TUM SCHOOL OF ENGINEERING AND DESIGN

TECHNICAL UNIVERSITY OF MUNICH

Master′s Thesis

Multimodal Navigation Applications for CityGML 3.0 using a Graph Database

Felix Sebastian Olbrich

TUM SCHOOL OF ENGINEERING AND DESIGN

TECHNICAL UNIVERSITY OF MUNICH

Master's Thesis

Multimodal Navigation Applications for CityGML 3.0 using a Graph Database

Multimodale Navigationsanwendungen für CityGML 3.0 mittels einer Graphdatenbank

Author:	Felix Sebastian Olbrich
Supervisor:	Univ.-Prof. Dr. rer. nat. Thomas H. Kolbe
Advisors:	Son H. Nguyen, M.Sc. and Christof Beil, M.Sc.
Study Course:	Geodesy and Geoinformation (Master)
Submission Date:	15th November 2023

2023

This version of the document was subject to editorial revision made after the grading of the thesis.

With this statement I confirm that this master's thesis is my own work and I have documented all sources and material used. The thesis was not previously submitted to another academic institution and has not yet been published elsewhere.

Munich, 15th November 2023 Felix Sebastian Olbrich

Abstract

While the focus of semantic 3D city models has so far been on 3D building models, the increasing availability of detailed representations of the street space allows numerous new applications. The international OGC standard CityGML which is used to model and exchange semantic 3D city models has been improved to cover street spaces. Version 3.0 includes extended concepts for modelling the street space. This thesis investigates how CityGML 3.0 compliant street space models can be used for multimodal navigation applications by mapping them to a graph database. In particular, the concepts required for multimodal navigation are being analyzed. Further, it is analyzed if those apply to CityGML 3.0. The combination of different means of transport in one application and the use of different dimensions in the geometric representation as well as their "level of detail"/granularity also play an essential part here.

Firstly, requirements for multimodal navigation applications are collected. Then, concepts of CityGML 3.0 are compared with these requirements. Missing elements, such as adjacency relationships and weights can be added to a graph representation. This is followed by the pre-processing of the CityGML test data in the graph database Neo4j. Using the graph database, a suitable network for multimodal routing is generated. The final routing network shall connect existing CityGML objects without removing elements or canceling existing relationships. In the following, different weights are added to the newly generated network, for example, length derived from the geometry or speed limits derived from semantic information of CityFurniture objects (street signs). The resulting multimodal network is then used to show that the analysis of shortest paths is independent of the geometric representation and the granularity of the street space objects. For the multimodal navigation application, two CityGML 3.0 test datasets are available (Ingolstadt and Grafing near Munich). To verify the 3D navigation capabilities, a new dataset containing the Grafing network as well as a hand-modelled 3D parking garage building with differing granularity is used. Finally, the routing results are validated.

Street space data from CityGML 3.0 can provide rich information for navigation applications. In addition to the geometric and semantic information, further connections between elements can be used. Furthermore, CityGML allows the modelling and use of different representations of the street space, for example, using the complete road or a lane-based representation. A network structure based on the predecessor and successor relationships of the TrafficSpace objects can be generated independent of the remaining structure and scope of the dataset. Furthermore, the network structure can be improved with additional connections. For querying the data and performing shortest path analyses, the graph database is an efficient platform. Fundamental multimodal navigation functionality could be implemented. However, there is a need for further development, e.g., the integration of additional transportation modes. In a real-world example, the combined usage of different granularities and dimensions of TrafficSpace objects was shown. Vehicle routing from the street to a parking space in the multi-story car park was carried out. It is then possible to continue the route from the parking spot to the footpath network. Findings from this work have also contributed to the further development of the open-source software r:trån.

Contents

1. Introduction

1.1. Motivation and Problem Statement

Mobility, including the transportation of people and goods, is a key factor shaping the development of society. This sector faces many different challenges and includes topics like infrastructure design and traffic flow management as well as day-to-day route planning and navigation applications used by non-specialists. An ordinary person can be assisted by navigation applications to find the best-fitting route for the journey from A to B. Many navigation applications and standards are available. However, they are often restricted in the number of transportation modes they can handle or the constraints they can consider. Recent developments in Germany, such as the introduction of the Germany-ticket ("Deutschlandticket") (Holly & AFP, 2023), which allows the usage of many public transportation providers with a single ticket, the focus on e-mobility and the rise of shared mobility services would profit from introducing ways to combine transportation modes during travel. Additionally, the introduction of digital urban twins and the rise of 3D city models provide more information about the real world digitally. Here, City Geography Markup Language (CityGML), an Open Geospatial Consortium (OGC) standard for representing 3D city models, is used to collect, model and exchange data in a standardized form. CityGML is capable of storing and exchanging semantic 3D city and landscape models including street spaces. With its newest version, CityGML 3.0, the street space modelling has been revised. This raises the question of the extent to which the standard can be used for navigation applications. As CityGML is a standard for 3D city models, it is possible to model multiple mobility infrastructure elements in one dataset. Furthermore, CityGML 3.0 natively supports modelling of 3D objects and relationships between its objects. This makes it possible to model the street space as a graph and allows the usage of graph algorithms to solve shortest path routing problems, whilst giving the possibility to include further information derived from the objects. Especially the usage of additional and highly accurate data like volumetric geometry information provided by 3D objects could open up new possibilities for navigation applications. Thus, developing a navigation application based on CityGML seems to be a promising approach. However, as CityGML is a modelling standard, it does not include any routing functionality. Therefore, a graph database is used to store the CityGML data and to perform basic routing queries.

Current navigation applications are already capable of solving complex routing tasks. Thus, this thesis does not aim to reinvent the core functionalities but rather aims to explore the possibilities and challenges of using CityGML 3.0 for providing information for solving or assisting in solving navigation-related problems. Additionally, the thesis will show the usage of graph databases as a backend for navigation applications and examine the combination of multiple transportation modes in one dataset to investigate multimodal routing problems.

1.2. Research Questions and Objectives

The following research questions are addressed in this thesis:

- Utilizing CityGML as a Data Source for Navigation Applications
    1. How can CityGML 3.0 be used to solve (multimodal) navigation problems
        - Which parts of the CityGML standard are needed for navigation applications?
        - What level of detail is needed to solve navigation problems?
        - Which semantic and geometric data can be utilized to improve navigation results (e.g., street furniture)?
        - How can a navigation application benefit from 3D data provided by CityGML (e.g., utilizing the volumetric information to use the clearance spaces over the street for big cargo transport routing, or modelling of indoor spaces, and parking garages)?
    2. How can the CityGML standard be improved regarding (multimodal) navigation applications?
        - What elements are identified as missing and would improve the usability of the standard for navigation applications?
        - If there is a need to change the CityGML standard for multimodal navigation applications, how can the standard be improved? Including an Application Domain Extension (ADE) for navigation applications.
- Graph Database as a backend for Navigation Applications
    1. What are the differences between graph databases like Neo4j compared to "conventional" relational databases?

  - Which benefits and disadvantages exist in comparison to relational databases (using Structured Query Language (SQL))?
  - How can graph databases and the CityGML standard work together?

2. How can the CityGML structure be changed or improved in a graph database?
  - What connections should be added/modified?
  - How can information, like the proximity of objects, be extracted?
  - Which information should be added to enhance the value of the graph structure (relationships, nodes, and attributes)?
3. How can graph databases be used for solving navigation problems?
  - Which benefits bring graph databases for handling CityGML data?
  - Which route finding/shortest path algorithms can be used?
  - Which Cypher queries are needed and how can they be optimized?

- (Multimodal) Navigation - improvements on existing applications
  1. How can CityGML data in a graph database be used efficiently during the execution of a multimodal algorithm?
     - How can existing algorithms, like Dijkstra or A*, be used for multimodal navigation?
     - Which benefits can one gain by using multimodal navigation?
     - Is the CityGML standard a capable base for multimodal navigation applications?
     - How can multimodal navigation algorithms be implemented for graph databases?

1.3. Methodology

In order to solve the research questions a selection of algorithms and tools is used. Firstly, a graph database in this case Neo4j, is used to store a CityGML test dataset. This data is further processed to create a suitable network structure for routing purposes. During the pre-processing, the query language Cypher, combined with the Neo4j Python driver and the programming language Python will be used. To compare and explore the structure of CityGML and the graph database representation, FME (Safe Software Inc, 2023), ArcGIS Pro (Environmental Systems Research Institute, Inc., 2023), Notepad++ (Don Ho, 2023) and Glogg (Nicolas Bonnefon, 2023) are used. To find the shortest path between two nodes of the routing network, the algorithms Dijkstra and A* will be considered with different ways to calculate weights for the algorithms (Dechter & Pearl, 1985; Dijkstra, 1959).

1.4. Used Tools and Scenario

This thesis uses CityGML datasets derived from OpenDRIVE data of the city of Ingolstadt and Grafing near Munich (Grafing bei München). This data has been converted to CityGML through the r:trån software (Schwab, Benedikt et al., 2023). Furthermore, the CityGML data has been mapped to graph database elements in a Neo4j graph database (Neo4j, Inc., 2023i). The first test dataset includes the scene of an intersection with road, bicycle and pedestrian paths in the city of Ingolstadt. Whilst the second dataset includes a larger region with streets in Grafing, it does contain different scenarios, e.g., includes parking areas, has a different style of sidewalks, and does not contain bike lanes. Based on the Grafing Dataset, a third synthetic dataset containing a hand-modelled parking garage building that is connected to the original data will be used. This allows testing of routing functionality within buildings and with different granularity levels. The test application will be implemented in Python. For interacting with the graph database the Cypher query language is used from within Python through the Neo4j Python driver. To solve routing problems, the functionalities of Neo4j and the Awesome Procedures on Cypher (APOC) extension are used in combination with Python libraries for spatial analysis and visualization. These are formally introduced later on.

1.5. Structure of the Thesis

The following chapter 2 introduces the theoretical background and important concepts used during the thesis. In Chapter 3, the focus is on the development of a concept for a navigation application using CityGML data. It explains the relevant steps and introduces the tools used for the implementation. Chapter 4 presents the results of the concept and their implementation. The fifth chapter reflects the results critically and discusses the limitations of the approach. Finally, chapter 6 concludes the thesis and gives an outlook on future work.

2. Theoretical Frame and Literature Review

2.1. Definitions and Used Terminology

Before starting with the actual theoretical background it is important to mention that there are terms like network and graph, point, vertex, and node, or edge, relationship, and connection that are used interchangeably in this thesis, as in the literature different terms are used to describe the same concepts. While a point is typically associated with a location having specific coordinate values in Geodesy, it can also be a node embedded in a network structure representing arbitrary information. Thus, there may be small differences in the terms depending on the field the literature originates from. However, for this thesis, they can be used interchangeably for the most part or will be specifically used according to the context. For example, node and relationship when talking about graph databases.

In the field of navigation, there are also different categories and types of systems. Starting with the related field of Intelligent Transportation System (ITS) , which are defined by the European Union directive 2010/40/EU in the following way. "Intelligent Transport Systems (ITS) are advanced applications which without embodying intelligence as such aim to provide innovative services relating to different modes of transport and traffic management and enable various users to be better informed and make safer, more coordinated and 'smarter' use of transport networks" (Smith, 2015). Thus, ITS also deal with transportation networks and making informed decisions based on information derived from those. Navigation will be formally defined in section 2.2.1. However, the term navigation includes the meaning of a complex navigation system consisting of multiple instances of hardware and software or just the logic part regarding routing and shortest path search. In this thesis, the focus is narrowed down to the latter and lies on the core functionality of finding optimal or shortest paths in a network structure. This is also the reason why the term routing is used interchangeably with navigation in this thesis. While routing can be considered a part of navigation, it is the main focus of this thesis.

Other important terms like ADE for the Extensible Markup Language (XML)-based CityGML or data structures and algorithms, like k-d trees or shortest path algorithms, used in the thesis are defined or explained in the respective sections.

2.2. Conceptual Structure of a Navigation System

2.2.1. The Parts of a Navigation System

As this thesis covers the general topic of navigation it is necessary to look at the different components of a navigation system. Navigation is defined by B. Hofmann-Wellenhof. "Navigation deals with moving objects (mostly vehicles) and involves trajectory determination and guidance. Trajectory determination relates to the derivation of the state vector of an object at any given time. Typically, the state vector includes position, velocity, and attitude. While trajectory determination only refers to deriving the motion characteristics of an object without interaction, guidance forces the moving object onto a predetermined route to reach a given destination" (Hofmann-Wellenhof et al., 2011). He further defines routing as "[...] routing in the sense of route planning is responsible for defining appropriate routes. It addresses questions like 'where to go?' and 'how to go?'" (Hofmann-Wellenhof et al., 2011). While route guidance is defined as "[...] guiding an object or vehicle along the predefined route. Thus, it answers questions like 'what to do next?' in terms of maneuvers" (Hofmann-Wellenhof et al., 2011). Another definition is given by the Cambridge online dictionary which defines navigation in the transport sector as "the act of directing a ship, aircraft, etc. from one place to another, or the science of finding a way from one place to another" (Cambridge University Press & Assessment, 2023a). Routing is defined as "the use of a particular path or direction for something to travel or be placed" (Cambridge University Press & Assessment, 2023b).

Those definitions show the formal difference between navigation and routing. Navigation covers the whole process of moving from one place to another while routing only covers the process of finding the ideal path out of multiple possible paths. Following the definition of Hofmann-Wellenhof, this thesis only focuses on the routing aspect of navigation and aims to solve the question "how to?". Figure 2.1 shows a core problem of navigation systems, finding the optimal route out of multiple possible routes. This graphic can also be converted into a question that represents a typical task that should be solved in the domain of navigation applications:

*How can a person reach a given location (**destination**) in the shortest time (**cost to minimize**) possible from its current location (**start, time-dependent**) using a number of given transportation modes (**constraint**)?*

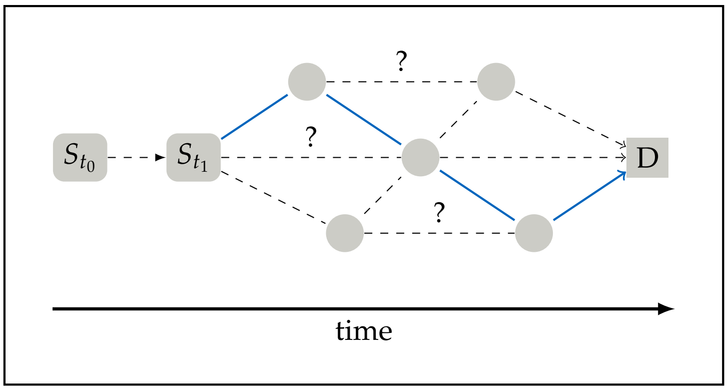


Figure 2.1.: A central problem of navigation: Finding the optimal route from a start location S at time t_i to a destination D.

To solve such problems a navigation system consists of two parts, hardware and software. The hardware part includes the physical devices for the end-user as well as computing units and other infrastructure to serve relevant information. This includes for example Global Navigation Satellite System (GNSS) antenna, processor, memory, storage, and display. GNSS satellites and server systems provide navigation services including positioning, routing and map displays. The hardware is used to calculate the best route, visualize the results on a display, or give audio instructions and allow a user to interact with the software. The employed devices range from smartphones over dedicated navigation systems to servers and navigation satellites. Additionally, the software part prepares the (collected) data and provides a base for performing fast queries on network datasets using implemented routing algorithms. This includes preprocessing as well as the actual routing engine that is used to solve navigation tasks. Depending on the chosen hardware, the usage of specialized software, data structures and algorithms is vital for real-time performance on mobile devices or the scalability of the system in a server environment (Walter et al., 2013; Waze Mobile Ltd., 2023b). As hardware components such as the choice of processor, the amount of memory and data storage capacities are important for the performance of the software, it is noteworthy that the development of such an optimization is not part of the thesis. However, the usage of a graph database rather follows the choice of having a centralized server-based system over local computing on the mobile device. On the one side, this would remove constraints like storage and memory limitations but on the other side, it would limit the application to online usage. Further hardware dependencies are not discussed in this thesis but could be a vital point for optimizing large-scale application usage. Lastly, a differentiation has to be made for the software part presented in the thesis, as the software of a navigation system is further divided into the user interface, including option selection and visualization of the results, a data structure, and the routing engine, which implements the algorithms. Here, the focus is set on the data structure and partly on the routing engine. Developing a practical and useful user interface for deployment or an optimized calculation framework is not part of the thesis.

2.2.2. Modelling the Real World

Industry Leaders

As for industry leaders, this thesis will look at the products of Google, Waze and HERE. While other companies such as TomTom, Apple, and Microsoft also provide navigation services, they are not further discussed in this thesis. The reason for this is that the internal functioning of all those navigation systems is not publicly available. Therefore, this selection is based on the availability of information and includes a general-purpose navigation system (Google Maps), a navigation system based on crowdsourced data (Waze) and a navigation system with a wider range of modifiers (HERE). Each of those navigation solutions has a specialized focus. This also shows that there is no one-size-fits-all solution for navigation. Rather, a unique solution for each use case is needed (Louise Wylie, 2023; Statista, 2023; Team Counterpoint, 2022).

Google Maps
In their application as well as the mobile smartphone pendant, Google Maps provides several modes of transportation for routing. These include car/road vehicles, public transportation, walking, flying (for long trips in the web application), shared rides, like taxis (in the mobile version), and biking. The public transportation option provides some multimodal capabilities as the suggested routes include foot walks to the nearest public transportation hub to start the trip as well as on-foot routes from the last stop of the public transportation network to the destination. While this uses two different modes of transportation, the combined analysis of the other transport types is not available. Thus, the multimodal routing capabilities are limited. Routes that e.g., take one to the nearest Park and Ride (P+R) parking lot and then continue with public transportation are not possible without some manual work. The underlying data model is not available as it is proprietary software of Google. The same applies to the used algorithms and the routing engine. When taking a look at the options it can be derived however that the data model contains different networks for the transportation types as well as different types of streets. There is also an option to differentiate elements like toll roads and highways. It is further possible to choose certain public transportation types. Furthermore, public transportation routes provide information about the necessary transits (Google Cloud EMEA Limited, 2023; "Google Maps", 2023).

Waze

The main data model structure and used algorithms are also unavailable to the public. However, some general concepts used for route finding are explained on the Waze website. These include some generalisation, simplification of the network as well as assumptions to shorten computation time. While the algorithm details are unknown, the main focus of Waze lies on crowd-sourced data providing highly up-to-date information about traffic jams, accidents, and dangers. Waze also allows users to update the map or mark closed roads and more. This is done via the Waze map editor. The real-time crowd-sourced data are directly submitted via the mobile application by the users (Waze Mobile Ltd., 2023c, 2023d). Waze was acquired by Google in 2013. After 2013 both Google Maps and Waze coexisted (Levine, 2023). In terms of navigation, Waze only provides car routing delivering no multimodal routing options (Waze Mobile Ltd., 2023a).

HERE

Whilst HERE provides two applications for the end-user, the mobile application HERE WeGo, as well as HERE Navigation for connected vehicles (HERE Global B.V., 2023a). Further tools are available via their Application Programming Interface (API) selection. Those services include map rendering and geocoding, positioning as well as routing, tour planning and transit. These APIs allow the analysis of a wide range of situations. In the context of this thesis, the routing API v8 and the HERE Intermodal Routing API v8 contain interesting and possibly similar information analysis. While the routing API is capable of accepting detailed restrictions on transportation mode as well as basic roads to avoid, it is also possible to specify information about the used vehicle like hazardous goods transport, weight and dimensions (width, length and height) and more. While this allows only one mode of transport, their so-called Intermodal Routing API allows the combination of four distinct transportation types. These types are vehicle, transit (public transportation), taxi, and rentals. Additionally, the service includes pedestrian routing between the transportation types. Furthermore, sharing and rental services include car sharing, taxi, bicycle and more. Thus, this seems to be a real multimodal solution. Nevertheless, as with the two other exemplary navigation providers the inner working is not available publicly (HERE Global B.V., 2023b, 2023c). Thus it is difficult to determine how the 3D elements like the vehicle dimensions are handled or how the real world is modelled.

2.3. Relevant Data Models and Standards

When creating an application the underlying data models are of great importance. This section introduces the most relevant standards related to the thesis topic.

2.3.1. Geographic Data Files (GDF)

The International Organization for Standardization (ISO) standard 20524-1:2020 Geographic Data Files (GDF) is primarily used in vehicle navigation to exchange information between map producers and navigation system integrators (International Organization for Standardization, 2020b). It supports the modelling of a wide range of objects and allows the representation of many street elements and their surroundings. Originally published in 2011 by the ISO, the standard consists of several sections and was updated in 2020 to version 5.1. GDF provides an XML schema and allows the modelling of 3D objects. This, for example, allows the correct representation of bridges. However, the 3D modelling is limited to a minimum and maximum height value above the terrain. Thus, elements like buildings have a relative height to the terrain elevation (Beil et al., 2020; International Organization for Standardization, 2020a, 2020b).

2.3.2. OpenDRIVE

OpenDRIVE was originally developed by VIRES Simulationstechnologie GmbH and is now maintained by ASAM e.V. (Association for Standardization of Automation and Measuring Systems). It is used in the automotive industry for driving simulations and is based on a XML structure. The current version is 1.7.0 and was published in 2021. The modelling principles of OpenDRIVE follow a reference line, which is the core piece of every road. Objects are added relative to this reference line. Global coordinates are supported as well. A road consists of individual segments which are linked together. Lanes can be linked between road segments and roads, allowing the modelling of complex, lane-based street networks. The focus of this standard lies on static objects around the road needed to facilitate driving simulations in a realistic manner (ASAM e.V., 2023; Beil et al., 2020).

2.3.3. CityGML

Overview

CityGML is an open data model and includes an XML-based exchange format for semantic 3D city and landscape models. The standard is provided by the OGC and can be used to store, exchange and visualize semantic 3D city models. It is currently in version 3.0.0 and used in many applications in the field of urban planning, architecture, and civil engineering (Kolbe et al., 2023).

Modules

The CityGML standard is divided into several modules, each describing a specific topic. Those modules can be seen in figure 2.2. The vertical boxes display the thematic modules. Horizontal boxes contain modules that apply to all thematic modules. The most important module for the thesis is the Transportation module as it models the street space. However, the other modules are also relevant as they can be used to model the surroundings of the street space or in the case of the CityFurniture module, provide additional information regarding street furniture, like street signs and traffic lights.

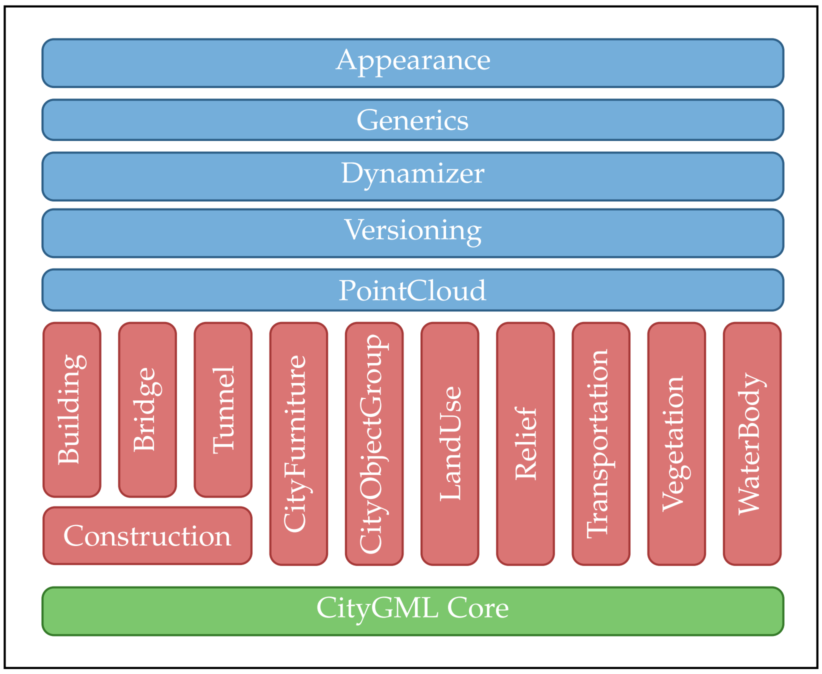


Figure 2.2.: CityGML modules according to (Kolbe et al., 2023)

CityGML 3.0.0

With the changes made from version 2.0 to 3.0, the Transportation module was extended and improved. The module consists of different classes. Starting from the core Core::Abstract-UnoccupiedSpace class, which is the base class for all transportation objects, the module is divided into the following classes:

AbstractTransportationSpace, which can be further divided into: Track, Road, Waterway, Railway, Section, Intersection, and Square. Furthermore, the traffic space is described by the following classes: ClearanceSpace, TrafficSpace, and TrafficArea with HoleSurface, as well as their counterparts for objects that are not primarily used for traffic, but are essential for the traffic space: AuxiliaryTrafficSpace, AuxiliaryTrafficArea, Marking, and Hole.

Additionally, it is possible to model the traffic direction and granularity of the traffic space. Other relevant classes are from the core module and are ClosureSurface and AbstractThematicSurface. Lastly, the Occupancy data type from the core module is also used in the Transportation module. A complete UML class diagram of the Transportation module is shown in figure 2.3 (Kolbe et al., 2023).

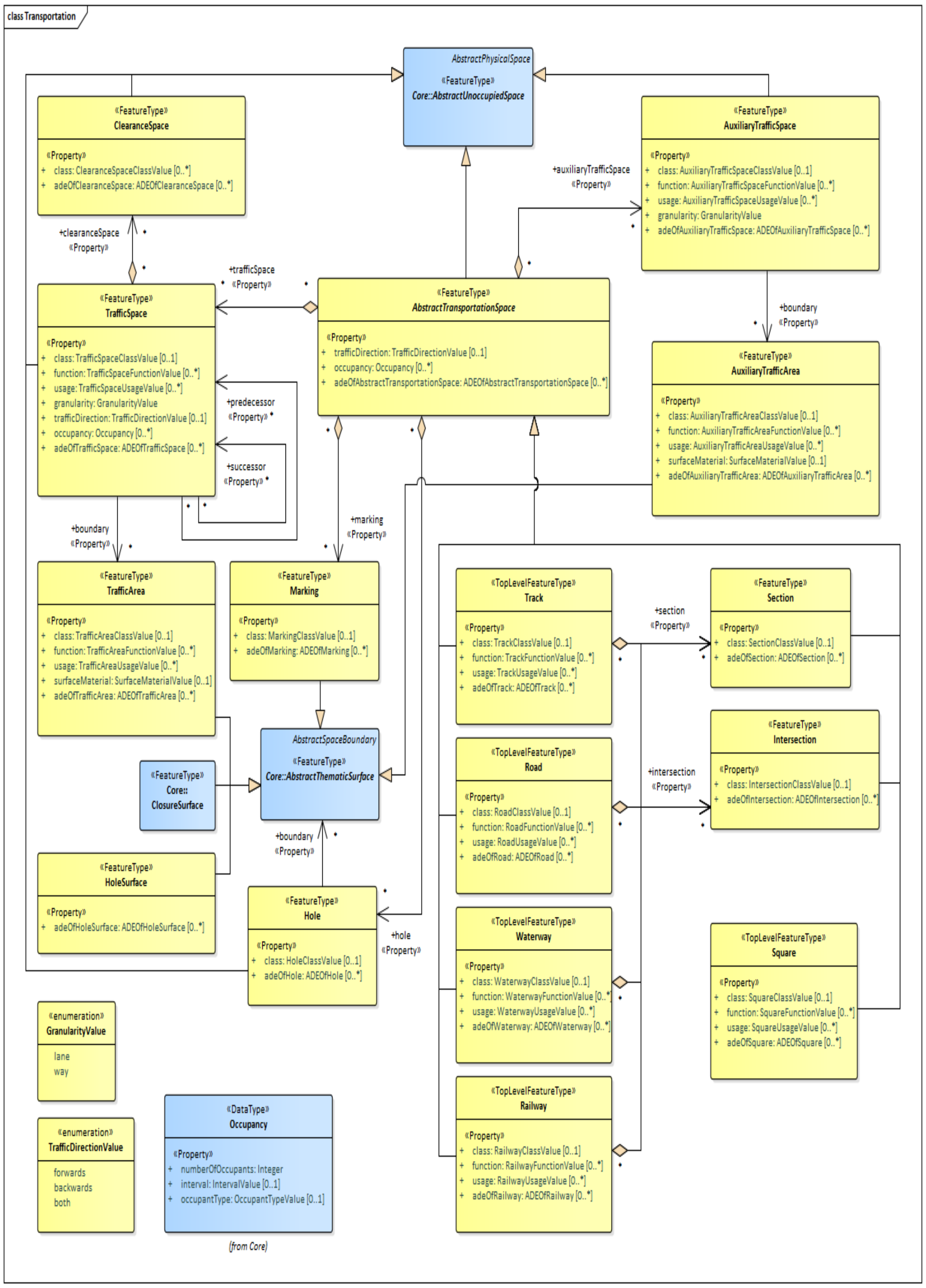

class Transportation
AbstractPhysicalSpace
«FeatureType»
Core::AbstractUnoccupiedSpace
«FeatureType»
ClearanceSpace
«Property»
+ class: ClearanceSpaceClassValue [0..*]
+ adeOfClearanceSpace: ADEOfClearanceSpace [0..*]
«FeatureType»
AuxiliaryTrafficSpace
«Property»
+ class: AuxiliaryTrafficSpaceClassValue [0..1]
+ function: AuxiliaryTrafficSpaceFunctionValue [0..*]
+ usage: AuxiliaryTrafficSpaceUsageValue [0..*]
+ granularity: GranularityValue
+ adeOfAuxiliaryTrafficSpace: ADEOfAuxiliaryTrafficSpace [0..*]
+auxiliaryTrafficSpace
«Property»
+clearanceSpace
«Property»
+trafficSpace
«Property»
«FeatureType»
TrafficSpace
«Property»
+ class: TrafficSpaceClassValue [0..1]
+ function: TrafficSpaceFunctionValue [0..*]
+ usage: TrafficSpaceUsageValue [0..*]
+ granularity: GranularityValue
+ trafficDirection: TrafficDirectionValue [0..1]
+ occupancy: Occupancy [0..*]
+ adeOfTrafficSpace: ADEOfTrafficSpace [0..*]
«FeatureType»
AbstractTransportationSpace
«Property»
+ trafficDirection: TrafficDirectionValue [0..1]
+ occupancy: Occupancy [0..*]
+ adeOfAbstractTransportationSpace: ADEOfAbstractTransportationSpace [0..*]
+boundary
«Property»
«FeatureType»
AuxiliaryTrafficArea
«Property»
+ class: AuxiliaryTrafficAreaClassValue [0..1]
+ function: AuxiliaryTrafficAreaFunctionValue [0..*]
+ usage: AuxiliaryTrafficAreaUsageValue [0..*]
+ surfaceMaterial: SurfaceMaterialValue [0..1]
+ adeOfAuxiliaryTrafficArea: ADEOfAuxiliaryTrafficArea [0..*]
+predecessor
«Property»
+successor
«Property»
+marking
«Property»
+boundary
«Property»
«FeatureType»
TrafficArea
«Property»
+ class: TrafficAreaClassValue [0..1]
+ function: TrafficAreaFunctionValue [0..*]
+ usage: TrafficAreaUsageValue [0..*]
+ surfaceMaterial: SurfaceMaterialValue [0..1]
+ adeOfTrafficArea: ADEOfTrafficArea [0..*]
«FeatureType»
Marking
«Property»
+ class: MarkingClassValue [0..1]
+ adeOfMarking: ADEOfMarking [0..*]
«TopLevelFeatureType»
Track
«Property»
+ class: TrackClassValue [0..1]
+ function: TrackFunctionValue [0..*]
+ usage: TrackUsageValue [0..*]
+ adeOfTrack: ADEOfTrack [0..*]
+section
«Property»
«FeatureType»
Section
«Property»
+ class: SectionClassValue [0..1]
+ adeOfSection: ADEOfSection [0..*]
AbstractSpaceBoundary
«FeatureType»
Core::AbstractThematicSurface
«FeatureType»
Core::
ClosureSurface
«TopLevelFeatureType»
Road
«Property»
+ class: RoadClassValue [0..1]
+ function: RoadFunctionValue [0..*]
+ usage: RoadUsageValue [0..*]
+ adeOfRoad: ADEOfRoad [0..*]
+intersection
«Property»
«FeatureType»
Intersection
«Property»
+ class: IntersectionClassValue [0..1]
+ adeOfIntersection: ADEOfIntersection [0..*]
+boundary
«Property»
«FeatureType»
HoleSurface
«Property»
+ adeOfHoleSurface: ADEOfHoleSurface [0..*]
«FeatureType»
Hole
«Property»
+ class: HoleClassValue [0..1]
+ adeOfHole: ADEOfHole [0..*]
+hole
«Property»
«TopLevelFeatureType»
Waterway
«Property»
+ class: WaterwayClassValue [0..1]
+ function: WaterwayFunctionValue [0..*]
+ usage: WaterwayUsageValue [0..*]
+ adeOfWaterway: ADEOfWaterway [0..*]
«TopLevelFeatureType»
Square
«Property»
+ class: SquareClassValue [0..1]
+ function: SquareFunctionValue [0..*]
+ usage: SquareUsageValue [0..*]
+ adeOfSquare: ADEOfSquare [0..*]
«enumeration»
GranularityValue
lane
way
«enumeration»
TrafficDirectionValue
forwards
backwards
both
«DataType»
Occupancy
«Property»
+ numberOfOccupants: Integer
+ interval: IntervalValue [0..1]
+ occupantType: OccupantTypeValue [0..1]
(from Core)
«TopLevelFeatureType»
Railway
«Property»
+ class: RailwayClassValue [0..1]
+ function: RailwayFunctionValue [0..*]
+ usage: RailwayUsageValue [0..*]
+ adeOfRailway: ADEOfRailway [0..*]

Figure 2.3.: UML class diagram of the Transportation module of CityGML 3.0 taken from the CityGML conceptual model (Kolbe et al., 2023)

Whilst CityGML is not focused on providing a specialized standard for navigation applications, it provides several structures that can be used. For utilizing different accuracies of routing, three levels of detail called granularity can be used. The granularity 'area' is the entire width of a street. It represents the entire section or intersection of a road. For the granularity 'way' a differentiation between different traffic types is made. This way individual spaces along the street can be addressed, such as sidewalks, carriageways, or green areas. Lastly, the granularity 'lane' is the most accurate, allowing the introduction of different lanes on a single surface. This can be used, for example, to model the different lanes for vehicles on a carriageway.

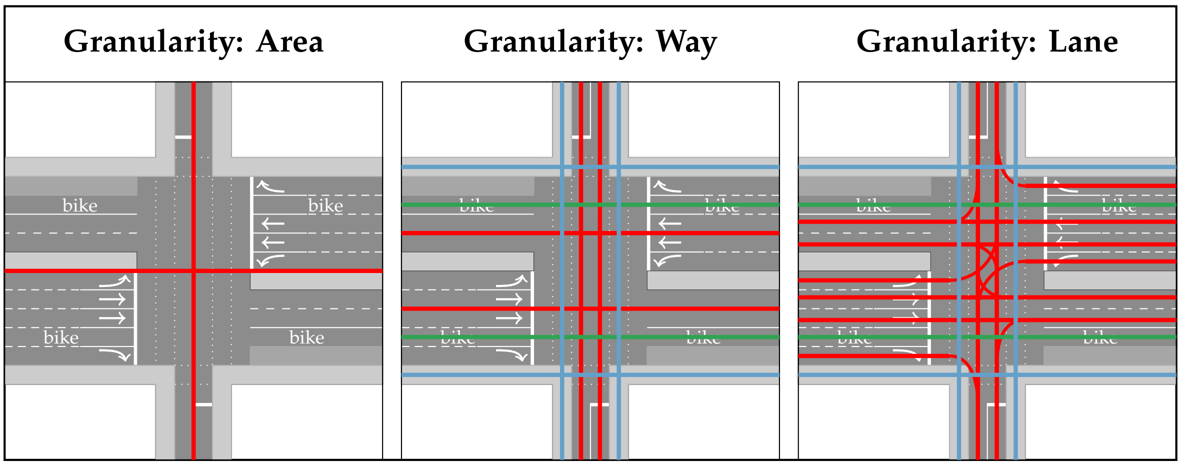


Figure 2.4.: The three granularity levels of the Transportation module, adapted from (Beil et al., 2022)

Each object requires a unique identifier, which is realized by the *"gml:id"*. Additionally, (Auxiliary)TrafficSpaces require a granularity attribute of way or lane. Furthermore, the usage of additional attributes is recommended, which improves the usability of the data for navigation applications (Beil et al., 2022). These recommended attributes are a name for Road objects and information on the relationship of Sections and Intersections to a Road object. Furthermore, this also applies to TrafficSpaces which should contain information to link them to a Section or Intersection, additionally to the implicit linkage through the hierarchical structure of CityGML. Lastly, it is recommended for (Auxiliary)TrafficAreas to include information about their usage type, e.g., driving lane, sidewalk, biking lane, and surface material. While this already covers some information for navigation applications, information, like speed limits, additional restrictions or the type of road is not possible to be stored in CityGML in a standardized way. Such information can be included using generic attributes. Thus, the standard can be extended to include all the needed information for a navigation application. However, as generic attributes are not standardized their attribute names and values can vary between different data providers or modellers. Therefore, it is necessary to homogenize the data stemming from different sources and different generic attributes. Additionally, methods that rely on the availability of certain information provided by generic attributes should be robust if such information is not present. Furthermore, the lane-based traffic space representation would benefit from including a lane id information to easily identify neighbouring lanes. Nevertheless, CityGML also stores information in an implicit way. For pedestrian applications, the possibility to model sidewalks and other pedestrian areas is important. Here, CityGML allows modelling even subtle differences, such as lowered curb stones and traffic islands. While this information is not explicitly stored, it can be derived from the geometry of the objects, e.g., by comparing the geometric height to typical curb stone heights. For the usage in a navigation application, this might be problematic and a pre-processing of the data is necessary to extract such information.

