Bachelor Thesis

Metro-Map Styled River Maps

Jianlan Shao

26.11.2020

Albert-Ludwigs-Universität Freiburg im Breisgau
Technische Fakultät
Institut für Informatik

Bearbeitungszeitraum
15. 07. 2020 – 26. 11. 2020

Gutachter
Prof. Dr. Hannah Bast

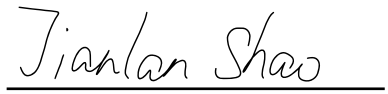
Betreuer
Patrick Brosi

Declaration

I hereby declare, that I am the sole author and composer of my 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 my thesis has not been prepared for another examination or assignment, either wholly or excerpts thereof.

München, 26. Nov 2020

Place, Date



Signature

Contents

Abstract 1

Zusammenfassung 2

Acknowledgements 3

1 Introduction 4
- 1.1 Motivation . . . 4
- 1.2 Our River Map vs. Metro Map: Similarities and Differences . . . 4
  - 1.2.1 Similarities . . . 4
  - 1.2.2 Differences . . . 5
- 1.3 An Example . . . 5

2 Related Work 8

3 Data Extraction and Storage Structure 9
- 3.1 Data Extraction . . . 9
  - 3.1.1 OSM Data Structure . . . 9
  - 3.1.2 Extracting Procedure . . . 10
- 3.2 Storage Structure . . . 10
  - 3.2.1 Node Map . . . 10
  - 3.2.2 Edge Map . . . 11
  - 3.2.3 River Map . . . 11
  - 3.2.4 Storage Class . . . 12

4 Data Processing 13
- 4.1 Baseline and Basic Procedure . . . 13
  - 4.1.1 Baseline . . . 13
  - 4.1.2 Basic Procedure . . . 14
- 4.2 Organizing Nodes . . . 15
- 4.3 Sorting Edges . . . 16
  - 4.3.1 Splitting Edges . . . 16
  - 4.3.2 Concatenating Edges . . . 17
- 4.4 Sorting Rivers . . . 18
- 4.5 Adding River Names . . . 19
- 4.6 Length Filter . . . 21
- 4.7 Adding River Colors . . . 21

4.8 Exporting Data to `GeoJSON` 23

5 Evaluation **26**

6 Possible Problems and Future Work **27**
6.1 Interruption Caused by Lakes 27
6.2 Inconsistent OSM Data 28
6.2.1 River Loops 28
6.2.2 Breaking Points 29
6.2.3 Repeated Edges 29
6.2.4 Wrong Direction of Rivers 31

Bibliography **32**

Abstract

In this thesis, we extract the waterway information from OpenStreetMap (OSM) data and organize it into a directed graph so that we can observe the tributaries of each river clearly. We rearrange the river map in such a way that each river looks like a single subway line starting at its source until its mouth, that is, each river segment is labeled with all its tributaries so far. For example, as a river drains off into the Rhine, the Neckar will be labeled on the Rhine as a parallel line. Eventually, the main stem of the Rhine will look like dozens of small subway lines next to each other. We use our tool LOOM (Line-Ordering Optimized Maps), which renders line graphs in a metro-map style, to present the result in the above-described way.

Zusammenfassung

Diese Arbeit behandelt die Extraktion von Wasserwegen aus OpenStreetMap (OSM) und deren Überführung in einen gerichteten Graph in dem jede Kante sämtliche Zuflüsse enthält. Wir visualisieren diesen Graphen so, dass jeder Zufluss einer U-Bahn-Linie entspricht, die von der Quelle bis zur Mündung verläuft. Auf diese Weise sind für jeden Flussabschnitt alle Zuflüsse ablesbar. Mündet ein Fluss wie der Neckar beispielsweise in den Rhein, erscheint der Neckar als parallele Linie auf allen folgenden Abschnitten des Rheins, bis der Rhein schließlich eine Ansammlung von dutzenden von parallelen ursprünglichen Zuflüssen ist. Wir nutzen LOOM (Line-Ordering Optimized Maps) um diesen sogenannten Line-Graphen in den Stil einer U-Bahn-Karte zu überführen.

Acknowledgements

I want to thank Prof Hannah Bast, not only for the support she gives me on this project, but also for the courses she taught me, including Programming in C++, Algorithms and Data Structures, and Information Retrieval. She is the lecturer that I learned from the most.

I want to thank Patrick Brosi for all the helpful feedback he gives me.

Special thanks to my husband for being encouraging and supportive. I also want to thank my family and my friends for encouraging me.

1 Introduction

1.1 Motivation

As a collaborative project to create a free editable map of the world started in 2006, OSM has been developing rapidly and has been favorably compared with proprietary data sources[1]. Prominent users of OSM include Facebook, Apple, Microsoft, Amazon Logistics, Uber, and so on. At the same time, thanks to the open data feature of OSM, a variety of tools orienting real-world needs are developed by different people and companies. LOOM (Line-Ordering Optimized Maps), is one of these tools which can automatically generate geographically accurate transit maps [BBS18]. The input to LOOM is data about the lines of a transit network: for each line, its station sequence and geographical course [BBS18]. Then an elegant transit map will be drawn according to their geographical course, and this transit map can be used as an overlay of geographical maps. In this thesis, we try to explore the possibility of applying LOOM to river maps. The motivation is that river maps have similar input to metro maps and they can also make use of the result that LOOM presents.

1.2 Our River Map vs. Metro Map: Similarities and Differences

1.2.1 Similarities

A metro map, or schematic transit map, is a topological map in the form of a schematic diagram used to illustrate the routes and stations within a public transport system[2]. We say our river map is metro-map styled for the following reasons:

1. We ignore the geographical width of rivers and regard them all as single lines.
2. When a tributary river contributes to its stem river, we see it as an extension of the tributary river, rather than the vanishing of the river. In this way, rivers can go from the source all the way to the sea and gather at the mouth, resembling metro lines going from the outskirts of the city all the way to the city center, gathering there.

1.2.2 Differences

Metro maps are often not based on the exact network geography. This is because passengers are usually not concerned with geographical accuracy and are more interested in how to get from one station to another as well as where to change vehicles[3]. Yet our river maps can be used as overlays of actual maps, which means they are geographically accurate. Consequently, there are two characteristics of some metro maps which our river maps do not have:

1. For metro maps, all the stations, or nodes in the context of a directed graph, are usually more or less equally spaced rather than a geographic map.
2. In metro maps, in order to be neat and esthetically pleasing, sometimes there is an "underlying grid", primarily rectilinear or octilinear.

1.3 An Example

Here is an example illustrating this project. We have a screenshot of a map, which is an area near Karlsruhe, Germany. We can observe the Rhine and other small rivers on the map, but unfortunately, it is not recognizable which river consists of which tributaries.

