Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Lecture with Computer Exercises:
Modelling and Simulating Social Systems with Python

Project Report

Implementation of a Traffic Light System at the Tannenbar-Intersection

Nico Burger, Leo Fent, Jérôme Landtwig & Pascal Lieberherr

Zürich,
December 2018

Agreement for free-download

We hereby agree to make our source code for this project freely available for download from the web pages of the SOMS chair. Furthermore, we assure that all source code is written by ourselves and is not violating any copyright restrictions.

Nico Burger	Leo Fent	Jérôme Landtwig	Pascal Lieberherr

.... Declaration of Originality needs to be added

Table of Contents

ADD APPENDIX A.

1. Abstract

Authors: Nico Burger, Leo Fent, Jérôme Landtwig, Pascal Lieberherr

Title: Implementation of a traffic light system at the intersection between Tannen- and Universitätsstrasse

In this project we want to have a closer look at the intersection between Tannenstrasse and Universitätsstrasse. Also known as the Tannenbar Intersection. We were motivated by our own experiences. During lecture breaks, early in the morning and at noon a lot of students desire to cross the street. While pedestrians just keep crossing the street cars start to line up. As a consequence, congestion and traffic holdups can be observed. Due to this situation, we asked ourselves whether traffic lights might improve the waiting time for cars. At the same time the waiting time for pedestrians should not be too high. To do this, we simulate pedestrians, cars and trams by agents with Python who behave according to the swiss traffic rules.

What needs to be added to this chapter next depends on the results of our simulation. This will be added once we know our results and our discussion

This document and all mentioned data can be downloaded from https://github.com/jerowing/gess_project (09.12.2018)

2. Individual contributions

Nico Burger:	path of agents and agent interaction
Leo Fent:	map, visualization, coordination
Jérôme Landtwig:	visualization, graphics and plots
Pascal Lieberherr:	path of agents, spawning of agents, project report, flash talk

Listed above are only the main tasks everyone of us took care of, but we shared most of the work. The github commit report is not always mirroring the work behind those uploads because we often worked together on the code, debugged, commented and worked on the documentation while swapping laptops.

3. Motivation

The intersection between the Tannenstrasse and the Universitätsstrasse is something all ETH students are well familiar with and so we are. It is located in a traffic hotspot with pedestrians, cars and trams. All use the road simultaneously. This leads to conflict especially at peak times. As we cross this Intersection close to every day we started to think about how the occurring traffic jam during peak times could be reduced. Lecture breaks, afternoon, noon and lecture breaks are considered as peak time. What came across our mind first was a traffic light system that controls the flow of the agents (pedestrians, cars and trams). This course offers the perfect opportunity to simulate the Tannenbar-Intersection with a traffic light system. While observing the crosswalks for a while, we observed special dynamics e.g. cars that hardly stop for pedestrians waiting at the crosswalk but also cars stopping way too early for pedestrians and also pedestrians who insist on their right of way in every situation. Based on these experiences we became curios about the effects of a traffic light system controlling the agents.

Add Picture of the Tannenbar Intersection

3.1 Fundamental Questions

We pose three guiding questions for our idea: How good is the current solution? As described earlier, large traffic jams can be observed during peak times. Is this necessary for smooth pedestrian flow? What would change if pedestrians stopped once in a while to let cars pass? Could the situation be improved by adding traffic lights? Traffic lights would force either the cars of the pedestrians to let the opposite party pass. How would that affect car waiting time? How much longer would pedestrians have to wait? What would be the best possible solution for all agents? Is there an ideal solution (e.g. using traffic lights only at peak times)? What would it look like?

3.2 Expected Results

To answer our three fundamental question, we will run different simulations (see Table 1). Each simulation represents a certain time during the day with appropriate car and pedestrian densities. Each simulation will be performed with and without traffic lights. Furthermore, we will also vary the green time for both agents. This way we can compare the current situation, without traffic lights, with the newly implemented traffic light system. We expect the following different results for each scenario:

Scenario 1: Since we have high density for both cars and pedestrians, we first of all await an improvement in flow for cars when traffic lights are on. Whereas pedestrians face an increased waiting time. Nevertheless, we expect an overall improvement due to the fact that the waiting time for cars can be reduced significantly while pedestrians waiting time increases slightly. To find the optimum we will also vary the green time for cars and pedestrians, respectively. We expect that the green time split up evenly between cars and pedestrians will lead to the best results.

