Bachelor's Thesis

Annotating OpenStreetMap data with elevation data

Urs Spiegelhalter

Examiner: Prof. Dr. Hannah Bast
Advisers: Patrick Brosi

University of Freiburg
Faculty of Engineering
Department of Computer Science
Chair for Algorithms and Data Structures

February 16th, 2022

Writing Period

16. 11. 2021 – 16. 02. 2022

Examiner

Prof. Dr. Hannah Bast

Advisers

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.

Place, Date	Signature

Abstract

We present *osmelevation*, a tool to annotate OpenStreetMap data with elevation data obtained from the NASA Digital Elevation Model (NASADEM). By adding a tag *ele=** containing the elevation to each node, we enrich the complete OpenStreetMap data with elevation data.

NASADEM provides near-global coverage and a horizontal resolution of 1 arc-second, which is approximately 30 meters. This results in 376GB of uncompressed elevation data. In combination with the ever-growing OpenStreetMap data, our tool manages the large amount of data by splitting both the OpenStreetMap and elevation data into smaller data pieces and processing them one by one.

Furthermore, we provide a second tool *correctosmelevation*. Based on already with elevation data annotated OpenStreetMap data from our first tool, we perform simple corrections on the elevation data. As a basis, we look at OpenStreetMap linear route map features like roads or rivers. Even with the high horizontal resolution of approximately 30 meters of the NASA Digital Elevation Model, obvious errors can be observed. For example, roads suddenly dropping by five meters, or rivers flowing uphill. Our tool *correctosmelevation* corrects those errors and updates the corresponding *ele=** tags with the corrected elevation.

Zusammenfassung

Wir präsentieren *osmelevation*, ein Werkzeug, um OpenStreetMap Daten mit externen Höhendaten von dem NASA Digital Elevation Model anzureichern. Wir fügen zu jedem Node einen Tag *ele=** hinzu.

NASADEM stellt eine nahezu globale Abdeckung bereit und verfügt über eine horizontale Auflösung von ungefähr 30 Metern. Das resultiert in 376GB unkomprimierten Daten. In Kombination mit den immer weiter wachsenden OpenStreetMap Daten verwaltet unser Werkzeug diese enormen Datenmengen, indem sowohl die Höhendaten, als auch die OpenStreetMap Daten aufgeteilt werden. Die aufgeteilten Daten werden nacheinander abgearbeitet.

Außerdem präsentieren wir ein zweites Werkzeug *correctosmelevation*. Basierend auf bereits mit Höhendaten angereicherten OpenStreetMap Daten führen wir einfache Korrekturen an den Höhendaten aus. Als Basis betrachten wir lineare OpenStreetMap Kartenmerkmale, wie zum Beispiel Straßen oder Flüsse. Obwohl NASADEM eine hohe horizontale Auflösung von ungefähr 30 Metern aufweist, können offensichtliche Fehler erkannt werden. Zum Beispiel können Straßen plötzlich um 5 Meter fallen, oder Flüsse können bergauf fließen. Unser Werkzeug *correctosmelevation* korrigiert solche Fehler und aktualisiert die dazugehörigen *ele=** Tags.

Contents

List of Figures

List of Tables

List of Algorithms

List of Listings

1 Introduction

OpenStreetMap stores, among other data, many different map features, e.g. roads, buildings or rivers. To represent these map features, a topological data structure of nodes, ways and relations is used with tags attached to each. Nodes store at least an id and the location of a single point in space by its longitude and latitude. Nodes can be expanded by the third dimension by adding a tag *ele=** containing the elevation above sea level in meters of the location. However, this tag is rarely used.

Publicly available elevation data vary in accuracy and coverage. The NASA Digital Elevation Model (NASADEM) is based on the Shuttle Radar Topography Mission conducted by NASA in 2000. The elevation data from NASADEM is distributed in the NASADEM_HGT dataset. The elevation data from this dataset is among the most accurate by offering a horizontal resolution of 1 arc-second, which is approximately 30 meters and near-global coverage. This results in large amounts of raw elevation data that are not directly usable in practical applications.

We present our tool *osmelevation* that adds the tag *ele=** to all nodes in OpenStreetMap containing the elevation of the node's location. By using the nodes of OpenStreetMap as the carriers, the elevation data directly gets propagated to the map features they are apart of. This significantly reduces the size of the raw elevation data from the NASADEM_HGT dataset while maintaining the essential data for the map features.

On uneven terrain, the horizontal resolution of approximately 30 meters used by the NASA Digital Elevation Model (NASADEM) is not sufficient to accurately represent the elevation of narrow map features like roads or rivers. Sudden drops and spikes in elevation by multiple meters can occur. Additionally, noise in NASADEM elevation data can contribute to such errors.

Therefore, we present a second tool *correctosmelevation* that performs corrections on already with elevation data annotated OpenStreetMap data. We apply a smoothing algorithm to all map features that can be traveled by car, train, bike, foot, or ship. Additionally, we correct other obvious errors found in roads passing through tunnels or crossing bridges and make sure that rivers do not flow uphill.

1.1 Problem

1.1.1 Elevation data in OpenStreetMap

OpenStreetMap proposes to use the tag *ele=** [11] to denote the elevation above sea level of a point or map feature. The elevation value should be measured in meters. We can use the website taginfo [10] to inspect the usage of this tag. The result can

Type	Number of objects	Usage of total number of objects
All	7.653.471	0.09%
Node	2.254.791	1.20%
Way	5.259.155	0.63%
Relation	139.525	1.45%

Table 1: *key:ele* tag usage in OpenStreetMap objects. Data provided by taginfo [8]. © OpenStreetMap contributors

be seen in Table 1. Clearly, the overall usage of 0.09% indicates that elevation data in OpenStreetMap is very lacking. Stating the OpenStreetMap wiki [11], this is by design:

> OpenStreetMap does not try to be a general elevation database so you should not tag elevation of nodes with no other tags and no specific meaning that suggests an elevation value is significant information in this case.

1.1.2 OpenStreetMap and elevation data size

We use the NASADEM_HGT[1] dataset from the NASA Digital Elevation Model (NASADEM) as the external source of elevation data. This dataset is available in the public domain. NASADEM_HGT provides a horizontal resolution of 1 arc-second, which is approximately 30 meters. With data being available from 180°W to 180°E and from 56°S to 60°N, it provides near-global coverage.

The elevation data in the NASADEM_HGT dataset is distributed in tiles of size 1 degree longitude by 1 degree latitude. The elevation data of each tile is compressed

[1] `https://lpdaac.usgs.gov/products/nasadem_hgtv001/`

into a single file and can be downloaded over the U.S. Government Computer FTP-Server[2].

For performance, it is essential that the uncompressed elevation data is stored in the main-memory at the time of access. As seen in Table 2, with an uncompressed size of 371GB, it is not feasible to just load all data into main-memory at once.

Number of files	Total size compressed	Total size uncompressed
14520	109GB	376GB

Table 2: NASADEM_HGT dataset size. The NASADEM_HGT dataset also contains images and meta data files for each tile. These files are excluded in this table as we do not need them. Additionally, the compressed files containing the raw elevation data for each tile also contain other data that we do not need. Hence, this data is not considered in the total uncompressed size.

Number of nodes	Number of ways	Number of relations
7.468.758.706	832.784.309	9.611.640

Table 3: Total number of objects in the OpenStreetMap. © OpenStreetMap contributors

When working with the complete OpenStreetMap data, two challenges arise. First, OpenStreetMap stores large amounts of data. As shown in Table 3, there are already over seven billion[3] nodes included in OpenStreetMap. This leads directly to the second challenge: The constant growth of OpenStreetMap data. Currently, about two million nodes[4] are added to OpenStreetMap everyday. That means our tools do not only need to work today, but also need reserves to still work with hundreds of millions of additional elements.

To cope with the main-memory constraints the large amounts of data introduce, we split both the elevation and OpenStreetMap data into smaller pieces of data. In our tools, we do not use any kind of disk storage as this would greatly reduce performance.

[2] `https://e4ftl01.cr.usgs.gov/MEASURES/NASADEM_HGT.001/2000.02.11/`
[3] `https://www.openstreetmap.org/stats/data_stats.html`
[4] `https://osmstats.neis-one.org/?item=elements`

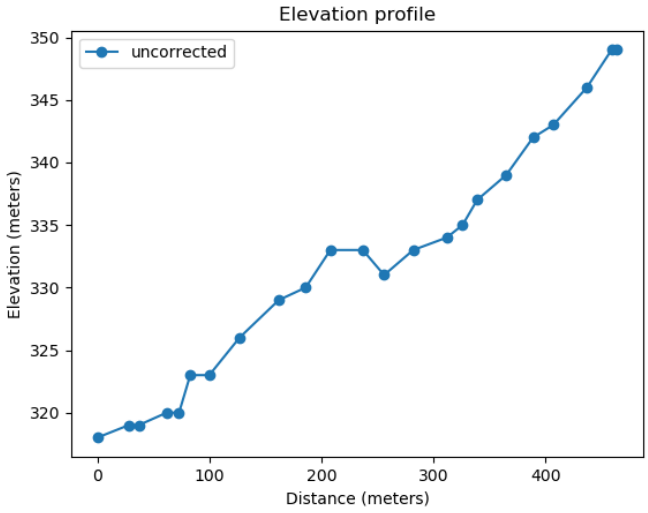

Figure 1: Uncorrected elevation profile of ascending road. Plot generated from data of the OpenStreetMap ways *19794411*, *30354354*, *319722236*, *143661523*. The dots represent the nodes along the road. Our tool *osmelevation* was used the to add the elevation data to OpenStreetMap.

1.1.3 Errors in the elevation data

For linear map features like roads, railways or rivers, the horizontal resolution of approximately 30 meters provided by the NASADEM_HGT elevation dataset can lead to obvious errors. Especially on uneven terrain, errors can be observed due to the data not being accurate enough. In Figure 1, we can see the elevation profile graph of a strictly ascending road that runs on a side slope. The shown segment has a length of 464 meters and gains 31 meters in elevation. The elevation profile shows short term fluctuations that are not expected for a road that can be traveled by car.

Furthermore, for specific map features, the correct elevation data is simply not available. These include roads going through tunnels or crossing bridges. Since the NASADEM_HGT dataset provides at least the elevation of the ground level, tunnels or bridges get assigned the elevation of the mountain or valley they cross respectively.

In Figure 2 the elevation profile of the Gotthard Road Tunnel[5] is shown. Clearly, this elevation data is completely wrong for a road that goes through a mountain.

[5] `https://en.wikipedia.org/wiki/Gotthard_Road_Tunnel`

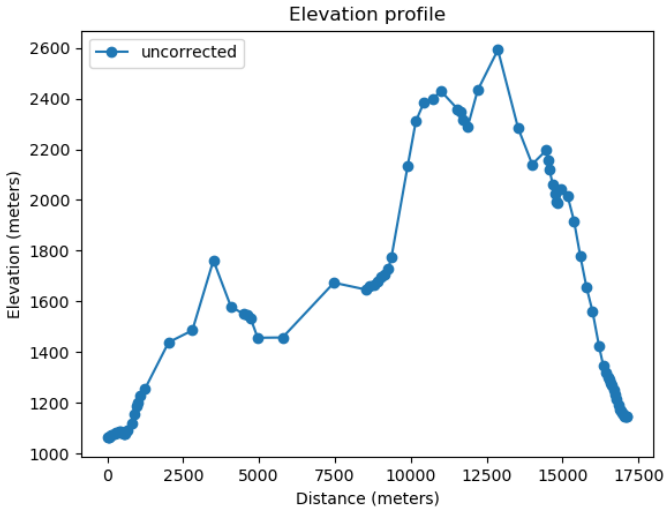

Figure 2: Uncorrected elevation profile of road going through a tunnel. Plot generated from data of the OpenStreetMap ways *304476718*, *4214708*, *49124512*, *431339440*, *29736069*. The dots represent the nodes along the road. Our tool *osmelevation* was used the to add the elevation data to OpenStreetMap. © OpenStreetMap contributors

The same occurs when roads cross steep valleys over bridges.

1.2 Contribution

Our tool *osmelevation* can be used to enrich OpenStreetMap data with external elevation data obtained from the NASADEM_HGT dataset from the NASA Digital Elevation Model. We do so by adding a tag *ele=** to each node of OpenStreetMap.

We provide a second tool *correctosmelevation* that performs simple corrections based on *ele=** tags. We specifically look at linear route map features in OpenStreetMap data. This includes all routes that can be traveled by car, train, bike, foot, or ship. Our tool looks at the elevation profiles of these routes. Obvious errors like sudden drops or spikes in elevation get corrected. As shown in Figure 3, we successfully smooth short term fluctuations in the elevation data, visualized by the red graph. Our tool ensures that rivers do not flow uphill. Additionally, the elevation profiles of routes going through tunnels or crossing bridges get corrected. Nodes that are affected by our corrections get their *ele=** tag updated.

In Figure 4, the corrected elevation profile of our previous example of the Gotthard

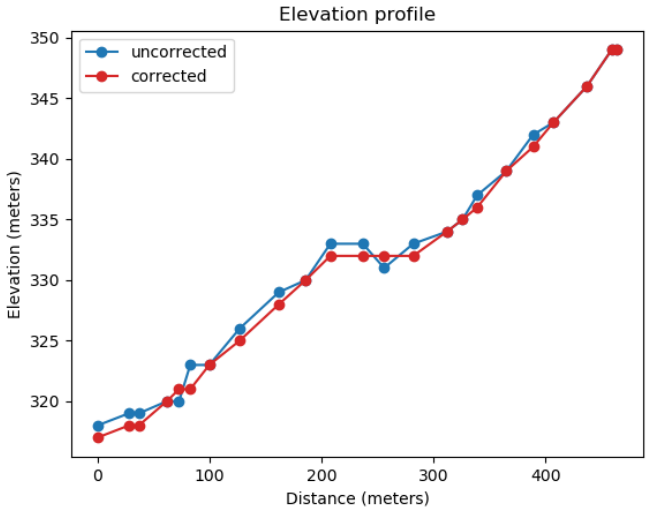

Figure 3: Uncorrected and corrected elevation profile of ascending road. Plot generated from data of the OpenStreetMap ways *19794411*, *30354354*, *319722236*, *143661523*. The dots represent the nodes along the road. Our tool *osmelevation* was used the to add the elevation data to OpenStreetMap. © OpenStreetMap contributors

Road Tunnel is shown in red. In Figure 18, we can see the uncorrected and corrected elevation profile of the *Kocher Viaduct*[6]. The bridge crosses a deep valley. The node seen in the middle of the graph takes the elevation of the valley in the uncorrected profile. With our corrections, we adjust the elevation of the node to the road elevation level in front of and after the bridge.

Both of our tools work with any size of OpenStreetMap data, ranging from small regional extracts to the whole planet.

