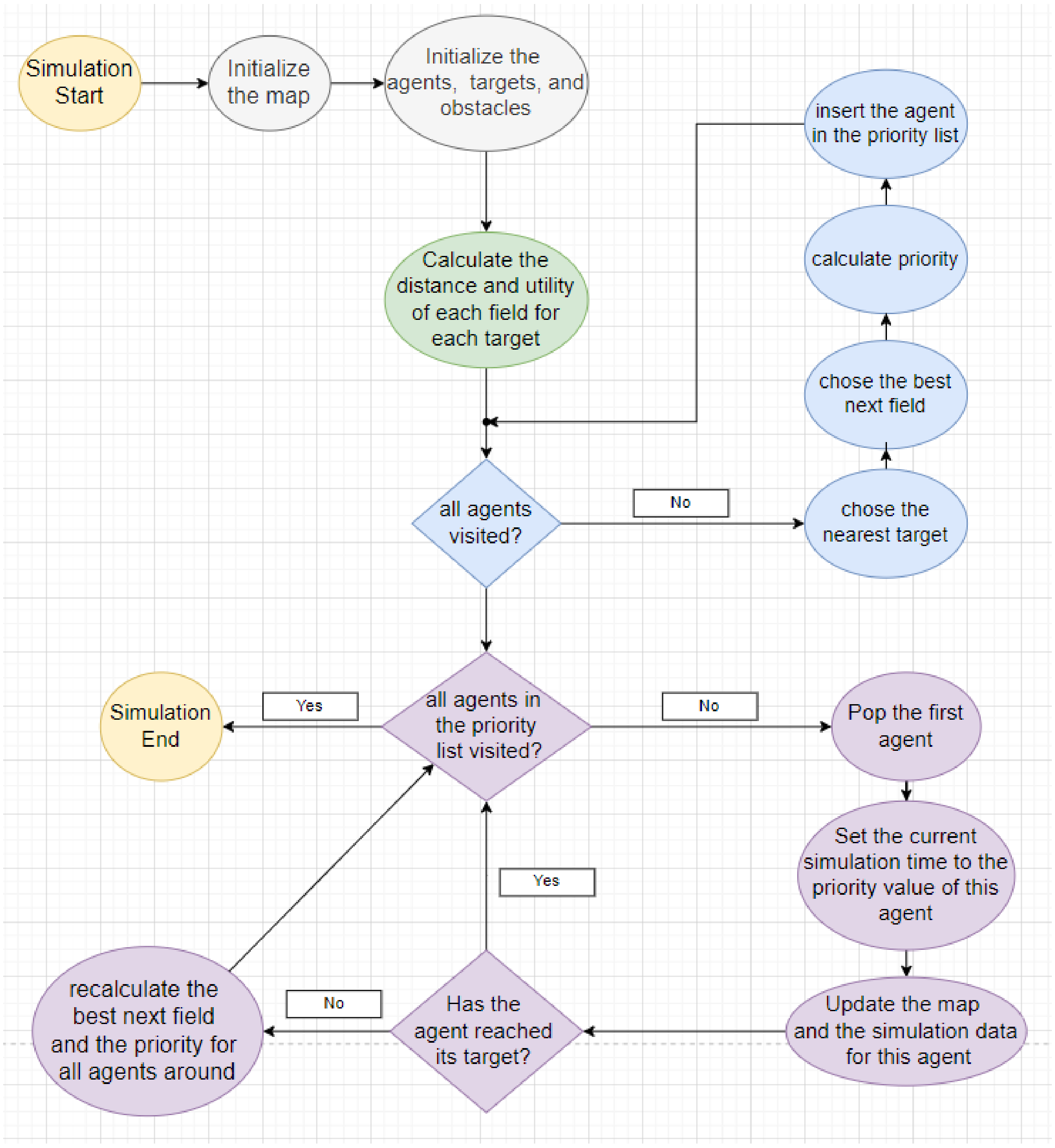

Simulation Start
Initialize the map
Initialize the agents, targets, and obstacles
Calculate the distance and utility of each field for each target
insert the agent in the priority list
calculate priority
chose the best next field
chose the nearest target
all agents visited?
No
all agents in the priority list visited?
Yes
No
Simulation End
Pop the first agent
Set the current simulation time to the priority value of this agent
Update the map and the simulation data for this agent
Has the agent reached its target?
Yes
No
recalculate the best next field and the priority for all agents around

Figure 13

As mentioned above, the simulation starts and finishes with the entering and leaving of a loop implemented in the *main.py* file. These two steps are visualized in yellow inthe flow diagram 13. During this loop, the simulation is initialized and then started.

The gray elements of the diagram show the initialization phase, which includes, in particular, the initialization of the map and the agents, targets, and obstacles populating it, according to the variable specified by the user.

The other elements of the diagram represent the starting phase of the simulation, which includes three main steps. First (green), the calculation of the distance and potential of each field with respect to each target, as described in section 3.1. Second (blue), the insertion of all agents in a priority list according to their priority values, as described in section 3.2. Third (violet), the extraction of the first agent in the queue and the movement to from its current field to its desired field. If the agent has not yet reached its target, it will be reinserted in the queue with newly calculated most utile field and priority value. By doing so, the list will gradually shrink until all agents have reached their desired targets. As described in sections 3.2, 3.2, and 8, all agents within the range of this agent's dip will have to be updated. More precisely, their most utile target and priority value will be recalculated, as the movement of the last extracted agent may influence them, for example by blocking an agent's desired field or imposing a strong dip on an agent's desired field.

3.5. Simulation Results: Evaluating the results of the pedestrin streams with our Cellular Network Model

In order to ensure the logical correctness and robustness of our model, we conducted four primary tests. These tests represent just a set from the standardized test cases commonly used within the pedestrian dynamics RiMEA community.