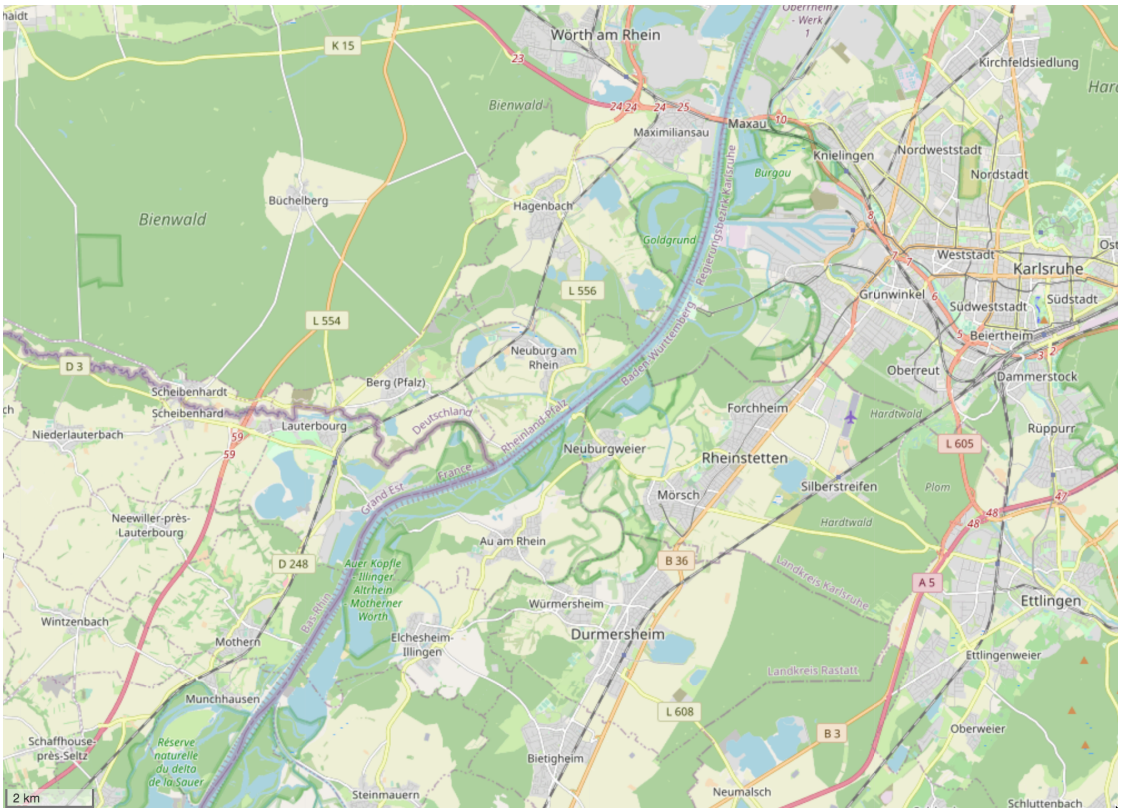


Figure 1.1: Rivers near Karlsruhe (original)

Now we have the result of our project expressed by LOOM of the same area. As we can see, only the information of the river system is shown in the picture, which is more straightforward to the observers. Besides, the rivers and their tributaries are represented clearly, and each tributary has a unique color. The Rhine and its tributaries flow from the south to the north. In the middle of the image, there are a couple of conspicuous canals that resemble the fingers of a hand, they are the Rhine ports of Karlsruhe that were built to transfer oil.

Figure 1.2: Rivers near Karlsruhe (rendered)

When rivers gather at its stem, the colors of them could be hard to distinguish, therefore the Rhine ports of Karlsruhe looks brownish in this picture. But when we zoom in, we can still observe the colors of their tributaries, as in the following image.

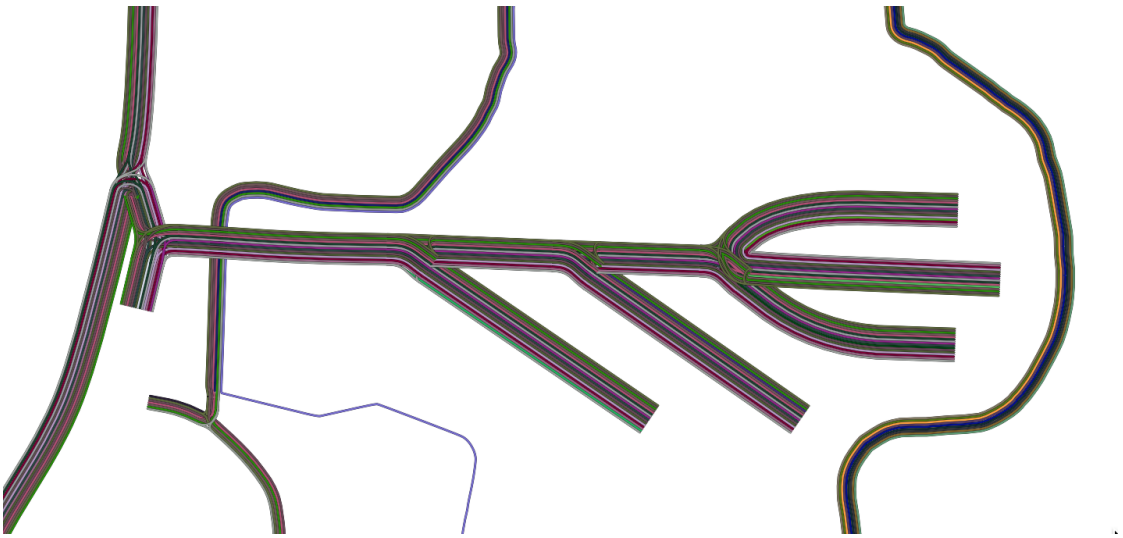


Figure 1.3: Rivers near Karlsruhe (zoomed in)

[1]https://en.wikipedia.org/wiki/OpenStreetMap
[2]https://en.wikipedia.org/wiki/Transit_map
[3]https://en.wikipedia.org/wiki/Harry_Beck

2 Related Work

Since metro map layouts are of high practical relevance, they have been studied extensively in the past. Our work is related to previous work on many topics, including map construction, crossing minimization and schematic metro maps.

1. *Map construction.* A map construction algorithm produces the graph of an underlying network from vehicle tracking data. In [AKPW15], there is an overview of map construction algorithms. Other than the existing work on street networks, in our approach, the input data already represents a multigraph and on that basis, we want to reorganize it into a directed graph.

2. *Crossing minimization.* In [BNUW07], the problem of minimizing intra-edge crossing in transit maps was introduced. The term MLCM, representing metro-line crossing minimization problem, was coined in [BKPS08]. In [ABKS08], [ABKS10] and [Nöl10], several variants of MLCM were defined, and for some of these variants, efficient algorithms were presented. An integer linear program (ILP) formulation for MLCM under the periphery condition was introduced in [AGM08], that is, lines ending in a station must be drawn at the left- or rightmost position in incident edges. While most research in the MLCM comes without experimental evaluations and without the production of actual maps, the article [BBS18] presents an automatic map generator which can yield visually pleasing maps efficiently, meanwhile, these maps can be used as overlays in actual maps.

3. *Schematic metro maps.* There is also research focusing on drawing schematic metro maps, for instance, by restricting the transit lines to octilinear polylines [HMN06], [BBS20] or Bézier curves [FHN$^+$13]. These approaches, like we mentioned in the introduction part, often strongly abstract from the geographical course of the lines. Thus the resulting maps usually can not be used as overlays in typical map services.

3 Data Extraction and Storage Structure

Before we explain how the program processes step by step, we need to illustrate how the structure of the data, that is, the class *RiverGraph* looks like and why it should be designed in that fashion.

3.1 Data Extraction

As an open-source data project, OSM provides geographic data of the world which can be used in the production of paper maps and electronic maps, geocoding of address and place names, and route planning[1]. Apparently, the information on waterways is only a small fraction of OSM data. So the first step of our work is to find out the data we need and to store it properly.

