TECHNICAL UNIVERSITY OF MUNICH

DEPARTMENT OF INFORMATICS

Bachelor's Thesis in Informatics

Automated Conversion of Road Networks from OpenStreetMap to Lanelets

Maximilian Rieger

TECHNICAL UNIVERSITY OF MUNICH

DEPARTMENT OF INFORMATICS

Bachelor's Thesis in Informatics

Automated Conversion of Road Networks from OpenStreetMap to Lanelets

Automatische Konvertierung von Straßennetzen aus OpenStreetMap zu Lanelets

Author:	Maximilian Rieger
Supervisor:	Prof. Dr.-Ing. Matthias Althoff
Advisor:	Moritz Klischat, M.Sc.
Submission Date:	17.09.2018

I confirm that this bachelor's thesis is my own work and I have documented all sources and material used.

Ich versichere, dass ich diese Bachelor's Thesis selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

Munich, 17.09.2018 Maximilian Rieger

Abstract

Testing path planning agents for autonomous driving requires detailed representations of traffic scenarios. In this thesis, we develop and implement a process to automatically convert OSM files to a lanelet format and save it to a *CommonRoad* scenario. The resulting scenario shall contain a detailed representation of the road network described by the OSM file. The fundamental difference between these formats is that OSM maps only describe the course of roads as a whole, while lanelets specify the geometry of the drivable space for each lane on roads and also on junctions. As the desired output contains more information than the input, it needs to be estimated in several steps. The presented tool generates realistic road networks with smooth courses regardless of the low accuracy of OSM maps. It completes its task within few seconds and provides satisfactory results for the majority of available OSM maps.

Contents

1 Introduction

Currently, the field of autonomous driving is a big subject of research, both in industry and in academia. Autonomous vehicles perceive their environment by sensors to react accordingly. Maps are often used to improve the perception of agents [1], [2]. Detailed representations of traffic scenarios can also be used for testing autonomous agents. For a reliable test, a large set of test cases is necessary. While maps for navigation are publicly available for free, there is a lack of detailed road representations. Navigation maps, however, cannot directly be used for testing purposes as they commonly only provide the rough course of roads instead of precise descriptions of drivable space.

To help overcome this issue, we implement an automated converter for transforming open-source maps stored in the *OpenStreetMap* format to a *lanelet* based format. Lanelets are atomic, drivable, interconnected road segments [3]. Sets of interconnected lanelets represent road networks. This format is commonly used in academia, for example by [3]. Using an automated converter processing the huge amount of data available in the OpenStreetMap (OSM) project could provide a very large set of detailed road representations. These road representations could be used as basis for testing scenarios by adding obstacles such as other traffic participants.

The focus of this work is developing and implementing a process to automatically convert OSM files to a lanelet format and save it to a *CommonRoad* scenario. It is intended that such an OSM file contains the map data of a area not larger than few hectares (one hectare equals $100m \times 100m$). The resulting scenario shall contain a detailed representation of the road network described by the OSM file. This is a non-trivial task due to fundamental differences between both formats. Since OSM maps do only describe the course of roads as a whole, the spatial information of individual lanes has to be estimated based on the provided data. Also, the turning options at intersections are not specified explicitly and have to be estimated. To the best of our knowledge this is the first approach to convert these formats.

This thesis is structured as follows: In Chapter 2 related research projects are introduced. Also the difference of the approach of this work is stated. Chapter 3 gives a more detailed overview of the formats used by the converter. The main differences between these formats are explained in further detail. We describe our approach for an automated conversion in Chapter 4. In order to make the conversion process traceable, each step of it is explained. Chapter 5 presents the implementation of

this approach and how to use it. Then, in Chapter 6, the implemented tool is evaluated and its capabilities and limitations are being pointed out. Chapter 7 summarizes the goal, approach, and results of this work. Finally, possible future improvements and extensions of the tool are presented in Chapter 8.

2 Related Work

Having introduced the focus of this work in the previous chapter, work related to this thesis will be introduced in the following. Then the differences to this work will be pointed out.

A work very similar to this was published recently [4]. In that work, a converter between *OpenDRIVE* and lanelets is presented. OpenDRIVE is a format similar to lanelets. Both formats give detailed representations of road networks and can be used to simulate traffic [5]. However, there are multiple differences between the representations of road network in these formats. In contrast to lanelets, OpenDRIVE specifies roads by reference lines and offsets for each lane [6].

It seems reasonable to create OpenDRIVE scenarios as an intermediate step and then use this converter to achieve a conversion between OpenStreetMap and Lanelets. However, we forwent that, because OpenDRIVE provides a road network description in a similar level of detail as lanelets. Therefore, taking this intermediate step would not solve the main issues of this work. The converter introduced by [4] the scenarios in OpenDRIVE format available in lanelet formats, but the set of these scenarios is also limited.

In contrast to that, OpenStreetMap provides vastly more data. Another work tries to use OSM data for smooth path planning [7]. This work focuses mainly on generating smooth courses for roads by interpolation the given way points with Bézier curves. We used the proposed interpolation slightly modified as described in Section 4.6. Nevertheless, it is not described in this work how the smooth paths for roads can be used to represent complex road networks and individual lanes. Methods requiring a lanelet format can therefore not use the results of this work for testing.

In a masters thesis, Han Shi presents a method to convert OSM maps to the OpenDRIVE format [8]. Using this in combination with the converter from OpenDRIVE to lanelet might make this work seem redundant. However, the method proposed in [8] does not perform a detailed conversion of OSM files to an OpenDRIVE format. Instead, it only converts the course of single roads. This does not include individual lanes or intersections. Therefore, this method is also not suitable to generate road networks in a lanelet format which can be used for testing autonomous agents.

The CommonRoad format introduced in [3] is based on road networks in a lanelet format. It is used to define path planning problems including other traffic participants

and their behavior. In this work, we will provide a tool converting OSM files to lanelets and saving them to a CommonRoad scenario.

To the best of our knowledge, the provided tool is the first converter allowing to generate detailed road networks in lanelet format based on OSM data. The novelty of this approach is primarily the creation of spatial information for individual lanes. In the approaches mentioned above, this information was either already given in the input [4] or it is limited to information about whole roads instead lanes [7], [8]. The approach proposed in this work estimates possible courses for lanes and turning possibilities. The main focus and the greatest challenge is to reproduce the course of lanes on junctions. In this process several estimation methods are used to approximate missing information. For smooth courses we use interpolation with Bézier curves as described in [7]. Finally, the resulting lanelets are saved in a CommonRoad scenario as defined in [3]. We aim to provide a possibility to create a basis for large test sets for autonomous path planning agents.

3 Underlying Formats

In this work we use the data of OpenStreetMap (OSM) as a basis to generate a CommonRoad (CR) scenario. This chapter first elaborates on the OpenStreetMap format. Then it describes the CommonRoad format. In the last section, it is described why these formats are fundamentally different. By mentioning these differences also the main challenges of an automated conversion are pointed out.

3.1 OpenStreetMap

The input for the conversion performed in this work is an OpenStreetMap file. Based on the information provided by this file we want to generate the corresponding road network in CommonRoad. OSM is a non-profit community-based project that collects, stores, and provides geographic data and information under Open Database License [9].

The main structure of data in OSM is defined by three elements: nodes, ways, and relations. Nodes are geographic points, defined by their latitude and longitude, that are referenced by other elements. Ways are ordered lists of nodes and can represent linear elements, such as *roads*, and boundaries of areas. In the following, we will use the word *road* for a street or a part of a street, that leads from one point to another. Regions on which one can move from one road to another will be called *intersections*. A road consists of one or more *lanes*, on which a vehicle can drive. Lanes have a direction and either lead along a road or lead across an intersection from one road to another. Relations can add additional information to several other elements. Each element can be annotated with further information by tags. These tags are stored in a key-value system.

One important tag key is the `highway` key. It is used for roads of various types such as freeways and residential streets as well as for footpaths, cycleways, and other small paths. The type of OSM way is defined by the value of the tag.

Another tag key, that we use in this work, is the `turn` key. It is mainly used for indicating turn markings for individual lanes. In this case, the key is extended to `turn:lanes` or even further to `turn:lanes:forward` and `turn:lanes:backward` to describe the lanes of only one direction. The values belonging to these keys specify the direction a lane can lead in for turning and merging with other lanes. The values

for merging are `merge_to_left` and `merge_to_right`. For turn lanes, the values `left`, `right`, and `through` are used. `Left` and `right` can be prefixed here with `slight_` or `sharp_`. If a lane has no direction indication, the value `none` is used for it. To indicate multiple directions, values can be concatenated in any order. They are separated by a semicolon. If multiple lanes are specified by a turn lane tag, the values are concatenated in the order of the lanes from left to right separated by a vertical bar. A common tag could therefore be: `turn:lanes:forward=left|none|through;right`. This can be shortened by skipping `none` values: `turn:lanes:forward=left||through;right`.

3.2 CommonRoad

CommonRoad (Composable benchmarks for motion planning on roads) is a format which defines numerical experiments for motion planning of road vehicles [3]. It lays its focus on the possibility to reproduce experiments. The data in this format is stored in XML files, which is inspired by OSM. There are three main elements of a scenario: *lanelets*, *obstacles*, and *planning problems*. In this work, we will focus on *lanelets*, which represent a drivable segments of streets. We will not generate *obstacles*, which specify different kinds of traffic participants, or *planning problems*, which define a start and one or more goal states for the agent.

Lanelets represent one drivable lane on a segment of a street. They are defined by their left and right bound, represented by a list of way points. Several successors and predecessors can be specified for each lanelet, which allows to mark consecutive lanelets. The left and right neighbors of a lanelet to which a vehicle can change while following the current lanelet are referenced by an *adjacentLeft* and *adjacentRight* element, respectively. These four elements are referenced by a unique id for each lanelet. Also, for two adjacent lanelets the driving direction *drivingDir* is specified by either `same` or `opposite`. Additionally, there are some rules for lanelets. Firstly, the bound between two adjacent lanelets has to be equal for both of these lanelets. If the lanelets have opposite directions, the list of way points of one lanelet is equal to the reversed list of way points of the other lanelet. Secondly, the way points of each bound of a lanelet has to have the same amount of points. Furthermore, the end and start points of two consecutive lanelets have to be equal. A way point in a CommonRoad scenario is defined by Cartesian coordinates in SI units.

3.3 Differences

Having introduced these formats, we will point out the similarities and differences between them and why a conversion is not a trivial task. As the XML format definition

of CR is inspired by OSM, the XML structure of them is similar. The main similarity is that both formats represent road networks. However, they do so in very different ways, because they fulfill distinct purposes. As OSM is used for large maps of the whole world, that can be used for navigation, it only represents streets with their rough location and course. In contrast to that, CR specifies the exact geometric properties of streets, to perform motion planning on it. To convert an OSM map to a CR scenario, we need to approximate these properties, since they are not specified in OSM. The most difficult obstacle in this task is specifying the location and connection of lanelets, since they do not have an equivalent element in OSM. As an intersection is represented by a single node which is referenced by several OSM ways in an OSM map, the same intersection is represented by multiple lanelets, which are interconnected according to the turn possibilities of each lane in an CR scenario. Additionally there are smaller differences such as distinct coordinate systems that divide these formats.

Another difference is that OSM maps exist for virtually any street on this world, while the availability of CR scenarios is low. As large test sets for motion planning are essential for autonomous agents, having the possibility to perform tests on the huge dataset of road networks given by OSM, would be a step forward. Our approach to accomplish the conversion from OSM to CR is described in the next chapter.

4 Approach for the Conversion

The approach to perform the conversion from OSM Maps to the CommonRoad format is described in the following chapter. Figure 4.1 depicts the main steps executed in the whole conversion. It starts by parsing an OSM file. The geographic coordinates saved in OSM need to be projected onto a plane in that step. With the information extracted from the OSM file, we create a new graph like data structure. This structure consists of nodes, edges and lanes, where each of those elements can hold information. We complete this information by a deduction and an estimation step. Then, we interconnect all lanes saved in the graph structure. To smooth the course of the edges, we perform interpolation on it. Afterwards, we align the lanes of an edge according to its course. At intersections we generate lane segments which lead across the intersection and connect two lanes leading to it. These segments are grouped to clusters in certain scenarios. Having created the course of all lanes, we generate their bounds, which are needed for a CommonRoad scenario. Finally, we need to make some adjustments to meet the specifications of a CR scenario. With these modifications the graph structure can be exported to a CR scenario and saved as an XML file.

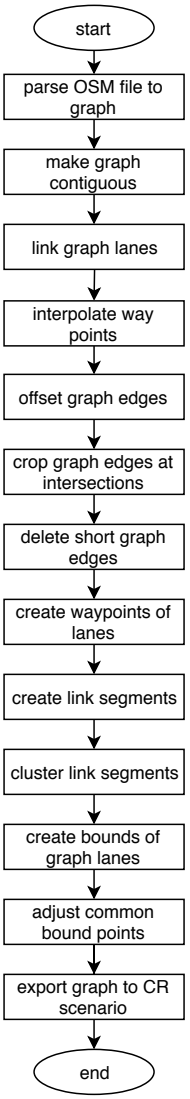


Figure 4.1

Most of these steps are described in a separate section in the following. If appropriate, steps are divided into several sections or multiple steps are combined into one section.

4.1 Parsing

This approach converts the entire road network saved in an OSM file to a CR scenario. To do so, we extract the ways stored in that file, which represent roads. Different types of roads are represented by a tag for each way. These types can be used to filter the extracted roads with a set of desired types. Only the roads of a desired type will be extracted.

We also parse the turn lane tags of each way. To simplify the processing of these tags, we distinguish only between six tags: `merge_to_left`,

`merge_to_right`, `left`, `right`, `through`, and `none`. If the tag is not composed of these, we parse it according to the following pattern. First, we create am empty list l_{tags} of tags. Then, we check if the tag contains `left`, if so we append `left` to l_{tags}. After that, we check if the tag contains `through` and if so, we again append `through` to l_{tags}. At last, we check if the tag contains `right` and add it to the list accordingly. Having done this, we concatenate all elements of l_{tags} separated by a semicolon. If l_{tags} is empty we use `none` as our result. This pattern converts tags specifying the direction any further than the given three directions to tags that only use these three directions. E.g. `sharp_left;through;slight_right` will be converted to `left;through;right`.

For memory efficient computation only the OSM nodes referenced in these ways are extracted. We save this data for further processing in the following steps.

4.2 Projection

Since OSM nodes are defined by their latitude and longitude, they are points on the earth's spheroidal surface and have to be projected onto a plane to fit the CR format. In order to do so, the center of all extracted nodes is calculated and used as origin of the local coordinates. To obtain the projection of each node, the difference to the center in latitude and longitude of each node is translated to distance in meters. For this purpose, we approximate the earth as a sphere with the radius $r = 6371000m$ [10]. Therefore, the distance on the earth's surface corresponding to one degree in latitude is given by the following formula [11]:

$$\Delta_{latitude} = \frac{\pi}{180} \cdot r \approx 111319m \tag{4.1}$$

The distance of a degree in longitude is dependent on the latitude and calculated by the following [11]:

$$\Delta_{longitude} = \frac{\pi}{180} \cdot r \cdot \cos(latitude) \tag{4.2}$$

This method ignores the fact that the earth is not perfectly spherical and projects all coordinates onto a plane. Therefore, the positions of all points are not exactly correct, but can vary by up to 0.5% [12]. For our purposes this should suffice, as we only want to convert relatively small regions. Furthermore, the OSM data is not necessarily accurate.

