Institute of Formal Methods in Computer Science

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit

Route planning based on destination signs in OpenStreetMap

Jan Rapp

Course of Study: Informatik

Examiner: Prof. Dr.-Ing. Stefan Funke

Supervisor: M. Sc. Florian Barth

Commenced: November 30, 2018

Completed: May 30, 2019

Abstract

This work uses destination signs to find and describe a route for the purpose of navigating a street graph. The description of a route in terms of a sequence of intermediate destinations could be: ”Drive towards city A, then B, then C”. Such directions are concise, easy to remember, and easy to communicate, especially when compared to commonly used turn-by-turn navigations. Ideally, along the route road signs exist that point the driver towards the intermedate destinations. This work extracts these so-called destination signs from OpenStreetMap data and assumes more destination signs at somewhat plausible locations to improve their connectivity. An overlay graph over the street graph is used to represent the location of, information on and connectivity between destination signs. With the overlay graph a route is generated that is described as sequence of intermediate destinations, with the guarantee that there are real or genreated destination signs along the route to guide the driver. Because destination signs are usually not available near the start and end of the route, turn-by-turn navigation is used for those segments.

Kurzfassung

Diese Arbeit nutzt Richtungswegweiser um eine Route im Straßennetz zu finden und durch diese zu beschreiben. Die Beschreibung einer Route durch eine Reihe von Zwischenzielen könnte lauten: ”Fahre in Richtung Stadt A, dann in Richtung B, dann in Richtung C”. Im Vergleich zu Turn-by-Turn-Navigation ist dies eine knappe Beschreibung, die man sich einfacher merken oder weitergeben werden kann. Idealerweise stehen entlang der Route Richtungswegweiser zu den Zwischenzielen. Diese Arbeit nutzt die Informationen über Richtungswegweiser aus den OpenStreetMap-Daten und nimmt an plausiblen Stellen noch zusätzliche Richtungswegweiser an, damit diese besser zusammenhängen. Ein dem Straßennetz überlagerter Graph wird genutzt, um die Richtungswegweiser, und wie sie zusammenhängen, zu repräsentieren. Mit diesem wird eine Route generiert, die durch Zwischenziele beschrieben werden kann und für die garantiert ist, dass die Zwischenziele durch richtige oder angenommene Richtungswegweißer gefunden werden können. Da es in der Nähe des Anfangs- und Endpunktes der Route nicht zwingend Richtungswegweiser gibt, wird hier Turn-by-Turn-Naviagtion genutzt.

Contents

1 Introduction **15**
1.1 Related Work 17
1.2 Outline 17

2 Preliminaries **19**
2.1 OpenStreetMap Street Graph 19
2.1.1 OpenStreetMap Road Representation 19
2.1.2 OpenStreetMap Destination Representation 19
2.1.3 Key Figures for Germany 22
2.2 Offset Array 22

3 Assessment of Destination Sign Coverage in OpenStreetMap **25**
3.1 Experimental Results 27

4 Routing with Destination Signs **29**
4.1 Graph Representation 30
4.1.1 Implementation 31
4.2 Augmenting the Street Graph 32
4.2.1 Label Inference 33
4.2.2 Label Promotion 36
4.3 Overlay Graph 37
4.3.1 Computation of the Overlay Graph 38
4.3.2 Implementation 39
4.4 The Routing Algorithm 40
4.5 Experimental Results 43

5 Conclusion and Future Work **47**
5.1 Future Work 47

Bibliography **49**

List of Figures

1.1 Example Route 16

2.1 Example of usage of `destination` tag in OpenStreetMap 20
2.2 Example for `destination_sign` relation usage in OpenStreetMap 21
2.3 Offset Array Illustration 22

3.1 User interface to manually evaluate the destination sign coverage in OpenStreetMap 26
3.2 Sign situation in Stuttgart city 27

4.1 Bounded Offset Array 32
4.2 Inferred Label 35
4.3 Exclusion Radius for Inferred Label 36
4.4 Label Promotion 37
4.5 Overlay Graph Example 40
4.6 Entry Q-nodes 41
4.7 Example Route 42
4.8 Runtime Plot 43

List of Tables

4.1 Used Default Maxspeed depending on Highway Tag 30
4.2 Inner Radius of Destination 34
4.3 Preprocessing Times 43
4.4 Performance Results 45

List of Listings

2.1 Example of how to traverse edges in a graph stored using offset array. 23

List of Algorithms

4.1 Computation of the Overlay Graph. 39

1 Introduction

In spoken language, a route from one location to another is often described by intermediate destinations. An example of this are the directions: “First drive towards city A, then through the towns B, C, and D. Then you arrive at E”. Such a description is concise, easy to remember and easy to communicate. The directions that common routing software such as google maps outputs are usually formulated in terms of turn by turn navigation. Especially compared to these, a description in terms of intermediate destinations is less verbose. Nonetheless, it is clear that turn by turn navigation cannot be replaced completely. This is the case when navigating to a specific address or in regions without destination signs.

In this work an approach is presented that finds routes between two locations that is based on destination signs from OpenStreetMap data [OSM] (see Figure 1.1). The resulting route is described in terms of intermediate destinations. The driver is then supposed to follow the signs towards these intermediate destinations in order, following the next intermediate destination as soon as they see a sign to it. This is implemented using an overlay graph over the regular street graph. The overlay graph contains only those routes that can be described as a sequence of intermediate destinations.

Consider a route that is based on and described with destination signs: Ideally, along such a route all necessary road signs that point the driver towards the intermediate destinations exist, such that the driver can follow the route. This work faces two problems regarding this idealization. The first problem is how a driver would “follow the road” after they have seen a destination sign. Usually, drivers follow the major road and drive straight ahead at an intersection. The path in the street graph that corresponds to the route they take is not easily figured out from the street graph. Secondly, the incompleteness of the available data sets of destination signs is problematic. These two problems are countered by assuming firstly that destination signs are placed in such a way that they lead to the destination on the fastest possible route and secondly that this route is well signposted. If a sign that is necessary for the description of the path doesn’t exist in the OpenStreetMap data, it is added to the graph representation. This supplements the signs that can be used for routing, but unfortunately it also results in conceptually wrong data in graph representation that appear in the computed route.

The experiments that were conducted show that finding a route with the approach in this work is usually faster than running the Dijkstra algorithm on the regular street graph. On the street graph of Germany the speedup for route with endpoints more than 200 km apart, is on average 18 times faster. The runtime on the Germany data seems largly uncoorelated to the distance between the endpoints of the route. By implementing more preprocessing and storage optimization, this could likely be improved by another order of magnitude.

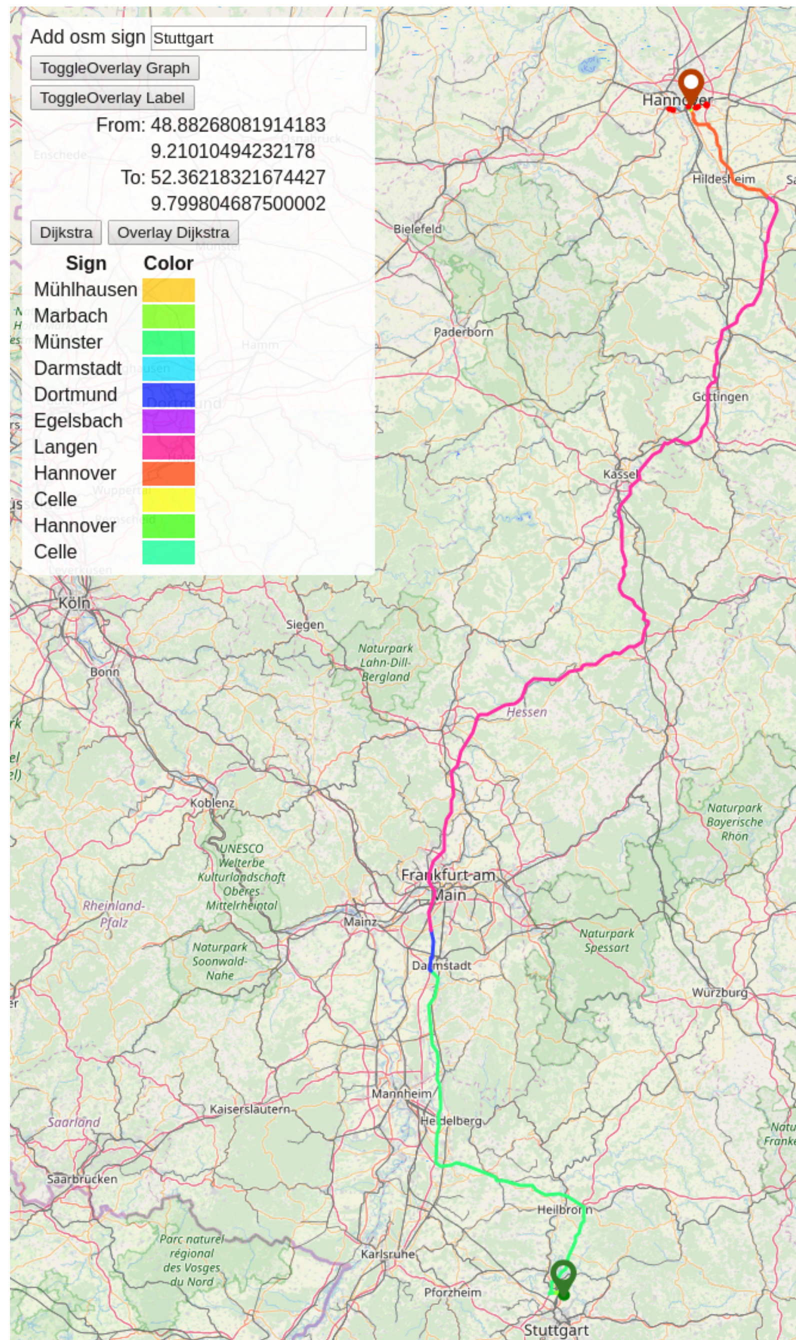

Figure 1.1: Shows a route from Stuttgart to Hanover with the algorithm described in Chapter 4. It consists of 11 different intermediate destinations. The differently colored parts in the image correspond to the route segments that the intermediate destinations correspond to. (Map credits: [OSM])

1.1 Related Work

The results of the algorithm presented in Funke et al. [FHS17] look very similar to the results of this work. Their approach differs from the one presented in this work in two fundamental aspects. Firstly, they compute, for a given path, a description in terms of intermediate destinations. While in this work, destination signs are used to compute the path itself, Funke et al. [FHS17] do not use any information related to destinations or destination signs to compute the path. Secondly, they do not use any information on real world destination signs at all. Instead they assume that destination signs occur within a certain radius of a settlement. The radius depends on the population size of that settlement.