3.5.1. Free-line of sight

The free-flow speed is the speed in which an agent moves through the grid without being obstructed by other agents or obstacles in its way. We observed no deviaten from the assigned free-flow speeds in multiple simulations. Movements across the grid's diagnoals are corrected and the influence of these step on the speed and distance covered mitigated.

In this concrete example we analyze the behaviour and metrics of three agents (Figure 14 left), which are positioned at 0°, 45°, and 90° in respect to the central target and have free line of sight towards it.

Each of the three agents in the simulation selects one of the two targets based on its proximity, determined by Euclidean distance from its current position. Once chosen, the selected target remains fixed for the duration of the simulation. Agents typically choose paths that involve less diagonal movement due to adjustments made to balance energy expenditure, aiming to reach their targets efficiently. When an agent is positioned directly beneath or beside a target, it consistently moves towards it along a direct path, minimizing unnecessary zigzagging.

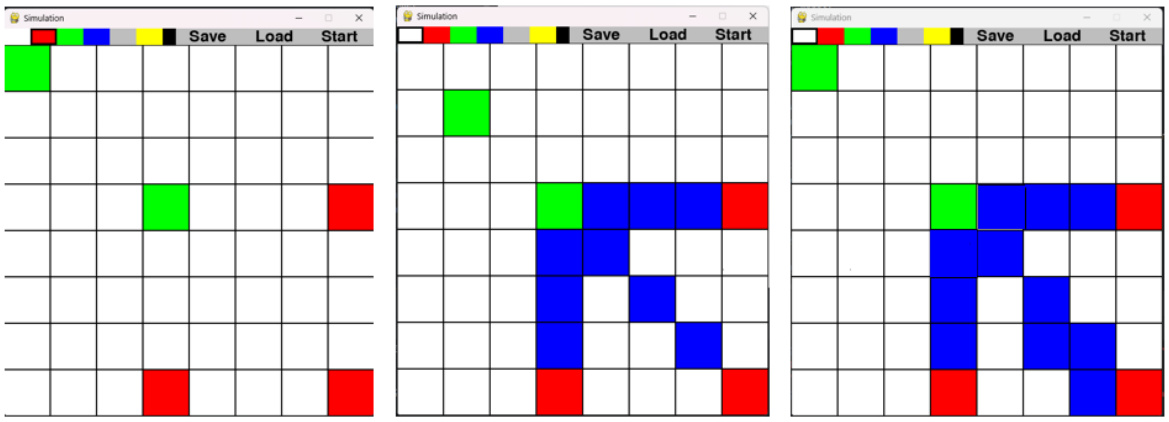


Figure 14. Starting position (left) and most utile fields for the agents without (center) and with (right) considering the steps effort of each agent

From figure 15 we can observe that all targets are able to recognize the central target as the best one to reach in terms of euclidian distance.

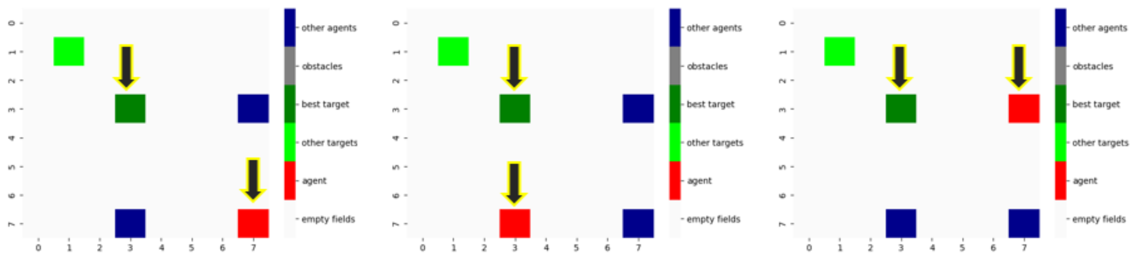
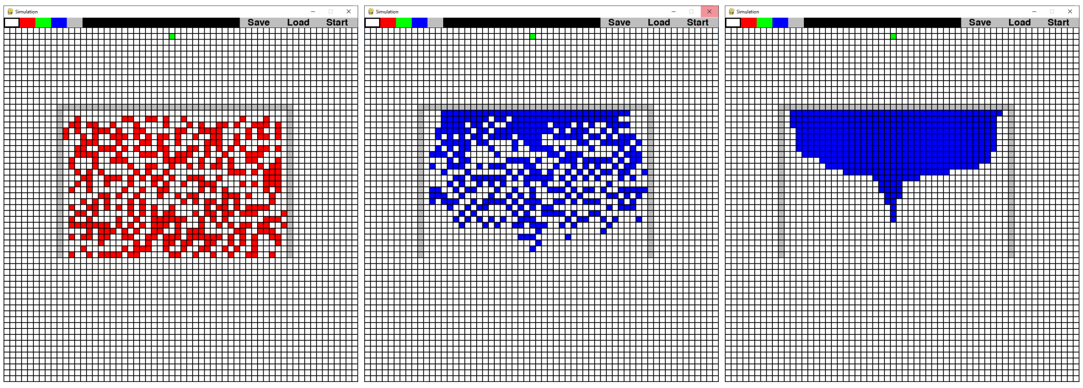