This projection will also not work if the map contains one of the poles. Since the resulting map is always oriented to north, areas around the poles will be distorted and shown below or above them.

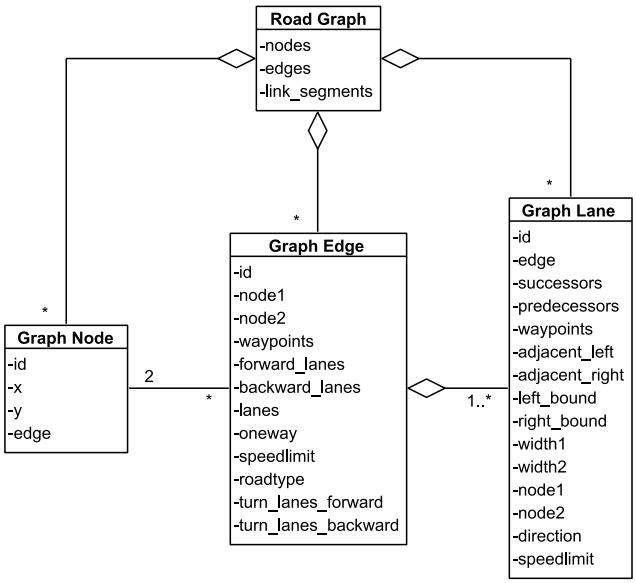


Figure 4.2: Class model for the graph structure

4.3 Graph Structure

Given the spatial information of OSM nodes, roads can be represented on a plane. Yet, this format does not contain the necessary information to represent roads in a CR scenario, which explicitly contains spatial information of every lane in the scenario. Therefore, it needs to be deducted or estimated. To do so, the scenario is represented by a graph structure, which can be manipulated easily and whose information content can be increased iteratively. This structure, as usual for graphs, mainly consists of nodes and edges, where a graph edge represents a road and a graph node represents an intersection. To hold information of individual lanes, we also define graph lanes, which can be part of an edge but do not have to be.

Graph Nodes

A node in the graph structure represents an intersection and can be derived directly from an OSM node. Each OSM node that is referenced by two or more OSM ways or is at the end of such a way is converted to a graph node. For further operations graph nodes have an id, contain information of their position, and reference edges that start or end at them.

Graph Edges

A graph edge is derived from an OSM way and therefore represents a road. It builds the connection of two graph nodes. A graph edge also contains the OSM nodes referenced by its corresponding OSM way as an ordered list of points starting with the position of its first referenced graph node, then passing an arbitrary finite number of way points and finally ending with the position of its other referenced graph node. Additionally, graph edges hold properties of the road they are representing. These consist of the number of lanes directing forward and backwards, the total sum of lanes, if the road is a one-way street, which type the road is of, what speed limit applies on it, and the turn lanes derived from the OSM way. This information is either given by tags of the OSM way or estimated as described in Section 4.4. Since CR scenarios are primary defined by lanelets, which represent single lanes. Graph edges, on the other hand, represent roads, which can have more than one lane and are therefore not suitable for the conversion. To generate lanelets, graph edges reference graph lanes, from which lanelets can be derived. The graph lanes of a graph edge are saved in an ordered list from left to right according to the direction of the graph edge.

There can, in contrast to usual graphs, exist more than one graph edge linking a pair of graph nodes. This is the case, because graph edges do not only describe, which graph nodes are linked, but also hold information about the link. An example could be two roads connecting the same two intersections. Graph edges in this format are directed, which is be important to set the direction and position of lanes.

Graph Lanes

Graph lanes represent lanes on a road. Graph lanes are either associated with a graph edge or represent a link between two different roads. In the latter case, they are not directly related with a graph edge. Their main purpose is to hold all information needed to derive a CR lanelet from them. This contains a unique id, adjacent lanes at the start, end, left, and right of a lane, the direction of left and right adjacent lanes, its boundaries and the applying speed limit. Most of this information cannot be trivially extracted from OSM data and has to be estimated. This estimation is described in the following sections and uses additional information about lanes, such as their turn lane, if existent their associated graph edge, and their width at the start and end.

4.4 Deduction and Estimation of Missing Information

To convert an OSM file to the graph structure, introduced in the previous section, one needs to specify information that is possibly not contained in the file. The given

information is often incomplete and sometimes even contradicting. This is the case, because OSM data is created by many users by hand, who sometimes interpret format specifications differently or make mistakes. This concerns primary the number of lanes on a road, but also whether the road is a one-way street, and which turn lanes apply to it. To handle this inconsistent format, the given information is processed in two steps. During the first step it is checked, if missing information can be logically deduced by the given data. Information that could not be deduced in the first step is estimated in the second step.

Step 1: Deducing Missing Information

In this step the total sum of lanes n_{sum} on a road, the number of lanes directed forward $n_{forward}$ and backwards $n_{backward}$ are deducted for each road if possible. A lane is directed forward, if it has the same direction as the OSM way. Since the turn lane tags in OSM are defined for each lane separately, the number of turn lanes n_{t-s} implies the sum of total lanes. So, if given, the number of lanes on a road is set to the count of turn lanes:

$$n_{sum} := n_{t-s} \tag{4.3}$$

The turn lane tags can be specified by tags for the forward directed lanes n_{t-f} and backward directed lanes n_{t-b}. In this case $n_{forward}$ and $n_{backward}$ are set individually:

$$n_{forward} := n_{t-f} \tag{4.4}$$

$$n_{backward} := n_{t-b} \tag{4.5}$$

This is done regardless of whether the number of lanes has already been specified or not, on grounds that in case of an inconsistency we expect the more detailed description of the street, which is defined by the turn lanes, to be more likely correct than the raw number of lanes.

In OSM data there is often only $n_{forward}$ and $n_{backward}$ or n_{sum} specified. The latter is often used for one-way streets. For a consistent format, we set all three numbers, i.e., $n_{forward}$, $n_{backward}$ and n_{sum}, so that the following holds:

$$n_{sum} = n_{forward} + n_{backward} \tag{4.6}$$

Again, in a case of inconsistency we prefer the more detailed specification, the count of directed lanes, over the number of total lanes.

$$n_{sum} \neq n_{forward} + n_{backward} \Rightarrow n_{sum} := n_{forward} + n_{backward} \tag{4.7}$$

Finally, the attribute defining whether the road is a one-way street *is_oneway* has to be updated, since it could have been not set yet, or it could be inconsistent with the number of lanes directed backwards.

$$is\_oneway \Leftrightarrow n_{backward} = 0 \tag{4.8}$$

One-way streets are usually represented by an OSM way that has no backward lanes, therefore, $n_{backward}$ should be zero. In rare cases, where an OSM way has backward directed lanes and no forward directed lanes, the whole way is turned after this step. To do so, $n_{forward}$ and $n_{backward}$ as are swapped and the direction of the road is reversed.

After this step, $n_{forward}$, $n_{backward}$, n_{sum}, and *is_oneway* are all set if enough values were given in the OSM data or could be derived from turn lane tags. But there can be cases in which some information is still missing, e.g., when nothing or only n_{sum} was given. Since these missing values cannot be deduced in a logical manner, they have to be assumed, which is described in step 2.

Step 2: Estimating Missing Information

To complete all missing values, we follow a simple pattern:

If we do not know whether a road is a one-way street, we just assume that it is not.

$$is\_oneway = undefined \Rightarrow is\_oneway := False \tag{4.9}$$

This can only happen if we also do not know the count of forward and backwards directed lanes, because given that, it could be deduced in the previous step whether the road is a one-way street.

If the n_{sum} is unknown and also $n_{forward}$ and $n_{backward}$ are not given, n_{sum} is set according to a configurable table, which defines how many lanes $n_{roadtype}$ each type of road typically has. An exception from that is when the count of lanes going in one direction is given. Then the number of lanes going in the opposite direction is set to half of the value given in the table, which sets the total number of lanes to the sum of these values.

$$n_{sum} := \begin{cases} n_{roadtype} & \text{if } \neg is\_oneway \\ \lceil n_{roadtype}/2 \rceil & \text{if } is\_oneway \end{cases} \tag{4.10}$$

If n_{sum} is unknown but either $n_{forward}$ or $n_{backward}$ is given by a value $n_{direction}$, n_{sum} is also set to the sum of $n_{direction}$ and the half of $n_{roadtype}$, but not smaller than $n_{roadtype}$.

$$n_{sum} := \max(n_{roadtype}, \lceil n_{direction} + n_{roadtype}/2 \rceil) \tag{4.11}$$

Now that the n_{sum} is set in every case, if missing, the $n_{forward}$ and $n_{backward}$ have to be set. If the road is a one-way street the number of $n_{backward}$ is set to zero and the count of forward oriented lanes is set equal to n_{sum}.

$$is\_oneway \Rightarrow n_{backward} := 0 \tag{4.12}$$

$$is\_oneway \Rightarrow n_{forward} := n_{sum} \tag{4.13}$$

If only one of the two values $n_{forward}$ and $n_{backward}$ is undefined, the respective other is set so that equation 4.6 holds. Otherwise, both numbers are set to half the total count and if that is odd, the number forward oriented lanes is increased by one.

$$n_{forward} := \begin{cases} n_{sum}/2 & \text{if } n_{sum} \text{ even} \\ \lfloor n_{sum}/2 \rfloor + 1 & \text{if } n_{sum} \text{ odd} \end{cases} \tag{4.14}$$

After these steps, all information is set to convert an OSM way to a graph edge. It is important to note that at this point assumptions about the road network have been made, which can be erroneous. Thus, the resulting scenario can differ from the real world.

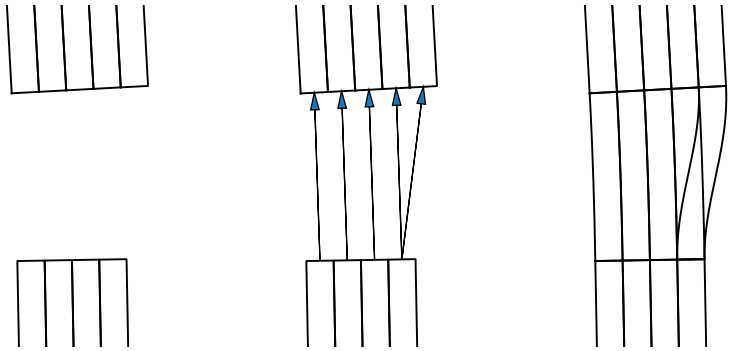
4.5 Linking Lanes

One big downside of OSM data for our purposes is, that other than the number and direction of lanes and the turn lane tag no information about the lanes of a street can be saved. Therefore, we cannot directly see which lanes are reachable from a certain lane at an intersection. Actually, we only know that roads with given numbers of lanes lead to certain intersections. This is surely enough for maps and navigation applications, which are the main utilizations of OSM data. To generate a realistic representation of real road networks in the CR format on the other hand, we need to know the exact courses and links of graph lanes at graph nodes.

The linkage of graph lanes at graph nodes is described in this section. The method in this section assumes right-hand traffic. Linking to graph lanes l_1 and l_2 means to set the successor of l_1 to l_2 and the predecessor of l_2 to l_1. For a simpler and more structured approach, graph nodes are divided into different degrees. The degree of a graph node is simply denoted by the number of graph edges connected to it. The linking process is performed on each graph node individually.

4.5.1 Linking a Graph Node of Degree Two

Intuitively, one probably would not consider two successive roads as an intersection. However, as we are working on our graph structure derived from OSM ways, these occur quite frequently, e.g. when the number of lanes on a road changes.



(a) Input: Graph lanes at a graph node

(b) Links between predecessors and successors visualized by arrows

(c) Desired output after the whole conversion

Figure 4.3: Visualization of an intersection of degree two

The linking task is divided into two steps, one for each graph edge *leading* to the graph node. We call a graph edge *leading* to a graph node if it is connected to it, regardless of whether it has lanes leading to it. In each step the graph lanes leading to the graph node l_{inc} and the graph lanes of the other graph edge starting at the graph node l_{out} are collected. Now every graph lane has to be connected to at least one other lane, otherwise they would start or end without a predecessor or successor.

Same Count of Lanes: In the easiest case l_{inc} and l_{out} have the same count. Since they are ordered in a list from left to right, each incoming graph lane $l_{inc}[i]$ with an index i can be linked with the outgoing graph lane at the same index $l_{out}[i]$. This can be done by calling Algorithm 1 with $inc\_start := 0$, $out\_start := 0$, and $length := length(l_{inc})$. The variable *inc_start* stands for the index of the first graph lane leading to the graph node, that will be linked. *out_start* stands for the index of the first outgoing graph lane.

Algorithm 1 link_interval(l_{inc}, l_{out}, *inc_start*, *out_start*, *length*)

1: **for** $i \leftarrow 0$ to $length - 1$ **do**
2: $\quad l_{inc}[inc\_start + i].successor \leftarrow l_{out}[out\_start + i]$
3: $\quad l_{out}[out\_start + i].predecessor \leftarrow l_{inc}[inc\_start + i]$
4: **end for**
5: **return**

As visible in Figure 4.3 l_{inc} and l_{out} can have different numbers of graph lanes. Then,

either new lanes start at the intersection modeled by the current graph node or lanes end at it.

Lanes Split: When new lanes start at the intersection, i.e. $length(l_{inc}) < length(l_{out})$, we decide first where the new lanes should be. Since new lanes are usually not created in the middle of the road, the new graph lanes are added either left or right of the other graph lanes. The new graph lanes do not have to be added, because they are already contained in l_{out}. Instead of adding graph lanes to the left or right we declare $\Delta = length(l_{out}) - length(l_{inc})$ graph lanes in l_{out} as *new*. Here, the first graph lanes of l_{out} correspond to the far left lanes on the road and the last correspond to the far right lanes on the road.

To decide whether we choose the first or last graph lanes, we follow a simple pattern. We assume that by default new lanes are created at the left of the road, which is usual for freeways as well as rural roads. However, we make an exception if the last graph lane in l_{out}, which corresponds to the far right lane on the road, has the turn lane tag `right`. We assume that in this case, a new turn-off lane is created.

Then, we link all lanes which are not *new* like before when $length(l_{out})$ and $length(l_{inc})$ were equal. This can be done by calling Algorithm 1 with either $inc\_start := 0$, $out\_start := \Delta$, and $length := length(l_{inc})$ when the first graph lanes of l_{out} are *new*, or with $inc\_start := 0$, $out\_start := 0$, and $length := length(l_{inc})$ in the other case. Then either the first or last Δ graph lanes in l_{out} are set as successors of the first or last graph lane in l_{inc} additionally to its regular successor.

