Institute for Formal Methods of Computer Science

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit

Consistent Pruning and Simplification of Administrative Subdivisions in OpenStreetMap

Florian Bauer

Course of Study:	Informatik
Examiner:	Prof. Dr. Stefan Funke
Supervisor:	Prof. Dr. Stefan Funke
Commenced:	May 18, 2022
Completed:	October 17, 2022

Abstract

Appealing and efficient map representations rely on simplifying their map elements in order to reduce the amount of data and rendered objects. However, simplifications inside the well-known OpenStreetMap database are not trivial due to nesting and the associated complex dependencies among elements. A good and consistent simplification of elements in a big way is, therefore, not feasible when working on this data.

This thesis aims to develop a consistent simplification strategy for administrative boundaries inside the OpenStreetMap database. In order to achieve this goal, the relevant map data inside the database gets filtered and transformed into an independent graph representation. Based on this representation, we develop different approaches to simplify, store and unpack the administrative boundaries based on their administrative levels and structure. Therefore, we introduce multiple consistent simplification approaches based on approximation and pruning, a geographical data structure for efficiently storing and accessing geographical data, and different approaches for unpacking simplified data.

Furthermore, we implement a progressive web application that uses the previously defined algorithms and an evaluation proxy for analysing the interaction between components. Finally, all components are benchmarked with static and dynamic analysis regarding the different approaches.

Kurzfassung

Eine ansprechende und effiziente Kartendarstellung beruht auf der Vereinfachung der Kartenelemente, um die Menge der Daten und zu zeichnenden Objekte zu reduzieren. Allerdings sind Vereinfachungen innerhalb der bekannten OpenStreetMap-Datenbank aufgrund der Verschachtelung und der damit verbundenen komplexen Abhängigkeiten zwischen den Datenelementen nicht trivial. Eine gute und konsistente Vereinfachung von geographischen Elementen im großen Stil ist daher nicht möglich, wenn man auf diesen Daten arbeitet.

Ziel dieser Arbeit ist die Entwicklung einer konsistenten Strategie zur Vereinfachung von Verwaltungsunterteilungen innerhalb der OpenStreetMap-Datenbank. Um dieses Ziel zu erreichen, werden die relevanten Datenelemente innerhalb der Datenbank gefiltert und in eine unabhängige Graphdarstellung transformiert. Basierend auf dieser Darstellung entwickeln wir verschiedene Ansätze, um die administrativen Grenzen auf Basis ihrer Administrationsebenen und Strukturen zu vereinfachen, zu speichern und zu entpacken. Für diesen Zweck entwickeln wir mehrere Ansätze zur konsistenten Vereinfachung auf Basis von Auslassung und Auslöschung, eine geografische Datenstruktur zur effizienten Speicherung und zum Zugriff auf geografische Daten sowie verschiedene Ansätze zum Entpacken vereinfachter Daten.

Des weiteren implementieren wir eine progressive Webanwendung, welche einen Einsatzzweck für die zuvor definierten Algorithmen darstellt und einen Evaluations-Proxy zur Auswertung von Interaktionen zwischen Komponenten. Abschließend werden alle entwickelten Komponenten mittels statischer und dynamischer Analyse im Hinblick auf die verschiedenen entwickelten Ansätze analysiert und evaluiert.

Contents

1 Introduction **17**
- 1.1 Motivation . . . 17
- 1.2 Problem Statement . . . 18
- 1.3 Outline of the Work . . . 18

2 Foundations **21**
- 2.1 Geographic Foundations . . . 21
- 2.2 OpenStreetMap . . . 23
- 2.3 Graph Theory . . . 27
- 2.4 Used Technology . . . 28

3 Graph Extraction **31**
- 3.1 Filtering of OpenStreetMap Elements . . . 31
- 3.2 Graph Creation . . . 34

4 Graph Simplification **37**
- 4.1 Data Management . . . 37
- 4.2 Simplification . . . 39
- 4.3 Geometrical Data Structure: Grid . . . 46
- 4.4 Unpacking . . . 50
- 4.5 Server and Query Handling . . . 55

5 Frontend **59**
- 5.1 Progressive Web Application . . . 59
- 5.2 Map Representation . . . 59
- 5.3 Usage . . . 60
- 5.4 Functionality . . . 61

6 Evaluation Proxy **63**
- 6.1 Query Generation . . . 63
- 6.2 Evaluation Modes . . . 63

7 Component Communication and Interfaces **65**
- 7.1 Production Mode . . . 65
- 7.2 Evaluation Mode . . . 66

8 Evaluation **67**
- 8.1 System Under Test . . . 67
- 8.2 Graph Extraction . . . 67
- 8.3 Graph Simplification . . . 68

8.4 Dynamic Evaluation 72
8.5 Visual Examples 79

9 Conclusion and Outlook **81**

Bibliography **83**

A Appendix **85**

List of Figures

1.1 OpenStreetMap cycle map of Western Europe 17

2.1 Mercator projection of the world between 85°S and 85°N 23

3.1 Filtering of a PBF file 31
3.2 Graph creation from a filtered PBF file 34

4.1 Graph class memory layout 38
4.2 Memory layout for an example graph 38
4.3 Data structure for simplification 39
4.4 Simplification hierarchy graph of a simplified administrative boundary graph . . 40
4.5 Approximation of a simple line segment with five edges over a period three iterations 42
4.6 Insertion of an edge into the grid 48
4.7 Data storage inside the grid 49
4.8 Unpacking of edges given a simplification hierarchy graph 50
4.9 Description of the full unpacking approach 52
4.10 Description of the evenly unpacking approach 52
4.11 Description of the waterfall unpacking approach 53
4.12 Description of the successive unpacking approach 54

5.1 Map representation of New Zealand 62
5.2 Settings in the advanced map 62

7.1 Component distribution and interaction in production mode 65
7.2 Component distribution and interaction in evaluation mode 66

8.1 Distribution of degrees among the nodes 69
8.2 Distribution of the administrative levels among the edges 69
8.3 Distribution of edges inside the cells of the grid when using the fast simplification approach on an administrative boundary planet graph 71
8.4 Distribution of edges inside the cells of the grid when using the minimal distance approach on an administrative boundary planet graph 71
8.5 Distribution of edges inside the cells of the grid when using the minimal surface approach on an administrative boundary planet graph 71
8.6 Edges returned by the different unpacking approaches for the coordinate pairs $(47, 6)$ and $(55, 15)$, including all administrative levels, when regarding different unpacking factors 73
8.7 Unpacking time of different unpacking approaches for 1,000 requests inside $[47°, 55°] \times [6°, 15°]$ in relation to the query area 74
8.8 Edges returned by different unpacking approaches for 1,000 requests inside $[47°, 55°] \times [6°, 15°]$ in relation to the query area 74

8.9 Network transportation times for requests inside $[47°, 55°] \times [6°, 15°]$ in relation to the response size . . . 75
8.10 Approximated rendering times for requests inside $[47°, 55°] \times [6°, 15°]$ in relation to the response size . . . 75
8.11 Number of edges inside the response of a requests in relation to the unpacking duration for 1,000 requests inside $[-90°, 90°] \times [-180°, 180°]$ . . . 76
8.12 Transformation time of requests in relation to the number of edges contained for 1,000 requests inside $[-90°, 90°] \times [-180°, 180°]$ . . . 77
8.13 Response size of requests in relation to the number of edges contained for 1,000 requests inside $[-90°, 90°] \times [-180°, 180°]$ . . . 77
8.14 Network transportation times for requests inside $[-90°, 90°] \times [-180°, 180°]$ in relation to the response size . . . 78
8.15 Approximated rendering times for requests inside $[-90°, 90°] \times [-180°, 180°]$ in relation to the response size . . . 78
8.16 Governorate Stuttgart with minimal surface simplification, successive unpacking, an unpacking factor of 25 and administrative levels 0 to 12 . . . 79
8.17 Paris with fast simplification, multi-mode activated, the evenly unpacking mode, an unpacking factor of 5 and the administrative levels 0 to 12 . . . 80
8.18 United States of America with minimal distance simplification, the waterfall unpacking mode, an unpacking factor of 84 and the administrative levels 0 to 6 . . . 80

A.1 Results from 1,000 requests inside $[47°, 55°] \times [6°, 15°]$ in relation to the request's area . . . 87
A.2 Graph search times from 1,000 requests inside $[47°, 55°] \times [6°, 15°]$ . . . 88
A.3 Unpacking times from 1,000 requests inside $[47°, 55°] \times [6°, 15°]$ . . . 88
A.4 Transformation times from 1,000 requests inside $[47°, 55°] \times [6°, 15°]$ . . . 89
A.5 Results from 1,000 requests inside $[-90°, 90°] \times [-180°, 180°]$ in relation to the request's area . . . 90
A.6 Graph search times from 1,000 requests inside $[-90°, 90°] \times [-180°, 180°]$ . . . 91
A.7 Unpacking times from 1,000 requests inside $[-90°, 90°] \times [-180°, 180°]$ . . . 91
A.8 Transformation times from 1,000 requests inside $[-90°, 90°] \times [-180°, 180°]$ . . . 92

List of Tables

2.1 Structure of OSM Nodes 24
2.2 Structure of OSM Ways 25
2.3 Structure of OSM Relations 25
2.4 Structure of relation members in OSM 26
2.5 Administrative Levels in Germany, France and Australia 27
2.6 PWA support of different browsers 30

4.1 Parameters for server queries 56

5.1 Visual graph settings for the frontend maps 60

8.1 Evaluation of the content of PBF and graph files 68
8.2 Duration and simplification depths of the different simplification approaches . . . 70

A.1 Static time analysis of the three simplification approaches 85
A.2 Degree distribution among the nodes of the administrative boundary planet graph 86
A.3 Administrative level distribution among the edges of the administrative boundary planet graph 86

List of Listings

2.1 XML representation of the OSM node of the Campus Beach in Vaihingen 24
2.2 XML representation of the OSM way of a soccer field in Stuttgart Vaihingen . . . 25
2.3 XML representation of the OSM relation of the University of Stuttgart 26
2.4 GeoJSON representation of a line consisting of four points 29
3.1 Method checking whether an *OsmWay* consists of a given tag or tag group 33
3.2 Method returning the administrative level of an *OsmWay* 35
3.3 Graph file for the House of St. Nicholas 36
4.1 Method for applying subgraph pruning 46
4.2 A basic HTTP/S request for the server 56
4.3 Transformation from a graph to GeoJSON 58
6.1 Generated query file with four queries 63
6.2 Evaluation file from clientless evaluation containing the information of five requests 64

List of Algorithms

4.1 Accessing all cells inside the AABB of two coordinate pairs 47

Acronyms

AABB Axis-Aligned Bounding Box. 47

API Application Programming Interface. 18

CCS Cartesian Coordinate System. 22

GCS Geographic Coordinate System. 21

HTTP/S Hypertext Transfer Protocol / Secure. 55

IRM IERS Reference Meridian. 22

JSON JavaScript Object Notation. 21

OSM OpenStreetMap. 18

PBF Protocolbuffer Binary Format. 23

PWA progressive web application. 21

WGS World Geodetic System. 21

WGS84 World Geodetic System, founded in 1984. 22

XML Extensive Markup Language. 23

1 Introduction

1.1 Motivation

When looking at real-time map renderings that are not based on pre-rendered tiles, the presentation of all map features in full detail is often not necessary or sometimes even disturbing. Customarily one assigns a higher priority to performance and visual appearance than accuracy when it comes to displaying maps, resulting in the need for simplification methods for map elements in order to be able to satisfy these needs.

For instance, one probably does not want to see every road in Europe when planning a bicycle trip from Stuttgart to Amsterdam, as seen in figure 1.1, but would instead like a visually appealing view of all crucial motorways with a decent simplification.

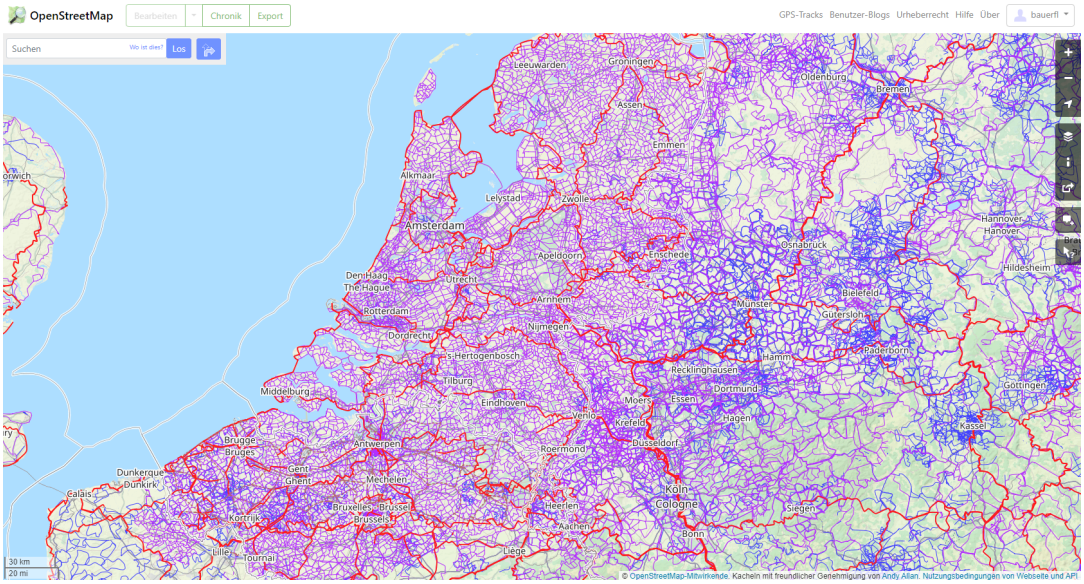


Figure 1.1: OpenStreetMap cycle map of Western Europe [1]

However, this necessity for pruning and simplification applies not only to roads and motorways but also to a lot of other geographical data, including boundaries and points of interest.

[1] https://www.openstreetmap.org/#map=8/51.941/5.114&layers=C

1.2 Problem Statement

In the scope of this thesis, we want to design a method that allows the consistent pruning and simplification of administrative subdivisions in the context of OpenStreetMap (OSM). However, consistent simplifications of map elements in Open Street Maps, especially administrative boundaries, is not a trivial task at all.

Even though there exist attributes in OSM to identify and categorize administrative boundaries, administration divisions are often represented by a set of independent OSM-ways, with a single OSM-way often being part of multiple administrative divisions. This kind of representation generates hierarchical dependencies, and therefore the issue that the simplification of a single administrative area also impacts other administrative regions, often resulting in inconsistencies.

To be able to pursue consistent simplification of administrative boundaries, it seems unavoidable to build up a representation of all administrative boundaries and, subsequently, develop a strategy for consistent approximation and pruning.

In this thesis, we will build up an administrative boundary graph and apply different consistent simplification methods to achieve the desired behaviour of fast performance and visually appealing representations. The desired result is a procedure representing OSM administrative boundaries in a data structure that allows the inquiry of specific map sections with specific detail levels. Another desired artefact of this thesis is a web application with a backend server to showcase the functionality of the data structure.

1.3 Outline of the Work

The thesis is structured as follows:

Chapter 2 - Foundations: The second chapter addresses the essential fundamentals one needs to understand this thesis. The fundamentals include geographical and technological foundations, the definition of graphs and a short introduction to the OSM project.

Chapter 3 - Graph Extraction: The third chapter deals with filtering OSM data using tags, defining a graph file format and creating graphs from filtered OSM data.

Chapter 4 - Graph Simplification: The fourth chapter is about the consistent simplification and storing of an output graph of the graph extraction in chapter 3. It also addresses the lossless unpacking of a simplified graph and the implementation of an Application Programming Interface (API).

Chapter 5 - Frontend: The fifth chapter is about the realization of a showcase for using the API we defined in chapter four. We will therefore create an application that is capable of rendering graphs.

Chapter 6 - Evaluation Proxy: In this chapter, we will concentrate on the dynamic evaluation of the frontend and graph simplification mentioned in the previous chapters. Therefore, we will implement an evaluation proxy as middleware to be able to analyze the data transfer between the frontend and backend.

Chapter 7 - Component Communication: In this seventh chapter, we will look at the collaboration and communication between the components defined in the previous chapters, the user and the external interfaces.

Chapter 8 - Evaluation: This eighth chapter evaluates the different elements built in chapters three to seven. The evaluation consists of a static analysis of every single component and dynamic analysis of the collaboration between the various system components.

Chapter 9 - Conclusion and Outlook: In this last chapter, we will summarize the results of this thesis and introduce a short outlook for potential future work.

Appendix: Appendix A shows additional benchmark results for components' static and dynamic evaluations and intercomponent interaction.

2 Foundations

This chapter deals with the essential foundations needed to understand the aspects of the following work. Those foundations include geographical foundations, the principles of OSM and the definition of graphs. Furthermore, the basics of some of the used technologies are explained, including the JavaScript Object Notation (JSON) and the fundamentals of progressive web application (PWA).

2.1 Geographic Foundations

To understand the visual representation and computational aspects of geometrical data in this thesis, one first needs to understand the foundations of the Geographic Coordinate System (GCS), the World Geodetic System (WGS) and the Mercator projection.

2.1.1 The Geographic Coordinate System

The GCS is an ellipsoidal coordinate system for position measurement on the planet earth. It uses a coordinate latitude and longitude tuple to define a position on the surface of a sphere or an ellipsoid [Cha19].

Latitudes and Longitudes

The latitude defines the angle between the equatorial plane and the line that goes through the given point and the centre of the ellipsoid. Since a point can be above or below the equatorial plane, the equator divides the ellipsoid into a northern and a southern hemisphere. We define the spectrum of the latitude angle as [-90°, 90°], regarding negative angles for the southern hemisphere and positives for the northern one. Sometimes the algebraic sign is substituted with the hemispheric sign, defining the spectrum as [90°S, 90°N].

The longitude defines the angle between a reference meridian and the line that goes through the given point and the centre of the ellipsoid. A meridian is half of a great ellipse converging at the northern and southern poles. A chosen prime meridian divides the ellipsoid into an eastern and western hemisphere. We define the spectrum of the longitude angle as [-180°,180°], regarding negative angles for the western hemisphere and positives for the eastern one. We could also alternatively use [180°W,180°E] as the spectrum.

The latitude and longitude coordinate pair specify a geographical position on the ellipsoid's surface without considering altitude or depth. In order to guarantee consistency and avoid ambiguity, we define all points x in this paper as $x = (\text{lat}, \text{lng})$ with $\text{lat} \in [-90°, 90°]$ and $\text{lng} \in [-180°, 180°]$.

2.1.2 The World Geodetic System

The WGS is a standard used for cartography and geodesy. A WGS consists of a coordinate system, a theoretical gravity, a magnetic model and a list of local transformations.

The WGS standard most acknowledged is the World Geodetic System, founded in 1984 (WGS84). It consists of a coordinate system, the Earth Gravitational Model and the World Magnetic Model. It defines its ellipsoid with the IERS Reference Meridian (IRM) as the prime meridian, a semi-major axis $a = 6,378.1370$km and a semi-minor axis $b \approx 6,356.7523$km [Nat00].

Since some computations are hard to perform on ellipsoids, the ellipsoid can be approximated by a sphere of radius $R = \frac{2a+b}{3} \approx 6,371.0088$km with an equator length of $\approx 40,075$km and a meridian length of $\approx 20,003$km.

For this project, we use the EPGS:4326 version of the WGS84, which uses the GCS as the coordinate system [Tec20].

2.1.3 Mercator Projection

In order to be able to visualize the geographical data of the WGS84 onto a two-dimensional plane, we need a conversion of the GCS into a Cartesian Coordinate System (CCS).

The Mercator projection is a cylindrical map projection, which is adequately distorted in the direction of the cylinder axis. The projection is conformal but, as a side effect, inflates the size of objects away from the equator. While multiple versions of the Mercator projection exist, we concentrate on the Standard Mercator with $\Theta = 0°$ [Mat22].

To be able to compute the Mercator projection, the coordinates must be first transformed into radians. We define

$$\rho_{\text{rad}} : [-90°, 90°] \times [-180°, 180°] \to [-\frac{\pi}{2}, \frac{\pi}{2}] \times [-\pi, \pi] : \begin{pmatrix} \text{lat} \\ \text{lng} \end{pmatrix} \mapsto \begin{pmatrix} \varphi \\ \lambda \end{pmatrix} \tag{2.1}$$

with

$$\varphi = \text{lat} \cdot \frac{\pi}{180°} \tag{2.2}$$

$$\lambda = \text{lng} \cdot \frac{\pi}{180°} \tag{2.3}$$

as the transformation of a coordinate pair from coordinate degrees to radians. Since this mapping is bijective, one can easily undo the transformation if needed.

The function of the Mercator projection itself is defined as

$$\rho_{\text{Merc}} : [-\frac{\pi}{2}, \frac{\pi}{2}] \times [-\pi, \pi] \to \mathbb{R}^2 : \begin{pmatrix} \varphi \\ \lambda \end{pmatrix} \mapsto \begin{pmatrix} x \\ y \end{pmatrix} \tag{2.4}$$

with

$$x = R(\lambda - \lambda_0) \tag{2.5}$$

$$y = R \cdot \ln[\tan(\frac{\pi}{4} + \frac{\varphi}{2})] \tag{2.6}$$

where R denotes the radius of the sphere that approximates the ellipsoid and $\lambda_0 \in [-\pi, \pi]$ denotes the longitude of an arbitrary central meridian as radian. In most cases, including this project, the IRM is chosen as the central meridian, resulting in $\lambda_0 = 0$.

Figure 2.1: Mercator projection of the world between 85°S and 85°N [Str11]

2.2 OpenStreetMap

OSM is a public editable geographic database founded as a collaborative project under the Open Data Commons license in 2004 [con04]. Its underlying geodata is the project's primary output, and it is based on the WGS84. Prominent applications based on OSM data are maps, humanitarian aid and navigation.

2.2.1 File Formats

OSM offers various data formats for its data, with the most common file formats being the Extensive Markup Language (XML) format and the Protocolbuffer Binary Format (PBF).

XML is a meta format with the idea of combining a good compression ratio with the aspect of human-readability and clear structure [con22c]. In the context of OSM, an XML file is usually encoded in UTF-8 and contains an OSM element with the version of the API as well as a block for each of the three main OSM data types that we will describe in then ext section.

The PBF, however, is intended as an alternative to the XML format [con22d]. It allows smaller file sizes and faster reading and writing speeds and is designed to support future extensibility and flexibility. Due to its complex, unclear structure and extreme compression, it is considered a human-unreadable format.

The whole database (planet) and more minor extracts are freely available. They can be downloaded in the XML format or the PBF from official sources and third-party providers such as Geofabrik[1].

2.2.2 Map Elements

The OSM database mainly consists of three basic elements: nodes, ways and relations. Those components can form different relationships resulting in even more meaningful structures. In addition, those elements can be further described by so-called tags, which we will introduce later.

Nodes

A node is geographically based on the WGS84 and defines a single point in space by its latitude, longitude and identification. Nodes can be used to describe standalone point features in which they typically contain at least one tag defining their purpose or can be used as a part of a way or relation. There also exist three-dimensional nodes with the altitude included as an elevation tag [con22b].

