Master's Thesis

Conversion of OSM data to RDF Data and the use of Simplified Geometries when computing Spatial Relations

Iradj Solouk

Summer 2022

University of Freiburg
Chair of Algorithms and Data Structures

Reviewer: Prof. Dr. Hannah Bast
Dr. Fang Wei-Kleiner(Subst. Professor)
Advisor: Patrick Brosi

Abstract

An OSM-data processing tool, which outputs RDF data, will be introduced. To achieve that a custom parser, specialized in the way OSM-data is represented, is used. The requirement of processing large OSM-data sets encourages the use of on-disk processing data structures. The rules of the conversion are presented in this work. For this, the underlying structure of OSM was taken into consideration.

The tool uses clustering and geometric analysis to infer additional data from the input. Furthermore simplified polygons are used in order to infer spatial relations as the containment relation. Those simplifications are computed with a modified Ramer-Douglas-Peucker algorithm. The efficient computation of the spatial relations is done via an RTREE and a directed acyclic graph.

Structure

Chapter 1

Introduction

1.1 Goal

Knowledge bases are used to store complex structured and non-structured information and reasoning can be performed on those. Queries can be performed on those when using SPARQL engines. E.g. in order to find all fountains in Freiburg-Altstadt the query found in Listing 1.1 can be performed. Given former query the SPARQL engine returns 25 elements which locations are depicted in figure 1.1.

Listing 1.1: SPARQL Query used to find all fountains inside "Altstadt" in Freiburg.

```
PREFIX osmnode: <https://www.openstreetmap.org/node/>
PREFIX osmway: <https://www.openstreetmap.org/way/>
PREFIX osmrel: <https://www.openstreetmap.org/relation/>
PREFIX osmt: <https://www.wiki.openstreetmap.org/wiki/key:>
PREFIX osmwiki: <https://wiki.openstreetmap.org/wiki/>
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>
PREFIX geo: <http://www.opengis.net/ont/geosparql#>
PREFIX wpd: <https://de.wikipedia.org/wiki/>
PREFIX wd: <http://www.wikidata.org/entity/>

SELECT ?osm_id ?hasgeometry WHERE {
  osmrel:1960176 geof:sfContains ?osm_id .
  ?osm_id osmt:amenity "fountain" .
}
```

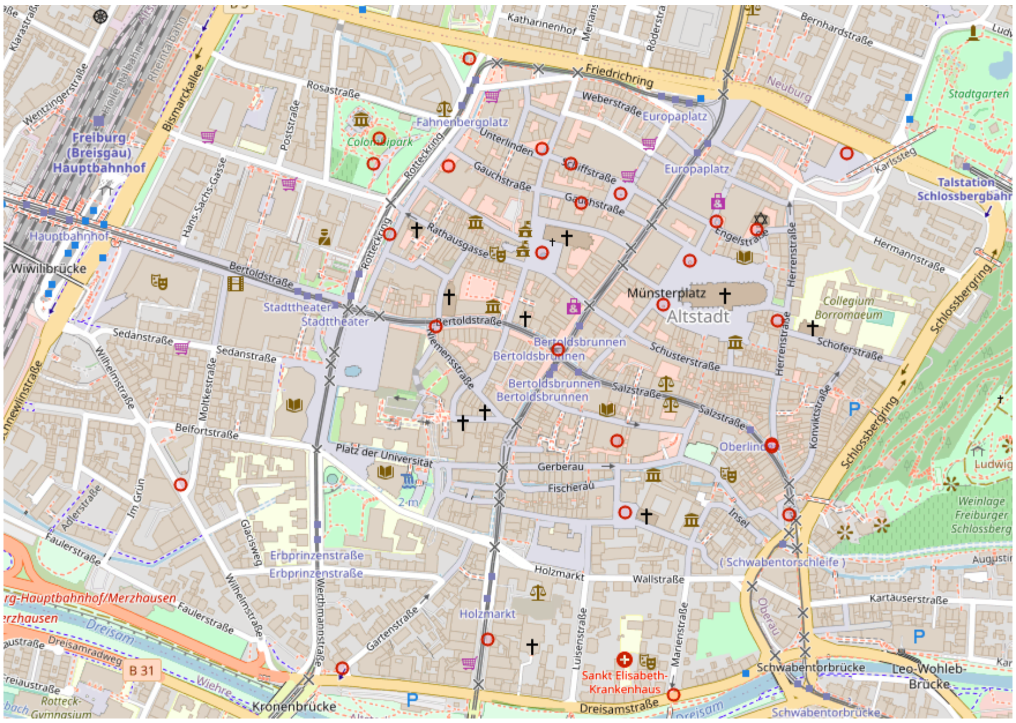
The entries of the knowledge base are stored in triples of subject, predicate and object. This works tool converts triple from geodata. More specifically from Open Street Maps (OSM). OSM is the largest geo-data set which is accesible.

The plain aim is the conversion of OSM data to ttl triples. Other subgoals emerged throughout the process of solving the main aim. The subgoals were the inference of new data points which are computed by combining given data points as well as the computation of the spatial relations. The rules of conversion are formulated with a certain desired conformity of structure between OSM and triples in mind.

The tool of this work may be regarded as a blackbox which takes an OSM data set as the input and a ttl file as the output. The interior of the blackbox is being revealed in the third chapter.

To further motivate and state a possible application of this tools output, one may consider the incorporation and linking of geographic data with a knowledge base that may contain general facts from heterogeneous sources. Complex queries on such a joined knowledge base as well as the visualization of

Figure 1.1: A map showing "Altstadt" in Freiburg. The red circles denote the locations of the returned result elements of the SPARQL query.



those become possible.

1.2 Introduction of terms

In the following, a few terms which are vital for understanding the underlying problems as well as related work, are being introduced. First of all, the terms, abbreviated by OSM, RDF/TTL and WKT, that relate to the fixed specification, defined by the goal of the thesis, are being introduced in the following.

1.2.1 OpenStreetMap (OSM)

The OpenStreetMap (OSM) [Foundation, 2022] project was founded in 2004. Any person can register and contribute to the OSM by adding and editing geodata of the OSM data. The data is available under the open database license. A usual workflow for voluntary mapper is the viewing of aerial photographies and subsequently, the outlining of structures or landmarks that have been recognized and the tagging of those according to OSM standards.

There are three classes of locations in OSM: nodes, ways and relations. Nodes are the most basic OSM elements, which consist of a longitude, latitude and an identifier (node ID). Ways consist of an ordered list of node IDs and a way id. A way is a either a line or a polygon, depending whether the the first and last node of the node list are identical, which results in an open or closed polyline. Relations are composed of an ordered list of nodes, ways and relations. Further context is given to any OSM element via tags. A tag consists of a key and a value. For instance the tag name=Central Park is used for a widely known recreation park found in Manhattan, New York that is mapped into OSM as a way, since it consists of nodes that form a closed polygon. Since relations may contain other relations and in contrast to the former two OSM element types, are enforced to have at least one tag they are the most complex of the three OSM element classes. Any item of the OSM element list of a relation may have a role associated with it. The role *inner* and *outer* is used to define multipolygons. The *inner* parts are subtracted from the *outer* parts.

The OSM data set consists of more than seven billion nodes, 800 million ways and nine million relations. More than eight million users are registered at this moment. Since the data is under open license, the data can be modified and customized for arbitrary applications or use. Hence, for the formerly mentioned reason and the availability, using OSM data for the input seemed fitting.

1.2.2 Resource Description Framework (RDF)

RDF is a model of information in the form of web resources. The RDF serialization format used as the output of this works tool is called Turtle (ttl). A ttl file consists of triples in the form of: subject, predicate and object, which are concluded by an ending "."-character. Between all of the triples terms as well as the ending character, there is a delimiter like a space or a tab. Furthermore prefixes can be introduced, which shorten the number of used characters in a data entry while keeping human readability.