Lanes Merge Together: The procedure for $length(l_{inc}) > length(l_{out})$ is similar. Either the first or last $\Delta' = length(l_{inc}) - length(l_{out})$ graph lanes in l_{inc} are declared as *new* and are set as predecessors of the first or last graph lane in l_{out} additionally to its regular predecessor.

To decide whether we choose the first or last graph lanes in l_{inc}, we follow a simple pattern again. Anew, we assume that by default lanes end at the left of the road. Here we make an exception if the last graph lane in l_{inc} has the turn lane tag `merge_to_left`. Here it is directly specified by the turn lane tag that the right lane merges to its left neighbor.

Then the graph lanes not marked as *new* are linked. Again, this can be done by calling Algorithm 1. When the first Δ' graph lanes are *new*, we use the arguments $inc\_start := \Delta'$, $out\_start := 0$, and $length := length(l_{out})$. In the other case, we use the arguments $inc\_start := 0$, $out\_start := 0$, and $length := length(l_{out})$.

4.5.2 Linking a Graph Node of Degree Three

Intersections with three roads can for example be T-junctions or road forks. Each road can but does not have to be a one-way street. To proceed systematically, we look at each graph edge leading to the graph node individually. We link only the graph lanes leading to the graph node to their successors. As these successors are the graph lanes heading away from the graph node, we do not need to treat them separately.

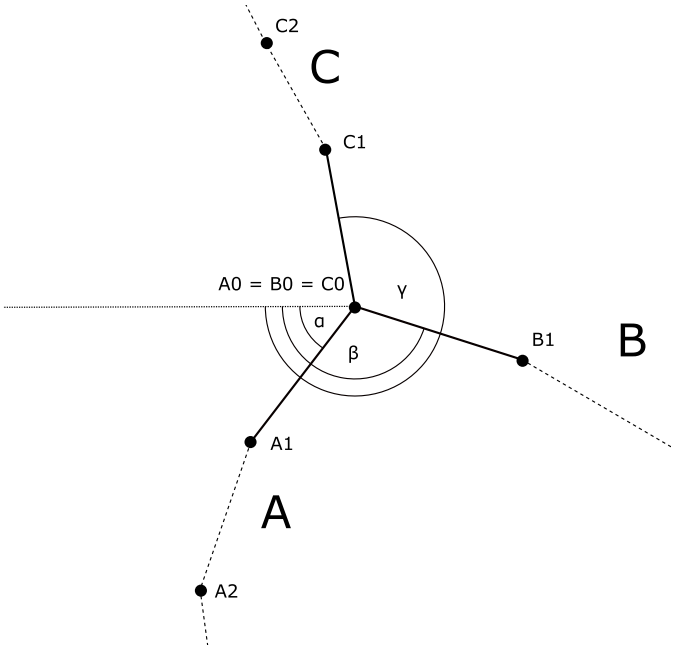


Figure 4.4: Visualization of orientation of graph edges

Left and Right Edges: When linking a specific graph edge, we divide the other two edges into its left and right neighbor. To do so, we take the set of graph edges and order it according to their orientation. As visible in Figure 4.4, the orientation of a graph edge depends not only on the edge but also on the graph node, at which we want to get the orientation. Since the course of a graph edge is defined by a list of way points, we first need the correct end of the list to compute the orientation. If the graph edge starts at the given graph node, we use the first two way points, otherwise the last. In the figure the edges A, B and C can be seen. A and C start at the graph Node, B ends at it. The orientation of A at this node is α, the angle between vector $\overrightarrow{A0A1} = A1 - A0$ and the negative x-axis, which is depicted as a dotted line. The orientation for edge B is given by β, which uses $\overrightarrow{B5B4}$ as reference vector. The angle and therefore the orientation can

be a value in the Interval $[0, 2\pi)$. To get the left neighbor of a graph edge in a sorted list, we take the previous element and for the right neighbor we take the subsequent element. If there is no previous or subsequent element, we take the last or first element of the list, respectively.

Turn Lane Tags: To decide whether a graph lane leads to its edges' left or right neighbor, we gather information of the turn lane tags if possible. Since the turn lane tags are designed for navigation, they are often unsuitable for this task. To ensure that we only use suitable tags for the linking process, we check several criteria on the set of graph lanes leading to the respective graph node. First, we check if there is a merge turn lane in that set. We assume that lanes do not merge on intersections and therefore decide that the turn lane tags are not suitable in this case. Second, we count the occurrence of turning directions. These directions are divided into *through*, *left*, *right* and *none*. A single lane can be counted multiple times, if it leads in several directions. For example, a lane with the tag `through;right` leads straight across the intersection and to the right. Then we match the numbers of directions with the actual lanes available. In the case of a graph node with a degree of three, this is slightly more complicated than for a node with degree of four. Since at graph nodes with a degree of three there are only two graph edges reachable from a graph edge, we divide these in left and right edge. In contrast to that, the turn lane tags may contain `through` tags, for example at T-junctions.

To handle this inconsistency, we switch the turn lane tags from `through` to either `right` or `left`. To do so, we use the previously counted numbers of directions. If the numbers for *through*, *left*, and *right* are all greater than zero, we have more directions given, than graph edges available in the current graph node. Therefore, we assume that the turn lane tags are placed wrongly or refer to another intersection and consider them as not suitable. If the number of *none* tags equals the number of graph lanes leading to the graph node, the turn lanes do not provide any useful information and we can consider them to be not suitable as well.

Next, we match the cases in which two directions have a greater count than zero. In the easiest case, these directions are *left* and *right*. Then we can simply check if the graph lanes available at the respective intersections are less than the count of graph lanes leading to them. If so, we consider the tags as not suitable. In the other cases with two directions, one of them must be *through*. Since we only have a left and a right graph edge, the *through* direction will be assigned to one of them. If the other direction is *left*, we assign it to the right graph edge. Otherwise, the other direction is *right*, so we assign the *through* direction to the left graph edge.

In case the count of only one direction is greater than zero, we match this count

with the respective graph lanes. If this direction is *left* or *right*, we can simply check if the number of available lanes at the left or right edge is lower than the count of the direction, respectively. Then we consider the tags as not useful. If the only direction greater than zero is *through*, we have to assign a graph edge to that direction. We use the orientation of the other graph edges like before to determine which one will be assigned to it. The graph edge with the largest angle to our current graph edge will be assigned to to the *through* direction. Here we calculate the angle α between two graph edges e_1 and e_2 as the follows:

$$\delta = orientation(e_1, node) - orientation(e_2, node) \tag{4.15}$$

$$\alpha = min(\delta, 2\pi - \delta) \tag{4.16}$$

Since we want to calculate the angle between the graph edges independent of their absolute orientation, we take the minimum of the angles clockwise (δ) and counter-clockwise ($2\pi - \delta$). After assigning the direction, it is checked again if there are enough graph lanes available and then it the tags are considered as suitable or not suitable accordingly.

Now, the turn lane tags are set according to the left and right graph edges and the linking of the individual tracks can be performed. This is done by the same function as for graph nodes with degree of four and will be described in the next subsection.

4.5.3 Linking a Graph Node of Degree Four

Intersections with four roads leading to it are common, these occur for example when two streets cross each other. The linking procedure for graph nodes of degree four is similar to the method for graph nodes of degree three. First, the graph edges are sorted accordingly to their orientation, which is described at the previous section. Then, we start the process for each graph edge independently. Now the remaining three graph edges are categorized into left edge, right edge, and through edge. For a graph edge at index i in the sorted list of graph edges l these are set as the following:

$$edge_{left} := l[(i-1) \bmod 4] \tag{4.17}$$

$$edge_{through} := l[(i-2) \bmod 4] \tag{4.18}$$

$$edge_{right} := l[(i-3) \bmod 4] \tag{4.19}$$

Where mod is the modulo function and $l[i]$ is an element of l at index i. Then it is checked whether the turn lane tags are suitable for the linking process. To do so, we again count again the occurrences of every direction in the tags. If the tags `merge_to_right` or `merge_to_left` occur, the tags are considered as not suitable. Also if the count of a direction is greater than the lanes available at the respective graph edge,

the tags are considered as unsuitable. Since every direction can easily be assigned to a graph edge, we do not have to switch any tags.

Linking Heuristics

This subsection describes the pattern specifying where links between two graph lanes will be created. The graph lanes are sorted from left to right. To connect each graph lane of one graph edge leading to a graph node to its successors, we categorize them into three groups. The graph lanes in a certain group have a successor in the respective direction of their group. These directions are *left*, *through* and *right*. For these groups the following rules apply: They can intersect by at most one graph lane. E.g. two lanes can lead to a graph node, the left leads only left, but the right can be used to turn right or drive across. Lanes that are left from a lane in the *left* group can only lead to the left. Lanes left from a lane in the *through* group can only lead left or through. Lanes right from the *through* group can only lead through or right. And at last, lanes right from the *right* group can only lead right. To our knowledge, these rules do apply to all junctions with structured traffic flow.

Getting Bounds of Groups: We describe the groups with the indices at which they start and end. Since the *left* group starts with the first graph lane and the *right* group ends with the last, we only need the following indices as borders: $last\_left$, $first\_through$, $last\_through$, and $first\_right$. To determine the borders for each group, we use also bounds in which the groups must be at any time. These bounds are $upper\_bound\_left$, $lower\_bound\_through$, $upper\_bound\_through$, and $lower\_bound\_right$.

$$upper\_bound\_left := \min(length(incoming) - 1, length(outgoing\_left) - 1) \tag{4.20}$$

$$lower\_bound\_through := 0 \tag{4.21}$$

$$upper\_bound\_through := length(incoming) - 1 \tag{4.22}$$

$$lower\_bound\_right := \max(0, length(incoming) - length(outgoing\_right)) \tag{4.23}$$

At the beginning, we can already set these borders to some initial values. The index $upper\_bound\_left$ cannot be larger than the number of incoming lanes $incoming$ or the number of lanes going out at the left $outgoing\_left$ minus one. $lower\_bound\_through$ cannot be smaller than zero. $upper\_bound\_through$ has to be smaller than the length of the incoming graph lanes. $lower\_bound\_right$ also cannot be smaller than zero and since there can only be as many graph lanes leading to the right as there are lanes outgoing right, it has to be at least $length(incoming) - length(outgoing\_right)$.

Then we iterate over the turn lane tags, if they were considered suitable before. At each tag with index i we set the bounds and borders according to the implications of the tag. If the current tag is `left`, *lower_bound_through* and *lower_bound_right* have to be larger than i. If it is `through`, $upper\_bound\_left$ has to be smaller than i and *lower_bound_right* has to be larger than i. For `right` we set $upper\_bound\_left$ and $upper\_bound\_through$ smaller than i. If a tag consists of more than one direction, we can set some specific borders. For `left;through` we can set $last\_left$ and $first\_through$ to i and *lower_bound_right* same as for `through` greater than i. For `through;right` we can set *last_through* and $first\_right$ to i and $upper\_bound\_left$ has to be smaller than i. If the tag is `left;right`, we know for sure that there are no lanes heading through. For this case we set *last_through* smaller than $first\_through$, the resulting interval is empty in this case. Also we can set $last\_left$ and $first\_right$ to i. When the tag is `left;through;right` we can set all borders directly, since all of them have to be at i. When the tag is `none` we cannot deduce any information.

Setting Borders for Suitable Turn Lane Tags: After this iteration, we update the bounds according to the borders we already got from the tags. Then we set the still missing borders according to the bounds. To do so, we begin with the graph lanes of the *through* group. If $first\_through$ is not set yet, we distinguish two cases: If *last_through* is set we compute like the following:

$$nr := \min(length(incoming), length(outgoing\_through)) \tag{4.24}$$

$$first\_through := \max(last\_through - nr + 1, lower\_bound\_through) \tag{4.25}$$

Where *outgoing_through* is the list of lanes leaving the graph node in *through* direction. Otherwise, we use this:

$$first\_through := \max(lower\_bound\_through, upper\_bound\_through - nr + 1) \tag{4.26}$$

We only make an exception from that if nr equals zero. Then we set:

$$first\_through := lower\_bound\_through \tag{4.27}$$

This is important, because the restrictions of the groups still hold, if the group *through* is empty. The value for $first\_through$ is set so that the through group does not interfere with the other groups. Then we set *last_through* accordingly:

$$last\_through := \min(upper\_bound\_through, first\_through + nr - 1) \tag{4.28}$$

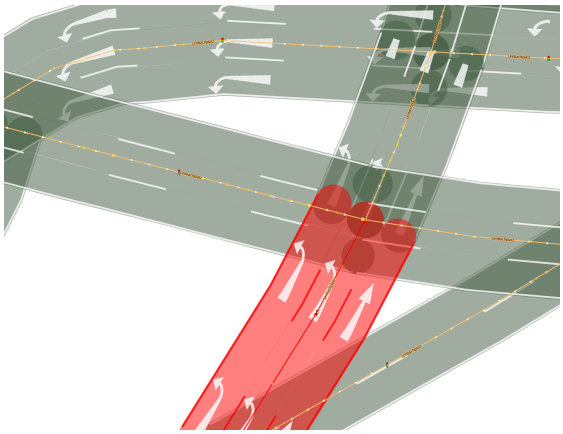


Figure 4.5: The turn lane tags of the street going from the lower left to the upper right are set at the part before the intersection (red), but should apply afterwards. The street in the middle going from the upper left to the right is a one-way street. This is a screen shot taken from the OSM file editor tool JOSM[1]

Next, we set the borders for the left and right group:

$$nr_{left} := \min(length(incoming), length(outgoing\_left)) \tag{4.29}$$

$$last\_left := \min(upper\_bound\_left, nr_{left} - 1) \tag{4.30}$$

$$nr_{right} := \min(length(incoming) - last\_through, length(outgoing\_right)) \tag{4.31}$$

$$first\_right := \max(lower\_bound\_right, length(incoming) - nr_{right}) \tag{4.32}$$

After this the borders for all groups are set, if the turn lane tags were considered suitable. Otherwise, we follow another method to set all borders.

Setting Borders Without Turn Lane Tags: First we use a little trick, that improves results for a very specific pattern. Sometimes the turn lane tags for one intersection are set multiple consecutive road in advance. As visible in Figure 4.5 the turn lane tags for some roads are also set for their predecessors although they do not apply to them. This is useful for navigation software, because the user the can see in advance which lane he has to get onto. However, this is inconvenient for our purpose, since we want to use these tags for every graph node, i.e. for every intersection. In this specific case, the turn lane tags are the same as they are for the following road. So to reduce errors in this case, we assume that if the turn lane tags of a graph edge are not suitable and the tags of the successor, i.e. $edge_{through}$, are the same, we set all lanes to lead *trough*.