1.2 Outline

The representation of destination signs in OpenStreetMap and the offset array data structure are explained in Chapter 2. Chapter 3 describes the experiments that were conducted to asses the quality and quantity of the existing destination sign data in OpenStreetMap. The results of the experiments are presented in Section 3.1. Chapter 4 contains everything about the routing using the destination sign data. First, it is described how the regular street graph with destination sign information is constructed from OpenStreetMap data (Section 4.1). Next, in Section 4.2, the street graph is extended and enriched in prepration for the computation of the overlay graph. The overlay graph and its generation are described in Section 4.3. Finally, how the overlay graph is used to find a route between two points in the street graph is explained in Section 4.4. Chapter 4 concludes with Section 4.5, which summarizes key figures about the precomputation and evaluates the runtime performance of the routing algorithm. The last chapter, Chapter 5, gives a conclusion, more examples, and discusses possible future work.

2 Preliminaries

2.1 OpenStreetMap Street Graph

OpenStreetMap is a collaborative effort by volunteers to collect geographical information of the world. It contains not only information on streets, but also on a wide range of other geographical features [OSMF], for example rivers [OSMR], various buildings [OSMB], power lines [OSMPl] and picnic tables [OSMPc]. The relevant data for this work is the data that OpenStreetMap provides on roads.

The entire information in OpenStreetMap is organized using three data structures: nodes, ways and relations. A node is a point on the surface of the earth, described by latitude and longitude. A way is an ordered sequence of nodes. A relation is a set of nodes, ways, and relations, where each member is assigned a string that describes its role within the relation. Each node, way and relation has a globally unique identifier that is used to reference nodes in ways or members in relations. Additionally, all three data structures can have an arbitrary number of tags, each consisting of a key and a value, both of which are strings. These data structures completely define the fundamental architecture of OpenStreetMap [OSME].

In order to use data, some sort of meaning must be attached to the data structure. The OpenStreetMap community standardize how geographical features are represented with their data structures. They define standards and conventions on how the data structures and the tags applied to them are used to convey a specific meaning. In the following, the standards for the representation of roads and destination signs are described in detail.

2.1.1 OpenStreetMap Road Representation

Roads are represented as ways with a tag that has a `highway` key and a value that specifies more precisely the type of the road [OSMH]. If two such ways share a node, a driver can physically drive from one road to the other. If ways cross but do not share a node, the driver is unable to turn onto the crossed way, as is the case if there is an overpass, for example. The ordering of the nodes in a way implies a direction. If a road is tagged with `oneway=true` (meaning that the key is `oneway` and the value is `true`) cars are only allowed to drive in the direction of the way. Otherwise the way represents a two-way road where driving in both directions is possible.

2.1.2 OpenStreetMap Destination Representation

In the context of this work, the road signs of interest are the ones that direct drivers towards a certain location or destination. Those signs are subsequently called destination signs. While the exact position of the sign post is not necessarily of interest, the information that the driver receives

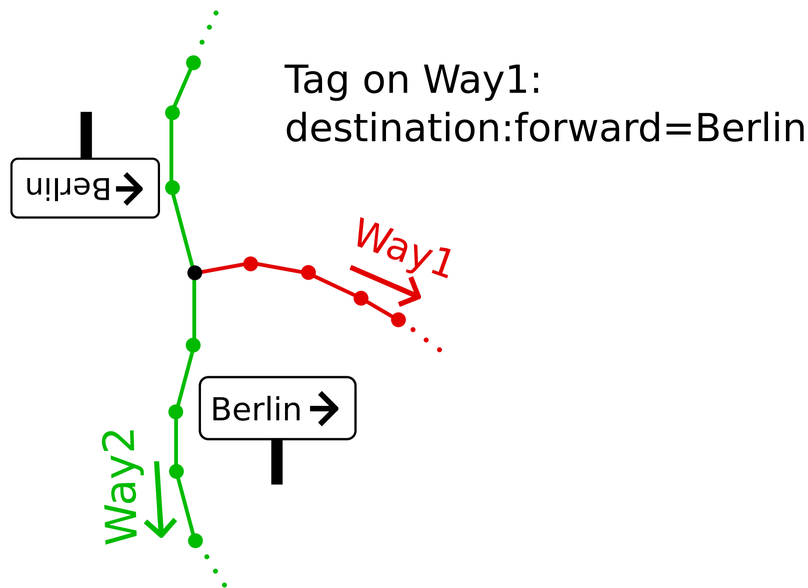

Figure 2.1: Example of the usage of the `destination` tag in OpenStreetMap. Both Way1 and Way2 represent two-way roads, but the arrows indicate the orientation of the node sequence in OpenStreetMap. Way1 is accessible from both directions of Way2. From both directions a driver sees a sign that indicates that turning onto Way1 eventually leads to Berlin. Because there is no possibility to turn onto Way1 in the forward direction without knowing that it leads to Berlin, Way1 is tagged with `destination:forward=Berlin`.

from it is relevant. This extends to the fact that if the driver gets information from multiple signs, sometimes the same information repeatedly, only the cumulative information that is available when the driver has to make a decision is relevant.

OpenStreetMap represents destination signs with one of two methods: using tags on ways or using a specific type of relation.

Destination Tag

With the first method a way is annotated with a tag that has `destination` or one of its variants as key and one or more destinations as value [OSMDT]. Multiple destination are separated by a semicolon. Such an annotated way signifies that a driver driving onto the corresponding road has seen a sign informing them that this road leads to the destinations in the value of the tag. So this method cannot be used if the driver does not get the same information from all incoming roads (see the example in the next section Figure 2.2). The `destination` key has three total variations: `destination` itself, `destination:forward` and `destination:backward`. A `destination` key without `:forward` and `:backward` part may only be used if the road is a one-way road. If it is a two-way road the `:forward` and `:backward` part indicate the direction in which the way is traveled. Figure 2.1 gives a simple example.

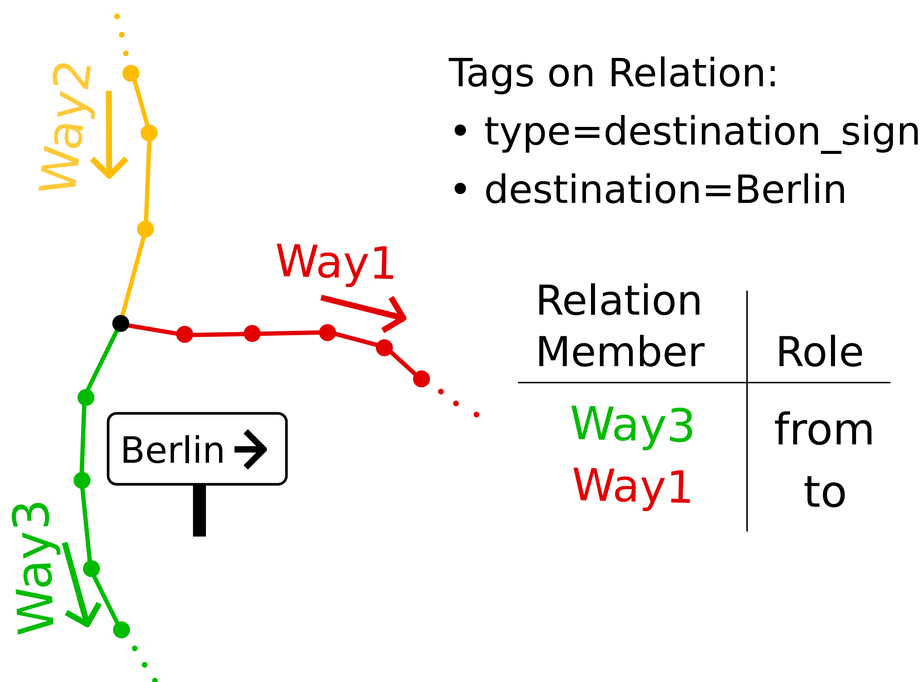

Figure 2.2: Example of the usage of the `destination_sign` relation in OpenStreetMap. This is a case where the destination tag method fails. The sign to Berlin cannot be mapped using a `destination` tag on Way1, because that would imply that a driver driving along Way2 has also seen such a sign. Instead a `destination_sign` relation must be used. In this relation Way1 has the `from` role and Way3 the `to` role. This relation would also work if the intersection (the black node in the picture) was not just a single node, but a more complex road network, for example a roundabout.

The `destination:lanes`, `destination:lanes:forward`, and `destination:lanes:backward` key do also exist. They work analogous to the tags without the `:lanes` part, but allow to specify the destinations on each lane separately. Different lanes are specified from left to right and separated using a vertical bar (`|`).

Destination Relation

Besides a tag on the way, a destination sign can also be mapped using a relation [OSMDR]. This method allows for much more freedom, but it is more complicated and less commonly used. The kind of relation used in this method has the `type=destination_sign` tag, a tag with a `destination` key and a single destination name as value. The two most important members of this relation are the ones with the `from` and `to` roles. The `from` member specifies from where the driver approaches the intersection. Typically this is the last way before reaching the intersection. It is also possible that the `from` member is the last node before reaching a node that is part of the intersection. Similarly, the `to` member specifies in which direction the driver drives to after the intersection if they follow the destination sign. The `to` member is also typically a way but it can be a node. It refers to the first way/node immediately after the intersection. Figure 2.2 provides a simple example of the destination sign relation.

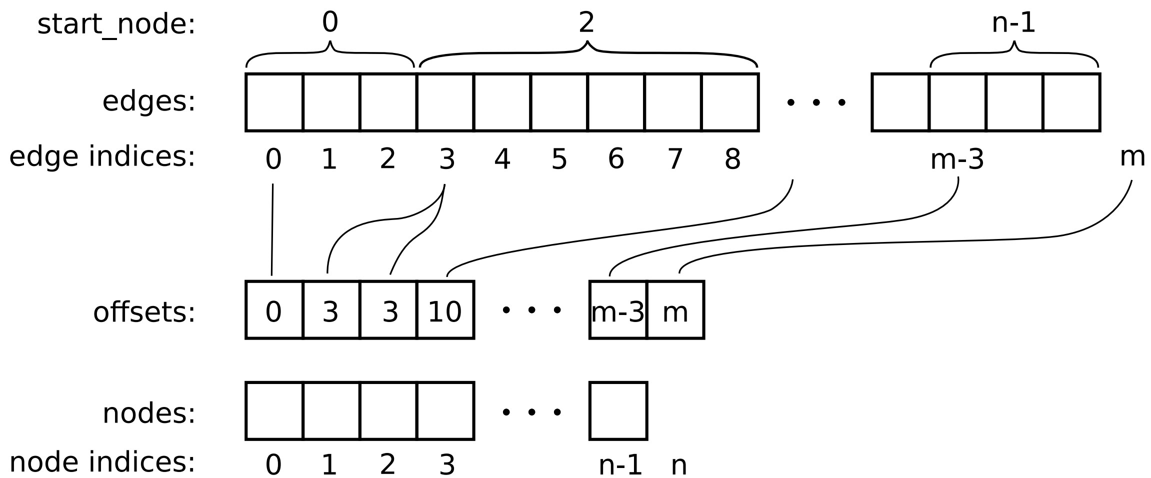

Figure 2.3: Offset Array Illustration. This offset array in this example belongs to a directed graph, where node 0 has three outgoing edges (edge indices 0,1, and 2), node 1 has no outgoing edges and node $n - 1$ has three outgoing edges.