3.1.1 OSM Data Structure

OSM data has three kinds of elements, that is, `nodes`, `ways` and `relations`. All of them can have one or more associated `tags` describing the meaning of the particular element. In an `osm` file, these three types of elements are presented in the order of their `id`s, that is, all the `nodes` from the least `id` to the greatest `id` in the first place, and then all the `ways` and all the `relations` in the same manner.

- A `tag` consists of two items, a `key` and a `value`. `Tags` describe specific features of map elements (`nodes`, `ways` or `relations`)[2]. For example, when a `way` has a `tag` with the `key` "waterway" and the `value` in one of the followings: "river", "stream", "canal", "drain", "ditch" or "brook", then this `way` is the talweg[3], i.e. the deepest points of a riverbed, which is related in our project.
- A `node` is a single point in space defined by its latitude, longitude and `node id`[4]. Specific to our project, a river is composed of numerous nodes in the downstream direction, that is, in the direction that the river flows from source to sea.
- A `way` is an ordered list of between 2 and 2,000 `nodes` that define a polygonal chain[5]. The contained `nodes` are referenced by their `id`s. Normally, a `way` has at least one `tag` or is included within a `relation`[6]. In our project, the

talweg of waterways are noted in the pattern we said in the `tag` part above. For large rivers that are wide enough to require mapping of distinct areas of water banks, more `ways` with other kinds of `tags` are used, but in this thesis, since we regard the rivers as metro lines, that information is insignificant.

- A `relation` is a multi-purpose data structure that documents a relationship between two or more data elements (`nodes`, `ways`, and/or other `relations`), for example, a highway route, a turn restriction or a multi-polygon[7]. `Relations` are also unessential to our project.

3.1.2 Extracting Procedure

The extraction of the needed data is done by the class *OSMFilter*. First, we go through the `ways` of the given `osm` file, by doing so, we can filter out the related `ways` which represent the talwegs of waterways and save the needed information, that is, the `node id`s which build up the respective `way`, the name of the `way`, and the type of the `way` (whether it is a stream or a canal, and so on). Now we have the `id`s of the `nodes` that make up the needed `way`, but we do not have the latitude and longitude information of them. So the second step is to go through the `osm` file for another time, find out and store the geographical information of the related `nodes`.

3.2 Storage Structure

Basically, we have three kinds of `unordered_map` type structures, respectively for the storing of nodes, edges, and rivers. For the last one, some kind of structures are temporary and will be rearranged into new structures, the details will be illustrated in the following sections.

3.2.1 Node Map

The key of the node map (`_nMap`) is the `id` of the corresponding node, the type of the key is `uint64_t`. The value is the instance of class *RiverNode*. In this class, other than the longitude and the latitude of each node, there is a third attribute presenting the edges in this node (`_eInN`), which has the type `unordered_set` and tracks the `id`s of the edges containing this node. Clearly, there is at least one edge in this set. In OSM data, if one can go from one part of the river to another part of the river via the water, there has to be a piece of `waterway=*` between them. This implies that a way tagged as `waterway=*` should not stop on the riverbank of another river, but should proceed to the central way of the other river[8]. Follow this topology, if there are more than one edges in the set `_eInN`, then the given node is an intersection point of multiple rivers or river segments.

3.2.2 Edge Map

For the edge map (`_eMap`), similar to the node map, the key is also the `id` of the corresponding edge, and the type of the key is `uint64_t`. The value is the instance of class *RiverEdge*. In this class, there are the following attributes:

- The name and the type of the river edge.
- The nodes representing the edge.
- `River id`: the edges with the same name and intersect with each other share the same `River id`, and they build up the same river group. We will discuss this further in section *Sorting Rivers* of chapter *4*.
- River in edge (`_rInE`), which is the set of upstream rivers of the current river edge.

3.2.3 River Map

The river map keeps all the rivers. By having a map like this, we can centralize all the river-related information, and use it in the data processing phase, for example in length filtering and in adding river colors.

The key of the river map (`_rMap`) when we extract the river information is the name of the river, it has the type `string`. But that brings problems in some cases. The rivers with the same name do not necessarily intersect with each other, that is, they can have the same name only by accident. This happens especially in small streams. For example, in Baden-Württemberg, a state in southwest Germany, in which the Black Forest is located, there are 228 river edges with the name "Schwarzenbach" (in English: black stream). These streams of the same name spread widely in the Black Forest, and many of them have independent sources, as shown in picture 3.1. In this case, we need to group the ones concatenated with each other (either directly or indirectly) together as the same river and separate the ones which do not intersect. In this procedure, the key of the type `string` is abandoned and a new key of the type `uint64_t` is used, it comes from the `id` of the first chosen edge.

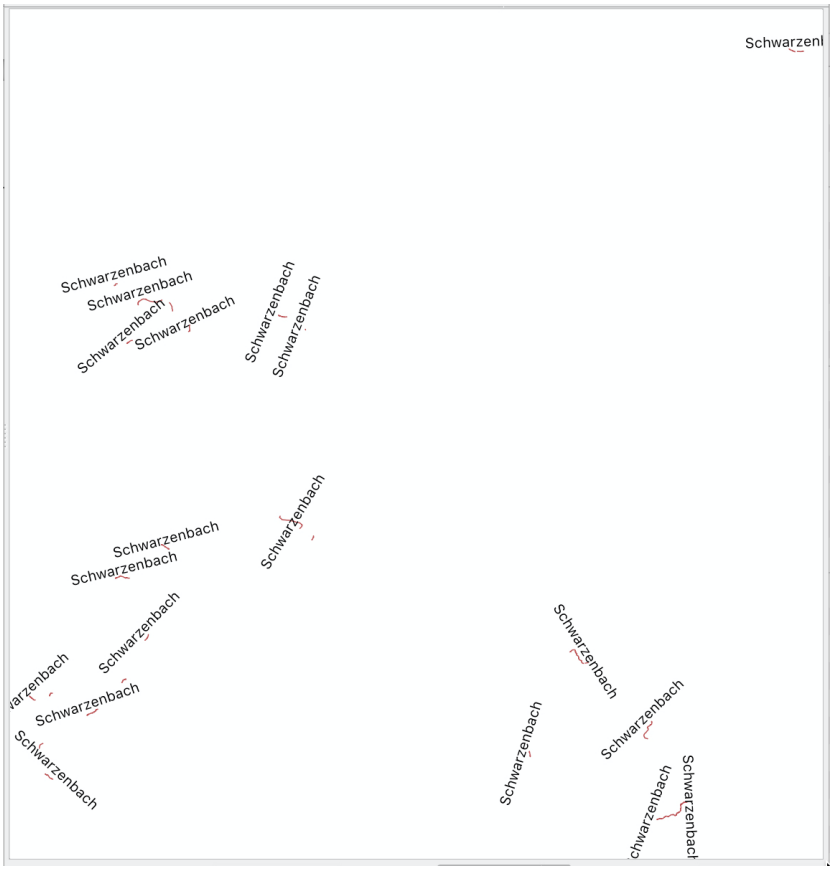


Figure 3.1: "Schwarzenbach" in the Black Forest

The value of the river map is the instance of class *River*. In this class, there are the following features:

- The length, the name, and the color of the river.
- Edge in river (`_eInR`), representing edges in this river, and it has the type `unordered_set`.

3.2.4 Storage Class

The extracted data of nodes, edges and rivers are stored in the class *RiverGraph*, it has three private map attributes representing the corresponding data types.