Scenario 2: For low car and high pedestrian density we don't expect such a clear result as for high car and high pedestrian density. As we have a high car density it is still likely that cars start to line up. Hence, we think that in this case traffic lights might be unnecessary. Simply because they don't improve the smooth agents flow significantly but increases the pedestrian waiting time a lot.

Scenario 3: In this case we expect similar results as for scenario 2. Namely, the flow doesn't get significantly better while the pedestrian waiting time increases. The only traffic light configuration that would make sense is one with more green time for the pedestrians

Scenario 4: Scenario 4 is expected to be an obvious case again. We think traffic lights are unnecessary since they don't affect the agent flow in any case because of low density for both cars and pedestrians.

Scenario	Time of day	Car density	Pedestrian density
1	08:00	High	High
2	12:00	High	Low
3	17:00	Low	High
4	20:00	Low	Low

Table 1: A Summary of the different scenarios. Each simulation will be performed twice, once with and once without traffic lights.

4. Description of the Model

4.1 Matrix

In order to keep track of where our agents are, we implemented a matrix (100x100) where each entry represents a field/box in our simulation. This matrix is adjusted whenever an agent makes a move, or the traffic lights switch their state.

We mainly used the matrix to regulate the interactions between our agents. However, it also came in handy for counting the number of agents in a queue.

The following notation was used for the matrix entries:

0	Empty cell
1	Pedestrian
2	Car
3	Tram
4	Red Light (for pedestrians)
5	Green Light (for pedestrians)

4.2 Paths

In order to reduce the complexity of slanted paths, it seemed logical to simplify the problem by making the paths of the intersection strictly horizontal/vertical. Since we are only interested in the intersection (including the crosswalks) we extended the streets up to the boundaries of our matrix and do not care what happens before/after an agent enters our simulation.

With this in mind, we ended up with the following paths:

