Modeling and Simulation

Traffic Jam

Valentin Bauer
Studienkennzahl - 01425252
Philip Vonderlind
Studienkennzahl - 11739128
Dario Giovannini
Studienkennzahl - 01227603
Julia Putz
Studienkennzahl - 51841360

February 4, 2023
Supervisor: Madlen Martinek

Contents

Description

Research on traffic jams, in particular those not caused by obvious causes like car accidents or joint lanes, is crucially important topic to improve the quality of commuting in the surrounding of cities. Hereby the most important question is, which car density levels cause the fluid traffic to break down, dependent on the number of lanes and the maximum allowed velocity.

1 Model Analysis

As a first step, the model is analyzed analytically. It is assumed that a car is always able to brake and therefore not cause an accident.

1.1 Sizes

The cell size is chosen to be the average size of a car, namely 4 meters. This also includes a small buffer in front and the back of the car. The entire lane consists of 200 cells (= 800 meters) per default. Moreover, if a car leaves the lane at the right, it re-enters the lane at the left.

A timestep is set to be 1 second. It has to move the car at least one cell forward, if it is currently driving.

1.2 Speed

The maximum speed is chosen to be similar to the maximum speed permitted on austrian freeways. Of course, the cell size and time step also have to be taken into account. Hence, the maximum speed is chosen to be 144 km/h, which is equivalent to 8 cells. The minimum speed, unless the car has stopped completely, would be 14.4 km/h.

Within the model, a car can move between 0 and 8 cells (0, 18, ..., 144 km/h). For the cell size of 4 meters and a time step of one second, multiples of 14.4 km/h are accessible. The speed is represented by the current cell state, where '-1' represents an empty cell.

1.3 Granularity

In order to obtain a finer granularity, one option would be increasing the time step. That way the speed is more granular, but on the other hand the simulation itself becomes less granular.

2 Single-Lane Model

A single lane model, where cars leave the lane on the right side and re-enter it on the left side, is implemented using Python. Therefore, the number of cars stays the same throughout the entire simulation. The simulation mainly depends on the dawning factor, maximum velocity and the number of cars. Also the lane length can be specified manually. The code for the entire implementation can be found in our git repository.

2.1 Initial Setup

The basic setup consists of a street which is represented by a numpy array. The length of the street is determined by the 'lane length' which can be selected before starting a simulation. The street is then initialized with a previously specified number of cars, with a randomly selected speed between 0 and maximum velocity. They are placed on random positions of the street. Meaning, random elements of the street array are set to the speed value and the rest is set to '-1', indicating there is no car.
The corresponding code can be found in 'src/ca.py'. A short snippet of the code is displayed in Listing 1. Even the setup for the single-lane model already allows multiple lanes. For the beginning, it is simply assumed that the _*lanes* parameter is set to 1. As already stated above, first the velocities of the cars are randomly selected (Line 4-6). The street is set up and filled with -1 in Line 8-9. Finally, the velocities can be inserted at random indices, to fill the empty street with cars (Line 10 - 15).

```
 1 class Street:
 2     .....
 3     def _init_state(self, seed: int) -> np.ndarray:
 4         rand_gen = np.random.RandomState(seed)
 5         velocities = [rand_gen.randint(0, self._v_max)
 6                             for _ in range(self._n_cars)]
 7 
 8         flat_len = self._lanes * self._lane_len
 9         flat_street = np.full(flat_len, -1)
10         indices = np.arange(flat_street.size)
11         np.random.seed(seed)
12         np.random.shuffle(indices)
13 
14         car_idxs = indices[:self._n_cars]
15         flat_street[car_idxs] = velocities
16         return flat_street.reshape(self._lanes, self._lane_len)
17     .....
```

Listing 1: Initial street setup

2.2 Rules

Each time step the street is updated based on the following rules. These rules are implemented in seperate classes, named accordingly, within the file 'src/rules.py'.

2.2.1 Accelerate

The first rule checks if the current speed of every car, which is denoted by the state, is smaller than the maximum velocity. If this is the case, the vehicle speed is increased by one, otherwise the speed stays the same (Figure 1). Here it is important to only check elements/cells which contain a value higher or equal to zero, in order to avoid cars to appear magically, which happens if a '-1' is increased. Hence, every element in the *state* array which fulfills the mentioned condition, implemented as *check_velocity*, is increased (Listing 2 / Line 4-5)

