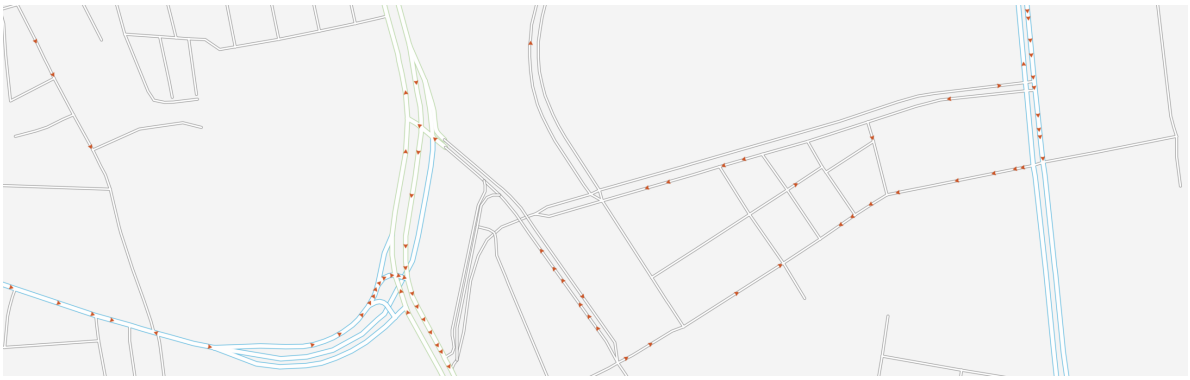
Microscopic Traffic Simulation

Maximilian Luz[†], Dominic Parga Cacheiro[‡], and Jan-Oliver Schmidt[*]



Abstract— Mobility plays a huge role in modern society and is the reason for the necessity of a well developed and carefully planned infrastructure. Unfortunately, the volume of traffic is irregular and its flow is influenced on certain, hardly controllable parameters (such as weather) which leads to the occurance of traffic jams. Observing reasons for the emergence of such phenomneas in reality is difficult, trying to predict bottlenecks on planned streets without a simulation almost impossible. This paper deals with an approach to a real-time microscopic traffic simulation, based on OpenStreetMap data and single-laned streets. We base this approach on the well known Nagel-Schreckenberg model which we later extend, and will focus on explaining the logic at crossroads.

Index Terms— Simulation, Microscopic, Traffic Simulation, Traffic, Nagel-Schreckenberg, OpenStreetMap, OSM

ACKNOWLEDGMENTS

We would like to thank the whole institute "Simulation Großer Systeme" for supporting our idea and our work. A very special thank is due to Fabian, who takes part in our meetings every time to talk and discuss current topics of our work.

1 INTRODUCTION

Every year, Stuttgart ranks in the top 3 of cities in Germany with the longest average waiting time during traffic [3]. Traffic planners can maximize traffic flow by regulating simulations and its parameters like speed limits, different crossroad types etc. While daily traffic can be observed, it is difficult to do this in situations like catastrophes or rare events, which is one main reason why simulations are recommended.

The mathematical modeling of traffic helps to optimize the traffic flow and offers an option to simulate different phenomena like traffic jams out of nowhere. In order to model traffic, there is the macroscopic and the microscopic approach.

The macroscopic one considers traffic like a fluid, which moves in waves through the streets, while the microscopic one considers each vehicle as smallest, individual and independant unit in a discrete road network. Contrary to the macroscopic approach, the microscopic approach is more flexible and allows simulating complex phenomena like jams out of nowhere just by defining a few rules.

This paper deals with the parsing from map data to a logical graph structure for singlelaned road network, which is used by our simulation. We explain the movement of vehicles and validate our crossing logic at street intersections, visualized and calculated with instant feedback. Furthermore, we analyze different traffic scenarios by comparing driving duration and waiting times of vehicles.

[†]*Maximilian Luz*
luzmn@studi.informatik.uni-stuttgart.de
[‡]*Dominic Parga Cacheiro*
st111127@stud.uni-stuttgart.de
[*]*Jan-Oliver Schmidt*
st103986@stud.uni-stuttgart.de

2 MODELING

In order to run a traffic simulation, a road network is required. Regarding performance and maintainability, the data structures used for roads, crossroads, vehicles etc. should be chosen accordingly.

We chose a *microscopic* approach to model the vehicle's behaviour. This delivers a few advantages like the fact that vehicles can act individually. This is used in the Nagel-Schreckenberg model (see section 5.2) to simulate dawdling drivers, which enables the occurance of traffic jams out of nowhere.

We use a graph with nodes representing crossroads and directed edges representing roads for each direction. A directed edge is a container for lanes accommodating vehicles. A node contains information about its geografic location and adjacent edges. Since a driver usually has a start and a destination, we calculate routes for vehicles. The routes can be calculated using a shortest path algorithm. When a vehicle reaches its destination, it will disappear.

Our simulation is divided into two phases:

- **pre-processing**
  Running a representative traffic simulation needs real map data, which has to be converted into our graph structure. The map data is also used for route calculations. Running the actual traffic simulation in real time warrants this kind of preparation.
- **traffic simulation**
  The actual traffic simulation consists of an offline phase and an online phase: The latter one handles moving vehicles on the prepared graph structure according to the real map data and traffic rules. We give the vehicles a route to drive, which has to be calculated in the offline phase.

The only traffic rules we support are street priorisation, priority-to-the-right (or left) and random priorisation. One may disable these

rules seperately, too. Traffic lights are not supported yet.

3 PRE-PROCESSING

The primary objective of the pre-processing stage is to extract the above discussed data from a given source and present it to the simulation- and visualization-framework. The secondary objective is to do so in a way, that the following stages are able to operate efficiently and in real-time on large data sets, such as the administrative district of Stuttgart (with over 250 000 relevant streets in OpenStreetMap).

In this section, we will be discussing OpenStreetMap as a source for this simulation, its implications on the pre-processing pipeline and finally give a detailed overview of said.

3.1 OpenStreetMap

The OpenStreetMap (OSM) project is a collaborative effort to provide free geographic data for the whole world. Founded in 2004 [5], it has long since become a viable alternative to large, commercial products like Google Maps and can not only be used for creating visual maps for multiple aspects (public transport, cycling, emergency healthcare support, etc.) but also for other applications, for instance, routing. OpenStreetMap provides a vast amount of features, including but not limited to many street properties, some of which we will discuss later. This and the fact that it is freely and easily accessible as well as editable has led to its widespread use in geographic applications.