Consider the following statements:
Michael Jackson is the performer of the song Thriller. Queen is the performer of the song Bohemian Rhapsody. Freddy Mercury wrote the latter song.
Those statements may be expressed in the ttl format as given by listing 1.2 and listing 1.3. Both listings model the same information, but the latter one makes use of prefixes.

```
<http://example.org/artist/Michael_Jackson> <http://example.org/relation/
    performerOf>
 <http://example.org/song/Thriller> .
<http://example.org/artist/Queen> <http://example.org/relation/performerOf>
 <http://example.org/song/Bohemian_Rhapsody> .
<http://example.org/song/Bohemian_Rhapsody> <http://example.org/relation/writtenBy>
 <http://example.org/artist/Freddy_Mercury> .
```

Listing 1.2: Content of a simple ttl file without prefixes.

```
@prefix artist: <http://example.org/artist/> .
@prefix relation: <http://example.org/relation/> .
@prefix song: <http://example.org/song/> .

artist:Michael_Jackson relation:performs song:Thriller .
artist:Queen relation:performs song:Bohemian_Rhapsody .
song:Bohemian_Rhapsody relation:writtenBy artist:Freddy_Mercury .
```

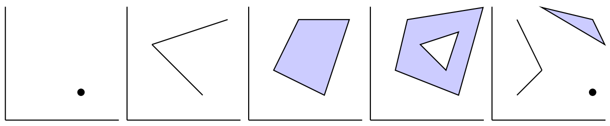
Listing 1.3: Content of a simple ttl file with prefixes.

1.2.3 Well-Known Text (WKT)

In order to store an OSM elements geometry inside a triples object, Well-known text (WKT) strings are used. Neglecting the coordinate reference system in the following. Using WKT one can express a point in space as *POINT (30 10)*. A point is visualized in the left-most graph of Figure 1.2. An open polyline for instance can in WKT be expressed as *LINESTRING (30 10, 10 30, 40 40)*. An open polyline is visualized in the second left-most graph of Figure 1.2 .A closed polyline in WKT can be expressed as *POLYGON ((30 10, 40 40, 20 40, 10 20, 30 10))*. A closed polyline is visualized in the center graph of Figure 1.2. A closed polyline with an inner polyline, which gets substracted, expressed using WKT may be written like *POLYGON ((35 10, 45 45, 15 40, 10 20, 35 10),(20 30, 35 35, 30 20, 20 30))*. Each coordinate sequence that comes after the first one, substracts from the initial coordinate sequence. A closed polyline with an inner polyline is visualized in the second right-most graph of Figure 1.2. In a WKT string the former mentioned WKT objects can be grouped via geometric collections. *GEOMETRYCOLLECTION (POINT (40 10), LINESTRING (10 10, 20 20, 10 40), POLYGON ((40 40, 20 45, 45 30, 40 40)))* is an instance for a geometric collection in WKT. A geometric collection is visualized in the right-most graph of Figure 1.2.

For the sake of visualizing WKT geometries, in Figure 1.2 drawings of the depicted examples for the different geometry types are presented. Other WKT geometry types exist, but these are the ones used by the tool. Decisions about the conversion rules of the OSM geometries were made based on the

Figure 1.2: Examples of the WKT section are drawn here. Instances of a point, an open polyline, a closed polyline, a closed polyline with an inner polyline as well as a geometric collection is denoted from left to right.



geometry types of WKT. The location of an OSM node is translated to a WKT point. The geometry of an OSM way with an open polyline is translated to a WKT linestring. The geometry of an OSM way with a closed polyline, potentially with several inner polygons, is translated to a WKT polygon. Since an OSM relation conceptually a grouping of OSM nodes, ways and relations, the WKT geometric collection is for OSM relations, since the WKT geometric collection does also group the other WKT primitives.

To give more contex of how the WKT string is written into a ttl file, view the following line of listing 1.4. The prefixes for the ttl file were omitted.

Listing 1.4: An OSM element's geometry expressed as a single ttl triple.

```
osmrel:13038953 geo:hasGeometry "GEOMETRYCOLLECTION(POINT(7.986672 48.506398),
    POINT(7.989988 48.502834),  LINESTRING(7.986689 48.506284,7.986729
    48.506251,7.987331 48.505752,7.987984 48.505333,7.988140 48.505199,7.988266
    48.505046,7.988351 48.504922,7.988450 48.504726,7.988661 48.504433,7.988891
    48.504216,7.988844 48.504160,7.989198 48.503798,7.989576 48.503397,7.990016
    48.502844))"^^geo:wktLiteral .
```

1.3 Tools and data structures in application

Relevant tools, data structures and their purpose for this works tool is introduced in this section.

1.3.1 Libosmium

The OSM data set is parsed via the libosmium library. The library is specialized on dealing with large OSM data sets, which do not fit into memory. Hence, the parser is used as an event-based parser, which means, the parser does process each OSM element as it is read on-the-fly rather than preloading all of the OSM elements into memory. Via libosmium, this works tool parses a .pbf tool with threading enabled. Using.pbf files, Google's protocol buffer, shrinks the data set to a fraction of the original size. The OSM data set of the globe is called planet.osm. The uncompressed size is larger than 1500 GB, whereas the size of the pbf-compressed data set of the globe has a size of about 60 GB. A custom parser was developed for this works tool, but the parser was not able to parse .pbf files, which is why ultimately, the libosmium parser was chosen. Reading the input by using libosmium requires the implementation of an osmium-handler interface. Functions for dealing with OSM nodes, OSM ways and OSM relations are required to be defined.

1.3.2 STXXL and boost::geometry

Since the available memory is not large enough to store the geometrical data of planet.osm the question arises of how to resolve the geometries of the OSM data set. To tackle this problem, STXXL was chosen to store the geometrical data. STXXL is a library which provides the functionality of storing into *external memory*. They implement containers and algorithms similar to standard template library (STL), which are being stored on to disk. In order to compute spatial relations as the *intersection* and the *containment*, the geometries are converted to classes of boost::geometry. Boost::geometry implements the computation of those mentioned spatial relations between two geometries.

1.3.3 RTree

A spatial index is used for an efficient lookup of OSM elements and their geometries. The Boost implementation of the RTree is used as the spatial index. The RTree uses a tree data structure, which groups objects within a local vicinity with a minimum bounding rectangle within the hierarchy of the tree. The Boost implementation of the RTree is in use. The documentation of Boosts RTree refer to [Leutenegger et al., 1997] and [García R et al., 1998] for bulk insertion. They claim the packing algorithm, which is used for bulk insertion, increases the performance of the RTree for the generation as well as the querying.

1.3.4 Directed Acyclic Graph

A graph in general is defined by its vertices and edges. Each edge connects two vertices. The edges of a directed graph are not bidirectional. An directed acyclicgraph is a directed graph with the condition of any vertex not being visited after the vertex occured in a path for each path that can be traversed.

Hence, paths with loops are not possible for a directed acyclic graph.

Chapter 2

Related Work

A selection of related articles will be introduced in the following. Relevant similarities and distinctions of those articles will be stated as well. Related work is sorted in chronological order.

2.1 YAGO and YAGO2

This section is about another known knowledge base.

YAGO2[Hoffart et al., 2013] is a knowledge base which is derived from processing a combination of Wikipedia, WordNet [Miller, 1995] and GeoNames data as a gazetteer. Wikipedia is a crowdsourced portal that provides information via Wikipedia articles. Wikipedia articles are converted to entities in YAGO. WordNet is a lexical database, which contains a taxonomy of nouns. GeoNames purpose is to store geographical entities. The data is stored as triples. These triples consist of subject, predicate and objects. Hence, it is analogous to RDF.

WordNet is used for classification of statements given by Wikipedia and GeoNames. GeoNames data can have links to wikipedia entities, which, when mapped, can be used to add geographical information to entities that were found before. YAGO2 extends the predecessor YAGO[Suchanek et al., 2007] by focusing on the spatial and temporal context. E.g. the lifespan of entities, facts or events is given by certain predicates like "wasCreatedOnDate", if it is given. An entity is a spatial entity if the entity has a real world location that is not volatile. Per spatial entity YAGO2 does not store polygons, instead one pair of coordinates is stored. Since locations are extracted from wikipedia there is less of an emphasis on the spatial information in contrast to a data set that focuses on geometric data.