[1] `josm.openstreetmap.de`

If this trick is not applied we will assume the direction of each graph lane by the following pattern: First, we compute the nr of links possible in each direction:

$$nr_{through} := \min(length(incoming), length(outgoing\_through)) \tag{4.33}$$
$$nr_{left} := \min(length(incoming), length(outgoing\_left)) \tag{4.34}$$
$$nr_{right} := \min(length(incoming), length(outgoing\_right)) \tag{4.35}$$

Additionally, we compute the angle between the current graph edge and $edge_{left}$ and $edge_{right}$. This is done as described in formula 4.15 and 4.16. If the angle is below a certain threshold, which can be adjusted manually, it is considered as a sharp angle. If the angle of one of these edges is sharp, it is not connected and the respective number is set to zero. Then we set the borders:

$$first\_through := \max(0, \min(nr_{left}, length(incoming) - nr_{through} - nr_{right})) \tag{4.36}$$
$$\begin{aligned} last\_through := \min(&length(incoming) - 1, \\ &\max(-1, length(incoming) - nr_{right} + 1), \\ &first\_through + nr_{through} - 1) \end{aligned} \tag{4.37}$$
$$last\_left := \max(-1, \min(first\_through, nr_{left} - 1)) \tag{4.38}$$
$$first\_right := \max(0, last\_through, length(incoming) - nr_{right}) \tag{4.39}$$

As described in formula 4.36, the *through* group starts at either the first graph lane after which all *through* and *right* lanes have room, or at the fist lane after the *left* group. To prioritize the *through* group, we always choose the smaller value, but it cannot be smaller than zero. For the upper border of the *through* group the following holds: It is not larger than the index of the last incoming graph lane. It must be lower or equal to the maximum of -1 and the index of the first lane of the *right* group. Finally it is not larger than the lower bound plus the nr of *through* lanes minus one. This assures that the group is not larger than the number of *through* lanes. In case of no *through* lanes, the upper bound will be the lower bound minus one. The *left* group extends to the nr_{left}th lane or to the first *through* lane, if that has a smaller index. It must not be smaller than minus one for a more structured value range. The index of the first *right* lane is set so that the *right* group has nr_{right} lanes, but must not be smaller than 0 or $last\_through$.

Linking Lanes According to Groups With the bounds set for each group, we can start linking the graph lanes of each group with their respective successors. The *left* group will be connected with the leftmost graph lanes of $edge_{left}$. We make an exception, if the leftmost lane of $edge_{left}$ hast the turn lane tag `left`, the next lane does not, and

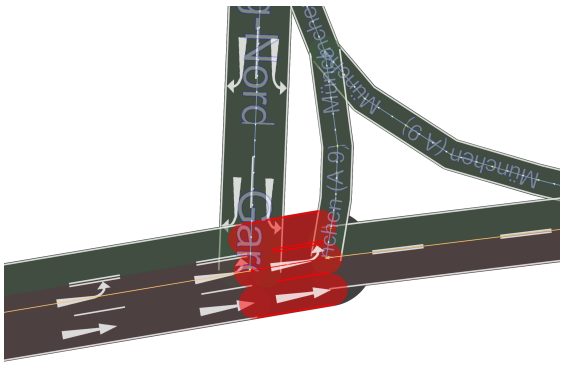


Figure 4.6: An intersection on which we do not link the left turning lane of the road heading downwards from the upper center to the first available lane on the left (red), because it turns left immediately afterwards

$edge_{left}$ has one more lane in this direction, that can be linked. By that exception we can avoid that lanes turn left on another lane, which will turn left immediately after that. Otherwise, these lanes could form U-turns instead of left turns. An example for this special case can be seen in Figure 4.6. To link the respective graph lanes, we call Algorithm 1 with the arguments $inc\_start := 0$, $out\_start := 0$, and $length := last\_left + 1$, in the normal case. For the special case we use $out\_start := 1$. The *through* group will be linked with graph lanes of $edge_{through}$ by calling Algorithm 1 with the arguments $inc\_start := first\_through$ and $length := last\_through - (first\_through - 1)$. The value for out_start is chosen so that the connected lanes are in the center of the outgoing lanes:

$$through\_nr := last\_through - (first\_through - 1) \tag{4.40}$$

$$out\_start := \lceil (length(outgoing\_through) - through\_nr)/2 \rceil \tag{4.41}$$

Please note that $through\_nr$ does not necessarily have to be equal to $nr_{through}$. This is the case because the restrictions at the border finding process can change the number of lanes actually linked in *through* direction. The *right* group will be connected to the rightmost lanes of $edge_{right}$.

$$right\_nr := length(incoming) - first\_right \tag{4.42}$$

The count of lanes connected to the right is $right\_nr$. For the linking to the *right* edge, we call Algorithm 1 with the arguments $inc\_start := first\_right$, $out\_start := length(outgoing\_right) - right\_nr$, and $length := right\_nr$.

After performing this procedure on every graph edge leading to the current node, all graph lanes at the current graph node are connected in the most common cases. It can occur in some cases that graph lanes at the current node have no predecessors or successors. Then they are linked according to a method described in the after next subsection.

4.5.4 Linking a Graph Node of a Higher Degree

Intersections with a higher degree than four can occur at large intersections mostly at the center of large cities. Since the turn lane tags are the same for all roads, it becomes more complex to match a turn lane tag to a graph edge. Therefore, and because they are not very frequent, we choose a simpler method to link all intersections with a degree higher than four.

This method iterates over all graph edges at the current graph node and connects the *incoming* graph lanes of that edge leading to the node to their successors. In contrast to graph nodes with degree three and four, we do not distinguish the other edges. The edges are again sorted counterclockwise based on their orientation with respect to the current node, as described before. For every edge we save the other edges in a counterclockwise sorted list *other_edges*, starting with its right neighbor. For every graph edge *other_edge* in *other_edges* we compute the angle to the current edge like described in the previous subsection. If the angle between the edges is not considered *sharp*, we link the *incoming* graph lanes with the *outgoing* graph lanes of the other edge.

$$right := \begin{cases} True & \text{if } i < \lfloor length(other\_edges)/2 \rfloor \\ False & \text{else} \end{cases} \tag{4.43}$$

$$nr_{links} := \min(length(incoming), length(outgoing)) \tag{4.44}$$

By the index i of *other_edge* we can determine if it is *right* from the current edge. The number of graph lanes that will be connected nr_{links} is the minimum of the count of the graph lanes *incoming* from the current edge and *outcoming* from *other_edge*. Then the respective graph lanes are linked differently, for *right* and left neighbors by calling Algorithm 1. The argument *length* is given by nr_{links}. For a *right* neighbor we choose $inc\_start := length(incoming) - nr_{links}$ and $out\_start := length(outcoming) - nr_{links}$. For a left neighbor we choose $inc\_start := 0$ and $out\_start := 0$. After this is done for every graph edge, it is till possible that some graph lanes do not have a successor or predecessor at the current graph node. To link these lanes, we use a method described in the next subsection.

Please note that this method simplifies the linking process considerably. Hence, it will produce unsatisfactory results for some scenarios.

Linking of Disregarded Lanes

After the linking process of graph nodes of degree 3 or higher, there can be graph lanes without successors or predecessors left. Since these successors and predecessors could not be found by our regular method, we use a different approach to link them.

For every edge, we have two lists of graph lanes. Lanes which are leading to the current graph node *incoming* and lanes which are heading away *outgoing*. If a graph lane with index i in *incoming* has no successors we find the next lane with index j in incoming, that has a successor. We do this by searching with an iteratively increasing distance k. So, at first we check if the elements with index $i-1$ and $i+1$ exist and have a successor. At the kth step we check for the $i-k$th and $i+k$th elements. If a graph lane with a successor is found, we also set it as a successor for the current graph lane. If the found graph lane has several successors, we arbitrarily chose one of them as a successor for our current graph lane. If no successor can be found, the graph lane remains unlinked.

If a lane in *outgoing* has no predecessor we search for the nearest graph lane in *outgoing* with a predecessor similarly to the way we proceeded above. We pick this predecessor and set it as a predecessor for the current lane.

This if performed for all graph edges at the current node. Please note, that this procedure is a simple approach, which is rarely necessary and for single unlinked graph lanes.

After linking the graph lanes at each graph node, the logical connections for the road graph are set. To continue, we need to determine the spatial courses of all roads and lanes. How the course of graph edges between way points is defined is described in the following sections.

4.6 Interpolation

The spatial course of an OSM way is specified by many OSM nodes. Thus, the course of a graph edge at the current state is specified by a list of way points. This course can be visualized by connecting all successive way points with straight lines. However, these way points have large distances at most OSM maps, which is not accurate enough for CR scenarios. To improve this we interpolate the course of roads between two way points. The main requirement for this interpolation is a smooth result. This can be achieved by representing each segment between two way points as a curve, that hast to be tangential to its predecessor and successor.

A way to perform this interpolation is described in [7]. Similar to that, we will use cubic Bézier curves for each segment between two way points. Bézier curves are parametrized by two or more control points, cubic Bézier curves are defined by four

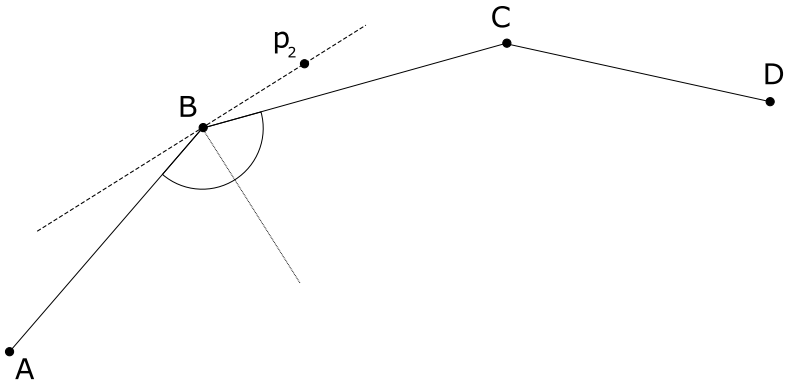


Figure 4.7: Determine p_2 for Bézier curve. The dotted line represents the angle bisector for $\overrightarrow{BA}$ and $\overrightarrow{BC}$. The dashed line represents the tangent

points. A Bézier curve is always located in the convex hull of its control points. A Bézier curve defined by the points P_0 - P_n can be evaluated at a certain point t with [13]:

$$P(t) = \sum_{i=0}^{n} B_{i,n}(t) P_i \tag{4.45}$$

Where $B_{i,n}(t)$ is a Bernstein polynomial:

$$B_{i,n}(t) = \binom{n}{i} t^i (1-t)^{n-i} \tag{4.46}$$

To get the control points p_1 - p_4 of a segment between two way points B and C we do as follows. The start point of the curve p_1 is set to B and the end point p_4 is set to C. At its start the curve will be tangential to the line going through p_1 and p_2. Therefore, we need to set p_2 accordingly to the tangent of the current way point segment and the previous one. To do so, we need the way point before B which we call A. We will define the tangent as the orthogonal of the bisector of $\overrightarrow{BA}$ and $\overrightarrow{BC}$. The point p_2 will be on this tangent, as visualized in Figure 4.7. The distance from B to p_2 is set by $\|\overrightarrow{BC}\|_2 \cdot d$ where d can be dynamically tuned in the interval $d \in [0, 0.5]$. The greater d, the stronger the resulting curvature. For p_3 we build the tangent of $\overrightarrow{CB}$ and $\overrightarrow{CD}$ as before, where D is the successor of C. At the ends of a graph edge, the way points do not have a successor or predecessor respectively. In that case p_2 or p_3 will be set on the line segment $\overrightarrow{BC}$.

After a Bézier curve is defined, we evaluate it at several points to get a new list of

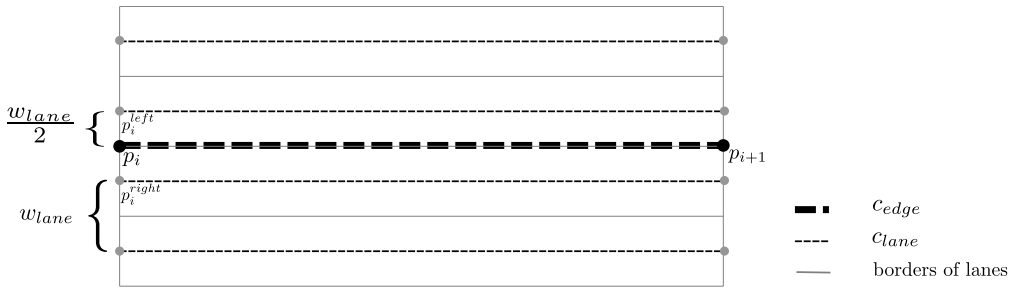


Figure 4.8: Visualization of generated lane courses between two way points

way points, which approximate the course of the graph edge better.

$$n_{segments} := \frac{\|\overrightarrow{BC}\|_2}{point\_distance} \tag{4.47}$$

$$t_i := \frac{i}{n_{segments}} \tag{4.48}$$

$$new\_point_i := P(t_i) \tag{4.49}$$

We set a desired distance between two points as *point_distance*. Then we calculate the distance between our two way points A and B. The required number $n_{segments}$ of new line segments with the length *point_distance* needed to connect A and B is calculated. The according $n_{segments} + 1$ new way points are created by evaluating the Bézier curve at their respective t_i. Please note that the resulting way points of this method are not necessarily equidistant. Hence, the generated points can have a smaller or larger distance than *point_distance*. We accept this inaccuracy, since this interpolation is not very computationally expensive and *point_distance* can be set as small as needed.

4.7 Geometry of Lanes

After the interpolation step, the courses of all graph edges are smooth. For a CR scenario, however, we do not need the course of roads, but the borders of each lane of the road. Since these were not specified in the OSM scenario, we have to create them from the course of the road and the number of graph lanes of the graph edge. Additionally, we use a table of widths for each type of road, which define the width of each individual lane w_{lane}. Please note that the method described in this section assumes right-hand traffic and must be adapted for left-hand traffic.

As an intermediate step we generate the course instead of the borders for each graph lane. For that, we assume that the specified course of the graph edge c_{edge} is equal to

the center of the road it represents. To generate the course of the graph lanes c_{lane}, we create parallels of c_{edge}. If the number of lanes nr_{lanes} is odd, we use the course ofc_{edge} as way points for the center graph lane. For each other graph lane we create parallels of c_{edge} with distance w_{lane} starting with the neighbors of the center graph lane. If nr_{lanes} is even, we create parallels of c_{edge} with distance $\frac{w_{lane}}{2}$ for the both center lines as depicted in Figure 4.8. After that, we create parallels of the new created parallels with distance w_{lane} for all graph lanes. Then the list way points of each graph lane will be reversed, if that lane is a backward lane, i.e. goes in the opposite direction of the graph edge.