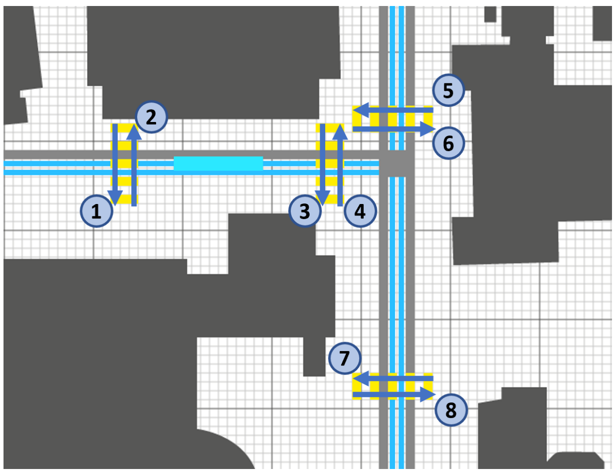

Figure 1: Pedestrian Paths

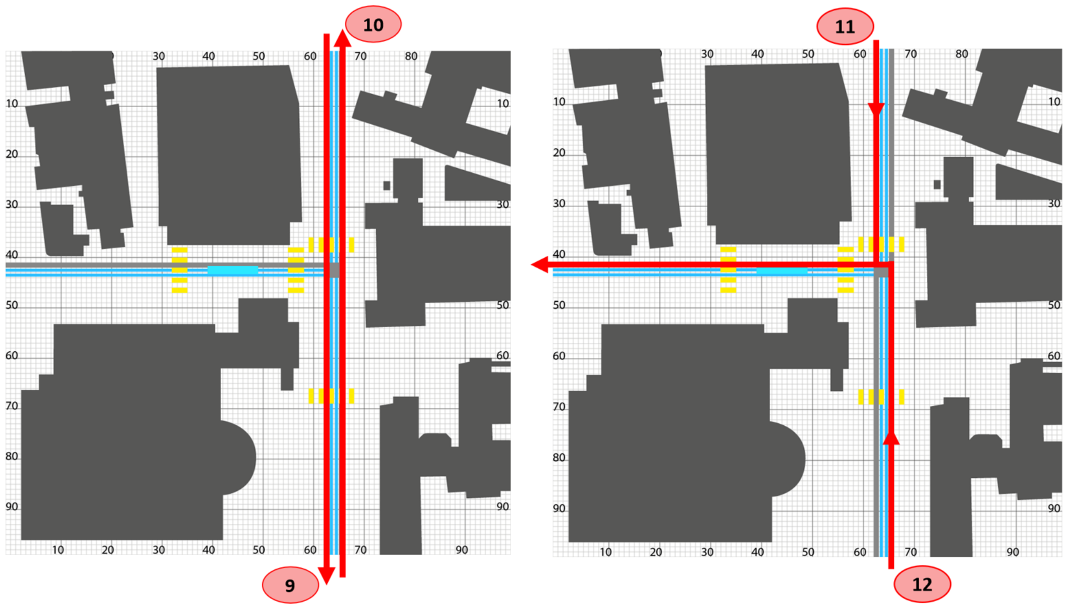

Figure 2: Car Paths

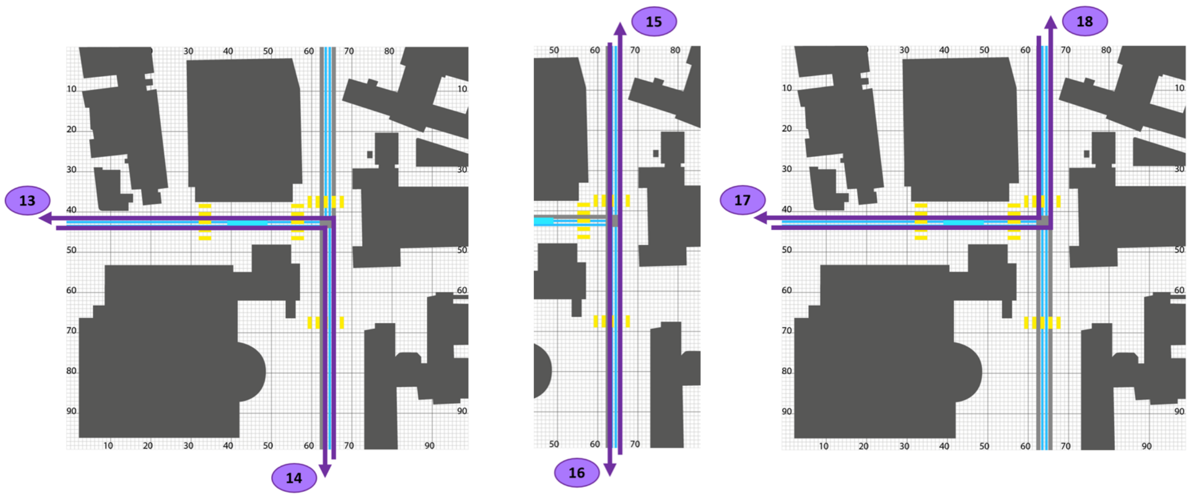

Figure 3: Tram Paths

Nr.	Crosswalks (Figure 1)	Nr.	Streets (Figure 2)	Nr.	Tram-Path (Figure 3)
1	Crosswalk_L_L	9	Car_L	13	6_Uni
2	Crosswalk_L_R	10	Car_R	14	6_Polybah
3	Crosswalk_R_L	11	Car_L_L	15	9_Uni
4	Crosswalk_R_R	12	Car_R_L	16	9_Haldenbach
5	Crosswalk_U_U			17	10_Haldenbach
6	Crosswalk_U_B			18	10_Polybahn
7	Crosswalk_B_U				
8	Crosswalk_B_B				

4.3 Agents

Our simulation would theoretically depend on countless agents of varying importance and attributes. Simplifying the problem, we narrowed it down to three agents: pedestrians, cars and trams. Therefore, we intentionally neglect all other potential agents which have a relatively small influence on the dynamics of the simulation (e.g. bicycles, ambulances). Simplifying the problem even more, we assume that all agents of the same type have unified attributes (e.g. speed, size). This in turn reduces the accuracy of our model. In order to reduce this inaccuracy, we concentrated on finding unified attributes that represent the general behavior of the type of agent.

By observing the intersection between Tannenstrasse and Universitätsstrasse we obtained the following generalizations for the speed of our agents:

• Pedestrian:	1 box / second
• Car:	6 boxes / second
• Tram:	3 boxes / second

Furthermore, we realized that for the simulation to be clear, pedestrians and cars should be of the same size, while trams are 8 times longer than a car. Therefore, pedestrians and cars have the size of a box and trams 8 consecutive blocks.

Lastly, we had to ensure that our agents get spawned randomly. We implemented this feature by giving every path a probability of an agent being spawned (for each second).