Attribute Name	Description
id	64-bit integer number ≥ 1, which is unique between nodes
lat	decimal number in $[-90, 90]$ with seven decimal places
lon	decimal number in $[-180, 180]$ with seven decimal places
tags	set of key/value pairs

Table 2.1: Structure of OSM nodes [con22b]

Listing 2.1 XML representation of the OSM node of the Campus Beach in Vaihingen

```
<node id="8281060669" visible="true" version="1"  lat="48.7459165" lon="9.1064940">
    <tag k="amenity" v="bar"/>
    <tag k="name" v="Campus Beach"/>
    <tag k="opening_hours:covid19" v="closed"/>
    <tag k="website" v="https://campusbeach.stuvus.uni-stuttgart.de/"/>
    <tag k="wheelchair" v="yes"/>
</node>
```

[1] https://download.geofabrik.de/

Ways

A way is an ordered list of nodes, which usually contains at least one tag or is included within a relation. Ways should contain between 2 and 2.000 nodes, but ways with other node counts do also exist. If the first node and last node of a way are identical, we call the way closed; otherwise, we call it open [con22i].

Attribute Name	Description
id	64-bit integer number ≥ 1, which is unique between ways
node refs	list of references of nodes
tags	set of key/value pairs

Table 2.2: Structure of OSM ways [con22i]

Listing 2.2 XML representation of the OSM way of a soccer field in Stuttgart Vaihingen

```
<way id="25873801" visible="true" version="3">
    <nd ref="282228781"/>
    <nd ref="282228782"/>
    <nd ref="282228783"/>
    <nd ref="282228784"/>
    <nd ref="282228781"/>
    <tag k="leisure" v="pitch"/>
    <tag k="sport" v="soccer"/>
</way>
```

Relations

A relation is a group with at least one tag defining a logical or geographic relationship between its members. The members are an ordered list consisting of nodes, ways and other relations, with the maximum recommended member size being 300. Because relations can contain other relations, constructing more extensive elements is feasible, allowing the assembly of any needed geographical structure [con22e].

Attribute Name	Description
id	64-bit integer number ≥ 1, which is unique between relations
node refs	list of references of nodes
tags	set of key/value pairs

Table 2.3: Structure of OSM relations [con22e]

Attribute Name	Description
type	type of the element (node/way/relation)
ref	reference to the member element
role	optional role of the member inside the relation

Table 2.4: Structure of relation members in OSM [con22e]

Listing 2.3 XML representation of the OSM relation of the University of Stuttgart

```
<relation id="25566" visible="true" version="76">
    <member type="node" ref="2934230475" role=""/>
    ...
    <member type="node" ref="5157388987" role=""/>
    <member type="way" ref="156754655" role=""/>
    ...
    <member type="way" ref="469419940" role=""/>
    <member type="relation" ref="2922269" role=""/>
    ...
    <member type="relation" ref="1342887" role=""/>
    <member type="relation" ref="5294338" role=""/>
    <tag k="name" v="Universitaet Stuttgart"/>
    <tag k="type" v="organization"/>
</relation>
```

2.2.3 Feature Descriptions

The basic elements represent geometrical or logic structures but cannot describe non-geographic information. In order to be able to annotate elements with additional information, so-called tags exist in OSM.

Tags

Tags are built as a key-value pair and describe specific features of map elements. Each map element can contain as many tags as needed to express itself. However, each tag should only be contained once in an element. If an element does not include a specific tag, there often exists a default value for this tag, or the values can be inherited from parent elements [con22h].

The tag is written as *key*=*value*, with the key of a tag being used to describe a category of a feature (i.e. boundary) and the value being used to provide more details about the key-specific feature (i.e. administrative).

Boundaries

A boundary tag (*boundary=**) marks the borders of specific areas. In this thesis, we will make particular use of the *boundary=administrative* tag, which is used to represent subdivisions of areas from recognized governments or organizations [con22f].

Each element containing this tag usually also contains the *admin_level=** tag, which describes the administrative level of a feature within a government hierarchy.

The administrative levels are usually between two and ten; however, some governments also use level eleven. Whereas level two marks national boundaries, levels three to eleven are used for sub-national boundaries. Some examples can be found in table 2.5.

Level	Germany	France	Australia
3	-	Territorial areas	-
4	Bundesland	Régions	State/Territory
5	Regierungsbezirk	Circonscription départemantale	-
6	Landkreis	Départements	Local Government
7	Verwaltungsgemeinschaft	Arrondissements	District Border
8	Stadt	Communes	-
9	Stadtbezirk	Communes associées	Locality Border
10	Stadtteil	Quartiers	-
11	Stadtviertel	-	-

Table 2.5: Administrative Level Mapping for different countries in OSM [con22g]

2.3 Graph Theory

We define a graph $G = (N,E)$ as a set of nodes N (also called vertices) and a set of edges E (also called links or lines). Nodes represent geographical or abstract points, which one can connect via edges. If two nodes are connected with an edge, we call them *adjacent.* An edge itself always consists of a start node and an end node and may be directed or undirected [Tru15].

In addition to this definition of a graph, one can add attributes to nodes or edges to enhance the meaningfulness of graphs. A simple extension would be the insertion of coordinates for nodes to create geometrical graphs.

2.3.1 Types of Graphs

In addition to the general definition of graphs, we will also define some essential attributes that graphs can have.

- We call a graph *finite* if both the node and edge sets of the graph are finite. In this thesis, we will only operate on finite graphs.
- We call a graph *planar* if the nodes and edges of the graph are drawable in a plane so that no edges intersect.
- We call a graph a *path* if it contains at least two nodes and the degree of all nodes is two except for exactly two edges. Such a graph has $n \geq 2$ nodes and $n - 1$ edges.
- We call a graph *cyclic* if it contains at least three nodes, and it can order a subset of its vertices in such a way that the first and last vertices are the same and that each pair of subsequent vertices is connected with an edge.

2.3.2 Relations between Graphs

We can furthermore define relations between different graphs.

- We call a graph G a subgraph of a graph H if the nodes and edges of G form a subset of those from H.
- We call a subgraph G of a graph H independent if all nodes in G have the same degree as they have in H.

2.4 Used Technology

To be able to understand some of the decisions made in the course of the thesis, it is essential to have an understanding of the fundamentals for some of the used technologies.

2.4.1 JavaScript Object Notation

Foundations of JSON

The JSON is a compact, language-independent and human-readable data format used for data transfer between applications. The JSON was specified in the early 2000s and is nowadays supported in most common programming languages [CM17]. The JSON Schema specifies a JSON-based format to define the structure of JSON data. The JSON Schema is based on the concepts of the XML Schema and is, therefore, pretty similar.

GeoJSON

GeoJSON is an open standard format based on JSON and is designed to represent simple geographical features and their non-spatial attributes [BDD+16]. The GeoJSON standard supports geometry primitives such as points, lines, polygons and multipart geometries such as multi-points or multi-lines. The representation of a single line with four points is shown in listing 2.4. The standard was specified in 2008, and many software projects, including Leaflet, support it.

Listing 2.4 GeoJSON representation of a line consisting of four points

```
{ "type": "FeatureCollection",
  "features": [
    { "type": "Feature",
      "geometry": {
        "type": "LineString",
        "coordinates": [
          [1.0, 0.0], [1.0, 1.0], [1.0, 0.0], [1.0, 1.0]
        ]
      }
    }
  ]
}
```

2.4.2 Leaflet

Leaflet is one of the leading open-source JavaScript libraries for interactive maps [Aga10]. The library was founded in 2011, designed for simplicity, performance and usability, and works across most major platforms. Leaflet allows the presentation of dynamic maps via the combination of a web map tile service and one's own geographical data.

The Leaflet API

Leaflet offers the usage of a map with different map methods on which one can place different layers, such as UI layers, vector layers or raster layers. In most cases, including this thesis, different layers are used to assemble a mix of functionalities [Aga10].

Typical usage would be the creation of a map with a raster layer for tile images. On top of this layer, one would place a vector layer for their geographical data and a UI layer for user interaction.

Web Map Tile Service

A Web Map Tile Service is a standard protocol specified by the Open Geospatial Consortium in 2010 for serving georeferenced map tiles over the internet. The service's idea is to offer pre-rendered tiles of specific map sections as images on demand for performant presentation in applications. Leaflet uses this service for a subcategory of the raster layer, the so-called tile layer.

2.4.3 Progressive Web Application

PWA combines the advantages of native applications with the benefits of responsive websites. It avoids double developments for an application and website by allowing the symbiosis of both [con22a].

The idea of a PWA is that one can build it as a regular website using HTML5, CSS3 and JavaScript but can extend it with a manifest and service workers allowing offline functionality, caching and downloading. The compatibility of PWAs can be seen in table 2.6.

Service Workers

A service worker is a script that runs in the browser's background and offers essential functions such as caching for offline usage. If we retrieve the data once, the service worker allows the website's usage without a server connection (offline usage).

VueJS

VueJS is a JavaScript framework developed by Evan You in 2014 [You14]. The framework is based on the model-view-viewmodel pattern and is designed to create user interfaces and single-page applications. It allows the creation of PWAs and supports client-based and server-side rendering.

Browser	**Support**					**Comment**
	Windows	Linux	macOS	Android	iOS	
Chromium-based	yes	yes	yes	yes	-	Chromium includes
Firefox	no	no	no	partial	no	Chrome, Edge, Brave,
Safari	-	-	yes	-	yes	Opera and others

Table 2.6: PWA support of different browsers [con22j]

3 Graph Extraction

After introducing all the essential foundations, this chapter aims to extract a graph containing nodes and edges from a regular OSM PBF file. We split the extraction into two phases: a filtering phase in which we reduce the PBF to the crucial elements and a graph creation phase in which we detach the elements from the OSM standard and transfer them into our new graph data standard.

For the extraction process, one can specify a single key-value pair (i.e. boundary=administrative) or a whole tag group (i.e. boundary). Furthermore, one can execute the two phases together or independently.

3.1 Filtering of OpenStreetMap Elements

In order to be able to achieve a valuable and fast data structure, it is inevitable to filter out unnecessary data. Therefore, we start by reducing the original OSM PBF file into a filtered one only containing the elements needed for the representation of specific tags.

In more detail, we need to extract all elements representing structures with these tags from the source file and copy them into a target file, as shown in figure 3.1. Those required elements contain all elements that include one of the specific tags and all recursively contained elements.

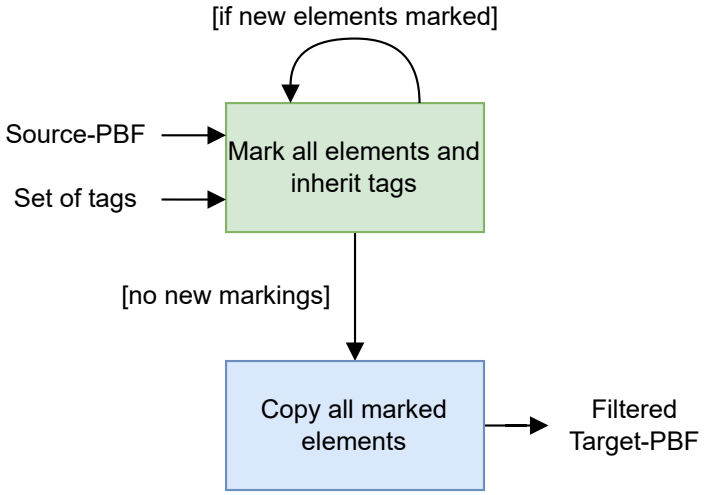


Figure 3.1: Filtering of a PBF file

3.1.1 PBF Reader

For this thesis, we use a Java library called *osm4j* [Kür22] for working with OSM data. It allows the reading and writing of PBF files and the representation of nodes, ways and relations as Java objects.

We will use the *PbfIterator*, which takes an input stream and iterates through entity containers. Each entity container represents a basic OSM element for which we can identify its entity type. We can cast the element into its affiliated class as soon as we know the entity type.

Each of these affiliated classes extends the class *OsmEntity*, which itself offers four different methods. It can return its 64-bit identification number via *getId()*, the number of associated tags via *getNumberOfTags()*, specific tag objects via *getTag(int position)*, as well as some metadata with *getMetadata()*.

The *OsmNode* class supplementary offers two methods *getLongitude()* and *getLatitude()* for the access of its coordinates.

The *OsmWay* class additionally offers a method *getNumberOfNodes()* returning the number of nodes in this way, as well as the method *getNodeId(int position)* returning the node's id at the position inside the way.

The *OsmRelation* expands the *OsmEntity* functionality by a *getNumberOfMembers()* returning the number of contained members and a *getMember(int position)* method returning one specific contained member.

In addition to those classes, there exist two more classes. The first class, *OsmTag*, offers two methods: *getKey()* and *getValue()* to read its key-value pair. Secondly, there exists the *OsmRelationMember* class, which offers a method *getType()* which is similar to the *getEntity()* of the *OsmEntity* class, as well as a *getId()* method returning its identification number and a method *getRole()* returning its role inside the relation.

3.1.2 Marking of Elements

For the filtering of elements, we need to mark all elements we want to copy into the new filtered PBF file. Therefore, we create three lists, one for each basic OSM element type, that store which elements contain the given tag. A single list would not be efficient because the identifications are unique for their specific type but not unique with respect to the elements of different OSM types. We then iterate through all elements in the source PBF file using variants of the code in listing 3.1 and check whether those contain one of the given tags.

Listing 3.1 Method checking whether an *OsmWay* consists of a given tag or tag group

```
/**
 * Checks whether a relation contains a specific tag.
 * If only the tag key is given, it returns true for all tag values.
 *
 * @param element the OSM element to be analyzed
 * @param tag the tag to be searched
 * @return boolean whether tag is contained in the element
 */
public boolean containsTag(final OsmWay element, final String[] tag) {
    int tags = element.getNumberOfTags();
    for (int i = 0; i < tags; i++) {
        if (element.getTag(i).getKey().equals(tag[0])) {
            if (tag.length == 1 || element.getTag(i).getValue().equals(tag[1])) {
                return true;
            }
        }
    }
    return false;
}
```

3.1.3 Inheritance of Markers

After the first round of markings, we marked all elements that explicitly contained one of the given tags. However, making a copy of only those elements would not be enough since their structure builds up on their contained elements. Those dependencies imply that we must recursively inherit markings to all elements contained in a marked element.

Thus, we iterate over the source PBF file and inherit the marking of an element to all of its members as long as at least one heredity exists in an iteration. Since the markings do not disappear when inheriting, there is also no risk of circling in this inheritance process.

As soon as the inheritance process is finished, we read the unfiltered PBF file one more time, check for each element if it is marked, and copy it into a new PBF file if the element is indeed marked. This emerging file is now self-contained and correct, considering the OSM and PBF standards, and we can use it for further computations.

3.1.4 Runtime for PBF Filtering

Given the number of OSM elements N in the PBF file, the average number of tags T for those elements and an inheritance depth D, the first round of marking runs in $O(NT)$ and the inheritance process runs in $O(ND)$. Copying all marked elements into the new file runs in $O(N)$.

3.2 Graph Creation

After successfully computing a PBF file containing only the essential elements, we aim to produce a file format independent of the OSM standard that concentrates on the crucial attributes needed for consistently simplifying administrative boundaries.

This transformation contains four main parts, as shown in figure 3.2. The first thing we do is remove all illegal elements that do not conform to the OSM standard. Afterwards, we determine additional attributes for the edges. Thirdly, we determine the extent of our resulting graph and map all indices such that we have no gaps between indices. As the last step, we write our graph file.

Since steps one and two strongly depend on the attributes one wants in their graph, those steps are only implemented for the *boundary=administrative tag*. The graph creation will perform a basic logic check in step one for all other tag inputs and skip the second step. It will therefore return a graph representation of the filtered source file without any additional attributes for all tags beside *boundary=administrative*.

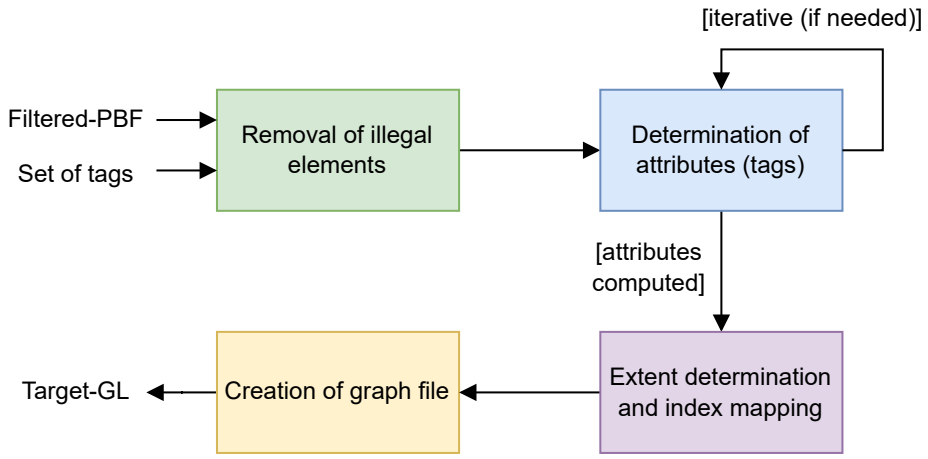


Figure 3.2: Graph creation from a filtered PBF file

3.2.1 Filtering of Illegal Elements

Before building a graph, we want to filter all illegal elements to be able to build a correct graph without dragging issues from the original PBF file.

We check whether a node consists of a legal coordinate pair (lat, lng) with $\text{lat} \in [-90, 90]$ and $\text{lng} \in [-180, 180]$ and whether a way contains between 2 and 2000 edges.

In addition to those checks, we verify the administrative level of OSM when building an administrative boundary graph. In this case, we search the elements for their administrative level using variants of the algorithm shown in listing 3.2 and store them in a list. If an element contains an illegal administrative level or does not contain an administrative level tag, we save -1 as its value to filter it out in the next step. The reason this check is not done in the PBF filtering phase is that there might exist someone who would like to use the filtered PBF file for other purposes for which we, therefore, facilitate those options.

Listing 3.2 Method returning the administrative level of an *OsmWay*

```
/**
 * Returns the administration level for a way If there is no "admin_level" tag
 * it returns -1 (marking for illegal value)
 *
 * @param element the OSM element to be analyzed
 * @return administration level
 */
public byte getAdminLevel(final OsmWay element) {
    int tags = element.getNumberOfTags();
    byte adminLevel = -1;
    for (int i = 0; i < tags; i++) {
        if (element.getTag(i).getKey().equals("admin_level")) {
            try {
                adminLevel = Byte.parseByte(element.getTag(i).getValue());
                if(adminLevel > 12 || adminLevel < 0) {
                    adminLevel = -1;
                }
            } catch (NumberFormatException e) {
                // Has no number as tag
                return adminLevel;
            }
        }
    }
    return adminLevel;
}
```

3.2.2 Determination of Attributes

Since multiple ways or relations using the same edge may exist, we have to decide which administrative level one specific edge should have. We, therefore, create a list in which we write the first occurrence of a legitimate administrative level for this edge. On each subsequent occurrence, we compare the level of the new occurrence with the one saved in the list and store the lower one. This method ensures that each edge has the administrative level of the most significant boundary in which it takes part.

3.2.3 Index Mapping

After removing illegal elements and determining attributes, we remap all identifications of the *OsmNode* and *OsmWay* elements to have a continuous numbering. Thus, we create two *HashMaps*, one for the nodes and one for the ways, and fill them with mappings from old to new identifications. Therefore, the sizes of those maps implicitly define the number of nodes and edges inside the graph we are building.

3.2.4 Creation of a Graph File

We want to create a simple file format containing the extracted nodes and edges with their mapped indices and attributes. Therefore, we design a simple file structure that implicitly assigns indices and stores edges in an adjacency list.

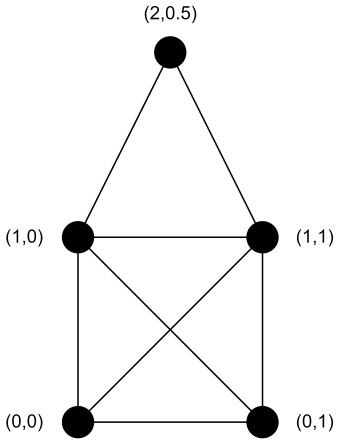
This design consists of four sections. The first section is a line containing the number of nodes in the graph, and the second section is a line containing the number of edges in the graph. The third section is a list of node coordinates, in which each node takes up one line with its latitude and longitude. The last section is an adjacency list of the edges, in which each edge takes up one line with its start node, end node and additional attributes.

Additionally, the position of a node or edge inside its section implicitly defines its index. We also want to mention that one can use the format for directed and undirected graphs. An example of this format can be found in listing 3.3.

With the previously made index mapping and attribute determination, we have everything we need to create such a graph file.

Listing 3.3 Graph file for the House of St. Nicholas

```
5
8
0.0 0.0
0.0 1.0
1.0 0.0
1.0 1.0
2.0 0.5
0 1
0 2
0 3
1 2
1 3
2 3
2 4
2 5
```



3.2.5 Runtime for Graph Creation

Given the number of OSM elements N in the PBF file and the average number of tags T for those elements, the graph creation process runs in $O(NT)$. More specifically, the removal of illegal elements runs in $O(NT)$, while all other steps run in $O(N)$.

4 Graph Simplification

In this chapter, we want to simplify the graph created in the previous chapter. In detail, we want to develop a method that allows accessing different simplification levels without doing a completely new graph simplification each time.

To obtain such a data structure, we will introduce a process for consistent graph simplification and another for unpacking the simplified data. In addition to those processes, we also define an API so that one can access the data structure and retrieve map sections with specific detail levels.

4.1 Data Management

To be able to start a simplification process, we first need to read a graph file and fill up an initial graph object. For this purpose, we define the graph object structure and the graph file reading process in this section.

4.1.1 Graph Reader

The graph reader is based on a simple *BufferedReader* from the Java Class Library and is adapted to read our graph file format. It reads the first two lines to determine the size of the graph and then allocates an empty graph object *GraphData(nodes, currentEdges, maxEdges)*. After this allocation, it starts inserting all nodes and edges into the graph object via its getter and setter operations.

4.1.2 Data Structures

Graph Data Structure

The graph itself is represented as a Java object. A graph object can be initialized with the number of nodes, the maximum number of edges and the current number of edges, which will also be stored inside the object after its initialization. The number of current edges is the number of edges in the graph file, and the number of maximum edges is the maximum number of edges that can be represented by this specific graph object, which will be relevant later.

The coordinates of all nodes are stored in two arrays, one for the latitude and one for the longitude. The index inside the given graph file corresponds to the index in the arrays. Additionally, we define an array that stores the node's degree at the corresponding index.

For the edges, we allocate three arrays, one for the start nodes, one for the end nodes and a third one for the administrative level of the edge. The index of an edge inside those arrays corresponds to the index of the edge inside the given graph file. If the current number of edges and the maximum number of edges are not equal, the additional slots for edges are filled with -1 due to the edge array size being defined by the maximum number of edges.

In addition to this basic graph representation, we also add a couple of status flags and additional data for each edge that are used for the simplification process described later on. That information includes whether an edge is active ($\Leftrightarrow$ not approximated) or minimal ($\Leftrightarrow$ cannot be approximated) and which two edges got approximated by an edge. Initially, after filling the graph, all edges are active, not minimal and do not approximate any other edges.

