Chair of Logistics

Master Thesis

Development of a tool to enhance freight transport efficiency through multimodal route optimization in Germany

in partial fulfillment of the requirements for the degree of
Master of Science (M.Sc.)

Technische Universität Berlin
Faculty VII – Economics and Management
Institute of Technology and Management
Chair of Logistics

submitted by Maria Serveto Font
0518344
Berlin, September 2025

Supervisor Prof. Dr. -Ing. Frank Straube
Kil-Young Lee, M.Sc.

Display

☒ „Hiermit erkläre ich mich damit einverstanden, dass meine Studien-, Bachelor- bzw. Masterarbeit in der Instituts- und Universitätsbibliothek ausgelegt und zur Einsichtnahme aufbewahrt werden darf.
"I hereby agree that my study, bachelor or master thesis may be displayed and kept for inspection in the institute and/or the university Library."

☐ „Hiermit erkläre ich, dass ich nicht mit der Auslage und Einsichtnahme meiner Studien-, Bachelor- bzw. Masterarbeit in der Instituts- und/oder der Universitätsbibliothek einverstanden bin."
"I hereby declare that I do not agree to the display and inspection of my study, bachelor or master thesis in the institute library and/or the university library."

Ich habe die Auswahl zur Auslage meiner Studien-, Bachelor- bzw. Masterarbeit in der Instituts- und/oder der Universitätsbibliothek zur Kenntnis genommen:
I have taken note of the selection for the display of my study, bachelor or master thesis in the institute and/or university library:

Berlin, 30 September 2025

Maria Serveto Font

Affidavit and AI disclaimer

I hereby declare on oath that I have completed this work independently and on my own, without unauthorised assistance and exclusively using the sources and aids listed.

Artificial intelligence (AI) technologies were used solely to support limited aspects of the research. Their use was restricted to grammar correction of the text, improving the clarity and structure of the developed code, assisting in the development of certain functions within the scripts used for visualising simulation results, and the storage of information. All outputs generated with AI assistance were subsequently reviewed and verified by the author to ensure that they perform as intended.

Berlin, 30 September 2025

Maria Serveto Font

Abstract

This work develops a tool to optimise freight transport routes within Germany by integrating road and rail networks into a multimodal framework. Using open-access data from OpenStreetMap, the project constructs a weighted graph in which each edge is assigned a distance, a transport cost, a travel time, and an emission factor, depending on the network to which it belongs. Dijkstra's algorithm is implemented to determine the optimal path between selected origin–destination pairs under different criteria: cost, time, and environmental impact.

The methodology includes data acquisition, network preprocessing, and the design of multimodal transfer points. Validation tests demonstrate that the tool is capable of generating feasible multimodal routes. Performance analyses highlight a clear tendency towards rail-based solutions in cost- and emissions-optimisation scenarios, whereas time minimisation tends to produce road-based itineraries. A sensitivity analysis of rail fixed costs further illustrates the adaptability of the model to different economic contexts.

The results confirm the potential of graph-based optimisation tools to support logistics decision-making. Although limitations arise from OpenStreetMap data quality and simplifying assumptions regarding operations, the tool provides a replicable framework for multimodal route optimisation. Future research could extend the approach by incorporating international networks and accounting for real-world constraints such as timetables and capacity restrictions.

Contents

List of Figures i

List of Tables ii

1 Introduction 1
- 1.1 Motivation and problem context . . . 1
- 1.2 Research Goal . . . 2
- 1.3 Objective and scope statement . . . 2
- 1.4 Outline of Work . . . 3

2 Theoretical background and literature review 4
- 2.1 Literature review on route optimization models . . . 4
- 2.2 Graph-based representation of networks . . . 5
- 2.3 Database: OpenStreetMap (OSM) . . . 7
  - 2.3.1 Tagging system in OSM . . . 8
- 2.4 Basic explanation on routing algorithms . . . 10
  - 2.4.1 Dijkstra's algorithm . . . 10

3 Methodology 12
- 3.1 Network data acquisition for route optimisation . . . 12
  - 3.1.1 OSM road network data . . . 12
  - 3.1.2 OSM rail network data . . . 14
  - 3.1.3 Points of interest (POIs) . . . 16
- 3.2 Algorithm design and graph construction . . . 18
  - 3.2.1 Criteria definition . . . 18
    - 3.2.1.1 Distance optimisation graph . . . 19
    - 3.2.1.2 Cost optimization graph . . . 20
    - 3.2.1.3 Time optimization graph . . . 22
    - 3.2.1.4 Emissions optimization graph . . . 22
  - 3.2.2 Graph construction: assembly of the final multimodal network . . . 24
    - 3.2.2.1 Implementation of Dijkstra's algorithm for multimodal transport routes optimization . . . 26

4 Algorithm validation and performance analysis 28
- 4.1 Algorithm validation . . . 28
- 4.2 Performance analysis under different routing criteria . . . 33
- 4.3 Sensitivity analysis on rail costs . . . 41

5 Conclusions and future research 46
- 5.1 Conclusions . . . 46
- 5.2 Suggestions for further research . . . 46

References I

A Konecranes SMV 4123 CC5 Datasheet **V**

B OSM road network extraction code: `overpass_map_road.py` **VII**

C OSM rail network extraction code: `overpass_map_rail.py` **X**

D OSM points of interest extraction code: `pois_extraction.py` **XIII**

E Weighted graph construction code: `weighted_graph.py` **XV**
E.1 Main script: `weighted_graph.py` XV
E.2 Auxiliar functions script: `functions_weighted_graph.py` XVII

F Routing algorithm code: `routing_algorithm.py` **XXII**
F.1 Main script: `routing_algorithm.py` XXII
F.2 Auxiliar functions script: `functions_routing_algorithm.py` XXIV

List of Figures

2.1 Graph examples with different β functions 6
2.2 Example of a multidigraph 7
2.3 Visual representation of OSM elements. Source: author's own work 8
2.4 Frankfurter Allee's OSM simplified representation 9

3.1 OSM road network 14
3.2 Rail network elements downloaded from OSM within Germany's borders . . . 15
3.3 Final POIs 18
3.4 Final network 26

4.1 Multimodal solution routes in validation tests 29
4.2 Non-multimodal solution routes in validation tests 30
4.3 Solution routes for validation tests considering transfer penalties 32
4.4 $Criteria^{i}_{BM}$ solution routes 36
4.5 $Criteria_{HF}$ solution route for every criterion 37
4.6 $Criteria_{HS}$ solution route for every criterion 37
4.7 $Criteria^{i}_{HM}$ solution routes 38
4.8 $Criteria^{i}_{MBir}$ solution routes 40
4.9 Cost optimisation routes between Berlin and Munich, with and without fixed costs 42
4.10 Cost optimisation routes between Hamburg and Munich, with and without fixed costs 42
4.11 Cost optimisation routes between Munich and Birkenfeld, with and without fixed costs 43
4.12 Route change between Berlin and Munich as a function of fixed costs 44
4.13 Route change between Munich and Birkenfeld as a function of fixed costs . . . 44

List of Tables

3.1 Road types for OSM road elements 13
3.2 Relevant attributes for nodes and edges in the road network 13
3.3 Key–value mapping for OSM railway features 15
3.4 Relevant attributes for nodes and edges in the railway network 16
3.5 Key–value mapping for OSM POIs 17
3.6 Counts of points of interest by label 17
3.7 Estimated times for loading and unloading transfer operations 20
3.8 Della's freight transport routes and rates (data from 13.08) 21
3.9 Cost expressions for each edge type in the network 22
3.10 Time weight expressions for each edge type in the network 22
3.11 Overview of CO_2 emissions, payload, mileage and shares per vehicle category 23
3.12 CO_2 emissions expressions for each edge type in the network 24
3.13 Model assumptions by network 24
3.14 Multimodal graph features 25

4.1 Algorithm validation tests summary 31
4.2 $Validation_{HM}$ test verifications 33
4.3 Route tags. 33
4.4 Criteria tests values summary 34
4.5 Criteria tests: overall features (by route) 34
4.6 Criteria test 5: route overview 39
4.7 Route costs under different rail fixed cost scenarios 45

1 Introduction

1.1 Motivation and problem context

The European Environment Agency, 2024 stated that the European Union's freight transport activity has significantly grown over the past decades, reaching a total of 3,471 billion tonne-kilometres in 2022. This growth tendency is expected to continue in the following years, according to the MIX-FF55 scenario established by the European Commission (Council of the European Union, 2024) — a policy roadmap that introduces different measures to reduce Europe's GHG emissions by 55% compared to 1990 levels by 2030 — since freight transport is expected to grow by 29.6% in volume compared to the 2015 scenario. This trend towards growth underlines the need to optimize freight transport systems and routes in order to make them as efficient and environmentally sustainable as possible.

The urgency to find feasible solutions is clear, since the transport sector in the EU accounted for 29% of total GHG emissions in 2022 (German Council of Economic Experts and French Council of Economic Analysis, 2025), with freight transport being responsible for 30% of the sector's CO_2 emissions (European Commission, 2023). This picture is due to the dominant use of road freight for land transport, a mode that relies on fossil fuels and contributes significantly to air pollution. According to Eurostat (as cited in German Council of Economic Experts & French Council of Economic Analysis (2025)), domestic road freight transport in the EU in 2022 accounted for 77.8% of total domestic freight traffic through the region. Because of that, new strategies such as multimodal transport and the use of electric vehicles should be enhanced to minimize the impact of the growth that the transportation sector will experience in the upcoming years. A clear example of this is the TEN-T policy (trans-European transportation network). This initiative is being enhanced by the European Union, and focuses on developing and executing efficient and high quality multimodal networks around its member states. This big network includes road, rail, sea shipping and inland waterway transportation, both for passengers and goods (European Commission, 2025). Given that Europe's extensive highway and railway networks support domestic flows within countries as well as international traffic across regions, it makes sense to take action on making these transportation systems as efficient and environmentally sustainable as possible, also to maintain its economic growth.

Given this context, it becomes essential for companies — particularly those operating in highly industrialised and logistics-intensive countries like Germany — to have access to intelligent tools that allow them to consult and determine the most efficient freight transport routes for their products. These tools should not only help identify the most cost-effective and time-efficient alternatives, but also prioritise transport options that minimise environmental impact. Given these objectives, multimodal transport can play a very important role in reducing greenhouse gas emissions within freight logistics, especially when it involves a shift from road to rail. As previously mentioned, road freight is significantly more polluting than rail: according to the European Environment Agengy (2023), transporting goods by truck emits on average five times more CO_2 per tonne-kilometre than by train. Freight transport through

the railway system enables the movement of larger volumes with less fuel and can result in a competitive mode of transport in many situations. Good management of production and supply chains can improve the environmental performance of companies without decreasing operational efficiency.

As a result of this, there is a growing interest in developing specialised tools that place multimodal transport at the core. Moreover, if such tools are designed around a limited set of transport modes, a more precise analysis of the available data can be executed, enabling more effective decision-making. Providing users with the ability to dynamically adjust their optimization criteria can also result in a more advantageous transport management setup — for instance, prioritising cost savings during budget cuts, or minimising environmental impact when aiming for sustainability certifications.

1.2 Research Goal

Considering the current challenges in the logistics sector and the growing interest in sustainable and efficient freight transport solutions, this thesis focuses on the development of a tool that can provide reliable information to make better-informed decisions regarding their transport routes. As discussed, multimodal transportation is a good option that can help reach these objectives, but merging different networks is a complex and difficult process that needs several aspects to be taken into account. In order to address this gap, the following research questions have been defined to guide the work carried out in this thesis.

Primary Research Question:
How can a multimodal freight transport route, considering road and rail transport modes, be determined and optimised, considering sustainability, cost-effectiveness and time-efficiency?

Secondary Research Questions:

1) *What specific parameters, related to sustainability, time efficiency and costs, are most relevant for optimising a multimodal freight transport model?*

2) *How can a tool be developed to provide the optimal modal route considering the mentioned parameters?*

1.3 Objective and scope statement

The objective of this thesis is to develop an optimisation tool for entities or individuals to use in the determination of the best freight transport route according to different criteria. In this process, the tool is to be designed to make decisions based on open-access data, to ensure it can be widely used by anyone interested.

This work will focus on freight transport routes within Germany, only considering two modes of transport: road and rail. This decision is made because the country's dimensions and its geographic characteristics make it unnecessary to consider other modes such as maritime or

air transport. The criteria to be considered for finding an optimal route will be cost, time, and CO_2 emissions to the atmosphere.

1.4 Outline of Work

This thesis is organised as follows. The current chapter frames the problem with an overview of the current logistics context and its motivation. Then it defines the project's objectives and scope. Chapter 2 presents the theoretical background on different optimisation approaches and reviews prior work on the problem. It includes a section on the theory of graph-based networks and another explaining the structure and information model of the database used in the project: OpenStreetMap. Chapter 3 explains the methodology followed at each stage of the thesis. It first describes the data acquisition process, and gives details on how the road and rail networks are extracted and how these data are cleaned, harmonised, and stored for later use in the tool. After that, the second section explains how the final multimodal network is transformed into a graph, as well as the reasoning behind each weight expression for each criterion. The third section describes the Python implementation and the code developed. Chapter 4 outlines the tool's validation process and explains two analyses conducted to study the algorithm's performance in a multimodal network. The first examines the tool's behaviour under separate optimisation criteria; the second is a sensitivity analysis of how the algorithm's final solution changes when introducing fixed costs in train transportation. Lastly, Chapter 5 synthesises the conclusions and insights of this thesis and offers an overview of possible future work. The Appendices provide additional information and code.

2 Theoretical background and literature review

Nowadays, there are different ways of representing transportation networks, and also various techniques are applied to determine the optimal route for freight transport in those grids. A large body of research focuses on mathematical optimisation procedures (models, algorithms) that aim to reproduce and solve complex real-world networks according to one or several criteria. These systems typically handle large amounts of data and often require considerable computation time to obtain a feasible solution. In the following section, a literature review on optimisation techniques will be presented. Graph representation of transportation networks will also be explained, and finally, a description of Dijkstra's algorithm (a mathematical optimisation algorithm) will be provided.

2.1 Literature review on route optimization models

In the last decades, the optimization of freight transport routes has become a topic of growing interest, both in research and in the logistics industry. Many resources have been allocated to the development of mathematical models that simulate the behavior of real-life transportation networks, as well as to other mathematical procedures — such as algorithms and heuristics — that aim to identify the optimal transportation route while reducing computational effort. Especially in recent years, some of these models have been adapted to multimodal transport, aiming to find the optimal way of moving goods using different transportation modes. However, when looking at the literature, there are still several common limitations that prevent these models from offering integrated and sustainable solutions. This chapter presents the main ideas from a set of papers published between 2000 and 2025, focusing on patterns that repeat across them and identifying the main gaps that remain to be solved.

A predominant tendency observed across the reviewed literature is the main focus on minimising the route's cost or time, rather than considering other relevant aspects such as the environmental impact of the selected route. Kaewfak et al. (2021) developed a multi-objective model based on Zero-One Goal Programming to reduce transport cost and transport time, also taking into account seven risk factors that could affect the transportation route. Bhattacharya et al. (2014) aimed to minimize the total supply chain costs and transportation times of freight transport in India, taking road traffic congestion as the main criterion for deciding whether to use a multimodal route or not. Pedersen (2005) proposed a different approach, considering a time-based network where arcs between nodes represent feasible time slots during which a train can travel between points. The objective function of this MIP model is also the minimization of fixed and variable costs, as well as the expenses related to transportation time. Other algorithm-based studies also prove this cost- or time-oriented optimization projects, such as Lusiani et al. (2021), that implements Dijkstra's algorithm to a weighted graph to find the shortest path according to distance, transportation costs or time efficiency.

On the other hand, some research studies do introduce an environmental perspective. Wang et al. (2020) included CO_2 emission costs as one of the elements in the objective function of a fuzzy MILP model aimed at minimizing overall costs. This was applied to improve freight transport routes in Vietnam while also addressing other logistics aspects such as node capacity, detours, and vehicle utilisation. Chen et al. (2024) tackles the problem of choosing optimal multimodal routes for cargoes with different value and time sensitivities. It proposes a model that minimizes the combined cost of carbon emissions and delivery time, adapting transport mode choices based on cargo attributes.

Other research projects try to focus on multimodal transportation, but they mainly lack the freight transportation approach (they are more passenger-oriented), or their main optimisation objectives don't place the environmental aspect as a priority. Barbosa et al. (2025), for example, adapts Dijkstra's algorithm to minimize the transportation carbon footprint, but it is focused on multimodal passenger networks, and not in goods transportation. On the other hand, Guo et al. (2024) studies multimodal freight transportation around modern cities (considering air rail and road modes), but focuses its study in finding a robust shortest path through a mixed-integer programming model.

Despite the fact that the reviewed studies take into account multimodal transport and emissions reduction, most of them are built for very specific cases or under fixed assumptions. This makes it difficult to apply their results to other scenarios or to use them as general tools for planning. Moreover, when more than one objective is considered, these are usually combined with predefined weights, which leaves little room for the user to decide what the most relevant matters are in each case. In real-life scenarios, priorities can change depending on the type of cargo, the distance, the available modes, or even the regulations in place. Because of this, there is still a need for a more generic model that can offer optimal routes based on different user-defined preferences. A tool like this should be able to adapt to different inputs, and provide route options that balance cost, time, and environmental impact according to the user's needs. This would make it easier to apply optimization models to real-life transport decisions, in a more flexible and useful way.

2.2 Graph-based representation of networks

When representing networks or geographic spaces, it is very common to use graph structures to capture their performance and flows of people or vehicles. In the following lines, an explanation on what a graph is, its structure and its features will be displayed. Unless otherwise noted, the definitions and the information provided follow Bender & Williamson (2010), with adaptations and paraphrasing to clarify some of the concepts, as well as a comparison to the terminology that will be used in this thesis to designate those elements in the represented networks.

A graph used for spatial network representation is a triple set G, composed of three different elements ($G = (V, E, \beta)$:

1) V, a finite set representing the vertices (also called nodes) of G.

2) E, a finite set representing the edges of G, which are connections established between different elements of the V set.

3) β, a function that, given an edge from set E, returns the two vertices in set V that it connects.

These different sets are mathematically described as follows:

$$V = \{A,B,C,D\}, \qquad E = \{a,b,c,d,e,f,g\}$$

$$\beta = \begin{pmatrix} a & b & c & d & e & f & g \\ \{A,B\} & \{A,B\} & \{A,C\} & \{B,C\} & \{B,C\} & \{B,C\} & \{B,D\} \end{pmatrix}$$

The β function is important, since it helps to specify that, on some occasions, there is more than one edge connecting the same two vertices. The fact that this duality exists implies that the features of these edges are different and can provide other advantages to the problem trying to be solved. Figure 2.1 shows a representation of a graph with the used terminology. The sets E and V are exactly the same, but the function β applied to the edges is a different one, resulting in a different edge display.

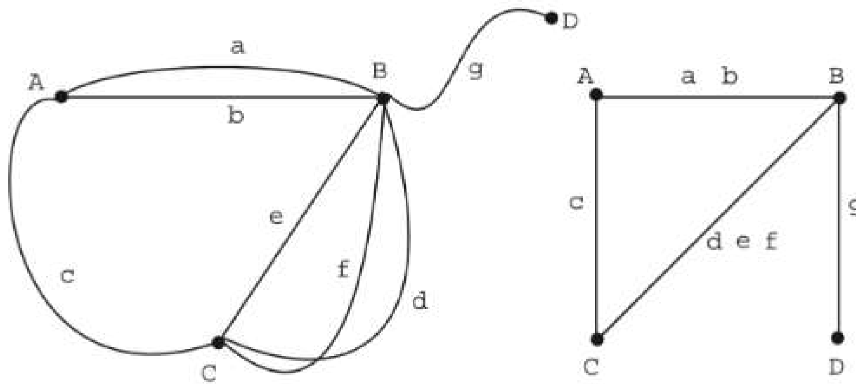


Fig. 2.1: Graph examples with different β functions. Reproduced from Bender & Williamson (2010, p. 149).

Once the structure of the graphs is clarified, an introduction to possible features is to be explained. Networks can be big and complex, and flows of vehicles or passengers can be difficult to represent if the graph does not have the most convenient display. The complexity of transportation networks is to be explained through an example. When the goal is to decide a route to go from city A to city B by road, there are multiple ways a driver can choose to perform the route, since those cities are connected by a motorway, but also by regional or secondary roads, that may imply a bigger distance but less traffic. Also, the existing roads can have lanes only in one direction, or both ways, to go from one point to another. Graphs can also have those features, and when they do, they are designated with a name that indicates so:

– **Directed graphs** (*digraphs*): are a pair of disjoint sets (E, V) that assign to every edge an initial vertex and a terminal vertex (Diestel, 2000, p. 25). This means the edges are

oriented and indicate how the information displayed through edges flows through the structure. In road networks, for example, this implies that if there is an edge connecting city A with city B (both vertices), there is a one-way road going from A to B, but not the other way around.

– **Multigraphs**: are also a pair of disjoint sets (E, V), where to every edge can be assigned one or more ends (vertices) (Diestel, 2000, p. 25).

These two features can be found in a graph at the same time. When this happens, the graph is described as a *multidigraph*. Figure 2.2 shows an example of this type of graphs.

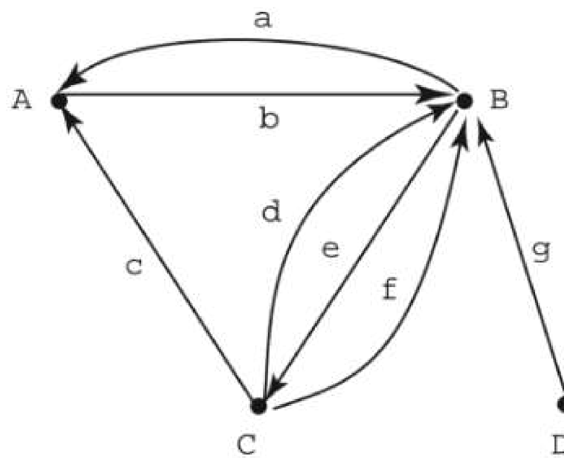


Fig. 2.2: Example of a multidigraph. Reproduced from Bender & Williamson (2010, p. 160).

In Fig. 2.2, vertex C has two edges (d, f) that connect it to vertex B (multigraph feature). Moreover, while vertex B is connected to C through edge e, there is no way to go from B to D, since edge g only connects those two points in the other direction (digraph feature). It is also important to clarify what a path in the graph context is. Given a graph $G = (V,E,\beta)$, a path is a feasible sequence of edges $e_1, e_2, ..., e_{n-1}$ that establish a non-disrupted connection with a sequence of vertices $v_1, v_2, ..., v_n$, such that $\beta(e_i) = \{v_i,v_{i+1}\}$.

2.3 Database: OpenStreetMap (OSM)

In the interest of creating a versatile and flexible tool that can be adapted to different needs and scenarios, it is necessary to acquire the data from an open source. OpenStreetMap (from now on, OSM) is an open map database that provides free data used in mapping applications (OpenStreetMap contributors, 2025). This information is collected by ground surveys, personal knowledge and government data, among other sources, and is regularly updated by volunteers.

The data provided is presented in different forms, which are called the elements of OSM. These elements are nodes, ways and relations, and their assembly makes up the digital representation of the physical world in OSM. The overall output is a very complex graph structure, that can be

displayed in different formats (see Sec. 2.2). All the information regarding the OpenStreetMap displayed in this section follows the OSM Wiki (OpenStreetMap Wiki contributors, 2025a,b), where all its documentation is explained.

Node
A node is a representation of a geographic point on the globe's surface. Every point is at least defined by its longitude and latitude coordinates, and an id number (*node id*). The database, as of August 2025, is composed of over 10 billion nodes. These elements are the core elements of the data structure, since they can define single features (like a fountain or a traffic sign), but also can be part of a shape (a building) or a path (a way).

Way
A way is an element composed of an ordered list of nodes, which define a polyline. It can be used to describe many of the features in OpenStreetMap:

- A *polyline* is used to define linear elements, such as roads, railways or rivers.
- A closed polyline can result in a *polygon*, which is used to represent areas of interest, such as a building or a station.

If an area has one or more "holes" (see Fig. 2.3) it cannot be represented with a single way, so it requires a *multipolygon* relation data structure.

