Hochschule Ulm

Masterproject

Geocoding and Routing with Pelias and Valhalla

Contributors:
Martin Schmid, Sergej Dechant

Expert: Prof. Dr. von Schwerin

Expert: Prof. Dr. Herbort

Expert: Prof. Dr. Goldstein

Tuesday 17th September, 2019

Contents

1 Project Presentation and Scope **2**
- 1 Introduction . . . 2
- 2 Requirements . . . 3

2 Pelias **4**
- 1 General . . . 4
  - 1.1 Capabilities . . . 4
  - 1.2 Database . . . 6
  - 1.3 Dependencies . . . 6
- 2 System Requirements . . . 6
  - 2.1 Software Requirements . . . 6
  - 2.2 Hardware Requirements . . . 7
- 3 Installation and Configuration . . . 7
  - 3.1 Installation with Docker . . . 8
  - 3.2 Installation from scratch . . . 10

3 Data Acquisition and Preparation **12**
- 1 Two-Digit Postcodes . . . 12

4 Routing Engines **14**
- 1 General . . . 14
  - 1.1 Comparison of Routing Engines . . . 14
  - 1.2 Graphopper vs Valhalla . . . 17
  - 1.3 Conclusion . . . 17
- 2 Valhalla . . . 19

C List of Abbreviations . . . 20

Chapter 1

Project Presentation and Scope

1 Introduction

The purpose of this paper is to document the progress of the "junior team" during the first half of the data science project in form of a technical report. Moreover, this report should allow readers to gain an understanding of the topics covered in the data science project as well as be able to reproduce and extend the developed and utilized solutions. The covered tasks during the first half of the project can be categorized into three main areas:

1. Infrastructure
    - Set up a virtual machine (Ubuntu Linux)
    - Install and configure Pelias and Elasticsearch
    - Install and evaluate different routing engines
2. Data acquisition and preparation
    - Gather postcode data of European countries from different sources
    - Merge postcode data into a single data source of Pelias and Elasticsearch
3. Geocodoing and Routing
    - Test geocoding with Pelias based on precalculated two-digit postcode centroids
    - Test routing between two-digit postcode centroids wit a routing engine

2 Requirements

The main requirements were to evaluate Pelias as an open source geocoding service and as an alternative to Nominatim as well as to realize routing from one two-digit postcode to another. In order to achieve this it was necessary to build a database of postcodes and create a map of Europe based on data provided by Openstreetmaps, Whosonfirst, Geonames and Postcode-info. Furthermore, routing engines as an alternative to Graphhopper had to be evaluated. Last but not least an adequate documentation on how these requirements can be fulfilled and the outcome reproduced had to be written.

Chapter 2

Pelias

1 General

1.1 Capabilities

Pelias is a software solution/library used for geocoding. Geocoding is the process of taking input text, such as an address or the name of a place and returning a latitude/longitude location on the Earth's surface for that place. The "senior team" used Nominatim for geocoding, which is a tool for geocoding just like pelias. One of our main tasks in the course of the first half of the project was to test and evaluate pelias as an alternative open-source geocoder to Nominatim.

Here are some benefits of the pelias API:

- Completely open-source and MIT licensed
- A powerful data import architecture: Pelias supports many open-data projects out of the box but also works great with private data
- Support for searching and displaying results in many languages
- Fast and accurate autocomplete for user-facing geocoding
- Support for many result types: addresses, venues, cities, countries, and more
- Easy installation with minimal external dependencies

As mentioned above, pelias has the ability to import data from many different open-data projects as well as own private data. The importers filter, normalize, and ingest geographic datasets into the Pelias database. Currently there are five officially supported importers:

- **OpenStreetMap:** supports importing nodes and ways from OpenStreetMap
- **OpenAddresses:** supports importing the hundreds of millions of global addresses collected from various authoritative government sources by OpenAddresses
- **Who's on First:** supports importing admin areas and venues from Who's on First
- **Geonames:** supports importing admin records and venues from Geonames
- **Polylines:** supports any data in the Google Polyline format. It's mainly used to import roads from OpenStreetMap
- **Custom Data Importer:** creates a Pelias record for each row in a CSV file. Each row must define a source, latitude, longitude, and either an address, name, or both. This feature was used to import two-digit postcodes into Pelias.