[6] `https://en.wikipedia.org/wiki/Kocher_Viaduct`

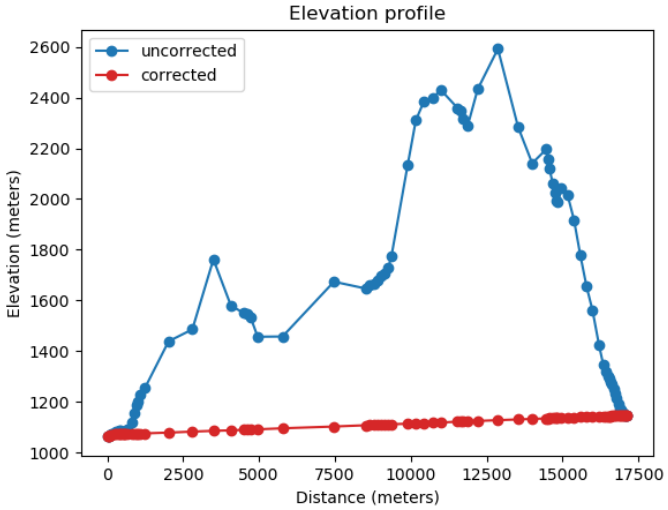

Figure 4: Uncorrected and corrected elevation profile of road going through a tunnel. Plot generated from data of the OpenStreetMap ways *304476718*, *4214708*, *49124512*, *43133944O*, *29736069*. The dots represent the nodes along the road. Our tool *osmelevation* was used the to add the elevation data to OpenStreetMap. © OpenStreetMap contributors

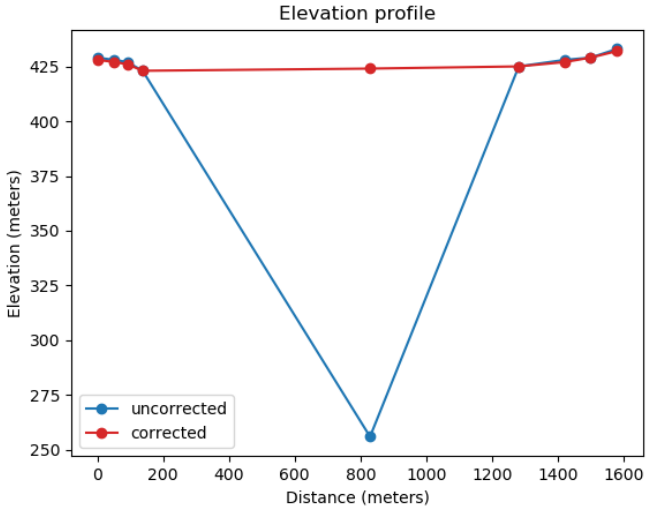

Figure 5: Uncorrected and corrected elevation profile of road crossing a bridge over a valley. Plot generated from data of the OpenStreetMap ways *403909403*, *320517373*, *320517370*, *24625636*, *24625669*. The dots represent the nodes along the road. Our tool *osmelevation* was used the to add the elevation data to OpenStreetMap. © OpenStreetMap contributors

2 Related Work

Our tools *osmelevation* and *correctosmelevation* both use OpenStreetMap data and external elevation data. In this chapter we look into already existing OpenStreetMap tools that also make use of external elevation data.
Before looking into other tools, we give a short overview of external elevation datasets.

2.1 Global digital elevation models

An important aspect of the usefulness of our tools is the quality of the external elevation data. There are several digital elevation models (DEMs) freely available. Coverage and accuracy of the DEMs are most important. Since OpenStreetMap stores map data for the entire planet, global or near-global coverage of the DEM is a necessary requirement. As for accuracy, a horizontal resolution of 1 arc-second, or 30 meters is the state of the art of freely available DEMs.
As already mentioned, we use the NASADEM digital elevation model (DEM). It provides near global coverage and a horizontal resolution of 1 arc-second, which is approximately 30 meters.

In the following, we will shortly present the three other freely available DEMs with the same or better characteristics as NASADEM and give a short insight why we did not choose them. For accuracy comparisons, we use the 2020 article "Vertical Accuracy of Freely Available Global Digital Elevation Models (ASTER, AW3D30, MERIT, TanDEM-X, SRTM, and NASADEM)" [2].

2.1.1 Advanced Spaceborne Thermal Emission and Reflection Radiometer (ASTER)

The Advanced Spaceborne Thermal Emission and Reflection Radiometer (ASTER) is an optical instrument operating onboard the Terra spacecraft[1]. Terra was launched

[1] `https://www.nasa.gov/mission_pages/terra/spacecraft/index.html`

in 1999 by the National Aeronautics and Space Administration (NASA). It collects data since 2000. The ASTER instrument acquires images in three bands which allow the creation of digital elevation models.
In 2019, the ASTER Global Digital Elevation Model (GDEM) Version 3 was publicly released. It provides data between 83° north latitude and 83° south latitude with an horizontal resolution of 1 arc-second[2].
Also, with ASTER GDEM version 3 redistribution requirements of the data were removed [1].
We did not choose this digital elevation model (DEM) because it delivers the least accuracy out of the DEMs with 1 arc-second horizontal resolution [2].

2.1.2 Shuttle Radar Topography Mission (SRTM)

The Shuttle Radar Topography Mission (SRTM) was conducted in 2000 as a cooperative project between the National Aeronautics and Space Administration (NASA) and the National Geospatial-Intelligence Agency (NGA), as well as participation from the German and Italian space agencies. On an 11-day flight of Space Shuttle Endeavour, onboard dual radar antennas acquired elevation data reaching between 60° north latitude and 56° south latitude. The data provided an horizontal resolution of 1 arc-second [4]. Due to limitations of radar technology, voids are present in the data, especially over water bodies.
In 2015, the latest version of the SRTM digital elevation model was released by NASA, namely SRTM NASA Version 3 [5]. Version 3 uses data from the previous version, SRTM Version 2. Additionally, Version 3 fills voids existing in the previous version by incorporating data mostly from ASTER Global Digital Elevation Model Version 2 (GDEM 002)[3].
To conclude, the SRTM Version 3 digital elevation model provides near-global coverage and a horizontal resolution of a 1 arc-second. As SRTM data is distributed by NASA's Land Processes Distributed Active Archive Center (LP DAAC), it is in the public domain [1].
SRTM Version 3 was overall suitable for our tools, but delivers slightly worse accuracy than NASADEM [2].

2.1.3 ALOS Global Digital Surface Model

The Advanced Land Observing Satellite (ALOS) was a Japanese Earth-imaging satellite that operated from 2006 to 2011. The optical PRISM instrument was used

[2] `https://lpdaac.usgs.gov/products/astgtmv003/`
[3] `https://lpdaac.usgs.gov/products/astgtmv002/`

for digital elevation mapping[4].
From the data obtained with PRISM, the commercial "ALOS World 3D" (AW3D) digital elevation model was created. It offers global coverage and an horizontal resolution of 5 meters [7].
Later, the Japan Aerospace Exploration Agency (JAXA) released the "ALOS World 3D-30m" (AW3D30) digital elevation model. It is based on AW3D data and is publicly available. AW3D30 offers the same global coverage and a horizontal resolution of 30 meters[5].
Contrary to the SRTM and ASTER digital elevation models, JAXA must be credited as the original distributor when using AW3D30 data[6].
Out of NASADEM, ASTER GDEM Version 3 and SRTM Version 3, AW3D30 provides the best accuracy [2]. But since AW3D30 is not in the Public Domain, it can not be used to update OpenStreetMap data. This makes AW3D30 unsuitable for our tools purposes as it conflicts with the OpenStreetMap license.

2.2 Existing OpenStreetMap tools using external elevation data

2.2.1 Relief/Contour maps

The vast majority of existing OpenStreetMap tools that use external elevation data are tools that produce relief/contour maps[7]. They do so by adding new ways and their corresponding nodes to OpenStreetMap. These new ways represent contour lines which each have a constant elevation. Mostly hiking maps make use of these contours, and there exist several map and tile servers that can visualize the contours[8].
This approach adds new elements to the OpenStreetMap which contain the elevation data while our tools *osmelevation* and *correctosmelevation* edits/updates existing OpenStreetMap objects.

2.2.2 Srtm to Nodes

The tool *Srtm to Nodes*[9] provides, according to the README, the same functionality as our tool *osmelevation* by adding a tag with the elevation data to all nodes. It uses

[4] `https://www.eorc.jaxa.jp/ALOS/en/alos/a1_about_e.htm`
[5] `https://www.eorc.jaxa.jp/ALOS/en/dataset/aw3d30/aw3d30_e.htm`
[6] `https://earth.jaxa.jp/en/data/policy/`
[7] `https://wiki.openstreetmap.org/wiki/Relief_maps`
[8] `https://wiki.openstreetmap.org/wiki/Hiking_Maps`
[9] `https://github.com/locked-fg/osmosis-srtm-plugin`

the SRTM digital elevation model.

Srtm to Nodes is an Osmosis[10] extension. Osmosis is a command line java application for processing OpenStreetMap data. Despite our best efforts, we could not get *Srtm to Nodes* to run on our machine. The tool was last maintained in November 2017. What we can say, *Srtm to Nodes* is, according to the README only compatible with the SRTMGL3 digital elevation model. SRTMGL3 provides a horizontal resolution of 3 arc-seconds, which is approximately 90 meters. The data was obtained by averaging the higher resolution data from SRTM Version 3. Since our tool uses NASADEM data with a horizontal resolution of 30 meters, we assume there is a noticeable difference in accuracy in favor of our tool. Also, *Srtm to Nodes* does not perform any corrections on the elevation data as our second tool *correctosmelevation*.

[10] `https://wiki.openstreetmap.org/wiki/Osmosis`

3 Background

In this chapter we introduce OpenStreetMap regarding the aspects we use in our tools. We introduce the NASADEM Digital Elevation Model (DEM) and explain how elevation data can be retrieved from this DEM. Furthermore, we introduce the inverse distance weighting interpolation algorithm and a simple moving average algorithm.

3.1 OpenStreetMap

OpenStreetMap stores huge amounts of data with the main objective being map features such as roads, buildings, or rivers. Additionally, a wide variety of to the map features related data is stored in OpenStreetMap, for example public transport data. All this data was and is contributed by volunteers around the globe who constantly submit changes and updates to the data. The OpenStreetMap data is free and open for everyone[1]. In this section we provide a short overview on how OpenStreetMap stores map features. We present the underlying data structure of OpenStreetMap.

3.1.1 Tags

To denote map features OpenStreetMap uses *tags*. A tag is a *key=value* pair. For the key and the value, any UTF-8 string can be chosen. For example, the tag *highway=footway* is used to denote a footpath for pedestrians. Also, any additional information for map features can be added as tags such as street names.
The OpenStreetMap wiki[2] provides conventions on how to tag common map features to ensure conformity.

[1] `https://wiki.osmfoundation.org/w/index.php?title=Mission_Statement`
[2] `https://wiki.openstreetmap.org/wiki/Tagging`

3.1.2 Objects

Map features in OpenStreetMap are built by using the objects nodes, ways, and relations. All objects can have one or more tags to denote the meaning of the map feature. They are the basic components of the data structure of OpenStreetMap.

A *node* represents a single point on the surface of the earth. A node at least stores a unique id and its location by longitude and latitude. For standalone map features, for example a waste basket, a node is used to represent them in OpenStreetMap.

For simple linear map features like roads or rivers, the *way* objects are used. A way combines at least two nodes which form a line. The way stores at least an unique id, a tag, and an ordered list of the node ids it uses.
Ways are also used to represent simple buildings or other closed areas. This is accomplished by forming a closed line with the nodes, i.e. the first and last nodes in the ordered list of nodes are the same.
If a map feature has a direction, for example a one-way road, the ordered list defines the direction with the start at the first and the end at the last node.

For more complex map features that can not be build by a single way, OpenStreetMap provides *relation* objects. A relation stores at least a unique id, a tag, and a list of objects by id, possibly even other relations. The objects are the *members* of the relation. The members and tags combined form a complex map feature.
Each member can optionally be assigned a *role* within the relation. For example, a member way can be given the role *backward*. This means the direction of the way should be used against the order of the way's node ids list.

To conclude, OpenStreetMap uses a data structure of nodes, ways, and relations with tags attached to each to store map features. An important property of this structure is that it is hierarchical. Relations consist of ways and nodes and ways consist of nodes. This means that changes to nodes directly get propagated to all objects they are part of, the same applies for ways that are part of relations.

3.2 NASADEM

In this section we present the NASADEM digital elevation model (DEM). We explain the used file structure and how elevation data can be extracted from the DEM.

The National Aeronautics and Space Administration digital elevation model (NASADEM) is derived from the original telemetry data from the SRTM project as presented in Section 2.1.2. The data was completely reprocessed with improved algorithms

which led to fewer data voids and overall higher vertical accuracy. To fill remaining voids, interpolation and newer data from ASTER Global Digital Elevation Model (GDEM) and ALOS World 3D 30-meter (AW3D30) digital elevation model were used[3].
The NASADEM digital elevation model provides a horizontal resolution of 1 arc-second, which is approximately 30 meters, and a near-global coverage by ranging from 180°W to 180°E and from 56°S to 60°N.

3.2.1 Obtaining NASADEM data

The NASADEM digital elevation model is distributed by the National Aeronautics and Space Administration's (NASA) Land Processes Distributed Active Archive Center (LP DAAC). The data of the DEM is available to download for free over a U.S. Government Computer FTP-Server[4]. It is required to create an Earthdata Login Profile to download from the server. Other than that, the data can be directly downloaded.
The data from the NASADEM digital elevation model is divided into data tiles of size 1 degree longitude by 1 degree latitude. The elevation data for each tile is stored in one file. As already shown in Table 2, this results in 14520 single files. Furthermore, each file is archived using the ZIP file format and includes other meta data. Each archive must be downloaded individually, there is no bulk download available.

3.2.2 NASADEM data files

Archive: NASADEM_HGT_n31e038.zip
n31e038.hgt
n31e038.num
n31e038.swb

Table 4: Contents of single NASADEM archive file.

In Table 4, the contents of a single NASADEM archive file are shown. Each archive is labeled with the name of the digital elevation model ("NASADEM_HGT") and the southwest corner of the tile the archive corresponds to. From the example in Table 4, the archive is named *NASADEM_HGT_n31e038.zip*. Thus this archive

[3] `https://lpdaac.usgs.gov/documents/1273/NASADEM_User_Guide_V11.pdf`
[4] `https://e4ftl01.cr.usgs.gov/MEASURES/NASADEM_HGT.001/2000.02.11/`

contains the elevation data from 31°N to 32°N and 38°E to 39°E.
Each tile archive contains 3 files with the same naming scheme for the southwest coordinate:

- The *.hgt* file contains the actual elevation data for the tile. We will introduce this file format in the next section.

- The *.num* file includes information from which source each data point in the *.hgt* file originates. For example, data can come from the SRTM, ASTER, or AW3D30 digital elevation model as explained in Section 3.2.

- The *.swb* file provides the water body data of the tile. The file contains coastline outlines if present in the tile for example.

Since we do not use the data of the *.num* and *.swb* files, we will not further elaborate on their internal formats.

3.2.3 Hgt file format

The elevation data from the NASADEM digital elevation model is provided by files of type *.hgt*. A single hgt file contains the elevation data for a tile of size 1 degree longitude by 1 degree latitude. The tile's location is encoded into the file name as we have seen in Table 4. The file *n31e038.hgt* contains the elevation data for the tile reaching from 31°N to 32°N and 38°E to 39°E inclusive.
Each hgt file stores a byte array using the following format:

- The elevation data is stored in a 2-dimensional array of size 3601 x 3601 using row-major order. By using row-major order, the data is arranged from west to east and then from north to south. This results in 12,967,201 available cells and corresponds to the 1 arc second horizontal resolution in a 1 degree by 1 degree tile.

- Each cell in the 2-dimensional array stores a signed 16-bit integer (2 bytes) in big-endian order. The range of -32,767 to 32,767 provided by this type can store any valid elevation on earth with meter used as the unit.

- The outermost rows and columns of each tile overlap with the corresponding rows and columns of adjacent tiles.

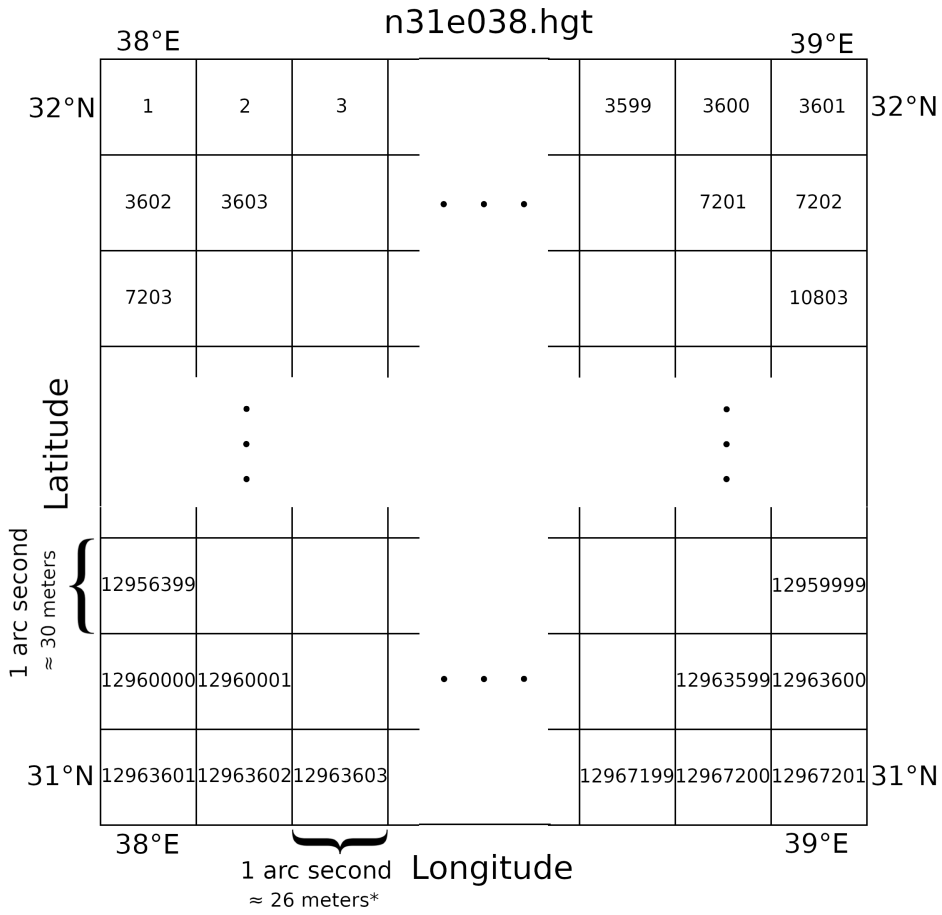

Figure 6: NASADEM n31e038.hgt file format. The numbers in the cells correspond to the ordering of the data in the file. *The east-west meters resolution decreases with latitude.