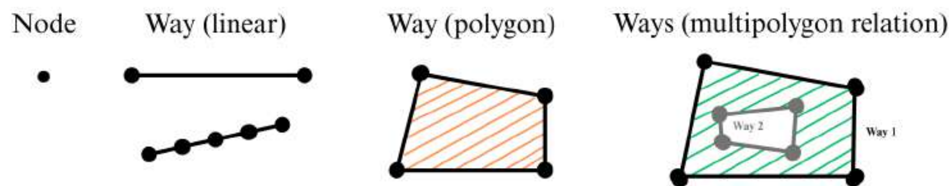


Fig. 2.3: Visual representation of OSM elements. Source: author's own work.

Relation
A relation establishes a relationship between two or more data elements in OSM. These elements can be nodes, ways or other relations. An example of this is the already mentioned *multipolygon*, which describes an area with holes, formed by two different closed ways. A route is another example of a relation, now between two or more ways that form a certain road (a highway, for example) or a hiking route.

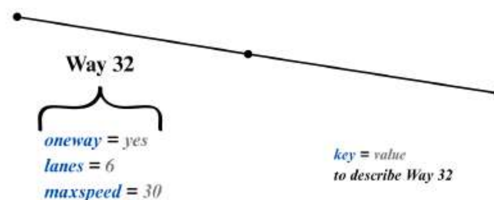
2.3.1 Tagging system in OSM

OSM uses tags to describe specific features of its elements, it works as a label that can be easily accessed by the users to understand what exactly represents a certain object in the database. This system consists of two items, a *key* and a *value*. The key is used to describe a category or a topic, and its value designates the specific nature of this element's topic. Every element can have multiple tags, which are saved in a list that can be accessed when referring to that specific element. To clarify these concepts, an example is presented.

Example for keys and values in OSM elements:
Frankfurter Allee is one of the main streets in the city of Berlin, where cars, bikes and pedestrians can move in both directions. It has several lanes for vehicle traffic and different intersections with other important streets, such as Warschauer Straße or Otto-Braun-Straße. This long street is presented in OSM as a way element composed of different nodes. To simplify the explanation and facilitate understanding of the concepts, the analysed street is restricted to the segment between the U-Bahn stations Frankfurter Tor and Weberstraße, and presents an illustrative representation of nodes and ways that does not reflect the actual OSM mapping. Figure 2.4a presents the mentioned segment. To indicate if the street is one or two ways, OSM uses a key named *oneway*, and through its value, which can be a *yes* or a *no*, users can acknowledge the flow of that certain street. Another feature to be described is the maximum speed a car can drive at in that street. The usual tag to describe this is the *maxspeed*, which can have different values, such as a number (50 km/h) or an indication (DE:rural, indicating that the maximal speed is the assigned in rural roads in Germany). To indicate the number of lanes a street has (regardless of their direction), the *lanes* tag is used. Figure 2.4b displays a sketch of the selected segment from Frankfurter Allee street and some representative values to describe its features[1].

(a) Google Maps overview of a segment of Frankfurter Allee, Berlin.

Source: Google Maps (map data © Google); modified by the author.



(b) Frankfurter Allee's segment representation in OSM elements.

Source: Author's own work..

Fig. 2.4: Frankfurter Allee's OSM simplified representation.

Every category (key) can also have subcategories (subkey), following the format *key:subkey=value*. This structure can provide a more accurate description of the nodes and ways of interest for this project. For example, a polygon used to designate the Brandenburger Tor in Berlin, can be described by the tag *name = Brandenburg Tor*. But if users want to know what its name in specific languages is, they can check specific tags for this, such as *name:en = Brandenburg Gate* to see its name in English. In this case, the key is *name:en*, and the second element in the key is used to clarify what other specific information is given in the tag. Because of its easy-access policy and the structure of its data, and also because it is widely used in many other applications around the world, it is considered that this open source fits the interests and needs of this project, and it will be used to obtain the necessary information for the tool to be developed.

[1]Values shown are illustrative, they do not reflect actual data

2.4 Basic explanation on routing algorithms

A route optimisation algorithm is a mathematical model or computational method designed to solve complex problems in the most efficient way. Different techniques are applied within these algorithms, making each one suitable for a specific type of problem, as the requirements in terms of computational time or solution quality may vary for each case. The main groups into which algorithms can be classified are the following:

1) **Exact algorithms:** these methods always solve an optimisation problem to optimality. Classical examples include Dijkstra's algorithm or the Bellman-Ford algorithm (NextBillion.ai, 2023).

2) **Heuristic algorithms:** these approaches find good (but not necessarily optimal) solutions by applying different rules or guidelines. They are usually applied when the dataset is too large, and sacrificing solution quality in exchange for reduced computational time is reasonable (Laporte, 2009).

3) **Metaheuristic algorithms:** these are iterative strategies that combine different methods (often including heuristics) to explore the search space and avoid being trapped in local optima (Boussaïd et al., 2013).

4) **Greedy algorithms:** in most optimisation problems, there are several stages in the solution process where decisions must be made. Greedy algorithms make the locally optimal choice at each stage, with the aim of eventually reaching a global optimum (Laporte, 2009).

2.4.1 Dijkstra's algorithm

One of the most important algorithms in route optimisation is Dijkstra's algorithm, an iterative mathematical formulation used to find the shortest path between two points in a weighted graph. This method was published in 1959, and since then it has been developed and improved throughout the years. As described in Javaid, 2013, given a graph with a set of nodes connected by edges that have been assigned a weight, the algorithm begins an iterative process to determine the shortest path according to these weights, from an origin point A to a destination point B. In the original formulation, these weights were treated as distances, and the algorithm was designed to find the shortest path between two points. The methodology of the algorithm is as follows.

At the outset, a list of unvisited nodes is created, with a value assigned to each of them. This value is 0 for the origin node A, and infinity for all other nodes. It represents the known distance from each node to the destination point B. In each iteration, the node with the smallest known distance in this list (designated as node z) is selected. Subsequently, the distances of all its unvisited neighbours are updated in the main list by adding to node z's value the length of the edge connecting the two nodes. In a multigraph, a node can be accessed through different edges, linking it to different nodes. Therefore, when updating the values in the list, it may

occur that for a given node, two distinct paths allowing access to it are identified. In such cases, the values of both paths are considered, with preference given to the one associated with the shorter distance.

Once this process is completed, node z is removed from the list, marked as visited, and the next node with the smallest distance becomes the new node z. This iterative process terminates when the selected node is the destination point B, thereby achieving the objective.

This methodology ensures that the final path is the shortest route from A to B, since in every iteration each node's distance is updated to the shortest length known at that stage. What was originally defined as length can be generalised to weights, allowing the algorithm to be applied to other domains, such as cost or time optimisation.

3 Methodology

This chapter presents the methodology used to develop a routing optimisation tool that helps users determine the best route for freight transport under different criteria. It begins with data acquisition and the choice of the optimisation approach, followed by the construction of a multimodal network graph that includes Germany's road and rail networks. After that, the graph is weighted according to different criteria, and then a Dijkstra's algorithm code is developed to provide an optimisation approach to the problem. The goal of this section is to ensure that every step in the process is clear, transparent, and replicable.

3.1 Network data acquisition for route optimisation

In order to determine the optimal route for freight transport, it is important to have an accurate representation of the railway and road network in Germany and their features. These data need to describe the different routes used for freight transport in the country, as well as the main transfer points or facilities. In this work, transfer points are referred to as points of interest (POIs), which are intermodal rail stations where the transfer of goods can take place. The goal of this section is to clarify how these data are obtained from the OSM database, and what difficulties and filters have been applied to do so.

3.1.1 OSM road network data

As mentioned in Section 2.3.1, OpenStreetMap has a tagging system that helps identifying the nature and features of its elements. In the road network, the main key used to designate roads is *highway*, and its most relevant values within the German territory are displayed in Tab. 3.1[1]. The main purpose of the tool to be developed is to determine freight multimodal transport routes, carried out by heavy goods vehicles (HGV) when it comes to road transportation. Therefore, it is considered that the relevant road categories for freight transport are those tagged as *motorway*, *trunk*, *primary*, and *secondary*. The remaining road types are excluded, as by definition they represent more local roads, which do not contribute significantly to the model's objective. Including them would only increase the volume of data to process, resulting in longer computation times without improving the quality of the analysis. Some interesting attributes for edges and nodes in the road network are shown in Tab. 3.2.

On a first instance, the *maxspeed* key was considered to provide relevant information to introduce in the model, in order to estimate the speed of the vehicles and therefore their transportation time. However, the speed limit applied to each category of roads is applicable to passenger cars, while trucks used for freight transport are subject to stricter limits. Therefore,

[1] Some elements can also be tagged as connecting roads using the *highway* key, using values that are not in the Table, such as *motorway_link*, *trunk_link*, or *primary_link*.

Tab. 3.1: Main OSM road types for the German network.

Value	German-specific description
motorway	Federal motorway (Autobahn). Dual carriageway with separated directions, with usually two or more lanes per direction and an emergency lane.
trunk	Major road similar to a motorway ("gelbe Autobahnen").
primary	Federal highway (Bundesstraße). Designates main national roads connecting major cities, and also serves interregional traffic.
secondary	State or well-developed district road (Landesstraße/Staatsstraße/Kreisstraße). Connects smaller cities or towns and serves regional traffic.
tertiary	Local or district road (Kreisstraße or well-developed municipal road). Connects villages or serves as an important urban road.
unclassified	Minor public road with simple construction, usually no centre line. Used for small municipal roads that are not private or placeholders.
residential	Streets in residential areas, not belonging to higher classes. Designates access to and within housing areas.

OpenStreetMap Wiki — Map features: Roads.

using the *maxspeed* tag to establish an average vehicle speed for the model is discarded, and instead it is assumed that speeds will be assigned according to the type of road, designated by the *highway* tag.

When downloading data from the OSM database, it is important to choose the most convenient way to obtain and process it. In this work, two options have been considered: Overpass API, and *osmnx* Python library. For the road data acquisition, the Overpass API is selected, because of its accuracy and direct control over the raw data. Although *osmnx* offers functions to download network elements, the large size of the data often performs irreversible simplifications that can lead to an unclear tagging system. For example, when downloading ways, *osmnx* tends to merge multiple edges or ways to reduce data size and optimise download time, resulting

Tab. 3.2: Relevant attributes for nodes and edges in the road network.

Element	Relevant attributes	Description
Node	x	Longitude coordinate of the node
	y	Latitude coordinate of the node
	highway	Designates point features related to the highway network (pedestrian facilities, traffic signals...)
Edge	osmid	OpenStreetMap unique identifier for the edge
	length	Length of the edge (metres)
	reversed	Indicates directionality (if the edge is reversed)
	maxspeed	Maximum allowed speed on the edge
	highway	Designates type of road

Source: OpenStreetMap Wiki — Map features: Highway.

in longer ways with ambiguous *highway* tags. The Overpass API does not apply these simplifications, and for this reason, it is used to develop the `germany_overpass_road_map.py` script, which downloads the road network information (see Appendix B).

This script first fetches all the OSM elements within the German border whose *highway* tag has one of the following values: *motorway*, *motorway_link*, *trunk*, *trunk_link*, *primary*, *primary_link*, *secondary*, *secondary_link*. It is important to include links among the downloaded elements, since they designate connections between different types of roads, and without them the downloaded network would be incomplete, not allowing continuous paths when looking for the optimised route. After this, all nodes and edges having one of these values are downloaded. Then, these data are processed and saved to a file (`german_road_edges.json`), containing all nodes and edges that belong to the German road network, their *highway* tag, and, for edges, their length. This network is displayed in Fig. 3.1.

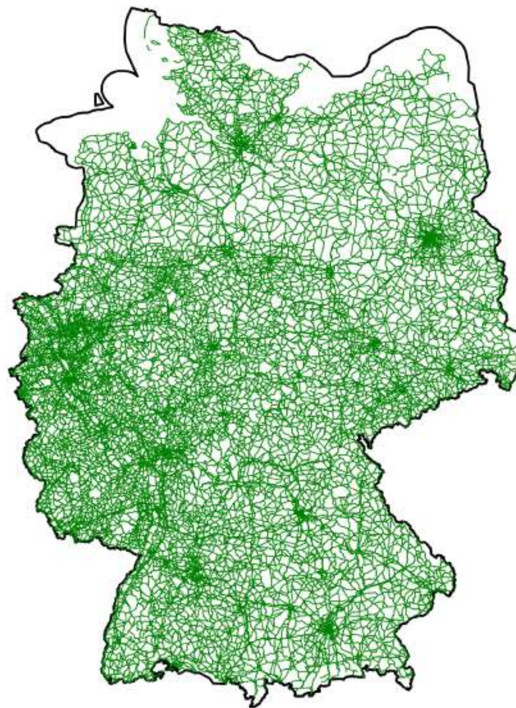


Fig. 3.1: Visual representation of the road network downloaded from OSM.

3.1.2 OSM rail network data

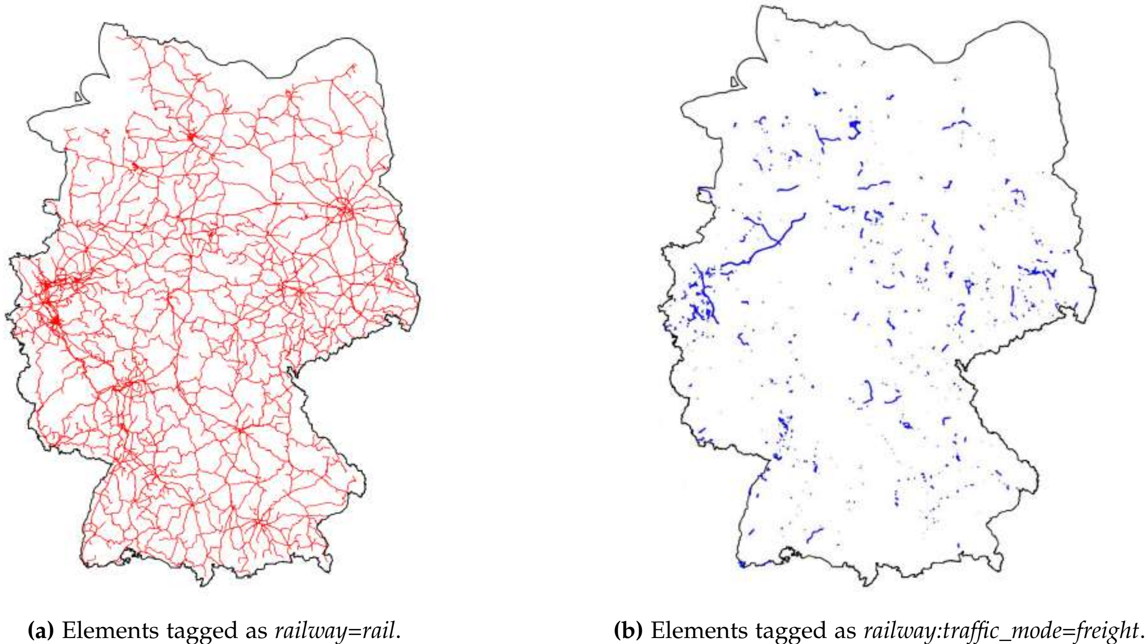
To obtain the data related to the railway infrastructure in Germany, the Overpass API has also been used in a Python script. Before explaining the final code, an explanation on how the data has been analysed and the assumptions made is presented. The relevant keys and values for identifying rail tracks in Germany are shown in Table 3.3 (all of them designate ways):

Tab. 3.3: Key–value mapping for OSM railway features.

Key	Possible Values	Description
railway	rail	Full-sized passenger or freight train tracks in the standard gauge for the country or state.
	industrial	Rail inside industrial facilities (e.g. factories, ports).
railway:traffic_mode	passenger	Designates only passenger rail lines.
	freight	Designates only freight rail lines.
	mixed	Designates lines used for freight and passenger transportation.

Source: OpenStreetMap Wiki — Map features: Railway .

The *railway=rail* tag is used to designate rail infrastructure for both passenger and goods transportation, and the subkey *traffic_mode* designates the specific use for these rail lanes. Figure 3.2a plots the elements tagged as *railway=rail* inside Germany, while Fig. 3.2b is a representation of all the elements from this network that are also tagged as *railway:traffic_mode=freight*. No elements tagged as *railway:traffic_mode=mixed* were found within Germany's borders in OSM. Figure 3.2 clearly shows how the *railway:traffic_mode* tag is not implemented for the majority of rail network elements. Therefore, it is assumed that the entire rail infrastructure is available for freight transport in the absence of more specific information.



(a) Elements tagged as *railway=rail*.

(b) Elements tagged as *railway:traffic_mode=freight*.

Fig. 3.2: Rail network elements downloaded from OSM within Germany's borders.

In Fig. 3.2, the country's border differs slightly from the actual land border. This is because the area considered includes parts of the surrounding sea. The elements to be downloaded then, contain information that can be relevant for the optimisation model. Some of their most interesting attributes are displayed in Tab. 3.4.

Tab. 3.4: Relevant attributes for nodes and edges in the railway network.

Element	Relevant attributes	Description
Node	x	Longitude coordinate of the node
	y	Latitude coordinate of the node
	railway	Indicates if the node is part of a railway
Edge	osmid	OpenStreetMap unique identifier for the edge
	length	Length of the edge (metres)
	reversed	Indicates directionality (if the edge is reversed)
	maxspeed	Maximum allowed speed on the edge

Source: OpenStreetMap Wiki — Map features: Railway.

The *maxspeed* attribute was initially considered a relevant variable to calculate the rail transportation time in the model, as it provides information about the maximum allowed speed along each edge of the network. However, after conducting an analysis of the available data, it was determined that only 31% of the network edges have a value assigned to this key. Therefore, it is considered that there is an insufficient amount of data labelled for *maxspeed*, and as a result, this attribute will not be taken into account in the model.

To download all this data, the script `germany_overpass_map_rail.py` has been developed (see Appendix C). This code accesses the OSM data through the Overpass API, selects all the elements within the German border tagged as *railway:rail* (nodes and ways), and returns them listed, including all their other tags used to describe other features. It then analyses all the downloaded elements, unifying their structure to different LineStrings (the way's structure) and tags them as *railway:rail* elements, saves their name (if those ways have it) and length. Finally, all these data are saved as a list of elements in a file (`germany_rail_edges.json`). It also plots the downloaded rail network on a map (see Fig. 3.2a).

3.1.3 Points of interest (POIs)

In a multimodal network, the most important nodes are those where a transfer of goods from one transportation mode to another can happen. Therefore, the points of interest for the model are considered all the possible intermodal terminals located in Germany. Although there is no specific tag in the OSM system that is used to refer to these types of rail station, there are other *key=value* sets that can be useful when looking for nodes or areas that represent these types of facilities. Table 3.5 displays the most relevant ones.

To access this information, the Python library *osmnx* has been used, because of its multiple functions that provide interesting and helpful information to find these elements. In the script, the *features_from_polygon()* function is used to get all the geometries within Germany that are tagged with one of the values displayed in Tab. 3.5. Its output is a GeoDataFrame that contains all the information and labels of 4806 elements which have either a node or an area geometry. To refine the obtained data, a filtering process is carried out. Firstly, all POIs labelled with a tag that contains the words *'abandoned'* or *'disused'* will be dismissed, since these keys are already describing the terminal's inoperability. The following filter targets stations tagged as public transport. The *public_station* label marks stations used for passengers' public transportation, and only if its value is 'no' may the station be treated as intermodal. This filtering process

Tab. 3.5: Key–value mapping for OSM POIs.

Key	Value	Description
railway	station	Designates a facility where trains load and unload passengers or goods (OpenStreetMap Wiki, 2025b).
railway	yard	Series of rail tracks used for sorting or storing railway wagons and locomotives (OpenStreetMap Wiki, 2025c).
railway	container_terminal	Rail terminal for freight transportation (**osm-container-terminal**).
building	station	Building constructed to be a train station, for passengers or goods (OpenStreetMap Wiki, 2025a).

discards 4163 of the previous points of interest, with a total of 643 POIs to be used for the model. Table 3.6 displays the distribution of the final POIs based on their tag.

Tab. 3.6: Counts of points of interest by label.

Tag	Number of elements
railway=yard	393
railway=station	156
railway=container_terminal	43
building=train_station	51
Total number of POIs	**643**

Once the final list of stations is generated, those elements with a non-Point geometry are converted into an equivalent point. This is calculated with the Shapely library function *representative_point()*, which returns a point cheaply calculated and guaranteed to be inside that geometry object (Shapely developers, 2025), and is used to standardise the data format. The final POIs locations are shown in Fig. 3.3. The script `pois_extraction.py` has been developed to download the points of interest mentioned (see Appendix D).

Fig. 3.3: Visual representation of the final POIs locations within Germany.

3.2 Algorithm design and graph construction

3.2.1 Criteria definition

The graph obtained from the OSM database serves as the foundation for building a weighted graph to be used with Dijkstra's algorithm. Since this mathematical method will be applied to optimise different criteria, a set of graphs will be developed to represent the characteristics of each node and edge in terms of cost, time, and environmental impact. The main aspect to address is how to determine the weight of each edge, considering that the algorithm will search for the path with the lowest total weight. In order to generate these graphs, certain assumptions will be made to make the problem more feasible and realistic. These statements are the following:

1) The algorithm is designed to optimise costs, emissions, or times strictly related to the transport route itself. Therefore, potential fixed external costs or emissions generated by elements associated with transportation, but not directly caused by the route, will not be considered.

2) The multimodal transport will be carried out in swap bodies, a transport container that can be easily attached to a truck and also to a railway platform. It is considered to be the most convenient container for multimodal transportation, since its characteristics eliminate the double handling of goods during the mode change. This results in cost and time reduction.

3) The definition of many criteria will depend on the amount of goods to be transported. To simplify and standardise the calculations, the amount of goods will be expressed in number of TEUs. A TEU is an acronym for "Twenty-foot Equivalent Unit", used to designate the load capacity of containers. Its dimensions are 20x8x8 feet. In the analysis of the algorithm, the number of TEUs to be transported (from now on q) will be defined.

4) The considered swap bodies are 40 feet long, which is the equivalent of 2 TEUs. Therefore, the heavy goods vehicles considered for the road transport in the algorithm will carry 2 TEUs. This is important to take into account when calculating costs and emission factors.

5) It is assumed that the swap bodies operate at full capacity, meaning they carry the maximum allowable load. According to German road legislation, a truck with the specified dimensions may not exceed a total weight of 40 tonnes (Eurowag, 2023). Considering the significant weight of the heavy goods vehicle structure itself, the maximum payload that the swap bodies can transport is 23 tonnes. Since one swap body is equivalent for 2 TEUs, the weight for each of these transport units is considered to be half of the total weight: 11.5 tonnes. This value will be used in the price calculations when the cost factors depend on the weight of the cargo.

Among the available data in OSM, the most reliable attribute is the length of each edge connecting the nodes. For this reason, each criterion will be defined based on this feature, establishing logical expressions that capture the most important aspects of each optimisation scenario. Another important element to consider is the set of connection edges, which represent the transfer operations between road and rail modes. In each case, the weights of these edges will be defined according to the actual characteristics and implications of the transfer process. It is important to properly define this penalty so that the algorithm reflects the fact that switching transport modes is not a simple operation and should be avoided when possible along the same route.

3.2.1.1 Distance optimisation graph

In this graph, the weight of the edges is defined as the distance in metres between the two nodes they connect. This attribute is provided by the OSM database for both the road and rail network edges. Since the connection edges have been manually added to the final graph, their predefined value for the length attribute is 0 m. Despite this being the most logical description of connection edges, since they represent a transfer operation and therefore don't imply additional distance, it is also interesting to consider somehow a length-related penalty for these edges. This way the algorithm is prevented from always choosing connection edges because of its null weight. To determine this penalty, the following assumption is made: the length of a connection edge will be the total distance a truck could travel on a German Autobahn (highway) during the time required for the mode-change operation. Therefore, it is important to determine the time needed to transfer goods from the road to the rail mode or the other way around. This operation takes place at intermodal train stations, which are facilities equipped with large areas for loading or unloading trucks, as well as storing containers, swap bodies, etc. Burdzik et al., 2014 analyses the usual steps involved in loading or unloading goods on a multimodal transport route, and then proposes ways to optimise the duration of

this process. Based on the description provided in that article, the most relevant activities in the process (excluding container loading or unloading) have been selected, and their duration in minutes is used as a reference to estimate the time needed for the transfer operation. The values are shown in Tab. 3.7.