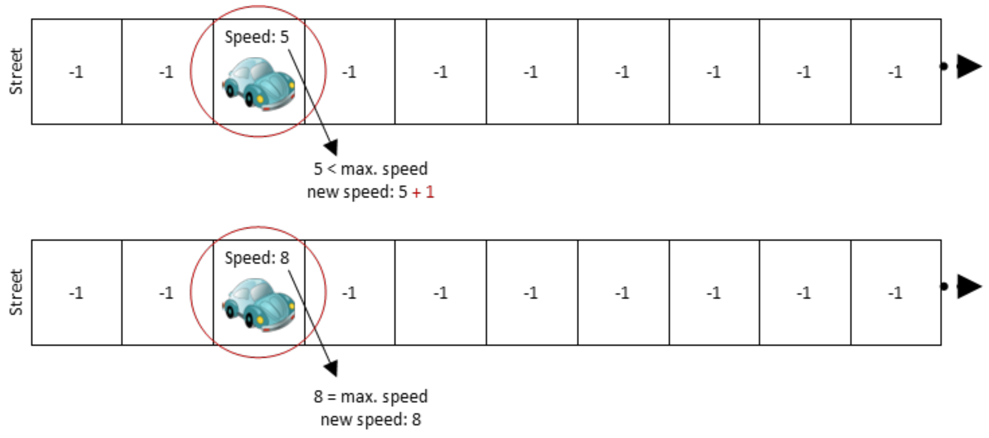


Figure 1: Accelerate (assuming a maximum speed of 8)

```python
class Accelerate(AbstractRule):
    .....
    def apply(self, state: np.ndarray) -> np.ndarray:
        check_velocity = (state < self.v_max) & (state >= 0)
        state[check_velocity] += 1
        return state
```

Listing 2: Accelerate

2.2.2 Avoid Collision

A very important rule is to avoid collisions. Therefore, it is checked for every car if the following x cells contain a car, where x is equal to the current speed of the observed car. Implementation wise we simply loop over the 'state' array and check the following aspects for every element (Listing 3 / Line 3). Looking at Figure 2, the cells which have to be checked are highlighted grey. Within the code, these cells are defined in the function *check_following_vehicles* (Listing 3 / Line 11-14). In case there is a car within this area, the 'gap size' to the closest car is calculated by simply counting the cells between, which is the return value of *get_gap* (Line 16-17). The speed of the observed car is then minimized to the 'gap size' (Line 7-8). This scenario is displayed in the top part of Figure 2. In case there is no car within the observed area, the speed simply stays the same. Like displayed in Line 4, it is important to only check cells with a speed higher than 0, as it is only necessary to check actually moving cars. As in the previous rule, this helps to prevent magically appearing cars.

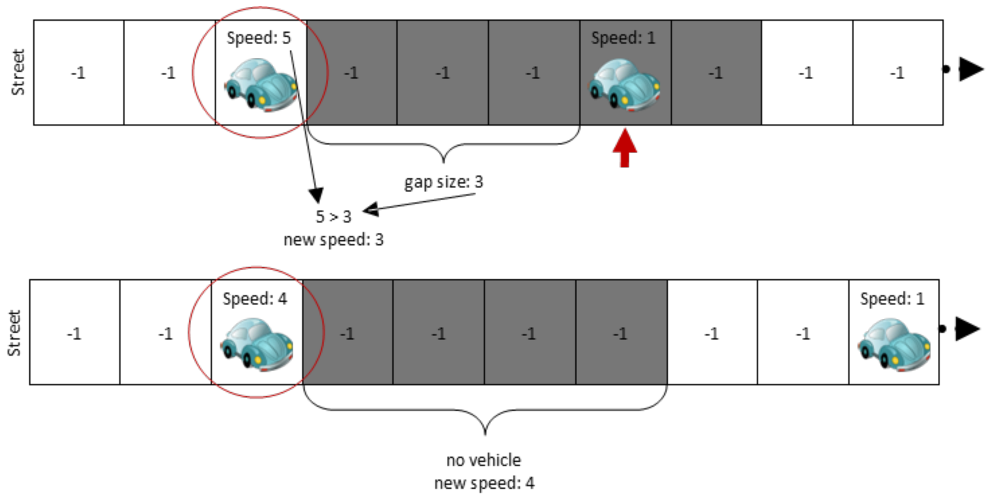


Figure 2: Avoid Collision

```
class AvoidCollision(AbstractRule):
    def apply(self, state: np.ndarray) -> np.ndarray:
        for index, speed in enumerate(state):
            if(speed>0):
                gap_ahead_of_car = self.check_following_vehicles(state, index,
                    speed)
                if gap_ahead_of_car.any():
                    reduced_speed = self.get_gap(gap_ahead_of_car)
                    state[index] = reduced_speed
        return state

    def check_following_vehicles(self, lane: np.ndarray, index: int, speed: int)
        -> np.ndarray:
        shifted_state = np.roll(lane, -(index+1))
        gap_ahead_of_car = shifted_state[:speed]
        return gap_ahead_of_car >= 0

    def get_gap(self, gap_ahead_of_car: np.ndarray) -> int:
        return np.where(gap_ahead_of_car)[0][0]
```

Listing 3: Avoid Collision

2.2.3 Dawdling

Another way to slow down cars is based on the dawning factor. With the probability p_d, which can be set manually before starting a simulation, the speed of a car is reduced by 1. This is displayed in Figure 3. It is important to only apply this rule to cells where the car is currently moving. Meaning, to cells with a state higher than 0, in order to not make cars disappear by decreasing a 0. Within the code, this is realized by using the condition *check_speed*. First, a new array with the same length as the street is randomly filled with either 0 or 1, based on the passed dawning factor (Listing 4 / Line 4-5). Those elements which fulfill the previously mentioned condition are set to 0 in the newly generated array (Line 7). Lastly, this array is simply subtracted from the passed state (Line 8).