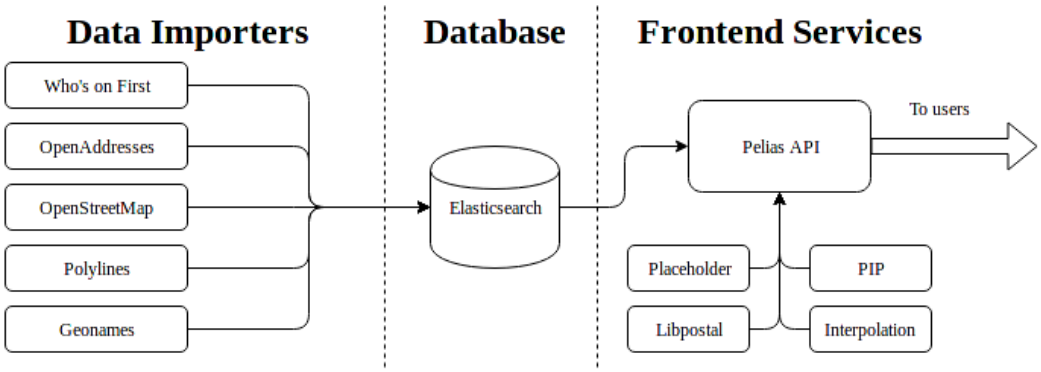

Figure 1.1: Overview of the pelias architecture

1.2 Database

The underlying datastore that powers the search results and does query-lifting is Elasticsearch. Currently version 2.4 is supported, with plans to support 5.x soon. The developers built a tool called pelias-schema that sets up Elasticsearch indices properly for Pelias.

1.3 Dependencies

These are software projects that are not used directly but are used by other components of Pelias:

- **model:** provide a single library for creating documents that fit the Pelias Elasticsearch schema. This is a core component of Pelias‘ flexible importer architecture
- **wof-admin-lookup:** A library for performing administrative lookup using point-in-polygon math. Previously included in each of the importers but now only used by the PIP service.
- **query:** This is where most of Elasticsearch's query generation happens.
- **config:** Pelias is very configurable, and all of it is driven from a single JSON file which is called pelias.json. This package provides a library for reading, validating, and working with this configuration. It is used by almost every other Pelias component
- **dbclient:** A Node.js stream library for quickly and efficiently importing records into Elasticsearch

2 System Requirements

2.1 Software Requirements

- **Node.js:** Version 8 or newer is required, version 10 is recommended for improved performance.
- **Elasticsearch:** Version 2.4 or 5.6
- **SQLite:** Version 3.11 or newer

- **Libpostal:** Pelias relies heavily on the Libpostal address parser. Libpostal requires about 4GB of disk space to download all the required data.

2.2 Hardware Requirements

- At a minimum 50GB disk space to download, extract, and process data
- 8GB RAM for a local build, 16GB+ for a full planet build. Pelias needs a little RAM for Elasticsearch, but much more for storing administrative data during import
- As many CPUs as possible. There's no minimum, but Pelias builds are highly parallelizable, so more CPUs will help make it faster.

Actual system used for the project (Europe build):

- 1 virtual machine (Ubuntu Linux) with 64 GB RAM, 500GB HDD, 4 CPU cores 5
- RAM utilization is at 30 GB, however during the import of openstreetmaps data and calculating polylines from it up to 40 GB of RAM were used. Imports and calculations maxed out all CPU cores. It is possible to reduce the required amount of RAM for imports and calculations. However, this requires splitting up openstreetmap files in smaller files with other tools beforehand.
- Including "raw data" (before the import and calculations) around 400GB of data are persisted on HDD, Elasticsearch uses 100GB.

3 Installation and Configuration

Pelias can be installed as Docker Image, manually from scratch or with Kubernetes. For testing purposes installing Pelias as a Docker Images is strongly recommended by the developers [**?**]. Pelias can also be installed manually from scratch, but due to the large amount of dependencies this is not recommended by the developers. To use Pelias in production, the development team suggests an installation with Kubernetes, which is by far the most well tested way to install Pelias according to the development team.