Figure 15. Recognition of the best target for each agent

3.5.2. Chicken Tests

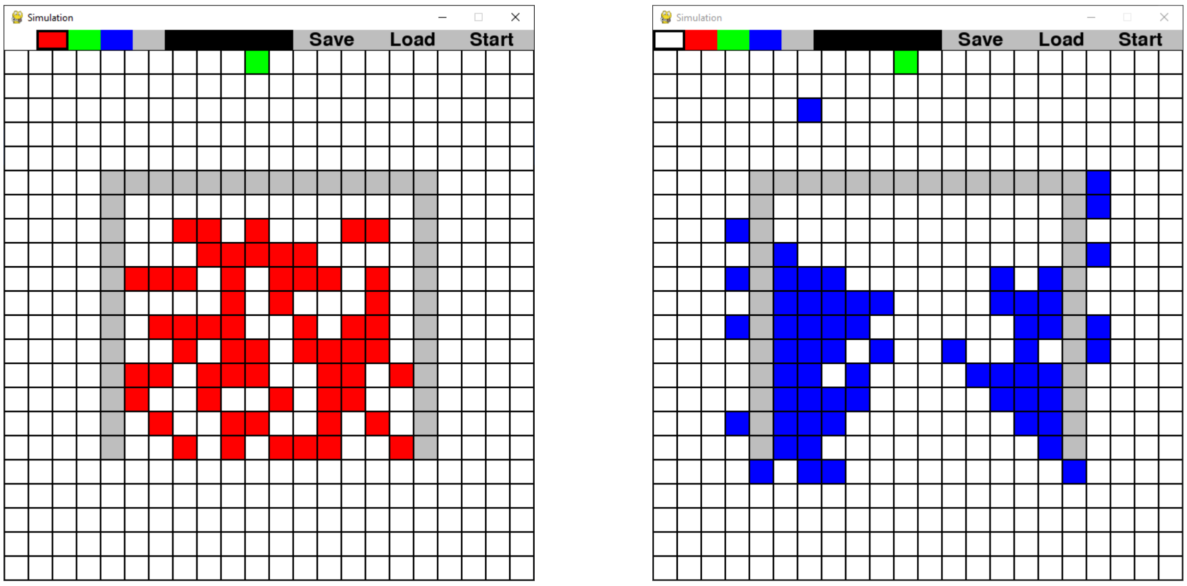
In the so called chicken test virtual entities move towards a destination but encounter a U-shaped obstacle directly in their path. Agents, which can still see the target through the obstacle in their way, such as through a fence, become stuck at the obstacle. In contrast, dogs are able to navigate around the obstacle and continue towards the destination. This metaphor is effective because the Euclidean distance method focuses solely on the immediate surroundings of the agent's current position, prioritizing the single field with the highest present utility without considering future benefits. In contrast, the Dijkstra algorithm assesses the entire map, determining the shortest possible route from the current position to the target.

In the simulation implementation, each approach leads to the expected conclusions. Figure 16c illustrates that agents within the fenced area, using the Euclidean approach, do not attempt to circumvent obstacles. Instead, they gather extremely close to each other, forming a dense crowd, and adopt a bell-shaped formation as near to the target as possible without overlapping. Conversely, the Dijkstra algorithm results in the successful completion of the simulation, with all agents reaching their assigned targets. Agents navigate around both sides of the enclosed area, moving left or right based on their starting positions. The edges of the fence, particularly where agents can exit, become the densest areas. Once outside the enclosure, the crowd density decreases significantly, and agents proceed directly towards their targets. There appears to be a significant bottleneck at the edge of the enclosure. When it is occupied, other agents cannot enter their prefered space. Ideally, agents avoid each other and navigate around other agents.



(a) **(b)** **(c)**

Figure 16. Change of formation over time while executing the chicken test using the euclidean distance as movement strategy

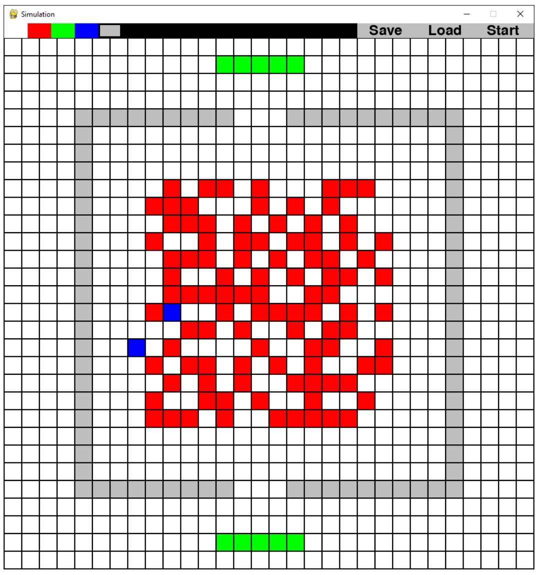


(a) **(b)**

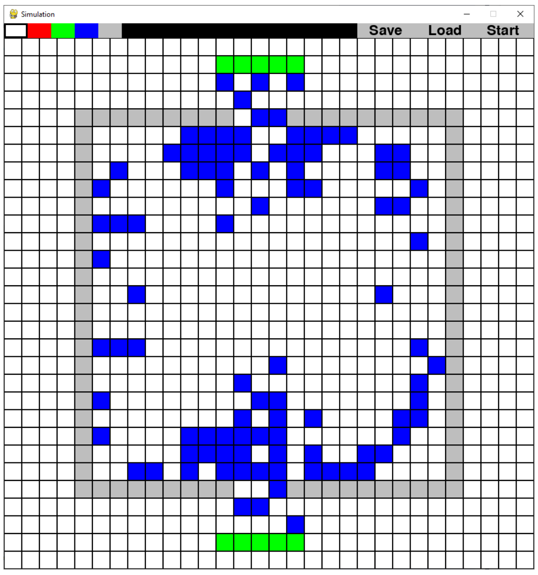
Figure 17. Change of formation over time while executing the chicken test using Dijkstra's algorithm as movement strategy