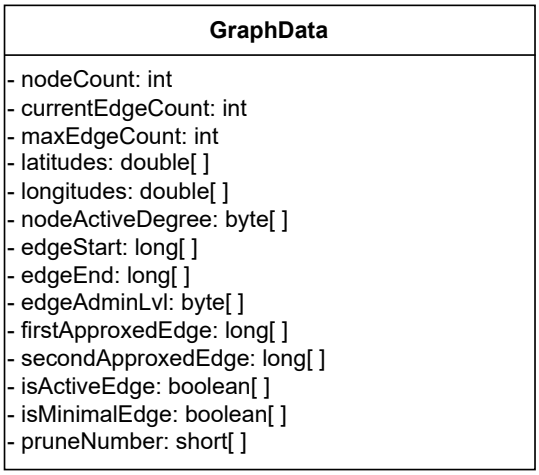


Figure 4.1: Graph class memory layout [1]

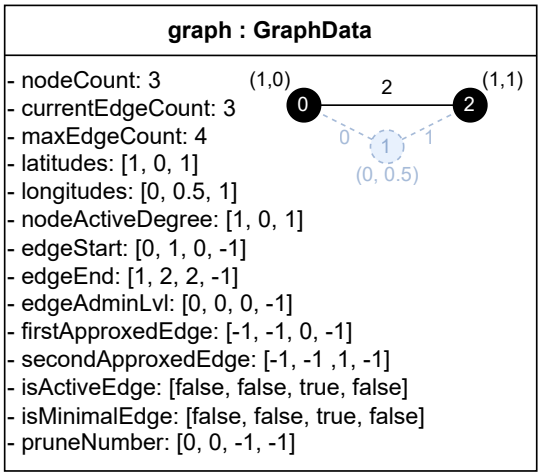


Figure 4.2: Memory layout for an example graph

The graph class offers getters and setters for all of its stored data and a method for recomputing the node degrees. It is also to mention that this degree computation only regards active edges.

Simplification Data Structure

Before we start describing the simplification approaches, we first need to ensure fast computation of the edges of a node and its adjacent nodes. We, therefore, use the combination of offset arrays and data arrays.

The offset arrays consist of N elements, for which the element at index n describes the offset of the node n inside the data arrays.

Given a node index n, we can look up the offset via *offset[n]*. The corresponding data is then contained inside the data array from *data[offset[n]]* to *data[offset[n+1]-1]*. If the index corresponds to the last node index in our graph, instead of computing *offset[n+1]-1*, we compute *offset.length - 1*.

[1] Class diagramm only shows variables but not methods

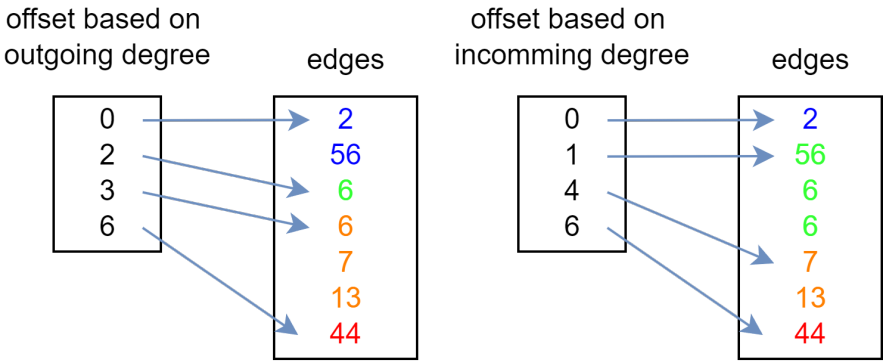


Figure 4.3: Data structure for simplification

For our purposes, we define two offset arrays with one corresponding data array each, as shown in figure 4.3. For the first pair, we define the offsets by the nodes' outgoing degree with the data array containing the corresponding edge indices. We do the same but use the nodes' incoming degree for the second pair's offset. Since we can quickly determine the other node of an edge given the edge index, we do not need to introduce other data arrays for nodes.

The filling of these arrays works in a two-step process. In the first step, we determine the amount of data and compute the offsets. In the second step, we will fill up the data arrays. Both measures run in $O(N + E)$ given the number of nodes N and the number of edges E.

Having established such a data structure, we can now compute all affiliated edges and adjacent nodes of a node quickly and efficiently. Given the node degree d, the offset lookup runs in $O(1)$ and the data lookup in $O(d)$.

4.2 Simplification

We now want to simplify this graph object consistently. In order to achieve this, we will use a simplification method consisting of two strategies: approximation and pruning. We will use the approximation strategy to simplify chains of nodes with degree two and the pruning strategy for the obliteration of subgraphs. Those two strategies will be applied iteratively, simplifying the administrative boundary graph bit by bit. The final simplified graph will then be inserted into a geometrical data structure to enable fast queries for specific map sections.

To be able to allow lossless unpacking, we additionally need to keep track of all simplified data and will therefore create a directed graph of simplifications. This simplification graph will be represented by the two arrays inside the graph object that store which two edges are approximated by an edge.

The idea of this directed simplification graph is the traceability of simplifications. Each edge in our initial administrative boundary graph represents one node in the simplification graph. Each approximation we do is then represented by a new node in the simplification graph with two edges to the nodes we simplified. If a node of this simplification graph has no incoming edges, the

corresponding edge in the simplified administrative boundary graph is minimal; otherwise, it means that the edge has been simplified. An example of this simplification graph idea can be found in figure 4.4.

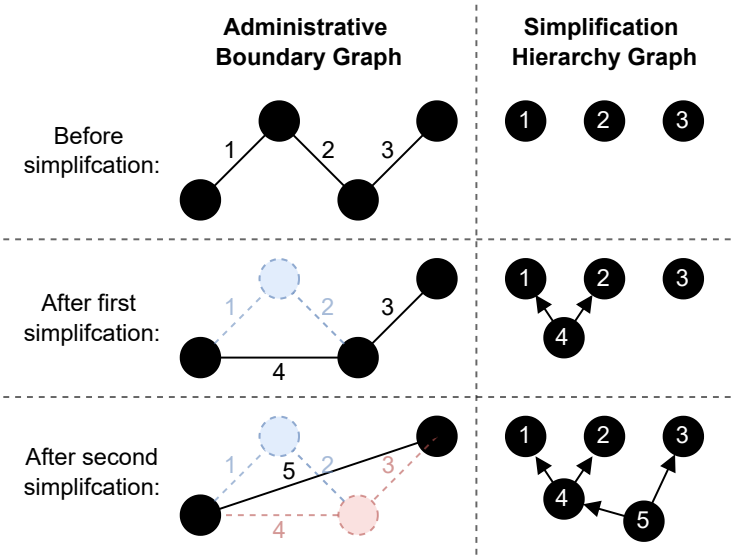


Figure 4.4: Simplification hierarchy graph of a simplified administrative boundary graph

Furthermore, we save a number for each edge in our simplification graph that represents in which simplification iteration the node got simplified. This is needed because even though we know the hierarchy of simplifications, we are unaware of the iteration in which an edge of the administrative graph got simplified.

While we could already reconstruct the initial graph with only the simplified data and the graph of simplifications, we keep all simplified edges and nodes inside the administrative graph object to avoid recomputing those in each unpacking process. This comes with the benefit of a higher unpacking speed but the drawback of higher memory requirements.

Since we will always simplify two edges with a new edge, the maximum number of edges inside a fully simplified graph will therefore be

$$\sum_{i=0}^{\infty} \frac{E}{2^i} = 2E$$

given the initial number of edges E before simplification. Because we do not add or remove nodes in our simplifying process, the number of nodes will constantly be N.

4.2.1 Approximation

Approximation, in our case, deals with simplifying two adjacent edges of the same administrative level that share a common node with degree two. For this simplification, we omit the common node and the two edges, mark those edges as inactive and insert a new edge which will use the other nodes of the simplified edges as its start and end nodes.

The newly created edge will then be inserted into our graph object with its start and end nodes and the administrative level of the edges it simplifies. Further, we extend our simplification graph by adding the simplified edge identification numbers into the two arrays at the index of the newly created edge.

Since many simplifications can be made in a single administrative boundary graph, we are making multiple approximations in a single simplification iteration. We will not approximate any adjacent nodes in a single iteration to keep the simplification consistent and even. Therefore, we will add the start node and end node of the newly created edges into a lock array that ensures that those nodes will not be erased in this same iteration. After each iteration, we will reset the locks and recompute the degree of nodes.

When looking at a single line segment, there often exist multiple edges that can be simplified, raising the question of which simplification may be the best. In most cases, one would see the best simplification as the one that generates a minimal error or works the fastest. In our simplification process, we develop three different approximation approaches that aim to satisfy those expectations of fast and low error simplification.

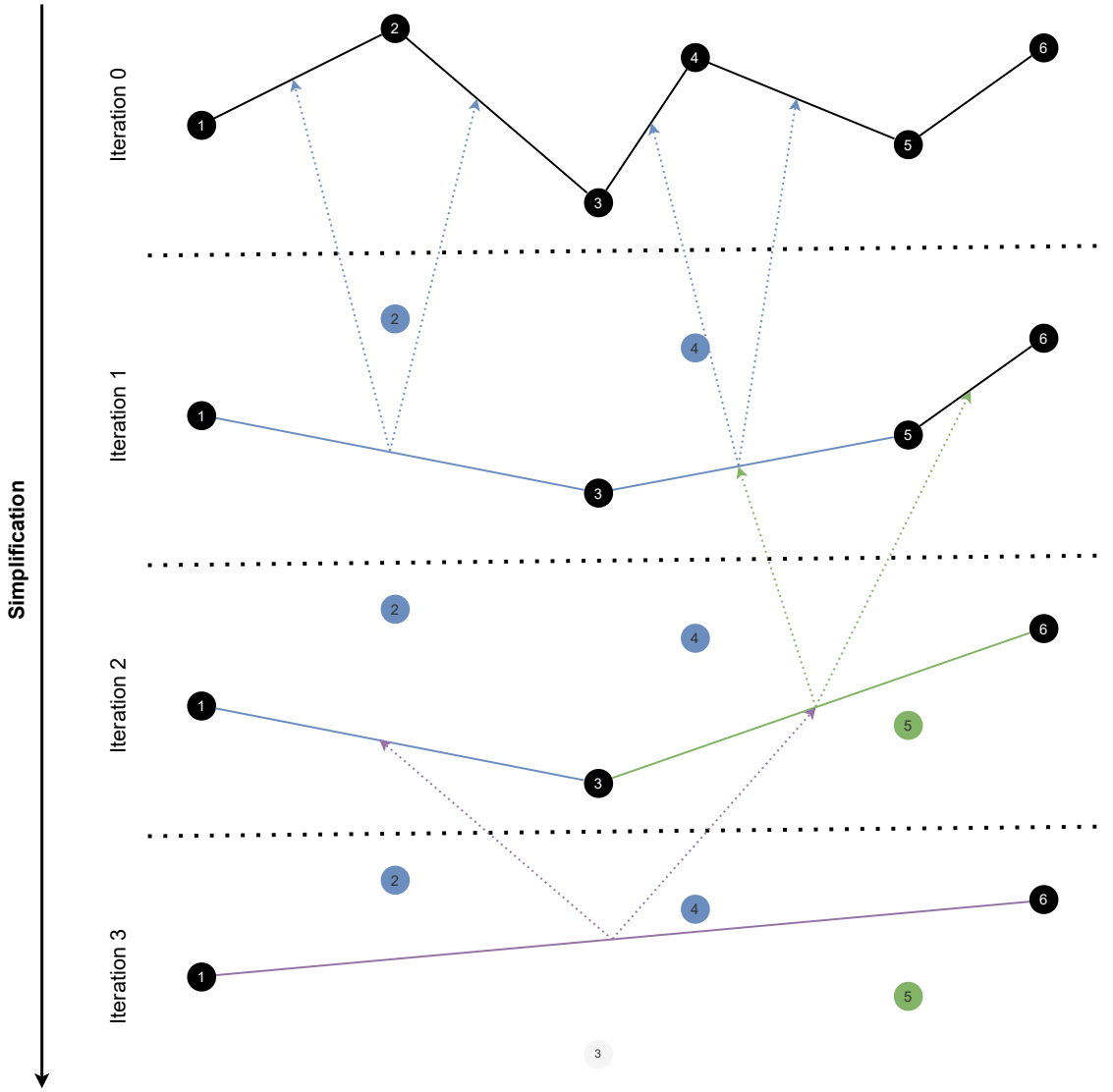


Figure 4.5: Approximation of a simple line segment with five edges over a period three iterations

Fast Approximation

The fast approximation approach is the simplest and most basic approach of the three. It does not try to minimize any occurring error but only tries to maximize the simplification speed.

It iterates through all nodes with degree two and checks whether its edges are of the same administrative level and it is not yet blocked. If this is the case, it immediately simplifies those two edges by adding a new one and blocking the affiliated nodes.

Since we do not check whether this is a good or bad approximation regarding the resulting error, this is the fastest approach we can take.

Surface-Minimizing Approximation

The surface-minimizing approximation approach aims to minimize the generated surface between the old and new edges on an approximation.

To compute the potential surface, we first transform the coordinates of the nodes of the potentially new edge and the removed edges into the euclidean space with Mercator's projection as described in 2.1.3. Subsequently, we compute the euclidean distances between those points and use Heron's formula to compute the surface area of the triangle those points stretch out [Wol22].

Given two points on the WGS84 as $(\text{lat}_1, \text{lng}_1)$ and $(\text{lat}_2, \text{lng}_2)$, we can transform those into radians $(\varphi_1, \lambda_1,)$ and (φ_2, λ_2) with the function ρ_{rad} defined in chapter 2.1.3 and then compute their euclidean distance as

$$\rho_{\text{euclidDist}} : ([-\frac{\pi}{2}, \frac{\pi}{2}] \times [-\pi, \pi])^2 \to \mathbb{R} : \begin{pmatrix}\varphi_1\\ \lambda_1\end{pmatrix} \times \begin{pmatrix}\varphi_2\\ \lambda_2\end{pmatrix} \mapsto \sqrt{(\varphi_1 - \varphi_2)^2 + (\lambda_1 - \lambda_2)^2} \tag{4.1}$$

Given the three euclidian distances *a, b, c*, we can define Heron's formula as

$$\rho_{\text{heron}} : \mathbb{R}^3 \to \mathbb{R} : \begin{pmatrix}a\\ b\\ c\end{pmatrix} \mapsto \sqrt{s(s-a)(s-b)(s-c)} \tag{4.2}$$

where s is a semiperimeter of the triangle; that is,

$$s = \frac{a+b+c}{2} \tag{4.3}$$

After computing the potential area resulting from the approximation, we are taking a look if there exists another adjacent non-blocked node of degree two with edges of the same administrative level in the line we currently approximate. If this is the case, we will also compute its potential error and compare it to the other one to define the better node for approximation out of those two.

We do this comparison iteratively multiple times until we either end at the node that would generate the minimum error on simplification in this specific line or we are stuck because the next node in the line is already blocked. In either case, we approximate this found node, generating only the minimal area error in this line section possible.

Since we only compare the relative surface areas, we do not care about exact results and can further enhance the computation speed by removing the square root of Heron's formula. We, therefore, use

$$\rho_{\text{fastHeron}} : \mathbb{R}^3 \to \mathbb{R} : \begin{pmatrix}a\\ b\\ c\end{pmatrix} \mapsto s(s-a)(s-b)(s-c) \tag{4.4}$$

Distance-Minimizing Approximation

The distance-minimizing approximation approach tries to minimize the distance between the newly inserted edge and the removed node on an approximation.

The basic steps of this approach are similar to the area-minimizing approximation approach. The only difference is that we modify Heron's formula to return the distance between a vertex and its opposite edge in a triangle. We, therefore, use the fact that for the height h and base b of a triangle in euclidean space, we can define the area A of a triangle as

$$A = \frac{bh}{2} \tag{4.5}$$

which we can adjust to receive the height, which corresponds to the distance we want to compute.

$$h = \frac{2A}{b} \tag{4.6}$$

Given three euclidean distances *a, b, c* for which b is the edge representing our potential new inserted edge, we can therefore define

$$\rho_{\text{heronDist}} : \mathbb{R}^3 \to \mathbb{R} : \begin{pmatrix} a \\ b \\ c \end{pmatrix} \mapsto \frac{2\sqrt{s(s-a)(s-b)(s-c)}}{b} \tag{4.7}$$

Since, in this approach, we only need to compare relative euclidean distances, we can further enhance the computation speed by removing the constant and square root. We, therefore, use

$$\rho_{\text{fastHeronDist}} : \mathbb{R}^3 \to \mathbb{R} : \begin{pmatrix} a \\ b \\ c \end{pmatrix} \mapsto \frac{s(s-a)(s-b)(s-c)}{b} \tag{4.8}$$

4.2.2 Pruning

In addition to the approximation strategy, we also apply a pruning strategy. The goal of this strategy is an intervention in situations where approximating is not suitable. This is the case when we either have the issue that we do not have any more nodes with degree two or have small independent subgraphs.

Pruning, in our context, defines the marking of an edge as minimal and inactive. It will, therefore, not be noted in the computation of node degrees and can also not be simplified any further.

Pruning of Administrative Levels

The first case describes the issue that the approximation process is already advanced to a stage where no more approximation possibilities exist in a specific administrative level. At this point, this administrative level is fully simplified and potentially starts preventing other administrative levels from further simplifications.

We define an array containing a simplification flag for each administrative level to solve this issue. The flag is reset after each iteration and is set as soon as an edge of the specific administrative level got simplified. If the flag of an administrative level is not set in an iteration and its subsequent level is already fully simplified, all remaining edges of this level are getting pruned.

Pruning of Independent Subgraphs

The second case is the problem of independent subgraphs. Those subgraphs tend to become simplified quickly and, as a result, often end up in a structure that does visually not represent the original boundary anymore.

To counteract this issue, we insert a pruning test for independent subgraphs before each approximation. This test uses the algorithm from listing 4.1 to build an abstract graph starting at the potentially simplified node. If the generated subgraph is smaller than a beforehand defined number of edges or it is a path with arbitrary length, we are pruning this whole subgraph. In order to keep the runtime reasonable, the algorithm immediately aborts if the graph cannot be a line anymore and already contains too many edges.

Listing 4.1 Method for applying subgraph pruning

```
/**
 * Given a node, prune its corresponding subgraph if needed
 *
 * @param graph the graph on which we operate
 * @param node that needs to be contained in the subgraph
 * @param maxSize the maximum size a subgraph can have for pruning
 */
public void pruneSubgraph(final GraphData graph, final int node, final int maxSize) {
    // Flag if subgraph could be a path
    boolean couldBePath = true;
    // Lists for building up the graph
    ArrayList<Integer> markedEdges = new ArrayList<>();
    ArrayList<Integer> visitedNodes = new ArrayList<>();
    ArrayList<Integer> queuedNodes = new ArrayList<>();
    // Fill up initial node
    queuedNodes.add(node);

    while (!queuedNodes.isEmpty()) {
        int currentNode = queuedNodes.remove(0);
        visitedNodes.add(currentNode);
        int[] edgesOfCurrentNode = computeEdgesOfNode(graph, currentNode);
        // Check if node has degree greater two
        if(edgesOfCurrentNode.length > 2) {
            couldBePath = false;
        }
        // Iterate through edges and update data
        for (int edge : edgesOfCurrentNode) {
            updateSubgraphData(currentNode, queuedNodes, markedEdges);
        }
        // Check if subgraph is already too big
        if(markedEdges.size() > maxSize && couldBePath = false) { return; }
    }
    pruneEdges(markedEdges);
}
```

4.3 Geometrical Data Structure: Grid

After we developed a method to simplify graphs, we now want to be able to insert those into a data structure that allows fast access to the data. More explicitly, we need a data structure that enables data queries for a specific section of the globe. Our approach to tackle this problem is based on a research project from Sophia Heim and Florian Bauer [HB22] and builds up on a simple two-dimensional grid.

For the data structure, we assume that the graph data is placed on the sphere which approximates the WGS84 and lays a grid over this sphere. This approximation of the WGS84 does not affect the correct storing and access but, on the other hand, has the benefit of allowing a direct mapping of the coordinates of a query to a specific cell inside the grid. All following methods are based on a row-major order starting in the bottom-left and ending in the top-right.

As mentioned in the geographical foundations, the sphere has a radius $R = 6,371.0088$km with an equator length of $\approx 40,075$km and a meridian length of $\approx 20,003$km.

4.3.1 Foundations of the Grid

We represent the grid in our project with a Java object and initialize it with *Grid(cellLength, cellHeight, graph)*. While the graph stands for the graph object that will be inserted in the next step, the cell length and cell height define each cell's horizontal and vertical size. In our implementation, we represent those sizes in kilometres; a specification based on latitudes and longitude would also work.

Given those sizes, the amount of needed cells is determined by multiplying the needed cells for the horizontal and vertical directions. For instance, given the cell size 10km × 10km, we would need $\left\lceil \frac{40,075\text{km}}{10\text{km}} \right\rceil = 4,008$ cells in the horizontal direction, $\left\lceil \frac{20,003\text{km}}{10\text{km}} \right\rceil = 2,001$ cells in the vertical direction and therefore $4,008 \cdot 2,001 = 8,020,008$ cells in total. Smaller cells increase insertion time and memory requirements but can, in exchange, reduce query times.

Since the dimensions of the grid are static, and each cell is defined with the same size, we can directly compute the cell in which a coordinate pair (lat, lng) is contained, given with the following formula:

$$\rho_{\text{cellIdx}} : [-90°, 90°] \times [-180°, 180°] \to \mathbb{N} : \begin{pmatrix} \text{lat} \\ \text{lng} \end{pmatrix} \mapsto \text{idx} \tag{4.9}$$

with

$$\text{idx} = \left\lfloor \left\lfloor \frac{\text{lng} + 180}{360} \right\rfloor + \left(\left\lfloor \frac{lat + 90}{180} \right\rfloor \cdot \left\lceil \frac{40,075}{x} \right\rceil \right) \right\rfloor \cdot \left\lceil \frac{20,003}{y} \right\rceil \tag{4.10}$$

in which x defines the horizontal cell length (in km) and y defines the vertical cell length (in km). Since $\left\lceil \frac{40,075}{x} \right\rceil$ and $\left\lceil \frac{20,003}{y} \right\rceil$ are already stored inside the grid object, we are not computing those each time but instead use those constants.

Given two cell indices, we can iterate through all the indices in those cells' Axis-Aligned Bounding Box (AABB) [Kow18]. This means that given a map section defined by two coordinate pairs, we can compute all cells in which elements of this map section could be stored.

Algorithm 4.1 Accessing all cells inside the AABB of two coordinate pairs

```
procedure accessCellsInBoundingBox(lat1,lng1,lat2,lng2)
    firstIndex ← ρ_cellidx(min(lat1, lat2), min(lng1, lng2))
    secondIndex ← ρ_cellidx(max(lat1, lat2), max(lat2, lng2))
    width ← abs((secondIndex mod cellsPerRow) − (firstIndex mod cellsPerRow))
    height ← (⌊secondIndex ÷ cellsPerRow⌋) − (⌊firstIndex ÷ cellsPerRow⌋)
    for i from 0 to height do                                  // Iterate vertically
        for j from 0 to width do                               // Iterate horizontally
            cellIdx ← (firstIndex + ((i ∗ cellsPerRow) + j))
            Access cell
        end for
    end for
end procedure
```

4.3.2 Element Insertion

We are inserting all edges marked as minimal from the given simplified graph. In other words, we want to insert all edges that get represented by a node in the simplification hierarchy graph that has no incoming edges. Since each edge is defined by two nodes, we can use algorithm 4.1 to determine all cells an edge may go through and insert it into all those cells.