In contrast to YAGO/YAGO2 this works tool does not cross match data between different tools, the tool has a focus on processing geometries and an RDF conversion that is close to OSM's data structure.

2.2 LinkedGeoData

This section is about another project which also does the OSM-data to RDF-data conversion.

This tool by Stadler et al. [Stadler et al., 2012] also converts OSM data to an RDF format. They map OSM nodes and ways to RDF by generating URIs in the form of **lgd:node<id>** as well as **lgd:way<id>**.

The nodes that a way refers to are given with **lgd:way<id>/nodes**. They state, that they try to evoke a conversion, which will have a structure which is similar to the OSM structure. Furthermore instance matching is done, in order to link elements of different RDF knowledge bases. They store the geometry of a way as a literal.

This works tool stores the geometry of tagged nodes, all ways and more than 99% of the relations in the form of WKT string literals. The geometry is given explicitly instead of the OSM element references. Although no instance matching is perfomed by this works tool, tags like wikidata=Q154797 or wikipedia=Deutscher_Bundestag will be translated to objects **wd:Q154797** and **wpd:Deutscher_Bundestag**, with the prefix **wd:** for **<http://www.wikidata.org/entity/>** and**wpd:** for **<https://de.wikipedia.org/wiki/>**.

2.3 TripleGeo

This tool also performs the geodata to RDF conversion. Also a lot of geometrical computations are involved which are interesting in the context if this work.

It is claimed, that the tool TripleGeo by Patroumpas et al. [Patroumpas et al., 2014] does extract geospatial features from various sources, access geometries and attributes from standard geographic formats or common data base management systems(e.g. shapefiles or PostGIS). It is capable to recognize points, linestrings, multi-linestrings, polygons and multi-polygons. Furthermore reprojection between coordinate reference systems is possible, before triples can be exported into notations as RDF/XML.

This works tool is also able to recognize those geometries with the addition of arbitrary groupings of the same geometries as long as the source OSM element, from which certain triples are derived, is not a relation that contains further relations. OSM relations that consist of one or more OSM relations make up less than one percent of all OSM relations.

2.4 OSCAR

This tool is about querying spatial relations using a custom spatial index.

This tool by Bahrdt et al. [Bahrdt et al., 2017] is capable of text search as well as querying spatial relations like "north of" or "near". A cell arrangement is infered by an OSM dataset. For each of these cells, all tags of the OSM elements that are contained by the cell are being linked to the cell. Then the cells and the individual tagged nodes are linked. In order to refine the cells, a triangulated mesh via delaunay triangulation and merging of triangles is computed. They have a live web app [Foundation,], which one can query for text search. In contrast to this, this works tool outputs a precomputed dataset.

2.5 OSM2RDF/OSM2TTL

This section is about a tool which also uses an RTree in combination with a directed acyclic graph in order to infer spatial relations.

The tool OSM2RDF by Bast et al. [Bast et al., 2021], formerly known as osm2ttl, converts most information of a given OpenStreetMap (OSM) dataset into Resource Description Framework (RDF) triples. Such a triple consists of a subject, a predicate and an object. For the RDF conversion, the geometries of the OSM elements are converted to well-known-text (WKT) strings. A possible application for this knowledge base consisting of those triples may be the combining of RDF datasets and querying the knowledge base with a tool like QLEVER [Bast and Buchhold, 2017].

Furthemore OSM2RDF computes certain spatial relationships between certain OSM Elements. For instance whether an OSM Element geographically resides within or intersects with certain (multi)polygons. Also the geometries of OSM elements, which do not depict a single point, are required to be resolved, since only references to other OSM elements, which compose the geometry, are given, instead of their explicit geometry.

Spatial relations like the intersection or the containment are also to be inferred by comparing the geometries of the OSM elements. In order to compute those spatial relations efficiently, a combination of an RTree and a directed acyclic graph (DAG) is used. If for an OSM element the containment relation was computed wrt. a certain one, the DAG provides the containment relations, which were already computed for that certain OSM element. Hence those precomputed spatial relations infer the spatial relations for the source element, additional containment checks can be omitted. OSM2RDF applies transitive reduction to the DAG, which is not done for this works tool.

OSM2RDF as well as this works tool uses libosmium for parsing the OSM dataset, as well as the geometry and the RTree implementations by boost.

The tool of this work also uses the approach of combining the RTree with a DAG. In order to enhance the efficiency, two kinds of simplified geometries are being additionally stored for certain OSM elements. Here the term simplified geometry signifies a geometry with a reduced number of points. Thus spatial relation checks take less runtime. The two kinds of simplified geometries are an over- and an underestimation of the former geometry. More of this is being explained in the next chapter. In comparison to OSM2RDF this works tool uses stxxl as "external" memory in order to store and resolve the geometries of the OSM elements afterwards.

Chapter 3

Approach

An overview of the tool as well as notable individual parts of the tool are presented in this chapter.

3.1 Program flow of the application

In the following the program flow of this works tool is presented. The program sequence of this works tool is divided into notable computation steps. If a part of the program is labeled with ▨ , data is written to the outputfile. If a part of the program is labeled with 🟫 , considerable amounts of data, for data, which is used in other computation steps, are stored into memory. Considerable amounts of data in this context means, that aggregated data like the computation of statistics is excluded. If a part of the program is labeled with 🟦 , data is stored into external memory.

1. Write tags of OSM elements and location of OSM nodes ▨
2. Compute statistics
3. Compute boundaries 🟫
4. Generate RTree 🟫
5. Generate directed acyclic graph 🟫
6. Store geometrical data of OSM elements and compute spatial relations for OSM nodes 🟦 ▨
7. Compute geometries of OSM ways 🟦
8. Generate new ways by clustering fragmented ways 🟦 ▨
9. Write geometries of OSM ways and their spatial relations ▨
10. Compute geometries of OSM relations 🟦
11. Write geometries of OSM relations and their spatial relations ▨

The OSM data set is read by using libosmium. Reading a data set as big as planet.osm, if in .pbf format, is done in less than five minutes, if no or hardly any data is written or stored. This is noticeably faster than the overall runtime of the program, which is why several reading passes on the data are performed.

3.1.1 Write tags of OSM elements and location of OSM nodes

In this part of the program, the first reading pass occurs. For each parsed OSM element, the OSM tags are converted to triples. If the parsed OSM element is an OSM node, the location of the node is converted to a triple as well. The triples are written to the output file after the ttl prefixes are written. The used prefixes are found in listing 3.1.

```
@prefix osmnode: <https://www.openstreetmap.org/node/> .
@prefix osmway: <https://www.openstreetmap.org/way/> .
@prefix osmrel: <https://www.openstreetmap.org/relation/> .
@prefix osmt: <https://www.wiki.openstreetmap.org/wiki/key:> .
@prefix osmwiki: <https://wiki.openstreetmap.org/wiki/> .
@prefix geof: <http://www.opengis.net/def/function/geosparql/> .
@prefix geo: <http://www.opengis.net/ont/geosparql#> .
@prefix wpd: <https://de.wikipedia.org/wiki/> .
@prefix wd: <http://www.wikidata.org/entity/> .
```

Listing 3.1: The used prefixes for the ttl file.

As an example of a typical conversion, the triple osmrel:62718 osmt:name "Bremen" . is generated when converting the OSM tag name=Bremen from the OSM relation, which has the ID 62718. The key is included in the triples predicate in combination with the osmt prefix. The value of the tag translates to the object of the triple, which is a string literal in the former example.

In certain circumstances, the conversion is done differently. Those other cases of conversion and their rules are defined in the following:

- if the value of an OSM tag starts with *http* or *www* while the key of the tag does not contain the string *note*, the object of the triple is enclosed with < and >, which states, that the object is a URL.
- if any data from a tag, which is included in a triple is going to be percent encoded. This means the object of the triple is not a literal.
- if the OSM key contains the string *wikidata* and not *note* and the OSM value has the structure of Q...(multiple instances starting with Q are possible), then for each instance of the value generates a triple.