[1]https://en.wikipedia.org/wiki/OpenStreetMap
[2]https://wiki.openstreetmap.org/wiki/Tags
[3]https://en.wikipedia.org/wiki/Thalweg
[4]https://wiki.openstreetmap.org/wiki/Node
[5]https://wiki.openstreetmap.org/wiki/Elements
[6]https://wiki.openstreetmap.org/wiki/Way
[7]https://wiki.openstreetmap.org/wiki/Relation
[8]https://wiki.openstreetmap.org/wiki/Tag:waterway=river

4 Data Processing

After extracting and storing the needed information, we want to integrate our data into a directed graph so that we can analyze and rearrange it conveniently. Our final goal is to add the name of every tributary river to its stem river and gain an output in which the tributaries of the rivers can be easily observed. To illustrate this procedure, we will first show the baseline we set for the extracted data, and then introduce the data processing procedure in general. After that, we will split the procedure into six steps (section 4.2 to 4.7) and in each step, we explain why we do it like this and how we are doing it.

4.1 Baseline and Basic Procedure

4.1.1 Baseline

In terms of building a baseline, we keep the procedure in a simple way. We consider the following functions as the ones the baseline should have:

1. Using the river itself as its only upstream river.
2. Filtering river length by its own length without checking its upstream rivers' length.
3. Using the same color for all the river lines.

By following these key points for the baseline, all the waterway related data should be loaded from the `osm` file. After that, we do not reorganize the river edges into a directed graph and we sort the rivers right away. That means, we split the rivers with the same name but do not intersect with each other, and we use the `edge id`s as the keys of the river map instead of river names. The next step would be to add dummy upstream river names, that is, to add only the river itself as its upstream river. In the length filter part, the rivers are simply filtered by their own length. In the last step, all the edges will have the same blue color with the hex value `#0000ff`. The following is a graph illustrating the progress of the baseline.

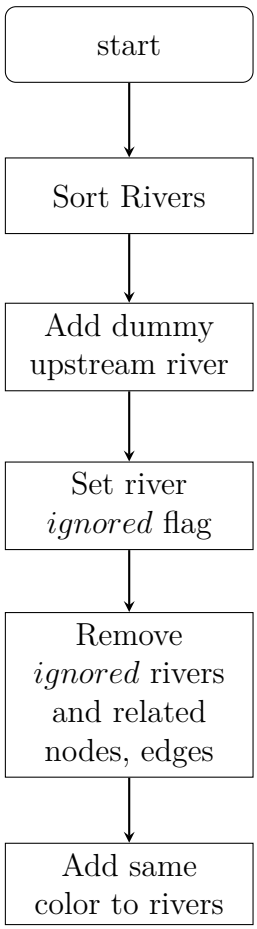


Figure 4.1: Process for baseline

4.1.2 Basic Procedure

As a comparison, we draw the procedure of our program as well. Comparing this with the baseline, it is easy to observe that the intersection points are clearly marked, the lines are colorful and the length filter is more accurate and reasonable. We will illustrate the respective steps (organizing nodes, sorting edges, sorting rivers, adding river names, length filter and adding river colors) in the next sections.

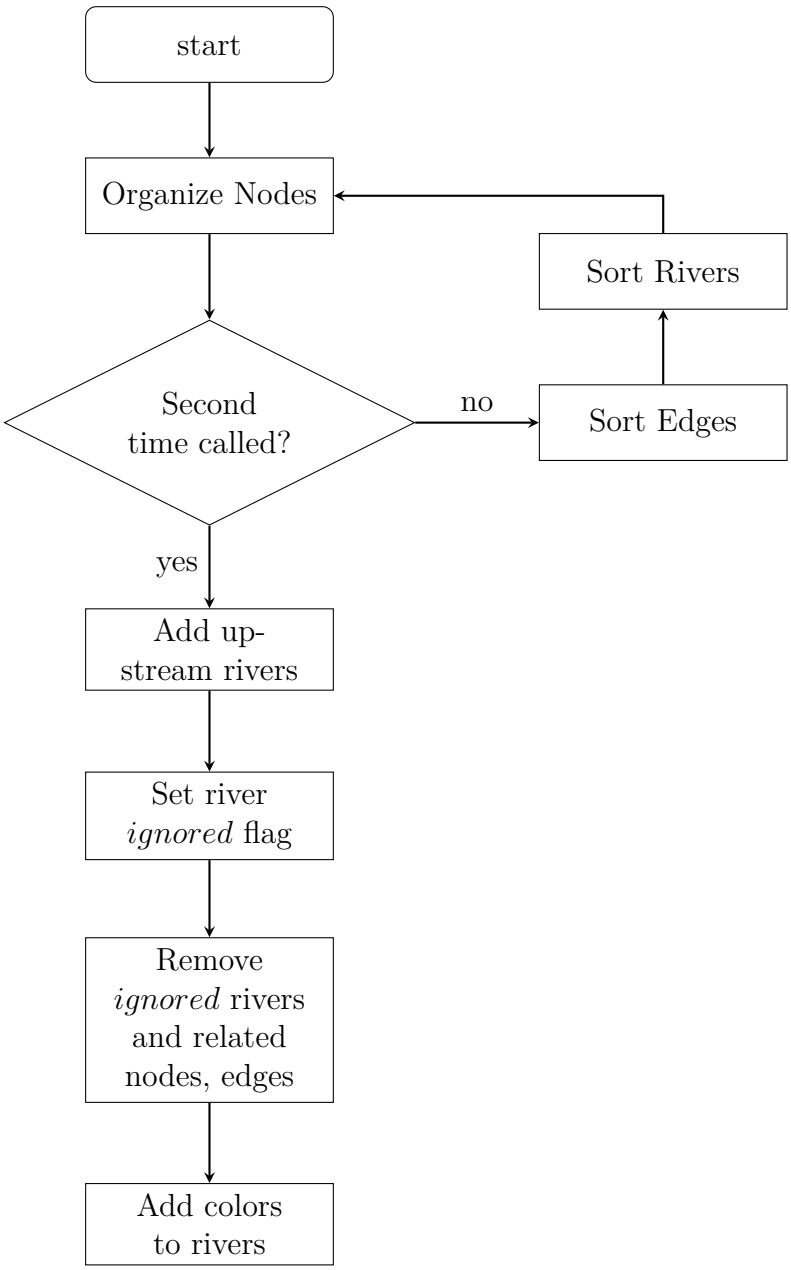


Figure 4.2: Process of our program

4.2 Organizing Nodes

First, we want to deal with the nodes of the graph. In the current river map data, if we want to go from any source to its sea, three kinds of points are significant.

- The starting points. An eligible source is the first node of a river edge and this node is not an intersect point with any other river edges.

- The intersection points. A node that is referenced by two or more edges is an intersection point.
- The destination points. A destination point is a qualified mouth, which is the last node of a river edge and does not intersect with any other river edges.

So in the first step, we want to mark these points, so that when we add the name of a tributary river to a stem river, we know that we are going in the right direction and no tributary river names are left out. We mark the `in` and `out` information for these significant nodes, the number of `in`s represents how many river edges are flowing into this node and the number of `out`s represents how many river edges are flowing from this node. By means of this step, the starting points will have `in` $= 0$ and `out` $\geqslant 1$, the intersection points will have `in` + `out` $\geqslant 2$ (`in` and `out` both $\neq 0$) and the destination points will have `in` $\geqslant 1$ and `out` $= 0$.

4.3 Sorting Edges

The next step is to organize the edges. There are two sub-steps: splitting edges and concatenating edges.