This results in the data of each tile, given by the byte array, having a size of 25.93MB (12,967,201 cells · 2 bytes per cell). In Figure 6, this internal format is visualized of the file n31e038.hgt. The numbers in the cells correspond to the cells data order inside the hgt file going from west to east and then from north to south.

3.3 Inverse distance weighting

Shepard [6] presents "A two-dimensional interpolation for irregularly-spaced data". The author defines a finite number N of data points D_i which are given as triplets (x_i, y_i, z_i). The locational coordinates of D_i are given by x_i, y_i, and z_i is the corresponding data value. The author introduces a function to interpolate the value of any point P in the plane. The function uses a weighted average of the existing data points where the weighting is the distance to those points. The Cartesian distance between P and D_i is $d[P, D_i]$. Then the interpolated value of P using the inverse distance weighting function is:

$$f(P) = \begin{cases} \frac{\sum_{i=1}^{N} d[P,D_i]^{-u} z_i}{\sum_{i=1}^{N} d[P,D_i]^{-u}} & \text{if } d[P, D_i] \neq 0 \text{ for all } D_i, \\ \mathrm{z}_i & \text{if } d[P, D_i] = 0 \text{ for some } D_i, \end{cases} \tag{1}$$

where u > 0. By increasing u, the influence of data points closest to P increases.

3.4 Simple moving average in unevenly spaced time series

In this section we introduce the concept of a time series and a moving average. Furthermore, we introduce the concept of a simple moving average applied to an unevenly spaced time series.

Time series

A *time series* denotes a series of data points indexed in time order. For a *time series* X with length *N(X)*, let $T(X) = t_1, ..., t_{N(X)}$ be the strictly-increasing sequence observation times and $V(X) = X_1, ..., X_{N(X)}$ the sequence of observation values. With this notation, a *time series* X consists of the data points $((t_n, X_n) : 1 \leq n \leq N(X))$.

Moving average

A *moving average*[5] creates a new *time series* that contains averages based on a *time series*. A *moving average* iterates over each $((t_n, X_n) : 1 \leq n \leq N(X))$ data point and does some calculations based on the current and previous data points. The results of the calculations go into the new *time series*. In the simplest form of a *moving average*, the *simple moving average* creates a new *time series* containing the *mean* of each data point and the previous k data points. It is assumed that the observation times *T(X)* are evenly spaced.

Simple moving average for unevenly spaced time series

Eckner [3] presents "Algorithms for Unevenly Spaced Time Series: Moving Averages and Other Rolling Operators". The author uses the notations of *time series* as above. The author specifically looks at *unevenly spaced time series*, meaning the observation times *T(X)* are not evenly spaced. In the context of unevenly spaced data points, it is not appropriate to refer to the last k data points when applying a *moving average*. Unlike evenly spaced data points, the actual used observation times can vary vastly in unevenly spaced data points. Instead, Eckner introduces the parameter τ that denotes the *length* of a *moving average window*. Only data points that fall within the *moving average window* are considered when applying a *moving average* for a data point.

Furthermore, Eckner introduces the function $X[t]_{linear}$. For a point $t \in \mathbb{R}$, $X[t]_{linear}$ denotes the linearly interpolated value of X at time t. This function is used to *sample* values at the edges of a *moving average window*.

Will all of the above, Eckner presents a simple moving average (SMA) for *unevenly spaced time series*. Eckner defines the function to apply a SMA with linear interpolation at the edges to a single data point as follows: Given an *unevenly spaced time series* X with a *moving average window* of length $\tau > 0$ and an observation time $t \in T(X)$, the function

$$\mathrm{SMA}_{\mathrm{linear}}(X, \tau)_t = \frac{1}{\tau} \int_0^\tau X[t - s]_{\mathrm{linear}} \, \mathrm{ds}. \tag{2}$$

As presented, the *moving average window* is of form $(t - \tau, t]$ for a $t \in T(X)$. According to Eckner, this can expanded to a *two-sided rolling time window* with form $(t - \tau, t + \mu]$, where $\tau > 0$ and $\mu \geq 0$. τ denotes the width of the window before

[5] `https://en.wikipedia.org/wiki/Moving_average`

and μ the width of the window after the current time $t \in T(X)$. This yields the *two-sided rolling time window* $\mathrm{SMA}_{\mathrm{linear}}$ function:

$$\mathrm{SMA}_{\mathrm{linear}}(X, \tau, \mu)_t = \frac{1}{\tau + \mu} \int_0^{\tau+\mu} X[t + \mu - s]_{\mathrm{linear}} \, \mathrm{ds}. \tag{3}$$

Both of the presented functions yield the result for a single observation time $t \in T(X)$. To get the complete simple moving average of an *unevenly spaced time series*, the functions must be applied to all observation times $t \in T(X)$.

3.5 Simple directed graph

In this section we introduce the basic terminology of a simple directed graph. Additionally, we introduce the concepts of degrees of graph vertices, paths, and cycles in a graph.

A simple directed graph $G = (V, E)$ consists of a set of vertices V and a set of directed edges $E \subseteq \{(u, v) | (u, v) \in V \times V \text{ and } u \neq v\}$. For example, for the vertices $u, v \in V$, the edge $(u, v) \in E$ represents the directed edge from u to v. The edge $(v, u) \in E$ is the corresponding *reversed* edge, going from v to u. In Figure 7, a visualized representation of a simple directed graph is shown.

The *in-degree* and *out-degree* of a vertex denote how many incoming and outgoing the vertex has, respectively. It holds $\text{in-degree}(u) = |\{(v, u) : (v, u) \in E\}|$ and $\text{out-degree}(u) = |\{(u, v) : (u, v) \in E\}|$.

A *path* in a directed graph $G = (V, E)$ is a sequence $u_1, u_2, u_3, ..., u_n \in V$, where $(u_1, u_2), (u_2, u_3), ..., (u_{n-1}, u_n) \in E$. A *cycle* is a *path* in which the first and last vertices are identical.

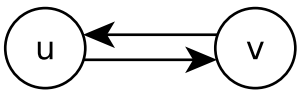

Figure 7: Simple directed graph with two vertices and two edges. The graph $G = (\{u, v\}, \{(u, v), (v, u)\})$ with two vertices and two edges. For both of the edges the *reversed* edge is also present.

4 Approach

In this section, we present the steps performed by our tools *osmelevation* and *correctosmelevation*. Both of our tools read and write OpenStreetMap data and share implementation details in this regard. Also, both of our tools are implemented in *C++* using the *C++ 13 Standard*. We use the *GNU Compiler Collection (GCC)* and *CMake* to build our tools. Before presenting our tools separately in detail, we explain how we read and write OpenStreetMap data.

4.1 Reading and writing OpenStreetMap data

Our tools read and write valid OpenStreetMap data. The *osmium* library provides readers and writers for the common OpenStreetMap data formats [9].

4.1.1 Reading OpenStreetMap data

The *osmium* library provides a *Handler* class to process OpenStreetMap data while reading. This class defines functions for each OpenStreetMap element *node*, *way*, and *relation*. While reading, the OpenStreetMap objects are fed one by one into their corresponding function making the data of the individual objects available.

By deriving from the *Handler* class, we can use our own implementations of the provided functions. This means we either have to process the OpenStreetMap objects one by one while reading or store relevant data and process the data later.
We implement several of our own handlers that derive from the *Handler* class provided by *osmium*.

Additionally, the *osmium* library provides the option to only read specified OpenStreetMap elements. For example, in our first tool we only need data from nodes. By omitting to read ways and relations in our first tool, we can save a not insignificant amount of time.

The *osmium* library also provides a class *RelationsManager*. This class gives the possibility to conveniently work with OpenStreetMap relations. When deriving from

this class, we can specify relations that we are interested in. *RelationsManager* collects these relations and performs a second pass over all OpenStreetMap ways. As soon as all referenced ways of a relation are found, the relation can be processed.

4.1.2 Writing OpenStreetMap data

We use the *Writer* class provided by the *osmium* library to write our output OpenStreetMap data. This class implements a *write* function that accepts OpenStreetMap objects one by one and writes them to the specified output OpenStreetMap file on disk.

Both our tools only add data to the input OpenStreetMap data or update existing data. Therefore, to write our results, we read the data from the input OpenStreetMap file and write the output file at the same time. We implement our own handler deriving from the *Handler* class that creates a copy of each object. When needed, an update or an addition to the copy is performed. The copy then gets moved into the *write* function.

4.2 Annotating OpenStreetMap data with elevation data

We start by presenting our first tool *osmelevation* that enriches OpenStreetMap data with elevation data.
The input as well as the output of our tool are valid OpenStreetMap data. We designed our tool specifically for the worst-case input, that is the OpenStreetMap data containing the whole planet. Therefore, we focus on the worst-case input in this presentation.

As we have learned in 3.1.2, the OpenStreetMap data is built from the elements nodes, ways, and relations. Since we only modify nodes by adding a tag *ele=** to each, we can completely ignore ways and relations in our first tool. We can also ignore any tags attached to the nodes because the elevation is only dependent on the location.

4.2.1 Problem Definition

In 1.1.2 we introduced the problem of the size of the OpenStreetMap data and the elevation dataset, *NASADEM_HGT*. On one side, there are 7.5 billion nodes in the OpenStreetMap. This leads to two problems:

- There are 7.5 billion random accesses to the elevation data. For each node's location, we need to locate the tile that contains the location, and then we need to locate the correct cell inside the tile that contains the elevation data.

- Over the course of the tool's calculations, the elevation data of nodes that have already been collected have to be stored somewhere. This means there needs to be storage for 7.5 billion 16-bit integers that can be matched to the node they belong to.

On the other side, there are 371GB of elevation data from the *NASADEM.HGT* elevation dataset. We have shown in Table 2 that the data is packed into 14520 single compressed files. It is clear that the elevation data can not be loaded all at once.

Both the OpenStreetMap and elevation data are a challenge of their own to handle concerning the sizes. This makes it a challenge to use both data simultaneously while not sacrificing too much performance.

4.2.2 Overview

For our first tool, we are only interested in the nodes of OpenStreetMap. This results in three favorable properties: First, to obtain all the data of the nodes, it is sufficient to parse the provided OpenStreetMap data only once. Conversely, to obtain all data of ways and relations, at least two passes over the OpenStreetMap data are necessary. Secondly, a node contains all the information that is associated with it. When reading a node from the OpenStreetMap data, the nodes can be processed one by one without the need to store some data that is needed later. Lastly, since the data in the OpenStreetMap files are sorted by all nodes first, all ways second, and lastly all relations, we do not need to parse the complete OpenStreetMap file to obtain the data of the nodes.

This makes it possible to read the OpenStreetMap data, more precisely only the nodes, multiple times without a big performance penalty.
We make use of this by partitioning the geographic extent of the input OpenStreetMap data. The OpenStreetMap nodes are read by the *OsmNodesHandler*. This handler specifically skips nodes that are not in the by *GeoPartiton* provided geographic partition. With this, only nodes which locations are inside the specified geographic partition are processed in one pass over all OpenStreetMap nodes. For each node, the elevation data gets extracted from the *NASADEM_HGT* dataset by our *GeoElevation* component. The only data we need for later is the node id and the elevation of the node's location. Both get stored in the *ElevationIndex* that is accessible by node id. The just described process gets repeated until all partitions were worked off. Note

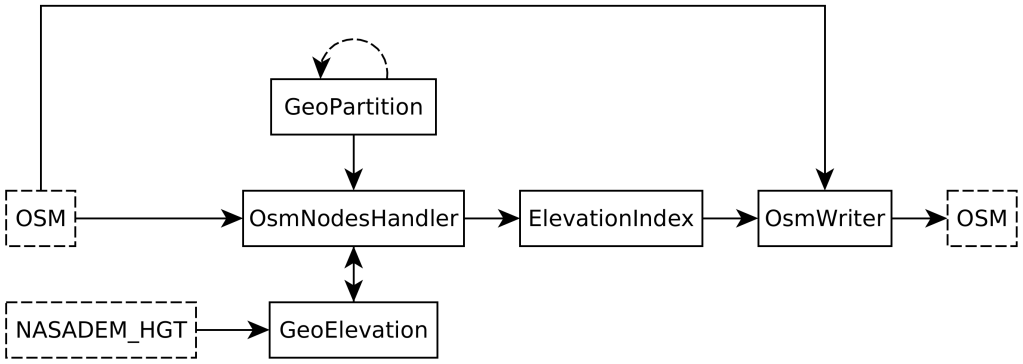

Figure 8: *osmelevation* dataflow. Solid boxes represent the main components, dashed boxes are input/output files. Solid edges represent the dataflow, dashed edges signal repeating components. OpenStreetMap data is read from the *OSM* file by the *OsmNodesHandler* in partitions. *GeoPartition* dictates which partition is read. The *OsmNodesHandler* retrieves for each node in the partition the elevation data from *GeoElevation*. *GeoElevation* handles getting the requested elevation data from the *NASADEM_HGT* dataset. Also, *OsmNodesHandler* stores the elevation data for each node in the *ElevationIndex* tied to their node ids. This just described procedure is recurring, on basis of *GeoPartition* until all geo partitions were handled. When done, the *OsmWriter* uses the input *OSM* file and the *ElevationIndex* to write a new output *OSM* file. The output file contains all the original data of the input *OSM* file and the elevation data for each node that was collected before.

again that one such pass of a partition equals one pass over all OpenStreetMap nodes. When done, the *ElevationIndex* stores the elevation of all nodes tied to the node ids. In the next step, the output *OSM* file gets written, containing the elevation data. The *OsmWriter* rewrites the complete input *OSM* file. This includes all nodes, ways, and relations as our tool does not lose any data. When writing the nodes, an elevation tag gets added to each node using the data from the *ElevationIndex*. The complete dataflow as just described in visualized in Figure 8.

4.2.3 Reading and processing OpenStreetMap nodes

The *OsmNodesHandler* is the main component of our tool. We use it to read OpenStreetMap nodes. One *pass* of *OsmNodesHandler* reads all nodes one by one, several passes may be performed. It represents the connection between the other components *GeoPartition*, *ElevationIndex*, and *GeoElevation*. The *OsmNodesHandler*

is directly derived from the *osmium* library *Handler* class. Our *OsmNodesHandler* only implements the *node* function since we are not interested in ways and relations in our first tool. Nodes are fed one by one into the *OsmNodesHandler* while reading and get processed immediately. In the following listing, we explain the connections to the other components:

- *GeoPartition* provides a geographic partition to the *OsmNodesHandler*. In simple terms, the geographic partition decides whether a node gets processed in the current pass or not. If a node is not in the geographic partition, it simply gets skipped. The overall number of geographic partitions is equivalent to the number of passes *OsmNodesHandler* performs.

- *GeoElevation* provides the interface to the *NASADEM_HGT* elevation dataset. If a node gets processed in the current pass, the elevation of the node's location gets requested from *GeoElevation*.

