CS166 LBA - Berlin Traffic Modeling

Leo Ware

Introduction

For this project, I built an agent based model of traffic flow in central Berlin.The goal of this simulation was to study how traffic flows work and how changes in the way intersections are managed could improve traffic flow. To implement the model, I relied primarily on two sources: the Nagel–Schreckenberg traffic model, which I extended to account for higher dimensional network topologies, and traffic data available through Google maps. I used empirical data from Google maps to design the street network topology and choose the parameters for the model. I employed linear markov chains to do a theoretical analysis of the model.

I studied multiple ways to manage intersections. Each *policy* was distributed in the sense that it required no central management and could be implemented at each intersection with only local information. My conclusion was that traffic lights that always give the green light to the lane with the most number of cars waiting (*adaptive* traffic lights) perform significantly better than traffic lights which randomly choose lanes or alternate between lanes. To get an absolute sense of the power of adaptive traffic lights, I compared them to another, physically unrealistic form of intersection which can let any number of cars use the intersection at the same time. I called this a *clover leaf*, and it represents an upper-bound on what we could expect to achieve with good intersection policy. Adaptive traffic lights worked almost as well as clover leafs, such that the difference between an alternating and an adaptive traffic light is less than the difference between an adaptive traffic light and a clover leaf.

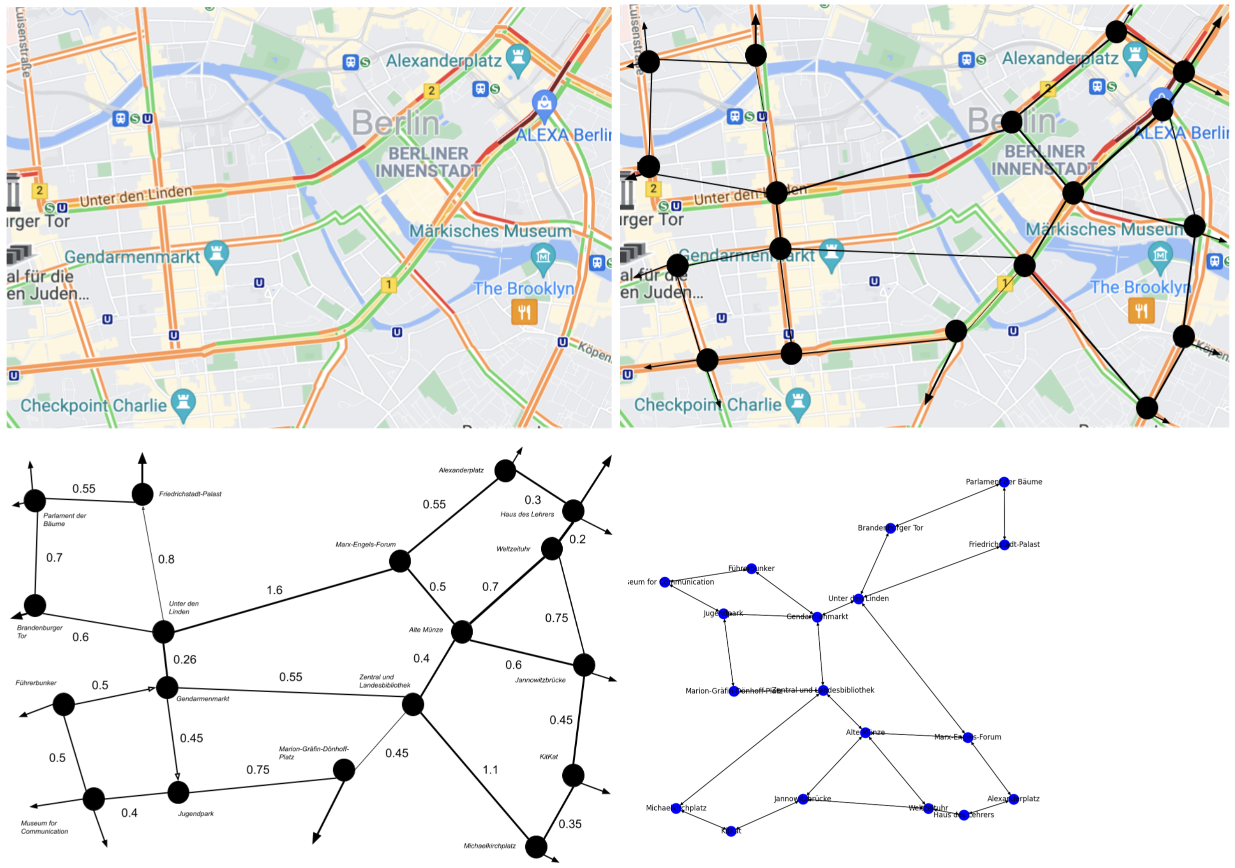
Data Collection and Network Modeling

For my model, I chose to study traffic patterns along main roads in central Berlin. To model the network topology, I took a screenshot of google maps and overlaid it with a diagram representing the main roads.