A parallel of a course is created individually for each section. For each point p_i we compute a left p_i^{left} and right p_i^{right} offset point. To do so, we compute the vector v_i to its successor p_{i+1} as the following:

$$v_i = \begin{pmatrix} v_i^x \\ v_i^y \end{pmatrix} := p_{i+1} - p_i \tag{4.50}$$

$$ov_i := \begin{pmatrix} v_i^y \\ -v_i^x \end{pmatrix} \tag{4.51}$$

$$p_i^{left} := ov_i \frac{d_{par}}{\|ov_i\|_2} \tag{4.52}$$

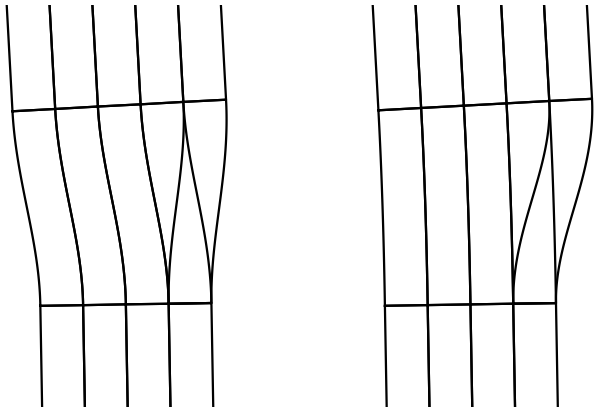
$$p_i^{right} := ov_i \frac{-d_{par}}{\|ov_i\|_2} \tag{4.53}$$

Then we follow it's orthogonal vector ov_i to the left and right the desired distance d_{par}. For the last section n there is no p_{n+1}, hence we choose $v_n := v_{n-1}$. Please note that this method yields no exact curve offset. The results, however, approximate the offset better the more way points are available. So, they can be improved by reducing *point_distance* in the interpolation step described in Section 4.6.

Offsets of Roads

However, we encountered a problem that could be solved more easily before creating the course of each individual graph lane. Since we arrange the graph lanes around their graph edge so that its course is the center, the course of consecutive lanes can jerk, when the number of lanes changes. An example for this can be seen in Figure 4.9. For simplification, we perform offsets only on one way graph edges. To identify cases in which an offset can improve results, we iterate over all graph nodes of degree two or three.

For graph nodes of degree two, we distinguish between two cases. Either the successor has more graph lanes than the predecessor or it has less lanes. If the count of graph lanes is the same for both graph edges, there is no need to perform an offset.



(a) Result without offset (b) Result after offset

Figure 4.9: Example for improved courses of consecutive lanes by offset

In the first case, we offset the successor, in the latter case we offset the predecessor graph edge. The offset distance is defined by the difference between the number of incoming and outgoing graph lanes. Whether the offset is performed to the left or right is determined analogous to the linking described in Section 4.5. If a graph lane at the right has more than one successor, the subsequent graph edge is moved to the right, if the leftmost graph lane has several successors, the subsequent graph edge is moved to the left. If a graph lanes at the left has several predecessors the predecessor is shifted to the left and accordingly for the rightmost graph lane.

For graph nodes with degree of three, we check if two of the graph edges represent two successive roads. To do so, we check if the angle between them at the current node is larger than 0.9π. The angle between two edges at a node is calculated as described in Equation 4.15 and 4.16. If the lane count of these consecutive graph edges is different we offset them as described for nodes of degree two.

An offset of a graph edge is performed on its course, i.e. on the way points generated at the interpolation step. Since we need the offset in each case only at one end of the graph edge, we shift the individual way points accordingly. Having set the offset distance d_{off}, as described above, but positive for a shift to the left and negative for a shift to the right, we iterate over all way points of the graph edge.

$$\delta_i := \begin{cases} d_{off} \cdot \left(1 - \frac{i}{length(waypoints)-1}\right) & \text{if } at\_start \\ d_{off} \cdot \frac{i}{length(waypoints)-1} & \text{else} \end{cases} \tag{4.54}$$

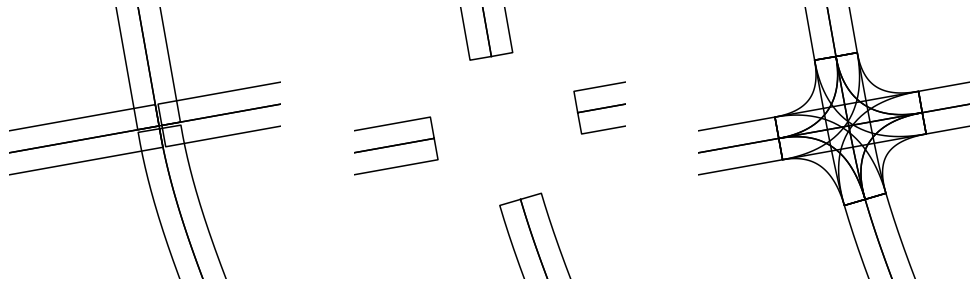
$$p'_i := p_i + \delta_i \cdot \frac{ov_i}{\|ov_i\|_2} \tag{4.55}$$

For each way point p_i at index i, we compute a vector v_i and its orthogonal vector ov_i as described in Equation 4.50 and 4.51. For the last vector v_n we again use $p_{n-1} - p_n$. Then we calculate the offset δ_i for each point. δ_i is linear to i so that the offset has the full desired length d_{off} at one end of the graph edge and is zero at the other end. Since we sometimes want to shift the way points at the start of the graph edge *at_start* and sometimes at the end of it, we distinguish between these cases. For graph edges we want to offset at their end we increase δ_i linearly to i, while we decrease it for edges we want to offset at their start.

After the course of a graph edge is offset like described in the previous paragraph, the graph lanes are created according to their manipulated course. Hence, jerking courses caused by a change of the number of graph lanes like depicted in Figure 4.9 can be avoided.

4.8 Geometry of Intersections

In all previous steps we created graph lanes as lanes of a graph edge. As illustrated in Figure 4.10(a), this is not sufficient. Since graph lanes are derived from OSM ways, they represent only the connection between intersections. The course of lanes on these intersections, however, is not defined yet. But we do have data about the successors of each graph lane. To connect the existing graph lanes, we crop them at intersections to free some space for the connecting lanes, which is depicted in Figure 4.10(b). Graph edges too short to be cropped appropriately will be deleted. Then we create new graph lanes as linking segments for each link between two graph lanes of graph edges as illustrated in Figure 4.10(c).



(a) Intersection before cropping and creating link segments

(b) Intersection with cropped graph edges

(c) Intersection with link segments

Figure 4.10: Geometry of an intersection unchanged, after cropping and with link segments

Cropping

Since we want to create link segments for all graph lanes, we need to free space for them at intersections. Therefore, we crop the ends of graph edges at each graph node. Similarly to the offset, this can be performed more easily by manipulating the course of a graph edge instead of cropping each of its graph lanes individually. Hence, the method described in this subsection is performed before the geometry of each lane is created according to Section 4.7.

To perform cropping on each graph edge at each graph node, we iterate over all graph nodes. If the current graph node V_c has a degree greater than one, we iterate over all its graph edges. Then, for our current graph edge E_c, we iterate over all of its other graph edges. To determine the distance we need to crop E_c, we compute the required distance $d_{edges}(E_c, E_{other})$ to the current other edge E_{other}.

$$width(E) := E_{lane\_nr} \cdot lane\_width \tag{4.56}$$

$$d_{edges}(E_c, E_{other}) := \frac{width(E_c)}{2} + \frac{width(E_{other})}{2} + d_{intersection} \tag{4.57}$$

This distance is defined by the width of both graph edges and a constant value $d_{intersection}$ which can be manually tuned for best results. The width of a graph edge is the product of the width $lane\_width$ of each of its graph lanes and their count $lane\_nr$.

The point to which we want to crop E_c ideally is the nearest point to V_c that has at least a distance of $d_{edges}(E_c, E_{other})$ to each E_{other} at V_c. Since the computation of distances to all points of E_{other} is computationally too expensive, we approximate this point. We do this by iterating over the way points of E_c and E_{other} equally starting at V_c. This means, for each step we increase the index of the point on E_c and E_{other} by one and check the distance of the resulting point pair. If this distance is greater than $d_{edges}(E_c, E_{other})$ we compute the length $l_c(E_{other})$ of the course of E_c to its current point. Having done this for all other edges, we take the maximum of l_c and save it for the current combination of E_c and V_c.

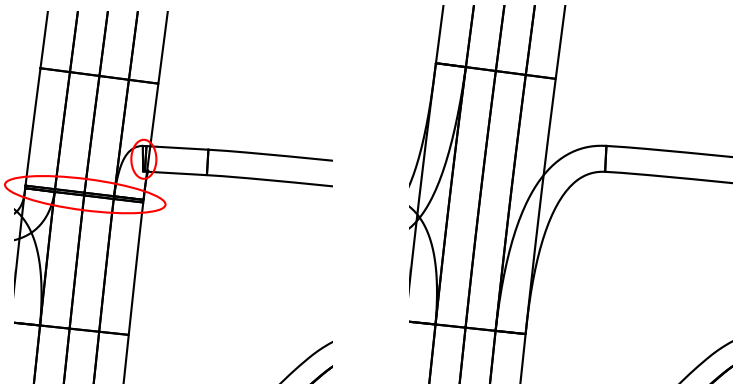
After this is done for all graph edges and graph nodes, we have the distance, we want to crop of each graph edge at each of its ends. We save these distances instead of cropping the edges directly to deal with graph edges that are shorter than their cropping distances should be. To deal with these, we first compute the indices i_{start} and i_{end} of the way points to which we want to crop the graph edge at both ends. If i_{end} is greater than i_{start} we set the new way points for the current edge to all points with an index in the interval $[i_{start}, i_{end})$.

$$i'_{start} := \frac{i_{start} + i_{end}}{2} - 1 \tag{4.58}$$

$$i'_{end} := i'_{start} + 2 \tag{4.59}$$

If i_{end} is smaller than i_{start}, we set them so that the resulting interval contains only two points in the middle of i_{start} and i_{end}. In this case, the corresponding graph edge is saved to a list *edges_to_delete*.

Deletion of Edges



(a) Result without deleting short edges (marked red)

(b) Result after deleting short edges

Figure 4.11: Example of an intersection which cannot be converted properly without deleting short graph edges

Graph edges in the list *edges_to_delete* could not be cropped properly and can therefore cause bad results. As visible in Figure 4.11, they can for example leave too little space to create a suitable course for a turning lane.

When a graph edge is deleted, we need must ensure that all connections traversing the graph lanes of the deleted graph edge are preserved. To do so, we iterate over all graph lanes of the deleted graph edge. Then we take a copy of the successors $succ_i$ and predecessors $pred_i$ of the current graph lane l_i.

$$\forall x \in succ_i : predecessors(x) := (predecessors(x) \cup pred_i\_copy) \backslash l_i \tag{4.60}$$

$$\forall x \in pred_i : successors(x) := (successors(x) \cup succ_i\_copy) \backslash l_i \tag{4.61}$$

Then we define the set of preceding graph lanes for all $succ_i$ so that they contain additionally all predecessors $pred_i\_copy$ of l_i but not l_i itself. Accordingly, we merge the successors of all $pred_i$ with the successors $succ_i\_copy$ of l_i but not l_i. After that, no graph lane is connected with a graph lane of the edge we want to delete anymore. Then the graph edge is removed from the graph structure of the scenario.

Creating Link Segments

To create the courses of lanes on intersections we create so called *link segments*. A link segment represents the drivable space on an intersection between two lanes leading to it. Link segments are stored as regular graph lanes, with the only difference of being not assigned to a graph edge.

Link segments are created for every link between two graph lanes. To do so, we iterate over all graph edges in the graph structure and over all graph lanes at the current graph edges. For each current graph lane l_i we create a link segment $ls_{i,k}$ to each of its successors. In this way, we create exactly one link segment for each link l_i with the successor $l_i\_succ_k$. The predecessors and successors of $ls_{i,k}$ will be defined as $\{l_i\}$ and $\{_i\_succ_k\}$ respectively. The width of $ls_{i,k}$ will be set to the width of l_i at the start and to the width of $l_i\_succ_k$ at the end.

We approximate the course of $ls_{i,k}$ by a new Bézier curve which will be tangential to its predecessors' and successors' course. Similarly to the interpolation step in Section 4.6, we have to define the control points of the Bézier curve. The ends of the curve p_1 and p_4 are defined as the last way point of l_i and the first way point of $l_i\_succ_k$ respectively. Since Bézier curves are always located in the convex hull of their control points [7], we use quadratic Bézier curves if possible. This avoids the course of the link segment to veer in the opposed direction of p_4. To find the second control point $p_{intersection}$ for a quadratic Bézier curve, we compute the tangential vectors v_1 and v_2 to the start and end of the link segment.

$$v_1 := waypoints(l_i)[n] - waypoints(l_i)[n-1] \tag{4.62}$$

$$v_2 := waypoints(l_i\_succ_k)[1] - waypoints(l_i\_succ_k)[0] \tag{4.63}$$

$p_{intersection}$ will be defined as the point on which l_i and $l_i\_succ_k$ would intersect if their course was extended straightly at their ends. This point solves the following equations:

$$p_{intersection} = p_1 + a_1 \cdot v_1 = p_4 + a_2 \cdot v_2 \text{ where } a_1 > 0 \text{ and } a_2 > 0 \tag{4.64}$$

If this constrained linear system of equations has a solution, we use $[p_1, p_{intersection}, p_4]$ as control points for the Bézier curve. Otherwise, we create two intermediate control points similarly to the interpolation step in Section 4.6 to create a cubic Bézier curve.

$$p_2 := p_1 + v_1 \cdot \frac{d}{\|v_1\|_2} \tag{4.65}$$

$$p_3 := p_4 + v_2 \cdot \frac{d}{\|v_2\|_2} \tag{4.66}$$

d is the same constant as in 4.6, which can be manually tuned for best results. The control points for the cubic Bézier curve will be $[p_1, p_2, p_3, p_4]$. Just as in Section 4.6 the

above defined Bézier curve will be evaluated at several points to create a list of n way points. This evaluation is performed exactly as described in the equations 4.47, 4.48, and 4.49.

To prevent the creation of unnecessary link segment, $ls_{i,k}$ will only be added to the graph if l_i and $l_i\_succ_k$ do not belong to the same graph edge and if its course is not curved too strongly. When l_i and $l_i\_succ_k$ belong to the same graph edge, a link segment between the two of them would create a U-turn on the intersection, which is not desired. The curvature of $ls_{i,k}$ is defined by the angle of the vectors v_1 and v_2. This angle needs to be above a certain threshold which can be manually tuned for best result.

After the creation of all link segments we need update the predecessors and successors of all other graph lanes. These are still set to the graph lanes at the next graph edges they are liked to. Now we want to set them to the link segments they are actually adjacent with. To do this, we first delete the sets of all predecessors and successors of all graph lanes assigned to a graph edge and replace them by an empty set. Then we iterate over the link segments. For every current link segment ls_i, we take its successor s_i and its predecessor p_i. We add ls_i to the set of successors of p_i and to the set of predecessors of s_i.