However, since the edge will always define the diagonal line of its AABB, there may still be cells in the AABB which should not contain the edge. Therefore, we improve the previous algorithm with an adjusted version of the Cohan-Sutherland algorithm [Kur97] to check whether the edge is contained in a cell.

The algorithm assigns outcode-bits to the two nodes of the edge, with the outcode-bits defined as flags (top, bottom, right, left) relative to the cell. It then first checks whether we have a trivial accept in which one of the edge's nodes is inside the cell (outcode: 0000) or we have a trivial reject because both nodes contain one identical outcode-bit (i.e. 1001 and 0001). If neither of these is the case, the algorithm will check via a clipping mask for which border of the cell we need to apply an intersection test. This intersection test tells us if an intersection exists between the edge and a border of the cell and therefore determines if a part of the edge is indeed contained in the cell.

Since we are only interested in whether an edge is contained, we will terminate the algorithm at this point instead of clipping the edge as one would do in the standard Cohen-Sutherland algorithm.

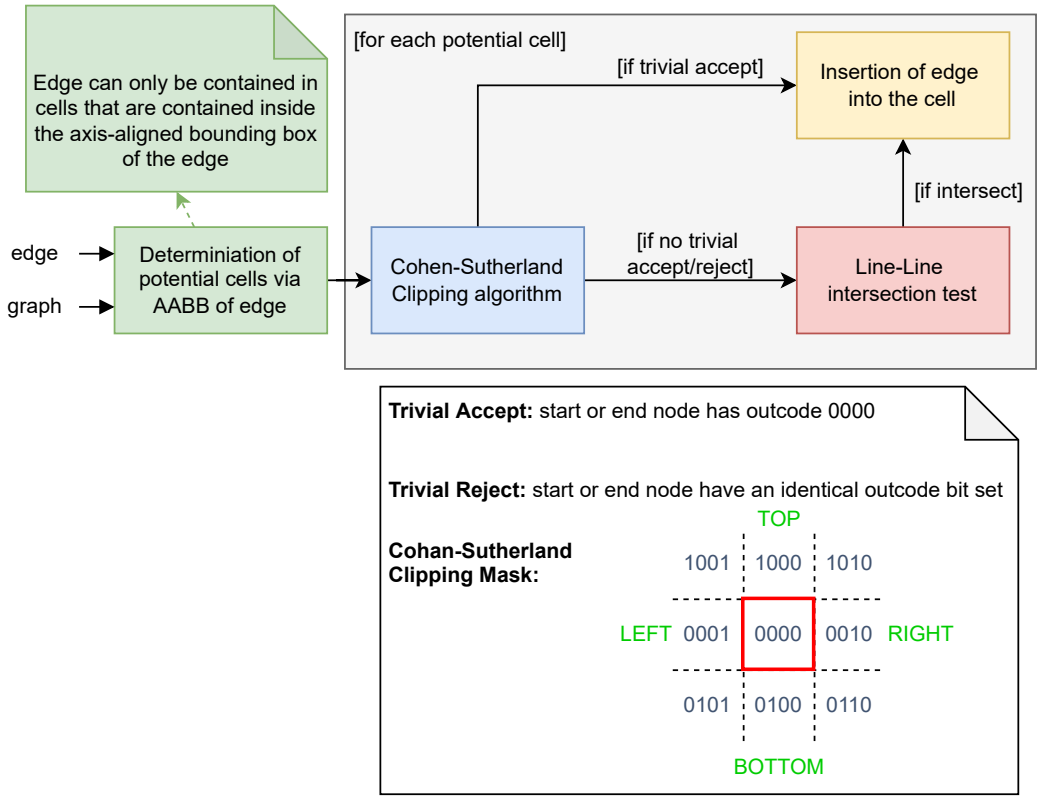


Figure 4.6: Insertion of an edge into the grid

The intersection test is based on Bézier curves and tests whether two finitely long lines, each defined by two points, have an intersection [Kam11]. Given two points for each line, we can define

$$L_1 = \begin{pmatrix} x_1 \\ y_1 \end{pmatrix} + t \begin{pmatrix} x_2 - x_1 \\ y_2 - y_1 \end{pmatrix} \tag{4.11}$$

$$L_2 = \begin{pmatrix} x_3 \\ y_3 \end{pmatrix} + u \begin{pmatrix} x_4 - x_3 \\ y_4 - y_3 \end{pmatrix} \tag{4.12}$$

in terms of first-degree Bézier parameters, for which $t, u \in \mathbb{R}$. We can then find the intersection point by computing

$$t = \frac{(x_1 - x_3)(y_3 - y_4) - (y_1 - y_3)(x_3 - x_4)}{(x_1 - x_2)(y_3 - y_4) - (y_1 - y_2)(x_3 - x_4)} \tag{4.13}$$

$$u = \frac{(x_1 - x_3)(y_1 - y_2) - (y_1 - y_3)(x_1 - x_2)}{(x_1 - x_2)(y_3 - y_4) - (y_1 - y_2)(x_3 - x_4)} \tag{4.14}$$

If $0 \leq t \leq 1$ and $0 \leq u \leq 1$, the two lines defined by the points have an intersection; otherwise, they do not intersect [Ant92].

The storing of edges inside the grid itself works similar to the approach of the graph simplification in section 4.1.2. The offset array contains the offset for each cell, with the data array containing the indices of all elements in the grid. Given a cell index c, the offset can be looked up via *offset[c]*, and all corresponding edges in this cell are then contained inside the data array from *data[offset[c]]* to *data[offset[c+1]-1]*. If c corresponds to the grid's last cell (top-right), instead of computing *offset[c+1]-1*, we compute *offset.length - 1*.

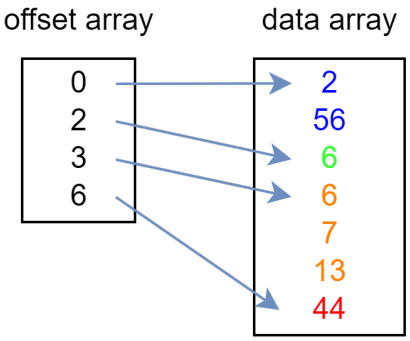


Figure 4.7: Data storage inside the grid

4.3.3 Queries

Querying the grid works similarly to the insertion. The difference is that instead of an edge, we will give two query coordinate pairs that define an AABB for which we want to retrieve the relevant data. We, therefore, iterate through all cells inside this AABB with algorithm 4.1 and return all contained edges.

Since we may be unpacking those retrieved elements after the query, we do not use our Cohan-Sutherland algorithm because this could result in the filtering of edges which, on the one side, are correctly not contained in the AABB. However, on the other side, their ancestors may be contained when unpacking the edges. Therefore, we apply the Cohan-Sutherland algorithm only after unpacking our edges.

4.4 Unpacking

Having inserted a simplified graph into our grid data structure, we now want to describe the unpacking. The unpacking methods deal with restoring details from a set of edges we retrieved from the grid. We can address this task with different approaches based on the simplification graph with the idea of creating a graph focusing on particular attributes.

4.4.1 Unpacking Approaches

While all approaches we define will use the principle of unpacking edges with their ancestors according to the simplification graph, multiple options exist when considering the order and extent of those unpackings. The initial idea behind the unpackings can be seen in figure 4.8.

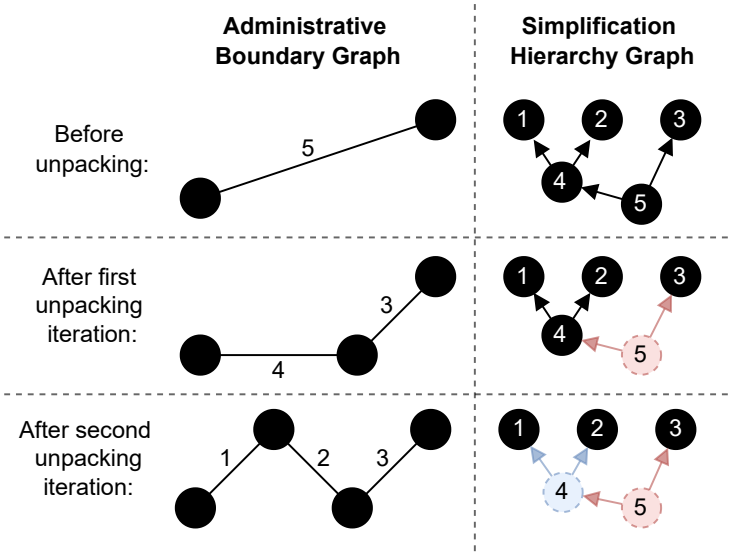


Figure 4.8: Unpacking of edges given a simplification hierarchy graph

Because administrative boundaries generate a hierarchy with their levels, we will consider different unpacking strategies primarily based on those administrative levels. Additionally, we will use the numbers of the iteration in which approximations and pruning took place as a second parameter.

We define the simplification depth as the number of simplification iterations applied on the initial graph. A simplification depth of five would consequently describe the graph's representation after applying a simplification strategy for five iterations.

Each approach takes a graph object and a set of edges contained in this graph object as input. It will unpack those edges according to its defined rules and then apply the modified Cohan-Sutherland algorithm with the query AABB on the resulting set of edges to filter out unwanted edges. Lastly, it will then build a new graph object upon those filtered edges by copying all needed nodes for those edges out of the given graph object.

In addition, one can define a minimum and maximum admin level to filter out undesired administrative levels and thereby potentially reduce the unpacking duration.

Furthermore, we call the result of an unpacking consistent if the following conditions hold.

- For all administrative areas, all edges representing the specific area's administrative boundary form adjacent pairs. If an administrative area forms a cycle in the initial graph, it will be represented as a cycle in the unpacked graph if unpacked consistently.
- All administrative regions sharing a line segment of edges use the same edges to represent this line segment. If two administrative regions share a line segment in the initial graph, they do not create any new regions due to unpacking or simplification errors.

On the other hand, we want to mention that this consistency definition does not imply the planarity of the whole graph object. More specifically, this definition does ensure the consistency of dependent graphs but does not state any behaviour of independent graphs.

Full Unpacking

Full unpacking is our first approach. It does not consider any hierarchy aspects or other limitations and unpacks the whole graph according to the structure given with the simplification graph.

Querying a map section inside the grid and then using the full unpacking approach results in a graph object that is a subgraph of our initial graph. This subgraph contains all nodes and edges of the initial graph in the AABB of the grid query.

Because we always unpack all edges and, as a result, have a subgraph of our initial graph, the resulting graph is consistent.

In figure 4.9, we can see that all administrative levels are getting unpacked from where they got pruned until they are fully unpacked. The same thing happens for all pruned independent subgraphs.

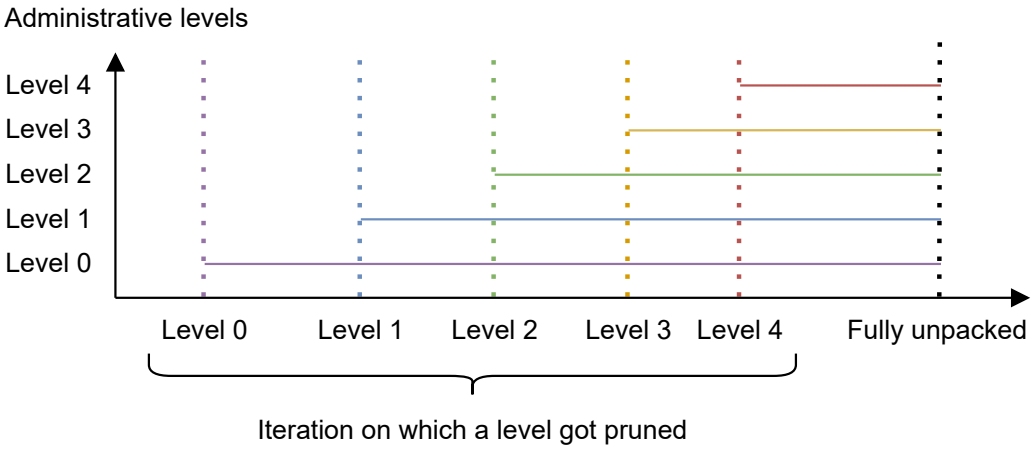


Figure 4.9: Description of the full unpacking approach

Evenly Unpacking

Evenly unpacking is an approach in which we do not consider the simplification depth or administrative levels of edges. The idea is to unpack all edges in the given set for the same number of iterations, independent of their simplification depth or administrative level. For this approach, we define the number of iterations by an unpacking factor. The idea of this approach can be seen in figure 4.10, where we unpack all administrative levels for the same amount of iterations specified by the unpacking factor.

While the evenly unpacking approach is suitable for looking at how the data structure works, this unpacking strategy is inconsistent without introducing additional constraints.

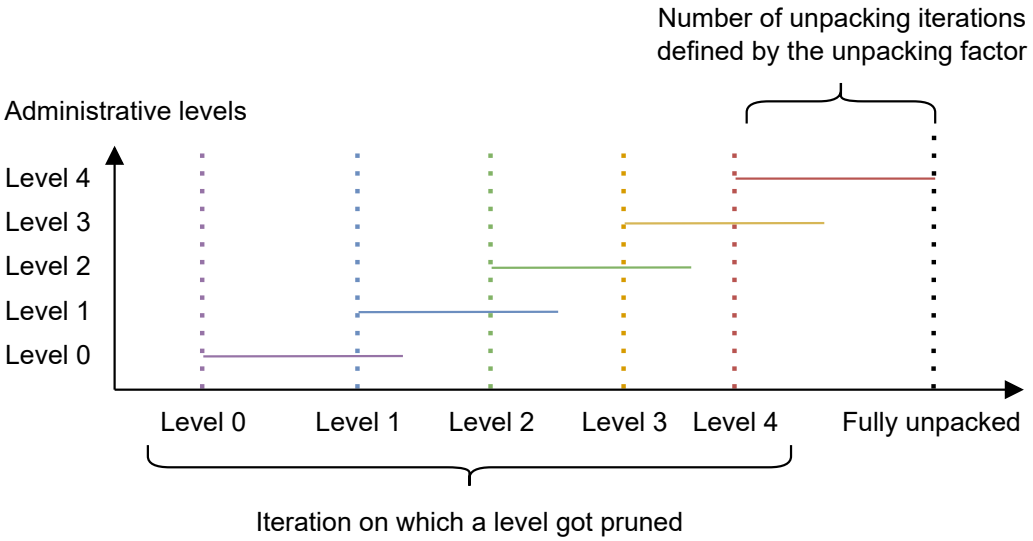


Figure 4.10: Description of the evenly unpacking approach

Waterfall Unpacking

The waterfall unpacking approach focuses on the hierarchy of administrative levels. The unpacking factor in this approach defines the simplification depth to which we will unpack the simplified graph. Its concept is to unpack administrative levels in a waterfall-like pattern so that we start unpacking with the most crucial level. We then include the next level into the unpacking process as soon as the unpacking has reached the simplification depth in which this administrative level got pruned. If edges have been pruned due to the subgraph criteria, we will unpack them as soon as we reach the simplification depth in which they have been pruned.

The principle of this approach can be seen in figure 4.11. We start unpacking the administrative levels as soon as we reach the simplification depth where they got pruned and continue unpacking them until we reach the simplification depth given by the unpacking factor.

Since the set of edges from the grid may already contain edges for which we will not reach the simplification depth in which they got pruned while unpacking, we will filter those out in the process.

This approach results in the principle that more important levels are getting more unpacked than unimportant ones and that we still have consistency due to compliance with all pruning points.

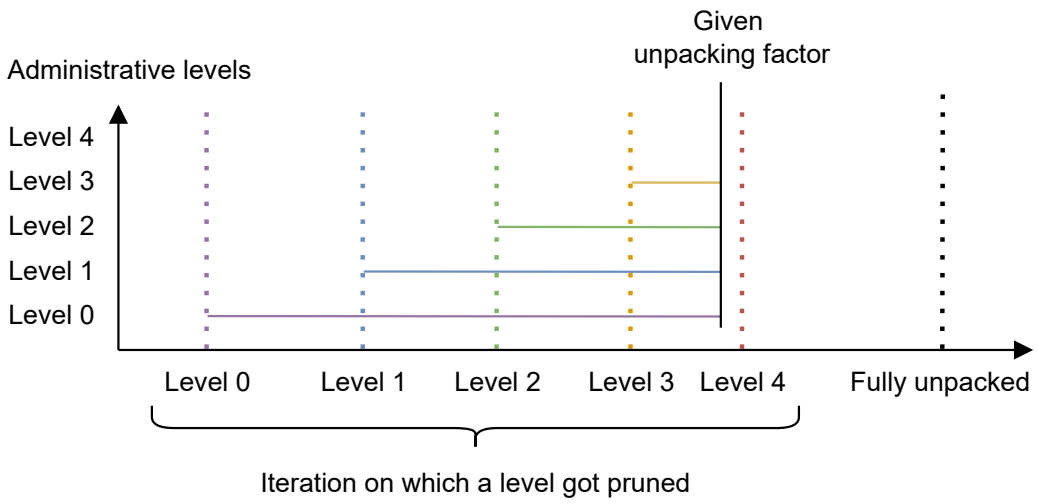


Figure 4.11: Description of the waterfall unpacking approach

Successive Unpacking

We take a similar approach to the waterfall unpacking in our successive unpacking approach. The difference is that instead of adding administrative levels into the unpacking process, we will unpack a specific level to the simplification depth in which the next higher level was pruned. We will only continue unpacking this next particular level and stop unpacking the previous one. As soon as we are at a simplification depth where all administrative levels we are going to return are no longer pruned, we will start unpacking all those administrative levels simultaneously, just like in the evenly unpacking approach.

The difference to the waterfall unpacking approach can be seen in figure 4.12. While the unpacking factor defines the simplification depth to which we will be unpacking, the iterations in which we unpack a specific administrative level differ.

The idea is to unpack all levels to the extent that they are consistent to the point where the next higher administrative level was pruned. After this, we stop unpacking this specific level until all pruning points of all relevant levels have been passed. We then apply an evenly unpacking until we reach the given unpacking factor. The resulting graph is consistent if each pruning of an administrative level only affects the directly underlying administrative level.

If, for example, we choose the unpacking factor so that levels eight and above will not be returned, the evenly unpacking process begins as soon as we reach the simplification depth in which administrative level seven got pruned.

If edges have been pruned due to the subgraph criteria, we will unpack them as soon as we reach the simplification depth in which they got pruned while still considering the administrative level. For instance, if we have a subgraph of edges with administrative level eight that has been pruned due to the subgraph criteria, and therefore way before the pruning of its administrative level, we will start unpacking them when reaching the simplification depth on which they got pruned. Still, we will only unpack them in an unpacking iteration when we would also unpack all other edges of administrative level eight in this iteration.

Suppose the initial set of edges from the grid does contain edges with administrative levels for which the unpacking does not reach the simplification depth in which they have been pruned. In that case, we will do the same thing as in the waterfall approach and filter them.

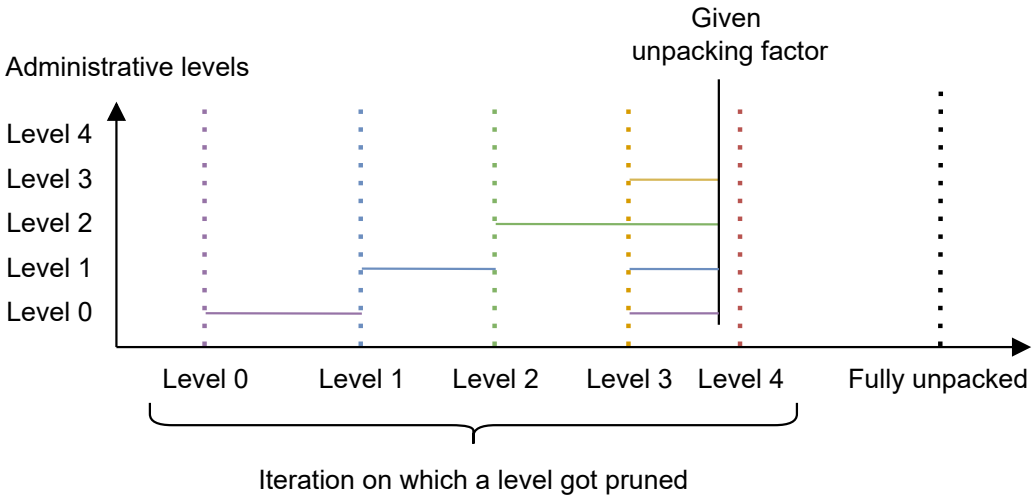


Figure 4.12: Description of the successive unpacking approach

Automatic Unpacking

The last unpacking approach is the automatic unpacking approach. This approach is based on the evenly unpacking approach but is designed for easy usability. We consider the two points $(\text{lat}_1, \text{lng}_1)$ and $(\text{lat}_2, \text{lng}_2)$ used for the grid query to compute the four vertices of the AABB and then use the great-circle distance [Cam17]

$$d = R(\arccos(\sin(\varphi_1)\sin(\varphi_2) + \cos(\varphi_1)\cos(\varphi_2)\cos(|\lambda_2 - \lambda_1|))) \tag{4.15}$$

to compute the distance between two vertices (φ_1, λ_1) and (φ_2, λ_2) of the AABB on the sphere that approximates the WGS84. We do this computation for all edges of the AABB and use those distances to compute the area of the AABB.

This area is then used to define the unpacking factor and the maximum administrative level we will use inside an even approximation. Therefore we define the maximum administrative level as

$$\text{maxAdminLvl} = \lfloor 14 - 0.7 \cdot \log(\text{area}) \rfloor \tag{4.16}$$

and the unpacking factor as

$$\text{unpackingFactor} = \max(0, \lceil 17 - 0.7 \cdot \log(\text{area}) \rceil) \tag{4.17}$$

Those equations have been determined empirically and can be replaced by other terms to create different variants of the automatic unpacking approach.

4.5 Server and Query Handling

After we have established all needed methods and approaches for simplification, data storing and unpacking, we now want to be able to use the combined result of those three aspects in action.

For that reason, we implement a Javalin server [Åse17] with different handlers to allow the users to query simplified graphs. Our server offers three handlers that can be accessed with Hypertext Transfer Protocol / Secure (HTTP/S) GET requests. The status handler (http(s)://ip:port/status) takes no parameters and returns whether the server is up and running. The query handler (http(s)://ip:port/query/params) takes request parameters and returns a computed administrative boundary graph as a GeoJSON object. The evaluation handler (http(s)://ip:port/evaluate/params) does the same as the query handler but returns additional data for evaluation.

4.5.1 Parameters for Queries

A user can request data from our backend with a GET request, which includes the following parameters:

Parameter	Value range	Default Value
firstLatitude	[-90,90]	-90
firstLongitude	[-180,180]	-180
secondLatitude	[-90,90]	90
secondLongitude	[-180,180]	180
unpackingApproach	String	full
unpackingFactor	[0,∞]	0
minAdminLvl	[0,12]	0
maxAdminLvl	[0,12]	12

Table 4.1: Parameters for server queries

A regular request could look like this:

Listing 4.2 A basic HTTP/S request for the server

```
http://129.69.181.66:8081/query/
    ?firstLatitude=48.747459747529554
    &firstLongitude=9.10645365715027
    &secondLatitude=48.745493033968536
    &secondLongitude=9.106099605560305
    &unpackingApproach=automatic
    &unpackingFactor=0
    &minAdminLvl=0
    &maxAdminLvl=12
```

4.5.2 Query Handling

Each query request surpasses four steps before the server answers the request. Those steps are the request parsing, the grid query, the unpacking and the JSON transformation.