- *ElevationIndex* stores the data that was processed and is needed later. If a node was processed in the current pass, the node's id and elevation get stored in the *ElevationIndex*.

Listing 1: *node* implementation of *OsmNodesHandler*

```
void OsmNodesHandler::node(const osmium::Node& node) {
  const double lon = node.location().lon();
  const double lat = node.location().lat();

  // Check if the node's location is inside the geographic partition.
  if (lon >= std::get<0>(_geoPartition) &&
      lon <= std::get<2>(_geoPartition) &&
      lat >= std::get<1>(_geoPartition) &&
      lat <= std::get<3>(_geoPartition)) {

    // Get the elevation at the node's location.
    const int16_t elevation =
      _geoElevation.getInterpolatedElevation(Coordinate(lon, lat));

    // Store the node's id and elevation in the elevation index.
    _elevationIndex.setElevation(node.id(), elevation);
  }
}
```

In Listing 1, the implementation of the *node* function is shown.

4.2.4 Geographic partitions

A main problem we encountered in the development of this tool is the size of the *NASADEM_HGT* elevation dataset. The data is provided by 14520 single zip archives. Each archive contains the elevation data for a 1 degree longitude by 1 degree latitude tile, as we have learned in Section 3.2. For performance, it is essential to have the uncompressed data in main-memory when accessing. With an uncompressed size of 371GB of this dataset, it is infeasible to have all this data in main-memory at once. Therefore, we needed to limit the number of elevation data tiles in main-memory.

Naive approach

A naive approach we tried was to manage the number of data tiles in main-memory with a first-in, first-out[1] (FIFO) queue. New data tiles can be loaded into main-memory as long as a specified length of the FIFO queue is not exceeded. Otherwise, the first added data tile gets removed from main-memory to make room for a new one.
This approach failed miserably. The problem is that the OpenStreetMap nodes are not geographically ordered. In the worst-case scenario, for every new location we need the elevation of, the corresponding data tile has to be unpacked from the archive and loaded into main-memory again. With 7,5 billion nodes currently in OpenStreetMap, each compressed data tile would be unpacked, and loaded thousands of times. It is obvious that this approach is incredibly inefficient and would result in a runtime of months for just the unpacking of the archives.

Partitioning nodes geographically

To solve the problem of the not existing geographic ordering of the nodes, we introduce geographic partitions. A geographic partition simply is a rectangular boundary on a map defined by its bottom-left (south-west) and upper-right (north-east) coordinates. These coordinates are stored in a tuple by (*minimum longitude*, *minimum latitude*, *maximum longitude*, *maximum latitude*).

Our *OsmNodesHandler* performs multiple passes over all OpenStreetMap nodes. Each pass, a different geographic partition is provided by *GeoPartition*. In each pass, we only process the nodes that are located inside the specified geographic partition. With this approach, we can determine in advance how many data tiles will be loaded

[1] `https://en.wikipedia.org/wiki/FIFO_(computing_and_electronics)`

into main-memory at maximum by adjusting the partition size. Also, and more significantly, the data tiles will be unpacked and loaded only once per geographic partition.

For example, the partition (*10*, *40*, *20*, *50*) reaches from 10°N to 20°N and from 40°W to 50°W. That means only the nodes that are located between these coordinates inclusively the edges will be processed. Since the each elevation data tile covers a 1° by 1° region, there will be $10 \cdot 10 = 100$ data tiles in main-memory. Additionally, the elevation data tiles that are vertically and horizontally adjacent to the geographic partition are also needed as we will see later. These account to $4 \cdot 10 = 40$ elevation data tiles. Overall, for the geographic partition (*10*, *40*, *20*, *50*) $100 + 40 = 140$ elevation data tiles will be unpacked and loaded into main-memory.

4.2.5 Retrieving elevation data

For a location given as longitude and latitude, there are several steps performed to retrieve the elevation from the *NASADEM_HGT* dataset.

Resolving the elevation data tile filename

Each elevation data tile is provided compressed in a zip archive with a distinctive name. The first step we perform is to resolve the filename for the correct archive containing the elevation data tile.

The zip archive filenames are composed of the dataset name *NASADEM_HGT* and the coordinate of the southwest corner of the elevation data tile, as shown in Section 3.2.2. The coordinate is given as the unsigned decimal degrees with *n/s/e/w* specifiers to indicate the compass direction. For example, *NASADEM_HGT_n31e038.zip* is the zip archive filename containing the elevation data tile with the southwest corner 31°N 38°E.

In the OpenStreetMap coordinate representation, the *n/s/e/w* specifiers are omitted by using signed decimals to indicate the compass direction. For example, the OpenStreetMap location *lat="47.0288378" lon="7.5252089"* gets resolved to the *NASADEM_HGT_n47e007.zip* archive file. We perform three simple steps to get the correct zip archive filename for a coordinate:

1. Use the *floor()* function on the longitude and latitude of the coordinate. This yields the southwest corner of the corresponding elevation data tile.

2. Determine the *n/s/e/w* specifiers and convert to unsigned integers. The specifiers can be obtained by simply checking if the longitude/latitude are signed or unsigned. Signed longitude/latitude result in s/w, otherwise in n/e respectively.

3. Pad the unsigned integer coordinates with zeros if needed. As seen in the example *NASADEM_HGT_n31e038.zip*, the latitude component is presented by 2 digits and the longitude component by 3 digits. If a coordinate component does not use 2/3 digits respectively by itself, we have to pad with zeros.

Loading the elevation data tile into main-memory

Priorly, we have located the correct zip archive that contains the elevation data tile for a coordinate. In the next step, we unpack the compressed *.hgt* file from the zip archive. Since we do not need the unpacked data on disk, we unpack directly into the main-memory. We use the *libzip*[2] library to unpack into main-memory. We then store the raw unpacked bytes inside a *C++* uint8_t[] byte array.

Accessing data from an elevation data tile

In this step, we explain how elevation data can be extracted from an elevation data tile. In the previous step, we stored the data in an uint8_t[] byte array. In the following, we refer to this array as the *elevation data array*.

The characteristics of the elevation data array are consistent with our explanations from Section 3.2.3. The elevation data array stores a 2-dimensional array of size 3601 x 3601 using row-major order. Since we store the data inside an uint8_t[] byte array, the elevation data array has a length of $3601 \cdot 3601 \cdot 2 = 25,934,402$. A *cell* from the elevation data tile is represented by two consecutive bytes starting at an even index position in the elevation data array. Thus, each cell stores a signed 16-bit integer. Since the 16-bit integers are encoded using the big-endian byte order, we have to manually decode the data for each cell when needed.

We use the following terms to denote characteristics of the elevation data array:

- *samples*: The number of rows and columns of the 2-dimensional array stored inside the elevation data array. For the *NASADEM_HGT* dataset, this corresponds to 3601 *samples*.

[2] `https://libzip.org/`

- *cell size*: The width and height of one cell in *degree*. The *elevation data array* stores the data of one *elevation data tile*. In the *NASADEM_HGT* dataset, one tile has a width and height of one degree. Thus, the *cell size* corresponds to $\frac{1}{samples-1}$.

- *cell center offset*: The offset of the center of a cell to one of its four edges. Calculated by $\frac{cell\ size}{2}$.

To extract the elevation of a coordinate given by longitude and latitude, we first calculate the exact cell that stores the data. This cell corresponds to the geographical location the coordinate lies in, inside the *elevation data tile*. We can calculate *row* and *column* of the cell as follows:

$$row = \lceil(\text{longitude} - \lfloor\text{longitude}\rfloor - cell\ center\ offset) \cdot (samples - 1)\rceil \quad (4)$$

$$column = \lceil(\lfloor\text{latitude}\rfloor + 1 - \text{latitude} - cell\ center\ offset) \cdot (samples - 1)\rceil \quad (5)$$

With the *row* and *column*, we can calculate the start index of the cell in the *elevation data array* and the center of the cell in the *elevation data tile* as coordinate:

$$cell\ index = (row \cdot samples + column) \cdot 2 \quad (6)$$

$$cell\ center\ longitude = \lfloor\text{longitude}\rfloor + (column \cdot cell\ size) \quad (7)$$

$$cell\ center\ latitude = \lfloor\text{latitude}\rfloor + 1 - (row \cdot cell\ size) \quad (8)$$

Since the *elevation data array* stores single bytes and the cells are represented by two bytes, the factor of 2 is involved in calculating the *cell index*. We now know that the elevation data is stored at index *cell index* and *cell index + 1*.
The final elevation data represented by a signed 16-bit integer must be calculated from the big-endian byte order. In Listing 2, we provide an implementation to extract the elevation from a cell using *C++* language features.

Listing 2: Extract the elevation data from a cell inside the *elevation data array*

```
int16_t elevation = (int16_t)(elevation data array[cell index] << 8) +
                    elevation data array[cell index + 1];
```

Interpolating elevation data

As of now, we use the elevation of the cell in which a coordinate is located in regarding the *elevation data tile*. The elevation accuracy can be improved by also using the elevation data of the neighboring cells the coordinate is located in. To access data from a neighboring cell, we can simply change the row and/or column from the original cell by one. For example, given a cell in column and row 100, the neighboring cell to the right is in column 101 and row 100.

We use the *inverse distance weighting* interpolation to make use of the additional data. We use the same notation as presented in Section 3.3. We want to interpolate the elevation for a coordinate P. The data point D_1 represents the center of the cell the coordinate P lies in. The eight data points from the neighboring cell centers are represented by D_2 to D_9. In Figure 9, we provide a visualization of the nine cells and the coordinate P as explained. The data value z_i of each data point D_i corresponds to the elevation of the cell as stored in the *elevation data array*.

To obtain the center coordinates of the cells, we use Equation (7) and Equation (8). To calculate the distances $d[P, D_i]$ from the coordinate P to each data point, i.e. cell center, we use an approximation *haversine approximate* of the haversine formula[3]. The approximation omits the curvature of the earth in the calculation of the distance. We can use the approximation because the distances we need to calculate are always smaller than 100 meters.

For the exponent u used in the inverse distance weighting, we use the value $u = 2$ as Shepard [6] recommends. We now have all information to apply Equation (3.3) on the data to obtain the interpolated elevation of a coordinate.

4.2.6 Storing processed elevation data of nodes

Over the course of our tool's calculations, the elevation data for more and more nodes get collected until all geographic partitions were worked off. When done, the stored elevation data must be accessible by node id while writing the output *OSM* file.

We accomplish this by using a *sparse elevation index* or a *dense elevation index*. Which is used depends on the size of the input OpenStreetMap data, i.e. how many nodes are present.

[3] `https://en.wikipedia.org/wiki/Haversine_formula`

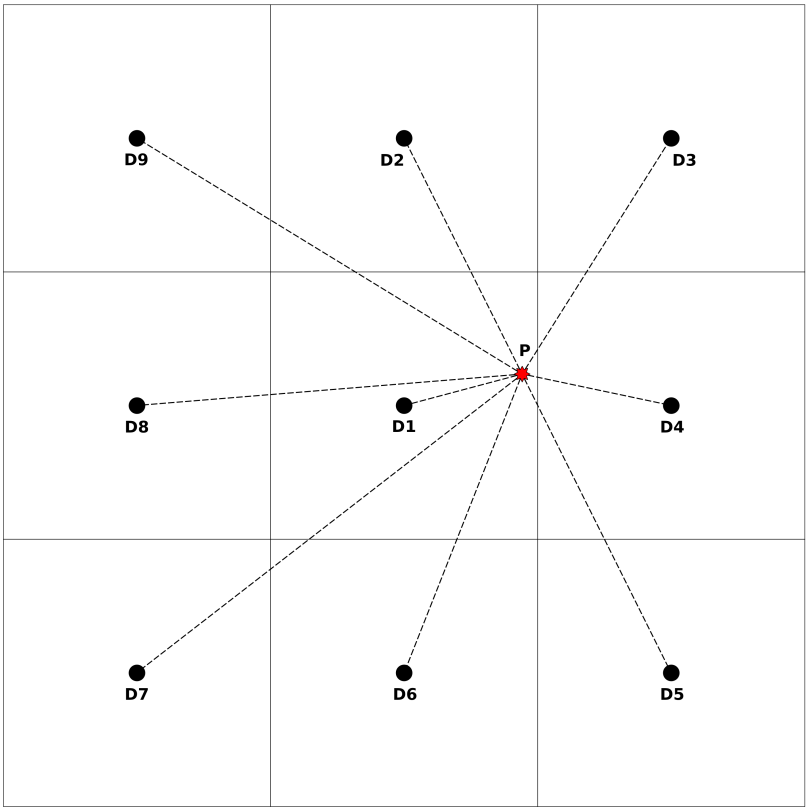

Figure 9: Interpolation with data from surrounding cells. Shown is a section of an *elevation data tile*. The black dots represent the center of each cell. The center of each cell also corresponds to its data point D_i. The red star is the coordinate P the elevation is requested for. The distance $d[P, D_i]$ from the coordinate to each data point is needed for the interpolation and is marked with the dashed lines.

Sparse elevation index

We use a sparse index type for smaller OpenStreetMap inputs that have less than one billion nodes. For each node we want to store the elevation of, we explicitly store the id of the node and its elevation. The node id needs to be stored as a 64-bit unsigned integer and the elevation as a 16-bit signed integer. We use an *IdElevation* `struct` to store both, as seen in Listing 3.

Listing 3: *IdElevation* `struct` to store the elevation of a node

```
struct IdElevation {
  uint64_t id;
  int16_t elevation;
};
```

Until all all geographic partition were worked off, *IdElevation* objects get appended to a `std::vector<IdElevation>` *sparseIndex*. When done, we sort *sparseIndex* by the *id* member of *IdElevation* objects, in ascending order. After the *sparseIndex* is sorted, we additionally remove duplicates. Duplicates can occur because the geographic partitions are overlapping at their borders.

With the sorted *sparseIndex*, we are ready to write the output *osm* file. To retrieve the elevation for a node, we perform simple binary search on the by id sorted *IdElevation* objects inside the *sparseIndex*.

Dense elevation index

For input OpenStreetMap data that contains more than one billion nodes, we use a dense index to store the elevation data accessible by node id. In the sparse index, we explicitly stored the node id for each entry since we need the id for later accessing it. We now directly store the elevation data in a `std::vector<int16_t>` *denseIndex*. The index of an entry directly corresponds to the node id the elevation data belongs to. This way, we can save the 8 bytes for each node needed by the id.

A problem we encountered was that the number of nodes and the maximum node id are not identical in OpenStreetMap. As of February 2022, there are 7,5 billion nodes in total, but the maximum node id present is around 9,5 billion. This results in the dense index not being as dense as we hoped. There are a lot of unused node ids in the OpenStreetMap.

To reduce the main-memory consumption of the *denseIndex*, we reduce the space needed to store a single elevation. As of now, a single elevation is stored as a 16-bit signed integer. A 16-bit signed integer provides a range of $-32{,}768$ to 32,767. But in reality, a range of -1000 to 10000 can cover any valid elevation above sea level on earth. This range of -1000 to 10000 can be stored using only 14 bits as follows:

- A signed 15-bit integer can store a range -16,374 to 16,374. This is still more than sufficient.

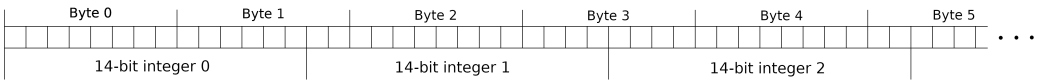

Figure 10: `std::vector<int8_t>` *denseIndex* used to store 14-bit integers. Shown is the layout of a `std::vector<int8_t>` array. The cells represent the individual bits of the array. An `int_8` occupies 8 bits (1 byte). The 14-bit integers are stored continuously on top of the bytes.

- Another bit can be omitted by removing the sign while storing. To still allow negative elevations, we simply add 1000 when storing the elevation and subtract 1000 when retrieving the elevation.

We implemented a custom 14-bit integer array by wrapping custom set and get functions around a `std::vector<int8_t>` *denseIndex*. Since we can not access individual bits, we need to work with bit shifts to modify the bytes on a bit level in the *denseIndex*. The layout of the *denseIndex* is visualized in Figure 10.

