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 Landtwing & 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 Landtwing Pascal Lieberherr

Declaration of Originality

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

Implementation of a Traffic Light System at the Tannenbar-Intersection

Authored by (in block letters):
For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
Burger	Nico
Fent	Leo
Landtwing	Jérôme
Lieberherr	Pascal

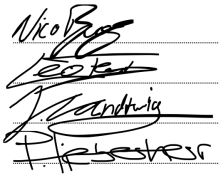
With my signature I confirm that
- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date

Zurich, 04.12.2018

Signature(s)



For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.

Table of Contents

1. Abstract

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

Title: Implementation of a Traffic Light System at the Tannenbar-Intersection

In this project we want to have a closer look at the intersection between the 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 want 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 reduce the waiting time for cars. At the same time the waiting time for pedestrians should not be too high. To do this, we simulated pedestrians, cars and trams with agents who behave according to the Swiss traffic rules. The whole simulation was done in Python.

To find out in which scenarios traffic lights make sense, we ran different simulations where we changed the car and pedestrian density from high to low. Furthermore, we figured out the best green/red time configuration of the traffic lights.

Finally, we arrived at the following conclusions: Traffic lights only show an improvement in traffic flow when car and pedestrian density are set to high. In this case the best green/red time configuration is 15 seconds waiting time for pedestrians and 25 seconds waiting time for cars. In all other cases, meaning when the density parameters were not set to *high* for both agents, traffic lights did not show any improvement compared to the simulation without traffic lights.

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, project report
Leo Fent:	map, visualization, coordination, project report
Jérôme Landtwing:	visualization, graphics and plots, project report
Pascal Lieberherr:	path of agents, spawning of agents, project report, flash talk

Listed above are only the main tasks every one 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

Figure 1: Birds eye view of the Tannenbar intersection.
(Source: Google Maps)

The intersection between the Tannenstrasse and the Universitätsstrasse is something all ETH students are well familiar with and so are we. It is located in a traffic hotspot with pedestrians, cars and trams. All of those use the road simultaneously. This leads to conflict, especially at peak times. Since 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 and noon are considered as peak time. What crossed our minds 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. During our observations of the crosswalk, we observed special dynamics e.g. cars that hardly stop for pedestrians waiting at the crosswalk as well as cars stopping way too early for pedestrians. Additionally, pedestrians who insist on their right of way in every situation could be observed. Based on these experiences we became curios about the effects of a traffic light system controlling the agents.

3.1 Fundamental Questions

We posed 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 or 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 questions, we ran different simulations. Each simulation represents a certain time during the day with appropriate car and pedestrian densities. Each simulation was performed with and without traffic lights. Furthermore, we also varied the green time for both agents. This way we could compare the current situation, without traffic lights, with the newly implemented traffic light system. We expected the following different results for each scenario:

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

Scenario 2: For low car and high pedestrian density we did not expect the result to be as clear as for high car and high pedestrian density. As we had a high car density it was still likely that cars started to line up. Hence, we thought that in this case traffic lights might be unnecessary, simply because they would not improve the smooth agents flow significantly but only increase the pedestrian waiting time.

Scenario 3: In this case we expected similar results as for scenario 2. Namely, the flow would not improve significantly while the pedestrian waiting time would increase. The only traffic light configuration that would make sense is one with more green time for the pedestrians.

Scenario 4: Scenario 4 was expected to be an obvious case again. We assumed that traffic lights would be unnecessary since they do not 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 did not care what happens before/after an agent enters our simulation.

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

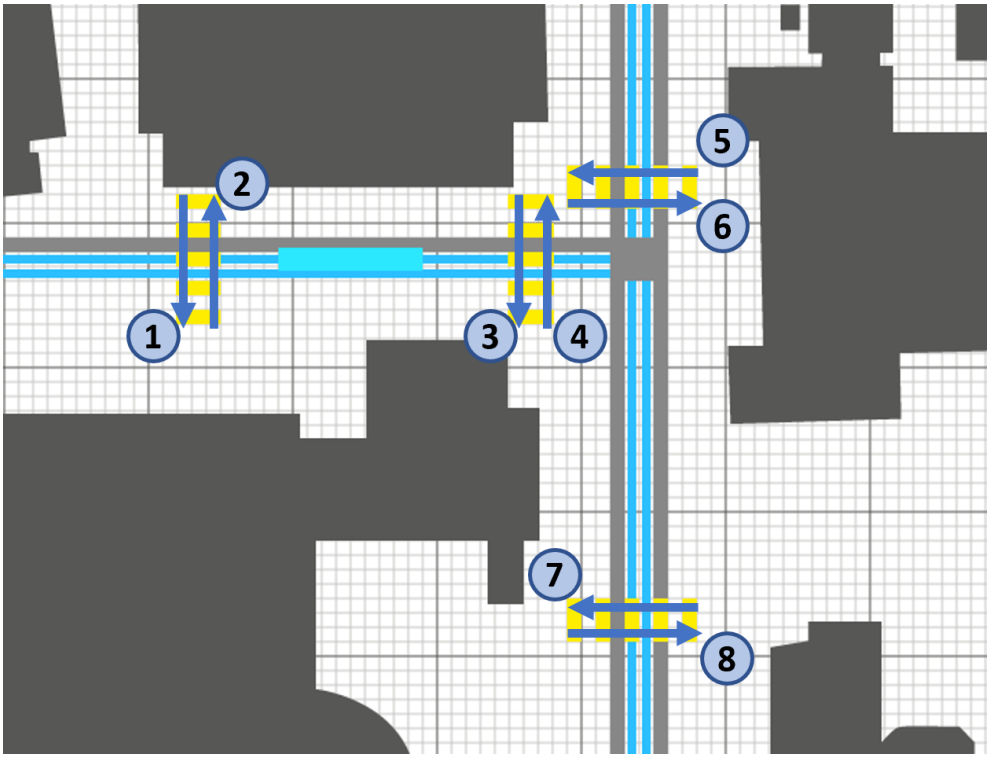


Figure 2: Pedestrian Paths

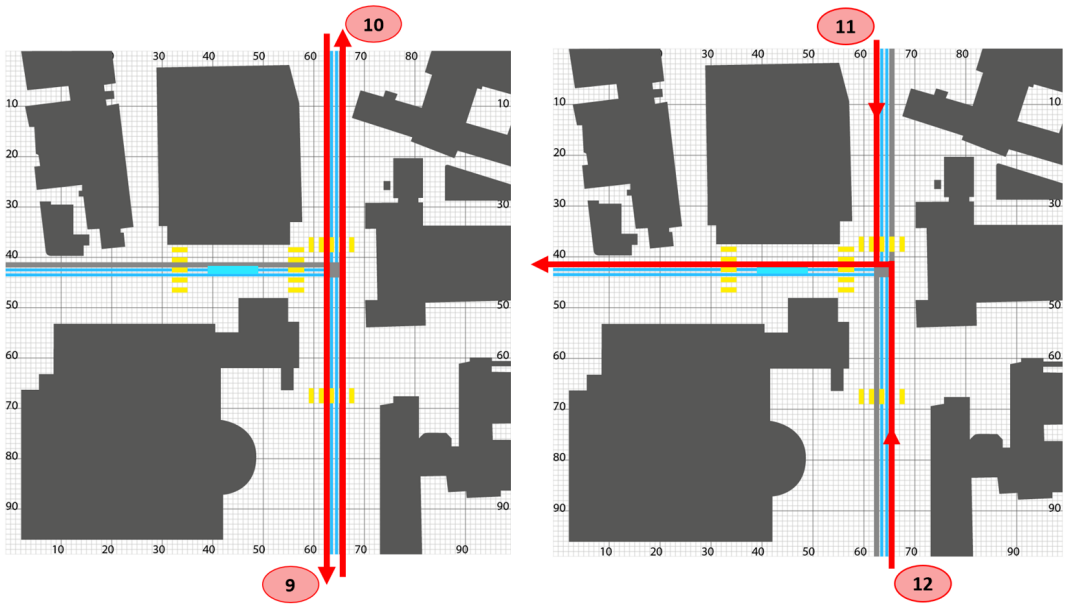


Figure 3: Car Paths

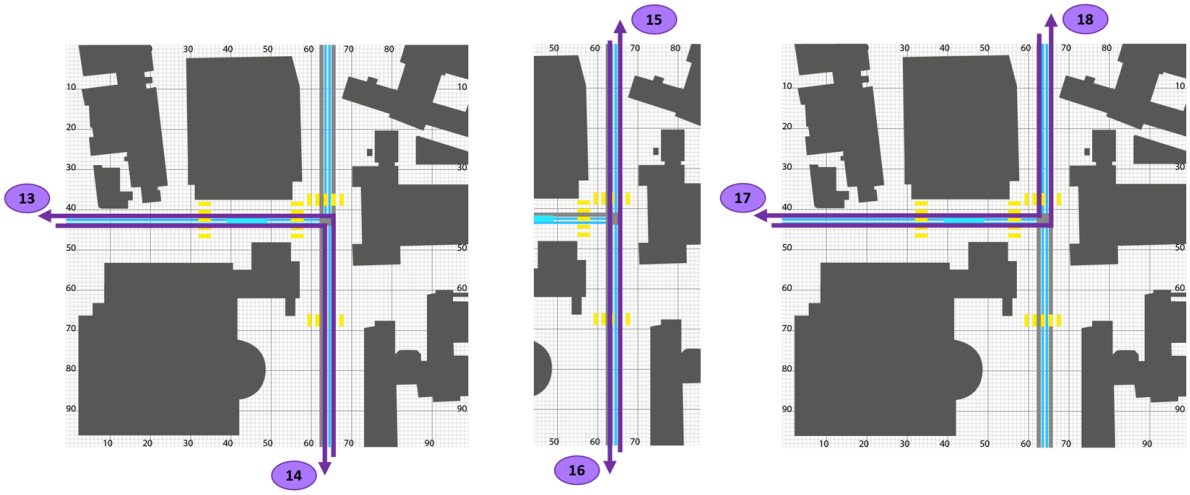


Figure 4: Tram Paths

In the Python code, the agent paths were labelled as follows:

Nr.	Crosswalks (Figure 2)	Nr.	Streets (Figure 3)	Nr.	Tram-Path (Figure 4)
1	Crosswalk_L_L	9	Car_L	13	6_Uni
2	Crosswalk_L_R	10	Car_R	14	6_Polybahn
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				

Table 2: Names of the paths in the code.

4.3 Agents

Our simulation would theoretically depend on countless agents of varying importance and attributes. To simplify the problem, we narrowed it down to three agents: pedestrians, cars and trams. Therefore, we intentionally neglected 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 assumed that all agents of the same type have unified attributes (e.g. speed, size). This in turn reduced the accuracy of our model. In order to reduce this inaccuracy, we focused on finding unified attributes that represent the general behaviour of each agent.

By observing the Tannenbar-Intersection, 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 single block and trams of 8 consecutive blocks.

Lastly, we had to ensure that our agents got spawned randomly. We implemented this feature by giving every agent a probability to spawn on a certain path.

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. Mirroring the real world, we came up with the following 4 rules:

1. Pedestrians only have to yield/give way to trams.
2. Trams must never give way.
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. However, to make our simulation applicable, we needed to assume that each agent follows the above 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.

