Exercise sheet 1
Modeling of human crowds

Due date: 2024-10-31
Tasks: 5

In this exercise, you will learn how to model and simulate a human crowd with a simple cellular automaton. In general, cellular automata are a computational tool to model a complex system [13, 12], i.e., a system that is built up of many interacting parts. Typically, each individual part is relatively simple, but the interactions can lead to very complicated behavior. A standard example of a cellular automaton with simple rules that lead to interesting behavior is Conway's "game of life" [4]. A crowd of humans can also be modeled with a cellular automaton, where individuals are placed in single cells and interact through simple rules, with each other, with obstacles, and with their targets. There are many other ways to model crowds, and we will discuss some of them during the course. You can find an overview of models and implementation details in the literature [3, 6], and several parts in the book of Boccara [1] discuss cellular automata more generally.

1 State space

In cellular automata we will use, the full state of the system is contained in the states of individual cells. In a crowd simulation, these cells are typically arranged in a two-dimensional grid (see Figure 1). A possible state space X_i for a single cell i may be

$$X_i := \{E, P, O, T\}, \tag{1}$$

where the symbols for a state are interpretable as

1. E: empty cell,
2. P: there is a pedestrian in this cell,
3. O: there is an obstacle in this cell,
4. T: this cell is a target for the pedestrians in the scenario.

If we arrange the cells of the cellular automata in a grid as shown in Figure 1, the state space of the entire system would be $X = \{E, P, O, T\}^{5\times5}$, i.e. 25 different cells with E, P, O, or T as their current state.

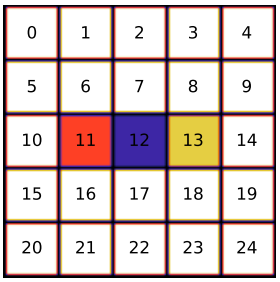

Figure 1: The state space of a cellular automaton, with three cells marked in different colors (red: pedestrian, blue: obstacle, yellow: target).

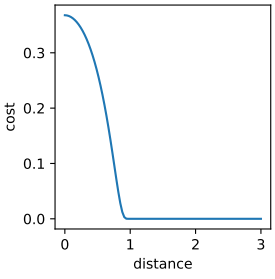

Figure 2: A typical cost function $c(r)$ modeling the interaction between two individuals at distance r.

2 Simulation

The simulation of a "crowd", here, defined as "all pedestrians in the scenario", can be done in many different ways. For the cellular automaton we discuss here, the concept of an update scheme is important. You can read more about update schemes in the literature [9], but for this exercise, a simple, discrete-time update scheme with constant time shifts suffices. This update scheme for the cellular automaton can be defined as follows:

1. Let $x^{(n)} \in X$ be the state of the system (the automaton) at time step n.
2. Define $x^{(n+1)} = f(x^{(n)})$ as the next state of the system, with the time step $n+1$ a fixed time shift $\Delta t \in \mathbb{R}$ after time step n and $f : X \to X$ a map between system states (the "evolution operator", defined below).
3. Reset the system state to $x^{(n+1)}$, advance n by one, and continue with step (1).

The evolution operator f of the cellular automaton can use all the information currently available to advance the system to the next time step. For a scenario with only one cell being in state P (one pedestrian in the scenario), and one cell being in state T (one non-absorbing target), the evolution operator may act like this:

1. If there is no cell in state T (the pedestrian has reached the target), return the current state space unchanged.
2. Else, for each cell in state P, define the neighboring cells N_P as a list of states.
3. Compute the distance from all neighbors in N_P to the cell in state T through

$$d(c_{N,ij}, c_{T,kl}) = \sqrt{(i-k)^2 + (j-l)^2},$$

that is, the Euclidean distance between the cell indices ij and kl, where i and k are the row indices, and j and l are the column indices of the cells. The neighboring cell is called c_N, and the target cell is called c_T.

4. Set the state of the cell in state P to the state E, and set the cell with the smallest distance to the target cell from state E to state P. Return this new state space.

Of course, the evolution operator becomes more complicated when there are more pedestrians and obstacles present in the scenario.