3.5.3. Room Evacuation

This test is designed to simulate evacuation times from a quadratic room with initially two doors, then four. Evacuation time stops once all occupants have exited safely. We will employ Dijkstra's algorithm to optimize evacuation speed. People will be positioned in the room in a pseudo-random manner to prevent clustering and ensure results are unbiased, except that agents will choose the nearest exits. Standard western doors are typically 120cm wide. Therefore, we will construct a wall opening that spans 3 fields in width. For this particular scenario, the room measures 20 fields in width and height, with each field measuring 40 cm. This makes the room dimensions 8m x 8m, totaling 64 square meters. The evacuation takes a total of 33 seconds for all 100 agents to reach the safe zone with two doors. When there are four doors installed, the evacuation speed is only 25 seconds compared to a two-door setup, which makes sense as this doubles the possible escape routes. The number of espace routes do not seem to be proportional to a decrease in evacuation time. We observe that agents converge from the sides and gather in the center, creating a bell-shaped crowd in front of each opening. This phenomenon would be more noticeable with a larger scale simulation, but this was not feasible in a timely matter due to the poor performance of the Dijkstra movement strategy implementation.

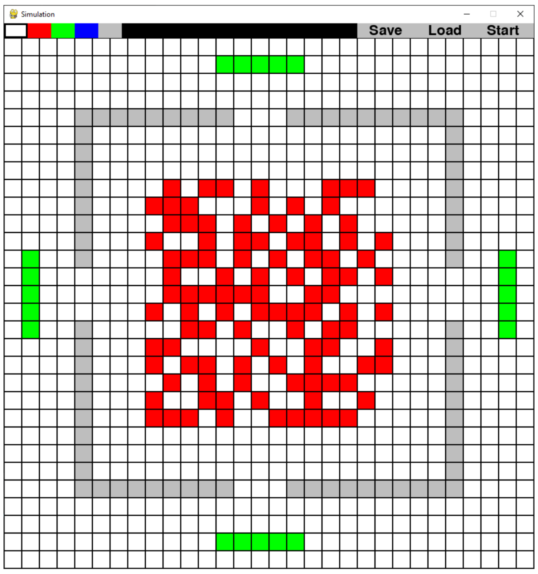


(a)

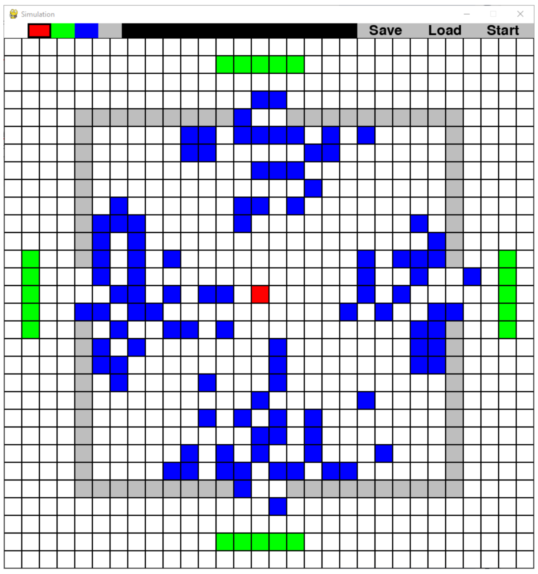


(b)

Figure 18. Experiment setup for measuring evacuation times for 100 agents in a square room with two doors.



(a)



(b)

Figure 19. Experiment setup for measuring evacuation times for 100 agents in a square room with four doors.

3.5.4. RIMEA 4

In this test scenario, agents navigate a corridor with a length of 65 meters and a height of 12 meters, corresponding to 162 and 30 grid cells, respectively. We conducted tests with various coverage percentages. The following figures present results for a 60% coverage rate. To analyze the velocity, density, and flow of the crowd moving through the corridor to the target, we placed the density measurement line described in section 3.3.4 and the flow measurement line described in section 3.3.5 at approximately half and three-quarters of the corridor's length. The Dijkstra algorithm was used to conduct these tests.

Figure 20 illustrates the average velocity of the crowd throughout the simulation. The values start at 0 and peak at 1.3 m/s. By the end of the simulation, the velocity stabilizes at around 1.21 m/s.