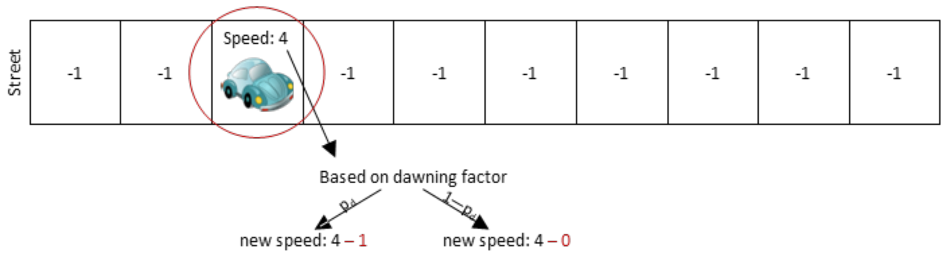


Figure 3: Dawdling

```python
class Dawdling(AbstractRule):
    .....
    def apply(self, state: np.ndarray) -> np.ndarray:
        selected = self.rand_gen.choice([0, 1], state.shape,
                                        p=[1 - self.dawning_fac, self.dawning_fac])
        check_speed = (state <= 0)
        selected[check_speed] = 0
        return state - selected
```

Listing 4: Dawdling

2.2.4 Move Forward

The last basic rule is to move each car forward, like it is displayed in Figure 4. Therefore, the index of the new position is calculated by simply adding the speed of the observed car to the current index. The state is then inserted into a new empty street, which only contains '-1' at the beginning, at the calculated new position. An important case to consider is, if the new position would be higher than the lane length (the car would leave the street), the lane length is subtracted from the new position. That way the car reenters the road at the left side. This case, as well as the addition of the speed to the index are implemented exactly like described in the separate function *get_new_position* (Listing 5 / Line 2-7). Also, like mentioned above, first an empty street *new_state* is initialized (Line 10). Then, for every car (elements where speed is higher or equal to 0), the new position is calculated. Where finally the speed of the currently observed car is inserted into *new_state* at the just calculated *new_position* (Line 12-15).

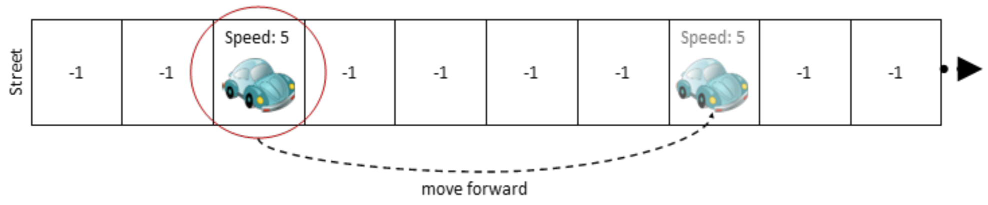


Figure 4: Move forward

```python
class MoveForward(AbstractRule):
    def get_new_position(self, lane: np.ndarray, index: int, speed: int) -> int:
        new_position = index + speed
        lane_len = lane.shape[0]
        if new_position >= lane_len:
            new_position -= lane_len
        return new_position

    def apply(self, state: np.ndarray) -> np.ndarray:
        new_state = np.full_like(state, -1, dtype=int)

        for index, speed in enumerate(state):
            if speed >= 0:
                new_position = self.get_new_position(state, index, speed)
                new_state[new_position] = speed
        return new_state
```

Listing 5: Move Forward

3 Multi-Lane Model

In case of a multi-lane road, the initial array is exchanged by a multi-dimensional array, in which each row represents one lane. Hence, some rule extensions and adjustments are necessary. The basic concept however, stays the same.

3.1 Extension of Avoid Collision to allow Overtake

One major difference to the single-lane model is, that cars are allowed to overtake. Therefore, the considered neighborhood has to include the default neighborhood plus the default neighborhood one lane over shifted one cell forward in the driving direction. This neighborhood is highlighted grey in Figure 5 and represented by the Code Lines 23-27 in Listing 6. In case there is a car within the default neighborhood (Line 11), but none in the shifted neighborhood (Line 12), the observed car is allowed to overtake. Meaning, it switches to the left lane instead of slowing down (Line 13-14). If the left lane is not clear, the car stays at its current lane and removes its speed to the gap size, like in the initial Avoid Collision rule (Line 16-18). As a car can only switch one lane per time step, it is important to make sure the overtaking is only executed once per car. This is implemented by simply adding the index of a car that took over to an array (Line 15) and check before generating the neighbourhood if the current cars' index is already saved in that array (Line 7).