For each road, I had to determine some measure of the traffic flow and some measure of the length. To determine length, I used Google maps to estimate distances in kilometers. For traffic flow, I used the google maps traffic layer. The traffic layer overlays streets with four different colors---green, yellow, orange, and red---which represent the speed of traffic on the street. For each road section, I gave it an integer speed rating between 1 and 4 depending on the worst area of congestion along that section of the street according to google maps. For use in the program, I used this information to construct a graph in networkx. I found landmarks near each intersection and used these to name the nodes in the graph.

Figure 1 shows the progression of this process. First is the screenshot from google maps. Second is the network overlay. Third is the network with distances and landmarks added. In the third, edge widths represent traffic flow, with heavier weight representing more congested traffic. Fourth is the internal networkx representation displayed with a kamadi-kawai projection, which tries to respect edge lengths (note this is rotated 90 degrees relative to the other maps).

Figure 1. *Development of the road model*



Model Design

To model the traffic, I used a version of the Nagel–Schreckenberg (NS) model. This model represents traffic as a cellular automata with a few simple rules. The street is modeled as a series of a fixed number of discrete cells. Each cell can either contain a car or not. Each car has a speed between 0 and the *speed-limit*, which is a parameter of the model. Streets are one way.

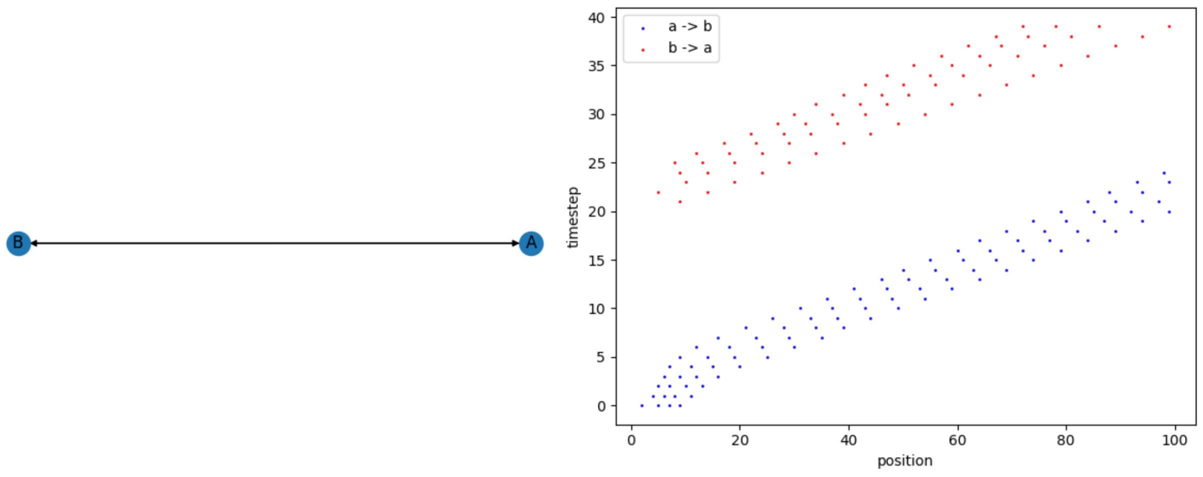
The model is discrete time. At each step, each car updates as follows.

1. *Acceleration*: each car increases its speed by 1.
2. *Slowing down*: each car's speed is decreased so that it is not higher than the speed limit and not higher than the number of spaces to the next car.

3. *Randomization*: each car decreases its speed by 1 with probability *p-slow*, a parameter of the model.
4. *Motion*: each car is moved forward a number of cells equal to its speed.

Since the NS model only models unidirectional traffic flow along one dimension, I used two parallel NS models (called *lanes*), one pointed in each direction, to model each *street*. Fig 2 demonstrates this concept for a simple road network consisting of two intersections connected by a single street---one lane in each direction. For this demonstration, four cars were started on the road near B, heading towards A. The image on the left shows the internal, networkx representation of this model. The space-time plot on the right shows the progress of the cars as the model runs (note that time runs up the y-axis). While they are on the lane towards B, they are marked blue. When they reach B and start returning to A, they are colored red.

Figure 2. *Simple two-intersection extension of the Nagel–Schreckenberg model*



When a car reaches the end of a lane, its behavior is governed by the intersection at the end of the lane. The goal of this project was to study the effect of different intersection designs on traffic patterns. To simplify the model, I assumed that when a car reached an intersection, it would choose an outgoing lane randomly. To simplify the coding, it is allowed for cars to pick the outgoing lane that goes in the opposite, parallel direction to the lane they just left. At each run, one policy would be chosen to model the behavior of all the intersections in the model.

In order to model intersections properly, I also added a feature where lanes can be blocked at the end. This prevents cars from leaving the lane by capping the velocity of the foremost car by its distance from the end of the lane.

I studied two main classes of intersection policy: the *clover leaf* and the *traffic light*. The *clover leaf* is an intersection that never delays cars; when they enter, they are immediately assigned to an outgoing lane. This is unrealistic, but simulating with the *clover leaf* gives us an upper bound on how well we can do when picking an intersection policy.