3.1 Installation with Docker

On the virtual machine Pelias was installed and maintained with Docker and Docker-Compose. Install Docker and Docker-Compose:

```
sudo apt-get update
sudo apt-get install \
apt-transport-https \
   ca-certificates \
        curl \
        gnupg-agent \
        software-properties-common
curl -fsSL https://download.docker.com/linux/ubuntu/gpg
    | sudo apt-key add -
sudo add-apt-repository \ "deb [arch=amd64] https://
   download.docker.com/linux/ubuntu \
        $(lsb_release -cs) \ stable"
sudo apt-get update
sudo apt-get install docker-ce docker-ce-cli containerd
   .io
sudo groupadd docker
sudo usermod -aG docker $USER
sudo systemctl enable docker
sudo curl -L "https://github.com/docker/compose/
   releases/download/1.24.0/docker-compose-$(uname -s)-
   $(uname -m)" -o /usr/local/bin/docker-compose
sudo chmod +x /usr/local/bin/docker-compose
```

Afterwards Pelias can be installed by cloning Pelias' git repository. In this repository Pelias' developers provide example projects (e.g. Beligum, Portland Metro, etc.). Pelias' "planet" project was used as a starting point for a Europe build. For this Pelias was forked on Github and cloned onto the VM. The project can be found in the following folder:

/home/dataproject/git/pelias-docker/projects/Europe

In order to build and run Pelias with data for Europe four configuration files in this folder are needed:

1. .env
2. Elasticsearch.yml

3. pelias.json

4. docker-compose.yml

TODO VERWEIS AUF ANHANG MIT DEN KONFIGURATIONSFILES In .env DATA_DIR and DOCKER_USER are important entries/variables. DATA_DIR specifies where Pelias will store downloaded data and build its other services. DOCKER_USER specifies the user id. This user id will be used for accessing files on the host filesystem in DATA_DIR since Pelias' processes run as non-root users in containers. In Elasticsearch.yml both thread pool sizes had to be increased since the default values were too small. Pelias importers delivered too much data concurrently for Elasticsearch which resulted in corrupted data. In pelias.json all Pelias services are configured. These services run as docker containers. Therefore, it is not necessary to provide complete full paths on the host filesystem or IP/DNS addresses. Paths are mapped to the paths provided in the docker compose file and .env file. Docker has its own networking and DNS. Services in a docker network can be addressed by using docker compose service names as well as container names and ids. Container ports can be mapped to host ports. The variables DOCKER_USER and DATA_DIR in docker-compose.yml are mapped to the corresponding entries in .env. Inside containers pelias.json is made available in /code/pelias.json. Ports are mapped in the following way: hostport:containerport. The "image" directive tells docker from where it has to pull the container image. In this case all images are pulled from the Pelias repository on Docker-Hub. After the colon a tag is specified (e.g. master or a version/hash). If no tag is provided, the latest version will be pulled. With this configuration it is possible to build Europe completely with the following commands and order (cd to Europe project folder first):

```
pelias compose pull
pelias elastic start
pelias elastic wait
pelias elastic create
pelias download all
pelias prepare all
pelias import all
pelias compose up
```

3.2 Installation from scratch

In order to do a clean installation of the pelias service and its dependencies on a production server at a later point in time we decided to try the installation from scratch and wrote an installation guide. The complete guide can be found in the appendix TODO VERWEIS AUF ANHANG MIT INSTALLATION GUIDE. We did the installation on a Linux VM running Ubuntu 18.04.

Installing Dependencies

Node.js: Version 8 or newer required, version 10 recommended

```
curl -sL https://deb.nodesource.com/setup_10.x | sudo -
    E bash -
sudo apt-get install -y nodejs
```

Elasticsearch: Version 2.4 or 5.6

```
wget -qO - https://artifacts.elastic.co/GPG-KEY-
    elasticsearch | sudo apt-key add -
echo "deb https://artifacts.elastic.co/packages/5.x/apt
    stable main" | sudo tee -a /etc/apt/sources.list.d/
    elastic-5.x.list
sudo apt update && sudo apt upgrade
sudo apt install apt-transport-https uuid-runtime pwgen
     openjdk-8-jre-headless
sudo apt-get update
sudo apt update
sudo apt install elasticsearch
```

