FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of bachelor's thesis

Title:	Web Application for OSM-Based Indoor Navigation
Student:	Matyáš Richter
Supervisor:	Ing. Marek Suchánek
Study program:	Informatics
Branch / specialization:	Knowledge Engineering
Department:	Department of Applied Mathematics
Validity:	until the end of summer semester 2023/2024

Instructions

Indoor navigation is very useful for larger buildings and groups of buildings (e.g. university campuses). However, it has various issues and different aspects when compared to traditional outdoor navigation. OpenStreetMap (OSM) provides so-called Simple Indoor Tagging to capture indoor map elements such as rooms, doors, elevators, and others. That can be used together with special map layers for indoor navigation. The goal of this thesis is to design and implement an OSM-based solution for indoor navigation:

- Survey and describe indoor navigation specifics and technologies. Also, describe the possibilities offered by OpenStreetMap.
- Design a method for creating a navigation graph from OSM data (for indoor navigation). Take into account the quality of open data in OSM. Then, design a method for finding an optimal path with various constraints or measures of optimality (e.g. minimizing total distance or wheelchair accessibility) with the use of path-finding algorithms. Formally describe the properties of the algorithms used.
- Briefly survey existing solutions for indoor navigation.
- Set requirements for your application and design the technical solution meeting the requirements in the form of a reusable web application (i.e. not limited to a specific building).
- Implement the application based on your design. Justify the selection of technologies used. Document your solution. Include a test suite and perform acceptance testing based on use case scenarios.

Electronically approved by Ing. Magda Friedjungová, Ph.D. on 18 November 2022 in Prague.

Bachelor's thesis

WEB APPLICATION FOR OSM-BASED INDOOR NAVIGATION

Matyáš Richter

Faculty of Information Technology
Department of Applied Mathematics
Supervisor: Ing. Marek Suchánek
May 11, 2023

Contents

Acknowledgments vii

Declaration viii

Abstract ix

List of abbreviations x

Introduction 1

1 Review of existing solutions and the OpenStreetMap project 3
- 1.1 Definition of (indoor) pedestrian navigation . . . 3
- 1.2 Survey of existing indoor mapping and navigation solutions . . . 4
  - 1.2.1 Google Maps . . . 4
  - 1.2.2 Apple Maps and IMDF . . . 5
  - 1.2.3 IndorGML . . . 5
  - 1.2.4 Other specialised applications . . . 7
- 1.3 OpenStreetMap . . . 7
  - 1.3.1 Organisational structure . . . 8
  - 1.3.2 Licencing . . . 8
  - 1.3.3 Data model . . . 9
  - 1.3.4 Editors . . . 12
- 1.4 Indoor data in OpenStreetMap . . . 15

2 Analysis 19
- 2.1 Formal requirements and use cases . . . 19
  - 2.1.1 Functional requirements . . . 19
  - 2.1.2 Ranking the requirements . . . 20
  - 2.1.3 Non-functional requirements . . . 20
  - 2.1.4 Use cases . . . 21
- 2.2 Architecture and technologies . . . 21
  - 2.2.1 Architecture overview . . . 21
  - 2.2.2 .NET and ASP.NET . . . 23
  - 2.2.3 Svelte . . . 23
  - 2.2.4 MapLibre and IndoorEqual . . . 23
  - 2.2.5 PostgreSQL, PostGIS, and pgRouting . . . 24
- 2.3 GIS concepts . . . 24
  - 2.3.1 Coordinate systems, projections and measuring distance . . . 24
  - 2.3.2 Spatial objects and the intersection matrix . . . 26
- 2.4 Routing . . . 28

3 Implementation **31**
3.1 Processing OpenStreetMap data 31
3.1.1 Import stage 31
3.1.2 Routing graph construction 32
3.1.3 Processing points 33
3.1.4 Processing lines 35
3.1.5 Processing polygons and multipolygons 35
3.1.6 Wall postprocessing 36
3.1.7 Extracting extra information for routing 38
3.1.8 Final graph cleanup 40
3.2 Client 40
3.3 Unit testing and CI/CD 44
3.3.1 GitLab Pipelines 45

4 Outcomes of user testing and implications for future work **47**
4.1 User testing 47
4.2 Future work 48

5 Conclusion **51**

Contents of the attached archive **59**

List of Figures

1.1 Google Maps Indoor in a mall in Bloomington, MN, USA 5
1.2 The data model of OpenStreetMap [34] 9
1.3 Example of a multipolygon relation representing a pedestrian area in Prague, Czech Republic [37] 12
1.4 Interface of the JOSM editor 13
1.5 Interface of the iD editor 14
1.6 Interface of the Rapid editor, the pink rectangle is a *Map With AI* suggestion . . 14
1.7 Interface of the StreetComplete app 15
1.8 Example of Simple Indoor Tagging, uploaded by user Pada [48] 17
1.9 Interface of the OsmInEdit editor 18

2.1 Diagram of the system's components. 22
2.2 The DE-9IM matrix of two intersecting polygons [61, 71]. 27

3.1 Entity diagram of the output tables of osm2pgsql. 32
3.2 Entity diagram of the routing graph tables. 33
3.3 Input and output of processing a way (line) with the `highway=*` tag 35
3.4 Output of processing a way (a closed polygon) using visibility graph construction, polygon nodes in blue, perimeter in black 37
3.5 Wall postprocessing that disconnects subgraphs in neighbouring rooms 39
3.6 The application displaying routing results with the default settings on a mobile device screen size. 41
3.7 The application displaying routing results with the default settings on a mobile device screen size with disallowed stairs. 42
3.8 The application displaying a routing result on a desktop screen size. 42
3.9 Sequence diagram of a user finding a route in the application. 43
3.10 Example of GitLab pipelines overview for a merged Merge request. [84] 45

4.1 A bug where the application suggests a route through a wall. 48
4.2 A bug where the application fails to find a route along a wall. 48

List of Tables

1.1 Example of the tags of a veterinary in Daejeon, South Korea [37] 10
1.2 Example of the tags of a footway in Cardiff, UK [37] 11
1.3 Example of the tags of a restaurant building in Bydgoszcz, Poland [37] 11
1.4 Common indoor tags, based on the Simple Indoor Tagging spec [48] and Taginfo 16

List of code listings

1 An example feature in the Indoor Mapping Data Format by Apple [11] 6
2 The `ProcessingResult` structure 34
3 Wall postprocessing algorithm pseudocode 38
4 An example from the test suite (with some cases removed for conciseness). 44

I would like to thank my supervisor, Ing. Marek Suchánek, for providing feedback and guidance throughout the whole process of writing this thesis. Additionally, I would like to express my gratitude to every single OpenStreetMap contributor, as they are the ones who made this thesis possible.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis. I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as a school work under the provisions of Article 60 (1) of the Act.

In Prague on May 11, 2023

Abstract

The goal of this Bachelor's thesis was to create an indoor routing application based on OpenStreetMap data. In the first chapter, a survey of existing indoor navigation technologies is presented, and the reader is introduced to the OpenStreetMap project and its specifics. Further on, the requirements for the application are analyzed, and key concepts of the implementation are described. A routing application has successfully been implemented, and the possibilities of its future development are discussed in the final chapter.

Keywords OpenStreetMap, pedestrian navigation, route planning, GIS, indoor maps, open data

Abstrakt

Cílem této bakalářeké práce byl návrh a implementace webové aplikace pro navigaci ve vnitřních prostorách na základě dat z projektu OpenStreetMap. V první kapitole je shrnuta rešerše existujících řešení pro navigaci ve vnitřních prostorách, a čtenáři je představen projekt OpenStreetMap a jeho specifika. V další části práce je provedena analýza požadavků na aplikaci, a poté popsány klíčové body implementace. Aplikace byla úspěšně implementována, a možnosti jejího budoucího rozvoje jsou rozebrány v závěrečné kapitole.

Klíčová slova OpenStreetMap, navigace pro chodce, plánování trasy, GIS, mapy vnitřních prostor, otevřená data

List of abbreviations

GIS	Geographic Information System
HTTP	Hypertext Transfer Protocol
JSON	JavaScript Object Notation
NGO	non-governmental organisation
ODbL	Open Data Commons Open Database License v1.0
OSM	OpenStreetMap
OSMF	OpenStreetMap Foundation
POI	Point of Interest
S3DB	Simple 3D Buildings
SIT	Simple Indoor Tagging
XML	Extensible Markup Language

Introduction

In large multi-floor buildings and complexes such as transport hubs, university campuses, and offices of public authorities, visitors are likely to know their destination but struggle with locating it. In an effort to make wayfinding easier for people not familiar with the building, pedestrian navigation systems can be of great help. Commercial solutions for indoor navigation exist but require a non-trivial investment by the building owner in terms of monetary and human resources, as well as vendor lock-in. The OpenStreetMap (OSM) project aims to create a map of the world in the form of a public database with an open data licence. In recent years, its community has agreed on a general-purpose indoor mapping schema called *Simple Indoor Tagging*. The primary targets for indoor mapping in OSM are public buildings, and many have already been mapped in detail. While tools for viewing those maps exist, there has not yet been a comprehensive effort to create a navigation system based on OSM data.

The goal of this thesis is to survey existing solutions in the field of indoor pedestrian navigation (or navigation in general) and to explore techniques for processing geospatial data. Subsequently, its goal is to design and implement an indoor navigation system based on OpenStreetMap data, available under an open-source licence.

Chapter 1

Review of existing solutions and the OpenStreetMap project

1.1 Definition of (indoor) pedestrian navigation

Navigation, routing, and wayfinding are all synonyms for the process of determining a path between two points on a map. With the rise of mobile computing, people have become accustomed to getting directions from a routing service. Such services are most popular among drivers, as shown in a 2022 study [1] by United Tires, which claims that about 60% of Americans use a GPS service at least once a week. However, road transport is not the only use case for routing. In dense urban environments, it could also be desirable to provide pedestrian directions.

In contrast to drivers, pedestrians are less constrained in their mobility options. While cars generally follow a network of roads made up of well-defined linear segments, people often have the option to walk freely in areas such as town squares and pedestrian zones. A pedestrian navigation service is expected to take open space into account. In addition to routing in areas, pedestrian navigation engines need to work with vertical dimensions as well. Since a building can contain multiple floors (up to 160 in the Burj Khalifa skyscraper, Dubai [2]), a single point identified by geographical coordinates can actually be ambiguous, representing multiple distinct locations. While this is also the case when it comes to road tunnels and bridges, even the most complex interchanges, such as the Judge Harry Pregerson Interchange [3] only contain a relatively low number of layers. Additionally, the layers of the road network can generally be easily mapped as distinct segments, removing the need to consider the vertical dimension when finding a route. Buildings introduce the concept of level connections, usually staircases, lifts, or ramps.

This thesis uses several basic concepts of graph theory, formally defined as [4]:

▶ **Definition 1.1.** *A **directed graph** G is an ordered pair (V, E), where*

- *V is a nonempty finite set of vertices (nodes), and*
- *E is a set of edges, ordered pairs of nodes.*

*$V(G)$ and $E(G)$ denote the sets of vertices and edges of G, respectively. A **weighted graph** is a graph (V, E, w), where $w : E \to \mathbb{R}^+$ is a **weight function** assigning a **weight** to each edge.*

▶ **Definition 1.2.** *Let G and H be two graphs. A mapping $f : V(G) \to V(H)$ is a **graph isomorphism** of G and H if:*

- *f is bijective and*

- *for every pair of vertices* $u, v \in V(G)$*:* $(u, v) \in E(G)$ *if and only if* $(f(u), f(v)) \in E(H)$.

If such mapping exists, we call graphs G *and* H ***isomorphic***.

▶ **Definition 1.3.** *Graph* H *is a* ***subgraph*** *of* G *if* $V(H) \subseteq V(G)$ *and* $E(H) \subseteq E(G)$*.* H *is an* ***induced subgraph*** *of* G *if* $E(H) = E(G) \bigcap \binom{V(H)}{2}$.

▶ **Definition 1.4.** *Let* $m >= 0$*. The* ***path*** P_m *is a graph with* m *edges:* $(\{0, \ldots, m\}, \{(i, i+1) | i \in \{0, \ldots, m-1\}\})$*. For a graph* G*, a subgraph isomorphic to some path* P *is called a* ***path in the graph*** G.

▶ **Definition 1.5.** *A graph* G *is* ***connected*** *if for any two vertices* u, v*, there is a path in the graph that contains* u *and* v*. An induced subgraph* H *of* G *is a* ***connected component*** *of* G *if* H *is connected and there is no connected subgraph* F *of* G *such that* $F \neq H$ *and* $H \subseteq F$.

For finding a route on a map, the map is first transformed into a weighted directed graph, where edges represent roads, footways and other elements that a human can traverse, called a *routing graph*. The goal is to find a path in this graph, given its end vertices. More specifically, the task is to find the path with the lowest possible sum of weights of its edges, or *length*.

Naturally, the result is dependent on the weight function used when constructing the graph. The choice of this function creates different measures of the optimality of a solution. When modelling routing graphs for finding paths in the real world, it is desirable to use a weight function that minimises the physical distance of the path. However, there are other factors to consider. Pedestrians usually prefer footpaths to roads. Some segments, such as stairs, may not be accessible to wheelchair users. The weight function must reflect these requirements by arbitrarily inflating the weight of unsuitable edges or removing such edges altogether.

1.2 Survey of existing indoor mapping and navigation solutions

The concepts of digital indoor mapping and navigation can hardly be called new. A number of commercial solutions and standards exist, originating both from major players in the mapping industry and specialised start-ups. In this section, I will point out several indoor mapping implementations and their specifics.

1.2.1 Google Maps

Google launched its mapping service in 2005 and has since become a leader in the business-to-customer (B2C) digital mapping market segment. The keys to their success, as thoroughly described in an article by Scott McQuire [5], were strategic acquisitions that helped Google deliver features that the competition lacked, such as a 3D view of the world with satellite imagery (Google Earth), or a global service for 360 ° street-level photos (Google Street View). The company also provided an API for embedding maps into third-party websites, which extended its market reach even further. Google even used to run a platform for map contributors, in some ways very similar to OpenStreetMap, with the significant difference of not providing the database back under an open licence as OSM does. To this day, many features of Google Maps rely on user-provided content: business reviews and photos are some examples of that.

Indoor mapping is also part of the company's product — Google allows the import of floor plans as part of their "business partners" program [6]. Upon zooming into a supported building, the user is shown floor plans, including an interface for switching floors. Since the implementation is proprietary and not publicly documented, the details are unclear, but as shown in Figure 1.1, indoor navigation is supported.

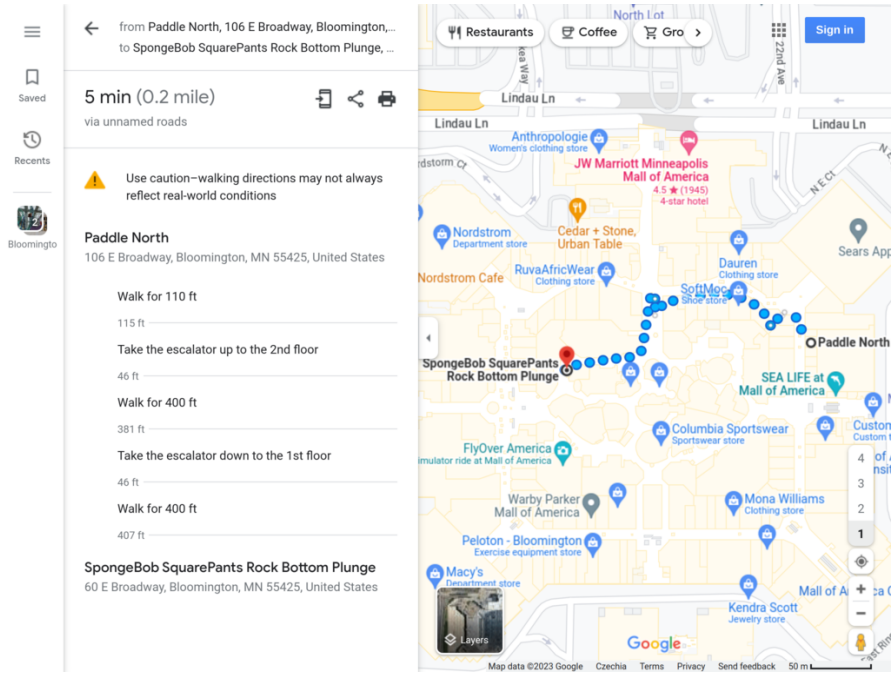

Figure 1.1 Google Maps Indoor in a mall in Bloomington, MN, USA

1.2.2 Apple Maps and IMDF

Apple's venture into the mapping industry has not been an easy one. After its launch as a replacement for Google Maps in 2012, the product received harsh criticism due to errors in map data, which even led to an apology from one of the company's executives [7]. Apple has continued to work on their map product, and public opinion seems to have also changed [8, 9].

When it comes to indoor mapping, Apple takes a vastly different approach compared to Google. The company developed a specification called *Indoor Mapping Data Format* (IDMF) and published it freely. It then worked with the Open Geospatial Consortium (OGC), an international consortium of businesses, government agencies, and research institutions, to have IDMF accepted as a "community standard" for indoor mapping [10]. Among the companies that pushed for this standard to be accepted was also Apple's competitor, Google. The format is based on GeoJSON, adding feature types to describe indoor structures [11]. Each building complex, like a public transport station, a shopping mall, etc., is represented as *venue*. A venue can have several levels, ordered by a numeric `ordinal` property. The geometry of a level describes its outline. Indoor features (*kiosks*, *openings*, …) have a mandatory `level_id` property. This system is very similar to OpenStreetMap's Simple Indoor Tagging, described in Section 1.4. For an example of an IMDF GeoJSON structure for a room, see Listing 1.

1.2.3 IndorGML

IndoorGML [12] is an extension of Geographic Markup Language (GML), an XML-based language for spatial information. It is an OGC standard for describing indoor spaces. The interior of a building is divided into *cells* that represent areas (rooms, corridors, stairs, …). The standard contains an Indoor Navigation Module, making the concepts of routable spaces and connections between them a first-class citizen. However, in my research, I have not come across any IndoorGML-based routing engines.

Another GML standard that allows mapping indoor spaces is CityGML. Its primary goal is

```
{
  "id": "11111111-1111-1111-1111-111111111111",
  "type": "Feature",
  "feature_type": "unit",
  "geometry": {
    "type": "Polygon",
    "coordinates": [
      [
        [100.0, 0.0],
        [101.0, 0.0],
        [101.0, 1.0],
        [100.0, 1.0],
        [100.0, 0.0]
      ]
    ]
  },
  "properties": {
    "category": "room",
    "restriction": null,
    "accessibility": null,
    "name": {"en": "Ball Room"},
    "alt_name": null,
    "display_point": {
      "type": "Point",
      "coordinates": [100.0, 1.0],
    },
    "level_id": "22222223-2222-2222-2222-222222222222"
  }
}
```

Code listing 1 An example feature in the Indoor Mapping Data Format by Apple [11]

to capture 3D building data. It allows multiple *levels of detail*, and the fourth level includes support for mapping indoor spaces. As noted by Ryoo, Kim and Li [13], CityGML has poor support for route analysis, because it is primarily concerned with modelling 3D space and the positions of doors, windows, ceilings, and walls, and omits information necessary for creating a routing network.