The 14-bit integers are stored continuously on top of the bytes. The *start bit* of a 14-bit integer is the bit position in the *denseIndex* of the most significant bit of the 14-bit integer. Analogously, the *start byte* is the byte that contains the *start bit*. The *start byte* is the most significant byte of a 14-bit integer.

We define that a 14-bit integer spans over three bytes in the *denseIndex*. The *start byte*, the byte at position *start byte* + 1, and the byte at position *start byte* + 2. With this definition, when looking at any three consecutive bytes in the *denseIndex*, there are exactly four possibilities how a 14-bit integer can be layed out in these three bytes.

We use *bitmasks* to set and extract a 14-bit integer from the three bytes it is contained in. For setting a 14-bit integer, we first want to reset the bits that belong to the 14-bit integer before writing the new bits. In Listing 4, the bitmasks for setting a 14-bit integer are shown. The `uint8_t setBits[4][3]` 2-dimensional array has four rows for the four possible layouts of the 14-integer inside the three bytes. In the columns, the individual bitmasks for the bytes are stored. The `0` entries represent the 14-bit integer, along a row. To reset the bits of a 14-bit integer, we perform *bitwise AND* with the three bytes of the 14-bit integer and the corresponding layout of a row of *setBits*.

To extract a 14-bit integer from the *denseIndex*, we use the `uint8_t getBits[4][3]` 2-dimensional array as shown in Listing 5. The approach is analogous. But instead of resetting the bits of the 14-bit integers, we want to temporarily hide the bits that do not belong to the 14-bit integer. This means the bits are `1` where they belong to the 14-bit integer, otherwise `0`.

Listing 4: Bitmasks to reset the bits a 16-bit integer

```
uint8_t setBits[4][3] = { {0b11111100, 0b00000000, 0b00001111},
                          {0b11110000, 0b00000000, 0b00111111},
                          {0b11000000, 0b00000000, 0b11111111},
                          {0b00000000, 0b00000011, 0b11111111} };
```

Listing 5: Bitmasks to hide the bits not belonging to a 16-bit integer

```
uint8_t getBits[4][3] = { {0b00000011, 0b11111111, 0b11110000},
                          {0b00001111, 0b11111111, 0b11000000},
                          {0b00111111, 0b11111111, 0b00000000},
                          {0b11111111, 0b11111100, 0b00000000} };
```

In Listing 6 and Listing 7, we provide the full implementations of how to set and get an elevation in the 14-bit integer *denseIndex*.

Listing 6: Implementation to store an elevation in the 14-bit integer *dense index*

```
void ElevationIndexDense::setElevation(uint64_t id, int16_t elevation) {

  elevation += 1000;  // Allow storage of negative elevations.

  // The position of the most significant bit
  // of the 14-bit integer in the dense array.
  uint64_t startBitPosArray = id * 14;

  // The position of the byte containing the most
  // significant bit of the 14-bit integer in the dense array.
  uint64_t startBytePos = startBitPosArray / 8;

  // The position of the least significant bit of the 14-bit
  // integer (LSB) on basis of the least significant bit of the
  // byte at position startBytePos + 2.
  // This can be at most at position 10.
  uint8_t posLSB = 10 - (startBitPosArray % 8);

  // The bitmask to clear all bits not used by the
  // 14-bit integer.
  uint8_t bitmask = (posLSB - 4) / 2;

  // The data the least significant byte of the 14-bit integer holds,
  // respectively to the position of the LSB.
  uint8_t byte2 = denseIndex[startBytePos + 2] & setBits[bitmask][2];
  byte2 |= (elevation << posLSB);

  // The data the second least significant byte,
  // i.e. second least significant byte of the 14-bit integer holds,
  // respectively to the position of the LSB.
  uint8_t byte1 = denseIndex[startBytePos + 1] & setBits[bitmask][1];
  if (posLSB <= 8) {
    byte1 |= (elevation >> (8 - posLSB));
  } else {
    byte1 |= (elevation << (posLSB - 8));
  }

  // The data the most significant byte of the 14-bit integer
  // holds, respectively to the position of the LSB.
  uint8_t byte0 = denseIndex[startBytePos] & setBits[bitmask][0];
  byte0 |= (elevation >> (16 - posLSB));

  // Update the data of the least,
  // second least, and most significant byte.
  denseIndex[startBytePos + 2] = byte2;
  denseIndex[startBytePos + 1] = byte1;
  denseIndex[startBytePos] = byte0;
}
```

Listing 7: Implementation to retrieve an elevation from the 14-bit integer *dense index*

```
int16_t ElevationIndexDense::getElevation(const uint64_t nodeId) {

  // The position of the most significant bit
  // of the 14-bit integer in the dense array.
  uint64_t startBitPosArray = (nodeId) * 14;

  // The position of the byte containing the most
  // significant bit of the 14-bit integer in the dense array.
  // Hence, the 14-bit integer uses the startByte,
  // startByte + 1, and startByte + 2.
  uint64_t startBytePos = startBitPosArray / 8;

  // The position of the least significant bit of the 14-bit
  // integer (LSB) on basis of the least significant bit of the
  // byte at position startBytePos + 2.
  // This can be at most at position 10.
  uint8_t posLSB = 10 - (startBitPosArray % 8);

  // The bitmask to clear all bits not used by the
  // 14-bit integer.
  uint8_t bitmask = (posLSB - 4) / 2;

  // The data stored in the least, second least,
  // and most significant byte of the 14-bit integer.
  uint8_t byte2 = denseIndex[startBytePos + 2];
  uint8_t byte1 = denseIndex[startBytePos + 1];
  uint8_t byte0 = denseIndex[startBytePos];

  // Extract the elevation from the least, second least,
  // and most significant byte.
  // The bits that do not belong to the 14-bit integer get
  // hidden by the corresponding bitmask from getBits.
  int16_t elevation =
    (((byte0 & getBits[bitmask][0])) << (16 - posLSB)) +
    ((byte2 & getBits[bitmask][2]) >> (posLSB));
  if (posLSB <= 8) {
    elevation += ((byte1 & getBits[bitmask][1]) << (8 - posLSB));
  } else {
    elevation += ((byte1 & getBits[bitmask][1]) >> (posLSB - 8));
  }

  // Substract 1000 to allow negative elevations.
  return elevation - 1000;
}
```

4.2.7 Runtime

We do not provide a detailed asymptotic runtime analysis of our tool. The runtime is dependent on the following two aspects:

- The number of nodes in the input OpenStreetMap file. The interpolation used for each node represents the main processing step of our tool. The complete processing runtime is linear in the number of nodes.

- The amount of main-memory that is available. Our tool *osmelevation* highly benefits from more main-memory. With more main-memory, the size of the geographic partitions can be increased. It follows that less time is spent parsing the OpenStreetMap nodes.

Also, the writing the output *OSM* file can take some time depending on the size of the input *OSM* file. In Table 5, three example runtimes of inputs with different sizes are shown.

	germany	planet
runtime tool	2 minutes	102 minutes
runtime writing output OSM	4 minutes	61 minutes
maximum used main-memory	10GB	94GB

Table 5: Runtimes of different input OpenStreetMap sizes of our tool *osmelevation*. Run on machine with *AMD Ryzen 7 3700X 8-Core/16-Threads and 128GB Ram.* Data was read from and written to *2TB NVME Samsung 970 Evo+.*

It is to be noted that the runtime of our tool also highly depends on the read/write performance of the *osmium* library. Reading and writing in *osmium* is multithreaded and therefore benefits from more CPU cores. On the same machine we ran the tests in Table 5, *osmium* can parse all OpenStreetMap nodes in a planet *OSM* file in just three minutes.

4.3 Correcting elevation data in OpenStreetMap

Our second tool *correctosmelevation* performs corrections on elevation data *in* OpenStreetMap. We do not use external elevation data for our second tool. As basis of the corrections, we look at OpenStreetMap linear route map features like roads or rivers. In fact, this includes all OpenStreetMap routes that can be traveled by car, train, bike, foot, or ship.

As our first tool, the input and output of *correctosmelevation* are valid OpenStreetMap data. Also, our second tool is specifically designed to handle the worst-case input, which is OpenStreetMap data containing the whole planet.

Since we look at route map features in OpenStreetMap, we need to work with all three OpenStreetMap elements nodes, ways, and relations. This induces new challenges as there is a lot more processing and storing of the data involved.

Before we proceed to explain the steps taken by our second tool in detail, we provide a short insight into OpenStreetMap routes.

4.3.1 Routes

A route defines an exact path between two points that can be traveled by foot, bike, vehicle, train, or ship. In OpenStreetMap, routes are stored in ways and relations. In the following, we refer to routes in relations as *route relations* and routes in ways as *route ways*

Routes in OpenStreetMap ways

An OpenStreetMap way consists of an ordered list of node ids and one or more tags to construct a map feature. The nodes to which the node ids reference represent the course of the map feature. To denote if a way, more precisely the course of the way, represents a *route way*, the following tags are used:

- *highway=** is used for anything that can be traveled by foot, bike, or vehicle[4]. The value *** of the tag denotes the specific use of the route, for example a *footway* or *cycleway*. Additionally, the tag *oneway=yes* can be used in *route ways* with tag *highway=**. This tag indicates that the route can only be traveled in the direction of the ordered node ids list of *route way*.

[4] `https://wiki.openstreetmap.org/wiki/Key:highway`

- *railway=** is used to tag any routes that are related to train routes[5].

- *waterway=** is used for anything that has a flow of water from one place to another, for example rivers[6].

Every *route way* in OpenStreetMap has one of these tags. In Listing 8, we can see an example *route way*, in this case a *highway=track*. The *nd ref=** entries represent the ordered list of node ids.

Listing 8: A route with tag *highway=track* in an OpenStreetMap way

```
<way id="52707298">
  <nd ref="664607210"/>
  <nd ref="668979932"/>
  <nd ref="668979920"/>
  <nd ref="668979921"/>
  <nd ref="668979922"/>
  <nd ref="668979923"/>
  <nd ref="668979918"/>
  <nd ref="668979919"/>
  <nd ref="668979912"/>
  <nd ref="668979913"/>
  <nd ref="668979914"/>
  <nd ref="470498978"/>
  <tag k="highway" v="track"/>
  <tag k="sac_scale" v="hiking"/>
  <tag k="tracktype" v="grade4"/>
  <tag k="width" v="2.5"/>
</way>
```

Routes in OpenStreetMap relations

Relations consist of *members* and *tags* to represent a more complex map feature. *Members* can be any other OpenStreetMap objects, meaning nodes, ways, or other relations. Each member is referenced by the id of the specific object. Also, each member can have a *role* that the object fulfills in the relation.

Route Relations in OpenStreetMap that can be traveled on land are tagged with *type=route*. This includes anything that can be traveled by car, train, bike, or foot.

[5] https://wiki.openstreetmap.org/wiki/Key:railway
[6] https://wiki.openstreetmap.org/wiki/Key:waterway

The tag *type=waterway* is used for relations that have a flow of water from one place to another. In Listing 9, a route in a relation with tag *type=route* is shown.

Listing 9: A route with tag *type=route* in an OpenStreetMap relation

```
<relation id="33839">
  <member type="way" ref="27262946" role=""/>
  <member type="way" ref="658516548" role=""/>
  <member type="way" ref="186168932" role=""/>
  <member type="way" ref="145404642" role=""/>
  <member type="way" ref="69340201" role=""/>
  <member type="way" ref="69340195" role=""/>
  <member type="way" ref="37708541" role=""/>
  <member type="way" ref="37708461" role=""/>
  <member type="way" ref="27262958" role=""/>
  <member type="way" ref="4616941" role=""/>
  <member type="way" ref="830987458" role="forward"/>
  <member type="way" ref="830987460" role="forward"/>
  <member type="way" ref="830987459" role=""/>
  <tag k="FIXME" v="incomplete"/>
  <tag k="name" v="Schauinsland-Radweg"/>
  <tag k="network" v="lcn"/>
  <tag k="ref" v="Sch"/>
  <tag k="route" v="bicycle"/>
  <tag k="type" v="route"/>
```

The course of a *route relation* is determined by the course of its member ways. To obtain the complete course of a *route relation*, the member ways must be connected in the correct order. That is, for each member way there is a successor member way. The first node of the successor member way is identical to the last node of the previous member way. Except for the beginning and end of the route, where no predecessor/successor member way exists. The first/last node of the first/last member way represent the start and endpoint of the route relation, respectively.

One special case of how routes are stored can occur. If a route has separated forward and backward directions, the route can be stored by using three different approaches. For example, this can occur if a road has separated forward and backward lanes. The three approaches are:

1. The forward and backward direction of a route get merged into a single *route relation*.

2. The forward and backward direction of a route are stored separately in two different *route relations*.

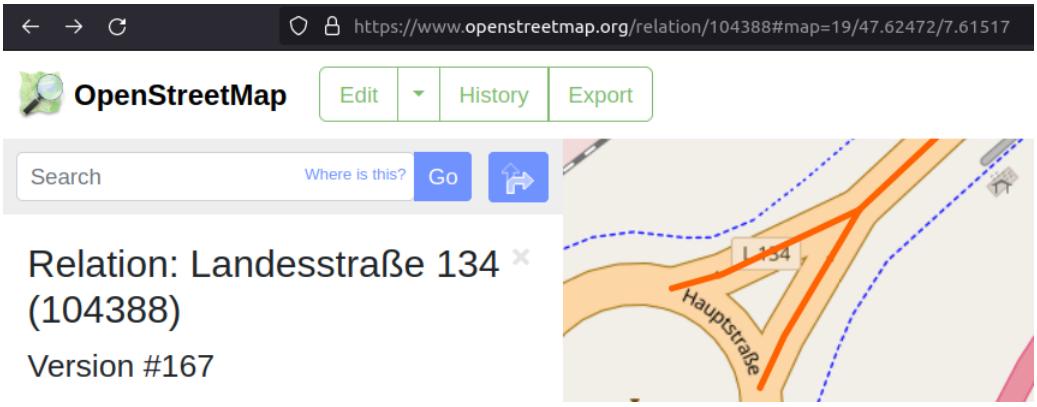

Figure 11: OpenStreetMap *route relation* containing forward and backward direction separately. The course of the *route* as stored in the relations is shown by the red lines. As seen, the *route* separates into both directions. Shown is a snippet of the end of the *route relation* with id *104388*. Provided by: OpenStreetMap | Map data © OpenStreetMap contributors

3. Both the first and second approach apply. There are in total three *route relations* that refer to the same route. One *route relation* containing the forward and backward direction and each one *route relation* containing the forward and backward direction separately.

Approach 1 and 3 violate our definition of a route that there is an *exact path between two points*. In both approaches, there can be two start and endpoints. In Figure 11, we can see an example where both directions of a route are stored in a single OpenStreetMap *route relation*. But in case of approach 1 and 3, the member ways that are used as a single direction are marked with *role="forward"* or *role="backward"*. Member ways that are used in both directions are marked with *role=""*.

It is to be noted that all *route relations* consist of *route ways*. But there are many *route ways* that are not part of any *route relation*.

4.3.2 Problem Definition

The focus of our second tool are OpenStreetMap routes. As we have just explained, routes are stored in *route ways* and *route relations*. The main challenge we faced were the *route relations* in OpenStreetMap relations. For the *route relations*, two main problems occurred:

- We need all the data that is connected to a *route relations*. Since OpenStreetMap relations are built from ways and nodes, these data need to be in

main-memory at the time of processing the *route relations*. For OpenStreetMap data of the whole planet, this accounts to several hundred gigabytes of data.

- To perform our corrections on *route relations*, we need to build the full path of the route from start to end. We call this a *route path*. A *route path* is a list of nodes that represents a path through the route. As we have explained in Section 4.3.1, it is not trivial to build a path from a *route relations* as their can be multiple. The individual member ways of the *route relations* have to be connected in the correct order.

To correct *route ways* is less complex. To obtain the full data of a *route way*, we only need the data of the nodes that are referenced by the *route way*. Also, a *route way* does not need to be *built*. The full path is already in the correct order, represented by the nodes in the ordered node ids list.

As in our first tool *osmelevation*, we need to store intermediate results from our corrections until we write to the output file when done. A simple *elevation index* as in our first tool seems be be sufficient at first. But this approach does not work. For example, two *route ways* can intersect each other at an intersection. This means there is a node that is referenced by both *route ways*. When performing a correction on both, there will be two correction results for the node in the intersection. Thus, an ordinary *elevation index* that stores a single elevation for a node is not sufficient since conflicts can occur.