Request Parsing

In the first step, we want to parse all the attributes from the request needed to answer it. If any of this data cannot be appropriately parsed or is outside its defined value range, we will replace it with its defined default value.

After the parsing, we have two correct coordinate pairs, an unpacking approach, the unpacking factor and a minimal and maximal administrative level.

Grid Query

Having parsed all parameters, we now query the grid with the two correct coordinates and iterate through all cells the AABB of this query contains. For each cell, we retrieve its content and, in the end, return a set of all edges we retrieved.

Unpacking

After searching for the edges inside the grid, we unpack them according to the unpacking approach, the unpacking factor and the minimal and maximal administrative level included in the request. The result of the unpacking is a new graph object.

JSON Transformation

Since it would not make sense to return any Java object as a response to an HTTP/S request, we transform our graph object into a GeoJSON object. The corresponding method can be found in listing 4.3.

We use the JSON library to create a *JSONObject* containing a feature collection. Each feature represents one administrative level and contains a geometry and property object. For the properties, we insert the administrative level for each feature. For the geometry, we define a *JSONArray* of polylines in which each polyline contains two *JSONArrays* which define the start and end node of the line. Those arrays are then filled with the latitude and longitude of the nodes.

After we fill the feature collection with content, it can then be parsed into a *JSONString* with a *JSONParser* and returned as the response to the request.

4.5.3 Evaluation

The evaluation handler works similar to the query handler with the slight difference that it returns a JSON array with two elements:

- The first element is the regular GeoJSON object that the query handler would return.
- The second element is evaluation data consisting of timings and GeoJSON information.

The timings contain the grid extraction time, unpacking time and the JSON transformation time with the GeoJSON information encompassing the number of nodes and edges of the response and the GeoJSON object size measured in bytes.

Listing 4.3 Transformation from a graph to GeoJSON

```
// Initialize result
JSONObject featureCollection = new JSONObject();
JSONArray features = new JSONArray();

// Create a MultiLine for each administrative level
for (int i = 0; i <= 12; i++) {
    JSONObject feature = new JSONObject();
    JSONObject geometry = new JSONObject();
    JSONObject properties = new JSONObject();
    JSONArray polylines = new JSONArray();
    properties.put("adminLvl", i);

    // Indexing for the edges of a specific administrative level
    int currentIdx = 0;

    for (int j = 0; j < queryGraph.getAmountOfCurrentEdges(); j++) {
        // If line has correct admin level, add to MultiLine
        if (queryGraph.getAdminLvl(j) == i) {
            // Add new polyline and increment index
            polylines.add(currentIdx, createPolyline(graph,j));
            currentIdx++;
        }
    }

    // Pack feature
    geometry.put("coordinates", polylines);
    geometry.put("type", "MultiLineString");
    feature.put("geometry", geometry);
    feature.put("type", "Feature");
    feature.put("properties", properties);
    features.add(feature);
}

// Pack features into collection
featureCollection.put("features", features);
featureCollection.put("type", "FeatureCollection");
```

5 Frontend

In this chapter, we want to introduce an application that uses the API from the graph simplification server. We, therefore, implement a PWA that can generate maps from GeoJSON data.

5.1 Progressive Web Application

As PWA, we use the VueJS [You14] framework as a basis and the Vue Material kit [Tim22]. We add the Vue-Cli Plugin [You18], which introduces service workers and allows downloadability and offline compatibility on various operating systems. Hence, we can use the PWA as a website on a browser or a web application with advanced functionality.

The principle of VueJS is the creation of different views (web pages) that can contain various components. For our purpose, we use two different views for the representation of a simple and an advanced map.

5.2 Map Representation

For the map representation, we use Vue Leaflet [MB16], a variant of the open-source JavaScript library Leaflet [Aga10] made for VueJS. We define a map component that spreads across the whole screen, on which we set a tile layer, two markers and two GeoJSON components.

The map component initially displays the University of Stuttgart $(48.74557°, 9.10623°)$ at a tile zoom level of 17. The view can be changed by holding the left click on the map and dragging around. One can also swipe to achieve the same effect. The zoom is controllable via the mouse wheel or a two-finger swipe gesture.

Our tile layer is bound to a tile server (https://s.tile.openstreetmap.org) and loads the corresponding map tiles for the viewport defined by the map component. In our simple map PWA component, this tile layer has full opacity, while on the advanced map, the opacity starts at 0.5 but can be individually adjusted if needed.

We use the two GeoJSON components to represent the graph data. Those components can display different GeoJSON objects according to a given style function. The style for our graph edges is shown in table 5.1. The style for the simple map is fixed, but one can change all style settings on the advanced map.

A blue marker will appear if one left clicks or touches a point on the map. A click on a marker will make it disappear again. Once two markers are visible, the application requests a GeoJSON object from the backend server and displays it. Another click on the map will remove the markers and the GeoJSON data.

	Administrative Level												
	0	1	2	3	4	5	6	7	8	9	10	11	12
Color													
Thickness	2.5	2.5	2.25	2.25	2.0	2.0	1.75	1.75	1.5	1.5	1.25	1.25	1.0
Opacity	1	1	1	1	1	1	1	1	1	1	1	1	1

Table 5.1: Visual graph settings for the frontend maps

5.3 Usage

Since the PWA should, on the one hand, be as user-friendly as possible but, on the other hand, offer all tools to test the backend data structure in full depth, we define two different map types.

5.3.1 Simple Map

The simple map is the user-friendly variant of the map component. All settings, like the backend server address or the edge style function, are already predefined, and the tile layer is always shown with full opacity. For all requests to the backend, we use the automatic mode, therefore abstracting the complex parameter settings from the user.

5.3.2 Advanced Map

On the other hand, we want to allow the user to individualize all aspects, for which we implement the advanced map. The settings for this map are split into four sections, which we will shortly describe:

- The first section allows the adjustment of the tile layer opacity and the activation and deactivation of the background and multi-mode. Once activating the background mode, the PWA will request the whole map viewport with an automatic unpacking approach from the backend server and draw the response with an opacity of 0.5 until the mode is deactivated. The multi-mode allows drawing multiple queries on the map. Once activated, clicking on the map with two active markers will not remove the rendered graph but only removes the markers. Each new query response will be united with the previous GeoJSON object by combining all edges. In order to be able to reset the graph with an active multi-mode, there exists a reset button inside this section. Once the user exits the multi-mode, clicking on the map with two active markers will reset the previously rendered graph. Initially, background mode and multi-mode are deactivated.
- The second section is the unpacking section, which enables the user to choose between all the different unpacking approaches defined in chapter four. One can also adjust all parameters defined in the API of the backend.

- The attribute section allows adjusting the display settings to someone's liking. Those settings include the color, thickness and opacity of each administrative level. Because the GeoJSON object is not affected by these changes, those display settings can be applied to an already rendered graph without contacting the server again.
- The last section is the backend section, in which one can update the server IP address, port and connection mode (HTTP or HTTPS). It is also possible to activate the evaluation mode there, which, once activated, measures each request's transit time and returns it to the server.

5.4 Functionality

Subsequently, to describe the usage, we will briefly explain the functionality behind the frontend application.

5.4.1 Settings

The mode and unpacking sections primarily work with buttons that switch from activated (green) to deactivated (red). Once clicking a button, the color changes and the corresponding flag will change. All other things in those sections are represented by fields, for which the corresponding variables will automatically adapt their value on each change. The instant adaption of value changes is possible since all of those numbers are independent of each other.

The display and backend sections work with forms the user can fill out and submit. We need this kind of structure because those forms' values depend on each other. For instance, when entering a new color, we first need to know which administrative level a user wants to adjust before we can apply the color change. Once a form is filled out and posted, all corresponding variables to the field inside this form change in one go.

5.4.2 Request Creation and Response Processing

The map component is fitted with an event listener that calls an *addMarker()* method on each click. This method is responsible for rendering a new marker at the given position. If there is already one marker on the map, it calls the *sendRequest()* method. This method then takes the coordinates of the two markers on the map, the server variables' values, and the unpacking section to create an HTTP/S GET request. It then sends the request and assigns the response to our GeoJSON component. The Vue Leaflet library will then asynchronously draw the graph according to our defined style function.

5.4.3 Evaluation Mode

As previously mentioned, the frontend consists of an evaluation mode. When the PWA finds itself in this mode, it will measure the time from sending its request until the complete response has been cached. It will then form a new HTTP request containing this information and send it back to the server.

Figure 5.1: Map representation of New Zealand

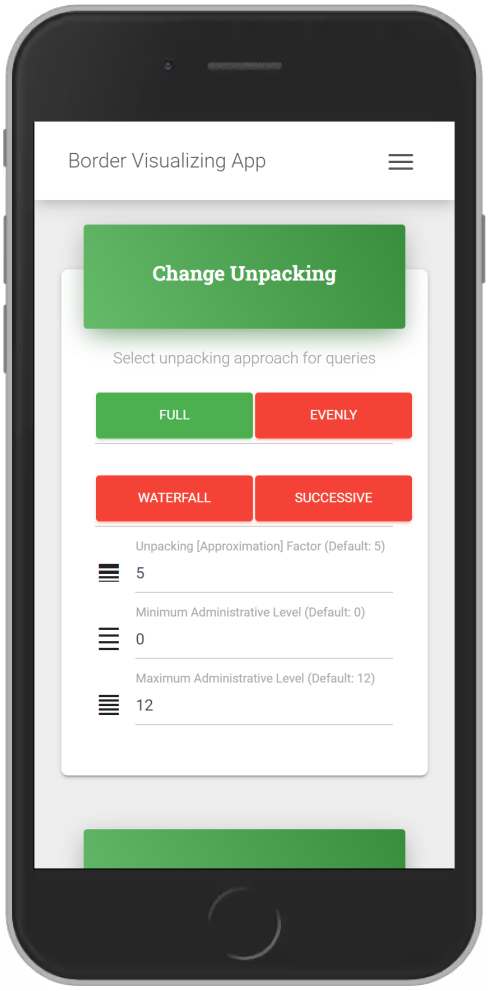


Figure 5.2: Settings in the advanced map

6 Evaluation Proxy

We will use this sixth chapter for a quick explanation of the evaluation proxy that we develop to evaluate the dynamic behaviour between the frontend and backend components.

This proxy can forward and analyze incoming requests from a client but can also generate requests. We, therefore, define a query generation method, a clientful evaluation mode and a clientless evaluation mode.

6.1 Query Generation

We build a query generation as an extended random number generator. Its basic idea is to define a value range for latitude and longitude coordinates and starts generating a specific number of query coordinates. For the randomness, we are using the *ThreadLocalRandom* Java class with the *nextDouble(min,max)* operation. It will then create a file in which two coordinate pairs will be written into one line of the file.

A generated file for the coordinate value range $[47, 55] \times [6, 15]$ (Germany) can look like this:

Listing 6.1 Generated query file with four queries

```
48.09150382882193 9.661063874973845 54.091452204519854 8.987993655936744
50.670937658115506 8.925765624267985 49.65974762560138 11.152858633563564
53.04952500449349 8.044750196336746 52.42945972142893 12.393866441292037
53.07786580812079 12.189336460249706 49.79450354812796 10.05981713599266
```

6.2 Evaluation Modes

The evaluation proxy is capable of two different evaluation modes. The clientless evaluation is designed for automatic stress testing and takes advantage of the query generation. On the contrary, the clientful evaluation relies on real users but can therefore measure in natural environments and gather additional data.

6.2.1 Clientless Evaluation

We can start the clientless evaluation by defining all request parameters except for the coordinate pairs. It will then take a file generated by the query generation and successively write requests to the server's evaluation handler.

It will then receive a response with the GeoJSON object and additional information. It then decodes the response, extracts the relevant information and writes it to a file, such as the one in listing 6.2. We already defined the extraction of data in an evaluation response in section 4.5.3.

Listing 6.2 Evaluation file from clientless evaluation containing the information of five requests

```
/*
 * 1. grid extraction time (in ms)
 * 2. unpacking time (in ms)
 * 3. json transformation time (in ms)
 * 4. amount of nodes in GeoJSON
 * 5. amount of edges in GeoJSON
 * 6. size of GeoJSON (in bytes)
*/
1 0 0 0 0 2134
70 403 65 210960 211633 22646904
54 0 117 210960 211633 22646904
3 0 0 0 0 2134
61 233 99 210960 211633 22646904
```

6.2.2 Clientful Evaluation

In the clientful evaluation, the evaluation proxy works like a real proxy. It receives a regular query from a client, changes it into an evaluation request and then forwards it to the server. The response from the server is then decoded and split into the GeoJSON object and the additional information. The proxy forwards the GeoJSON object to the client and will process the remaining data.

If the client has the evaluation mode activated, the evaluation proxy will receive the transfer duration from the client's perspective transmitted in a subsequent HTTP/S request. As soon as a predetermined number of client requests run through the proxy, it will write an evaluation file and will not analyze any further requests but will still forward the requests to the backend server.

A written evaluation file looks similar to the one from the clientless evaluation, with an added number for the network transfer time in each line.

7 Component Communication and Interfaces

We have developed many different components for different purposes throughout this thesis. As those components cooperate, we now want to show the big picture by combining all of those components.

7.1 Production Mode

The production mode, as seen in figure 7.1, is the normal mode in which users can use the PWA and access the API offered from the backend. In this mode, we firstly filter an OSM PBF file and then transform it into a graph. We then insert this graph into the *Graph Simplification*, which then simplifies and stores it.

A user can then access the PWA from the application host service, download the background tiles from a tile server, and request data from the server. On each request, the server will query its grid, unpack the edges and generate a new graph which will then transform into a GeoJSON object that the server returns to the client.

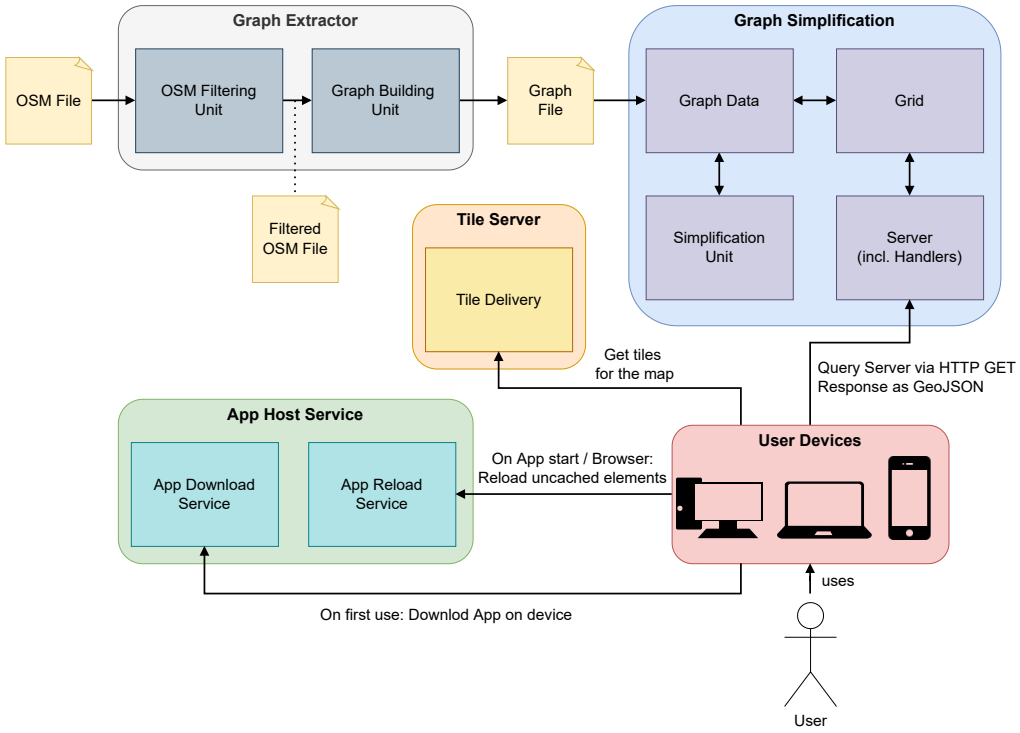


Figure 7.1: Component distribution and interaction in production mode

7.2 Evaluation Mode

As seen in figure 7.2, the evaluation mode is an additional mode used for evaluating the frontend and backend for dynamic analysis. In this case, we insert our proxy between the frontend and backend. Using the proxy in the clientless mode, the components in the yellow zone are not needed, and only the backend is evaluated. However, if the proxy is in the clientful mode, we need all components.

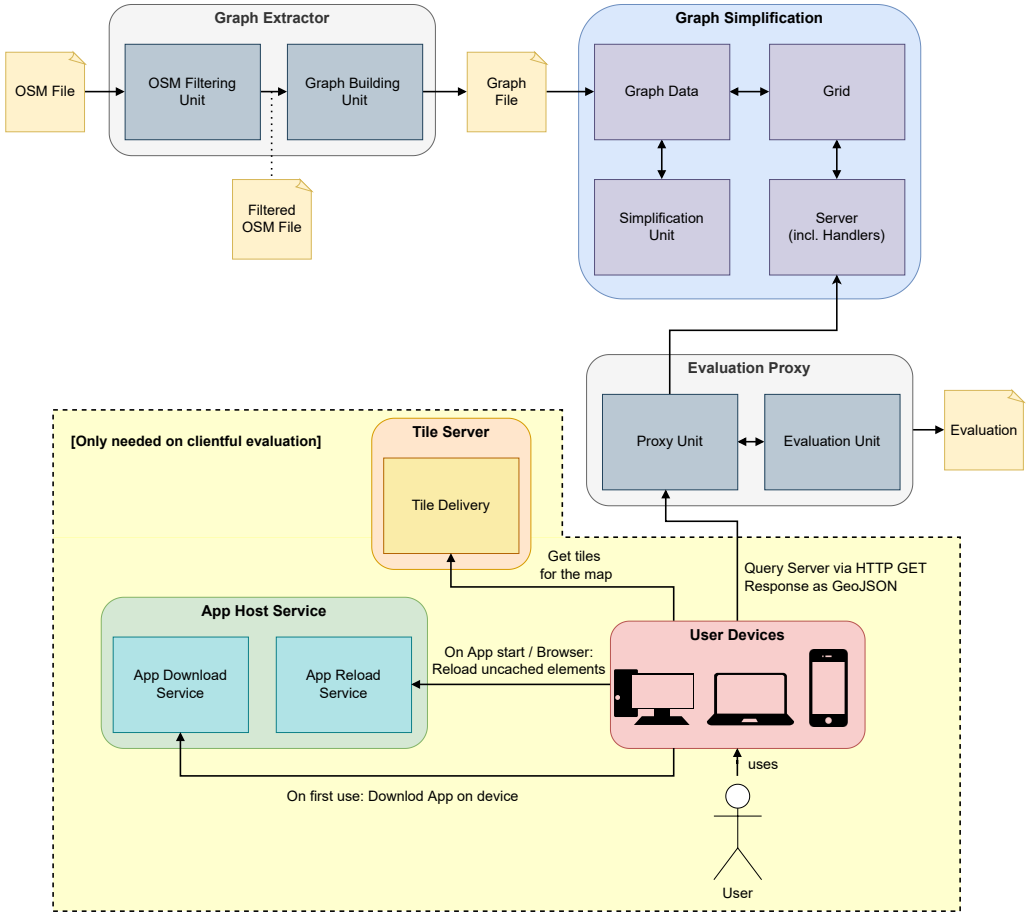


Figure 7.2: Component distribution and interaction in evaluation mode

8 Evaluation

This eighth chapter deals with the evaluation of all developed algorithms, components and collaborations. Therefore, we define our system under test, our evaluation processes, and present the results.

Timing results are measured five times and shown as a closed interval by its shortest and longest results (i.e. [2ms;15ms]). We measure timings on the backend with the Java System class (*System.currentTimeMillis()*) and timings on the frontend and network via the JavaScript DOMHighRestTimeStamp (*performance.now()*). In both variants, we use a time granularity of 1ms. If one is interested in the results of each simplification run, all data is listed in the appendix.

All evaluations are based on the OSM planet database as of 14 May 2022.

8.1 System Under Test

Our system under test contains two different components: the backend component, which we use for graph extraction and simplification, and the frontend component, which simulates the usage of the PWA.

The backend component runs on Ubuntu 64-bit 20.04.4 LTS (GNU/Linux 5.4.0-121-generic). It contains an AMD Ryzen(TM) Threadripper 1950X processor, 256GB DDR4 RAM, an NVIDIA GeForce 8400 GS Rev. 2 graphics card and a Toshiba RD400-400A solid-state drive. The graph extraction and simplification processes were each assigned 240GB RAM on execution.

The frontend component is an emulated mid-tier mobile device from Google Chrome. It consists of a 4x down-throttled Intel(R) Core(TM) i7-3820 with disabled hardware concurrency, 8GB DDR4 RAM and a simulated network speed of HSPA+ with $42\frac{mb}{s}$ (fast 3G).

8.2 Graph Extraction

For the graph extraction, we implemented a static evaluator that measures timings, analyzes the input PBF, the temporal filtered PBF and the resulting graph and writes those results into a solution file. The filtering and extraction phases are subsequently executed as one process for the evaluation. The corresponding results are displayed in table 8.1.

The filtering phase takes [18,935,400ms; 19,186,171ms] (≈ [5.25h; 5.33h]) and the graph creation phase takes [231,559ms; 232,449ms] (≈ [3.85min; 3.87min]).

The original planet PBF file consists of 8,540,264,237 elements, which are filtered down to only 121,548,347 administrative elements, corresponding to a filtration rate of 70.2623. The distribution of elements types also changes from 89,94% nodes, 10,04% ways and 0,1158% relations to 93,14% nodes, 2,002% ways and 0,4856% relations, meaning that the filtered file contains percentage-wise a lot fewer ways but a lot more relations.

When transforming the filtered PBF file into a graph file, the number of edges stays the same, but the 2,433,431 ways are getting divided into their edges, resulting in 111,797,596 edges. This separation corresponds to a factor of 45.94, implying that we have ≈ 46 edges per way on average. Since we do not care about relation structures in the graph representation, the graph does not contain any relations.

File	Number of nodes	Number of ways[1]	Number of relations
Planet PBF	7,672,928,832	857,445,016	9,890,389
Administrative PBF	113,212,594	2,433,431	590,232
Administrative graph	113,212,594	111,797,596	-

Table 8.1: Evaluation of the content of PBF and graph files

8.3 Graph Simplification

For the graph simplification, we implemented a static evaluator measuring timings and analyzing the input graph file, the simplification and the resulting distribution inside the used grid. After the evaluation, the evaluator writes a solution file with all information.

8.3.1 Graph Analysis

The graph created by the graph extraction process is analyzed in further depth in the graph simplification process. This analysis contains the distribution of node degrees and the distribution of administrative levels among the edges.

Degree of Nodes

Looking at the degree of nodes inside our generated graph, as seen in figure 8.1, we have an average of 1.975 edges connected to a node. We also have 2,186,922 nodes with zero or one edge, corresponding to 1.931% of our total nodes. Those phenomena' root cause is the filtration of illegal elements in the graph creation process.

Another aspect we can see is that 99.69% of the nodes are connected to exactly two edges and therefore are only used in a single administrative boundary between two areas. In comparison, only 1.076% are used for connecting different boundaries, having a degree higher than two.