Being fully community driven, the provided data is constantly improving in both, accuracy and completeness. While it may not be able to match the overall quality of geographic data provided by professional companies or government agencies such as the British Ordnance Survey, it is suitable for almost any use-case.

The actual data can be obtained in several formats, including XML and Protocol Buffer based versions. Most available formats follow the same structure, however due to simplicity we will focus on the more popular XML variant. The described data is separated into three different primitive types: *node*, *way* and *relation*. Nodes describe a spatial position, ways a succession of nodes, and relations a relation between any of these three primitives. In general, streets are modeled as ways with certain properties. Intersections between streets are represented as a shared node, complex intersections are often expressed using multiple smaller intersections. To store additional information such as the previously mentioned properties, every primitive may contain several key-value-pairs, the so called *tags*. Through these, OpenStreetMap provides all the necessary information for this simulation. Using the *highway*-tag, we can detect and classify streets as well as give them logical priorities for the street-graph. For the scenarios below, the included types are: *motorway*, *trunk*, *primary*, *secondary*, *tertiary* as well as their *link*-types, *unclassified*, *residential*, *living-street*, and *road*. There are several additional types, however, many of these may only be accessible to certain vehicle-types or pedestrians. Furthermore, the provided data not only contains the type of the street, but also the amount of lanes per direction (*lanes*-tag), the speed-limit (*maxspeed*-tag), the information whether a street is a one-way-street (*oneway*-tag) and several other tags currently not used by the simulation such as tags for traffic-lights and signs. There are also several tags purely for rendering, such as the *bridge*- or *tunnel*-tags. However, not every property of a street can be expressed using tags. For example, turn-restrictions are modeled using relations which are themselves categorized and expanded by tags.

The dynamic tag-based approach allows for flexibility and a simple parser base implementation, but it also has several drawbacks. While most of the tags are described clearly in the OpenStreetMap wiki, some differences and inaccuracies may arise in the actual data set, as it is community-based and every user has their own idea of how things should be tagged. This problem gets worse when codependent tags like the number of lanes per direction and the one-way information are inconsistent, although such things do not occur often.

As seen above, OpenStreetMap is able to supply enough detail for traffic simulation. Although it may not be able to provide the necessary quality for highly professional applications, its availability and the large amount of existing tools (including editors and format converters) make it a viable alternative to commercial products, and a reasonable choice for this project.

3.2 Pre-Processing Pipeline

The pre-processing pipeline consists of several stages including parsing, sanitization, pre-processing for runtime-efficiency improvements and the actual generation of the desired data structures. To not extend the scope of this paper, we will only cover some more important aspects in depth and outline others.

3.2.1 Intermediate Representation of Extracted Data

To keep the pre-processing pipeline flexible and extendable, we represent the nodes and ways from OpenStreetMap using a modified entity-component system. Entity-component systems are a dynamic architectural pattern based on composition. Due to its flexibility, it is often used in game-development. Objects are modeled as entities whose behavior and data is defined through their attached components. Normally, entities do not contain any information itself other than a unique identifier and, depending on the implementation, a list of their components. Since we only have two general entity types, nodes and ways (relations are handled differently due to their unique types), we also use them to store the basic information of the primitives they represent. Components are used to store groups of tags describing the primitive concerning a common aspect, e.g. the simulation parameters (*highway*, *lanes*, etc.). This allows to almost directly model OpenStreetMaps data representation and gives the flexibility to easily extend the pre-processing pipeline, for example to increase diversity of the visualization by simply adding a new component-type and its parser module.

3.2.2 Parsing and Feature Sets

Parsing the OpenStreetMap data is the first step of the pipeline. As mentioned earlier, implementing a base-parser for the XML-format is quite simple and the required data can be filtered out easily using predicates on the primitives and their tags. However, ways and relations depend on other primitives such as nodes, and in general OpenStreetMap files contain much additional information we do not care about in our simulation. To include dependent primitives without loading all nodes, ways and relations into memory, the parser runs multiple passes over the input file.

Additionally to extracting the intermediate format, the parser categorizes the created entities by assigning them to feature sets, again using predicates based on tags. These feature sets describe how the contained entities are used later in the simulation and/or visualization. For example, one feature set may contain all relevant streets used in the simulation, while a second one contains all streets of major-, and a third all streets of minor type. In this example the second and third feature set are used purely for visualization. Note that a single entity may be assigned to multiple feature sets. Based on the assigned feature sets, the pre-processing pipeline is later able to determine which data structures should be generated and what they contain. In context of visualization, this approach also allows the creation of layers based on feature sets (see figure 1).

3.2.3 Sanitization

Except for filtering, categorizing and evaluating strong implications (e.g. $oneway = ungiven \wedge oneway \neq motorway \rightarrow oneway = no$), the parser directly extracts the given OpenStreetMap data to the intermediate representation. As described previously, this data may contain inconsistencies; the sanitization step aims to eliminate these. Furthermore, some required properties may be unspecified or contain values unsupported by the simulation (e.g. *maxspeed*=*signals* – the simulation cannot handle speed-limits imposed by dynamic signals such as electronic signs) and thus have to be set correctly.

An issue affecting the simulation, which cannot be handled in this step (without introducing additional complexity), is double geometry, respectively double streets. We approach this problem in the street-processing step.

3.2.4 Pre-Processing Streets

In OpenStreetMap, streets are modeled by a succession of nodes, where a node shared between two or more streets represents a crossing. A junction-node may not only be the first or last, but any node of this sequence, furthermore a street may intersect itself. A naïve approach to generate a graph representation of this street-network would be to simply model each connection between two nodes as an edge. This, however, has two flaws. Both of them are based on the fact that, using the naïve approach, a street with a complex course containing numerous nodes will be divided into many graph-nodes and edges. The first flaw is, that each edge of the final graph is divided into a discrete number of cells, depending on the length of the represented street segment. Creating many small edges instead of fewer but larger ones increases the error introduced by rounding the segment length, even more so considering that an edge must contain at least one cell. The second and more important flaw is, that each graph-node is interpreted as a crossroad. This means that using the naïve approach would introduce a large amount of simple two-street-connections (end to end) that are unnecessarily interpreted as crossings, and thus have a major negative influence on the runtime-performance.

One solution to this problem is – in theory and for an undirected graph – simple: Generate a minimal graph by removing nodes with only two adjacent edges and merging these edges to keep the connectivity. In practice, we do not work on a graph to begin with, but the OpenStreetMap street-network as described above. Additionally, the resulting graph is directed and the edges may have different properties based on which they sometimes cannot be merged. Despite said complications this solution is still feasible and it can be divided into three major steps: *apply restrictions*, *split* and *merge*, a general outline is given as Algorithm 1.