1.2.4 Other specialised applications

The demand for custom-made indoor navigation solutions led to a number of start-ups that specialise in this area, as well as several open-source projects. Companies in this field typically offer both mapping services (creating an indoor map from floorplans) and an application for viewing the map that contains a routing engine. They use a custom data format specific to each application. Since all of these services are very similar, I will not go into too much detail for each individual one, and only provide a list of a select few:

1. Steerpath is a Finnish company that provides a full suite for the mapping of building complexes: a map editor, a routing engine, hardware (Bluetooth beacons) for indoor positioning and APIs for integration with tools like booking systems. [14]
2. Mapstead offers a similar set of products and adds a marketing platform for location-bound promotions and an analysis tool for visits to the building. [15]
3. Indrz is a free and open source indoor mapping and navigation project built on top of the Django web framework and PgRouting. The application is used by several European universities including TU Wien and AAU Klagenfurt. [16]
4. iQNavs is an indoor navigation application developed at CTU in Prague. It contains an editing interface for floor plans and a routing engine. It is simpler than the other solutions mentioned, which could make deployment easier, but only supports navigation within a single building (or several directly connected buildings) without the context of the building's outdoor position. [17, 18]

1.3 OpenStreetMap

The OpenStreetMap (OSM) project [19] aims to provide a free, editable map of the whole world, built and maintained by volunteers and released with an open content licence.

Founded in 2004, OSM celebrated its 10,000,000th registered user in January 2023. The founder, Steve Coast, originally had a goal of mapping the United Kingdom, aiming to create an alternative to government-funded geospatial databases [20]. Although the organisations creating these databases had accumulated large volumes of detailed data, they did not make it easily accessible to the general public. The Ordnance Survey, Great Britain's national mapping agency, allowed third parties outside of the public sector to use its maps, but enforced a strict interpretation of "derived works" and collected high licencing fees. Thanks to its open licence, OpenStreetMap quickly rose as a competition to Ordnance Survey and similar bodies in other countries. Its influence, along with pushes from both nongovernmental and commercial organisations, played a role in opening up a large part of Ordnance Survey's data in 2010 [21, 22, 23].

In November 2010, OpenStreetMap finalised an agreement with Microsoft, which allowed royalty-free use of the Bind Maps Imagery Editor API [24]. This was a major stepping stone in attracting new mappers, as it was no longer necessary to create GPS track recordings as a source of positional data. Instead, editors could track map features based on Microsoft's high-resolution satellite and aerial imagery. This lowered the barrier of entry for new contributors. At the same time, OSM became available as a "map style" on Bing maps [25]. Today, imagery from several

other providers is supplied with a compatible licence and available in OpenStreetMap editors alongside the aforementioned Bing Imagery.

As the map improved over time, more businesses and organisations started using OSM data. Consumers range from online businesses such as Amazon, Apple, Facebook, and Microsoft through transport companies (Air France, Deutsche Bahn, Uber), geodata-related service providers (Garmin, Mapbox, ESRI), public and government organisations, other open databases (Wikimedia and Wikipedia), to small firms using OpenStreetMap for the purpose of displaying a map on their website [26]. Use for humanitarian purposes is also notable. The Humanitarian OpenStreetMap Team (HOT) connects volunteers from all over the world and helps organise the mapping of areas affected by natural disasters, underdeveloped regions, and previously uncharted territories of little value to commercial map providers [27].

1.3.1 Organisational structure

In August 2006, the OpenStreetMap Foundation (OSMF) was founded as a *company limited by guarantee*. It does not own OSM data, but plays several key roles in the project's operation [28, 29]:

- It acts as a legal entity representing the OpenStreetMap project.
- It manages the infrastructure necessary for its hosting.
- It participates in the organisation of fundraising for OSM-related activities.
- It organises the annual *State-of-the-map* conference.
- It supports and delegates tasks to its working groups. Working groups are made up of volunteers and act as organisational units in OSM's otherwise decentralised community. Among the most important groups are the Data Working Group (which resolves disputes regarding map editing, cases of vandalism, copyright infringement claims, etc.), and the Operations and Engineering working groups (which handle server maintenance and first-party software development, respectively).

Apart from the global Foundation, local communities have formed around OpenStreetMap. They are typically centred around a single country or another geographical area and provide a forum for discussing local mapping specifics, host mapping projects, and make map editing more accessible to people in their region. Many of them have become legal entities, usually NGOs, to be able to raise and distribute funds for mapping projects, make communication with authorities easier, and assist mappers in other ways. In recent years, it has become possible for them to become official *Local Chapters* registered with OSMF [30], allowing them to publicly associate themselves with the Foundation and potentially receive funding or other resources through its channels.

1.3.2 Licencing

Originally, the OpenStreetMap database was published under the Creative Commons Attribution-ShareAlike 2.0 (CC-BY-SA 2.0) licence. This was problematic for several reasons [31]:

- The CC-BY-SA 2.0 licence relies on the copyright of the data. Based on several cases in the United States, a precedent had been set, which stated that the information in a database cannot be copyrighted, only the collection as a whole. A requirement for copyrightability in the US (and possibly other jurisdictions) is the creativity of the "work" in question, which is, in the case of a geospatial database mirroring the real world, debatable.

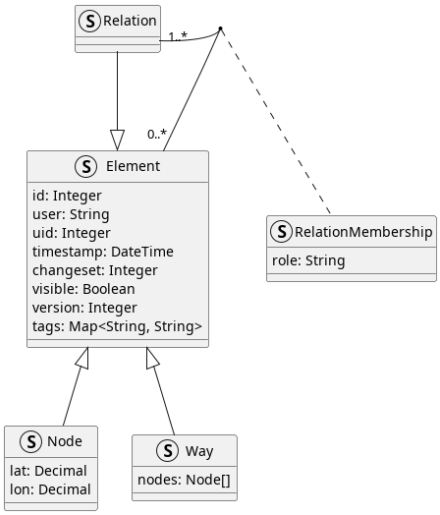

Figure 1.2 The data model of OpenStreetMap [34]

- A common use case for OSM data is rendering them on a map along with other information. The CC-BY-SA 2.0 licence requires the result to be released with the same licence, causing legal uncertainty in the case of combining the data with proprietary information.
- The licence did not encourage consumers of the data who made improvements to them to contribute those improvements back to OpenStreetMap's database.
- It was unclear whether the licence required attribution to every single contributor and legal liability should any of them file for copyright infringement.

In 2010, due to the concerns listed above, the OpenStreetMap Foundation held a vote on relicencing the database [32]. The licence was changed to the Open Database Licence 1.0 (ODbL). This licence creates a distinction between *Derivative Databases* and *Produced Works*. While Derivative Databases created from OSM data — databases which extend or modify the data — are subject to a Share-Alike and must also be licenced under ODbL, most OSM-based projects such as interactive maps, routing engines, machine learning models, or images, video and other media created from the data are considered Produced Works, and the licence only requires proper attribution [33].

1.3.3 Data model

The OpenStreetMap model aims for simplicity. All data in the database is modelled as one of the following *elements* [34]:

- *nodes* define points in space,
- *ways* are sequences of points and represent linear features or area boundaries,
- *relations* are used to add additional information to sets of nodes and ways.

As presented in Figure 1.2, elements are identified by numerical IDs, unique per element type. There are also several metadata attributes associated with each element, such as the timestamp, author, and changeset[1] of the last modification, or a numerical incrementing version.

[1] A batch of changes to the database, similar to a Git commit.

Node 368770565	
Key	**Value**
amenity	veterinary
name	둔산동물병원
name:en	Dunsan Animal Hospital
name:ko	둔산동물병원
name:ko-Latn	Dunsandongmulbyeongwon
ncat	동물병원

Table 1.1 Example of the tags of a veterinary in Daejeon, South Korea [37]

On their own, these records only describe the location and geometry of the real-world feature they represent. To interpret them, consumers need to know what the feature is. For that purpose, OpenStreetMap associates a set of *tags* with each element. Tags are key-value pairs, so each element's tags form a dictionary data structure. There are no validations in place for tags, so it is possible to add any key-value pair to any element. However, the OpenStreetMap community has agreed on a set of commonly used tags, which are documented in the project's wiki. The OpenStreetMap Foundation regularly holds votes on new tags to be standardized by adding them to the wiki. The *taginfo* [35] service provides statistics on the usage of tags and allows filtering by area, element type, and other criteria. It is a useful resource for determining which tags are used in practice and comparing the popularity of different tags to information from the wiki.

1.3.3.1 Nodes

Nodes are the basic building blocks of OpenStreetMap. As of March 2023, the database contains about 8.2 billion nodes [36]. They are defined by latitude and longitude, although, as discussed in greater detail later in this thesis, there are ways to differentiate points along the Z axis, too: the `ele=*` (elevation), `layer=*` and `level=*` tags describe the vertical location of a node. When used on their own, nodes usually represent features that are too small or insignificant to have their geometry represented on the map, such as wastebaskets, tourist shelters, or parcel lockers. In some countries (including the Czech Republic), address nodes placed on top of buildings are preferred over adding address tags to the building itself. Another common use case of standalone nodes is amenities inside a building that do not take up the entire building, such as shops in mixed-use neighbourhoods.

In Table 1.1, we can see the tags of a node in Daejeon, South Korea, which represents a veterinary. Other than the widespread `amenity=*` tag, the example contains two notable keys. The `name=*` tag describes what the real-world feature is called, and is often accompanied by "co-tags" with a language code suffix. This way, the map can be rendered in different languages. The other interesting tag is `ncat`. It is an example of a tag used by a local community — searching for it on taginfo [35] shows that it is used exclusively in South Korea. In this case, its value translates to "animal hospital". Since there is no wiki entry for this tag, it is not clear what exactly it is used for, but it seems that it might be a case of a widespread issue in OpenStreetMap: mappers sometimes add custom tags even though a standard for that type of feature already exists (`amenity=veterinary`).

1.3.3.2 Ways

Ways are ordered sequences of nodes. Two kinds of ways exist, they can be either linear or form a polygonal area. Whether a way belongs in one category or the other depends on both its shape (ways representing areas must be closed, i.e. start and end at the same node) and tags. Not all

closed ways are areas - a roundabout, for instance, can be mapped as a circular way, usually with a `highway=*` tag and optionally `junciton=roundabout`. For most tags, there is a consensus on whether they imply that an object is an area. The `building=*` tag is an example of a tag that is used to map the area of a building, and therefore should not be interpreted as a line (though this behaviour could be overridden with `area=no`).

Table 1.2 shows the tags of a way located in Cardiff, UK. From the tags, we can see that the way represents a footway with a designated bicycle path (mixed with a pedestrian path). The way is located on a bridge, and since it overlaps another feature (in this case, the river under the bridge), it is marked with `layer=1`, which tells the renderer to draw the footway above the river.

Table 1.3 shows the tags of a building in Bydgoszcz, Poland. We can see that the way has a `building=yes` tag. This key-value combination is in fact the most commonly used one in the entirety of the OSM database — according to taginfo [35], 4.81% of all tags are `building=yes`. As for the other tags: `amenity=*` signifies that the element is a *point of interest*, in this case, a restaurant. The set of tags prefixed with `addr:`, as the name hints, contains address information. In some areas, the convention is to put `addr:` tags on the building itself, while in others, a separate "addres node" is preferred.

Way 443633472	
Key	**Value**
bicycle	designated
bridge	yes
footway	sidewalk
highway	footway
layer	1
segregated	no
surface	asphalt

Table 1.2 Example of the tags of a footway in Cardiff, UK [37]

Way 197799694	
Key	**Value**
addr:city	Bydgoszcz
addr:housenumber	6a
addr:postcode	85-030
addr:street	Świętego Floriana
amenity	restaurant
building	yes
name	Restauracja Zatoka

Table 1.3 Example of the tags of a restaurant building in Bydgoszcz, Poland [37]

1.3.3.3 Relations

The most complex data structures in the OSM database are relations. They are sets of other elements which add new attributes to them. Below is a list of the most common relation types, which are differentiated by the `type=*` tag on the relation itself.

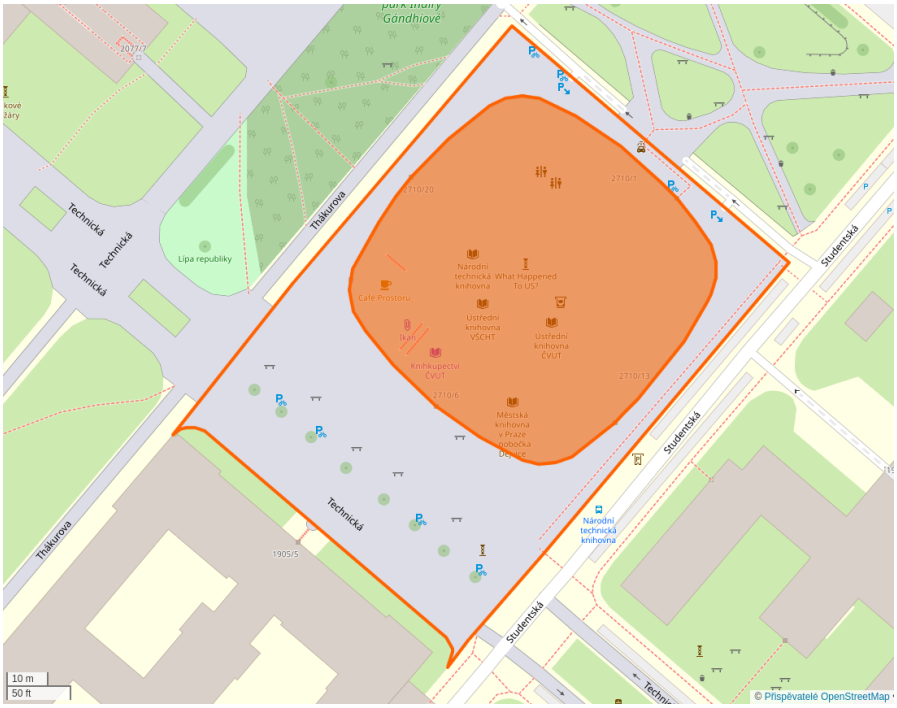

Figure 1.3 Example of a multipolygon relation representing a pedestrian area in Prague, Czech Republic [37]

- *Multipolygons* (54% of all relations [35]) are used to model complex areas, such as ones with "holes" in them. Figure 1.3 shows an example of a multipolygon relation. The relation is composed of two ways, one representing the outer boundary of the area, and the other representing the inner boundary (in this case, a building). The inner way is marked with `role=inner`, and the outer way with `role=outer`.
- *Restrictions* (16%) describe traffic turn restrictions. They usually contain the `from`, `to` and `via` member nodes, and are used in applications that provide driving directions.
- *Routes* (8%) are used to model long-distance routes, such as cycle routes and hiking trails, that span multiple ways. Public transport routes can also be mapped using this relation type.
- *Boundary* (6%) relations define administrative boundaries of countries, counties, cities etc.

1.3.4 Editors

The software used to add and alter the OpenStreetMap database is referred to as **editors**. This section briefly describes a subset of them. All of these programs are distributed free of charge with an open-source licence.

1.3.4.1 JOSM

The **Java OpenStreetMap** editor [38] is a desktop application written in, as the name implies, Java. It is arguably the most advanced OSM editor, popular among experienced mappers. Compared to other tools, its interface (Figure 1.4) has high information density at the expense of higher complexity. A major advantage of JOSM is extensibility. Its architecture allows straightforward extension of its features through Java plugins, with tools aimed at specific areas of

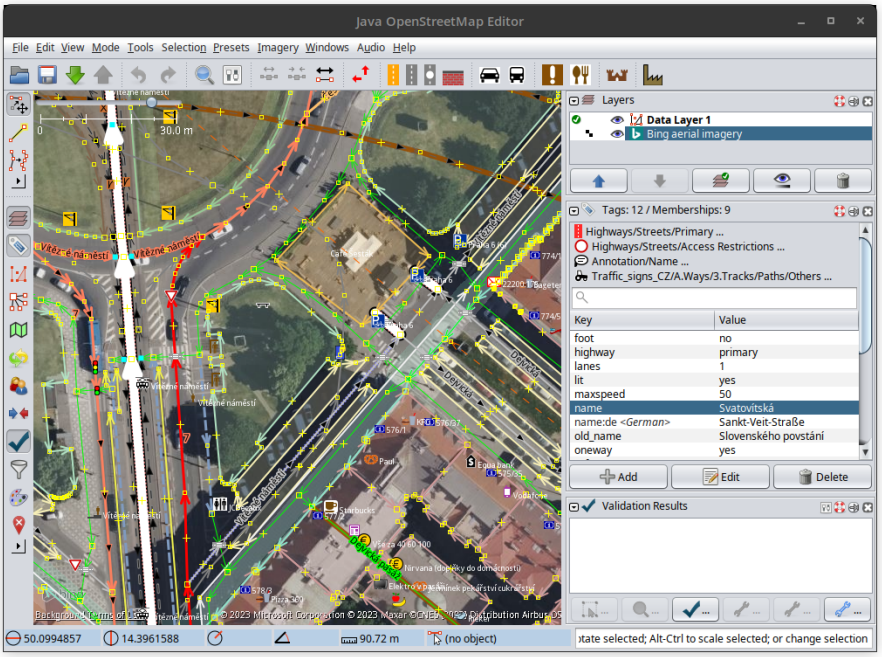

Figure 1.4 Interface of the JOSM editor

mapping, such as advanced geometry manipulation tools, different imagery providers, or helpers for different tagging schemes (e.g. Simple Indoor Tagging, see Section 1.4).

1.3.4.2 iD

iD [39] is in many ways a polar opposite of JOSM. It is browser-based, eliminating the need for local installation, and aims for simplicity and user-friendliness. The *Edit* button on the OpenStreetMap homepage opens iD by default, making it the first editor most people encounter when trying to edit OpenStreetMap. It includes several imagery sources, including local ones that it only suggests for covered areas (for example, orthophoto by the Prague Institute of Planning and Development). The editor also has a number of presets, sets of predefined tags, for common features — opening hours for businesses, surface types for roads, etc. — and quick documentation previews that partially relieve the user of having to study the OSM wiki thoroughly.

1.3.4.3 Rapid

Originally called **RapiD**, this editor is a forked and modified version of iD [40] developed by *Meta Platforms*. It offers two additions to the original:

- The rendering engine has been rewritten using *WebGL* to improve performance in areas with a large number of elements.
- It includes “Map With AI” - a component that suggests edits based on generated datasets — Facebook roads, Microsoft Buildings, and government-published open datasets. The user can choose to accept these suggestions instead of mapping the road or building outline by hand, which can speed up mapping new areas.