CityGML further utilizes the XLink concept to connect TrafficSpaces to each other. This way, it is possible to use TrafficSpace elements and follow predecessor or successor relations between them, generating a graph structure. The concept applies to other geometric representations as well and can be used for different granularities. The general Section and Intersection principle applies to streets, railways and waterways. Additionally, the XLink concept can be used for other elements, such as buildings, bridges, and tunnels. Thus, it is possible to include streets through or inside buildings like garages or tunnels. The predecessor and successor relations between the TrafficSpace objects in combination with the attribute values from the other Transportation module objects can be used to model the connections of a street network and add rich information. Combined with its other modules, like the CityFurniture, Vegetation, and Dynamizer, CityGML is capable of modelling a wide range of objects and allows the representation of many street elements and the street surroundings (Beil et al., 2022; Beil et al., 2020; Kolbe et al., 2023; Kutzner et al., 2020).

Comparison of GDF, OpenDRIVE, and CityGML

A comparison of the three standards GDF, OpenDRIVE, and CityGML is provided in the table 2.1. It shows the different capabilities of the standards and their focus within the five main categories: geometry, semantics, topology, appearance and other aspects and their subcategories. The table is based on the comparison table of the standards by (Beil et al., 2020) and was revised to reflect changes due to updated standards.

Legend	🟥 Not available	🟨 Partially available	🟩 Fully available

	GDF 5.1	**OpenDRIVE 1.7**	**CityGML 3.0**
Geometry			
Coordinate Space	3D	3D	3D
Straight Line Segments	🟩 ✓	🟩 ✓	🟩 ✓
Splines	🟥 -	🟩 ✓	🟩 ✓
Clothoids	🟩 ✓	🟩 ✓	🟩 ✓
Parametric Rep.	🟨 a	🟩 ✓	🟥 -
Semantics			
Surface Material	🟩 ✓	🟩 ✓	🟩 ✓
Function	🟩 ✓	🟩 ✓	🟩 ✓
Driving Ways	🟩 ✓	🟥 -	🟩 ✓
Driving Lanes	🟩 ✓	🟩 ✓	🟩 ✓
Driving Direction	🟩 ✓	🟩 ✓	🟩 ✓
Traffic Logic	🟩 ✓	🟩 ✓	🟩 ✓
Bridge Model	🟨 b	🟨 c	🟩 ✓
Tunnel Model	🟨 b	🟨 c	🟩 ✓
Road Marking	🟩 ✓	🟩 ✓	🟩 ✓
Street Furniture	🟩 ✓	🟩 ✓	🟩 ✓
Vegetation Objects	🟩 ✓	🟨 d	🟩 ✓
Multiple Traffic Types	🟩 ✓	🟨 d	🟩 ✓
Level of Detail	🟩 ✓	🟥 -	🟩 ✓
Topology			
Linear Ref.	🟩 ✓	🟩 ✓	🟥 -

Road/Lane Linkage	✓	✓	✓
Appearance			
Texture	e	-	✓
Other Aspects			
Main Application/Purpose	Navigation	Driving Simulation	City Models and their applications
Encoding	XML, binary	XML	GML/XML/JSON
Developer/Issuer	ISO/TC204	ASAM	OGC

(a) Attributes such as Road Surface Type or Road Surface Condition exist. Enclosed TrafficAreas are used for parking areas;
(b) Modelled in a generic way as "Structures";
(c) Indicated if Roads are part of Tunnels/Bridges;
(d) Vegetation can be represented using "objects"/Railroad objects can be represented but only in context with roads;
(e) The attribute "texturedSurfaceAvailable" can be used to indicate if textured surfaces are available.

Table 2.2.: Comparison of standards dealing with street space modelling and navigation (Beil et al., 2020) (revised)

CityGML Graph Format - CityGML in Neo4j

As mentioned in section 2.3.3, CityGML is an XML-based standard and application of Geography Markup Language (GML). This means that the data is stored in a hierarchical structure. The hierarchical structure is further expanded by the use of XLinks. Thus, the data does not follow a tree-like structure typical for hierarchies, but rather a graph-like structure. To utilize this advanced structure, the data can be mapped to a graph database. This is explained by Son Nguyen covering the comparison of two CityGML files using the graph database Neo4j (Nguyen, S. H., 2017). After the data has been mapped to the graph database, it is possible to use the graph structure to perform queries on the data. These queries can take advantage of the specialized data structure and algorithms applicable to graphs. This is further explained in section 2.4.

The following figures show exemplary mapping of the CityGML structure to a graph representation.

```xml
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<CityModel [xmlns ...] >
  <gml:boundedBy>
    <gml:Envelope>
      <gml:lowerCorner>678013.2694661662 5405245.044439525
   371.97267048774773</gml:lowerCorner>
      <gml:upperCorner>678277.1085440451 5405400.673101281
   401.8074700173935</gml:upperCorner>
    </gml:Envelope>
  </gml:boundedBy>
  <cityObjectMember>
    ...
  </cityObjectMember>
</CityModel>
```

Code 2.1: XML code showing a part of the CityGML structure.

The information about the bounding box coordinates is stored within the tags *lowerCorner* and *upperCorner*. Those again are within an *Envelope* element inside the *boundedBy* tag. Besides this information, *cityObjectMember* elements are within the bounds of the *CityModel* tag. When looking at the graph representation of this excerpt in figure 2.5, one can identify tags like *CityModel* and *Envelope* as nodes while *boundedBy* and the tags *lowerCorner* and *upperCorner* are represented via a combination of relationships and nodes.

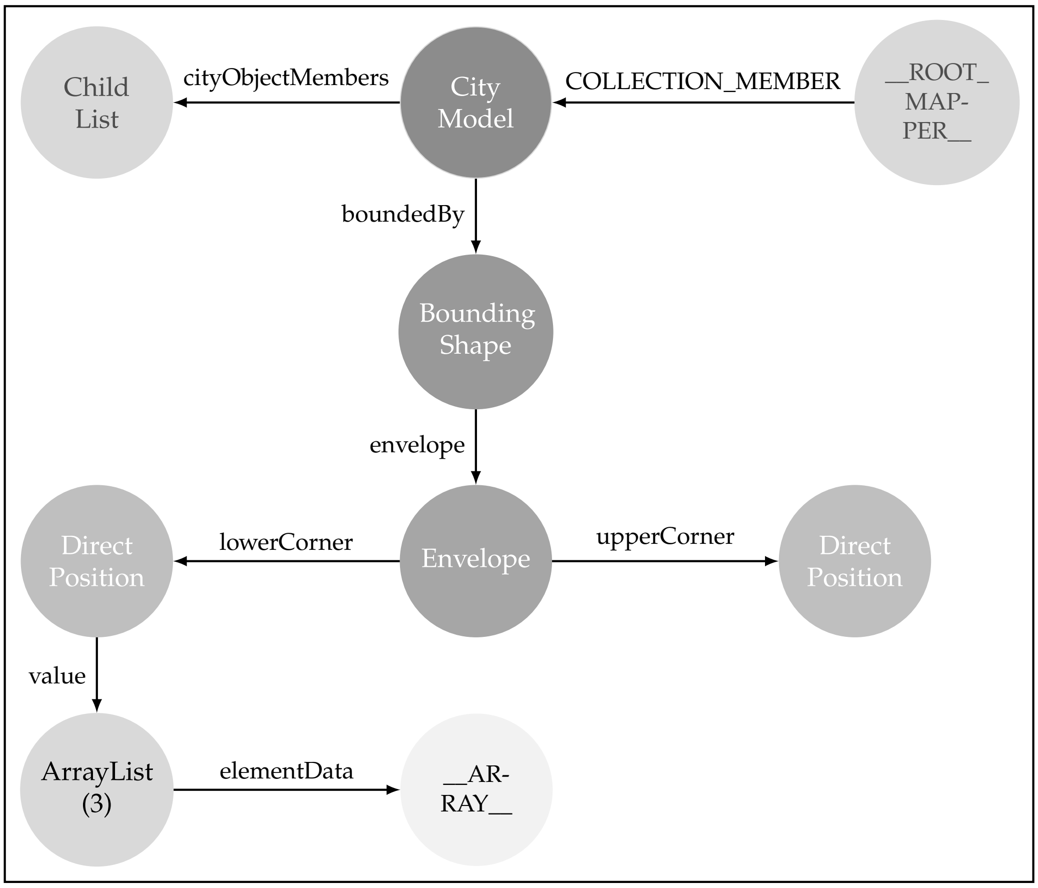


Figure 2.5.: Mapping of the CityGML XML structure to a graph representation.

```xml
<tran:trafficSpace>
  <tran:TrafficSpace gml:id="UUID_TrafficSpace">
    <boundary>
      <tran:TrafficArea gml:id="UUID_TrafficArea">
      ...
      </tran:TrafficArea>
    </boundary>
    <lod2MultiCurve>
      ...
    </lod2MultiCurve>
    <tran:granularity>lane</tran:granularity>
    <tran:trafficDirection>backwards</tran:trafficDirection>
    <tran:predecessor xlink:href="#UUID_TrafficSpacePredecessor"
  />
    <tran:successor xlink:href="#UUID_TrafficSpaceSuccessor"/>
  </tran:TrafficSpace>
</tran:trafficSpace>
```

Code 2.2: XML code showing a CityGML TrafficSpace example

When taking a look at the core element of the Transportation module, the TrafficSpaces, a similar pattern can be observed. The tag *trafficSpace* contains all the elements connected to a *TrafficSpace* node which is defined within the *TrafficSpace* tag. Relationships represent the hierarchical structure of the file, for example, following the *boundary* tag to the *TrafficArea* which is related to the TrafficSpace node connection to the TrafficArea node. In the figure 2.6 only the first relationship with the name *'boundaries'* of the chain is visible. Similarly, *lod2MultiCurve*, *granularity* and *trafficDirection* are represented in the graph model. One important difference is that the graph only contains **one** node for the value *'backwards'* while all TrafficSpaces with this *trafficDirection* link to the node instead of storing it as an attribute value. Lastly, the XLinks *predecessor* and *successor* are also represented as chains of relationships connecting two TrafficSpaces.

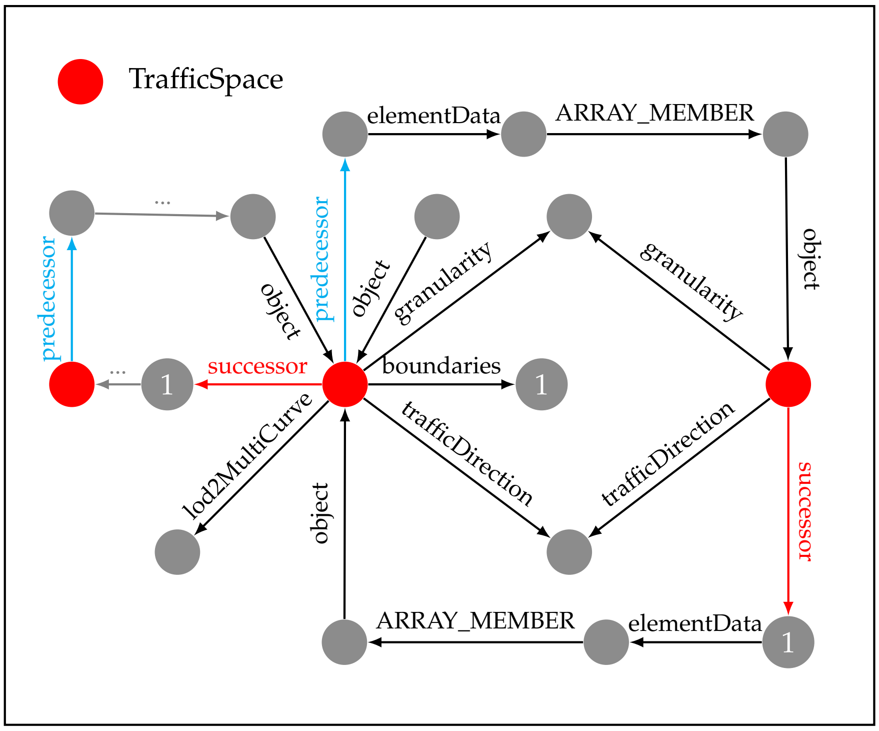


Figure 2.6.: Exemplary mapping of TrafficSpace objects to the graph model.

For preserving the actual traffic flow direction, the information about the traffic direction is stored in the attribute *'trafficDirection'*. This attribute can have the values *'forwards'*, *'backwards'* or *'both'* as defined in the Transport module. The attribute corresponds to the reference line of a street as used in OpenDRIVE. As such a reference line does not exist in CityGML, a value describing the direction is used. It is needed in combination with the predecessor and successor links to restore the actual traversal direction because in Neo4j relationships are always traversable in both directions. In the graph representation, this value is needed to correct the successor and predecessor relationships, as they also follow the non-existing reference line. Figure 2.7 shows the issue for driving on the right-hand side.

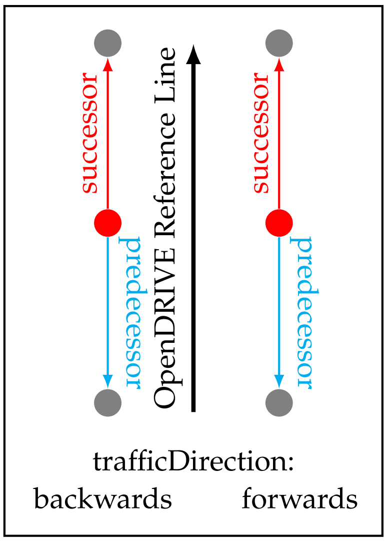


Figure 2.7.: Successor/Predecessor mapping in CityGML and thus the graph representation follow the reference line of OpenDRIVE.

If the CityGML data is modelled the trafficDirection attribute can be used to model the traffic flow direction, whilst the successor and predecessor relationships should also reflect the correct direction according to the trafficDirection attribute. This is especially important when traffic rules like one-way streets shall be modelled correctly.

2.3.4. Other Standards

LandInfra

The OGC standard LandInfra (Land and Infrastructure Conceptual Model Standard) focuses on land and civil engineering infrastructure and provides methods to describe relevant information. It includes different subject areas with "Alignment" and "Road" being the most important in terms of street space modelling (Beil et al., 2020).

INSPIRE

INSPIRE (Infrastructure for Spatial Information in Europe) is a European Union directive to establish an infrastructure for spatial information in Europe (European Commission, 2023). Its main goal is to combine spatial data seamlessly across different sources and borders. INSPIRE relies on the ISO 19100 series (Beil et al., 2020). The directive came into force in 2007 and the full implementation was required by 2021. In total, it contains 34 spatial data themes for environmental applications. In annex I, the transportation network theme is defined (European Union, 2023). It is possible to model all major transportation networks. However, topology is not explicitly modelled though the use is possible as data providers must provide information on how to reconstruct topologic relationships (Beil et al., 2020).

OpenStreetMap

OpenStreetMap (OSM) is a project that aims to create a free editable map of the world. It is a collaborative and open project. The main elements of OSM are nodes, ways, and relations. Nodes define points in space, ways linear features and boundaries, and relations are used to model relationships between elements. Additionally, tags can be added to these elements to describe them further (OpenStreetMap Foundation contributors, 2023a). The tags include, for example, Aerialway, Aeroway, Building, Highway, Place, Public transport, Railway, Route, and Waterway. Those keys can have further values for specifying the type of the element (OpenStreetMap Foundation contributors, 2023b). OSM is available in the data formats PBF (Protocolbuffer Binary Format), OSM XML, OSM JSON, and o5m (compressed binary format, which uses the same structure as OSM XML) (OpenStreetMap Foundation contributors, 2023c).

OKSTRA

The German Federal Ministry for Digital and Transport (BMDV) provides the 'Objektkatalog für das Straßen- und Verkehrswesen' (OKSTRA) data standard which includes street object characteristics and guidelines on recording and manipulation of those (Bundesministerium für Digitales und Verkehr, 2023). Together with the "Anweisung Straßeninformationsbank" (ASB) (Instruction road information bank), they are used for facility and asset management of German streets. The OKSTRA®-Version 2.021 was released on 19th May 2023 (Bundesanstalt für Straßenwesen, 2023). OKSTRA models the street space based on the ISO standards including ISO 19107 (International Organization for Standardization, 2019) and ISO 19109 (Beil et al., 2020; International Organization for Standardization, 2015).

Vissim

"Vissim is one of the most used multimodal traffic simulation software and can be applied for planning of different traffic scenarios or with regard to traffic light control. Potential traffic members include cars, buses, trucks, bikes, pedestrians, or trams" (Beil et al., 2020). As Vissim is commercial software, the data format and internal specifications are not publicly available. However, Roland Ruhdorfer provides a detailed best-effort analysis of the data format and internal structure of Vissim (Ruhdorfer, 2017).

2.4. Graph Theory - Graph Structures of Neo4j

2.4.1. General Concepts

Figure 2.8 shows the general conception of a graph. A graph G is defined as a tuple $G = (V, E)$, where V is a set of vertices and E is a set of edges. The edges can be directed or undirected. In the case of a directed graph, the edges are ordered pairs of vertices u, v, where u is the source vertex and v is the target vertex. In the case of an undirected graph, the edges are unordered pairs of vertices u, v. In the case of a weighted graph, the edges are additionally assigned a weight $w(u, v)$, which can be used to represent the cost of traversing the edge u, v (Reinhard Diestel, 2016; Sven Oliver Krumke & Hartmut Noltemeier, 2012).

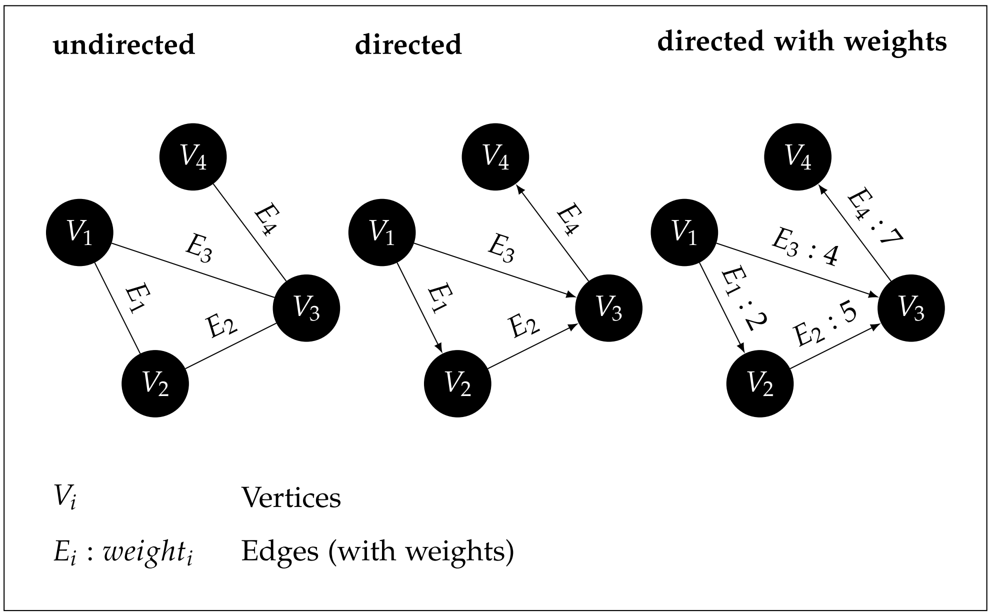
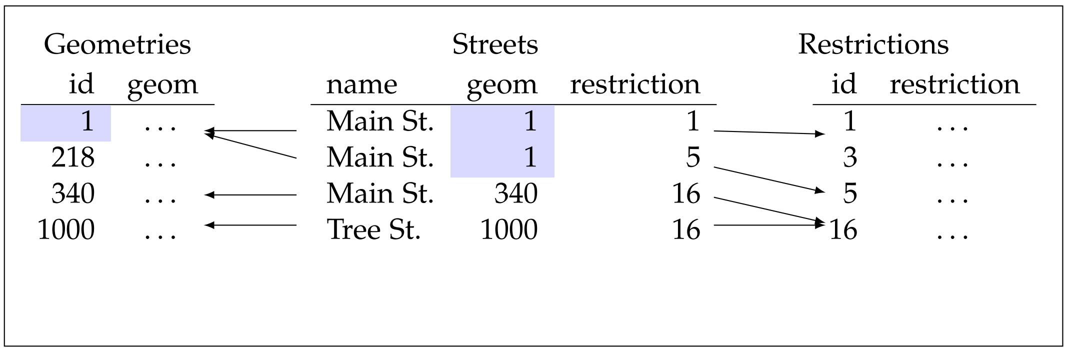


Figure 2.8.: Exemplary graph structure of an undirected, a directed and a directed graph with weights.

2.4.2. Structure and Functionality of Neo4j

Neo4j is called a native graph database as it represents the graph as a true graph model even on the storage level. Thus, the data is stored as it is modelled. This is in contrast to other graph databases, which can have a graph abstraction on top of another storage system. Furthermore, Neo4j provides ACID transactions, cluster support and runtime failover (Neo4j, Inc., 2023l). While the data model chosen is important in determining the logic of queries and the structure of the stored data, it is not required to have a fixed schema in Neo4j. Neo4j does not need a schema, it is a schema-free system. This means that the data model can be changed and adapt easily to new data requirements (Neo4j, Inc., 2023f).

While relational databases rely on a fixed schema and consist of multiple tables connected via index lookups combined with table joins to obtain desired connected data stored in multiple tables, this is not the case for Neo4j. Performance issues when joining multiple tables with a large number of rows on each table can occur. Additionally, queries can become complex in order to obtain the desired information. Multiple joins can be necessary. The following figure 2.9 illustrates this problem, showing a relational database with multiple tables for storing street names, the corresponding driving restrictions and geometry stored in separate tables.



Geometries

id	geom
1	...
218	...
340	...
1000	...

Streets

name	geom	restriction
Main St.	1	1
Main St.	1	5
Main St.	340	16
Tree St.	1000	16

Restrictions

id	restriction
1	...
3	...
5	...
16	...

Figure 2.9.: Exemplary table structure of a relational database.

Here, the different tables must be joined to obtain the connected data. In the case of the graph database, this issue does not exist, as the data is already stored in a connected way. Thus, there is no need for joins and index lookups (Neo4j, Inc., 2023g).

Extending the general graph definition of a graph $G = (V, E)$, the graph database Neo4j uses the following structure and terms:
Vertices are called nodes and edges are called relationships. Nodes can describe any entity or discrete object. A node label and a relationship type can have any type associated with it. The type of a node is called a label. Nodes can have zero or more labels, whereas relationships must have exactly one type to define the relationship. Furthermore, each node and relationship can have an arbitrary amount of attributes called properties. The properties are defined as key-value pairs. As mentioned before, when creating new data in Neo4j, no schema has to be followed that has been defined beforehand, as Neo4j is schema optional. This means that the structure of the graph can be changed. In order to gain performance benefits or to ensure a certain modelling structure it is possible to use indexes and constraints (Neo4j, Inc., 2023e). The missing schema also means that it

is not possible to create a graph consisting of a conceptual structure. Thus, the graph is always filled with concrete data. Lastly, there are some naming conventions for Neo4j data models (Neo4j, Inc., 2023h). These are listed in the following table 2.3:

Graph entity	Recommended style	Example
Node label	Camel case	:Street or :TrafficSpace
Relationship type	Upper case, using underscore to separate words	:SUCCESSOR_OF
Property	Lower camel case, beginning with a lowercase character	id or laneType

Table 2.3.: Naming conventions for graph entities in Neo4j (Neo4j, Inc., 2023h)

2.4.3. Cypher - Data Querying

Cypher is the query language used to obtain data from the graph database. It was developed by Neo4j to be a query language for graph databases, like SQL for relational databases. Today, Cypher is used and supported by many other graph databases, e.g., Amazon Neptune, SAP HANA Graph and RedisGraph and is an open standard provided by the openCypher project. Cypher is a declarative query language, which means that the user only has to specify what data is needed and not how to obtain it. The base structure of a Cypher query follows the graph structure and is represented by ASCII art (Neo4j, Inc., 2023j). A simple query to obtain all nodes connected to a start node with a certain relationship type and direction is shown in the code example 2.3.

```
// Return all nodes m connected to n by a relationship of type
  r:RELATIONSHIP_TYPE
MATCH (n)-[r:RELATIONSHIP_TYPE]->(m) RETURN m;
```

Code 2.3: Basic Cypher query matching a relationship of type RELATIONSHIP_TYPE

As described in the previous section 2.4.2, Cypher can use nodes and relationships in combination with their labels and properties. These can be used to filter the results or find specific relations in a dataset. Figure 2.10 shows the general structure of a Cypher query. In blue the nodes with a variable to reference them, the corresponding label to use and a property to specify a certain node. In green the relationship is depicted which is used to find all nodes connected to the first node. The first node is found by the combination of a *'Label'* and a *'Property'* value. Additionally to the here presented id as a node *'Property'*, each node has an internal ID that can be used to identify a specific node as well. This internal ID can be used for fast index lookup. Lastly, the nodes are returned using the variable name *'what'*, here coloured in red (Neo4j, Inc., 2023d).

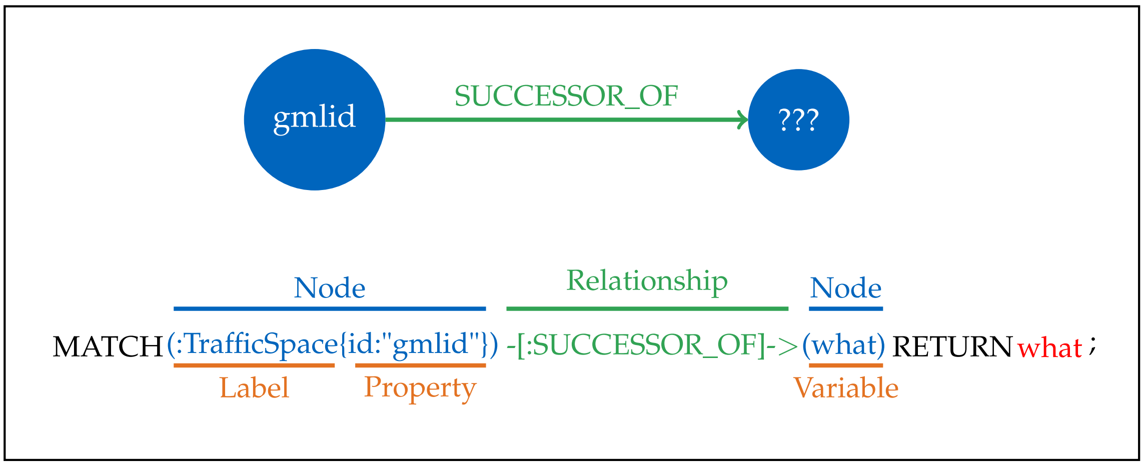

gmlid
SUCCESSOR_OF
???
Node
Relationship
Node
MATCH (:TrafficSpace{id:"gmlid"}) -[:SUCCESSOR_OF]->(what) RETURN what ;
Label
Property
Variable

Figure 2.10.: Exemplary Cypher query and its graph structure, original design (Neo4j, Inc., 2023d)

2.4.4. The Neo4j Extension APOC

APOC is an extension library for Neo4j enabling the use of additional procedures and functions. With Neo4j version 5 and upwards, APOC is split into two parts meaning that the previously community-driven library is now supported as the APOC-Core add-on library. Additionally, the community-developed extension still exists and provides additional functionality. The APOC-Core library contains about 450 functions and procedures, while the extended version contains another 50 procedures (Neo4j, Inc., 2023a). The installation process is already integrated into the Neo4j Desktop version and the plugin can be installed with a single click (Neo4j, Inc., 2023b). Especially interesting for the thesis are the path-finding procedures provided by APOC. These are the shortest path algorithms Dijkstra and A* (Dechter & Pearl, 1985; Dijkstra, 1959; Neo4j, Inc., 2023k).

2.5. Spatial Indexing - Kd-Tree

In order to find the geometrical nearest element in the graph database to a point, the data structure has to be indexed. For finding nearest neighbours the k-d tree data structure is considered. The k-d tree was defined by Friedman et al. in 1977. A k-d tree is a generalization of the binary tree. It is used for sorting and searching. Each node represents a subset of the data. Additionally, each node has two children or successors if it is not the final node of a branch. The final nodes called leaves or terminal nodes represent the smallest subset of the data. (Friedman et al., 1977). When the data is stored in a k-d tree structure the closest records can be queried efficiently. This reduces computation to find the n closest matches for a given record. In the following the search algorithm is described as a recursive procedure.

The process starts with a node to examine. First, the root of the tree is handed over. A partition divides the current subtile at each node. This way a lower and upper limit is recorded for each record of the two new subsets. These limits define a cell. Subsets of nodes deeper in the tree have a smaller cell volume. If the investigated node is terminal, then all remaining elements in the cell are examined. During the search, a list of the n closest records encountered as well as their dissimilarity to the query is obtained as a priority queue. This list is updated whenever a closer match is found. If the investigated node is not terminal the procedure is called recursively. A so-called "boundsoverlap-ball" test is made to determine which subset to take into account. "If the bounds-overlap-ball test fails, then none of the records on the opposite side of the partition can be among the m records closest to the query record" (Friedman et al., 1977). If the bounds do overlap, then the records must be considered and the process is continued recursively for the node representing the subset (Friedman et al., 1977).

With this search, it is possible to identify n closest points to a given point. This can be used to find the closest TrafficSpaces to one another. Alternatively, other elements can be searched in the vicinity of a TrafficSpace or the geometry of a TrafficSpace can be used to find the closest TrafficSpace to a given coordinate.

2.6. Graph-based Routing Algorithms

2.6.1. Pattern Matching

One of the methods to obtain information from the graph database is using pattern matching combined with the Cypher query language. This way it is possible to search for specific patterns in the graph or to find connections between nodes. Furthermore, it is possible to use pattern matching to query information about a single node or similar nodes. Pattern matching is essential during **MATCH**, **CREATE**, and **MERGE** operations.

Pattern matching works for nodes as well as relationships. It is further possible to find path patterns that consist of a sequence of nodes and relationships. Additionally, it is possible to use equijoin operations which require more than one node or relationship of the path to be the same. This can be used to match cycles. Another matching option is called quantified path patterns. Those can be used, for example, when searching for all nodes that can be reached from an anchor node. Other usage possibilities are finding all paths connecting two nodes or traversing hierarchy with differing depths (Neo4j, Inc., 2023c). Lastly, there is a shortest path functionality that can be used to find the shortest path between two nodes. This is determined by the number of relationships between the nodes. If two or more paths with the same minimum length are found, one is picked arbitrarily. There is also a variety to explore all the shortest paths between two nodes (Neo4j, Inc., 2023c). The matching works exactly as shown in the exemplary Cypher query seen in the previous figure 2.10. To find a node a **MATCH** operator is used. This is followed by the syntax to describe the structure that shall be searched for in the database. A simple MATCH includes a node which can be referenced by a variable *n*. Lastly, the node is returned using the **RETURN** operator.

```
MATCH (n) RETURN n;
```

Code 2.4: Basic Cypher MATCH returning all nodes.

A node can be additionally matched by a label or the value of a property. This is shown in the following example. Here, the node with the label "Street" and the property "id" with the value "UUID_Street" is matched. The node is referenced by the variable *n* and returned using the RETURN operator.

```
MATCH (n:Street WHERE n.id="UUID_Street") RETURN n;
```

Code 2.5: Cypher matching with a label and property.

Furthermore, it is possible to match a relationship or a path between nodes. This can be used to access information stored throughout the graph. The following example shows how to match a relationship between two nodes. A geometry node and a node containing the point geometry. The relationship POINT_OF_GEOMETRY is referenced by the variable r and returned using the RETURN operator. This could be used to find all points of a geometry.

```
MATCH (g:Geometry)-[r:POINT_OF_GEOMETRY]->(p:Point) RETURN r;
```

Code 2.6: Cypher matching a relationship.

Additionally, it is possible to match a pattern over multiple nodes and relationships which is shown in the following example. The pattern shall find all points that make up the geometry of a street element.

```
MATCH (s:Street)-[:HAS_GEOMETRY]->(g:Geometry)-[:
 POINT_OF_GEOMETRY]->(p:Point) RETURN p;
```

Code 2.7: Cypher matching a path.

More complex patterns can be matched as well. An extensive explanation of the pattern matching syntax can be found in the Neo4j documentation (Neo4j, Inc., 2023c).

2.6.2. Shortest Path Algorithms

The basic problem is given a graph $G = (V, E)$ how can the shortest path from a given source vertex $s \in V$ to any other vertex $v \in V$ be found? This problem can be found in some variants. Single-destination shortest-paths (1), single-pair shortest path (2), and all-pairs shortest-paths (3) problem. The single-destination shortest-paths problem finds a shortest path to a given destination vertex d from each vertex v. If the direction of each edge in the graph is reversed the problem can be reduced to a single-source problem. Secondly, the single-pair shortest-path problem finds the shortest path from a vertex u to a vertex v. "All known algorithms for this problem have the same worst-case asymptotic running time as the best single-source algorithms" (Cormen et al., 2009). Lastly, the all-pairs shortest-paths problem finds shortest paths for all possible pairs of vertices. Although this problem can be solved by running single-source algorithms once for each vertex pair, it usually can be solved faster (Cormen et al., 2009).

The shortest path problem can be defined for directed, undirected or mixed graphs. In the considered problem for this thesis, the single-pair shortest path problem is most relevant, while other combinations like the all-pairs shortest path exist as well. The problem originates in graph theory where the breadth-first search is usable to find the shortest path to all nodes in a graph with unit edge costs. For non-negative edge costs the Dijkstra algorithm can be used to find the shortest paths.

Dijkstra's Algorithm

Dijkstra's algorithm uses a breadth-first search and searches the graph in order of non-decreasing distance. Thus, Dijkstra is a greedy algorithm. It is used to find the shortest path from a single source node to all other nodes in a graph with non-negative edge costs. While it has improved runtimes compared to the Bellman-Ford algorithm, Dijkstra does not allow negative edge weights. The algorithm works by maintaining a set of nodes for which the shortest path is known. During the procedure, the algorithm iteratively adds the node with the shortest path to the set of known nodes. When the destination node is added to the set of known nodes the algorithm terminates. Furthermore, the expected number of decreaseKey operations is $O(|V|^2)$ or $O(|E| + |V| \log |V|)$. The procedure is described in the following pseudocode (Dijkstra, 1959; Kurt Mehlhorn & Peter Sanders, 2008; Melanie Herzog et al., 2013; Simic, 2021) :

Algorithm 1: Dijkstra Algorithm

```
Function Dijkstra(($s$ : $NodeId$) : $NodeArray \times NodeArray$)
  $d = \langle \infty, \dots, \infty \rangle$ : $NodeArray$ of $\mathbb{R} \cup \{\infty\}$
  $parent = \langle \perp, \dots, \perp \rangle$ : $NodeArray$ of $NodeId$
  $parent[s] := s$
  $Q$ : $NodePQ$
  $d[s] := 0$; $Q.insert(s)$
  while $Q \neq \emptyset$ do
      $u := Q.deleteMin$
      foreach edge $e = (u, v) \in E$ do if $d[u] + c(e) < d[v]$ then
          $d[v] := d[u] + c(e)$
          $parent[v] := u$
          if $v \in Q$ then $Q.decreaseKey(v)$
          else $Q.insert(v)$
  return (d,parent)
```

Adapted from (Kurt Mehlhorn & Peter Sanders, 2008)

A* Algorithm

Compared to the Dijkstra algorithm, the A* algorithm only computes the shortest path between two nodes in a graph. It was first presented by Hart et al. (Hart et al., 1968) with the intention of solving the shortest path problem for a single start and destination. If all shortest paths are required, the algorithm must be executed several times. The algorithm uses additional information to guide the search towards the destination node. In the context of geospatial data, the distance to the destination node can be used as a heuristic. Typically, the Euclidean distance is chosen as it can be computed using two pairs of coordinates and is a lower bound for the actual distance. Additionally, the A* algorithm uses a set of candidate nodes to be processed next, similar to the Dijkstra algorithm. A best-first approach is applied to select the next node from the list for further processing and expansion. Where the selection of the next node to process in the Dijkstra algorithm is based on the distance from the source node, the A* algorithm uses an informed search procedure that also includes a heuristic to include information about the destination node. Thus, the search has a forward-looking part which is guided towards the destination node. Especially in the geospatial domain information is available to improve the performance of shortest path searches, making the algorithm an interesting candidate for network analysis. The algorithm has a time complexity of $O(|V| + |E|)$ (Dechter & Pearl, 1985; Melanie Herzog et al., 2013; Simic, 2021). A detailed comparison between Dijkstra and A* algorithms on street networks has been done by Zeng and Church (Zeng & Church, 2009).