Tab. 3.7: Estimated times for loading and unloading transfer operations.

Task	Description	Time (min)
Arrival at the terminal	Arrival of the trucks to the terminal, includes documentation management	31
Waiting time	Average waiting time to load or unload the trucks	60
Departure	Includes securing cargo and leaving the terminal	20

For the loading and unloading of TEUs, it is assumed that the cargo is moved directly from the truck to the train platform or vice versa, so no intermediate storage times are considered at terminal. An average of 100 seconds is assumed for the loading or unloading of one TEU (Wong, 2019). Once this criterion is defined, the expression for the operation time in hours during the mode change at an intermodal station is given as follows.

$$t = 1.85 + 0.028 \cdot q \tag{3.1}$$

Where t is the time and q is the number of TEUs to be transferred. Since the speed limit in highways for heavy vehicles is set at 80 km/h (DHL Freight Connections, 2023), the final expression for calculating the equivalent length (l) of connection edges is the following (in metres):

$$l = t \cdot 80{,}000 = (1.85 + 0.028 \cdot q) \cdot 80{,}000 \tag{3.2}$$

Once this expression is set, a preliminary graph is generated which will be used to validate the correct performance of the adapted algorithm.

3.2.1.2 Cost optimization graph

For the cost optimization of freight transport routes, a different rule should be applied to every type of edge. Since road and rail transport don't have the same characteristics, different aspects are taken into account.

Rail edges
According to METRANS, 2024, DB InfraGO released a report in 2023 stating that the price for transporting goods by rail in Germany would increase a 16% during 2024, reaching the value of 3.73 €/km. Considering the objective and scope of this work, it is assumed that the cost per train and kilometer is that stipulated by this company, also assuming that in reality it may be influenced in various ways by the characteristics of the train, the number of TEUs or the total weight of the vehicle, among many other factors.

Road edges
Road edges' cost takes into account two main elements. The first one considers the average price per km offered by transportation companies. The second one is a toll applied to every truck circulating on German roads. This charge is based on factors such as vehicle weight and emissions. To determine the average price per km in road transport, some data offered by Della, a road transportation company, has been analysed. The company displays the price

for their most recent orders on routes within Europe, giving an overview of the origins and destinations, the transported volume, and the average price per km. This data was checked on the 13th of August 2025, and is displayed in Tab. 3.8.

Tab. 3.8: Della's freight transport routes and rates (data from 13.08).

Date	Route	Distance	Load	Rate	Price per km
13.08	Włocławek — L'viv	697 km	22 t	1100.00 EUR	1.58 EUR/km
13.08	Warsaw — L'viv	399 km	22 t	1200.00 EUR	3.01 EUR/km
13.08	Pervomaisk — Poti	1209 km	22 t	3800.00 USD	3.14 USD/km
13.08	Lublin — Ternopil	346 km	22.5 t	700.00 EUR	2.02 EUR/km
13.08	Chełm — Ternopil	281 km	22.5 t	700.00 EUR	2.49 EUR/km
13.08	Chełm — Boromlya	932 km	22.5 t	1500.00 EUR	1.61 EUR/km
13.08	Lublin — Boromlya	964 km	22.5 t	1500.00 EUR	1.56 EUR/km
13.08	Zamość — Ternopil	258 km	22.5 t	700.00 EUR	2.71 EUR/km
13.08	Shostka — Chisinau	769 km	20 t	1200.00 EUR	1.56 EUR/km
13.08	Gebze — Kyiv	1251 km	22 t, 86 m^3	3909.70 EUR	3.13 EUR/km
13.08	Istanbul — Dnipro	1401 km	23 t, 86 m^3	3909.70 EUR	2.79 EUR/km
13.08	Brześć Kujawski — L'viv	703 km	20 t, 86 m^3	1250.00 EUR	1.78 EUR/km

Source: https://della.eu/cost/local/.

Multimodal transport is beneficial for companies when distances are big enough. Because of this, and considering Germany's dimensions, the average price per km is determined by calculating the mean of the routes in Tab. 3.8 that are between 300 km and 800 km long, given that their characteristics are probably similar to the possible domestic transport routes in Germany. The average value for km pricing is 1.99 €/km.

Regarding the road toll, the price per km is determined according to the type of vehicle (the more polluting, the higher the toll will be), and the weight of the truck. According to Eurostat, 2024, good vehicles of 5 years or less accounted for almost 80% of the vehicle-kilometers in Germany in 2023. Since 2016, the Euro VI regulation (Association for Emissions Control by Catalyst (AECC), 2024) has been in force, requiring new vehicles commercialized to meet certain emission standards and be categorised as Euro 6 vehicles. Because of this, it is assumed that all Euro 6 HGV are selected for the road transportation in the tool. Assuming a payload of 23 tonnes per truck, and based on the data presented in mtonroad.com, 2024 regarding prices in 2023, the toll to be considered is **0.19 €/km**. Considering these two elements, the final price per km in road transportation would be **2.18 €/km·truck**. Since every truck is transporting 2 TEUs, the final factor for each TEU is **1.09 €/km·TEU**.

Connection edges

Hintjens et al. (2020) presented a study on how cooperation between neighboring seaports can improve hinterland logistics. Despite its focus on sea transport, it also analyses the transfer of goods between different transportation modes, including road and rail. Part of the study involves comparing prices for various activities in logistics operations from the case study, to eventually determine an average price for each of them. Regarding the transfer of containers, the study states that it has an average cost of **50 €/TEU**. This rate will be considered to calculate the cost of connection edges in the cost optimization graph. Table 3.9 displays the functions used for cost calculation in each edge type. All of them depend on the total number of TEUs to be transported (q), and in the case of rail and road, also on the edge length (l).

Tab. 3.9: Cost expressions for each edge type in the network.

Edge type	Factor	Expression (€)
Rail	0.0374 €/m	$0.00374 \cdot l$
Road	0.00109 €/m·TEU	$0.00109 \cdot q \cdot l$
Connection	50 €/TEU	$50 \cdot q$

3.2.1.3 Time optimization graph

The time optimization graph is based on the distance optimization one (Section 3.2.1.1). For road and rail edges, the required time to travel the edge's distance is calculated considering the average speed of trucks or trains on those routes. The speed limit on German roads varies depending on the type of road. Since the graph network is composed of roads with different characteristics, a distinction between them is made in terms of speed. According to DHL Freight Connections, 2023, trucks are allowed to drive at a maximum speed of 80 km/h on main roads and highways, while on secondary or rural roads, the limit decreases to 60 km/h for vehicles over 7.5 tonnes. The final expressions for each edge's travel time are shown in Tab. 3.10.

According to Hintjens et al., 2020, the average train speed for freight transport is 55 km/h. Because train transportation is less dependent on fluctuations and less sensitive to possible traffic (as there is typically only one lane), this will be the average speed considered for calculating the travel time on railway edges. Regarding connection edges, the expression for their time weight has already been defined in Eq. (3.1), which depends on the number of TEUs to be transported (q). Table 3.10 summarizes all the expressions related to time-weighted edges.

Tab. 3.10: Time weight expressions for each edge type in the network.

Edge type	Time weight expression (h)
Road *[motorway/trunk/primary]*	$\frac{l}{80{,}000}$
Road *[secondary]*	$\frac{l}{60{,}000}$
Rail	$\frac{l}{55{,}000}$
Connection	$1.85 + 0.028 \cdot q$

Note: l is the length of the edge in meters, and q is the number of TEUs to be transported.

3.2.1.4 Emissions optimization graph

The emissions generated by road transport are determined using a factor that estimates the average grams of CO_2 per tonne·km transported. To do so, a report on CO_2 emissions from heavy-duty vehicles (European Automobile Manufacturers' Association (ACEA), 2020) has been taken into consideration to find a suitable value for this factor. The document was released to provide a preliminary estimate of the CO_2 emission performance of new HGVs in Europe registered during a specific period in 2019. To do so, the real performance of different

types of freight vehicles was analyzed using VECTO (Vehicle Energy Consumption Calculation Tool). This simulation software was developed in 2019 to accurately study the environmental impact of heavy-duty vehicles. Its estimations rely on characteristic parameters to determine the power consumption of each vehicle component, which include masses and inertias, rolling resistance, and engine performance, among others. All these inputs are used to simulate fuel consumption and CO_2 emissions for different vehicles. Based on how this tool works, the ACEA report provided average emission performance values for several vehicle types, as shown in Tab. 3.11.

Tab. 3.11: Overview of CO_2 emissions, payload, mileage and shares per vehicle category.

Category	Q3–Q4 share	Average CO_2 (g/tkm)	Payload [t]	Annual mileage [km]	Annual CO_2 [% of total] excl. 4-UD
4-UD	0.4%	—	2.7	60,000	—
4-RD	7.9%	198.1	3.2	78,000	4.7%
4-LH	1.9%	102.9	7.4	98,000	1.7%
5-RD	0.8%	84.0	10.3	78,000	0.6%
5-LH	62.8%	56.5	13.8	116,000	68.2%
9-RD	7.2%	110.9	6.3	73,000	4.4%
9-LH	9.2%	64.7	13.4	108,000	10.3%
10-RD	0.1%	84.0	10.3	68,000	0.1%
10-LH	9.7%	58.6	13.8	107,000	10.1%

Source: https://www.acea.auto/files/ACEA_preliminary_CO2_baseline_heavy-duty_vehicles.pdf.

Because it is assumed that trucks will be transporting 40 ft swap bodies, it is considered that the subgroup that best fits the context of this problem is the 9-LH vehicles, since they are long-haul vehicles with 4x2 tractor configurations. Given that the values provided are only for the transport of a single TEU, and considering that the trucks in this specific problem will be transporting 2 TEUs, the final value is doubled to remain consistent. The resulting factor for road edges is 129.4 g CO_2/t·km. Regarding the rail emissions, an article released by Escola Europea, 2024 states that an average freight train emits 24 g CO_2/t·km transported.

Finally, for the connection edges, an assumption has been made to simplify an estimation of this operation's environmental impact. Since the loading and unloading of swap bodies is carried out by reach stackers (vehicles used to transport containers over short distances, such as in intermodal exchanges), the emissions considered in these operations will be those released due to the use of fuel for these vehicles. It is assumed that the loading and unloading is carried out with only one reach stacker, model SMV 4123 CC5 from the Konecranes company. This vehicle can lift up to 41 tonnes, and its technical datasheet indicates that its average consumption during normal use is 15 to 18 L/h of fuel. Given that its engine is a 4-stroke Diesel, and that according to Autolexicon, 2024, 2,640 g CO_2 are produced when burning 1 liter of Diesel, the expression for the final emissions is described in Eq. (3.3). This equation takes into account the total time needed for loading and unloading a certain amount of TEUs (q), already described in Eq. (3.1) and dependent on the number of TEUs to be transported, and considers the average fuel consumption to be 16 L/h. The final emissions are expressed

in g of CO_2:

$$e_{\text{connection}} = 16 \cdot 2{,}640 \cdot 0.028 \cdot q = 42{,}240 \cdot 0.028 \cdot q \tag{3.3}$$

Table 3.12 presents the final expressions used to calculate emissions for each edge in the emissions graph, which is built upon the structure of the distance optimization graph.

Tab. 3.12: CO_2 emissions expressions for each edge type in the network.

Edge type	Factor	Expression (g CO_2)
Rail	129,500 g CO_2/tm	$129{,}500 \cdot q \cdot w \cdot l$
Road	24,000 g CO_2/tm	$24{,}000 \cdot q \cdot w \cdot l$
Connection	42,240 g CO_2/h	$42{,}240 \cdot 0.028 \cdot q$

Note: q is the number of TEUs, w is the weight per TEU in tonnes, and l is the length of the edge in meters.

3.2.2 Graph construction: assembly of the final multimodal network

Once the road and rail networks are downloaded from the OSM database, and the criteria are defined, the different optimization graphs described in Section Sec. 3.2.1 have to be generated. As previously mentioned, the weight assigned to each edge in every criterion is mainly a function of its length (l) and/or the number of TEUs to be transported (q). Table 3.13 displays a summary of all the assumptions made in the final processed data (described in Sec. 3.1), and applied when building this final multimodal network.

Tab. 3.13: Model assumptions by network.

Network	Assumption
Road	Local roads are not considered in the road network because of low relevance in long-distance multimodal routes. The *maxspeed* attribute is considered irrelevant since HGVs are subject to more restrictive speed limits on every type of road; other assumptions will be made.
Rail	All elements tagged as *railway=rail* are considered suitable for freight transport. The *maxspeed* attribute will not be considered because of its low usage among railway elements; other assumptions will be made for train speed.

The construction process is implemented in Python and is as follows. Firstly, the graph is created based on the files where OSM data is stored (see sections Sec. 3.1.1 and Sec. 3.1.2). Once all the information is loaded, it creates two separate undirected graphs: one with the rail network and the other with the road network. After that, both graphs are merged into the final multimodal network graph. This preliminary graph only contains length information about the edges (it is the distance optimization graph). Once this data structure is generated, the points of interest are uploaded from their file (see Sec. 3.1.3), and a search is launched to find the nearest node in both the road and the rail network to each point of interest. If the distance between the equivalent road node and the POI is less than 15 km, and less than 3 km

between the rail equivalent node and the POI, the point of interest is considered suitable for the network, and a connection edge is established between the road and the rail equivalent nodes, in both ways. This edge is described with a length attribute, which corresponds to the connection distances explained in Sec. 3.2.1.1.

Once the distance optimization graph is created, every edge in the graph has a length attribute. With this parameter, together with the number of TEUs being transported (indicated as an input to this script), the weight according to each criterion and type of network is calculated and assigned to each edge (see Sec. 3.2.1). This results in every edge having a list of different attributes, which represent the weight of the edge according to a certain criterion: length, cost, time or emissions. The completed network that will be used within the optimization model is the one in Fig. 3.4, and its general features are displayed in Tab. 3.14.

Tab. 3.14: Multimodal graph features.

Multimodal graph features	
Number of rail nodes	975,426
Number of rail edges	977,071
Number of road nodes	5,032,529
Number of road edges	5,118,635
Number of connection edges	600

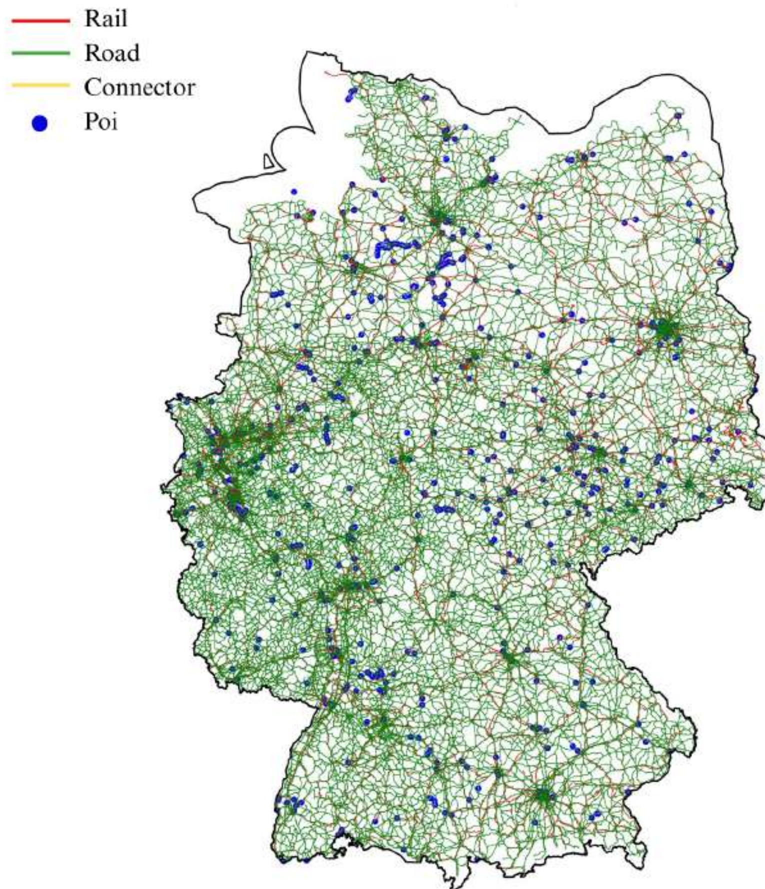


Fig. 3.4: Visual representation of the final multimodal network to be used in the tool.

This graph construction process has been implemented with the Python script `weighted_graph.py` (see Appendix E). The code also incorporates a function that, given a previously built file with the final multimodal network according to a certain number of TEUs (q), can recalculate its edges weights values when a new q value is given. This feature is interesting to replicate graphs with different characteristics in a more efficient and fast way.

3.2.2.1 Implementation of Dijkstra's algorithm for multimodal transport routes optimization

The final step for the tool implementation is the coding of Dijkstra's algorithm. A Python code has been developed, which given certain parameters, the optimal multimodal route between two points is calculated. These parameters are the following ones:

1) **Number of TEUs (q)**: To be considered in the route.

2) **Start and goal locations**: these are the longitude and latitude coordinates of the origin and destination of the route of interest.

3) **Optimization criteria**: Dijkstra's algorithm will only take into account the weights related to the selected criterion. This input indicates the criterion approach: distance, cost, time or emissions.

Once the algorithm has found a solution, the final path is saved in a file, and plotted in a figure to be observed. All this process has been implemented in the `routing_algorithm.py` file (see Sec. F.2).

4 Algorithm validation and performance analysis

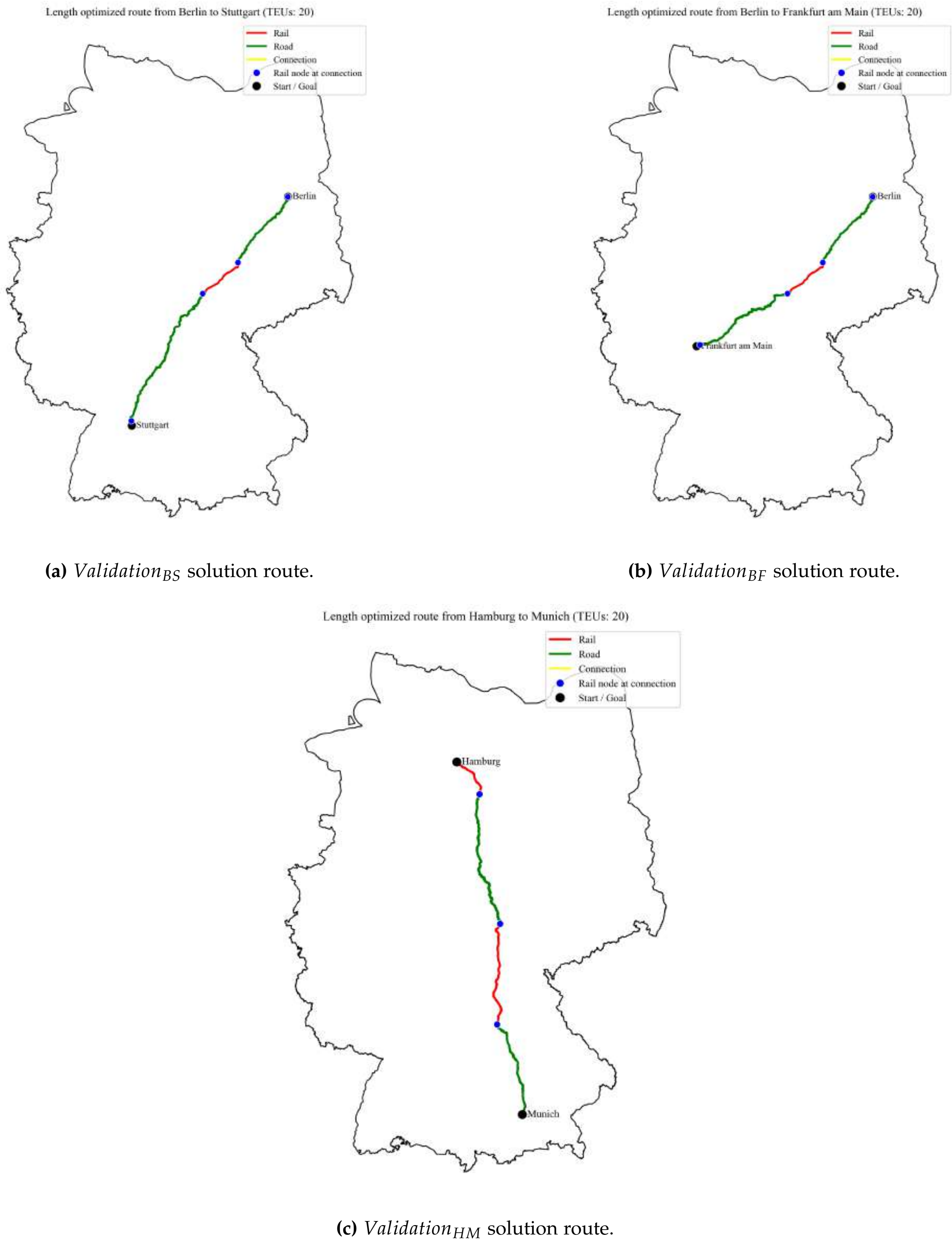
In this chapter, the performance of Dijkstra's algorithm in a multimodal network is analysed. First, a validation of its right processing of data will be conducted. After that, the behaviour of the algorithm according to different criteria will be displayed. Finally, a sensitivity analysis will be conducted, regarding the tool's behaviour when introducing a new structure to rail costs.

These analyses have been conducted considering different locations around Germany. Most of these locations have been selected because of the type of industry around that area (Munich) or because of its other logistics facilities (Hamburg because of its port), since this implies a higher flow of goods in the city, and therefore it makes it more interesting to see how the algorithm creates routes from these logistically relevant hubs. To designate each of these locations, a single point has been selected.

4.1 Algorithm validation

To verify the algorithm's performance, and to ensure it can build multimodal routes, different tests have been run. For these tests, the distance optimisation graph (see Sec. 3.2.1.1) has been used, so that the algorithm finds the shortest route between the origin and goal point. In a first instance, **the connection edges' distance weight has been established to 0 m**, since connection operations do not imply more transportation distance. These tests have all been executed considering a total of 20 TEUs to be transported, both in the rail and road transport. For more clarity and easy identification of every test, a "Test ID" label has been created: $Validation_{OD}$, where O is the first letter of the city located at the origin of the route, and D the first letter from the destination city. Table 4.1 displays the results for the validation tests: if the solution route is multimodal or not, its total length, and the distance executed in each transportation mode for every solution route, besides the execution time. In Tab. 4.1, the routes that have a rail route below 10 km are not considered to be multimodal. On average, Dijkstra's algorithm needs 35.56 seconds to find the optimal path in the provided network. It has also been observed, that higher execution times happen when two or more tests are run simultaneously. The multimodal routes provided by the algorithm in the validation tests are displayed in Fig. 4.1. The ones considered as not multimodal are represented in Fig. 4.2.

In Fig. 4.2, the plotted routes are considered to be unimodal despite showing one or two mode changes. These routes are mainly road-based, whose start point is closer to a railway node, and therefore the algorithm looks for the shortest path to a POI to transfer the goods to road mode, since it implies less route distance. To explain this, the $Validation_{BM}$ test will be analysed (see Fig. 4.2a). The route starts in Berlin transporting goods in rail mode, but in the same city, 1.01 km further from the start point, a mode change is conducted, so the rest of the route uses the road network. The rest of examples also present some transfer operations, but all of them happen less than 10 km away from the origin or destination point.