The *traffic light* blocks all incoming lanes but one (the *green* lane). Every *p* steps, it changes which lane is green according to different policies. There are a few different kinds of traffic lights. The *alternating traffic light* keeps incoming lanes on rotation, always setting the green lane to be the one which has been red for the longest time. The *random traffic light* picks a random lane to be green. The *adaptive traffic light* picks the lane with the longest queue of waiting traffic, where this is defined as the number of contiguous grid cells at the end of the street which are filled with cars.

Note that the traffic light policies all rely on only local information. Thus, there is no need for any kind of global coordination. The idea was to prescribe one of these policies as well as an optimal switching period for the light.

To model cars entering and leaving the system at the edges, I introduce two new concepts to the model. Instead of a normal lane, one of the incoming paths to an intersection can be an *on ramp*, which, if it is not blocked, generates a new car at each step with probability *p_new_car*, which is a parameter of the *on ramp*. Also, I introduce a *parking lot*, which can take the place of an outgoing lane from an intersection. Parkings lots keep an internal tally of the cars which have been added, but do nothing else with them. The number of parking lots attached to each intersection control the number of cars leaving the system at that intersection.

Theoretical Analysis

We can perform a theoretical analysis of a simplified version of the system by modeling the number of cars on each street as a positive continuous quantity. Recall that a street is a set of two lanes pointed in opposite directions which connect the same two intersections. So, we represent the state of the system as some time *t* using a positive, real-valued vector **X** of length *n*, where *n* is the number of roads in the system. We model the evolution of the system over time as a linear markov process. So, a step in the model is equivalent to left multiplying the state vector by a transition matrix **M**.

Consider the case where we make two simplifying assumptions: that it takes cars exactly one step to traverse a road and that no cars enter or leave the system. Then, to get the transition matrix, we can start with the adjacency matrix of the graph edges. So, a square *nxn* matrix, where the (*i, j*)th entry is a 1 if the ith and jth roads both connect to the same intersection. A street is considered adjacent to itself under this formulation. Each column *j* will represent what happens to the cars in street *j* at the end of each step. The (*i, j*)th will represent the portion of the cars in street *j* that go to street *i*. This means that the columns in the transition matrix must add to 1 if we do not want to introduce new cars.

By normalizing the columns of the edge adjacency matrix, we get the transition matrix of the system. This models the behavior of cars randomly choosing a street to go to from among the streets connected to their current one. I call this the *unadjusted* model.

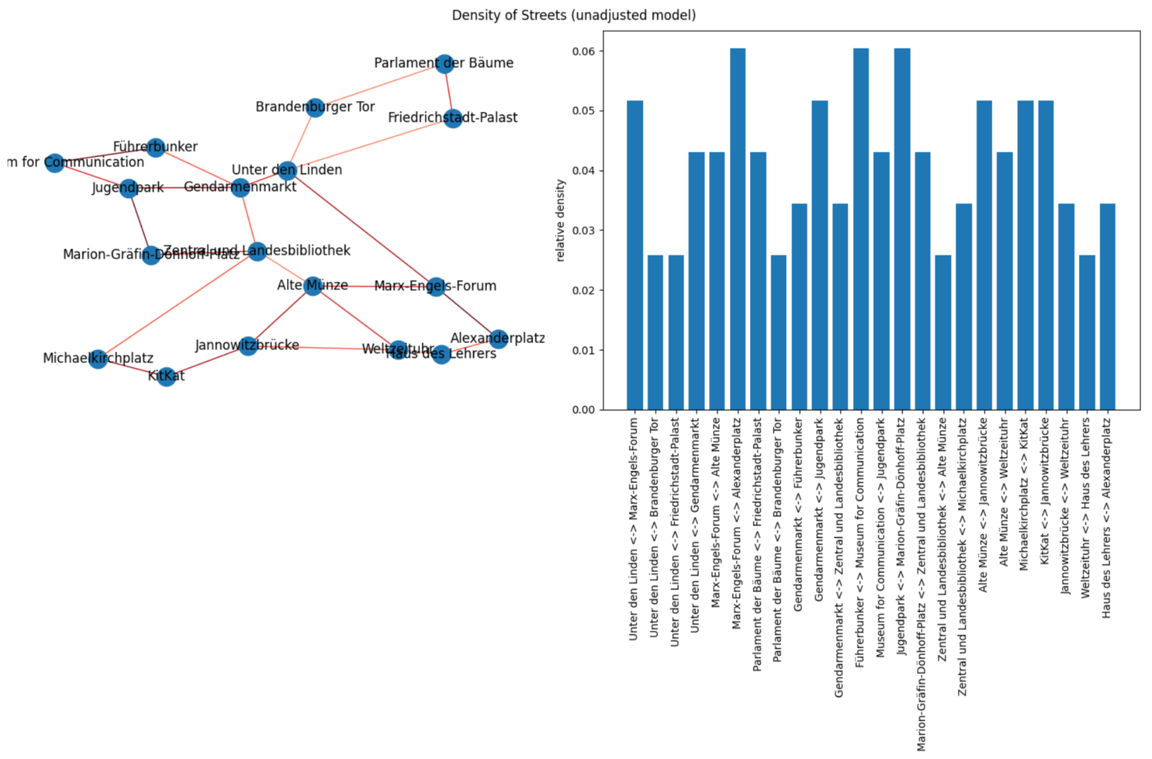
By modifying this matrix, we can relax the assumption that every street takes exactly one step for cars to traverse. We can augment the model with the additional feature that instead of choosing a new, adjacent lane at random at the end of each step, some portion *r_i* of cars remain on street *i*, modeling the fact that they are in transit along the road section. To get *r_i*, I made the simplifying assumption that it is proportional to length. Then, I set the largest *r_i* to 1/2 and scaled the others proportionally. The choice of 1/2 is arbitrary because I haven't defined the length of a timestep. We can scale each column *i* of the transition matrix so that its sum is *1-r_i*. Then, we can replace each entry (*i, i*) with *r_i*. I call this the *length-adjusted* model.