Overview of Shortest Path Algorithms

The following table 2.4 provides an overview of additional prominent shortest-path algorithms.

Algorithm	Time Complexity	Negative Edge Weights	Negative Cycles	Description
Dijkstra	$O(\lvert V\rvert^2)$ or $O(\lvert E\rvert + \lvert V\rvert \log \lvert V\rvert)$	No	No	From a single source to all other nodes in a graph with non-negative edge weights.
A*	$O(\lvert V\rvert + \lvert E\rvert)$	No	No	From a single source to a single destination in a graph with non-negative edge weights.
Bellman-Ford	$O(\lvert V\rvert\lvert E\rvert)$	Yes	Yes	From a single source to all other nodes in a graph with negative edge weights.
Floyd-Warshall	$O(\lvert V\rvert^3)$	Yes	No	Shortest path between all pairs of nodes in a graph with negative edge weights.
Johnson	$O(\lvert V\rvert\lvert E\rvert + \lvert V\rvert^2 \log \lvert V\rvert)$	Yes	No	Shortest path between all pairs of nodes in a graph with negative edge weights.

Dijkstra, A*: (Simic, 2021), Bellman-Ford: (Sryheni, 2020), Floyd-Warshall: (Datta, 2020), Johnson: (Engibaryan, 2023)

Table 2.4.: Overview of shortest-path algorithms: Dijkstra, A*, Bellman-Ford, Floyd-Warshall, and Johnson (The runtime complexity of A* highly depends on the chosen heuristic).

More advanced algorithms and concepts to perform shortest-path searches exist. For a deeper understanding of the topic the book "Algorithmics of Large and Complex Networks" provides a good base (Delling et al., 2009).

2.7. Multimodal Networks

2.7.1. Definitions

The word multimodal appears across many different domains and with varying meanings depending on the word it relates to. Examples that lie close to the thesis topic are multimodal routing, multimodal transport and multimodal transportation, multimodal navigation or multimodal navigation systems. While the term multimodal navigation is rarely defined in the literature, there is a definition in the context of robot navigation. Here, multimodal navigation is defined as "[...] the implementation of several different modes of navigation that enable a robot to move around an environment with different configurations, according to the task in hand" (Bettencourt & Lima, 2021). While this defines different modes as different configurations it is not exactly the same as having a variety of options and choosing the optimal combination of those. Thus it might be insightful to take a look at further multimodal definitions. Multimodal Navigation Systems, for example, are defined as "Navigation Systems that use different modes to communicate with the user [...]" (Kuriakose et al., 2020). Additionally, Multimodal Transportation is defined as "[a] transport system operated by One carrier with more than one mode of transport under the control or ownership of One Operator" (Wisetruangrot, 2020). In this context also international multimodal transport is defined. "'International multimodal transport' means the carriage of goods by at least two different modes of transport on the basis of a multimodal transport contract from a place in one country at which the goods are taken in charge by the multimodal transport operator to a place designated for delivery situated in a different country" (United Nations, 1980). Another source defines Multimodal Transportation as "[...] a kind of transportation that uses at least two transportation manners (e.g. air, inland water, ocean, rail and road) while delivering goods from origins to destinations. It is explicitly different from the ordinary transportation manner which adopts only one single manner with the participation of a freight forwarder" (Xiong & Wang, 2014). All these definitions contain the multimodal aspect, nevertheless, the definitions vary and are not sufficient for the thesis. Thus, the need for defining multimodal navigation in the context of this thesis arises.

***Multimodal navigation** can be defined as the process of navigation using multiple, at least two different, modes of transportation for one trip.*

Within this thesis, the routing part of a navigation system is the main focus. A multimodal routing analysis might not always return a multimodal result if a single mode of transport can fulfil the search requirements better than a combination of multiple ones.

Therefore, some results can seem like typical monomodal navigation routes whilst the underlying logic provides multimodal analysis capabilities.

2.7.2. General Concepts

As the name suggests, multimodal networks consist of multiple modalities - in this case transportation types - combined in a network. These transportation types include walking, bicycle, car, train, tram, metro, and bus, as well as rental car, bike, scooter, and taxi, aeroplane, and ship/ferry, etc. In order to represent the real world in a simplified way, these transportation modes make up their own network of locations that can be travelled to, e.g., in the case of a car, the road network, or for a train the train stations. For multimodal networks, those subnetworks are combined to generate a routing supernetwork that represents the real world more closely. To model such a network it might be helpful to have a top-down view of the situation; one network which connects the start point and all possible destinations. While parts of the network might be covered by multiple transportation types, e.g., a road for walking, cycling and driving, others can be reached via a single mode only. Here, a differentiation can be made between modes of transportation. Some require a fixed location, like bus stops, train stations, harbours or airports, while others can in principle access every place, like cars, bikes or pedestrians. In-between are transportation modes like rental services or taxis which require a certain start point but can be more flexible in terms of destinations, e.g., e-scooters that can be left anywhere within a certain service region. However, as there are still many offers which require a fixed start and end location for rental services after a certain amount of time, e.g., car or bike rental, this category is more shifted to the fixed location category. Another similar classification was introduced by Zhang et al. In this classification, transportation modes are grouped into public and private networks, as well as functional and physical representations. It is described as follows: "Private networks offer continuous service at any time associated with both physical nodes and physical links. On the other hand, public transportation networks offer discrete services according to timetables whereby physical nodes (e.g. stops, stations) are visible while physical links are usually invisible. Therefore, the functional view is suitable for modeling the multimodal transportation network" (Zhang et al., 2011). Following Zhang's structure, after having a functional representation of the different transportation types, a suitable model for later multi-criterion route analysis must be found. For the private or reach everywhere category the physical structure can be used. This means using the geographic correct representation of the network. Public or fixed location transport modes however are not only restricted in terms of geographic availability but also restricted to certain (usage) times defined by service timetables, opening hours or reservations. Thus, complicating the modelling. For modelling this, two common solutions can be identified in the literature according to Zhang. The first one is a connection cost function, linking physical locations and mapping the timetable to the connection as weight attributes. The second one is generating time-dependent event nodes for arrival and departure.

This way a supernetwork is generated by combining the subnetworks into one large network containing the different transportation types and their requirements. Lastly, to create a network capable of performing multi-criterion analysis, the weights/costs of the subnetworks have to be homogenized. This can be achieved by choosing a common attribute in all networks, like traversal time in seconds. Alternatively, a suitable weight must be calculated. Sometimes introducing assumptions is necessary. Multi-criteria weights further combine different costs into one weight, e.g., travel time, cost, and comfort. There are also existing index calculations that can be used as a multi-criterion weight, e.g., the walkability index or Bicycle Level of Service. The chosen weights of this thesis are explained in more detail in chapter 3. This final network will be further referred to as a routing network. Additionally, a multicriteria routing algorithm for multimodal networks is proposed by Dib et al. (Dib et al., 2017). However, this will not be integrated into the thesis, as basic shortest-path algorithms are used.

2.7.3. Switch Nodes

As could be seen in the previous section, modelling and facilitating routing on multimodal networks is a complex task. While there are different options for modelling time dependencies and changing costs of connections, there are also multiple ways to include transportation mode changes, transits and physical connections between the different transportation networks. One of the methods to connect two subnetworks is the switch point introduced by Lu Liu (Liu, 2011). This concept of representing concrete intermodal facilities like P+R lots or public transit stations in an abstract manner can be adapted to switch nodes. These switch nodes represent physical locations where the transit takes place, e.g., the parking lot with a node representation for the individual parking space where the transportation mode is changed from car to pedestrian or vice versa (Liu, 2011). Similarly, bike routing stations or platforms of train stations can be modelled. With this approach, a more detailed representation of the actual transit is possible allowing to accurately include walking which could include indoor navigation in large train stations, airports, or parking garages. This eliminates the mentioned issue of many-to-many connections where, e.g., a parking lot has multiple entry and exit points which require separate weights when the physical location is moved to the abstract transit location parking lot rather than using the actual parking spaces. Furthermore, default shortest path algorithms can be used as the switch location is represented as a node in the network and connections to the corresponding subnetworks. Attributes with weight information can be introduced to the connecting edge. Lastly, it is possible to connect multiple adjacent nodes of different subnetworks, if, for example, the parking space is adjacent to a train station - with no explicit interior representation - and a sidewalk section. Another mentioned switching action, intro-modal switches can be realized in the same manner, whether by linking the physical locations of two subway lines, e.g., U1 and U2 of Munich, or by linking their timetable entries at the station according to feasible transit times. By updating the weight connecting the switch node to the subnetworks, it is possible to model dynamic data, e.g., if the parking space is available or if higher waiting times are expected due to a delayed train (Liu, 2011).

3. Methodology

3.1. Pre-processing: Data Preparation and Requirements

3.1.1. Network Structure

In order to utilize shortest path algorithms, such as A*, the 3D modelling of the CityGML data shall be reduced into a simpler routing graph. This graph consists of nodes, representing CityGML object instances and edges, which connect those objects using their parent/child relationship, XLinks, as well as their other semantic and spatial connections. Thus, it shall be possible to store the geometry and semantic information while allowing the routing algorithms to use a subset of the entire CityGML data creating a network to perform connection and shortest-path analyses. In particular, the routing network shall contain shortcuts for representing complex relationships in the CityGML data that are not modelled explicitly or stem from mapping the CityGML data to the graph database. Furthermore, a network for each transportation mode is created. These networks are connected using switch nodes to transfer between two travel modes. The nodes contain important information such as the type of transportation mode, or they have a relationship to further nodes containing such information. Those attribute nodes are not part of the routing process but provide additional information. The edges of the routing network contain information about the weights which will be addressed in the next section. Depending on the task and user selection, the weights may need to be combined to retrieve a final weight for the traversal costs of the edge. In the implementation, only pre-processed weights will be used and no on-the-fly calculations are used. An exemplary part of the network design is shown by figure 3.1

In this instance, N_1, N_3 and N_4 serve as switch nodes connecting different networks. Attribute nodes A_0–A_2 can be directly connected to the routing network nodes N_i or have a node relationship pattern that is similar to access further data of each node. For example, the geometry values are connected via a chain of relationships to the network nodes.

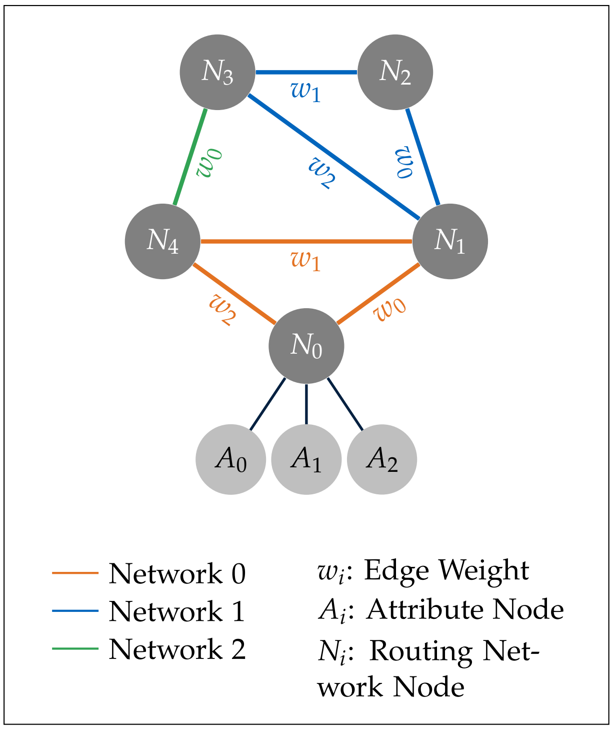


Figure 3.1.: Exemplary network structure with three sub-networks for different transportation modes and one network node with attached attribute nodes.

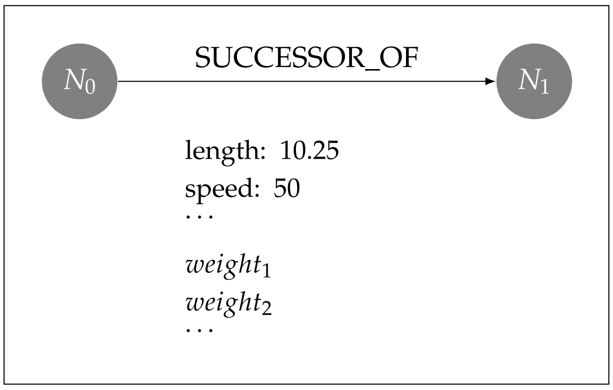


Figure 3.2.: Exemplary mapping of information as SUCCESSOR_OF relationship attributes

Figure 3.2 shows the available information present at a *SUCCESSOR_OF* relationship that can be used for routing. This consists of information derived from geometry as well as semantic information describing the rules and properties of the segment. During the pre-processing, information is added to the relationships. Some of the data can be directly used during the routing process, while others require a special weight to make the data compatible with the requirements of the shortest path algorithms. This includes, e.g., no negative values for using the Dijkstra algorithm. The first step is to extract the needed information. As the routing graph is a subset of the CityGML data, the information is not directly available at the relationships. Thus, the pre-processing step defines which information can be used in the later shortest path analysis. In the second step, the routing weights are added. These pre-calculated values serve as a metric to evaluate the path during the traversal of the shortest path search and can combine multiple information into a single value. Before any weights are calculated, it is necessary to check if all relevant connections within the dataset exist. The process of creating the routing network can be split into three steps. First, check and add missing connections. Second, extract useful information for weights. Third, calculate weights and add them to the connecting edges. As no automated tool exists to generate missing links and manual editing of the dataset will take a long time, the thesis shows the introduction of missing neighbouring lane connections using additional information. Thus, the main challenges are to generate the routing graph and extract useful information as well as enhance the graph with additional information. Here, especially finding or calculating usable weights is important. Therefore, challenges in the process of finding usable weights for the routing graph are presented in the following section 3.1.2. Once adequate weights have been found those can be added to the relationships.

3.1.2. Weights

To enable the use of shortest path-finding algorithms, the edges of the graph need to be weighted. For this purpose, a variety of different factors can be used. In this section, some factors are discussed and the ones used in this thesis are explained. The following list gives an overview of factors that could be used to weigh the edges of the graph:

- Length/distance
- Inclination
- Height and width ($\rightarrow$ volume of a segment)
- Speed (minimum, maximum, average)
- Time (travel time, waiting time), average traversal time
- Surface material
- Usability (e.g., for pedestrians, cyclists, cars, etc.)

- Accessibility (e.g., for people with disabilities)
- Availability, e.g., restricted due to construction sites or accidents
- Traffic flow
- Weather conditions, e.g., rain, snow, ice
- CO2 emissions and air/noise pollution
- Time of day and day of the week
- Stop signs and traffic lights
- Street types, e.g., highway, main street, side street, or toll road
- Predetermined indices such as the walkability index or BLOS
- Transport mode, e.g., car, bicycle, pedestrian, bus, train, etc.
- Transit times for transfers and changes of transportation modes
- Travel budget and expenses
- User preferences, e.g., avoiding certain road types or areas
- Lowered curb stones and other obstacles
- Maximum weight or size of the vehicle/cargo

From these factors, some general categories can be derived. These categories are the following:

1. Geometry, e.g., distance/length or inclination
2. Time, e.g., travel time or waiting time
3. Usability, e.g., blocked or suitable only for certain transportation modes
4. Convenience or user preferences, e.g., avoiding certain road types and areas or allowing higher travel distances/times for more convenient routes

Additionally, some of the weights are static while others might have a real-time or time-dependent variation. Long-time construction sites are also considered as static weights, while real-time weights might include the current traffic flow, or road obstructions due to short-term construction sites or accidents. Lastly, time-dependent weights might include speed limits or traffic lights which are only active during certain times of the day or days of the week. The following sections will discuss some factors in more detail and explain how they can be derived from the data and how they can be introduced to the routing graph.

Length

The first weight uses a rather simple weight, the length of a street segment which is represented by an edge in the graph. To use this weight one can look for a corresponding value in the dataset. If no length value for individual segments is given it can be calculated. In this case, there are different options. For the length weights in this thesis, two length calculations were chosen. Both calculations use the TrafficSpace geometry to calculate the length of a segment. The first and simpler calculation uses the Euclidean distance between the first and last point of the geometry. As the TrafficSpace geometry in the lane granularity consists of an ordered list of points, the start and end points can be easily identified. This does only apply to a line representation of the TrafficSpace. Volumetric TrafficSpaces require a modified calculation. For the dataset, the points are using 3D coordinates in a UTM coordinate system. Therefore, the Euclidean distance is calculated in 3D space and the result of the formula returns the distance in meters. If a coordinate reference system using a different unit, for example, the geographic coordinate system WGS84 is used, the coordinates must be projected to a coordinate system using meters as the unit.

Generally, the Euclidean distance for n-dimensional points is calculated by the following formula:

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2 + (z_2 - z_1)^2 + ... + (n_2 - n_1)^2} \tag{3.1}$$

As the TrafficSpace geometry is a 3D geometry, the formula can be simplified to the following:

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2 + (z_2 - z_1)^2} \tag{3.2}$$

Then the weight for a segment is the calculated length $w_{length} = d$. This weight is sufficient to find the shortest path between two network nodes. However, if the TrafficSpace object does not have a linear geometry, the Euclidean distance is not the actual length of the segment. Therefore, a second length calculation can be carried out. This calculation uses each point available in the TrafficSpace geometry to calculate the length of a segment. The final length is calculated by summing up the Euclidean distance between each consecutive pair of points in the ordered geometry list. The formula for the calculation of the length of a segment is as follows:

$$d_i = \sqrt{(x_{i+1} - x_i)^2 + (y_{i+1} - y_i)^2 + (z_{i+1} - z_i)^2} \tag{3.3}$$

Then the individual distances are summed up to get the final length of the segment:

$$w_{length} = \sum_{i=0}^{n-1} d_i \tag{3.4}$$

where n is the number of points in the TrafficSpace geometry.

Inclination

For the inclination, the height difference between the start and end point of a segment is used. The height difference is calculated by subtracting the height of the start point from the height of the endpoint. To normalize the values, the height difference is divided by the length of the segment. The formula for the calculation of the inclination is as follows:

$$w_{inclination} = \frac{z_{end} - z_{start}}{d} + 100 \tag{3.5}$$

For this process, the TrafficSpace geometry is used. The start and end points are identified as is the case in the calculation of the Euclidean distance and the height values are extracted. The length of the segment is calculated using the Euclidean distance between the start and end points as described in the previous section. The result is the inclination of the segment in percent. To make the weight compliant with the requirement of no negative values, the value is increased by 100. This way the weight is always positive and the inclination is not neglected. If the segments are large and a more accurate value for the height changes is required, the inclination must be calculated for each consecutive point pair or a selection of points that ensure a higher sampling rate with smaller distances between the used points to calculate the inclination. Figure 3.3 shows the issue of a low sampling rate with a long segment and height changes in between. The terrain between the two points cannot be represented accurately. Any height variations between those height values are ignored.

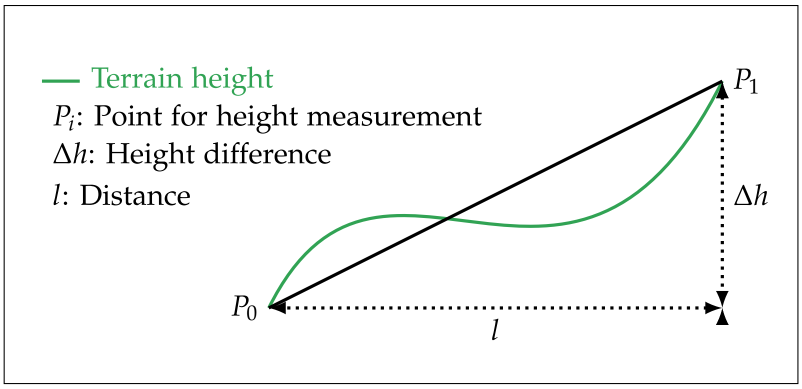

— Terrain height
P_i: Point for height measurement
Δh: Height difference
l: Distance
P_1
Δh
P_0
l

Figure 3.3.: Low sampling rate: The two points cannot capture terrain changes in between. Additional height measurement information is needed to capture the actual surface.

Lane Width

To derive such information, either the width of a line can be available or it must be calculated. To calculate the width of a lane, the ordered geometry of a *TrafficArea* object is used to find point pairs and calculate the distance between respective points, which represents the width of the segment. This can be used if the TrafficSpace is of granularity lane and has an attached *TrafficArea* object. The *TrafficArea* object is used to represent the surface of a lane. For granularity area or way, the TrafficSpace can be used directly to derive the width, if they are represented using a plane or volume. Additionally, a centre line can be used. For the granularity type lane, the *TrafficArea* is used. Each *TrafficArea* object is represented by *SurfaceMembers*. One *SurfaceMember* contains a list of ordered points. The list of ordered points contains the start point two times, as the first and the last point. For the following considerations and calculations, the duplicate last point is removed from the list.

A point pair is the first and the last point of the *TrafficArea SurfaceMember* geometry. The next point pair would be the second and the second last point of the geometry, and so on.

The following formula describes the point pair calculation:

$$d_i = \sqrt{(x_i - x_{n-i})^2 + (y_i - y_{n-i})^2 + (z_i - z_{n-i})^2} \tag{3.6}$$

where n is the number of points in the *SurfaceMember* geometry.

For each point pair the distance is calculated and the minimum width is stored. This process is repeated again for each *SurfaceMember* geometry of the *TrafficArea* object. The minimum width of all *SurfaceMember* geometries is the width of the segment and will be assigned to the TrafficSpace node.

$$w_{width} = \min_{i=0}^{n-1} d_i \tag{3.7}$$

with n being the number of *SurfaceMember* geometries.

This way, small *TrafficArea*s will not be used or be used rather as a last resort (depending on the weight implementation). A weight for the width could use the calculated width and be compared to a predefined list of widths derived from regulations for the use and or construction of roads.

In the implementation, the following weight for the width will be used: $w_{min\ width} = \frac{1}{min(width)}$ If the width is large it will have a smaller weight, if the width is small it will have a higher weight. Thus, wider road segments are prioritized over narrow elements.

Street Type

This weight can be used to prioritize certain streets over others. Such a value can be represented by a string, e.g., main street, highway, etc. The string should be mapped to an integer or float value to be used as a weight. A higher value means a not favoured street type and a lower value means a good street type. For the data used in this thesis, no information about the street type is available. It could be derived by using the street types available in OSM and mapping them to the corresponding edges of the graph.

Material and Friction

The surface material can be provided in the CityGML model via the *surfaceMaterial* attribute of the *(Auxiliary)TrafficArea* object. The value is connected to a code list. In the test dataset, a surface material attribute is also available as a generic attribute from the OpenDRIVE data. This material should be mapped to an integer or float value to be used as a weight. A higher value means a bad surface material and a lower value means a good surface material. Alternatively, a weight for the surface roughness could be used. This information is also available as a generic attribute. It is represented as a float value and can be used directly as a weight.

Intersections - Traffic Lights and Street Signs

Driving for longer periods on a road without a stop is beneficial. However, this is hard to model. To tackle this, the number of intersections, the availability of traffic lights and traffic signs granting a stop-free path could be analysed. First, the number of ingoing streets could be used as a weight. Then, the availability of traffic lights and traffic signs could be used as a weight. If there is a traffic light, a standard value for waiting time could be added, for street signs demanding a stop, a waiting time could be added to the weight as well, whilst a street sign granting a right of way would not increase the weight or could be used to reduce the weight. In general, a higher value could be introduced for edges leading to a 'busy' intersection with traffic lights or a stop sign. This weight information is most useful to improve travel time predictions.

Time

The time weight is a weight that can be used in a simple way or can be expanded to a complex weight. Starting with the simple version, the time weight is derived by the length of a segment combined with the average travel speed of the transportation mode used to traverse the edge. The formula for the calculation of the time weight is as follows:

$$w_{time} = \frac{d}{v} \tag{3.8}$$

where d is the length of the segment and v is the average travel speed of the transportation mode used.

For more accurate results, the time weight can be expanded to account for many factors. The following list shows some of the factors that could be used to improve the time weight:

- Maximum speed for each segment
- Average speed for each segment (derived by long-time observations)
- Type of street
- Transportation mode/vehicle
- Waiting times, e.g., for traffic lights and traffic signs
- Inclination of the segment
- Surface material
- Weather
- Current traffic situation
- Time of day and day of the week: determine rush hour

- CO2 emissions
- Noise pollution
- For multimodal routing: waiting times for transfers
- Directly measuring traversal times in real-time by observing current traffic

To break these factors down into some categories, we can identify the following: Speed, waiting times and real-time information. A weight for speed can combine information about the maximum speed allowed, the type of street and transportation mode/vehicle, inclination and surface material, as well as emissions and pollution reduction. The value can be determined by a formula that combines the different factors. However, some of these values are redundant. For example, the average speed already includes most values, as it describes the recorded speeds over a longer period. Also, the type of street could be used in case there is no maximum speed available. Combined with the transportation mode and some logic to take the time of day and the day of the week into account, an artificial value for an average travel speed could be derived. To make the prediction even more advanced, different waiting times can be introduced. These include waiting times at stops, waiting times at junctions with traffic lights or traffic signs demanding a stop, or transit times. Such data could be available from some recordings or can be estimated by combining a standard waiting time with factors such as the number of incoming and outgoing streets or the type of traffic sign. Lastly, real-time information can be used to improve the prediction. This will be discussed in the next section.

For weighting time weights the formula can combine the different factors.

$$w_{waiting} = \sum_{i=0}^{n-1} waitingtime_i \tag{3.9}$$

The final formula for the time weight:

$$w_{time} = \frac{d}{v_{avg}} + w_{waiting} + w_{realtime} \tag{3.10}$$

where d is the length of the segment, v_{avg} is the average travel speed of the transportation mode used, $w_{waiting}$ is the waiting time weight and $w_{realtime}$ is the real-time information weight.

An approximative function could combine the aforementioned elements in the following manner:

$$w_t = (\frac{d}{v} \cdot f_1 + \sum waiting\ times \cdot f_2 + delay_{traffic\ conditions} \cdot f_3) \tag{3.11}$$

with f_i factors to normalize and adjust the importance of the individual weight information.

Real-time Data - Traffic Information and Weather

Improving the time weight by using real-time data is a complex task. First, the data must be available. Second, the data must be processed and stored in real-time to be used for the routing process. Third, the weights derived from the real-time data must be combined with the other static weights to improve the prediction. Furthermore, the real-time data does not only include information on reduced speeds or longer travel times but also information on the availability, like blocked roads. Under the category of real-time data falls weather information, current traffic flow as well as short-term construction sites or accidents blocking lanes or completely blocking the road. The weather information delivers a rough estimation of the current travel conditions. For heavy rain or snowfall, the travel speed could be reduced. Information on the real traffic flow could be used to adjust the travel time for the corresponding segment, whilst information on construction sites and accidents could be used to block the corresponding lane or street. Alternatively, expected delays could be added to the time weight.

User Preferences

Lastly, there are some factors that could be interesting for certain user groups. These fall in the convenience or accessibility category. For example, a user might want to avoid streets with a certain inclination, a certain surface material (e.g., cobblestone), or want to use a route including lowered curbs. Such preferences could influence the weights that were previously mentioned or reduce the weights of certain segments to make them more attractive. Thus, for example, a longer route might be optimal because it is overall more convenient or accessible for the user. Finding suitable values for changing the weights is a difficult task and requires a lot of testing and user feedback. Therefore, this thesis will not focus on this topic. However, the implementation will show ways to extract information from CityGML to include such weights to later improve the routing process. An overview of potential information for weights for this category is given in the following list:

- Inclination
- Surface material
- Lowered curb stones and obstacles
- Width of the street/lanes
- Number of crossings
- Number of transfers/changes of transportation modes
- Predetermined indexes such as the walkability index or BLOS
- Avoiding certain road types, e.g., highways
- Avoiding certain areas, e.g., industrial areas
- Allowing a maximum distance for walking or any other transportation mode
- Reducing emissions
- Maximum budget for the route

Some of the mentioned points can be directly used as a weight, like the inclinations, while others can be used as static values to influence the weight calculation, e.g., information on the road type. However, some use cases also require a modification of the existing shortest path functions available in the graph database. This is the case if for example a maximum distance for a transportation type is given or certain elements shall be avoided. In this case, it is better to modify the algorithm and not consider these paths during the search process than to create multiple weight versions which lead to high storage usage and a more complex weight calculation. However, the option of creating a temporary routing graph with the desired weights could be an alternative as well and could be investigated in future work. Here only the available edges are used during the creation of the routing graph. This process will most likely come at the cost of a higher computation time as the network has to be changed for every routing query. Also, a mixed-use could be possible, a good example is the user preference to avoid toll roads. Here, two routing graphs could be created one containing toll roads and one without them. If the links for toll roads have a different name, it is even possible to differentiate the two networks in the graph database. This means that in Neo4j two versions of the shortest path function can be implemented, one using the toll road relationships and the other one ignoring them. Thus, it is possible to change the network structure without highly increasing the storage requirements to include this user preference. However, this approach might only be feasible for modelling certain user preferences.

Used Weights

Due to time restrictions, only some weights are selected for the implementation. These weights were chosen based on the following criteria:

- **Usability:** The weight should be typically used in navigation applications.
- **Intuitiveness:** The weight should be intuitive and easy to understand.
- **Availability:** The information for the weight should be available in the CityGML data.
- **Performance:** The weight should be simple to compute.

Thus, the following weights were chosen:

- **Distance:** A distance weight is typically used in navigation applications. It is intuitive and easy to understand. The distance between two points can be calculated using the Euclidean distance between the start and the end point of a TrafficSpace geometry. Additionally, a more accurate distance value can be calculated using all points of the TrafficSpace geometry.
- **Inclination:** The inclination weight is easy to understand and shows the issue of using possibly negative weights as well as utilizing 3D information.
- **Width:** While the width of a road segment is easy to understand, it requires the 2D surface information of the road segment which is not necessarily available in lane-based road networks. However, the width of a road segment can be calculated using the 2D geometry of TrafficAreas.
- **Speed:** Using speed limits to calculate the traversal time is challenging as the information is not necessarily and explicitly available in the CityGML data. However, it can be derived from CityFurniture objects such as traffic signs. This shows how implicit information can be used to calculate weights. Additionally, some assumptions combined with external information stored in generic attributes can help to fill in missing information.
- **Time:** This weight is typically available in navigation applications. It is intuitive and shows the combination of two different weights. However, it is dependent on the availability of speed limit information or other information related to the traversal time, like average travel speed for a transportation type, long-time observations or real-time measurement data.

3.1.3. Data Acquisition

The used datasets originate from street surveying projects in Ingolstadt and Grafing near Munich. The street space was originally modelled in the OpenDRIVE data standard. It was converted to CityGML using the r:trån conversion tool (Schwab, Benedikt et al., 2023). This CityGML data was then further mapped to nodes and relationships in the Neo4j graph database. These steps were already performed and are not part of this thesis. Thus, the CityGML structure is extended with additional information supported by the OpenDRIVE standard. Such additional data is stored as generic attributes. Similarly, other data sources, e.g., from OSM could be added via generic attributes to the respective CityGML objects. As this data is optional and not part of the CityGML standard, general functionality shall work without using the additional data. Nevertheless, as it provides further information, the generic attributes will be used in the implementation as well.

3.1.4. Input Data Structure

To understand some of the variables and the overall data structure of the graph database, it is necessary to know the data structure the data was originally modelled in and how the graph database represents the CityGML data structure. A more detailed look at these data structures will be taken in the following section.

As the final state of the data is represented in a Neo4j graph database, it is important to familiarize with the query language Cypher in order to get any information by performing queries on the database. For learning the fundamentals of Cypher, the official Neo4j documentation, as well as their introductory courses and example datasets can be helpful.

First, a general overview of the datasets used in this thesis is given in table 3.1. Some important remarks are that not all elements needed for a (multimodal) navigation application are available. Additionally, the data originates from OpenDRIVE. Therefore, the modelling focus lies on the representation of elements in the street space and the near surroundings. This means that they are primarily focused on a consistent network for cars. The tested datasets themselves cover a limited amount of different elements that are supported by the CityGML model. Thus, the datasets are not directly comparable. However, combined they cover a larger set of elements and thus are used in the implementation to test the routing process with different elements supported in the CityGML model.

The mapping process from CityGML to the graph representation was already explained in section 2.3. A routing network is made up of connected TrafficSpace nodes.

	Ingolstadt Intersection	Grafing	Grafing with Garage
Number of nodes*	421,244	6,428,260	7,681,023
Number of relationships*	425,561	6,574,711	7,826,259
Number of node labels*	66	80	86
Number of relationship types*	47	66	67
Graph database storage size*	787 MB	1.86 GB	2.75 GB
CityGML storage size	34 MB	535 MB	496 MB

Table 3.1.: Metadata of test datasets; Number of nodes, relationships and their types. Additionally, storage size comparison between CityGML and graph database (*after pre-processing).

These have connections to TrafficArea nodes. In the first version of the converted dataset, only the TrafficArea elements contained additional generic attributes. In the later versions, the information was added to the TrafficSpace objects and thus the TrafficSpace nodes as well. Connected to the TrafficSpace node is all the information stored within the TrafficSpace tag from the CityGML structure.

CityFurniture objects near the street were added to the nearest TrafficSpace road segment in a later version of the converted data as well. These are available via a *'relatedTo'* relationship connection and can be used to retrieve restrictions imposed by street signs. Additional information is also provided by AuxiliaryTrafficSpaces and AuxiliaryTrafficAreas. Analysing the structure of the database representation also helps in preparing some general-purpose Cypher query frameworks which can be used with slight modifications to extract information from the database. This is especially useful if the same query is used multiple times with different parameters. For example, when extracting attributes of a CityGML object which are stored as nodes connected to the node of interest via the same chain of relationships.

The fundamental structure of the CityGML data is still present in the graph database representation. Thus, it is important to extract a layer of data that can be used to perform shortest-path queries. Here, the successor and predecessor relationships of the TrafficSpace nodes are of particular interest, as those connections provide the basis to generate a routing network. Besides the linkages in a network, the weights of those edges are of interest if more complex shortest-path algorithms are used and one wants to represent real-world relationships more closely. As the graph database starts with no weights for its relationships those weights must be added for later use. Additionally, the need for extracting and calculating meaningful weights for the routing process arises. Possible weights have been discussed previously. However, the data structure of the graph database must be analysed to find the information needed to calculate the weights. This process is described in the next section. Lastly, some connections within the graph database are missing or are not modelled in a way that allows the use of the built-in shortest-path algorithms. Therefore, some additional relationships must be added to the graph database. This process is also described in the following section 3.1.5.

3.1.5. Preparation of the Neo4j Routing Network

As pointed out in the previous sections, there are some differences in the structure of the modelling of the street space to the requirements presented. To utilize the graph representation for navigation purposes some pre-processing is necessary to prepare the data for the routing functions which implement the different routing algorithms. This includes adding missing relationships as well as the mentioned weights/costs. Additionally, some networks must be "repaired" to use them to the same extent as their real-world counterpart.

Adding Predecessor and Successor Shortcuts

This step is necessary to use the *apoc.algo.dijkstra* or the *apoc.algo.astar* functions with the correct order of the connecting relationships between the two TrafficSpace elements that are connected via a predecessor and or successor relationship. As the *apoc.algo.dijkstra* function only considers the relationships given as a parameter, it is necessary to add the predecessor and successor connections as a single relationship to the graph database. This is necessary as otherwise relationships connecting information rather than actual connections of traffic infrastructure, can be used by the shortest path function. A result of this could be a shorter route using metadata connections rather than the physically available relationships between TrafficSpaces. This is done by the following steps. The Cypher query finds each pair of nodes matching the pattern of the relationships used to represent the successor or predecessor connection and adds a new relationship *"SUCCESSOR_OF"* or *"PREDECESSOR_OF"* which acts as a shortcut to traverse the TrafficSpace nodes. Figure 3.4 visualizes this process. The result is a graph database containing the original relationships as well as the new relationships. The new relationships are further used and will get the weights for the routing process. While the original connection chain is not used in the further application it is still available in the graph database for consistency. The following Cypher query exemplarily shows the successor relationship creation:

```
  // Create Shortcut 4 TrafficSpace SUCCESSORS
  MATCH (a:`org.citygml4j.core.model.transportation.
TrafficSpace`)-[:successors]-()-[:elementData]-()-[:
ARRAY_MEMBER]-()-[:object]-(b:`org.citygml4j.core.model.
transportation.TrafficSpace`) CREATE (a)-[:SUCCESSOR_OF]->(b)
;
```

Code 3.1: Cypher query to create the successor relationship shortcut SUCCESSOR_OF.