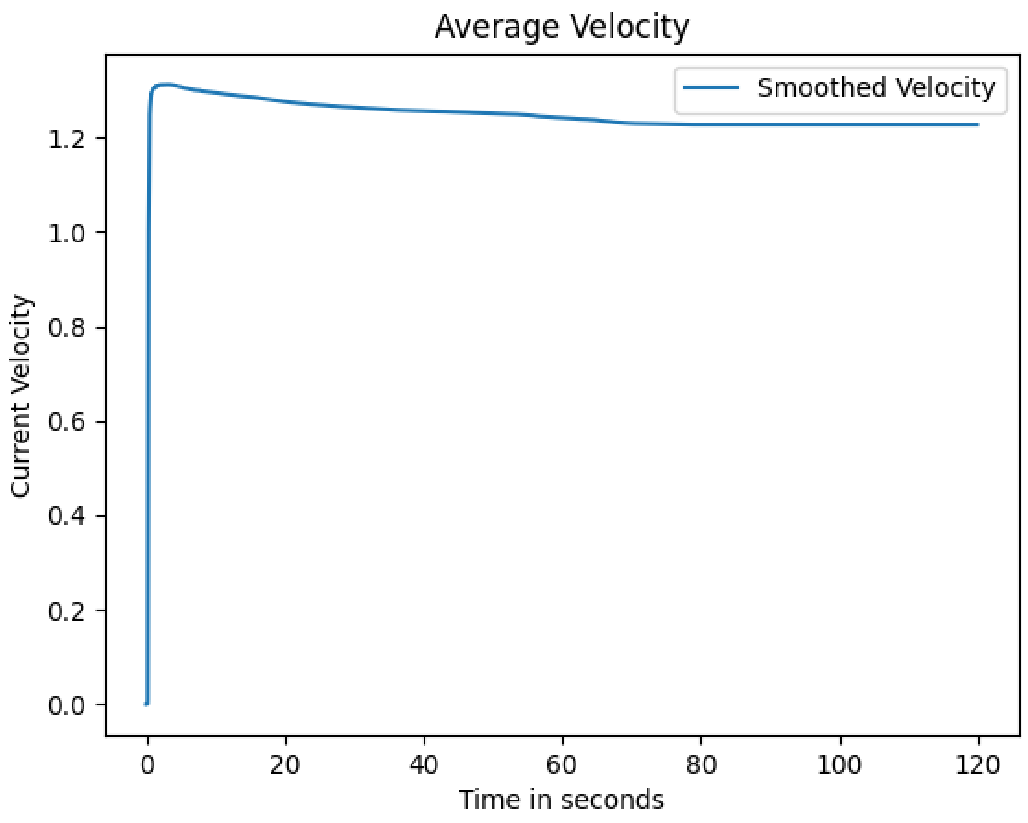


Figure 20. Velocity

Figures 21 and 22 depict the density and flow of the crowd. At the start of the simulation, when no agent has yet crossed the density and flow lines, the values are zero and then change.