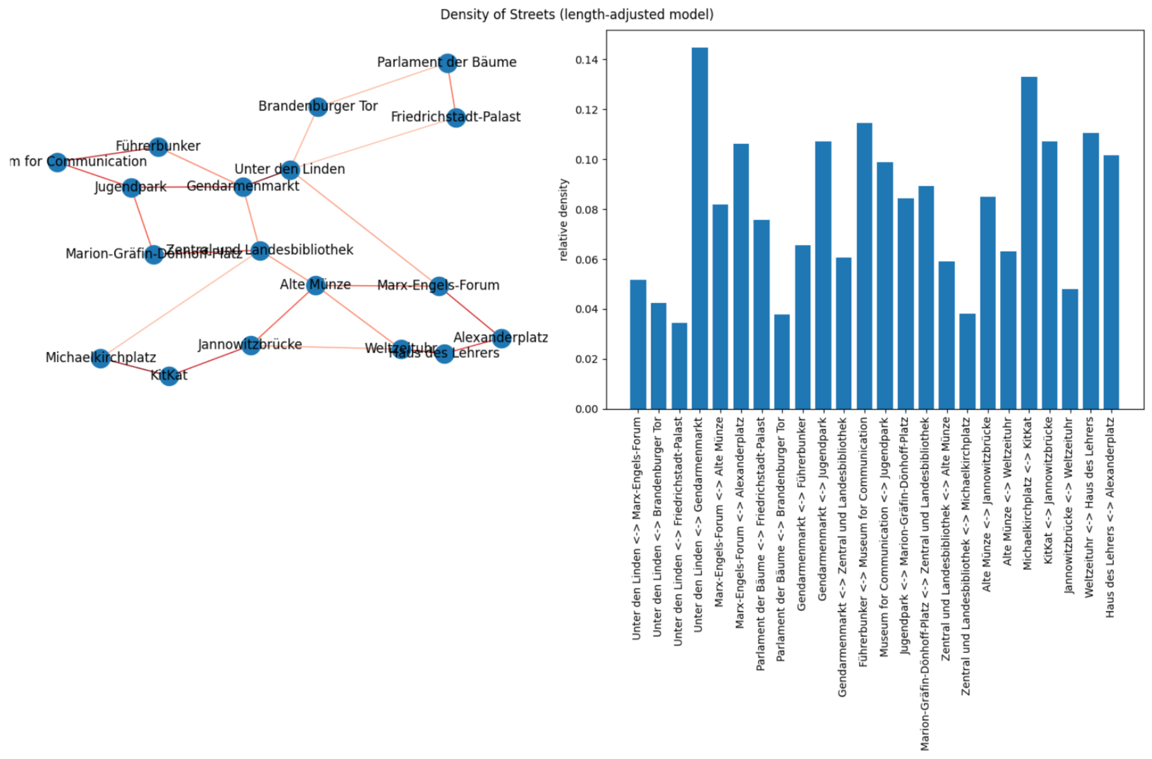
The resulting transition matrix will model the desired behavior. The intuition here is that each column *i* represents what happens to the cars on the road *i* at the end of each step. Our new matrix assigns a proportion *r_i* to remain in the same lane and distributes the rest evenly among adjacent lanes (including this one.)

We can arrive at the steady states of this system by finding all the positive eigenvectors of these transition matrices. Each positive eigenvector will represent a steady-state of the system. By normalizing the eigenvectors, we can see how the traffic is distributed among the streets---what proportion of the total traffic is currently on each street. To get steady state predictions for density, we can simply divide by the length of the corresponding street (1 in the unadjusted model, a length in the unadjusted model). The resulting values can be interpreted as the portion of total traffic occupying each unit of length. The units here aren't that easy to interpret intuitively, but they will be proportional to any other measure of density. So, later, when we get density numbers for each street in the simulation, we can normalize these and normalize those and compare the shapes to assess the quality of the prediction.

For the Berlin traffic network, we arrive at the following results. The first row of graphs represent the numbers for the unadjusted model; the second represent the length-adjusted numbers. For each, the map show a heatmap of the network where edge color represents the density of traffic at the steady state. The bar charts show relative densities of traffic along each street at the steady state.