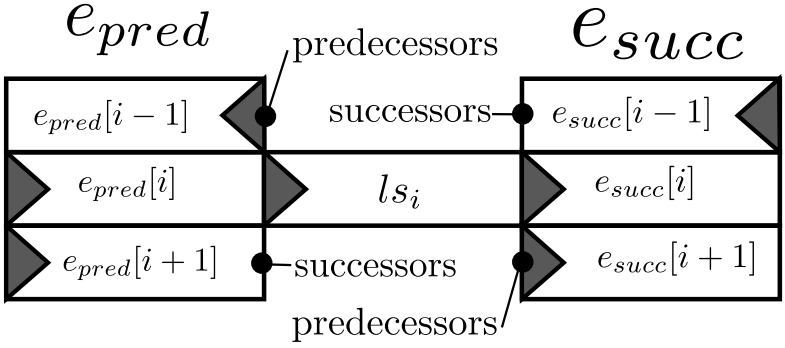


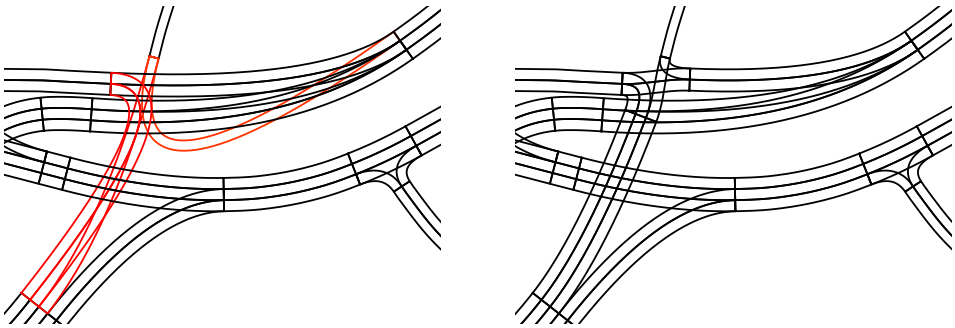
Figure 4.12: Structural visualization of a generic lane segment

In addition to the successors and predecessors of graph lanes we want to set the adjacent left and right neighbors of the link segments. These are determined by checking the neighbors of predecessor $e_{pred}[i]$ and successor $e_{succ}[i]$ of each links segment ls_i. First we get the left $e_{pred}[i-1]$ and right $e_{pred}[i+1]$ adjacent of $e_{pred}[i]$. Please note that this notation does not mean, that these neighbors have necessarily the index $[i+1]$ or $[i-1]$ at the list of graph lanes of the edge of $e_{pred}[i]$. Also, these neighbors can be both directed the same or the opposite to $e_{pred}[i]$. We choose potential adjacent link segments p_sg_i of ls_i from the successors or predecessors of $e_{pred}[i-1]$ and $e_{pred}[i+1]$. If a neighbor has the same direction as $e_{pred}[i]$, we add its successors to p_sg_i, otherwise

we add its predecessors to p_sg_i. Then we check the other ends of all potential adjacent link segments p_sg_i. Each current lane segment p_sg_c in p_sg_i is either a successor or a predecessor of a neighbor of $e_{pred}[i]$. If it is a successor, we check if its successor is a neighbor of the successor $e_{succ}[i]$ of ls_i. Accordingly, if it is a predecessor, we check if its predecessor is a neighbor of $e_{succ}[i]$. If that is the case, we have found an adjacent lane segment. That is because this lane segment is consecutive to both, a neighbor of the successor of ls_i and a neighbor of the predecessor of ls_i. This lane segment is set to the adjacent right of ls_i if it is consecutive to $e_{pred}[i+1]$ and $e_{succ}[i+1]$, and to the adjacent left of ls_i if it is consecutive to $e_{pred}[i-1]$ and $e_{succ}[i-1]$. Now we can set the properties *adjacent_left* and *adjacent_right*. Additionally we set *adjacent_left_direction_-equal adjacent_right_direction_equal* to *True* if the direction of the respective link segment is the same as the direction of ls_i and to *False* otherwise.

After these steps, we have created a link segment for each connection of two graph lanes. We have set the successors and predecessors of all graph lanes including the link segments correctly. And additionally, we have set the adjacent graph lanes of the link segments.

Clustering of Link Segments



(a) Result without clustering link segments, erroneous lane segments marked in red

(b) Result after clustering link segments

Figure 4.13: Example scenario benefiting from segment clustering

In scenarios with several consecutive short graph edges, link segments can become undesirable long. Adjacent link segments, that should run parallel for a certain section, can then intersect, which results in chaotic courses for lane segments. An example for this is depicted in Figure 4.13. To prevent this, we group link segments leading to the same and let them run together until a certain point.

First, we need to find potential groups of link segments to cluster. We find these groups by iterating over all link segments. For every current segment ls_i, we check if it connects two different graph nodes. This can be determined by comparing the respective nodes of the graph edge the successor and predecessor of ls_i are assigned to. The link segments are grouped by each pair of nodes they are connecting. Since we search for link segments with a long track, we filter out all link segments, which are shorter than a certain threshold $d^1_{cluster}$. For each group we divide the link segments into subsets. Each subset connects two graph edges. We also keep a list of edges e_i which are connected by these subsets in the current group g_i.

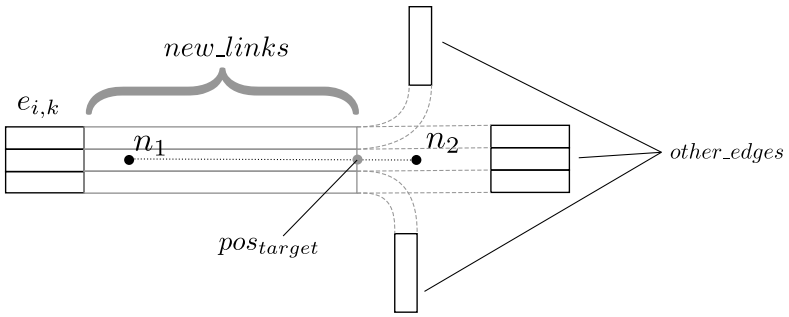


Figure 4.14: Structural visualization of a generic cluster of lanes

Then we iterate over e_i. If the number of subsets assigned to the current edge $e_{i,k}$ is greater than one, this edge is connected to more than one other edge, in the current group. If all link segments $ls_{cluster}$ of these subsets are adjacent, they will be grouped to a cluster. This means that they will lead as an adjacent cluster from $e_{i,k}$ in the direction of their other end. To do this we create new link segments *new_links*. These link segments extend graph lanes of $e_{i,k}$, which are consecutive to graph lanes in $ls_{cluster}$. Each link segment in *new_links* connects a graph lane on $e_{i,k}$ with a point near the *target* of the cluster. This *target* pos_{target} is defined on the line segments between the two graph nodes of the current group g_i. The node at $e_{i,k}$ is n_1, the node located at the target is n_2. The distance between pos_{target} and n_2 is defined by the maximum of $d_{intersection}$ and the maximal distance the start of a graph edge assigned to n_2 has to it. $d_{intersection}$ is a constant introduced in the subsection **Cropping** in this section. If the distance $\|pos_{start} - pos_{target}\|_2$ between the start and the target point is smaller than a certain threshold $d^2_{cluster}$, we discard the current cluster. This value $d^2_{cluster}$ can also be manually adjusted.

After the start point pos_{start} and pos_{target} are defined, we need to specify the course of the link segments *new_links*. To do so, we compute the central course of all link segments in *new_links*. We model this course as a quadratic Bézier curve which

starts at the last way point of $e_{i,k}$ and ends at pos_{target}. The intermediate control point is set, similarly to Section 4.6, tangential to the end of $e_{i,k}$ with a distance of $d \cdot \|pos_{start} - pos_{target}\|_2$. d is a constant defined in Section 4.6 which can be manually tuned in the interval $[0, 0.5]$.

Based on this central course, the courses of *new_links* are created. This is done by offsetting the central course as described in Section 4.7. Also, we need to readjust the link segments of $ls_{cluster}$. Their courses now start at then end of *new_links* and are defined as described in the paragraph **Creating Link Segments** in this section. Their predecessors or successors are set to the link segments of *new_links* respectively. Also, the predecessors and successors of $e_{i,k}$ and *new_links* have to be updated.

After this step, we have clustered link segments connecting two different graph nodes over a large distance leading to several other graph edges.

After the geometry of each intersection is defined by the previous steps, nearly all information needed for deriving a CR scenario is stored in our graph structure.

4.9 Meeting CommonRoad Specifications

To derive a CR scenario from our graph structure, we have to perform few more steps. As we have generated the tracks for all graph lanes in the graph structure, we need to derive the borders of each graph lane to convert it into a lanelet. These borders must be adjusted so that adjacent graph lanes use the same points for their common borders to meet the CR specifications as described in Section 3.2. Also, we want to create a contiguous road network.

Creating Borders of Graph Lanes

To generate a CR scenario, we need to represent the graph structure by lanelets. We derive lanelets from the graph lanes in our graph structure. For a lanelet, we need to extract several features of graph lanes such as an unique id, the successor and predecessors, the right and left adjacent lanelets, and the speed limit. Most importantly, we need to extract the bounds of all graph lanes.

To do so, we have to note, that the border between two lanelets in a CR scenario must be defined by the same way points for both lanelets. Also, the left and the right border of a lanelet must have the same number of way points. To meet this specification, we first set the number of way points equal for all adjacent graph lanes. This number is defined as the minimum of the count of way points in each graph lane. If the number of way points is reduced to n for a certain graph lane, its new way points are set as follows: The first and last way points stay the same. The remaining $n - 2$ way points are set to be equidistant on the course of the previous way points.

After readjusting the way points, we can derive the borders of each graph lane. To do this, we check for each graph lane if the respective border is already defined by an adjacent graph lane. If that is the case we just adopt it or, if the adjacent graph lane has the opposite direction, we adopt it in reversed order. Otherwise, we create a new border as an offset of the course of the lane. As our graph lanes have their width defined by two values w_1 and w_2, one for the start and one for the end, we need to offset the course of the lane by different distances at each point. We do this similarly to the offset step described in Section 4.7 by defining the vector v_i for each of the n sections between two way points p_i at index i and p_{i+1} like in equation 4.50. Then we compute its orthogonal vector as in equation 4.51.

$$d_i := w_1 \cdot \left(1 - \frac{i}{n}\right) + w_2 \cdot \frac{i}{n} \tag{4.67}$$

$$p_i^{left} := ov_i \cdot \frac{d_i}{\|ov_i\|_2} \tag{4.68}$$

$$p_i^{right} := ov_i \cdot \frac{-d_i}{\|ov_i\|_2} \tag{4.69}$$

The points on the left and right border p_i^{left} and p_i^{left} placed with an individual distance d_i to p_i. d_i is an affine combination of w_1 and w_2 where w_1 is weighted linearly less with increasing i. This allows a smooth transition between two different widths at the start and the end of a graph lane. The last points of the borders p_n^{left} and p_n^{right} have to be constructed with $v_n := p_n - p_{n-1}$, since the way point at index $n+1$ does not exist.

Adjusting Common Border Points for Several Graph Lanes

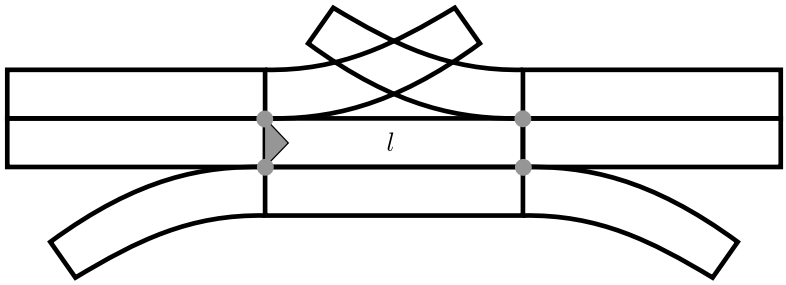


Figure 4.15: Visualization of points shared by several graph lanes

The borders created in the previous step are only created once for two adjacent graph lanes. Hence, the points defining the common border of two adjacent lanes are equal for each graph lane. However, this does not hold for two consecutive graph lanes. To

set the last border points of a lane equal to the first border points of its successors, we have to note the following: If we change one border point, we also need to change it for all other graph lanes sharing the same point. Figure 4.15 depicts points shared by various graph lanes.

$$p_{center} := \frac{1}{n} \sum_{i=0}^{n-1} p_i \tag{4.70}$$

To set these points equal for those n graph lanes we take the centroid p_{center} of the positions of these points p_i, which can differ for different graph lanes. Then we set the positions for all p_i to p_{center}. After this step we can derive a CR scenario from the graph structure.

Making the Graph Connected

The OSM map converted by this method is defined my a OSM file. As this OSM file can hold an unspecified subset of the map date of the whole world, we provide a method to convert only a connected subset of the road network in that file. This method chooses the graph node n_{center} nearest to the center of the area covered by the file as root of the connected graph.

$$n_{center} := \arg\min_{n_i \in N} \|n_i\|_2 \tag{4.71}$$

N is the set of all graph nodes in the graph structure. As the origin of coordinates after the projection step described in Section 4.2 is the center of the scenario, n_{center} is set to the graph node with the lowest magnitude. All graph nodes and connected to n_{center} by graph edges and all these graph edges connecting them are kept in the graph structure, all remaining graph nodes and edges are removed from the graph structure. The connected elements of the graph can be found by performing a depth first graph search algorithm on the graph structure starting with n_{center}. Since this step can remove elements from the graph, we perform it right after the creation of the graph structure to spare computing time for possibly removed elements.

Exporting the CommonRoad Scenario

After we created a graph structure based on an OSM file and performed all steps described above on it, we can export a CR scenario. To do this, we convert each graph lane to a lanelet and add it to the scenario. A graph lane can be converted to a lanelet by creating a new lanelet and setting its properties according to the graph lane. These properties comprise an unique id, the bounds and course of the lane, the applying

speed limit, and the ids of adjacent and consecutive lanes. The tools provided by CommonRoad allow to easily create lanelets and scenarios. The resulting scenario can then be saved to disk in a XML format.

5 Implementation

We implemented the approach described in the previous chapter in a python tool. This chapter will introduce the tool. The first section will explain how to use the tool. Then the available methods of the tool will be described. Afterwards, an overview of all parameters is given and finally, we introduce the possibility to improve results of the tool by modifying OSM Files.

5.1 Using the Tool

There are several requirements to be able to run the tool. It is written in Python 3 and requires at least Python 3.6 to run. Also, it requires the packages `numpy` and `scipy`. Additionally, it uses the CommonRoad Python Tools, which are part of the CommonRoad repository[1]. The CR Python Tools themselves require the package `shapely`.

The tool consists of a folder `osm2cr` in which you can find the executable python file `main.py` and two folders `files` and `converter`. To import the CR Python Tools, the path must be set in `converter/config.py`.

```
CR_TOOLS_PATH = 'C:/Users/Maxim/commonroad/tools/Python/'
```

With these prerequisites installed, the tool can be executed. One can provide the input for the conversion in two ways. Firstly one can provide a OSM file which will be opened converted as a whole. To do so, one has to execute `main.py` with the argument `o` to open a file and additionally with the file to open.

```
$ python main.py o map.osm
```

The other way is to set the coordinates and the size of the desired area in `config.py` and let the tool download the map automatically.

```
DOWNLOAD_EDGE_LENGTH = 100
DOWNLOAD_COORDINATES = 48.137927, 11.565442
```

[1] `gitlab.com/commonroad/commonroad.gitlab.io`

DOWNLOAD_EDGE_LENGTH specifies the radius in meters of the area around the provided coordinates which will be downloaded. DOWNLOAD_COORDINATES holds the coordinates of the point around which the map will be downloaded in latitude and longitude. The values assigned to it are latitude and longitude separated by a comma. The map will be downloaded to the files folder.

```
$ python main.py d
```

To start the process with downloading a map, execute main.py with the argument d.