4.4 Interaction between Agents

Because pedestrians, cars and trams should not collide, we needed to ensure that our agents follow a set of rules in order to prevent this from happening. Mirroring the real world, we came up with the following 4 rules:

1. Pedestrians never have to yield/give way
2. Trams must give way only for pedestrians.
3. Cars have to give way to pedestrians and trams.
4. If an agent wants to move to point (x, y), it is only allowed to do so if point (x, y) is not occupied by another agent.

We are aware that some people violate these rules. To make our simulation applicable, we need to assume that each agent follows these rules.

4.5 Iteration Speed

One iteration is to be understood as one cycle, during which all agents are updated to their new positions in the matrix. Because the minimal step for an agent in our simulation is one box, we had to set the iteration speed to the amount of time at which our slowest agent moves from one box to the next. Our slowest agents are the pedestrians, which move at a speed of one box per second (see 4.3). Therefore, one complete iteration of our simulation takes one second.

4.6 Traffic lights

A central element of the simulation are the traffic lights which regulate whether the agents are allowed to continue on their path or have to stop. In order to simplify the model and due to time constraints we chose not to indulge in the complex topic of traffic planning and thus only to use two situations; one where all pedestrians have to wait and one where all cars have to wait whilst the pedestrians cross the street. Since we only have one street where cars pass and the crosswalks are relatively close together, we felt that this was a pretty good approximation of a real-life situation.

4.7 Graphical Output

In order to get a better understanding of the agents' behaviour, we chose to output the movements of our agents to a map. Additionally, we feel that the graphical reference made coding easier and the simulation more realistic.

Starting with a map by the Federal Institute for Topography *swisstopo,* a simplified version was drawn and adapted for our uses.

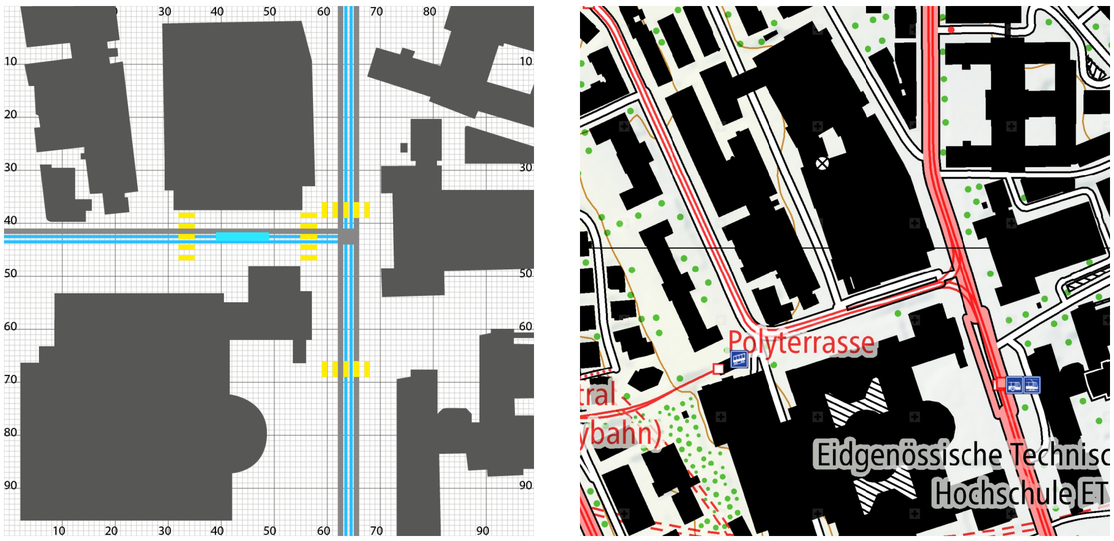

Figure 4: The simplified map used in the Simulations (left) and the original map by swisstopo[1] (right).

The agents were then placed onto the map on the paths determined by the map. Finally, the traffic lights were added to have a visual representation of the agents having to stop.

[1] (swisstopo, 2018)

5. Implementation

5.1 Matrix

In the program, the matrix is named "raster". To initialize the matrix, we used a function called "`initialize_gitter()`". Using the *numpy* library, the function creates a quadratic matrix (100x100) of type integer.

From then on, each iteration updates our matrix to contain information about the positions of our agents as well as the state of the traffic lights. This is achieved by plugging the matrix into all functions that update, either the positions of the agents or the states of the traffic lights.