4.3.3 Overview

Our second tool *correctosmelevation* performs corrections on the elevation data in OpenStreetMap based on routes. In OpenStreetMap, routes are present in *route relations* and *route ways*. We first process and correct all routes found in *route relations*. We do so by splitting the *route relations* into *routes ranges*. A *routes range* defines a fixed number of *route relations* that we process and correct at once. After we have corrected all *route relations*, we correct remaining routes found in *route ways*. The *route ways* also get corrected in *routes ranges*.

We perform the following corrections on *route relations*:

- We correct the elevation of tunnels and bridges. Nodes belonging to a tunnel or bridge should not have the elevation of the mountain or valley the tunnel/bridge crosses as it is the case in the uncorrected elevation data.

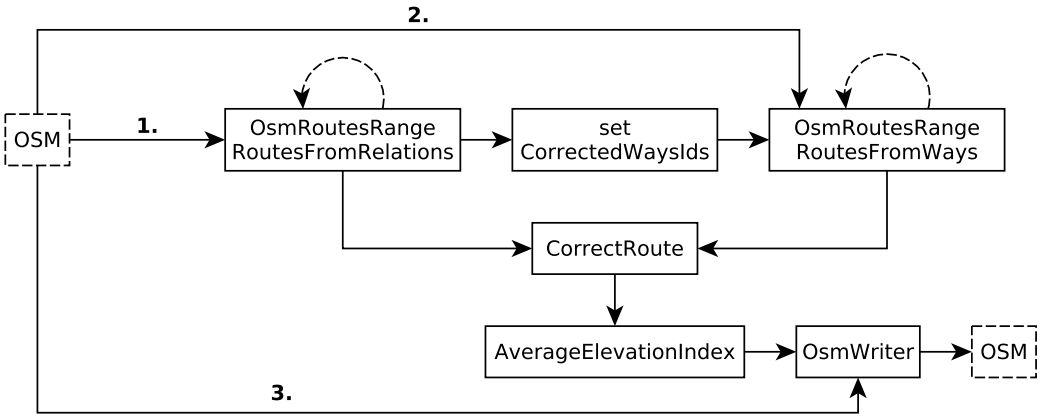

Figure 12: *correctosmelevation* dataflow. Solid boxes represent the main components, dashed boxes are input/output files. Solid edges represent the dataflow, dashed edges signal repeating components. Numbered edges indicate the order in which the input files are read. *Route relations* and *route ways* get processed and corrected in *routes ranges* by the *OsmRoutesRange*. In each *OsmRoutesRange* the input *OSM* gets parsed multiple times to collect all data that is needed. The *route ways* get corrected after all *route relations* were corrected. A set *CorrectedWaysIds* is used to exclude all ways that were already corrected in *route relations*. These already corrected ways do not get corrected again in *route ways*. Corrected elevation data for nodes get stored in the *AverageElevationIndex*. After all routes were corrected in *route relations* and *route ways*, the *OsmWriter* uses the corrected elevation data in the *AverageElevationIndex* to update the the old elevation data in the input *OSM* file.

- We apply a smoothing algorithm to all *route paths* that we have found in *route relations* to correct fluctuations in the elevation data.

- We correct rivers. Rivers should never flow uphill. We check that this property applies to rivers that we have found in *route relations*. If not, we correct the elevation.

In *route ways*, we apply a smoothing algorithm to all remaining *route paths* that were not already contained in a *route relations*.

In the last step, the output *OSM* file gets written, containing the corrected elevation data. The *OsmWriter* rewrites the complete input *OSM* file. This includes all

nodes, ways, and relations as our tool does not lose any data. When writing the nodes, the elevation data in existing tags *ele=** get updated using the data from the *AverageElevationIndex*.

4.3.4 Collecting OpenStreetMap routes in *routes ranges*

To cope with all that data, we work off the *route relations* and *route ways* in ranges. Our component *OsmRoutesRange* defines a *routes range* as follows: Given the original order of all relations/ways in the OpenStreetMap data, a *routes range* $[i, i+n)$, collects the i-th to the $(i+n-1)$-th *route relation*/*route way*. This happens in one pass over all relations/ways. With the parameter n, we can specify how many *route relations*/*route ways* we want to collect and subsequently process at once. The *OsmRoutesRange* is a recurring component. In each loop, the next *routes range* gets applied. For the previous example, the next *routes range* would be $[i+n, i+2n)$. This happens until all *route relations*/*route ways* were collected and processed.

4.3.5 Storing OpenStreetMap node data

For our corrections later, we will need access to the locations and elevations of nodes. We use a *NodeIndex* to store these data. On the basis, *NodeIndex* is a sparse index that maps from node ids to some data. It is built exactly as the sparse index of our first tool in 4.2.6, except that the we store *Node* objects in the `std::vector<Node>` *NodeIndex*. The entries of a *Node* object are shown in Listing 10.

Listing 10: *Node* `struct` to store the location and elevation of a node

```
struct Node {
  uint64_t id;
  double lon;
  double lat;
  int16_t elevation;
};
```

On the by id sorted `std::vector<Node>` *NodeIndex*, we can access the data of the nodes using binary search.

We build the *NodeIndex* using the *OsmNodesHandler* which derives from the *osmium* library *Handler* class. The *OsmNodesHandler* passes over all OpenStreetMap nodes and collects all nodes that we specified in a `set` *RequiredNodes*.

4.3.6 Storing corrected elevation data for nodes

As in our first tool, we need to store corrected elevation data temporarily until the data gets written to the output OpenStreetMap file. We use an *AverageElevationIndex* to do so. This index is a sparse index and the basic concept is similar as in the sparse index of our first tool 4.2.6. We use *IdElevationAverage* objects to store an average, which can be seen in Listing 11.

Listing 11: *IdElevationAverage* `struct` to store the average elevation of a node

```
struct IdElevationAverage {
  uint64_t id;
  int32_t elevationSum;
  uint16_t count;
  bool tunnelOrBridge;
};
```

These *IdElevationAverage* objects are stored in a vector `std::vector<IdElevationAverage>` *AverageElevationIndex*. We use the index as follows:

- Every time we want to add the elevation data for a node, we simply add a new *IdElevationAverage* object to the *AverageElevationIndex*. For the added object, we set *elevationSum* to the elevation of node and *count* to 1.

- Before we can access an average elevation, we need to process the *AverageElevationIndex*. The processing merges duplicates the following way, where duplicates have the same *id*: The *elevationSum* and *count* of duplicates get summed up. Then the merged *IdElevationAverage* object stores the new average.

- If *tunnelOrBridge* is set to true, we do not want to store an average of the elevation for the node. When merging such an *IdElevationAverage* object, this object remains unchanged, nothing is summed up.

Every time the *AverageElevationIndex* gets processed, we need to sort the index first to merge duplicates. Also, before we can retrieve elevation data for node ids, we need to sort the *AverageElevationIndex*.

To retrieve an elevation for a node, we use simple binary search to get the corresponding *IdElevationAverage* object of a node. The average elevation can then be calculated by $\frac{elevationSum}{count}$.

4.3.7 Processing *route relations* in a *routes range*

There are several steps performed in a single *routes range* for *route relations*. In this section, we explain what steps are taken and how they are connected. In the following, we brake down the component *RoutesFromRelations* which is derived from *OsmRoutesRange* as shown in the Overview:

- 1. The *OsmRelationsManager* performs a single pass over all OpenStreetMap relations. At the same time, *route relations* that fall within the specified *routes range* get collected.

- 2. The *OsmRelationsManager* performs a second pass, this time over all OpenStreetMap ways. At the same time, by collected *route relations* referenced ways get collected. As soon as all referenced ways of *route relations* are present, the *route relations* directly get processed. The result of the processing are single *route paths* consisting of node ids that we could find in the *route relations*. While processing, we also store ways that were used in the *route paths* in a `set` *CorrectedWaysIds* by id. Also, we store all used nodes in a `set` *RequiredNodesIds* by id.

- 3. We perform a pass over all OpenStreetMap nodes with the *OsmNodesHandler*. The *OsmNodesHandler* stores the nodes that were present in the `set` *RequiredNodesIds* in the *NodeIndex*. The *NodeIndex* maps from node ids to the locations and the elevations of the nodes.

- 4. We perform corrections on the found *route paths*. The *NodeIndex* stores all remaining data that is needed. At the same time, we add both uncorrected and corrected elevations of nodes to the *AverageElevationIndex*.

- 5. We process the *AverageElevationIndex*. Duplicates get merged and the index gets sorted.

In Figure 13, the broken down *RoutesFromRelations* component from Figure 12 is shown. We will present the step taken in *Process route* in the next section.

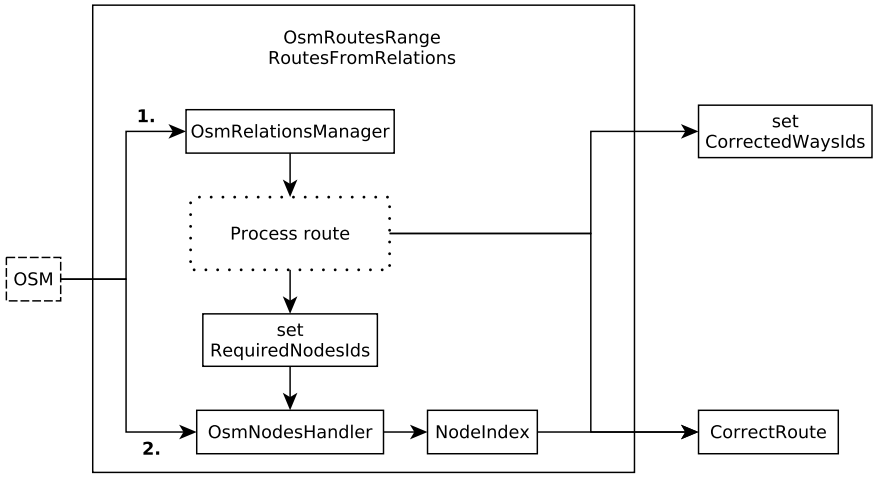

Figure 13: Processing of *route relations* in a single *routes range*. Solid boxes represent the main components, dashed boxes are input/output files, and dotted boxes indicate a placeholder component. Solid edges represent the dataflow. Numbered edges indicate the order in which the input files are read. The *OsmRelationsManager* performs two passes. In the first pass over all OpenStreetMap relations, *route relations* that lie within the specified *routes range* get collected. During the second pass over all OpenStreetMap ways, by *route relations* referenced ways from the first pass get collected. As soon as all referenced ways of a *route relation* were collected, the *route relation* gets processed. Nodes that were used in the processing get stored in the *RequiredNodesIds* `set` by id. Additionally, all ways that were used in the processing step get stored in the `set` *CorrectedWaysIds* by id. After all *route relations* were processed during the second pass of the *OsmRelationManager*, a *NodeIndex* is built. The *NodeIndex* gets filled by the *OsmNodesHandler*. The *OsmNodesHandler* parses all OpenStreetMap nodes. While parsing, nodes that are present in *RequiredNodesIds* get stored in the *NodeIndex*. With the processed *route relations* and the *NodeIndex*, the *route relations* can now get corrected by *CorrectRoute*.

4.3.8 Processing a *route relation*

In this section, we look at the steps taken to process a *route relation*. The data for the processing is provided by the *OsmRelationsManager*. This includes the data from the *route relation* itself and all referenced ways. Note that we do not need the referenced nodes of the ways yet.

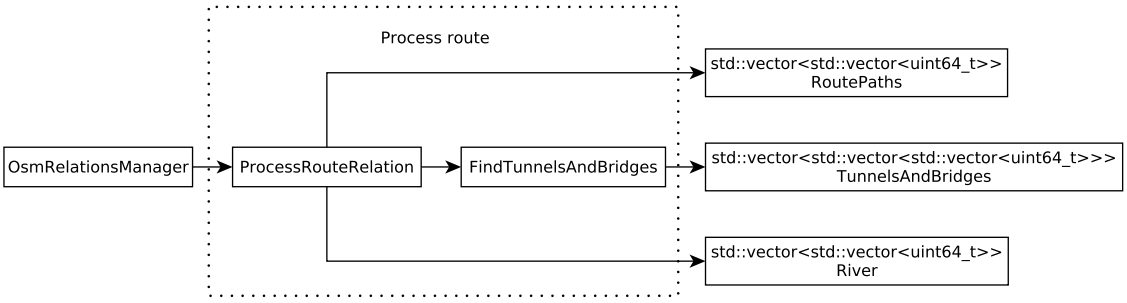

Figure 14: Output of processing of a single *route relation*. Solid boxes represent the main components, solid edges represent the dataflow. Shown is the dataflow of the processing of a single *route relation*. The *OsmRelationsManager* provides all data from the *route relation* and the referenced ways. *ProcessRouteRelation* extracts *route paths* from the *route relation*. The *route paths* get categorized into the output *RoutePaths*, *TunnelsAndBridges*, and *River*. Tunnels and bridges have been found before with *FindTunnelsAndBridges*.

The processing takes place in *ProcessRouteRelation* and *FindTunnelsAndBridges*. Before we look into the processing steps, we explain the output of the processing. In Figure 14, the dataflow and output of is shown. As we can see, there are three outputs:

- `std::vector<std::vector<uint64_t>>` *RoutePaths*: A list of individual *route paths* that could be found in the *route relation*. A *route path* consists of the node ids along the *route path* and is represented by a `std::vector<uint64_t>`.

- `std::vector<std::vector<std::vector<uint64_t>>>` *TunnelsAndBridges*: A list of tunnels and bridges found in the *route relation*. A single tunnel or bridge is represented by a `std::vector<std::vector<uint64_t>>` and separated into three segments. Each segment is represented by a `std::vector<uint64_t>` and consists of the node ids along the segment. The first segment is the OpenStreetMap way directly in front of the tunnel or bridge. The second segment consists of the way(s) that represent the tunnel or bridge itself. The third segment is the way directly after the tunnel or bridge.

- `std::vector<std::vector<uint64_t>>` *River*: An ordered list of waterways that belong to the same river. Each waterway is represented by a `std::vector<uint64_t>` which represents the node ids along the waterway, ordered in the direction of the flow. There can be multiple waterways since rivers can split up and merge

back into a single waterway. In this case the waterways are sorted by their time of discovery along the direction of flow.

Note that if the *route relation* represents a river or not, or does not contain any tunnels or bridges, the corresponding output(s) are empty. *RoutePaths* are only used for *route relations* that do not represent a river.

Building a graph of a *route relation*

The main processing step involves building a (simple) directed graph from the ways of the *route relation* for later traversal. The ways from the *route relation* are represented by edges in the directed graph. The endpoints of the ways of the *route relation* are represented by vertices in the directed graph. The endpoints are the first and last node ids of each way's ordered node ids list. This produces the following implementation of an edge in the directed graph, as seen in Listing 12:

Listing 12: `struct` to store a directed edge

```
struct Edge {
  uint64_t edgeId;
  uint64_t pointsTo;
  bool reversedEdgeExists;
};
```

Each edge has an unique id *edgeId*, which corresponds to the id of the way it represents. The direction of the edge is signalised by *pointsTo*. *pointsTo* stores the id of the vertex it points to. Additionally, an edge stores the Boolean *reversedEdgeExists*. This indicates if the *reversed edge*, as explained in Section 3.5, exists.

We store the directed graph in an adjacency list `std::unordered_map<uint64_t, std::vector<Edge>>` *DirectedGraph*. This representation maps vertices to their outgoing edges. To add ways as edges to the undirected graphs, we have to categorize the ways by their direction. We can accomplish this by looking at the *role* of the ways in the *route relation*:

- *role="forward"*: The direction of the way is identical to the ordering of the node id list of the way. The edge representing this way points to the last node id in the way's node id list. *reversedEdgeExists* is `false`.

- *role="backward"*: The direction of the way is reversed to the ordering of the node id list of the way. The edge representing this way points to the first node id in the node's node id list. *reversedEdgeExists* is `false`.

- *role=""*: Both directions, against and in the order of the way's node id list, are valid. We add two edges representing one direction each. Also, we set *reversedEdgeExists* to `true` for both edges.