Either way the tool reads the OSM data, exports a CR scenario and then displays the scenario in an interactive plot. The exported CR scenario will be saved in the files folder. To change this and several other settings such as the name of the benchmark, one can adjust the parameters in config.py as described in the following section.

5.2 Parameters

There are many parameters in config.py to furthermore adjust the tool.

```
SAVE_PATH = 'files/'
```

With the parameter SAVE_PATH the location of downloaded and exported files is specified. By default it is set to files/.

```
CR_TOOLS_PATH = 'C:/Users/Maxim/commonroad/tools/Python/'
```

As described in the previous section CR_TOOLS_PATH needs to be set to the CR Python Tools to make this tool work.

```
DOWNLOAD_EDGE_LENGTH = 100
DOWNLOAD_COORDINATES = 48.137927, 11.565442
```

These parameters are only important when a map is downloaded. Then DOWNLOAD_COORDINATES specify the point around which the OSM map will be downloaded. DOWNLOAD_EDGE_LENGTH specifies the maximal horizontal and vertical distance an OSM object can have to be on the downloaded map. OSM ways will be downloaded if one of their OSM nodes is in that area. Therefore, the resulting map often includes roads outside of the area and its center can be shifted.

```
BENCHMARK_ID = 'Testbench_01'
AUTHOR = 'Automated converter by Maximilian Rieger'
AFFILIATION = 'Technical University of Munich, Germany'
SOURCE = 'OpenStreetMaps (OSM)'
TAGS = '...'
```

```
TIMESTEPSIZE = 0.1
```

These parameters are important for the header of the resulting CR scenario. They can be freely changed to the users desires. `BENCHMARK_ID` also determines the filenames of downloaded maps and exported CR scenarios. A downloaded map will be named by `BENCHMARK_ID+'_downloaded.osm'`, which evaluates in this example to `Testbench-01_donwloaded.osm`. Resulting CR scenarios will be named by `BENCHMARK_ID+'.xml'`, i.e. `Testbench-01.xml` in this example.

```
EARTH_RADIUS = 6371000
```

For the projection step described in Section 4.2, the radius of the earth is required. The default value can ba adjusted if needed for regions closer or further from the equator. However, in most cases, this should not be necessary.

```
SOFT_ANGLE_THRESHOLD = 55
```

`SOFT_ANGLE_THRESHOLD` is the threshold in degrees used in Section 4.5 to determine whether two graph edges should be linked or not if the turn lane tags provide no information about that. Setting this value to zero will allow all graph edges to interconnect in this case.

```
INTERPOLATION_DISTANCE = 0.25
BEZIER_PARAMETER = 0.35
```

In Section 4.6 we used two parameters to specify the interpolation process. `INTERPOLATION_DISTANCE` sets the approximate distance between two way points after the interpolation step. Increasing this distance too much can cause erroneous courses of roads near intersections. A distance above 1 meter is recommended. `BEZIER_PARAMETER` affects the control points for cubic Bézier curves in the interpolation step. This value must be in the interval $[0, 0.5]$. The higher it is set, the more sweeping the resulting curves become.

```
INTERSECTION_DISTANCE = 5
CLUSTER_LENGTH = 10
LEAST_CLUSTER_LENGTH = 10
LANE_SEGMENT_ANGLE = 30
```

As described in Section 4.8, there are several parameters specifying the geometry of link segments at intersections. `INTERSECTION_DISTANCE` specifies the distance two roads should minimally have, which is used in the cropping step. `CLUSTER_LENGTH` sets the minimal distance two graph nodes must have so that clustering of link segments between them will be performed. `LEAST_CLUSTER_LENGTH` sets the minimal length a

cluster of link segments needs to have to be added to the graph structure. Finally LANE_SEGMENT_ANGLE is the minimal angle between the ends a link segment may have to be added to the graph structure. Setting this to zero may cause odd U-turns at complex intersections, setting it too high will prevent the creation of any link segment.

```
DELETE_SHORT_EDGES = True
```

In some cases one might want to prevent graph edges from being deleted. Then DELETE_SHORT_EDGES can be set to False.

```
LOAD_TUNNELS = False
```

For some cases one might to want include or exclude tunnels from the scenario. As CR does not support any information for the third dimension, tunnels can intersect in an undesired manner with other roads. To exclude tunnels, set LOAD_TUNNELS to False, else to True.

```
ACCEPTED_HIGHWAYS = ['motorway', 'motorway_link']
```

The roads extracted from an OSM file can be filtered by their types. All types contained in this list will be extracted. Recommended types are: 'motorway', 'trunk', 'primary', 'secondary', 'tertiary', 'unclassified', 'residential', 'service', 'motorway_link', 'trunk_link', 'primary_link', 'secondary_link', 'tertiary_link', 'living_street'.

```
LANECOUNTS = {'motorway': 6, 'trunk': 4, ...}
```

LANECOUNTS is a dictionary which specifies the count of lanes for each type of road. Please note, that these numbers will be divided by two if the respective road is a one-way street. The dictionary must hold values for all types in ACCEPTED_HIGHWAYS. We recommend to only use values greater than 0.

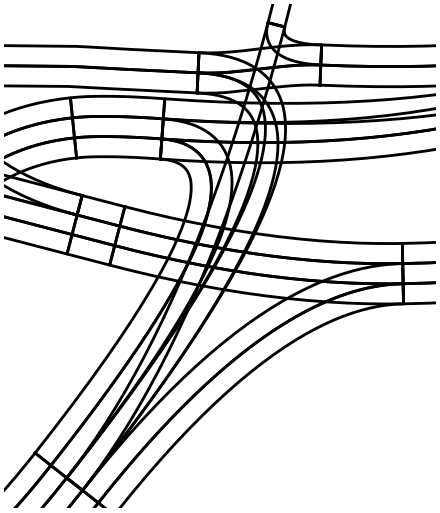
```
LANEWIDTHS = {'motorway': 3.2, 'trunk': 3.2, ...}
```

LANEWIDTHS is a dictionary similar to LANECOUNTS and specifies the width for lanes for each type of roads. Values must be set for all types introduced in ACCEPTED_HIGHWAYS.

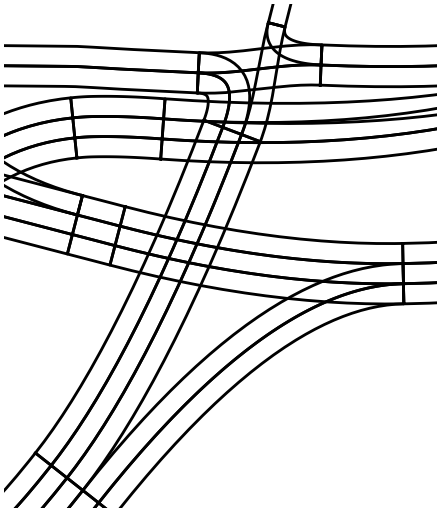
5.3 Improvements in OSM Files

In some cases, the results of the tool can be improved by minor modifications of the underlying OSM file. This can happen when the provided information of an OSM file is simply wrong. E.g. sometimes streets are split in multiple short OSM ways on which the number of lanes is not always set. Other cases in which this can help are complex

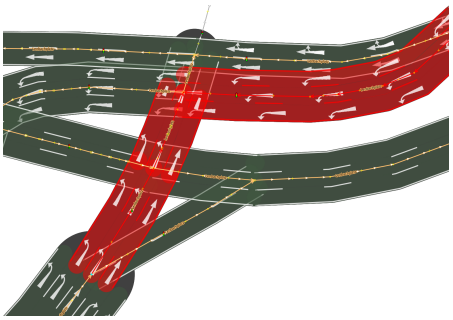
intersections where the turn lane tags are set unsuitable for our tool. By only changing the number of lanes and some turn lane tags, we could successfully convert a complex scenario near Karlsplatz, Munich, which is depicted in Figure 5.1.



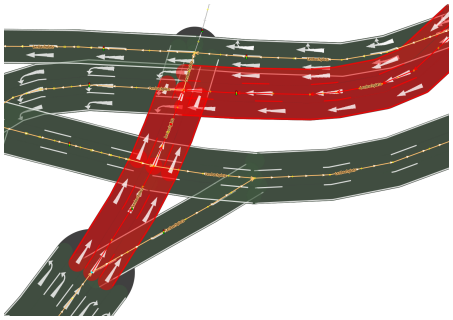
(a) Result using the original OSM data



(b) Result after modifying the OSM file



(c) Screen shot of the original OSM data visualized in JOSM



(d) Screen shot of the modified OSM file visualized in JOSM

Figure 5.1: Example for improvements by modifying OSM file

As visible in 5.1 (c) and (d), we only needed to change the turn lane tags for the red marked OSM ways to `through`. The lanes of these ways head straight across the following intersections in reality, but the tags represent the turn lanes for an upcoming intersection.

Please note that these modifications are only useful for this tool and should not be added to the OSM database.

6 Evaluation

As the implemented tool relies on incomplete information for the conversion, the resulting scenarios do not represent the reality perfectly. We also made several approximations and estimations during the process. In this chapter we will show several examples of conversions and evaluate them. Then the possibilities and limitations of the tool will be pointed out. Finally, a short overview of the run time will be given.

6.1 Examples

Parking Lot As a first example we convert a parking lot of the Technical University Munich with default parameters and unmodified OSM data. The resulting scenario

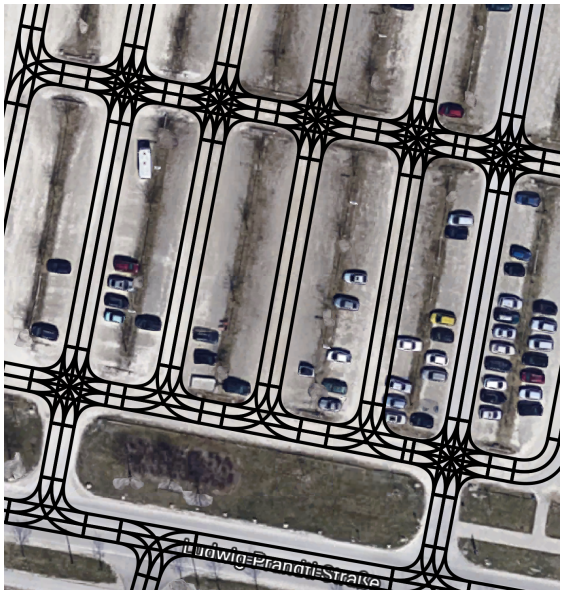


Figure 6.1: Scenario of a parking lot with a satellite image [1]in the background (48.262272, 11.663661)

[1]Satellite image by Google Maps: `maps.google.com`

shows smooth and consistent courses for roads as well as for intersections. As there is no information given for number or interconnection of lanes all values are automatically assumed by the tool. In Figure 6.1 it is also visible, that the created lanelets match the real street well. The geographic coordinates of the scenario are mentioned in the caption of Figure 6.2.

Freeway Access In this example we convert a freeway access. As visible in Figure

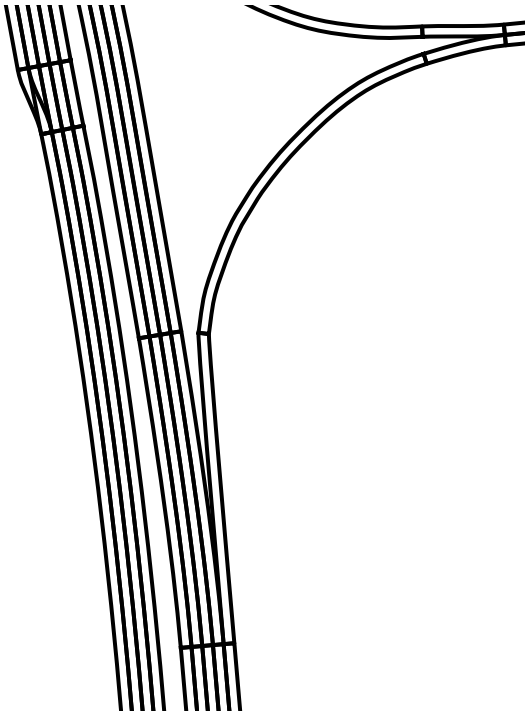
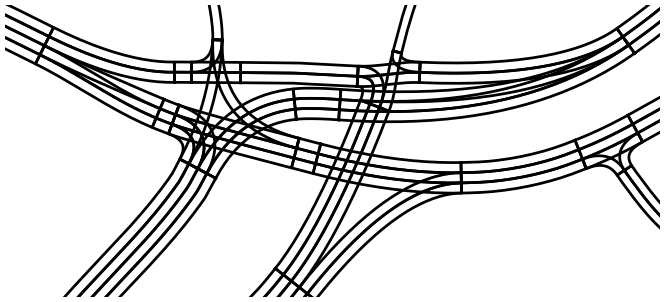


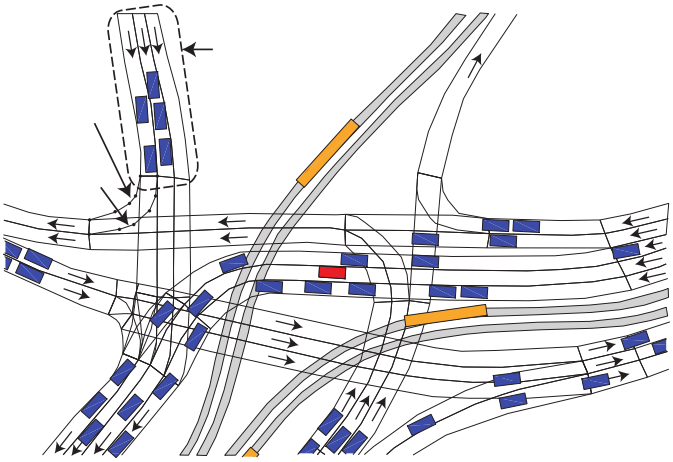
Figure 6.2: Scenario of a freeway access (48.262909, 11.647058)

6.2, the result is adequate. One can clearly see the lanelets passing straight through, even if the count of lanes changes between two successive street segments. This is made possible by the offset of roads described in Section 4.7.

Complex Intersections near Karlsplatz Munich For this example we converted a region in the center of Munich. The OSM data used for the results in Figure 6.3 (a) has been modified as described in Section 5.3. For comparison, there is a hand crafted scenario of the same location in Figure 6.3 (b). There are several differences to notice: First, the tool presented in this work does only convert the road network



(a) Generated scenario



(b) Hand crafted scenario [3]

Figure 6.3: Scenario near Karlsplatz Munich (48.140535, 11.566745)

and does not create obstacles or tram tracks. Second, the bounds of lanelets at the hand crafted scenario are specified by much less way points. An advantage of the automated generation of a scenario is that the number of points evaluated can be increased as desired without causing considerable additional expense. Therefore, the generated lanelets look smoother than hand crafted ones. At the intersection at the lower left of the graphic another difference is visible: This intersection has a degree of five, which means that the simplified linking pattern described in Section 4.5.4 creates its interconnections. Therefore, lanes are falsely interconnected. Also the count of lanes differs between the examples. This is the case because the OSM data does not store the number of lanes heading down from the upper left. Also the hand crafted scenario omits a lane existing in reality at the left. To compare the scenarios to reality there is a satellite image of this region in Figure 6.4. The generated scenario differs from reality at several points, but provides a realistic road network.