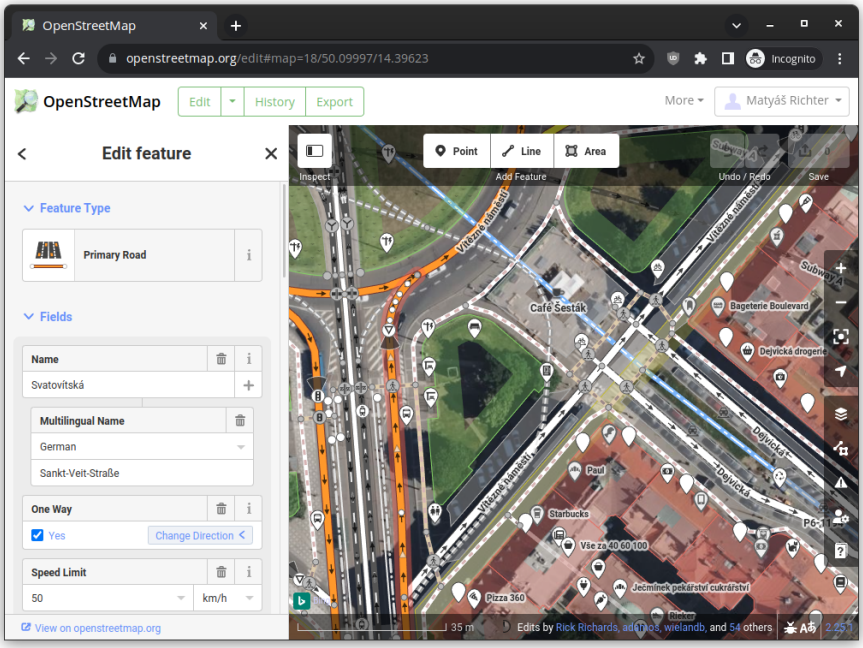

Figure 1.5 Interface of the iD editor

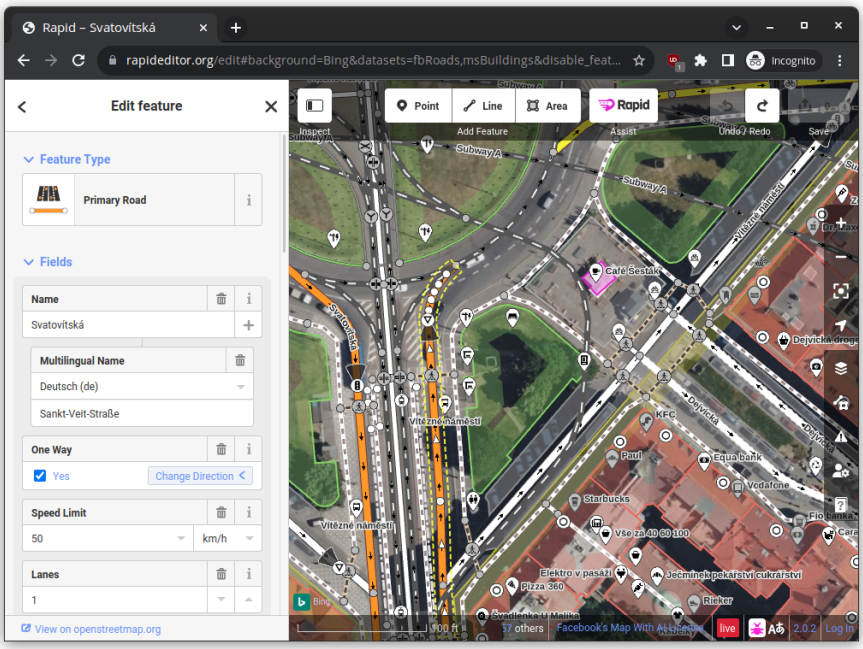

Figure 1.6 Interface of the Rapid editor, the pink rectangle is a *Map With AI* suggestion

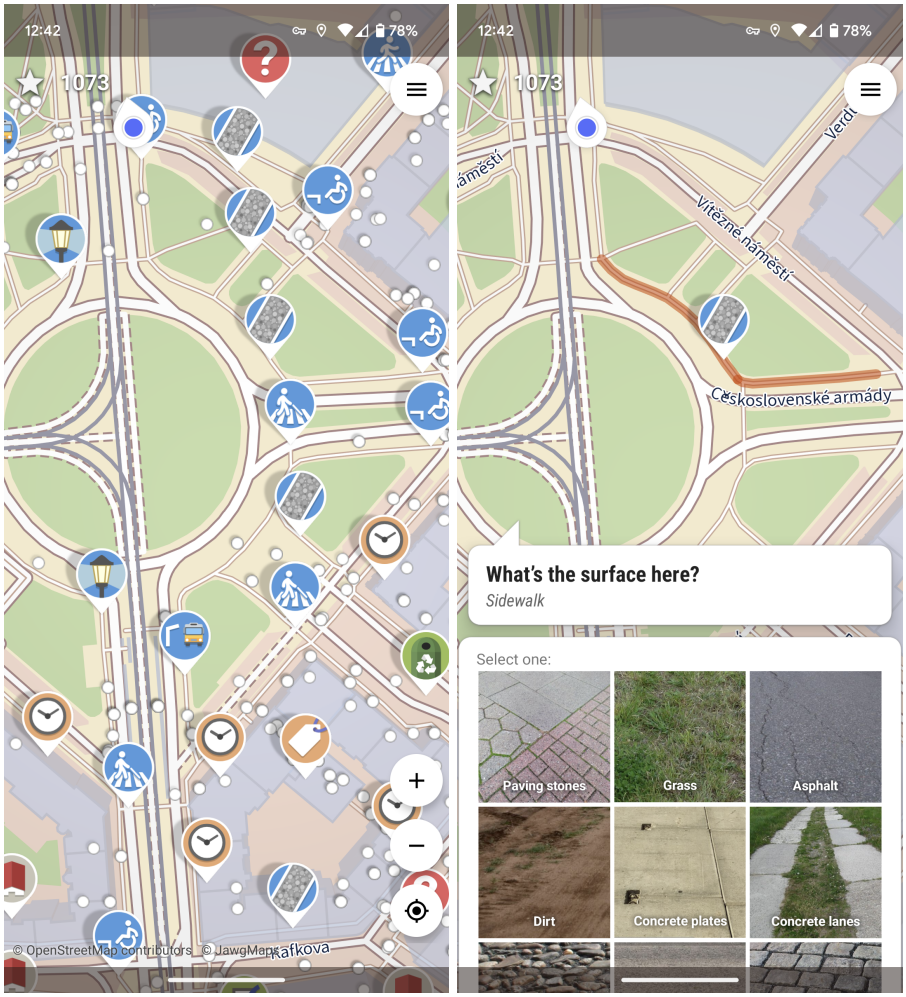

Figure 1.7 Interface of the StreetComplete app

1.3.4.4 StreetComplete

StreetComplete [41] is an Android application that takes a very different approach to map editing. Instead of trying to become a full-fledged mobile editor, simplicity is placed above everything else on the list of priorities. The target audience are casual mappers without extensive knowledge of tagging standards. The app prompts the user with *quests*, easy-to-answer questions about things around them in the real world, such as "What kind of surface does this road have?", "How many levels does this building have?", "Is there a bicycle lane on this road?" etc., often with straightforward multichoice answers and example pictures. It also promotes different OSM projects through non-intrusive gamification — for instance, answering a preset number of bicycle infrastructure quests "unlocks" a link to a cycling map renderer.

1.4 Indoor data in OpenStreetMap

Over time, as the OpenStreetMap database kept growing, some mappers started to shift their focus to smaller and smaller details in their local areas. This trend was caused mainly by the fact that those areas were already well-mapped in terms of primary roads, place names, city boundaries, etc., and people who wanted to contribute were looking for things that had not been

mapped before. When it started to gain traction, the term *micromapping* [42] was established as the name of this trend. A natural extension of efforts to map fine details in cities was mapping indoor spaces.

In the 2010s, drafts of indoor tagging schemas first started appearing [43]. In the years 2011-2014, four notable alternatives were proposed:

- **IndoorOSM** [44] and **CompoundFacility** [45] were heavily based on a hierarchy of relations. Each floor was to be modelled as a relation containing all the features on that floor as members. The level relations were members of another relation.
- The **Termite** [46] editor introduced an indoor tagging schema that also used relations for different floors. The author's goal was to make standardised indoor mapping accessible to non-technical people. The project was very ambitious but eventually became abandoned.
- The **F3DB** [47] proposal, short for "Full 3D buildings" took a different approach. An `indoor:*` tag namespace was added, with a set of tags similar to existing outdoor tags, such as `indoor:highway=*`, `indoor:level=*` etc.

In the end, the OSM community has converged on using a modification of F3DB. Proposed in 2014, the **Simple Indoor Tagging** (SIT) [48] schema's main objective is to stay as close to the outdoor tagging style as possible. It only introduces new tags for concepts that do not already exist, such as rooms. Doors, footways, stairs, elevators and other elements are mapped just like their outdoor counterparts, allowing existing routers, renderers and other software to partially support them even without special indoor modifications.

Additionally, the schema includes an `indoor=*` tag, which can be used to map corridors, rooms, and other indoor areas. Individual elements can have a `level=*` tag with numeric values that describe their vertical location. The connections between levels are mapped using a node with `level=*` and `door=*` tags, with a value of `door=no` for connections without doors, such as a node connecting an area representing stairs to another area on a different floor. Features that span multiple floors are mapped with a range value in the `level=*` tag. See Table 1.4 for an overview of common tags and values used in indoor mapping.

Tag and value	Usage
`indoor=yes`	Marks an object as indoor.
`indoor=room`	Room with walls. Walls can be passed through via `door=*` nodes.
`indoor=wall`	A free-standing wall, or a wall at the edge of an area.
`indoor={area, corridor}`	Unwalled area or passage.
`level=X`	Floor number the element is located on.
`level={X-Y, X;Y}`	Used when the element spans multiple floors, e.g. a staircase, a lift, …
`repeat_on={X-Y, X;Y}`	Replicates an element on multiple floors.
`door={yes, no, *}`	Doors and other openings in walls, or level connections at ending nodes of stairs.
`highway=elevator`	A lift. Can be mapped as a node, a way, or a `indoor=room` polygon.
`highway={footway, stairs, …}`	Same as outdoor `highway=*` tags, paired with `level=*` inside buildings.

Table 1.4 Common indoor tags, based on the Simple Indoor Tagging spec [48] and Taginfo

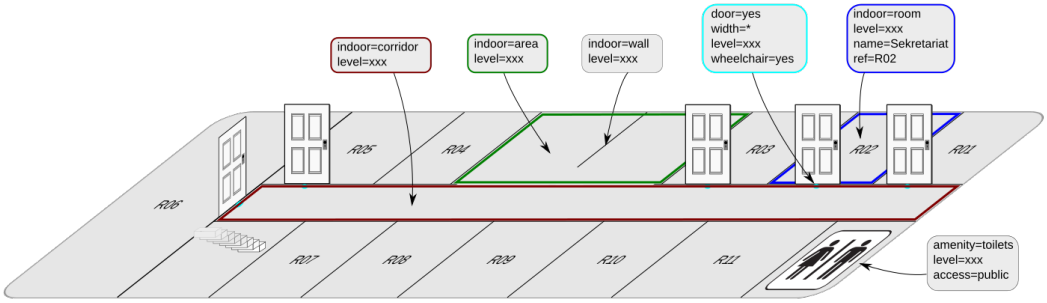

Figure 1.8 Example of Simple Indoor Tagging, uploaded by user Pada [48]

1.4.0.1 Indoor support in editors

The topic of this thesis is an indoor routing engine based on OpenStreetMap data. For it to be useful, it is important to be able to add indoor data to OSM easily, using an application that supports Simple Indoor Tagging. JOSM includes only basic filtering of elements by their `level=*` tag values. The iD editor does not support SIT natively either. It is, of course, possible to add indoor tags just like any other, but the user experience suffers from the inability to edit floors one by one, and the user is overwhelmed by a large number of elements stacked on top of one another. JOSM allows filtering by tags, which could seem satisfactory at first glance, but simply filtering by `level=*` is not enough, since it ignores *repeat_on=**. The same problem arises in iD. This inspired the creation of several specialised editors and modifications.

There have been two notable efforts to add indoor support to iD. In his master's thesis [49], Pavel Zbytovský aimed to add a special "indoor mode" to iD's interface and proposed modifications to the SIT schema regarding the use of ways for corridors. Unfortunately, his pull request was never merged. Parallel to his work, Adrien Pavie created a fork of iD called **iD-indoor** [50]. The project was halted in the hope that Zbytovský's work would soon be accepted into the iD core.

IndoorHelper [51] is a JOSM plugin that helps with SIT mapping. It extends the built-in level filtering to automatically populate the `level=*` tag of new elements and validates compliance with SIT. Its main advantage over other solutions is that experienced users can leverage the power of JOSM and its other plugins.

With efforts to include indoor support in iD ceasing, Pavie created an alternative browser-based editor, **OsmInEdit** [52]. Instead of forking a general-purpose editor, he chose the path of writing a dedicated indoor editor from scratch. Therefore, OsmInEdit focuses on the concepts of buildings, levels, and indoor features. It assists the user with input of `level=*` and `repeat_on=*` and provides a number of presets specific to the environment, for example, rooms with different `amenity=*` values. In my experience, however, it sometimes behaves unexpectedly when determining which nodes should be merged and which should be duplicated on different levels, leading to the rooms I created not being connected to one another.

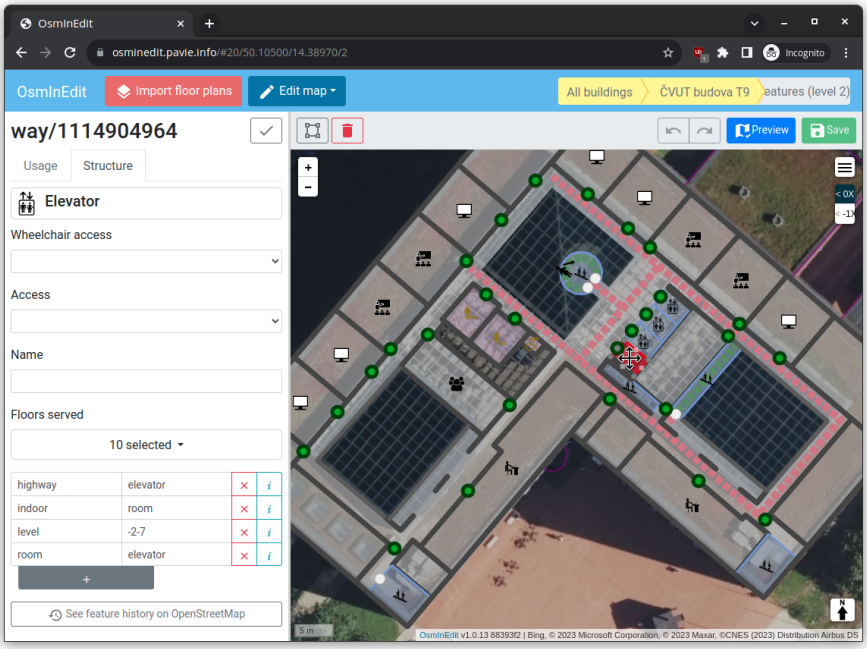

Figure 1.9 Interface of the OsmInEdit editor

1.4.0.2 Indoor support in renderers

The software that creates a map from OpenStreetMap data is called a *renderer* [53]. Renderers usually produce *tiles* by dividing the map into square areas of multiple tiers for each level of "zoom". They either produce raster image tiles, or vector tiles, blobs of binary data that the client application draws on the screen.

Indoor support in renderers is lacking, mainly because indoor data needs another component to work - a floor switcher, which is out of the scope of rendering a flat map. Two notable solutions for rendering indoor data exist, both of which support the Simple Indoor Tagging schema:

- **OpenLevelUp!** [54] is an open-source web application for viewing indoor data. Alongside pure rendering, it offers data filtering and tools for inspecting indoor elements on-click.
- **IndoorEqual** [55] is similar to OpenLevelUp, but more focused on displaying the data in a visually pleasing way. It also includes an API for embedding its indoor layer into other mapping applications.

Chapter 2
Analysis

2.1 Formal requirements and use cases

In this section, I will define the requirements for implementing an OpenStreetMap-based indoor navigation application. I will divide them into functional and non-functional requirements, and prioritise them using the MoSCoW method.

2.1.1 Functional requirements

Functional requirements describe features and functions that the application should offer.

F1: Pathfinding — The application will allow the user to find a path between two points on a map.

- F1.1 **Map browsing** — The main screen of the application will be a zoomable and pannable map.
- F1.2 **Start and target selection from the map** — The user will be able to pick route points from the map.
- F1.3 **POI search** — The application will let the user search for named points of interest, such as room codes.
- F1.4 **Multi-floor route display** — When presenting the route to the user, the application will clearly distinguish segments on different floors.

F2: Data import — The application will perform routing based on OpenStreetMap data.

- F2.1 **Routing area selection** — The administrator will choose a bounding rectangle for routing, such as an area around the building of interest.
- F2.2 **Raw OpenStreetMap data input** — The only necessary input will be an unmodified OSM data extract in a standard file format; or an equivalent API.
- F2.3 **Data updates** — When OSM data for the selected location changes, the application will allow updates.
- F2.4 **Data enhancement** — It will be possible to extend the imported data custom elements, such as extra POIs.

Requirement	Priority	Note
F1.1	MUST HAVE	Map zooming and panning is necessary to give the user an overview of the entire route while allowing them to view close-up details.
F1.2	MUST HAVE	The map view needs to be as interactive as possible.
F1.3	COULD HAVE	Extracting POIs from the map will let a use who find directions when they only know the name of their destination. It is, however, a complex process, resulting in a lower priority.
F1.4	SHOULD HAVE	After a route is found, users are interested in directions on each floor separately.
F2.1	MUST HAVE	This is useful for the administrator, who can only let users query paths in an area they have verified to be well-mapped.
F2.2	SHOULD HAVE	The person deploying the application should not be forced to waste time with pre-processing using external tools.
F2.3	COULD HAVE	Reimporting from scratch upon changes is also a viable strategy, hence the priority is lower.
F2.4	WILL NOT HAVE	Combining datasets brings a large set of issues with routability between points from different sources. It would, however, be an interesting feature to add in the future, and could be accompanied by a QR code-based positioning system (like iQNavs [17, 17]).

2.1.2 Ranking the requirements

I will use the MoSCoW method [56] to rank the requirements. The requirements will be divided into four categories:

- **Must have** — These requirements are essential for the application to work at all.
- **Should have** — These requirements are important, but not essential. They add value but are not strictly necessary for the application to be considered functional.
- **Could have** — Features that are nice to have, but have a lower impact on the user experience.
- **Will not have** — These requirements will not be implemented in the first version of the application. They may be implemented in the future.

2.1.3 Non-functional requirements

Non-functional requirements describe the properties of the application that are not directly related to its functionality and affect the technology choices during design and implementation.

NF1: Web interface — The application will be accessible in the end user's browser without the need for installation.

NF2: Responsivity — The user interface will be usable on a wide range of screen sizes.

NF3: Thin client — The application will not require significant computing power on the client side, i.e. it will be possible to use the application on a mobile device.