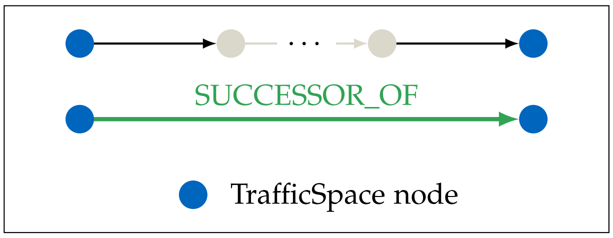


Figure 3.4.: A shortcut for a successor connection chain is represented by a single SUCCESSOR_OF relationship.

First, two TrafficSpace nodes are matched that are connected via the successor chain. Then a new *SUCCESSOR_OF* relationship is added to connect both nodes.

Introducing Lane Change Relationship

Adding a relationship to connect neighbouring lane segments is more challenging as it requires the combination of multiple attributes stored in nodes connected to the TrafficArea nodes which again are connected to TrafficSpace nodes. To achieve this automatically, further analysis in Python is necessary. For retrieving the data from the graph database, the Neo4j Python driver was used. In order to reduce the number of comparisons and thus the memory usage the procedure of finding neighbouring lanes is done separately for each road section. This is done by the following steps:

1. Find all TrafficSpace nodes of a road segment
2. Find all TrafficArea nodes connected to the TrafficSpace nodes
3. Find needed attributes connected to the TrafficArea nodes
4. Use logic to find neighbouring lanes - by comparing IDs (lane and segment) stored as generic attributes

5. Add *"NEIGHBOURS_LANE"* relationship between each pair of neighbouring TrafficSpace nodes

The logic to find neighbouring lanes is based on the following assumptions:

1. The lane segments have the same road ID and the same section ID, as well as the same type of lane (e.g. *'DRIVING'*, *'BICYCLE'*, *'SIDEWALK'*)
2. The lane segments are just one value apart in the lane ID (e.g. 1 and 2, 2 and 3, 3 and 4, etc.)
3. If another element is between the lane segments, e.g., a marking, the lane segments are connected as long as they have the same road ID, section ID, and lane type. This allows connecting lanes that require crossing a marking or other elements in order to change lanes, like biking lanes between a straight and a turning lane.

As the road section contains the lanes in both directions, a differentiation is made to not mix lanes going in different directions of the road. This is done by checking if the lane ID is positive or negative. Alternatively, it is also possible to add a second relationship to connect the two neighbouring lanes in opposite directions. This would allow a temporary usage of a normally prohibited lane, e.g., to pass an accident, for temporary construction sites or other road blockages. However, this is not fully implemented. Also, it is especially important to use the special connection not for regular routing as it leads to false routes and to potentially dangerous situations in real-world scenarios.

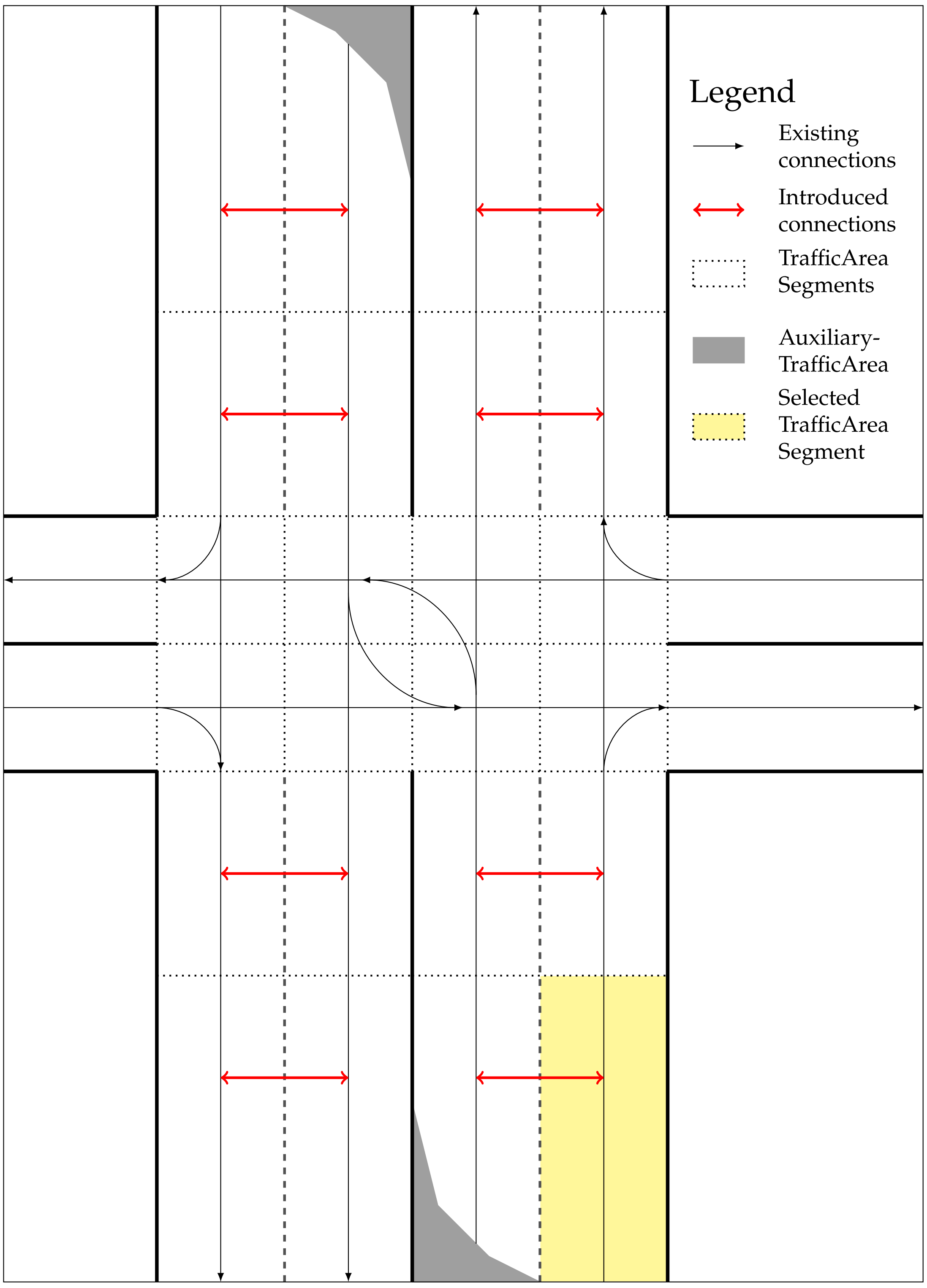


Figure 3.5.: Exemplary network, based on the CityGML representation with introduced connections in the graph representation in red.

Street Crossing Relationships

For modelling the pedestrian network, not only the present sidewalks are of importance, but also the ability to cross the street and the possibility to use elements designated to other transportation types in order to get to the next pedestrian segment. As the pedestrian or walking transportation type is very flexible and the base of linking other transportation modes, it is important to have a comprehensive network allowing the representation of movement patterns as accurately as possible to get realistic results. However, here the typical freedom of movement provides challenges when trying to implement a strictly line-based network. For example, when modelling a square a person can walk freely on this plane which is difficult to model by a node representation. While this could be solved through a fine grid layer over the square or by simply connecting ingoing and outgoing connections to the square another issue arises in the dataset. As the individual geometries are linked via successor and predecessor connections to form a continuous network, many natural walking paths are not represented. Only neighbour spaces along the connected elements can be followed. The successor and predecessor connections additionally follow the direction of the reference street, thus there is no perpendicular connection between pedestrian walkways. This makes the square issue rather difficult to solve without some greater modification of the test datasets. Thus, this will not further be explored in this thesis. Figure 3.6 shows the well-connected *'DRIVING'* and the disrupted *'SIDEWALK'* TrafficSpace connections.

This linking process leads to another issue for the modelling of the pedestrian subnetwork. The test datasets only contain sidewalks that shall be regarded as traffic areas for pedestrian movement. This however fragments the pedestrian network into many unconnected smaller networks that follow a sidewalk until it is necessary to cross a road or other type of surface not categorized as a sidewalk. If the need to cross a street arises, the pedestrian network is interrupted. Here, the data origin model OpenDRIVE is not helpful as the model is focused on car navigation and modelling sidewalk connections across the street vary between data modellers. In the current state of the test datasets, such connections are not modelled. This is not a realistic representation of the pedestrian network. Therefore, a solution must be found to connect the pedestrian networks. This can be done by adding a relationship connecting the two TrafficSpace nodes of the pedestrian network that are closest to each other. However, an automated implementation faces multiple issues:

1. Especially at the typical locations where a pedestrian crosses a street, e.g., at a junction, there are many TrafficSpace nodes with irregular TrafficSpace geometries. This makes it difficult to find the correct nodes to connect.

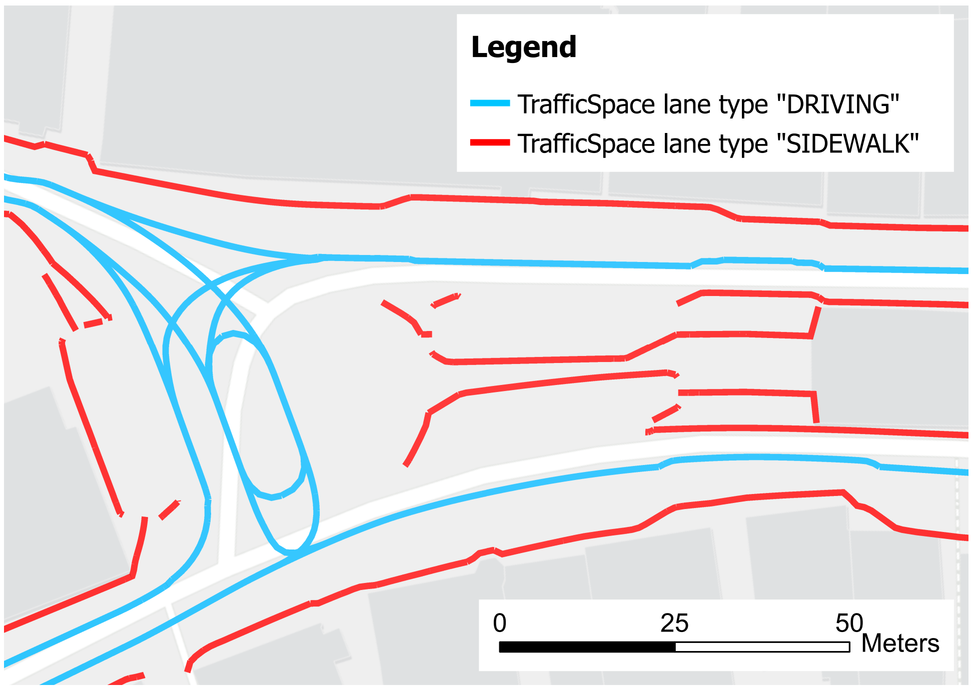

Legend
TrafficSpace lane type "DRIVING"
TrafficSpace lane type "SIDEWALK"
0
25
50
Meters

Figure 3.6.: Overview of the TrafficSpace connectivity; interrupted sidewalks in red and driving in blue.

2. There is no additional information available to find the correct nodes to connect. The knowledge about the road, lane, and lane segment IDs as well as the lane type could help narrow down the search. However, there is still a need for geometry information and assumptions to only connect nodes that might be connected in the real world. For example, by using a maximum distance between the nodes to connect and only connect nodes that are within some distance of the walking direction.

3. Finding suitable nodes to perform a connection to the other side of the road is difficult. Sometimes there is no indication in the graph representation that a road crossing connection should be introduced, as is the case for a continuous chained sidewalk. On the opposite sometimes many unconnected sidewalk elements need to be connected to form a continuous sidewalk where by just using dead-end logic the wrong nodes - opposite sides of the street - are connected.

4. Lastly, some issues occur in the real world. This is for example the case in the Grafing dataset. While the dataset also lacks the aforementioned issues, it correctly models unconnected sidewalk elements that are naturally present in the area of the test data region. In some parts of the area, small, sometimes interrupted sidewalks

exist. While pedestrians in the real world change the side of the road to follow a sidewalk from there or walk on the side of the road to the next section of a sidewalk these natural connections a pedestrian identifies on site are not present in the dataset. On the one side, this reflects the built reality, whilst on the other side pedestrian movements cannot be represented accurately as it is unclear where to allow changing the side of the road or which parts of the road pedestrians are allowed to walk on to reach the next sidewalk.

A general idea is presented in the following paragraph. (1) Connect the fragmented sidewalk elements of one side of the street. (2) Repeat step one for all occurrences. (3) Follow the road representation and log IDs that lack a sidewalk type if there previously was a sidewalk. As is the case in street crossings. (4) Check whether the sidewalk continues after a maximum distance or if the sidewalk ends naturally. (5) If the sidewalk continues within the set maximum distance from the last sidewalk element, introduce a connection between the last and the next element. The three main ideas to improve the pedestrian network are described in more detail in the following as visualized in figure 3.7.

1. Connecting two sidewalk blocks This is rather difficult to perform automatically, as some streets prohibit the crossing, e.g., on multi-lane roads. Additional, information that could help is the presence of crosswalks or traffic lights indicating a pedestrian crossing. For smaller roads, the crossing could generally be allowed. One issue remains, however. This is identifying the correct TrafficSpace objects that should be used to connect both sides of the road. Furthermore, it is to be investigated if only TrafficSpace objects on the sidewalk should be used or if bike, car, and other lanes and their surfaces should be included as well. The second idea would use elements of a different transportation type but also would use an uninterrupted surface representation.

2. Connecting continuing sidewalks on opposite sides of the road Connecting sidewalks that continue on the other side of the road could utilize the lane information of the OpenDRIVE model present as generic attributes. First, all sidewalk dead-ends are identified. For the last segment, the lane information is checked to find possible sidewalk elements on the other side of the road by comparing lane IDs and lane types. If another sidewalk element is found a connection between the two can be added.

3. Connecting fragmented sidewalk segments Here two types have to be mentioned: One, the directly adjacent sidewalk areas that lack a connection. Two, sidewalks along a stretch of road that have gaps in-between, e.g., due to driveways or other interruptions. To reach the next sidewalk it is necessary to walk on the side of the road. When those sidewalks are within a tolerable distance for walking on the side of the road they can be connected. To find such candidates some GIS tools like buffering could be used. Then, the closest elements could again be connected to improve the pedestrian subnetwork.

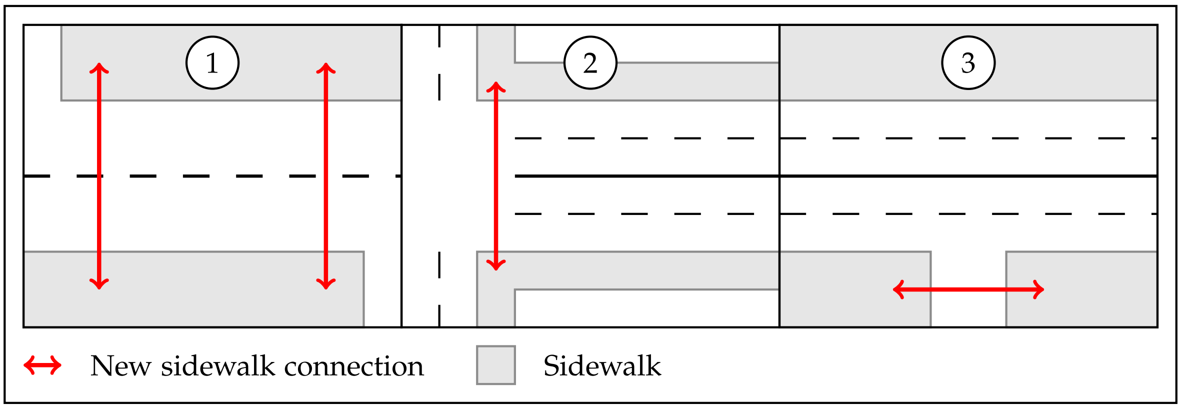


Figure 3.7.: Three issues of sidewalk connections with suggested connections in red.

Other alternatives remain in the form of manual modelling or semi-automatic solutions that identify possible transitions and need to be accepted by a modeller to be included in the dataset. As these problems stem from the original dataset and the conversion to CityGML, the issue is not pursued further. This is because it can be assumed that if the street space is modelled correctly in CityGML, these connections will be correct or at least better than they are in the case of the conversion process. Furthermore, it shall be tested if the application works with the available data and the data quality.

Adding Weights to the Relationships

In order to introduce weights for the shortest path algorithms, meaningful measures must be found. A list of weights has been introduced in section 3.1.2. The following weights can be derived from the CityGML data and directly used as costs for the shortest path search:

- Length of a segment (simplified or exact calculation using the geometry)
- Inclination using the geometry - in order to obtain a positive value add a value for the minimum inclination to the value shifting the minimum to 0, $-100\% \rightarrow 0\%$
- Width of a segment - Using the multisurface geometry multiple width values for a segment can be calculated. This results in some options to choose as a weight, e.g., the minimum, maximum, average, or median width.
- Clearance space height (not usable in the test datasets)

In the test dataset, additional information is provided through the generic attributes created during the conversion process from OpenDRIVE to CityGML. This information includes:

- Maximum speed
- Material roughness (of the surface)
- Material friction (of the surface)
- Lane type (e.g., driving, bicycle, sidewalk, etc.)
- Height offset, e.g., for lowered curb stones

However, the generic attributes can not be considered as constantly available. Furthermore, even the attributes themselves are not present within all types of transportation. For example, the maximum speed attribute is only available for *'DRIVING'* lanes.

Additional data is implicitly stored in the CityGML file and requires extended preprocessing to be used as a weight. This includes:

- Street signs $\rightarrow$ regulations on the street, e.g., speed limits, in Germany the StVO (Straßenverkehrsordnung) is used to regulate the traffic
- Traffic lights and other city furniture
- Vegetation objects, e.g., trees.
- 3D analysis results, e.g.:
  - Violations of traffic corridor volume (e.g., trees, traffic lights, etc.)
  - Shadow analysis, including buildings and vegetation
- Number of intersecting roads
- Building facades and entries facing the street. This could be used to link the entry of a building to the street and thus allow routing to use information about a building. Potentially, this could even be used to connect indoor and outdoor navigation.

The 3D analysis is very limited, as the test datasets (Ingolstadt) only contain some building elements and there is no crown information for the trees. Some assumptions can be made to calculate additional weights based on existing information. This includes:

- Maximum speed for each mode of transportation $\rightarrow$ traversal time

- Maximum inclination for each mode of transportation
- CO2 emissions, for each transportation mode
- Travel expenses, for each transportation type
- Indexes like the walkability index or the bicycle path quality using BLOS analysis

Lastly, additional data sources can be combined with the CityGML data to improve the available data. This includes real-time data like blocked roads, traffic jams, or current weather conditions. Due to the time constraints only a limited number of those weights can be implemented.

Thus, the data is now analysed to find attributes and information that can be used to calculate these weights. Furthermore, the weights must be added to the graph database. To understand how the weights are applied to the graph representation it is important to understand how the graph database represents the CityGML data. As an example, the Euclidean distance between the start and end points of each TrafficSpace geometry is calculated and added to the relationship *SUCCESSOR_OF*. This is done by a multi-step process:

1. Find all TrafficSpace nodes
2. *For each TrafficSpace:* Find the start and end point of the TrafficSpace geometry
3. Calculate the Euclidean distance between the start and end point
4. Add the calculated distance as a property to the relationship *SUCCESSOR_OF* of the corresponding TrafficSpace nodes

Figure 3.8 shows how the length-weight is derived from the CityGML geometry. The length calculated for a single TrafficSpace geometry will be used as the weight for the *SUCCESSOR_OF* relationships of the corresponding TrafficSpace nodes. The geometry representation exemplarily shows straight and curved geometries as well as the simplified Euclidean distance in orange. In the graph representation, each TrafficArea - rectangle in the above representation - and the corresponding TrafficSpace are represented as nodes in the graph database. In the figure, the TrafficSpace nodes are represented by the blue nodes and with the green *SUCCESSOR_OF* relationship to connect successors. The middle rectangle shows the more accurate distance calculation using all points of the geometry. The graph representation is the same, however.

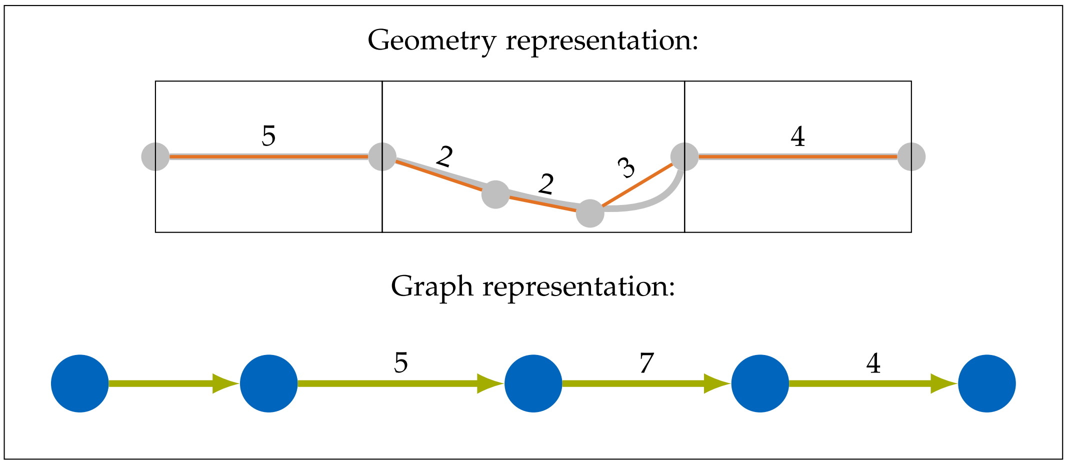


Figure 3.8.: Geometry to graph mapping: The geometry is represented in total by a single node. Node connections include weights based on the complete section. The graph representation is independent of geographic locations.

3.2. Data Analysis

3.2.1. Data Structure Analysis

Getting insights into the data is an important part and the first step of data analysis. The CityGML data can be visualized in a 3D environment, or the data can be queried from the graph database using Cypher queries. Due to the complex structure, the Cypher queries can be difficult to construct. In order to extract complex dependencies, the combined use of Cypher and a driver, like the Neo4j Python driver, is useful. To inspect the CityGML file, the FME Data Inspector (Safe Software Inc, 2023) and ArcGIS Pro (Environmental Systems Research Institute, Inc., 2023) were used. Lastly, a look at the CityGML data structure is helpful to understand the data and to find the information needed for the routing process. This also helped identify key elements in the graph database.

Relationships

During the process of familiarizing with the data, it was found that the relationships between the TrafficSpace and TrafficArea nodes are of particular interest. TrafficSpace nodes combined with their predecessor and successor connections make up the base of the routing network. Connected to these nodes are the TrafficArea nodes. The TrafficArea elements contain additional information about the street segments. These help derive weights that can be used during the routing. This leads to the general structure of the graph data. No existing connections are removed and only a limited number of new data connections are added in combination with the weights of those relationships.

3.2.2. Spatial Analysis

While this topic includes a wide range of analyses, in the thesis only some spatial analyses are performed to increase the use of shortest path searches. One of those is the identification of the nearest TrafficSpace element to the coordinates of a given point. This allows the user to give the start and destination by coordinates or an address that can be geolocated and transformed into coordinates. The coordinates are then used to find the nearest TrafficSpace element. This process returns the UUID of the nearest TrafficSpace object. The IDs are required for the shortest path function to find the corresponding start and destination nodes in the graph database. While the IDs are theoretically enough to test the shortest path routing, it is more convenient to use coordinates. To find the nearest TrafficSpace element first the start and end point of every geometry of the TrafficSpace objects is stored in the spatial structure of an k-d tree. Once this step is completed, a k-nearest-neighbour-search can be performed to determine the closest TrafficSpace element to the given coordinates. The k-d tree is implemented using the Python library Open3D (Zhou et al., 2018). To keep the number of temporarily stored points small, only the start and end points of the TrafficSpace geometries are stored in the k-d tree structure. The k-d tree is generated once in the beginning. Then an arbitrary amount of routing queries can be performed. Thus, the k-d tree is generated for each search session. The other spatial search includes the topology search to find neighbouring lanes and connect them. This "spatial" search uses information about the road, lane, and lane segment. The information is stored as generic attributes. Together with the type of lane, neighbouring lanes in the traffic network can be identified and additional lane-changing connections can be introduced. Here, however, the geometric information of the elements is not used in the current implementation concept.

3.2.3. Network Analysis - Shortest Path Search

For performing shortest-path analyses several options exist in the Neo4j environment. First, there is a default shortest path function that finds the closest path from a node to another given node. For the evaluation of the shortest path, only the number of nodes along the route is taken into account. While this works for simple graphs, it is not suitable for performing more complex searches. Since no weights or relationships can be defined to be used, this function is not suitable. However, the APOC extension includes two more shortest path analysis functions, one implements the Dijkstra algorithm and the other the A* algorithm. Both algorithms can be used to perform shortest path searches on the chosen data model for representing the transportation networks of the mapped CityGML dataset. The Dijkstra function provides the following parameters: start and end node, relationship types that shall be used as well as the direction they shall be traversed, and the relationship property that should be used as the weight for evaluation. It is further possible to define a default weight if the relationship property is not available at a relationship as well as the number of shortest paths that are returned. The number of shortest paths returned is 1 by default. Returned values include the path that is found as well as the total weight of the path.

```
 // Dijkstra function call
 MATCH (from:Loc{name:'A'}), (to:Loc{name:'D'})
 CALL apoc.algo.dijkstra(from, to, 'ROAD', 'dist') yield path
 as path, weight as weight
 RETURN path, weight
```

Code 3.2: Exemplary APOC Cypher query for the Dijkstra algorithm

The A* function provides similar parameters including start and end nodes, relationship types and the direction of traversal that shall be used as well as the relationship property for the weight. Additionally, the A* function uses a latitude and longitude node property as a heuristic to determine the distance between two nodes and to guide the search towards the end node. A default value for the weight property can be set as well. The returned values are the found shortest path and the total weight of the path.

```
 // A* function call
 MATCH (from:Loc{name:'A'}), (to:Loc{name:'D'})
 apoc.algo.aStar(from, to, 'ROAD>|SIDEWALK>', {weight:'dist',
default:10, x:'lon',y:'lat'}) yield path as path, weight as
weight
 RETURN path, weight
```

Code 3.3: Exemplary APOC Cypher query for the A* algorithm

This work uses the available Dijkstra and A* functions of APOC. However, both of these do not work with negative weights, e.g., for the use of inclination values. Thus, applications using negative weight values would benefit from the development and implementation of the Bellman-Ford algorithm.

3.3. Multimodal Routing

3.3.1. Preparation of the Multimodal Routing Network

Introducing Switch Nodes and Relationships

To generate a multimodal network, the graph database must be enriched with additional relationships and weights needed for combining the existing subnetworks. This includes the switch node concept (Liu, 2011) which was adapted and modified for the purpose of the thesis. In the graph database, a location for performing a transportation mode switch is found. The type *'PARKING'* of TrafficSpace (previously AuxiliaryTrafficSpace) is used for the purpose. It is checked whether a *'PARKING'* TrafficSpace element lies between two TrafficSpace elements of a different type, e.g., *'DRIVING'* and *'SIDEWALK'*. If this is the case, the TrafficSpaces are connected via two relationship types, *SWITCH_TO* and *SWITCH_TO_PARKING*. This way, both subnetworks are connected using the TrafficSpace of type *'PARKING'* in between as a switch node. The weights of *SWITCH_TO_PARKING* are by default set to 0 as to reflect that only one weight shall represent the transportation mode change. However, the *SWITCH_TO_PARKING* relationship also serves a purpose: Controlling the availability of a parking space. The *SWITCH_TO* relationship contains the weights for the whole switching process. This, for example, includes the whole distance between the TrafficSpace of type *'DRIVING'* and *'SIDEWALK'* instead of the partial weights between *'DRIVING'* and *'PARKING'* and *'PARKING'* and *'SIDEWALK'*.

Adding Weights for Transfer Relationships

While the general concept of splitting the transfer into two relationships with two weights was explained in the above section, the actual implementation is more complex. This is due to the fact that the routing algorithms like Dijkstra and A* are not designed to work with multiple weights. Therefore, the weights of the transit must be the same used as in the subnetworks. This means that either values for simple weights like length or time must be found or multiple values must be combined into a single weight. Additionally, it has to be noted that very high values are used to "block" the transit by making a transfer very costly, e.g., in the case of an unavailable parking space. However, if there is no other route that leads to the desired destination or the alternative route is still more costly, the routing algorithm will still use the transit. One alternative would be to remove and add the first transit relationship as it is available because this can lead to some unwanted results. This issue should be addressed with a more sophisticated solution if the routing process is used in a real-world application.

3.3.2. Advanced Routing

To analyse a wider range of potential routes, the inclusion of different types of granularity can potentially add value to the routing process. This includes the use of different geometric and semantic representations of the street space, which adds another set of challenges. For combining two different granularities, the logic behind using the traffic space must be consistent. For example, if a granularity lane combined with a granularity way is used, the successor and predecessor links should still connect elements of the same type, e.g., instead of representing each driving lane only the vehicle surface is available. However, when the granularity area is used to represent the street space containing different transportation modes, e.g., a sidewalk and a driving lane, inherent issues occur. While this is certainly an option for single transportation mode roads like the autobahn, the combination of way and lane granularities is more relevant as it still includes a separation of transport types. In terms of representing different granularities, the graph model does not change significantly. The backbone of the routing network is still available via successor and predecessor connections. Those are independent of the geometrical representation and the granularity of the TrafficSpace element. When it comes to calculating weights, especially when using geometries of different dimensions, new methods are required. In the case of lane-based routing, the length of a segment can be used as a weight. While this is directly available following the points of the geometry, a surface or volume representation requires the calculation of a center line to serve as a reference. Yet, 2D and 3D geometries provide additional information, e.g., the width and height which are not available for the lane-based representation of the TrafficSpace. To show the differences in the routing process, a combination of TrafficSpace line and volume with the change from granularity lane to granularity way can provide insights. Therefore, the Grafing dataset is extended by a parking garage building. The building consists of 3-dimensional TrafficSpace elements in the granularity way. This synthetic dataset is then used to test the developed concepts for compatibility. As a garage building typically serves to switch between the transportation modes of walking and driving, the test consists of a multimodal routing query. In order to use both transportation modes, the entrance of the garage is connected to the sidewalk and both driving lanes which represent the two driving directions. The goal of this test is to show a routing scenario in which a route is using the driving element which leads via the parking garage to a pedestrian destination. Thus, the routing shall also include routing within the parking building to guide the driver to a parking space. Then the driver shall be guided from the parking space within the building to the pedestrian destination outside.

4. Case Study and Results

This chapter presents the implementation of the developed methodology and the results of the case study. The case study is based on a CityGML dataset derived from OpenDRIVE data of the city of Grafing near Munich. The chapter covers three aspects: improving structure for navigation purposes, multimodal routing, and routing with dynamic data. It shows how CityGML data can be used to model different transportation modes and their connections in a graph database, how multimodal routing can be implemented using graph databases and shortest path algorithms with different weight functions, and how dynamic data can be incorporated into the routing process to reflect changes in the network. The Ingolstadt test dataset only covers a small area and does not contain locations to switch transportation modes. This limits the number of connections as well. Thus, only one path between start and destination node pairs exists making the use of weights irrelevant. For this purpose, the main CityGML dataset used is the Grafing dataset. It is pre-processed and afterwards, the implemented methodology is applied to the dataset. The results of this process are presented in the following.

4.1. Pre-processing and Implementation

4.1.1. Data Quality and Availability

Data Availability

As already mentioned, the used datasets do not include all information CityGML is capable of utilizing. This includes, for example, information on the clearance space height of the street elements and tree crown diameters and makes volumetric analysis of the street difficult. Furthermore, the data was collected for a specific project and no widespread dataset for the whole of Bavaria, Germany, Europe or the world is available. Additionally, each data provider uses a different data collection process and only includes information that is relevant to the project use case. In the case of the data used in this thesis, the focus lies on the modelling of the street space and the nearby surroundings. Figures 4.1 to 4.3 show the extent of the available test regions.