Apply Restrictions Before we can bring the street-network into a graph-friendly representation (using *split* and *merge*), we need to look at turn-restrictions. In OpenStreetMap, these restrictions are modeled using relations between streets and nodes, and relations reference their participants using unique IDs. The proposed steps, however, will destroy the binding between street and ID, thus relations concerning these streets have to be applied before. To handle restriction relations, we introduce *street-connectors*. These connectors are triplets $(\text{from}, \text{via}, \text{to})$ and indicate an unrestricted connection from one street via a node to another street. For each street we now store three sets of connectors: *incoming*, *outgoing* and *u-turns*. Changes to a street can easily be propagated onto these sets, and during construction of the street-graph the required *lane-connectors* can be derived from these *street-connectors*.

Split The *split*-step splits each street on every inner node, shared between at least two streets and therefore the resulting street-network contains crossroads only on start or end nodes of street-segments. Such a segment could now be directly mapped to one edge per street-direction of a valid street-graph.

Merge The *merge*-step ensures that the generated graph is minimal by – if possible – merging streets on crossroads with only two participants. Note that due to incompatible *street-connectors* or different properties, not every crossing between exactly two streets can be removed. After this step, the street network can be easily converted to a minimal directed graph (see Algorithm 1).

Removing Double Streets Detecting and removing double street segments can best be done between the *split*- and *merge*-step. Between these two steps, double street segments will have exactly two nodes (double segments intersect on every node), making comparison simple.

Algorithm 1: Generating a minimal street-graph.

Input: street-network as *nodes*, *streets*, *restrictions*
Output: directed graph $G = (V, E)$

```
/* apply restriction relations                          */
foreach restriction in restrictions do
    apply restriction to streets;

/* store streets adjacent to nodes                      */
∀n ∈ nodes : adjacent[n] ↦ ∅_m;
foreach street in streets do
    foreach node in street do
        adjacent[node] ← adjacent[node] ∪ {street};

/* split streets                                        */
foreach street in streets do
    foreach inner node in street do
        if |adjacent[node]| ≥ 2 then
            split street on node;
            propagate changes to streets and adjacent;

/* remove double streets                                */
foreach node in nodes do
    rem ← {};
    foreach street in adjacent[node] do
        if street ∉ rem then
            rem ← rem ∪ doubles of street;
    remove all streets in rem from streets and adjacent;

/* merge streets                                        */
foreach node in nodes do
    if |adjacent[node]| = 2 then
        if the two streets in adjacent[node] are mergeable
        then
            merge both streets in adjacent[node] on node;
            propagate changes to streets and adjacent;

/* generate graph                                       */
V ← {};   E ← {};
foreach node in nodes do
    if |adjacent[node]| ≥ 2 then
        V ← V ∪ node;
foreach street in streets do
    if forwards is a valid direction of street then
        E ← E ∪ (start(street), end(street));
    if backwards is a valid direction of street then
        E ← E ∪ (end(street), start(street));
```

Notation: $\{\}$ is used to denote an empty set, $\emptyset_m$ is used to denote an empty multiset.
For simplicity, this algorithm assumes that modifying a collection during iteration has a well defined semantic: Any changes made to the collection during iteration are executed after the iteration has finished.

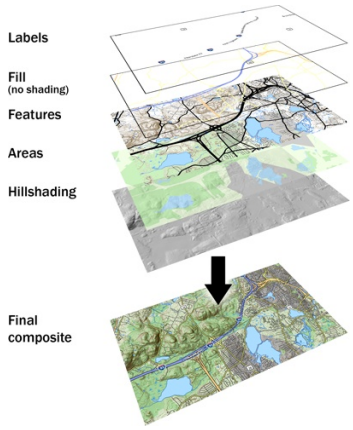


Figure 1: Illustration of the layer-based visualization approach. (https://wiki.openstreetmap.org/wiki/File:Layer-stack.jpg)

4 Visualization

For real-time simulations such as the one discussed in this paper, displaying live data is a key feature. While many observations can be summarized during execution and visualized off-line, recording and storing details often requires a large amount of memory. This is, in case of easily repeatable, deterministic[1] real-time simulations, not necessary and makes live visualization the preferred method. However, executing visualization and simulation in parallel puts stricter constraints (e.g. memory, performance) on the said subsystem.

We will begin this section with an overview of the implemented visualization subsystem, followed by a short introduction to map projections. Lastly, we will discuss two more technical aspects: Line rendering and vehicle visualization in context of OpenGL.

4.1 Overview of the Subsystem

Classic online map visualizations, such as Google Maps or the OpenStreetMap web interface, use pre-rendered images, so called image-tiles. Tiles are the result of subdividing the map to be drawn at fixed zoom-levels into equally sized and axis aligned rectangular slices (usually creating a quad-tree with one tile per tree-node). The benefit of using pre-rendered tiles is a runtime-efficient visualization, however due to the considerable amount of memory[2] required for storing these tiles, this approach is not feasible in our case. An alternative would be to utilize the OpenStreetMap tile-server, but this would limit the visualization to a predefined style and worse, it would disallow the use of custom generated or modified map files.

For our implementation we decided to use the Open Graphics Library (OpenGL) version 3.3 and a strictly two-dimensional, tile-based approach, compositing the tiles on the fly as required. OpenGL allows for fast, hardware-based and platform independent rendering of basic primitives including triangles, lines and points. Similar to common map visualization methods, we generate layers by associating the map-features described in Section 3.2.2 with customizable styles and blending them together as illustrated in Figure 1. In addition, we differentiate between layers of static and dynamic data: Static data is assumed to change infrequently or not at all, thus making it possible to create, maintain, and render via a tiled data structure. Dynamic data is assumed to change every frame, therefore a tiled data structure cannot be maintained efficiently. We display dynamic data (such as the simulated vehicles) directly, atop of the static tiles.

4.2 Projections

OpenStreetMap uses the WGS 84 coordinate reference system (also known as EPSG:4326) to provide its geographic information [5]. This system is based on a spheroid, coordinates are given in latitude ($-90°$ to $90°$, south to north) and longitude ($-180°$ to $180°$, west to east). It is mainly used by the Global Positioning System (GPS) and very suitable for the task of providing accurate geospatial data, however, projections are required to transform the spherical coordinates into a two dimensional, visualization friendly, planar representation.