A `destination_sign` relation can have additional tags, that indicate the color of the sign or the distance to the destination on the sign. The relation can also have additional members: a `sign` member can be a node that corresponds to the location of the sign; a `intersection` member can be a node that designates the intersection (the black node in Figure 2.2).

If a sign shows multiple destinations, a different relation is created for each of them. On the other hand, relations that only differ in the `from` and `sign` members can be combined into a single relation with multiple `from` and `sign` members.

2.1.3 Key Figures for Germany

The OpenStreetMap data used for determining the following key figures for Germany was downloaded from geofabrik [] at the end of 2018. The street graph extracted from this snapshot of Germany data contains 27 083 413 nodes, 54 654 683 edges, 179 793 destination tags on ways and 21 962 destination relations. However, if a destination relation has several `from` members, it is counted once for every such member. In total there are 20,697 distinct destinations.

2.2 Offset Array

An efficient representation of a directed graph is the offset array. The nodes are stored in a contiguous array and their index in this array is used to refer to them. The edges are also stored in a contiguous array, which is sorted by the start node of the edges. For a graph with n nodes and m edges the offset array has the size $n + 1$. For $i = 0, \ldots, n - 1$, the ith element of the offset array `offset[i]` contains the index of the first edge that has the node i as start node (see Figure 2.3). The last element of the offset array `offset[n]` is set to the value m. Therefore, all outgoing edges of node i are the edges

with indices j for `offset[i]` $\leq j <$ `offset[i+1]`. Listing 2.1 shows how this fact is used to iterate over the outgoing edges of a node. For the last node i with $i = n - 1$ the element `offset[i+1]` exists, because the offset array has the length $n + 1$ (and its entry is m).

Listing 2.1 Example of how to traverse edges in a graph stored using offset array.

```
#include <vector>
#include <iostream>

struct Node {
  double latitude;
  double longitude;
};

struct Edge {
  int32_t start_node;
  int32_t end_node;
  int32_t cost;
};

struct Graph {
  std::vector<Node> nodes;
  std::vector<Edge> edges;
  std::vector<int32_t> offsets;
};

void print_outgoing_edges(Graph &g, int32_t node_idx) {
  for (int32_t edge_idx = g.offsets[node_idx]; edge_idx < g.offsets[node_idx + 1]; ++edge_idx) {
    Edge edge = g.edges[edge_idx];
    Node end_node = g.nodes[edge.end_node];
    std::cout << "edge to (" << end_node.latitude << ", "
              << end_node.longitude << ") with cost " << edge.cost << "\n";
  }
}
```

3 Assessment of Destination Sign Coverage in OpenStreetMap

In order to assess the completeness and reliability of the destination tags in OpenStreetMap, the following method was used to compare the OpenStreetMap data to real world signs.

A phone camera is used to record the street signs on a road trip, like a dashcam. The phone is mounted behind the windshield using a car phone holder and runs a custom app, that was developed as part of this work. This app records not only a video of the street ahead, but also the current GPS location and time once per second. The recorded time of the GPS locations can be correlated with the time of the video frames. Thus, the frames in the video stream can be matched to approximate real world locations.

Connected to the phone is a Bluetooth keyboard on which a passenger presses the space key when a destination sign is in view. The app records the time of such a button press as a marker. When the data is evaluated in a later step, these markers help to find video frames that show destination signs. The GPS location at these frames is used to determine whether corresponding destination signs are tagged in OpenStreetMap.

After some trials, two additional markers, linked to other buttons on the Bluetooth keyboard, were added: one to indicate that the last marker is wrong (”wrong”marker) and one to indicate that a sign was very recently forgotten to be marked (”too late”marker).

The app is a static HTML and JavaScript page, that runs on the mobile version of Chromium on a regular Android phone. Doing all this with JavaScript is possible thanks to modern Web Application Programming Interfaces that allow access to the cameras, to store videos, retrieve the GPS location and react to keyboard presses. It is even possible to select the rear camera of a device and to be notified when a new GPS location is available. The Web API also takes care of encoding the video into chunks. Every ten minutes the app saves a chunk of the video and a JSON file containing the GPS locations and markers on the phone. The saving process works similarly to a download from a web server, the only difference being that it downloads data from the working memory of the webpage.

In order to evaluate the recorded data, a separate application is used at a later time. It plots the recorded GPS locations onto a map and shows the video. Using the recorded time of a marker and the fact that the GPS locations are in temporal order, binary search is used to find the closest GPS locations and to get an approximate location for the marker. Then that location is marked with a small circle on the map and colored according to the type of the marker. Also, the locations of destination tags in OpenStreetMap are marked on the map if they are within a radius of five kilometers of the marker location. The user interface can be seen in Figure 3.1.

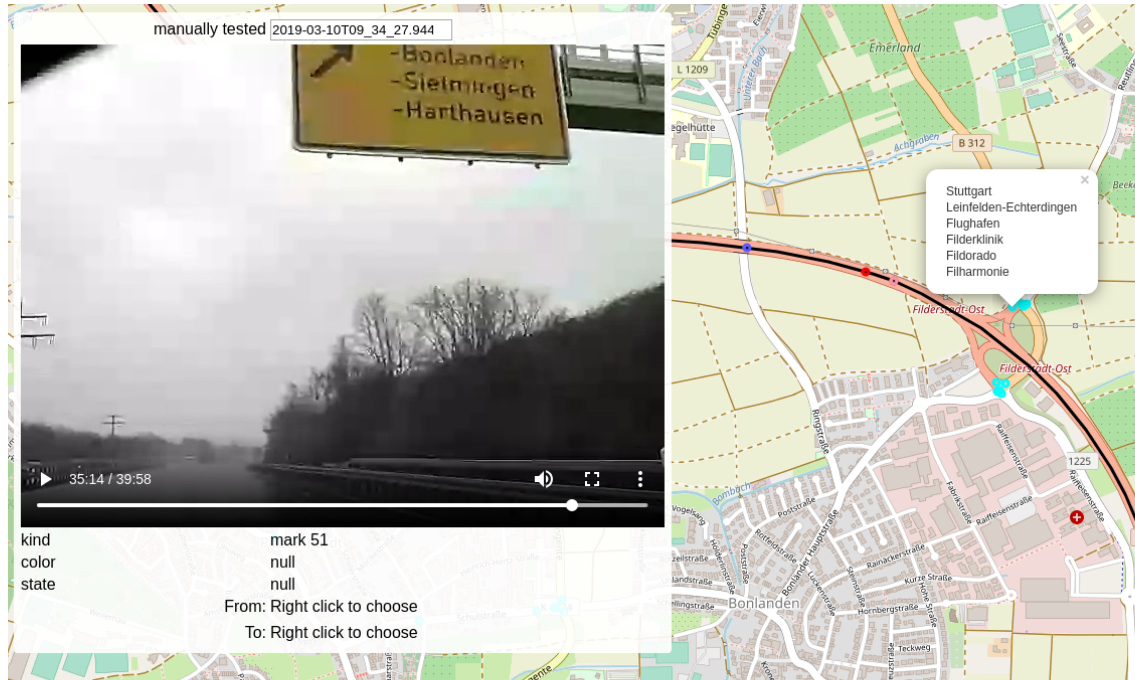

Figure 3.1: User Interface to manually evaluate the destination sign coverage in OpenStreetMap. The black line is the driving route of the car as derived from the recorded GPS locations. The cyan dots are the destination signs that are tagged in OpenStreetMap. Clicking on them reveals their destinations in a bubble. The blue dots are the markers that were made when a sign was seen while driving. The pink dot is the currently selected marker. The red dot is the GPS location closest to the currently seen frame in the video. In this scenario the complete information of the real world sign seen in the video is correctly tagged in OpenStreetMap. (Map credits: [OSM])

After using the application for a while, more keyboard shortcuts were added for easier use: to seek the video by 2 seconds or 1 frame forwards or backwards, to play/pause the video, to select a marker and jump to the next and to the last one, and to record whether the selected marker corresponds destination sign in OpenStreetMap. Another helpful addition to the application is showing the GPS location that corresponds to the current frame of the video.

Even with these aides the evaluation of the video and the comparison to OpenStreetMap data are still labor intensive. Difficult weather conditions like rain or darkness make reading a sign very difficult or impossible. Therefore, those runs were not used. Still the video quality is not very good, making it sometimes quite hard to find a video frame where the sign is readable. Nonetheless, it was possible to eventually decipher the sign content in the evaluated runs. Several difficulties arose when comparing the recorded signs to the OpenStreetMap data. The real world signs are often not close to the destination sign tag in OpenStreetMap. Understanding how a sign in the video relates to the top down map can be quite difficult, especially in a city where many signs are tagged and streets are crowded (see Figure 3.2). Additionally, real world signs are sometimes repeated several times. This information must be accumulated in order to asses the correctness of the OpenStreetMap tags.

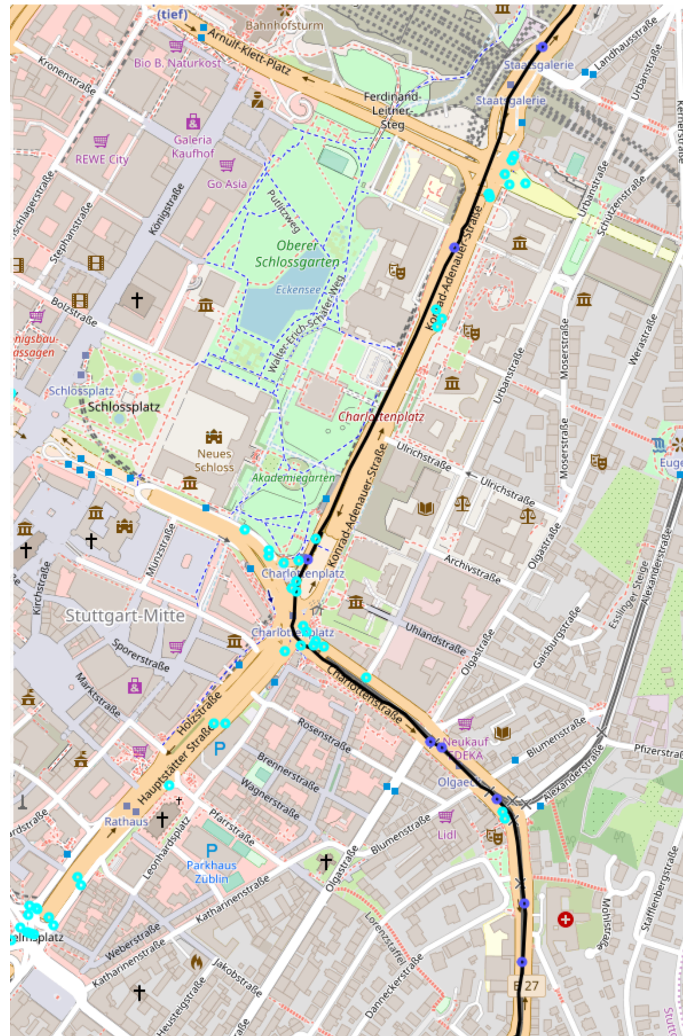

Figure 3.2: Sign situation in Stuttgart City: It can be quite difficult to assess whether a sign is missing or correctly labeled. (Map credits: [OSM])

3.1 Experimental Results

Using this setup 29 intersections where inspected along a path within Stuttgart and in the countryside around Stuttgart. All the signs visible to a driver approaching an intersection from one direction were compared to the OpenStreetMap tags near the intersection, using the following criteria:

- A real world sign or accumulation of real world signs leading up to an intersection with information that would need several tags in OpenStreetMap is only counted as one intersection.
- If the only missing destination tags are ones for destinations that lie straight ahead, the intersection is counted as correct in this evaluation.
- Similarly, if it is only a u-turn missing in the destination tag, the intersection is counted as correct.