[1] In the administrative graph the ways correspond to edges

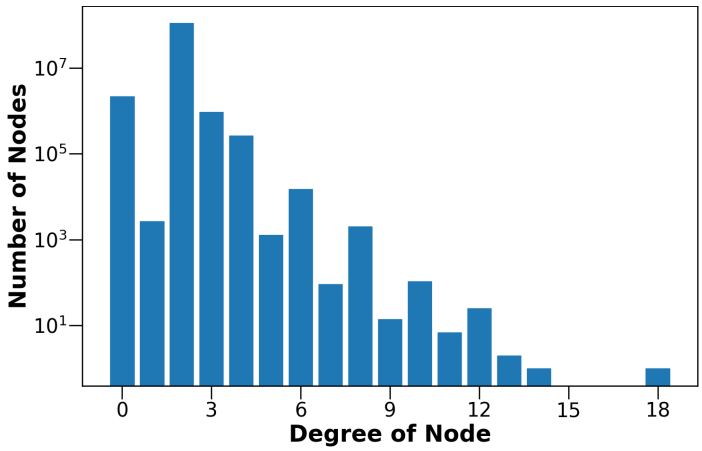


Figure 8.1: Distribution of degrees among the nodes

Administrative Level of Edges

Looking at the distribution in figure 8.2, we can see that the administrative levels are pretty evenly distributed among the levels two to ten, with four, six, eight and ten being the most used levels taking up 77.30% of the total edges.

The percentage-wise distribution overall is very similar to the distribution of administrative levels among OSM data, with the deviations being caused by the division of ways into its edges and the adjustment of the administrative levels of edges to the lowest level of all elements that contain this edge.

The administrative level one only occurs 68 times in the whole administrative boundary graph. This level was formerly used for super-national administrations but is no longer part of the OSM standard. Administrative level twelve is also not part of the official OSM standard but is still used by some countries and occurs 5282 times in our graph. On the other hand, administrative level eleven is part of the official OSM standard, but only some countries, including Germany, use it. We have 213,498 edges at this administrative level, making up only 0.019% of all edges.

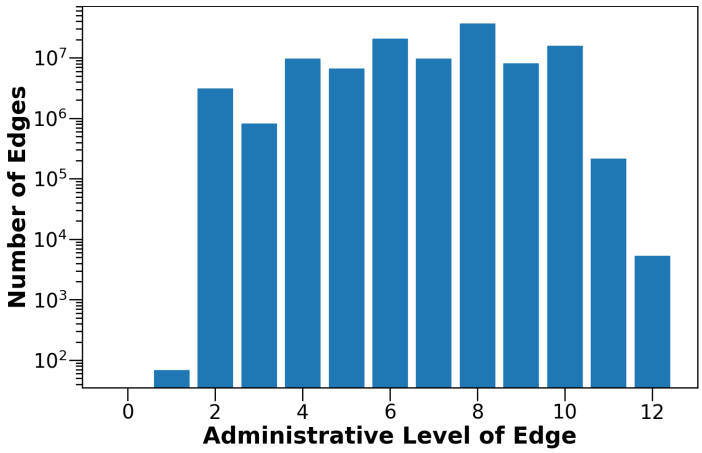


Figure 8.2: Distribution of the administrative levels among the edges

8.3.2 Simplification Strategies

We evaluate the simplification process for all three different simplification strategies. To be able to create comparable results, all strategies run on the same parameters. Those parameters are a 10.0km × 10.0km geometrical grid cell size, resulting in 8,014,006 cells, and *five* as the pruning factor for independent subgraphs.

Simplification Process

When analyzing the simplification process with the three different approaches, we can see in table 8.2 that the duration of graph reading and grid insertion is very similar among the approaches. This similarity is not surprising because the graph reading is independent of the approach and the grid insertion is only dependent on the number of edges that have to be inserted, which is pretty equal in all approaches.

However, the simplification's duration and the simplification depth vary significantly. Those differences are caused by the additional computations needed for the selection of the best simplification in the minimal distance and minimal surface approach, as well as the blocking of nodes resulting in more iterations needed for the simplification.

All in all, the three approaches can build up the data structure in a reasonable time, taking between 360.12s (fastest measurement of the fast approach) and 479.67s (slowest measurement of the minimal surface approach).

Approach	**Phase Durations**			**Depth**
	Graph reading	Simplification	Grid insertion	
Fast	[68.48s; 69.72s]	[288.20s; 295.56s]	[2.44s; 2.59s]	102
Minimal Distance	[68.51s; 70.37s]	[348.67s; 355.28s]	[2.12s; 2.31s]	113
Minimal Surface	[68.44s; 70.21s]	[404.25s; 408.68s]	[2.45s; 2.55s]	115

Table 8.2: Duration and simplification depths of the different simplification approaches

Grid

Looking at the resulting distribution of edges among the cells of the grid, we can see that choice of the simplification approach has little to no effect on the distribution.

In all approaches, the amount of empty cells varies from 7,353,163 (minimal distance approach) to 7,356,995 (fast approach), creating a difference of only 3,832 cells. All other non-empty cells also behave similarly, with insignificant deviations among all three approaches.

Most filled cells only contain a low number of edges, with a small number of cells containing comparatively large numbers of edges. Those cells usually geographically represent large cities or areas with a fine granular administrative boundary structure.

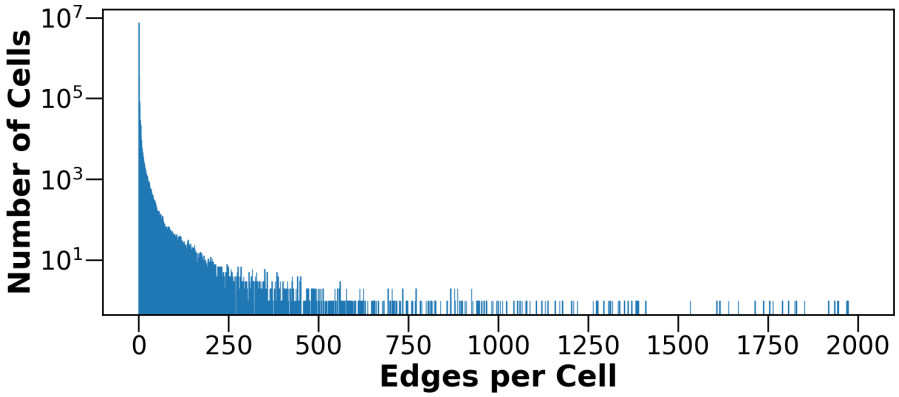


Figure 8.3: Distribution of edges inside the cells of the grid when using the fast simplification approach on an administrative boundary planet graph

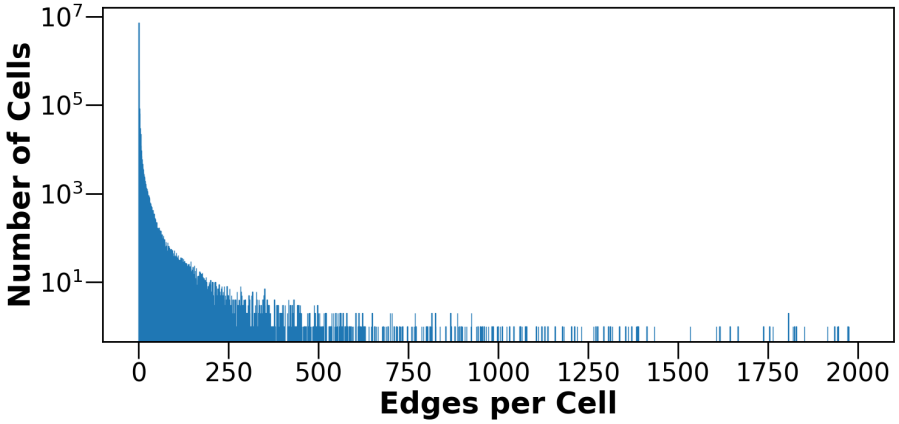


Figure 8.4: Distribution of edges inside the cells of the grid when using the minimal distance approach on an administrative boundary planet graph

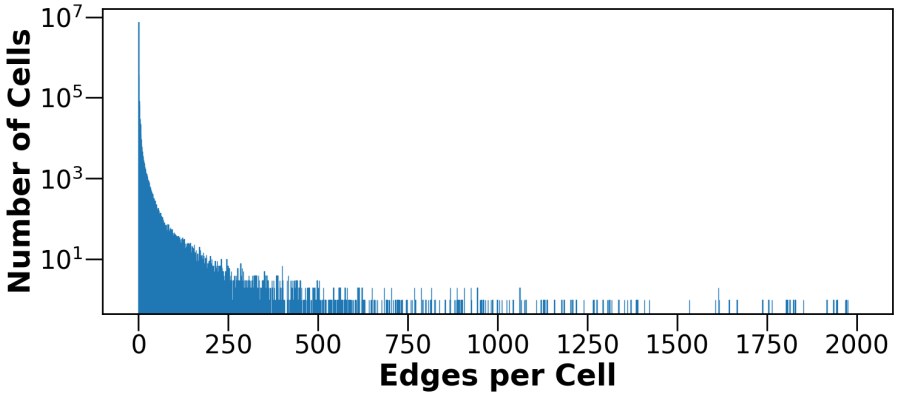


Figure 8.5: Distribution of edges inside the cells of the grid when using the minimal surface approach on an administrative boundary planet graph

8.4 Dynamic Evaluation

The dynamic evaluation deals with the analysis and assessment of all intercomponent interactions. This evaluation includes the query handling of the backend for incoming client requests, the network traffic between client and server and the drawing time on our PWA.

For the backend API evaluation, we will use the query generation from 6.1 and the clientless evaluation mode defined in 6.2.1. For the network times, we use the clinetful evaluation mode from chapter 6.2.2. All requests in the assessment were sent successively to a server, which stores the administrative boundary planet graph fully simplified by the fast simplification approach.

Since we will only feature the most relevant data, one can find more evaluations and figures in the appendix.

8.4.1 Germany

In this evaluation, we restrict our queries to the coordinate value range $[47°, 55°] \times [6°, 15°]$, which are the boundaries of Germany. Nevertheless, we will still access a backend server storing the full administrative boundary planet graph.

When writing requests for the whole area with the coordinates $(47, 6)$ and $(55, 15)$ and all administrative levels, we can see some interesting behaviour when changing the unpacking factor, as seen in figure 8.6. While the automatic unpacking approach works independently of the unpacking factor and always returns the same amount of edges, the evenly unpacking approach increases its edges for each increment of this factor until it comes to a point where it always returns all edges inside the requests AABB.

The waterfall and successive unpacking approaches work the other way around, resulting in fewer edges for a higher unpacking factor. The reason for this behaviour has its root in the definition of the approaches in 4.4.1. We can see that the waterfall and successive unpacking approaches work similarly, with a small amount of disparity at the pruning points of the different administrative levels. At unpacking factor 100, they finally return no edges, meaning all edges are pruned at that stage.

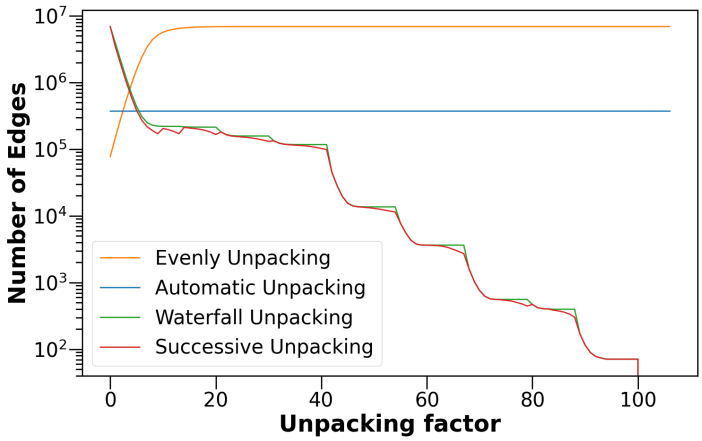


Figure 8.6: Edges returned by the different unpacking approaches for the coordinate pairs $(47, 6)$ and $(55, 15)$, including all administrative levels, when regarding different unpacking factors

Clientless Evaluation

We analysed 1,000 generated requests per unpacking approach and found some remarkable behaviour between the different unpacking approaches. Furthermore, we found that while the number of edges and the unpacking time correlate to the request area, the unpacking time and the number of edges do not correlate.

To measure this, we generated queries and used all approaches to query for the administrative levels zero to twelve on an administrative planet graph fully simplified by the fast simplification approach. The waterfall and successive approaches use an unpacking factor of twelve, and the evenly approach a factor of four.

We found that the graph search time in all four approaches is very low. All 4,000 measured requests have a grid searching time in $[0ms, 9ms]$. Since the search time, however, is independent of the chosen approach, the similarity in data was expected. When taking a closer look at the unpacking duration, as shown in figure 8.7, for the requests, we can see that the automatic unpacking approach is the fastest overall, with the other approaches taking a multiple of its time. The waterfall and successive unpacking approaches are the slowest, taking up to 1640ms for a request. For the transformation duration of the requests, we can see another behaviour. While the automatic, waterfall and successive approaches behave similarly, the evenly approach generally takes much longer.

From another perspective shown in figure 8.8, we can see that the unpacking time and number of edges increase when the request area increases. We can, however, also see that while the waterfall and successive unpacking approaches generally take more time to unpack, the number of edges they return is usually lower than with an evenly unpacking approach. It is also interesting to see how the automatic unpacking approach behaves with increasing request area. Since it automatically adapts to the request area, it returns the most edges for small but generally the least for extensive areas while keeping a mostly steady unpacking time.

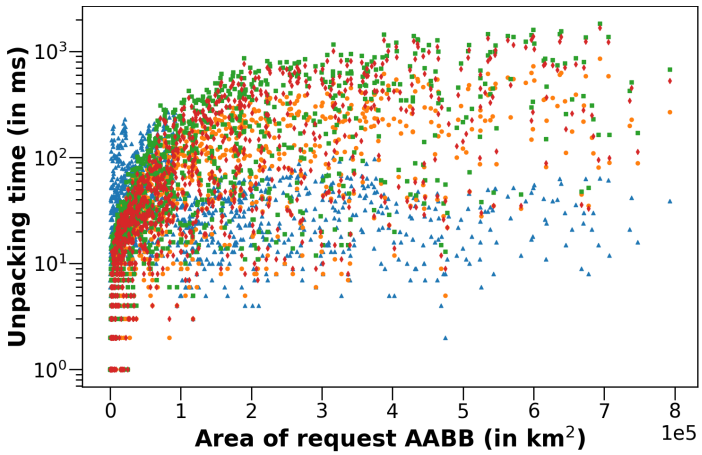


Figure 8.7: Unpacking time of different unpacking approaches for 1,000 requests inside $[47°, 55°] \times [6°, 15°]$ in relation to the query area

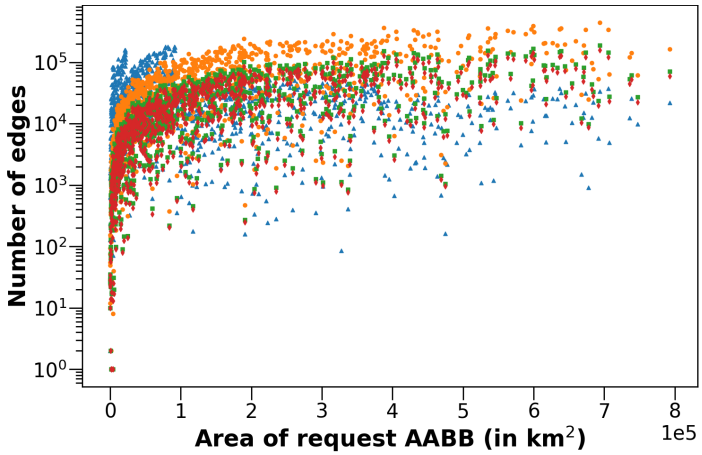
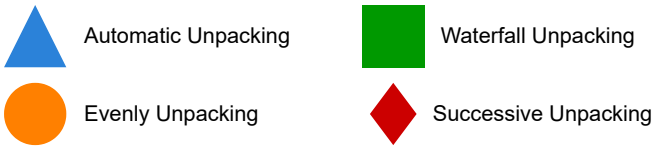


Figure 8.8: Edges returned by different unpacking approaches for 1,000 requests inside $[47°, 55°] \times [6°, 15°]$ in relation to the query area



Clientful Evaluation

For the clientful evaluation, we use the same settings already defined in the clientless evaluation. We generated 30 requests per unpacking approach with the frontend component defined in 8.1.

As shown in figure 8.9, the network transportation time measured is linear to the size of the returned GeoJSON object with minimal deviation.

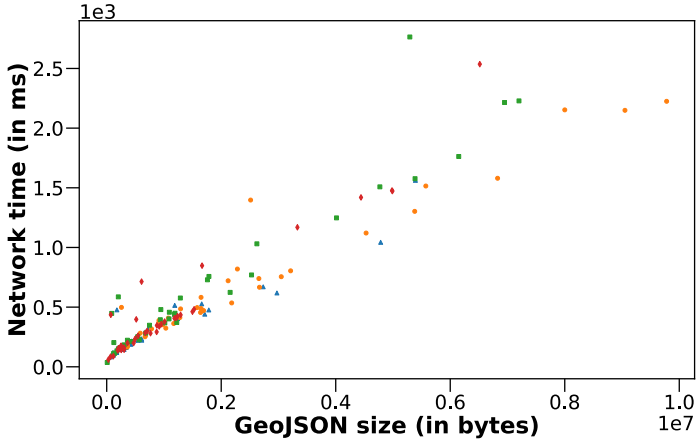


Figure 8.9: Network transportation times for requests inside $[47°, 55°] \times [6°, 15°]$ in relation to the response size

Using the Google Chrome Developer Tools to analyze the rendering times, we can see a correlation between the network transportation times and drawing times. In the analysis of 30 arbitrary requests, we found that the rendering times take $[75.7\%, 91.9\%]$ of the measured network transportation times with a mean of 80.14%. However, this correlation is not a coincidence because the rendering depends on the resource loading time of network requests. Applying those factors to figure 8.9, we can display approximated timings for the rendering time in the dependency of the network transportation time, as seen in figure 8.10.

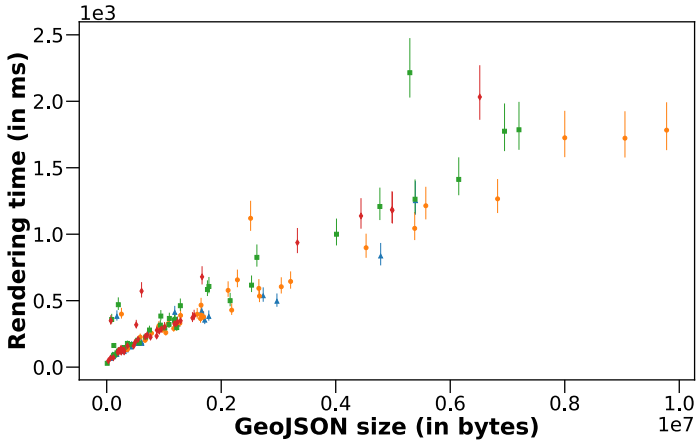


Figure 8.10: Approximated rendering times for requests inside $[47°, 55°] \times [6°, 15°]$ in relation to the response size

8.4.2 Planet

For the planet evaluation, we allow all kinds of queries, therefore restricting the coordinates to $[-90°, 90°] \times [-180°, 180°]$.

Clientless Evaluation

Comparing the measured timings of the different approaches for the same 1,000 generated requests, we can see the same behaviour as we saw when evaluating the Germany map section.

For the measurements of the planet, we used all approaches to query for the administrative levels zero to four on an administrative planet graph fully simplified by the fast simplification approach. The waterfall and successive approaches use an unpacking factor of 65, and the evenly approach a factor of three.

When looking at the grid searching time, all four approaches have the same data with little to no deviation. For the unpacking time, the evenly approach is generally the fastest, minimally faster than the automatic unpacking approach. The waterfall and successive unpacking approaches are very similar, with the waterfall approach being slightly slower overall. In the transformation timings, the tide turns with the waterfall and successive approaches being way faster than the automatic and evenly unpacking approaches.

Moreover, we can see in figure 8.11 that the evenly and automatic unpacking approaches return a higher number of edges when taking the same unpacking time as the waterfall and successive approaches. More specifically, we can see that we can build groups of approaches that behave similar, namely the evenly and automatic approaches as the first group and the waterfall and successive unpacking approaches as the second group. Those groups also correspond to the groupings we see in the unpacking and transformation timings.

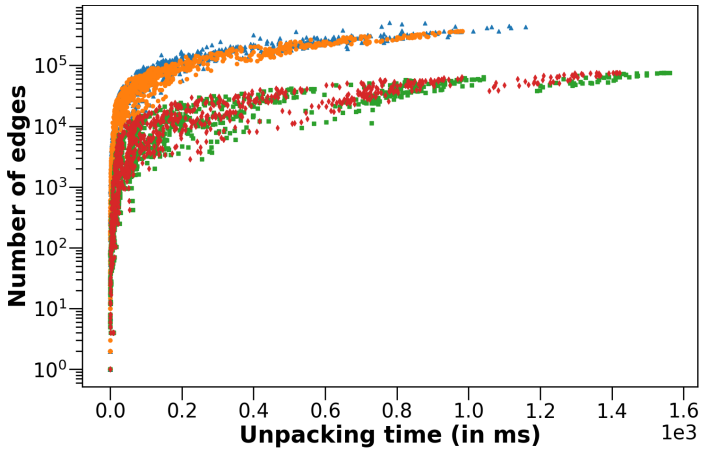


Figure 8.11: Number of edges inside the response of a requests in relation to the unpacking duration for 1,000 requests inside $[-90°, 90°] \times [-180°, 180°]$

Regardless of the previous findings, the transformation from a graph object to a GeoJSON object in figure 8.12 behaves linearly, with a rising number of edges increasing the transformation time. Although most data behaves linear, we can see that some single data points, especially from the automatic unpacking approach, break the trend.

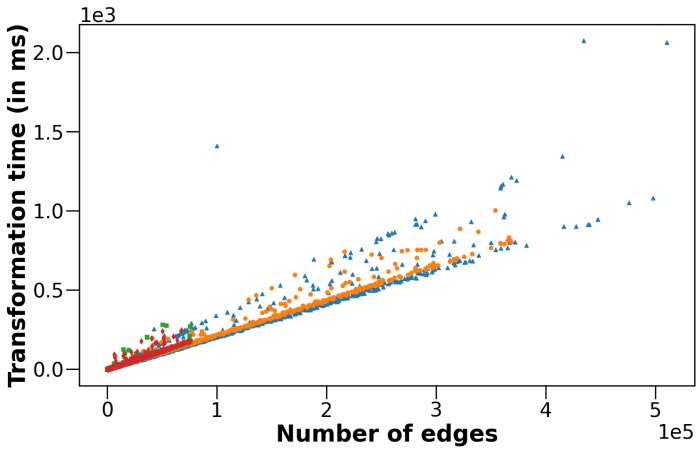


Figure 8.12: Transformation time of requests in relation to the number of edges contained for 1,000 requests inside $[-90°, 90°] \times [-180°, 180°]$

In figure 8.13, we want to show that the response size also behaves linearly regarding the number of edges contained. This time there does not occur any spiking in the data that opposes the trend. However, this is not surprising because the only deviation from a linear ratio would be the number of nodes inside the GeoJSON.