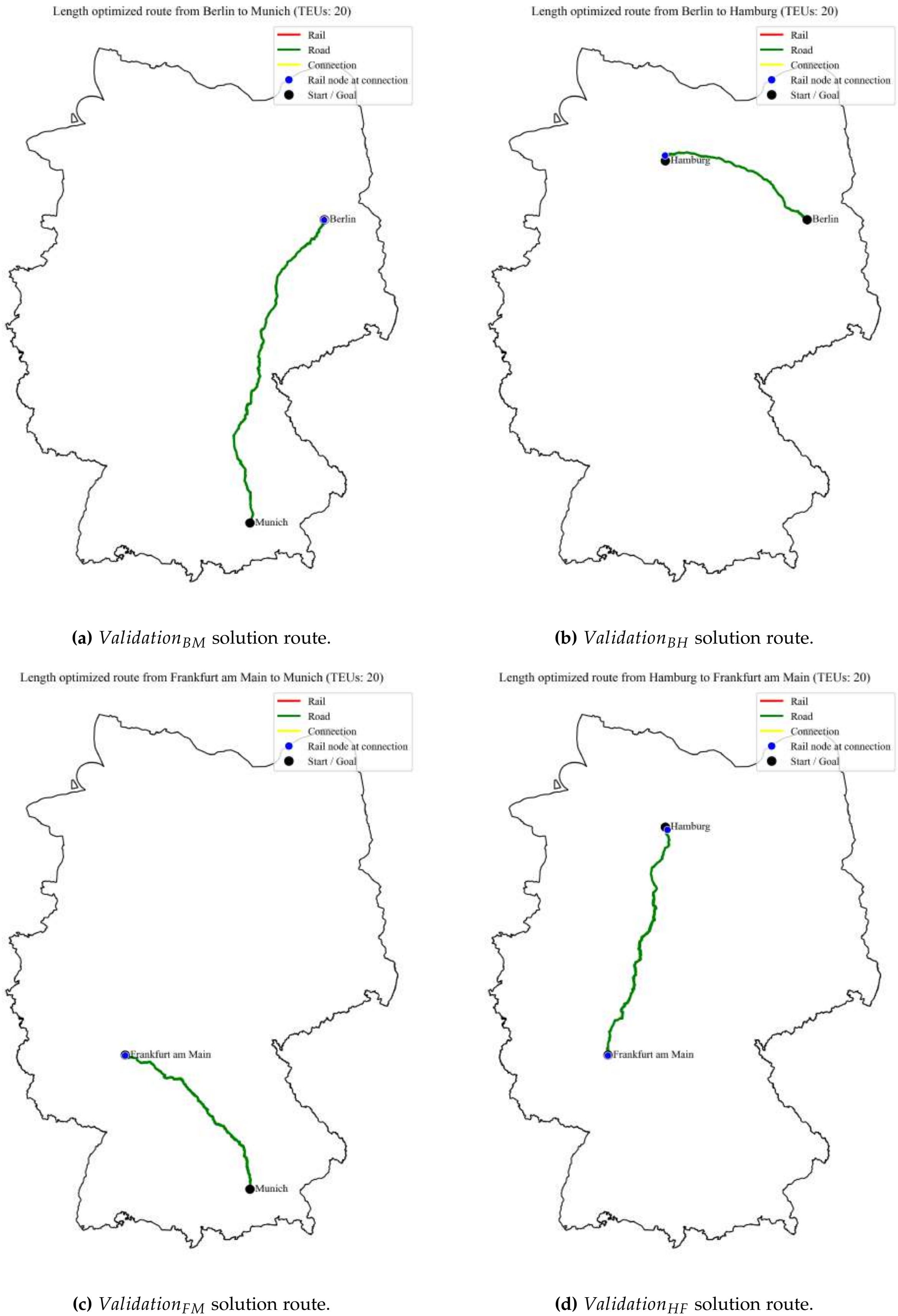


(a) $Validation_{BS}$ solution route.

(b) $Validation_{BF}$ solution route.

(c) $Validation_{HM}$ solution route.

Fig. 4.1: Multimodal solution routes in validation tests.



(a) $Validation_{BM}$ solution route.

(b) $Validation_{BH}$ solution route.

(c) $Validation_{FM}$ solution route.

(d) $Validation_{HF}$ solution route.

Fig. 4.2: Non-multimodal solution routes in validation tests.

Tab. 4.1: Algorithm validation tests summary.

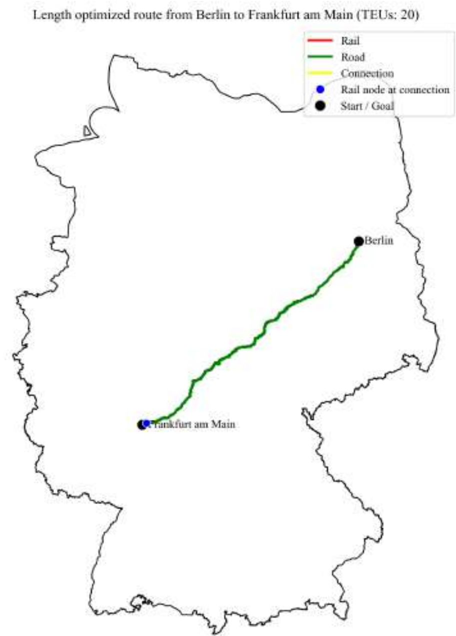
Test ID	**Origin**	**Destination**	**Multimodality**	**Total route length (km)**	**Road distance (km)**	**Rail distance (km)**	**Run time (s)**
$Validation_{BS}$	Berlin	Stuttgart	yes	591.35	478.35	113.01	40.13
$Validation_{BM}$	Berlin	Munich	no	583.03	582.01	1.01	22.74
$Validation_{BF}$	Berlin	Frankfurt am Main	yes	503.99	395.24	108.75	38.50
$Validation_{BH}$	Berlin	Hamburg	no	283.26	273.48	9.78	14.73
$Validation_{MH}$	Munich	Hamburg	yes	691.41	433.27	258.15	23.69
$Validation_{MF}$	Munich	Frankfurt am Main	no	377.30	367.84	9.46	65.82
$Validation_{HF}$	Hamburg	Frankfurt am Main	no	43.35	451.78	17.43	43.35

The other validation tests have a multimodal solution, and their routes are shown in Fig. 4.1. These scenarios present routes with more than 2 transfers between the different networks, dividing the route in different segments that combine road and rail freight transportation. All of these validation tests provide clear evidence that the algorithm can find multimodal paths as a solution. However, these routes are not being reproduced in real-life scenarios. Because the transfer of goods is a costly and time-consuming operation, a multimodal route with more than one or two changes of modes is not convenient. Adding a weight as a penalty to connection edges in the distance optimization graph allows the tests to be more realistic and provide more coherent routes. To observe if the behaviour of the algorithm is different adding penalties to transfer operations, the multimodal routes obtained in the validation test are again tested, **now considering a weight for connection edges** that follows the expression Eq. (3.2). The new results are shown in Fig. 4.3

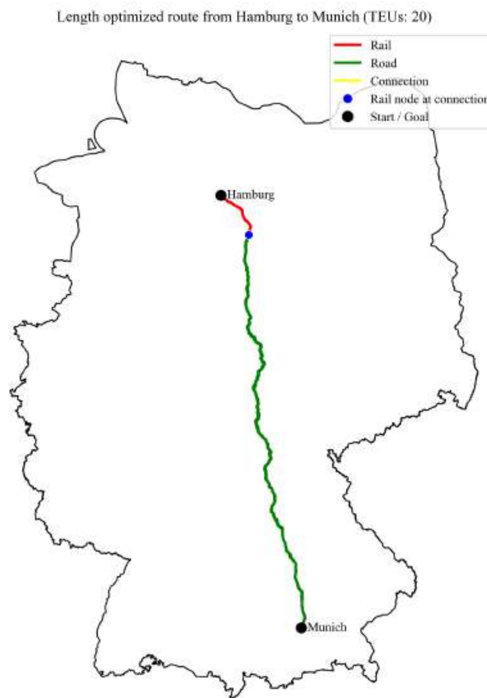
These new validation tests clearly show how adding a penalty every time a change of transportation mode occurs can change the algorithm behaviour. In the $Validation_{BS}$ and $Validation_{BF}$ tests, the routes have been transformed to mainly road-based paths (see Fig. 4.3a and Fig. 4.3b, respectively), avoiding the various transfer operations presented in the equivalent tests without penalties. The $Validation_{HM}$ test still provides a multimodal solution, but the route presents a single transfer operation, giving a more coherent and realistic path to be executed in a real-life scenario. This proves that the algorithm's results are more realistic when introducing a weight to a transfer operation, and that it is necessary to replicate this methodology when studying the optimal route according to other criteria.

Finally, to verify that the algorithm indeed provides the shortest route, the $Validation_{HM}$ scenario is analysed in detail. Taking into account the results provided in the simulation of the shortest route between Hamburg and Munich without penalties in the transfer operations, the path plotted in Fig. 4.1c is considered to be the shortest in the multimodal network. If the algorithm could only consider road or rail paths between these two locations, the route would imply more distance and therefore it could still be optimized. Table 4.2 displays the length of the only-road or only-rail routes between Hamburg and Munich, clearly showing that, distance-wise, the algorithm's solution is the shortest path.

(a) $Validation_{BS}$ solution route (with transfer penalty).



(b) $Validation_{BF}$ solution route (with transfer penalty).



(c) $Validation_{HM}$ solution route (with transfer penalty).

Fig. 4.3: Solution routes for validation tests considering transfer penalties.

Tab. 4.2: $Validation_{HM}$ test verifications.

Origin	**Destination**	**Type of route**	**Route length (km)**	**Road distance (km)**	**Rail distance (km)**
Hamburg	Munich	Multimodal	691.99	433.27	258.15
Hamburg	Munich	Only road	701.87	701.87	0.00
Hamburg	Munich	Only rail	763.14	0.00	763.14

Tab. 4.3: Route tags..

Test ID	**Origin - Destination**
$Criteria^{i}_{BM}$	Berlin – Munich
$Criteria^{i}_{HF}$	Hamburg – Frankfurt am Main
$Criteria^{i}_{HM}$	Hamburg – Munich
$Criteria^{i}_{HS}$	Hamburg – Stuttgart
$Criteria^{i}_{MBir}$	Munich – Birkenfeld

4.2 Performance analysis under different routing criteria

After the performance of the algorithm is validated, an analysis of its behaviour according to different criteria is conducted. To do so, the algorithm has been applied to different routes considering the three criteria stated in the scope of this thesis: cost, time, and CO$\underbrace{2}$$emissions.Thetestscarriedoutinthissectionareidentifiedwithanew TestID$:Criteria$^{i}_{OD}$. In this label, O is once again the first letter of the origin point in the route, D is used to designate the final destination, and the over index i indicates the optimization criterion used for that test: *cost*, *time* or *emissions*. Table 4.3 presents a summary of the origins and destinations of the routes analysed in this section. For each one of these itineraries, three scenarios have been tested, and the corresponding solution routes features are displayed in Tab. 4.4. It can also be interesting to compare and contrast the features of every test, such as the route total distance, cost, time, and total emissions. All this information is summarized in Tab. 4.5. All tests are run considering 20 TEUs to be transported.

Tab. 4.4: Criteria tests values summary.

Test ID	Criterion	Units	Values			
			Total	**Road**	**Rail**	**Connection**
$Criteria_{BM}^{cost}$	cost	€	4,672.44	322.96	2,349.48	2,000.00
$Criteria_{BM}^{time}$	time	h	7.30	7.30	0.00	0.00
$Criteria_{BM}^{emissions}$	emissions	t CO_2	3.95	0.38	3.47	0.09
$Criteria_{HF}^{cost}$	cost	€	1,890.57	0.00	1,890.57	0.00
$Criteria_{HF}^{time}$	time	h	9.19	0.00	9.19	0.00
$Criteria_{HF}^{emissions}$	emissions	t CO_2	2.79	0.00	2.79	0.00
$Criteria_{HM}^{cost}$	cost	€	4,003.86	129.60	2,874.26	1,000.00
$Criteria_{HM}^{time}$	time	h	11.28	8.73	0.15	2.41
$Criteria_{HM}^{emissions}$	emissions	t CO_2	4.44	0.18	4.24	0.02
$Criteria_{HS}^{cost}$	cost	€	2,524.93	0.00	2,524.93	0.00
$Criteria_{HS}^{time}$	time	h	12.28	0.00	12.28	0.00
$Criteria_{HS}^{emissions}$	emissions	t CO_2	3.73	0.00	3.73	0.00

Tab. 4.5: Criteria tests: overall features (by route).

Test ID	Total distance (km)	Total cost (€)	Total time (h)	Total emissions (t CO_2)
$Criteria_{BM}^{cost}$	643.02	4,672.44	16.42	3.96
$Criteria_{BM}^{time}$	583.87	12,728.39	7.30	17.38
$Criteria_{BM}^{emissions}$	642.06	6,633.26	21.22	3.95
$Criteria_{HF}^{cost}$	505.50	1,890.57	9.19	2.79
$Criteria_{HF}^{time}$	505.50	1,890.57	9.19	2.79
$Criteria_{HF}^{emissions}$	505.50	1,890.57	9.19	2.79
$Criteria_{HM}^{cost}$	774.46	4,003.86	16.45	4.44
$Criteria_{HM}^{time}$	706.12	16,249.34	11.28	20.85
$Criteria_{HM}^{emissions}$	774.46	4,003.86	16.45	4.44
$Criteria_{HS}^{cost}$	675.12	2,524.93	12.27	3.73
$Criteria_{HS}^{time}$	675.12	2,524.93	12.27	3.73
$Criteria_{HS}^{emissions}$	675.12	2,524.93	12.27	3.73

To study the algorithm's behaviour, each scenario is examined separately. For the $Criteria_{BM}^{i}$ tests, the algorithm suggests three different routes, shown in Fig. 4.4. In this case, the algorithm returns a distinct route for each optimisation criterion. For cost and emissions optimisation, the most convenient routes mainly use the rail network. These two routes are very similar, differing only slightly near the start of the route (close to Berlin). In the emissions scenario, the

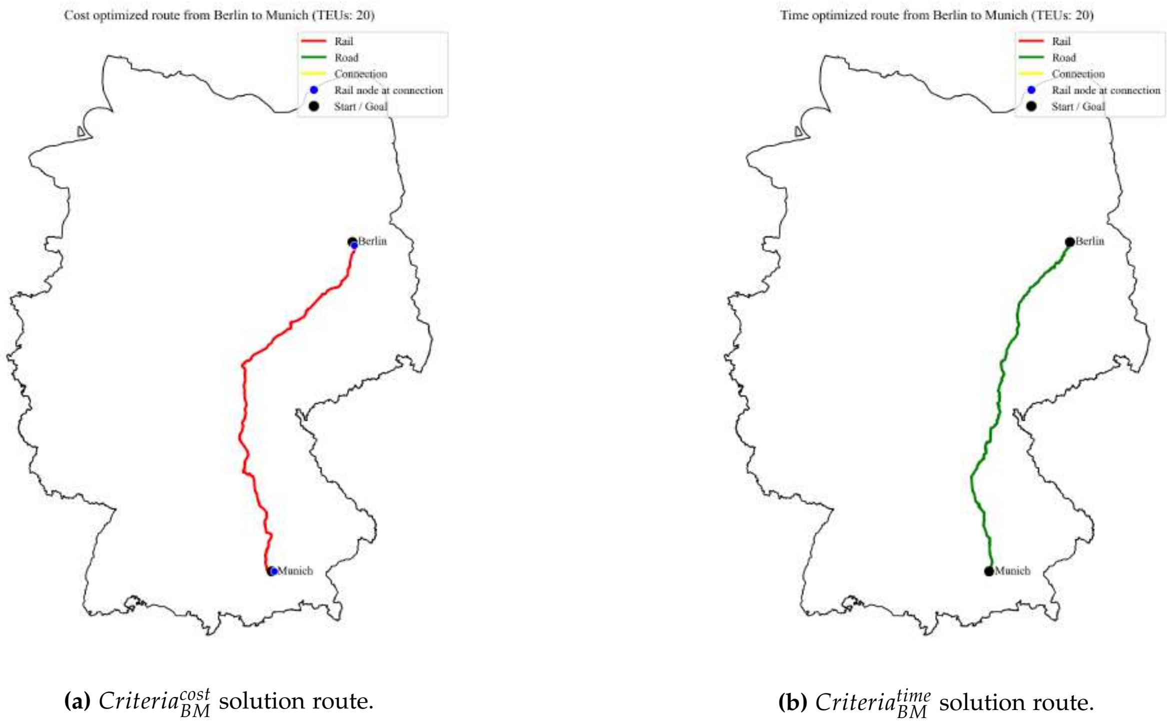
algorithm begins by road, performs two transfer operations near the city, then follows rail for most of the journey, and finally transfers near Munich to complete the last segment by road. In the cost scenario, the route is almost the same but with only one initial transfer, followed by the same rail-to-road change near Munich. In the time-optimisation test, the algorithm chooses an only-road route, presenting better time results, but worse cost and emissions features. In the itinerary between Berlin and Munich, the solution obtained in the cost-optimisation test presents better results than the emissions' solution path, given that — relative to the cost-optimised path — cost and time increase by 41.9% and 29.2%, respectively, while emissions decrease by only 0.2% in the emissions-optimised path. This shows that, even when optimising for a single criterion, it is important to analyse all scenarios and assess whether the opportunity cost of improving one feature is worth the loss in others.

For other origin–destination pairs, the algorithm suggests the same route across all criteria scenarios. This occurs for the Hamburg–Frankfurt and Hamburg–Stuttgart itineraries. As shown in Tab. 4.5, the $Criteria^{i}_{HF}$ results are identical for every value of i, and the same behaviour appears in the rows for the $Criteria^{i}_{HS}$ tests. Figure 4.5 and Fig. 4.6 present the corresponding route visualisations. In all scenarios, both itineraries run entirely on the rail network.

For the Hamburg–Munich itinerary, the algorithm shows behavior similar to the Berlin–Munich case. For cost and emissions optimisation, the final path is the same rail-based route, with a single transfer near Munich to complete the last segment by road (see Fig. 4.7a and Fig. 4.7b). By contrast, the time-optimised route prioritises the road network, with one initial transfer near Hamburg (see Fig. 4.7c).

Overall, across the four itineraries, the algorithm tends to produce single-mode routes (even if short early transfers occur). When cost or emissions are the optimisation criterion, rail is prioritised, and the resulting rail-based routes deliver better cost and emissions outcomes—regardless of the criterion—than comparable road-based alternatives. In contrast, when optimising for time, the algorithm oscillates between rail- and road-based routes. This likely reflects that the downloaded road network is larger than the rail network, increasing the chance that origin and destination are connected by an all-road path, which reduces the number of transfers and, consequently, transfer times. The algorithm's tendency to favour rail follows directly from the emissions and cost assumptions used. Rail has lower specific emissions than road for comparable freight movements, so any criterion that weights environmental performance will naturally lean towards rail. This is clearly exemplified in the $Criteria^{i}_{HM}$ itinerary: the time-optimised route is a road path, while for the emissions and cost optimisation, the route is completely carried out in the rail network. Analysing both road and rail routes, the tonnes of CO_2 caused by the rail route are 4.44, with a total distance of 774.46 km. Meanwhile, in the road route, the total emissions rise to 20.84 tonnes of CO_2, over four times more than in the other route. Despite the road route being 68 km shorter, the emissions' difference with the rail route is too big to consider a change of modes. On the cost side, rail charges are modelled as distance-based (€/km) and independent of the number of TEUs, whereas road costs scale with both distance and TEU count. As shipment size grows, road becomes relatively more expensive while rail remains distance-driven. Given these settings, it is expected that rail is preferred in both cost- and emissions-oriented scenarios, matching the results observed.

The algorithm tends to choose a route mainly based on one transportation mode, sometimes with a modal change at the beginning of the route due to the connection with the origin point selected. This behaviour can be explained through the fact that all the selected locations to be origin or destination points are mainly big and relevant cities in Germany. Therefore, it



(a) $Criteria_{BM}^{cost}$ solution route.

(b) $Criteria_{BM}^{time}$ solution route.

(c) $Criteria_{BM}^{emissions}$ solution route.

Fig. 4.4: $Criteria_{BM}^{i}$ solution routes.

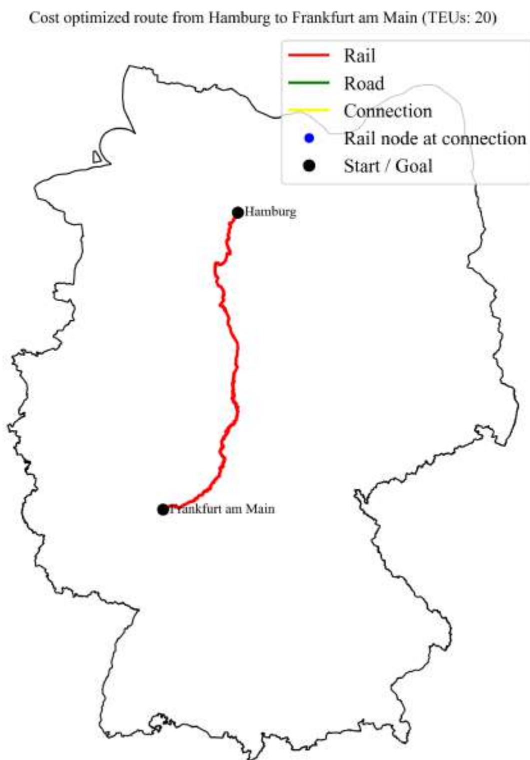


Fig. 4.5: $Criteria_{HF}$ solution route for every criterion.

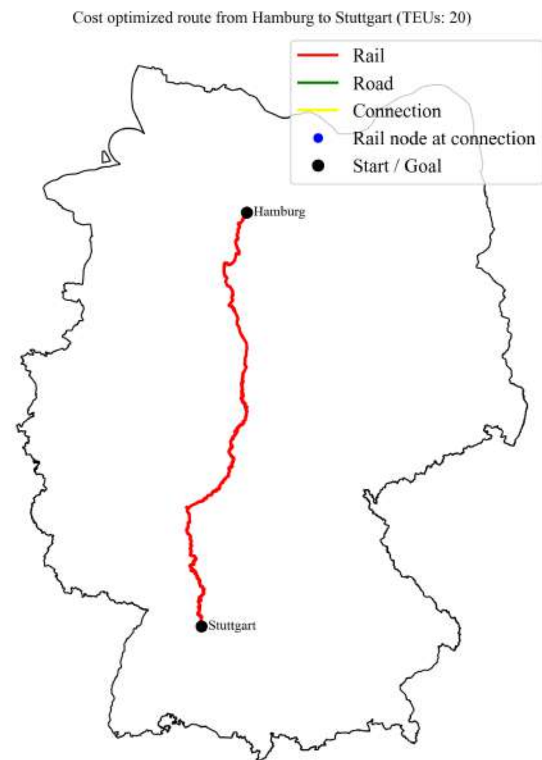
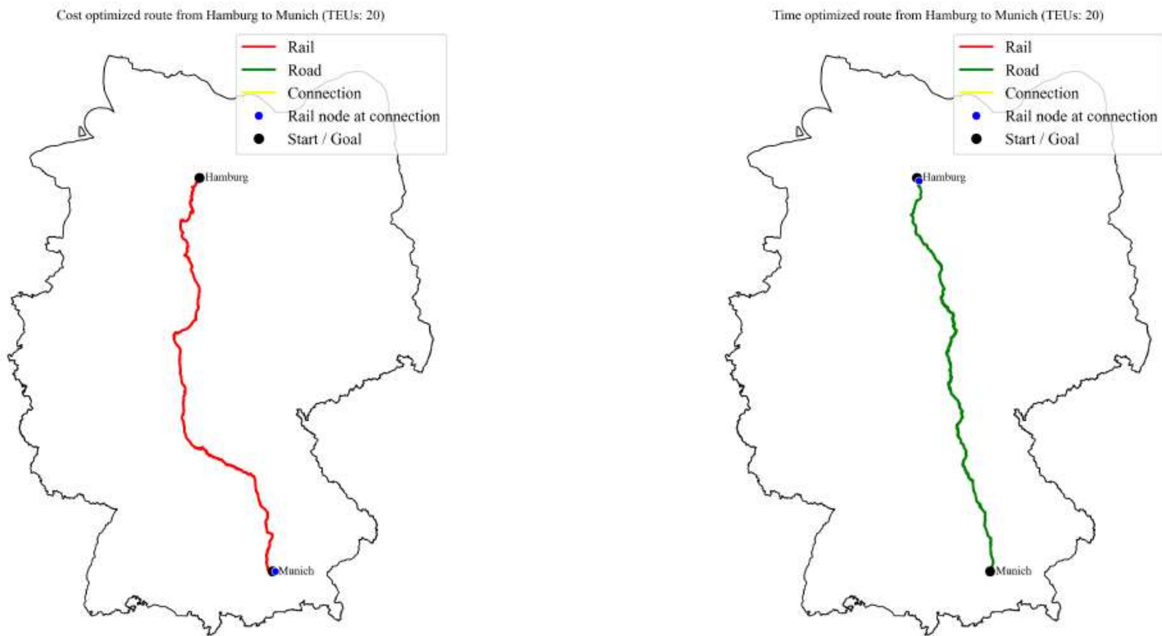
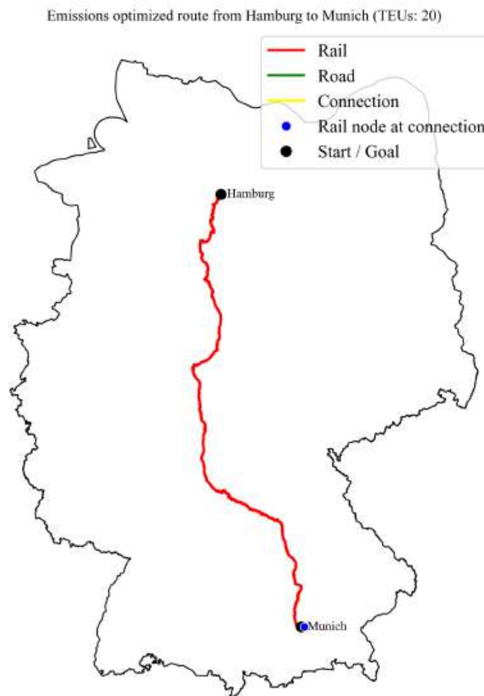


Fig. 4.6: $Criteria_{HS}$ solution route for every criterion.