4.3.1 Splitting Edges

For the river edges which cross the intersecting nodes, we need to split them at the intersection point, in order to avoid mistakes in the name adding phase. We explain this via the following pictures.

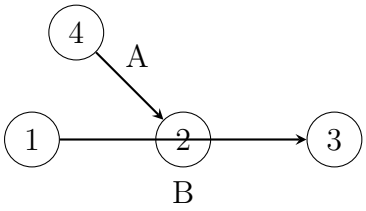


Figure 4.3: A possible intersecting point of rivers

Suppose we have two river edges `A` and `B`, and they intersects at the point `2`. If we add the name of river `A` directly into river `B`, then the result will be inaccurate, because river `A` is not yet a tributary of river `B` between points `1` and `2`. So we need to split rivers at intersection points to get the correct output, as shown in the next image.

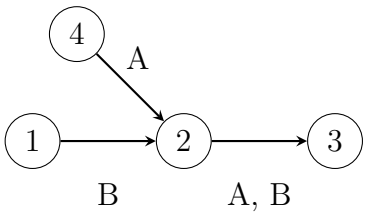


Figure 4.4: The proper result after adding names

In this procedure, if a river edge needs to be split, only the first section will keep the original edge `id`. For the next sections, new edges with new `id`s will be created, and the related information (river name, river type, etc.) will be copied to the new river edges.

4.3.2 Concatenating Edges

To make the directed graph more compact, we concatenate the two river edges with the same river name at the point that only these two edges intersect in a successive way, that is, the last node of the former edge intersects with the first node of the latter edge. In OSM data, there are many fragmented river edges which hinder the observation of the rivers, here is an example of two actual rivers before and after concatenation.

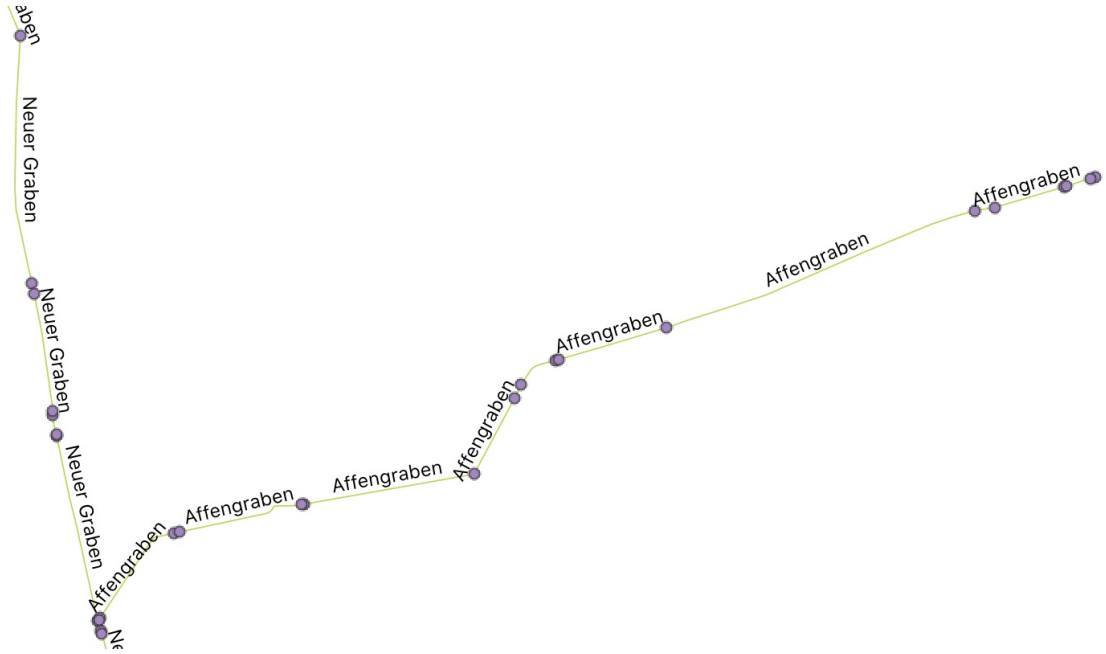


Figure 4.5: Two rivers before concatenation

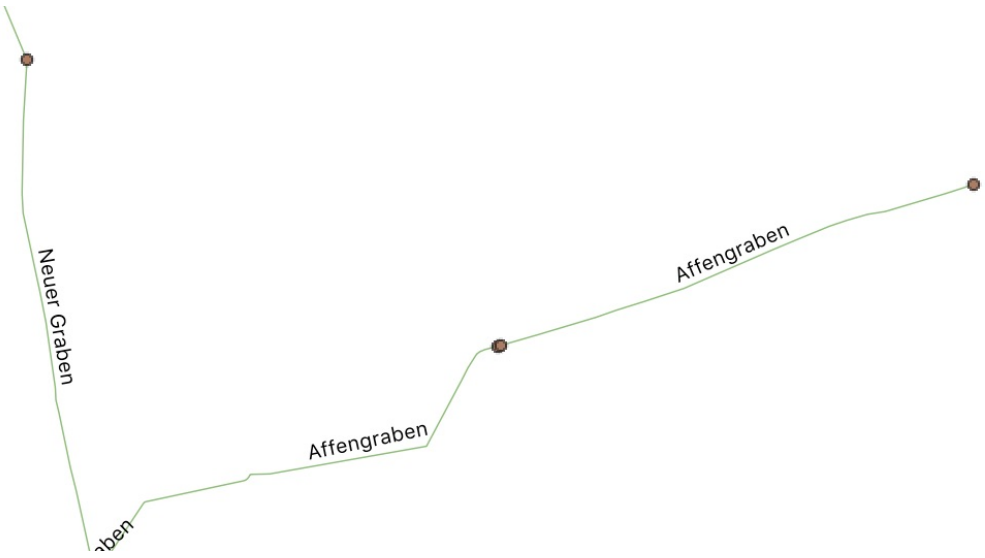


Figure 4.6: Same rivers after concatenation

As we can see, there is still a breakpoint in the river "Affengraben", this is because the central line of the two river edges does not overlap, that is, the last node of the prior river and the first node of the subsequent river are not the same node. This does not meet the topological rules of OSM data, and we will talk about this further in chapter *6*.

4.4 Sorting Rivers

In this step, as mentioned in the *River Map* section of chapter *3*, the river name map extracted from the original OSM data will be deleted and a new map with `river id`s as keys will be stored. The `river id` is the first explored `id` of the `edge` in the current river group. At the same time, the `river id` is also stored in the relevant edges of the edge map so that we can reference the river information from each edge.

After splitting edges, concatenating edges and sorting rivers, the node information (which nodes are starting points, intersection points or destination points) will probably differ from before, so we organize the nodes again to get the current result of `in`s and `out`s.

4.5 Adding River Names

Now we want to add the name of the tributary river to its stem river. First, we find globally the river source nodes and add them to the explore set.

Algorithm 1: Find all the global river source nodes

```
build_source_map (rg)
    input  : A RiverGraph rg
    outputs: A List of RiverNode l
             A Map of {RiverNode: Set of River} m
1   l ← List();
2   m ← Map();
3   foreach RiverEdge re ∈ rg do
4       RiverNode rn ← re.front_node();
5       if rn.in = 0 and rn ∉ l then
6           l.add(rn);
7           m.add(rn, {});
8   return l, m;
```

As mentioned in the section *Organizing Nodes*, the source nodes have `in` = `0` and `out` = `1`. We then add the river name of the edge beginning with the source node to the edges beginning with the last node of the before-mentioned edge. The last node of this edge must have `in` $\geqslant$ `1`. We subtract `1` from the `in` of this node. If the `in` of this node becomes `0`, then we add this node into the explore set. If a node has `out` = `0`, we wipe it out of the explore set. In this way, all the nodes with edges going in will be explored, and all the tributary river names will be added to the stem river.