Figure 3. *Theoretical predictions*





These results give us approximations which should estimate what we see in the simulations. If the simulation results map cleanly onto the theoretical predictions, then we can be more confident that the code is correct. If they differ significantly, then it becomes a task to explain this difference in terms of the simplifying assumptions of the theoretical analysis.

Parameter Tuning

The model as I have described has a number of parameters. To accurately model Berlin traffic, it is necessary to tune these parameters so that they are physically realistic and the behavior of the model reflects the real world traffic data from Google maps.

The main parameters of the model are:

- the probability of car's randomly slowing
- the speed limit
- the probability of cars joining at each on ramp
- the number of parking lots attached to each outgoing intersection

The probability of slowing is hard to choose because it is not clear how to measure this or get a proxy using real data. So, I followed the original Nagel & Schreckenberg (1992) and chose a value of 0.1. In the paper, this value was shown to reproduce some nontrivial qualitative features of real traffic flows with a speed limit of 5. So, it will serve for this model.

For simplicity of implementation, I decided to use one speed limit for the entire model. I scaled the model such that each cell represents 2 meters (the approximate length of a car). Speed limits in Berlin are typically around 30 kilometers/hours. Each step in the model represents 5 seconds.

30 kilometers/hour = 8.33 meters/second = ~4.16 cells/second

So, to represent a speed limit of 30 km/hour, the speed limit would need to be ~4.16 cells/second. Since the speed limit must be integer-valued, I decided to use a speed-limit of 4.

A speed limit of four is convenient for other reasons as well. Google maps gives traffic speed data on a scale with four values (green, yellow, orange, red). For the purposes of this model, I assumed that these roughly correspond to speeds of 1 cell/second, 2 cells/second, etc. In SI units, this gives.

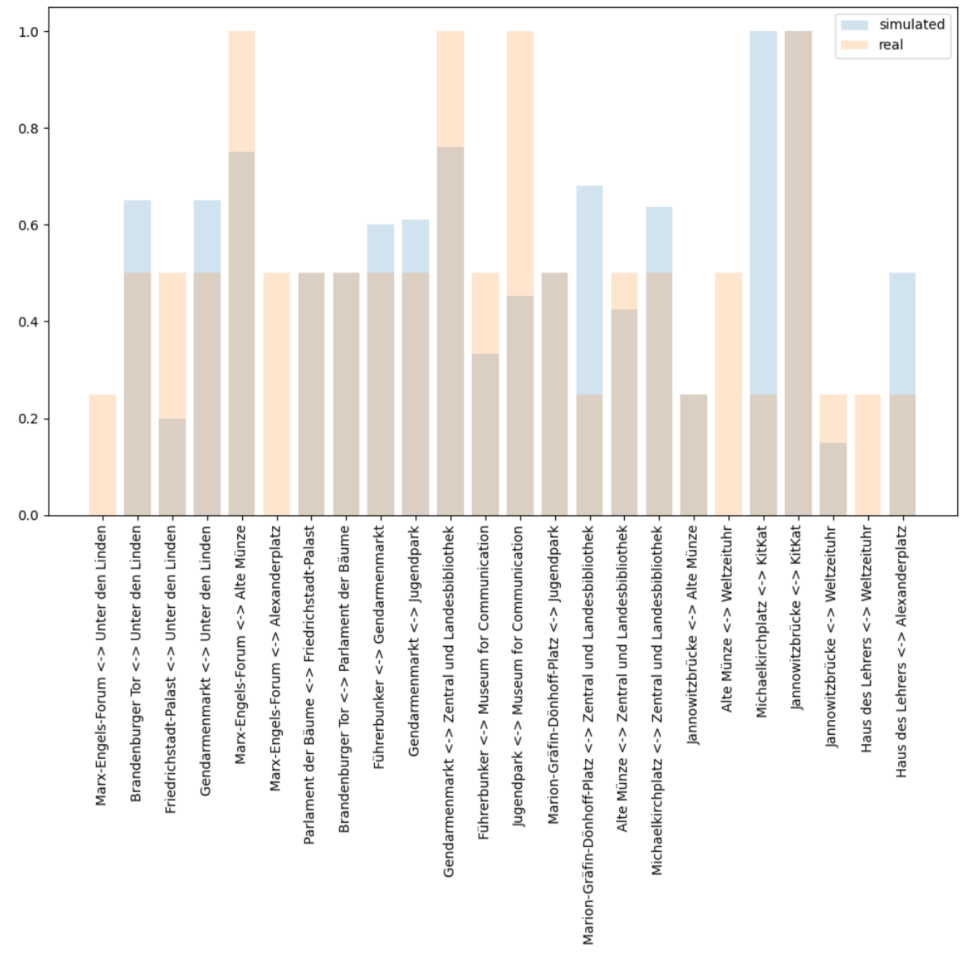
- green: 28.8 km/hour
- yellow: 21.6 km/hour
- orange: 14.4 km/hour
- red: 7.2 km/hour

These roughly correspond to my intuitive sense of how fast traffic travels on areas which google maps as green, yellow, orange, and red, respectively. So, I decided this conversion was reasonable.