SQLite: Version 3.11 or newer

```
sudo apt-get update
sudo apt-get install sqlite3
sqlite3 --version
sudo apt-get install sqlitebrowser
```

Libpostal: In order to install libpostal you will have to manually compile the source code.

```
sudo apt-get install curl autoconf automake libtool pkg
    -config
```

```
cd /
git clone https://github.com/openvenues/libpostal
cd libpostal
./bootstrap.sh
./configure --datadir=[...some dir with a few GB of
    space...]
make -j4
sudo make install
sudo ldconfig
```

Installing Pelias

Once you are done installing all the dependencies and downloaded the data for your pelias build you can start installing pelias itself.

```
for repository in schema whosonfirst geonames
    openaddresses openstreetmap polylines api
    placeholder interpolation pip-service; do
git clone https://github.com/pelias/${repository}.git #
    clone from Github
pushd $repository > /dev/null # switch into importer
    directory
npm install # install npm dependencies
popd > /dev/null # return to code directory
done
```

After the installation you will have to set up the elasticsearch schema in order to use pelias.

```
cd /pelias/schema # assuming you have just run the bash
    snippet to download the repos from earlier
./bin/create_index
```

Chapter 3

Data Acquisition and Preparation

1 Two-Digit Postcodes

CSV Data provided by geonames.org [**?**] and postcode.info, which was scrapped and provided by a fellow student [**?**], was used as basis for calculating two-digit postcodes for European countries. At the first iteration two-digit postcodes were calculated from Geonames data. In order to achieve this Python scripts were written [**?**]. These scripts process a Geonames CSV file and provide a new CSV file with two-digits postcodes including their centroids. Here is an example of six calculated two-digit postcodes in Germany:

DE80	geonames2d	80	postalcode	48.1615	11.5509
DE81	geonames2d	81	postalcode	48.1254	11.5726
DE82	geonames2d	82	postalcode	47.911	11.2502
DE83	geonames2d	83	postalcode	47.8713	12.2803
DE84	geonames2d	84	postalcode	48.4086	12.4327
DE85	geonames2d	85	postalcode	48.4174	11.6308

Table 3.1: Geonames Two-Digit Postcodes

At the second iteration Geonames and Postcode data were combined after having done some preparation steps on the Postcode data. Again Python scripts were written [**?**]. Here is an excerpt:

DE80	geonamesandpostcodeinfo	80	postalcode	48.1512	11.5938
DE81	geonamesandpostcodeinfo	81	postalcode	48.1331	11.6046
DE82	geonamesandpostcodeinfo	82	postalcode	47.9419	11.2759
DE83	geonamesandpostcodeinfo	83	postalcode	47.888	12.2627
DE84	geonamesandpostcodeinfo	84	postalcode	48.4367	12.4206
DE85	geonamesandpostcodeinfo	85	postalcode	48.4171	11.6294

Table 3.2: Geonames and Postcode.info Two-Digit Postcodes

Comparing these two tables one can see that the coordinates changed slightly. This is due to the fact that now two different data sources were used for the calculation of centroids resulting in more accurate coordinates.

Chapter 4

Routing Engines

1 General

The Pelias API and Pelias services are only suited for the purpose of geocoding and reverse geocoding. Geocoding retrieves coordinates (latitude and longitude) for a given address or postode and reverse geocoding finds the nearest known address or postcode for a provided pair of latitude and longitude. In order to find the shortest or fastest route between two given addresses Pelias has to be used in conjunction with a routing engine. The two addresses are fed into Pelias and Pelias provides coordinates for them which are then used as input values for finding a route from one coordinate to the other using a routing engine and the metrics inside the routing engine. The senior team used the routing engine Graphhopper in connection with their geocoding service Nominatim. Graphhopper as you will see later in this report is a very good and fast routing engine, however the developers of the Pelias service recommend using the routing engine Valhalla which is developed by the same company (Mapzen) as Pelias and therefore has better service interoperability with Pelias than any other routing engine.