Figure 6.4: Satellite image of intersections near Karlsplatz Munich (48.140535, 11.566745)[2]

Problematic intersection Close to the parking lot mentioned above there is an intersection which is a harder problem for the tool. The resulting scenario is shown

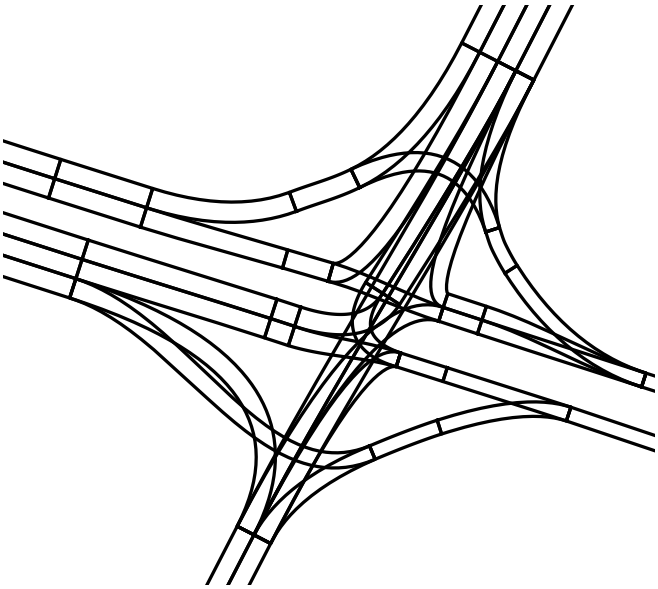


Figure 6.5: Scenario of a intersection (48.262601, 11.658168)

in Figure 6.5. The street on this intersection going from west to east (left to right) is represented by two distinct OSM ways. Therefore, there are two graph nodes of degree four in the center of the intersection. In fact, the intersection is composed of eight intersections in the OSM representation. Each of those is marked with a red

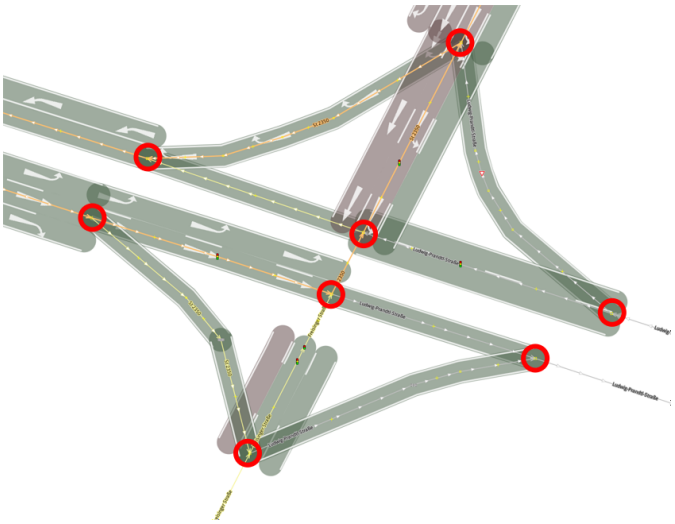


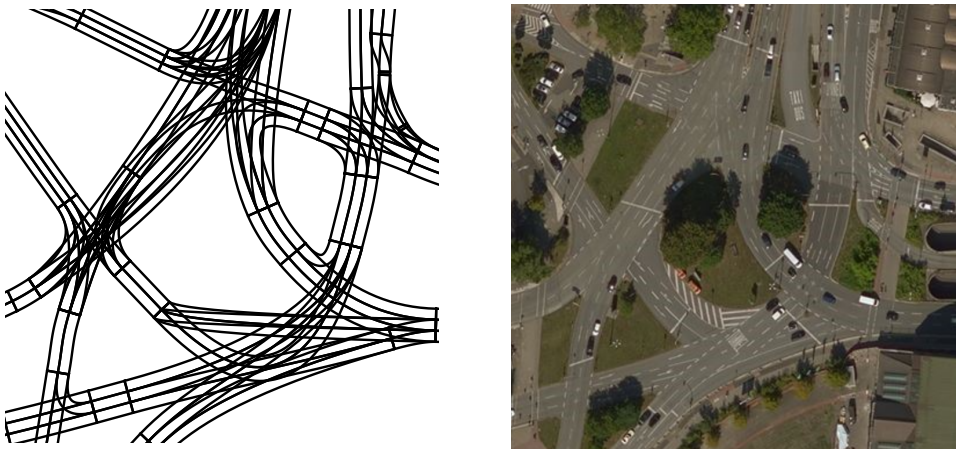
Figure 6.6: OSM representation of the intersection

circle in Figure 6.6. The center graph nodes are very close together, so the graph edge connecting the sa well as other graph edges are deleted. Now, link segments which should only be in the center of the intersection start far away from their desired location. This problem should be taken care of by the clustering step described in Section 4.8. In this case, however, the clustering does not apply. This happens, because the link segments starting next to each other end at different graph nodes.

This results in multiple lanelets intersecting each other at positions where a single lanelet could represent the road. Also the courses of central turning lanes are unpleasant. Additionally, there are two lanelets at the north and south (top and bottom) of the intersection, which resemble a inexistent connection. These can be easily prevented by modifying the OSM file so that the possible successors of these lanes are specified.

Complex Intersections near Deichtorplatz Hamburg One of the most complex intersections in Germany is at Deichtorplatz in Hamburg. As shown in Figure 6.7 (a) the resulting scenario contains multiple confusing link segments which overlap in many points. This is the case, because there are many intersections with high degrees close together. Also the turn lane tags are not set for all lanes in the OSM data. However, adding these turn lane tags to get better results does not fix the main problems, which are intersection with high degrees located close together. This example overextends the capabilities of the provided tool.

[2]Satellite images by Bing Maps: `www.bing.com/maps/`



(a) Resulting scenario (b) Satellite image

Figure 6.7: Scenario at Deichtorplatz Hamburg (53.547980, 10.005660)

6.2 Possibilities

As visible in the examples shown above, the tool implemented in this work can convert OSM maps of real road networks to lanelets and save them in a CR scenario. The resulting lanelets have a smooth course, regardless the low accuracy of OSM maps. At intersections there are lanelets created to interconnect lanelets of roads. It is possible to generate realistic scenarios based on the data provided by OSM for most traffic scenarios. This includes intersections of a degree up to four and simple intersections of a higher degree. The results can also be improved by various parameters, which can be optimized for every conversion.

6.3 Limitations

The provided tool creates realistic scenarios based on OSM data. However, it cannot perfectly resemble the reality. Since OSM data does not provide all information necessary to describe the road network, the results of the tool deviate from the reality. The tool also does not handle complex road networks perfectly. It fails to create realistic scenarios if intersections have a high degree or are too close together.

Also the implementation brings some limitations. Since the borders of lanelets are generated based on the course of a lane, they can be erroneous when the parameter for the interpolation distance is set to high. Thus, creating scenarios with few points can have a negative impact on their quality. As an OSM file is taken is as input, the tool is

not aware of roads not included in that file. Therefore, intersections at the outer edge of the cutout can be processed erroneously. To avoid this problem, one can simply convert a sightly larger area, so that the important region is not located at the outer edge. Also the tool is only suitable for roads with right hand traffic. Using it for regions with left hand traffic will cause lanes at the wrong side of roads and intersecting at some situations. Furthermore, the course of roads and therefore also the course of lanelets depends on the way points of OSM files. Although the tool smooths the path between these way points, it still traverses each of them. Thus, outliers can cause jerking courses in the course of the resulting lanelets.

6.4 Run Time

The implemented tool runs in Python and is not parallelized. Therefore, its run time for a given input only depends on single thread CPU performance and disk speed. Of course the run time depends primarily on the size of the converted map and the number of streets within that area. The distance of interpolated way points also effects the run time, since it determines the number of points created, processed, and saved in the process. All implemented methods performed in the tool run in linear time complexity, making the tool scalable for larger areas. However, we also used libraries such as Pythons XML implementation, which potentially have a higher time complexity. Also the linear time complexity of the implemented methods does only hold under the assumption that the degree of intersections is bounded above by a non infinite number.

We encountered run times of few seconds for most of our conversions. To test the runtime we converted larger areas from $10000m^2 (= 200m \times 200m)$ to $16000000m^2 (= 4000m \times 4000m)$. Although conversions of this size are possible, for most use cases areas below $500m \times 500m$ will probably be large enough. The run times of these conversions can be seen in Figure 6.8. These results indicate a almost linear time complexity.

All tests were performed on a laptop with an Intel Core i7 6650U processor at up to 3.4 GHz and 16 GB RAM running Windows 10. Meaningful tests with larger areas could not be performed, because the memory consumption of the tool would exceed the available RAM.

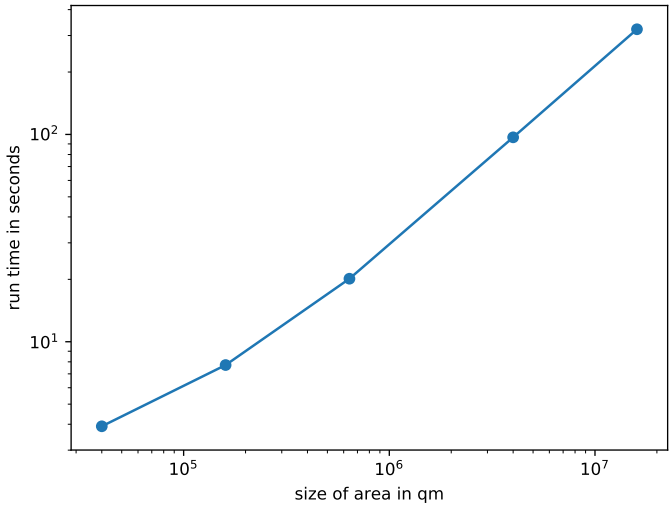


Figure 6.8: Diagram of run time given different sizes of maps, axes scaled logarithmically

7 Conclusion

The main goal of this thesis was to automatically convert *OpenStreetMap* maps to a lanelet format and save them to a *CommonRoad* scenario. This would allow to use the large amount of available OSM maps for testing purposes.

To do so, we converted OSM maps to a graph-like structure. In this structure edges represent roads and nodes represent intersections. This structure is modified in several steps, iteratively approaching a lanelet structure. First, lanes are created for each road. These lanes are interconnected at intersections. Then the spatial course of lanes on roads and intersections is estimated. Each lane segment is then converted to a lanelet. Finally the structure can be exported to a *CommonRoad* scenario.

We presented a tool implementing this conversion. Since these formats are fundamentally different with regard to the stored information, this procedure can result in erroneous outputs in some cases. While simpler scenarios are reliably converted, results often deviate considerably from reality for complicated traffic flows. As OSM maps lack detailed spatial information of individual lanes, the resulting scenarios cannot resemble reality perfectly. They do, however, pose realistic scenarios which can be used for testing purposes. The tool we implemented in this work performs this conversion within seconds for areas with a size of few hectares.

In the future, this tool could be extended by methods that handle more complex scenarios in a more effective way. Furthermore, it could be used as a basis to automatically generate path planning problems.

8 Future Work

In this work we provided a tool that can convert OSM files to CR scenarios. However, there are several ways to improve or extend this program.

As described in Chapter 6, there are situations in which the tool does not generate realistic scenarios. Complex intersections or short edges can cause erroneous results. To improve the programs behavior, it could be helpful to modify the clustering process described in Section 4.8. This implementation clusters only lane links leading to the same graph node. Clustering lane links based on the direction they are leading to rather than on the graph node they are leading to, could improve this. However, we do not know if this will fix these problems for the general case. To handle very complex scenarios, it could also be helpful to create a graphical editor which allows the user to adjust the graph structure by hand. Another improvement could be done by taking not only turn lane tags but also OSM `restrictions`[1] into account. This additional information could improve the quality of the lane linking process, since it describes turn restrictions for junctions.

Additionally, the tool does not set the adjacent lanelets as recommended by CommonRoad in every case. Lanelets which are intersection for a certain section and then separate from each other, should be split and set adjacent so that two lanelets are adjacent just when a vehicle can change between them. Also, the program is not capable of converting regions with left hand traffic correctly, which could be added. Without a large amount of work, it should also be possible to convert not only roads but also tram tracks. An interesting aspect which could become relevant in the future is a new feature proposed for OSM. It is called *Lane Connectivity*[2] and it is meant to define the connection between two lanes. If this new relation was used for OSM maps, the whole linking process described in Section 4.5 would become trivial. However, as this is only a proposed feature, it is not certain when and if at all it will be implemented.

It would also be conceivable to extend the tool by a automatic traffic generator, which places obstacles on the generated lanelets with a realistic behavior. This way one could not only create a road network based on OSM maps but full CommonRoad benchmarks, which could be used for testing purposes.

[1] `wiki.openstreetmap.org/wiki/Relation:restriction`

[2] `wiki.openstreetmap.org/wiki/Proposed_features/lanes_General_Extension/ProposalPreVoting`

List of Figures

Bibliography

[1] S. Kammel et al. Jan. 2009.

[2] C. Urmson et al. "Autonomous Driving in Urban Environments: Boss and the Urban Challenge." In: 25 (Jan. 2008), pp. 425–466.

[3] M. Althoff, M. Koschi, and S. Manzinger. "CommonRoad: Composable benchmarks for motion planning on roads." In: *2017 IEEE Intelligent Vehicles Symposium (IV)*. 2017, pp. 719–726.

[4] M. Althoff, S. Urban, and M. Koschi. "Automatic Conversion of Road Networks from OpenDRIVE to Lanelets." In: 2018.

[5] M. S. M. Dupuis and H. Grezlikowski. "OpenDRIVE 2010 and beyond – status and future of the de facto standard for the description of road networks." In: (2010), 231–242.

[6] *OpenDRIVE Format Specification*. 2008. URL: `www.opendrive.org/`.

[7] A. Artuñedo, J. Godoy, and J. Villagra. "Smooth path planning for urban autonomous driving using OpenStreetMaps." In: (June 2017), pp. 837–842.

[8] H. Shi. "Automatic generation of OpenDrive roads from road measurements." In: (2011).

[9] *OpenStreetMap Wiki*. Aug. 2018. URL: `wiki.openstreetmap.org`.

[10] F. Chambat and B. Valette. "Mean radius, mass, and inertia for reference Earth models." In: *Physics of The Earth and Planetary Interiors*. 2001, pp. 237–253.

[11] P. Osborne. *The Mercator Projections*. 2013.

[12] The Stationery Office. *Admiralty Manual of Navigation, Volume 1*. 1987.

[13] M. Kamermans. *A Primer on Bézier Curves*. 2018.