Actually, this approach is a variant of the breadth-first search (BFS) algorithm. By using it, we can easily gather the upstream rivers, such that our final result is accurate.

To make the name adding procedure more clear, we write the pseudo-code here.

Algorithm 2: Add upstream rivers' names to each downstream river, and filter river by its total length

```
add_river_names (rg, riverLengthThreshold)
    inputs : A RiverGraph rg
             A length threshold value for filtering riverLengthThreshold
    output: The updated RiverGraph rg
1   exploreNodes, riversInNode ← build_source_map(rg);
2   while exploreNodes ≠ ∅ do
3       RiverNode rn ← exploreNodes.pop_front();
4       foreach RiverEdge re ∈ rn.referenced_edges() do
5           if rn ≠ re.front_node() then
6               continue
7           RiverNode rnNext ← re.back_node();
8           rnNext.in ← rnNext.in − 1;
9           rnNext.out ← rnNext.out − 1;
10          if rnNext ∉ riversInNode then
11              riversInNode.add(rnNext, {});
12          River river ← re.river();
13          if riversInNode[rn] ≠ ∅
             or river.length ≥ riverLengthThreshold then
                /* Condition 1:  the length of one of river's upstream
                   rivers ≥ riverLengthThreshold
                   Condition 2:  river's length ≥ riverLengthThreshold
                   */
14              river.ignored ← false;
                /* the default value of river.ignored is true          */
15              re.upstream_rivers ←
                 re.upstream_rivers ⋃ riversInNode[rn];
                /* extend re.upstream_rivers with rivers in
                   riversInNode[rn]                                    */
16              riversInNode[rnNext] ← riversInNode[rn];
17              riversInNode[rnNext].add(river);
                /* copy all rivers from riversInNode[rn] to
                   riversInNode[rnNext], and add current river to
                   riversInNode[rnNext] as well                        */
18          if rn.out = 0 then
19              riversInNode.remove(rn);
20          if rnNext.in = 0 then
21              exploreNodes.add(rnNext);
22  return rg;
```

4.6 Length Filter

During the name adding procedure, we also calculate the length of each river group and check if it is less than the given filter value. Once we find a river with qualified length, we will skip the length check for its downstream rivers. For edges related to the unqualified river group, they will be deleted. And then the nodes which are referenced only by the unrelated edges will also be removed. In the end, the unqualified river group will be removed from the data as well.

4.7 Adding River Colors

For the color-adding procedure, we use the HSV (hue, saturation, lightness) model rather than the RGB model, because the former is based more upon how colors are organized and conceptualized in human vision in terms of other color-making attributes, such as hue, lightness, and chroma; as well as upon traditional color mixing methods, e.g. in painting, that involve mixing brightly colored pigments with black or white to achieve lighter, darker, or less colorful colors[1]. Hence the HSV model is more comprehensible when we are choosing colors. Here is the pseudo-code of the color selecting process.

Algorithm 3: Generate given number human eyes distinct colors

```
generate_colors (totalUsedColors, avoidColors, num)
    inputs : A Set of HSV colors totalUsedColors
             A Set of HSV colors avoidColors
             An integer num
    output: A Set of distinct HSV colors result
    // minimum delta value for h, s, v
1   minDeltaH ← 30, minDeltaS ← 10, minDeltaV ← 10;
2   result ← {};
3   while result.size() < num do
4       hsv ← HSV(h = 0, s = 0, v = 0);
5       v ← a random integer in [19, 100];
6       if v < 20 then
            // avoid to get too many not distinguishable black colors
7           hsv.h ← 0, hsv.s ← 0, hsv.v ← 0;
8       else
9           hsv.h ← a random integer in [0, 359];
10          hsv.s ← a random integer in [0, 100];
11          hsv.v ← v;
12      if hsv.v = 100 and hsv.s < 5 then // ignore color white
13          continue
14      else if hsv ∈ avoidColors or hsv ∈ totalUsedColors then
            // avoid duplicated color
15          continue
16      else if avoidColors ≠ ∅ then
17          isQualified ← true;
18          foreach color ∈ avoidColors do
19              if hsv.v = color.v = 0 then
20                  isQualified ← false;
21                  break
22              else if |color.h − hsv.h| < minDeltaH
                 and |color.s − hsv.s| < minDeltaS
                 and |color.v − hsv.v| < minDeltaV then
23                  isQualified ← false;
24                  break
25          if isQualified = false then
26              continue
27      result.add(hsv);
28      avoidColors.add(hsv);
29  return result;
```

After having the right colors, we add these colors to our rivers.

Algorithm 4: Add color to each river

```
add_river_colors (rg)
    input  : A RiverGraph rg
    output: The updated RiverGraph rg
1   totalUsedColors ← {};
2   foreach RiverEdge re ∈ rg do
3       River river ← re.river;
4       if river ∉ re.referenced_rivers() then
5           re.add_referenced_river(river);
6       avoidColors ← {};
7       num ← 0;
8       foreach River r ∈ re.referenced_rivers() do
9           if r has a color then
10              avoidColors.add(r.color);
11          else
12              num ← num + 1;
13      generatedColors ←
         generate_colors(totalUsedColors, avoidColors, num);
14      foreach River r ∈ re.referenced_rivers() do
15          if r doesn't have a color then
16              color ← generatedColors.pop();
17              r.color ← color;
18              totalUsedColors.add(color);
19  return rg;
```

4.8 Exporting Data to GeoJSON

Our tool LOOM expects the graph as a GeoJSON file, so the last step is to export the output data in the right form. GeoJSON is an open standard format designed for representing simple geographical features, along with their non-spatial attributes[2]. The output file of our program consists of points and line strings in GeoJSON format.

The following is an example of an output file composed of two points and one edge. In this file, the edge indicating the Neckar with color `#a4b020` goes from the point "1656844591" to the point "507930239", its upstream river set ("lines") includes only the edge itself.


```
{
  "features":[
    {
      "geometry":{
        "coordinates":[945322.1435861, 6358143.7429898],
        "type":"Point"
      },
      "properties":{
        "color":"ff0000",
        "id":"1656844591"
      },
      "type":"Feature"
    },
    {
      "geometry":{
        "coordinates":[936620.2210671, 6376420.5050057],
        "type":"Point"
      },
      "properties":{
        "color":"ff0000",
        "id":"507930239"
      },
      "type":"Feature"
    },
    {
      "geometry":{
        "coordinates":[
          [945322.1435861, 6358143.7429898],
          [936620.2210671, 6376420.5050057]
        ],
        "type":"LineString"
      },
      "properties":{
        "color":"a4b020",
        "id":"514832816",
        "name":"Neckar",
        "length":"1.0633858",
        "from":"1656844591",
        "to":"507930239",
        "lines":[
          {
            "label":"Neckar",
            "color":"a4b020",
            "id":"514832816"
          }
        ],
        "type":"river"
```

```
48          },
49          "type":"Feature"
50       }
51    ],
52    "type":"FeatureCollection"
53 }
```

This step is accomplished by the class *GeoJSON*.