  Continuing with the former example, the triple osmrel:62718 osmt:wikidata wd:Q1209 . is generated from the OSM tag wikidata=Q1209 .
- if the OSM key contains the string *wikipedia* and not *note*, then a triple is generated.

Continuing with the former example, the triple `osmrel:62718 osmt:wikipedia wpd:de%3AFreie_Hansestadt_Bremen .` is generated from the OSM tag `wikipedia=de: Freie_Hansestadt_Bremen`. The percent-encoding rule is applied in this example as well.

- if the key of the OSM tag, which is going to be converted, is *is_in*, several triples are created for the comma-separated split of the value of the OSM tag.

For this part of the program, if the OSM element that is parsed, is an OSM node, the location is also written to the output file. A possible triple would be `osmnode:20982927 geo:hasGeometry "POINT(8.807165 53.075820)"^^geo:wktLiteral .`, which has been derived from an OSM node. The introduced WKT-string are used in order to express the geometries.

3.1.2 Compute statistics

Another parsing pass is done in this part of the program. This part does compute statistics regarding the OSM data set. The number of OSM nodes, ways and relations, relations, which contain relations, relations, which do not contain relations as well as the number of `multipolygon=yes` tagged are computed in this program part. The maximum ID of the set of OSM ways is also stored, since it will be used as the starting index for the newly generated clustered-ways in subsection 3.1.8.

3.1.3 Compute boundaries

Before the majority of the OSM geometries is computed, a specific small subset, along with their geometry, is processed. For planet.osm, the small subset contains 600 000 OSM relations. Those specific relations are tagged with `boundary=administrative` and `type=boundary`. The geometry of a relation, which is tagged with the formerly mentioned tags, consists of one or multiple polygons. Throughout the rest of the work, the formerly mentioned tagged OSM relations are defined as *boundaries*. Their geometry depicts an area. Since the subset is small enough, the data that is computed and stored this part of the program is done in memory.

Listing 3.2 depicts a basic struct, which is used to store the locations of the OSM nodes. Listing 3.3 depicts a struct, which is used to store the references of an arbitrary OSM way. Listing 3.4 depicts a struct, which is used to store the location of an arbitrary OSM way. A comment refers to the statement below the comment. The comment states the data type as well as the purpose of the struct's members.

Listing 3.2: Struct used to store the location of an OSM node.

```
// name of the struct
nodecoord {
  // long integer, denotes an ID of a an OSM node
  ID
  // type double, denotes longitude of an OSM node
  lon
  // type double, denotes latitude of an OSM node
  lat
}
```

Listing 3.3: Struct used to store the references of an OSM way.

```
// name of the struct
wayref {
  // long integer, denotes the ID of the OSM way
  ID
  // integer, stores the order of the OSM node refID does refer to
  order
  // long integer, denotes the ID of the refered OSM node
  refID
  // integer, encoding of the name of the OSM way
  nameID
  // bool, the value is true, if the element contains a polygon
  isPolygon
  // bool, true, if the OSM way is tagged
  hastag
}
```

Listing 3.4: Struct used to store the location of an OSM way.

```
// name of the struct
waycoord {
  // long integer, denotes the ID of the OSM way
  ID
  // integer, stores the order of the OSM node refID does refer to
  order
  // double, used to store the longitude of the refered OSM element
  lon
  // double, used to store the latitude of the refered OSM element
  lat
  // integer, encoding of the name of the OSM way
  nameID
  // bool, the value is true, if the element contains a polygon
  isPolygon
  // bool, true, if the OSM way is tagged
  hastag
}
```

The first parsing run is used to store the IDs of the boundaries as well as the types and IDs of their references associated to them. The second parsing run is used to store the IDs of OSM ways, which occur in the stored references of the first parsing run. Along with the IDs of the OSM ways, their associated references are stored. Hence, the third parsing run in this part of the program, stores the IDs of the formerly referenced OSM nodes and their coordinates via instances of the previously defined nodecoord struct. The ID references of the OSM relations and ways are resolved by replacing the references with the actual coordinates. Thus, the geometries can be constructed.

In the following, the vector of type nodecoord, which stores the IDs and locations of the OSM nodes, is defined as *nodecoords*. The vector of type wayref, which stores the IDs and references of the OSM ways is defined as *wayrefs*. The entries of nodecoords are sorted by ID and the entries of wayrefs are sorted by refID. Now, using the merging algorithm(shown in 1), the vector containing the locations of the OSM ways, defined as *wayrefs* is filled.

Algorithm 1 Merge nodecoords and wayrefs to waycoords

```
// initialize iterator
currentnode ← nodecoords.begin()
// initialize iterator
currentwayref ← wayrefs.begin()
while neither currentnode nor currentwayref reached the end do
    if currentnode.ID = currentwayref.refID then
        // copy all common members of currentwayref, except the refID
        // instead add lat and lon members from currentnode
        tempway ← initWaycoord(currentnode, currentwayref)
        // append to the waycoord container
        waycoord.add(tempway)
        // increment the iterator
        next(currentwayref)
    else
        // increment the iterator
        next(currentnode)
    end if
end while
```

Consider the example of a rudimentary OSM data set containing the OSM nodes with ID 1, 2 and 3 as well as an OSM way with the ID 1, which spans a polygon defined by the OSM nodes. The referenced IDs occur in ascending order. The OSM way with an ID of 1 references OSM nodes with IDs 1 and 2, in that specific order. The entries of the vectors nodecoords and wayrefs after the parsing passes of this program part can be depicted in listing 3.5 and 3.6.

Listing 3.5: Entries of the nodecoords vector for the trivial example.

```
ID:1    lon:1.0   lat:1.0
```

```
ID:2    lon:2.0   lat:1.0
ID:3    lon:1.5   lat:1.5
```

Other parts of the program make use of the members nameID, isPolygon and hastag, while this one does not.

Listing 3.6: Entries of the wayrefs vector for the trivial example.

```
ID:1    order:1   refID:1   nameID:1    isPolygon:false   hastag:true
ID:1    order:4   refID:1   nameID:1    isPolygon:false   hastag:true
ID:1    order:2   refID:2   nameID:1    isPolygon:false   hastag:true
ID:1    order:3   refID:3   nameID:1    isPolygon:false   hastag:true
```

The entries of this rudimentary example are already sorted by ID and if possible by refID. Before applying the merging algorithm, nodecoords is sorted by ID and wayrefs is sorted by refID. Then the entries of vector waycoords are found in listing 3.7.

Listing 3.7: Entries of the waycoords vector for the trivial example.

```
ID:1    order:1   lon:1.0   lat:1.0   nameID:1    isPolygon:false   hastag:true
ID:1    order:4   lon:1.0   lat:1.0   nameID:1    isPolygon:false   hastag:true
ID:1    order:2   lon:2.0   lat:1.0   nameID:1    isPolygon:false   hastag:true
ID:1    order:3   lon:1.5   lat:1.5   nameID:1    isPolygon:false   hastag:true
```

If the geometries of the OSM ways, contained in waycoords, are to be assembled, waycoords needs to be sorted primarily by ID and secondarily by order. The structs, which are used to store OSM relations are found in listings 3.8 and 3.9 .Resolving the structs for the OSM relations adds another layer via a *suborder* member which is considered similiar to the order member one level further. Conceptually, the assembly of geometries of OSM relations is done in the same way via sorting and merging. Since the merging algorithm for relcoords is simliar to the previousley mentioned merging algorithm for waycoords, it is not stated.