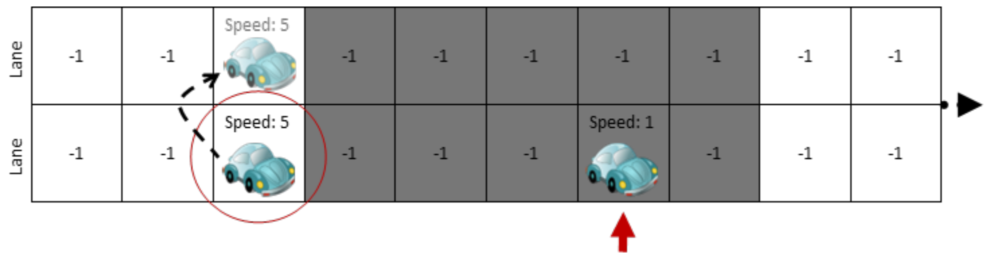


Figure 5: Overtake

```
class BreakOrTakeOver(AbstractRule):
    def apply(self, state: np.ndarray) -> np.ndarray:
        cars_indices_that_took_over = []

        for i, lane in enumerate(state):
            for index, speed in enumerate(lane):
                if index in cars_indices_that_took_over:
                    continue
                if speed > 0:
                    gap_ahead_of_car = self.check_following_vehicles(lane, index, speed)
                    if gap_ahead_of_car.any():
                        if (i < (state.shape[0] - 1)) and self.is_left_lane_clear(state, i, index, speed):
                            state[i, index] = -1
                            state[i + 1, index] = speed
                            cars_indices_that_took_over.append(index)
                        else:
                            reduced_speed = self.get_gap(gap_ahead_of_car)
                            state[i, index] = reduced_speed

            cars_indices_that_took_over = []
        return state
    .....
    def is_left_lane_clear(self, state: np.ndarray, lane_idx: int, index: int, speed: int) -> bool:
        if state[lane_idx + 1, index] != -1:
            return False
        left_gap = self.check_following_vehicles(state[lane_idx + 1], index, speed)
        return not left_gap.any()
    .....
```

Listing 6: Overtake

3.2 Adjustment of Move Forward

Move Forward needed a simple adjustment in order to handle multi-dimensional arrays properly. Therefore, simply an additional loop was added to loop over all lanes and apply the initial moving process to each lane separately (Listing 7 / Line 5).

```
1  class MoveForward(AbstractRule):
2      .....
3      def apply(self, state: np.ndarray) -> np.ndarray:
4          .....
5          for i, lane in enumerate(state):
6              for index, speed in enumerate(lane):
7                  if speed >= 0:
8                      new_position = self.get_new_position(lane, index, speed)
9                      new_state[i, new_position] = speed
10         return new_state
```

Listing 7: Adjusted Move Forward

3.3 MergeBack

Due to the right-hand driving rule, cars have to merge back to the right-hand lane if there is enough space. Therefore, it is checked if the new position, namely current position + speed, is empty. If this is the case, the car switches to the right lane, like displayed in Figure 6. Moreover, it is important to note that a car can only swap one lane per time step, not multiple ones. This is why the swapping is done in reverse order.

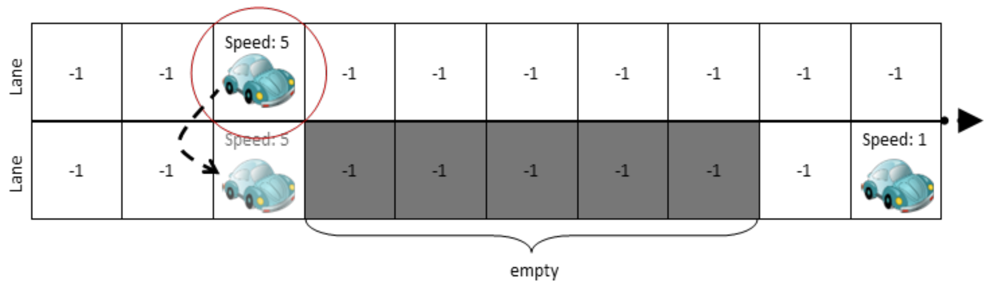


Figure 6: Merge back

```python
class MergeBack(AbstractRule):
    def apply(self, state: np.ndarray) -> np.ndarray:
        for i, lane in enumerate(state):
            if i == state.shape[0] - 1:
                break
            for index, speed in enumerate(lane):
                if state[i, index] == -1:
                    if state[i + 1, index] != -1:
                        state[i, index] = state[i + 1, index]
                        state[i + 1, index] = -1
        return state
```

Listing 8: Merge Back

4 Experiments & Sensitivity Analysis

The following section presents the results of a sensitivity analysis performed in the course of this assignment. This analysis was conducted to investigate the impact of different parameter values on the simulation outcome, in order to understand the system's sensitivity to variations in specific variables and determine the range of values that yield reasonable results. The results provide insight into the robustness of the simulation model and help optimize its performance.

Primary focus was put on identifying a breakdown point for the fluidity of traffic, which was evaluated via the average car speed across all timesteps of a simulated experiment. To achieve this, the average value of all cells that were not empty (that is, had a value greater than -1) was taken and recorded in each timestep, normalized by the maximum velocity to maintain comparability across different values for this parameter. These averages were then aggregated and averaged across all timesteps to give a single value which quantizes traffic fluidity for a specific parameter setup.

There are three principal variables that have a direct impact on traffic fluidity: The maximum speed allowed (v_{max}), the dawdling factor which controls the random chance of a car slowing down (p_d), and the number of cars present on the road (n). All experiments presented here were run over 1000 timesteps, and repeated five times for each set of parameters to reduce variability due to random initialization and the inherent randommness of the dawdling factor p_d.