One of the most simplistic projections is the equidistant cylindrical (also referred to as equirectangular) projection. This projection is achieved by replacing the spheroid with a cylinder defining its height as latitude, and then unrolling it. The radius of this cylinder defines the standard parallel, i.e. the latitude at which the scale of the projection is one. Using the equator as the standard parallel ($\varphi_1 = 0$ in Equations 1) yields the Plate Carrée projection. In formulas, this projection can be written as (omitting terms of east-west adjustment and scaling)

$$\begin{aligned} x &= \lambda \cos \varphi_1 \\ y &= \varphi \end{aligned} \tag{1}$$

with λ as longitude, φ as latitude, and φ_1 as standard parallel [6]. Besides its simplicity, an important aspect of this projection is that meridians and parallels are equidistant straight lines, intersecting at right angles [6]. A major problem with this projection is the distortion: The scale in x-direction is dependent on the latitude, while the scale on the y-axis is constant.

The Mercator projection fixes said scale problems while keeping the equidistant and orthogonal properties of meridians and parallels: The scale changes uniformly in x- and y-direction, depending on the latitude. Different variations of this projection (e.g. Web-Mercator) are often used for map visualization such as OpenStreetMap and Google Maps. In formulas, the Mercator projection can be written as

$$\begin{aligned} x &= \lambda \\ y &= \ln \tan \left(\tfrac{\pi}{4} + \tfrac{\varphi}{2} \right) \end{aligned} \tag{2}$$

(again omitting terms of east-west adjustment and scaling) [6]. By limiting y to $[-\pi, \pi]$ (and thus φ to approx $[-85°, 85°]$) the resulting rectangular plane can be turned into a square making it ideal for tile based rendering.

4.3 Technical Aspects

In the following paragraphs we will briefly discuss a few technical aspects, mainly addressing performance. Please note that this is not a computer graphics paper, thus implementation details will be omitted.

4.3.1 Drawing Lines in OpenGL

The line rendering capabilities of OpenGL are, to say the least, limited. OpenGL does neither support joins nor caps, and the available values for the line width (other than one pixel) are implementation dependent. Lines may be drawn anti-aliased, but when doing so, the line width may be restricted even more. However one has to keep in mind that OpenGL is designed for performance, providing a small set of functions around a flexible rendering pipeline, and is not a large abstract drawing library.

To display lines realiably, we have to transform them into a triangle mesh as shown in Figure 2. Anti-aliasing can be achieved using a signed distance vector and exploiting the parameter interpolation on fragments between vertices: The length of distance vector indicates the distance from a fragment to the center of the line, based on which the OpenGL `smoothstep` function (Hermite interpolation) can be applied to the alpha value. This generates a smooth transition of the alpha value between zero and one in a definable radius.

[1] The presented simulation uses pseudo-random number generators, thus determinism can be achieved by initializing them with known seeds.

[2] Assuming a quad-tree based tiling scheme and approximately 633 B per tile [5], the required memory would be $m \approx 633\,\mathrm{B} \cdot \sum_{i=0}^{n} 2^{2i}$. For a zoom-level difference (full zoom minus the zoom level on which the complete map-segment can be displayed in one tile) of $n = 10$, this results in $m \approx 844\,\mathrm{MiB}$, with a zoom-level difference of $n = 15$ already in $m \approx 844\,\mathrm{GiB}$ of pre-rendered imagery. A real world example, using OpenStreetMap: The district of Tokyo can fit on one tile at zoom-level 7. With a maximum zoom level of 19, it would require approximately 13.2 GiB.

Using the triangulation scheme shown in Figure 2, for correct line distance vectors up to eleven vertices per original line-vertex may need to be emitted. Using a geometry shader, we can generate these vertices on the fly, eliminating the need to store more than two vertices per line segment. However, this has a considerable impact on performance when drawing many lines and executing said shader every frame. Solutions to this problem include reducing the amount of lines through view frustum culling, level-of-detail or other methods, and using a tile-based approach, caching visible tiles in GPU memory rendering them only when newly visible or updated.

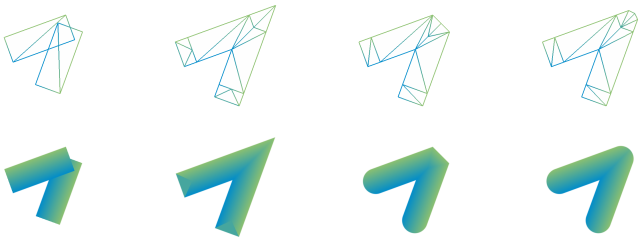


Figure 2: Lines and their triangle meshes rendered using different join- and cap-types.
Anti-aliased lines (bottom) and their triangle meshes (top) rendered using different join- and cap-types (left to right): *none/none*, *miter/butt*, *bevel/round*, *round/round*. The color-gradient indicates the y-component of the signed line distance vector: green as negative, blue as positive *linewidth*.

4.3.2 Visualizing Vehicles

In the context of this simulation, visualizing vehicles means displaying a large amount of dynamic data, changing every frame. To achieve this task with reasonable performance, we evaluated two different approaches: First, drawing the vehicles as simple triangles generated by a geometry shader, and second, drawing the vehicles as point-sprites. In both cases different vehicle types can be indicated by different base-colors, for the sprites a simple color multiplication is performed. We could not detect any notable performance differences between these two methods on various hardware.

5 Traffic Simulation

This section deals with the calculation of routes for the vehicles, the Nagel-Schreckenberg model for the vehicles' movement and our defined crossing logic for their behaviour at a node.

5.1 Routing

For the route calculation, we use two A* implementations [2], which improved the time for the preprocessing by a substantial amount compared to a simple Dijkstra [1], especially on big maps. One of the two implementations uses the shortest route, the other one uses the fastest route calculated with the streets' speed limit and length.

5.2 Nagel-Schreckenberg-model

The basic idea of the Nagel-Schreckenberg-model [4] is a cellular automaton. It defines streets with cells accommodating at most one vehicle. One cell corresponds to 7.5 m and one step to one second. On one hand, this guarantees an appropriate distance between two consecutively driving vehicles. On the other hand, it allows roughly realistic speed limits for integer vehicle speeds between 0 and 5. For example, a vehicle of speed 3 (after a step) has been moved 3 cells forward in the past step. One step consists of 4 parts:

1. **accelerate**
   Each vehicle increases its velocity by one if its velocity is smaller than its maximum velocity.

2. **brake**
   Each vehicle u checks whether it has a vehicle v in front at the same lane. If that is the case and the cell of v would get reached with the current velocity in the next simulation step, u has to reduce its velocity to stay behind the current position of v.