1.1 Comparison of Routing Engines

Part of this project was to research and evaluate possible routing engines for Pelias. Internet research conducted by the junior them revealed a comparison of open source routing engines which was done by one of the members of Openstreetmaps. The following two figures illustrate the required computing

time (in ms) to calculate a route depending on the length of the route (in km)[**?**]:

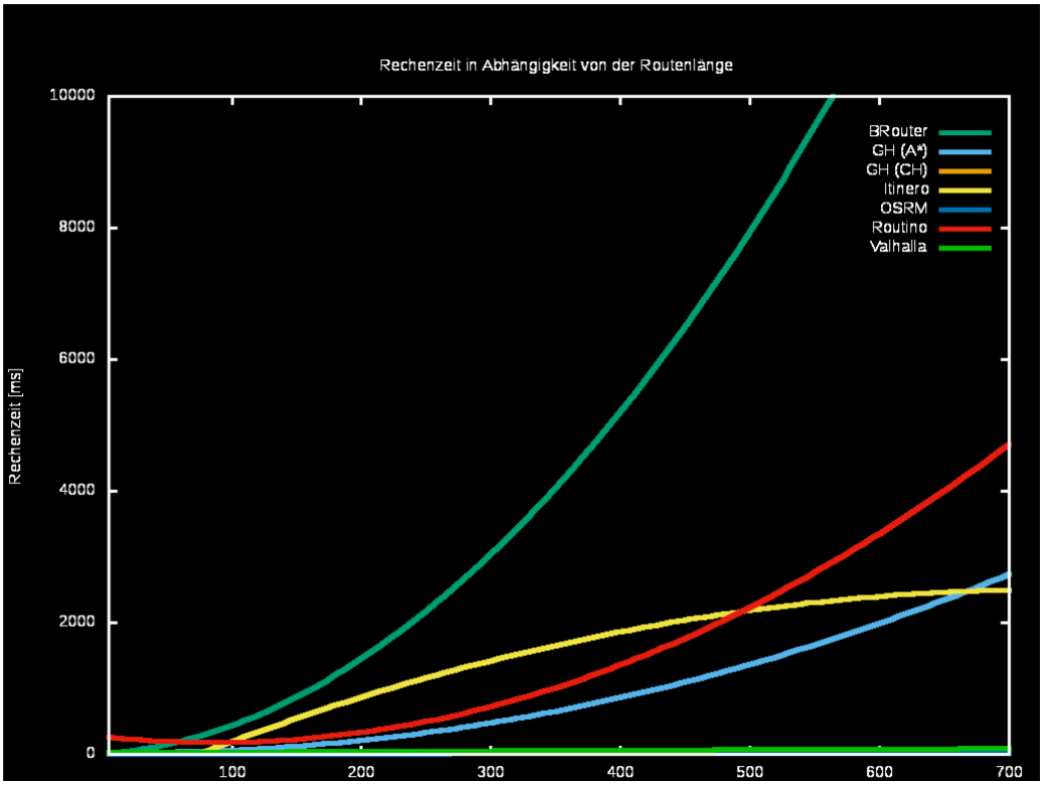

Figure 1.1: Comparison of all open source Routing Engines

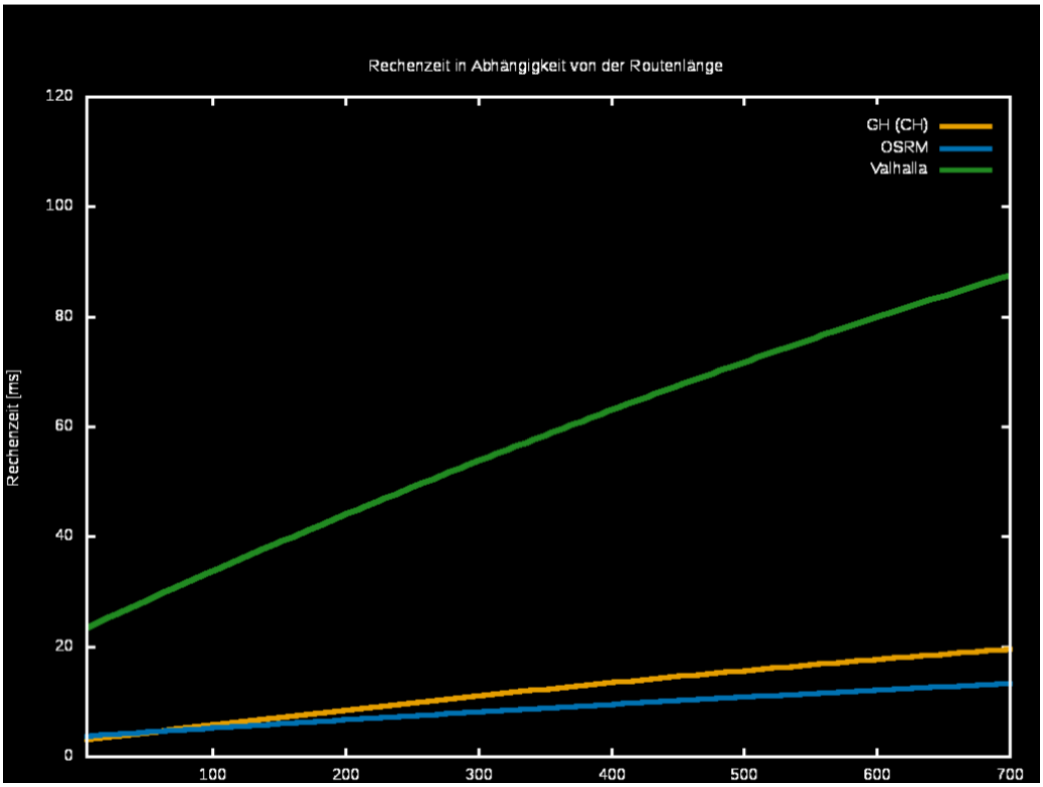

Figure 1.2: Comparison of fastest open source Routing Engines

As can be seen, Graphopper (GH), Open Source Routing Machine (OSRM) and Valhalla are the best-performing open source routing engines. Therefore, a closer look shall be taken at these three:

TODO Tabelle einfügen (LIZENZ, OS, ALGORITHMS, ETC)

Unfortunately OSRM has very high hardware requirements[**?**]. Preprocessing the car profile requires at least 175 GB of RAM and 280 GB of disk space. Additionally, 35 GB are needed for the planet osm.pbf (Openstreetmaps) and 40 to 50 GB for the generated data files. For the foot profile 248 GB of RAM are needed. During runtime the car profile requires around 64 GB of RAM, the foot profile even more. Basically OSRM loads the preprocessed files completely into RAM[**?**]. The project team's VM had only 64 GB of RAM and half of it was already used for Pelias and Elasticsearch. Hence, it was not possible to install OSRM and evaluate it completely.

1.2 Graphopper vs Valhalla

We performed an in-depth comparison of the time it takes for routing from one point to another with Graphhopper and Valhalla. The results of these test series can be seen in the table 4.1.