[1]https://en.wikipedia.org/wiki/HSL_and_HSV
[1]https://en.wikipedia.org/wiki/GeoJSON

5 Evaluation

We implemented our code on OSM maps of different sizes within Germany, from the city of Hamburg and Freiburg, to the state Bavaria, and finally to the map of the entire country. As far as we know, our project is the first work on this topic, so we do not have any other works to compare with. The following table shows the results of the tests on a laptop with Intel Core i7 2,7 GHz (4 Cores) with 16 GB Memory.

	Map size	Running time in seconds	Amount of points	Amount of lines
Hamburg	588 MB	19	1148	953
Freiburg	2.16 GB	72	7142	6377
Bavaria	10.52 GB	362	26832	23549
Germany	56.96 GB	2019	124610	106394

From these results, we can observe the following characters of the program:

- The runtime of the program is rather short compared to the size of the data.
- The code is robust in large and complicated data sets.
- When the size of the data gets larger, the runtime of the program presents a quasi-linear growth, which is efficient and healthy.

In development, we keep using simple and efficient data structures to avoid storing duplicated information. We implemented the program in a well-organized way, for example, we avoid nested loops which can cause exponential runtime growth. Added to this, we constantly test it with fully covered unit tests and various large data sets as robustness and stability measures.

6 Possible Problems and Future Work

6.1 Interruption Caused by Lakes

If a river drains into a lake, rather than into a sea, its route can be interrupted because there is no proper waterway between this river and the downstream river that it contributes to.

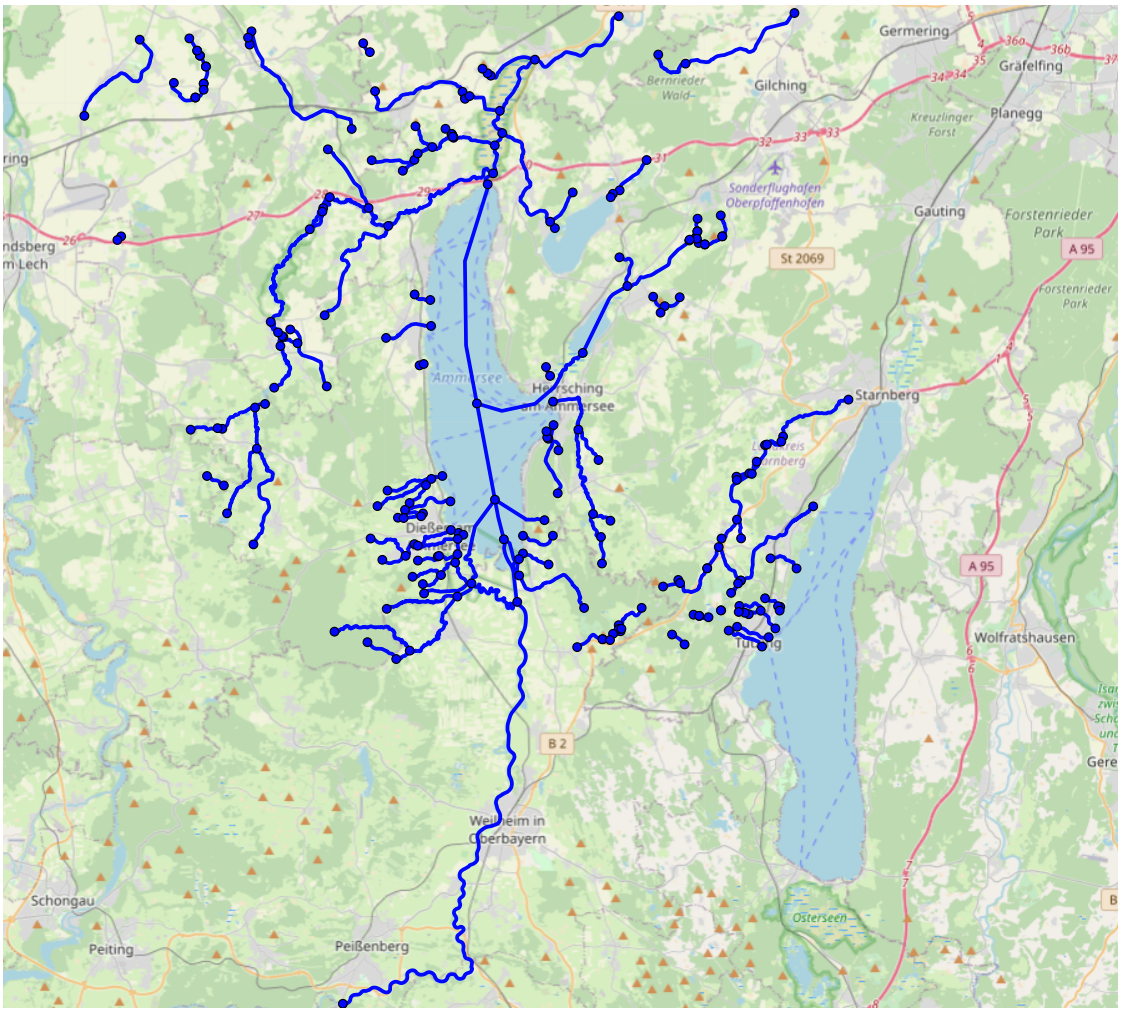


Figure 6.1: An example of interrupted rivers by a lake

The above image shows the waterways around Ammersee, which is a lake in Bayern, Germany. The main river called the Amper flowing from the south to the north through the lake Ammersee was linked by a waterway edge. But at the same time, we can see that many small rivers contributing to the lake were not lined to the main river. They contribute to the Amper and eventually to the Danube and the Black Sea. In OSM data, it is common that a river is cut off by a lake and thus can not run into the sea that it contributes to. Linking this kind of river to its stem would be an improvement in our project.

6.2 Inconsistent OSM Data

In reality, there are always OSM data that do not meet the topological rules, identifying and correcting this kind of data will also make our project better.

6.2.1 River Loops

Loops can occur in the river map data of OSM, it can be a result of a wrong record on river direction. For example, the following picture shows the `node id` and `id` of the beginning node for each edge, and we can see that the edges beginning with "1702935482", "254853968" and "254853942" build a loop.