Listing 3.8: Struct used to store references of an OSM relation.

```
// name of the struct
relref {
  // long integer, denotes the ID of the OSM way
  ID
  // integer, stores the order of the OSM node refID does refer to
  order
  // long integer, denotes the ID of the refered OSM node
  refID
  // type encodes whether the reference is an OSM node, way or relation
  type
  // encodes the role of the reference: inner, outer and so on
  role
  // integer, encoding of the name of the OSM way
  nameID
  // bool, the value is true, if the element contains a polygon
  hastag
}
```

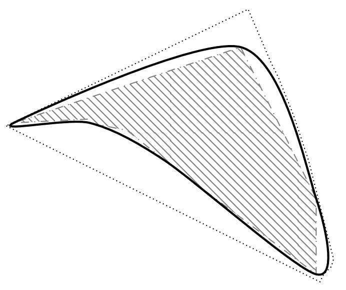
Listing 3.9: Struct used to store locations of OSM relations.

```
// name of the struct
relcoord {
  // long integer, denotes the ID of the OSM way
  ID
  // integer, stores the order of the OSM node refID does refer to
  order
  // layer of order when dereferencing OSM elements
  suborder
  // double, used to store longitude
  lon
  // double, used to store latitude
  lat
  // type encodes whether the reference is an OSM node, way or relation
  type
  // encodes the role of the reference: inner, outer and so on
  role
  // integer, encoding of the name of the OSM way
  nameID
  // bool, the value is true, if the element contains a polygon
  hastag
}
```

When the geometries of the boundaries are fully resolved, they are stored in an own container and used in the next part of the program. Based on the geometry of a boundary, if possible, two additional simplified geometries are computed and stored together with the original geometry. One of the simplifications is an overestimation, i.e. the original geometry does not overshoot the overestimation and is contained in it. The other simplification is an underestimation, i.e. the underestimation does not overshoot the original geometry. Hence, if any geometry is compared to a boundary geometry, then in some cases, the spatial relation can be implied, by comparison with the simplified polygons. If a geometry is not inside the overestimation, it cannot be inside the original geometry and is rejected. If a geometry is inside the underestimation, then it has to be within the original geometry. Since the spatial comparison of polygon benefits from a reduction of the number of points, the effect of this approach is being evaluated in chapter 4. A possible over- and underestimation in relation to the source polygon is skteched in 3.1.

The simplification algorithm is a variation of the Ramer-Douglas-Peucker algorithm [Douglas and Peucker, 1973] [Ramer, 1972]. The used simplification algorithm has an input parameter, which defines whether the output polygon is an underestimation or an overestimation. The applied simplification may output a polygon, which does self-intersect. Hence, a post processing step is performed, which is an attempt to repair self-intersections, which might occur. The reparation attempt, in case of the overestimation, is achieved by computing the unified area of the original geometry and the

Figure 3.1: Sketching of over- and underestimation. The dotted line denotes the overestimation. The underestimation is denoted as the polygon filled with the dashes pattern. The actual polygon is denoted by a black continuous line.



simplified one. The reparation attempt, in case of the underestimation, is achieved by computing the intersected area of the original geometry and the simplified one.

Overestimation-Polygon

A parameter called *epsilon* is used in the simplification algorithm. The procedure for computing the overestimation is denoted in 2. For the sake of clarity and simplicity, some input parameters and repetitive validation and correction commands of lesser significance, were not displayed in the algorithm.

In case the given geometry has a low number of points(fewer than 30 points), the bounding box is returned as the overestimation-polygon. The value of 0.013 for *epsilon* was found experimentally. This value is the only one used for as *epsilon* when computing the overestimation-polygon. If the simplification returns a self intersected or in another sense broken geometry, a reparation attempt is performed, by trying to unify the source geometry as well as the simplified one. The simplified geometry does not to be valid for this, it is attempted non the less. If the number of points of the unified geometry is too low compared to the simplified geometry, the union is rejected and the simplified geometry is further processed. In order to process it, points occuring in segments between the self intersections are removed, then another simplification algorithm with the same epsilon value is applied. In the end, before using the computed geometry, it is checked for validity, if it is not valid, the trivial case of a bounding box is used instead.

Algorithm 2 Procedure of computing overestimation.

```
// given to the procedure
inputGeometry
// passed as reference, this is the geometry that is being returned
outputGeometry
// numberOfPoints() computes the number of points occuring in the geometry
if numberOfPoints(inputGeometry) = 0 then
    // premature exit of the procedure
    return
end if
if numberOfPoints(inputGeometry) < 30 then
    // envelope() returns the bounding box of the passed geometry
    outputGeometry ← envelope(inputGeometry)
    // premature exit of the procedure
    return
end if
// initialize the epsilon parameter used for the polygon-simplification
epsilon ← 0.013
// the simplification algorithm is based on the douglas-peucker algorithm
// the first parameter is the source geometry which needs to be simplified
// the second parameter is the geometry that is generated and passed as reference
// the third parameter is the epsilon parameter which controls the amount of simplification
// the fourth parameter controls whether to compute the over- or underestimation
//  it is set to true for the overestimation and false for the underestimation
simplifyPolygon(inputGeometry, outputGeometry, epsilon, true)
// coveredBy() checks if first geometry is inside second one
// valid() checks if geometry can be safeley used for further processing
if not coveredBy(inputGeometry, outputGeometry) or not valid(outputGeometry) then
    // union() merges the given geometries and returns the new one
    unionPoly ← union(inputGeometry, outputGeometry)
    if numberOfPoints(unionPoly) < 0.5 * numberOfPoints(outputGeometry) then
        // resolveintersection() removes points between intersecting segments
        resolveintersection(outputGeometry)
        simplifyPolygon(outputGeometry, outputGeometry, epsilon, true)
    else
        outputGeometry ← unionPoly
    end if
end if
if not valid(outputGeometry) then
    outputGeometry ← envelope(inputGeometry)
end if
```

Underestimation-Polygon

The polygon-simplification method for the underestimation uses local optimization within a specified range wrt. the *epsilon* parameter. The procedure is denoted in 3. Several steps of lesser significance are omitted for the sake of simplicity.

Starting with a value of 0.00000001 for epsilon for the simplification. In 6 equidistant steps until the approximate value of 0.013 for epsilon the simplified geometries are scored and compared. If a concurrent score is decreasing, the optimization is stopped prematurely. The local maximum is returned if the geometry is valid. If it is not valid, a reparation attempt is performed by intersecting the non-valid simplification with the source geometry, in order to remove occuring self-intersections. In general using an operation as the union or the intersection of geometries is not guaranteed to work properly for non-valid geometries. Nontheless, via trial and error, this approach has the desired outcome of repairing the geometry in many cases. The score is computed with the following formula:

$$pointTerm(A,B) = 1 - (\frac{numberOfPoints(B)}{numberOfPoints(A)})$$
$$areaTerm(A,B) = (\frac{area(B)}{area(A)})$$
$$score(A,B) = 0.4 * pointTerm(A,B) + 0.6 * areaTerm(A,B)$$

For the application, A is the source geometry and B is the simplfiied geometry. The function *numberOfPoints(A)* computes the number of points for geometry A. The function *area(A)* computes the area for geometry A. A high score is achieved for a combination of a high point reduction as well as preserving a lot of the source area.

3.1.4 Generate RTree

After the assembly of the geometries of boundaries, these geometries are stored in an in-memory container. This container is passed to an RTree boost implementation, which is why the assembled geometries are stored using boost::geometry. The RTree is a spatial index which computes spatial relations for bounding boxes of geometrical elements. In this context the RTree is used to fetch boundaries. We defined Boundaries as OSM relations, which have certain tags like `type=boundary`, that might have a certain spatial relationship to a geometry used in a query. After all boundaries were computed, the container containing all boundaries is passed to the RTree for initialization.The RTree is kept in-memory.