This gave me a way to pick entry and exit parameters for the model. I had to set the entry and exit parameters such that the average speeds I saw in the model matched the empirical data from Google maps. So, for example, if Google maps said that the traffic on a stretch of road was red, then my goal was to get the average speed on that road as close as possible to 1 cell/second. To perform this fitting, I used a genetic algorithm to minimize the mean-squared error between the simulated and empirical speed values.

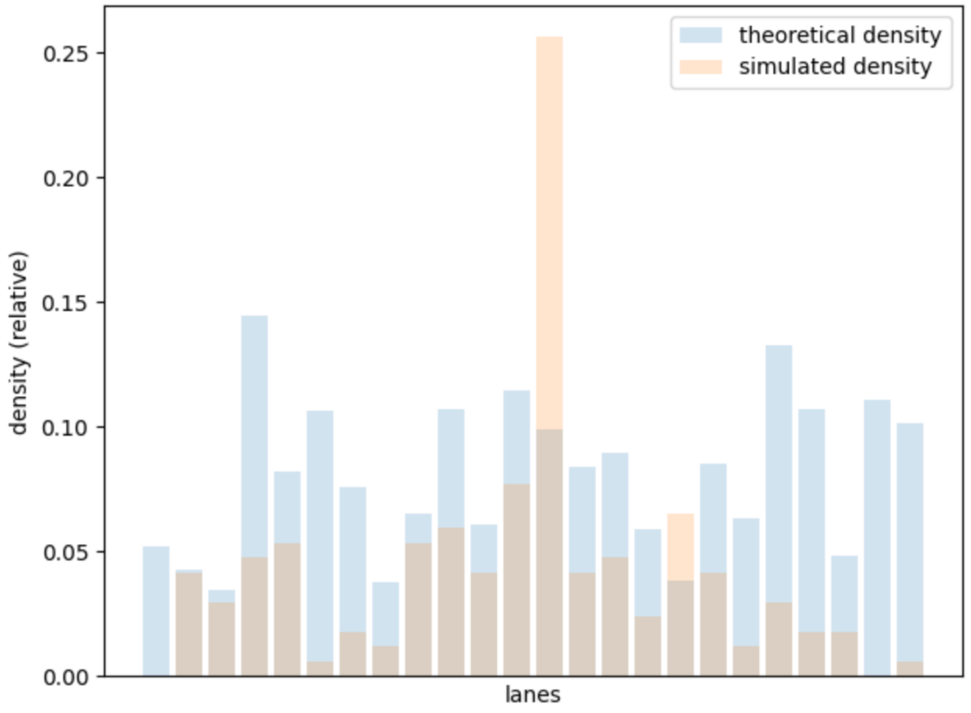
These visualizations show how well this program fared. Fig 4 shows how the Google maps empirical speeds (orange) compare to equilibrium average values derived from a single simulation run (blue). Both are in simulation units of cells/step. The fit is decent but not great. It is a challenging task for the optimizer to find a good solution in such a challenging optimization problem.

Figure 4. *Simulated vs empirical average speeds after parameter optimization*



As a sanity check, we can compare the behavior of the model using these parameters to the results of the theoretical analysis. Fig 5 compares the theoretical densities derived above to equilibrium simulated densities using the optimized parameters. In this plot, intersection names are dropped for readability. Again, the fit is not great. We have much less confidence in the theoretical values than in the empirical data from Google maps. Still, this does not reflect well on the output from the optimizer.

Figure 5. *Theoretical vs simulated relative car densities using optimized parameter values*



Simulation Analysis

Now that we have a simulation of the behavior of the system reasonably well, we can experiment with different interventions in the simulation to see how this affects traffic. The goal of this simulation is to determine what kind of traffic light leads to the fastest average traffic speed. Traffic speed is important because it determines how long it takes users of the system to get to their destinations, and so it will likely determine users' satisfaction with the system. I define the street speed as the mean of the average speed of cars on each lane.

I ran the model with each kind of intersection (CloverLeaf, PeriodicAdaptiveTrafficLight, PeriodicAlternatingTrafficLight, PeriodicRandomTrafficLight) for 30 runs each. On each run, the simulation ran for 500 steps, or about 40 minutes of simulation time. These numbers were chosen because they were sufficient to ensure that the confidence intervals had converged, and visual inspection suggested that the model had reached equilibrium. At the end of each run, the average speed of cars on each street was calculated. The results are shown in the next few figures.

Fig. 6 shows the equilibrium average speed of cars on each street across all simulations along with 95% nonparametric confidence intervals for this quantity.[1] The x-axis labels, which would be street names, are omitted for readability. The yellow dashed lines mark the average of the blue lines, making the average (across streets) average (across sims) average (across lanes) average (across cars) speed. I call this quantity Y, and it measures how fast the average street is moving.[2] We can see that the CloverLeaf policy does best, which is unsurprising as it is here to give a physically unrealistic upper bound on the possible performance of intersection policies. Next comes the PeriodicAdaptiveTrafficLight, at Y just below 2. The PeriodicAlternatingTrafficLight and PeriodicRandomTrafficLight are comparable and worse, at Y just above 1.