NF4: Containerized deployment — The server part of the application will use industry-standard containerized deployment techniques that remove dependencies on OS libraries (other than the container runtime).

NF5: Documented deployment without domain expertise — It will be possible to deploy an instance of the application without a thorough understanding of the underlying concepts of OSM and geographic information systems. The operations engineer deploying the application should not need to expend more resources on understanding how it works than necessary.

2.1.4 Use cases

When analysing the problem and designing the application, I have identified two main use cases, each with a different actor.

Use case 1: Data import

The actor in this use case is the *administrator* (for the lack of a better term), the person who is deploying the application, to offer navigation inside their building to their customers or visitors. After adding the data from their building's floorplans to the OSM database, they are ready for deployment. They follow instructions bundled with the application to find the bounding box of the routing area, start an import script, and end up with a database ready for routing.

Use case 2: Routing

The actor in this use case is a person seeking help with navigation inside the building. The use case has several linear steps:

1. The user selects a pair of start/target points from the map, potentially each on a different floor.
2. They click a button to start routing.
3. They are presented with a route drawn on the map, with a clear separation of levels.
4. They can zoom and pan the map to view route details on their way to the destination.

2.2 Architecture and technologies

In this section, I will briefly introduce each of the technologies selected for the implementation and compare them to other alternatives that were considered. Additionally, I will describe the architecture of the system and the way it is deployed.

2.2.1 Architecture overview

The system is composed of two main parts: import services and routing services. Both parts communicate with a PostgreSQL database, which stores a copy of OpenStreetMap data, extracted features, and the routing graph created from those features. See Figure 2.1 for a component diagram of the system.

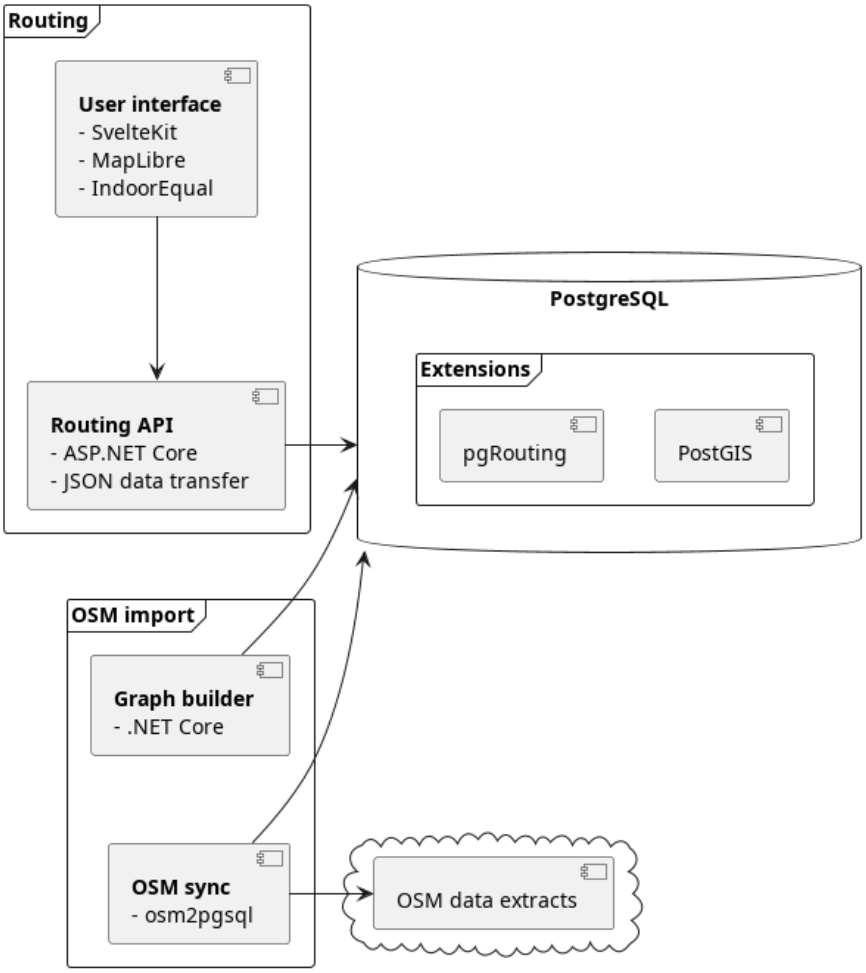

Figure 2.1 Diagram of the system's components.

Import is a one-time process with an optional periodic task for syncing the latest changes from the OSM database (or, more precisely, from data extracts). When the import is finished, the routing application comes into play. It follows a client-server model, where the server exposes a JSON HTTP API for the client running in a web browser. Thanks to the decoupling of the client and server parts of the application, the user interface can be interactive, and changes in the displayed data do not require a page reload.

2.2.2 .NET and ASP.NET

.NET [57] (formerly .NET Core) is a cross-platform open source developer platform. It supports several languages, primarily C# and F#, which run in the Common Language Runtime (CLR), a virtual machine that abstracts away the underlying operating system. Prior to the release of .NET, Microsoft had a product of a similar name, the *.NET Framework* — a proprietary platform for creating Windows apps. In 2014, they created .NET Core, a new cross-platform implementation of .NET. The "Core" suffix was kept during the transition, and dropped with the release of .NET 5 in 2020.

ASP.NET is a web application framework for .NET and C#. It has support for building web APIs, server-rendered web pages, websockets, and even single-page applications. It is a part of the .NET ecosystem and is maintained by Microsoft.

In this project, .NET and ASP.NET are used in two backend components: the routing API and the map processing service. It's a choice driven primarily by personal preference, but backed by objective reasoning. Being maintained by a large company, some of the points of pride of the ecosystem are its extensive and well-written documentation, and an established community, both of which elevate the developer experience. Another compelling reason to choose .NET for this project was the *NetTopologySuite* (NTS) library, a port of *JTS Topology Suite* from Java. It offers a set of tools for creating and manipulating geometries, and several related algorithms and data structures, such as spatial indexes. Many of these features are useful during map processing.

The arguments stated above also hold for the Python language paired with the Django web framework, which I considered as an alternative, and which I am more familiar with. However, I decided to go with C# and .NET for the benefit of better type safety, which I value highly.

2.2.3 Svelte

Svelte [58] is a JavaScript or TypeScript framework for web applications. It uses a component model similar to other alternatives like React and Vue.js. Unlike those alternatives, it does not use a virtual DOM (Document Object Model) — a technique that keeps the application's tree in memory and synchronizes it with the DOM. Instead, it compiles into JavaScript code that directly manipulates the DOM. In addition to Svelte, the application uses SvelteKit, a framework that adds a directory-tree-based routing system and server-side rendering on top of Svelte.

Since the main part of the application is the map, implemented using a separate set of tools, the choice of a frontend framework was not as important as it would be in a form-heavy "business" application. Due to that fact, I chose Svelte out of curiosity, because it is a tool I had not tried before, and without thorough evaluation of all alternatives.

2.2.4 MapLibre and IndoorEqual

Maplibre GL JS [59] (GL stands for WebGL, JS for JavaScript) is a library for browser map rendering. It is a fork of Mapbox GL JS, the product of the American company Mapbox specialising in custom maps and other related solutions. When Mapbox changed the license of their library to a proprietary one, MapLibre was born as a community-driven fork.

Alternatives include Leaflet and OpenLayers. Leaflet is not being actively developed anymore and only supports raster tiles. When deciding between MapLibre and OpenLayers, I inspected the documentation of both libraries and found MapLibre to be more approachable, although with fewer features.

The tile server is configurable, and the example configuration uses the MapTiler[1] cloud service. Alongside the base map, the application also uses IndoorEqual [55] (stylised indoor=), a tile server for a layer with SIT-compliant indoor maps. IndoorIqual also provides a plugin for MapLibre which adds a map control for switching between different floors. Both MapTiler and IndoorEqual can be replaced with a self-hosted version in order to avoid relying on external services.

2.2.5 PostgreSQL, PostGIS, and pgRouting

PostgreSQL [60] is an open-source relational database system. It is one of the most popular databases on the market, used heavily in the industry. PostgreSQL is extensible, allowing users to load third-party *extensions* into the database. Extensions add all sorts of functionality, such as new column types (*hstore*), indexing methods (*ZomboDB*, an abstraction over *ElasticSearch*), or new methods of data storage (*TimescaleDB* for time-series data).

I picked PostgreSQL for this project to be able to use two such extensions:

1. *PostGIS* [61] adds column types for storing geometric and geographic objects. It also provides a set of functions for manipulating these objects, and spatial indexes for efficient querying.
2. *pgRouting* [62] is an extension that adds several stored procedures for pathfinding and other graph algorithms. It is described in more detail in the following sections.

2.3 GIS concepts

2.3.1 Coordinate systems, projections and measuring distance

Locations on the surface of Earth are usually represented using tuples of numbers called *coordinates* [63]. A given location can be represented in many ways, depending on the *coordinate system* used, a reference for locating data across an area relative to each other. They are usually two or three-dimensional, depending on whether altitude is considered.

Coordinate systems can be divided into two categories [63]. *Geographic coordinate systems* (GCS) use a spherical model of Earth and usually describe locations using degrees of longitude and latitude. The most important distinction between geographic systems is their *datum*, which determines which model of Earth is used — since the planet is not perfectly spherical, different systems use different approximation methods. *Projected coordinate systems* (PCS), on the other hand, leverage a projection of the planet's surface onto a flat plane. The units of measurement in projected systems are usually linear, such as meters or feet. However, the farther from the centre of the projection a location is, the more distorted the coordinates become in that area, as the Earth's curvature diverts from the plane it is being projected onto. Geographic systems are useful for storing the location of data, and projected systems are necessary for drawing the same data onto a flat map. Sets of parameters for working with geospatial data, *spatial references*, are standardized in the EPSG Geodetic Parameter Dataset and identified by a *well-known ID* (WKID) or *spatial reference ID* (SRID) [64].

Applications that work with geospatial data combine both types of coordinate systems. OpenStreetMap is no exception [65, 66] — the database itself uses the 1984 version of the World

[1] `https://www.maptiler.com/`

Geodetic System (WGS-84, WKID 4326), but maps based on it (and web maps in general [64]) most commonly use the Web Mercator (WKID 3857) projection for displaying the data. The MapLibre library used in the web frontend of my routing application implementation works is aware of this — it draws the map using the Web Mercator projection but accepts coordinates as latitude and longitude.

When working with coordinates in any part of the application, the WGS-84 latitude/longitude is used. An important part of the construction of the routing graph is distance calculation. There are three "levels of approximation" we can use to calculate distance between coordinates [67, 68].

If we pretend the Earth is flat, we can use the Pythagorean theorem, converting degrees to meters using the length of meridians (20,003.93 km) and an estimate for the length of the parallel at the latitude of the line we are measuring — 40,901 km at the equator, scaling with the cosine of latitude:

$$\delta y = 20\,003\,930 \cdot \frac{|lat_1 - lat_2|}{180}$$

$$\delta x = 40\,075\,016{,}7 \cdot \frac{|lng_1 - lng_2|}{360} \cdot \cos\left(\frac{lat_1 + lat_2}{2}\right)$$

$$d = \sqrt{\delta x^2 + \delta y^2}$$

The distance d is an approximation, and its error increases with the distance between the two points, and is also larger in the vicinity of the poles. For small distances, however, it is "accurate enough". Mapbox, for example, uses it in their analytics pipeline, which is optimized for maximum performance [68]. Since it only uses two computationally expensive functions: the cosine and the square root, it is much faster than other more precise geodesic methods.

Another option is to assume the Earth is a sphere and use the *haversine formula*. The name "haversine" comes from a spherical trigonometric function:

$$\text{haversine}(\theta) = \sin^2\left(\frac{\theta}{2}\right)$$

Using the haversine formula, we can calculate the *great circle distance*, the distance along the perimeter of a circle on the surface of a sphere, which the two points lie at:

$$a = \sin^2\left(lat_2 - \frac{lat_1}{2}\right) + \cos(lat_1) \cdot \cos(lat_2) \cdot \sin^2\left(lng_2 - \frac{lng_1}{2}\right)$$

$$c = 2 \cdot \text{atan2}(\sqrt{a}, \sqrt{1-a})$$

$$d = R \cdot c$$

where R is the mean radius of the Earth. An advantage of this spherical approach is that the error is independent of distance, but rather tied to the latitude of the points, guaranteed to be lower than 0.5% even at the poles.

One last method exists, the Vincenty's formulae set for ellipsoidal distance and azimuth calculation [69]. This method is extremely precise, up to 0.5 mm, but also more computationally expensive, since it narrows down the approximation iteratively, using trigonometric functions heavily. And since the WGS-84 datum is only accurate to 1 m, the extra effort expended to calculate a more precise distance is wasted.

When calculating the lengths of edges in the routing graph, the vast majority of distances will be less than 100 meters, so the euclidean approximation would likely be sufficient. However, I decided to use the Haversine formula as a middle ground between precision and performance. Profiling the graph construction process later revealed that (ignoring time spent waiting for the database and parsing query results), distance calculation takes up less than 1% of the total CPU time.

2.3.2 Spatial objects and the intersection matrix

In GIS, we can come across several kinds of spatial objects. In this section, I will briefly describe the basic ones, introduce the concept of spatial relationships, and highlight both shared and differing concepts between general-purpose GIS and OpenStreetMap. Most of the text in this section paraphrases the PostGIS documentation [61] and the NTS documentation[70].

The most common spatial objects in two-dimensional space are:

- **Point** — a pair of coordinates.
- **Linestring** — a sequence of points, forming a line. When a linestring starts and ends with the same node, it is called a *closed linestring*.
- **Polygon** — a representation of an area. Its outer boundary is a closed linestring called a ring. A polygon may also contain inner rings, which represent holes in the area.

Each of these objects can be contained in a *geometry collection*: a *multi-point*, *multi-linestring*, *multi-polygon* or mixed collection.

OpenStreetMap data types can also be mapped spatial types, but it is not as straightforward as it may seem at first glance. The geometry of a node is a point. The geometry of a way is a linestring, but closed ways can also represent polygons, depending on their tags. In the case of relations, there are even more options: some multipolygon relations are simply polygons with inner rings, others relations, such as routes, are geometry collections (multilinestrings). The processing of OSM data is described in more detail in Section 3.1.1.

When working with multiple objects (geometries), we are often interested in their topological relationship. The Dimensionally Extended 9-Intersection Model (DE-9IM) is a model for describing that relationship used as a de-facto standard in many geospatial applications, libraries (NetTopologySuite), and databases (PostGIS).

Each kind of spatial object (points, linestrings, polygons etc.) can be decomposed into three parts: the *interior*, the *boundary* and the *exterior*.

- For a polygon, the boundary are its rings, both inner and outer. The interior is the area between those rings, and the exterior is the rest of the plane.
- For a linestring, the boundary are its endpoints. The interior is the line itself, and the exterior is, once again, the rest of the plane.
- In the case of points, the decomposition can be a bit confusing at first: the boundary is empty, the interior is the actual point, and the exterior is the rest of the plane.

To relate two geometries, we look at each of the three parts of both of them and compute the intersection. The result is a 3x3 matrix, filled with the dimensions of the intersections, in the order of *interior-interior*, *interior-boundary*, *interior-exterior*, *boundary-interior*, ..., *exterior-exterior*. Every two-dimensional relationship can be described using such an intersection matrix. In Figure 2.2, we can see the matrix for two intersecting polygons. The matrix is usually serialized to a 9-character string by reading it left to right, top to bottom. In the case of this polygon intersection, the matrix is `212101212`.

When asking the question "Is the relationship of these two objects *XXX*?", an *intersection matrix pattern* is usually used: for each of its 9 cells, the "query matrix" contains either the exact dimension (`-1/F`, `0`, `1`, `2`), or one the two wildcard symbols, `T` for "any intersection of dimension >= 0" or `*` for "any relationship". Because some kinds of relationships are very common, they have their own names. In the routing application implementation, the following named relationships are used (for geometries *a*,*b*):

- *Covers* — `{T*****FF*,*T****FF*,***T**FF*,****T*FF*}` — all points of *b* lie in the interior or on the boundary of *a*.

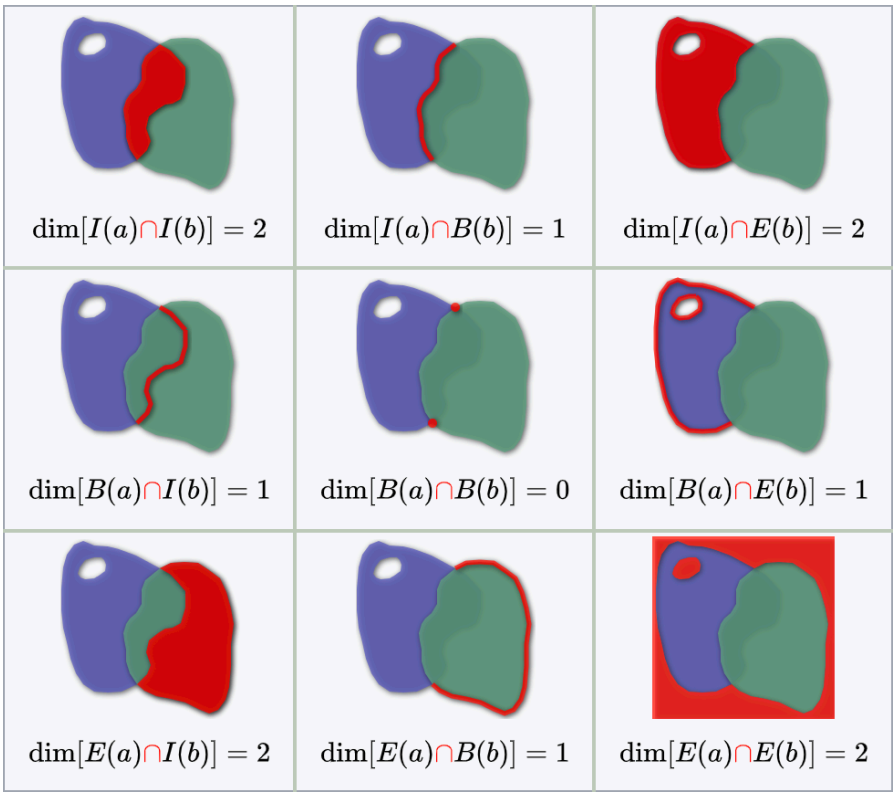

Figure 2.2 The DE-9IM matrix of two intersecting polygons [61, 71].

- *Contains* — `{T*****FF*}` — the interiors intersect, and no points of b lie in the exterior of a. Unlike *Covers*, boundary-only intersections do not match this relationship.
- *Intersects* — `{T********,*T*******,***T*****,****T****}` — a and b share at least one point.
- *Disjoint* — {FF*FF****} — a and b do not share any points.

Notice that some named relationships are logical negations of each other — i.e. *Intersects* means *Not Disjoint*.

2.4 Routing

I have identified several approaches to implementing routing. First, I will discuss their advantages and disadvantages, and then justify my final choice. I will refer to tasks, logic and computations executed during pre-processing OSM data as "processing-time", in contrast to "routing-time" code executed during pathfinding.