We add edges as just explained for all *route relations* with tag *type=route*. For *route relations* representing a river, we always add the edges as in *role="forward"*. This is because the ways of a river always and only point in the direction of their node ids list ordering as this represents the flow direction of the river.

Traversing the graph of a *route relation*

For traversing the directed graph, we do not distinguish between *type=route* and *waterway=river route relations*. The goal for both is the same: We want to extract all *route paths* from the *route relation*. The *route paths* should be as long as possible. This translates to finding as long as possible paths in the directed graph. Note that a path in a graph and the *route paths* are not identical. A *route path* consists of the individual nodes of the ways connected in the correct order. A path in our directed graph represents the edges/ways in a correct order to connect them later. In a subsequent step, we need to convert the edges/ways to their individual nodes to gain the *route path* from the path in the directed graph.

To get paths that are as long as possible, we use a modified depth-first search[7] (DFS) algorithm. The modification we made is as follows: If there is more than one unvisited edge available for a vertex, we prefer an edge that is set to *reversedEdgeExists = true*.

As we have explained in Section 4.3.8, edges that are set to *reversedEdgeExists = true* represent ways that can be traveled in both direction. On the other hand, edges set to *reversedEdgeExists = false* represent ways in the *route relation* that can only be traveled in one direction. By preferring ways that can be traveled in both directions, we avoid running into very short *route paths*, as we found out.

We visualized this based on the earlier example of the route with multiple start and endpoints. The visualization is shown in Figure 15. Shown is a snippet of the end of an OpenStreetMap *route relation*. The directed graph of *route relation* is visualized as an overlay with colors. The vertices of the graph are shown by the black dots

[7] `https://en.wikipedia.org/wiki/Depth-first_search`

and the ids by the black numbers. There are 4 vertices with the ids *1*, *2*, *3*, and *4*. The edges of the graph are visualized by the orange lines with the orange numbers as their ids. The edges with ids *10*, *11*, *12*, and *13* are present. The yellow arrows along the edges indicate the direction of the edge.

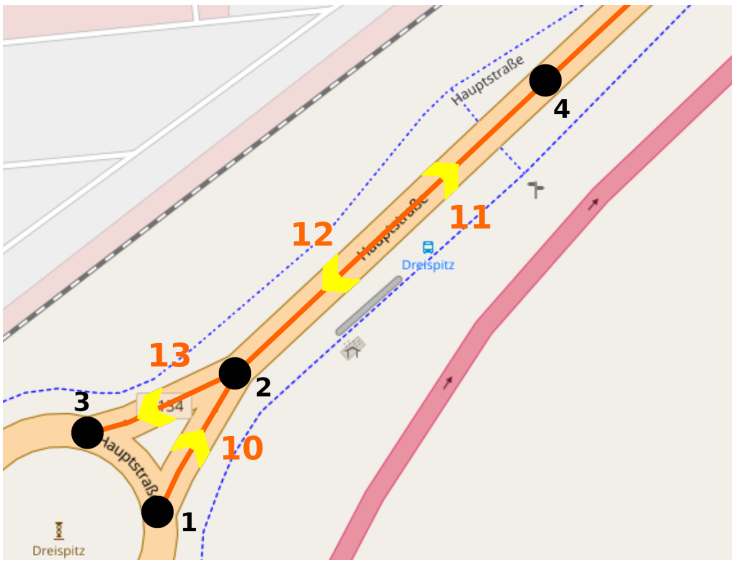

Figure 15: Directed graph vertices and edges visualized on an OpenStreetMap *route relation*. The vertices are visualized and numbered in black by id. The edges are visualized and numbered in orange by id. The direction of the edges is indicated by the yellow arrows. Shown is a snippet of the end of the *route relation* with id *104388*. Provided by: OpenStreetMap | Map data © OpenStreetMap contributors

Since the edge marked with *11* and *12* goes in both directions, there are two edges. Also, edge *11* and *12* are both marked with *reversedEdgeExists* = *true*, edge *10* and *13* are marked with *reversedEdgeExists* = *false*. The vertex with id *1* is a startpoint of the *route relation*. Thus, when starting a traversal of the directed graph, there are two choices for the next edge at vertex *2*. Edge *13* and edge *11* are available. Without our modification, it *can* happen that edge *13* gets chosen. This would result in the path [10, 13]. Clearly, the path [10, 11, ...] is the longer path which would be taken with our modification.

Generally, we found that if there is the choice between an edge marked with *reversedEdgeExists* = *true* and an edge marked with *reversedEdgeExists* = *false*, the latter often ends in a dead end regarding the *route relation*. Edge marked with *reversedEdgeExists* = *true* on the other hand, mostly are part of the main path from start to end.

In Algorithm 1, an implementation of the traversal of a single depth-search is described with the modification of preferring with *reversedEdgeExists = true* marked edges. Additionally, there is a set *usedEdges* that keeps track of priorly traversed edges. In the queue *visitAgain*, we insert vertices that have to be traversed again.

Additionally, we make another modification to Algorithm 1 that is not shown. If a traversal ends early because there was only an already visited edge available, we continue traversing until we have traversed a specified amount of visited edges. The already visited edges also get added to the path. We call this *padding*. We also *pad* paths from the front. Instead of having a queue *visitAgain* that holds vertices, we keep a queue of *unfinished paths*. When a branch with more than 1 available edge gets encountered in the traversal, we add a path consisting of the last n, where n is the specified *padding*, edges to the *unfinished paths*. When traversing the *unfinished path* later, we start from the *pointsTo* vertex of the last edge in the *unfinished path*. This is the vertex where the branch was encountered. This way we *pad* each path from the front and back. Obviously, nothing can be *padded* in front of a startpoint or after an endpoint of the graph.

With the *padding*, it can occur that paths yielded from traversals started from *unfinished paths* were fully seen before. In this case, every edge in the path was already contained in the `set` *visitedEdges*. If that happens, we discard the path since the path is already covered in another path.

Algorithm 1 Directed Graph: Traverse

Require: Directed graph *dg* is a map *vertexId* → *outgoing edges*,
usedEdges is a set containing already visited edges from prior traversals by id,
visitAgain is a list where vertices can be added by id

```
function TraverseGraph(startVertexId)
    visitedVertices ← {}                          ▷ Empty set
    path ← []                                     ▷ Empty list
    visitedVertices.insert(startVertexId)
    currentVertexId ← startVertex
    while currentVertexId ∈ dg do                 ▷ While the vertex has
                                                    outgoing edges ...
        nextEdge ← null                           ▷ Empty edge
        foundPreferredEdge ← false
        availableBranches ← 0
        for edge ∈ dg[currentVertexId] do         ▷ For each outgoing edge ...
            edgeUnused ← edge.edgeId ∉ usedEdges
            noCycle ← edge.pointsTo ∉ visitedVertices
            if edgeUnused and noCycle then
                availableBranches ← availableBranches + 1
                preferredEdge ← edge.reversedEdgeExists
                if not foundPreferredEdge and preferredEdge then
                    nextEdge ← edge
                    foundPreferredEdge ← true
                else
                    if not foundPreferredEdge then
                        nextEdge ← edge
                    end if
                end if
            end if
        end for
        if nextedge is not null then
            if availableBranches > 1 then
                visitAgain.append(currentVertexId)
            end if
            path.append(nextedge)
            currentVertexId ← nextedge.pointsTo
            visitedVertices.insert(nextEdge.pointsTo)
        end if
    end while
    return path
end function
```

Extracting all *route paths* of a *route relation*

We handle *route relations* with tag *waterway=river* and tag *type=route*. Depending on which one we need to process, we use the just explained graph of the route with some small differences. For a *type=route route relations* we perform the following steps to extract all *route paths*:

1. Build the graph of the route.

2. Collect the first and last nodes of the route. This is the first node of the first member way of the route, and the last node of the last member way of the route, respectively.

3. First, traverse the graph starting from the first node id and then from the last node id and collect the paths. The node ids correspond to the vertices in the graph.

4. Find all other startpoints in the graph. A startpoint in the graph is a vertex with an in-degree of zero. Additionally, vertices that have in-degree and out-degree of one and the one incoming/outgoing edge are the *reversed edges* of each other, it is also a startpoint.

5. Traverse the graph from all found startpoints and collect the paths.

6. Traverse the graph from all *unfinished paths* that have accumulated over the prior traversals. The *unfinished paths* get traversed in the order they were discovered. That means *unfinished paths* from step 3 get traversed first. Also, traverse all newly added *unfinished paths* that occurred during this step.

The following additional steps take place during steps 3, 5, and 6:

1. Directly after a path was found in a traversal, we search for tunnels and bridges in the path. Found tunnels and bridges get added to `std::vector<std::vector<std::vector<uint64_t>>>` *TunnelsAndBridges*.

2. After the search for tunnels and bridges on a path, the path gets converted to its *route path* representation. Until now, the path consisted of the edges in a correct order. With each edge along a path, the corresponding way with its node id list can get retrieved from the *OsmRelationsManager*. The individual *route paths* get added to `std::vector<std::vector<uint64_t>>` *RoutePaths*.

For river *route relations*, tagged with *waterway=river*, the same steps get performed as for a *type=route* route. The only difference appears in step 3. A river can only have one startpoint, which is the first node of the first member way. Hence we do not traverse from the last node. Also, we do not apply any *padding* for rivers in *unfinished paths*. For the later corrections of rivers, *padding* offers no advantage. The result of the processing of a river *route relation* is a `std::vector<std::vector<uint64_t>>` *River* containing the ordered *route paths* representing the waterways of the river.

Finding tunnels and bridges in a path

Given a path in the directed graph of a *route relation*, we iterate over the individual edges of the graph. The *edgeId* of an edge corresponds to the way id the edge represents. With the *OsmRelationsManager*, we then can access the data of the ways. To find tunnels or bridges along the path, we are interested in specific tags. If the following properties apply for at least three consecutive ways, we have found a tunnel or bridge:

- There are one or more consecutive ways that are either tagged with *bridge=** or *tunnel=**.

- There is at least one way before and after the bridge/tunnel way(s). Thus, the ways before and after are not tagged with *bridge=** or *tunnel=**. Additionally, the ways before and after the bridge/tunnel are not allowed to have the tags *embankment=** or *incline=**. These two tags signalise some kind of ramp that leads to the bridge/tunnel.

If consecutive ways fulfill the just explained properties, we directly convert the edges/ways to their *route path* representation. We store the *route path* splitted in three segments in a `std::vector<std::vector<uint64_t>>` as follows:

- At index 0 of the `std::vector`, the *route path* of the way before the tunnel/bridge is stored.

- At index 1, the *route path* containing of all ways that represent the tunnel/bridge is stored. These are the ways that were tagged with textitbridge=* or *tunnel=**.

- At index 2, the *route path* of the way after the tunnel/bridge is stored.

4.3.9 Processing *route ways* in routes ranges

After all *route relations* were processed and corrected, we correct all remaining *route ways*. That includes all *route ways* that were not already corrected in a *route relation*. We kept track of which *route ways* we do not need to correct in the `set` *CorrectedWaysIds*. We are only interested in *route ways* that have a tag *highway=**.

There is no processing involved to extract *route paths* from *route ways*. We simply iterate over the *route ways* using *routes ranges* and collect the *route paths* from ways that are not present in *CorrectedWaysIds*. We use our component *OsmWaysHandler* to do so. *OsmWaysHandler* is derived from the *osmium* library *Handler* class, as introduced in Section 4.1.1.

A *route path* is represented by the node ids list of a *route way*. After all *route paths* were collected, we build a *NodeIndex* containing all needed node data with the *OsmNodesHandler*. With the *route paths* and the *NodeIndex*, we can perform our corrections on the elevation data of the nodes. In Figure 16, the dataflow of the just described process is visualized.

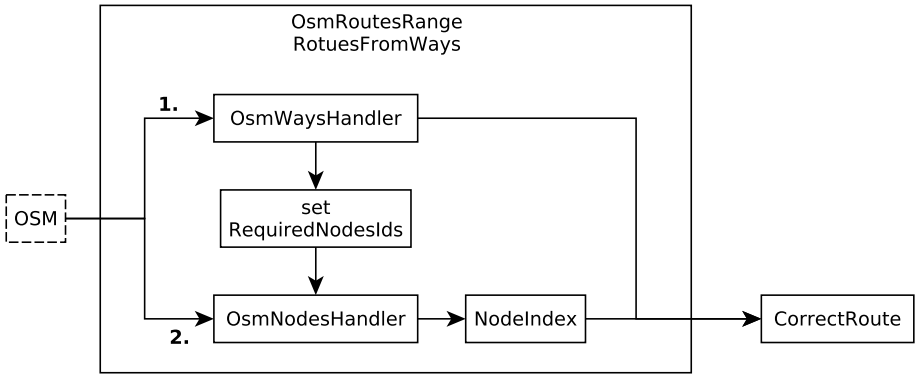

Figure 16: Dataflow of the processing of *route ways* in a *routes range*. Solid boxes represent the main components, dashed boxes are input/output files. Solid edges represent the dataflow. Numbered edges indicate the order in which the input files are read. In a *routes range*, the *OsmWaysHandler* performs a pass over all OpenStreetMap ways. Additionally to checking if a way is in the specified *routes range*, it is also checked that a way is not present in the `set` *CorrectedWaysIds*. If not, a *route path* is simply created from the way's ordered node ids list. After all *route paths* in a *routes range* were collected, a *NodeIndex* is built by the *OsmNodesHandler*. Corrections on the elevation can now be performed.

4.3.10 Correcting elevation data with *route paths*

In this section, we explain how we perform corrections on elevation data in the OpenStreetMap based on *route paths*. The *route paths* were before collected and processed from relations and ways. To perform the corrections, we need the *route paths* itself, a *NodeIndex* where the location and elevation data of the nodes can be retrieved, and the *AverageElevationIndex* to store the results.

We use three types of corrections, based on the type of the *route paths*. In the following sections, we introduce the types of corrections one by one.

Correcting a tunnel or bridge

In our correction process, we do not differentiate between a tunnel and a bridge. For both, we assume that the elevation data along the tunnel/bridge section is either incorrect or not available. The elevation data of the nodes along a tunnel/bridge should represent the elevation of the road itself that uses the tunnel/bridge to bypass an obstacle. But for a tunnel, this incorrect elevation data can correspond to the elevation of a mountain the tunnel crosses, for instance. The same applies to a road using a bridge to cross a deep valley.

The input of the correction is as already explained in Section 4.3.8 a `std::vector<std::vector<uint64_t>>` *TunnelOrBridge* that stores the following:

- $TunnelOrBridge[0]$: The *route path* directly in front of the tunnel/bridge segment.
- $TunnelOrBridge[1]$: The *route path* that represents the complete tunnel/bridge segment.
- $TunnelOrBridge[2]$: The *route path* directly after of the tunnel/bridge segment.

For our correction, we assume that the *route paths* in front of and after the tunnel/bridge have correct elevation data. We make use of this by spanning a 3-dimensional plane between a point on the *route path* in front of and a point on the *route path* after the tunnel/bridge. To obtain corrected elevation data for the tunnel/bridge section, we simply sample z-coordinates from the plane using the x-y-coordinates of the nodes locations. Before we can go into detail on how we do this, we need to build some data for the *route paths* nodes.

The data is extracted from the *NodeIndex* which stores node ids mapped to the locations and elevations of the nodes.

- `std::vector<std::vector<Coordinate>>` *NodeCoords*: Stores the coordinates by longitude and latitude of each node along the three *route paths* of the tunnel/bridge. *NodeCoords* has a size of three. For example, for the node $TunnelOrBridge[1][0]$, which is the first node of the tunnel/bridge segment, the location can later be retrieved with $NodeCoords[1][0]$. The locations can simply be retrieved from the *NodeIndex*.

- `std::vector<std::vector<double>>` *Elevations*: Stores the elevation of each node as retrieved from the *NodeIndex*. They can be accessed in the same way as we explained for *NodeCoords*.

With the *NodeCoords* and *Elevations* we can start spanning the plane. We first need a start and endpoint of the plane. For this we define *tunnel/bridge start* as the first node $TunnelOrBridge[1][0]$ and *tunnel/bridge end* as the last node $TunnelOrBridge[1][-1]$ of the of the tunnel/bridge section. We then choose the *plane start* and *plane end* as following:

- The *plane start* is the first node in $TunnelOrBridge[0]$ that has a distance of more than 30 meters from *tunnel/bridge start*. The distance is measured *along* the *route path* of $TunnelOrBridge[0]$.