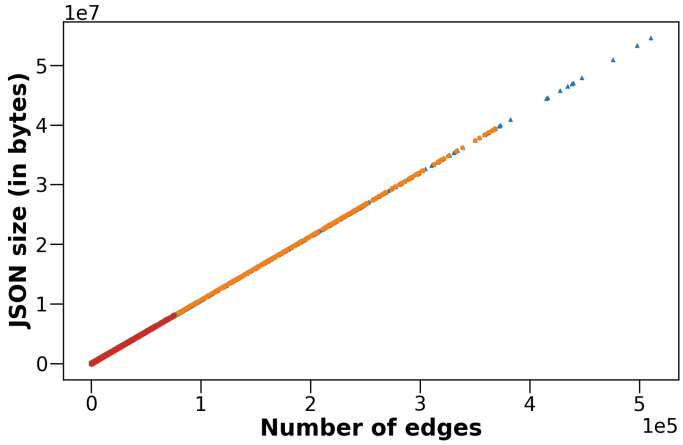


Figure 8.13: Response size of requests in relation to the number of edges contained for 1,000 requests inside $[-90°, 90°] \times [-180°, 180°]$

Clientful Evaluation

For the clientful evaluation, we use the same settings already defined in the clientless evaluation for the planet. We generated 30 requests per unpacking approach with the frontend component defined in section 8.1.

The network transportation in figure 8.14 is mainly linear to the size of the returned GeoJSON object. Nevertheless, the deviation is considerably higher than in the evaluation of the Germany map section.

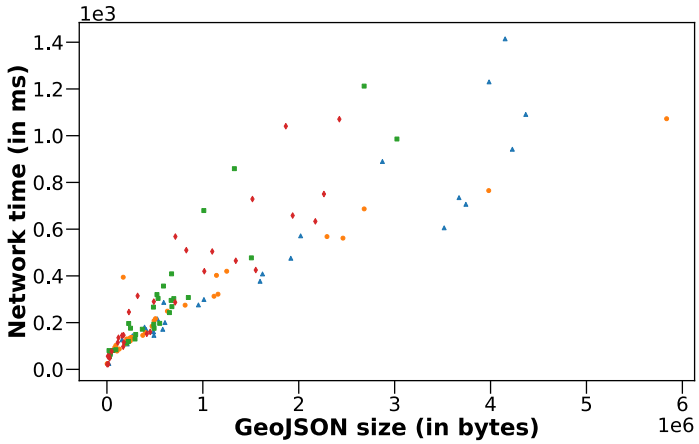


Figure 8.14: Network transportation times for requests inside $[-90°, 90°] \times [-180°, 180°]$ in relation to the response size

We can see the same correlation between the network transportation times and the rendering times that we have already seen in section 8.4.1. To keep the results comparable, we use the same factors $[75.7\%, 91.9\%]$ with a mean of 80.14%. Applying those factors to the network times in figure 8.14 results in the rendering times in figure 8.15.

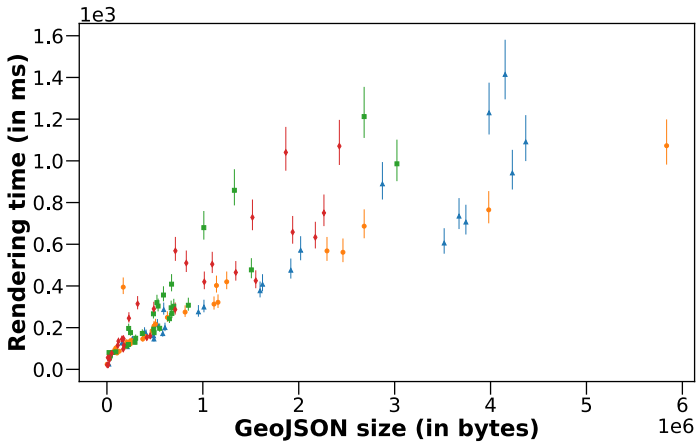


Figure 8.15: Approximated rendering times for requests inside $[-90°, 90°] \times [-180°, 180°]$ in relation to the response size

8.5 Visual Examples

8.5.1 Governorate Stuttgart

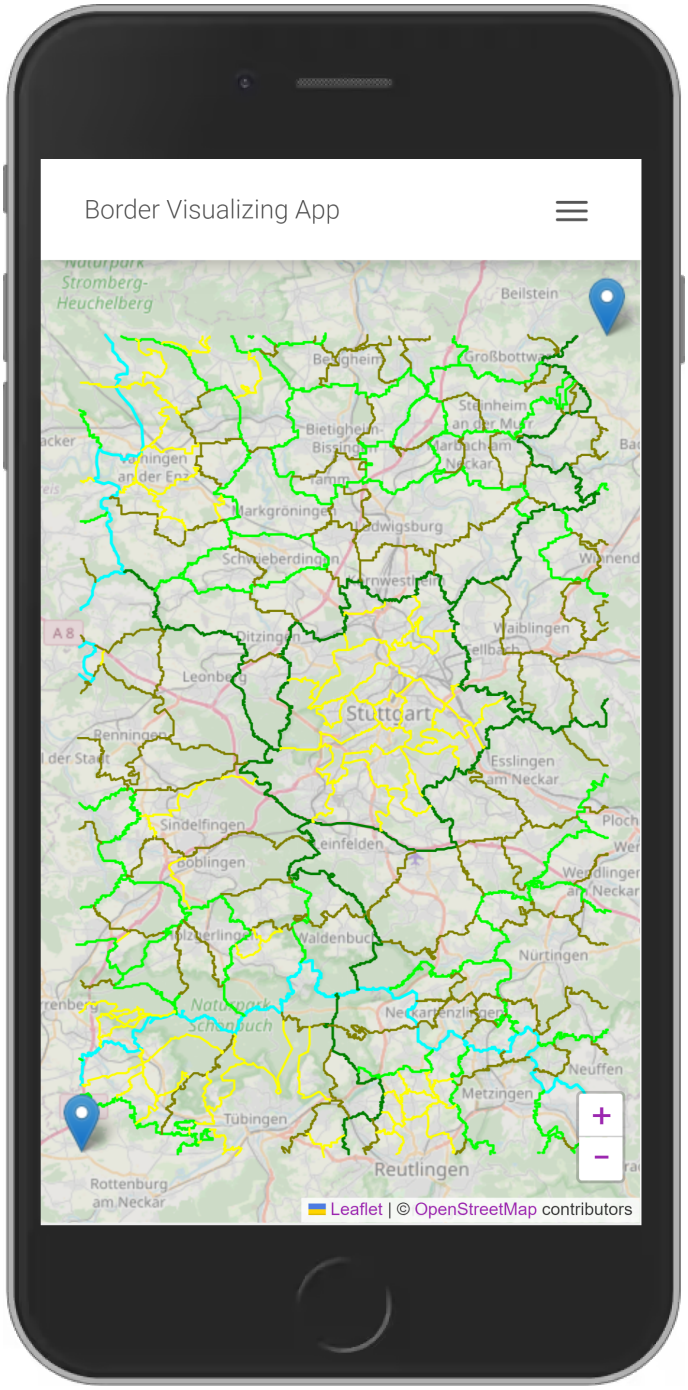


Figure 8.16: Governorate Stuttgart with minimal surface simplification, successive unpacking, an unpacking factor of 25 and administrative levels 0 to 12

8.5.2 Paris with multi-mode Enabled

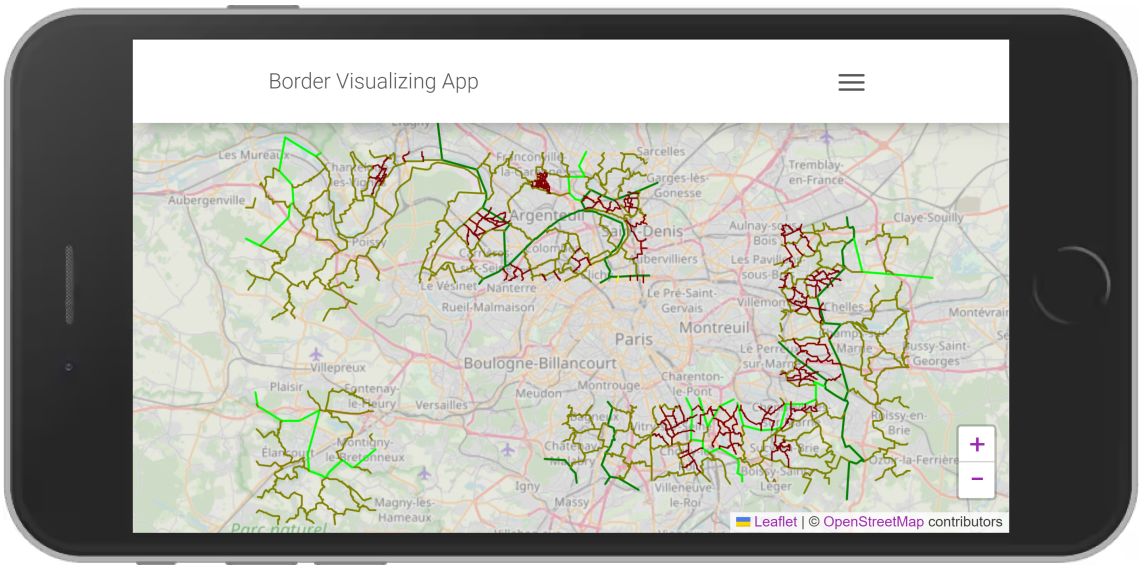


Figure 8.17: Paris with fast simplification, multi-mode activated, the evenly unpacking mode, an unpacking factor of 5 and the administrative levels 0 to 12

8.5.3 United States of America

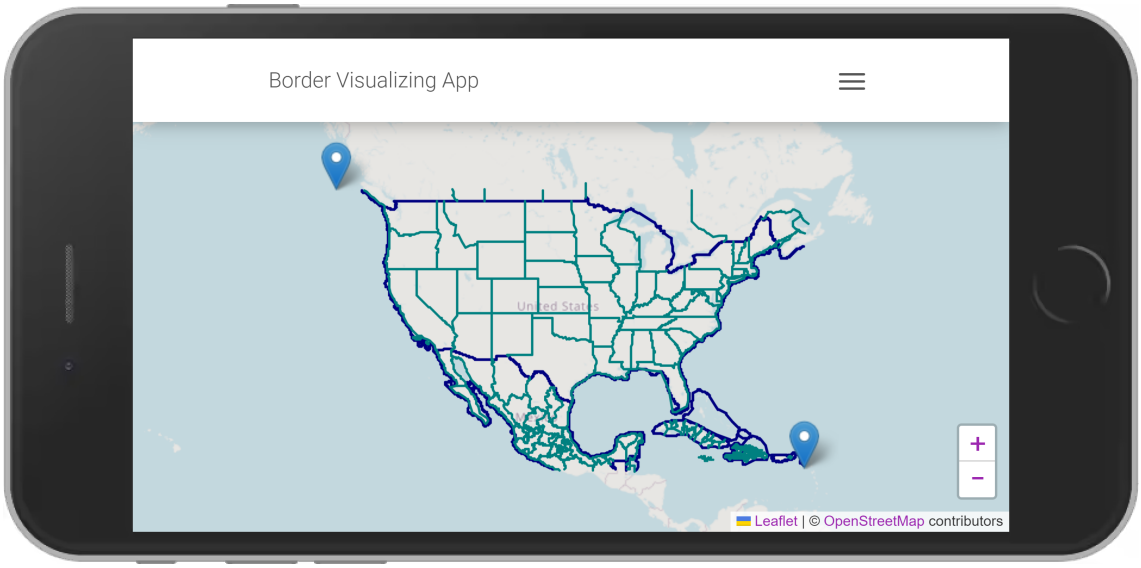


Figure 8.18: United States of America with minimal distance simplification, the waterfall unpacking mode, an unpacking factor of 84 and the administrative levels 0 to 6

9 Conclusion and Outlook

This last chapter contains a brief summary and conclusion of the thesis and a short outlook on future application areas, research fields and improvements of the work.

9.0.1 Conclusion

The target of this work was the development of a consistent simplification method of administrative boundary OSM data. Herefore, we needed to detach the data from the dependencies of OSM standards and find appropriate ways to simplify, store and unpack this data.

In this work, we introduced the foundations of geographical data, the OSM project, the definition of graphs and the foundations of relevant technologies. We then developed a graph extraction tool that extracts OSM elements by tags or tag groups into an independent graph file. After being able to generate graphs, we defined consistent simplification strategies to simplify the generated graphs with different error-minimizing approaches. Subsequently, we introduced a geographical data structure based on a two-dimensional grid to store simplified graphs and allow fast access to specific map sections. We defined a server with an API and implemented different unpacking approaches to retrieve data from the grid and unpack it with the simplification hierarchy graph according to specific strategies. To show how the data structure works with its simplification and unpacking approaches, we implemented a PWA that sends requests to the server's API and renders the returned GeoJSON data. Afterwards, we defined an evaluation proxy for the dynamic assessment of intercomponent interaction and combined all components to describe their interaction and communications. Having all the components defined, we then described our system under test, did a static and dynamic evaluation of all relevant features, and described our results regarding extents, timings and correlations. Lastly, we have shown some examples showing the functionality and result of the work.

Having developed an extraction tool that can generate graphs that no longer depend on the OSM standard, we enable many opportunities to process data, which would not have been feasible before. Our simplification component is a prime example of these opportunities, being able to simplify consistently beyond the borders of OSM elements. Last but not least, our work shows that with efficient approaches and an excellent geometrical data structure, on-the-fly graph creation can be a viable option, especially in a client-server environment.

9.0.2 Outlook

Our work shows that many more opportunities arise with the extraction of OSM data into data formats independent of the OSM standard. Due to our highly modular system, we see massive potential in the possible use of different components in future research. Possible use cases include using our graph extraction component to detach data from the OSM standard, the graph simplification component with its API to deliver simplified graphs on the fly for visualizations or our PWA for the presentation of GeoJSON objects.

While our implementations show that consistent simplifications are possible, the work can be further improved and extended in future work. An instance of improving this work would be implementing parallel processing to increase performance further. Potential areas of future research could, therefore, include evaluating other simplification and unpacking approaches and exploring simplifications for non-planar or independent graphs.

Bibliography

[Aga10] V. Agafonkin. *An open-source JavaScript library for interactive maps*. 2010. URL: https://leafletjs.com/ (cit. on pp. 29, 59).

[Ant92] F. Antonio. "IV.6 - FASTER LINE SEGMENT INTERSECTION". In: *Graphics Gems III (IBM Version)*. Ed. by D. KIRK. San Francisco: Morgan Kaufmann, 1992, pp. 199–202. ISBN: 978-0-12-409673-8. DOI: https://doi.org/10.1016/B978-0-08-050755-2.50045-2. URL: https://www.sciencedirect.com/science/article/pii/B9780080507552500452 (cit. on p. 49).

[Åse17] D. Åse. *A lightweight java and Kotlin web framework. create rest apis in Java or Kotlin easily.* 2017. URL: https://javalin.io/ (cit. on p. 55).

[BDD+16] H. Butler, M. Daly, A. Doyle, S. Gillies, S. Hagen, T. Schaub. *The GEOJSON format*. Aug. 2016. URL: https://www.rfc-editor.org/rfc/rfc7946 (cit. on p. 29).

[Cam17] U. of Cambridge. *The Great Circle Distance Add to Your Resource Collection remove from your resource collection add notes to this resource view your notes for this resource*. 2017. URL: https://undergroundmathematics.org/trigonometry-compound-angles/the-great-circle-distance (cit. on p. 55).

[Cha19] K.-T. Chang. *Introduction to geographic information systems*. McGraw-Hill Education, 2019 (cit. on p. 21).

[CM17] D. Crockford, C. Morningstar. *Standard ECMA-404 The JSON Data Interchange Syntax*. 2017. DOI: 10.13140/RG.2.2.28181.14560 (cit. on p. 28).

[con04] O. contributors. 2004. URL: https://www.openstreetmap.org/ (cit. on p. 23).

[con22a] M. contributors. *Progressive web apps (pwas)*. 2022. URL: https://developer.mozilla.org/en-US/docs/Web/Progressive_web_apps (cit. on p. 30).

[con22b] O. contributors. *Node*. 2022. URL: https://wiki.openstreetmap.org/wiki/Node (cit. on p. 24).

[con22c] O. contributors. *OSM XML*. 2022. URL: https://wiki.openstreetmap.org/wiki/OSM_XML (cit. on p. 23).

[con22d] O. contributors. *PBF format*. 2022. URL: https://wiki.openstreetmap.org/wiki/PBF_Format (cit. on p. 24).

[con22e] O. contributors. *Relation*. 2022. URL: https://wiki.openstreetmap.org/wiki/Relation (cit. on pp. 25, 26).

[con22f] O. contributors. *Tag:boundary=administrative*. 2022. URL: https://wiki.openstreetmap.org/wiki/Tag:boundary%3Dadministrative (cit. on p. 27).

[con22g] O. contributors. *Tag:boundary=administrative*. 2022. URL: https://wiki.openstreetmap.org/wiki/Tag:boundary%3Dadministrative#11_admin_level_values_for_specific_countries (cit. on p. 27).

[con22h] O. contributors. *Tags*. 2022. URL: https://wiki.openstreetmap.org/wiki/Tags (cit. on p. 26).

[con22i] O. contributors. *Way*. 2022. URL: https://wiki.openstreetmap.org/wiki/Way (cit. on p. 25).

[con22j] W. contributors. *Progressive web app*. Sept. 2022. URL: https://en.wikipedia.org/wiki/Progressive_web_app (cit. on p. 30).

[HB22] S. Heim, F. Bauer. *Analysis of OpenStreetMap Data Items*. 2022 (cit. on p. 46).

[Kam11] M. P. Kamermans. *A primer on Bézier curves*. 2011. URL: https://pomax.github.io/bezierinfo/ (cit. on p. 49).

[Kow18] W. Kowarschick. *Multimedia-Programmierung*. Hochschule Augsburg, Fakultät für Informatik, gehalten im Wintersemester 2018. Augsburg, 2018. URL: https://glossar.hs-augsburg.de/Axis_Aligned_Bounding_Box (cit. on p. 47).

[Kür22] S. Kürten. *OSM4J - topobyte.de*. 2022. URL: https://www.topobyte.de/projects/osm4j.html (cit. on p. 32).

[Kur97] J. Kurhila. *Cohen-Sutherland Line Clipping*. 1997. URL: https://www.cs.helsinki.fi/group/goa/viewing/leikkaus/lineClip.html (cit. on p. 48).

[Mat22] W. Mathworld. *Mercator projection*. 2022. URL: https://mathworld.wolfram.com/MercatorProjection.html (cit. on p. 22).

[MB16] N. M. Mezzopera, M. Bouchaud. *Vue Leaflet*. 2016. URL: https://vue2-leaflet.netlify.app/ (cit. on p. 59).

[Nat00] National Imagery and Mapping Agency. *Department of Defense World Geodetic System 1984: its definition and relationships with local geodetic systems*. Tech. rep. TR8350.2. St. Louis, MO, USA: National Imagery and Mapping Agency, Jan. 2000. URL: http://earth-info.nga.mil/GandG/publications/tr8350.2/tr8350_2.html (cit. on p. 22).

[Str11] D. R. Strebe. File: Mercator_projection_Square.jpg. 2011. URL: https://commons.wikimedia.org/w/index.php?curid=17700069 (cit. on p. 23).

[Tec20] K. Technologies. *EPSG:4326*. 2020. URL: https://epsg.io/4326 (cit. on p. 22).

[Tim22] C. Tim. *Creativetimofficial/vue-material-kit: Vue material kit - open source material design UI kit*. 2022. URL: https://github.com/creativetimofficial/vue-material-kit (cit. on p. 59).

[Tru15] R. J. Trudeau. *Introduction to graph theory*. Dover Publications, 2015 (cit. on p. 27).

[Wol22] M. Wolfram. *Heron's formula*. 2022. URL: https://mathworld.wolfram.com/HeronsFormula.html (cit. on p. 43).

[You14] E. You. *The progressivejavascript framework*. 2014. URL: https://vuejs.org/ (cit. on pp. 30, 59).

[You18] E. You. *Vue Cli*. 2018. URL: https://cli.vuejs.org/ (cit. on p. 59).

All links were last followed on October 14, 2022.

A Appendix

Administrative boundary planet graph simplification. Table A.1 shows all collected data runs of the three simplification approaches defined in section 4.2. All runs have been performed with a 10.0km × 10.0km grid cell size. The fastest and slowest timings of all test runs of one approach are marked in green (fastest) and red (slowest) colors.

Test Run	Phase Durations			Depth
	Graph reading	Simplification	Grid insertion	
1	69.55s	290.38s	2.52s	102
2	68.48s	289.57s	2.44s	102
3	69.72s	295.56s	2.57s	102
4	68.95s	293.28s	2.59s	102
5	68.88s	288.20s	2.50s	102

(a) Fast simplification approach simplification run timings

Test Run	Phase Durations			Depth
	Graph reading	Simplification	Grid insertion	
1	68.51s	349.21s	2.26s	113
2	69.43s	348.67s	2.12s	113
3	69.56s	352,81s	2.31s	113
4	70.21s	354.72s	2.20s	113
5	68.88s	355.28s	2.30s	113

(b) Distance-Minimizing approach simplification run timings

Test Run	Phase Durations			Depth
	Graph reading	Simplification	Grid insertion	
1	69.51s	405.38s	2.52s	115
2	68.97s	408.68s	2.53s	115
3	70.02s	407.33s	2.55s	115
4	70.21s	406.22s	2.48	115
5	68.44s	404.25s	2.45s	115

(c) Surface-Minimizing approach simplification run timings

Table A.1: Static time analysis of the three simplification approaches

Administrative boundary planet graph data. Table A.2 shows the distribution of node degrees, and table A.3 shows the administrative level distribution among the edges of the administrative boundary graph file.