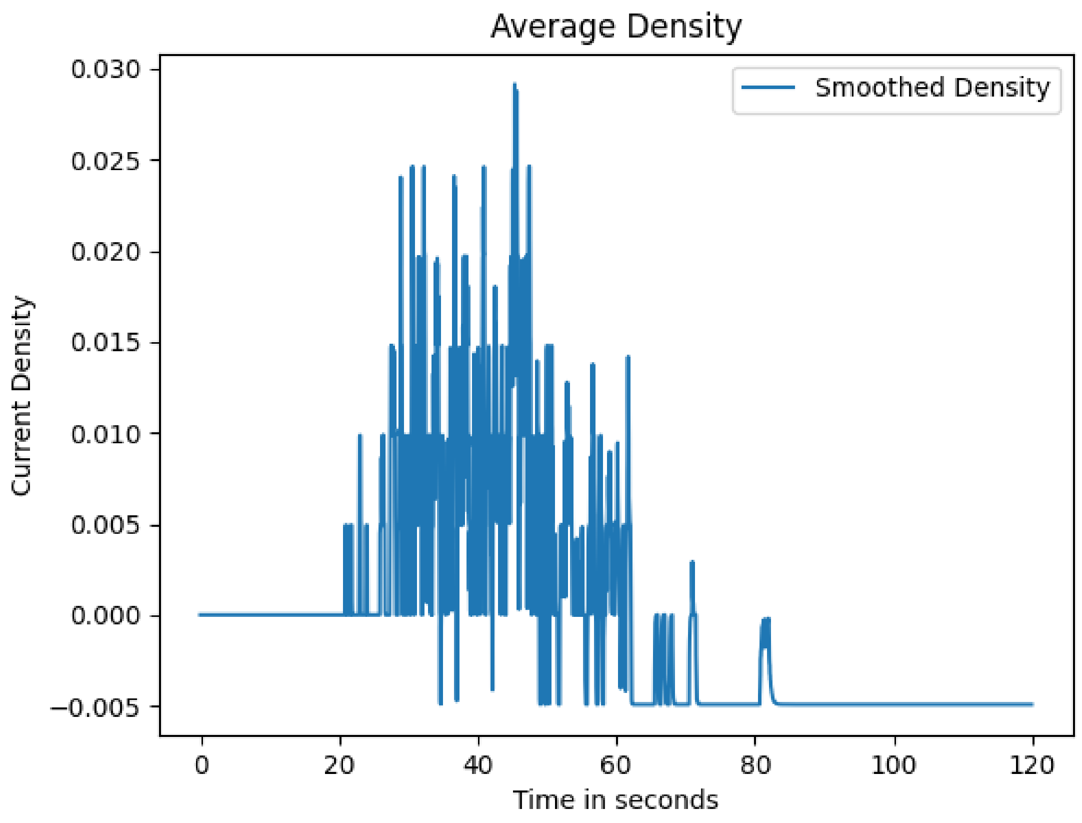


Figure 21. Density

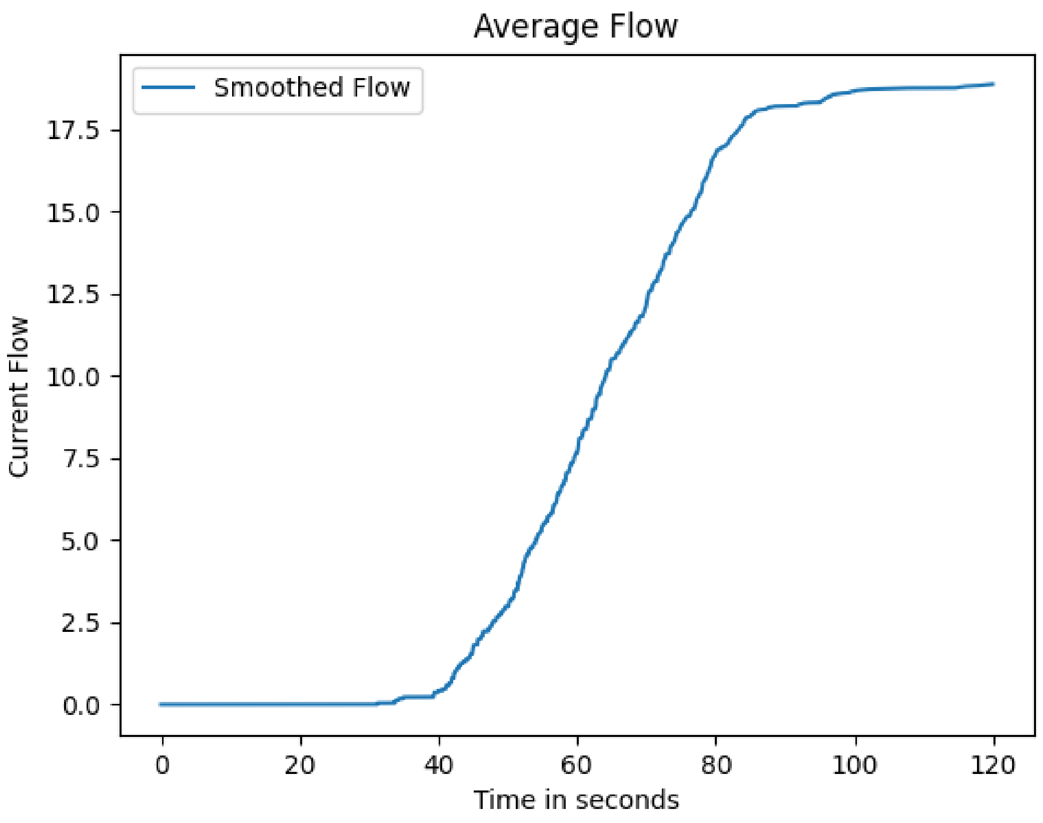


Figure 22. Flow

Figures 23 and 24 present the results of fundamental diagrams. The first figure plots velocity against density, while the second plots flow against density. The results contradict the from Weidmann's established patterns. According to Weidmann's first fundamental diagram, pedestrian density should increase while velocity decreases because, at low densities, pedestrians can move freely and at higher speeds. According to Weidmann's second fundamental diagram, flow should initially increase with density up to an optimal point and then decrease again.