Algorithm 3 Procedure of computing overestimation.

```
// given to the procedure
inputGeometry
// passed as reference, this is the geometry that is being returned
outputGeometry
// numberOfPoints() computes the number of points occuring in the geometry
if numberOfPoints(inputGeometry) $<$ 30 then
    // premature exit of the procedure
    return
end if
// initialize the epsilon parameter used for the polygon-simplification
currentepsilon $\leftarrow$ 0.00000001
// upper boundary for epsilon
maxepsilon $\leftarrow$ 0.013
deltaepsilon $\leftarrow$ (maxepsilon - currentepsilon) / 6
// keep track of last score in order to compute the difference
lastscore $\leftarrow$ -1000000
// keep track of best score so far
bestscore $\leftarrow$ -1000000
// score for the currentepsilon
currentscore $\leftarrow$ -1000000
// remember best geometry so far
bestGeometry
while currentepsilon $<$ maxepsilon do
    // initialize Geometry
    currentGeometry
    // view simplifyPolygon() comments at the overestimation algorithm
    simplifyPolygon(inputGeometry, currentGeometry, epsilon, false)
    // compute the score based on
    // ...the areas of the source and the target geometry
    // ...the points of the source and the target geometry
    currentscore $\leftarrow$ score(inputGeometry, currentGeometry)
    if numberOfPoints(currentGeometry) $<$ 7 then
        // too less points, exit
        abort()
    end if
    if currentscore $>$ bestscore then
        bestscore $\leftarrow$ currentscore
        bestGeometry $\leftarrow$ currentGeometry
    end if
    if currentscore $>=$ lastscore then
        lastscore $\leftarrow$ currentscore
    else
        breakWhile
    end if
    epsilon $\leftarrow$ epsilon + deltaepsilon
end while
// if bestpoly is not valid, try to repair bestpoly once more
...
if valid(bestpoly) then
    outputGeometry $\leftarrow$ bestpoly
end if
```

3.1.5 Generate directed acyclic graph

For the generation of the directed acyclic graph, a similar algorithm as the one called *Create DAG* from [Bast et al., 2021] is used. This tools approach has the modification of using simplified polygons to imply spatial relations. The DAG is used for transitive comparisons. For example if the containment relation is computed for the structure called *Big Ben* in London then the RTree will return a group of possible candidates. But the spatial relations between *London* and *England* was already computed. Hence the DAG can be queried for this and we can apply transitive inference to find that *Big Ben* is also contained in *England*. Thus explicit spatial comparisons between *London*, *England* or any previously computed boundary are omitted. The algorithm is depicted in algorithm 4. The evaluation of using simplified polygons, in order to reduce runtime is found in 4. The DAG is stored in-memory as well, since only mappings of IDs to other IDs, which are bounded by boundaries, are stored.

3.1.6 Store geometrical data of OSM elements and compute spatial relations for OSM nodes

The used structures were introduced earlier in listings 3.2 , 3.3, 3.4, 3.8 and 3.9. Throughout the first pass of this part of the program locations of all OSM nodes via the nodecoord struct, geometrical references of OSM ways via the wayref struct and geometrical references of OSM relations via the relref struct, are stored. The number of entries of wayrefs that are computed, is so large, that the used memory would not be able to fit them all. This is why STXXL is used. We can use containers and algorithms similar to STL, but the elements are stored on disk(external memory). Sorting is possible, which is handy, since the algorithms like the merging rely heavily on sorting. The structs, which are used for the STXXL containers, need to be plain old data(POD). One cannot use dynamic vectors as the struct members.

When the locations of the nodes are stored, they are also converted to boost::geometry objects and their spatial relations against boundaries are computed.

3.1.7 Compute geometries of OSM ways

Then, nodecoords and wayrefs are merged to waycoords as presented in subchapter 3.1.3 of the current chapter. Prior to that, the nodecoords are sorted by ID and wayrefs are sorted by refID(in ascending order). For this part of the program the container of the nodecoords, wayrefs and the generated

Algorithm 4 Generate DAG

```
// sorts the vector of boundaries by area in ascending order
sortByArea(boundaries)
for boundary from boundaries do
    // initialize skiplist, in order to remember which IDs were already processed
    skiplist()
    // returns the result vector from querying the RTree for elements, that are within
    candidates ← RTreeWithinQuery(boundary)
    // this function sorts the vector of candidates by area in ascending order
    sortByArea(candidates)
    for c from candidates do
        if c is not in skiplist then
            // RTree returns also simplified geometries
            if boundary is within underestimation(c) then
                // add the edge between the input params to DAG
                addEdge(boundary, c)
                // add c to skiplist, as it has been processed
                skiplist.append(c)
                // traverse DAG to find successors and return as a vector
                succs ← successor(c)
                // add successors to skiplist, as they have been processed
                skiplist.insert(succs)
            else
                // RTree returns also simplified geometries
                if boundary is within overestimation(c) then
                    if boundary is within c then
                        // add the edge between the input params to DAG
                        addEdge(boundary, c)
                        // add c to skiplist, as it has been processed
                        skiplist.append(c)
                        // traverse DAG to find successors and return as a vector
                        succs ← successor(c)
                        // add successors to skiplist, as they have been processed
                        skiplist.insert(succs)
                    end if
                end if
            end if
        end if
    end for
end for
```

Algorithm 5 Compute spatial relations for any geometry

```
// the given geometry of an OSM element
geometryElement
// initialize skiplist, in order to remember which IDs were already processed
skiplist()
// returns the result vector from querying the RTree for elements, that intersect with input
candidates ← RTreeIntersectsWithQuery(geometryElement)
// this function sorts the vector of candidates by area in ascending order
sortByArea(candidates)
for c from candidates do
    if c is not in skiplist then
        // RTree returns also simplified geometries
        if geometryElement is within underestimation(c) then
            // traverse DAG to find successors(including c) and return as a vector
            succs ← successor(c)
            for succ from succs do
                // write succ intersects c and vice versa and...
                // ...write succ within c and ...
                // ...write c contains the succ
                writeFourRelations(succ, c)
            end for
            // add successors to skiplist, as they have been processed
            skiplist.insert(succs)
        else
            // RTree returns also simplified geometries
            if geometryElement is within overestimation(c) then
                if geometryElement is within c then
                    // traverse DAG to find successors and return as a vector
                    succs ← successor(c)
                    for succ from succs do
                        // write succ intersects c and vice versa and...
                        // ...write succ within c and ...
                        // ...write c contains the succ
                        writeFourRelations(succ, c)
                    end for
                    // add successors to skiplist, as they have been processed
                    skiplist.insert(succs)
                end if
            end if
        end if
    end if
end for
for c from candidates do
    if c is not in skiplist then
        if geometryElement intersects c then
            // writes: c intersects geometryElement and vice versa
            writeIntersectsRelations(geometryElement, c)
        end if
    end if
end for
```

waycoords are STXXL vectors, since they too large to fit into memory.

3.1.8 Generate new ways by clustering fragmented ways

In general, a street or a path in the real world is not found as a whole inside an OSM data set. Instead the street or path is represented as several OSM ways, which fragments the street or path as a whole. An example for this, would be the *Kaiser-Joseph-Straße* in the old town of Freiburg. More than ten OSM ways are used to describe this street. Hence, with this part of the program, the computation of streets and paths as a whole is attempted. We define the newly generated OSM elements as *wayclusters*.This part of the implementation is roughly divided into the following steps:

- Sort wayrefs by nameID and refID
- Find the connected components
- Intersect tags for connected components
- Resolve connected components
- Write connected components

First of all, the wayrefs vector is sorted by the refID and nameID. Then, the wayrefs vector is traversed and if for different consecutive entries with the same refID different IDs are found, the mapping of those links is stored in memory. We define a closed group of those mappings as a *connected component*. Those mappings are stored in-memory. Using those mappings, the OSM data set is parsed again, in order to write the tags. If for a connected component all OSM ways were parsed, only the common tags of those OSM ways are written to the output file. Furthermore, triples for each fragment which is contained by a waycluster. The geometry is assembled by traversing the waycoords vector, while checking whether the stored mappings contain the ID of the current entry.

3.1.9 Write geometries of OSM ways and their spatial relations

In order to assemble the entries of waycoords to actual geometries, the container holding all waycoords is sorted primarily by ID, secondarily by order. The sorted container of waycoords is traversed. Blocks of the same ID are read and concatenated to the geometry. If the first and last entry of the block have the same longitude and latitude, the geometry is a closed polyline, else an open polyline. Triples containing the geometry as WKT strings, in this case *POLYGON* or *LINESTRING* respectively, are written to the output file.