The API requests were executed directly on the Graphhopper and Valhalla host machines using the command line programs time and curl. We can see, that Graphhopper compared to Valhalla does a significantly better job when calculating a new route for the very first time. The execution time in Valhalla varies from the first calculation to the second calculation by up to the factor of ten. This means calculating a route or part of it, which has already been calculated before, the execution time is almost ten times faster compared to the first calculation. Valhalla achieves this with caching routes in RAM. Graphhopper however has a problem, if there is no road or street to be routed from or to for a given start- or end-point. Valhalla in this case just takes the closest routable point instead. Choosing one routing engine over the other depends on the goal which should be achieved. For fastest execution time (not regarding first or second execution) Graphhopper fits best. If you want to make sure, that you receive a route whichever point you calculate from or to, then it is recommended to use Valhalla.

1.3 Conclusion

OSRM is the fastest routing engine on the open-source market. But because of the very high memory requirements of OSRM it is not suitable for the use-case of our project and the cost/benefit-factor is too low. Valhalla would be a good alternative to Graphhopper, because it is compared to other routing engines nearly as fast as Graphhopper and is designed to work with Openstreetmaps-data and also recommended by the Pelias developers to be used in connection with Pelias as a geocoder. Also Valhalla is capable of routing from or to points, which do not have a road or street directly nearby. A very valuable feature especially for two-digit postcode centroids, which Graphhopper does not have. However, in this early state of the project Graphhopper totally fits all the needs and therefore there is no need in replacing Graphhopper with Valhalla.