3. **dawdle**
   With a certain probability, each vehicle decrease its velocity by one. Using a probability is much less complex than simulating drivers' individual behaviours.

4. **move**
   Each vehicle moves forward corresponding to its velocity.

5.3 Crossing logic

Vehicle movement seperates into two parts: movement in the middle of the street and movement bordering on crossroads. The movement in the middle of the street depends solely on (our extended) Nagel-Schreckenberg-model. However, crossing vehicles requires additional logic at the nodes. The difficulty lies in reducing the complexity of different crossroad structures to simple rules, which don't depend on any individual crossroad structure. Moreover, the vehicles' brake and move operations need to get modified to prepare them for entering a crossroad. Thus, vehicles have to register at the nodes. In order to decide, which vehicle may cross the node during the following simulation step, nodes have to manage and assess all registered vehicles.

5.3.1 Braking and moving

Most of the vehicle's complexity is handled during the *brake* operation (see Figure 3). Basically, the vehicle checks whether it has to brake for any border, either at its *own* road or at the *next* road (if it would cross the node during this step, considering its current velocity). A border is a vehicle in front or the end of the road. Since the *brake* operation calculates the velocity considering every potential conflict, all vehicles can *move* depending merely on their velocity.

5.3.2 Vehicle management at crossroads

The nodes handle all conflicts between vehicles with the intention to traverse the same crossroad. This includes managing and assessing incoming vehicles to decide, which vehicle is allowed to cross.

The process of assessing vehicles at a node consists of two aspects:

- **registration and deregistration**
  Upon reaching a certain proximity to a certain node, the vanward vehicle of its current edge registers at this node, which compares the incoming vehicle with each registered vehicle. For this, each vehicle receives a *priority counter* when registrating. If the *priority counter* is increased, the associated vehicle "wins", if it is decreased, the vehicle "loses" a comparison against another vehicle. The decision, which vehicle is allowed to cross, depends on these *priority counters*.

  A vehicle is allowed to cross, if it won against all other registered vehicles. This may apply to more than one vehicle, if e.g. two vehicles don't cross each others way. If there is no such vehicle, a deadlock has occured and in order to resolve it, one vehicle receives permission at random.

- **assessing two vehicles**
  Our method for comparing two vehicles at a crossroad is shown in Algorithm 2 in pseudocode. Interesting routines are `crossing(u.way, v.way)` and `priority_to_the_right(u.way, v.way)`. Both routines are based on our self-defined crossing indices.

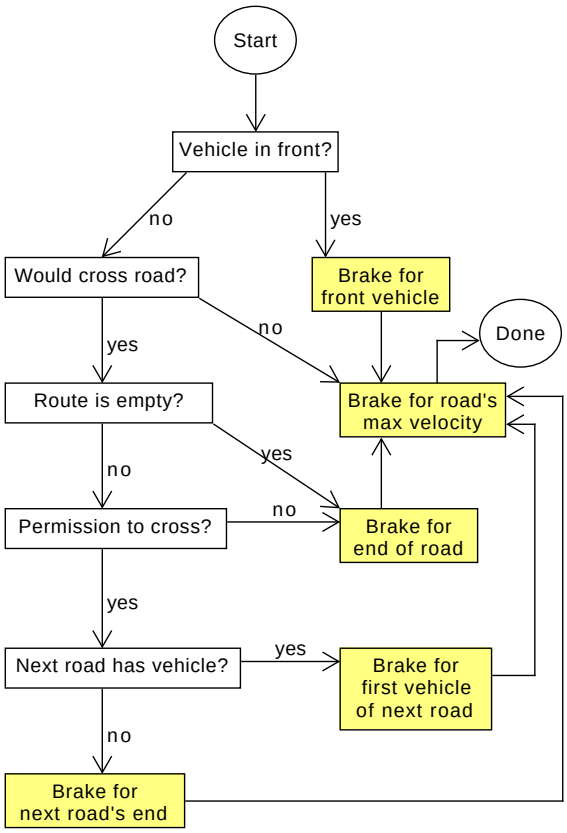


Figure 3: Flow diagram for step 2 (*brake*) in Nagel-Schreckenberg model

Algorithm 2: Assessing two vehicles at the same node

Input: Vehicles u and v
Output: Vehicle(s) with priority

```
1  if crossing(u.way, v.way) then
2      cmp ← u.way.origin.priority − v.way.origin.priority;
3      if cmp == 0 then
4          cmp ← u.way.dest.priority − v.way.dest.priority;
5          if cmp == 0 then
6              return priority_to_the_right(u.way, v.way);
7      else
8          return cmp > 0? {u} : {v};
9  else
10     return {u, v};
```

Notation: A way is an abstract crossing route from the incoming lane to the destination lane.

5.3.3 Crossing indices

During the preprocessing, each node calculates indices for its incoming and leaving edges (reminder: one street corresponds to either one or two directed edges depending on the street type: one-way or both-way):

Every edge has a 2D-vector, that describes, in which direction it enters/leaves the node. These vectors are sorted by their angle to an arbitrary chosen zero vector; counterclockwise for priority-to-the-right, respectively clockwise for priority-to-the-left. It is important to take either all incoming or all leaving edges' vectors reversed, so that all edges associated to the same road have the same angle. In the cause of this paper, we always assume priority-to-the-right. Priority-to-the-left works analogously.

Now, each edge receives a nodewide-unique integer according to the order of the sorted vectors they are corresponding to. Because each node sorts its adjacent edges like this, one edge has one index per adjacent node.

The calculated indices allow us to determine whether two vehicles' ways *u.way* and *v.way* intersect or not. At first, we create two arrays $array_u$ and $array_v$, one per way. Each array is built as described in Algorithm 3. Metaphorically speaking, this is

Algorithm 3: Creating a crossing indices-array

Input: Way w of a vehicle
Input: # of edges n
Output: Indices-array from w.origin to w.dest

```
1  i ← w.origin;
2  array ← [ ];
3  while i ≠ w.destination do
4      array ← array + [ i ];
5      i ← (i + 1) % n;
   return array
```

like walking counterclockwise along the node's edges from the origin edge to the destination edge. With such indices-arrays, we can calculate the result of `crossing(u.way, v.way)` and `priority_to_the_right(u.way, v.way)`.