- Slight variations due to abbreviations of destination names are also not counted as wrong, if the meaning doesn't change. For example, while according to OpenStreetMap guidelines the destination in the tag value is supposed to be exactly as the one the sign, a variation like "S-Birkach" on the sign and "Stuttgart-Birkach" in the tag is counted as correct.
- Additional information about the sign in OpenStreetMap tags, like color or icons, is ignored.

From the 29 inspected intersections 9 ($\approx 32\,\%$) were correctly represented in OpenStreetMap, 13 ($\approx 44\,\%$) missing entirely and 7 ($\approx 24\,\%$) partially correct. In this context partially correct can mean anything from almost everything is correct to only a single entry is tagged. The results varied greatly and there was no clear tendency towards either side of the spectrum.

When considering these results, the following points need to be kept in mind.

- Most notably: there is little evidence that this evaluation is representative of other parts in Germany. Since OpenStreetMap is a community effort it is natural that some places have much better coverage than others.
- Additionally, it must be taken into account that this evaluation has many manual steps, all of which are error prone to some extent. The most error prone manual step is probably marking the signs while driving. The necessity of introducing the "too late marker" is indicative of this shortcoming.
- In the later evaluation care was taken to ensure correctness, but errors could have slipped in nevertheless.

4 Routing with Destination Signs

In order to follow a route that is described in terms of intermediate destinations, a driver needs to know only two things at any given time: the destination towards which they are currently driving and the next destination in the sequence. The driver finds the route by following destination signs towards the current destination until they see a sign to the next destination. At this point, they start driving towards the next destination, which becomes the new current destination.

The algorithm presented below takes as input a start and an end location and outputs a route from start to end. The route consists of three segments. The first segment begins at the start location and lasts until the driver sees the first destination sign. The last segment covers the part from the last destination sign that the driver sees to the final destination. In between, the route is described in terms of a sequence of intermediate destinations.

The underlying data structure for this algorithm is the overlay graph. The overlay graph is a precomputed graph that directly represents how a driver follows destination signs. The algorithm itself is then a combination of three runs of the Dijkstra algorithm [Dij59], one for each of the three route segments mentioned above. For the first and the last route segment, the Dijkstra algorithm is run on the regular street graph. These two segments are computed first in order to find suitable entry and exit locations into and out of the overlay graph. These points become the start and end locations of the middle segment. The Dijkstra algorithm for the middle segment is then run in the overlay graph.

This chapter describes all steps that are needed to, starting from the OpenStreetMap data, arrive at the final routing algorithm. First, the street graph representation and its computation from OpenStreetMap data is described in Section 4.1. In preparation for the computation of the overlay graph, the street graph is augmented, as shown in Section 4.2. Finally, the overlay graph and the routing algorithm ontop of it are explained in Sections 4.3 and 4.4, respectively.

Concerning the implementation of the procedures that are described in this chapter, the following data and hardware were used. The OpenStreetMap data used in this work is the OpenStreetMap data of Germany downloaded from geofabrik [] at the end of 2018. The reference hardware used to run the software that implements what is described below is a virtual machine with about 30 GB of RAM that has 6 hyperthreads of a Intel(R) Core(TM) i7-2700K CPU available. With the exception of a single part of the precomputation, all the software is not parallelized. All the specified execution times of various parts of the software were recorded on this reference hardware.

higway Tag	Maxspeed
motorway	130 km/h
motorway_link	70 km/h
trunk	130 km/h
trunk_link	70 km/h
primary	100 km/h
primary_link	50 km/h
secondary	80 km/h
secondary_link	40 km/h
tertiary	70 km/h
tertiary_link	30 km/h
unclassified	50 km/h
residential	30 km/h
living_street	5 km/h
road	50 km/h
service	30 km/h

Table 4.1: Used Default Maxspeed depending on Highway Tag

4.1 Graph Representation

The street graph is based on data that is extracted from OpenStreetMap and it incorporates information about the road network, travel times along those roads and the real world destination signs. Real world locations and roads are represented as nodes and edges, the travel time is stored in the cost of edges and destination signs are translated into labels along edges.

Let the directed graph $G = (V, E)$ be the street graph that is extracted from the OpenStreetMap data. It consists of the nodes V and the edges E. As explained in the Section 2.1.1, roads in OpenStreetMap are represented as ways with a `highway` tag. Therefore, the street graph only depends on those ways and their nodes. The nodes V of the graph G are the OpenStreetMap nodes of the ways with `highway` tags. They correspond to real world locations and their exact latitude and longitude is stored. Every edge of the street graph represents a road segments that connects nodes. Along a way with a `highway` tag, every two nodes in its sequence of nodes are translated to an edge. Let $n, m \in V$ be graph nodes that correspond to two neighboring OpenStreetMap nodes in the sequence of a way. Then the tuple $e = (n, m) \in E$ is an edge of the street graph. n and m are called its start and end node, respectively. If the road is a two-way road, the reverse edge is also added to the graph.

The cost of an edge $e \in E$, denoted by $C(e)$, is the time it takes to travel along the corresponding road segment with maximum speed. The maximum speed is extracted from the value of the OpenStreetMap tag with a `maxspeed` key [OSMM]. If that is not available, a default depending on the `highway` tag is used (see Tabel 4.1. Every time a fastest path is computed, this is the cost that is minimized.

OpenStreetMap represents real world destination signs in one of two ways: destination tags on ways or destination sign relations (see Section 2.1.2). In this work, the information from destination sign relations is not used. Only a minority of the destination sign information is lost by this omission, because OpenStreetMap stores most of the destination sign information in destination tags on ways. Compared to the 179 793 destination tags on ways, there are only 21 962 destination sign relations. In those 21 962 relations a relation with multiple `from` members is counted multiple times, once for every such member. For a counted `destination_sign` relation the information is much less than for a destination tag on a way: On the way a destination tag gives information about every incoming edge and a counted `destination_sign` relation only for one of the incoming edges (see Section 2.1.2).

The information of road signs that is stored in destination tags on ways is translated to labels on edges. Destinations in `destination`, `destination:lanes`, `destination:forward` and `destination:lanes:forward` tags are inserted as a label on the very first edge of the way. Similarly, destinations in `destination:backwards` and `destination:lanes:backwards` tags result in a label on the last reverse edge of the way (see Section 2.1.2). For every edge $e \in E$ the set $L(e)$ is the set of labels on this edge. The set of all labels that appear in the Graph is denoted by L, so that $L = \bigcup_{e \in E} L(e)$.

4.1.1 Implementation

Two offset arrays (see Section 2.2) are used to implement the street graph. One gives access to the forward edges, the other to the backward edges.

The cost per edge is measured by the number of milliseconds necessary to drive along this edge. This value is saved in a signed 32 bit integer. The 31 bits available for the numeric value allow for a timespan greater than 24 days. This is sufficient for a car traveling along fastest routes within Germany.

All distinct labels are stored in a single array. A label is referenced using the index into this array. Therefore, instead of every edge storing multiple labels, edges store the indices of their labels in this array. An edge stores these indices using the bounded offset array, which is introduced in the next section.

The street graph extracted from the OpenStreetMap data contains 27 083 413 nodes, 54 654 683 edges and 179 793 destination tags on ways. In total there are 20 697 distinct destinations. The construction of the graph takes 18 minutes on the reference hardware.

Bounded Offset Array

An offset array is a data structure that was introduced to efficiently store the set of outgoing edges for each node. It could similarly be used to store the set of labels for each edge. However, the fact that many edges have the same set of labels can be exploited to save space[1]. This is realized by bounded offset arrays, which are a slight variation of the offset array.

[1] Later, another type of labels is introduced: the inferred labels. For inferred labels a bounded offset array is a comparatively worthwhile optimization, that results in the space efficiencies that are described in this section.

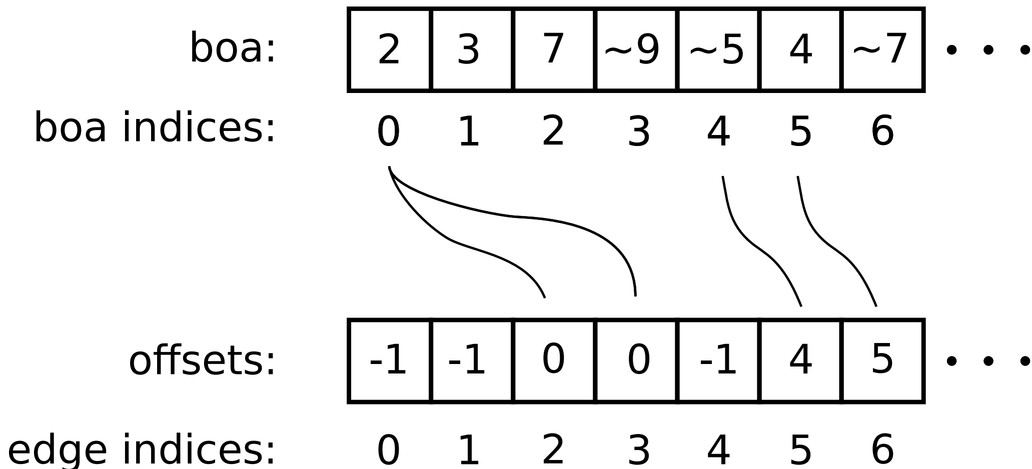

Figure 4.1: In this example, the edges with indices 2 and 3 have the labels 2, 3, 7, and 9. Because the 9 is saved as its binary complements it marks the end of the labels. The edge with index 5 only has the label 5 and the edge with index 6 has the labels 4 and 7. None of the other edges have any labels.