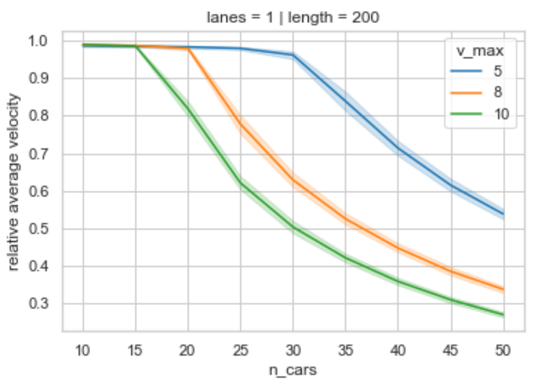
4.1 Parameter sweep in one lane

The first set of experiment focuses on comparing the impact of n vs p_d as well as v_{max} by running several experiments that cover a range of parameter values. Specifically, the following values were used here:

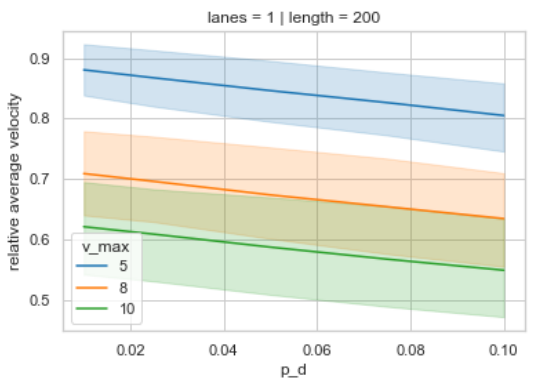
parameter	v_{max}	p_d	n
values	5, 8, 10	1%, 2.5%, 5%, 7.5%, 10%	10-50 (steps of 5)

Table 1: Parameter space for the first experiment, in one lane of length 200

This results in a total of 135 runs, repeated 5 times each to reduce variability. The results, as shown below graphically, illustrate three findings immediately.



(a) Velocity plot vs n_{cars} for different v_{max}



(b) Velocity plot vs p_d for different v_max

Figure 7: Average speed as a function of n and p_d at different v_{max}

First, it is evident that the number of cars has a significant impact on car fluidity as measured by the average velocity, and figure 7a illustrates this as the characteristic break in the line can be identified easily at 15, 20 and around 30 cars for maximum speeds of 10, 8 and 5 immediately. This also shows the second finding, namely that reducing the maximum velocity increases the capacity of the road. The intuitive explanation for this behaviour is that a lower maximum velocity reduces the (average) number of cars any one car "sees", and thus can be impacted by their randomly slowing down.

Lastly, from figure 7b it can be concluded that the dawdling factor does not contribute greatly to the breakdown point where traffic fluidity is sharply reduced. Rather, as long as a relevant chance for slowing down is present, it is the traffic density (given by the number of cars per road length) that primarily determines the formation of traffic jams. Very low and very high values for p_d were not included in this parameter sweep, although conclusions can be drawn about the behaviour of the simulation at those extremes: At very high dawdling chances, it effectively acts as a reduction in maximum velocity as the acceleration in each timestep has a significant chance to be undone immediately. At very low dawdling chances however it still only takes a single slowdown to cause a traffic jam at sufficient densities.

The immediate next question is to verify the assumption that car density, rather than another effect influenced by the number of cars, is responsible for the fluidity breakdown. To test this, the hypothesis that a road that is twice as long should be able to accommodate twice as many vehicles is formed. To test this, the same experiments as before are repeated, for a slightly adjusted range of n (20-60 instead of 10-50) and with a constant value of 5% for the dawdling factor p_d, on a road that is 400 units long instead of the previous 200. The result is shown in the plot below.

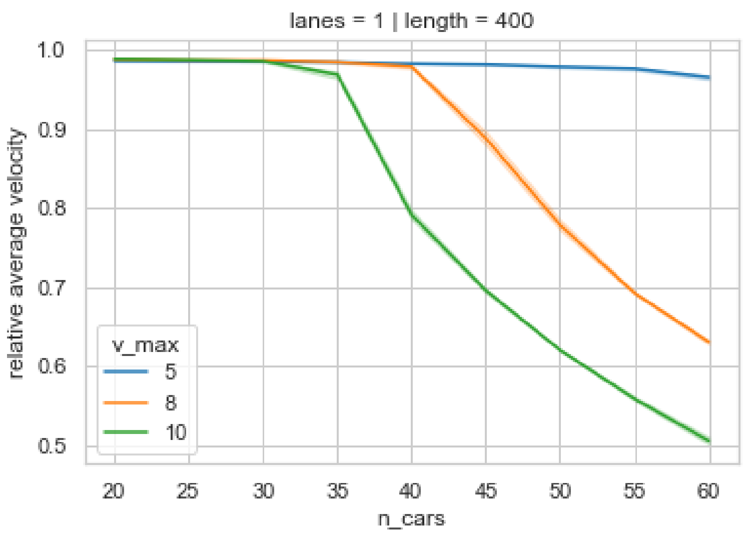


Figure 8: Average speed as function of n on a long road at different v_{max}

Compared to figure 7a, figure 8 shows a very convincing doubling of the number of cars before the breakdown point is reached - from 15 to 30 for $v_{max} = 10$, from 20 to 40 for $v_{max} = 8$, and from somewhere between 25 and 30 to somewhere between 55 and 60 for $v_{max} = 5$. This supports the conclusion that the car density - that is, the number of cars per road length - is the principal contributor to the phenomenon of traffic fluidity breakdown.