The deterministic finite automaton, depicted by figure 4, describes the process of checking whether two ways are crossing. This DFA checks an indices-array starting with u_o and ending with $u_o - 1$ built analogously to the algorithm depicted in Algorithm 3. The basic idea of the DFA is: Let k be the order of the edges u_o, u_d, v_o, v_d, how you pass them, if you would go counterclockwise along the node's edges starting from u_o. Provided, the ways intersect, k would consist of an alternating order of the edges of the way of u and v. In the case the ways are not intersecting, k would consist of either $u_o u_d v_{\{o,d\}} v_{\{d,o\}}$ or $u_o v_{\{o,d\}} v_{\{d,o\}} u_d$.

After illustrating `crossing(u.way, v.way)`, only `priority_to_the_right(u.way, v.way)` demands further explanation.

Our algorithm processes two indices-arrays describing an intersection of two ways. If they didn't intersect, it wouldn't be necessary to detect the prioritized vehicle. An intersection results in the two indices-arrays having precisely one common pattern. The reason for this is based on the construction of the indices-arrays:

- There are no dublicate indices in one indices-array.
- An indices-array's structure is independantly defined from its origin and destination.
- An indices-array's length is limited.

Let u.way be represented by $u_0 u_1 \dots u_n$ and v.way by $v_0 v_1 \dots v_m$. Since only one vehicle per edge is able to register at the node, the indices-arrays can't share their first index. Assuming the order of

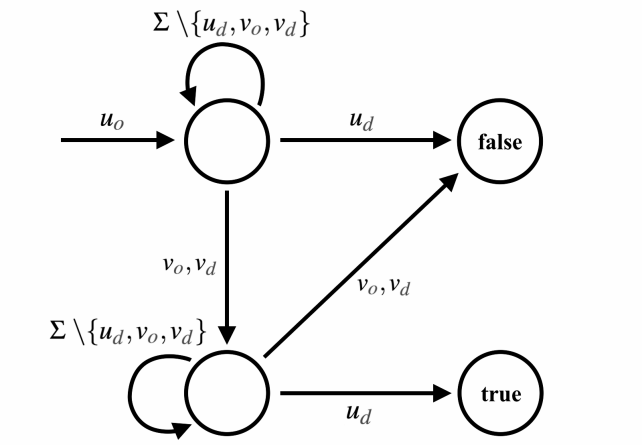


Figure 4: DFA to check, whether two ways are intersecting. $\Sigma = \{0, 1, \ldots, \#\text{edges} - 1\}$ and u_d represents the index of the *destination* edge of the vehicle u. The meaning of u_o, v_o, v_d is analogous to u_d. This DFA returns whether the ways are crossing.

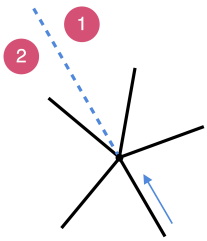


Figure 5: We are assuming, that the non-prioritized vehicle is coming from the street in the bottom right. The numbers represents the cases, where the prioritized vehicle comes from: right or left of the blue line.

the indices u_0, u_n, v_0, v_m is $u_0 v_0 u_n v_m$, the common pattern would be $v_0 \ldots u_n$. Because v_0 is the first index, vehicle v has priority over u. Metaphorically speaking, if u went counterclockwise around the node, u would reach the origin of v before it would reach its own destination edge. For the purpose of validation, there are two relevant cases to differentiate, as you can see in figure 5. In the first case, v is coming from the right. Thus, u has to give way. In the second case, v is coming from the left, but since u is crossing v's way by turning left, u has to give way.

6 Scenarios and Analysis

Whilst obtaining real traffic data is rather costly and the fact, that only assigning arbitrary routes to cars will not result in a credible traffic simulation. Therefore, we choose to create a scenario. But first of all, we elaborate on the aspect, that our simulation runs in realtime and introduce some small scenarios to validate our crossing logic.

6.1 Realtime

People perceive and analyse data visually very well. If you wanted to change the parameter setting in order to see its repercussions, it would be advantageous to get responses in realtime. For us, realtime means the simulation runs with at least 25 frames per second. To achieve this, we implemented the substeps of the Nagel-Schreckenberg-model multithreaded: each thread gets a certain number of vehicles to process their current step. Compared with singlethreaded, we could quarter the time per whole step using a 4-core cpu running 10.000 cars on Stuttgart's metropolian area.

6.2 Validation scenarios

After going over the theory, we want to display several scenarios, which validate the crossing logic. Our visualization does not show more than one line per street yet. Each street consists of two directed single-laned edges (one incoming and one leaving edge).

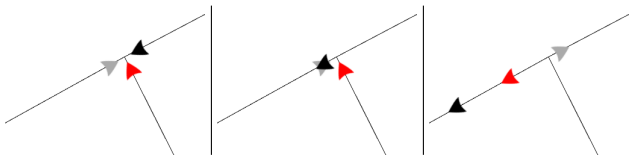


Figure 6: Three vehicles are reaching the crossroad simultaneously. The gray (left) one gives priority to the red (bottom) one. The red one gives priority to the black (right) vehicle, which is allowed to traverse the crossroad first. We assume, the index of the left incoming edge is 0.

In order to comprehend the decision, which vehicle has permission to drive, we need their crossing indices-arrays. The gray vehicle wants to drive straight forward from 0 to 3, thus its indices-array is $[0, 1, 2, 3]$. The red vehicle's indices-array is $[2, 3, 4, 5]$ and the black one's $[4, 5]$. These indices-arrays yield the following comparisons:

cmp	gray	red	black	priority counter
gray	-	-1	+1	0
red	+1	-	-1	0
black	+1	+1	-	2

Table 1: cmp stands for "compare", which is an interpreted result of a comparison of two vehicles. Vehicles decreases their priority counter by 1, if they have to give way; otherwise they increase it by 1. This table belongs to Figure 6.

The black vehicle has permission to drive, because no other vehicle has higher priority over it, which you see in the *prioritiy counter*. After black traversed the crossroad, only red and gray remain waiting at the node. Sweeping all rows and columns, that contain black in the previous table, and updating the priority counters result in red having priority over gray.

Another interesting case is a deadlock situation, as depicted in Figure 7. The table shows, that there is no vehicle whose priority counter is equal to the number of all vehicles minus one. This means no vehicle has priority over all other vehicles. The deadlock can be resolved by choosing a prioritized vehicle at random. In the picture, it is the red one.

6.3 Simulation attributes

There are several parameters, which are set in Table 3 and explained in the following.

We can set the **amount of vehicles**, that are to be added to the simulation. Each vehicle has a **probability for dawdling**, which is set to 20%.