- The *plane end* is the first node in $TunnelOrBridge[2]$ that has a distance of more than 30 meters from *tunnel/bridge end*. The distance is measured *along* the *route path* of $TunnelOrBridge[2]$. We use an approximation *haversine approximate* of the haversine formula[8] to calculate the distances along the *route paths*.

We want to obtain the parametric form of the plane which can be transformed to obtain the z-coordinate given x- and y-coordinates. The elevation of *plane start* and *plane end* can be retrieved from *Elevations*. Then, the longitude represents the x-coordinate, the latitude represents the y-coordinate, and the elevation represents

[8] `https://en.wikipedia.org/wiki/Haversine_formula`

the z-coordinate of the *plane start* and *plane end*. Thus,

$$startVector = (plane\ start\ lon,\ plane\ start\ lat,\ plane\ start\ elevation) \quad (9)$$

$$endVector = (plane\ end\ lon,\ plane\ end\ lat,\ plane\ end\ elevation) \quad (10)$$

$$directionVector1 = endVector - startVector \quad (11)$$

$$directionVector2 = (-1 \cdot directionVector1.y,\ directionVector1.x,\ 0) \quad (12)$$

$$planeNormalVector = directionVector1 \times directionVector2 \quad (13)$$

$$d = -1 \cdot (planeNormalVector \cdot startVector) \quad (14)$$

directionVector2 is parallel to the x-y-plane and normal to directionVector1. d is obtained by solving the linear equation $ax + by + cz + d = 0$ where a, b, c are the coordinates of the *planeNormalVector* and x, y, z are the coordinates of the *startVector*.

With this, we can create the formula

$$z = \frac{-d - planeNormalVector.x \cdot x - planeNormalVector.y \cdot y}{planeNormalVector.z} \quad (15)$$

to obtain the z-coordinate for given x-y-coordinates.

Using this formula (15), we can directly plug in longitude-latitude coordinates to get the elevation for a point in the tunnel/bridge section. We do this for all nodes of the tunnel/bridge section.

To store the results of the sampled elevation data from the plane, we add the sampled data to the *AverageElevationIndex*. Additionally, we update the elevation entries in the *NodeIndex* of the nodes that are in the tunnel/bridge segment. We will see in the next section why we do this.

Smoothing of *route paths*

The second type of corrections we perform is to apply is smoothing algorithm. We apply this to all *route paths* except for rivers. The *route paths* come from *route relations* and *route ways*. The input for the smoothing is a single *route path* represented by `std::vector<uint64_t>` *RoutePath*.

As in the correction of tunnels and bridges, we first need to extract some data from the *NodeIndex*:

- `std::vector<Coordinate>` *NodeCoords*: Stores the coordinates by longitude and latitude of each node along the *route path*. For example, the coordinate of the 6-th node along the *route path* can be extracted with $NodeCoords[5]$. $NodeCoords[1][0]$. The location data are retrieved from the *NodeIndex*.

- `std::vector<double>` *Elevations*: Stores the elevation of each node as retrieved from the *NodeIndex*. They can be accessed in the same way as we explained for *NodeCoords*.

- `std::vector<double>` *Distances*: Stores the distance in meters from each node along the *route path* to the first node in the *routepath*. For example, *Distances* can start with $[0, 5, 15, 20, ...]$. This means the second node has a distance of 5 meters from the first node. The fourth node has a distance of 20 meters from the first node. We build these distances by using the haversine formula[9].

We use the simple moving average for unevenly spaced time series as the smoothing algorithm. We have presented this algorithm in Section 3.4. The algorithm asks for a sequence of strictly-increasing observation times $T(X) = t_1, ..., t_{N(X)}$, and a sequence of observation values $V(X) = X_1, ..., X_{N(X)}$. We can directly use *Distances* as the sequence of observation times and *Elevations* as the sequence of observation values:

$$T(X) = t_1, ..., t_{N(X)} = Distances \tag{16}$$

$$V(X) = X_1, ..., X_{N(X)} = Elevations \tag{17}$$

We use a *two-sided rolling window* with a window width of 30 on both sides. This means we smooth each node's elevation along the *route path* using other nodes that are within distance 30 meters. We chose this rather small window size of 30 meters because we want to smooth short-term fluctuations only. Eckner [3] also provides an implementation in form of a *C library* on Github[10] of his presented simple moving average algorithm. We use this library in our implementation.

A shortcoming of this approach are the first few and last few observation values/*Elevations* values. Especially the first and last value get biased towards the trend of the subsequent and prior values, respectively. The first *Elevations* value does not have prior values available, thus only subsequent values will be used. If the corresponding *route path* is on a slope, the first smoothed value will be too high in comparison, the last value will be too low. This is the reason we try to extract *route*

[9] `https://en.wikipedia.org/wiki/Haversine_formula`
[10] `https://github.com/andreas50/utsAlgorithms`

paths that are as long as possible in the processing of *route relations*. The longer the *route paths* are, the less these biased first and last smoothed values can occur.

We store the smoothed elevation data in the *AverageElevationIndex*. The reason we stored the corrected elevation data from the tunnel/bridge corrections from Section 4.3.10 in the *NodeIndex* is as follows: We always perform the corrections on tunnels/bridges first. Bridges and tunnels were extracted from *route paths* that also get smoothed. This way the *route paths* containing the tunnels/bridges have access to the already corrected elevation data. If we would store the corrected elevation data of the tunnels/bridges only in the *AverageElevationIndex*, we would access the uncorrected elevation data. The result would be worse.

Correcting rivers

For rivers that we could find in *route relations*, we make sure that the rivers do not flow uphill. From the origin of the river onwards, the elevation of the nodes along the *route paths* should not increase.

A river is given as `std::vector<std::vector<uint64_t>>` *River*. A river can consist of multiple *route paths*. This happens if the river has side streams that split up from the main stream at some point. Side streams can also flow back into the main stream. The origin of the river is always contained in the first *route path* and represented by the first node, thus *River*[0][0]. Any subsequent *route paths* *River*[1], *River*[2], *River*[3], ... in the river are sorted by their time of first discovery.

To perform the corrections, we build the following data:

- `std::vector<std::vector<double>>` *Elevations*: Stores the elevation of each node as retrieved from the *NodeIndex*. The indices from *River* applied to *Elevations* yield the elevation for the nodes.

- `std::unordered_map<uint64_t, RiverNode>` *RiverNodes*: Stores all nodes of all *route paths*. Maps from the ids of the nodes to additional data. *RiverNode* stores the elevation of the node and a set `std::set<size_t>` *RoutePaths*. *RoutePaths* stores in which *route paths* the node is present by the index of the *route path* in *River*. Corrections in the elevation get stored in the *RiverNode*.

In Listing 13, the `struct` *RiverNode* is shown.

Listing 13: *RiverNode* `struct` to store the elevation and all *route paths* of the node

```
struct RiverNode {
```

```
  int16_t elevation;
  std::set<size_t> RoutePaths;
};
```

The trivial case represents a river that has a single main stream and no side streams. In this case it is sufficient to iterate over the nodes of the single *route path* and simply check if the current node's elevation is not higher than the prior node's elevation. But if there are multiple streams splitting and merging back together, hence multiple *route paths*, multiple passes over the *route paths* are necessary.

To cope with more complex cases, we iterate over the *route paths* until no more errors in the elevation were found. We use a set `std::set<size_t>` *CorrectAgain* to keep track of *route paths* that we need to check and correct again. We use the *RoutePaths* set in *RiverNode* to know all *route paths* the node is part of. If a correction was made to a node, all *route paths* in the *RoutePaths* set of the node are added to *CorrectAgain* and subsequently get checked again.

After no more errors could be found, the corrected elevation data gets stored in the *AverageElevationIndex*.

4.3.11 Runtime

We do not provide a detailed asymptotic runtime analysis of our second tool *correctosmelevation*. The runtime is dependent on the following two aspects, similarly as in our first tool:

- The number of *route relations* and *route ways* in the input OpenStreetMap file. The *route ways* have a lot more influence on the runtime as there are a lot more. The processing and correcting of *route paths* is linear in the total count.

- The amount of main-memory that is available. For our second tool, this is even more important than in our first tool. With more memory, we can decrease the number of *routes ranges* to cover all OpenStreetMap routes. This is important because in an *OsmRoutesRange*, there are non-linear data structures involved, mainly the sorting of the *NodeIndex* and *AverageElevationIndex* each *OsmRoutesRange*.

As for our first tool, the read and write performance of the *osmium* library benefits our second tool a lot.

In Table 6, a short example of the runtimes with of different input OpenStreetMap sizes is shown.

	germany	planet
runtime tool	4 minutes	102 minutes
runtime writing output OSM	4 minutes	61 minutes
maximum used main-memory	9GB	95GB

Table 6: Runtime of different input OpenStreetMap sizes of our tool *correctosmelevation*. Run on machine with *AMD Ryzen 7 3700X 8-Core/16-Threads and 128GB Ram.* Data was read from and written to *2TB NVME Samsung 970 Evo+*.

5 Experiments

In this section, we shortly present our results of using *correctosmelevation*. We compare the elevation data from our first tool with our second tool.

5.0.1 Results of our corrections on routes

In Figure 17, the uncorrected and corrected elevation profile of the Gotthard Base Tunnel[1] is shown.

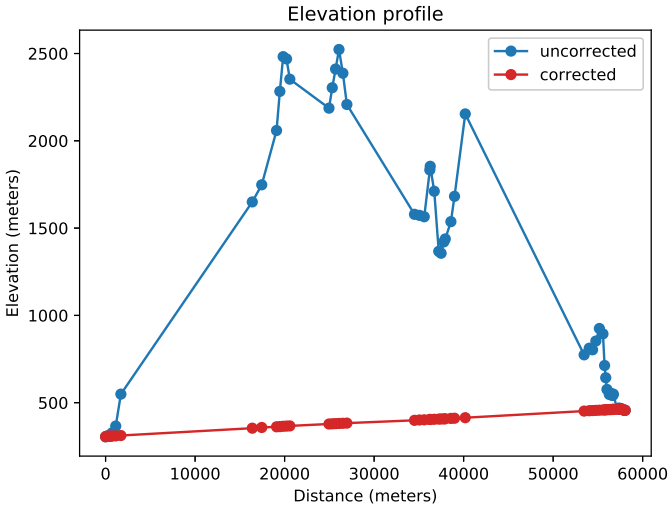

Figure 17: Uncorrected and corrected elevation profile of the Gotthard Base Tunnel. Plot generated from data of the OpenStreetMap ways *410992902*, *646970587*, *646970586*, *199658003*, *733561106*. The dots represent the nodes along the rails. Our tool *osmelevation* was used the to add the elevation data to OpenStreetMap. © OpenStreetMap contributors

This tunnel is a newly constructed railway tunnel. The tunnel has a length of around 57 km. In blue, we can see the uncorrected nodes along the railway. Uncorrected,

[1] `https://en.wikipedia.org/wiki/Gotthard_Base_Tunnel`

they represent the elevation of the mountain surface, the profile of the mountain. As for the elevation data from the NASA Digital Elevation Model (NASADEM), this is correct. But we want to represent the elevation of the map feature with the nodes elevation, in this case the railway which goes *through* the mountain. With our second tool *correctosmelevation*, we can correct this. As seen by the red marked nodes, our corrected elevation data for the nodes is more plausible for the railway.

In the introduction, we showed an elevation profile of a bridge.

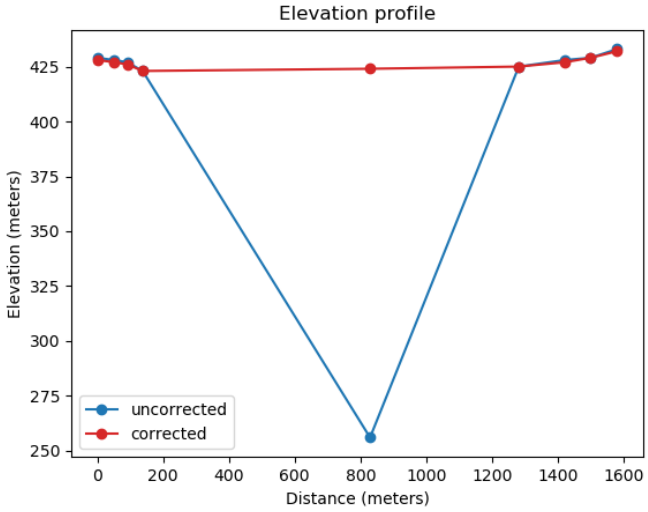

Figure 18: Uncorrected and corrected elevation profile of road crossing a bridge over a valley. Plot generated from data of the OpenStreetMap ways *403909403*, *320517373*, *320517370*, *24625636*, *24625669*. The dots represent the nodes along the road. Our tool *osmelevation* was used the to add the elevation data to OpenStreetMap. © OpenStreetMap contributors

We use this example again, as shown in Figure 18. For a bridge, the exact opposite to a tunnel happens. As we can see in the blue uncorrected elevation data, the elevation of the deep value is used by our first tool. Our second tool successfully corrects this. The corrected elevation proceeds as expected for a bridge.

6 Conclusion

Our tool *correctelevation* provides a simple way to enrich the complete OpenStreetMap data with elevation data. The elevation data can be taken as a basis to develop other tools that benefit from elevation data.

With our second tool *correctosmelevation*, we also provide a first use case for the elevation data in OpenStreetMap. We perform corrections on the elevation data we added before. Especially in tunnels and bridges, we can see good improvements with our corrections.

7 Acknowledgments

I want to thank *Patrick Brosi* to meet up with me on a regular basis. He provided help and suggestions regarding OpenStreetMap details.

I thank my friends who proof read parts of my thesis on last notice.

Also, I want to thank my parents who supported me and provided help.

Bibliography

[1] LP DAAC. *Data Citation and Policies — LP DAAC*, [Online; accessed 31-January-2022]. 2022. URL: https://lpdaac.usgs.gov/data/data-citation-and-policies/.

[2] B. Montibeller M. Muru E. Uuemaa, S. Ahi and A. Kmoch. Vertical accuracy of freely available global digital elevation models (aster, aw3d30, merit, tandem-x, srtm, and nasadem). *Remote Sensing*, 12(21):3482, 2020. https://doi.org/10.3390/rs12213482.

[3] Andreas Eckner. Algorithms for unevenly spaced time series: Moving averages and other rolling operators. *Working Paper*, 2019. http://www.eckner.com/papers/Algorithms%20for%20Unevenly%20Spaced%20Time%20Series.pdf.

[4] Farr and Kobrick. Shuttle radar topography mission produces a wealth of data. *Eos Transactions American Geophysical Union*, 81(48):583–585, 2000.

[5] Jet Propulsion Laboratory. *U.S. Releases Enhanced Shuttle Land Elevation Data — Jet Propulsion Laboratory*, [Online; accessed 31-January-2022]. 2014. URL: https://www2.jpl.nasa.gov/srtm/.

[6] Donald Shepard. A two-dimensional interpolation function for irregularly-spaced data. *Proceedings of the 1968 ACM National Conference*, page 517–524, 1968. https://doi.org/10.1145/800186.810616.

[7] F. Oda S. Naito K. Minakawa T. Tadono, H. Ishida and H. Iwamoto. Precise global dem generation by alos prism. *ISPRS Annals of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, II-4:71–76, 2014.

[8] TagInfo. *key=ele — Taginfo*, [Online; accessed 27-January-2022]. 2022. URL: https://taginfo.openstreetmap.org/keys/?key=ele.

[9] Jochen Topf. *Osmium Library — A fast and flexible C++ library for working with OpenStreetMap data*, 2013. URL: https://osmcode.org/libosmium/.

[10] OpenStreetMap Wiki. *Taginfo — OpenStreetMap Wiki*, [Online; accessed 27-January-2022]. 2022. URL: https://wiki.openstreetmap.org/wiki/Taginfo.

[11] OpenStreetMap Wiki. *Key:ele — OpenStreetMap Wiki*, [Online; accessed 28-January-2022]. 2022. URL: https://wiki.openstreetmap.org/wiki/Key:ele.