Figure 6. *Performance of intersection policies at each street*

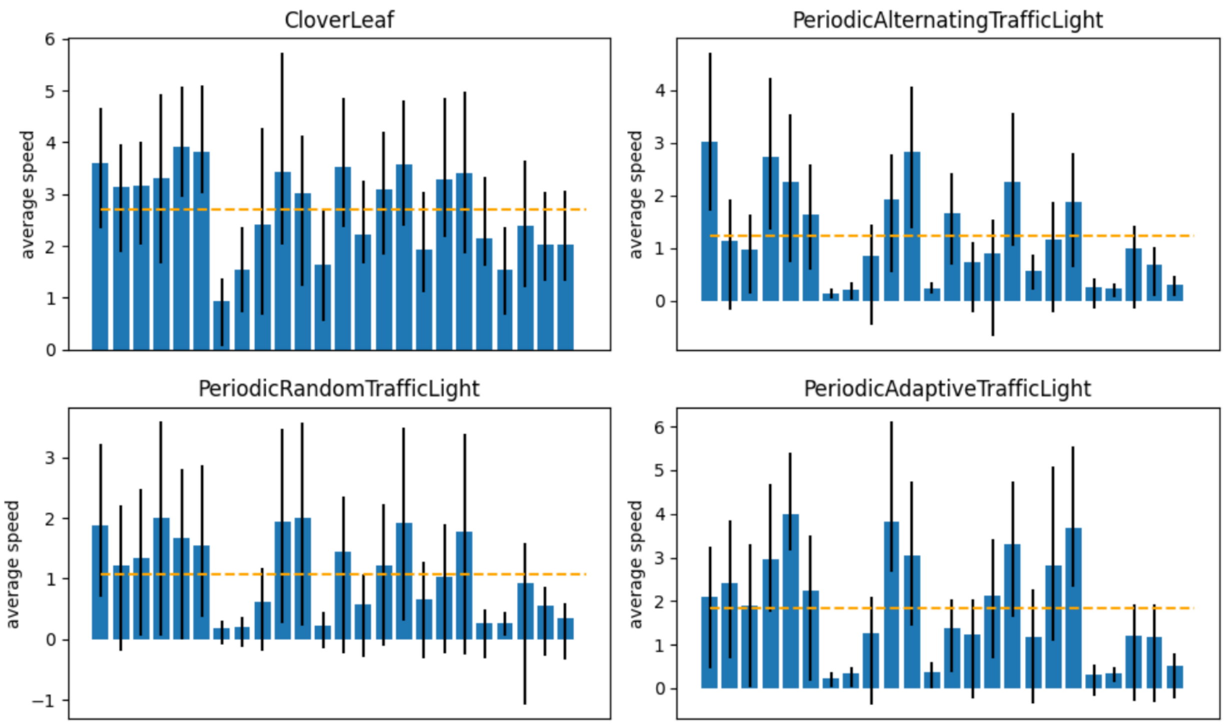


Figure 7. *Average street speed across intersection policies*

[1] The confidence interval for average speed on each street was calculated by measuring the equilibrium average speed on that street across all simulations, then calculating the 2.5% and 97.5% percentiles of this value across all runs.
[2] The simple interpretation of this quantity is "how well it did". The formal interpretation is that this is the average across streets of a quantity X. X is the mean of the average speed of cars on this street across all simulations. So, if we were to draw another street N from the distribution of streets and calculate the average speed of cars along N, the yellow line is our best guess at what this quantity would be.