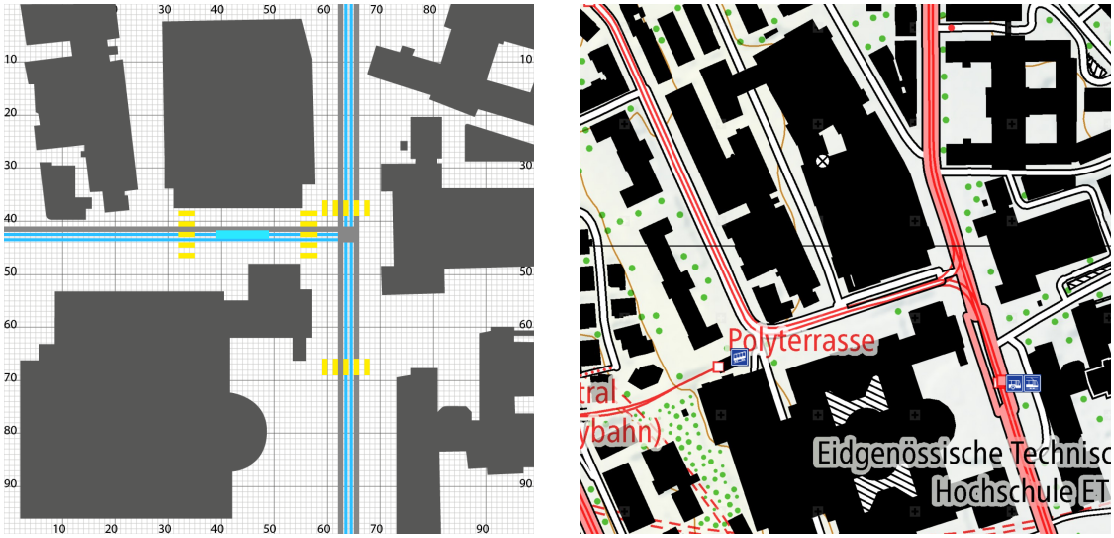


Figure 5: The simplified map used in the Simulations (left) and the original map by swisstopo[1] (right).

The agents were then placed onto the map onto 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 use 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 the matrix to contain information about the positions of the 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 path name, which determines which path it will take. In 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 its starting and end position are. 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 its first endpoint. After reaching the midpoint, the agent's endpoint gets updated to its final destination and its direction is changed. If an agent does not have a midpoint, the midpoint in the dictionary is a string "no" (for "no" midpoint) and it skips the previously stated process.

5.3 Agents

The agents are stored in lists (walkers, drivers, tram). When a new agent is spawned, its 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 with a list (walkers, drivers or tram) as input. This function then updates the positions of all agents in that list by iterating 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 whether that position is occupied already. If the position is occupied by another agent `(entry != 0)`, no changes are made. 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 graphical window object of the agent to the new position.

Another thing checked by the `move` function, 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 by 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 before, the trams are to be three times faster and the cars to be six times faster than the pedestrians. Therefore, the `iterate` function of the trams/cars is called 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. This is implemented 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. In order to control this, two parameters were implemented. 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 analyzed 4 different scenarios, the global variable `variante_zeiten` sets the spawning frequencies of each path by modifying the parameters mentioned above. These spawning frequencies are adjusted to the different scenarios, meaning that at 08:00 o'clock the probability for an agent to spawn is 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).

5.4 Interaction between Agents

In order to make our agents 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 lastly the cars, we achieve exactly the desired behavior,

since 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 (an agent can only move to an empty cell) 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 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. This is used for simulations without traffic lights and to reset the traffic lights before they change.
- `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 crossings are set to 5. This in turn means that cars have to wait and pedestrians can cross.

5.6 Graphical Output

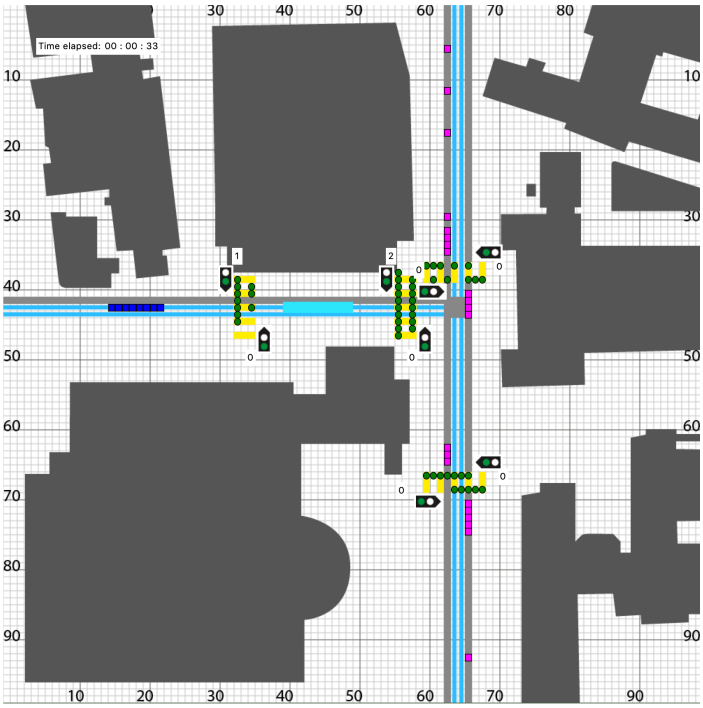


Figure 6: A running simulation.

5.6.1 Background and Agents

The graphical output was achieved using Python's *tkinter* module[2]. The above described map (Figure 5) was first drawn in Adobe Illustrator and then exported as GIF. This made it possible

[2] (Python Software Foundation, 2018)

to use it as a background for the tkinter window (later called canvas). On the canvas, the 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 contents of 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

6.1 Time Points

We wanted to analyze the impact of different traffic lights on our modeled system of the intersection. Since the amount of pedestrians and the density of cars passing the intersection varies depending on the time of day, we chose to simulate four different times to get an overview of how good a version of a traffic light is in general. We chose the following generic time points:

6.1.1 Scenario 1

We chose the time point 8 o'clock as an example with a high pedestrian density and a high car density. This matches the reality pretty well as most of the students attend lectures that start at 8:15 and a lot of people 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, most students leave their lectures and walk to a nearby cafeteria. This leads to high pedestrian density. Most of the people working stay at their work place 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 lectures finish earlier than others and some students stay at university to keep studying on their own. This results in a low pedestrian density. Meanwhile, the people who commuted to work in the morning leave work around this time and drive back home which results in a high car density.

The parameters of scenario 3 are as follows:

```
# 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 is neither a high pedestrian density nor a high car density. Most students have left by this time and commuters will be home already.