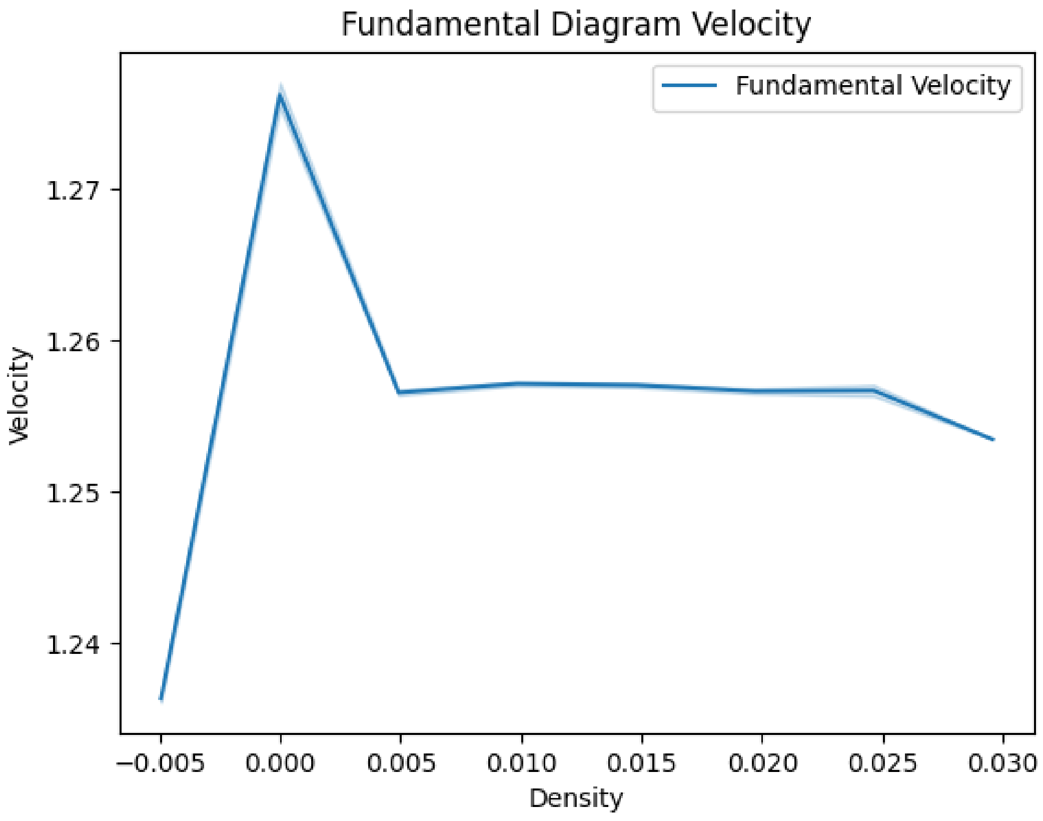


Figure 23. Fundamental diagram 1

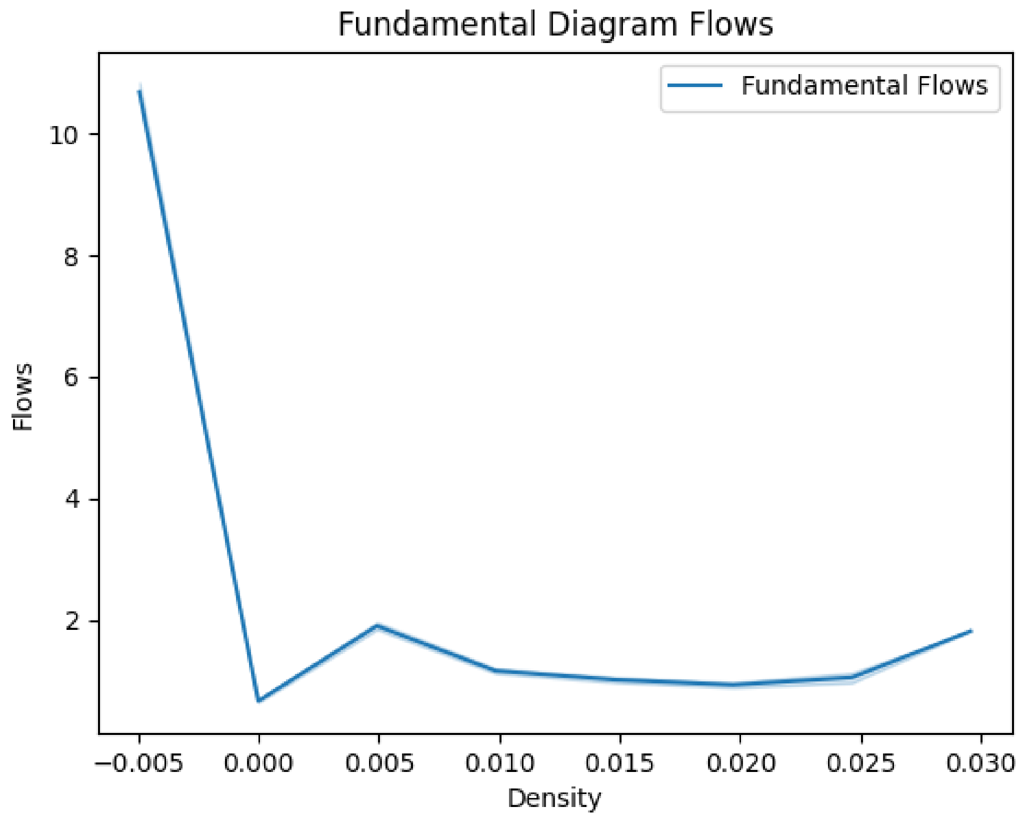


Figure 24. Fundamental diagram 2

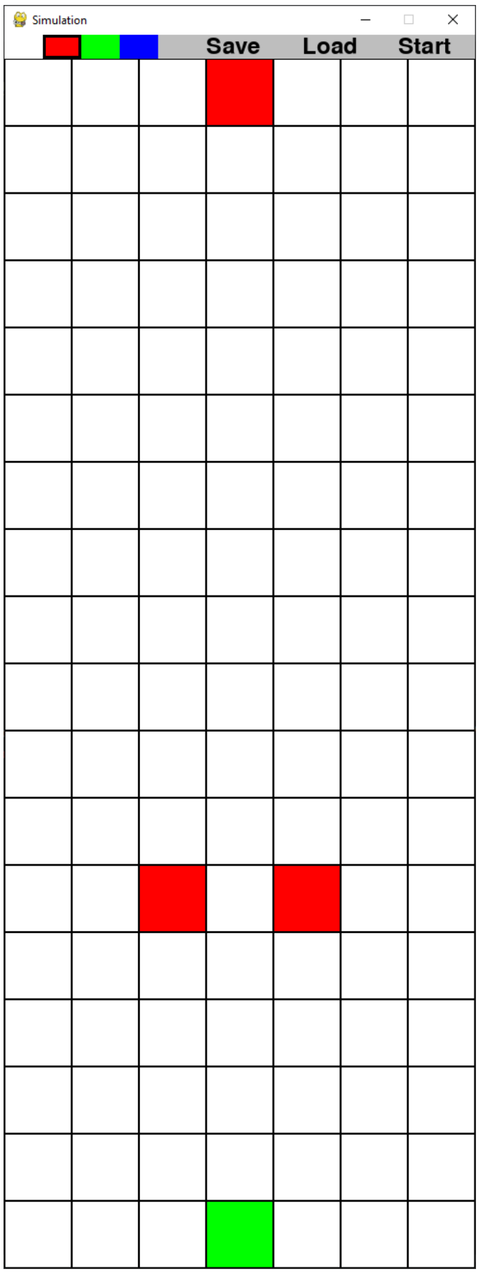
3.5.5. Reenactment of the evasion experiment

In the first assignment, we participated in an experiment to observe whether people evade or go through a gap between two others based on their interpersonal distances. As the repelling fields of agents overlap in our simulation, the observed agent should either evade or go through the gap left by the obstructing pair. The combined dips of the overlapping repelling fields reduce the utility of the path directly between the two pedestrians, making evasion to the right or left more desirable. As the distance between the agents increases, the utility of the middle path improves until the agent opts to go through the middle.

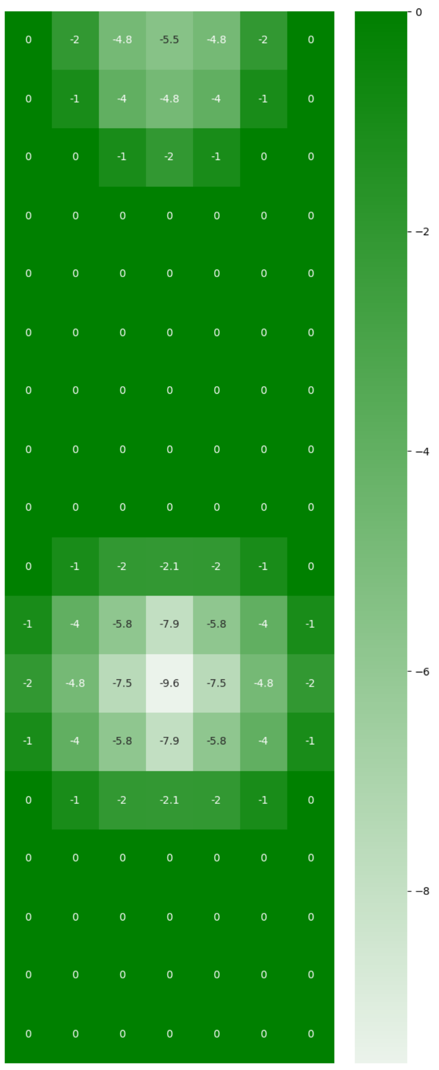
There are two possible ways to influence the results of this experiment. We can for one crank up the calculated values of the dip function, depending on the extrovertedness of the person going through the obstructing pair and chnage the distance between these two people. In the case of this simulation where there is a fixed range of rings around a person where the repelling field takes affect, we can ssay with confidence that after the obstruction pair is having a distnace of five fields between them, there is no force acting at all as the repelling field is not coded this way, therefore the agent in questio will just go trough the middle. But if th obstructiong pairs fields overlap and therefore decrese the likelyhood of an agent going throughj the middle. We can observe that if the dip functions returned values arent significant enough, the perosn just goes through th epair most of the time, as with the dijkstra algrithm the over all cost of reaching a target matters and not personla preference as there is no personal preferene coded into the agents acting.

There are two ways to influence the experiment results: increase the dip function values based on the supposed introvertedness of the person passing through the obstructing pair and change the inter-personal distance between them. In the simulation, with a fixed range of the repelling field around a person - here two -, if the distance between the obstruction pair is five fields or more, no force acts, and the agent will go through the middle as there is no impacting force whatsoever. If the repelling fields overlap, and the dip function's values are still insignificant, the person goes through the pair due to the Dijkstra algorithm prioritizing overall cost to reach a target, not personal preference. Only if the values calculated be the dip function are significant enough, the agent will decide to evade the obstructing pair.

In the case of Figure 25 the agent decided to trough the gap, as the overall cost would have been higher when evading the situation. After increasing the dip function's return values, the agent randomly evades either right or left due to symmetrical force distribution.



(a)



(b) Dip values from the obstructing agent pair and the test agent that decides whether to go through the gap.

Figure 25. Starting position of the reenactment of the evasion experiment and the repelling fields acting upon eachother.

4. Discussion and Conclusion

The implemented simulation creates agents, simulates their movements with various strategies, and navigates around obstacles. Agents dynamically influence each other using dips, and a simple point-and-click map generator was used to test different RiMEA guideline-specified cases, including room evacuation with two and four doors, constant free flow speed, the chicken test, and the fundamental diagram using density, flow, and velocity from the RiMEA 4 test case. The results look promising but lack scale; a larger map or more agents might have better reflected changes in the agents' movements.

Defining an agent's dips is challenging because, in reality, personal space is variable rather than a fixed range with a repelling field. This assumption simplifies the model for control purposes and reproducibility. Adding even minor variability could make results less structured, as the negative energy field varies among individuals.

Another aspect was optimizing algorithms for efficient performance. In the case of the chicken test, it was anticipated that agents near the edges would be aggressively bypassed by others to expedite reaching their targets. Future improvements could include considering not only connection weights between fields but also the time agents have waited, potentially reducing delays caused by obstructing agents.

Given that most tests, plots, and statistics support our model's logic, we believe that a review of the code will help identify the errors causing these outcomes, bringing the results fully in line with theoretical expectations.

4.1. Free flow speed

The free flow speeds remain constant regardless of whether the agent moves diagonally or straight forward, with velocity and distance covered accurately adjusted for the grid structure. The generated free flow speeds align with the specified parameters of the normal distribution. The drawn values are tested for adherence using the Kolmogorv-Smirnov Test.

4.2. Chicken test

The chicken test demonstrated that employing different movement strategies—specifically, euclidean distance and Dijkstra algorithm—yielded distinct outcomes. While the euclidean distance approach resulted in agents forming a crowded bell shape against the obstacle wall without reaching the target, using the Dijkstra algorithm enabled every agent to reach their target, forming a ball-shaped formation on both sides of the U-shaped obstacle. However, friction along the encapsulation edges significantly slowed down the evacuation process while using dijsktra. Despite its simulation success, Dijkstra is computationally intensive and prolongs the real time simulation duration.

4.3. Room evacuation

The evacuation tests with two and four doors were successful. As anticipated, evacuation was faster with four doors compared to two. The tests utilized the Dijkstra algorithm, revealing bell-shaped crowds around points of interest, namely the exits, similar to previous findings, caused by some congestion that was also noted near the edges of the u-shaped obstacle during the chicken test.

4.4. Fundamental Diagram

The evacuation tests provide accurate data for velocity, as it stabilizes after 80 seconds, and for density and flow, as they represent the agents passing through the corridor. However, the relationship between these metrics does not align with the results depicted in Weidmann's diagrams.

4.5. Evasion Experiment

In the experiment, we observed how agents decide to evade or go through a gap based on the overlapping repelling fields of obstructing pairs. The utility of the middle path increases as the distance between agents grows, making it more likely for the observed agent to go through the middle. There are two ways to influence these results: adjusting the dip function values based on the agent's extrovertedness and changing the distance between the obstructing pair. When the repelling fields overlap, evasion is more likely unless the dip function values are very insignificant. The Dijkstra algorithm prioritizes overall cost, leading the agent to go through the middle unless significant forces prompt evasion. Increasing the dip function's values causes the agent to randomly evade right or left due to symmetrical force distribution of the repelling field.

References

1. Kolmogorov–Smirnov Test. In *The Concise Encyclopedia of Statistics*; Springer New York: New York, NY, 2008; pp. 283–287. https://doi.org/10.1007/978-0-387-32833-1_214.
2. DSA Dijkstra's Algorithm; 2024. https://www.w3schools.com/dsa/dsa_algo_graphs_dijkstra.php.
3. Dietrich, F.; Köster, G.; Seitz, M.; Sivers, I. Bridging the Gap: From Cellular Automata to Differential Equation Models for Pedestrian Dynamics. *Journal of Computational Science* **2014**, *5*. https://doi.org/10.1016/j.jocs.2014.06.005.