1. **Implement pathfinding on raw OSM data, or with minor pre-processing.**
   - \+ Low complexity of pre-processing code, great processing-time performance.
   - \+ Flexible routing with different constraints, since they can be interpreted directly from OSM tags.
   - – High complexity of routing-time code, since we need to evaluate the traversability of each edge, or even create edges on-the-fly.
   - – Slow routing-time performance.
   - – Redundant re-calculation of weights/costs for each routing request.
2. **Implement pathfinding on data pre-processed into a complete routing graph.**
   - \+ Relatively low complexity of routing code, essentially just an implementation of a well-known pathfinding algorithm.
   - \+ Expected relatively fast routing, and ability to fine-tune the algorithm.
   - \+ Construction of edges and weight calculation is only done once.
   - – Complex pre-processing code.
   - – A custom routing algorithm implementation is required.
3. **Use a universal routing service with a pre-processed routing graph.**
   - \+ Virtually no routing code - just a thin wrapper around the routing service.
   - \+ Expected high performance thanks to efficient implementation on the side of the routing service.
   - – Complex pre-processing code, which needs to prepare the data to the format expected by the service.
   - – The routing service may not support all necessary features, such as multiple nodes on the same coordinates. This depends on the service used.
4. **Extend an existing OpenStreetMap routing service with indoor routing support.**
   - \+ Routing is already implemented.
   - \+ Expected high performance thanks to optimisations done by the routing service.

+ Many pre-processing rules may be reused.
– Requires understanding of a foreign codebase.
– It might be difficult (or even impossible) to add some indoor routing requirements without major changes to the routing service. This is hard to assess beforehand.

After considering the options described above, I have decided to use approach #3. It provides a balance between flexibility and not "reinventing the wheel". OpenStreetMap routing services like GraphHopper and OSRM generally optimise for huge routing graphs (e.g. all of Europe), and focus on advanced algorithms such as *contraction hierarchies*[72], an algorithm that adds "shortcuts" to the graph during pre-processing, virtual edges based on paths between hotspots in the graph. Since our application is expected to be used within a small area, such as a university campus, these kinds of optimisations for a large graph are not critical.

The final choice is not a "service" per se, but rather a PostgreSQL database extension. Extensibility is one of the core features of PostgreSQL, allowing contributors to add new column types, functions, and even replication features. Other parts of this project are already using two major ones: *PostGIS* for spatial data types and indexes, and *hstore* for key-value column types. In addition to those, I have selected the *pgRouting* [62] extension for pathfinding. It adds a set of stored procedures not only for shortest-path algorithms but also useful graph analysis tools, such as a depth-first-search-based algorithm for finding connected components. Internally, it uses the C++ Boost Graph Library.

PgRouting needs the routing graph to be stored in the database, which is an approach I probably would have chosen anyway. All of its routing functions share a similar interface. They accept an SQL query string as a parameter, which must return a set of graph edges. The edges are then loaded into memory and processed by the selected routing algorithm. An implication of this is that all routing constraints and metadata must be representable in the edges table since this edges query should be as lightweight as possible. I will address this in more detail in a later section about OSM data processing.

My only concern with this choice is that it moves processing away from the application layer, into the database, which is not a common practice, and introduces scalability issues. For instance, should there be a need to run multiple replicas of the application for load-balancing, the database would become an even bigger bottleneck than normally. However, since routing is a read-only operation, this limitation could be circumvented by using read-only database replicas. PostgreSQL supports *streaming replication* out of the box, which is a good fit for this use case. It allows scaling the database horizontally by assigning primary and standby database nodes, where primary nodes send changes to the standby nodes, maximising the throughput of database read operations. This is, however, outside the scope of the current stage of this project, and should be only considered when proven necessary in the future.

For routing, the A^* algorithm [73] is used. A* is a search algorithm that uses a *heuristic function* to estimate the distance to the goal node and combines it with the length of the shortest known path to each explored node to determine the next node to explore. It can find the optimal path in graph G with a worst-case time complexity of $O(|V(G)| + |E(G)|)$, like the Dijkstra's algorithm which it is based on, but improves the "average case" using the heuristic function by exploring nodes with better prospects of being part of the optimal path first.

For A* to return optimal paths, the heuristic function h on a graph G, searching for paths to a target node n_t, must be:

- *admissible* - considering the hypothetical "optimal heuristic fuction" h^*, which always returns the length of the real shortest path to the target, the heuristic function h must, for every node $n \in V(G)$ and target node, satisfy $h(n) \leq h^*(n)$.
- *consistent* - for every pair of nodes $n_1, n_2 \in V(G)$, it must satisfy $h(n_1) \leq w(n_1, n_2) + h(n_2)$, where $w(n_1, n_2)$ is the weight of the edge between n_1 and n_2.

The heuristic function used for routing in this application is euclidean distance, calculated using the formula described in Section 2.3.1, with a slight optimisation. Because the implementation uses edges with metric weights (based on distance), but their coordinates are latitude and longitude, the heuristic function needs another parameter: a *factor*, which is used to calculate the δx in the equation. The factor is calculated using the average of the two latitudes of the routing area's bounding box. Since the routing area is rather small, it is a "good enough" approximation to compute the value of the heuristic function. This design choice is part of osm2pgsql, and the `pgr_astar` function expects a factor parameter.

Chapter 3
Implementation

3.1 Processing OpenStreetMap data

In this section, I will describe the implementation of raw OpenStreetMap data conversion into a routing graph.

As mentioned in the previous section, the routing graph needs to be in a format compatible with *pgRouting*. The output must be a database table with edges and their costs. As a side-effect of the simplicity of the format, swapping the routing engine for another one should be relatively easy, should that be necessary in the future.

The process is divided into two parts (stages) — first, OSM data is imported, and then the routing graph is constructed. It can run periodically and consume OSM data updates, so that changes to the live map are always reflected in the routing application. During updates, routing queries are not interrupted, and the routing engine only transitions to the updated graph once the update is complete. This is achieved using the concept of *graph versions*. When the graph update job starts, a new record is created in the `RoutingGraphVersions` table. Then, a new graph is produced, and this record is marked as active. All components of the routing graph have an attribute which associates them with a specific graph version. This way, the routing part of the application can keep using any given graph version and only switch to the latest one once it is ready.

3.1.1 Import stage

I used the *osm2pgsql* [74] tool to import OpenStreetMap data into the database before building the routing graph. It is a general-purpose solution for importing and pre-processing files in the OSM data format.

The import data source is configurable. OpenStreetMap publishes downloadable copies of the database known as *Planet OSM*[1], along with daily, hourly and minutely *diffs* — sets of changes in a special format meant to be used for replicating the database locally. To reduce the load on the main server, several organisations mirror these copies as a form of support for the OSM project. Often, they also create extracts from the database, limited to common areas of interest, such as to a single city, country or continent. This is ideal for the application developed in this thesis since it only needs data for the area around the target building.

Before starting the import, the administrator needs to specify two *bounding boxes* - rectangular areas described using their southwest and northeast corners. The first bounding box is for the processing stage, the second is for routing. As recommended in the manual for osm2pgsql,

[1] Available from `https://planet.osm.org/`

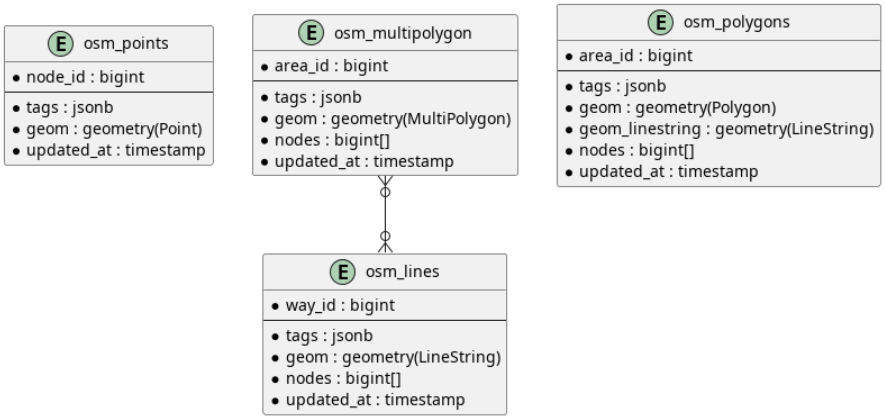

Figure 3.1 Entity diagram of the output tables of osm2pgsql.

the bounding box for processing should be slightly larger than for routing and fully contain it. The reason is that data on the edge of the bounding box may end up being incomplete, for example when a street starts inside the box, then heads out of it, before finally curving back inside. The issue is more severe for multipolygon relations. Since some members may be missing, multipolygons that represent an area can become unclosed and therefore invalid.

Osm2pgsql uses a user-provided script in the Lua programming language to configure the import. The script included in this project is based on an example from the osm2pgsql manual, and sets up five tables, as shown in Figure 3.1: a table for points (nodes with non-empty tags), lines (unclosed ways, e.g. highways), polygons (closed ways with tags that specify that the element is an area), and multipolygons (relation of type multipolygon, along with a many-to-many relationship table). It also strips tags irrelevant for routing, such as the `source` and `note` tags or reference numbers from data imports. The same script is used for the replication mode — elements from the changefiles are processed similarly, overwriting or deleting existing data.

I have included an example configuration in the source code repository for the application, set up for routing in the university campus in Dejvice, Prague. It uses extracts by the German company Geofabrik[2], and is set up as follows:

- When running the application for the first time, a Geofabrik extract for the Czech Republic is downloaded, cropped, and imported.
- Replication is initialized using Geofabrik's daily diffs service for the same area.
- The replication job is scheduled to run daily in the morning.

When deploying an instance of the application, the configuration can be updated to suit the individual needs of the user.

3.1.2 Routing graph construction

Once osm2pgsql has finished importing the data, it is time for the routing graph to be built. The *updater* service is implemented in .NET and makes heavy use of the NetTopologySuite library, which I already mentioned in Section 2.2. Graph construction is performed in several steps.

First, osm2pgsql output for the routing area is loaded from the database and stored in memory. The output contains all four types of elements: points, lines, polygons and multipolygons. Processing follows the same order.

[2]Available from `https://www.geofabrik.de/data/download.html`

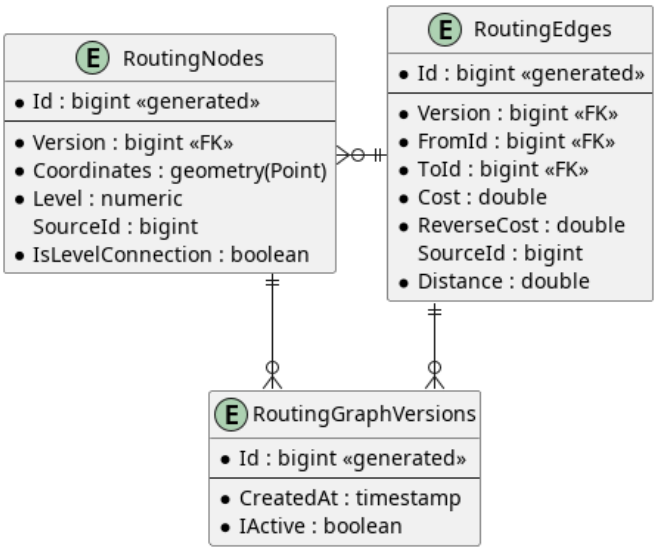

Figure 3.2 Entity diagram of the routing graph tables.

Each element is processed into a structure containing lists of nodes, edges, and wall edges, called `ProcessingResult` (see Listing 2). Each result is self-contained, meaning that the node IDs of each edge refer to indices in the list of nodes within the result. Once results for all elements (of one kind, e.g. for all points) are collected, nodes are concatenated and de-duplicated, edge IDs are re-mapped to their "global" equivalents, and everything is stored in an instance of a stateful `GraphHolder` class. Along with the list of nodes, a hash table with a (SourceId, Level) pair as the key is created. Before adding nodes from new `ProcessingResult`s, the hash table is checked, and if a node with the same SourceId and Level is already present, it is reused, and edges from two distinct elements end up with the same node ID, extending a connected component of the graph. Since processing an element into a `ProcessingResult` is independent, they can be processed in parallel, and only brought together afterwards, speeding up the graph construction process.

3.1.3 Processing points

First in line for processing are points, entries from the `osm_points` table created using osm2pgsql. Based on each point's tags, it is decided whether (and how) it should be added to the graph or not. Several kinds of points are considered:

- Points with the `elevator=yes` or `highway=elevator` tags are identified as **elevator points** and processed into one node per level, plus edges between those nodes. For example, a point with tags `highway=elevator; level=0-2` is processed into three nodes on the same coordinates, with levels 0, 1 and 2, and two edges between levels $(0, 1)$, $(1, 2)$. For the `Cost` property, a constant value of 3 is used as an estimate of floor height.

- Points with the `entrance=*` tag are processed into a node on the floor specified by their `level=*` tag, plus a node with `Level=0`. Outdoor nodes implicitly have `Level` value of 0. Buildings located in rising terrain often have entrances on different floors on each side of the building. For routing, outdoor nodes just outside the entrance would end up with a different `level=*` value, and thus would not get connected to the entrance. This is solved using an edge with `Cost=0` between the entrance node on its indoor floor and its copy with `Level=0`.

```
using NetTopologySuite.Geometries;

public enum SourceType
{
    Point,
    Line,
    Polygon,
    Multipolygon
}

public record struct Source(SourceType Type, long Id);

public record InMemoryNode(
    Point Coordinates,
    decimal Level,
    Source? Source,
    bool IsLevelConnection = false
);

public record InMemoryEdge(
    long FromId,
    long ToId,
    LineString Geometry,
    double Cost,
    double ReverseCost,
    Source? Source,
    double Distance
);

public record ProcessingResult(
    IList<InMemoryNode> Nodes,
    IList<InMemoryEdge> Edges,
    IList<(decimal Level, (int FromId, int ToId) Edge)> WallEdges
);
```

Code listing 2 The `ProcessingResult` structure

- Points with any value of `door=*` are processed into a node on the specified floor. This tag is used in Simple Indoor Tagging to mark level connections. Creating a node for these level connections in the first stage of processing ensures that they already exist when processing polygons, allowing nodes between the boundaries of indoor areas and corridors and a level connection in the middle of said areas to be created. For the application to support cases where mappers forget to add the `door=*` tag, all points with the `level=*` tag are also added, even though they are technically not SIT-compliant. Polygon processing is described in more detail later in this section.

- The same process applies to points with any of the `amenity`, `shop`, `name` or `ref` tags, as those are often the targets of routing requests.

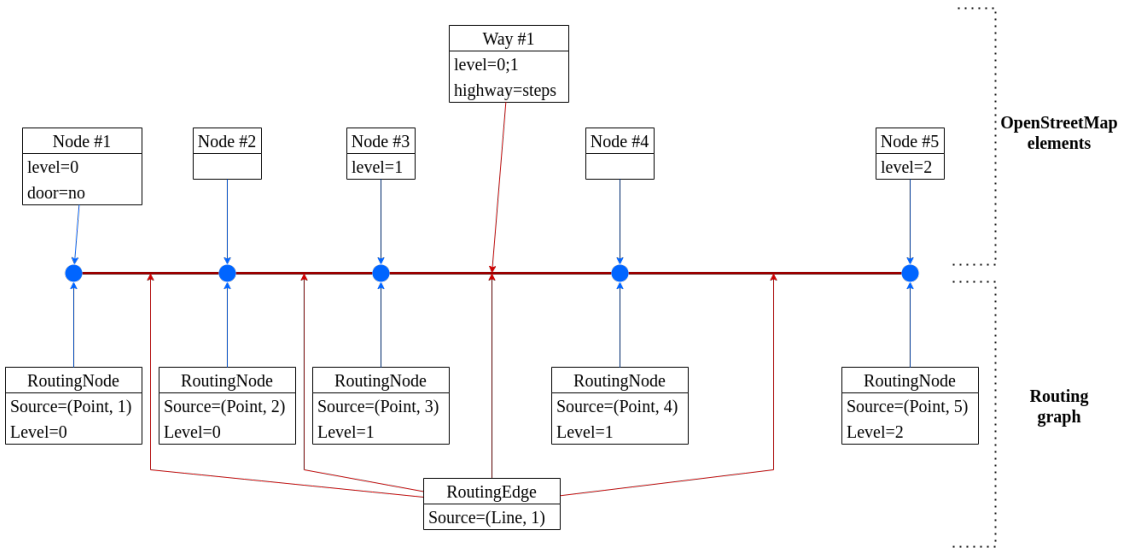

Figure 3.3 Input and output of processing a way (line) with the `highway=*` tag

3.1.4 Processing lines

Processing lines is also fairly straightforward. OpenStreetMap ways are sequences of nodes, and each node pair of that sequence can directly be mapped to an edge. The osm2pgsql import step outputs both a list of node IDs and a LineString. Iterating over these two sequences at the same time produces the information necessary for constructing routing graph edges. The `Cost` for edges created from way segments is calculated as the length of the segment in meters.

- Ways with the `highway=*` tag are cut into segments at each node, producing a sequence of edges (or multiple sequences depending on the value of the `repeat_on=*` tag). Although the Simple Indoor Tagging schema discourages the use of ways for mapping indoor pathways, many mappers use them. For instance, steps and escalators at the Berlin Main Station are mapped as ways with the `highway=steps` tag, connected to different levels using SIT-compliant `indoor=corridor` polygons[3]. To accommodate for level connections mapped this way, ways tagged with multiple `level=*` values are assumed to be meant to connect floors, and each of their OSM nodes is checked for the presence of its own `level=*` tag. If such tag is present, the rest of the line is treated as being on that level. This is visualised in Figure 3.3.
- Other than highways, there is a second kind of line that needs to be processed: walls and other linear barriers, identified by the `indoor=wall` and `barrier={wall,fence}` tags. No routing edges are created for these lines. Instead, their nodes are mapped to routing nodes, and the line itself is converted into wall edges, which are then used for wall postprocessing described in Section 3.1.6.

3.1.5 Processing polygons and multipolygons

The last kind of OSM element that needs to be processed is polygons. In the case of outdoor navigation, polygons are often overlooked, because only a few tags exist that represent routable outdoor areas. The most common one is the combination of `highway=pedestrian` and `area=yes` used for — as the name implies — pedestrian areas and plazas. Since area routing is a niche

[3] https://www.openstreetmap.org/way/268633035

use case for outdoor routing, many engines either do not support it at all, or route along the perimeter of the area, producing a detour. Implementing better area routing is a low priority for them, and feature requests for it have stayed in the "open" state in their issue trackers for years [75, 76].

Several algorithms exist for constructing routing subgraphs from polygons. Among the most commonly suggested are two types: visibility-based and grid-based. Several variations have recently been compared by a research group at Heidelberg University [77]. The visibility approach considers every pair of nodes of the polygon and creates an edge if the direct line between them is fully contained in the polygon. Its greatest strength is the fact that routing on the resulting graph produces optimal routes. Variations of the base algorithm aim to reduce the number of extra edges, for example by only keeping edges between nodes connecting the polygon's subgraph to the rest of the routing graph, trading path optimality for performance. Grid algorithms, on the other hand, create "virtual" nodes inside the area and connect them, forming a grid. Other variations construct also add diagonal edges to the grid, creating a mesh where each node has eight neighbours, called a *spider grid*. The advantage of grids and spiders over visibility graphs is that the start and target of routing can be inside a polygon.

For the indoor routing application, I selected the raw visibility graph approach. Indoor polygons tend to be smaller than large outdoor plazas, and the application is expected to operate on smaller graphs, too (as in, not spanning an entire continent), so I prioritised the precision of resulting paths. In the case of lines, edges were produced by iterating through lists of node IDs and coordinates and creating edges between each pair of consecutive nodes. For polygons, the algorithm is similar but considers all (unordered) pairs of nodes. For each pair, an edge is constructed, and related to the geometry of the polygon. If the polygon *covers* the edge, it is added to the graph. If the edge was to be partially outside of the polygon, it would mean that there is not a direct traversable line between the two nodes. The algorithm works both for polygons and multipolygons, since the *X covers Y* operation is defined for both types of geometries. See Figure 3.4 for a visualisation.

Additionally, edges need to be added for any routing node that is inside the polygon. The nodes could have been created in either of the previous two stages of processing - from a point or a line. To reduce the number of points that need to be checked, a k-d tree (or, in this case, 2-d tree) implementation from the NetTopologySuite library is used to retrieve all nodes within the polygon's bounding box. It is a binary search tree optimised for spatial data[78]. Its nodes divide space into two parts, causing each subtree to only contain nodes in the respective section of the space. A perfectly balanced two-dimensional tree has a maximum depth of $\log_2(n)$. For each node retrieved from the tree that the polygon *covers*, edges are created between it and each of the polygon's nodes, using the same visibility check as for edges created from the polygon itself.

3.1.6 Wall postprocessing

When processing polygons, edges are created between nodes on their boundaries. Because polygons share nodes with other elements, there are paths in the resulting graph that start inside one polygon and end in a neighbouring one, going through a node shared by both polygons. This is the desired outcome since the only way to get from one element to another is a shared node - whether the elements are lines, points or polygons. Routing engines that only operate on lines, for example only on the network of ways tagged as `highway=*`, are fine with it and only need to disconnect barriers tagged as nodes, such as `barrier=gate`.

Simple Indoor Tagging uses polygons as the primary routable elements. Many indoor polygons are rooms (`indoor=room`), and their boundaries are not supposed to be crossable, even if the node is shared. Walls can also be mapped as `indoor=wall` ways in the middle of a `indoor=area`, or using other combinations of tags representing a linear barrier, like `barrier=fence`. In either case, the routing graphs must not contain paths that penetrate a linear barrier, other than

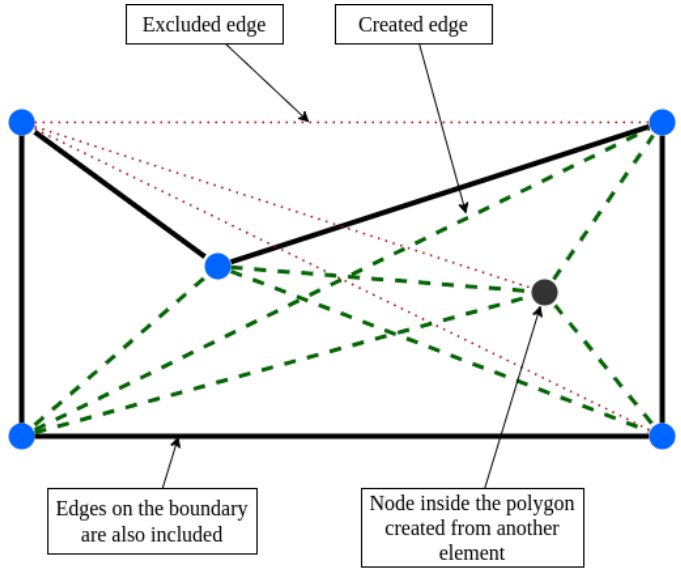

Figure 3.4 Output of processing a way (a closed polygon) using visibility graph construction, polygon nodes in blue, perimeter in black

through nodes that are explicitly tagged as *wall openings*, such as nodes with the `door=*` or `barrier=turnstile` tags.

Removing wall nodes is not an acceptable solution, because edges inside rooms are not redundant just because they are connected to a wall node. Instead, the edges need to be split into groups on opposite sides of the wall. During element processing, wall nodes have been collected alongside the rest of the routing graph. In the wall postprocessing step, they are aggregated and used to split wall nodes into copies on each side of the wall. Edges originally leading to wall nodes are then remapped, causing the two subgraphs in each of the rooms to get disconnected.

In Listing 3, we can see the algorithm used for wall postprocessing. While it may seem overwhelming at first, the core method is fairly straightforward — as visualised in Figure 3.5. For each wall node (that is also a routing node), the geometric shape of walls connected to it (its "wall edge star") is inspected, and a copy of the wall node is created, moved slightly into the space next to the wall, along the bisector of the angle between the two walls. On lines 19-27, edges that were originally connected to the wall nodes are instead re-connected to the new copy. Then, the geometry of the new edge is checked — and it is only added to the graph if it does not intersect a wall. This way, edges in each room are only connected to nodes inside the same room, that are no longer shared with areas behind a wall. Finally, edges on that level that cross a wall with their interior are also removed from the graph.

To speed up the process of finding candidate wall edges for intersection checks, another spatially indexed data structure is used. Unlike in the polygon processing step (Section 3.1.5), where a k-d tree was constructed, an R-tree is queried for all wall edges in the routing edge candidate's bounding box, and the geometric check is only performed against this small set of wall edge geometries.

An R-Tree [79] is a binary search tree for spatial data. It stores values in leaf nodes, with a preset capacity of values per leaf. When the capacity is exceeded, a new branch is created in the tree. Other nodes of three, on the other hand, store bounding boxes of their children. When inserting a new value, the tree is traversed, searching for the leaf where after the insertion, its bounding box would be expanded the least. A great strength of R-tree is the ability to query by range. In the case of wall postprocessing, the tree is queried using the bounding rectangle of an edge, but R-trees support querying by any object that supports the *contains* and *intersects*

```
    wallNodes := all wall nodes on level L
    wallEdgeStars := map of wallNodes indices
                     to sets of coordinates of neighbouring wall

    for each wallNode in wallNodes:
        if wallNode is a wall opening:
            // wall openings are accessible from both sides
            continue
        walEdgeStar := wallEdgeStars[wallNode.index]
                       ordered by polar angle from wallNode
        if walEdgeStar only contains a single edge:
            // end of a wall in space is routable
            continue
        for each pair (l, r) in walEdgeStar:
            bisectorAngle := (l, wallNode, r) angle bisector
            // create a new node that is just inside the room
            newNode := wallNode moved a tiny bit along the bisector
            add newNode to the graph
            for each oldEdge in edges connected to wallNode:
                edgeCandidate := oldEdge with wallNode replaced by newNode
                if edgeCandidate intersects a wall:
                    continue
                if edgeCandidate comes from a polygon but no longer intersects it:
                    // this prevents edges from getting moved to another room
                    continue
                add edgeCandidate to the graph
        remove original edges connected to wallNode from the graph
    for each edge on level L:
        if the interior of edge intersects any wallEdge:
            // this check handles routing edges
            // that intersect a wall in the middle of a polygon
            remove edge from the graph
```

Code listing 3 Wall postprocessing algorithm pseudocode

operations. Range lookups are fast since branches whose bounding box the queried object neither contains nor intersects can be skipped.

3.1.7 Extracting extra information for routing

When looking for a route between two rooms in a building on different floors, users might want to place constraints on the route. In particular, they are interested in vertical passages: stairs and elevators.

The solution I implemented is quite simple. During element processing, each item's tags are inspected and converted into a set of *flags* on the routing graph edges: `IsStairs`, `IsElevator` and `IsEscalator`.

- Elements with the `highway=elevator` tag are marked with the `IsElevator` flag.
- Elements with the `highway=steps` or `stairs=yes` tags are either marked with `IsStairs` or `IsEscalator`, based on the value of the `conveying=*` tag.

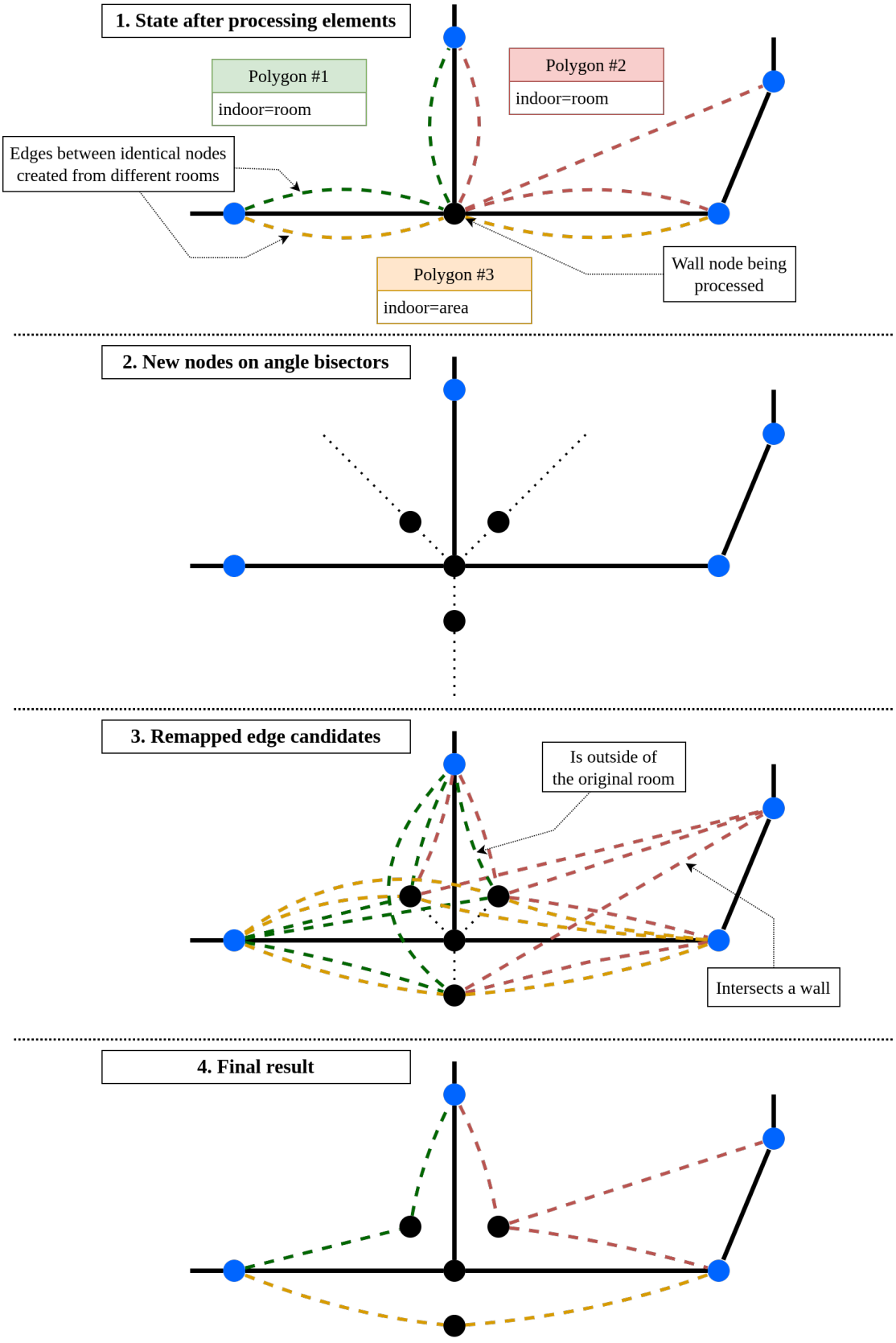

Figure 3.5 Wall postprocessing that disconnects subgraphs in neighbouring rooms

The routing API then accepts optional parameters `disableStairs`, `disableEscalators` and `disableElevators`, which are reflected in the SQL query for selecting edges passed to pgRouting. As a result, the pathfinding algorithm then operates on a graph with the matching edges removed.

3.1.8 Final graph cleanup

Groups of edges that form "islands" in the routing graph are an issue for all routing engines. Formally, these subgraphs are called *connected components* of the routing graph. OpenStreetMap editors try to prevent components of `highway=*` ways from being created by issuing a warning[4]. The issue is even more prevalent in the routing graph created using the steps described in previous sections. Routing nodes are added based on points, with the hope of some other element getting connected to each of them. Understandably, that does not always happen, creating a necessity for a cleanup step. Points are not the only issue, though — polygons with the `indoor=room` tag without any wall openings cause a connected component to be created inside that room. When the user selects a point inside that room, they would be surprised by the inability to find a path to other parts of the map, since they are unaware of the fact there is no door in the source data.

The *graph cleanup* step has two stages, and runs after the graph has been saved into the database, but before its version has been marked as active. First, the pgRouting function `pgr_connectedComponents` (which is implemented as a depth-first-search-based algorithm) is used to find all connected components. The number of nodes in each component is counted using the SQL `GROUP BY` clause, and compared to the total number of nodes in the routing graph. Edges in components smaller than 10% of the graph are then removed.

Then, "deserted" nodes are cleaned up. The operation is simple: delete all nodes where no edge exists that leads from or to this node. This ensures that the user cannot request routing from a point that is not connected to the rest of the graph.

3.2 Client

The client is a Svelte application, acting as a wrapper around a map rendered using the MapLibre library. It communicates with a tile server delivering vector tiles and renders indoor data using the IndoorEqual service, which serves tiles for a second layer with indoor data only. Both the start/target selection dialog and the other routing UI elements are implemented as a *map control* for the Maplibre `Map` instance.

The application relies on the backend, which provides a JSON API for routing. Figure 3.9 describes the communication during the entire process of finding a route.

1. The user navigates to the URL the application is served at. On the server, the client application requests the current *config* from the routing API. This config contains the bounding box of the map, the identifier of the latest version of the routing graph, and a set of *flags*, which carry information about available route options.

2. In the user's browser, a map is rendered. When zooming and panning, the map communicates with two data sources: a tile server for the base map, and the IndoorEqual service. On top of the vector base map, the IndoorEqual API is used to render an indoor map. The switcher control on the map is IndoorIqual also provided by IndoorEqual — not by the API, but as a plugin for MapLibre.

3. The user is presented with two buttons for start and target. Clicking one of them triggers the *point selection* mode, and they can click on the map to select one of the two route points. Once the user selects coordinates for a route point, the client application sends a request to

[4] iD uses a simple breadth-first-search approach: `https://github.com/openstreetmap/iD/blob/develop/modules/validations/disconnected_way.js`

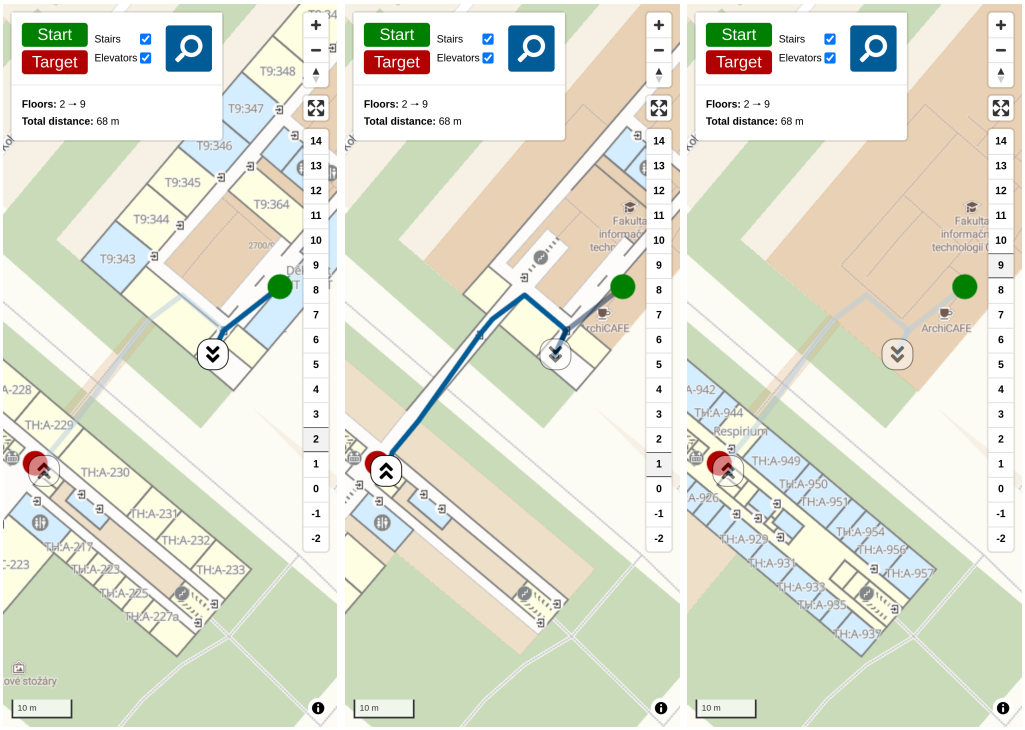

Figure 3.6 The application displaying routing results with the default settings on a mobile device screen size.

the routing API to find the routing node closest to the selected coordinates, on the currently selected floor. Additionally, they can toggle checkboxes for route flags to disable features. Only checkboxes for features present in the routing graph are available — if there are, for example, no escalators, the checkbox for escalators is not shown at all.

4. Once both start and target are selected, the "Find route" button becomes active. Clicking the button triggers a request to the routing API containing the IDs of the start and target nodes, the graph version, and edge types that the user marked as excluded (e.g., stairs).

5. The routing API either responds with a route (a list of nodes along with total length), or with an HTTP 404 error, meaning that no route was found.

6. If a route was found, it is rendered on the map. The client divides the route into segments, each segment being a sequence of nodes on the same level. Then, it switches to the level the start node is located on and the segments are drawn on the map, with the current level highlighted. Nodes that connect levels are marked with "Up" or "Down" icons, which can be used to quickly move to the level of the next segment.

The user interface is designed to be as simple as possible, especially on mobile devices. In Figure 3.6, we can see the interface of the application after selecting routing points on different levels in the CTU T9 and CTU A buildings and searching for a route between them. In Figure 3.7, we can see a route between the same points, but routing through stairs had been disabled before the search, causing the route to take a "detour" to the elevators. Figure 3.8 shows the same route on a desktop screen size.