```
# 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]}
```

In summary, the different scenarios look like this:

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 3: A Summary of the different scenarios.

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

Table 4: Traffic light period times for different scenarios.

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	4.3	4.4

Table 5: Numeration of the simulation scenarios.

7. Simulation Results and Discussion

7.1 Scenario 1:

In the first scenario we simulated a high/high pairing, meaning both the pedestrian and car density were high. Depending on the traffic light period we got very different results. In scenario 1.1 (no traffic lights, Figure 7) we got a deadlock in the system. This meant that neither the pedestrians nor the cars were able to advance anymore. Obviously, this would not happen in the real world and we view this as a failure of our model. Nonetheless, scenario 1.1 mirrors our experience in the real world, where the cars cannot move anymore due to the number of people wanting to cross the street. That's where, in real life, the "rule breakers" come in. We did not simulate pedestrians who wait for cars to cross or cars that push their way through pedestrians. Either of them would have a significant effect on the simulation.

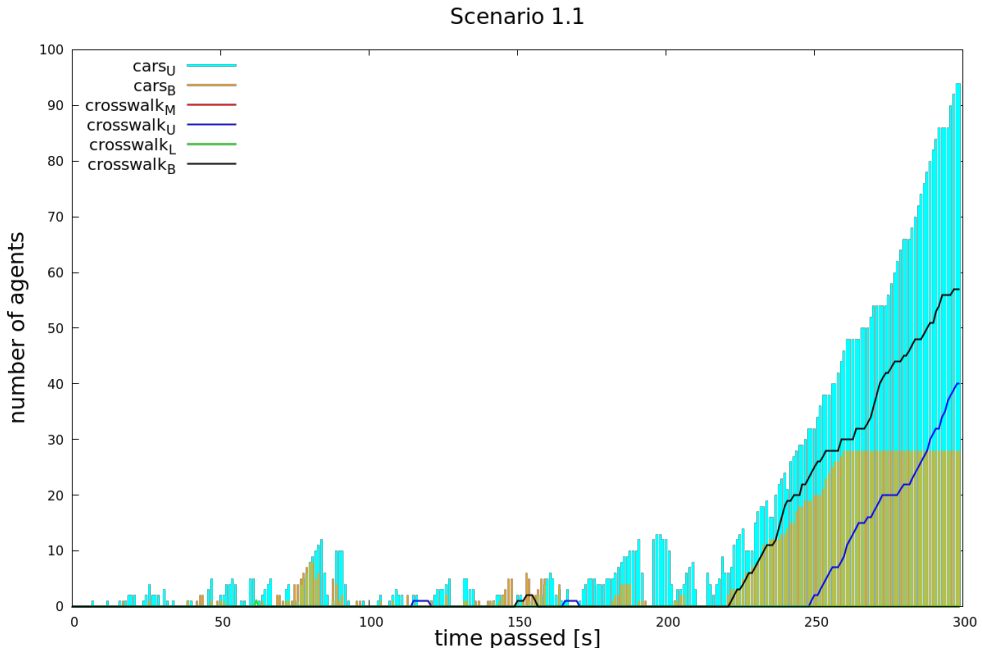


Figure 7: Scenario 1.1: High pedestrian and car density as well as no traffic lights.

We did see, however, a significant increase in the fluidity of traffic in scenario 1.2 (15s pedestrian waiting time, 25s car waiting time, Figure 8). As can be seen in the graph, the peaks of waiting pedestrians and cars reduce to zero after each period. The only exception is crosswalk$_U$ where we can see a build-up of waiting pedestrians after roughly 200 seconds. Note however, that the maximum number of people having to wait is five. In real life, that would pose no problem since people are able to walk in larger groups over the pedestrian crossing. Thus the last five people would be able to cross as well.

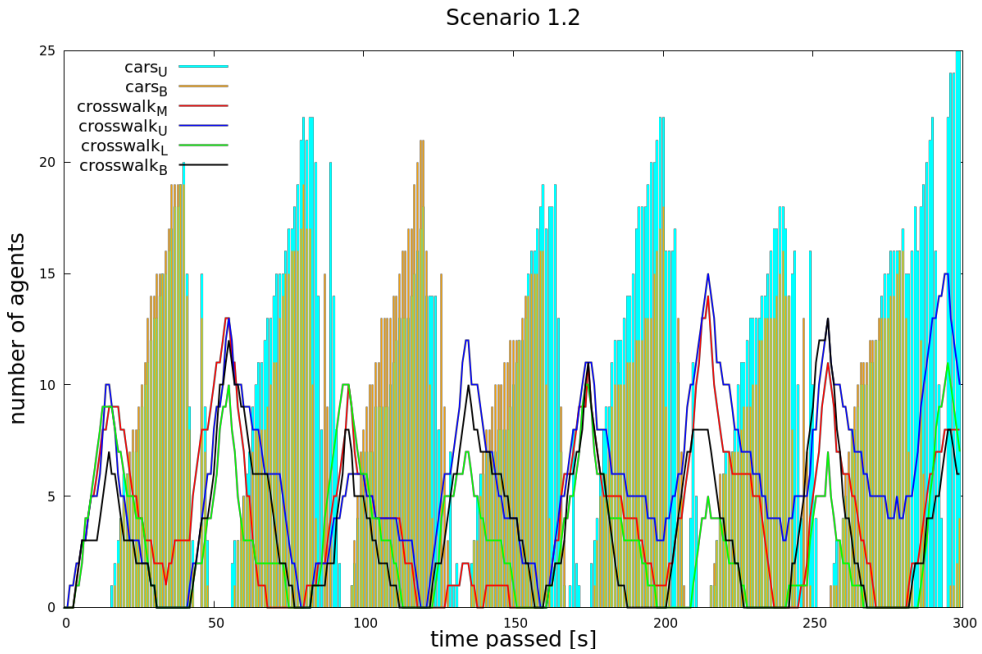


Figure 8: Scenario 1.2: High pedestrian and car density, pedestrian waiting time 15s.

In scenario 1.3 and 1.4 the build-up of pedestrians and cars waiting was considerably more significant (see Figure 15 and Figure 16).

Taking that into account, we feel that scenario 1.2 is a good solution for traffic jams occurring when there are lots of pedestrians as well as lots of cars. Other traffic lights periods did not prove to have a positive effect on traffic flow.

7.2 Scenario 2:

In scenario 2, high car density along with a low pedestrian density was simulated. In Figure 9 it can clearly be seen that traffic flows very well without any traffic lights. Even though the peaks of cars having to wait build up very quickly, they do not grow very tall. The rapid build-up (steep slope) of the peaks can be explained by the high number of cars arriving at a pedestrian crossing. The waiting times for cars remain short however, due to the number of pedestrians being rather small.

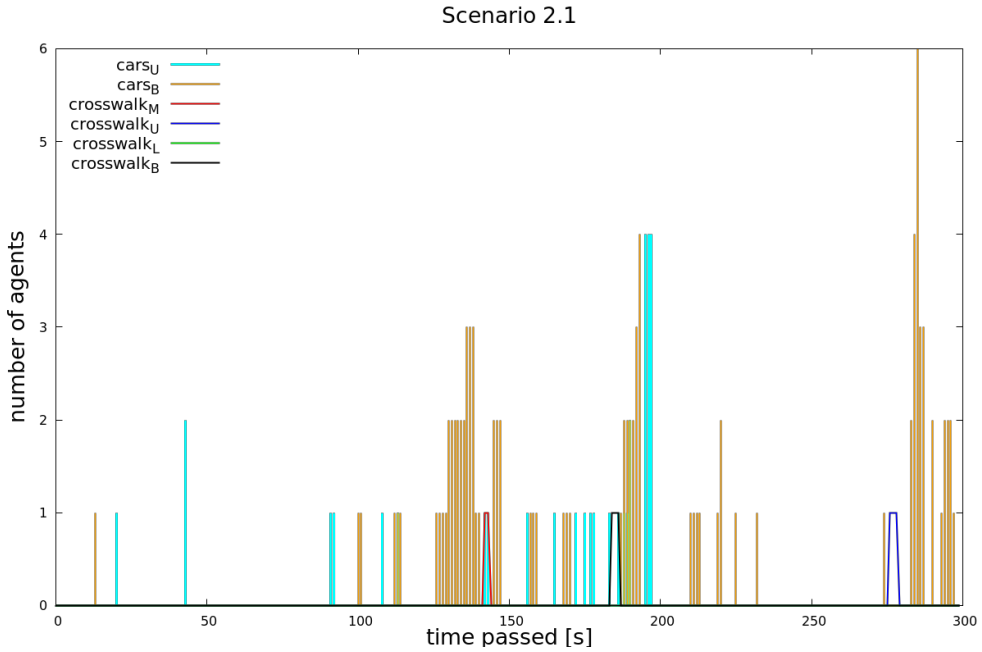


Figure 9: Simulation 2.1: High car and low pedestrian density, no traffic lights.

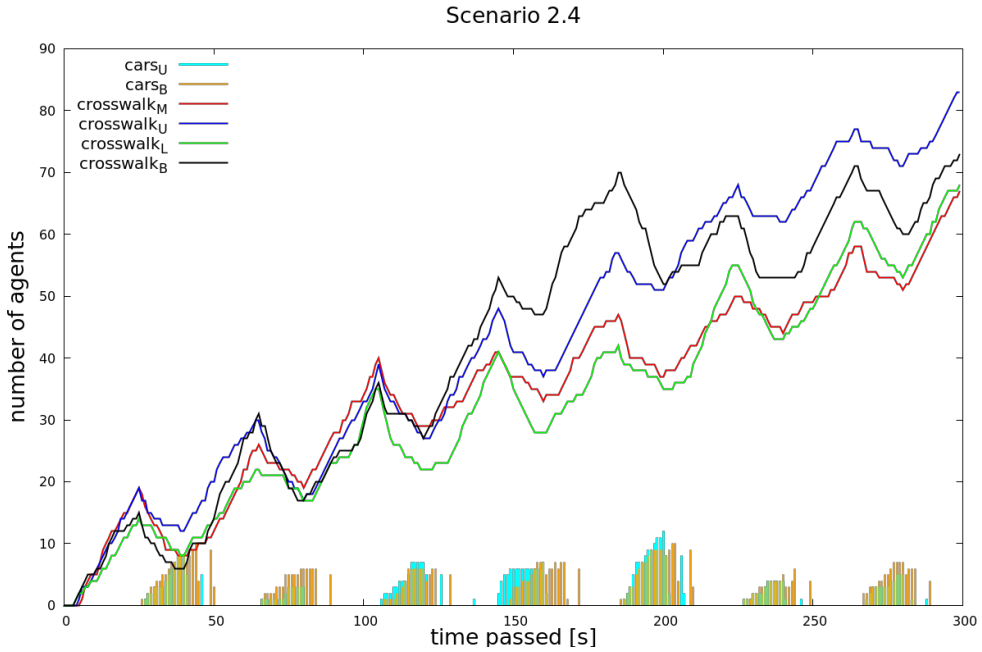


Figure 10: Scenario 2.4: High car and low pedestrian density, pedestrian waiting time 25s.

Figure 10 shows a wildly different picture. Even though there are relatively few pedestrians, them having to wait for 25 seconds leads to a massive and linearly increasing build-up of pedestrians at the crossings. Shorter waiting times (see scenarios 2.2 and 2.3 in the appendix) show a similar phenomenon although less dramatic.