Our **route calculation** is influenced by a probability for the vehicles to choose the shortest path algorithm either for the fastest route or the shortest route.

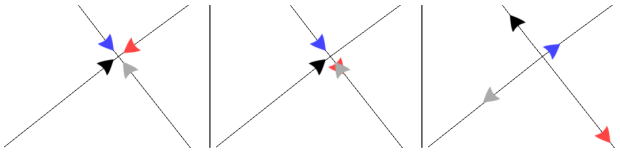


Figure 7: These four vehicles are reaching the crossroad at the same time and want to turn left. Therefore, every vehicle has to give way their right neighbour. The black vehicle is on the bottom left, the gray one on the bottom right, the red one on the top right and the blue one at the top left.

cmp	gray	red	blue	black	priority counter
gray	-	-1	+1	+1	1
red	+1	-	-1	+1	1
blue	+1	+1	-	-1	1
black	-1	+1	+1	-	1

Table 2: This table belongs to Figure 7

The boolean **Priority-to-the-right** determines, whether the crossing logic calculates priority-to-the-right or priority-to-the-left. All scenarios are running with priority-to-the-right.

We alter the crossing logic by adding booleans, which decides whether a **traffic rule** is considered. Our used traffic rules are 1) street priority and 2) either priority-to-the-right or random.

All vehicles are appended to a node, before any other vehicle starts driving its individual route.

We want to compare, how different traffic rules effect our simulation. We've run a simulation with the same parameter setting multiple times to compare the average of the resulting measured data. The Table 3 shows our settings, which we compare.

6.4 Anger

We added an anger value for each vehicle. A vehicle increases its anger by one if its velocity is 0 for more than one simulation step consecutively. If its velocity is not 0, its anger decreases by one. In addition to that, a vehicle counts another anger value without decreasing it. This allows us to evaluate how much the vehicle was forced to wait until it has disappeared.

6.5 Evacuation scenario

This scenario describes the situation, in which "all" people in and around Stuttgart want to leave it, driving away from the center. We don't support traffic lights yet, but you can imagine, that there was a power failure. The vehicles start driving distributed uniformly at random over all nodes.

scenario	1	2	3	4
#vehicles	10,000			
P(fastest route)	1.0			
street priority	true		false	
priority to the	right	random	right	random

Table 3: This table shows the parameter settings corresponding to our scenarios.

We explain and discuss two distinctive plots of data out of a simulation run. First, there is the number of vehicles, which are reaching their destination at a particular age, with age being the number of simulation steps. Second, our anger attribute allows us to conclude several interesting statements in combination with the first plot. First of all, in Figure 8, you can see, that the slowest vehicles need up to 4500 simulation steps. This equals 4500 seconds, which is over one hour. For drivers starting in the middle of the city, this sounds like quite a realistic time.

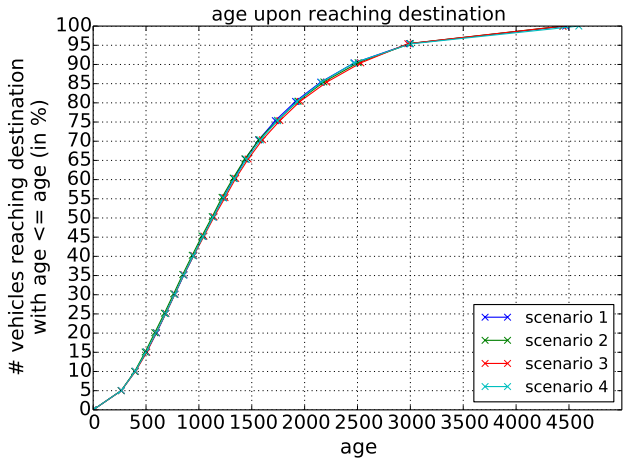


Figure 8: This plot shows the age of vehicles, when reaching their destination for **scenario 1-4**. As you can see, they are almost identical to each other.

Furthermore, the ages of all vehicles upon reaching the destination is independant from the traffic rules, because the figure looks very similar for all traffic rules.

In contrary to that, the anger plots look quite different depending on the fact, whether street priorities are enabled or not, as you can see in the figures: Figure 9a and 9b look similar, Figure 9c and 9d do so, too. The former two show, that the mood of the driver is fairly moderate in the beginning of the simulation. The main difference of Figure 9a and 9b towards Figure 9c and 9d lays in the general mood, and especially, at the end of the simulation, where only about 10 % of all drivers remain (see Figure 8). The general mood in the figures with street priority enabled is up to twice as bad as it is in the other ones, which is expressed by the maximum anger. The last 10 % are a lot more angry in the first two scenarios. In this case, "a lot" means, they have had to wait at least 1800 steps. Finishing the route with an age of 4500 steps, the last remaining drivers have had to wait around 40 % of their whole driving time.

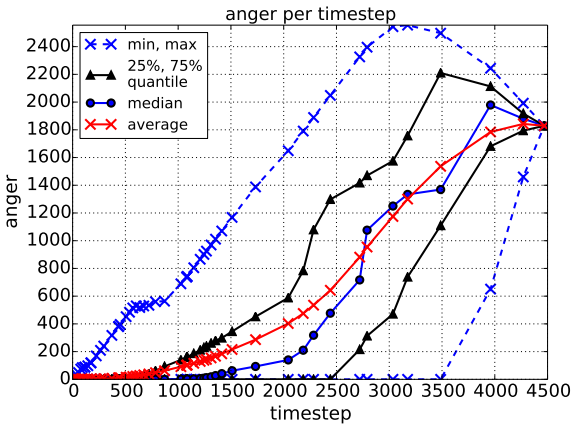
We think, that the drivers are getting more angry in the middle of the simulation, since many drivers try to enter highways/motorways, where they have to wait for vehicles already driving on the highway/motorway. It is in the middle of the simulation, because the drivers have to get to the highway/motorway firstly.

In conclusion, street priorities seem to skew the waiting time of vehicles, rather than distributing it uniformly, on the premise that the streets are singlelaned. In addition to this, regarding our results of the anger plots, we would prefer only priority-to-the-right, because all drivers are less angry.