		Route 1	**Route 2**	**Route 3**	**Route 4**	**Route 5**	**Route**
	Route from	Catania, Italy Latitude: 37.502236 Longitude: 15.08738	1100-148 Lisbon, Portugal Latitude: 38.707751 Longitude: -9.136592	Ulm, Baden-Württemberg, Germany Latitude: 48.3974 Longitude: 9.993434	Paris, France Latitude: 48.856697 Longitude: 2.351462	37011 Bardolino VR, Italy Latitude: 45.553553 Longitude: 10.637519	North Cape, 69, N Latitu 71.169 Longit 25.785
	Route to	1357 Copenhagen, Denmark Latitude: 55.686724 Longitude: 12.570072	Warsaw, Warszawa, Poland Latitude: 52.231924 Longitude: 21.006727	Munich, Bavaria, Germany Latitude: 48.137108 Longitude: 11.575382	Venice, Venezia, Italy Latitude: 45.437191 Longitude: 12.33459	Gunterstraße 8, 70191 Stuttgart, Germany Latitude: 48.806576 Longitude: 9.178105	89032 Bianco RC, Latitu 38.087 Longit 16.148
GH	**first try**	real 0m0.735s user 0m0.013s sys 0m0.006s	real 0m0.300s user 0m0.016s sys 0m0.006s	real 0m0.031s user 0m0.004s sys 0m0.011s	real 0m0.083s user 0m0.016s sys 0m0.000s	-	real 0m0.2 user 0m0.0 0m0.0
	second try	real 0m0.188s user 0m0.019s sys 0m0.000s	real 0m0.407s user 0m0.014s sys 0m0.005s	real 0m0.022s user 0m0.014s sys 0m0.000s	real 0m0.042s user 0m0.014s sys 0m0.004s	-	real 0m0.2 user 0m0.0 0m0.0
	distance (in km)	2753	3318	139	1113	-	5112
VH	**first try**	real 0m2.098s user 0m0.014s sys 0m0.013s	real 0m4.805s user 0m0.013s sys 0m0.021s	real 0m0.668s user 0m0.012s sys 0m0.006s	real 0m3.121s user 0m0.020s sys 0m0.003s	real 0m1.037s user 0m0.012s sys 0m0.009s	real 0m1.78 user 0m0.0 0m0.0
	second try	real 0m0.279s user 0m0.018s sys 0m0.008s	real 0m0.498s user 0m0.030s sys 0m0.008s	real 0m0.083s user 0m0.014s sys 0m0.004s	real 0m0.571s user 0m0.014s sys 0m0.008s	real 0m0.086s user 0m0.017s sys 0m0.005s	real 0m0.6 user 0m0.0 0m0.0
	distance (in km)	2682	3408	141	1116	579	4996

Table 4.1: Graphhopper vs. Valhalla test row

2 Valhalla

Valhalla was installed and configured according to the official documentation on Github. Tiles and polylines were calculated using the same openstreetmaps pbf file (Europe) which was already used for Pelias. Routing can be achieved by querying Valhalla's api:

```
curl http://141.59.29.110:8002/route --data '{"
    locations":[{"lat":48.1331,"lon":11.6046,"type":"
    break"},{"lat":47.9419,"lon":11.2759,"type":"break
    "}],"costing":"auto","directions_options":{"units":"
    km"}}' | jq '.'
```

Result:


```
"summary": {
"max_lon": 11.605507,
"max_lat": 48.133167,
"time": 2397,
"length": 38.536,
"min_lat": 47.943157,
"min_lon": 11.260186
},
"locations": [
{
"original_index": 0,
"type": "break",
"lon": 11.6046,
"lat": 48.133099,
"side_of_street": "right"
},
{
"original_index": 1,
"type": "break",
"lon": 11.2759,
"lat": 47.941898,
"side_of_street": "right"
}
```


The complete output and routing instructions are in Appendix TODO VERWEIS AUF APPENDIX EINFÜGEN.

B Illustration Directory

1.1 Overview of the pelias architecture 5

1.1 Comparison of all open source Routing Engines 15
1.2 Comparison of fastest open source Routing Engines 16

C List of Abbreviations