4.2 Experiments with increasing lane count

The experiments with one lane showed that the car density is paramount to predicting the formation of traffic jams, at least when it comes to road length. But what if we made the road not longer (which is an artifact of the simulation model and does not reflect real-world scenarios anyway), but wider? In this section, the effect of adding more lanes is investigated, once more under constant p_d (at 5%) as the influence of this parameter on the breakdown point has been shown to be negligible.

In theory, going from one lane to two lanes doubles the capacity on the road. The comparison between figure 9 and figure 7a however does not show this. Of note is the reduced spread, shown by the transparent colored area surrounding each curve, due to the exclusion of the random chance of the dawdling factor. So while the increase of lanes from 1 to 2 does show some improvement in the number of cars before the breakdown point is reached, this is only really visible for $v_{max} = 5$ where the breakdown occurs gradually enough to be captured by multiple datapoints. For v_{max} values of 8 and 10 the breakdown evidently occurs between 20 and 25, and 15 and 20 respectively and only shifts slightly in this range going from one lane to two lanes,

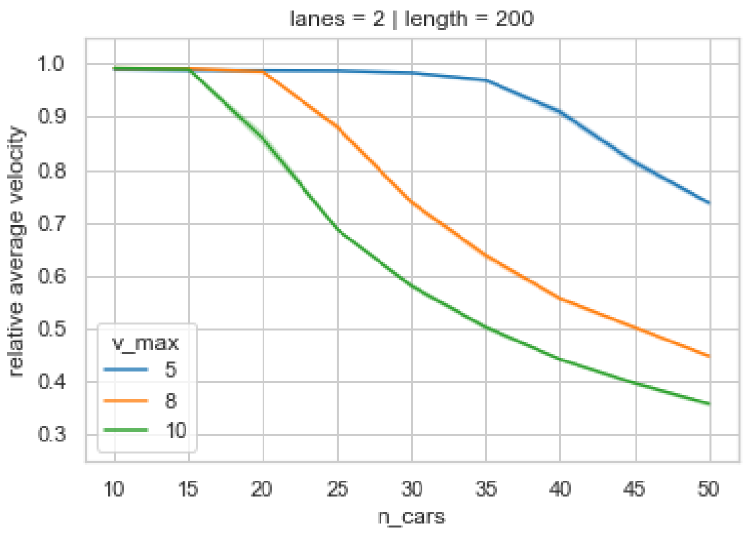


Figure 9: Average speed as a function of n on a 2-lane road

which results in the slightly less steep slope on this interval in figure 9.

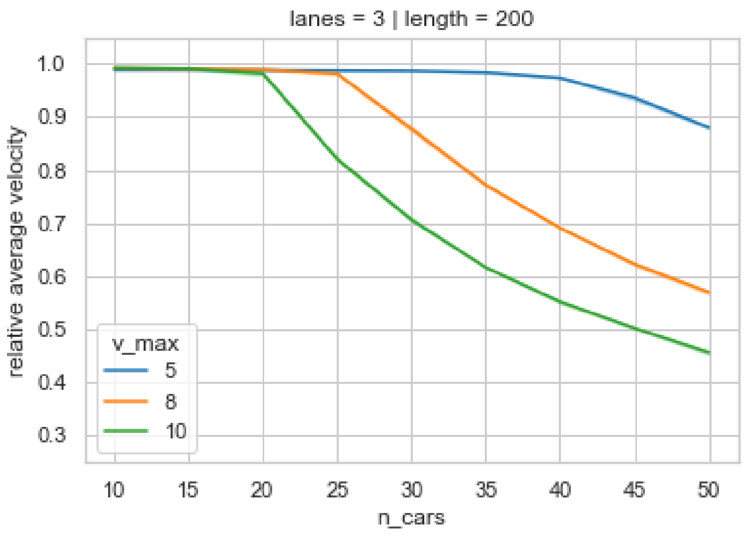


Figure 10: Average speed as a function of n on a 3-lane road

Going from 2 lanes to 3 lanes shows a larger difference in apparent breakdown points than before (figure 10), though likely this is more an artefact of the resolution in the parameter space than a real effect due to specifically a 3-lane setup. As mentioned before, going from 1 lane to 2 lanes improved the breakdown point only slightly, but not enough to put the next value of n in the "no breakdown" zone where the relative average velocity is close to 1. In turn, going from 2 lanes to 3 does seem cause the breakdown point to move to the next value.

Further increasing the lane count to 4 lanes moves the breakdown points slightly further yet again (figure 11). Compared to figure 7a however, it is evident that quadrupling the capacity in terms of actual space available did not quadruple the car capacity, unlike the experiment with

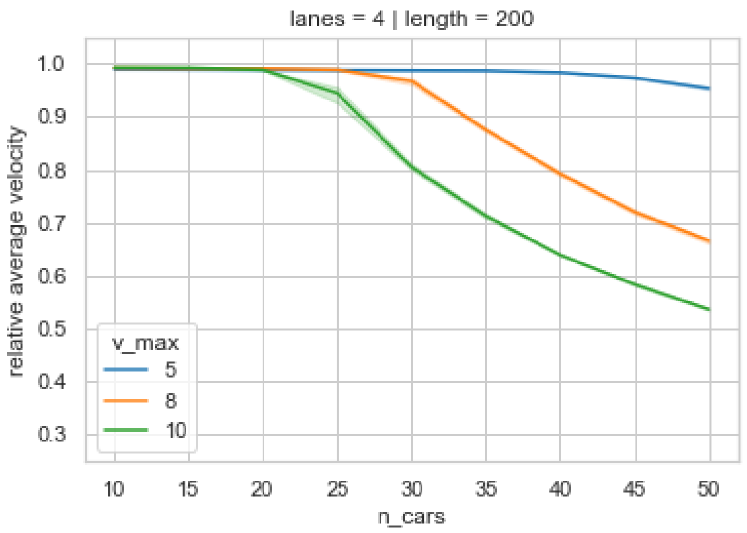


Figure 11: Average speed as a function of n on a 4-lane road

doubling the road length - which did double the car capacity (figure 8).