The bounded offset array data structure consists of two arrays: the `offsets` and the `boa` (bounded offset array). The size of the `offsets` array is the number of edges of the street graph and for every edge the array stores an index into the `boa` array. Sections of the `boa` array correspond to different sets of labels. The set of labels of the edge with index i is found in the following way. Starting at `boa[offsets[i]]` all entries of the `boa` array are part of the set of labels until one of them is negative (see Figure 4.1). That negative number is the binary complement (`operator~` in C++) of the last entry for edge i. Note that the binary complement of a nonnegatvie number is always negative and that applying the binary complement twice results in the original number. If an edge has no inferred labels the `offsets` array contains a `-1` for this edge. In this case, the `boa` array is not consulted at all.

The size of the parts in the graph representation where a bounded offset array instead of a regular offset array is used, is considerably reduced from 45 MB to 1.8 MB. However, since the total the graph representation is more than 2 GB in size, this is a negligible reduction.

4.2 Augmenting the Street Graph

When navigating to a certain destination in the real world, drivers don't expect destination signs at every intersection. Instead, without further input, they drive straight ahead instead of turning into other streets or they follow major roads when it's obviously the right choice. In this sense, destination signs imply a path that may well extend past the next intersection.

Ideally, this implied path would be the foundation of the routing algorithm. However, the task of finding this path in the graph proves difficult. For example there is no obvious solution to the problem of determining which road lies straight ahead at an intersection. It might be possible to compute angles for incoming and outgoing edges of the graph. However, if the intersection is a

roundabout, which in itself consists of several nodes and edges, it is definitely more complicated. Moreover, it is similarly not obvious how to reliably identify the major roads which a driver would follow.

In this work this problem is sidestepped by making two fundamental assumptions. The first assumption is that every real world destination sign directs a driver on the fastest path to that destination, which is a path that is easily computed. The second assumption is that any driver who comes across this shortest path sees a sign that informs them of its destination. This implies that whenever a driver must a make decision contrary to intuition, there is a destination sign informing them of this. Additionally, the second assumption compensates for the lack of tagged destination signs in OpenStreetMap (see Section 3.1) by assuming more signs at somewhat plausible locations.

In the following two sections it is explained how the fastest path from the first assumption is computed and stored and how the locations of the additional assumed signs from the second assumption are found.

4.2.1 Label Inference

The aim of label inference is to create a new kind of label, the inferred label, at all the edges along the path that a driver takes if they follow a destination sign. As previously mentioned, the assumption is that a destination sign leads the driver along the fastest path towards the destination it points to. In this step inferred labels are added along the fastest path that leads from the destination sign to the destination on the sign (see Figure 4.2). The set of all inferred labels on an edge $e \in E$ is referred to as $I(e)$. Where the destination is located is the only aspect that remains to be determined.

What can be extracted from OpenStreetMap data is an approximate center of a place. This is done by searching the nodes with both a place and a name tag. A node with a place tag corresponds to a named geographical feature, such as a city or a town square. The name tag contains that name, which is matched to the value on the destination sign. The node in the street graph that is closest to this OpenStreetMap node is selected as the assumed center of the destination of the sign.

A destination sign usually does not send the driver to the center of a place but rather to its boundary. To resolve the discrepancy between the boundary implied by the destination sign and the assumed center of the destination found before, a certain radius around the assumed center is annotated with inferred labels (see Figure 4.3). The radius is varied depending on the type of the destination. The place tag contains some information about the type of the place, for example if it is a city, a town, or a village. The radius is chosen accordingly (see Table 4.2): A city receives a larger radius than a town, which in turn receives a larger radius than a village.

A problem that arises is that sometimes several different places share the same name. In the data from Germany this is the case for 28 % of the destination names that have corresponding places. In such a case, the information about the type of the place is used to select one of them (see Table 4.2). A city, for example, is selected over a village. Naturally, this leads to some wrong inferred labels. Consider a destination sign to a village of the same name as a far away city. In this case, that destination sign is connected to the city using inferred labels, all of which are wrong.

The OpenStreetMap guideline for destination sign specifies that the exact content of the sign be entered in the OpenStreetMap data. Real world destination signs often use abbreviations or shortcuts and leave out obvious or add unobvious information. Examples are:

place Tag	Radius	
city	4	km
town	2	km
village	1	km
neighbourhood	500	m
suburb	500	m
county	15	km
municipality	30	km
farm	500	m
hamlet	50	m
isolated_dwelling	20	m
locality	10	m

Table 4.2: The radius around the destination center, in which no inferred labels are added. The order in this table is also used as priority for which place to pick. Higher entries are picked with a higher priority.

- Stuttgart-Birkach is sometimes referred to as Stuttgart-Birkach, S-Birkach, or Birkach.
- Esslingen is usually signposted as "Esslingen", but the full name is "Esslingen am Neckar".
- There are also signs to "Esslingen-Zentrum".
- If the first line of a sign reads "Filderstadt-West" and the second line reads "-Plattenhardt", "Filderstadt-Plattenhardt" is implied in the second line.

This makes it difficult to identify equivalent destinations or relate destinations to real world locations.

Implementation

Of the 20 696 distinct destinations on destination signs only for 9 977 a destination location is found using this method. But 74 % of the destinations on destination signs point to one of those 9 977 destinations. Therefore, only about 26 % of destinations are lost.

For the 179 793 destination signs for which the destination was found a total of 9 072 431 inferred labels were inserted. That is an average of 84 inferred labels per regular label. Those labels are not yet deduplicated, that happens in a later step.

The fastest path from the destination sign to the destination can also be computed from the destination to the destination sign in the reverse graph. This allows computing the fastest path from the destination to all corresponding destination signs with a single run of the Dijkstra algorithm. This Dijkstra algorithm on the reverse graph starts at the destination. As soon as it finds the fastest path to a destination sign, it adds inferred labels from the sign up to the exclusion radius.

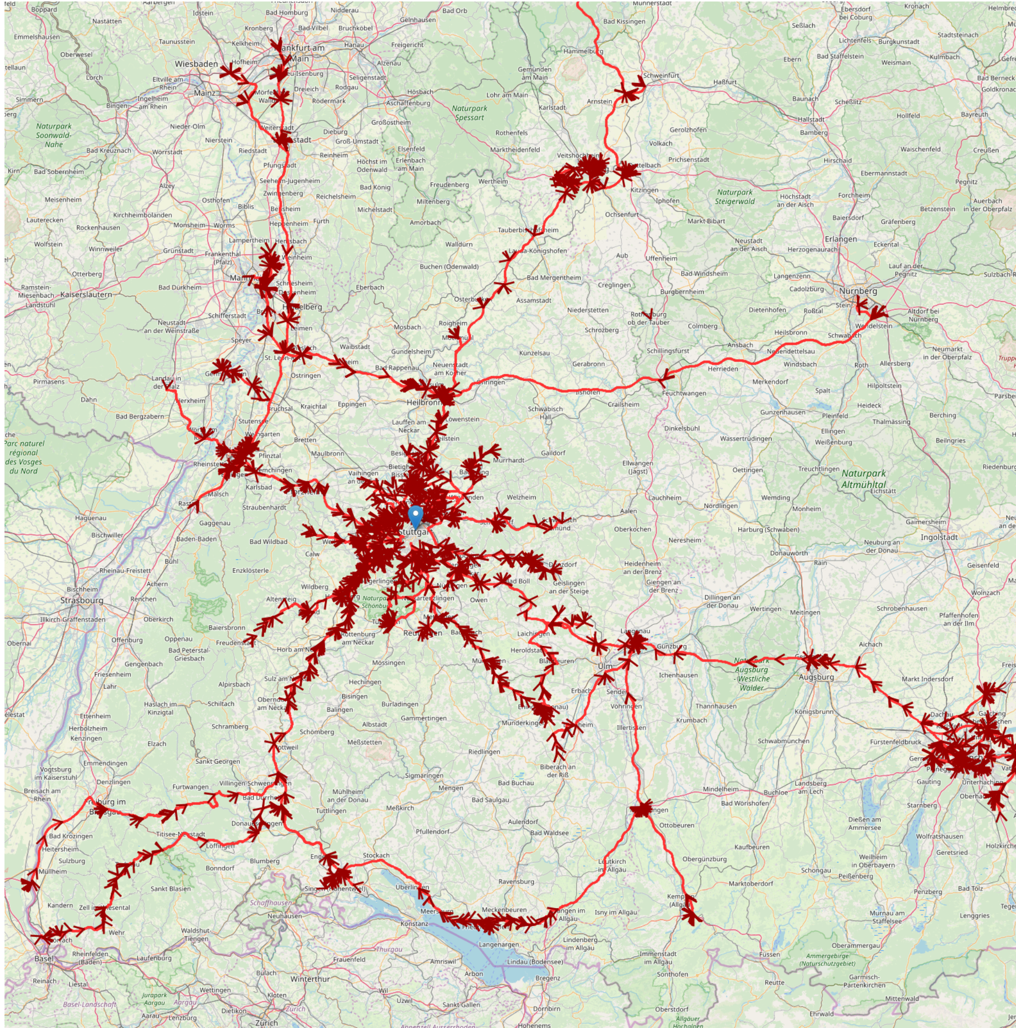

Figure 4.2: Shows the destination signs and the inferred labels for Stuttgart. The inferred labels are the red lines. The destination signs and the promoted labels (see the next section) are represented as dark red arrows. (Map credits: [OSM])

This is by far the computationally most intensive precomputation step, so it was parallelized using OpenMP. Label inference is almost perfectly parallelizable. One reverse Dijkstra algorithm for one destination is run per core. On the reference hardware this step takes around 11 hours. With this runtime it dwarfs all the other precomputation steps. The nonparallelizable tasks are deduplicating the inferred labels and inserting them in the graph, which only take about 1.4 seconds.

After the inferred labels are precomputed, they are stored per edge using a bounded offset array (see section 4.1.1).