Furthermore, for each assembled geometry, the algorithm 5 is applied to, in order to write the spatial relations to the output file. The algorithm attempts to infer the spatial relationship by computing the containment relation for the given geometry wrt. the underestimation of a proposed query result of the RTree. If the spatial relationship could not be infered, then proceed by computing the containment relation for the given geometry wrt. the overestimation. If the containment relation could not be disqualified, only then use the actual polygon, in order to infer the spatial relation.

3.1.10 Compute geometries of OSM relations

Analogously, relcoords are similarly computed as waycoords are computed. The type member of the relref struct is used to distinguish whether the reference is an OSM node, way or relation. References to other OSM relations are neglected, since the number of OSM relations, which refer to other ones makes up less than 1% of the OSM data set. Not all relrefs/relcoords are processed in one processing sequence. The processing sequence is applied on two sets of relrefs/relcoords of about similiar size. The tag type=multipolygon is used to discriminate into the two sets. The set of OSM relations with the given tag and the set of OSM relations without the given tag. This has a positive effect on the sorting parts.

3.1.11 Write geometries of OSM relations and their spatial relations

Prior to traversing the relcoord vectors, they are sorted by ID, order and suborder. Blocks of entries with the same ID are assembled to geometries. Those are converted to WKT strings and written to the output file. Additionaly, the fully assembled geometry undergoes the algorithm 5, in order to write spatial relations to the output file.

Chapter 4

Experimental Evaluation

This chapter introduces and presents the results of this work and discusses further attempted approaches.

4.1 Evaluation Setup

4.1.1 Static grid

The application of a static grid, in order to perform local lookup of geometries, was examined in the scope of this work. By static grid, the division of the local plane into cells is meant, for which the cells have the same constant dimensions. Although the implementation is easy to realize, the approach is inefficient for the following reasons:

- Not using hierarchy when storing local vicinities of OSM elements.
- The approach is not sparse, in the sense that the encoding of the cells describes each cell of the grid. A vast amount of cells will have no OSM elements assigned to it.

Hence, this approach was rejected and an RTree, which solves those two problems, is used instead.

4.1.2 Simplified polygons

This subchapter covers the evaluation of the application of simplified polygons in order to estimate polygon comparisons. The following data sets were used: Freiburg, BaWü(Baden-Württemberg), Germany, Europe and also the Planet dataset. They are listed in ascending order. Furthermore, each of the listed datasets is contained in their next larger succeeding dataset.

Table 4.1: Statistics on usage of simplfied polygons.

	Freiburg	BaWü	Germany	Europe	Planet
Accepts via underestimation	1035909	2976069	19637209	-	-
Rejections via overestimation	1964445	9108820	59858222	-	-
Regular accepts/rejections	251566	1219006	13687931	-	-

Relative coverage by simplifications	92.26%	90.84%	85.31%	-	-

The statistic of the employed simplified polygons used to infer spatial relations is found in table Table 4.1. The remaining cases, when the simplification did not suffice to infer the spatial relation, are counted as well. The header lists the datasets at hand. The kinds of statistics used are found in the first column. The occurences were computed and assigned to the table wrt. to the kind of statistic and the used dataset. If a large proportion of the spatial relations, given as in the last row of the earlier mentioned table, suffices for an application, one might consider working only with simplified polygons.

The relative coverage, given by the simplifications, seems to decline, if a dataset from a bigger geographical extract is used. The elicited effect here is a growing proportion of boundaries with a low amount of points, since the boundaries with less than 30 points are not simplified.

An example of the simplified polygons together with their source polygon is sketched in Figure 3.1. The dotted line in the sketch denotes the overestimation. If any geometry is not contained in the overestimation, the geometry cannot be found inside the source polygon. The underestimation is denoted as the polygon filled with the dashed pattern. If any geometry is not contained in the underestimation, the geometry has to be inside the source polygon. As for the datasets at hand, about 10% of the elements, which are to be compared, is within the area given by the geometrical difference of the overestimation and the underestimation.

Statistics were only recorded for counting whether an OSM element is accepted/rejected due to a simplification or not. Other ways of measuring the effect of using simplified polygons to estimate spatial relations may be of interest as well. For instance, comparing the runtime as well the net area of OSM that is lost when only relying on simplified polygons, were not examined in the scope of this work.

Table 4.2: Statistics on usage of simplfied polygons.

	Freiburg	BaWü	Germany	Europe	Planet
# boundaries	868	3011	27085	327347	-
# computed overestimations	682	2673	26186	326640	-
# repairs for overestimation	35	124	1104	11841	-
# computed underestimations	675	2622	25497	296726	-
# repairs for underestimation	59	318	2791	24932	-
averaged point reduction for overestimations	82.4%	81.6%	76%	74.1%	-
averaged point reduction for underestimation	76%	74.79%	69.45%	71.8%	-

In this work, *boundaries* were defined as relations, which represent territories. Certain combination of tags are assigned to those. The spatial relations of a certain OSM element are computed wrt. those boundaries. Considering column 2 of Table 4.2, denoting the Baden-Württemberg data set in Germany, the statistics have the following meaning. Out of 3011 computed boundaries, for 2673 of those, overestimations were computed and 2622 underestimations respectively. The amount of cases, for which the reparation step was needed is also given in the table.

At last, the arithmetic mean of all relative point reduction computations, is given for both simplifications. For the dataset we are considering, this means, that on average 81.6% less points are found in the overestimations. This means for the mentioned dataset 100% - 81.6% = 18.4% of the total amount of points of the boundaries is used for the overestimations. Considering the relative coverage of simplifications in Table 4.1 the total number of points can be reduced to a fraction.

4.2 Runtime of the program parts

In this section, the net runtimes as well as the runtimes of the individual parts are being revealed.

Table 4.3: Runtime statistics of the program and the parts wrt. the datasets. The datasets marked with * were processed by skipping the computation of spatial relations except for the DAG.

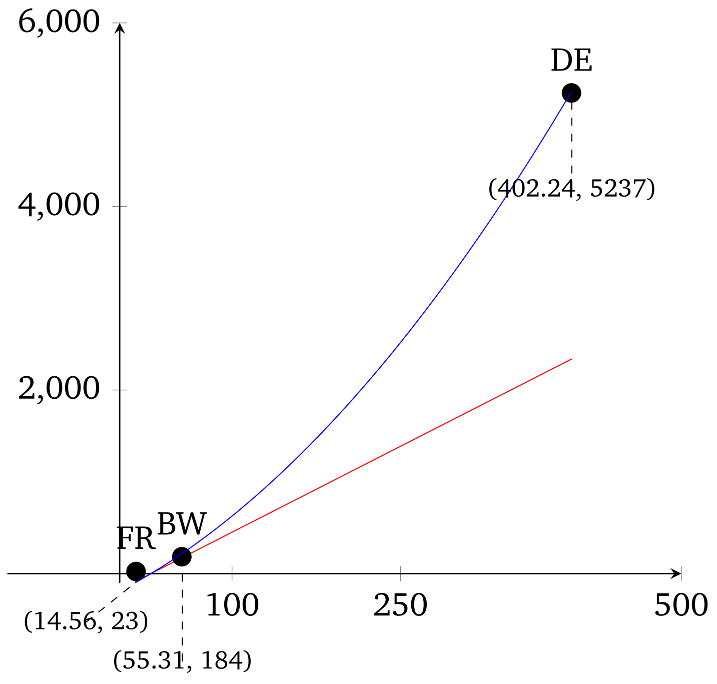
	Freiburg	BaWü	Germany	*Europe	Planet
Printing tags	3min	12min	88min	678min	-
Compute boundaries	5s	34s	9min	86min	-
Create RTree and DAG	6s	1min43s	58min	373min	-

Store and process OSM nodes	4min	24min	828min	22min	-
Compute nameclusters	13s	1min23s	14min30s	264min	-
Process and print OSM ways	7min41s	64min	2458min	325min	-
Process and print OSM relations	10min	79min	1741min	276min	-
Net runtime	23min	184min	5237min	2021min	-

The given net runtime in Table 4.3 is the true time measured outside the application. The measured times for the other program parts are measured inside the application. The real duration of the parts measured inside the application is at least as long as the measured time, which is why they can be used as estimates. The program parts which denote the processing of the OSM elements include the computation of spatial relations. The datasets marked with "*" were run without computing the spatial relations. This is done since an unfeasible run time is expected.