A more sophisticated (but also more useful) way to update the state is the use of a utility function $u : I \times X \to \mathbb{R}$. This function takes a cell index in the index set I as well as the current state of the cellular automaton, and results in a utility at the given index. The evolution operator f then only needs to check the neighboring cells of a given cell for the value of u, and move the pedestrian to the cell with the highest utility (which may also be the current cell, i.e. the pedestrian does not move at all). An example of a utility function (or rather, a cost function!) is shown in Figure 4. Here, the cost function just returns the distance to the target cell. Interactions of individuals with others or obstacles in the environment are typically modeled through utility or cost functions that depend on the distance to other pedestrians in addition to the distance to the target. Figure 2 shows the following cost function for the interaction between two individuals, which can simply be added to a cost function for the target to obtain simple avoidance behavior. The parameter $r_{\max}$ can be adjusted to change the avoidance behavior:

$$c(r) = \begin{cases} \exp\left(\frac{1}{r^2 - r_{\max}^2}\right) & \text{if } r < r_{\max} \\ 0 & \text{else} \end{cases} \tag{2}$$

You do not need to implement this version of the cost function in this exercise, and a simple evolution operator described above is sufficient.

Note: the number of points per exercise is a rough estimate of how much time you should spend on each task.

Task 0/5: Good coding practices **Points: 25/100 (tests: 0, manual: 25)**

Your code should not only be correct but also well-written. Thus, you can get points for following good coding practices.

Checklist:

- ☐ Proper document the code with docstrings.
- ☐ Keep the code modular.
- ☐ Use meaningful naming.

Task 1/5: Setting up the modeling environment **Points: 5/100 (tests: 3, manual: 2)**

The provided template defines a simple GUI for the simulation shown in Figure 3. It also provides basic classes for the simulation elements. In this task, you are asked to complete the code and implement adding pedestrians, targets and obstacles to the grid in `Simulation.__init__()`. You should also implement the `Simulation.get_grid()` method to return the grid for the visualization. You can use a configuration in `configs/toy_config.json` to visually test your implementation.

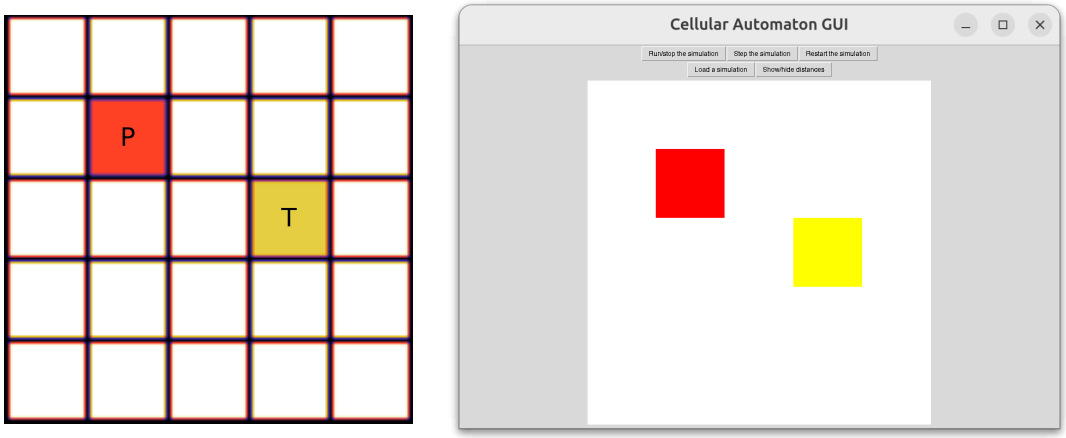

Figure 3: Left: An example visualization of the state of a cellular automaton with 25 cells, 23 of which are empty (white), one in state P at position (1,1) and one in state T at position (3,2). Right: The graphical user interface from the template.

Checklist:

- ☐ Implement adding elements into cells.
- ☐ Return a grid with all elements for visualization.
- ☐ Describe the setup and your implementation in detail.