(a) $Criteria_{HM}^{cost}$ solution route.

(b) $Criteria_{HM}^{time}$ solution route.



(c) $Criteria_{HM}^{emissions}$ solution route.

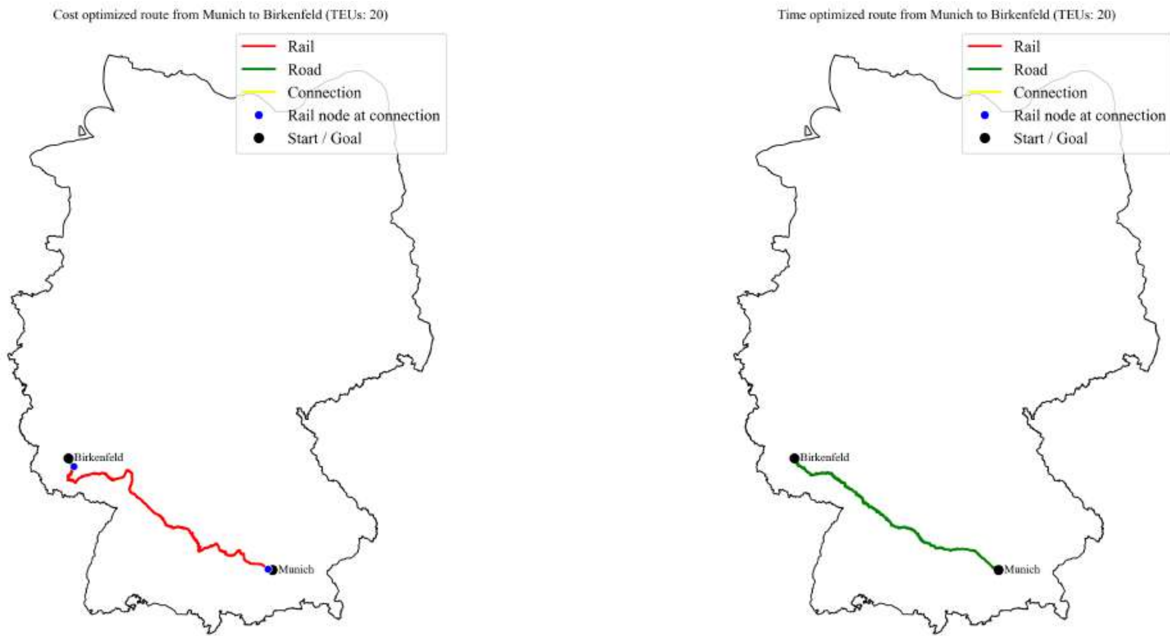
Fig. 4.7: $Criteria_{HM}^{i}$ solution routes.

is logical that their road and rail connections are good enough to, given one certain criterion, avoid the penalties of modal changes and take an uninterrupted path to get to their destination. Because of that, a scenario with a smaller town as one of the endpoints has been tested. To exemplify the code's performance in this scenario, the Munich–Birkenfeld route has been chosen ($Criteria^{i}_{MBir}$). Birkenfeld is a town located in the south-west of Germany, and it has been chosen because, observing the networks, it may be less easily connected with other important cities in Germany. Table 4.6 displays the characteristics of the three solutions found when optimising the route between these two locations.

Tab. 4.6: Criteria test 5: route overview.

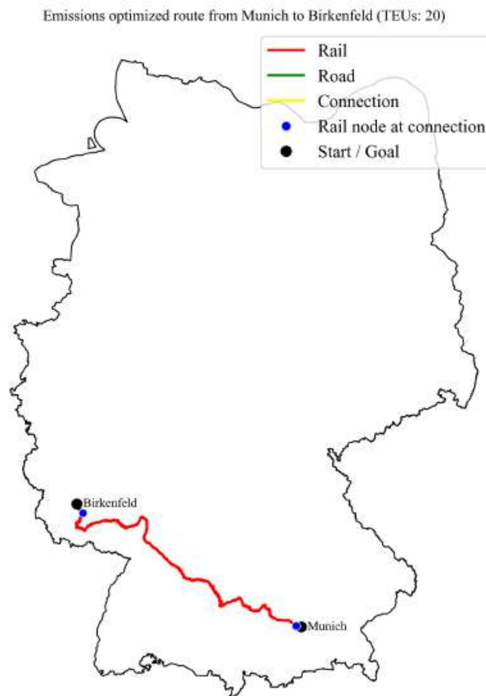
Test ID	Origin	Destination	Criteria	Total distance (km)	Route description
$Criteria^{cost}_{MBir}$	Munich	Birkenfeld	cost	504.58	Road – Rail – Road
$Criteria^{time}_{MBir}$	Munich	Birkenfeld	time	424.71	Only road route
$Criteria^{emissions}_{MBir}$	Munich	Birkenfeld	emissions	504.58	Road – Rail – Road

The route description in Tab. 4.6 indicates that the algorithm's performance in optimising time is similar to previous tests: it provides a path only by road, a unimodal route (see Fig. 4.8b). However, when optimising costs and emissions, the route presents a stronger multimodal approach (see Fig. 4.8a or Fig. 4.8c): it begins with an 8 km road segment to a point of interest, where a transfer of goods occurs; the following 476 km proceed on the rail network; finally, a second transfer operation takes place, and the last 20 km are completed by truck, reaching the town of Birkenfeld. This is a clear example of how the algorithm can provide coherent multimodal routes. In this specific case, the aim was to maximise the rail network usage to reduce CO_2 emissions or costs, adding short road segments where necessary to provide access to it. These tests put in evidence that the algorithm is capable of giving coherent and logical multimodal routes, that make sense to be implemented in a real-world scenario.



(a) $Criteria_{MBir}^{cost}$ solution route.

(b) $Criteria_{MBir}^{time}$ solution route.



(c) $Criteria_{MBir}^{emissions}$ solution route.

Fig. 4.8: $Criteria_{MBir}^{i}$ solution routes.

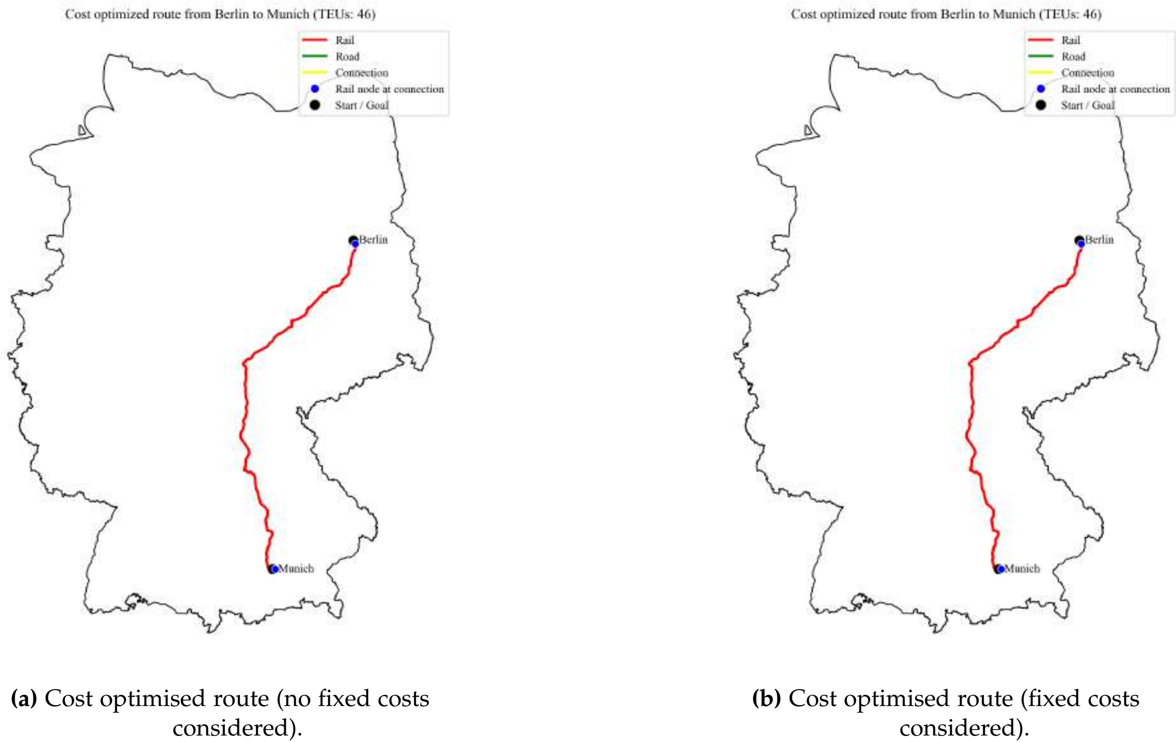
4.3 Sensitivity analysis on rail costs

The previous sections in this chapter show how the algorithm clearly tends to provide mainly rail-based solutions with different optimisation criteria. This occurs because the rail cost formulation is based on route mileage, regardless of the number of TEUs transported (since the estimate provided by DB Cargo (METRANS, 2024) presented the information on freight trains in this way). When comparing the per-TEU price of road and rail, road appears cheaper at first glance; however, once you multiply by the number of trucks needed to move all the stated TEUs, the cost of the road option is increased.

This section aims to analyse how the algorithm's behaviour changes as the rail transport cost is increased. We assume that the information provided by DB Cargo corresponds to the price of rail transport. Even so, the cost of hiring or making a train available to a company to move its product goes beyond a per-kilometre rate. Using a train entails fixed costs such as the use of a locomotive and the potential rental of wagons, among others. For that reason, an analysis of the differences between the algorithm's current setup and an alternative in which an initial fixed charge is applied each time a rail movement begins is conducted. In practice, this means that when the algorithm decides to start the route by rail, or chooses to switch from road to rail (but not the other way around), a fixed cost for using that train is added. Consequently, if a route starts by rail, switches to road, and several kilometres later switches back to rail to finish, the total trip cost would increase—on top of what the algorithm already accounts for—by twice the fixed price of putting a train into service.

To determine a basic fixed price for a train, the values reported by the Department of Business Administration at the School of Business, University of Gothenburg (Sweden), within the project *Strategic modelling of combined transport between road and rail in Sweden* are taken into account (Flodén, 2011). That report estimates the price of moving goods by rail and splits it into fixed and variable costs, proposing several scenarios with trains of different characteristics. The first scenario is taken as a benchmark: a 30-wagon train (60 TEUs) operating at 75% capacity, thus considering 23 wagons, which translates into 46 TEUs. In this scenario, fixed costs (covering traction, labour, and wagon use) are estimated at 23,979 SEK, roughly 2,170 €. With this value set, an analysis is conducted, of cost optimisation for specific routes—first without fixed costs, and then including them. The analysis will be carried out on the following routes (already examined earlier): Berlin–Munich, Hamburg–Munich, and Munich–Birkenfeld. In all cases, the route is optimised by cost and assumes a total of **46 TEUs**. Figure 4.9, Fig. 4.10 and Fig. 4.11 display the solution routes for the cost optimisation of the mentioned itineraries, with and without the fixed costs.

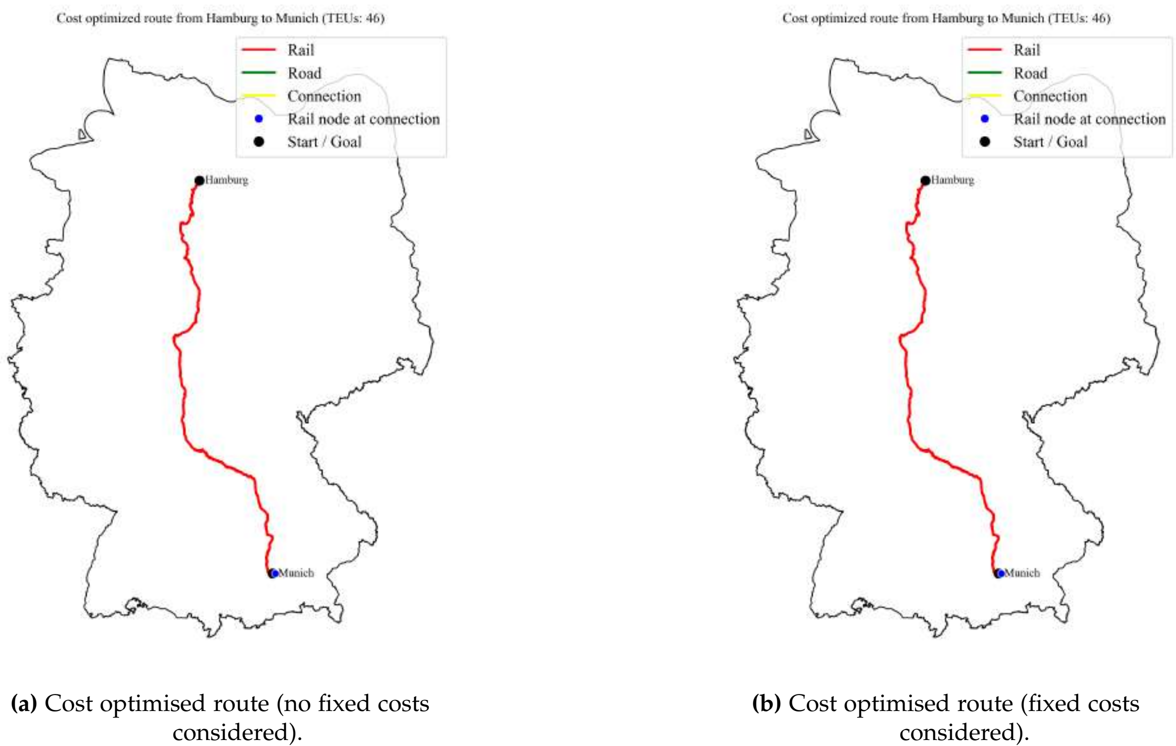
The described pictures (see Fig. 4.9, Fig. 4.10 and Fig. 4.11) clearly show how, despite adding some fixed costs, the algorithm still chooses the same rail route as in the test without fixed costs. This indicates that, if fixed costs are around 2,000 € per train, rail remains a competitively priced transport mode. The analysis draws on data from Flodén, 2011, though the fixed cost used should be understood as an illustrative train-cost calculation. Such fixed costs can depend on numerous parameters that may change according to the country, the rail operator managing the network, the time of year, or even the route segment. Therefore, keeping the scenario characteristics unchanged, several tests are conducted that increase the fixed cost up to 40,000 € to observe any change in the algorithm's behaviour. Initial trials use 5,000 € increments; after that, the breakpoint at which the algorithm's behaviour shifts is identified (in thousands of euros).



(a) Cost optimised route (no fixed costs considered).

(b) Cost optimised route (fixed costs considered).

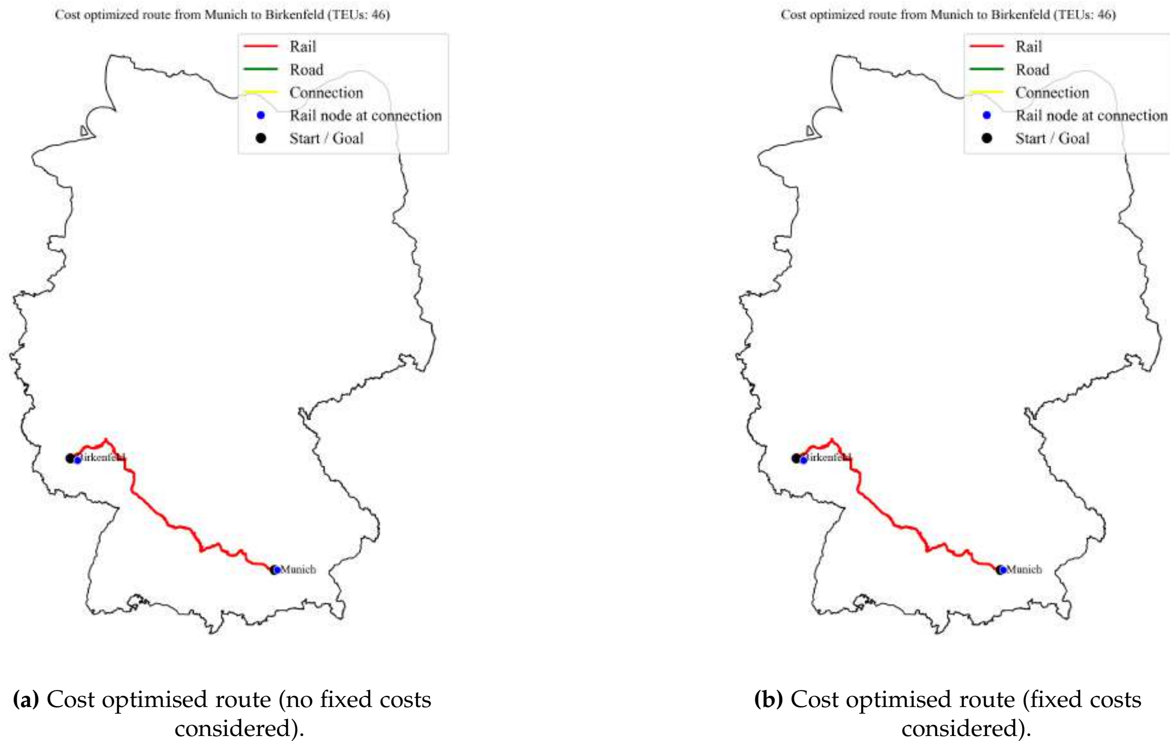
Fig. 4.9: Cost optimisation routes between Berlin and Munich, with and without fixed costs.



(a) Cost optimised route (no fixed costs considered).

(b) Cost optimised route (fixed costs considered).

Fig. 4.10: Cost optimisation routes between Hamburg and Munich, with and without fixed costs.

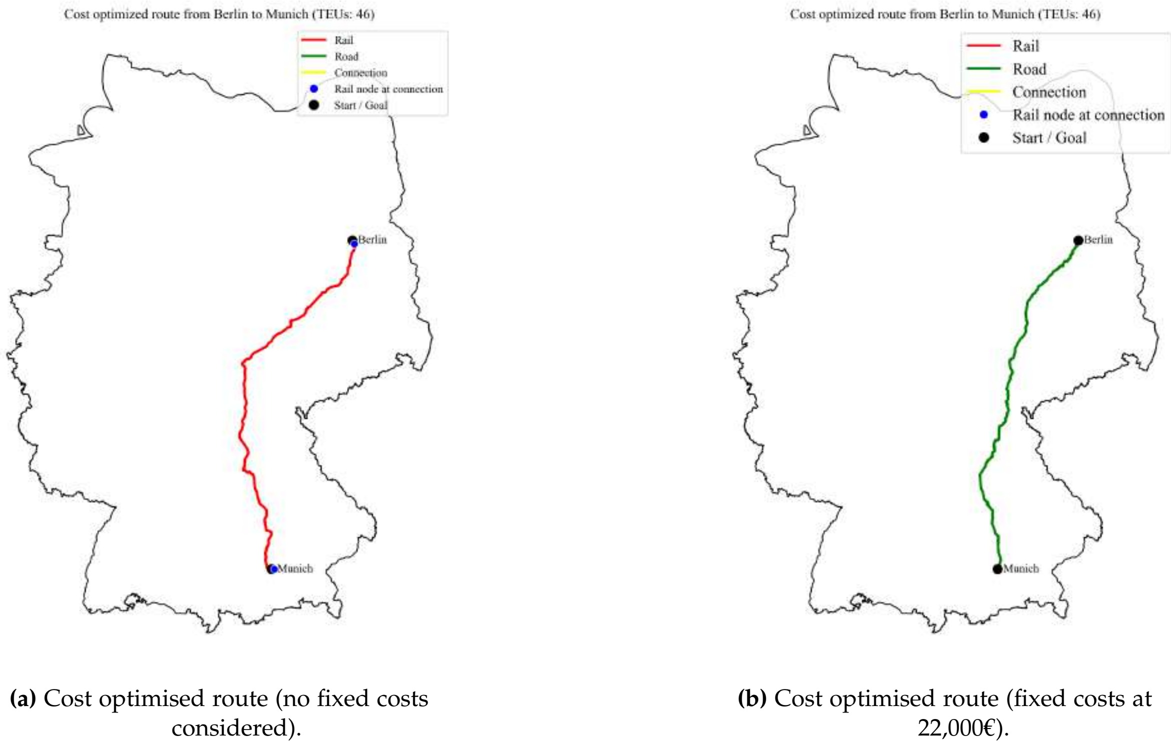


(a) Cost optimised route (no fixed costs considered).

(b) Cost optimised route (fixed costs considered).

Fig. 4.11: Cost optimisation routes between Munich and Birkenfeld, with and without fixed costs.

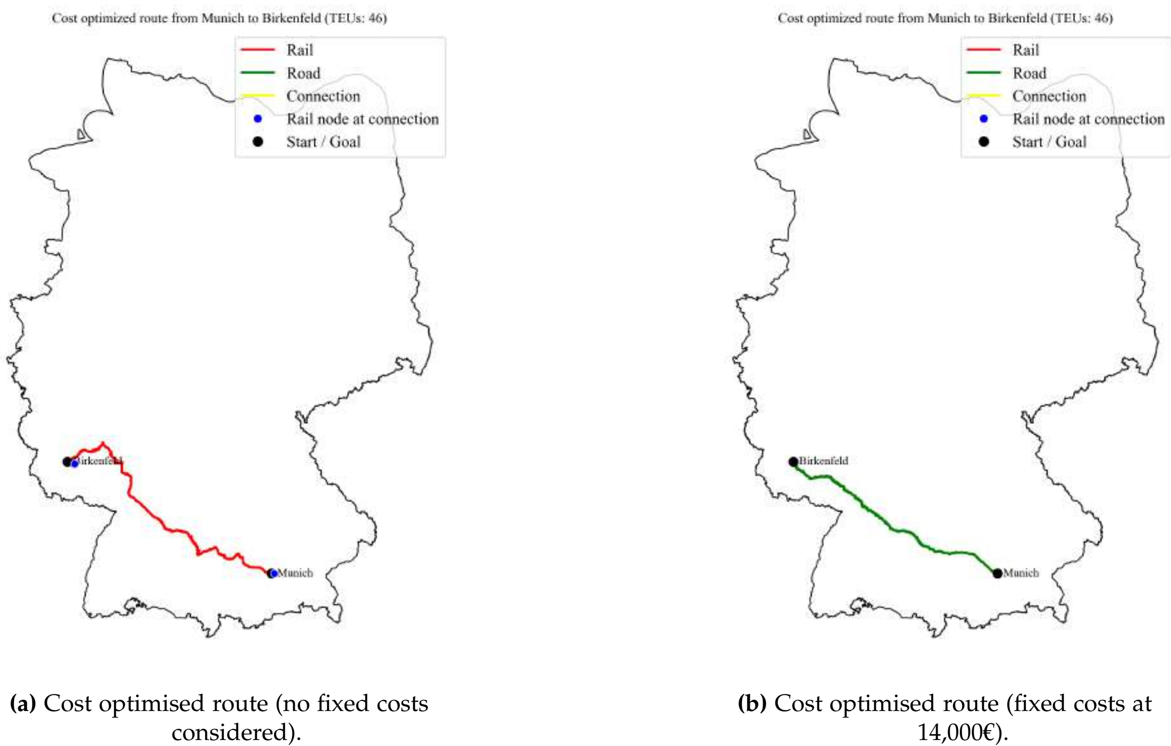
For the Berlin–Munich route, the algorithm chooses a different path once fixed costs exceed 21,000 €. The new path is executed entirely on the road network, from origin to destination (see the comparison in Fig. 4.12). The same behaviour is observed in the Munich–Birkenfeld route, where, from 14,000 € of fixed costs, the path switches to a road-based alternative (see Fig. 4.13b). Finally, for the Hamburg–Munich route, when fixed costs reach 40,000 €, the algorithm still returns the same rail path shown in Fig. 4.10b. This can be explained by the fact that, depending on the locations of the start and end points, the rail network may be easier to access than the road network. Consequently, the algorithm is compelled to select rail as the initial mode, which triggers the fixed charge; it then avoids additional transfers so as not to increase the total cost. Table 4.7 shows how the total cost of the solution route changes according to the fixed cost for rail transport considered.