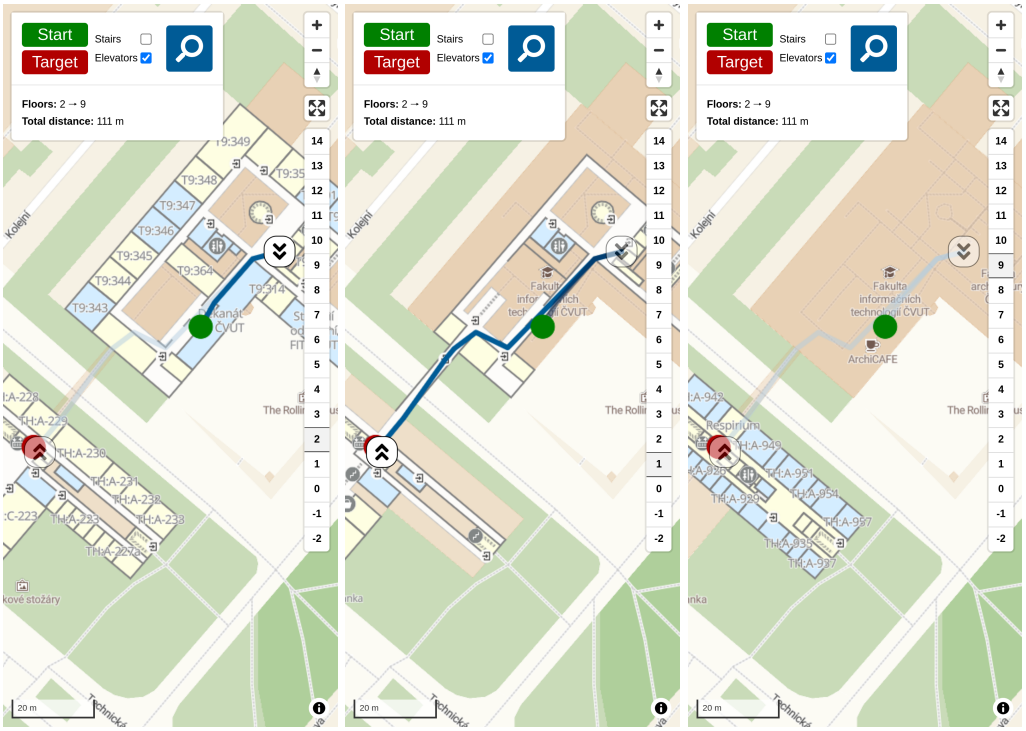

Figure 3.7 The application displaying routing results with the default settings on a mobile device screen size with disallowed stairs.

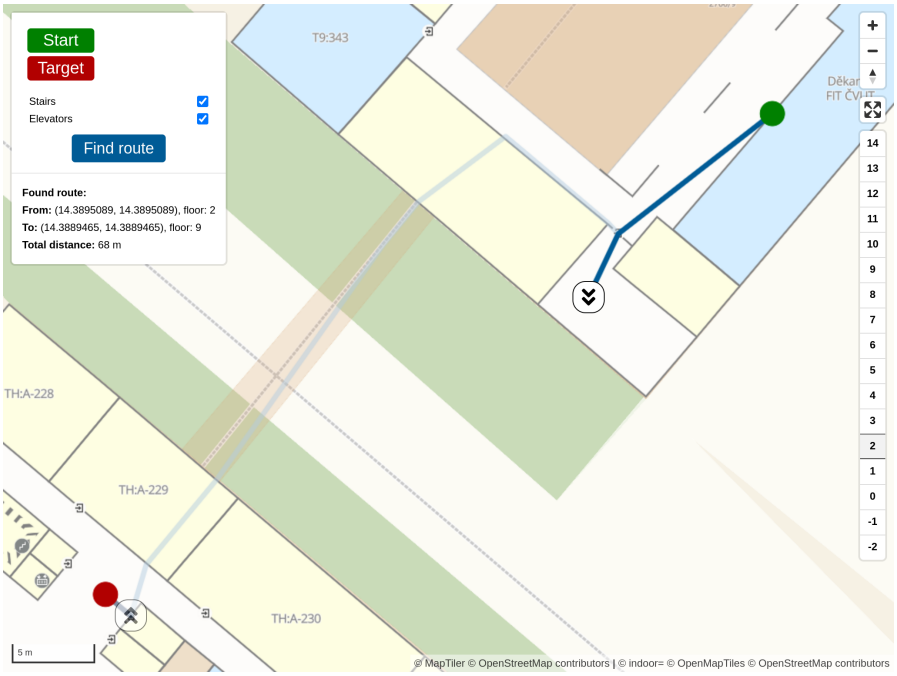

Figure 3.8 The application displaying a routing result on a desktop screen size.

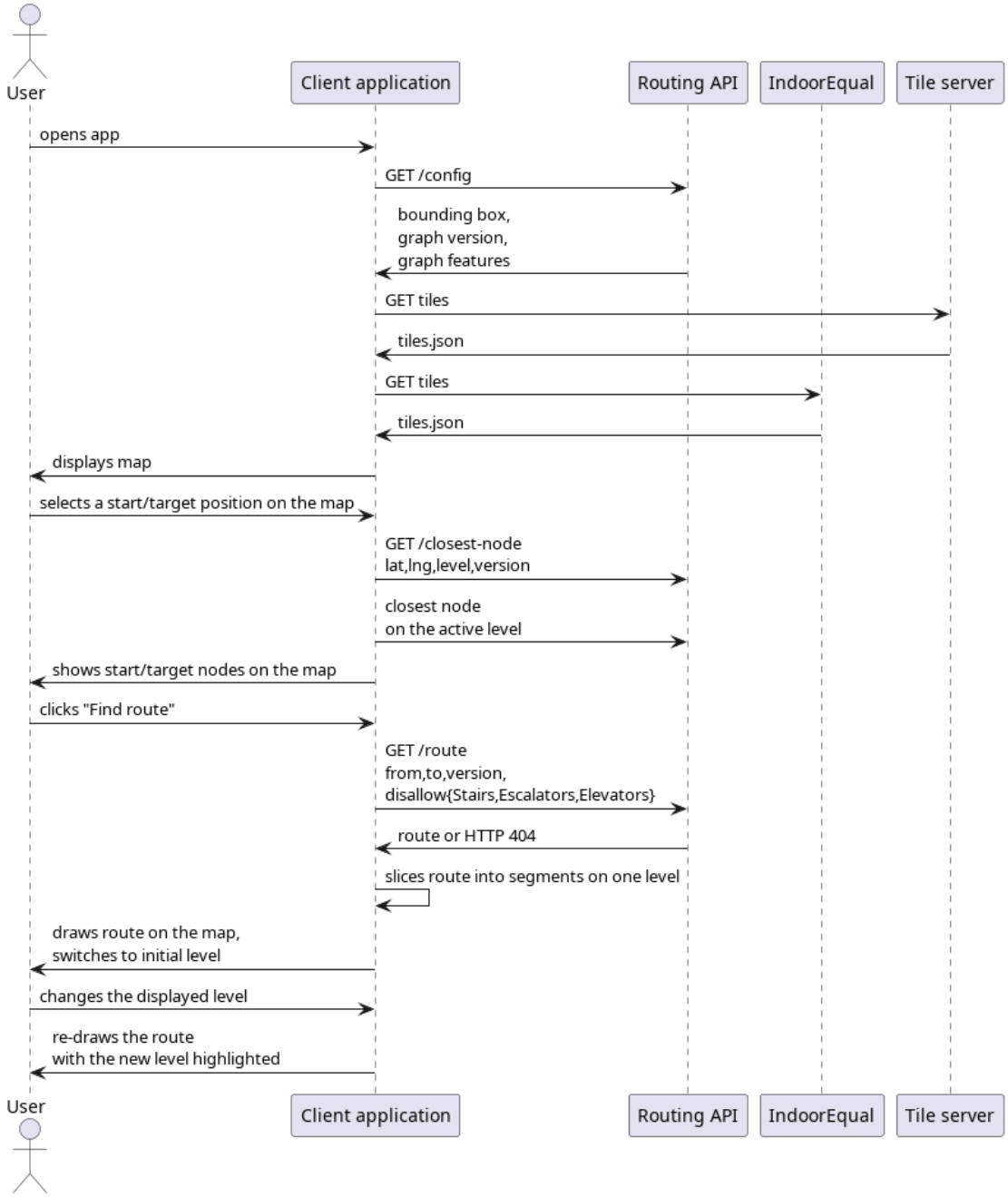

Figure 3.9 Sequence diagram of a user finding a route in the application.

```
public static TheoryData<string, IEnumerable<decimal>> ParsingData =>
    new()
    {
        { "", new[] { 0m } },
        { "0", new[] { 0m } },
        { "G", new[] { 0m } },
        { "1", new[] { 1m } },
        { "1.0", new[] { 1m } },
        { "+1", new[] { 1m } },
        { "2", new[] { 2m } },
        { "-1", new[] { -1m } },
        { "0;1", new[] { 0m, 1m } },
        { "-1;2;5", new[] { -1m, 2m, 5m } },
        { "2-4", new[] { 2m, 3m, 4m } },
        { "0;2-3", new[] { 0m, 2m, 3m } },
        { "-1-1", new[] { -1m, 0m, 1m } },
        { "-3--1", new[] { -3m, -2m, -1m } },
    };

[Theory]
[MemberData(nameof(ParsingData))]
public void Parses(string input, IEnumerable<decimal> expected) =>
    new LevelParser(Mock.Of<ILogger<LevelParser>>())
        .Parse(input)
        .Should()
        .BeEquivalentTo(expected);
```

Code listing 4 An example from the test suite (with some cases removed for conciseness).

3.3 Unit testing and CI/CD

During development, unit tests were written along with the code. The test suite contains about 100 tests, which cover important parts of the application. They have proven very useful in preventing regressions when introducing new kinds of element processing.

The test suite uses four main testing libraries:

- *xUnit* [80], a testing framework. XUnit uses the concepts of *facts* for simple test cases and *theories* for parametrised tests — tests that are run multiple times with different inputs.
- *Fluent Assertions* [81], a library for writing readable assertions. This library allows writing self-documenting tests by exposing an API that reads like a sentence.
- *Moq* [82], the most popular .NET mocking library.
- *Testcontainers* [83], a library that starts Docker containers from code. In the test suite, I used it to start a PostgreSQL database for integration and database layer tests. Tests are then run against the database running in a container, which is destroyed after the tests finish.

In Listing 4, we can see a test taken from the test suite which used xUnit, FluentAssertions and Moq. The unit under test is `LevelParser`, a class which handles the parsing of the `level=*` OSM tag values.

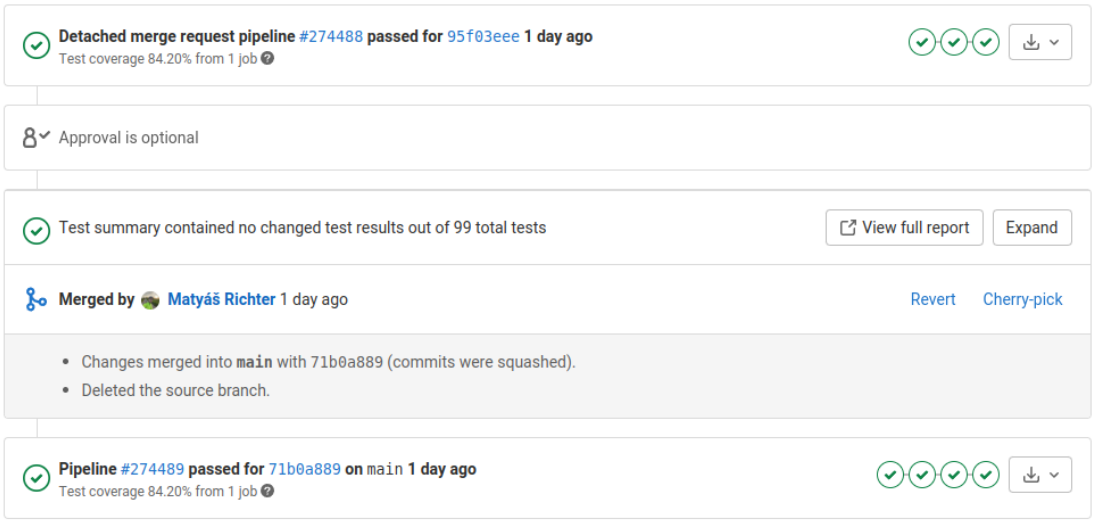

Figure 3.10 Example of GitLab pipelines overview for a merged Merge request. [84]

3.3.1 GitLab Pipelines

The Git repository for the project is hosted on the faculty instance of GitLab. On every push to a branch with an open *Merge request*, a GitLab pipeline [84] is triggered, which executes the following stages:

1. Code formatting check using the *csharpier* tool. The repository also contains a *pre-commit hook*, a script which runs the formatting tool before committing, which keeps the codebase consistent.
2. Build stage, where dependencies are fetched and it is verified that the application compiles without errors.
3. Test stage, where unit tests are run.

On the default branch, `main`, two additional stages are run on each commit:

1. Docker build stage, which builds Docker images for the application and pushes them to the GitLab container registry.
2. Deploy stage, where a new version of the application is deployed to a Docker Swarm environment, serving as a live demo for the project.

Chapter 4

Outcomes of user testing and implications for future work

4.1 User testing

User testing was conducted using a deployment of the application with a bounding box of `(N 50.100700, E 14.386007; N 50.105917, E 14.395190)`, which contains the area of the university campus in Dejvice, Prague. In terms of indoor areas coverage at the time of testing, OpenStreetMap data in the area contained maps for the ground floor of the Czech National Library of Technology, the ground and second floors of the Czech Technical University (CTU) T9 building, and several floors in the CTU building A. Users were not given an exact script to follow in order to simulate their first impression of the application. Rather asked vaguely to pick points of their choice and find a route between them. The users were not familiar with any technical details of the application, such as the fact that the data comes from OpenStreetMap. Naturally, the users followed the steps of Use Case 1 Section 2.1.4.

Testing sessions were held with four users in total. Two of them were students at the Faculty of Information Technology at CTU and were therefore already familiar with the area. The other two users had never visited the area before. In terms of devices, one user in each category accessed the application using a mobile device with a small screen (a smartphone), one used a desktop computer with an ultrawide monitor, and one used a laptop with a Full HD screen.

During user testing, the following issues and suggestions were collected:

1. When selecting floor 2, the user picked the start point on that floor and tried to pick a target point outside. The application snapped their target point to the closest routing node on the second floor, far away from the location they picked. The user expected to be able to select a point outside of the building without switching back to level 0.

2. A user who is familiar with the area was confused by the presented floor numbering. In the CTU T9 building, floor 2 is labelled as floor 3, while the application uses zero-based indexing from OSM.

3. The user who had a wide-screen monitor found it hard to locate map controls since they were pushed far to the edges of their screen.

4. One user suggested that the application should visualise areas that are not properly mapped, which would help them understand why trying to pick a point in an unmapped building failed.

Figure 4.1 A bug where the application suggests a route through a wall.

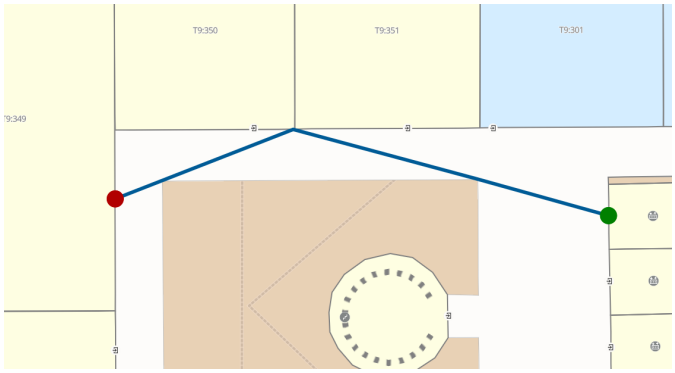

Figure 4.2 A bug where the application fails to find a route along a wall.

5. For one of the mobile users, the floor switching control took a long time to load, even though they were on a stable internet connection.

6. A bug was discovered, where the application would find a route through the walls of a room on level 2 in the CTU T9 building. See Figure 4.1.

7. A bug was discovered, where the application would find a route that "ricochets" between nodes on opposite sides of a corridor instead of following its wall. See Figure 4.2.

4.2 Future work

In its current state, the application can be considered a working prototype. Before it can be deployed to a production environment, the points raised during user testing need to be addressed, and unfulfilled requirements implemented. The following list summarises them. Issues from testing are labelled I1, I2, …, original requirementsSection 2.1.1 are labelled F1, F2,…

1. A solution for I1 could be to only allow snapping to a point up to, for instance, 10 meters away from the clicked location. Additionally, the API could return a point on the ground level if no result was found within that radius on a higher level.

2. Floor numbering (I2) will be more complex. Simple Indoor Taggings does support adding names to levels, but support for displaying those names would need to be implemented in the IndoorEqual floor switching plugin.

3. I consider the issue with large screen sizes (I3) to be a low priority since the primary target audience of the application are mobile users.

4. I4 is an interesting suggestion. After graph processing, the application could generate a heatmap of nodes for each level, and display an inverted version as a map layer, hiding areas without routing nodes. A potential problem comes from areas that do not contain any nodes by design, such as parks and bodies of water, because those should not be hidden.

5. I5 should hopefully be resolvable by self-hosting IndoorEqual instead of waiting for the response from the public instance, which can sometimes take a long time. In my testing, the response time from the public API was not consistent across requests, which is probably caused by varying load on IndoorEqual's servers.

6. The two bugs, I5 and I6, need to be fixed. From my inspection of the OSM map, the area seems to be mapped correctly, so they are likely caused by an issue in graph processing.

7. To better reach the audience of users who are not familiar with the area at all, search functionality should be implemented (F1.3, originally marked COULD HAVE). I consider this to be a high-priority feature.

8. Implementing requirement F2.4 (Data Enhancement, WILL NOT HAVE) would allow the application to have a feature similar to iQNavs [17, 18], a network of QR codes around the building mapped to routing nodes. In my opinion, the solution implemented in iQNavs is a very elegant solution for indoor positioning.

I intend to keep working on the application in the future, focusing on the steps outlined above.

Chapter 5
Conclusion

In this Bachelor's thesis, I created an application for finding multi-floor routes based on OpenStreetMap data. I surveyed existing indoor navigation technologies and found that they are either proprietary or based on a custom data format. Then, I introduced the OpenStreetMap project and its specifics, including the support for indoor maps. I analyzed the requirements for the application and selected appropriate technologies for implementation. I became more familiar with geospatial concepts and introduced them to the reader.

Then, I implemented an indoor routing application. The application has the ability to download and import OpenStreetMap data extracts and process them into a routing graph. The main challenges I faced when implementing the solution were level connections and routing in areas bounded by impassable linear barriers. I also implemented a user interface with an interactive map, which allows the user to search for a route between two points and view the result. In the source code, I included a test suite and set up a CI pipeline for automatic tests.

All requirements implied by the thesis assignment were fulfilled, and the result is a functional prototype, which I intend to further develop in the future. I included a list of minor shortcomings of the solution and options for its extension and improvement.