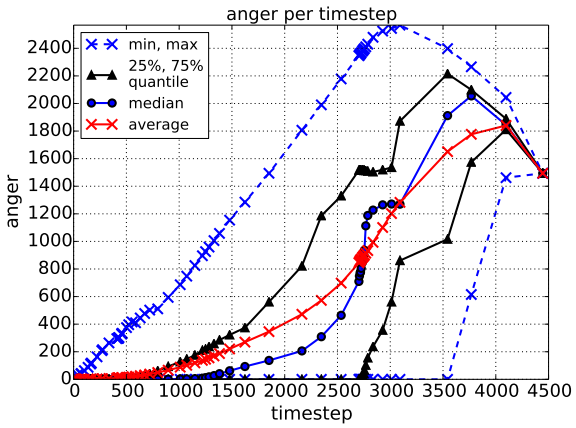
6.6 General traffic problems

While simulating different scenarios and cities, we noticed unresolved overarching deadlocks. Our crossing logic handles deadlocks at one crossroad, but not spread over many crossroads. An example is shown in Figure 10.

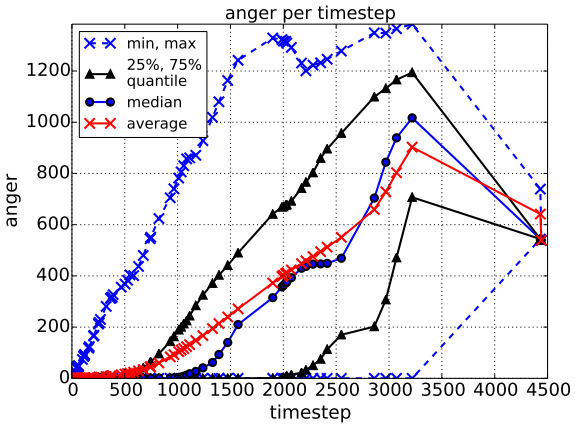
In general, this deadlocks occurs in road networks including many small cycles, like in Stuttgart around roundabouts or in New York with its typical block structure. This is worse, if e.g. two roundabouts



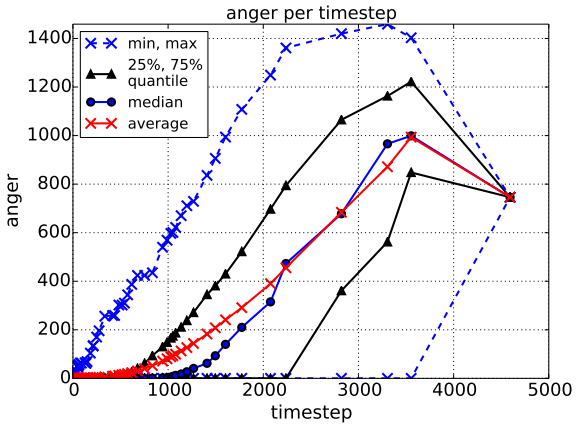
(a) This figure belongs to **scenario 1**.



(b) This figure belongs to **scenario 2**.



(c) This figure belongs to **scenario 3**.



(d) This figure belongs to **scenario 4**.

Figure 9: In these plots of **scenario 1-4**, the red line represents the average anger of all vehicles at a particular timestep, whereas the blue line is the median. Analogous to a boxplot, the dashed blue lines above and below are maximum and minimum, the black lines stand for the 25%/75% quantile.

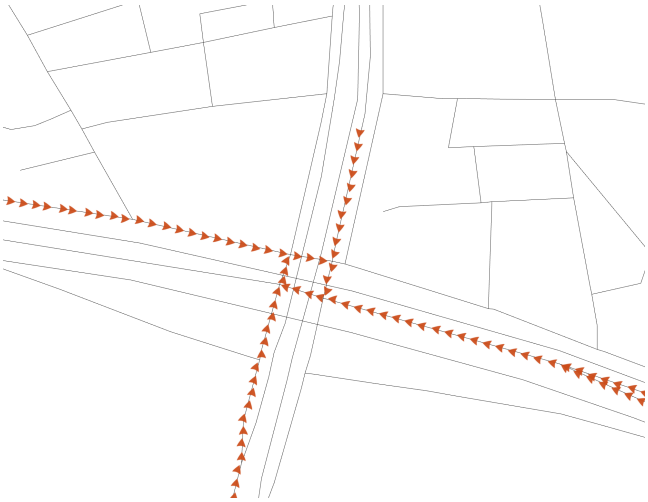


Figure 10: This deadlock is located in Tokyo, but it is representive for small cycles in cities' road networks. Especially with much traffic, this kind of traffic jam occurs frequently.

are next to each other, connected by a bidirectional street. In the case, this construct is heavily used, it can cause massive traffic jams.

7 Conclusions

We explained, which demands are put forth by our traffic model and how we could fulfill these by using OpenStreetMap data. Besides describing the visualization aspect in detail, e.g. the layer organisation or line joins, we mentioned the projection of the map data and how we extracted important information from the OSM XML files.

Our traffic model is based on the popular Nagel-Schreckenberg model, which is a microscopic approach using a cellular automaton. Our traffic rules contain priority-to-the-right (or left) and street priorities. We added an anger to the vehicles, representing a counter of how much time a vehicle has to wait during its route. This anger could show, that different traffic rules do not influence the average waiting time, but the distribution, how many vehicles have to wait a certain time. Furthermore, we detected problems of heavily used streets, which form a cycle. The smaller the cycles, the higher is the chance of detecting overarching deadlocks.

Future work is the implementation of multilaned streets and traffic lights, to obtain even more realistic simulation results. For now, our bottleneck is the route calculation in the preprocessing. We are planning the use of contraction hierarchies. These need their own preprocessing, which depends on the used map, but the route calculation can be done much faster. This would allow us to implement dynamic routing, whereas our routing is still static.

An additional future feature is the use of a graphical user interface for our simulation, which let the user control simulation parameters and measured data. We also plan a serialization of preprocessed data like map data to seperate the preprocessing from the simulation itself.

References

[1] E. W. Dijkstra. *A Note on Two Problems in Connexion with Graphs*, pages 269 – 271. 1959.

[2] P. E. Hart, N. J. Nilsson, and B. Raphael. *A Formal Basis for the Heuristic Determination of Minimum Cost Paths*, pages 100 – 107. 1968.

[3] INRIX. Jährliche INRIX Traffic Scorecard – Wichtigste Ergebnisse. `http://inrix.com/scorecard/key-findings-de/`. Accessed: 2016-04-12.

[4] K. Nagel and M. Schreckenberg. *A cellular automaton model for freeway traffic*, pages 2221 – 2229. 1992.

[5] OpenStreetMap Wiki. `https://wiki.openstreetmap.org`. Accessed: 2016-03-27.

[6] J. P. Snyder. Map Projections Used by the U.S. Geological Survey. Technical Report 1532, U.S. Geological Survey, 1982.