This leads us to the conclusion that traffic lights do not have a positive effect on traffic flow when only few pedestrians are present. In the real world, we have made similar observations. When there are only few pedestrians, the cars have to wait only for short amounts of time which results in a steady traffic flow even when car density is high.

7.3 Scenario 3:

In scenario 3 the parameter car density was set to low and the pedestrian density to high. Looking at Figure 11 (no traffic lights) leads us to the conclusion that the flow of the agents is guaranteed even without traffic lights. Due to the fact that pedestrians never had to wait and only very few cars had to stop at the crossroad. The average of waiting cars is approximately 1.5. This exactly fulfils our requirements: As little as possible waiting time for pedestrians and only short waiting times for cars.

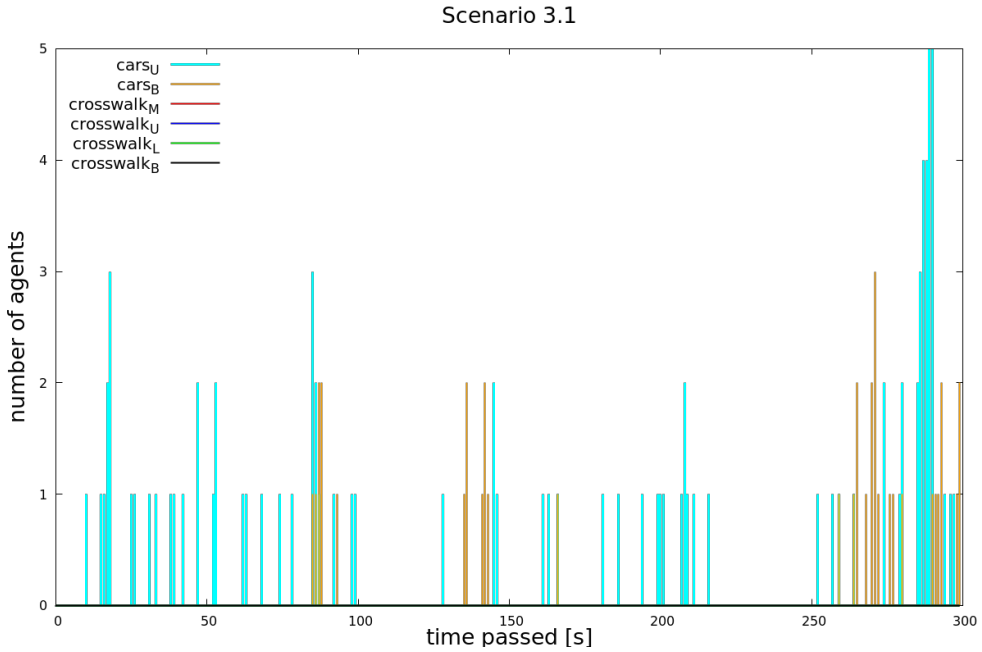


Figure 11: Scenario 3.1: Low car density, high pedestrian density, no traffic lights.

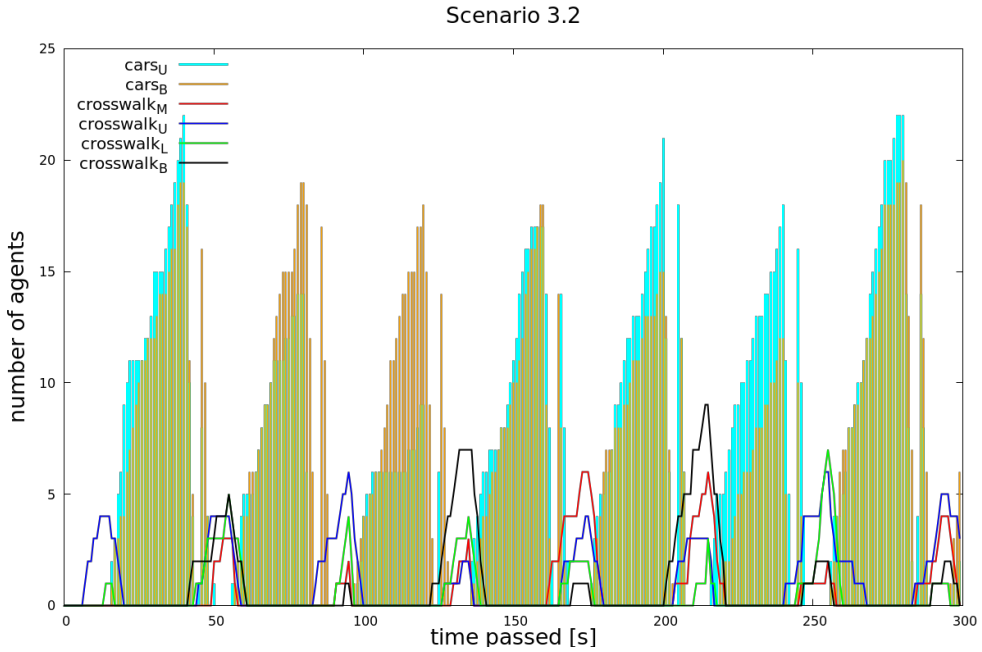


Figure 12: Scenario 3.2: Low car and high pedestrian density. Waiting time for pedestrians 15s, for cars 25s.

Figure 12 (waiting time pedestrians 15s, waiting time cars 25s) shows approximately a maximum of 17 cars waiting while the traffic light is on red. The number of waiting pedestrians is

around three to six. This is not an improvement compared to scenario 3.1 (no traffic lights). Without traffic lights, no pedestrians were waiting to cross the street. On the other hand, it is possible for all agents to either cross the street or the crosswalk when their traffic light is set to green. Therefore, the traffic light does not cause any traffic jam. Nevertheless, traffic lights do not make sense since they increase waiting time for both cars and pedestrians. The only reason why it could make sense in this case is when further aspects, such as safety are taken into account. However, this is not part of our thesis. The diagrams of scenarios 3.3 and 3.4 (to be found in the Appendix) depict a similar situation as we have in diagram 3.2.

7.4 Scenario 4:

Scenario 4 with low car and low pedestrian density shows a very similar behavior to the plots of scenario 3. Namely, the flow is ensured without traffic lights (see Figure 13). We cannot observe any waiting pedestrians and only once in while a car that has to stop at the crossroad. Thus, in this situation there is no need for traffic lights.

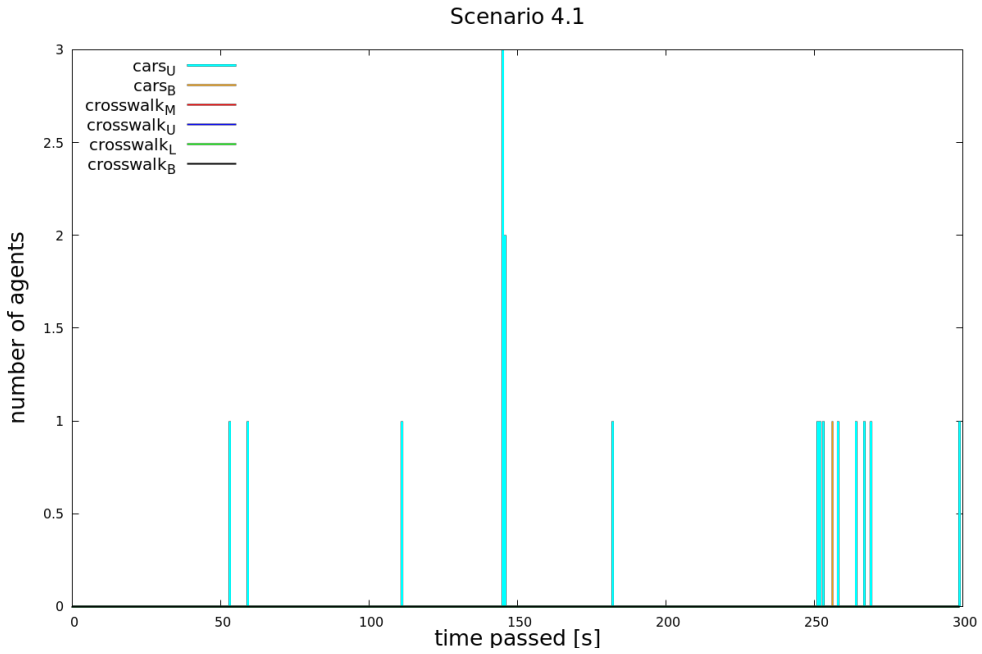


Figure 13: Scenario 4.1: Low car and pedestrian density, no traffic lights.

Having a look at scenario 4.3 where the green time between pedestrians and cars is equally split up leads us to a similar conclusion as for scenario 3.2. Implementing a traffic light is not an improvement compared to no lights at all. Additionally, there occurred no waiting pedestrians without traffic lights whereas pedestrians have to wait in scenario 4.3. As a result, implementing traffic lights worsens the situation for both cars and pedestrians. Scenario 4.2 and 4.4 are similar to scenario 4.3 and can be found in the appendix.

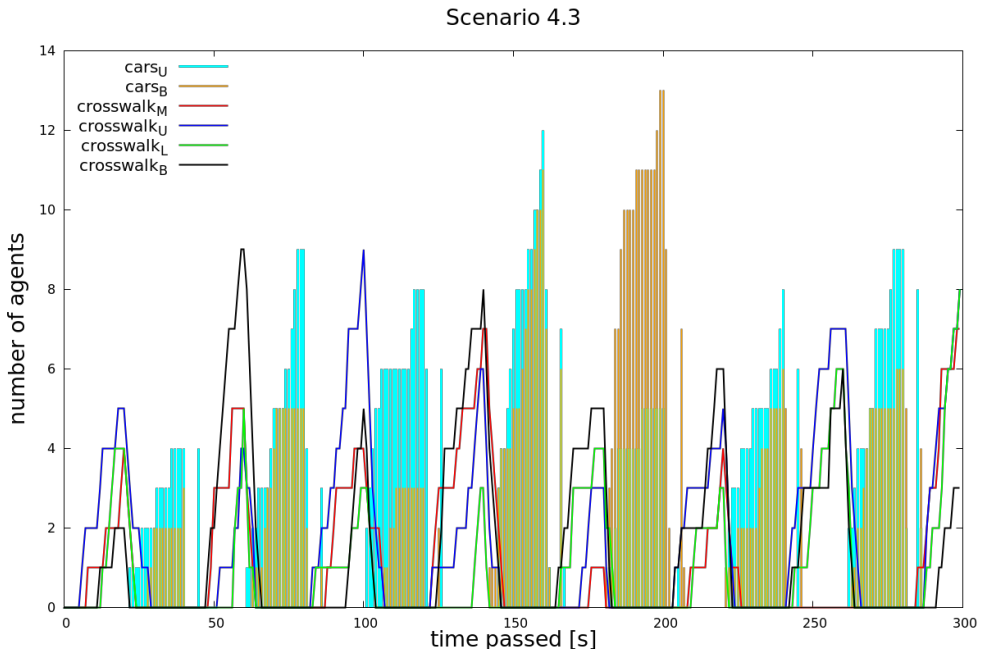


Figure 14: Scenario 4.3: Low car and pedestrian density. Waiting time 20s for pedestrians, 20s for cars.

7.5 Summary and Outlook

The aim of this project was to figure out whether the implementation of traffic lights at the Tannenbar-Intersection would lead to a significant reduction in waiting time for cars along with only a slight increase in pedestrian waiting time. To answer this question, we ran four different simulations:

In scenario 1 we simulated a high/high pairing, meaning both the pedestrian and car density were high. In scenario 2 pedestrian density was set to low and car density to high and in scenario 3 pedestrian density was set to high and car density to low. Scenario 4 was simulated with a low/low pairing. Each scenario was simulated with and without traffic lights. To find the best green/red time configuration for the traffic lights we also varied the green/red time for both agents (see Table 4).

Evaluating the graphs of the different scenarios lead us to the following conclusion: Only in scenario 1 the traffic lights show a significant flow improvement compared to the situation without traffic lights. However, in this case our model shows its limitations since deadlocks occur. This means no agent was able to advance anymore. For further research this would be something that needs to be improved. We also figured out that the best green/red time configuration of the traffic lights is 15 seconds waiting time for the pedestrians and 25 seconds for the cars. This way the peaks of waiting pedestrians and cars reduces to zero each period. The scenarios 2,3 and 4 did not show any improvement with traffic lights. Either the peak of the waiting pedestrians did not reduce to zero after each period and therefore the number of pedestrians waiting at a crosswalk increased over time or the number of waiting pedestrians and cars was simply smaller without traffic lights. Note that we only considered waiting time and no other aspects such as safety. Taking other aspect into account would need further research.

The following points listed below would make sense to improve when further research is done:

- Adapt the code such that no deadlocks can occur

- In general, it is worth to put effort in to a model that is closer to reality. For example, take the fact into account that cars slow down several meters before the crosswalk in case of a waiting pedestrian and also that they have a certain acceleration period after they stopped. Furthermore, pedestrians tend to wait at the crosswalk until the car has slowed down only then they cross the street.
- To decide whether traffic lights make sense it is also worth to take other aspects into account such as safety reasons.

In summary, it would only make sense to implement traffic lights when there are a lot of cars and many pedestrians. Furthermore, the best green/red time configuration would be 15 seconds waiting time for pedestrians and 25 seconds waiting time for cars. In all other cases traffic lights do not show an improvement in traffic flow.

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

9. Table of Figures

A. Additional Graphs

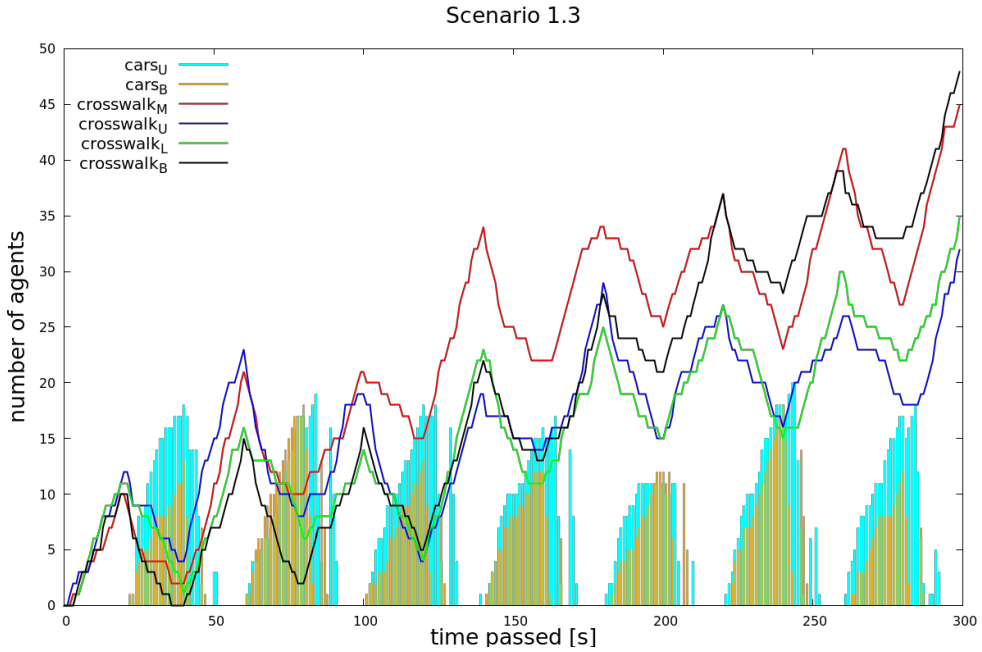


Figure 15: Scenario 1.3: High pedestrian and car density, pedestrian waiting time 20s, car waiting time 20s.

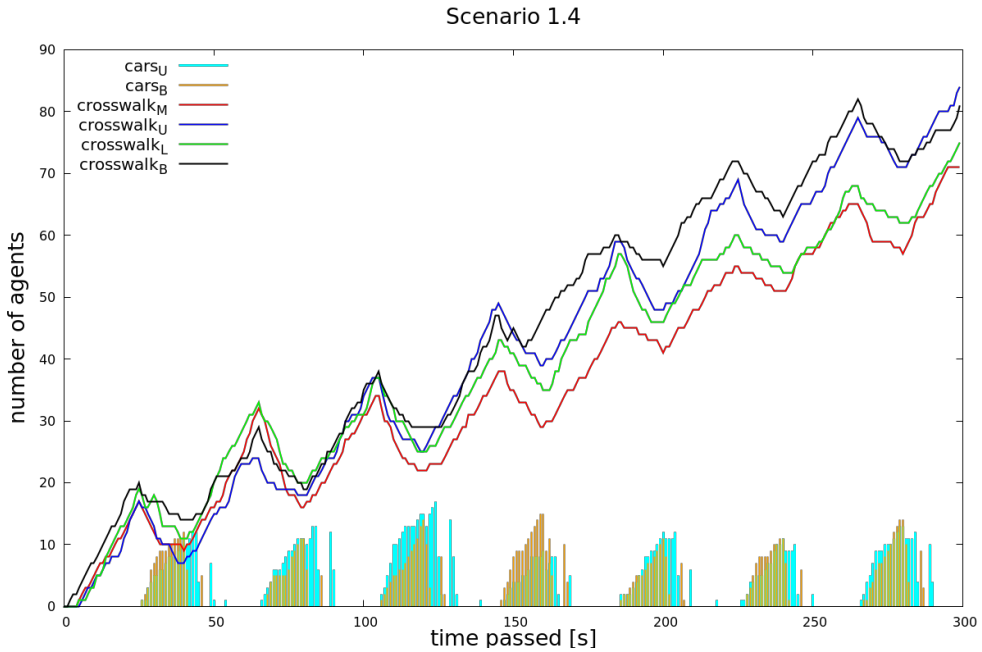


Figure 16: Scenario 1.4: High pedestrian and car density, pedestrian waiting time 25s, car waiting time 15s.

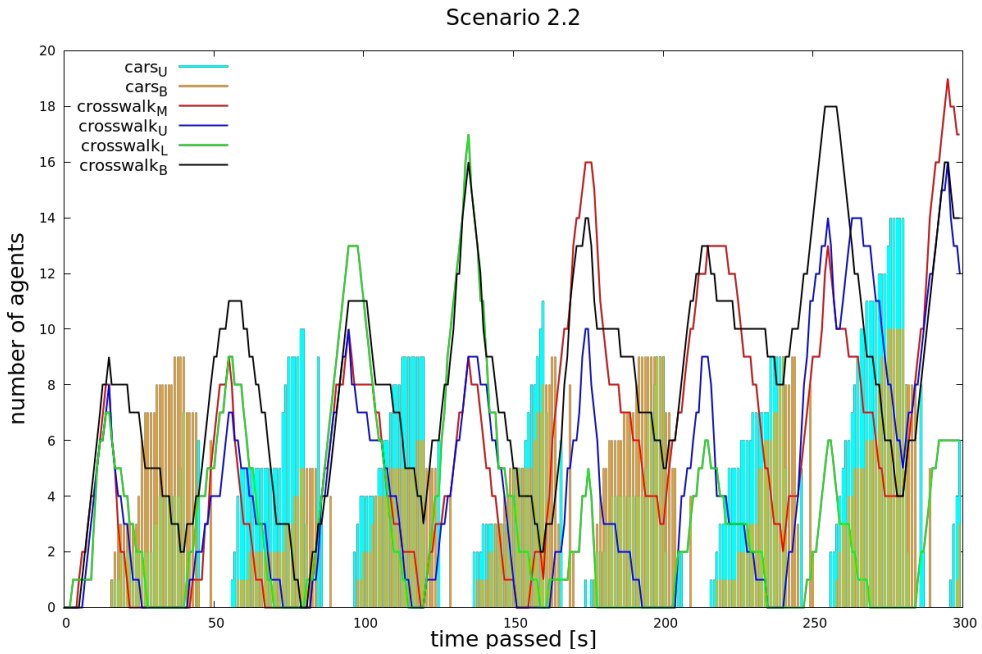


Figure 17: Scenario 2.2: Low pedestrian and high car density, pedestrian waiting time 15, car waiting time 25s.

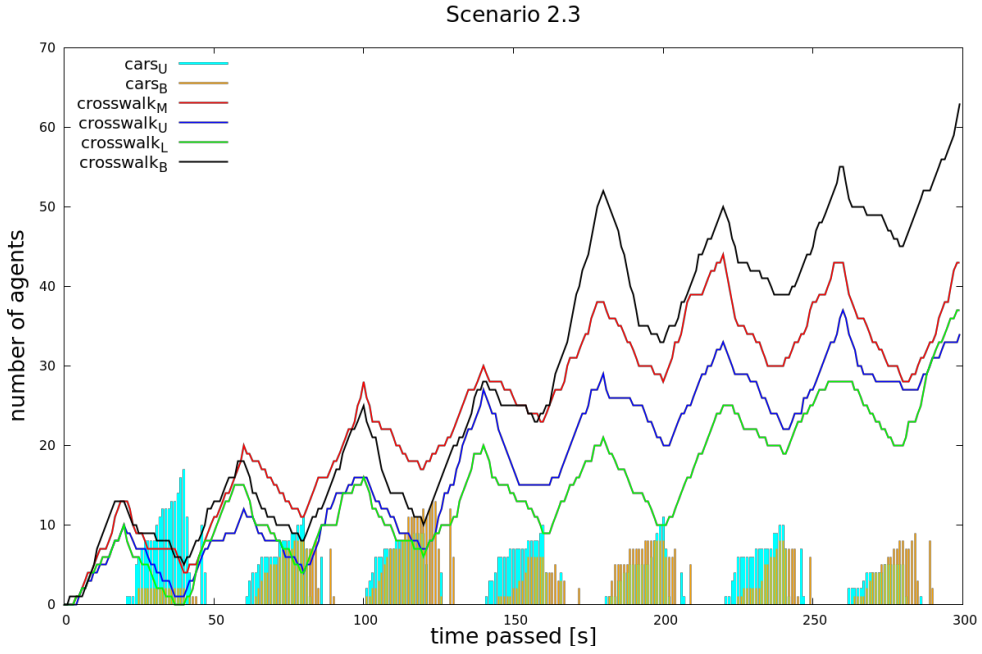


Figure 18: Scenario 2.3: Low pedestrian and high car density, pedestrian waiting time 20s, car waiting time 20s.

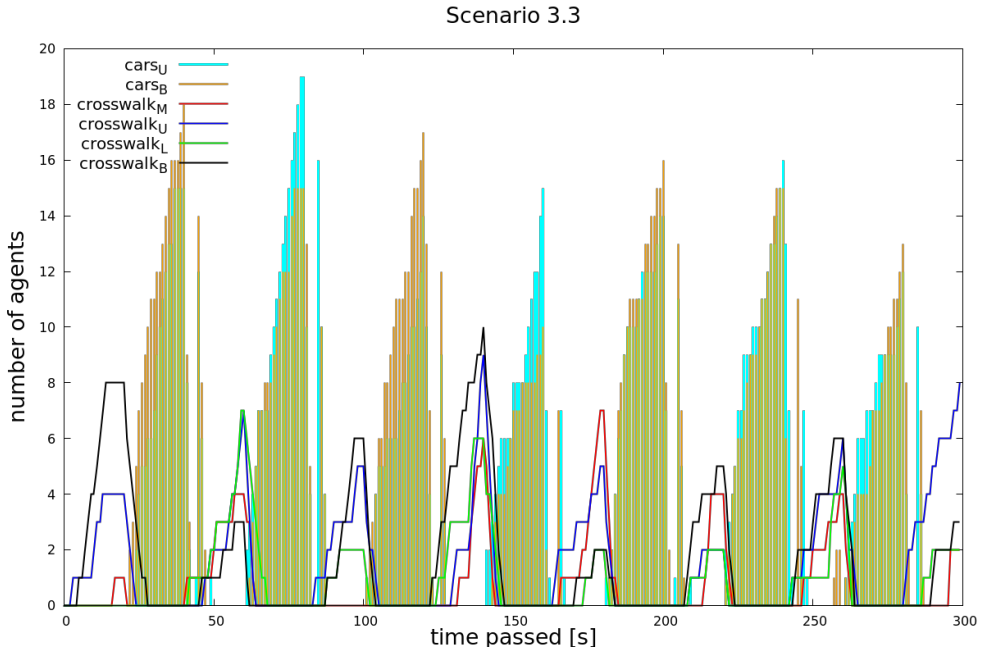


Figure 19: Scenario 3.3: High pedestrian and low car density, pedestrian waiting time 20s, car waiting time 20s.

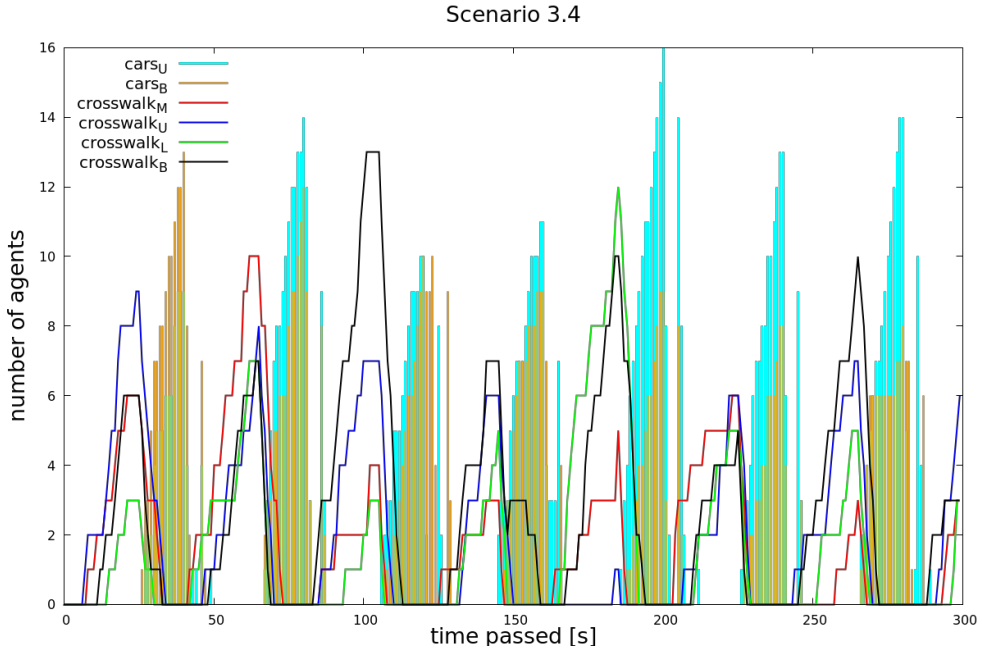


Figure 20: Scenario 3.4: High pedestrian and low car density, pedestrian waiting time 25s, car waiting time 15s.

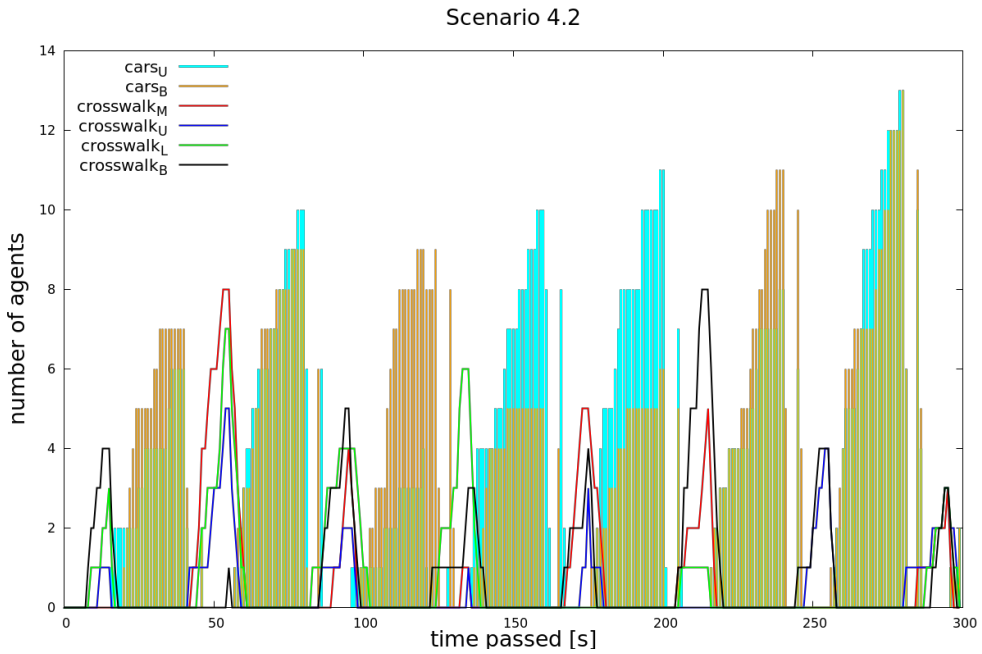


Figure 21: Scenario 4.2: Low pedestrian and car density, pedestrian waiting time 15s, car waiting time 25s.

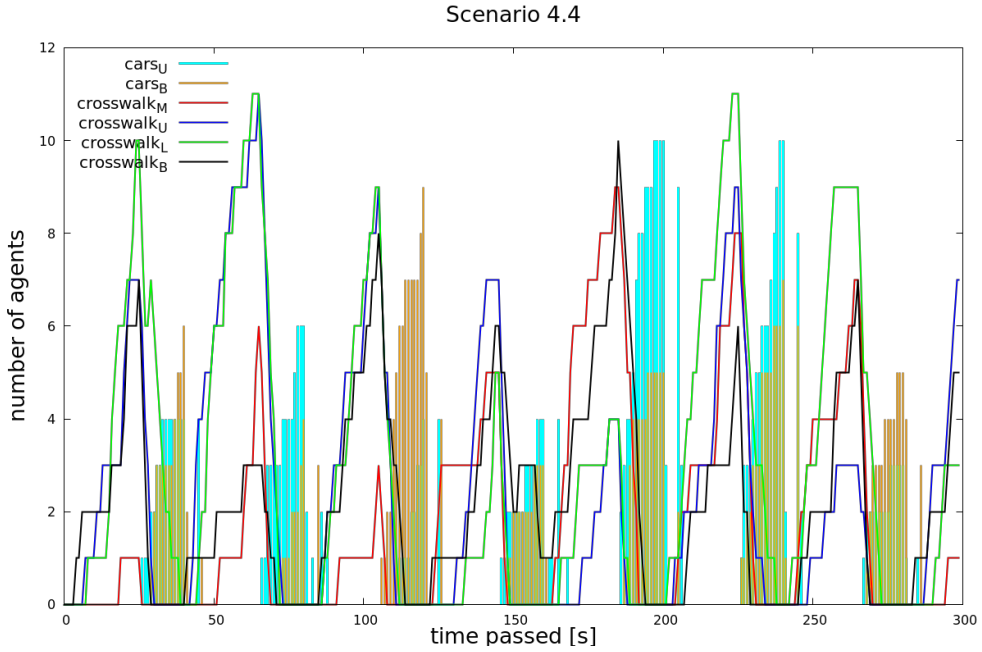


Figure 22: Scenario 4.4: Low pedestrian and car density, pedestrian waiting time 25s, car waiting time 15s.

B. Python source Code