4.3 Sensitivity Analysis Conclusion

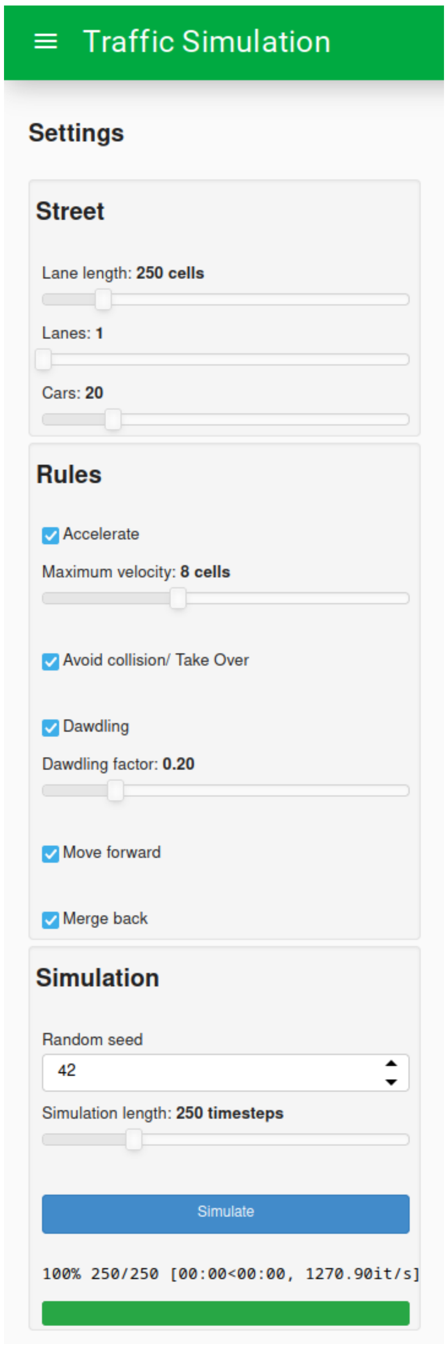
Of the three principal variables discussed at the beginning of this section, it has been shown that only v_{max} and n contribute significantly to the breakdown point where traffic fluidity is significantly impacted, while the dawdling factor p_d plays a background role.

Lowering the maximum speed showed an increase in car capacity, which is the number of cars a road can accommodate before the formation of traffic jams due to random slowdowns significantly impacts traffic fluidity as measured by average car velocity. Doubling the length of a road segment also doubled the capacity, which is intuitively immediately apparent, but also not a useful strategy for real-life applications.

When increasing the number of lanes, the increase in capacity was not proportional - that is, doubling the lanes from 1 to 2 or from 2 to 4 did not double the number of cars a road could accommodate before traffic jams started to form. A part of this effect is likely due to not entirely optimized rules regarding overtaking and merging, but unlike the doubling in length, doubling in width still has interactions between cars on neighbouring lanes. This means that the theoretical increase in space cannot be utilized fully, and in a way having a second, separate road would lead to more throughput increase than doubling the lane count on a single road.

5 UI

An UI is implemented, in order to visualise the simulations and get a better understanding about what is happening. The source code is located in 'src/ui.py. To make it run, simply run the 'python3 main.py' file and the UI automatically opens in your Browser window. For more detailed setup instructions please check the git repository. Now, one can select different parameters, like the amount of cars and lanes, maximum velocity, which rules to apply. This is displayed in Figure 12. By clicking 'Simulate', the simulation is started and can be observed on the right hand side (13. Additionally the car throughput and the average relative speed is calculated for each time step and visualised in the Metrics section (14). Lastly, there is the option to download the current simulation by clicking the corresponding button.