(a) Cost optimised route (no fixed costs considered).

(b) Cost optimised route (fixed costs at 22,000€).

Fig. 4.12: Route change between Berlin and Munich as a function of fixed costs.



(a) Cost optimised route (no fixed costs considered).

(b) Cost optimised route (fixed costs at 14,000€).

Fig. 4.13: Route change between Munich and Birkenfeld as a function of fixed costs.

Tab. 4.7: Route costs under different rail fixed cost scenarios.

Itinerary	Rail fixed costs (€)	Total route cost (€)	Route composition
Berlin – Munich	0	8,392.28	Road – Rail – Road
Berlin – Munich	2,100	10,492.28	Road – Rail – Road
Berlin – Munich	22,000	21,295.28	Road
Hamburg – Munich	0	5,472.30	Rail – Road
Hamburg – Munich	2,100	7,572.30	Rail – Road
Hamburg – Munich	40,000	45,472.30	Rail – Road
Munich – Birkenfeld	0	7,718.31	Road – Rail – Road
Munich – Birkenfeld	2,100	9,818.31	Road – Rail – Road
Munich – Birkenfeld	14,000	21,295.28	Road

These results show that the tool is able to incorporate variations and adapt to such changes. This adaptability is a strong point for its use when studying the optimal route between two points in terms of costs, but it can also be extended to other scenarios where the goal is to improve travel time or reduce emissions.

5 Conclusions and future research

5.1 Conclusions

The objective of this thesis was to develop a tool with an easy-to-access data source that is simple for users to consult in order to obtain the optimal route within Germany, considering a multimodal network. After the development, validation and analysis of the tool, the conclusions of this project are the following ones.

The OpenStreetMap database is considered convenient for the developed tool. Its open-source policy and wide range of information make it suitable for downloading the road and rail networks in Germany. Despite this, the labels used to describe relevant network elements are sometimes not rigorous, and some assumptions had to be made. Also, Dijkstra's algorithm has been shown to be effective in multimodal networks. A graph was built with both the road and rail networks, establishing points of interest (intermodal stations) to connect these two networks. The edges of the final multimodal graph were assigned different weights according to three criteria: cost, time, and emissions. These values depend on parameters such as the number of TEUs transported along the route (q), the length of the edge (l), and the total tonnes of goods to be transported (w). Dijkstra's algorithm was first validated with a distance-optimisation graph, and it provided multimodal routes as a solution, with its runtime being 35.56 seconds on average. This ensures the tool is easy for users to obtain a possible route between two locations.

Overall, the algorithm tends to yield unimodal routes. This behaviour is likely due to Germany's network layout. The main logistics hubs are well connected in both the road and rail networks, which facilitates the use of only one mode, or the predominant use of one mode over the other. In cost- and emissions-optimisation tests, the tool tends to provide rail-based routes. Prices for rail transportation are relatively lower compared to the road ones, and environmentally, the emissions factor is lower for the railway mode, and the difference between road and rail route lengths is usually not large enough. On the other hand, the time-optimised tests also tend to produce unimodal paths, but the chosen mode is usually the road, even though it sometimes oscillates between road and rail. No clear pattern explains which mode is more likely to be used. Finally, a sensitivity analysis has been conducted to study how the algorithm's behaviour changes if fixed costs are introduced in the rail transport. Results show that, for most of the analysed scenarios, there is a certain value for fixed costs that induces a change from a mainly rail-based route, to an only road route to save costs. Besides this, it also shows how the tool can be adapted to different needs and real-life scenarios.

5.2 Suggestions for further research

While this study has shown that Dijkstra's algorithm can be used on multimodal graphs built with OpenStreetMap (OSM) data, several aspects could be improved. Although the OSM

database is a good source of information, the tagging is sometimes unreliable and assumptions must be made. This may lead to considering unavailable railways as possible paths, or disused or non-freight stations as points of interest due to out-of-date data. Also, this study developed a multimodal route-optimisation tool based on separate criteria. Another interesting approach would be to construct a weighted graph that integrates the three criteria jointly, and to examine whether the algorithm's performance differs from the other tests and whether the multimodal behaviour improves or worsens. Another interesting aspect to be analysed would be to focus on a real-life scenario, with a more defined scope, so that the algorithm uses less generic, more specific data tailored to the case study. Finally, the graph used in every test includes only Germany's road and rail networks. It would be interesting to expand the geographic scope to international freight transport (the European Union or the broader European continent). In this scenario, constraints like border crossings and network-specific regulations should be incorporated, which would result in a more sophisticated and comprehensive optimisation tool. All these aspects could be improved to make the tool more efficient and accurate for optimising multimodal routes.

References

Association for Emissions Control by Catalyst (AECC) (2024): *Heavy-duty Vehicles. Legislative updates*. Latest update: May 2024. URL: https://www.aecc.eu/legislation/heavy-duty-vehicles/ (last access 09/22/2025) (cit. on p. 21).

Autolexicon (2024): *Calculation of CO Emissions*. Accessed: 2025-08-13. URL: https://www.autolexicon.net/en/articles/vypocet-emisi-co2/ (cit. on p. 23).

Barbosa, G., P. J. Pereira, V. Abelha, R. Mendes, P. Cortez (2025): A Dijkstra Seeded Evolutionary Multiobjective Optimization System for a Sustainable User Multimodal Transport Routing. *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO '25)*. New York, NY, USA: Association for Computing Machinery, 1319–1327. ISBN: 979-8-4007-1465-8. DOI: 10.1145/3712256.3726363. URL: https://doi.org/10.1145/3712256.3726363 (cit. on p. 5).

Bender, E. A., S. G. Williamson (2010): *Lists, Decisions and Graphs: With an Introduction to Probability*. San Diego, CA: University of California, San Diego. URL: https://cseweb.ucsd.edu/~gill/BWLectSite/Resources/LDGbookCOV.pdf (last access 08/30/2025) (cit. on pp. 5–7).

Bhattacharya, A., S. A. Kumar, M. Tiwari, S. Talluri (2014): An intermodal freight transport system for optimal supply chain logistics. *Transportation Research Part C* 38, 73–84. DOI: 10.1016/j.trc.2013.10.012 (cit. on p. 4).

Boussaïd, I., J. Lepagnot, P. Siarry (2013): A survey on optimization metaheuristics. *Information Sciences* 237, 82–117. DOI: 10.1016/j.ins.2013.02.041 (cit. on p. 10).

Burdzik, R., M. Cieśla, A. Sładkowski (2014): Cargo Loading and Unloading Efficiency Analysis in Multimodal Transport. *Promet - Traffic & Transportation* 26 (4), 323–331. DOI: 10.7307/ptt.v26i4.1356. URL: https://www.researchgate.net/publication/269786800 (cit. on p. 19).

Chen, X., X. Hu, H. Liu (2024): Low-carbon route optimization model for multimodal freight transport considering value and time attributes. *Socio-Economic Planning Sciences* 96, 102108. DOI: 10.1016/j.seps.2024.102108 (cit. on p. 5).

Council of the European Union (2024): *Fit for 55*. URL: https://www.consilium.europa.eu/en/policies/fit-for-55/ (last access 09/21/2025) (cit. on p. 1).

DHL Freight Connections (2023): *Truck Speed Limits in Europe*. Accessed: 2025-08-12. URL: https://dhl-freight-connections.com/en/business/truck-speed-limits-europe/ (cit. on pp. 20, 22).

Diestel, R. (2000): *Graph Theory*. 2nd ed. Vol. 173. Graduate Texts in Mathematics. Electronic edition. New York: Springer-Verlag. URL: https://www.emis.de/monographs/Diestel/en/GraphTheoryII.pdf (last access 08/30/2025) (cit. on pp. 6, 7).

Escola Europea (2024): *On Track for the Future: Rail Freight in Europe (2024)*. Accessed: 2025-08-13. URL: https://escolaeuropea.eu/did-you-know/on-track-for-the-future-rail-freight-in-europe-2024/ (cit. on p. 23).

European Automobile Manufacturers' Association (ACEA) (2020): *CO Emissions from Heavy-Duty Vehicles – Preliminary CO Baseline (Q3–Q4 2019 Estimate)*. *techreport*. Accessed: 2025-08-13. ACEA. URL: https://www.acea.auto/files/ACEA_preliminary_CO2_baseline_heavy-duty_vehicles.pdf (cit. on p. 22).

European Commission (2023): *Green Deal: Greening freight for more economic gain with less environmental impact**. URL: https://ec.europa.eu/commission/presscorner/detail/en/ip_23_3767 (last access 05/16/2025) (cit. on p. 1).

European Commission (2025): *Trans-European Transport Network (TEN-T)*. URL: https://transport.ec.europa.eu/transport-themes/infrastructure-and-investment/trans-european-transport-network-ten-t_en (last access 08/22/2025) (cit. on p. 1).

European Environment Agency (2024): *Sustainability of Europe's mobility systems. Freight transport activity*. URL: https://www.eea.europa.eu/en/analysis/publications/sustainability-of-europes-mobility-systems/freight-transport-activity (last access 05/16/2025) (cit. on p. 1).

European Environment Agengy (2023): *Reducing greenhouse gas emissions from heavy-duty vehicles in Europe*. URL: https://www.eea.europa.eu/publications/co2-emissions-of-new-heavy (last access 05/09/2025) (cit. on p. 1).

Eurostat (2024): *Road freight transport by vehicle characteristics*. Statistics Explained article. Data extracted in September 2024. URL: https://ec.europa.eu/eurostat/statistics-explained/index.php?title=Road_freight_transport_by_vehicle_characteristics (last access 09/22/2025) (cit. on p. 21).

Eurowag (2023): *Rules for Truck Drivers in Germany*. Accessed: 2025-08-12. URL: https://www.eurowag.com/blog/rules-for-truck-drivers-in-germany (cit. on p. 19).

Flodén, J. (2011): *Rail Freight Costs: Some Basic Cost Estimates for Intermodal Transport*. Project "Strategic modelling of combined transport between road and rail in Sweden". Gothenburg, Sweden: University of Gothenburg, School of Business, Economics and Law, Department of Business Administration. URL: https://gupea.ub.gu.se/bitstream/handle/2077/25877/gupea_2077_25877_3.pdf?sequence=3 (last access 09/24/2025) (cit. on p. 41).

German Council of Economic Experts, French Council of Economic Analysis (2025): *Join statement: Decarbonising road freight transport*. *Policy Report*. Sachverständigenrat zur Begutachtung der gesamtwirtschaftlichen Entwicklung (GCEE) and Conseil d'analyse économique (CAE). URL: https://www.sachverstaendigenrat-wirtschaft.de/fileadmin/dateiablage/Publikationen/FGCEE/CAE_FGCEE_Joint_statement_250320.pdf (last access 05/16/2025) (cit. on p. 1).

Guo, J., T. Liu, G. Song, B. Guo (2024): Solving the Robust Shortest Path Problem with Multimodal Transportation. *Mathematics* 12 (19), 2978. DOI: 10.3390/math12192978. URL: https://doi.org/10.3390/math12192978 (cit. on p. 5).

Hintjens, J., E. van Hassel, T. Vanelslander, E. Van de Voorde (2020): Port Cooperation and Bundling: A Way to Reduce the External Costs of Hinterland Transport. *Sustainability* 12

(23), 9983. DOI: 10.3390/su12239983. URL: https://www.mdpi.com/2071-1050/12/23/9983 (cit. on pp. 21, 22).

Javaid, M. A. (2013): *Understanding Dijkstra's Alogirthm*. *techreport* 2340905. CompTIA. URL: https://papers.ssrn.com/sol3/papers.cfm?abstract_id=2340905 (cit. on p. 10).

Kaewfak, K., V. Ammarapala, V.-N. Huynh (2021): Multi-objective Optimization of Freight Route Choices in Multimodal Transportation. *International Journal of Computational Intelligence Systems* 14 (1), 794–807. DOI: 10.2991/ijcis.d.210126.001 (cit. on p. 4).

Laporte, G. (2009): Fifty Years of Vehicle Routing. In: *Handbooks in Operations Research and Management Science*. Ed. by B. L. Golden, S. Raghavan, E. A. Wasil. Vol. 14. Elsevier, 11–36. DOI: 10.1016/S0927-0507(09)14002-7 (cit. on p. 10).

Lusiani, A., E. Sartika, A. Binarto, E. Habinuddin, I. Azis (2021): Determination of the Fastest Path on Logistics Distribution by Using Dijkstra Algorithm. *Proceedings of the 2nd International Seminar of Science and Applied Technology (ISSAT 2021)*. Vol. 207. Advances in Engineering Research. Atlantis Press International B.V., 246–250 (cit. on p. 4).

METRANS (2024): *Development of Rail Transport Prices in Germany*. Accessed: 2025-08-12. URL: https://metrans.eu/development-of-rail-transport-prices-in-germany/ (cit. on pp. 20, 41).

mtonroad.com (2024): *Transportpreise pro km – Was kostet ein Transport?* Accessed: 2025-08-12. URL: https://mtonroad.com/news/transportpreise-pro-km (cit. on p. 21).

NextBillion.ai (2023): *What is a Route Optimization Algorithm?* Accessed: 2025-08-17. URL: https://nextbillion.ai/blog/what-is-route-optimization-algorithm (cit. on p. 10).

OpenStreetMap contributors (2025): *OpenStreetMap*. URL: https://www.openstreetmap.org/#map=5/51.32/4.66 (last access 09/29/2025) (cit. on p. 7).

OpenStreetMap Wiki (2025a): *Key: building*. OpenStreetMap Wiki. URL: https://wiki.openstreetmap.org/wiki/Key:building (last access 09/01/2025) (cit. on p. 17).

OpenStreetMap Wiki (2025b): *Tag: railway=station*. OpenStreetMap Wiki. URL: https://wiki.openstreetmap.org/wiki/Tag:railway%3Dstation (last access 09/01/2025) (cit. on p. 17).

OpenStreetMap Wiki (2025c): *Tag: railway=yard*. OpenStreetMap Wiki. URL: https://wiki.openstreetmap.org/wiki/Tag:railway%3Dyard (last access 09/01/2025) (cit. on p. 17).

OpenStreetMap Wiki contributors (2025a): *Elements*. URL: https://wiki.openstreetmap.org/wiki/Elements (last access 08/30/2025) (cit. on p. 8).

OpenStreetMap Wiki contributors (2025b): *OpenStreetMap Wiki*. Community-maintained documentation. URL: https://wiki.openstreetmap.org/ (last access 08/30/2025) (cit. on p. 8).

Pedersen, M. B. (2005): Optimization models and solution methods for intermodal transportation. *PhD thesis*. Technical University of Denmark (cit. on p. 4).

Shapely developers (2025): *Shapely Manual: object.representative_point*. https://shapely.readthedocs.io/en/2.1.1/manual.html#object.representative_point. Accessed 29 Sep 2025; version 2.1.1 documentation (cit. on p. 17).

Wang, C.-N., T.-T. Dang, T. Q. Le, P. Kewcharoenwong (2020): Transportation Optimization Models for Intermodal Networks with Fuzzy Node Capacity, Detour Factor, and Vehicle Utilization Constraints. *Mathematics* 8 (12), 2109. DOI: 10.3390/math8122109 (cit. on p. 5).

Wong, M. (2019): *How long does it take to unload a container train?* Accessed: 2025-08-12. URL: https://wongm.com/2019/08/how-long-does-it-take-to-unload-a-container-train/ (cit. on p. 20).

A Konecranes SMV 4123 CC5 Datasheet

REACH STACKER SMV 4123 CC5

LIFTING DATA	
Lift capacities (at load center LC1 / LC2 / LC3)	41000 kg / 23000 kg / 9000 kg
Lift capacities (at load center LC1 / LC2 / LC3 at max lifting height)	41000 kg / 23000 kg / 9000 kg
Lift capacities in trailer legs (at load center LC1)	41000 kg
Load centers (LC1 / LC2 / LC3)	2000 mm / 3850 mm / 6350 mm
Lifting speed, unloaded / at 40 % load / at rated load	0.38 m/s / 0.35 m/s / 0.23 m/s
Lowering speed, unloaded / at rated load	0.35 m/s / 0.40 m/s

DRIVING DATA	
Drive speed forward, unloaded / at rated load	25 km/h / 22 km/h
Drive speed reverse, unloaded / at rated load	25 km/h / 22 km/h
Incline (driving ability) at rated load at 0 km/h / 2 km/h	25 % / 20 %
Towing (power ability) at rated load at 0 km/h / 2 km/h	300 kN / 246 kN

SERVICE WEIGHT / AXLE PRESSURE	
Service weight	72500 kg
Axle pressure front at load center LC1, unloaded / at rated load	43000 kg / 101900 kg
Axle pressure front at load center LC2, unloaded / at rated load	48700 kg / 88500 kg
Axle pressure rear at load center LC1, unloaded / at rated load	29500 kg / 11600 kg
Axle pressure rear at load center LC2, unloaded / at rated load	23800 kg / 7000 kg
Axle pressure in driving position, front/rear (at rated load)	93600 kg / 19900 kg

ENGINE (ELECTRONIC CONTROLLED)			
Engine make / model name	Volvo TAD-1181-VE	Torque ISO 3046 / at speed	1785 Nm / 1400 rpm
Emission approval EU / US	EU 5	Displacement / No. of cyl. / type	11 l / 6 / Inline
Monitoring / EC / CanBus	Yes / Yes / Yes	Fuel consumption, normal usage	15 - 18 l/h
Fuel / type of engine / intercooler	Diesel / 4-stroke / Yes	Alternator, type/power / capacity	AC / 3080 W / 110 A
Power ISO 3046 / max speed	265 kW / 2100 rpm	Start battery, voltage/capacity	2 x 12 V / 140 Ah

TRANSMISSION (ELECTRONIC CONTROLLED)	
Transmission make / model / amount of gears (forward /reverse)	Dana TE-30 / 5 + 3 gears
Monitoring / reverse protection / CanBus / Clutch, type	Yes / Yes / Yes / Torque converter
Transmission type / type of shift gear	Softshift – Powershift / Automatic

STEERING AXLE & STEERSYSTEM & DRIVE AXLE & BRAKE SYSTEM	
Steer axle type / steering system	Double acting cylinder / hydraulic servo assisted
Drive axle make / model name	Kessler D102
Drive axle type / drive axle width	Differential + hub reduction / 4160 mm
Driving brake system, type / affected wheels	Oil cooled Wet Disc Brakes / drive wheels
Parking brake system, type / affected wheels	Dry disc brake / spring release / drive wheels

WHEELS	
Number of wheels, front + rear / type (*driven)	4*+2 / pneumatic
Tire pressure front / rear	1.00 MPa / 1.00 MPa
Tire dimensions (Ply rating) front / rear	18.00 x 25" (PR 40) / 18.00 x 25" (PR 40)
Rim dimension front / rear	13.00 x 25" / 13.00 x 25"

HYDRAULIC SYSTEM	
Hydraulic pump make / model name	Parker & Hannifin / P2-series
Hydraulic system / pump type	Load sensing system / variable piston pump
Power-on-demand / Low energy / Separate oil tanks	Yes / Yes / Yes (hydraulics & brakes)
Hydraulic oil pressure mast / spreader	24 MPa / 15 MPa

TANK VOLUMES			
Diesel tank volume	650 l	Hydraulic tank volume	850 l

LIFTING EQUIPMENT & SPREADER	
Spreader make / type	ELME 857-PPS / Combilift (container & trailer)
Spreader functions	20-30-35-40 ft containers / trailer bottom lift
Spreader locking containers / lifting eyes	4 top twist locks + 4 lift legs-shoes / 4 corners
Sideshift stroke / rotation / pile slope type (pile slope stroke)	±800 mm / +195 mm : -105 deg / Hydraulic PPS (±6 deg)

NORMS & STANDARDS		OTHERS	
Machine Directive in Europe	2006/42/EC	Noise level (inside Cab/ LpAZ)	EN 12053 75 dB(A)
Stability of Industrial Trucks	ISO22915 series / EN15000	Noise level (outside at 7 m/ Lwa)	2000/14/EC 111 dB(A)
Safety of Industrial trucks	EN ISO 3691-2	Safety, Monitoring & overload system / programmable / colour	Electronic system EMC / Yes / Yes
Noise level (inside Cab / Lm)	DIN 45635 68 dB(A)	Manual sliding cabin / stroke	Yes / (Stepless)

2020-8

B OSM road network extraction code: `overpass_map_road.py`

Code B.1: OSM Overpass rail extraction and map for Germany (01_overpass_map_road.py).

```
import requests
import geopandas as gpd
import matplotlib.pyplot as plt
import pyproj
import json
from shapely.geometry import LineString
from pathlib import Path
from shapely.ops import transform

# Inputs:
BASE_DIR = Path(__file__).resolve().parent
OUTPUT_DIR = BASE_DIR / "input_files"


##############################
# 1) Query road network in Germany (motorways, trunk, primary and secondary roads)
##############################
def query_highways():
    road_query = """
    [out:json][timeout:1800];
    area["name"="Deutschland"]->.searchArea;
    (
      way["highway"="motorway"](area.searchArea);
      way["highway"="motorway_link"](area.searchArea);
      way["highway"="trunk"](area.searchArea);
      way["highway"="trunk_link"](area.searchArea);
      way["highway"="primary"](area.searchArea);
      way["highway"="primary_link"](area.searchArea);
       way["highway"="secondary"](area.searchArea);
      way["highway"="secondary_link"](area.searchArea);
    );
    out body;
    >;
    out skel qt;
    """
    response = requests.post("https://overpass-api.de/api/interpreter",
    data={"data": road_query})
    return response.json()


def parse_road_data(data):
    nodes = {e["id"]: (e["lon"], e["lat"]) for e in data["elements"] if e["type"]
    == "node"}
    features = []

    for e in data["elements"]:
        if e["type"] == "way":
```

```python
            coords = [nodes[n] for n in e["nodes"] if n in nodes]
            if len(coords) >= 2:
                tags = e.get("tags", {})
                features.append({
                    "geometry": LineString(coords),
                    "highway": tags.get("highway"),
                    "ref": tags.get("ref"),
                    "name": tags.get("name")
                })
    return features


##############################
# 2) Query German border
##############################
def query_germany_border():
    border_query = """
    [out:json];
    rel["name"="Deutschland"]["admin_level"="2"];
    out body;
    >;
    out skel qt;
    """
    response = requests.post("https://overpass-api.de/api/interpreter", data={"data": border_query})
    return response.json()


def parse_border_data(data):
    nodes = {e["id"]: (e["lon"], e["lat"]) for e in data["elements"] if e["type"] == "node"}
    border_lines = []

    for way in data["elements"]:
        if way["type"] == "way" and "nodes" in way and len(way["nodes"]) >= 2:
            coords = [nodes[n] for n in way["nodes"] if n in nodes]
            if len(coords) >= 2:
                border_lines.append({"geometry": LineString(coords)})

    return gpd.GeoDataFrame(border_lines, crs="EPSG:4326")


##############################
# 3) Save edges to JSON with length in meters
##############################
def save_edges_to_json(features, filename):
    project = pyproj.Transformer.from_crs("EPSG:4326", "EPSG:32633", always_xy=True).transform
    edges_data = []

    for feature in features:
        geometry = transform(project, feature["geometry"])
        length = geometry.length
        edges_data.append({
            "geometry": list(feature["geometry"].coords),
            "highway": feature["highway"],
            "ref": feature.get("ref"),
            "name": feature.get("name"),
            "length": length
        })