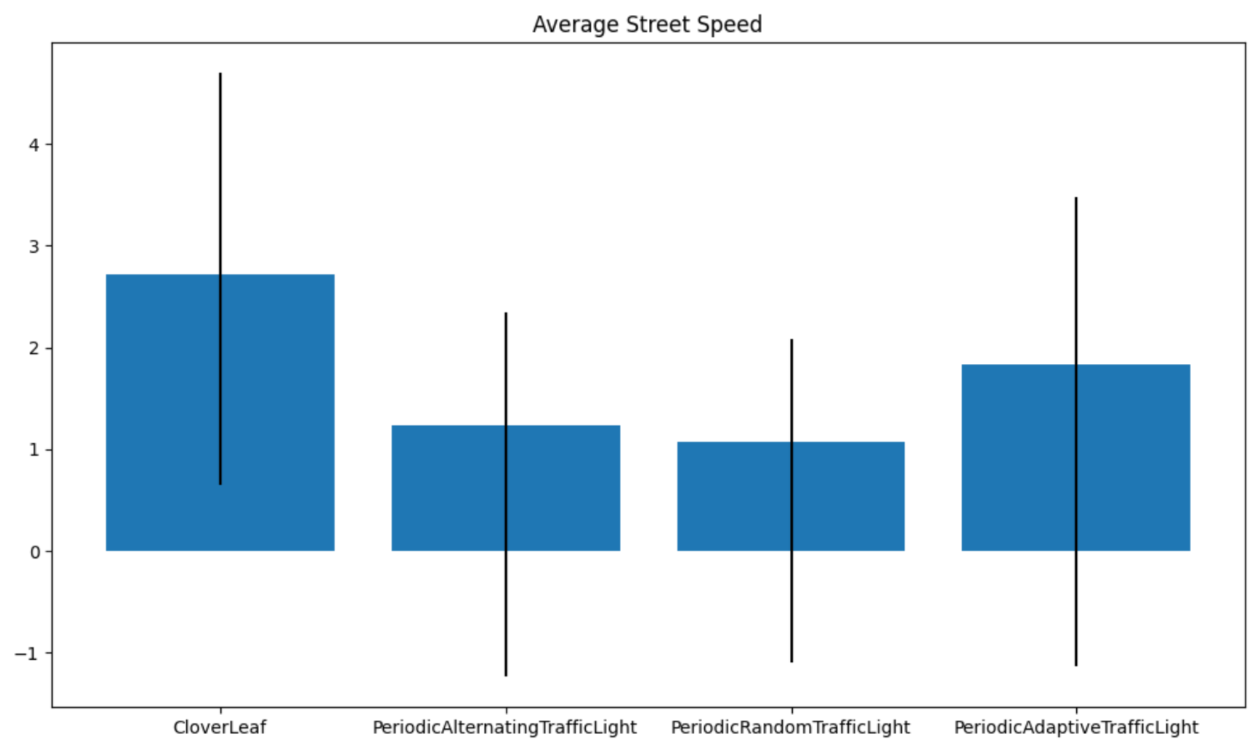


Fig 7 shows the equilibrium average street speed (across simulation and streets) of each intersection policy, along with a 95% nonparametric confidence interval. Although the CloverLeaf and PeriodicAdaptiveTrafficLight seem to be doing better, the confidence intervals are extremely wide, and so it is hard to say for sure.

The most interesting and important takeaway though is how well the PeriodicAdaptiveTrafficLight did compared to the CloverLeaf. Although the CloverLeaf represents a totally unrealistic upper-bound on the performance of an intersection, the adaptive traffic light did almost as well at keeping traffic moving. In fact, in both graphs it appears that the difference in performance between the adaptive traffic light and the clover leaf is less than the difference between the adaptive traffic light and the other light-based intersection policies.

Technical Appendix: Implementation and Testing

The simulation was implemented as a discrete-time step-based simulation in Python. A central Network class was responsible for assembling and running the simulation. An AbstractAsphalt class defines an interface for every class that directly manages cars. An AbstractIntersection class defines an interface for all intersections. To improve performance, almost all of the logic was implemented in numpy, including the lanes and most of the intersection logic.

Functions in src/dataviz.py generate the graphs. Each of the files in the analysis folder generates a subset of the graphs from this paper. You can run them to reproduce the results.

To ensure correctness, I implemented unit tests using the pytest library. There are a lot of these, and they test all the major behaviors of each component. You can see the tests and the original repository on github.

A project like this is not naturally expressed as a jupyter notebook. That said, if you're 100% sure that's really what you want, I copy-pasted all the files into one so you can read them in sequence like that. It's available here.

I used a number of Python libraries, most importantly:

- numpy
- networkx
- matplotlib
- pygad
- pytest

HC/LO Appendix

#organization - I present a clear line of reasoning through four main sections: model design, theoretical analysis, parameter fitting, simulation analysis. Each section is cohesive and builds on the last.

#modeling - I build a complex and sophisticated model of traffic in Berlin. I rely on past work in traffic modeling and on empirical data to design my model and choose parameters.

#audience - This report is targeted at a semi-technical audience of urban planners and decision makers. I balance two concerns: making the report understandable to non technical readers and providing enough technical detail to make my argument credible to field experts. I do this by presenting each quantity that I measure as both an intuitive and a technical concept.

#dataviz - I present my data in a series of professional and easy to understand graphs. I use a minimum number of graphs, each conveying an important and specific point. I explain each graph in detail.

#composition - I communicate with a clear and precise style.

#descriptivestats - For each statistical quantity that I measure, I present a clear explanation of what it is and what it is measuring. I provide a surface level explanation in the text so less technical readers can understand my conclusions. However, I am careful to explain exactly what each quantity is in footnotes.

#confidenceintervals - I create a series of nonparametric confidence intervals and interpret them correctly. The width of the confidence intervals informs the confidence of my voice.

#cs166-Modeling - I draw on a number of techniques from class including markov chains, network theory, and cellular automata. I use these techniques to inform each other, building a network of grid-based simulations and then analyzing the composite using markov chains.

#cs166-PythonImplementation - I write clean, object-oriented code. I used numpy to achieve very fast performance by vectorizing every operation that affected more than one car at once. I used networkx to enable cleaner code and for visualization.

#cs166-CodeRedability - I provide extensive comments to my code and use good variable names. I keep lines short, and I explain “magic numbers”. The code is well organized and object-oriented.