Traffic Simulation

Settings

Street

Lane length: **250 cells**

Lanes: **1**

Cars: **20**

Rules

✓ Accelerate

Maximum velocity: **8 cells**

✓ Avoid collision/ Take Over

✓ Dawdling

Dawdling factor: **0.20**

✓ Move forward

✓ Merge back

Simulation

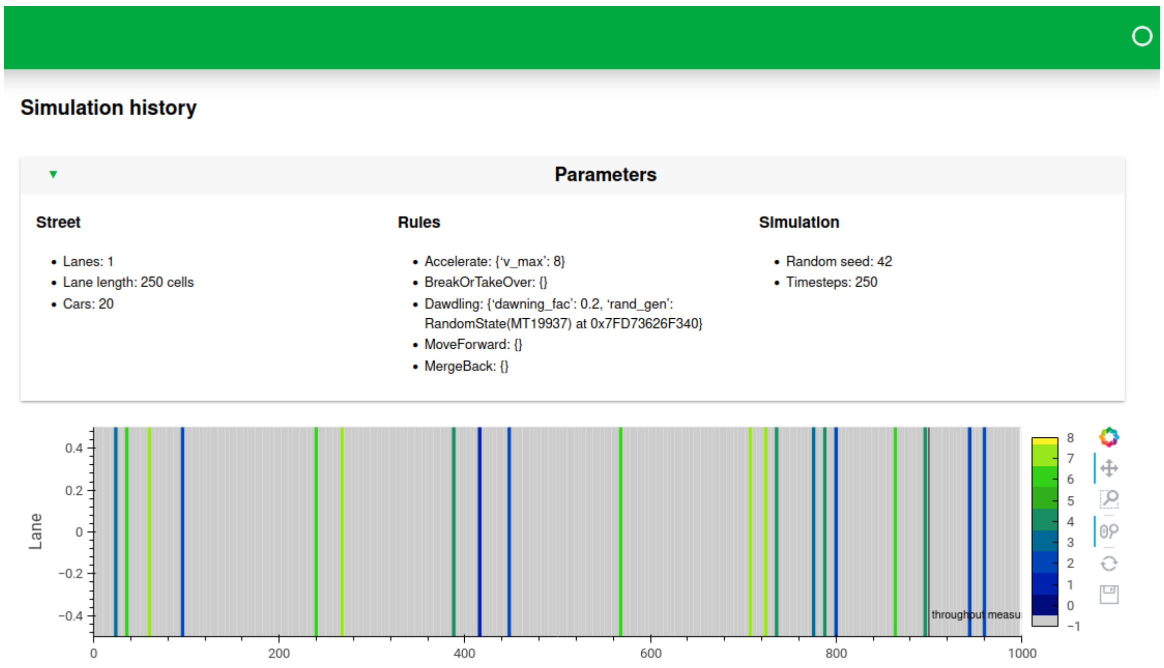
Random seed

42

Simulation length: **250 timesteps**

```
100% 250/250 [00:00<00:00, 1270.90it/s]
```

Figure 12: UI - set parameters



Simulation history

Parameters

Street

- Lanes: 1
- Lane length: 250 cells
- Cars: 20

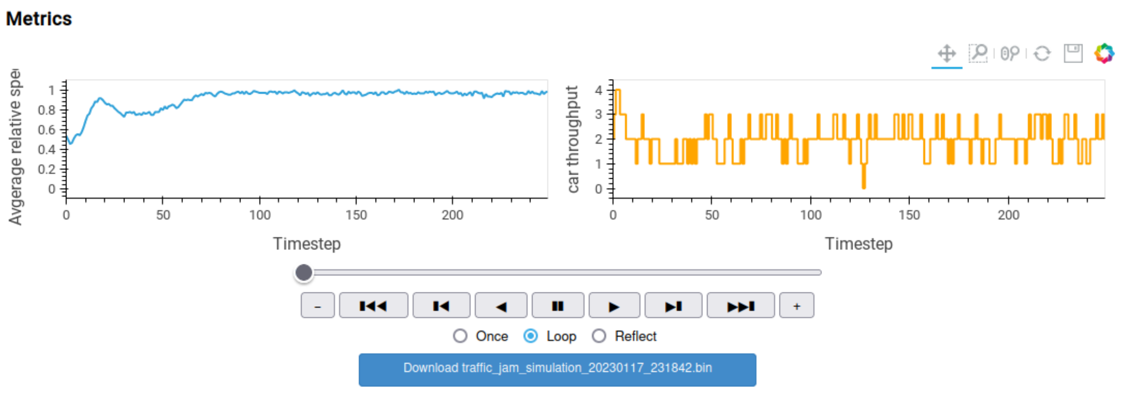
Rules

- Accelerate: {'v_max': 8}
- BreakOrTakeOver: {}
- Dawdling: {'dawning_fac': 0.2, 'rand_gen': RandomState(MT19937) at 0x7FD73626F340}
- MoveForward: {}
- MergeBack: {}

Simulation

- Random seed: 42
- Timesteps: 250

Figure 13: UI - Simulation history & visualization



Metrics

Figure 14: UI - Simulation metrics

6 Summary

Finally, it can be said that the main challenge was not understanding the model and its rules or the basic implementation, but the bug fixing. One issue we encountered here, was that cars tended to disappear magically, which of course should not happen. Finding the problem which lead to this phenomenon was quite time-consuming. Other than that, it at first seemed somehow surprising, that the dawdling factor does not play a big role when it comes to the traffic fluidity. At a second glimpse it definitely makes sense, because if e.g. the probability is high that a car slows down, the average speed will simply be reduced. It would be interesting to fine-tune the rules and add some more to make the simulation even more realistic.

However, we have also seen that the boundaries of modelling with Cellular Automatons quickly show up, because we can not extend our model to an arbitrary precision of speed levels or car sizes. We are restricted by the fact that a car has to be represented by a single cell. A car in our model is only unique by having a speed in the cell of the street, which implicitly defines the smallest cell size and thereby also the speed levels. To overcome these limitations we would need to switch the modelling approach, for example, to an Agent-based simulation, where cars can be uniquely identified and the environment, the street, is decoupled from the agents, the cars.