```
# -*- coding: utf-8 -*-
from tkinter import *
from numpy import *
import random
import time
from PIL import ImageTk, Image
import csv

""" ------------------------- GLOBAL VARIABLES ------------------------- """

# For testing, change the following variables: ---------------

# Variance of time of the day:
# [1 = 08:00] SITUATION 1
# [2 = 12:00] SITUATION 2
# [3 = 17:00] SITUATION 3
# [4 = 20:00] SITUATION 4
variante_zeiten = 2


# Trafficlight: complete iteration time; green-light-time for cars [V15:
=15, V20: = 20, V25: 25]
variante_ampeln = True
light_phase = 40
car_phase = 25


# SPEED OF ANIMATION:
animation_speed = 0.05
# sets the simulation to run at full speed (recommended only and if only you
are not intressted in the datasheet)
maxspeed = False
# ----------------------------------------------------------------

# Height/Width of window:
LENGTH = 1000
# Squares horizontally/vertically:
units = 100
# Sidelength of squares:
size = LENGTH / units


# SPEED OF CARS is "car_speed" times as fast as a pedestrian:
car_speed = 6
# LENGTH OF TRAM
tram_length = 8


# TIME DISPLAY
```

```
t_s = 0
t_min = 0
t_h = 0

# PEDESTRIANS WAITING
waiting_ped = {'time_t': 0, 'crosswalk_L_L': 0, 'crosswalk_L_R': 0,
               'crosswalk_M_L': 0, 'crosswalk_M_R': 0, 'crosswalk_U_U': 0,
               'crosswalk_U_B': 0, 'crosswalk_B_U': 0, 'crosswalk_B_B': 0,
               'cars_L': 0, 'cars_R': 0}

waiting_ped2 = {'time': waiting_ped['time_t'],
                'crosswalk_L': waiting_ped['crosswalk_L_R'] +
waiting_ped['crosswalk_L_L'],
                'crosswalk_R': waiting_ped['crosswalk_M_L'] +
waiting_ped['crosswalk_M_R'],
                'crosswalk_U': waiting_ped['crosswalk_U_U'] +
waiting_ped['crosswalk_U_B'],
                'crosswalk_B': waiting_ped['crosswalk_B_B'] +
waiting_ped['crosswalk_B_U'],
                'cars_L': waiting_ped['cars_L'], 'cars_R':
waiting_ped['cars_R']}


""" ---------------------- FUNCTION DECLARATIONS ---------------------- """


def spawning_frequency(variante_zeiten):
    '''
    defines spawning frequency of cars and pedestrians
    depending on the day time [Chance, Amount]
    '''
    if variante_zeiten == 1:
        # 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]}

    if variante_zeiten == 2:
        # 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]}

    if variante_zeiten == 3:
        # 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]}

    if variante_zeiten == 4:
        # 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]}

    return amount_ped, amount_car


def schachbrett(canvas):
    '''
    creates vertical and horizontal lines in the canvas
    '''
    for x in range(LENGTH):
        canvas.create_line(size * x, 0, size * x, LENGTH, fill="#476042")
    for y in range(LENGTH):
        canvas.create_line(0, size * y, LENGTH, size * y, fill="#476042")


def initialize_gitter():
    '''
    Initializes coordinates matrix with zeros
    '''
    matrix = array(zeros((units, units)), dtype=int)
    return matrix


def print_matrix(matrix):
    '''
    prints matrix
    '''
    print("Matrixprinter at time: ", t_display)
    print(matrix)


def speed(agent):
    '''
    sets the speed of the agent according
    to their position
    '''
    xspeed = 0
    yspeed = 0
    if agent.cordx < agent.endx:
```

```
        xspeed = 1
    if agent.cordx > agent.endx:
        xspeed = -1
    if agent.cordy < agent.endy:
        yspeed = 1
    if agent.cordy > agent.endy:
        yspeed = -1
    return xspeed, yspeed


def spawn_ped(walkers, i):
    '''
    spawns pedestrians according to the defined amount and
    chance in the function spawning frequency
    '''
    for k in range(0, amount_ped["crosswalk_L_L"][1]):
        if random.randint(1, 101) <= amount_ped["crosswalk_L_L"][0]:
            walkers.append(Pedestrian("crosswalk_L_L"))

    for k in range(0, amount_ped["crosswalk_L_R"][1]):
        if random.randint(1, 101) <= amount_ped["crosswalk_L_R"][0]:
            walkers.append(Pedestrian("crosswalk_L_R"))

    for k in range(0, amount_ped["crosswalk_M_L"][1]):
        if random.randint(1, 101) <= amount_ped["crosswalk_M_L"][0]:
            walkers.append(Pedestrian("crosswalk_M_L"))

    for k in range(0, amount_ped["crosswalk_M_R"][1]):
        if random.randint(1, 101) <= amount_ped["crosswalk_M_R"][0]:
            walkers.append(Pedestrian("crosswalk_M_R"))

    for k in range(0, amount_ped["crosswalk_U_U"][1]):
        if random.randint(1, 101) <= amount_ped["crosswalk_U_U"][0]:
            walkers.append(Pedestrian("crosswalk_U_U"))

    for k in range(0, amount_ped["crosswalk_U_B"][1]):
        if random.randint(1, 101) <= amount_ped["crosswalk_U_B"][0]:
            walkers.append(Pedestrian("crosswalk_U_B"))

    for k in range(0, amount_ped["crosswalk_B_U"][1]):
        if random.randint(1, 101) <= amount_ped["crosswalk_B_U"][0]:
            walkers.append(Pedestrian("crosswalk_B_U"))

    for k in range(0, amount_ped["crosswalk_B_B"][1]):
        if random.randint(1, 101) <= amount_ped["crosswalk_B_B"][0]:
            walkers.append(Pedestrian("crosswalk_B_B"))


def spawn_cars(drivers, i):
    '''
    spawns pedestrians according to the defined amount and
    chance in the function spawning frequency
```

```
    '''
    for k in range(0, amount_car["car_L"][1]):
        if random.randint(1, 101) <= amount_car["car_L"][0]:
            drivers.append(Driver("car_L"))

    for k in range(0, amount_car["car_R"][1]):
        if random.randint(1, 101) <= amount_car["car_R"][0]:
            drivers.append(Driver("car_R"))


def spawn_tram_raster(tram, str, raster):
    '''
    Spawns the new tram in raster and on window
    '''
    agent = tram[-1]
    if str == 'vertical':
        for n in range(0, tram_length):
            # Spawn in Raster and on Window, vertically
            raster[agent.cordy_last + n * agent.yspeed][agent.cordx_last] =
3
            agent.tram_list.append(
                window.create_rectangle(agent.cordx * size + 3,
(agent.cordy_last + n * agent.yspeed) * size,
                                        (agent.cordx + 1) * size + 2,
(agent.cordy_last + n * agent.yspeed + 1) * size,
                                        fill='blue'))
    if str == 'horizontal':
        for n in range(0, tram_length):
            # Spawn in Raster and on Window, horizontally
            raster[agent.cordy_last][agent.cordx_last + n * agent.xspeed] =
3
            agent.tram_list.append(
                window.create_rectangle((agent.cordx_last + n *
agent.xspeed) * size + 3, agent.cordy * size,
                                        (agent.cordx_last + n * agent.xspeed
+ 1) * size + 2, (agent.cordy + 1) * size,
                                        fill='blue'))


def spawn_tram(tram, i, raster):
    '''
    spawns trams according to their schedule
    '''
    if i != 0:
        if i % 720 == 0:
            tram.append(Tram("6_Uni"))
            spawn_tram_raster(tram, "vertical", raster)

        if (i + 600) % 720 == 0:
            tram.append(Tram("6_Polybahn"))
            spawn_tram_raster(tram, "horizontal", raster)
```

```
        if (i + 480) % 720 == 0:
            tram.append(Tram("9_Uni"))
            spawn_tram_raster(tram, "vertical", raster)

        if (i + 360) % 720 == 0:
            tram.append(Tram("9_Haldenbach"))
            spawn_tram_raster(tram, "vertical", raster)

        if (i + 240) % 720 == 0:
            tram.append(Tram("10_Haldenbach"))
            spawn_tram_raster(tram, "vertical", raster)

        if (i + 120) % 720 == 0:
            tram.append(Tram("10_Polybahn"))
            spawn_tram_raster(tram, "horizontal", raster)
    else:
        tram.append(Tram("6_Uni"))
        spawn_tram_raster(tram, "vertical", raster)


def iterate(list, raster):
    '''
    A whole list gets updated => Each agent in that list gets to move
(order: from first to last)
    Returns if list is empty or not (True:= Empty)
    '''
    entities = len(list)
    iterator = 0

    # In case the list is empty
    if entities == 0:
        return True

    if isinstance(list[iterator], Tram):
        while iterator < entities:
            agent = list[iterator]
            finished = move_tram(agent, raster)
            if finished:
                # DELETES AGENT when he has arrived at destination
                # => Raster-entries gets set to 0, tram_list & List-entry
get deleted, List-Length (entities) gets shorter by -1
                if agent.xspeed_last == 0:
                    for i in range(0, tram_length):
                        raster[agent.cordy_last + i *
agent.yspeed_last][agent.cordx_last] = 0
                        window.delete(agent.tram_list[i])
                else:
                    for i in range(0, tram_length):
                        raster[agent.cordy_last][agent.cordx_last + i *
agent.xspeed_last] = 0
                        window.delete(agent.tram_list[i])
                del (list[iterator])
```

```
                entities -= 1
            else:
                iterator += 1
    else:
        while iterator < entities:
            agent = list[iterator]
            finished = move(agent, raster)
            if finished:
                # DELETES AGENT when he has arrived at destination
                # => Raster-entry gets set to 0, Shape & List-entry get
deleted, List-Length (entities) gets shorter by -1
                raster[agent.cordy][agent.cordx] = 0
                window.delete(agent.shape)
                del (list[iterator])
                entities -= 1
            else:
                iterator += 1
    return False


class Pedestrian:
    def __init__(self, path):
        # Variable declaration for Crosswalks
        #    crosswalk_L_L  very left crosswalk (see map) and left lane
        #    crosswalk_L_R  very left crosswalk / right lane
        #    crosswalk_M_L  middle crosswalk / left lane
        #    crosswalk_M_R  middle crosswalk / right lane
        #    crosswalk_U_U  upper crosswalk / upper lane
        #    crosswalk_U_B  upper crosswalk / bottom lane
        #    crosswalk_B_U  bottom crosswalk / upper lane
        #    crosswalk_B_B  bottom crosswalk / bottom lane