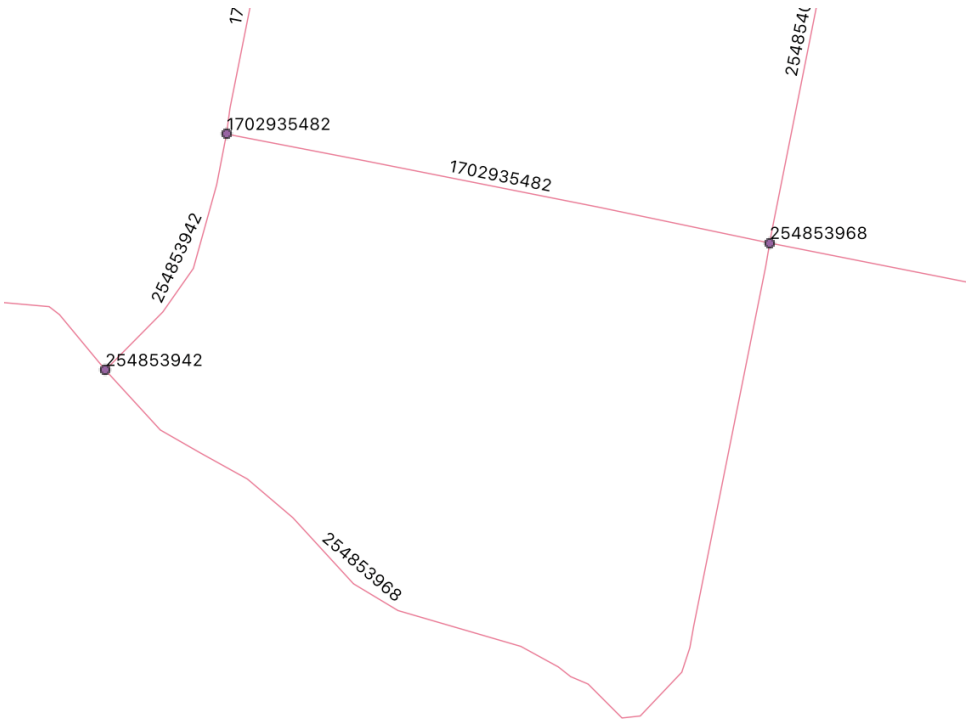


Figure 6.2: An example of loop in river

When there is a loop in the data, the river names will not be correctly added for the related rivers. The good news is that, due to our observation, river loops almost only happen on small rivers which are often filtered by the length condition.

6.2.2 Breaking Points

As mentioned in the section *Concatenating Edges* of the chapter *4*, breaking points lead to inaccuracy in the concatenation. Besides, breaking points also cause problems in adding river names. If two river edges are not properly linked at the same node, the upstream river names could not be passed to the next node and the river name information will be lost here.

6.2.3 Repeated Edges

Some edges in the OSM data are described repeatedly, and this will cause problems, especially in the concatenation part.

Figure 6.3: An example of repeated edges

Here is the ditch "Nördlicher Bahngraben" in Hamburg, it is marked in green color in the above picture. This ditch is represented by multiple edges, among them, four are expressed the same for two times (most of these are at the bottom right corner of the image). In this case, the intersection points of these edges will be regarded as a meeting point of three or four edges, therefore they can't be concatenated properly.

6.2.4 Wrong Direction of Rivers

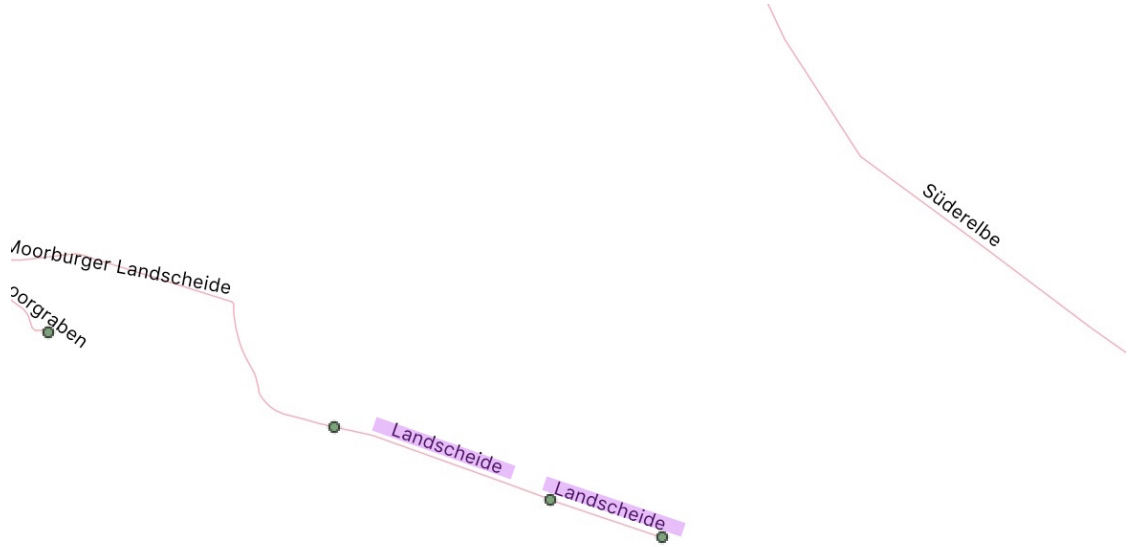


Figure 6.4: River with wrong direction

In this image, we have the canal “Landscheide” described in two parts, but these two edges have the same origin, which is the intersection point of them. Edges like this can also not be concatenated in the right way. In nature, it is hardly possible that the two rivers have the same source point. But in the OSM data, this could happen.

Bibliography

[ABKS08] Argyriou, Evmorfia N. ; Bekos, Michael A. ; Kaufmann, Michael ; Symvonis, Antonios: Two polynomial time algorithms for the metro-line crossing minimization problem. In: *Proceedings of the 16th International Symposium on Graph Drawing*, Springer, 2008 (GD'08), pp. 336–347

[ABKS10] Argyriou, Evmorfia N. ; Bekos, Michael A. ; Kaufmann, Michael ; Symvonis, Antonios: On metro-line crossing minimization. In: *J. Graph Algor. Appl.* Vol. 14, 1, 2010, pp. 75–96

[AGM08] Asquith, Matthew ; Gudmundsson, Joachim ; Merrick, Damian: An ILP for the metro-line crossing problem. In: *Proceedings of the 14th Symposium on Computing: The Australasian Theory* Vol. 77, Australian Computer Society, 2008, pp. 49–56

[AKPW15] Ahmed, Mahmuda ; Karagiorgou, Sophia ; Pfoser, Dieter ; Wenk, Carola: A comparison and evaluation of map construction algorithms using vehicle tracking data. In: *Geoinformatica 19* Vol. 3, 2015, pp. 601–632

[BBS18] Bast, Hannah ; Brosi, Patrick ; Storandt, Sabine: Efficient Generation of Geographically Accurate Transit Maps. In: *26th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*. New York, NY, USA : ACM, November, 2018 (SIGSPATIAL'18)

[BBS20] Bast, Hannah ; Brosi, Patrick ; Storandt, Sabine: Metro Maps on Octilinear Grid Graphs. In: *EuroVis Eurographics Conference on Visualization*, 2020 (Band: 39, Nummer: 3)

[BKPS08] Bekos, Michael A. ; Kaufmann, Michael ; Potika, Katerina ; Symvonis, Antonios: Line crossing minimization on metro maps. In: *Proceedings of the 15th International Conference on Graph Drawing*, Springer, 2008 (GD'07), pp. 231–242

[BNUW07] Benkert, Marc ; Nöllenburg, Martin ; Uno, Takeaki ; Wolff, Alexander: Minimizing intra-edge crossings in wiring diagrams and public transportation maps. In: *Proceedings of the 14th International Conference on Graph Drawing*, Springer, 2007 (GD'06), pp. 270–281

[FHN+13] Fink, Martin ; Haverkort, Herman ; Nöllenburg, Martin ; Roberts, Maxwell ; Schuhmann, Julian ; Wolff, Alexander: Drawing metro maps using Bézier curves. In: *Proceedings of the 20th International Conference on Graph Drawing*, Springer, 2013 (GD'12), pp. 463–474

[HMN06] Hong, Seok-Hee ; Merrick, Damian ; do Nascimento, Hugo A. D.: Automatic visualisation of metro maps. In: *J. Vis. Lang. Comput. 17* Vol. 3, 2006, pp. 203–224

[Nöl10] Nöllenburg, Martin: An improved algorithm for the metro-line crossing minimization problem. In: *Proceedings of the 17th International Conference on Graph Drawing*, Springer, 2010 (GD'09), pp. 381–392