As seen in Figure 4.1, the runtime is more than linear wrt. the number of input elements. Comparing the runtime of the program parts(Table 4.3), which process the OSM elements for the europe dataset, which skips the spatial relations, to the runs, which do not skip the spatial relations, querying the RTree as well as comparing the geometries and writing the spatial relations is the major bottleneck. In general

Figure 4.1: Plot runtimes for Freiburg (FR), Baden-Württemberg (BW) and Germany (DE). The horizontal axis denotes the runtime in minutes. The vertical axis denotes the number of OSM elements in the input data set in millions. The blue curve shows, the data is not linear. The red line shows the linear function based on the first two data points.



Chapter 5

Conclusion

5.1 Summary

An OSM dataset is being read in. On-disk memory is used in order to resolve the geometries wrt. the OSM structure. Boundaries are specific OSM relations, which have certain tags and denote an administrative boundary. Spatial relations between boundaries were precomputed. Simplified polygons which overestimate and underestimate the area of a boundary were computed for the boundaries. Those are used to avoid spatial comparison with the source polygon which on average contains many times over the number points than a simplified one. Nameclusters are newly generated elements, which are derived by combining the geometry of ways with the same name, that also share same nodes. Nameclusters are also computed. All the data is converted to RDF (ttl) format.

5.2 Findings

The computation of the spatial relations is one of the major bottlenecks. The use of a spatial index, custom serialization and algorithms solve the given problem in a feasible runtime. Using simplified polygons for boundaries is a way to drastically reduce the number points which need to be regarded for computing spatial relations. If for the data set of Germany about 85% of the boundaries (those OSM elements against which the spatial containment relationship is checked) are sufficient to be considered, then only the simplified polygons need to be considered instead of the former ones. Additionaly, new OSM elements can be inferred by geometrically unification of OSM ways with same "name" tags, if they share the same coordinates in their geometry. The newly generated OSM elements denote entire streets in their whole geometric extent instead of a fragment.

5.3 Future Works

This section is about approaches and ideas, which came up while working on the thesis, which were not examined in the scope of the work but might be of acadamic interest and might warrant further investigation.

5.3.1 Batch queries

The number of RTree queries can be reduced. Using efficient data structures, in order to process groups of OSM elements at the same time. Eventually, the sought spatial relations are computed for each OSM element of the group of OSM elements.

A possible approach for this might be the following: Mapping coordinates to a single integer, which is used as a key for groups of OSM elements. The mapping could have a resolution of $0.01°$ degree, which translates to about 1.11km. When OSM elements and their geometry is being processed, they can be added to the value of the map by calculating the corresponding key via the mapping function. Periodically, after a set amount of additions, for each key and value pair of the map, they can be checked whether the size of the group of OSM elements has changed since the last time the set amount of additions occured. If the difference of the group size is zero, then a batch query for the group of OSM elements can be applied. An increased amount of co-occurence of locally close OSM elements is used as an assumption with this approach in mind. The mapping and their resolution described in the latter uses a fixed grid approach for the heuristic.

5.3.2 Parallel processing

A reduction in runtime may be achieved by parallel spatial comparisons, the generation of string as well as writing output to dedicated files per thread and merge the outputs at the end.

5.3.3 Evaluation of epsilon parameter in simplification algorithm

Further examination of the epsilon parameter for the simplification algorithm can be done in order to find better fits for the simplified polygon or in order to reduce the range of possible epsilon values, which are relevant. Also, research on whether the modified Douglas-Peucker algorithm has a monotonous relation between the epsilon value and the area of the resulting polygon, might be of interest. In [Friedrich et al., 2019] the parameter of the ramer-douglas-peucker algorithm is analyzed as well as a modified version of the algorithm with a changing epsilon value is introduced.

5.3.4 More spatial relations

This works tool only computes the active and passive containment as well as the intersection as spatial relations. If for any OSM element the maximum and minimum coordinates values are provided as an RDF triple, an RDF query engine could index those values, in order to compare the coordinates. Hence, relative direction queries between OSM elements, as well as distance queries might be possible.

5.3.5 Comparison of implementations

Comparing the used approach to the implementation which does not use simplified polygons as well as an implementation which only uses simplified polygons raises theoretic questions which were not yet explored.

Bibliography

[Bahrdt et al., 2017] Bahrdt, D., Funke, S., Gelhausen, R., and Storandt, S. (2017). Searching osm planet with context-aware spatial relations. In *Proceedings of the 25th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, pages 1–4.

[Bast et al., 2021] Bast, H., Brosi, P., Kalmbach, J., and Lehmann, A. (2021). An efficient rdf converter and sparql endpoint for the complete openstreetmap data. In *Proceedings of the 29th International Conference on Advances in Geographic Information Systems*, pages 536–539.

[Bast and Buchhold, 2017] Bast, H. and Buchhold, B. (2017). Qlever: A query engine for efficient sparql+ text search. In *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management*, pages 647–656.

[Douglas and Peucker, 1973] Douglas, D. H. and Peucker, T. K. (1973). Algorithms for the reduction of the number of points required to represent a digitized line or its caricature. *Cartographica: the international journal for geographic information and geovisualization*, 10(2):112–122.

[Foundation,] Foundation, O. Oscar web app. [Accessed on april 2022] `https://www.oscar-web.de/`.

[Foundation, 2022] Foundation, O. (2022). Main page of osm foundation wiki. [Accessed on april 2022] `https://wiki.osmfoundation.org/wiki/Main_Page`.

[Friedrich et al., 2019] Friedrich, T., Heinzl, L., Fischbeck, P., and Schirneck, M. (2019). Analysis of the parameter configuration of the ramer-douglas-peucker algorithm for time series compression.

[García R et al., 1998] García R, Y. J., López, M. A., and Leutenegger, S. T. (1998). A greedy algorithm for bulk loading r-trees. In *Proceedings of the 6th ACM international symposium on Advances in geographic information systems*, pages 163–164.

[Hoffart et al., 2013] Hoffart, J., Suchanek, F. M., Berberich, K., and Weikum, G. (2013). Yago2: A spatially and temporally enhanced knowledge base from wikipedia. *Artificial Intelligence*, 194:28–61.

[Leutenegger et al., 1997] Leutenegger, S. T., Lopez, M. A., and Edgington, J. (1997). Str: A simple and efficient algorithm for r-tree packing. In *Proceedings 13th international conference on data engineering*, pages 497–506. IEEE.

[Miller, 1995] Miller, G. A. (1995). Wordnet: A lexical database for english. *Commun. ACM*, 38(11):39–41.

[Patroumpas et al., 2014] Patroumpas, K., Alexakis, M., Giannopoulos, G., and Athanasiou, S. (2014). Triplegeo: an etl tool for transforming geospatial data into rdf triples. In *Edbt/Icdt Workshops*, pages 275–278. Citeseer.

[Ramer, 1972] Ramer, U. (1972). An iterative procedure for the polygonal approximation of plane curves. *Computer graphics and image processing*, 1(3):244–256.

[Stadler et al., 2012] Stadler, C., Lehmann, J., Höffner, K., and Auer, S. (2012). Linkedgeodata: A core for a web of spatial open data. *Semantic Web*, 3(4):333–354.

[Suchanek et al., 2007] Suchanek, F. M., Kasneci, G., and Weikum, G. (2007). Yago: A core of semantic knowledge. In *Proceedings of the 16th International Conference on World Wide Web*, WWW '07, page 697–706, New York, NY, USA. Association for Computing Machinery.

Declaration

I hereby declare that I am the sole author and composer of this thesis and that no other sources or learning aids, other than those listed, have been used. Furthermore, I declare that I have acknowledged the work of others by providing detailed references of said work. I hereby also declare that this thesis has not been prepared for another examination or assignment, either wholly or excerpts thereof.

Place, Date: ________________ Signature: ________________