Task 2/5: First step of a single pedestrian **Points: 10/100 (tests: 5, manual: 5)**

In this task, you should make a pedestrian move toward a target. For this, you may need to implement additional functions or add new fields to the existing classes.

As a test, define a scenario with 50 by 50 cells (2500 in total), a single pedestrian at position $(5, 25)$ and a target 20 cells away from them at $(25, 25)$. Then, simulate the scenario with your cellular automaton for 25 time steps, so that the pedestrian moves towards the target and waits there. As there are no obstacles, you can use the naive distance computation for this task.

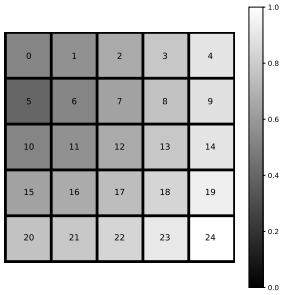

Figure 4: Distances to a target in cell 5 stored in the grid.

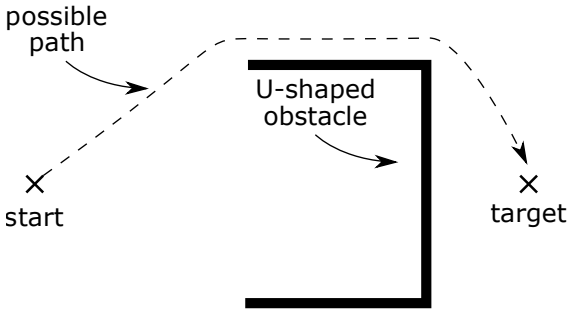

Figure 5: The "chicken test". Without obstacle avoidance, pedestrians will get stuck and cannot reach the target.

Checklist:

- ☐ Implement the `_get_neighbors()` method.
- ☐ Implement the `update_step()` method.
- ☐ Define and simulate the scenario above.
- ☐ Describe the setup and your implementation in detail (don't forget the figures!).

Task 3/5: Interaction of pedestrians — Points: 15/100 (tests: 6, manual: 9)

Now the pedestrians have to interact with each other. Additionally, this task will test if pedestrians can correctly move in arbitrary directions.

Insert five pedestrians in a large, square scenario. Place the pedestrians (approximately) equally spaced (i.e., at angles of $\alpha = i/5 * 360$ degree, $i = 0, \ldots, 4$) on a circle in a fairly large distance around a single target in the center of the scenario. You should make the target "absorbing" so that pedestrians will be removed from the scenario once they reach it.

Run the scenario and report your findings. What is the configuration of the pedestrians around the target after the simulation? Do the pedestrians all reach the target roughly at the same time? They should because they start at the same distance! Be careful that "distance" here does not mean "number of cells", but Euclidean distance. If not, implement a way to correctly traverse the space in arbitrary directions with roughly the same speed. You also should be careful not to let two pedestrians occupy the same cell (even if it is the target one!).

Checklist:

- ☐ Implement pedestrian avoidance.
- ☐ Implement speed adjustment.
- ☐ Define and simulate the scenario above.
- ☐ Describe the setup and your implementation in detail (don't forget the figures!).

Task 4/5: Obstacle avoidance — Points: 15/100 (tests: 5, manual: 10)

Up to now, there were no obstacles in the path to the target. Implement rudimentary obstacle avoidance for pedestrians by adding a penalty (large cost in the cost function) for stepping onto an obstacle cell. What happens in the scenario shown in Figure 10 of [7] (bottleneck)? What happens for the "chicken test" scenario, Figure 5?

Implement the Dijkstra algorithm to flood the cells with distance values, starting with zero distance value at the target, such that obstacle cells are not included in the set of possible cells. Can the pedestrians reach the targets now? If the cell is an obstacle or it is unreachable from the target, you should set the distance value for this cell to infinity.

Note: a more accurate computation of the distance field is possible through the Fast Marching algorithm by Sethian [10, 5, 11]. A report on its implementation[1] was written by Bærentzen [2].