```

```
104     with open(filename, "w") as f:
105         json.dump(edges_data, f, indent=4)
106 
107 
108 ##############################
109 # 4) Plot Map
110 ##############################
111 def plot_map(gdf_border, gdf_autobahn, output_path="germany_overpass_map.png"):
112     fig, ax = plt.subplots(figsize=(12, 12))
113 
114     gdf_border.plot(ax=ax, edgecolor="black", linewidth=2, label="German Border")
115     gdf_autobahn.plot(ax=ax, color="red", linewidth=1, label="Autobahn (motorway +
    link)")
116 
117     plt.title("German Road Network: motorways, trunks, primary and secondary roads")
118     plt.axis("off")
119     plt.legend()
120     fig.savefig(output_path, dpi=300, bbox_inches="tight")
121     plt.show()
122 
123 
124 ##############################
125 # Main
126 ##############################
127 if __name__ == "__main__":
128     # 1) Query and process highway data
129     road_data = query_highways()
130     highway_features = parse_road_data(road_data)
131 
132     # 2) Filter Autobahns
133     autobahn_features = [f for f in highway_features if f["highway"] in
    ["motorway", "motorway_link","trunk","trunk_link",
134 
    "primary","primary_link","secondary","secondary_link"]]
135     gdf_autobahn = gpd.GeoDataFrame(autobahn_features, crs="EPSG:4326")
136 
137     # 3) Query and process border
138     border_data = query_germany_border()
139     gdf_border = parse_border_data(border_data)
140 
141     # 4) Save to JSON with lengths
142     save_edges_to_json(autobahn_features, OUTPUT_DIR /
    "germany_all_road_edges.json")
143 
144     # 5) Plot the result
145     plot_map(gdf_border, gdf_autobahn)
```

C OSM rail network extraction code: `overpass_map_rail.py`

Code C.1: OSM Overpass rail extraction and map for Germany (01_overpass_map_rail.py).

```
import requests
import geopandas as gpd
from shapely.geometry import LineString
from pathlib import Path

import matplotlib.pyplot as plt
import pyproj
from shapely.ops import transform
import json
import datetime


# Inputs:
BASE_DIR = Path(__file__).resolve().parent
OUTPUT_DIR = BASE_DIR / "input_files"


##############################
# Auxiliar Functions
##############################

# Query railways in Germany
# ----------------------------
def query_railways(kinds=("rail",), include_service=False):
    kinds_re = "|".join(kinds)
    service_filter = "" if include_service else '["service"!~"."]'
    rail_query = f"""
    [out:json][timeout:1800];
    area["ISO3166-1"="DE"]->.searchArea;
    (
      way["railway"~"^({kinds_re})$"]{service_filter}(area.searchArea);
    );
    out body;
    >;
    out skel qt;
    """
    r = requests.post("https://overpass-api.de/api/interpreter", data={"data":
    rail_query})
    r.raise_for_status()
    return r.json()


def parse_rail_data(data):
    nodes = {e["id"]: (e["lon"], e["lat"]) for e in data["elements"] if e["type"]
    == "node"}
    features = []
    for e in data["elements"]:
```

```
46          if e["type"] == "way" and "nodes" in e:
47              coords = [nodes[n] for n in e["nodes"] if n in nodes]
48              if len(coords) >= 2:
49                  tags = e.get("tags", {})
50                  features.append({
51                      "geometry": LineString(coords),
52                      "railway": tags.get("railway"),     # rail, light_rail, subway,
    tram,
53                      "ref": tags.get("ref"),
54                      "name": tags.get("name")
55                  })
56      return features
57
58
59
60  # Query Germany Border
61  # ---------------------------
62  def query_germany_border():
63      border_query = """
64      [out:json];
65      rel["name"="Deutschland"]["admin_level"="2"];
66      out body;
67      >;
68      out skel qt;
69      """
70      response = requests.post("https://overpass-api.de/api/interpreter",
    data={"data": border_query})
71      return response.json()
72
73
74  def parse_border_data(data):
75      nodes = {e["id"]: (e["lon"], e["lat"]) for e in data["elements"] if e["type"]
    == "node"}
76      border_lines = []
77
78      for way in data["elements"]:
79          if way["type"] == "way" and "nodes" in way and len(way["nodes"]) >= 2:
80              coords = [nodes[n] for n in way["nodes"] if n in nodes]
81              if len(coords) >= 2:
82                  border_lines.append({"geometry": LineString(coords)})
83
84      return gpd.GeoDataFrame(border_lines, crs="EPSG:4326")
85
86
87
88  # Save edges to JSON with length in meters
89  # ---------------------------
90  def save_rail_edges_to_json(features, filename):
91      geod = pyproj.Geod(ellps="WGS84")
92      edges_data = []
93      for f in features:
94          length_m = geod.geometry_length(f["geometry"])
95          edges_data.append({
96              "geometry": list(f["geometry"].coords),  # [lon, lat]
97              "railway": f["railway"],
98              "ref": f.get("ref"),
99              "name": f.get("name"),
100             "length": length_m
101         })
102     with open(filename, "w") as fh:
103         json.dump(edges_data, fh, indent=4)
```

```
104
105
106
107 # Plot Map functions
108 # ----------------------------
109 def plot_map(gdf_border, gdf_rail, output_path="germany_rail_map.png"):
110     fig, ax = plt.subplots(figsize=(12, 12))
111     gdf_border.plot(ax=ax, edgecolor="black", linewidth=2, label="German Border")
112     gdf_rail.plot(ax=ax, color="green", linewidth=0.7, label="Rail")
113     plt.title("Germany Rail Network")
114     plt.axis("off")
115     plt.legend()
116     fig.savefig(output_path, dpi=300, bbox_inches="tight")
117     plt.show()
118
119
120 ##############################
121 # Main Workflow
122 ##############################
123 if __name__ == "__main__":
124     # 1) Fetch and parse rails
125     rail_data = query_railways(kinds=("rail",), include_service=False)
126     rail_features = parse_rail_data(rail_data)
127     gdf_rail = gpd.GeoDataFrame(rail_features, crs="EPSG:4326")
128
129     # 2) Border
130     border_data = query_germany_border()
131     gdf_border = parse_border_data(border_data)
132
133     # 3) Save JSON
134     save_rail_edges_to_json(rail_features, OUTPUT_DIR)
135
136     # 4) Plot
137     plot_map(gdf_border, gdf_rail)
```

D OSM points of interest extraction code: `pois_extraction.py`

Code D.1: OSM Overpass rail extraction and map for Germany (01_pois_extraction.py).

```
import osmnx as ox
import networkx as nx
import matplotlib.pyplot as plt
import time
import geopandas as gpd
from shapely.geometry import Point
from geopy.distance import geodesic
import openpyxl

#################################################################
#Creation of a Bounding box that includes all Germany:
germany = ox.geocode_to_gdf("Germany")
west, south, east, north = germany.total_bounds #
print(west, south, east, north)

#################################################################
#GET POI (Points of Interest)
tags = {
    "railway": ["station", "yard","container_terminal"],
    "building": ["station"]
}
non_filtered_pois = ox.features_from_bbox((west, south, east, north), tags)

# List of columns tagged as non used terminals:
del_col = []
for col in non_filtered_pois.columns:
    if 'abandoned' in col:
        if col not in del_col:
            del_col.append(col)
    elif 'disused' in col:
        if col not in del_col:
            del_col.append(col)

# 1rst Filter --> delete non-used/abandoned terminals
mask= non_filtered_pois[del_col].isna().all(axis=1)
filtered_pois= non_filtered_pois[mask].copy()

print(f"Kept {len(filtered_pois)} non-abandoned POIs out of
    {len(non_filtered_pois)} total")
print(f'Deleted POIs: ' + str(len(non_filtered_pois)-len(filtered_pois)))


# 2nd Filter --> delete public_transport terminals
pois = filtered_pois[filtered_pois["public_transport"].isna() |
    (filtered_pois["public_transport"] == "no")]
print(f"Kept {len(pois)} POIs out of {len(filtered_pois)} total")
print(f'Deleted POIs: ' + str(len(filtered_pois)-len(pois)))
```

```

# Only keep the relevant columns:
relevant_columns = ['railway','building','geometry','name']
pois = pois[relevant_columns]

print('POIs geometries count: ')
counts = pois.geometry.geom_type.value_counts()
print(counts)

# Changing all geometries to equivalent points:
pois['geometry'] = pois.geometry.apply(lambda geom: geom
                 if geom.geom_type == 'Point'
                 else geom.representative_point())

print('FINAL POIs geometries count: ')
counts = pois.geometry.geom_type.value_counts()
print(counts)

# Determine nature of pois
# A) POIs tagged as yards
pois_yard = pois[pois["railway"] == 'yard']

# B) POIs tagged as stations (already filtered to be non-public-transport related
pois_stations = pois[pois["railway"] == "station"]

# C) POIs tagged as container_stations
pois_cterminals = pois[pois["railway"] == "container_terminal"]

# D) POIs tagged as building:station
pois_building = pois[pois["building"] == "station"]

# Possible intersections:
common_idx_1 = pois_stations.index.intersection(pois_yard.index)
common_idx_2 = pois_stations.index.intersection(pois_cterminals.index)
common_idx_3 = pois_yard.index.intersection(pois_cterminals.index)
common_idx_4 = pois_yard.index.intersection(common_idx_2)

print(f" Intersection count (station  yard): {len(common_idx_1)}")
print(f" Intersection count (station  container_terminal): {len(common_idx_2)}")
print(f" Intersection count (container_terminal  yard): {len(common_idx_3)}")
print(f" Intersection count (container_terminal  yard  station):
    {len(common_idx_4)}")

# 5) Verification
print('Total number of POIs:',len(pois))
print(" Stations:", len(pois_stations))
print(" Yard:", len(pois_yard))
print(" Container_terminal:", len(pois_cterminals))
print(" Building:station:", len(pois_building))

# Save POIs to gpkg to work with it any time [elements structure and functions is
    preserved]
pois.to_file("final_pois.gpkg", layer="rail_pois", driver="GPKG")
```

E Weighted graph construction code: weighted_graph.py

E.1 Main script: weighted_graph.py

Code E.1: Weighted graph construction code (weighted_graph.py).

```
import json
from pathlib import Path
from statistics import mean
import csv
import math
import geopandas as gpd
from shapely.geometry import LineString
import datetime
from collections import defaultdict

from functions_weighted_graph import *

##############################
# inputs
##############################
UPDATE = 1
BASE_DIR = Path(__file__).resolve().parent

ROAD_JSON = BASE_DIR / 'input_files' / "germany_all_road_edges.json"
RAIL_JSON = BASE_DIR / 'input_files' / "germany_rail_edges.json"
POIS_GPKG = BASE_DIR / 'input_files' / "final_pois.gpkg"

Q_TEUS = 46.0   # number of TEUs

OUTPUT_DIR = BASE_DIR / 'graphs'
OUTPUT_FILENAME = f"graph_q_{int(Q_TEUS)}.json"
OUTPUT_EDGES_JSON = OUTPUT_DIR / OUTPUT_FILENAME
INPUT_EDGES_JSON = BASE_DIR / 'graphs' / 'graph_q_20.json'

EXPORT_CSV = True
OUTPUT_EDGES_CSV = OUTPUT_EDGES_JSON.with_suffix(".csv")

##############################
# parameters (only informative,
# in case of value change, change in functions_weighted_graph.py)
##############################
KMH_MOTORWAY = 80.0
KMH_SECONDARY = 60.0
KMH_RAIL = 55.0

G_PER_TKM_ROAD = 64.7*2
G_PER_TKM_RAIL = 24.0
TONS_PER_TEU = 11.5
G_PER_HOUR_TRANSFER = 42240.0
```

```
45
46 SECONDS_PER_TEU_CONNECTION = 100.0
47 MINUTES_FIXED_CONNECTION = 111.0
48
49 # Costs /kmTEU
50 ROAD_COST_PER_KM_PER_TEU = (1.99 + 0.19)/2
51 RAIL_COST_PER_KM = 3.74
52 CONNECTION_COST_PER_TEU = 50.0
53
54
55
56 ##############################
57 # Main
58 ##############################
59 if __name__ == "__main__":
60     if UPDATE == 0:
61         print('Building new graph...')
62         road_edges = load_edges(ROAD_JSON)
63         rail_edges = load_edges(RAIL_JSON)
64         road_graph = build_graph_with_mode(road_edges, mode="road")
65         rail_graph = build_graph_with_mode(rail_edges, mode="rail")
66         graph = merge_graphs(road_graph, rail_graph)
67         added, skipped, pairs = add_poi_connections(
68             graph,
69             road_graph,
70             rail_graph,
71             pois_gpkg=POIS_GPKG,
72             q_teus=Q_TEUS
73         )
74         print("Connections added:", added, "skipped:", skipped)
75
76     else:
77         print('Loading existing graph and refreshing connection lengths...')
78         with open(INPUT_EDGES_JSON, "r", encoding="utf-8") as f:
79             data = json.load(f)
80         graph = defaultdict(list)
81         for e in data:
82             u = (e["u"]["mode"], float(e["u"]["lon"]), float(e["u"]["lat"]))
83             v = (e["v"]["mode"], float(e["v"]["lon"]), float(e["v"]["lat"]))
84             w = float(e["length_m"]); m = e["mode"]
85             if m == "connection":
86                 w = (1.85 + 0.028 * Q_TEUS) * 80000
87             graph[u].append((v, w, m))
88             graph[v].append((u, w, m))
89
90     # Unique edges and weight calculation
91     edges = []
92     seen = set()
93     for u, nbrs in graph.items():
94         for v, length_m, mode in nbrs:
95             key = frozenset((u, v))
96             if key in seen:
97                 continue
98             seen.add(key)
99             road_mode = ""
100            e_out = {
101                "u": {"mode": u[0], "lon": u[1], "lat": u[2]},
102                "v": {"mode": v[0], "lon": v[1], "lat": v[2]},
103                "length_m": float(length_m),
104                "mode": mode
105            }
```

```
106                e_out["time_h"]        = compute_time_hours(length_m, mode, Q_TEUS,
        road_mode)
107                e_out["emissions_g"]   = compute_emissions_grams(length_m, mode, Q_TEUS)
108                e_out["cost_eur"]      = compute_cost_eur(length_m, mode, Q_TEUS)
109                edges.append(e_out)
110
111        OUTPUT_DIR.mkdir(parents=True, exist_ok=True)
112        save_edges_json(OUTPUT_EDGES_JSON, edges)
113        if EXPORT_CSV:
114            save_edges_csv(OUTPUT_EDGES_CSV, edges)
115        print(f"Graph saved to {OUTPUT_EDGES_JSON} with {len(edges)} edges.")
116        summarize_edges(edges)
```

E.2 Auxiliar functions script: functions_weighted_graph.py

Code E.2: Weighted graph auxiliar functions code (functions_weighted_graph.py).

```
1   import json
2   from pathlib import Path
3   from statistics import mean
4   import csv
5   import math
6   import geopandas as gpd
7   from shapely.geometry import LineString
8   import datetime
9   from collections import defaultdict
10
11
12  # ----------------------------
13  # parameters
14  # ----------------------------
15  KMH_MOTORWAY = 80
16  KMH_SECONDARY = 60
17  KMH_RAIL = 55.0
18
19  G_PER_TKM_ROAD = 64.7*2
20  G_PER_TKM_RAIL = 24.0
21  TONS_PER_TEU = 11.5
22  G_PER_HOUR_TRANSFER = 4224.0
23
24  SECONDS_PER_TEU_CONNECTION = 100.0
25  MINUTES_FIXED_CONNECTION = 111.0
26
27  # Costs /kmTEU
28  ROAD_COST_PER_KM_PER_TEU = (1.99 + 0.19)/2
29  RAIL_COST_PER_KM = 3.74
30  CONNECTION_COST_PER_TEU = 50.0
31
32
33  # ----------------------------
34  # Helpers
35  # ----------------------------
36  def load_edges(file_path):
37      with open(file_path, 'r', encoding='utf-8') as file:
38          return json.load(file)
39
40  def haversine_m(p1, p2):
41      lon1, lat1 = map(math.radians, p1)
```

```
42     lon2, lat2 = map(math.radians, p2)
43     dlon = lon2 - lon1
44     dlat = lat2 - lat1
45     a = math.sin(dlat / 2) ** 2 + math.cos(lat1) * math.cos(lat2) * math.sin(dlon /
       2) ** 2
46     c = 2 * math.atan2(math.sqrt(a), math.sqrt(1 - a))
47     return 6371000.0 * c  # meters
48
49 def haversine_km(p1, p2):
50     return haversine_m(p1, p2) / 1000.0
51
52 def node_lonlat(n):
53     # n = (mode, lon, lat)
54     return (n[1], n[2])
55
56 # Node structure (mode, lon lat)
57 # graph[node]: [(neighbor_1, length, mode), (neighbor_2, length, mode)...]
58 def build_graph_with_mode(edges, mode):
59     dedup = {}
60
61     for edge in edges:
62         coords = edge.get("geometry")
63         if not coords or len(coords) < 2:
64             continue
65
66         oneway_val = edge.get("oneway")
67         oneway_yes = isinstance(oneway_val, str) and oneway_val.strip().lower() ==
       "yes"
68
69         for a, b in zip(coords, coords[1:]):
70             u = (mode, float(a[0]), float(a[1]))
71             v = (mode, float(b[0]), float(b[1]))
72
73             if (u[1], u[2]) == (v[1], v[2]):
74                 continue
75
76             w = haversine_m((u[1], u[2]), (v[1], v[2]))
77             if w <= 0:
78                 continue
79
80             # forward edge u -> v
81             if u not in dedup:
82                 dedup[u] = {}
83             cur = dedup[u].get(v)
84             if cur is None or w < cur[0]:
85                 dedup[u][v] = (w, mode)
86
87             # reverse edge v -> u unless explicitly oneway 'yes'
88             if not oneway_yes:
89                 if v not in dedup:
90                     dedup[v] = {}
91                 cur_back = dedup[v].get(u)
92                 if cur_back is None or w < cur_back[0]:
93                     dedup[v][u] = (w, mode)
94
95     # finalize adjacency list
96     graph = {u: [(v, lw[0], lw[1]) for v, lw in nbrs.items()] for u, nbrs in
       dedup.items()}
97     return graph
98
99 def merge_graphs(graph_a, graph_b):
```

```
100     merged = defaultdict(dict)
101     def add_from(g):
102         for u, edges in g.items():
103             nbrs = merged[u]
104             for v, w, m in edges:
105                 best = nbrs.get(v)
106                 if best is None or w < best[0]:
107                     nbrs[v] = (w, m)
108     add_from(graph_a)
109     add_from(graph_b)
110     return {u: [(v, w_m[0], w_m[1]) for v, w_m in nbrs.items()] for u, nbrs in
        merged.items()}
111
112 def find_nearest_node(nodes_iterable, point_lonlat):
113     best_node = None
114     best_km = float('inf')
115     for n in nodes_iterable:
116         d = haversine_km(point_lonlat, node_lonlat(n))
117         if d < best_km:
118             best_km = d
119             best_node = n
120     if best_node is None:
121         raise ValueError("No nodes provided.")
122     return best_node, best_km
123
124 def node_set_from_graph(graph):
125     s = set(graph.keys())
126     for _, lst in graph.items():
127         for v, *_ in lst:
128             s.add(v)
129     return s
130
131 # ----------------------------
132 # Connection edges
133 # ----------------------------
134 def add_connection_edge(graph, u, v, q_teus):
135     length_m = (1.85 + 0.028 * q_teus) * 80000
136     def upsert(a, b):
137         lst = graph.setdefault(a, [])
138         for i, (nb, w, m) in enumerate(lst):
139             if nb == b and m == 'connection':
140                 lst[i] = (b, float(length_m), 'connection')
141                 return
142         lst.append((b, float(length_m), 'connection'))
143     upsert(u, v)
144     upsert(v, u)
145
146 def add_poi_connections(graph, road_graph, rail_graph, pois_gpkg,
147                         q_teus,
148                         road_threshold_km=15.0, rail_threshold_km=3.0):
149     pois_path = Path(pois_gpkg)
150     if not pois_path.exists():
151         raise FileNotFoundError(f"POI GPKG not found: {pois_path}")
152     gdf = gpd.read_file(str(pois_path))
153     if gdf.crs is None:
154         gdf.set_crs(epsg=4326, inplace=True)
155     else:
156         gdf = gdf.to_crs(epsg=4326)
157     road_nodes = [n for n in node_set_from_graph(road_graph) if n[0] == "road"]
158     rail_nodes = [n for n in node_set_from_graph(rail_graph) if n[0] == "rail"]
159     added, skipped, pairs = 0, 0, []
```

```
160     for geom in gdf.geometry:
161         if geom is None or geom.is_empty or geom.geom_type != 'Point':
162             skipped += 1
163             continue
164         poi = (float(geom.x), float(geom.y))
165         try:
166             r_node, r_km = find_nearest_node(road_nodes, poi)
167             t_node, t_km = find_nearest_node(rail_nodes, poi)
168         except ValueError:
169             skipped += 1
170             continue
171         if r_km > road_threshold_km or t_km > rail_threshold_km:
172             skipped += 1
173             continue
174         add_connection_edge(graph, r_node, t_node, q_teus)
175         pairs.append((r_node, t_node))
176         added += 1
177     return added, skipped, pairs
178 
179 # ----------------------------
180 # Weight functions
181 # ----------------------------
182 def compute_time_hours(length_m: float, mode: str, q_teus: float, road_type: str)
        -> float:
183     if mode == "road":
184         km = length_m / 1000.0
185         if road_type == 'secondary' or road_type == 'secondary_link':
186             return km / KMH_SECONDARY
187         else:
188             return km / KMH_MOTORWAY
189     elif mode == "rail":
190         return (length_m/1000.0) / KMH_RAIL
191     elif mode == "connection":
192         return (SECONDS_PER_TEU_CONNECTION * q_teus) / 3600.0 +
        (MINUTES_FIXED_CONNECTION / 60.0)
193     return 0.0
194 
195 def compute_emissions_grams(length_m: float, mode: str, q_teus: float) -> float:
196     km = length_m / 1000.0
197     if mode == "road":
198         return G_PER_TKM_ROAD * TONS_PER_TEU * q_teus * km
199     elif mode == "rail":
200         return G_PER_TKM_RAIL * TONS_PER_TEU * q_teus * km
201     elif mode == "connection":
202         return G_PER_HOUR_TRANSFER * 0.028*q_teus
203     return 0.0
204 
205 def compute_cost_eur(length_m: float, mode: str, q_teus: float) -> float:
206     km = length_m / 1000.0
207     if mode == "road":
208         return ROAD_COST_PER_KM_PER_TEU * km * q_teus
209     elif mode == "rail":
210         return RAIL_COST_PER_KM * km
211     elif mode == "connection":
212         return CONNECTION_COST_PER_TEU * q_teus
213     return 0.0
214 
215 # ----------------------------
216 # Save functions
217 # ----------------------------
218 def save_edges_json(path: Path, edges):
```

```
    with open(path, "w", encoding="utf-8") as f:
        json.dump(edges, f, indent=2)

def save_edges_csv(path: Path, edges):
    headers = [
        "u_mode","u_lon","u_lat","v_mode","v_lon","v_lat",
        "length_m","mode","time_h","emissions_g","cost_eur"
    ]
    with open(path, "w", newline="", encoding="utf-8") as f:
        writer = csv.writer(f)
        writer.writerow(headers)
        for e in edges:
            u = e["u"]; v = e["v"]
            writer.writerow([
                u["mode"], float(u["lon"]), float(u["lat"]),
                v["mode"], float(v["lon"]), float(v["lat"]),
                float(e.get("length_m", 0.0)),
                e.get("mode",""),
                float(e.get("time_h", 0.0)),
                float(e.get("emissions_g", 0.0)),
                float(e.get("cost_eur", 0.0)),
            ])

def summarize_edges(edges):
    modes = {}
    for e in edges:
        mode = e.get("mode", "unknown")
        modes.setdefault(mode, []).append(e)
    print("\nEdge summary:")
    total = len(edges)
    print(f"Total: {total} edges")
    for mode, lst in modes.items():
        times = [float(x.get("time_h", 0.0)) for x in lst]
        ems   = [float(x.get("emissions_g", 0.0)) for x in lst]
        costs = [float(x.get("cost_eur", 0.0)) for x in lst]
        if times and ems and costs:
            print(f"  - {mode}: {len(lst)} edges")
            print(f"    time_h:      min={min(times):.6f}, max={max(times):.6f},
    avg={mean(times):.6f} h")
            print(f"    emissions_g: min={min(ems):.2f},  max={max(ems):.2f},
    avg={mean(ems):.2f} g")
            print(f"    cost_eur:    min={min(costs):.4f}, max={max(costs):.4f},
    avg={mean(costs):.4f} ")

    print("\nSample edges by mode:")
    for mode, lst in modes.items():
        if lst:
            e = lst[0]
            print(f"  {mode} -> length_m={float(e['length_m']):.2f}, "
                  f"time_h={float(e['time_h']):.6f},
    emissions_g={float(e['emissions_g']):.2f}, "
                  f"cost_eur={float(e['cost_eur']):.4f}")
```