5.2 Paths

When an agent is spawned, it receives a pathname. Within the three different agent classes, there are dictionaries, which contain the start- and endpoints of the different paths. For example, in class Pedestrians:

```
self.startposx = {‘crosswalk_L_L’: 33, ‘crosswalk_L_R’: 35, ... }
self.startposy = {‘crosswalk_L_L’: 38, ‘crosswalk_L_R’: 48, ... }
self.endposy = {‘crosswalk_L_L’: 33, ‘crosswalk_L_R’: 35, ... }
self.endposy = {‘crosswalk_L_L’: 48, ‘crosswalk_L_R’: 38, ... }
```

The agent therefore knows where it starts and where it has to end up at. The function "speed(agent)" is called which returns the direction (x, y coordinates) which the agent must take to reach its endpoint. The direction is found in the agent's class as "xspeed" and "yspeed". For example, in class Pedestrians:

```
self.xspeed, self.yspeed = speed(self)
```

For paths which require a change in direction a midpoint is given. The agent sees this as it's first endpoint. After reaching the midpoint, the agent's endpoint gets updated to its final destination, the direction is changed, and it proceeds towards its endpoint. If an agent does not have require a midpoint, the midpoint in the dictionary is a string "no" (for "no" midpoint) and it skips the previously stated process of using a midpoint as an intermediate endpoint.

5.3 Agents

The agents are stored in lists (walkers, drivers, tram). When a new agent is spawned, it's address is added to the end of the list, from which it can be accessed. When an agent reaches its destination its address in the list gets deleted.

The "iterate" function is called by plugging in a list (walkers, drivers or tram). This function then updates the positions of all agents in that list, hence, it iterates through the list. The "iterate" function is not to be confused with a complete iteration which has been defined above.

5.3.1 Agent Move

An agent makes a move, when it is called by the "move(agent, matrix)" function. This function checks which position the agent would like to move to and if that position is occupied already.

If the position is occupied by another agent (`entry != 0`), it leaves everything the way it was before the function was called. Otherwise, the "move" function:

1. sets the old position of the agent in the matrix to 0
2. sets the new position of the agent in the matrix to either 1, 2 or 3 (depending on the agent type)
3. updates the agents coordinates to the new position
4. moves the window object of the agent to the new position

Another thing it checks, is whether the agent is at its endpoint. If that is the case, the function returns "True", otherwise it returns "False". This in turn is used during the "iterate" function, which deletes the agent if the "move" function returns "True".

5.3.2 Agent Speed

One iteration moves our slowest agents, the pedestrians, once per second. As stated, we wanted the trams to be three times faster and the cars to be six times faster than the pedestrians. Therefore, we called the "iterate" function of the trams/cars 3/6 times respectively during one iteration so that the "move" function is called 3/6 times for each tram/car during one iteration. This results in the trams/cars being 3/6 times as fast as pedestrians. We implemented this in the main function:

```
# iteration of all agents for each second
# car_speed = 6
iterate(walkers, raster)
i = 1
while i <= car_speed:
     if i % 2:
          iterate(tram, raster)
     iterate(drivers, raster)
     tk.update()
     i += 1
```

5.3.3 Agent Spawning

For each iteration, there should be a possibility for one or more new agents to be spawned at each path. We created two parameters to code and work with this prerequisite. These parameters are:

1. The chance of an agent being spawned at a given path (given in percent)
2. The number of possible agents being spawned at a given path

Since we analyze 4 different scenarios, we have a global variable (`variante_zeiten`) which sets the spawning frequencies of each path with the two mentioned parameters. These spawning frequencies are adjusted to the different scenarios. Hence, at 08:00 o'clock the probabilities for an agent to spawn are higher than at 20:00 o'clock.

Calling the "spawn" functions once per iteration leads to new agents being created and added to the lists (walkers, drivers, tram) just as planned.

5.4 Interaction between Agents

In order to make our agents to follow the rules we initially set (see 4.3), we can use the order in which the "iterate" functions are called during one iteration to our advantage. By first iterating the pedestrians, second the trams and at last the cars, we achieve exactly this behavior. That is, because the matrix is updated after each iteration. Hence, if we first use the "iterate" function on the pedestrians, they will be the first to move.

The 4th rule is dealt with in the "move" function, which eliminates the possibility for an agent to move to a new position whilst another agent is occupying that position.