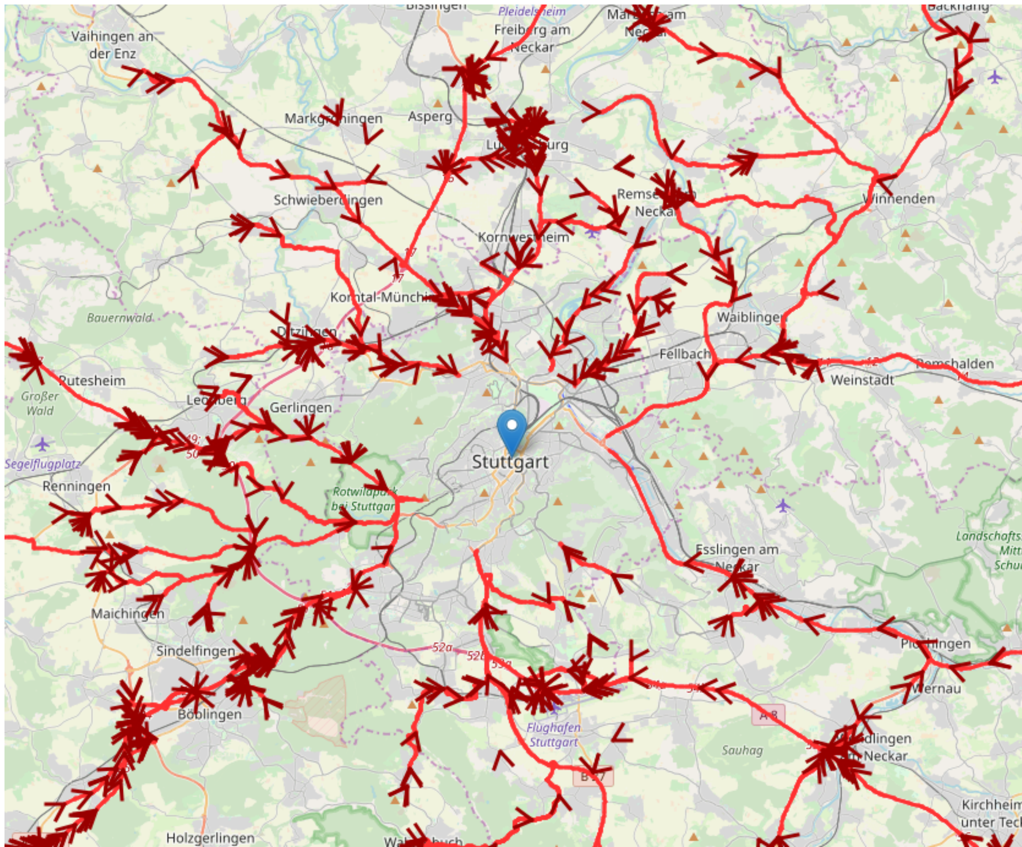

Figure 4.3: Closeup of Figure 4.2. There are no inferred label close the center of Stuttgart, because that part is explicitly excluded. (Map credits: [OSM])

4.2.2 Label Promotion

In the last section a lot of inferred labels were added along the path between destination sign and its destination. However, the second assumption is that there are destination signs along those paths at every intersection, and destination signs correspond to regular labels not inferred labels. Therefore, on all intersections that the path passes, inferred label are promoted to regular labels. In this context, any kind of fork in the road, where the driver could leave the path, is understood as an intersection. To be precise, a node $n \in V$ is an intersection if and only if it has at least one incoming edge $e = (m, n) \in E$ and at least two outgoing edges which don't lead back to m:

$$|\{(n, o) \in E : o \neq m\}| \geq 2$$

On all the outgoing edges that don't lead back to m the inferred labels are promoted, that is they are removed and added as regular labels. Finally, labels and inferred labels are deduplicated. An example can be seen in Figure 4.4.

After the label promotion all the inferred labels are exactly at those edges where the driver does not have to make any decisions. Regular or promoted labels allow the driver to make all the relevant decisions.

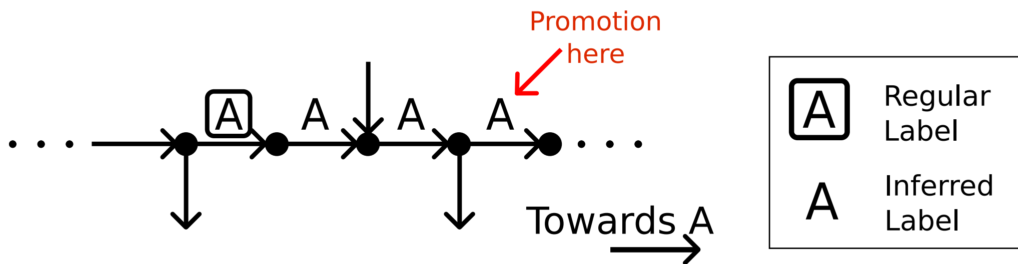

Figure 4.4: The start node of the edge with the promoted label has two (or more) outgoing edges. Therefore, it counts as an intersection. Even if there were reverse edges for all the horizontal edges shown in the figure, only the indicated label would be promoted, because those reverse edges would explicitly be excluded from the count of outgoing edges. Note that those reverse edges would have neither the labels nor the inferred labels.

Implementation

The label promotion is implemented as a single iteration over all nodes. For every node every incoming edge is selected and it is checked if two or more suitable outgoing edges exist. An outgoing edge is suitable if it is not the reverse of the selected incoming edge. On all suitable outgoing edges the labels are promoted.

On the reference hardware this step takes around 11 seconds and a total of 92 951 inferred labels are promoted.

4.3 Overlay Graph

The overlay graph is the heart of routing algorithm. An edge in the overlay graph represents a path that the driver finds by themself after having seen a destination sign. Along this path, they don't want to follow any other destination signs that they see.

The overlay graph is the directed graph $Q = (P, T)$. The nodes P of the graph Q are referred to as Q-nodes, the edges T of Q are referred to as Q-edges. The Q-nodes $P \subset (V \times L)$ are each a tuple of a node from G and one of the labels in L. The Q-edges $T \subset (P \times P)$ connect two Q-nodes. A single Q-edge $t \in T$ with $t = ((n, l), (n', l'))$ represents a path from n to n' in the street graph G that the driver follows. The driver starts following that path at n because a destination sign to l at n pointed them onto this path. They stop following that path at n' because at n' there is a sign to l'. The labels l and l' may well be the same label. In that case the driver just continues to follow the same destination.

A Q-edge represents a path in the street graph that a driver can find without receiving directions. All they need to know at its start point n are the label they currently follow and the one that they follow next. The label they currently follow is the start label l of the Q-edge. The end label l' is either the next label they need to follow or the same label as l again, in the case that the driver continues to follow the same label on the next Q-edge.

Additionally, every Q-edge $t \in T$ has a cost $C(t)$, which is the cost of the path in the street graph that it corresponds to, i.e. the time it takes to traverse the path with maximum speed.

One interesting alternative to the structure of the overlay graph is to have the labels on the Q-edges and not in the Q-nodes. The reason to put the labels in the Q-nodes is to support different signs from different directions, which is what is necessary to support `direction_sign` relations[2]. This would not be possible if the labels are on the edges. The structure of the overlay graph that is used in this work is able to support the following scenario. Consider an intersection where incoming roads have different destinations on destination signs concerning one outgoing road. In this case the overlay graph has several Q-nodes at this intersection, one for each different destination of the outgoing road. They all share the same location but have different labels that represent different destinations. From each incoming road a driver can only access those Q-nodes that are actually signposted from that side.

If a is the number of different destinations of an outgoing road at an intersection, and b is the number of other destination signs reachable from that road, then the overlay graph represents this single outgoing road by a Q-nodes at the intersection and $a \cdot b$ Q-edges. The alternative structure described above is interesting because it would have much fewer nodes and intersections. For the example scenario it would only have a single node at the intersection and only b outgoing edges.

4.3.1 Computation of the Overlay Graph

In order to compute the overlay graph from the street graph the Q-edges of the overlay graph are computed, and only their start and end nodes are inserted as Q-nodes.

If an edge $e = (n, m) \in E$ has a regular label $l \in L(e)$, starting at node m edges with inferred label l are followed. Every time a node n' is encountered that has an outgoing edge with any regular label $l' \in L$, the Q-nodes (n, l) and (n', l') and the Q-edge $t = ((n, l), (n', l'))$ are added to the overlay graph Q. However, if the label l' has already been encountered, no additional nodes and edges are added (see Listing 4.1).

Inferred labels capture the notion of a driver following the road. Consequently, when following the inferred labels in the procedure above, all locations a driver reaches by following the road are traversed in the correct order. As soon as the driver sees a sign to a desired destination for the first time, they follow it. Therefore, a Q-edge is added only at the first encounter of a destination and not at later ones (see Figure 4.5).

To ensure that routing in the overlay graph is always possible, the largest strongly connected component of the overlay graph is selected and used for routing. Subsequently, the overlay graph refers to its largest strongly connected component, unless explicitly indicated otherwise.

[2] But `direction_sign` relations are currently not used.

Algorithm 4.1 Computation of the Overlay Graph.

```
P = {}
T = {}
for all v ∈ V do
    for all e = (v, v′) ∈ E do
        for all l ∈ L(e) do
            newStartNode ← (v, l)
            DFS(newStartNode,e,l,C(e),)
        end for
    end for
end for
procedure DFS(startNode,e = (v, v′),l,cost,encountered)
    for all e′ = (v′, v′′) ∈ E do
        for all l′ =∈ L(e′) do
            if l′ ∉ encountered then
                newEndNode ← (v′, l′)
                newEdge ← (startNode, newEndNode)
                C(newEdge) ← cost
                P ← P ∪ {startNode, newEndNode}
                T ← T ∪ {newEdge}
                encountered ← encountered ∪ l′
            end if
        end for
        if l′ ∈ I(e) then
            // Note that the encountered array is copied when it is passed as a parameter.
            DFS(startNode,e′,l,cost + C(e′),encountered)
        end if
    end for
end procedure
```

4.3.2 Implementation

Like the street graph, the overlay graph is stored using the offset array data structure, which is described in Section 2.2. The largest strongly connected component of the overlay graph is computed with Kosaraju's algorithm.

Before the overlay graph is restricted to its largest strongly connected component, it consists of 258 131 Q-nodes and 1 439 628 Q-edges. 95 149 nodes of the street graph have one or more Q-nodes. The largest strongly connected component of the overlay graph has a total number of 176 450 Q-nodes and 1 064 120 Q-edges. In the street graph 51 247 nodes have one or more Q-nodes in the restricted overlay graph.

The computation of the overlay graph and the extraction of its largest strongly connected component is part of the precomputation and takes around 5 seconds. The serialized precomputed overlay graph has a size of approximately 220 MB.