        # Dictionary that stores x-coordinatess of the starting position for
the pedestrians
        self.startposx = {
            'crosswalk_L_L': 33,
            'crosswalk_L_R': 35,
            'crosswalk_M_L': 56,
            'crosswalk_M_R': 58,
            'crosswalk_U_U': 69,
            'crosswalk_U_B': 59,
            'crosswalk_B_U': 69,
            'crosswalk_B_B': 59
        }

        self.class_ = "Pedestrian"

        # Stores Path of Pedestrian
        self.path = path

        # Dictionary that stores y-coordinatess of the starting position for
the pedestrians
```

```
        self.startposy = {
            'crosswalk_L_L': 38,
            'crosswalk_L_R': 48,
            'crosswalk_M_L': 38,
            'crosswalk_M_R': 48,
            'crosswalk_U_U': 37,
            'crosswalk_U_B': 39,
            'crosswalk_B_U': 67,
            'crosswalk_B_B': 69
        }

        # Dictionary that stores x-coordinatess of the end position for the
pedestrians
        self.endposx = {
            'crosswalk_L_L': 33,
            'crosswalk_L_R': 35,
            'crosswalk_M_L': 56,
            'crosswalk_M_R': 58,
            'crosswalk_U_U': 59,
            'crosswalk_U_B': 69,
            'crosswalk_B_U': 59,
            'crosswalk_B_B': 69
        }

        # Dictionary that stores y-coordinatess of the end position for the
pedestrians
        self.endposy = {
            'crosswalk_L_L': 48,
            'crosswalk_L_R': 38,
            'crosswalk_M_L': 48,
            'crosswalk_M_R': 38,
            'crosswalk_U_U': 37,
            'crosswalk_U_B': 39,
            'crosswalk_B_U': 67,
            'crosswalk_B_B': 69
        }

        # Matching of start- and endposition.
        self.startx = self.startposx[path]
        self.starty = self.startposy[path]
        self.endx = self.endposx[path]
        self.endy = self.endposy[path]

        # Position of Agent
        self.cordx = self.startx
        self.cordy = self.starty

        # Determine Direction/Speed of agent
        self.xspeed, self.yspeed = speed(self)

        self.shape = window.create_oval(self.cordx * size + 3, self.cordy *
size, self.cordx * size + size + 2,
```

```
                                        self.cordy * size + size,
fill='green')


class Driver:
    def __init__(self, path):
        # Variable declaration:
        #    car_L car      driving on the left lane -> upwards
        #    car_R car      driving on the right lane -> downwards
        #    car_left_turn   driving on the left lane until crossroad then
left turn

        self.class_ = 'Driver'

        # Dictionary containing x-startcoorindates for instances of DRIVER
        self.startposx = {
            'car_L': 63,
            'car_R': 66,
        }

        # Dictionary containing y-startcoordinates for instances of DRIVER
        self.startposy = {
            'car_L': 0,
            'car_R': 99,
        }

        self.middleposx = {
            'car_L': 63,
            'car_R': 66,
        }

        self.middleposy = {
            'car_L': 42,
            'car_R': 42,
        }

        # Dictionary containing x-endcoordinates for instances of DRIVER
        self.endposx = {
            'car_L': 63,
            'car_R': 66,
        }

        # Dictionary containing y-endcoordinates for instances of DRIVER
        self.endposy = {
            'car_L': 99,
            'car_R': 0,
        }

        self.path = path
        self.startx = self.startposx[path]
        self.starty = self.startposy[path]
        if random.randint(0, 30) == 5:
```

```
            self.endx = self.middleposx[path]
            self.endy = self.middleposy[path]
        else:
            self.endx = self.endposx[path]
            self.endy = self.endposy[path]

        # Position of Agent
        self.cordx = self.startx
        self.cordy = self.starty

        # Determine Direction/Speed of agent
        self.xspeed, self.yspeed = speed(self)

        self.shape = window.create_rectangle(self.cordx * size + 3,
self.cordy * size, self.cordx * size + size + 2,
                                             self.cordy * size + size,
fill='magenta')


class Tram:
    def __init__(self, number):

        # Variable declaration:
        # The following trams stop at Eth/Universitätsspital:
        # 6, 9, 10
        # They either drive in the direction of 1. Polybahn, 2. Haldenbach
or 3. Uni (Universität Zürich)

        # Dictionary containing x-startcoorindates for Trams
        self.startposx = {
            '6_Uni': 65,
            '6_Polybahn': tram_length - 1,
            '9_Uni': 65,
            '9_Haldenbach': 64,
            '10_Haldenbach': 64,
            '10_Polybahn': tram_length - 1,
        }

        self.class_ = "Tram"

        # Dictionary containing y-startcoorindates for Trams
        self.startposy = {
            '6_Uni': 99 - tram_length + 1,
            '6_Polybahn': 44,
            '9_Uni': 99 - tram_length + 1,
            '9_Haldenbach': tram_length - 1,
            '10_Haldenbach': tram_length - 1,
            '10_Polybahn': 44,
        }

        # Dictionary containing x-endcoordinates for Trams
        self.startposx_end = {
```

```
            '6_Uni': 65,
            '6_Polybahn': 0,
            '9_Uni': 65,
            '9_Haldenbach': 64,
            '10_Haldenbach': 64,
            '10_Polybahn': 0,
        }

        # Dictionary containing y-endcoordinates for Trams
        self.startposy_end = {
            '6_Uni': 99,
            '6_Polybahn': 44,
            '9_Uni': 99,
            '9_Haldenbach': 0,
            '10_Haldenbach': 0,
            '10_Polybahn': 44,
        }

        # Dictionary containing middle-x-coordinates for Trams
        self.middleposx = {
            '6_Uni': 65,
            '6_Polybahn': 64,
            '9_Uni': 'no',
            '9_Haldenbach': 'no',
            '10_Haldenbach': 64,
            '10_Polybahn': 65,
        }

        # Dictionary containing middle-y-coordinates for Trams
        self.middleposy = {
            '6_Uni': 43,
            '6_Polybahn': 44,
            '9_Uni': 'no',
            '9_Haldenbach': 'no',
            '10_Haldenbach': 43,
            '10_Polybahn': 44,
        }

        # Dictionary containing x-endcoordinates for Trams
        self.endposx = {
            '6_Uni': 0,
            '6_Polybahn': 64,
            '9_Uni': 65,
            '9_Haldenbach': 64,
            '10_Haldenbach': 0,
            '10_Polybahn': 65,
        }

        # Dictionary containing y-endcoordinates for Trams
        self.endposy = {
            '6_Uni': 43,
            '6_Polybahn': 99,
```

```
            '9_Uni': 0,
            '9_Haldenbach': 99,
            '10_Haldenbach': 43,
            '10_Polybahn': 0,
        }

        # Saves agents number
        self.number = number

        # Matching of start and end positions for Trams
        self.startx = self.startposx[number]
        self.starty = self.startposy[number]
        self.startx_end = self.startposx_end[number]
        self.starty_end = self.startposy_end[number]
        # If change in direction => middleposition is an integer
        if self.middleposx[number] == "no":
            self.endx = self.endposx[number]
            self.endy = self.endposy[number]
        else:
            self.endx = self.middleposx[number]
            self.endy = self.middleposy[number]

        # Position of Agent
        self.cordx = self.startx
        self.cordy = self.starty
        self.cordx_last = self.startx_end
        self.cordy_last = self.starty_end

        # Determine Direction/Speed of agent
        self.xspeed, self.yspeed = speed(self)
        self.xspeed_last = self.xspeed
        self.yspeed_last = self.yspeed

        # Create a list of Tram Waggons; first waggon of tram at tram_list[-
1] and last waggon at tram_list[0]
        self.tram_list = []


def rotlicht(number, matrix):
    '''
    function locks the proper entries in the matrix. Hence, the agents
cannot go
    on that field
    '''
    if number == 0:
        matrix[41][33] = 0
        matrix[45][35] = 0
        matrix[41][56] = 0
        matrix[45][58] = 0
        matrix[39][62] = 0
        matrix[37][67] = 0
        matrix[69][62] = 0
```

```
        matrix[67][67] = 0
        matrix[42][36] = 0
        matrix[42][59] = 0
        matrix[36][63] = 0
        matrix[40][66] = 0
        matrix[70][66] = 0
        matrix[66][63] = 0
        return matrix
    if number == 1:
        # Pedestrians are not allowed to pass crosswalks.
        # Matrix[x][y] = 4 for all coordinates right in front of a
crosswalk. => Artificial obstacle that blocks their way
        matrix[41][33] = 4
        matrix[45][35] = 4
        # crosswalk2
        matrix[41][56] = 4
        matrix[45][58] = 4
        # crosswalk3
        matrix[39][62] = 4
        matrix[37][67] = 4
        # crosswalk4
        matrix[69][62] = 4
        matrix[67][67] = 4
        return matrix
    if number == 2:
        # Cars must wait in front of crosswalks
        # crosswalk1:
        matrix[42][36] = 5
        # crosswalk
        matrix[42][59] = 5
        # crosswalk3
        matrix[36][63] = 5
        matrix[40][66] = 5
        # crosswalk
        matrix[70][66] = 5
        matrix[66][63] = 5
        return matrix


def move(agent, matrix):
    '''
    Saves the current coordinates (old) and the future ones (new)
    '''
    xn_old = agent.cordx
    xn_new = agent.cordx + agent.xspeed
    ym_old = agent.cordy
    ym_new = agent.cordy + agent.yspeed


    if matrix[ym_new][xn_new] == 0:
        #valid move.
        if matrix[ym_old][xn_old] <= 3:
```

```
            matrix[ym_old][xn_old] = 0
        if isinstance(agent, Driver):
            matrix[ym_new][xn_new] = 2
        if isinstance(agent, Pedestrian):
            matrix[ym_new][xn_new] = 1
        agent.cordx = xn_new
        agent.cordy = ym_new
        window.move(agent.shape, agent.xspeed * size, agent.yspeed * size)

    if agent.class_ == 'Pedestrian' and agent.starty == agent.cordy and
agent.startx == agent.cordx:
        waiting_ped[agent.path] += 1

    # Returns True if Car is at Endposition or Changes speed if it is at the
middleposition
    if isinstance(agent, Driver):
        if agent.cordx == agent.endx and agent.cordy == agent.endy:
            if agent.cordx == agent.middleposx[agent.path] and agent.cordy
== agent.middleposy[agent.path]:
                agent.endx = 0
                agent.endy = 42
                agent.xspeed, agent.yspeed = speed(agent)
            else:
                return True

    if agent.cordx == agent.endx and agent.cordy == agent.endy:
        return True
    else:
        return False


def move_tram(agent, matrix):
    '''
    moves tram
    '''
    number = agent.number

    # Returns True if Agent is at endposition & Changes speed if FIRST
WAGGON is at the middleposition
    if agent.cordx == agent.endx and agent.cordy == agent.endy:
        if agent.cordx == agent.middleposx[number] and agent.cordy ==
agent.middleposy[number]:
            agent.endx = agent.endposx[number]
            agent.endy = agent.endposy[number]
            agent.xspeed, agent.yspeed = speed(agent)
        else:
            return True

    xn_last = agent.cordx_last
    xn_new = agent.cordx + agent.xspeed
    ym_last = agent.cordy_last
    ym_new = agent.cordy + agent.yspeed
```

```
    # Changes speed if LAST WAGGON is at the middleposition
    if agent.cordx_last == agent.middleposx[number] and agent.cordy_last ==
agent.middleposy[number]:
        agent.endx_last = agent.endposx[number]
        agent.endy_last = agent.endposy[number]
        xspeed = 0
        yspeed = 0
        if agent.cordx_last < agent.endx_last:
            xspeed = 1
        if agent.cordx_last > agent.endx_last:
            xspeed = -1
        if agent.cordy_last < agent.endy_last:
            yspeed = 1
        if agent.cordy_last > agent.endy_last:
            yspeed = -1
        agent.xspeed_last = xspeed
        agent.yspeed_last = yspeed

    if matrix[ym_new][xn_new] == 0:
        #valid move
        matrix[ym_last][xn_last] = 0
        matrix[ym_new][xn_new] = 3
        agent.cordx = xn_new
        agent.cordy = ym_new
        agent.cordx_last = xn_last + agent.xspeed_last
        agent.cordy_last = ym_last + agent.yspeed_last
        window.delete(agent.tram_list[0])
        del (agent.tram_list[0])
        agent.tram_list.append(window.create_rectangle(agent.cordx * size +
3, (agent.cordy) * size,
                                                       agent.cordx * size +
size + 2,
                                                       (agent.cordy + 1) *
size, fill='blue'))
    return False


def print_time(s_i, min_i, h_i):
    """
    Updates the current time displayed in the window
        Input: s, min, h as integers
    """
    time_label = Label(window, text="Time elapsed: ")
    time_label.place(x=50, y=50)
    if s_i < 10:
        s = '0' + str(s_i)
    else:
        s = str(s_i)
    if min_i < 10:
        minute = '0' + str(min_i)
    else:
```

```
        minute = str(min_i)
    if h_i < 10:
        h = '0' + str(h_i)
    else:
        h = str(h_i)
    current_time = Label(window, text=(h, ':', minute, ':', s))
    current_time.place(x=145, y=50)


def draw_lights(number):
    """
    Draws traffic lights in Canvas:
     1 = pedestrians WAIT
     2 = pedestrians GO
    """
    if number == 1:
        light_ped_b = PhotoImage(file='lights/red_bottom.gif')
        light_ped_t = PhotoImage(file='lights/red_top.gif')
        light_ped_r = PhotoImage(file='lights/red_right.gif')
        light_ped_l = PhotoImage(file='lights/red_left.gif')

        cwp_l_l = Label(window, image=light_ped_b)
        cwp_l_l.place(x=310, y=375)
        cwp_l_r = Label(window, image=light_ped_t)
        cwp_l_r.place(x=365, y=460)
        cwp_m_l = Label(window, image=light_ped_b)
        cwp_m_l.place(x=540, y=375)
        cwp_m_r = Label(window, image=light_ped_t)
        cwp_m_r.place(x=595, y=460)
        cwp_u_r = Label(window, image=light_ped_l)
        cwp_u_r.place(x=675, y=345)
        cwp_u_l = Label(window, image=light_ped_r)
        cwp_u_l.place(x=595, y=401)
        cwp_b_r = Label(window, image=light_ped_l)
        cwp_b_r.place(x=675, y=645)
        cwp_b_l = Label(window, image=light_ped_r)
        cwp_b_l.place(x=590, y=701)

    if number == 2:
        light_ped_b = PhotoImage(file='lights/green_bottom.gif')
        light_ped_t = PhotoImage(file='lights/green_top.gif')
        light_ped_r = PhotoImage(file='lights/green_right.gif')
        light_ped_l = PhotoImage(file='lights/green_left.gif')

        cwp_l_l = Label(window, image=light_ped_b)
        cwp_l_l.place(x=310, y=375)
        cwp_l_r = Label(window, image=light_ped_t)
        cwp_l_r.place(x=365, y=460)
        cwp_m_l = Label(window, image=light_ped_b)
        cwp_m_l.place(x=540, y=375)
        cwp_m_r = Label(window, image=light_ped_t)
        cwp_m_r.place(x=595, y=460)
```

```
        cwp_u_r = Label(window, image=light_ped_l)
        cwp_u_r.place(x=675, y=345)
        cwp_u_l = Label(window, image=light_ped_r)
        cwp_u_l.place(x=595, y=401)
        cwp_b_r = Label(window, image=light_ped_l)
        cwp_b_r.place(x=675, y=645)
        cwp_b_l = Label(window, image=light_ped_r)
        cwp_b_l.place(x=590, y=701)


def display_waiters(waiters):
    '''
    displays the number of waiters during the simulation
    '''
    w_ll = Label(window, text=str(waiters['crosswalk_L_L']))
    w_ll.place(x=330, y=350)
    w_lr = Label(window, text=str(waiters['crosswalk_L_R']))
    w_lr.place(x=349, y=495)

    w_ml = Label(window, text=str(waiters['crosswalk_M_L']))
    w_ml.place(x=550, y=350)
    w_mr = Label(window, text=str(waiters['crosswalk_M_R']))
    w_mr.place(x=580, y=495)

    w_uu = Label(window, text=str(waiters['crosswalk_U_U']))
    w_uu.place(x=705, y=365)
    w_ub = Label(window, text=str(waiters['crosswalk_U_B']))
    w_ub.place(x=590, y=370)

    w_bu = Label(window, text=str(waiters['crosswalk_B_U']))
    w_bu.place(x=710, y=665)
    w_bb = Label(window, text=str(waiters['crosswalk_B_B']))
    w_bb.place(x=565, y=685)


def count_cars_waiting(raster, str):
    '''
    counts the number of waiting cars
    '''
    waiting_cars = 0
    if str == 'L_U':
        i = 35
        while raster[i][63] == 2:
            waiting_cars += 1
            i -= 1
    if str == 'L_B':
        i = 65
        while raster[i][63] == 2:
            waiting_cars += 1
            i -= 1
    if str == 'R_U':
        i = 42
```

```
        while raster[i][66] == 2:
            waiting_cars += 1
            i += 1
    if str == 'R_B':
        i = 71
        while raster[i][66] == 2:
            waiting_cars += 1
            i += 1
    return waiting_cars


""" ------------------------ Main function ---------------------------- """
tk = Tk()
window = Canvas(tk, width=LENGTH, height=LENGTH)
schachbrett(window)
tk.title("simulation_try4")
window.pack()
# K: sets the map in the background
img = Image.open('map_nico.gif')
backgr = ImageTk.PhotoImage(img)
canvas_img = window.create_image(503, 500, image=backgr)

# Variables, Arrays, Matrix for iteration
amount_ped, amount_car = spawning_frequency(variante_zeiten)
t_display = 0
walkers = []
drivers = []
tram = []
raster = initialize_gitter()

with open('values.csv', 'w') as f:
    w = csv.DictWriter(f, waiting_ped2.keys())
    w.writeheader()

    for i in range(300):

        # Creates new agents in the lists (including random startingpoints)
        spawn_tram(tram, i, raster)
        spawn_ped(walkers, i)
        spawn_cars(drivers, i)

        display_waiters(waiting_ped)
        waiting_ped['time_t'] += 1
        temp_time = waiting_ped['time_t']

        # counts cars queing up in front of a redlight
        cars_waiting_L_U = count_cars_waiting(raster, "L_U")
        cars_waiting_L_B = count_cars_waiting(raster, "L_B")
        cars_waiting_R_U = count_cars_waiting(raster, "R_U")
        cars_waiting_R_B = count_cars_waiting(raster, "R_B")
        waiting_ped2 = {'time': waiting_ped['time_t'],
```

```
                'crosswalk_L': waiting_ped['crosswalk_L_R'] +
waiting_ped['crosswalk_L_L'],
                'crosswalk_R': waiting_ped['crosswalk_M_L'] +
waiting_ped['crosswalk_M_R'],
                'crosswalk_U': waiting_ped['crosswalk_U_U'] +
waiting_ped['crosswalk_U_B'],
                'crosswalk_B': waiting_ped['crosswalk_B_B'] +
waiting_ped['crosswalk_B_U'],
                'cars_L': cars_waiting_L_B + cars_waiting_L_U, 'cars_R':
cars_waiting_R_B + cars_waiting_R_U}
        w.writerow(waiting_ped2)
        waiting_ped = {'time_t': temp_time, 'crosswalk_L_L': 0,
'crosswalk_L_R': 0,
                       'crosswalk_M_L': 0, 'crosswalk_M_R': 0,
'crosswalk_U_U': 0,
                       'crosswalk_U_B': 0, 'crosswalk_B_U': 0,
'crosswalk_B_B': 0,
                       'cars_L': 0, 'cars_M': 0, 'cars_U_U': 0, 'cars_U_B':
0,
                       'cars_B_U': 0, 'cars_B_B': 0}


        t_s += 1
        if t_s >= 60:
            t_min += 1
            t_s = 0
        if t_min >= 60:
            t_h += 1
            t_min = 0
        print_time(t_s, t_min, t_h)

        if variante_ampeln == True:
            # Activates / Deactivates Red light
            if i % light_phase == 0:
                raster = rotlicht(0, raster)
                raster = rotlicht(1, raster)
                # Setting traffic lights for when pedestrians have to wait
                light_ped_b = PhotoImage(file='lights/red_bottom.gif')
                light_ped_t = PhotoImage(file='lights/red_top.gif')
                light_ped_r = PhotoImage(file='lights/red_right.gif')
                light_ped_l = PhotoImage(file='lights/red_left.gif')

                cwp_l_l = Label(window, image=light_ped_b)
                cwp_l_l.place(x=310, y=375)
                cwp_l_r = Label(window, image=light_ped_t)
                cwp_l_r.place(x=365, y=460)
                cwp_m_l = Label(window, image=light_ped_b)
                cwp_m_l.place(x=540, y=375)
                cwp_m_r = Label(window, image=light_ped_t)
                cwp_m_r.place(x=595, y=460)
                cwp_u_r = Label(window, image=light_ped_l)
                cwp_u_r.place(x=675, y=345)
```

```
                cwp_u_l = Label(window, image=light_ped_r)
                cwp_u_l.place(x=595, y=401)
                cwp_b_r = Label(window, image=light_ped_l)
                cwp_b_r.place(x=675, y=645)
                cwp_b_l = Label(window, image=light_ped_r)
                cwp_b_l.place(x=590, y=701)

            if i % light_phase == car_phase:
                raster = rotlicht(0, raster)
                raster = rotlicht(2, raster)
                # setting traffic lights for when pedestrians can walk
                light_ped_b = PhotoImage(file='lights/green_bottom.gif')
                light_ped_t = PhotoImage(file='lights/green_top.gif')
                light_ped_r = PhotoImage(file='lights/green_right.gif')
                light_ped_l = PhotoImage(file='lights/green_left.gif')

                cwp_l_l = Label(window, image=light_ped_b)
                cwp_l_l.place(x=310, y=375)
                cwp_l_r = Label(window, image=light_ped_t)
                cwp_l_r.place(x=365, y=460)
                cwp_m_l = Label(window, image=light_ped_b)
                cwp_m_l.place(x=540, y=375)
                cwp_m_r = Label(window, image=light_ped_t)
                cwp_m_r.place(x=595, y=460)
                cwp_u_r = Label(window, image=light_ped_l)
                cwp_u_r.place(x=675, y=345)
                cwp_u_l = Label(window, image=light_ped_r)
                cwp_u_l.place(x=595, y=401)
                cwp_b_r = Label(window, image=light_ped_l)
                cwp_b_r.place(x=675, y=645)
                cwp_b_l = Label(window, image=light_ped_r)
                cwp_b_l.place(x=590, y=701)

        # Iterate all agents for one time period
        iterate(walkers, raster)
        i = 1
        while i <= car_speed:
            if i % 2:
                iterate(tram, raster)
            iterate(drivers, raster)

            tk.update()
            if maxspeed == False:
                time.sleep(animation_speed)
            i += 1

tk.mainlooptk.mainloop
```