5.5 Traffic Lights

In each cell of the above described 100x100 Matrix information can be found about whether the next position of the agent is occupied is free. If so, the agent can move on to the next position. In order to stop pedestrians and cars from running red lights, the function `rotlicht()` is used. `rotlicht()` has three different cases as inputs:

- `rotlicht(1, matrix)`: All cells in front of the traffic lights are set to 0.
- `rotlicht(2, matrix)`: The cells in front of the pedestrian crossings are set to 4. This means that pedestrians recognize an "obstacle" in front of them and thus wait.
- `rotlicht(3, matrix)`: The cells in front of the cars at the pedestrian crossings are set to 5. This in turn means that cars have to wait and pedestrians can cross.

5.6 Graphical Output

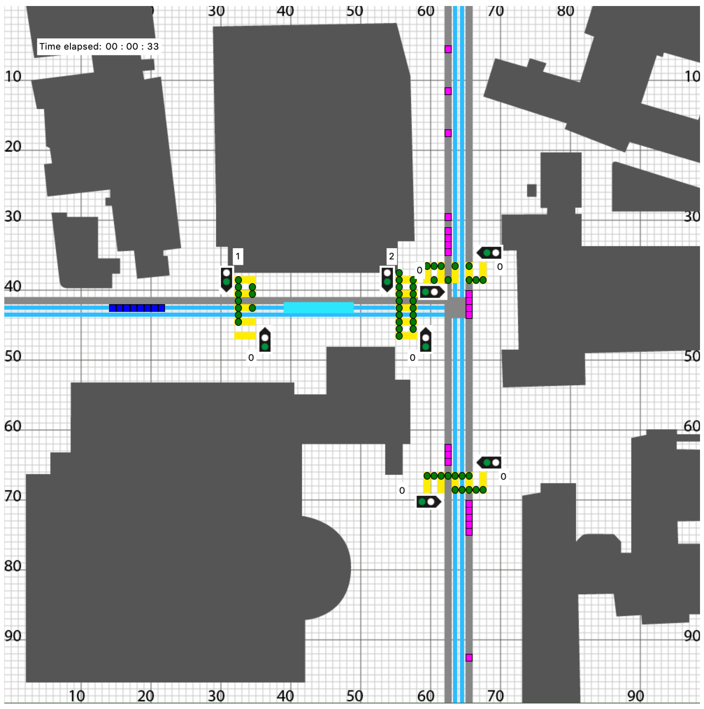

Figure 5: A running simulation.

5.6.1 Background and Agents

The graphical output was achieved using Python's *tkinter* module[2]. The above described map (Figure 4) was first drawn in Adobe Illustrator and then exported as gif. This made it possible to use it as a background for the tkinter window (later called canvas). On the canvas, the

[2] (Python Software Foundation, 2018)

agents were placed using the tkinter functions `.create_oval` and `.create_rectangle`. These functions are called each time an agent is moved using the `move()` or `move_tram()` function.

5.6.2 Traffic Lights

The traffic lights were again drawn as gifs in Adobe Illustrator and then placed as a label onto the canvas. In order to keep the graphical output as clean as possible, it was decided to only have traffic lights for pedestrians. For each crosswalk, when the pedestrians are allowed to walk, the lights change to green, which in turn means that cars have to wait.

5.6.3 Waiting Pedestrians

In order to quantify the amount of waiting the pedestrians have to do, we chose to count, how many pedestrians were stationary at their initial position. When calling the `move()` function, the amount of waiting pedestrians is updated for each crosswalk. The dictionary is then output to the canvas with the `display_waiters()` function.

The `display_waiters()` function takes the dictionary as an input and the places labels at the corresponding places on the canvas.

5.6.4 Elapsed Time Display

For the viewer to have a feeling of the speed of the simulation, we implemented a label which displays the amount of time passed. For ease of simulation and graphing, time is tracked by the global variable `i` of the main `for` loop. However, to display Time, three variables `t_s`, `t_min` and `t_h` are used which then get placed onto the canvas using the function `print_time()`.

6. Performed simulations

We wanted to analyze the impact of different traffic lights on our modeled system of the intersection. As the amount of pedestrians and of course the density of car drivers passing the intersection vary during the time of a day we chose to make measurements at four different times to get a birds eye view how good a version of a traffic light is in general. We chose the following generic timepoints:

6.1 Timepoints

6.1.1 Scenario 1

We chose the timepoint 8'o'clock as an example with a high pedestrian density and a high car density as well. This means that we set the chances of spawning pedestrians and cars higher than in low density cases. This is matches the reality pretty well as most of the students attend lectures that start at 8:15 and a lot of workman commute to work at this time. For this simulation we chose the following parameters.

```
# 08:00 [Chance, Amount] of Pedestrians/Cars spawned (per second)
amount_ped = {'crosswalk_L_L': [60, 1], 'crosswalk_L_R': [20, 1],
'crosswalk_M_L': [60, 1], 'crosswalk_M_R': [20, 1], 'crosswalk_U_U': [60,
1],
'crosswalk_U_B': [20, 1], 'crosswalk_B_U': [60, 1], 'crosswalk_B_B': [20,
1]}
amount_car = {'car_L': [60, 1], 'car_R': [60, 1]}
```

6.1.2 Scenario 2

Around lunchtime. Loads of studens leave their lectures and walk to a nearby mensa. Most of the workman stay at their institution which means that the density of cars passing our system is rather small.

```
# 12:00 [Chance, Amount] of Pedestrians/Cars spawned (per second)
amount_ped = {'crosswalk_L_L': [20, 1], 'crosswalk_L_R': [60, 1],
'crosswalk_M_L': [20, 1], 'crosswalk_M_R': [60, 1], 'crosswalk_U_U': [20,
1],
'crosswalk_U_B': [60, 1], 'crosswalk_B_U': [20, 1], 'crosswalk_B_B': [60,
1]}
amount_car = {'car_L': [30, 1], 'car_R': [30, 1]}
```

6.1.3 Scenario 3

Not all students leave university at the same time. Some lecture finish earlier than others. Some students stay at university and keep studying on their own. Resulting in a low pedestrian density. All the workman that commuted to work in the morning leave work around this time and drive back home. Ending up with a high car-density.

```
# 17:00 [Chance, Amount] of Pedestrians/Cars spawned (per second)
amount_ped = {'crosswalk_L_L': [20, 1], 'crosswalk_L_R': [20, 1],
'crosswalk_M_L': [20, 1], 'crosswalk_M_R': [20, 1], 'crosswalk_U_U': [20,
1],
'crosswalk_U_B': [20, 1], 'crosswalk_B_U': [20, 1], 'crosswalk_B_B': [20,
1]}
amount_car = {'car_L': [60, 1], 'car_R': [60, 1]}
```

6.1.4 Scenario 4

In the evening there isn't much traffic around our intersection. There are neither much students around nor much cars passing our intersection. Resulting in low pedestrian density and low car density

```
# 20:00 [Chance, Amount] of Pedestrians/Cars spawned (per second)
amount_ped = {'crosswalk_L_L': [20, 1], 'crosswalk_L_R': [20, 1],
'crosswalk_M_L': [20, 1], 'crosswalk_M_R': [20, 1], 'crosswalk_U_U': [20,
1],
'crosswalk_U_B': [20, 1], 'crosswalk_B_U': [20, 1], 'crosswalk_B_B': [20,
1]}
amount_car = {'car_L': [30, 1], 'car_R': [30, 1]}
```

6.2 Traffic Light Periods

For each scenario we decided to analyse four different traffic light waiting periods (**T**raffic **L**ight scenario). The values were chosen based on observations around Zürich.

	Pedestrian waiting time [s]	**Car waiting time [s]**
TL scenario 1	0 (no traffic lights)	0 (no traffic lights)
TL scenario 2	15	25
TL scenario 3	20	20
TL scenario 4	25	15

6.3 Simulation Numeration

For a better overview we decided to number the simulation scenarios as follows

Simulation nr.	**TL scenario 1**	**TL scenario 2**	**TL scenario 3**	**TL scenario 4**
Scenario 1	1.1	1.2	1.3	1.4
Scenario 2	2.1	2.2	2.3	2.4
Scenario 3	3.1	3.2	3.3	3.4
Scenario 4	4.1	4.2	3.4	4.4

Hinweis: Der Leitfaden für unsere Arbeit ist die Arbeit «Pedestrian Dynamics in narrow, long hallways» Link: https://github.com/ratheile/MSSSM

~~Welche Parameter wurden bei der Simulation verwendet?~~

Verschiedene Simulationen auflisten und entsprechende Diagramme einfügen

Neue Diagramme mit neuen Parmeter benennen/beschreiben-

Ziel: Es soll für uns ersichtlich sein welche Situationen simuliert wurden und was die Parameter sind.

7. Simulation Results and Discussion

7.1 Situation1:

Our simulations lead us to different results depending on the different parameters. In Scenario 1.1 a deadlock situation occurred. That would of course not appear in the everyday life. We interpreted it as a failure of the system which matches the real life experience pretty well. In real life the worst congestions happen during the times when there are many pedestrians and many cars are around.