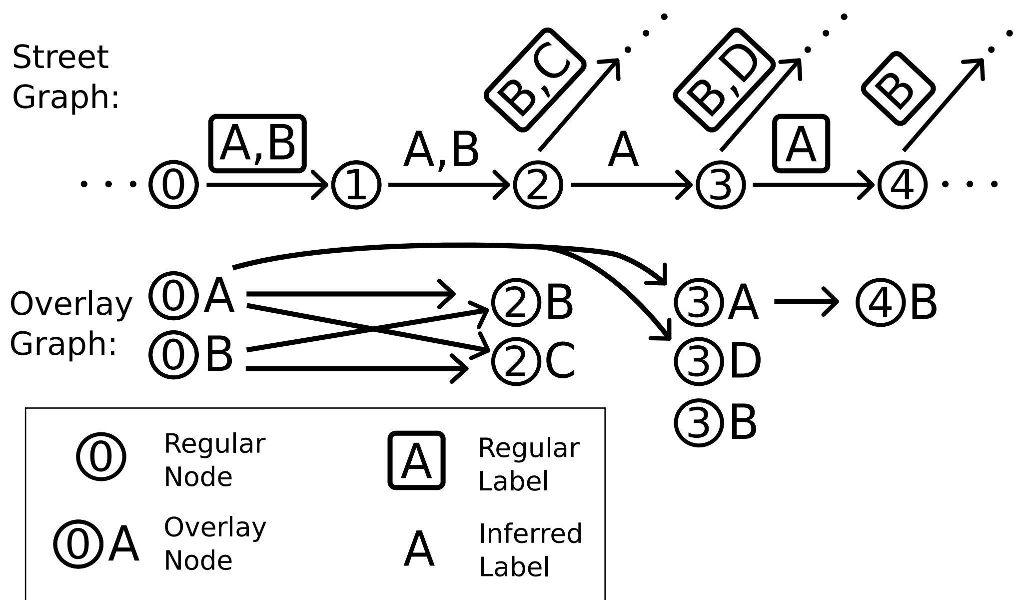

Figure 4.5: This example shows the overlay graph that corresponds to the depicted street graph. The overlay graph is constructed in the following way. Starting with the edge 0→1 and its label A, Q-edges starting at 0A are added while following inferred labels A. The inferred label A on the edge 1→2 is followed. Because node 2 has an outgoing edge with the labels B and C, the Q-edges 0A→2B and 0A→2C are added to the overlay graph. Then the inferred label A on edge 2→3 is followed resulting in the Q-edges 0A→3D and 0A→3A. The edge 0A→3B is not added, because label B was already encountered at node 2. After node 3 no more inferred labels A are encountered. All Q-edges originating from the Q-node 0A were computed. The same procedure is repeated for the label B of the edge 0→1 where the Q-node 0B is the starting point. This time the inferred labels are only followed until node 2, because the path of inferred labels B ends there. Now all Q-edges on node 0 are computed. Note that at some other time the Q-edge 3A→4B is added even though there are no inferred labels involved.

4.4 The Routing Algorithm

A shortest path in the overlay graph Q can be found by using the Dijkstra algorithm. The route to be calculated starts at a node $s \in V$ and ends at a node $t \in V$ of the street graph G. In order to find a path that is described by intermediate destinations, the route should be calculated in the overlay graph Q. This cannot be done directly, because nodes in the graph G don't necessarily have corresponding nodes in Q. In fact, of the nodes in the street graph of Germany only about 0.2 % have corresponding Q-nodes.

The routing algorithm presented here first runs two Dijkstra algorithms in the street graph G, at the start and end location respectively, until suitable Q-nodes for entering and exiting the overlay graph Q are found. They are subsequently called entry and exit Q-nodes. Then the Dijkstra algorithm is run in Q. These three runs of the Dijkstra algorithm correspond to the three segments of the resulting route that were mentioned above: The middle segment is the part of the route that is

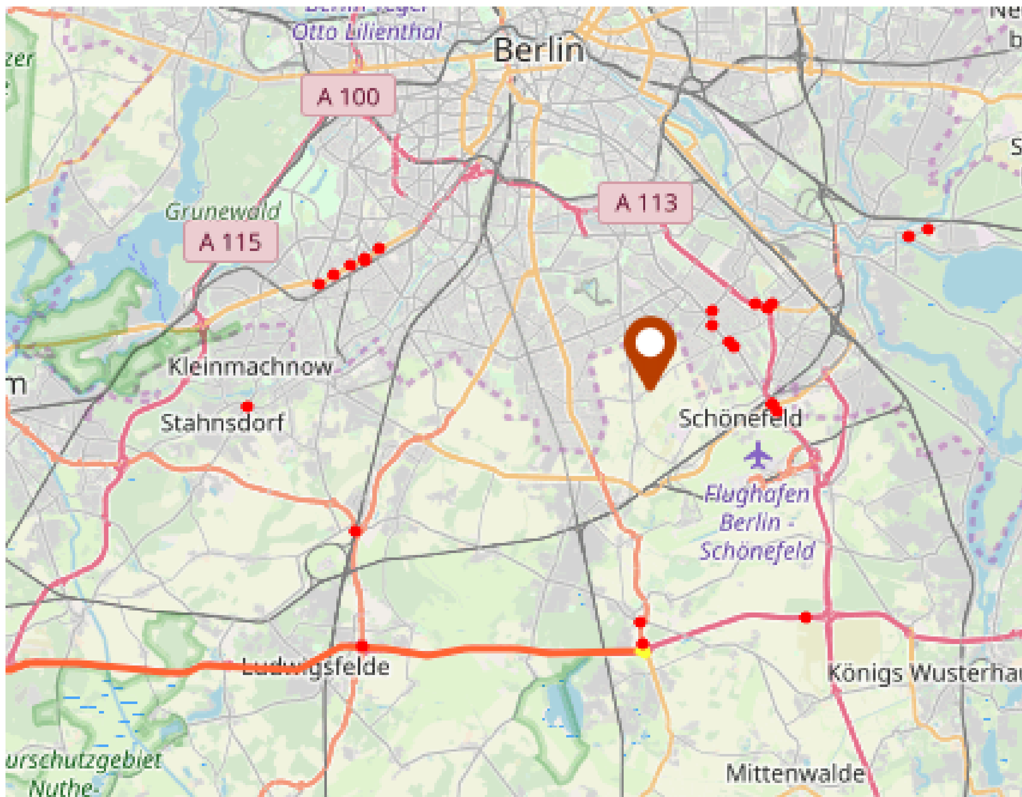

Figure 4.6: Closeup of Figure 4.7 showing the entry Q-nodes as small green circles. (Map credits: [OSM])

expressed in terms of intermediate destinations. In this part, the driver follows only destination signs. The first and the last segment connect the start and end location of the route to the first and last destination signs of the middle segment.

The Dijkstra algorithms in G only stop when every currently considered fastest path contains a Q-node. Then, for each fastest path the first Q-nodes encountered are selected as suitable entry and exit Q-nodes Eisner and Funke [EF12][3]. This ensures that for every fastest path from the start location to any other point in the graph G a Q-node is passed and that the Dijkstra algorithm has found this Q-node. The same is true for fastest paths connecting any point of G to the end location. The Dijkstra algorithm for the entry Q-nodes is run on the forward edges of G starting at s, while for exit Q-nodes the backward edges starting at t are used.

For the Dijkstra algorithm in Q the entry Q-nodes are the start locations. Their costs are initialized with the costs to reach them from s, which were computed by the Dijkstra algorithm in G. Then the Dijkstra algorithm is run in Q, until fastest paths are found to all exit Q-nodes. The final route is found by minimizing the sum of the costs from s to the exit Q-nodes and from the exit Q-nodes to t. Unless the Dijkstra algorithm in G finds the fastest path that connects the start and the end location without Q-nodes, there is a middle segment that is described by a sequence of destination signs. This sequence consists of the labels of the Q-nodes in the selected path in Q.

[3]This approach is the same Eisner and Funke [EF12] uses to find their transit nodes.

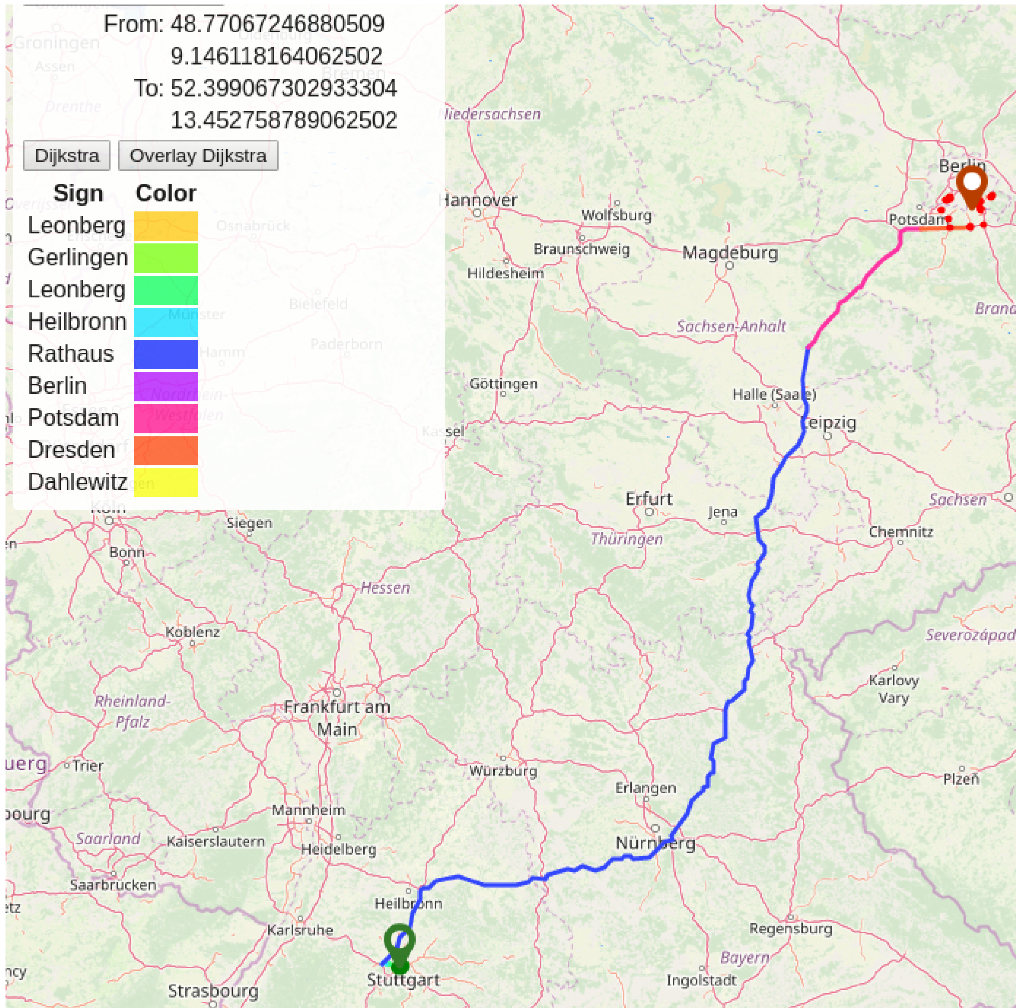

Figure 4.7: An example of a route from Stuttgart to Berlin. Note the “Rathaus” (German for town hall) destination. It is an erroneously introduced result from Section 4.2. Many town halls exist in Germany and a single one was picked as the destination of all signs with the “Rathaus” inscription. (Map credits: [OSM])

When showing this sequence to the driver, subsequences that consist of a single label repeated multiple times are reduced to a single instance of that label.

In order to show the complete route on the map, the middle segment, which is a sequence of Q-edges in Q, needs to be translated back into edges of the street graph. This is achieved with the same algorithm that was used to create the Q-edge in the first place (see Section 4.3), but this time the visited edges are recorded.

An example of the resulting route is presented in Figure 4.7.