Checklist:

- ☐ Implement obstacle avoidance.
- ☐ Define and simulate the "bottleneck" and the "chicken test" scenarios.
- ☐ Implement the Dijkstra algorithm and re-run the simulations using it.
- ☐ Describe the setup and your implementation in detail (don't forget the figures!).

Task 5/5: RiMEA tests **Points: 30/100 (tests: 6, manual: 24)**

After you implemented pedestrian and obstacle avoidance, you have to test your implementation with the following scenarios from the RiMEA guidelines[2]. They provide support for verification and validation of simulation software for crowds [8]. The tests in the guideline may contain features that are not implemented in your cellular automaton. Discuss why you need to implement them to complete the test, or why you can neglect the particular feature and still obtain reasonable test results. For example, the premovement time can be ignored for the tests below (why?). But the body size (e.g., the cell size in meters) is needed to complete the tests (why?).

RiMEA scenario 1: moving in a straight line with correct speed.

RiMEA scenario 4: plotting a fundamental diagram. For this, you need to complete the `MeasuringPoint` class that will measure pedestrian flow passing through the point. You can run into computational issues in this test, so it is possible to reduce the size of the scenario to complete the task.

RiMEA scenario 6: moving around a corner.

RiMEA scenario 7: simulating demographic parameters. In this test, you are asked to simulate pedestrians with speeds depending on their age. The data in `configs/rimea_7_speeds.csv` contains an approximation of the mean speed for each age. You can use this data to sample your pedestrians' parameters.

Checklist:

- ☐ Discuss why you need to implement or ignore some features.
- ☐ RiMEA scenario 1 (straight line).
- ☐ RiMEA scenario 4 (plotting a fundamental diagram: density vs. flow).
- ☐ RiMEA scenario 6 (movement around a corner).
- ☐ RiMEA scenario 7 (demographic parameters).
- ☐ Describe the setup and your implementation in detail (don't forget the figures!).

[1] Also see `http://en.wikipedia.org/wiki/Fastmarchingmethod`.
[2] Find the guidelines here: `https://rimeaweb.files.wordpress.com/2016/06/rimea_richtlinie_3-0-0_-_d-e.pdf`

References

[1] Nino Boccara. *Modeling Complex Systems*. Graduate Texts in Physics. Springer, 2 edition.

[2] J. Andreas Bærentzen. On the implementation of fast marching methods for 3d lattices.

[3] Felix Dietrich, Gerta Köster, Michael Seitz, and Isabella von Sivers. Bridging the gap: From cellular automata to differential equation models for pedestrian dynamics. 5(5):841–846.

[4] Martin Gardner. Mathematical Games. 223(4):120–123.

[5] R. Kimmel and J. A. Sethian. Computing Geodesic Paths on Manifolds. In *Proceedings of the National Academy of Sciences of the United States of America*.

[6] Robert Lubaś, Jakub Porzycki, Jaroslaw Was, and Marcin Mycek. Validation and verification of CA-based pedestrian dynamics models. 11:285–298.

[7] RiMEA. *Guideline for Microscopic Evacuation Analysis*. RiMEA e.V., 3.0.0 edition.

[8] RiMEA. *Richtlinie Für Mikroskopische Entfluchtungsanalysen(RiMEA),*. RiMEA e.V., 2.2.1 edition.

[9] Michael Seitz, Gerta Köster, and Alexander Pfaffinger. Pedestrian Group Behavior in a Cellular Automaton. In Ulrich Weidmann, Uwe Kirsch, and Michael Schreckenberg, editors, *Pedestrian and Evacuation Dynamics 2012*, pages 807–814. Springer International Publishing.

[10] J. A. Sethian. A fast marching level set method for monotonically advancing fronts. 93(4):1591–1595.

[11] J. A. Sethian. *Level Set Methods and Fast Marching Methods: Evolving Interfaces in Computational Geometry, Fluid Mechanics, Computer Vision, and Materials Science*. Cambridge University Press.

[12] Stephen Wolfram. Cellular automata as models of complexity. 311:419–424.

[13] Stephen Wolfram. Statistical mechanics of cellular automata. 55(3):601–644.