7.2 Situation2:

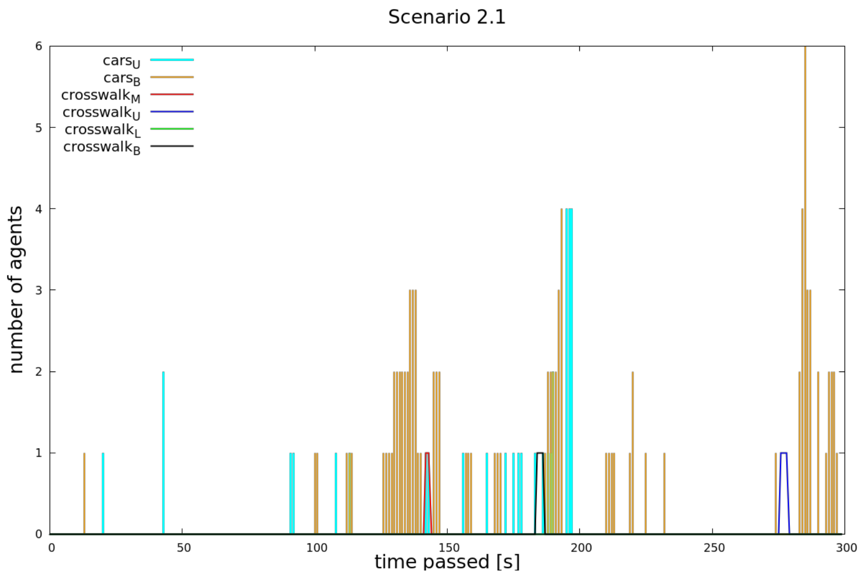

Figure 6: Simulation 2.1

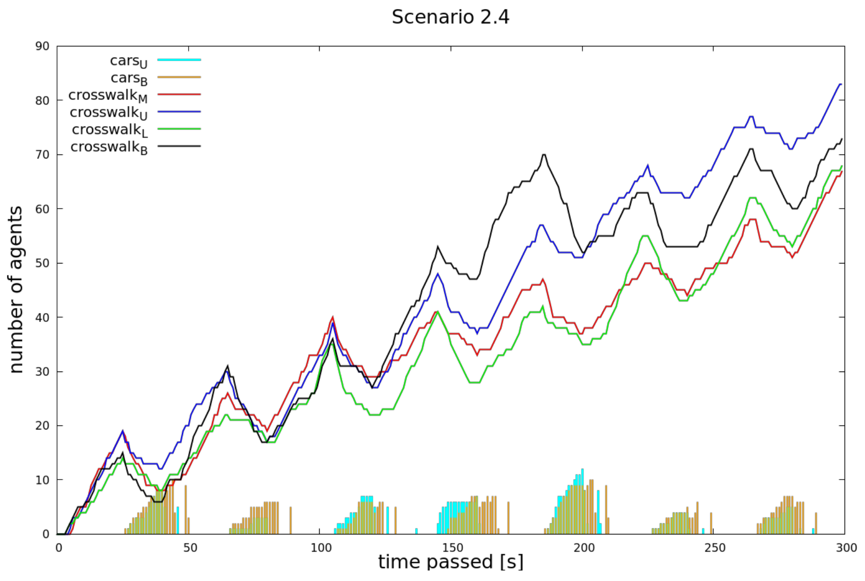

Figure 7: Scenario 2.4

As in Figure 6 can be seen, the traffic flows very well without any external regulation. Nevertheless the system even works worse with an exernal regulation as can be seen in Figure 7. A traffic light leads to a collaps for the pedestrians.

7.3 Sitiuation3 and 4:

Situation 3 and situation 4 lead us to almost identical results. Both situations work without traffic lights as well as they do with traffic lights. Both parties pedestrians and cars are affected in a negative way by traffic lights **but** the system still works with traffic lights.

7.4 Summary and Outlook

8. References

Python Software Foundation. (2018). *24.1. Tkinter — Python interface to Tcl/Tk — Python 2.7.15 documentation*. Retrieved from https://docs.python.org/2/library/tkinter.html

swisstopo. (2018, 11 07). *Karten der Schweiz - Schweizerische Eidgenossenschaft - map.geo.admin.ch*. Retrieved from https://s.geo.admin.ch/7e18fb5b95

A. Python source Code

```
Python code will be here
```