F Routing algorithm code: routing_algorithm.py

F.1 Main script: routing_algorithm.py

Code F.1: Routing algorithm code (routing_algorithm.py).

```python
import json
import math
import heapq
import time
from pathlib import Path
from collections import defaultdict
from pathlib import Path

import geopandas as gpd
import matplotlib.pyplot as plt
from shapely.geometry import LineString
from matplotlib.lines import Line2D
import datetime
import pandas as pd

from functions_routing_algorithm import *

##############################
# inputs (Modifyable parameters)
##############################
BASE_DIR = Path(__file__).resolve().parent

# 1) Number of TEUs to transport in the route (Q_TEUS)
Q_TEU = 60

GRAPH_DIR = BASE_DIR / "graphs" / f'graph_q_{Q_TEU}_no_length.json'

# 2) Start/goal points

locations_dic = {
    'Berlin': (13.315287220634708, 52.584036356012966),
    'Munich': (11.573904620929909, 48.13446772388894),
    'Frankfurt am Main': (8.656278, 50.102292),
    'Stuttgart': (9.185551, 48.787472),
    'Hamburg': (10.000624, 53.447532),
    'Freiburg im Breisgau': (7.837422, 47.993896),
    'Birkenfeld':(7.175697, 49.645312),
    'Salzwedel': (11.171322, 52.839897),
    'Wolfsburg' : (10.778848, 52.416499)
}# (lon, lat)

# CHANGE HERE
start_location = 'Berlin'
goal_location    = 'Munich'
```

```
45
46 START_POINT = locations_dic[start_location]
47 GOAL_POINT  = locations_dic[goal_location]
48
49 # 3) Criteria used by Dijkstra: 'length' | 'cost' | 'time' | 'emissions'
50 WEIGHT_ATTR = 'length'
51
52 ##############################
53 # Main
54 ##############################
55 if __name__ == "__main__":
56     # 1) Load graph
57     print('Section 1:' , datetime.datetime.now(datetime.timezone.utc))
58     graph = load_graph_from_edge_list(GRAPH_DIR)
59
60     # 2) Route report
61     print('Route details:')
62     print('Origin:', start_location)
63     print('Destination:', goal_location)
64     print('Optimization criteria:', WEIGHT_ATTR)
65     print('TEUs:', Q_TEU)
66
67     # 3) Equivalent nodes for start and goal points
68     all_nodes = set(graph.keys())
69     for u, nbrs in graph.items():
70         for e in nbrs:
71             all_nodes.add(e["v"])
72
73     start_node, _ = find_nearest_node(all_nodes, START_POINT)
74     goal_node,  _ = find_nearest_node(all_nodes, GOAL_POINT)
75
76     # 4) Dijkstra's algorithm execution
77     t0 = time.perf_counter()
78     print('Start Dijkstra with weight =', WEIGHT_ATTR)
79     path, total_weight = dijkstra(graph, start_node, goal_node, WEIGHT_ATTR)
80     t1 = time.perf_counter()
81
82     # 5) Final path summary
83     path_length_m = sum_path_metric(graph, path, criteria="length")
84
85     units = {'length':'m','time':'h','emissions':'g','cost':''}[WEIGHT_ATTR]
86     print(f"Optimized total {WEIGHT_ATTR}: {total_weight:.6f} ({units})")
87     print(f"True path length (m): {path_length_m:.2f}")
88     print(f"Number of path points: {len(path)}")
89     print(f"Dijkstra runtime: {t1 - t0:.3f} s\n")
90
91     segments = summarize_path_by_mode(graph, path, criteria=WEIGHT_ATTR)
92     print_mode_summary(segments)
93
94     # 6) Path plotting and saving
95     print('Section 5:', datetime.datetime.now(datetime.timezone.utc))
96
97     # Save path into .geojson file
98     save_path_geojson(path, path_length_m, BASE_DIR / 'solutions' /
       f"o_{start_location}_d_{goal_location}_q_{Q_TEU}_a_{WEIGHT_ATTR}.geojson")
99
100    # Save information to excel
101    write_run_to_excel(
102        excel_path=BASE_DIR / "solutions" / "results.xlsx",
103        graph=graph,
104        path=path,
```

```
            start_location=start_location,
            goal_location=goal_location,
            q_teu=Q_TEU,
            weight_attr=WEIGHT_ATTR,
            dijkstra_runtime_s=(t1 - t0)
        )
        print(f"Saved: "+ str(BASE_DIR / "solutions" / "results.xlsx"))

        # Plot and save solution
        out_png = plot_route_with_border_final(
            graph=graph,
            path=path,
            start_point=START_POINT,
            goal_point=GOAL_POINT,
            start_name=start_location,
            goal_name=goal_location,
            weight_attr=WEIGHT_ATTR,
            q_teu=Q_TEU,
            base_dir=BASE_DIR,
        )
        print(f"Saved: {out_png}")
```

F.2 Auxiliar functions script: functions_routing_algorithm.py

Code F.2: Auxiliar functions for routing algorithm code (functions_routing_algorithm.py).

```
'''
Helping functions for 04_routing_algorithm.py file
'''

import json
import math
import heapq
import time
from pathlib import Path
from collections import defaultdict
from pathlib import Path

import geopandas as gpd
import matplotlib.pyplot as plt
from shapely.geometry import LineString
from matplotlib.lines import Line2D
import datetime
import pandas as pd


##############################
# Math helpers
##############################
def haversine_m(p1, p2):
    lon1, lat1 = map(math.radians, p1)
    lon2, lat2 = map(math.radians, p2)
    dlon = lon2 - lon1
    dlat = lat2 - lat1
    a = math.sin(dlat / 2) ** 2 + math.cos(lat1) * math.cos(lat2) * math.sin(dlon /
   2) ** 2
    c = 2 * math.atan2(math.sqrt(a), math.sqrt(1 - a))
    return 6371000.0 * c
```

```
32
33 def haversine_km(p1, p2):
34     return haversine_m(p1, p2) / 1000.0
35
36 ##############################
37 # Load graph
38 ##############################
39 def load_graph_from_edge_list(path):
40     '''
41     Output: graph[u] -> list of edge-dicts
42     edge-dict: {"v": v, "mode": m, "length_m": ..., "time_h": ..., "emissions_g":
       ..., "cost_eur": ...}
43                   where --> u,v = (mode, lon, lat)
44     '''
45     with open(path, "r", encoding="utf-8") as f:
46         data = json.load(f)
47
48     g = defaultdict(list)
49     for e in data:
50         u = (e["u"]["mode"], float(e["u"]["lon"]), float(e["u"]["lat"]))
51         v = (e["v"]["mode"], float(e["v"]["lon"]), float(e["v"]["lat"]))
52         edge_uv = {
53             "v": v,
54             "mode": e["mode"],
55             "length_m": float(e["length_m"]),
56             "time_h": float(e["time_h"]),
57             "emissions_g": float(e["emissions_g"]),
58             "cost_eur": float(e["cost_eur"]),
59         }
60         edge_vu = {
61             "v": u,
62             "mode": e["mode"],
63             "length_m": float(e["length_m"]),
64             "time_h": float(e["time_h"]),
65             "emissions_g": float(e["emissions_g"]),
66             "cost_eur": float(e["cost_eur"]),
67         }
68         g[u].append(edge_uv)
69         g[v].append(edge_vu)
70
71     return dict(g)
72 ##############################
73 # Weight helpers
74 ##############################
75 def edge_weight(edge, weight_attr):
76     if not weight_attr:
77         raise ValueError("edge_weight requires a non-empty weight_attr
       ('length'|'time'|'emissions'|'cost').")
78     criteria = weight_attr.lower()
79     field_by_attr = {
80         "length":    "length_m",
81         "time":      "time_h",
82         "emissions": "emissions_g",
83         "cost":      "cost_eur",
84     }
85     if criteria not in field_by_attr:
86         raise ValueError(f"Unknown WEIGHT_ATTR={criteria!r}. Use 'length' | 'time'
       | 'emissions' | 'cost'.")
87     field = field_by_attr[criteria]
88     try:
89         return float(edge[field])
```

```
90      except KeyError as e:
91          raise KeyError(f"Edge missing expected field '{field}': {edge}") from e
92  
93  ##############################
94  # Dijkstra (dict-based adjacency)
95  ##############################
96  def dijkstra(graph, start, goal, criteria):
97      queue = [(0.0, start)]
98      best  = {start: 0.0}
99      prev  = {start: None}
100 
101     while queue:
102         cur_w, u = heapq.heappop(queue)
103         if u == goal:
104             break
105         if cur_w > best.get(u, float('inf')):
106             continue
107         for edge in graph.get(u, []):
108             v = edge["v"]
109             w = edge_weight(edge, criteria)
110             new = cur_w + w
111             if new < best.get(v, float('inf')):
112                 best[v] = new
113                 prev[v] = u
114                 heapq.heappush(queue, (new, v))
115 
116     # reconstruct
117     path = []
118     cur = goal
119     while cur is not None:
120         path.append(cur)
121         cur = prev.get(cur)
122     path.reverse()
123     return path, best.get(goal, float('inf'))
124 
125 ##############################
126 # Node utilities
127 ##############################
128 def node_lonlat(n):
129     # n = (mode, lon, lat)
130     return (n[1], n[2])
131 
132 def find_nearest_node(nodes_iterable, point):
133     best_node = None
134     best_km = float('inf')
135     for n in nodes_iterable:
136         d = haversine_km(point, node_lonlat(n))
137         if d < best_km:
138             best_km = d
139             best_node = n
140     if best_node is None:
141         raise ValueError("No nodes provided.")
142     return best_node, best_km
143 
144 def node_set_from_graph(graph):
145     s = set(graph.keys())
146     for _, lst in graph.items():
147         for e in lst:
148             s.add(e["v"])
149     return s
150 
```

```python
##############################
# Path saver and info saver
##############################
def save_path_geojson(path, distance_m, out_geojson: Path):
    import json
    from pathlib import Path
    out_geojson = Path(out_geojson)
    out_geojson.parent.mkdir(parents=True, exist_ok=True)

    # Build coordinates safely from path nodes
    coords = []
    for node in path:
        # node can be (mode, lon, lat) or (lon, lat)
        if len(node) == 3:
            _, lon, lat = node
        else:
            raise ValueError(f"Unexpected node format: {node}")
        coords.append([float(lon), float(lat)])

    fc = {
        "type": "FeatureCollection",
        "features": [{
            "type": "Feature",
            "properties": {
                "distance_m": float(distance_m),
                "n_points": len(coords)
            },
            "geometry": {
                "type": "LineString",
                "coordinates": coords
            }
        }]
    }

    with open(out_geojson, "w", encoding="utf-8") as f:
        json.dump(fc, f, ensure_ascii=False, indent=2)

def write_run_to_excel(
    excel_path,
    graph,
    path,
    start_location,
    goal_location,
    q_teu,
    weight_attr,
    dijkstra_runtime_s):

    def edge_weight_inline(edge, criteria):
        field_by_attr = {
            "length": "length_m",
            "time": "time_h",
            "emissions": "emissions_g",
            "cost": "cost_eur",
        }
        key = field_by_attr[criteria.lower()]
        return float(edge[key])

    def edge_lookup_inline(u, v):
        for e in graph.get(u, []):
            if e["v"] == v:
                return e
```

```
212         for e in graph.get(v, []):  # fallback
213             if e["v"] == u:
214                 return e
215         raise KeyError(f"Edge not found between {u} and {v}")
216 
217     def sum_over_path(criteria, mode_filter=None, unit_divisor=1.0):
218         total = 0.0
219         if not path or len(path) < 2:
220             return 0.0
221         for u, v in zip(path, path[1:]):
222             e = edge_lookup_inline(u, v)
223             if (mode_filter is None) or (e.get("mode") == mode_filter):
224                 total += edge_weight_inline(e, criteria)
225         return total / unit_divisor
226 
227     def summarize_sequence(criteria):
228         if not path or len(path) < 2:
229             return ""
230         field_unit = {
231             "length": ("length_m", "km", 1000.0),
232             "time": ("time_h", "h", 1.0),
233             "emissions": ("emissions_g", "g", 1.0),
234             "cost": ("cost_eur", "", 1.0),
235         }[criteria.lower()]
236         fld, unit_label, divisor = field_unit
237 
238         segs, cur_mode, acc = [], None, 0.0
239         for u, v in zip(path, path[1:]):
240             e = edge_lookup_inline(u, v)
241             m = e["mode"]
242             val = float(e[fld]) / divisor
243             if m == cur_mode:
244                 acc += val
245             else:
246                 if cur_mode is not None:
247                     segs.append((cur_mode, acc))
248                 cur_mode, acc = m, val
249         if cur_mode is not None:
250             segs.append((cur_mode, acc))
251 
252         def deccomma(x):
253             s = f"{x:.3f}".replace(".", ",")
254             while s.endswith("0"):
255                 s = s[:-1]
256             if s.endswith(","):
257                 s = s[:-1]
258             return s
259 
260         return " -> ".join([f"{m}({deccomma(val)} {unit_label})" for m, val in
    segs])
261 
262     def has_mode(mode_name):
263         for u, v in zip(path, path[1:]):
264             e = edge_lookup_inline(u, v)
265             if e.get("mode") == mode_name:
266                 return True
267         return False
268 
269     def next_test_number(df):
270         if df.empty:
271             return 1
```

```
272         try:
273             return int(pd.to_numeric(df["TEST"], errors="coerce").max()) + 1
274         except Exception:
275             return len(df) + 1
276 
277     def deccomma(x, nd=3):
278         s = f"{x:.{nd}f}".replace(".", ",")
279         while s.endswith("0"):
280             s = s[:-1]
281         if s.endswith(","):
282             s = s[:-1]
283         return s
284 
285     # weights
286     crit = weight_attr.lower()
287     path_len_km = sum_over_path("length", unit_divisor=1000.0)
288     road_km = sum_over_path("length", mode_filter="road", unit_divisor=1000.0)
289     rail_km = sum_over_path("length", mode_filter="rail", unit_divisor=1000.0)
290     conn_km = sum_over_path("length", mode_filter="connection", unit_divisor=1000.0)
291 
292     attr_total = sum_over_path(crit, unit_divisor=(1000.0 if crit == "length" else
    1.0))
293     attr_road  = sum_over_path(crit, mode_filter="road", unit_divisor=(1000.0 if
    crit == "length" else 1.0))
294     attr_rail  = sum_over_path(crit, mode_filter="rail", unit_divisor=(1000.0 if
    crit == "length" else 1.0))
295     attr_conn  = sum_over_path(crit, mode_filter="connection", unit_divisor=(1000.0
    if crit == "length" else 1.0))
296 
297     seq_str = summarize_sequence(crit)
298     multimodal = "YES" if (has_mode("road") and has_mode("rail")) else "NO"
299 
300     excel_path = Path(excel_path)
301     excel_path.parent.mkdir(parents=True, exist_ok=True)
302     columns = [
303         "TEST","ORIGIN","DESTINATION","Q_TEU","ATTRIBUTE",
304         "DIJKSTRA TIME (S)","PATH LENGTH","MULTIMODAL","ROAD KM","TRAIN
    KM","CONNECTION","SEQUENCE",
305         "attribute_total","attribute_road","attribute_rail","attribute_connection"
306     ]
307     if excel_path.exists():
308         try:
309             df = pd.read_excel(excel_path)
310         except Exception:
311             df = pd.DataFrame(columns=columns)
312     else:
313         df = pd.DataFrame(columns=columns)
314 
315     for col in columns:
316         if col not in df.columns:
317             df[col] = pd.Series(dtype="object")
318 
319     test_id = next_test_number(df)
320 
321     new_row = {
322         "TEST": test_id,
323         "ORIGIN": start_location,
324         "DESTINATION": goal_location,
325         "Q_TEU": q_teu,
326         "ATTRIBUTE": crit,
327         "DIJKSTRA TIME (S)": deccomma(float(dijkstra_runtime_s)),
```

```
328            "PATH LENGTH": deccomma(float(path_len_km)),
329            "MULTIMODAL": multimodal,
330            "ROAD KM": deccomma(float(road_km)) if road_km > 0 else "",
331            "TRAIN KM": deccomma(float(rail_km)) if rail_km > 0 else "",
332            "CONNECTION": deccomma(float(conn_km)) if conn_km > 0 else "",
333            "SEQUENCE": seq_str,
334            "attribute_total": deccomma(float(attr_total)),
335            "attribute_road": deccomma(float(attr_road)) if attr_road > 0 else "",
336            "attribute_rail": deccomma(float(attr_rail)) if attr_rail > 0 else "",
337            "attribute_connection": deccomma(float(attr_conn)) if attr_conn > 0 else "",
338        }
339
340        df = pd.concat([df, pd.DataFrame([new_row])], ignore_index=True)
341        df.to_excel(excel_path, index=False)
342
343        print(f"Updated Excel: {excel_path}")
344
345    ##############################
346    # Plot helpers
347    ##############################
348    def fetch_germany_border_gdf(base_dir:Path=None):
349        if base_dir is None:
350            base_dir = Path(__file__).resolve().parent
351        border_file = Path(base_dir) / "input_files" / "germany_border.geojson"
352        if not border_file.exists():
353            raise FileNotFoundError(f"Border file not found: {border_file}")
354        return gpd.read_file(border_file)
355
356    def _edge_mode_dict(graph, u, v):
357        for e in graph.get(u, []):
358            if e["v"] == v:
359                return e["mode"]
360        for e in graph.get(v, []):
361            if e["v"] == u:
362                return e["mode"]
363        return "unknown"
364
365    def _is_rail_node_dict(graph, node):
366        return any(e["mode"] == "rail" for e in graph.get(node, []))
367
368    def rail_node_for_connection_edge(graph, u, v):
369        def rail_degree(n):
370            return sum(1 for e in graph.get(n, []) if e.get("mode")=="rail")
371        ru, rv = rail_degree(u), rail_degree(v)
372        if ru > rv: return u
373        if rv > ru: return v
374        return v
375
376    def plot_route_with_border_final(
377        graph,
378        path,
379        start_point,
380        goal_point,
381        start_name: str,
382        goal_name: str,
383        weight_attr: str,
384        q_teu: int,
385        base_dir: Path,
386        fetch_border_fn=fetch_germany_border_gdf):
387
388        plt.rcParams["font.family"] = "Times New Roman"
```

```
389
390     gdf_border = fetch_border_fn(base_dir)
391     title_weight = weight_attr.capitalize()
392     title = f"{title_weight} optimized route from {start_name} to {goal_name}
        (TEUs: {q_teu})"
393
394     out_dir = base_dir / "route_figures"
395     out_dir.mkdir(parents=True, exist_ok=True)
396     out_png = out_dir /
        f"o_{start_name}_d_{goal_name}_q_{q_teu}_a_{weight_attr}.png"
397
398     fig, ax = plt.subplots(figsize=(10, 10))
399     ax.set_aspect("equal", adjustable="box")
400     ax.axis("off")
401     ax.grid(False)
402     fig.patch.set_facecolor("white")
403
404     # Border
405     gdf_border.plot(ax=ax, facecolor="none", edgecolor="black", linewidth=1.0,
        zorder=1)
406
407     road_color = "green"
408     rail_color = "red"
409     conn_color = "yellow"
410     rail_conn_nodes = []
411
412     # Draw path edges by mode
413     if path and len(path) >= 2:
414         for u, v in zip(path, path[1:]):
415             mode = _edge_mode_dict(graph, u, v)
416             xs = [u[1], v[1]]
417             ys = [u[2], v[2]]
418
419             if mode == "road":
420                 ax.plot(xs, ys, color=road_color, linewidth=2.0, zorder=3)
421             elif mode == "rail":
422                 ax.plot(xs, ys, color=rail_color, linewidth=2.0, zorder=3)
423             elif mode == "connection":
424                 ax.plot(xs, ys, color=conn_color, linewidth=2.0, zorder=3)
425                 rail_node = rail_node_for_connection_edge(graph, u, v)
426                 rail_conn_nodes.append(rail_node)
427             else:
428                 ax.plot(xs, ys, color="gray", linewidth=1.5, linestyle=":",
        zorder=2)
429
430         # Start/Goal points
431         ax.scatter([start_point[0]], [start_point[1]], s=60, c="black", zorder=4)
432         ax.scatter([goal_point[0]],  [goal_point[1]],  s=60, c="black", zorder=4)
433         ax.text(start_point[0], start_point[1], f"  {start_name}", va="center",
        ha="left", fontsize=10)
434         ax.text(goal_point[0],  goal_point[1],  f"  {goal_name}",  va="center",
        ha="left", fontsize=10)
435
436     if rail_conn_nodes:
437         rx = [n[1] for n in rail_conn_nodes]
438         ry = [n[2] for n in rail_conn_nodes]
439         ax.scatter(rx, ry, s=45, c="blue", edgecolor="white", linewidth=0.6,
        zorder=5)
440
441     ax.set_title(title)
442     handles = [
```

```
443         Line2D([0], [0], color=rail_color, lw=2, label="Rail"),
444         Line2D([0], [0], color=road_color, lw=2, label="Road"),
445         Line2D([0], [0], color=conn_color, lw=2, label="Connection"),
446         Line2D([0], [0], marker="o", color="w", markerfacecolor="blue",
447                markeredgecolor="white", markersize=8, label="Rail node at
    connection"),
448         Line2D([0], [0], marker="o", color="w", markerfacecolor="black",
449                markeredgecolor="black", markersize=8, label="Start / Goal"),
450     ]
451     ax.legend(handles=handles, loc="upper right", frameon=True)
452     plt.rcParams["legend.fontsize"] = 14
453 
454     fig.savefig(out_png, dpi=300, bbox_inches="tight")
455     plt.close(fig)
456     return out_png
457 
458 # ----------------------------
459 # Sum helpers and summaries
460 # ----------------------------
461 def _edge_info(graph, u, v):
462     for e in graph.get(u, []):
463         if e["v"] == v:
464             return e
465     raise KeyError(f"Edge not found: {u} -> {v}")
466 
467 def summarize_path_by_mode(graph, path, criteria="length"):
468     if not path or len(path) < 2:
469         return []
470     field_by = {
471         "length":    "length_m",
472         "time":      "time_h",
473         "emissions": "emissions_g",
474         "cost":      "cost_eur",
475     }
476     if criteria not in field_by:
477         raise ValueError("criteria must be one of: length | time | emissions |
    cost")
478     field = field_by[criteria]
479 
480     segments = []
481     current_mode = None
482     acc = 0.0
483 
484     for u, v in zip(path, path[1:]):
485         e = _edge_info(graph, u, v)
486         m = e["mode"]
487         w = float(e[field])
488         if m == current_mode:
489             acc += w
490         else:
491             if current_mode is not None:
492                 segments.append((current_mode, acc))
493             current_mode = m
494             acc = w
495     if current_mode is not None:
496         segments.append((current_mode, acc))
497     return segments
498 
499 def print_mode_summary(segments):
500     print("Path segments (per selected metric):")
501     for m, val in segments:
```

```
502         print(f"  {m}: {val:.3f}")
503 
504 def sum_path_metric(graph, path, criteria):
505     if not path or len(path) < 2:
506         return 0.0
507     total = 0.0
508     for u, v in zip(path, path[1:]):
509         edge = None
510         for e in graph.get(u, []):
511             if e["v"] == v:
512                 edge = e
513                 break
514         if edge is None:
515             raise KeyError(f"Edge not found: {u} -> {v}")
516         total += float(edge_weight(edge, criteria))
517     return total
```