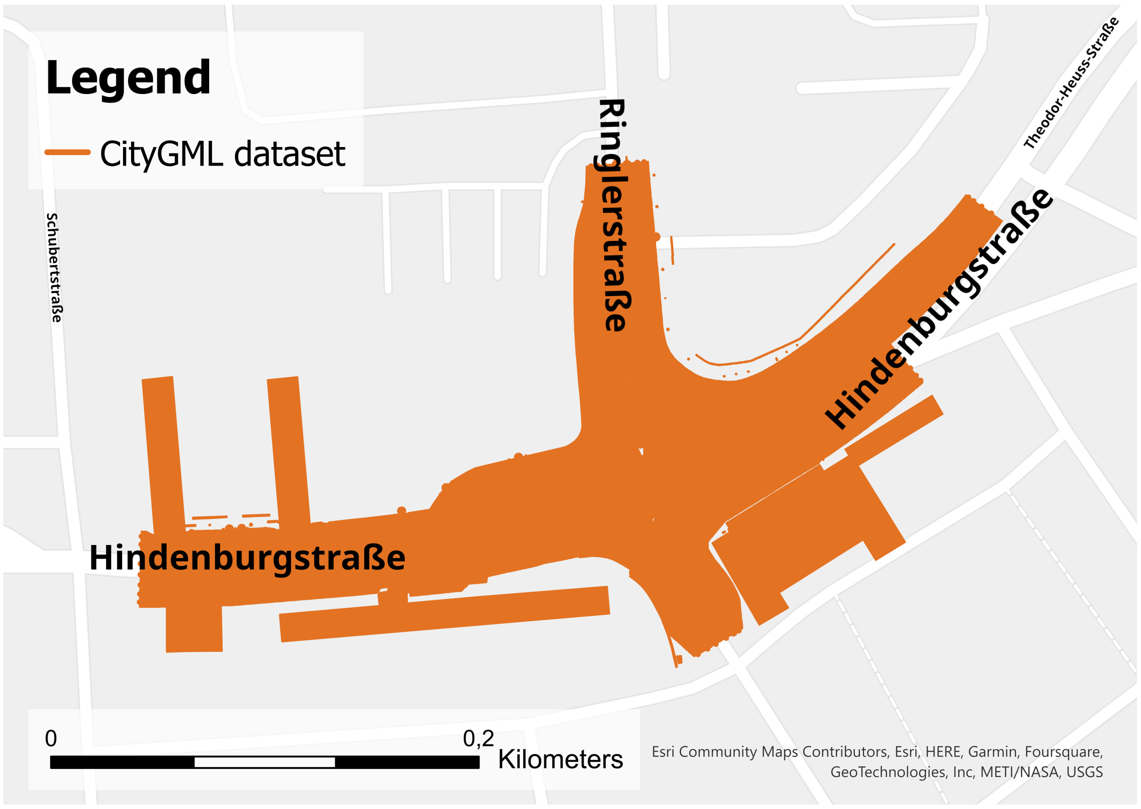


Figure 4.1.: Overview showing the extend of the CityGML Ingolstadt dataset

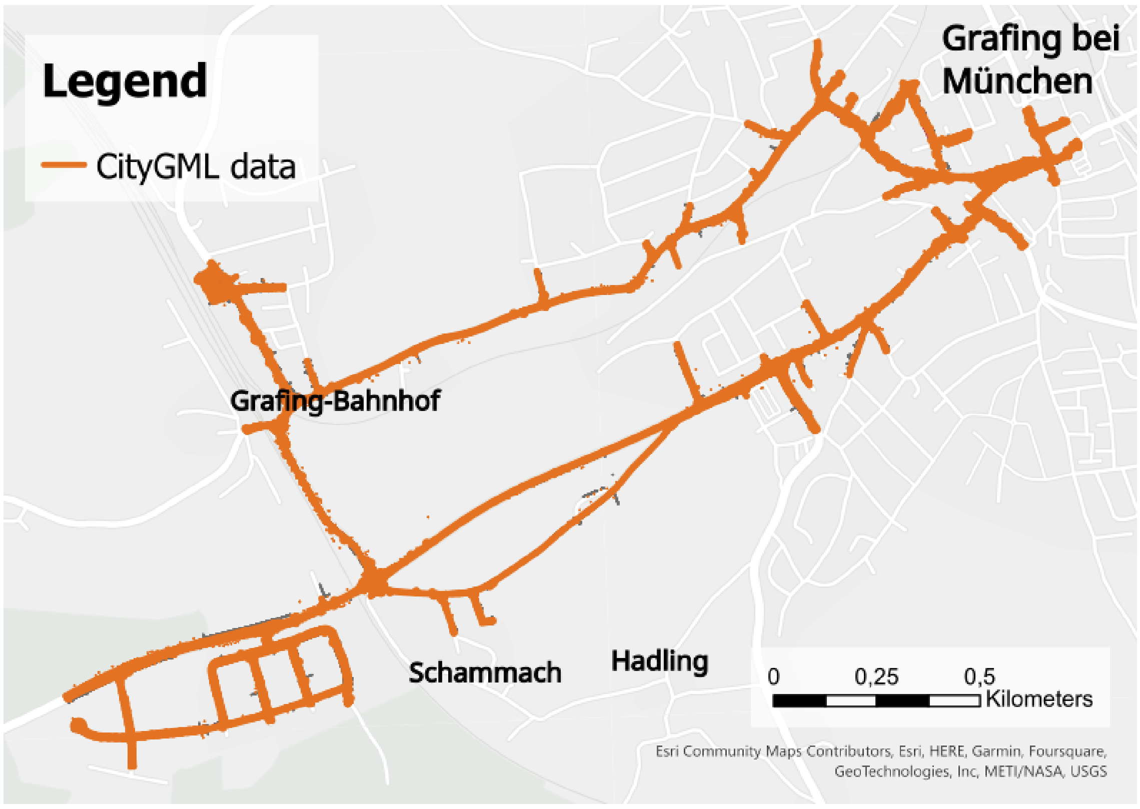

Legend
CityGML data
Grafing bei München
Grafing-Bahnhof
Schammach
Hadling
0
0,25
0,5
Kilometers
Esri Community Maps Contributors, Esri, HERE, Garmin, Foursquare, GeoTechnologies, Inc, METI/NASA, USGS

Figure 4.2.: Overview showing the extend of the CityGML full Grafing dataset

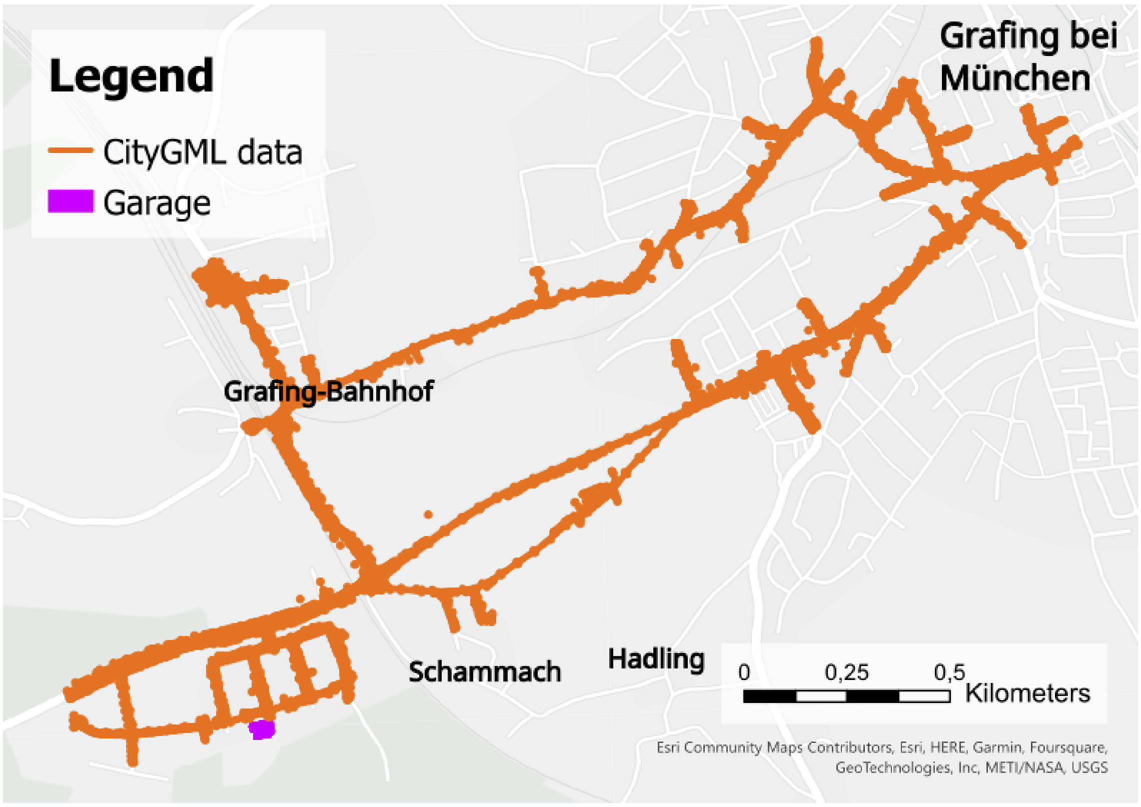

Legend
CityGML data
Garage
Grafing bei München
Grafing-Bahnhof
Schammach
Hadling
0 0,25 0,5 Kilometers
Esri Community Maps Contributors, Esri, HERE, Garmin, Foursquare, GeoTechnologies, Inc, METI/NASA, USGS

Figure 4.3.: Overview showing the extend of the CityGML Grafing dataset with a garage building

The issue is further amplified by the data origin OpenDRIVE because that data is primarily focused on a consistent network for cars. Furthermore, even though the Ingolstadt and Grafing near Munich datasets are provided by the same data provider not all of the street elements supported by CityGML are present in both datasets. While the Ingolstadt dataset includes information on bike lanes, for example, those are not available in the Grafing dataset. On the other side, the Grafing dataset includes specified parking areas, which are not available to the limited extent of the Ingolstadt excerpt. Nevertheless, it is possible to use the datasets in combination to test the routing process with different elements supported in the CityGML model. While this poses another challenge, it is also an opportunity to test the routing process under different conditions. When comparing the cities of Ingolstadt and Grafing, it is obvious that there are differences in the built infrastructure which is also reflected in the dataset and therefore also in the usable data. At the moment the data collection is a further necessity for the use of CityGML for more advanced routing and navigation purposes. Especially the collection of all the information CityGML supports in order to use the full potential of the data model. Furthermore, the data collection process is not standardized and therefore the data quality and available information vary from city to city. This makes it difficult to use the data for routing purposes in a wider context. This issue is further complicated if the CityGML data is created through a transformation process as is the case for the data used in this thesis. Some of the attributes used to create the advanced connections in the graph database are generic attributes and can therefore vary from dataset to dataset. This makes it difficult to create a generalized process for the creation of a routing network in the graph database. However, the process described in this thesis can provide a starting point for the creation of a generalized process and can be finalized when a standardized data collection process and or attribute naming is established.

Data Quality

The underlying OpenDRIVE data is of high quality and provides a standardized way of modelling street networks. However, there are some issues with the data that need to be addressed if it is used in CityGML for routing. As of right now, there are many artefacts in the CityGML data, the generic objects and references to the reference line, for example. On the one hand, the OpenDRIVE data does not provide all the information that CityGML is capable of storing and utilizing. However on the other hand some artifacts result from missing equivalent objects or attributes for each OpenDRIVE element in CityGML meaning that not all modelling principles of both standards are compatible. This leads to information with no clear representation in the CityGML model. Such additional information can at least be represented by generic attributes in

CityGML and thus require additional pre-processing. Lastly, there are some issues with the data itself, e.g., lane connections that are provided by attributes and currently not using CityGML XLinks or the linking direction being still dependent on the missing reference line. Lastly, OpenDRIVE is focused on vehicle navigation such as cars and trucks while CityGML is a data model for 3D city models. These issues are addressed in the next sections which also describe the steps taken to prepare the data for routing. Some of these can also be ignored if adequate transformation is used in the r:trån tool. This led to some suggestions that were made to improve the conversion tool. The developer of r:trån already started implementing those suggestions and further improvements will be implemented in the future.

4.1.2. Pre-processing of the CityGML Graph Dataset

Preparation of the Multimodal Routing Network

To start the implementation of the methodology, the graph database containing mapped CityGML data required some pre-processing. This is necessary for using the shortest path algorithms (Dijkstra and A*) provided by the APOC extension. Thus, several changes and optimizations were made to the graph database. For this purpose, Cypher queries as well as additional analysis in Python were performed. As a result of the pre-processing steps, a class was implemented containing functions to pre-process the graph database. These interactions were implemented in the Python tool interactor4neo4j. This Python tool consists of two classes and several helper functions. The first class Neo4jPreProcessor contains functions to pre-process the graph database and will be explained in more detail in this section. The second class Neo4jNavigator contains functions to navigate through the graph database and will be explained in more detail in the following section 4.2. First, the pre-processing steps will be explained using the example pre-processing script which was used for the case study and introduces all necessary modifications to the graph database. Afterward, the results of the pre-processing steps will be presented.

The following code example shows the script used for calling the needed functions for pre-processing. Here, two types are called. The *generate()* and *insert()* functions. In short, the *generate()* functions call *insert()* functions when there are multiple changes to the database to reduce the number of function calls by the user of the class. However, if needed the *insert()* functions can be accessed individually. Only inserts were made and no data was deleted. The introduced changes include the addition or modification of information or relationships resulting in shortcuts.

```python
# Test script for the Neo4jPreProcessor class
# Imports
import datetime

from constants import password, username
from interactor4neo4j import Neo4jPreProcessor

# Constants
uri = "bolt://localhost:7687"

# Test function
def test_insert():
    now = datetime.datetime.now()
    preprocessor = Neo4jPreProcessor(uri, username, password)
    # Adding relationships
    # ! Call these functions only once to prepare the database!
    preprocessor.create("predecessor_shortcut")
    preprocessor.create("successor_shortcut")
    preprocessor.create("lane_changes")
    preprocessor.create("transport_mode_switch")
    # Adding properties
    preprocessor.create("transportation_type")
    preprocessor.insert("speed_limits", [])
    preprocessor.create("weight_attributes")
    preprocessor.insert("coords", [])

    preprocessor.insert("
   add_bidirectional_successor_sidewalk_relationship", [])

    preprocessor.close()
    print(f"Pre-Processing took: {datetime.datetime.now() - now}
   ")

if __name__ == "__main__":
    test_insert()
```

Code 4.1: Pre-processing script for the case study

In the beginning, a new connection to the database is established. This is done by creating an instance of the Neo4jPreProcessor class. Then the subsequent function calls can access the graph database.

The presented order of function calls is important because first relationships need to be created to assign them properties. However, for example, the addition of coordinates to the TrafficSpace nodes (*insert("coords", [])*) could be performed as the first function call. Adding attributes to a node does not depend on newly generated relationships. In the following the individual function calls are explained in more detail.

At first, predecessor and successor shortcuts are introduced to the graph database. The successor connections (*SUCCESSOR_OF* relationships) are the backbone of the routing graph. *PREDECESSOR_OF* shortcuts will not be used actively during routing but are helpful to understand the connections in a graph view during debugging and can assist in further analysis. As the current CityGML design connects only elements following a line the need to improve the structure of the graph by adding lane changes arises. Thus the next step is to add lane changes to the graph database. These are represented by *NEIGHBOURS_LANE* relationships. The general process of introducing lane change connections was explained in the methodology chapter 3.1.5. For implementing these relationships, a multi-step approach was chosen. First, all sections are extracted from the graph database. Then each section is analysed individually. Thus, there is no need to search through the whole graph database. Each section is used to find neighbours that can only be in the same section. This reduces the number of elements that must be searched through and stored in memory. For each section, all TrafficArea elements are extracted. These are then used to derive lane information - stored as generic attributes - and to find the neighbours of each road segment. The analysis sorts the segments by road ID, segment ID and lane ID. Then the neighbouring elements are compared if they lie on the same site and have the same sign and the same lane type, e.g., *'SIDEWALK'* or *'DRIVING'*. If these conditions are met, a *NEIGHBOURS_LANE* relationship is introduced between the two elements. This process is repeated for all sections. As this procedure has to perform multiple Cypher queries, including analysis, sorting operations and comparisons, it is time-consuming and requires an implementation within Python. After the lane changes are introduced to the routing network, it is possible to add relationships to connect different types of transportation. Due to time limitations and dataset restrictions, only transportation mode switches at *'PARKING'* areas are implemented. The implementation is similar to the lane change introduction. However, only objects of the type *'PARKING'* are used to find neighbouring lanes of different types. Other than that the analysis is performed similarly to the lane changes. The result are two types of relationships (*SWITCH_TO* and *SWITCH_TO_PARKING*) between the two TrafficSpaces and the *'PARKING'* TrafficSpace. When all main connections are added to the graph database, the shortest path algorithms can already use the routing graph as they only require nodes that are connected via a single relationship. Chains of connected nodes are not supported. As the newly introduced relationships have no properties, as is the case for all relationships present in the graph representation of the CityGML dataset, the usability of the shortest path functions is limited to giving each relationship a weight of one. Thus, results find shortest paths based on the number of nodes that must be traversed. To better reflect reality, additional information must be added to the relationships and routing weights have to be calculated. This is done via the next set of function calls in the pre-processing script.

First, the transportation type is added to the relationships. Then, speed limits are added. As the CityGML dataset provides different information about the street space, the CityFurniture, or more precisely, the street signs with speed limit information shall be used exemplarily to derive additional information. As this information is not available for every road segment and transportation type, the implementation uses a fallback for adding default speed limits for different modes of transportation. If no speed value is available via street signs, a default value depending on the transportation type was added. The main idea is to use the CityFurniture objects related to a street segment as a base for this analysis. First, CityFurniture which represents speed limits is determined via a combination of type and a sign code, e.g., "274-50" where "274" indicates a speed limit and the second number (50) the maximum allowed speed in kilometers per hour, as used in the German StVO, the road traffic regulations of Germany. The speed limit value in kilometers per hour is added as information to the corresponding *SUCCESSOR_OF* relationships. Then the relationships with speed limit information are called. Those serve as a starting point. For each *SUCCESSOR_OF* relationship, the chain of *SUCCESSOR_OF* relations is followed. Each successor *SUCCESSOR_OF* relationship gets the speed limit property of the starting relationship. This chain is stopped when there is no successor connection or when the next *SUCCESSOR_OF* relationship already has a speed limit property. Lastly, the remaining relationships without a speed limit property get a default value assigned based on their transportation type. After this, additional weight information and the final routing weights are added to the relationship. The information includes:

- Euclidean distance between the start and end point of the traffic space geometry
- Accurate distance between all points that make up the geometry of the traffic space
- Inclination

- Minimum width of the traffic area related to a traffic space

As the number of possible weights is large, a selection of weights was made. Based on the information stored as relationship properties the following final routing weights are calculated:

- Euclidean distance
- Accurate distance
- Inclination, translated to a positive value range
- Width, using the inverse minimum width
- Speed, using the inverse speed
- Time, using the accurate distance and speed limit

In order to use the shortest path algorithms, all relationship types of the routing network must have these routing weights. For deriving the weights a differentiation between the relationship types is made. Lane changes use the speed limit of the *SUCCESSOR_OF* relationship. Here three scenarios can occur. Both TrafficSpace nodes have a successor (1), only one has a successor (2) or both have no successor (3). In case (1), the minimum of both speed limits is chosen, for (2) there is one value that will be used and if both lanes have no successor that means there is a dead end. Thus, it is irrelevant to change to the neighbour lane and a value of 0.1 km/h is set by default. This allows traversing but prevents division by zero errors as the speed limit is used to calculate traversal times. The Euclidean distance is used as a base length value for all elements, this includes the distance calculation of lane changes or transportation mode changes, which just consider the first point of the geometry. The advanced distance is calculated for the traffic spaces in the traversal direction and is added to the *SUCCESSOR_OF* relationship. All other types have the same value for advanced and Euclidean distance. For calculating the inclination again just the first and last points of the geometry are used as is the case for the Euclidean distance. The same principle applies to the other relationship types. The information about the width of a segment requires more steps to calculate. First, the geometry of the corresponding traffic area is used instead of the traffic space geometry. Second, the traffic space geometry consists of MultiArea elements, each represented by a polygon made up of four points. Those four points are ordered in a way that point pairs for width calculation can be generated. For each element, the width calculation is performed and only the smallest width is kept to assign a minimum width value to the segment.

For the calculation of routing weights, the available information is used. The distance information remains unchanged. For using the Dijkstra algorithm, the weight values cannot be negative. Thus the inclination values, theoretically ranging from -100 to 100 are shifted to 0 to 200 by adding 100. If negative inclinations shall be used for the calculation of other weights, values between 0 and 100 must be handled differently, e.g., for lowering weight values. Width information represents the smallest segment width. However, for transportation, a larger width is better as it is easier to traverse. Therefore, the value is flipped with $\frac{1}{min\ width}$ resulting in large weights for narrow segments and small values for wide segments. As the algorithms use the minimal cost this modification is necessary. The same applies to the speed limits which are represented in a speed weight with the same flipping logic to prioritize higher speed limit segments over lower speed limits. Lastly, a traversal time is calculated showing the combined usage of information in the dataset, distance information derived from geometry and semantic information of street signs. For transportation mode changes a default waiting time could be added as well but it was set to 0 for this implementation. For the usage of the A* algorithm the TrafficSpace nodes require some information for the distance heuristic. Here, the coordinates of the first point in the geometry are stored as individual node properties. At last, the graph is enriched with a final relationship connection allowing *'SIDEWALK'* segments to be traversed in both directions with the correct values for inclination. This requires a new relationship type that is only added between TrafficSpace nodes of type *'SIDEWALK'*. Then the property values for the relationship are copied from the SUCCESSOR_OF relationship except all inclination information which is reversed to reflect the reversed travel direction. With this last step the pre-processing is finished and the graph database is ready to be used for routing.

This leads to the actual implementation of the pre-processing tasks via functions and Cypher queries to the database. For the interaction with the graph database, two layers of functions are needed. The first one handles the direct interaction with the database and contains the Cypher queries which are performed. Additionally, if needed they return the results of the Cypher query. The second layer controls these functions and calls them when needed. They also change the variable values as needed. Furthermore, the second layer analyses query results. Lastly, there is a third layer of functions (*create()*) which can call multiple of the second layer functions to reduce the number of function calls by a user of the Neo4jPreProcessor class. Functions of the first layer start with an underscore "_" and a first word indicating the purpose like "find", "insert", "set", or "get". The second level function is *insert()*. The insert function takes a string variable to use the right selection of first-level functions to also perform advanced tasks which require database interaction. This includes result analysis and result-based modifications to the graph structure. For an automatic one-click pre-processing experience even the Cypher queries that could be run directly in the Neo4j browser, without additional modifications or analysis in Python, were implemented. This leads to the following interaction chain:

(1) **Neo4jPreProcessor** → (2) ***create()*-function** → (3) ***insert()*-function** → (4) ***_[insert/set]()*-function(s)** → (5) **Cypher query** to the graph database.

Alternatively, a second call is required to first analyse the graph database and then update the content based on the analysis results. Such an interaction chain looks as follows:

(1) **Neo4jPreProcessor** → (2) ***create()*-function** → (3) ***insert()*-function** → (4) ***_[find-/get]()*-function** → (5) **Cypher query** to the graph database returning data → (6) analysis of the results inside the ***insert()*-function** → (7) ***_[insert/set]()*-function** → (8) **Cypher query** to update the graph database based on the analysis results.

The following code example shows the *insert()* function that introduces successor shortcuts to represent the successor connection via a single relationship.

```
@staticmethod
def _insert_successor_shortcut(tx, vars):
    tx.run("MATCH (a:`org.citygml4j.core.model.transportation.
  TrafficSpace`)-[:successors]-()-[:elementData]-()-[:
  ARRAY_MEMBER]-()-[:object]-(b:`org.citygml4j.core.model.
  transportation.TrafficSpace`) CREATE (a)-[:SUCCESSOR_OF]->(b)
  ;")
```

Code 4.2: Insert function for successor shortcuts

This function is called by the *insert()* function and modifies the graph structure with a single Cypher query - with no return value - that is not dependent on any analysis of the results of the query. The next example sets the distance weight as a property of the newly introduced *SUCCESSOR_OF* relationship. This function requires analysis and the following code example shows the *insert()* function that introduces the distance weight depending on the currently analysed segment to update the property of the *SUCCESSOR_OF* relationship. Here, two variables are used within the Cypher query. An ID value and a variable containing the Euclidean distance. The ID value is used to identify the currently analysed segment and to find the right node with the outgoing *SUCCESSOR_OF* relationship.

```
@staticmethod
def _insert_distance(tx, vars):
    tx.run('MATCH (a:`org.citygml4j.core.model.transportation.
  TrafficSpace` WHERE a.id=$id)-[r:SUCCESSOR_OF]->(b:`org.
  citygml4j.core.model.transportation.TrafficSpace`) WITH a, b
  MERGE (a)-[r:SUCCESSOR_OF]-(b) SET r.euclidean_segment_length
  =$weight RETURN a, b;', id=vars[0], weight=vars[1])
```

Code 4.3: Insert function for distance weight

An overview over the *insert()* function:

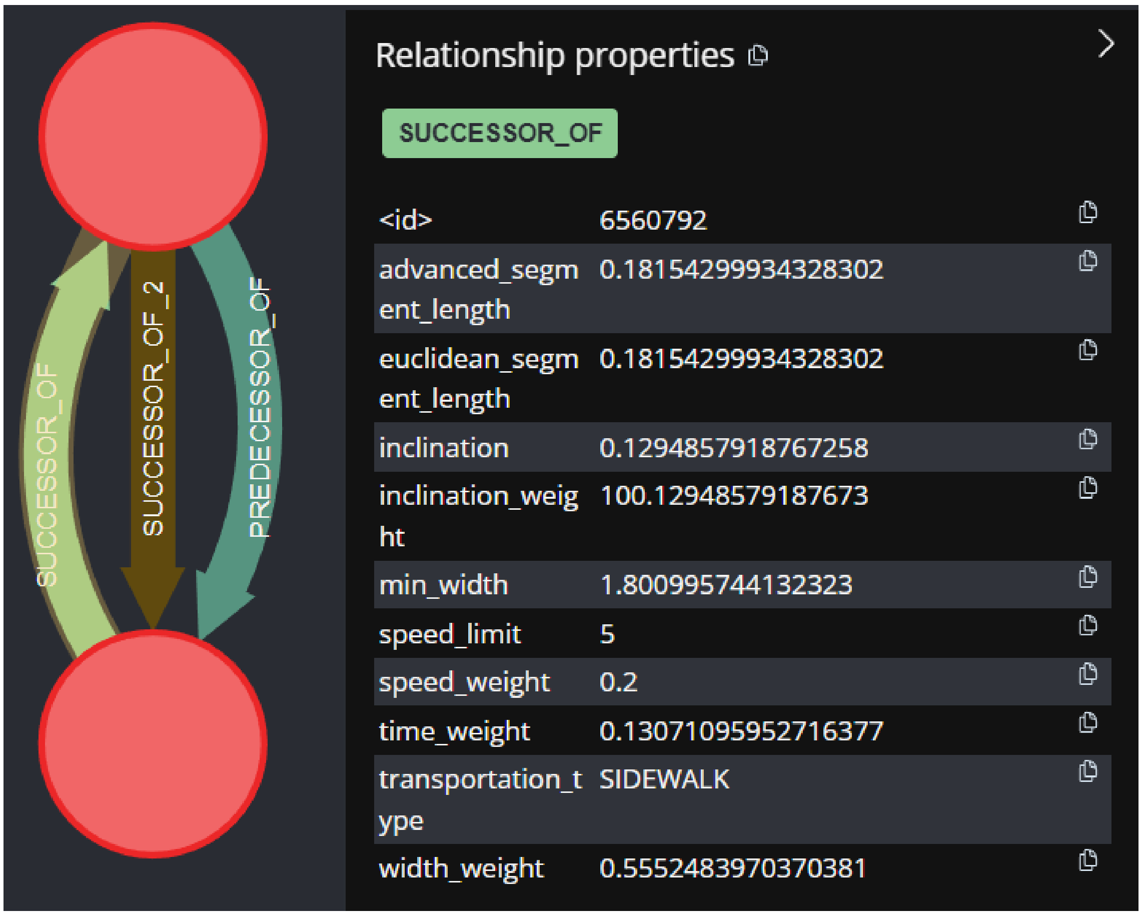
```
    def insert(self, query, vars):
        with self.driver.session() as session:
            if query == "lane_changes":
                session.execute_write(self._insert_lane_changes,
    vars)
            if query == "predecessor_shortcut":
                ...
            if query == "successor_shortcut":
                ...
            if query == "distance_weight":
                ...
            if query == "advanced_distance_weight":
                ...
            if query == "coords":
                ...
            if query == "inclination":
                ...
            if query == "min_width":
                ...
            if query == "lane_change_attributes":
                ...
            if query == "transport_mode_switch":
                ...
            if query == "add_transportation_type":
                ...
            if query == "
    add_bidirectional_successor_relationship":
                ...
            if query == "
    add_bidirectional_successor_of_2_weights":
                ...
            if query == "
    add_bidirectional_successor_sidewalk_relationship":
                ...
            if query == "traffic_signs":
                ...
            if query == "speed_limits":
                ...
            if query == "successor_of_properties":
                ...

        session.close()
```

Code 4.4: Insert function overview

Results of the Pre-processing

During the pre-processing steps, some changes were made to the graph database dataset. These changes include adding new relationships, as well as properties to relationships and nodes. Thus, the full Grafing database includes 22 135 relationships more. Furthermore, the database size increased from 1.80 GB to 1.86 GB. The following figures 4.4 to 4.6 show the network representation after the pre-processing in the Neo4j Browser. Testing the runtime of the pre-processing script for the Grafing dataset showed that the script takes about 880 seconds on average. The Grafing garage building dataset without the weight pre-processing takes an average of 263 seconds. To check the consistency of the pre-processing runtime, the script was run ten times. The results showed that the runtime is consistent as can be seen in the appendix C.

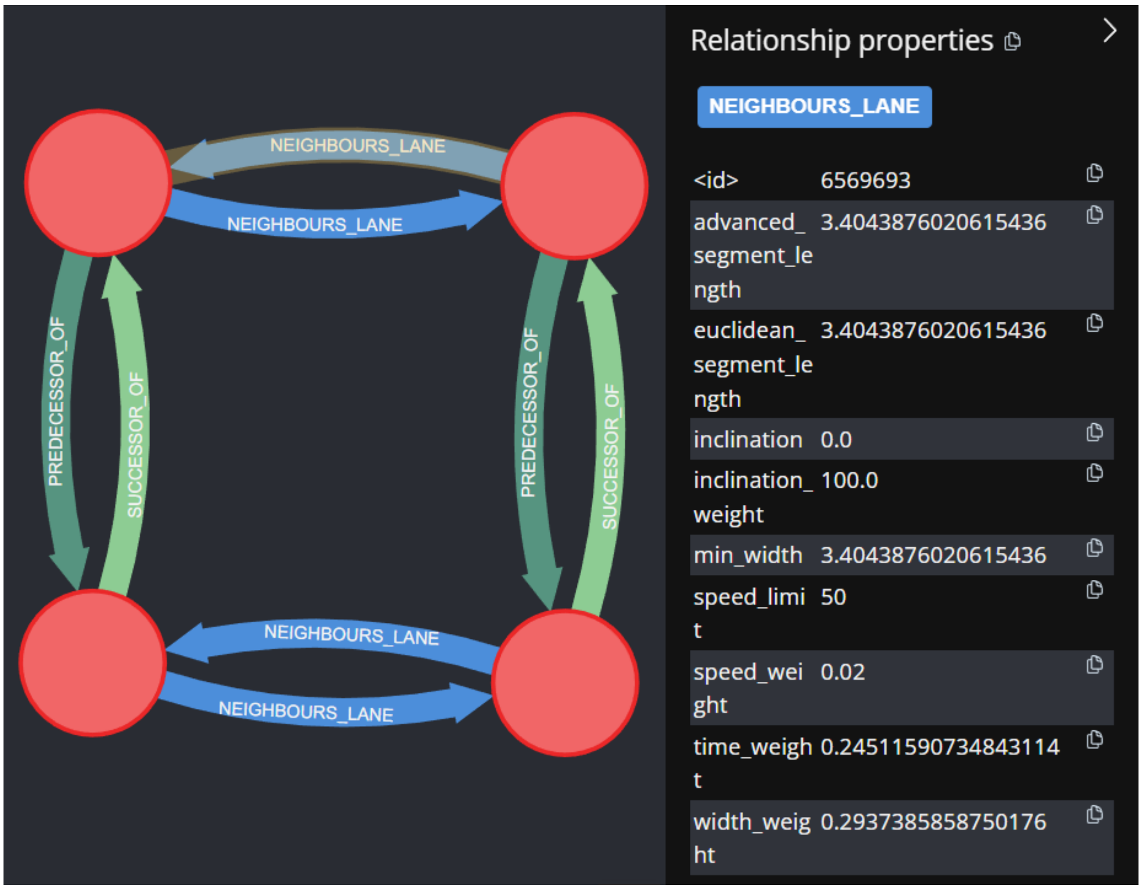


Relationship properties

SUCCESSOR_OF

<id>	6560792
advanced_segment_length	0.18154299934328302
euclidean_segment_length	0.18154299934328302
inclination	0.1294857918767258
inclination_weight	100.12948579187673
min_width	1.800995744132323
speed_limit	5
speed_weight	0.2
time_weight	0.13071095952716377
transportation_type	SIDEWALK
width_weight	0.5552483970370381

Figure 4.4.: SUCCESSOR_OF and SUCCESSOR_OF_2 (inverse SUCCESSOR_OF connection allowing pedestrian movement in both directions for sidewalk elements) relationships in the Neo4j Browser display

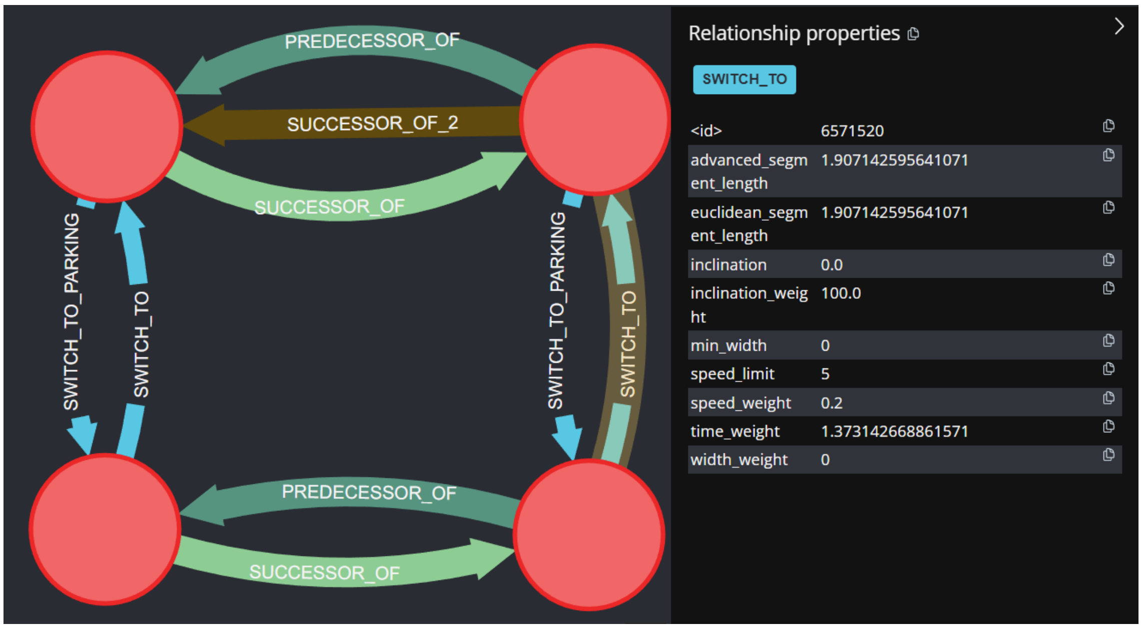


Relationship properties

NEIGHBOURS_LANE

<id>	6569693
advanced_segment_length	3.4043876020615436
euclidean_segment_length	3.4043876020615436
inclination	0.0
inclination_weight	100.0
min_width	3.4043876020615436
speed_limit	50
speed_weight	0.02
time_weight	0.24511590734843114
width_weight	0.2937385858750176

Figure 4.5.: *NEIGHBOURS_LANE* relationships in the Neo4j Browser display



Relationship properties

SWITCH_TO

<id>	6571520
advanced_segment_length	1.907142595641071
euclidean_segment_length	1.907142595641071
inclination	0.0
inclination_weight	100.0
min_width	0
speed_limit	5
speed_weight	0.2
time_weight	1.373142668861571
width_weight	0

Figure 4.6.: *SWITCH_TO_PARKING* and *SWITCH_TO* relationships in the Neo4j Browser display

4.1.3. Neo4jNavigator Class

The second part of the implementation includes the logic for using the graph database for routing. The needed function calls to the database are implemented in the Neo4j-Navigator class. This class extends the core functionality of querying the database using the APOC shortest path functions and receiving a sorted list of nodes along the shortest path as well as the total weight. The class contains additional helper functions to identify the start and destination nodes via geographical coordinates as well as some visualizing features. First, the core functionality is explained. Then the additional functions are presented. For the core functions, three versions were developed. Two multimodal functions, one using the Dijkstra, the other using the A* algorithm as implemented in the Neo4j APOC extension. Additionally, there is one single transportation mode Dijkstra function stemming from early testing. The single-mode Dijkstra function does not include the transportation mode switch relationships. Thus, only nodes of the start node transportation type can be traversed.

The functioning of querying the database is the same as in the pre-processing class Neo4jPreProcessor. Two functions are needed to send a query to the database. The first layer contains the Cypher query with integrated variables to use any UUID present in the database for identifying the start and destination nodes. A third variable is used to specify the weight property that shall be used during the shortest path search. The second layer function handles the session and result returns. To make the interaction with the database easier, a third layer of functions is implemented. These perform helper functionalities like visualizing the results or finding the start and destination nodes based on geographical coordinates. However, if one is purely interested in the nodes and the total costs along the shortest path, these core functions are enough. The following code example shows the first layer function using a Cypher query to find and return the shortest path between two nodes using the multimodal APOC Dijkstra function.

```
 def _find_shortest_path_apoc_djikstra_multimodal(tx, vars):
     result = tx.run("MATCH (from:`org.citygml4j.core.model.
transportation.TrafficSpace`{id:$id1}), (to:`org.citygml4j.
core.model.transportation.TrafficSpace`{id:$id2}) CALL apoc.
algo.dijkstra(from, to, 'SUCCESSOR_OF>|SUCCESSOR_OF_2>|
NEIGHBOURS_LANE>|SWITCH_TO_PARKING>|SWITCH_TO>', $weight)
yield path as path, weight as weight RETURN path, weight",
id1=vars[0], id2=vars[1], weight=vars[2])
     for record in result:
         if len([record['path'], record['weight']]) == 2:
             return [record['path'], record['weight']]
```

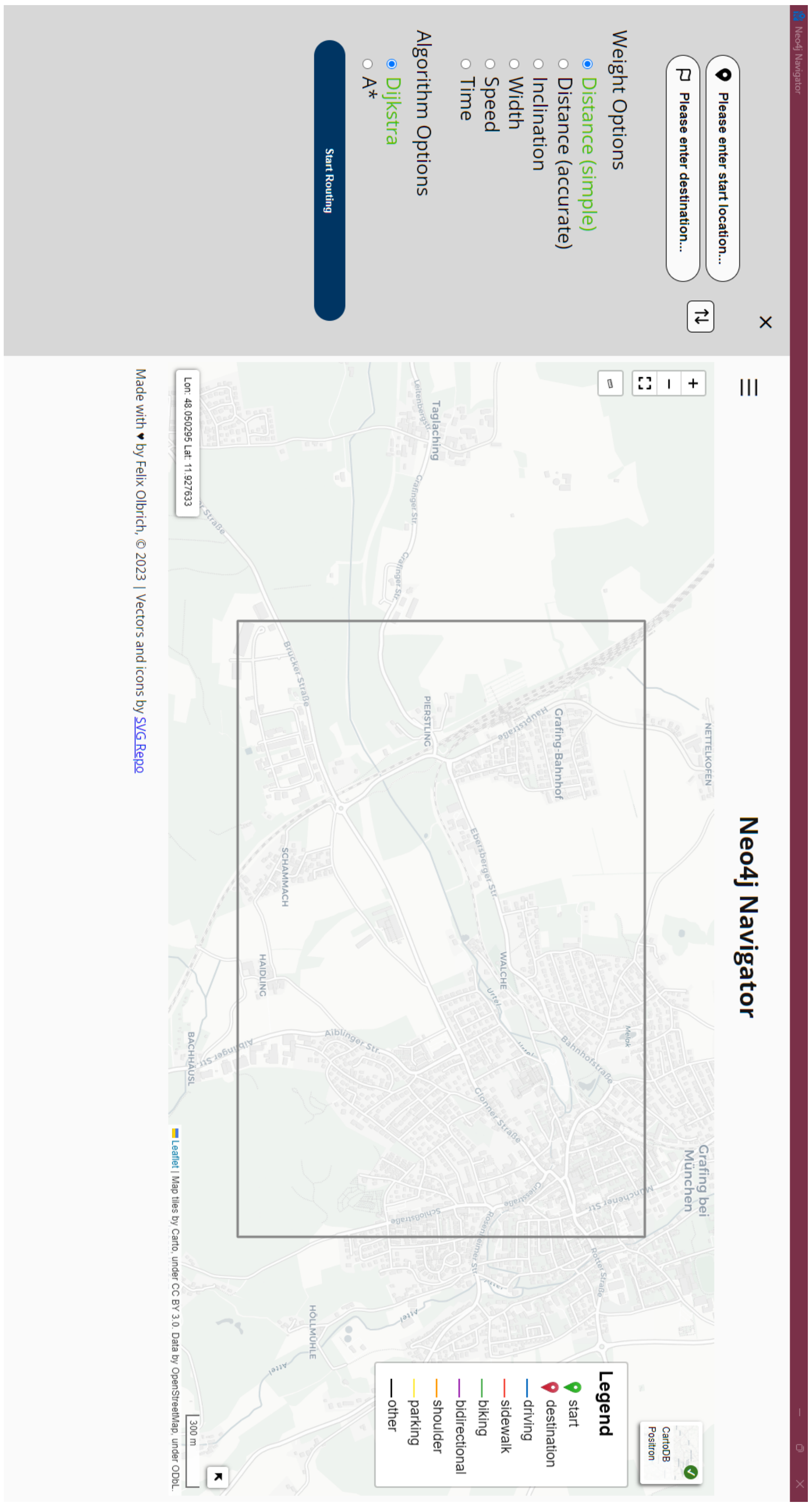
Code 4.5: Core function: multimodal shortest path query using the APOC Dijkstra function

The Dijkstra APOC function takes a start and destination node. These are specified using the provided UUIDs. Additionally, the relationships that shall be used during the shortest path search are defined. Thus, the function can only use *SUCCESSOR_OF*, *SUCCESSOR_OF_2*, *NEIGHBOURS_LANE*, and *SWITCH_TO_PARKING* and *SWITCH_TO* relationship types. The ">" symbol is needed to specify the direction of the relationship as connections between nodes are bidirectional in Neo4j. In order to use the right weight values and forbid the opposite relationship traversal direction, this specification is needed. Lastly, the weight property that shall be used for cost calculation is handed over to the Cypher query. The query is then executed and the results are returned. As there are two return values, the path and the total weight, the result records are modified to return the path and the weight as a list. The A* APOC function is implemented similarly, with the only difference that after the weight two variable names 'x' and 'y' are added. These define the x and y values of the nodes and are used for the heuristic to guide the search.

The second level function handles the session and passes the variables for the Cypher query to the first level function. Additionally, the result is analysed to check if a path was found. If not, the function returns a list with *'None'* values. If a shortest path is found, the function returns the nodes of the path and the total weight. As the chain of linked node IDs is unintuitive to analyse as a human, the addition of a graphical representation of the path is needed. For this, the Neo4jNavigator includes a visualization function that utilizes the Leaflet interactive web map library (Leaflet maintainers, 2023). More of this is explained in the Graphical User Interface section 4.1.4. With this, the core functionality still needs the UUIDs of the start and destination nodes. Those can, for example, be found by analysing the CityGML file or using TrafficSpace nodes from the Neo4j browser view. However, this is not very user-friendly. Thus, the Neo4jNavigator class includes functions to find the start and destination nodes based on geographical coordinates. Finding the nearest TrafficSpace node to a given latitude and longitude value is done via several steps. First, a k-d tree is generated from the TrafficSpace geometry. To reduce the size, only the first and last point in the geometry is stored within the k-d tree. The implementation uses the open3d Python library. The second step is to convert the latitude and longitude values into the same coordinate system that the CityGML geometry. This is done via the utm library. Then the nearest point in the k-d tree can be searched. The result contains the UUID of the corresponding TrafficSpace node. This search is performed once for the start and a second time for the destination. While the application is running, the tree only needs to be created once. However, it is newly generated every time the application is started. By using latitude and longitude values and converting them to identify the UUIDs of the nearest TrafficSpace nodes, it was possible to add geocoding functionality to the application or let a user select a point on an interactive map.

4.1.4. Graphical User Interface

For convenience and to help understand the results, a Graphical User Interface (GUI) was implemented. The GUI uses the Python library eel, which is a framework connecting Python functions with a web frontend that is locally hosted (Chris Knott & Samuel Williams, 2017, December 27/2023). First, the GUI only consisted of an HTML file containing the leaflet code to visualize the shortest path. However, this was later extended to also include the selection of start and destination location as well as weight and algorithm that shall be used. Thus, the eel library is used to connect the user input with the Python functions to query the Neo4j graph database using the Neo4jNavigator class and prepare the results. Those are then visualized in the local web browser environment using an interactive web map generated with the Leaflet library. The GUI itself consists of two parts. Once the main application is started via the GUI script, a splash screen appears whilst the k-d tree is generated for allowing the usage of latitude and longitude coordinates. After this step is completed the main window of the application opens. It includes a selection panel on the left and a map display on the right. The map display is automatically updated to show the extent of the CityGML dataset by using the bounding box values of the city model. This bounding box is additionally visualized via a rectangle. The selection panel contains input fields for the start and destination. Here, a UUID, latitude and longitude values, or an address can be inserted. Furthermore, if the input field is selected, the user can click on a point on the map and the coordinate values are automatically inserted into the input field. Below the location selection, two radio button list selections allow the user to select a weight as well as the algorithm that shall used for routing. Lastly, a button below those selection elements allows the starting of the shortest path searching process. This button can only be clicked when both input fields for start and destination are filled. Additionally, it is disabled until a result is returned from the database. This prevents triggering multiple shortest path searches at once. The following figure 4.7 shows the GUI with the selection panel on the left and the map display on the right. A result is then also presented in the map display in the screenshot figure 4.8.



Neo4j Navigator

Please enter start location...

Please enter destination...

Weight Options

[✓ Distance (simple)]
Distance (accurate)
Inclination
Width
Speed
Time

Algorithm Options

[✓ Dijkstra]
A*

Start Routing

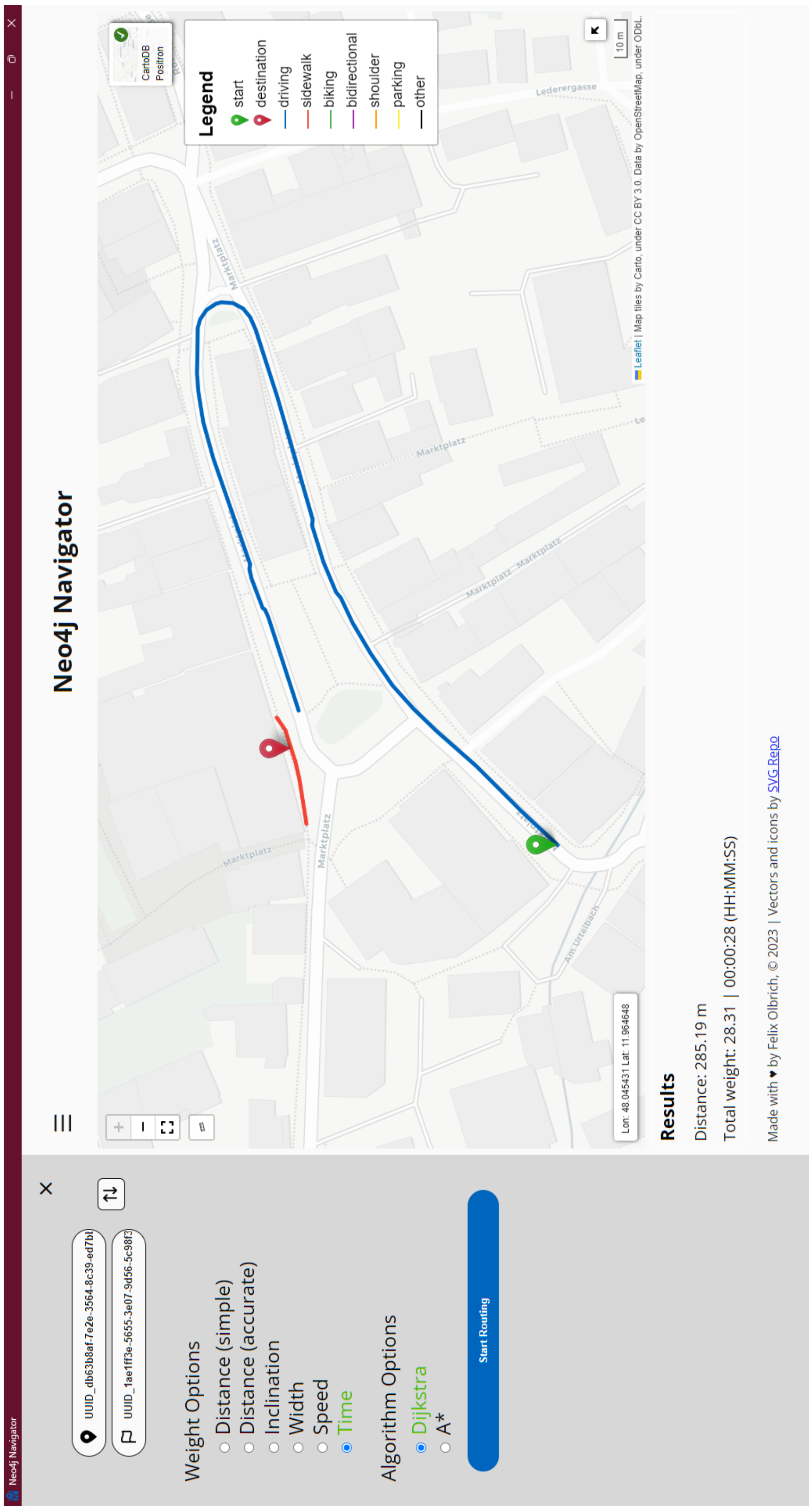
Lon: 48.056295 Lat: 11.927633

Legend
start
destination
driving
sidewalk
biking
bidirectional
shoulder
parking
other

Made with ♥ by Felix Olbrich, © 2023 | Vectors and icons by SVG Repo

Leaflet | Map tiles by Carto, under CC BY 3.0. Data by OpenStreetMap, under ODbL.

Figure 4.7.: Graphical User Interface of the Neo4j Navigator



Neo4j Navigator

UUID_db63b8af-7e2e-3564-8c39-ed7bl

UUID_1ae1ff3e-5655-3e07-9d56-5c98f3

Weight Options
- Distance (simple)
- Distance (accurate)
- Inclination
- Width
- Speed
- [✓ Time]

Algorithm Options
- [✓ Dijkstra]
- A*

Start Routing

Legend
- 🟢 start
- 🔴 destination
- driving
- sidewalk
- biking
- bidirectional
- shoulder
- parking
- other

Lon: 48.045431 Lat: 11.964648

Results

Distance: 285.19 m

Total weight: 28.31 | 00:00:28 (HH:MM:SS)

Made with ♥ by Felix Olbrich, © 2023 | Vectors and icons by SVG Repo

CartoDB
Positron
Marktplatz
Ledererergasse
Am Urtelbach
10 m
Leaflet | Map tiles by Carto, under CC BY 3.0. Data by OpenStreetMap, under ODbL.

Figure 4.8.: GUI of the Neo4j Navigator showing a multimodal result.

4.2. Multimodal Routing

To perform multimodal routing, the pre-processing step of adding switch connections between two modes of transport is crucial. Without this, shortest path algorithms work, as long as there is a connection between the start and destination location provided by one transportation type. This can be seen in figure 4.9. The screenshot shows the result of a query using the single mode Dijkstra function of the Neo4jNavigator class in the Neo4j Browser view.

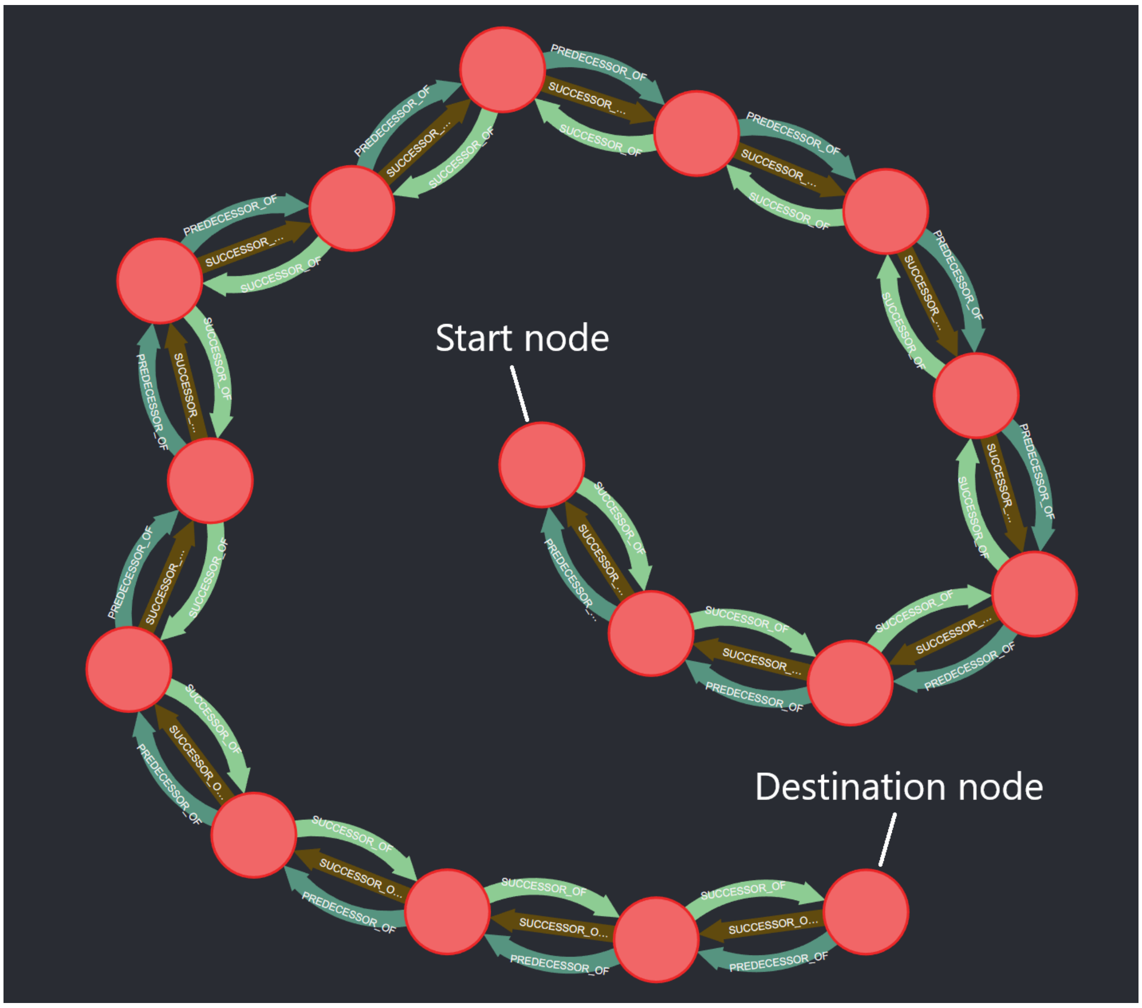


Figure 4.9.: Shortest path without switch connections in the Neo4j Browser.

In red, the TrafficSpace nodes can be seen. Those are connected from the marked start to the marked destination node via three connections. Three connections exist because the type of the nodes is *'SIDEWALK'*. Here, the light green arrow indicates a successor relationship (*SUCCESSOR_OF*) while the two arrows in the opposite direction represent the predecessor relationship (dark green) and the second successor relationship (*SUCCESSOR_OF_2*) in brown which allows to traverse the sidewalk in both directions. In the Neo4j Browser, the positioning of nodes does not represent the spatial location or distance to other nodes. Rather the nodes are positioned optimally around a center point in a planar embedding if possible. A transportation mode switch was not possible as only *SUCCESSOR_OF*, *SUCCESSOR_OF_2* and *NEIGHBOURS_LANE* relationships are allowed to be traversed. The Cypher query can be seen in the code 4.6. However, if the start and destination location require to switch between transportation modes as is the case when they lie in different transportation networks, no route is found.

```
 MATCH (from:`org.citygml4j.core.model.transportation.
TrafficSpace`{id:"UUID_51a69d53-5195-31e5-b1dd-8a81811104c0"
}), (to:`org.citygml4j.core.model.transportation.TrafficSpace
`{id:"UUID_c6490fd4-7be3-3006-abca-321a5dd6409d"}) CALL apoc.
algo.dijkstra(from, to, 'SUCCESSOR_OF>|SUCCESSOR_OF_2>|
NEIGHBOURS_LANE>', 'advanced_segment_length') yield path as
path, weight as weight RETURN path, weight
```

Code 4.6: Cypher query for shortest path without switch connections.

As the multimodal aspect is one of the key elements of this thesis, the remaining analyses are performed with a multimodal dataset and multimodal routing queries. Figures 4.12 to 4.19 show different shortest path results for start and destination combinations using the weights **accurate distance** (A), **inclination** (B), **width** (C), and **time** (D). The following UUIDs were used:

#	**Start UUID**
1	UUID_7ab89af9-b7be-343b-b676-139155f225b8
2	UUID_ba5a6e93-e86f-3984-90c2-b60832803dce
3	UUID_1731f3e1-ce60-3aef-a145-20d7ad1df52f
4	UUID_db63b8af-7e2e-3564-8c39-ed7bb6e456aa
#	**Destination UUID**
1	UUID_e821f115-6310-3f6c-a768-c94584f65b9e
2	UUID_bbf8537c-3f1e-328a-b928-08b178e8c05b
3	UUID_570a2073-09f7-3fe4-98e7-70de5745b151
4	UUID_1ae1ff3e-5655-3e07-9d56-5c98f34e7d0a

Table 4.1.: UUIDs used for the shortest path analysis

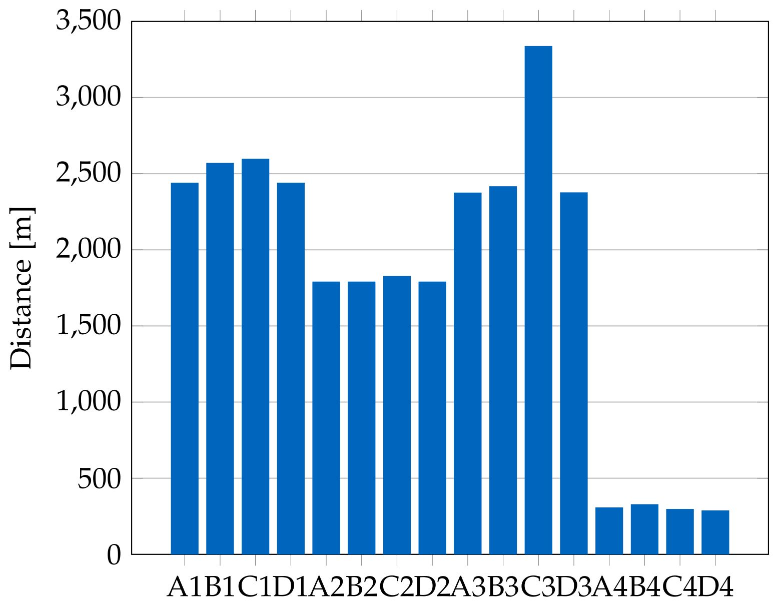


Figure 4.10.: Dijkstra Routing Analysis Results - Four different routes four weights each: accurate distance (A), inclination (B), width (C), and time (D).

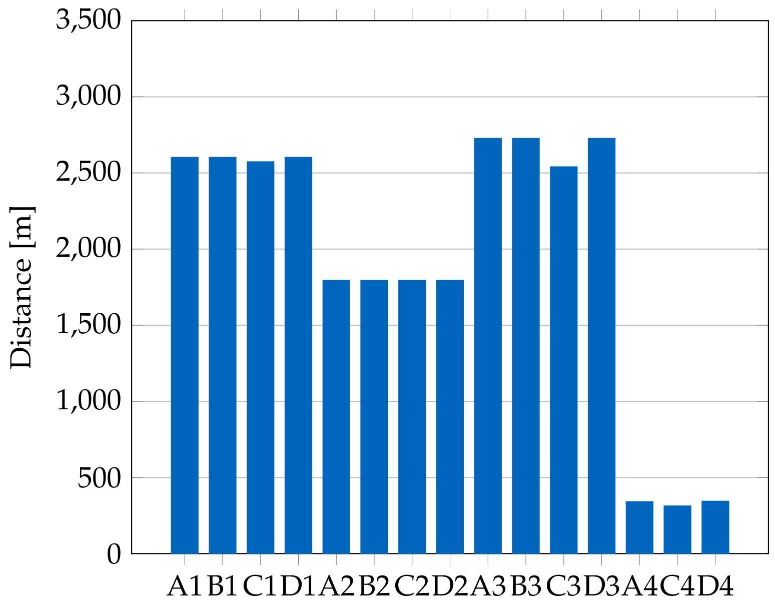


Figure 4.11.: A* Routing Analysis Results - Four different routes four weights each: accurate distance (A), inclination (B), width (C), and time (D).

This results in 16 routes per shortest path algorithm, with blocks of four weights for one route. To compare the results using the two shortest path algorithms as well as to compare the optimal paths found by the different weights, the comparison of the four weight options by final route length (Euclidean distance of start and end point of each TrafficSpace geometry) was chosen. This metric shows whether the overall travel distance is equal or changes depending on the selected options. A similar value suggests that the final route looks similar too. This can be confirmed by comparing the actual path results as visualized in figures 4.12 to 4.15 for Dijkstra and figures 4.16 to 4.19 for the A* algorithm.

The blocks in figures 4.10 and 4.11 can be differentiated well with four bars belonging to one start and destination combination. The corresponding combinations are listed in table 4.1. Bar values 0-3 on the x-axis belong to the first combination, 4-7 to the second and so on. When comparing the two algorithms it is apparent that the A* variant overall results in larger values for the total route weight except the width weight in block three. Comparing the weights in each block for the Dijkstra algorithm, the resulting routes look similar, which can be confirmed by the map visualization. The same applies to the A* algorithm. However, in block two even all weight options result in the same route.

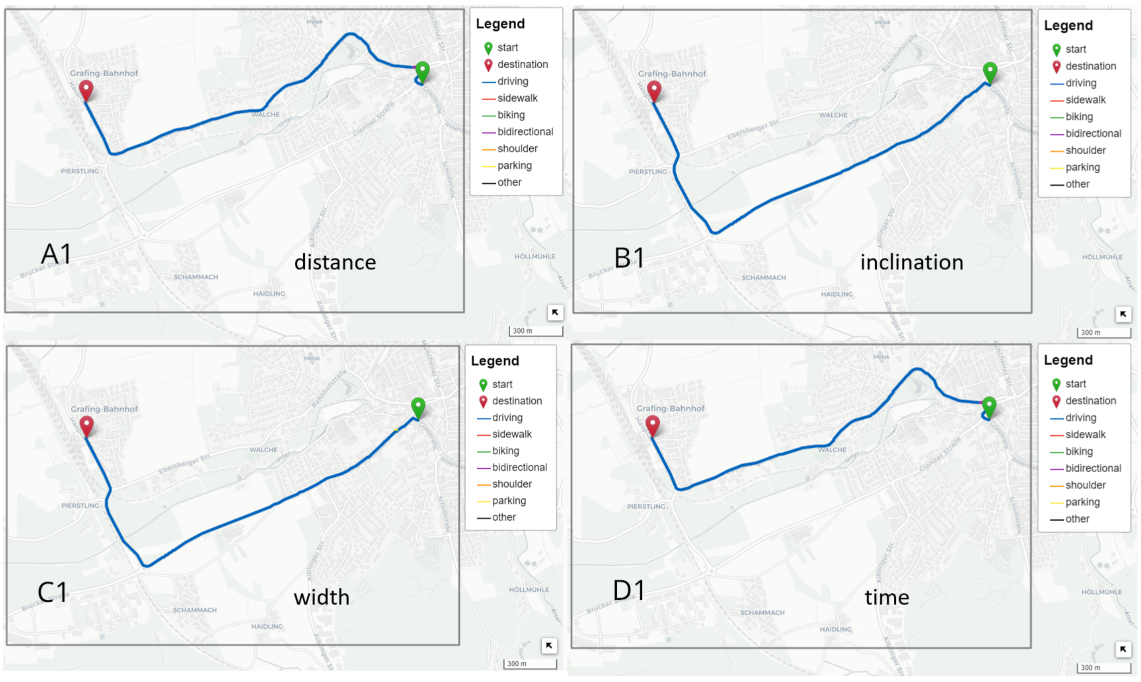


Figure 4.12.: Dijkstra Shortest Path Combination 1 Map Display

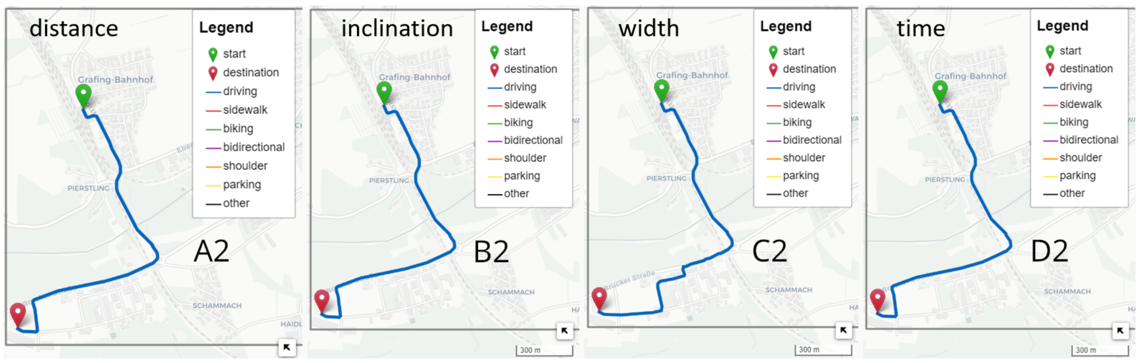


Figure 4.13.: Dijkstra Shortest Path Combination 2 Map Display

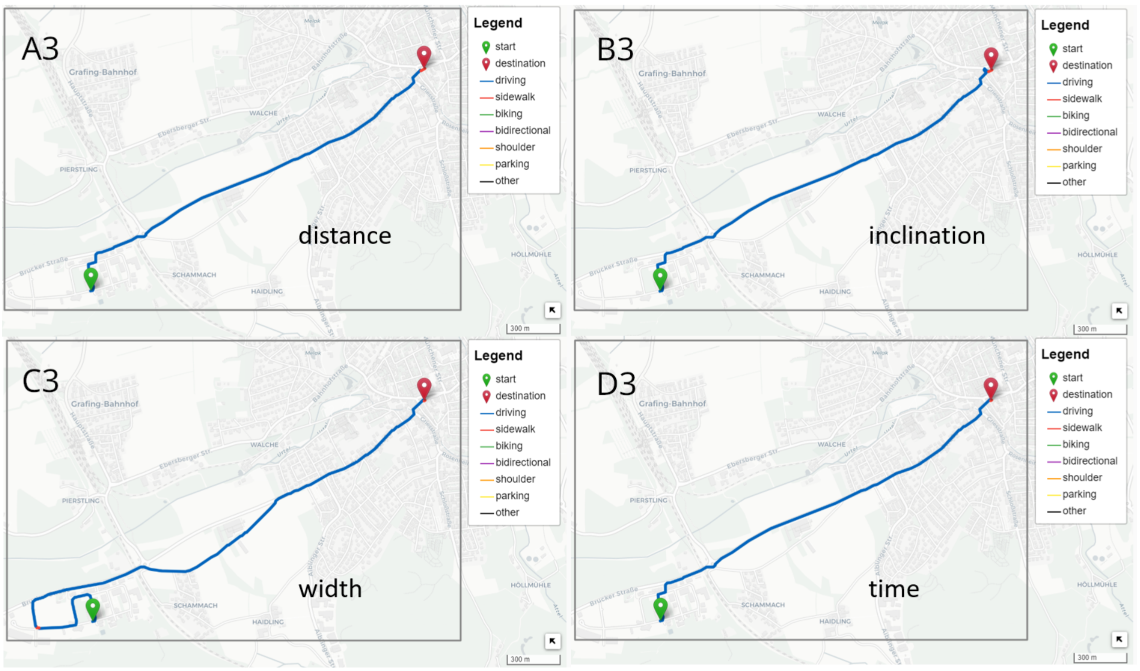


Figure 4.14.: Dijkstra Shortest Path Combination 3 Map Display

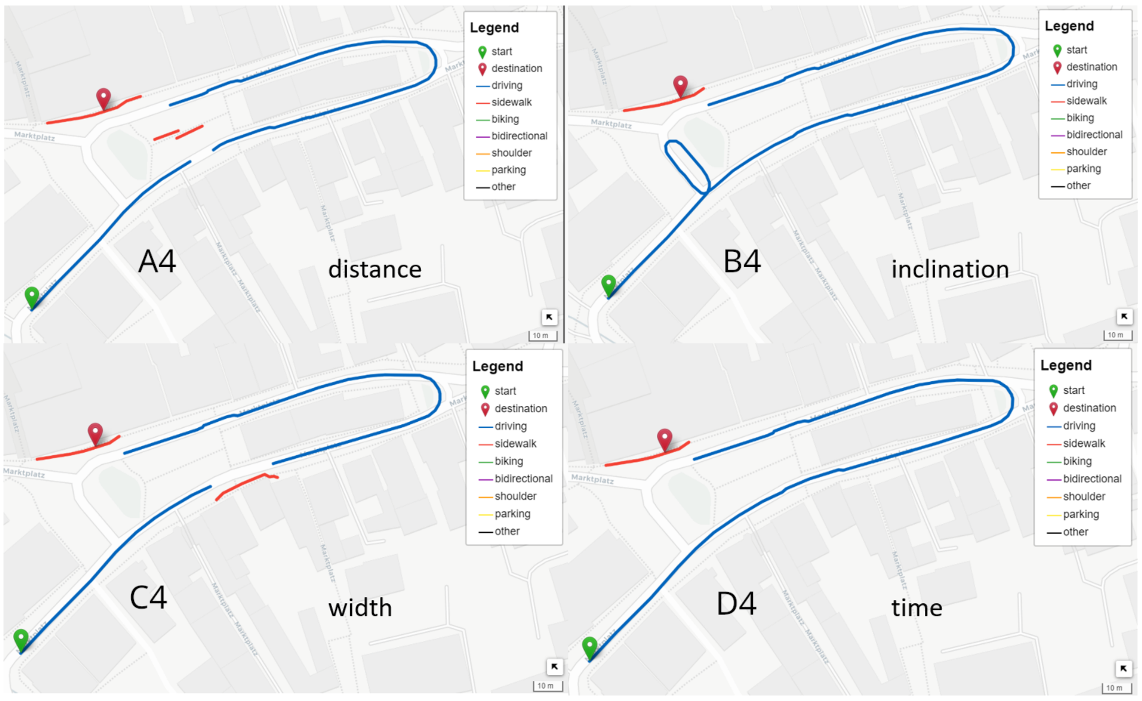


Figure 4.15.: Dijkstra Shortest Path Combination 4 Map Display

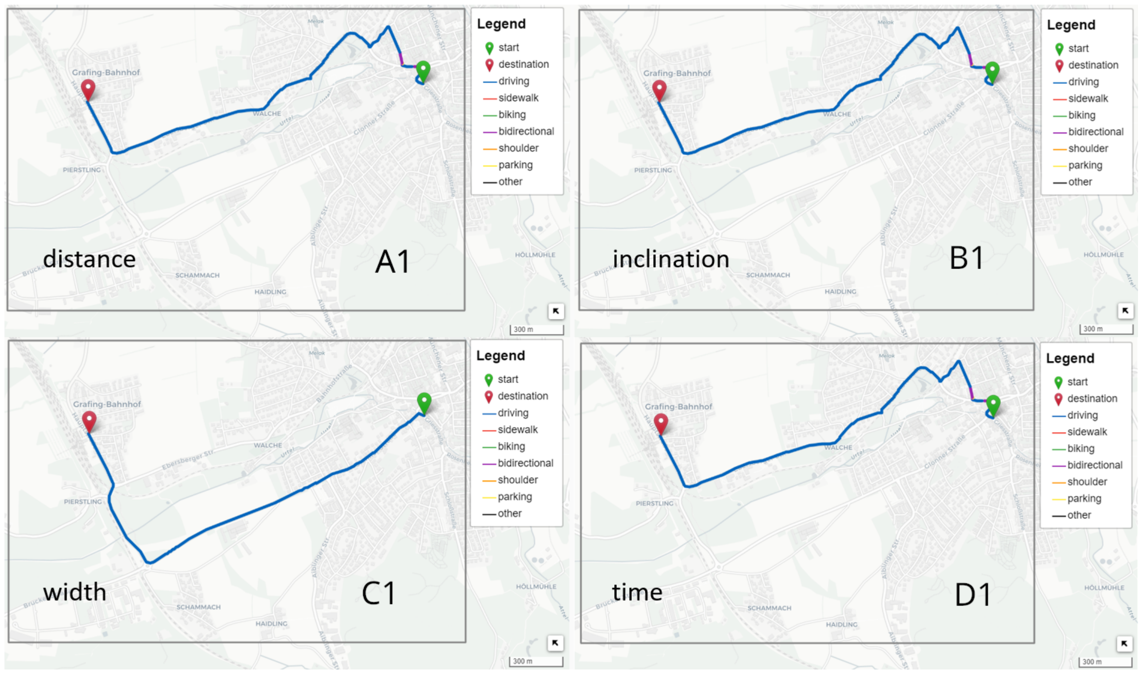


Figure 4.16.: A* Shortest Path Combination 1 Map Display

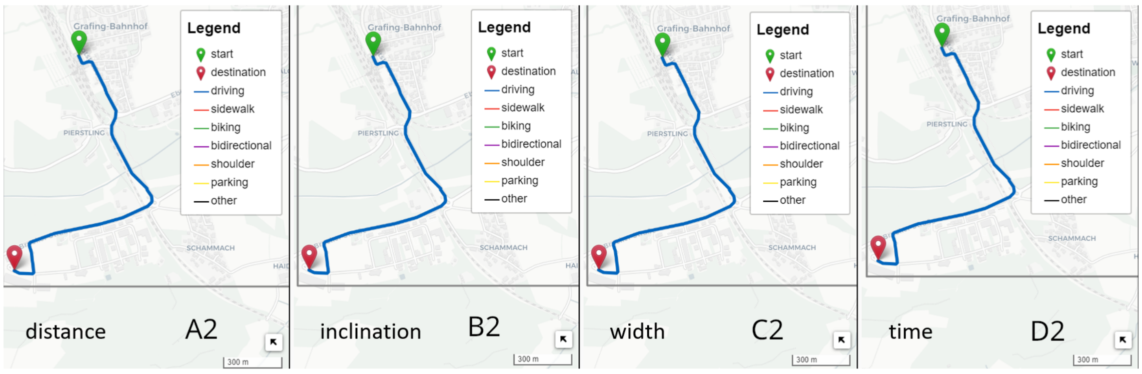


Figure 4.17.: A* Shortest Path Combination 2 Map Display

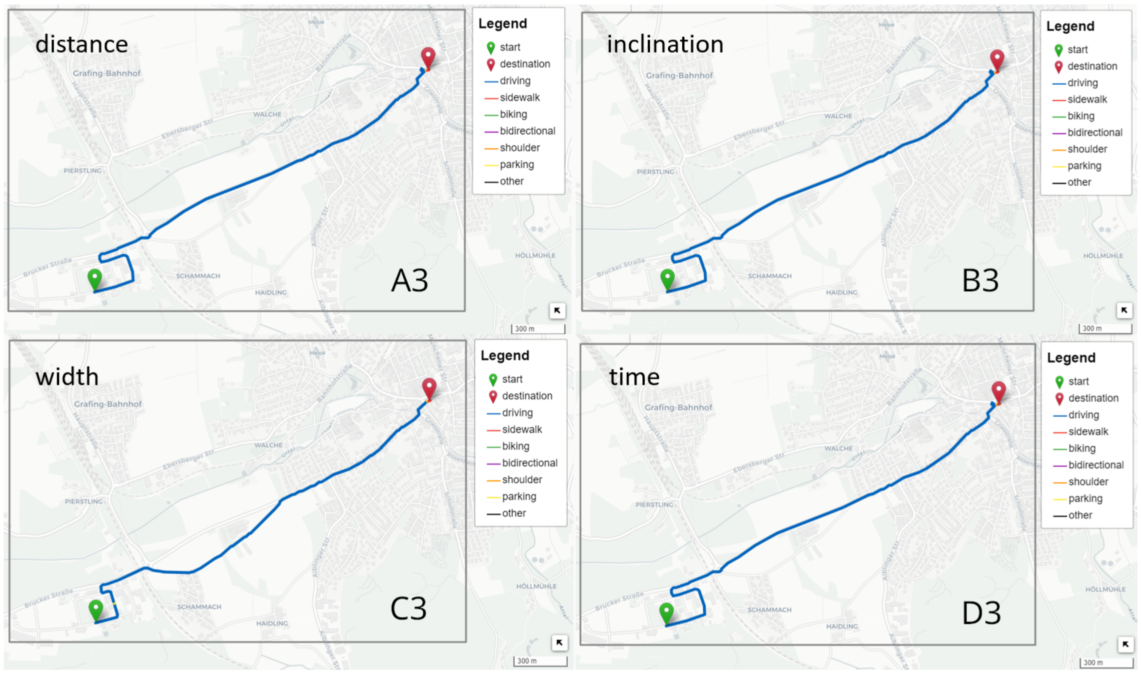


Figure 4.18.: A* Shortest Path Combination 3 Map Display

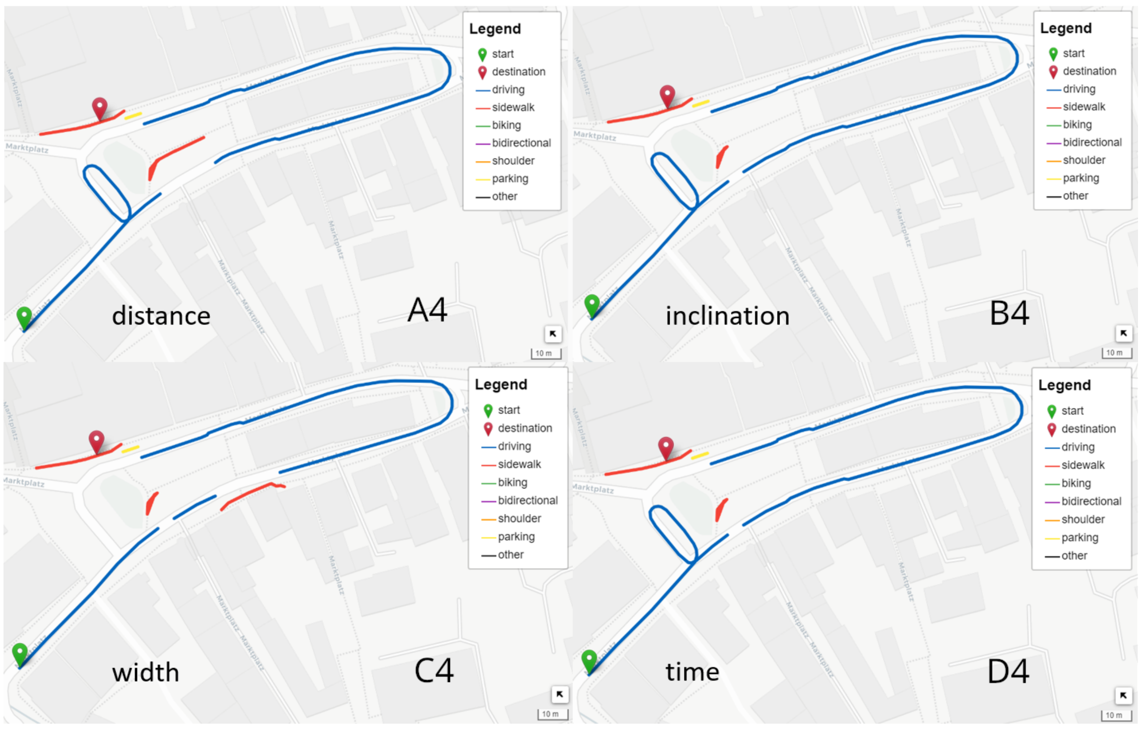


Figure 4.19.: A* Shortest Path Combination 4 Map Display

Overall, the results of the shortest path algorithms are to be expected. Some optimal paths change depending on the chosen weight, e.g., in figure 4.12 it can be seen that weights distance and time return an optimal route via a northern path and width and inclination return a southern path. There are some results that require some more detailed analysis as they do not look right at first. In figure 4.14 the optimal path for the width weight is unintuitive at first, as it contains more turns at the beginning of the route. However, if one takes a look at the weight calculation and data it is clear why this route is returned. First, the weight prioritises wider lane segments which is calculated via the inverse minimum width. Second, turn segments are modelled especially wide. This is an artefact from the OpenDRIVE data where multiple lanes can overlap. The turning lanes in a crossing are wider than the normal lane. Thus, the wider minimum segments result in overall smaller weights. The last noticeable result is present in figure 4.14 for the inclination, where it is difficult to detect due to the map scale, but it also appears for the inclination weight in figure 4.15 and figure 4.19 for the weights accurate distance, inclination and time. The loop is also visualized in a magnified version in figure 4.20. This result can be explained by the calculation of the inclination weight and the A* node prioritizing. First, the calculation of the inclination only uses the start and end points of the geometry, thus it is possible to "jump" over smaller bumps or bulges in the surface when those only occur in the middle of the segment. In this selected location the turn lanes contain rather long segments that allow to "jump" over bulges that are existent in the segments going straight. Thus, it is possible to calculate a smaller weight value by using the loop connections with two large turning segments allowing to ignore the higher height introduced by the small bulge. With regard to the results of the A* function implemented in the APOC extension, there are more changes in the optimal routes. First, the distance and time weight utilize the loop connection as well. The reason for this lies in prioritizing the nodes that are closer to the destination node. As the heuristic prefers nodes that are closer to the destination, based on the Euclidean distance, the shortest paths between Dijkstra and A* can vary. The same applies to the different route segments chosen for combination 1 as can be seen in the magnified view in figure 4.21. Here, the final and longer route is chosen based on the first segment that is closer to the destination. Once the A* algorithm has found a path, the procedure is stopped and other, possibly shorter paths are ignored. This shows again the importance of smaller segments in crossing areas.

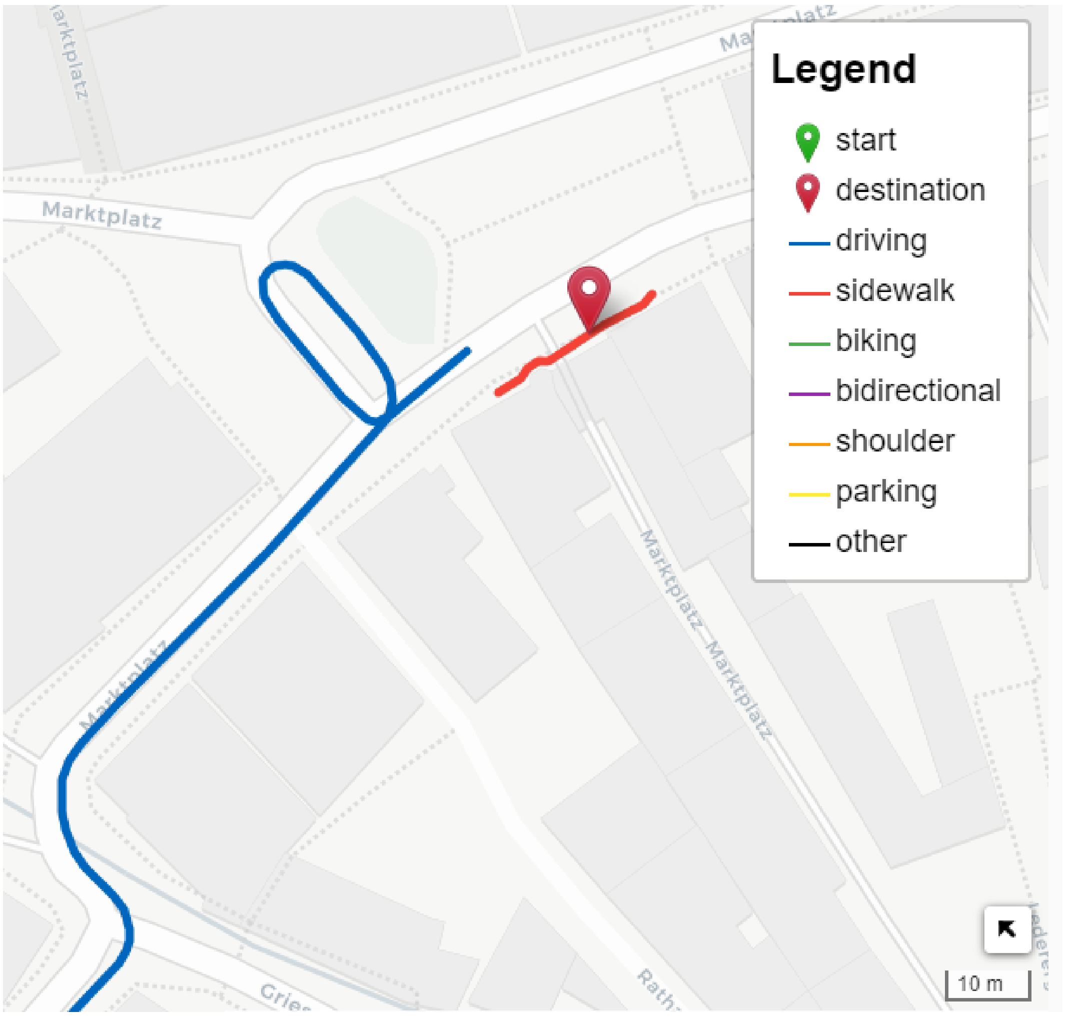


Figure 4.20.: Special Remark Inclination Weight

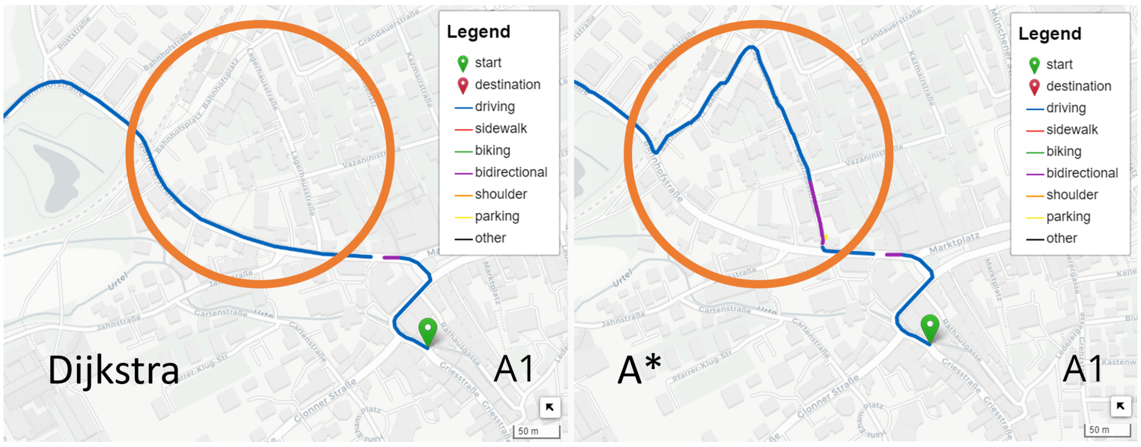


Figure 4.21.: Shortest Path Result Comparison Combination 1 - Dijkstra vs. A*

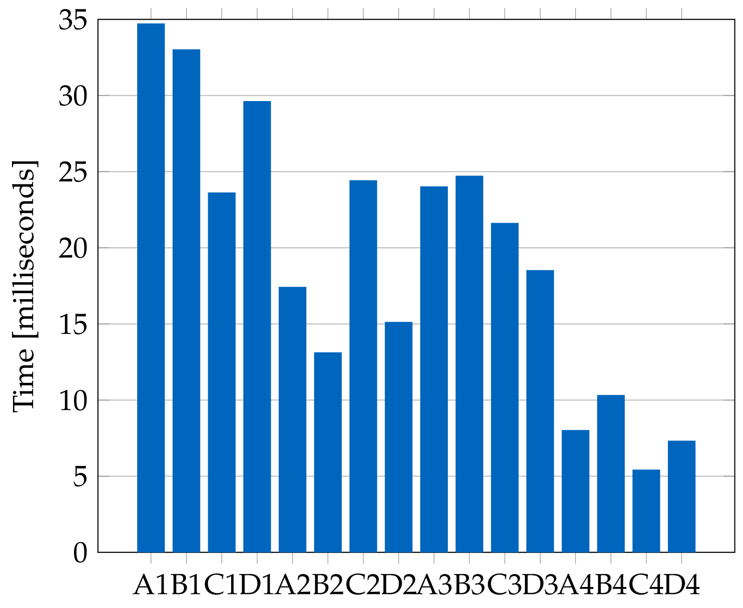


Figure 4.22.: Routing runtime comparison using the Dijkstra function.

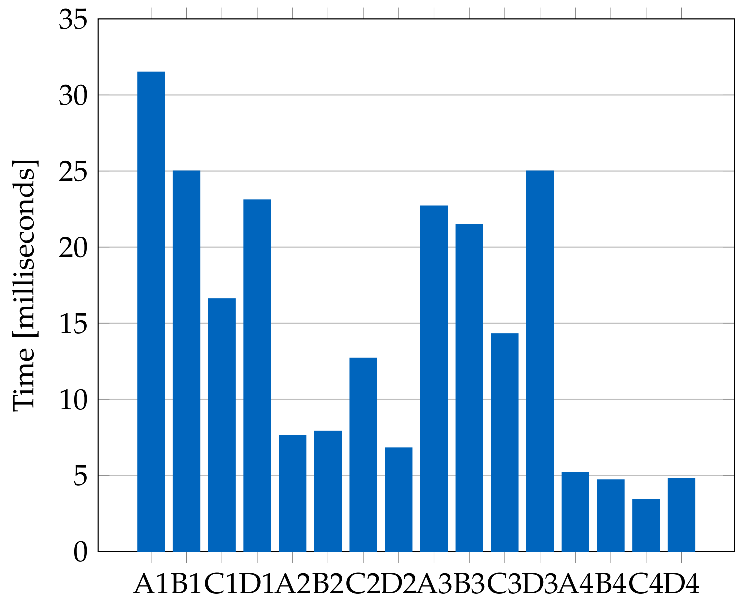


Figure 4.23.: Routing runtime comparison using the A* function.

When comparing the runtimes the results look as expected. The runtime was obtained by the Neo4j Browser. Under the table tab of the results, a runtime for the query is listed. For the comparison, each route and weight combination was run ten times and the total runtime was noted. The average of the ten runtime measurements is presented in figure

4.22 for the Dijkstra function and figure 4.23 for the A* function. The figures follow the structure of figures 4.10 and 4.11 with blocks of four for each route combination with a bar for each weight. Thus, bars 0-3 belong to the combination 1, 4-7 to the combination 2 and so on. It can be seen that the runtimes roughly match the length of the paths. Longer route results have a higher runtime than the shorter paths. Additionally, it can be seen that the A* performs overall better than the Dijkstra algorithm when it comes to the runtime. With runtimes in the millisecond range, the shortest path search is fast enough to be used in a real-time application. However, the runtime is also dependent on the hardware used and was not tested on different machines and hardware configurations. The full list of runtime measurements can be found in appendix A.

4.3. Advanced Applications using Capabilities of CityGML

4.3.1. Parking Garage Routing

Besides navigating the street space, CityGML also allows the modelling of street elements within a building. This allows the correct representation of buildings used for parking vehicles, such as garages. In large parking buildings, it is often difficult to find a free parking space. Thus, an advanced parking system could track the available parking spaces and communicate them to the car navigation system to guide the driver to the nearest free parking space. Such a scenario can be tested using the concepts of CityGML by connecting TrafficSpaces inside and outside of building elements. In the implementation, it could be shown that it is possible to use TrafficSpace objects of different granularity as well as elements inside a building or in the default street space. The synthetic dataset consisted of the Grafing data with an additional hand-modelled parking garage. Due to the two different modelling aspects not all of the previously developed concepts could be used. The main issues are missing additional information that is provided via the generic attributes and the different granularities of both the street space and the building elements. Thus, only the following concepts were implemented: Routing using the predecessor and successor connections as well as all lane changes and transportation mode switches that are within the Grafing data. Weight calculation was ignored during the pre-processing and is using the default behaviour of the APOC shortest path functions; counting the number of traversed nodes. Furthermore, the routing was split into two parts, first detecting if the result of a routing query contains the entrance node of the parking garage. This is important as the entrance serves as a connection to both the *'DRIVING'* and the *'SIDEWALK'* type. Thus, resembling a transportation mode change. However, continuing with the logic a direct connection

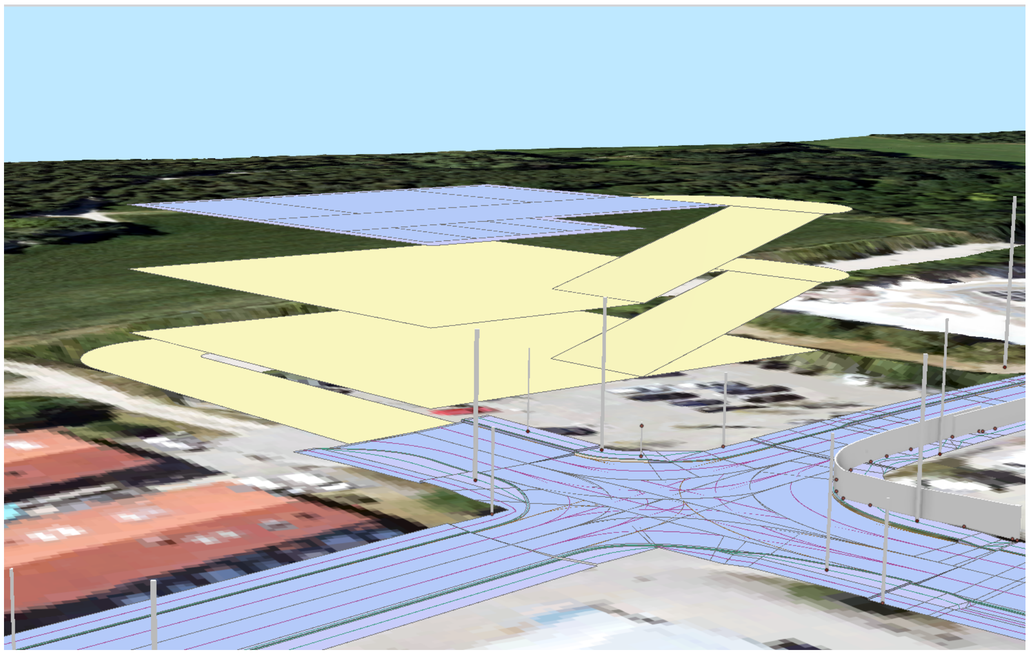


Figure 4.24.: Parking Garage Building CityGML, TrafficArea and Section planes visualized in ArcGIS Pro

allows a driver to drive on the sidewalk. Therefore, this case is broken down into two parts. Once a route uses the entrance a secondary query is performed routing to a randomly selected TrafficSpace within the uppermost level of the parking garage, which serves as a parking lot deck. Then another query uses this parking lot TrafficSpace as its start location and continues the routing to the previous destination segment. This way the opposite routing works as well. When a routing procedure is started from a *'SIDEWALK'* element only accessible via the parking garage. The final returned result is a combination of the segments leading to the garage, the routing within the building and the routing from the parking lot within the building to the destination. If additional weights are used, the transportation types of driving and walking could be addressed separately as well.

As can be seen in figure 4.26, the selected parking spot is located on the uppermost level of the parking garage. The blue colour gradient shows the order of traversed TrafficSpace elements starting with the bright light blue TrafficSpace neighbouring the *'DRIVING'* type TrafficSpace. The last element is the dark blue TrafficSpace with the UUID *"trafficspace1"*. Then the parking spot is reached. For the routing inside the garage building from the parking spot, the same procedure is applied. In figure 4.27, the reverse route is visualized. Again, a blue gradient shows the order of the traversed TrafficSpaces. The first element is in light blue to the last element is in dark blue.

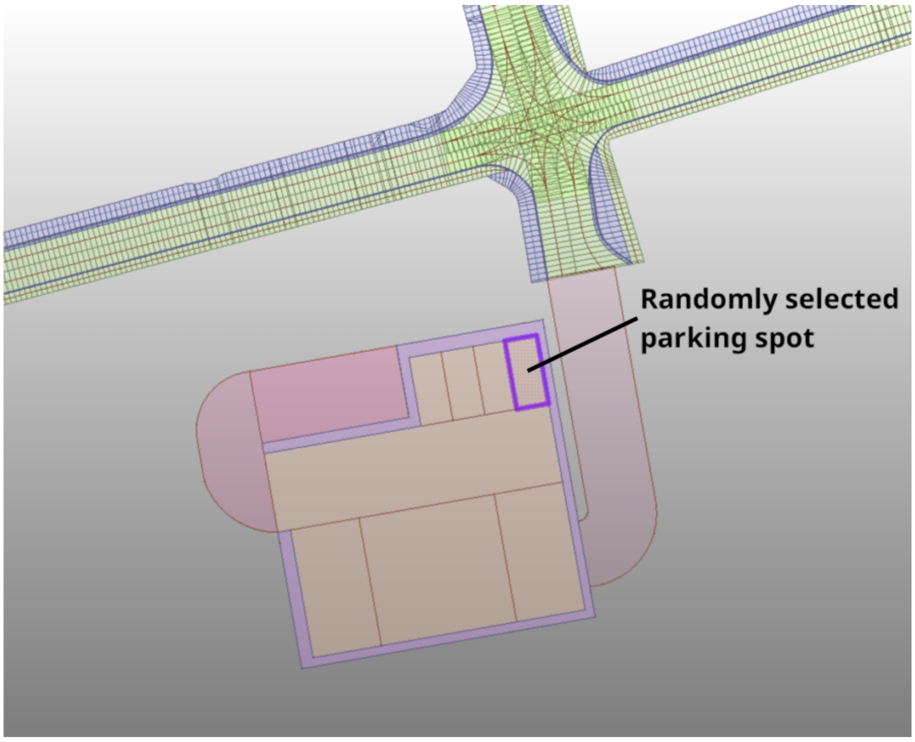


Figure 4.25.: Garage Building - Selected Parking Spot visualized in FME Data Inspector

Figure 4.28 shows the representation of the different granularities, with a volumetric representation of TrafficSpaces in the parking garage building and linear TrafficSpaces in combination with corresponding 2D TrafficAreas.

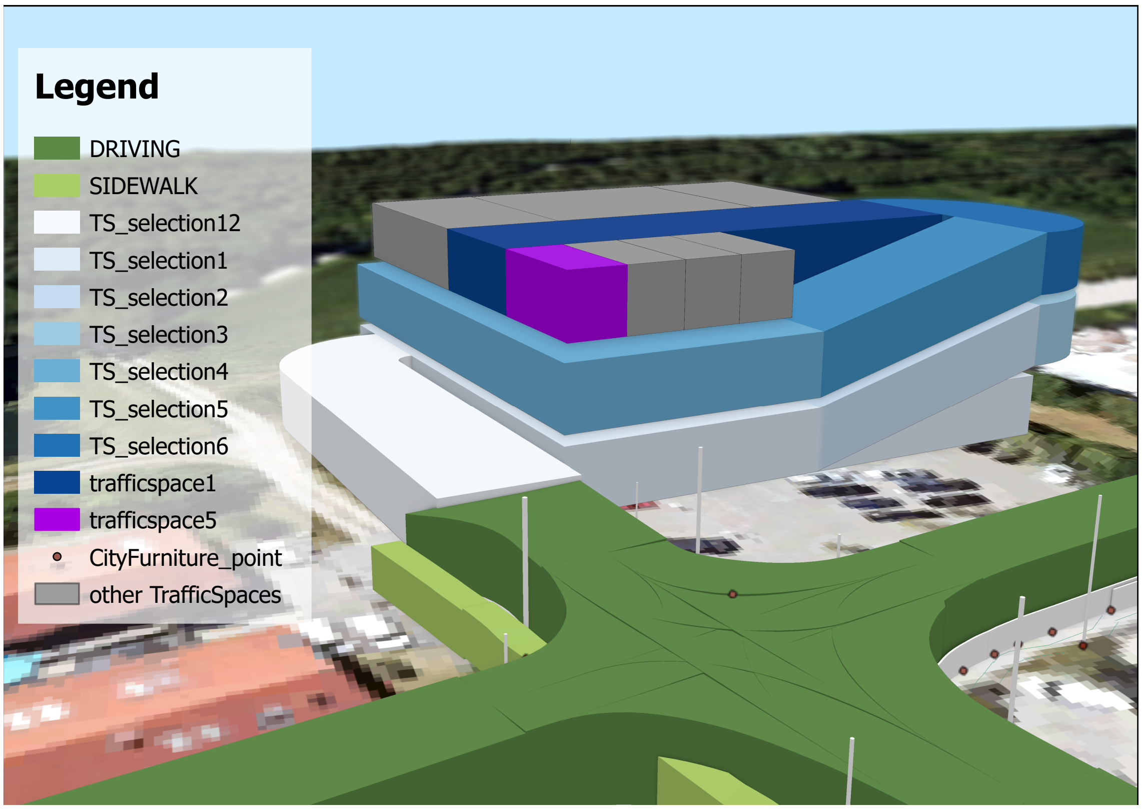


Figure 4.26.: Parking Garage - Routing to Parking Spot: The routing starts at the white TrafficSpace and follows the blue colour gradient to the parking spot in purple.

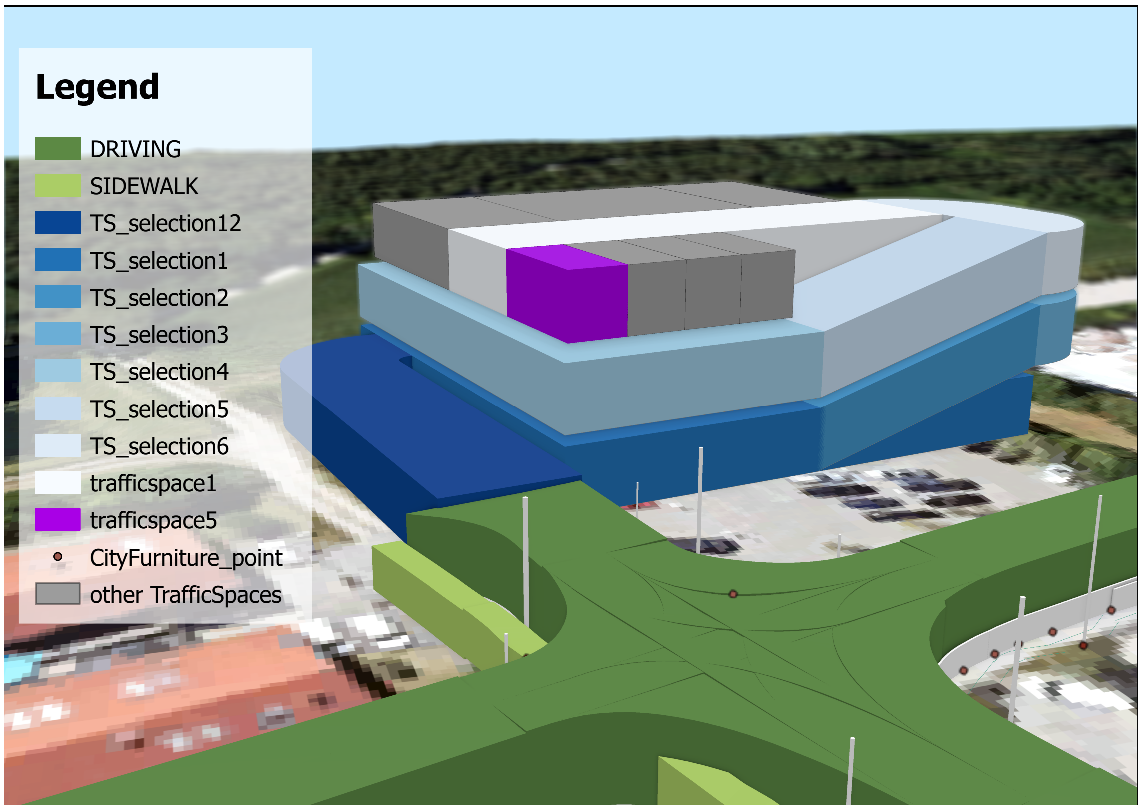


Figure 4.27.: Parking Garage - Routing from Parking Spot: The routing starts at the parking spot in purple and follows the blue colour gradient to the exit TrafficSpace in dark blue.

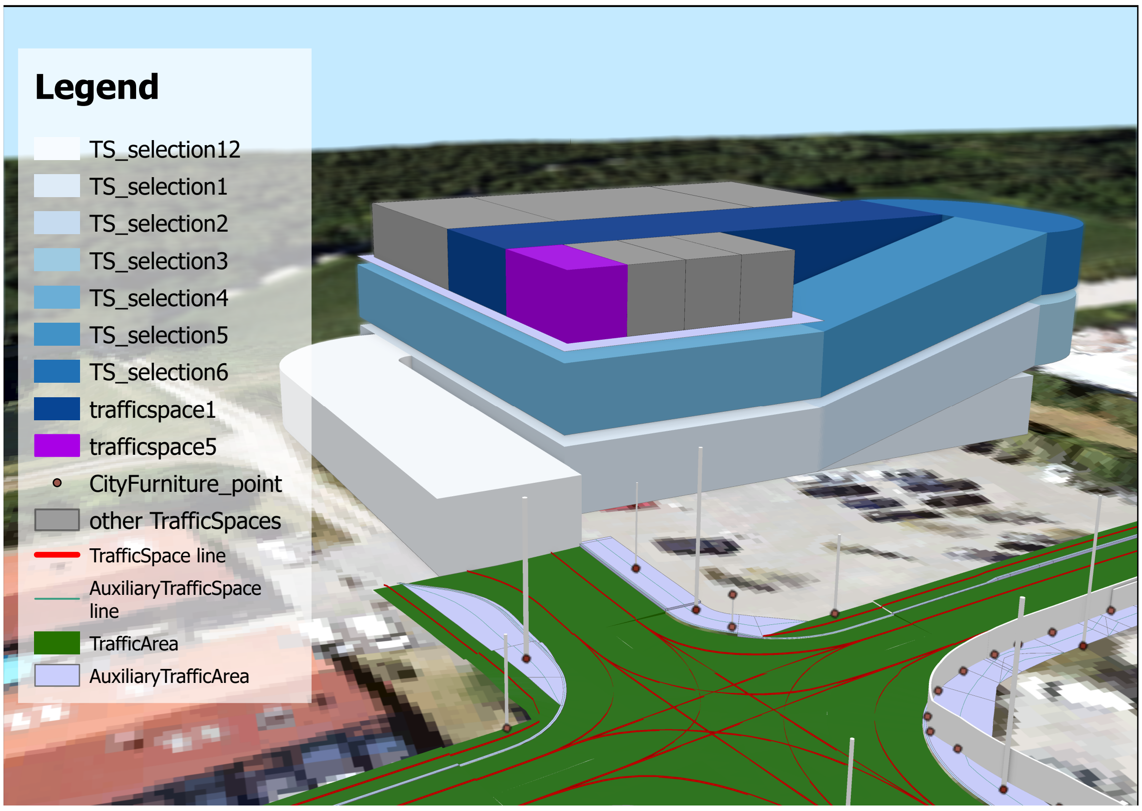


Figure 4.28.: Parking Garage - Routing to the parking spot showing the different granularities of the dataset. The garage uses a volumetric representation of the TrafficSpace in the granularity ’way’. The rest of the data uses a linear representation with the granularity ’lane’ for the TrafficSpaces.

4.3.2. Improvements based on findings

The results and development of the implementation of this thesis can be used for other applications as well. Based on the findings of this thesis, some improvements have been introduced to the r:trån transformation software. These changes include the correct connection direction of successor and predecessor connections based on the traversal direction. As mentioned, the previous connection followed the reference line of OpenDRIVE which is not present in CityGML. Thus, the linkage of successor and predecessor elements was unintuitive. Now, the successor and predecessor XLinks follow the traversal direction of the road. The travel direction attribute *"trafficDirection"* is still present and can be used to verify the links and is important for bidirectional connections. As a second change, CityFurniture elements are now related to the nearest TrafficSpace elements. Thus, allowing the usage of street signs in combination with the information present at the TrafficSpace. This information can be used via new XLinks called *"relatedTo"*. The third modification is based on the lane-changing functionality. This information is now available via the *"relatedTo"* linking as well. The differentiation is made using the relation type ("leftLaneChange" or "rightLaneChange").

5. Discussion

5.1. Test Setup

All tests were done on a local hardware setup with an Intel(R) Core(TM) i7-10870H CPU @ 2.20GHz processor, 32GB of RAM, and a NVIDIA GeForce RTX 3080 Laptop GPU. The operating system was Windows 11 Pro 64-bit. For software, the Neo4j Desktop version 1.5.9 with a Neo4j database version 5.12.0 was used. The Python version 3.10.8 was used. The full list of Python packages used can be found in the appendix B. The input database for the full Grafing dataset has a size of 1.80 GB. In the database, there are already 6,428,260 nodes and 6,552,576 relationships. After pre-processing the graph database contained 6,428,260 nodes and 6,574,711 relationships with a total disc space of 1.87 GB. It can be seen that the number of nodes stays constant and only the number of relationships increases by **22,135**.

The database containing the garage building dataset has a size of 2.66 GB. In the database, there are already 7,681,023 nodes and 7,804,069 relationships. After pre-processing the graph database size increases to 2.75 GB. The number of nodes stays constant again and the number of relationships increases by **22,190** to 7,826,259 nodes.

5.2. Discussion of Results

5.2.1. Shortest-Path Analysis

Running the final application with the provided test datasets showed that the shortest path analysis works as expected. As the test datasets were small, the results were calculated in a few milliseconds, see figures 4.22 and 4.23. When viewing the results in the GUI, some path results seem unintuitive or even wrong. However, when inspecting the resulting chain of TrafficSpace objects, e.g., in the FME Data Inspector, it can be seen that the result is correct. Such strange routing results are caused by the way the resulting shortest paths are visualized in the GUI compared to the available connections in the graph database. The base map, for example, does not always align perfectly with the routing network. A base map can also suggest routes that do not exist in the routing network due to missing connections and does not reflect the modelling of certain aspects, like the wider segment of turning lanes at crossings. Choosing different weights also does not always result in different routes, as there are few alternatives in the test dataset. However, for selected start and destination combinations, the results are as expected with the chosen weights, for example, prioritizing wider streets or faster routes. Only the results of the A* functions are surprising as they in general prioritize route segments that lead to higher weights compared to the Dijkstra algorithm. The runtimes are however as expected lower for the A* algorithm. This is due to the heuristic function that is used to estimate the remaining distance to the destination.

5.2.2. Concepts

To evaluate the developed concepts, the weight functions, multimodal routing and the use of different granularities are analysed.

Weight Functions

As important as advanced weights are for realistic routing results, the base functionality of finding the shortest path in a network is the main priority of this thesis. It could be shown that the chaining of elements via predecessor and successor links can be used to derive a network to perform shortest-path analyses. Furthermore, besides counting traversed edges to find the optimal path between two nodes, the information available in the CityGML model is sufficient to perform more advanced shortest-path analyses. The test dataset could provide explicit and implicit information to derive distance, inclination, width, speed, and traversal time weights. Additional information from the generic attributes can be used to extend the list of weights. Nevertheless, some assumptions were necessary to supply each edge with all the required weight information. Furthermore, as there is no standardized form of storing additional information for routing purposes, the information has to be derived from the CityGML data and other sources. Here, the use of generic attributes can be helpful to store additional data. However, the development of an ADE to define a standardized way of storing routing information would be beneficial. At the moment, the various weight information needs to be stored separately and is only available in the graph database. This requires additional processing of all elements every time the CityGML data is updated. A better solution would be to store the calculated weights inside the CityGML structure and change them whenever the parent element changes. Furthermore, when exchanging the underlying CityGML data, the calculated weights are lost as they are not stored in the CityGML data itself.

In the case of the garage building dataset, additional challenges for calculating weight information arose. As the garage is modelled in another granularity - way instead of lane - and TrafficSpaces represented as volumes instead of lines the developed pre-processing methods do not apply and the weight information has to be derived differently. Nonetheless, using the shortest path analysis without specified weights and weighting by the number of visited nodes, a combined usage of different granularities is possible. Thus, only an improved pre-processing methodology for addressing the different granularities is needed.

Multimodal Routing

The developed concepts and the application can be used for multimodal navigation. However, when testing the implementation it became apparent that some more advanced strategies are needed to make the results compliant with real-world scenarios. The main issues encountered in the multimodal results were the use of shortcuts and switching between car and pedestrian multiple times. For the shortcuts, the transport mode was shortly changed to take advantage, e.g., of wider road segments or shorter sidewalk paths to optimize the overall costs. This connects to the second issue of multiple switches between transport modes that should only be used once, for example when parking the car and continuing as a pedestrian it is necessary to return to the car to continue by car. However, this is not the case for the current implementation. Here, a car is parked and the person can continue as a pedestrian using a new car at any parking spot. This is not realistic and should be improved in future work. As for the tested transportation modes, only a car and pedestrian type were used due to the available network specifications derived from the OpenDRIVE generic attributes in the Grafing dataset. Transportation modes that can switch between different transportation networks, such as a bicycle, are not supported by the current implementation. However, it could be tested that the routing functionality works independent of transportation type and the developed concepts can be adjusted to use and implement the use of additional transportation modes. The main challenge is to develop a modified version of the shortest path algorithms already available via the Neo4j APOC extension. This implementation should model the correct usage of the different transportation modes, e.g., only switching back to the car at the same parking spot by blocking further car usage until the destination is reached. This requires checking the current transportation mode and the currently allowed transportation modes during the shortest path analysis, which is not supported by the supplied APOC functions. Also, suitable switch locations for other transportation modes should be identified, e.g., bus stops or bike-sharing stations.

Different Granularities

Combining different granularities in one routing network was possible. As the successor and predecessor linkage is independent of the granularity representation a routing network can be generated. Regarding the shortest path analyses, only the connections between the elements are relevant. Nevertheless, as mentioned before, specialized pre-processing is required to extract weight information, e.g., when using geometries with different dimensions. Lastly, different granularities can provide different information, e.g., while a lane contains accurate information on transportation type, the granularity area cannot represent more than one transportation type for the whole street.

5.2.3. Implementation

Performance and Results

The performance and results will be discussed in three parts, the pre-processing, the routing, and the route visualization. First, the pre-processing can be improved by solving some of the tasks on the graph database side using Cypher queries and less Python to analyse the results. Additionally, runtime improvements could be achieved by using another language than Python for the additional analysis outside the graph database. The pre-processing further requires additional analysis to include more different weight functions if the data is available in another dataset. Furthermore, a modified version of the pre-processing concepts is needed to include all geometric information of different dimensions as weights. To use additional information, the generic attributes could be used similarly to the current state. However, the inclusion of all required information for routing in a standardized way would be beneficial. For this, the development of an ADE or the integration of further attributes in the CityGML standard is needed. Alternatively, a conversion of the calculated weights to generic attributes within the CityGML structure could be tested. This would allow the usage of the calculated weights in other applications and would not require additional processing of the CityGML data. One of the main advantages of the underlying data is the semantic data provided by the different elements which can be used to extend the navigation application in future work. The core concepts of routing, however, are dependent on the presence of successor connections. The predecessor connections can give additional information about the direction of the traversal, e.g., missing when two lanes merge at a crossing. In general, the current approach of routing on successor connections and introduced lane changes and transportation mode switches is limited to the availability of the successor connections. If elements are not connected via a successor/predecessor link or do not provide additional semantic information on their lane neighbour relationships, a geometric comparison fallback could be introduced. Here, spatial indexing structures like the presented k-d tree might be used in combination with a distance threshold to find neighbouring objects that can be connected automatically. However, a separate investigation is needed to determine the best approach for this. Another limitation of the routing implementation is the usage of the provided shortest path algorithms in the APOC extension. Here, a separate implementation could improve multimodal routing by adding functionality to check different parameters during the shortest path analysis, e.g., the current transportation mode and which transportation mode changes are currently possible. This would allow the implementation of more advanced routing strategies and would improve the multimodal routing results, e.g., the elimination of multiple transportation mode switches that are not realistic. Lastly, the route visualization was no core part of this thesis and thus only a basic visualization was implemented for testing purposes. The visualization of the results is not perfect and can be improved in future work by adding support for 3D visualization, different granularities and differentiation of transport types rather than semantic types of the elements, e.g., car and pedestrian instead of *"DRIVING"*, *"SIDEWALK"*, *"BIDIRECTIONAL"*, etc. Furthermore, the visualization could show the actual routing network that is available in the graph database. This would allow the user to understand the routing results better as the base map can suggest a different result, containing connections that are not present in the routing network. The implementation of querying the information of all elements of a result route takes a considerable amount of time, even for short routes, and should be improved. For production deployment, the visualization should use a different framework that supports 3D visualization, e.g., the Cesium viewer (Cesium GS, Inc., 2023). The GUI was only developed to provide easier access to the graph database and visualize the routing results to assist the evaluation of those results.

Limitations

The implementation has some limitations as well which have been addressed beforehand. Besides the conceptual and modelling issues other limitations include the availability of the data. At the moment there is no widespread data to perform routing analyses on. Compared to other navigation applications the spatial extent of the routing network is rather small. Additionally, the implementation does not include functionality to exclude certain elements from the routing, e.g., not using a certain type of street like toll roads or the autobahn. This is on the one side due to the limited test area which does not include such elements. On the other side, the functions use predefined weight functions and weights that are not combined during the runtime of the application. Thus, any changes must be introduced to the database before starting the application.

Of course, the results can only be as good as the underlying data. The CityGML data model does not enforce the presence of all information needed for advanced routing analysis. Thus, any subsequent applications rely on the information provided by the modeller. While the CityGML modelling standard can provide much information, a correct and complete representation of the traversable paths in the real world is necessary to get realistic results, too. Relationships to connect adjacent elements can be added to the graph representation, but the graph database does not contain more information than the linkages provided within the CityGML data source. To retrieve more information on neighbouring objects, additional analysis is needed. During the pre-processing some implicitly stored information can be used to add missing connections. While lane changes could be added in the example dataset, the predecessor and successor XLink concept of CityGML in general is not sufficient to represent all kinds of relationships between traffic spaces. Furthermore, other datasets might not contain additional information to recreate neighbouring lanes without additional analysis. Whilst these new connections are supported by the graph database, such analysis is limited in terms of pure database usage and external tools are required for advanced analysis and modification of the data.

5.2.4. Comparison to Other Approaches

When compared to line-based approaches which use a parametric representation, some functionality is missing. Here, the determination of restrictions through signs must be mentioned in particular. To represent the correct start and end positions of a restriction, the geometry of the segments must align with the location of the street signs. However, this can lead to additional small segments in the street network when larger segments must be cut in two to represent the correct location of the restriction. This is possible in CityGML but as mentioned before, there is no enforcement that the geometry of the segments aligns with the location of the street signs. Thus, the correct location of the restriction cannot be determined reliably. Either the restriction starts and stops earlier if the TrafficSpace segment next to the street sign with the restriction is chosen, or all restrictions begin at the following segment. Both versions are not suitable for accurate representation, however. Additionally, the current approach relies on the presence of a physical location to switch between transportation modes. While this makes sense in the context of parking spots, modelling the bicycle and pedestrian switch is difficult to accomplish, as in principle a switch can take place at every adjacent TrafficSpace segment. As traditional navigation applications do not consider this, there also is no good comparison. Nevertheless, the bicycle transportation mode faces another issue. At the moment, only bicycle lanes are used for the transportation mode. In other applications, it is possible to use driving lanes if no separate bike lane is available. Replicating this in the current network design again faces the problem that the TrafficSpace objects only have one transportation type associated and would require additional information on where bicycles can use the driving subnetwork. Furthermore, the shortest path algorithms must be used differently to ensure that no transportation mode change occurs once the bicycle uses the driving lane. At the moment, the faster connection would be prioritized for example. However, the distinguishing features of the developed approach are the usage of the semantic information provided by CityGML and the combined usage of different transportation modes as well as routing using different granularities and geometric dimensions. This is not possible in other approaches and enables a wide range of applications for future transport applications that benefit from additional semantic information, like autonomous driving, heavy goods transport, or handicapped accessible routing. Furthermore, the usage of different granularities and geometric dimensions allows the usage of different datasets, e.g., a volumetric representation of a parking garage or tunnel and a lane-based representation of the road network.

6. Conclusion

The main findings of the thesis are that basic routing is possible using the open OGC standard CityGML 3.0. Furthermore, the semantic and feature-rich data model is a good candidate for multimodal routing. However, the standard lacks some functionality needed for advanced routing. In this thesis, a concept for multimodal routing using CityGML 3.0 data was developed. Additionally, a prototype for multimodal routing based on a graph representation of CityGML 3.0 data using the graph database Neo4j and Python was implemented. The findings of the thesis also contributed to improvements in the r:trån software, which is used to convert OpenDRIVE data to CityGML 3.0. The software was updated to include lane changes, connections of CityFurniture and TrafficSpace objects, and the correct linking direction of successor and predecessor links based on the traversal direction rather than the reference line used in OpenDRIVE.

The prototype implementation showed that multimodal routing using CityGML 3.0 data is possible. However, the prototype implementation also showed that the routing results are not always optimal when compared to expected results in the real world as the available network connections are incomplete. Besides improved input data, the routing results could be improved by using a more sophisticated routing engine. Furthermore, the implementation showed that the routing results differ based on the chosen weights. Adding support for more routing modes, e.g., public transport, weights and restrictions, e.g., maximum inclination, road type, etc., could improve the usefulness of the application. To ensure transferability, needed information, such as weights and restrictions, should be stored in a standardized way in CityGML. Thus, an ADE could be developed to allow storing and exchange of this data. Based on the availability, the additional information could be automatically included in the routing process. As the data model of CityGML 3.0 does not enforce the inclusion of predecessor and successor connections for TrafficSpace objects, the development of an automated process to retrieve these connections will be beneficial. As the successor XLinks build the core structure of the routing network they must be present and accurate to get realistic routing results. Lastly, the integration of other data sources, e.g., OSM or real-time traffic information, could further improve the overall routing results and usefulness.

List of Figures

List of Tables

Acronyms

ADE Application Domain Extension. 2

API Application Programming Interface. 9

APOC Awesome Procedures on Cypher. 4, 27, 65, 66, 74, 85, 86, 99, 102, 111, 113

ASB "Anweisung Straßeninformationsbank". 23

CityGML City Geography Markup Language. 1

GDF Geographic Data Files. 10, 15

GML Geography Markup Language. 18

GNSS Global Navigation Satellite System. 7

GUI Graphical User Interface. 87, 88, 90, 109, 113, 120

ISO International Organization for Standardization. 10, 22, 23

ITS Intelligent Transportation System. 5

OGC Open Geospatial Consortium. 1, 10, 22, 117

OKSTRA 'Objektkatalog für das Straßen- und Verkehrswesen'. 23

OSM OpenStreetMap. 23

P+R Park and Ride. 8, 36

SQL Structured Query Language. 3, 26

XML Extensible Markup Language. 5, 10, 18, 20, 23

Glossary

ADE (Application Domain Extension) is an extension to the CityGML data model that adds additional semantic information to the model (Kolbe et al., 2023). 2, 5, 110, 112, 117

API (Application Programming Interface) is an interface that allows communication between two software applications (Reddy, 2011). 9

BLOS (Bicycle Level of Service) indicators are used to provide objective ratings of the bicycle suitability (or quality) of links or intersections in transport networks (Pritchard et al., 2019). 36, 42, 50, 63

CityGML (City Geography Markup Language) is a common semantic information model for the representation of 3D urban objects that can be shared over different applications (Kolbe et al., 2023). 1, 2, 3, 4, 5, 10, 13, 14, 15, 18, 20, 21, 22, 39, 41, 46, 49, 51, 52, 53, 57, 61, 62, 63, 64, 66, 69, 70, 71, 72, 73, 74, 76, 77, 87, 88, 102, 103, 108, 110, 112, 114, 115, 117, 119, 120, 123, 127

Cypher is a declarative graph query language developed by Neo Technology, Inc. for the Neo4j graph database. 3, 4, 26, 28, 29, 30, 52, 53, 54, 55, 64, 66, 74, 76, 79, 80, 85, 86, 92, 112

edge An edge is a connection between two nodes in a graph. It can have properties that represent the cost of traversing the edge. In the context of this thesis, it is used interchangeably with the term link, relationship, connection, or line, when referring to network or graph structures. 5, 24, 25, 30, 39, 41, 43, 46, 47, 50, 53, 110, 128

graph A graph is the mathematical term for a network structure that consists of nodes and edges. It is used interchangeably with the term network in the context of this thesis. 1, 3, 5, 15, 18, 20, 21, 24, 25, 26, 28, 29, 30, 31, 39, 41, 43, 46, 50, 52, 54, 59, 63, 64, 65, 68, 76, 77, 79, 80, 114, 117, 123

k-d tree A k-d tree is a data structure that is used to efficiently find the nearest neighbour of a given point in a set of points. It is a binary tree that splits the points into two sets at each level. The splitting is done by a hyperplane that is perpendicular to one of the axes of the coordinate system. The splitting axis alternates at each level. The kd-tree is a generalization of the binary search tree to higher dimensions (Bentley, 1975). 5, 27, 28, 65, 87, 88, 113

Neo4j is a graph database management system developed by Neo Technology, Inc. 2, 3, 4, 18, 21, 24, 25, 26, 27, 30, 50, 52, 55, 64, 65, 80, 82, 83, 84, 85, 86, 87, 91, 92, 101, 109, 111, 117, 120, 127, 128

network A network is a graph structure that consists of nodes and edges. It is used interchangeably with the term graph in the context of this thesis. 3, 4, 5, 7, 8, 9, 10, 15, 22, 23, 32, 35, 36, 37, 39, 43, 50, 51, 52, 53, 54, 58, 64, 65, 66, 67, 68, 69, 73, 76, 82, 92, 110, 111, 112, 113, 114, 115

node A node is a single element in a graph. It can have properties and connections to other nodes. In the context of this thesis, it is used interchangeably with the term vertex and point, when referring to network or graph structures. Node is the term used by Neo4j to refer to the point elements in a graph database. 3, 4, 5, 18, 20, 23, 25, 26, 27, 28, 29, 30, 31, 32, 36, 37, 39, 43, 46, 52, 53, 54, 55, 56, 58, 59, 63, 64, 65, 66, 67, 76, 77, 78, 79, 80, 82, 85, 86, 87, 91, 92, 99, 102, 109, 110, 111, 128, 129

OpenDRIVE is a file format for the exchange of road networks. It is maintained by ASAM e.V. (Association for Standardization of Automation and Measuring Systems) (ASAM e.V., 2023). 21, 22, 46, 52, 73, 117, 119

OSM (OpenStreetMap) is a publicly available mapping service. The data is collected by a large number of mappers who collect and maintain data on all kinds of geolocated things worldwide. This includes for example streets and buildings and even individual trees. 23, 46, 52, 117

point see node in the context of graph or network structures. 5, 23, 27, 28, 29, 35, 36, 37, 43, 44, 45, 50, 51, 63, 65, 68, 73, 77, 78, 79, 87, 88, 92, 94, 99

relationship see edge in the context of graph or network structures. It is the term used by Neo4j to refer to the connections between nodes in a graph database. 1, 3, 5, 14, 18, 20, 21, 22, 23, 25, 26, 28, 29, 39, 40, 41, 50, 52, 53, 54, 55, 56, 58, 63, 64, 65, 66, 67, 74, 76, 77, 78, 79, 80, 82, 83, 85, 86, 91, 92, 109, 112, 114, 120

shortest path algorithm A shortest path algorithm is an algorithm that finds the shortest path between two nodes in a graph. The shortest path is the path with the lowest cost. The cost can be distance, time, or any other metric that is used to determine the best path through the network. 3, 5, 27, 37, 39, 41, 61, 69, 74, 77, 78, 91, 93, 99, 111, 113

vertex see node. 5, 24, 30

Bibliography

ASAM e.V. (2023, June 23). *ASAM OpenDRIVE®*. Retrieved June 23, 2023, from https://www.asam.net/standards/detail/opendrive/

Beil, C., Kutzner, T., Schwab, B., & Kolbe, T. H. (2022). *Road2CityGML3*. manual. https://tum-gis.github.io/road2citygml3/

Beil, C., Ruhdorfer, R., Coduro, T., & Kolbe, T. H. (2020). Detailed Streetspace Modelling for Multiple Applications: Discussions on the Proposed CityGML 3.0 Transportation Model. *ISPRS International Journal of Geo-Information*, *9*(10), 603. https://doi.org/10.3390/ijgi9100603

Bentley, J. L. (1975). Multidimensional Binary Search Trees Used for Associative Searching. *Communications of the ACM*, *18*(9), 509–517. https://doi.org/10.1145/361002.361007

Bettencourt, R., & Lima, P. U. (2021). Multimodal Navigation for Autonomous Service Robots. *2021 IEEE International Conference on Autonomous Robot Systems and Competitions (ICARSC)*, 25–30. https://doi.org/10.1109/ICARSC52212.2021.9429771

Bundesanstalt für Straßenwesen. (2023). *OKSTRA®*. Retrieved June 29, 2023, from https://www.okstra.de/

Bundesministerium für Digitales und Verkehr. (2023, June 29). *BMDV - BIM – Grundlagen und Vorarbeiten*. BIM – Grundlagen und Vorarbeiten. Retrieved June 29, 2023, from https://bmdv.bund.de/SharedDocs/DE/Artikel/DG/bim-grundlagen-und-vorarbeiten.html

Cambridge University Press & Assessment. (2023a, June 22). *Navigation*. Retrieved June 22, 2023, from https://dictionary.cambridge.org/dictionary/english/navigation

Cambridge University Press & Assessment. (2023b, June 21). *Routing*. Retrieved June 22, 2023, from https://dictionary.cambridge.org/dictionary/english/routing

Cesium GS, Inc. (2023). *Cesium: The Platform for 3D Geospatial*. Cesium. Retrieved November 2, 2023, from https://cesium.com/

Chris Knott, & Samuel Williams. (2023, December 28). *Eel*. Retrieved December 28, 2023, from https://github.com/python-eel/Eel

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009, August). *Introduction to Algorithms, Third Edition* (3rd). The MIT Press. https://edutechlearners.com/download/Introduction_to_algorithms-3rd%20Edition.pdf

Datta, S. (2020, July 14). *Floyd-Warshall Algorithm: Shortest Path Finding | Baeldung on Computer Science*. Retrieved August 28, 2023, from https://www.baeldung.com/cs/floyd-warshall-shortest-path

Dechter, R., & Pearl, J. (1985). Generalized Best-first Search Strategies and the Optimality of A*. *Journal of the ACM*, *32*(3), 505–536. https://doi.org/10.1145/3828.3830

Delling, D., Sanders, P., Schultes, D., & Wagner, D. (2009). Engineering Route Planning Algorithms. In J. Lerner, D. Wagner, & K. A. Zweig (Eds.), *Algorithmics of Large and Complex Networks: Design, Analysis, and Simulation* (pp. 117–139). Springer. https://doi.org/10.1007/978-3-642-02094-0_7

Dib, O., Manier, M.-A., Moalic, L., & Caminada, A. (2017). A Multimodal Transport Network Model and Efficient Algorithms for Building Advanced Traveler Information Systems. *"19th EURO Working Group on Transportation Meeting, EWGT2016, 5-7 September 2016, Istanbul, Turkey"*, *22*, 134–143. https://doi.org/10.1016/j.trpro.2017.03.020

Dijkstra, E. W. (1959). A Note on Two Problems in Connexion with Graphs. *Numerische Mathematik*, *1*(1), 269–271. https://doi.org/10.1007/BF01386390

Don Ho. (2023). *Notepad++*. Retrieved October 31, 2023, from https://notepad-plus-plus.org/

Engibaryan, R. (2023, March 20). *All-Pairs Shortest Paths: Johnson's Algorithm | Baeldung on Computer Science*. Retrieved August 28, 2023, from https://www.baeldung.com/cs/all-pairs-shortest-paths-johnsons-algorithm

Environmental Systems Research Institute, Inc. (2023). *2D, 3D & 4D GIS Mapping Software | ArcGIS Pro*. Retrieved October 31, 2023, from https://www.esri.com/en-us/arcgis/products/arcgis-pro/overview

European Commission. (2023). *INSPIRE Geoportal*. Retrieved December 28, 2023, from https://inspire-geoportal.ec.europa.eu/srv/eng/catalog.search#/home

European Union. (2023). *INSPIRE*. Retrieved June 29, 2023, from https://inspire.ec.europa.eu/

Friedman, J. H., Bentley, J. L., & Finkel, R. A. (1977). An Algorithm for Finding Best Matches in Logarithmic Expected Time. *ACM Transactions on Mathematical Software*, *3*(3), 209–226. https://doi.org/10.1145/355744.355745

Google Cloud EMEA Limited. (2023, October 30). *Routes API-Übersicht*. Google for Developers. Retrieved June 20, 2023, from https://developers.google.com/maps/documentation/routes/overview?hl=de

Google Maps. (2023, January 26). Google Maps. Retrieved October 20, 2023, from https://www.google.com/maps

Hart, P. E., Nilsson, N. J., & Raphael, B. (1968). A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on Systems Science and Cybernetics*, *4*(2), 100–107. https://doi.org/10.1109/TSSC.1968.300136

HERE Global B.V. (2023a, October 20). *HERE WeGo*. HERE WeGo. Retrieved January 26, 2023, from https://wego.here.com/

HERE Global B.V. (2023b, October 20). *Routing API v8*. Routing API v8. Retrieved October 20, 2023, from https://www.here.com/docs/bundle/routing-api-v8-api-reference/page/index.html

HERE Global B.V. (2023c, October 20). *Routing APIs from HERE: Intelligent Algorithms for Efficient Journeys | HERE*. Retrieved October 20, 2023, from https://developer.here.com/products/routing

Hofmann-Wellenhof, B., Legat, K., & Wieser, M. (2011, June 28). *Navigation: Principles of Positioning and Guidance*. Springer Science & Business Media. https://books.google.de/books?id=dMXcBQAAQBAJ&printsec=frontcover&hl=de#v=onepage&q&f=false

Holly, L., & AFP. (2023). 49-Euro-Ticket: Bundestag beschließt Gesetz zur Einführung des Deutschlandtickets [newspaper]. *Die Zeit*. Retrieved June 22, 2023, from https://www.zeit.de/mobilitaet/2023-03/bundestag-beschliesst-gesetz-zur-einfuehrung-des-deutschlandtickets?utm_referrer=https%3A%2F%2Fduckduckgo.com%2F

International Organization for Standardization. (2020a, March). *Intelligent Transport Systems — Geographic Data Files (GDF) GDF5.1 — Part 1: Application Independent Map Data Shared Between Multiple Sources (ISO Standard No. 20524-1:2020)*. Retrieved March 16, 2023, from https://www.iso.org/standard/68244.html

International Organization for Standardization. (2020b, November). *Intelligent Transport Systems — Geographic Data Files (GDF) GDF5.1 — Part 2: Map Data Used in Automated Driving Systems, Cooperative ITS, and Multi-modal Transport (ISO Standard No. 20524-2:2020)*. ISO. Retrieved March 16, 2023, from https://www.iso.org/standard/72494.html

International Organization for Standardization. (2019). *ISO 19107:2019*. Retrieved June 29, 2023, from https://www.iso.org/standard/66175.html

International Organization for Standardization. (2015). *ISO 19109:2015*. Retrieved June 29, 2023, from https://www.iso.org/standard/59193.html

Kolbe, T. H., Kutzner, T., Smyth, C. S., & Roensdorf, C. (2023, January 23). *CityGML | OGC*. Retrieved January 23, 2023, from https://www.ogc.org/standards/citygml

Kuriakose, B., Shrestha, R., & Sandnes, F. (2020). Multimodal Navigation Systems for Users with Visual Impairments—A Review and Analysis. *Multimodal Technologies and Interaction*, *4*, 73. https://doi.org/10.3390/mti4040073

Kurt Mehlhorn, & Peter Sanders. (2008). *Algorithms and Data Structures*. Springer. https://doi.org/10.1007/978-3-540-77978-0

Kutzner, T., Chaturvedi, K., & Kolbe, T. H. (2020). CityGML 3.0: New Functions Open Up New Applications. *PFG – Journal of Photogrammetry, Remote Sensing and Geoinformation Science*, *88*(1), 43–61. https://doi.org/10.1007/s41064-020-00095-z

Leaflet maintainers. (2023, November 6). *Leaflet* (Version 1.9.4). Retrieved November 6, 2023, from https://github.com/Leaflet/Leaflet

Levine, U. (2023, June 9). *Was Selling Waze To Google A Good Decision? Founder Of Waze Reflects On The Deal*. Forbes. Retrieved October 20, 2023, from https://www.forbes.com/sites/urilevine/2023/06/09/was-selling-waze-to-google-a-good-decision-founder-of-waze-reflects-on-the-deal/

Liu, L. (2011). Data Model and Algorithms for Multimodal Route Planning with Transportation Networks. https://www.researchgate.net/publication/49940710_Data_model_and_algorithms_for_multimodal_route_planning_with_transportation_networks

Louise Wylie. (2023, July 11). *Navigation App Revenue and Usage Statistics (2023)*. Business of Apps. Retrieved October 31, 2023, from https://www.businessofapps.com/data/navigation-app-market/

Melanie Herzog, Wolfgang F. Riedl, & Richard Stotz. (2013). *Shortest Paths*. Retrieved July 20, 2023, from https://algorithms.discrete.ma.tum.de/spp/

Neo4j, Inc. (2023a). *Awesome Procedures On Cypher (APOC) - Neo4j Labs*. Retrieved December 28, 2023, from https://neo4j.com/labs/apoc/

Neo4j, Inc. (2023b). *Awesome Procedures On Cypher (APOC) - Neo4j Labs*. Neo4j Graph Data Platform. Retrieved August 24, 2023, from https://neo4j.com/labs/apoc/

Neo4j, Inc. (2023c). *Concepts*. Neo4j Graph Data Platform. Retrieved September 8, 2023, from https://neo4j.com/docs/cypher-manual/5/patterns/concepts/

Neo4j, Inc. (2023d, July 3). *Cypher Query Language - Developer Guides*. Neo4j Graph Data Platform. Retrieved July 3, 2023, from https://neo4j.com/developer/cypher/

Neo4j, Inc. (2023e). *Graph Database Concepts*. Neo4j Graph Data Platform. Retrieved August 24, 2023, from https://neo4j.com/docs/getting-started/appendix/graphdb-concepts/

Neo4j, Inc. (2023f). *Modeling Designs*. Neo4j Graph Data Platform. Retrieved August 24, 2023, from https://neo4j.com/docs/getting-started/data-modeling/modeling-designs/

Neo4j, Inc. (2023g). *Modeling: Relational to Graph*. Neo4j Graph Data Platform. Retrieved August 24, 2023, from https://neo4j.com/docs/getting-started/data-modeling/relational-to-graph-modeling/

Neo4j, Inc. (2023h). *Naming Rules and Recommendations - Cypher Manual*. Neo4j Graph Data Platform. Retrieved October 20, 2023, from https://neo4j.com/docs/cypher-manual/5/syntax/naming/

Neo4j, Inc. (2023i). *Neo4j Graph Database & Analytics – The Leader in Graph Databases*. Graph Database & Analytics. Retrieved December 27, 2023, from https://neo4j.com/

Neo4j, Inc. (2023j). *openCypher · openCypher*. Retrieved August 24, 2023, from https://opencypher.org/

Neo4j, Inc. (2023k). *Path Finding Procedures - APOC Documentation*. Neo4j Graph Data Platform. Retrieved August 24, 2023, from https://neo4j.com/docs/apoc/5/algorithms/path-finding-procedures/

Neo4j, Inc. (2023l). *What Is A Graph Database?* Neo4j Graph Data Platform. Retrieved August 24, 2023, from https://neo4j.com/docs/getting-started/get-started-with-neo4j/graph-database/

Nguyen, S. H. (2017, May 15). *Spatio-semantic Comparison of 3D City Models in CityGML using a Graph Database* (Master's thesis). Technische Universität München, Department of Informatics. München. Retrieved November 7, 2022, from https://mediatum.ub.tum.de/doc/1374646/1374646.pdf

Nicolas Bonnefon. (2023). *Glogg — Glogg - the Fast, Smart Log Explorer*. Retrieved October 31, 2023, from https://glogg.bonnefon.org/

OpenStreetMap Foundation contributors. (2023a, June 29). *Elements – OpenStreetMap Wiki*. Retrieved June 29, 2023, from https://wiki.openstreetmap.org/wiki/Elements

OpenStreetMap Foundation contributors. (2023b, June 29). *Map Features – OpenStreetMap Wiki*. Retrieved June 29, 2023, from https://wiki.openstreetmap.org/wiki/Map_features

OpenStreetMap Foundation contributors. (2023c, June 29). *OSM File Formats – OpenStreetMap Wiki*. Retrieved June 29, 2023, from https://wiki.openstreetmap.org/wiki/OSM_file_formats

Pritchard, R., Frøyen, Y., & Snizek, B. (2019). Bicycle Level of Service for Route Choice—A GIS Evaluation of Four Existing Indicators with Empirical Data. *ISPRS International Journal of Geo-Information*, *8*(5), 214. https://doi.org/10.3390/ijgi8050214

Reddy, M. (2011, March 14). *API Design for C++*. Elsevier.

Reinhard Diestel. (2016). *Graphentheorie* (5th ed.). https://diestel-graph-theory.com/basic.html

Ruhdorfer, R. (2017). Kopplung von Verkehrssimulation und semantischen 3D-Stadtmodellen in CityGML. https://mediatum.ub.tum.de/doc/1396796/1396796.pdf

Safe Software Inc. (2023). *Products*. Safe Software. Retrieved October 31, 2023, from https://www.safe.com/products/

Schwab, Benedikt, Beil, Christof, & Kolbe, Thomas H. (2023). R:trån. *Zenodo*. https://doi.org/10.5281/zenodo.7702313

Simic, M. (2021, October 24). *Dijkstra vs. A* – Pathfinding | Baeldung on Computer Science*. Baeldung. Retrieved July 20, 2023, from https://www.baeldung.com/cs/dijkstra-vs-a-pathfinding

Smith, R. (2015). Directive 2010/41/EU of the European Parliament and of the Council of 7 July 2010. *Core EU Legislation* (pp. 352–355). Macmillan Education UK. https://doi.org/10.1007/978-1-137-54482-7_33

Sryheni, S. (2020, July 27). *Dijkstra's vs Bellman-Ford Algorithm | Baeldung on Computer Science*. Retrieved August 28, 2023, from https://www.baeldung.com/cs/dijkstra-vs-bellman-ford

Statista. (2023). *Most Popular Navigation Apps in the U.S. 2022*. Statista. Retrieved October 31, 2023, from https://www.statista.com/statistics/865413/most-popular-us-mapping-apps-ranked-by-audience/

Sven Oliver Krumke, & Hartmut Noltemeier. (2012, June 13). *Graphentheoretische Konzepte und Algorithmen* (3rd ed.). Vieweg+Teubner Verlag Wiesbaden. https://doi.org/10.1007/978-3-8348-2264-2

Team Counterpoint. (2022, January 11). *HERE Maintains the Location Platform Leadership, Ahead of Google, and TomTom in 2021 - Counterpoint*. Retrieved October 31, 2023, from https://www.counterpointresearch.com/insights/maintains-location-platform-leadership-ahead-google-tomtom/

United Nations. (1980, May 24). 1. United Nations Convention on International Multimodal Transport of Goods. https://treaties.un.org/doc/Treaties/1980/05/19800524%2006-13%20PM/Ch_XI_E_1.pdf

Walter, O., Schmalenstroeer, J., Engler, A., & Haeb-Umbach, R. (2013). Smartphone-based Sensor Fusion for Improved Vehicular Navigation, 1–6. https://doi.org/10.1109/WPNC.2013.6533261

Waze Mobile Ltd. (2023a, October 20). *Routenanweisungen, Echtzeit-Informationen zu Verkehr und Straßenverhältnissen*. Waze. Retrieved September 28, 2022, from https://www.waze.com/de/live-map/

Waze Mobile Ltd. (2023b). *Routing Server - Wazeopedia*. Retrieved October 31, 2023, from https://www.waze.com//wiki/USA/Routing_server?rdfrom=https%3A%2F%2Fwww.waze.com%2Fwiki%2FCommunityHub%2Findex.php%3Ftitle%3DRouting_server%26redirect%3Dno

Waze Mobile Ltd. (2023c, October 20). *Waze Map Editor*. Retrieved October 20, 2023, from https://www.waze.com/en-US/editor

Waze Mobile Ltd. (2023d, October 20). *Wie Waze Routen kalkuliert – Wazeopedia*. Retrieved September 28, 2022, from https://wazeopedia.waze.com/wiki/Germany/Wie_Waze_Routen_kalkuliert

Wisetruangrot, S. (2020). Multimodal Transportation Concept and Framework.

Xiong, G., & Wang, Y. (2014). Best Routes Selection in Multimodal Networks Using Multi-objective Genetic Algorithm. *Journal of Combinatorial Optimization*, *28*(3), 655–673. https://doi.org/10.1007/s10878-012-9574-8

Zeng, W., & Church, R. L. (2009). Finding Shortest Paths on Real Road Networks: The Case for A*. *International Journal of Geographical Information Science*, *23*(4), 531–543. https://doi.org/10.1080/13658810801949850

Zhang, J., Liao, F., Arentze, T., & Timmermans, H. (2011). A Multimodal Transport Network Model for Advanced Traveler Information Systems. *Procedia - Social and Behavioral Sciences*, *20*, 313–322. https://doi.org/10.1016/j.sbspro.2011.08.037

Zhou, Q.-Y., Park, J., & Koltun, V. (2018). *Open3D: A Modern Library for 3D Data Processing*. http://www.open3d.org/

A. Appendix I - Routing Runtime Measurements

route name	average runtime	standard deviation	meas. 1	meas. 2	meas. 3	meas. 4	meas. 5	meas. 6	meas. 7	meas. 8	meas. 9	meas. 10
route 1: advanced length	34.7	4.84	42	35	33	31	28	29	37	40	41	31
route 1: inclination	33.0	4.98	34	32	28	32	33	26	28	37	44	36
route 1: width	23.6	3.32	23	22	22	26	26	21	22	21	32	21
route 1: time	29.6	5.2	31	26	26	32	27	28	27	26	44	29
route 2: advanced length	17.4	2.62	16	16	16	18	17	16	16	17	25	17
route 2: inclination	13.1	2.77	13	12	11	14	11	12	13	12	21	12
route 2: width	24.4	2.65	24	24	21	27	26	21	25	22	30	24
route 2: time	15.1	3.42	14	12	13	14	15	15	14	25	15	14
route 3: advanced length	24.0	4.07	20	29	25	29	23	28	19	20	28	19
route 3: inclination	24.7	4.8	26	26	23	24	22	21	24	20	38	23
route 3: width	21.6	3.47	20	21	18	20	20	19	20	26	30	22
route 3: time	18.5	4.01	20	17	16	17	18	16	17	30	16	18
route 4: advanced length	8.0	3.03	7	7	7	8	6	7	7	7	17	7
route 4: inclination	10.3	4.29	9	10	8	9	8	8	10	9	23	9
route 4: width	5.4	2.91	5	4	5	5	4	4	4	5	14	4
route 4: time	7.3	2.33	7	7	6	6	7	6	6	6	14	8
route 1: advanced length A*	31.5	11.27	64	31	27	27	26	27	26	25	36	26
route 1: inclination A*	25.0	2.37	25	31	24	22	26	26	23	25	25	23
route 1: width A*	16.6	2.37	15	15	21	21	14	15	16	15	17	17
route 1: time A*	23.1	1.7	26	22	23	24	25	20	23	23	24	21
route 2: advanced length A*	7.6	0.92	8	7	7	7	7	7	8	10	7	8
route 2: inclination A*	7.9	1.14	9	7	9	8	10	8	7	8	6	7
route 2: width A*	12.7	1.0	14	14	11	12	13	13	13	11	13	13
route 2: time A*	6.8	0.98	7	5	6	7	7	6	7	7	9	7
route 3: advanced length A*	22.7	2.69	29	23	21	21	20	21	25	25	21	21
route 3: inclination A*	21.5	3.83	18	25	19	31	18	22	19	20	20	23
route 3: width A*	14.3	2.24	14	13	12	12	20	15	14	14	13	16
route 3: time A*	25.0	3.1	25	27	21	22	21	26	28	22	29	29
route 4: advanced length A*	5.2	0.87	7	4	6	5	5	4	5	5	6	5
route 4: inclination A*	4.7	1.0	4	6	4	5	6	4	4	3	6	5
route 4: width A*	3.4	0.8	4	3	3	3	3	2	3	5	4	4
route 4: time A*	4.8	1.17	4	4	4	5	4	7	4	5	7	4

Table A.1.: Runtime Measurements in Milliseconds

B. Appendix II - Python Libraries

The following list only contains the libraries used in the Neo4j Navigator project. This includes the Neo4jPreProcessor and the Neo4jNavigator classes in the interactor4neo4j file as well as the pre-processing, GUI and testing scripts. A list of all libraries used in the project can be found in the conda_package_requirements.yaml file in the root directory of the project which is part of the digital submission.

Name	**Version**
eel	0.16.0
geopy	2.3.0
json	
neo4j	5.3.0
numpy	1.25.0
open3d	0.17.0
pprint	
random	
re	
tkinter	
tqdm	4.65.0
utm	0.7.0

Table B.1.: Used Python Libraries

C. Appendix III - Pre-processing Runtime Measurements

Table C.1 shows the runtime measurements of the pre-processing for the full Grafing dataset as well as the runtimes for the Grafing garage dataset. The Grafing garage pre-processing does not include any weight calculations or the addition of relationship weight properties.

Test run #	**Grafing Runtime [s]**	**Grafing with garage Runtime [s]**
1	864.67	270.55
2	859.30	259.20
3	882.72	258.65
4	898.26	262.51
5	860.21	270.97
6	868.87	258.90
7	864.75	260.60
8	916.49	259.83
9	882.18	267.03
10	903.90	265.39

Table C.1.: Runtime comparison of the pre-processing for the Grafing dataset and the Grafing garage dataset without weights. Measurements in seconds.