Parse OpenStreetMap data to graph:	18	minutes
Infer Labels:	11	hours
Promote Labels:	11	seconds
Compute overlay graph:	5	seconds

Table 4.3: Preprocessing Times

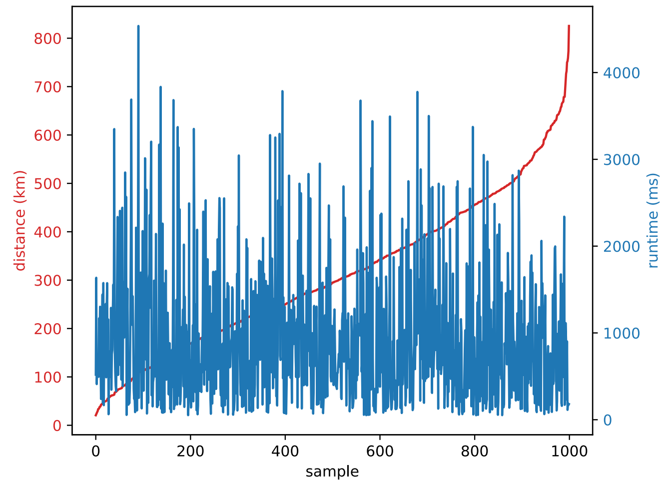

Figure 4.8: Each of the 1000 sample runs plotted with their runtime and the distance between the endnodes. The samples are sorted by the distance.

4.5 Experimental Results

An overview of the execution times of the different preprocessing steps that where discussed in this Chapter is presented in Table 4.3.

In order to evaluate the performance of the algorithm, 1000 pairs of random nodes in the street graph were chosen as input for the routing algorithm. The results of those test runs are summarized in Table 4.4. The most expensive steps by far are almost always running the Dijkstra algorithms in G, that calculate the entry and exit Q-nodes. Their execution times vary significantly. The large variation and the high maximum of the execution times makes sense considering that the execution time depends on how densely the Q-nodes are located around the start and end location. On the other hand, the Dijkstra algorithm in Q very consistently takes 52 ms to finish. In this evaluation run the runtime seems largly uncoorelated to the distance between the endnodes of the route (see Figure 4.8).

The Dijkstra algorithm for the computation of the exit Q-nodes is faster than the one for the entry Q-nodes. This could be explained by the fact that entry Q-nodes are computed first. By the time the second Dijkstra algorithm on G is run, the CPU caches are hot.

One way to further reduce significantly the computation time is to precompute the entry and exit Q-nodes for every node in G [EF12]. On the one hand, this would eliminate all the work that otherwise is necessary to compute them. On the other hand, this leads to increased storage requirement. However, as can be seen in Table 4.4, there are on average only about 13 entry and exit Q-nodes. Therefore, for street graphs, the storage requirements increases linear with the number of nodes.

The execution time of the whole algorithm is very low in comparison to a single run on the Dijkstra algorithm on the street graph. This is due to the fact that the overlay graph is so much smaller than the street graph. Why the overlay graph is that much smaller can be understood with the following example. Residential areas contain very few, if any, destination signs. Usually, those residential areas are not part of the overlay graph at all.

	Mean	Median	Std	Min	Max	at 10%	at 90%
Distance between nodes (kilometers)	310.38	294.76	157.48	20.68	825.4	113.39	526.49
Runtime to calculate entry Q-nodes (ms)	464.4	231.0	597.34	0	3440	5.0	1185.6
Runtime to calculate exit Q-nodes (ms)	410.16	226.0	500.52	0	3195	5.0	1003.2
Runtime of Dijkstra algorithm in Q (ms)	51.67	52.0	1.3	50	75	51.0	52.0
Runtime of Restoring edges.	0.4	0.0	0.49	0	2	0.0	1.0
Total routing runtime (ms)	927.42	760.0	760.27	51	4537	151.8	1988.0
Number of resulting intermediate destinations	17.61	15.0	11.41	1	71	6.0	32.0
Number of entry Q-nodes (exit Q-nodes are very similar)	13.97	11.0	10.69	1	68	3.0	28.0
Travel time along route (minutes)	131.97	118.46	70.19	11.81	377.98	50.86	232.7
Travel time on fastest Route (minutes)	128.41	114.97	69.82	11.81	374.08	47.53	226.78
Runtime comparison Dijkstra	5168.21	5081.0	3023.94	70	10720	933.1	9354.3
Factor(1) that Dijkstra takes longer[1]	14.4	6.24	23.19	0.05	169.45	0.96	40.84
Factor(2) compared to Dijkstra in Q[2]	100.04	98.7	58.47	1.0	206.15	18.2	181.53
Factor(1)[1] of Routes with distance ≥ 100 km	15.61	6.95	23.83	0.21	169.45	1.75	43.46
Factor(1)[1] of Routes with distance ≥ 200 km	18.11	8.13	25.73	0.55	169.45	2.64	48.19
Factor(1)[1] of Routes with distance ≥ 400 km	24.62	11.68	31.55	1.15	169.45	4.26	60.78
Factor(2)[2] of Routes with distance ≥ 100 km	108.01	107.11	54.26	2.9	206.15	34.46	182.54
Factor(2)[2] of Routes with distance ≥ 200 km	124.68	127.73	47.11	13.96	206.15	57.72	186.17
Factor(2)[2] of Routes with distance ≥ 400 km	154.92	164.51	36.82	38.54	206.15	98.34	192.9

Table 4.4: Statistical aggregates from 1000 route computations. Each route was computed between two random nodes; between the same nodes a comparison Dijkstra algoritm was also run. The “Std” column is the standard deviation and the “at 10%” and “at 90%” columns are the 10th and 90th percentile, respectively.

[1] Factor that the comparison Dijkstra takes longer than the total runtime of the routing algoritm of this work

[2] Factor that the comparison Dijkstra takes longer than the Dijkstra in Q

5 Conclusion and Future Work

The experiments that were conducted in this work show that the quality and quantity of destination signs in the OpenStreetMap data is suboptimal at best. In order to prepare the data extracted from OpenStreetMap for usage with the overlay graph, several steps are taken to enrich and extend the destination sign data, which does lead to some unitended results.

The fundamental data structure that this work presents is the overlay graph described in Section 4.3. It allows to represent the notion of how destination signs are followed by a driver. Based on the overlay graph a routing algorithm is presented, that allows routing in terms of destination signs. The result is a concise route description consisting of a sequence of intermediate destinations.

5.1 Future Work

Ideally, an algorithm that computes a path that a driver would follow intuitively after having seen a destination sign, would be incorporated in this work. Instead of assuming that the driver follows the fastest path to the destination, such an algorithm could use other information from the OpenStreetMap data to determine this path. For example, the type and the name of the street could be used as an additional indication of which street is followed after a complex intersection or a roundabout. With such an algorithm and a more complete set of destination signs, label inference and label promotion would not be necessary.

For a more complete set of destination signs, `destination_sign` relations need to be incorporated into the overlay graph. This can be done without much effort because the overlay graph already supports them.

The described routing algorithm in Section 4.4 only computes the fastest path in the overlay graph. A more sophisticated algorithm might also try to reduce the number of resulting intermediate destinations. On a related note, the current selection criteria for entry and exit Q-nodes into the overlay graph are very restrictive. Choosing Q-nodes that lead to a slightly longer first and last part of the route may lead to a vastly better route overall. This probably requires an algorithm that takes a more holistic approach to optimizing the route.

The solution concept could also be extended to allow interleaved turn by turn navigation and navigation in terms of intermediate destination signs. This could even be combined with a probabilistic approach to where destination signs are located, which path they indicate and which destination they describe.

Bibliography

[] *Geofabric Download Server, Download OpenStreetMap data for Germany*. URL: https://download.geofabrik.de/europe/germany.html (cit. on pp. 22, 29).

[Dij59] E. W. Dijkstra. "A Note on Two Problems in Connexion with Graphs". In: *Numer. Math.* 1.1 (Dec. 1959), pp. 269–271. ISSN: 0029-599X. DOI: 10.1007/BF01386390. URL: http://dx.doi.org/10.1007/BF01386390 (cit. on p. 29).

[EF12] J. Eisner, S. Funke. "Transit Nodes: Lower Bounds and Refined Construction". In: *Proceedings of the Meeting on Algorithm Engineering & Expermiments*. ALENEX '12. Kyoto, Japan: Society for Industrial and Applied Mathematics, 2012, pp. 141–149. URL: http://dl.acm.org/citation.cfm?id=2790265.2790279 (cit. on pp. 41, 44).

[FHS17] S. Funke, C. Haag, S. Storandt. "Generating Concise and Robust Driving Directions". In: *Proceedings of the 25th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*. SIGSPATIAL '17. Redondo Beach, CA, USA: ACM, 2017, 91:1–91:4. ISBN: 978-1-4503-5490-5. DOI: 10.1145/3139958.3140010. URL: http://doi.acm.org/10.1145/3139958.3140010 (cit. on p. 17).

[OSM] OpenStreetMap contributors. *OpenStreetMap Data*. URL: https://www.openstreetmap.org (cit. on pp. 15, 16, 26, 27, 35, 36, 41, 42).

[OSMB] The OpenSteetMap Wiki Contributors. *OpenStreetMap Wiki, building tag*. URL: https://wiki.openstreetmap.org/wiki/Key:building (cit. on p. 19).

[OSMDR] The OpenSteetMap Wiki Contributors. *OpenStreetMap Wiki, destination_sign relation*. URL: https://wiki.openstreetmap.org/wiki/Relation:destination_sign (cit. on p. 21).

[OSMDT] The OpenSteetMap Wiki Contributors. *OpenStreetMap Wiki, destination tag*. URL: https://wiki.openstreetmap.org/wiki/Key:destination (cit. on p. 20).

[OSME] The OpenSteetMap Wiki Contributors. *OpenStreetMap Wiki, OpenStreetMap Elements*. URL: https://wiki.openstreetmap.org/wiki/Elements (cit. on p. 19).

[OSMF] The OpenSteetMap Wiki Contributors. *OpenStreetMap Wiki, OpenStreetMap Features*. URL: https://wiki.openstreetmap.org/wiki/Map_Features (cit. on p. 19).

[OSMH] The OpenSteetMap Wiki Contributors. *OpenStreetMap Wiki, highway tag*. URL: https://wiki.openstreetmap.org/wiki/Key:highway (cit. on p. 19).

[OSMM] The OpenSteetMap Wiki Contributors. *OpenStreetMap Wiki, highway tag*. URL: https://wiki.openstreetmap.org/wiki/Key:maxspeed (cit. on p. 30).

[OSMPc] The OpenSteetMap Wiki Contributors. *OpenStreetMap Wiki, leisure=picnic_table tag*. URL: https://wiki.openstreetmap.org/wiki/Tag:leisure=picnic_table (cit. on p. 19).

[OSMPl] The OpenSteetMap Wiki Contributors. *OpenStreetMap Wiki, power=line tag*. URL: https://wiki.openstreetmap.org/wiki/Tag:power=line (cit. on p. 19).

[OSMR] The OpenSteetMap Wiki Contributors. *OpenStreetMap Wiki, waterway=river tag*. URL: https://wiki.openstreetmap.org/wiki/Tag:waterway=river (cit. on p. 19).

All links were last followed on May 29, 2019.

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature