Exercise sheet 1
Modeling of human crowds

Due date: 2021-11-04
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 [12, 13], 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 for 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 modelled 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], several parts in the book of Boccara [2] 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.

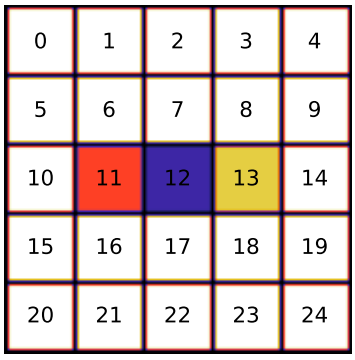

Figure 1: The state space of a cellular automata, with three cells marked in different color (red: pedestrian, violet: obstacle, yellow: target).

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 target), the evolution operator may act like this:

1. For each cell in state P, define the neighboring cells N_P as a list of states.
2. If any cell N_P is in the state T, return the current state unchanged.
3. Else, 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.

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 for 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 is typically modelled 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:

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

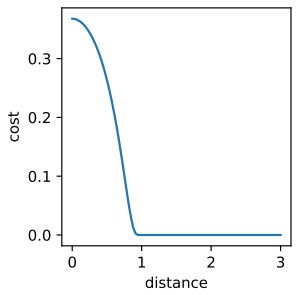

Figure 2: A typical cost function $c(r)$ modeling the interaction between two individuals at distance r.

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

Task 1/5: Setting up the modeling environment **Points: 25/100**

Setup your implementation and simulation environment with the following requirements:

1. Basic visualization.
2. Adding pedestrians in cells.
3. Adding targets in cells.
4. Adding obstacles by making certain cells inaccessible.
5. Simulation of the scenario (being able to move the pedestrians).

It is not necessary to implement a graphical user interface, where a user can change pedestrians, targets, and obstacles. Each scenario may be defined in code, even though I strongly advise you to think about an efficient way to create new, more complex scenarios. It is important to be able to change the scenario without changing the code (e.g., through a command line interface, or a rudimentary scenario file that can be read in). A helpful tutorial on a more complex grid structure can be found at the Red Blob Games blog entry for hexagonal grids `https://www.redblobgames.com/grids/hexagons/`. You do not need to implement a hexagonal grid, a square cell grid is sufficient. Figure 3 illustrates a possible result of this task, i.e. a visualization of the state of a cellular automaton with a single pedestrian and a target, in a grid of 25 cells.

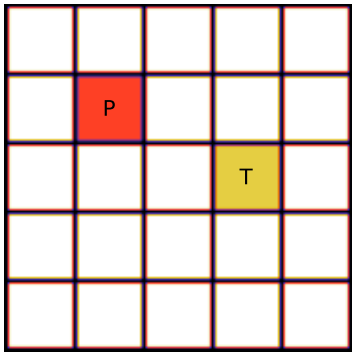

Figure 3: An example visualization of the state of a cellular automaton with 25 cells, 23 of which are empty (white), one in state P at position (2,2) and one in state T at position (4,3).

Checklist:

- ✔ Basic visualization done?
- ✔ Adding pedestrians, targets in cells possible?
- ✔ Adding obstacles possible, by making certain cells inaccessible?
- ✔ Simulation of the scenario (being able to move pedestrians)?
- ✔ Is it possible to change the scenario without changing the code (through GUI or CLI)?
- ✔ Verbose description of setup in the report?
- ✔ **Code:** Modular and clear, e.g. split into Python files and notebooks?
- ✔ **Code:** Concise documentation everywhere, e.g. with Python docstrings?

Task 2/5: First step of a single pedestrian **Points: 5/100**

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 $(25, 25)$. Simulate the scenario with your cellular automaton for 25 time steps, so that the pedestrian moves towards the target and waits there.

Checklist:

- ✔ Scenario with 50 by 50 cells (2500 in total).
- ✔ Start a single pedestrian at position (5, 25) and a target 20 cells away from them at (25, 25).
- ✔ Simulate the scenario with your cellular automaton for 25 time steps.
- ✔ Description of experiment and results in the report, including figures?

Task 3/5: Interaction of pedestrians **Points: 20/100**

Now the pedestrians have to interact with each other. Insert five pedestrians in the previous scenario from task 2, arranged in a circle in a fairly large distance (30-50m) around a single target in the center of the scenario. Run the scenario and report your findings. What is the configuration of the pedestrians around the target after the simulation? It is also possible to have them removed entirely, if you made your target "absorbing". Do the pedestrians all reach the target roughly at the same time? They should, because they start at the same distance! If not, implement a way to correctly traverse the space in arbitrary directions with roughly the same speed.

Checklist:

- ✔ Insert five pedestrians in the scenario of task 2, arranged in a circle around a single target.
- ✔ What is the configurations of the pedestrians around the target after the simulation?
- ✔ Verbose description of experiment and results in the report?
- ✔ **Code:** Modular and clear, e.g. split into Python files and notebooks?
- ✔ **Code:** Concise documentation everywhere, e.g. with Python docstrings?
- ✔ **Code:** Speed of individual pedestrians adjusted if they move diagonally?

Task 4/5: Obstacle avoidance **Points: 10/100**

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 [8] (bottleneck), if obstacle avoidance is not implemented? What happens for the "chicken test" scenario, figure 5?

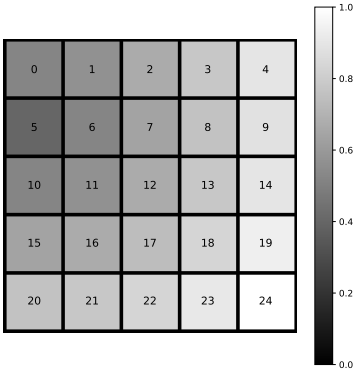

Figure 4: Distances to a target in cell 5 stored in the grid.

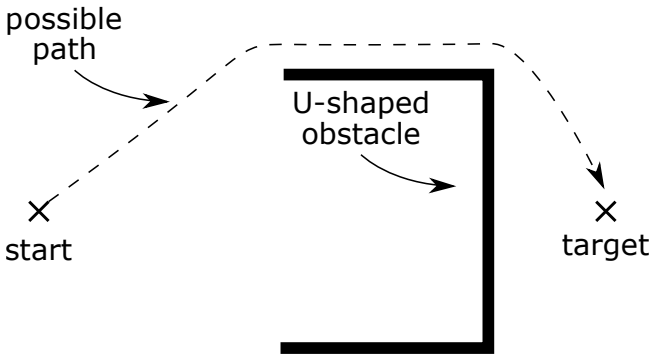

Figure 5: The "chicken test". Without obstacle avoidance, pedestrians will get stuck and cannot reach the target.

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? 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 [1]. An implementation of the Fast Marching Method is not necessary but will give 10 bonus points (if it is properly tested). You do not need the Dijkstra algorithm if you implement the Fast Marching algorithm

Checklist:

- [x] Implement rudimentary obstacle avoidance for pedestrians.
- [x] What happens if obstacle avoidance is not implemented?
- [x] What happens for the "chicken test" scenario, figure 5?
- [x] Verbose description of experiment and results in the report?
- [x] **Code:** Modular and clear, e.g. split into Python files and notebooks?
- [x] **Code:** Concise documentation everywhere, e.g. with Python docstrings?
- [ ] **Bonus:** Fast Marching algorithm implemented, tested, and described?

[1] Also see `http://en.wikipedia.org/wiki/Fastmarchingmethod`.

Task 5/5: Tests **Points: 40/100**

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 [7]. 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. A good example for a feature that can be ignored for the tests below is the premovement time (why?).

TEST1: RiMEA scenario 1 (straight line, ignore premovement time)

TEST2: RiMEA scenario 4 (fundamental diagram, be careful with periodic boundary conditions).

TEST3: RiMEA scenario 6 (movement around a corner).

TEST4: RiMEA scenario 7 (demographic parameters, visual comparison of figure and results is sufficient. Simple and correct statistical test gives 5 bonus points).

Checklist:

- ☐ TEST1: RiMEA scenario 1 (straight line).
- ☐ TEST2: RiMEA scenario 4 (fundamental diagram: density vs. velocity or density vs. flow).
- ☐ TEST3: RiMEA scenario 6 (movement around a corner).
- ☐ TEST4: RiMEA scenario 7 (demographic parameters).
- ☐ Discussed why you need to implement features, or why you can neglect them?
- ☐ Verbose description of experiments and results in the report?
- ☐ **Code:** Modular and clear, e.g. split into Python files and notebooks?
- ☐ **Code:** Concise documentation everywhere, e.g. with Python docstrings?

References

[1] J. Andreas Bærentzen. On the implementation of fast marching methods for 3d lattices. Technical report, Technical University of Denmark, 2001.

[2] Nino Boccara. *Modeling Complex Systems*. Graduate Texts in Physics. Springer, New York, 2nd edition, 2010.

[3] Felix Dietrich, Gerta Köster, Michael Seitz, and Isabella von Sivers. Bridging the gap: From cellular automata to differential equation models for pedestrian dynamics. *Journal of Computational Science*, 5(5):841–846, 6 2014.

[4] Martin Gardner. Mathematical games. *Scientific American*, 223(4):120–123, 1970.

[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*, 1998.

[6] Robert Lubaś, Jakub Porzycki, Jarosław Was, and Marcin Mycek. Validation and verification of ca-based pedestrian dynamics models. *Journal of Cellular Automata*, 11:285–298, 2016.

[2] Find the guidelines here: `https://rimeaweb.files.wordpress.com/2016/06/rimea_richtlinie_3-0-0_-_d-e.pdf`

[7] RiMEA. *Richtlinie für Mikroskopische Entfluchtungsanalysen(RiMEA),*. RiMEA e.V., 2.2.1 edition, March 2009. `www.rimea.de`.

[8] RiMEA. *Guideline for Microscopic Evacuation Analysis*. RiMEA e.V., 3.0.0 edition, 2016. `www.rimea.de`.

[9] Michael J. Seitz and Gerta Köster. How update schemes influence crowd simulations. *Journal of Statistical Mechanics: Theory and Experiment*, 2014(7):P07002, 2014.

[10] J. A. Sethian. A fast marching level set method for monotonically advancing fronts. *Proceedings of the National Academy of Sciences*, 93(4):1591–1595, 1996.

[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, Cambridge, 1999.

[12] Stephen Wolfram. Statistical mechanics of cellular automata. *Review of Modern Physics*, 55:601–644, 1983.

[13] Stephen Wolfram. Cellular automata as models of complexity. *Nature*, 311:419–424, 1984.