Bibliography

1. UNITED TIRES. *Study Reveals Where Drivers Are Most Reliant on Their GPS* [online]. 2022. [visited on 2023-04-02]. Available from: https://archive.is/VL268.
2. *Figures about the Burj Khalifa* [online]. [visited on 2023-04-03]. Available from: https://www.burjkhalifa.ae/en/the-tower/facts-figures/.
3. WIKIDATA. *Judge Harry Pregerson Interchange* [online]. 2021. [visited on 2023-04-03]. Available from: https://www.wikidata.org/w/index.php?title=Q806003&oldid=1396703579.
4. SAUMELL, Maria. *Algorithms and graphs 1 lecture slides* [online]. Faculty of Information Technology, Czech Technical University in Prague, 2022.
5. MCQUIRE, Scott. One map to rule them all? Google Maps as digital technical object. *Communication and the Public*. 2019. Available from DOI: 10.1177/2057047319850192.
6. GOOGLE LLC. *Indoor Maps* [online]. [visited on 2023-04-22]. Available from: https://www.google.com/maps/about/partners/indoormaps/.
7. ARTHUR, Charles. Apple Maps: Tim Cook says he is 'extremely sorry'. *The Guardian* [online]. 2012 [visited on 2023-04-22]. Available from: https://www.theguardian.com/technology/2012/sep/28/apple-maps-tim-cook-apology.
8. PETERS, Jay. Apple Maps turns 10 —and it's finally worth using - The Verge. *The Verge* [online]. 2022 [visited on 2023-04-23]. Available from: https://www.theverge.com/23323550/apple-maps-10-year-anniversary-iphone-google.
9. VONAU, Manuel. Apple Maps is good now, and that's a big problem for Google. *Android Police* [online]. 2023 [visited on 2023-04-23]. Available from: https://www.androidpolice.com/apple-maps-good-now-problem-google-maps/.
10. OPEN GEOSPATIAL CONSORTIUM. *OGC membership approves new Community Standard: Indoor Mapping Data Format (IMDF)* [online]. 2021. [visited on 2023-04-23]. Available from: https://www.ogc.org/press-release/ogc-membership-approves-new-community-standard-indoor-mapping-data-format-imdf/.
11. OPEN GEOSPATIAL CONSORTIUM. *Indoor Mapping Data Format (1.0.0)* [online]. 2021. [visited on 2023-04-23]. Available from: https://docs.ogc.org/cs/20-094/index.html.
12. LEE, Jiyeong; LI, Ki-Joune; ZLATANOVA, Sisi; KOLBE, Thomas H.; NAGEL, Claus; BECKER, Thomas; KANG, Hye-Young. *IndoorGML 1.1* [online]. 2020-11. [visited on 2023-04-23]. Available from: http://www.opengis.net/doc/IS/indoorgml/1.1.
13. RYOO, Hyung-Gyu; KIM, Taehoon; LI, Ki-Joune. Comparison between two OGC standards for indoor space. In: *Proceedings of the Seventh ACM SIGSPATIAL International Workshop on Indoor Spatial Awareness*. ACM, 2015. Available from DOI: 10.1145/2834812.2834813.

14. *Steerpath* [online]. 2023. [visited on 2023-04-23]. Available from: https://www.steerpath.com/.
15. *Mapstead* [online]. 2023. [visited on 2023-04-23]. Available from: https://mapsted.com/indoor-navigation/#.
16. *INDRZ* [online]. 2023. [visited on 2023-04-23]. Available from: https://www.indrz.com/.
17. BURKOŇ, Richard. *Frontend iQR navigačního systému*. 2022. Available also from: http://hdl.handle.net/10467/99085. MA thesis. Czech Technical University in Prague.
18. KREJČÍ, David. *Backend iQR navigačního systému*. 2022. Available also from: http://hdl.handle.net/10467/103758. Bachelor's Thesis. Czech Technical University in Prague.
19. OPENSTREETMAP WIKI CONTRIBUTORS. *About OpenStreetMap* [online]. OpenStreetMap Wiki, 2022 [visited on 2023-03-06]. Available from: https://wiki.openstreetmap.org/w/index.php?title=About_OpenStreetMap&oldid=2310396.
20. OPENSTREETMAP WIKI CONTRIBUTORS. *History of OpenStreetMap*. OpenStreetMap Wiki, 2023. Available also from: https://wiki.openstreetmap.org/w/index.php?title=History_of_OpenStreetMap&oldid=2475549. [Online; accessed 19-April-2023].
21. ARTHUR, Charles. How free will Ordnance Survey's maps be? Your last chance to decide. *The Guardian*. 2010. ISSN 0261-3077. Available also from: https://www.theguardian.com/technology/blog/2010/mar/17/ordnance-survey-consultation-ending.
22. JWALSH. *Response to the consultation on opening access to Ordnance Survey data –Open Knowledge Foundation blog* [online]. 2010-03. [visited on 2023-03-06]. Available from: https://blog.okfn.org/2010/03/15/response-to-the-consultation-on-opening-access-to-ordnance-survey-data/.
23. *Locus Association's response to the Ordnance Survey Consultation* [online]. 2010-03. [visited on 2023-03-06]. Available from: https://data.europa.eu/sites/default/files/report/2010_locus_associations_response_to_the_ordnance_survey_consultation.pdf.
24. COAST, Steve. *Microsoft Imagery details | OpenStreetMap Blog*. 2010. Available also from: https://blog.openstreetmap.org/2010/11/30/microsoft-imagery-details/.
25. MICROSOFT. *Bing Maps Adds OpenStreetMap*. 2010. Available also from: https://blogs.bing.com/maps/2010/08/02/bing-maps-adds-openstreetmap.
26. OPENSTREETMAP FOUNDATION [online]. Openstreetmap.org, 2023 [visited on 2023-03-07]. Available from: https://welcome.openstreetmap.org/about-osm-community/consumers/.
27. HUMANITARIAN OPENSTREETMAP TEAM. *What we do* [online]. Hotosm.org, 2023 [visited on 2023-03-07]. Available from: https://www.hotosm.org/what-we-do.
28. OPENSTREETMAP FOUNDATION. *About* [online]. 2023. [visited on 2023-03-06]. Available from: https://blog.openstreetmap.org/about/.
29. OPENSTREETMAP WIKI CONTRIBUTORS. *Foundation* [online]. OpenStreetMap Wiki, 2022 [visited on 2023-03-06]. Available from: https://wiki.openstreetmap.org/w/index.php?title=Foundation&oldid=2450475.
30. OPENSTREETMAP FOUNDATION. *Local Chapters and Communities Working Group — OpenStreetMap Foundation* [online]. 2023. [visited on 2023-03-06]. Available from: https://wiki.osmfoundation.org/w/index.php?title=Local_Chapters_and_Communities_Working_Group&oldid=10374.
31. OPENSTREETMAP FOUNDATION CONTRIBUTORS. *Why CC BY-SA is Unsuitable* [online]. OpenStreetMap Foundation, 2016-07 [visited on 2023-03-07]. Available from: https://wiki.osmfoundation.org/w/index.php?title=Licence_and_Legal_FAQ/Why_CC_BY-SA_is_Unsuitable&oldid=3927.

32. OPENSTREETMAP FOUNDATION CONTRIBUTORS. *We Are Changing The License* [online]. OpenStreetMap Foundation, 2016-03 [visited on 2023-03-07]. Available from: https://wiki.osmfoundation.org/w/index.php?title=Licence/Historic/We_Are_Changing_The_License&oldid=4141.
33. OPENSTREETMAP FOUNDATION CONTRIBUTORS. *Licence* [online]. OpenStreetMap Foundation, 2021-08 [visited on 2023-03-07]. Available from: https://wiki.osmfoundation.org/w/index.php?title=Licence&oldid=8605.
34. OPENSTREETMAP WIKI CONTRIBUTORS. *Elements* [online]. OpenStreetMap Wiki, 2023 [visited on 2023-03-11]. Available from: https://wiki.openstreetmap.org/w/index.php?title=Elements&oldid=2459386.
35. OPENSTREETMAP CONTRIBUTORS. *taginfo* [online]. 2023. [visited on 2023-05-06]. Available from: https://taginfo.openstreetmap.org/.
36. OPENSTREETMAP FOUNDATION. *OpenStreetMap Statistics* [online]. [visited on 2023-03-11]. Available from: https://planet.openstreetmap.org/statistics/data_stats.html.
37. OPENSTREETMAP CONTRIBUTORS. *OpenStreetMap* [online]. 2023. [visited on 2023-05-06]. Available from: https://www.openstreetmap.org/.
38. OPENSTREETMAP WIKI CONTRIBUTORS. *JOSM* [online]. OpenStreetMap Wiki, 2023 [visited on 2023-04-16]. Available from: https://wiki.openstreetmap.org/w/index.php?title=JOSM&oldid=2499791.
39. OPENSTREETMAP WIKI CONTRIBUTORS. *ID* [online]. OpenStreetMap Wiki, 2023 [visited on 2023-04-16]. Available from: https://wiki.openstreetmap.org/w/index.php?title=ID&oldid=2495702.
40. OPENSTREETMAP WIKI CONTRIBUTORS. *Rapid* [online]. OpenStreetMap Wiki, 2023 [visited on 2023-04-16]. Available from: https://wiki.openstreetmap.org/w/index.php?title=Rapid&oldid=2502687.
41. OPENSTREETMAP WIKI CONTRIBUTORS. *StreetComplete* [online]. OpenStreetMap Wiki, 2023 [visited on 2023-04-16]. Available from: https://wiki.openstreetmap.org/w/index.php?title=StreetComplete&oldid=2483306.
42. OPENSTREETMAP WIKI CONTRIBUTORS. *Micromapping* [online]. OpenStreetMap Wiki, 2023 [visited on 2023-04-14]. Available from: https://wiki.openstreetmap.org/w/index.php?title=Micromapping&oldid=2496545.
43. OPENSTREETMAP WIKI CONTRIBUTORS. *Indoor Mapping* [online]. OpenStreetMap Wiki, 2023 [visited on 2023-04-14]. Available from: https://wiki.openstreetmap.org/w/index.php?title=Indoor_Mapping&oldid=2499378.
44. OPENSTREETMAP WIKI CONTRIBUTORS. *Proposed features/IndoorOSM* [online]. OpenStreetMap Wiki, 2022 [visited on 2023-04-15]. Available from: https://wiki.openstreetmap.org/w/index.php?title=Proposed_features/IndoorOSM&oldid=2299157.
45. OPENSTREETMAP WIKI CONTRIBUTORS. *Proposed features/CompoundFacility* [online]. OpenStreetMap Wiki, 2018 [visited on 2023-04-15]. Available from: https://wiki.openstreetmap.org/w/index.php?title=Proposed_features/CompoundFacility&oldid=1599475.
46. OPENSTREETMAP WIKI CONTRIBUTORS. *Termite* [online]. OpenStreetMap Wiki, 2016 [visited on 2023-04-15]. Available from: https://wiki.openstreetmap.org/w/index.php?title=Termite&oldid=1321283.
47. OPENSTREETMAP WIKI CONTRIBUTORS. *F3DB* [online]. OpenStreetMap Wiki, 2021 [visited on 2023-04-15]. Available from: https://wiki.openstreetmap.org/w/index.php?title=F3DB&oldid=2191286.

48. OPENSTREETMAP WIKI CONTRIBUTORS. *Simple Indoor Tagging* [online]. OpenStreetMap Wiki, 2023 [visited on 2023-04-14]. Available from: https://wiki.openstreetmap.org/w/index.php?title=Simple_Indoor_Tagging&oldid=2478681.
49. ZBYTOVSKÝ, Pavel. *Indoor mapping on the OpenStreetMap platform.* Prague, 2016. MA thesis. Czech Technical University in Prague, Faculty of Information Technology.
50. PAVIE, Adrien. *iD-indoor source repository.* Available also from: https://framagit.org/PanierAvide/iD-indoor. [Online; accessed 16-April-2023].
51. OPENSTREETMAP WIKI CONTRIBUTORS. *JOSM/Plugins/indoorhelper* [online]. OpenStreetMap Wiki, 2022 [visited on 2023-04-16]. Available from: https://wiki.openstreetmap.org/w/index.php?title=JOSM/Plugins/indoorhelper&oldid=2263317.
52. PAVIE, Adrien. *OsmInEdit source repository.* Available also from: https://framagit.org/PanierAvide/osminedit. [Online; accessed 16-April-2023].
53. OPENSTREETMAP WIKI CONTRIBUTORS. *Rendering* [online]. OpenStreetMap Wiki, 2023-02-08 [visited on 2023-05-06]. Available from: https://wiki.openstreetmap.org/w/index.php?title=Rendering&oldid=2475960.
54. PAVIE, Adrien. *OpenLevelUp* [comp. software]. 2023. [visited on 2023-05-06]. Available from: https://openlevelup.net/.
55. METZ, François de. *indoor=* [online]. 2023. [visited on 2023-05-06]. Available from: https://indoorequal.org/.
56. PRODUCTPLAN. *MoSCoW Prioritization* [online]. 2023. [visited on 2023-05-07]. Available from: https://www.productplan.com/glossary/moscow-prioritization/.
57. MICROSOFT. *.NET (and .NET Core) - introduction and overview* [online]. [visited on 2023-04-25]. Available from: https://learn.microsoft.com/en-us/dotnet/core/introduction.
58. SVELTE CONTRIBUTORS. *Svelte Documentation* [online]. 2023. [visited on 2023-05-06]. Available from: https://svelte.dev/docs.
59. MAPLIBRE. *MapLibre GL JS Documentation* [online]. 2023. [visited on 2023-05-06]. Available from: https://maplibre.org/maplibre-gl-js-docs/api/.
60. THE POSTGRESQL GLOBAL DEVELOPMENT GROUP. *PostgreSQL 15.2 Documentation* [online]. 2023. [visited on 2023-05-06]. Available from: https://www.postgresql.org/docs/.
61. RAMSEY, Paul; LESLIE, Mark; POSTGIS CONTRIBUTORS. *Introduction to PostGIS* [online]. 2023. [visited on 2023-05-06]. Available from: https://postgis.net/workshops/postgis-intro/index.html.
62. PGROUTING CONTRIBUTORS. *pgRouting documentation* [online]. 2023. [visited on 2023-04-25]. Available from: https://docs.pgrouting.org/latest/en/index.html.
63. SMITH, Heather. *Geographic vs Projected Coordinate Systems* [online]. 2020-02-27. [visited on 2023-05-06]. Available from: https://www.esri.com/arcgis-blog/products/arcgis-pro/mapping/gcs_vs_pcs/.
64. ESRI. *ArcGIS Developer Glossary* [online]. 2023. [visited on 2023-05-06]. Available from: https://developers.arcgis.com/documentation/glossary/.
65. OPENSTREETMAP WIKI CONTRIBUTORS. *Projection* [online]. OpenStreetMap Wiki, 2022-08 [visited on 2023-05-06]. Available from: https://wiki.openstreetmap.org/w/index.php?title=Projection&oldid=2371538.
66. OPENSTREETMAP WIKI CONTRIBUTORS. *Web Mercator* [online]. OpenStreetMap Wiki, 2022-08-13 [visited on 2023-05-06]. Available from: https://wiki.openstreetmap.org/w/index.php?title=Web_Mercator&oldid=2371529.

67. KETTLE, Simon. Distance on a Sphere: the Haversine Formula. *Esri Community Blog* [online]. 2017 [visited on 2023-05-06]. Available from: https://community.esri.com/t5/coordinate-reference-systems-blog/distance-on-a-sphere-the-haversine-formula/ba-p/902128.
68. AGAFONKIN, Vladimir. Fast geodesic approximations with Cheap Ruler. *maps for developers* [online]. 2016 [visited on 2023-05-06]. Available from: https://blog.mapbox.com/fast-geodesic-approximations-with-cheap-ruler-106f229ad016.
69. VENESS, Chris. Vincenty solutions of geodesics on the ellipsoid. *Movable Type Scripts* [online]. 2022 [visited on 2023-05-06]. Available from: https://www.movable-type.co.uk/scripts/latlong-vincenty.html.
70. NETTOPOLOGYSUITE CONTRIBUTORS. *NetTopologySuite Documentation* [online]. 2022. [visited on 2023-05-06]. Available from: https://nettopologysuite.github.io/NetTopologySuite/index.html.
71. TOEWS, M. W.; RAMSEY, Paul; LESLIE, Mark; POSTGIS CONTRIBUTORS. *DE-9IM* [online]. 2015-07-24. [visited on 2023-05-06]. Available from: https://en.wikipedia.org/w/index.php?title=DE-9IM&oldid=1090012465.
72. KARICH, Peter. *Now flexible routing is at least 15 times faster* [online]. 2017-08-14. [visited on 2023-05-07]. Available from: https://www.graphhopper.com/blog/2017/08/14/flexible-routing-15-times-faster/.
73. SURYNEK, Pavel; ŘEHOŘEK, Tomáš. *Artificial Intelligence Fundamentals lecture slides* [online]. Faculty of Information Technology, Czech Technical University in Prague, 2022.
74. *Osm2pgsql Manual* [online]. 2023. [visited on 2023-05-02]. Available from: https://osm2pgsql.org/doc/manual.html.
75. *Route through squares (area=yes)* [online]. 2018. [visited on 2023-05-06]. Available from: https://github.com/Project-OSRM/osrm-backend/issues/64.
76. *Area routing for pedestrian/bike* [online]. 2022. [visited on 2023-05-06]. Available from: https://github.com/graphhopper/graphhopper/issues/82.
77. HAHMANN, Stefan; MIKSCH, Jakob; RESCH, Bernd; LAUER, Johannes; ZIPF, Alexander. Routing through open spaces –A performance comparison of algorithms. *Geo-spatial Information Science*. 2018, vol. 21, no. 3, pp. 247–256. Available from DOI: 10.1080/10095020.2017.1399675.
78. BEREZOVSKÝ, Marko; MAŘÍK, Radek. *Search trees, k-d tree* [online]. Faculty of Electrical Engineering, Czech Technical University in Prague, 2012 [visited on 2023-05-06]. Available from: https://cw.fel.cvut.cz/old/_media/courses/a4m33pal/paska13.pdf.
79. SYPYTKOWSKI, Bartosz. *R-Tree: algorithm for efficient indexing of spatial data* [online]. 2022-04-29. [visited on 2023-05-07]. Available from: https://www.bartoszsypytkowski.com/r-tree/.
80. .NET FOUNDATION. *About xUnit.net* [online]. 2023. [visited on 2023-05-07]. Available from: https://xunit.net/.
81. DOOMEN, Dennis. *About - Fluent Assertions* [online]. 2023. [visited on 2023-05-07]. Available from: https://fluentassertions.com/about/.
82. MOQ CONTRIBUTORS. *moq* [online]. 2023. [visited on 2023-05-07]. Available from: https://github.com/moq/moq4.
83. TESTCONTAINERS CONTRIBUTORS. *Testcontainers for .NET* [online]. 2023. [visited on 2023-05-07]. Available from: https://dotnet.testcontainers.org/.
84. GITLAB. *GitLab CI/CD* [online]. 2023. [visited on 2023-05-07]. Available from: https://docs.gitlab.com/ee/ci/.

Contents of the attached archive

```
repository ..... GIT repository with the implmentation
└── README.md ..... repository information, including steps for running the application
thesis ..... thesis source code in the LaTeX format
```