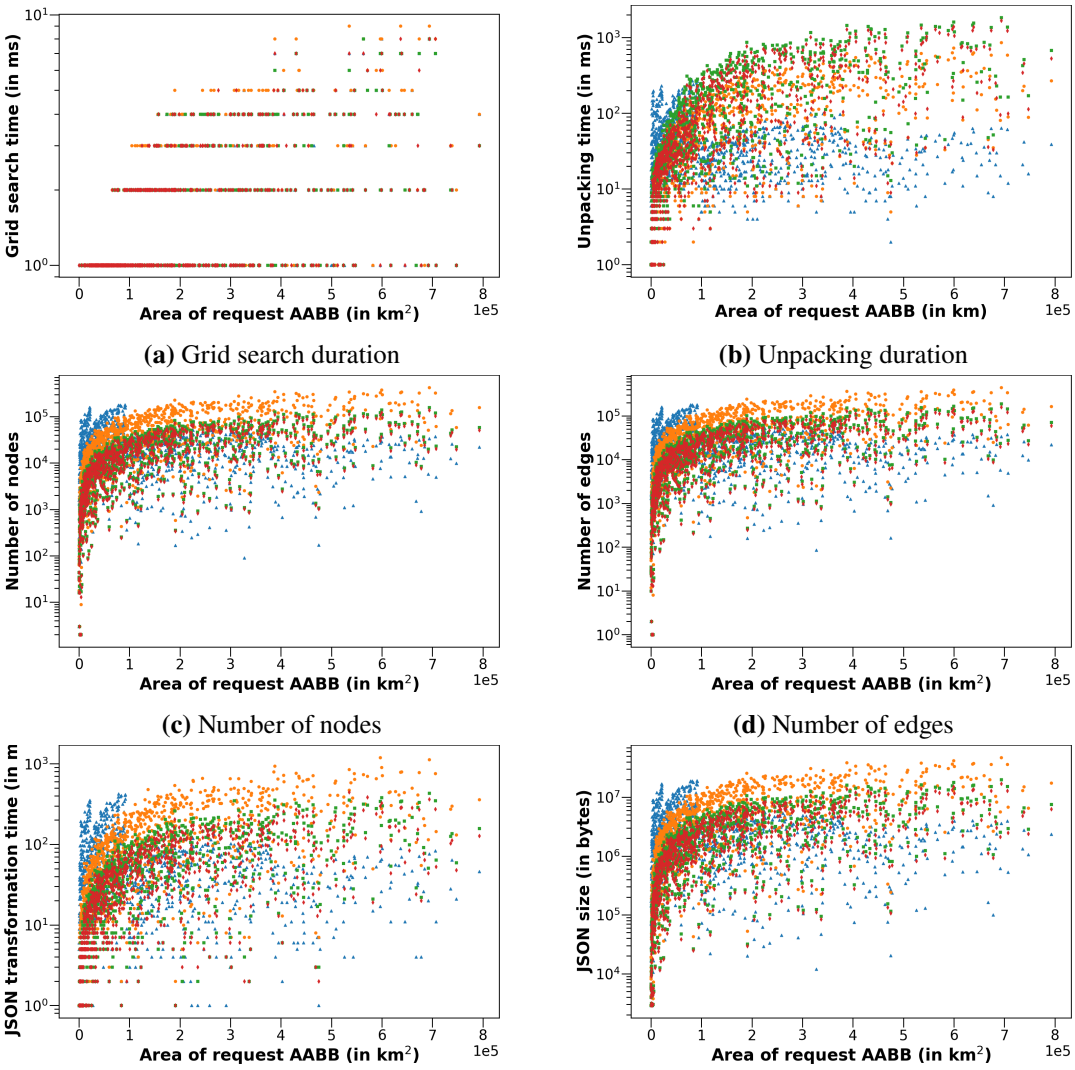
Degree	Number of nodes
0	2184268
1	2654
2	109807087
3	936921
4	262977
5	1299
6	15093
7	93
8	2046
9	14
10	106
11	7
12	25
13	2
14	1
18	1

Table A.2: Degree distribution among the nodes of the administrative boundary planet graph

Administrative level	Number of edges
0	0
1	68
2	3109535
3	819800
4	9666235
5	6682010
6	20687073
7	9612821
8	37089575
9	8040241
10	15871404
11	213498
12	5282

Table A.3: Administrative level distribution among the edges of the administrative boundary planet graph

Germany: Complete dynamic evaluation. Figures A.1 to A.4 show the results of 1,000 requests with $(\text{lat}, \text{lng}) \in [47°, 55°] \times [6°, 15°]$ for each of the four different unpacking approaches. All approaches query for the administrative levels zero to twelve on an administrative planet graph fully simplified by the fast simplification approach. The waterfall and successive approaches use an unpacking factor of twelve, and the evenly approach a factor of four.



(a) Grid search duration

(b) Unpacking duration

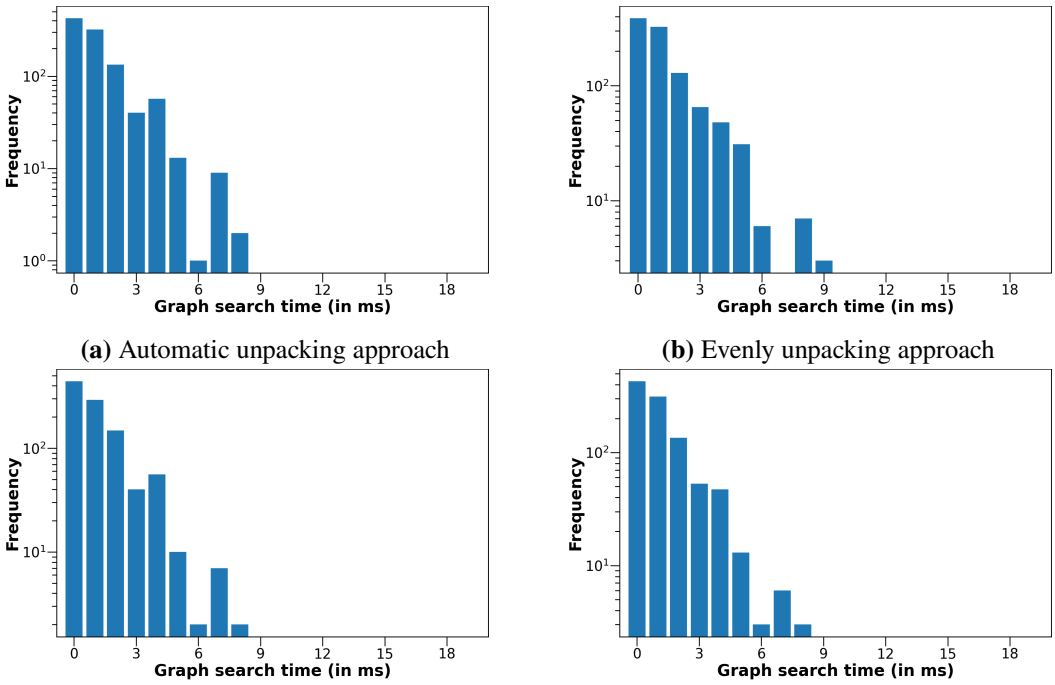
(c) Number of nodes

(d) Number of edges

(e) JSON Transformation duration

(f) JSON size

Figure A.1: Results from 1,000 requests inside $[47°, 55°] \times [6°, 15°]$ in relation to the request's area



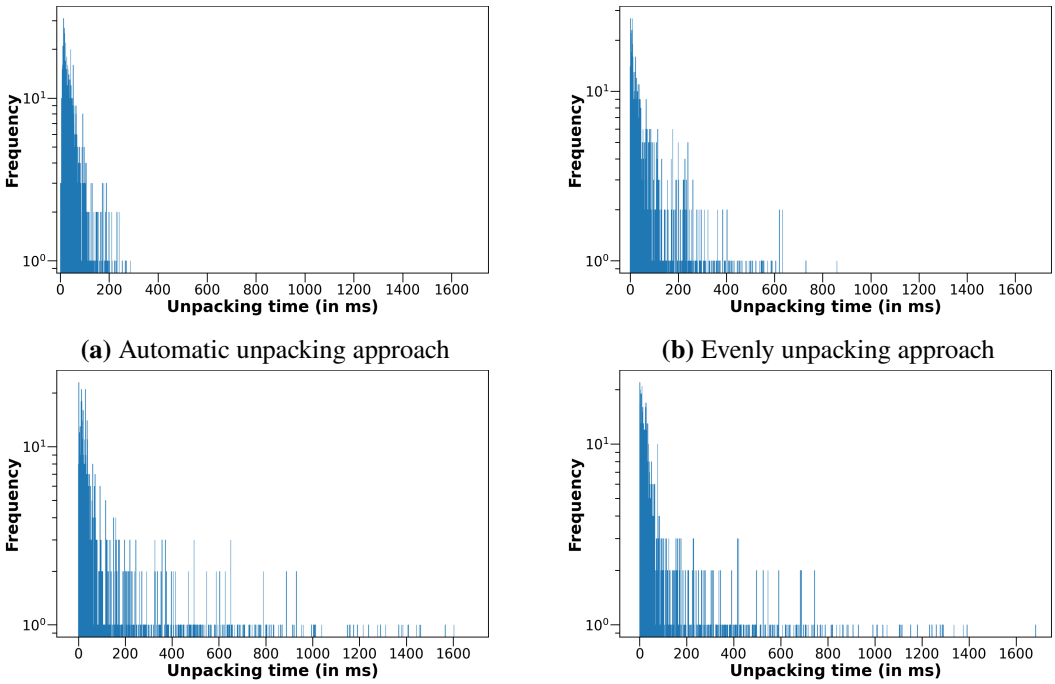
(a) Automatic unpacking approach

(b) Evenly unpacking approach

(c) Waterfall unpacking approach

(d) Successive unpacking approach

Figure A.2: Graph search times from 1,000 requests inside $[47°, 55°] \times [6°, 15°]$



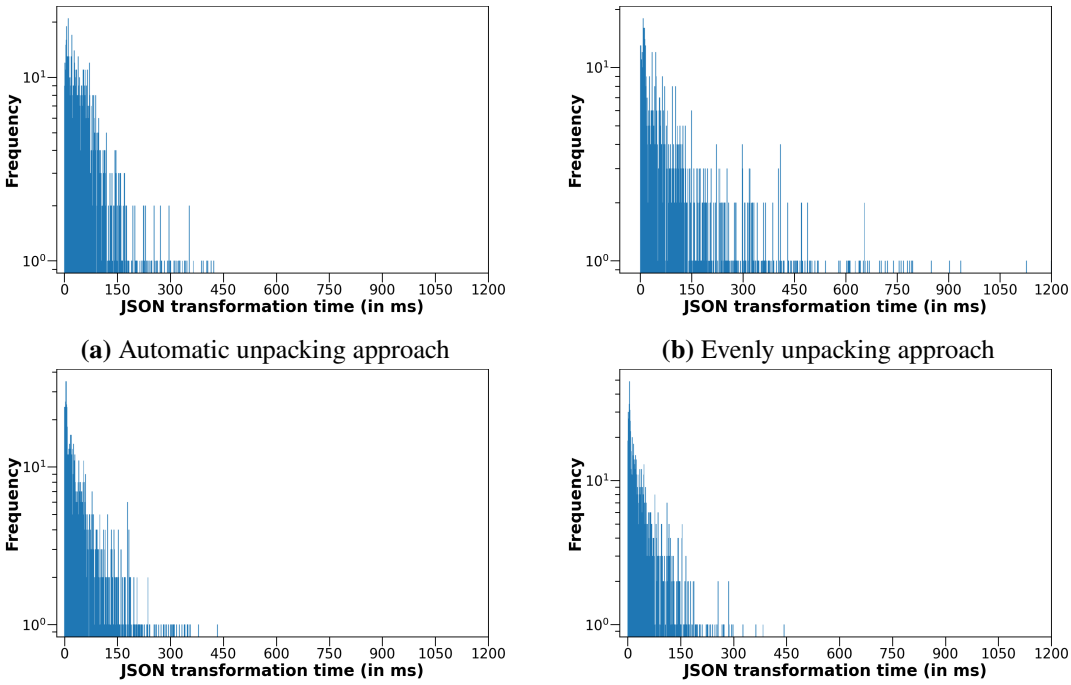
(a) Automatic unpacking approach

(b) Evenly unpacking approach

(c) Waterfall unpacking approach

(d) Successive unpacking approach

Figure A.3: Unpacking times from 1,000 requests inside $[47°, 55°] \times [6°, 15°]$



(a) Automatic unpacking approach

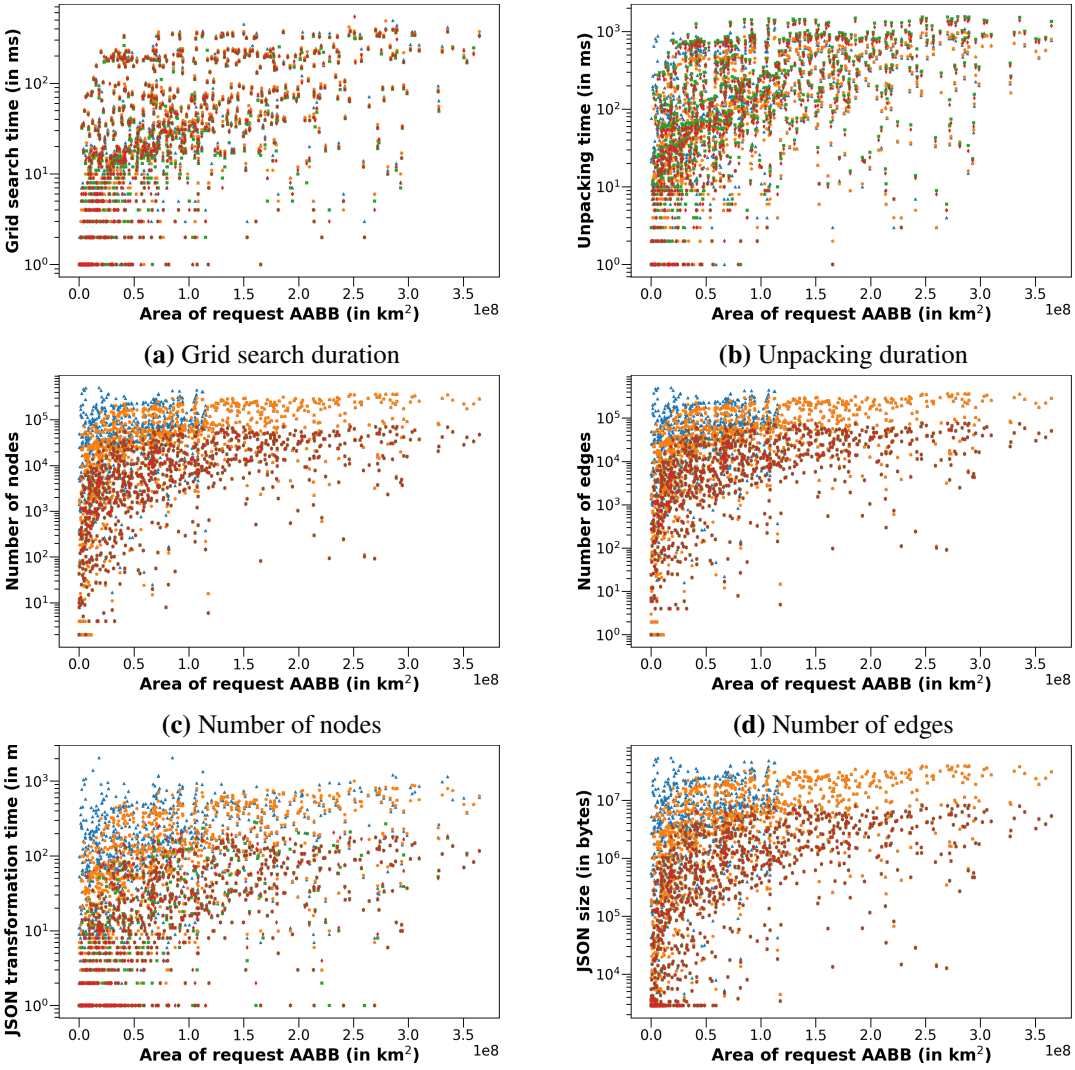
(b) Evenly unpacking approach

(c) Waterfall unpacking approach

(d) Successive unpacking approach

Figure A.4: Transformation times from 1,000 requests inside $[47°, 55°] \times [6°, 15°]$

Planet: Complete dynamic evaluation. Figures A.5 to A.8 shows the results of 1,000 requests with $(\text{lat}, \text{lng}) \in [-90°, 90°] \times [-180°, 180°]$ for each of the four different unpacking approaches. All approaches query for the administrative levels zero to four. The waterfall and successive approaches use an unpacking factor of 65, and the evenly approach a factor of three.



(a) Grid search duration

(b) Unpacking duration

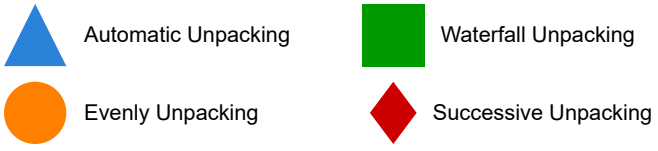
(c) Number of nodes

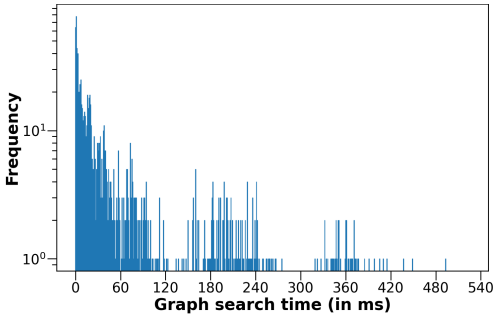
(d) Number of edges

(e) JSON Transformation duration

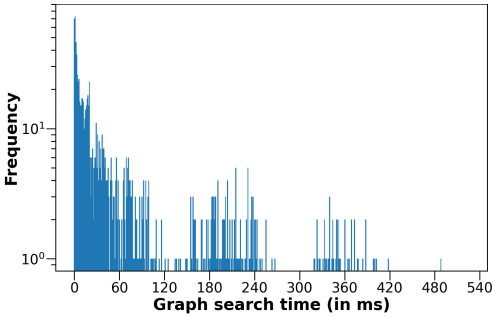
(f) JSON size

Figure A.5: Results from 1,000 requests inside $[-90°, 90°] \times [-180°, 180°]$ in relation to the request's area

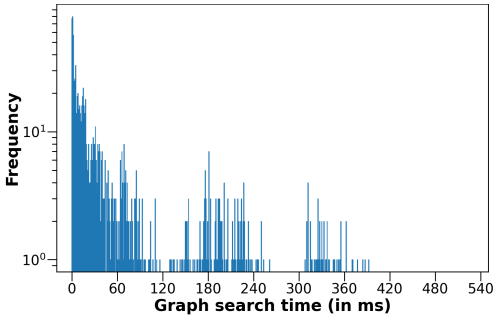




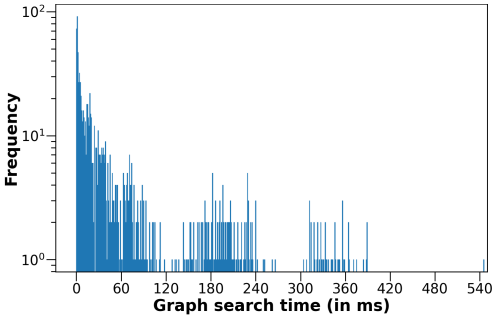
(a) Automatic unpacking approach



(b) Evenly unpacking approach

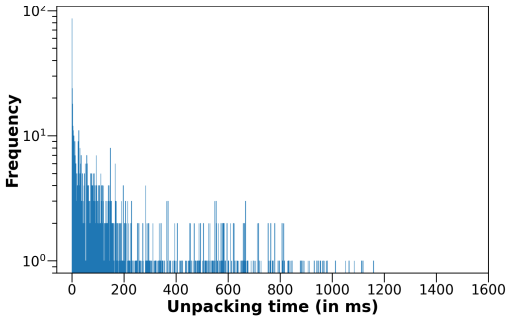


(c) Waterfall unpacking approach

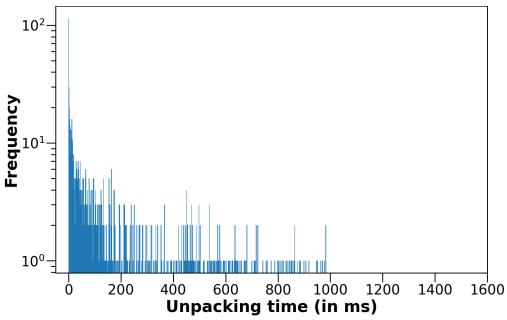


(d) Successive unpacking approach

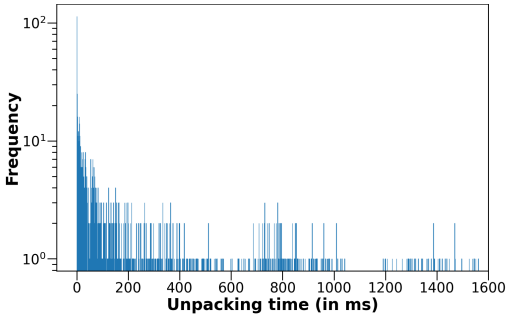
Figure A.6: Graph search times from 1,000 requests inside $[-90°, 90°] \times [-180°, 180°]$



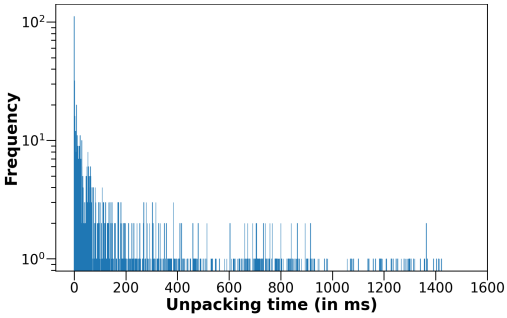
(a) Automatic unpacking approach



(b) Evenly unpacking approach

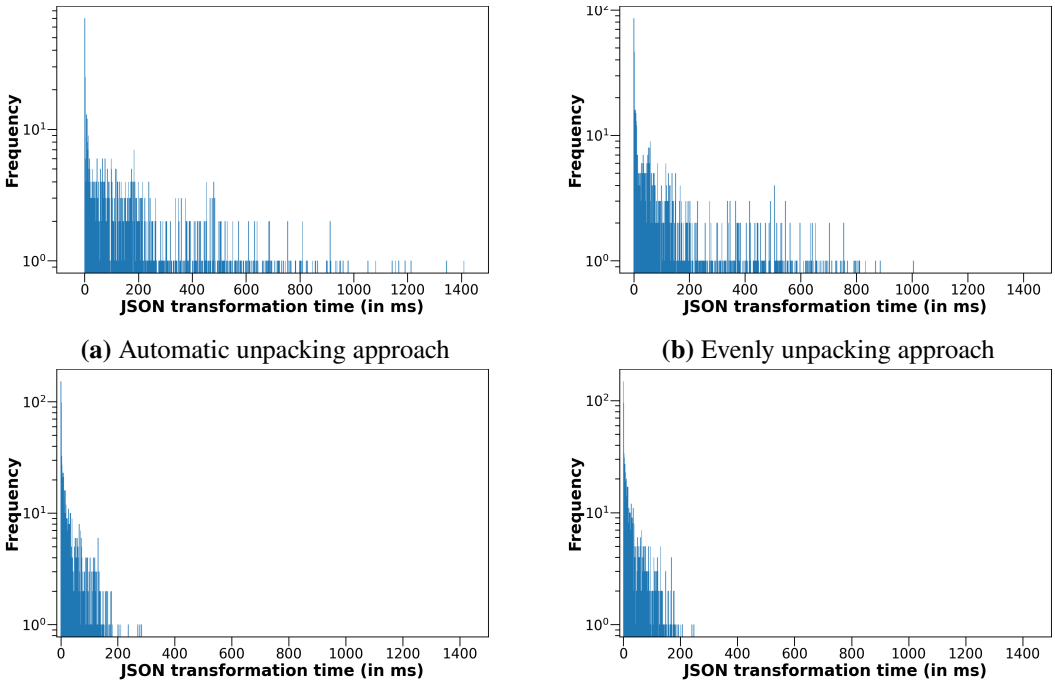


(c) Waterfall unpacking approach



(d) Successive unpacking approach

Figure A.7: Unpacking times from 1,000 requests inside $[-90°, 90°] \times [-180°, 180°]$



(a) Automatic unpacking approach

(b) Evenly unpacking approach

(c) Waterfall unpacking approach

(d) Successive unpacking approach

Figure A.8: Transformation times from 1,000 requests inside $[-90°, 90°] \times [-180°, 180°]$

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